ABSTRACT

SPARSE MATRIX PRODUCT IMPLEMENTATION ON FIELD PROGRAMMABLE GATE ARRAYS (FPGAS)

by
Amit Mahendra Sheth

If dense matrix multiplication algorithms are used with sparse matrices, they can result in a large number of redundant calculations, as numerous elements in sparse matrices are zero valued, thus available resources and time may be wasted. The algorithm discussed here aims to take advantage of the sparseness of the matrices by multiplying only non-zero elements.

The NIOS development board from Altera is used for implementing the above algorithm. First a sequential program in the C programming language is downloaded onto the FPGA and run by the NIOS soft-processor. Then the same board is also used for a parallel implementation of the above algorithm using three NIOS soft-processors within the same FPGA.

Such an approach is very critical because current FPGAs do not contain enough resources to solve large problems. For example, we cannot build large memory systems within FPGAs so we need to employ algorithms that have rather limited memory requirements. Our proposed matrix multiplication algorithm for sparse matrices uses the available memory space very cautiously and also results in good execution times. Performance results testify to this fact.

SPARSE MATRIX PRODUCT IMPLEMENTATION ON FIELD PROGRAMMABLE GATE ARRAYS (FPGAS)

by
Amit Mahendra Sheth

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 Electrical Engineering

Department of Electrical and Computer Engineering

May 2003

APPROVAL PAGE

SPARSE MATRIX PRODUCT IMPLEMENTATION ON FIELD PROGRAMMABLE GATE ARRAYS (FPGAS)

Amit Mahendra Sheth

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

Dr. Vishwani D. Agrawal, Thesis Co-advisor — Date
Visiting Professor of Electrical and Computer Engineering, and
Center for Advanced Information Processing, Rutgers University

Dr. John D. Carpinelli, Committee Member — Date
Associate Professor of Electrical and Computer Engineering, and
Computer and Information Science, NJIT

Dr. Roberto Rojas-Cessa, Committee Member — Date
Associate Professor of Electrical and Computer Engineering, NJIT

BIOGRAPHICAL SKETCH

Author: Amit Mahendra Sheth

Degree: Master of Science

Date: May 2003

Undergraduate and Graduate Education:

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

- Bachelor of Engineering in Electronics,
  Father Conceicao Rodrigues College of Engineering, Mumbai, India, 2001

- Diploma in Digital Electronics
  Bombay Institute of Technology, Mumbai, India, 1998

Major: Electrical Engineering

Presentations and Publications:

Amit M. Sheth and Jacob Savir
"Single Clock, Single Latch, Scan Design,"
The IEEE Instrumentation and Measurement Technology Conference (IMTC/02), Anchorage, Alaska, USA, May 2002.

Scholarships and awards:

- Research assistantship for 3 semesters under Dr. Sotirios G. Ziavras, Department of Electrical and Computer Engineering, N.J.I.T. (Newark), USA, for the Power Grid Project.

- Research Assistantship under Dr. S. S. S. P. Rao, Department of Computer Science and Engineering, I.I.T. (Bombay), India, for Vikram Sarabhai Space Center project

- Travel Grant for presenting a research paper titled "Single Clock Single Latch Scan Design", IEEE-IMTC-02, Anchorage, AK, USA, May 2002

To my beloved family, friends and teachers

ACKNOWLEDGMENT

I would like to express my deepest appreciation to my advisor, Dr. Sotirios G. Ziavras for his valuable guidance and support throughout the research work. Without his timely and valuable suggestions, it would not have been possible to accomplish the work that I have done which to me has been a very satisfactory experience.

My deepest gratitude to Prof. Jacob Savir who was the catalyst for me to work on this project and provided me with a wonderful opportunity to become involved with this highly prestigious project.

I also would like to express my sincere thanks to my co-advisor, Dr. Vishwani D. Agrawal, who despite his busy schedule took time out to provide me with valuable guidance and suggestions which proved very helpful.

I also appreciate the efforts and suggestions of the committee members, Dr. John Carpinelli and Dr. Roberto Rojas-Cessa. I would like to thank them too.

I thank the distinguished faculty of the ECE department who have been excellent guides throughout my graduate study and have provided me with the skills and expertise to be a successful professional in my field of study in future.

And last but not the least, I take this opportunity to thank my Family and also my dear friends for their unflinching support and encouragement throughout my Graduate study and who have been a great source of inspiration for me at every step of my academic career. I shall remain indebted to them for this and my success would not have been possible without their ceaseless efforts.

TABLE OF CONTENTS

Chapter		Page
1	INTRODUCTION	1
	1.1 Matrix Multiplication	1
	1.2 Why FPGAs over Traditional Computers	2
	1.3 Motivation and Objectives	3
2	MATRIX MULTIPLICATION ON FPGAs	5
	2.1 SOPC Board	5
	2.2 Algorithm	8
	2.3 Parallel Implementation	12
	2.4 System Development	15
3	PERFORMANCE RESULTS	27
	3.1 Sequential Implementation Results	27
	3.2 Parallel Implementation Results	30
4	CONCLUSIONS	34
APPENDIX A	Source Code For Sequential Implementation	35
APPENDIX B	Source Code For Parallel Implementation	41
REFERENCES		54

LIST OF TABLES

Table **Page**

3.1 Performance Results for Sequential Implementation (8 X 8 matrices) 27

3.2 Performance Results for Parallel Implementation 30

LIST OF FIGURES

Figures **Page**

1.1 Inner product of row and column vectors 1

1.2 Matrix multiplication example 2

1.3 Sparse matrix 3

1.4 Sparse matrix 3

2.1 SOPC Board 6

2.2 Flowchart of our matrix multiplication algorithm 10

2.3 Flowchart (continued) 11

2.4 Parallel architecture 12

2.5 Parallel architecture interconnections 15

2.6 NIOS processor CPU architecture 16

2.7 NIOS processor hardware configuration 17

2.8 NIOS processor software configuration 18

2.9 NIOS processor local RAM 19

2.10 Global RAM 20

2.11 Boot ROM configuration 21

2.12 GERMS monitor 21

2.13 UART configuration 22

2.14 Hardware timer configuration 23

2.15 Priorities 24

2.16 Programmer 25

Figures **Page**

3.1 Graph of CPU clock cycles versus matrix density.....28

3.2 Graph of speed up versus matrix density compared to the conventional matrix multiplication algorithm.....29

3.3 Graph of CPU clock cycles versus matrix density for parallel implementation....31

3.4 Graph of speed up versus matrix density for parallel implementation 32

CHAPTER 1

INTRODUCTION

1.1 Matrix Multiplication

Matrix multiplication is a very useful operation in mathematics. It involves inner-vector product implementations. Let us now see how we multiply a row vector with a column vector of the same length - that is, with the same number of entries. The result is a number (which can be viewed as a 1x1 vector if one insists that the product of two vectors must be a vector).

Step 1: Row vector times column vector

$$\begin{bmatrix} 1 & 2 & 3 & 4 \end{bmatrix} \begin{pmatrix} 5 \\ 6 \\ 7 \\ 8 \end{pmatrix} = (1 \text{ X } 5) + (2 \text{ X } 6) + (3 \text{ X } 7) + (4 \text{ X } 8) = 70$$

Figure 1.1 Inner product of row and column vectors

In this example, the row and the column both have length 4, but the same pattern obviously works whenever they are of the same length; just multiply the corresponding entries and sum up the products.

Step 2: The General Case

Assume now two general matrices, say A and B. The matrix product C=A*B is formed by multiplying every row of A with every column of B, in the way described in Step 1 above. Assume n*n matrices, for the sake of simplicity. The elements C_{ij} of matrix C is calculated as

$$C_{ij} = \sum_{k=1}^{n} a_{ik} * b_{kj}$$

where $1 \leq i, j \leq n$.

The time required for this matrix multiplication is O (n^3)

$$\begin{pmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \end{pmatrix} \begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \\ 10 & 11 & 12 \end{pmatrix} == \begin{pmatrix} 70 & 80 & 90 \\ 158 & 184 & 210 \end{pmatrix}$$

Figure 1.2 Matrix multiplication example

1.2 Why FPGAs Over Traditional Computers?

Large matrices (e.g. of the order of 1000 X 1000 and higher) are traditionally solved by large super-computers such as CRAY. Workstations have also been used to solve the above problem by, first improving the algorithm, such as for Strassen's algorithm [6] that has a time complexity of O ($n^{2.376}$).

FPGAs are primarily used in application development because:

1) They can be configured as and when required by the application.
2) They are reprogrammable and can be conditioned to give hardware acceleration by offering the best of both software and hardware.
3) They are also becoming relatively inexpensive, at less than $20 per million gates.

1.3 Motivation and Objectives

To multiply two 3000 X 3000 matrices using the conventional method requires on the order of O (n^3), $3000^3 = 27,000,000,000$ floating-point multiplications and additions.

Sparse Matrices:

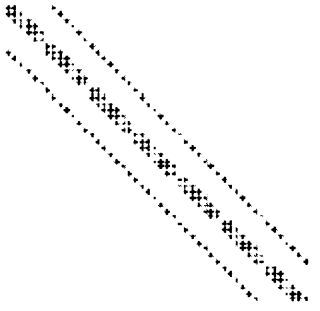

Figure 1.3 Sparse matrix

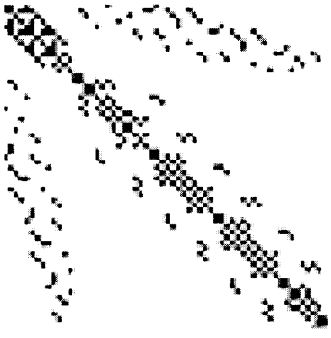

Figure 1.4 Sparse matrix

Figures 1.3 and 1.4 show examples of sparse matrices. As we can see, most of the elements in the above matrices are zero-valued. In other words, the above matrices are sparse in nature. This implies that most of the multiplications would result in zeroes and hence are redundant. These redundant calculations can be avoided by using an algorithm in which only the required elements are multiplied.

If sparse matrices are stored as regular matrices in two dimensional arrays, they will consume a lot of space in memory. Various formats are used to store sparse matrices; among them, which a popular one is the matrix market file format [4] which only stores the non-zero elements along with their row and column indices.

A sample file saved in the matrix market format is shown below:

```
5    5    8
1    1    4.7e12
2    2    3.5e-11
3    2    3.9e14
2    3    6.2e21
3    3    2.7e-05
4    3    1.3e-1
4    4    7.5e18
5    5    4.9e14
```

The first line specifies the maximum number of rows, columns and the total number of non-zero elements, respectively. From the next line onwards, the first column specifies the row index, the second column specifies the column index and the third column specifies the corresponding value of the matrix element.

CHAPTER 2

MATRIX MULTIPLICATION ON FPGAs

2.1 SOPC Board [5]

We have used an Altera board which serves as a development and prototyping platform that provides system designers with an economical solution for hardware verification. The system-on-a-programmable chipboard supports a variety of microprocessor-based designs incorporating memory, debugging, and interface resources. The development board is primarily designed for implementing microprocessor functions and other standard IP (Intellectual Property) functions in the on-board APEX FPGA device. The board includes physical interfaces for widely used standard interconnects. Control logic for the interconnects can be implemented in the device. Some of the available IP solutions include:

- Processor cores (MIPS, RISC, and Harvard architectures).
- Peripheral and I/O cores (PCI, SDRAM controllers, UART, USB, Ethernet, IEEE1394, IEEE 1284).
- Other cores developed by Altera and other Altera Mega function Partners.

The board also supports EJTAG for development and debugging of MIPS-like microprocessor functions, as well as JTAG for system testing. For additional analysis, the JTAG port can be used with the Signal Tap embedded logic analyzer available with the Quartus® II development software. Figure 2.1 shows a layout diagram of the development board.

