The LINPACK Benchmark on a Multi-Core Multi-FPGA System

by

Emanuel Caldeira Ramalho

A thesis submitted in conformity with the requirements
for the degree of Masters of Applied Science
Graduate Department of Electrical and Computer Engineering
University of Toronto

Abstract

The LINPACK Benchmark on a Multi-Core Multi-FPGA System

Emanuel Caldeira Ramalho

Masters of Applied Science

Graduate Department of Electrical and Computer Engineering

University of Toronto

2008

The LINPACK Benchmark is used to rank the most powerful computers in the world. This thesis is an implementation of the benchmark on a multi-FPGA system to see how the performance compares to the processor-based implementation. TMD-MPI is the MPI implementation used to parallelize the software portion of the algorithm while the TMD-MPE provides the same functionality for the hardware engines. Results show that, when using small sets of data, one FPGA can provide a speedup of 1.94 over a high-end processor running the LINPACK Benchmark with Level 1 BLAS. However, there is still opportunity to do better, especially when scaling to larger systems.

Dedication

I would like to dedicate this thesis to all of the people that have made it possible. To my parents for all of their love, support and financial effort throughout the past two years and without whom this would have never been possible. To my girlfriend Jas, whom I love from the bottom of my heart, that has supported me in the good and the not so good moments and has made this last year seem like a dream. To all my family, but especially to my grandma, “Xica” that has always encouraged me to strive forward and that passed away during my stay here in Canada. Grandma “Xica”, you will always be in a special place in my heart.

I would also like to thank my supervisor Professor Paul Chow for sharing his knowledge with me, for his patience, guidance and feedback that helped me achieve my objectives. Special thanks to all my research group for all of their help, comments and availability whenever I needed. Particularly Manuel, Arun, Chris, Daniel, Alex, Danny, Dan, Keith, David, Andrew, Ali, Richard, Tom and Mike. Last but not least I would like to thank Professor João Cardoso that made this opportunity possible through his encouragement of taking graduate studies abroad at the University of Toronto and by teaching me the fundamentals of the field of computer engineering.

Acknowledgements

I would like to acknowledge all the institutions that supported and provided the necessary hardware and infrastructure that allowed me to complete this research: the University of Toronto, CMC/SOCRN, NSERC and Xilinx.

Contents

1 Introduction **1**

1.1 Motivation . . . 1

1.2 Research Contributions . . . 3

1.3 Thesis Organisation . . . 3

2 Background **4**

2.1 LINPACK . . . 4

2.1.1 The Benchmark . . . 5

2.1.2 HPL: The Parallel Benchmark . . . 8

2.2 Parallel Computing . . . 10

2.2.1 MPI . . . 11

2.2.2 TMD-MPI and MPE . . . 13

2.3 Related Work . . . 14

3 Hardware Testbed **16**

3.1 Hardware and Software Overview . . . 16

3.2 Processing Units and the Network . . . 18

3.3 Design Flow . . . 20

4 The LINPACK Hardware Engine **23**

4.1 The LINPACK Implementations . . . 23

4.2 Analysis and Parallelization 25
4.3 The Hardware Engine 27
4.3.1 Main FSM 28
4.3.2 BLAS1 Engine 29
4.3.3 MPE Header FSM 31
4.4 FPGA Utilization 32

5 Performance Analysis of Results **35**
5.1 Methods of Analysis 35
5.2 Results 37
5.2.1 One LINPACK Engine vs. One Processor 38
5.2.2 One FPGA vs. One Processor 39
5.2.3 Scalability Test 40
5.3 Scaled Performance Analysis 44
5.4 Theoretical Peak 47
5.5 Scaling to Larger Systems 48

6 Conclusions **50**

7 Future Work **52**

Bibliography **54**

List of Tables

2.1 Level 1 BLAS vs. Level 2 BLAS vs. Level 3 BLAS 8

4.1 More significant cores in terms of device utilization 34

5.1 Average percentage of the Computation, Send and Receive events 42

List of Figures

2.1 Time comparison between DGEFA and DGESL 7

2.2 Time comparison between IDAMAX, DSCAL and DAXPY 8

2.3 Example of the two-dimensional block distribution with $P = 2$ and $Q = 4$ . . . 10

2.4 TMD-MPI Layers 14

3.1 Simplified schematic of a BEE2 board 17

3.2 Different TMD-MPI configurations depending on the TMD-MPE use 20

3.3 Design Flow 21

4.1 Main functions of LINPACK Benchmark 25

4.2 Code of the DGEFA function 25

4.3 Parallelization of the code 27

4.4 LINPACK Engine High-Level Schematic 28

4.5 First tag received by the engine 29

4.6 Simplified data path of the BLAS1 Engine 30

4.7 TMD-MPE protocol 31

4.8 Synthesis report of the LINPACK Engine 33

5.1 Simplified schematic of the system used to test the LINPACK engine with the TMD-Profiler 38

5.2 Speedup of the LINPACK engine with fixed problem size up to 8 processing units 40

5.3 Efficiency of the LINPACK engine with fixed problem size up to 8 processing units 41
5.4 Jumpshot image of the broadcast performed by rank 1 43
5.5 Speedup of the LINPACK engine with scaled problem size up to 8 processing units 45
5.6 Efficiency of the LINPACK engine with scaled problem size up to 8 processing units 45
5.7 Latency effects with different column distribution 46

Glossary

ASIC. Application-Specific Integrated Circuit.

BEE2. Berkeley Emulation Engine 2.

BRAM. Block Random Access Memory.

BLAS. Basic Linear Algebra Subprograms.

DDR. Double Data Rate (RAM).

DVI. Digital Video Interface.

FIFO. First In First Out.

FLOPS. Floating-Point Operations per Second.

FPGA. Field Programmable Gate Array.

FSL. Fast Simplex Link.

FSM. Finite State-Machine.

GUI. Graphical User Interface.

HDL. Hardware Description Language.

HPL. Highly-Parallel LINPACK.

IC. InterChip.

ILP. Instruction-Level Parallelism.

LAM MPI. Local Area Multicomputer MPI.

LINPACK. LINear (Algebra) PACKage.

LUT. Look-Up Table.

MPI. Message-Passing Interface.

MPICH. MPI CHameleon.

NetIf. Network Interface.

RISC. Reduced Instruction Set Computer.

RTC. Real-Time Clock.

TMD. Originally meant *Toronto Molecular Dynamics* machine, but this definition was rescinded as the platform is not limited to Molecular Dynamics. The name was kept in homage to earlier TM-series projects at the University of Toronto.

TMD-MPE. TMD-Message-Passing Engine.

USB. Universal Serial Bus.

VSIPL. Vector Signal Image Processing Library.

Chapter 1

Introduction

As FPGAs have been gaining popularity, so has their range of applications. Several studies have made use of FPGAs to perform heavy-duty computational tasks as a way of improving the performance of an algorithm. However, its use is not limited to co-processing, as some researchers have already started to build larger systems of FPGAs in a supercomputer-like approach (RC200 [24], BEE2 [3] and the Maxwell [1]). By applying a known algorithm to one of these systems it is possible to better understand the potential of such machines.

Section 1.1 explains the motivation behind this thesis. Section 1.2 summarizes the main contributions of this work. Section 1.3 presents an overview of what can be found in the remainder of this thesis.

1.1 Motivation

Until recently, one the main factors for improvements in computer performance was the clock frequency increase of the main processor. However, due to poor wire scaling and overheating issues, which arise from the shrinkage in the size of the transistors, increases in the chip's clock rate have become compromised. Therefore, as a solution to this obstruction, manufacturers started to place multiple cores on a single die instead of increasing the clock speed. The idea is to increase the computation power by dividing the workload across the available processors.

This is an example of parallel computing.

However, this idea is not new, as parallelism has been used in supercomputers for over 20 years now. Supercomputers are high-end machines, usually consisting of thousands of cores used to calculate computationally-intensive tasks, such as weather forecasting (predicting the weather for short- and long-term effects), molecular modeling (studying structure and properties of molecules), and physical simulations (such as simulating airplanes in wind tunnels). Examples of these machines include the IBM Roadrunner [15], the IBM BlueGene/L [14], the NEC Earth Simulator [19], and the CrayXT4 [5].

Another alternative to achieve performance improvements is to create special-purpose hardware engines to accelerate a given computation task. This can be done with an Application-Specific Integrated Circuit, or ASIC, which is an integrated circuit specifically customized for a particular use. However, the costs of an ASIC are forbiddingly high and, due to the impossibility of further modifications, its scope of usability is very limited. In contrast, FPGAs can provide the core specialization without the loss of configurability, since its hardware can be reprogrammed as needed.

Recent research has shown that FPGAs, as supercomputers, have the potential to provide significant performance improvements by applying the principles of parallelism. A High-End Reconfigurable Computer (HERC) is a supercomputer-like platform based on reconfigurable devices as the processing elements that can be configured to solve a given computation task.

The LINPACK Benchmark [7] is the official benchmark used to rank the supercomputers in the Top500 list [18]. This is a software implementation and further performance gains may be achieved with a hardware version of the algorithm. By using a multi-FPGA system, LINPACK may be converted to hardware and still get the parallelism performance present when run on a supercomputer. The goal of this work is to implement LINPACK on real hardware and determine how well a multi-FPGA system can perform compared to the standard processor-based approach.

1.2 Research Contributions

In this thesis, a BEE2-platform is used to calculate the most known benchmark in the supercomputer world, the LINPACK Benchmark. For the first time, LINPACK is completely coded using a Hardware Description Language (HDL) and run on a multi-FPGA system. The design flow, previously presented in [21], and the message-passing model (TMD-MPI), an MPI based model created by Manuel Saldaña [23], are used to implement the system. Performance results show that one FPGA can provide a maximum speedup of 1.94x over the processor. However, when compared to a better algorithm, the FPGA model used in this thesis is not fast enough. These results show that there is much room for improvement and the main points of future improvement are presented.

1.3 Thesis Organisation

The remainder of this thesis is organised as follows. Chapter 2 presents a short description of the ideas and concepts needed to better understand this thesis, followed by related work. First, a description of the LINPACK Benchmark and its parallel variation is explained. Then, a compact introduction to parallel computing and MPI, the parallel paradigm used in this thesis, is described. In Chapter 3, the hardware and tools used in this thesis are described. The processing elements, the network components and the design flow used are also part of this chapter. Chapter 4 explains the approach chosen for implementing LINPACK and also the details of the hardware used to implement the parallel engine. Results and the methodology are presented in Chapter 5. Finally, Chapter 6 presents the conclusions and Chapter 7 describes what else can be done to improve the performance of the LINPACK Engine created in this thesis.

Chapter 2

Background

This chapter presents a succinct explanation of the background information needed to better understand this thesis. The following sections give a brief description of the target algorithm and the parallel techniques used to approach it. Section 2.1 gives a brief overview of the LINPACK software package and explains the mathematical knowledge behind the LINPACK Benchmark and some of the techniques used to improve its performance. Section 2.2 describes the parallel paradigm used in this work as well as some details on the software and hardware used to implement the parallelism. Finally, Section 2.3 refers to previous work and the methods used by others in the development of this benchmark and the functions used on FPGAs.

2.1 LINPACK

The LINPACK software package consists of a set of Fortran subroutines that are used to analyze and solve systems of linear equations. The package solves the linear systems by factorizing or decomposing a given matrix into simpler matrices that are posteriorly used to, more easily, find the solution to the system. It supports several matrix and data types along with a vast list of options.

To perform these calculations LINPACK makes use of another collection of subroutines called the Level 1 BLAS [16]. At this level of BLAS, only vector-vector operations are permitted. These subroutines work at a much lower level than LINPACK, which means that these are the functions that actually execute the floating point operations.

2.1.1 The Benchmark

The LINPACK Benchmark was originally created as a way of aiding the users of the LINPACK software package. This was done by measuring the time the algorithm takes to solve a system of linear equations for a matrix problem size of 100. Results for the most widely used computers (23 different machines) could be found in the LINPACK User's Guide (1979) [8]. This would allow users to estimate the time required to solve their matrix problem size.

Today, the collection of performance results spans over more than 1300 computers. However, the changes are not limited to the number of computers in the performance list, the Benchmark has, also, suffered modifications. Due to memory limitations of computers in 1979, 100 by 100 was chosen as the problem size featured in the LINPACK Benchmark. However, that is not a "large enough" problem size anymore and the newer version uses 1000 by 1000 as the new default size, with the possibility of increasing this value to get the highest performance possible. In addition to changes to the problem size, the new LINPACK Benchmark was optimized to take advantage of different architectures. Most of these optimizations were performed on the parallel version of the Benchmark, which was created to run on multiple processors and thereby taking advantage of the parallelism present in this algorithm. This scalable parallel program is called (HPL) [7] and will be explained in Section 2.1.2.

