# Hardware Acceleration of a Stereo Navigation Application with High-Level C to HW Synthesis

João Vilela Teixeira[1], José C. Alves[1], João M. P. Cardoso[2], Zlatko Petrov [3]
[1]*University of Porto - Faculty of Engineering / DEEC - INESC Porto*
[2]*University of Porto - Faculty of Engineering / DEI*
[3]*Honeywell International s.r.o, Czech Republic*

## Abstract

*Hardware acceleration is a well established solution to accelerate computational intensive applications, in particular for embedded applications based on performance constrained processors. However, the effort for designing hardware architectures capable of effectively improving the execution time of a software application, makes this approach almost exclusively affordable to experienced hardware designers. This paper describes a case study on how effective an automated high-level synthesis tool can be for the creation of RTL architectures from untimed software algorithms. The high-level synthesis tool CatapultC from Mentor Graphics was used for translating the most time consuming sections of an image based navigation application. The lack of support of floating point arithmetic was identified as a major limitation of this tool, when synthesizing C code that really requires floating point computations. To overcome this limitation, a set of floating point C routines available from an open source floating point emulation C library was used to perform the floating point arithmetic required. Although this approach does not allow to make use of the optimized floating point arithmetic cores provided by the FPGA design tools, it was demonstrated that compiling to hardware C code that implements floating point arithmetic can still lead to acceleration of software applications. With the implementation of these critical functions synthesized by Catapult C, a global speedup of* 1.5 *was measured for the whole application, when compared to the original software version running in an embedded PowerPC processor.*

## 1. Introduction

The high abstraction level present in software programming languages is a very appealing feature in the development of complex embedded processing systems for most real-life problems. The system designer is focused on a structured description of what the system is expected to perform. Then, he/she firstly relies on the performance obtained by executing the compiled binary code on the microprocessor (usually a General Purpose Processor - GPP) of the target system. However, having to be sequentially executed by a GPP, the software application usually executes in a very inefficient way, as the parallelism present in the algorithms is not conveniently explored.

By using hardware description languages such as Verilog or VHDL, it is possible to develop hardware cores that can effectively implement custom solutions with high degrees of parallelism that can execute at a fraction of the time they would require in the GPP. There are many examples of the successful acceleration of applications with custom hardware cores, such as the acceleration of a Monte Carlo application [1] where hardware was designed to process the same operations but 80 times faster than its software counterpart. Another example [2] is the case where, both an FPGA and a reconfigurable multimedia array coprocessor were used to achieve speedups ranging from 2.3 to 7.3 times.

However, hardware design is a much more complex task than software programming, requiring a different way of thinking and a solid knowledge of computer architecture. Besides describing the desired functional description with conventional hardware description languages (VHDL or Verilog), the designer needs to take into account the target platform's architecture with its specific timing constraints. As opposed to software development where inefficient coding of part of an application may impact only in the program running time, in hardware design it is imperative to meet strict timing requisites for the system components to guarantee the correct behavior of the whole systems.

High-Level Synthesis (HLS) tools generate RTL (Register-Transfer Level) designs from purely untimed algorithms, where the timing of operations is not explicitly specified. By doing this effectively and compiling standard programming languages such as C, even a software programmer may take advantage of hardware acceleration. Using a HLS tool, the developer only has to identify the critical sections (usually named as hot-spots) of the software application and provide them to the HLS tool. Then, this tool generates the description of the hardware cores with the original functionalities. These descriptions are then given to low level back-ends that perform the logic synthesis, mapping, and placement and routing stages to produce the final implementation.

The work presented in this paper is part of REFLECT [5], an FP7 European funded project whose main objective is to research and develop novel approaches to reduce the efforts for developing FPGA based embedded computing systems. In this paper we report the results of accelerating

a software application (a Stereo Navigation application to be used in autonomous vehicles), by making use of hardware cores synthesized from computing intensive C functions selected in the application, thus creating a hardware/-software system with increased performance than the original C code. The high-level synthesis was performed by Mentor Graphics's Catapult C [12], a commercially available HLS tool. Although the synthesis tool does not support floating point arithmetic, an open source floating point software library was used and synthesized with the application code, reaching to an effective speedup of the complete application.