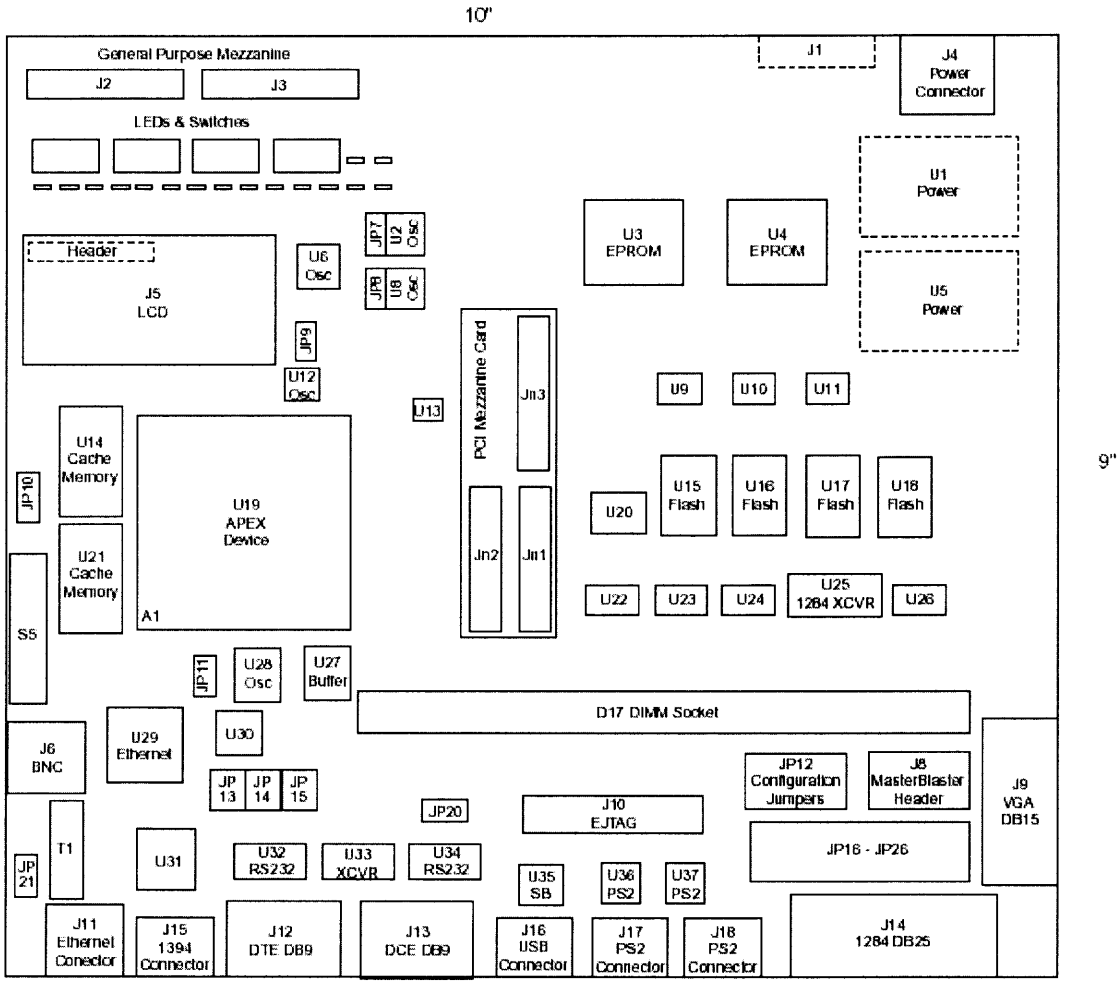

Figure 2.1 SOPC Board [5]

APEX Device

The on-board APEX EP20K1500E device features 1,500,000 ASIC-equivalent gates in a 652-pin BGA package. The device has 51,840 logic elements (LEs) and 442,368 RAM bits.

Clocks

The board supports up to six unique clocks that can be selected by the designer. The board has two BNC connectors to support communications systems designs. The APEX global clock input is driven by a 66-MHz oscillator or by an external clock via a BNC

connector. The second global clock signal is connected to an oscillator that can drive a PCI function at either 33 or 66 MHz.

Memory

To support processor functions implemented in the APEX device, the board includes a memory system consisting of the following:

- Volatile memory: 64 Mbytes of synchronous DRAM, organized as 8 Mbytes X 64 Mbytes.
- Non-volatile memory: 4 Mbytes of Flash memory and a 256-Kbyte EPROM memory.
- Pipelined cache memory with burst SRAM organized as 256 Kbytes X 32 Kbytes.

Programming the APEX Device

The EP20K1500E board cannot store configuration data for the APEX device; you must configure the device directly via the JTAG interface. You can program the APEX device directly using the Quartus software version 2000.02 and higher, any version of the Quartus II software[9], or the MAX+PLUS® II software version 9.5[9] and higher using either the MasterBlaster™ or ByteBlasterMV™ cable.

2.2 Matrix Multiplication Algorithm

The algorithm comprises three main parts:

1) Pre-conditioning
2) Main program
3) Post conditioning

Assume that matrix A has m rows and n columns, and matrix B has prows and q columns. [Note that n must equal p.]

Then each of the above parts is as explained below:

1) Pre-conditioning:

a) This involves sorting matrix A in column order and matrix B in row order.

b) Matrices in the matrix market format for sparse matrices are already sorted in the column order. So matrix A need not be sorted.

c) Matrix B, however, needs to be sorted in row order.

d) The Quick sort algorithm is used to sort matrix B in column order..

2) Main program:

a) The inputs are two matrices in the matrix market format for sparse matrices.

b) The resultant matrix is obtained as a two dimensional array with m rows and q columns.

The idea is as explained in brief:

a) For a particular element in the A matrix in column 'i', all the elements of the B matrix in row 'i' are multiplied and stored in the result matrix C.

b) The above step is repeated for all the non-zero elements in matrix A. However, in subsequent iterations the result of the multiplication is added to the previous result for the corresponding element in the C matrix.

3) Post processing:

The resultant matrix C is represented as a two dimensional array. It may be required to have the resultant matrix in the matrix market format, in which case some post processing will be required.

Figure 2.2 shows a flowchart for this algorithm.

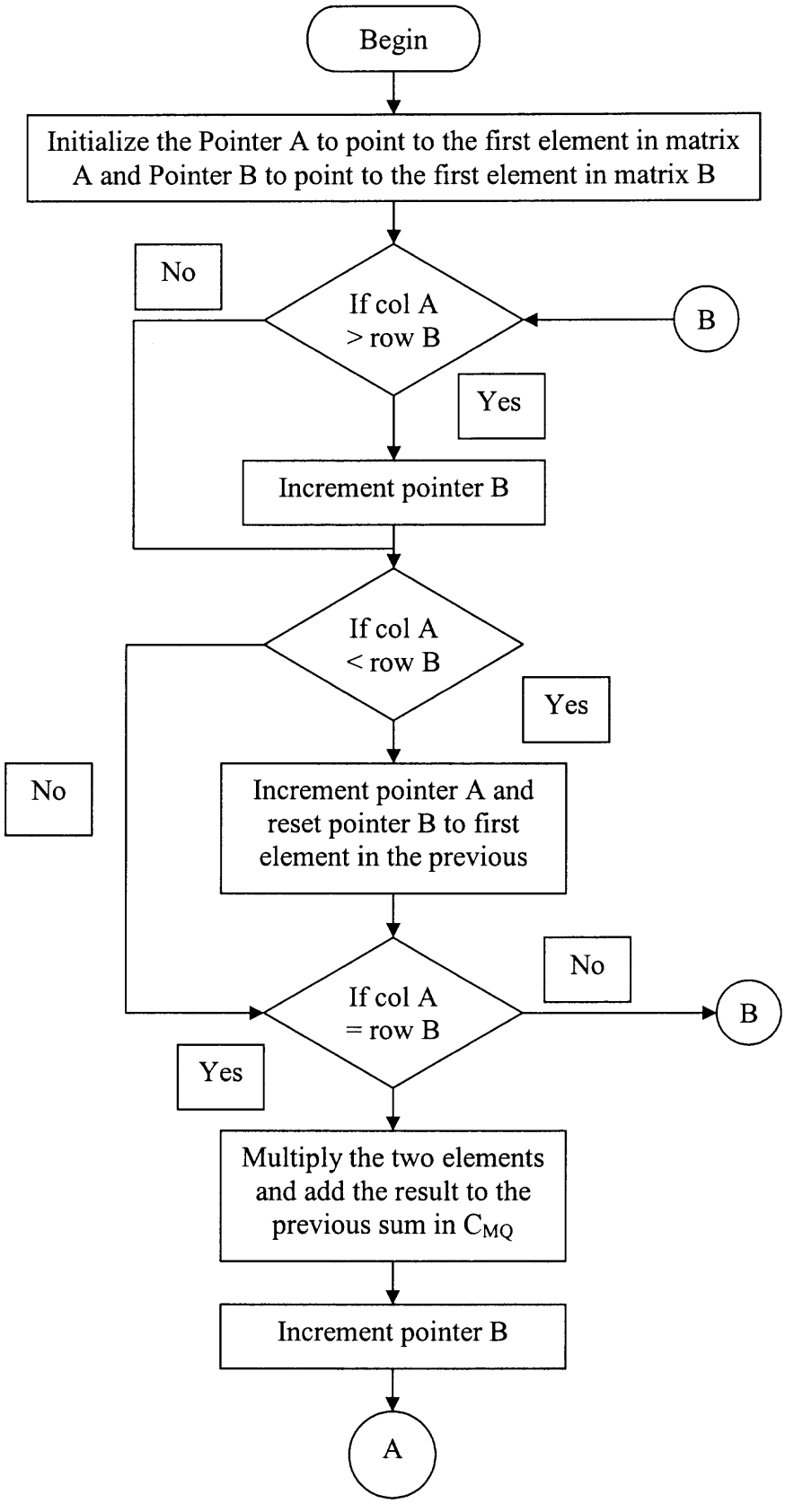

Figure 2.2 Flowchart of our matrix multiplication algorithm

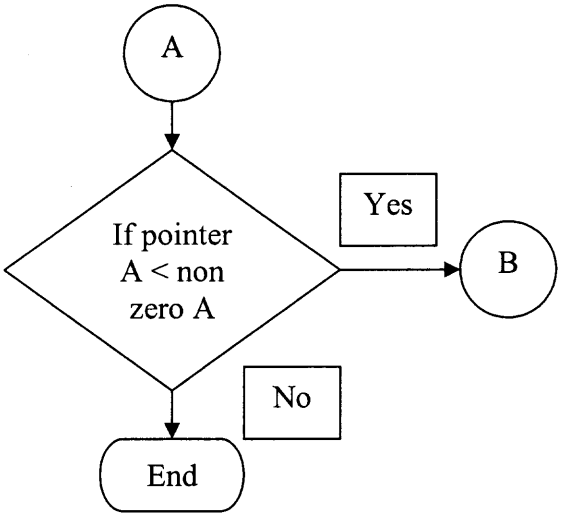

Figure 2.3 Flowchart (continued)

2.3 Parallel Implementation

A parallel architecture[7] [8] has been designed for the implementation of the matrix multiplication algorithm described earlier. A block diagram is shown in figure 2.4.