As already mentioned, the LINPACK Benchmark measures the performance of a computer by calculating the execution time to solve a system of linear equations. To achieve this, the algorithm first randomly generates a general dense matrix input, as shown Equation (2.1). The random method used to generate the matrix is not the most complex one, however this does not represent an issue since this part of the algorithm is not included in the time measurements.

$$Ax = b \tag{2.1}$$

To solve the problem, the algorithm calls two routines: DGEFA and DGESL, both of them double precision (although this can be changed to single precision), i.e., they operate over 64-bit floating-point numbers. The first routine, DGEFA, performs an (LU) decomposition with partial pivoting of matrix A. As shown in Equation (2.2), A is decomposed into a lower triangular matrix (L) and an upper triangular matrix (U). P is a permutation matrix used to exchange the rows of matrix A, this is what is called partial pivoting and is used to ensure the numerical stability of the algorithm.

$$PA = LU \tag{2.2}$$

After decomposing matrix A, DGESL uses that decomposition and vector b to solve the system of linear equations. This routine is divided in two steps: in the first step L and b are used to solve y and in the second step U and y are used to find x, Equation (2.3).

$$LUx = Pb \Rightarrow Ly = Pb \Rightarrow Ux = y \tag{2.3}$$

Finally, as a verification method, the algorithm performs a residual calculation. For that, the initial matrix A and vector b are regenerated and the following residuals computed:

$$r = ||Ax - b|| \tag{2.4}$$

$$r_{norm} = \frac{||Ax - b||}{n \cdot ||A|| \cdot ||x|| \cdot \epsilon} \tag{2.5}$$

where ϵ is the relative machine precision, r the residual and r_{norm} the normalized residual, which measures the accuracy of the computation. The result should be of the order $O(1)$.

For the purpose of performance evaluation, which is measured in Floating Point Operations per Second or FLOPS, only DGEFA and DGESL are considered. To obtain this value the algorithm measures the time that DGEFA and DGESL take to solve the system and it divides the number of floating-point operations by the calculated time. The number of floating point operations, on the other hand, can be approximated by the following expression: $2n^3/3 + 2n^2$. The first part of this expression, $2n^3/3$, reflects the number of operations present in the DGEFA function, while the second part, $2n^2$, accounts for the number of operations in the DGESL function. For $n = 100$ the algorithm spends more than 95% of the time executing the DGEFA routine (Figure 2.1), and this number increases even more for larger values of n.

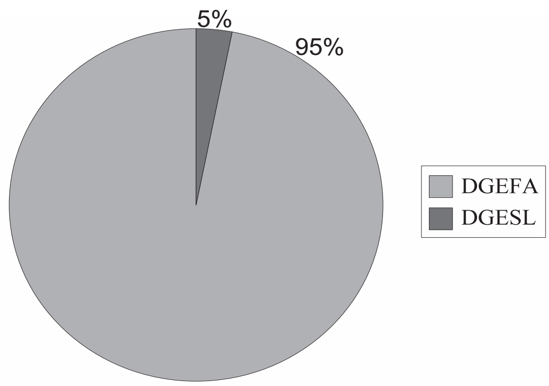

Figure 2.1: Time comparison between DGEFA and DGESL

DGEFA and DGESL call, in turn, three Level 1 BLAS routines: IDAMAX, DSCAL and DAXPY. IDAMAX is used to find the index of the value with the largest modulus within a given vector and is called $n-1$ times by DGEFA. DSCAL scales a given vector by a constant and is called $n-1$ times, by DGEFA as well. Finally, DAXPY multiplies a vector by a scalar and adds the result to another vector, and is called $n(n-k)$ times, with k varying from 1 to n in the DGEFA case, and roughly $2n$ times in DGESL. DAXPY is clearly the subroutine, among the BLAS functions, that consumes the major portion of time, more than 90% of the time for $n = 100$.

The original LINPACK Benchmark uses vector-vector operations (Level 1 BLAS) to calculate the most intensive part of its algorithm. The problem with this approach is that these types of operations incur a large number of high-level memory retrievals, which results in great losses in performance. One way of overcoming this issue is to restructure the algorithm to use matrix-vector operations (Level 2 BLAS). This method results in a data reuse increase by

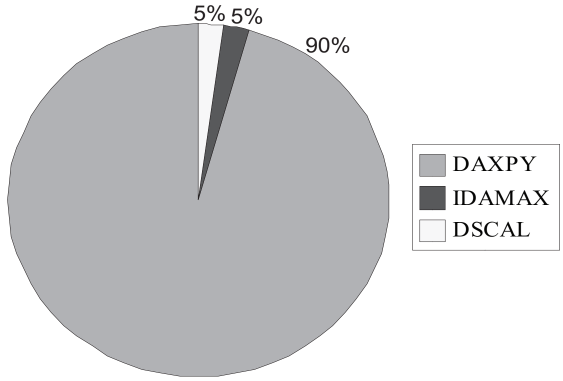

Figure 2.2: Time comparison between IDAMAX, DSCAL and DAXPY

increasing the ratio between the floating-point operations and the references to memory.

However, it is possible to further improve data reuse by partitioning the matrix into blocks and performing matrix-matrix operations on these blocks (Level 3 BLAS). With this method, data reuse is maximized since the blocks are held in the fast, local level of memory while they are being computed. This results in less data transfers between the different levels of memory and consequently a higher computation to data movement ratio. Table 2.1 shows a few performance examples[1] of the different levels of BLAS computing the LINPACK Benchmark.

Table 2.1: Level 1 BLAS vs. Level 2 BLAS vs. Level 3 BLAS

Machine	Level 1 BLAS (MFLOP/s)	Level 2 BLAS (MFLOP/s)	Level 3 BLAS (MFLOP/s)
Intel Pentium 4 (1.7GHz)	238.8	312.5	1393.0
AMD Athlon (1.2GHz)	91.0	173.7	998.3
IBM Power4 (1.3GHz)	301.2	625.0	2388.0
Sun UltraSparc (750MHz)	171.5	219.2	734.8
PowerPC G4 (533MHz)	76.2	125.2	477.6

2.1.2 HPL: The Parallel Benchmark

Highly-Parallel LINPACK (HPL) NxN Benchmark is the latest version of the LINPACK Benchmark to date, and it is also the official Benchmark used to rank the TOP500 List, which is a

[1]The values provided in this table are courtesy of [7]

list that comprises the 500 most powerful computer systems in the world. Although HPL can be run on any computer, it was specifically designed to take advantage of Distributed Memory (DM) machines. These machines are mainly composed of a group of Processing Units (PU) or processors with local memories organized in a hierarchical fashion. The interconnection between the PUs may be direct (point-to-point) or indirect (switch-based). The greatest advantage of these machines is their scalability. Increasing performance is as easy as adding more processors or increasing the memory or the network bamdwidth.

HPL introduces a whole new level to the LINPACK Benchmark. The main objective is still to generate and solve a random dense linear system of equations, however a new range of options are now available. Besides being targeted to distributed memory computers, it allows the user to select an algorithm to perform the factorization and it also provides more detailed timing and accuracy information of the final solution. Furthermore, its main routines are now isolated in high-level modules, which facilitates future improvements to the code since it is possible to rewrite the modules separately in terms of machine specific operations. All the code is written in C and to run the software, MPI [25] and either BLAS or the Vector Signal Image Processing Library VSIPL must be installed.

The algorithm solves a system of linear equations by first computing the LU factorization with row partial pivoting of the n by $n+1$ coefficient matrix, as shown on Equation 2.6.

$$P[A, b] = [[L, U], y] \tag{2.6}$$

However, since the row pivoting and the lower triangular matrix, L, are applied to b during the factorization process, the only step left to obtain the solution is to perform $Ux = y$.

HPL uses a 2-D cyclic block data distribution, i.e. the n by $n+1$ matrix is partitioned into blocks of data (of dimensions N_B by N_B defined by the user) that are cyclically distributed according to a two-dimensional grid (of dimensions P by Q, also, defined by the user) across the processors. This scheme is cut to ensure a good load balance as well as the scalability of the benchmark. Figure 2.3 shows an example of an 8-processor grid with a $P = 2$ by $Q = 4$

distribution scheme.

P0	P1	P2	P3
P4	P5	P6	P7
P0	P1	P2	P3
P4	P5	P6	P7

Figure 2.3: Example of the two-dimensional block distribution with $P = 2$ and $Q = 4$

After solving the system, the algorithm verifies the result, as in the original LINPACK, by regenerating A and b and by calculating the scaled residuals. The difference is that the residual calculations were expanded and the new expressions are shown below:

$$r_n = \frac{||Ax - b||_\infty}{||A||_1 \cdot n \cdot \epsilon} \tag{2.7}$$

$$r_1 = \frac{||Ax - b||_\infty}{||A||_1 \cdot ||x||_1 \cdot \epsilon} \tag{2.8}$$

$$r_\infty = \frac{||Ax - b||_\infty}{||A||_\infty \cdot ||x||_\infty \cdot \epsilon} \tag{2.9}$$

As in the original LINPACK, a solution is considered numerically correct when all the calculated residuals are of order $O(1)$.

2.2 Parallel Computing

Traditionally, software programs have been written to be executed serially. The programmer's model of the CPU is as if it processes the instructions, one at a time, one after the other. This has been the prominent style of programming ever since. However, due to limitations related to wire delays and power consumption issues, it has been impossible to further increase processor frequency. Therefore, an alternative to increase the performance of our programs is to execute tasks in parallel or what is called parallel computing.

Parallel computing consists of the use of simultaneous computer resources to solve a computational problem. The idea is to increase performance of a given problem by dividing it into smaller tasks that can be run concurrently. There are several types of parallelism: bit-level parallelism, instruction-level parallelism (ILP), data parallelism and task parallelism. While, the first two types of parallelism are usually already implemented by the compiler or present in the architecture where the program is being run, meaning that the programmer has almost no control over it, the same does not happen with the second two. Here the programmer has to implement the algorithm in order to run in parallel. Several factors have to be considered, such as synchronization, the number of processing nodes, the network and its memory distribution.

The main memory on a parallel computer can be one of two types: shared (where all the nodes share a single address space) or distributed (where each node has its own local memory). While in shared memory machines programming laguages communicate by manipulating shared memory variables, distributed memory machines use the message-passing paradigm. Since HPL was built to run on distributed memory machines, due to their scalability, the focus of the next section is entirely dedicated to the Message Passing Interface (MPI) [25], which is the most common message passing communications protocol in parallel programming.

2.2.1 MPI

MPI is a public-domain, language-independent interface used for message-passing in parallel programming. MPI is the result of the necessity to have a standard that enables the portability of message-passing application codes and its first standard was completed in 1994 by the MPI Forum. MPI is targeted for portability, scalability and performance and its main target platforms are distributed memory systems.

MPI's routines can be divided into two categories: point-to-point and collective operations. While in point-to-point only two nodes are involved, the sender and the receiver, in collective operations all the nodes of the network are considered. For example an MPI_Send or an

MPI_Recv are point-to-point routines and are used to send and receive messages from one node to another, respectively. On the other hand, MPI_Gather and MPI_Scatter are examples of collective operations that are used to gather/scatter data from/across multiple nodes. Within these two categories, these routines can be either synchronous or asynchronous. If a routine is called in synchronous mode, the execution of the program is blocked until the communication is totally completed between the sender and the receiver(s). In asynchronous mode, the routine is called but the program is not blocked, which means that the program will proceed with its execution while still transferring data. Here, the programmer has to be aware of the computation being executed after the MPI function call because data might be corrupted by future instructions if not used appropriately.

A basic MPI_Send/Recv message is, essentially, composed of four parameters: the source or destination rank, the tag, the communicator and the data. The source/destination rank (an integer number that uniquely identifies each node in an MPI network) is represented by a field where the user specifies which rank is sending the data, the source (in the MPI_Recv case), and which rank is receiving it, the destination (in the MPI_Send case). Ranks, starting from 0 and ending at *size-1* (*size* is a variable that defines the number of nodes in an MPI network), are assigned to each node in the network in the beginning of an MPI program. The second parameter, the tag, is given by a field in the MPI message that identifies a specific message. Since the tag is a completely definable parameter, the user can use it to send extra information to the pertaining node. The communicator is an object that defines a group of processors in an MPI session. Finally, the data is represented by three different fields where the user defines the number of data words, the type and the location of the data to be sent/received. To send and receive messages successfully both headers have to match, i.e., the ranks have to be correctly specified on the sender and receiver sides, and the tag, the communicator, the type and the number of data words have to be the same on both sides. If any of these criteria is not met, the program will stall due to invalid specification of the parameters.

As a defined standard, MPI is not complete without an implementation. Examples of MPI implementations include MPICH [11], LAM [2] and OpenMPI [10]. MPICH (for the software implementation of the LINPACK benchmark) along with TMD-MPI (used on the FPGA parallel version of LINPACK developed in this thesis), which is an MPI implementation based on MPICH developed at the University of Toronto, are the implementations used on this thesis work.

2.2.2 TMD-MPI and MPE

TMD-MPI is a light-weight implementation of the MPI standard based on a well-known implementation called MPICH. This version was specifically created to target FPGA processors, namely the Xilinx MicroBlaze soft-processor [28] and the IBM PowerPC embedded processor [13]. Although its light-weight characteristics make this implementation more suitable for small processors, that does not prevent it from being run on different types of processors by performing the necessary modifications to the lower layers.

Figure 2.4 shows the layered approach used in TMD-MPI. This approach provides good portability since to adapt this implementation to a different platform only the lower layers have to be modified.

Layer 1 is responsible for the API function prototypes and datatypes available and is implemented in C header files. Layer 2 converts collective operations, such as MPI_Barrier and MPI_Bcast, to simple point-to-point function calls. Layer 3 refers to the actual implementation of the MPI_Send and MPI_Recv functions. Finally, layer 4 includes the assembly-language macros that provide access to the FSL interface.

