Acceleration of Block-Aware Matrix Factorization on Heterogeneous Platforms

Gregory W. Somers

Thesis submitted to the
Faculty of Graduate and Postdoctoral Studies
in partial fulllment of the requirements for the
Master of Applied Science degree
in Electrical and Computer Engineering

School of Electrical Engineering and Computer Science
University of Ottawa
Ottawa, Canada

© Gregory W. Somers, Ottawa, Canada, 2016

Abstract

Block-structured matrices arise in several contexts in circuit simulation problems. These matrices typically inherit the pattern of sparsity from the circuit connectivity. However, they are also characterized by dense spots or blocks. Direct factorization of those matrices has emerged as an attractive approach if the host memory is sufficiently large to store the block-structured matrix. The approach proposed in this thesis aims to accelerate the direct factorization of general block-structured matrices by leveraging the power of multiple OpenCL accelerators such as Graphical Processing Units (GPUs).

The proposed approach utilizes the notion of a Directed Acyclic Graph representing the matrix in order to schedule its factorization on multiple accelerators. This thesis also describes memory management techniques that enable handling large matrices while minimizing the amount of memory transfer over the PCIe bus between the host CPU and the attached devices. The results demonstrate that by using two GPUs the proposed approach can achieve a nearly optimal speedup when compared to a single GPU platform.

Dedication

This thesis is dedicated in memory of Richard Roy, Yvonne Roy, Gerald Somers and Bonnie Somers.

Acknowledgements

I would like to express my sincere gratitude to my supervisors Dr. Emad Gad and Dr. Miodrag Bolic, the University of Ottawa, and the Natural Sciences and Engineering Research Council of Canada (NSERC) for their support, which made this thesis possible.

Contents

1 Introduction **1**

1.1 The Problem . . . 1

1.2 The Motivation . . . 2

1.3 The Contributions . . . 3

1.4 Thesis Organization . . . 3

2 Background and Related Work **5**

2.1 Introduction to Circuit Simulation . . . 5

2.1.1 Modified Nodal Analysis (MNA) . . . 5

2.2 Steady-State Harmonic Balance Analysis . . . 9

2.2.1 Single Tone Excitation . . . 9

2.2.2 Discrete Fourier Transformation . . . 10

2.2.3 Harmonic Balance for Linear Circuits . . . 11

2.2.4 Harmonic Balance for Non-Linear Circuits . . . 12

2.2.5 Newton-Raphson Iterative Method for Nonlinear Systems . . . 16

2.2.6 Solving HB Using Newton-Raphson . . . 17

2.3 Uncertainty Quantification via Polynomial Chaos . . . 19

2.4 Structure of the Jacobian Matrix . . . 20

2.5 Introduction to General Purpose GPU . . . 21

2.5.1 Introduction . . . 21

2.5.2 AMD Graphics Core Next Architecture . . . 21

2.6 Introduction to OpenCL . . . 22

2.6.1 OpenCL Overview . . . 22

2.6.2 Platform Model . . . 23

2.6.3 Execution Model 24
2.6.4 Memory Model 25
2.6.5 Programming Model 25
2.6.6 Synchronization in OpenCL 26
2.7 Mapping OpenCL to GCN Architecture 28
2.8 Related Work and Existing Approaches 29
2.9 Summary 31

3 Parallelism in Matrix Structure **32**
3.1 Introduction 32
3.2 Classical KLU Factorization 32
3.3 Block-Aware LU Factorization 34
3.4 Identifying Column Level Dependencies 35
3.5 Scheduling Parallel Factorization 36
3.6 Summary 39

4 Block KLU with Multiple OpenCL Accelerators **40**
4.1 Introduction 40
4.2 Device Context Separation 41
4.3 Memory Layout 41
4.3.1 Block Compressed Column Storage 41
4.3.2 Device Memory 42
4.3.3 Host Memory 43
4.4 Description of Execution 44
4.5 General Improvement Strategies 46
4.5.1 Introduction 46
4.5.2 Asynchronous Communication 47
4.5.3 Synchronization 48
4.5.4 Task Splitting 48
4.5.5 Order of Operations 49
4.6 Summary 50

5 Memory Optimizations **51**
5.1 Introduction 51

5.2 Device Memory Definition . . . 53
5.3 Memory Optimizations for the $\mathcal{J}$ Matrix . . . 53
5.3.1 Column Transfer Mode . . . 54
5.3.2 Block Transfer Mode . . . 54
5.3.3 Performance Impact . . . 54
5.4 Zero Column Transfer Mode . . . 55
5.4.1 Zero Column Transfer Description . . . 56
5.5 Optimal Replacement Strategy . . . 59
5.5.1 Optimal Replacement (OPT) . . . 61
5.5.2 The Forward Distance Optimization . . . 62
5.5.3 Optimal Column Replacement . . . 63
5.5.4 Optimal Block Replacement . . . 65
5.6 Memory Optimization for the $\boldsymbol{\mathcal{U}}$ Factor . . . 67
5.7 Summary . . . 67

6 Experimental Results 69
6.1 System Setup . . . 69
6.2 Benchmark Matrices . . . 69
6.2.1 Kronecker Product . . . 70
6.2.2 Overview of Benchmark Matrices . . . 70
6.3 Experimental Results . . . 73
6.4 Execution Profile . . . 76
6.4.1 Profiling Device Kernels . . . 76
6.5 Summary . . . 77

7 Profiling of Benchmark Matrices 78
7.1 Introduction . . . 78
7.2 Factorization of the Harmle1 Matrix . . . 78
7.3 Factorization of the Rajat11 Matrix . . . 85
7.4 Zero Column Transfer on Rajat11 . . . 89
7.5 Optimal Column Replacement on Rajat11 . . . 91
7.6 Workspace Swapping on Hamrle1 . . . 93
7.7 Performance Impact of Optimal Column Replacement . . . 93
7.8 Summary . . . 94

8 Challenges of Small Block Factorization **97**
8.1 Introduction 97
8.2 Naïve Gaussian Elimination 98
8.3 Performance of Naïve Gaussian Elimination on GPU 98
8.4 Summary 101

9 Conclusion **103**
9.1 Future Work 104

Bibliography **105**

List of Tables

2.1 Basic circuit element stamps 8
2.2 Comparison of previous work 31

5.1 Comparison of device memory usage for the $\mathcal{J}$ transfer modes 55
5.2 $\mathcal{L}$ Buffer Usage Without Dependence Reduction on Dual Accelerator System 57
5.3 Lasted used values for each column and device 60
5.4 Reducing $\mathcal{L}$ By Dependence on Dual Accelerator System 60
5.5 Reduced $\mathcal{L}$ by dependency analysis for a single accelerator applied to various matrices from the University of Florida Sparse Matrix Collection [1] * In megabytes, assuming a block size of 320×320 60
5.6 Optimal Replacement Algorithm (OPT) Example 62
5.7 Computation time spent finding forward distances for various counts of total blocks and block accesses 62
5.8 Memory Optimization Matrix 67

6.1 Test system specifications 69
6.2 Simulation Results 74

7.1 Hamrle1 Matrix Details 80
7.2 Parallel Schedule for the Hamrle1 Matrix 82
7.3 Hamrle1 matrix factorization details for GPU 82
7.4 Overview of Hamrle1 synchronization events 83
7.5 Rajat11 Matrix Details 85
7.6 Parallel Schedule for the Rajat11 Matrix 88
7.7 Rajat11 matrix factorization details 88
7.8 Overview of Hamrle1 synchronization events 89

7.9 Rajat11 matrix factorization details 90
7.10 OCR profiling for three benchmark matrices 90
7.11 Rajat11 matrix factorization details scenario one 91
7.12 Rajat11 matrix factorization details for scenario two 92
7.13 OCR profiling for three benchmark matrices 92
7.14 Hamrle1 matrix factorization with workspace swapping 94
7.15 Workspace Swapping profiling for two benchmark matrices 94

8.1 Test system math libraries 99
8.2 Performance comparison of matrix operations a 32×32 and 64×64 dense matrix using test system 100

List of Figures

2.1 Sample Circuit . . . 6

2.2 Ordering of $\bar{\boldsymbol{X}}^{(j)}$ and $\bar{\boldsymbol{X}}^{(j)}_{node}$ . . . 18

2.3 Simplified Compute Unit Architecture . . . 22

2.4 OpenCL Platform Model . . . 23

2.5 OpenCL Execution Model . . . 24

2.6 OpenCL Memory Model . . . 26

2.7 OpenCL Device Mapping . . . 29

3.1 Gilbert-Peierls Column Level LU factorization . . . 35

3.2 Sparsity pattern of example matrices . . . 36

3.3 Dependency graph based on $\boldsymbol{U}$ factor . . . 37

3.4 Using the DAG to extract the parallelism . . . 38

4.1 Example of Block Compressed Column Storage format . . . 42

4.2 OpenCL device memory buffers for example Device 1 . . . 43

5.1 Device memory usage of various benchmark matrices from [1] without optimization . . . 52

5.2 Performance of $\mathcal{J}$ transfer modes . . . 55

5.3 Buffer requirements of $\mathcal{J}$ transfer modes for various matrices from [1] (in Megabytes). . 56

5.4 Dependency graph and parallel schedule . . . 57

5.5 Memory Techniques Applied . . . 68

6.1 Speedup for benchmark matrices, BKLU with two GPUs versus BKLU-CPU with six threads . . . 75

6.2 Speedup for benchmark matrices, BKLUX with one GPU versus BKLUX with two GPUs 75

6.3 Single GPU kernel execution profile for rajat05 matrix with a block size of 320×320 . . . 76

7.1 Hamrle1 Sparsity Pattern 79
7.2 Hamrle1 Dependency Graph 81
7.3 Hamrle1 Execution Profile Scenario One 84
7.4 Rajat11 Sparsity Pattern 86
7.5 Rajat11 Dependency Graph 87
7.6 Summary of ZCT performance results 90
7.7 Summary of OCR performance results 93
7.8 Summary of Workspace swapping performance results 95
7.9 Optimal Column Replacement PCI transfers and speedup 95
7.10 Random Column Replacement PCI transfers 96

8.1 Naive Gaussian Elimination Example 99
8.2 Execution time and speedup for LU decomposition on the CPU and GPU 100
8.3 Number of wavefronts available during Naive Gaussian Elimination on a 32×32 block . . 101
8.4 Number of threads available during Naive Gaussian Elimination on a 192×192 block . . 102
8.5 GPU pivot factorization time for various block sizes 102

List of Symbols and Abbreviations

CAD	Computer Aided Design
GPU	Graphics Processing Unit
AMD	Advanced Micro Devices Inc.
NR	Newton-Raphson
CPU	Central processing unit
FLOPS	Floating-Point Operations per Second
OpenCL	Open Compute Language
PCI	Peripheral Component Interconnect
MNA	Modified Nodal Analysis
N	Number of equations in the MNA equation
$\boldsymbol{j}$	A vector
$\boldsymbol{J}$	A matrix
$\boldsymbol{\mathcal{J}}$	A block structured matrix
HB	Harmonic Balance
IDFT	Inverse Discrete Fourier Transformation
DFT	Discrete Fourier Transformation
3-D	Three dimensional
CUDA	Compute Unified Device Architecture
GCN	Graphics Core Next
RISC	Reduced Instruction Set Computing
VLIW	Very Long Instruction Word
CU	Compute Unit
SIMD	Single Instruction Multiple Data
ALU	Arithmetic logic unit
APU	Accelerated Processing Unit
FPGA	Field-Programmable Gate Array
C99	C programming language standard
PE	Processing Element
G-P	Gilbert-Peierls
FS	Forward Solve
BKLU	Block-KLU

DAG	Directed Acyclic Graph
BKLUX	Block-KLU-X (this work)
BCCS	Block Compressed Column Storage
C_f	Column Finished
C_p	Column Present
GEMM	General Matrix Multiplication
BLAS	Basic Linear Algebra Subprograms
TRSM	TRiangular Solve Matrix
API	Application Programming Interface
ZCT	Zero Column Transfer
Tx	Transfer
OPT	Optimal Replacement Algorithm
OCR	Optimal Column Replacement
FD	Forward Distance
OBR	Optimal Block Replacement
DRAM	Dynamic random-access memory
MAC	Multiply-Accumulate-Subtract
LAPACK	Linear Algebra Package
MPI	Message Passing Interface

Chapter 1

Introduction

1.1 The Problem

Circuit simulators are an integral part of most modern Computer-Aided Design (CAD) tools. Simulators allow circuit designers to mathematically model a circuit's behavior and verify their designs before physically building them. The main computational task at the core of a circuit simulation or analysis problem is typically a very large and sparse matrix which needs to be factorized into its LU components. The size and structure of this matrix depends on the simulation task. For example, whether it is time-domain transient simulation or nonlinear frequency-domain analysis using a technique such as Harmonic Balance [2], [3].

The Harmonic Balance approach is used to calculate the steady-state response of non-linear circuits to a quasi-periodic stimulus. The circuit waveforms are represented by Fourier series where the steady-state response is obtained by solving a system of nonlinear differential equations in the frequency domain. In order to solve this nonlinear system, an iterative method such as the Newton-Raphson (NR) method, which successively computes better approximations to the system's solution, must be used. At each iteration an adjustment matrix, known as the Jacobian Matrix, must be factorized into its LU components to compute the approximation which will be used in the next iteration.

In many circuit analysis situations, including Harmonic Balance, the Jacobian matrix that arises shows a peculiar pattern which inherits the sparse structure from the interconnectivity of the circuit nodes,

while also exhibiting dense spots or blocks. Some approaches successfully avoid the direct factorization of these types of matrices by resorting to the Krylov-subspace-based method [4] which requires only multiplying the matrix by a vector. On the other hand, direct factorization of the Jacobian matrix becomes an attractive avenue if the target system's main memory is of sufficient size to store the block structured matrix in its entirety [5]. Still, the computational power required to factorize these types of matrices in a timely fashion remains an issue for solving problems via the CPU.

Modern Graphical Processing Unit (GPU) platforms with their massively parallel architectures provide a distinct computational edge to the direct factorization of block structured matrices. The key advantage offered by GPU architectures, compared to modern CPUs, is the sheer number of floating-point operations (flops) which can be performed in a single second, typically peaking at a few thousand gigaflops (billion floating-point operations per second).

Recently, an OpenCL-based approach was presented in [6]. This approach utilizes a platform with a single GPU in the direct factorization of such block structured matrices. It is based on a block form of the KLU factorization algorithm [7] and achieves significant speedup over CPU based approaches. However, even though [6] performs well against the CPU, it is unable to scale to platforms with multiple GPUs. Additionally, [6] assumes that the GPU's main memory is sufficiently large to hold the entire problem, placing a limit on the size of problems which can be tackled.

1.2 The Motivation

Since the previous work related to the factorization of the Harmonic Balance Jacobian Matrix and other block structured matrices have not leveraged the power of multiple GPUs, a scalable multi-GPU approach to the factorization of block structured matrices is a problem worth addressing. Additionally, there are several other questions inherent to parallel processing which present significant hurdles to taking advantage of multiple GPUs, namely

1. How to identify independent factorization tasks and partition the tasks between multiple OpenCL accelerators?

2. The overhead introduced by synchronization has a significant impact on the performance of a parallel algorithm. As such, how can synchronization be accomplished with minimal impact on performance?

3. A heterogeneous system uses more than one kind of specialized devices such as GPUs, CPUs and FPGAs. These are connected over a communication medium, typically a PCI(e) bus, where data transfer rates between the devices and the main system memory is slow. Thus, how can the number of memory transfers sent over the PCI bus be minimized?

4. The massively parallel architectures of OpenCL devices rely on high levels of data parallelism to achieve optimal performance. That begs the question of how to determine what block size is sufficiently large for the problem to be solved by a parallel architecture?

1.3 The Contributions

This thesis demonstrates an approach for solving the problem of block-aware LU factorization on heterogeneous platforms with multiple accelerators. It also demonstrates methods for alleviating the issues associated with the memory belonging to discrete accelerators. Through careful memory handling, it is shown to optimally reduce the number of PCI bus transfers and exploit the structure of matrices to significantly reduce the amount of on-chip memory resources required to factorize matrices. This is all shown by demonstrating the process of block-aware LU factorization on several matrices using two GPUs. Ultimately, this thesis presents an approach for handling the parallel execution and managing the memory resources of block LU factorization.

1.4 Thesis Organization

This thesis is organized as follows. **Chapter 2** details two circuit simulation problems where block structured matrices arise and the necessary background on GPU architectures, the OpenCL programming paradigm and how it maps to the GPU architecture. **Chapter 3** introduces the block aware KLU algorithm and how column independence is determined from the structure of matrices and the scheduling of parallel execution. **Chapter 4** discusses the implementation of the block KLU algorithm using multiple OpenCL devices. **Chapter 5** introduces several techniques used for reducing the memory requirements of the OpenCL accelerator and optimizing the number of bus transfers required between devices and the host. **Chapter 6** shows the experimental results of the work presented in this thesis. **Chapter 7** provides a detailed profile and performance analysis of the multi-device execution on various sample matrices. **Chapter 8** offers insights into the weaknesses of GPUs and massively parallel accelerators

when small block sizes are used in the matrix factorization. Finally **Chapter 9** provides the concluding remarks.

Chapter 2

Background and Related Work

This chapter presents a brief overview of the background concepts that form the foundation of this thesis. Section 2.1 to Section 2.3 briefly discusses the details of two areas where block structured matrices arise - simulation of nonlinear circuits as well as uncertainty quantification via polynomial chaos. Section 2.4 covers the structure of the resulting block matrix. Finally, Section 2.5 introduces the general purpose GPU and the programming paradigm used to accelerate general purpose application on accelerators such as the GPU.

2.1 Introduction to Circuit Simulation

Circuit simulation is the process of mathematically modeling the behavior of electronic circuits. This allows designers to test and analyze their designs with various inputs and differing conditions before physically constructing the circuits. The starting point of circuit simulation is usually the representation of general circuits in the mathematical domain using the Modified Nodal Analysis (MNA) approach [8]. The next section describes Modified Nodal Analysis in detail.

2.1.1 Modified Nodal Analysis (MNA)

Modified Nodal Analysis is performed by writing Kirchhoff Current Law (KCL) at each node in the circuit to obtain N equations where N is the total number of nodes in the circuit. The equations

obtained from KCL are arranged into a system of differential-algebraic equations (DAE) of the form

$$\boldsymbol{G}\boldsymbol{x}(t) + \boldsymbol{C}\frac{\mathrm{d}\boldsymbol{x}(t)}{\mathrm{d}t} + \boldsymbol{f}(\boldsymbol{x}(t)) = \boldsymbol{b}(t) \tag{2.1}$$

where

- $\boldsymbol{x}(t) \in \mathbb{R}^N$ is a vector of waveforms of node voltages
- $\boldsymbol{G} \in \mathbb{R}^{N\times N}$ is the matrix of memory-less elements (e.g. resistors)
- $\boldsymbol{C} \in \mathbb{R}^{N\times N}$ is the matrix of memory elements (e.g. capacitors and inductors)
- $\boldsymbol{f} \in \mathbb{R}^N$ is the vector of non-linear elements (e.g. transistors and diodes)
- $\boldsymbol{b}(t) \in \mathbb{R}^N$ is the vector of independent sources (e.g. voltage and current sources)
- N is the number of unknown waveforms

Each circuit element contributes to the vectors and matrices of the MNA equation (Equation (2.1)). This contribution can be captured by the element's stamp. The stamps of some basic circuit elements are shown in Table 2.1. The circuit shown in Figure 2.1 and the accompanying MNA formulation in Equation (2.5) show an example of the application of MNA stamps.

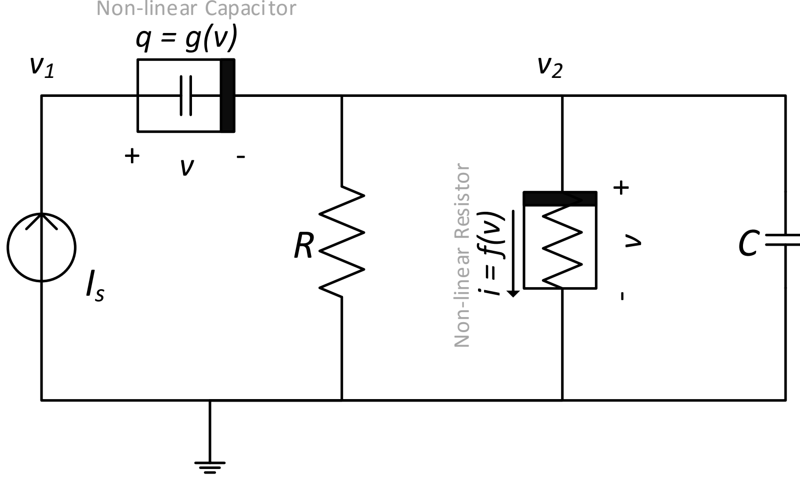


Figure 2.1: Sample Circuit

Performing KCL at nodes v_1 and v_2, the following equations are obtained

$$0 = I_S + \frac{dq}{dt} \tag{2.2}$$

$$0 = \frac{1}{R}v_2 + f(v_2) + C\frac{\partial v_2}{dt} - \frac{dq}{dt} \tag{2.3}$$

$$0 = q - g(v_1 - v_2) \tag{2.4}$$

Arranging the above in matrix form, the following system of equations

$$\begin{bmatrix} 0 & 0 & 0 \\ 0 & 1/R & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} v_1 \\ v_2 \\ q \end{bmatrix} + \begin{bmatrix} 0 & 0 & 1 \\ 0 & C & -1 \\ 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} \frac{\partial v_1}{dt} \\ \frac{\partial v_2}{dt} \\ \frac{\partial q}{dt} \end{bmatrix} + \begin{bmatrix} 0 \\ f(v_2) \\ -g(v_1 - v_2) \end{bmatrix} = \begin{bmatrix} I_s \\ 0 \\ 0 \end{bmatrix} \tag{2.5}$$

One can now see how the stamps in Table 2.1 are derived, for example for the resistor between nodes v_2 and ground, $\frac{1}{R}$ is placed in the $\boldsymbol{G}(x)$ matrix at (2,2).

Element	Symbol	Stamp
Resistor	R, i, j	$\begin{array}{c} \\ i \\ \\ j \end{array} \begin{array}{c} \begin{array}{ccc} i & & j \end{array} \\ \begin{bmatrix} 1/R & \cdots & -1/R \\ \vdots & \ddots & \vdots \\ -1/R & \cdots & 1/R \end{bmatrix} \end{array}$ G matrix
Capacitor	C, i, j	$\begin{array}{c} \\ i \\ \\ j \end{array} \begin{array}{c} \begin{array}{ccc} i & & j \end{array} \\ \begin{bmatrix} C & \cdots & -C \\ \vdots & \ddots & \vdots \\ -C & \cdots & C \end{bmatrix} \end{array}$ C matrix
Inductor	I_L, L, i, j	$\begin{array}{c} \\ i \\ \\ j \\ \text{New row} \end{array} \begin{array}{c} \begin{array}{cccc} i & & j & \text{New col} \end{array} \\ \begin{bmatrix} 0 & \cdots & 0 & 1 \\ \vdots & \ddots & \vdots & 0 \\ 0 & \cdots & 0 & -1 \\ 1 & 0 & -1 & -L \end{bmatrix} \end{array}$ C matrix
Voltage Source	I_E, E, $+$, $-$, i, j	$\begin{array}{c} \\ i \\ \\ j \\ \text{New row} \end{array} \begin{array}{c} \begin{array}{cccc} i & & j & \text{New col} \end{array} \\ \begin{bmatrix} 0 & \cdots & 0 & 1 \\ \vdots & \ddots & \vdots & 0 \\ 0 & \cdots & 0 & -1 \\ 1 & 0 & -1 & 0 \end{bmatrix} \end{array}$ G matrix $\begin{bmatrix} 0 \\ \vdots \\ 0 \\ E \end{bmatrix}$ b vector
Current Source	J, i, j	$\begin{array}{c} i \\ \\ j \end{array} \begin{bmatrix} J \\ \vdots \\ -J \end{bmatrix}$ b vector
Non-linear Conductance	$i = g(v_i - v_j)$, i, j, $+$, v, $-$	$\begin{array}{c} i \\ \\ j \end{array} \begin{bmatrix} -g(v_i - v_j) \\ \vdots \\ g(v_i - v_j) \end{bmatrix}$ $f(x(t))$ vector
Non-linear Capacitor	$q = g(v_i - v_j)$, i, j, $+$, v, $-$	$\begin{array}{c} \\ \text{New row} \end{array} \begin{bmatrix} \vdots \\ -g(v_i - v_j) \end{bmatrix}$ $f(x(t))$ vector $\begin{array}{c} \\ \text{New row} \end{array} \begin{array}{c} \text{New col} \\ \begin{bmatrix} 1 \end{bmatrix} \end{array}$ G matrix $\begin{array}{c} \\ i \\ \\ j \end{array} \begin{array}{c} \text{New col} \\ \begin{bmatrix} 1 \\ \vdots \\ -1 \end{bmatrix} \end{array}$ C matrix

Table 2.1: Basic circuit element stamps

Classical circuit analysis proceeds by applying some discretization formula, e.g. the Gear's method [9], to approximate the differentiation operator which transforms the system (2.1) into a purely nonlinear algebraic system solved by the iterative Newton method [10]. The core computational part of the Newton method is LU factorization of the Jacobian matrix of the system (2.1), which is typically an asymmetric and very sparse matrix. For later usage, this Jacobian matrix is denoted $\boldsymbol{J}$ with size $N \times N$.

2.2 Steady-State Harmonic Balance Analysis

Harmonic Balance Analysis [11] is used to calculate the steady-state response of non-linear circuits to a quasi-periodic stimulus. The steady-state response is obtained by solving a system of nonlinear differential equations in the frequency domain. Harmonic Balance is the main focus of this thesis and is one of two main areas in circuit simulation where block structured matrices arise. The remainder of this section will explain the mathematical concepts behind the Harmonic Balance approach for circuits with single tone excitation.

2.2.1 Single Tone Excitation

A single tone input to a circuit contains a single angular frequency along with integer multiples and can be represented by an infinite Fourier series such as

$$u(t) = U_0 + \sum_{k=1}^{\infty} \left(U_k^C \cos(kw_0 t) + U_k^S \sin(kw_0 t) \right) \tag{2.6}$$

where w_0 is the angular frequency and U_0, U_k^C, U_k^S are the real coefficients of the series. The output waveform $x(t)$ is also represented by a Fourier series, however in order to discretize the series for computation by a computer system, the waveform is truncated to K terms giving the form in Equation (2.7).

$$x(t) = X_0 + \sum_{k=1}^{K} \left(X_k^C \cos(kw_0 t) + X_k^S \sin(kw_0 t) \right) \tag{2.7}$$

The choice of K which will provide a reasonably good approximation to the waveform is largely determined by the shape of $x(t)$, such that a smooth co-sinusoid only requires that $K = 1$. Sharper waveforms such as a square wave require a larger value of K to accurately reproduce $x(t)$ in the Fourier series expansion.

2.2.2 Discrete Fourier Transformation

Expanding Equation (2.7) for all possible values of K we get the following matrix

$$\begin{bmatrix} x_0 \\ x_1 \\ \vdots \\ x_{2K} \end{bmatrix} = \begin{bmatrix} 1 & \cos(w_0 t_0) & \sin(w_0 t_0) & \dots & \cos(K w_0 t_0) & \sin(K w_0 t_0) \\ 1 & \cos(w_0 t_1) & \sin(w_0 t_1) & \dots & \cos(K w_0 t_1) & \sin(K w_0 t_1) \\ \vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\ 1 & \cos(w_0 t_{2K}) & \sin(w_0 t_{2K}) & \dots & \cos(K w_0 t_{2K}) & \sin(K w_0 t_{2K}) \end{bmatrix} \begin{bmatrix} X_0 \\ X_1^C \\ X_1^S \\ \vdots \\ X_K^C \\ X_K^S \end{bmatrix} \tag{2.8}$$

which can also be represented by

$$x = \Gamma^{-1} X \tag{2.9}$$

As long as the time samples are equally spaced in the period $[0, T]$, the Γ^{-1} matrix presented in Equation (2.8) is independent of the fundamental angular frequency w_0 and can be used to represent any periodic stimulus to a circuit. Γ^{-1} is then used to convert any set of $H = 2K + 1$ Fourier coefficients into H equally spaced time-domain samples. Γ^{-1} is henceforth termed as the Inverse Discrete Fourier Transformation (IDFT) operator. Rearranging Equation (2.9) yields

$$X = \Gamma x \tag{2.10}$$

where in the opposite fashion, the operator Γ is the Discrete Fourier Transformation (DFT) operator. As one would expect, the DFT operator converts H evenly spaced time domain samples to their corresponding frequency domain Fourier coefficients.

