UNIVERSITY OF SOUTHAMPTON
FACULTY OF PHYSICAL AND APPLIED SCIENCES
Electronics and Computer Science

Parallel Sparse Matrix Solution for Direct Circuit Simulation on a Multiple FPGA System

by

Tarek Nechma

A thesis submitted for the degree of
Doctor of Philosophy

December 2012

UNIVERSITY OF SOUTHAMPTON

ABSTRACT

FACULTY OF PHYSICAL AND APPLIED SCIENCES
Electronics and Computer Science

Doctor of Philosophy

Parallel Sparse Matrix Solution for Direct Circuit Simulation on a Multiple FPGA System

by Tarek Nechma

SPICE, from the University of California, at Berkeley, is the de facto world standard for circuit simulation. SPICE is used to model the behaviour of electronic circuits prior to manufacturing to decrease defects and hence reduce costs. However, accurate SPICE simulations of today's sub-micron circuits can often take days or weeks on conventional processors. In a nutshell, a SPICE simulation is an iterative process that consists of two phases per iteration, namely, model evaluation followed by a matrix solution. The model evaluation phase has been found to be easily parallelisable unlike the subsequent phase, which involves the solution of highly sparse and asymmetric matrices.

In this thesis, we present an FPGA implementation of a sparse matrix solver hardware, geared towards matrices that arise in SPICE circuit simulations. As such, we demonstrate how we extract parallelism at different granularities to accelerate the solution process. Our approach combines static pivoting with symbolic analysis to compute an accurate task flow-graph which efficiently exploits parallelism at multiple granularities and sustains high floating-point data rates. We also present a quantitative comparison between the performance of our hardware protrotype and state-of-the-art software package running on a general purpose PC equipped with a 2.67 GHz six-core 12-thread Intel Core Xeon X5650 microprocessor and 6 GB memory. We report average speedups of 9.65×, 11.83×, 17.21× against UMFPACK, KLU, and Kundert Sparse matrix packages respectively. We also detail our approach to adapt our sparse LU hardware prototype from a single-FPGA architecture to a multi-FPGA system to achieve higher acceleration ratios up to 38× for certain circuit matrices.

Contents

List of Tables

List of Figures

List of Algorithms

Abbreviations

AMD	**A**pproximate **M**inimum Degree
ASIC	**A**pplication **S**pecific **I**ntegrated **C**ircuit
BCE	**B**ranch **C**onstitutive **E**quations
BDB	**B**ordered **D**iagonal **B**lock
BLAS	**B**asic **L**inear **A**lgebra **S**ubprograms
BRAM	**B**lock **R**andom **A**ccess **M**emory
CCS	**C**ompressed **C**olumn **S**torage
CLB	**C**onfigurable **L**ogic **B**lock
CRS	**C**ompressed **R**ow **S**torage
COLAMD	**COL**umn **A**pproximate **M**inimum Degree
DBB	**D**iagonal **B**ordered **B**lock
DRAM	**D**ynamic **R**andom **A**ccess **M**emory
FLOPS	**FL**oating point **O**perations **P**er **S**econd
FPGA	**F**ield **P**rogammable **G**ate **A**rray
LUT	**L**ook-**U**p **T**able
MNA	**M**odified **N**odal **A**nalysis
NNZ	**N**umber of **N**on **Z**eros
RAM	**R**andom **A**ccess **M**emory
SRAM	**S**tatic **R**andom **A**ccess **M**emory
UFMC	**U**niversity of **F**lorida **M**atrix **C**ollection
VLSI	**V**ery **L**arge **S**cale **I**ntegration
VHSIC	**V**ery **H**igh **S**peed **I**ntegrated **C**ircuit
VHDL	**V**HSIC **H**ardware **D**escription **L**anguage

Declaration of Authorship

I, Tarek Nechma, declare that the thesis entitled Parallel Sparse Matrix Solution for Direct Circuit Simulation on a Multiple FPGA System, and the work presented in it are my own, I confirm that:

- this work was done wholly or mainly while in candidature for a research degree at this University;
- where any part of this thesis has previously been submitted for a degree or any other qualification at this University or any other institution, this has been clearly stated;
- where I have consulted the published work of others, this is always clearly attributed;
- where I have quoted from the work of others, the source is always given. With the exception of such quotations, this thesis is entirely my own work;
- I have acknowledged all main sources of help;
- where the thesis is based on work done by myself jointly with others, I have made clear exactly what was done by others and what I have contributed myself;
- parts of this work have been published as listed in Section 1.4 of this thesis.

Signed: _____

Date : _____

Acknowledgements

I would like to thank all those who help me throughout my research. I am highly thankful to my supervisors Prof. Mark Zwoliński and Dr. Jeff Reeve from the School of Electronics and computer Science, University of Southampton, whose help, stimulating suggestions, and encouragement helped me in all the time of research and writing of this thesis. I am particularly indebted to **Prof. Zwoliński** for his unconditional counsel and support throughout a turbulent chapter of my life. I am eternally grateful for his invaluable advice and precious help.

I would also like to express my thanks to all involved in my academic career for influencing me and helping me achieve my goals. I thank my teachers and professors throughout my studies for their support and teaching. Likewise, I would like to thank, in no particular order, Dr Imed Bouchrika, Dr Asma Ounnas, Jurga Puodžiukaitė, Abdeldjalil Belouettar, Dr Issam Maamria, Mohamed Al Tahs Al Salehi, Kheiredine Derouiche, Dr Ahmed Maache, Nadjib Mammeri, Issam Souilah, and Mourad Khelifa, for their emotional support throughout my PhD. I am particularly grateful to my mentors Abdelwaheb and Youcef Djahel who introduced me to the world of electronics and computer science, and without their guidance I would have not followed the path that led me to write this thesis.

Finally, I would like dedicate this work to the memory of my grand parents who passed away whilst writing this thesis. I also thank my family, especially my beloved parents Abdelwaheb and Djahida; and my dear brothers Issam, Chemsedine, and Mohamed Lamine, for their love, and for always supporting and encouraging me.

To my parents and

the everlasting memory of
my grandparents...

Chapter 1

Introduction

1.1 Accelerating SPICE Circuit Simulations

The design of modern Very Large Scale Integration (VLSI) systems requires extensive and exhaustive circuit simulations. A circuit simulator allows a design to be tested and analysed thoroughly with respect to its behaviour and projected targets, prior to committing it to expensive silicon. However, circuit simulation is a computationally demanding task and its complexity grows faster than the number of nodes in the circuit [1], as it will be demonstrated in Section 3.4. Consequently, simulations become dramatically time-consuming and almost impractical with today's multi-million transistor VSLI circuits. Moreover, miniaturisation-induced variations increasingly impact the electrical behaviour of a design. This is often tackled by performing Monte Carlo simulations, resulting in a significant increase in the overall simulation [2]. This highlights further the increasing need to accelerate the circuit simulation kernels.

Simulation Program with Integrated Circuit Emphasis (SPICE) is a widely used circuit simulator that models the analogue behaviour of semiconductor circuits using a non-linear differential equation solver. In essence, a SPICE algorithm is an iterative process that consists of two phases per iteration, namely, model evaluation phase followed by the matrix solution phase. In the first step, a set of non-linear differential equations is

generated, from the layout and the components of the underlying circuit, using modified nodal analysis (MNA) [3]. The equations produced are then discretised and linearised using implicit integration and Newton-Raphsons method respectively. The resulting sparse system is solved, in the matrix solution phase, for the unknown nodal voltages using sparse matrix techniques, such as LU decomposition. The SPICE algorithm will be revisited in more detail in Chapter 3 whereas LU decomposition will be throughly studied in Chapter 4.

SPICE simulations of large sub-micron circuits with can often take days or weeks of runtime on current processors. SPICE simulations are typically infeasible for circuits larger than 20,000 devices [4]. Moreover, SPICE is difficult to parallelise on conventional processors due to its irregular and unpredictable compute structure, modest peak floating-point capacities, and limited memory bandwidth. In effect, it has been observed that less than 7% of the floating-point operations in SPICE are automatically vectorisable [1, 5]. As such, the SPICE algorithm is used in the SPEC92 benchmark collection, which represents a set of challenging problems for microprocessors [6].

Over the past couple of decades, the Electronic Design Automation (EDA) community has relied on on innovations in computer architecture and clock frequency increases to speedup applications such as SPICE. However, the performance gains using these traditional computer organisations have now hit the so-called *"speed wall"*, as it will be explained in Section 2.1. This has led to the adoption of multi-core architectures as a solution to sustain performance increases. This is a clear indication that further performance improvements must be driven by parallelism harnessed at the hardware level. This is further evidenced in the great interest that research community has recently shown in taking advantage of parallel architectures devices, such as FPGA and GPUs, to boost the performance of the current EDA tools [7, 8, 9, 10]. FPGAs also have the advantage of being reconfigurable devices, which enables the creation of custom datapaths and controllers for the problem at hand with the promise of greater performance. On the other hand, programming FPGAs requires specialist knowledge of hardware design techniques and Hardware Description Languages (HDLs). As such, this thesis details

our approach to study the SPICE simulator runtime, identity algorithms that can extract parallelism at the software level, which can be then harnessed at the hardware-level using a multiple Processing Element (PE) parallel architecture.

FPGA-based computing offers the potential for acceleration well beyond Moore's Law improvements in microprocessors. This has led to intensive research to accelerate numerically-intensive algorithms in general and Computer-Aided Design (CAD) related applications, such as the SPICE simulator more specifically [7, 10]. Given the recent advances in FPGA densities and their built-in interconnect technology, a key question to ask is whether a multiple FPGA system can be leveraged to accelerate large circuit simulations. As such, this thesis explores potential ways to achieve the latter.

1.2 Research Scope and Objectives

The SPICE simulator components have varying degrees of inherent control and data parallelism. Consequently, the overall execution time can be improved by parallelising the numerically intensive parts of the simulation process. Therefore, one of the main objectives of this project is to investigate a design methodology for an FPGA accelerator that exploits the inherent parallelism in the SPICE simulator. This involves analysing the SPICE algorithm to identify the key parts most suitable for FPGA implementation in addition to the hardware design and algorithmic related decisions.

However, SPICE simulation runtime analysis shows that for large circuits the matrix solver dominates the overall time [11]. Moreover, the model evaluation phase has been found to be easily parallelisable unlike the matrix solution phase, which involves the solution of a highly sparse, unstructured (i.e. do not follow a particular pattern), and asymmetric matrix [12]. The SPICE runtime will be analysed thoroughly in Chapter 3 (Section 3.4). As such, this thesis will focus on demonstrating how a spatial implementation of the matrix solution phase of the SPICE circuit simulator, can be designed and optimised to leverage the characteristics of circuit simulations matrices to harness a greater degree of parallelism.

In order to sustain performance gains with the ever-increasing matrix sizes, we also investigate algorithmic and hardware decisions that can improve the scalability of our design. Nevertheless, a completely spatial implementation targeting large matrices cannot fit on a single FPGA. Hence, another key objectives of this research project is to look how to our design can span over several FPGAs whilst minimising the communication overhead.

This thesis addresses the following research questions:

- What is the acceleration potential of the SPICE simulator?
- How could we parallelise the matrix solution phase of the SPICE simulator? What are the different degrees of parallelism present in this phase?
- How do we leverage FPGA features to accelerate SPICE matrix solution phase ?
  - How can we take advantage of the properties of circuit matrices at the software and hardware level?
  - How can we deal with the irregularity inherently present in sparse matrix calculations?
  - How can we exploit the parallelism present in SPICE matrix calculations at different granularities?
  - What are the algorithmic decisions or compromises that can be taken to enhance the potential Speedup?
- Can FPGAs outperform modern multi-core processors for solving large matrices that arise circuit simulations?
- Can off-the-shelf FPGA boards used to effectively create a high performance multi-FPGA System?
- What are the scalability issues of a multi-FPGA design?

1.3 Thesis Overview and Contributions

The intent of this thesis is the study the parallelisation of the Matrix Solution phase of a SPICE simulation on a multiple FPGA system, which has not been previously reported. This thesis also surveys relevant literature to accelerating SPICE simulation. Consequently, we propose a parallel implementation of a sparse matrix solver, which optimally exploits matrix sparsity to harness parallelism at different granularities. Our implementation is optimised for execution on a single FPGA node and can be also used as a Processing Element (PE) within a larger multi-FPGA design. Therefore, we investigate a methodology of how to partition huge matrices into almost independent blocks that can be factorised in parallel over several FPGAs. We also provide empirical data to demonstrate the merits of our design.

This thesis is structured as follows:

- Chapter 2: ***Literature Review***

  This chapter summarises the state of the art in high performance computing and surveys efforts to parallelise sequential code. We look at attempts to use FPGAs as acceleration engines.

- Chapter 3: ***Accelerating SPICE Circuit Simulations***

  This chapter gives an overview of the SPICE simulation process, explains the core algorithms involved, and sheds light on the theory that underpins a typical SPICE simulation. In this chapter, we also present our first key contribution by providing an empirical analysis for the SPICE runtime and matrices that typically arise in circuit simulations. As such, we highlight how the total SPICE execution time copes with the ever-increasing element count of modern circuits. We also study the scaling trends of the two key components of SPICE, i.e. the model evaluation and matrix solution phases, in terms of complexity, execution time, and parallelism potential. We also review the various studies and research projects that have attempted to parallelise SPICE in the last couple of decades.

- Chapter 4: ***Sparse Matrix Solution***

  In this chapter, we cover sparse LU decomposition theory from the ground up. We also show how matrices and graph theory are closely related, especially in the realm of parallelism extraction. This chapter provides a critical review of prior research relevant to the techniques employed to accelerate the LU factorisation process. It also offer an analysis of the algorithms used in our experiments. We conclude the chapter by providing details of our second key contribution, i.e., demonstrating how we employ static pivoting and symbolic analysis to create an accurate task-flow execution graph which efficiently exposes column-level parallelism.

- Chapter 5: ***Single-FPGA Matrix Solution***

  In this chapter, we present a novel parallel FPGA implementation for a sparse matrix LU decomposition hardware optimised for execution on a single FPGA. We show how our design realistically harnesses the parallelism inherently present SPICE circuit matrices. This chapter also provides the benchmark results of the prototype implementation using circuit matrices obtained from University of Florida Matrix Collection. We evaluate the performance of our solver against some of the state-of-the art sparse matrix packages, such UMFPACK, Kundert Sparse, and KLU. We evaluate and gauge the operational performance of the Sparse LU Hardware using a Xilinx Virtex-5 LX110T FPGA, but we also extrapolate our results to the more recent XC7V200T Virtex 7 FPGA. We also study the effect of matrix sparsity on the performance of our hardware design. We show that our 16-PE design configuration outperforms KLU running on a 2.67 GHz 6-core 12-thread Intel Xeon X5650 microprocessor by an average of 9.65× using a Virtex 5 FPGA.

- Chapter 6: ***Multi-FPGA Matrix Solution***

  In this chapter, we explain how we adapt our sparse LU hardware prototype from a single-FPGA architecture to a multi-FPGA system. As such, we demonstrate how we leverage the FPGAs internal Multi-Gigabit Transceivers (MGTs) to link several FPGA. We also show the design changes necessary to minimise the inter-FPGA communication and ensure that acceleration scales accordingly. We conclude the

chapter by illustrating our prototype's ability to accelerate certain circuit matrices up to 38× when compared a commodity CPU solution and up to 2.8× when compared to single-FPGA accelerator system. We also project the performance gains that can be achieved using a greater number of FPGAs.

- Chapter 7: ***Conclusions and Future Work***

  The final chapter draws some conclusions by reviewing the key points and linking them to the findings achieved. The chapter also discusses the shortcomings of our prototype and suggests various enhancements. The chapter ends with some future research directions.

1.4 List of Publications

So far the following papers have been published:

1 - Parallel Sparse Matrix Solver for Direct Circuit Simulations on FPGAs, Tarek Nechma, Mark Zwolinski, Jeff Reeve, *ISCAS*, Paris, France 2010

2 - Sparse Matrix Solver for Direct Circuit Simulations on a Multiple FPGA System (to be submitted), Tarek Nechma, Mark Zwolinski, Jeff Reeve, *International Conference on ReConFigurable Computing and FPGAs.*

Chapter 2

Literature Review

2.1 The High-Performance Computing Landscape

In the last decade, a considerable amount of research has been conducted into new ways to accelerate numerically intensive algorithms in general, and how to speed up the solution of large scientific problems more specifically [13, 14, 15, 16, 17]. In effect, solving such problems efficiently has been a great challenge to conventional computing platforms as they perform poorly on several fronts. Firstly, most scientific calculations demand floating-point arithmetic to achieve numerical stability and meet their large dynamic range data requirements [18]. However, general-purpose microprocessors exhibit modest peak floating-point performance, which limits the acceleration potential [19]. Secondly, the memory hierarchy of a conventional computer is highly unsuitable for solving such scientific problems as the irregularity in the data access pattern leads to a high rate of cache misses, and thus increases latency [20, 21].

Nevertheless, improvements in scientific applications performance have historically relied on Central Processing Unit (CPU) performance growth, which in turn relied on exploiting ever larger numbers of transistors operating at higher frequencies [22]. This trend has, however, dramatically slowed down in recent years due to physical limitations associated with miniaturisation on one hand, and high power consumption associated

with higher frequencies on the other [23]. In effect, whilst Moore's law continues, three other metrics impacting computer performance hit a peak in 2002, namely, clock speed, power consumption, and number of FLoating point Operations Per Second (FLOPS), as can be seen in Figure 2.1.

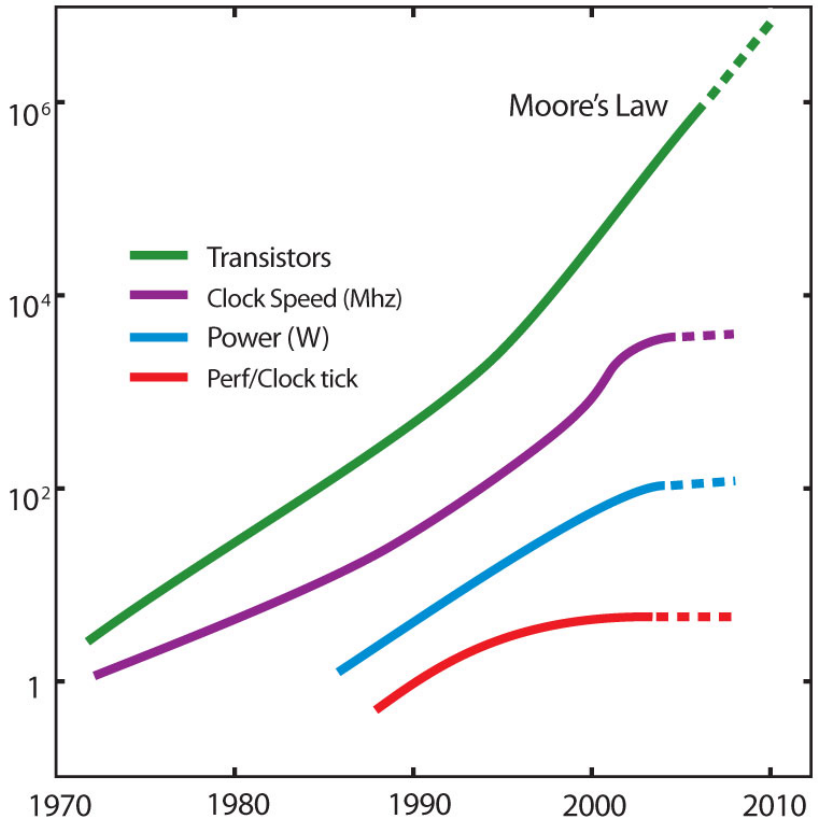


FIGURE 2.1: Moore's Law Versus Performance [23]

To overcome this so-called "speed wall" and to sustain performance improvements, the silicon industry has been moving away from single-core computer organisations to multi-core microprocessor architectures [24]. Nonetheless, the parallelisation leverage offered by multi-core machines, such as modern Graphical Processing Units (GPUs) and CPUs, remains highly dependent on the software algorithms and implementation used [25]. This is a clear indication that in order to achieve effective acceleration, parallelism has to be exposed at software level using modified or carefully chosen algorithms. Only then can the exposed parallelism be harnessed at the hardware level using some form of a special architecture [26].

Despite the significant advances in microprocessor technology, keeping up with the ever-increasing demands for computational power remains a challenge for General Purpose Processors (GPPs) [27]. This growing gap between performance of GPPs and the growing algorithmic complexity of today's applications is illustrated in Figure 2.2. High Performance Computing (HPC) refers to the use of supercomputers and computer clusters to tackle complex problems which are overwhelming for conventional GPPs. These problems are typically data-intensive and computationally demanding. HPC systems usually operate in the teraFLOPS region and exhibit high data throughputs. In the most common form, a HPC system consists of a network of commodity processors (e.g Intel, AMD) interconnected via high-speed links, as evidenced by the systems surveyed in the TOP500® list [28]. This configuration enables software engineers to write code that exploits any coarse-grain parallelism present in the problem at hand, and thus speed up the overall solution process [29].

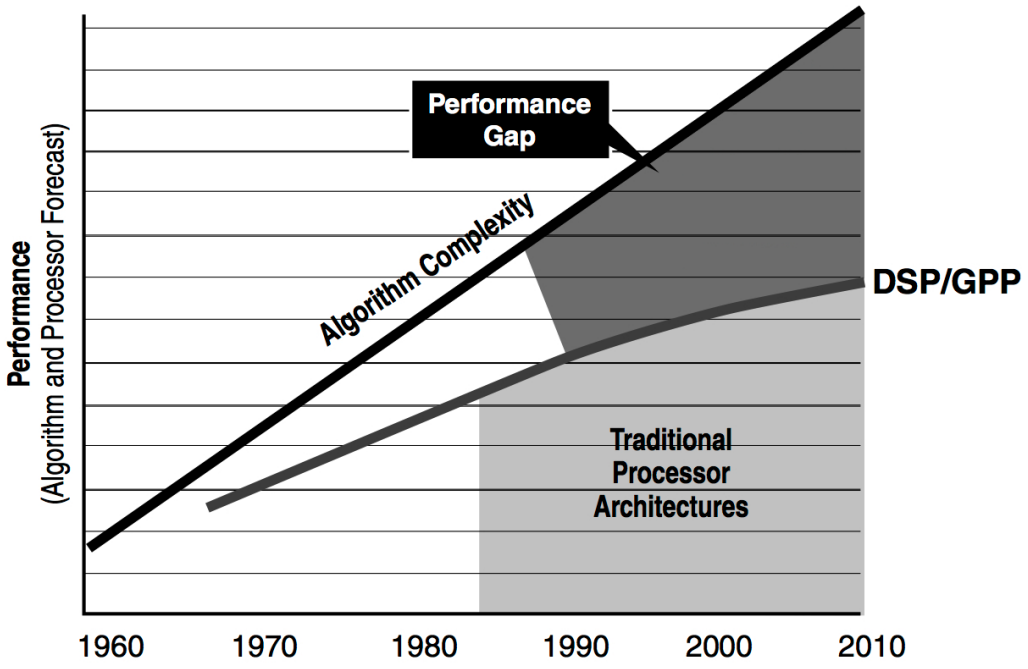


FIGURE 2.2: The Performance Gap [30]

HPCs have accomplished a great deal of success in solving computationally intensive problems [31, 32, 33]. However, their high price and the recurring high maintenance costs limited their accessibility to certain high-end applications only. According to research

conducted by International Data Corporation (IDC), for every $1.00 spent on new data centre hardware, at least an additional $0.50 is spent on power and cooling [34], as can be seen in Figure 2.3. IDC also projects that the expense of power and cooling will reach 70% of new server spending by the end of 2012, as illustrated in Figure 2.4.

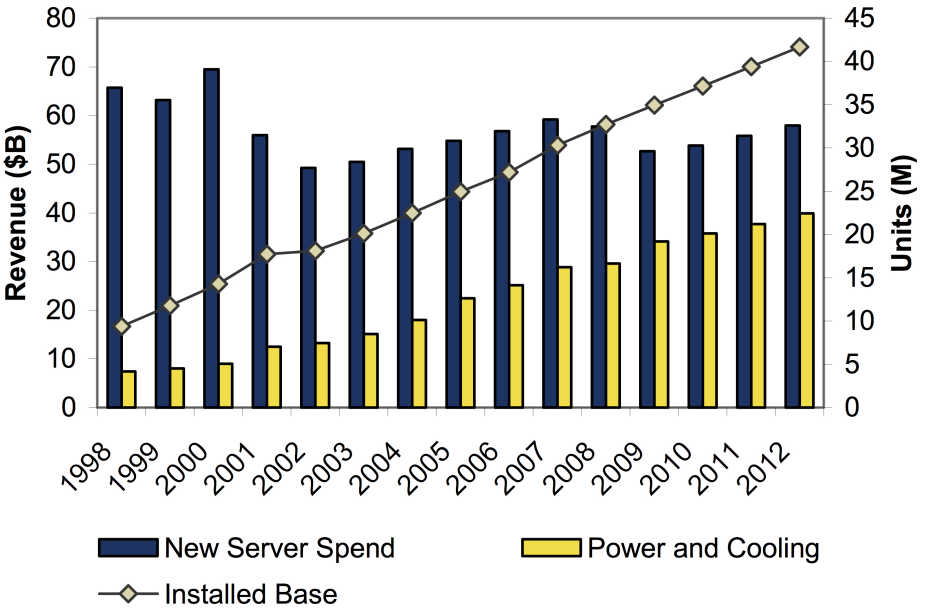


FIGURE 2.3: Worldwide Cost to Power and Cool Server Installed Base, 1998-2012 [34]

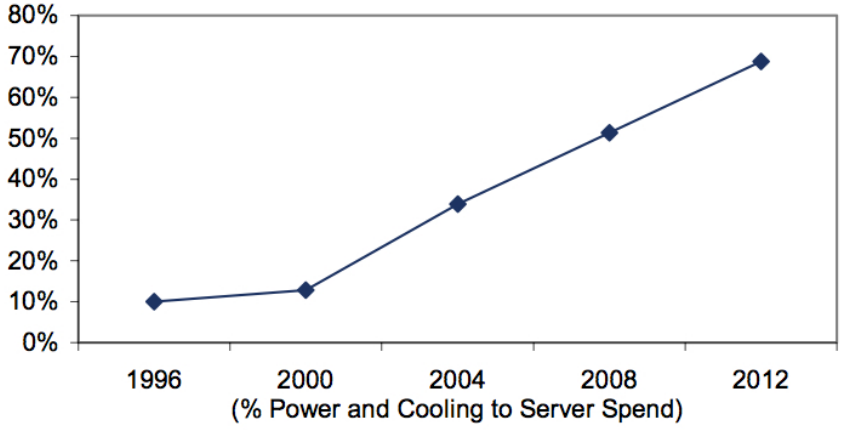


FIGURE 2.4: Worldwide Power and Cooling Server Expense as a Percentage of New Server Spend, 1996-2012 [34]

The recent advances in hardware and software technologies, including low power processors, solid state drives, and energy efficient management techniques have helped

to alleviate the energy consumption issue to a certain degree [35, 36]. However, due to the ever-increasing demand for computational power, the reduction in the energy consumption remains one of the key focus areas when designing such systems [37]. Hence, High Performance Reconfigurable Computers (HPRCs) have emerged as an alternative solution [38]. Reconfigurable computing aims at coupling the flexibility of software with the high performance of hardware through the use of Field Programmable Gate Arrays (FPGAs). Hence, computing clusters have been augmented with built-in FPGA accelerators in order to boost their computational performance while reducing the power consumption significantly [39, 40, 41, 42].

In simplified terms, an FPGA is a semiconductor device that consists of an array of programmable logic elements, configurable interconnect, and I/O (Input/Output) blocks which can be user-configured to implement complex digital circuits [43]. This highly reprogrammable structure enables FPGAs to exploit parallelism at different granularities. Moreover, FPGAs allow the execution of applications at near Application Specific Integrated Circuit (ASIC) speeds whilst circumventing the high cost of creating custom silicon [44, 45]. However, HPC applications are usually very large algorithms and cannot be fitted onto a single FPGA. In effect, it has been observed from the literature surveyed that there has been a recent trend towards using multi-FPGA systems to accommodate ever-larger applications and to offer greater multilevel parallelism leverage [46, 47, 48].

The heterogeneous nature of HPRCs offers the ability to harness parallelism at different granularities. However, parallelism has to be exposed at the software level before it can be exploited by the underlying architecture. For instance, in order to harness coarse-grain parallelism, HPC applications can be manually structured for parallel execution across a cluster of processors using special compiler directives such as multi-threading, Message Passing Interface (MPI), Open Multi-Processing (OpenMP), and so forth [49]. The finer-grained parallelism, in the case of general-purpose CPUs, can be extracted automatically by a combination of complier optimisation techniques and specialised operating system scheduling algorithms. In the case of FPGAs, fine-grained parallelism is extracted via a combination of finely-tuned behavioural descriptions and a spatial/temporal hardware synthesis process. CPUs usually have to use their own built-in functional

units to perform computations, however, FPGA designs can be finely customised and pipelined to a much higher degree, thanks to their reconfigurable architecture [50].

To sum up, reconfigurable computing architectures provide the capability for spatial parallel computations (i.e. multiple processing elements), and hence can outperform conventional computing systems in many scientific applications. While there is potential for enormous speedup using FPGA acceleration of HPC applications, achieving it requires both selecting appropriate algorithms and specific design methods that ensure parallelism is effectively harnessed.

2.2 Efficiency and Scalability of Parallel Systems

Current high performance computers boast a large number of Processing Elements (PEs) that work in a parallel fashion to accelerate computationally intensive tasks [51]. Generally speaking, the cost of a parallel system with N identical processors is less than the cost an N times faster single-core processor [25]. Hence, it is possible to use cheaper lower performance processing elements to build higher performance parallel systems. Consequently, a number of cheap Commercial Off-The-Shelf (COTS) FPGAs can be used to build a higher performance hardware accelerator. However, potential bottlenecks such as memory bandwidth and I/O bandwidth, if they do not scale with the number of PEs, can hinder if not destroy the acceleration gain of adding PEs [52]. Hence, one of the objectives of this research project is to look at how to design a hardware accelerator that spans over several COTS FPGAs whilst minimising both the inter-FPGA and intra-FPGA communication overhead.

Nonetheless, as discussed in the previous section, the parallelisation leverage, offered by FPGAs and multi-core machines, highly depends on the software algorithms and implementation used. In effect, the possible improvement gains are limited by the portion of the software that can be parallelised to run simultaneously, as illustrated in Figure 2.5.

This is known as Amdahls Law [53], which states that if P is the proportion of a software that can parallelised, and $(1 - P)$ is the proportion that is serial, i.e. cannot be parallelised, then the maximum speedup that can be achieved by using N processors is:

$$S(N) = \frac{1}{(1 - P) + \frac{P}{N}} \tag{2.1}$$

For example, if only 90% of an algorithm can be parallelised, the theoretical maximum acceleration that can be achieved is 10 times, as shown in Figure 2.5, regardless of the number processors used.

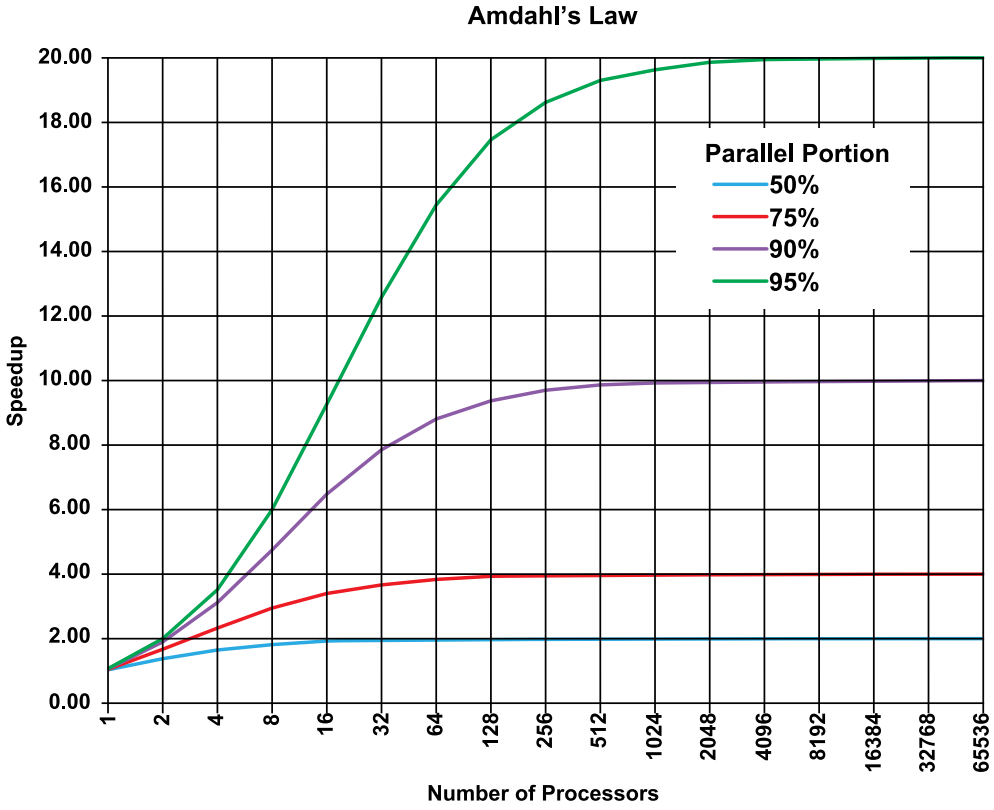


FIGURE 2.5: Amdahl's Law [53]

A closely related performance measure to Amdahl's law is "Parallelism Efficiency" [54], which can be expressed as a ratio of the time that would take an algorithm to execute on a single processor (i.e. T_1) over the n times upscaled execution time of the same algorithm on a n number of processors (i.e. T_n) :

$$E_n = \frac{T_1}{nT_n} \tag{2.2}$$

In general, acceleration and efficiency provide rough estimates of the performance changes that can be expected in a parallel processing system by increasing the parallelism degree N, e.g. by adding more processors. Therefore, in order to achieve high efficiency with a parallel implementation of an algorithm, one must carefully tune the application to ensure that there is an adequate number of PEs while minimising the parallelisation overhead of increasing the number of PEs.

2.3 The FPGA Supercomputing Paradigm

For many years, FPGA use has been limited to applications such as ASIC prototyping and verification. In the recent years, however, there has been a renewed interest to utilise FPGAs to accelerate numerically-intensive scientific problems [55, 56, 57, 58]. This intense interest is mainly due to the fact that FPGA densities have grown to such an extent that floating-point operations, which most scientific kernels rely on, can be now easily accommodated [59]. *Underwood* [60] was among the first researchers to show that the FPGAs floating-point computational ability exceeds general-purpose processor performance in single-precision and double-precision floating-point operations. In this section, we briefly review the FPGA architecture and highlight some of the key features of an FPGA that make it well-suited to accelerate SPICE simulations. We also shed light on the current technological trends of FPGAs.

2.3.1 The FPGA Architecture

A Field Programmable Gates Array (FPGA) is a semiconductor device with a massively-parallel reprogrammable architecture. Modern FPGAs consist of up to hundreds of thousands of Configurable Logic Blocks (CLBs), and interconnect wires that can be configured at the bit- and wire-level to implement arbitrary logic functions. Xilinx and

Altera are the current main FPGA vendors. Modern FPGAs also incorporate high performance DSP blocks (e.g. binary multipliers), embedded memory blocks (BRAMs), high speed programmable Input/Outut (IO) devices, and even fully functional microprocessors into the reconfigurable fabric of certain high-end models [61]. Figure 2.6 shows the typical Xilinx FPGA architecture.

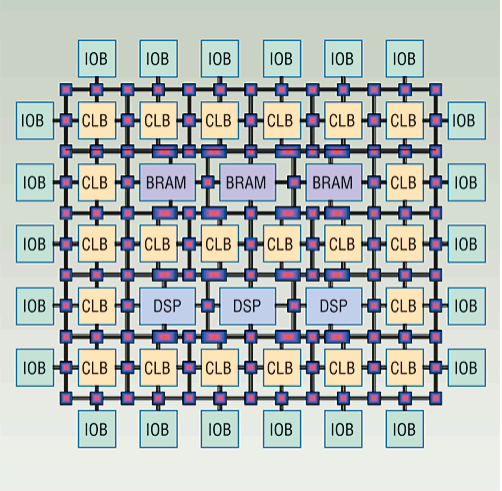


FIGURE 2.6: The General Xilinx FPGA Architecture [61]

CLB design varies between different FPGA vendors and FPGA families. They share, however, the same basic components and architecture. A typical CLB contains: one or more lookup tables (LUTs), routing fabric, and a flipflop that can be used to register data synchronously. CLBs may also contain some enhancements, such as carry propagation chains for faster distributed arithmetic [62, 63]. For instance, in the Xilinx Virtex 7 series FPGAs, CLBs are made up of two slices. Each slice consists of four six-input LUT and eight registers, as shown in Figure 2.7. Figure 2.8 shows one LUT and its associated two registers and omits the carry chain. In a full slice, there are four LUTs and eight registers.

The inherently parallel architecture of an FPGA allows computations to be performed in space rather than time by simultaneously evaluating independent operations in a fine-grained fashion. For instance, in a single-core CPU, instructions stored in an instruction memory are processed one at a time by the Arithmetic Logic Unit (ALU). Intermediate results are stored in a data memory. On an FPGA, operations can be translated into

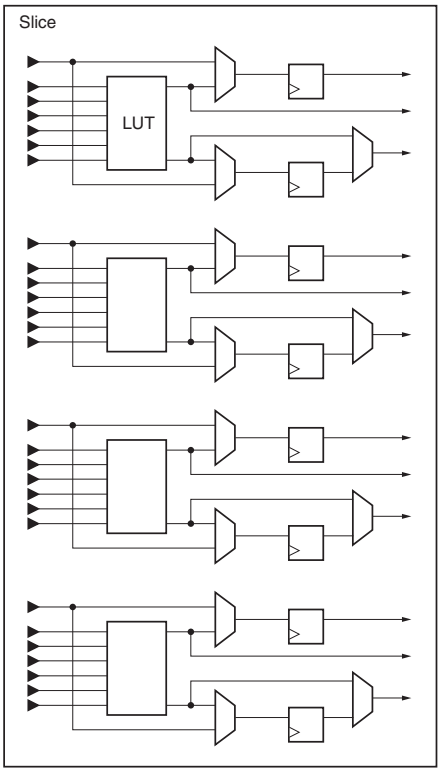


FIGURE 2.7: Slice Architecture in the Xiling Virtex 7 Series FPGAs [64]

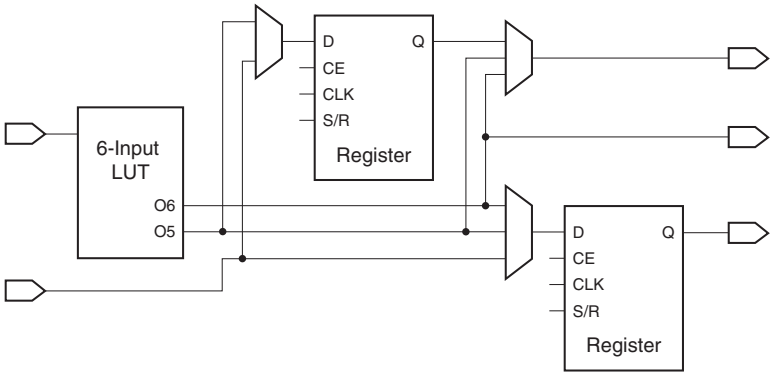


FIGURE 2.8: Layout of 6-Input LUT within a Xiling Virtex 7 Slice [64]

spatial circuits that implement the dependencies between operations physically using pipelined wires. Additionally, certain operations, such as division, may require multiple CPU cycles, whereas a custom pipelined FPGA design for those operations on can deliver a much higher throughput [65].

FPGAs are not able to achieve comparable frequencies when implementing the same

logic function on ASICs, due to the delay associated with reprogrammability [66, 67]. However, FPGAs have some clear advantages over ASICs. In effect, the implementation of smaller memories on FPGAs is relatively straightforward as they contain embedded BRAM blocks and a rich interconnect. Furthermore, pipelining on FPGAs bears no additional costs as it can be achieved by using the built-in registers. These registers can be also used to construct smaller memories, whereas in an AISC design the additional data and address lines may have a significant impact on the design routing and size.

2.3.2 The FPGA Technological Trends

In terms of transistor densities, FPGAs closely follow the trend described by Moore's Law. Figure 2.9 plots the characteristics of all Xilinx Virtex FPGA family devices since 2002. As can be seen from the graphs, FPGAs have continued to double in LUT area density every 18 to 24 months. For example, the Xilinx largest Virtex 7 FPGA now boasts more than one million LUT. To put the latter in context, one million LUTs would be sufficient to synthesise over 800 minimally-configured soft the Xilinx MicroBlaze processors in a single FPGA device [68]. Furthermore, the FPGAs' built-in resources, such as BRAMs, multipliers, and Multi-Gigabit Transceivers (MGTs), also continues to grow. In effect, the largest FPGAs today provide enough on-chip memory (tens of megabytes) to rival the capacity of todays state-of-the-art multicores caches whilst offering an unprecedented increase in external I/O bandwidth. In fact, FPGA built-in MGTs can now deliver speeds up to 28.05 Gbps per transceiver. Therefore, high-end FPGAs, such as the Virtex-7 XT FPGAs, can provide up to 2,515.2 Gbps serial bandwidth [69].

To sum up, the parallel architecture of the FPGA can be used to exploit algorithm parallelism by performing computations spatially, rather than time-multiplexing them. Meanwhile, FPGA capacities keep increasing at a much faster rater than CPU speeds, around 4 times faster as reported by *Betz et al.* [70]. As such, FPGAs promise an ever-increasing acceleration potential over conventional microprocessors. Moreover, the built-in DSP and memory blocks can be leveraged to create high-performance pipelined

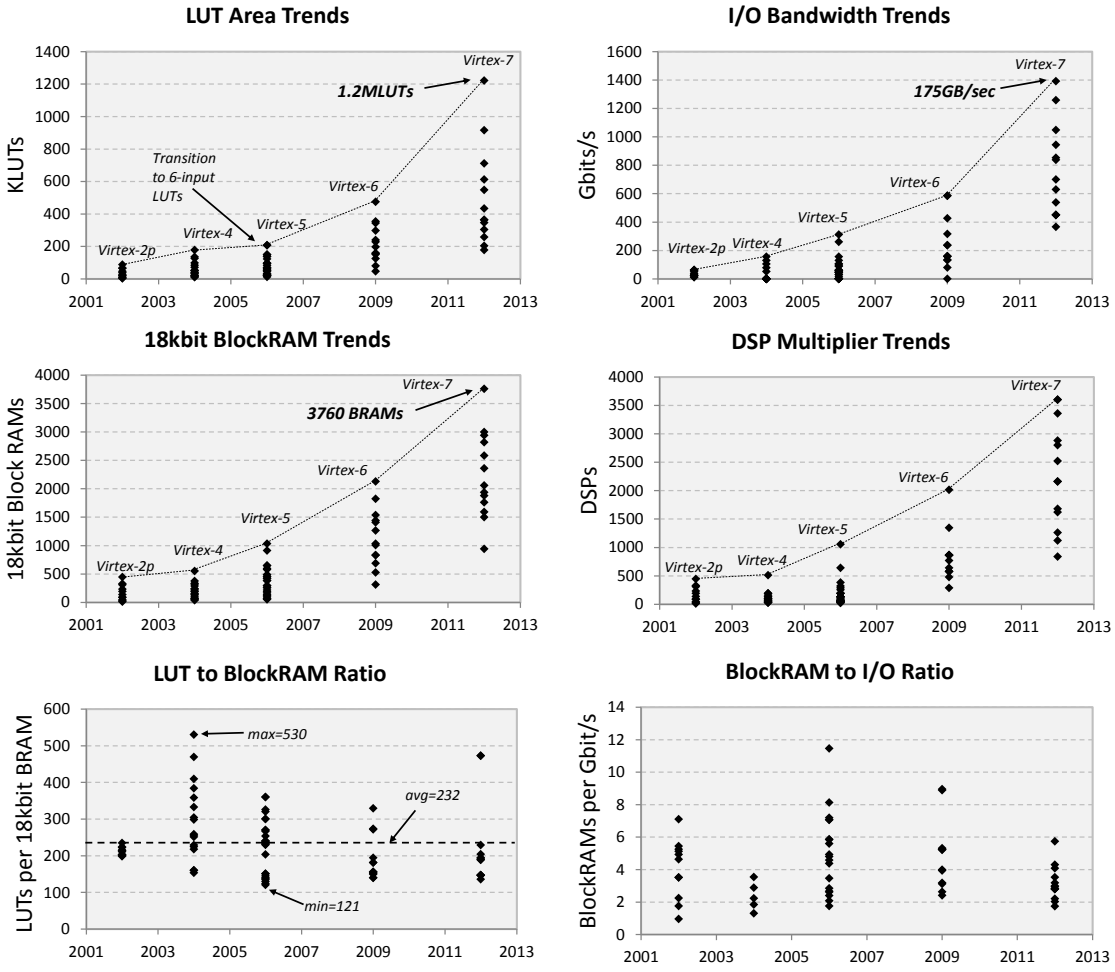


FIGURE 2.9: Xilinx FPGA Technology Trends [68]

floating-point operations, and thus accelerating the overall solutions even further. On the other hand, the high-speed transceivers can be utilised to connect several medium-range FPGAs to build a high-performance multi-FPGA hardware accelerator using the principles briefly discussed in Section 2.2. While FPGAs have been traditionally successful at accelerating inherently parallel algorithms [46, 39, 42], the migration of applications with irregular computational patterns, such as the SPICE circuit simulator [7], to FPGAs remains a great challenging for hardware designers. Hence, one of the key objectives of this thesis is to explore a methodology to migrate the computationally-intensive tasks within SPICE to a multi-FPGA design.

2.4 FPGA Acceleration of LU Decomposition

Extensive research has been conducted to accelerate sparse LU decomposition on general-purpose PCs and HPCs [71, 72, 73, 74, 75]. With the advent of the FPGA supercomputing paradigm, a considerable number of researchers investigated FPGA acceleration for LU decomposition. However, only a few FPGA implementations have been reported. In fact, FPGA implementations of direct LU factorisation only began to surface in the previous decade. Moreover, most of these implementations [76, 77, 78, 79] are generally tailored towards a specific scientific problem, where the matrix to be solved is structurally symmetric and diagonally dominant. Such matrices are relatively easy to solve and parallelise, compared to asymmetric ones. In [78], *Johnson et al.* presented a right-looking (i.e. sub-matrix based) LU sparse matrix decomposition on FPGAs for the symmetric Jacobian matrices that arise in power flow computations. Fine-grained parallelism is achieved by the use of a special cache designed to improve the utilisation of multiple floating-point units. The authors report an order of magnitude LU decomposition speedup compared to matrix package UMFPACK running on a 3.2 GHz Pentium 4. Accelerating the front and back substitutions were not considered in their work. Figure 2.10 shows a detailed diagram of the pivot search logic and the sub-matrix update logic used.

In [76, 77, 79], *Wang et al.* presented a parallel sparse LU decomposition that has been implemented using an FPGA-based shared-memory multiprocessor architecture, known as MPoPC. Each processing element (PE) consists of an Altera Nios processor attached to a single-precision floating-point unit. Coarse-grained parallelisation is achieved using node tearing to partition sparse matrices into small diagonal subproblems which can be solved in parallel. Such partitioning is known as the Doubly Bordered Block Diagonal (DBBD) form. The authors also considered only diagonally-dominant symmetric positive matrices that arise in power systems; thus, enabling them to use static data structures as pivoting is not needed and fill-in can be easily predetermined for such matrices. They report a considerable speedup for power flow analysis compared

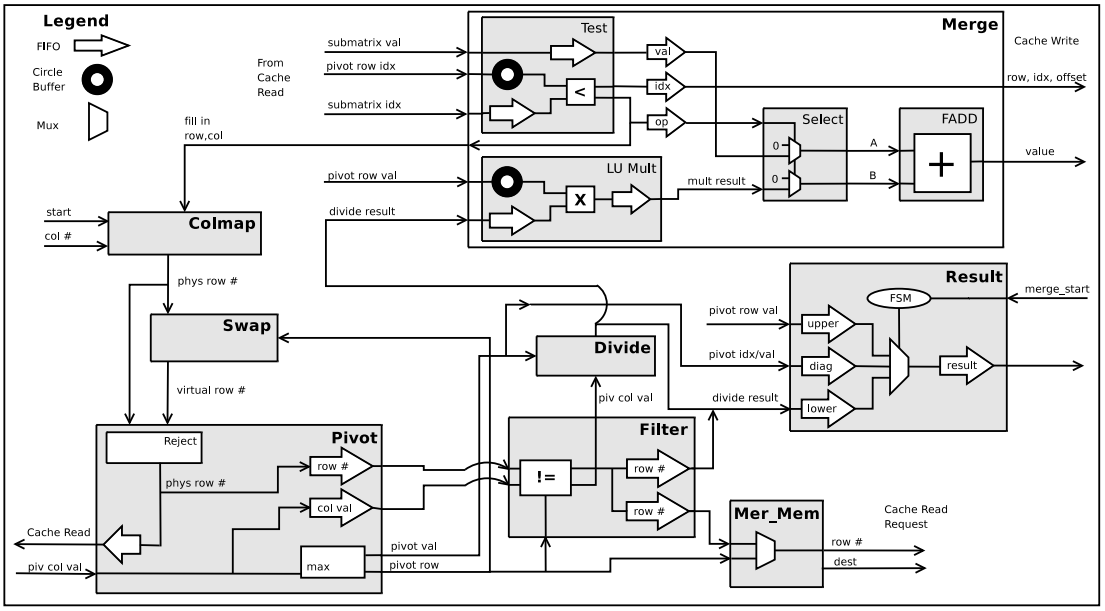


FIGURE 2.10: Pivot and Sub-matrix Update Logic, as proposed by *Johnson et al.* [78]

to a single Nios implementation. Their results, however, were not compared to existing FPGA or software implementation. Moreover, their comparison was not baselined against modern and highly-optimised LU matrix kernels such as KLU and UMFPAK.