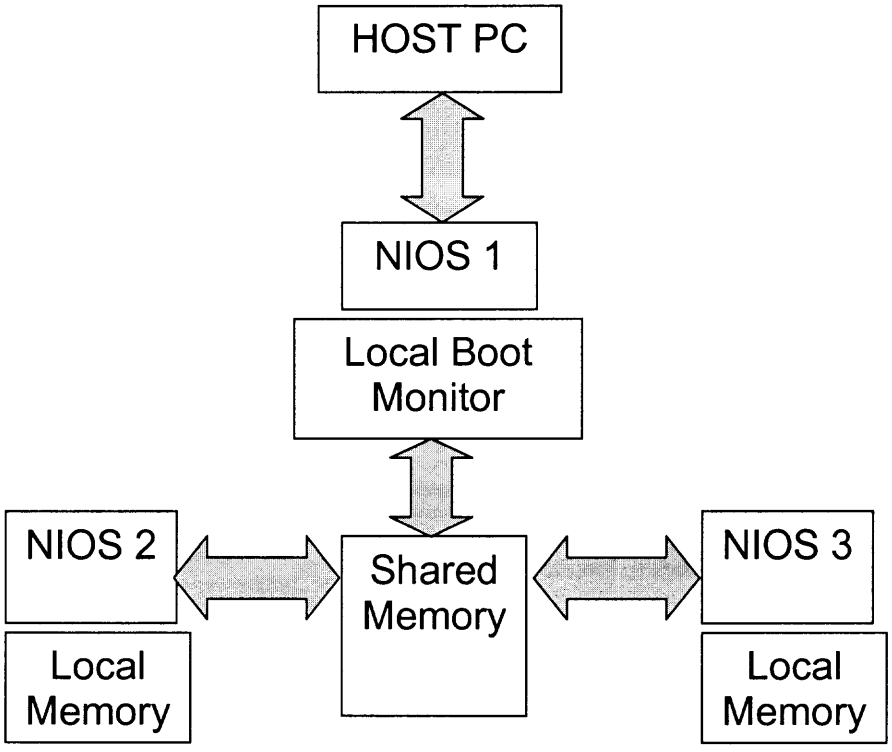

Figure 2.4 Parallel architecture

More details of the architecture follow:

1) The above architecture contains three NIOS soft processors. The restriction on a maximum of three processors is mainly due to the size of the FPGA on the current development board.

2) The NIOS 1 processor contains the Boot ROM and is also connected to the UART for communication with the host PC. It is used to download programs as well as data from the host PC into its own internal memory as well as that of the of the other two processors. It also takes part in the matrix multiplication process.

3) The NIOS 2 and NIOS 3 processors only take part in the matrix multiplication process. These processors have their own local memory (on chip) and they also share a common global on-chip memory (which is the referred to as the Global RAM).

4) These processors get information from NIOS 1 processor regarding the location of the program to be run and data to be operated on. This information is passed using semaphores.

5) In the current implementation, the matrix multiplication program along with the data is first downloaded in to the NIOS 2 and NIOS 3 processor's internal memories.

6) Using the external reset switch both the processors (NIOS 2 and NIOS 3) are reset to their starting address from which they start running the downloaded program. This program waits in an infinite loop until the main controlling program is downloaded into internal memory of NIOS 1 processor.

7) Once the program is downloaded, the NIOS 1 processor starts running the program. This program initializes the hardware timer and sets a flag which brings NIOS 2 and NIOS 3 out of their wait loops; then they both start operating on their respective elements of the matrix.

8) The NIOS 1 processor also starts multiplying matrix elements assigned to it. The assignment of elements to be multiplied by each NIOS processor is done on the host PC.

9) Each sub-program generates its own output matrix in a separate area in its own local memory, so there are no write conflicts.

10) Each NIOS processor sets a flag when they have finished generating their output. The NIOS 1 processor checks for these flags after it has completed generating its own output.

11) Once all the output sub-matrices are generated, the main program combines them all to obtain the final resultant matrix. The main program also uses the hardware timer to time the entire operation.

12) The above architecture has been successfully implemented and tested for small matrices. We are still in the process of running matrix multiplication on large matrices.

2.4 System Development

The Quartus II© version 2.0 software from Altera® [9] was used for hardware design. An already existing minimal configuration file provided by Altera among a set of sample files was used as the starting point for our project; the remaining modules were added to it.

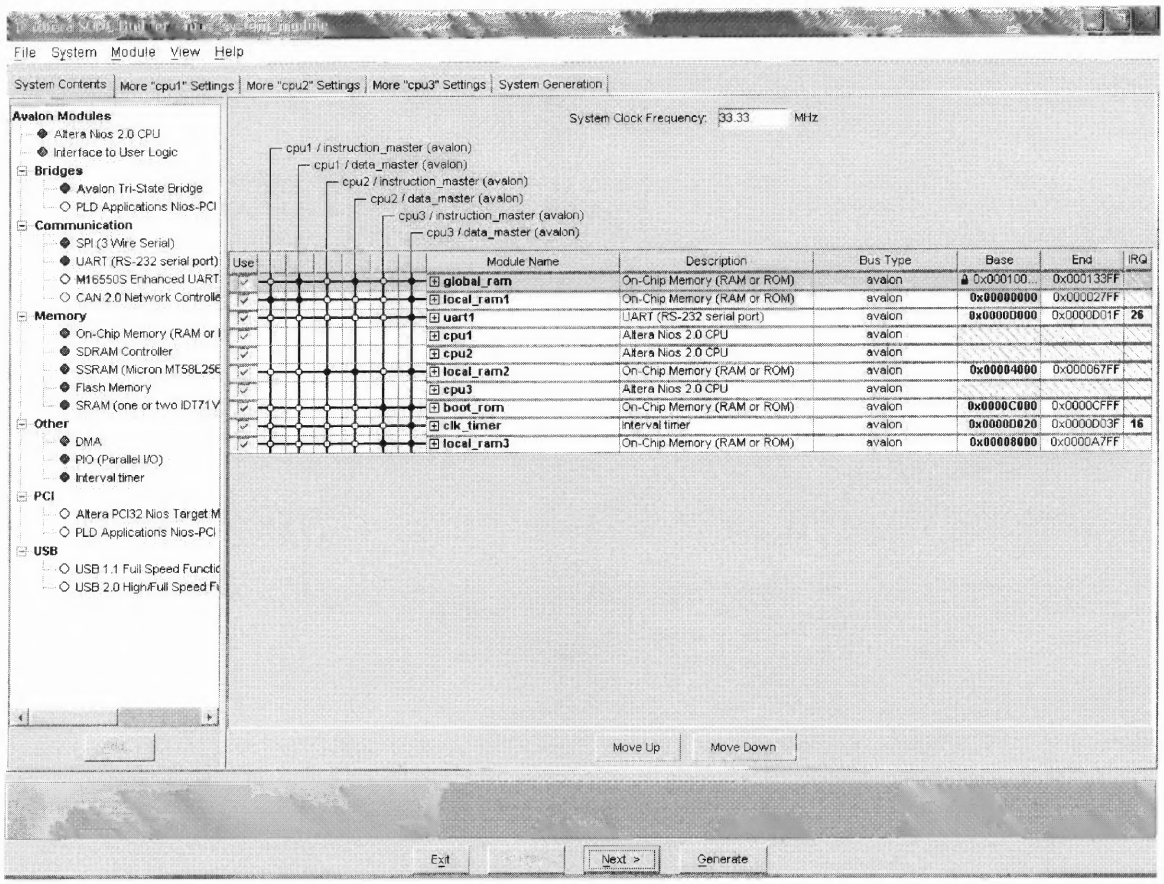

Figure 2.5 Parallel architecture interconnections

Figure 2.5 shows the interconnections for the 3-CPU configuration. It also shows the various modules included in the above system. Each of the modules is described below in more detail:

1) CPU: The above system contains three identical CPUs. Each one of the three CPUs comprises a NIOS soft-processor core whose architecture, hardware as well

as software can be customized. Snapshots of the processor configuration are shown below:

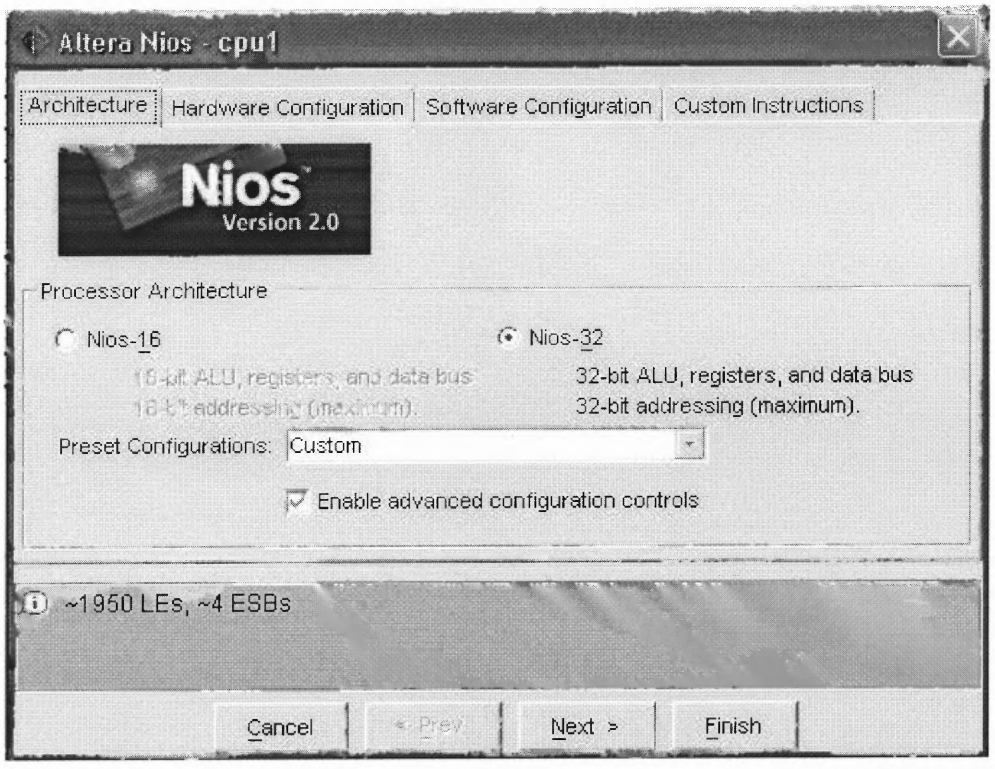

Figure 2.6 NIOS processor CPU architecture

As seen above we have selected the 32-bit NIOS configuration. This includes a 32-bit Arithmetic and Logic Unit (ALU), registers and data bus. The addressing scheme used is also 32 bits.

The next snapshot in Figure 2.7 shows the hardware configuration setup of the above CPU. A discussion of the various parameters selected is as given below:

1. 128 registers in the register file are selected, which is the lowest available option. Other options available are 256 and 512. The lowest option is selected to save ESBs (Embedded System Blocks) which can then be used for the on-chip memory implementation.

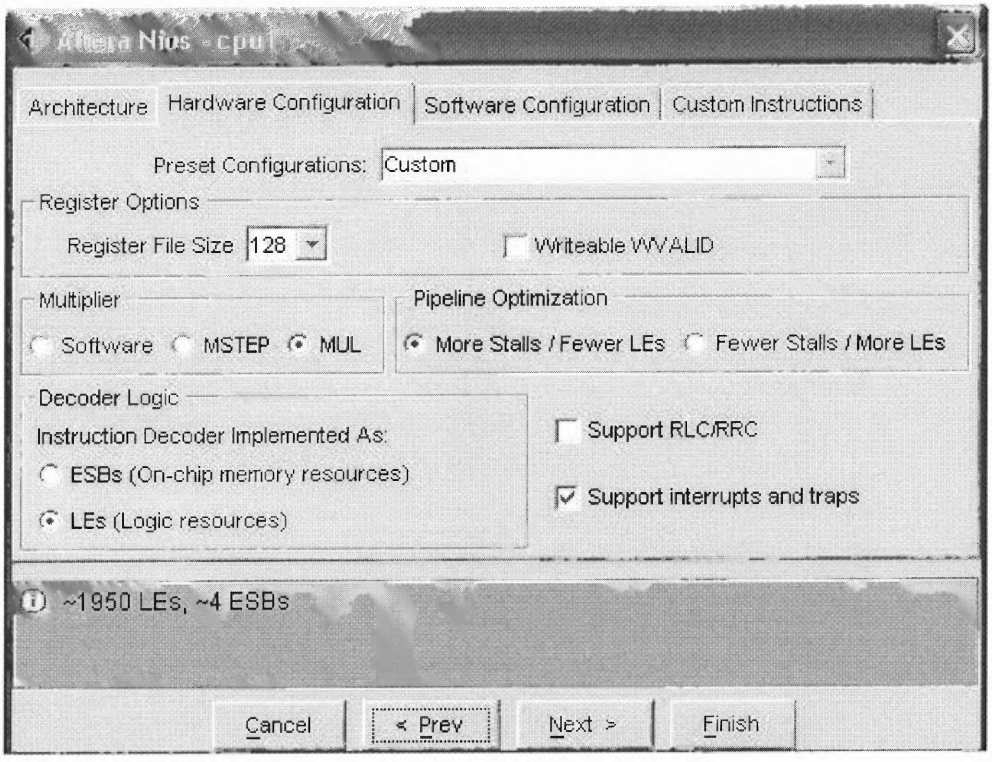

Figure 2.7 NIOS processor hardware configuration

2. A hardware multiplier is used primarily speed up operations compared to software solutions. This option consumes many LEs (Logic Elements); however, as the primary goal is to obtain good performance and multiplications constitute the major portion in our algorithm, some on-chip logic resources are sacrificed for better performance.

