Architectural Support for Direct Sparse LU Algorithms

A Thesis

Submitted to the Faculty

of

Drexel University

by

Timothy Chagnon

in partial fulfillment of the

requirements for the degree

of

Master of Science in Computer Science

March 2010

Acknowledgements

I would like to express my deepest gratitude to my advisor, Professor Jeremy Johnson, for his continuous support and guidance throughout my career at Drexel. I would also like to thank Professor Prawat Nagvajara and Professor Chika Nwankpa for their insight and dedication to the work that we have done together.

I am grateful to my colleagues who have worked on the Power project and in our lab for their friendship and help, including Petya Vachranukunkiet, Pranab Shenoy, Michael Andrews, Doug Jones, Lingchuan Meng, Gavin Harrison, Kevin Cunningham and Anupama Kurpad.

Dedications

To my amazing wife for her support and love.

Table of Contents

List of Figures

List of Tables

Abstract

Architectural Support for Direct Sparse LU Algorithms

Timothy Chagnon

Advisor: Jeremy Johnson, PhD

Sparse linear algebra algorithms typically perform poorly on superscalar, general-purpose processors due to irregular data access patterns and indexing overhead. These algorithms are important to a number of scientific computing domains including power system simulation, which motivates this work. A variety of algorithms and techniques exist to exploit CPU features, but it has been shown that special purpose hardware support can dramatically outperform these methods. However, the development cost and scaling limitations of a custom hardware solution limit widespread use. This work presents an analysis of hardware and software performance during sparse LU decomposition in order to better understand trade-offs and to suggest the most promising approach for future research. Experimental results show that hardware support for indexing operations provides the greatest performance improvement to these algorithms and techniques or hardware that facilitate indexing operations should be explored.

1. Introduction

1.1 Overview

Sparse linear algebra algorithms typically perform poorly on superscalar, general-purpose processors due to irregular data access patterns and indexing overhead [37]. These algorithms are important to a number of scientific computing domains including power system simulation, which motivates this work [4]. A variety of algorithms and techniques exist to exploit CPU features [20], but it has been shown that special purpose hardware support can dramatically outperform these methods [6]. However, the development cost and scaling limitations of a custom hardware solution limit widespread use. This work presents an analysis of hardware and software performance during sparse LU decomposition in order to better understand trade-offs and to suggest the most promising approach for future research. Experimental results show that hardware support for indexing operations provides the greatest performance improvement to these algorithms and techniques or hardware that facilitate indexing operations should be explored.

In Chapter 1, background information on power system simulation, sparse linear algebra, general-purpose and reconfigurable architectures is provided. Chapter 2 is a discussion of straightforward Gaussian Elimination, and its implementation in software and special purpose hardware. Chapter 3 reviews advanced algorithms designed to efficiently utilize general-purpose processors. Chapter 4 compares the characteristics of power system matrices to sparse matrices from other applications. Chapter 5 presents several performance experiments and their results which support the conclusion that indexing operation hardware mechanisms can provide a significant performance boost to sparse algorithms without the need for a complete custom processor.

1.2 Power Systems

Power transmission systems are regularly simulated during normal operation so that operators have better insight into the behavior of equipment under their control. The power flow (or load flow) calculation models a system as a loosely connected graph of vertices representing connection points and edges representing transmission lines. A system of equations based on Kirchoff's current laws can be formed which create a simple mathematical model of the system. This model is initialized with measurements taken from the current state of the power system and then solved for the power flowing on each of the transmission lines. An iterated solution leads to the steady state of the power system, giving operators an indication of whether the system is stable or not [3].

Several kinds of power system analysis use the power flow calculation as a model of the power system. During day-to-day operation, contingency analysis is important to limit the cascading effects of equipment failure and reduce the risk of widespread blackouts. Contingency analysis can be performed by modifying one of the links or nodes in the power flow model to simulate a single equipment failure. Running the power flow calculation until the system reaches a steady state will determine what effect the single outage has on the whole system. Even though the power flow computation takes less than a second on modern computers, a full contingency analysis requires it to be run thousands of times for every possible component failure [31]. Other types of analysis that use power flow include future planning studies and real-time energy market pricing [3].

The equations used in the power flow model are sparse, complex and non-linear. The Newton-Raphson method is commonly used to solve the system of equations by converting to a linear approximation at each step of an iterative solution. At each step, the admittance (Y_{bus}) matrix which directly reflects the structure of the power system is converted to a linear system called the Jacobian, which is solved using sparse linear methods [30]. The solution to this system is used to update the Y_{bus} matrix and check for convergence to a steady state. Direct sparse LU decomposition of the Jacobian is the preferred method for solving the linear system. Iterative decomposition methods such as Conjugate-Gradient are not

effective at finding a solution to the power flow computation with the same performance as direct solvers due to convergence issues [30]. As a result, this work focuses on direct methods for this application. Performance analysis of the power flow calculation shows that about 85% of the total computation time is spent on LU decomposition [31, 29].

1.3 Sparse Linear Algebra

A matrix is considered sparse if it contains a large number of elements of value zero. When computing with such matrices it can be very advantageous in terms of time and memory to skip operations involving these zero elements. Operations on sparse matrices often take time and space proportional to some function of the number of non-zeros in the matrix, which can be significantly smaller than their dense equivalent. As an example, dense matrix multiplication takes $O(n^3)$ floating-point operations, but sparse matrix multiplication takes only $O(n \cdot nz)$ where nz is the number of non-zeros in the matrices.

Sparse linear algebra algorithms are important for a number of applications. The Berkeley View report on parallel computing identifies sparse matrix algorithms as one of the 12 dwarfs of high-performance computing, key algorithm families who's performance has a large impact on a variety of applications [4]. Solving sparse systems of equations is commonly used for finite-element methods (FEM) applications such as structural analysis and computational fluid dynamics. Sparse methods are also very useful for circuit simulation and analysis of power transmission systems, which is the motivating application in this work. Recent work by Kepner also suggests that sparse matrix methods can be used to efficiently implement graph algorithms [25]. The relationship of sparse matrices and graphs leads to the use of graph algorithms during parts of efficient sparse matrix algorithms.

To efficiently store, transmit and compute with sparse matrices, they must be stored in formats that take advantage of their sparsity. A very common, basic format for storing matrices on disk and transmitting them over a network is the triplet format, sometimes also referred to as Harwell-Boeing (HB) format [15]. Triplet format stores each element with its row number, column number and numeric value. A small 6-bus Jacobian matrix is shown in Figure 1.1 along with part of a triplet file representing it in Figure 1.2.

$$\begin{pmatrix}
8.17 & 0.06 & & & & & & -4.24 \\
-2.06 & 7.98 & & & & & & 0.87 \\
 & & 12.52 & 1.41 & -7.20 & & & -5.32 \\
 & & -3.36 & 12.47 & 2.19 & & & 1.17 \\
 & & -7.44 & -1.13 & 24.55 & -8.74 & -1.75 & \\
 & & & & -8.56 & 21.32 & 3.27 & -4.32 \\
 & & & & 2.44 & -6.14 & 21.15 & 0.87 \\
-4.32 & -0.13 & -5.43 & -0.45 & & -4.38 & -0.57 & 14.13
\end{pmatrix}$$

Figure 1.1: Sparse Jacobian Matrix for a 6-bus Power System

```
1  0 0 8.169161
2  1 0 -2.058465
3  7 0 -4.321671
4  0 1 0.062476
5  1 1 7.98146
6  7 1 -0.134087
7  2 2 12.519152
8  3 2 -3.356573
9  4 2 -7.444455
10 7 2 -5.430451
```

Figure 1.2: Part of a Triplet File for 6-bus Matrix

Within a program a matrix may be stored in triplet format for reading and writing to disk, but it is generally inefficient to compute with this format. An example structure for storing a triplet matrix is shown in Figure 1.3, where i, j, and x are arrays of length nz and each set of entries $i[p]$, $j[p]$ and $x[p]$ denote a row, column and value entry for all integers p in the range $[0, nz-1]$.

The most common sparse matrix formats for efficient computation are compressed sparse row (CSR) and compressed sparse column (CSC). These formats are internally the same and usage dictates whether they are used in a row-major (CSR) or column-major (CSC) format. Figure 1.4 shows a C struct based on one used by the CSparse package [11] to hold CSC and CSR matrices.

The arrays i and x are length $nzmax$ store all non-zero elements minor-index (row numbers for CSC) and values contiguously. All elements in a single row are contiguous and

```
1 struct triplet {
2     int nz;
3     int *i;
4     int *j;
5     double *x;
6 };
```

Figure 1.3: Triplet Structure

```
1 typedef struct sparse
2 {
3     int nzmax ;      /* maximum number of entries */
4     int m ;          /* number of rows */
5     int n ;          /* number of columns */
6     int *p ;         /* column pointers (size n+1) */
7     int *i ;         /* row indices, size nzmax */
8     double *x ;      /* numerical values, size nzmax */
9 } cs ;
```

Figure 1.4: CSC Structure

the rows are in ascending order. The array p is length $n+1$ and contains the starting index in i and x for each column. The entry $p[n]$ contains nz, the total non-zero entries in the matrix. Figure 1.5 shows a graphical representation of the 6-bus matrix in CSC format.

To understand how to compute with matrices in compressed form, it is useful to look at some basic operations. Since the rows or columns are stored contiguously, it is very expensive to insert an element into a matrix at an arbitrary location. All non-zeros in the matrix following the insertion point would have to be moved. To avoid this, operations are typically performed such that elements are added to the results a row or column at a time. Sometimes it is necessary to pre-calculate row or column lengths beforehand or to scatter intermediate results into a temporary full vector while working on a sparse computation. Figure 1.6 shows pseudo-code for transposing a matrix based on the `cs_transpose` function from the CSparse package presented in [11] which operates on CSC matrices.

In the case of the transpose operation, the desired result is that $C = A^T$, or equivalently

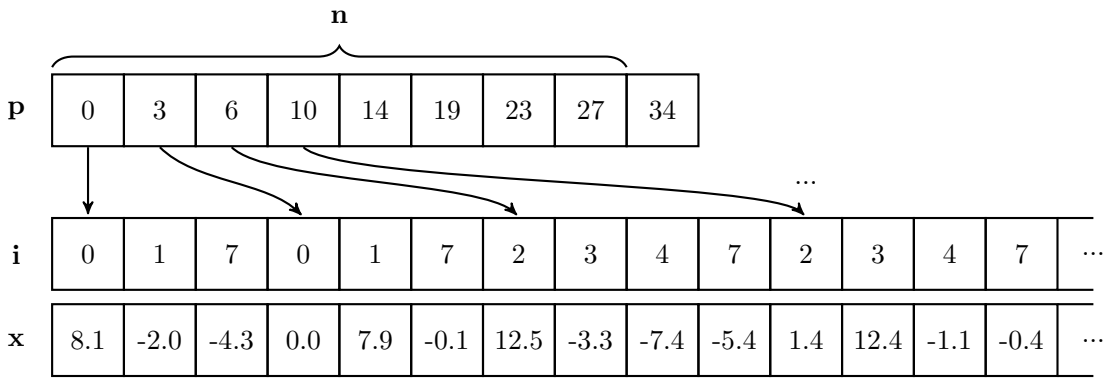


Figure 1.5: Graphical Representation of CSC Structure

```
1  transpose(A)
2      C = allocate n x m CSC matrix with Ap(n) non-zeros
3      w = allocate integer array of m zeroes
4      for every row index i in Ai
5          w(i)++
6      Cp = cumulative-sum(w)
7      w = copy Cp
8      for j in 0 .. n-1
9          for p in Ap(j) .. Ap(j+1) - 1
10             q = w(Ai(p))++
11             Ci(q) = j
12             Cx(q) = Ax(p)
13     return C
```

Figure 1.6: CSC Transpose Pseudo-code

$$C_{ji} = A_{ij} \qquad \forall\ 0 \leq i < m,\ 0 \leq j < n. \tag{1.1}$$

It can also be viewed as changing A from CSC to CSR format or vice-versa. This operation uses the strategy of pre-computing the resulting row lengths and row pointer array Cp in lines 4-7. The work array w first acts as a count of the number of elements in each row. During the copy loop in lines 8-12, w acts as a list of indices into the Ci and Cx arrays where the next element should go in each row. The for loops in lines 8-9 are a typical iteration idiom for a CSC matrix over every column j and every index p in that column. Lines 10-12 simply assign $C_{ji} = A_{ij}$ and update the work array.

The number and variety of sparse matrix software packages available reflects the importance of these algorithms and the intensity of research effort devoted to improving their performance. Some packages such as Sparsity focus on using dense blocking for lower level operations such as sparse matrix vector product (SpMV) and sparse triangular solve (SpTS), which can be used by direct or iterative solvers as a base kernel operation [22]. Other packages such as UMFPACK [10], WSMP [21] and SuperLU [12] offer specific implementations of direct LU decomposition which are optimized over the entire operation to take advantage of parallelism and high-speed dense BLAS [1] routines. Additionally, there have been recent efforts to create a uniform Sparse BLAS [17] interface which could be used to interface with a number of implementations. Generally, however, all of these packages suffer from relatively low floating-point efficiency due to the explicit indexing operations and unpredictable memory accesses associated with all sparse matrix algorithms.

1.4 Commodity Architectures

The superscalar, pipelined, multi-core commodity processors that are dominantly available in desktop computers, servers, clusters and even modern supercomputers have evolved over years to perform well on a wide variety of computing tasks. Sparse linear algebra algorithms are not among the operations that these processors perform efficiently. In contrast, dense linear algebra algorithms have a higher profile in benchmarks and perform extremely

well by taking advantage of architecture features that favor regular unit-stride memory access and independent operations. The indexing operations required to maintain sparse matrix data structures and the dependence of subsequent operator and memory access on indexing can cause sparse algorithms to poorly utilize hardware features that are optimized for dense algorithms.

Modern processors are heavily pipelined, containing anywhere from 4 to 31 stages, each taking a clock cycle, that an instruction may have to pass through before finishing [23]. Processor designs with more stages can reduce the total clock cycle period required to complete each stage and thus increase the overall clock speed. The program executing, however, must have enough instruction level parallelism (ILP), or independent operations to keep the pipeline full of useful instructions. Branch and memory load instructions can stall the pipeline and reduce the utilization of chip resources while later instructions must wait for these to finish before executing. Modern processors contain branch prediction units and complex caches to reduce the number of pipeline stalls. When branch results or memory accesses are unpredictable, as in the case of sparse matrix algorithms, these measures can fail to improve performance.

