HARDWARE/SOFTWARE CO-DESIGN FOR MATRIX INVERSION

HARDWARE/SOFTWARE CO-DESIGN PARA INVERSÃO DE MATRIZES

Maurício Acconcia Dias*

ABSTRACT

Matrix inversion is a well-known widely used operation. Control systems usually demand a high number of matrix operations that need to be performed meeting real-time constraints. Methods that are able to optimize execution time of matrix inversion operations, as the LU decomposition, are good alternatives. Considering this scenario, this work presents a hardware/software profile-based method applied to the LU decomposition algorithm in order to optimize matrix inversion execution time. The optimization method used in this work resulted in a hardware/software co-design whose optimizations are focused on execution time, precision and area. Software and hardware optimization tools were used. Two different soft-processors were used to create the hardware/software co-design systems analyzed in this work, Leon 3 and Altera's Nios II. Nios II processor with hardware custom instructions and a dedicated co-processor achieved better execution times compared to a quadcore Leon-based system suggesting that simple hardware systems are able to achieve better results.

Keywords: Hardware/Software co-design. Matrix Inversion. Reconfigurable Computing. Soft Processor.

RESUMO

A inversão de matrizes é uma operação matemática amplamente utilizada. Sistemas de controle demandam normalmente um alto número de operações com matrizes que precisam ser executadas em tempo real para que o sistema atinja os requisites de tempo. Métodos que são capazes de otimizar a operação de inversão de matrizes, como a decomposição LU, se mostraram boas alternativas. Considerando o cenário apresentado, este trabalho apresenta um método de hardware/software co-design baseado em profiling aplicado ao algoritmo de decomposição LU para otimizar seu tempo de execução. O método de otimização utilizado neste trabalho resultou em um hardware/software co-design cujas otimizações possuem foco no tempo de execução, precisão e área. Ferramentas para otimização e hardware foram utilizadas. Dois diferentes soft-processors foram utilizados para criar os hardware/software co-designs analisados neste trabalho, Leon 3 e Nios II da fabricante Altera. Os melhores tempos de execução foram atingidos pelo processador Nios II com instruções customizadas e um co-processador, em comparação com um sistema quadcore compost por processadores Leon 3. Este resultado sugere que hardwares simples são capazes de atingir bons resultados.

Palavras-chave: Hardware/Software co-design. Inversão de Matriz. Computação Reconfigurável. Soft Processor.

* Universidade de São Paulo (USP) – Laboratório de Robótica Móvel (LRM). macdias@icmc.usp.br

Introduction

Matrix inversion is a well-known operation that is used by several algorithms in different areas: obtaining air gamma-ray spectrum, message encryption, multi-component 3-D imaging of ground penetrating radar data, solve simultaneous equations. Matrix inversion algorithms usually execute the procedure a considerable number of times during execution and this procedure consumes a considerable amount of time to be executed. Hardware and software optimizations can accelerate the execution of matrix inversion algorithms. Before analyzing this work optimizations, the used matrix inversion algorithm is presented.

Matrix inversion is a very important mathematical procedure (PRESS et al, 2002). Considering that matrices are used in a lot of algorithms for representation and calculus, as presented, all basic operations involving matrices should be optimized and available for each one of these algorithms. Matrix addition and subtraction are more direct operations and easier to implement. Most of the problems occur when is necessary to use matrix multiplication, division, and inversion. Many researchers work with matrix multiplication algorithms and this problem is already solved in many ways (BLASER, 2013). The division of two matrices is a very complex problem. Due to the high complexity of the division operation the usual solution is to multiply the matrix by its multiplicative inverse. This operation, for 2 matrices *A* and *B* with size *n x n* is presented by equation 1 and the definition of a multiplicative inverse for a matrix *A* is presented by equation 2 where *I* is the identity.

$$A/B = A * B^{-1} \quad (1)$$

$$A * A^{-1} = A^{-1} * A = I \quad (2)$$

Square matrices that have an inverse are called invertible, otherwise they are called singulars or degenerates. Non-square matrices are not invertible. However, in some cases, these matrices can have a left inverse or a right inverse. Consider a matrix *A* with size *mxn*, if the rank of matrix *A* (that is the dimension of the vector space generated (or spanned) by its columns) is equal to *n*, then matrix *A* has a left inverse. Otherwise, if *A* has rank *m*, then it has a right inverse. Equation *2* presents *3* equal terms; the first term is a matrix multiplication where the inverse matrix is in the right position and in this case the right

inverse should be used. The same procedure should be applied in the second term. A square matrix is singular if and only if its determinant is *0*. Singular matrices are rare in the sense that if you pick a random square matrix, it will almost surely not be singular (PRESS et al, 2002). There are lots of methods that can be used to find the multiplicative inverse of a matrix, in this work the method called LU decomposition was chosen.

