FACULDADE DE ENGENHARIA DA UNIVERSIDADE DO PORTO

Acceleration of a Stereo Navigation Application for Autonomous Vehicles

João Diogo Vilela Teixeira

Master in Electrical and Computer Engineering

Supervisor: José Carlos dos Santos Alves, PhD

Co-supervisor: João Paulo de Castro Canas Ferreira, PhD

September 2011

Resumo

Ao longo dos últimos anos, sistemas de processador embebido têm vindo a ser cada vez mais utilizados em aplicações nos mais diferentes campos. Algumas dessas aplicações são demasiado exigentes em termos temporais para serem executadas unicamente por estes processadores. De forma a aliviar a carga de processamento, blocos de hardware dedicados podem ser acoplados ao processador para auxiliar no processamento de zonas da aplicação que sejam computacionalmente mais intensivas.

Nesta dissertação, o principal objectivo foi acelerar uma aplicação de terceiros, originalmente em linguagem C, preparada a executar num processador convencional. A aplicação a acelerar está direccionada para um sistema de navegação estéreo onde, através do processamento de imagens vindas de duas câmaras, um veículo autónomo poderá inferir a sua posição, rotação e translação relativas. O objectivo de acelerar esta aplicação está inserido no projecto Europeu REFLECT.

Para o desenvolvimento deste trabalho foram identificadas três das funções que consumiam mais tempo relativo de execução da aplicação num processador PowerPC 440 a 400MHz. Utilizando a ferramenta de síntese de alto nível Catapult C, foram criados três módulos de hardware para substituir essas mesmas funções. Os módulos criados foram finalmente implementados numa FPGA da Familia Virtex-5 da Xilinx e executados a 100MHz.

Foi criado um sistema híbrido hardware/software onde as zonas computacionalmente mais criticas foram executadas pelos blocos de hardware criados enquanto o resto da aplicação continuou a ser executado como software. Ao usar estes módulos de hardware para auxiliar a aplicação executada pelo processador PowerPC 440, conseguiu-se uma aceleração de cerca de 1,5 vezes quando comparada com a mesma aplicação executando exclusivamente sob a forma de software no processador.

Abstract

Throughout the years, embedded processor systems are becoming more and more used for applications in the most diverse fields. There are however some applications that are too time demanding to be executed solely by a standalone embedded processor. To relieve some of the processing load, dedicated hardware blocks can be coupled to the processor aiding it by processing the parts of the application that are, computationally speaking, more intensive.

The main objective of this thesis was to hardware accelerate a third party C-language application that was aimed to execute in a general purpose processor. The application to be accelerated is aimed for a stereo navigation system where, through image processing of data coming from two cameras, an autonomous vehicle could infer its relative position, rotation and translation. The objective to accelerate this application is inserted into the European REFLECT project.

In this thesis, the three functions that consumed most of the application's time were identified when executed in an PowerPC 440 processor at 400MHz. Using the high level synthesis tool Catapult C, the corresponding modules that would replace this three functions were created. These modules were synthesized in a Xilinx Virtex-5 FPGA and executed at 100MHz.

It was created an hardware/software hybrid where the most computational intensive functions were carried by the hardware blocks, being the rest left to be carried by the processor. Using these hardware modules to aid the application that was executed by the PowerPC 440 processor, a speed-up of 1,5 times was achieved when compared to the same application being carried solely as software by the processor.

Agradecimentos

Agora que uma etapa importante da minha vida chega ao fim, gostaria de aproveitar esta oportunidade para agradecer a algumas pessoas, sem as quais não seria possível chegar onde cheguei hoje.

Queria em primeiro lugar, agradecer ao meu orientador, professor Dr. José Carlos dos Santos Alves e ao meu co-orientador, Professor Dr. João Paulo de Castro Canas Ferreira, pela oportunidade de trabalhar num projeto tão desafiante e motivador, bem como, toda a sua ajuda dispendida.

Queria também agradecer aos meus colegas da sala I224 do departamento de Engenharia Eletrotécnica e Computadores da Universidade do Porto, que contribuíram para um excelente ambiente: Duarte Azevedo, Helder Campos, Eng. João Gonçalves, João Santos, Nuno Paulino, Pedro Carneiro, Ricardo Pereira, Tiago Campos e especialmente ao Eng. Eduardo Sousa por toda a sua ajuda e partilha de informação no decorrer do trabalho.

Por fim, não poderia deixar de dedicar um agradecimento muito especial à minha família, os principais responsáveis pela pessoa em que me tornei hoje e pelos caminhos que segui para estar onde estou. Ao meu irmão, por toda a sua ajuda e apoio mesmo quando as coisas correm mal quando não deviam. À minha mãe, por ser uma constante fonte de inspiração e a quem dedico este trabalho; por todo o seu apoio e grande espírito de sacrifício, que conseguiu fazer com que nunca nada me faltasse mesmo quando tal não parece possível e sem a qual tenho a certeza que não seria possível chegar tão longe. Ao meu pai, por me ensinar que devo sempre acreditar em mim sobre todas as outras coisas e por ser um modelo inalcançável a seguir.

A todos, o meu muito obrigado.

João Diogo Vilela Teixeira

"The winner ain't the one with the fastest car,
it's the one who refuses to lose."

Dale Earnhardt

Contents

List of Figures

List of Tables

Abbreviations

ASIC	Application-Specific Integrated Circuit
CPU	Central Processing Unit
DPRAM	Dual-Port Random Access Memory
EDK	Embedded Development Kit
ELDK	Embedded Linux Development Kit
FPGA	Field Programmable Gate Array
FPU	Floating-Point Unit
GPU	Graphics Processing Unit
HDL	Hardware description language
HLS	High Level Synthesis
IC	Integrated Circuit
IP	Intellectual Property
ISE	Integrated Synthesis Environment
kB	kiloByte
lsb	Least significant bit
LUT	Look-Up Table
MHz	Mega Hertz
msb	Most significant bit
nm	Nanometer
PAR	Place And Route
PLB	Processor Local Bus
PowerPC	Performance Optimization With Enhanced RISC – Performance Computing
PPC440	PowerPC 440
RAM	Random Access Memory
REFLECT	Rendering FPGAs to Multi-Core Embedded Computing
RISC	Reduced Instruction Set Computing
RTL	Register Transfer Level
XPS	Xilinx Platform Studio
XST	Xilinx Synthesis Technology

Chapter 1

Introduction

With the advances in IC technology, more transistors can be arranged in a smaller area resulting in more and more processing power at lower costs. With this constant evolution, one type of hardware is becoming widely used for prototyping and final product implementation, that is the case of Field Programmable Gate Arrays (FPGA). This type of hardware can be reconfigured by a programmer "in the field" to do a specific functionality, having a flexibility similar to software programming making them preferable to the more static ASIC implementation when facing low volume applications. The possibility that these systems have of being customized allows them to be specialized to perform the operation that they were tailor-made for, and thus makes them able of achieving better performance than conventional computing systems.

A field that is gaining interest is the hardware acceleration of software applications like image processing, data streaming, signal processing and other computational intensive applications. A large community of software developers, usually makes use of C-language based programming to provide the embedded systems with the desired behaviour. However, for cases where performance, timing, power or area constrains are strict, custom hardware to optimize the performance can be the only solution. Two approaches can be followed, depending on the application at hand, to create a system that obeys to the constraints imposed: create a custom hardware device to execute all the application or divide the system in two parts where the most critical parts are handled by dedicated hardware while the rest is executed by software.

1.1 Motivation and objectives

The objective of this thesis is to hardware accelerate a stereo navigation application, making use of reconfigurable hardware modules of a Virtex-5 FPGA, in order to be apt for an efficient use in autonomous vehicles. The stereo navigation application to be accelerated (with C source code provided by Honeywell [2]), is intended to be integrated in an airplane localisation system, in case

that the GNSS[1] used is temporarily unavailable and the plane has to localise itself for some period of time.

The idea of the application is that from independent images derived from two cameras looking in approximately the same direction, dominant entities in the image, invariant to rotation and translation, can be extracted. With the cameras taking the pictures at the same time, this features can be localized in a three dimensional space. Images taken in adjacent instants can be used to calculate the vehicle's rotation and translation. The thesis's work is inserted in the European project REFLECT (Rendering FPGAs to Multi-Core Embedded Computing) [3].

These application is based on image processing which puts major demands on the processors. By making use of dedicate hardware blocks to process the image data, its expected to relieve the processor of the most intensive computations and obtain some degree of acceleration.

The parts of the applications that are consuming most of the time and are delaying the other parts (and the application itself) will be implemented in hardware, while the "lighter" parts will continue to execute in the PowerPC processor present in the development board that will be used during this thesis. With this approach, both software and hardware will be working together for a faster application.

1.2 Work Procedure

The basis of the proceeded work throughout this thesis had, as the main objective, to accelerate the stereo navigation application with as little as possible changes to the original source code ,i.e, no code optimization, no function inlining and no alterations that could, in any way, make the original application execute faster.

The work flow followed through the project can be seen in figure 1.1.

1.3 Thesis Structure

After this chapter, the document is composed of five more chapters. In chapter 2, it is made a study of the State of the Art. The chapter is comprised by hardware acceleration solutions, technology options and the used tools for the development of the project.

Chapter 3 presents an analysis of the application running in the ML507 development board. Composing this chapter is a detailed description of the application itself (functionalities, characteristics *etc.*), how the application was ported (reconfigured in order to run) to the development board and the detection of the slowest parts using some profiling techniques.

In chapter 4 it is made a description of how the hardware modules were developed using Catapult C, as well as, the main dificulties and the solutions found.

Chapter 5 describes how the software-hardware interface was constructed. The required verifications are made along with some critical analysis of the obtained solutions.

[1] Global Navigation Satellite System

Finally, chapter 6 presents a conclusion of the thesis, as well as, some possible considerations for future work.

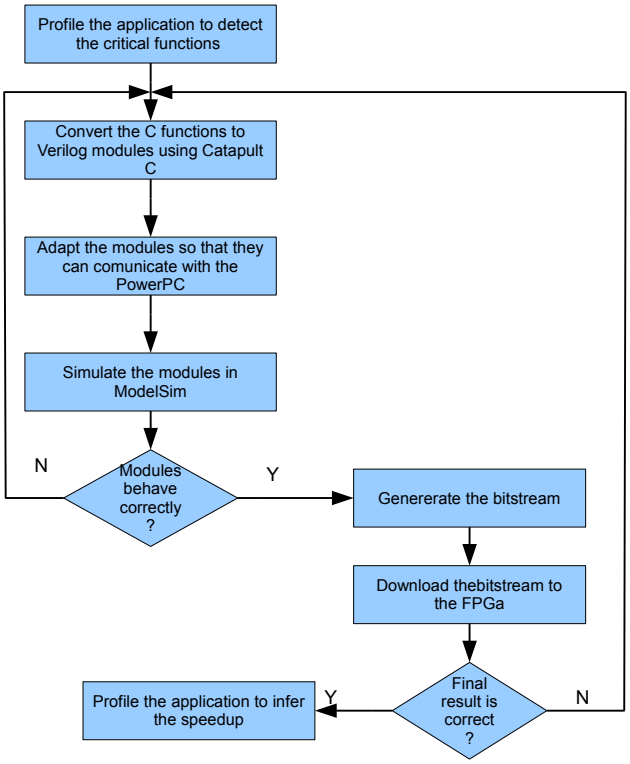

Figure 1.1: Work flow.

Chapter 2

State of the Art

2.1 Hardware Acceleration

Many of today's applications, like data streaming, image processing, etc., are putting higher demands on computing devices. At the same time, production costs and time-to-market targets are getting more and more strict. Having a general computing device isn't, normally, the best choice to process computational intensive algorithms because of the overhead caused by the instruction fetch, decode and execute cycle. General-purpose computing devices are designed to perform a wide variety of tasks with reasonable performance rather than having high performance on specific tasks. These devices are not well suited for applications that aim in executing one specific task with high degree of processing speed. Designing hardware blocks dedicated solely in the execution of these tasks provides a degree of specialization which gives place to higher performance degree.

The two main reasons why hardware can accelerate a given application is: parallelization and data flow direction. By making use of a parallel architecture, multiple operations can be executed at the same time. For instance, by parallelyzing the arithmetic operation $D = (a+b) \times (a-c)$, the addition and subtraction operations can be made at the same time while the multiplication would proceed in the following instant. This would not be possible in a general purpose processing unit where, the same operation would take three cycles (addition, then subtraction and, finally, multiplication) instead of the two cycles used in the hardware alternative. The second reason why an application is accelerated when executed making use of hardware blocks is, that by being able to organize all the functional units, it is possible to direct the data flow such that the maximum number of functional units are operating at the same time.

Applications where hardware acceleration can provide excellent results are the ones that require:

- high processing power requirements ;
- high bandwidth requirements ;
- real-time constraints;
- data formats for which the general purpose CPUs are not optimized;

- hardware interface (example: digital camera interface).

One example where hardware acceleration was used, was the "Hardware acceleration of a Monte Carlo simulation for photodynamic therapy treatment planning". There, a TM-4 board with four FPGAs of the Stratix I family provided the hardware blocks to accelerate a Monte Carlo application based on the Monte Carlo for Multi-Layered media software. The hardware performed the MC simulation, on average, 80 times faster and 45 times more energy efficient than its software counterpart running on 3-GHz Intel Xeon processor[4].

2.1.1 Hardware Alternatives

To provide the required hardware acceleration, two main alternatives can be chosen: either use an Application-Specific Integrated Circuit (ASIC) or a Field-Programmable Gate Array (FPGA).

An ASIC is an IC designed specifically for a particular use, rather than general purpose use. Having the task to be carried well defined, the IC is designed to operate in the most efficient way. By specializing this IC, only the required logic is implemented in the circuit thus eliminating unnecessary transistors giving place to the most energy efficient solution, as well as, the smallest area achievable. Also, hardware blocks with higher degrees of data transaction, can be placed as close as possible, lowering the data transfer times, resulting in lower processing times.

The ASIC implementation provides a natural mechanism for implementing the large amount of parallelism found in many computational intensive applications, according to [5]. However, designing a custom integrated circuit, requires a lot of time from non-recurring engineering. Another discouraging factor of this type of implementation is the mask costs which, according to [6], can get as high as $4 million for an ASIC in a 40 nm process.

Perhaps the main disadvantage of an ASIC that makes them out of reach when designing a prototype system is its lack of flexibility. After being fabricated, the circuit can no longer be altered and, even if the slightest change is made to any part of its circuit, the chip can no longer be used and therefore, a new one has to be redesigned and re-fabricated. To avoid that, hardware designers spend most of the development time simulating the behaviour of the circuit to try and eliminate any flaw. This disadvantages make the ASIC alternative out of reach for prototypes and low volume applications.

Another alternative (and the one used in this project) to hardware accelerate an application is to make use of the reconfigurable characteristics of an FPGA. This ICs are pre-fabricated using the same techniques as the ASICs however, an array of programmable logic blocks interconnected by a programmable routing fabric gives it the ability of being electrically programmed by the designer to implement virtually any digital design. This characteristic makes it almost perfect for the development of prototypes or low volume applications since the flexibility of software and the processing power of hardware can be met at the same time.

According to [1], current FPGAs are SRAM-programmable, meaning that the SRAM bits are "connected" to the configuration points in the FPGA so, programming the SRAM bits configures the FPGA. The way this works is, the SRAM bits are used to turn on or off a pass gate of the programmable routing fabric, this in turn, will allow the signal to flow from one wire to another. Configuration bits can also be used to control computational units, multiplexers, LUTs, etc. To configure this devices, the designer makes use of an Hardware Description Language (HDL) similar to the ones used to define the ASICs functionalities.

Because FPGAs are pre-fabricated by third parties, the hardware designer only has to configure the connections between the required basic blocks to produce the desired system. This eliminates the mask costs that an ASIC with the same functionality would have. Also, because of its reconfigurable characteristics, all the layout can be changed without having to alter the physical characteristic of the FPGA, which results in reduced non-recurring engineering and shorter time-to-market.

The reconfigurable fabric of an FPGA consist of a set of reconfigurable functional units, reconfigurable interconnect (which are used to connect the used functional units among themselves), and a flexible interface to connect the fabric to the rest of the system as stated by [5]. There are some different approaches of the reconfigurable fabrics topologies: fine-grained fabrics (useful for bit-level manipulations), i.e., highly reconfigurable fabrics capable to better adapt to the application specification or, coarse-grained fabrics (useful in datapath applications) that are more efficient and smaller. These approaches can influence the constituents of the FPGA.

Fine-grained functional units perform a single function in a small number of bits. The most abundant example of fine grained functional units are look up tables. By shifting a combination of bits, these units can output any kind of result. This flexibility gives them the power of implement any digital circuit. However, fine grained functional units use significant areas, have significant delays, and have high power consumptions when compared with the coarse grained alternatives. Although less flexible, coarse-grained functional units can eliminate these problems, like the DSP48E (fig.2.1) blocks present in the Xilinx Virtex family FPGAs. These functional units although not as flexible, can perform faster and using less area.

The topology of the reconfigurable interconnects has also some impacts in the efficiency of the FPGA's reconfigurable fabric. In the fine-grained topology (fig.2.2a), every wire of a bus can be switched independently while in the coarse-grained alternative all the bus' wires have to be mutually switched on or off (fig.2.2b). Usually, the reconfigurable interconnects and the functional units go hand-in-hand, i.e., coarse grained units are, normally, connected using coarse grained interconnects. While the fine grained approach provides flexibility, it also has the downside of reconfiguration overhead, since every wire of a bus has to be configured.

Some disadvantages are observed when comparing an FPGA with an ASIC. In the design of an FPGA, the basic logic blocks (AND gates, OR gates, NOT gates, etc.), the harder blocks (multipliers,etc.), as well as all the other components, are already placed in the IC. When the

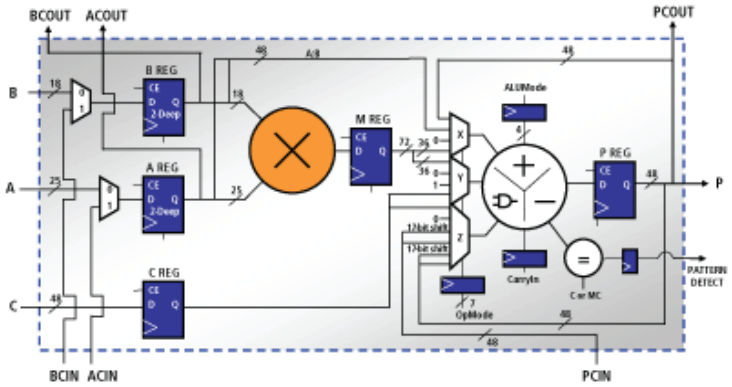

Figure 2.1: DSP48E slice.

designer defines the connections of two components, the routing may not be the best solution. For instance, if an output of a Multiplier is to be connected to an AND logic gate that is very far, because of architecture constraints, a large routing wire (along with all the switching transistors) will be used, causing propagation delays, reducing the overall performance of the circuit. This also gives place to a greater silicon area and power consumptions.

Besides these two, more commonly used hardware alternatives, there are cases where pre-made graphics processing units were used to hardware accelerate applications. For instance, in [7] a GPU was used as a coprocessor to accelerate a Finite-Difference Time-Domain method. Another example is the FFT acceleration processes using a GPU [8].

Through OpenGL and Direct X applications, modern GPUs can become programmable hardware devices, being possible to implement various applications. GPUs can provide the required processing speed because of their high parallel application purpose and floating point data type capabilities. However, when using GPUs, the application has to be modelled to the architecture,

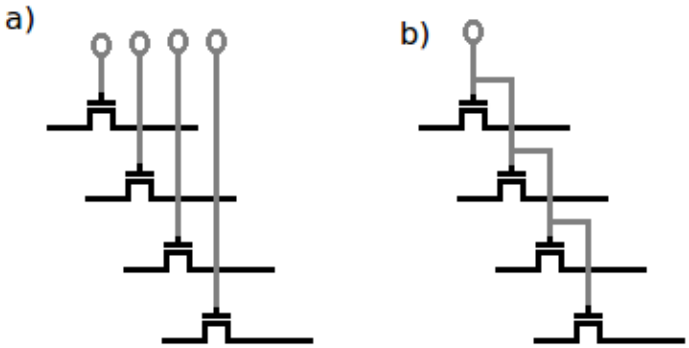

Figure 2.2: Different interconnect configurations. Fine-grained on the left and coarse-grained on the right.

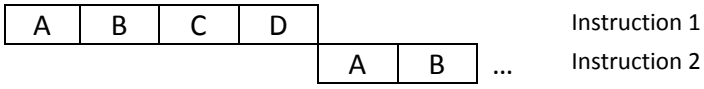

Figure 2.3: Execution of an instruction without pipelining.

instead of generating a specialized architecture.

2.1.2 Hardware Techniques

Some techniques are used when proceeding to hardware acceleration. For Example, by converting the floating-point data representation to fixed-point, the number of functional units are reduced hence reducing the size used in the FPGA. Since the amount of parallelism that can be used in an FPGA is directly proportional to its available logic elements, using fixed-point data types will result in more units available to use in parallelization.

Another technique is the use of look up tables for expensive operations (trigonometric functions, as an example). By storing the already computed values in memory, it is avoided the use of larger computational blocks that would use a large number of elements. This technique also saves FPGA space and can be used to provide hardware acceleration in the same way as the previous technique.

The pipelining technique is widely used for hardware acceleration. This technique consists in dividing the instruction's execution in individual tasks so that each one can be processed by different functional units. As an example, consider the figure 2.3. There, an instruction composed by 4 tasks is being executed in line and, upon execution, the processor will start to execute the next instruction. By pipelining this instruction, every task is executed by a different functional unit, as shown in fig 2.4. In the ideal case, pipelining enables one instruction to be done at every cycle. This however does not mean that the instruction will execute faster.

Since every sub-task is performed independently, more than one task can be performed at the same time. Taking as an example the operation $D = (a+b) \times (a-c)$, it could be divided in two stages: in the first, the addition and subtraction could be proceeded in parallel; the second stage is composed by the multiplication. If the operation was to be made multiple times, the multiplication

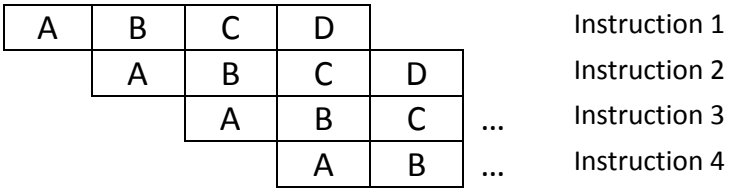

Figure 2.4: Execution of an instruction with pipelining.

of the instruction n could be made at the same time as the addition and subtraction of the instruction n+1.

2.1.3 Architecture Alternatives

There is a significant interest in hardware/software co-design aiming to accelerate a software application where the most time critical parts are implemented in hardware blocks of the FPGA while the parts that cannot be efficiently accelerated are handled by the embedded processor. Normally, complex control sequences such as variable length loop are better implemented in software, while fixed datapath operations obtain better acceleration when mapped to hardware. This can provide high degrees of acceleration. By using an hardware/software hybrid "significant performance improvements, approaching super-computer speeds, can be achieved in a number of application areas, including cryptography, data compression and string matching" [9].