Effective use of the memory hierarchy is one of the most important factors in the performance of a program. As processor speeds have increased, so have memory speeds, but at a lesser rate. This has led to an ever increasing gap in access time between on-chip and off-chip memory. In general there is an inverse relationship between the time it takes to access memory from the CPU and how expensive it is. This has led to the common use of a multi-layer memory hierarchy as depicted in Figure 1.7. Small amounts of on-chip registers and memory are at the top of the hierarchy and larger, less expensive RAM and disk paging is lower. The access time for memory at lower levels can be one or several orders of magnitude longer than the layer above it. Complex caches have been developed to effectively utilize the upper layers of the memory hierarchy to keep data which has a high temporal or spatial locality to previous accesses. Algorithm implementations must be carefully tuned to take the most advantage of cache based on processor specific cache layout and behavior. Some of the fastest implementations of important algorithms such as the Discrete Fourier

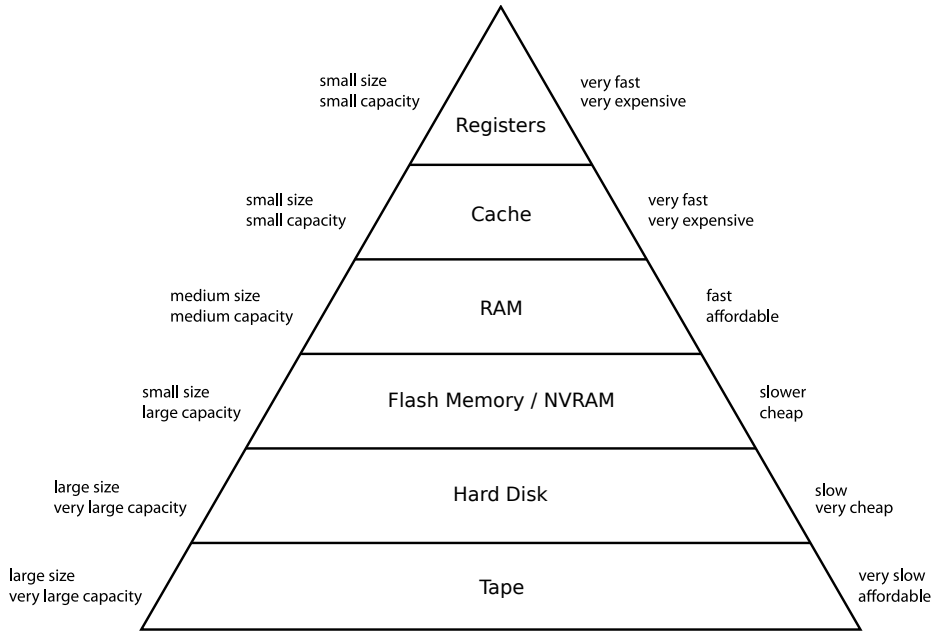


Figure 1.7: Computer Memory Hierarchy [34]

Transform (DFT) and dense BLAS search a large parameter space to find the fastest code for a target processor [28, 33].

In addition to adapting to a target architecture's cache and instruction pipeline, there are usually several levels of parallelism available that a program must take advantage of. Many processors are superscalar, allowing multiple instructions to be issued to the pipeline at once. This adds to the ILP required to effectively keep the pipeline full. SIMD (Single Instruction Multiple Data) parallelism is also common on PC-grade processors. Intel/AMD processors contain SSE instructions and IBM Power processors contain Altivec instructions to utilize short-vector floating point units. These instruction sets normally require explicit programming to take advantage of the 2, 4 or 8-way operations and the memory they operate on must be vector-aligned. To top it all off, chip speeds have plateaued and chip-makers are steadily increasing the number of cores in multi-core chips, requiring programmers to exploit multi-threaded, shared-memory or message passing parallelism.

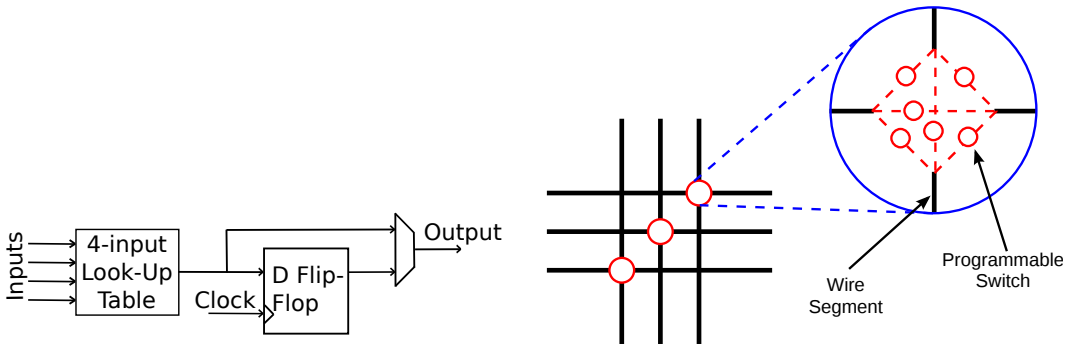
Using all of the above mentioned methods for fully utilizing on-chip resources, peak theoretical computation rates are commonly in the 10-100 GFlop/s range. On a recent

Intel Core i7 processor with 4 cores running at 3.2 GHz, the peak theoretical performance is roughly 70 GFlops/s [8]. Very few applications are actually capable of coming near this rate, however. The codes which have the highest floating-point rates are generally dense linear algebra routines which can take advantage of much of the hardware parallelism and cache hierarchy effectively. The GotoBLAS2 [19] library, which contains hand-written architecture-specific assembly code, achieves greater than 90% of peak floating point efficiency on a variety of architectures including Pentium 4, Opteron, Itanium2, PowerPC and Core2 [19]. Sparse linear algebra routines achieve a small fraction of this rate.

1.5 Field Programmable Gate Arrays

To avoid the architectural choices imposed by commodity chip makers, it is possible to design a custom co-processor for a particular application. Hardware designs that use a similar manufacturing process to CPUs, etching circuitry into a Silicon wafer, are referred to as Application Specific Integrated Circuits. ASICs have clock speeds and logic density similar to that of CPUs, but the design and fabrication process is much more expensive and time consuming than software development. As a middle-ground, Complex Programmable Logic Devices (CPLDs) and Field Programmable Gate Arrays (FPGAs) offer the flexibility of re-programmable logic at approximately 1/10th speed of CPUs and ASICs. Commonly, FPGAs are used as testbeds for new hardware designs destined for traditional fabrication, but the use of these devices as reconfigurable computing platforms has grown in popularity as the technology has matured.

FPGAs work by providing a large number of very small, simple logic units, called slices, which can be programmed to act like a variety of traditional logic gate configurations. The logic slices are connected in a mesh topology with each other and other basic resources such as Block RAM, high-speed multipliers, and I/O pins. Figure 1.8 shows an example of a programmable logic slice and the programmable routing resources that connect them. FPGAs can be connected to virtually any other digital circuits, but for the purposes of using them as co-processors, they can be connected to the CPU via a communications bus as well as some external memory dedicated to the FPGA.



(a) Basic Logic Block [35] (b) Routing Switch Box [36]

Figure 1.8: Programmable Logic and Routing in an FPGA

The largest FPGA manufacturer, Xilinx, produces chips as of this writing that contain 758,784 logic cells, 3.2 MB of on-chip Block RAM, 864 embedded DSP blocks, and are capable of running at 600 MHz [39]. There are a wide variety of configurations designed for different application areas, power requirements, and cost levels. Many chip manufacturers and 3rd party vendors make demo boards with these chips, but there are only a handful of vendors which provide FPGAs in a configuration suitable for use as a co-processor to software running on a standard CPU.

Because of the large amount of on-chip logic and Block RAM resources available for application logic, it is common to create streaming architecture models where data flows through a computation pipeline. This is aided by the heavy use of FIFO buffers to connect logic units together. Instead of operating on register memory, data can be read in from memory or CPU and processed at multiple stages throughout logic that the data passes through. This is effectively an alternative way to make use of instruction level parallelism, while decoupling the data from specific memory addresses.

Designing application specific hardware for an FPGA co-processor can be an expensive, time-consuming and error-prone task. The basic work flow for compiling and testing an FPGA design is shown in Figure 1.9. During the Synthesis and Place and Route steps, a hardware design must be mapped onto specific logic resources on the FPGA, and those units connected by signal routing resources. Because of this, compiling an abstract hardware

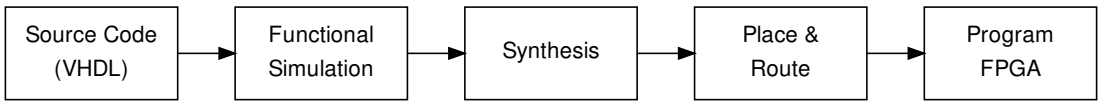


Figure 1.9: Basic FPGA Work Flow

design from Hardware Design Language (HDL) code to an FPGA programming file is very time consuming compared to software compilation. The electrical signals that travel over routing lines suffer from propagation delay. It is necessary to place and route logic resources such that no delay on the chip is longer than the clock cycle period. This place and route problem is NP-hard and a near optimal solution must be approximated with time-consuming algorithms to ensure that the logic works as intended [32]. Even the simplest designs can take several hours to implement into an FPGA programming file. Designs with very large combinatorial logic delays or difficult routing requirements may not meet timing constraints at all and require design changes.

This long implementation cycle time breaks the fast edit-compile-test-debug cycle that software programmers use to iteratively make progress on large projects. Software simulation of HDL code makes this cycle faster, but the event driven simulations are much slower, so can only be effective for short runs of the logic.

Another hindrance to the widespread use of FPGAs for reconfigurable computing is the lack of standardized libraries and platforms. Because of their predominant use as hardware test devices, FPGA boards commonly come with little or no interface logic for connecting to off chip RAM or even the CPU. Those devices that do come with supported software and hardware libraries use custom, proprietary ones that lack the widespread use to become stable and mature and prevent application code from interfacing with other vendor's platforms. The HDL languages that are commonly in use, VHDL and Verilog are very low level mechanisms for describing basic circuitry and lack advanced features such as Transaction Level Modeling [27]. Tools that support such features are proprietary and expensive, reflecting the overall investment usually applied to ASIC and other traditional hardware designs. Some tools that exist for applications such as digital signal processing (MATLAB/Xilinx

System Generator for DSP [xilinx.com]) provide cores for common operations and bus architectures for connecting cores that can induce a substantial performance overhead. All of the above factors make FGPA co-processor designs prohibitively difficult for software developers interested in accelerating a particular algorithm.

2. Sparse Gaussian Elimination and LU Hardware

This chapter contains a discussion of straightforward methods for solving a sparse system of linear equations. Basic right-looking and left-looking methods are are shown first without considering sparse data structures in sections 2.1 and 2.2. Implementations in software and hardware that use sparse algorithms are then discussed in 2.3 and 2.4.

2.1 Gaussian Elimination

Gaussian Elimination is a method for decomposing a matrix into lower and upper triangular factors by eliminating entries below the diagonal, one column at a time. It is typically taught in introductory linear algebra classes at the undergraduate level as a means of solving a system of equations represented by a matrix. Decomposing a matrix A into lower (L) and upper (U) triangular factors

$$LU = A \tag{2.1}$$

makes it possible to solve systems with multiple right-hand sides using a single LU decomposition step. Generally the input matrix is also permuted during this process when zeros are obtained as the pivot element.

Gaussian Elimination is considered a right-looking method because it uses matrix entries to the right of the current column being solved. This allows us to view Gaussian Elimination using the block recursive formula

$$\begin{pmatrix} l_{11} & \\ l_{21} & L_{22} \end{pmatrix} \begin{pmatrix} u_{11} & u_{12} \\ & U_{22} \end{pmatrix} = \begin{pmatrix} a_{11} & a_{12} \\ a_{21} & A_{22} \end{pmatrix}, \tag{2.2}$$

where L, U and A are $n \times n$ matrices, l_{11}, u_{11} and a_{11} are scalar values, l_{21} and a_{21} are column vectors, u_{12} and a_{12} are row vectors, and L_{22}, U_{22} and A_{22} are $(n-1) \times (n-1)$ block matrices. We obtain the following equations by block matrix multiplication of Equation 2.2.

$$l_{11}u_{11} = a_{11} \tag{2.3}$$

$$l_{11}u_{12} = a_{12} \tag{2.4}$$

$$l_{21}u_{11} = a_{21} \tag{2.5}$$

$$l_{21}u_{12} + L_{22}U_{22} = A_{22} \tag{2.6}$$

By convention, the diagonal entries of L are all set to 1. Equations 2.3 to 2.6 can be rearranged to solve for a column of L, a row of U and a $(n-1) \times (n-1)$ matrix which can be solved recursively.

$$l_{11} = 1 \tag{2.7}$$

$$u_{11} = a_{11} \tag{2.8}$$

$$u_{12} = a_{12} \tag{2.9}$$

$$l_{21} = a_{21}/u_{11} \tag{2.10}$$

$$L_{22}U_{22} = A_{22} - l_{21}u_{12} \tag{2.11}$$

The trivial base case occurs when the matrices are 1×1 and can be treated simply as the scalar values l_{11}, u_{11} and a_{11}. To form the matrix used in the recursive step, A_{22} must be updated with the outer-product $l_{21}u_{12}$. This sub-matrix update is the most computationally intensive part of each step. Consequently, it holds most of the focus for increasing the performance of this method. Running the update takes $O(n^2)$ at each of the n steps, resulting in a total asymptotic running time of $O(n^3)$ for this method in the case of dense matrices.

As a simple example of this method, the dense matrix in Equation 2.12 will be used to

```
1 for k = 1:n
2     U(k,k:n) = A(k,k:n)
3     L(k,k) = 1
4     L(k+1:n,k) = A(k+1:n,k) / U(k,k)
5     A(k+1:n,k+1:n) = A(k+1:n,k+1:n) - L(k+1:n,k) * U(k,k+1:n)
```

Figure 2.1: Dense Gaussian Elimination in MATLAB

illustrate the algorithm.

$$\begin{pmatrix} 2 & 3 & 4 \\ 4 & 5 & 6 \\ 6 & 7 & 9 \end{pmatrix} \tag{2.12}$$

The code in Figure 2.1 is a simple implementation of Gaussian Elimination in MATLAB code. The block formula from Equation 2.2 is tail-recursive and can easily be transformed into a non-recursive form similar to this code.

This code updates A in-place after every step. The contents of A can be copied to U and U used for submatrix-updates to keep the original matrix intact. For illustrative purposes and to facilitate discussion of our streaming model, we choose to modify A in-place. Equations 2.13 to 2.16 illustrate the code in Figure 2.1 on matrix 2.12 after every loop iteration.

$$L, U, A \tag{2.13}$$

$$\begin{pmatrix} 1 & & \\ 2 & & \\ 3 & & \end{pmatrix}, \begin{pmatrix} 2 & 3 & 4 \\ & & \\ & & \end{pmatrix}, \begin{pmatrix} 2 & 3 & 4 \\ 4 & -1 & -2 \\ 6 & -2 & -4 \end{pmatrix} \tag{2.14}$$

$$\begin{pmatrix} 1 & & \\ 2 & 1 & \\ 3 & 2 & \end{pmatrix}, \begin{pmatrix} 2 & 3 & 4 \\ & -1 & -2 \\ & & \end{pmatrix}, \begin{pmatrix} 2 & 3 & 4 \\ 4 & -1 & -2 \\ 6 & -2 & 1 \end{pmatrix} \tag{2.15}$$

$$\begin{pmatrix} 1 & & \\ 2 & 1 & \\ 3 & 2 & 1 \end{pmatrix}, \begin{pmatrix} 2 & 3 & 4 \\ & -1 & -2 \\ & & 1 \end{pmatrix}, \begin{pmatrix} 2 & 3 & 4 \\ 4 & -1 & -2 \\ 6 & -2 & 1 \end{pmatrix} \tag{2.16}$$

To improve numerical stability during LU decomposition, partial pivoting is typically used. Row-partial pivoting requires adding a row-row exchange at the beginning of every step of the decomposition. A row is selected to be swapped with row k during step k and is called the pivot row. The row is selected by comparing the numerical values in column k, the pivot column, in order to maximize U_{kk}, the pivot element. By maximizing the pivot element at every step, partial pivoting reduces the chance of dividing the other pivot column entries by a very small number. The row exchanges at every step lead to the decomposition in equations 2.17 and 2.18.

$$LU = PA \tag{2.17}$$

$$P^{-1}LU = A \tag{2.18}$$

Where P is an $n \times n$ permutation matrix containing a single 1 entry in every row and column. As an example, in the first step of the previous decomposition example, row 3 would have been selected as the pivot row because $A_{31} = 6$ is the largest element below the diagonal in the first column of A. The first row of P would then contain a 1 in P_{13} and be zero elsewhere.

