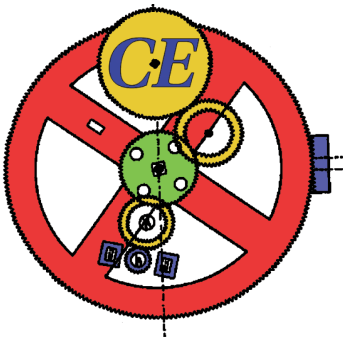

Computer Engineering
Mekelweg 4,
2628 CD Delft
The Netherlands
http://ce.et.tudelft.nl/

2016

MSc THESIS

Software To Hardware: Alternatives For Reducing Design Time Of Optimized FPGA Implementations In Medical Devices [CP]

S.P. Metman

CE-MS-2016-08

Abstract

Current X-ray machines often rely on off-the-shelf PCs to implement image processing chains. These PCs have a limited lifetime and need to be replaced every few years. If such a PC is not commercially available anymore, the image processing chain has to be redeveloped for a new target platform to ensure the exact same performance and functional correctness and as is required in medical devices and embedded solutions. A solution to this is to develop a Register-Transfer Level (RTL) design from the image processing algorithm, and load this on a Field-Programmable Gate Array (FPGA). Certain types of FPGAs have increased lifetimes compared to PCs, and will decrease total maintenance cost. Also, this design will have the same performance and correctness when transferred between compatible boards. An added advantage is a performance increase in throughput.
However, an RTL design takes a long time to develop and is thus very costly. To accelerate the design process, Computer-Aided Design (CAD) tools can be used. Yet the performance of such a generated RTL design may be low, compared to a manually crafted RTL design. A functionally correct design is required in medical devices as well. This thesis investigates if it is possible to reduce the design time of hardware-software co-design while maintaining a level of optimized performance, and ensuring the same functional correctness of the original algorithm.
To this end, a workflow is produced, implementing a proof-of-concept system using Vivado HLS. This CAD tool generates an RTL design automatically which can be optimized using user-input. This design is implemented on Zync 7z030 System-on-Module (SoM) and is connected using the Dyplo framework. A detailed optimization guide is given to explore the trade-offs between throughput and area usage.
A rough estimate of the development time of the final produced workflow indicates that the development time of an RTL design can be decreased up to ten times compared to manual creation, while ensuring portability, easy modification and functional correctness. Using optimizations and restructuring a high performance close to a manual design could be reached for individual filters, as well as an estimated performance for the entire hardware implementation.
This thesis represents a public summary of the original thesis with summarized background, implementation and results to protect the confidentiality of its information.

TU Delft
Delft University of Technology

Faculty of Electrical Engineering, Mathematics and Computer Science

Software To Hardware: Alternatives For Reducing Design Time Of Optimized FPGA Implementations In Medical Devices [CP]

THESIS

submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE

in

COMPUTER ENGINEERING

by

S.P. Metman
born in Amsterdam, the Netherlands

Computer Engineering
Department of Electrical Engineering
Faculty of Electrical Engineering, Mathematics and Computer Science
Delft University of Technology

Software To Hardware: Alternatives For Reducing Design Time Of Optimized FPGA Implementations In Medical Devices [CP]

by S.P. Metman

Abstract

Current X-ray machines often rely on off-the-shelf PCs to implement image processing chains. These PCs have a limited lifetime and need to be replaced every few years. If such a PC is not commercially available anymore, the image processing chain has to be redeveloped for a new target platform to ensure the exact same performance and functional correctness and as is required in medical devices and embedded solutions. A solution to this is to develop a Register-Transfer Level (RTL) design from the image processing algorithm, and load this on a Field-Programmable Gate Array (FPGA). Certain types of FPGAs have increased lifetimes compared to PCs, and will decrease total maintenance cost. Also, this design will have the same performance and correctness when transferred between compatible boards. An added advantage is a performance increase in throughput.

However, an RTL design takes a long time to develop and is thus very costly. To accelerate the design process, Computer-Aided Design (CAD) tools can be used. Yet the performance of such a generated RTL design may be low, compared to a manually crafted RTL design. A functionally correct design is required in medical devices as well. This thesis investigates if it is possible to reduce the design time of hardware-software co-design while maintaining a level of optimized performance, and ensuring the same functional correctness of the original algorithm.

To this end, a workflow is produced, implementing a proof-of-concept system using Vivado HLS. This CAD tool generates an RTL design automatically which can be optimized using user-input. This design is implemented on Zync 7z030 System-on-Module (SoM) and is connected using the Dyplo framework. A detailed optimization guide is given to explore the trade-offs between throughput and area usage.

A rough estimate of the development time of the final produced workflow indicates that the development time of an RTL design can be decreased up to ten times compared to manual creation, while ensuring portability, easy modification and functional correctness. Using optimizations and restructuring a high performance close to a manual design could be reached for individual filters, as well as an estimated performance for the entire hardware implementation.

This thesis represents a public summary of the original thesis with summarized background, implementation and results to protect the confidentiality of its information.

Laboratory : Computer Engineering
Codenumber : CE-MS-2016-08

Committee Members :

Advisor: Dr. ir. Z. Al-Ars, CE, TU Delft

Chairperson:	Prof. dr. ir. K. Bertels, CE, TU Delft
Member:	Ir. S.C. van der Vlugt, Philips Healthcare
Member:	Dr. A.E. Zaidman, ST, TU Delft

Dedicated to:
My amazing Lara for her amazing, loving support;
My parents, whose help and understanding is beyond this world;
My brother for our kinship and long-lasting friendship;
My grandfather for his philosophical and helpful advice;
My university buddies for their help and putting up with my rants;
And all my family and friends,
for whom I will always carry a special place in my heart.

