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

Satchidanand G. Haridas and Sotirios G. Ziavras
Department of Electrical and Computer Engineering
New Jersey Institute of Technology
Newark, New Jersey 07102, USA
ziavras@njit.edu

Address for Correspondence

Sotirios G. Ziavras
Department of Electrical and Computer Engineering
New Jersey Institute of Technology
Newark, New Jersey 07102, USA
ziavras@njit.edu

* This research was supported in part by the U.S. Department of Energy under grants ER63384 and DE-FG02-03CH11171.

ABSTRACT

Solving a system of linear equations is a key problem in 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 often 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. Hence, efforts are on to develop more affordable alternatives. In this paper, we propose such an approach. We present an implementation of a parallel version of the Cholesky matrix factorization algorithm on a single-chip multiprocessor built inside an APEX20K series FPGA (Field-Programmable Gate Array) developed by Altera. Our multiprocessor system uses an *asymmetric*, *shared-memory* MIMD architecture and was built using the configurable Nios™ processor core which was also developed by Altera. Our system was developed using Altera's SOPC (System-On-a-Programmable-Chip) Quartus II development environment. Our Cholesky implementation is based on an algorithm described in George, et al. [**6**]. This algorithm is *scalable* and uses a "queue of tasks" approach to ensure *dynamic load-balancing* among the processing elements. Our implementation assumes *dense matrices* in the input. We present performance results for uniprocessor and multiprocessor implementations. Our results show that the implementation of multiprocessors inside FPGAs can benefit matrix operations, such as matrix factorization. Further benefits result from good dynamic load-balancing techniques.

Keywords: parallel Cholesky factorization, FPGA, performance evaluation, matrix inversion.

1. INTRODUCTION

Solving a linear system of equations is a fundamental problem which one comes across in many engineering and scientific fields. Assume a linear system of equations represented in the form

$$\mathbf{Ax = b} \qquad (1)$$

One approach to solve this problem is to use Cholesky factorization [**6**]. This method can be adapted quite easily to parallel architectures and has been a focus of attention in the area of parallel matrix factorization. Supercomputers and off-the-shelf multiprocessor systems have usually been the target of this line of research [**5-8**]. However, the high cost of such systems is

often a prohibitive factor in making marketable products. With recent advances in programmable logic and high-capacity FPGAs that employ advanced VLSI technologies, computer engineers can now create parallel systems within a single chip and adapt programs for these embedded systems. We present here our implementation of a Cholesky factorization algorithm [**6**] on Altera's APEX EP20K series FPGA. Our parallel implementation employs a shared-memory MIMD architecture.

There are many approaches to solve a linear system of equations, 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, the indirect approaches include methods like Gauss-Seidel and Newton-Raphson. Many of these approaches have been studied for factorizing a matrix in a parallel environment [**4-8, 17**]. Each of the above categories 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, especially for iterative solutions. The structure of the matrix is another factor. A sparse matrix, wherein the percentage of non-zero elements is very small, is especially suitable for direct methods such as Supernodal LU [**4**], Multifrontal [**11, 12**], Cholesky [**6, 8**], etc. In our implementation, we chose the Cholesky factorization method over LU decomposition [**15, 16**]. There were a number of reasons for this.

- **Faster execution**

  Only one factor needs to be calculated as the other factor is simply the transpose of the first one. Thus, the number of operations for dense matrices is approximately $n^3/3$ under Cholesky; this corresponds to a 50% speedup compared to about $2*n^3/3$ operations required for traditional LU decomposition.
- **Highly adaptive to parallel architectures**

  Pivoting is not required. This makes the Cholesky algorithm especially suitable for parallel architectures as it reduces inter-processor communications. A lot of work [**6, 8**] 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 must be stored, resulting in significant savings in terms of memory requirements.

In the case of sparse matrices, using suitable preconditioning techniques the execution time for Cholesky factorization can be reduced further. The preconditioning step can result in lesser memory requirements as well. Sparse matrices and their factorization techniques [**5, 7, 8, 10-12**], although they have contributed significantly to the area of parallel factorization, are not the focus of our work and hence will not be discussed further. We assume dense matrices in this paper. In the next subsection, we discuss the generic form of Cholesky factorization and then briefly discuss the scope for parallelization, leaving the details for a later section.