TMD-MPI is a software implementation and therefore can only be used with processors. However, FPGA's greatest advantage is the possibility of incorporating specialized hardware engines to execute heavier tasks or even a whole algorithm. Therefore, to communicate with other nodes, through the MPI protocol, a hardware implementation of the TMD-MPI has to be used. This implementation is called the Message Passing Engine (MPE) and it provides to the compute engines what TMD-MPI provides to the soft-processors.

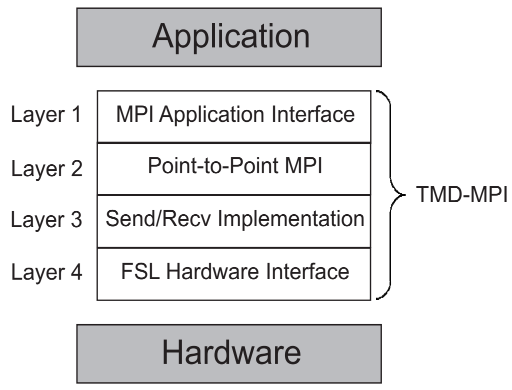

Figure 2.4: TMD-MPI Layers

2.3 Related Work

More than 30 years have passed since the first LINPACK Benchmark has been released. There is an extensive research on how to improve this benchmark's computation for multi-purpose computers. Examples of that research are Dongarra [9] where they present a better algorithm to perform matrix factorization by executing matrix-matrix operations rather than matrix-vector operations. Calahan [6] shows that the use of block algorithms is more effective on architectures with vector or parallel processing capabilities where multiple memory levels are one of the main causes of performance deterioration. Finally, Toledo [26] showed that recursion can even achieve better results than block algorithms by lowering the memory traffic.

All the research described above refers to optimizations to the LINPACK Benchmark and the libraries used in the benchmark, BLAS, for vector and generic-purpose processors. However, FPGAs do not comprise such an exhaustive work involving LINPACK Benchmark or the BLAS libraries. Examples of work to improve the BLAS libraries performance include Rousseaux [22] where a high-performance FPGA-based implementation of DGEMM is shown. C. He [12] proposed implementations of two BLAS subroutines, matrix-vector multiplication and matrix-matrix multiplication, by using a floating-point adder developed by the authors. Others, like Zhuo [29] presented design tradeoffs of implementations such as vector product, matrix-vector multiplication and matrix multiply. Finally, on the LINPACK Benchmark and the most similar work to this thesis, Turkington [27] implemented a high-level code transformation, using Handel-C, of the LINPACK Benchmark onto an FPGA.

In this thesis an implementation of the LINPACK Benchmark targeted to FPGAs is proposed. In contrast to Turkington's work, this implementation is meant to be run on a multi-FPGA system. This implementation uses TMD-MPI as the message protocol between the hardware engines, which provides an extra flexibility to the hardware implemented here since it is possible to increase the system size just by adding engines to it. This adds to the scalability of the system since as long as there is data to compute, more engines and FPGAs can be used.

Chapter 3

Hardware Testbed

This chapter describes the machine configuration and specifications used to test the hardware engine implemented in this thesis, the design flow used to build the engine and some insight on the processing elements used as well as the network infrastructure. Section 3.1 describes the characteristics of the hardware and the software used to build the system. Section 3.2 provides a high-level explanation of the processing units used in this system, as well as the possible configurations for a network node. Finally, Section 3.3 explains the design flow, introduced in Patel [21], that was used as a guide to implement the system.

3.1 Hardware and Software Overview

The work developed in this thesis uses the Berkeley Emulation Engine 2 (BEE2) [3] boards as the machine testbed. Figure 3.1 shows a BEE2 board and a simple schematic of its components. Each BEE2 board comprises five Xilinx Virtex-II Pro XC2VP70 FPGAs [28] where each has access to four DDR2 DIMMs and four high speed links (IB4X) for the user FPGAs and two for the control FPGA. The center FPGA, or control FPGA, is responsible for the management of the board and can communicate with the user FPGAs, at a maximum rate of 2.5GB/s, through parallel busses attached to the general purpose I/Os included in the Xilinx FPGAs. The user FPGAs are connected to each other in a ring fashion and they can communicate at a rate of 5GB/s. Each DRAM is 72-bits wide and can transmit at a maximum throughput of 3.4GB/s. Finally, each IB4X port can achieve an effective rate of 8Gb/s, with Infiniband cables, or 10Gb/s if the 10Gb-Ethernet-CX4 are used. Since each IB4X port uses four RocketIOs, which are high speed serial blocks, and each FPGA has twenty of these blocks, the remaining ports are connected to Serial Advanced Technology Attachment (SATA) connectors.

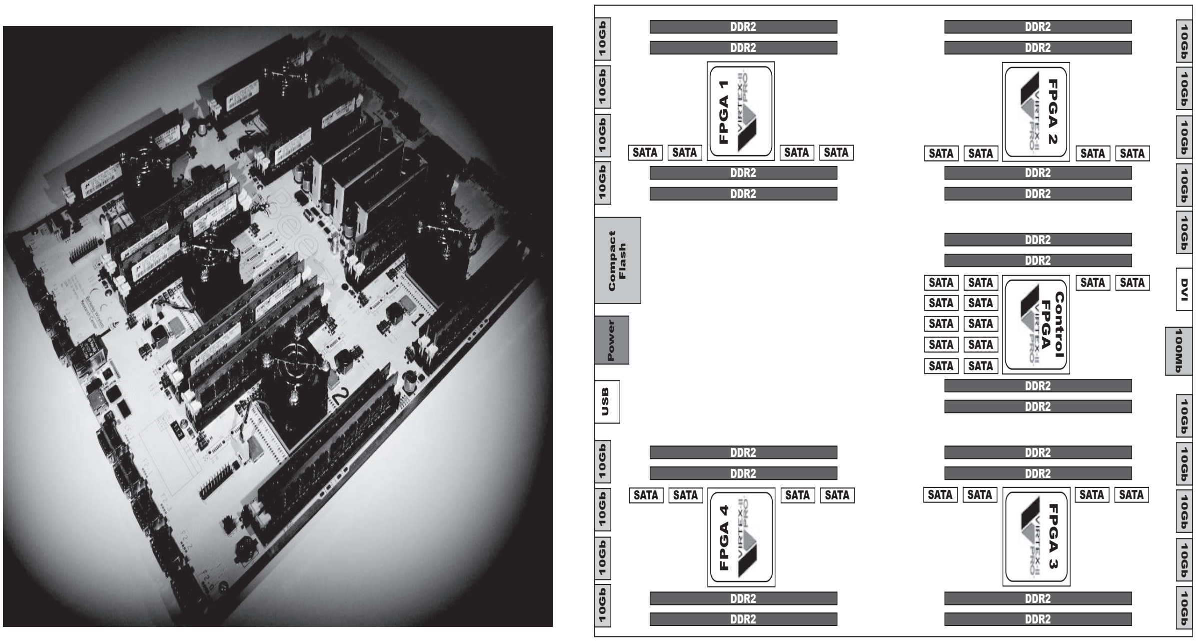

Figure 3.1: Simplified schematic of a BEE2 board

The following peripheral components are present in the BEE2: a Compact Flash card socket, an RS232 serial connector, a 100BaseT Ethernet port, a Real Time Clock (RTC), a Universal Serial Bus (USB) and a Digital Video Interface (DVI) port. Since the Virtex-II Pro has two embedded PowerPCs, it is possible to run the Linux Operating System on the control FPGA by loading the Compact Flash with the respective bitstream. This allows the user to access the board remotely by using the Ethernet interface provided. The RS232 serial port is used as an output for debugging purposes and can also be used to access the Linux command shell. Since there is only one RS232 port, a special cable was designed to provide two independent serial connections. This was done by using the *ctsN* and the *rtsN* pins as the second set of *rx* and *tx* bits. To be able to visualize a wider range of outputs, a specialized core, called the *UART_Aggregator* and designed by Mike Yan, was implemented. This block provides multiplexing from a total of 32 UART ports and redirects the data to a soft-processor that will use the RS232 port available to print the result. The USB and the DVI port are not used in this work.

To implement the multi-core multi-FPGA system presented in this thesis the Xilinx EDK and ISE8.1i [28] suite of tools and the scripts created by Manuel Saldaña and the BEE2 team are used. The two most used application GUIs in this thesis were Xilinx Project Navigator and Xilinx Platform Studio. While the first GUI is intended for hardware design through HDL coding where the programmer has access to a hierarchical view of his hardware, the second GUI is meant for creating on-chip projects by using the core created with the first tool or the vast choice of cores supplied by Xilinx. Although, the GUIs allow the programmer to synthesize, place and route, and generate the bitstream ready to be downloaded to the chip, they were only used for editing and synthesizing small amounts of code due to their quick capability in finding synthesis and syntax errors. Since the applications that perform these tasks are externally accessible, the command line was the preferred method to synthesize, place and route large amounts of code. Finally, Manuel's scripts were used to compile TMD-MPI code while BEE2's scripts were used to create the specific file used to load the bitstreams to the board through the Compact Flash.

3.2 Processing Units and the Network

Three different processing units are used in this work. The first processing unit used in this work is IBM PowerPC405, which is directly embedded in the FPGA chip. The PowerPC uses a RISC instruction set architecture and each FPGA contains two PowerPCs. In this work, PowerPCs are used as a debugging tool and to provide Linux on the main control FPGA of the system. This Linux platform allows us to program any FPGA chip within the same BEE2 board without having to re-write the Flash card every time the chips need to be programmed.

This is done by ssh-ing into the Linux control FPGA and by executing a few commands that load the bitstreams into the FPGA chips.

The second unit is the Xilinx MicroBlaze soft-processor, which is a core provided by Xilinx that is completely implemented in the memory and logic fabric of the FPGAs. The advantage of using this processor is its selective configurability, which means that the programmer can add/remove extra co-processing units such as the floating-point unit or the 32-bit barrel shifter. In this work, the Microblaze is, as the PowerPC, used as a debugging tool in the TMD-MPI network to help figuring out bugs related to the flow of messages inside the network as well as problems in the hardware engines created. It is also used in the final system as the number generator for the matrix A and vector b, to execute the DGESL function and for the residual check.

Finally, the third processing element, and the main unit in this network, is the LINPACK Benchmark engine. The LINPACK Benchmark engine is a specialized hardware computing engine created with the specific purpose of calculating the main parts of the LINPACK Benchmark algorithm. The main objective of this hardware block is to accelerate the most intensive portions of the algorithm by taking advantage of its parallelism. The other main goal for this engine is to explore the flexibility of TMD-MPI, by facilitating the addition or removal of hardware engines as necessary.

As previously mentioned, the processing units explained above interact with each other by means of a network, the TMD-MPI network. Essentially, each processing node needs two components to communicate with the rest of network, the TMD-MPI protocol and a Network Interface (or NetIf) [17]. The first component, the TMD-MPI protocol, may be implemented in software, as the TMD-MPI library or hardware, as TMD-MPE. In the case where the node is a MicroBlaze or a PowerPC either the TMD-MPI library or the TMD-MPE engine can be used, however for hardware engines, the only option is the TMD-MPE. The second component, the NetIf, is connected to the processing core or the TMD-MPE (depending on which approach is used), and is used to send/receive packets from the network by routing them to their right destination. Examples of the possible configurations are shown in Figure 3.2.

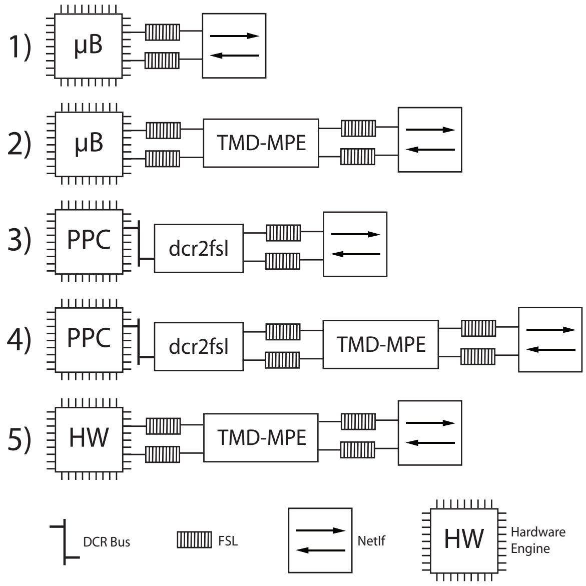

Figure 3.2: Different TMD-MPI configurations depending on the TMD-MPE use

As shown in Figure 3.2, all network blocks communicate with the computation nodes and between each other by using Xilinx FSLs (Fast Simplex Links). FSLs are uni-directional point-to-point FIFOs used for fast communication between two blocks. The greatest advantage of FSLs resides with its asynchronous mode that allows the isolation of two hardware blocks clocked at different frequencies (in this mode the FSL has to be fed with two clocks: the master side clock and the slave side clock). Since the MicroBlaze has FSL support, the FIFO may be directly connected to the processor. PowerPCs, on the other hand, need an external interface, in this case the *dcr2fsl* (core built by Patel [21]), between a processor bus (DCR in the case) and the FSL, in order to communicate with the FSL.

3.3 Design Flow

The design flow used in this work, was first introduced by Patel, and is here used as a high-level plan for the construction of this engine. Figure 3.3 shows the four steps present in this design flow. Since this flow was already thoroughly explained by Patel [21], this thesis only presents a brief explanation of the referred flow.