In [80], *Kapre et al.* proposed an FPGA accelerator geared towards parallelising the sparse matrix solution phase of the spice35 open-source simulator. Using a 250 MHz Xilinx Virtex-5 FPGA, the authors reported speedups of 1.2-64 times over KLU direct solver running on an Intel Core i7 965 processor. The KLU direct solver reorganises matrices into sub-blocks, using the Block Triangular Form (BTF) techniques, and then factorise them using the Gilbert-Peierls Algorithm. KLU has been written to specifically targets SPICE circuit matrices that arise in the Newton-Raphson iteration. The acceleration, reported by *Kapre et al.*, is achieved by leveraging the standalone symbolic analysis capabilities of the KLU solver, to generate a data flow of the fine-grained floating-point operation required. The data flow graph is then mapped to a network of PEs interconnected by a packet-switched Bidirectional Mesh routing network. Figure 2.11 depicts the FPGA design presented by *Kapre et al.*,. The architecture proposed, however, focuses mainly on exploiting the fine-grained dataflow parallelism available in

KLU, potentially overlooking the coarser-grained parallelism inherently present in sparse matrices.

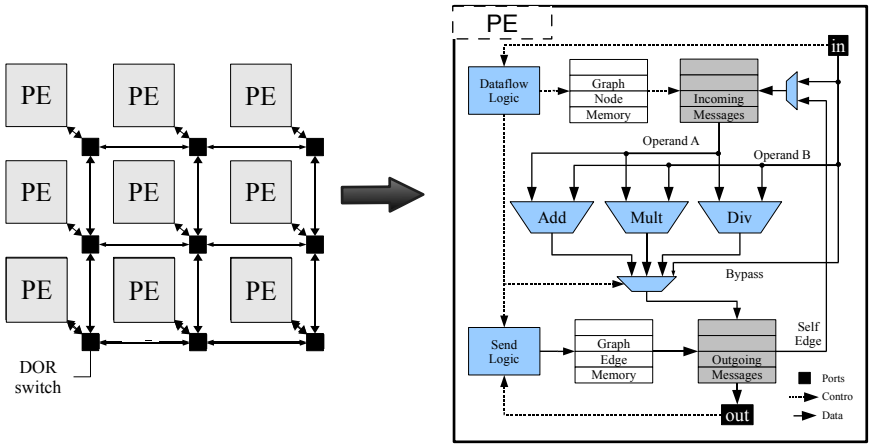


FIGURE 2.11: FPGA Dataflow Architecture for SPICE Sparse-Matrix Solve, proposed by *kapre et al.* [80].

More recently, *Wu et al.* [81] presented a 16-PE FPGA implementation of the Gilbert-Peierls Algorithm, on an Altera Stratix III EP3SL340. Fine-grained parallelism is harnessed via sharing the computation burden, to compute a given column, over a number of PE. No other levels of parallelism were explicitly considered. The basic architecture of the PE employed is shown in Figure 2.12. The reported speedups varied between 0.5-5.36X, when compared to KLU runtimes on an Intel i7 930 microprocessor. However, the benchmark matrices used are not only relatively small in terms of their size, but also have a small number of nonzeros. The latter is the main factor that dedicate the number of FLOPs needed to factorise a given matrix. Moreover results were not compared to previous FPGA implementations.

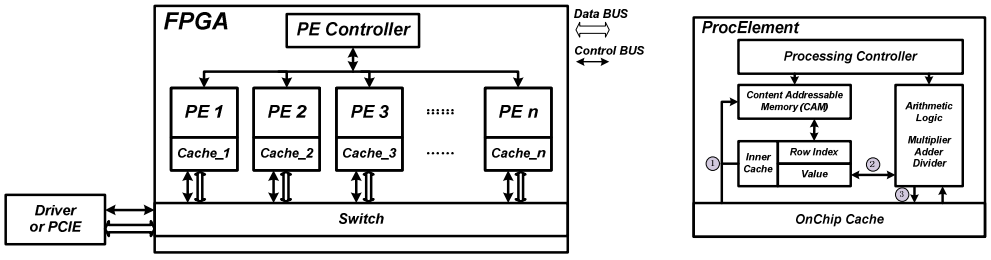


FIGURE 2.12: Basic PE architecture used for spare LU decomposition acceleration, proposed by *Wu et al.* [81]

Chapter 3

SPICE Circuit simulation

"Failures are not something to be avoided. You want to have them happen as quickly as you can so you can make progress rapidly."
Gordon Moore, Intel Co-founder

Circuit simulation is one of the most critical and time-consuming computational tasks in circuit design. State-of-the-art VLSI circuit design requires extensive and accurate simulation under nominal conditions as well as a variety of operating conditions. Moreover, modern circuit simulators have to account for a wide range of variations that could affect the manufacturing process and thus impact the quality and performance of the end product. SPICE is the industry de facto standard for circuit simulations. In this chapter, we review the fundamentals and theory that underpins a typical SPICE simulation. We review existing literature and and critique previous attempts to parallelise SPICE. Using empirical data, we also shed light on the characteristics of the matrices that arise in circuit simulations and how the SPICE runtime copes with various matrix sizes.

3.1 Overview of SPICE

SPICE is a general-purpose circuit simulation program that was initially developed by the University of California, Berkeley in 1975 [82]. SPICE simulation is an essential step in the design and verification of modern integrated circuits as it enables engineers to check the integrity of their circuit designs and to predict their behaviour. SPICE provides several types of circuit simulations for modern VLSI design, namely operating point analysis, transient analysis, and AC analysis. More types of analysis, associated with the previous three basic simulations, were added to subsequent SPICE versions. These include but are not limited to sensitivity analysis, Fourier analysis, and Noise Analysis. The latest version of the open-source SPICE simulator is spice3f5 [83].

The SPICE algorithm and its variants use a matrix representation of the circuit to find the nodal voltages over a period of time using the following key steps:

1. Formulation of circuit equations using Modified Nodal Analysis (MNA) [3].
2. Evaluating the time-varying behaviour of the design using numerical integration techniques applied to the nonlinear elements of the circuit.
3. Solving the nonlinear circuit model using Newton-Raphson (NR) based iterations.
4. Solving the resulting linear system of equations using sparse matrix techniques such as “Sparse LU Decomposition”.

Figure 3.1 shows a basic flowchart of the SPICE transient simulation algorithm. First of all, the circuit netlist, describing the interconnection of the electronic devices and their respective parameters, is parsed by SPICE and the corresponding data structures are generated. Secondly, the circuit matrix and its related data structures are set up. Then, for every time step in the transient analysis, the model calculations for each device, such as transistor, resistor, capacitor, and so on, are performed. The electrical parameters, such as conductance and current for each instance, instantiated from the corresponding device model, are computed and put into the matrix elements. Nonlinear elements are then linearised using Newton-Raphson’s method [84].

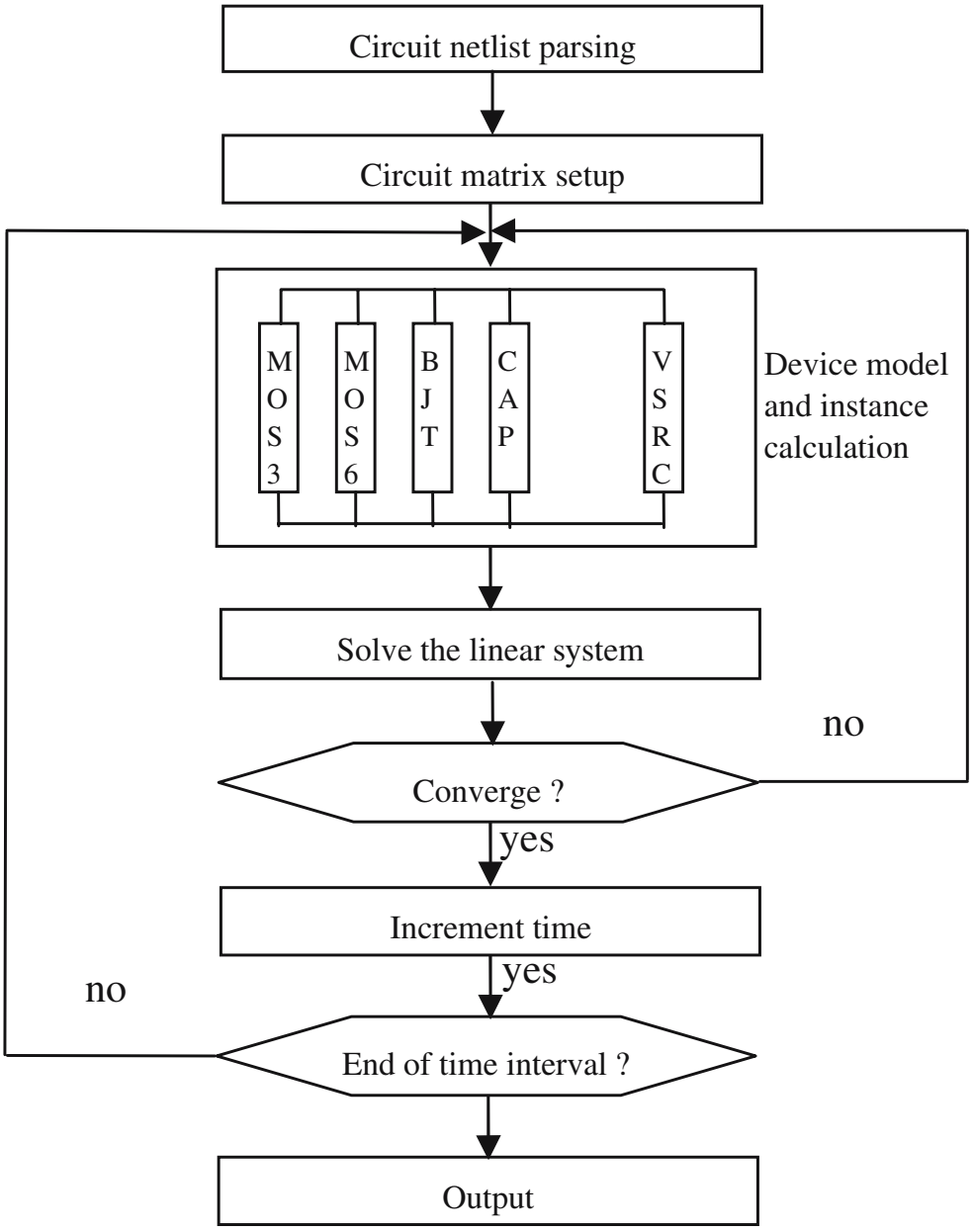


FIGURE 3.1: Basic configuration of a SPICE simulator [85]

After the device model evaluation phase, all elements in the matrix represent a linear system ready for the sparse matrix solver. The matrix calculations for the linear system, such as the LU decomposition and forward/backward elimination in each iteration, are carried out until convergence is obtained. This process continues until the final transient time is reached. Finally, the simulation results for all the time steps simulated are output.

3.1.1 Modified Nodal Analysis

As previously explained, a circuit simulator usually starts by taking a netlist, describing the circuit, as input. The netlist is then parsed and translated into a set of equations, which model the circuit behaviour. The most widely used method of formulating circuit equations is nodal analysis, which is based on the application of Kirchhoff's current law (KCL) and Kirchhoff 's voltage law (KVL) [86] . However, voltage sources, current-controlled elements, and the direct evaluation of branch currents cannot be handled easily using nodal analysis. To tackle this, *Ho et al.* [3] extended nodal analysis to Modified Nodal Analysis (MNA). The latter uses the element's Branch Constitutive Equations (BCEs) for voltage-defined elements to augment the current equations.

MNA represents an electrical circuit using a matrix containing devices' conductances and constraint equations. This matrix is built by summing the contribution of each element in the circuit. Each contribution is called a *"matrix stamp"*, which is itself a matrix containing nonzero elements only at positions occupied by the the corresponding device. MNA applied to a circuit with passive elements, independent current and voltage sources, and active elements results in a matrix equation of the form:

$$Ax = b \tag{3.1}$$

For a circuit with N nodes and M independent voltage sources: The A matrix is $(N+M) \times (N+M)$ in size, and consists only of known quantities. x is an $(N+M) \times 1$ vector that holds the unknown quantities (node voltages and the currents through the independent voltage sources), such that the top N elements are the N node voltages

and the bottom M elements represent the currents through the M independent voltage sources in the circuit. b is an $(N + M) \times 1$ vector that holds only known quantities, such that the top N elements are either zero or the sum and difference of independent current sources in the circuit, and the bottom M elements represent the M independent voltage sources in the circuit.

The A matrix can be described as the combination of 4 smaller matrices, G, B, C, and D:

$$A = \begin{bmatrix} G & B \\ C & D \end{bmatrix} \tag{3.2}$$

The smaller matrices are defined as follows:

- G is $N \times N$ is a reduced-form of the nodal matrix excluding the contributions from voltage sources, current controlling elements, and so on.
- B is $N \times M$ that contains partial derivatives of the Kirchhoff current equations with respect to the additional current variables and thus contains ± 1s for the elements whose branch relations are introduced.
- C is $M \times N$ and is determined by the connection of the voltage sources .
- D is $M \times M$ and is zero if only independent sources are considered.

The branch constitutive relations, differentiated with respect to the unknown vector, are represented by the matrices C and D. The zero-nonzero pattern of C is basically the same as that of B^T. This creates a great source of **structurally symmetry** in circuit matrices, as will be illustrated in Section 3.2.

3.1.2 The Newton−Raphson Method

SPICE uses the Newton-Raphson iterative algorithm to solve circuits with nonlinear current/voltage (I/V) relationships [87]. The method relies on the fact that nonlinear devices can be treated as linear elements over a small range. The method works by finding successively better approximations to the zeros of a real-valued function. SPICE begins by guessing the initial voltage for a given nonlinear element. The element is then linearised using this guessed value using the derivative of I/V curve. The new solution becomes the starting point of the next iteration of the Newton-Raphson algorithm and the process continues until the difference in successive solutions becomes very small i.e. convergence is reached.

Newton's method can often converge remarkably quickly, provided that it begins with a sufficiently close guess. Unfortunately, it can easily fail to converge if it starts far from the desired root. Non-convergence has always been one of the biggest hurdles in analogue simulation. This is generally a result of strong nonlinearity and discontinuity in the equations that describe the analogue parts. The Newton-Raphson algorithm can be mathematically described as follows: given a function $f(x)$ and its derivative $f'(x)$, we begin with a first guess x_0. Provided that the function is reasonably well-behaved, a better approximation x_1 can be found as follows [88]:

$$x_1 = x_0 - \frac{f(x_0)}{f'(x_0)}. \tag{3.3}$$

The process is repeated until the desired accuracy is reached:

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}. \tag{3.4}$$

To illustrate the process just outlined, we apply the Newton-Raphson method to the following example $f(x) = x^2 - 5 = 0$ (i.e. $\sqrt{5}$):

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)} = x_n - \frac{x_n^2 - 5}{2x_n} \, .$$

Taking $x_0 = 2$ gives:

$$\begin{aligned}
x_0 &= 2 \\
x_1 &= \underline{2.2}5 \\
x_2 &= \underline{2.236}1111111111111111111111111111 \\
x_3 &= \underline{2.2360679779}158040027605244996549 34 \\
x_4 &= \underline{2.2360679774997896964}47872828327110 \\
x_5 &= \underline{2.2360679774997896964091736687312 76}
\end{aligned}$$

3.1.3 Solution of the Sparse Linear System

Once the system of linear equations describing a circuit is formulated and linearised, the matrix solution phase follows. Intuitively, a system of the from $Ax = b$ can be solved by computing the inverse of A (i.e. $x = A^{-1}b$), for $n \times n$ nonsingular matrix. However, the matrix inversion process is not only a computationally demanding task, but also destroys sparsity, and hence almost never done in practice [89, 90]. There are a number of more efficient methods available for solving such systems and they can be broadly grouped into two main approaches: direct and iterative.

The iterative approach starts with a guess, which is then refined over an indeterminate sequence of solutions that may converge to a consistent result if rather strong conditions on A are satisfied [91]. This method is usually very efficient in terms of computational time and storage, however, very prone to numerical inaccuracies and convergence issues. Jacobi [92], Gauss-Seidel [93] and Conjugate gradient [94] algorithms are examples of such a technique.

Direct methods, on the other hand, are very robust and able to compute the exact solution in a predictable amount of time and storage. In effect, they are able to solve the system in a fixed and finite number of steps. One of the popular direct algorithms is LU factorisation which is used in the open source spice3f5 simulator [83]. LU decomposition

is the process whereby a matrix A is factored into two matrices: an upper triangular matrix U and a lower triangular matrix L i.e. $A = LU$, as shown in Equation 3.5. Once the elements in L and U are calculated, the unknown vector x can, in a system of the form $Ax = b$, be computed by forward substitution and backward substitution using the following two equations $Ly = b$ and $Ux = y$ respectively. LU factorisation will be covered in more detail in Chapter 4.

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{bmatrix} = \begin{bmatrix} l_{11} & 0 & 0 \\ l_{21} & l_{22} & 0 \\ l_{31} & l_{32} & l_{33} \end{bmatrix} \begin{bmatrix} u_{11} & u_{12} & u_{13} \\ 0 & u_{22} & u_{23} \\ 0 & 0 & u_{33} \end{bmatrix} \tag{3.5}$$

3.1.4 Example Circuit

In order to illustrate how the SPICE simulation process works, a simple circuit is used as an example. Figure 3.2 shows a circuit that contains a current source, one resistor, one capacitor, and a diode. The circuit equations are derived as follow:

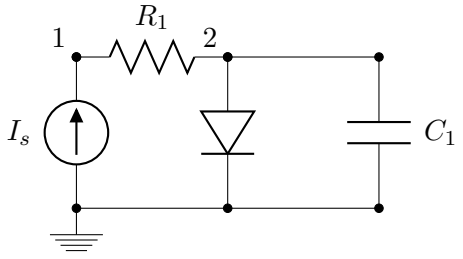


FIGURE 3.2: SPICE Circuit Example

We wish to find the voltages V_1 and V_2. 2. To do this, we write down equations that sum the currents into each node. By Kirchhoff's current law these must be zero:

At node 1:

$$I_s - I_R = 0 \tag{3.6}$$

At node 2:

$$I_R - I_D - I_C = 0 \tag{3.7}$$

Where I_S, I_R, I_D, I_C are the input current of the current source, the current of the resistor, the diode current, and the capacitor current respectively.

$$I_S = I_R \tag{3.8}$$

$$I_R = (V_1 - V_2) \cdot \frac{1}{R_1} \tag{3.9}$$

The non-linear (diode) and time-varying (capacitor) devices can be represented by their equivalent linearised models so that any circuit using them can be solved using nodal analysis as described in the previous section [95]:

$$I_D = G_D^{eq} \cdot V_2 + I_D^{eq} \tag{3.10}$$

$$I_C = G_C^{eq} \cdot V_2 + I_C^{eq} \tag{3.11}$$

Where G refers to the electrical conductance of the different circuit elements. The circuit equations can be then reorganised into the matrix form $Ax = b$ as follows:

$$\begin{bmatrix} G_R & -G_R \\ -G_R & G_R + G_D^{eq} + G_D^{eq} \end{bmatrix} \begin{bmatrix} V_1 \\ V_2 \end{bmatrix} = \begin{bmatrix} I_s \\ I_D^{eq} + I_C^{eq} \end{bmatrix} \tag{3.12}$$

In the model evaluation phase, SPICE calculates conductances and currents through different circuit elements and updates their corresponding entries in the circuit matrix. For the linear time-independent elements, such as resistors, computations are only performed once at the start of the simulation. For non-linear elements, the simulator must search for an operating-point using Newton-Raphson iterations which requires repeated evaluation of the model equations multiple times per time-step, i.e. Equation 3.13. For time-varying components, such as capacitors, the simulator must recalculate their contributions at each time-step based on voltages at several previous time-steps, i.e.

Equation 3.14. This also requires repeated re-evaluations of the device-model

$$I_D = (\frac{I_{st}}{V_j} e^{V_2/V_j}) \cdot V_2 + I_{st} \cdot (e^{V_2/V_j} - 1) \tag{3.13}$$

$$I_C = (\frac{2 \cdot C}{\delta t}) \cdot V_2 - (\frac{2 \cdot C}{\delta t} \cdot V_2^{old} + I_C^{old}) \tag{3.14}$$

Where I_{st} is the Saturation current, V_j is the Junction potential, C is the capacitance, and V_2 i is the potential at node 2 of the circuit. In the matrix solution phase, SPICE solves the resulting linear system using LU factorisation.

3.2 Characteristics of Circuit Matrices

As already discussed, the SPICE simulator employs the MNA technique to organise circuit equations into matrix A. These circuit matrices typically exhibit high sparsity as each node in the underlying circuit has only few devices connected to it. In other words, the MNA circuit matrix with $O(N^2)$ entries is generally highly sparse with $O(N)$ nonzero entries. This means that approximately 99% of matrix A entries are zeros [96]. The underlying nonzero structure of the matrix is dictated by the topology of the circuit and thus remains unchanged throughout the duration of the simulation. In each iteration, only the numerical values of the nonzero locations are updated in the Model Evaluation phase of SPICE with contributions from the non-linear element, as was illustrated in Section 3.1.4. Table 3.1 shows the characteristics of a number of circuit matrices taken at some Newton-Raphson step during a transient simulation of a circuit. The matrices are publicly available from the University of Florida Matrix collections [97]. The matrices were plot using Matlab's `spy(A)` sparse matrix plotting function, where A is the matrix to be plotted. The *"blue"* dots represent the nonzero element of the matrices [98].

TABLE 3.1: Characteristics of Circuit Matrices [97]

Matrix Name	Matrix Order	# NNZ*	Zeros (%)	Structural Symmetry**	Numerical Symmetry***
fpga_dcop_01	1813	5892	99.82%	65%	1.6%
bomhof1	2624	35823	99.47%	100%	21 %
bomhof2	4510	21199	99.89%	81%	41 %
bomhof3	12127	48137	99.96%	77%	30 %
bomhof4	80209	307604	99.99%	83%	36 %
rajat19	1157	3699	99.72%	91%	92%
rajat01	6833	43520	99.99%	99%	99%
rajat20	86916	604299	99.99%	99%	11%

* Number of nonzero elements.
** Numerical Symmetry is the fraction of nonzeros matched by equal values in symmetric locations.
*** Structural Symmetry is the fraction of nonzeros matched by nonzeros in symmetric locations.

Definition 1. Structural symmetry of a matrix A is defined as the number of matched off-diagonal nonzero elements, divided by the total number of off-diagonal nonzero elements. A matrix element a_{ij} is matched if a_{ji} is also a nonzero element. They need not be numerically equal.

Definition 2. Numerical symmetry of a matrix is defined as the fraction of nonzero elements matched by equal values in symmetric locations.

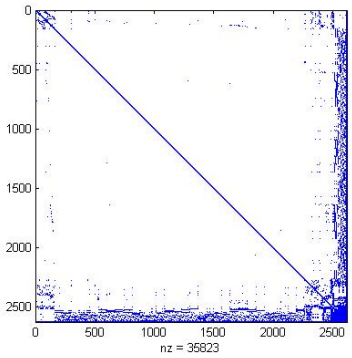
As can be seen from the matrix plots in Figure 3.3, and Table 3.1, circuit matrices are highly sparse and unstructured (i.e. do not follow a particular pattern). Nevertheless, the matrix structure is mostly symmetric with the asymmetry arises from the presence of independent sources (e.g. input voltage source) and inductors which produce an asymmetric MNA matrix stamp [99]. Circuit matrices also exhibit high sparsity as each node has only few devices connected to it, typically 2 to 4 elements. This makes them unsuitable for dense matrix kernels such as the Basic Linear Algebra Subprograms (BLAS)[100]. Sparse LU algorithms such as supernodal and multifrontal methods have been developed to group rows or columns with similar nonzero pattern in the factors into supernodes [101]. BLAS can be then applied on these supernodes. However, circuit matrices typically do not have large supernodes since the interconnection among nodes

is not similar across all the nodes in the circuit [102]. The matrices also have a zero-free diagonal unless voltage sources are present in which case a permutation such as maximum transversal [103, 104] can be used to ensure a zero-free diagonal, as will be explained in Section 4.1.4.

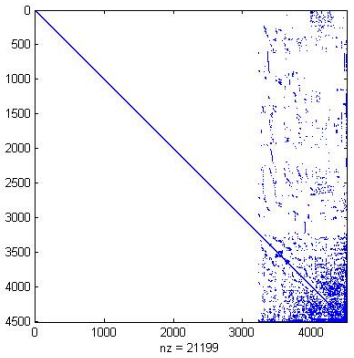
3.3 Sparsity and Optimal Reordering of Circuit Equations

Nonlinear circuit analysis in the time domain requires typically several thousand repeated solutions of the linear system at different iterations and time-steps. Moreover, Newton-Raphson's method typically needs three to four iterations to produce the solution of each system of nonlinear equations [105]. Thus, the efficient solution of the linear equations plays a critical role in the total computation time. In this section, we briefly discuss how the circuit matrix properties, studied in Section 3.2, impact the performance of the linear solver. In fact, the efficiency of the equation solution can be improved by exploiting certain properties of circuit matrices.

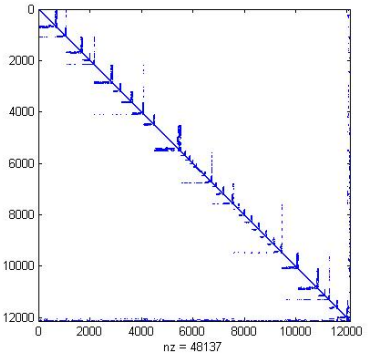
In effect, the high sparsity peculiar to circuit matrices permits the implementation of considerably faster solvers, which only operate on the nonzero entries of matrices. Therefore, the number of operations required may be dramatically reduced to be approximately proportional to the number of equations N, i.e. $O(N)$, rather than $O(N^3)$ for dense matrices. However, the sparsity may be severely reduced during the solution process as a result of a phenomenon known as "fill-in". Fill-in occurs when a previously zero entry becomes a non-zero during the solution process. This results in a change in the matrix structure as well as an increase on the amount of computation and storage required.



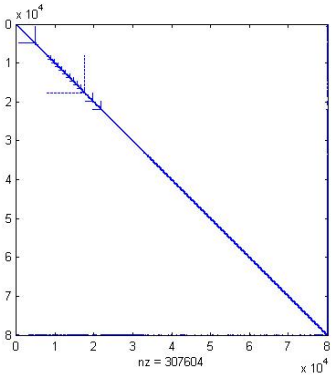
(a) bohomf1



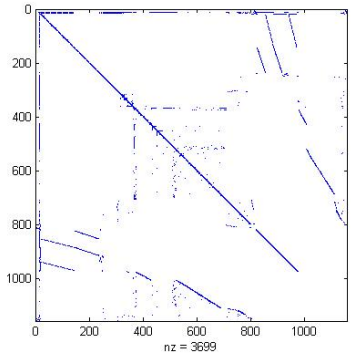
(b) bohomf2



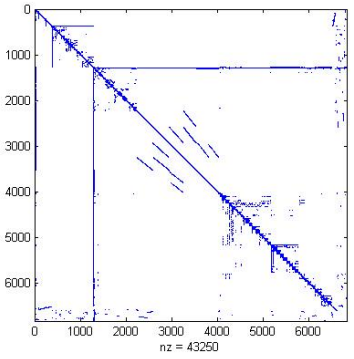
(c) bohomf3



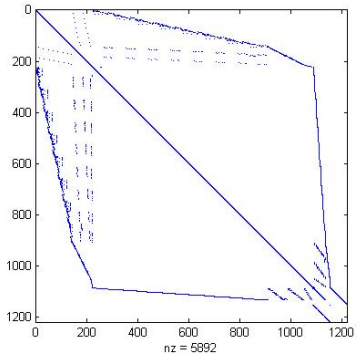
(d) bohomf4



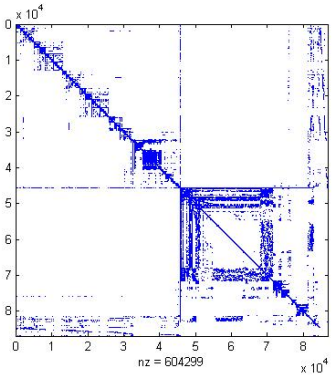
(e) rajat19



(f) rajat01



(g) fpga_dcop_01



(h) rajat20

FIGURE 3.3: Matrix Plots for Selected Circuit Matrices

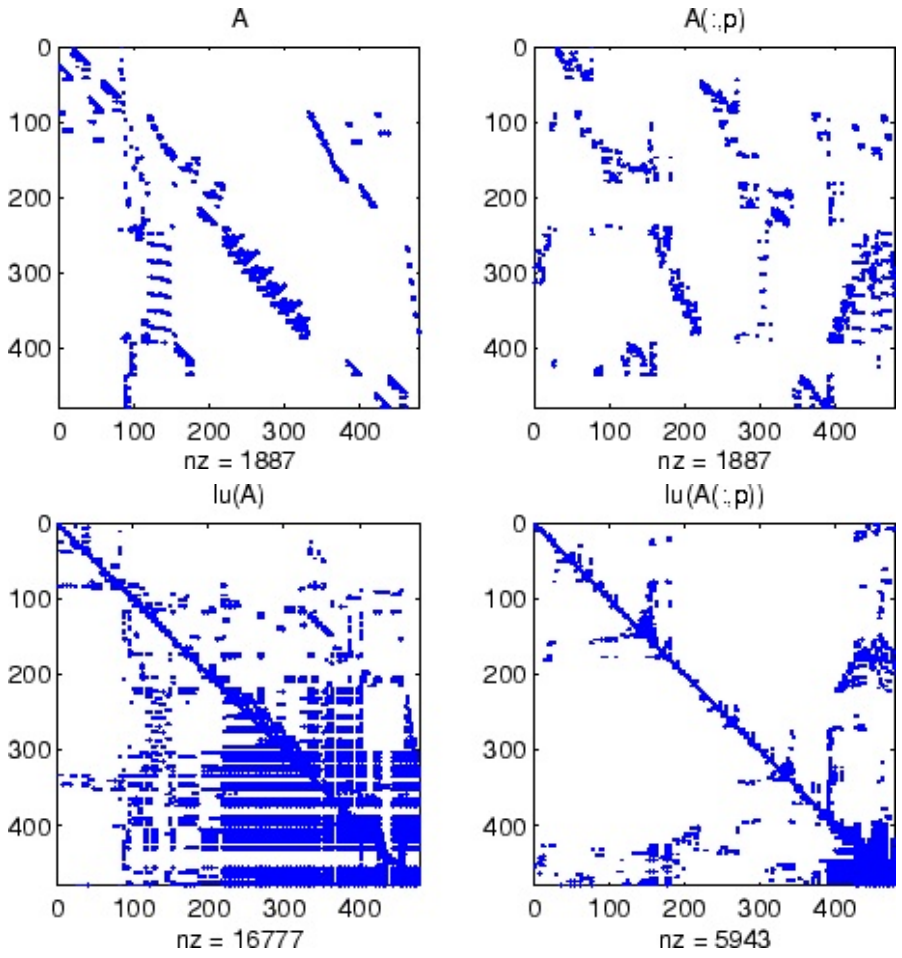


FIGURE 3.4: Effect of ordering on the sparsity of the LU factors: $A(:, p)$ permuted matrix A with column permutation p, lu() denotes Matlab's LU factorisation function

In order to limit the amount of fill-in that occurs and to preserve sparsity, the nonzero structure of the sparse matrix can be altered by reordering, i.e., permuting the rows or columns of the matrix prior to the linear solution process. Figure 3.4 illustrates the effect of reordering on the sparsity of an LU factorised matrix. We can see that fill-in caused the number of nonzeros element to increase by almost $3\times$, and hence reducing the sparsity of the resulting matrix. Fill-in will be discussed in more detail in Section 4.1.3.

However, for certain systems and algorithms, complete pivoting may be required to achieve an acceptable accuracy. Complete pivoting is more computationally demanding as it considers all entries in the whole matrix, interchanging rows and columns to achieve

the highest accuracy. In circuit simulation, the pivot is normally limited to diagonal elements due to the fact that the circuit matrices often exhibit strong diagonal dominance, which can be exploited [106]. Moreover, any round-off errors that may arise can be generally tolerated and compensated for by the Newton-Rapshon iterative method.

Finally, finding the optimal ordering, which ensures numerical stability whilst preserving sparsity, is an NP-complete problem [107]. This means that the number of operations needed to find the optimum ordering rises exponentially with the matrix size. Nonetheless, while the numerical values of the nonzero entries change during the solution process, the matrix structure, i.e., the pattern of the nonzeros remains the same as it only depends on the topological structure of the network. Therefore, there is no need for the reordering to be performed every time the linear system is re-evaluated. Instead, the reordering can be performed symbolically, based on the predicted matrix structure, not its numerical values. It is clear that accuracy cannot be taken into account if reordering is done symbolically, unless the computationally expensive dynamic reordering is used during the course of the solution [99].

3.4 SPICE Runtime Analysis

In this section, we study the performance of the SPICE simulator and analyse its scaling ability with ever-increasing circuit sizes. We use the open-source spice3f5 package [83] to simulate a wide range of circuits on a morden general-purpose PC.

3.4.1 Testing Methodology

We first explain our testing strategy using the spice3f5 simulator with a range of benchmark circuits ob a six-core 12-thread Intel Core Xeon X5650 microprocessor. We use "Rusage [resource]" spice3f5 built-in function [108] to gather usage and performance statistics per circuit and per simulation run. Some of the valid resources are:

- **all** Displays all resources.
- **time** Total Analysis Time.
- **totiter** Total iterations.
- **loadtime** Time spent loading the circuit matrix and RHS (Right Hand Side).
- **reordertime** Matrix reordering time.
- **lutime** LU decomposition time.
- **solvetime** Matrix solve time.

For instance, running "Rusage lutime" would give the LU decomposition time taken on a particular SPICE circuit, and so on. We illustrate this functionality by simulating the "passive half-wave rectifier" example circuit shown Figure 3.5. The corresponding SPICE netlist description is shown in Figure 3.6. Table 3.2 shows some of the output results of "rusage all" for the same circuit.

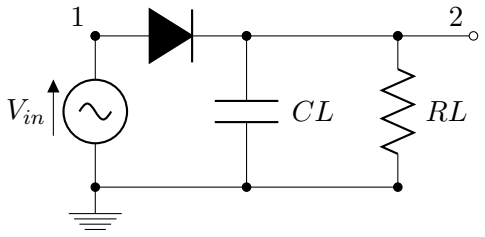


FIGURE 3.5: Passive half-wave rectifier.

```
Passive half-wave rectifier
* Lines starting with * are comments
**** SEMICONDUCTOR MODELS
.model 1N4148 D (IS=0.1PA, RS=16 CJO=2PF TT=12N BV=100 IBV=1nA)
**** CIRCUIT TOPOLOGY DEFINITION SECTION
RL 2 0 10K
CL 2 0 100n
D1 1 2 1N4148
Vin 1 0 DC 0 SIN( 0.0V 10V 2kHz )
****  COMMANDS SECTION
* Insert interactive commands into the source using:
.control
echo "Processing..."
* Run a .TRAN analysis and print the name of the active plot
tran 10us 2000ms 1ms 10us
echo " $curplot: transient analysis"
* End interactive commands with:
echo "Done."
.endc
* The last line  in the file must always be:
.END
```

FIGURE 3.6: Passive half-wave rectifier SPICE Netlist.

We gauge the performance of the spice3f5 simulator with the ISCAS85/89 benchmark circuits [109]. These benchmark circuits are a group of well-defined, gate-level netlist and functions based on common building blocks. They are widely used by the research community for IC design verification, test generation, clock distribution, power consumption and timing analysis [110]. However, the benchmark files provided just specify logic-level connections and do not provide any circuit-level information. Therefore, a considerable amount of work must be completed before we can use these circuits for our

TABLE 3.2: Sample output of the spice3f5 *rusage* statistical function.

Metric	Value
Total CPU time (s)	2.043
Nominal temperature (°)	27
Operating temperature (°)	27
Total iterations	1042327
Circuit Equations	5
Transient timepoints	411455
Total Analysis Time (s)	1.9
Transient time (s)	1.899
Matrix reordering time (s)	0.009
LU decomposition time (s)	0.17
Matrix solve time (s)	0.134
Load time (s)	0.61

testing purposes. In effect, our final aim is to perform the SPICE simulation for the ISCAS85/89 benchmark circuits. Therefore, we have to translate the gate-level netlists to the final SPICE netlists. The latter must be extracted once the real circuit layout is completed, so that the real impact of interconnect length, coupling issues as well as the parasitic parameters can be extracted and incorporated into the final SPICE simulation. We use *iscas2spice* software suite [111] to translate ISCAS85/89 benchmark circuits into SPICE netlists. The *iscas2spice* package also contains a 130nm standard cell library consisting of NAND, NOR, AND, OR gates with up to four inputs and some other usual gates such as INV and XOR. The main steps performed using *iscas2spice* software suite are as follows:

1. Match the components of the ISCAS85 benchmark circuits with the standard cells.
2. Translate the ISCAS85 ".bench" files into the input files for the existing placer and do the placement.
3. Translate the output file of the placer to the format of the input file of the router and performing routing using the global router

4. Extract the routing information from the router output file and translate the original benchmark circuits into the SPICE netlist using the standard cell library models.

5. Run a transient simulation for the extracted SPICE netlist and collect the "rusage" information.

The overall procedure is illustrated in Figure 3.7, however, detailed steps can be found in [111]. The tests results from the spice3f5 built-in "rusage" function are summarised in Table 3.3.

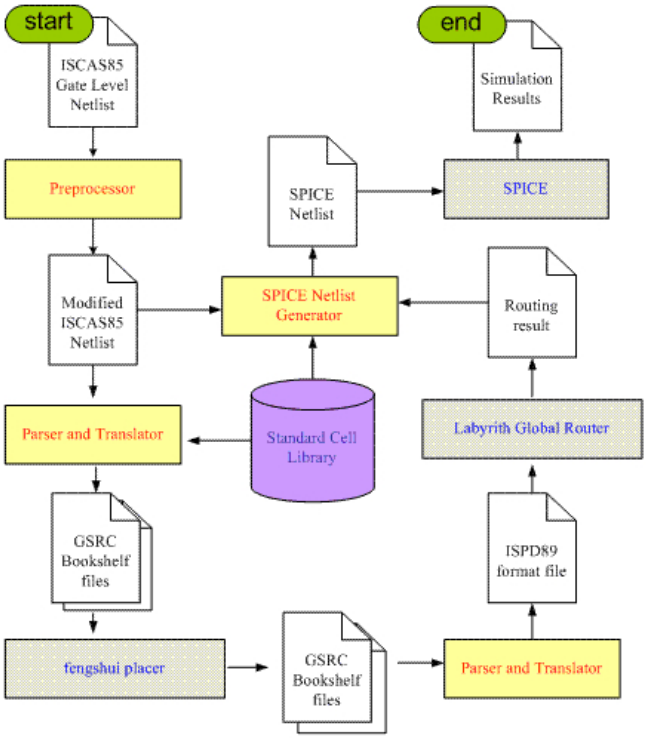


FIGURE 3.7: Performing the SPICE Simulation of ISCAS85/89 Benchmark Circuits using *iscas2spice* software suite [111]

TABLE 3.3: Circuit Simulation Benchmark Matrices

Circuit Name	Matrix Size	Zeros (%)	Circuit Load Time (ms)	Total Analysis Time* (ms)	LU Reordering Time* (ms)	LU Decomp. Time* (ms)	LU Solution Time* (ms)	LU Total Time* (ms)	Mod. Eval Time* (ms)
s27	189	99.56	13	0.05	0.00	0.03	0.00	0.03	0.01
s208	1296	99.47	49	0.82	0.04	0.32	0.08	0.44	0.38
s298	1801	99.60	120	1.93	0.06	0.36	0.18	0.60	1.33
s344	1992	99.65	138	2.04	0.09	0.40	0.14	0.63	1.41
s349	2017	99.65	176	2.45	0.13	0.36	0.19	0.68	1.77
s382	2219	99.68	167	2.28	0.13	0.40	0.11	0.64	1.64
s444	2409	99.70	132	1.57	0.10	0.78	0.11	0.99	0.58
s386	2487	99.71	198	2.35	0.16	0.29	0.13	0.58	1.77
s510	2621	99.69	145	1.52	0.11	0.68	0.18	0.98	0.54
s526n	3154	99.76	171	1.74	0.16	1.01	0.22	1.38	0.36
s526	3159	99.76	185	1.84	0.16	0.57	0.42	1.15	0.69
s641	3740	99.80	259	2.48	0.25	1.66	0.28	2.19	0.29
s713	4040	99.81	314	2.77	0.27	1.25	0.73	2.26	0.52
s820	4625	99.82	407	3.58	0.38	1.79	0.71	2.87	0.71
s832	4715	99.83	548	4.81	0.50	2.10	1.19	3.80	1.01
s953	4872	99.84	731	6.06	0.75	1.14	0.68	2.57	3.49
s1196	6604	99.81	941	7.31	0.98	3.11	1.51	5.59	1.71
s1238	6899	99.86	999	7.60	1.09	2.74	1.57	5.41	2.19
s1423	9304	99.92	1105	8.11	1.38	3.23	1.20	5.81	2.30
s1488	9849	99.92	1834	13.23	2.21	5.16	1.90	9.26	3.97
s1494	9919	99.92	2285	15.19	2.73	5.07	2.61	10.40	4.79

* Per Iteration

3.4.2 Total Runtime Analysis

Figure 3.8 illustrates how the circuit size impacts the SPICE simulation runtime. We can see that the runtime scales as $O(N^{1.3})$ as the circuit size increases, as shown by the trend line in the graph. This means that the SPICE sequential runtime will get increasingly slower as we pack more devices into same silicon die area, in accordance with Moore's Law. To shed more light on the SPICE runtime, we examine, in Figure 3.9, the SPICE runtime breakdown per its two main phases, namely, model evaluation and matrix solution phases. Generally speaking, we can see that the model evaluation phase dominates the runtime for smaller circuits whereas the matrix solution phase dominates for bigger circuits [112].

However, the runtime may fluctuate depending on the makeup of the underlying circuit. In effect, the model evaluation phase tends to dominate the SPICE runtime for circuits that are mostly composed of non-linear transistor elements. On the other hand, the matrix solution execution time dictates the runtime for circuits with large parasitic components (e.g. capacitors, resistors) where the non-linear devices are a

small portion of total circuit size. Similar conclusions have been drawn by *Kapre et al.* [113]. Figure 3.10 and Figure 3.11 show the effects of parasitics on the overall SPICE simulation runtime as well as the time taken by SPICE's two main phases respectively. We can see that the inclusion of parasitics not only affecst the SPICE runtime adversely but also cause the runtime distribution to swing in favour the Matrix Solution Phase.

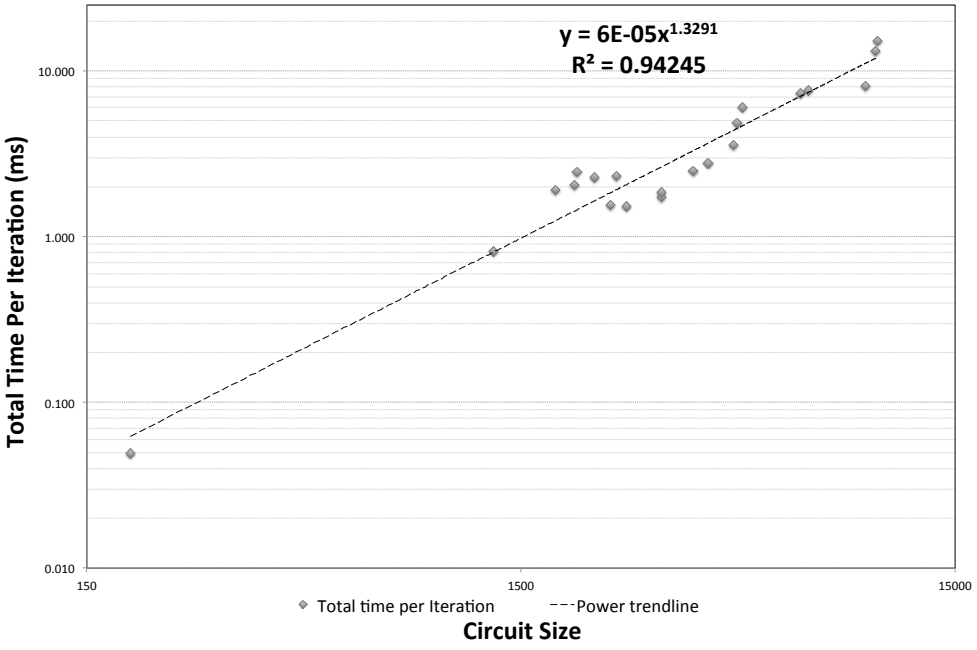


FIGURE 3.8: SPICE total runtime scaling trends with ISCAS85/89 benchmark circuits

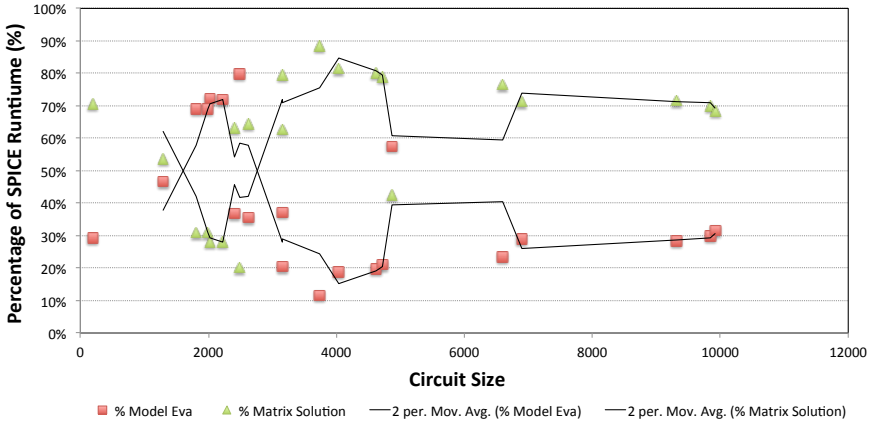


FIGURE 3.9: SPICE Runtime Breakdown

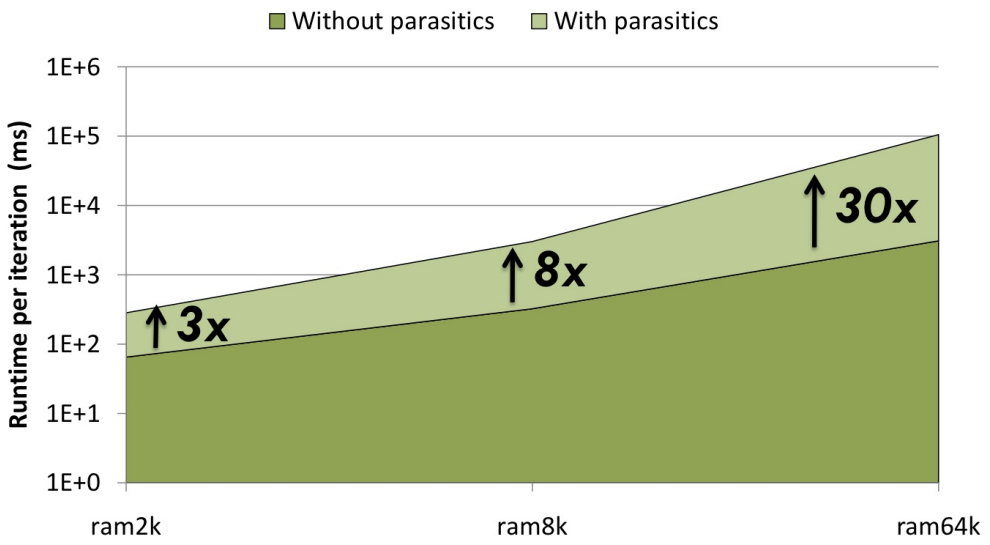


FIGURE 3.10: Effect of Parasitics on SPICE Runtime [113]

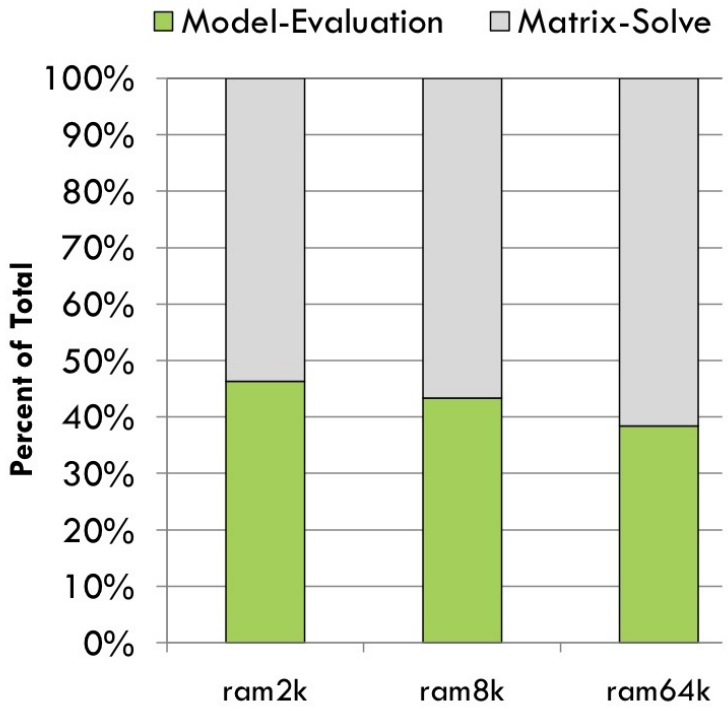


FIGURE 3.11: Effect of Circuit Size on SPICE Runtime Distribution [113]

3.4.3 Runtime Scaling Trends

As explained earlier, a SPICE simulation is an iterative process that consists of two phases per iteration, namely, model evaluation phase followed by a matrix solution phase. In this section, we examine how the execution time of these two phases scales with the ever-increasing circuit sizes. In Figure 3.12, we graph the runtime per SPICE simulation phase as a function of the circuit size. From the trend lines in graph, we can see the model evaluation phase scales as $O(N^{1.1})$ as the circuit size increases, compared to $O(N^{1.4})$ for the matrix solution phase. From that, we can conclude that for extremely large circuits, the matrix solution will most certainly dominate the overall SPICE simulation runtime. These results are in line with similar findings in previous works [112, 114, 115].

