Copyright Warning & Restrictions

The copyright law of the United States (Title 17, United States Code) governs the making of photocopies or other reproductions of copyrighted material.

Under certain conditions specified in the law, libraries and archives are authorized to furnish a photocopy or other reproduction. One of these specified conditions is that the photocopy or reproduction is not to be "used for any purpose other than private study, scholarship, or research." If a, user makes a request for, or later uses, a photocopy or reproduction for purposes in excess of "fair use" that user may be liable for copyright infringement,

This institution reserves the right to refuse to accept a copying order if, in its judgment, fulfillment of the order would involve violation of copyright law.

Please Note: The author retains the copyright while the New Jersey Institute of Technology reserves the right to distribute this thesis or dissertation

Printing note: If you do not wish to print this page, then select "Pages from: first page # to: last page #" on the print dialog screen

NJIT
New Jersey's Science &
Technology University

The Van Houten library has removed some of the personal information and all signatures from the approval page and biographical sketches of theses and dissertations in order to protect the identity of NJIT graduates and faculty.

ABSTRACT

FPGA IMPLEMENTATION OF A CHOLESKY ALGORITHM FOR A SHARED-MEMORY MULTIPROCESSOR ARCHITECTURE

by
Satchidanand G. Haridas

Solving a system of linear equations is a key problem in the field of engineering and science. Matrix factorization is a key component of many methods used to solve such equations. However, the factorization process is very time consuming, so these problems have traditionally been targeted for parallel machines rather than sequential ones. Nevertheless, commercially available supercomputers are expensive and only large institutions have the resources to purchase them or use them. Hence, efforts are on to develop more affordable alternatives. This thesis presents one such approach.

The work presented here is an implementation of a parallel version of the Cholesky matrix factorization algorithm on a single-chip multiprocessor built on an APEX20K series FPGA developed by Altera. This multiprocessor system uses an *asymmetric*, *shared-memory* MIMD architecture, built using a configurable processor core called Nios, which was also developed by Altera. The whole system was developed on Altera's SOPC Development Kit using the Quartus II development environment.

The Cholesky algorithm is based on an algorithm described in George, et al. [**9**]. The key features of this algorithm are that it is *scalable* and uses a "queue of tasks" approach [**9**], which ensures *dynamic load-balancing* among the processing elements. The implementation also assumes *dense* matrices in the input.

Timing, speedup and efficiency results based on experiments run on uniprocessor and multiprocessor implementations are also presented.

FPGA IMPLEMENTATION OF A CHOLESKY ALGORITHM FOR A SHARED-MEMORY MULTIPROCESSOR ARCHITECTURE

by

Satchidanand G. Haridas

A Thesis
Submitted to the Faculty of
New Jersey Institute of Technology
in Partial Fulfillment of the Requirements for the Degree of
Master of Science in Computer Engineering

Department of Electrical and Computer Engineering

May 2003

Blank Page

APPROVAL PAGE

FPGA IMPLEMENTATION OF A CHOLESKY ALGORITHM FOR A SHARED-MEMORY MULTIPROCESSOR ARCHITECTURE

Satchidanand G. Haridas

Dr. Sotirios Ziavras, Thesis Advisor — Date
Professor of Electrical and Computer Engineering, and
Computer and Information Science, Associate Chair for Graduate Studies, NJIT

Dr. Edwin Hou, Committee Member — Date
Associate Professor of Electrical and Computer Engineering, and
Computer and Information Science, Associate Chair for Undergraduate Studies, NJIT

Dr. Alexandros Gerbessiotis, Committee Member — Date
Assistant Professor of Computer and Information Science, NJIT

BIOGRAPHICAL SKETCH

Author: Satchidanand G. Haridas

Degree: Master of Science

Date: May 2003

Undergraduate and Graduate Education:

- Master of Science in Computer Engineering
  New Jersey Institute of Technology, Newark, NJ, 2003

- Bachelor of Science in Computer Engineering
  New Jersey Institute of Technology, Newark, NJ, 2001

- Bachelor of Science in Physics
  University of Mumbai, Mumbai, India, 1999

Major: Computer Engineering

DEDICATION

To God, my family, and my friends

ACKNOWLEDGMENT

First of all, I want to thank Dr. Sotirios Ziavras for his guidance and support during the course of my research for this thesis. Without his advice and encouragement, this thesis would not have seen the light of the day. At the same time, I am also highly indebted to Dr. Edwin Hou and Dr. Alexandros Gerbessiotis for participating in the thesis committee and providing valuable suggestions for improvement.

I also would like to take this opportunity to thank Dr. Jacob Savir, for his support and advice during the course of this project and more importantly during the time of my graduate and undergraduate studies at NJIT.

I would like to express my gratitude for the Chair of the ECE department, Dr. Atam Dhawan and Joan Mahon for their constant encouragement, and also to the entire staff of the Electrical and Computer Engineering department at NJIT for all their patient assistance during this time.

And finally, although not in the least, I want to thank Sunil, Amit, Tirupati, Prabhakar, Abhishek, Xizhen and Xiaofang for their help at various stages of this project. On more than one occasion, their suggestions helped me find a better solution to a problem I faced, than the one I already had.

TABLE OF CONTENTS

Chapter **Page**

1 INTRODUCTION 1

1.1 Objective 1

1.2 Computer Solution of a System of Linear Equations 1

1.3 Cholesky Factorization 4

1.4 Field Programmable Gate Arrays 7

1.5 Configurable Computing 9

1.5.1 Development Environment 10

1.5.2 Intellectual Property (IP) Cores 11

1.5.3 Hardware-Software Co-Design 12

2 HARDWARE IMPLEMENTATION 13

2.1 Architecture of the Shared-Memory Multiprocessor System 13

2.2 Nios Configurable Processor IP Cores 15

2.3 Interconnections using the Avalon™ Bus System 16

2.4 The SOPC Development Board 17

2.5 System Setup Configuration 18

3 SOFTWARE IMPLEMENTATION 20

3.1 The Parallel Cholesky Factorization Algorithm 20

3.2 Analysis of the Parallel Cholesky Algorithm 24

3.3 The Parallel Cholesky Program Design 27

3.4 Application Execution Cycle 29

4 RESULTS AND ANALYSIS 32

TABLE OF CONTENTS
(Continued)

Chapter **Page**

4.1 Timing, Speedup and Efficiency Results 32

4.2 Other Relevant Observations 36

5 CONCLUSION 39

APPENDIX A SYSTEM CONFIGURATION 39

APPENDIX B SOURCE CODE 44

B.1 Parallel Cholesky Program 44

B.2 Matrix Loader Program 48

REFERENCES 50

LIST OF TABLES

Table **Page**

3.1 Global Variables in the Parallel Cholesky Program 28

4.1 Speedup and Efficiency results for $p = 2$ 33

4.2 Comparison of Speedup and Efficiency for $p = 2$ and $p = 3$ 34

4.3 Comparison of Sequential Runtimes for Cholesky Factorization on P_1 and P_2 37

A.1 Summary of the Multiprocessor System 40

LIST OF FIGURES

Figure **Page**

1.1 General form of the Cholesky algorithm 5

1.2 Subtasks in a column-oriented Cholesky algorithm 6

2.1 Architecture of the 3-Nios system 13

2.2 Slave-side arbitration in the Avalon bus system 16

3.1 Top level routine for the parallel Cholesky program 20

3.2 Pseudo-code for the *cmod* routine 21

3.3 Subtasks in the *Tcol* routine 22

3.4 Tasks and subtasks for a matrix of order n = 5 23

3.5 Processor scheduling diagram for matrix of order n = 5 24

4.1 Speedup and efficiency for p = 2 34

4.2 Comparison of speedup and efficiency for p = 2 and p = 3 35

CHAPTER 1

INTRODUCTION

1.1 Objective

Solving a linear system of equations of the form **Ax** = **b** is one of the fundamental problems which one comes across in many engineering and scientific fields. Here, **A** is the coefficient matrix. One approach to solve this problem is to use Cholesky factorization. This method can be adapted quite easily to parallel architectures and has been the focus of a lot of work in the area of parallel matrix factorization. Supercomputers and off-the-shelf multiprocessor systems were usually the target of this line of research [8-10, 13, 14]. Unfortunately, the high cost was the prohibitive factor in making marketable products out of it. However, with recent advances in programmable logic and high-capacity Field-Programmable Gate Arrays (FPGAs) which use advanced VLSI technologies, computer engineers can create parallel systems within a single chip and adapt the above programs for these embedded systems. This thesis presents an implementation of an algorithm described in a paper by George, et al. [9] on Altera's APEX EP20K series FPGA. The algorithm is a parallel implementation of Cholesky factorization and uses shared-memory MIMD architecture.