1.1 Cholesky Factorization

Let us assume that **A** in equation (**1**) is a symmetric and positive definite matrix of order *n*, and **x** and **b** are *n*-element vectors. By positive definite, we mean that

$$\mathbf{x}^T\mathbf{A}\mathbf{x} > \mathbf{0} \qquad \forall\ \mathbf{x} \neq \mathbf{0} \qquad (\mathbf{2})$$

Then, to find the vector **x** using Cholesky factorization, we need to find the factor **L** such that

$$\mathbf{L}\mathbf{L}^T = \mathbf{A} \qquad (\mathbf{3})$$

where L is a lower triangular matrix. For A of the form in (**2**) above, decomposition (**3**) exists and is unique. A general form of the Cholesky algorithm is given in **Fig. 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, which are described as follows:

- **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.

```
for j = 1 to n
    for k = 1 to j-1
        for i = j to n
            a(i,j) = a(i,j) - a(i,k) x 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. General form of the Cholesky algorithm

- **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.

We implemented the column-oriented variation in our work. This variation can be better understood if the algorithm is written in the pseudo-code form shown in **Fig. 2**. We can identify two distinct subtasks that need to be carried out in the course of a regular column-oriented Cholesky algorithm. 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. 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 on a column 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 either assign 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 [**6**]; this is the one we adopted in our implementation. Hence, all the *cmod* and *cdiv* subtasks associated with a column are carried out in the same processor.