1 LU Decomposition

In linear algebra, LU decomposition is a matrix decomposition which computes a matrix as the product of two other matrices: a lower triangular matrix (that has zeros above the main diagonal) and an upper triangular matrix (that has zeros below the main diagonal). The product sometimes includes a permutation matrix as well. LU decomposition is used in numerical analysis to solve systems of linear equations or calculate the determinant of a matrix.

$$A = LU \tag{3}$$

$$\begin{bmatrix} a_{00} & a_{01} & a_{02} \\ a_{10} & a_{11} & a_{12} \\ a_{20} & a_{21} & a_{22} \end{bmatrix} = \begin{bmatrix} l_{00} & 0 & 0 \\ l_{10} & l_{11} & 0 \\ l_{20} & l_{21} & l_{22} \end{bmatrix} \begin{bmatrix} u_{00} & u_{01} & u_{02} \\ 0 & u_{11} & u_{12} \\ 0 & 0 & u_{22} \end{bmatrix} \tag{4}$$

Equations *3* and *4* represents the LU decomposition where *A* is a square matrix, *L* is the lower triangular matrix and *U* is the upper triangular matrix. An invertible matrix admits an LU decomposition if, and only if, all its leading principal minors are different from *0*. The decomposition is unique if the main diagonal of *L* (or *U*) consists only of *1*'s. The matrix has a unique LDU factorization under the same conditions (PRESS et al, 2002). To apply the LU decomposition to invert a matrix, consider the solution of linear equations. Equation *5* presents a system that needs to be solved for *x* given *A* square matrix with size *n x n*.

$$Ax = LUx = b \tag{5}$$

First the equation $Ly = b$ is solved for y then the equation $Ux = y$ is solved for x. It is important to notice that in both cases there are triangular matrices (lower and upper) which can be solved directly using forward and backward substitution. To find the inverse matrix, instead of vector b its considered a matrix B that is a square matrix with size $n \times n$ (Equation 6).

$$AX = LUX = B \tag{6}$$

The same procedure previously presented can be used to solve each column of matrix X. Supposing that in this case matrix B is the identity matrix of size n. It would follow that the result X must be the inverse of A (PRESS et al, 2002).

2 Related Works

Hardware implementations for LU decomposition started with pipelined designs (THIBALT; MULLIN, 1994). Syed (2002) investigated a hardware interpreter for sparse matrix LU factorization. Presented results indicated that his algorithm needs a considerably fast floating-point unity and that the memory access is critical for execution time. Wang and Ziavras (2004) designed a hardware system composed of small processors to solve linear systems. Developed system was configured in an Altera EP20KE FPGA achieving *40*MHz. Execution time achieve *41ms* for a *118 x 118* matrix. Another work with LU decomposition and sparse matrices, by Johnson et. al. (2008), presented a hardware structure that had problems with power and floating-point operations.

One way to efficiently perform large matrix LU decomposition on FPGAs with limited local memory is to block the algorithm. Wu, Duo and Peterson (2010) designed processing blocks for LU decomposition algorithm. With *36* blocks in a Xilinx Virtex-5 xc5vlx330 FPGA proposed hardware achieved *8.50* GFLOPS at *133*MHz. Design the block optimized floating point operations was the main problem of the algorithm.

Benson et. al. designed Gusto (IRTUK et al, 2010) that is an automatic generation and optimization tool for matrix inversion architectures. Method chosen was LU decomposition for matrix inversion and it was compared to other matrix inversion methods achieving the best results. Another possible hardware implementation is using Graphic

Processing Units (GPU). Relatively recent examples are the works of Curry and Skjellum (2006) and Ye (2009).

Based on previous presented works the design of a hardware/software co-design for LU decomposition, applied to matrix inversion problem, is needed because this method is able achieved the faster results compared to other inversion methods and all presented approaches developed solutions *100*\% hardware. Execution time can be better on all-hardware solutions, but the system loses all its flexibility. This flexibility can be achieved using soft-processors with hardware accelerators or multi-core systems. In this case flexibility is important because the inversion is usually used as a step for many methods and algorithms that are rarely implemented in *100*% hardware systems. Next section presents this work proposed method and used tools.

3 Tools and Methods

The main goal of this work is a hardware/software co-design for LU decomposition algorithm applied on matrix inversion. The hardware/software co-design method chosen is a profile-based method. Hardware was designed for Altera Cyclone II FPGA[1] in a DE2-70 terasic board[2]. Altera provides hardware (Quartus II and Qsys)[3] and software (Nios II EDS)[4] design tools. Altera also provides a soft-processor (Nios II) that can be configured on Qsys and programmed on Nios II EDS. All these tools and the profile-based design method are described in following sections.

3.1 Development Tools

Considering flexibility, cost, efficiency, and easy prototyping (BOBDA, 2007) a reconfigurable hardware was chosen for this work's system implementation. Previous experience with Altera devices resulted in the choice for Altera Cyclone II FPGA available in Terasic DE2-70 development board. Despite of the medium performance characteristics

[1] http://www.altera.com/products/devices/cyclone2/cy2-index.jsp
[2] http://www.terasic.com.tw/cgi-bin/page/archive.pl?No=226
[3] http://www.altera.com/literature/lit-index.html
[4] http://www.altera.com/devices/processor/nios2/ni2-index.html

of Cyclone II FPGA, this board has many advantages as many input/output interfaces and enough memory for this work's purpose.

Altera devices are programmed by an integrated development environment (IDE): Quartus II[5]. This IDE has internal environments for hardware design, verification, synthesis, optimization and a fast interface to Qsys[6]. Qsys is another IDE used to configure system interconnections (with or without Nios II soft-processor) using Altera's AVALON[7] memory mapped interface bus.

The first version of the system, implemented in software, used Nios II as a processor. Nios II is fully configurable on Qsys and there are three versions of available which comparison is presented on Table 1. The effects of choosing each one of the three processors is discussed in results section.

After soft-processor configuration and synthesis, Nios II EDS was used for software development. Nios II EDS generates all libraries and drivers to be used in the soft-processors configured using Qsys. After that the software was compiled by a GCC-based compiler that generates the *.elf* executable using designed processor's instruction set.