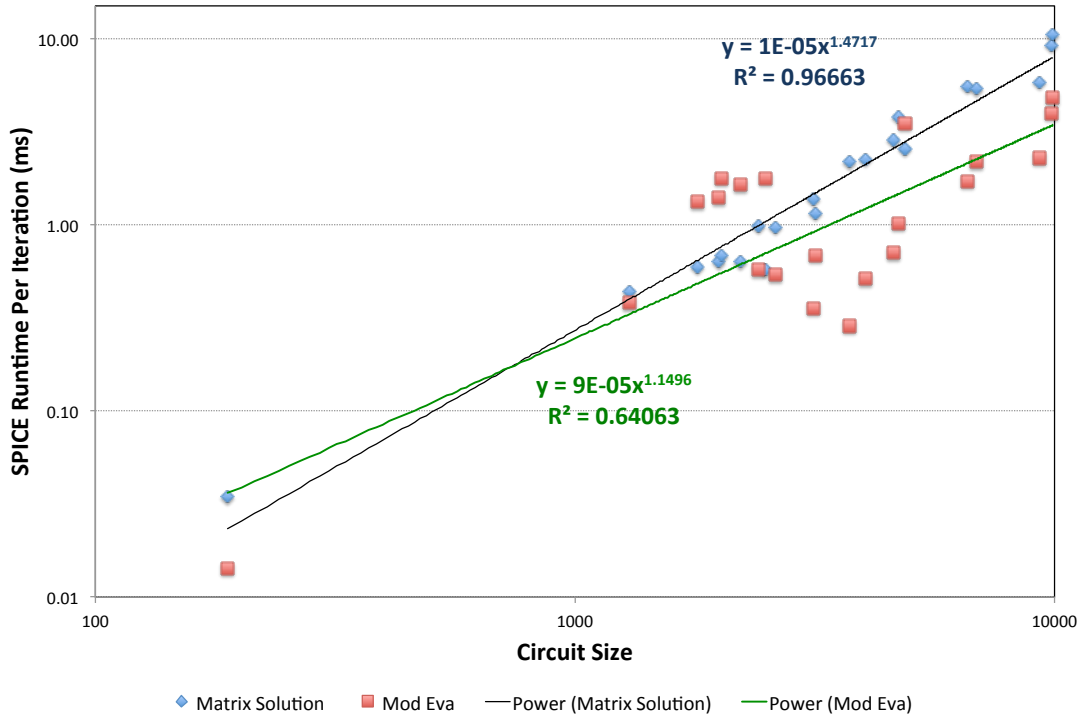


FIGURE 3.12: SPICE Runtime Scaling Trends Per Phase

In the matrix solution phase, there are 3 main steps: matrix reeordering, LU decomposition, and LU matrix solution. In fact, matrix reordering refers to the fact that

SPICE performs dynamic pivoting during the LU decomposition process in order to maintain numerical stability. Looking more closely at the runtime of each these 3 steps, we can see that matrix reordering time scales at a rate of $O(N^{1.74})$ compared to $O(N^{1.35})$ for the LU decomposition time, as depicted in Figure 3.13. This means that in order to speed the matrix solution phase, one cannot ignore the effect of dynamic reordering on matrix solver runtime. In the next chapter, we will explore ways to eliminate the need of dynamic pivoting and hence improve the runtime without compromising accuracy.

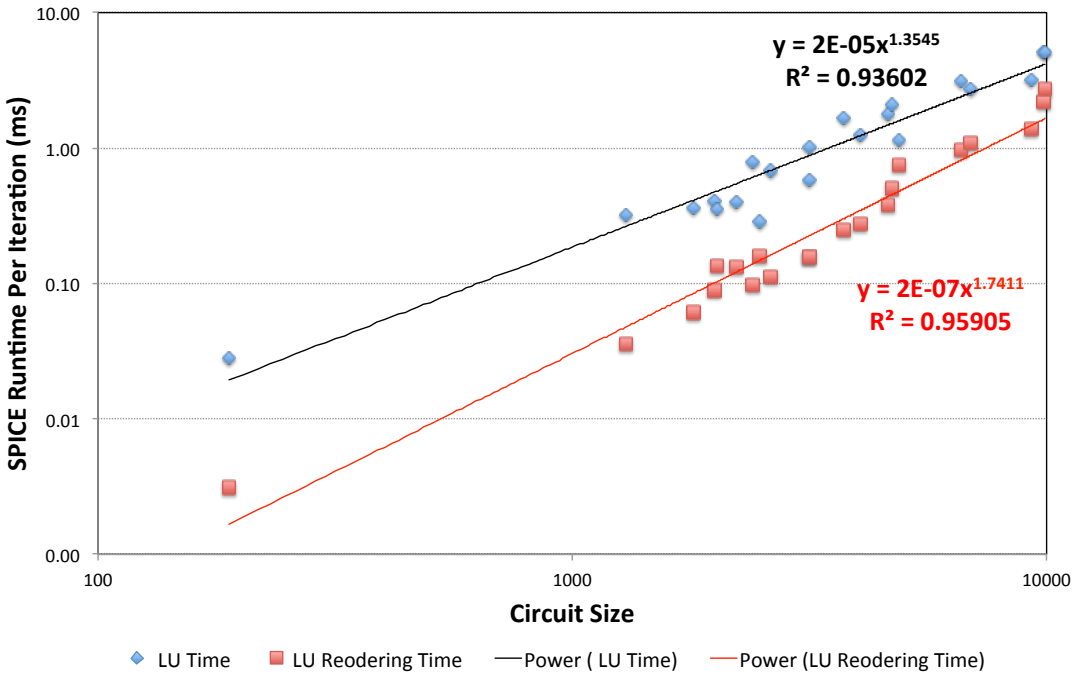


FIGURE 3.13: SPICE Matrix Reodering Scaling Trends

Moreover, following Moore's Law device miniaturisation trend, the underlying device models are becoming larger and more complex in order to account for physical effects that may arise [116]. Figure 3.14 highlights the increasing complexity of Metal-Oxide-Semiconductor Field-Effect Transistor (MOSFET) model. Scaling down device sizes also increases the impact of tighter coupling and interference between the circuit elements. This requires more rigorous modelling of parasitic elements (e.g. capacitors, resistors) and thus increasing the size of the SPICE circuit matrix, which in turn increases the time

spent in the matrix solution phase. The increase in the SPICE runtime per iteration due to inclusion of parasitic effects is shown in Figure 3.10.

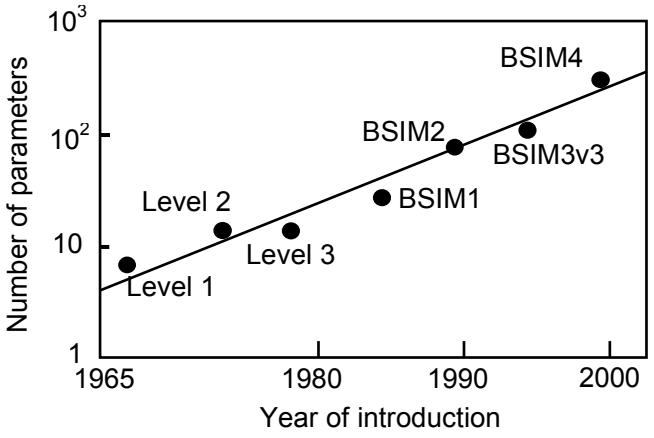


FIGURE 3.14: The increase of MOSFET model parameters [117]

Moreover, the runtime of the matrix solver does not scale well with the number of processing elements used as was demonstrated in [118, 119]. This is, in effect, another clear indication that in order to successfully speedup the SPICE runtime, the matrix solution phases has be effectively parallelised in a scalable fashion in accordance with Amdhal's Law (as it was explained in Section 2.2). The work presented in this thesis is based on parallelising the matrix solution phase. As such, we aim to investigate how to build a scalable low-latency multi-FPGA accelerator with a processing architecture capable of efficiently harnessing parallelism available within the matrix solution phase of the SPICE simulator.

3.4.4 Parallel Potential Analysis

We have so far established that the SPICE simulation has two computationally-intensive phases that can be parallelised. The first phase is the device model evaluation, in which non-linear device models are evaluated (e.g. diodes, transistors). The other phase is the matrix solution phase, in which a linear system of the form $Ax = b$ is solved for the unknown vector x. In the model evaluation phase, the non-linear device computations are inherently independent from each other, and hence each device can be

evaluated concurrently, in a data-independent fashion, on different processes. Therefore, the amount of parallelism is proportional to the number of non-linear devices in the circuit. This not only makes this phase vastly parallelisable but also highly scalable [113, 120].

In the matrix solution phase, fine-grained parallelism can be extracted at the scalar-level by concurrently performing independent floating-point computations within a particular matrix operation, such as column normalisation, column multiplication, column updates, and so on. However, spreading sparse matrix computations over a number of processing elements introduces a number of constraints. In effect, the SPICE matrix solver employs the Markowitz algorithm [121] to carry out dynamic pivoting during the matrix factorisation process. Pivoting is more complex in parallel implementations because the permutation of rows or columns requires global synchronisation between all processing elements (PEs). This has two key implications. Firstly, the reduction of the amount of pivoting required during the factorisation process enable a more effective matrix partitioning and hence will increase the parallelism potential. Similarly, it may be also more desirable to use a static data distribution scheme which would eliminate the need of performing dynamic pivoting, as it will be shown in the next Chapter. Secondly, performing sparse calculations in a distributed manner requires an adequate inter-PE communication mechanism that scales well with the number of PEs in terms of bandwidth. Otherwise, any acceleration gains will be destroyed by the communication overhead.

Fill-in is another phenomenon that could undermine efficiency of sparse matrix decomposition as it it could lead to more operations and memory requirements. The stability and sparsity requirements for pivot selection are often contradictory and most strategies involve some sort of a compromise and the generalised Markowitz strategy is an example of that. Selecting pivots for parallelism add a third constraint. Therefore, one of the key contribution of this thesis is to identify to a reordering or a preconditioning strategy that offers the best compromise in terms of maintaining numerical stability and preserving sparsity whilst increasing the parallelism potential. This will be explored in more detail in the next chapter. We follow *Liu's* [120] template in identifying three

potential levels of granularity that we aim to exploit in using a parallel implementation of matrix factorisation process:

- Fine-grain parallelism: concurrently evaluating independent scalar operations (e.g. multiplication, addition, division, etc).
- Medium-grain parallelism: concurrently evaluating independent columns.
- Large-grain parallelism: concurrently evaluating of groups of columns or sub-matrices.

3.5 Parallel Circuit Simulation

Transistor-level circuit simulation is a fundamental computer-aided design technique that enables the design and verification of an extremely broad range of integrated circuits. In effect, circuit simulation enables the prediction of circuit performance and thus makes it possible to disqualify a failing design before the start of the expensive chip fabrication process. Therefore, it is not surprising that parallel circuit simulation is not a new concept. In fact, as early as 1982, researchers have attempted to develop parallel simulation capabilities on a variety of computer architectures such as vector machines [112, 122], multi-processors [123, 124, 125, 126], and supercomputers [114, 127]. With the proliferation of multi- and many- core processor technology [128, 129, 130, 131], general purpose PCs now offer an amount of computing power that rivals the processing muscle of expensive supercomputers from a couple of decades ago.

This architectural shift sparked a renewed interest to parallelise CAD simulations on commodity PCs. More importantly, it has reinvigorate active development of modern commercial parallel circuit simulators from all major EDA tool vendors and stimulated research in parallel circuit simulation [132]. In effect, several parallel simulators of electronic circuits have been developed recently, such as Xyce [133], TITAN [134], and SEAMS [135]. FineSim Spice [136] is a commercial circuit simulator for mixed-signal SoCs that can run over distributed networks or multi-CPU workstations. Commercial

SPICE simulators such as HSPICE [137] and Virtuoso Accelerated Parallel Simulator [138] use multithreading simulation capabilities to exploit multicore processors to simulate of large post-layout designs.

Moreover, the emergence of modern commodity heterogeneous platforms, comprising homogenous multicore microprocessors with attached accelerators such as GPUs and FPGAs, has brought new opportunities for accelerating circuit simulations using domain-specific partitioning. In effect, impressive speedups may be achieved if the task at hand is optimally partitioned, and the resulting subtasks are efficiently mapped to either the CPU or the hardware accelerator depending on subtasks characteristics [139]. Additionally, programming model for these heterogeneous systems are also becoming more user-friendly [140, 141, 142, 143, 144, 145, 146]. As such, a diverse array of parallel hardware platforms exists today, ranging from heterogeneous processors and hardware accelerators (GPUs and FPGAs) to computer clusters and supercomputers, which the research community can leverage towards the ongoing efforts to accelerate large-scale circuit simulation.

A variety of parallel simulation approaches for SPICE exist. Algorithmic-based approaches aim to harness parallelism available within the underlying algorithms of the SPICE simulator. As such, parallelism can be explored at the levels of device evaluation [4, 11, 113], matrix solution [147, 148, 149, 150], or the nonlinear equations [151, 152]. Parallelism can be also explored via concurrently evaluating individual subcircuits [150, 152]. One of the first published algorithms for circuit-level partitioning is *"Node Tearing"* [153]. This algorithm starts from an input voltage source and gathers adjacent elements until it reaches a specified partition size. If there are several possibilities for the selection of an adjacent element, the algorithm takes the node with fewer connections. In the signal domain, parallelism is explored along the time or frequency axis. For instance, computations used to find the circuit responses at different time points may be processed in parallel [114, 154].

3.6 Summary

In this chapter, we have established that model evaluation phase of a SPICE simulator is rather straightforward to parallelise as the model evaluations are independent of each other. The parallelisation of the matrix solution phase, however, is more complicated because of the dependency relationships that exist within the matrix solution process. In circuit simulation, the use of pivoting, or matrix reordering, during the computation is usually avoided. In fact, in most circuit simulation programs, pivoting for accuracy is not performed during the transient analysis unless a zero (or a value close to zero) is encountered on the diagonal. This is acceptable in practical terms as the linear equation solution is used as part of Newton-Raphson's method and an occasional small error during the iterative process does not affect the integrity of the final solution, although it may have some influence on convergence.

In addition, circuit matrices are often diagonally dominant. Thus, matrix reordering is usually only performed to preserve sparsity and to enhance parallelism. This tends to increase parallelisation potential of the SPICE matrix solution, when compared to the most general case of the parallel sparse linear problem [114]. In direct circuit simulation, the linear equation solution is usually performed using LU factorisation followed by forward elimination and backward substitution. There are a variety of different methods for LU decomposition which will be covered in Chapter 4. The forms of parallelism available in LU decomposition can be categorised as follows:

- **Fine-grain** parallelism associated with element-level update operations
- **Medium-grain** parallelism associated with independent colums/pivots
- **Coarse-grain** parallelism associated with independent sub-blocks.

The extent to which these three forms of parallelism can be exploited depends on the structure and sparsity of the circuit matrix and the particular method of LU factorisation used. Matrix reordering schemes that balance increasing parallelism against minimising fill-in (that is, maintaining sparsity) are clearly important in the development of an

efficient parallel circuit matrix solver and therefore will be covered in more detail in the next chapter.

Chapter 4

Sparse Matrix Solution

In the previous chapter, we have empirically shown that the speed of the linear solution becomes crucial in large-scale simulations, as the computational complexity of the linear solution grows faster than the size of the circuit. We have also established that the linear solver becomes a main problem in parallelisation of circuit simulators, due to the fact Matrix Solution phase has inherently much lower parallelism than the other parts of a circuit simulator (e.g. data-parallelism in model device evaluations). In circuit simulations, direct methods, namely the sparse Lower/Upper triangular (LU) decompositions, are preferred over iterative methods which suffer from convergence issues. Thus, this chapter provides the conceptual grounding and theory that underpin sparse LU decomposition. It also defines the key terminology and the algorithms used in subsequent chapters. Finally, we demonstrate how we leverage the graph representation of a matrix to create a dependency-driven task model schedule that maximises the parallelism potential for the matrix solution phase.

4.1 Theory: Sparse LU Decomposition

4.1.1 Dense LU Decomposition

LU decomposition is the process whereby a matrix, A, is factored into two matrices: an upper triangular matrix U and a lower triangular matrix, L, i.e. $A = LU$. Once the elements in L and U are calculated, the unknown vector x, in a system of the form $Ax = b$, can be computed by forward substitution and backward substitution using the following two equations $Ly = b$ and $Ux = y$ respectively. For example, for a 3-by-3 matrix A, its LU decomposition looks like this:

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{bmatrix} = \begin{bmatrix} l_{11} & 0 & 0 \\ l_{21} & l_{22} & 0 \\ l_{31} & l_{32} & l_{33} \end{bmatrix} \begin{bmatrix} u_{11} & u_{12} & u_{13} \\ 0 & u_{22} & u_{23} \\ 0 & 0 & u_{33} \end{bmatrix} \tag{4.1}$$

LU decomposition is particularly attractive when solving the same left hand-side Ax for many different right hand sides i.e. b. In effect, the decomposition effort will be a one-off overheard and then x will be solved repeatedly for different b using forward and back substitutions. The total time, $\mathcal{C}$, required by the LU decomposition solver is approximately:

$$\mathcal{C} \approx \mathcal{T} \times \left[\underbrace{\frac{n^3}{3}}_{LU} + i \underbrace{(\frac{n^2}{2} + \frac{n^2}{2})}_{FS+BS} \right] \tag{4.2}$$

where $\mathcal{T}$ is the time needed to execute a multiply-divide floating-point operation (flop), i is the number of iterations for the same coefficient matrix A; and LU, FS, and BS stand for LU factorisation, forward and back substitutions respectively. From Equation 4.2, it is clear that for $i \gg n$, the time needed for the LU factorisation becomes negligible and hence giving an overall complexity of $\mathcal{O}(n^2)$ compared to $\mathcal{O}(n^3)$ for Gaussian elimination [155]. Over the years, a great deal of research has been conducted

to find efficient ways to perform LU decomposition. Although many algorithms exist, the generic algorithm can be written as three nested loops as follows:

Algorithm 4.1 LU Decomposition Generic Pseudo Code

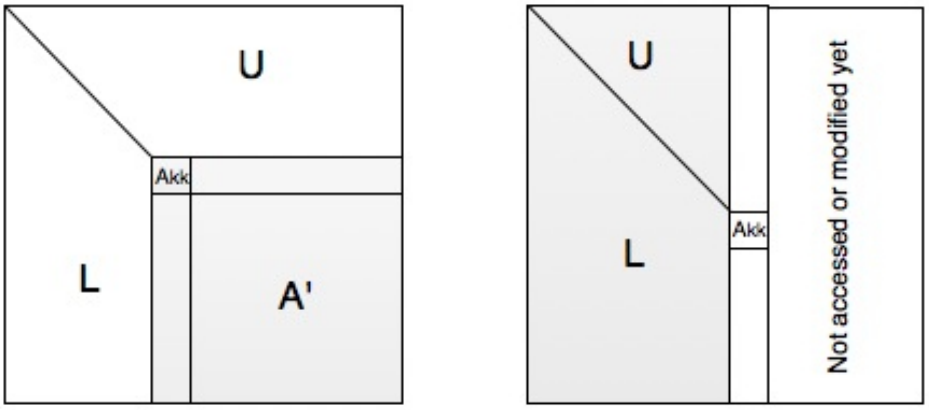
```
1: for ——— do
2:   for ——— do
3:     for ——— do
4:       a_ij = a_ij − (a_ik × a_kj) /a_kk
5:     end for
6:   end for
7: end for
```

The loop indices have variable names i, k, and j, but have different ranges and as such were left empty in Algorithm 4.1. The organisation of these neested loops imply that six possible permutations are possible of i, k, and j in the nested loops. In [156], *Dongarra et al.* studied the performance impact of each permutation for dense LU decomposition algorithm on vector pipeline machines. The division operation is usually performed outside the inner loop, leaving a multiply and a subtract in the innermost loop. The selection a particular loop permutation does not affect of the outcome of LU decomposition or the number of floating-point operations required, provided that pivoting is not employed.

In effect, selecting a different permutation changes the data and computation pattern of the method utilised and may result in a significant performance impact of the computing platform used. The six permutations can be broken down into two groups, namely column-based factorisation and row-based factorisation. The difference between these two groups consists in the role of column and row during the LU decomposition process. Historically, column-based algorithms have been favoured, due to influence of scientific programming languages, such as FORTRAN [157].

Two popular methods of the 6 variants of LU decomposition are right-looking LU and left-looking LU. In right-looking methods, columns below and to the right of the k^{th} pivot of A as accessed and subsequently modified, as shown in Figure 4.1(a). A left-looking LU factorisation, however, computes L and U one column at a time. At the k^{th}

step, it accesses columns 1 to $(k-1)$ of L and column k of A, as shown in Figure 4.1(b). A left-looking LU decomposition is advantageous if the matrix is stored column-wise.



(a) Right-looking LU (b) Left-looking LU

FIGURE 4.1: Right and left looking LU decomposition

4.1.2 Sparse LU Decomposition

Sparse matrices are ubiquitous in scientific calculations when modelling systems with a large number of variables with limited coupling. A sparse matrix is a matrix with enough zeros that it pays to take advantage of them as was defined by *Wilkinson* [158]. In other words, a sparse matrix is defined as one that has few nonzeros in it, typically $\mathcal{O}(n)$ entries, where n is the order of the matrix. Ideally, sparse matrices can benefit from algorithms which exploit their sparsity to reduce the number of operations needed (e.g. avoid operations on zero entires) whilst minimising overall storage requirements (e.g. optimised data structures). As such, the aim of sparse LU algorithms is to solve equations of the form $Ax = b$ in time and space proportional to $\mathcal{O}(n) + \mathcal{O}(nnz)$, for a matrix A of order n with nnz nonzeros [106]. Furthermore, the loose-coupling between elements within sparse matrices enables us to reorder and partition them into almost independent sub-matrices, hence requiring minimal communication and increasing the parallelisation potential.

4.1.2.1 Sparse LU Decomposition Issues

Adapting the numerical methods from the dense LU decomposition to the sparse case, however, introduces extra constraints. One of main issue for sparse LU factorisation is the presence of small or zero values on the main diagonal. To ensure a zero-free diagonal and to maintain numerical stability, pivoting is usually applied. Pivoting is more complex in parallel implementations because the permutation of rows or columns requires global synchronisation between all Processing Elements (PEs). Furthermore, pivoting may cause load imbalance among PEs. Fill-in could undermine efficiency of sparse matrix decomposition. Nonetheless, there are special ordering techniques that can be used to minimise the occurrence of fill-in, as will be discussed in in Section 4.1.3. On the other hands, static symbolic LU factorisation at the preconditioning stage can determine in advance all possible fill-ins. Symbolic LU factorisation algorithms reply only the graph representation of the matrix at hand, which makes them cheaper computationally speaking when compared to the actual numerical factorisation. Symbolic factorisation will be covered in detail in section Section 4.2.1.

Furthermore, sparse LU methods suffer from irregular computation patterns that are dependent on the nonzero structure of the matrix, which in turn depends on the fill-in properties of the matrix and the pivot choices. Nevertheless, symbolic analysis can be used to predetermine the nonzero structure before the numerical factorisation takes place. The symbolic analysis typically requires computations that only depend on the nonzero pattern of the underlying matrix, not the numerical values. This allows the numerical factorisation to be repeated for a sequence of matrices with identical nonzero pattern. If pivoting is employed, however, symbolic analysis has to precede every step of the factorisation process, as it is impossible to predict the nonzero entries without prior knowledge of which matrix elements will be chosen as pivots. As such, static pivoting technique is more suitable for parallel LU factorisation, as it permits a priori identification of pivots, effectively decoupling symbolic and numerical factorisations [159]. In Section 4.3, will show how to leverage static pivoting along with symbolic factorisation to create a task schedule more suitable for a parallel processing.

4.1.2.2 Sparse Matrices Data Structures

Dense matrices are typically represented by a two dimensional array. To save storage, sparse matrices can be represented with more compact data structure such as linked lists, a collection of sparse vectors, or using a coordinate scheme. Each technique has its advantages and disadvantages depending on the application and architecture targeted, as discussed by *Duff et al.* [106]. The Compressed Row Storage (CRS) and the Compressed Column Storage (CCS) formats are the most general as they make no assumptions about the sparsity structure of the matrix and do not store any unnecessary elements. Storing the nonzero elements of a sparse matrix is performed by traversing each column (in the case of CCS) or each row (in the case of CRS), and writing the nonzero elements to an array in the order they appear.

CCS (also called the Harwell-Boeing sparse matrix format) [160] consists of three arrays: *val*, *row_ind* and *column_ptr*. The *val* array stores the values of the nonzero elements of the matrix A as they are traversed in a column-wise fashion. The *row_ind* array stores the row indices of each nonzero. The *col_ptr* array stores the index of the elements in *val* which start a column of A. By convention, *col_ptr* has a length of $(nnz + 1)$ where $col\_ptr[nnz + 1] = nnz$. Thus, the elements of k^{th} column are held in $val\,[col\_ptr[k]]$ through $val\,[col\_ptr[k]]$ and their corresponding row indices are stored in the same locations in *row_ind*.

The CRS format is identical to the CCS format except that A is traversed a row-wise fashion. In other words, the CRS format is the CCS format for A^T. To illustrate the two formats, consider matrix A in Equation 4.3 and its equivalent CCS and CRS formats. CRS and CCS are very economical in terms of memory for sparse matrices as they need only $(2nnz + n + 1)$ storage locations as opposed to $\left(n^2\right)$ for the dense matrix representation [106]. On the other hand, they require an indirect addressing step for every single scalar operation [106].

$$A = \begin{bmatrix} 3 & 0 & 4 & 0 & 2 & 0 \\ 0 & 0 & 1 & 3 & 0 & 0 \\ 5 & 7 & 0 & 0 & 4 & 0 \\ 9 & 0 & 0 & 8 & 0 & 0 \\ 0 & 4 & 0 & 0 & 3 & 1 \end{bmatrix} \tag{4.3}$$

The CCS format for matrix 4.3 is specified by the arrays *val*, *row_ind*, *col_ptr* as follows:

$$\begin{aligned} val &= [3, 5, 9, 7, 4, 4, 1, 3, 8, 2, 4, 3, 1] \\ row\_ind &= [0, 2, 3, 2, 4, 0, 1, 1, 3, 0, 2, 4, 4] \\ col\_ptr &= [0, 3, 5, 7, 9, 12, 13] \end{aligned}$$

And its equivalent CRS format for is specified by the arrays *val*, *col_ind*, *row_ptr* as follows:

$$\begin{aligned} val &= [3, 4, 2, 1, 3, 5, 7, 4, 9, 8, 4, 3, 1] \\ col\_ind &= [0, 2, 4, 2, 3, 0, 1, 4, 0, 3, 1, 4, 5] \\ row\_ptr &= [0, 3, 5, 8, 10] \end{aligned}$$

4.1.2.3 Elimination Graphs

As mentioned earlier, symbolic LU factorisation is a technique whereby the graph representation of the matrix at hand is used to predetermine fill-ins they may appear during the actual numerical step. Symbolic analysis for symmetric matrices is a well understood topic and can be efficiently performed using a pruned version of the undirected graph associated with the matrix, known as "the elimination tree" [161]. The elimination tree is used to precompute the **all possible** positions of fill-ins as well as to identify column dependencies for parallelism. The elimination tree is defined for any sparse matrix whose sparsity pattern is symmetric. For a sparse matrix of order n, the elimination tree is a

tree on n nodes such that node j is the father of node i if entry $(i, j), j > i$ is the first entry below the diagonal in column i of the triangular factors. Figure 4.2(a) shows a matrix and its corresponding elimination tree. For instance, columns 1 and 2 can be processed in parallel as they do not have any dependencies (i.e. no offsprings). However, column 4 cannot be processed unless column 2 have been already processed. Smilarly, columns 3 and 4 can be processed in parallel once their column offsprings (i.e. columns 1 and 2 respectively) have been evaluated.

An analogous graph for asymmetric sparse matrices is the elimination Directed Acyclic Graph (elimination DAG) [162], which was introduced by *Gilbert and Liu* [162]. The main property that we can exploit in these elimination graphs is that computations corresponding to nodes that are not ancestors or descendants of each other are independent [163]. Thus, the elimination graph can be used to exploit parallelism. In effect, the dependency in terms column-level updates order is determined by the elimination graph. If each node is associated with a column, a column can only be modified by columns corresponding to nodes that are descendants of the corresponding node in the elimination graph. Elimination DAGs will be used extensively in Section 4.3, as part of our work to develop a dependency-driven scheduling algorithm for parallel sparse matrix factorisation.

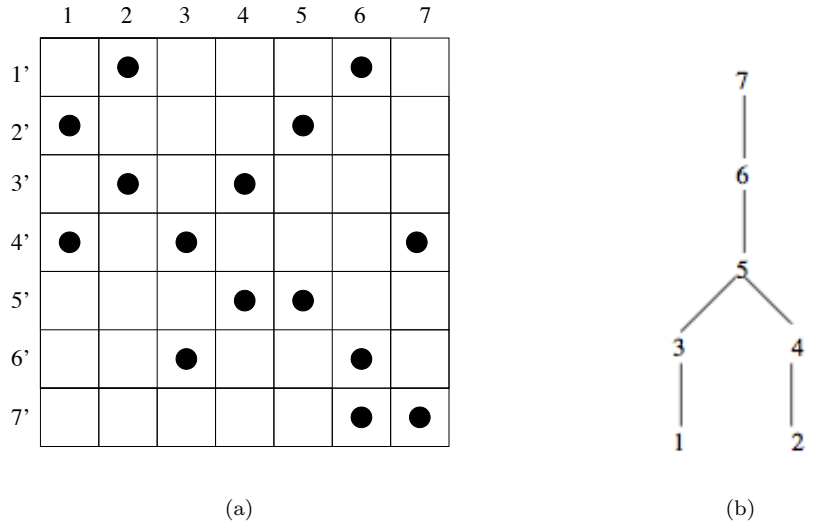
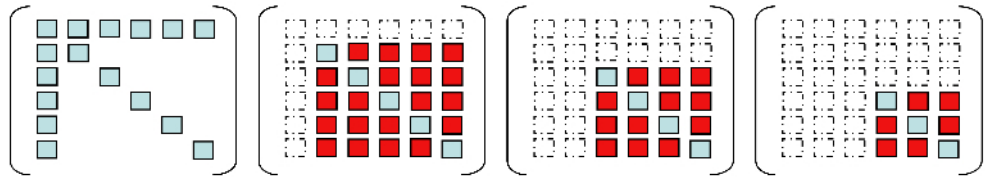


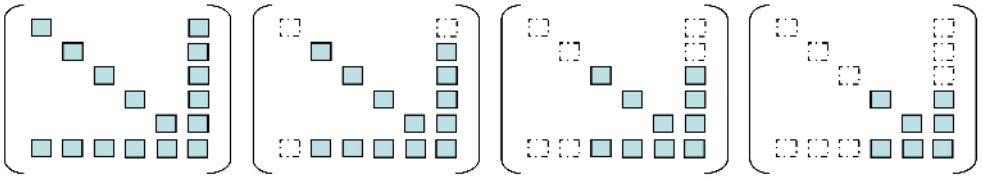
FIGURE 4.2: (a) A matrix and its (b) elimination tree

4.1.3 Fill-reducing Orderings

Fill-in during sparse LU decomposition is caused by the nonzero structure of the matrix prior to and during the LU decomposition process. In order to limit the amount of fill-in that occurs, the nonzero structure of the sparse matrix can be altered by reordering the rows or columns of the matrix prior to LU decomposition. Figure 4.3 shows the effect of reordering on the amount of fill-in generated during the the factorisation process, where the blue and red boxes represent the the initial nonzero and fill-ins respectively. Reordering only affects the order of the variables in the system of equations, or the order in which the equations are eliminated during LU decomposition.



(a) Before ordering



(b) After ordering

FIGURE 4.3: The effect of ordering on fill-in during LU factorisation

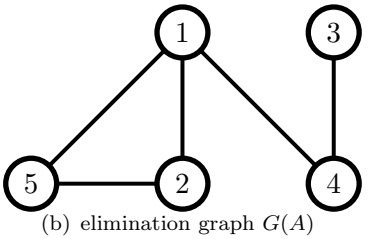
Mathematically speaking, the fill-in minimisation problem consists in finding a row and column permutation P and Q such that the number of nonzeros in the factorisation of PAQ, or the amount of work required to compute it, are minimised. However, *Rose and Tarjan* [164] have proved that finding the best ordering for symmetric matrices which results in minimum fill-in is an NP-complete problem. *Yannakakis* [107] proved the same for asymmetric matrices. In effect, allowing fill-in may be computationally cheaper than finding such an ordering. Therefore, heuristics that attempt to reduce fill-in are used instead. Ordering schemes typically take into account only the matrix structure, without considering the numerical values of its elements. Partial pivoting during factorisation

changes the row permutation P and hence could potentially increase fill-in, compared to the estimate produced by the ordering scheme prior to factorisation process.

Ordering heuristics are essentially graph-based algorithms. In fact, any symmetric matrix corresponds to an undirected graph called the elimination graph. To construct such a graph, a vertex is associated with each row and edge from i to j exists if a_{ij} is nonzero, as shown in Figure 4.4. Graphically, fill-ins are equivalent to the new edges introduced to the nodes connected to the node to be eliminated when removed. Figure 4.5(a) shows the matrix A of Figure 4.4 after the first elimination step (i.e. A_1), where X denotes an initial nonzero elements and F is the incurred fill-in. Figure 4.5(b) shows the elimination graph associated with A_1, where the dashed lines represent the fill-ins (i.e. new edges) introduced where the first node was removed.

$$\begin{bmatrix} X & X & 0 & X & X \\ X & X & 0 & 0 & X \\ 0 & 0 & 0 & X & 0 \\ X & 0 & X & X & 0 \\ X & X & 0 & 0 & X \end{bmatrix}$$

(a) symmetric matrix

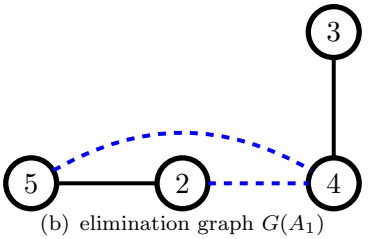


(b) elimination graph $G(A)$

FIGURE 4.4: A square symmetric matrix and its equivalent elimination graph

$$\begin{bmatrix} X & X & 0 & X & X \\ X & X & 0 & F & X \\ 0 & 0 & 0 & X & 0 \\ X & F & X & X & F \\ X & X & 0 & F & X \end{bmatrix}$$

(a) A_1



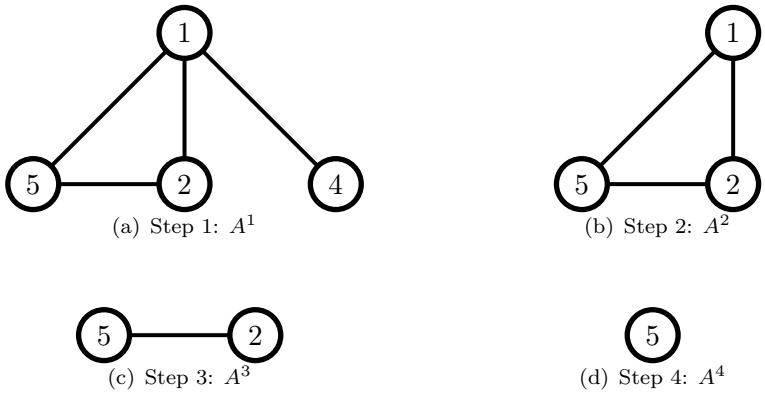
(b) elimination graph $G(A_1)$

FIGURE 4.5: Elimination graph after the first elimination step

Although finding the optimal ordering is NP-complete [164], in practice there are several efficient fill-in reducing heuristics. They can be grouped into two classes; local and global heuristics. The first class uses local greedy heuristics to reduce the number of fill-ins at each step of factorisation. One of the representative heuristics is the minimum degree algorithm. The second class is based on global heuristics that uses graph partitioning, such as nested dissection, to restrict the fill to only specific blocks of the permuted matrix.

4.1.3.1 Minimum Degree Ordering

The minimum degree algorithm [165] is a widely used heuristic for finding a permutation P such that PAP^T has fewer nonzeros in its factorisation. The key idea of the minimum degree algorithm is to select the node which has the the least number of edges connected to it (i.e. minimum degree) as the next elimination node. Figure 4.6 shows the elimination step of the matrix shown in Figure 4.4, following a minimum degree fashion incurring no fill-ins. If the input matrix A is asymmetric, then the permutation of the matrix $A + A^T$ can be used. This is know as symmetrisation. Approximate Minimum Degree (AMD) [166] improves the conventional minimum degree algorithm, in terms of time and memory usage. Another variant specifically created for asymmetric matrices is known as Column Approximate Minimum Degree (COLAMD) [167]. COLAMD orders the matrix AA^T without forming it explicitly.



(a) Step 1: A^1 (b) Step 2: A^2

(c) Step 3: A^3 (d) Step 4: A^4

FIGURE 4.6: Minimum degree elimination steps

4.1.3.2 Nested Dissection Ordering

Nested dissection [168] uses a divide and conquer strategy on the graph of a sparse symmetric matrix to find an elimination ordering. The key concept is the computation of a vertex separator, that splits the matrix into new roughly equal-sized subgraphs on which LU factorisation may be performed separately. The variables corresponding to the first part are ordered, followed by those of the second part, and finally by those of

the separator. The disconnected parts can be themselves further divided by the computation of new separators, with the recursion continuing to any depth. The results for the two parts may then be combined to find the solution of the entire graph. The main advantage of this partitioning is that the resulting form of the matrix is suitable for parallel execution. State-of-the-art nested dissection algorithms use multilevel partitioning. A widely used nested dissection routine is *"METIS NodeND"* from the METIS graph partitioning package [169].

It has been observed in practice that minimum degree is better at reducing the fill for smaller problems, while nested dissection works better for larger problems. This observation has lead to the development of hybrid heuristics that consist in applying several steps of nested dissection, followed by the usage of a variant of the minimum degree algorithm on local blocks [170]. For asymmetric matrices, the algorithms discussed above use the graph associated with the symmetrised matrix $A + A^T$ or $A^T A$. The approach of symmetrising the input matrix works well in practice when the matrix is almost symmetric. However, when the matrix is very asymmetric, the information related to the asymmetry of the matrix is not exploited, as too many *"false"* dependencies are created [106].

4.1.4 Zero-free Diagonal Orderings

As previously discussed in Section 3.2, circuit matrices are mostly diagonally-dominant and enjoy a largely zero-free diagonal. However, they can be permuted, by *Duff's* maximum transversal algorithm [103, 171], to ensure a zero-free diagonal. The algorithm works by determining the maximum possible transversal of the underlying matrix. A transversal is defined as a set of nonzeros on the diagonal of the permuted matrix. A transversal of maximum length is the maximum transversal. *Duff's* algorithm attempts to find the maximum transversal on a graph, in which each vertex corresponds to a row in the matrix at hand. An edge $i_k \rightarrow i_{k+1}$ exists in the graph if $A(i_k, j_{k+1})$ is a nonzero and $A(i_k + 1, j_{k+1})$ is an element in the transversal set. *Duff's* maximum transversal transversal algorithm has a worst case time complexity of $O(n\mathcal{T})$ where $\mathcal{T}$ is the number

of nonzeros in the matrix and n is the order of the matrix. However, in practice, the time complexity is closer to $O(n + \mathcal{T})$ [172].

The maximum transversal problem can also be interpreted as a maximal matching problem on bipartite graphs [173], as illustrated by the example problem in Figure 4.7. In most of our experiments, we use the HSL_MC64 ordering subroutine [174] to ensure that our test matrices have a zero-free diagonal. The subroutine attempts to find row and column permutations such that the permuted matrix has n entries on its diagonal, where n is the order of the matrix. If the matrix is structurally nonsingular, the subroutine can also compute a row and column permutation of the matrix so that the sum of the diagonal entries of the permuted matrix is maximised. This helps to put big nonzeros values on the diagonal and thus increases numerical stability during the LU factorisation process.

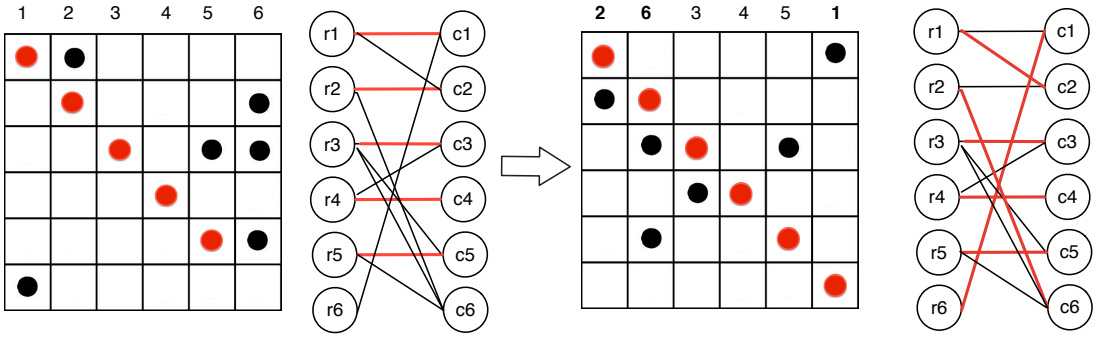


FIGURE 4.7: Example of finding a zero-free diagonal matrix permutation via maximal matching on a bipartite graph.

4.2 Parallelising Sparse LU Decomposition

One of the most important aspects of designing any parallel algorithm is identifying of the appropriate level of granularity, which can be then adequately mapped to the targeted processing architecture [175]. For instance, fine-grain parallelism (i.e. at the level of individual floating point operations) is available in either the dense or sparse linear systems. It can be exploited effectively by using a streaming-like processing architecture such as a vector processor or a systolic array. Medium-grain parallelism arises from

the fact that many column operations can be computed concurrently across a number of processing elements. An elimination tree-like graph can be used to characterise this type of parallelism such that columns in the same graph level can be evaluated in parallel. This level of granularity is an extremely important source of parallelism for sparse matrix factorisation, as sparsity increase the the number of columns that can be operated on in parallel. This may, however, cause a load imbalance in the the case where an entire column operation only requires a few floating point operations.

Large-grain parallelism for space matrices can be also identified by the means of a tree-like elimination graph. Therefore, if T_i and T_j are disjoint sections of the elimination graph, then all of the columns corresponding to nodes in T_i can be computed completely independently of the columns corresponding to nodes in T_j, and vice versa. Thus, these computations can be done concurrently on separate processing elements with no communication between them. In the dense case, however, operations must be performed sequentially as there is never more than one leaf node at any given time. It should be also noted that structure of the elimination graph is highly dependant on the fill-in properties of the matrix, which is in turn depends on the ordering heuristics used. Roughly speaking, sparsity and parallelism are largely compatible, since the large-grain parallelism is due to sparsity in the first place. As such, an ordering that increases sparsity can also increase the parallelism potential.

Many parallel sparse system solvers employ a technique called the "the multifrontal scheme" [101] to parallelise computations by rewriting the original problem into a collection of "frontal matrices" . In effect, multifrontal solvers [104, 176] rely on a directed acyclic graph, called an assembly DAG, to extract and organise the parallel work. Each node (i.e. frontal matrix) of the DAG represents a given computation. This may include pivot eliminations, normalisation, and handling data from the offsprings. All leaf nodes of the DAG (i.e nodes without an offspring) can be evaluated in parallel, while internal nodes can only be computed once their children have been computed. A pool of the available work, that is, the nodes in the tree that are available for computation, is maintained in shared memory. This multifrontal approach, if organised correctly, can provide large and medium grain parallelism. However, the method is best suited for

matrices with near-symmetric patterns and where the pivot sequence is constrained. Moreover, this method involves relatively significant amounts of data exchanges between the tree nodes, requiring a considerable communication bandwidth. Therefore, multifrontal solvers work best in shared memory environments.

Another approach to parallel sparse solvers revolves around evaluating many pivots in parallel [177, 178]. At each stage of the the factorisation, these algorithms maintain a list of pivots that can be applied in parallel and perform the corresponding updates. These solvers typically concentrate on the medium and fine grain parallelism, and tend to be most efficient on a moderate number of processors with fairly tight synchronisation [179]. An important part of any sparse solver is the algorithm controlling the amount of fill-in that is generated during the solution process. other aspect of pivot selection is the maintenance of stability. Typically, this is done by choosing a pivot element that is within a specified multiple of the largest element in the pivot row or pivot column or the active part of the matrix depending on the efficiency of these tests given the data structures assumed.

The stability and sparsity requirements for pivot selection are often contradictory and most strategies involve some sort of a compromise. Selecting pivots for parallelism add a third constraint. For the medium and fine grain algorithms mentioned above, these three constraints can be considered in a reasonably straightforward way, potentially with respect to the entire active portion of the matrix. The exploitation of larger grain parallelism, however, often imposes a static decomposition on the structure of the matrix which further constrains pivot selection. The effect of these constraints, for asymmetric problems, can be seen by considering tearing techniques or nested bisection. These techniques have proposed to expose large-grain structure, suitable for parallel execution, by reordering the matrix into a form such as the Bordered Diagonal Block (BDB) form [180], as will be demonstrated in Chapter 5.

4.2.1 Gilbert-Peierls' Algorithm

In Section 4.1.2, we mentioned that the aim of a sparse LU algorithm is to solve the linear system $Ax = b$ in time and space proportional to $\mathcal{O}(n) + \mathcal{O}(nnz)$, for a matrix A of order n with nnz nonzeros [106]. In practice, this is much harder to achieve as the underlying nonzero structure of the matrix may dramatically change in course of factorisation. To tackle this issue, *Gilbert and Peierls* [181] proposed a left-looking sparse LU algorithm that achieves an LU decomposition with partial pivoting, in time proportional to the floating-point operations performed i.e. $\mathcal{O}(flops(LU))$. It is called a left-looking algorithm because it computes k^{th} column of L and U only by using the already computed columns 1 to $(k-1)$. In other words, to compute k^{th} column of L and U, the algorithm needs only to look at the already computed columns that are to the left of the current column, as shown by the shaded portion of the matrix in Figure 4.8. Appendix A details how a left-looking decomposition can be mathematically derived from the general Gaussian Elimination algorithm [182].

The core of the Gilbert-Peierls factorisation algorithm is solving a lower triangular system $Lx = b$, where L is a spare lower triangular matrix, x and b are sparse vectors [102]. It consists of a symbolic step to determine the nonzero pattern of x and a numerical step to compute the values of x. This lower triangular solution is repeated n times during the entire factorisation (where n is the size of the matrix) and each solution step computes a column of the L and U factors. The entire left-looking algorithm is described in Algorithm 4.2. The lower triangular solution (i.e. line 3) is the most expensive portion of the Gilbert-Peierls algorithm and includes a symbolic and a numeric factorisation step.

Algorithm 4.2 Gilbert-Peierls LU factorisation of a n-by-n asymmetric matrix A

1: $L = I$
2: **for** $k = 1$ to n **do**
3: solve the lower triangular system $Lx = A(:k)$
4: do partial pivoting on x
5: $U(1:k, k) = x(1:k)$
6: $L(k:n, k) = x(k:n)/U(k, k)$
7: **end for**

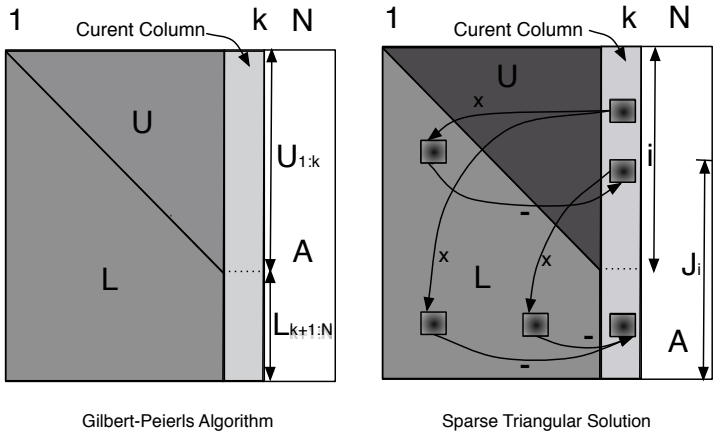


FIGURE 4.8: Gilbert-Peierls Algorithm Data Flow Pattern [181]

4.2.1.1 Symbolic Analysis

As mentioned in the previous paragraph, Gilbert-Peierls Algorithm revolves around the efficient solution of $L_k x = b$ in order to compute the k^{th} column, where L_k is a unit diagonal representing the already computed $(k-1)$ columns and the column vector b is sparse. By avoiding unnecessary operations on zero entries, the general forward substitution Algorithm can be described as follows:

Algorithm 4.3 Sparse forward substitution - Version 1

1: $x = b$
2: **for** $j = 1$ to n **do**
3: **if** $x_j \neq 0$ **then**
4: **for** *each* $i > j$ for which $l_{ij} \neq 0$ **do**
5: $x_i = x_i - l_{ij} x_j$
6: **end for**
7: **end if**
8: **end for**

If Algorithm 4.3 is implemented, the time taken would be $\mathcal{O}(n + nnz + f)$ where nnz is the number of nonzeros and f is the number of floating operations performed. Since typically, $f > nnz$, the overall time approximates $\mathcal{O}(n + f)$. However, the process is repeated n times in order to compute all the columns of the LU factors leading to a $\mathcal{O}(n^2)$ factorisation time. Algorithm 4.3 can be optimised further if we can replace the outer loop (i.e. line 2) with a smaller list $\mathcal{X}$ of j indices for which we know x_j will

a be nonzero, $\mathcal{X} = \{j \mid x_j \neq 0\}$, in ascending order. In effect, this would reduce the computation time to $\mathcal{O}(f)$. The refined algorithm is shown in Algorithm 4.4.

Algorithm 4.4 Sparse forward substitution - Version 2

```
1: x = b
2: for each j ∈ X do
3:    for each i > j for which l_ij ≠ 0 do
4:       x_i = x_i − l_ij x_j
5:    end for
6: end for
```

Symbolic analysis is the process whereby the set $\mathcal{X}$ is defined. From the pseudo code in Algorithm 4.4, it can be seen that entries in x can become nonzero in only two places, namely, the first and the fourth lines. If numerical cancellation is ignored, these two statements can be written as two logical implications 4.4 and 4.5 respectively.

$$line\ 1 : [b_i \neq 0 \implies x_i \neq 0] \tag{4.4}$$

$$line\ 4 : [x_j \neq 0 \wedge \exists i(l_{ij} \neq 0) \implies x_i \neq 0] \tag{4.5}$$

These two implications can be expressed as a graph traversal problem. Let G_{L_k} be the directed graph of L_k such that $G_{L_k} = (V, E)$ with nodes $V = \{1 \ldots n\}$ and edges $E = \{(j, i) \mid l_{ij} \neq 0\}$. Thus, statement 4.4 is equivalent to marking all the nodes of G_{L_k} that are nonzeros in the vector b, whereas statement 4.5 implies that if a node j is marked and it has an edge to a node i, then the latter must be also marked. Figure 4.9 graphically highlights these two relationships.