The IDFT matrix also has some convenient features, such as its columns are orthogonal and have a norm equal to $H/2$ except the first column which has a norm of H. Thus, for any two columns in Γ^{-1}, γ_m and γ_n the following holds true

$$\gamma_m^T \gamma_n = \begin{cases} 0 & m \neq n \\ H/2 & m = n \neq 1 \\ H & m = n = 1 \end{cases} \tag{2.11}$$

therefore, the product $(\Gamma^{-1})^T\Gamma$ yields a diagonal matrix

$$(\Gamma^{-1})^T\Gamma = \begin{bmatrix} H & 0 & \dots & & 0 \\ 0 & \frac{H}{2} & 0 & \dots & 0 \\ \vdots & \ddots & \dots & & 0 \\ 0 & \dots & & & \frac{H}{2} \end{bmatrix} \tag{2.12}$$

2.2.3 Harmonic Balance for Linear Circuits

The Harmonic Balance method is used primarily for steady-state analysis of non-linear circuits. However, its application to linear circuits is a simplified case which eases the introduction of the fundamental understanding of Harmonic Balance. With only linear components, Equation (2.1) becomes

$$\boldsymbol{G}\boldsymbol{x}(t) + \boldsymbol{C}\frac{\mathrm{d}\boldsymbol{x}(t)}{\mathrm{d}t} = \boldsymbol{b}(t) \tag{2.13}$$

where the response of the linear circuit $\boldsymbol{x}(t)$ is periodic with the same period T as the input stimulus $\boldsymbol{b}(t)$ and contains the same angular frequencies. In this case, $\boldsymbol{b}(t)$ can be represented by a Fourier expansion

$$\boldsymbol{b}(t) = \boldsymbol{B}_0 + \sum_{k=1}^{K}\Big(\boldsymbol{B}_k^C \cos(kw_0 t) + \boldsymbol{B}_k^S \sin(kw_0 t)\Big) \tag{2.14}$$

where $w_0 = \frac{2\pi}{T}$ and $\boldsymbol{B}_0$, $\boldsymbol{B}_k^C$, and $\boldsymbol{B}_k^S$ are vectors which collect the kth Fourier coefficients of each independent input source. The output $\boldsymbol{x}(t)$, which is also periodic with period T, can likewise be represented by

$$x(t) = \boldsymbol{X}_0 + \sum_{k=1}^{K}\Big(\boldsymbol{X}_k^C \cos(kw_0 t) + \boldsymbol{X}_k^S \sin(kw_0 t)\Big) \tag{2.15}$$

with vectors of unknowns being $\boldsymbol{X}_0$, $\boldsymbol{X}_k^C$, and $\boldsymbol{X}_k^S$. Substituting Equation (2.14) and Equation (2.15) into Equation (2.13) and taking the derivative to obtain

$$\begin{aligned} &\boldsymbol{G}\Bigg(\boldsymbol{X}_0 + \sum_{k=1}^{K}\Big(\boldsymbol{X}_k^C \cos(kw_0 t) + \boldsymbol{X}_k^S \sin(kw_0 t)\Big)\Bigg) + \\ &\qquad \boldsymbol{C}\Bigg(\boldsymbol{X}_0 + \sum_{k=1}^{K}\Big(-kw_0\boldsymbol{X}_k^C \sin(kw_0 t) + kw_0\boldsymbol{X}_k^S \cos(kw_0 t)\Big)\Bigg) \\ &\qquad\qquad = \boldsymbol{B}_0 + \sum_{k=1}^{K}\Big(\boldsymbol{B}_k^C \cos(kw_0 t) + \boldsymbol{B}_k^S \sin(kw_0 t)\Big) \end{aligned} \tag{2.16}$$

Given that $\cos(kw_0t)$ and $\sin(kw_0t)$ are orthogonal in the period $0 < t < T$, both sides of Equation (2.16) can be multiplied by $\cos(pw_0t)$ and $\sin(pw_0t)$, integrated from 0 to T and finally multiplied by $T/2$ to give the following two systems of equations

$$\boldsymbol{G}\boldsymbol{X}_p^C + pw_0\boldsymbol{C}\boldsymbol{X}_p^S = \boldsymbol{B}_p^C \tag{2.17}$$

$$\boldsymbol{G}\boldsymbol{X}_p^S - pw_0\boldsymbol{C}\boldsymbol{X}_p^C = \boldsymbol{B}_p^S \tag{2.18}$$

where the index p is free to range from 1 to K giving K systems of equations. Finally, the DC coefficient $\boldsymbol{X}_0$ is obtained by integrating Equation (2.16) from 0 to T and dividing by T which produces

$$\boldsymbol{G}\boldsymbol{X}_0 = \boldsymbol{B}_0 \tag{2.19}$$

2.2.4 Harmonic Balance for Non-Linear Circuits

Similar to linear circuits, the output of a general non-linear circuit excited by a single periodic tone will be periodic with the same period as the input signal. However, the non-linearity of the circuit also produces a typically infinite number of harmonics of the input frequency. The output is truncated to order M, assuming that the harmonics contributed by orders higher than M have negligible impact on the steady-state response of the system. Augmenting Equation (2.16) to include non-linear components, the following equation is obtained

$$\begin{aligned}\boldsymbol{G}\Bigg(\boldsymbol{X}_0 + \sum_{m=1}^{M}\Big(\boldsymbol{X}_m^C\cos(mw_0t) + \boldsymbol{X}_m^S\sin(mw_0t)\Big)\Bigg) + \\ \boldsymbol{C}\Bigg(\boldsymbol{X}_0 + \sum_{m=1}^{M}\Big(-mw_0\boldsymbol{X}_m^C\sin(mw_0t) + mw_0\boldsymbol{X}_m^S\cos(mw_0t)\Big)\Bigg) + \\ f\Bigg(\boldsymbol{X}_0 + \sum_{m=1}^{M}\Big(\boldsymbol{X}_m^C\cos(mw_0t) + \boldsymbol{X}_m^S\sin(mw_0t)\Big)\Bigg) \\ = \boldsymbol{B}_0 + \sum_{k=1}^{K}\Big(\boldsymbol{B}_k^C\cos(kw_0t) + \boldsymbol{B}_k^S\sin(kw_0t)\Big)\end{aligned} \tag{2.20}$$

The non-linear component $\boldsymbol{f}(\boldsymbol{x}(t))$ can be expanded to the following Fourier series

$$\boldsymbol{f}(\boldsymbol{x}(t)) = \boldsymbol{F}_0 + \sum_{m=1}^{\infty}\Big(\boldsymbol{F}_m^C\cos(mw_0t) + \boldsymbol{F}_m^S\sin(mw_0t)\Big) \tag{2.21}$$

where $\boldsymbol{F}_0$, $\boldsymbol{F}_m^C$ and $\boldsymbol{F}_m^S$ are vectors containing the mth Fourier coefficients of the non-linear functions in $\boldsymbol{f}(\boldsymbol{x}(t))$. Here it can be noted that $\boldsymbol{F}_0$, $\boldsymbol{F}_m^C$ and $\boldsymbol{F}_m^S$ are in turn dependent upon all $\boldsymbol{X}_0$, $\boldsymbol{X}_m^C$ and $\boldsymbol{X}_m^S$. All values of $\boldsymbol{F}_0$, $\boldsymbol{F}_m^C$ and $\boldsymbol{F}_m^S$ are collected into a single vector and denoted as $\bar{\boldsymbol{X}}$ which appears as follows

$$\bar{\boldsymbol{X}} = \begin{bmatrix} \boldsymbol{X}_0 & \boldsymbol{X}_1^C & \boldsymbol{X}_1^S & \boldsymbol{X}_2^C & \boldsymbol{X}_2^S & \dots & \boldsymbol{X}_K^C & \boldsymbol{X}_K^S \end{bmatrix}^T \tag{2.22}$$

where $\bar{\boldsymbol{X}} \in \mathbb{R}^{N \times (2M+1)}$ and N is the number of MNA variables. $\boldsymbol{f}(\boldsymbol{x}(t))$ can now be re-written as

$$\boldsymbol{f}(\boldsymbol{x}(t)) = \boldsymbol{F}_0(\bar{\boldsymbol{X}}) + \sum_{m=1}^{\infty} \Big(\boldsymbol{F}_m^C(\bar{\boldsymbol{X}}) \cos(mw_0 t) + \boldsymbol{F}_m^S(\bar{\boldsymbol{X}}) \sin(mw_0 t) \Big) \tag{2.23}$$

Using the same orthogonal property as in the linear circuit case, both sides of Equation (2.20) are multiplied by $\cos(pw_0 t)$ and $\sin(pw_0 t)$, integrated from 0 to T and finally multiplied by $T/2$ to give the following two systems of equations

$$\boldsymbol{G}\boldsymbol{X}_p^C + pw_0 \boldsymbol{C}\boldsymbol{X}_p^S + \boldsymbol{F}_p^C(\bar{\boldsymbol{X}}) = \boldsymbol{B}_p^C \tag{2.24}$$

$$\boldsymbol{G}\boldsymbol{X}_p^S - pw_0 \boldsymbol{C}\boldsymbol{X}_p^C + \boldsymbol{F}_p^S(\bar{\boldsymbol{X}}) = \boldsymbol{B}_p^S \tag{2.25}$$

and similar to the linear case, the DC component is

$$\boldsymbol{G}\boldsymbol{X}_0 + \boldsymbol{F}_0(\bar{\boldsymbol{X}}) = \boldsymbol{B}_0 \tag{2.26}$$

With p ranging over $\{1, 2...\infty\}$ this gives an infinite number of systems. As an example, Equation (2.24), Equation (2.25), and Equation (2.26) can be expanded with $2M+1$ systems of equations as follows

$$\begin{aligned}
\boldsymbol{GX}_0 + \boldsymbol{F}_0(\bar{\boldsymbol{X}}) &= \boldsymbol{B}_0 \\
\boldsymbol{GX}_1^C + w_0\boldsymbol{CX}_1^S + \boldsymbol{F}_1^C(\bar{\boldsymbol{X}}) &= \boldsymbol{B}_1^C \\
\boldsymbol{GX}_1^S - w_0\boldsymbol{CX}_1^C + \boldsymbol{F}_1^S(\bar{\boldsymbol{X}}) &= \boldsymbol{B}_1^S \\
\vdots &= \vdots \\
\boldsymbol{GX}_K^C + Kw_0\boldsymbol{CX}_K^S + \boldsymbol{F}_K^C(\bar{\boldsymbol{X}}) &= \boldsymbol{B}_K^C \\
\boldsymbol{GX}_K^S - Kw_0\boldsymbol{CX}_K^C + \boldsymbol{F}_K^S(\bar{\boldsymbol{X}}) &= \boldsymbol{B}_K^S \\
\boldsymbol{GX}_{K+1}^C + (K+1)w_0\boldsymbol{CX}_{K+1}^S + \boldsymbol{F}_{K+1}^C(\bar{\boldsymbol{X}}) &= \boldsymbol{0} \\
\boldsymbol{GX}_{K+1}^S - (K+1)w_0\boldsymbol{CX}_{K+1}^C + \boldsymbol{F}_{K+1}^S(\bar{\boldsymbol{X}}) &= \boldsymbol{0} \\
\vdots &= \vdots \\
\boldsymbol{GX}_M^C + Mw_0\boldsymbol{CX}_M^S + \boldsymbol{F}_M^C(\bar{\boldsymbol{X}}) &= \boldsymbol{0} \\
\boldsymbol{GX}_M^S - Mw_0\boldsymbol{CX}_M^C + \boldsymbol{F}_M^S(\bar{\boldsymbol{X}}) &= 0 \\
\boldsymbol{F}_{M+1}^C(\bar{\boldsymbol{X}}) &= \boldsymbol{0} \\
\boldsymbol{F}_{M+1}^S(\bar{\boldsymbol{X}}) &= \boldsymbol{0}
\end{aligned} \tag{2.27}$$

where each equation in Equation (2.27) is of size N, the number of MNA variables. Notice that for $p > K$ the input coefficients, $\boldsymbol{B}_1^C$ and $\boldsymbol{B}_1^S$, become zero because the Fourier series is truncated to K. Truncating the system to M equations, Equation (2.27) can be written as

$$\bar{\boldsymbol{Y}}\bar{\boldsymbol{X}} + \bar{\boldsymbol{F}}(\bar{\boldsymbol{X}}) = \bar{\boldsymbol{B}} \tag{2.28}$$

with size $N \times (2M+1)$ and $\bar{\boldsymbol{Y}} \in \mathbb{R}^{N(2M+1)\times N(2M+1)}$ being the following block diagonal matrix

$$\bar{\boldsymbol{Y}} = \begin{bmatrix} \boldsymbol{G} & \boldsymbol{0} & \boldsymbol{0} & \dots & \dots & \dots & \dots & \dots & \boldsymbol{0} \\ \boldsymbol{0} & \boldsymbol{G} & w_0\boldsymbol{C} & \boldsymbol{0} & \boldsymbol{0} & \dots & \dots & \dots & \boldsymbol{0} \\ \boldsymbol{0} & -w_0\boldsymbol{C} & \boldsymbol{G} & \boldsymbol{0} & \boldsymbol{0} & \dots & \dots & \dots & \boldsymbol{0} \\ \boldsymbol{0} & \boldsymbol{0} & \boldsymbol{0} & \boldsymbol{G} & 2w_0\boldsymbol{C} & \boldsymbol{0} & \dots & \dots & \boldsymbol{0} \\ \boldsymbol{0} & \boldsymbol{0} & \boldsymbol{0} & -2w_0\boldsymbol{C} & \boldsymbol{G} & \boldsymbol{0} & \dots & \dots & \boldsymbol{0} \\ \vdots & \vdots & \vdots & & \ddots & \ddots & \ddots & \vdots & \vdots \\ \boldsymbol{0} & \boldsymbol{0} & \boldsymbol{0} & \dots & \dots & \dots & \dots & \boldsymbol{G} & Mw_0\boldsymbol{C} \\ \boldsymbol{0} & \boldsymbol{0} & \boldsymbol{0} & \dots & \dots & \dots & \dots & -Mw_0\boldsymbol{C} & \boldsymbol{G} \end{bmatrix} \tag{2.29}$$

The vectors $\bar{\boldsymbol{F}}(\bar{\boldsymbol{X}})$ and $\bar{\boldsymbol{B}}$ are $N \times (2M+1)$ sized collections of Fourier coefficients,

$$\bar{\boldsymbol{F}}(\bar{\boldsymbol{X}}) = \begin{bmatrix} \boldsymbol{F}_0(\bar{\boldsymbol{X}}) \\ \boldsymbol{F}_1^C(\bar{\boldsymbol{X}}) \\ \boldsymbol{F}_1^S(\bar{\boldsymbol{X}}) \\ \vdots \\ \boldsymbol{F}_M^C(\bar{\boldsymbol{X}}) \\ \boldsymbol{F}_M^S(\bar{\boldsymbol{X}}) \end{bmatrix} \tag{2.30}$$

$$\bar{\boldsymbol{B}} = \begin{bmatrix} \boldsymbol{B}_0 \\ \boldsymbol{B}_1^C \\ \boldsymbol{B}_1^S \\ \vdots \\ \boldsymbol{B}_p^C \\ \boldsymbol{B}_p^S \\ \boldsymbol{0} \\ \boldsymbol{0} \end{bmatrix} \tag{2.31}$$

Solving the nonlinear system of equations in Equation (2.28) can only be done using iterative techniques, such as the Newton-Raphson (NR) method which is discussed next.

2.2.5 Newton-Raphson Iterative Method for Nonlinear Systems

Given a system of nonlinear equations, the Newton-Raphson method seeks to find a vector $\boldsymbol{x}$ which can satisfy the equation

$$\Psi(\boldsymbol{x}) = \mathbf{0} \tag{2.32}$$

for a given system of n nonlinear equations such as

$$\begin{aligned} \psi_1(x_1, x_2, ..., x_n) &= 0 \\ \psi_2(x_1, x_2, ..., x_n) &= 0 \\ &\vdots \\ \psi_n(x_1, x_2, ..., x_n) &= 0 \end{aligned} \tag{2.33}$$

with a solution vector

$$\boldsymbol{x} = \begin{bmatrix} x_1 & x_2 & \dots & x_n \end{bmatrix}^T \tag{2.34}$$

Solving a given nonlinear system of equations starts with an initial guess for the solution vector denoted by $x^{(0)}$ and is also referred to as the nth trial vector. Substituting $x = x^{(0)}$ into Equation (2.32) gives

$$\Psi(\boldsymbol{x}^{(0)}) = \Delta^{(0)} \tag{2.35}$$

where $\Delta^{(0)}$ is the error. To find the next trial vector $x^{(1)}$, the Jacobian matrix is constructed from the derivatives of the nonlinear system of equations with the following structure

$$\boldsymbol{J} = \begin{bmatrix} \frac{\partial \psi_1}{\partial x_1} & \frac{\partial \psi_1}{\partial x_2} & \cdots & \frac{\partial \psi_1}{\partial x_n} \\ \frac{\partial \psi_2}{\partial x_1} & \frac{\partial \psi_2}{\partial x_2} & \cdots & \frac{\partial \psi_2}{\partial x_n} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial \psi_n}{\partial x_1} & \frac{\partial \psi_n}{\partial x_2} & \cdots & \frac{\partial \psi_n}{\partial x_n} \end{bmatrix} \equiv \frac{\partial \Psi}{\partial \boldsymbol{x}} \tag{2.36}$$

the correction term is then given by

$$\boldsymbol{\delta_x} = (\boldsymbol{J}|_{x=x^{(0)}})^{-1} \boldsymbol{\Delta}^{(0)} \tag{2.37}$$

The next trial vector is computed by

$$\boldsymbol{x}^{(1)} = \boldsymbol{x}^{(0)} - \boldsymbol{\delta_x} \tag{2.38}$$

This process is repeated iteratively until an acceptable error term threshold is reached and $\boldsymbol{x}^{(p)}$ is taken as the solution vector where $p+1$ is the number of iterations required to reach the threshold.

2.2.6 Solving HB Using Newton-Raphson

To apply the Newton-Raphson to the Harmonic Balance system of equations, Equation (2.28) can be rewritten as

$$\Phi(\bar{\boldsymbol{X}}) = \bar{\boldsymbol{Y}}\bar{\boldsymbol{X}} + \bar{\boldsymbol{F}}(\bar{\boldsymbol{X}}) - \bar{\boldsymbol{B}} = \mathbf{0} \tag{2.39}$$

such that the solution to $\Phi(\bar{\boldsymbol{X}}) = 0$ is sought. The initial guess $\bar{\boldsymbol{X}}^{(0)}$ is typically taken to be the DC solution of the system giving

$$\Phi(\bar{\boldsymbol{X}}^{(0)}) = \bar{\boldsymbol{Y}}\bar{\boldsymbol{X}}^{(0)} + \bar{\boldsymbol{F}}(\bar{\boldsymbol{X}}^{(0)}) - \bar{\boldsymbol{B}} \tag{2.40}$$

The nonlinear term $\bar{\boldsymbol{F}}(\bar{\boldsymbol{X}}^{(0)})$ cannot be directly computed and must be computed numerically using Newton-Raphson.

1. By default, the vector $\bar{\boldsymbol{X}}^{(0)}$ is arranged in Harmonic-major/Node-minor ordering, which means that components are ordered by their harmonic index then by their MNA index. To proceed, the harmonics of each MNA variable need to be collected in what is referred to as Harmonic-minor/Node-major ordering, denoted as $\bar{\boldsymbol{X}}^{(0)}_{node}$, by multiplying with a suitable permutation matrix $\boldsymbol{P}$ then

   $$\bar{\boldsymbol{X}}^{(0)}_{node} = \boldsymbol{P}\bar{\boldsymbol{X}}^{(0)} \tag{2.41}$$

   The differences in the ordering schemes between $\boldsymbol{P}\bar{\boldsymbol{X}}^{(j)}$ and $\bar{\boldsymbol{X}}^{(j)}_{node}$ are shown in Figure 2.2.

2. The harmonics of each MNA variable are multiplied by the IDFT operator to get the corresponding time domain waveform. Each of the subvectors in $\bar{\boldsymbol{X}}^{(j)}_{node}$ is multiplied by Γ^{-1}.

   $$\bar{\boldsymbol{x}}^{(j)}_{node} = \bar{\boldsymbol{\Gamma}}^{-1}\bar{\boldsymbol{X}}^{(j)}_{node} \tag{2.42}$$

   where $\bar{\boldsymbol{\Gamma}}^{-1}$ is a block diagonal matrix of size $N(2M+1) \times N(2M+1)$ with the Γ^{-1} matrix along its diagonal.

3. In this step the nonlinear component $\boldsymbol{f}(\boldsymbol{x}(t))$ is computed by evaluating each component $f_i(x(t))$

$$\bar{\boldsymbol{X}}^{(j)} = \begin{bmatrix} \begin{bmatrix} \boldsymbol{X}_{0,1}^{(J)} \\ \boldsymbol{X}_{0,2}^{(J)} \\ \vdots \\ \boldsymbol{X}_{0,N}^{(J)} \end{bmatrix} \\ \begin{bmatrix} \boldsymbol{X}_{1,1}^{(J),C} \\ \boldsymbol{X}_{1,2}^{(J),C} \\ \vdots \\ \boldsymbol{X}_{1,N}^{(J),C} \end{bmatrix} \\ \begin{bmatrix} \boldsymbol{X}_{1,1}^{(J),S} \\ \boldsymbol{X}_{1,2}^{(J),S} \\ \vdots \\ \boldsymbol{X}_{1,N}^{(J),S} \end{bmatrix} \\ \vdots \\ \begin{bmatrix} \boldsymbol{X}_{M,1}^{(J),S} \\ \boldsymbol{X}_{M,2}^{(J),S} \\ \vdots \\ \boldsymbol{X}_{M,N}^{(J),S} \end{bmatrix} \end{bmatrix}$$

(a) Harmonic-major/Node-minor ordering of $\bar{\boldsymbol{X}}^{(j)}$

$$\bar{\boldsymbol{X}}_{node}^{(j)} = \begin{bmatrix} \begin{bmatrix} \boldsymbol{X}_{0,1}^{(J)} \\ \boldsymbol{X}_{1,1}^{(J),C} \\ \boldsymbol{X}_{1,1}^{(J),S} \\ \vdots \\ \boldsymbol{X}_{M,1}^{(J),C} \\ \boldsymbol{X}_{M,1}^{(J),S} \end{bmatrix} \\ \vdots \\ \begin{bmatrix} \boldsymbol{X}_{0,N}^{(J)} \\ \boldsymbol{X}_{1,N}^{(J),C} \\ \boldsymbol{X}_{1,N}^{(J),S} \\ \vdots \\ \boldsymbol{X}_{M,N}^{(J),C} \\ \boldsymbol{X}_{M,N}^{(J),S} \end{bmatrix} \end{bmatrix}$$

(b) Harmonic-minor/Node-major ordering of $\bar{\boldsymbol{X}}_{node}^{(j)}$

Figure 2.2: Ordering of $\bar{\boldsymbol{X}}^{(j)}$ and $\bar{\boldsymbol{X}}_{node}^{(j)}$

at the time samples $t_0, t_1, ...t_{2M}$. The result is a vector denoted as $\bar{\boldsymbol{f}}^{(j)}_{node}$ which contains the time domain samples for each component $f_i(\boldsymbol{x}(t))$.

4. The Fourier series coefficients of $\bar{f}^{(j)}_{node}$ are obtained by multiplying by the DFT operator such that

$$\bar{\boldsymbol{F}}^{j}_{node} = \bar{\boldsymbol{\Gamma}}\bar{\boldsymbol{f}}^{(j)}_{node} \tag{2.43}$$

where $\bar{\boldsymbol{\Gamma}}$ is a $N(2M+1) \times N(2M+1)$ block diagonal matrix having the matrix $\bar{\boldsymbol{\Gamma}}$ as its diagonal blocks.

5. The final step is to return $\bar{\boldsymbol{F}}^{(j)}_{node}$ to the harmonic-major/node-minor ordering scheme by using the permutation matrix $\boldsymbol{P}$.

$$\bar{\boldsymbol{F}}(\bar{\boldsymbol{X}}^{j}) = \boldsymbol{P}^{T}\bar{\boldsymbol{F}}^{(j)}_{node} \tag{2.44}$$

$\bar{\boldsymbol{F}}(\bar{\boldsymbol{X}}^{j})$ can now be used to substitute back into Equation (2.39) and continue with the computation.

The Jacobian matrix $\partial\Phi(\bar{\boldsymbol{X}})/\partial\bar{\boldsymbol{X}}|_{\bar{\boldsymbol{X}}=\bar{\boldsymbol{X}}^{(j)}}$ is obtained by taking the partial derivatives Φ with respect to $\bar{\boldsymbol{X}}$ which yields

$$\frac{\partial\Phi}{\partial\bar{\boldsymbol{X}}} = \bar{\boldsymbol{Y}} + \frac{\partial\bar{\boldsymbol{F}}(\bar{\boldsymbol{X}})}{\partial\bar{\boldsymbol{X}}} \tag{2.45}$$

It is now sought to compute $\frac{\partial\Phi}{\partial\bar{\boldsymbol{X}}}$ at the jth trial vector $\bar{\boldsymbol{X}}^{(j)}$, with the majority of the computational complexity coming from the nonlinear term $\bar{\boldsymbol{F}}(\bar{\boldsymbol{X}})$. The structure of the Jacobian matrix $\frac{\partial\Phi}{\partial\bar{\boldsymbol{X}}}$ will be covered in an upcoming section.

2.3 Uncertainty Quantification via Polynomial Chaos

Briefly, another area in circuit simulation which produces block-structure matrices is Uncertainty Quantification via Polynomial Chaos. In this situation the circuit is characterized by a set of, say d, design parameters whose values are subject to uncertainty due to the inherent semi-conductor process variability. The objective is to quantify the corresponding uncertainty in the circuit performance. To this end, the Polynomial Chaos [12] approach expands the circuit variables $\boldsymbol{x}(t)$ as follows

$$\boldsymbol{x}(t,\boldsymbol{\xi}) = \sum_{\boldsymbol{\alpha}\in\Lambda_M} \boldsymbol{X}_{\boldsymbol{\alpha}}(t)\phi(\boldsymbol{\xi}) \tag{2.46}$$

where $\boldsymbol{\xi} \in \mathbb{R}^d$ is a vector grouping the uncertain design parameters and $\phi(\boldsymbol{\xi})$ is orthogonal multi-dimensional polynomials of the Askey-Wiener type [13], whereas $\boldsymbol{X}_{\boldsymbol{\alpha}}(t) \in \mathbb{R}^N$ are time-dependent coefficients.

Substitution from (2.46) into (2.1) and using the process of Galerkin projection eliminates the vector of parameters $\boldsymbol{\xi}$ and leaves a system of nonlinear DAE of size NM, where M is the number of coefficients in the series expansion of (2.46). Again, another level of discretization can be invoked to approximate the differential operator and leading to system of nonlinear algebraic equations that is solved by the Newton iterative method.

2.4 Structure of the Jacobian Matrix

The Jacobian matrix spawned by the Harmonic Balance and Uncertainty Quantification via Polynomial Chaos problems has size that is M times the size of the original Jacobian matrix, $\boldsymbol{J}$. This matrix (denoted $\boldsymbol{\mathcal{J}} \in \mathbb{R}^{NM \times NM}$) inherits the sparse structure of $\boldsymbol{J}$ which arises mainly from the sparse circuit interconnections. However, the coefficients introduced by the series expansions (e.g., (2.7) and (2.46)) are strongly coupled, a fact that is reflected in $\boldsymbol{\mathcal{J}}$ by the appearance of dense spots.

It is possible to describe the structure of $\boldsymbol{\mathcal{J}}$ more concisely by specifying the ordering scheme of the NM unknowns of the resulting nonlinear problem. For example, assume that those unknowns are ordered in such a way that M coefficients of each of the N circuit variables are grouped in contiguous subvectors. The resulting structure of $\boldsymbol{\mathcal{J}}$ can be thought of as being identical to $\boldsymbol{J}$, except that each non-zero scalar entry of $\boldsymbol{J}$ is replaced by a block of size M.

Typically, the blocks of $\boldsymbol{\mathcal{J}}$ are dense due to the strong coupling between the expansion coefficients. In addition the size of the block, M, can reach up to the order of few hundreds or thousands. For example, if a third-order degree ($P = 3$) polynomial of the Hermite type is used for a circuit with say $d = 9$ uncertain design parameters, then the size of the block becomes $M = (P + d)!/(P!d!) = 715$.

2.5 Introduction to General Purpose GPU

2.5.1 Introduction

Graphics processing units (GPUs) have traditionally been designed specifically for graphics processing applications such as three dimensional (3-D) graphics. With the advent of General Purpose GPU computing, GPU architectures have evolved to allow acceleration of general applications that were typically executed on one or more CPUs. With GPU hardware now being a commodity component, they are readily available in most consumer devices and advanced supercomputers. Applications now have access to hardware acceleration through these massively parallel platforms which in the past was not a possible avenue for accelerating general purpose applications. Languages such as CUDA and OpenCL are at the forefront of enabling general purpose workloads on graphics hardware. The work presented in this thesis utilizes GPU hardware based on AMD's Graphics Core Next (GCN) architecture, and as such, the discussion will center around AMD architecture and device specifics.

2.5.2 AMD Graphics Core Next Architecture

Graphics Core Next (GCN) is AMD's first GPU architecture designed with an emphasis on General Purpose GPU (GPGPU) computing. GCN uses a RISC (Reduced Instruction Set Computing) instruction set instead of the previous generation's VLIW (Very Long Instruction Word) architecture. In VLIW architectures, parallelism in tasks must be detected and scheduled at compile time whereas with the shift to RISC based architecture, dynamic scheduling of parallel work can be done at run-time. The remainder of this section discusses the details of the AMD GCN architecture as it relates to general purpose computing.

The GCN architecture consists of several Compute Units (CU) which are each comprised of four Single Instruction Multiple Data (SIMD) units as shown in the CU top-level architecture diagram in Figure 2.3. As the name implies, the SIMD units follow the classification defined by Flynn [14] whereby a single instruction is executed for multiple pieces of data. Each SIMD unit further contains 16 ALUs with the SIMD often being referred to as a single vector ALU with a width of 16. The SIMD executes a single instruction over a vector width of 16 elements simultaneously. The native vector width of GCN architecture is 64 elements and is referred to as a wavefront. On a single SIMD, one instruction is completed for an entire wavefront over four clock cycles. For example, in cycle one the instruction is

executed for elements 0-15, in cycle two the same instruction is executed for elements 16-31, and so on. Each SIMD can handle 10 ready state wavefronts with an entire CU capable of 40 wavefronts. Coupled with hardware-based context switching this allows an SIMD to remain busy during a memory access stall by switching out a stalled wavefront and selecting another which is ready for execution. Additionally, each SIMD can function independently of the others in a single compute unit. A single scalar unit is

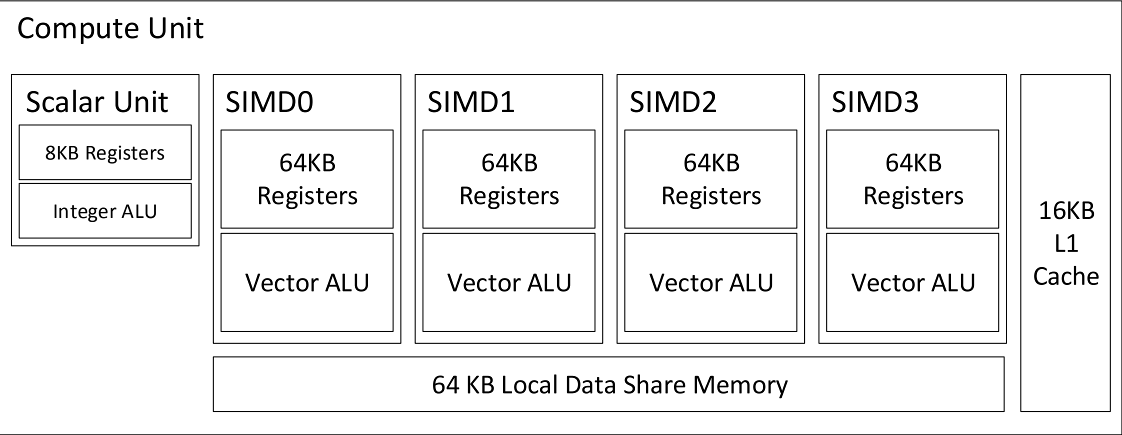


Figure 2.3: Simplified Compute Unit Architecture

shared by all four SIMDs in the compute unit and is used for control flow and other scalar operations. The scalar unit's 8KB register file is divided evenly between all four SIMDs, giving each 512 entries of 32 bits each. Each SIMD also has its own 64KB vector register file as well as access to a 64KB local data share memory which is used for synchronization and communication.

2.6 Introduction to OpenCL

2.6.1 OpenCL Overview

OpenCL (Open Computing Language) is a standard for writing portable parallel applications which run across a multitude of heterogeneous platforms. Applications written using the OpenCL API (Application Programming Interface) can target accelerator devices such as Accelerated Processing Units (APUs), multi-core CPUs, GPUs, and Field-Programmable Gate Arrays (FPGA) without modification or maintaining multiple versions of the application. An OpenCL system consists of a host and one or more accelerators, which run kernels written in a subset of the C99 standard defined by [15]. OpenCL operations are exposed to host applications via a standardized API while the underlying host-to-hardware

interaction is implemented by the device vendor. The OpenCL standard is best described by four models: the platform model, the execution model, the memory model, and the programming model. The four models as well as methods for performing synchronization are discussed in the following subsections.

2.6.2 Platform Model

The OpenCL platform model defines the relationship between the host and the attached accelerators (Such as GPUs) which support OpenCL. OpenCL uses an abstracted view of the underlying hardware that is logically divided into one or more Compute Units, which are further split into Processing Elements (Figure 2.4).

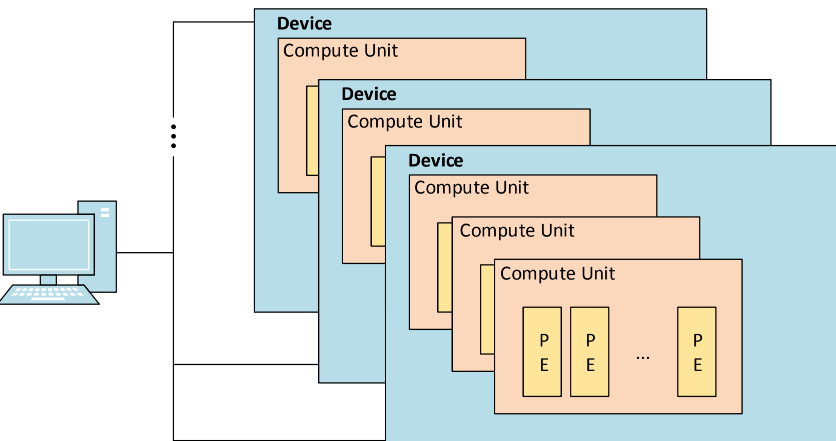


Figure 2.4: OpenCL Platform Model

The hardware vendor's OpenCL implementation maps this OpenCL abstraction onto their specific underlying hardware which can vary drastically between devices such as GPUs, CPUs and FGPAs. In a single host system there can be multiple platforms present depending on the installed implementations, with each platform supporting one or more devices. For example, a host with two GPUs, each from different vendors, and an FPGA, could have three platforms implemented by each of the GPU vendors as well as the FPGA vendor.

2.6.3 Execution Model

The execution model is divided into two parts, the host bound execution which invokes the OpenCL API, and the kernel execution which is performed on the accelerator. On the device side, the smallest unit of work is referred to as a work-item, which represents a single instance of a kernel. A work-item is indexed in a space referred to as the NDRange. The NDRange can be one, two, or three dimensional, and for each point in the NDRange there exists a kernel instance (work-item). A work-group is defined as a grouping of multiple work-items which execute concurrently on a single compute unit.

Figure 2.5 shows a 2-dimensional NDRange, where the range is composed of $N_x \times N_y$ work-items. The work-items are segregated into work-groups of size $WG_x \times WG_y$ where each work-item is given a unique identifier within the work-group. Work-items are also indexed globally, which identifies them based on their index in the NDRange space. The total number of work-groups in the space is given by

$$(W_x, W_y) = (\frac{N_x}{WG_x}, \frac{N_y}{WG_y}) \tag{2.47}$$

where all of the parameters WG_x, WG_y, N_x and N_y are defined by the application and the problem to be solved. Target devices will typically have a limit for the work-group size due to hardware constraints.