Table1 – NIOS II Processors Features

	Nios II /e	Nios II /s	Nios II /f
License	Free	Buy	Buy
Jtag Debug	Yes	Yes	Yes
Custom Instructions	256	256	256
Brench Prediction	No	Static	Dynamic
Pipeline	No	5-Stage	6-Stage
Cache	No	Instruction	Data+Instr.
FPU	No	No	No

The advantage of having a GCC-based compiler in this case is that a profiling tool was already designed for it. The Gnu Profiler is a profiling tool that is integrated to GCC compiler and uses intrusive debugging code to measure each line and function execution times. Gnu Profiler also has options that generates different types of logs with flow graphs.

[5] http://www.altera.com/products/software/quartus-ii/web-edition/qts-weindex.html
[6] http://www.altera.com/products/software/quartus-ii/subscription-edition/qsys/qts-qsys.html
[7] http://www.altera.com/literature/manual/mnl_avalon_spec.pdf

Compared to other profiling tools, Gprof proved to be an interesting choice. All implementations for a general purpose processor (GPP) were executed by an Intel Core i3 processor with 3GB of DDR3 RAM using Code::Blocs IDE and MingW compiler on Windows 7. All tools described in this section were used to follow the chosen profile-based method for hardware software co-design.

3.2 Design Method

Figure 1 presents a flowchart of the chosen profile-based method. Two main cycles can be noticed: the first one refers to software development and the second one to hardware design. Sometimes system requirements can be met only with software optimizations. When software optimizations reach their limits without satisfying requirements, hardware design starts. On embedded systems, hardware design is a costly task comparing to software development, so a large amount of time can be saved choosing this method. Together with this fact the final solution can be more interesting considering cost, performance and energy consumption because only the portion of the system that needs acceleration will be implemented in hardware.

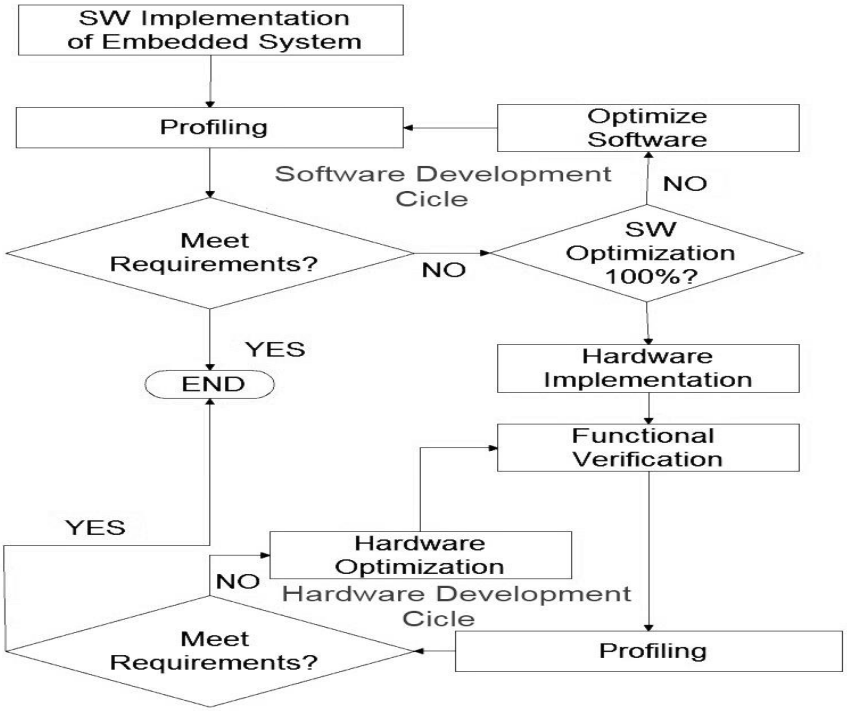

Figure 1 - Modified Profile-Based Method.

Profiling-based methods have an important feature: hardware/software partitioning, that is the hardware/software co-design step where the system will be partitioned in hardware modules and software modules, is done intrinsically. This problem is very important in hardware/software co-design because solutions configure a very large search space and a wrong choice in this step may result in a system that may not respect initial constraints (HUBERT; STABERNACK, 2009).

Proposed method was used to design the Nios II based solution that was used to achieve the main goal of this work. Results were compared to other implementation, a multicore system, that executes the same algorithm, modified to explore the concurrency of the system. Next section presents the other designed system.

3.3 Multicore System

Results of Nios II based hardware/software co-design were compared to a multi-core implementation. A parallel version of the matrix inversion using LU decomposition was executed in a quad-core system of Leon 3 soft-processors. This multicore system was based on the basic implementation suggested in grlib manual[8] for each processor. This multicore system was chosen because parallel versions of algorithms for matrix computations tend to achieve better results and explores the natural parallelism on matrix computations memory accesses (GALLOPOULOS; PHILLIPE; SAMEH, 2016).

```
for k ← 0 to (n-1) do
    /*Element Division*/
    for i ← k+1 to (n-1) do
        a[i][k] ← a[i][k]/a[k][k]
    /*Element Elimination*/
    for i ← k+1 to (n-1) do
        for j ← k+1 to (n-1) do
            a[i][k] ← a[i][j] - a[i][k]*a[k][j]
```

Figure 2 – Parallel Algorithm.