Most of the time, in the software/hardware hybrid, the reconfigurable fabric works as coprocessor to a microprocessor host (this can also be made with a custom circuit). In the reconfigurable logic the most computational intensive parts are handled while, variable length loops and branch control operations that cannot be efficiently mapped to the FPGA are handled by the microprocessor. An inherent advantage of using the reconfigurable fabric as a coprocessor is the speed-up resultant of the parallel computing. This speed up can be characterized by Amdahl's law. The modern version of Amdahl's law states that if a fraction f of a computation is enhanced by a speed-up S, the overall system speed-up is:

$$Speed - up_{enhanced}(f,S) = \frac{1}{(1-f)+\frac{f}{S}} \tag{2.1}$$

This of course, when both the processor and the reconfigurable mesh are executing in parallel. Also, it should be noted the the maximum speed-up achievable depends on the time needed for the sequential fraction of the program. As an example consider an algorithm that takes five minutes to execute and of those five minutes, one cannot be parallelized. Regardless of the parallelization degree, the overall algorithm cannot be executed in less than one minute.

Many computational structures, making use of a standard microprocessors and one or more reconfigurable meshes have been proposed. Some with higher communication speeds between the processor and the mesh, others with higher degree of computing independence. This structures are presented in figure 2.5 and explained with more detail in [1].

In the first case (fig.2.5a) the reconfigurable unit is used within the microprocessor as functional units connected to its main datapath. These functional units can provide the processor the ability of being programmed with custom instructions that can be altered in the future. These functional units require constant supervision of the processor and, therefore, is the alternative with the greatest communication overhead. An example of this kind of architecture is the one present in the Chimaera reconfigurable functional unit, a system design to overcome communication bottleneck

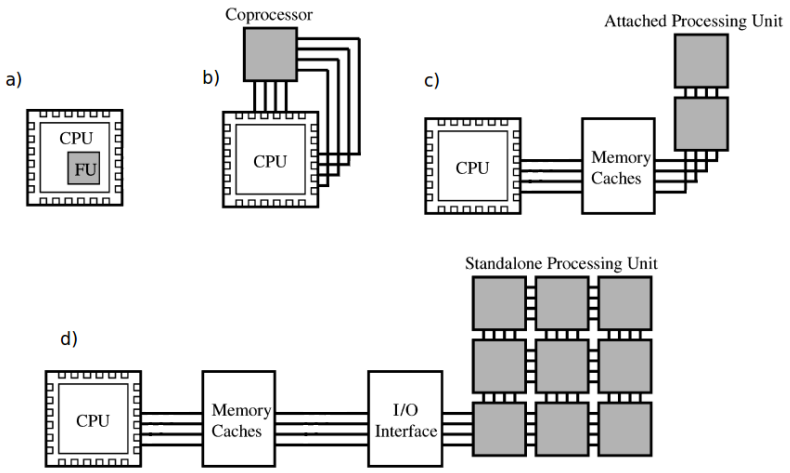

Figure 2.5: Various Processor-FPGA architectures (adapted from[1]).

by integrating reconfigurable logic into the host processor itself [10]. The interconnect structure of Chimaera lends itself well to automatic compilation.

The second case(fig.2.5b), uses the reconfigurable fabric as a coprocessor to which the microprocessor sends data (and controls). The coprocessor executes independently of the processor, i.e., doesn't require constant supervision from it, and after completion returns the results. These kinds of architectures are designed to exploit large amounts of fine-grained parallelism in applications. An architecture similar to this was presented by [11] where both, an FPGA and a reconfigurable multimedia array coprocessor solutions were used to achieve speedups ranging from 2.3 to 7.3 times.

The alternative present in fig.2.5c, uses a reconfigurable mesh as a processor in a multiprocessor system. This implementation provides greater computing independence but at the cost of communication speed.

In the alternative 2.5d the communication speed is even lower but the reconfigurable mesh executes almost as a stand alone computing system.

Other alternatives can be followed. In the COSYMA[12] system, it is attempted to have most of the operations executed by software while hardware is only synthesized for the operations that violate timing constraints. On the other hand, the PAM system, is a reconfigurable hardware coprocessor (making use of an FPGA) tightly coupled to a host workstation [13]. The PRISM[14] approach compiles a program to produce an hardware image used to characterize a reconfigurable platform, and a software image which is similar, in function, to the executable image produced by a conventional compiler consisting of machine code ready to execute in a microprocessor along with code that integrates the synthesized hardware.

2.2 High-Level Synthesis

Using a HDL to develop an hardware design capable of performing complex functionalities is, sometimes, a cumbersome and error-prone task. In fact, the creation of hardware modules from both VHDL and Verilog requires the developer to have in mind the characteristics of the target FPGA, timing constraints, communication delays, etc. For larger projects it is needed a set of tools with a certain degree of abstraction where, only a description of the final implementation's desired behaviour is described.

Because C is one of the most widely used programming languages with high degree of abstraction, and because most of the software to be hardware accelerated is written in C, more and more hardware designers and software programmers are looking at ways to configure hardware by making use of this programming language.

But standard C is not a good choice for designing hardware, given that although it ensures algorithm causality, timing constrains like execution time of a given function or when its input values should be ready to be used are not specified. Since a software algorithm executes in a sequential fashion, failing to meet temporal requisites does not have serious complications (being the only exception in Real-Time systems) and only the execution time of the overall application is affected. However, the same is not true for an hardware module where failing to respect temporal constraints has catastrophic implications such as, errors in the handled data or even shutting the hardware normal functioning.

"Successful hardware synthesis from C seems to involve languages that vaguely resemble C, mostly its syntax"[15]. Some of those C-like hardware languages are:

- **Cones** — Synthesizes a single function to combinational blocks. Supports loops (unrolling them), conditionals and arrays;
- **HardwareC** — Behavioural hardware syntax with C-like syntax ;
- **SystemC** — Set of C++ classes and macros that supports hardware and system modelling;
- **Handel-C** — C variant where every assignment takes exactly one clock cycle;
- **SpecC** — Extension to C with constructor for concurrency, pipelining, etc.;

The main disadvantage of this C-like tools is that their low resemblance to C implies that the programmer has to, effectively, learn a new language. Also, providing software programmers with these tools is not enough to make them good hardware designers since they are used to implement algorithms that run in a sequential manner and, therefore, their ability of designing concurrent tasks may not be well developed. According to [15], software follows a sequential, memory-based execution model derived from Turing machines[1], whereas hardware is fundamentally concurrent. The reason for this comes from the difficult task of designing parallel algorithms, as well as, a disagreement in which are the most efficient ways to provide parallelism to the software's algorithms.

[1] Theoretical device that help to investigate the extent and limitations of what can be computed.

Having these tools a different syntax from standard C, they're also unsuitable when it comes to accelerate functions of a pre-existing C application (or all the application) since, the code would have to be rewritten.

It is the role of the HLS (High Level Synthesis) tools to transform algorithmic level behavioural specification into an HDL capable of synthesize hardware modules that execute the same functionalities as the application (or as the application's function) to be accelerated. This tools also improve the time-to-market since the debug, verification and validation processes of the design are simplified when done at an high abstraction level. By guiding the automated synthesis flow to generate the architecture that meets the desired goals, instead of manually transform the software language to an HDL, it is obtained a gain in development and debugging time.

Some of these tools, generate only the hardware configuration of the system, i.e, when an HLL (High Level Language) is processed by these tools, only the equivalent hardware is inferred. In a software/hardware co-design, the designers will have to adapt the applications software to account for the new generated hardware. There are however tools that enable the designer to specify which parts will be mapped to hardware and which will continue to be executed as software, either by the definition of pragmas or automatically by determining the acceleration gained through the execution of a code fragment in hardware.

According to [16] an HLS tool shall execute the following steps: Compilation, allocation, scheduling, binding and RTL (Register Transfer Level) generation.

In the compilation phase, a series of optimizations such as, loop transformation and dead-code elimination, are made over the algorithm to be synthesized. Also, it is created a formal representation where the control and data flow are defined.

In the second phase, i.e, the allocation phase, the type and number of hardware resources needed to satisfy the design constraints are defined. This components are selected from an RTL library where their power consumption, area and delay must be present.

The scheduling phase is responsible to define when the variables needed by a functional unit shall be present in its inputs. Also in this phase it is specified when the operation should begin its execution.

When there are multiple functional units capable of executing a function, the ones that optimize the final hardware module shall be chosen in the binding phase.

Having all the decisions been made in the previous phases, they can be used to generate an RTL model that respects the design.

One example of these HLS tools is Mentor Graphics's Catapult C. This tool analyses pure and untimed C++ to generate an RTL netlist in Verilog, VHDL and SystemC. There are some reports[17][18] of applications using Catapult to transform C into RTL.

2.3 Tools

A brief description of the tools that were used throughout the thesis' project, will be described in this section.

2.3.1 Virtex-5

To hardware accelerate the application at hand, it was used a Virtex-5 FPGA; more precisely a XC5VFX70T with an embedded PowerPC 440 hard-core processor. The Virtex-5 is an evolution of other FPGAs of the Virtex family, using 65-nm copper process technology and provide a programmable alternative to custom ASIC technology. Some of its common features are [19]:

- 6-LUT + Express Fabric;
- 36Kb Dual-Port Block RAM / FIFO with ECC;
- SelectIO with IDELAY/ODELAY and SerDes;
- 10/100/1000 Mbps Ethernet MAC;
- PCI-Express Endpoint Blocks;
- GTP 3.75 Gbps Transceivers;
- GTX 6.5 Gbps Transceivers;
- PowerPC440 Processors (one in the case of the XC5VFX70T);
- 550 MHz Clock Management;
- 25x18 DSP Slices;
- Integrated System Monitor A/D Converter;
- Advanced Configuration Options.

The Virtex-5 family has also some additional capabilities to extend the functionalities of the PowerPC processor, like crossbar connections or auxiliary processing-unit controllers which can be used to connect the processor to a 128-bit Floating Point Unit which is a soft co-processing unit. This hardware coprocessor is used to address the lack of a floating point unit in the PowerPC processor. Because it is external to the processor, arithmetic operations using floating point datatypes will take longer to execute.

The XC5VFX70T belongs to the FXT sub-family characterized as being high-performance embedded systems with advanced serial connectivity. The main reason for choosing an FPGA of this sub-family instead of any other sub-family (LX for high-performance general logic applications, LXT for high-performance logic with advanced serial connectivity, SXT for High-performance signal processing applications with advanced serial connectivity or TXT for high-performance systems with double density advanced serial connectivity) was justified by the fact

that the FXT sub-family is the only one that contains one (or two) PowerPC 440 RISC Core[20], indispensable to execute the software application.

Perhaps the most important feature of this FPGA are its 128 DSP48E slices that operate at 550MHz[20]. With this, multiple slower operations can be implemented, using time-multiplexing methods. These slices provide improved flexibility and utilization, supporting 40 dynamically controlled operating modes, including multiply, multiply accumulate (MACC), multiply add, three-input add, barrel shift, wide-bus multiplexing, magnitude comparator, bit-wise logic functions, pattern detect, and wide counter.

To be able to work with this FPGA in a easier way, the ML507 (fig 2.6) development board was used. This development board, provides a wide set of features, which saves much of the development time if one would have to manually configure the controlling components. Some of those are [21]:

- **XC5VFX70T-1FFG1136** — Virtex-5 FPGA;
- **Two Xilinx XCF32P Platform Flash PROMs** — for storing large device configurations (32 Mb each);
- **Xilinx System ACE CompactFlash configuration controller** — used to load the FPGA's configuration, as well as, the embedded Linux operating system.
- **64-bit wide, 256-MB DDR2 small outline DIMM**
- **10/100/1000 tri-speed Ethernet PHY transceiver and RJ-45 with support for MII, GMII, RGMII, and SGMII Ethernet PHY interfaces** — Used in the project to send commands and transfer executable files via Ethernet.

2.3.2 PowerPC 440

The PowerPC 440 (PPC440) is a processor of the IBM's PowerPC 400 family. These are 32-bits embedded RISC processors that are built using the Power Architecture technology. These processors are designed to fit inside dedicated circuits like ASICs, microcontrollers and FPGAs.

Having first appeared in 1999, the PPC440 core is a high-performance, low-power consumption engine that implements the flexible and powerful Book-E Enhanced PowerPC Architecture. Embedded in the XC5VFX70T FPGA, is capable of up to 550 MHz operation, 1000 DMIPS and Out-of-order execution[20].

Some of the main features of this processor are[22]:

- High performance, dual-issue, superscalar[2] 32-bit RISC CPU;
- Separate 32 kB instruction and data L1 caches;

[2]Replication of some components built in order to allow more than one instruction to be executed in each cycle

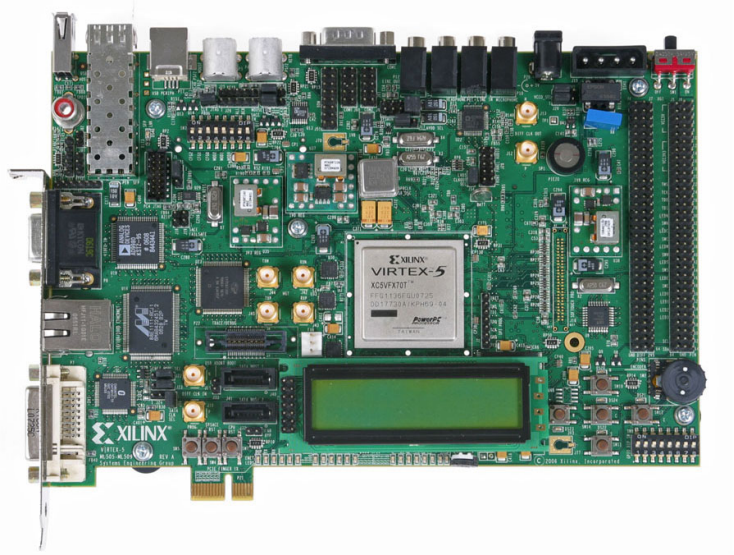

Figure 2.6: ML507 development board.

- Instruction cache parity;
- Data cache parity;
- D-cache full-line flush capability;
- Memory management unit;
- UTLB parity;
- Multiple core Interfaces defined by the IBM CoreConnect on-chip system architecture.

This processor has also a PLB(Processor Logic Bus) interface of 128 bits and is fully compatible with the IBM CoreConnect on-chip system architecture, providing the framework to efficiently support system-on-a-chip (SOC) designs[23]. The PLB interface serves as a mean of communication with all the peripherals external to it. In the case of this thesis, the PLB will be mostly used for communicating with the hardware modules of the FPGA.

To simplify on-chip devices attachments, the core contains various interfaces[22], namely:

- Processor local bus (PLB);
- Device configuration register (DCR) interface;
- Auxiliary processor unit (APU) port;

- JTAG, debug, and trace ports;
- Interrupt interface;
- Clock and power management interface.

The PPC440 includes a seven-stage pipelined PowerPC processor, which consists of a three-stage, dual-issue instruction fetch and decode unit with attached branch unit, together with three independent, four-stage pipelines for complex integer, simple integer, and load/store operations, respectively. It also includes a memory management unit (MMU), separate instruction and data cache units, JTAG, debug, trace logic, and timer facilities.

This processor is available in the ML507 development board used in this thesis. It's aim will be to execute the software part of the application. Also, through the PLB, it will send the data to the FPGA's RAMs, as well as the, the handshake signal to control the hardware modules.

2.3.3 Catapult C

Catapult C Synthesis is an algorithmic synthesis tool that outputs RTL (Register-Transfer-Level), netlists (VHDL, Verilog, and SystemC), simulation scripts, schematics and reports from C++ working specifications. This was the tool that was used throughout this thesis to convert some of the functions of the application that were initially in C to Synthesizable hardware. The strongest feature of this tool is it's ability to accept untimed C++ algorithms to be synthesized to RTL. Typically, the C++ code from a system designer can be used to generate correct results that meet latency and timing constraints, having however, to be made some code changes to meet area and power goals. Besides the C++ to hardware compilation, Catapult C also provides:

- **C++ compiler and file editing** — This compiler has more complete checking than a standard C++ compiler;
- **Algorithm and architecture analysis** — To analyse various algorithm parameters such loop size, variable size, etc.;
- **Micro-architecture constraints** — Loop unrolling, pipeline, memory access,etc.;
- **Optimization and RTL hardware generation** — For better area and/or latency solutions;
- **SystemC Verification Flow** — Automates the generation of a SystemC testbench;
- **Integrated tool flows**— For power analysis, formal verification and source code linting.

The synthesis process flow to create an hardware module using Catapult C can be seen in figure 2.7. The two leftmost blocks are the action that are not performed in Catapult C.

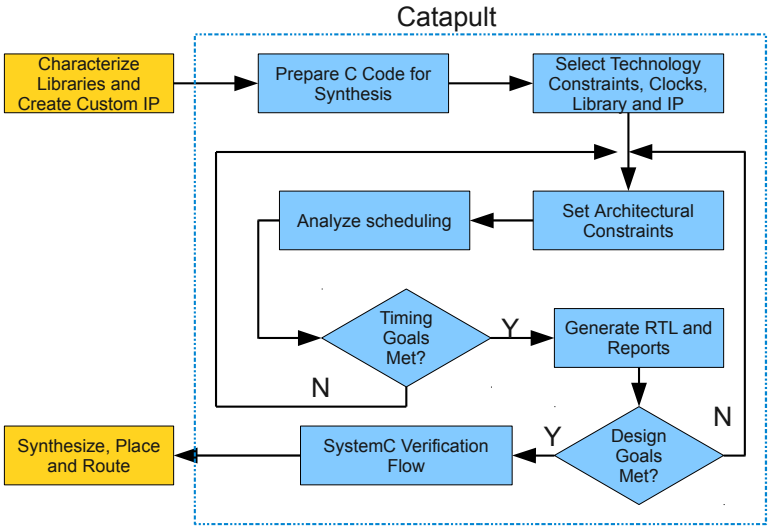

Figure 2.7: Catapult C process flow.

In addition, the integrated SystemC verification flow automates the generation of a SystemC testbench which allows to verify that the C++ design matches the resulting hardware. All these automated processes can increase productivity significantly over standard RTL writing.

Catapult C works in a Solution basis, meaning that changes made to the source code, architecture constraints etc., will be branched to different solutions so that all the alternative implementations are available for comparison.

2.3.4 Other Tools

- Gprof

The hardware acceleration process is best achieved if the most time consuming and computationally critical functions are identified. To that end, the profiling tool Gprof was used.

Gprof analyses the execution time of each and every function of a software application. This analysis is proceeded in the actual execution of the program, using a sampling process. Afterwards, two tables can be viewed where both the total and relative time spent by a function (both by itself and all it's children), the number of time it was called and functions' hierarchy is shown.

- Precision RTL

Precision RTL is a comprehensive toolsuite, providing design capture in the form of VHDL, Verilog and SystemVerilog entry, advanced register-transfer-level logic synthesis, constraint-based optimization, timing analysis, schematic viewing, and encapsulated place-and-route. This tool was mainly used because the Verilog files that were generated by Catapult C, were in too high level to be recognizable by XILINX ISE, i.e., it contained statements not recognized by XILINX ISE.

Precision RTL was needed to synthesize the gate level specification to be effectively used by the ISE.

- Xilinx ISE

Development environment for synthesis and analysis of HDL designs. ISE (Integrated Synthesis Environment) also enables to perform timing analysis and simulation of the synthesized hardware when faced to external stimulus.

The Verilog files created by Precision RTL were passed to ISE where they were integrated in a more complex core (in Verilog also) to create a netlist file that would describe the FPGA logic blocks and connections.

- Modelsim

ModelSim is an advanced simulation and debugging tool. Either Verilog, VHDL or system C languages are accepted to be simulated.

This tool was used to check if all the hardware modules to be synthesized, were performing the required functionalities. By creating a testbench file containing input stimulus, they could be then applied to the modules (in simulation) to analyse both their internal and external logic values.

- Xilinx XPS

The XPS (Xilinx Platform Studio) enables an easier integration of the designed hardware blocks with the embedded PowerPC processor and all the other constituents of the FPGA(serial ports, Ethernet ports etc.).

Chapter 3

Analysis of the Stereo Navigation Application

In this chapter, the Stereo Navigation application will be analysed to determine which parts of the source code can be replaced by dedicated hardware blocks to obtain an acceleration in its execution time. A description of the application will be made, followed by the steps made in order to port the application's algorithm to the targeted development board. The chapter is finalized with an analysis of the application's most critical parts.

3.1 Application Overview

The application to be accelerated, the Stereo Navigation application will analyse a stream of images, in pairs, coming from two cameras. The algorithm analyses one image from each camera taken at the same time by:

- **Rectifying the image** — To remove distortion in each image caused by the lenses of the camera. One such example is the barrel distortion caused by the lenses curvature, present in figure 3.1 (taken from [24]);
- **Extracting features** — To find significant elements in the images and describe them using an hash-like information vector. Here, the Harris corner detector is used;
- **Matching features** — Compares the features extracted in a pair of images at a given moment with the features of another pair of images analysed in the preceding moment. This step is used to increase the probability of a correct 3D reprojection;
- **Making a 3D reprojection** — Derive 3D coordinates of a point from its image projections;

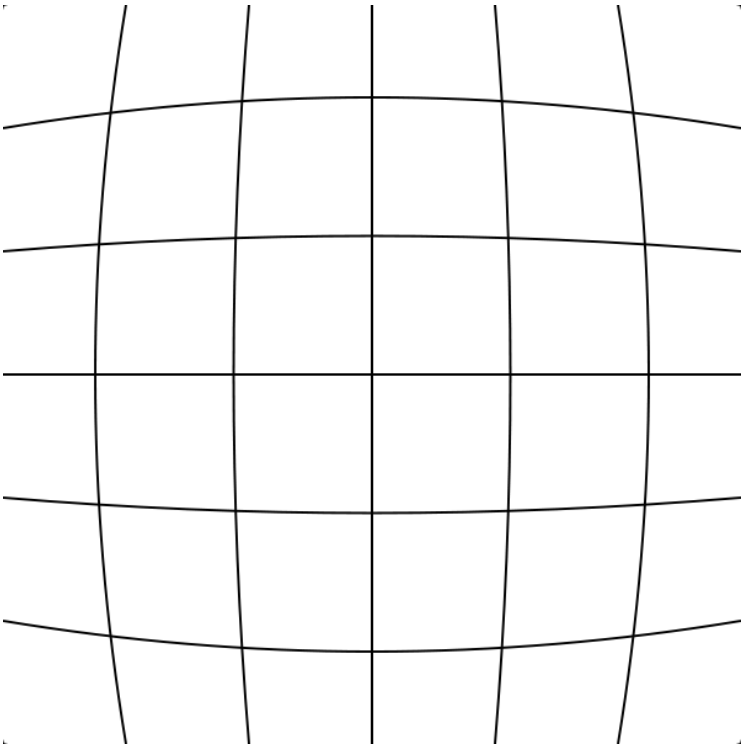

Figure 3.1: Example of image distortion caused by the camera lenses. In this case the Barrel distortion.

- **Robust pose estimation** — To reliably remove members that were matched incorrectly and to produce the estimate of the transformation between the two sensor reference frames, allowing to detect motion.

By analysing pairs of pictures and comparing, in consecutive pairs, how the features of the pictures have changed, the system can also calculate translation, rotation, speed, and position of the vehicle.

In figure 3.2, the two top images are taken at the same time t1 by the left and right cameras. The bottom images are taken also by the same two cameras in the next frame t2 to determine the vehicle position and movement. However in theory it could be possible to infer the position of a feature in a three-dimensional space using only a pair of images, the probability of having a correct match in a cluttered urban environment is often around 20% as stated by [24]. To raise the correct match probability, it is used a circular check where, the features detected by the two cameras are compared with themselves and with the ones of the previous capture instant (the two left images are matched with each other, being done the same to the two right images). This circular check is presented in figure 3.2. With this, the probability of a correct assignment is increased.