```
Cholesky( ) {
    for j = 1 to n
        for k = 1 to j-1
        Modify col. j by A [j, k] x col. k          …. cmod: Sub-task 1
        end
        Divide col. j by the square-root of A [j, j]    …. cdiv: Sub-task 2
    end
}
```

Figure 2. Subtasks in the column-oriented Cholesky algorithm

1.2 FPGAs

FPGAs are ubiquitous in the field of configurable computing, a field that has drawn considerable attention since their invention. FPGAs are general-purpose programmable devices that can be configured appropriately to implement the desired digital designs [3]. 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, Actel and Xilinx. We used in this project 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) interweaved 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. 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 input and produce as 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 shorter design and development cycles made them appropriate for certain tasks, especially as tools to prototype 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 (Application-Specific Integrated Circuits) and fully-custom silicon chips. The clock frequency of FPGAs is now high enough to satisfy the demand of many commercial applications. Moreover, their capacity is large enough to often contain more than one processor. This has led to their inclusion in real-time systems, functioning as microcontrollers or digital signal processors.

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/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, standards organizations are being setup to standardize interfaces for IP cores, with a vision of making their use widespread. In our design, we used instances of Altera's Nios soft-processor core which is a general-purpose RISC processor. We used the 32-bit variant. In addition, we also used IP cores for UART (Universal Asynchronous Receiver Transmitter) and on-chip memory.

2. OUR PARALLEL ARCHITECTURE

2.1 A Shared-Memory Multiprocessor

Our multiprocessor system for the parallel Cholesky algorithm contains three or more instances of the 32-bit variant of Altera's Nios soft-processor IP core. In this text, we have used the term "Nios-32" to mean the 32-bit variant of Nios. The multiprocessor system implements an asymmetric shared-memory architecture. Each processor 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 the architecture is shown in **Fig 3.**

As shown in **Fig 3**, although there are no direct connections between the three Nios-32 processors, communications are possible via shared-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 our 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 slave processors. Note that it is the slave processors which participate in matrix factorization. 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 on the user side.
- 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 GERMS, the Altera monitor program that accepts commands from the host machine. This allows the user to interact with the system at run-time. The other two processors execute the parallel Cholesky program.

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

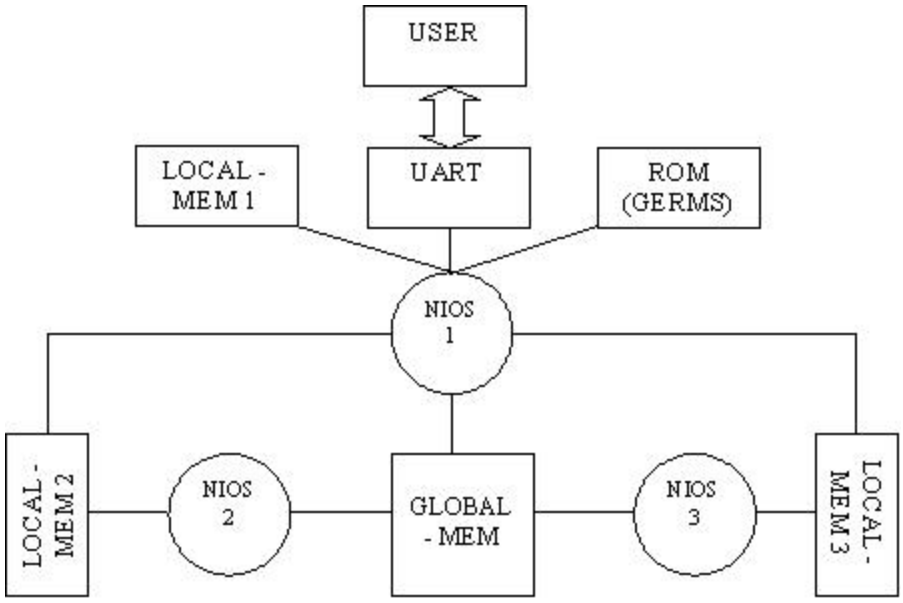

Figure 3. Architecture of the 3-Nios system

2.2 Nios Configurable Processor IP Core

In our implementation, we chose the Nios 2.0 soft-processor core as our processing element. Altera defines Nios as a "pipelined general-purpose RISC microprocessor" [**1**]. Nios is configurable so that its features can be selected by the user from a variety of available options depending upon memory and logic requirements, such as floating-point support, support for hardware interrupts, etc. Nios also supports custom instructions. These instructions can be added by the user to a Nios 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 Builder™ software environment [2]. 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 cycles, 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.

2.3 Inter-Connections via the Avalon Bus System

Our architecture uses the Avalon bus system, provided by Altera with the SOPC Builder, to connect processors and peripherals. This system 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 master, the Avalon bus system uses slave-side arbitration, which is automatically enabled by the SOPC Builder whenever the situation arises. This means that an arbitration unit is added on the slave side. The Avalon bus system uses a weighted round-robin scheduling policy whenever arbitration is needed. The SOPC Builder uses default values and these weights can be changed by the user. The concept of the Avalon bus system significantly simplifies hardware design as well as software programming. The user does not have to program an arbitration policy.

In our implementation, 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 Altera's SOPC Development Board. 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 LEs and 442,368 RAM bits [**2**]. Other important features of this board are:

- Support for six clocks, including a BNC connector that can be attached to an external oscillator. The largest on-board clock frequency is 66 MHz. In our setup, we 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. In our setup, we only made use of the on-chip memory.

- Various interfaces such as the IEEE Standard 1394a (Firewire), RS-232 serial, USB, as well as 10/100 Ethernet with full- and half-duplex communications. In this setup, we made use of 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, and finally transferring the Cholesky factor L from the global memory back to the host PC.
- JTAG interface for configuring the FPGA.

2.5 Parallel System Setup

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

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

The above components were individually added and configured using Altera's SOPC Builder tool that came with Quartus II. The system was synthesized using Altera's Quartus II 2.0, which is an integrated environment for logic design and synthesis. Upon synthesis, the design consumed 23% of the LEs, 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 clock, one for the global reset and two for the RS-232 interfaces.

3. SOFTWARE IMPLEMENTATION

3.1 The Parallel Cholesky Factorization Algorithm

The central concept of the algorithm described in George, et al. [**6**] and adopted in the current work is a task queue which contains tasks to be performed by processing elements. The tasks can be divided by rows, columns, or sub-matrices, although the column based division can achieve a higher degree of efficiency [**6**] and this is the variation we will concentrate on in this paper. 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, 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 j^{th} 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 **Fig. 4** below.

```
Cholesky( ) {
    for j = 1 to n
    begin
        Pick up task Tcol(j) from task-queue
    end
}
```

Figure 4. 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, where *k* = 1, 2, …, *j*-1. For a given value of k, this can be denoted by *cmod(j,k)* and its pseudo-code is shown in **Fig. 5**.
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 5. Pseudo-code for the *cmod* routine

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 latter 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, [**6**] mentions the use of an array, *ready[.]*. This data structure has been used in our program too.