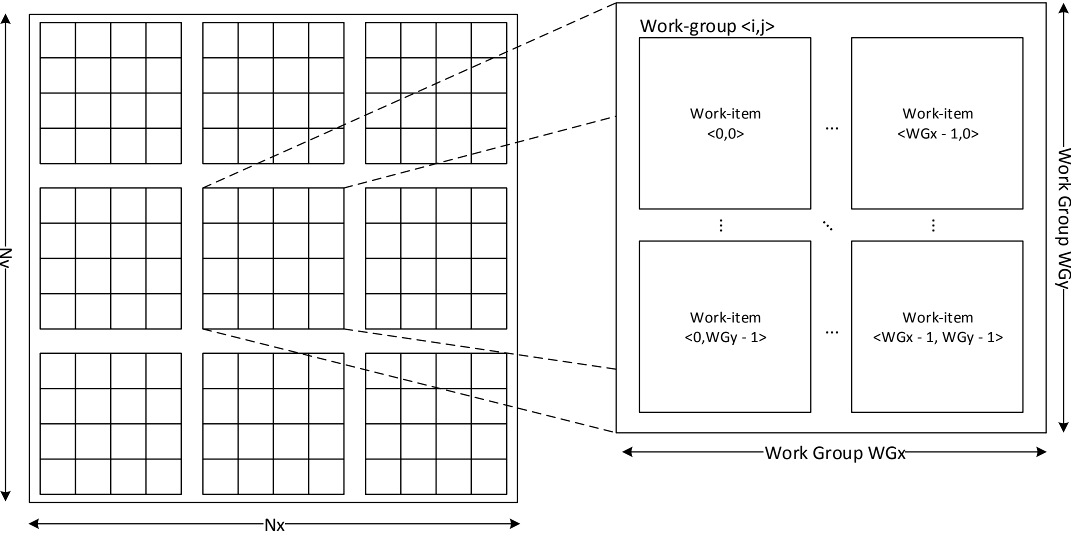


Figure 2.5: OpenCL Execution Model - Relationship between a 2-d NDRange, work-groups, and work-items.

On the host side, the OpenCL execution model uses contexts to group resources containing one or more

of the following: devices, kernels, memory objects, and command queues. These resources cannot be shared between contexts, but only between devices in the same context. Devices belonging to the same platform can be contained in a single context or each belong to their own context. Command queues are assigned to devices within the context and facilitates issuing of memory and kernel execution commands to that device.

2.6.4 Memory Model

The memory model defines an abstract hierarchy which OpenCL kernels use without requiring any knowledge of the underlying hardware organization. The memory abstraction is split up into 4 sections: global, constant, local, and private memories. This is shown in Figure 2.6. Global and constant memory are accessible by any work-item on the device, while local memory is shared between work-items within the same work-group. Private memory is only accessible by the work-item to which it is assigned. As its name implies, constant memory does not change during the execution of a kernel and is set by the host. All work-items have both read and write access to the entire global memory while variables defined in local memory can be accessed by any work-item within the work-group.

OpenCL has relaxed assumptions about the consistency of memory, thus the state of memory is not guaranteed to be consistent in all situations [15]. Local and global memory is guaranteed to be consistent for work-items in a single work-group only after a work-group barrier is encountered by all work-items. There are no guarantees of consistency between work-groups. Additionally, since memory is not shared between contexts, using multiple contexts requires the explicit transfer of data between devices belonging to different contexts. Finally, memory objects in OpenCL are referred to as buffers and can be allocated in host and device memory.

2.6.5 Programming Model

The OpenCL programming model supports two parallel paradigms: task-parallel and data-parallel. OpenCL's primary use-case is primarily directed towards the data parallelism paradigm.

Data Parallel Model

OpenCL provides a data parallel model whereby the application programmer defines the number of work-items which execute in parallel. The number of work-items per work-group can either be defined

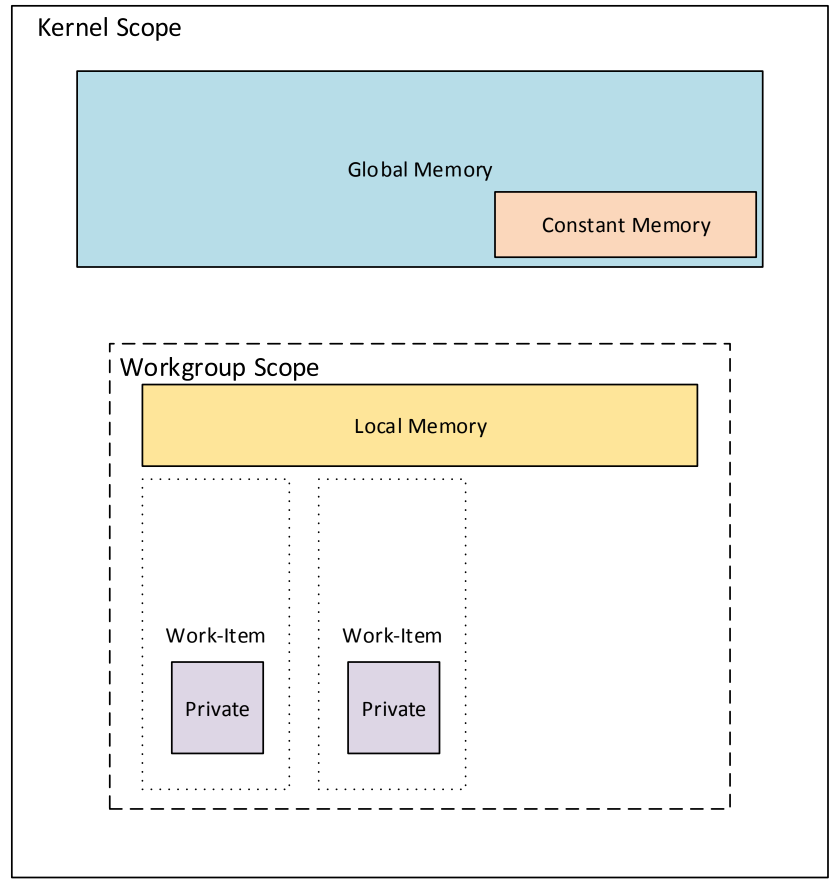


Figure 2.6: OpenCL Memory Model

or chosen by the OpenCL implementation.

Task Parallel Model

Task parallelism in OpenCL supports the execution of a kernel as a single work-item. Thus, it executes without any dimensionality (i.e. an NDRange of 1) allowing the issuing of multiple kernels intended to be run in parallel.

2.6.6 Synchronization in OpenCL

Support for synchronization in the OpenCL API is limited and warrants some discussion around how synchronization constructs can be used to achieve data consistency and enforce execution order. In OpenCL, synchronization can only be performed at two levels, on the host side between commands issued within the same context and on the device between work-items in the same work-group. Aside

from these two avenues, further synchronization is not supported. Barrier operations are used to perform synchronization between the work-items within a work-group. No work-items in a work-group can proceed beyond a barrier until each one has reached the barrier. It is not possible to synchronize between work-groups, even if they belong to the same kernel.

On the host side, most OpenCL API calls are asynchronous and return to the calling thread, without any guarantee that the requested operation has been scheduled or submitted to the device. Specifying a command queue as in-order is one way of enforcing synchronization, as this forces that kernels are executed in the same order they were submitted to the queue. The following three API functions will be used to exemplify the other avenues for host side synchronization.

1. $clEnqueueWriteBuffer(..., blocking_write, ..., *event_wait_list, *event)$ - Enqueue a command to write to a buffer in device memory from host memory.

2. $clEnqueueReadBuffer(..., blocking_read, ..., *event_wait_list, *event)$ - Enqueue a command to read from a buffer in device memory to host memory.

3. $clEnqueueNDRangeKernel(..., *event_wait_list, *event)$ - Enqueues a kernel for execution on a device

Host Blocking Synchronization

The Functions $clEnqueueWriteBuffer()$ and $clEnqueueReadBuffer()$ both accept a parameter ($blocking_write$ / $blocking_read$) which indicates the function should return before the memory transfer has been completed. If an in-order queue (commands as guaranteed to be executed in the order they were submitted) is used, the function will block until all previously enqueued commands plus the memory transfer are complete. The function $clEnqueueNDRangeKernel()$, has no such parameter, instead the $clFinish(command_queue)$ function must be used to achieve blocking synchronization by blocking the thread until the device has completed all previously enqueued work. In all of these scenarios, the application will suffer from the latency involved in scheduling and executing commands on the device then waiting for their completion.

Host Event Synchronization

Instead of enforcing synchronization though blocking functions, OpenCL provides an eventing system which is used to build a dependency tree that will be respected during scheduling and execution, even across multiple command queues within the same context. Event objects are set by the last parameter of the three functions above and represent the state of the command to which it has been attached. The

following are the possible states an event object can represent.

- CL_QUEUED: The command has been added to the command queue
- CL_SUBMITED: The command has been submitted to the device
- CL_RUNNING: The compute device is currently executing the command
- CL_COMPLETE: The command finished executing on the device
- ERROR_CODE: A negative value indicting an error condition

An event object can be added to any command's *event_wait_list* and the operation will only start execution after the event's state is changed to CL_COMPLETE. User defined callbacks can also be attached to an event's state, where upon changing to that state, a callback function is asynchronously invoked. Event objects can also be created by the user, with their state also being set by the user. User events are useful for coordinating tasks between multiple contexts as non-user event objects cannot be shared across contexts. The main advantage of using OpenCL eventing is to encourage the submission of commands in bulk which allows the OpenCL implementation to handle the dependencies between kernels. This effectively lowers the latency between when a command is enqueued and its execution on the device.

2.7 Mapping OpenCL to GCN Architecture

This section covers the mapping of OpenCL concepts onto AMD's GCN architecture and will aid in the discussion presented in Chapter 8 on the challenges of factorizing matrices with small block sizes. Figure 2.7 shows how the OpenCL execution model is applied to the GCN hardware, where all work-items of a single work-group are executed on the same compute unit.

Work-groups are scheduled into one or more wavefronts, depending on the number of work-items they consist of. On the GCN architecture, the wavefront is a collection of 64 work-items. The work-items of the wavefront are executed in lock-step on a single SIMD, while the wavefronts which comprise the entire work-group can be executed on any of the SIMD's in the same compute unit. Wavefronts which contain less than 64 work-items will leave some of the SIMD's processing elements idle, with the side effect of contributing to poor hardware utilization. This places a strong importance on the ability to

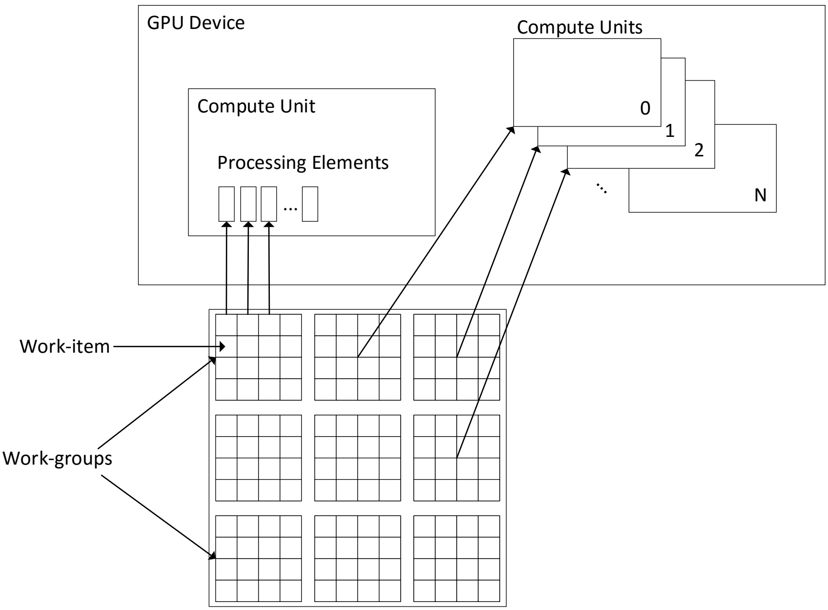


Figure 2.7: OpenCL Device Mapping

extract enough parallelism from an algorithm to satisfy the hardware's optimal execution width.

GCN uses active wavefronts, which are wavefronts ready to execute, as a means to hide memory access latency. A read to global memory has a latency between 300 and 600 cycles [16], leaving the wavefront which initiated the memory access idle until the memory operation completes. In a single cycle the GPU can switch to another ready-state wavefront which will hide the memory access latency of a stalled wavefront. This puts additional emphasis on providing the GPU with an adequate amount of work to perform. To hide access latency, the GCN architecture requires a minimum of four ready to run wavefronts.

When flow control operations such as branches are encountered in a work-item, it causes the entire wavefront to execute all possible paths serially. This can have a negative impact on performance since a single work-item taking a divergent path will force the entire wavefront to also execute the path.

2.8 Related Work and Existing Approaches

With the factorization of the Jacobian Matrix being one of the most common operations in circuit simulation, current research has explored how to increase the computational advantage in computing

the LU factors of $\mathcal{J}$. The large size and dense structure of $\mathcal{J}$ has prompted early research to investigate ways of handling it within the Newton iterative solving technique. One research avenue has sought to entirely avoid direct factorization of the matrix through using the Krylov-subspace-based iterative techniques [4]. Those techniques rely on using an iterative sequence of matrix-vector products to solve the problem $\mathcal{J}\bar{\boldsymbol{X}} = \boldsymbol{B}$. As a consequence, these techniques do not necessitate the explicit construction and storage of $\mathcal{J}$ in the system memory, but only require the ability to compute its effect on a vector. However, efficient convergence of the iterative sequence is dependent on certain properties of the matrix such as its diagonal dominance and spectral radius [17].

Recently, research focus turned to the direct LU factorization of $\mathcal{J}$. The success in these approaches is premised on the system's main memory being sufficiently large to permit the storage of $\mathcal{J}$. The first work in this regard appeared in [5], where a block form of regular LU factorization was utilized to perform the LU factorization of the Jacobian arising from Harmonic Balance circuit simulation. Following this idea, a block-based approach was built on top the KLU framework [7] (BKLU) to solve the system of equations that arise from the High-order Obreshkov-based transient simulation [18], showing a significant reduction in the number of fill-ins that result from applying KLU. One must note, however, that all of these works were strictly CPU-based computations.

With the power of the modern GPU's massively parallel architecture, GPU-based LU factorization of $\mathcal{J}$ was recently proposed in [6], where it was demonstrated that speedups close to two-orders-of-magnitude, compared with CPU-based BKLU can be achieved in the Harmonic Balance steady-state analysis. In that same vein, the work presented in this thesis aims to introduce the ability to factorize $\mathcal{J}$ using multiple OpenCL accelerators, such as multiple GPUs. To that end, there are several existing approaches which identify the inherent parallelism between the columns of a given matrix. Firstly, [19] introduced a parallel scheduler for shared-memory systems with multi-core CPUs, based on the KLU algorithm. This was followed by a parallel adaptive sparse matrix solver in [20] and further enhanced for GPU architectures in [21]. One must note that the GPU variant in [21] suffers from the inability to exploit multiple GPUs, it is strictly limited to a single GPU. Table 2.2 shows a comparison between the previous works mentioned.

The main focus of most previous literature is in solving extremely sparse scalar matrices. The unique nature of the structure of the Harmonic Balance and Polynomial Chaos Jacobian Matrices presents a challenge for sparse solvers to tackle as the matrices are characterized by very sparse patterns of dense blocks, which is the main focus of this work. This type of matrix allows for two levels of parallelism -

Existing Solution	Block-aware	Platform	Multi-threaded
NICSLU [20]	No	CPU	Yes
NICSLU GPU [21]	No	GPU	No
KLU [7]	No	CPU	No
BKLU [18]	Yes	CPU	No
BKLU-GPU [6]	Yes	CPU/GPU	No
This work	Yes	GPU	Yes

Table 2.2: Comparison of previous work

parallelism inherent to dense block operations such as block-block multiplication and the parallelism between the columns of the Jacobian Matrix, with the latter being the novel concept presented throughout the remainder of this thesis.

2.9 Summary

In this chapter, the background and related work was presented which gives the reader an understanding for the basis on which this thesis is formed. The chapters that follow rely on the concepts presented on OpenCL, block-structure matrices, and GPU architecture in general.

Chapter 3

Parallelism in Matrix Structure

3.1 Introduction

This chapter introduces the classical KLU approach to matrix factorization and the block aware augmentation of KLU which is pivotal to the work developed in this thesis. Later sections explore achieving parallelism in matrix factorization by exploiting the structure of sparse matrices and developing a schedule for the parallel execution.

3.2 Classical KLU Factorization

The KLU factorization algorithm [7] solves sparse linear systems in the form of the following equation

$$\boldsymbol{J}\boldsymbol{x} = \boldsymbol{b} \tag{3.1}$$

where $\boldsymbol{x}$ is unknown, by factorizing $\boldsymbol{J}$ into its LU factors such that the following is true.

$$\boldsymbol{J} = \boldsymbol{L}\boldsymbol{U} \tag{3.2}$$

To determine the LU factors of $\boldsymbol{J}$, KLU uses the left-looking Gilbert-Peierls (G-P) algorithm [22] which

computes $\boldsymbol{L}$ and $\boldsymbol{U}$ one column at a time by repeatedly solving the lower triangular system

$$\boldsymbol{L}\boldsymbol{y} = \boldsymbol{J}(:,k) \tag{3.3}$$

for $k = 1$ to N. The basic algorithm (Algorithm 1) starts with setting $\boldsymbol{L}$ equal to the identity matrix $\boldsymbol{L} = \boldsymbol{I}$ and at each step proceeds to solve Equation (3.3) updating $\boldsymbol{L}$ and $\boldsymbol{U}$ with the column contained in the solution vector $\boldsymbol{y}$.

Solving the lower triangular system (Algorithm 2) involves finding all of the already factorized columns to the left. This is typically done by constructing a directed graph of $\boldsymbol{L}$, G_L, representing the non-zero entries of $\boldsymbol{L}$. G_L has an edge $j \rightarrow i$ if and only if $\boldsymbol{L}(i,j) \neq 0$. $Reachables_{G_L}(i)$ is the set of nodes reached by performing a depth-first search starting with node i.

Algorithm 1: Basic KLU Gilbert-Peierls Algorithm

```
   Input  : A ∈ R^{N×N}
   Output: L, U ∈ R^{N×N}
 1 begin
 2 |  L ← I;
 3 |  for p ← 1 to N do
 4 |  |  x = A(:, p);
 5 |  |  y = FS(L, x);
 6 |  |  k = max_i |y(i)|;
 7 |  |  Do Partial Pivoting by interchanging rows k and p;
 8 |  |  U(1 : p, p) ← y(1 : p);
 9 |  |  L(p, N : p) ← y(p : N)/U(p, p);
10 |  end
11 end
```

Algorithm 2: FS: Forward Solve $Ly = x$

```
  Input  : L ∈ R^{N×N}, x ∈ R^N
  Output: y ∈ R^N
1 begin
2 |  Y ← Reachables_{G_L}(x); // Compute the reachability set for all non-zero entries in x
  |   y = x;
3 |  for p ∈ Y do
4 |  |  y(p + 1 : N) ← y(p + 1 : N) − L(p + 1 : N, p) × y(p);
5 |  end
6 end
```

The full KLU algorithm also involves techniques aimed at exploiting matrix sparsity and enhancing the numerical conditioning. These additional steps are summarized by the following three main points:

1. Ordering the columns and rows of $\boldsymbol{J}$ to reduce the fill-ins[1] using one of the ordering approaches

[1]Fill-ins refer to entries that are structurally zero in $\boldsymbol{J}$, but are replaced with nonzero entries in the resulting $\boldsymbol{L}$ or $\boldsymbol{U}$.

such as the Approximate Minimum Ordering (AMD) [23], COLAMD, or METIS ordering schemes.

2. Scaling the rows of $\boldsymbol{J}$ 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.

3. Partial pivoting, which refers to the process of permuting the rows of $\boldsymbol{J}$ to avoid dividing by very small diagonal elements.

The first step takes place in the symbolic factorization phase with the final two steps being performed in the numerical factorization stage.

Finally, with the LU factors obtained, forward and backward substitution are used to solve for the unknown vector $\boldsymbol{x}$ presented in Equation (3.1).

3.3 Block-Aware LU Factorization

Introduced in [18] is a CPU-based block-aware extension to KLU termed as BKLU. BKLU is based on the fact that the structural non-zero pattern of the MNA matrix $\boldsymbol{J}$ can be used to locate non-zero block entries of the augmented block structured Jacobian matrix $\boldsymbol{\mathcal{J}}$. A test matrix $\boldsymbol{J}$ is used in the classical KLU symbolic and numerical phases with the objective of obtaining 1) the row/column ordering used to reduce fill-ins, 2) the set of scaling factors, and 3) the row permutations for choosing appropriate pivots. Based on the information provided by the test matrix $\boldsymbol{J}$, the non-zero structure of both $\boldsymbol{L}$ and $\boldsymbol{U}$ become known prior to factorizing the block-structured matrix $\boldsymbol{\mathcal{J}}$.

BKLU replaces the scalar operations in classical KLU (Algorithm 1 & Algorithm 2) with matrix operations each of size $M \times M$. For example, the scalar multiplication and subtraction operation in line 4 of Algorithm 2 is replaced by a matrix-matrix multiplication and matrix-matrix subtraction. This approach significantly reduces the number of fill-ins that result from classical KLU factorization while also allowing for the potential of parallel acceleration. This approach of block-aware matrix factorization was realized in the previous work [6] by implementing BKLU on a single GPU to take advantage of the parallelism inherent in matrix-matrix operations.

3.4 Identifying Column Level Dependencies

As discussed in Section 3.2, the G-P algorithm computes the LU factors of the input matrix $\boldsymbol{J}$ on a column-by-column basis. As indicated by the forward solve algorithm, the computation of a given column k can depend on a previously factorized column j of $\boldsymbol{L}$, where $j < k$, which is to the left of k (see Figure 3.1). The column level dependencies are identified by examining the sparsity pattern (the non-zero structure) of the $\boldsymbol{U}$ factor, which is determined prior to factorizing the block structure matrix $\mathcal{J}$. The column level dependencies are determined such that column k depends on column j if $\boldsymbol{U}(j,k) \neq 0$

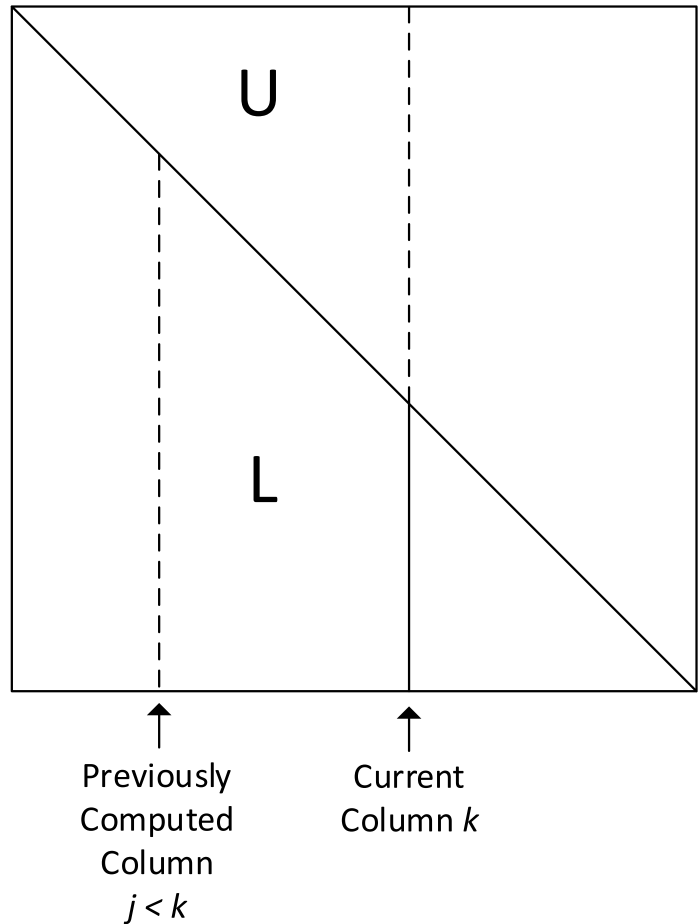
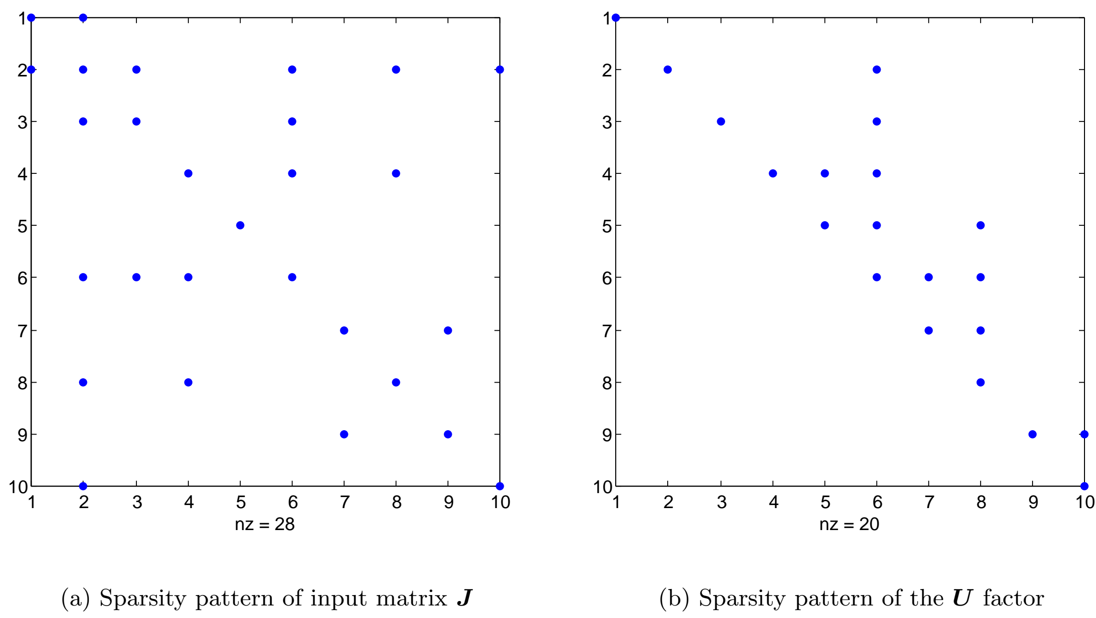


Figure 3.1: Gilbert-Peierls Column Level LU factorization

for $j < k$. Applying this constraint to the structure of $\boldsymbol{U}$, a Directed Acyclic Graph (DAG) is typically constructed to represent the column level dependencies. Nodes of the DAG represent columns of the $\boldsymbol{L}$ factor while edges represent data dependence between the columns. Two nodes, say i and j, are connected by an edge directed from i to j, $i \rightarrow j$, if column j depends on column i in the Forward Solve

step (Algorithm 2) of the G-P factorization. In this case, node i is labeled as a child node while node j is labeled the parent node. Childless nodes are termed as leaf nodes.

The sparsity pattern of an example matrix $\boldsymbol{J}$ with size 10×10 is shown in Figure 3.2a. The sparsity pattern of $\boldsymbol{U}$ corresponding to $\boldsymbol{J}$ is given in Figure 3.2b. The column level dependencies for a column k are identified by examining the preceding columns where $U(1 : k-1, k) \neq 0$. For example, column $k = 8$ depends on column $j = 7, 6, 5$ as there are non-zero entries in rows 5, 6, 7 for each column j and k. After analyzing all of the columns, the DAG in Figure 3.3 is constructed. From this DAG it can be seen that columns $k = 1, 2, 3, 4, 9$ have no dependencies to their left while the remaining columns $k = 5, 6, 7, 8, 10$ depend on one or more other columns.



(a) Sparsity pattern of input matrix $\boldsymbol{J}$

(b) Sparsity pattern of the $\boldsymbol{U}$ factor

Figure 3.2: Sparsity pattern of example matrices

3.5 Scheduling Parallel Factorization

The main goal of the work presented in this thesis is to allow the factorization of multiple block columns in parallel on multiple OpenCL devices. To that end, each node in the DAG corresponds to a unique task of both the Forward Solve operation (Algorithm 2) and the pivot division operations (Line 9 of Algorithm 1) for the corresponding column. The parallelism between those tasks is identified by analyzing the DAG of the matrix $\boldsymbol{J}$, where, similar to the NICSLU approach [19], each task (node k) is assigned

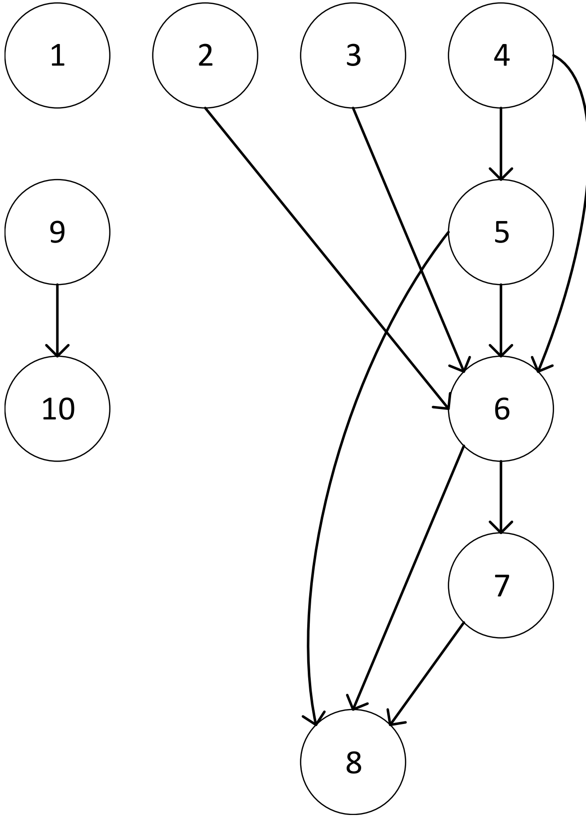


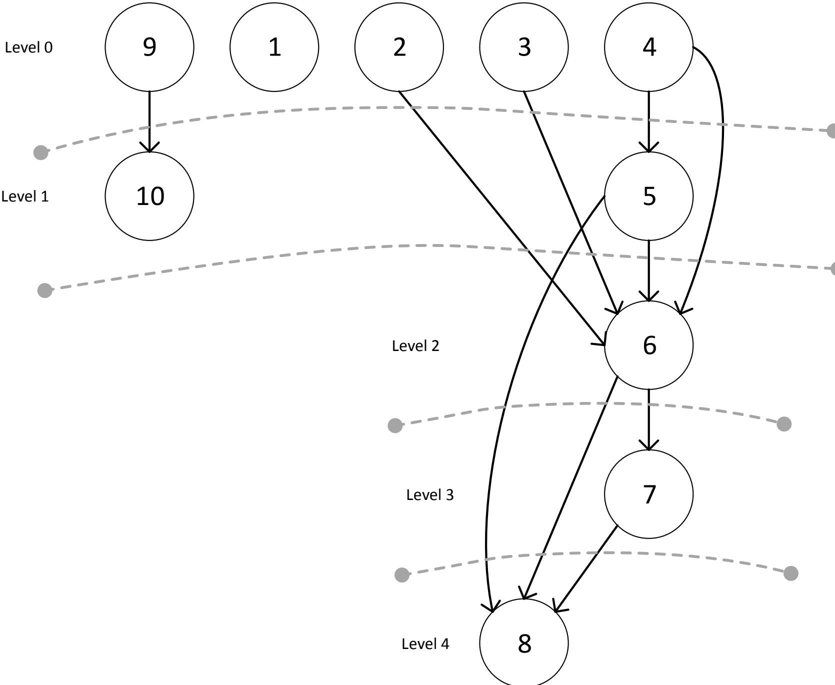
Figure 3.3: Dependency graph based on $\boldsymbol{U}$ factor

a numerical value denoted as $level(k)$,

$$level(k) = max(-1, level(c_1), level(c_2), ...) + 1 \tag{3.4}$$

where $c_1, c_2, ...$ are the children of node k. Figure 3.4a shows the levels assigned to the nodes of the example matrix $\boldsymbol{J}$ used in the example presented in the previous section.

Columns which have the same value $level(k)$ are independent and therefore, their corresponding tasks can be assigned to different devices. Independent columns can be factorized in parallel without the need for inter-device communication or synchronization during the Forward Solve and division operations. A level l which has a single column k, typically depends on the result of a task in the previous level $l-1$. Levels with only a single task are factorized sequentially in a pipeline like fashion, where the column in level l is updated by dependencies which have been previously been computed while the dependent column from level $l-1$ is being computed concurrently. As can be seen in the example problem, columns $k = 1, 2, 3, 4$ and 9 are independent and thus belong to the same level while columns $k = 6, 7$ and 8 belong to single column levels, each depending on the result of a previous level.