The version that was used in this thesis was, however, a little different in the way that the system did not have the two cameras installed. To simulate the camera input, two options were

Figure 3.2: Detection of an obstacle, as well as, speed, translation and rotation of the vehicle by the application.

possible: either the images in grayscale were stored in a file where the application would access them or, a representation of the images was already present in the source code as data. The second option was chosen. In this option, the image is represented by a matrix whose elements's values represent a pixel representation in 8-bit grayscale format, being that the value 0 represents white, the 255 black and all the values in the middle represent shades of gray.

3.2 Porting the Application

In a first approach, there was the need to execute the original Stereo Navigation application in the embedded processor of the ML507 board. This first step was of the utmost importance, in the matter that it gave crucial data that was useful to, in the later stages, infer the degree of acceleration obtained. The most critical information considered was:

- **Execution time of each function** — To determine which are the functions that take the most time to execute and, therefore, are slowing down the overall application.
- **Execution time of the overall algorithm** — In order to be able to tell the amount of acceleration that was achieved.
- **Output of the application** — To ensure that the solution found with the hardware acceleration is working correctly.

To make it easy to integrate the application's source code into the embedded processor, it was chosen to install a Linux kernel in the development board. The reason for that, is because the interface between the software, the CPU, the memory and other devices is not specified by the program.

There was the possibility of executing the application in a standalone fashion. Using XPS, it was possible to import the application to run in the PowerPC processor however, some changes

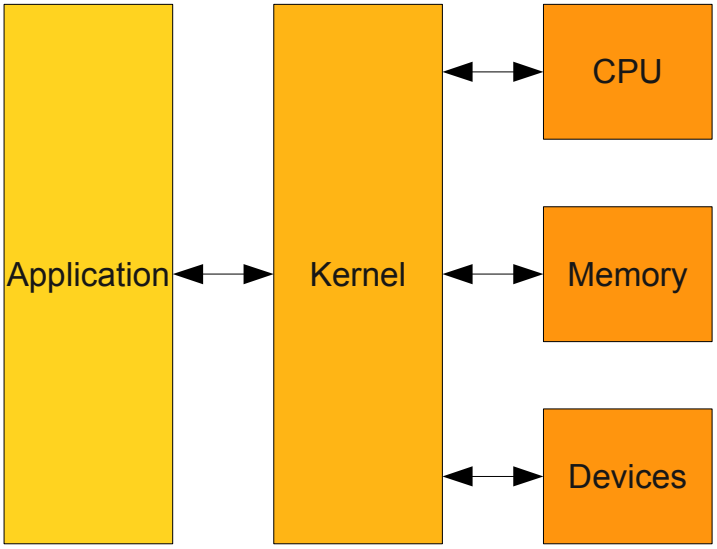

Figure 3.3: Kernel making the bridge between hardware and software.

to the code had to be made which could affect the final execution time of the application. The biggest disadvantage of using this alternative was, that the profiling tool gprof could not be used to determine the execution time of each function.

The use of an operating system eases the software development by providing a series of convenient services like access to a (RAM-based) disk file system, remote network access, virtual memory management and also simplified procedures for profiling the application to estimate the execution time of each function. These advantages far exceed the disadvantage of the overhead caused by the operating system in the processor.

3.2.1 Installing the Kernel

As said before the Kernel makes the bridge in a machine between the software running on it and its hardware. For that reason, and since the architecture of the processor and the other devices present in the development board are not the same as the ones present in a standard computer, a version of the Linux Kernel aiming the PowerPC 440 processor had to be installed. This kernel was obtained from [25] and the instructions present there were followed.

The first step was to configure the reference design in XPS. The reference design is a set of files recognized by XPS that represents all the components present in a given development board (processor, LEDs, serial port, USB port, user defined hardware modules, etc.) and how they are interconnected in the FPGA to the processor by being assigned to each an address code. Every time a given core is needed to execute a specified action, the required data is sent to its address via the PLB (Processor Local Bus). An example of how the cores are connected to the processor in the reference design is shown in figure 3.4.

Instance	Name	Base Address	High Address	Size	Bus Interface(s)
my_ip_0	C_BASEADDR	0xcf200000	0xcf20ffff	64K	SPLB
plb_v46_0	C_BASEADDR			U	Not Applicable
xps_bram_if_cntlr_1	C_BASEADDR	0xffff0000	0xffffffff	64K	SPLB
Push_Buttons_5Bit	C_BASEADDR	0x81400000	0x8140ffff	64K	SPLB
LEDs_Positions	C_BASEADDR	0x81420000	0x8142ffff	64K	SPLB
LEDs_8Bit	C_BASEADDR	0x81440000	0x8144ffff	64K	SPLB
DIP_Switches_8Bit	C_BASEADDR	0x81460000	0x8146ffff	64K	SPLB
IIC_EEPROM	C_BASEADDR	0x81600000	0x8160ffff	64K	SPLB
xps_intc_0	C_BASEADDR	0x81800000	0x8180ffff	64K	SPLB
Hard_Ethernet_MAC	C_BASEADDR	0x81c00000	0x81c0ffff	64K	SPLB
SysACE_CompactFlash	C_BASEADDR	0x83600000	0x8360ffff	64K	SPLB
xps_timebase_wdt_1	C_BASEADDR	0x83a00000	0x83a0ffff	64K	SPLB
xps_timer_1	C_BASEADDR	0x83c00000	0x83c0ffff	64K	SPLB
RS232_Uart_1	C_BASEADDR	0x83e00000	0x83e0ffff	64K	SPLB
ppc440_0	C_IDCR_BASEADDR	0b0000000000	0b0011111111	256	Not Connected
my_ip_0	C_MEM0_BASEADDR	0x85000000	0x850fffff	1M	SPLB
FLASH	C_MEM0_BASEADDR	0x86000000	0x87ffffff	32M	SPLB
DDR2_SDRAM	C_MEM_BASEADDR	0x00000000	0x1fffffff	512M	PPC440MC
ppc440_0	C_SPLB0_RNG_MC_BASEAD...			U	Not Connected
ppc440_0	C_SPLB1_RNG_MC_BASEAD...			U	Not Connected

Figure 3.4: Example of the addresses given by XPS to each of the cores.

A device tree was added to the design which, according to [26], is scanned during the boot phase by the Linux Kernel to build an internal representation that will be used at run time to give the information about the device. In other words, this will describe the hardware to the Kernel by giving names to the components (like ttyS0 for serial port 0 or eth1 for the Ethernet port 1, for instance), describing their functions, etc. The changes made were then processed by XPS to generate a bitstream file. This file has the informations of the layout of all the IP Cores and is used to configure the FPGA at the startup.

When the Kernel was, finally, operational in the development board, it could be accessed via telnet[1]; by inputting commands in the development host's console they were sent to the ML507 board. With this, the development board could be used to execute the application, but first it would have to be compiled to a type of executable that the development PowerPC processor could recognize.

For a more detailed explanation of how the Kernel was compiled to the target board, refer to the appendix A.5.

3.2.2 Cross-Compiling the Application

The C-language files are, in fact, just text files which describe in a structured and well defined manner the behaviour of the desired executable file. This files are intended for readability by the user/programmer and not to be executed by the processor. It is the task of the compiler to interpret the source code and translate it to machine code, producing a file that can be executed.

When writing a code, there are some functions that are widely used by the programmers. To make the writing of the code easier, some functions are already present in the C standard library. So, if a programmer wants to make an application that, for instance, calculates the tenth power of

[1] Network protocol based on bidirectional interactive text-oriented communication.

the number two, instead of having to make himself a function that multiplies the number two, ten times, the function pow() could be used. This facilitates the work and, also, eliminates the chances of more complex functions to have errors or bugs.

Different platforms run different types of executable files. A C-language program compiled to run in a platform may not work in a different one. The reasons for this are the different endianness, word size, word alignment, etc. that each architecture has. Normally a computing device has a compiler in order to build an executable file that respects its architecture constraints.

Because of the limited resources that the used embedded device has, having a compiler installed in it would consume too much memory space and, for that reason, most embedded devices do not contain any compiler. To work around this problem, a cross-compiler is used which compiles the C-code in the cross-development host aiming for the target device. In this case, a general purpose computer running a Linux distribution was used as a cross-development host to compile the source code in order to build executable files that could, effectively, run in the development board's processor. To create an executable file, capable of running on an embedded system, the editor of the cross-development host is used to create/alter the source code, the debugger is used to check for errors, and the cross-compiler to create the executable file. After that, the executable file is ported to the embedded system where it will be run.

The standard library of functions is not present in the development board, due to its limited resources also. Instead, the libraries used are the ones present in the cross development host. To include the standard library's files in the final executable, the *-static* flag must be set when compiling. This command instructs the compiler to copy all the functions used by the application directly to the executable from the standard library during the linking phase. This makes an executable file bigger in size but relieves the embedded system to have all the libraries present. On the other hand, in the dynamic linking alternative, the executable file references only the library from which to get the used function so, they would have to be present somewhere in the development board, using memory space.

The cross-compiler used was the powerpc-linux-gcc contained in the ELDK 4.2 (Embedded Linux Development Kit) package. According to [27], the provided package includes the GNU cross development tools, such as the compilers, binutils, gdb, etc., and a number of pre-built target tools and libraries necessary to provide some functionality on the target system. To cross-compile the application using the powerpc-linux-gcc cross compiler the following command had to be imputed in the console:

powerpc-linux-gcc -static nameOfSourceFile.c -o nameOfExecutableFile

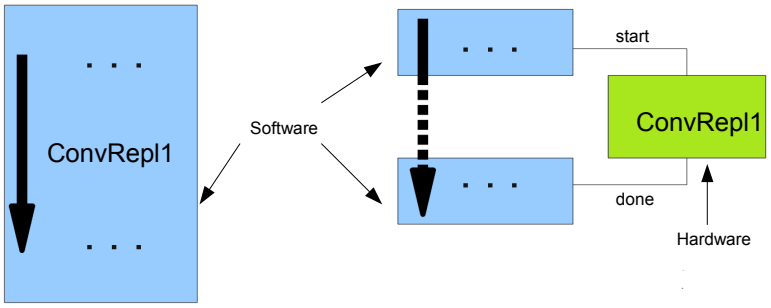

Figure 3.5: Hardware accelerating the application (on the right) vs software-only implementation (on the left).

With this, the executable file was set to run in the PowerPC 440 processor. Other compilation options could be set in the same way that is done for the regular gcc compiler.

3.3 Profiling the Application

Because the application was running in one processor, when a function of the program that was more time consuming was executing, it would delay the quicker functions. While in execution, the slower function would make the others to be stopped waiting until they were called. This causes a bottleneck. By speeding up the most time consuming functions of the code, the bottleneck is "enlarged" and the application shall be accelerated respecting a balanced trade-off between speed and FPGA area.

Only the most critical parts of the algorithm, or in other words, the most time consuming functions, are desired to be hardware accelerated. The main reason for this is the limited reconfigurable mesh area available to create each hardware module which, would not be enough to create hardware modules to replace all the application's functions. Giving priority in the conversion of the most critical functions to hardware modules, provides the highest degree of acceleration.

In order to determine the most critical functions, powerpc-linux-gprof was used to profile the code compiled with four degrees of optimization: no optimization, -O1, -O2 and -O3. The powerpc-linux-gprof is identical to gprof, the only difference being that it is aimed to profile executable files cross-compiled to the specific development board's processor. For comparison with the outputted values of the application and the speed obtained, the application was also compiled in a computer with a i5 quad-core processor running at 3.20GHz.

The following results (from subsection 3.3.1 through 3.3.5) were taken from the flat profile[2] tables obtained in Gprof (and powerpc-linux-gprof).

[2]Tells the total amount of time the program spent executing each function.

% time	cumulative secs	self secs	calls	self ms/- call	name
28.20	76,59	76,59	1440	53,19	ConvVBRepl2_uS_hS_yS
28.13	152,98	76,39	1440	53,05	ConvVBRepl1_uS_hS_yS
13,49	189,62	36,64	960	38,17	ConvVBConst_uS_hS_yS
7,48	209,94	20,32	10	2032,22	R_c4_RobustPoseEstimate_library
7,06	229,12	19,18	2700	7,1	sift_match_one
6,52	246,84	17,72	480	36,92	harrisTile_model_step

Table 3.1: Profiling with no optimization in the PowerPC processor (ten executions).

3.3.1 Profiling With No Optimization

The first approach was to profile the code with no optimizations. By doing this, the compiler's goal is to reduce the cost of compilation and to make debugging easier. However, since no optimizations are made, the created executable file will take longer to execute.

With this in mind, the profiling of the executable file was made. Because the runtime figures that gprof gives are based on a sampling process, they are subject to statistical inaccuracy. The solution advised by [28] is to combine data from several runs. For better accuracy, the results that will be presented throughout this document are based on ten executions of the profiled application. The result of the "No optimization" flag can be seen in the table 3.1.

The information contained in each column of the table 3.1 represents:

- **% time** — Percentage of the time that is spent in the function present in the column "name". This values do not account for the time the function is stopped waiting for others to execute.
- **Cumulative secs** — The time spent in the function plus the time spent in all the functions above.
- **Self secs** — The amount of time spent on the function throughout the execution of the application. It's only accounted its execution time.
- **Calls** — Number of times the function was called in the application.
- **Self ms/call** — Average time spent on the function per call.
- **Name** — Name of the function.

It can be seen that there are three functions that consume much more time than the others namely, ConvVBRepl2_uS_hS_yS, ConvVBRepl1_uS_hS_yS and ConvVBconst_uS_hS_yS. With the use of powerpc-linux-gprof it could be determined that the total execution time for the application took 27,16 seconds (or 271,57 seconds for ten executions).

% time	cumulative secs	self secs	calls	self ms/- call	name
25,41	34,40	34,40	1440	23,89	ConvVBRepl2_uS_hS_yS
23,54	66,26	31,86	1440	22,12	ConvVBRepl1_uS_hS_yS
13,67	84,77	18,51	960	18,28	ConvVBConst_uS_hS_yS
9,73	97,94	13,17	480	27,44	harrisTile_model_step
8,16	108,98	11,04	10	1104	R_c4_RobustPoseEstimat
6,60	117,92	8,94	2700	3,31	sift_match_one

Table 3.2: Profiling with the -O1 optimization in the PowerPC processor (ten executions).

3.3.2 Profiling With the -O1 Optimization Flag

With the optimization flag set to -O1, the compiler proceeds to a series of optimizations with the objective of reducing both the size of the executable file and execution time, without performing optimizations that require a great deal of compilation time. The results of this optimization can be seen in table 3.2. Again, the same three functions detected in the subsection 3.3.1 are consuming most of the time.

With -O1 optimization the application ran faster (as expected), having spent 13.54 seconds for execution.

3.3.3 Profiling With the -O2 Optimization Flag

Setting the optimization flag to -O2 enables almost all the optimizations that do not involve space-speed trade-off, as well all the optimizations that -O1 enables. The set of optimizations contain alignment of the start of loops, integration of small functions into their callers (only when the size of the code will not increase by this action) among others. The results of the profile with -O2 can be seen in table 3.3.

The execution time of the application was 13.5 seconds. In this case, the two top functions switched place however, looking from the percentages present in both the table 3.3 and 3.2 it can

% time	cumulative secs	self secs	calls	self ms/- call	name
23,98	32,37	32,37	1440	22,48	ConvVBRepl1_uS_hS_yS
23,97	64,73	32,36	1440	22,47	ConvVBRepl2_uS_hS_yS
13,76	83,31	18,58	960	19,35	ConvVBConst_uS_hS_yS
11,07	98,25	14,94	480	31,12	harrisTile_model_step
8,32	109,48	11,23	10	1123	RobustPoseEst_RANSAC_diag
6,58	118,36	8,88	2700	3,29	sift_match_one

Table 3.3: Profiling with the -O2 optimization in the PowerPC processor (ten executions).

% time	cumulative secs	self secs	calls	self ms/- call	name
24,53	32,95	32,95	1440	22,88	ConvVBRepl2_uS_hS_yS
24,01	65,20	32,25	1440	22,39	ConvVBRepl1_uS_hS_yS
13,87	83,83	18,63	960	19,4	ConvVBConst_uS_hS_yS
11,04	97,31	13,48	480	28,08	harrisTile_model_step
8,42	108,62	11,31	10	1131	R_c4_RobustPoseEstimate_library
6,66	117,57	8,95	20	447,5	matc_lr

Table 3.4: Profiling with the -O3 optimization in the PowerPC processor (ten executions).

be seen that the presented values did not change much and did not influence the obtained profiling data.

3.3.4 Profiling With the -O3 Optimization Flag

The -O3 optimization flag instructs the compiler to produce the fastest possible executable file. Various optimizations are made such as function inlining. The obtained results are present in table 3.4.

With -O3, the application's execution time wasn't significantly altered, taking 13.43 seconds. This optimization, as expected, was the one that produced the fastest executable file since all the speed optimizations are implemented.

3.3.5 Profiling in the Computer

The code was also compiled but, using gcc for the cross development platform, to be compared. The results of the profiling using gprof are present in tables 3.5, 3.6, 3.7 and 3.8.

% time	cumulative secs	self secs	calls	self us/call	total ms/call	name
32,45	0,86	0,86	1440	597,222	0.60	ConvVBRepl2_uS_hS_yS
26,79	1,57	0,71	1440	493,0556	0.49	ConvVBRepl1_uS_hS_yS
12,08	1,89	0,32	960	333,333	0.33	ConvVBConst_uS_hS_yS
8,30	2,11	0,22	480	458,333	4.48	harrisTile_model_step
6,79	2,29	0,18	2700	66,667	0.07	sift_match_one
3,40	2,38	0,09	10	9000	22.00	R_c4_RobustPoseEstimate_library

Table 3.5: Profiling with no optimization in the PC (ten executions).

% time	cumulative secs	self secs	calls	self us/call	total ms/call	name
27.37	0.26	0.26	1440	180,556	0.18	ConvVBRepl2_uS_hS_yS
22.11	0.47	0.21	1440	145,833	0.15	ConvVBRepl1_uS_hS_yS
14.74	0.61	0.14	960	145,833	0.15	ConvVBConst_uS_hS_yS
10.53	0.71	0.10	2700	37,037	0.04	sift_match_one
5.26	0.76	0.05	15400	3.247	0.01	RobustPoseEstimate_AbsOrQuat
5.26	0.81	0.05	480	104.167	0.10	sfun_peakidx_tile_Outputs_wrapper

Table 3.6: Profiling with the -O1 optimization in the PC (ten executions).

% time	cumulative secs	self secs	calls	self us/call	total ms/call	name
32.86	0.23	0.23	1440	159,722	0.16	ConvVBRepl2_uS_hS_yS
18.57	0.36	0.13	480	270,833	1.17	harrisTile_model_step
14.29	0.46	0.10	1440	69,444	0.07	ConvVBRepl1_uS_hS_yS
14.29	0.56	0.10	960	104,167	0.10	ConvVBConst_uS_hS_yS
5.71	0.60	0.04	15400	2,597	0.00	RobustPoseEstimate_AbsOrQuat
4.29	0.63	0.03	2700	11,111	0.01	sift_match_one

Table 3.7: Profiling with the -O2 optimization in the PC (ten executions).

% time	cumulative secs	self secs	calls	self us/call	total ms/call	name
19.12	0.13	0.13	1440	90,278	0.09	ConvVBRepl1_uS_hS_yS
17.65	0.25	0.12	480	250	0.98	harrisTile_model_step
17.65	0.37	0.12	960	125	0.13	ConvVBConst_uS_hS_yS
13.24	0.46	0.09	1440	62,5	0.06	ConvVBRepl2_uS_hS_yS
10.29	0.53	0.07	20	3500	3.50	match_lr
7.53	0.58	0.05	10	5000	11.00	R_c4_RobustPoseEstimate_library

Table 3.8: Profiling with the -O3 optimization in the PC (ten executions).

3.4 Analysis of the Critical Functions

From the previous section, three functions were identified as the most time consuming, those are: ConvVBCons_uS_hS_yS, ConvVBrepl1_uS_hS_yS and ConvVBrepl2_uS_hS_yS, hereafter defined as ConvConst, ConvRepl1 and ConvRepl2. These three functions are all children of the same function, named harrisTile_model_step, i.e., they are called by harrisTile_model_step. Analysing the results given by linux-powerpc-gprof for the -O3 optimization (table 3.4) it can be seen that this three functions spend 62.41% of the total execution time.

All the three functions named previously are called to analyse a matrix of 96 by 96 elements (called the U matrix of the module) using an array with constant values (called the H array of the

module) needed to make the appropriate calculations. In fact, the U matrix is represented in the source code as an array consisting of 9216 elements (the U array). The main operation of these three target functions is the same, a convolution. An equation representing the convolution present in these modules is the following:

$$acc_n = U[u] \times H[h] + acc_{n-1}; \tag{3.1}$$

Where U and H are the inputs of the the function, u and h are the floating-point elements of the arrays and acc is an accumulator that starts at zero. This operation is made eleven times in both ConvRepl1 and ConvRepl2 and nine times for ConvConst. After that the value of the acc (which contains the sums of the products of U[u] with H[h]) is passed as the y element of the matrix Y (Y[y]) which is the output of these functions. This is made for every one of the 9216 elements of the Y matrix and although the PowerPC has a single cycle multiplier, this is only true when the operation is done between integer values, and not with real values. The problem with real values (or floating-point values, in programming language) is that the PowerPC processor of the used board cannot, by itself, do operations like additions and multiplications with them. Instead, the values are passed to the Xilinx soft floating-point unit which is slower than the processor. According to [29], using this unit for a multiplication with two floating-point values takes 4 cycles at 200MHz which yields 20ns. For the case of the PowerPC processor, given that the defined operating frequency is 400MHz and it has a single cycle multiplier, an integer multiplication takes only 2,5ns.

The functions ConvRepl1 and ConvRepl2 are very similar, except that the matrix received by the first one comes from ConvConst and is processed in a horizontal way. The ConvRepl2 receives the matrix that comes from ConvRepl1 and does the calculations in a vertical way. ConvConst receives a matrix composed by values ranging from 0 to 255, and besides the constant H array, it has also six more arrays that are used as inputs namely, bSStart, bSEnd, bSPreEdges, bSPostEdges, bSNumPreEdges, bSNumPostEdges. This six arrays are however, just used for control purposes.

A figure representing the data flow inside harrisTile_model_step can be seen in figure 3.6. The operations represented by circles are simple multiplications of the matrices' elements. Since the software application executes in a sequential fashion, every function can only execute when its preceding function has finished. This is the case of these three functions and therefore, harrisTile_model_step calls them one after another starting by the convconst function and using the order shown in the figure 3.6. As it will be shown in the chapter 5 one of the main advantages of the hardware acceleration is that the modules corresponding to these functions can be executing in parallel to further improve the final application's execution time.

Since these functions were the ones that were going to be synthesized, special attention was paid to how they were described in software and, more importantly, the output values of each of them specially, the output of the function ConvRepl2, since this was the function that would drive the final values of the flow.