1.2 Computer Solution of a System of Linear Equations

Consider a system of equations

$$\begin{aligned} a_{11}.x_1 + a_{12}.x_2 + \ldots a_{1n}.x_n &= b_1 \\ a_{21}.x_1 + a_{22}.x_2 + \ldots a_{2n}.x_n &= b_2 \\ a_{31}.x_1 + a_{32}.x_2 + \ldots a_{3n}.x_n &= b_3 \\ &\vdots \\ a_{n1}.x_1 + a_{n2}.x_2 + \ldots a_{nn}.x_n &= b_n \end{aligned} \qquad (1.1)$$

This system can be conveniently solved if methods of linear algebra are used. Assume that the above system is represented in the form

$$\mathbf{Ax} = \mathbf{b} \qquad (1.2)$$

where **A** is an n x n coefficient matrix, **x** is an n-dimensional vector containing the x_i's, that is the unknowns, and b is an n-dimensional vector containing the constants. One comes across problems of this kind in many engineering and scientific fields. There are many approaches to solve this problem, especially using computer-based techniques. These are generally classified by their nature into two main categories – Direct and Indirect. Among these, while the former include algorithms such as Gaussian Elimination, Gauss-Jordan Elimination, and Cholesky Factorization that factorize the matrix **A** into a product of a lower-triangular and an upper-triangular matrix, the indirect approach includes methods like the Gauss-Seidel method and Newton-Raphson approach that try to find the inverse of the matrix **A**, $\mathbf{A}^{-1}$ using iterative techniques. Both of these approaches have been studied for solving a linear system of equations in a parallel environment [5-8, 10, 12, 14]. This work deals only with direct techniques. Among the direct factorization methods mentioned above, each has its advantages and disadvantages and there may be certain cases, depending on the characteristics of the matrices obtained, where a certain method may be more suitable. Accuracy is a factor which usually limits the choices available. The structure of the matrix is another factor. Sparse matrices, wherein the percentage of non-zero elements is very small, are especially suitable for direct methods such as Supernodal LU [6], Multifrontal [18, 19] and Cholesky [9, 12, 14]. For this work, the Cholesky factorization method was chosen over LU decomposition. There were a number of reasons for this as listed on the next page.

- **Faster Execution**

  Only one factor needs to be calculated as the other factor is simply a transpose of the first one. Thus, the number of operation counts for dense matrices, which are the kind of matrices assumed for the present work is $n^3/3$ under Cholesky; this corresponds to a 50% speedup compared to $2*n^3/3$ required for traditional LU decomposition.

- **Highly Adaptive to Parallel Architectures**

  Pivoting is not required in Cholesky factorization. This makes the algorithm especially suitable for parallel architectures as it reduces inter-processor communication. A lot of work [9, 14] has been devoted to parallelizing the Cholesky factorization method.

- **Significantly Lower Memory Requirements**

  In the Cholesky algorithm, only the lower triangular matrix is used for factorization, and the intermediate and final results (the Cholesky factor L) are overwritten in the original matrix. Thus, only the lower triangular matrix can be stored resulting in significant savings in terms of memory requirements.

For the case of sparse matrices, using suitable preconditioning techniques, the execution time for Cholesky factorization can be reduced further than if the original matrix had not been preconditioned. The preconditioning step can result in lesser memory requirements as well. Sparse matrices and their factorization techniques, although they have contributed significantly to the area of parallel factorization, are not the focus of this work and hence will not be discussed further. Readers may refer to [7, 8, 10-14, 17-19] for further information about the issues involved, especially in the preconditioning phase.

In this implementation, only dense matrices are considered, although future work will involve adapting the algorithm for sparse matrices as well. The next section introduces the Cholesky factorization method and the scope for parallelism within it

1.3 Cholesky Factorization

Consider the linear system of equations 1.2 (rewritten here for reference),

$$\mathbf{Ax} = \mathbf{b} \qquad (1.3)$$

where **A** is assumed to a symmetric and positive definite matrix of order *n*, and **x** and **b** are *n*-element vectors. By positive definite, it is meant that **A** satisfies the condition,

$$\mathbf{x}^T\mathbf{Ax} > 0 \qquad \forall\, \mathbf{x} \neq 0 \qquad (1.4)$$

Then to solve equation 1.3, that is to find the vector **x**, the first step is to factorize the matrix **A**. Using the Cholesky algorithm to factorize **A** means that one needs to find the factor **L** such that

$$\mathbf{LL}^T = \mathbf{A} \qquad (1.5)$$

where **L** is a lower triangular matrix. For **A** of the form in (1.4) above, the decomposition (1.5) exists and is unique. A general form of the Cholesky algorithm is given in Figure. 1.1, where the row-order representation is assumed for stored matrices. In the above algorithm, if one changes the order of the three *for* loops, one can get different variations which give different behavior in terms of memory access patterns and the basic linear algebraic operation performed in the innermost loop. Out of the six different variations possible, only three are of interest. They are the row-oriented, column-oriented and sub-matrix forms and are described on the next page.

- **Row-oriented**

  L is calculated row by row, with the terms for each row being calculated using terms on the preceding rows that have already been evaluated.

- **Column-oriented**

  In this variation, the inner loop computes a matrix-vector product. Here, each column is calculated using terms from previously computed columns.

- **Sub-matrix**

  The inner loops apply the current column as a rank-1 update to the partially-reduced sub-matrix.

```
For j = 1 to n
    For k = 1 to j - 1
        For i = j to n
            A[i, j] = A[i, k] - A[i, k].A[j, k]
        End
    End
    A[j, j] = √ A[j, j]
    For k = j + 1 to n
          A[k, j] = A[k, j] /A[j, j]
    End
End
```

Figure 1.1 General form of the Cholesky algorithm.

The column-oriented variation was implemented for this work. This variation can be better understood if the algorithm is re-written in the pseudo-code form shown in Figure.1.2. In this figure, one can identify two distinct subtasks – referred to in literature as cmod and cdiv – that need to be carried out during the course of a regular column-

oriented Cholesky algorithm. A brief description of the behavior of these tasks and the ordering relationship between them is discussed next. The same will be discussed in further detail in Chapter 3. Firstly, for every column except the first one, a number of *cmod* operations must be carried out which modify the target column using terms from the preceding columns. Next, a *cdiv* operation is performed in which, first the diagonal term is replaced by its square-root; next, all the terms below the diagonal element of the target column are divided by the new diagonal term.

```
Cholesky( ) {
  For j = 1 to n
    For k = 1 to j-1
      Modify col. j by A [j, k] * col. k          .... Subtask 1
    End
    Divide col. j by square-root of A [j, j]      .... Subtask 2
  End
}
```

Figure 1.2 Subtasks in the column-oriented Cholesky algorithm.

There exists a clear ordering relation between these two subtasks. Firstly, for all the columns except the first, the *cdiv* subtask can be carried out only after all the *cmod* operations have been carried out on it. For the first column itself, as no preceding columns exist, the *cdiv* task can be carried out directly. Moreover, only after the *cdiv* operation is carried out on a particular column, can that column be used in a *cmod* operation to modify succeeding columns. Depending upon the granularity of the processors, one can assign either individual subtasks or a number of subtasks to a single

processor. Thus, dividing these distinct subtasks among the available processors is one way to parallelize the Cholesky algorithm. Another is to assign an entire column to a single processor, whereby, all the subtasks required for that column are carried out by that processor. This is the path taken in [9], and is also the one that has been implemented here. Hence, all the *cmod* and *cdiv* subtasks associated with a column are carried out in the same processor. A more detailed description of the division of tasks and our implementation in general is provided in the Chapter 3.

The next section presents an introduction to *Field-Programmable Gate Arrays* (FPGAs), which were used as a platform for implementing the multiprocessor system. FPGAs have become very popular in recent times because of the development in areas such as configurable computing, a field where FPGAs have become ubiquitous, and has drawn considerable attention since the invention of these devices and *hardware-software co-design*.

1.4 Field-Programmable Gate Arrays

FPGAs are general purpose programmable devices that can be configured appropriately to implement the desired digital designs. The first FPGA devices were introduced by Xilinx in 1985. Since then a number of major companies have entered this field, including Altera, AT&T and Actel. This project made use of FPGAs and development kits developed by Altera. Although the internal structure of these devices varies depending upon the manufacturer, in general they are composed of arrays of configurable elements known as Logic Elements (LEs) interleaved by routing channels for interconnecting the LEs. A single LE is composed of memory elements known as Look-Up Tables (LUTs) that function similar to the truth-table of a Boolean function, some

storage elements such as flip-flops and some other associated logic such as carry logic for adder circuits. FPGAs also consist of Embedded System Blocks (ESBs) which can be configured by the user to serve as RAM or ROM memory blocks. In general, FPGAs are more complex in terms of their internal structure compared to other programmable devices, such as CPLDs and PLAs, and they offer much higher logic capacities. FPGAs can be programmed by the user using development tools such as Altera's Quartus II, etc. These tools accept the user's design as an input and produces as an output a bit-stream which is used to configure the FPGA. The input design can be in the form of a system description coded in a hardware description language such as VHDL or Verilog, or in a graphical form containing symbols of the logical units comprising the system and the interconnections between them.

When they were first introduced, FPGAs were slow devices. Hence, they could not be used in real life applications. However, their ease of programmability and their smaller design and development cycles made them handy devices for certain tasks, especially as tools to explore novel computer architectures. Because of their shorter development cycle, these devices were also used to test designs for feasibility and fault tolerance before actual transfer into ASICs and fully-custom silicon chips. With the incorporation of VLSI techniques into the making of FPGAs, the clock speed supported by these devices is now high enough to satisfy the demand of many commercial applications. Moreover, their capacity is large enough to often hold an entire processor. This has led to their application to real-time systems, functioning as microcontrollers or digital signal processors. Such systems are now referred to as System-on-a-Chip (SoC) as all the necessary controller logic as well as any supporting interfaces can be configured

into a single chip; SoCs are slowly making their way into the commercial arena. One field that has been tremendously influenced by FPGAs is that of configurable computing, which is briefly discussed next.