3. To compensate for the extra LEs used by the hardware multiplier, the "More stalls / Fewer LEs" option is selected in the Pipeline Optimizations category.

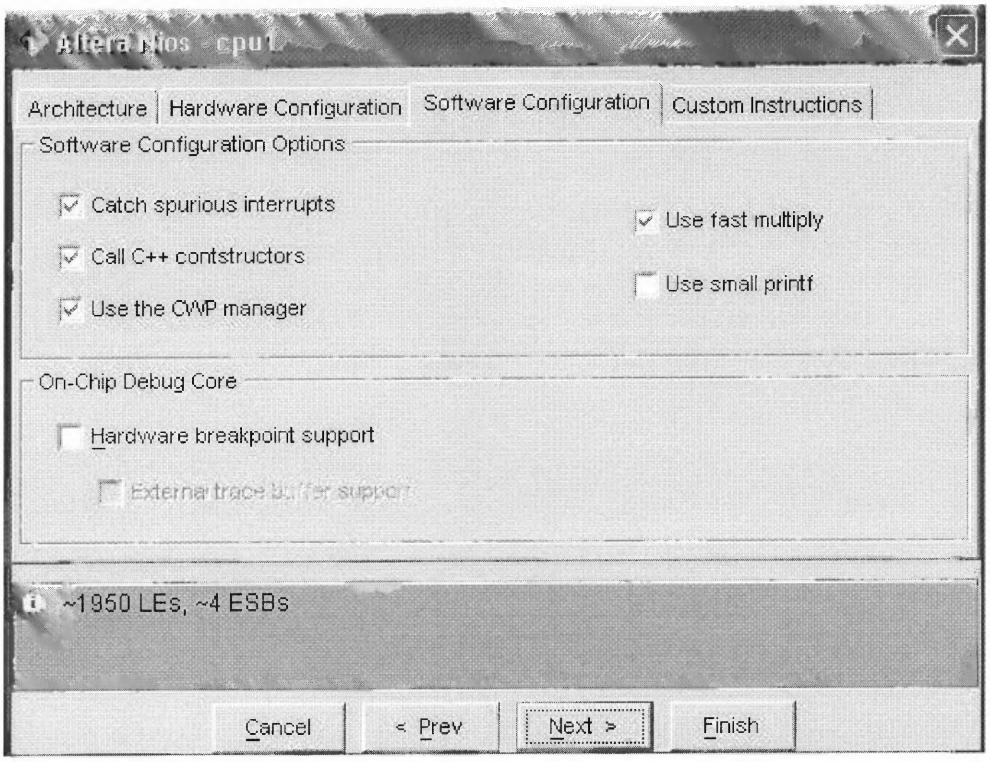

Figure 2.8 NIOS processor software configuration

The snapshot in Figure 2.8 shows the software configuration selected. Hardware breakpoint support is not used to save LEs. Other options are left to their default values.

The NIOS processor also provides the option to add custom logic in the form of VHDL source code which gets compiled with the rest of the processor core. This custom logic can be used in the form of custom instructions. For the above design, no custom logic is required.

2) Local RAM: Each of the three NIOS processors has its own local RAM which is used to download code from the host PC. The snapshot in Figure 2.9 shows the configuration of the local RAM. The data-width is 32 bits and the total memory size is 10 Kbytes. This is true for each of the three NIOS processors.

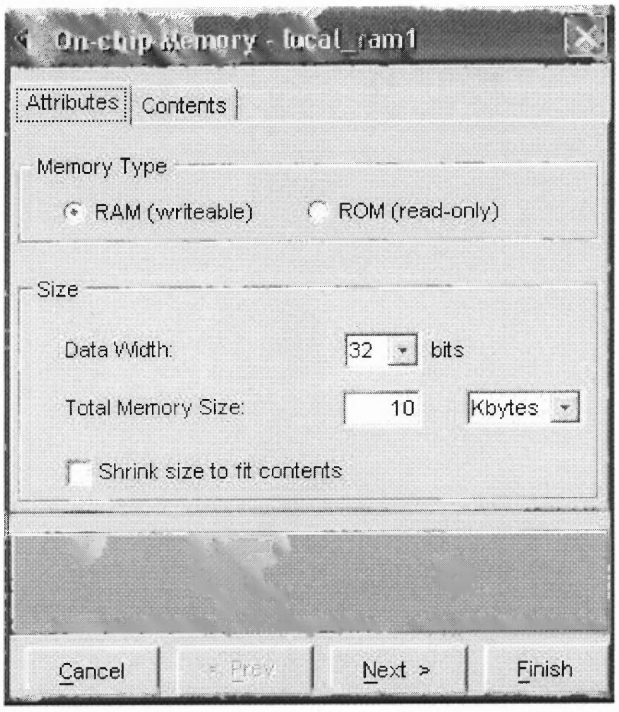

Figure 2.9 NIOS processor local RAM

The starting address of the local RAMs is given below:

1. Local RAM 1: 0x0 (for NIOS 1)
2. Local RAM 2: 0x4000 (for NIOS 2)
3. Local RAM 3: 0x8000 (for NIOS 3)

3) Global RAM: Besides the local RAM of each CPU, a global RAM is also provided which is used to store the output matrix generated by the three processors. Figure 2.10 shows a snapshot of the global RAM configuration. Note that the size of the global RAM is 13 Kbytes.

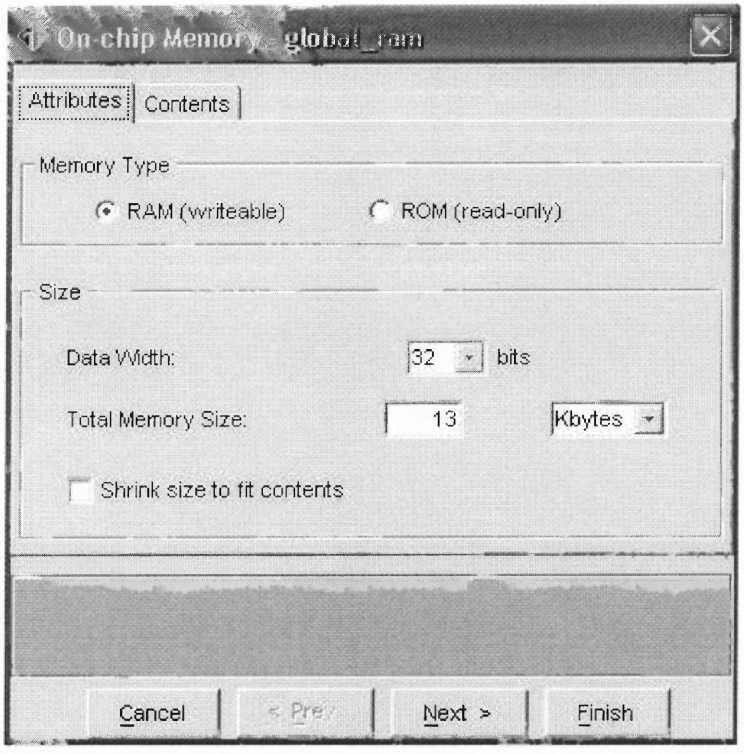

Figure 2.10 Global RAM

4) Boot ROM: This is used to store the Boot program for NIOS 1 which is connected to the UART. A snapshot of the Boot ROM configuration is shown in Figure 2.11. The Boot program here is called the GERMS monitor program. This mainly supports the communication provided by the UART between the NIOS 1 processor and the host computer. The NIOS-1 CPU's reset location is set to the starting address of the Boot ROM. Figure 2.12 shows that the GERMS monitor is built inside the Boot ROM during compilation.

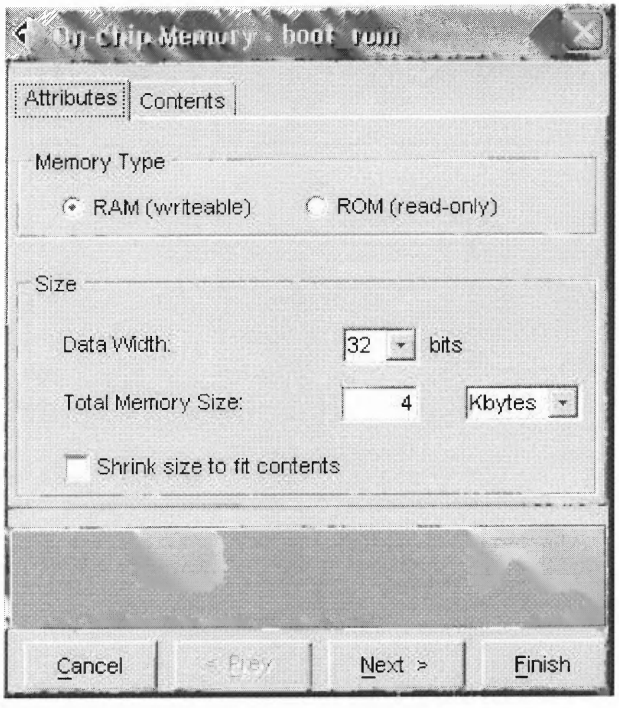

Figure 2.11 Boot ROM configuration

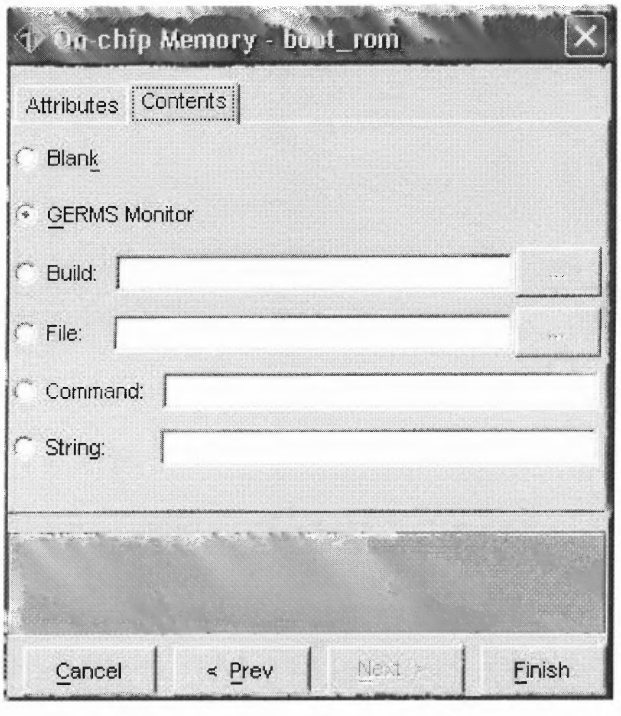

Figure 2.12 GERMS monitor

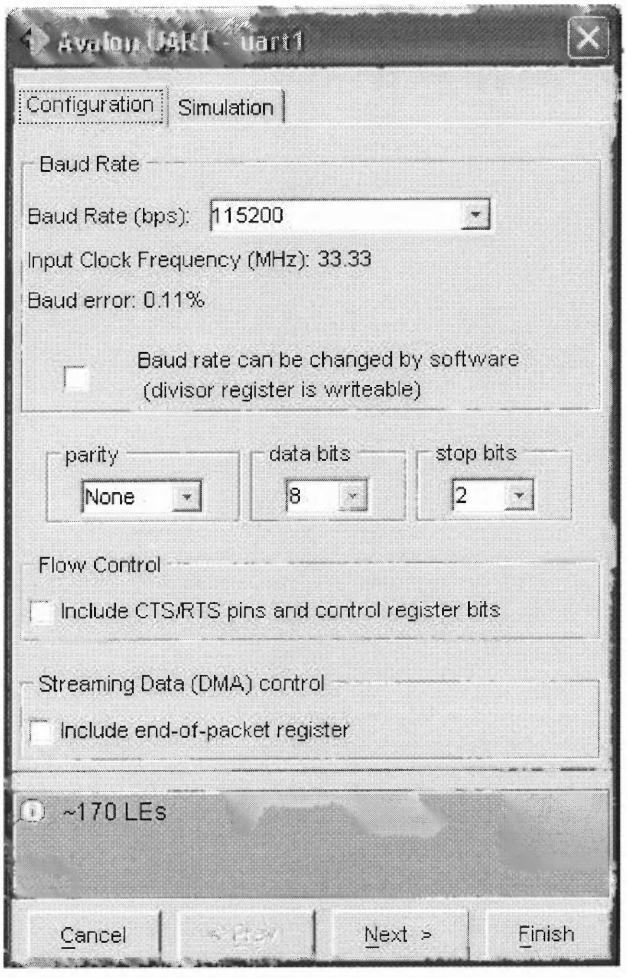

Figure 2.13 UART configuration

5) UART: The UART (Universal Asynchronous Receiver / Transmitter) is used for communication between the NIOS 1 processor and the host computer. The host computer sends code and data to the NIOS processor, which then operates on this data and then sends the results back to the host processor. The configuration options for the UART are shown above in figure 2.13 and are kept at their default values.