(a) Levels assignment in a DAG

Level	Column
0	1 2 3 4 9
1	5 10
2	6
3	7
4	8

Device 1

Device 2

(b) Device task assignment based on the level value

Figure 3.4: Using the DAG to extract the parallelism

Each column belonging to the same level are further separated by which device will perform the computation. For block structured matrices, it is ideal for the workload to be split evenly between each accelerator in the system. Figure 3.4b shows this separation for the graph in 3.4a using two accelerators.

As an example of single level factorization, suppose that at the start of level 4, $Device2$ was not yet finished computing column $k = 7$. Instead of stalling and waiting on the completion of k, $Device1$ will proceed to update column $k = 8$ with the already computed dependencies $j = 6, 5$. Once the column has been updated with all of the available dependencies, $Device1$ will wait for completion of column 7 from $Device2$.

3.6 Summary

In summary, this chapter introduced the classical KLU left-looking LU factorization along with the block aware variant. Further, the concept of data dependence between the columns of a matrix was introduced along with a strategy for scheduling parallel execution of the LU factorization algorithm based on this data dependence.

Chapter 4

Block KLU with Multiple OpenCL Accelerators

4.1 Introduction

This chapter details the implementation of the proposed Block KLU capable of using multiple OpenCL acceleration devices on a single host. For simplicity, the proposed method is referred to by the acronym BKLUX throughout this chapter and the remainder of the thesis. The details covered in this chapter are applicable to any OpenCL device (such as CPU, GPU, and FPGA), however, the focus of the work done in this thesis has been on a system with two AMD GCN GPUs.

When using multiple discrete accelerators, there are inherent issues associated with synchronization and memory coherence. Many aspects of the BKLUX implementation have been carefully designed by minimizing inter-device communication and avoiding costly inter-device synchronization operations. Additionally, it is important to ensure the bandwidth of the PCI bus is efficiently used by performing only necessary communication between the host and attached devices. BKLUX is designed to scale while additional devices are added to the system and thus efficiency of communication and synchronization will be a recurring theme throughout this chapter and again in Chapter 5.

4.2 Device Context Separation

BKLUX makes no assumptions about the memory resources available on a given OpenCL device or the type of device. Separate contexts allow devices from different vendors, each with their own OpenCL implementation to work in harmony within BKLUX. As such, each device is associated with its own context ensuring that each device maintains its own independent memory buffers and command queues. This approach allows the memory system to be treated as distributed, allowing for non-uniform memory sizes between devices in the system. Devices such as FGPAs with limited discrete memory, possibly ranging in the hundreds of megabytes, can be paired with a GPU which has memory ranging in the gigabytes. As a result of separate contexts, explicit transfers are required to synchronize data between devices. Chapter 5 will cover in detail how explicit transfers can be used to reduce the amount of memory used on each device and to reduce the strain on the system bus by minimizing communication.

4.3 Memory Layout

4.3.1 Block Compressed Column Storage

The block structure matrix $\mathcal{J}$ and its LU factors are stored in Block Compressed Column Storage (BCCS) format. BCCS is an extension of Compressed Column Storage format which is used almost universally to represent sparse matrices in computer systems as described in [24]. BCCS maintains three arrays for each matrix A_p, A_i and A_x. To better explain BCCS, let $M \times M$ be the size of each block in $\mathcal{J}$, $nnzb$ be the number of non-zero blocks in $\mathcal{J}$, and n be the scalar dimension of $\mathcal{J}$, i.e. $n = N/M$. BCCS is illustrated on an example matrix in Figure 4.1 where each shaded block is of size $M \times M$. The arrays A_p, A_i and A_x are then defined by,

- A_x is an array of memory pointers to the blocks of $\mathcal{J}$ which are dense floating-point blocks of size $M \times M \times$ `sizeof(double)` bytes.
- A_i of length $nnzb$ which stores the row indices of all non-zero blocks of $\mathcal{J}$. In the example $nnzb = 10$.
- A_p of length $n + 1$, stores the index of A_i which holds the first non-zero block of each column.

In the example above, the blocks of column k are stored in $A_x(A_p(k))$, $A_x(A_p(k){+}1)$, ... $A_x(A_p(k{+}1){-}1)$.

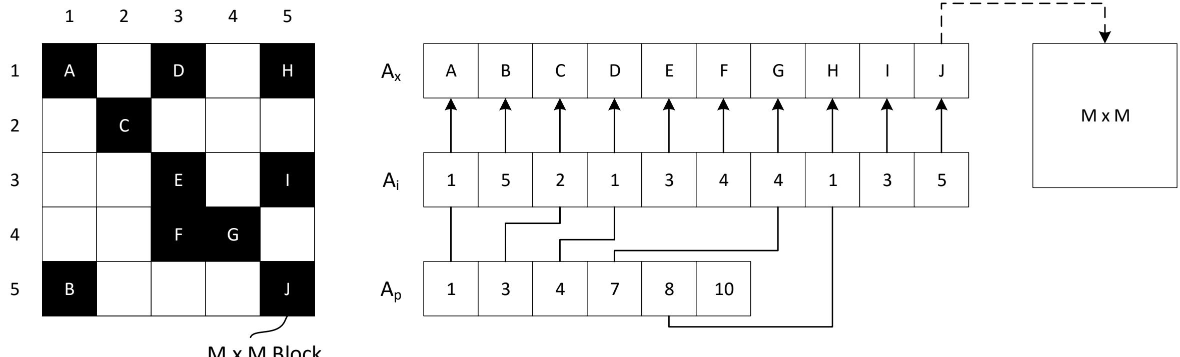


Figure 4.1: Example of Block Compressed Column Storage format

The row indices of each block in column k are $A_i(A_p(k)), A_i(A_p(k)+1), ..., A_i(A_p(k+1)-1)$. Note that contrary to the above example, a typical implementation would use zero-based arrays.

4.3.2 Device Memory

The global memory of each OpenCL device is divided into four groups where each group is comprised of one or more buffers. As shown in Figure 4.2, the first three groups organize the blocks of $\boldsymbol{\mathcal{J}}$ and its LU factors. The fourth buffer is a dedicated workspace where intermediate work is performed. Each buffer within the groups stores a single block of size $M \times M$, consuming `sizeof(double)` $\times M^2$ bytes of device memory. An additional single buffer of size $M \times M$ is also reserved in device memory to be used as temporary storage for a single block. The number of buffers in each of the four groups is determined by several factors, such as the number of devices cooperating in the factorization and the memory management modes used by the algorithm, which are detailed in Chapter 5. For now, it suffices to say that the groups are defined as follows:

- The $\boldsymbol{J}$ group ranges from 0 to J_{nnz} blocks, where J_{nnz} is the number of non-zero blocks in $\boldsymbol{\mathcal{J}}$;
- The $\boldsymbol{U}$ group ranges from 1 to U_{nnz} blocks, where U_{nnz} is the number of non-zero blocks in $\boldsymbol{U}$;
- The $\boldsymbol{L}$ group ranges from L_{max} to L_{nnz} blocks, where L_{max} is the greatest number of non-zero blocks in any column of $\boldsymbol{L}$ and L_{nnz} is the number of non-zero blocks in $\boldsymbol{L}$;
- The workspace, $\boldsymbol{X}$, ranges between 2 and LU_{max} blocks, where LU_{max} is the greatest number of non-zero blocks in any column of the combined $\boldsymbol{LU}$ matrix;

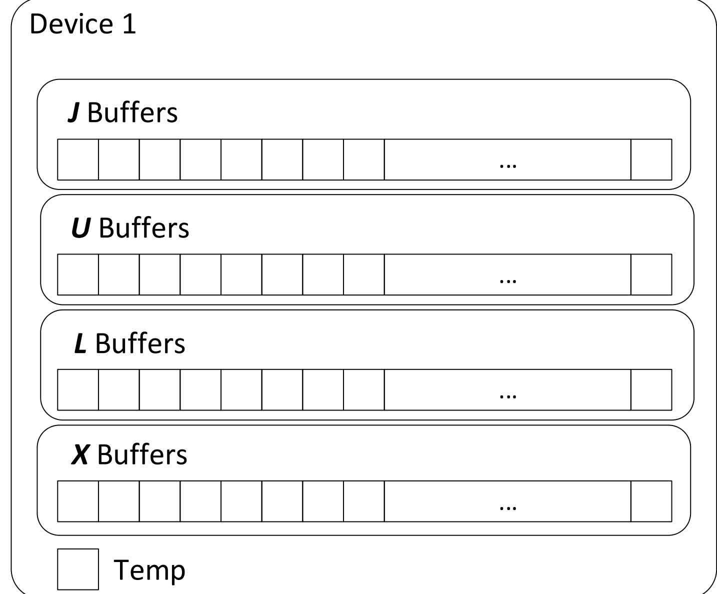


Figure 4.2: OpenCL device memory buffers for example Device 1

- The temporary block is always 1 block of size M^2.

4.3.3 Host Memory

The main memory organization of the host is similar to the accelerator organization. Three buffers are dedicated to storing the $\boldsymbol{\mathcal{J}}$ matrix and its LU factors, however unlike the accelerator, these three buffers are memory mode independent with sizes determined by the non-zero pattern of the matrix $\boldsymbol{\mathcal{J}}$ and its LU factors. Each accelerator device is controlled by a dedicated thread, with each of these threads maintaining its own workspace buffer of N blocks. The buffers are summarized as follows.

- $\boldsymbol{\mathcal{J}}$ matrix: Holds the full input matrix $\boldsymbol{\mathcal{J}}$. The size of this memory object is the number of non-zero blocks in $\boldsymbol{\mathcal{J}}$.

- $\boldsymbol{L}$ Factor: Holds the full L Factor. The size is the number of non-zero blocks in $\boldsymbol{L}$.

- $\boldsymbol{U}$ Factor: Holds the full U Factor. The size is the number of non-zero blocks in $\boldsymbol{U}$.

- Workspace (X): Each thread has its own workspace, X_i, where i is the thread index. The host-side workspaces each hold N blocks, where N is the dimension of $\boldsymbol{J}$.

The host memory stores the BCCS integer arrays that describe the structure of the $\boldsymbol{\mathcal{J}}$ and its LU factors.

Those are (collectively) denoted by $\{\boldsymbol{J}, \boldsymbol{L}, \boldsymbol{U}\}_{\{p,i\}}$. Each block in $\boldsymbol{J}$, $\boldsymbol{L}$, $\boldsymbol{U}$, and $\boldsymbol{X}$ are instances of a C++ class to which each member of the storage arrays $\{\boldsymbol{J}, \boldsymbol{L}, \boldsymbol{U}\}_{\{x\}}$ points to.

The host also stores two boolean arrays, the first being C_f of size N which is shared between all threads. It indicates if a column has been completed, transfered from the device, and is present in the host memory. The second array, C_p, is a set of N-sized arrays, one per each accelerator device, indicating if a column is currently present in that device's global memory. The arrays are summarized as follows.

- Column Finished (C_f): A single set shared between all threads which holds the column indices of the LU factor which have been computed and synchronized back to CPU memory.
- Column Present (C_p): Each OpenCL accelerator maintains a set which contains the column indices of the L factor which are currently in that device's memory.

Since tasks are divided among devices, and no two devices collaborate on the same task, no two threads ever simultaneously write to a single region of shared memory, thus eliminating the need to employ costly memory access protocols. However, OpenCL's host side kernel constructs are not thread safe and consequently requiring each device to have its own instances of the compiled kernels, even if they have the same underlying architecture.

4.4 Description of Execution

Each OpenCL device in the system is assigned a CPU thread responsible for issuing all OpenCL commands to that device. Additionally, each device is assigned two command queues, $Q1$ for issuing kernel launch commands and $Q2$ for asynchronous GPU-to-host memory transfers. The procedure outlined in Algorithm 3 describes the main steps that a thread goes through during the computation of the pth column of a matrix $\mathcal{J}$. The following is a more detailed description of the main tasks performed in the execution path of Algorithm 3.

1. **Data Transfer**
   (a) If the blocks of column p are not present in the device $\mathcal{J}$ buffer, the blocks of column p are transfered from the host $\mathcal{J}$ buffer to the device.
2. **Copy-Scale**

Algorithm 3: Basic host algorithm for the factorization of column p

```
1  begin
2     for each block in p do
3        Transfer block from the host to device 𝓙 buffer;
4        Scale and move the block to the workspace buffer 𝓧;
5     end
6     for each dependency q of p do
7        if column q was assigned to another device and is not finished then
8           Wait until q is factorized and in host memory;
9        end
10       if column q is not in device memory then
11          Transfer column q from host memory to the device buffer 𝓛;
12       end
13       Update the blocks of p by q;
14    end
15    for blocks 1 to p of column p do
16       Save the block to the device 𝓤 buffer;
17       Transfer block back to the host;
18    end
19    Compute LU factorization of pivot block;
20    for blocks p + 1 to N of column p do
21       Solve the division by the pivot block saving the result to the 𝓛 buffer;
22       Synchronize the block with the host;
23    end
24 end
```

(a) All blocks of the pth column of $\boldsymbol{\mathcal{J}}$ are copied from the device $\boldsymbol{\mathcal{J}}$ buffer, multiplied by the block's scaling factors and then saved to the workspace buffer $\boldsymbol{\mathcal{X}}$. Scaling factors are determined during the second step of the regular KLU which is performed on the test matrix $\boldsymbol{J}$.

3. **Multiply-Subtract-Accumulate**

(a) The blocks of the pth column are updated by columns to the left with a multiply-subtract-accumulate operation using level 3 BLAS GEMM from AMD's clBLAS library [25] which is used to compute $C \leftarrow \alpha AB + \beta C$ operation, with $\alpha = -1$ and $\beta = 1$. A single kernel is launched for each block-wise update to column p.

(b) During step (a), it is possible for a column which p depends on to be absent from device memory. If this is the case, as can be detected by the C_p array, the required column is queued for transfer from the host to the device. The transfer is enqueued to the kernel launch command queue, $Q1$, to preserve the order of operations without requiring the explicit synchronization.

(c) If the above dependent column is found to be undergoing processing by another device, as

tracked by the C_f array, the issuing of kernels is stalled until the column has been transfered to the host by the other device. Note that while the issuing of commands to the device has been stalled, the device is free to continue working on previously issued kernels. Once the transfer to the host is complete, the column is queued for transfer to the device and issuing of kernels resumes.

4. **$\boldsymbol{\mathcal{U}}$ Block Copy and Transfer**

   (a) Copy non-zero blocks in the workspace $\boldsymbol{\mathcal{X}}$ between 1 and p to the blocks of the pth column of the $\boldsymbol{\mathcal{U}}$ buffer. One kernel is launched per block.

   (b) Enqueue all the blocks of pth column of $\boldsymbol{\mathcal{U}}$ for asynchronous transfer back to the host, one kernel is launched per block using queue $Q2$.

5. **Pivot Division and $\boldsymbol{\mathcal{L}}$ Block Transfer**

   (a) The pivot block is decomposed into its LU factors using the method discussed in [6].

   (b) The triangular system of equations solver (TRSM) from AMD's clBLAS [25] is used to obtain the division of $\boldsymbol{\mathcal{L}}$ blocks with the pivot block for all non-zero blocks of $\boldsymbol{\mathcal{X}}$ between $p+1$ and N. The result is saved in the corresponding blocks of the pth column of the $\boldsymbol{\mathcal{L}}$ buffer in the same kernel.

   (c) Enqueue the pth column of $\boldsymbol{\mathcal{L}}$ for asynchronous transfer back to the host using queue $Q2$.

6. **Proceed to the next scheduled column.**

4.5 General Improvement Strategies

4.5.1 Introduction

Parallel programming and OpenCL programming in general are fraught with difficulties and areas for optimization. In the proposed BKLUX, several optimization were done to improve the pitfalls of parallel execution. Firstly, communication overhead plays a large part in determining if an algorithm is suitable for acceleration on discrete devices. Asynchronous communication has an important role in BKLUX, reducing the latency and synchronization time of data transfers between devices. The approach adopted

in BKLUX is described in Section 4.5.2. Further, the action of synchronization between devices and the host can have a significant impact on performance without carefully choosing synchronization methods, as described in Section 4.5.3. Additionally, two last areas of improvement are covered in Section 4.5.4 and Section 4.5.5, dealing with strategies for splitting tasks among the many devices in the system and ordering operations such to reduce the impact of blocking synchronization conditions.

4.5.2 Asynchronous Communication

Many vendor-provided OpenCL implementations allow for the overlapping of kernel execution and host-device communication. For example with a GPU supporting AMD's OpenCL implementation, the use of two command queues on a single device allows such overlap of computation and communication [16]. All device-host transfers can be performed using a data-transfer only command queue used only for communication. Figure 4.3 shows that by using the second data-transfer only queue for each AMD GPU, the execution time of the factorization of the Rajat05 matrix with a block size of 512×512 is improved by 12.1%.

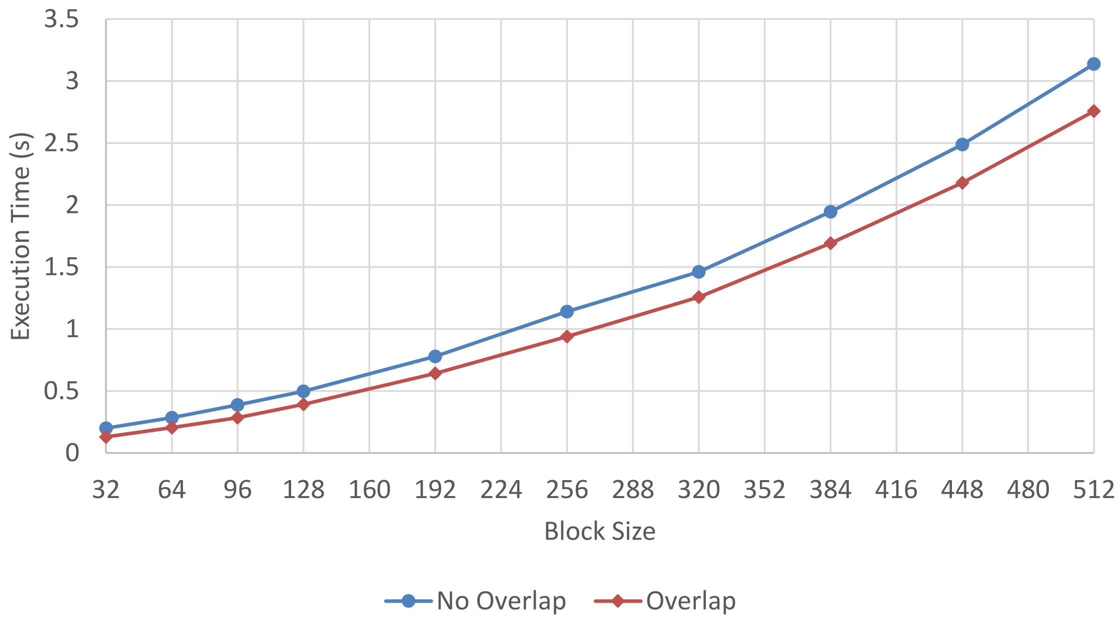


Figure 4.3: Communication Overlap for the Rajat05 matrix from The University of Florida Sparse Matrix Collection [1] with block size of 512×512

4.5.3 Synchronization

Limiting the impact of synchronization in the multi-device BLKUX is paramount to its performance. The proposed BKLUX implementation is designed such that blocking OpenCL API calls on the host are avoided whenever possible. API calls such as $clFinished()$ and $clWaitForEvents()$ are examples of blocking OpenCL API calls which wait for the device to finish work before releasing the calling thread. Independent of any memory optimization techniques, the BKLUX implementation has only one condition which causes a blocked state, which is when a device has a dependency that has not yet been transfered to the host memory from another device. In such a case, the host thread must wait before issuing the memory transfer command. To avoid expensive OpenCL synchronization calls when this condition arises, the host simply refrains from issuing any additional commands or kernels to the blocked device until the blocking condition is resolved. The device remains free to continue processing any outstanding work that was submitted prior. The blocking condition is resolved when the last piece of the column dependency is transfered to the host memory, as indicated by the C_f array. The transfer of the last block of a column is associated with an OpenCL event whose callback function sets the entry in C_f corresponding to the column as finished. With the host memory updated, the blocked device is free to issue the command to transfer the now finished column from the host memory and resume factorization.

By ensuring kernels are queued in bulk, this allows dependency graphs to be constructed for the various tasks thus the device execution and resource allocation can be scheduled in advance. Advanced scheduling reduces the latency between submitting a task to the OpenCL device and the actual start of execution.

4.5.4 Task Splitting

Recall that a task represents the factorization of a single column and that tasks belonging to the same *level* are independent. For levels with multiple tasks, there are various permutations for the order and division of the tasks among the devices in the system. In the examples discussed thus far, the factorization tasks are divided in a linearly chunked fashion. As an example, a given level l has columns, say, $k = 1, 2, 3, 4, 10, 13, 15, 21$ which would have their tasks divided between two devices by splitting them in half and retaining the linear ordering such that $device1$ is assigned tasks 1,2,3,4 and $device2$ has 10,13,15,21.

The choice of task division scheme can ultimately impact the overall performance of the factorization.

Some choices of task division can introduce memory transfers that otherwise would not occur if the tasks were partitioned and ordered differently. For example, if a task A is assigned to *device*1, however *device*2 holds seven of task A's dependencies, the choice of device assignment will introduce seven additional memory transfers that otherwise would not have been required of task A was assigned to *device*2. This can also lead to the creation blocking conditions should task A rely on tasks which remain unfinished on another device.

Another scheme investigated was task division which considers how many of a given task's dependencies each device currently holds in its memory. Tasks are assigned to the device which has the greatest number of dependencies present in memory. The objective of this method aims to reduce the number of memory transfers and the possibility of blocking conditions. However it is not a trivial exercise to determine the optimal path which will yield the least number of memory transfers. The BKLUX implementation described in this thesis uses the simple linear chunk method for dividing tasks. This thesis leaves optimal tasks division as a subject for future investigation.

4.5.5 Order of Operations

The forward solve operation, summarized by the for loop on line 6 in Algorithm 3 describes the process of updating the blocks of a column p. In the algorithm, a dependency q which has not yet been transfered to the host memory causes a blocked state such that no additional commands are issued to the device. Only after the host memory has been synchronized by the device responsible for factorizing q can the memory transfer command be issued. When using OpenCL accelerators, this is an undesirable state as discussed above. It is advantageous to issue as many commands as possible to the OpenCL runtime hiding the latency involved in scheduling. To mitigate the affects of blocking conditions, the loop is modified as shown in Algorithm 4 to defer updates by any dependencies which will result in blocking conditions until after all other eligible commands are issued.

In Algorithm 4 the set Y contains all the L columns which update column p. Once a column $q \in Y$ has successfully updated p, it is removed from Y. This is repeated done by iterating through $Y|$ until $|Y| = 0$. While iterating, any dependency which isn't available in host memory (i.e. it is still undergoing processing on another device) is skipped and remains in Y. This allows any other columns in Y, which are either present in host memory or already in device memory, to update p without being unnecessarily delayed.

Algorithm 4: Modified column update order

```
1  begin
2      while |Y| > 0 do
3          for p ∈ Y do
4              if column q was assigned to another device and is not finished then
5                  continue;
6              end
7              if column q is not in device memory then
8                  Transfer column q from host memory;
9              end
10             Update the blocks of p by q;
11             Remove q from Y;
12         end
13     end
14 end
```

4.6 Summary

In this chapter, the implementation of the block-aware KLU with multiple OpenCL devices (BKLUX) was presented. It was shown how the design and implementation of BKLUX allows multiple OpenCL devices to collaborate in the factorization of block-structured matrices.

Chapter 5

Memory Optimizations

5.1 Introduction

Accelerators such as GPUs have a finite amount of typically non-expandable discrete memory. Using these memory resources optimally plays a central role in leveraging the performance of OpenCL devices. In a single accelerator setting, such as a single GPU as presented in [6], the ideal choice is to have all of the matrices ($\mathcal{J}$, $\mathcal{L}$, $\mathcal{U}$ and the workspace $\mathcal{X}$) stored in the device memory throughout the factorization. However, it becomes challenging to accommodate matrices with large dimensions or large block sizes, quickly exceed the device's memory capacity. These matrices can range from several gigabytes to terabytes in size. The implementation of [6] required that, at least, the columns of the $\mathcal{L}$ factor and a full array of blocks used for the workspace $\mathcal{X}$, be stored on the GPU throughout the factorization process.

In the multi-OpenCL device paradigm proposed in this thesis, this mode of storage is no longer the best choice. This is due to the fact that each device will be tasked with a disjoint set of columns to factorize, which depend on different previous sets of columns of the $\mathcal{L}$ factor in the update process. Therefore, storing the whole $\mathcal{L}$ factor on each device, even though it may not be required in the update process, represents an unnecessary use of resources. To eliminate the requirement of storing the entire $\mathcal{L}$ factor on each device, the following subsections propose memory management modes based on the size of the matrix. The memory modes are built on the memory system required for handling multiple devices and allow for larger matrices and higher block sizes to be factorized on limited memory devices, while still

achieving significant speedup versus CPU implementations.

A noteworthy remark is that, although the proposed memory management approaches target the multi-device setting, they can be used in a single device platform. In fact, the techniques presented below eliminate the assumption that the GPU memory must be large enough to store the blocks of the $\mathcal{L}$ factor and a full workspace array of blocks throughout the factorization. This widens the scope of problem sizes that can be addressed through the GPU-based BKLU direct factorization approach from [6].

Figure 5.1 details the memory usage of various benchmark matrices from [1] without any modification to their memory usage. This example will be used throughout the chapter to show the effectiveness of the various memory management methods.

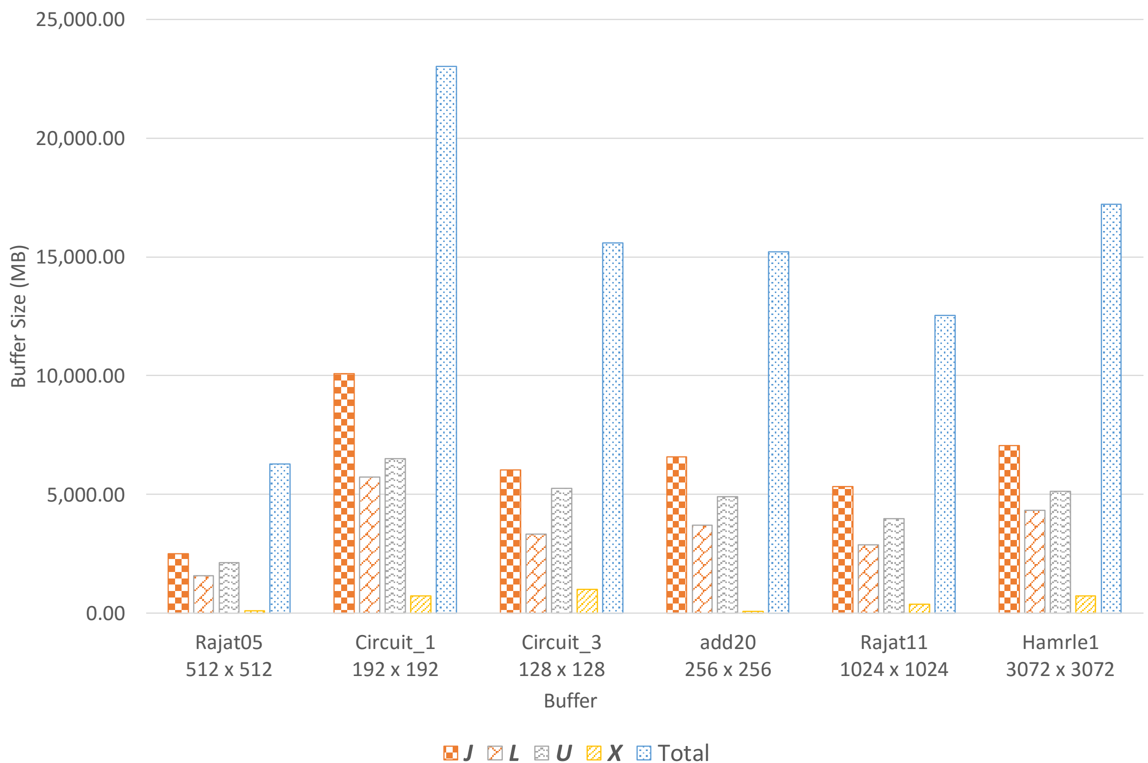


Figure 5.1: Device memory usage of various benchmark matrices from [1] without optimization

5.2 Device Memory Definition

Let $M_{i,total}$ represent the total amount of memory on a device i in the system. The amount of free memory after each buffer allocation is given by

$$M_{i,free} = M_{i,total} - (\boldsymbol{\mathcal{J}}_i + \boldsymbol{\mathcal{L}}_i + \boldsymbol{\mathcal{U}}_i + \boldsymbol{\mathcal{X}}_i + Temp_i) \times \texttt{sizeof(double)} \times M^2 \tag{5.1}$$

where sizeof(double) is the size in bytes of a double-precision floating-point on the device architecture, M is the block size, $\boldsymbol{\mathcal{J}}_i$ is the number of blocks in the $\boldsymbol{\mathcal{J}}$ buffer, $\boldsymbol{\mathcal{L}}_i$ is the number of blocks in the $\boldsymbol{\mathcal{L}}$ buffer, $\boldsymbol{\mathcal{U}}_i$ is the number of blocks in the $\boldsymbol{\mathcal{U}}$ buffer, $\boldsymbol{\mathcal{X}}_i$ is the number of blocks in the workspace buffer $\boldsymbol{\mathcal{X}}$, $Temp_i$ is the number of blocks in the temporary buffer. Since the temporary buffer always holds exactly one block, $Temp_i$ can be replaced by 1.

It is more convenient to consider memory quantities based on the number of blocks which can be stored within a given buffer. To that end, let $M_{i,thresh}$ represent the device memory threshold such that

$$M_{i,thresh} = floor(\frac{\alpha M_{i,total}}{\texttt{sizeof(double)} \times M^2}) \tag{5.2}$$

where $0 < \alpha < 1$ indicates how much of the device memory to make available for use. Finally, Equation (5.3) must be satisfied at all times throughout the factorization.

$$M_{i,thresh} \geq \boldsymbol{\mathcal{J}}_i + \boldsymbol{\mathcal{L}}_i + \boldsymbol{\mathcal{U}}_i + \boldsymbol{\mathcal{X}}_i + 1 \tag{5.3}$$

where each buffer $\{\boldsymbol{\mathcal{J}}, \boldsymbol{\mathcal{L}}, \boldsymbol{\mathcal{U}}, \boldsymbol{\mathcal{X}}\}_i$ is an integer representing the number of $M \times M$ blocks stored within. The block notation will be used throughout the remainder of this chapter. Conversion back to Bytes is $\{\boldsymbol{\mathcal{J}}, \boldsymbol{\mathcal{L}}, \boldsymbol{\mathcal{U}}, \boldsymbol{\mathcal{X}}\}_i \times \texttt{sizeof(double)} \times M^2$.

5.3 Memory Optimizations for the $\boldsymbol{\mathcal{J}}$ Matrix

A significant portion of the device memory is allocated to storing the blocks of the $\boldsymbol{\mathcal{J}}$. Throughout the factorization $\boldsymbol{\mathcal{J}}$ is not modified and thus synchronization with the host is not required. Figure 5.1 shows the memory usage of 6 matrices, each storing the entire $\boldsymbol{\mathcal{J}}$ matrix on the device. When the matrix size is sufficient such that computation time exceeds the overhead of host-device communication, it is

ideal to use alternative methods for accessing the columns of $\mathcal{J}$ instead of transferring the matrix in its entirety before factorization begins. Previous work [6] has shown that the device needs only to store a single column of $\mathcal{J}$ at a time, with new columns being transfered to the device on demand. This finding is extended to give the device the freedom to allocate enough space for a single block of size $M \times M$ instead of an entire column of $\mathcal{J}$. The already existing column transfer mode and the proposed block transfer mode are explored in the next two subsections.

5.3.1 Column Transfer Mode

The column transfer mode alters the Data Transfer and Copy-Scale steps shown in Section 4.4. By default the Data Transfer and Copy-Scale steps assume that the kth column of $\mathcal{J}$ is already resident in device memory. Instead with column transfer mode, the Data Transfer step initiates a transfer to the device for column k, then the Copy-Scale operation operates normally on each block of the column.

Column transfer mode allows the $\mathcal{J}$ buffer size to be reduced such that it needs only to hold J_{max} blocks, where J_{max} is the greatest number of non-zero blocks in any column of $\mathcal{J}$. The size of J_{max} ensures that the single largest column of $\mathcal{J}$ fits into the buffer and benefits from the sparsity of the matrix in such cases that $J_{max} << N$ where N is the dimension of $\boldsymbol{J}$.