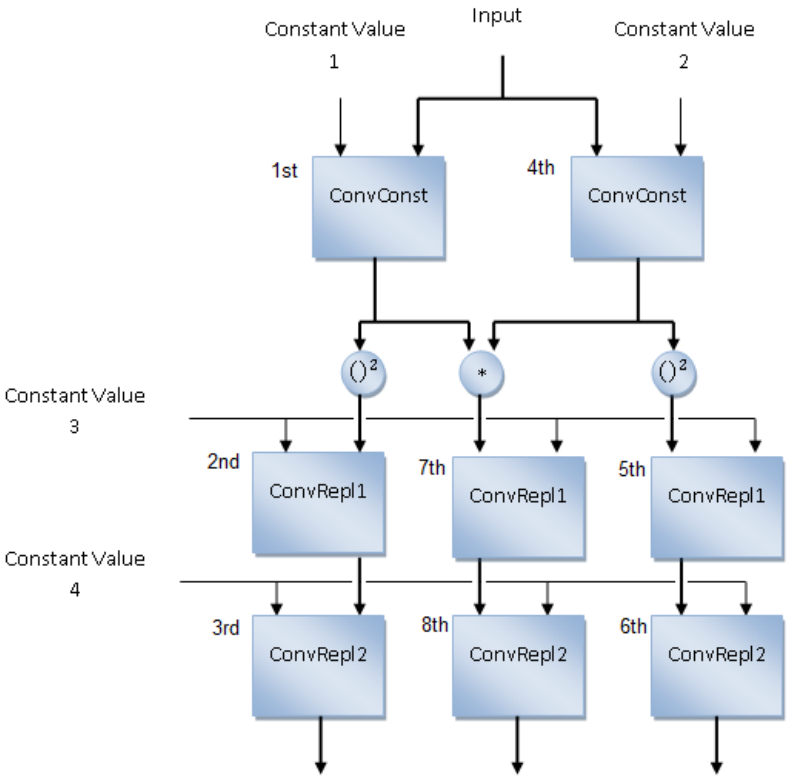

Figure 3.6: Dataflow of the application.

3.4.1 ConvConst

The ConvConst function receives, as an input, the grayscale values of the pictures (the U matrix), divided in tiles, captured by the cameras. Each of this tile is composed by 96×96 pixels and each captured picture contains 24 tiles. This function has, also as an input, two constant matrices(*Constant Value 1* and *Constant Value 2* in figure 3.6), one for each time this function is called by harrisTile_model_step. This two matrices have the values present in equations 3.2 and 3.3 and will be used as the H matrices along with the pictures's tiles in a convolution process, as per figure 3.7 .

$$ConstantValue1 = \begin{bmatrix} -1 & -1 & -1 \\ 0 & 0 & 0 \\ 1 & 1 & 1 \end{bmatrix} \tag{3.2}$$

$$ConstantValue2 = \begin{bmatrix} -1 & 0 & 1 \\ -1 & 0 & 1 \\ -1 & 0 & 1 \end{bmatrix} \tag{3.3}$$

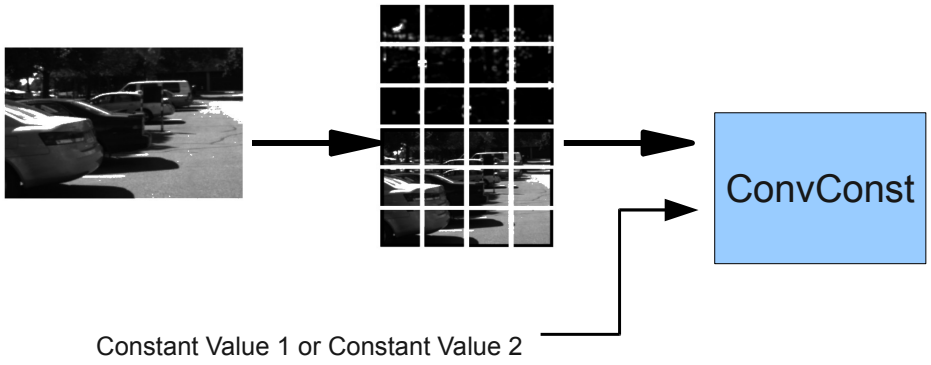

Figure 3.7: Input data of the ConvConst model.

As said before, the input tiles are in fact 96×96 matrices. These, along with the H matrix are used in a convolution process to crate a new matrix (the Y matrix) that will serve as the ConvRepl1 input (U matrix).

The calculus of the output array (array Y) follows the 2D convolution expression. The general calculus for this 2D convolution is shown in the following equation, where u[i,j] is the element in the column i and row j of the U matrix (tile of the picture) and h[i,j] is the element in the column i and row j of the H matrix (either *Constant Value 1* or *Constant Value 2*).

$$\begin{aligned} y[i,j] =& u[i-1,j-1].h[i+1,j+1]+u[i,j-1].h[i,j+1]+u[i+1,j-1].h[i-1,j+1]+ \\ & u[i-1,j].h[i+1,j]+u[i,j].h[i,j]+u[i+1,j].h[i-1,j]+ \\ & u[i-1,j+1].h[i+1,j-1]+u[i,j+1].h[i,j-1]+u[i+1,j+1].h[i-1,j-1] \\ & \forall i \in [1:94], j \in [0:95] \end{aligned} \tag{3.4}$$

This function is called two times by the harrisTile_model_step. In the first time, one tile is processed with the use of the *Constant Value 1*, while in the second time the same tile is processed, but now, with the use of the *Constant Value 2*. This will generate two new matrices (*Y1ofConvConst* and *Y2ofConvConst*). This matrices, after a multiplication or a power of two will give place to three new matrices *U1ofConvRepl1*, *U2ofConvRepl1* and *U3ofConvRepl1*.

$$U1ofConvRepl1 = Y1ofConvConst^2 \tag{3.5}$$

$$U2ofConvRepl1 = Y2ofConvConst^2 \tag{3.6}$$

$$U3ofConvRepl1 = Y1ofConvConst \times Y2ofConvConst \tag{3.7}$$

The ConvConst function has also six more inputs: bSStart, bSEnd , bSPreEdges, bSPostEdges, bSNumPreEdges and bSNumPostEdges. These are used for control purposes for the case of the outer margins of the U matrix where the normal convolution of equation 3.4 could not be used.

3.4.2 ConvRepl1

The ConvRepl1 proceeds differently from the ConvConst function. In this, H is an array instead of a matrix. The H vector (represented by *Constant Value 3* in figure 3.6) is composed by eleven elements (see array on equation 3.8) and is convoluted through eleven elements of the same line of the U matrix (either *U1ofConvRepl1*, *U2ofConvRepl1* or *U3ofConvRepl1*).

$$\begin{aligned} ConstantValue3 \quad = [&-3,548294306 \times 10^{-2}; -5,850147083 \times 10^{-2}; -8,630958945 \times 10^{-2}; \\ &-1,139453053 \times 10^{-1}; -1,346104741 \times 10^{-1}; -1,423004717 \times 10^{-1}; \\ &-1,346104741 \times 10^{-1}; -1.139453053 \times 10^{-1}; -8,630958945 \times 10^{-2}; \\ &-5,850147083 \times 10^{-2}; -3,548293561 \times 10^{-2}] \end{aligned} \tag{3.8}$$

$$\begin{aligned} &y[i,j] = u[i-5,j] \times h[10] + u[i-4,j] \times h[9] + \\ &u[i-3,j] \times h[8] + u[i-2,j] \times h[7] + u[i-1,j] \times h[6] + \\ &u[i,j] \times h[5] + u[i+1,j] \times h[4] + u[i+2,j] \times h[3] + \\ &u[i+3,j] \times h[2] + u[i+4,j] \times h[1] + u[i+5,j] \times h[0] \end{aligned} \tag{3.9}$$

The convolution process made in the ConvRepl1 function is present in the equation 3.9. This function processes the input matrix U, line by line. Note that the index of the U matrix's column can never be smaller than 0 nor greater than 95 (if it is, the convolution process truncates it to 0 or 95 respectively). For instance, the first element of the U matrix that will be used in the convolution for the element y[0,0] is u[0-5,0]. This value is automatically converted to u[0,0]. The output matrices Y (*Y1ofConvRepl1*, *Y2ofConvRepl1* and *Y3ofConvRepl1*) will be directly fed to the ConvRepl2 function (*U1ofConvRepl2*, *U2ofConvRepl2* and *Y3ofConvRepl2*)

3.4.3 ConvRepl2

The ConvRepl2 function is very similar to ConvRepl1. The only differences are its inputs where, the U matrix comes directly from the outputted matrix of the ConvRepl1 function and its H array (matrix 3.10). Also, the convolution process itself is different since the U matrix is processed vertically.

$$\begin{aligned} ConstantValue3 \quad = [&-3,548293561 \times 10^{-2}; -5,850147083 \times 10^{-2}; -8,630958945 \times 10^{-2}; \\ &-1,139453053 \times 10^{-1}; -1,346104741 \times 10^{-1}; -1,423004419 \times 10^{-1}; \\ &-1,346104741 \times 10^{-1}; -1.139453053 \times 10^{-1}; -8,630958945 \times 10^{-2}; \\ &-5,850147456 \times 10^{-2}; -3,548293188 \times 10^{-2}] \end{aligned} \tag{3.10}$$

3.5 Conclusions

The functions *ConvConst*, *ConvRepl1* and *ConvRepl2* were identified as the target functions to be hardware accelerated. Through profiling methods it was determined that these functions could consume as much as 62.41% of the application's total execution time. All the three functions execute additions and multiplications of floating-point variables (which is known to be a computational intensive task for processors), as well as, memory accesses (by making use of arrays).

To better understand the operations involved, a brief description of the target functions was made. It was also explained some of the steps necessary to execute the application in the PowerPC 440 processor of the the ML507 development board (installation of the embedded Linux Kernel and cross-compilation of the application).

Chapter 4

Creating the Hardware Modules

Having the application been executed in the embedded processor of the ML507 and the most time consuming functions been determined, the next step was to start building the hardware modules that would replace them. These dedicated modules would operate a set of data provided by the software application, the same way as their respective functions.

To create the hardware modules from the C language functions, it was used Catapult C from Mentor Graphics. This tool, as said before, can convert the High-level language that is C, to Verilog. However, the original C code cannot be directly converted by Catapult C instead, small changes had to be made since assumptions taken by the programmer that wrote the code are valid when implemented in software but not in hardware.

4.1 Catapult C

The first step taken to start working with Catapult was to adapt the C source files. That meant optimizing the code so that redundant assignments of variables were eliminated, and defining the size of the array values that served as the functions's inputs. If the array size was not defined, there was no way Catapult could tell how many elements the module needed to access. Although when executing in software this care isn't needed, hardware designing requires a strict definition of the inputted and outputted values since its internal structure can't be changed.

With the source code changes done, the modules to be synthesized could be characterized in Catapult. In the first step, called the "setup design" phase, some options were made, like choosing the target FPGA, setting the clock frequency and the handshake signals that would drive the module. The handshake signals used in the modules were the *start* and *done* signals. These two signals are used only for control purposes. At startup time, the *done* signal is initially at a high level signalling that the module is not operating any data. By setting the Start signal to a logic one,

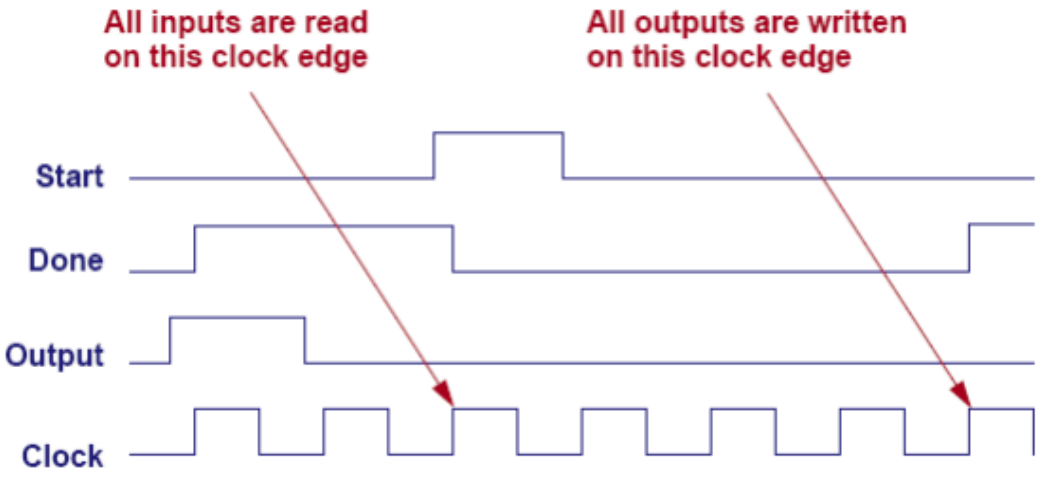

Figure 4.1: Start and done handshake signals. Taken from [30].

the module starts to execute in the following clock cycle thus, bringing the *done* logic value from one to zero. The design sets the *done* signal to high in the same cycle that the final return value is written to the outputs. This process can be seen in figure 4.1 where the *done* signal coming from a module is put to zero in the next clock cycle that it received a *start* from an external source. Upon execution, the module brings again the Done signal to a logic one.

This will be used in the final application in the following way:

1. The processor is executing the application;
2. The application reaches the point where one of the functions that are now replaced by hardware blocks were. The data needed by the module is stored in a memory by the processor where the hardware module can have access;
3. The *start* signal is sent to the respective module ;
4. The application (running in the processor) waits until the module finishes its execution by reading the logic value of the *done* signal;
5. The processor continues to run the application with the values retrieved from module's operations.

Catapult C has the ability to convert a function in C to synthesizable hardware. The hardware module, has all the same functionalities of the function. When creating an hardware module from a function, Catapult C will analyse all the the variables that are passed to the function and all the variables that it returns. With that information it will infer the type of interface needed. For the case of the function ConvRepl1 (which has the header shown in listing4.1) when analysed,

Catapult will create a module with inputs for both the u and h values, since they are declared as constants, while y will be defined as an output because it is detected that the module only writes values to that variable.

Listing 4.1: ConvRepl1 function header

```
ConvRepl1(const real32 u[9216],const real32 h[9],real32 y[9216]);
```

At this point, two alternatives were possible: either the input and output vales of the modules were stored in a RAM or, every individual value was stored in a block of registers.

When using registers, every value is constantly present at the module's inputs being used when needed. Although in this alternative the module executes faster than when using a RAM, for large matrices too much connections have to be made to the module, making the final architecture complex. With a RAM, the connections between itself and the module are simplified, being only used a wire for the enable signal (to activate the RAM) a bus where the module specifies the address to be read, and another one to which the RAM's values will be sent (it is also needed a wire to enable to write to the memory and a bus to send data to the RAM when writing). However, this alternative is slower because when the module requests a given value present in an address, it has to send the corresponding address to the RAM memory where , in the next clock cycle, it will send the requested value. A representation for both architectures using a RAM memory and a set of registers is shown in figure 4.2

As said before, the three target functions operate the U matrix composed by 96x96 elements and store their results in the matrix Y with the same number of elements. Having registers to store this values would not be a good solution because it would be created a module with too much inputs and outputs and, to connect them all to registers would be too much time consuming and error prone (Catapult C does not handle the creation of the data storage elements outside the modules). So, the RAM solution was adopted, where one RAM for each the U and Y matrices was used. Although the H matrices/arrays were quite small and could be stored more efficiently

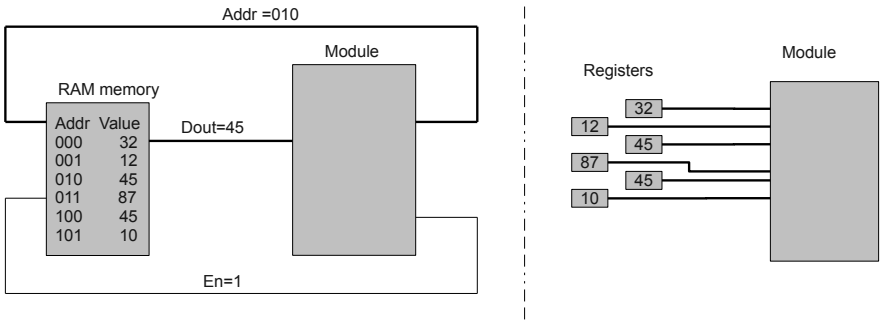

Figure 4.2: Differences in architecture designs when using a RAM memory for data storage (left side of the figure) and using individual registers (right side of the figure).

in registers, that alternative was not pursued. Instead, another RAM was used for each. The reason for this is justified by the convolution's characteristics, where U and H are used at the same time. Since the module would read both the values at the same time, if the H vector was stored in individual registers, the values were immediately available but, since the operation would not start until the elements of the vector U were available, H would not be used. So having the H values readily available would not bring any speed advantages, but design complexity.

The RAM alternative was also used to store the bSStart, bSEnd , bSPreEdges, bSPostEdges, bSNumPreEdges and bSNumPostEdges values of the ConvConst module.

When trying to synthesize the modules, a problem appeared: Catapult C could not accept floating-point values in the functions that would be translated to synthesizable hardware. According to [30] the floating-point variables are automatically converted to fixed-point, using the "hls_fixed" pragma. This was a major setback since the Virtex-5 FPGA can make use of IP-Cores designed specifically for floating-point operations. So, although the rest of the application could be left unchanged, the functions to be converted to hardware had to eliminate the use of floating-point datatypes. Because it was desired to keep the application functionalities unchanged to the maximum, the only acceptable solution was to have it using floating-point datatypes outside the target functions and, just before these functions are called, convert the data from float to any other datatype that Catapult C can accept. Immediately after the completion of each one of the target functions, the values would have to be converted again to floats and the rest of the application could resume normally.

But using the "hls_fixed" didn't work for the intended application since although a fixed-point could be created from a float, the same was not true for the opposite operation. A better solution had to be found.

4.2 Solutions

To work around the no floating-point problem, the application's source code had to be altered so that all the functions that were going to be synthesized to hardware made use of other datatypes. The rest of the application could, however, continue using floating-point variables.

So, to interface the target functions with the rest of the program, it was needed to create one function that would take as argument the float value and return the resultant emulated value. The target functions were also tuned to proceed to the multiplication and addition over the emulated values. Another function was created to retrieve the floating-point value.

Three solutions were considered: a lighter solution composed by a 32-bit fixed-point datatype, an highly flexible 32-bit soft floating-point solution and a mid-ground solution using a 64-bit fixed-point datatypes.

4.2.1 Fixed-point With 32 Bits

The first alternative that was studied was to make use of a 32-bit fixed-point notation. From a series of tests made to the *acc* values used in the convolution equation in the target functions (refer to equation 3.1) it was observed that the maximum value that it had to represent was 565931.75 (absolute value). To be able to represent this value, at least 20 bits were needed (as shown in equation 4.1). Because the *acc* variable also comprises negative values, one more bit was used as a sign bit.

$$log_2(565931.75) \approx 19.1103 => 20 \tag{4.1}$$

Having 21 bits for the integer part, leaves only 11 more for the fractional part. This gives a precision of approximately 0,000488 which is not enough precision for application's values, according to the tests that were made. Anyway, the application was tested to see how this solution would affect the final application. To emulate this fixed-point solution, an int datatype was used.

Because it was used an int to emulate this fixed-point datatype, the multiplication process had to be redefined (the addition did not needed to be changed). For that, the *mymult* function was created. This function proceeded to a multiplication of two values and shifted the result 11 bits to the right to maintain the same number of bits in the fractional part. The *mymult* was then included in the three target functions.

When the application was run in the PowerPC processor (still with the *mymult* function in software) the outputted results were incorrect (appendix A.2), having deviations from the values outputted by the original application. So, the solution proved to be of no use. However, it was decided to synthesize hardware blocks for the *mymult* function for comparison reasons.

Because the Virtex-5 has IP-Cores capable of doing multiplications and additions over floating-point variables, *mymult* was converted to sinthesizable hardware using Catapult C in order to estimate how much processing power it was lost due its use. Five design solutions were made were the only difference between them was their operating clock frequencies. The obtained results can be seem in the table 4.1.

In this table, the column "Freq. From Catapult" represents the values that were asked, to Catapult C, to be met by the hardware modules created. "Freq. in XST" is the frequency of the actual modules obtained when synthesized by Xilinx ISE. The "PAR frequency" is relative to the frequency obtained by the modules after being placed and routed in the FPGA. Both the synthesis and the Place and Route processes (and their meaning) will be explained later in the document. The column "Throughput cycles" represents the number of cycles that the module takes to execute. The final two columns represent the time spent by the module when using the frequencies given by the synthesis and the place and route process.

4.2.2 Fixed-point With 64 Bits

The 64-bit fixed-point solution was very similar to the 32-bit one, but now a long long datatype (or long int) had to be used.

Freq. From Catapult (MHz)	Freq. in XST (MHz)	PAR frequency (MHz)	Throughput cycles	Throughput time XST (ns)	Throughput time PAR (ns)
50	781,25	317,058	6	7,680	18,924
100	529,381	427,35	9	17,001	21,060
200	64,366	61,316	10	155,362	163,090
400	61,705	53,339	15	243,092	281,220
500	61,705	53,76	18	291,711	334,821

Table 4.1: Timimg results for the *mymult* module (32-bit multiplication).

From the tests made to the application, it was determined that one bit for the signal, 20 for the decimal part, and 20 for the fractional part would be a good solution. Having 64 bits to dispose, it was opted to have 32 bits to represent the integer part and signal, while the rest used to the fractional part.

This solution was a bit more complex and slower than the one described in 4.2.1 because, although the addition could be maintained, the multiplication involved, in fact, four multiplications with integers.

The procedures that were made for the 32-bit fixed-point solution were redesigned having in mind the 64 bits. A new function *mymult64* was created to multiply the the new data types.

Again, to check the processing speed of the resultant modules of the *mymult64*, the function was converted using Catapult C for five operating frequency scenarios. Those results are present in table 4.2.

Freq. From Catapult (MHz)	Freq. in XST (MHz)	PAR frequency (MHz)	Throughput cycles	Throughput time XST (ns)	Throughput time PAR(ns)
50	154,131	117,398	4	25,952	34,072
100	80,682	71,495	7	86,760	97,909
200	64,366	61,316	15	233,042	244,634
400	61,705	53,339	20	324,123	374,960
500	76,511	71,649	23	300,610	321,009

Table 4.2: Timing results for the *mymult64* module (64-bit multiplication).

This solution, at first sight, was good enough since the final output values were the same as of the original application (appendix A.1). But a closer look has shown that the intermediate values that were passed between functions changed by a certain degree and, since the tests made

to the application were based only on two pairs of images , there was no guarantee that the 64-bit fixed-point was, indeed, a good solution.

The fixed-point solution with 64 bits was discarded because of the lack of test pictures to use. Nevertheless, this solution could be a serious candidate in future work, provided that some care is taken with its integration in the application.

4.2.3 Soft Floating-point With 32 Bits

The soft floating-point approach used, flowed the IEEE 754 standard definition where, a 32-bit single-precision float is divided in three components. The first (the leftmost), has only one bit, the sign bit. This bit when set to zero represents a positive value, when set to one represents a negative value. The second component has 8 bits that represent the exponential part of the number. The third component contains the 23 remaining bits that compose the fractional [31]. The expression of the soft float representation is as shown in equation 4.2:

$$(-1)^S \times 2^{exponent-127} \times 1.fractional \tag{4.2}$$

The soft floating-point is an approach to represent and operate real numbers using software routines (or functions) instead of hardware. This solution although mostly used to emulate floating-point datatypes and operations in hardware architectures that do not contain an internal nor external FPU (Floating-Point Unit), can be used to overcome the no floating-point problem that Catapult C has, since a data type accepted by Catapult C can be used to replace the float. Once again, a 32-bit int was used to store the binary representation of the original float.

The main disadvantages of the software process to handle floating-point variables, relative to the hardware and the fixed point alternative is the slow operational speed as well as the memory overhead of the final application since the libraries necessary for its calculus have to be compiled along with the main source code. However, an advantage of the soft-float relative to the fixed point notation is both its order of magnitude and precision.