6) Clock Timer: A hardware timer is used to count the clock cycles required by the program to run. Figure 2.14 shows a snapshot of the hardware timer. This timer is

connected to NIOS 1 (which is connected to the UART) as this processor is used to download code into the other two processors and also runs the main program.

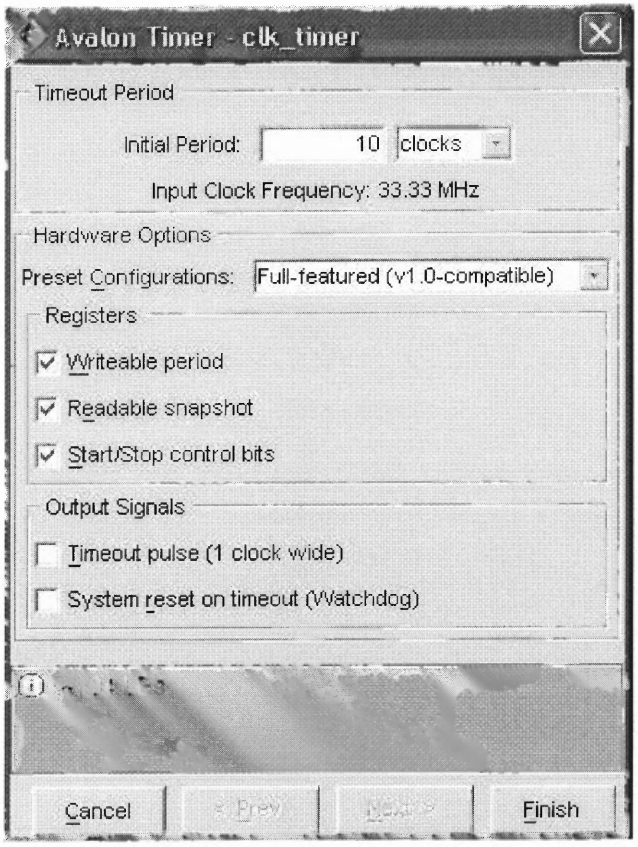

Figure 2.14 Hardware timer configuration

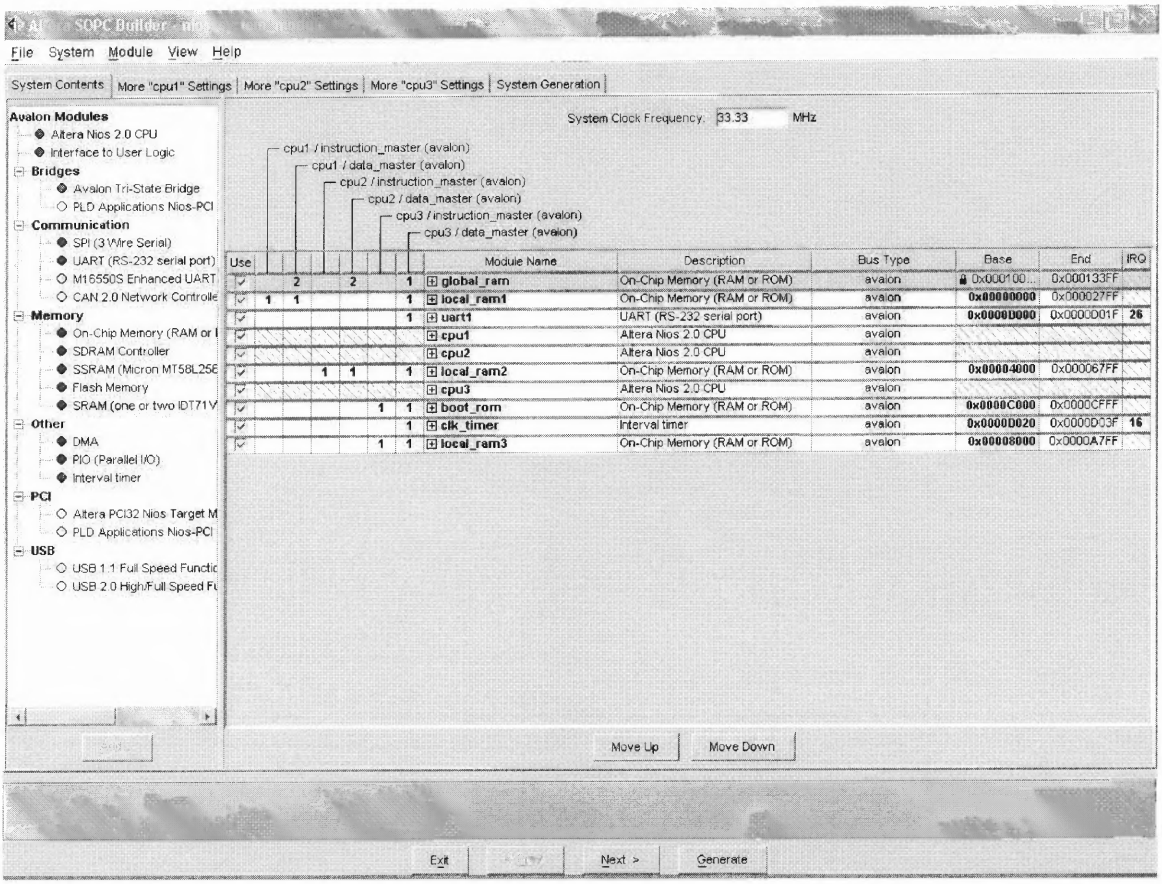

Figure 2.15 Priorities

Figure 2.15 show the priorities each processor has for using the local or global RAM, and other resources connected to the bus.

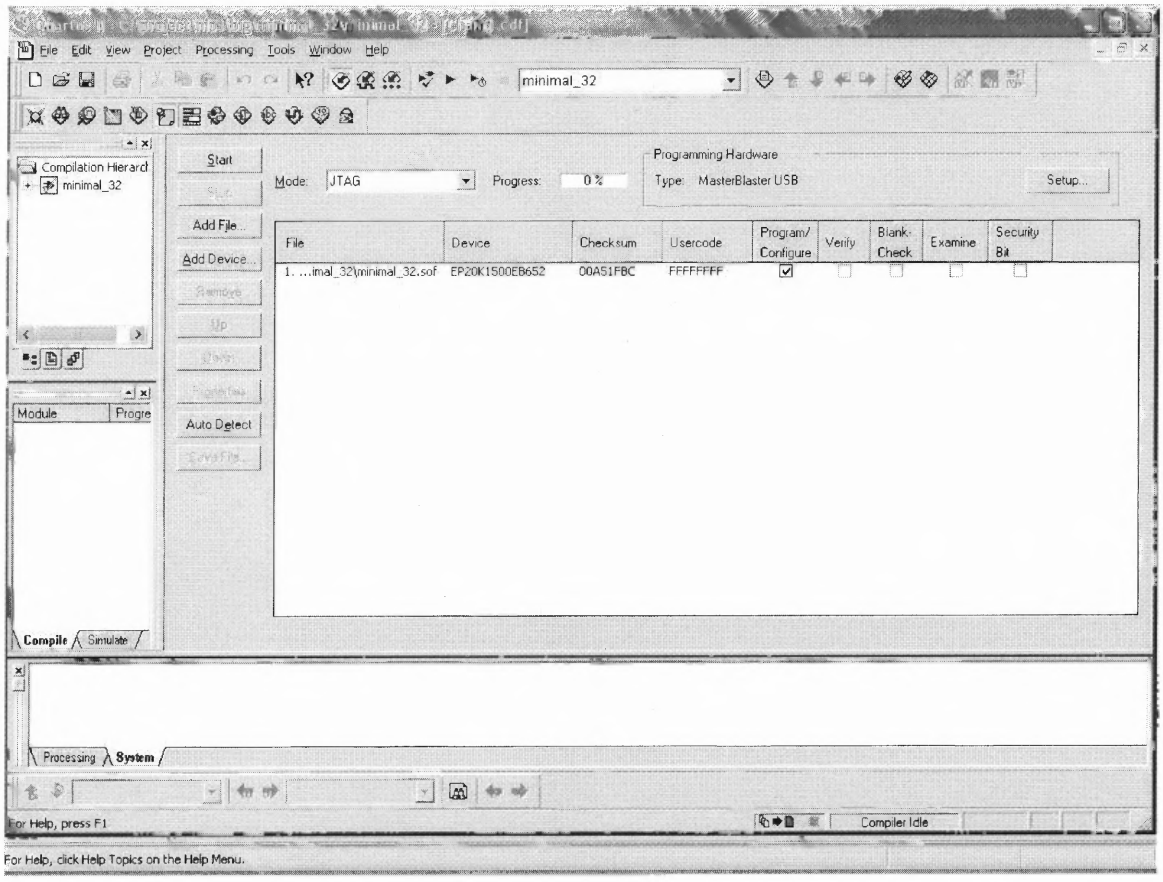

Figure 2.16 Programmer

Figure 2.16 shows the programmer which is used to burn the compiled and synthesized design into the FPGA. The design programmer uses the USB port through the MasterBlaster™ interface provided by Altera® to burn our three-NIOS-processor configuration discussed above into the Altera FPGA. An indication is given once the design has been successfully burned into the chip. Now the code to be run on the chip is downloaded using the NIOS SDK (Software Development Kit) shell, which is a UNIX-like shell. The "nb" command is used to build (compile) the source code of the matrix multiplication program written in the C programming language. If the compilation is successful, then an out file with an extension of ".srec" is generated which can then be

downloaded into a local or the global memory. It can then be executed using the "nr" command.

The "nb" command, when used with the –b option builds the program from a specific memory location. Similarly, the "nr" command, when used with the –x option, just downloads the program into the processor but does not execute it. This feature is particularly useful to us as we use the NIOS 1 processor to download the code first into the local memory of NIOS 2 and NIOS 3 CPUs before we press the reset switch. This causes the NIOS 2 and NIOS 3 CPUs to start running the code from their starting memory location, which now makes them wait in a loop until a particular memory location is set to 1. This happens when NIOS 1 downloads the main controlling program into its own local memory and then starts running the code. When all the processors have finished running their programs, they set another flag. The NIOS 1 processor checks this flag after it has finished running its own code. It then prints out the timing results for the entire process.

CHAPTER 3

PERFORMANCE RESULTS

3.1 Sequential Implementation Results:

Performance results for sequential implementation of the above code are as shown in Table 3.1 for 8 X 8 matrices having various density levels. These matrices were produced by hand. The single NIOS processor used employs a hardware multiplier which requires 16 clock cycles for each multiplication.