2.2 Left-looking LU Decomposition

As an alternative to right-looking Gaussian Elimination, a left-looking factorization which relies on a sparse triangular solve can be used. Left-looking methods solve a single column of both L and U at each step using the columns of L that have already been solved to the left and a single column of A. The following block recursive formula 2.19 describes this method.

$$\begin{pmatrix} L_{11} & & \\ l_{21} & 1 & \\ L_{31} & l_{32} & L_{33} \end{pmatrix} \begin{pmatrix} U_{11} & u_{12} & U_{13} \\ & u_{22} & u_{23} \\ & & U_{33} \end{pmatrix} = \begin{pmatrix} A_{11} & a_{12} & A_{13} \\ a_{21} & a_{22} & a_{23} \\ A_{31} & a_{32} & A_{33} \end{pmatrix}, \tag{2.19}$$

where L, U and A are $n \times n$ matrices, u_{22} and a_{22} are scalar values, l_{32}, u_{12}, a_{12} and a_{32} are column vectors, l_{21}, u_{23}, a_{21} and a_{23} are row vectors, and the remaining entries are block matrices. We obtain the following equations by block matrix multiplication of Equation 2.19.

$$L_{11}u_{12} = a_{12} \tag{2.20}$$

$$l_{21}u_{12} + u_{22} = a_{22} \tag{2.21}$$

$$L_{31}u_{12} + l_{32}u_{22} = a_{32} \tag{2.22}$$

Equation 2.20 can be solved using a sparse triangular solve algorithm for the column of U. Equations 2.21 and 2.22 can then be solved with Sparse matrix-vector multiplication and vector scaling for the pivot element u_{22} and the column of L, l_{32}.

The CSparse [11] package uses this method for its straightforward sparse LU algorithm, but rearranges the computation into a single sparse triangular solve step for each column. The most computationally intensive part of this algorithm is the sparse triangular solve, which relies on a depth-first search of the pattern of L to determine the non-zero pattern of the new column.

The SuperLU [13] package is a fully-featured supernodal implementation of left-looking LU decomposition. This method and implementation are given a limited treatment here because other methods have been shown to have better performance on power system matrices [31].

2.3 Sparse Gaussian Elimination

When a significant portion of the entries in the matrix A are zero, it can be very advantageous to use sparse decomposition methods in terms of both memory and computation time. Sparse methods require changes to the storage of the matrices and the algorithm required to perform operations on only non-zero entries. Matrices are typically stored in compressed form, with both the floating point values and an integer index for each non-zero entry. Operations on compressed column or row vectors and matrices require comparison of the indices in addition to floating point operations on values. Additionally, the running time of some operations may change. For example, accessing column j of a matrix in dense form is an $O(1)$, constant time operation. In CSR format, accessing a column requires a linear search of each row for entries with indices matching column j. This requires time linear in the total number of non-zeros in the matrix, $O(nz)$.

During factorization of a sparse matrix, some updates may create non-zero entries where there previously was no entry. These additional non-zero entries are called fill-in, and they increase the density of the matrix, creating more computation at later steps of the decomposition. To improve performance, a fill-reducing ordering algorithm is often used to permute the matrix before factorization. Finding the reordering which results in the minimum fill-in is an NP-hard problem, but approximation algorithms can be used to find very good orderings in a short amount of time relative to the rest of the factorization [11]. Fill-reducing pre-ordering algorithms are outside the scope of this work. The Approximate Minimum Degree (AMD) algorithm proposed by Amestoy, Davis and Duff[2] and implemented by Timothy Davis as part of the SuiteSparse package is used uniformly prior to numerical factorization for the matrices presented here.

Our implementation of sparse LU decomposition used for performance analysis in this

```
1  typedef struct gs_matrix {
2      int n;       /* number of rows and columns */
3      int *r;      /* row counts */
4      int *j;      /* row-major indices */
5      double *x;   /* row-major values */
6      int *c;      /* column counts */
7      int *i;      /* column-major indices */
8  } gs;
```

Figure 2.2: C struct for Storing A During Gaussian Elimination

work and as the basis for special purpose hardware in the LUHW design first reported by Petya Vachranukunkiet in [30] is based on straightforward Gaussian Elimination. Like the MATLAB code above, A is modified in-place and the factors L and U are output a column and row at a time, respectively. Accesses to the pivot column and pivot row are therefore linear and local to the current step being worked on. The L and U factors are output simply in CSC and CSR form using the CSparse library's structure for compressed matrices. Access to A for the update portion of each step dominates the rest of data accesses. In our implementation, A is stored primarily in CSR format with arrays for column indices and values for each entry. Rows are padded with unused space such that each row starts at a regular stride offset in the index and value arrays. If the row-stride is 128, then a maximum of 128 non-zero entries can exist in each row and each row i starts at entry $(i << 7)$. Row-strides of a power of two make calculating the start of a row fast and convenient in hardware. Rows contain unused padding space to allow for fill-in during sub-matrix updates. Because rows may contain fewer than the maximum number of entries, an array containing the length of each row is also maintained. Entries within a row must be stored contiguously starting at the row offset in ascending order of column index.

To make pivot search faster, the column-major non-zero pattern of the matrix is also kept in the same strided format, but without numeric values and without the requirement that entries be sorted. In the software version of Gaussian Elimination, A is stored in the C struct described in Figure 2.2.

```
1  gaussianLU(A)
2      for k in 0 .. n-1
3          pivot_row, pivot_column, pivot_value
4              = pivot_search(A, k)
5          U(k,k) = pivot_value
6          U(k,k+1:n) = pivot_row
7          L(k,k) = 1
8          L(k+1:n,k) = pivot_column / pivot_value
9          for li,lx in L(k+1:n, k)
10             A(li,:) = merge(A(li,k+1:n), pivot_row)
```

Figure 2.3: Gaussian Elimination Pseudo-code

Where n is the dimension of the matrix, r and c are arrays of length n containing the length of rows and columns, respectively. The arrays j and x make up index and values for the strided compressed-row form of the matrix. The array i contains row indices in the strided compressed-column format, which we refer to as column-mapping or colmap.

Using the matrix data structures described above, we perform LU decomposition one column and row at a time with 3 phases in each step.

1. Pivot Search
2. Output of Pivot Row and Column
3. Sub-matrix update

The main loop performing these operations is shown in pseudo-code in Figure 2.3. The pivot search in lines 3-4 finds the row that will become the pivot row and extracts the pivot column values from the compressed-row storage. The result output phase in lines 5-8 copies the pivot row to row k of U and divides the pivot column by the pivot value before outputting it as column k of L. The submatrix update phase in lines 9-10 loops over every non-zero element in the scaled pivot column. The pivot column indices indicate rows that must be updated in the remaining sub-matrix $A_{k+1:n,k+1:n}$. Each row that must be updated has the scaled pivot row subtracted from it in the merge operation.

The pivot search operation finds an appropriate row to use as the pivot row by looking

for the largest value in the pivot column. This search is facilitated by the colmap data in A which contains non-zero entries for each column. Each entry in the pivot column is checked against the P^{-1} vector to see if it's row has already been used as a pivot row. If it has, then that row is above the diagonal and doesn't need to be considered. Otherwise, the pivot value is read from the first entry of its row's value array. The largest pivot value and index (pivot row number) are kept and updated throughout the search. If no valid pivot was found, then the matrix is singular and LU decomposition is unable to continue.

The merge operation is the most important step to consider when evaluating performance because it will dominate the runtime of the rest of the straightforward Gaussian Elimination algorithm. The merge operation is a sparse row-vector add operation that updates the structure of A in-place. Merge is linear in the number of non-zeros in each row and it gets run $O(Lnz)$ times, where Lnz is the number of non-zero elements in the factor L. To update a row of A in-place, merge first copies the row to work arrays wi and wx. It then merges the scaled pivot row with the row in the work arrays, one element at a time. Comparing the column-indices of w and the pivot row can result in 3 cases and their resulting operations. If the pivot row index and sub-matrix row index are equal, the scaled pivot-row value is subtracted from the sub-matrix value in wx and the index and new value are output to the row's location in A. If the sub-matrix row index is less than the pivot row index, the operation is a simple copy, and the original sub-matrix row index and value are copied to the output row in A. If the current pivot row index is less than the sub-matrix index, this is a fill-in and the pivot index and value are copied into the output row of A.

Because the merge operation contains the inner-most loop of the Gaussian Elimination algorithm, and because it contains unpredictable, data-dependent branches, this operation can suffer from poor throughput. On modern, heavily pipelined super-scalar processors with branch prediction, each loop iteration can cause a branch mis-prediction and stall the pipeline for the next iteration. An analysis of this operation's performance is covered in Section 5.3.

2.4 LU Hardware

To improve the overall performance of sparse LU decomposition on power system matrices, an application specific co-processor was designed and implemented on an FPGA platform as first reported in [30]. The following description of the hardware appears in [6].

Our FPGA based sparse LU hardware implements a row-wise, right-looking method of Gaussian elimination with row partial pivoting. To maximize performance, the design of the sparse LU hardware focuses on maintaining regular computation and memory access patterns that are parallel and fully pipelined wherever possible. Synchronous First-In-First-Out buffers implemented with embedded memory blocks are used for high speed buffering of data words throughout the pipelined design. A separate column-oriented mapping (colmap) of the non-zero structure of the matrix reduces pivot search from $O(n^2)$ to $O(n)$ time. The empirical study of power system matrices in [31] provides parameters for the hardware design such as cache line size, total cache size, and buffer depths, minimizing the need to handle more general cases and error conditions with extra logic.

A high level diagram of the sparse LU hardware implementation and basic data flow is depicted in Figure 2.4. The design of the hardware can be broken down into four main partitions. A central control, implemented as a state machine, tracks the progress of the functional units to ensure synchronized operation. The pivot logic and sub-matrix update logic implement the necessary computations required for sparse LU decomposition. The last partition, cache, handles sparse matrix data retrieval and storage for the pivot search and sub-matrix update.

Not shown are the external memory interfaces to the Sparse LU Hardware, which depend on the FPGA prototype board used for implementation. Our design assumes independent memory banks for the units which require access to external memory such as SDRAM. The colmap utilizes one memory interface to store a column-wise representation of the sparse matrix structure for fast pivot search capability. The cache utilizes another memory interface to store a row-wise representation of the sparse matrix in compressed form. Having two separate memory banks and controllers allow concurrent operation for the colmap and

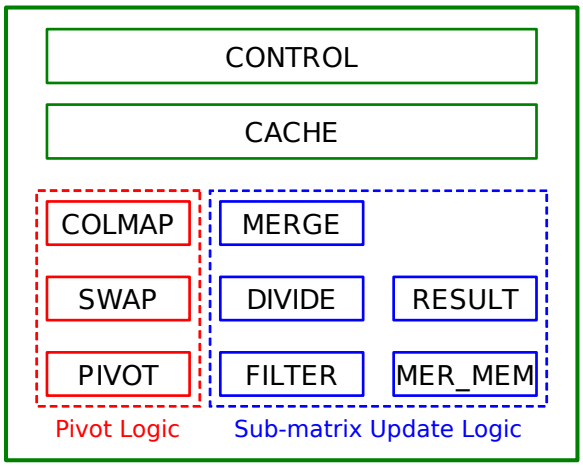


Figure 2.4: Top Level Sparse LU Hardware Block Diagram

cache units.

A detailed diagram of the pivot search logic and the submatrix update logic is depicted in Figure 2.5. The logic to perform the pivot search consists of three units, referred to as **colmap**, **swap**, and **pivot**. The pivot selection algorithm used in the hardware design is row partial pivoting based solely on numerical criteria and does not perform any analysis for potential fill-in reduction. The **pivot** logic performs a search, element by element, of the current column for the LU decomposition. The highest magnitude element is selected as the pivot element. In our hardware rows are not swapped physically, instead a record of whether each row has been used as pivot rows is maintained.

The **colmap** unit first performs a burst read of the column-wise matrix representation to form the pivot column. The **swap** unit maintains a record of the pivoting operations that have occurred. This is used to reject candidate rows from the colmap which have already been eliminated. Rows which are not rejected are sent to the cache read queue as single word read requests. The **pivot** unit compares pivot column values returned from cache, selecting the element with the highest floating point magnitude as the pivot element. Once the exhaustive search of the pivot column is complete the swap unit updates the row mappings and the sub-matrix update can begin.

The sub-matrix update logic has two main computations, the normalization of the pivot

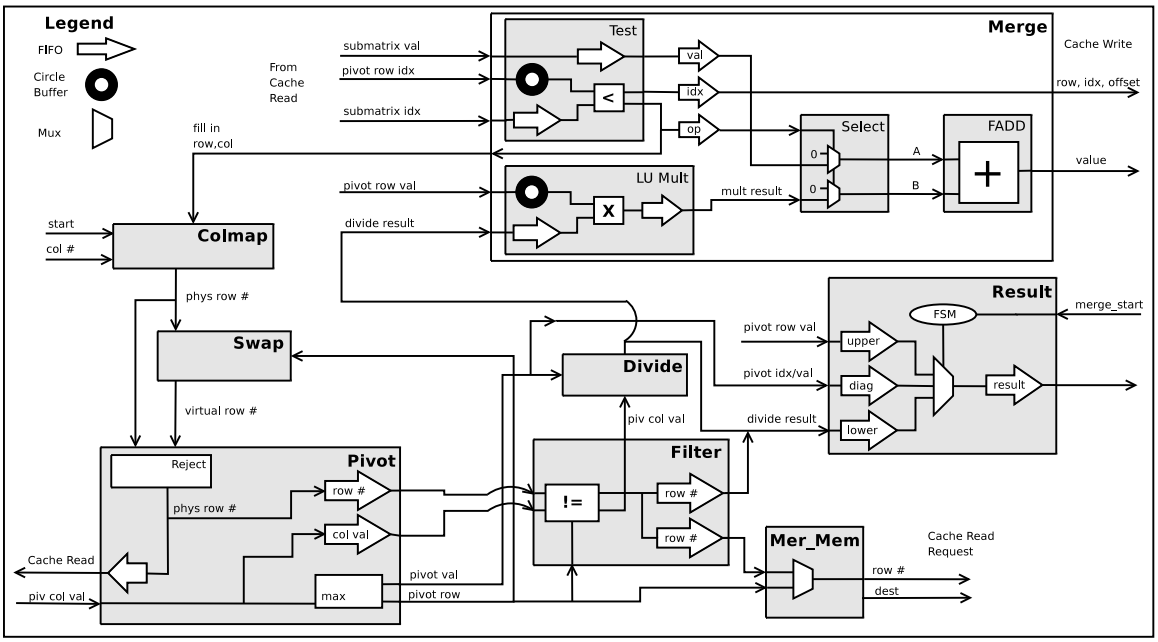


Figure 2.5: Pivot and Sub-matrix Update Logic

column by the pivot element, followed by a reduction of the remaining sub-matrix by the product of the pivot row and the pivot column. The **filter** unit feeds the pivot column elements to the divide unit to be normalized. The **mer_mem** unit handles cache requests and schedules computation for row updates by the merge unit(s). The **result** unit records the pivot element, pivot row, and normalized pivot column as parts of the final L and U matrices.

The **merge** unit performs three tasks in parallel which make up the bulk of computation. The first is calculating the product of the pivot row and an element of the normalized pivot column. The second is a comparison of the pivot row indices to the sub-matrix row indices to determine the non-zero structure of the reduced row. Finally, the scaled pivot row and sub-matrix row are merged into the new non-zero structure as operands to the floating point addition unit. Additional parallelism is possible by increasing the bandwidth to the cache and instantiating multiple merge units to allow row reductions in parallel. Once the sub-matrix update has completed the hardware signals the control unit so the pivot logic can begin the search for the next pivot element.