5.3.2 Block Transfer Mode

In the event that both J_{nnz} and J_{max}, the greatest number of non-zero blocks in any column of $\mathcal{J}$ are simply too large to be allocated in device memory, block transfer mode allows for the kth column to be transfered from the host to the device block-by-block. The single block is stored in the Temp buffer (shown in Figure 4.2) and the Copy-Scale step discussed in Section 4.4 is performed after each block is successfully transferred from the host to the device.

5.3.3 Performance Impact

The performance of each mode is compared in Figure 5.2 for the rajat05 matrix. With block sizes above 192×192, the column transfer becomes preferred over full Transfer. Column transfer mode is typically the logical choice when using multiple devices since the division of factorization tasks has each device using a subset of the columns of $\mathcal{J}$. The subset of $\mathcal{J}$ assigned to each gets smaller as more devices are

used in the factorization thus full transfer mode would waste both bandwidth and memory resources given that not all of the columns will be used by every device. Block transfer should be avoided except in the case where the memory cannot hold an entire column as it gives the trade-off of using less device memory at the cost of additional host-device memory transfers. Figure 5.3 shows the memory usage of various matrices for all three transfer modes. Both column and block mode significantly reduce the memory needed to store $\mathcal{J}$. Block transfer mode reduces the memory footprint of an otherwise ten gigabytes required for the circuit_1 matrix to a fraction of a megabyte. The number of memory transfer for each column increases proportionally from one to the number of non-zero blocks in each column. Table 5.1 shows a comparison of the memory requirements for each transfer mode.

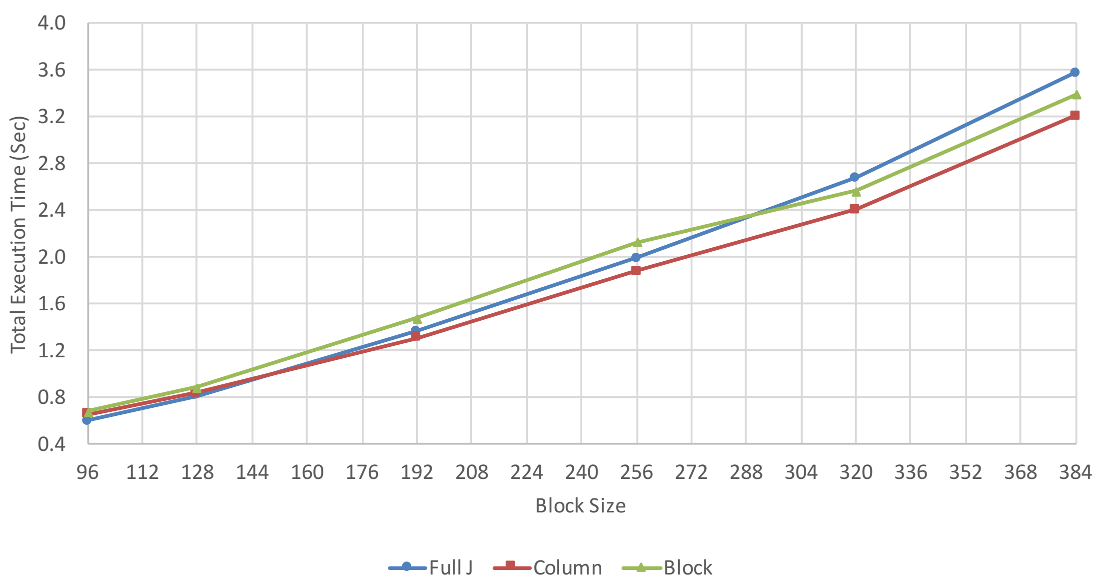


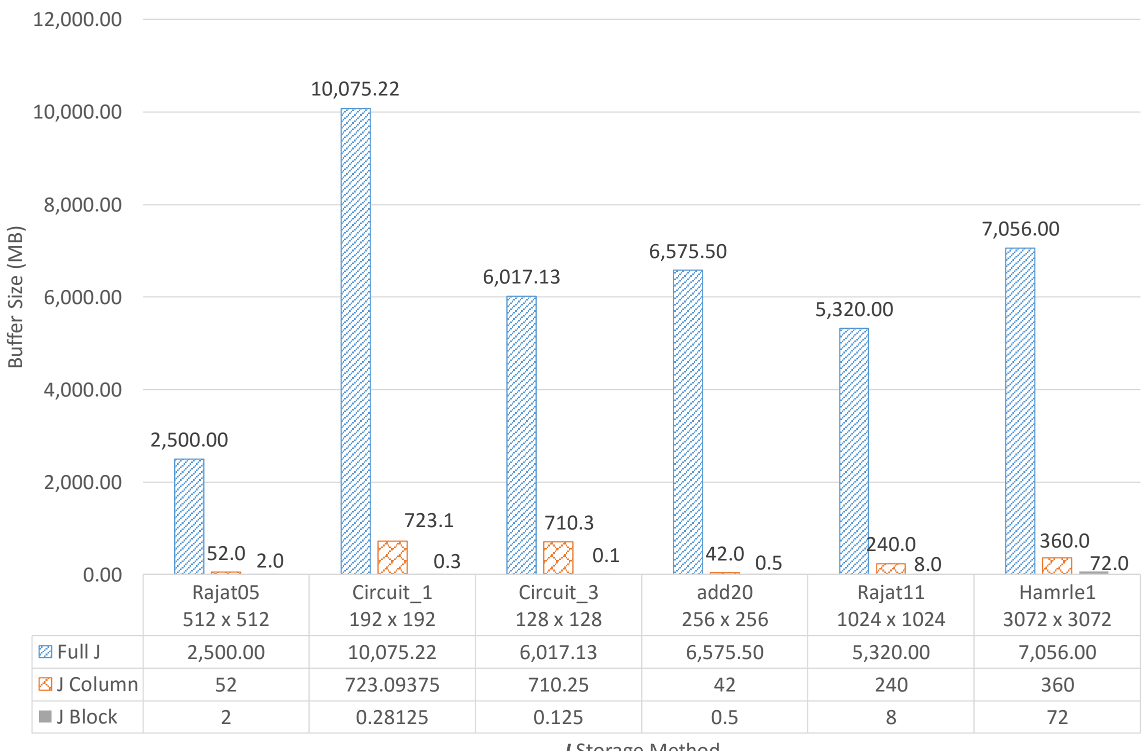
Figure 5.2: Performance of $\mathcal{J}$ transfer modes with two GPUs and varying block sizes for Rajat05

Transfer Mode	**Device Memory Usage (Blocks)**	Transfers
Full Transfer	$J = J_{nnz}$	1
Column Transfer	$J = J_{max}$	N
Block Transfer	$J = 1$	J_{nnz}

Table 5.1: Comparison of device memory usage for the $\mathcal{J}$ transfer modes

5.4 Zero Column Transfer Mode

In the previous work [6], the assumption was made that the entire $\mathcal{L}$ matrix could fit in the global memory of the accelerator and remain there for the duration of the factorization. This assumption places a hard



	Rajat05 512 x 512	Circuit_1 192 x 192	Circuit_3 128 x 128	add20 256 x 256	Rajat11 1024 x 1024	Hamrle1 3072 x 3072
Full J	2,500.00	10,075.22	6,017.13	6,575.50	5,320.00	7,056.00
J Column	52	723.09375	710.25	42	240	360
J Block	2	0.28125	0.125	0.5	8	72

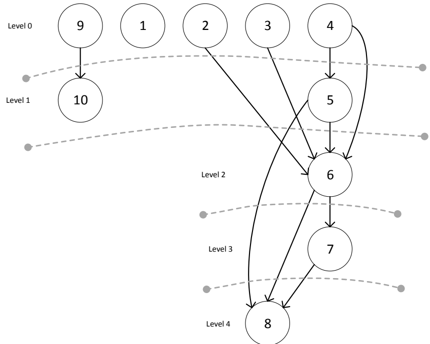
Figure 5.3: Buffer requirements of $\mathcal{J}$ transfer modes for various matrices from [1] (in Megabytes).

limit on the size of the problem which can be factorized on heterogeneous systems with discrete memory accelerators. Since the entire $\mathcal{L}$ matrix remains on the device during the factorization, it becomes the single largest consumer of device memory and effectively determines the maximum matrix and block sizes which can be factorized. This section introduces the Zero Column Transfer (ZCT) mode which reduces the number of $\mathcal{L}$ blocks that need to be stored on the device without requiring any additional host-device transfers.

5.4.1 Zero Column Transfer Description

Consider the example matrix introduced in Chapter 3 (repeated in Figure 5.4). Table 5.2 shows the state of the $\mathcal{L}$ buffer throughout the factorization, using the proposed algorithm, on a dual accelerator system. By the end of the factorization, Device 1 holds a total of eight columns while Device 2 holds five. The Tx field indicates any dependencies which were computed on another device and transferred from the host.

The basic idea behind Zero Column Transfer (ZCT) stems from the observation that the computation



(a) Dependency graph with level designations (repeated from page 38)

Level	Column
0	1 2 3 4 9
1	5 10
2	6
3	7
4	8

Device 1 (dashed red box): 1 2 3, 5, 6, 8
Device 2 (solid blue box): 4 9, 10, 7

(b) Parallel schedule derived from level designations (repeated from page 38)

Figure 5.4: Dependency graph and parallel schedule

Column	**Columns In $\mathcal{L}$ Buffer**	**Column Tx**
1	1	-
2	1, 2	-
3	1, 2, 3	-
5	1, 2, 3, 4, 5	4
6	1, 2, 3, 4, 5, 6	-
8	1, 2, 3, 4, 5, 6, 7, 8	7

(a) Device 1

Column	**Columns In $\mathcal{L}$ Buffer**	**Column Tx**
4	4	-
9	4, 9	-
10	4, 9, 10	-
7	4, 6, 7, 9, 10	6

(b) Device 2

Table 5.2: $\mathcal{L}$ Buffer Usage Without Dependence Reduction on Dual Accelerator System

of a column, say p, depends only on a subset of past columns j ($j < p$) from the $\mathcal{L}$ factor. In the example dependency graph and schedule (Figure 5.4), it is clear that not every column is required to remain in device memory throughout the factorization. There are no columns to which column $j = 1$ is a dependency, and as such it remains unused and present in device memory after being completed, thereby consuming unnecessary memory resources. The main idea is to reduce the overall amount of device memory required to store $\mathcal{L}$ by removing columns which are not required in the factorization of future columns.

The dependency graph and the parallel schedule are used to determine the minimum amount of memory to allocate to the $\mathcal{L}$ buffer without introducing any additional host-device transfers. Since work is scheduled statically, the schedule in Figure 5.4b can be analyzed to determine the memory usage characteristics before execution begins. The schedule coupled with the DAG determine when a column j is no longer needed and can be removed from memory, a value referred to as `LastUsed`(j) or column j's Last Use. The process for determining the last used values is presented next.

The first step is to find for each column $p \leftarrow 1, \ldots, N$, the last column j which p updates. This is accomplished by going through the schedule backwards and examining the DAG for each column. For Device 1, $p = 8$ is the last column to be generated and p's dependencies 5, 6, 7 must remain in memory until p is complete, making their Last Used value 8, $\texttt{LastUsed}(5, 6, 7) = 8$. Working backwards through the schedule, the last used values are determined and presented in Table 5.3. Algorithm 5 uses the last used values to determine the minimum amount of memory which can be allocated to the $\boldsymbol{\mathcal{L}}$ buffer for device i. For each column p in the schedule for device i, if column p is column j's last used value, j is removed from memory before moving p from the workspace $\boldsymbol{\mathcal{X}}$ buffer to $\boldsymbol{\mathcal{L}}$ buffer. The in-memory count variable is decreased by the number of non-zero blocks in j. The transfer of p from the workspace into $\boldsymbol{\mathcal{L}}$ increments the count by number of non-zero blocks in p. L_{peak} is the peak number of blocks which are in device memory and is the minimum amount of memory which $\boldsymbol{\mathcal{L}}$ requires on device i.

Algorithm 5: Calculate maximum $\boldsymbol{\mathcal{L}}$ buffer size for device i

```
1  begin
2      L_peak ⟵ 0;
3      count ⟵ 0;
4      for p ← 1 to N do
5          for j ← 1 to N do
6              if lastUsed(j) = p then
7                  count ⟵ count − nnz(L(j));
8              end
9          end
10         count ⟵ count + nnz(L(p));
11         if count > L_peak then
12             L_peak ⟵ count;
13         end
14     end
15 end
```

The factorization process for a column p is altered as shown in Algorithm 6. The operation of evicting a column is synchronous, thus if the column to be evicted is not yet transfered back to the host, the factorization must wait until the transfer is complete before the space it occupied is marked as free.

Using the example last used values in Table 5.3, Table 5.4 is the updated factorization reflecting columns removed from device memory. The peak value indicates the peak number of columns in $\boldsymbol{\mathcal{L}}$, for example at $p = 6$, columns 2,3,4,5 are present to update p. Columns 2,3,4 are then evicted from memory, leaving only column 6 in $\boldsymbol{\mathcal{L}}$. p is then transfered to L, giving a peak $\boldsymbol{\mathcal{L}}$ usage of 4 columns.

In the unoptimized factorization, $\boldsymbol{\mathcal{L}}$ stored eight and five columns for device 1 and 2, respectively, as shown in Table 5.2. In contrast, and as indicated in Table 5.4, using ZCT cuts the peak values and the

Algorithm 6: Basic host algorithm for the factorization of column p - augmented for ZCT

```
1  begin
2      for each block in p do
3          Transfer block from the host to device 𝓙 buffer;
4          Scale and move the block to the workspace 𝓧;
5      end
6      for each dependency j of p do
7          if column j was assigned to another device and is not finished then
8              Wait until j is factorized and in host memory;
9          end
10         if column j is not in device memory then
11             Transfer column j from host memory;
12         end
13         Update the blocks of p by j;
14     end
15     for blocks 1 to p of column p do
16         Transfer block back to the host;
17     end
18     Compute LU factorization of pivot block;
19     Find and evict all blocks with LastUsed() = p;
20     for blocks p + 1 to N of column p do
21         Solve the division by the pivot block saving the result to the 𝓛 buffer;
22         Synchronize the block with the host;
23     end
24 end
```

memory requirements in half for device 1 and by a factor of five for device 2. Table 5.5 shows the results of applying ZCT on various matrices from [1]. The results vary from marginal at a 0.015% reduction to a 63% reduction. It is clear that this optimization is not universal and results will vary, the effectiveness being ultimately determined by the structure of the matrix.

5.5 Optimal Replacement Strategy

The previous section identified the ZCT strategy for reducing the amount of memory by analyzing the dependencies of each column in the matrix. The amount of memory which can be recovered using ZCT is influenced entirely by the structure of the $\mathcal{J}$ matrix and its pivot ordering. While it was shown to effectively reduce the memory usage for some matrices, those which exhibit strong dependencies in many of the columns will not enjoy a significant reduction in memory. This result necessitates a method with less reliance upon the structure of the matrix. Without optimization, the size of both the $\mathcal{L}$ buffer and the workspace $\mathcal{X}$ can prevent the problem from being solved on an OpenCL device. The objective in this section is to introduce an optimal strategy for managing the accelerator's memory belonging to both

Column	Device 1 Last Used	Device 2 Last Used
1	1	-
2	6	-
3	6	-
4	6	4
5	8	-
6	8	7
7	8	7
8	8	-
9	-	10
10	-	10

Table 5.3: Lasted used values for each column and device

p	$\mathcal{L}$ Buffer	Tx	Evicted	Peak
1	1	-	1	1
2	2	-	-	1
3	2, 3	-	-	2
5	2, 3, 4, 5	4	-	4
6	2, 3, 4, 5, 6	-	2, 3, 4	4
8	5, 6, 7, 8	7	5, 6, 7, 8	1

(a) Device 1

p	$\mathcal{L}$ Buffer	Tx	Evicted	Peak
4	4	-	4	1
9	9	-	9	1
10	9, 10	-	9, 10	1
7	6, 7	6	6, 7	1

(b) Device 2

Table 5.4: Reducing $\mathcal{L}$ By Dependence on Dual Accelerator System

Matrix	$\mathcal{L}$ Blocks	$\mathcal{L}$ Reduced	Difference	% Difference	Memory Savings*
add20	7,382	3,521	3,861	52.30	3,016.40625
circuit_1	20,375	20,251	124	0.61	96.875
circuit_3	26,591	16,859	9,732	36.60	7,603.125
rajat05	780	289	491	62.95	383.59375
rajat11	357	157	200	56.02	156.25
rajat17	351,374	267,126	84,248	23.98	65,818.75
ASIC_100k	2,019,393	2,019,092	301	0.015	235.16
ASIC_100ks	1,769,784	1,361,186	408,598	23.09	319,217.1875
G2_circuit	9,908,638	6,340,819	3,567,819	36.01	2,787,358.594

Table 5.5: Reduced $\mathcal{L}$ by dependency analysis for a single accelerator applied to various matrices from the University of Florida Sparse Matrix Collection [1]
* In megabytes, assuming a block size of 320×320

$\mathcal{L}$ and $\mathcal{X}$.

To achieve further memory optimization of the $\mathcal{L}$ buffer and to introduce optimization to the workspace $\mathcal{X}$ buffer, it is necessary to resort to transferring data between the system's main memory and the device memory. Data which is not present in device memory is transfered from the host the next time

it is requested. When a newly factorized column or a column transfered from the host is to be stored in a buffer with insufficient space available, data is chosen to be evicted from device memory until there is enough storage space. The problem becomes appropriately choosing which data to evict such that it does not cause an excessive number of subsequent transfers as a result of the choice. If the accelerator's memory is considered as extension of the system's memory hierarchy, it becomes natural to apply the same memory replacement techniques which are used in traditional hierarchical memory systems such as Least Recently Used (LRU), Most Recently Used (MRU), and Random Replacement (RR). The commonality between each of the aforementioned replacement algorithms is that they assume no knowledge of future memory requests. However since scheduling for the column level factorization is determined statically, there is fore-knowledge into the memory requests to both the $\boldsymbol{\mathcal{L}}$ buffer and workspace throughout the numerical factorization. This fact allows for the utilization of the Optimal Replacement Algorithm (OPT) [26] [27], an optimal algorithm which always produces the least or equal number of memory access misses compared to all other existing methods. The next section briefly describes the principles of OPT.

5.5.1 Optimal Replacement (OPT)

Considering a memory hierarchy with a fast cache and a slower system memory, the objective of OPT is to minimize the number of cache misses requiring a transfer from the slower memory. Assume that a sequence of T memory requests are represented by w_i, $i = 0, ..., T-1$ and are known in advance. Each element in w belongs to the set Ω of items in slower memory. $S_t \subset \Omega$ is the set of items in cache memory with fixed size $G = |S_t|$. If at time t, $|S_t| = G$ and $w_t \notin S_t$, the request for w_t can only be completed after at least one element in S_t is evicted from the cache and replaced by w_t. OPT selects the element to evict from S_t such that its next access is farthest in the future compared to all others in S_t.

For example, let $T = 9$, $G = 3$, and a letter e.g. $A, B, C, ...$ represent each element in the slower memory Ω. A sequence of memory accesses is then defined as $\{A, C, A, D, F, I, A, C, D\}$. Further, let $S_0 = \{A, C\}$, which indicates that the cache initially contains blocks marked A and C. Table 5.6 shows the operation of the OPT algorithm and eviction choices made at each time step. It is shown that at $t = 4$ there is a cache miss which requires a decision to evict an item from S_t to be replaced by F. The forward distances is the means by which OPT determines the cache block which is accessed farthest in the future. Again, at $t = 4$ the forward distances (FD_i) for each element in S_t are as follows, A is required again at $t = 6$ giving a forward distance of $FD_A = 2$, C is used at $t = 7$ giving $FD_C = 3$, and

finally D is used at $t = 8$ with $FD_D = 4$. The block selected for eviction is identified by $\texttt{max}(FD)$, in this case $\texttt{max}(FD) = FD_D = 4$.

Table 5.6: Optimal Replacement algorithm (OPT) Example.

t	0	1	2	3	4	5	6	7	8
w_t	A	C	A	D	F	I	A	C	D
S_t	A C	A C	A C	A C D	A C F	A C I	A C I	A C I	D C I
Evicted					D	F			A
Cache Miss				D	F	I			D

5.5.2 The Forward Distance Optimization

The forward distance calculation presented above is an integral component to the OPT algorithm. Since the forward distance needs to be refreshed at each occasion where data must be chosen for eviction, it is important to achieve an efficient implementation. This section presents the forward distance calculation adopted in this work. Firstly, let $d_m(t)$ be an array containing the forward distance for each block in Ω. Upon the first call to OPT, the forward distances are calculated for every element in Ω using the function $Distance(t, q)$. For a single element q, $Distance(t, q)$ finds the forward distance of q from time t. The results are saved in $d_m(t)$. The next time, say $t + p$, when OPT is called to find an element to evict, $d_m(t)$ must be updated to reflect the forward distances at $t + p$. Matrices with very large dimensions significantly increase the computational burden of re-calculating forward distances. The increase was found to be substantial, particularly when using a single accelerator. Table 5.7 show the time taken to calculate the forward distances for all elements in Ω given an access list w, the computation time significantly increases with the size of both Ω and w.

$\lvert\Omega\rvert$	$\lvert w\rvert$	First FD (s)
12,127	29,815	0.1963
2,624	20,444	0.009323
301	761	0.0001022

Table 5.7: Computation time spent finding forward distances for various counts of total blocks and block accesses

Similar to [26], with the forward distances being known at some time $t-p$, the updated forward distances for time t can be calculated without traversing the request list w for each element in Ω. The forward

distance of element m at time t is updated by the following rule

$$d_m(t) = \begin{cases} d_m(t-p) - (t-p) & \text{if } 0 < d_m(t-p) - (t-p) < \infty \\ d_m(t-p) & \text{if } d_m(t-p) = \infty \\ Distance(t,m) & \text{if } d_m(t-p) - (t-p) \leq 0 \end{cases} \tag{5.4}$$

5.5.3 Optimal Column Replacement

Overview

To apply the principals of OPT to the problem of matrix factorization, it is extended to apply to the columns stored in the $\boldsymbol{\mathcal{L}}$ buffer. The adapted algorithm is referred to as Optimal Column Replacement (OCR). Since OPT achieves the minimum number of cache misses, OCR therefore also achieves the minimum number of host-device memory transfers. When the $\boldsymbol{\mathcal{L}}$ buffer is at capacity and cannot store any additional data, OCR selects one or more columns to evict from device memory. Recall from the execution description that blocks are placed in the $\boldsymbol{\mathcal{L}}$ buffer on two occasions:

1. At the end of the factorization of column p, the blocks belonging to p are transfered from the workspace $\boldsymbol{\mathcal{X}}$ buffer to the $\boldsymbol{\mathcal{L}}$ buffer.

2. The host-to-device transfer of a dependency j which is not present on the device. j was either previously evicted from the device or it was computed on another device.

The host algorithm modifications required to support OCR are reflected in Algorithm 7. The amount of free memory is verified both before a dependency j is transfered from the host (Line 11) and before column p undergoes the division by the pivot block (Line 22). In both cases, if the amount of memory is determined to be insufficient, OCR is used to evict one or more columns until there is enough space to continue.

Evicting a column from the $\boldsymbol{\mathcal{L}}$ buffer is relatively inexpensive since the columns are not modified after being initially computed and have already been synchronized with the host. The space in $\boldsymbol{\mathcal{L}}$ occupied by an evicted column are marked as free and overwritten as needed.

Algorithm 7: Basic host algorithm for the factorization of column p - augmented for OCR

```
1  begin
2      for each block in p do
3          Transfer block from the host to device 𝓙 buffer;
4          Scale and move the block to the workspace 𝓧;
5      end
6      for each dependency j of p do
7          if column j was assigned to another device and is not finished then
8              Wait until j is factorized and in host memory;
9          end
10         if column j is not in device memory then
11             if 𝓛 buffer free memory < size of column j then
12                 Use OCR to determine the column(s) to evict until there is enough space to store j;
13             end
14             Transfer column j from host memory;
15         end
16         Update the blocks of p by j;
17     end
18     for blocks 1 to p of column p do
19         Transfer block back to the host;
20     end
21     Compute LU factorization of pivot block;
22     if 𝓛 buffer free memory < size of column p then
23         Use OCR to determine the column(s) to evict until there is enough space to store p;
24     end
25     for blocks p + 1 to N of column p do
26         Solve the division by the pivot block saving the result to the 𝓛 buffer;
27         Synchronize the block with the host;
28     end
29 end
```

Description of Implementation

The implementation of OCR starts by defining, for each device, a memory access list $Q = q_0, q_1, ..., q_k$, $k \in \mathbb{N}$ as a sequence which contains the columns of $\mathcal{L}$ in the order in which they are requested. Q is determined during the symbolic factorization and is fully known for each device upon finalizing the scheduling task. The length of Q is determined by the total number of accesses to the columns of $\mathcal{L}$ throughout the factorization and is likely to contain repeated entries. The function `Distance`(p, q) calculates the forward distance between the current position in the sequence p and the column q's next access. `Distance`(p, q) is ≥ 1 for q which exists in $(Q_k)_{k>p}$ and infinity if q does not exist.

Each time there is insufficient space in $\mathcal{L}$ to store new data, Algorithm 8 is used to determine the optimal column to evict. The algorithm attempts to find a column with the greatest distance from the current position p. Recall that columns which will not be used after p have distances of infinity, therefore

Algorithm 8 will always select such a column before a column which will be used again in the future.

Algorithm 8: Optimal column replacement algorithm (OCR)

```
1 begin
      input : An ordered sequence Q, current index in the sequence p, set of columns currently in
              device memory C_p
      output: The column n to evict from memory
2     highest ← 0;
3     for q ∈ C_p do                           // Find the column used furthest in the future
4         if highest < distance(p, q) then
5             highest = distance(p, q);
6             n = q;
7         end
8     end
9 end
```

Allowable Buffer Sizes

Optimal column replacement requires the $\boldsymbol{\mathcal{L}}$ buffer to hold a minimum of L_{max} blocks, where L_{max} is the greatest number of non-zero blocks in any column of $\boldsymbol{\mathcal{L}}$. This bound ensures that the device can store any single column of $\boldsymbol{\mathcal{L}}$ in the buffer. The upper bound of the $\boldsymbol{\mathcal{L}}$ buffer is L_{peak}, which is the peak memory usage determined by dependency analysis performed by the Zero Column Transfer mode. A size of $L_{nnz} \geq L \geq L_{peak}$ will not introduce additional device-host transfers, and behaves in the same fashion as ZCT. The closer the size of $\boldsymbol{\mathcal{L}}$ is to L_{peak}, the lower the number of bus transfer introduced by evicting columns. With the minimum size of L_{max}, only one column resides in $\boldsymbol{\mathcal{L}}$ at a given time. In this situation, the blocks of the pth column of $\boldsymbol{\mathcal{L}}$ reside in the workspace during computation, a column j which update p is transfered into $\boldsymbol{\mathcal{L}}$, j updates p and is then replaced with the next column that updates p.

5.5.4 Optimal Block Replacement

Utilizing all of the memory schemes developed thus far, the last determining factor to fit the problem in device memory becomes the workspace. The workspace is used to store the blocks of $\boldsymbol{\mathcal{L}}$ and $\boldsymbol{\mathcal{U}}$ belonging to the pth column while it's being computed. Without optimization, the device must be able to fit the largest column of the combined LU factor, LU_{max}, in the workspace. With the sparse nature of the matrices, the workspace tends to be such that $LU_{max} << N$ where N is the dimension of $\boldsymbol{J}$. Nonetheless, a single column in the LU factor close to N will define the workspace size and this will place a limit

on the matrix size which can be factorized. This section describes the optimization of the workspace $\boldsymbol{\mathcal{X}}$ using a modified OPT algorithm.

The Multiply-Subtract-Accumulate operation is defined as follows:

$$C \leftarrow \alpha AB + \beta C \tag{5.5}$$

with $\alpha = -1$ and $\beta = 1$. This operation is performed repeatedly to update the blocks of column p in the workspace. The operand A resides in $\boldsymbol{\mathcal{L}}$ while B and C are contained in $\boldsymbol{\mathcal{X}}$. This implies that the workspace is at a minimum two blocks in size, one for each B and C, and a maximum of N blocks. When the workspace with a size less than N has reached its maximum storage capacity, a currently residing block must be chosen for eviction and transferred to the host memory. The choice of block to evict is accomplished through an approach similar to Optimal Column Replacement, however, applied to the blocks of the workspace. A memory access list for each column's workspace is constructed and stored during the symbolic phase, allowing the OPT algorithm to be applied in the same fashion as OCR, where the workspace block accessed furthest in the future is evicted from the device memory.

Unlike OCR, an evicted block from the workspace may need to be synchronized back to the host instead of being overwritten. For this reason, a dirty bit is kept for each block in the workspace to indicate if the contents of the block have changed since the last synchronization with the host. Upon completion of all Multiply-Subtract-Accumulate operations, the modified contents of the device workspace are synchronized back to the host. Since the host now stores the most up-to-date blocks, there is no need to maintain a full $\boldsymbol{\mathcal{U}}$ buffer in the device's memory. The blocks of $\boldsymbol{\mathcal{U}}$ are copied from the host workspace buffer into the host $\boldsymbol{\mathcal{U}}$ buffer. The device $\boldsymbol{\mathcal{U}}$ buffer need only store the pivot block which will be decomposed on the device.

To limit the number of memory transfers resulting from OBR, the blocks of the $\boldsymbol{\mathcal{J}}$ matrix are scaled and copied by the CPU to a host memory workspace. This removes the need to store or transfer any $\boldsymbol{\mathcal{J}}$ blocks on the device. Each subsequent operation involving a host stored workspace block which has not yet been transfered to the device invokes a transfer between the host workspace and the device workspace.

The workspace OBR method is most effectively used in situations where a large matrix prevents the workspace from being stored in its entirety in device memory.

5.6 Memory Optimization for the $\boldsymbol{\mathcal{U}}$ Factor

Previous work [6] has shown that once computed, the columns of the $\boldsymbol{\mathcal{U}}$ factor are not used to update any future columns. The amount of memory allocated to store the $\boldsymbol{\mathcal{U}}$ factor on the device can therefore be significantly reduced. The $\boldsymbol{\mathcal{U}}$ buffer can have a minimum size of 1 block and a maximum of U_{nnz}, which is the number of non-zero blocks in $\boldsymbol{\mathcal{U}}$.

5.7 Summary

The optimizations introduced in this chapter can be used in varying combinations to solve large problems on OpenCL accelerators. The combinational choices of memory optimization methods are outlined in Table 5.8. The top row lists the 6 memory modes, with each column showing the size limitation imposed on each buffer in that mode. A dash indicates a free choice of any method which optimizes that buffer. For example, the $\boldsymbol{\mathcal{J}}$ Column transfer method can be combined with any $\boldsymbol{\mathcal{L}}$ buffer and $\boldsymbol{\mathcal{U}}$ optimization.

Buffer	$\boldsymbol{\mathcal{J}}$ Column	$\boldsymbol{\mathcal{J}}$ Block	Reduced $\boldsymbol{\mathcal{U}}$	ZCT $\boldsymbol{\mathcal{L}}$	OCR $\boldsymbol{\mathcal{L}}$	OBR $\boldsymbol{\mathcal{X}}$
A	A_{max}	1	-	-	-	0
U	-	-	$U_{nnz} \geq \boldsymbol{\mathcal{U}} \geq 1$	-	-	1
L	-	-	-	L_{peak}	$L_{peak} \geq \boldsymbol{\mathcal{L}} \geq L_{max}$	-
X	LU_{max}	LU_{max}	LU_{max}	-	-	$2 \leq \boldsymbol{\mathcal{X}} \leq LU_{max}$

Table 5.8: Memory Optimization Matrix

Each of the methods have their own impact on the overall performance of the factorization. As well, the performance of each method will vary depending on the system specifics. Based on experimental results, the following performance impacts were observed.

- $\boldsymbol{\mathcal{J}}$ transfer methods have limited impact on performance
- The is no significant performance impact when using any $\boldsymbol{\mathcal{U}}$ buffer size within the allowed bounds of $U_{nnz} \geq \boldsymbol{\mathcal{U}} \geq 1$
- OBR $\boldsymbol{\mathcal{X}}$ has a very significant impact on performance
- ZCT $\boldsymbol{\mathcal{L}}$ has no impact on performance
- OCR $\boldsymbol{\mathcal{L}}$ has impact on performance related to the number of additional transfers and synchroniza-

tion overhead

To demonstrate the different memory reductions, Figure 5.5 shows the memory usage of the rajat05 matrix with a block size of 320×320 on a device with 2 gigabytes of memory. Without using any mode, the device memory is overcommitment by 20%. Zero Column Transfer reduces the memory requirements to just above the device's 2GB threshold. Only with a choice of OCR, $\mathcal{J}$ block, or OBR does the problem to fit within the device memory.