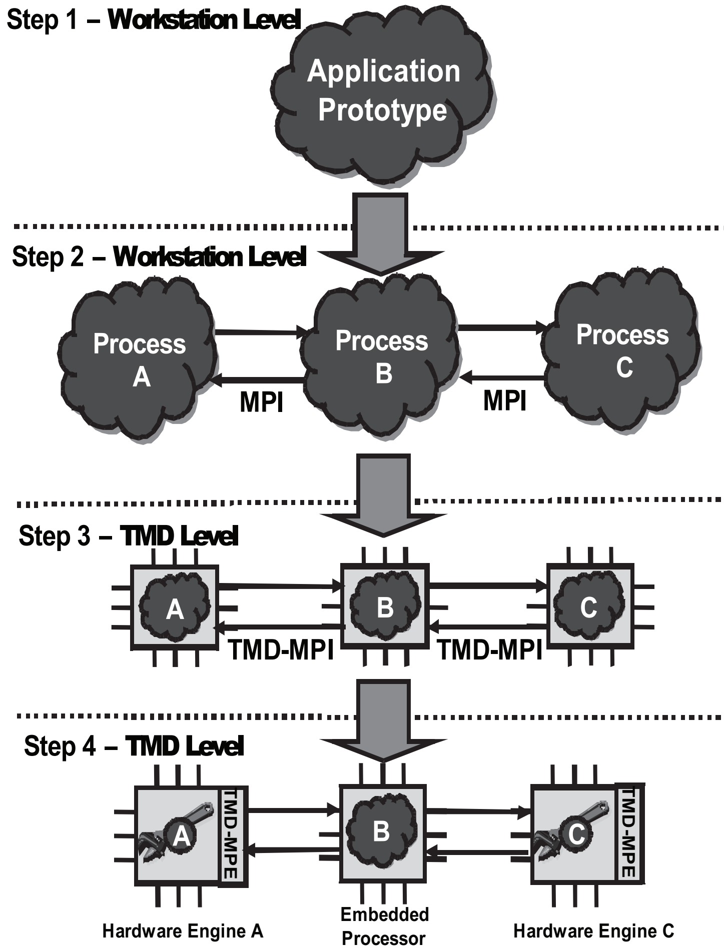

Figure 3.3: Design Flow

In the first step a sequential application is chosen (in the case of this thesis the chosen application is the LINPACK Benchmark). This sequential application may be already written in the intended language, C in this case, and therefore the only task left is to test the application on a chosen workstation. The second step is to parallelize the application by focusing on the most intensive parts of the code and by using the MPI standard as the chosen protocol. In this case MPICH is the chosen library to create the parallel program that is later tested for correctness on a cluster of four workstations.

Now that a parallel version of the intended algorithm is in place, the third step consists of porting this application to the FPGA level (here called TMD level since TMD is the name given to our cluster of FPGAs). In this step only soft-processors are used and as long as TMD-MPI, which is the light-weight MPI library specifically created for this purpose, includes all the MPI functions and datatypes used in the workstation level, this passage should occur with very little or even no modifications to the MPI code. Finally, in step four, hardware engines may be created, as an option, to accelerate the most demanding parts of the algorithm where TMD-MPE has to be used so they can communicate with the rest of the network.

Chapter 4

The LINPACK Hardware Engine

In the previous chapters we defined the plaftform and the software along with a short description of the plan used to implement the hardware designed in this thesis. In this chapter the actual hardware block and the several phases of its implementation onto the targeted platform will be explained. Section 4.1 gives a brief overview on the LINPACK implementations explained in Section 2.1 and the reason for the implementation used in this thesis. In Section 4.2 a careful analysis is provided followed by parallelization considerations targeted to the FPGA system. Section 4.3 explains the actual hardware implementation of the engine. Finally, in Section 4.4 a brief synthesis report of the LINPACK engine and an analysis of the space distribution, when mapping a real system, inside the FPGA are presented.

4.1 The LINPACK Implementations

Two different implementations of the LINPACK Benchmark were explained in Section 2.1. The first and simpler one uses the DGEFA and the DGESL functions, which use Level 1 BLAS subroutines that are based on vector-vector operations to solve a linear system of equations. The second, HPL is an optimized, parallelized and parametrizable version of LINPACK, in which a recursive variant based on Level 3 BLAS (matrix-matrix multiply) is used to compute the critical part of the algorithm. Both these algorithms perform the same number of floating-point operations and can therefore be compared in terms of performance results.

Due to its optimized recursive method based on matrix-matrix functions, HPL can fully exploit the multiple levels in the memory hierarchy of general-purpose computers. However, in FPGAs the situation is different. Besides off-chip memory (such as DDR memory), all the memory inside the FPGA, which can be implemented through logic or BRAMs in the Xilinx case, may be accessed with a throughput of one word per cycle and can be configured to feed multiple parts of a computation engine at the same time. Moreover, since the execution flow is done in hardware, the programmer is completely aware of what is inside of each memory, which means that no "misses" will occur when looking for a specific element of data. Therefore the memory hierarchy problem seen in processor systems does not happen. For this reason and for its simplicity, the algorithm chosen as the baseline to target the hardware engine approach was the LINPACK Benchmark with Level 1 BLAS instead of the more complex HPL version.

Although the LINPACK Benchmark is meant to be run with 64-bit floating-point numbers, the implemented hardware version in this thesis uses 32-bit floating-point numbers. The reason is that double-precision floating-point units use much more FPGA resources making it difficult to replicate many engines inside the FPGA. In this work the interest is more focused in revealing architectural issues rather than exactly replicating the benchmark. Using 32-bit number will satisfy this requirement. Moreover, the network cores used in this work are only prepared to work with 32-bit numbers and an adaptation to 64-bit cores is not in the scope of this thesis. For a fair comparison, the software versions are modified to run in 32-bit mode. This approach will still help with understanding architectural issues. Modern FPGAs are able to support double precision better. In this case, the MPE and network datapath should also be increased to 64 bits.

```
/* Generate random matrix A and vector b */
matgen(*A[][], *b[]);

/* Calculate LU factorization with partial pivoting of A */
dgefa(*A[][], *ipvt[]);

/* Solve the system using the LU factorization */
dgesl(*A[][], *ipvt[], *b[]);

/* Calculate residual */
residual(*A[][], *b[]);
```

Figure 4.1: Main functions of LINPACK Benchmark

4.2 Analysis and Parallelization

Since a serial version in C of this algorithm already exists (as shown in Figure 4.1), the next step to this implementation is to analyze the algorithm and find possible parallelization targets. A higher level analysis of this algorithm has been previously done, and is explained in Section 2.1.1. As previously seen, DGEFA is the routine that occupies the largest slice of time. Therefore, the parallelization of the algorithm is completely focused on DGEFA while DGESL is not parallelized. DGESL is only executed by the first rank in the network.

```
/* dgefa(*A[][], *ipvt[]) */
for (k = 0 : n-2)                                (loop k)
   pivot = idamax(A[k][k]) + k;                  (loop idamax)
   ipvt[k] = pivot;
   if (A[pivot][k] != 0)
      t = -1/(A[pivot][k]);
      swap(&A[pivot][k], &A[k][k]);
      dscal(&A[k+1][k], t);                      (loop dscal)
      for (j = k+1 : n-1)                        (loop j)
         t = A[pivot][j];
         swap(&A[pivot][j], &A[k][j]);
         daxpy(&A[k+1][j], A[k+1][k], t);        (loop daxpy)
```

Figure 4.2: Code of the DGEFA function

Figure 4.2 shows the code of the DGEFA routine. Three major Level 1 BLAS subroutines are called here: IDAMAX, DSCAL and DAXPY. When analyzed carefully, a total of five loops, with a maximum depth of three, can be found in the DGEFA routine: two explicit loops, *loop j* and *loop k* and three other loops comprised by the Level 1 BLAS functions. *Loop k* is the loop on the highest level of the hierarchy by containing all the loops referred to before, and *loop*

DAXPY lies on the bottom of this hierarchy by being the loop that is executed more often and therefore the critical path of the DGEFA routine, representing about 95% of the total time spent in this routine. Since *loop j* (which includes *loop DAXPY*) contains no loop dependencies, it becomes the chosen loop to be parallelized. Parallelizing this loop consists of simply splitting the iterations across the different computation nodes.

Although *loop j* is the only loop being truly executed in parallel, to reduce the number of messages flowing across the network, all the computation nodes contain the full DGEFA function. This way, each rank performs the IDAMAX and DSCAL functions on the corresponding column of matrix A and broadcasts the result to the rest of the network.

To maintain the load balance across the system, after generating the initial data, the main rank cyclically distributes the matrix A columns across the ranks, i.e., 1st column goes to rank 0, 2nd column to rank 1 and so on. This way, the load balance across the network is kept and no further messages have to be sent to re-balance the load.

In short, three main MPI sections are introduced: just after generating the initial data where rank 0 sends each rank its respective portion of data; inside the DGEFA routine, just after performing DSCAL, where the rank that computed IDAMAX and DSCAL broadcasts the computed column and the respective pivot; and, finally, after the DGEFA function, where all the ranks send the factorized matrix and the respective pivots back to rank 0. Figure 4.3 shows these modifications.

After implementing and testing this new parallelized version for correctness, the next step is to port it to the FPGA system. The only requirement in this step is to build the system with the desired amount of soft-processors and load them with this code, which should be the same, with the exception that now the TMD-MPI library is used. Finally, the last step in this design-flow is to implement the most demanding portions of the algorithm in hardware.

```
matgen(*A[][], *b[]);
MPI_Scatter(*A[][]);
dgefa(*A[][], *ipvt[]);
MPI_Gather(*A[][], *ipvt[]);
dgesl(*A[][], *ipvt[], *b[]);
residual(*A[][], *b[]);
```

```
/* dgefa(*A[][], *ipvt[]) */
for (k = 0 : n-2)
   pivot = idamax(A[k][k]) + k;
   ipvt[k] = pivot;
   if (A[pivot][k] != 0)
      t = -1/(A[pivot][k]);
      swap(&A[pivot][k], &A[k][k]);
      dscal(&A[k+1][k], t);
      MPI_Bcast(*A[k+1][k], pivot);
      for (j = k+1 : n-1)
         t = A[pivot][j];
         swap(&A[pivot][j], &A[k][j]);
         daxpy(&A[k+1][j], A[k+1][k], t);
```

Figure 4.3: Parallelization of the code

4.3 The Hardware Engine

As observed in the previous section, *loop j* (inside DGEFA) is the most intensive portion of this algorithm and is therefore the main focus of attention. However, as in the software version, the complete DGEFA function is implemented. Unlike what happens in software, in hardware this brings extra costs since implementing the complete function instead of just the critical loop, implies the use of more logic on the FPGA. In this case the extra logic corresponds to five more states in the main state-machine, a few registers, a floating-point comparator and a floating-point divider, which represent a considerable percentage of the total size of the core. Although this can be optimized, it is not crucial to the final size of the core, since even if we removed this logic from all the cores besides the main one, due to the amount of space occupied by the other blocks in the FPGA adding extra LINPACK engines would still be impossible.

The hardware engine implemented in this work consists of three main blocks: the main state-machine (or main FSM), the Level 1 BLAS engine (referred to as BLAS1 engine from this point forward), and the MPE header state-machine (or MPE Header FSM) used to interface with TMD-MPE. Figure 4.4 shows a high-level schematic of the LINPACK engine connected to the TMD-MPE interface. The following sections describe each one of these components and how they interact with each other.

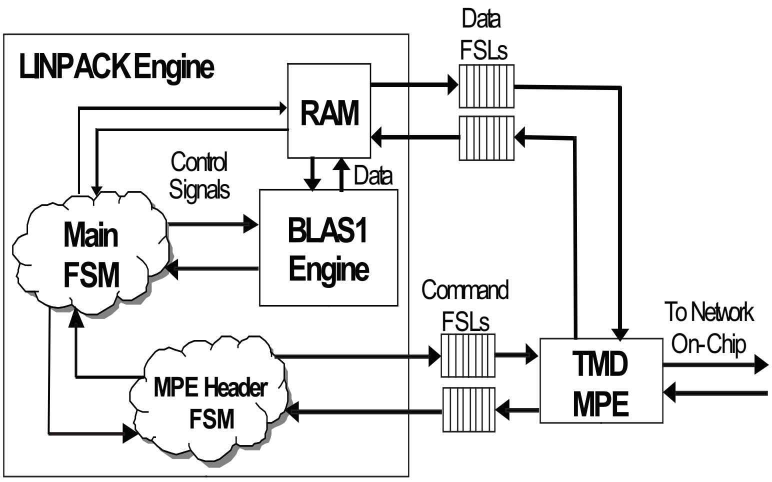

Figure 4.4: LINPACK Engine High-Level Schematic

4.3.1 Main FSM

The main FSM is a large state-machine that controls the data flow inside the engine. The data flow starts by receiving the engine's respective rank from the first rank in the network. Every time a message is sent or received by the engine, the MPE Header state-machine is called. This is done by sending the necessary control signals to the MPE Header FSM that ensures the right protocol is used (this protocol is explained in Section 4.3.3).

After all ranks have been initialized, the DGEFA flow starts. Since, in the beginning, the engine has no information on the matrix and network parameters being used, the first message to be received contains no data-words[1]. Instead, a zero-length message is received and all the initial information the engine needs to know to start its computation is encoded in the tag of the message. Figure 4.5 shows the format of the first tag. The last four bits are used to send the opcode, the next eight bits contain the number of columns assigned to the engine, the following eight bits are used to encode the MPI Size or the number of total ranks in the network, and, finally, the first twelve bits describe the size of one column or the variable *n* in the DGEFA routine.