The software functions that provide the soft-float addition and multiplication were obtained by a library provided by [32]. According to the website, this software is completely faithful to the IEEE Standard. This library is comprised by a wide range of operators like division, multiplication, addition, subtraction, square root, greater than, smaller than, etc. Because most of this operations were not needed, only the ones related to the addition and multiplication were kept. Also, to reduce the execution time, all the exception handling routines (like, multiplication with infinity) were deleted and only one rounding mode was used (round to the nearest even).

Once again, the Stereo Navigation application had to be altered. In the harrisTile_model_step function, the floating point values had to be casted to the new soft floating-point data type before they could be used in the function ConvConst, ConvRepl1 and ConvRepl2. To do this, a function making use of pointers was created, the crtFloat, which proceeded the following way:

- Get the address of the floating-point number and store it in the float pointer;
- Copy the value that is stored in the float pointer to an int pointer;
- Get the value pointed by the int pointer to an int data type.

With this, the same binary representation of the floating-point value was stored in the int datatype.

The tests made to the application using the soft float alternative, gave results equal to the original Stereo Navigation application (appendix A.1). Also, the values that were passed between the modules, remained very similar to the originals. The same can be said to the tests made to the acc variable of the modules. So, the soft floating-point has proven to be a good solution to emulate the float datatype. With this, it was found the solution for the no floating-point problem.

Because the addition and multiplication of the floating-point values were, with this solution, processed purely by software functions, how the duration of the application as well as of all its functions was influenced, needed to be found. Again, powerpc-linux-gprof was used to profile the application. Since all the floating-point operations were made in software, it was expected that the application would suffer from a decrease in processing speed. The profiling results of the application, using again the four optimization degrees (no optimization, -O1, -O2, -O3), are present in the tables 4.3, 4.4 and 4.5. The total time of the application using the same four optimization degrees is present in table 4.6

Optimization level	% time	self (ses)	children(secs)	called
No optimization	18,2	10,45	231,08	960
-O1	15,9	5,23	43,87	960
-O2	15,9	4,83	38,66	960
-O3	14,7	4,63	27,44	960

Table 4.3: Profiling results for the ConvConst using the soft floating-point solution (ten executions).

Optimization level	% time	self(secs)	children(secs)	called
No optimization	34,0	28,00	423,66	1440
-O1	29,8	11.56	80,43	1440
-O2	28,6	9.78	68,54	1440
-O3	27,2	9.58	49,99	1440

Table 4.4: Profiling results for the ConvRepl1 using the soft floating-point solution (ten executions).

Where "% time" represents the percentage of the total time that was spent in this function, including time spent in subroutines called by it, in this case, the soft floating-point functions are

Optimization level	% time	self (secs)	children(secs)	called
No optimization	33,9	27,57	423,66	1440
-O1	29,9	11,57	80,43	1400
-O2	28,7	10,17	68,36	1440
-O3	27,2	9,58	49,99	1400

Table 4.5: Profiling results for the ConvRepl2 using the soft floating-point solution (ten executions).

taken in account; “self” is the total amount of time spent in this function; “children” is the total amount of time propagated into the function by its children (the multiplication and addition using the soft floating-point alternative).

Optimization level	Execution time (seconds)		Deceleration (times)
	Soft float	original	
No optimization	1329,58	271,57	4,9
-O1	308,17	135,50	2,27
-O2	273,51	135,00	2,03
-O3	218,81	134,31	1,63

Table 4.6: Profiling results of the application using the soft floating-point solution (ten executions)

It can be seen from table 4.6 that this software alternative of emulating floating-point variables, puts a major overhead on the application, delaying it 4,9 times in the worst case scenario. This is expected to affect the final hardware structure since the soft floating-point functions will be mapped in the modules created by Catapult C.

As done for the fixed point solutions, the hardware equivalent of the soft floating-point functions was tested to determine its processing speed for both the multiplication (table 4.7) and addition (table 4.8).

Freq. From Catapult (MHz)	Freq. in XST (MHz)	PAR frequency (MHz)	Throughput cycles	Throughput time XST (ns)	Throughput time PAR(ns)
50	76,857	66,176	5	65,056	75,556
100	90,145	93,676	9	99,839	96,076
200	178,126	110,558	17	95,438	153,765
400	198,255	199,282	31	156,364	155,558
500	205,719	185,356	35	170,135	188,826

Table 4.7: Timimg results for the soft floating-point multiplication

Freq. From Catapult (MHz)	Freq. in XST (MHz)	PAR frequency (MHz)	Throughput cycles	Throughput time XST (ns)	Throughput time for normalized frequency (ns)
50	88,667	65,007	5	56,391	76,915
100	113,792	91,676	9	79,092	98,172
200	184,69	147,145	16	86,632	108,736
400	178,405	122,339	31	173,762	253,394
500	215,515	156,372	37	171,682	236,615

Table 4.8: Timimg results for the soft floating-point addition

It was tried to simplify the readability of the code by making use of operator overloading to replace the multiplication and addition functions (listings 4.2 and 4.3 respectively). This required that the application made use of classes. However, after testing that the application was working correctly with this changes, when it was used in Catapult C to generate the hardware modules, it ceased functioning and exited abnormally without giving any information about the cause of the malfunction and therefore, could not be used as a solution.

Listing 4.2: Multiplication function header

```
result=float32_mul(A,B);
```

Listing 4.3: Addition function header

```
result=float32_add(A,B);
```

4.2.4 IP-Cores

Although Catapult C cannot accept floating-point variables to create the modules targeting the Virtex-5, that does not mean that this FPGA cannot operate this values. In fact, there are a set of pre-generate cores specifically for mathematical functions over floats. These IP-Cores (Intellectual Property Cores) are present in the standard library of components used by the Xilinx ISE.

To make a rough estimate about how much processing power was lost because of the solution used (and the fixed-point solutions), two Verilog modules were created: one for the floating-point addition and another for the multiplication. The used core was the Floating-point v4.0. The comparison results are observed in the table 4.9.

As it can be seen from the table 4.9 the solutions found were always slower than the IP-Core. This happens because the Floating-point v4.0 core is optimized at the maximum for the Virtex-5 FPGA. Because it is designed by Xilinx for a Xilinx product, all the constraints and design optimizations are known and strictly followed.

	Multiplication (ns)	Addition (ns)
Floating-point v4.0	11,664	20,899
Soft floating point	75,556	76,915
Fixed-point (32-bit)	18,924	-
Fixed-point (64-bit)	34,072	-

Table 4.9: Execution time for the multiplication and addition process for different approaches.

4.3 Creating the ConvConst Module

With the solution of the no floating-point problem found, the modules could be finally synthesized using Catapult C.

The first C function to be translated to a synthesizable hardware module was the ConvConst. In the setup design phase of Catapult C, the module's technology was chosen by defining the target board to be used. Also, a Start and Done control signal was added to the module. To generate a smaller and simple block, it was asked for the module to meet a low frequency constraint of 50MHz. Forcing the module to work at higher clock frequencies would make it larger since the most used operations of the module would be parallel which involved cloning some groups of components.

At this phase, the ConvConst function was chosen to be the top-level function. According to [30] everything called by the top-level function will be hardware and everything outside of that function is considered to be software or a test bench and will be ignored during the synthesis process. Given that ConvConst included the soft floating-point functions created, those were also synthesized.

With the design setup completed, the architectural constraints could be defined. Two options were possible for the modules architecture: achieve the minimum possible area or the minimum latency. This options could be chosen from the design goal constraint to specify the optimization goal for the new scheduler. By choosing the area optimization, a module using less components would be produced, however its latency time would be worse. Since the objective of this thesis was to accelerate an application, it was opted to configure the module for a low latency time. At this point, it was instructed that the module should obtain all its input values from RAM memories and store its output values also in a RAM memory.

With this done, the hardware description language of the module was generated. However, Catapult C creates a Verilog file that is too high-level for ISE to understand and synthesize. For that, it was used PrecisionRTL which is also a synthesis tool just like ISE. This tool accepted the high-level code (which contained statements like for loops and others) and was used to create a Verilog file that Xilinx ISE could use.

When analysing the resources that the module takes after synthesis it was noted that most of the FPGA components were being used, letting very few available for the other two modules. So another code optimization was needed.

As said earlier, the ConvConst function analyses a Matrix of 96 by 96 floating-point values making use of an array H with nine floating-point elements, along with six other small arrays composed by integer values. The matrix that serves as an input to ConvConst is in fact a tile of the image being processed and, as stated earlier in chapter 3, the image is coded in gray-scale using 8 bits, which represents only integer numbers ranging from 0 to 255. Also, the values of the matrix H are either -1, 0 or 1. This values although being defined as of the floating-point type in the original Stereo Navigation application's source code are, in fact, integer values. By using the int data type instead of float, all the operations using multiplications and additions can be made without involving the soft floating-point functions described earlier and, a simpler module could be synthesized.

Because the module ConvConst was altered, a new series of profiling were made to the application.

	ConvConst			application time
	% time	Self (sec)	children (sec)	
No optimization	0,7	7,66	0	1083,22
O1	0,8	2,22	0	262,92
O2	0,8	1,83	0	234,87
O3	0,7	1,26	0	190,91

Table 4.10: Profiling results of the Convconst function using int data types running on the PowerPC (ten executions).

From table 4.10 it can be seen that ConvConst relative duration dropped significantly. By replacing the soft floating-point alternative with the integer one, the relative time of ten executions of ConvConst, dropped from 14,7% (from table 4.3) to 0,7%. Although following this solution makes the function ConvConst to cease to be critical in the duration of the application, it was decided to continue to create its respective module to infer the degree of optimization obtained from a function using ints when compared with a function that is using soft floats.

The ConvConst function using ints, was passed by Catapult to be analysed. To respect a trade-of between area and latency, four solutions were implemented where the operating frequency of the modules to be created were changed. Catapult C after generating the HDL that would describe the modules outputted a series of characteristics for each that can be seen in table 4.11

The column "Minimum frequency" has the constraint asked to be met by the generated module. "Latency cycles" represent the number of clock cycles that the module spends executing the operation. Dividing the latency cycles by the minimum frequency, gives the latency time of the module. "Throughput cycles" and "Throughput time" are referent to the overall module operation, i.e, the time/cycles spent since a start signal is sent to the module until it ends its execution sending a done signal. The "slack" is used by Catapult to inform if the timing constraints were met, being

Minimum frequency(MHz)	Latency cycles	Latency time(ns)	Throughput cycles	Throughput time(ns)	Slack
50	97937	1958740	97938	1958760	4,77
100	106845	1068450	106846	1068460	-0,02
200	142485	712425	142486	712430	-0,25
400	196569	491422,5	196570	491425	0

Table 4.11: Results of the Catapult report referring the ConvConst module.

a value greater than 0 presented in the positive case.

Looking at the column "slack" from table 4.11, the module operating at a minimum frequency of 50MHz is the only one that is guaranteed to perform as expected in the respective frequency. From the column "Throughput time", the module designed to meet the minimum frequency of 400MHz is the one that is the quicker to execute.

After creating the Verilog modules with Catapult C, and converting them to a description language recognizable by the Xilinx ISE, the modules were synthesized to determine how the estimates made by Catapult C were close to be correct, namely if the frequency constraints were met. The Synthesis is a process made by Xlinx ISE that accounts for all the throughput times of each individual component (flip-flops, logic gates, etc.) that is used in the module. The frequency values resultant from the synthesis are presented in the table 4.12

Freq. from Catapult(MHz)	Max. Freq. obtained in XST
50	120,584
100	125,235
200	177,085
400	179,324

Table 4.12: Frequency obtained for the ConvConst module in XST.

These frequencies deviated from the required ones which altered the throughput time as shown in table 4.13. Where “Max. Freq. in XST” is the maximum frequency that the module can operate correctly determined from ISE after the synthesis process. The “throughput time from XST” is the time that the module takes to operate at the frequency given in “Max. Freq. in XST”.

Normally the values of the clock frequency used to drive the modules are multiples of 50MHz. The normalized frequency is the biggest multiple of 50MHz that can be contained by value “Freq. in XST”.

As it can be seen from table 4.13, because the frequency constraints were not met, the fastest module solution was not the one that was designed to operate at 400MHz. In fact using the module designed for a frequency of 200 MHz would produce a better result.

Freq. From Catapult (MHz)	Max. Freq. in XST (MHz)	Normalized frequency (MHz)	Latency Cycles	Throughput time from XST(us)	Throughput time for normalized frequency(us)
50	120,584	100,00	97837	811,36	978,37
100	125,235	100,00	106845	853,16	1068,45
200	177,085	150,00	142485	804,61	949,9
400	179,324	150,00	196569	1096,17	1310,46

Table 4.13: Throughput time obtained for the module ConvConst.

4.4 Creating the ConvRepl1 Module

Again, the same minimum frequencies were chosen for this module the same way as described in the previous section. These results can be seen in the table 4.14.

Minimum frequency(MHz)	Latency cycles	Latency time(ns)	Throughput cycles	Throughput time(ns)	Slack
50	839233	16784660	839234	16784680	-1,39
100	1447489	14474890	1447490	14474900	-0,36
200	3080065	15400325	3080066	15400330	-0,62
400	6159745	1539962,5	6159746	15399365	-1,41

Table 4.14: Results of the catapult report referring the convRepl1 module.

Proceeding the same way to synthesize the module, i.e., to generate a post-synthesis report, new frequency values were obtained. These values are present in table 4.15 along with the normalized frequency values .

Freq. From Catapult (MHz)	Freq. in XST (MHz)	Normalized frequency (MHz)	Latency Cycles	Throughput time from XST(us)	Throughput time for normalized frequency(us)
50	66,095	50	839233	12697,37	16784,66
100	103,831	100	1447489	13940,82	14474,89
200	162,602	150	3080065	18942,36	20533,77
400	163,747	150	6159745	37617,45	41064,97

Table 4.15: Throughput time obtained for the module convRepl1.

From table 4.15 the best solution is to use the module designed for 100MHz frequency since it is the one that has the lower normalized throughput time.

4.5 Creating the ConvRepl2 Module

Again, the same minimum frequencies were chosen for this module (as shown in table 4.16).

Minimum frequency(MHz)	Latency cycles	Latency time(ns)	Throughput cycles	Throughput time(ns)	Slack
50	839233	16784660	839234	16784680	-1,39
100	1447489	14474890	1447490	14474900	-0,36
200	3080065	15400325	3080066	15400330	-0,62
400	6159745	1539962,5	6159746	15399365	-1,41

Table 4.16: Results of the catapult report referring the convRepl2 module.

After synthesis, the module frequencies were the ones presented in table 4.17.

Freq. From Catapult (MHz)	Freq. in XST (MHz)	Normalized frequency (MHz)	Latency Cycles	Throughput time from XST(us)	Throughput time for normalized frequency(us)
50	63,307	50	837889	13235,33	16757,78
100	103,831	100	1446145	13927,87	14461,45
200	159,388	150	3078721	19315,89	20524,81
400	168,714	150	6150509	36455,24	41003,39

Table 4.17: Throughput time obtained for the module convRepl2

Once again, the best solution is to choose a module designed for 100MHz which has a throughput time of 14461,45us.

4.6 Conclusion

In this chapter was presented how the critical functions were used by Catapult C to create hardware modules. Because Catapult C did not accept floating-point datatypes, a solution making use of of soft floating-point had to be used, which penalized the original application's execution time.

Although there were requested modules that could be driven by clocks with a frequency ranging from 50MHz to 400MHz, catapult had an hard time to create modules with more than 150MHz. Because of that, the faster modules that were obtained, where driven with a clock frequency of 100MHz.

Chapter 5

Module Design

To obtain a software-hardware interface where the created modules could communicate with the PowerPC processor, the reference design used originally in XPS when the Linux kernel operating system was ported to the FPGA in 3.2.1, had to be altered. To simplify the communication process between the processor and the created modules, a single IP core was added to the device tree. This core contains the three implemented modules and the dual-port RAMs (DPRAMs)[1] used to store and obtain the required matrices/arrays of the modules. There was also added inside the core some additional logic for the correct communication between the modules, the DPRAMs and the outside.

A simple representation of this core is in figure 5.1. To keep this diagram simple, the DPRAMs used for the control arrays (bsStart, bsEnd, bsPreEdges, bsNumPreEdges, bsPostEdges and bsNumPostEdges) are not represented. It can also be seen that the PowerPC processor, through the PLB, can access one set of ports of the DPRAMs while each module can access the other set of ports. With this architecture, both the PPC440 and the modules "see" the DPRAMs as single port RAMs. Furthermore, although the modules were designed to make use of three RAMs (nine for the ConvConst module), in the figure 5.1 only two are connected to each module. The reason for this is that the array H was stored in the same RAM used for storing the matrix Y, as it will be explained later in the subsection 5.1.1.

Because the processor communicates with the core using just one bus, it was easier to have it access data from only one memory instead of the 12 necessary. So, the DPRAMs were made to look like a single bigger RAM to the PLB (the RAMX6 of the figure 5.1). This is done using the three msb of the address coming from the PLB as enable signals for the RAM as will be explained in section 5.3

[1] Type of random access memory that contains duplicated ports to enable multiple reads or writes to occur at the same time.

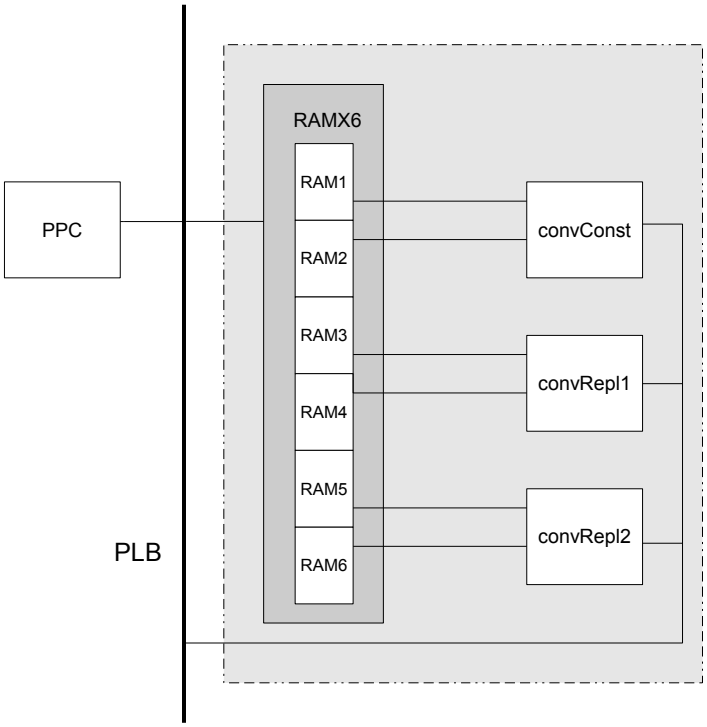

Figure 5.1: Architecture of the core to be implemented (in dashed line) as well its interface with powerPC via PLB.

The used core was derived from one pre-generated by XPS through the "create and inport peripheral wizard". By using this wizard, XPS creates an empty core with all the connections to the PLB already defined and respecting the communication protocol. With this module, there is no need to redefine all the PLB specifications like port connections, transaction signals, etc. This core, named user_logic, was represented by a Verilog file (user_logic.v). This file could be imported inside a project in ISE where, it could be altered to include the created modules. The generated core's interface can be seen in figure 5.2.

In figure 5.2 all the core's outputs are the ports on the right of the module, while the inputs are on the left side (this pattern will be used through the rest of the document). The ports were used in the design the following way:

- **Bus2IP_Addr** — Used to determine the address of the DPRAM that will be accessed;
- **Bus2IP_BE** — Not used in the design;
- **Bus2IP_CS** — Used to enable the DPRAM;
- **Bus2IP_Data** — Used to transfer data from the PLB bus to the IP core;
- **Bus2IP_RdCE** — Used to point the register to be read from;
- **Bus2IP_WrCE** — Used to point the register to be written to;

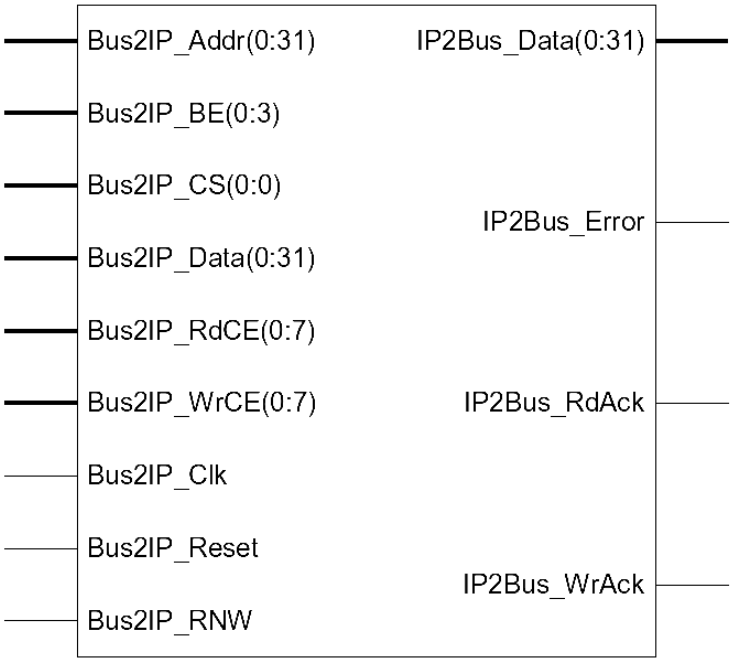

Figure 5.2: Core's interface diagram (left ports are inputs, right are outputs).

- **Bus2IP_Clk** — The clock signal to drive the modules;
- **Bus2IP_Reset** — To reset the module;
- **Bus2IP_RNW** — Used to enable the writing mode of the RAM;
- **IP2Bus_Data** — Sends the value present in the RAM's address pointed by Bus2IP_Addr if Bus2IP_CS is 0 else, sends the value present in the control register pointed by Bus2IP_RdCE;
- **IP2Bus_Error** — Never used, always at 0;
- **IP2Bus_RdAck** — Only used to confirm the reception of the Bus2IP_RdCE signal;
- **IP2Bus_WrAck** — Only used to confirm the reception of the Bus2IP_WrCE signal.

5.1 The Target Modules

The modules that were created and described in chapter 3, are very similar: all of the three analyse a 96 by 96 matrix and, using the H array, proceed to a series of operations to produce an output that will, itself, be a 96 by 96 array. The ConvConst module has six more input arrays that are used for control purposes, however its basic procedure is similar to the other two modules.

5.1.1 ConvRepl1 and ConvRepl2

The ConvRpel1 module included in the user_logic core, is the one created by Catapult C that was designed to achieve a minimum frequency of 100MHz. The representative schematic is present in figure 5.3. The module's port names were given by Catapult C and are relative to the used RAM outputs and inputs.

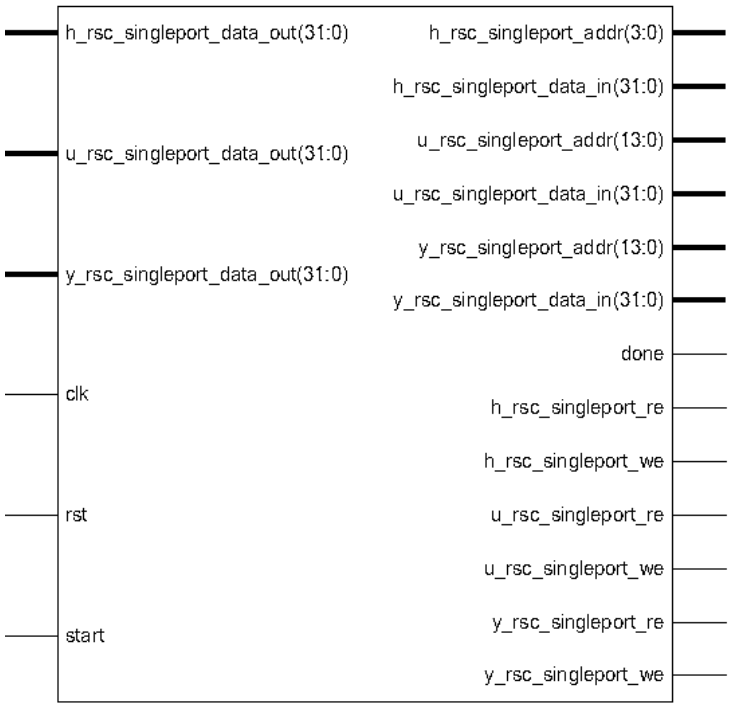

Figure 5.3: Interface representation of the ConvRepl1 module (left ports are inputs, right are outputs).