1.5 Configurable Computing

The field of configurable computing involves the use of programmable logic devices to implement custom designed hardware logic to perform specific tasks. This field owes its existence to the invention of FPGAs. The key component of a configurable computer is a general-purpose processor which acts as a controller that takes in an input from the user, processes it using user-provided software instructions and then, provides an output that can perform meaningful tasks. For example, consider an application where the device is used as a temperature controller, taking in room-temperature as an input and then, based on the user-defined settings, provides output signals that control a thermostat. Earlier these kinds of signal processing applications were implemented using ASICs. However, the high cost and time of design and development limited its use. Only major players in the field like Motorola and Texas Instruments (TI) could afford the time and money involved with designing and manufacturing these devices. These factors limited their use to certain fields such as networking and telecommunications. Moreover, these devices were not flexible. If a client wanted to make a small change to the design, the whole design and development cycle had to be repeated and a whole batch of chips would have to be fabricated all over again, which made this process prohibitively expensive. With the advent of FPGAs and recent developments in the field of configurable computing though, users can now design applications and implement them at costs that are significantly

lower. Moreover, the designer has the opportunity to control all aspects of the design and development process and can change the design at any stage of the process without incurring any significant costs. Additionally, with recent developments in the field, design and development cycles have become much shorter and quicker. One of these key developments is the appearance of third-party configurable modules known as Intellectual Property (IP) cores that can be plugged right into a user design to save time. IP cores are discussed in further detail ahead. Other developments that have influenced this field include new software-based automation tools that speed up the design and implementation of new ideas, faster and larger (in terms of capacity) FPGAs, and also new design methodologies that simplify the design of new applications. Designers can improve the quality of their products by incorporating some of these techniques into their designs. Some of these are discussed next.

1.5.1 Development Environment

To develop designs for FPGAs with capacities in the range of several thousands of gates, one needs to have at hand a design-entry and development environment which allows the user to handle effectively the complexity of that design. Having the right set of tools can result in significant reductions in development times and also in the size of the final design. In fact, recent design automation tools allow the user total control of the design process, from developing individual blocks right up to synthesizing the design and putting it into the FPGAs. Most of these commercially available tools also provide a graphical interface to aid the designer. These tools allow the user to place the designed blocks into specific parts of the FPGAs in order to reduce latencies, etc. They also provide simulation tools with timing analysis, etc., to test the design at every step of the

development process. Many current development tools also support integrated hardware-software co-design methodologies. Thus, the user can use the same tools to develop the software as well as the hardware systems.

In the current project, Altera's Quartus II software and the associated SOPC Builder tool were used to put together the various components using their easy to use graphical user interface (GUI). This interface gave a high-level view of the entire design and simplified the development process considerably.

1.5.2 Intellectual Property (IP) Cores

As mentioned above, IP cores are ready-to-use, ready-to-synthesize modules that can be plugged into any design to speedup the development cycle. Using these cores, the designer can skip or significantly reduce the otherwise steep learning curve that he or she has to undergo before starting work on a new idea. Moreover, designers can create cores out of their own designs to be used in future work. Recently, organizations are being setup to standardize interfaces for IP cores, with a vision of making their use widespread. With the advent of low cost development tools, new companies are coming up that specialize in developing IP cores for specific markets, such as telecommunications, networking, etc. Developers of networking and communication devices, then license and use these cores in their design to speed up development cycles and reduce costs.

In the work presented here, 32-bit instances of a soft-processor core developed by Altera called Nios were used. In addition, IP cores for UART, and on-chip memory were also used. Using these IP cores significantly reduced development time for this project once the architecture was designed.

1.5.3 Hardware-Software Co-Design

Configurable computing has seen a paradigm shift from traditional design flows, which consisted of clearly defined and demarcated hardware and software flows, towards a flow that integrates the two. Previously, hardware and software were two distinct components of any system and they were treated as such. Because of the complexity of previous design processes, the development team would consist of two separate groups, a hardware group and a software one. Interaction between these groups would be minimal. Hardware, because of the higher development costs, would influence the critical design decisions and normally would be developed first. Only after this, would the software team develop the software to fit the designed hardware. This not only lengthened the design cycle, but also made the system as a whole inefficient. Now with the reduced costs of development and also the ease of implementing the developed designs, the two teams can be integrated for even better performance. Applications are designed for the two components to complement each other. Thus, the decisions made while designing the software influences the hardware design and vice versa. This paradigm has been termed in the literature as *hardware-software co-design*. It has resulted in a framework where the limitations of the hardware process are offset by the software developed and vice versa. Application implementation is now seen not just as a piece of hardware on which to run some software, but instead as an integrated hardware-software system.

CHAPTER 2

HARDWARE IMPLEMENTATION

2.1 Architecture of the Shared-Memory Multiprocessor System

The multiprocessor system used to implement the parallel Cholesky algorithm contains three or more instances of the 32-bit variant of Nios soft-processor core (discussed in the next section) developed by Altera and also various other peripheral IP softcores such as on-chip memory blocks (discussed ahead). This text uses the term "Nios-32" to denote the 32-bit variant of Nios. The multiprocessor system described in this chapter implements asymmetric, shared-memory architecture. Each processor in this system has its own local memory where instructions and data can be stored. There is additional memory that can be accessed globally and is used to store the initial matrix, the data structures, as well as the final Cholesky factor **L**. A block diagram of this architecture is shown in Figure 2.1 and is described next. Note that the following discussion assumes a 3-processor system.

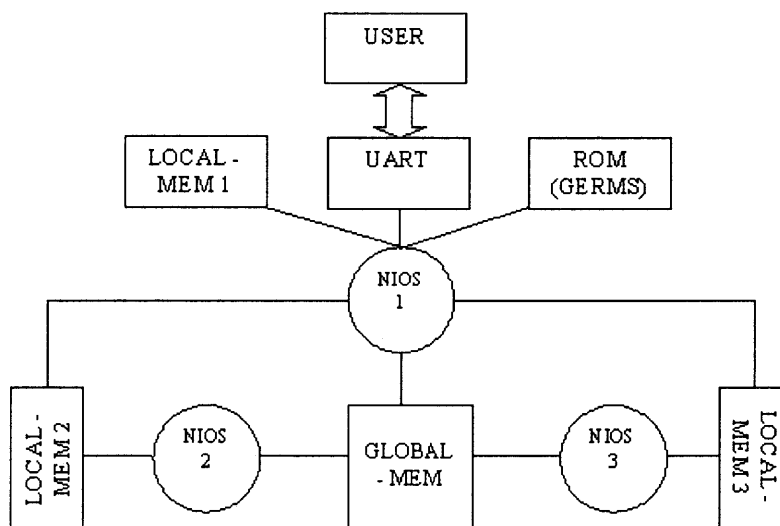


Figure 2.1 Architecture of the 3-Nios system.

As shown in Figure 2.1, although there are no direct connections between the three Nios-32 processors, each can communicate with another through the intermediate memory modules. Thus, if Nios1 wants to communicate with Nios2, it can put the information in either Local-mem2 or Global-mem. Note that in this architecture, while Nios1 has access to the local memories of Nios2 and Nios3, the reverse is not true. Because of this asymmetry, Nios1 can be considered the master processor while Nios2 and Nios3 can be considered the slave processors, although the terms 'master' and 'slave' are not used in the regular sense here. Note that it is the slave processors which participate in the matrix factorization. In this text, the slave processors are also referred to as processing elements. The master-processor, Nios1 also has certain other privileges not available to the other two processors, such as the following:

- It has a UART connection for interaction with the Host computer.
- It can write into the program memory of Nios2 and Nios3, thus letting the user change dynamically, the code being executed by them at run-time.
- In the current configuration, it runs a monitor program called GERMS, provided by Altera, which accepts commands from the Host machine. This allows the user to interact with the system at runtime. The timing program used for measuring the sequential and parallel runtimes is also executed by this processor.

In this system, the global memory is used to store data that all three processors have to access. In the case of the Cholesky program, the task-queue and other global variables were stored here. The connections between the various components in the system were made using the Avalon™ Bus System which is discussed in detail in a later.

2.2 Nios Configurable Processor IP Cores

The system described above was implemented using instances of Altera's Nios™ 2.0 soft-processor core. Altera defines Nios as a "pipelined general-purpose RISC microprocessor" [2]. Nios is configurable so that its features can be selected by the user from a variety of available options depending upon memory requirements and logic requirements, such as floating point support, support for hardware interrupts, etc. Nios also supports custom instructions. These are instructions that can be added by the user to a design to perform specific tasks that may not already be supported by Nios. These instructions, which are implemented as hardware blocks designed using a hardware description language such as VHDL or Verilog, are added to Nios using Altera's SOPC (System-on-a-Programmable Chip) Builder development environment. This option offers significant improvements in terms of timing, efficiency, etc. In the current project, a pipelined single precision floating-point square root unit with an initial latency of twenty-seven, was added as a custom instruction to serve as a substitute for the software coded one. This resulted in a compiled source code with a smaller memory footprint, as well as improved timing results than the one that would have been achieved if a software simulated square root unit was used. For the rest of the floating-point operations, regular software libraries were used. The configuration of a Nios processor can be modified using the SOPC Builder environment. The specific configuration chosen for the processors in the system described here is listed in Appendix A.