After having received this message the engine starts by receiving its portion of matrix A.

[1]All the words in this work are 32-bit wide

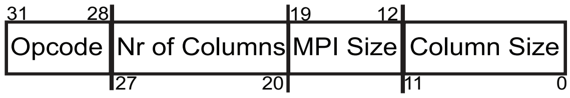

Figure 4.5: First tag received by the engine

This data is completely stored in dual port BRAMs that allow simultaneous write and read operations, which is very useful since it allows the same BRAMs to be used when feeding and storing data from the floating-point blocks, and hence maximizing the use of the floating-point path pipeline.

This is where the computation begins and, as in the software version, this is the point where only the engine that contains the column corresponding to iteration k computes the IDAMAX and DSCAL functions. The result is then broadcast to all the other engines and *loop j* is executed.

After computing all iterations in *loop k*, all the computed data is sent to the main rank that will compute the DGESL function concluding the benchmarked routines.

4.3.2 BLAS1 Engine

This engine is used to calculate all the Level 1 BLAS functions used in the algorithm (IDAMAX, DSCAL and DAXPY shown in Figure 4.2). To perform a desired subroutine, the engine has to be fed with a valid signal (to enable the engine), an opcode (specifying which function to execute) and the required data (which might be stored in either a FIFO or a register, depending on the operations to be executed). Inside the engine, the data flows through the necessary pipelined floating-point operations, according to the specified function, until their exit point where the results, as the incoming data, might be saved in either a FIFO or a register. This data flow and opcode decoding is controlled by an FSM tailored for this purpose. Three different floating-point operations are realized inside the engine: multiplications (used in DSCAL and DAXPY), additions (used in DAXPY) and “greater than” comparisons (used in IDAMAX). Figure 4.6 shows the BLAS1 Engine.

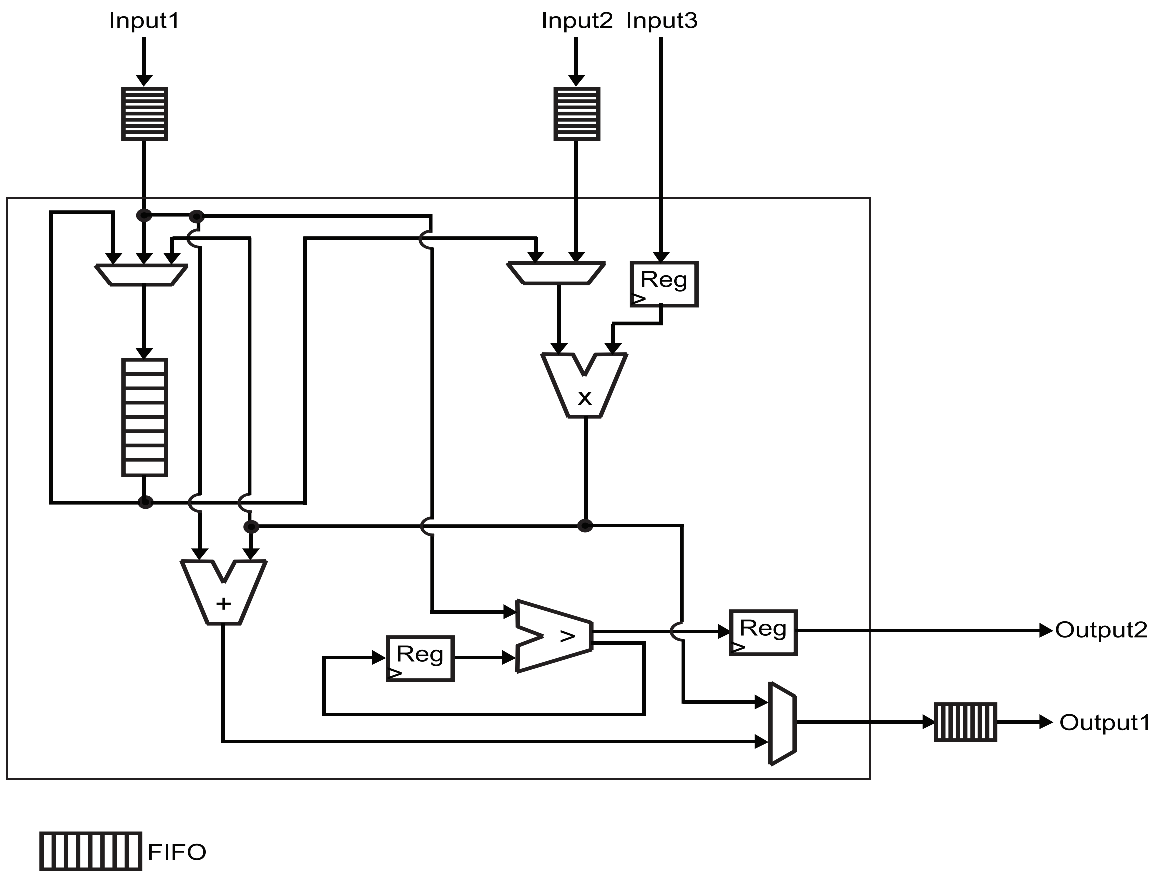

Figure 4.6: Simplified data path of the BLAS1 Engine

In IDAMAX only one external input is used (Input1). The values that come from this input are then compared to the value stored in the register that is sitting by the comparator and, if a larger value is found, the new value is stored in this register. After comparing against all the values of the vector (column of the matrix), the engine outputs the index corresponding to the larger value found.

DSCAL scales a vector by a constant (as shown in Equation 4.1). In the engine this is done by feeding Input2 with the vector to be scaled, and Input3 with the respective scalar. When the values start to come out of the floating-point multiplier, the result is directed to both the Output1 and the engine's internal FIFO. The values that are outputted (Output1) are then saved back in the main memory. These values are also kept in the internal FIFO of the engine because the next function to be executed, DAXPY, will make use of this vector. This way time can be saved by not having to load this vector again.

$$v_1 = \alpha v_2 \tag{4.1}$$

DAXPY scales a vector by a constant and adds the result to another vector (as shown in Equation 4.2). Two external inputs plus the vector stored in the internal FIFO are used in this function. First, the function starts by multiplying the vector that is inside the internal FIFO by Input3, which is a scalar. Since this vector (that is inside the internal FIFO) is used more than once, a feedback path allows the values to be saved back in the same FIFO. When the values of the multiplier are ready, Input1 is read and they are fed to the adder. The result is then outputted and saved back in the main memory.

$$v_1 = \alpha v_2 + v_3 \tag{4.2}$$

4.3.3 MPE Header FSM

The MPE Header FSM is a state-machine used to create the messages being sent by the LINPACK engine. These messages have to obey a defined format [17] to be accepted by the TMD-MPE. The format is shown in Figure 4.7. The first word sent is the header of the

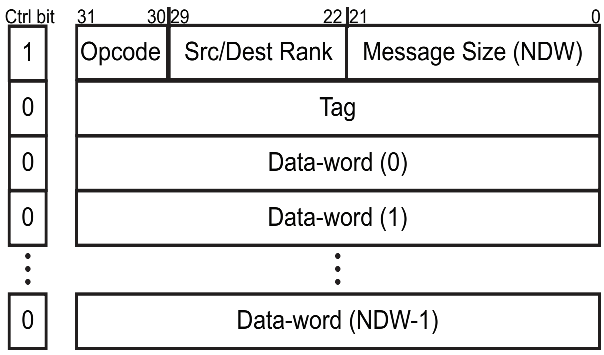

Figure 4.7: TMD-MPE protocol

message and is used to define the parameters of the message. The opcode field (the 2 leftmost bits) defines the direction of the message, which may be either incoming (MPI_Recv) or outgoing (MPI_Send). The second field (8 bits) is the rank, which determines the destination rank, in case of an MPI_Send, or the source rank, in the case of an MPI_Recv. The third and final field (22 bits) determines the number of data-words to be sent or received. This word is sent through the FSL to the TMD-MPE core and is the only element of the message that has the control bit of the FSL assigned to ’1’.

The second word is the tag and is used to identify a message. This field is 32 bits wide and, as seen before, this field can also be used to encode necessary information for the ranks receiving the message. The remaining elements are the data, and the number of data-words being sent here has to correspond to the number defined in the third field of the header.

4.4 FPGA Utilization

One of the most important characteristics when implementing a hardware engine on an FPGA is its size. Since each FPGA has very limited space, the size of each component plays an important role in the performance that may be achieved with a single chip. For example, in some cases it is preferrable to have two slower cores instead of a slightly faster one.

Figure 4.8 shows the LINPACK Engine device utilization and timing summary. This engine utilizes 4360 4-input LUTs, which on this chip, the 2VP70, represents about 6.5% of the chip (66176 LUTs). If this core were to be instantiated alone, it would be possible to fit around 15 engines inside this chip. The timing summary estimates a clock rate of 108MHz. The actual clock speed utilized in the hardware is 100MHz.

Although the engine is large enough to fit 15 times inside the FPGA, after adding all the components necessary to test the system only six engines were successfully mapped onto the chip. Table 4.1 shows the device utilization, in terms of 4-input LUTs, of the larger cores in the chip. The major portion of space, besides the engine, lies with the network cores (represented by the first five blocks after the LINPACK Engine), that allow engines to communicate with each other within the same FPGA or to another FPGA (including the PowerPC). Other cores, such as the ones that have to be attached to the PowerPC (remaining blocks on the table), also,

```
Device utilization summary:
---------------------------

Selected Device : 2vp70ff1704-7

 Number of Slices:                    2970  out of  33088     8%
 Number of Slice Flip Flops:          3282  out of  66176     4%
 Number of 4 input LUTs:              4360  out of  66176     6%
    Number used as logic: 3714
    Number used as Shift registers: 134
    Number used as RAMs: 512
 Number of BRAMs:                       18  out of    328     5%
 Number of MULT18X18s:                   7  out of    328     2%


Timing Summary:
---------------
Speed Grade: -7

   Minimum period: 9.259ns (Maximum Frequency: 108.003MHz)
   Minimum input arrival time before clock: 4.930ns
   Maximum output required time after clock: 6.853ns
   Maximum combinational path delay: 2.043ns
```

Figure 4.8: Synthesis report of the LINPACK Engine

represent a significant portion of the hardware.

Table 4.1: More significant cores in terms of device utilization

Cores		4-Input LUTs	Number of Occurrences	Total of 4-input LUTs	% Occupied
Engine	LINPACK Engine	4360	6	26160	40
Network	TMD-MPE	896	6	5376	8
	NetIf	579	11	6369	10
	PLB-MPE	2685	1	2685	4
	FSLs	44	154	6776	10
	FSL2IC	349	4	1396	2
PPC	Async DDR2	566	1	566	1
	DDR2 Controller	1586	1	1586	3
	PLB DDR2	1367	1	1367	2
	PLB Bus	852	1	852	1
	PLB2OPB Bridge	754	1	754	1
	UART	512	1	512	1
Other		11777	-	11777	17

Chapter 5

Performance Analysis of Results

This chapter is divided into five sections. Section 5.1 describes the methodology used to get the results that are presented in the following sections. The profiler project, realised by Daniel Nunes [20], is introduced and a brief explanation of its functionality is also given in this section. In Section 5.2, performance and efficiency results are presented. Here, it is possible to observe the engine behaviour when compared against the software version, as well as its main problems and focuses for improvement. Section 5.3 describes a scaled performance analysis of the LINPACK engines. Section 5.4 presents the theoretical peak performance that may achieved with the engine and the future possibilities of implementing the engine on a different platform. Finally, Section 5.5 describes the advantages of having an FPGA cluster compared to a cluster where processors are used.

5.1 Methods of Analysis

Three different methods were used to analyse the performance of the hardware engine developed in this thesis. Method 1 consists of creating a simulation model, in Modelsim [4], of the hardware to be tested and measuring performance by analysing the resulting waveforms. By feeding the LINPACK engine with the desired data it is possible to determine the exact number of cycles the engine takes to complete a given task and hence its performance. A C program, using the *matgen* function used in the LINPACK algorithm, was created for this purpose. After generating the matrix to be tested it also generates the testbench VHDL file that is used to perform the simulation. The waveform is then used to measure the computation. The advantage of using the waveform generated by Modelsim is that it is possible to measure different parts of the computation with great detail. This method was, essentially, used for correctness and performance measurements when only one engine is used.

However, Method 1 has two great drawbacks: the simulation time and the complexity. When computing smaller sets of data, simulation time is tolerable, however when large sets of data are used the waiting period may become overbearing and counterproductive. The complexity is related to the amount of data to be analysed. If the analysis done includes several engines, this job may be very difficult and tedious. Method 2 is used to measure the performance of multiple LINPACK Engines by doing the measurement on a system running in real time. To use this method an MPI system is created, where one PowerPC and multiple LINPACK engines are connected through an MPI network. The PowerPC generates the matrix to be sent to the engines, measures the time, performs the DGESL function and calculates the residue while the engines only calculate the DGEFA function and send the data back to the PowerPC. To measure the time, the PowerPC uses the function *MPI_Wtime* (implemented in TMD-MPI). The timer starts counting when the PowerPC gives the start order to the first engine and it stops counting only when the first engine sends the first portion of data back to the PowerPC. This means that the first data distribution and the final data gathering are not accounted for in the time measurements, as the most important measurement is the DGEFA function computation.