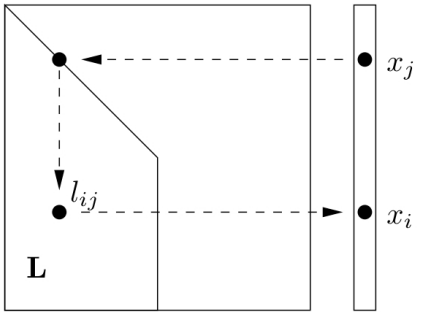


FIGURE 4.9: Nonzero pattern for a sparse triangular solve

Therefore, if we have a set $\mathcal{B} = \{i \mid b_i \neq 0\}$ that denotes the nonzeros of b, the nonzero pattern $\mathcal{X}$ can be computed by the determining the vertices that are reachable from the vertices of the set $\mathcal{B}$ i.e. $\mathcal{X} = Reach_{G_L}(\mathcal{B})$. The reachability problem can be solved using a classical depth-first search in G_{L_k} from the vertices of the set $\mathcal{B}$. The depth-first search takes time proportional to the number of vertices examined plus the number of edges traversed. The depth-first search does not sort the set $\mathcal{X}$, however, it computes its topological order. This topological ordering is useful to maintain the precedence relationship in the eliminating process of the numerical factorisation step. The computation of $\mathcal{X}$ and x both take time proportional to the floating-point operation count [155].

To illustrate the overall process, consider the solutions of the sparse linear system $Lx = b$ for sparse x, using the sparse lower triangular matrix L and the sparse vector b shown in Figure 4.10. Vector b has two nonzero elements are at indices $\{4, 6\}$. Therefore, we perform the reachability function using the following set, $\mathcal{B} = \{4, 6\}$. Then starting a depth-first search at node 4 gives $Reach(4) = \{4, 9, 12, 13, 14\}$ in topological order. Next, $Reach(6) = \{6, 9, 10, 11, 12, 13, 14\}$, but some of these nodes are already marked. So the final set $\mathcal{X} = \{6, 10, 11, 4, 9, 12, 13, 14\}$, which is also in topological order. The forward solve traverses the columns of L in this order.

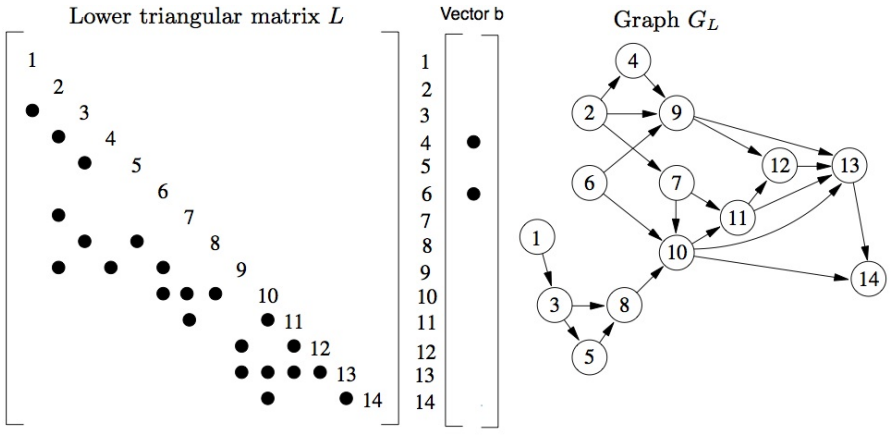


FIGURE 4.10: Example of a symbolic analysis for a lower triangular sparse system [155]

4.2.1.2 Numerical Factorisation

Normally, this step consists of numerically performing the sparse triangular solution for each column k of L and U in the the increasing order of the row index, as shown in Figure 4.8. The nonzero pattern computed by the symbolic analysis is, however, in a topological order. Sorting the indices would increase the time needed for the solutions. Nevertheless, topological order is sufficient as it gives the order in which elements of the current column are dependent on each other. For instance, the depth first search would have finished traversing vertex i before it finishes traversing vertices j. Therefore, in the topological order j would appear before i. The entire left-looking algorithm can be summarised in MATLAB notation in Figure 4.11, where $x = L\backslash b$ denotes the solution of a sparse lower triangular system.

```
%————————————————————————————
% Gilbert–Peierls Algorithm (A=LU)
% input: sparse matrix A
% output:  L and U factors
%————————————————————————————
L = I               % I is the identity matrix
for k = 1 : n
    b = A( : , k);  % kth column of A
     x = L \  b;    % the backslash \ is MATLABs Lx=b solve function
     U(1:k,k) = x(1 : k);
     L(1+k : n) = x(k+1 : n) / U(k, k);
end;
```

FIGURE 4.11: Gilbert-Peierls Algorithm (A=LU) in the MATLAB notation

```
%————————————————————————————
% Sparse Triangular Solution (Lx=b)
% input: Matrix L (1: k–1)
% output:  kth column of L
%————————————————————————————
x = b;
for i = 1 : k–1 where x(i) !=0
    for j = i+1 : N where L(j,i) != 0
        x(j) = x(j) – L(j,i) * x(i);
    end;
end;
```

FIGURE 4.12: Pseudocode of the Sparse Triangular Solution (Lx=b)

4.2.1.3 Symmetric Pruning

Symmetric pruning is technique whereby structural symmetry in matrices is exploited to reduce the time taken by the symbolic analysis [183]. The basic idea of the technique revolves around decreasing the time taken by the depth-first search by pruning unnecessary edges in the graph of a matrix (i.e. G). In effect, G can be replaced by a reduced graph H that has fewer edges but preserves the path structure. In fact, any graph H can be used in lieu of G if it preserves the paths between vertices of the original graph. In other words, if an edge $i \rightarrow j$ exist in G, it should also exist in H.

Figure 4.13 illustrates how symmetric pruning works. As demonstrated in the example, an edge $r \rightarrow s$ is removed (i.e. pruned) by setting $l_{sr} = 0$, provided that $l_{jr} \neq 0$ and $u_{rj} \neq 0$. The justification behind this is that for any a_{rk}, a_{sk} will still fill-in from column r. The just computed column j of L is used to prune earlier columns. This means that any future depth-first search from vertex i will not visit vertex s, since s would have been already visited via vertex j. In the the context of LU factorisation, the graph of L, (G_L) can be pruned by leveraging the symmetry in the structure of the factors L and U. In our work, will use symmetric pruning to speed up the depth-first search in the symbolic factorisation stage of the Gilbert-Peierls Algorithm, covered in Section 4.2.1.1.

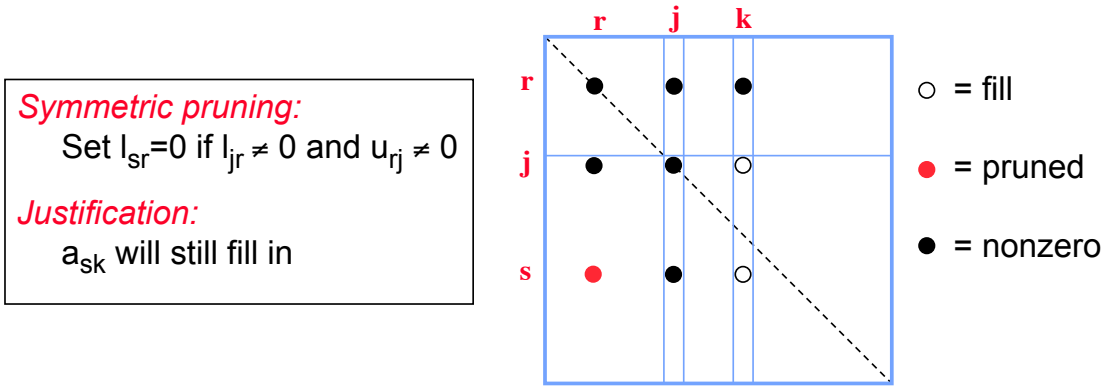


FIGURE 4.13: Symmetric pruning example [183]

4.3 Dependency-Aware Matrix Operations Scheduling

In this section, we explain one of the main contributions of this thesis, which revolves around the construction of a deterministic and accurate task model for parallel LU factorisation. As such, we present our Dependency-Aware Matrix Operations Scheduling (DAMOS) algorithm. DAMOS is a scheduling algorithm that leverages the graph representation of a matrix, computed using symbolic factorisation, to create an operations schedule that takes into account column-level dependencies. The generated static scheduled can be then used to parallelise and control the dataflow of LU matrix operations on the FPGA. The main steps of the algorithm are as follows:

1. Preorder matrix A to minimise fill-in (e.g. minimum degree) and to ensure a zero-free diagonal (e.g. maximum traversal).
2. Perform symbolic factorisation and determine the structure of the lower triangular matrix L and upper triangular matrix U.
3. Determine column dependencies using the structure of upper triangular matrix U.
4. Building a Directed Acyclic Graph (DAG) that represents the computed column-level dependencies.
5. Annotate nodes of the Column-Dependency DAG (CD-DAG) with their corresponding level of parallelism.
6. Derive the ASAP (As Soon As Possible) schedule for the column operations required.
7. Refine the ASAP schedule using *modulo i scheduling*, where i is the maximum number of columns that can reside at any level of the CD-DAG.

To illustrate how our DAMOS algorithm works, consider the matrix A shown in Figure 4.14. For the sake of simplicity, it is assumed that the matrix has a zero-free diagonal and it has been already pre-ordered with some fill-in minimising heuristic. First of all, we need to carry out the Gilbert-Peierls factorisation symbolically, using the principles studied in Section 4.2.1, to work out the pattern of the LU factors.

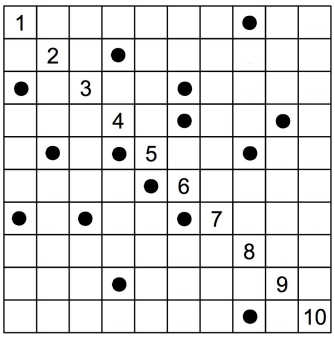


FIGURE 4.14: Matrix A with an asymmetric nonzero pattern

We compute the nonzero structure of the LU matrix column by column starting from the left, following the left-looking LU factorisation pattern of the Gilbert-Peierls algorithm. Therefore, in order to compute the k^{th}column, we first need to construct a Direct Acyclic Graph (DAG) of L_{k-1} (i.e. $G_{L_{k-1}}$), where L_{k-1} is the unit lower triangular matrix of the columns that has been computed so far (i.e 1 to (k-1) columns). The graph $G_{L_{k-1}}$ has an edge $j \rightarrow i$ if $l_{ij} \neq 0$. Then the nonzero pattern of the k^{th} column is given by the reach of nonzero elements of column k in $G_{L_{k-1}}$. In other words, if we have a set $\mathcal{B} = \{i \mid b_i \neq 0\}$ that denotes the existing nonzeros of the k^{th} column, the new nonzero pattern can be computed by the determining the vertices that are reachable from the vertices of the set $\mathcal{B}$ i.e. $Reach_{G_{L_{k-1}}}(\mathcal{B})$. In practice, the reachability problem is solved using a Depth-First Search (DFS) algorithm, as it was demonstrated inSection 4.2.1.1. For the sake of simplicity, we will visually identity of the reachable vertices in our subsequent examples.

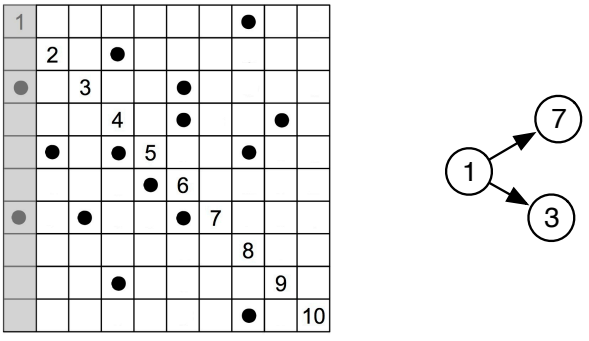


FIGURE 4.15: Symbolic Gilbert-Peierls factorisation example: step 1.

For instance, to compute the nonzero pattern of column 2, we need to construct the graph of the lower components of columns to its left (i.e Column 1 in Figure 4.15). The columns required at any step of the factorisation process are represented by the shaded portion of the matrix in all the subsequent figures of this section. In column 2, there are two nonzeros at indices {2, 4}. Therefore, $Reach(2) = \{2\}$, $Reach(4) = \{2\}$ and hence $Reach(2, 4) = \{2, 4\}$. We can see that the reachability function has returned the input set itself. This implies that column 2 structure remains unchanged and it will not suffer from any fill-in during the actual numerical factorisation process. The structure of columns 3, 4, 5 also remains unchanged, as can be seen from the symbolic factorisation steps illustrated in Figure 4.16.

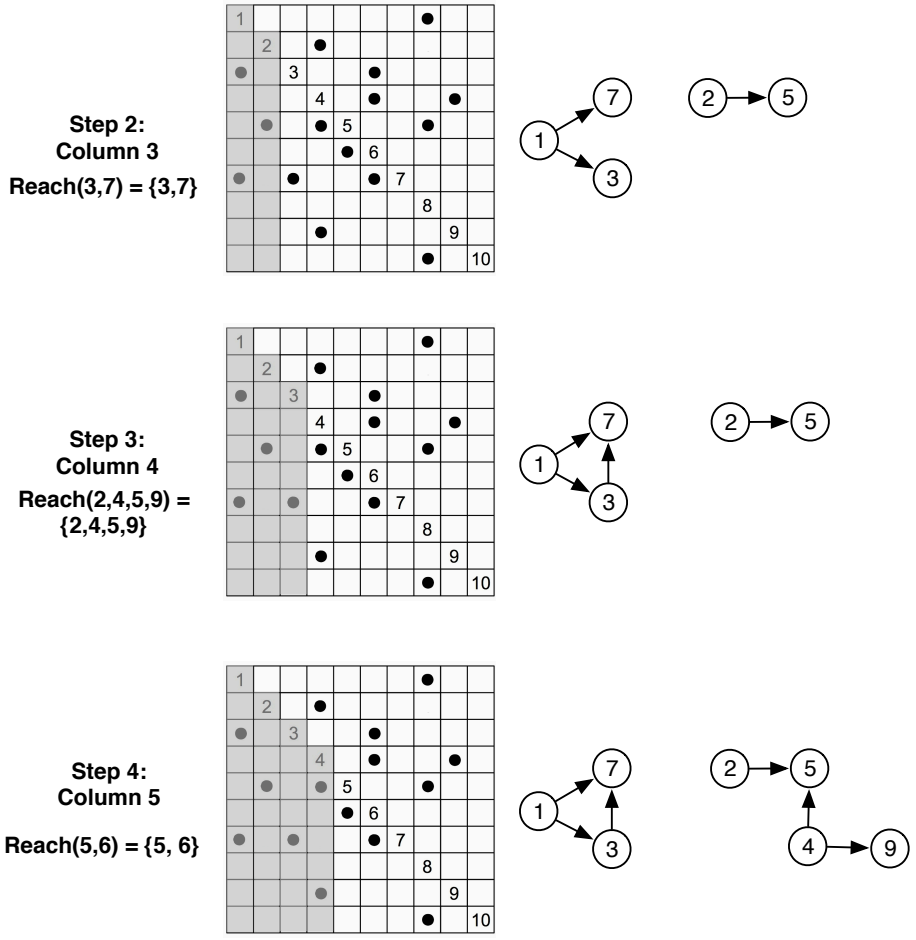


FIGURE 4.16: Symbolic Gilbert-Peierls factorisation example: step 2 - step 4.

Starting from step 5, however, we start to see the impact of fill-in on the nonzero structure of the matrix. In effect, column 6 has four nonzero elements at indices {3, 4, 6, 7}. The new nonzero pattern of column 6, including fill-ins, is given by Equation 4.6-4.8:

$$Reach(3,4,6,7) = Reach(3) \cup Reach(4) \cup Reach(6) \cup Reach(7) \tag{4.6}$$

$$= \{3,7\} \cup \{4,5,6,9\} \cup \{6\} \cup \{7\} \tag{4.7}$$

$$= \{3,7,4,5,6,9\} \tag{4.8}$$

$$Fillin(Col_6) = Reach(3,4,6,7) - \{3,4,6,7\} \tag{4.9}$$

$$= \{3,7,4,5,6,9\} - \{3,4,6,7\} \tag{4.10}$$

$$= \{5,9\} \tag{4.11}$$

From Equation 4.9-4.11, on the other hand, we can see that we can also expect the appearance of two fill-in elements at indices $\{5,9\}$ in the new nonzero structure of the column 6. $Fillin(Col_k)$ is a function that returns the row indices of the new fill-ins in column k. Figure 4.18 shows the remaining steps of the symbolic factorisation. Figure 4.17 shows the resulting matrix structure once all the steps of the symbolic factorisation are performed.

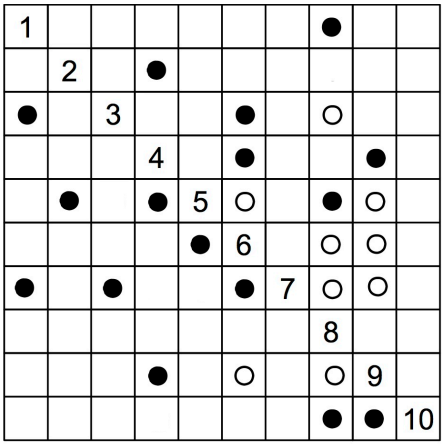


FIGURE 4.17: The predicted the nonzero pattern of the LU factors of matrix A.

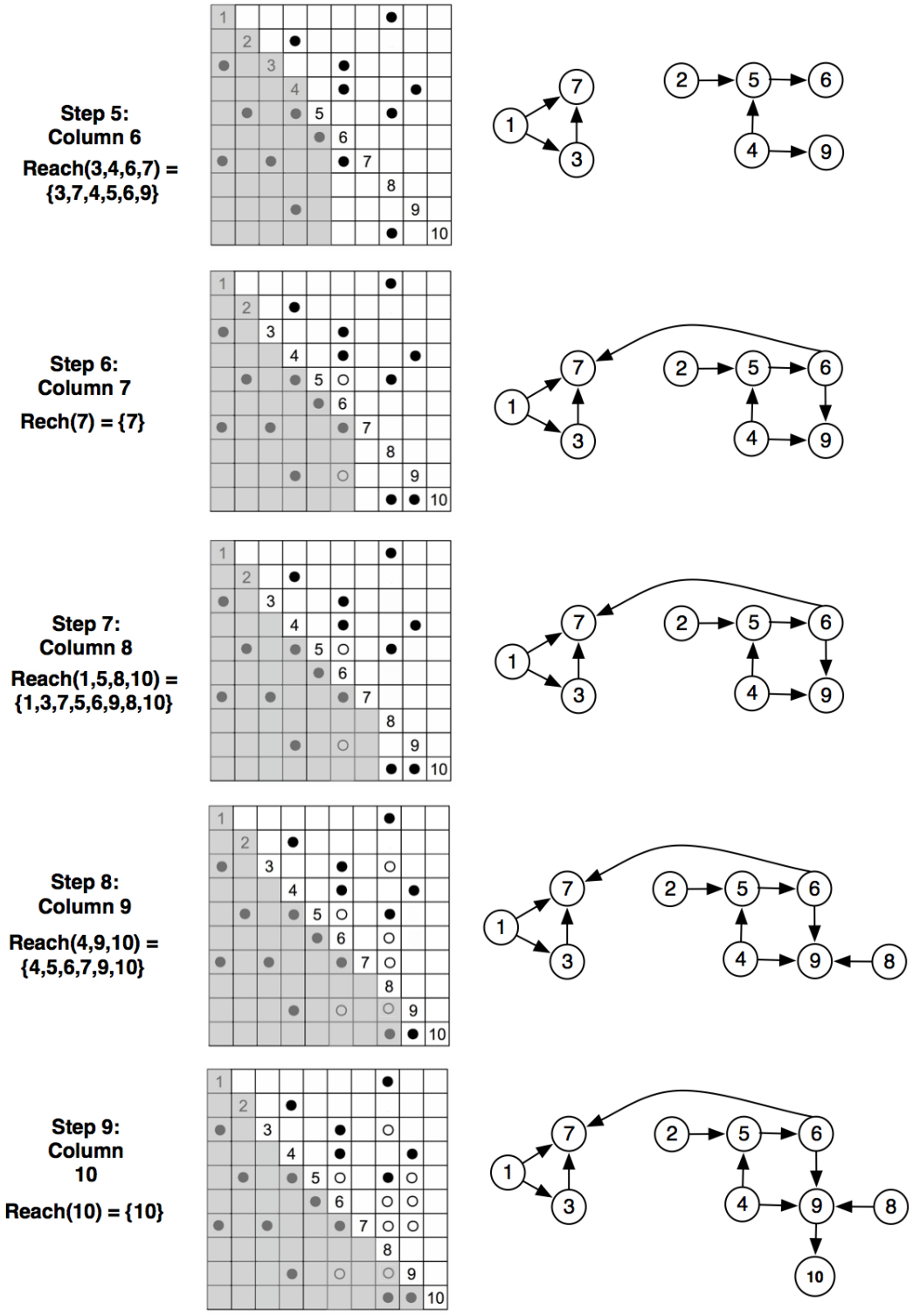


FIGURE 4.18: Symbolic Gilbert-Peierls factorisation example: step 5 - step 9.

Now that we have computed the nonzero pattern of resulting LU factors, we need to determine the columns dependencies that may arise during the numerical factorisation process. In Gilbert-Peierls' algorithm, the flow of computation follows two steps, which are repeated sequentially until the entire matrix is processed. The first step is *"the sparse triangular solution"*, in which the elements of the current column are factorised by the means of solving $Lx = b$ for x, where L represents the triangular matrix of leftmost columns factorised so far, b is the current column to be decomposed, and x is the decomposed column. In the next step, the computed column is normalised by dividing all its lower off-diagonal elements over the pivot. As the column normalisation operation is self-contained (i.e. does not require any other column), it is clear that any column dependencies in the overall Gilbert-Peierls algorithm only arise from the underlying dependencies in the *"the sparse triangular solution"* step. However, when computing a column k using the sparse triangular solution algorithm, not all the columns to its left are needed, as it was illustrated in Section 4.2.1.1. In effect, the factorisation of column k only depends on the columns that satisfy the following criteria:

$$Dependency(Col_k) = \{j | a_{jk} \neq 0, j < k\} \tag{4.12}$$

In other words, column-level dependency information can be derived by just analysing structure of U matrix, which is computed in the symbolic factorisation phase. Applying this principle to our example factored matrix A (i.e LU), gives the following:

$$Dependency(Col_1) = \{\} \tag{4.13}$$
$$Dependency(Col_2) = \{\} \tag{4.14}$$
$$Dependency(Col_3) = \{\} \tag{4.15}$$
$$Dependency(Col_4) = \{2\} \tag{4.16}$$
$$Dependency(Col_5) = \{\} \tag{4.17}$$
$$Dependency(Col_6) = \{3, 4, 5\} \tag{4.18}$$
$$Dependency(Col_7) = \{\} \tag{4.19}$$
$$Dependency(Col_8) = \{1, 3, 5, 6, 7\} \tag{4.20}$$
$$Dependency(Col_9) = \{4, 5, 6, 7\} \tag{4.21}$$
$$Dependency(Col_{10}) = \{\} \tag{4.22}$$

Information conveyed by Equation 4.13-4.22 can be graphically presented with the aid of Directed Acyclic Graph (DAG), such that if column k depends on column i, then a directed edge exist from node i to node k (i.e. $i \rightarrow k$). We call such graph a DAMOS Scheduling Graph. In the latter, leaf nodes are eliminated first, then their parents, and processing carries on upwardly until all nodes are eliminated. This implies that a parent node cannot be eliminated unless all its children have been processed. Two columns are said to be independent if they belong to two different subgraphs/trees. Moreover, all nodes at the same level can be evaluated in parallel. Orphan nodes in the DAG, if they exist, denote columns which do not contribute to the factorisation process of other columns and thus can be included at any level of the DAMOS graph.

Definition 3. We define a DAMOS graph as a Direct Acyclic Graph (DAG) such that if column k depends on column i, then a directed edge exist from node i to node k (i.e. $i \rightarrow k$) where $i < k$.

Definition 4. We define the following type of nodes. A "leaf node" is a node that has no incoming edges. In contract, a "parent node" is a node that has incoming edges. if a parent node has no outgoing edges, it is then called a "a root node". An "orphan node" is a node that has no incoming or outgoing edges.

Definition 5. We define the DAMOS level of each node as the length of the longest critical path from any "leaf node" to the node itself. In our implementation of the DOMS algorithm, we use *Liao and Wong's* algorithm [184] to find the longest path.

Figure 4.19 illustrates, by the means of a DAMOS graph, the column dependencies that will arise arise during the LU factorisation of our example matrix A. The DAMOS graph was computed using the predicted nonzero structure of matrix U only. All the nodes at same DAMOS level can be computed independently. For instance, columns 1, 2, 3, 4, 5, 7 can be evaluated in parallel, however, column 9 cannot be processed until columns 4, 5, 6 are computed first. Column 10 is represented by an orphan node, which implies that it can be placed at any given DAMOS level. Generally speaking, the sparser the matrix is, the fewer dependencies there are, and hence the node count per level also increases. Thus, pre-ordering a matrix for sparsity can dramatically increase the

parallelism potential, as it will be empirically demonstrated in Section 4.4. Although our DAMOS algorithm efficiently derives a list of columns that can be evaluated in parallel within a given time-slot, it assumes that the same time is taken to compute each column. In reality, however, columns have different nonzero structures and thus the number of floating-point operations per column will also differ, ultimately impacting the column computation time. In Section 4.4, we will explore ways to distribute the computational efforts more evenly across the columns of a given matrix.

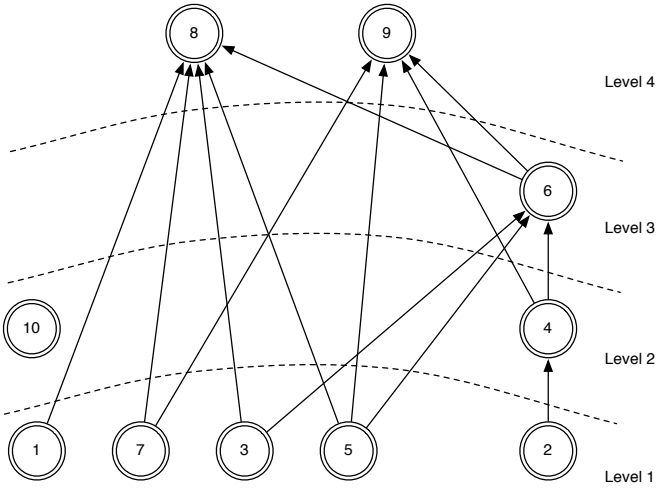


FIGURE 4.19: Unconstrained DAMOS Schedule Graph for Matrix A.

DAMOS level	Columns
Level 1	1, 2, 3 5, 7
Level 2	4, 10
Level 3	6
Level 4	8, 9

TABLE 4.1: Unconstrained DAMOS Schedule for Matrix A.

Assuming it takes roughly the same time to compute all the columns, the DAMOS schedule, shown in Table 4.1, is actually equivalent to the unconstrained As Soon As Possible (ASAP) schedule for the LU column operations [185]. The ASAP schedule unrealistically assumes that there will always be enough computational resources to concurrently process all columns within the same level. Therefore, in our DAMOS algorithm, we introduce a resource-constrained scheduling algorithm we refer to as *"modulo i scheduling"*, where i refers to maximum number of nodes that can reside within any

given DAMOS level. For instance, a *modulo 3* schedule assumes that there are only 3 computational units, each capable of independently processing a column, and thus it limits the number of nodes per DAMOS level to a maximum 3. Figure 4.20 and Table 4.3 define *"the modulo 3 schedule"* derived from the unconstrained DAMOS graph depicted in Figure 4.19. *"modulo i scheduling"* is particularly attractive if it is mapped to a pipelined FPGA architecture, where area is traded off for latency, such that it takes advantage of elongated schedule defined by Figure 4.21 and Table 4.3. In effect, LU factorisation can be computed using 2 computational units (i.e. less area) at the expense of increasing the DAMOS schedule by one level (i.e. increasing latency).

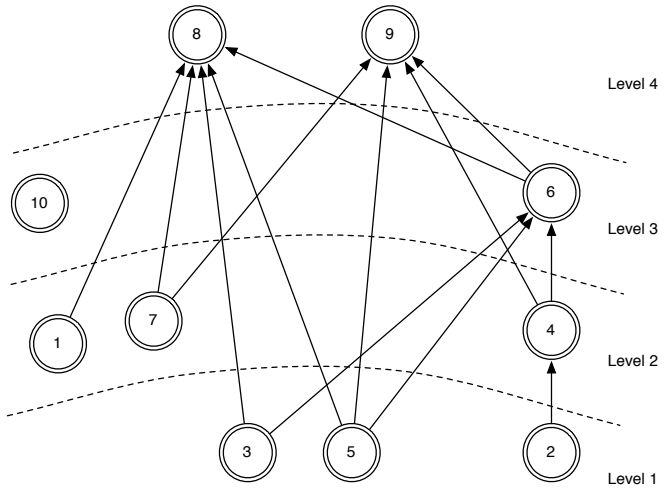


FIGURE 4.20: DAMOS Schedule Graph for Matrix A with *modulo 3*.

DAMOS level	Columns
Level 1	2, 3, 5
Level 2	1, 4, 7
Level 3	6, 10
Level 4	8, 9

TABLE 4.2: Modified DAMOS Schedule for Matrix A with *modulo 3*.

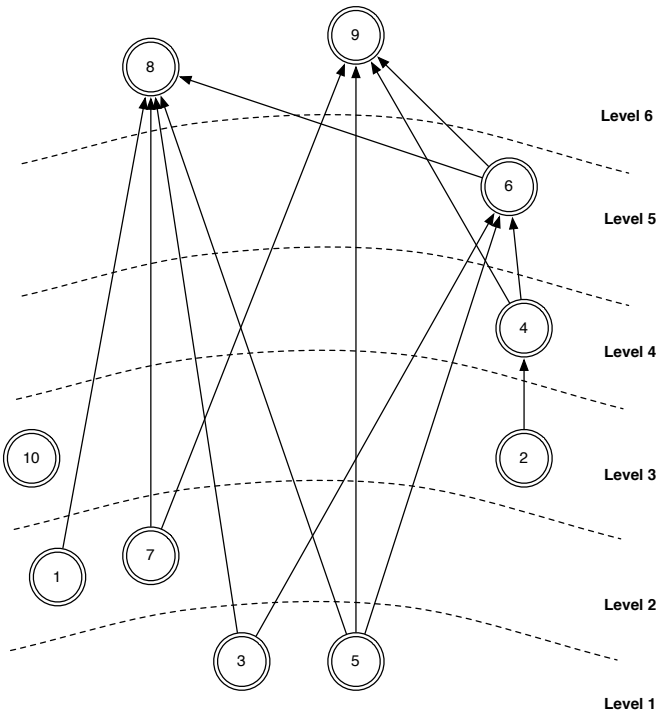


FIGURE 4.21: DAMOS Schedule Graph for Matrix A with *modulo 2*.

DAMOS level	Columns
Level 1	3, 5
Level 2	1, 7
Level 3	2, 10
Level 4	4
Level 5	6
Level 6	8, 9

TABLE 4.3: Modified DAMOS Schedule for Matrix A with *modulo 2*.

Figure 4.22 shows the overall generic DAMOS algorithm and the key constituents of its two main phases, namely, the symbolic factorisation and the scheduling phase. The algorithm was implemented using SuiteSparse Matrix [186], which is a suite of sparse matrix libraries. In our implementation, matrix pre-ordering is achieved by using the HSL MC64 routine [174], which ensures matrices are diagonally dominant, to eliminate the need of dynamic pivoting. We also employ the AMD reordering algorithm at the pre-processing stage to minimise fill-in since it offers the best results for circuit matrices, as it will be discussed in the Section 4.4.

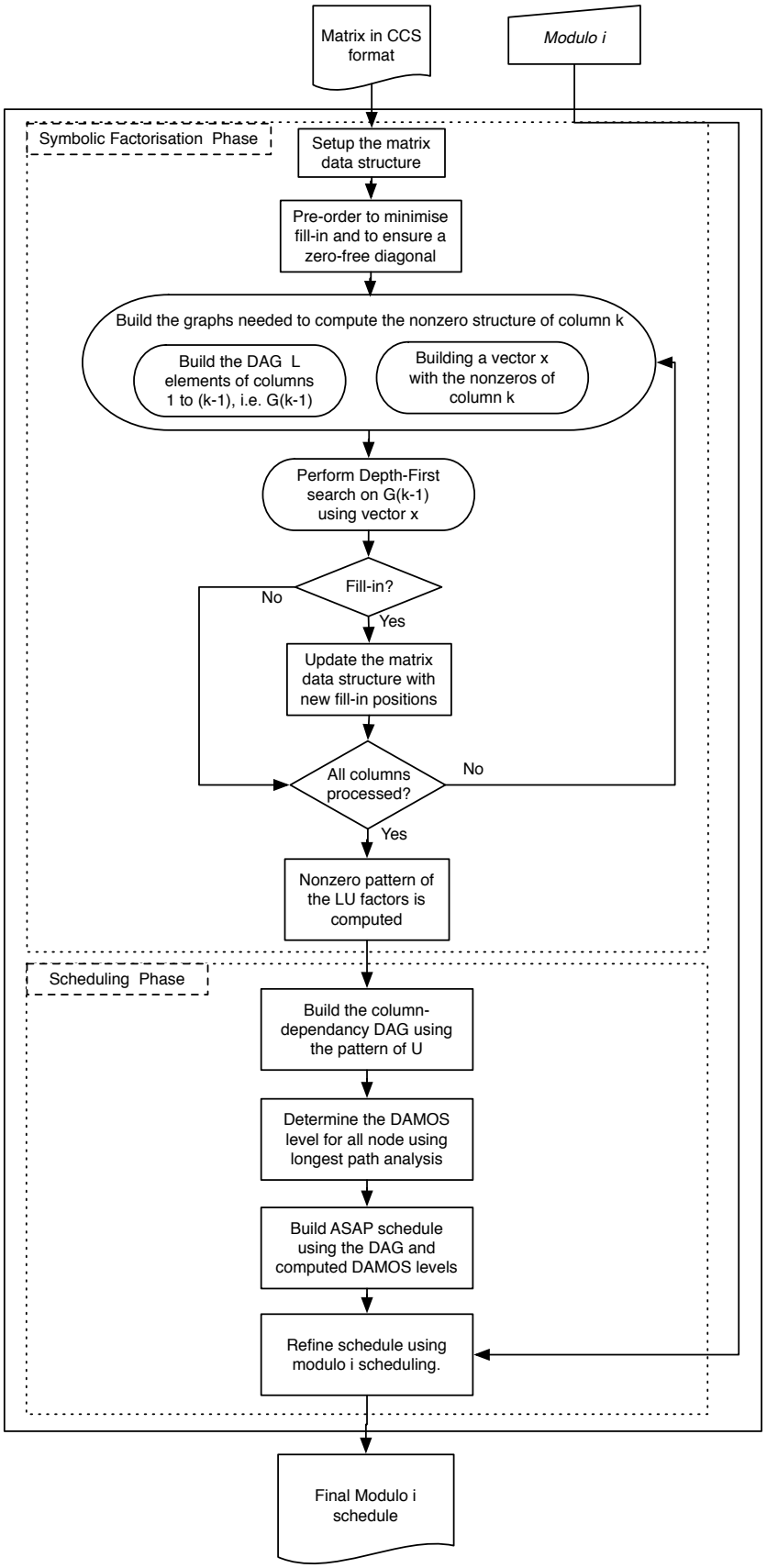


FIGURE 4.22: Overview of the Dependency-Aware Matrix Operations Scheduling (DAMOS) Algorithm.

Our DAMOS implementation was subsequently tested using a variety of circuit matrices from the University of Florida Matrix Repository [97]. Also, a number of moduli were applied to the same test matrices. The results of the tests are tabulated in Table 4.4. In our DAMOS implementation, a *modulo 1* input gives a schedule constrained to one computational unit. In other words, *modulo 1* effectively represents the schedule of the sequential LU factorisation algorithm. The sequential schedule has the same length as the number of matrix columns, as illustrated by the example schedule in Figure 4.23. Table 4.5 shows the predicted speedup that can be achieved for each test matrix, if **identical** computational units are used. We reiterate that the assumption here is that a computational unit is responsible for independently factorising a given column within a predetermined time-slot.

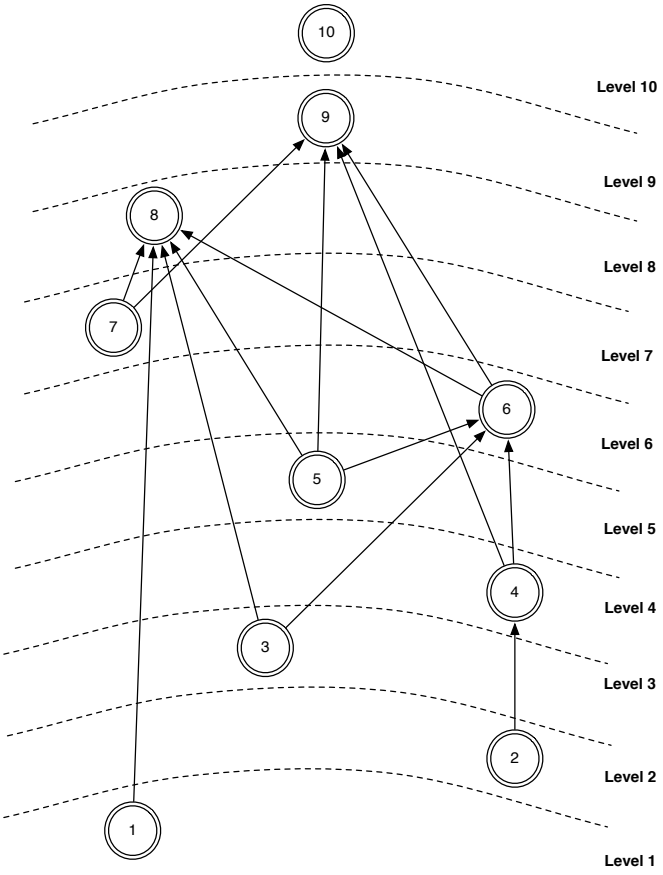


FIGURE 4.23: DAMOS Schedule Graph for Matrix A with *modulo 1*.

Matrix		Number of levels in the schedule				
Name	Size (n)	*Modulo 1*	*Modulo 2*	*Modulo 4*	*Modulo 6*	*Modulo 8*
rajat19	1157	1157	891	550	384	312
oscil_dcop_01	1813	1813	1089	706	491	407
fpga_dcop_01	1813	1813	1010	593	493	375
Hamm_add20	2395	2395	1497	923	628	511
bomhof1	2624	2624	1670	1083	788	596
Grund/meg1	2904	2904	2757	2652	2641	2511
bomhof2	4510	4510	2282	1504	1152	1095
Hamm/add32	4960	4960	3699	2016	1436	1312
Grund/meg4	5860	5860	3551	1810	1260	1136
rajat01	6833	6833	4180	2585	1753	1067
bomhof3	12127	12127	6866	3816	2884	2265
Hamm/memplus	17758	17758	12257	6970	5725	4475
bomhof4	80209	80209	40558	28060	24307	21741
rajat27	86916	86916	44870	23091	16981	13517

TABLE 4.4: DAMOS performance measurements with different moduli.

Matrix		Speedup ($\times$ compared to *Modulo 1*)			
Name	Size (n)	*Modulo 2*	*Modulo 4*	*Modulo 6*	*Modulo 8*
rajat19	1157	1.30	2.10	3.01	3.71
oscil_dcop_01	1813	1.66	2.57	3.70	4.46
fpga_dcop_01	1813	1.79	3.06	3.68	4.84
Hamm_add20	2395	1.60	2.60	3.81	4.69
bomhof1	2624	1.57	2.42	3.33	4.40
Grund/meg1	2904	1.05	1.09	1.10	1.16
bomhof2	4510	1.98	3.00	3.91	4.12
Hamm/add32	4960	1.34	2.46	3.45	3.78
Grund/meg4	5860	1.65	3.24	4.65	5.16
rajat01	6833	1.63	2.64	3.90	6.40
bomhof3	12127	1.77	3.18	4.21	5.35
Hamm/memplus	17758	1.45	2.55	3.10	3.97
bomhof4	80209	1.98	2.86	3.30	3.69
rajat27	86916	1.94	3.76	5.12	6.43
Average		1.62	2.68	3.59	4.44

TABLE 4.5: Predicted acceleration using DAMOS with different moduli

4.4 Empirical Analysis of LU Decomposition

In order to design (an) application specific hardware that capitalises on the features of the Gilbert-Peierls factorisation algorithm while optimally harnessing the parallelism exposed by our DAMOS scheduling algorithm, empirical analysis is necessary. In effect, sparse matrices in many domains, including SPICE simulations, do not share identical nonzero patterns. Moreover, the computation pattern during sparse LU decomposition is dependent on the nonzero structure of the matrix, which is in turn is dependent on the pre-orderings used. As such, empirical testing was conducted to identify what ordering techniques and algorithms can be used to reduce the computational effort needed to factorise circuit matrices. We also attempt to identify a pre-ordering strategy that spreads the computational effort more uniformly across the columns of the matrix at hand. This is particularly advantageous when used in conjunction with our DAMOS scheduling algorithm, which assumes that the same effort is needed to evaluate different columns. A summary of the key features of the benchmark circuit simulation matrices used in our test is provided in Table 4.6.

Matrix Name	Matrix Order	NNZ Count	Zeros (%)	Pattern Symmetry	Numeric Symmetry
fpga_dcop_01	1813	5892	99.82%	65%	1.6%
bomhof1	2624	35823	99.47%	100%	21 %
bomhof2	4510	21199	99.89%	81%	41 %
bomhof3	12127	48137	99.96%	77%	30 %
bomhof4	80209	307604	99.99%	83%	36 %
rajat19	1157	3699	99.72%	91%	92%
rajat01	6833	43520	99.99%	99%	99%
rajat20	86916	604299	99.99%	99%	11%

TABLE 4.6: A selection of test matrices from the UFMC repository [97]

In our tests, we preorder our benchmark matrices using a variety of fill-in minimising heuristic. We then perform sparse LU decomposition function using MATLAB's built-in (i.e. `[L,U,P] = lu(A,thresh)`). By default, MALTLAB's LU function employs

the Gilbert-Peierls' algorithm to perform a left-looking sparse LU decomposition with pivoting. However, pivoting can be restricted to diagonal elements using the `thresh` input. The latter is a two-element vector that defaults to [0.1, 0.001]. In effect, for matrices with a mostly symmetric structure and mostly nonzero diagonal, MATLAB ensures that the diagonal elements meet the following criterion:

```
A(i,j) ≥ thresh(2) * max(abs(A(j:m,j)))
```
(4.23)

If a diagonal entry fails this test, MATLAB then selects a pivot entry from below the diagonal, using `thresh(1)` instead of `thresh(2)`:

```
A(i,j) ≥ thresh(1) * max(abs(A(j:m,j)))
```
(4.24)

For all other type matrices (e.g. asymmetric pattern matrices) MATLAB only performs the inequality test of Equation 4.24. Therefore, in order to restrict pivoting to diagonal elements, we set both values of the `threshold` vector to artificially low values. Addionally, we use the HSL MC64 subroutine [174] to pre-condition our matrices. The HSL MC64 subroutine ensures that matrices are diagonally dominant by computing a matrix permutation that maximises the sum of the diagonal entires, effectively eliminating the need for dynamic pivoting [159]. Moreover, it ensures that there are no zero values on the diagonal, as can be seen in Figure 4.24. Once pivoting is restricted to the diagonal, the number of nonzeros in the LU factors, generated by MATLAB, will be identical to the results of the symbolic analysis conducted by the DAMOS algorithm, since it also does not consider pivoting during LU decomposition.

As previously mentioned, matrix ordering heuristics alter the nonzero structure of a sparse matrix with the aim to reduce the number of fill-in elements that may arise during the course of a matrix factorisation. This also has the effect of reducing the number of computations required and the amount of data storage necessary to perform the sparse LU decomposition. Therefore, we study the effect of different ordering techniques on the LU decomposition of circuit matrices. Various minimum degree orderings such as AMD and COLAMD function; and the Nested Dissection ND routine from METIS [169], were

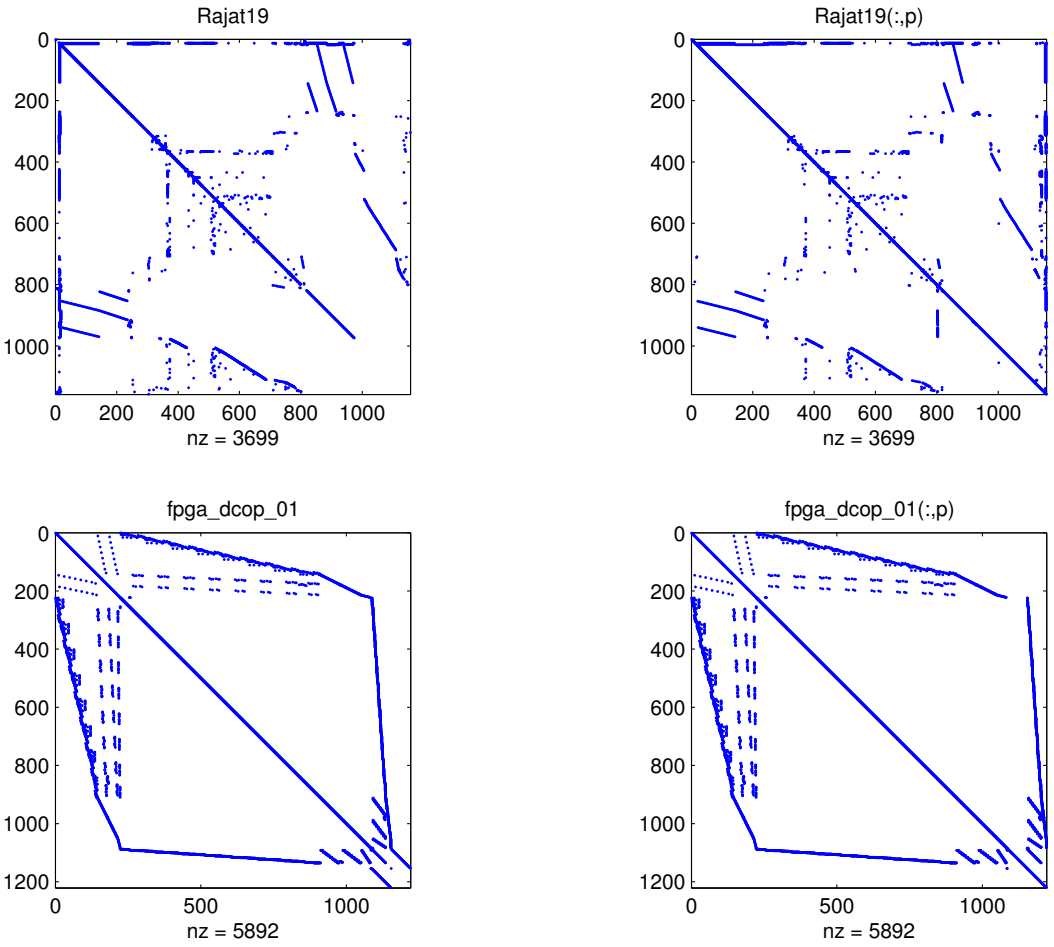


FIGURE 4.24: Zero-free Diagonal Circuit Matrices using a Maximum Traversal Permutation

used to order the test matrices prior to LU decomposition. These orderings were applied symmetrically (i.e. rows and columns) to the matrices in order to preserve the zero-free diagonal. A quantitative comparison of these orderings' performances is summarised in Table 4.7. The latter reports the number of nonzeros in the lower and upper triangular factors $(L + U)$ after various orderings have applied symmetrically to the benchmark sparse matrices. The results indicate that the AMD ordering algorithm produces the best results on circuit matrices. This result is consistent with a previous study [105] that reported that the minimum degree-based ordering methods provide the best orderings for sparse LU decomposition of circuit simulation matrices. Moreover, AMD assumes no

numerical pivoting and therefore is suitable for a static pivoting strategy. As such, in our subsequent experiments and for designing the hardware prototype, the AMD ordering method was used. Figure 4.25 illustrates the nonzero structure of the "fpga_dcop_01" matrix with different orderings, prior to LU decomposition. Figure 4.26 shows the resulting nonzero structure of the same matrix after LU decomposition applied on the different ordering permutations.