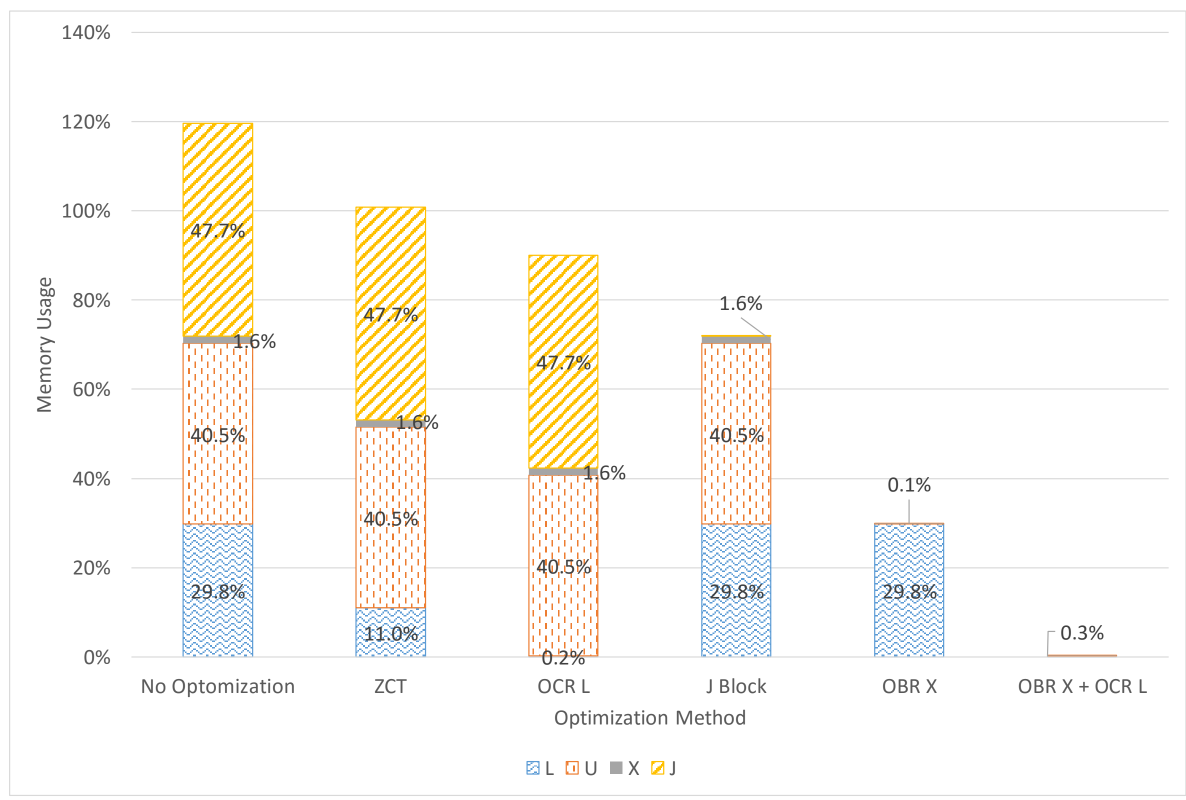


Figure 5.5: Memory techniques applied to Rajat05 Matrix with block size 320×320 and total device memory of 2GB

Chapter 6

Experimental Results

6.1 System Setup

The specifications of the system used to factorize the benchmark matrices are presented in Table 6.1. The system has two AMD 7950 GPUs, one clocked at 900 MHz while the other is 925 MHz. Both GPUs have 3 GB of on-board memory and are connected over a PCIe 2.0 bus. The host CPU is a 6-core Intel Xeon E5-2620 clocked at 2 GHz and with 48 GB of system memory.

	CPU	GPU	
Device	Intel Xeon E5-2620	AMD Radeon 7950 #1	AMD Radeon 7950 #2
DRAM	48 GB	3 GB	3 GB
Cores	6	112 x 16-Lane SIMDs	112 x 16-Lane SIMDs
Clock Freq.	2 GHz	900 MHz	925 MHz
OpenCL Version	1.2	1.2	1.2

Table 6.1: Test system specifications

6.2 Benchmark Matrices

Six benchmark matrices from the University of Florida Sparse Matrix Collection [1] are used to obtain simulation results. The matrices are naturally scalar and do not exhibit the targeted dense block structure, however their sparsity pattern can be used as a template. The Kronecker Product is used to

transform the sparsity pattern of the benchmark matrices into block-structured matrices and is outlined in the next section.

6.2.1 Kronecker Product

To obtain block structure matrices resembling that of the Jacobian matrix obtained from circuit simulation, the Kronecker product operation is used on each of the benchmark matrices. The Kronecker product is defined as follows.

$$\boldsymbol{\mathcal{J}} = \boldsymbol{J} \otimes \boldsymbol{B} = \begin{pmatrix} j_{1,1}\boldsymbol{B} & j_{1,2}\boldsymbol{B} & \cdots & j_{1,n}\boldsymbol{B} \\ j_{2,1}\boldsymbol{B} & j_{2,2}\boldsymbol{B} & \cdots & j_{2,n}\boldsymbol{B} \\ \vdots & \vdots & \ddots & \vdots \\ j_{m,1}\boldsymbol{B} & j_{m,2}\boldsymbol{B} & \cdots & j_{m,n}\boldsymbol{B} \end{pmatrix} \tag{6.1}$$

The Kronecker product operation takes two matrices, $\boldsymbol{J}$ of size $m \times n$ and B of size $p \times q$. Each scalar non-zero of $\boldsymbol{J}$ is expanded in-place to a size of $p \times q$ and replaced by its product with the matrix $\boldsymbol{B}$. This results in $\boldsymbol{\mathcal{J}}$, a block structured $mp \times nq$ matrix. More explicitly, the Kronecker equation above is expanded to create a block structure matrix as follows.

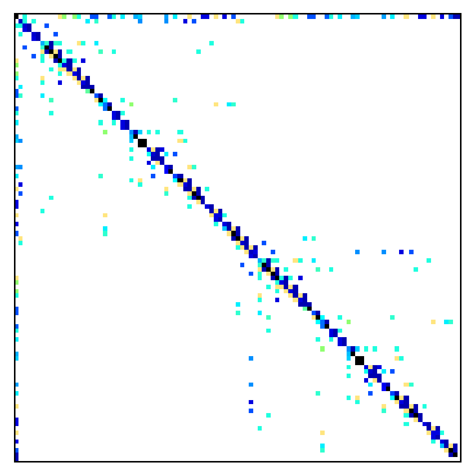
$$\boldsymbol{J} \otimes \boldsymbol{B} = \begin{pmatrix} \begin{pmatrix} j_{1,1}b_{1,1} & \cdots & j_{1,1}b_{1,q} \\ \vdots & \ddots & \vdots \\ j_{1,1}b_{p,1} & \cdots & j_{1,1}b_{p,q} \end{pmatrix} & \cdots & \begin{pmatrix} j_{1,n}b_{1,1} & \cdots & j_{1,n}b_{1,q} \\ \vdots & \ddots & \vdots \\ j_{1,n}b_{p,1} & \cdots & j_{1,n}b_{p,q} \end{pmatrix} \\ \vdots & \ddots & \vdots \\ \begin{pmatrix} j_{m,1}b_{1,1} & \cdots & j_{m,1}b_{1,q} \\ \vdots & \ddots & \vdots \\ j_{m,1}b_{p,1} & \cdots & j_{m,1}b_{p,q} \end{pmatrix} & \cdots & \begin{pmatrix} j_{m,n}b_{1,1} & \cdots & j_{m,n}b_{1,q} \\ \vdots & \ddots & \vdots \\ j_{m,n}b_{p,1} & \cdots & j_{m,n}b_{p,q} \end{pmatrix} \end{pmatrix} \tag{6.2}$$

6.2.2 Overview of Benchmark Matrices

This section introduces the six benchmark matrices by visually presenting their sparsity patterns along with the number of non-zeros in the scalar matrix, and hence the number of non-zero blocks that will be present after the performing the Kronecker product. The values of LU_{max} (the greatest number of non-zero blocks in any column of the LU matrix), J_{max} (the greatest number of non-zero blocks in any

column of J), L_{nnz} (the number of non-zero blocks in the $\mathcal{L}$ factor) and U_{nnz} (the number of non-zero blocks in the $\mathcal{U}$ factor) are also provided.

Rajat05



Dimensions = 301×301

Number of non-zero blocks = 1,250

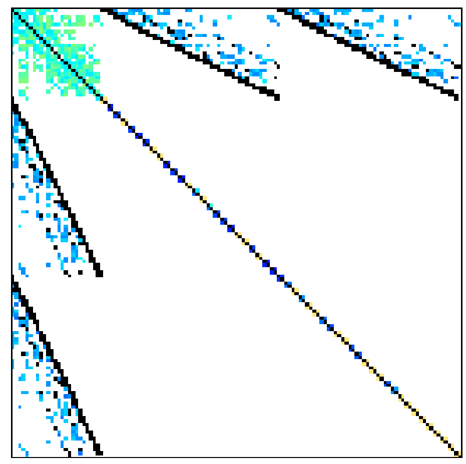
$J_{max} = 26$ Blocks

$LU_{max} = 43$ Blocks

$L_{nnz} = 781$ Blocks

$U_{nnz} = 1,062$ Blocks

Add20



Dimensions = $2,395 \times 2,395$

Number of non-zero blocks = 13,151

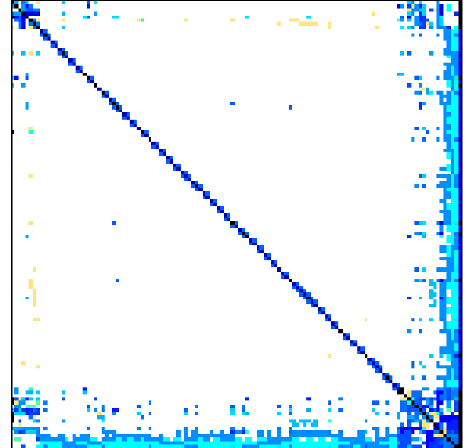
$J_{max} = 84$ Blocks

$LU_{max} = 126$ Blocks

$L_{nnz} = 7,383$ Blocks

$U_{nnz} = 9,777$ Blocks

Circuit_1



Dimensions $= 2,624 \times 2,624$

Number of non-zero blocks = 35,823

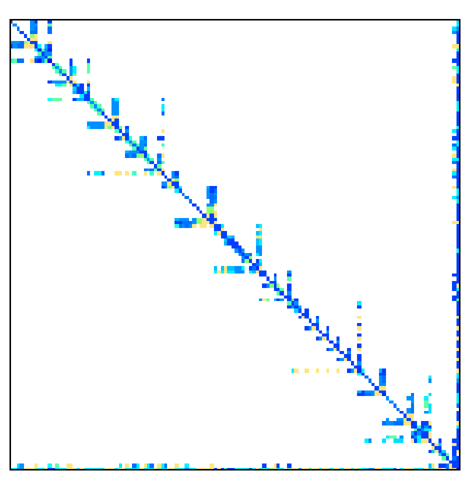
$J_{max} = 2,571$ Blocks

$LU_{max} = 2,572$ Blocks

$L_{nnz} = 20,375$ Blocks

$U_{nnz} = 23,068$ Blocks

Circuit_3



Dimensions $= 12,127 \times 12,127$

Number of non-zero blocks = 48,137

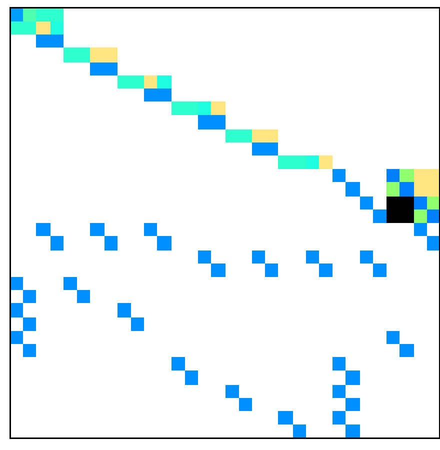
$J_{max} = 5,682$ Blocks

$LU_{max} = 7,971$ Blocks

$L_{nnz} = 26,591$ Blocks

$U_{nnz} = 41,942$ Blocks

Hamrle1



Dimensions $= 32 \times 32$

Number of non-zero blocks = 98

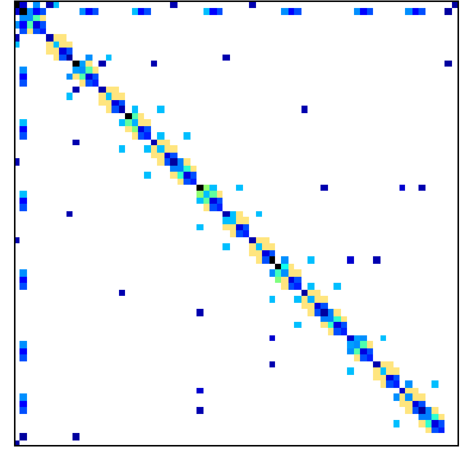
$J_{max} = 5$ Blocks

$LU_{max} = 10$ Blocks

$L_{nnz} = 60$ Blocks

$U_{nnz} = 71$ Blocks

Rajat11



Dimensions = 135 × 135

Number of non-zero blocks = 665

J_{max} = 30 Blocks

LU_{max} = 47 Blocks

L_{nnz} = 358 Blocks

U_{nnz} = 497 Blocks

6.3 Experimental Results

The results are summarized in Table 6.2 where the factorization time for the following four solvers is shown:

1. CPU-based parallel NICSLU [20], written in C, and executed with 6 threads
2. A parallel implementation of BKLU from [6], re-written in C++ and executed with 6 threads
3. BKLUX, with one GPU scheduled sequentially, written in C++/OpenCL
4. BKLUX with two GPUs, written in C++/OpenCL

The parallel CPU solvers (NICSLU [20] and BKLU [6]) use 6 threads, which is the maximum available on the test platform. All solvers have disabled partial pivoting to avoid pivots caused by the choice of expansion block $\boldsymbol{B}$. The first column of Table 6.2, M, shows the dimension of the expansion block $\boldsymbol{B}$, the second and third columns, $\mathcal{J}$ and LU + X, show the memory size of the generated Jacobian matrix and the memory size of the LU factors with the workspace. Columns 4-7 show the execution time for the four direct factorization methods. The final two columns indicate the speedup obtained from the proposed BKLUX with two GPUs versus BKLU-CPU and BKLUX with a single GPU respectively.

The speedup that BKLUX with two GPUs provides over the 6 threaded BKLU is shown visually in Figure 6.1. As the block size increases, the computational edge provided by the GPU becomes more apparent. Taking the Rajat05 matrix as an example, a block size of 96 × 96 yields a modest 1.78x

Table 6.2: Simulation Results

	M	$\mathcal{J}$ (MB)	LU + X (MB)	Time (seconds) NICSLU [20]⋆	BKLU-CPU⋆	BKLU-GPU†	**This work**‡	Speedup⋆⋆	Speedup⋆⋆⋆
Rajat05	96	87.89	132.68	2.27	1.01	1.08	**0.57**	1.78	1.91
	256	625.00	943.50	36.84	14.47	3.29	**1.75**	8.27	1.88
	384	1,406.25	2,122.88	121.62	48.24	5.98	**3.08**	15.66	1.94
	512	2,500.00	3,774.00	549.18	113.47	9.98	**5.03**	22.57	1.98
	720	4,943.85	7,463.23	–	358.76	27.45$^{\alpha}$	**14.22**$^{\alpha}$	25.23	1.93
	848	6,857.91	10,352.70	–	603.05	39.06$^{\alpha}$	**20.42**$^{\alpha}$	29.54	1.91
Add20	64	410.97	540.22	8.79	5.08	6.93	**3.73**	1.36	1.86
	96	924.68	1,215.49	27.67	15.36	9.84	**5.34**	2.88	1.84
	128	1,643.88	2,160.88	70.87	36.64	13.22	**7.37**	4.97	1.79
	160	2,568.55	3,376.37	162.45	68.68	16.57$^{\alpha}$	**9.69**$^{\alpha}$	7.09	1.71
	256	6,575.50	8,643.50	–	277.90	33.08$^{\alpha}$	**18.37**$^{\alpha}$	15.13	1.80
	320	10,274.22	13,505.47	–	–	47.47$^{\beta}$	**25.18**$^{\alpha}$	–	1.89
Circuit_1	48	629.70	808.88	31.19	16.44	18.27	**11.63**	1.41	1.57
	64	1,119.47	1,438.00	72.40	35.82	15.39	**10.33**	3.47	1.49
	96	2,518.80	3,235.50	–	110.81	24.09	**15.67**	7.07	1.54
	128	4,477.88	5,752.00	–	251.26	33.04$^{\beta}$	**19.48**$^{\beta}$	12.90	1.70
	192	10,075.22	12,942.00	–	811.37	74.83$^{\beta}$	**45.63**$^{\beta}$	17.78	1.64
	208	11,824.39	15,188.88	–	994.23	127.36$^{\beta}$	**76.39**$^{\beta}$	13.02	1.67
Circuit_3	48	846.16	1,344.81	18.07	6.63	26.84	**14.20**	0.47	1.89
	64	1,504.28	2,390.78	38.52	14.21	24.76	**13.10**	1.08	1.89
	96	3,384.63	5,379.26	116.63	44.03	34.12$^{\alpha}$	**19.6**$^{\alpha}$	2.25	1.74
	128	6,017.13	9,563.13	261.93	103.18	48.62$^{\beta}$	**25.37**$^{\beta}$	4.07	1.92
	160	9,401.76	14,942.38	–	196.03	62.40$^{\beta}$	**38.68**$^{\beta}$	5.07	1.61
Hamrle1	128	12.25	17.75	0.22	0.19	0.13	**0.10**	2.01	1.35
	256	49.00	71.00	2.39	1.19	0.27	**0.19**	6.40	1.47
	512	196.00	284.00	43.59	9.16	0.74	**0.46**	19.92	1.61
	1024	784.00	1,136.00	201.65	79.13	2.42	**1.48**	53.46	1.64
	3072	7,056.00	10,224.00	–	2251.61	29.22$^{\beta}$	**17.17**$^{\beta}$	131.10	1.70
	4352	14,161.00	20,519.00	–	–	102.58$^{\gamma}$	**57.58**$^{\gamma}$	–	1.78
Rajat11	64	20.78	28.22	0.33	0.19	0.34	**0.21**	0.88	1.60
	128	83.13	112.88	2.46	1.13	0.65	**0.38**	2.96	1.70
	384	748.13	1,015.88	76.87	28.12	2.70	**1.53**	18.40	1.77
	512	1,330.00	1,806.00	183.73	66.79	4.38	**2.46**	27.09	1.78
	720	2,630.13	3,571.44	–	201.58	12.57$^{\alpha}$	**6.76**$^{\alpha}$	29.82	1.86
	1536	11,970.00	16,254.00	–	–	41.92$^{\alpha}$	**22.39**$^{\alpha}$	–	1.87

⋆ NICSLU and BKLU-CPU are implemented on 6 threads.
† BKLUX implemented on single GPU.
‡ BKLUX implemented with 2 GPUs.
– Host memory exhausted
$^{\alpha}$ ZCT used
$^{\beta}$ OCR used
$^{\gamma}$ Workspace swapping used
⋆⋆ The proposed with 2 GPUs versus BKLU-CPU with 6 threads.
⋆⋆⋆ The proposed with 2 GPUs versus a single GPU.

speedup, while a block size of 848×848 yields 29.5x. Further, Figure 6.2 shows the speedup of BKLUX with a single GPU versus BKLUX with two GPUs. Many of the results are close the optimal case od a 2x speedup.

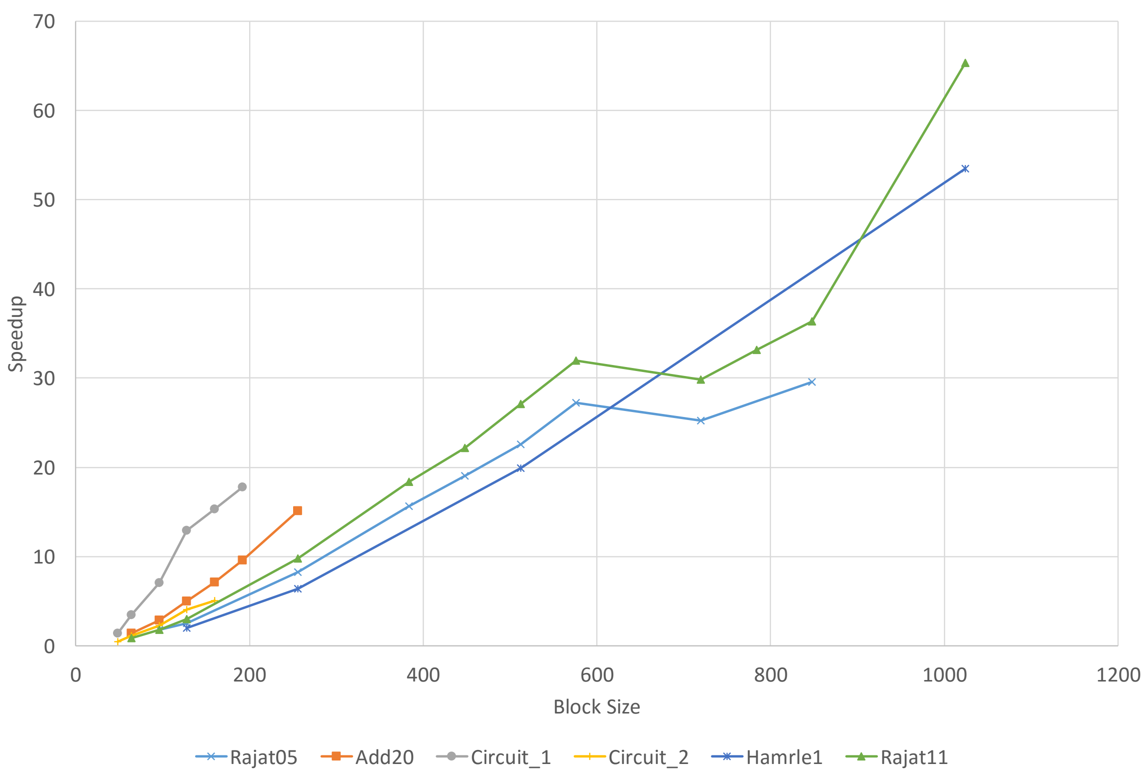


Figure 6.1: Speedup for benchmark matrices, BKLU with two GPUs versus BKLU-CPU with six threads

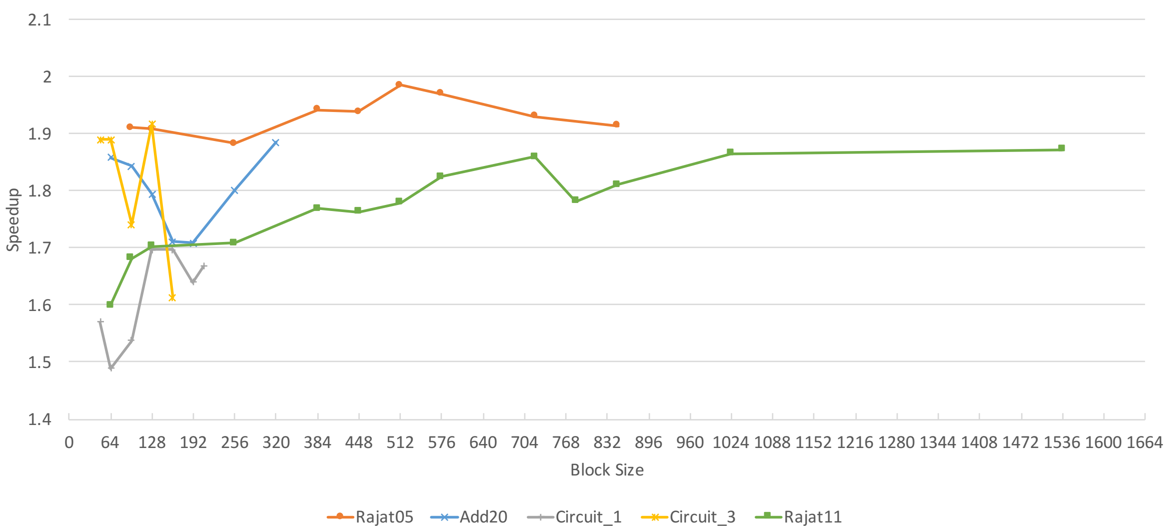


Figure 6.2: Speedup for benchmark matrices, BKLUX with one GPU versus BKLUX with two GPUs

6.4 Execution Profile

6.4.1 Profiling Device Kernels

The device executes four different OpenCL kernels numerous times during the factorization of each column. The four kernels are

- Block Scale - Multiplies all entries in a block by a scalar value.
- Multiply Accumulate Subtract - Performs the GEMM operation.
- LU factorization - Factorizes the pivot block into its LU factors
- Triangular solve - Performs the division operation for each $\mathcal{L}$ block by the pivot block.

Figure 6.3 shows the relative device time spent executing each of these kernels during the factorization of the rajat05 matrix with a block size of 320×320. Triangular solve is shown to be the most time intensive operation followed by the pivot factorization as both of these operations are inherently sequential in nature, especially so for dense blocks.

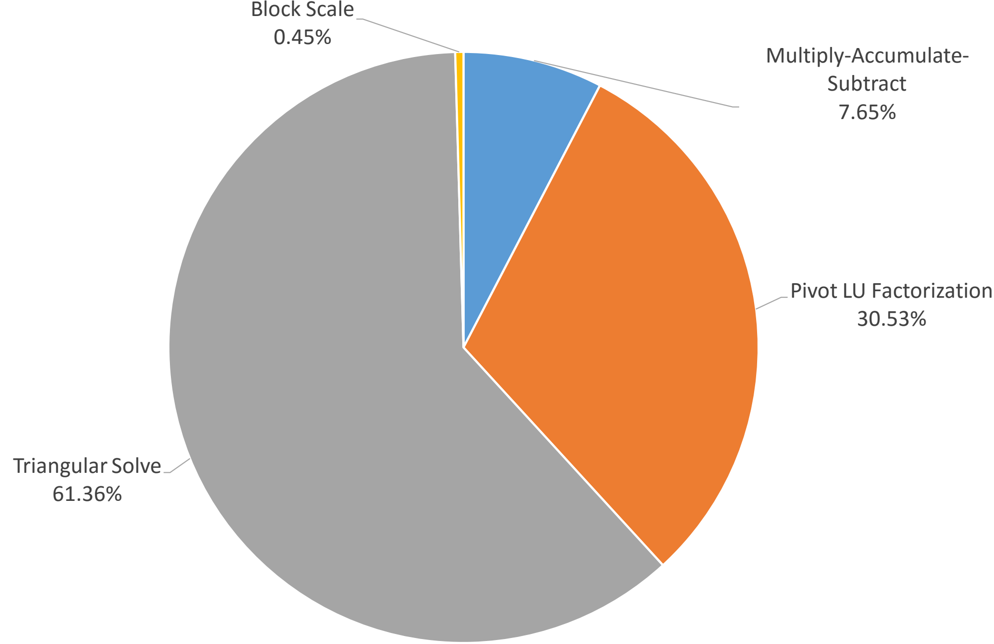


Figure 6.3: Single GPU kernel execution profile for rajat05 matrix with a block size of 320×320

6.5 Summary

In this chapter, the experimental results of proposed were presented using six benchmark matrices. It was shown that BKLUX provides a significant improvement compared to CPU based approaches showing speedups ranging from 2 times to 131 times. It was also shown that the for many of the matrices, the proposed approach scales to two GPUs almost optimally. The final section presented a profile OpenCL kernel's execution of the GPU.

Chapter 7

Profiling of Benchmark Matrices

7.1 Introduction

This chapter is dedicated to profiling the factorization of the hamrle1 and rajat11 benchmark matrices, presenting details about the execution and the effect of using the memory optimization techniques discussed in Chapter 5. The Hamrle1 matrix is used due to its relatively small size which will make the presentation less cumbersome while the rajat11 with its larger size offers a better perspective for situations where the problem size is larger than the device's memory. The following two sections will present details about both of the benchmark matrices and their factorization on a two GPU system as described in Chapter 6. This is followed by several sections which outline varying execution scenarios. Note that due to the overhead of profiling, the timing results will differ slightly from those reported in Chapter 6.

7.2 Factorization of the Harmle1 Matrix

The scalar Hamrle1 matrix is 32×32 with 98 non-zeros, the details of the matrix are summarized in Table 7.1 and the sparsity pattern is shown in Figure 7.1. The Directed Acyclic Graph depicting the column dependencies of the matrix is captured in Figure 7.2. After performing parallel analysis of the DAG, the parallel schedule presented in Table 7.2 is produced.

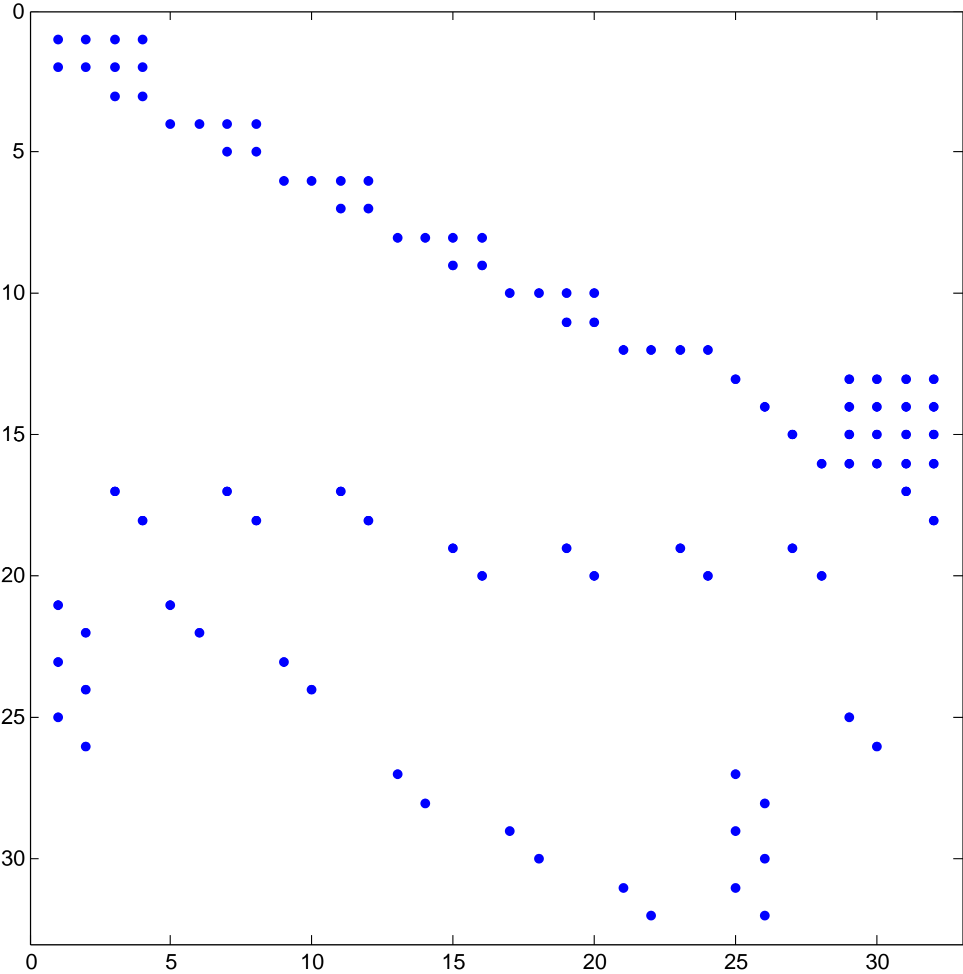


Figure 7.1: Hamrle1 Sparsity Pattern

Dimension	32×32
$\mathcal{J}$ Non-zeros	98 blocks
$\mathcal{L}$ Non-zeros	60 blocks
$\mathcal{U}$ Non-zeros	71 blocks
Longest Column in $\mathcal{L}$	4 blocks
Longest Column in combined LU factor (LU_{max})	10 blocks
Longest Column in $\mathcal{J}$ (J_{max})	5 blocks

Table 7.1: Hamrle1 Matrix Details

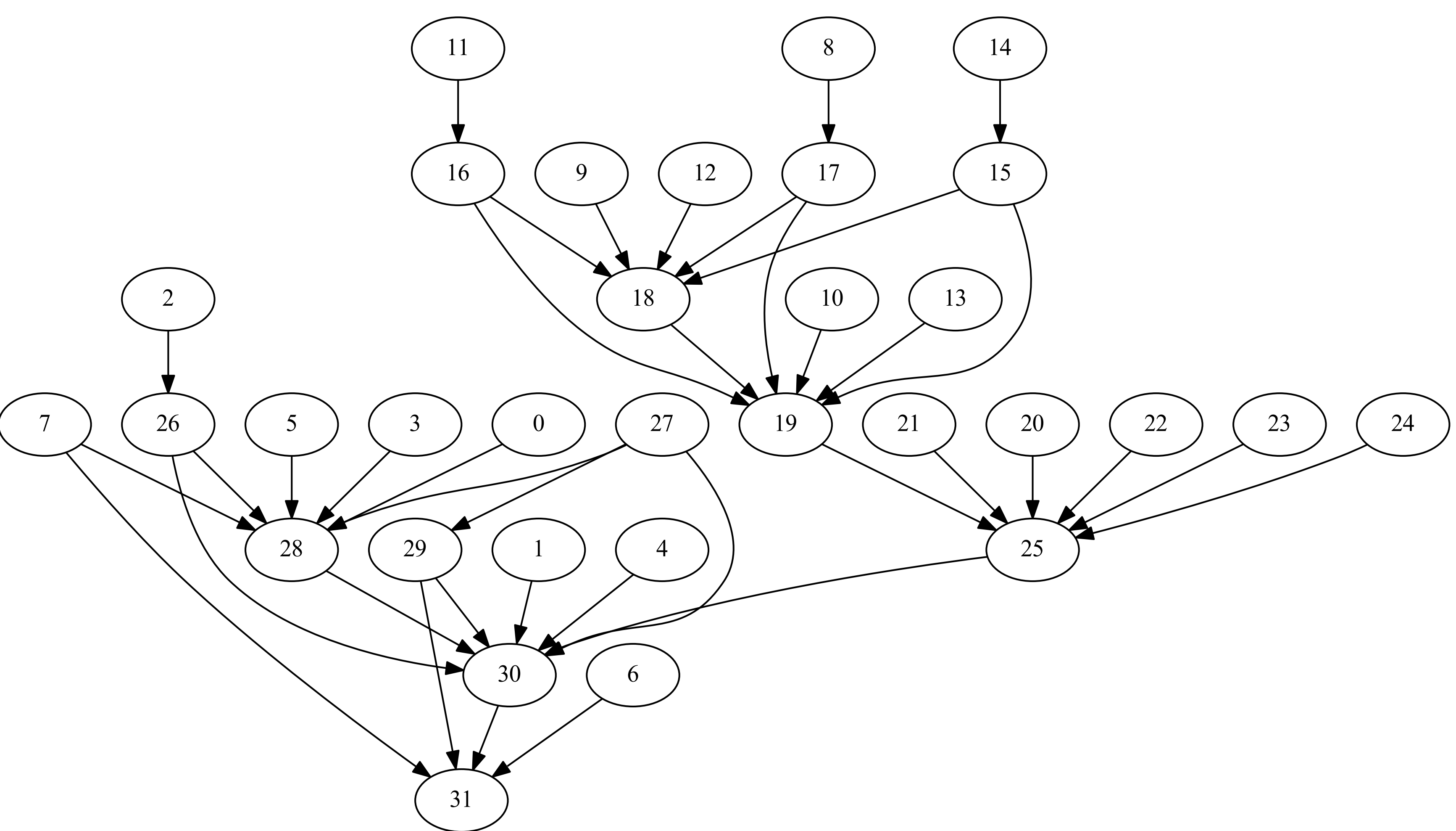


Figure 7.2: Hamrle1 Dependency Graph

Level	Columns Assigned to GPU1	Columns Assigned to GPU2
0	0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13	14, 20, 21, 22, 23, 24, 27
1	15, 16	17, 26, 29
2	18	28
3	19	
4	-	25
5	30	-
6	-	31

Table 7.2: Parallel Schedule for the Hamrle1 Matrix

The factorization is carried out at a block size of 1024×1024 and the entire problem to fits into GPU memory. Table 7.3 outlines the details of the factorization.