```
Tcol( j ) {
    for k = 1 to j-1
    begin
       Wait until the ready[k] flag has been set
       Perform cmod(j,k) operation
    end
    Perform cdiv(j) operation
    Set the ready[j] flag
}
```

Figure 6. Subtasks in the *Tcol* routine

A pseudo-code for the task *Tcol(j),* in terms of the *cmod* and *cdiv* operations, is shown in **Fig. 6**. 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 **Fig 7.** This figure 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 potential parallelism in this algorithm. Consider a scenario for two processors, P_1 and P_2, where each processor handles each time a single task *Tcol(j)*. Without loss of generality, we can assume that P_1 starts working on *Tcol(1)* which contains only a single sub-task *cdiv*(1). During this time, processor P_2, which is currently idle, will pick up *Tcol(2)* from the queue. *Tcol(2)* consists of sub-tasks *cmod(2,1)* and *cdiv(2)*, which need to be carried out in this order. For sub-task *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 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),* as shown in **Fig. 8**. The authors of the original paper [**6**] name this "self-scheduling" since it does not require any explicit load balancing on the part of the programmer.

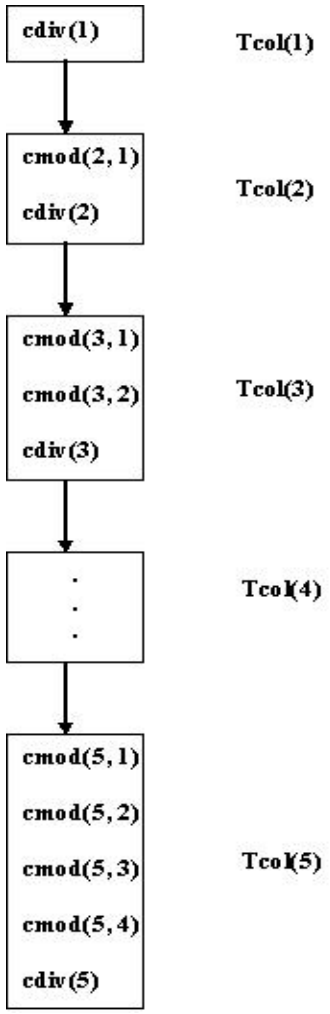

Figure 7. Task and subtasks for a matrix of order n = 5

Since 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 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 [**6**] is called the *busy-wait* time (shown in **Fig 7**), is of a lesser order than the total execution time of the entire program. Another advantage is that, as a particular column is modified by a single task run on a single processor, there is no need for external synchronization. This algorithm is well suited to MIMD architectures as the granularity level is quite large. 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.

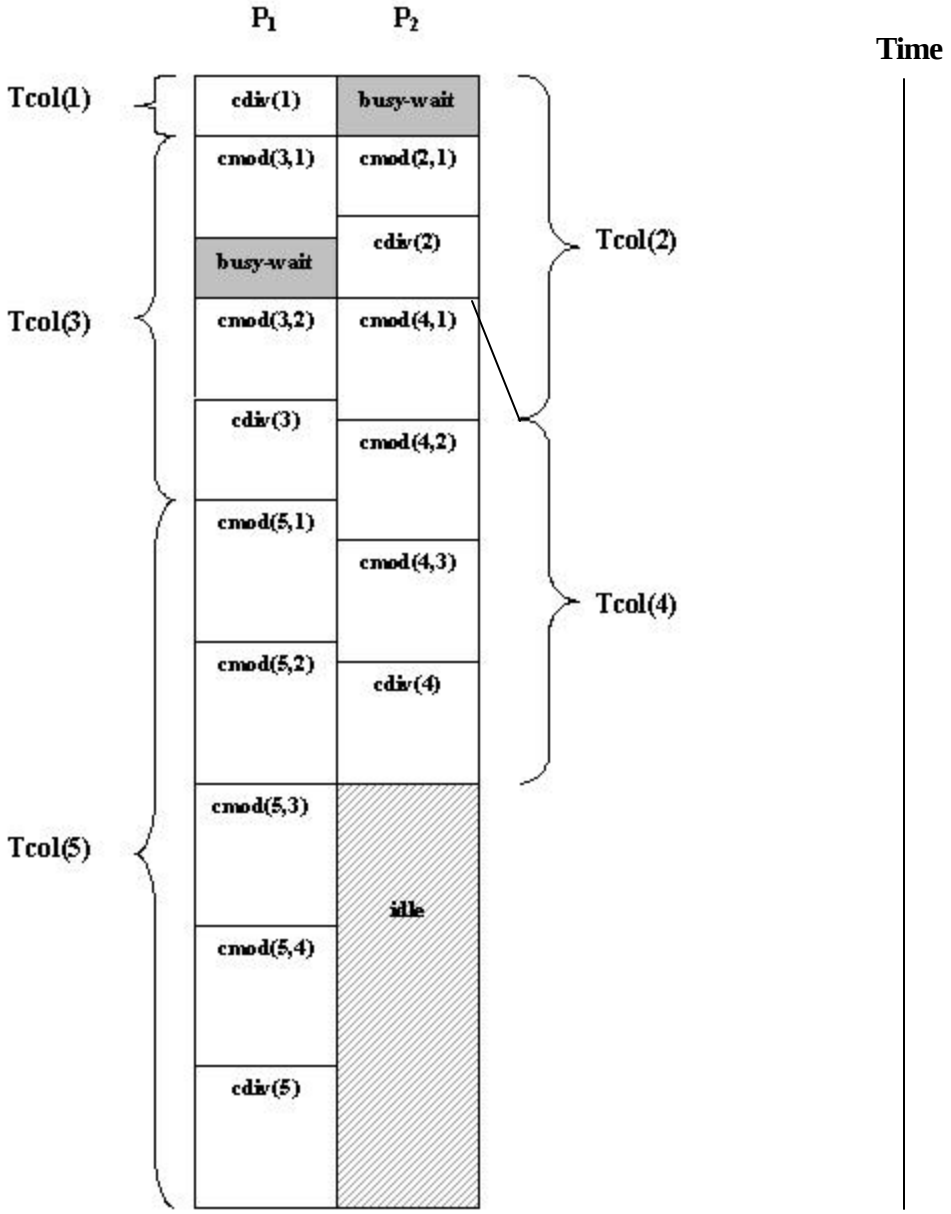

Figure 8. Processor scheduling diagram for a matrix of order n = 5

We now calculate the complexity of our algorithm by making three assumptions:

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

Then, to calculate each column j of the Cholesky factor L, we need

$$\Gamma\{\,\mathrm{Tcol}(j)\,\} \quad = \quad j\,(n-j+1)$$

floating-point operations. The total number of operations to completely factorize a matrix of order n is

$$\sum_{i=1}^{n} \Gamma\{Tcol\,(i)\} = O\,(n^3)$$

3.2 Parallel Cholesky Program

As the parallel system in our 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 we had to make use of explicit memory references in the code for each processing element to access structures in the global memory. As each of the Nios processors came with a C compiler, we made use of pointers to make our 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. Then, the addresses of the various globally accessible flags and arrays were fixed within this global block using C-based pointers. **Fig. 9** displays the section of the code which was used to declare these global variables. The details are provided in **Table 1**.