Although Method 2 provides an accurate measurement of the work done by the engines, it is very difficult to know exactly where, within the engine cluster, the time is being spent. Finally, Method 3 is a real-time profiling approach using the TMD-Profiler, which is a profiler specifically created for systems using TMD-MPI [23]. This profiler gives a much better visibility to how all the LINPACK engines are interacting between each other. The TMD-Profiler is composed of four main cores: the cycle counter, the tracer blocks, the gather block, and the collector block. The cycle counter is a 64-bit counter used to count the number of clock cycles. Each FPGA has one cycle counter and all the counters are synchronized between each other. The tracer block is used to keep track of the time taken by a given event, which can be any type of computation that needs to be measured. Specialized blocks were created to measure the *send* and *receive* times of a hardware block using the TMD-MPE. In this case, three tracer blocks per hardware engine are used: two for communications (*sends* and *receives*) and one for computation. At the end of each event the gather block receives the timing information collected by each tracer and redirects this data, acting as a multiplexer, to the collector block that will store all the timing data in the DDR memory. As Figure 5.1 shows, each User FPGA contains one gather block keeping track of all the tracers and the Control FPGA contains one collector block receiving from all the gather blocks. Finally, the PowerPC gets the final statistics, which can be visualized with Jumpshot [11].

5.2 Results

The results obtained in this thesis are compared to a high-end CPU (Intel Xeon dual processor 3.4GHz with 1MB of L2 cache and 3GB of DDR RAM) with Linux and GCC4.2.3 installed. The benchmark being run by the processor is the LINPACK Benchmark with Level 1 BLAS. Since one engine cannot hold a 100×100 matrix on its own because larger memories were not successfully placed and routed, results given for one engine (when the matrix is 100×100) are taken with the simulation method (Method 1). Because this does not happen when multiple engines are computing a 100×100 matrix size, due to the data being equally partitioned across all the engines, Method 2 and Method 3 were used to perform all the other tests. Since, in the case of the scaled performance test, the amount of data per engine does not change, the size of the matrix had to be reduced to 80×80.

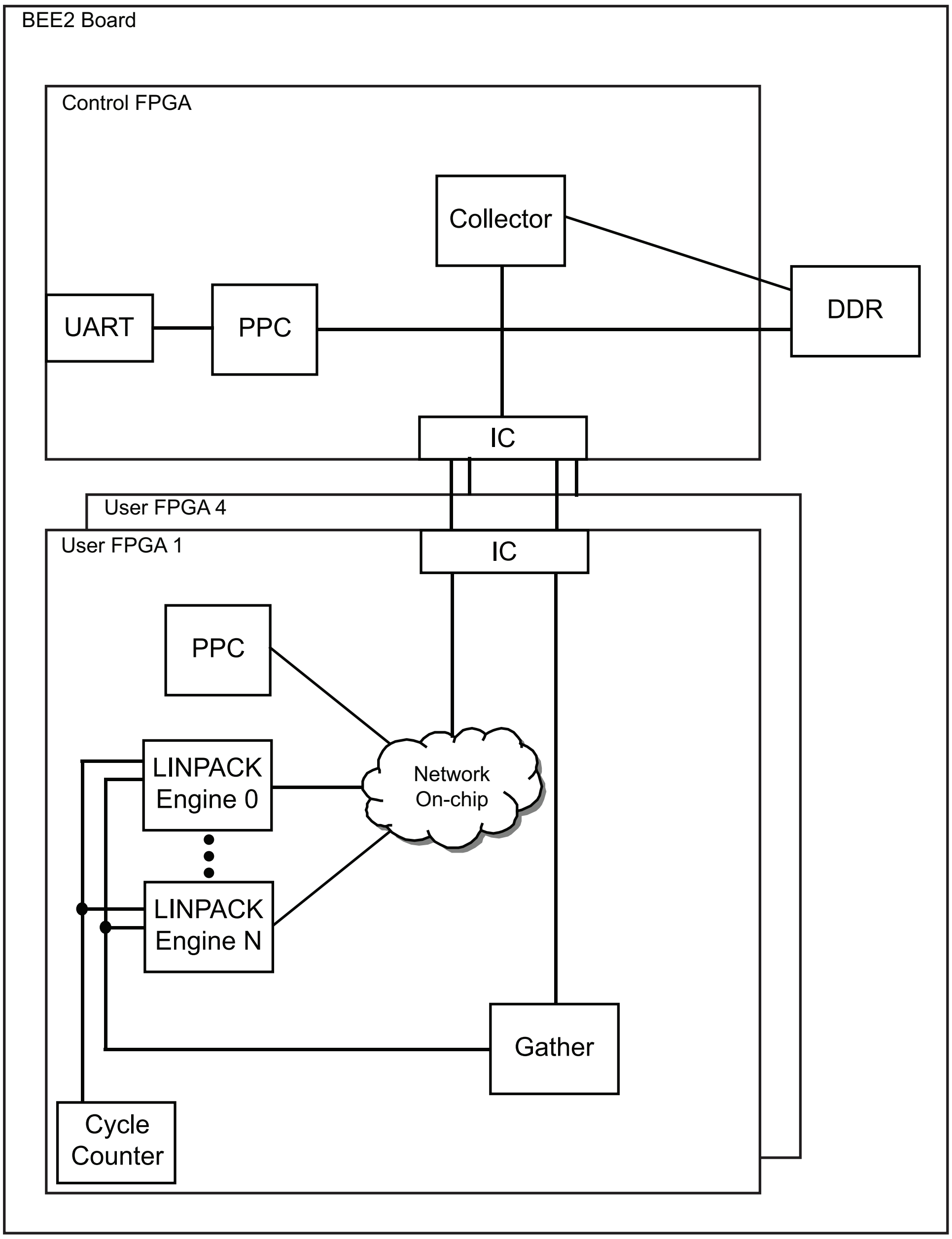

Figure 5.1: Simplified schematic of the system used to test the LINPACK engine with the TMD-Profiler

5.2.1 One LINPACK Engine vs. One Processor

In the first performance test the dimensions of the matrix used are 100×100 (or *n=100*) and one LINPACK engine, running both the DGEFA and the DGESL benchmarks, is compared to the processor. For this matrix size the engine takes 0.00568s to complete the full benchmark (corresponding to 121 MFLOPS), while the reference processor spends 0.00215s to calculate the benchmark (corresponding to 319 MFLOPS). This corresponds to a speedup of 0.38 when comparing one engine to one processor, where speedup is given by the following expression:

$$Speedup = \frac{Performance(MFLOPS)_{engine}}{Performance(MFLOPS)_{processor}} \tag{5.1}$$

Because the DGESL routine is not parallelized, all the performance tests that follow (where the algorithm is performed in parallel), only consider the main part of the benchmark, i.e. the DGEFA function. The DGEFA routine is the more significant portion of the code accounting for more than 95% of the computation. The DGESL function is performed by the PowerPC just to test the correctness of the results. In a complete system DGESL would also be parallelized.

In the next test, one engine, running only DGEFA is compared to one processor. While one engine takes 0.00558s (123 MFLOPS) to perform the DGEFA routine, the processor takes 0.00211s (315 MFLOPS), which corresponds to a speedup of 0.39.

5.2.2 One FPGA vs. One Processor

In this test, one FPGA (or six engines) is compared to one processor. Here, one FPGA takes 0.00176s (379 MFLOPS) to perform the DGEFA routine, corresponding to a 1.20-fold speedup when compared to the processor.

When comparing the speedup of one engine against the processor and the speedup of six engines against the processor it is possible to see that the increase in performance is not linear. Although beating the processor's time, the speedup of six engines, when compared to only one engine, is only 3.1x, which is about half of the ideal speedup mark of 6.0 (since six engines are being compared to one). In the next sections a scalability test is performed and this problem is addressed.

5.2.3 Scalability Test

In this performance test the engine is replicated up to eight times. Figure 5.2 shows a speedup graph of the engine that represents the engine scalability for a fixed size matrix of dimensions 100×100. Here, the speedup is given by the following equation:

$$Speedup = \frac{Performance(MFLOPS)_N}{Performance(MFLOPS)_1} \tag{5.2}$$

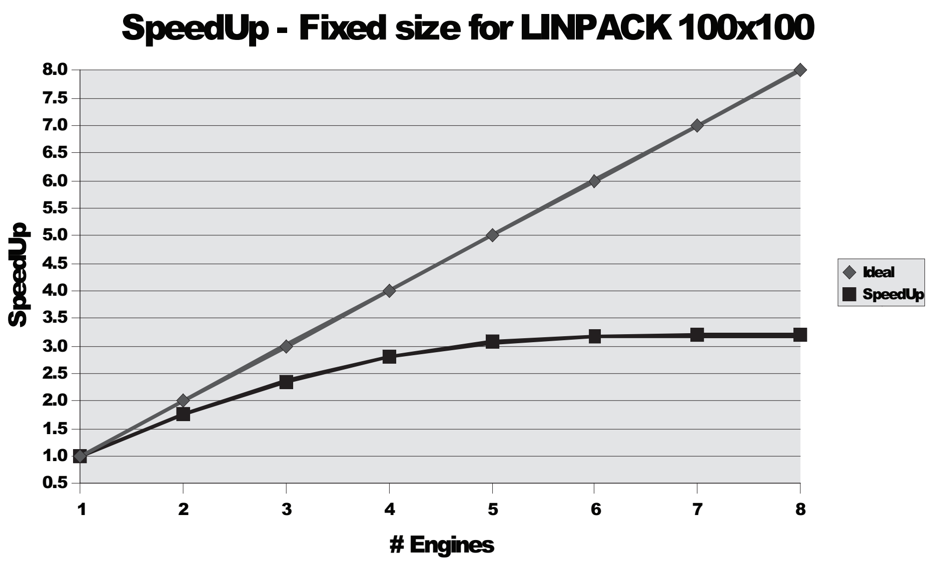

Figure 5.2: Speedup of the LINPACK engine with fixed problem size up to 8 processing units

As it is possible to observe in Figure 5.2, the engines cannot yield a sustained speedup for such a small matrix, which contributes to poor scalability of this system for this case to keep the engines busy. As shown by the figure, the maximum speedup obtained is 3.19 at the seventh engine, which is the point where adding more engines will not improve the performance of the system. To better understand the problem, an efficiency graph is plotted in Figure 5.3. The efficiency measured here is given by the following expression:

$$Efficiency = \frac{Speedup}{N_{processors}} \tag{5.3}$$

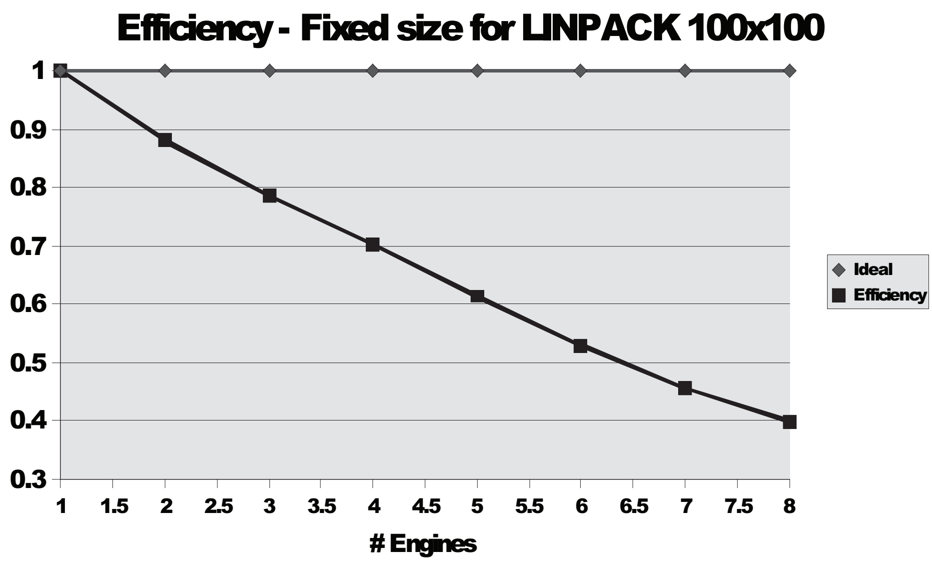

Figure 5.3: Efficiency of the LINPACK engine with fixed problem size up to 8 processing units

The objective of this graph is to show that the engines are not being efficiently used, i.e. they are not busy all of the time. Moreover, the efficiency of the engines drops to about 88%, as soon as the second engine is introduced and keeps dropping continuously to a low of 40% by the eighth engine.

The speedup and efficiency problems seen in Figure 5.2 are explained by the fact that computation time is being severely overcome by non-computation time. By not considering the overlap between the computation time and the non-computation time (which is almost inexistent), and that the non-computation time is either communication time or idle time, the following expression is obtained:

$$T_{total} = T_{computation} + T_{communications} + T_{idle} \tag{5.4}$$

Considering that the total computation on each processor decreases as the number of engines is increased (since overall computation is divided amongst the engines) the problem seen in the plots represented in Figure 5.2 is either related to an increase in communication time, an increase in idle time or an increase in both.

For the next experiment, the TMD-Profiler is used. The experiment parameters are kept the same (eight engines with $n = 100$) and the profiler is added to the system.