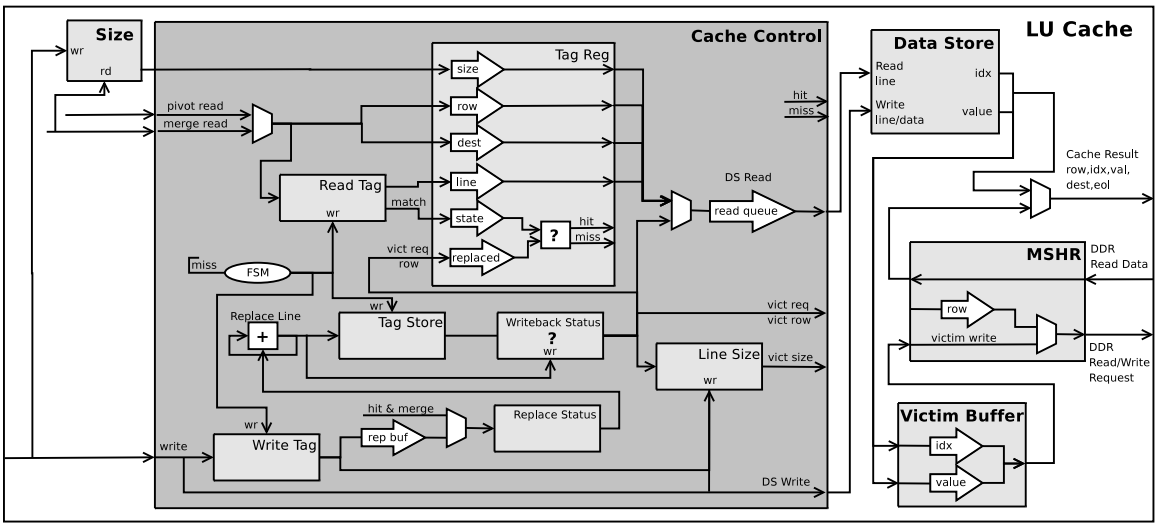


Figure 2.6: Special Purpose Cache

The use of a memory hierarchy consisting of one or more levels of cache has been used for quite some time in order to address the growing disparity between memory performance and the performance of high speed logic. The use of a cache for our FPGA based Sparse LU Hardware is two fold. The first is to reduce the latency of memory read operations and therefore idle cycles where computations could occur. The second reason, and perhaps most important, is to supply the merge unit with enough scalable read/write bandwidth for high performance.

A detailed diagram of the special purpose cache is depicted in Figure 2.6. The cache design is single level and utilizes the embedded FPGA memory blocks for cache data storage and tag data arrays. The cache policy is write-back with read miss allocation and a modified First-In-First-Out (FIFO) replacement policy. The cache is fully associative and stores entire compressed matrix rows to allow high speed constant burst read/write operations. The tag array logic uses content addressable memory (CAM) functionality based on [38] to look up a cache line from a matrix row number. Additional logic guarantees that no rows in-process will be replaced and all writes will be a cache hit.

This cache is an example of where the reconfigurable nature of the FPGA can allow application specific design for performance enhancements tailored to a specific algorithms

Table 2.1: Sparse LU Hardware Performance Model Parameters

Parameter	**Description**
CACHE_ROWS	Number of rows in cache
DEFAULT_FREQ	Default frequency
MASKED_PIVOT	Pivot thresholding by masking mantissa
FORCE_DIAG	Force diagonal pivot selection
PARTIAL_PIVOT	Partial pivoting
PIVOT_THRESHOLD	Threshold for partial pivoting
PROC_SCHED	Scheduling algorithm for multi-merge
NUM_PROC	Number of merge units
CLOCKS_COLMAP	Request to valid data
CLOCKS_COLMAP_PERWORD	Rate of LLRAM read
CLOCKS_SDRAM	Request to valid data
CLOCKS_SDRAM_PERWORD	Rate of DDRAM read
SDRAM_WIDTH	Width of transfer (index+value pairs)
HIT_TO_MISS	Situational cache latency
HIT_TO_HIT	Situational cache latency
MISS_TO_HIT	Situational cache latency
MISS_TO_MISS	Situational cache latency
FMUL_TO_FMUL	Multiply to multiply cycles
ROW_TO_ROW	Row to row merge cycles
CLOCKS_CACHE	Cache request to data out
CLOCKS_FMUL	FP multiply latency
CLOCKS_FADD	FP add latency
CLOCKS_FDIV	FP divide latency
CLOCKS_TRANSLATE	Pivot translation
CLOCKS_PIVOT	Latency pivot search/compare

data access requirements. Simulation results show that our cache design results in a row read hit rate of ∼85% (word read hit rate over 90%) including compulsory misses; all row writes are hits as previously mentioned.

A software performance model was written that simulates the exact operation of the LUHW design. The model counts cycles, floating point operations, cache hits and misses and other statistics during its operation on a matrix. The model is also parameterized with key hardware parameters such as row sizes, fifo depths, and cache size. The combination of parameterization and feedback from the performance model allow rapid design exploration without the need for the lengthy FPGA synthesis, place and route process. A list of the performance model parameters appears in Table 2.1

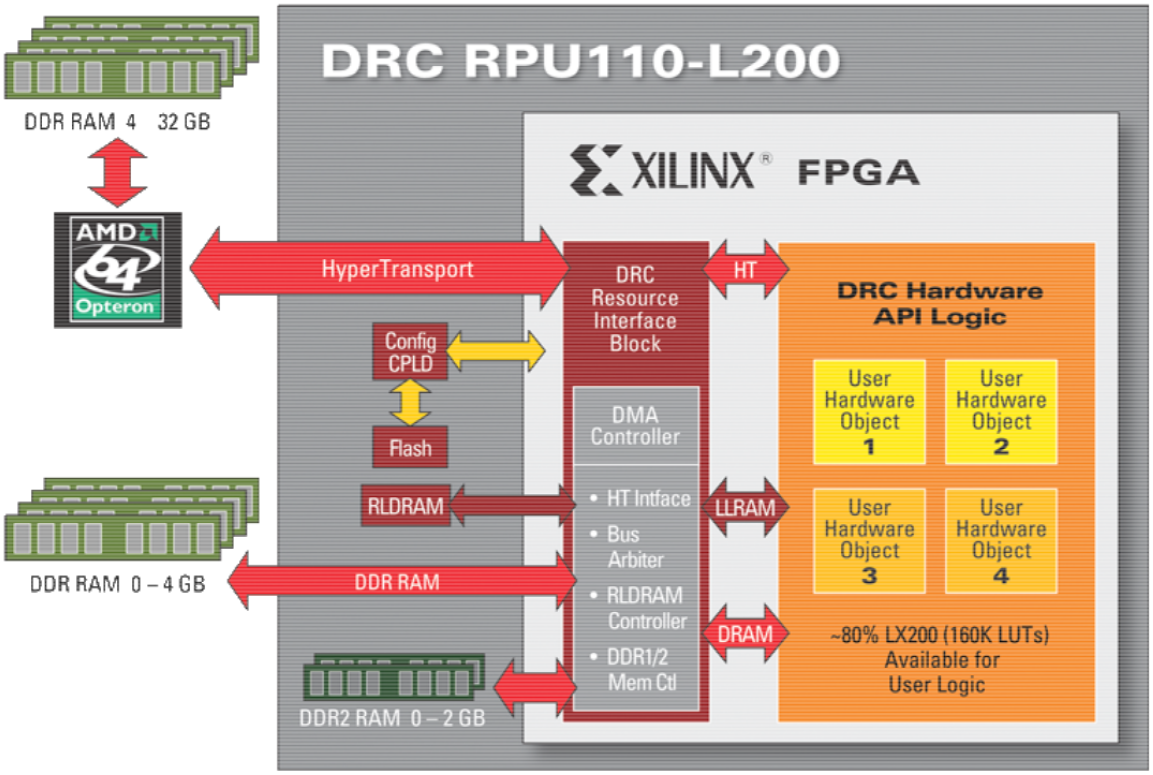


Figure 2.7: DRC Architecture [7]

The design has been implemented on a DRC Computer RPU110-L200 module containing a Xilinx Virtex 4 LX200 FPGA with more than 200,000 logic cells, 756 KB on chip BRAM, 128 MB RLDRAM, 2 GB DRAM, and Hyper-transport connection to an Opteron host processor. A high-level diagram of the DRC module's architecture and connectivity to CPU and DRAM is shown in Figure 2.7. The large amounts of block RAM and external memory available to the FPGA allows the Sparse LU hardware to target a 26,828 bus system used in industry[1]. The prototype has been verified up to a 10,278 bus system at 133 MHz. Table 2.2 details the Virtex4 FPGA resource usage for the Sparse LU Hardware capable of processing the 26,828 bus system. Available memory for caching and buffering is the principal limitation when running the hardware on larger power system matrices.

The number of clock cycles required to perform the LU decomposition for the FPGA

[1]provided by PJM (pjm.com)

Table 2.2: Sparse LU Hardware Resource Utilization

	Logic Slices	DSP48 Blocks	16kb RAM Blocks
Usage	27,927	4	293
Percent of Available	31%	4%	87%

based hardware was measured using a hardware counter that increments every clock cycle during LU decomposition. This hardware cycle count is used to verify the accuracy of the software performance model for the Sparse LU architecture.

To interact with the power flow software that requires the accelerated LU decomposition, a library was written to interact with the LUHW from software running on the connected Opteron processor. The Opteron processor runs a standard Linux operating system with additional drivers required to interact with the DRC processor module. The LUHW library uses lower-level DRC library routines to interact with the DRC module and LUHW on the FPGA over the Hyper-Transport bus. Routines exist to load a matrix from standard compressed form into the LUHW, start the LUHW, check status and retrieve results. Using this library, a complete power flow application can be run with the assistance of the LUHW co-processor.

The LUHW design is parameterized to meet the requirements for decomposing particular power system matrices. Because these parameters are statically assigned at hardware implementation time, the LUHW has limitations on what matrices it can successfully decompose and the speed at which it operates. The current LUHW design has been optimized to handle matrices up to $n = 65{,}536$, with a maximum of 256 non-zero elements in each row. The cache is configured to hold 128 rows. Also, due to routing limitations within the LUHW and interfacing with the DRC connectivity logic, the speed of the chip is currently limited to 133 MHz.

3. Multifrontal Methods

This chapter contains a discussion of multifrontal methods for solving a sparse system of linear equations [11]. The elimination tree structure is discussed first in Section 3.1 followed by a description of multifrontal and supernodal algorithm operation in Section 3.2. The performance implications of using multifrontal and supernodal methods are discussed in Section 3.3.

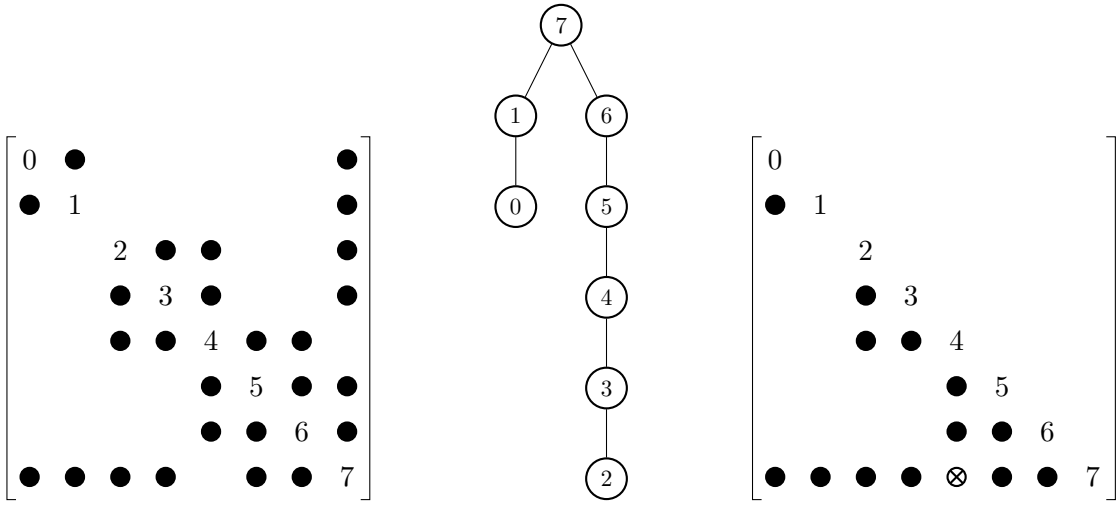
3.1 Elimination Tree

Multifrontal and left-looking decomposition methods as well as many other sparse matrix algorithms rely on the elimination tree of a matrix as a tool for understanding the non-zero structure of a matrix during computation. The elimination tree is computed during a symbolic analysis phase prior to numeric factorization of a matrix. The tree is a subgraph of $Reach_A$, the reachability graph of the matrix A, where the edge $(i, j) \in Reach_A$ if and only if there is a path $i \rightsquigarrow j$ in the graph of A through nodes numbered less than i and j. The reachability graph is a common component for the non-zero pattern of factorization results, so the elimination tree is an efficiently computable and traversable representation of this graph. The tree is defined by the statement

$$\begin{aligned} &\forall i < j \in G_A \\ &\qquad j = P(i) \\ \text{iff } \nexists k \in G_A \text{ s.t. } &\; i < k < j \;\wedge\; (i,k) \in Reach_A \end{aligned} \tag{3.1}$$

That is, j is the parent of i in the elimination tree if it is the least numbered node connected to i in $Reach_A$. The elimination tree for the 6-bus Jacobian matrix along with the matrices non-zero pattern and the non-zero pattern of its factor L are shown in Figure 3.1

To calculate the elimination tree, the CSparse routine `cs_etree` traverses the matrix A in nearly linear time. For each non-zero entry A_{ik} in the matrix, it follows a path in the



(a) Matrix Non-Zero Pattern (b) Elimination Tree (c) L Non-Zero Pattern

Figure 3.1: 6-bus Matrix

current tree from node i to the root of the tree, then sets the parent of the current tree's root to k. The algorithm uses path compression during this traversal to update an ancestor array to point to k, the greatest known ancestor so far. Using path compression makes this algorithm very nearly linear in $|A|$, the number of non-zeros in A [11].

The elimination tree is used in a variety of algorithms to compute the resulting non-zero pattern of an operation. During sparse triangular solve, the goal is to solve for a sparse vector x in the equation

$$Lx = b \tag{3.2}$$

where L and b are also sparse. The non-zero pattern of x is determined by $Reach_L(b)$, the set of vertices reachable in the graph of L starting at vertices in b. This pattern can be discovered by a depth-first search of L starting at the non-zero entries of b. Using the elimination tree, however, the non-zero pattern of x is determined by traversing the path to the root of the tree from every node i in b. Using this inexpensive traversal, it is also possible to accumulate column or row counts before computation so that a CSC or CSR result matrix can be compiled out-of-order. During multifrontal factorization, the elimination tree is used to determine dependence between operations.

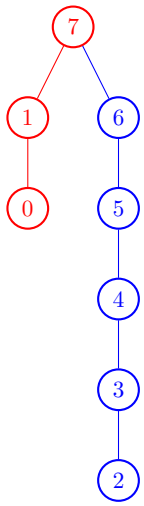


Figure 3.2: Elimination Tree Disjoint Paths

The elimination tree can be broken up into disjoint paths, also referred to as chains by the multifrontal algorithms. Each path contains a series of consecutive parent links that are not shared with any other path. Paths end at either a node which is part of another path or at the root of the tree. A set of disjoint paths for the 6-bus matrix is shown in Figure 3.2 with different colors for each path.

3.2 Basic Multifrontal Technique

The multifrontal method for decomposing matrices originated from Duff and Reid's work in [16] based on the frontal method introduced by Irons [24]. The technique was additionally popularized by a tutorial written by Liu [26], which is a more accessible introduction to the method. Davis extended the method to work on pattern-unsymmetric matrices and implemented the well known high-performance UMFPACK implementation [10]. The WSMP software also uses this method to parallelize sparse LU decomposition using the independent operations represented in the elimination tree.