```
/*****************************************************************************/
/*              GLOBAL VARIABLES MEMORY MAP
/*****************************************************************************/
        flag= base;

        semaphore = base+1;

        task_id = base+2;

        order = base+3;

        n = *order;

        ready = base+4;

        processor = ready+n;

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

Figure 9. Declaration of global variables in the source code

Variable Name	Data Type	Description
base	integer pointer	Base address.
flag	integer	A flag used to signal that the data matrix has been stored into the global memory (i.e., the processors can begin factorization).
semaphore	integer	Protects the critical section in the program (for picking up a unique task from the queue).
task_id	integer	Stores the number of columns still left to be worked upon. The current value also acts as the task-id for a processor. The processor 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 by one. Factorization is complete

		when the value is equal to that of the order of the matrix.
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 column are ready to be used by other columns, the value in that particular element of the array is set to non-zero. 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.

Table 1. Global variables in the parallel Cholesky program

3.3 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 the SOPC Builder. Each processor then waits until the input matrix is loaded into memory. To implement this, each processor checks if the variable *flag*, whose initial value is zero, is set. This variable is set once the input matrix is loaded using the program that uploaded the matrix into the global matrix. This program 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. In our setup, this is the case when the value of the variable *task_id* is equal to the value of the variable *order*.

1. The input matrix is uploaded into the global memory using the *master* processor. Once the input matrix is in place, the program will also set *flag* to a non-zero value. In our setup, 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 let us calculate with relative ease the serial and parallel run times, and, hence, get the speedup values for the various test matrices.
2. The program which uploads the input matrix in the global memory will continue to run during the factorization process. In our setup, it is 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 as the number of clock cycles required for the factorization. This number multiplied by the clock frequency gives the actual run time. If need be, this program can also be used to download the Cholesky factor onto the host PC.