Three different events are profiled: the computation, the *sends* and the *receives*. After profiling the system, the final statistics clearly show that computation is being overcome by communications. Table 5.1 shows these statistics, where only 40.6% of the time the engines are performing computation, and as a result adding more engines will only increase communication time and reduce the performance. Figure 5.4 shows the problems present with this

Table 5.1: Average percentage of the Computation, Send and Receive events

Computation (%)	Sends (%)	Receives (%)
40.6	8.7	46.7

configuration, which, in the algorithm, is represented by the broadcast performed just before the *loop j* (that includes DAXPY). In the figure each rank is represented by three rows: one for computation, one for the MPI_Send and another for MPI_Recv; each of these descriptions is preceded by the number associated to its rank, for example Rank_1_RECV is the MPI_Recv of rank 1. As the figure shows, rank 1, is performing the broadcast, which can be seen by the arrows that connect the large bar in Rank_1_SEND to all the other bars in the MPI_Recv of the other ranks. Eventually, this broadcast becomes a problem because it delays rank 1 so much that when rank 1 is starting its computation, rank 2 will be almost finished and when rank 2 finishes its computation, rank 1 will still be performing its computation delaying the MPI_Send performed by rank 2. The time spent by each rank performing the broadcast (shown by the MPI_Send of rank 1 in the figure) added up to the time each broadcast source has to wait for other ranks to be ready (shown when the MPI_Send of rank 2 is waiting for rank 1 to finish) are the main problems of this configuration.

Although hidden by the way the profiling is performed, the elongated bars, in MPI_Send and MPI_Recv, represent not only the communications time, but also idle time. This idle time corresponds to the period where either MPI_Send or MPI_Recv are waiting for the other node to be ready (in MPI_Send when the node is waiting for the receiver to be ready, and in MPI_Recv when the node is waiting for data).

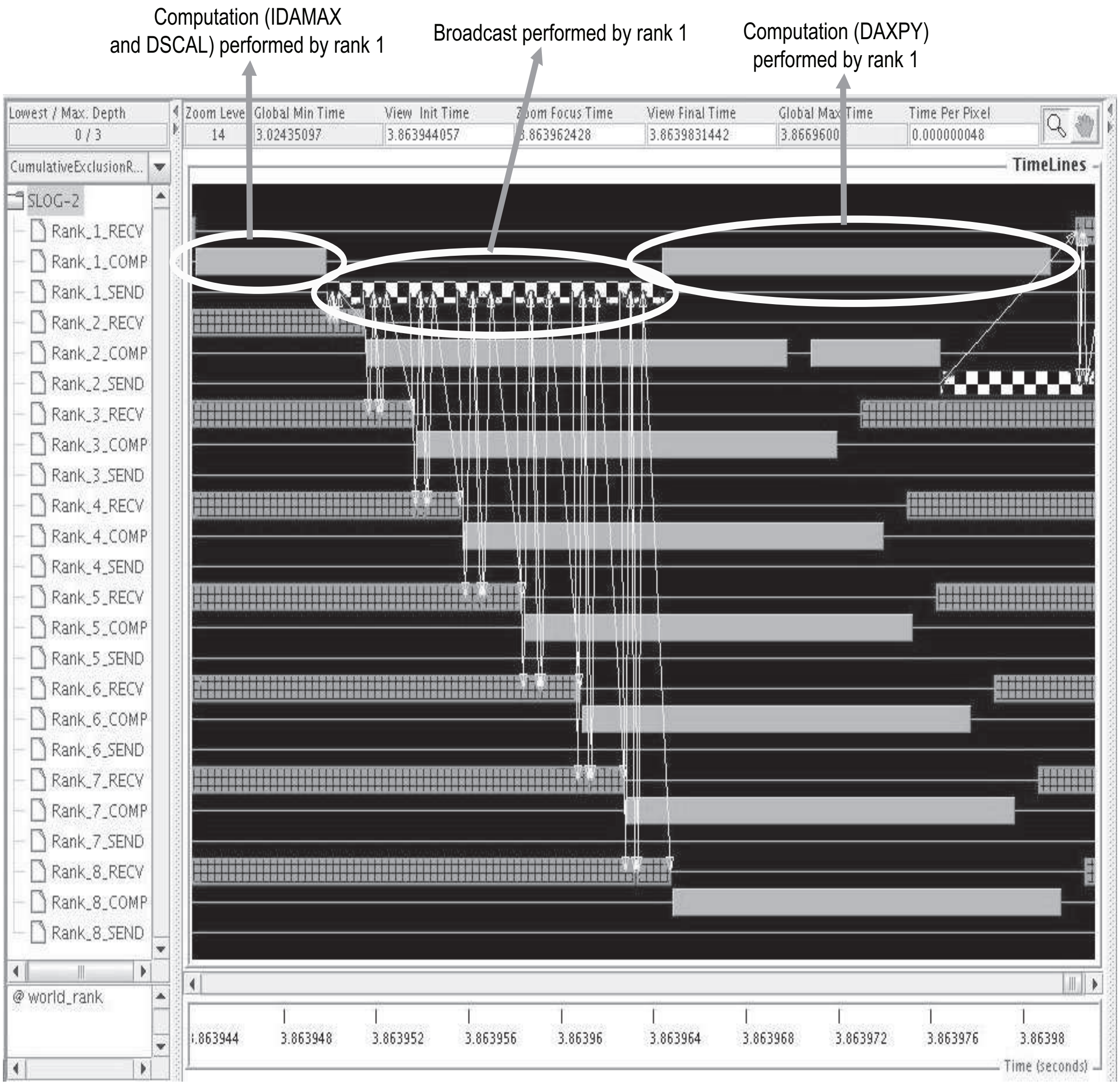

Figure 5.4: Jumpshot image of the broadcast performed by rank 1

The main problem with this broadcast is its sequential approach, i.e. the rank performing the broadcast will send the data to all the other ranks one after the other. In larger systems this might become a bottleneck since by the time the source has finished sending the message to the last rank, the next broadcast source might already be ready to send its data, idling itself and all the other ranks that are waiting for the data. One of the possible solutions for this is to implement a different broadcast method, such as the tree-based broadcast. This way, instead of having to perform *n* sends, only $log_2(n)$ sends would be performed. Another solution that might alleviate this problem is to parallelize the send and the computation of the source rank, so that when the next source rank is ready to broadcast, the previous source rank would also be ready to receive the data.

5.3 Scaled Performance Analysis

Another way to look at the results shown in the previous section is by considering that the computation being performed by the engines is not enough, i.e. the computation-to-communication ratio is too low, not because the communication is too high but because there is not enough data to keep the engines busy. Therefore, to show that it is possible to hide the communications and that the engine has potential to achieve a sustained speedup, the next test is a performance analysis with a scaled problem size.

Figure 5.5 shows a scaled problem size speedup plot, with the problem size starting with an 80×80 matrix and varying according to the following expression: $80 \times 80 \times N_p$, e.g., if the number of processors being tested is three, the total amount of data being computed is $80 \times 80 \times 3 = 19200$ and the size of the new matrix being computed would be $\lfloor\sqrt{19200}\rfloor \times \lfloor\sqrt{19200}\rfloor$ or 138×138, i.e. the square root of the resulting size is truncated to the nearest integer number. The reason for starting with an 80×80 matrix and not 100×100 is, that, in contrast to previous performance results presented here, where in only one test the engine has to hold so much data, here the engines will always hold the same amount of data. The idea of this test is to maintain the workload per engine while increasing the number of engines.

The results show that the engine is able to keep sustained the speedup for up to eight engines with this configuration. However, this workload per engine might still not be enough for a very large system since it is possible to observe that after the eighth engine, the speedup starts to fall below the ideal curve, and a larger matrix might be needed to obtain a higher efficiency. An efficiency plot is shown in Figure 5.6.

It is also possible to observe that, between two and five engines, the speedup of the engines

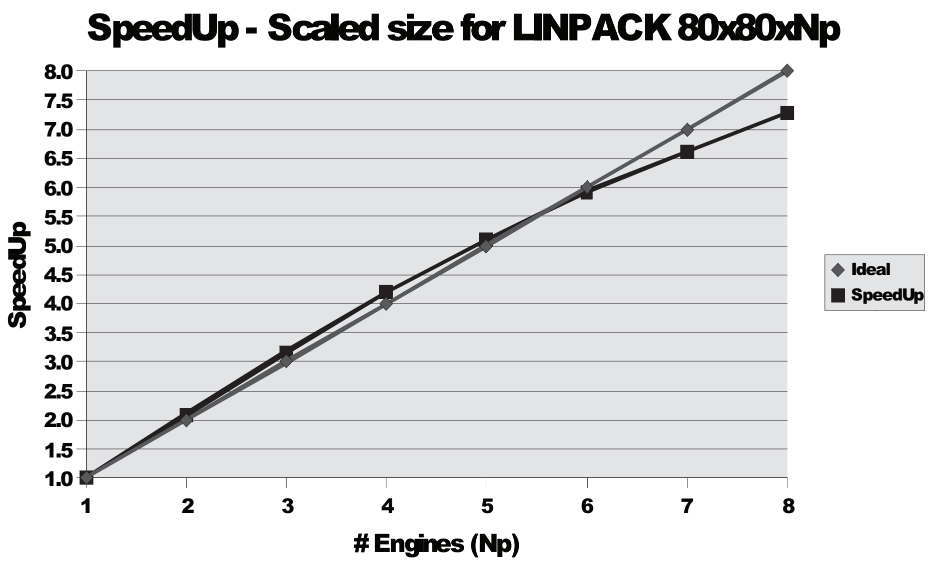

Figure 5.5: Speedup of the LINPACK engine with scaled problem size up to 8 processing units

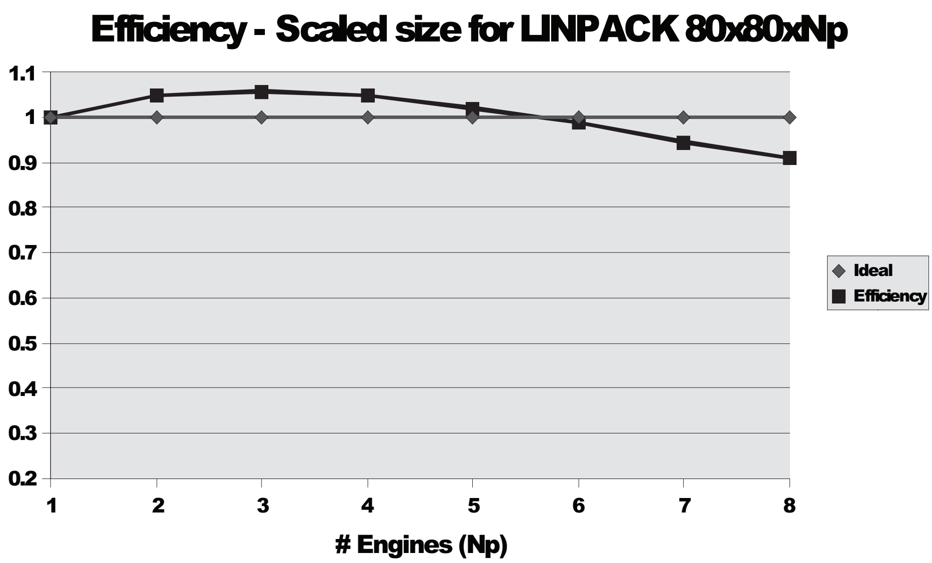

Figure 5.6: Efficiency of the LINPACK engine with scaled problem size up to 8 processing units

rises above the ideal speedup and therefore its efficiency goes over 100%. This can be explained by the increase of the dimensions of the matrix as the number of engines grow. What happens is that to maintain the workload per engine, the dimensions of the matrix are increased every time the number of engines is increased, and consequently the number of columns per engine is reduced. Although with the same workload, the engines will have fewer but larger columns of data to compute. Since the engines compute the data column by column (in the DAXPY loop), having fewer columns to compute reflects in less overhead due to the latency, and since the latency is around 19 cycles (adder + multiplier) per column computed, which is considerable for a matrix this small, having the latency being accounted for fewer times will result in a small improvement in performance. To illustrate this situation, Figure 5.7 shows two different cases where a total of twenty elements are being computed, with two different column organisations. In case 1, there are two columns with ten elements each. Since the pipeline is fed on a column basis, the latency will only be accounted for two times. On the other side, in case 2, four columns of five elements each are computed, and latency affects the computation four times. Although, computing the same amount of data, case 2 suffers twice as much to latency and hence reducing its performance.

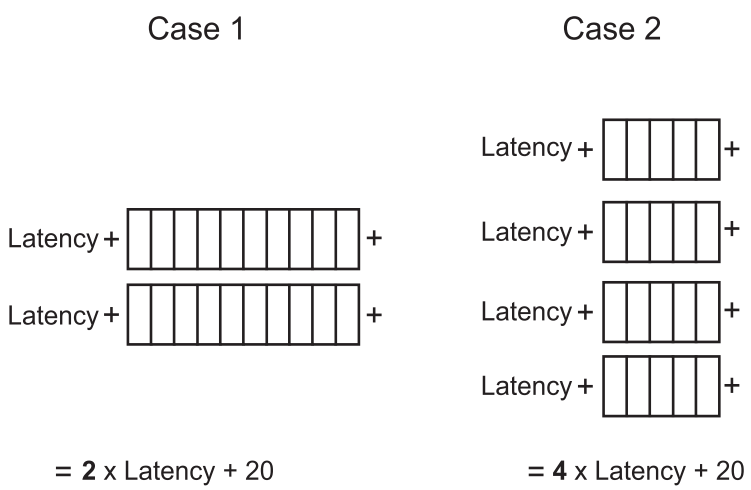

Figure 5.7: Latency effects with different column distribution

By comparing the new results for six engines (628 MFLOPS), or one full FPGA, with the reference processor (324 MFLOPS) for the new matrix size, which for six engines is 195×195, the new speedup, compared with the processor, is 1.94.