[8] http://www.gaisler.com/products/grlib/grlib.pdf

Parallel version of the algorithm proposed by Michailidis e Margaritis (2010) (Figure 2) needed *3* loops for a parallel execution. For each iteration of the external loop there is a division and elimination of matrix lower and upper elements. Sequential LU decomposition in *k* iteration executed *(n-k-1)* arithmetic operations while parallel algorithm executes *(n-k-1)² * 2*. Assuming that for each arithmetic operation one time unit is consumed, parallel algorithm's execution time is given by Equation *7* that results in a time complexity of *O(n³).*

$$T = \sum_{k=0}^{n-1}(n-k-1) + 2\sum_{k=0}^{n-1}(n-k-1)^2 \qquad (7)$$

Leon 3 processor, whose architecture is presented in Figure 2, is a 32-bit soft-processor compatible to SPARC V8 architecture. Processor is developed to multi-processing systems up to 16 processors that can be implemented in AMP (*Asymmetric Multiprocessing*) and SMP (*Synchronous Multiprocessing*). One interesting feature of this processor is that it offers high performance for low frequencies applications optimizing area and energy consumption together with a floating-point unit already implemented.

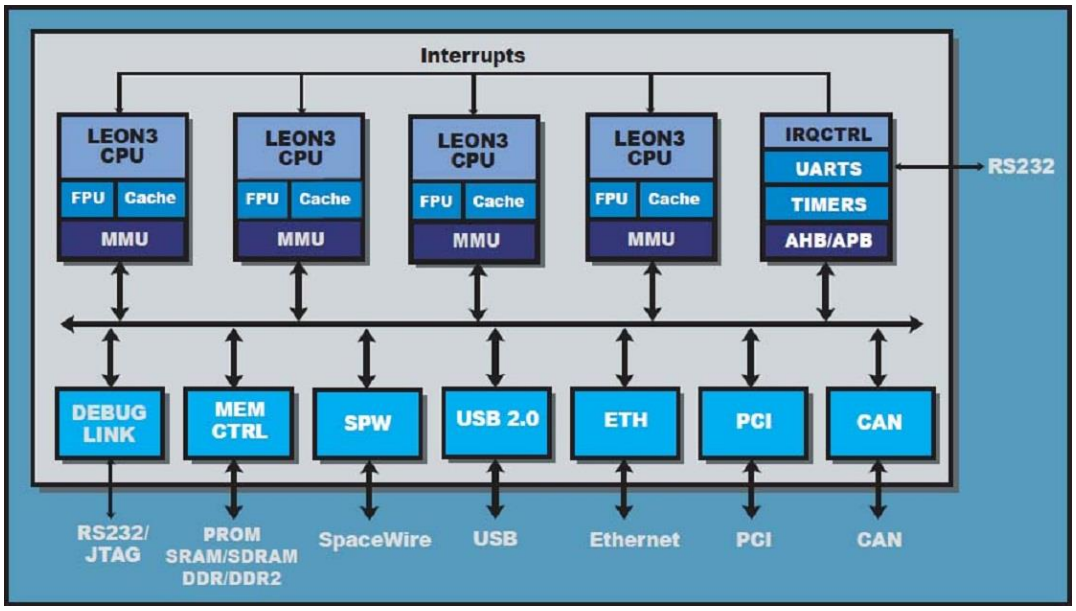

Figure 2 – Leon 3 processor architecture. Available in: http://www.esa.int/var/esa/storage /images/esa_multimedia/images/2012/02/leon3_architecture/14422173-1-eng-GB/LEON3_ architecture_large.jpg

Designed quad-core system (Figure 3) is composed by four Leon *3* processors configured with all components as presented in the processor's diagram[9]. System memory is composed by internal instruction cache memories of *32*Kb and internal data cache memories of *16*Kb together with an external DDR *2* memory of *256*MB in four internal *64*-bit blocks. Processors access external DDR *2* memory using a mutual exclusion policy. Multi-processor control was managed by the real-time operating system (RTOS) eCos[10]. This system was synthesized for a Xilinx ML507 Evaluation Platform[11] that contains a Virtex \$5\$ FPGA. Next section presents the results and analysis.