Density	Non Zero Elements	CPU clock cycles	Speed improvement over conventional algorithm
10%	6	2429	36.86
20%	13	5774	15.51
30%	19	11409	7.84
40%	26	19688	4.55
50%	32	32081	2.79
60%	38	42083	2.13
70%	45	60314	1.48
80%	51	76683	1.17
90%	58	94121	0.95
100%	64	107699	0.83

Table 3.1 Performance results for sequential implementation

The speed-up shown is for a comparison with the conventional algorithm. It becomes obvious that the performance deteriorates for very dense matrices because of the required comparisons.

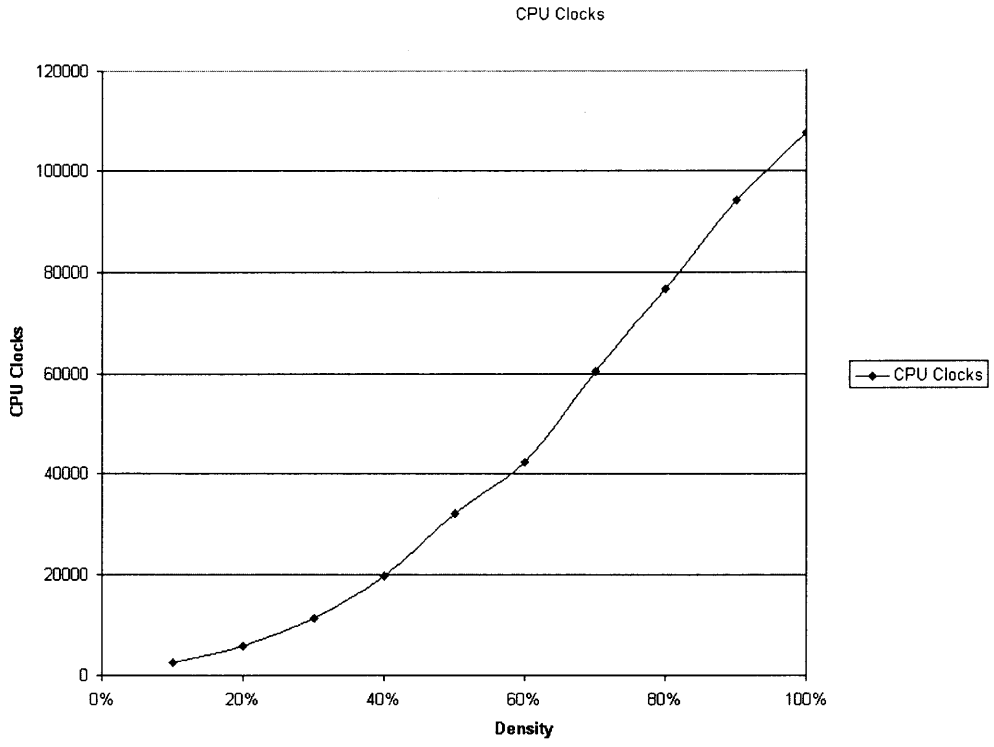

Figure 3.1 Graph of CPU clock cycles versus matrix density (8 X 8 matrices)

Figure 3.1 shows a graph of the CPU clock cycles required versus the matrix density. Note here that in spite of the increase in density the curve remains almost as a straight line with the slope only increasing negligibly at certain points. In the case of the conventional algorithm, the required clock cycles would increase exponentially.

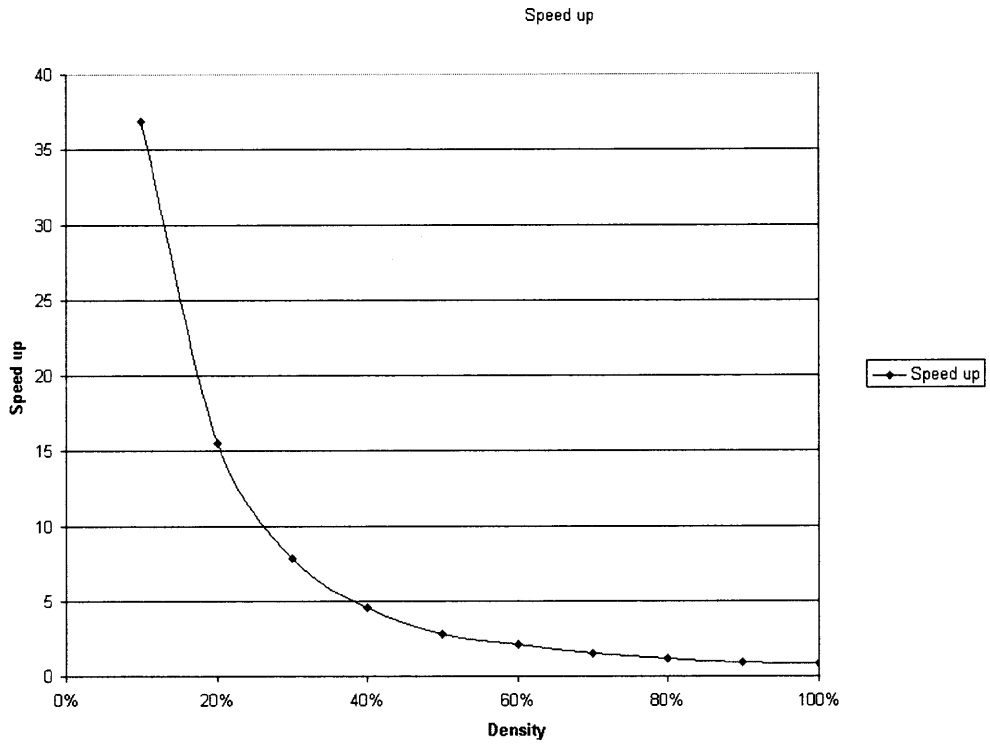

Figure 3.2 Graph of speed up versus matrix density compared to the conventional matrix multiplication algorithm

The graph in Figure 3.2 shows the speed up obtained by using the above algorithm, when compared to the conventional algorithm. Here, we observe that as the density (number of non-zero elements in the matrix) reduces, the speed up increases exponentially. Typically at 30% density the speed up is around 7.8. The speed up equals 1 at around 88% density and reduces for higher densities as the overhead of comparisons becomes significant.

Note that the clock cycles mentioned above include the overhead for post-processing. However, it does not include the overhead for pre-processing which is done on the host PC. It also does not include the overhead for initializations required before the timer starts counting. However, these overheads are not significant. Also the overhead for

pre-processing which involves sorting matrix B in row order is not much. The quick sort algorithm is used to sort Matrix B. Matrix A which is in the matrix market format is already sorted in the column order as required by the our algorithm

3.2 Parallel Implementation Results

Density	Non-Zero Elements	CPU clock cycles (without overhead)	CPU clock Cycles (with overhead)	Speed up A	Speed up B	Speed up C
30%	19	4917	6183	14.48	2.32	1.84
60%	38	19604	25076	3.57	2.14	1.67
100%	64	52794	55728	1.606	2.04	1.93

Speed up A: Speed up (with post processing overhead) compared to conventional algorithm for sequential implementation.

Speed up B: Speed up (without post processing overhead) compared to our algorithm for sequential implementation.

Speed up C: Speed up (with post processing overhead) compared to our algorithm for sequential implementation

Table 3.2 Performance results for parallel implementation

The speed up in Table 3.2 is obtained by comparing the clock cycles in the above table with the clock cycles required by a regular unblocked algorithm used for

multiplying dense matrices on a single NIOS processor. The clock cycles required by the latter dense matrix multiplication algorithm were 89,539.

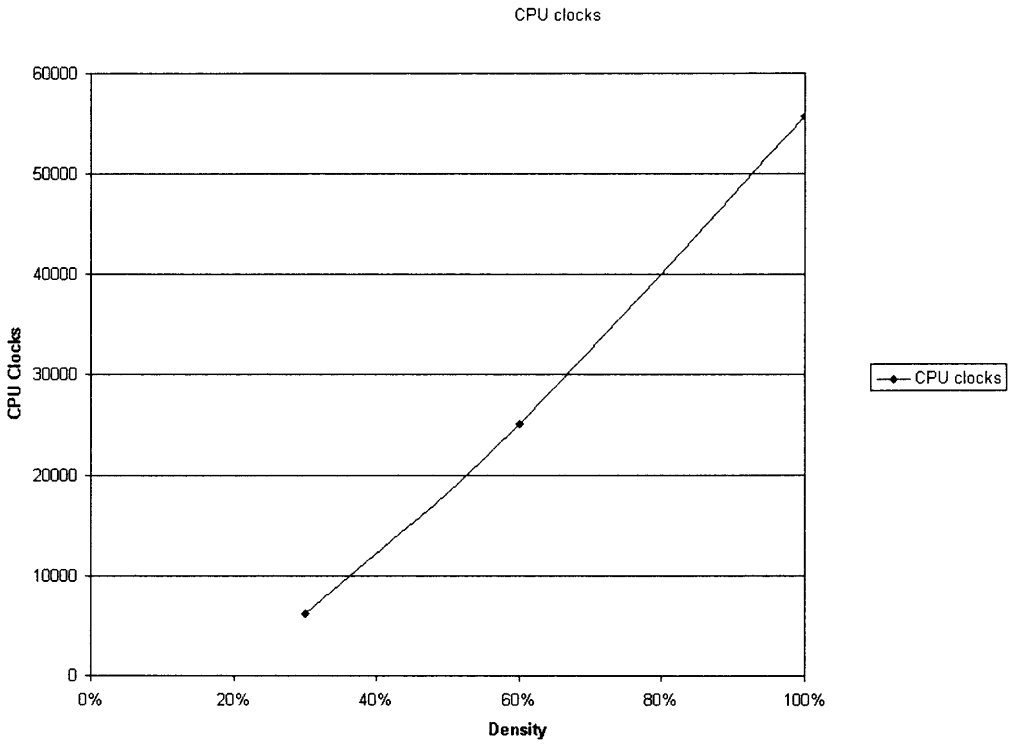

Figure 3.3 Graph of CPU clock cycles versus matrix density for parallel implementation

From Figure 3.3 shown above we can see that just like the graph in figure 3.1, the curve of CPU clocks versus the matrix density is almost as a straight line, except that the slope of the curve in Figure 3.3 is less than that shown in Figure 3.1. This demonstrates further improvement in performance. This is expected for the parallel implementation.

Similarly, Figure 3.2 shows an exponentially decreasing curve similar to that in Figure 3.2; however the new curve is steeper than the previous one, which should be expected.

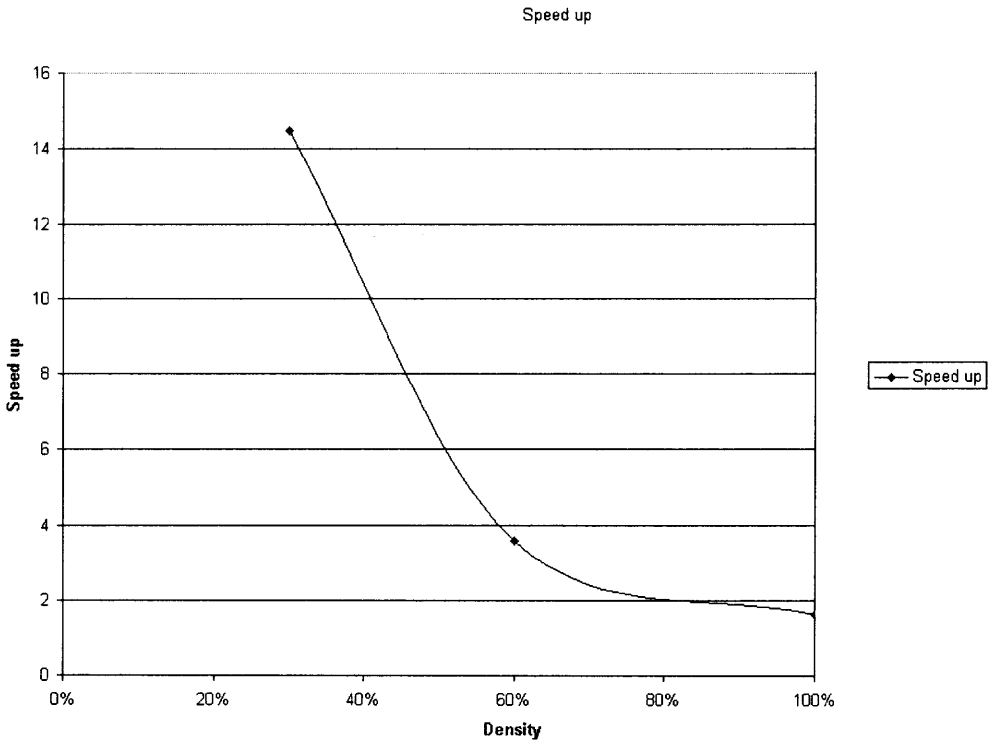

Figure 3.4 Graph of speed up versus matrix density for parallel implementation

Apart from the overheads mentioned for the sequential implementation, the parallel implementation has several more overheads:

1. The matrix is divided into three equal parts during pre-processing. Elements of matrix A whose column indices match with the row indices of matrix B are grouped together and assigned to the same processor. This suffices as a simple load balancing technique. More advanced techniques, however, can be used to do a better job at runtime and save on the time required for preprocessing.
2. During the main execution time of the program, the generated elements of the output matrix are placed in the global RAM. Since each of the three processors is generating outputs, bus arbitration adds to the overhead. Some overhead is saved as each processor has the input elements of the matrices in its own local memory.