TABLE 4.7: Impact of different ordering heuristics on the number of nonzeros in the LU of some selected circuit matrices

Input Matrix		Number of nonzeros in LU factors			
Name	Initial NNZ	No ordering	AMD	COLAMD	METIS
oscil_dcop_01	1544	11540	2320	2931	2484
fpga_dcop_01	5892	55433	7697	10579	7367
bomhof1	35832	41353	43443	773096	42598
bomhof2	21199	124674	37088	164341	43585
bomhof3	48137	150206	72853	110589	90738
bomhof4	307604	468882	422532	8028555	442633
rajat19	3699	6128	3974	4546	4939
rajat01	43250	1340727	49597	152543	67286
rajat27	97353	21599231	144214	1051066	8207645

The number of nonzeros in the lower and upper triangular factors impacts the minimum memory size required to store the results of the factorisation. The amount of storage required is proportional to the number of bits used to store the indices and values in the matrix. For example the number of nonzeros in the $L+U$ factors of the largest test matrix, i.e. bomhof4, using the AMD ordering applied symmetrically is 422,532. For each nonzero there will be an entry in the matrix storage. The matrix representation contains an index and a floating point value. Using 32 bits to represent the indices and single precision 32-bit values for the matrix nonzero entries requires 422,532 $\times$ 64 bits ($\approx$ 26 Mbits) of data storage minimum to complete the sparse LU decomposition. This amount of data exceeds the embedded memory resources available on today's FPGAs (e.g. $\approx$ 6.5 Mbits of block and distributed RAM on a Viretx 5 LX110T). As such, an

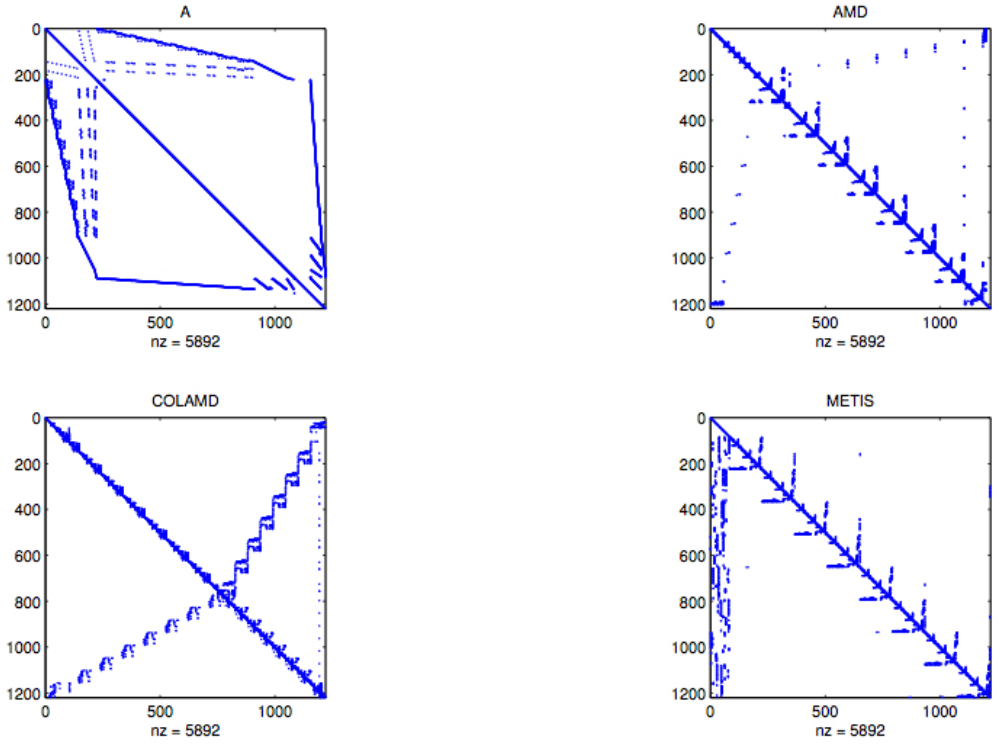


FIGURE 4.25: Nonzero structure of "fpga_dcop_01" prior to LU decomposition

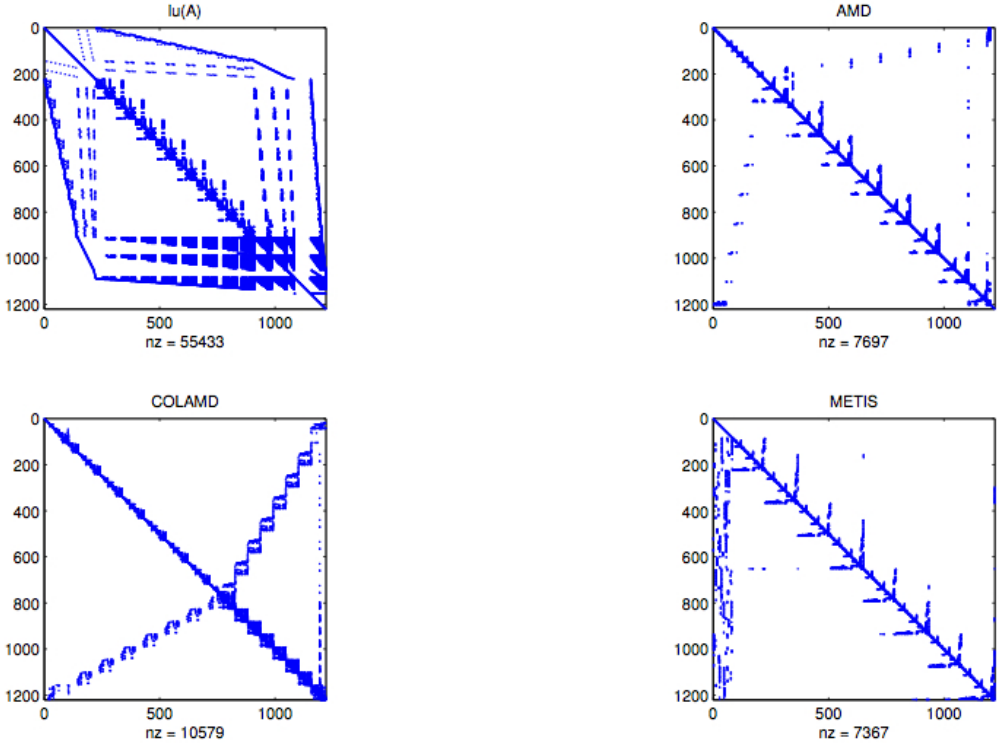


FIGURE 4.26: Nonzero structure of "fpga_dcop_01" after LU decomposition

external storage device with a high data density is required for matrices that have more than 85,000 nonzero elements if a Viretx 5 LX110T is used. The embedded memory block can be then used to buffer portions of the matrix to be factored and hence hide the latency associated with the external memory transfers.

Generally speaking, the number of floating-point operations required for sparse left-looking LU decomposition with no pivoting is proportional to the number of operations required to multiply the resulting factors (i.e. L and U), as it was demonstrated in Section 4.2. In the normalisation step, a floating-point division is required for every element below the diagonal in the pivot column. In the sparse triangular solution step, a floating point multiply-subtract operation is required for every element in the pivot column and all the elements in the update columns involved in the *"sparse triangular solution"*. The update columns are defined as the children columns of the current pivot column in DAMOS graph, as shown in Figure 4.1. Table 4.8 summarises the number of FLoating-point OPerations (FLOPs) performed during the sparse LU decomposition of benchmark matrices.

This number of floating-point operations was acquired via profiling a purposely written MATLAB script that performs left-looking LU decomposition with no pivoting. All the input matrices were initially permuted using maximum traversal to ensure a zero-free diagonal, and then ordered using the AMD algorithm. The script also accounts for numerical cancellations that may occur during the factorisation process. These cancellations, even though very rare, lead the appearance of zeros on the diagonal and ultimately halt the factorisation algorithm during the normalisation phase (i.e. division over zero). As can be seen from Table 4.8, the number of floating point operations required to update the pivot columns (i.e. sparse triangular solution step) clearly dominates the total number of floating point operations, that is on average 90% of the total FLOPs required to compute the LU factors. Therefore, in order to accelerate the overall Gilbert-Peierls algorithm, the sparse triangular solution has to be parallelised efficiently in accordance with Amdahl's Law.

TABLE 4.8: Floating-point operations count of Gilbert-Peierls LU Decomposition of some selected circuit Matrices

FLOP / Matrix	Division		Add		Multiply		Multiply-Add		Total
	Count	(%)*	Count	(%*	Count	(%)*	Count	(%)*	FLOPs
oscil_dcop_01	960	17	2419	42	5798	42	4838	83	5798
fpga_dcop_01	2632	13	8560	43	19752	43	17120	87	19752
bomhof1	20381	2	444043	49	908467	49	888086	98	908467
bomhof2	14688	2	295102	49	604892	49	590204	98	604892
bomhof3	28817	9	144475	45	317767	45	288950	91	317767
rajat01	21702	9	110795	46	243292	46	221590	91	243292
rajat19	1206	22	2086	39	5378	39	4172	78	5378
rajat27	61222	6	471579	47	1004380	47	943158	94	1004380
	Avg**	10	Avg**	45	Avg**	45	Avg**	90	

* % of Total FLOPs ** Arithmetic Average

To put the FLOP count figures into context, we need to refer back to the assumption we made earlier as part of developing our DAMOS scheduling algorithm. In effect, the DAMOS algorithm assumes it takes roughly the same time to evaluate independent columns. In practice, however, columns have different nonzero structures and thus the number of floating-point operations per column will also differ, ultimately impacting the column computation time. So far, we have empirically established that the AMD ordering heuristic offers the best results in terms of efficiently reducing fill-in for circuit matrices, which in turn reduces the number of FLOPs required. Nonetheless, this finding lends itself to the following question: what does the distribution of the FLOPs required over the columns of the matrix looks like?

In order to answer this question, we empirically collected the FLOP count required to factor each column of our benchmark matrices, before and after the AMD ordering is applied. Figure 4.27 to Figure 4.30 plot the FLOP count per column associated with the Gilbert-Peierls factorisation for the following matrices: fpga_dcop_01, oscil_dcop_01, Bomhof2, and Rajat19, before and after the AMD algorithm is applied. We can see that using the AMD ordering not only reduces the number of fill-in elements but also results in much sparser LU factors, and thus produces a more balanced workload across the columns. This is particularly attractive in a distributed computing architecture, where

the columns are spread over many processing elements. Furthermore, a lower FLOP count per column reduces the amount of resources required to compute a given column in parallel. For instance, in Figure 4.28, the highest column FLOP count recorded prior to ordering was just under 250,000 floating-point operations and then decreased to under 300 floating-point operations after the AMD ordering was applied. It is clear now that pre-ordering matrices for sparsity not only reduces the overall FLOP count but also distributes the computational efforts more evenly between columns of a given matrix. This, in turn, increases the degree of the parallelism that can be exploited using specialised algorithms, such as DAMOS.

4.5 Summary

In this chapter, we have demonstrated how we can leverage the graph representation of the original matrix to predict the nonzero structure of resulting LU factors. We have also shown how the Gilbert-Peirels (G/P) symbolic analysis, in conjunction with predicted nonzero pattern, can be used to create a column-dependency driven task graph that maximises the parallelism potential for the LU matrix factorisation. As such, we have introduced our Dependency-Aware Matrix Operations Scheduling (DAMOS) preprocessing stage, which we employ to generate a parallel operations schedule. The latter can be then used to parallelise and control the dataflow of G/P LU matrix operations on the FPGA, as will be illustrated in the next chapter. Our DAMOS algorithm assumes it takes roughly the same time to evaluate independent columns. In practice, however, columns have different nonzero structures and thus the number of floating-point operations per column will also differ, ultimately impacting the column computation time. Nonetheless, our empirical testing showed that pre-ordering matrices for sparsity not only reduces the overall FLOP count but also distributes the computational effort more evenly between columns of a given matrix.

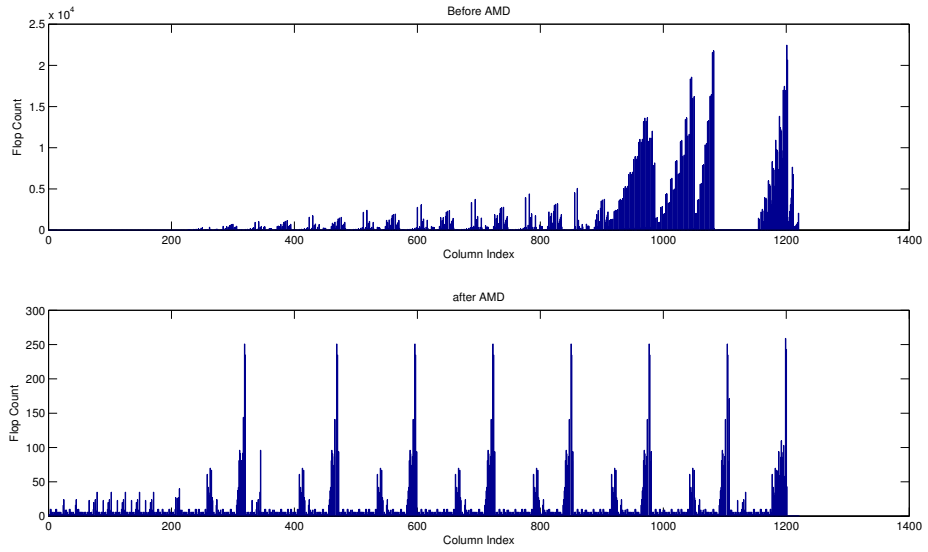


FIGURE 4.27: The Effect of Matrix Ordering on the Column Flop Count of LU Decomposition of the "fpga_dcop_01" matrix

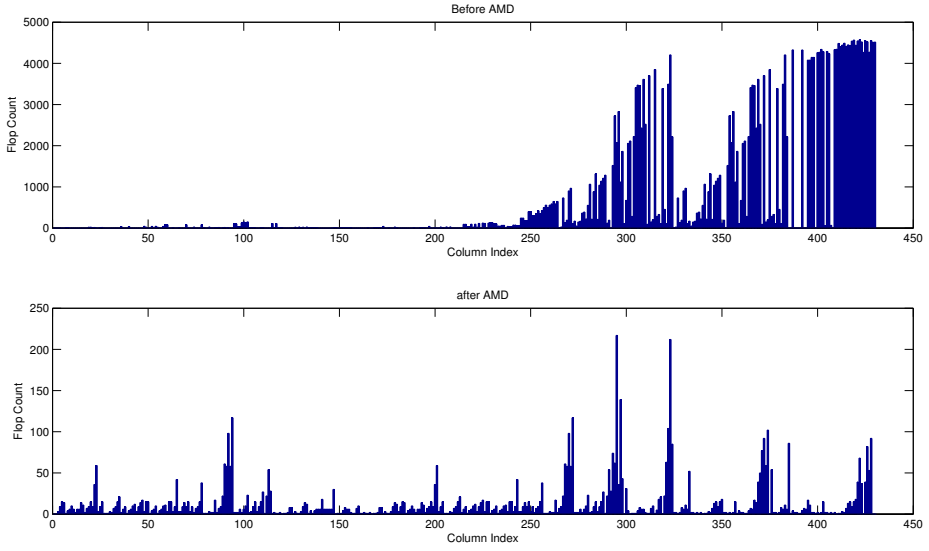


FIGURE 4.28: The Effect of Matrix Ordering on the Column Flop Count of LU Decomposition of the "oscil_dcop_01" matrix

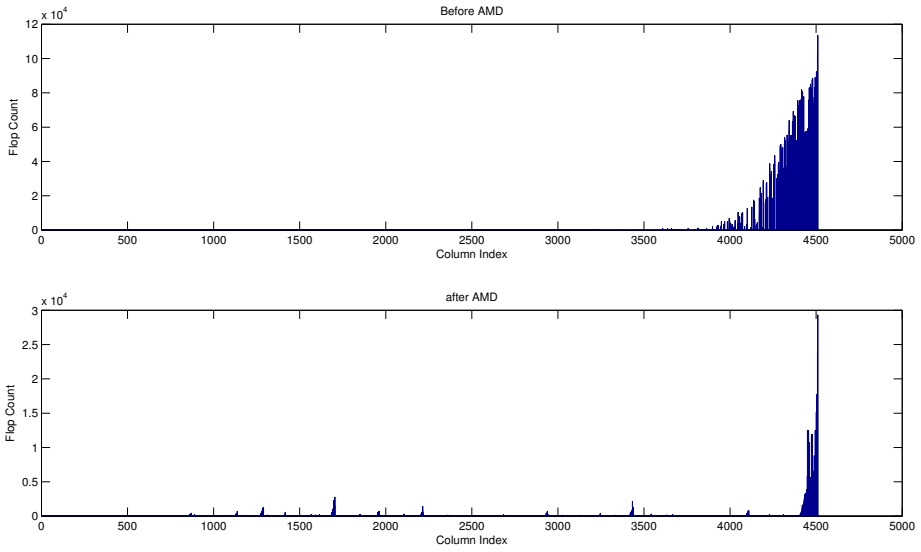


FIGURE 4.29: The Effect of Matrix Ordering on the Column Flop Count of LU Decomposition of Bomhof2

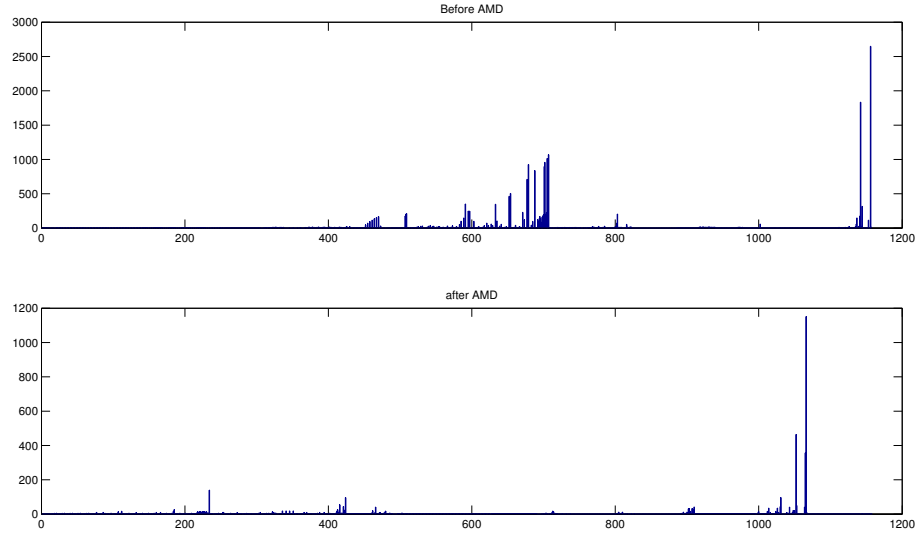


FIGURE 4.30: The Effect of Matrix Ordering on the Column Flop Count of LU Decomposition of Rajat19

Chapter 5

Single-FPGA Matrix Solution

In Chapter 3, we have empirically shown that the speed of the linear solution phase becomes crucial in large-scale circuit simulations. We have also established this phase is more challenging to parallelise than the Model Evaluation phase, due to the seemingly inherent data-dependencies that exist within the course of the LU factorisation of a matrix. Nonetheless, in Chapter 4 (Section 4.3), we have introduced our DAMOS scheduling algorithm, which is able to leverage the sparsity of the matrix at hand to identify columns that can be evaluated in parallel (i.e. medium-grained parallelism). In this chapter, we demonstrate how to create an FPGA design that is able to harness the medium-grained parallelism exposed by DAMOS, without neglecting the finer-grained parallelism presented within the operations relating to column updates (i.e. sparse triangular solution, and column normalisation).

5.1 FPGA Design Objective

The main objective in designing today's high performance sparse LU decomposition software is to employ algorithms that maximise the number of Basic Linear Algebra Subprogram(s) (BLAS) operations performed, while minimising the number of scalar computations required [187, 188]. This is mainly due to the fact that modern microprocessors rely on fast integrated multi-level caches as well as complex memory hierarchies to keep the computation pipeline optimally utilised, and hence sustain high throughputs. Therefore, BLAS operations are tailored to enhance cache data locality and to provide a sustained stream of arithmetic operations.

However, LU factorisation on highly sparse circuit matrices fails to effectively exploit data locality and regularity, resulting in frequent cache misses and thus degrading the overall performance [189]. Moreover, Sparse LU algorithms, such as supernodal methods, attempt to group rows or columns with similar nonzero pattern into *"supernodes"* [190], on which BLAS operation can be performed. However, circuit matrices typically do not have large supernodes since the interconnection among nodes is not similar across all the nodes in the circuit. The overall performance of BLAS is further degraded by the growing discrepancy between the CPU speed and the memory latency [191].

The inability of modern sparse matrix LU solvers to maintain a high utilisation of the processor's floating-point units, suggests that designing a more efficient application specific hardware may lead to a significant improvement in performance. In effect, rather than adapting the problem to the general purpose hardware, the design of a hardware that specifically capitalises on the features of sparse LU decomposition is proposed as an alternative solution. As such, we aim to use the column-level dependencies, exposed by our DAMOS algorithm, to generate a dataflow and an operations' schedule that maximises the busy time of a multiple-PE architecture on an FPGA. In this distributed architecture, independent columns can be mapped to different PEs and thus minimising the communication overhead. Column-level updates can also take advantage of pipelined floating-point operations to achieve a higher throughput.

5.2 Parallel Sparse LU FPGA Architecture

In Section 4.3, we demonstrated that that the seemingly sequential flow of the Gilbert-Peierls LU factorisation algorithm can be effectively parallelised by explicitly exposing column-level concurrency, by the means of a DAMOS scheduling graph. This graph only depends on the nonzero structure of the circuit matrix. The nonzero pattern of a circuit matrix reflects the couplings and the connections that exist in the underlying circuit, which does not change during the course of a SPICE simulation. This means that the matrix to be solved retains the same nonzero pattern over the SPICE transient iterations, and it only undergoes changes in numerical values. Hence, the symbolic analysis cost is justifiable and can be easily amortised over a number of iterations. Therefore, the column-level dependency graph can be cheaply computed offline (see Section 5.5.1) before the actual numerical factorisation takes place on the FPGA accelerator.

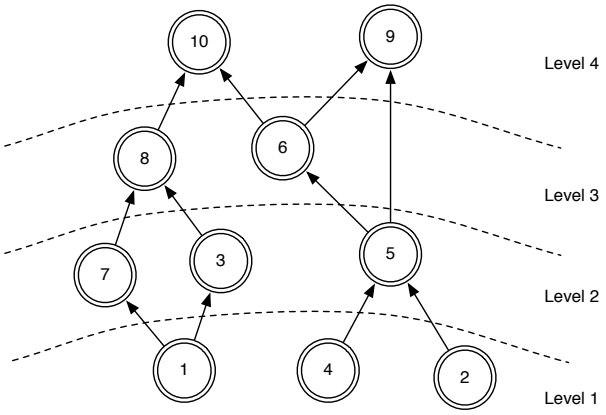


FIGURE 5.1: Example DAMOS Scheduling Graph with *modulo 3*.

The column-level dependency graph can be then loaded onto the FPGA and used to dictate a parallel execution flow of LU column operations. However, it may not be possible to fit the entire graph for a large matrix onto the FPGA, in which case, the column-dependency information can be also used to pre-compute a column loading order. The latter can be then used to dynamically load columns to the FPGA such that computations and memory loads are overlapped, effectively hiding the latency associated with the external memory interface. To illustrate this concept, consider the DAMOS graph shown in Figure 5.1 as an example. For instance, columns 1, 2, 3, 4, 5, 7 can be

loaded to the FPGA first. In the second stage, columns 6, 8 can be loaded in lieu of column 1, 2, 4 while columns 3, 5, 7 are being normalised. In last stage, columns 9, 10 are loaded to replace columns 3, 7 while columns 6, 8 are being normalised.

5.2.1 Resolving Dataflow Dependencies

So far, we have established that Gilbert-Peierls sequential column factorisation process can be altered to expose column-level parallelism. Despite this exposed column-evaluation concurrency, dataflow dependencies may still exist within column-level updates themselves. In order to illustrate this, consider Figure 5.3, in which we show all the dataflow dependencies and operations needed to computed the LU factorisation of the example matrix A, depicted in Figure 5.2, according to its unconstrained DAMOS Schedule. We note two types of dataflow dependencies: inter-column and intra-column data dependencies.

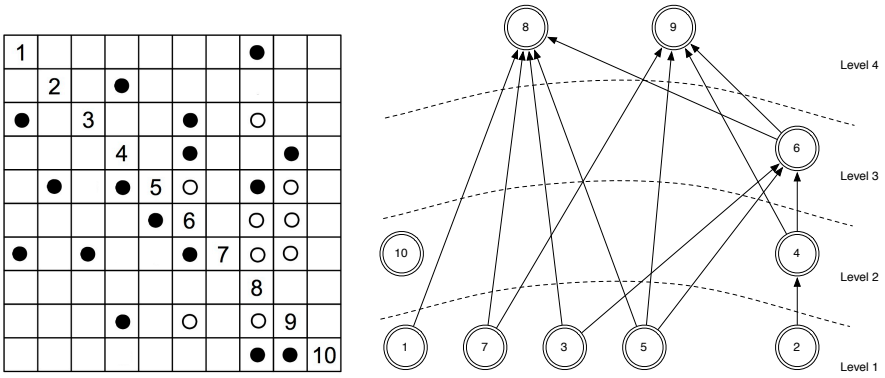


FIGURE 5.2: Example of a Matrix A and it is corresponding DAMOS Scheduling Graph.

The inter-column data dependencies represent the inherent column-level dependencies that exist in the Gilbert-Peierls algorithm. This type of dependency can be naturally resolved by simply following the execution order determined by the corresponding DAMOS schedule, factorising columns in level 1 first, then columns in level 2, and so forth. The intra-column dependencies relate to the order at which the current column element updates, in the sparse triangular solution, should be calculated. Nevertheless,

in Section 4.2.1.1, we have established that Gilbert-Peierls' symbolic analysis of a particular column effectively computes a topological order that maintains the precedence relationship in the numerical factorisation step. In effect, this computed topological order can be used to sustain a dataflow stream to the pipelined floating-point operations on the FPGA. Studying the dataflow graph more closely, we can also see that division operations associated with the column normalisation stage (e.g. columns 1, 2, 3, and 5) can be performed concurrently, creating another source of parallelism that can be exploited at the hardware level.

5.2.2 Design Flow

Our work implements the Gilbert-Peierls LU factorisation (i.e. algorithm shown in Figure 4.11), in conjunction with the static pivoting algorithm introduced by *Li and Demmel* in [159], which they showed to be as accurate as partial pivoting algorithms for a number of problems including circuit simulations. The main advantage of static pivoting is that it permits a priori optimisation of static data structures and the communication pattern, effectively decoupling symbolic and numerical factorisations steps. This makes sparse LU factorisation more scalable on a distributed memory architecture. The overall algorithm implemented can be summarised as follows:

1 First, we find diagonal matrices D_r, D_c and a row permutation P_r such that $P_r D_r A D_c$ is more diagonally dominant to decrease the probability of encountering small pivots during the LU factorisation. To achieve this, we use the HSL MC64 routine [174] with option 4. The latter computes a permutation of the matrix so that the sum of the diagonal entries of the permuted matrix is maximised.

2 We find a permutation P_c such that the resulting matrix in step (1) incurs less fill-in in the course of the LU factorisation. We can use many heuristics such as nested dissection or minimum degree on the graph of $A + A^T$ or AA^T. However, we shall use the approximate minimum degree (AMD) as it produces the best results for circuit matrices, as we have empirically shown in Section 4.4. In order to preserve the diagonal computed in step (1), any ordering used should be applied symmetrically.

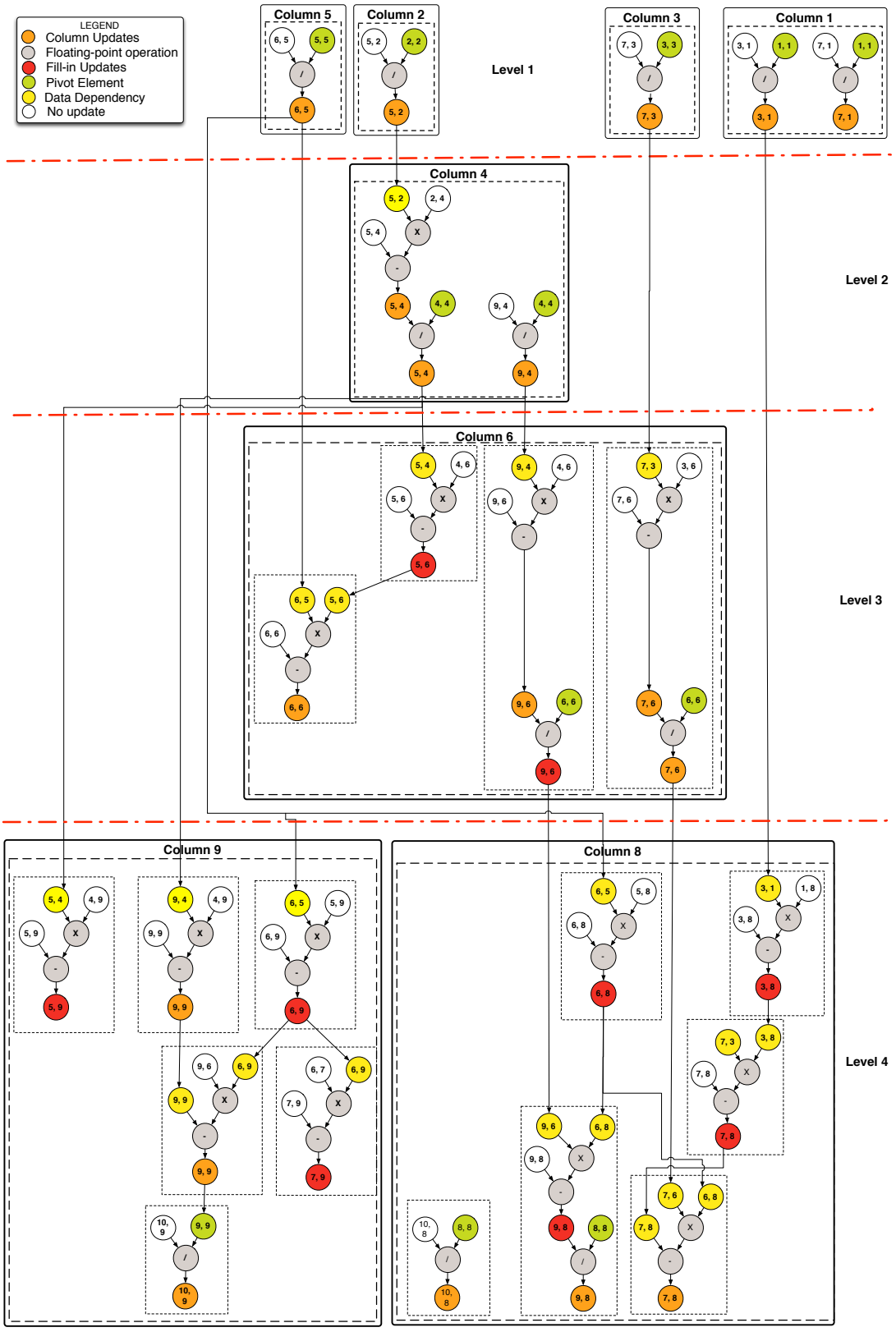


FIGURE 5.3: Dataflow of a Gilbert-Peierls LU factorisation

3 We perform symbolic analysis to identify the locations of the nonzero entries of L and U. In this step, we also compute task-flow graph by performing the LU decomposition symbolically, i.e. only using the resulting structure.

4 In this step, we perform left-looking LU factorisation on the FPGA and replace any tiny pivots (i.e $|a_{ii}| < \sqrt{\varepsilon}.\|A\|$) by $\sqrt{\varepsilon}.\|A\|$, where ε is machine precision (e.g. 2^{-24}, 2^{-53} for single and double precision IEEE 754 formats respectively), and $\|A\|$ is the matrix norm. This is acceptable in practical terms as the SPICE linear equation solution is used as part of Newton-Raphsons method, and an occasional small error during the iterative process does not affect the integrity of the final solution [159]. We calculate the matrix norm at the symbolic factorisation phase, using the SuiteSparse API [186]. The use of the HSL MC64 routine in step (1) decreases the likelihood of encountering tiny pivots. Furthermore, selecting the diagonal as the pivot entry ensures the fill-reducing ordering from the symbolic phase is maintained.

Step 1 to step 3 form the *"matrix preconditioning phase"*, and they are conducted as part of our DAMOS Scheduling Algorithm implementation, as detailed in Chapter 4 (Section 4.3). DAMOS takes a sparse matrix as input, applies the AMD ordering, and then symbolically generates the column-level dependencies as well as the nonzero pattern of the LU factors. For step 4, we implement the parallelised version of the Gilbert-Peierls factorisation algorithm on the FPGA, using a multi-PE distributed architecture. Since we do not consider dynamic pivoting in our design, all possible fill-ins as well as column and dataflow dependencies are determined at the matrix preconditioning phase.

5.2.3 Top Level Design

Our parallel FPGA architecture features multiple PEs interconnected by a switch network. Figure 5.4 shows the top level diagram of the our sparse LU hardware implementation. Essentially, our design consists of a controller connected to n PEs. In each PE, there is a multiplier, a subtractor, a divider, and a local Block Random Access Memory (BRAM) with a reconfigurable datapath. An approximate schematic for a processing

element is shown in Figure 5.6. The maximum number of PEs, and their local memory size are limited by the available resources of the FPGA. We use the information gathered from symbolic analysis to instantiate PEs accordingly. The PEs are interconnected by high speed switches to minimise the communication overhead while increasing concurrency.

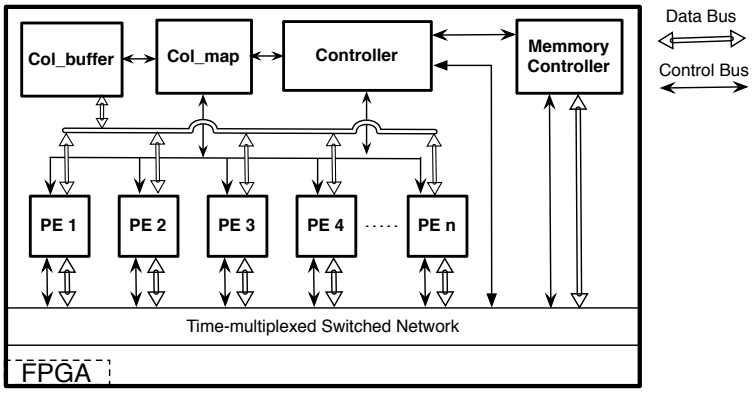


FIGURE 5.4: Top Level Design for the LU Decomposition FPGA Hardware

The controller implements a four stage pipeline, as shown in Figure 5.5. Stage 1 consists in loading the matrix data from the off-chip DRAM to the PEs on-chip BRAM. The PEs' local BRAMs can be also preloaded with matrix data at the FPGA programming phase such that the matrix data is included in the "bitstream". Stage 2 performs a triangular sparse solve on the current column of A to compute the current columns of L and U. Stage 3 normalises the component of L with the diagonal entry. Stages 2 and 3 are executed iteratively until all columns are evaluated. At any given time, PEs collectively perform either the sparse triangular solve or the column normalisation.

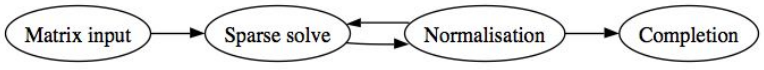


FIGURE 5.5: State machine for the proposed LU decomposition hardware

In the sparse solve phase, the "Col_map" unit first performs a burst read across all PEs to form a column-wise representation of the pivot column and saves it to the column buffer. Then, elements of the column buffer are broadcast to the PEs one at a time to perform the bulk computation of the sparse triangular solution (i.e. line 9 in Algorithm

4.11). Figure 5.6 depicts an approximate datapath of the PE during the column sparse solve phase (i.e line 9 of algorithm in Figure 4.12).

In the normalisation phase, the controller fetches the pivot entry from its corresponding PE and broadcasts it to all PEs to perform all the divisions in parallel. To fill the deep pipelines of our floating-point units, the controller uses the column-dependency graph as a task flow-graph. Data are streamed from the memory, through the arithmetic units for computation, and stored back to the memory in each stage.

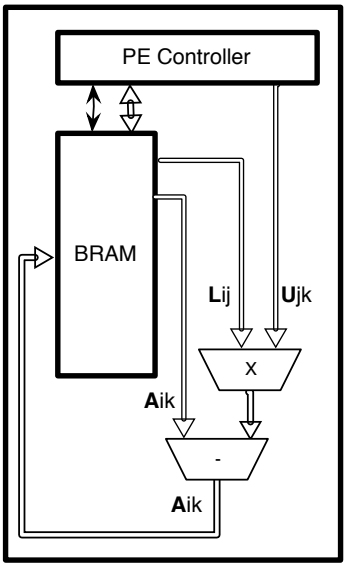


FIGURE 5.6: PE at the sparse triangular solution phase

The controller's main objective is to maintain optimal usage of the computation pipeline optimally utilised while following a deterministic task execution flow. Figure 5.7 shows the main constituents of the "Controller". The Control unit implements the state machine described in Section 5.2.3. The Status Logic Unit registers the different status signals from other functional units, monitors their functionality, and generates state triggering signals for the Control Unit. The "Address Map" unit stores the column dependency information computed in the symbolic analysis. The "Address Map" can be either initially preloaded when the FPGA is programmed or can be re-initialised at the "Matrix Input" stage via the Memory Controller.

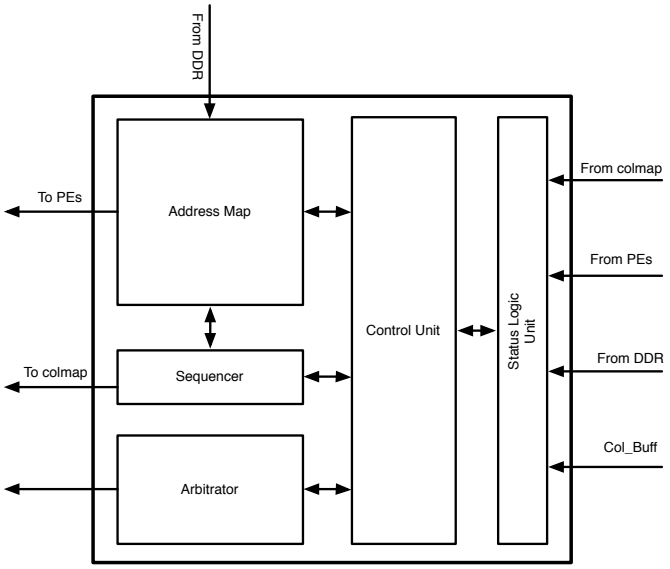


FIGURE 5.7: High-level schematic of LU hardware controller

The "Sequencer" utilises the column dependency information, stored in the "Address Map", to implement a look-up table that generates the correct memory addresses for the column indices to be processed. The "Sequencer" unit then broadcasts the addresses generated to their respective PEs, while maintaining a record of the columns processed. The "Arbitrator" unit maintains the interconnection between the PEs, the "Col_Buffer", and the "Control" unit. In other words, it acts as a datapath controller for the the reconfigurable interconnect linking the different functional units of the LU decomposition hardware.

At the start of "the triangular sparse solution" stage, the "Control" unit instructs the "Sequencer" to fetch the addresses of all the elements involved in computing the current column. The Sequencer, in turn, instructs the "Col_map" unit to read the current column into the "Col_buffer". Meanwhile, the "Sequencer" also instructs the "Address Map" to generate the addresses for the update columns elements associated with the current column (i.e. the column being read into the column buffer). Next, the Arbitrator maintains a stream connection between the "Col_buffer" and the PEs, broadcasting every element of the current column, stored in the buffer, one at a time to all PEs. Once the column buffer is drained, the "Control" unit is notified, prompting it

to move to the next stage, i.e. the normalisation stage, provided that all the the states triggering signals, from the "Status Logic" unit, allow it to do so.

In the "column normalisation" stage, the "Control" unit instructs the "Sequencer" to perform the column normalisation. As such the "Sequencer", via the "Address Map", generates the addresses for the elements below the current pivot and sends them to their respective PEs. At the same time, the column buffer broadcasts to all PEs so that all divisions can proceed in parallel. Once the column divisions are performed, the "Control" unit promoting to move to process the next column. The "Sequencer" acts as program counter keeping track of the columns that have been processed. The "Control" unit alternates between "the triangular sparse solution" and "column normalisation" stages until the "Sequencer" has processed all the columns.

5.3 Experimental Setup

In this section, we explain the experimental setup used to build and test our LU decomposition FPGA hardware prototype.

5.3.1 FPGA Implementation

To implement a prototype for our design, we target the Xilinx XUPV5-LX110T development board (Appendix B), which features a Virtex 5 LX110T FPGA. As mentioned in Section 5.2.3, the controller of our design utilises the column-dependency graph of a matrix as a task flow-graph to stream data from the memory, through the arithmetic units for computation, and stores the results back to the memory in each stage. As such, the relative placement between the memory blocks and the computational blocks is important and can significantly impact performance. The targeted Virtex-5 FPGA benefits from the physical proximity of these blocks as they are arranged close to each other in special lanes within the fabric (i.e. BRAM and DSP48 blocks).

Therefore, in our implementation, we use the floating-point subtract, multiply/divide (DSP48 blocks), and compare units from the Xilinx Floating-Point library. The latter

TABLE 5.1: Sparse LU Hardware Prototype Resource Utilisation on Virtex-5 LX110T

	% of 69120 LUTs		Latency		BRAM		DSP48		Clocks (MHz)	
Precision	SP	DP	SP	DP	SP	DP	SP	DP	SP	DP
Adder	245	734	11	14	0	0	2	3	410	355
Multiplier	89	309(1%)	8	16	0	0	3	11	493	410
Divider	769	3206(4%)	28	57	0	0	0	0	438	410
2 PEs	2822 (7%)	16%	-	-	10	18	6	22	150	150
4 PEs	6232 (14%)	40%	-	-	20	46	12	33	150	150
8 PEs	14493 (32%)	88%	-	-	40	-	24	-	150	150
16 PEs	(71%)	-	-	-	64	-	48	-	150	-

is readily available from Xilinx's CoreGen [192]. These units can be customised with regards to their wordlength, latency and resource utilisation. We also use Xilinx's FIFO Generator to implement the "Col_buffer", which works in concert with the "Col_Map" unit. We use Synplify Pro 9 and Xilinx ISE 10.1 to implement our prototype on a Xilinx Virtex-5 LX100T FPGA. We limit our implementations to fit on a single FPGA and use off-chip DRAM memory resources for storing the matrix date before it is loaded onto the on-chip BRAM for processing.

Table 5.1 gives the resource cost for different blocks present in our multiple PE design. We can only fit a system of 8 double-precision PEs on a Virtex-5 LX110T with 88% of logic resources being used, whereas 16 single-precision PEs can be easily accommodated. We also notice that as the number of PEs increases beyond 16, the frequency of the system decreases impacting performance. This can be possibly due to a longer critical path. Therefore, we anticipate using an implementation on a bigger FPGA or multiple FPGAs would resolve the issue. Latency is not shown for the multiple-PE design configurations as it greatly depends on the matrix input, as will be illustrated in Section 5.5.

5.3.2 Hardware Debugging

Debugging hardware design on FPGAs requires the design or the insertion of additional logic to monitor and record data outputs during the hardware's operation. In order to debug and verify the behaviour of our implementation, we integrate Xilix's ChipScope cores into our design. ChipScope Pro [193] is an embedded software-based logic analyser, which provides several IP cores, namely, the Integrated Controller (ICON) and the Integrated Logic Analyzer (ILA), Virtual Input/Output (VIO) cores:

The ILA core can be embedded in an FPGA design to collect data when trigger conditions are satisfied. The data size, target signals, and basic trigger conditions can be easily customised during the design phase. ILA can acquire samples from up to 256 nodes, support up to 64 internal trigger and one external trigger signal, and has one clock input. The ILA core uses internal block RAM to store data samples collected.

VIO core is a customisable core that can both monitor and drive internal FPGA signals in real time. Unlike the ILA, no on- or off-chip RAM is required. Two different kinds of inputs (virtual buttons) and two different kinds of outputs (virtual LEDs) are available, both of which are customisable in size to interface with the FPGA design.

The ICON core is embedded in an FPGA design to control up to 15 ILA/VIO cores. This ICON core controls each ILA/VIO core and handles the communication with the ChipScope Logic Analyser software running on a PC over the JTAG Boundary Scan interface.

By inserting the ICON and ILA/VIO cores into a design and connecting them properly, we are able monitor the important signals in the design. In effect, ChipScope also provides the user with a convenient software-based interface (i.e. ChipScope Pro Analyzer) for controlling the ILA/VIO core via setting the triggering options and viewing the waveforms. When a trigger signal becomes active, data is saved onto the BRAMs

before being streamed to the end computer through RS232 or parallel cable for viewing. ILA is customisable in terms of the number of samples it fetches and also the number of triggers it responds to. Figure 5.8 illustrates a simple ChipScope design example showing the interaction of the mentioned cores with an FPGA design.

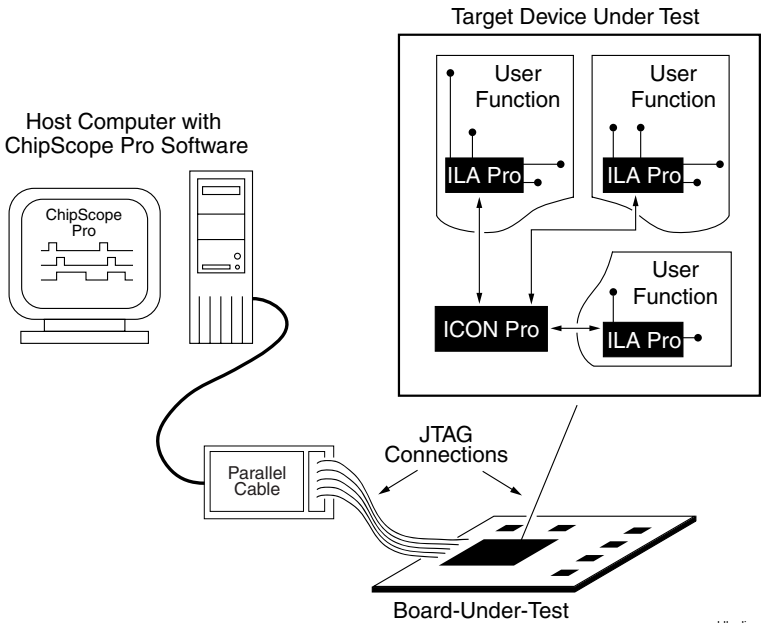


FIGURE 5.8: ChipScope Pro System Block Diagram [193]

5.4 Benchmark Baseline

Prior to evaluating the performance of our hardware design, the performance of three sparse LU factorisation packages (i.e. UMFPACK 5.4, KLU 1.2, and Kundert Sparse 1.3) is measured in terms of their LU decomposition execution times. We intend to use these runtimes as a baseline to measure the hardware acceleration achieved against each of these off-the-shelf packages. To highlight the algorithmic differences between the packages used, we briefly describe them:

UMFPACK [194] implements a right-looking multifrontal algorithm tuned for asymmetric matrices that makes extensive use of BLAS kernels. In our tests, we used UMFPACK's default parameters. In this mode, UMFPACK evaluates the symmetry of the nonzero pattern and selects either the AMD ordering on $A + A^T$ and a strong diagonal preference if the matrix at hand is highly symmetric , otherwise it uses the COLAMD ordering with no preference for the diagonal.

Kundert Sparse [195], implements a right-looking LU factorisation algorithm that preforms dynamic pivoting on the active sub-matrix using the Markowitz ordering algorithm. It is also the sparse solver used in spice3f5, the latest version of the open-source SPICE simulator. Kundert Sparse does not assume matrix symmetry, and hence treats symmetric and asymmetric matrices indifferently. In other words, it does not implement algorithms that take advantage of the structural symmetry of the underlying matrix.

KLU [102] is an LU matrix solver written in C that employs the left-looking Gilbert-Peierls LU factorisation algorithm. KLU has been written specifically to target circuit simulations. A sample KLU code is shown in Figure 5.9. As such, in the first iteration, KLU performs a one-off partial pivoting numerical factorisation (i.e. klu_factor() function) to determine the nonzero structure of the LU factors. In subsequent iterations, KLU reuses the previously-computed nonzero pattern to reduce the factorisation runtimes (i.e. klu_refactor() function). The KLU solver uses matrix preordering algorithms, such as BTF and COLAMD, to minimise fill-in during the initial factorisation phase.

The LU factorisation runtimes for the UFMC benchmark matrices used are reported in Table 5.2. The same pre-ordering (i.e. AMD) was applied to the test matrices prior to factorisation. The tests were performed on a general-purpose linux PC equipped with a 2.67 GHz six-core 12-thread Intel Core Xeon X5650 microprocessor and 6 GB memory. As can be seen from the results, Kundert Sparse offers comparable factorisation runtimes to UMFPACK and KLU for small matrices, outperforming both on several occasions (e.g. Rajat11, Rajat14, Rajat04, fpga_trans_01, fpga_trans_02). However, as the matrix size

```
/* klu_simple: a simple KLU demo */

#include <stdio.h>
#include "klu.h"

int    n = 5 ;
int    Ap [ ] = {0, 2, 5, 9, 10, 12} ;
int    Ai [ ] = { 0,  1,  0,  2,  4,  1,  2,  3,  4,  2,  1,  4} ;
double Ax1 [ ] = {2., 3., 3., −1., 4., 4., −3., 1., 2., 2., 6., 1.} ;
double Ax2 [ ] = {1., 4., 3., −2., 6., 4., −5., 1., 2., 2., 6., 1.} ;

int main (void)
{
    klu_symbolic *Symbolic ;
    klu_numeric *Numeric ;
    klu_common Common ;
    int i ;
    klu_defaults (&Common) ;
    Symbolic = klu_analyze (n, Ap, Ai, &Common) ;
    Numeric = klu_factor (Ap, Ai, Ax, Symbolic, &Common) ;

    // The nonzero patten computed for Ax1 using klu_factor function can be reused by
    // klu_refactor function for matrices (i.e. Ax2) with same pattern
    // but with different nonzero values
     klu_refactor (Ap, Ai, Ax2, Symbolic, Numeric, &Common) ;

    klu_free_symbolic (&Symbolic, &Common) ;
    klu_free_numeric (&Numeric, &Common) ;
    return (0) ;
}
```

FIGURE 5.9: KLU sample code [102]

increases, the performance of Kundert Sparse deteriorates considerably. This is due to fact that at every step of the factorisation, the Markowitz product for all the off-diagonal elements (of the current column and row) has to computed in order to determine the next pivot. Consequently, the Markowitz product computations take longer on bigger matrices, and hence slow down the overall runtime.

Smilarly, we note that UMFPACK outperforms Kundert Sparse for large matrices. In effect, UMFPACK, in contrast to Kundert Sparse, has a higher-level view of the factorisation process as it organises the different computations in a tree structure, thus

enhancing data locality. The latter enables UMFPACK to better utilise the computational resources in the case of high fill-in rates. UMFPACK remains, however, on average about 40% slower than KLU for matrices larger than 1813×1813. This reflects the UMFPACK's inability to effectively reorganise the highly sparse circuit matrices into multiple "frontal" denser matrices, on which BLAS operations can be then applied, as it was discussed in Section 3.2. Overall, KLU demonstrated the shortest LU factorisation runtimes across most of our benchmark matrices, outperforming UMFPACK, and Kundert Sparse by an average of 20% and 80% respectively.

5.5 Performance Analysis

In this section, we present the performance results of the hardware prototype designed. As such, we detail the testing set-up used to evaluate and gauge the operational performance of our sparse LU hardware. We also study the effect of matrix sparsity on the performance of our hardware design. In order to evaluate the performance of our hardware design, we test our parallel architecture with circuit simulation matrices from the University of Florida Sparse Matrix Collection (UFMC). The performance measurements are then compared to the state-of-the-art UMFPACK, KLU, and Kundert sparse LU decomposition matrix packages. In our performance evaluation, we use the CPU time reported by UMFPACK 5.4, Kundert Sparse 1.3, and KLU 1.2 on a 64-bit Linux system running on a 6-core Intel Xeon 2.6 GHz processor with 6 GB RAM, as a benchmark.