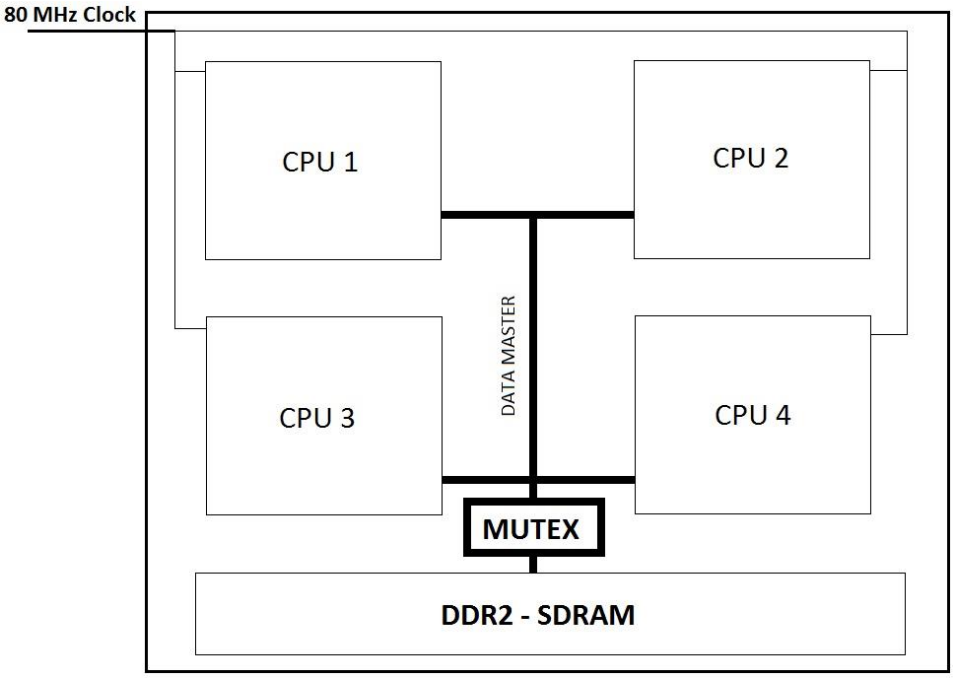

Figure 3 – Quad-core architecture.

4 Results

This section presents a comparison between the results of the two designed systems. Leon *3* CPU, as described in previous section, composed by a *32*kb cache, a floating-point unit, *256* MB of DDR2 SDRAM achieving *80*Mhz clock. Four different versions of Nios II soft-processor were configured: Nios II /f (the pure fast version), Nios II /ff (fast version with

[9] http://www.gaisler.com/doc/leon3_product_sheet.pdf
[10] http://ecos.sourceware.org/
[11] http://www.xilinx.com/products/boards/ml507/docs.htm

a floating-point unit(FPU) without exclusive hardware for floating-point division), Nios II /ffd (full-featured floating-point unit) and Nios II /ffdpll (full-featured floating-point unit and a Phase-Locked Loop(PLL) to increase clock frequency from *50*MHz to *100*MHz). Designed processors without PLL achieved a *50*MHz clock due to the DE2-70[12] board oscillator, and all versions had *90*Kb of on-chip memory. Floating-point unit using in Nios II processors were configured as suggested on Nios II manual[13]. Figure 4 presents the results of the initial experiments with Nios II designed processors and the single-core Leon 3 processor calculating a LU decomposition for a *50 x 50* matrix.

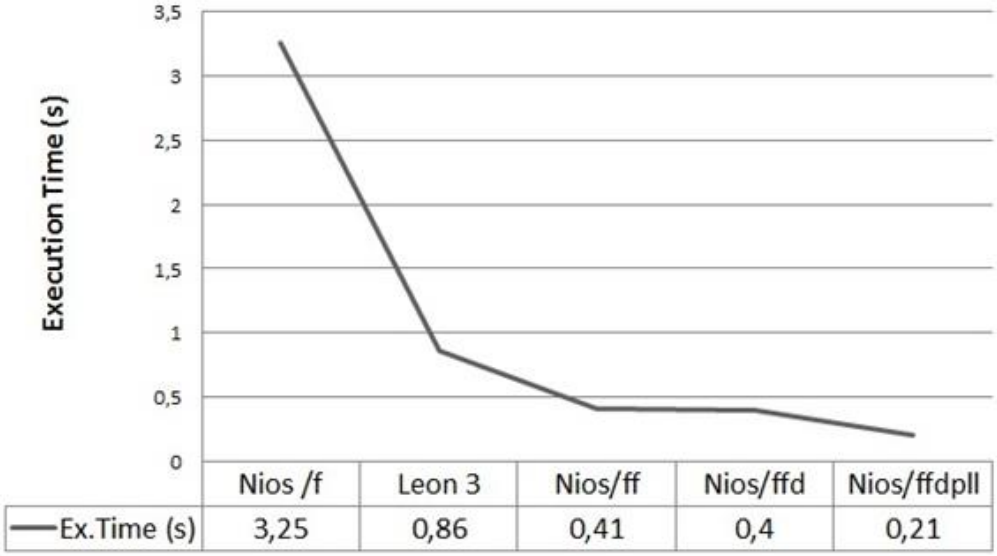

Figure 4 – Initial Results for NIOS II processors.

Nios II processors that included a FPU achieved better execution time when compared to Leon 3 single-core processor, this result is caused by eCos system overhead on Leon *3* and DDR2 SDRAM access time. Nios II had only the HAL as an operating system and no external memory access. These initial execution times were measured using GCC compiler tool Gprof that allowed the execution time of each algorithm line to be analyzed.

Initial designs of Nios II processors included a *90*Kb cache that was not sufficient to store larger matrices. This problem was solved by a SDRAM of *64*MB that was added to Nios II hardware. In this point, after using Gprof to evaluate the algorithm lines execution

[12]http://www.terasic.com.tw/cgi-bin/page/archive.pl?No=226
[13] https://www.altera.com/en_US/pdfs/literature/hb/nios2/n2cpu_nii5v1.pdf

time, the execution time results were obtained using hardware timers. Table 2 presents the results of an *50 x 50* matrix LU decomposition for the following processors: Leon *3* single-core, Nios II /ffdpllram (fast with full-featured floating-point unit, phase-locked loop and SDRAM) and Nios II /ffdpllramon (fast with full-featured floating-point unit, phase-locked loop and SDRAM together with on-chip memory). SDRAM memory controller that was added to Nios II processors caused an overhead on execution time. However, times are lower because there was no Gprof debug directives added to generated code and so execution time tends to be lower.