The remainder of this paper is organized as follows. In Section 2, the software application is introduced. Section 3 describes the hardware platform used to implement the application. Section 4 presents the profiling analysis that lead to the identification of the most critical functions of the application. The generation of the hardware cores is described in Section 5. Section 6 analyses the results of the FPGA implementation and the performance of the hardware assisted solution and Section 7 concludes the paper.

## 2. Software Application

As mentioned before, the application considered in this work is a stereo navigation application for autonomous vehicles provided by Honeywell [6]. This application, intended to run on the embedded processor of a vehicle, processes images captured by a pair of cameras with a stereo vision configuration, and using the Harris corner detection technique identifies features in the images. These features are points of interest of an image that may represent obstacles in the path of the vehicle, or just reference elements in both images that will be used to derive the vehicle movement. By analyzing pairs of frames (from the two cameras) and comparing, in consecutive images, how the features in the pictures have changed through the time, the system can calculate the relative translation, rotation and speed of the vehicle as well as the position of relevant obstacles in its path. Although in theory it could be possible to infer the position of a feature in a three-dimensional space using only one picture from each camera, the probability of having a correct match in a cluttered urban environment is often around 20% [7]. The process of analyzing an obstacle's feature is represented in Figure 1. There, a lamp pole is identified as a potential obstacle and represented as a feature. This feature is compared across the two cameras and across the time (process called circular check). After eliminating the mismatched features, a 3D Reprojection [1] representing the vehicle's surroundings is created.

As a computationally demanding image processing is the basis of this system and as the data coming from the 3D reprojection of the features in the image have to be readily available and with minimal delay for an efficient vehicle navigation, this application proved to be an interesting basis for analyzing the effectiveness of hardware acceleration

[1] Process to analytically derive 3D coordinates of a point from image projections of the point

Figure 1. Feature extraction process (image from [7]).

from the original C code.

## 3. Hardware Platform

The reconfigurable characteristic of FPGAs makes this a convenient technology for the early stages of a digital design development, since various implementation alternatives can be experimented in the same physical platform without incurring in fabrication costs.

Due to the complexity of the original algorithm, replacing completely the software application with its hardware equivalent would result into a solution requiring too much hardware resources of a target FPGA. Although this is naturally possible, the acceleration degree provided by converting the less critical parts of the application, when compared with the scenario where only the most critical functions are selected to accelerate the application, would not justify the need for a large FPGA required to host the whole implementation. Also, because of the nature of some sections of the code (e.g. variable length loops) not all the functions could be replaced by efficient digital cores, or, in some cases, the replacing hardware blocks may also be less efficient than the corresponding software functions.

Because of this, the focus of the hardware acceleration was concentrated into the most time consuming functions, leaving the rest of the application to be executed by a GPP. Having the basic architecture concept well defined, we developed the hardware/software system in a Xilinx ML507 development board containing a XC5VFX70T (Virtex-5 family FPGA). This FPGA includes a 32-bit embedded RISC PowerPC440 processor, indispensable to execute the software component of the application, and the reconfigurable area that will host the custom computing cores for the most time consuming functions. This embedded processor has a maximum frequency of 400 MHz and its Processor Local Bus (PLB), which is responsible for the connection and data transaction between the processor and all the peripherals (including the designed hardware modules), is capable of running with frequencies of up to 100 MHz. Although this processor does not have native hardware support for floating point data types, Xilinx provides an optimized Floating Point Unit (FPU) for the PowerPC, connected via its FCB (Fabric Co-processor Bus). The FPU implementation supports the IEEE-754 floating-point arithmetic operations in single or double precision [8]. This

FPU however, is implemented in the FPGA reconfigurable fabric and operates at lower frequency when compared to the embedded processor (200 MHz).