4. EXPERIMENTAL RESULTS AND ANALYSIS

4.1 Performance Results

We tested our 2-processor system for actual factorization using matrices of various orders and the results are summarized in **Table 2.** For n = 48, we used the BCSSTK01 test matrix (sparse matrix containing 224 non-zero terms) which was obtained from the Harwell-Boeing matrix set available at the Matrix Market [**13**]. 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 in the lower triangular matrix. Thus, 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 2**, the speedups as well as the efficiency in general increase as the order of the matrix increases. Note that, except for 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 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 [**6**] and at the end when processors are idle because there are no more tasks in the queue. During the busy-wait time the processors are waiting for the calculations on the preceding columns to be completed so that the terms on these columns can be used in further calculations. This time influences the speedup and efficiency obtained. However, because this time is of a lesser order than the run-time of the factorization process, as the order of the input matrix increases the busy-wait time becomes negligible compared to the total sequential run-time of Cholesky factorization. Thus, the speedup and efficiency values approach their ideal values. This is seen in **Table 2** and graphically represented in **Fig. 10** for a matrix of order n = 48, where the efficiency is as high as 99.9%.

Order of matrix (n)	**Sequential run-time (ms)**	**p = 2**		
		Parallel run-time (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

Table 2. Comparison of speedup and efficiency for p = 2

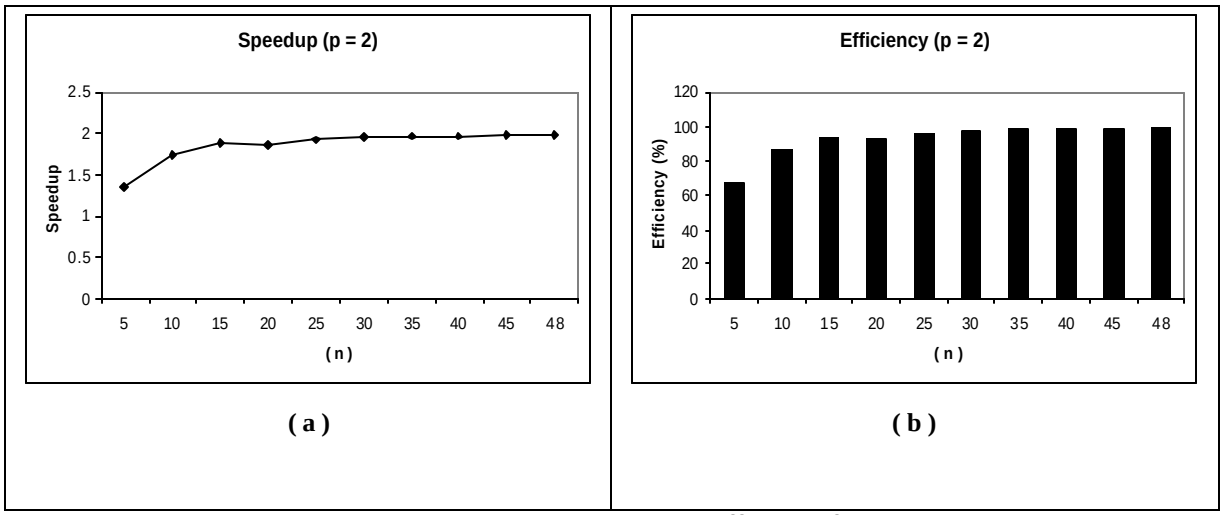

Figure 10. (a) Speedup and (b) efficiency for p = 2

We also obtained speedup and efficiency results for a 4-Nios system, out of which three processors performed the factorization (p = 3) while the fourth was the master processor. We compare in **Fig. 11** the results obtained for p = 3 with those for p = 2.