Frontal LU methods are right-looking like Gaussian Elimination, updating sub-matrices with the outer-product of a pivot column and pivot row at every step. This outer-product matrix is the frontal matrix for a particular step, named for the graph front it represents.

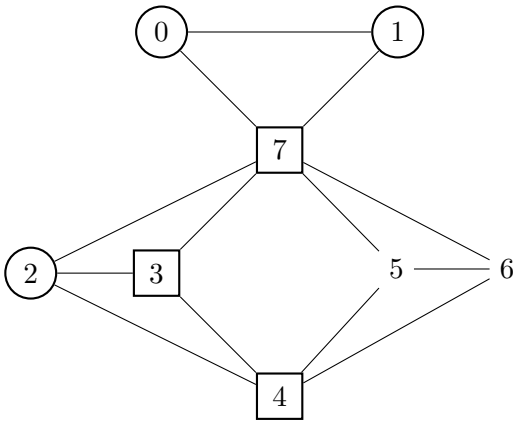


Figure 3.3: Graph Front During Factorization, $k = 2$

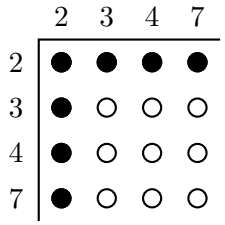


Figure 3.4: Frontal Matrix 2

The graph front is the set of nodes reachable in one step from the previously visited set of nodes, and identifies the border between the part of the system that has been solved and the part that has not. In Figure 3.3, the square nodes are the front reachable from the previously visited nodes during step 2 of decomposing the 6-bus matrix. The frontal matrix contains only the entries that are updated by the outer-product of the pivot row and pivot column. The frontal matrix entries are gathered together into a dense matrix along with the pivot row and column, as shown in Figure 3.4.

Multifrontal techniques use the elimination tree to order all the computation required during LU decomposition into a series of frontal matrix operations. Each node in the elimination tree corresponds to a frontal matrix where the node label is the diagonal pivot entry. Figure 3.5 shows an example of frontal matrices organized around the elimination tree for the 6-bus matrix. The lower-right block of each frontal matrix shown with open circles is

the contribution block for the matrix. These values are obtained by the outer product of the pivot row and column from each matrix. Each contribution block is used by other frontal matrices that are ancestors in the elimination tree, requiring that the contribution elements be added to elements in the parent matrix before computing with the parent matrix.

The assembly of contribution block elements into parent matrices requires that the indices of the matrices be maintained separately from the dense storage for the matrix itself. Each frontal matrix may have multiple children with contribution blocks that must be added together. If frontal matrices are computed using a post-order traversal of the elimination tree, then a stack of frontal matrices from children can be kept until all child matrices are finished computing and the parent's frontal matrix can be assembled. Additionally, if the tree is split into disjoint paths, or chains, then a single memory space can be allocated to hold all frontal matrices in a chain. The chain's dense storage space must be large enough to hold the largest frontal matrix on the chain.

To exploit parallelism, multiple frontal matrices can be computed concurrently. The elimination tree relationship between frontal matrices describes the data dependence for computation. Matrices can be computed independently until their least common ancestor matrix is encountered and requires data from its children. WSMP uses this technique to distribute independent matrices to multiple processors working together to decompose a large matrix. To utilize SIMD or ILP parallelism, supernodal techniques can be used to process multiple pivot rows/columns with the same pattern within the same frontal matrix. In this case the contribution block is calculated as a matrix multiplication instead of simple vector outer-product. Figure 3.6 shows the supernodal assembly tree for the 6-bus matrix.

3.3 Performance Implications of Multifrontal Methods

The most obvious performance advantage of the multifrontal method is the ability to use dense BLAS kernels to compute the contribution blocks in frontal matrices. Computation is split among many small matrices, which limits the peak performance achievable by the dense BLAS routines, but for large sparse matrices, the average frontal matrix size can be fairly large. Indexing information is used only during the assembly of frontal matrices, so

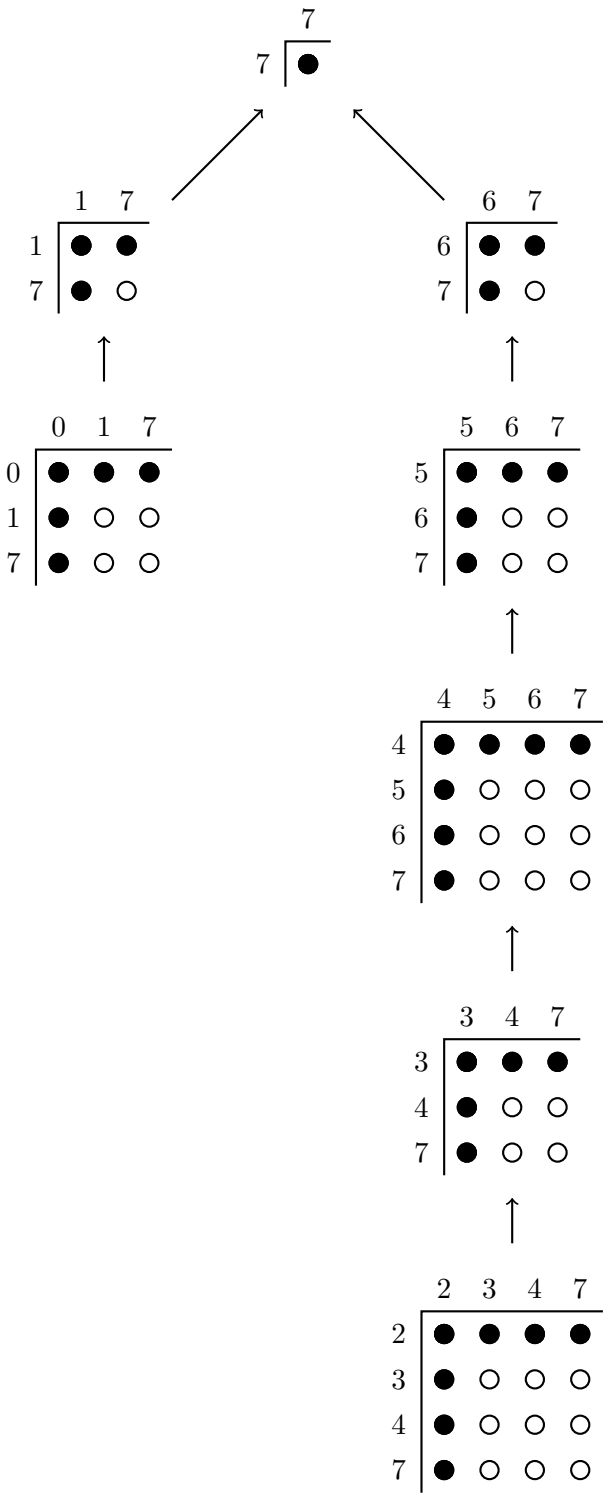


Figure 3.5: Assembly Tree for 6-bus Matrix

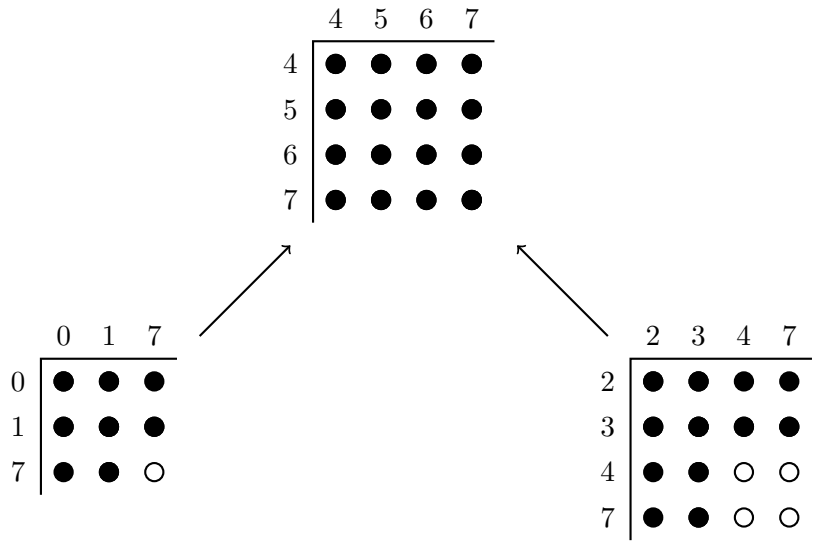


Figure 3.6: Supernodal Assembly Tree for 6-bus Matrix

floating-point computation is not dependent on a recently calculated indexing operation. The indexing operations that copy or add contribution elements to a frontal matrix are not interdependent, so they can take advantage of ILP.

The multifrontal method also benefits from temporal and spatial locality when accessing frontal matrices. For each frontal matrix being constructed, it obtains contribution values from its children, one of which was the last frontal matrix worked on during post-order traversal. The dense BLAS routines used to compute the outer-product on each matrix are also carefully tuned to take advantage of cache locality by using blocked matrix algorithms. The disadvantage is that more memory must be used to store the frontal matrices separately from the input and result matrices. This extra memory usage negatively impacts cache behavior by increasing the overall memory footprint that must be cached.

It also must be acknowledged that assembly of the independent frontal matrices requires additional floating point operations to add contribution elements together into parent frontal matrices. In other forms of LU decomposition, these contribution values could be added to the existing value in the matrix at the time of computation instead of after. These assembly steps create a significant amount of additional data movement which is not present in other

algorithms

The additional storage requirements for frontal matrices also require allocation and deallocation of memory during the decomposition process. Frequent calls to `malloc` and `free` can cause significant operating system overhead for a program. This has a significant enough impact on performance that the author of UMFPACK suggests using an alternative high-performance version of `malloc` [18].

The performance advantages gained by using high-speed dense BLAS routines have to be balanced against the additional overhead of using frontal matrices. Because sparse algorithm performance is highly dependent on the matrix structure being worked on, this balance can vary from matrix to matrix. This relationship is investigated for power system and other matrices in Section 5.2.

4. Benchmark Matrices

This chapter presents the benchmark matrices used for the performance experiments in Chapter 5. Section 4.1 describes matrices from the power system analysis domain and Section 4.2 describes matrices that were chosen from other domains for comparison.

4.1 Power System Matrices

The power system matrices used in this work were obtained from PJM Interconnection, the regional transmission authority for Pennsylvania and surrounding states. The matrices are Jacobian representations of the original Y_{bus} matrices and have been pre-ordered to reduce fill-in with the AMD [2] algorithm. Generally, power matrices are very sparse, with moderate size and do not contain any regular patterns. Table 4.1 lists the power matrices and some of their statistics. Table 4.2 lists some of the matrix properties after LU decomposition. Figure 4.1 shows the non-zero pattern of one of the power matrices.

4.2 Comparison Matrices

A selection of 15 matrices from the University of Florida Sparse Matrix Collection[9] is used for performance comparison in this work. These matrices came from a variety of domains including structure analysis, computational fluid dynamics, and circuit simulation. These matrices were selected to be a similar size as the power matrices (100,000 to 150,000 non-zeros) and represent a wide range of applications and patterns. Table 4.3 lists the

Table 4.1: Power Matrix Properties

Matrix	**n**	**nz**	**Sparsity**	**Avg. nz per row**
jac2k	2,982	21,196	0.238 %	7.1
jac7k	14,508	105,522	0.050 %	7.3
jac10k	19,285	134,621	0.036 %	7.0
jac26k	50,092	351,200	0.014 %	7.0

Table 4.2: Power Matrix LU Properties

Matrix	Lnz	Unz	Fill Ratio	L+U nz per row	MFlop Count
jac2k	23,927	23,894	2.12	15.04	0.5
jac7k	123,109	122,955	2.19	15.96	3.5
jac10k	135,708	134,905	1.87	13.03	3.1
jac26k	430,462	403,876	2.23	15.66	21.0

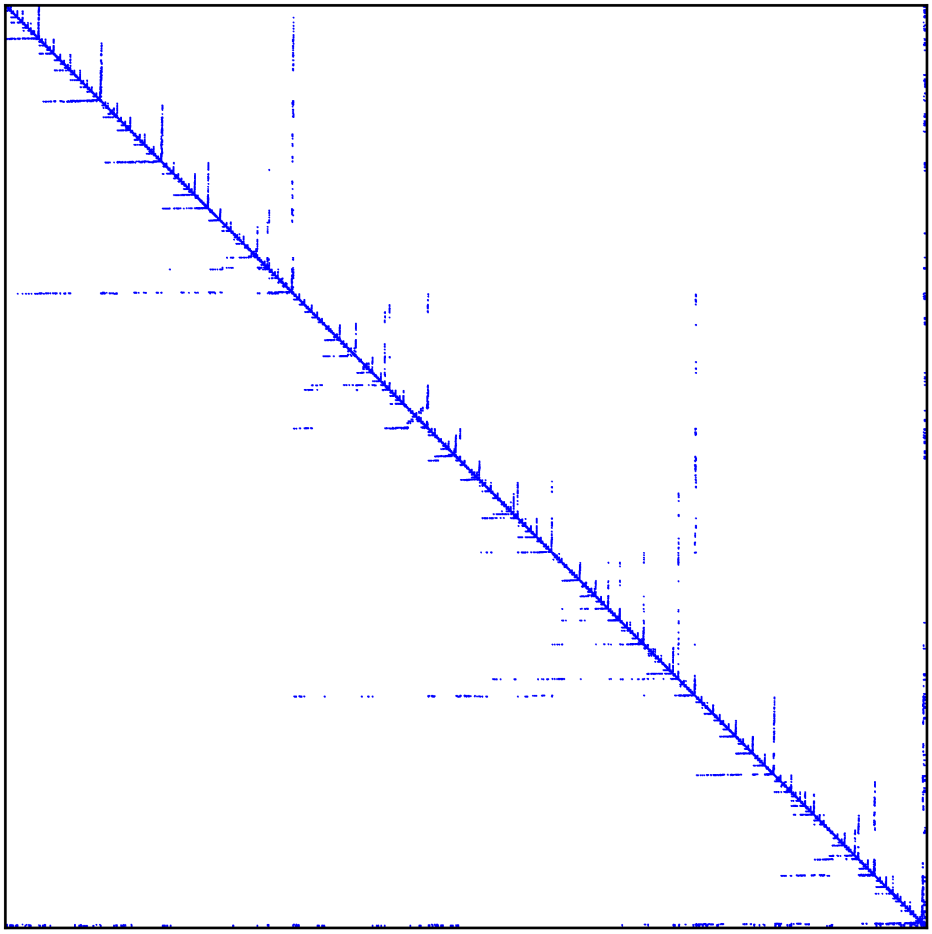


Figure 4.1: jac26k Non-zero Pattern

Table 4.3: Comparison Matrix Properties

Matrix	**n**	**nz**	**Sparsity**	**Avg. nz per row**	**Application**
piston	2,025	100,015	0.024 %	49.4	model reduction
ford1	18,728	101,576	0.000 %	5.4	structural
c-41	9,769	101,635	0.001 %	10.4	optimization
mhd4800a	4,800	102,252	0.004 %	21.3	electromagnetics
shuttle_eddy	10,429	103,599	0.001 %	9.9	structural
nasa4704	4,704	104,756	0.005 %	22.3	structural
crystm01	4,875	105,339	0.004 %	21.6	materials
mark3jac040	18,289	106,803	0.000 %	5.8	economic
bloweybl	3,003	109,999	0.012 %	36.6	materials
cvxqp3	17,500	114,962	0.000 %	6.6	optimization
bcsstk15	3,948	117,816	0.008 %	29.8	structural
bodyy4	17,546	121,550	0.000 %	6.9	structural
aft01	8,205	125,567	0.002 %	15.3	acoustics
igbt3	10,938	130,500	0.001 %	11.9	semiconductor
stokes64	12,546	140,034	0.001 %	11.2	fluid dynamics

comparison matrices and some of their statistics and origins. Table 4.4 lists some of the comparison matrix properties after LU decomposition. Figures 4.2 through 4.7 show the non-zero pattern of some of the comparison matrices.