This ports are used for:

- **u/h/y_rsc_singleport_data_out** — To read the data that comes from the RAM memory;
- **u/h/y_rsc_singleport_addr** — To select the index of the array/vector to be read/written to the RAM;
- **u/h/y_rsc_singleport_data_in** — Sends the data to be stored in the RAM;
- **u/h/y_rsc_singleport_re**— Enables the read mode of the RAM;
- **u/h/y_rsc_singleport_wr**— Enables the write mode of the RAM;
- **clk**— Input for the clock that drives the module;
- **rst**— Used to reset the module;
- **start**— Used to start the module;

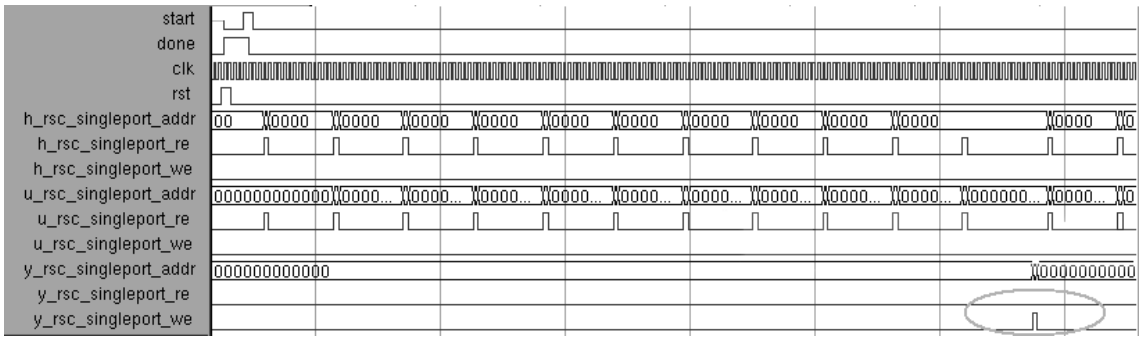

Figure 5.4: Waveform for some of the ConvRepl1 module's outputs .

- **done**— Used to signal that the module completed its operations.

The way the module requests a value from one of its DPRAMs is by putting a logic 1 in one of u/h_rsc_singleport_re ports by one clock cycle. During that period, the module presents a stable address value in the corresponding u/h_rsc_singleport_addr, the rest of the time, the addr values that the module outputs in these ports are ignored. The ConvRepl1 module (and the other two modules, for that matter) is configured so that in the following clock cycle it will expect the RAM answered value in the u/h_rsc_singleport_data_out input port.

ConvRepl1 only reads the u and h values from the RAMs and therefore, it will never need to use the output ports u_rsc_singleport_wr and h_ rsc_singleport_wr. So in the design, this ports were left unconnected. The same happens with the output buses u_rsc_singleport_data_in and h_rsc_singleport_data_in.

The RAM used to store the y values that the ConvRepl1 module outputs, will never be used to transmit its y values to the module so, the module's output y_rsc_singleport_re and input y_rsc_singleport_data_out are not used, and therefore, are not connected.

Proceeding to a behavioral analysis of this module (figure 5.4) it can be seen that the module, when reseted, has its *done* output at one, signaling that no operation is being made at the moment. Putting the *start* line to high, makes the module run (the *done* signal is brought to 0). When the module finishes executing its output signal *done* is brought again to a logic one.

Looking at the waveform of figure 5.4, u_rsc_singleport_re and h_rsc_singleport_re are at one exactly at the sime time (at the same clock cycle). As said before these signals are used to request a RAM value present in the address requested. Because the signals are at one at the same time, both the RAMs will access their values at the same time. That however does not happen for the RAM that is used to store the y values. As seen in the figure 5.5 (a zoomed version of the figure 5.4), the y_rsc_singleport_we is at one in a different clock cycle. This means that the RAM will access its values at different times. This enables the H array and the Y matrix to be stored in the same DPRAM (in different addresses though), making the module interfacing with only two DPRAMs which allows less memory space to be wasted.

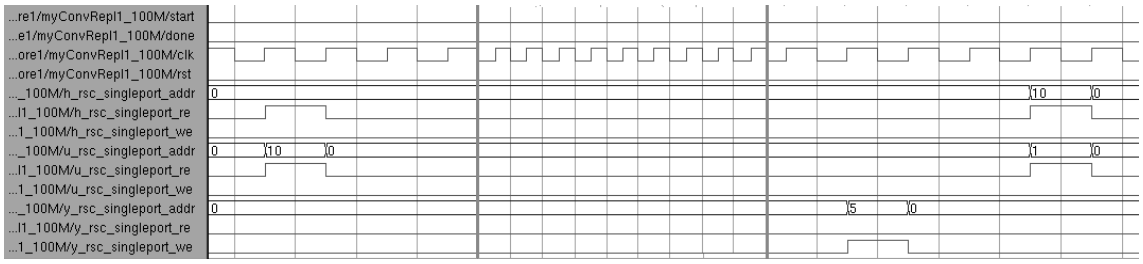

Figure 5.5: Waveform for some of the ConvRepl1 module's outputs (the image was compressed for readability reasons).

The ConvRepl2 module is similar to the ConvRepl1 in its interface and communication logic, being the only difference the sequence of words accessed in the RAMs. So, all the connections and controls are exactly the same. The module that was included in the user_logic core was also the one designed to have a minimum clock frequency of 100MHz.

5.1.2 ConvConst

The ConvConst module is very similar to the two ConvRepl modules. The connections to the two main RAMs for the U, H and Y matrices/vectors are the same. Besides that, the module also makes use of six more smaller RAMs for storing the values bSStart, bSEnd, bSPreEdges, bSPostEdges, bSNumPreEdges and bSNumPostEdges. Like the main arrays, these are also stored in DPRAMs, being that each of the first four made use of a DPRAM of 16 words of 32 bits, while the last only needed to save 8 words of 32 bits.

The interface representation of this module can be seen in the appendix A.3. Because the module never writes values to the six smaller RAMs, all the (...)_we and (...)_data_in ports correspondent to this arrays aren't connected to any DPRAM.

Analysing the waveform of the module (figure 5.6), it is possible to see that the h values are accessed a clock cycle before the u values (looking at the (...)_re signals), while the y values are only stored five clock cycles a after the h vector ceases to be read. Therefore, just like ConvRepl1 and ConvRepl2, one of the memories can be used for both the y and h values.

Four of the six control values are accessed at the same time (refer to the waveforms to the signals (...)_re of figure 5.6) while the other two are accessed at the previous clock cycle. Although only four DPRAM modules could be used for these 6 values, since two of them are required at different instants, 6 RAMs were used. These RAMs were mapped to some of the same addresses of the RAM2, i.e., the RAM used for the Y and H matrices of the ConvConst module.

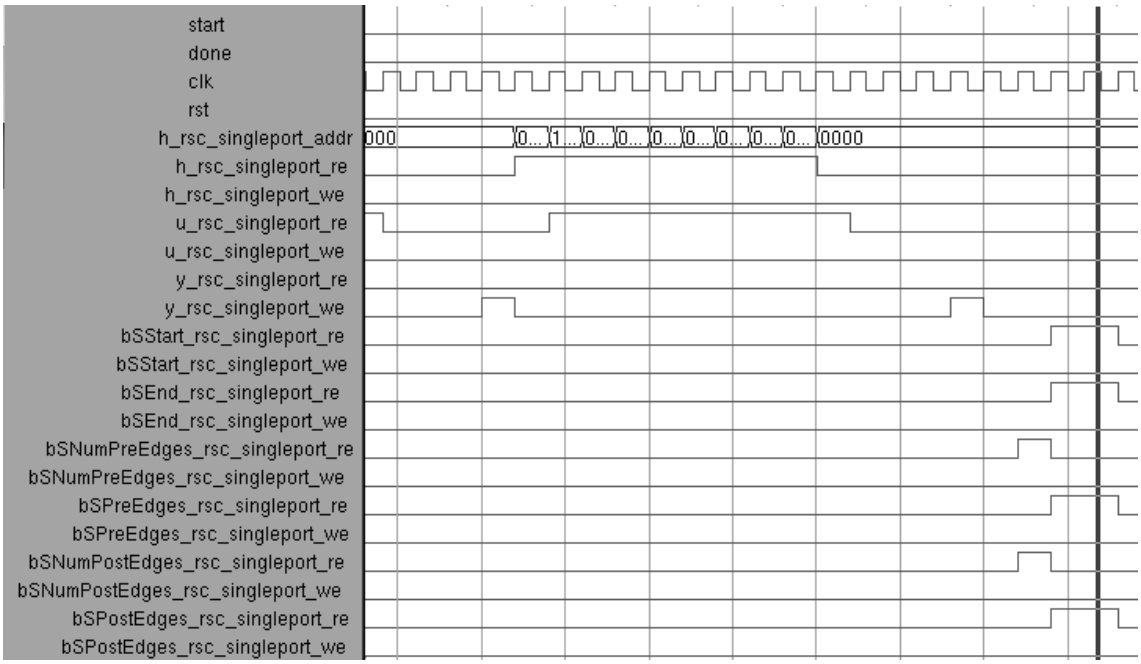

Figure 5.6: Waveform for some of the ConvConst module's outputs.

5.2 DPRAM Module

Each DPRAM is shared by the processor and one of the created modules. Being that every port is duplicated (two DOUT ports, two ADDR ports, two EN ports, etc.), half of the ports will be used by the processor and the other half by one of the modules to control the RAM, as shown in figure 5.7 .

Linking the DPRAM in this way, both the processor (via the PLB) and the connecting module actually "see" the DPRAM as a single port RAM. The advantage of using a DPRAM, is that both its sides can be operated at the same time which eliminates the need to multiplex its input signals.

To store the input and output data of each module, a pair of DPRAMs is used. One memory receives all the values of the U matrix sent by the PowerPC via the PLB, keeping them until requested by its corresponding module. The second memory is used to store both the h values sent by the processor and the Y matrix coming from the module's output.

As said before each element of the used matrices and arrays are composed by 32 bits. The DPRAMs used, were designed so each element had one specific address. Because both the U and Y matrices are composed by 9216 elements (resulting in the same number of required addresses), a minimum of 14 bits is needed to address all their elements in the DPRAM. Since it's common practice to have a potency of 2 as the number of words that a memory has, it was opted to create two DPRAMs with 16384 (2^{14}) words of 32 bits for each module.

For the control arrays of ConvConst, six more smaller RAMs were added to the user_logic

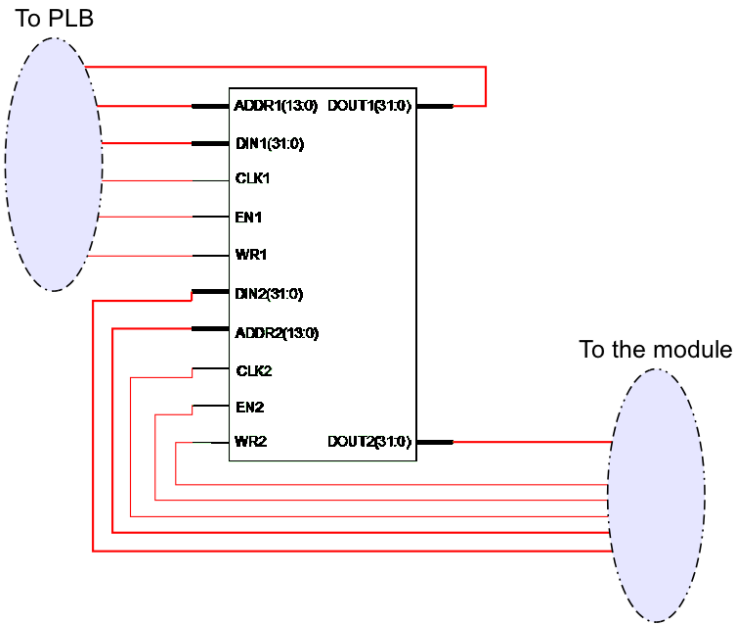

Figure 5.7: RAM's port connections (left ports are inputs, right are outputs).

module. Four of those, were designed with 16 words of 32 bits and assigned to the bSStart, bSEnd, bSPreEdges and bSPostEdges. The two other RAMs, had 8 words of 32 bits and were assigned to bSNumPreEdges and bSNumPostEdges.

As said in the previous section, the Y matrix and the H array are never used at the same time by the same module. For that reason, the same RAM is used to store both this two data sets, but in different addresses. The y values are stored in the RAM starting at address 0 and ending in address 9215. The address code of the final y value is therefore 10001111111111_b and, to avoid overlap the H array can be placed anywhere from that address up.

For the used solution, the H array's data were stored in the RAM starting in the address 14336 (11100000000000_b), as shown in figure 5.8. This required some adaptation of the connecting wires since the addresses that the modules send to the RAM, when requesting for the h values, range from 0 to 11 (or 9 in the case of the ConvConst module), i.e., always start at zero and have only 4 address bits. To extend the number of bits of the H address from four to 14 and also to give the desired offset value, ten more wires were added to the bus that connects the address output of the module to the address input of the DPRAM. Three of added wires (the three msb) were fixed at the high logic level and the remaining seven were fixed to the logic 0 (connected to ground).

Because both the h_rsc_singleport_addr and y_rsc_singleport_addr outputs of the module drive the DPRAM ADDR2 input, a multiplexer is used to select which of the module's outputs are used as the DPRAM's input. The selection bit of the multiplexer was connected to the h_rsc_singleport_re output of the module that, when set to one (meaning that the module is requesting to read an h value from the RAM), will let through the h_rsc_singleport_addr otherwise,

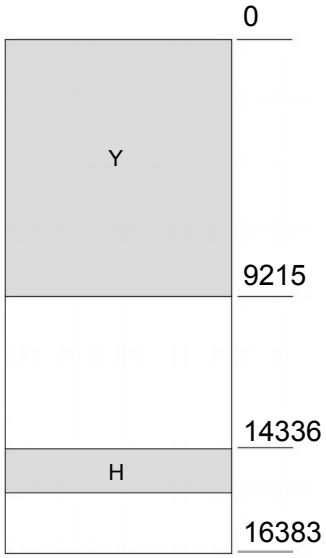

Figure 5.8: Example of one RAM organization.

the y_rsc_singleport_addr is connected to the DPRAM ADDR input. This logic can be seen in figure 5.9

5.3 The user_logic Core

The user_logic core contains the three modules and the 12 DPRAMs (6 for the u, h and y values and 6 for the control arrays of the ConvConst module). Because the PLB can only communicate with one DPRAM memory, a 17-bit address is used to communicate with all the DPRAMs making it appear like one bigger RAM to the bus. The 14 lsb are used to access each word of a given memory, while the 3 msb used to distinguish between RAMs as shown in table 5.1.

Address Bit			**RAM**	**Stored data Structure**
16	**15**	**14**		
0	0	0	1	U of ConvConst
0	0	1	2	Y and H of ConvConst
0	1	0	3	U of ConvRepl1
0	1	1	4	Y and H of ConvRepl1
1	0	0	5	U of ConvRepl2
1	0	1	6	Y and H of ConvRepl2

Table 5.1: Address of each RAM in the user_logic module

If, for instance, the first element of the H array of the ConvRepl1 module is to be accessed by the processor (which is stored in the RAM4), the address value would be $011\_111\_00000000000_b$.

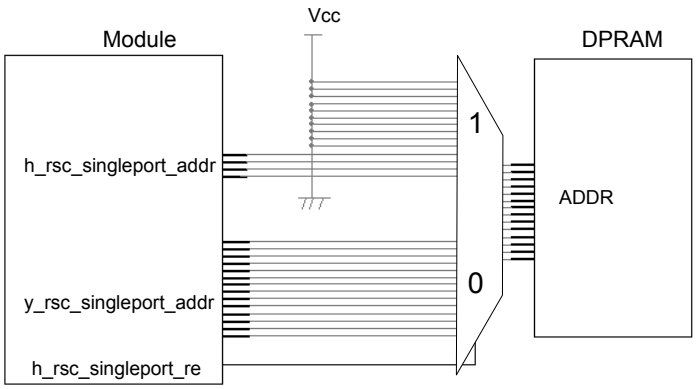

Figure 5.9: Interface between a module and the DPRAM.

The first three bits(011_b) are the RAM selecting bits and the following three (111_b) are the offset used for the H array.

These address bits were connected to the enable ports of each RAM through AND gates along with the Bus2IP_CS (Refer to figure 5.10 for an example). If the three msb (seven for the case of the smaller RAMs) of the address coming from the PLB correspond to the number of the RAM and Bus2IP_CS is at logic 1, that RAM will be enabled and, altering the input values will only take effect on it. So, if the PLB is moving data to RAM5, the 3msb of the address will be 101_b, this will activate that RAM while all the others are deactivated and, although the buses of DIN, ADDR, etc. are connected between all the RAMs and the PLB, only RAM5 will "listen" to these stimulus.

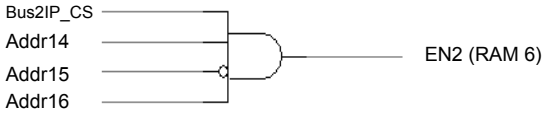

Figure 5.10: Example of how the DPRAM6 is enabled.

To simplify how the six smaller RAMs would be accessed by the PLB, they were mapped to some of the same addresses of the RAM2. The processor stores data in them just the same way as it would store the h values of ConvConst however, these values are stored in 6 individual RAMs that will be used by the ConvConst module. The base addresses seen by the PLB for each of these small DPRAMs is as shown in table 5.2. Note that the 3 msb are 001_b meaning that they are mapped to the RAM2.

The organization of this RAMs and how they are put together to create the RAMX6 can be seen in appendix A.4.

Address Bit							Stored Data Structure
16	**15**	**14**	**13**	**12**	**11**	**10**	
0	0	1	1	0	0	1	bSStart
			1	0	1	0	bSSEnd
			1	0	1	1	bSPreEdges
			1	1	0	0	bSPostEdges
			1	1	0	1	bSNumPreEdges
			1	1	1	1	bSNumPostEdges

Table 5.2: Address of each RAM of the ConvConst module.

There was still the need to define how the *start* and *done* ports of each module would be accessed. The core that was automatically generated by XPS made use of height 32-bit registers that could be accessed from the outside. Four of them are used to store the data coming from the input Bus2IP_Data, while the other four can send their stored values to the IP2Bus_Data output. The input bus Bus2IP_WrCE selects which register will save the value of the Bus2IP_data, while the bus Bus2IP_RdCE chooses which register will send its value to the IP2Bus_Data. By connecting six of these eight registers to the the modules, the *start* and *done* ports could be accessed from the outside.

This registers are connected to the modules in the way described in table 5.3

Reg	**Connected to pin**
In0	*start* of ConvConst
In1	*start* of ConvRepl1
In2	*start* of ConvRepl2
In3	Not connected
Out4	*done* of ConvConst
Out5	*done* of ConvRepl1
Out6	*done* of ConvRepl2
Out7	Not connected

Table 5.3: Register interface with the modules.

Having the my_ip core been completely defined and tested using ModelSim to simulate all the input, output and internal signals, it could be synthesized to produce a netlist that could be used by XPS. This netlist describes the logic components used by the module (logic gates, look up tables, etc.) and their connecting paths.

Finally, the netlist of the my_ip core was added to the reference deign used in XPS where the bitstream for configuring the FPGA at start time was created.

5.4 Software-Hardware Interface

With the user_logic core defined, it was time to integrate it with the software application. For that, the the original ConvConst, ConvRepl1 and ConvRepl2 software functions were deleted and re-edited so that they would only make use of the designed modules. This involved sending the required u and h values to the module's RAMs, send a start signal, wait for the done signal and retrieve the y values from the other module's RAM memory.

To establish an hardware-software communication, Linux device drivers were used. These, are software functions specific to the operating system that allow high level programs in the user space to communicate with the hardware peripherals. In other words, the device driver acts as a translator between the application and the hardware, which for the case at hand will be useful to communicate to the user_logic core. To gift the operating system with the ability to communicate with the designed core, it was created a kernel module associated with the address of the core that was previously defined in XPS.

To send a *start* signal to one of the target modules, it is needed to send that information to the register that is connected to its input. The stimulus are sent by the processor by including in the application the function setone() . Listing 5.1 shows the header of this function.

Listing 5.1: setone function header

```
setone(register,value);
```

Where "register" is the identifier of the desired register to send the value present in "value". In the case of sending a *start* signal to the module ConvRepl2, the register 2 shall have a logic 1 in its lsb so, setone(2,1) is used to start the module.

To check when one of the target modules has finished executing, the function getone() (listing 5.2) is used inside a loop. This function gets the value of a register and can be used to determine the value of the done output of the modules.

Listing 5.2: getone function header

```
value=getone(register);
```

To read and write to the DPRAMs of the my_ip module (which is seen as only one), the functions setm and getm are used (listings 5.3 and 5.4).

Listing 5.3: setm function header

```
setm(number_of_elements, starting_address, array);
```

Listing 5.4: getm function header

```
array=getm(number_of_elements, starting_address);
```

Where "array" is the name of the array that is used inside the C application that will be passed to/retrieved from the RAMs, "staring_address" is the first position where the array will be stored

in the RAMX6 defined for the core module and “number_of_elements” is simply the number of elements that is desired to write to/read from the RAM.

By using these functions, the Software running in the PowerPC could communicate with the hardware modules. Since the algorithms of the functions ConvConst, ConvRepl1 and ConvRepl2 are now substituted by the modules with the same name, all the algorithm present inside them was substituted by the appropriate functions to communicate with the modules. The listings 5.5, 5.6 and 5.7 have a small portion of the code that was included in each function

Listing 5.5: example of the new ConvConst function.

```
setm(16,25600,bSStart);
setm(16,26624,bSEnd);
setm(16,27648,bSPreEdg);
setm(16,28672,bSPstEdg);
setm(8,29696,bSNumPreEdges);
setm(8,31744,bSNumPostEdges);
setm(9216, 0, u);
setm(9, 30720, h);
setone(0,1);              //Sends a start signal to ConvConst
while(getone(4)!=0){}     //Waits until done is at 0
setone(0,0);              //Lowers the start signal
while(getone(4)==0){}     //Waits until the done signal
                          // is at one again
getm(9216, 16384 , y);    //Retrieves the value present in the
                          //DPRAM to the code's variable
```

Listing 5.6: example of the new ConvRepl1 function.

```
setm(9216, 32768, u);
setm(11, 63488, h);
setone(1,1);              //Sends a start signal to ConvRepl1
while(getone(5)!=0){}     //Waits until done is at 0
setone(1,0);              //Lowers the start signal
while(getone(5)==0){}     //Waits until the done signal
                          // is at one again
getm(9216, 49152 , y);    //Retrieves the value present in the
                          //DPRAM to the code's variable
```

Listing 5.7: example of the new ConvRepl2 function.

```
setm(9216, 65536, u);
setm(11, 96256, h);
setone(2,1);              //Sends a start signal to ConvRepl2
```

```
while(getone(6)!=0){}    //Waits until done is at 0
setone(2,0);             //Lowers the start signal
while(getone(6)==0){}    //Waits until the done signal
                         // is at one again
getm(9216, 81920, y);    //Retrieves the value present in the
                         //DPRAM to the code's variable
```

These changes to the target functions were tested again and the obtained results, making use of the hardware modules and the my_ip core, were equal to the original ones. It was time to see the obtained acceleration degree.