2.3 Inter-Connections Using the Avalon™ Bus System

The interconnections in the system were made using the Avalon™ bus system, provided by Altera with the SOPC-Builder™. This system was used to connect processors and peripherals together. It creates point-to-point connections between the master (in this case the Nios-32 processors) and the slave (in our case the memory modules). Thus, as no bus contention occurs, there is no need for external bus arbitration. In the case where a particular slave component - say a memory component - is connected to more than one masters, the Avalon bus system uses slave-side arbitration, which is automatically enabled by the SOPC Builder software whenever the situation arises. This means that an arbitration unit (as shown in the Figure 2.2) is automatically added on the slave side.

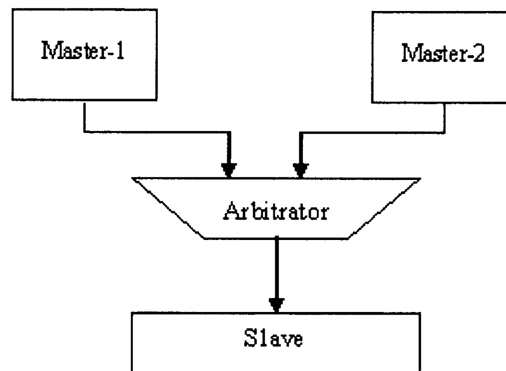


Figure 2.2 Slave-side arbitration in the Avalon bus system.

The Avalon bus system uses a weighted round-robin scheduling policy whenever arbitration is needed. The SOPC Builder uses equal weights as default values which can be changed by the user. The concept of the Avalon bus system significantly simplifies the hardware as well as the software programming on the part of the user. The user does not have to program an arbitration policy explicitly.

At power-on, the three processors start executing from the first location of their program memory. Thus, while Nios1 executes the GERMS monitor program, Nios2 and Nios3 execute their respective pieces of the parallel Cholesky program. The parallel Cholesky program picks up a task from the queue stored in the global memory and modifies the original matrix, which is also stored in the global memory. This matrix decomposition occurs in place, which means that at the end of execution the Cholesky factor **L** is found in the place of the original matrix. The program execution ends when there are no more tasks in the queue.

2.4 The SOPC Development Board

The above 3-processor architecture was implemented on the Altera's SOPC Development Board. The SOPC board has been produced for quick prototyping and development. The main component of this board is the APEX EP20K1500E FPGA which has the capacity to hold 1,500,000 ASIC-equivalent gates. It comes in a 652-pin package, has 51,840 logic elements (LEs) and 442,368 RAM bits [3]. Other important features of this board, in addition to the APEX chip, include:

- Support for six clocks, which includes a BNC connector that can be attached to an external oscillator. The largest on-board clock frequency is 66 MHz. The setup described in this chapter used the on-board 33 MHz clock.
- Numerous off-chip memory devices such as 64-Mbytes of DRAM, 4-Mbytes of Flash memory, as well as 256-Kbytes of EPROM memory. The work described here only made use of on-chip memory resources.
- Various interfaces such as IEEE Std. 1394a (Firewire), RS-232 serial, USB, as

well as 10/100 Ethernet will full- and half-duplex communications. The setup used for this work (listed in the next section) used the RS-232 serial port for transferring user code to the various processors, as well as transferring the data matrix from the host PC to the on-chip global-memory. At the end of the factorization process, the Cholesky factor **L** was transferred from the global memory back to the host PC using the same serial port.

- JTAG interface for configuring the FPGA.

Note that the above information was taken from the Altera's SOPC Development Board User Guide [3] and interested readers can refer to it for further information on this development kit.

2.5 System Setup

The synthesized parallel system which was configured onto the APEX device had the following components:

1. Three Nios-32 processors (standard configuration chosen for each of them).
2. Four on-chip 10 KByte RAM modules (three local and one global).
3. One 4 Kbyte on-chip RAM module containing the GERMS boot monitor.
4. 33.33 MHz system clock.
5. A serial port (115,200 baud and N82).
6. A timer with initial period of 1 msec.

The above components were all individually added and configured using the SOPC Builder tool. The configuration chosen for each of these individual components is

listed in Appendix A. The system was synthesized using Altera's Quartus II, which is an integrated environment for logic design and synthesis. Quartus II 2.0 was used for the work described in this thesis. Upon synthesis, the system occupied 23% of the logic elements, which amounts to 12,003 LEs, and 92% of the embedded system blocks (ESBs), which amounts to 409,600 RAM bits. Only four pins of the APEX device needed to be used. They were one for the 33.33 MHz clocks, one for the global reset and two for the RS-232 interfaces. The SOPC development kit was used for all experiments.

The next chapter presents a detailed description of the parallel Cholesky application, the software that was developed to run on the hardware system described in this chapter.

CHAPTER 3

SOFTWARE IMPLEMENTATION

3.1 The Parallel Cholesky Factorization Algorithm

The central concept of the algorithm described in George, et al. [9] and adapted in the current work is a task queue which contains tasks to be performed by each processing element in the system. The tasks can be divided by rows, columns, or sub-matrices, although the column based division can achieve a higher degree of efficiency [9]. This work focuses on the column variation. In the original implementation, which the authors implemented on a Denelcor HEP multiprocessor, each processor picks up a task *Tcol(j)*, $1 \leq j \leq n$, from a global task-queue, where the tasks are ordered on the basis of increasing column numbers. Thus, *Tcol(1)* appears before task *Tcol(2)* in the task-queue task, which in turn appears before *Tcol(3)*, and so on. Thus, the last task in the queue, and thus the last task to be performed, is *Tcol(n)*, where n is the order of the matrix. The order of tasks in the queue is important and at the end of completion of *Tcol(j)*, column j is the jth column for the Cholesky factor **L** of the original matrix. A high-level structure of the program in terms of the above tasks would be as shown in Figure 3.1.

```
Cholesky( ) {
    For j = 1 to n
    Begin
       Pick up task Tcol(j) from task-queue
    End
}
```

Figure 3.1 Top level routine for the parallel Cholesky program.

Each task *Tcol(j)* is composed of a number of column modification operations. These operations are of two types:

1. A column *j* is modified by using data from all the preceding columns $k = 1$ to $k = j-1$. For a given value of k, this can be denoted by *cmod(j,k)* and its pseudo-code is shown in Figure 3.2 below.
2. The elements of column *j* are divided by the square-root of the diagonal element on the same column. This can be denoted by *cdiv(j)*.

```
cmod( j, k ) {
    For i = j to n
     Begin
        A [i, j] = A [i, j] – A [j, k] * A [i, k]
     End
}
```

Figure 3.2 Pseudo-code for the *cmod* routine.

Again, there is a fixed order for these operations. A *cdiv* operation can only be carried out on column *j* only after the column *j* elements have been modified by data from all the preceding columns using *cmod* operations. Moreover, a *cmod* operation on a particular column can use a preceding column only when the later is ready, that is, after the *cmod* and *cdiv* operations on that column have already been performed. To indicate the status of the particular column, that is to indicate whether it can be used in *cmod* operations on succeeding columns, George, et al. [9] mention the use of an array, *ready[.]*. This data structure has been used in the implementation presented here as well.

```
Tcol( j ) {
    For k = 1 to j-1
    Begin
        Wait until flag ready[k] has been set
        Perform cmod(j,k) operation
    End
    Perform cdiv(j) operation
    Set flag ready[j]
}
```

Figure 3.3 Subtasks in the *Tcol* routine.

A pseudo-code for the task *Tcol(j)*, in terms of the *cmod* and *cdiv* operations, is shown in Figure 3.3. As seen from the above pseudo-code, in each task *Tcol(j)*, a number of *cmod* operations are performed on a column at the end of which a *cdiv* operation is executed. The scheduling is better illustrated in Figure 3.4 on the next page.

Figure 3.4 illustrates how each task is further divided into the above defined *cmod* and *cdiv* subtasks, taking the specific example of a matrix of order $n = 5$. From the ordering shown in the figure, one can observe the scope for parallelism in this algorithm. Consider a scenario for two processors, P_1 and P_2, where each processor handles a single task *Tcol(j)*. Without loss of generality, it can be assumed that processor P_1 picks up and starts working on task *Tcol(1)* which contains only a single subtask *cdiv*(1). During this time, processor P_2, which is currently idle, will pick up *Tcol(2)* from the queue. *Tcol(2)* consists of subtasks *cmod(2,1)* and *cdiv(2)*, which need to be carried out in that order. For subtask *cmod(2,1)* to be performed, column 1 needs to be ready first. Hence, P_2 will be idle while P_1 finishes its work on column 1. Once this is done, P_2 will resume execution.

In the meantime, P_1 can now pickup *Tcol(3)* from the queue. While P_2 is still working on column 2, P_1 can at least complete subtask *cmod(3,1)* as column 1 is complete. Once column 2 is ready, P_1 can perform the rest of the task, *Tcol(3)*. During this time P_2, which is now idle will pick up *Tcol(4)* from the queue and, until *Tcol(3)* is fully completed, will perform sub-tasks *cmod(4,1)* and *cmod(4,2)*. This process will continue until no more tasks remain to be performed, which in this case happens once P_1 picks up the remaining task *Tcol(5)*. The authors of the original paper [9] term this as "self-scheduling" which does away with any explicit load balancing on the part of the programmer.