## 4. Profiling the Application

To simplify porting the software application to the embedded CPU, a Linux kernel specific for the processor [9] was used as a base operating system. This provides a series of convenient services such as 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. Although the underlying operating system always consumes some of the CPU processing power, it eases the software development task and allows the utilization of the standard Linux libraries, thus making the code portable for compilation in any other Linux computer. Another alternative would be to adapt the original code to run as a standalone application, but where specific parameters of the development board and target FPGA would have to be taken in account.

Table 1. Summary of profiling results for the stereo navigation application, compiled with -O3 optimization and executed by the PowerPC at 400 MHz.

| function name | times called | Execution time | |
|---|---|---|---|
| | | % | seconds |
| convRepl2 | 144 | 24.5 | 3.30 |
| convRepl1 | 144 | 24.0 | 3.22 |
| convConst | 96 | 13.9 | 1.86 |

Having the software application being executed by the FPGA embedded processor, the critical functions were identified using `powerpc-linux-gprof`, a variant of the well known `gprof` profiling tool for the embedded Linux kernel. From the results present in Table 1, we can see that the application contains three functions that consume most of its execution time: `convConst`, `convRepl1` and `convRepl2`. These three functions have high similarities between them. All of them execute a series of convolutions over a matrix of $96 \times 96$ elements, with single precision floating point data types. With the aid of smaller matrices and arrays of constant values (with nine elements for `convConst` and eleven elements for `convRepl`), also in single precision floating point, each function outputs a $96 \times 96$ matrix. This convolution process is highly demanding, involving a series of floating-point multiplications and additions, as shown in equation 1, where $U[u]$ represents the element $u$ of the $U$ matrix (larger matrix) and $H[h]$ represents the $h$ element of the $H$ matrix (smaller, constant matrix).

$$acc_n = U[u] \times H[h] + acc_{n-1} \quad (1)$$

Due to the nature of the floating-point data types used in these functions, and because the PowerPC processor does not have native support for floating point, a floating-point auxiliary processor was attached to the main CPU.

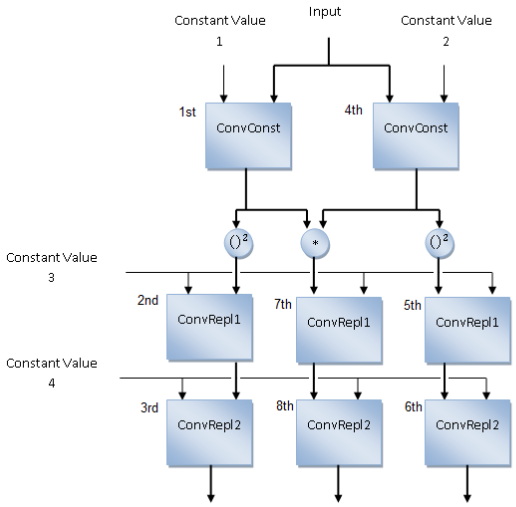


Figure 2. Harris data flow.

As the three most time consuming functions always analyze the same sized input matrix with the same sized constant array/matrix and include a small computing core based on multiply-accumulate operations, they were selected as good candidates to be implemented as specialized hardware cores.

These three functions are part of the Harris process and the sequence of their invocation is presented in Figure 2 as a simplified dataflow diagram. The input for the whole process is a tile of the image represented by a matrix in 8 bit gray-scale. Making use of a fixed value vector, the `convConst` function analyses that tile and, after convolving the $96 \times 96$ matrix, stores the values in another $96 \times 96$ matrix that will itself be convoluted by the `convRepl1` function. The process repeats then for the `convRepl2` function. When `convRepl2` finishes the `convConst` function is called again to process the input matrix with a different constant array, following the order shown in the Figure 2. The process is repeated for every tile analyzed by the application.

## 5. Generating the Hardware Cores

Catapult C is a mature HLS tool used to compile algorithms described in C into RTL implementations (see, e.g., the examples presented in [3] and [4]). Although Catapult C can generate RTL from entirely untimed C algorithms, some minor changes to the original application had to be made such as defining the size of the vectors that are passed as arguments to the functions. Also, dynamic memory allocation is not supported.