Some overhead could be saved here if each processor could generate its output matrix in its local memory and then the main processor could sum up all the results to obtain the final resultant matrix in the global RAM. However, this scheme could not be implemented due to the lack of on-chip memory resources.

CHAPTER 4

CONCLUSIONS

We have presented an algorithm here for matrix multiplication of sparse matrices. Our algorithm is suitable for FPGAs that have limited resources. The proposed algorithm has limited memory space requirements and also results in very good performance. More specifically, the following can be concluded from our results for matrix multiplication:

1) The effectiveness of the algorithm depends on the sparseness of the matrix, i.e. as the number of non-zero elements decreases, the number of calculations also decreases.
2) The algorithm uses extra comparisons that incur an additional overhead that increases with the number of non-zero elements. However, this overhead is negligible, as comparisons typically require a single clock cycle as opposed to floating-point multiplication or addition operations that require several clock cycles depending on the floating-point unit used.
3) Both the input matrices are stored in the matrix market format and, thus, less amount of memory is required. The output matrix, however, is generated as a two-dimensional array. This may consume some space depending on the number of rows and columns in the output matrix. In the case of the parallel architecture, multiple output matrices are generated (one for each processor) which are finally combined to obtain the resultant matrix. Thus, if there are m processors then m+1 output matrices will be generated.

4) In the case of the parallel architecture, the bus arbitration also adds an additional overhead because all three NIOS processors use a shared data memory.

APPENDIX A

Source code for Sequential Implementation

This is a 30% dense matrix, i.e. only 30% of the total elements in the matrix are non-zero valued.

Sequential_30.c

```
#include "nios.h"
#include <math.h>
//*************************************************
// Included all this extra declaration for timer -- ALTERA
//*************************************************
#define TIMER_LOAD_VAL 0xFFFFFFFF
#define take_float_as_int32(x) (*((int *)(&(x))))
#define take_int32_as_float(i) (*((float *)(&(i))))
np_timer *timer = na_timer1;
typedef unsigned long DWORD;
typedef struct
        {
        long interruptCount;   // increment with each interrupt
        np_timer *timer;
        } TimerISRContext;
static TimerISRContext gC = {0,0};
//global variable
int interrupt_count;
```

```
void MyTimerISR(int context)
{
        TimerISRContext *c;
        c = (TimerISRContext *)context;
        c->interruptCount++;
        printf("\n(timer #%d!)",c->interruptCount);
        interrupt_count = c->interruptCount;
        c->timer->np_timerstatus = 0;          // write anything to clear the IRQ
}
long GetTickCount()
{
        volatile long timerVal;
        volatile long timerPeriod;
        timer->np_timersnapl = 0;     // snapshot
        timerVal = (timer->np_timersnapl & 0x0ffff)
                + ((long)timer->np_timersnaph << 16);
        return timerVal;
}
void InitTimer()
{
  long timerPeriod = TIMER_LOAD_VAL;
  //Initialize timer
  timer->np_timerperiodh = timerPeriod >> 16;
```

```
    timer->np_timerperiodl = timerPeriod & 0xffff;
    //Set timer to continuous
    timer->np_timercontrol = timer->np_timercontrol
                                | np_timercontrol_cont_mask;
    //Enable timer interrupt
    gC.timer = na_timer1;
    nr_installuserisr(na_timer1_irq,MyTimerISR,(long)&gC);
    gC.timer->np_timercontrol = gC.timer->np_timercontrol | np_timercontrol_ito_mask;
    printf("\n\nTimer interrupt enabled.\n\n");
     //Start the timer
    timer->np_timercontrol = (timer->np_timercontrol & 3)
                                + np_timercontrol_start_mask;
}
int main(void)
        {
        volatile float a,b;
        volatile float res_a;
        volatile int a_as_int, b_as_int, res_a_as_int;
        volatile DWORD timer_overhead;
        volatile DWORD dwStartTick;
        volatile DWORD lTicksUsed;
        volatile DWORD our_dwStartTick;
        volatile DWORD our_lTicksUsed;
```

```
volatile DWORD aStart=0,aStop=0,aTime=0;
int rowA[]={1,2,3,2,3,2,3,4,3,4,5,3,4,5,5,6,7,8,8};
int colA[]={1,1,1,2,2,3,3,3,4,4,4,5,5,5,6,6,7,7,8};
int valA[]={3,2,5,3,7,5,9,5,1,7,3,6,2,1,9,8,3,3,9};
int rowB[]={1,1,2,2,2,3,3,3,4,4,5,5,6,6,7,7,7,8,8};
int colB[]={1,2,1,2,3,3,4,5,3,4,5,6,5,6,6,7,8,7,8};
int valB[]={7,1,3,2,4,8,6,5,7,4,9,2,5,4,2,1,8,7,9};
int nonZeroA=19, nonZeroB=19, maxRowA=8, maxColA=8, maxRowB=8,
maxColB=8,temp,temp1;
int indexA=0, indexB=0, indexC=0, i=0, j=0, k=0, currentA=0, currentB=0,
currentC=0;
int matC[10][10];
int aBegin,aEnd,bBegin,bEnd;
printf("\n\n\n");
printf("Hello, from Nios!\n");
//Initialize the timer
InitTimer();
//timer_overhead, assume no interrupts
dwStartTick=GetTickCount(); /* record start time*/
lTicksUsed=GetTickCount();  /* record end time */
timer_overhead = dwStartTick - lTicksUsed; //This is used to calculate the timer
overhead
// Matrix C Initialization
```

```
        for(i=1;i<=maxRowA;i++)
                for(j=1;j<=maxColB;j++)
                        matC[i][j]=0;
//      InitTimer();
        aStart=GetTickCount(); /* record start time*/
        while(indexA<nonZeroA && indexB<nonZeroB)
        {
                if(colA[indexA]==rowB[indexB])
                {
                        aBegin=indexA;
                        bBegin=indexB;
                        for(;colA[aBegin] == colA[indexA];indexA++)
                        {}
                        aEnd=indexA;
                        for(;rowB[bBegin] == rowB[indexB];indexB++)
                        {}
                        bEnd=indexB;
                        for(indexA=aBegin;indexA<aEnd;indexA++)
                                for(indexB=bBegin;indexB<bEnd;indexB++)
                                        matC[rowA[indexA]][colB[indexB]] +=
                valA[indexA] * valB[indexB];
                }
                if(colA[indexA] < rowB[indexB])
```

```
                indexA++;
            if(colA[indexA] > rowB[indexB])
                indexB++;
        }
aStop=GetTickCount();  /* record end time */
aTime=(aStart - aStop) - timer_overhead ;
printf("Start= %ld, Stop= %ld, overhead= %ld\n",
dwStartTick,lTicksUsed,timer_overhead);
printf("Time to calculate was %ld and start count :%ld, stop count: %ld\n",
aTime,aStart,aStop);
for(i=1;i<=maxRowA;i++)
{
        for(j=1;j<=maxRowB;j++)
            printf("%d ",matC[i][j]);
        printf("\n");
}
        printf("Good bye.\n\004");     // control-D tells terminal program, We are done.
} // The end.
```

APPENDIX B

Source Code for Parallel Implementation

Program on Processor-1

Program for an 8 X 8, 100% dense matrix. All elements are non-zero.

mm1.c

```
#include "nios1.h"

int main(void)

{

int rowA[] = {1,2,3,4,5,6,7,8,1,2,3,4,5,6,7,8,1,2,3,4,5,6,7,8,1,2,3,4,5,6,7,8
,1,2,3,4,5,6,7,8,1,2,3,4,5,6,7,8,1,2,3,4,5,6,7,8};

int colA[] = {1,1,1,1,1,1,1,1,2,2,2,2,2,2,2,2,3,3,3,3,3,3,3,3,4,4,4,4,4,4,4,4,5,5,5,5,5,5,5,5,
6,6,6,6,6,6,6,6,7,7,7,7,7,7,7,7,8,8,8,8,8,8,8,8};

int valA[] = {4,8,3,4,4,3,4,3,8,5,3,6,3,2,4,9,6,7,2,4,2,3,1,6,1,2,8,5,7,9,5,1,3,2,4,6,7,9,2,1,9
,7,4,8,4,9,5,6,3,6,2,4,5,2,4,8,9,6,3,8,4,7,3,7};

int colB[] = {1,2,3,4,5,6,7,8,1,2,3,4,5,6,7,8,1,2,3,4,5,6,7,8,1,2,3,4,5,6,7,8,1,2,3,4,5,6,7,8,1
,2,3,4,5,6,7,8,1,2,3,4,5,6,7,8,1,2,3,4,5,6,7,8};

int rowB[] = {1,1,1,1,1,1,1,1,2,2,2,2,2,2,2,2,3,3,3,3,3,3,3,3,4,4,4,4,4,4,4,4,5,5,5,5,5,5,5,5
,6,6,6,6,6,6,6,6,7,7,7,7,7,7,7,7,8,8,8,8,8,8,8,8};

int valB[] = {5,9,4,6,3,2,8,3,7,3,3,4,3,9,2,7,1,9,5,8,2,3,7,4,1,2,7,4,5,6,9,5,2,2,6,3,4,2,8,6,
3,2,6,5,7,1,2,8,7,6,5,2,3,4,1,8,6,9,5,3,6,7,8,9};

int nonZeroA=16, nonZeroB=16, maxRowA=8, maxColA=8, maxRowB=8,
maxColB=8,temp,temp1;

int indexA=0, indexB=0, indexC=0, i=0, j=0, k=0, currentA=0, currentB=0, currentC=0;
```

```
int *matC_1=(int *)0x10200;
int aBegin,aEnd,bBegin,bEnd;
int *flag1_start=(int *)0x10000;
int *flag1_end=(int *)0x10010;
*flag1_start=0;
for(i=1;i<=maxRowA;i++)
        for(j=1;j<=maxColB;j++)
                matC_1[(i*(maxColB+1))+j]=0;
while(indexA<nonZeroA && indexB<nonZeroB)
{
        if(colA[indexA]==rowB[indexB])
        {
                aBegin=indexA;
                bBegin=indexB;
                for(;colA[aBegin] == colA[indexA];indexA++)
                {}
                aEnd=indexA;
                for(;rowB[bBegin] == rowB[indexB];indexB++)
                {}
                bEnd=indexB;
                for(indexA=aBegin;indexA<aEnd;indexA++)
                        for(indexB=bBegin;indexB<bEnd;indexB++)
```

```
                matC_1[((rowA[indexA])*(maxColB+1))+(colB[indexB])] +=
valA[indexA] * valB[indexB];
        }
        if(colA[indexA] < rowB[indexB])
                indexA++;
        if(colA[indexA] > rowB[indexB])
                indexB++;
}
*flag1_end=1;
} // The end.
```