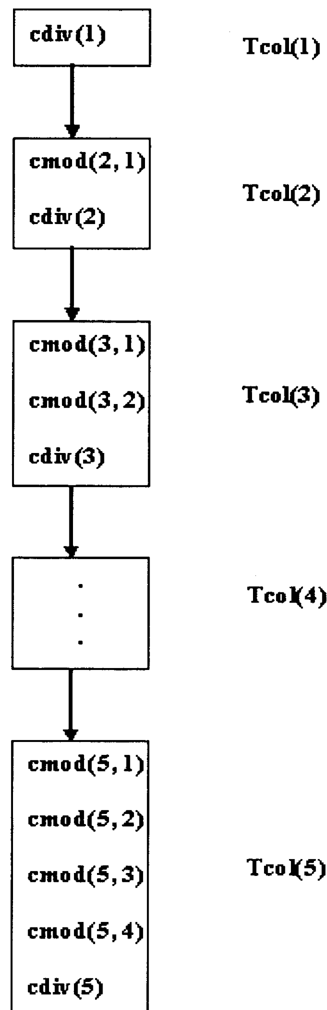


Figure 3.4 Task and subtasks for a matrix of order n = 5.

3.2 Analysis of the Parallel Cholesky Algorithm

One major advantage of this algorithm, as can be seen from the above discussion, is that load balancing is embedded in it. Because each processor picks up a new task as soon as it is done with an old one, no processor sits idle unless there are no more tasks to be carried out. Nevertheless, note that this approach does not imply that each processor is

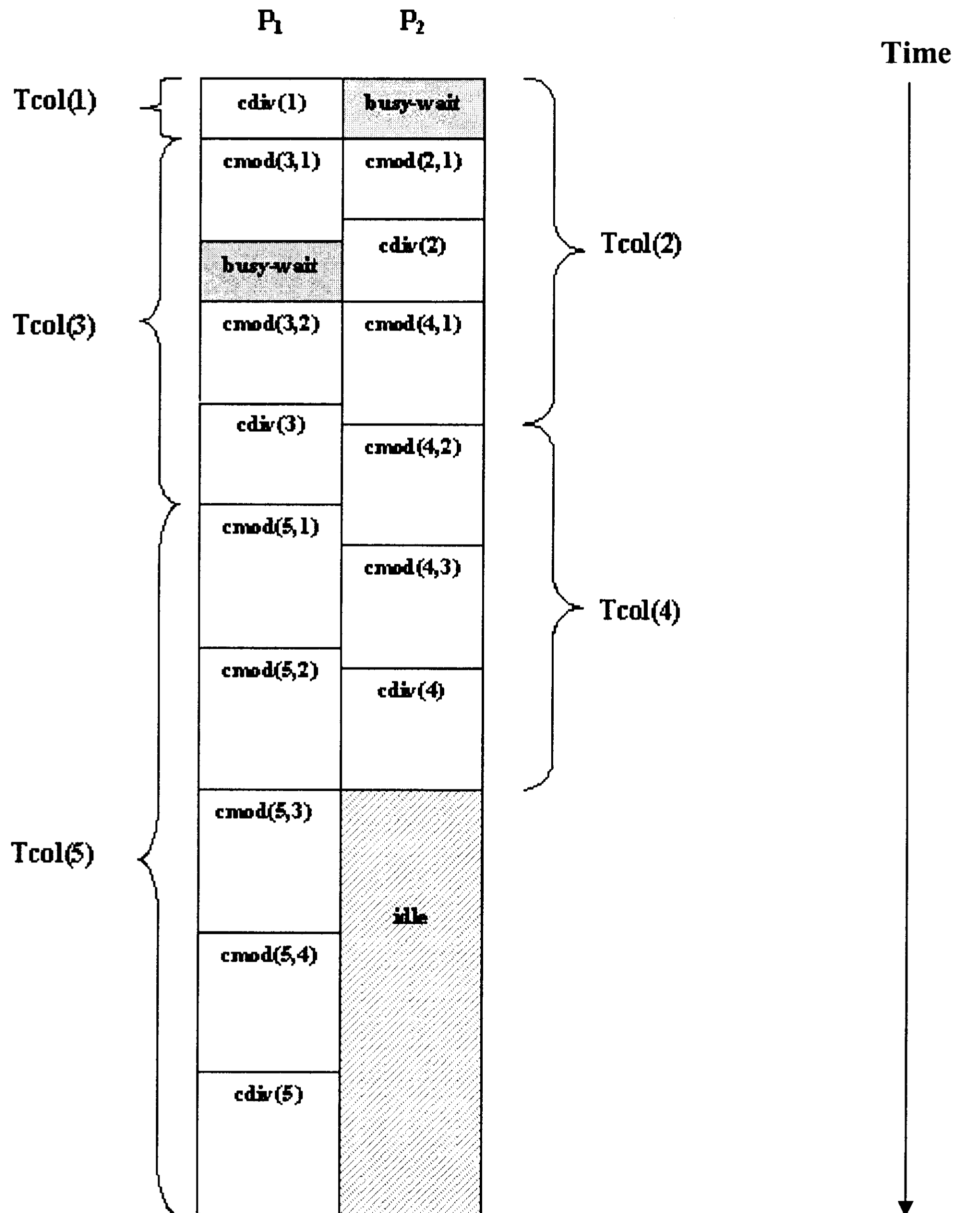


Figure 3.5 Processor scheduling diagram for a matrix of order n = 5.

busy throughout the program's execution. As discussed above, during the execution of a particular task there exists a certain interval during which the processor is waiting for a preceding column to be ready. That is, for the *ready* flag of the preceding column to be set. However, this idle time, which in [9] is called the *busy-wait* time (shown in Figure 3.5), is of a lesser order than the total execution time for the entire program. Another advantage of this algorithm is that as a particular column is modified by a single task that is run on the same processor, there is no need for external synchronization. This algorithm is well suited to MIMD architectures as the granularity level is quite large. One requirement is that the task queue should be visible to all the processors, and, hence, the shared-memory must be global to all the processors and large enough to hold the entire queue.

The complexity of the algorithm is calculated next, for which the following three assumptions are made:

1. Matrices are dense, and
2. Multiplication, division and finding the square-root are equivalent floating-point operations. That is, it assumed that they require the same amount of time.
3. Moreover, the time taken for addition/subtraction operations is neglected as it is small compared to the time required for the operations listed above.

Then, to calculate each column j of the Cholesky factor L, one needs:

$$\Gamma\{\,\mathrm{Tcol}(j)\,\} \quad = \quad j\,(n-j+1) \text{ floating-point operations} \qquad \textbf{(3.1)}$$

Thus the total number of operations to completely factorize a matrix of order n is

$$= \sum_{i=1}^{n} \Gamma\{Tcol(i)\}$$

$$= \Gamma\{\text{Tcol}(1)\} + \Gamma\{\text{Tcol}(2)\} + \Gamma\{\text{Tcol}(3)\} + \ldots + \Gamma\{\text{Tcol}(\text{n-1})\} + \Gamma\{\text{Tcol}(\text{n})\}$$

$$= 1.n + 2.(n-1) + 3.(n-2) + \ldots + (n-1).(n-(n-2)) + n.(n-(n-1))$$

$$= \{1.n + 2.n + 3.n + \ldots + (n-1).n + n.n\} - \{2.1 + 3.2 + 4.3 + \ldots + (n-1).(n-2)) + n.(n-1)\}$$

$$= n\sum_{i=1}^{n} i - \sum_{i=1}^{n-1} i.(i+1)$$

$$= n\sum_{i=1}^{n} i - \left\{\sum_{i=1}^{n-1} i^2 + \sum_{i=1}^{n-1} i\right\}$$

$$= n.\frac{n.(n+1)}{2} - \left\{\frac{n.(n-1).(2.n-1)}{6} + \frac{n.(n-1)}{2}\right\}$$

$$= \frac{n}{2}\left\{n^2 + n - \left[\frac{2.n^2 - 3.n + 1}{3} + n - 1\right]\right\}$$

$$= \frac{n}{2}\left\{\frac{n^2 + 3.n}{3}\right\}$$

$$= \frac{n^3}{6} + \frac{n^2}{2}$$

$$= O(n^3) \qquad \textbf{(3.2)}$$

This complexity is the same for the parallel implementation of the Cholesky algorithm as well. Again, note that the derivation above does not take into account the time required for addition and subtraction operations, which normally require one clock cycle to complete. In terms of exact number of floating-point operations, the Cholesky factorization requires $n^3/2$ operations, which are half as many as that required in the traditional LU decomposition algorithm.

3.3 The Parallel Cholesky Program Design

As the parallel system used for this application was built using components that are normally used only for uniprocessor designs, support for parallelism, such as a parallelizing compiler, was not available. Hence, to parallelize the application, explicit memory references were used in the code for each processing element to access structures in global memory. As each of the processors came with a C compiler, C-styled pointers were used to make the various data structures globally accessible. Thus, during the time of designing the memory architecture, a base address and the size for the global memory were decided upon and fixed using Altera's SOPC Builder development environment. The rest of the system was built around this memory block. Then, the addresses of the various globally accessible flags and arrays were fixed within this global block again using pointers. This is shown in Figure 3.6.

```
/*****************************************************************************/
/*              GLOBAL VARIABLES MEMORY MAP
/*****************************************************************************/
        semaphore = base+1;

        task_id = base+2;

        order = base+3;

        n = *order;

        ready = base+4;

        processor = ready+n;

        A = (float *)processor+n;
/*****************************************************************************/
```

Figure 3.6 Declaration of global variables in the source code.