As implementing floating-point arithmetic in custom hardware leads to complex logic circuits that, in general, require more logic resources than fixed point arithmetic operators and exhibit longer propagation delays or clock latency. Because of this, the use of fixed point arithmetic is preferable whenever it is enough to accommodate the pre-

cision and range requirements of the application. Catapult C does not synthesize floating-point operations and instead automatically converts floating-point data types present in the algorithm to fixed-point representations. It is possible for the designer to specify the fixed-point data type representation to be used by redefining a pragma specifying the total number of bits of the data type, the number of bits that will be used for the fractional part, the rounding method, and whether saturated arithmetic is used. With a careful analysis of the `convConst` function, we have found that, although the data types present in this function were declared as floats, the actual values used were only 8-bit unsigned integers and the fixed value array contains only elements equal to 1, 0 and $-1$. This change alone provided some acceleration to this function, as referred in the next section.

However, for the other two functions, we have found that fixed-point arithmetic would not give the necessary precision and range, even for data formats with 64 bits, with 32 fractional bits. Thus the use of floating-point was mandatory to build hardware cores capable of executing these two functions. As the high-level synthesis tool used does not support this data type, the solution was to perform the floating point arithmetic in software as fixed point operations and let the CatapultC compile that code along with the application.

### 5.1. Soft Floating-Point Library

To turnaround the unsupported floating-point data types by Catapult C, we opted to adopt the use of a software library with implementations of the necessary floating-point operations. Simplified versions of the add and multiply elementary operations were adapted from the softFloat library (obtained at [10]), originally designed to emulate the floating-point operations in processors that do not contain hardware support for floating-point arithmetic. As only integer data types are used to store the same binary data that represent floats, Catapult C is thus able to generate hardware implementations for the soft float operations. To simplify the original software implementation we removed the treatment of special IEEE-754 cases such as NaNs (Not-a-Number) and consider only the rounding to the nearest even.

By using this software emulation of the floating point operations the performance of the application was reduced, compared to the initial profiling done the original code and the FPU attached to the processor, almost doubling the execution time for the two most demanding functions. On contrary, the execution time of the integer version of function `convConst` was reduced to only 7% of the original code (Table 2).

### 5.2. Generating the Hardware Cores using the SoftFloat Library

With the solution for the no floating-point problem found, tested and validated, the three functions could be analyzed and compiled by Catapult C to RTL logic. Catapult C interprets the header of the function and, in the case of non pointer arguments, they are automatically inferred as inputs; in the case of a pointer argument or arguments such as arrays, Catapult C infers input, output or input/output ports, depending if they are read, written, or read and written within the function [11].

Table 2. Execution times of the selected functions, with FPU and using the emulation floating point library and integer arithmetic for the function `convConst`.

| function name | FPU | | SoftFloat & int | |
|---|---|---|---|---|
| | % time | seconds | % time | seconds |
| `convConst` | 13.9 | 1.86 | 0.7 | 0.13 |
| `convRepl1` | 24.0 | 3.22 | 27.2 | 5.96 |
| `convRepl2` | 24.5 | 3.30 | 27.2 | 5.96 |

Thus, in the case of the function `convRepl1`, after analyzing its header (listing 1) as all the variables are represented by vectors and as `U` and `H` are defined as constants, these two values are automatically determined as the module's inputs. By analyzing the body of the function it is determined that `Y` is only used to store data in the function and, therefore, is identified as an output.

Listing 1. convRepl1 function header

```
void convRepl1(const U[u],const H[h],Y[y])
```

At this point, the interfacing ports can be designed to obtain and store their values from RAMs or registers. Designing the modules to use data from registers is advisable when accessing scalar variables or small arrays, since only a bus is built to connect the module to the register. However, for functions using as arguments large arrays, the access as registers would translate into wide buses, providing in parallel all the input elements. Therefore, in this case it is advisable to use RAMs for these cases as only the buses for address, data and read/write control are synthesized. Because of the size of the matrices involved in the computing process, the interface between the core and the PowerPC was implemented with RAMs.