5.4 Theoretical Peak

Since it was not possible to get the numbers for larger matrices, due to the lack of memory that was successfully placed and routed for the engines, this section presents the maximum theoretical performance that this engine can achieve. For this purpose, a frequency of 100MHz is considered. From this number, the peak performance for one FPGA (six engines) is extrapolated (and communications are not considered).

To find the maximum performance that can be obtained with one LINPACK hardware engine, the datapath of the engine is analysed and the portion where the most number of pipelined or parallelized floating-point operations is considered (the latency associated to floating-point operations and the time between floating-point operations are not considered). Since the maximum number of floating-point operations being performed at the same stage is two, the theoretical peak performance calculated for the LINPACK hardware engine is 200 MFLOPS. Considering an ideal speedup, a full FPGA (with the XC2VP70 chip) would have a peak performance of 1200 MFLOPS, which would correspond to a speedup of 3.4 when compared to the processor.

However, considering that the FPGA being used in this work is two generations old, a newer device would have a much better chance of achieving higher speedups, assuming more engines could be placed. For example, considering a Virtex 5 LX330 that has 207360 slice LUTs (of 6-input), and that one engine occupies 3830 slice LUTs (synthesized for this chip), about 54 engines could be placed if no other cores are considered (15 was the maximum number of engines for the Virtex 2 Pro chip). By making a rough approximation of the space occupied by the network in the new chip, more than twenty engines would be able to fit in the Virtex 5 LX330, which gives a new peak performance for the engine of 4000 MFLOPS and a speedup, compared to the processor, of 11.4.

Although using a different methodology to calculate the final result, HPL also solves the the linear system problem using the same number of floating-point operations. For this reason and because this represents the maximum performance the processor can achieve to solve this problem, a comparison with HPL is made in this section.

The maximum performance achieved with HPL running in the processor was 911 MFLOPS. If we compare this number with the theoretical maximum performance that an FPGA, using the engines implemented in this thesis, can achieve to the processor the maximum speedup that can be obtained is 1.32 for the XC2VP70 chip and 4.39 with the Virtex 5 chip. This means that it is possible to perform better than a general CPU, however for better performance results several improvements still have to be performed.

5.5 Scaling to Larger Systems

As observed, a single FPGA system can provide performance comparable to a high-end CPU. However, the most interesting characteristic of the LINPACK Benchmark is not to compare single processing units to each other, but large multiprocessor systems (usually with thousands of processors), as shown by the Top500 list. This is where a multi-FPGA system gets its advantage when compared to the multiprocessor system.

The problem is in the network latencies that arise with the creation of a large system. Due to the tighter coupling and lighter protocol used in our multi-FPGA system, there is a better potential to scale the system than in a typical multiprocessor system. Furthermore, considering that the boards used in this thesis have five very tightly coupled FPGAs each, latency can be reduced even more since between FPGAs inside each board the latencies are very low.

When compared to the case of a computing system that uses the model of FPGAs as a co-processors, the chances of scaling are still better in multi-FPGA systems. Although the communications between the processor and the FPGA are fast, the network between the processors is the same as in a typical multiprocessor network, with the same high latencies present in the typical protocols. Moreover, when one FPGA needs to communicate with another FPGA, the data has to go across multiple stages before reaching the receiver FPGA, i.e. FPGA-to-processor-to-processor-to-FPGA. Scalability of a system with this configuration will always be compromised by these factors.

Chapter 6

Conclusions

In this thesis a hardware version of the LINPACK Benchmark is described. This benchmark is used to rank the Top500 list, which is the list that ranks the most powerful supercomputers in the world. The idea is to show a parallel hardware implementation of this benchmark and a look at what else is possible to do in the future.

To implement the LINPACK Hardware Engine, a specific design flow and programming model were used. To communicate between each other, TMD-MPI, a based MPI protocol, was used (TMD-MPE being the actual engine that connects the hardware engines). The addition of this message-passing engine (plus the cores that have to be added for the network to work) occupies a significant amount of space in the FPGA, which could be used to instantiate more engines. The advantage is the great flexibility it provides, since, with this configuration, adding or removing engines to the network is facilitated and no modifications to the code are necessary.

Although, the implementation shown in this thesis does not include DDR memory, which heavily limits its memory range, it is possible to observe that, when the engines present a good computation-to-communication ratio, the results show a more sustained speedup and a better scalability of the system. When compared to a general-purpose processor running the LINPACK benchmark with Level 1 BLAS, for a matrix of 100x100, the FPGA (6 engines) shows a 1.20 speedup and 1.94 if a 195×195 matrix is considered.

By considering that the peak performance for one engine is 200 MFLOPS, the only way of improving performance inside the FPGA is by multiplying the number of engines being used. At the moment, six engines can be placed inside one FPGA giving a 1200 MFLOPS of peak performance, which is 1.32 times faster than a processor running the best known algorithm (HPL) that runs the LINPACK benchmark. Although this does not represent a very significant speedup in terms of performance gain, more can be done to improve the engine and its performance.

Considering a newer FPGA model as a way of achieving better performance with the current engine implementation, the study made in this thesis shows that a maximum of 4000 MFLOPS may be achieved with the LINPACK engine, which represents a 4.39 speedup over the HPL algorithm.

Although the results achieved with this implementation of the LINPACK Benchmark do not show great improvements when compared to a processor, they show that with some modifications to the engine and with a newer technology of FPGAs there is a chance of achieving better results.

Since the LINPACK Benchmark is directed towards large multi-processor systems, scalability is a major factor for this type of system. Because of the tighter coupling and the lighter protocols used in the network present in our multi-FPGA system, scaling to a larger multi-FPGA will not incur in as much overhead as in a typical computer network, and thus increasing the chances of success when implementing a larger multi-FPGA system.

Chapter 7

Future Work

As seen in previous chapters, there is a large room for improvement for the LINPACK Hardware Engine and its communication method. The first point for improvement mentioned in this thesis is its memory range. The present engine makes an exclusive use of BRAMs and logic FPGA fabric as its memory, which strongly limits its ability to store larger matrices with smaller systems. To improve this, DDR memory should be included in the system. This would bring an extra level of control that would have to be aware of the number of engines in the FPGA, because there are not enough DDR memories available for the number of engines present in the FPGA.

The second improvement, which was also mentioned before, is related to the way the engines perform the broadcast and that becomes a bottleneck when smaller matrix sizes are used or when the number of nodes in the system is increased. By using a tree-based broadcast implementation the number of sends can be reduced from n to $log_2(n)$. Further time can, also, saved by parallelizing the MPI_Send with the computation.

The main computation (DAXPY loop) can also be further optimized if matrix-vector operations are used instead of vector-vector operations. Instead of starting a new computation, and losing 19 cycles of latency, every time a vector is processed, the portion of the matrix, present in each engine, can be queued in a way that the 19 cycles of latency would only be processed once, in the beginning of each computation. While this optimization would have a larger impact on smaller matrices, larger matrix sizes would not significantly benefit from this optimization.

More importantly, by replicating the DAXPY flow inside the state-machine of the main engine, it is possible to reduce the amount of overhead present in the initial sequence of the algorithm (IDAMAX and DCSAL functions). Although a single engine would be increased in size (because a smaller BLAS1 Engine and a new set of states would have to be built to perform the new DAXPY flow), each engine would be able to perform twice as many DAXPY functions (which are the main portion of the computation) than before with less hardware than two full engines.

The last improvement would also have a positive impact in the network overhead. Although the number of network components per engine remains the same, since the engines would be performing twice as many DAXPY computations, the network overhead per DAXPY computation would be reduced.

Bibliography

[1] Rob Baxter, Stephen Booth, Mark Bull, Geoff Cawood, James Perry, Mark Parsons, Alan Simpson, Arthur Trew, Andrew McCormick, Graham Smart, Ronnie Smart, Allan Cantle, Richard Chamberlain, and Gildas Genest. Maxwell - a 64 FPGA Supercomputer. In *AHS '07: Proceedings of the Second NASA/ESA Conference on Adaptive Hardware and Systems*, pages 287–294, Washington, DC, USA, 2007. IEEE Computer Society.

[2] Greg Burns, Raja Daoud, and James Vaigl. LAM: An Open Cluster Environment for MPI. In *In Proceedings of Supercomputing Symposium*, pages 379–386, 1994.

[3] Chen Chang, John Wawrzynek, and Robert W. Brodersen. BEE2: A High-End Reconfigurable Computing System. *IEEE Des. Test '05*, 22(2):114–125, 2005.

[4] Mentor Graphics Corporate. `http://www.model.com/`.

[5] Cray. Cray XT4. `http://www.cray.com/products/xt4/`.

[6] D. A. Calahan. Block-oriented local-memory-based linear equation solution on the CRAY-2: Uniprocessor algorithms. In *Proceedings International Conference on Parallel Processing*, pages 375–378. IEEE Computer Society Press, 1986.

[7] J. Dongarra, P. Luszczek, and A. Petitet. The LINPACK Benchmark: Past, Present and Future. `http://www.netlib.org/utk/people/JackDongarra/PAPERS/hpl.pdf`.

[8] J. J. Dongarra, C. B. Moler, J. R. Bunch, and G.W. Stewart. *LINPACK Users' Guide*. SIAM, 1979.

[9] Jack J. Dongarra, Iain S. Duff, Danny C. Sorensen, , and Henk A. van der Vorst. *Numerical Linear Algebra for High-Performance Computers*. SIAM, Philadelphia, PA, USA, 1998.

[10] Richard L. Graham, Timothy S.Woodall, and Jeffrey M. Squyres. Open MPI: A Flexible High Performance MPI. In *In Proceedings, 6th Annual International Conference on Parallel Processing and Applied Mathematics*, Poznan, Poland, 2005.

[11] W. Gropp, E. Lusk, N. Doss, and A. Skjellum. A High-Performance, Portable Implementation of the MPI Message Passing Interface Standard. *Parallel Computing*, 22(6):789–828, 1996.

[12] Chuan He, Guan Qin, Richard E. Ewing, and Wei Zhao. High-Precision BLAS on FPGA-enhanced Computers. In *ERSA*, pages 107–116, 2007.

[13] IBM. `http://www-01.ibm.com/chips/techlib/techlib.nsf/products/PowerPC_405_Embedded_Cores`.

[14] IBM. IBM BlueGene/L. `http://www.research.ibm.com/bluegene/`.

[15] IBM. IBM Roadrunner. `http://www.lanl.gov/roadrunner/`.

[16] C. L. Lawson, R. J. Hanson, D. R. Kincaid, and F. T. Krogh. Basic Linear Algebra Subprograms for Fortran Usage. *ACM Transactions on Mathematical Software (TOMS)*, 5(3):308–323, 1979.

[17] Manuel Saldaña. *A Parallel Programming Model for a Multi-FPGA Multiprocessor Machine, Department of Electrical and Computer Engineering, University of Toronto, Masters Thesis*, 2006.

[18] Hans Meuer, Erich Strohmaier, Jack Dongarra, and Horst Simon. Top500 Supercomputer Sites. `http://www.top500.org/`.

[19] NEC. Earth Simulator System. `http://www.thocp.net/hardware/nec_ess.htm`, 2002.

[20] Daniel Nunes. *A Profiler for a Heterogeneous Multi-Core Multi-FPGA System,Department of Electrical and Computer Engineering, University of Toronto, Masters Thesis*, 2008.

[21] A. Patel, M. Saldaña, C. Comis, P. Chow, C. Madill, and R. Pomès. A Scalable FPGA-based Multiprocessor. In *Proceedings of the 13th Annual IEEE Symposium on Field-Programmable Custom Computing Machines*, California, USA, 2006.

[22] S. Rousseaux, D. Hubaux, P. Guisset, and J. D. Legat. A High Performance FPGA-Based Accelerator for BLAS Library Implementation. In *Proceedings of the Reconfigurable Systems Summer Institute 2007, RSSI 07*, 2007.

[23] M. Saldaña and P. Chow. TMD-MPI: An MPI Implementation for Multiple Processors Across Multiple FPGAs. In *Proceedings of the 16th International Conference on Field-Programmable Logic and Applications*, Madrid, Spain, 2006.

[24] SGI. `http://www.sgi.com/products/rasc/`.

[25] The MPI Forum. MPI: a message passing interface. In *Supercomputing ’93: Proceedings of the 1993 ACM/IEEE conference on Supercomputing*, pages 878–883, New York, NY, USA, 1993. ACM Press.

[26] Sivan Toledo. Locality of Reference in LU Decomposition with Partial Pivoting. *SIAM J. Matrix Anal. Appl.*, 18(4):1065–1081, 1997.

[27] Kieron Turkington, Konstantinos Masselos, George A. Constantinides, and Philip Leong. FPGA Based Acceleration of the Linpack Benchmark: A High Level Code Transformation Approach. In *FPL*, pages 1–6. IEEE, 2006.

[28] Inc. Xilinx. `http://www.xilinx.com`.

[29] Ling Zhuo and Viktor K. Prasanna. Design Tradeoffs for BLAS Operations on Reconfigurable Hardware. In *ICPP '05: Proceedings of the 2005 International Conference on Parallel Processing*, pages 78–86, Washington, DC, USA, 2005. IEEE Computer Society.