Table 3.1 Global Variables in the Parallel Cholesky Program

Variable Name	Data Type	Description
base	integer pointer	Base address. All other addresses are defined in terms of this address.
flag	integer	A flag used to signal the event that the data matrix has been stored into the global memory so that the processors can begin the factorization.
semaphore	integer	Used as a flag for protecting the critical section in the program (corresponds to picking up a unique task from the queue).
task_id	integer	This variable stores the number of columns still left to be worked upon. The current value of this variable also acts as the task-id for a processor. The processor then performs the *cmod* and *cdiv* operations on the column with the current value of this variable. Once the processor has obtained this value, it increments the value of this variable by one. The factorization is complete when the value of this variable is equal to that of the order of the matrix.

Table 3.1 Global Variables in the Parallel Cholesky Program (Continued)

Variable Name	Data Type	Description
order	integer	Stores the order of the matrix to be compared with the value in the variable *task_id*.
ready	integer array	The length of this array is equal to the order of the matrix. When the values in a particular column are ready to be used by other columns, the value in that particular element of the array is set to a non-zero value. As long as the value in a particular element of this array is zero, that column cannot be used to modify other columns.
processor	integer array	The length of this array is equal to the order of the matrix. This array stores the ID of the processor that worked on the columns. Thus, if the second element contains the value '2', column 2 was modified by processor 2.
A	float array	Contains the data matrix and the Cholesky factor matrix.

3.4 Application Execution Cycle

When the reset switch is pressed on the development board, all the processors that are going to take part in the actual matrix factorization start executing code from the starting address in their respective program memory. This starting address can be specified by the

user while configuring the system using SOPC Builder. Each processor then waits until the input matrix is loaded into memory. For this, each processor keeps polling the *flag* variable (whose value is initialized to zero) to check if it is set. This variable is set only after the input matrix is loaded into the global matrix. This matrix is loaded using a program that is executed by the master processor, which is also connected to the UART. Once the *flag* variable is set, all the slave processors resume their respective execution. The processors stop execution when there are no more tasks left in the task queue. For the setup used in this work, this is the case when the value of the variable *task_id* is equal to the value of the variable *order*. Given a matrix, in order to factorize it, the following steps need to be performed to get the final result, which is the Cholesky factor **L**.

1. Assuming the programs for the actual processing elements are already in place, the system is reset using the *reset* switch on the development board. This causes the processors that are involved in the factorization to start executing code in their respective program memories. This causes each of those processing elements to wait for the input matrix to be uploaded into the global memory. This, as described above, is indicated by the *flag* variable.
2. The input matrix is uploaded into the global memory using the *master* processor. The matrix is embedded in a program uploaded into the program memory of the master processor using the serial port. The processor starts executing the program immediately upon uploading. This program also initializes the other variables and arrays for the actual factorization. Once the input matrix is in place, the program will also set the *flag* variable to a non-zero value. For the setup described in this thesis, which consisting of two processing elements P_1 and P_2, a value other than

2 or 3 causes the system to run in parallel and this means that both the processors that factorize the matrix will run in parallel. If the value of *flag* is 2, only processor P_2 will factorize the matrix. If *flag* is 3, only processor P_1 will take part in the factorization. This approach enables one to calculate the serial and the parallel runtimes with relative ease, and, hence, get the speedup and efficiency values for the various test matrices.

3. The program which uploads the input matrix in the global memory will continue to run during the factorization process. It is also used to get timing results. When the final column of the Cholesky factor is in place, the program will print the timing results on the screen and exit. The program outputs the timing results in terms of number of clock cycles required for the factorization. This multiplied by the clock frequency represents the actual runtime of the factorization. If need be, this program can also be used to download the Cholesky factor onto the host PC.

The source code for the parallel Cholesky implementation as well as the matrix loading programs are provided in the Appendix B. The next chapter provides experimental results and analysis for the developed system.

CHAPTER 4

EXPERIMENTAL RESULTS AND ANALYSIS

4.1 Timing, Speedup and Efficiency Results

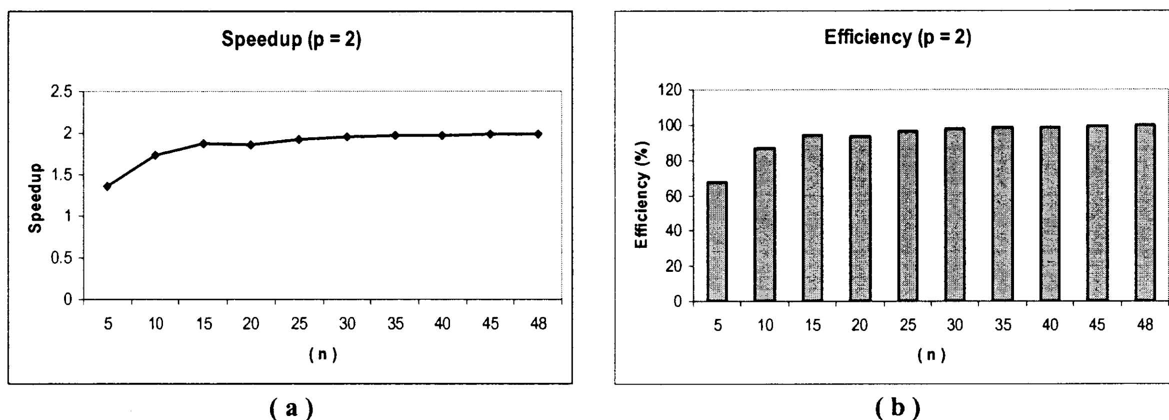
The results of the experiments carried out on the two-processor system using matrices of various orders are summarized in Table 4.1 on the next page. For n = 48, the BCSSTK01 test matrix (a sparse matrix containing 224 non-zero terms), which was obtained from the Harwell-Boeing matrix set available at the Matrix Market [20] was used. The rest of the matrices were generated in MATLAB using the *gallery* function, passing 'minij' as a parameter, which generates SPD matrices with terms A[i, j] = min(i, j). Thus, they were dense matrices with a simple structure. Their Cholesky factor L has all ones for the lower triangular matrix, and, hence, their results could be verified easily. Note that the parallel Cholesky program does not offer specific support for sparse systems and all the matrices were treated as dense, although only the lower triangular part was stored in each case.

As one can see from Table 4.1, the speedups as well as the efficiency in general increase as the order of the matrix increases. Note that, except for the case p = 1, where p is the number of processors that take part in the factorization, the speedup and efficiency will always be less than the ideal values (p for the speedup and 100% for the efficiency), because of factors such as inter-processor communication, idling of certain processors while waiting for other calculations to complete or because of contention for resources such as bus, memory, etc. In the parallel Cholesky program, the processors are idle during the busy-wait period (discussed in Chapter 3), and in the end when processors are idle because there are no more tasks in the queue. During the busy-wait time the

Table 4.1 Speedup and Efficiency for p = 2

Order of Matrix (n)	Sequential Runtime (ms)	p = 2		
		Parallel Runtime (ms)	Speedup	Efficiency (%)
5	8.56	6.33	1.352	67.6
10	64.56	37.21	1.735	86.75
15	211.64	112.49	1.881	94.05
20	493.46	264.54	1.865	93.25
25	953.72	495.45	1.925	96.25
30	$1.636 * 10^3$	839.60	1.948	97.4
35	$2.584 * 10^3$	$1.308 * 10^3$	1.976	98.8
40	$3.842 * 10^3$	$1.945 * 10^3$	1.975	98.75
45	$5.453 * 10^3$	$2.758 * 10^3$	1.977	98.85
48	$6.343 * 10^3$	$3.175 * 10^3$	1.99	99.9

processors are waiting for the calculations on the preceding columns to be completed so that the terms on these columns can be used for further calculations. This time influences the speedup and efficiency obtained. However, because this time is of a lesser order than the runtime of the factorization process, as the order of the input matrix increases the busy-wait time becomes negligible compared to the total sequential runtime of the Cholesky factorization. Thus, the speedup and efficiency values approach their ideal values. This is seen in Table 4.1 and graphically represented in Figure 4.1 for matrix of order n = 48, where the efficiency is as high as 99.9%.



(a) (b)

Figure 4.1 (a) Speedup and **(b)** efficiency for p = 2.

Table 4.2 Comparison of Speedup and Efficiency for p = 2 and p = 3

Order of Matrix (n)	p = 2		p = 3	
	Speedup	**Efficiency**	**Speedup**	**Efficiency**
5	1.352	67.6	1.67	55.7
10	1.735	86.75	2.325	77.49
15	1.881	94.05	2.435	81.18
20	1.865	93.25	2.664	88.79
25	1.925	96.25	2.640	88.0
30	1.948	97.4	2.845	94.84
35	1.976	98.8	2.857	95.23
40	1.975	98.75	2.945	98.17
45	1.977	98.85	2.937	97.91
48	1.99	99.9	2.754	91.80

Speedup and efficiency results for a 4-Nios system were also obtained, out of which three processors performed the factorization (p = 3), while the fourth was the master processor. These results are provided in Table 4.2 above. Results obtained for the p = 3 were then compared with those obtained for p = 2. These results are presented graphically in Figure 4.2 below.