This way, each hardware module will be connected to two DPRAMs (Dual-Port RAM), one for receiving the input values that would originally be passed to the software function and a different DPRAM for storing the results of the computation. One of the memory ports is attached to the PowerPC peripheral bus (PLB) and the other port is connected to the memory interface of the hardware module. The reason why DPRAMs were used was because the PLB bus could drive one set of the ports and the hardware module could drive the other, thus each one could operate over the data independently. Each module also has a "start" input and a "done" output which serve as handshake signals to be controlled by the code running in the PowerPC, through a small set of memory mapped ports (refer to Figure 3).

To operate the hardware module the application executing in the PowerPC has to fill the input memory with the data to be processed by the core, issue a start signal and wait for the completion of the execution. The PowerPC

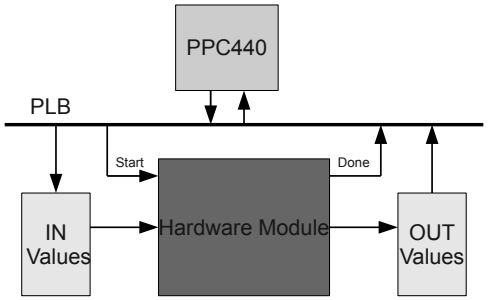


Figure 3. Organization of the PowerPC peripheral interface to the hardware modules.

will be idle until the module issues a done signal, when the processor can start reading all the result values from the module's output DPRAM.

After the creation of each function's hardware core, they can be validated via ModelSim simulation. Again, Catapult C can help to reduce the test, debug and verification time, since, after analyzing all the rest of the application, a testbench is automatically created where all the values that would be passed to the function are used as input signals of the module, and the output signals of the module (including the results) are retrieved and analyzed. In the end, the obtained values are automatically compared with the original software based results.

## 6. Experimental Results

For each of the three hardware modules, we have implemented four solutions with increasing target operating frequencies, as shown in Table 3. For frequencies greater or equal than 450 MHz, Catapult C was not able to generate solutions and after logic synthesis with XST the designs created for 200 MHz and 400 MHz fail to reach by far those target frequencies. As the target frequency increases, the number of clock cycles also increases, due to the way the algorithms are scheduled by Catapult C in order to try to reach the specified clock period. For the function `convConst`, the synthesis results indicate that the higher the clock frequency, the better the performance. However, for the other two modules (`convRepl1` and `convRepl2`), higher clock frequencies do not necessarily lead to smaller execution times. According to these results, and assuming that the specified target frequencies are satisfied, the clock frequency among the 4 cases that minimizes the overall execution time of these functions is 100 MHz.

However, the target clock frequency referred in Table 3 is only specified as a target for the high-level synthesis process and we have verified that the actual maximum clock frequency allowed for each implementation is very different from the specified values. To verify whether the synthesized cores would actually reach the desired target clock frequencies, we have synthesized the four versions of each module with XST in order to have a first estimate of the maximum clock frequency actually allowed for each design. As presented in Table 4, the designs created by CatapultC for 200 MHz and 400 MHz will not reach these frequencies when implemented in the target FPGA. If we consider that each module runs with the maximum clock frequency reported by XST, the execution times show that, besides the simplest module `convConst`, the best solutions are those created for the lower target frequency (50 MHz). However, after Place&Route the maximum clock frequencies reached for the 100 MHz and 50 MHz versions show that the solution that minimizes the overall execution time, using a common clock for all modules, are the circuits synthesized for 100 MHz.