Program on Processor-2

mm2.c

```
#include "nios2.h"

int main(void)

{

int rowA[] = {1,2,3,4,5,6,7,8,1,2,3,4,5,6,7,8,1,2,3,4,5,6,7,8,1,2,3,4,5,6,7,8,
1,2,3,4,5,6,7,8,1,2,3,4,5,6,7,8,1,2,3,4,5,6,7,8};

int colA[] = {1,1,1,1,1,1,1,1,2,2,2,2,2,2,2,2,3,3,3,3,3,3,3,3,4,4,4,4,4,4,4,4,5,5,5,5,5,5,5,5,
6,6,6,6,6,6,6,6,7,7,7,7,7,7,7,7,8,8,8,8,8,8,8,8};

int valA[] = {4,8,3,4,4,3,4,3,8,5,3,6,3,2,4,9,6,7,2,4,2,3,1,6,1,2,8,5,7,9,5,1,3,2,4,6,7,9,2,1,
9,7,4,8,4,9,5,6,3,6,2,4,5,2,4,8,9,6,3,8,4,7,3,7};

int colB[] = {1,2,3,4,5,6,7,8,1,2,3,4,5,6,7,8,1,2,3,4,5,6,7,8,1,2,3,4,5,6,7,8,1,2,3,4,5,6,
7,8,1,2,3,4,5,6,7,8,1,2,3,4,5,6,7,8,1,2,3,4,5,6,7,8};

int rowB[] = {1,1,1,1,1,1,1,1,2,2,2,2,2,2,2,2,3,3,3,3,3,3,3,3,4,4,4,4,4,4,4,4,5,5,5,5,5,5,5,5,
6,6,6,6,6,6,6,6,7,7,7,7,7,7,7,7,8,8,8,8,8,8,8,8};

int valB[] = {5,9,4,6,3,2,8,3,7,3,3,4,3,9,2,7,1,9,5,8,2,3,7,4,1,2,7,4,5,6,9,5,2,2,6,3,4,2,8,6
,3,2,6,5,7,1,2,8,7,6,5,2,3,4,1,8,6,9,5,3,6,7,8,9};

int nonZeroA=40, nonZeroB=40, maxRowA=8, maxColA=8, maxRowB=8,
maxColB=8,temp,temp1;

int indexA=16, indexB=16, indexC=0, i=0, j=0, k=0, currentA=0, currentB=0,
currentC=0;

int *matC_2=(int *)0x10400;

int aBegin,aEnd,bBegin,bEnd;
```

```
int *flag1_start=(int *)0x10000;
int *flag2_end=(int *)0x10030;
while(*flag2_start!=1){}
for(i=1;i<=maxRowA;i++)
        for(j=1;j<=maxColB;j++)
                matC_2[(i*(maxColB+1))+j]=0;
while(indexA<nonZeroA && indexB<nonZeroB)
{
        if(colA[indexA]==rowB[indexB])
        {
                aBegin=indexA;
                bBegin=indexB;
                for(;colA[aBegin] == colA[indexA];indexA++)
                {}
                aEnd=indexA;
                for(;rowB[bBegin] == rowB[indexB];indexB++)
                {}
                bEnd=indexB;
                for(indexA=aBegin;indexA<aEnd;indexA++)
                        for(indexB=bBegin;indexB<bEnd;indexB++)
                        matC_2[((rowA[indexA])*(maxColB+1))+(colB[indexB])] +=
valA[indexA] * valB[indexB];
        }
```

```
        if(colA[indexA] < rowB[indexB])
                indexA++;
        if(colA[indexA] > rowB[indexB])
                indexB++;
}
*flag2_end=1;
while(*flag2_end!=1){}
} // The end.
```

Program on Processor-3

mm3.c – Main program

```
#include "nios.h"
//*************************************************
// Included all this extra declaration for timer -- ALTERA
//*************************************************
#define TIMER_LOAD_VAL 0xFFFFFFFF
#define take_float_as_int32(x) (*((int *)(&(x))))
#define take_int32_as_float(i) (*((float *)(&(i))))
np_timer *timer = na_clk_timer;
typedef unsigned long DWORD;
typedef struct
{
        long interruptCount;   // increment with each interrupt
        np_timer *timer;
} TimerISRContext;
static TimerISRContext gC = {0,0};
//global variable
int interrupt_count;
void MyTimerISR(int context)
{
        TimerISRContext *c;
        c = (TimerISRContext *)context;
```

```
        c->interruptCount++;
        printf("\n(timer #%d!)",c->interruptCount);
        interrupt_count = c->interruptCount;
        c->timer->np_timerstatus = 0;          // write anything to clear the IRQ
}
long GetTickCount()
{
        volatile long timerVal;
        volatile long timerPeriod;
        timer->np_timersnapl = 0;     // snapshot
        timerVal = (timer->np_timersnapl & 0x0ffff)
                + ((long)timer->np_timersnaph << 16);
        return timerVal;
}
void InitTimer()
{
  long timerPeriod = TIMER_LOAD_VAL;
  //Initialize timer
  timer->np_timerperiodh = timerPeriod >> 16;
  timer->np_timerperiodl = timerPeriod & 0xffff;
  //Set timer to continuous
  timer->np_timercontrol = timer->np_timercontrol
                                | np_timercontrol_cont_mask;
```

```
  //Enable timer interrupt
  gC.timer = na_clk_timer;
  nr_installuserisr(na_clk_timer_irq,MyTimerISR,(long)&gC);
  gC.timer->np_timercontrol = gC.timer->np_timercontrol | np_timercontrol_ito_mask;
  printf("\n\nTimer interrupt enabled.\n\n");
   //Start the timer
  timer->np_timercontrol = (timer->np_timercontrol & 3)
                                + np_timercontrol_start_mask;
}
int main(void)
        {
        volatile float a,b;
        volatile float res_a;
        volatile int a_as_int, b_as_int, res_a_as_int;
        volatile DWORD timer_overhead;
        volatile DWORD dwStartTick;
        volatile DWORD lTicksUsed;
        volatile DWORD our_dwStartTick;
        volatile DWORD our_lTicksUsed;
        volatile DWORD aStart=0,aStop=0,aTime=0;
        int rowA[] = {1,2,3,4,5,6,7,8,1,2,3,4,5,6,7,8,1,2,3,4,5,6,7,8,1,
        2,3,4,5,6,7,8,1,2,3,4,5,6,7,8,1,2,3,4,5,6,7,8,1,2,3,4,5,6,7,8};
        int colA[] = {1,1,1,1,1,1,1,1,2,2,2,2,2,2,2,2,3,3,3,3,3,3,3,3,4,4,4,4,4,4,4,4,
```

```
5,5,5,5,5,5,5,5,6,6,6,6,6,6,6,6,7,7,7,7,7,7,7,7,8,8,8,8,8,8,8,8};
int valA[] = {4,8,3,4,4,3,4,3,8,5,3,6,3,2,4,9,6,7,2,4,2,3,1,6,1,2,8,5,7,9,5,1,3,2,4,6
,7,9,2,1,9,7,4,8,4,9,5,6,3,6,2,4,5,2,4,8,9,6,3,8,4,7,3,7};
int colB[] = {1,2,3,4,5,6,7,8,1,2,3,4,5,6,7,8,1,2,3,4,5,6,7,8,1,2,3,4,5,6,7,8,1,2
,3,4,5,6,7,8,1,2,3,4,5,6,7,8,1,2,3,4,5,6,7,8,1,2,3,4,5,6,7,8};
int rowB[] = {1,1,1,1,1,1,1,1,2,2,2,2,2,2,2,2,3,3,3,3,3,3,3,3,4,4,4,4,4,4,4,4,5,5
,5,5,5,5,5,5,6,6,6,6,6,6,6,6,7,7,7,7,7,7,7,7,8,8,8,8,8,8,8,8};
int valB[] = {5,9,4,6,3,2,8,3,7,3,3,4,3,9,2,7,1,9,5,8,2,3,7,4,1,2,7,4,5,6,9,5,2,2,6,3,
4,2,8,6,3,2,6,5,7,1,2,8,7,6,5,2,3,4,1,8,6,9,5,3,6,7,8,9};
int nonZeroA=64, nonZeroB=64, maxRowA=8, maxColA=8, maxRowB=8,
maxColB=8,temp,temp1;
int indexA=40, indexB=40, indexC=0, i=0, j=0, k=0, currentA=0, currentB=0,
currentC=0;
int *matC_1=(int *)0x10200;
int *matC_2=(int *)0x10400;
int *matC_3=(int *)0x10600;
int *matC=(int *)0x10800;
int aBegin,aEnd,bBegin,bEnd;
int *flag1_start=(int *)0x10000;
int *flag1_end=(int *)0x10010;
int *flag2_start=(int *)0x10020;
int *flag2_end=(int *)0x10030;
printf("\n\n\n");
```

```
printf("Hello, from Nios!\n");
//Initialize the timer
InitTimer();
//timer_overhead, assume no interrupts
dwStartTick=GetTickCount(); /* record start time*/
lTicksUsed=GetTickCount();  /* record end time */
timer_overhead = dwStartTick - lTicksUsed;
for(i=1;i<=maxRowA;i++)
        for(j=1;j<=maxColB;j++)
                matC[(i*(maxColB+1))+j]=0;
for(i=1;i<=maxRowA;i++)
        for(j=1;j<=maxColB;j++)
                matC_3[(i*(maxColB+1))+j]=0;
//      InitTimer();
aStart=GetTickCount(); /* record start time*/
        *flag1_start=1;
        *flag2_start=1;
while(indexA<nonZeroA && indexB<nonZeroB)
{
        if(colA[indexA]==rowB[indexB])
        {
                aBegin=indexA;
                bBegin=indexB;
```

```
                for(;colA[aBegin] == colA[indexA];indexA++)
                {}
                aEnd=indexA;
                for(;rowB[bBegin] == rowB[indexB];indexB++)
                {}
                bEnd=indexB;
                for(indexA=aBegin;indexA<aEnd;indexA++)
                        for(indexB=bBegin;indexB<bEnd;indexB++)
matC_3[((rowA[indexA])*(maxColB+1))+(colB[indexB])] += valA[indexA] *
valB[indexB];
        }
        if(colA[indexA] < rowB[indexB])
                indexA++;
        if(colA[indexA] > rowB[indexB])
                indexB++;
}
for(i=1;i<=maxRowA;i++)
        for(j=1;j<=maxColB;j++)
                matC[(i*(maxColB+1))+j] = matC_1[(i*(maxColB+1))+j] +
matC_2[(i*(maxColB+1))+j] + matC_3[(i*(maxColB+1))+j];
aStop=GetTickCount();  /* record end time */
aTime=(aStart - aStop) - timer_overhead ;
```

```
        printf("Time to calculate was %ld and start count :%ld, stop count: %ld\n",
        aTime,aStart,aStop);
for(i=1;i<=maxRowA;i++)
{
        for(j=1;j<=maxColB;j++)
                printf("%d ",matC[(i*(maxColB+1))+j]);
        printf("\n");
}
        printf("Good bye.\n\004");     // control-D tells terminal program, We are done.
                                        } // The end.
```

REFERENCES

1. A. Amira and F. Bensaali, *An FPGA based parameterisable system for matrix product implementation*, IEEE Workshop on Signal Processing Systems (SIPS'02), San Diego, California, USA, October 16-18, 2002.

2. Anonymous, *Numerical Linear Algebra*, Computational Science Education Project, 1995.

3. http://gams.nist.gov/MatrixMarket/ Matrix market sample files.

4. http://phase.hpcc.jp/mirrors/MatrixMarket/formats.html
Matrix market file format explanations.

5. Anonymous, *NIOS Hardware/Software programmers reference manual* – Assembly language instruction set, 2001.

6. V. Strassen, *Gaussian Elimination is not optimal*, Numer. Math., 13:354-356, 1969.

7. Hossam ElGindy and Yen-Liang Shue, *On Sparse Matrix-vector Multiplication with FPGA-based System,* Proceedings of the 10th Annual IEEE Symposium on Field-Programmable Custom Computing Machines. (FCCM'02.)

8. C.P. Kruskal, L. Rudolph and M. Snir, *Techniques for Parallel-Implementation of Sparse Matrices,* Theoretical Computer Science 64 (1989)

9. http://www.altera.com Altera website