To gauge the time taken by our FPGA-based LU decomposition architecture, we use Xilinx's ChipScope Integrated Logic Analyser (ILA) to count the number of clock cycles required to perform the LU decomposition. The ILA is triggered and stopped by two handshaking signals, namely a *start* and a *done* signal, we added to our design for this purpose. We used the same the pre-ordering (i.e. AMD) for LU matrix packages and our Sparse LU Hardware. Table 5.2 contains the relevant properties of the test matrices used and the corresponding LU decomposition runtimes reported by UMFPACK, KLU, and Kundert Sparse. Table 5.3 shows the execution time of LU FPGA hardware as

TABLE 5.2: Performance comparison of UMFPACK, Kundert Sparse, and KLU runtimes

Matrix properties						CPU runtimes for		
Matrix	Order	NNZ *	Sparsity (%)	Str Sym**	Num Sym***	UMFPACK (ms)	Kundert Sparse (ms)	KLU (ms)
Rajat11	135	665	3.600	89.10%	63%	0.003	0.002	0.019
Rajat14	180	1,475	0.040	100%	2%	0.020	0.011	0.029
oscil_dcop_11	430	1544	0.800	97.60%	69.80%	0.583	0.793	0.329
circuit204	1020	5883	5.600	43.80%	37.30%	0.243	0.909	0.482
Rajat04	1,041	8725	0.800	100%	4%	0.035	0.021	0.033
Rajat19	1157	3699	0.298	91%	92%	0.217	0.333	0.202
fpga_dcop_50	1220	5892	0.400	81.80%	33.20%	1.093	1.200	0.685
fpga_trans_01	1,220	7,382	0.500	100%	21%	0.030	0.011	0.043
fpga_trans_02	1,220	7,382	0.500	100%	21%	0.032	0.010	0.051
fpga_dcop_01	1813	5892	0.179	65%	1.60%	0.547	1.087	0.511
init_adder1	1813	11156	0.300	65.40%	1.60%	0.567	1.035	0.480
adder_ dcop_57	1813	11246	0.300	64.80%	0.80%	0.464	1.464	0.363
adder_trans_01	1,814	14,579	0.440	100%	3%	0.024	0.044	0.039
adder_trans_02	1,814	14,579	0.440	100%	3%	0.023	0.048	0.041
Rajat12	1,879	12,818	0.360	100%	45%	0.119	0.121	0.118
Rajat02	1960	11,187	0.300	100%	100%	1.034	1.028	0.921
add20	2,395	13,151	0.230	100%	53%	0.861	1.021	0.460
bomhof1	2624	35823	0.520	100%	21%	4.550	7.181	2.675
bomhof2	4510	21,199	0.104	81%	41%	3.944	5.974	1.950
add32	4,960	19,848	0.080	100%	31%	1.740	3.088	1.412
meg4	5,860	25,258	0.070	100%	100%	0.723	0.923	0.514
hamrle2	5952	22162	0.600	0.10%	0%	0.693	2.075	0.551
Rajat01	6833	43520	0.093	99.60%	99%	1.910	1.981	1.181
Rajat13	7,598	48,762	0.080	100%	30%	1.941	3.150	1.014
Rajat03	7,602	32,653	0.060	100%	40%	1.096	2.113	0.935
Rajat06	10,922	46,983	0.040	100%	100%	1.096	1.246	0.972
bomhof3	12127	48137	0.300	77%	30%	5.428	7.764	3.306

* Number of nonzero elements.
** Numerical Symmetry is the fraction of nonzeros matched by equal values in symmetric locations.
*** Structural Symmetry is the fraction of nonzeros matched by nonzeros in symmetric locations.

reported by ChipScope, and the FPGA acceleration achieved using 16 single-precision PEs running at 150 MHz. The acceleration is calculated as a ratio of the CPU time taken by a given LU matrix package over the time spent by the sparse LU hardware on the same circuit matrix:

$$Speedup = \frac{T_{CPU}}{T_{FPGA}} = \frac{T_{CPU}}{FPGA_{cycles} \times (1/frequency)} \tag{5.1}$$

where T_{CPU} is the LU factorisation time taken by the software package, T_{FPGA} is the LU factorisation time taken by the hardware prototype, $FPGA_{cycles}$ is the number of clock cycles taken by the hardware prototype to compute the LU factorisation of given matrix, and $frequency$ is the overall clock frequency of the hardware design.

The speedup results tabulated in Table 5.3 are also illustrated graphically in Figure 5.10. For the test matrices used, we can clearly see that our 16-PE LU hardware outperforms KLU, UMFPACK, and Kundert Sparse on average by factors of 9.65, 11.83, 17.21, respectively. Furthermore, we note a correlation between the matrix sparsity and speedup ratio of our design. We also remark that the best acceleration results were achieved when the matrix is very sparse and has a symmetric or near-symmetric pattern (e.g. rajat13, add32, meg4). In effect, high sparsity implies that less column-level dependencies will exist during the course of Gilbert-Peierls LU factorisation, and thus increases the parallelism potential as shown in Section 4.3. On the other hand, higher structural symmetry implies a more balanced elimination graph, which translates into a more balanced workload which minimises the idle time of the different PEs, leading to a busier computational pipeline.

To illustrate the correlation observed between the hardware acceleration ratios achieved and matrix sparsity, we isolate the effect of matrix sparsity by selecting test matrices that have symmetric nonzero patterns with varying sparsities, as shown in Table 5.4. Then, we plot the acceleration achieved by our LU hardware as a function of the matrix sparsity, as depicted in in Figure 5.11. We can see that as the nonzero density decreases, the acceleration ratio also increase. In other words, the sparse LU hardware performance

increases as sparsity increases and vice versa. In effect, the sparser the matrix, the wider the column elimination graph and hence more columns can be processed in parallel.

TABLE 5.3: LU decomposition hardware acceleration achieved versus UMFPACK, Kundert Sparse, and KLU

Matrix Name	FPGA		FPGA speedup*** (×) achieved versus		
	Latency* (Cycles)	Time** (ms)	UMFPACK	KLU	Kundert Sparse
Rajat11	249	0.002	2.05	11.14	1.44
Rajat14	370	0.002	8.10	11.75	4.53
oscil_dcop_11	3,397	0.023	25.74	14.54	35.02
circuit204	9,103	0.061	4.00	7.94	14.97
Rajat04	1,049	0.007	5.00	4.72	2.97
Rajat19	3,047	0.020	10.68	9.96	16.41
fpga_dcop_50	9,960	0.066	16.47	10.31	18.07
fpga_trans_01	1,100	0.007	4.09	5.86	1.46
fpga_trans_02	1,007	0.007	4.76	7.59	1.56
fpga_dcop_01	7,055	0.047	11.62	10.87	23.11
init_adder1	5,479	0.037	15.52	13.13	28.33
adder_ dcop_57	7,981	0.053	8.71	6.82	27.51
adder_trans_01	1,221	0.008	2.95	4.79	5.40
adder_trans_02	1,116	0.007	3.09	5.51	6.45
Rajat12	2,023	0.013	8.82	8.77	8.97
Rajat02	17,866	0.119	8.68	7.74	8.63
add20	9,710	0.065	13.30	7.11	15.77
bomhof1	68,651	0.458	9.94	5.84	15.69
bomhof2	37,081	0.247	15.95	7.89	24.17
add32	13,320	0.089	19.59	15.90	34.77
meg4	3,694	0.025	29.35	20.85	37.48
hamrle2	16,670	0.111	6.23	4.96	18.67
Rajat01	10,219	0.068	28.04	17.34	29.08
Rajat13	15,126	0.101	19.25	10.05	31.24
Rajat03	20,405	0.136	8.06	6.87	15.53
Rajat06	10,344	0.069	15.89	14.10	18.06
bomhof3	60,266	0.402	13.51	8.23	19.33
Average	-	-	11.83	9.65	17.21

* Number of the FPGA clock cycles taken to compute the LU factorisation.
** Time taken to complete the LU factorisation on an FPGA accelerator running at 150 MHz.
*** Using 16 single-precision PEs running at 150 MHz.

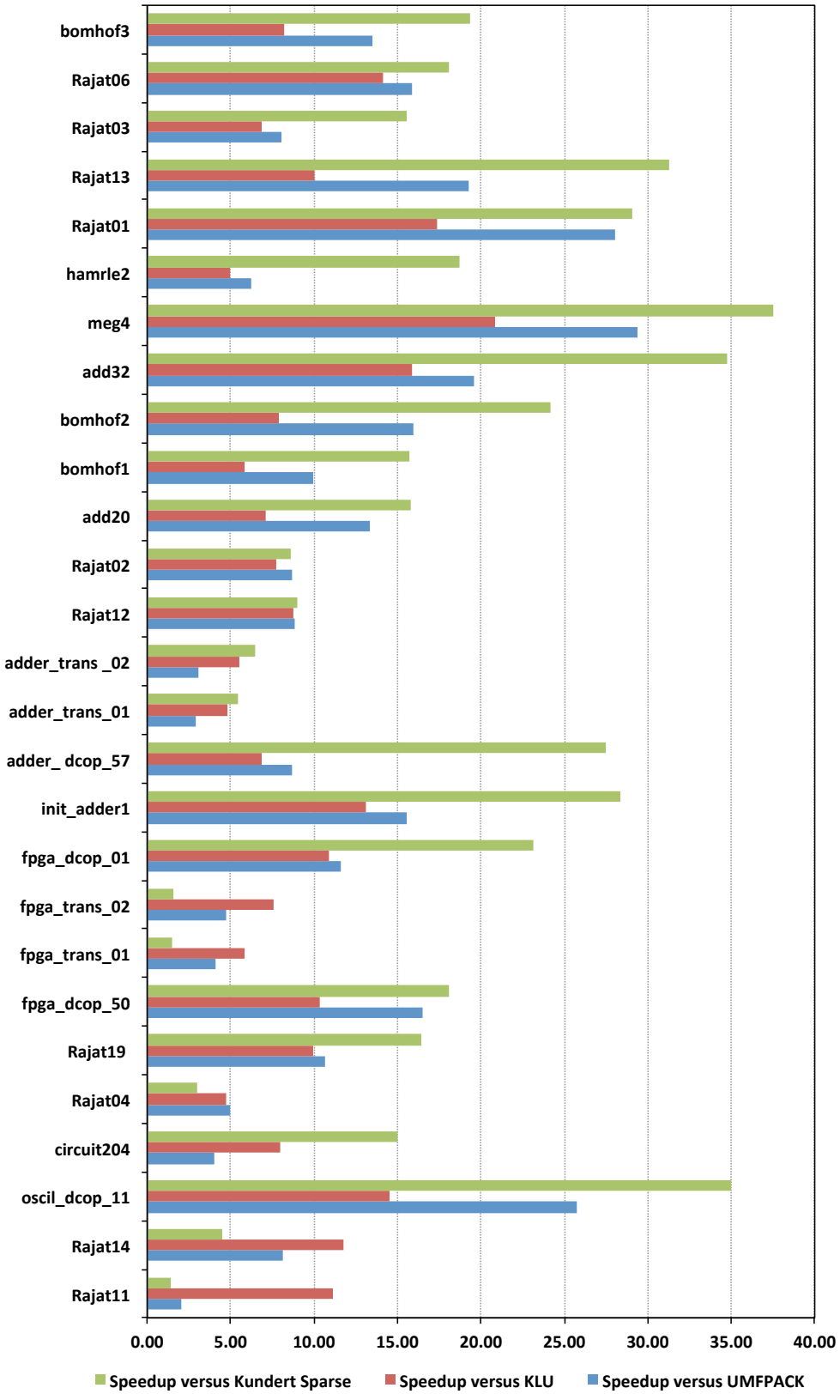


FIGURE 5.10: LU decomposition FPGA acceleration achieved versus KLU, Kundert Sparse, and UMFPACK

TABLE 5.4: Sparsity effect on the acceleration ratios of the LU hardware prototype

			Speedup** achieved versus		
Matrix	Sparsity (%)	Str Sym* (%)	UMFPACK (×)	KLU (×)	Kundert Sparse (×)
Rajat04	0.800	100	5.00	4.72	2.97
bomhof1	0.520	100	9.94	5.84	15.69
fpga_trans_01	0.500	100	4.09	5.86	1.46
fpga_trans_02	0.500	100	4.76	7.59	1.56
adder_trans_01	0.440	100	2.95	4.79	5.40
adder_trans_02	0.440	100	3.09	5.51	6.45
Rajat12	0.360	100	8.82	8.77	8.97
Rajat02	0.300	100	8.68	7.74	8.63
add20	0.230	100	13.30	7.11	15.77
add32	0.080	100	19.59	15.90	34.77
Rajat13	0.080	100	19.25	10.05	31.24
meg4	0.070	100	29.35	20.85	37.48
Rajat03	0.060	100	8.06	6.87	15.53
Rajat14	0.040	100	8.10	11.75	4.53
Rajat06	0.040	100	15.89	14.10	18.06

* Structural Symmetry is the fraction of nonzeros matched by nonzeros in symmetric locations.
** Using 16 single-precision PEs running150 MHz.

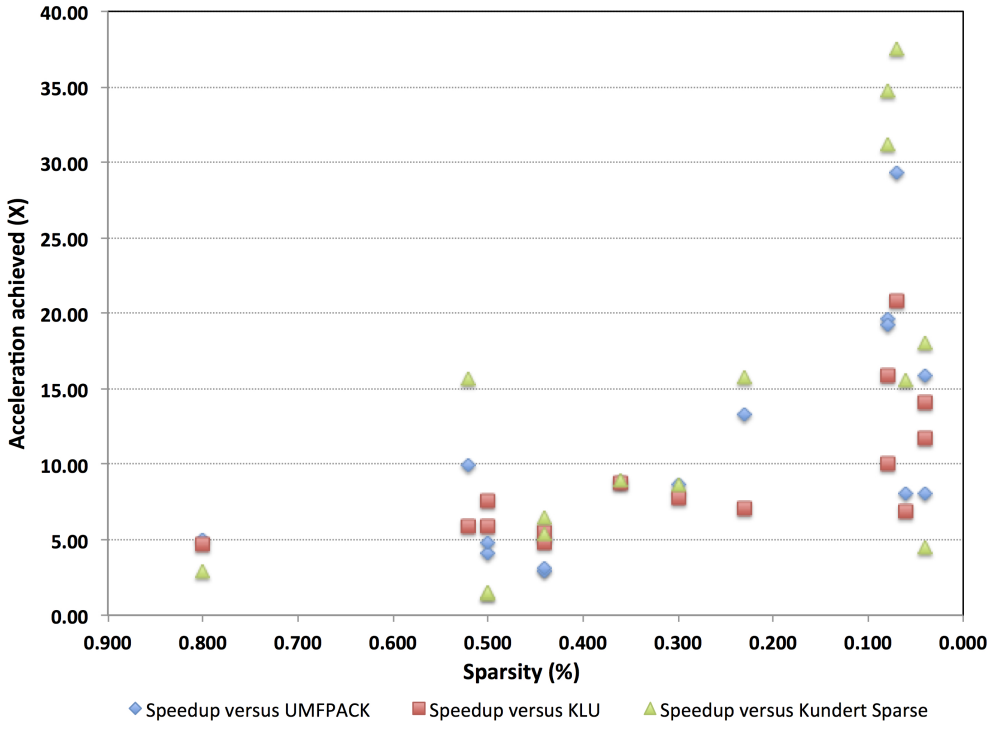


FIGURE 5.11: The impact of matrix sparsity on the performance of the LU FPGA hardware

5.5.1 Cost of the pre-processing stage

As we mentioned earlier, KLU and our FPGA design rely on information computed in the symbolic stage to speedup subsequent factorisations. In effect, during the symbolic stage, KLU performs a one-off partial pivoting numerical factorisation to determine the nonzero structure of the LU factors. In the subsequent iterations, KLU reuses the previously-computed nonzero pattern to reduce the factorisation runtimes. In our work, we use the pre-processing steps described in Section 4.3 to perform symbolic analysis and to compute the DAMOS scheduling graph. The latter is used to parallelise the actual numerical factorisation on the FPGA. Therefore, we demonstrate how the cost of this symbolic stage can be amortised over a number of iterations such as the SPICE iterations. Table 5.5 tabulates the CPU runtimes for the symbolic stage of KLU as well the time taken by our pre-processing stage (i.e. DAMOS). From the reported runtime figures, we note that our pre-processing stage is on average 20% faster than KLU's symbolic analysis stage. This reflects the fact that KLU performs a one-time numerical factorisation during this stage, whereas in our symbolic analysis step we only rely on the graph representation of the underlying matrix. We can also see that the time taken by the KLU symbolic stage is on average 5.1× the KLU factorisation runtime on a CPU. On the other hand, the time taken by our DAMOS pre-processing stage is on average 36× the factorisation time on the FPGA. However, this symbolic overhead is a one-off effort, which can be easily amortised over a number of iterations, as demonstrated by the following equation:

$$Overhead_{symbolic} = \frac{T_{symbolic}}{T_{factorisation}} = \frac{36 \times T_{FPGA}}{i \times T_{FPGA}} = \frac{36}{i} \tag{5.2}$$

where $T_{symbolic}$ is the time taken by our DAMOS pre-preprocessing step on a 6-core Intel Xeon microprocessor, $T_{factorisation}$ is the time taken by our FPGA LU decomposition hardware, and i is the number of SPICE iterations. For instance, if a given simulation requires 10,000 iterations, then symbolic analysis overhead will only account for 0.36% of the overall LU factorisation runtime.

TABLE 5.5: Cost of the symbolic analysis in KLU and DAMOS

	KLU		FPGA	
Matrix	Symbolic stage (ms)	LU (ms)	Symbolic stage (ms)	LU (ms)
Rajat11	0.081	0.019	0.089	0.002
Rajat14	0.103	0.029	0.124	0.002
oscil_dcop_11	1.542	0.329	1.314	0.023
circuit204	1.562	0.482	1.125	0.061
Rajat04	0.158	0.033	0.147	0.007
Rajat19	1.070	0.202	0.747	0.020
fpga_dcop_50	2.425	0.685	2.724	0.066
fpga_trans_01	0.209	0.043	0.170	0.007
fpga_trans_02	0.255	0.051	0.192	0.007
fpga_dcop_01	2.559	0.511	2.150	0.047
init_adder1	2.586	0.480	1.725	0.037
adder_ dcop_57	4.642	0.363	1.376	0.053
adder_trans_01	0.212	0.039	0.150	0.008
adder_trans_02	0.190	0.041	0.161	0.007
Rajat12	0.559	0.118	0.446	0.013
Rajat02	4.275	0.921	4.018	0.119
add20	2.332	0.460	1.937	0.065
bomhof1	16.193	2.675	9.979	0.458
bomhof2	9.685	1.950	7.881	0.247
add32	7.475	1.412	5.945	0.089
meg4	2.515	0.514	0.860	0.025
hamrle2	2.983	0.551	2.419	0.111
Rajat01	5.493	1.181	4.611	0.068
Rajat13	5.098	1.014	3.918	0.101
Rajat03	4.222	0.935	3.782	0.136
Rajat06	5.745	0.972	4.027	0.069
bomhof3	24.180	3.306	12.914	0.402

* Number of the FPGA clock cycles taken to compute the LU factorisation.
** Time taken to complete the LU factorisation on a accelerator running at 150 MHz.
*** Using 16 single-precision PEs running150 MHz.

5.5.2 Scalability

In order to study the scalability trends of our design, we gauge the performance of our design with 2, 4, 8, and 16 PEs configurations. We use the KLU runtimes, reported in Table 5.2, as a benchmark to calculate the speedups achieved per design configuration using Equation 5.1. The FPGA LU factorisation runtimes per PE count and their corresponding speedups are reported in Table 5.6. We then plot the acceleration achieved for the benchmark matrices as a function of the number of PEs, as illustrated in Figure 5.12. We can see that the acceleration grows almost linearly with the number of PEs, with an average 60% acceleration boost as we double the PE count. This suggests that if we employ higher PE configurations on a larger FPGA (i.e more than 16 PEs), we may able to attain higher speedups ratios, provided that the observed acceleration trend is maintained (e.g. Equation 5.3).

TABLE 5.6: Sparse LU FPGA accelerator performance scaling trends

	2 PEs*		4 PEs*		8 PEs*		16 PEs*	
Matrix	LU Time** (ms)	Speedup*** (×)	LU Time** (ms)	Speedup (×)	LU Time** (ms)	Speedup*** (×)	LU Time** (ms)	Speedup*** (×)
Rajat11	0.005	3.46	0.003	6.14	0.002	9.20	0.002	11.14
Rajat14	0.016	1.80	0.009	3.21	0.005	5.88	0.002	11.75
oscil_dcop_11	0.126	2.62	0.069	4.77	0.037	8.98	0.023	14.54
circuit204	0.379	1.27	0.257	1.88	0.113	4.26	0.061	7.94
Rajat04	0.030	1.09	0.024	1.40	0.016	2.06	0.007	4.72
Rajat19	0.120	1.68	0.063	3.22	0.045	4.46	0.020	9.96
fpga_dcop_50	0.326	2.10	0.194	3.53	0.117	5.84	0.066	10.31
fpga_trans_01	0.035	1.21	0.026	1.64	0.015	2.96	0.007	5.86
fpga_trans_02	0.032	1.60	0.024	2.12	0.011	4.75	0.007	7.59
fpga_dcop_01	0.333	1.54	0.179	2.86	0.082	6.24	0.047	10.87
init_adder1	0.132	3.65	0.079	6.06	0.043	11.19	0.037	13.13
adder_ dcop_57	0.255	1.42	0.175	2.07	0.115	3.15	0.053	6.82
adder_trans_01	0.032	1.22	0.028	1.41	0.015	2.58	0.008	4.79
adder_trans_02	0.037	1.10	0.024	1.74	0.013	3.26	0.007	5.51
Rajat12	0.098	1.21	0.054	2.20	0.019	6.30	0.013	8.77
Rajat02	0.801	1.15	0.604	1.52	0.251	3.68	0.119	7.74
add20	0.381	1.21	0.238	1.93	0.080	5.73	0.065	7.11
bomhof1	1.851	1.45	1.631	1.64	0.985	2.72	0.458	5.84
bomhof2	1.464	1.33	1.281	1.52	0.596	3.27	0.247	7.89
add32	0.723	1.95	0.338	4.18	0.146	9.69	0.089	15.90
meg4	0.079	6.49	0.043	11.96	0.029	17.79	0.025	20.85
hamrle2	0.509	1.08	0.322	1.71	0.157	3.52	0.111	4.96
Rajat01	0.527	2.24	0.292	4.05	0.152	7.74	0.068	17.34
Rajat13	0.583	1.74	0.331	3.07	0.128	7.91	0.101	10.05
Rajat03	0.676	1.38	0.490	1.91	0.248	3.77	0.136	6.87
Rajat06	0.516	1.89	0.238	4.08	0.128	7.57	0.069	14.10
bomhof3	2.794	1.18	1.758	1.88	0.927	3.57	0.402	8.23
Arithmetic Average	-	1.85	-	3.10	-	5.85	-	9.65
Geometric mean	-	1.66	-	2.63	-	5.10	-	8.90

* Single-precision PEs running150 MHz.
** Time taken to complete the LU factorisation on the FPGA accelerator running at 150 MHz.
*** Speedup verus KLU runtimes on a 6-core Intel Xeon microprocessor reported in Table 5.2.

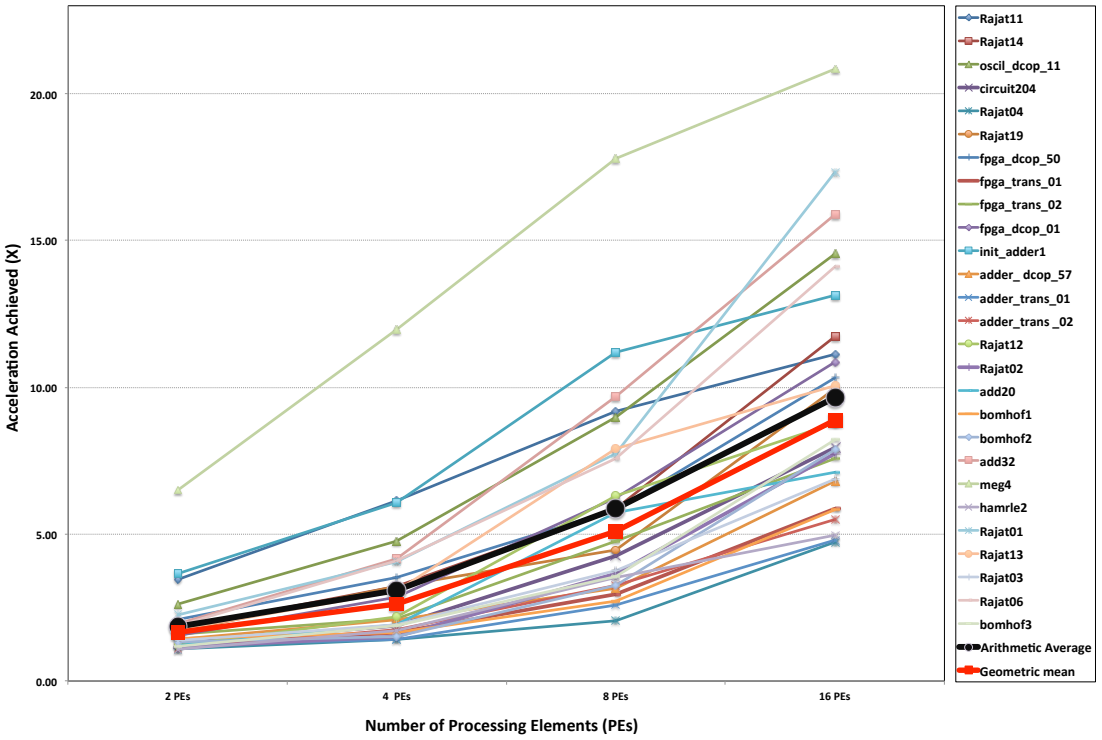


FIGURE 5.12: Sparse LU FPGA acceleration scaling trends in terms of PEs

The acceleration potential of our design can be further improved by increasing the frequency of the overall design clock. Referring to Equation 5.1, we can see that if we manage, for instance, to double the design's frequency, we will be effectively cutting down the FPGA LU time to half, and thus doubling the acceleration ratios achieved so far. The frequency of our design is primarily limited by two things: the frequency & latency of the CoreGen floating-point operators and the inter-PE fully connected switch. The frequency & latency of the CoreGen floating-point operators greatly depend on the Xilinx FPGA family used and the degree to which the physical DSP48 blocks are used (i.e. none, full, maximum). Table 5.7 shows the resource utilisation of our design if a Virtex-7 XC7V200T is used. As we can see, Xilinx ISE 14 synthesis results indicate that the overall design frequency has increased from 150 MHz to 250MHz. This is mainly due to customising the CoreGen floating-point divider latency to 1 clock cycle as compared to 28 cycles for the same operator on the Virtex 5. This higher overall frequency indicates that we can now expect that our acceleration ratios on the Virtex 7 to increase at same rate (i.e 1.6×), as illustrated by Equation 5.4. In other words, changing the target

FPGA from Virtex 5 to Virtex 7 improves the average 16 PEs speedup ratio from 9.65× 15.44× (i.e 9.65 ×1.6). The overall predicted speedup that can be achieved by using a 32-PE configuration on the more modern Viretx 7 is shown in Equation 5.5.

$$Speedup^{32PEs} = Speedup^{16PEs} \cdot (1.6) \tag{5.3}$$

$$Speedup^{32PEs}_{viretx7} = \frac{frequency_{viretx7}}{frequency_{viretx5}} \cdot Speedup^{32PEs}_{viretx5} \tag{5.4}$$

$$Speedup^{32PEs}_{viretx7} = \frac{frequency_{viretx7}}{frequency_{viretx5}} \cdot Speedup^{16PEs}_{viretx5} \cdot (1.6) \tag{5.5}$$

TABLE 5.7: Sparse LU Hardware Prototype Resource Utilisation on a Virtex-7 XC7V200T

	Usage of 1,954,560 LUTs		Latency		BRAM		DSP48		Clocks (MHz)	
Precision	SP*	DP**	SP	DP	SP	DP	SP	DP	SP	DP
Adder	407	794	8	8	0	0	0	0	472	436
Multiplier	103	279	6	16	0	0	3	11	463	403
Divider	1,106	3,412	1	1	0	0	0	0	482	375
1 PEs	4,931	16,080	-	-	5	10	3	11	250	250
16 PEs	17,2121 (8%)	590,576 (30%)	-	-	64	136	48	176	250	250
32 PEs	467,342 (24%)	1,456,950 (74%)	-	-	142	283	96	352	250	250

*Single-precision **Double-precision

5.6 Summary

In this chapter, we showed an FPGA implementation of the "Sparse LU Factorisation", key computational kernel to the SPICE matrix solution phase, that harnesses the parallelism exposed at the pre-precessing stage of circuit matrices using specialised techniques. Using benchmark matrices from the UFMC repository, we empirically demonstrated that our 16-PE LU Virtex 5 implementation outperforms modern LU matrix software packages, running on a 6-core 12-thread Intel Xeon X5650 microprocessor, by many times.

In effect, we showed that our LU FPGA implementation is on average 9.65×, 11.83×, 17.21× faster than KLU, UMFPACK, and Kundert Sparse matrix packages respectively. We have also extrapolated our acceleration result to the more modern Virtex 7 FPGA family and we predict that acceleration results to be 1.6× faster than the same PE configuration on the Virtex 5 due to the improved overall design frequency. In the next chapter, in line with the principles covered in Section 2.2, we study the feasibility of creating a multiple FPGA system using cheaper medium-range Virtex 5 COTS board able to outperform the more modern and more expensive Virtex 7 boards.

Chapter 6

Multi-FPGA Matrix Solution

In the previous chapter, we evaluated the performance our single-FPGA LU decomposition prototype and we demonstrated that it can outperform modern software packages by many times. Scaling trends of our design also suggest that doubling the number of PEs can on average lead to a 60% performance increase. However, as a design increases in size and area, the critical path also increases and the circuit's frequency of operation gets reduced as a result. Nonetheless, empirical results from Chapter 5 (Section 5.5.2) show that utilising a larger FPGA improves the design's operating frequency leading to higher acceleration ratios. To achieve even higher speedup ratios, the sparse matrix at hand can be partitioned into smaller pseudo-independent entities that can be spread over a number of FPGA for processing. In this chapter, we explore how our single-FPGA design can be adapted to a multi-FPGA LU factorisation system able to harness the coarse-grain parallelism present within circuit matrices.

6.1 Objective

In addition to the fill-in reducing orderings covered in Chapter 4, the nonzero pattern of sparse matrices can be reorganised into specialised forms that expose data parallelism, which can be then harnessed by a Single Instruction Multiple Data (SIMD) [196] processing architecture. Such orderings exploit the sparsity of a given matrix to reorder it into sub-matrices with can be solved concurrently. As such, one of our main objectives in this chapter is to demonstrate a methodology to effectively partition sparse circuit matrices into almost independent smaller sparse matrices interconnected by an interface matrix (i.e. coupling equations), as shown in Figure 6.1. The resulting sub-matrices can be then factorised concurrently using a multiple FPGA system in a SIMD fashion, where each FPGA node will responsible for independently factorising a sub-matrix using the single-FPGA design we proposed in previous chapter. The interface problem is factorised last, once the factorisation of all the sub-matrices is complete. One of most widely used partitioning schemes is to reorder a sparse matrix into the Bordered Diagonal Block (BDB) form [197, 198, 199, 200] using the node tearing technique [153], nested dissection [168], or similar heuristics. In this next section, we discuss the the advantages of the BDB matrix form in the realm of parallelising the LU factorisation of sparse matrices.

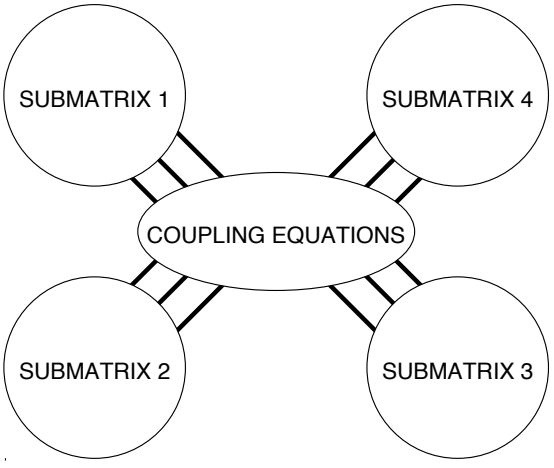


FIGURE 6.1: Graph with four independent sub-matrices [201]

6.2 Ordering for Coarse-grain Parallel Factorisation

As mentioned earlier, to achieve coarse-grained parallelism, sparse matrices can be pre-ordered into the BDB matrix form. The latter exposes data parallelism inherently present within sparse matrices by reordering them into the form shown in Equation 6.1. A_{in} and A_{nj}, A_{nn}, are known as *"the right border"*, *"the bottom border"*, and *"the diagonal blocks"* respectively, where A is $n \times n$ sparse matrix. The blocks A_{nn}, A_{in} and A_{nj} are said to form a *"3-block group"* (e.g. $[A_{11}, A_{1n}, A_{n1}]$). All other off-diagonal blocks contain only zeros, and hence no fill-in elements will appear in these blocks. The factorisation of the last block A_{nn} requires the data produced in the right and bottom border blocks. Therefore, all other 3-block groups can be processed in parallel and the last diagonal block, A_{nn}, is factorised last. The factorised BDB matrix retains the BDB structure and hence data parallelism can be also harassed at the forward reduction and back substitution stages of the matrix solution. Once a BDB ordering is obtained, local fill-in reducing heuristics can be applied to sub-matrices.

$$\begin{bmatrix} A_{11} & & & & A_{1n} \\ & A_{22} & & & A_{2n} \\ & & \ddots & & \vdots \\ & & & A_{n-1n-1} & A_{n-1n} \\ A_{n1} & A_{n2} & \dots & A_{nn-1} & A_{nn} \end{bmatrix} \tag{6.1}$$

Assuming that no pivoting is required or restricted within diagonal blocks, the LU factorisation of the BDB sparse matrix involves four steps:

- Factorisation of the independent 3-block group sub-matrices.
- Multiplication of the right and bottom border blocks to generate the partial sums.
- The accumulation of the partial results for the last diagonal block.
- Factorisation of the last diagonal block using the accumulated partial sums.

The computation of the last diagonal block cannot begin until all the contributions from the diagonal block are accumulated. Figure 6.2 illustrates how a matrix in the BDB form can be mapped to four processing elements (i.e. P1, P2, P3, and P4) for parallel factorisation. For the method to work well in a parallel environment, the order of the interface problem (i.e. the last diagonal block) should be small compared with the size of the original matrix so that the cost of factorising the interface problem is significantly less than that of factorising the blocks on the diagonal [202]. In effect, the smaller the interface block A_{nn} gets, less the communication overhead will be. However, as A_{nn} gets smaller, it also gets denser.

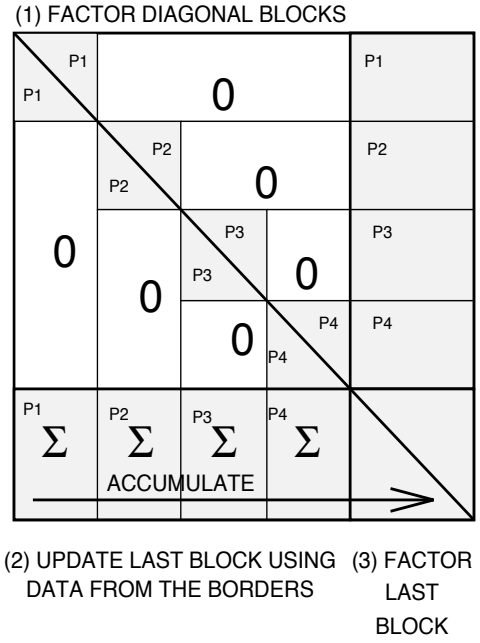


FIGURE 6.2: Factorisation steps of a matrix in the Bordered Diagonal Bock (DBD) form [201]

Several algorithm can be used to re-organise a sparse matrix into the BDB form [203, 204, 205, 206, 207, 208, 209]. However, the most widely used technique to generate the BDB form is recursive partitioning of the graph associated with the matrix at hand using dissection algorithms [168]. These algorithms attempt to split the matrix graph into equal partitions. The resulting partitions are connected by a set of node referred to as the *"node separators"*. The edges which have to be cut as a result of removing the the

node separators is referred to as the *"edge cut"*. In the context of a BDB structure, each graph partition represent a sub-matrix whereas the node separators reflect the coupling equations (i.e. the interface matrix). Graph dissection can be applied recessively to the resulting leading to the nested BDB illustrated in Figure 6.16.

State-of-the-art nested dissection algorithms use *"multilevel graph partitioning"*. A widely used nested dissection routine is *"METIS NodeND"* from the METIS graph partitioning package [207]. Multilevel schemes aim to balance the time required to determine a partition and its quality. These methods are called multilevel because they operate by repeatedly simplifying the original graph and using the resulting graph to generate the partitions. The basic steps in a multilevel scheme are: coarsening, partitioning, and refinement. During coarsening, the original graph is simplified by collapsing the edges and the vertices to create a smaller simpler graph. In the next phase, the simplified graph is partitioned into two roughly equal-sized parts, while maintaining a small edge-cut. In the refinement step, the bisected simplified graph is transformed back into the original graph. The latter has now more freedom in selecting nodes, which can be used to refine further the coarse bisections. For asymmetric matrices, the algorithms discussed above use the graph associated with the symmetrised matrix $A + A^T$ or $A^T A$.

6.3 Inter-FPGA Communication

High-Speed communication is a crucial and an integral part of digital systems and their performance. However, nowadays, systems interconnect is considered to be the primary bottleneck at all communication levels; intra-chip, inter-chip or board-to-board [210]. Parallel I/O remains one of the most popular interconnect technology to date. It usually employs a central arbiter (i.e. Master) that allows sharing a common bus between several clients (i.e slaves). However, the obvious limitation of such buses is the restricted scalability due to the limited bandwidth, which in turn limits the capabilities of the clients. Moreover, as clock speeds continue to grow, signal skews grow dramatically causing communicating partners to go out of phase [211]. High clock speeds also cause

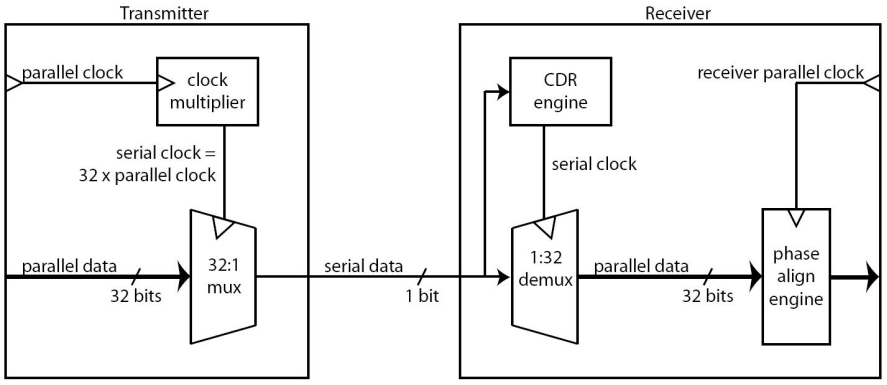


FIGURE 6.3: A Simplified Serial Communication Example

more interference and cross talk undermining the signal integrity. To remedy these pitfalls, complex and expensive synchronisation logic is used which increases the design costs. Additionally, this technology puts more strain on PCB engineers due the huge number of traces to deal with. Therefore, designers opt for multilayer PCBs which in turn increase the overall costs.

6.3.1 FPGA High Speed Serial Transceivers

As part of overcoming the issues discussed earlier, the silicon industry has been shifting focus to multi-gigabit serial I/O [212]. This trend has been reflected in the offerings of leading FPGA manufacturers such as Altera and Xilinx. In effect, they have incorporated MGTs into some of their high-end devices. MGTs effectively eliminate clock-to-data skew through the use of Clock and Data Recovery (CDR) [213]. CDR consists in sending high speed serial data streams without an embedded clock. At the receiving end, an approximate clock is generated from a known reference point. The clock is then phase-aligned to the transitions in the data stream with a Phase-Locked Loop (PLL) as shown in Figure 6.3. But in order for this scheme to work, a data stream must transition frequently enough to ensure that any drift in the PLL's oscillator is corrected. 8B/10B encoding is commonly used to produce a DC-balanced and transition-rich data stream [214]. This technology also reduces the number of traces running across boards significantly and hence decreases the number of PCB layers needed considerably. Serial

communication has also many other obvious advantages, namely, the reduction of power consumption and pin number usage [215].

6.3.2 The Xilinx Aurora Protocol

In our work, we make use of the Xilinx Aurora protocol [216] for serial communication through Serial-ATA (SATA). Aurora is a scalable and lightweight point-to-point protocol that provides a simplified interface to the FPGA MGTs. Figure 6.4 shows how Aurora can be used to connect two user applications in two different FPGAs. As illustrated in the diagram, each connection between MGTs is called a lane. Any number of lanes can be bonded to create an Aurora channel. Randomised idle sequences are injected into a channel whilst it is not used. Aurora uses 8B/10B encoding for DC balance, error detection, and to allow control characters in the data stream.

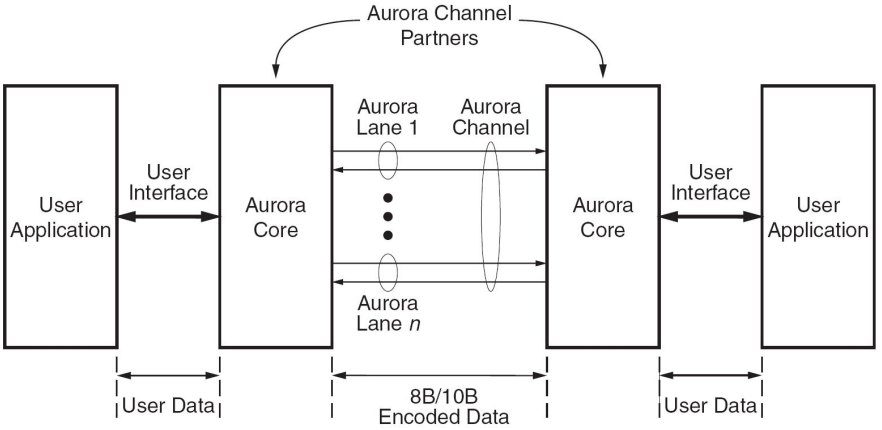


FIGURE 6.4: Functional view of the Aurora Protocol [216]

Figure 6.5 depict the top-level interfaces available in Aurora. The LocalLink interface is the primary interface for the communication of raw data. When a data packet is passed to it on the sending ports, Aurora encapsulates it in 8B/10B control characters as necessary to be correctly interpreted by the MGT core. Upon reception, the control characters are stripped and data is presented to the LocalLink interface receiving ports [216]. Aurora supports two modes of operation: framing mode and streaming mode. The framing interface comprises signals necessary for transmitting and receiving framed user data. Conversely, the streaming interface allows users to send data without any special

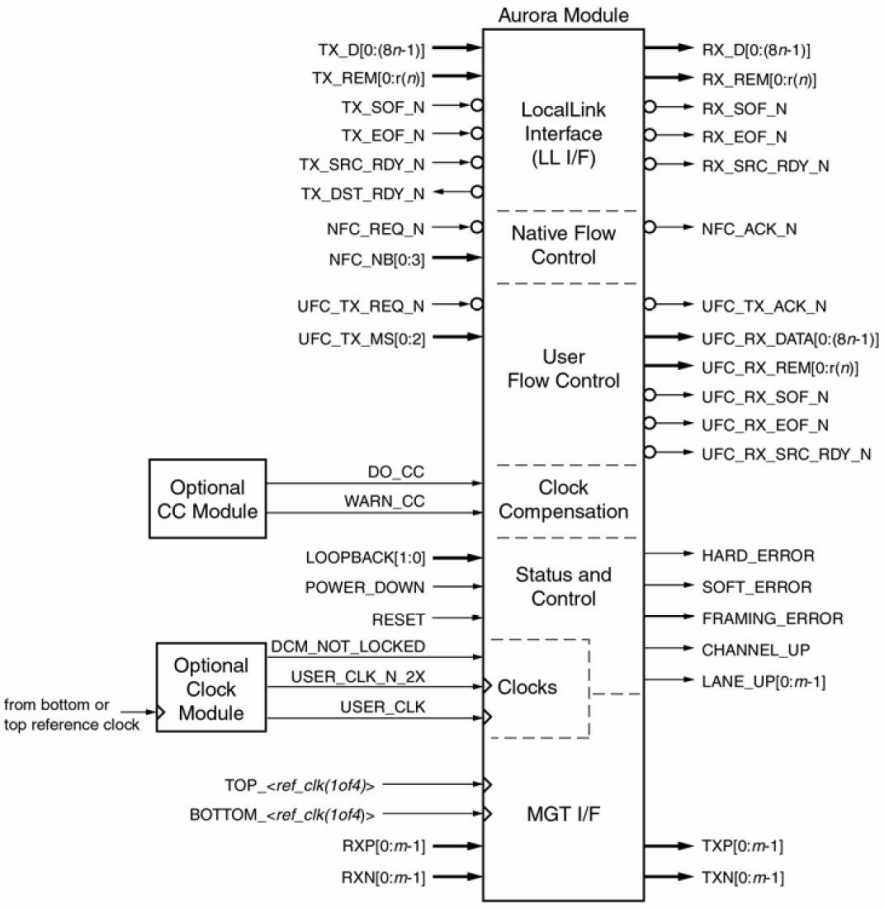


FIGURE 6.5: Aurora interfaces [216]

frame delimiters, allowing the Aurora channel to be used as a pipe. Words written into the TX side of the channel are delivered to the RX side after some latency. The streaming interface is simple to operate and uses fewer resources than framing. Two optional flow control interfaces can be associated with the framing interface. Native flow control (NFC) is used for regulating the data transmission rate to prevent FIFO overflows. User flow control (UFC) is used to exchange high priority messages between application partners. Additionally, Aurora cores can be configured as full-duplex or simplex modules. Full-duplex modules provide high-speed TX and RX links whereas simplex modules provide a link in only one direction.

All data transferred via Aurora is sent in 2-byte code groups which naturally fit Xilinx's 16-bit MGTs interface. The 8B/10B encoding allows the Aurora core to detect

all single bit errors and most multi-bit errors that occur in the channel. Aurora resets itself upon detecting a hard error. The Aurora protocol has an average latency depends on the customisation options the user choses. However, The Aurora protocol has a constant throughput and thus the data to be sent can be sampled every clock cycles and the data on the receiving end has to be consumed instantly, otherwise it gets destroyed by the following word in the next clock cycle.

6.3.3 Experimental Aurora Tests

For our research, we conducted a number of tests using the Aurora protocol. All the tests were performed using the Xilinx XUPV5-LX110T development board (see Appendix B), which features a Virtex 5 LX110T FPGA. The latter has 16 built-in MGTs, however, only 2 of these are terminated at SATA connectors. The MGTs are equipped with a high-quality variable differential clock source, which is independent of the board's system clock. This differential clock source can be set to 75MHz or 156.25 MHz to deliver Aurora speeds of 1.5 Gbps or 3.125 Gbps respectively. This clock source separation enables the data receive/send Aurora logic to be decoupled from the user logic. The board ships with a Xilinx SATA crossover cable, which we use to perform two tests: a loopback connection between the two SATA connectors on the same FPGA (i.e. Figure 6.6) and a two-board test by connecting two MGT transceivers on different XUPV5 boards (i.e. Figure 6.7).

In the first test, we generate a number sequence using a 16-bit counter, which we then send from one MGT to another MGT on the same FPGA using a full duplex Aurora channel. We then check the data received is in the expected order. If the number received does not match the expected number, we increase the error counter by one. We added ChipsScope ILA cores to our Aurora designs to monitor the send and receive data. Figure 6.8 shows the ModelSim simulation waveforms obtained using the VHDL Aurora simulation model provided by Xilinx. Figure 6.9 shows the FPGA Aurora waveforms collected using Xilinx's ChipScope LogicAnalyzer. We can see that Aurora channel has a latency of 38 clock cycles (difference between the X and O cursors in Figure 6.9) with

a constant throughput of 16 bit per clock cycle. Clock synchronisation did not occur as both MGTs are operated using the same clock. Clock synchronisation refers to the periodic transmission of special characters to prevent errors due to small clock frequency differences between the connected Aurora cores.

For the second test, a clock compensation module was required to prevent any potential clock differences as the channel partners sit on different boards and hence do not use the same clock source. Each Aurora core is accompanied by an optional clock compensation which can be enabled when required. management module. The counting sequence has been observed on both ends using ChipScope. The waveform extracted looked identical to the previous test, except from the fact the communication was interrupted by the clock compensation module for 2 clock cycles every 5000 clock cycle to send synchronisation characters.