Table 3. Timing results of the hardware modules synthesized by Catapult C and estimated execution times based on the target clock frequency.

| module | Target Frequency (MHz) | Latency (clock cycles) | Execution time (ms) |
|---|---|---|---|
| convConst | 50 | 97,937 | 1.95 |
| | 100 | 106,845 | 1.07 |
| | 200 | 142,485 | 0.71 |
| | 400 | 196,569 | 0.49 |
| convRepl1 | 50 | 839,233 | 16.78 |
| | 100 | 1,447,489 | 14.47 |
| | 200 | 3,080,065 | 15.40 |
| | 400 | 6,159,745 | 15.40 |
| convRepl2 | 50 | 837,889 | 16.76 |
| | 100 | 1,446,145 | 14.46 |
| | 200 | 3,078,721 | 15.39 |
| | 400 | 6,150,509 | 15.38 |

Table 4. Maximum clock frequencies estimated by XST and execution times for the three modules.

| module | Required frequency (MHz) | Clock Frequency (MHz) | Execution time(ms) |
|---|---|---|---|
| convConst | 50 | 120.5 | 0.82 |
| | 100 | 125.2 | 0.85 |
| | 200 | **177.0** | **0.81** |
| | 400 | 179.3 | 1.10 |
| convRepl1 | 50 | **66.0** | **13.32** |
| | 100 | 103.8 | 14.05 |
| | 200 | 162.6 | 19.01 |
| | 400 | 163.7 | 37.79 |
| convRepl2 | 50 | **63.3** | **13.30** |
| | 100 | 103.8 | 14.04 |
| | 200 | 159.3 | 19.36 |
| | 400 | 168.7 | 36.61 |

As the arrays used for the convolution process of each function were composed by constant values, there was no need for the modules to obtain them from the main processor, through the dual-port memories. Instead, these constant values could be embedded into the module's logic circuit, thus potentially reducing the logic complexity and increasing the maximum clock frequency. With this additional optimization, the best implementations among the four solutions generated by Catapult C were still the modules created for a target clock frequency of 100 MHz. A slight increase in performance was observed for functions `convConst` and `convRepl1`, as shown in Table 5.

To evaluate the real gain in performance with the hardware modules integrated in the application, including the

Table 5. Comparison of the number of clock cycles for the original and optimized versions (100 MHz target frequency).

| module | Clock cycles | | |
|---|---|---|---|
| | original | optimized | optimized/original |
| convConst | 106,845 | 80,265 | 75% |
| convRepl1 | 1,447,489 | 1,346,113 | 93% |
| convRepl2 | 1,446,145 | 1,446,145 | 100% |

time used for data transfer, the original code was modified in order to interface them with the PLB through the dual port RAM blocks. The real execution time of the target functions was measured, including also the time taken to transfer data between the main memory of the PowerPC and the interface memories. The speedups obtained for each function, both for the modules compiled from the original code and for the versions with the embedded constants, is presented in Table 6. In this table we present the speedups obtained by equation 2, where $t_{SW}$ is the execution time of the software version and $t_{HW} + t_{data}$ represents the sum of the execution time of the hardware module and the total time used by the data transfer between the PowerPC and the module. To measure the impact in performance due to the utilization of the software emulation of the floating point arithmetic operations, Table 6 includes the values of speedup obtained by comparing to the software execution time taken by the original C code (column "original") and the execution time of the software versions using the softFloat library (for functions `convRepl1` and `convRepl2`) and integer arithmetic (for the function `convConst`).

$$speedup = t_{SW} / (t_{HW} + t_{data}) \quad (2)$$

Table 6. Execution time including data transfer and speedups obtained.

| solution | Name | Execution time (s) | Speedup: HW core vs. | |
|---|---|---|---|---|
| | | | SoftFloat+int | original |
| Original | convConst | 0.33 | 0.40 | 5.72 |
| | convRepl1 | 2.35 | 2.54 | 1.37 |
| | convRepl2 | 2.34 | 2.55 | 1.41 |
| Constant | convConst | 0.27 | 0.48 | 6.89 |
| | convRepl1 | 2.20 | 2.71 | 1.46 |
| | convRepl2 | 2.34 | 2.55 | 1.41 |

The results in Table 6 show that the `convRepl` hardware modules are approximately 2.5 times faster when compared with the software function that was used by Catapult to create them, using the software library for the floating point operations. Regarding the original software application, the speedup values are more modest. Nevertheless, these results still demonstrate the effectiveness of using a high-level synthesis tool to improve the performance of a C application with floating point arithmetic.