5.4.1 Profiling the Application

With the application running aided by the hardware modules, the overall system was profiled to calculate the obtained degree of acceleration. Once again it was used powerpc-linux-gprof to profile the application. The first function to be profiled was the ConvConst which, as said before, had all its original operations replaced by algorithms to control the hardware module.

The target functions were redesign to make use of the functions setm(), setone(), getm() and getone() to interact with the hardware modules. The functions setm() and setone() made themselves use of the pwrite() function, while the getm() and getone() made use of the pread() function. These functions were considered by the profiler as spontaneous and, therefore, it was not possible to know which of their parent functions called them. Because of this, it was not possible to determine how much time was spent by the pread() function when called by the getm() or the getone() function. If the getone() function execution time could not be identified, it was not possible to determine the time the hardware modules took. So, the functions used to retrieve and store data to the memory had to be removed from the altered ConvConst function to determine how much time it was spent by the modules. With this, it was known that the time spent by pread() was relative only to the function getone() while the time spent from pwrite() was only realative to the setone().

The same problem would appear if the application made use of more than one hardware module: because both the three modules' *done* flags were read by the getone() function, it was not possible to know how much individual time was spent by this function when reading a given module. For that, it was opted to execute the application with the ConvConst function prepared to call the respective module while the ConvRepl functions were left has they originally were, i.e., executing completely as software. Having only one part of the application where the pread() function was called, made it possible to determine the module's execution time, since this execution time was equal to the execution time of getone() function and, looking at listing 5.5, it is possible to note that the module's throughput time is approximately the same as the time spent in the getone() function.

These changes would not produce a working application with the desired results but nevertheless, it enabled to retrieve the actual execution time of the hardware modules. The profiling results of the ConvConst function now making use only of the functions to start the module and detect

the end of the module's execution (which will translate in the hardware module's execution time) and a comparison of the function that was used to create the hardware module (the ConvConst function making use of int datatypes) can be seen in the table 5.4.

Optimization Level	Original (secs)	Hardware Module (secs)	Times Called	Speed-up
No optimization	7,66	1,44	960	5,32
-O1	2,22	1,44	960	1,54
-O2	1,83	1,44	960	1,27
-O3	1,26	1,44	960	0,88

Table 5.4: Relative speed-up obtained by the ConvConst hardware module when compared with its software homologous (ten executions).

The ConvConst module took relatively the same time as the value predicted in 4.13 (1,44s/960 = 1500us). This module did not provide a serious acceleration when compared to the software function with the same name. In fact, with the -O3 optimization flag, the software function took less time to execute in the processor.

The same procedure was made to the ConvRepl1 function: only the signals to start the module and to check when its was finished were used. Once again, the other two functions had to be operating has software module so that their pwrite() and pread() functions did not interfere with the ones needed to determine the module's execution time.

Optimization Level	Original (secs)	Hardware Module (secs)	Times Called	Speed-up
No optimization	451,66	17,42	1440	25,93
-O1	91,99	17,42	1440	5,28
-O2	78,41	17,42	1440	4,50
-O3	59,57	17,42	1440	3,41

Table 5.5: Relative speed-up obtained by the Convrepl1 hardware module when compared with its software homologous (ten executions).

As it can be seen in table 5.5 an high degree of acceleration was obtained when the hardware module is compared with the function that was used in Catapult to generate it (the function using soft floating-point datatypes).

The final hardware module to be profiled was the ConvRepl2. The same changes made for the previous modules were proceeded.

Being the ConvRepl modules very similar, the obtained speed-up factor of the ConvRepl2 module is itself very similar to the ones obtained for the ConvRepl1 module (table 5.6).

Optimization Level	Original (secs)	Hardware Module (secs)	Times Called	Speed-up
No optimization	451,23	17,84	1440	25,29
O1	92,00	17,84	1440	5,16
O2	78,53	17,84	1440	4,40
O3	59,57	17,84	1440	3,34

Table 5.6: Relative speed-up obtained by the Convrepl2 hardware module when compared with its software homologous (ten executions).

These individual speed-up times will, as expected, contribute to a faster final application. The obtained total execution times for each optimization level can be seen in the table 5.7.

Optimization level	Execution time (secs)
No optimization	145,46
-O1	105,55
-O2	108,26
-O3	107,59

Table 5.7: Total execution time of the application making use of hardware modules (ten executions).

When analysing the profiling results of the total Stereo Navigation application it was noted that most of the time was spent in the pread() function. As said before, this function is used for both reading the done signals of the hardware modules and the RAMs for their outputted values (RAM2, RAM4, RAM6). Since the time spent by the pread() function when it was used by getone() was already accounted for as the execution time of the hardware modules, it was subtracted from the overall time spent. The resulting values are a reflex of the time that was spent to read the output RAMs, i.e, the time spent by the getm() function. The execution time of these functions, along with time spent by the setm() is present in the table 5.8.

Optimization Level	Setm() (secs)	Getm() (secs)
No optimization	5,04	5,44
O1	4,27	5,97
O2	5,14	6,94
O3	5,11	6,06

Table 5.8: Total time spent to read and write to the core's memories (ten executions).

These actions of reading and writing to the RAMs consume a relatively high percentage of the application run time, ranging from 7,2% to 11,16% of the total. This results make the final application a little slower and should be taken into account.

Recalling the results of the original application in the four optimization scenarios shown in the subsections 3.3.1, 3.3.2, 3.3.3 and 3.3.4, the execution time for the application using the soft floating-point datatypes (table 4.10) and the results of the total application's execution time from the table 5.7, a comparative analysis about the relative acceleration could be made (as shown in the table 5.9). In the table 5.9, the column "hardware aided" is relative to the software application with hardware acceleration; "Soft & int" to the solution with the ConvConst function using only int datatypes and ConvRepl functions using the soft floating-point datatypes; column "Original" has the total application running time of the original algorithm.

	Total execution time			**Relative execution time**	
Optimization Level	**Hardware aided**	**Soft & int**	**Original**	**Hardware /Soft**	**Hardware/Original**
No optimization	145,46	1083,22	271,58	13,43%	53,56%
-O1	105,55	262,92	135,36	40,15%	77,98%
-O2	108,26	234,87	135,01	46,09%	80,19%
-O3	107,59	190,91	134,30	56,36%	80,11%

Table 5.9: Relative execution time of the solutions found (ten executions). Time in seconds.

The columns "relative execution time" of the table 5.9, represent the duration of the solution found with hardware acceleration compared with the original application. The calculus made for the column Hardware/Original was:

$$\frac{timeOfHardwareAcceleratedApplication}{timeOfOriginalApplication}(\%) \tag{5.1}$$

The equivalent calculus was made for the column "Hardware/Soft".

From table 5.9 it can be seen that the best case of the obtained acceleration happens when no optimization to the code is made, giving a relative execution time of 13,43% relative to the total execution time of the application that was redesigned to make use of the soft-float datatypes for the ConvRepl functions and ints for the ConvConst function. When compared with the original application, the solution that was found, made the same operations taking only 53.56% of the execution time (again, in the best case scenario).

The fastest solution took 10,76 seconds to execute which corresponds to a relative execution time of 80,11%.

5.5 Improving the Hardware Acceleration

In this section, some alternatives to improve the performance of the Stereo navigation application are studied. Some are based in redundancy elimination, while others are based in architecture optimization.

5.5.1 Memory Mapping

Because too much time was being spent with the actions of writing and retrieving the data from the core's RAMs, a faster alternative had to be found. The solution that was found maps the RAMs in the hardware into the user space.

Instead of using the getm() and setm() functions, a page of memory relative to the core module's address is mapped into the user space. This way, when a variable is changed in the hardware side it will be propagated to the software side and vice-versa. Because this solution does not make use of kernel module functions that cause high software overhead, the time spent in this operations is only due to the communication delay between the processor and the core.

This alternative reduced a little the time spent to send/retrieve the values from the RAMs, as shown in table 5.10.

Optimization Level	**Hardware aided (secs)**	**Time memmap (secs)**	**Difference (secs)**
No optimization	145,56	144,43	1,13
-O1	105,55	101,37	4,18
-O2	108,26	102,69	5,57
-O3	107,59	102,63	4,96

Table 5.10: Comparison of the total execution time of the StereoNav application with and without memory mapping (ten executions).

5.5.2 Memory Sharing

Recalling the data flow of the three target functions inside the harrisTile_model_step presented in the figure 3.6, it can be seen that the data that is inputted in the ConvRepl2 function is the direct output of the ConvRepl1 function. This of course, is also true for the hardware modules. Therefore, a single RAM can be used to store the Y matrix of the ConvRepl1 module and feed it to the ConvRepl2 module as its U matrix decreasing the overall used component count of the FPGA.

In the original design, the RAM that stored the y values of ConvRepl1 was a different one that of the RAM for the u values of ConvRepl2. This required that in the final system, after ConvRepl1 finished to write its y values in RAM4, the software application retrieved the data storing it inside a matrix/array and then send it to the RAM of u values of the ConvRepl2 module (RAM5). This puts an unnecessary overhead that would cause delays to the application. By using the same RAM to store this values, both a faster application with less occupied area is obtained. The reason why this makes the application faster is that both one call to the function getm() and setm() is discarded every time harris_Tile_model_step is called

Although shared by the two modules, only one of the RAM's ports is accessed by both, that is

the ADDR port. The bus y_rsc_singleport_addr[2] of the ConvRepl1 module and u_rsc_singleport_addr of the ConvRepl2 had to be multiplexed so that both could drive the ADDR input port of the RAM module. To select which bus would drive the ADDR input port, the u_rsc_singleport_re was used as a select bit for the multiplexer. This bit when set to one would let through the u_rsc_singleport_addr bus, making it drive the ADDR RAM's input port otherwise, it was the y_rsc_singleport_addr that would pass thought the multiplexer to drive the same RAM port (shown in figure 5.11).

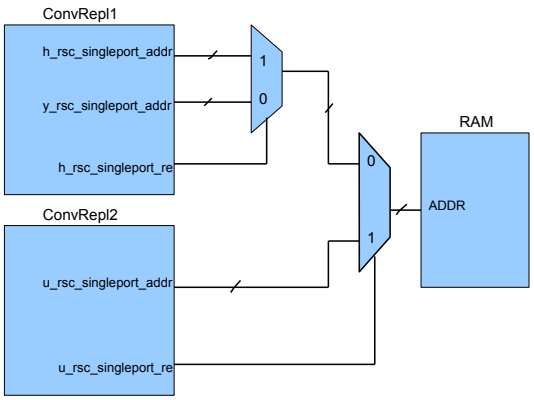

Figure 5.11: RAM shared by ConvRepl1 and ConvRepl2.

The application was again executed, using now the new generated hardware architecture aided with the memory mapping solution. As it can be seen in the table 5.11 as much as 0.71 seconds can be eliminated in an application that took 10.76 seconds by sharing one of the memories (and therefore eliminating one getm() and setm() function).

Optimization Level	Hardware aided (secs)	Time 5 Mems(secs)	Difference (secs)
No optimization	145,46	139,97	5,49
-O1	105,55	97,96	7,59
-O2	108,26	100,86	7,40
-O3	107,59	100,45	7,14

Table 5.11: Comparison of the total execution time of the StereoNav application with 6 and 5 memories (ten executions).

[2]Although called y_rsc_singleport_addr, this signal is in fact the multiplexed signal derived from y_rsc_singleport_addr and h_rsc_singleport_addr presented in the end of the section 5.2.

5.5.3 Including the Constant Values Inside the Modules

As said previously, the h values used by the three modules are in fact constant values. There is no need to keep them in external memories where the module would have to access them so, the H arrays were included inside the modules. To do this, the application source code was altered, so that the target functions have these constants declared inside them instead of receiving the h values as their arguments.

This small changes in the functions were then processed by Catapult C to generate the equivalent modules. Again the modules designed around 100MHz were chosen to integrate the user_logic core. The timing results for each of these modules is present in the table 5.12.

Hardware Module	Freq. in XST (MHz)	Normalized frequency (MHz)	Latency Cycles	Throughput time from XST(us)	Throughput time for normalized frequency(us)
ConvConst	144,702	100	80265	554,69	802,65
ConvRepl1	103,767	100	1346113	12972,46	13461,13
ConvRepl2	102,302	100	1446145	14136,04	14461,45

Table 5.12: Throughput time obtained for the modules using internal constants.

Because these modules take less logic to address the constants that were, in the previous version, in external memories, they could be designed by Catapult to have less latency cycles. Making this modules operating at the same frequency as the previous modules will make them execute much faster giving place to a faster final application.

The modules were included inside the same user_logic core defined in the subsection 5.5.2. Because they do not have any h values to read from the RAMs the logic interconnects (the multiplexers defined to choose if the H array is to be read from the RAM or if the Y matrix is to be stored) that connect the modules to its seconds RAMs (RAM2, RAM4 and RAM5) could be deleted, making the final module simpler. After a general simulation of the produced module, its bitstream could be generated to configure the FPGA.

The software part of the application could also be simplified. There was no need to use the functions to send the h values to the FPGA's RAMs. This alone could also provide a small acceleration degree. Once again the application produced the expected results and at a faster pace, as shown in the table 5.13.

This time, the application was accelerated due to the optimization of the modules instead of redundancy elimination. Aided by the memory sharing and mapping solutions used in the subsection 5.5.2, the modules with the included constant values eliminated a maximum of 0,9 seconds of a 10,83 seconds application, almost a 10% drop in execution time. The minimum execution time is present at the -O3 optimization flag, where one execution of the application would take approximately 9,87 seconds.

Optimization Level	Hardware aided (secs)	Time Const Inside(secs)	Difference (secs)
No optimization	145,46	137,96	7,5
-O1	105,55	96,56	8,99
-O2	108,26	99,17	9,09
-O3	107,59	98,72	8,87

Table 5.13: Comparison of the total execution time of the StereoNav application with five memories and the constant h values inside the hardware modules as opposed to the application making use of six memories (ten executions).

Comparing this solution with both the original and the soft floating-point software application running on the PowerPc processor (table 5.14) it can be seen that the highest degree of acceleration is obtained when no optimization is made to the application making use of the soft float solution, the designed hardware/software hybrid takes only 12,74% of the execution time. In the fastest scenario, i.e, with the -O3 optimization flag, the application takes 73,51% of the execution time of the original application.

	Total execution time			Relative execution time	
Optimization Level	Hardware aided	Soft & int	Original	Hardware /Soft	Hardware/Original
No optimization	137,96	1083,22	271,58	12,74%	50,8%
-O1	96,56	262,92	135,36	36,73%	71,34%
-O2	99,17	234,87	135,01	42,22%	73,45%
-O3	98,72	190,91	134,3	51,71%	73,51%

Table 5.14: Relative execution time of the solutions making use of the included constants (ten executions).

5.5.4 Using Parallel Modules

The first alternatives shown up until now, used an approach where the convConst, convRepl1 and convRepl2 functions' algorithms were replaced by the actions needed to execute the hardware modules and retrieve the data of the resulting operations. These alternatives made the final application faster because the resulting functions (the functions making use of the hardware modules) executed faster. This however, does not exploit one of the strongest characteristics of the hardware implementation, which is the parallelization of independent operations.

Recalling the dataflow representation of figure 3.6, it can be seen that there are 8 phases of processment inside the harrisTile_model_step function. Making use of hardware modules, there is no reason why the convConst module can't be executing at the same time of the convRepl1 or convRepl2 module, since they are independent.

To use the three designed modules in parallel, some constraints had to be met in the design phase of the new user_logic core, those are:

- A module can only start to execute if its preceding module has finished. For instance, convRepl1 can only start after convConst has finished executing;
- Two instances of the same module can't be executing at the same time.
- Different modules can't access to the same RAM at the same time

There was also a constraint of using only one instance of each hardware module (i.e, not using repeated hardware modules) to save FPGA final area. This new alternative made possible a new system configuration where the same operations could be made in 5 phases as opposed to the initial 8 phases present in figure 3.6. The phases of this solution are shown in figure 5.12. As it can be seen, two modules can be operating at the same time. If more instances of the modules were used, the same operation could be made in 4 or even 3 phases.

It is important to note that the operations that are represented by circles in the figure 5.12 are carried out by the processor. This causes some delay because the output values of convConst that were stored in the FPGA's DPRAMs (RAM2 and RAM3) must be read to the software space where the multiplication will be proceeded and, after that, the results will be sent, again, to the FPGA's DPRAMs.

To meet this new architectural constraint, the three used modules are the same that were presented in the previous subsection (subsection 5.5.3). There was however the need to alter the user_logic core's architectural design to have in mind the new connections between the modules and the RAMs since now, one module can access more than one memory in both the writing and reading operations and, more than one module access the same memory. These changes to the core were made by using multiplexers in all the modules' and RAMs' inputs. These multiplexers used as the selector bits, registers that represented the 5 phases. For instance, for the Addr input of the RAM2, both the convConst and the convRepl2 modules use it to indicate the address that they will use to store their y values; also, convRepl1 will use this input to indicate the address where its u values will be read. So there are three output signals that will drive the addr input of the RAM2. By using a multiplexer just before the addr input of the RAM, it can be selected which module will drive the RAM's input: convConst in phase 1, convRepl1 in phase 2 and convRepl2 in phase 4. The same logic is used for everyone of the other inputs of both the modules and the RAM in the modules-RAMs interface.

With the new solution validated with modelsim, it could be synthesized and its bitstream could be transfered to the ML507 development board. To attend to the changes made to the user_logic core, the harrisTile_model_step function had to be altered so it could make use of two modules in parallel. A small flowchart of the software source code operations to control the modules can be seen in figure 5.13 (the actions to write/read from the memories are not shown as they are the same presented earlier in this section).

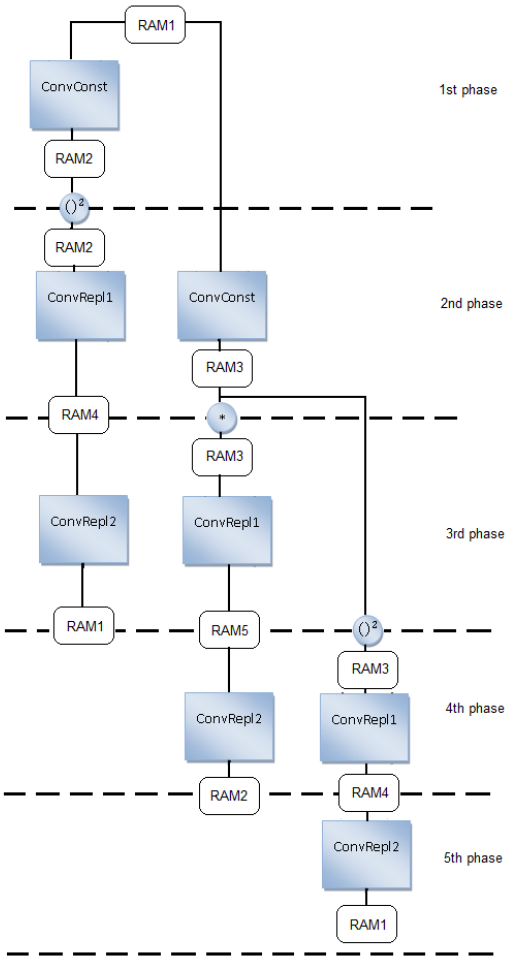

Figure 5.12: Data and timeflow of the hardware solution making use of parallel modules.

After the the new solution was validated, it was time to, once again, profile the application to determine the speed-up factor. As it can be seen in the table 5.15, the total execution time could be reduced to 8,4 seconds per execution of the application, representing only 62,06% of the time that the original Stereo Navigation application took. When comparing with the application using the software functions that gave place to the hardware modules, i.e., the application with the soft float datatypes, it can be seen, also from table 5.15, that a greater speed-up was obtained. This was the best solution found thus far on this thesis and the one that concludes the work done to optimize the original Stereo Navigation application.

5.6 Comparing the Hardware Solutions

The Xilinx ISE is capable to determine how much of the FPGA's resources are used by a given module when it is implemented. This information is obtained through the place and routing process. As it is implied this process consists of two tasks: in the placement task it is defined where

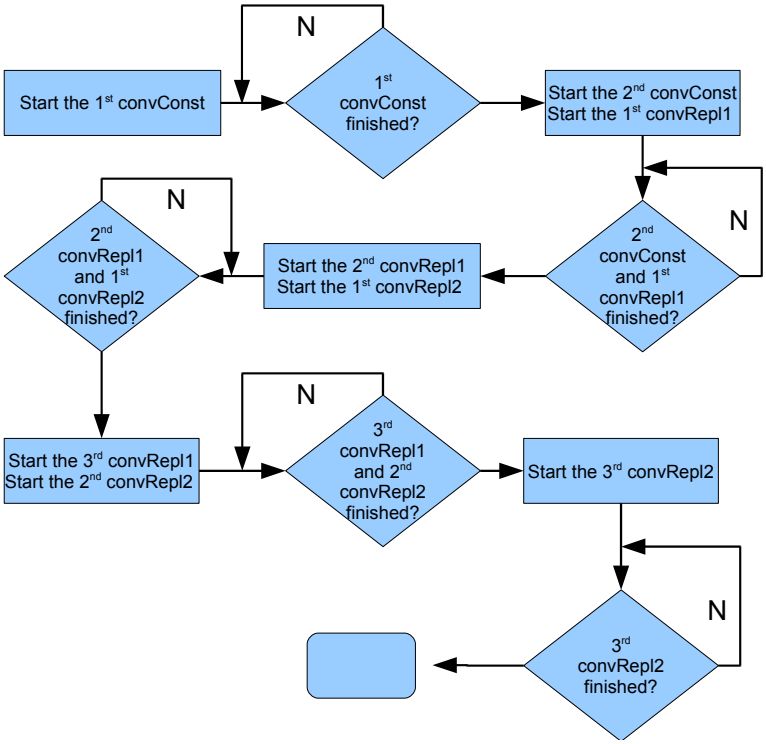

Figure 5.13: Flowchart of the software control operations.

to place all and logic elements in the reconfigurable fabric while, the routing task is where it is decided the exact design of all the wires needed to connect the placed components. This measurement is very important for the project because it gives the information about all the the resources used by the functional components of the designed core as well as all its interconnects. The post place and route characteristics of the four different hardware alternatives for the user_logic are in table 5.16 where “Original” is relative to the original solution found in 5.3, “5 mems” to 5.5.2 and “Constant h” to the 5.5.3 and “Parallel” is referent to the solution found in 5.5.4.

The solution “Constant h” is the most area efficient. Because it makes use of 5 memories and because its h values are internal to the modules, both memory and logic space are reduced. The parallel solution uses a little more logic, since every connection between the DPRAMs and the modules had to be through multiplexers.

	Total execution time			Relative execution time	
Optimization Level	**Hardware aided w/ parallel modules**	**Soft & int**	**Original**	**Hardware /Soft**	**Hardware/Original**
No optimization	125,49	1083,22	271,58	11,58%	46,21%
-O1	84,00	262,92	135,36	31,95%	62,06%
-O2	84,90	234,87	135,01	36,15%	62,88%
-O3	88,99	190,91	134,30	46,61%	66,26%

Table 5.15: Relative execution time of the solutions found using the hardware modules in parallel (ten executions).