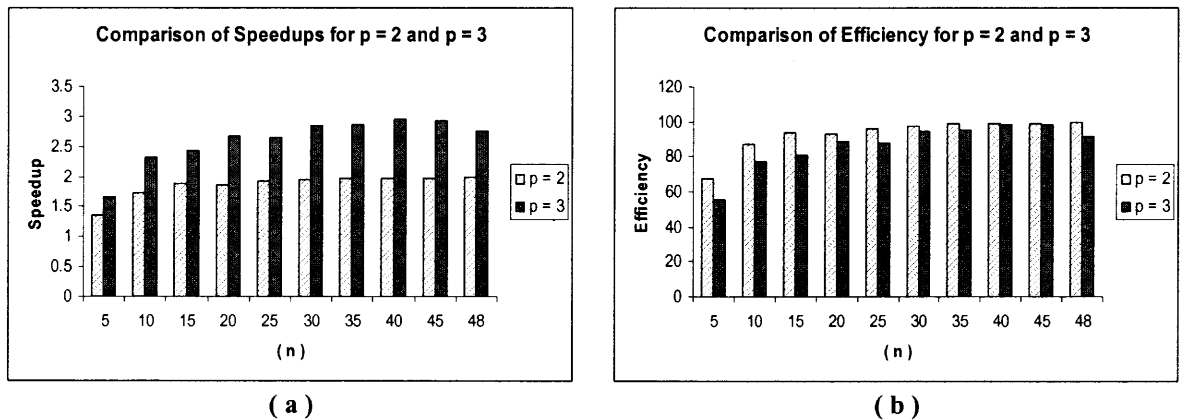


Figure 4.2 Comparison of **(a)** speedup and **(b)** efficiency for p = 2 and p = 3.

One can observe from the graphs in Figure 4.2 that while the speedups for p = 3 are much higher than that for p = 2, the latter performs much better in terms of efficiency. This could happen because for the case p = 3, the contention for memory resources will be higher. Hence, the amount of time, a processor may have to wait to get some information is more than that for p = 2. Moreover, the total amount of time the processors are idle because of *busy-wait* for the case p = 3 is also more than that for p = 2. All these factors influence the efficiency of the parallel system. But, in general, one observes significant speedups for the factorization process as the number of participating processors (p) increases.

One important observation that was made during the course of the experiments was that between two different runs of the parallel Cholesky application on the same matrix, the parallel runtimes would change. The reason for this was that sometimes one processor would start to pick-up tasks much later than the others. In a normal run of the application, if there are p processors in the system, each should pick up only one out of the first p columns for processing. For example, one case is where the first processor picks the first column; the second processor picks the second column and so on. However, in some runs of the experiment, it was found that the first few consecutive columns were all picked up and processed by the same processor. This in effect meant that, during this time, the parallel system was not making use of the other processors in the system, which as a result remained idle. The number of these columns changed between runs and in each case once all the processors did participate in the factorization, the application functioned normally and all the remaining columns were picked up one by one by different processors. This effect was partly responsible for the lower-than-expected speedup values obtained for some test matrices. The reason for this delayed start of some of the processors is not known. This effect will be investigated in future experiments.

4.2 Other Relevant Observations

For the sake of curiosity, in the 3-Nios case (p = 2) a comparison was also made between the sequential times on each of the two processors P_1 and P_2 that were involved in the factorization process. For this the same Cholesky program was executed on both the processors, each time only one of them factorizing the entire matrix. These results are provided below in Table 4.2 on the next page.

Table 4.3 Comparison of Sequential Runtimes for Cholesky Factorization on P_1 and P_2

Order of Matrix (n)	**Runtime for P_1** (ms)	**Runtime for P_2** (ms)	**Speedup** (P_1/P_2)
5	8.86	8.52	1.04
10	67.16	64.41	1.04
15	220.19	211.64	1.04
20	513.44	493.46	1.04
25	992.40	953.72	1.04

As one can see from Table 4.2, the runtimes for the sequential version of Cholesky factorization on P_1 are slower than those on P_2. P_2 is consistently faster than P_1 by about 4%. This is the case despite the fact that a common 33.33 MHz clock was used for both the processors. A number of reasons could account for the above difference:

1. The clock distribution lines are not of uniform dimensions for the two processors. In the current project, placement and routing (P&R) was automatically carried out by the Quartus II software itself. Hence, there may be a difference in the length of the clock distribution lines from the source to the above processors. This can result in clock skew, which may account for the observed delay.
2. It is also possible that the global memory is closer to P_2 than P_1. Thus, the physical distances are not the same. This could again result in the observed delays. [22] discuses the effects of locality in single-chip multiprocessors and report performance improvements obtained by controlling it. In future work, advanced

features provided by the development software will be used to tightly control these issues.

Please note that no experiments were performed to determine the actual reason for the delay. But these experiments will be part of future work.

CHAPTER 5

CONCLUSIONS

Summarizing this thesis, it can now be said that with the FPGA devices that are now available in the market, single-chip multiprocessors are very much a realizable alternative to develop for parallel applications. Although the timing results that have been obtained are not exceptional, they are due to using a slow FPGA device that was introduced more than a year ago. Moreover, the system used a clock with a frequency of just 33.33 MHz. Under these constraints, it is felt that the performance of the system did meet acceptable standards. Still there is plenty of scope for improvements. One of the planned improvements includes using more recent FPGAs. Some of these FPGAs support speeds in excess of 400 MHz. This approach should significantly improve the execution time of the parallel Cholesky application.

It is also hoped that the work presented in this thesis will offer ideas for further research in this direction. Already, the field of reconfigurable computing is making great strides. This field takes advantage of modifying the configuration of FPGAs at runtime without first having to power-off the device to develop applications with dynamically changing logic blocks and hardware architectures to make better use of the available FPGA resources. In view of these developments, single-chip multiprocessors hold that much more potential as they can utilize the advantages offered by reconfigurable computing to offset any inherent drawbacks.

APPENDIX A

SYSTEM CONFIGURATION

The parallel system used for the parallel Cholesky application was built using the SOPC Builder tool provided by Altera and came along with the Quartus II development environment. SOPC Builder provides a graphical user interface using which, one can build a system from scratch by adding one or more instances of the Nios processor cores and connecting them to other peripheral IP cores such as memory modules, UART, etc. All of the IP cores that are added to the system are configured using the same tool. Table A.1 (below) provides a summary of various components in our system and the interconnections using a grid format. SOPC Builder uses a similar grid interface.

Table A.1 Summary of the Multiprocessor System

	Nios1 (Controller)	**Nios2 (PE)**	**Nios3 (PE)**
local_ram1	1	-	-
local_ram2	1	1	-
local_ram3	1	-	1
global_ram3	1	2	2
boot_ram	1	-	-
UART	1	-	-
Timer	1	-	-

For peripherals connected to more than one processor, the number in the corresponding cell indicates the values used for the weighted-round-robin scheduling algorithm used for the slave-side arbitration by the Avalon bus system. Note that while Nios1 was connected to all the peripherals, the same was not true of the other two processors. The details of the configuration chosen for each of them are listed next. Note that the details we provide are those chosen from the options provided by the SOPC Builder tool.

I. PROCESSORS

1. Nios1

- **Address/Data Bus Width**: 32
- **Register file size**: 128
- **Multiplier**: MUL (3 cycles for 16 X 16 → 32 bit multiplication)
- **Writable WVALID**: No
- **Pipeline Optimization**: More Stalls/Fewer LEs
- **Decoder Logic**: LEs
- **Support RLC/RRC**: No
- **Support Interrupts and Traps**: Yes
- **Catch Spurious Interrupts**: Yes
- **Call C++ Constructors**: Yes
- **Use the CWP Manager**: Yes
- **Use Fast Multiply**: Yes
- **Hardware Breakpoint Support**: No
- **Custom Instructions**: None

2. Nios2

- **Address/Data Bus Width**: 32
- **Register file size**: 128

- **Multiplier**: MUL (3 cycles for 16 X 16 $\rightarrow$ 32 bit multiplication)
- **Writable WVALID**: No
- **Pipeline Optimization**: More Stalls/Fewer LEs
- **Decoder Logic**: LEs
- **Support RLC/RRC**: No
- **Support Interrupts and Traps**: Yes
- **Catch Spurious Interrupts**: Yes
- **Call C++ Constructors**: Yes
- **Use the CWP Manager**: Yes
- **Use Fast Multiply**: Yes
- **Hardware Breakpoint Support**: No
- **Custom Instructions**: Double-Precision Pipelined FP Square-root Unit

3. Nios3
   - **Address/Data Bus Width**: 32
   - **Register file size**: 128
   - **Multiplier**: MUL (3 cycles for 16 X 16 $\rightarrow$ 32 bit multiplication)
   - **Writable WVALID**: No
   - **Pipeline Optimization**: More Stalls/Fewer LEs
   - **Decoder Logic**: LEs
   - **Support RLC/RRC**: No
   - **Support Interrupts and Traps**: Yes
   - **Catch Spurious Interrupts**: Yes
   - **Call C++ Constructors**: Yes
   - **Use the CWP Manager**: Yes
   - **Use Fast Multiply**: Yes
   - **Hardware Breakpoint Support**: No

- **Custom Instructions**: Double-Precision Pipelined FP Square-root Unit

II. PERIPHERALS

1. local_ram1

- **Size:** 10 Kbytes
- **Contents:** Monitor program

2. local_ram2

- **Size:** 10 Kbytes
- **Contents:** Parallel Cholesky program

3. local_ram3

- **Size:** 10 Kbytes
- **Contents:** Parallel Cholesky program

4. global_ram

- **Size:** 10 Kbytes
- **Contents:** Input matrix and global variables

5. boot_ram

- **Size:** 4 Kbytes
- **Contents:** GERMS monitor program

6. UART

- **Baud Rate (bps):** 115200
- **Clock Frequency:** 33.33 MHz
- **Parity:** None
- **Data Bits:** 8

- **Stop Bits:** 2
- **Include CTS/RTS:** No
- **Include end-of-packet registers:** No

7. clk_timer

- **Initial Period:** 1 msec
- **Preset Configuration:** Fully featured (v1.0 compatible)
- **Writeable Period:** Yes
- **Readable Snapshot:** Yes
- **Start/Stop Control Bits:** Yes
- **Output Signals:** None

APPENDIX B

SOURCE CODE

This appendix lists the source code for the parallel Cholesky program as well as the matrix loading program.