Table 4.4: Comparison Matrix LU Properties

Matrix	Lnz	Unz	Fill Ratio	L+U nz per row	MFlop Count
piston	68,442	68,442	1.35	66.60	5.0
ford1	908,945	428,152	12.98	70.40	117.6
c-41	364,024	269,242	6.13	63.82	60.2
mhd4800a	153,553	210,001	3.51	74.74	15.9
shuttle_eddy	91,070	119,631	1.93	19.20	2.9
nasa4704	283,225	283,225	5.36	119.42	72.3
crystm01	326,955	326,964	6.16	133.14	61.7
mark3jac040	2,212,336	3,146,694	50.01	292.02	4200.5
bloweybl	70,002	70,001	1.00	3.67	0.0
cvxqp3	5,703,962	10,977,667	144.95	952.24	20185.1
bcsstk15	614,587	614,587	10.40	310.34	292.8
bodyy4	572,607	572,607	9.28	64.27	101.5
aft01	289,180	289,180	4.54	69.49	33.1
igbt3	604,996	654,567	9.57	114.15	112.8
stokes64	804,168	1,123,022	13.67	152.61	293.7

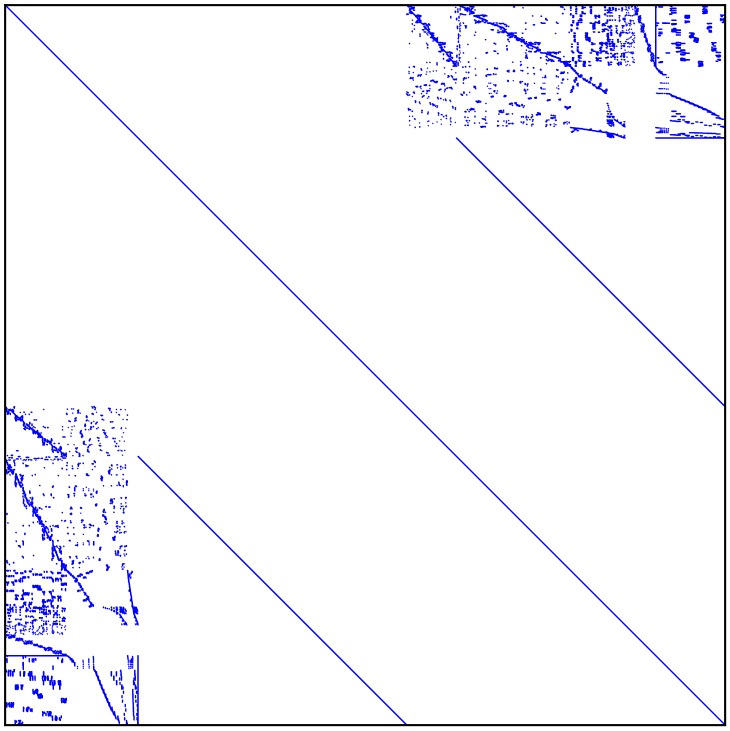


Figure 4.2: c-41 Non-zero Pattern

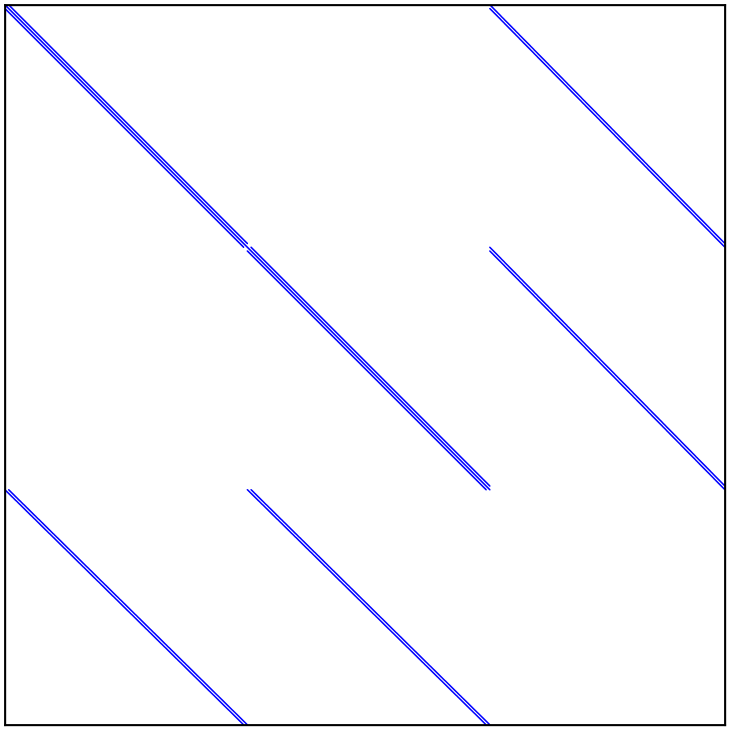


Figure 4.3: stokes64 Non-zero Pattern

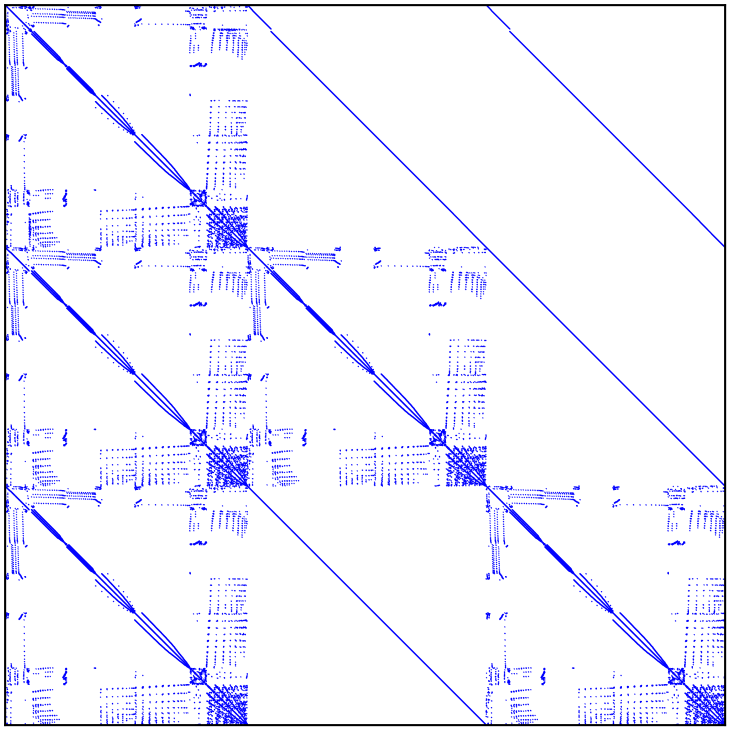


Figure 4.4: igbt3 Non-zero Pattern

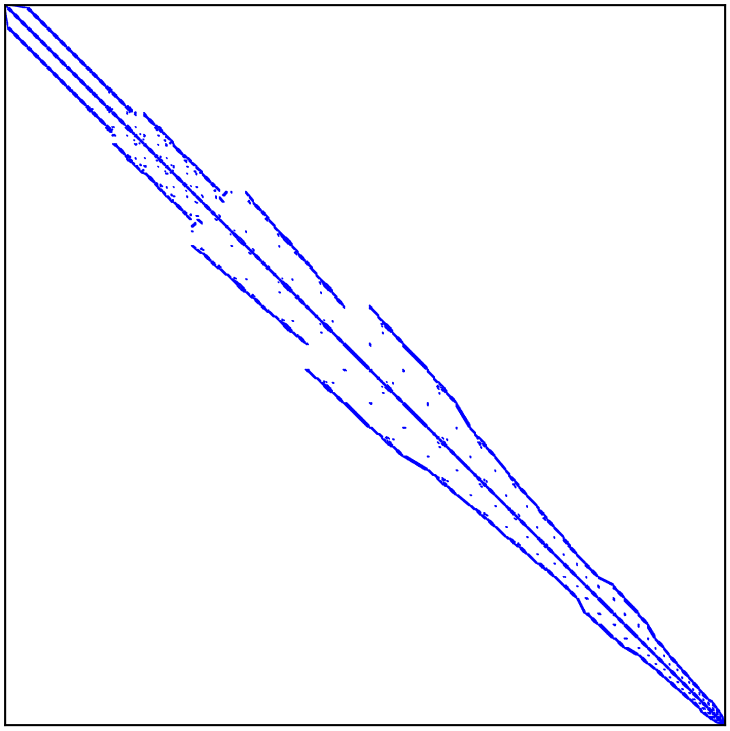


Figure 4.5: nasa4704 Non-zero Pattern

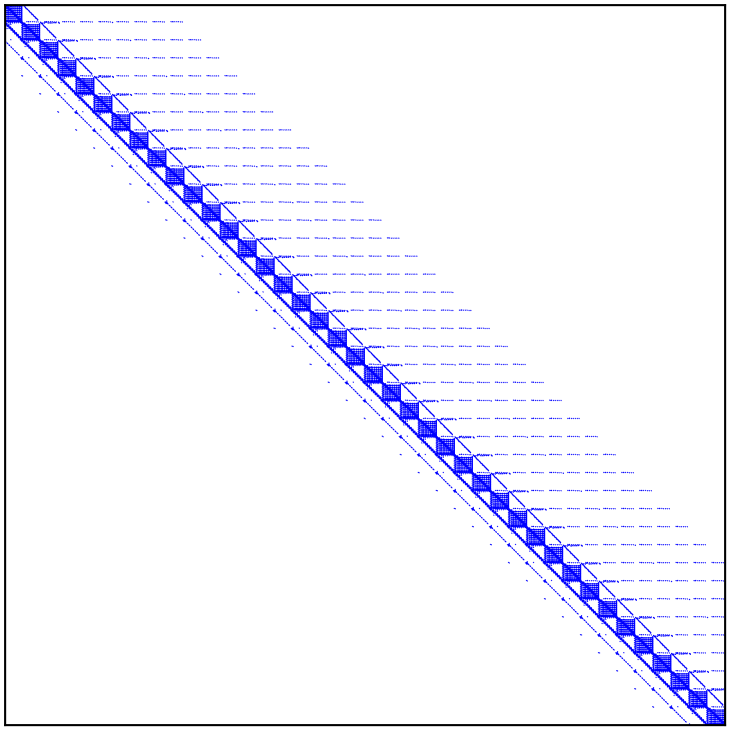


Figure 4.6: mark3jac040 Non-zero Pattern

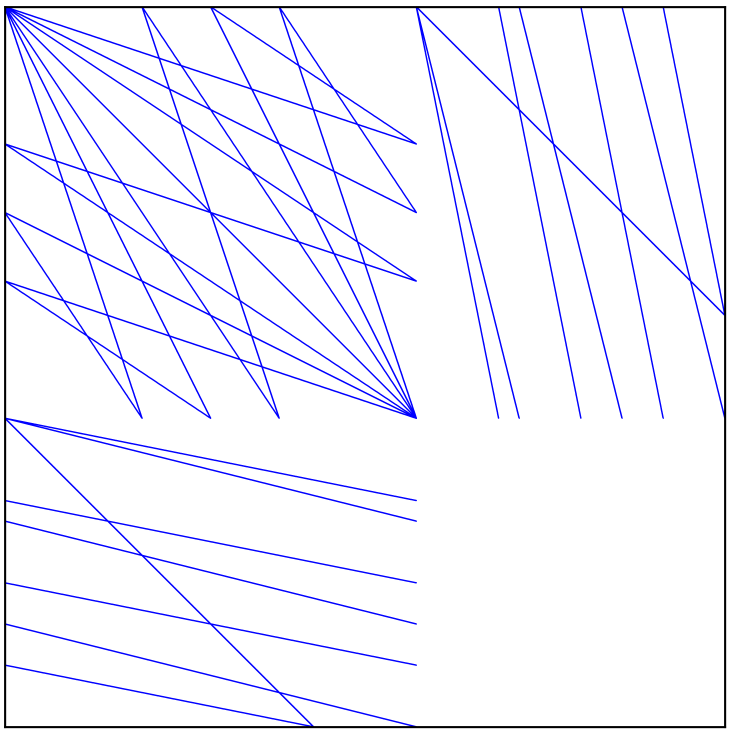


Figure 4.7: cvxqp3 Non-zero Pattern

5. Performance Data and Analysis

This chapter presents performance experiments and results used to understand bottlenecks in sparse LU decomposition on general-purpose processors and compare to the LUHW design. Section 5.1 compares the LUHW performance to software methods. Section 5.2 explores the performance of multifrontal and straightforward methods on power and comparison matrices. Sections 5.3 and 5.4 explore the performance of the merge operation and cache hierarchy during software decomposition.

5.1 Comparison of Methods

In Section 2.4, the LUHW special purpose hardware design was introduced as an alternative to software running on a general-purpose processor for decomposing sparse power matrices. It was designed to perform the key operations required during sparse LU decomposition efficiently as possible. Previous work has shown that this design is capable of close to an order of magnitude performance improvement over general-purpose processors when the design is scaled up in terms of operating frequency and parallel merge units are used. However, the limitations of working with a custom architecture in terms of development cost and limited potential for adoption lead us to investigate which parts of the design are most critical to its success. With an understanding of what makes the LUHW design perform well on power matrices, it may be possible to exploit advanced features of a general-purpose processor or to implement specific hardware features that improve sparse algorithm performance.

To get an overall understanding of performance, the first question to ask is: How does LUHW performance compare to software methods? The working LUHW prototype was implemented on an FGPA with a single merge unit running at 133 MHz, so we will use this setup to compare to software. The design performance could potentially be increased several-fold by using multiple merge units, operating at a higher frequency or implementing the design on an ASIC. Without speculating on additional performance we can use the

Table 5.1: Sparse LU Hardware Performance Model Parameter Values

Parameter	**Description**
CACHE_ROWS	128
DEFAULT_FREQ	133
MASKED_PIVOT	1
FORCE_DIAG	0
PARTIAL_PIVOT	1
PIVOT_THRESHOLD	0.001
PROC_SCHED	0
NUM_PROC	1
CLOCKS_COLMAP	2
CLOCKS_COLMAP_PERWORD	4.78
CLOCKS_SDRAM	28
CLOCKS_SDRAM_PERWORD	2.4
SDRAM_WIDTH	1
HIT_TO_MISS	10
HIT_TO_HIT	0
MISS_TO_HIT	3
MISS_TO_MISS	8
FMUL_TO_FMUL	2
ROW_TO_ROW	10
CLOCKS_CACHE	30
CLOCKS_FMUL	11
CLOCKS_FADD	10
CLOCKS_FDIV	28
CLOCKS_TRANSLATE	2
CLOCKS_PIVOT	2

existing prototype and the performance model that has been tuned to match results from the prototype, to understand the benefits of the design. The parameter values used for the performance model are listed in Table 5.1. Cycle counts, floating-point operations, and cache use statistics are reported by the performance model for each single run of decomposing a matrix. As reported in [6], the performance model successfully projects performance to within 95% of actual hardware results.

The software used in this comparison was benchmarked on an Intel Core i7 965 Extreme Edition processor running at 3.2 GHz. The software used includes UMFPACK 5.4.0, a simple left-looking method from the CSparse package [11], and an implementation of Gaussian Elimination. UMFPACK is a popular sparse solver package that performs the multifrontal

method described in Chapter 3. It relies on machine specific, high-performance Dense BLAS routines to execute the frontal matrix outer-product operations and derives much of its performance from these routines. For these experiments the GotoBLAS2 [19] library was used for underlying dense BLAS routines. The CSparse package is a pedagogical sparse solver library written by the author of UMFPACK. The LU decomposition routine in CSparse uses a straightforward left-looking algorithm which contains a sparse triangular solve step which dominates its running time. The software implementation of Gaussian Elimination was written for the analysis presented in this work. It was designed to match the operation of the LUHW as closely as possible within the limits of running as software on a general purpose processor.