	Original	**5 mems**	**Constant h**	**Parallel**
Number of Slice Registers used as Flip Flops	13046	13046	12359	12371
Number of slice LUTs	24487	24490	23916	24194
Number of occupied Slices	8395	8596	8383	8239
Number of LUT Flip Flop pairs	26426	26547	25812	25924
Number of BlockRAM/FIFO	102	86	86	86
Number of 36 k BlockRAM used	6	6	6	6
Number of 18 k BlockRAM used	192	160	160	160
Total Memory used	3672	3096	3096	3096
Number of DSP48Es	17	17	17	17

Table 5.16: Comparison of the place and route characteristics, in a Virtex-5 FPGA, for the four solutions found.

Chapter 6

Conclusion

6.1 Conclusion

Throughout this project, some hardware alternatives to accelerate a software by deriving an HDL from C-language using Catapult C were shown. After being identified the application's most critical functions, the hardware modules created specifically to replace them have indeed accelerated the Stereo Navigation application running on the Vitex-5 FPGA with the embedded PowerPC processor. The three hardware modules that were created and operating at a frequency of 100MHz were used to aid the rest of the software application. This solution proved to take only 66,26% of the execution time (or executing 1,5 times faster) of the same application executing solely in the software format in the PowerPC processor at 400MHz. This results proved that providing hardware acceleration to a software application can improve the system's performance.

This speed-up factor between software and hardware could be further improved by taking some care in the development of the software application. Such an example was the use of the floating point variables inside the convConst function where they were not needed. Also, for the convRepl functions, because these made use of the floating point datatypes, some performance in the final solutions was lost. By using other kind of variables such as fixed point, the conversion from software to hardware could produce improved modules.

The soft floating-point solution found to work around the no floating-point problem inherent from Catapult C, did make the hardware acceleration less effective when comparing the multiplication and addition modules generated by Catapult with the the floating-point ones designed specifically to the Virtex-5 FPGA by the CoreLogic wizard of the Xilinx ISE.

It was determined that the operations to read and write to the user_logic's DPRAMs took a considerable percentage of the application's execution time and therefore, should be reduced to the minimum possible.

Including the constant values inside the hardware modules, provided Catapult C the ability of creating simpler and faster solutions.

6.2 Future Work

There could be some improvements to enhance the final system.

One improvement would be the use of duplicated hardware modules in order to decrease the number of execution phases from five (as shown in the subsection 5.5.4) to four, or even three. There could also be investigated some alternative designs created by Catapult C by setting some architecture specifications like, pipelining and loop unrolling.

Regarding the data types, some further work could be made to study how the fixed point alternatives studied here would adapt to the application an infer also how much acceleration degree could be obtained. Also, it should be made an attempt to create modules that would use the the floating-point multiplication and addition modules provided by the CoreLogic functionality of Xilinx ISE that are targeted to the Virtex-5 FPGA.

One of the main problems that was found when using Catapult C, was its inability to create modules that could operate over the 200MHz frequency. This could be due to incorrect definition of parameters and could be an area of interest to explore in the future.

To further improve the application, other software functions could be converted to hardware, like the harrisTile_model_setp function.

Finally each and every hardware module could make use of more input RAMs (9 for the ConvConst module and 11 for the ConvRepl) enabling various u elements to processed at the same time.

Appendix A

Appendix

A.1 Output of the Original Application

main(): start
...
===== FRAME=59 ==========================
cmp=0 do_sort L: found 139 features
cmp=0 do_sort R: found 116 features
cmp=0 do_match_lr: found 34 matches
===== FRAME=60 ==========================
cmp=0 do_sort L: found 131 features
cmp=0 do_sort R: found 117 features
cmp=0 do_match_lr: found 40 matches
cmp=0 L do_match_t1t2: found 29 matches
cmp=0 R do_match_t1t2: found 30 matches
cmp=0 do_circular_check: found 22 matches
cmp=0 do_reproj: reprojected 22 points
cmp=0 do_ransac: n_inliers_ransac=10.000000, pass_ransac=1.000000, n_inliers_joint_2=10.000000, n_inliers_final_2=10.000000
Vector dR: 001.00 000.01 -00.00 -00.01 001.00 -00.01 000.00 000.01 001.00
Vector dT: 000.56 000.01 000.00
===== TERMINATE ==========================
...
main(): finish

A.2 Output of the Application Using the 32-bit Fixed-point Solution

```
main(): start
...
===== FRAME=59 ==========================
cmp=0 do_sort L: found 140 features
cmp=0 do_sort R: found 117 features
cmp=0 do_match_lr: found 34 matches
===== FRAME=60 ==========================
cmp=0 do_sort L: found 132 features
cmp=0 do_sort R: found 116 features
cmp=0 do_match_lr: found 39 matches
cmp=0 L do_match_t1t2: found 29 matches
cmp=0 R do_match_t1t2: found 29 matches
cmp=0 do_circular_check: found 22 matches
cmp=0 do_reproj: reprojected 22 points
cmp=0 do_ransac: n_inliers_ransac=10.000000, pass_ransac=1.000000, n_inliers_joint_2=10.000000,
n_inliers_final_2=10.000000
Vector dR: 001.00 000.01 -00.00 -00.01 001.00 -00.01 000.00 000.01 001.00
Vector dT: 000.56 000.01 000.00
===== TERMINATE ==========================
...
main(): finish
```

A.3 ConvConst module

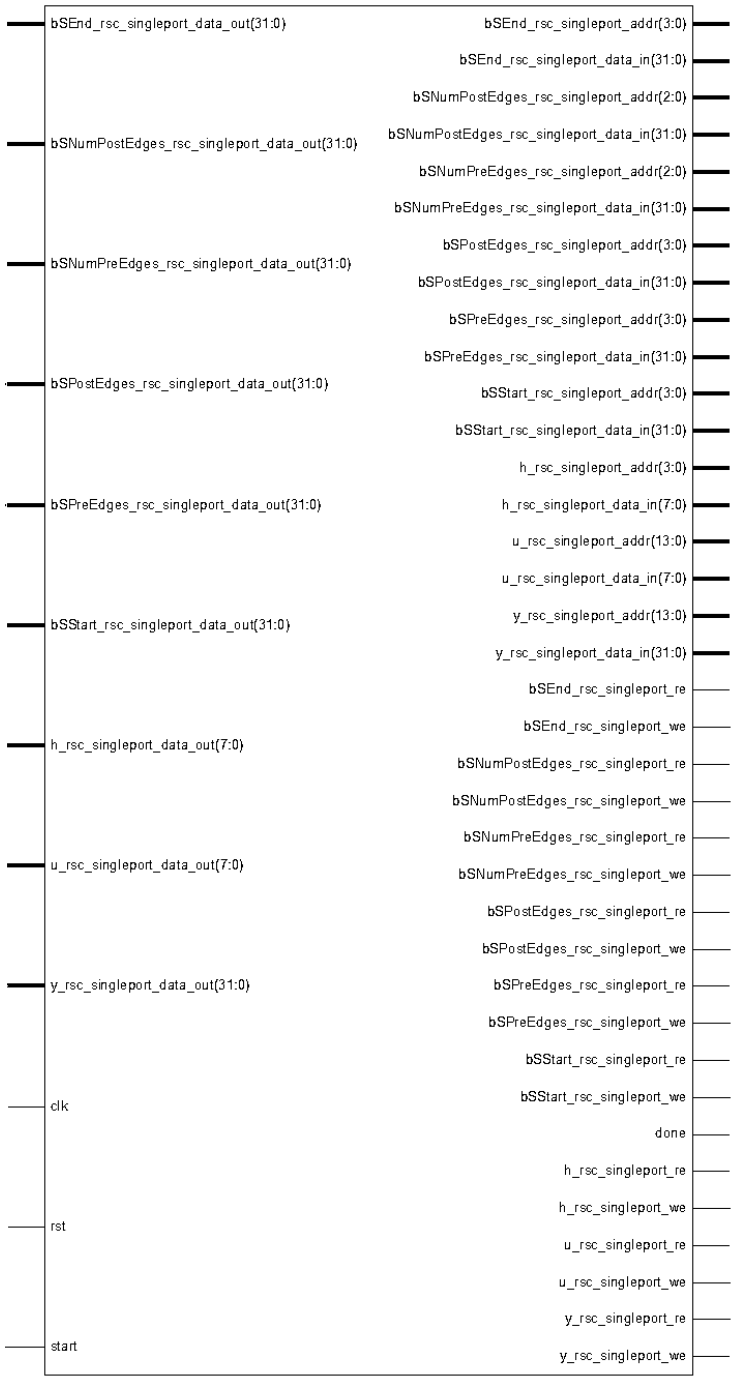

Figure A.1: Interface representation of the ConvConst module (left ports are inputs,right are outputs).

A.4 Memory Organization

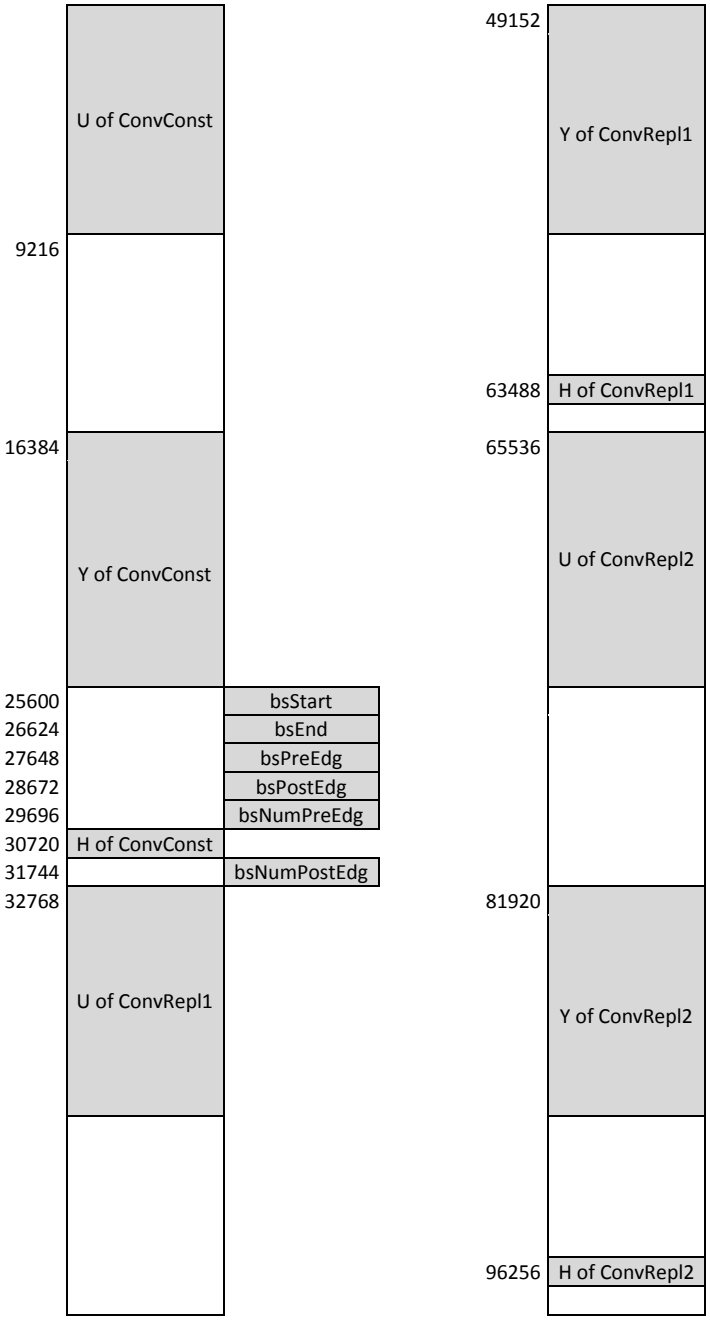

Figure A.2: Memory organization for the user_logic core.

A.5 Configure the Powerpc Linux Kernel in the Xilinx ML507 Development Platform

This appendix section specifies the installation steps to build the PowerPC Linux Kernel source code and run it on an ML507 Xilinx development board. Most of the steps were followed from [25] and [33].

The first step is to download the ppc-2008-04-01.iso file. After obtaining this file, a virtual drive shall be created.

sudo mkdir -p /media/virtual

After that, the directory where the iso file is located, must be accessed. There, the image file shall be mounted by inputting the following command:

sudo mount –o loop –t iso9660 ppc-2008-04-01.iso /media/virtual

The directory /media/virtual will now have an emulated version of the installation DVD. Now, the files can be installed with the command:

sudo ./install –d /opt/dev/ELDK/4.2 ppc_4xxFP

After changing to the /opt/dev/ELDK/4.2 directory:

source eldk_init 4xxFP
sudo /media/virtual/ELDK_FIXOWNER –a ppc_4xxFP
sudo /media/virtual/ELDK_MAKEDEV –a ppc_4xxFP

With these steps, the tools for cross-compiling the software application are ready to be used. Among them is the powerpc-linux-gcc and the powerpc-linux-gprof.

After obtaining the reference design from [34] (user account required), a device tree must be downloaded. This device tree is used to describe the hardware so that the kernel can be configured during boot. The device tree can be obtained using git:

git clone git://git.xilinx.com/device-tree.git

After that, the bsp folder in the device-tree must by joined with the folder with the same name in the reference design. With this done, it's time to run the XPS software. In there, it shall be open the downloaded reference design. By selecting "software platform settings", the window in figure A.3 shall open. In the OS drop-down menu the device-tree shall be selected.

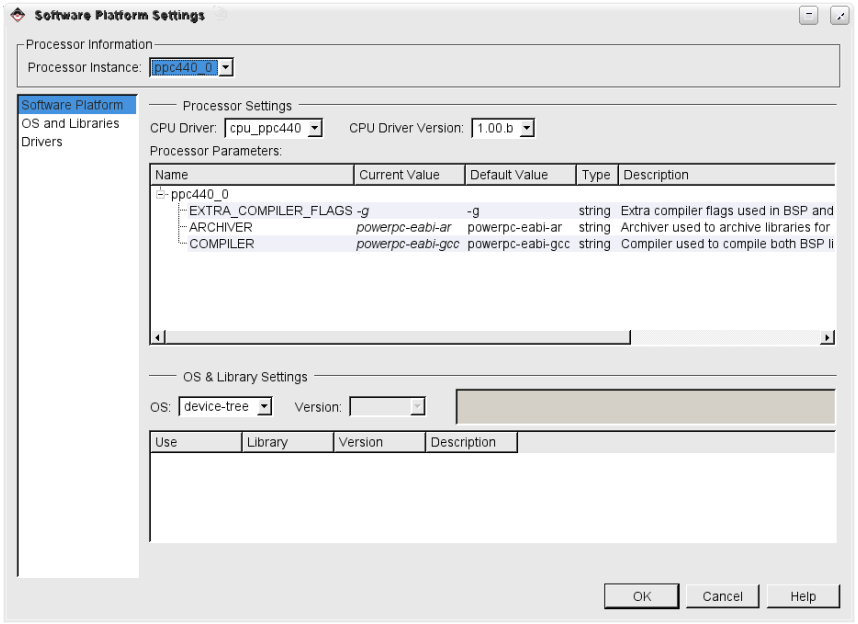

Figure A.3: Software platform settings.

On the left menu, selecting the "OS and Libraries" will open a new tab. There, lines shall be changed as shown in A.4:

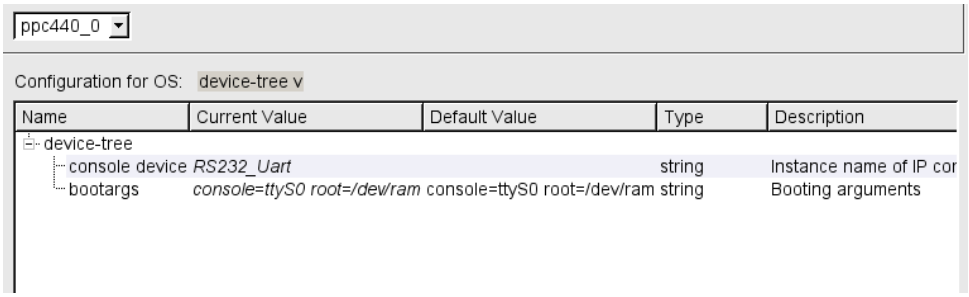

Figure A.4: Changes to be made.

In the main XPS window, click on “Generate libraries and BSP’s”. A file called Xilinx.dts will be created inside the project in the directory ppc440_0/libsrc/device_treev0_00_x. Click on "update bitstream" to generate the download.bit file in the implementation folder inside the project.

Now the linux-2.6-xlnx kernel can be download by inputting the following command in the command line

git clone git://git.xilinx.com/linux-2.6-xlnx.git

When the download is complete, the Xilinx.dts file shall be copied inside the linux-2.6-xlnx directory in the arch/powerpc/boot/dts directory and rename it to virtex440-ml507.dts. Also, a ramdisk image shall be copied to arch/powerpc/boot (refer to [25] for instructions on how to download and [35] for instructions on how to expand its capacity).

After changing to the linux-2.6-xlnx directory the following commands shall be inputted

Source /opt/dev/ELDK/4.2/eldk_init 4xxFP
Make menuconfig ARCH=powerpc 44x/virtex5_defconfig

In the menu that will appear, “Enable Xilinx Soft FPU” shall be selected. After that, the following line must be inputted to create an *.elf file.

Make ARCH=powerpc simpleImage.initrd.virtex440_ml507

An *.elf file will be created. This file, along with the file download.bit, can be used to create an *.ace file. This file is the one that will configure the development board, at boot time.

Xmd –tcl genace.tcl -jprog -hw «path_to_file_download.bit» -ace my_ace.ace -board ml507 -elf «path_to_elf_file» -target ppc_hw -start_address 0x00800000

Finally, the my_ace file can be transferred to the cfg6 directory of the compact flash disk provided with the development board.

References

[1] Katherine Compton and Scott Hauck. Reconfigurable computing: a survey of systems and software. *ACM Comput. Surv.*, 34:171–210, June 2002.

[2] Honeywell website. Available at http://honeywell.com/Pages/Home.aspx, last time accessed in 4 of October, 2011.

[3] REFLECT project. Available at http://www.reflect-project.eu/, last time accessed in 4 of October, 2011.

[4] William Chun, Yip Lo, Keith Redmond, Jason Luu, Paul Chow, Jonathan Rose, and Lothar Lilge. Journal of biomedical optics 014019 january/february 2009 hardware acceleration of a monte carlo simulation for photodynamic therapy treatment planning.

[5] T.J. Todman, G.A. Constantinides, S.J.E. Wilton, O. Mencer, W. Luk, and P.Y.K. Cheung. Reconfigurable computing: architectures and design methods. *Computers and Digital Techniques, IEE Proceedings* -, 152(2):193 – 207, mar 2005.

[6] V. Betz and S Brown. Fpga challenges and opportunities at 40nm and beyond. 2009.

[7] S.E. Krakiwsky, L.E. Turner, and M.M. Okoniewski. Acceleration of finite-difference time-domain (fdtd) using graphics processor units (gpu). In *Microwave Symposium Digest, 2004 IEEE MTT-S International*, volume 2, pages 1033 – 1036 Vol.2, june 2004.

[8] Kenneth Moreland and Edward Angel. The fft on a gpu. In *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware*, HWWS '03, pages 112–119, Aire-la-Ville, Switzerland, Switzerland, 2003. Eurographics Association.

[9] M.D. Edwards, J. Forrest, and A.E. Whelan. Acceleration of software algorithms using hardware/software co-design techniques. *Journal of Systems Architecture*, 42(9-10):697 – 707, 1997.

[10] S. Hauck, T.W. Fry, M.M. Hosler, and J.P. Kao. The chimaera reconfigurable functional unit. *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, 12(2):206 –217, feb. 2004.

[11] T. Miyamori and U. Olukotun. A quantitative analysis of reconfigurable coprocessors for multimedia applications. In *FPGAs for Custom Computing Machines, 1998. Proceedings. IEEE Symposium on*, pages 2 –11, apr 1998.

[12] R. Ernst, J. Henkel, Th. Benner, W. Ye, U. Holtmann, D. Herrmann, and M. Trawny. The cosyma environment for hardware/software cosynthesis of small embedded systems. *Microprocessors and Microsystems*, 20(3):159 – 166, 1996.

[13] P. Bertin and H. Touati. PAM programming environments: practice and experience. In *FPGAs for Custom Computing Machines, 1994. Proceedings. IEEE Workshop on*, pages 133 –138, apr 1994.

[14] P.M. Athanas and H.F. Silverman. Processor reconfiguration through instruction-set metamorphosis. *Computer*, 26(3):11 –18, mar 1993.

[15] S.A. Edwards. The challenges of synthesizing hardware from c-like languages. *Design Test of Computers, IEEE*, 23(5):375 –386, may 2006.

[16] P. Coussy, D.D. Gajski, M. Meredith, and A. Takach. An introduction to high-level synthesis. *Design Test of Computers, IEEE*, 26(4):8 –17, july-aug. 2009.

[17] Andres Takach, Bryan Bowyer, and Thomas Bollaert. C based hardware design for wireless applications. In *Proceedings of the conference on Design, Automation and Test in Europe - Volume 3*, DATE '05, pages 124–129, Washington, DC, USA, 2005. IEEE Computer Society.

[18] C. Economakos and G. Economakos. Fpga implementation of plc programs using automated high-level synthesis tools. In *Industrial Electronics, 2008. ISIE 2008. IEEE International Symposium on*, pages 1908 –1913, 30 2008-july 2 2008.

[19] XILINX Peter Alfke. Virtex-5 fxt a new fpga platform, plus a look into the future, August 2008.

[20] XILINX. Virtex-5 family overview, February 2009.

[21] XILINX. Ml505/ml506/ml507 evaluation platform user guide, May 2011.

[22] Applied Micro Circuits Corporation. Ppc440 processor user's manual. March 2008.

[23] IBM. Powerpc 440x6 embedded processor core user's manual. September 2009.

[24] ICT-2009-4 REFLECT, April 2009. Deliverable 1.1 "Repository of application from Honeywell".

[25] Xilinx Open Source Wiki. Available at http://xilinx.wikidot.com/powerpc-linux, last time acessed in 28 of July,2011.

[26] David Gibson and Benjamin Herrenschmidt. Device trees everywhere. Technical report, zLabs, IBM Linux Technology Center, February 2006.

[27] DENX. Available at http://ftp.denx.de/pub/eldk/4.2/ppc-linux-x86/distribution/, last time acessed in 28 of July,2011.

[28] GNU gprof, June 2011. Available at http://www.cs.utah.edu/dept/old/texinfo/as/gprof.html, last time accessed in 28 of July, 2011.

[29] Virtex-5 APU Floating-Point Unit v1.01a. March 2011.

[30] Mentor Graphics Corporation. *Catapult C Synthesis C to Hardware Concepts*. October 2009.

[31] IEEE Standard for Floating-Point Arithmetic. *IEEE Std 754-2008*, pages 1 –58, 29 2008.

[32] John Hauser, June 2011. Available at http://www.jhauser.us/arithmetic/SoftFloat.html, last time acessed in 28 of July,2011.

[33] Linux for the Xilinx Virtex4/5 FPGAs. Available at http://npg.dl.ac.uk/MIDAS/DataAcq/EmbeddedLinux.html, last time acessed in 28 of July,2011.

[34] ML507 reference design. Available at https://secure.xilinx.com/webreg/clickthrough.do?cid=111799, last time acessed in 28 of July,2011.

[35] Expanding File System. Available at http://xilinx.wikidot.com/expanding-file-system, last time acessed in 28 of July,2011.