Concerning the `convConst` module, although the hardware version presents a relevant speedup when compared to the original software version, there is an effective reduction of the processing speed when compared with the optimized `convConst` function (in software) that uses only integer arithmetic. This is mainly due to the overhead caused by the data transfer between the software and the hardware and suggests that in this case it would be better to keep this function implemented in software rather then in hardware.

### 6.1. FPGA Implementation

Table 7 presents the summary of the Virtex 5 FPGA resource utilization. The values refer only to the optimized hardware modules with embedded constants and do not comprise the block RAMs used to implement the DPRAM.

Table 7. Resource utilization for the optimized hardware modules (FPGA XC5VFX70T).

| Resources used | convConst | convRepl1 | convRepl2 |
|---|---|---|---|
| Slice Registers used as Flip Flops | 1,032 | 5,987 | 5,899 |
| Slice LUTs | 1,180 | 11,676 | 11,709 |
| Occupied Slices | 726 | 4,492 | 4,304 |
| DSP48Es | 3 | 7 | 7 |

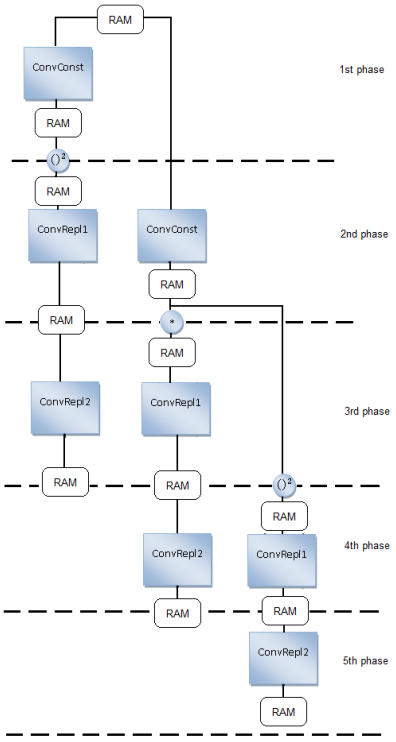


Figure 4. Parallel flow.

These hardware cores were used in a complete hardware/software system of the Stereo Navigation application. This implementation allowed a global speedup of 1.36 times when executing them sequentially, i.e., following the same execution order as in the original software version, according to the data flow illustrated in Figure 2. Further exploiting the possibility of allowing the hardware modules to operate in parallel, an improved version was built

that schedules the execution of each function as depicted in Figure 4. This implementation obtained a speedup of 1.5 times over the original Stereo Navigation application.

## 7. Conclusion

This paper studied a design flow to build hardware blocks to improve the processing speed of an application making use of a commercial C-RTL high-level synthesis tool. The steps that have to be followed to create a fully functional hardware block for a specific operation can help reducing the development time and time-to-market of hardware design.

Using simple source-to-source optimizations and Catapult C, the hardware implementations of the selected functions were able to achieve speedups of almost 1.46 times, operating at a quarter of the processor clock frequency. The whole hardware/software implementation achieved a global speedup of 1.5 times over the pure software implementation.

Ongoing work is considering further improvements in order to achieve hardware cores with higher performance and on other hot-spots that can be migrated to hardware cores to produce higher global speedups. A special effort is being made to reduce the overhead impact caused by the soft float implementation.

## 8. Acknowledgments

This work was partially supported by the European Com-munity's Framework Programme 7 (FP7) under contract No. 248976. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the European Community. The authors are grateful to Honeywell for making the Stereo Navigation application available.

## References

[1] 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.

[2] 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.

[3] 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.

[4] 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.

[5] REFLECT project. Available at http://www.reflect-project.eu/, last time accessed in November the 12th, 2011.

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

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

[8] XILINX. LogiCORE IP Virtex-5 APU Floating-Point Unit (v1.01a), March 2011.

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

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

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

[12] © Mentor Graphics, Catapult C Synthesis, Available at http://www.mentor.com/esl/catapult, last time accessed in 6 of January, 2012.