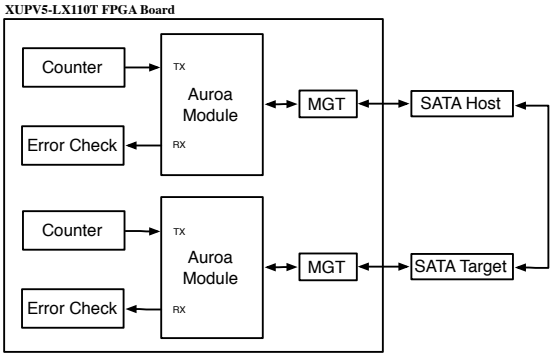


FIGURE 6.6: Single FPGA Board Aurora Loopback Test

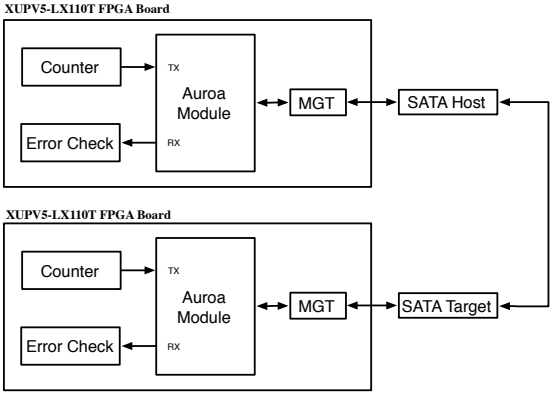


FIGURE 6.7: Two FPGA Boards Aurora Test

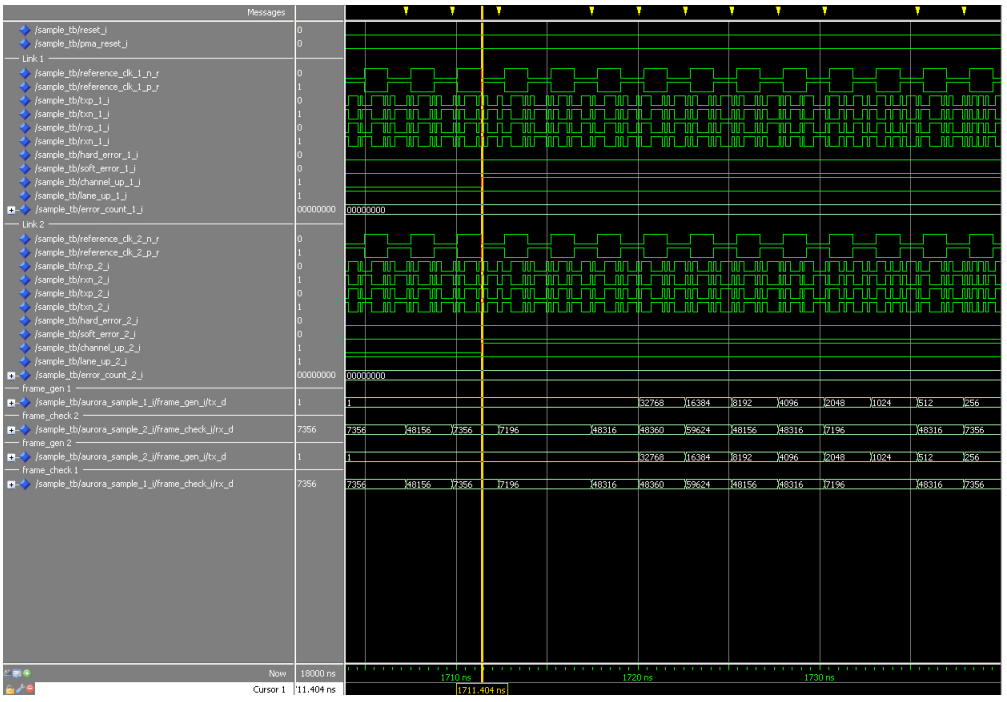


FIGURE 6.8: Aurora Loopback Test ModelSim Waveforms

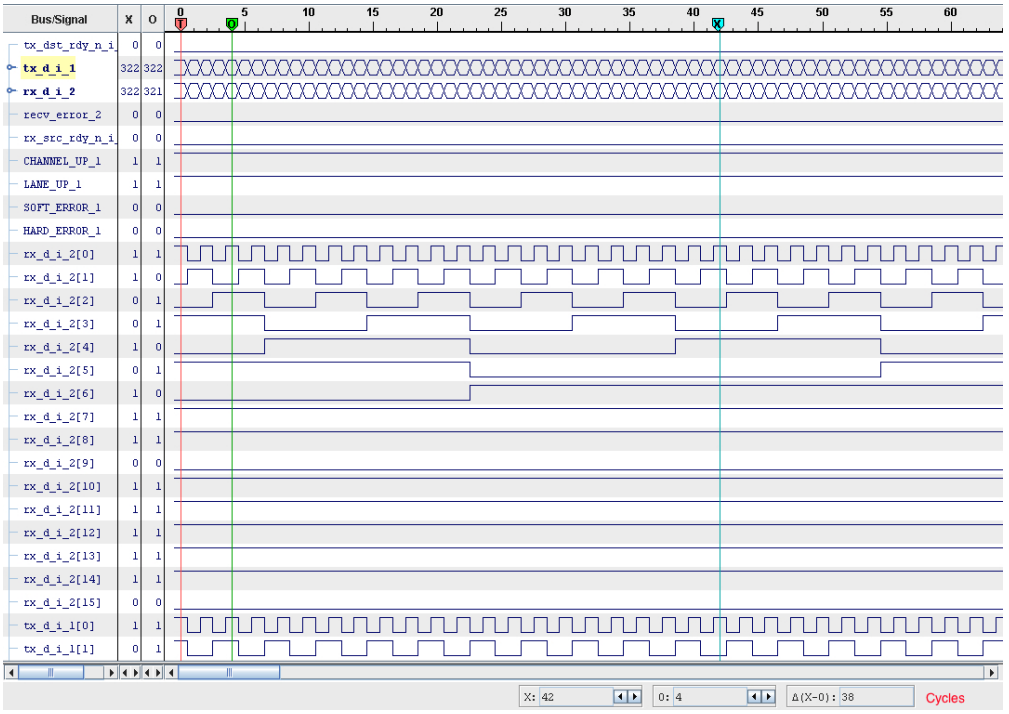


FIGURE 6.9: Aurora Loopback Test ChipScope Waveforms

6.4 Multi-FPGA LU Factorisation

In this section, we explain how the LU factorisation of a sparse matrix in the BDB form can be mapped to a multiple FPGA system. We also illustrate how we adapt our single-FPGA accelerator to a multi-FPGA system that performs LU factorisation in a SIMD fashion.

6.4.1 System Architecture

In order to have an SIMD-like architecture, all FPGAs should ideally perform the same computations on different datasets. Figure 6.10 shows the proposed multi-FPGA architecture that can used to capitalise on the features of the BDB form. In effect, the BDB sub-matrices are factorised using the FPGA nodes and their contributions are then sent to the root FPGA. The latter sums the node's contributions before it factorises the interface matrix. However, the parallel LU factorisation of circuit matrices in the BDB form involves irregular computation patterns and blocks of various sizes, as a result of the physical characteristics of the underlying circuit [79]. The higher the variance in block sizes, the larger will be the resulting FPGA idle times as the factorisation of interface matrix cannot proceed unless all other blocks have been already processed. To reduce the FPGAs idle time, we aim to overlap the intra-FPGA computations and the inter-FPGA communication such that contributions from the independent sub-matrices are sent back to the interface matrix as soon as they are computed. Additionally, we use lightweight multi-gigabit serial connections to minimise the inter-FPFA communication overheard.

The BDB sub-matrices can be factorised using the single FPGA sparse LU hardware we proposed in Chapter 5. The BDB form ensures that there is no communication between the sub-matrices, except when the output data needs to be sent back to the root FPGA to factorise the last block (i.e. interface matrix). Adapting our sparse LU hardware prototype to accommodate the coarse-grained parallelism exposed by the BDB form is straightforward. The distribution of the BDB matrix elements involves only the

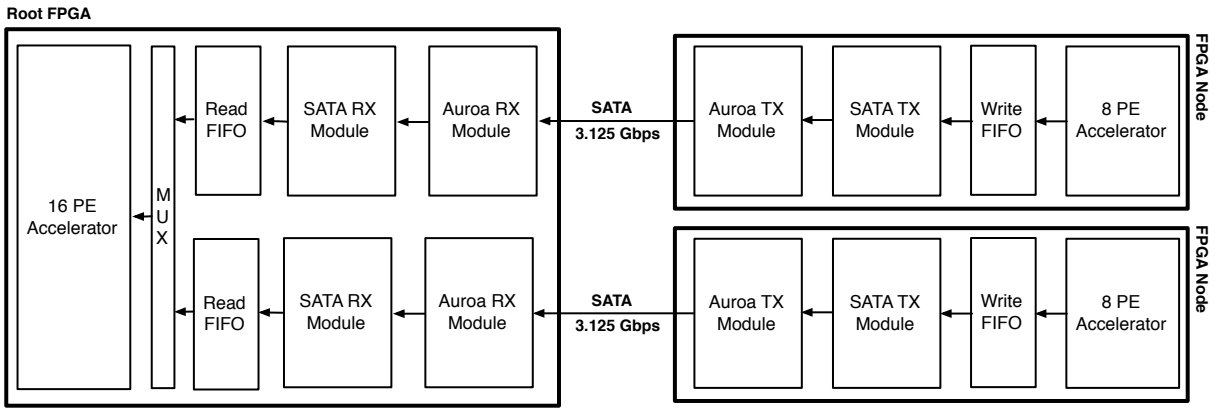


FIGURE 6.10: Architecture of the multi-FPGA Sparse LU Accelerator

distribution of data to their corresponding FPGAs processing nodes. The summation of contributions from the different FPGA nodes is accomplished using the accumulators already present within our single FPGA sparse LU hardware.

Each FPGA node hosts an 8-PE sparse LU accelerator, which is responsible for factorising a BDB sub-matrix. The root FPGA contains a sparse LU accelerator with 16 PEs to compensate for the fact that the last diagonal block gets denser as a result of summing of contributions from the FPGA nodes. Inter-FPGA communication is handled through Xilinx's Aurora protocol, which interfaces with FPGAs' internal MGTs. The FPGAs are interconnected via SATA links running at 3.125 Gbps (2.5 Gbps effective rate because of the 8b/10b encoding). The sparse LU hardware accelerators communicates with the Aurora interfaces through the our custom SATA Receive (RX) and Transmit (TX) modules, as illustrated in Figure 6.10. In effect, to increase concurrency, the Col_buffer and the Col_map units of the node accelerators have been altered to commit columns as they are computed to the 64-bit wide "Write FIFOs", as illustrated in Figure 6.11. On the other hand, the Col_buffer and the Col_map units of the root FPGA accelerator have been configured to read data from 64-bit wide "Read FIFOs", as illustrated in Figure 6.12. The Col_buffer contains the "current" factorised column while the Col_map unit contains the corresponding matrix indices.

Furthermore, our SATA TX and SATA RX modules contain a TX and RX FIFOs respectively. The RX and TX FIFOs create a buffered link between the read/write FIFOs

and the Aurora core. This buffered link is necessary because the hardware accelerator clock and the MGTs clock are independent on the XUPV5 board. The accelerator clocks data in and out of the read/write FIFOs at 150MHz (250MHz on Virtex 7) while the user logic clocks data in and out of the Aurora core at156.25 MHz (can be also set to 75 MHz). For this reason, we use RX and TX FIFOs with independent read and write clock inputs.

The TX and RX FIFOs are connected to the Aurora core through a multiplexer and demultiplexer respectively. The FIFOs have a width of 64 bits to utilise the complete Col_buffer/Col_map data width. We use an Aurora core with a 16 bit wide interface because the MGTs are optimised for a width of 2 words, that is 16/20 bits using 8B/10B encoding. The MUX and DEMUX are needed to connect the 64 bit FIFO interface to the 16 bit Aurora core interface. The MUX converts each 64 bit word from the TX FIFO into groups of 16 bits spread over 4 clock cycles. The DEMUX buffers 4 x 16 bit words from the Aurora core into one 64 bit entry to the RX FIFO. This configuration allows us to make full use of the hardware accelerator throughtput.

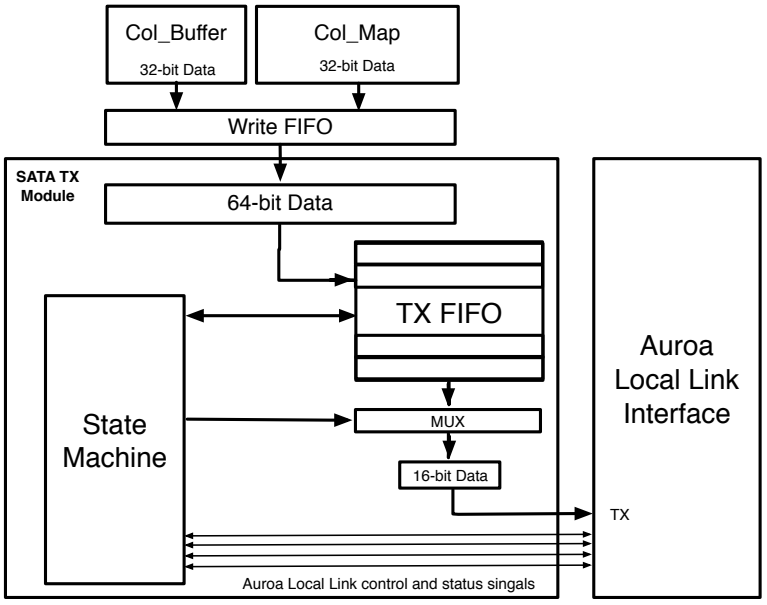


FIGURE 6.11: Architecture of the SATA TX Module

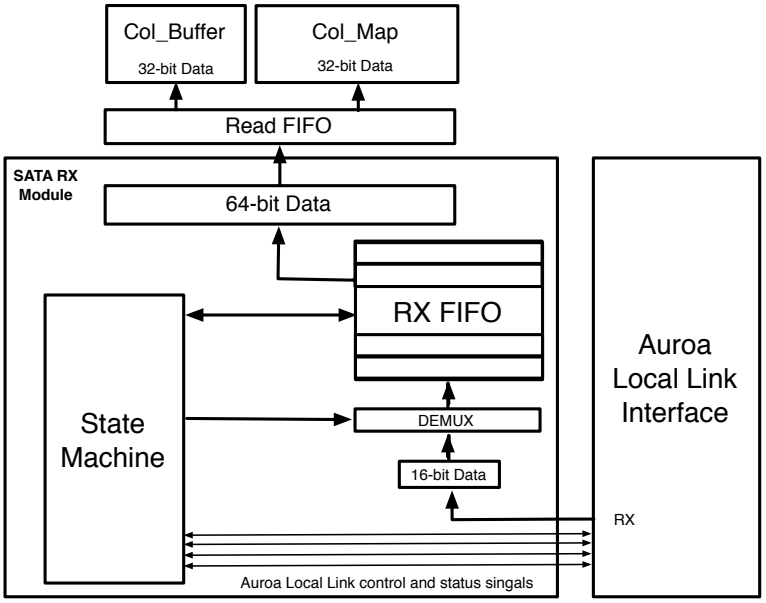


FIGURE 6.12: Architecture of the SATA RX Module

6.4.2 Experimental Setup

We build a prototype of our multi-FPGA accelerator using three Xilinx XUPV5 development boards (Appendix B) interconnected by Xilinx's SATA crossover cables according to the topology depicted in Figure 6.10. We use Xilinx's FIFO Generator [217] to implement TX, RX, Read, and Write FIFOs. We also use the Xilinx's CoreGen to instantiate the required Aurora modules. We use Synplify Pro 9 and Xilinx ISE 10.1 to synthesis and implement the different components of our multi-FPGA prototype. We set the MGT clocks to 156.25 MHz in order to deliver inter-FPGA link speeds of 3.125 Gbps.

We use the MESHPART toolbox [218], which in turn uses METIS graph-partitioning packages [207] to partition our test matrices. The toolbox contains several graph and mesh partitioning routines to generate recursive multiway partitions, vertex separators, and nested dissection orderings. Using MESHPART's nested dissection routine (i.e. *metisnd*), we partition our test matrices into almost two equal-sized partitions with the view to organise them in the BDB form shown in Figure 6.13. We assign the resulting two 3-block groups (i.e. [A3, A11, A13] and [A32, A22, A23]) to the node FPGAs, while the interface matrix (i.e. A33) is assigned to the root FPGA.

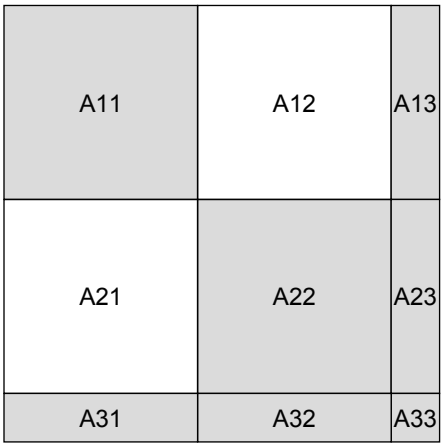


FIGURE 6.13: The Targeted BDB Matrix Form

6.4.3 Performance Analysis

In this section, we compare the acceleration ratios achieved using our multi-FPGA accelerator prototype with speedup ratios obtained using a 16-PE single-FPGA system and reported in Table 5.3. To gauge the time taken by each node of our multi-FPGA system, we use Xilinx's ChipScope ILA cores to count the number of clock cycles taken by each FPGA. We calculate the overall LU factorisation time as follows:

$$T_F(A) = \max_{i=1} T_F(A_{ii}) + \max_{i=1} T_{tx}(B_{ii}) + T_F(A_{33}) \tag{6.2}$$

Where $T_F(A)$ is the total factorisation time for matrix the A, $T_F(A_{ii})$ is the time taken to factorise the diagonal block A_{ii}, $T_{tx}(B_{ii})$ is the time taken to send the border contributions (associated with the diagonal block A_{ii}) back to the main FPGA, and $T_F(A_{33})$ is the time taken to factorise the interface matrix including the time taken to sum contributions from FPGA nodes. We compare the LU factorisation time achieved with the KLU runtimes reported in Table 5.2. The speedups achieved are illustrated in Figure 6.14. We can see that our multi-FPGA prototype achieves acceleration ratios between 3.5-38× (17× on average). Figure 6.15 shows the relatives speed achieved using our 3-FPGA system compared to the speedups reported in Table 5.3 for our 16-PE single-FPGA system over KLU. We note that our 3-FPGA system is on average 1.9× faster than the single-FPGA system. However, the mutli-FPGA system under-performed on a couple

of occasion (i.e. fpga_dcop_50 and fpga_trans_01). This mainly due to the fact that the time needed to send back the contributions from BDB sub-matrices of these two matrices is relatively high when compared with the time needed to factorise them.

From Figure 6.2, it is also clear that in order to reduce the parallel factorisation time for a BBD matrix, one would like to reduce the size of the diagonal blocks as well as the border (as it impacts the size of the interface matrix). Such requirements are conflicting as reducing the size of the diagonal blocks, thereby increasing their number, may cause a corresponding increase in the size of the border. To achieve an effective trade-off between these two conflicting requirements, a nested BDB form is often employed as shown in Figure 6.16 and Figure 6.17. Such processing tree can be easily mapped to a multiple FPGA system following the same topology.

6.5 Summary

In this chapter, we have demonstrated the strategy we followed to partition sparse matrices into smaller pseudo-independent entities that can be spread over a number of FPGA for a coarse-grain parallel LU factorisation. We have also provided details of how our single-FPGA design can be adapted to a multi-FPGA LU factorisation system to harness the coarse-grain parallelism exposed by the BDB form. We have empirically illustrated our prototype's ability to accelerate certain circuit matrices up to 38 times over KLU running on a 6-core Intel Xeon 2.6 GHz processor with 6 GB RAM.

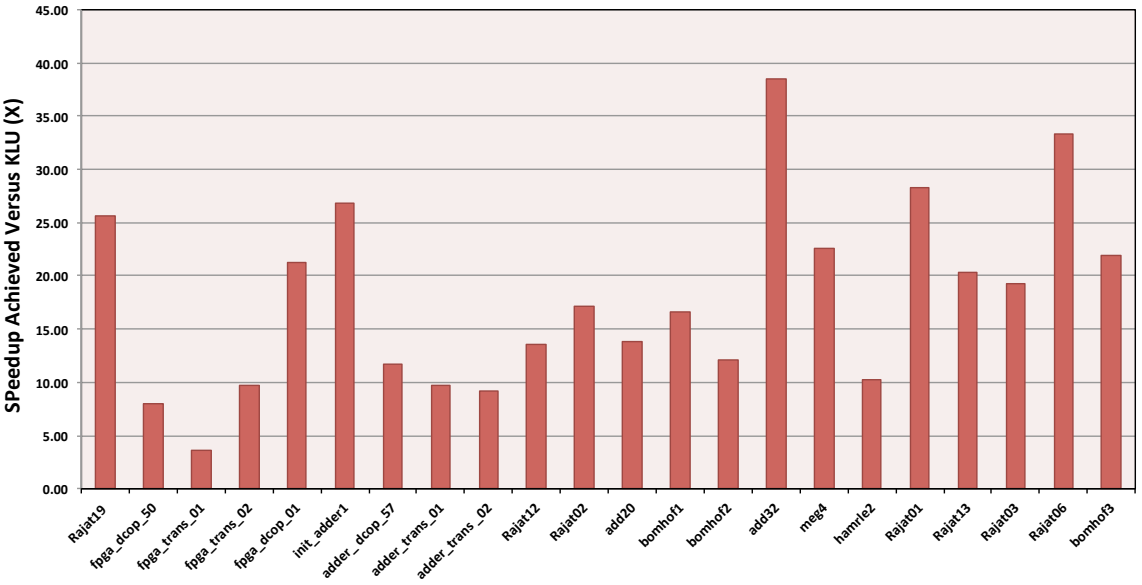


FIGURE 6.14: Multi-FPGA LU Decomposition Accelerator Performance Versus KLU

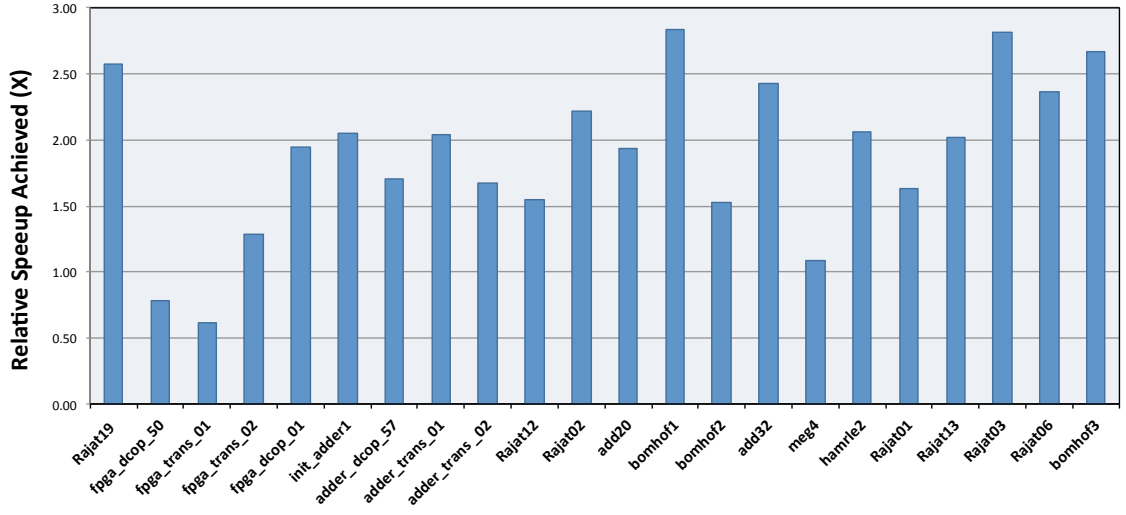


FIGURE 6.15: Multi-FPGA LU Decomposition Accelerator Performance Relative to a 16-PE single-FPGA Accelerator

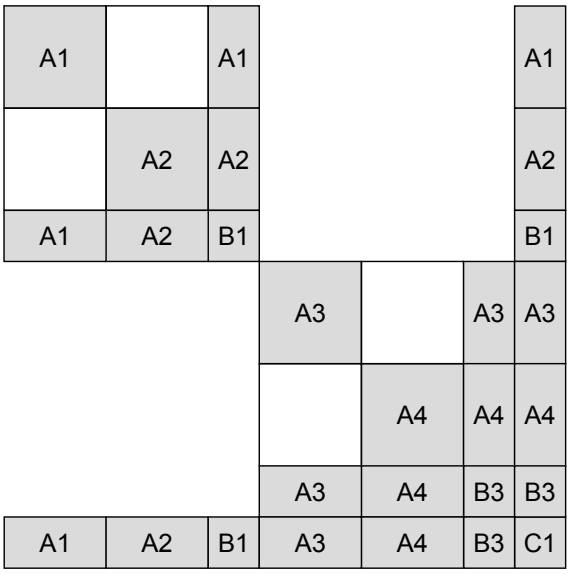


FIGURE 6.16: Two-level Nested BDB Form

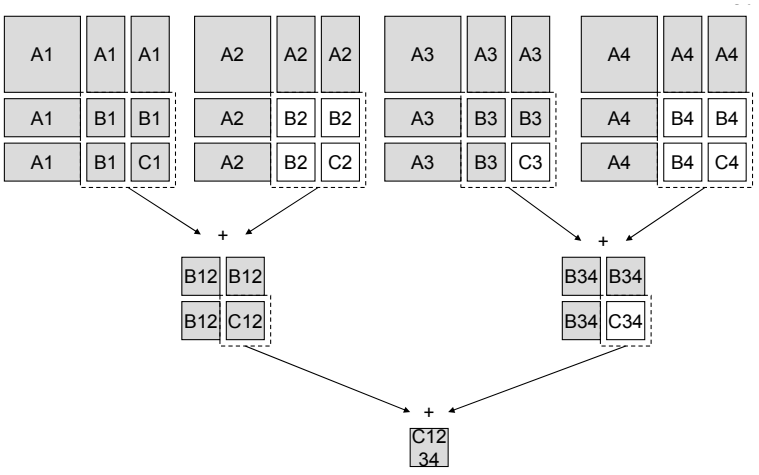


FIGURE 6.17: Two-level Nested BDB Processing Tree

Chapter 7

Conclusion and Future Works

This thesis provided a proof of concept that FPGAs, in conjunction with the appropriate algorithms, can be leveraged to implement a tailored hardware solution able to accelerate the LU decomposition of circuit matrices by many times. In this chapter, we reflect on the achieved objectives and findings. We then conclude with possible follow-up work and further research directions.

7.1 Conclusion

In this thesis, we covered the parallelisation of circuit matrices' LU factorisation on FPGAs using a bottom-up methodology. First, we demonstrated the importance of accelerating matrix calculations for SPICE simulations, in order to keep up with increasing VLSI circuit densities. We also established that general-purpose PCs are inadequately designed and ill-equipped to cope with the irregularity of computations associated with the LU factorisation of the highly sparse circuit matrices. Moreover, we argued that current parallel programming tools, based on the conventional multi-threading model, are inherently inefficient as they were historically developed for sequential machines.

Secondly, we empirically analysed the properties of circuit matrices in order to identify the features that may facilitate or complicate the design of the application-specific hardware accelerator. We found that circuit matrices are highly sparse and roughly structurally symmetric. A hardware design can benefit from sparsity by avoiding performing computing on the zero elements and hence speedup the solution process. However, we identified two phenomenon, namely pivoting and fill-in, that adversely affect sparsity and thus degrade performance.

Subsequently, we studied several algorithms that can be used to derive LU factorisation and explored their impact on the computations and data access patterns. Moreover, we investigated the different ordering techniques that can be used to maintain sparsity and enhance parallelism. For sparse LU decomposition, the choices of the particular algorithm used, e.g. right-looking or left-looking, matrix ordering, and matrix data representation scheme, can significantly impact the hardware design.

We finally presented a prototype implementation of the sparse matrix solver hardware we designed and optimised for execution on a single FPGA node. The hardware was designed such that it is able to evaluate independent columns (i.e. medium-grained parallelism) without overlooking the finer-grained parallelism when performing scalar operations within a particular column. Therefore, we demonstrated how static pivoting and symbolic analysis can be utilised to create an accurate task-flow execution graph, which efficiently exploits parallelism at multiple granularities and sustains high floating-point data rates. Experimental results showed average speedups of 9.65×, 11.83×, 17.21× against UMFPACK, KLU, and Kundert Sparse matrix packages respectively. We also detailed the approach we used to adapt our sparse LU hardware prototype from a single-FPGA architecture to a multi-FPGA system to achieve higher acceleration ratios, up to 38× for certain circuit matrices.

To summarise, in this thesis, we presented the following key contributions:

- We presented an empirical analysis for the SPICE runtime and the type of matrices that typically arise in circuit simulations. We studied the total SPICE execution time and we demonstrated that the runtime scales as $O(N^{1.3})$ as the circuit size

increases. We also studied the scaling trends of the two key components of SPICE, i.e. the model evaluation and matrix solution phases, in terms of complexity, execution time, and parallelism potential. We found that the model evaluation phase scales as $O(N^{1.1})$ as the circuit size increases, compared to $O(N^{1.4})$ for the matrix solution phase. We have also detailed our methodology to evaluate circuit matrices and algorithm properties useful in the design of an FPGA hardware accelerator

- We illustrated how we leveraged the Gilbert-Peirels (G/P) symbolic analysis, in conjunction with predicted nonzero pattern, to create a column-dependency driven task graph that maximises the parallelism potential for the LU matrix factorisation of sparse matrices. As such, we have introduced our Dependency-Aware Matrix Operations Scheduling (DAMOS) pre-processing stage. We employed the latter to generate parallel operations schedule used to parallelise and control the dataflow of G/P LU matrix operations on the FPGA.

- We provided detailed analysis of the algorithms used in our experiments and we empirically demonstrated that pre-ordering matrices for sparsity not only reduces the overall FLOP count but also distributes the computational efforts more evenly between columns of a given matrix, making more suitable for a distributed computing architecture.

- We presented an implementation of a sparse direct LU decomposition hardware on FPGAs geared towards matrices that arise in SPICE circuit simulations and optimised for execution on a single FPGA. We evaluated the performance of our design against some of the stat-of-the-art sparse matrix packages such as UMFPACK, Kundert Sparse, and KLU. We gauged the operational performance of the Sparse LU Hardware using a Xilinx Virtex 5 LX110T FPGA and we then extrapolated the results to the more recent XC7V200T Virtex 7 FPGA. We also studied the effect of matrix sparsity on the performance of our hardware design. We showed that our 16-PE design configuration outperforms KLU running on a 2.67 GHz 6-core 12-thread Intel Xeon X5650 microprocessor by an average of 9.65 times using a Virtex 5 FPGA.

- We demonstrated how we adapt our sparse LU hardware prototype from a single-FPGA architecture to a multi-FPGA system. As such, we illustrated how we leverage the FPGAs internal Multi-Gigabit Transceivers (MGTs) to link several FPGAs. Therefore, we showed the design changes necessary to minimise the inter-FPGA communication and ensure that acceleration scales accordingly. The multi-FPGA system accelerated certain circuit matrices up to 38 times when compared a commodity CPU solution.

- We illustrated how we extract parallelism at different granularities to accelerate the matrix solution process: fine-grained parallelism at the scalar level using a dataflow graph, medium-grained parallelism with the aid of a tree-like execution flow graph to evaluate independent columns, and coarse-grained parallelism using nested dissection.

7.2 Future Work

In relation to the topics covered in this thesis, there are a number of points that can be further researched:

Algorithms: As previously mentioned, we rely on the multi-level graph partitioning from METIS to produce the BDB structures used in our work. METIS employs a divide and conquer approach to recursively bisect the symmetrised graph of the input matrix. This works well in practice as circuit matrices are roughly structurally symmetric and thus symmetrisation doesn't create too many false connections (i.e. dependencies). However, to optimise the design further, we propose to use Hypergraph-based Unsymmetric Nested Dissection ordering algorithm (HUND) [209] to produce the nested BDB form. The major advantage of HUND over previous methods is that it produces orderings of consistently high quality using the structure of the original matrix without the need of symmetrisation. The HUND Algorithm itself can be run in parallel, significantly reducing the time required for the pre-conditioning phase of a matrix and currently work is in progress to

include parallel implementation of HUND in the Zoltan parallel applications package [219]. Hypergaph partitioning has been shown to reduce communication by 30% to 38% over conventional graph partitioning and to provide a more accurate communication model [208].

Hardware: In the development boards we used for this research, there are three performance limiting factors: the inter-FPGA communication bandwidth, the embedded memory size, and the speed of the external memory. In effect, the Virtex 5 FPGA used features 16 MGTs, each able to achieve a speed of 3 Gbps (over 10 Gpbs in the newer FPGA offerings). A number MGTs can be aggregated together using the Aurora protocol to form a faster communication channel. However, only two of MGTs present in the FPGA are brought forward to SATA connectors, hence liming the benefits of using MGTs in a multi-FPGA system context. Furthermore, great portions of the FPGA's embedded memory is utilised to buffer data from external memory in order to hide the relatively longer latencies associated with accessing data in the external DRAM. These limitations can be over overcome by designing a custom multi-FPGA board with rich SATA/MGT ports to enable a variety of topologies, and multiple external memory banks to enable the optimisation of the data layout for concurrency.

Integration: Finally, the design we proposed can be integrated with a software solution to streamline and automate the overall solution process. Provided that a fast memory exist, a driver can be written to memory map the internal BRAMs of the PEs to the external DRAM accordingly and hence provide a seamless integration between the software and the hardware accelerator.

Appendix A

Left-looking LU Factorisation

A.1 Solving Triangular Systems

A triangular matrix is a matrix where all the entries either below or above the main diagonal are zero. If all the elements above its main diagonal are zeros, the matrix is called a "lower triangular matrix" and it is usually denoted by **L**. Conversely, an "upper triangular matrix" is a matrix where all the elements below the main diagonal are zeros and it usually denoted by **U**. A template for **L** and **U** is shown in A.1.

$$\mathbf{L} = \begin{bmatrix} l_{11} & 0 & 0 & \dots & 0 \\ l_{21} & l_{22} & 0 & \dots & 0 \\ l_{31} & l_{32} & \ddots & \ddots & 0 \\ \vdots & \vdots & \ddots & \ddots & 0 \\ l_{n1} & l_{n2} & \dots & l_{nn-1} & l_{nn} \end{bmatrix}, \qquad \mathbf{U} = \begin{bmatrix} u_{11} & u_{12} & u_{13} & \dots & u_{1n} \\ 0 & u_{22} & u_{23} & \dots & u_{2n} \\ 0 & 0 & \ddots & \ddots & \vdots \\ 0 & \vdots & \ddots & \ddots & u_{n-1n} \\ 0 & \dots & 0 & 0 & u_{nn} \end{bmatrix} \tag{A.1}$$

Solving a matrix equation in the form $\mathbf{Ly} = \mathbf{b}$ or $\mathbf{Ux} = \mathbf{b}$ is relatively straightforward as it does not require inverting the matrix. In effect, it is done by the means of an iterative process known as "forward substitution" for lower triangular matrices and as "back substitution" for upper triangular matrices. In forward substitution, $(x_1 = b_1/l_{11})$

is computed first, then the answer is substituted forward into the next equation to solve for x_2, which in turn is substituted forward into the equation of x_3 and so forth until x_n is solved. Forward substitution can be summarised in Algorithm A.1. In back substitution, a similar process is followed with the minor difference that y_n is computed first and then substituted back into the previous equation to solve for y_{n-1}, and so on until y_1 is calculated.

Algorithm A.1 Forward substitution

```
1: x = b
2: for j = 1 to n do
3:     x_j = x_j / l_jj
4:     for each i > j for which l_ij ≠ 0 do
5:         x_i = x_i − l_ij x_j
6:     end for
7: end for
```

A.2 Gaussian Elimination

Gaussian Elimination (GE) is a process, named after the German mathematician Carl Friedrich Gauss [182], that capitalises on the ease of solving triangular linear systems. GE solves a nonsingular system of linear equations in two steps. Firstly, using a sequence of elementary row operations, a matrix is reduced to an upper triangular matrix and it is known as the "forward elimination" step. The second step consists in solving the new triangularised matrix by the means of "back substitution".

To illustrate the process just described, consider solving the following nonsingular linear system of n equations for n unknowns :

$$\begin{cases} a_{11}x_1 + a_{12}x_2 + a_{13}x_3 + \cdots + a_{1n}x_n = b_1 \\ a_{21}x_1 + a_{22}x_2 + a_{23}x_3 + \cdots + a_{2n}x_n = b_2 \\ a_{31}x_1 + a_{32}x_2 + a_{33}x_3 + \cdots + a_{3n}x_n = b_3 \\ \vdots \\ a_{n1}x_1 + a_{n2}x_2 + a_{n3}x_3 + \cdots + a_{nn}x_n = b_n \end{cases} \tag{A.2}$$

The set of equations in A.2 can written more elegantly in the matrix form $Ax = b$ as follows:

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} & \cdots & a_{1n} \\ a_{21} & a_{22} & a_{23} & \cdots & a_{2n} \\ a_{31} & a_{32} & a_{33} & \cdots & a_{3n} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & a_{3n} & \cdots & a_{nn} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ \vdots \\ x_n \end{bmatrix} = \begin{bmatrix} b_1 \\ b_2 \\ b_3 \\ \vdots \\ b_n \end{bmatrix} \tag{A.3}$$

In the first step of gaussian elimination, the first row of A.3 is multiplied by $-\frac{a_{21}}{a_{11}}$ and added to the second equation to eliminate x_1 from it. Then, the first row is again multiplied by $-\frac{a_{31}}{a_{11}}$ and added to the third row. The same process repeated for the remaining equations. Hence, once step one finishes, x_1 is eliminated from the second through the n^{th} equations and thus A.3 becomes:

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} & \cdots & a_{1n} \\ 0 & a_{22}^{(1)} & a_{23}^{(1)} & \cdots & a_{2n}^{(1)} \\ 0 & a_{32}^{(1)} & a_{33^{(1)}} & \cdots & a_{3n}^{(1)} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & a_{n2}^{(1)} & a_{3n}^{(1)} & \cdots & a_{nn}^{(1)} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ \vdots \\ x_n \end{bmatrix} = \begin{bmatrix} b_1 \\ b_2^{(1)} \\ b_3^{(1)} \\ \vdots \\ b_n^{(1)} \end{bmatrix} \tag{A.4}$$

where $a_{22}^{(1)} = a_{22} - \frac{a_{21}}{a_{11}} \times a_{12}$, $a_{32}^{(1)} = a_{32} - \frac{a_{32}}{a_{11}} \times a_{12}$, $\cdots$, $a_{n2}^{(1)} = a_{n2} - \frac{a_{n1}}{a_{11}} \times a_{12}$ and so forth. Variables x_2, x_3, $\ldots$, and x_{n-1} are eliminated in the same fashion as x_1. The superscripts here denote the step of the elimination. Therefore, after $(n-1)$ elimination steps, the matrix A is transformed to an upper triangular matrix, as shown in A.5, which easily solvable by the means of back substitution.

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} & \cdots & a_{1n} \\ 0 & a_{22}^{(1)} & a_{23}^{(1)} & \cdots & a_{2n}^{(1)} \\ 0 & 0 & a_{33^{(2)}} & \cdots & a_{3n}^{(2)} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & \cdots & a_{nn}^{(n-1)} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ \vdots \\ x_n \end{bmatrix} = \begin{bmatrix} b_1 \\ b_2^{(1)} \\ b_3^{(2)} \\ \vdots \\ b_n^{(n-1)} \end{bmatrix} \tag{A.5}$$

Since the multipliers are chosen so that entries below the main diagonal are calculated to be zero, those entries should be assigned to zeros rather than computed. This would save $(n-k)$ subtractions at every k^{th} elimination step. The Gaussian elimination process just described can be summarised in Algorithm A.2.

Algorithm A.2 Gaussian elimination

```
1: for k = 1 to n − 1 do
2:    for i = k + 1 to n do
3:       m_ik = a_ik / a_kk
4:       a_ik = 0
5:       for j = k + 1 to n do
6:          a_ij = a_ij − (m_ik × a_kj)
7:       end for
8:    end for
9: end for
```

It is clear that Algorithm A.2 will halt if it encounters a diagonal element (e.g a_{kk}) that is a zero. It should be also noted that GE is prone to numerical inaccuracies if elements on the diagonal are very small as it will cause the multipliers (e.g. m_{ik}) to grow towards infinity or amplify round-off errors. To remedy this pitfall, rows can be interchanged at every elimination step to ensure that the element on the main diagonal is bigger than the elements below it. The process of switching rows is known as partial pivoting. Generally speaking, gaussian elimination with partial pivoting (GEPP) is considered to be numerically stable, even though there are examples for which it is unstable [220].

In terms of numerical effort, Gaussian elimination factorises a system of n equations for n unknowns in roughly $\left(\frac{2}{3}n^3\right)$ operations, and consequently has a complexity of $\mathcal{O}(n^3)$. Back substituation requires (n^2) operations and hence has a complexity of $\mathcal{O}(n^2)$.

Figure A.1 shows the data access pattern for the Gaussian elimination algorithm at the k^{th} step in which nonzero subdiagonal elements in column k are eliminated by subtracting appropriate multiples of the k^{th} (pivot) row.

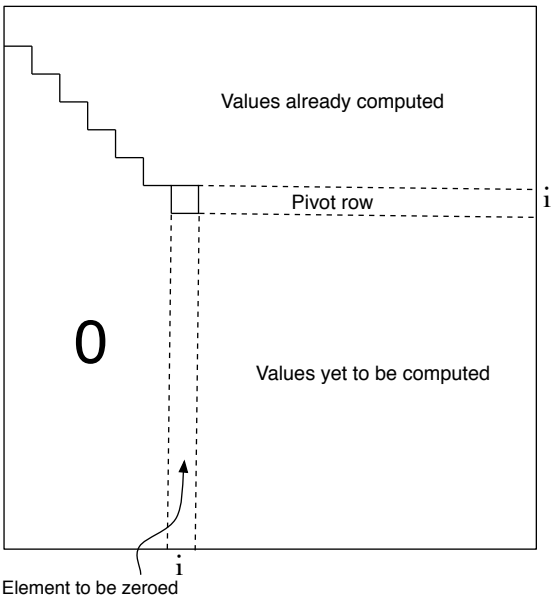


FIGURE A.1: Gaussian Elimination Data Access and Computation Pattern

A.3 Left-looking LU Decomposition

Gaussian elimination is called a right-looking (or submatrix-based) algorithm as in the k^{th} computation step, the columns below and to the right of the k^{th} column of A as accessed and subsequently modified, as shown in Figure A.1. These algorithms are unsuitable if the matrix A is stored column-wise. A left-looking LU factorisation algorithm, however, computes L and U one column at a time. At the k^{th} step, it accesses columns 1 to $(k-1)$ of L and column k of A. Thus, this category is also known as column-based methods. For illustrative purposes consider A.6 where the matrix L is assumed to have a unit diagonal.

$$\begin{bmatrix} L_{11} & & \\ l_{21} & 1 & \\ L_{31} & l_{23} & L_{33} \end{bmatrix} \begin{bmatrix} U_{11} & u_{12} & U_{13} \\ & u_{22} & u_{23} \\ & & U_{33} \end{bmatrix} = \begin{bmatrix} A_{11} & a_{12} & A_{13} \\ a_{21} & a_{22} & a_{23} \\ A_{31} & a_{23} & A_{33} \end{bmatrix} \tag{A.6}$$

The following set of equation can be derived from A.6:

$$L_{11}u_{12} = a_{12} \tag{A.7}$$

$$l_{21}u_{12} + u_{22} = a_{22} \tag{A.8}$$

$$L_{31}u_{12} + l_{32}u_{22} = a_{32} \tag{A.9}$$

However, assuming we have already computed L_{11} , l_{21} and L_{31}, equations A.7, A.8 and A.9 can be written in the form of $Lx = b$ as follows:

$$\begin{bmatrix} L_{11} & & \\ l_{21} & 1 & \\ L_{31} & 0 & I \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} a_{11} \\ a_{21} \\ a_{31} \end{bmatrix} \tag{A.10}$$

The solution to this system gives $u_{12} = x_1$, $u_{22} = x_2$, and $l_{32} = x_3/u22$ and hence effectively computing the second column of L and U using only columns to the left of current pivot column. This mechanism of computing column k of L and U by solving a lower triangular system $Lx = b$ is the key step in a left-looking factorisation algorithm. The algorithm just described does not take sparsity or pivoting into account.

Appendix B

Xilinx XUPV5-LX110T Development Board

The XUPV5-LX110T development board provides an advanced hardware platform that consists of a high performance Virtex-5 LX110T FPGA surrounded by a comprehensive collection of peripheral components, as shown in Figure B.1. The various peripherals inlcude a 256MB DDR2 memory, SATA connectors, RS232 port. The board also features SMA and SATA connectors which can be linked to the FPGA's internal Multi-Gigabit Transceivers (MGTs). These connectors can be then used to connect multiple boards, either as part of processing chain or to be aggregated into a "super FPGA" tackling a particular task.

The featured FPGA has, but not limited to, 110,952 logic cells, 64 DSP48E slices, and 148 of 36Kb Block Rams. The FPGA also has 16 MGTs but only of 5 of these are brought out to physical connectors. Only 2 of these are terminated at SATA connectors whilst the the the other three terminate at user-supplied Sub-Miniature A (SMA) connectors. The MGTs are equipped with a high-quality variable differential clock source (75 or 150 MHz) which is independent of the system clock. This enables the data receive/send logic to be decoupled from the user logic.

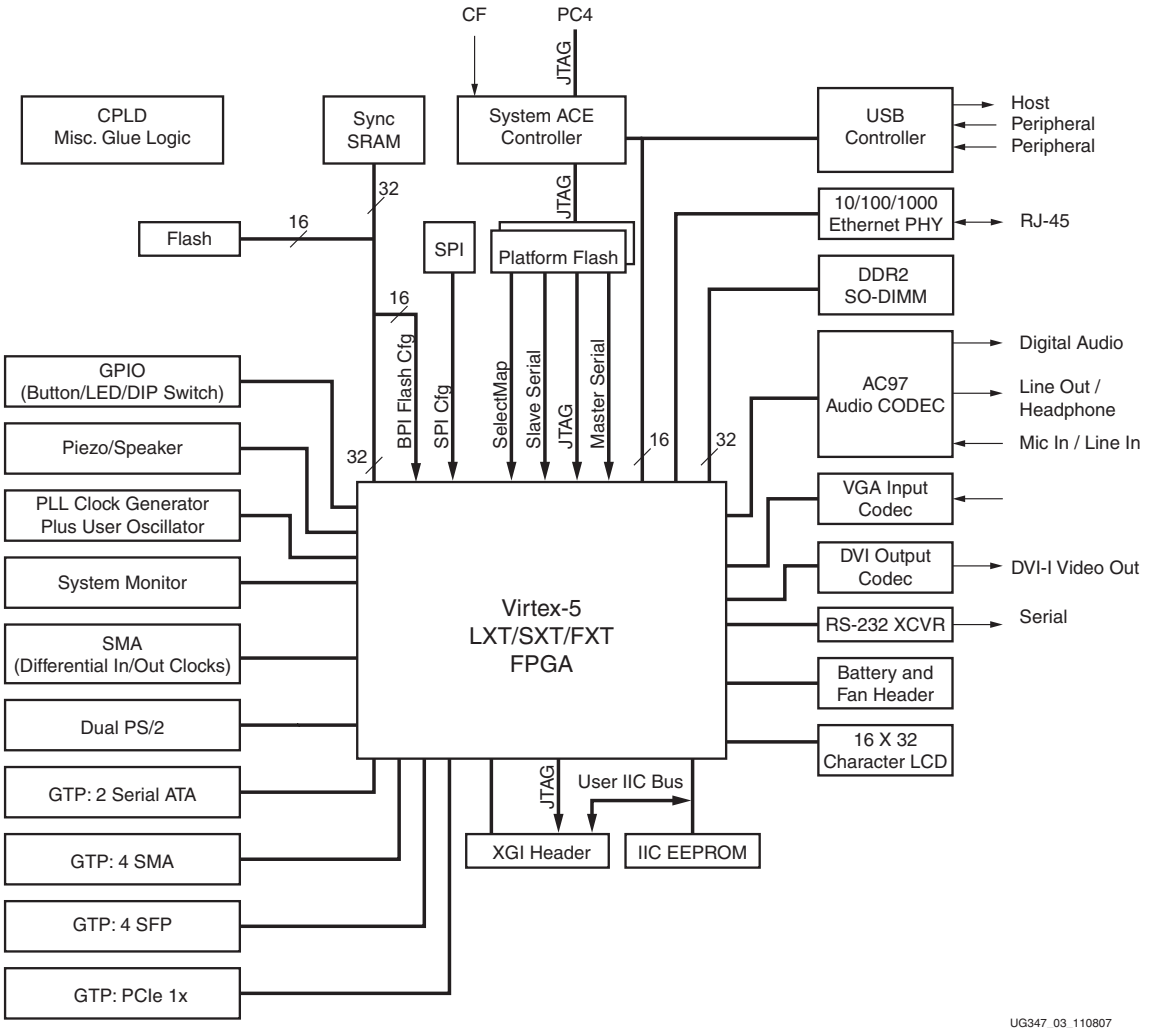


FIGURE B.1: XUPV5 Development Board Block Diagram

SATA can also be used as a convenient and low cost medium for connecting 2 or more FPGA development boards. The SATA physical interface can carry signals up to 3 Gb/s for general-purpose usage. The board ships with a special Xilinx SATA crossover cable that is used as a loopback connection between the two SATA host connectors for loopback testing and bit error rate testing (BERT). The SATA crossover cable can also be used to connect to two boards or more.