Table 2 – Execution times for NIOS II processors with SDRAM

Processor	Leon 3	NiosII/ffdpllram	NiosII/ffdpllramon
Exec. Time(s)	0.765	0.185	0.172

Code profiling analysis using Gprof revealed that the most time-consuming routines included floating-point arithmetic operations. This fact explains the difference between Nios II /f and Nios II /ffd results. In this point of the design two alternatives were available: the first was to design specific hardware to accelerate the more time-consuming routines without doing any software changes and the second was to optimize compilation and use fixed-point arithmetic. Developers decided to take the second option first. Fixed-point arithmetic will reduce hardware size because the FPU was no longer necessary. Table 3 presents compiler optimization options results for Nios II processor considering a LU decomposition of a 50 x 50 matrix.

Table 3 – Compiler Optimizations

Optimization Level	off	O1	O2	O3	OS (size)
Execution Time (s)	0.186	0.185	0.186	0.187	0.193

The size optimization achieved the worst results and the final generated code was not reduced significantly. The difference between the optimization levels are the number of optimization directives the compiler uses. Higher matrix sizes achieved better execution times using O1 because only the most significant optimization directives were used. Fixed-point arithmetic was implemented in software by a C++ library. 16-bit fixed-point

representation achieved an acceptable error of *0.0001*, but the execution time was *0.980*s. Execution time results were better than Nios II /f results (2.670s) but unacceptable when compared to processors using FPU execution time (*0.173s*). Cyclone II FPGA logic elements utilization decrease from *15*% to *6*% without the FPU and this fact should be considered in cases that area constraints are more important than execution time. With this code, the software development cycle found an acceptable optimized code.

Hardware development cycle started with the multi-core Leon *3* processors evaluation and the design of a co-processor for Nios II /ffdpllramon. The results for Leon *3* processors with *2,3* and *4* cores executing the LU decomposition for a *50 x 50* matrix (Figure 5). Execution times for *3* and *4* cores are similar because memory access became an issue for more than *3* cores. Compared to Nios II execution times Leon *3* multi-core system results are considerably slow. These bad results are caused by operating system overhead, thread creation overhead, memory access, algorithm parallelization and the system clock of *80*MHz.

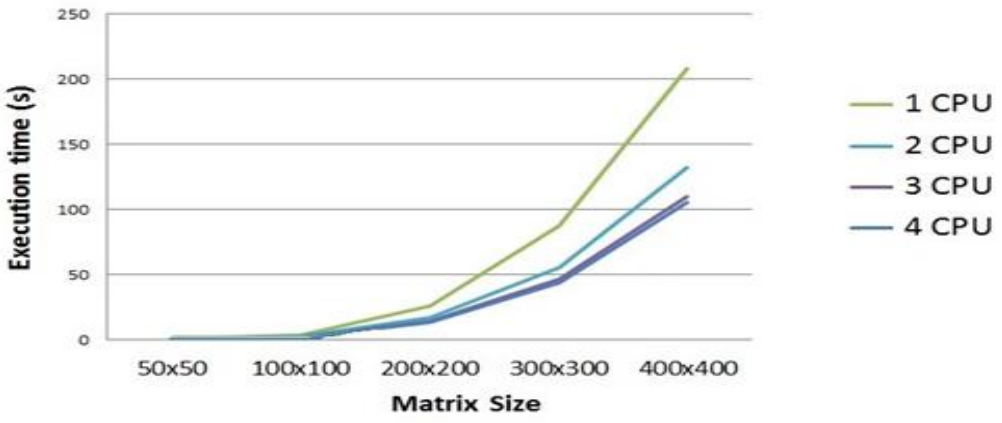

Figure 5 – Multi-core Leon *3* processors results.

Code profiling classified one specific line as the most time-consuming:

*sum += ((j==k)?1.0:matrix[j*maxsize+k])*matrix[k*maxsize+i];*

where *sum* and matrix elements are floating-point numbers. This line executes some floating-point operations to rejoin L and U matrices. Designed co-processor for Nios II /ffdpllramon executes the same operations. To increase hardware optimization level other

three custom instructions were added for index comparisons, index calculation and type conversion (Figure 6).