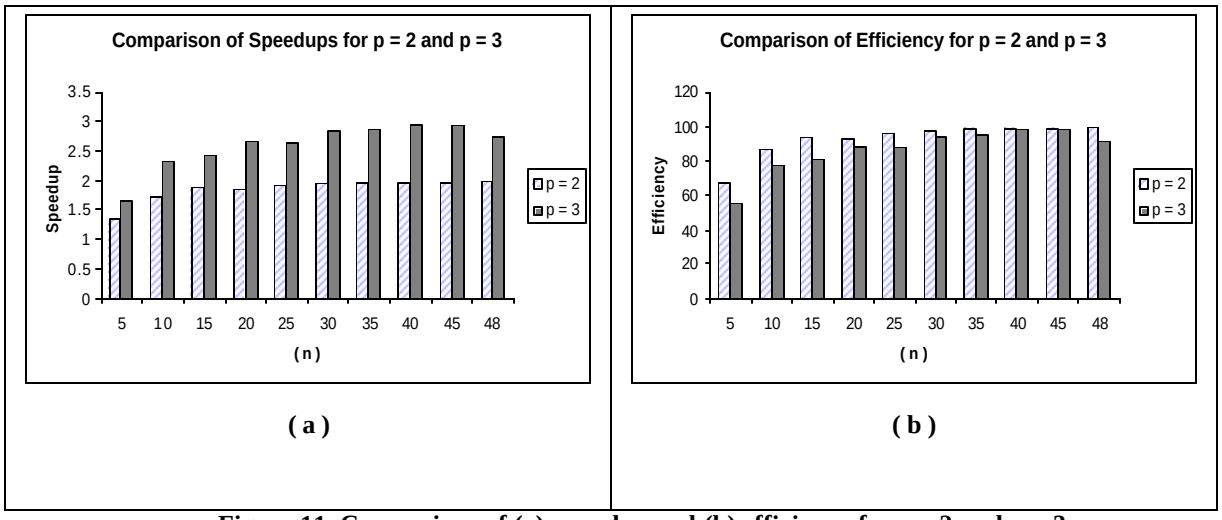

Figure 11. Comparison of (a) speedup and (b) efficiency for p = 2 and p = 3

One can observe from the plots in **Fig. 11** that while the speedup for p = 3 is much higher than that for p = 2, the latter performs much better in terms of efficiency. This could happen because for p = 3 the contention for memory resources will be higher. Moreover, the total amount of time the processors are idle because of *busy-wait* for 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

speedup for the factorization process as the number of participating processors (p) increases.

4.2 More Observations

For the sake of curiosity, in the 3-Nios case (p = 2) we also compared the sequential times of the two processors P_1 and P_2, that were involved in the factorization process. The results are provided below in **Table 3**.

Order of matrix (n)	Run-time for P_1 (ms)	Run-time for P_2 (ms)
5	8.86	8.52
10	67.16	64.41
15	220.19	211.64
20	513.44	493.46
25	992.40	953.72

Table 3. Comparison of sequential run-times for Cholesky factorization on P_1 and P_2

As one can see from **Table 3**, the run-times for the sequential version of Cholesky factorization on P_1 are larger 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 similar lengths for the two processors. Note that in the current project the placement and routing of hardware resources 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 skewing 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. [**14**] discusses the effects of locality in single-chip multiprocessors and reports performance improvements obtained by controlling it. One can employ more advanced features provided by the development software to tightly control resource placement.

CONCLUSIONS

With recent advances in FPGA technology, multiprocessor systems with very substantial capabilities can now be embedded into single FPGAs. This, in turn, can speed up several parallel processing applications. We have shown in this paper that matrix operations, such as Cholesky factorization, benefit tremendously from such hardware setups and appropriate hardware/software codesign. Since several applications in engineering and science have requirements similar to Cholesky factorization, our results can give further impetus to the design of parallel computing engines implemented with configurable devices. Eventually the result will be high performance implementations at low cost.

REFERENCES

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

[2] 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.

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

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

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

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

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

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

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

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

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

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

[13] Matrix Market, *BCSSTK01 Test Matrix*, http://math.nist.gov/MatrixMarket/data/ Harwell-Boeing/bcsstruc1/bcsstk01.html, (referred to in: 2003)

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

[15] X. Wang and S.G. Ziavras, "Parallel LU Factorization of Sparse Matrices on FPGA-Based Configurable Computing Engines," Concurrency and Computation: Practice and Experience, 16.4:319-343, April 2004.

[16] X. Wang and S.G. Ziavras, "Parallel Direct Solution of Linear Eqautions on FPGA-Based Machines," *11th Int. Conf. Paral. Distr. Real-Time Syst*., Nice, France, April 22-23, 2003.

[17] X. Xu and S.G. Ziavras, "Iterative Methods for Solving Linear Systems of Equations on FPGA-Based Machines," *18th Int. Conf. Comput. Appl.*, Honolulu, Hawaii, March 26-28, 2003.