1 Virtex-5 FPGA LX110T

2 256 MB SODIMM DDR2 SODIMM

3 Differential Clock Input and Output with SMA Connectors

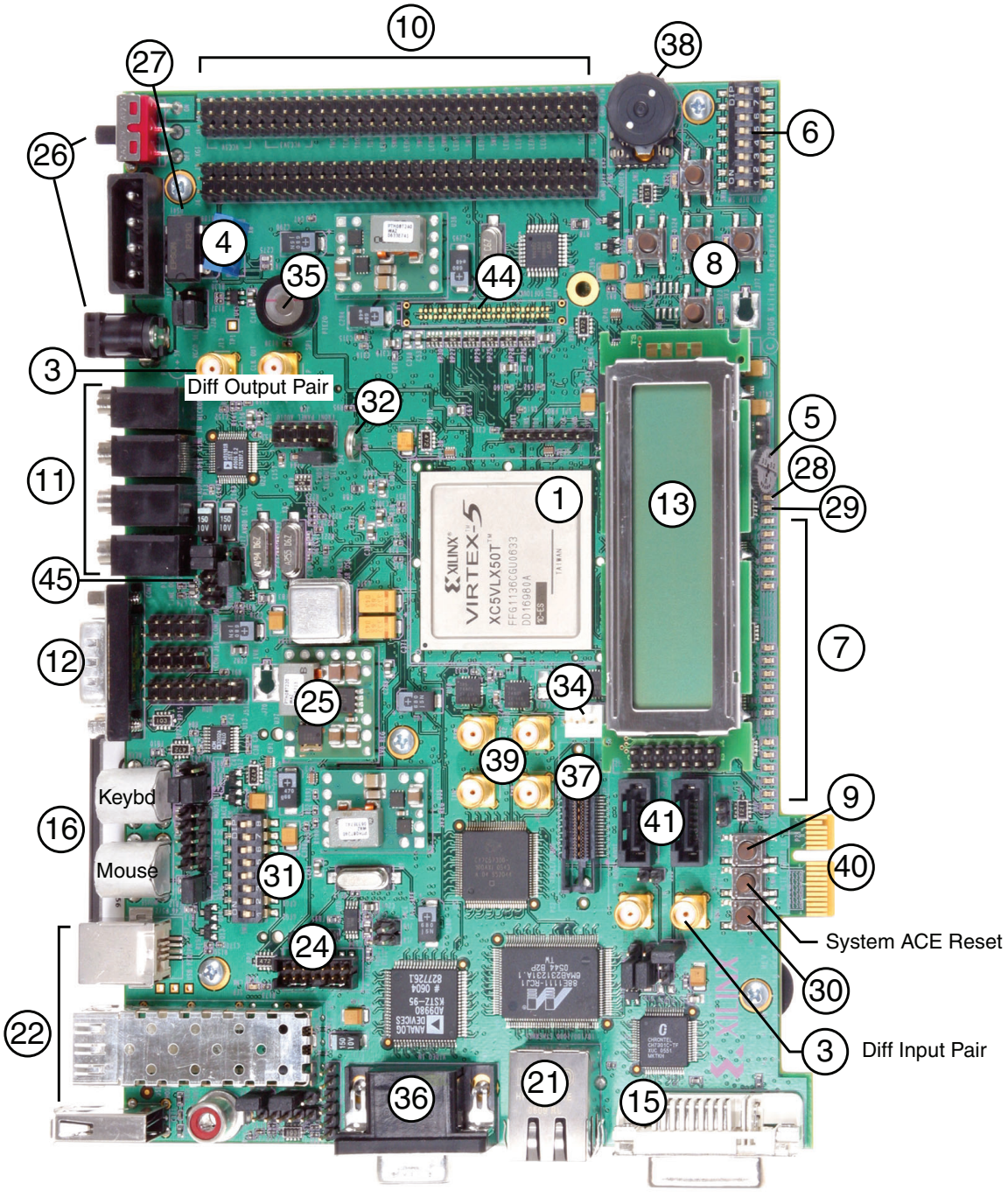


FIGURE B.2: Detailed Description of XUPV5-LX110T Components: (Front)

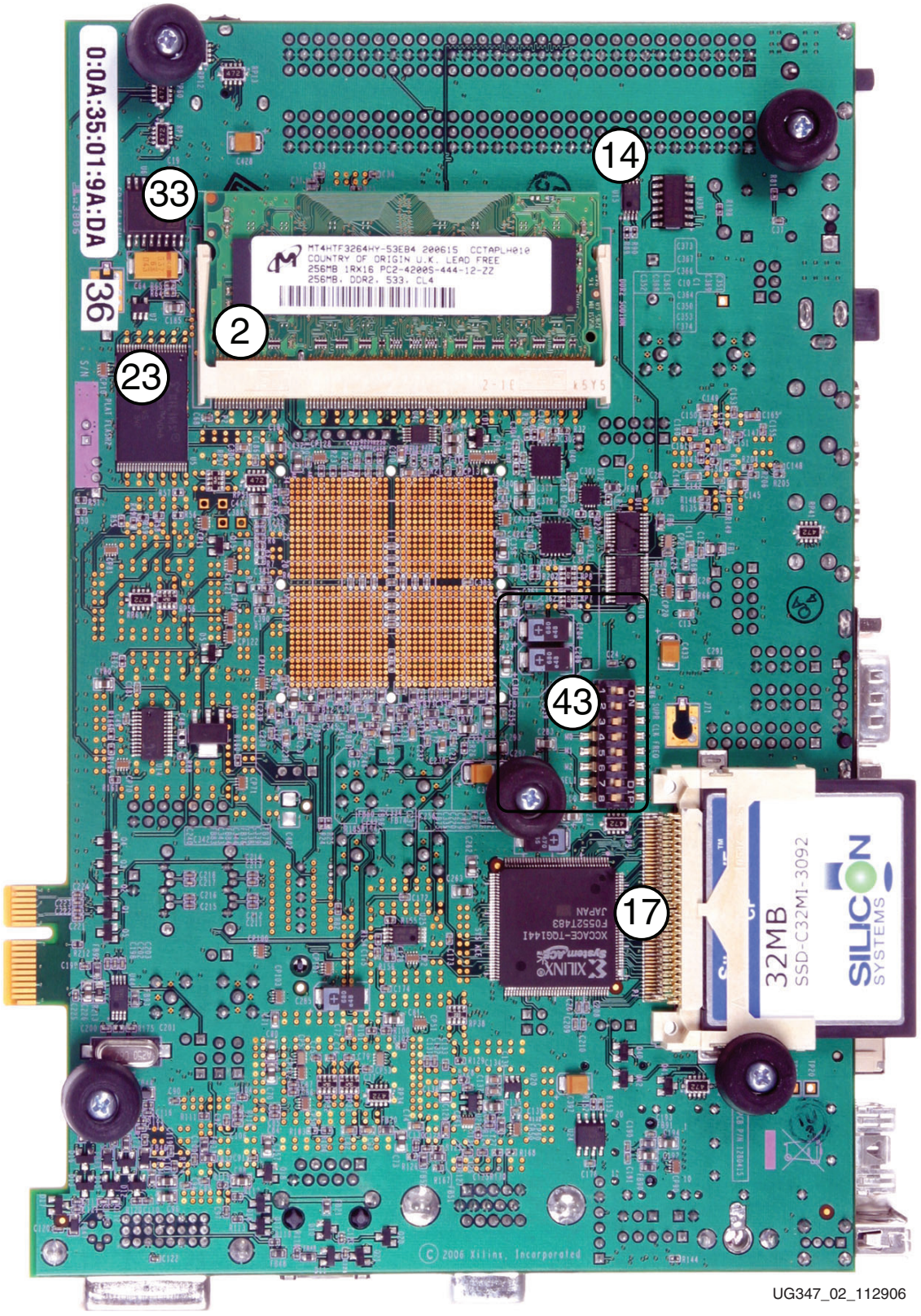


FIGURE B.3: Detailed Description of XUPV5-LX110T Components: (Back)

4 Oscillators

5 LCD Brightness and Contrast Adjustment

6 GPIO DIP Switches (Active-High)

7 User and Error LEDs (Active-High)

8 User Pushbuttons (Active-High)

9 CPU Reset Button (Active-Low)

10 XGI Expansion Headers

11 Stereo AC97 Audio Codec

12 RS-232 Serial Port

13 16-Character x 2-Line LCD

14 IIC Bus with 8-Kb EEPROM

15 DVI Connector

16 PS/2 Mouse and Keyboard Ports

17 System ACE and CompactFlash Connector

18 ZBT Synchronous SRAM

19 Linear Flash Chips

20 Xilinx XC95144XL CPLD

21 10/100/1000 Tri-Speed Ethernet PHY

22 USB Controller with Host and Peripheral Ports

23 Xilinx XCF32P Platform Flash PROM Configuration Storage Devices

24 JTAG Configuration Port

25 Onboard Power Supplies

26 AC Adapter and Input Power SwitchJack

27 Power Indicator LE The PWR Good LED lights when the 5V supply is applied

28 DONE LED lighted when the FPGA is successfully configured

29 INIT LED: lights upon power-up to indicate that the FPGA has successfully powered up and completed its internal power-on process

30 Program Switch: This switch grounds the FPGA's Prog pin when pressed. This action clears the FPGA

31 Configuration Address and Mode DIP Switches

32 Encryption Key Battery used to hold the encryption key for the FPGA.

33 SPI Flash can be used for FPGA configuration or to hold user data.

34 IIC Fan Controller and Temperature/Voltage Monitor

35 A piezo audio transducer

36 VGA Input Video Codec

37 JTAG Trace/Debug

38 Rotary Encoder

39 Differential GTP/GTX Input and Output with SMA Connectors

40 PCI Express Interface

41 Serial-ATA Host Connectors

42 SFP Connector

43 GTP/GTX Clocking Circuitry

44 Soft Touch Landing Pad

45 System Monitor

References

[1] M. Rewienski, "A perspective on fast-spice simulation technology," *Simulation and Verification of Electronic and Biological Systems*, p. 23, 2011.

[2] M. Merrett, P. Asenov, Y. Wang, M. Zwolinski, D. Reid, C. Millar, S. Roy, Z. Liu, S. Furber, and A. Asenov, "Modelling circuit performance variations due to statistical variability: Monte carlo static timing analysis," in *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2011*, pp. 1–4, IEEE, 2011.

[3] C. Ho, A. Ruehli, and P. Brennan, "The modified nodal approach to network analysis," *Circuits and Systems, IEEE Transactions on*, vol. 22, no. 6, pp. 504–509, 1975.

[4] K. Gulati, J. Croix, S. Khatr, and R. Shastry, "Fast circuit simulation on graphics processing units," in *Proceedings of the 2009 Asia and South Pacific Design Automation Conference*, pp. 403–408, IEEE Press, 2009.

[5] N. Kapre, *SPICE 2–A Spatial Parallel Architecture for Accelerating the SPICE Circuit Simulator*. PhD thesis, California Institute of Technology, 2010.

[6] K. Dixit, "Overview of the spec benchmarks," *The Benchmark Handbook*, pp. 489–521, 1993.

[7] K. Gulati and S. Khatri, *Hardware Acceleration of EDA Algorithms: Custom ICs, FPGAs and GPUs*. Springer Verlag, 2010.

[8] M. Edahiro, "Parallelizing fundamental algorithms such as sorting on multi-core processors for eda acceleration," in *Proceedings of the 2009 Asia and South Pacific Design Automation Conference*, pp. 230–233, IEEE Press, 2009.

[9] Y. Deng, B. Wang, and S. Mu, "Taming irregular eda applications on gpus," in *Computer-Aided Design-Digest of Technical Papers, 2009. ICCAD 2009. IEEE/ACM International Conference on*, pp. 539–546, IEEE, 2009.

[10] H. Qian, Y. Deng, B. Wang, and S. Mu, "Towards accelerating irregular eda applications with gpus," *Integration, the VLSI Journal*, 2011.

[11] A. Bayoumi and Y. Hanafy, "Massive parallelization of spice device model evaluation on gpu-based simd architectures," in *Proceedings of the 1st international forum on Next-generation multicore/manycore technologies*, p. 12, ACM, 2008.

[12] P. Li, "Parallel circuit simulation: A historical perspective and recent developments," *Foundations and Trends® in Electronic Design Automation*, vol. 5, no. 4, pp. 211–318, 2011.

[13] J. Michalakes and M. Vachharajani, "Gpu acceleration of numerical weather prediction," in *Parallel and Distributed Processing, 2008. IPDPS 2008. IEEE International Symposium on*, pp. 1–7, IEEE, 2008.

[14] L. de P Veronese and R. Krohling, "Swarm's flight: accelerating the particles using c-cuda," in *Evolutionary Computation, 2009. CEC'09. IEEE Congress on*, pp. 3264–3270, IEEE, 2009.

[15] M. Baboulin, A. Buttari, J. Dongarra, J. Kurzak, J. Langou, J. Langou, P. Luszczek, and S. Tomov, "Accelerating scientific computations with mixed precision algorithms," *Computer Physics Communications*, vol. 180, no. 12, pp. 2526–2533, 2009.

[16] G. Dasika, A. Sethia, T. Mudge, and S. Mahlke, "Pepsc: A power-efficient processor for scientific computing," in *Parallel Architectures and Compilation Techniques (PACT), 2011 International Conference on*, pp. 101–110, IEEE, 2011.

[17] F. Lu, J. Song, X. Cao, and X. Zhu, "Cpu/gpu computing for long-wave radiation physics on large gpu clusters," *Computers & Geosciences*, 2011.

[18] D. Bailey, "High-precision floating-point arithmetic in scientific computation," *Computing in science & engineering*, vol. 7, no. 3, pp. 54–61, 2005.

[19] J. Johnson, P. Vachranukunkiet, S. Tiwari, P. Nagvajara, and C. Nwankpa, "Performance analysis of loadflow computation using fpga," in *Proc. of 15th Power Systems Computation Conference*, 2005.

[20] J. Hennessy and D. Patterson, *Computer architecture: a quantitative approach*. Morgan Kaufmann Pub, 2011.

[21] C. Edwards, "game on for acceleration," *Engineering & Technology*, vol. 3, no. 11, pp. 36–38, 2008.

[22] G. Moore, "Cramming more components onto integrated circuits," *Proceedings of the IEEE*, vol. 86, no. 1, pp. 82–85, 1998.

[23] K. Olukotun and L. Hammond, "The future of microprocessors," *Queue*, vol. 3, no. 7, pp. 26–29, 2005.

[24] J. Held, J. Bautista, and S. Koehl, "From a few cores to many: A tera-scale computing research overview," *Intel*, 2006.

[25] M. Hill and M. Marty, "Amdahl's law in the multicore era," *Computer*, vol. 41, no. 7, pp. 33–38, 2008.

[26] A. Sodan, J. Machina, A. Deshmeh, K. Macnaughton, and B. Esbaugh, "Parallelism via multithreaded and multicore cpus," *Computer*, vol. 43, no. 3, pp. 24–32, 2010.

[27] V. Kindratenko and P. Trancoso, "Trends in high-performance computing," *Computing in Science & Engineering*, vol. 13, no. 3, pp. 92–95, 2011.

[28] H. Meuer, "The top500 project," *URL: http://www. top500. org/*, 2011.

[29] R. González, C. Zato, R. Benito, M. Hernández, J. Hernández, and J. De Paz, "Samasgc: Sequencing analysis with a multiagent system and grid computing," in *6th International Conference on Practical Applications of Computational Biology & Bioinformatics*, pp. 209–216, Springer, 2012.

[30] G. Hawkes, "Dsp: Designing for optimal results. high-performance dsp using virtex-4 fpgas," 2005.

[31] N. Gourdain, M. Montagnac, F. Wlassow, and M. Gazaix, "High-performance computing to simulate large-scale industrial flows in multistage compressors," *International Journal of High Performance Computing Applications*, vol. 24, no. 4, pp. 429–443, 2010.

[32] A. Hunter, F. Saied, C. Le, and M. Koslowski, "Large-scale 3d phase field dislocation dynamics simulations on high-performance architectures," *International Journal of High Performance Computing Applications*, vol. 25, no. 2, pp. 223–235, 2011.

[33] S. Swaminarayan, T. Germann, K. Kadau, and G. Fossum, "369 tflop/s molecular dynamics simulations on the roadrunner general-purpose heterogeneous supercomputer," in *High*

Performance Computing, Networking, Storage and Analysis, 2008. SC 2008. International Conference for, pp. 1–10, IEEE, 2008.

[34] V. Turner, C. Ingle, and R. Bigliani, "Reducing greenhouse gases through intense use of information and communication technology," in *IDC White Paper*, IDC, 2009.

[35] A. Beloglazov, R. Buyya, Y. Lee, A. Zomaya, *et al.*, "A taxonomy and survey of energy-efficient data centers and cloud computing systems," *Advances in Computers*, vol. 82, pp. 47–111, 2011.

[36] J. Baliga, R. Ayre, K. Hinton, and R. Tucker, "Green cloud computing: Balancing energy in processing, storage, and transport," *Proceedings of the IEEE*, vol. 99, no. 1, pp. 149–167, 2011.

[37] G. Valentini, W. Lassonde, S. Khan, N. Min-Allah, S. Madani, J. Li, L. Zhang, L. Wang, N. Ghani, J. Kolodziej, *et al.*, "An overview of energy efficiency techniques in cluster computing systems," *Cluster Computing*, pp. 1–13, 2011.

[38] T. El-Ghazawi, E. El-Araby, M. Huang, K. Gaj, V. Kindratenko, and D. Buell, "The promise of high-performance reconfigurable computing," *Computer*, vol. 41, no. 2, pp. 69–76, 2008.

[39] C. Schryver, I. Shcherbakov, F. Kienle, N. Wehn, H. Marxen, A. Kostiuk, and R. Korn, "An energy efficient fpga accelerator for monte carlo option pricing with the heston model," in *Reconfigurable Computing and FPGAs (ReConFig), 2011 International Conference on*, pp. 468–474, IEEE, 2011.

[40] M. Lin, I. Lebedev, and J. Wawrzynek, "Openrcl: low-power high-performance computing with reconfigurable devices," in *Field Programmable Logic and Applications (FPL), 2010 International Conference on*, pp. 458–463, IEEE, 2010.

[41] H. Lange, F. Stock, A. Koch, and D. Hildenbrand, "Acceleration and energy efficiency of a geometric algebra computation using reconfigurable computers and gpus," in *Field Programmable Custom Computing Machines, 2009. FCCM'09. 17th IEEE Symposium on*, pp. 255–258, IEEE, 2009.

[42] K. Benkrid, A. Akoglu, C. Ling, Y. Song, Y. Liu, and X. Tian, "High performance biological pairwise sequence alignment: Fpga versus gpu versus cell be versus gpp," *International Journal of Reconfigurable Computing*, vol. 2012, 2012.

[43] M. Zwolinski, *Digital System Design with SystemVerilog*. Prentice Hall Press, 2009.

[44] V. Chandrasetty, *VLSI Design: A Practical Guide for FPGA and ASIC Implementations*. Springer Verlag, 2011.

[45] C. Valderrama, L. Jojczyk, P. DaCunha Possa, and J. Dondo Gazzano, "Fpga and asic convergence," in *Programmable Logic (SPL), 2011 VII Southern Conference on*, pp. 269–274, IEEE, 2011.

[46] S. Yang and T. McGinnity, "A biologically plausible real-time spiking neuron simulation environment based on a multiple-fpga platform," *ACM SIGARCH Computer Architecture News*, vol. 39, no. 4, pp. 78–81, 2011.

[47] D. Yong, C. Lei, W. Yucheng, Y. Min, Q. Xiameng, H. Shaoyang, and J. Yunde, "A real-time system for 3d recovery of dynamic scene with multiple rgbd imagers," in *Computer Vision and Pattern Recognition Workshops (CVPRW), 2011 IEEE Computer Society Conference on*, pp. 1–8, IEEE, 2011.

[48] S. Asaad, R. Bellofatto, B. Brezzo, C. Haymes, M. Kapur, B. Parker, T. Roewer, P. Saha, T. Takken, and J. Tierno, "A cycle-accurate, cycle-reproducible multi-fpga system for accelerating multi-core processor simulation," in *Proceedings of the ACM/SIGDA international symposium on Field Programmable Gate Arrays*, pp. 153–162, ACM, 2012.

[49] H. Jin, D. Jespersen, P. Mehrotra, and R. Biswas, "High performance computing using mpi and openmp on multi-core parallel systems," *Parallel Computing*, 2011.

[50] F. de Dinechin and B. Pasca, "Designing custom arithmetic data paths with flopoco," *IEEE Design & Test*, vol. 28, no. 4, pp. 18–27, 2011.

[51] K. Asanovic, R. Bodik, J. Demmel, T. Keaveny, K. Keutzer, J. Kubiatowicz, N. Morgan, D. Patterson, K. Sen, J. Wawrzynek, *et al.*, "A view of the parallel computing landscape," *Communications of the ACM*, vol. 52, no. 10, pp. 56–67, 2009.

[52] P. Jogalekar and M. Woodside, "Evaluating the scalability of distributed systems," *Parallel and Distributed Systems, IEEE Transactions on*, vol. 11, no. 6, pp. 589–603, 2000.

[53] G. Amdahl, "Validity of the single processor approach to achieving large scale computing capabilities," in *Proceedings of the April 18-20, 1967, spring joint computer conference*, pp. 483–485, ACM, 1967.

[54] D. Eager, J. Zahorjan, and E. Lazowska, "Speedup versus efficiency in parallel systems," *Computers, IEEE Transactions on*, vol. 38, no. 3, pp. 408–423, 1989.

[55] N. Woods and T. VanCourt, "Fpga acceleration of quasi-monte carlo in finance," in *Field Programmable Logic and Applications, 2008. FPL 2008. International Conference on*, pp. 335–340, IEEE, 2008.

[56] H. Guo, L. Su, Y. Wang, and Z. Long, "Fpga-accelerated molecular dynamics simulations system," in *Scalable Computing and Communications; Eighth International Conference on Embedded Computing, 2009. SCALCOM-EMBEDDEDCOM'09. International Conference on*, pp. 360–365, IEEE, 2009.

[57] G. Morris, D. Thomas, and W. Luk, "Fpga accelerated low-latency market data feed processing," in *High Performance Interconnects, 2009. HOTI 2009. 17th IEEE Symposium on*, pp. 83–89, IEEE, 2009.

[58] J. Chen, J. Cong, M. Yan, and Y. Zou, "Fpga-accelerated 3d reconstruction using compressive sensing," in *Proceedings of the ACM/SIGDA international symposium on Field Programmable Gate Arrays*, pp. 163–166, ACM, 2012.

[59] F. de Dinechin, J. Detrey, O. Cret, and R. Tudoran, "When fpgas are better at floating-point than microprocessors," in *Proceedings of the 16th international ACM/SIGDA symposium on Field programmable gate arrays*, pp. 260–260, ACM, 2008.

[60] K. Underwood, "Fpgas vs. cpus: trends in peak floating-point performance," in *Proceedings of the 2004 ACM/SIGDA 12th international symposium on Field programmable gate arrays*, pp. 171–180, ACM, 2004.

[61] D. Buell, T. El-Ghazawi, K. Gaj, and V. Kindratenko, "Guest editor's introduction: High-performance reconfigurable computing," *Computer*, pp. 23–27, 2007.

[62] T. Preußer, M. Zabel, and R. Spallek, "Accelerating computations on fpga carry chains by operand compaction," in *2011 20th IEEE Symposium on Computer Arithmetic*, pp. 95–102, IEEE, 2011.

[63] S. Hauck, M. Hosler, and T. Fry, "High-performance carry chains for fpga's," *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, vol. 8, no. 2, pp. 138–147, 2000.

[64] N. Mehta, "Xilinx 7 series fpgas: The logical advantage," *Xilinx White Paper: 7 Series FPGAs*, 2012.

[65] S. Banescu, F. De Dinechin, B. Pasca, and R. Tudoran, "Multipliers for floating-point double precision and beyond on fpgas," *ACM SIGARCH Computer Architecture News*, vol. 38, no. 4, pp. 73–79, 2011.

[66] I. Kuon and J. Rose, "Measuring the gap between fpgas and asics," *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, vol. 26, no. 2, pp. 203–215, 2007.

[67] I. Kuon and J. Rose, "Exploring area and delay tradeoffs in fpgas with architecture and automated transistor design," *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, vol. 19, no. 1, pp. 71–84, 2011.

[68] E. Chung, J. Hoe, and K. Mai, "Coram: an in-fabric memory architecture for fpga-based computing," in *Proceedings of the 19th ACM/SIGDA international symposium on Field programmable gate arrays*, pp. 97–106, ACM, 2011.

[69] Xilinx, "Virtex 7 product table." "http://www.xilinx.com/publications/prod-mktg/Virtex7-Product-Table.pdf", 2012.

[70] V. Betz and S. Brown, "Fpga challenges and opportunities at 40nm and beyond," in *Field Programmable Logic and Applications, 2009. FPL 2009. International Conference on*, p. 4, IEEE, 2009.

[71] A. Heinecke and M. Bader, "Towards many-core implementation of lu decomposition using peano curves," in *Proceedings of the combined workshops on UnConventional high performance computing workshop plus memory access workshop*, pp. 21–30, ACM, 2009.

[72] I. Venetis and G. Gao, "Mapping the lu decomposition on a many-core architecture: challenges and solutions," in *Proceedings of the 6th ACM conference on Computing frontiers*, pp. 71–80, ACM, 2009.

[73] D. Maurer and C. Wieners, "A parallel block lu decomposition method for distributed finite element matrices," *Parallel Computing*, 2011.

[74] J. Dongarra, "Performance of various computers using standard linear equations software," *Rapport technique, Computer Science Department, University of Tennessee, Knoxville, Tennessee*, 2011.

[75] J. Demmel, L. Grigori, M. Hoemmen, and J. Langou, "Communication-optimal parallel and sequential qr and lu factorizations," *SIAM Journal on Scientific Computing*, vol. 34, p. A206, 2012.

[76] X. Wang and S. Ziavras, "Parallel lu factorization of sparse matrices on fpga-based configurable computing engines," *Concurrency and Computation: Practice and Experience*, vol. 16, no. 4, pp. 319–343, 2004.

[77] X. Wang and N. J. I. of Technology, *Design and Resource Management of Reconfigurable Multiprocessors for Data-parallel Applications*. PhD thesis, New Jersey Institute of Technology, 2006.

[78] J. Johnson, T. Chagnon, P. Vachranukunkiet, P. Nagvajara, and C. Nwankpa, "Sparse lu decomposition using fpga," in *International Workshop on State-of-the-Art in Scientific and Parallel Computing (PARA)*, 2008.

[79] X. Wang, S. Ziavras, C. Nwankpa, J. Johnson, and P. Nagvajara, "Parallel solution of newton's power flow equations on configurable chips," *International Journal of Electrical Power & Energy Systems*, vol. 29, no. 5, pp. 422–431, 2007.

[80] N. Kapre and A. DeHon, "Parallelizing sparse matrix solve for spice circuit simulation using fpgas," in *Field-Programmable Technology, 2009. FPT 2009. International Conference on*, pp. 190–198, IEEE, 2009.

[81] W. Wu, Y. Shan, X. Chen, Y. Wang, and H. Yang, "Fpga accelerated parallel sparse matrix factorization for circuit simulations," in *Proceedings of the 7th international conference on Reconfigurable computing: architectures, tools and applications*, pp. 302–315, Springer-Verlag, Springer, 2011.

[82] L. Nagel, "Spice2: A computer program to simulate semiconductor circuits," *Univ. California*, 1975.

[83] T. Quarles, "Spice3f5 users' guide," tech. rep., Technical report, University of California-Berkeley, Berkeley, California, 1994.

[84] T. Ypma, "Historical development of the newton-raphson method," *SIAM review*, pp. 531–551, 1995.

[85] T. Weng, R. Perng, and B. Chapman, "Openmp implementation of spice3 circuit simulator," *OpenMP Shared Memory Parallel Programming*, pp. 361–371, 2008.

[86] C. Desoer and E. Kuh, *Basic circuit theory*. Tata McGraw-Hill Education, 1984.

[87] W. Gautschi, *Numerical analysis*. Birkhauser, 2011.

[88] S. Venkata *et al.*, "Computational methods for electric power systems [book reviews]," *Power and Energy Magazine, IEEE*, vol. 9, no. 2, pp. 78–80, 2011.

[89] H. Niessner and K. Reichert, "On computing the inverse of a sparse matrix," *International journal for numerical methods in engineering*, vol. 19, no. 10, pp. 1513–1526, 1983.

[90] P. Amestoy, I. Duff, Y. Robert, F. Rouet, and B. Uçar, "On computing inverse entries of a sparse matrix in an out-of-core environment," *SIAM J. Sci. Comput., to appear*, 2010.

[91] R. Varga, *Matrix iterative analysis*, vol. 27. Springer, 2010.

[92] H. Rutishauser, "The jacobi method for real symmetric matrices," *Numerische Mathematik*, vol. 9, no. 1, pp. 1–10, 1966.

[93] W. Kahan, *Gauss-Seidel methods of solving large systems of linear equations*. PhD thesis, University of Toronto, 1958.

[94] M. Hestenes and E. Stiefel, "Methods of conjugate gradients for solving linear systems," *Journal Of Research Of The National Bureau Of Standards*, vol. 49, no. 6, pp. 409–436, 1952.

[95] Catena Software Ltd, *Technical Note: How SPICE Works*, July 2003.

[96] F. Najm, *Circuit Simulation*. Wiley-IEEE Press, 2010.

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

[98] T. Davis, *MATLAB Primer*. CRC Press, Inc., 2010.

[99] V. Litovski and M. Zwolinski, *VLSI circuit simulation and optimization*. Springer, 1997.

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

[101] J. Liu, "The multifrontal method for sparse matrix solution: Theory and practice," *Siam Review*, pp. 82–109, 1992.

[102] T. Davis and E. Natarajan, "Algorithm 8xx: Klu, a direct sparse solver for circuit simulation problems," *ACM Trans. MS*, vol. 5, no. 1, pp. 1–14, 2009.

[103] I. Duff, "On algorithms for obtaining a maximum transversal," *ACM Transactions on Mathematical Software (TOMS)*, vol. 7, no. 3, pp. 315–330, 1981.

[104] I. Duff, "Parallel implementation of multifrontal schemes," *Parallel computing*, vol. 3, no. 3, pp. 193–204, 1986.

[105] I. Naumann and H. Dirks, "Efficient reordering for direct methods in analog circuit simulation," *Electrical Engineering (Archiv fur Elektrotechnik)*, vol. 89, no. 4, pp. 333–337, 2007.

[106] I. Duff, A. Erisman, and J. Reid, *Direct methods for sparse matrices*. Clarendon Press Oxford, 1986.

[107] M. Yannakakis, "Computing the minimum fill-in is np-complete," *SIAM Journal on Algebraic and Discrete Methods*, vol. 2, no. 1, pp. 77–79, 1981.

[108] T. Quarles, A. Newton, D. Peterson, and A. Vincentelli, "Spice3f5 user's manual," *University of California, Berkeley*, 1994.

[109] D. Bryan, "The iscas'85 benchmark circuits and netlist format," *North-Carolina State University*, 1985.

[110] M. Hansen, H. Yalcin, and J. Hayes, "Unveiling the iscas-85 benchmarks: A case study in reverse engineering," *Design & Test of Computers, IEEE*, vol. 16, no. 3, pp. 72–80, 1999.

[111] J. Xu, "Perform the spice simulation of iscas85 benchmark circuits for research." http://www.ece.uic.edu/ masud/iscas2spice.htm, 2008.

[112] P. Cox, R. Burch, D. Hocevar, P. Yang, and B. Epler, "Direct circuit simulation algorithms for parallel processing [vlsi]," *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, vol. 10, no. 6, pp. 714–725, 1991.

[113] N. Kapre and A. DeHon, "Accelerating spice model-evaluation using fpgas," in *Field Programmable Custom Computing Machines, 2009. FCCM'09. 17th IEEE Symposium on*, pp. 37–44, IEEE, 2009.

[114] R. Saleh, K. Gallivan, M. Chang, I. Hajj, D. Smart, and T. Trick, "Parallel circuit simulation on supercomputers," *Proceedings of the IEEE*, vol. 77, no. 12, pp. 1915–1931, 1989.

[115] P. Lee, S. Ito, T. Hashimoto, J. Sato, T. Touma, and G. Yokomizo, "A parallel and accelerated circuit simulator with precise accuracy," in *Design Automation Conference,*

2002. Proceedings of ASP-DAC 2002. 7th Asia and South Pacific and the 15th International Conference on VLSI Design. Proceedings., pp. 213–218, IEEE, 2002.

[116] B. Murmann, P. Nikaeen, D. Connelly, and R. Dutton, "Impact of scaling on analog performance and associated modeling needs," *Electron Devices, IEEE Transactions on*, vol. 53, no. 9, pp. 2160–2167, 2006.

[117] M. Chan and C. Hu, "The engineering of bsim for the nano-technology era and beyond," *Modeling and Simulation Microsistem*, pp. 662–665, 2002.

[118] S. Markus, S. Kim, K. Pantazopoulos, A. Ocken, E. Houstis, P. Wu, S. Weerawarana, and D. Maharry, "Performance evaluation of mpi implementations and mpi based parallel ellpack solvers," in *MPI Developer's Conference, 1996. Proceedings., Second*, pp. 162–169, IEEE, 1996.

[119] H. Kotakemori, H. Hasegawa, and A. Nishida, "Performance evaluation of a parallel iterative method library using openmp," in *High-Performance Computing in Asia-Pacific Region, 2005. Proceedings. Eighth International Conference on*, pp. 5–pp, IEEE, 2005.

[120] A. Maache, *A prototype parallel multi-FPGA accelerator for SPICE CMOS model evaluation.* PhD thesis, University of Southampton, 2011.

[121] H. Markowitz, "The elimination form of the inverse and its application to linear programming," *Management Science*, vol. 3, no. 3, pp. 255–269, 1957.

[122] A. Vladimirescu, *LSI circuit simulation on vector computers.* PhD thesis, University of California, Berkeley, 1982.

[123] J. Deutsch and A. Newton, "A multiprocessor implementation of relaxation-based electrical circuit simulation," in *Proceedings of the 21st Design Automation Conference*, pp. 350–357, IEEE Press, 1984.

[124] G. Jacob, A. Newton, and D. Pederson, "An empirical analysis of the performance of a multiprocessor-based circuit simulator," in *Proceedings of the 23rd ACM/IEEE Design Automation Conference*, pp. 588–593, IEEE Press, 1986.

[125] C. Yuan, R. Lucas, P. Chan, and R. Dutton, "Parallel electronic circuit simulation on the ipsc system," in *Custom Integrated Circuits Conference, 1988., Proceedings of the IEEE 1988*, pp. 6–5, IEEE, 1988.

[126] M. Chang and I. Hajj, "ipride: A parallel integrated circuit simulator using direct method," in *Computer-Aided Design, 1988. ICCAD-88. Digest of Technical Papers., IEEE International Conference on*, pp. 304–307, IEEE, 1988.

[127] J. White and A. Sangiovanni-Vincentelli, *Relaxation techniques for the simulation of VLSI circuits*. Kluwer Academic Publishers, 1987.

[128] W. Knight, "Two heads are better than one [dual-core processors]," *IEE Review*, vol. 51, no. 9, pp. 32–35, 2005.

[129] R. Ramanathan, "Intel® multi-core processors," *Making the Move to Quad-Core and Beyond*, 2006.

[130] J. Dorsey, S. Searles, M. Ciraula, S. Johnson, N. Bujanos, D. Wu, M. Braganza, S. Meyers, E. Fang, and R. Kumar, "An integrated quad-core opteron processor," in *Solid-State Circuits Conference, 2007. ISSCC 2007. Digest of Technical Papers. IEEE International*, pp. 102–103, Ieee, 2007.

[131] S. Borkar, "Thousand core chips: a technology perspective," in *Proceedings of the 44th annual Design Automation Conference*, pp. 746–749, ACM, 2007.

[132] S. Sapatnekar, E. Haritan, K. Keutzer, A. Devgan, D. Kirkpatrick, S. Meier, D. Pryor, and T. Spyrou, "Reinventing eda with manycore processors," in *Proceedings of the 45th annual Design Automation Conference*, pp. 126–127, ACM, 2008.

[133] S. Hutchinson, E. Keiter, R. Hoekstra, H. Watts, A. Waters, R. Schells, and S. Wix, "The xyce parallel electronic simulator-an overview," in *IEEE International Symposium on Circuits and Systems, Sydney (AU)*, 2000.

[134] N. Frohlich, V. Glockel, and J. Fleischmann, "A new partitioning method for parallel simulation of vlsi circuits on transistor level," in *Design, Automation and Test in Europe Conference and Exhibition 2000. Proceedings*, pp. 679–684, IEEE, 2000.

[135] D. Martin, R. Radhakrishnan, D. Rao, M. Chetlur, K. Subramani, and P. Wilsey, "Analysis and simulation of mixed-technology vlsi systems," *Journal of parallel and distributed computing*, vol. 62, no. 3, pp. 468–493, 2002.

[136] A. Devgan, "Accelerated design of analog, mixed-signal circuits with finesim^{TM} and titan^{TM}," in *SoC Design Conference (ISOCC), 2009 International*, pp. 282–286, IEEE, 2009.

[137] R. Daniels, H. Sosen, and H. Elhak, "Accelerating analog simulation with hspice precision parallel technology," tech. rep., Synopsys, Tech. Rep, 2010.

[138] Cadence, "Virtuoso accelerated parallel simulator," *http://www.cadence.com*, 2009.

[139] F. Lu, J. Song, F. Yin, and X. Zhu, "Performance evaluation of hybrid programming patterns for large cpu/gpu heterogeneous clusters," *Computer Physics Communications*, vol. 183, no. 6, pp. 1172–1181, 2012.

[140] J. Stone, D. Gohara, and G. Shi, "Opencl: A parallel programming standard for heterogeneous computing systems," *Computing in science & engineering*, vol. 12, no. 3, p. 66, 2010.

[141] F. Jacob, D. Whittaker, S. Thapaliya, P. Bangalore, M. Mernik, and J. Gray, "Cudacl: A tool for cuda and opencl programmers," in *High Performance Computing (HiPC), 2010 International Conference on*, pp. 1–11, IEEE, 2010.

[142] M. Saldana, D. Nunes, E. Ramalho, and P. Chow, "Configuration and programming of heterogeneous multiprocessors on a multi-fpga system using tmd-mpi," in *Reconfigurable Computing and FPGA's, 2006. ReConFig 2006. IEEE International Conference on*, pp. 1–10, IEEE, 2006.

[143] A. Anderson, G. Morris, and K. Abed, "Achieving true parallelism on a high performance heterogeneous computer via a threaded programming model," in *Southeastcon, 2011 Proceedings of IEEE*, pp. 283–286, IEEE, 2011.

[144] V. Aggarwal, G. Stitt, A. George, and C. Yoon, "Scf: A framework for task-level coordination in reconfigurable, heterogeneous systems," *ACM Transactions on Reconfigurable Technology and Systems (TRETS)*, vol. 5, no. 2, p. 7, 2012.

[145] Y. Corre, J. Diguet, D. Heller, and L. Lagadec, "A framework for high-level synthesis of heterogeneous mp-soc," in *Proceedings of the great lakes symposium on VLSI*, pp. 283–286, ACM, 2012.

[146] J. Kim, S. Seo, J. Lee, J. Nah, G. Jo, and J. Lee, "Snucl: an opencl framework for heterogeneous cpu/gpu clusters," in *Proceedings of the 26th ACM international conference on Supercomputing*, pp. 341–352, ACM, 2012.

[147] A. Gupta, "Recent advances in direct methods for solving unsymmetric sparse systems of linear equations," *ACM Transactions on Mathematical Software (TOMS)*, vol. 28, no. 3, pp. 301–324, 2002.

[148] M. Heath, E. Ng, and B. Peyton, "Parallel algorithms for sparse linear systems," *SIAM review*, pp. 420–460, 1991.

[149] W. Dong and P. Li, "A parallel harmonic-balance approach to steady-state and envelope-following simulation of driven and autonomous circuits," *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, vol. 28, no. 4, pp. 490–501, 2009.

[150] H. Thornquist, E. Keiter, R. Hoekstra, D. Day, and E. Boman, "A parallel preconditioning strategy for efficient transistor-level circuit simulation," in *Computer-Aided Design-Digest of Technical Papers, 2009. ICCAD 2009. IEEE/ACM International Conference on*, pp. 410–417, IEEE, 2009.

[151] D. Webber and A. Sangiovanni-Vincentelli, "Circuit simulation on the connection machine," in *Proceedings of the 24th ACM/IEEE Design Automation Conference*, pp. 108–113, ACM, 1987.

[152] U. Wever and Q. Zheng, "Parallel transient analysis for circuit simulation," in *System Sciences, 1996., Proceedings of the Twenty-Ninth Hawaii International Conference on,*, vol. 1, pp. 442–447, IEEE, 1996.

[153] A. Sangiovanni-Vincentelli, L. Chen, and L. Chua, "An efficient heuristic cluster algorithm for tearing large-scale networks," *Circuits and Systems, IEEE Transactions on*, vol. 24, no. 12, pp. 709–717, 1977.

[154] W. Dong, P. Li, and X. Ye, "Wavepipe: parallel transient simulation of analog and digital circuits on multi-core shared-memory machines," in *Proceedings of the 45th annual Design Automation Conference*, pp. 238–243, ACM, 2008.

[155] T. Davis, *Direct methods for sparse linear systems*, vol. 2. Society for Industrial Mathematics, 2006.

[156] J. Dongarra, F. Gustavson, and A. Karp, "Implementing linear algebra algorithms for dense matrices on a vector pipeline machine," *Siam Review*, pp. 91–112, 1984.

[157] W. Press, *Numerical recipes in FORTRAN: the art of scientific computing*, vol. 1. Cambridge Univ Pr, 1992.

[158] C. Moler and R. Schreiber, "Sparse matrices in matlab: Design and implementation," 1998.

[159] X. Li and J. Demmel, "Making sparse gaussian elimination scalable by static pivoting," in *Proceedings of the 1998 ACM/IEEE conference on Supercomputing (CDROM)*, pp. 1–17, IEEE Computer Society, 1998.

[160] I. Duff, R. Grimes, and J. Lewis, "Sparse matrix test problems," *ACM Transactions on Mathematical Software (TOMS)*, vol. 15, no. 1, pp. 1–14, 1989.

[161] J. Liu, "The role of elimination trees in sparse factorization," *SIAM Journal on Matrix Analysis and Applications*, vol. 11, p. 134, 1990.

[162] J. Gilbert and J. Liu, "Elimination structures for unsymmetric sparse lu factors," *SIAM Journal on Matrix Analysis and Applications*, vol. 14, pp. 334–334, 1993.

[163] I. Duff and H. Van Der Vorst, "Developments and trends in the parallel solution of linear systems," *Parallel Computing*, vol. 25, no. 13-14, pp. 1931–1970, 1999.

[164] D. Rose and R. Tarjan, "Algorithmic aspects of vertex elimination on directed graphs," *SIAM J. Appl. Math.*, vol. 34, pp. 176–197, 1978.

[165] A. George and W. Liu, "The evolution of the minimum degree ordering algorithm," *SIAM Review*, vol. 31, no. 1, pp. 1–19, 1989.

[166] P. Amestoy, T. Davis, and I. Duff, "An approximate minimum degree ordering algorithm," *SIAM J. Matrix Analysis & Applic*, vol. 17, no. 4, pp. 886–905, 1996.

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

[168] A. George, "Nested dissection of a regular finite element mesh," *SIAM Journal on Numerical Analysis*, vol. 10, no. 2, pp. 345–363, 1973.

[169] G. Karypis and V. Kumar, "Analysis of multilevel graph partitioning," in *Proceedings of the 1995 ACM/IEEE conference on Supercomputing*, p. 29, ACM, 1995.

[170] B. Hendrickson and T. Kolda, "Graph partitioning models for parallel computing," *Parallel computing*, vol. 26, no. 12, pp. 1519–1534, 2000.

[171] I. Duff, "Algorithm 575: Permutations for a zero-free diagonal [f1]," *ACM Transactions on Mathematical Software (TOMS)*, vol. 7, no. 3, pp. 387–390, 1981.

[172] E. Palamadai Natarajan, *KLU-a high performance sparse linear system solver for circuit simulation problems.* PhD thesis, MS Thesis, CISE Department, University of Florida, 2005.

[173] Z. Galil, "Efficient algorithms for finding maximum matching in graphs," *ACM Computing Surveys (CSUR)*, vol. 18, no. 1, pp. 23–38, 1986.

[174] A. HSL, "collection of fortran codes for large-scale scientific computation," *See http://www. hsl. rl. ac. uk*, 2007.

[175] C. Fu and T. Yang, "Sparse lu factorization with partial pivoting on distributed memory machines," in *Supercomputing, 1996. Proceedings of the 1996 ACM/IEEE Conference on*, pp. 31–31, IEEE, 1996.

[176] I. Duff and J. Reid, "The multifrontal solution of indefinite sparse symmetric linear," *ACM Transactions on Mathematical Software (TOMS)*, vol. 9, no. 3, pp. 302–325, 1983.

[177] G. Alaghband and H. Jordan, "Sparse gaussian elimination with controlled fill-in on a shared memory multiprocessor," *Computers, IEEE Transactions on*, vol. 38, no. 11, pp. 1539–1557, 1989.

[178] T. Davis, "A parallel algorithm for sparse unsymmetric lu factorization," tech. rep., Illinois Univ., Urbana, IL (USA), 1989.

[179] C. Ashcraft, R. Grimes, J. Lewis, B. Peyton, H. Simon, and P. Bjørstad, "Progress in sparse matrix methods for large linear systems on vector supercomputers," *International Journal of High Performance Computing Applications*, vol. 1, no. 4, pp. 10–30, 1987.

[180] A. Erisman, R. Grimes, J. Lewis, W. Poole Jr, and H. Simon, "Evaluation of orderings for unsymmetric sparse matrices," *SIAM journal on scientific and statistical computing*, vol. 8, p. 600, 1987.

[181] J. Gilbert and T. Peierls, "Sparse spatial pivoting in time proportional to arithmetic operations," *SIAM journal on scientific and statistical computing*, vol. 9, no. 5, pp. 862–874, 1988.

[182] C. Gauss, *Theory of the motion of the heavenly bodies moving about the sun in conic sections: a translation of Carl Frdr. Gauss" Theoria motus": With an appendix. By Ch. H. Davis.* Little, Brown and Comp., 1857.

[183] S. Eisenstat and J. Liu, "Exploiting structural symmetry in a sparse partial pivoting code," *SIAM Journal on Scientific Computing*, vol. 14, p. 253, 1993.

[184] Y. Liao and C. Wong, "An algorithm to compact a vlsi symbolic layout with mixed constraints," in *Proceedings of the 20th Design Automation Conference*, pp. 107–112, IEEE Press, 1983.

[185] G. Micheli, *Synthesis and optimization of digital circuits*. McGraw-Hill Higher Education, 1994.

[186] T. Davis, I. Duff, P. Amestoy, J. Gilbert, S. Larimore, E. Natarajan, Y. Chen, W. Hager, and S. Rajamanickam, "Suite sparse: a suite of sparse matrix packages."

[187] I. Duff, M. Heroux, and R. Pozo, "An overview of the sparse basic linear algebra subprograms: The new standard from the blas technical forum," *ACM Transactions on Mathematical Software (TOMS)*, vol. 28, no. 2, pp. 239–267, 2002.

[188] H. Avron, G. Shklarski, and S. Toledo, "Parallel unsymmetric-pattern multifrontal sparse lu with column preordering," *ACM Transactions on Mathematical Software (TOMS)*, vol. 34, no. 2, p. 8, 2008.

[189] P. Vachranukunkiet, J. Johnson, P. Nagvajara, S. Tiwari, and C. Nwankpa, "Performance analysis of load flow on fpga," in *15th Power Systems Computational Conference August*, vol. 22, 2005.

[190] S. Toledo and A. Uchitel, "A supernodal out-of-core sparse gaussian-elimination method," *Parallel Processing and Applied Mathematics*, pp. 728–737, 2008.

[191] S. McKee, "Reflections on the memory wall," in *Proceedings of the 1st conference on Computing frontiers*, p. 162, ACM, 2004.

[192] Xilinx, *Xilinx Core Generator Floating-Point Operator v4.0*, 2010.

[193] Xilinx, *Chipscope Pro Software and Cores User Guide*. Xilinx, 2007.

[194] T. Davis, "Algorithm 832: Umfpack v4. 3—an unsymmetric-pattern multifrontal method," *ACM Transactions on Mathematical Software (TOMS)*, vol. 30, no. 2, pp. 196–199, 2004.

[195] K. Kundert and A. Sangiovanni-Vincentelli, "Sparse user's guide–a sparse linear equation solver version 1.3 a," *University of California, Berkeley*, 1988.

[196] D. Patterson and J. Hennessy, *Computer organization and design: the hardware/software interface*. Morgan Kaufmann, 2009.

[197] R. Borndörfer, C. Ferreira, and A. Martin, "Decomposing matrices into blocks," *SIAM Journal on Optimization*, vol. 9, p. 236, 1998.

[198] W. ZHAO, M. LIU, and N. MIAO, "A decomposition algorithm for multi-area reactive-power optimization based on the block bordered diagonal model [j]," *Automation of Electric Power Systems*, vol. 4, 2008.

[199] J. Rommes, P. Lenaers, and W. Schilders, "Reduction of large resistor networks," *Scientific Computing in Electrical Engineering SCEE 2008*, pp. 555–562, 2010.

[200] A. Grothey, "Massively parallel asset and liability management," in *Euro-Par 2010 Parallel Processing Workshops*, pp. 423–430, Springer, 2011.

[201] D. Koester, S. Ranka, and G. Fox, "Parallel block-diagonal-bordered sparse linear solvers for electrical power system applications," in *Scalable Parallel Libraries Conference, 1993., Proceedings of the*, pp. 195–203, IEEE, 1993.

[202] I. Duff and J. Scott, "Stabilized bordered block diagonal forms for parallel sparse solvers," *Parallel Computing*, vol. 31, no. 3, pp. 275–289, 2005.

[203] B. Kernighan and S. Lin, "An efficient heuristic procedure for partitioning graphs," *Bell System Technical Journal*, vol. 49, no. 2, pp. 291–307, 1970.

[204] R. Lucas, T. Blank, and J. Tiemann, "A parallel solution method for large sparse systems of equations," *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, vol. 6, no. 6, pp. 981–991, 1987.

[205] M. Khaira, G. Miller, and T. Sheffler, "Nested dissection: A survey and comparison of various nested dissection algorithms," tech. rep., CMU-CS-92-106R, Computer Science Department, Carnegie Mellon University, 1992.

[206] A. Zecevic and D. Siljak, "Balanced decompositions of sparse systems for multilevel parallel processing," *Circuits and Systems I: Fundamental Theory and Applications, IEEE Transactions on*, vol. 41, no. 3, pp. 220–233, 1994.

[207] G. Karypis and V. Kumar, *METIS-Unstructured Graph Partitioning and Sparse Matrix Ordering System*, 2011.

[208] C. Aykanat, A. Pinar, and Ü. Çatalyürek, "Permuting sparse rectangular matrices into block-diagonal form," *SIAM Journal on Scientific Computing*, vol. 25, no. 6, pp. 1860–1879, 2004.

[209] L. Grigori, E. Boman, S. Donfack, and T. Davis, "Hypergraph-based unsymmetric nested dissection ordering for sparse lu factorization*," *SIAM Journal on Scientific Computing*, vol. 32, no. 6, 2010.

[210] S. Pasricha, "Exploring serial vertical interconnects for 3d ics," in *Design Automation Conference, 2009. DAC'09. 46th ACM/IEEE*, pp. 581–586, IEEE, 2009.

[211] C. Lenzen, T. Locher, P. Sommer, and R. Wattenhofer, "Clock synchronization: Open problems in theory and practice," *SOFSEM 2010: Theory and Practice of Computer Science*, pp. 61–70, 2010.

[212] B. Von Herzen, "Use rocket i/o multi-gigabit transceivers to double your fpga bandwidth," *Xcell Journal, Spring*, 2002.

[213] B. Razavi, *Phase-locking in high-performance systems: from devices to architectures*. John Wiley & Sons, Inc., 2003.

[214] A. Widmer and P. Franaszek, "A dc-balanced, partitioned-block, 8b/10b transmission code," *IBM Journal of research and development*, vol. 27, no. 5, pp. 440–451, 1983.

[215] R. Dobkin, A. Morgenshtein, A. Kolodny, and R. Ginosar, "Parallel vs. serial on-chip communication," in *Proceedings of the 2008 international workshop on System level interconnect prediction*, pp. 43–50, ACM, 2008.

[216] I. LogiCore, "Xilinx ug353 logicore ip aurora 8b/10b v5.2 user guide." http://www.xilinx.com/, 2010.

[217] I. LogiCORE, "Fifo generator v5. 3," 2009.

[218] J. Gilbert and S. Teng, "Meshpart: Matlab mesh partitioning and graph separator toolbox." http://www.cerfacs.fr/algor/Softs/MESHPART/, 2010.

[219] K. Devine, E. Boman, R. Heaphy, B. Hendrickson, and C. Vaughan, "Zoltan data management services for parallel dynamic applications," *Computing in Science & Engineering*, vol. 4, no. 2, pp. 90–96, 2002.

[220] W. Cheney and D. Kincaid, *Linear Algebra: Theory and Applications*. Jones & Bartlett Publishers, 2011.