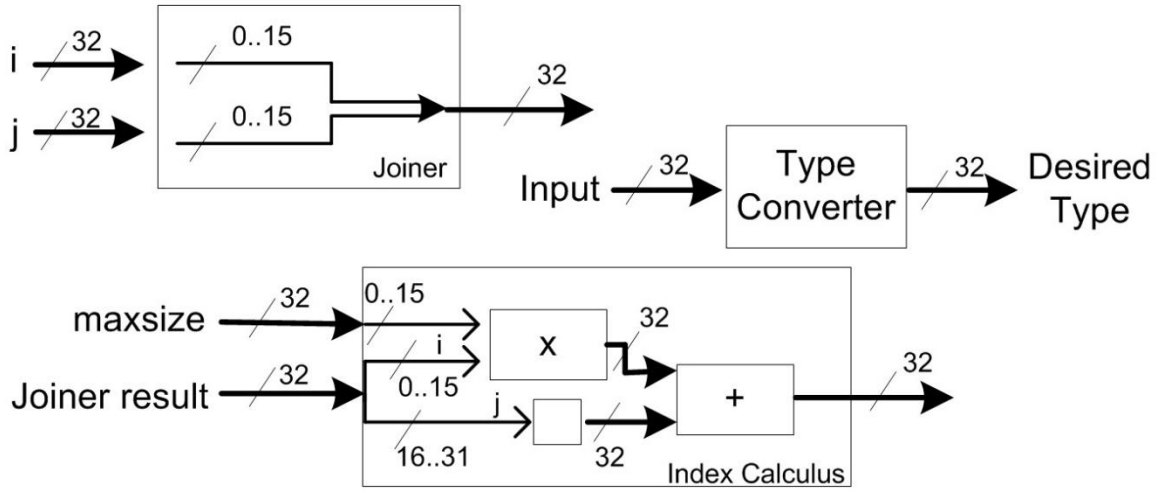

Figure 6 – Custom instructions designed for hardware optimizations

Co-processor's interface to Nios II was composed by parallel input-output (PIO) interfaces of *32* bits (Figure 7). Custom instructions are all combinational with two operands of 32 bits and 32 bits results. All values needed for co-processor's execution (*j*,*k* and *2* matrix values) were sent to the hardware using input parallel interfaces and, after the execution, the result was available in the output parallel interface. Floating-point operations were implemented using Altera megafunctions that are designed in IEEE 754 pattern.

Table 4 – Area consumption for designed processors

System	Leon 3 quadcore	Nios II
Logic Elements	23860	12868

Comparing final systems logic elements consumption, Leon *3* quad-core implementation in a Virtex 5 Xilinx FPGA is compared to Nios II implementation in a Cyclone 2 Altera FPGA. Considering that Xilinx and Altera logic elements design are different this comparison is just an illustrative comparison of area consumption to help developer's choice for one or another design approach. Table 4 presents the results. Consumed area of Leon *3* quad-core system is almost 2x Nios II area with all optimization hardware included.

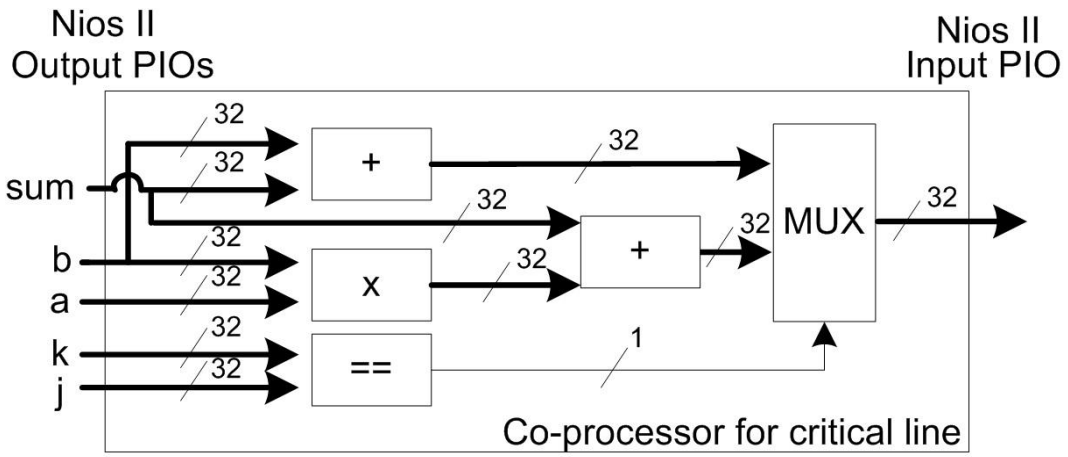

Figure 7 – Designed co-processor.

Table 5 shows execution times for the final architectures. These results suggest that hardware co-processor accelerates execution time for matrix sizes lower than *400 x 400*. For bigger matrices, the execution time increases significantly. This analysis showed that the simpler hardware, only with a floating-point unit as custom instruction, can achieve good execution times. In Table 5 the execution of an LU decomposition by different processors is presented. The processors were: a Leon 3 processor with one core and eCos; Nios II with HAL as operating system only with floating-point unit; Nios II with HAL, FPU and all designed hardware and Nios II with all designed hardware running µC/OS-II[14] to compare to eCos OS. This table suggests that all designed hardware achieved better execution times and that Leon 3 processor, although configured with only one core, is considerably slow.

Figure 8 presents a comparison of the results for each processor and different matrix sizes. Leon 3 processors were slower than all Nios II processors and µC/OS-II overhead is the small difference between Nios II + FPU+ HW + OS and Nios II + FPU+ HW processors. Faster results were achieved by Nios II processor only with FPU unit. The worse execution times of the other two Nios II processors were caused by communication overhead between the processor the hardware optimizations.

[14] http://www.altera.com/literature/lit-index.html

Table 5 – Final results for designed processors

	Leon 3	Nios II	Nios II + HW	Nios II+μC
4x4 (1000x)	0,891	0,172	0,128	0,130
50x50	0,863	0,131	0,170	0,172
100x100	3,67	1,053	1,36	1,39
200x200	26,22	8,792	11,35	11,51
300x300	87,30	30,897	39,49	40,16
400x400	207,58	74,594	94,95	96,37