B.1 Parallel Cholesky Program

```
/* nios.h contains the memory map for the entire system */
#include "nios.h"


/* Cholesky function */
void cholesky( );


/* Supporting functions */
void Tcol(int);
void cdiv(int);
void cmod(int, int);
int D2_to_D1(int, int);


// GLOBAL DATA 10000 – 103FF
/*******************************************************************************/
int n;
volatile int *base = (int *) 0x10000;
volatile int *flag  = (int *) 0x10000;
volatile int *semaphore;
volatile int *task_id;
volatile int *order;
volatile int *ready;
volatile int *processor;
volatile float *A;
/*******************************************************************************/
```

```
main( ) {
        int i, j = 1;

        /* wait until the input matrix is loaded into global memory
        while (*flag == 0) {
                ;
        }

/*****************************************************************************/
/*              GLOBAL VARIABLES MEMORY MAP                                  */
/*****************************************************************************/

        semaphore       =       base + 1;
        task_id         =       base + 2;
        order           =       base + 3;
        n               =       *order;
        ready           =       base + 4;
        processor       =       ready + n;
        A               =       (float *) (processor + n);
/*****************************************************************************/

        cholesky( );
        while(1) {
                ;
        }
}

/*****************************************************************************/
/*              DEFINITION OF CHOLESKY FUCTION                               */
/*****************************************************************************/
void cholesky( ) {
        int i, j, k;
        while (*task_id != n) {
                while (*semaphore != 0) {
                        ;
                }
                // critical section
                {
                        *semaphore = 1;
```

```
                    j = *task_id;
                    *task_id = *task_id + 1;
                    processor[j] = processor[j] + 1;
                    *semaphore = 0;
                }
                Tcol(j);
        }
}

/****************************************************************************/
/*              DEFINITION OF SUPPORTING FUNCTIONS                          */
/****************************************************************************/
void Tcol(int j) {
        int i, k;
        for (k = 0; k <= j-1; k = k+1) {
                while (ready[k] != 7) {
                        ; //do nothing
                }
                cmod(j,k);
        }
        cdiv(j);
        ready[j] = 7;

        while(ready[j] != 7) {
                ;
        }
}

void cmod(int k, int i) {
        int j;
        for (j = k; j < n; j = j+1) {
                A[D2_to_D1(j, k)] = A[D2_to_D1(j, k)] - A[D2_to_D1(j, i)]*A[D2_to_D1(k, i)];
        }
}

void cdiv(int i) {
        int j;
        A[D2_to_D1(i, i)] = nm_sqrt1(A[D2_to_D1(i, i)]);
```

```
        for (j = i+1; j < n; j = j+1) {
                A[D2_to_D1(j, i)] = A[D2_to_D1(j, i)]/A[D2_to_D1(i, i)];
        }
}

int D2_to_D1(int row_no, int col_no) {
        int tmp = 0;
        tmp = row_no*n + col_no;
        return tmp;
}
```

B.2 Matrix Loader Program

```
/* nios.h contains the memory map for the entire system */
#include "nios.h"

// GLOBAL DATA 10000 – 103FF
/*******************************************************************************/
int n = 3;
volatile int *base = (int *) 0x10000;
volatile int *flag   = (int *) 0x10000;
volatile int *semaphore;
volatile int *task_id;
volatile int *order;
volatile int *ready;
volatile int *processor;
volatile float *A;
/*******************************************************************************/

main( ) {

        int i = 0, j = 1;

/*******************************************************************************/
/*              GLOBAL VARIABLES MEMORY MAP                     */
/*******************************************************************************/
        flag            =       base;
        semaphore       =       base + 1;
```

```
        task_id         =       base + 2;
        order           =       base + 3;
        ready           =       base + 4;
        processor       =       ready + n;
        A               =       (float *) (processor + n);
/*****************************************************************************/


/*****************************************************************************/
/*              INITIALISATION                                               */
/*****************************************************************************/
        *task_id = 0;
        *semaphore = 0;
        *order = n;

        for (i = 0; i < n; i = i+1) {
                ready[i] = 0;
                processor[i] = 0;
        }
        i = 0;

/*****************************************************************************/
/*              LOADING THE INPUT MATRIX                                     */
/*****************************************************************************/
        A[i]    = 4.0;
        A[i+1]  = 1.0;  A[i+2]  = 0.5;
        A[i+3]  = 2.0;  A[i+4] = 0.0;   A[i+5] = 3.0;
        A[i+6]  = 0.5;  A[i+7] = 0.0;   A[i+8] = 0.0;   A[i+9] = 5.0/8.0;
        A[i+10] = 2.0;  A[i+11] = 0.0;  A[i+12] = 0.0;  A[i+13] = 0.0;  A[i+14] = 16.0;

/*****************************************************************************/

        /* Set flag to indicate that the matrix has been loaded */
        *flag = 1;
}
```

REFERENCES

[1] Altera Corp., *Simultaneous Multi-mastering with Avalon Bus*, Altera Application Note AN-184-1.0, Altera Corporation, San Jose, CA, 2002.

[2] Altera Corp., *Nios 2.0 CPU Data Sheet*, Altera Data Sheet DS-NIOSCPU-1.0, Altera Corporation, San Jose, CA, Jan. 2002.

[3] Altera Corp., *System-on-a-Programmable Chip Development Board User Guide*, Altera Document A-UG-SOPC-1.3, Altera Corporation, San Jose, CA, Oct. 2001.

[4] S. Brown and J. Rose, *Architecture of FPGAs and CPLDs: A Tutorial*, IEEE Design and Test of Computers 13.2: 42-57, 1996.

[5] K. Compton and S. Hauck, *Configurable Computing: A System of Systems and Software*, Northwest Univ., ECE Dept., Technical Report, 1999.

[6] J.W. Demmel, et al., *A Supernodal Approach to Sparse Partial Pivoting*, SIAM Journ. Matrix Anal. Appl. 20.3: 720-755, 1999.

[7] I.S. Duff, *Sparse Numerical Linear Algebra: Direct Methods and Preconditioning*, Technical Report TR/PA/96/22, CERGACS, Toulouse Cedex, France, 1996.

[8] K.A. Gallivan, et al., *Parallel Algorithms for Matrix Computations*, SIAM, Philadelphia, PA, 1990.

[9] A. George, M.T. Heath and J. Liu, *Parallel Cholesky Factorization on Shared-Memory Multiprocessor*, Linear Algebra and its Applications 77:165-187, 1986.

[10] A. George and J W.H. Liu, *Computer Solution of Large Sparse Positive Definite Systems*, Prentice-Hall, Englewood Cliffs, N.J., 1981.

[11] A. George and J W.H. Liu, *The Evolution of the Minimum Degree Ordering Algorithm*, SIAM Review 31.1:1-19, Mar. 1989.

[12] G.H. Golub and C.V. Loan, *Matrix Computation (Second Edition)*, The John Hopkins University Press, Baltimore, MD, 1989.

[13] A. Gupta, G. Karypis and V. Kumar, *A Highly Scalable Parallel Algorithm for Sparse Matrix Factorization*, IEEE Trans. Parallel and Distributed Systems, 8.5:502-520, May 1997.

[14] M.T. Heath, E. Ng and B.W. Peyton, *Parallel Algorithms for Sparse Linear Systems*, SIAM Review 33.3: 420-460, Sep. 1991.

[15] V. Kumar, et al., *Introduction to Parallel Computing: Design and Analysis of Algorithms*, Benjamin/Cummings, Redwood City, CA, 1994.

[16] P.L. Levin and R. Ludwig, *Crossroads for Mixed-Signal Chips*, IEEE Spectrum 39.3: 38-43, Mar. 2002.

[17] J W.H. Liu, *The Role of Elimination Trees in Sparse Factorization*, SIAM J. Matrix Anal. Appl. 11.1:134-172, Jan. 1990.

[18] J W.H. Liu, *The Multifrontal Method for Sparse Matrix Solution: Theory and Practice,* SIAM Review 34.1:82-109, 1992.

[19] J W.H. Liu, *The Multifrontal Method and Paging in Sparse Cholesky Factorization,* ACM Trans. Math. Software 15.4:310-325, 1989.

[20] Matrix market, *BCSSTK01 Test Matrix*, http://math.nist.gov/MatrixMarket/data/ Harwell-Boeing/bcsstruc1/bcsstk01.html.

[21] S. Pritchard, *Integrating IP into Wireless Systems*, RF Design, Premedia Publ., Mar. 2003.

[22] K.A. Shaw and W.J. Dally, *Migration in Single Chip Multiprocessors*, Computer Architecture Letters 1.3:2-5, Nov. 2002.