Block Size	1024×1024
$\mathcal{J}$ Buffer Size	5
$\mathcal{J}$ Buffer Memory Management Mode	Column Transfer
$\mathcal{L}$ Buffer Size	60
$\mathcal{L}$ Buffer Memory Management Mode	None
$\mathcal{U}$ Buffer Size	71
$\mathcal{U}$ Buffer Memory Management Mode	None
Total Factorization Time	1.58 seconds
Number of memory transfers between devices	16
Memory Usage	896 MB

Table 7.3: Hamrle1 matrix factorization details for GPU

The execution timeline from the perspective of the two host threads is captured in Figure 7.3. During the factorization, Thread 0 encounters three blocking conditions while thread 1 sees two. The directed arrows between threads represent the resolution of a blocking state by pointed from the dependency causing the stall to the end of the blocking condition. For example, Thread 0 is blocked for approximately 0.23 seconds while waiting for Thread 1 to complete column 17 and similarity Thread 1 is blocked for approximately 0.75 seconds waiting for column 19. The timeline shows that a blocking condition on the host does not necessarily translate to a blocked state on the accelerator. This is apparent by the fact that despite Thread 0 being bocked at (1), the GPU continues to factorize previously issued column tasks independently, completing columns 9, 10, 11 and 12. The five blocking conditions occurring throughout the factorization are summarized in Table 7.4, during which the host thread busy waits.

The factorization from the perspective of the host is useful, however it does not reveal the amount of time the GPU spends idle. To get an idea of the idle time, a second pass at the factorization is done which inserts *clFinish* function calls before a thread begins waiting. This prevents the host thread from advancing until the device has finished executing all previously issued commands, allowing for the measure of idle time between the moment when all previously issued commands are completed and resuming execution after the dependency has been made available. The fourth column in Table 7.4 indicates this GPU idle time. It should be noted that the overhead involved in the additional synchronization (inherent when using *clFinish*) forces the GPU idle results to be approximate.

#	Thread/GPU	CPU (sec)	GPU (sec)	Column	Unfinished Dependency
①	0	0.226	0.0	18	17
②	0	0.151	0.0	30	26
③	0	0.545	0.17	30	25
④	1	0.748	0.26	25	19
⑤	1	0.281	0.10	31	30
GPU0 Total Idle: 0.17s			GPU1 Total Idle: 0.36s		

Table 7.4: Overview of the amount of time the CPU and GPU spend waiting on synchronization events during the factorization of the Hamrle1 matrix

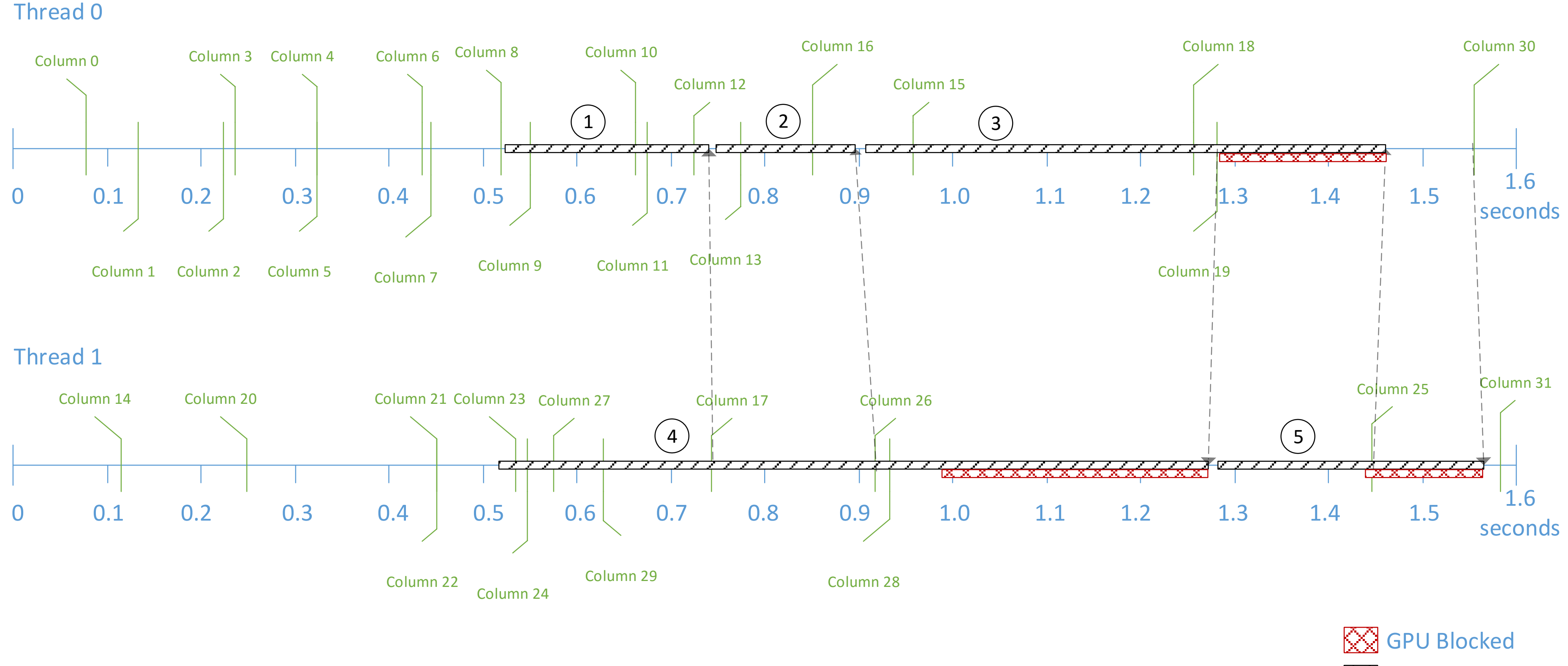


Figure 7.3: Hamrle1 Execution Profile Scenario One

7.3 Factorization of the Rajat11 Matrix

The scalar Rajat11 matrix is 135×135 with 665 non-zeros, the details of the matrix are summarized in Table 7.5 with the sparsity pattern is shown in Figure 7.4. The Directed Acyclic Graph depicting the column dependencies of the matrix is captured in Figure 7.5. After performing parallel analysis on the DAG, the parallel schedule presented in Table 7.6 is produced.

Dimension	135×135
$\mathcal{J}$ Non-zeros	665 blocks
$\mathcal{L}$ Non-zeros	358 blocks
$\mathcal{U}$ Non-zeros	497 blocks
Longest Column in $\mathcal{L}$ (L_{max})	5 blocks
Longest Column in combined LU factor (LU_{max})	47 blocks
Longest Column in $\mathcal{J}$ (J_{max})	30 blocks

Table 7.5: Rajat11 Matrix Details

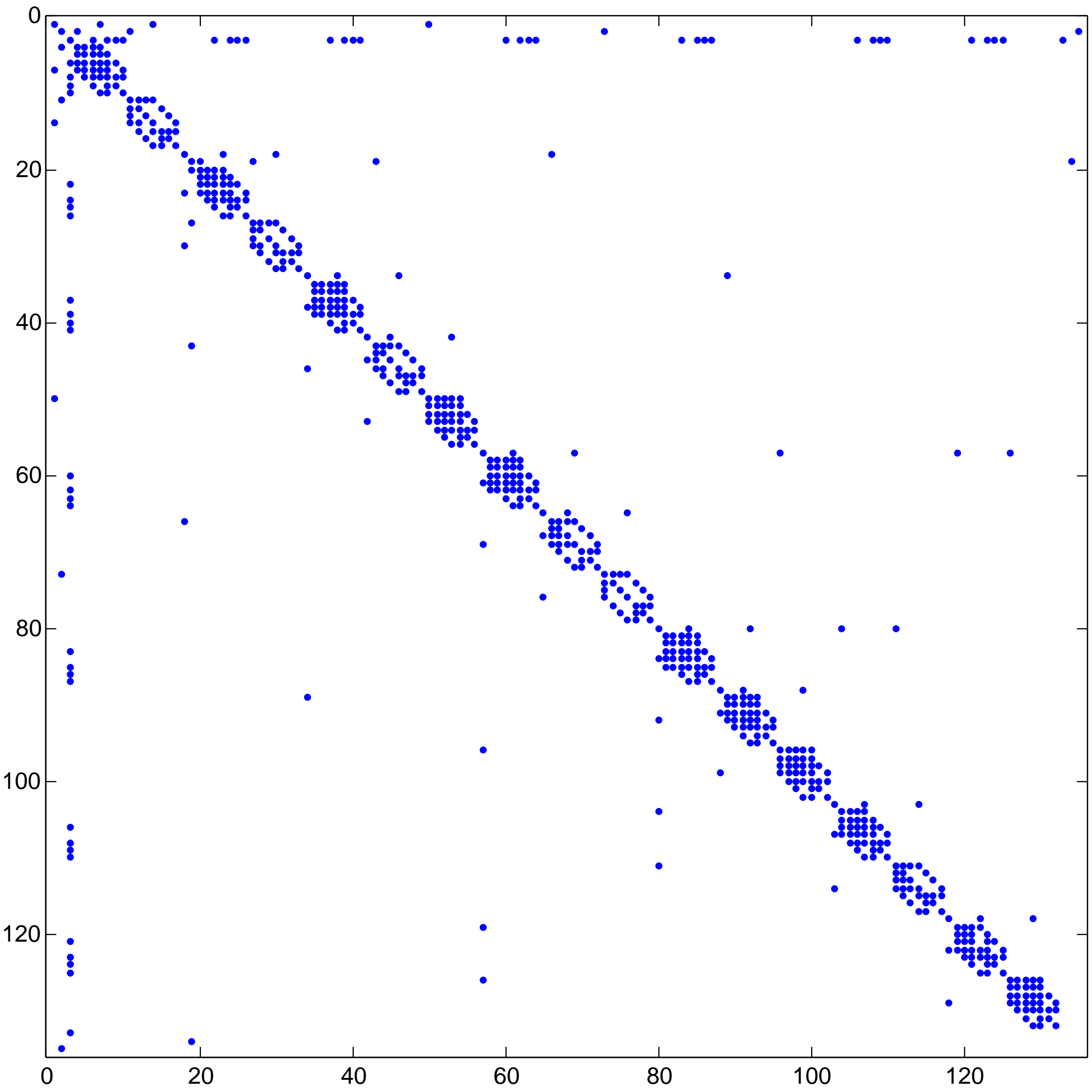


Figure 7.4: Rajat11 Sparsity Pattern

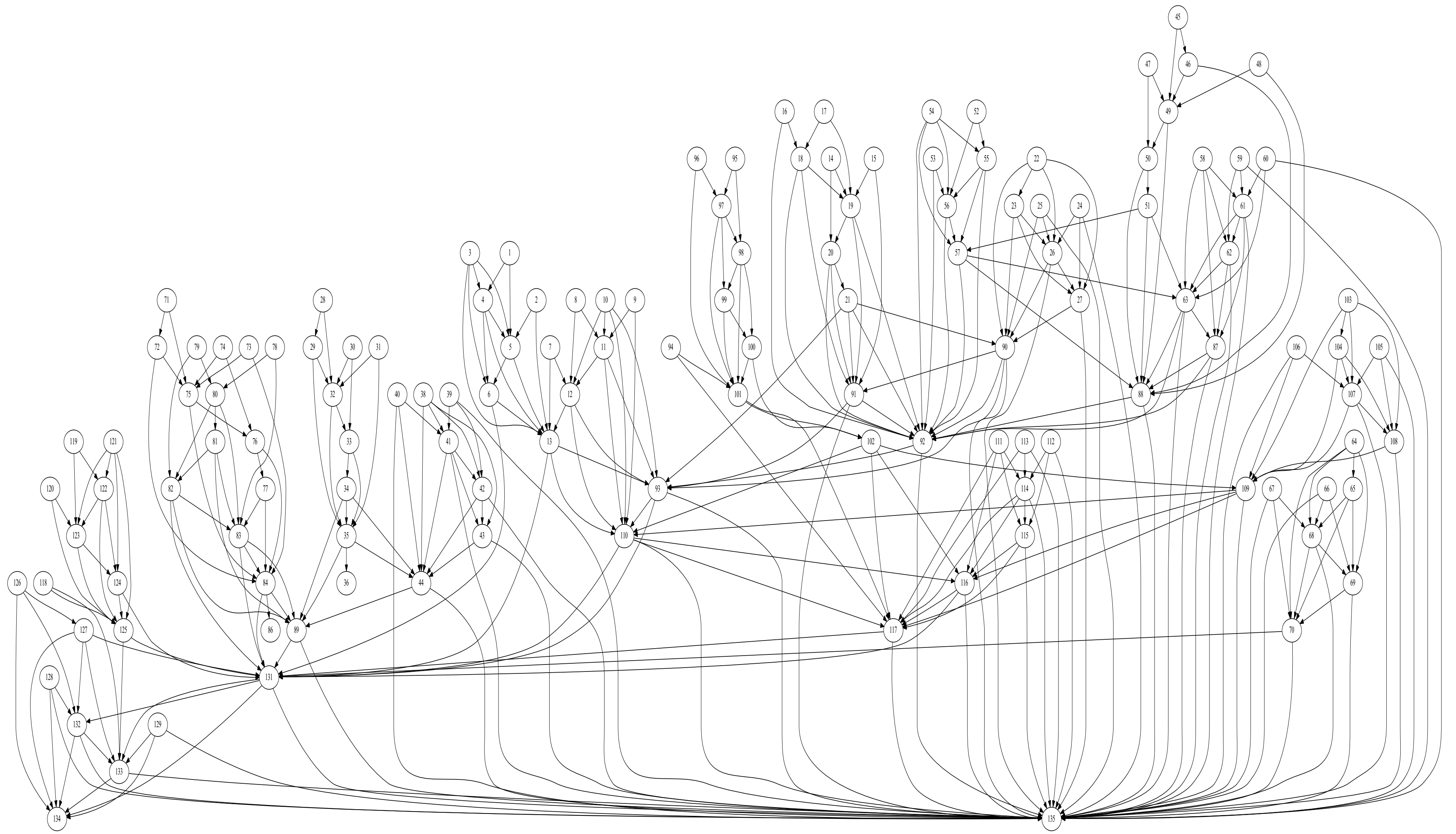


Figure 7.5: Rajat11 Dependency Graph

Level	Columns Assigned to GPU1	Columns Assigned to GPU2
0	0, 1, 2, 6, 7, 8, 9, 13, 14, 15, 16, 21, 23, 24, 27, 29, 30, 36, 37, 38, 39, 44, 46, 47, 51, 52, 53, 57	58, 59, 63, 65, 66, 70, 72, 73, 77, 78, 84, 93, 94, 95, 102, 104, 105, 110, 111, 112, 117, 118, 119, 120, 125, 127, 128, 129
1	3, 10, 17, 22, 28, 40, 45, 54, 60	64, 71, 79, 96, 103, 113, 121, 126
2	4, 11, 18, 25, 31, 41, 48, 55, 61	67, 74, 80, 97, 106, 114, 122
3	5, 19, 26, 32, 42, 49	68, 75, 81, 98, 107, 123
4	12, 20, 33	50, 69, 76, 99, 124
5	34, 56, 82	89, 100
6	35, 43, 62	83, 90, 101
7	85, 86	88, 108
8	87	-
9	-	91
10	92	-
11	-	109
12	115	-
13	-	116
14	130	-
15	-	131
16	132	-
17	133	134

Table 7.6: Parallel Schedule for the Rajat11 Matrix

The factorization is performed and yields the results shown in Table 7.7.

Block Size	512×512
$\mathcal{J}$ Buffer Size	30
$\mathcal{J}$ Buffer Memory Management Mode	Column Transfer
$\mathcal{L}$ Buffer Size	358
$\mathcal{L}$ Buffer Memory Management Mode	None
$\mathcal{U}$ Buffer Size	497
$\mathcal{U}$ Buffer Memory Management Mode	None
Total Factorization Time	2.46 seconds
Number of memory transfers between devices	70
Memory Usage	1,866 MB

Table 7.7: Rajat11 matrix factorization details

Eleven blocking conditions are encountered during the factorization (Table 7.8). With a total factorization time of 2.46 seconds, the GPU idle time represents an extremely small fraction of the overall time.

#	Thread/GPU	CPU (sec)	GPU (sec)	Column	Unfinished Dependency
①	0	0.24	0.036	92	91
②	0	0.084	0.038	115	109
③	0	0.076	0.0064	130	116
④	0	0.075	0.026	132	131
⑤	1	0.045	0.0	91	62
⑥	1	0.042	0.034	91	86
⑦	1	0.090	0.019	91	87
⑧	1	0.095	0.031	109	92
⑨	1	0.095	0.029	116	115
⑩	1	0.081	0.052	131	130
⑪	1	0.061	0.018	133	132
GPU0 Total Idle: 0.1064s			GPU1 Total Idle: 0.183s		

Table 7.8: Overview of the amount of time the CPU and GPU spend waiting on synchronization events during the factorization of the Hamrle1 matrix

7.4 Zero Column Transfer on Rajat11

The Zero Column Transfer analysis reduces the rajat11 $\mathcal{L}$ buffer to 132 and 109 blocks on the first and second GPU respectively. The factorization is summarized in Table 7.9, note that there are no extra memory transfers resulting from ZCT. The discrepancy in speedup versus non-ZCT can be found in the time it takes to evict a column, allowing the space it occupied to be overwritten. Before the space is overwritten in GPU memory, the column must have already been transfered back to the host, in the event that the transfer is in progress or has not yet begun, the CPU thread must wait. In this factorization, the CPU waits for 0.72 seconds on Thread 0 and 0.031 seconds in Thread 1, the performance impact of ZCT on the overall factorization time is an increase of only 0.06 seconds. Two other benchmark matrices are profiled with ZCT and the results are summarized in Table 7.10 and Figure 7.6. The results show that the zero column transfer method has little impact on the overall performance of the factorization while significantly reducing the device memory required for $\mathcal{L}$.

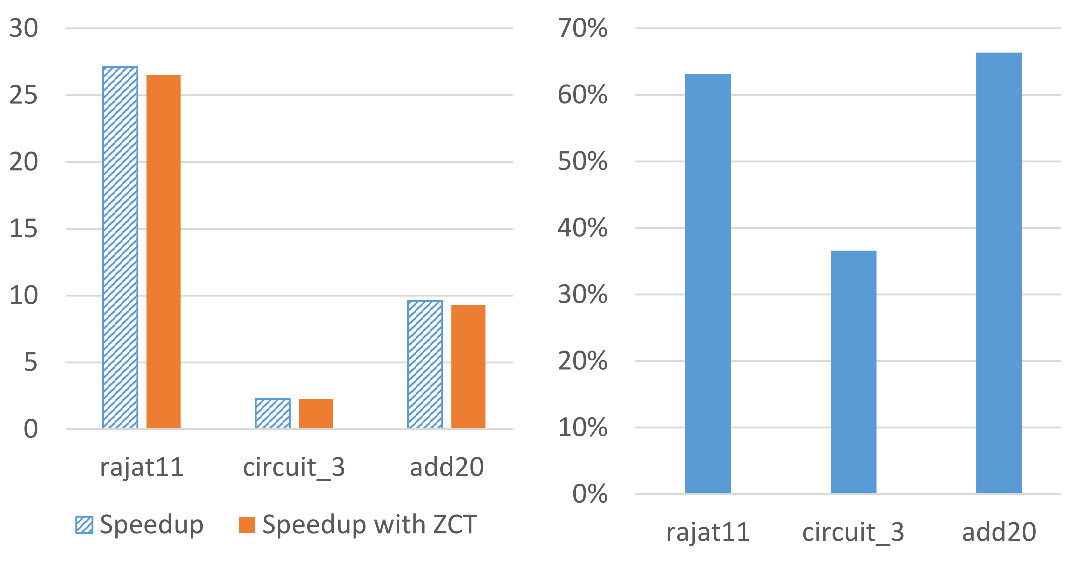
Block Size	512 × 512
$\mathcal{L}$ Buffer Size	132
$\mathcal{L}$ Buffer Memory Management Mode	ZCT
Total Factorization Time	2.52 seconds
Number of memory transfers between devices	70
Memory Usage	1,414 MB

Table 7.9: Rajat11 matrix factorization details

					Evict Time (ms)	
	$\boldsymbol{L_{NNZ}}$	$\boldsymbol{L_{ZCT}}$	Speedup$^{\alpha}$	Speedup$^{\beta}$	T0	T1
Rajat11*	358	132	27.1	26.5	720	31
Circuit_3†	26,591	16,859	2.25	2.24	0	0.699
Add20‡	7,383	2,485	9.6	9.3	54	0.43

* Block size of 512 × 512, $\mathcal{L}$ was reduced by 452MB
† Block size of 96 × 96, $\mathcal{L}$ reduced by 684.28MB
‡ Block size of 192 × 192, $\mathcal{L}$ reduced by 1.35GB
$^{\alpha}$ Without ZCT versus 6 thread BKLU-CPU
$^{\beta}$ With ZCT versus 6 thread BKLU-CPU

Table 7.10: OCR profiling for three benchmark matrices



(a) Comparison of speedup (b) Percent decrease in $\mathcal{L}$ buffer size

Figure 7.6: Summary of ZCT performance results

7.5 Optimal Column Replacement on Rajat11

Each GPU in the test system is equipped with 3GB of RAM, allowing the entire Rajat11 problem to fit into memory at a block size of 512×512. However, as a first scenario assume that the factorization of Rajat11 will be carried out on two GPUs each with a main memory of 1.2GB. ZCT was able to reduce the $\mathcal{L}$ buffer from 358 blocks to 132, with the total problem requiring 1.38GB of GPU memory. By only modifying the $\mathcal{L}$ buffer, using OCR with a size of 30 blocks gives a device memory usage of 1.18GB and allows the problem to fit within the new device's memory constraints. The details of the

Block Size	512×512
$\mathcal{J}$ Buffer Size	30
$\mathcal{J}$ Buffer Memory Management Mode	Column Transfer
$\mathcal{L}$ Buffer Size	30
$\mathcal{L}$ Buffer Memory Management Mode	OCR
$\mathcal{U}$ Buffer Size	497
$\mathcal{U}$ Buffer Memory Management Mode	None
Total Factorization Time	2.60 seconds
Number of memory transfers between devices	224
Memory Usage	1,210 MB

Table 7.11: Rajat11 matrix factorization details scenario one

factorization are given in Table 7.11 with the total factorization time taking 2.60 seconds, compared to 2.46 seconds (from Table 6.2) without any memory optimization. The number of memory transfers is bumped up from 70 to 224, a 220 percent increase. However comparing with the execution time of the multi-threaded CPU implementation, the speedup when using OCR drops insignificantly from 27.09 times to 25.69 times. The end result is a degradation in speedup of 5.17 percent, but resulting speedup is still enough to justify employing a memory technique so that the problem can be solved using the GPUs.

In a second scenario, the $\mathcal{L}$ buffer is further reduced to 12 blocks, the memory required for the problem becomes 1.15GB. The relevant details of this factorization are presented in Table 7.12. The number of transfers increase to 298 while the speedup drops from 27.09 to 24.92. To further profile OCR, there are three situations which cause the execution to be delayed, they are as follows.

- the overhead of the additional transfers;

$\mathcal{L}$ Buffer Size	12
$\mathcal{L}$ Buffer Memory Management Mode	OCR
Total Factorization Time	2.68 seconds
Number of memory transfers between devices	298
Memory Usage	1,174 MB

Table 7.12: Rajat11 matrix factorization details for scenario two

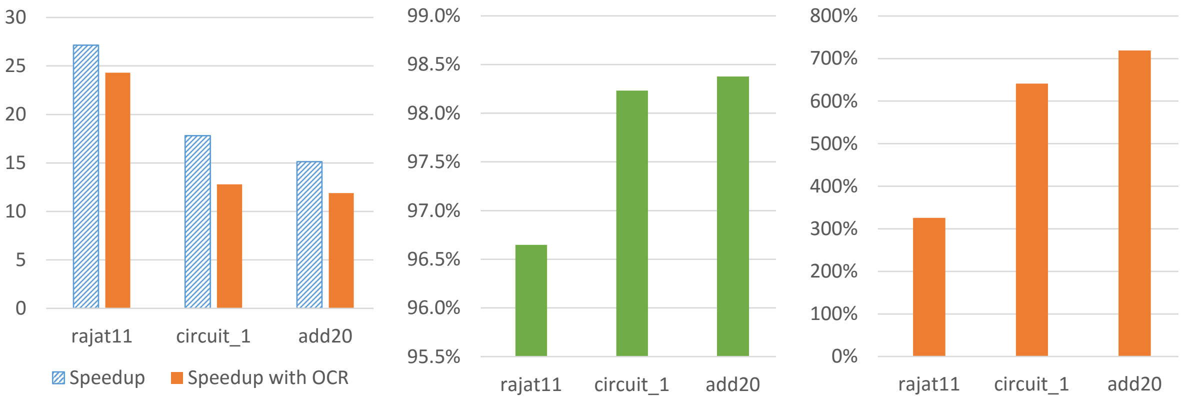
- the CPU time it takes to determine the optimal column to remove and;
- the time waiting for a free space in $\mathcal{L}$ on the GPU, which is typically less than or equal to the time it takes to transfer a column back to the host.

Similar to the case with ZCT, the slow-down associated with free space in $\mathcal{L}$ only occurs when the column chosen for eviction has not yet been transferred to the host, the transfer is either queued or in progress. The column cannot be overwritten until it has been synchronized with the host. In the factorization of rajat11 with an $\mathcal{L}$ buffer size of 12, the CPU time required for all OCR related computations was measured to be 0.298 milliseconds for thread 0 and 0.332 milliseconds for thread 1. The time required to wait for free space in $\mathcal{L}$ is 1.15 seconds and 1.14 seconds respectively for threads 0 and 1. Table 7.13 show the profiling of extreme cases of OCR on three benchmark matrices, the latter two matrices have their $\mathcal{L}$ buffers heavily reduced by 5.6GB and 3.55GB respectively. It is clear that the sheer number of additional bus transfers required to accommodate OCR is the largest penalty, as the CPU resources required are relatively low compared to the total execution time in all three matrices. However, even with the reduction in speedup caused by OCR, it is still highly beneficial by maintaining a significant speedup over the CPU-only factorization methods.

						OCR Time (ms)		Evict Time (s)	
	L_{NNZ}	L_{OCR}	Tx$^{\alpha}$	Speedup$^{\beta}$	Speedup$^{\gamma}$	T0	T1	T0	T1
Rajat11*	358	30	224	27.09	25.69	-	-	-	-
Rajat11*	358	12	298	27.09	24.92	0.298	0.332	1.15	1.14
Circuit_1†	20,375	360	16,377	17.8	12.8	189	165	9.85	12.31
Add20‡	7,383	120	5,503	15.13	11.90	61.5	50.94	12.16	7.27

* with block size of 512×512, $\mathcal{L}$ was reduced by 692 MB
† with block size of 192×192, $\mathcal{L}$ reduced by 5.6 GB
‡ with block size of 256×256, $\mathcal{L}$ reduced by 3.55 GB
$^{\alpha}$ Additional transfers caused by OCR
$^{\beta}$ Without OCR versus 6 thread BKLU-CPU
$^{\gamma}$ With OCR versus 6 thread BKLU-CPU

Table 7.13: OCR profiling for three benchmark matrices



(a) Comparison of speedup (b) Percent decrease in $\mathcal{L}$ buffer size (c) Percent increase in bus transfers

Figure 7.7: Summary of OCR performance results

7.6 Workspace Swapping on Hamrle1

The Hamrle1 matrix has a workspace buffer size of 10 blocks, which is determined by the largest column in the combined LU matrix. At a block size of 2048×2048, a constant 320MB of workspace storage is reserved on each GPU. Using workspace swapping the requirement of 10 blocks is reduced to 2 blocks using 64MB of GPU memory. Workspace swapping causes a much more significant drop in performance compared to the other memory methods due to the sheer increase of bus transfers and synchronization of dirty workspace blocks back to the host. The details of the factorization are shown in Table 7.14. The factorization of the Hamrle1 matrix at this block size uses ZCT with a speedup of 117.2 times over the parallel CPU. With workspace swapping enabled this speedup drops to 85.9. While the drop in performance is not insignificant, it reduced the workspace memory requirements by a factor of 5 while still achieving a sizable speedup over the CPU. This trend is similar for the matrices summarized in Table 7.15, where OBR time indicates the CPU time used to select a workspace block for eviction and evict time is the CPU time taken to perform the eviction for each thread.