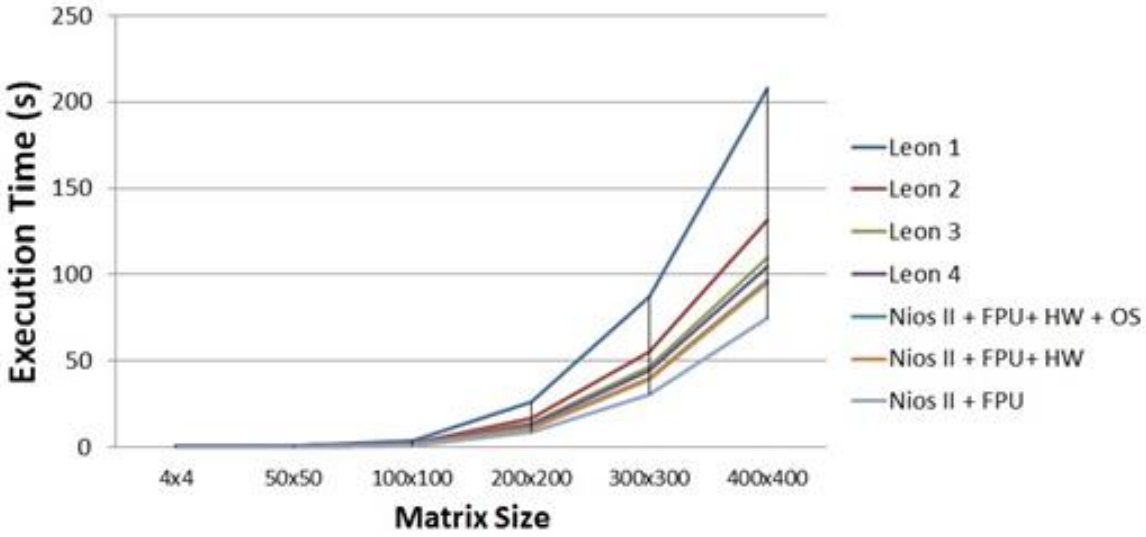

Figure 8 – Final execution time comparison for selected processors.

Conclusion

This work presented a hardware/software co-design for LU decomposition algorithm applied on matrix inversion. Results were compared between different processors: Leon 3 soft-processor (with *1* to *4* cores) and hardware/software co-designs using Nios II processor with hardware co-processors and custom instructions. Nios II hardware/software co-design achieved better execution times, small final system area (as expected) with the same error of *0.00001* magnitude.

The parallelization of the algorithm, the need of a operating system, a floating-point unit that is not pipelined, memory access problems and policies were responsible for the bad Leon 3 quad-core system performance. Nois II results showed that a simpler hardware can achieve good results without consuming design time and respecting the main constraints. Compared to presented previous works, this work results achieved higher execution times but increased the flexibility of the system by using a soft-processor. In this specific case where matrix inversion is only a small part of many systems, flexibility is a key constraint.

Future works leads to hardware development for Leon 3 processors and/or core specialization. With a hybrid quad-core system the execution time could become more acceptable. Other possible experiments can be executed using Leon 3 and Nios II quad-core systems, and between these systems and other system that uses Xilinx MicroBlaze soft-processor.

References

BLASER, M. Fast Matrix Multiplication. **No. 5 in Graduate Surveys**, Theory of Computing Library, 2013.

BOBDA, C. **Introduction to Reconfigurable Computing**: Architectures, Algorithms, and Applications. First Edition, Springer Publishing Company, Incorporated, 2007.

CURRY, M.; SKJELLUM, A. Parallel lu factorization of sparse matrices on fpga-based configurable computing engines: **Research articles. In: Supercomputing '06**. p. 1. John Wiley and Sons Ltd., Tampa, FL, 2006.

GALLOPOULOS, E.; PHILLIPE, B., SAMEH, A. Parallelism in Matrix Computations. No. 1 in **Scientific Computation**, Springer Netherlands, 2016.

HUBERT, H.; STABERNAK, B. Profiling-based hardware/software coexploration for the design of video coding architectures. **IEEE Circuits and Systems for Video Technology** n.19. v. 11, p. 1680-1691, 2009.

IRTURK, A.; BENSON, B.; MIRZAEI, S.; KASTNER, R. Gusto: An automatic generation and optimization tool for matrix inversion architectures. **ACM Trans. Embed. Comput. Syst**. n. 9, p. 1-21, 2010.

JOHNSON, J., CHAGNON, T., VACHRANUKUNKIET, P., NAGVAJARA, P., NWANKPA,C. Sparse lu decomposition using fpga. **Para International Workhop Proceedings**. 2008.

MICHAILIDIS, P.; MARGARITIS, K.. Implementing parallel lu factorization with pipelining on a multicore using openmp. **IEEE CSE** p. 253 –260. 2010.

PRESS, W. H.; TEUKOLSKY, S .A.; VETTERLING, W.T.; FLANNERY, B. P. **Numerical recipes in C.** Second Edition. The art of scientific computing. Cambridge University Press, New York, NY, USA, 1992.

SYED, A. A Hardware Interpreter for Sparse Matrix LU Factorization. Master dissertation in Computer Science **. University of Cincinnati**, 2002.

THIBALT, S.; MULLIN, L. **A pipeline implementation of lu-decomposition on a hypercube** pp. 1–6, 1994.

WANG, X.; ZIAVRAS, S.G. Parallel lu factorization of sparse matrices on fpga-based configurable computing engines: Research articles. **Concurrent Computing : Practical. Experiments**. n. 16, p. 319-343, 2004.

WU, G.; DOU, Y.; PETERSON, G. Blocking lu decomposition for fpgas. **Proceedings of 18th IEEE Annual International Symposium on Field Programmable Custom Computing**. p. 109-112, 2010.

YE, Z. Implementation of LU Decomposition and QR Decomposition on Parallel Processing Systems. **TU/e Electronic Systems**, 2009.