Each of the software routines were run on the benchmark power matrices and measured using high-resolution timers and counters provided by the PAPI performance counter library [5]. Only the numeric part of LU decomposition was timed, excluding any matrix loading and symbolic analysis. Figure 5.1 shows the overall performance of each software routine and the LUHW in terms of millions of floating-point operations per second of run time (MFlop/s).

The CSparse left-looking routine clearly performed the best over the power system matrices, outperforming even UMFPACK. Overall the performance in the 50-300 MFlop/s range is far below the peak performance available on the Core i7 processor. Even not counting SIMD or multi-core parallelism, the peak rate should be 6.4 GFlop/s (with dual-issue multiply and add), making these results less than 10% of peak. Counting all available resources, the i7 has a peak performance around 70 GFlops, making these results 0.5% of peak. The LUHW prototype has the lowest performance, unable to match chip improvements made in the few years since it was designed. Previous benchmarks of UMFPACK performance on the jac26k power matrix resulted in 89 MFlop/s on a 2.6 GHz Pentium 4 and 101 MFlop/s on a 2.4 GHz Core2. This reveals nearly a 2× speedup on the Core i7 system. Based on the performance model projection, an increase in the LUHW speed to approximately 500 MHz would result in performance that matches or beats the CSparse left-looking algorithm on the Core i7.

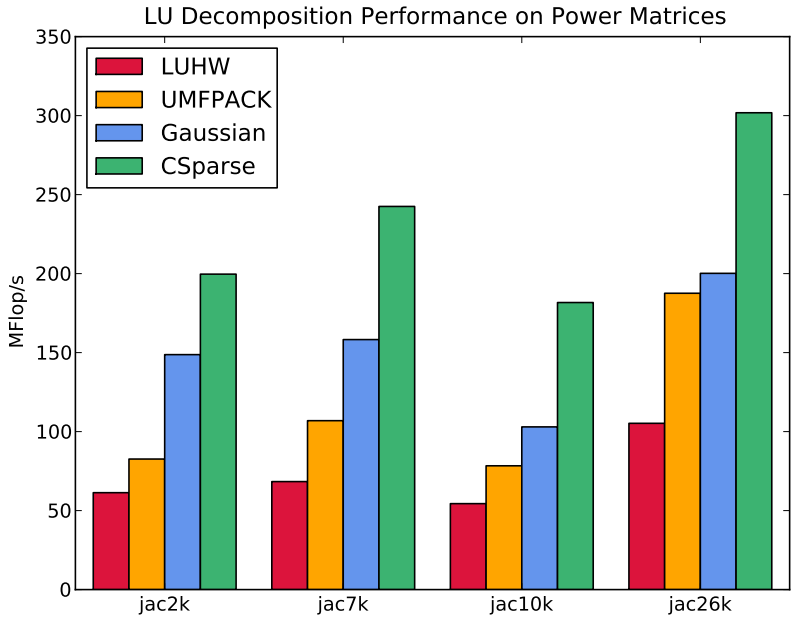


Figure 5.1: LU Decomposition Performance

Figure 5.2 presents an alternative view of performance which takes into account the low speed of the FPGA based LUHW. This chart shows floating-point efficiency by comparing flops per cycle. The LUHW is capable of performing close to one floating-point operation every cycle of its execution. This performance efficiency is what would be desirable to duplicate on a general purpose processor.

5.2 Multifrontal vs. Straightforward

The unexpected results from the previous section prompt the next question: Why do the straightforward Gaussian Elimination and left-looking methods perform better than UMFPACK? UMFPACK is supposed to be one of the fastest sparse solvers available and uses machine specific BLAS routines. To confirm UMFPACK's overall performance it is compared to left-looking LU on a variety of sparse matrices from different application domains. Figure 5.3 shows performance measurements taken on the comparison matrices selected from the University of Florida sparse matrix collection [9]. These results confirm

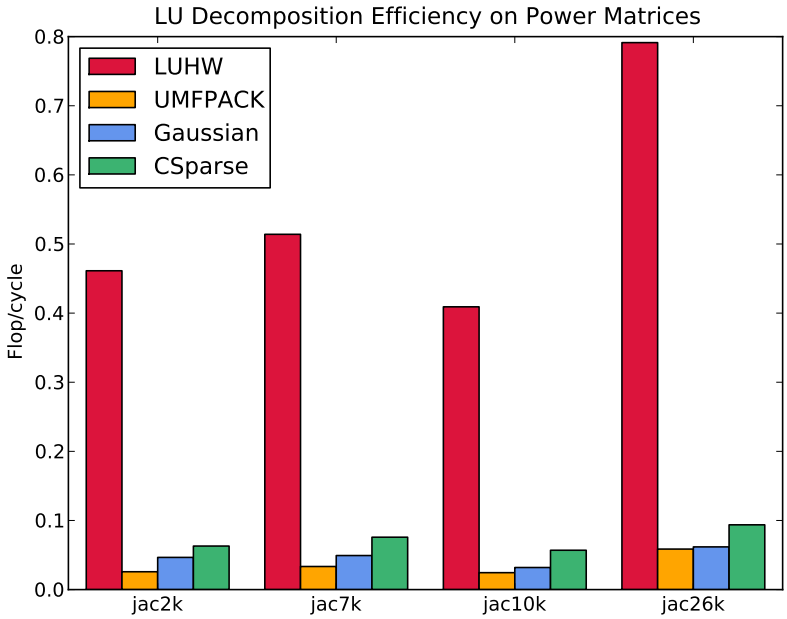


Figure 5.2: LU Decomposition Efficiency

that UMFPACK performance is generally much higher than straightforward methods.

Since UMFPACK relies on Dense BLAS routines for performance, it is useful to investigate their use and performance impact. During decomposition, UMFPACK breaks up computation into a set of frontal matrices, each of which correspond to a BLAS operation. Dense BLAS routines have higher peak performance on larger matrices as setup and calling costs are amortized over the computation. It is a reasonable guess then that larger frontal matrices will result in higher performance.

To test this hypothesis, a large sampling of matrices from the UF sparse matrix collection was factorized using UMFPACK. The time, number of flops and number of frontal matrices was gathered for each matrix. An arbitrary measure of BLAS operation size was conceived called "chunk size" which describes the size of the BLAS operation in terms of flops per frontal matrix. The chunk size was found to be closely related to UMFPACK performance. Figure 5.4 shows the results of these observations. All matrices tested are shown as red points in the chart. The benchmark power matrices and benchmark comparison matrices

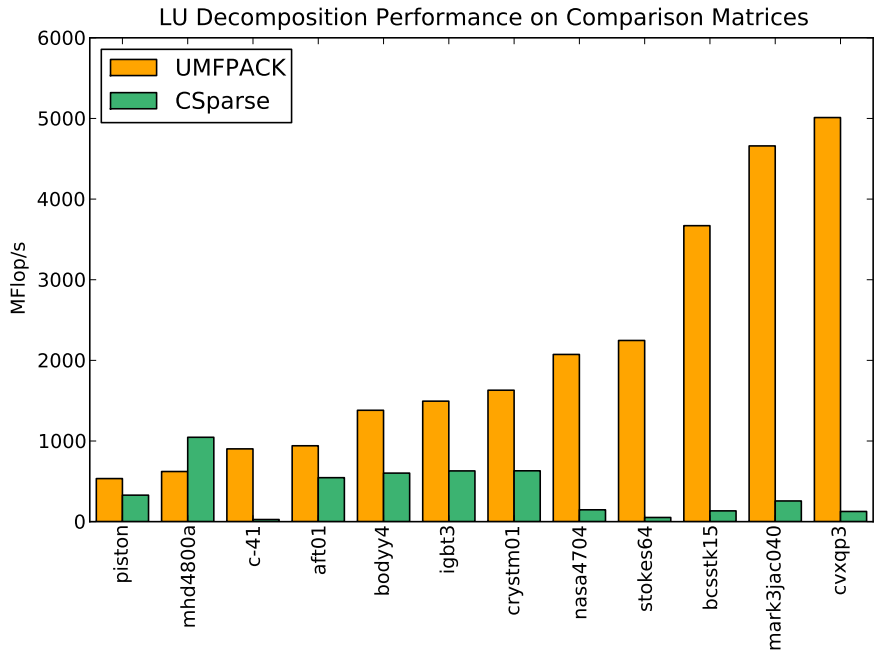


Figure 5.3: LU Decomposition Performance on Comparison Matrices

are also highlighted. It is notable that the power matrices are clustered around smaller chunk sizes with lower resulting performance. The set of comparison matrices mostly has large frontal matrices and better performance. There are a couple of outliers in the comparison set which correspond to diagonal or banded matrices.

These results mirror those found by Davis in his comparison of Cholesky decomposition performance[9]. Their results show that matrices with small chunk sizes do not benefit from supernodal methods due to the extra overhead required. Since power system matrices are have small chunks and UMFPACK is both supernodal and requires additional floating-point operations and data movement overhead for each frontal matrix, it is not effective to use this method on power system matrices. This result also supports the case for special purpose hardware or instructions when decomposition of these matrices cannot take advantage of SIMD vector hardware designed for dense BLAS routines.

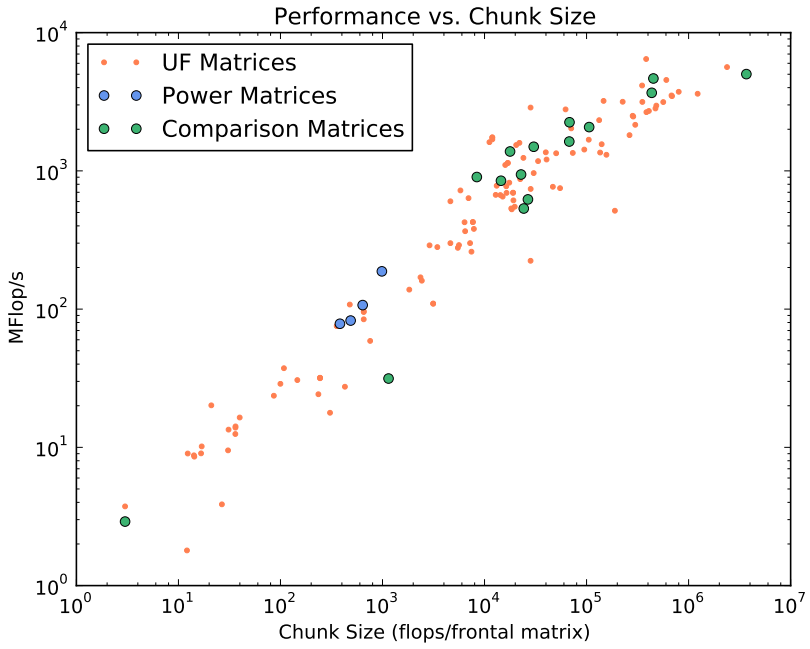


Figure 5.4: Chunk Size to Performance Relationship

5.3 Merge Performance

Despite the significant differences in performance of the methods compared here, the performance of all of them is very low in comparison to the peak theoretical performance of general-purpose processors. In an effort to better understand the causes of this performance gap, it is helpful to experiment on parts of the algorithms separately. The two main performance improving features of the sparse LU hardware that are not reproducible in software are the custom merge unit compute pipeline and the application specific cache tuned to power matrices. The merge unit is responsible for a compressed row-add operation which updates submatrix rows with the scaled pivot row.

In software, the main loop of the merge operation shown in Figure 5.5 has to conditionally increment pointers into the pivot and submatrix rows. The loop body contains a 3-way conditional set of statements for the cases when a pivot row index and submatrix row index are equal (update), the submatrix row index is smaller (copy), or pivot row index is smaller

```
1  for (p = 1, s = 1; p < pivot_row_len && s < subm_row_len; ) {
2      pi = AjPiv[p];
3      si = AjSub[s];
4      if (si == pi) {
5          /* update */
6          wj[rnz] = si;
7          wx[rnz++] = AxSub[s] - lx*AxPiv[p];
8          p++; s++;
9      } else if (si < pi) {
10         /* copy */
11         wj[rnz] = si;
12         wx[rnz++] = AxSub[s];
13         s++;
14     } else {
15         /* fill-in */
16         wj[rnz] = pi;
17         wx[rnz++] = -lx*AxPiv[p];
18         /* colmap fill */
19         fillj[fnz] = pi;
20         filli[fnz++] = subm_row;
21         p++;
22     }
23 }
```

Figure 5.5: Primary Merge Loop

(fill-in). This loop is heavily data-dependent from one iteration to the next, and requires a large number of extra operations for indexing and loop maintenance.

In hardware this merge operation is fully pipelined. A comparison at the beginning of the pipeline determines the operation based on row indices, which is later carried out by the floating-point add unit. The unit is able to execute up to 2 flops per cycle, one multiply and one add.

To determine if the software merge is a significant bottleneck, its operation is benchmarked in the Gaussian Elimination code. A random sampling of merge operations is taken during decomposition of the jac26k matrix. For each sample, the merge operation is run once as normal to fulfill any cache misses that may occur during merge. The merge operation is then rerun 1000 times on the same input pivot and submatrix rows to gather a large enough run time. Row counts and the average merge operation time are output. From this

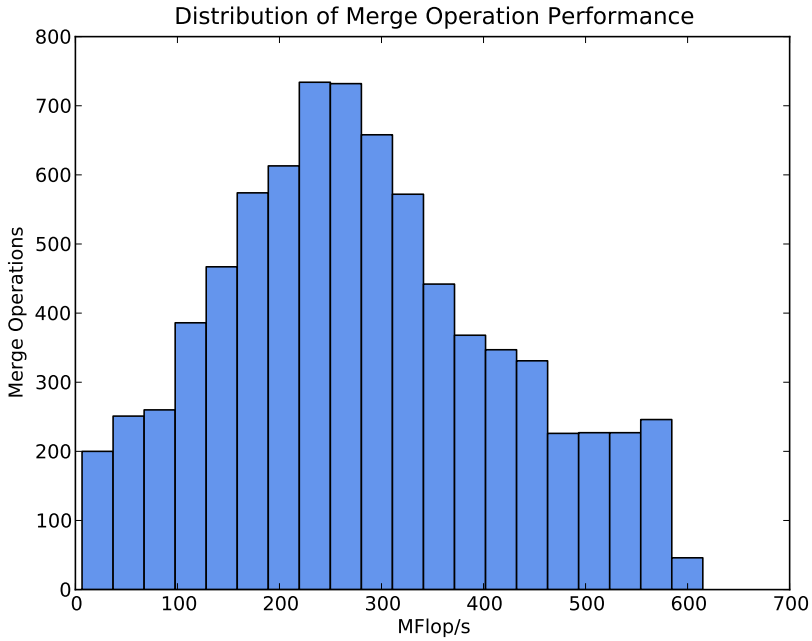


Figure 5.6: Merge Operation Performance Distrubution

information, the performance of each merge operation can be determined. Figure 5.6 shows the distribution of merge operation performance. This experiment shows that the range of possible performance rates for the merge operation is in the 0-600 MFlop/s range.

To understand the performance range of merge operations, additional benchmarking was done with specific row non-zero patterns. Based on this data, the merge operation performance is affected within the range above by three main factors: the number of copy operations, the number of unmatched operations at the end of rows, and branch mis-prediction. Copy operations do not get counted as useful flops, so a large proportion of copy operations can reduce the merge performance to near 0 MFlop/s. Unmatched operations occur when either the pivot or submatrix still contains elements, but the other has run out. This can occur at the end of a row and is handled by simple single-iterator loops without conditional statements that perform the remaining fill-ins or copies. Unmatched fill-ins run about 100 MFlop/s faster than those occurring during the main merge loop. Finally, the branch prediction units appear to favor either long sequences (at least 32) of the same operation, or

alternating every 1 or 2 operations. Alternating operations at a medium stride such as 4 or 8 negatively impacts performance by up to 85 MFlop/s.