7.7 Performance Impact of Optimal Column Replacement

This section briefly touches on the performance impact associated with the Optimal Column Replacement method. OCR is not without its impact, Figure 7.9 shows the number of transfers relative to the $\mathcal{L}$ buffer size augmented with the impact on speedup achieved with a single GPU over the CPU for the

Block Size	2048 × 2048
$\mathcal{J}$ Buffer Size	0
$\mathcal{J}$ Buffer Memory Management Mode	None
$\mathcal{L}$ Buffer Size	60
$\mathcal{L}$ Buffer Memory Management Mode	None
$\mathcal{U}$ Buffer Size	1
$\mathcal{U}$ Buffer Memory Management Mode	None
$\mathcal{X}$ Buffer Size	2
Total Factorization Time	7.77 seconds
Number of memory transfers between devices	162
Memory Usage	2,048 MB

Table 7.14: Hamrle1 matrix factorization with workspace swapping

						OBR Time (ms)		Evict Time (s)	
	$\boldsymbol{X_{NNZ}}$	$\boldsymbol{X_{WS}}$	Tx^{α}	Speedup^{β}	Speedup^{γ}	T0	T1	T0	T1
Rajat11*	47	2	1,463	65.34	46.72	5.1	13.8	2.96	3.61
Hamrle1‡	10	2	146	117.2	85.9	0.15	0.14	1.75	1.58

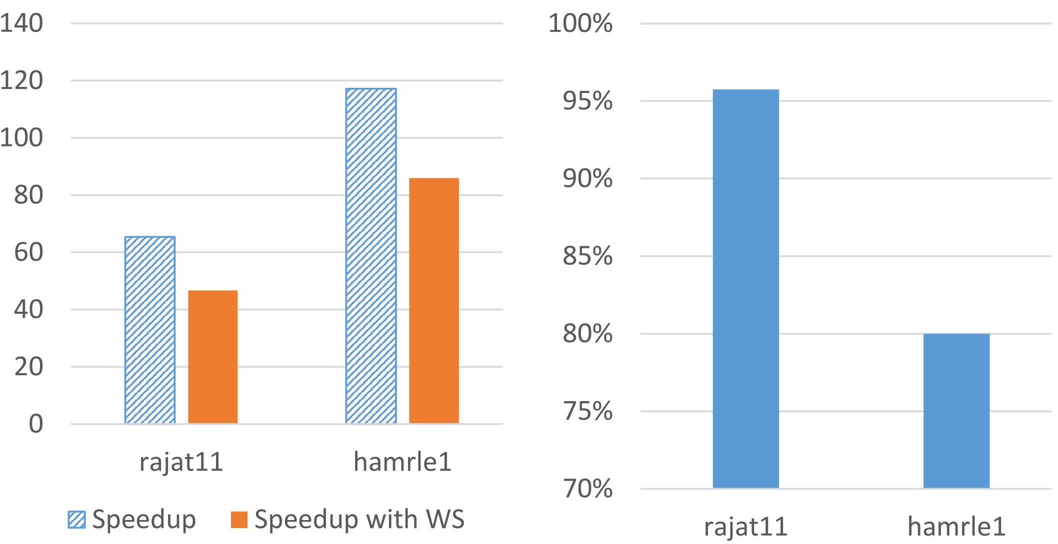
* with block size of 1024 × 1024, $\mathcal{X}$ was reduced by 360 MB
† with block size of 2048 × 2048, $\mathcal{L}$ reduced by 256 MB
$^{\alpha}$ Additional transfers caused by workspace swapping
$^{\beta}$ Without workspace swapping versus 6 thread BKLU-CPU
$^{\gamma}$ With workspace swapping versus 6 thread BKLU-CPU

Table 7.15: Workspace Swapping profiling for two benchmark matrices

Rajat05 matrix $L_{peak} = 300$ blocks and $L_{max} = 6$ block. A buffer size of L_{peak} produces zero PCI transfers and yields a speedup of nearly 6.5 times compared to the minimum L_{max} which produces 456 additional transfers with the speedup dropping to 6.16 times. Figure 7.10 shows the number of transfers OCR causes compared to a random replacement policy. At each buffer size, OCR produces significantly less transfers as a result of eviction choices, up to 3,319 less for a buffer size of 50 blocks.

7.8 Summary

In this chapter, a detailed profile was presented for parallel matrix factorization on two GPUs, and for each BKLUX memory mode. It was shown that while the host threads can block on data dependencies, the OpenCL device typically blocks for much less time and continues processing previously issued work.



(a) Comparison of speedup (b) Percent decrease in $\mathcal{X}$ buffer size

Figure 7.8: Summary of Workspace swapping performance results

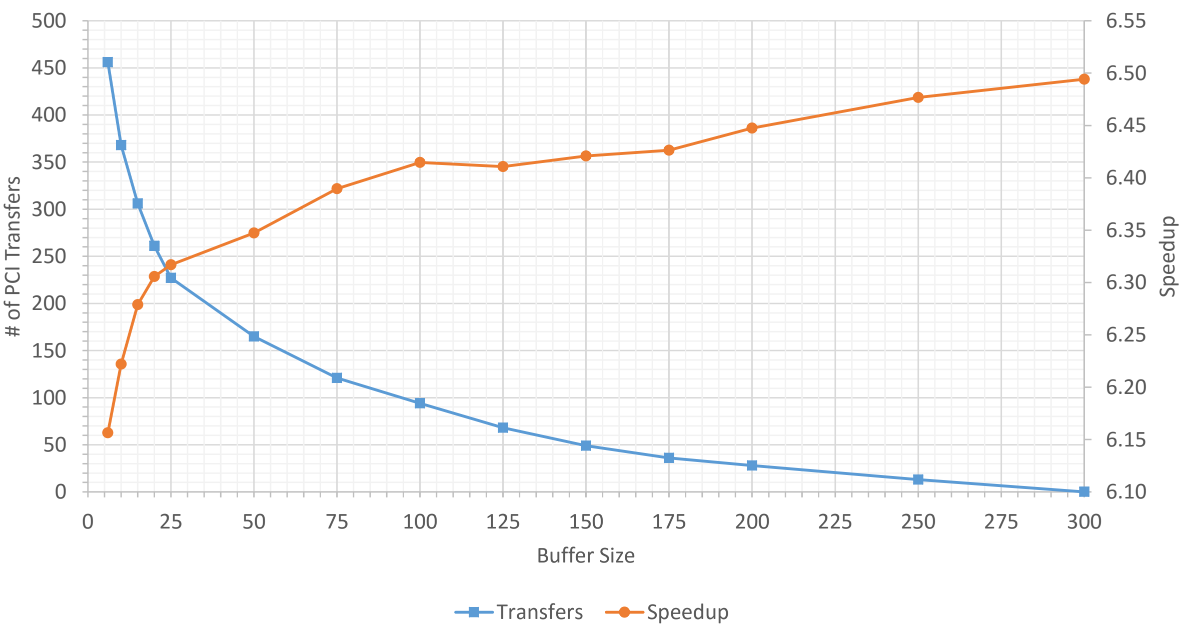


Figure 7.9: Optimal Column Replacement transfers and speedup versus CPU for various L buffer size. Rajat05 Matrix with block size 320×320

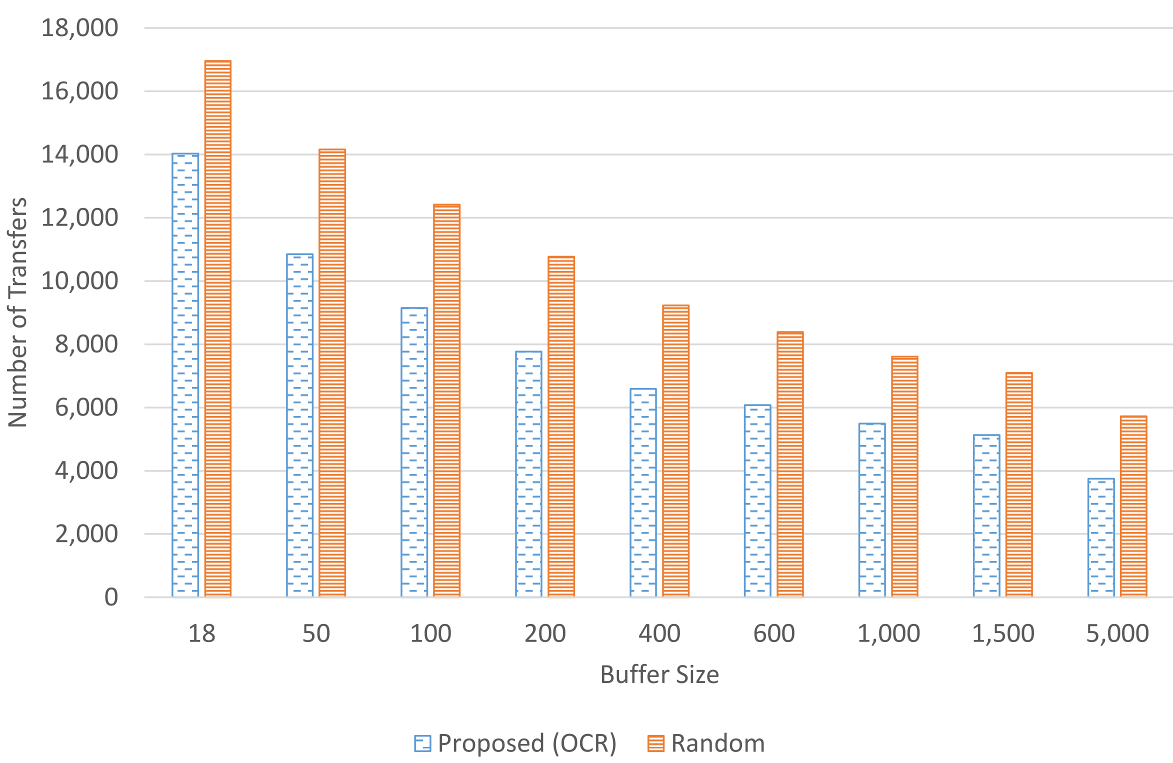


Figure 7.10: Random Column Replacement PCI transfers compared to OCR. Rajat05 Matrix with block size 320×320

Chapter 8

Challenges of Small Block Factorization

8.1 Introduction

There are significant hurdles to accelerating block-wise LU factorization on GPUs and other parallel accelerators where the block size, M, is relatively small. The massively parallel architectures of OpenCL devices rely on high levels of data parallelism to achieve optimal performance. This chapter is dedicated to providing insights into why this is the case, and to determine a practical minimum block size for the AMD 7950 GPU.

In block-wise LU factorization, the blocks of the $\boldsymbol{\mathcal{L}}$ factor must be divided by the pivot block, which is typically dense. Once the pivot's LU factors have been found, there are two possible ways to update the column of $\boldsymbol{\mathcal{L}}$, either through matrix multiplication with inverse of the pivot, or by solving the system of equations using forward/backward substitution. The previous work [6] opted for direct multiplication with the inverse while this work favors forward/backward substitution due to the enhanced numerical stability of the latter. In either case, the LU factor of the pivot block itself must be computed. For small block sizes, computing the pivot block's LU factor on parallel architectures such as GPUs presents a challenge. Additionally, the pivot decomposition operation can only be started after all of the blocks in a column are updated by columns to the left during the Forward Solve step of the KLU algorithm;

no further operations can be performed until the pivot's LU factor is computed. This presents a unique performance issue directly related to block size.

8.2 Naïve Gaussian Elimination

One method for decomposition the pivot blocks is Naïve Gaussian Elimination, which assumes that the matrix is positive-definite or diagonally dominant and thus does not perform any pivoting. The decomposition is performed over the columns of the matrix with $(n-1)$ steps, where n is the dimension of the matrix. At the kth step, $A_{ij} \leftarrow A_{ij} - A_{ik} \times A_{kj}$ where the dimension of A_{ij} is $(n-k-1) \times (n-k-1)$.

The algorithm iterates through the columns of the matrix sequentially using the results from columns 0 to $k-1$ in generating column k. At any given column k, the number of entries which can be updated in parallel is $(n-k-1) \times (n-k-1)$.

As Figure 8.1 shows, assuming $k = 20$ and $N = 32$, each entry in the shaded 12×12 area can be updated in parallel. After completing column k, the following column $k+1$ will have less work to do in parallel, ultimately until the last column has no work which can be done in parallel. GPU devices are designed for massively parallel algorithms, the shortage of parallel work in the LU decomposition of small blocks results in inefficient use of the hardware. Experimental results have shown that for block sizes between 16×16 and 96×96, the parallel Gaussian Elimination routine performs better than the LU method adopted in [6]. As such, the performance of parallel Gaussian Elimination on the AMD 7950 will be explored in the next section.

8.3 Performance of Naïve Gaussian Elimination on GPU

In the overall block-aware matrix factorization, three matrix operations dominate the computation time:

1. Multiply-Accumulate-Subtract (MAC): Updates each block of column k with columns to the left.
2. LU Factorization: Factorizes the pivot block.
3. Triangular Solve: Solves the system $A_{piv}X = L(k)$ to update block k of the $\mathcal{L}$ factor by the pivot block.

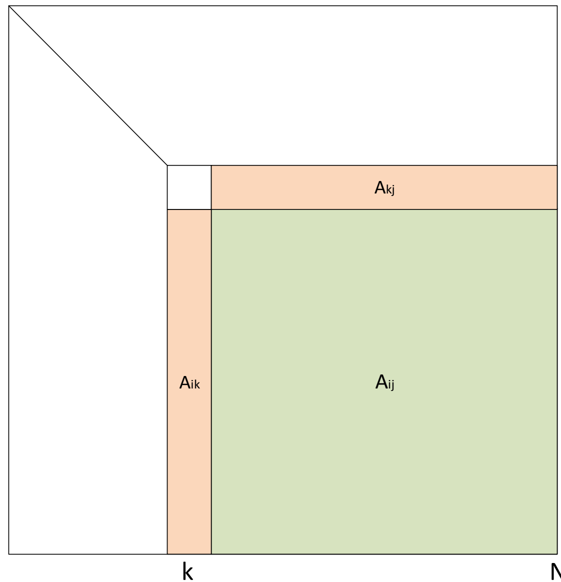


Figure 8.1: Naive Gaussian Elimination Example

The operations above are implemented using LAPACK and BLAS on the CPU and clBLAS [25] provided by AMD on the GPU. Table 8.1 shows the library routines used for each matrix operation. The execution time for each of the three operations are shown in Table 8.2. With a matrix size of 32×32, the LU decomposition operation is 5.84 times slower on GPU than CPU while the triangular solve is 2.37 times slower on the GPU. The MAC operation is the only one in which the GPU outperforms the CPU with a speedup of 3.34 times. At a matrix size of 64×64, all of the operations perform better on the GPU with the exception of LU Factorization, which performs close to two times slower on the GPU compared to the CPU.

	Implementation	
	CPU	GPU
Multiply-Accumulate-Subtract	BLAS GEMM	clBLAS GEMM
LU Decomposition	LAPACK DGETRF	AMD App SDK Gaussian Elimination
Triangular Solve	LAPACK DGETRS	clBLAS TRSM

Table 8.1: Test system math libraries

Figure 8.2 shows the execution time of the LU operation sequentially on CPU and in parallel on the GPU for varying block sizes. The GPU execution time matches the CPU only after a matrix size of 144×144 with the speedup growing from there onward.

	32×32 Matrix			64×64 Matrix		
	CPU (ms)	GPU (ms)	Speedup	CPU (ms)	GPU (ms)	Speedup
MAC	0.05558	0.01666	+3.34	0.4783	0.02861	+16.72
LU Decomposition	0.02954	0.1725	-5.84	0.1782	0.3539	-1.99
Triangular Solve	0.1506	0.3562	-2.37	1.0434	0.5335	+1.96

Table 8.2: Performance comparison of matrix operations a 32×32 and 64×64 dense matrix using test system

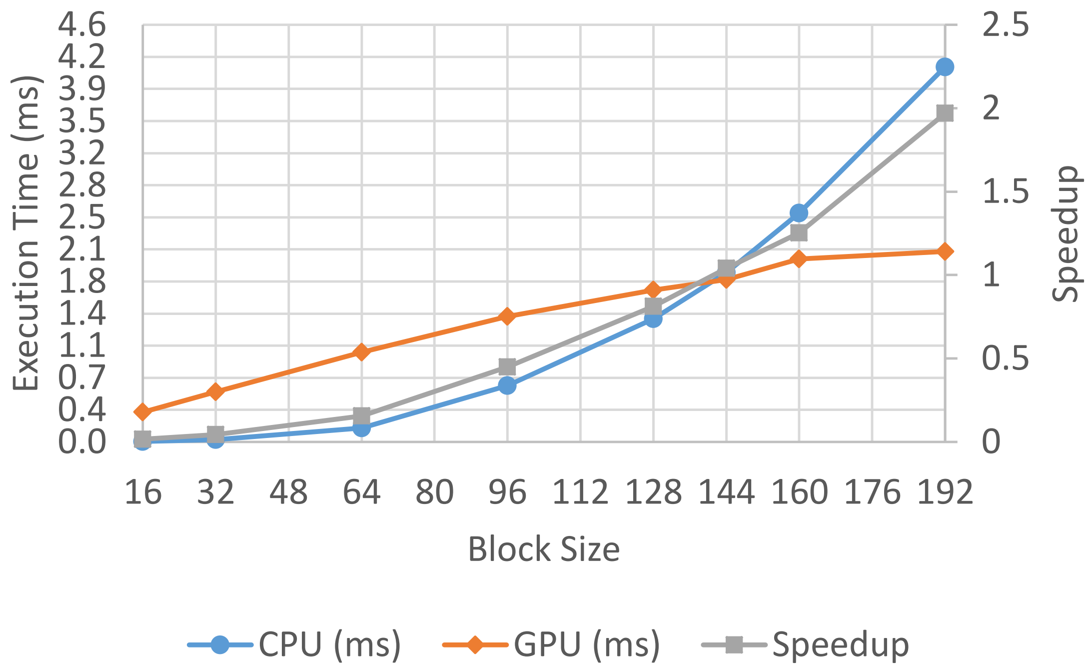


Figure 8.2: Execution time and speedup for LU decomposition on the CPU and GPU

The performance of Gaussian Elimination on the GPU can be anticipated by considering the amount of parallel work available for the desired block size. In OpenCL, the number of available wavefronts can be used as a gauge for the suitability of the block to be decomposed using the parallel algorithm. Figure 8.3 shows the number of wavefronts available for execution at each stage of the decomposition of a 32×32 dense matrix. The peak number of wavefronts in this example is four, which decreases to two wavefronts before the decomposition is half way complete. From column $k = 17$ to $k = N$, there is only enough parallel work for a single wavefront. The AMD GCN Compute Unit requires, at a minimum, four ready-state wavefronts to effectively hide the memory access latency [16]. It is expected that for problem sizes which are unable to provide sufficient work, the GPU performance will be degraded. A compute unit on the AMD 7950 can handle 40 active wavefronts, with only four, $1/10^{\text{th}}$ of the work capacity of the compute unit is utilized. Across the entire device one out of 28 available compute units

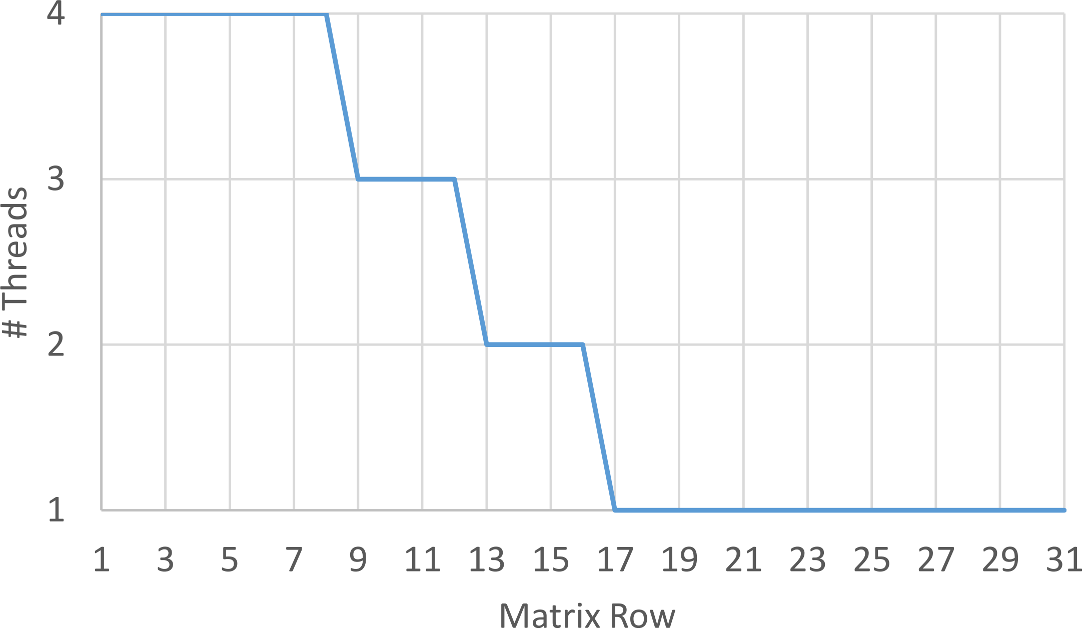


Figure 8.3: Number of wavefronts available during Naive Gaussian Elimination on a 32×32 block

is utilized. In contrast, the decomposition of a 192×192 block (Figure 8.4) yields much more parallel work for the GPU. It is at column $k = 170$ that the number of wavefronts drops below the threshold of four. Note that the variations in the number of wavefronts in Figure 8.4 is due to selecting group sizes optimal for the target hardware with the trade-off of having some idle work-items.

Figure 8.5 shows the kernel execution time for the factorization of each column of four matrices, 32×32, 96×96, 128×128, and 192×192. The inefficiency of small workloads can clearly be seen in comparing the execution times for 32×32 and 192×192 where the lowest execution time of the former is above 12500ns, this is slower than any single execution time for 192×192, even though significantly more work is done in each column.

8.4 Summary

In this chapter, it was shown that the GPU factorization of matrices does not always outperform CPU based factorization. Since parallel architectures, like those found in GPUs, require a sufficient workload to perform in parallel, small block sizes can negatively affect the performance of the factorization.

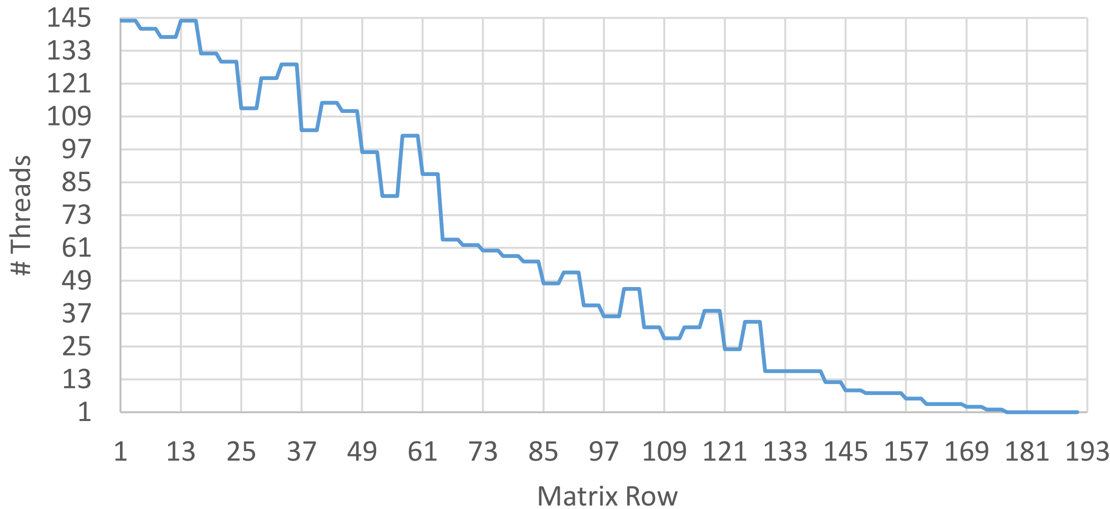


Figure 8.4: Number of threads available during Naive Gaussian Elimination on a 192×192 block

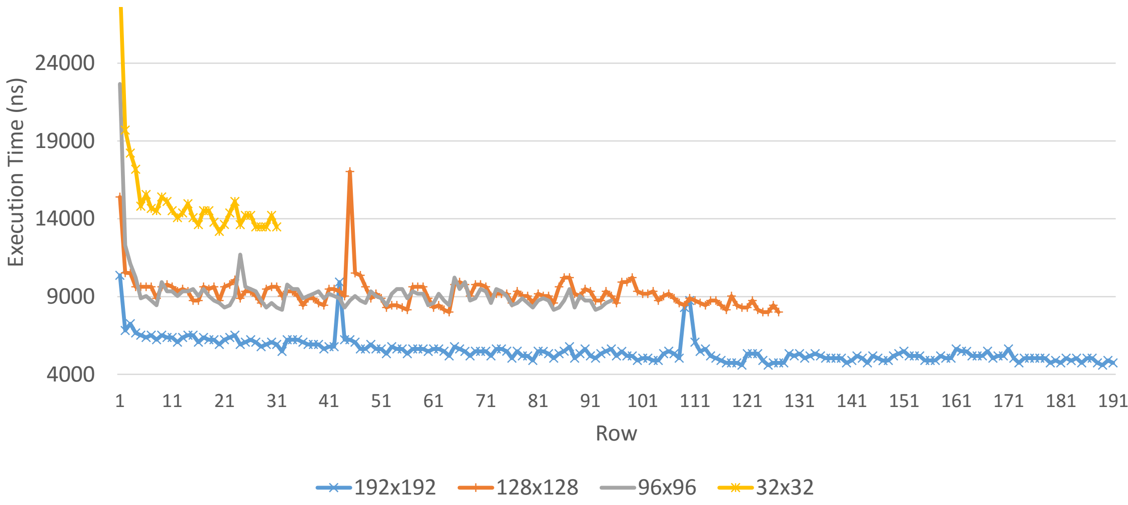


Figure 8.5: GPU pivot factorization time for various block sizes

Chapter 9

Conclusion

In conclusion, this thesis set out to achieve the main goal of enabling multiple OpenCL devices to participate in the factorization of general block-structured matrices. In doing so, problems arise which are inherent to the parallelization of the algorithm on heterogeneous platform where each OpenCL device has its own discrete memory. The problems encountered relate to ensuring efficient communication between multiple OpenCL devices, synchronization between an OpenCL device and the host, and finally effectively using of each OpenCL device's on-chip memory.

As a result, this thesis produced a generalized approach for the factorization of block-structured matrices which leverages the massively parallel performance of multiple OpenCL accelerators. The approach taken is general, in the sense that, it can be used in conjunction with any devices which support OpenCL, whether that is an AMD or Nvidia GPU, an Altera FPGA, or some customized hardware.

Also demonstrated was two main memory strategies, firstly, optimal memory swapping, termed Optimal Column Replacement and Optimal Block Replacement, which handles data transferred between the host on demand and is controlled by the amount of memory resources available on the device. Secondly, an innovative approach, Zero Column Transfer, was introduced to handle device memory resources based on an analysis of the matrix dependency graph.

The overall results show an almost optimal speedup when using two AMD GPUs with speedups reaching 131x as compared to CPU-based parallel factorization.

9.1 Future Work

In addition to further optimization of the algorithm, there are many other challenges and research avenues which this work opens the door to. Some of those opportunities are outlined below.

1. The computer system used to generate results (described in Table 6.1 from Chapter 6) was unable to accommodate more than two GPUs. As a result, there was a limited ability to verifying the algorithm's scalability beyond two OpenCL devices. Future work is encouraged to test the scalability at greater device numbers, such as 6 or 8 GPUs.

2. A stretch goal of the work presented in this thesis was to incorporate MPI, which would enable multiple distributed nodes equipped with one or more OpenCL devices to participate in the factorization. This would enable the algorithm to run on some of the most advanced supercomputers which feature high performance GPUs on each node. Unfortunately this goal was not reached, and the area is ripe for further investigation.

3. In this thesis, independent columns were split between devices as evenly as possible and linearly, meaning a grouping of columns, say, [1,2,3,4,6,7,9,10], all belonging to the same level, would be split between two devices as [1,2,3,4] for the first device and [6,7,9,10] for the second. This is not the optimal division of work, as a result of this choice, future unnecessary transfers may be required to synchronize dependencies between devices. further investigation into an optimal strategy is warranted.

4. The Khronos Goup released OpenCL 2.0 in 2013, there are currently a limited number of devices on the market which support the full OpenCL 2.0 standard. There are many new features in 2.0, such as shared virtual memory and dynamic parallelism. The incorporation of these new features into the work done in this thesis is likely to provide better results and more efficient communication between devices.

Bibliography

[1] T. A. Davis and Y. Hu, "The University of Florida sparse matrix collection," *ACM Trans. Math. Softw.*, vol. 38, pp. 1:1–1:25, Dec. 2011.

[2] K. Kundert, "Simulation methods for RF integrated circuits," in *Computer-Aided Design, 1997. Digest of Technical Papers., 1997 IEEE/ACM International Conference on*, pp. 752–765, Nov 1997.

[3] E. Gad, R. Khazaka, R. S. Nakhla, and R. Griffith, "A circuit reduction technique for finding the steady-state solution of nonlinear circuits," *IEEE Transactions on Microwave Theory and Techniques*, vol. 48, pp. 2389–2396, Dec 2000.

[4] Y. Saad, *Iterative Methods for Sparse Linear Systems.* Philadelphia, PA, USA: Society for Industrial and Applied Mathematics, 2nd ed., 2003.

[5] 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), pp. 370–375, ACM, 2009.

[6] B. Bandali, E. Gad, and M. Bolic, "Accelerated harmonic-balance analysis using a graphical processing unit platform," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 33, pp. 1017–1030, July 2014.

[7] T. A. Davis and E. Palamadai Natarajan, "Algorithm 907: KLU, a direct sparse solver for circuit simulation problems," *ACM Trans. Math. Softw.*, vol. 37, pp. 36:1–36:17, Sept. 2010.

[8] C.-W. Ho, A. Ruehli, and P. Brennan, "The modified nodal approach to network analysis," *IEEE Transactions on Circuits and Systems*, vol. 22, pp. 504–509, Jun 1975.

[9] C. Gear, "Simultaneous numerical solution of differential-algebraic equations," *IEEE Transactions on Circuit Theory*, vol. 18, pp. 89–95, Jan 1971.

[10] J. M. Ortega and W. C. Rheinboldt, *Iterative Solution of Nonlinear Equations in Several Variables.* Computer science and applied mathematics, New York, NY: Academic Press, 1970.

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

[12] K. Strunz and Q. Su, "Stochastic formulation of SPICE-type electronic circuit simulation with polynomial chaos," *ACM Trans. Model. Comput. Simul.*, vol. 18, pp. 15:1–15:23, Sept. 2008.

[13] C. F. Dunkl and Y. Xu, *Orthogonal Polynomials of Several Variables.* Encyclopedia of Mathematics and its Applications, Cambridge, 2001.

[14] M. Flynn, "Some computer organizations and their effectiveness," *Computers, IEEE Transactions on*, vol. C-21, pp. 948–960, Sept 1972.

[15] Khronos OpenCL Working Group, "The opencl specification." `https://www.khronos.org/registry/cl/specs/opencl-1.2.pdf`, 2012.

[16] Advanced Micro Devices Inc., "Amd accelerated parallel processing opencl programming guide." `http://developer.amd.com/wordpress/media/2013/07/AMD_Accelerated_Parallel_Processing_OpenCL_Programming_Guide-rev-2.7.pdf`, 2013.

[17] J. W. Demmel, *Applied Numerical Linear Algebra.* Philadelphia, PA: SIAM Publishers, 1997.

[18] Y. Zhou, E. Gad, M. S. Nakhla, and R. Achar, "Structural characterization and efficient implementation techniques for A-stable high-order integration methods," *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, vol. 31, pp. 101 –108, Jan. 2012.

[19] X. Chen, W. Wu, Y. Wang, H. Yu, and H. Yang, "An escheduler-based data dependence analysis and task scheduling for parallel circuit simulation," *IEEE Transactions on Circuits and Systems II: Express Briefs*, vol. 58, pp. 702–706, Oct 2011.

[20] X. Chen, Y. Wang, and H. Yang, "NICSLU: An adaptive sparse matrix solver for parallel circuit simulation," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 32, pp. 261–274, Feb 2013.

[21] X. Chen, L. Ren, Y. Wang, and H. Yang, "GPU-accelerated sparse LU factorization for circuit simulation with performance modeling," *IEEE Transactions on Parallel and Distributed Systems*, vol. 26, pp. 786–795, March 2015.

[22] 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.

[23] 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, pp. 353–376, Sept. 2004.

[24] W. H. Press, B. P. Flannery, S. A. Teukolsky, and W. T. Vatterling, *Numerical Recipes in C*. New York: Cambridge University Press, 1988.

[25] Advanced Micro Devices, Inc., "clblas." `https://github.com/clMathLibraries/clBLAS`, 2013.

[26] R. Mattson, J. Gecsei, D. Slutz, and I. Traiger, "Evaluation techniques for storage hierarchies," *IBM Systems Journal*, vol. 9, no. 2, pp. 78–117, 1970.

[27] B. Van Roy, "A short proof of optimality for the min cache replacement algorithm," *Inf. Process. Lett.*, vol. 102, pp. 72–73, Apr. 2007.