These results indicate that the merge operation is a major bottleneck during LU decomposition. Profiling results show that calls to the merge operation take a majority of the time of the LU decomposition. In the CSparse LU routine, profiling indicates that the depth-first search during sparse triangular solve takes the most time. This DFS determines the non-zero pattern of the result column, requiring a large amount of data dependent indexing operations. In both cases, hardware support for indexing would greatly benefit performance.

5.4 Cache Performance

The custom cache on the LUHW is the other major performance enhancing design feature. The cache supports storing entire rows and streaming them to the computation units and maintains intricate logic to select which rows are cached or evicted at any given point in the operation. To determine if this special cache is an important performance enhancement, it is necessary to find out if cache misses in the LU software are a significant bottleneck. Measuring cache effects can be difficult and error prone due to the complex design of modern caches and pipelined processors, but some measurements can be indicative of whether an application is memory bound.

First it is useful to compare the miss rates of the LUHW to software as seen in Figure 5.7. The miss rates shown are averaged word-miss rates across all words read from a row. The miss rates for both software and hardware are reasonably low. The LUHW misses and the software L2 misses would incur similar penalties since they both use off-chip DDR-type memory as a next-level memory. Since the LUHW cache is organized per-row, the miss penalty can be amortized over the entire row and it may be possible to prefetch rows to reduce the miss rate. The general purpose cache makes much more efficient use of its available space since the LUHW must pad rows that it stores in their entirety.

Cycles per instruction is shown to be an indicator of memory bound applications in [14]. Most modern processors are dual issue, able to dispatch two instructions every cycle. For

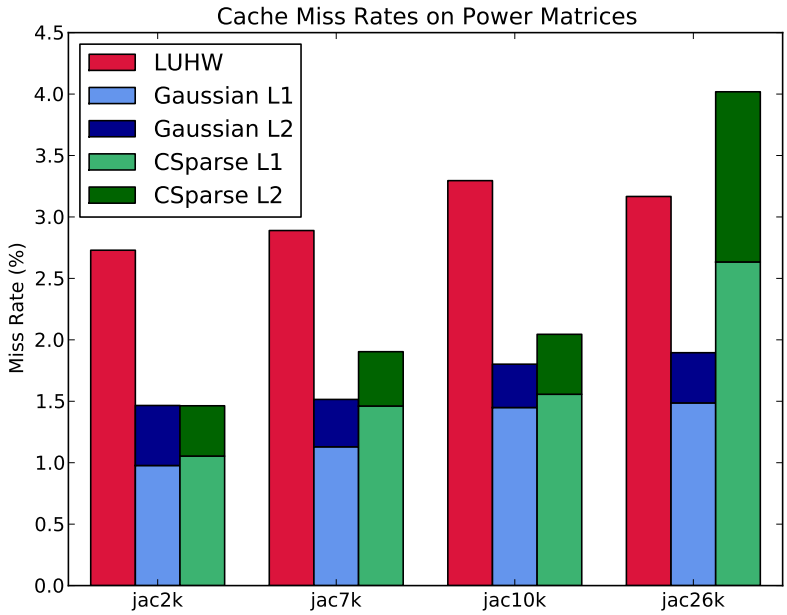


Figure 5.7: Cache Miss Rates

compute bound programs, the CPI rate will approach 1/2. For memory bound programs, if the working set does not fit into L1, cache miss penalties cause the CPI to jump up to around 3. Applications with a large number of L2 misses have a CPI of around 15-20. Figure 5.8 shows the CPI for LU algorithms running on power matrices to be about 0.5 to 0.6, indicating that cache misses are not a significant bottleneck.

To further support this claim, measurements of total L1 and L2 cache misses was used to estimate the maximum possible time used by miss penalties for the LU decomposition software. The Intel Architectures Optimization Reference Manual indicates that the L1 miss penalty is 10 cycles and L2 penalty is about 40 cycles [23]. Figure 5.9 shows the maximum miss penalty cycles based on these penalties and the number of L1 and L2 misses measured relative to the total cycles during LU decomposition. It is important to note that these penalties would only be fully realized if all cache misses completely stalled all computation. Even considering these maximum penalties, eliminating all misses would increase the performance of at most a factor of 2x. This does not account for the full factor

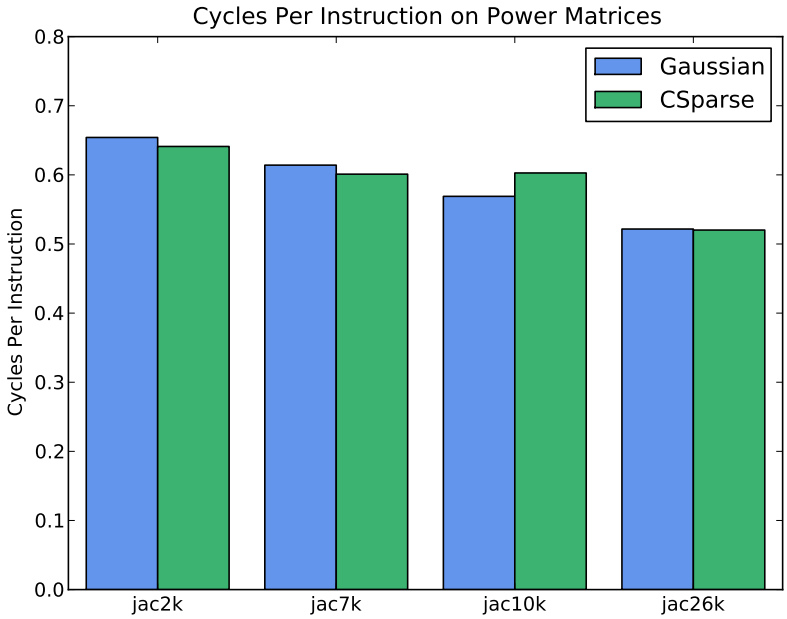


Figure 5.8: Cycles Per Instruction

of 10-200x gap between theoretical peak performance and actual sparse LU performance.

Based on these results, cache misses may have some impact on performance, but not as significant as the indexing operations required by the merge operation. Some memory performance improvements might be made by exploring additional software prefetch instructions to reduce cache misses.

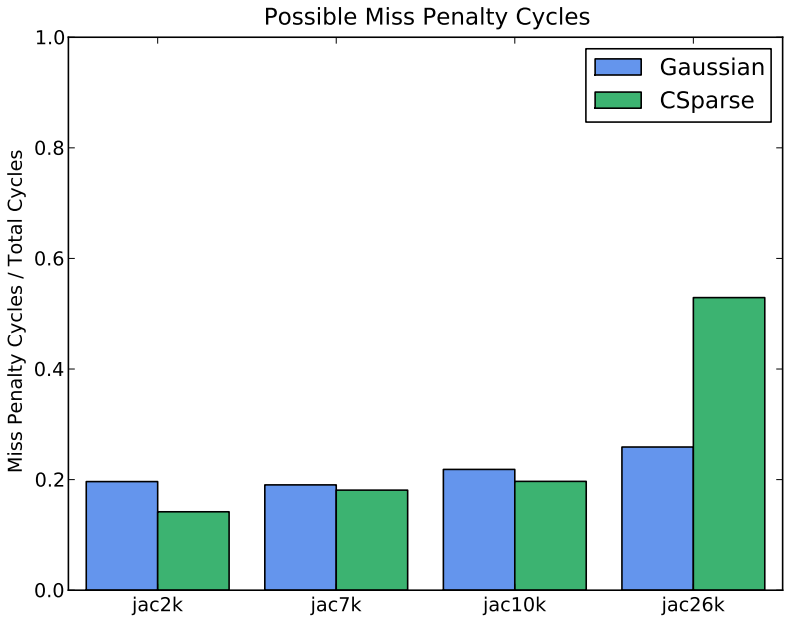


Figure 5.9: Possible Cycles Spent on Miss Penalties

6. Conclusion

Direct methods for sparse LU decomposition are an important set of algorithms used in a variety of applications including power flow computation. There is significant room for improving their performance on general-purpose processors. This work shows that a promising direction for research in this area is the addition of indexing support via a specific merge operation unit or other instruction support. To a lesser extent, some performance improvement may be gained from a customizable cache that is capable of streaming or prefetching rows based on the sparse data access pattern.

Bibliography

[1] An updated set of basic linear algebra subprograms (BLAS). *ACM Transactions on Mathematical Software (TOMS)*, 28(2), 2002.

[2] Patrick R. Amestoy, Enseeiht-Irit, Timothy A. Davis, and Iain S. Duff. Algorithm 837:AMD, an approximate minimum degree ordering algorithm. *ACM Transactions on Mathematical Software (TOMS)*, 30(3), 2004.

[3] Vijay Vittal Arthur R. Bergen. *Power Systems Analysis*. Prentice Hall, 1999.

[4] Krste Asanovic, Rastislav Bodik, James Demmel, Tony Keaveny, Kurt Keutzer, John Kubiatowicz, Nelson Morgan, David Patterson, Koushik Sen, John Wawrzynek, David Wessel, and Katherine Yelick. A view of the parallel computing landscape. *Communications of the ACM*, 52(10):11, 2009.

[5] S. Browne. A Portable Programming Interface for Performance Evaluation on Modern Processors. *International Journal of High Performance Computing Applications*, 14(3):189–204, August 2000.

[6] Timothy Chagnon, Jeremy Johnson, Petya Vachranukunkiet, Prawat Nagvajara, and Chika Nwankpa. Sparse lu decomposition using fpga. In *PARA'08: 9th International Workshop on State-of-the-Art in Scientific and Parallel Computing*, May 2008.

[7] DRC Computer Corporation. *DRC Coprocessor System User's Guide*, July 2007.

[8] Intel Corporation. Intel core i7 processor extreme edition i7-965 processor specification. http://processorfinder.intel.com/details.aspx?sSpec=SLBCJ.

[9] Timothy A. Davis. The university of florida sparse matrix collection. Technical report.

[10] Timothy A. Davis. Algorithm 832:UMFPACK V4.3—an unsymmetric-pattern multifrontal method. *ACM Transactions on Mathematical Software (TOMS)*, 30(2), 2004.

[11] Timothy A. Davis. *Direct Methods for Sparse Linear Systems*. SIAM, 2006.

[12] James W. Demmel, Stanley C. Eisenstat, John R. Gilbert, Xiaoye S. Li, and Joseph W. H. Liu. A supernodal approach to sparse partial pivoting. *SIAM J. Matrix Analysis and Applications*, 20(3):720–755, 1999.

[13] J.W. Demmel, S.C. Eisenstat, J.R. Gilbert, X.S. Li, and J.W.H. Liu. SuperLU: A supernodal approach to sparse partial pivoting. *SIAM Journal on Matrix Analysis and Applications*, 20(3):720755, 1999.

[14] Ulrich Drepper. What every programmer should know about memory. Technical report, 2007.

[15] I. S. Duff, Roger G. Grimes, and John G. Lewis. Sparse matrix test problems. *ACM Transactions on Mathematical Software (TOMS)*, 15(1), 1989.

[16] I. S. Duff and J. K. Reid. The Multifrontal Solution of Indefinite Sparse Symmetric Linear. *ACM Transactions on Mathematical Software (TOMS)*, 9(3), 1983.

[17] Iain S. Duff, Michael A. Heroux, and Roldan Pozo. An overview of the sparse basic linear algebra subprograms:The new standard from the BLAS technical forum. *ACM Transactions on Mathematical Software (TOMS)*, 28(2), 2002.

[18] Sanjay Ghemawat and Paul Menage. Tcmalloc : Thread-caching malloc. http://goog-perftools.sourceforge.net/doc/tcmalloc.html.

[19] Kazushige Goto and Robert A. van de Geijn. Anatomy of high-performance matrix multiplication. *ACM Transactions on Mathematical Software (TOMS)*, 34(3), 2008.

[20] Anshul Gupta and Yorktown Heights. Recent Advances in Direct Methods for Solving Unsymmetric Sparse Systems of Linear Equations. *ACM Transactions on Mathematical Software (TOMS)*, 28(3), 2002.

[21] Anshul Gupta, Mahesh Joshi, and Vipin Kumar. WSMP: A High-Performance Shared- and Distributed-Memory Parallel Sparse Linear Equation Solver. *IBM Research Report*, 2001.

[22] Eun-Jin Im, Katherine Yelick, and Richard Vuduc. Sparsity: Optimization Framework for Sparse Matrix Kernels. *International Journal of High Performance Computing Applications*, 18(1):135–158, February 2004.

[23] Intel Corporation. *Intel 64 and IA-32 Architectures Optimization Reference Manual*, October 2009.

[24] B.M. Irons. A frontal solution scheme for finite element analysis. *Journal of Numerical Methods Engineering*, 2:5–32, 1970.

[25] Jeremey Kepner and John Gilbert. *Graph Algorithms in the Language of Linear Algebra*. SIAM, 2008.

[26] Joseph W. H. Liu. The Multifrontal Method for Sparse Matrix Solution: Theory and Practice. *SIAM Review*, 34(1):82 – 109, 1992.

[27] Sudeep Pasricha, Nikil Dutt, and Mohamed Ben-Romdhane. Extending the transaction level modeling approach for fast communication architecture exploration. *Annual ACM IEEE Design Automation Conference*, 2004.

[28] Markus Püschel, José M. F. Moura, Jeremy Johnson, David Padua, Manuela Veloso, Bryan Singer, Jianxin Xiong, Franz Franchetti, Aca Gacic, Yevgen Voronenko, Kang Chen, Robert W. Johnson, and Nicholas Rizzolo. SPIRAL: Code generation for DSP transforms. *Proceedings of the IEEE, special issue on "Program Generation, Optimization, and Adaptation"*, 93(2):232– 275, 2005.

[29] Feng Tu and A J Flueck. A message-passing distributed-memory parallel power flow algorithm. In *Power Engineering Society Winter Meeting*, pages 211–216. IEEE, 2002.

[30] Petya Vachranukunkiet. *Power flow computation using field programmable gate arrays*. PhD thesis, Drexel University, 2007.

[31] Petya Vachranukunkiet, Jeremy Johnson, Prawat Nagvajara, S Tiwari, and Chika Nwankpa. Performance analysis of load flow computation using fpga. In *15th Power Systems Computational Conference*, August 2005.

[32] R. Venkateswaran and Pinaki Mazumder. A survey of da techniques for pld and fpga based systems. *Integration, the VLSI Journal*, 17:191–240, 1994.

[33] R. Clint Whaley, Antoine Petitet, and Jack Dongarra. Automated empirical optimization of software and the atlas project. *Parallel Computing*, 27(1-2):3–35, 2001.

[34] Wikipedia. File:computermemoryhierarchy.svg. http://en.wikipedia.org/wiki/File:ComputerMemoryHierarchy.svg.

[35] Wikipedia. File:logic_block2.svg. http://en.wikipedia.org/wiki/File:Logic_block2.svg.

[36] Wikipedia. File:switch_box.svg. http://en.wikipedia.org/wiki/File:Switch_box.svg.

[37] Samuel Williams, Leonid Oliker, Richard Vuduc, John Shalf, Katherine Yelick, and James Demmel. Optimization of sparse matrix-vector multiplication on emerging multicore platforms. *Conference on High Performance Networking and Computing*, 2007.

[38] Xilinx, Inc. *Designing Flexible, Fast CAMs with Virtex Family FPGAs*, September 1999. xapp203.

[39] Xilinx, Inc. *Xilinx Virtex-6 Family Overview*, January 2010.