Contents

List of Figures

List of Tables

List of Acronyms

ALMARVI Algorithms, Design Methods, and Many-core Execution Platform for Low-Power Massive Data-Rate Video and Image Processing

API Application Program Interface

ASCII American Standard Code for Information Interchange

CAD Computer-Aided Design

CSAS Carry-Save Add-Shift

CPU Central Processing Unit

DMA Direct Memory Access

DSP Digital Signal Processing

DYPLO DYnamic Process LOader

ECG ElectroCardioGram

FAST Fixed-point Analysis & Scaler Tool

DC Detector Controller

FIFO First-In First-Out

FPGA Field-Programmable Gate Array

FPS Frames Per Second

FSM Finite State Machine

FSP Fast Serial-Parallel

FXD Flat X-ray Detector

GPU Graphics Processing Unit

GUI Graphical User Interface

HIL Hardware-In-the-Loop

HLS High-Level Synthesis

IGT Image Guided Therapy

II Initiation Interval

IP Intellectual Property

iXR interventional X-Ray

LP LaPlacian

LUT Look-Up Table

OpenCL Open Computing Language

OS Operating System

PC Personal Computer

PHC Philips HealthCare

PNG Portable Network Graphics

BRAM Block Random Access Memory

RTL Register-Transfer Level

SD Secure Digital

RTOS Real-Time Operating System

SOC System-On-Chip

SOM System-On-Module

TDD Test-Driven Development

VHDL VHSIC Hardware Description Language

VHSIC Very High Speed Integrated Circuit

Acknowledgements

Zaid Al-Ars for his enthusiasm and insight
Panagiotis Mitsis for his help and companionship
Steven van der Vlugt for his guidance and making me feel welcome
Rob de Jong for his wisdom and amazing bug-fixes
Bart van Rijnsoever for his support and supervision
Rachana Arunkumar for her assistance and helpfulness

S.P. Metman
Delft, The Netherlands
September 7, 2016

1 Introduction

Translating software to hardware, or in a general sense co-design, is a difficult and time-consuming task. [5] A simple task in software engineering, for example writing and reading a file, becomes a monumental one in a hardware design; a separate memory has to be designed to store the file, a bus needs to be connected from the processing unit to the memory, communication needs to be handled using a certain protocol, and so on. Even then, this hardware design would be suited for only one specific task. So why even bother when in our software design the same exact results can be achieved?

There are multiple answers to this question. One is performance. A custom-built hardware design – for example defined at Register-Transfer Level (RTL) – focused on executing only one task, will often have higher throughput than a software design. [6] [7] Especially in real-time systems, this performance may be necessary to fulfill requirements. A second reason why hardware design may be preferred is portability without performance differences. In general, a software design is often portable to many different platforms. However, this design may vary in performance depending on the target platform. To ensure the exact same performance and correctness when porting to a new platform, which may be necessary in an embedded solution, is a time-consuming option. A Register-Transfer Level (RTL) hardware design will have neither of these problems; if it can be implemented on a board or chip, the performance is fixed.

Yet constructing an RTL design manually takes much development time, especially compared to a software design. To reduce the development costs associated with this relatively long development time, Computer-Aided Design (CAD) tools can be used to automatically convert higher-level software designs to an RTL design. As described earlier, such a generated RTL design can be portable to different technologies, while still offering a better performance in terms of throughput compared to a software design. [8] [9]

The focus of this thesis is to investigate to what extent current CAD tools are useful to convert a software solution to an embedded platform. This is done by investigating an alternative workflow incorporating software to hardware translation, or as it is also known, a software/hardware co-design. Most workflows found in common literature about co-design focus mostly on performance in throughput or latency. The goal of this thesis rather is to see if the design effort, the development time, of the final hardware design can be lowered, while still meeting a certain performance requirement.

The rest of this section will: introduce the context in which this investigative research will be done, provide background on the subject and environment, detail a specific problem statement and research question, and finally outline the rest of this document.

1.1 Context

1.1.1 Philips Healthcare

Royal Philips is a Dutch company focused on making technology in different areas, mainly centered around electronics, lighting and healthcare. A sub-divison of this company, Philips Healthcare, has activities in many different fields, like:

- Diagnostic Electrocardiogram (ECG)
- Fluoroscopy
- Ultrasound
- Magnetic Resonance
- Computed Tomography
- Mammography
- Radation Oncology
- Diagnostic and Interventional X-Ray

Their mission is to provide everyone with quality healthcare everwhere, investing in health improvement at every step: prevention, treatment and follow-up. This thesis was written with help of the Image Guided Therapy (IGT) department of Philips Healthcare, tackling a real-life problem in the firmware team of the interventional X-Ray (iXR) department located in their Research & Development site in Best, the Netherlands.

1.1.2 Firmware for interventional X-Ray (iXR)

The interventional X-ray department at Philips Healthcare aims to improve patient outcomes and save lives. The tools used by this department assist patient treatment using minimally invasive surgical procedures. The systems researched and developed by this department include X-ray systems to support interventional cardiac, vascular, neuro, oncology and electrophysiology.

Firmware is the soft line between software and hardware. The X-ray systems designed in this department uses firmware to perform image processing on raw X-ray input images. This firmware is the link between the algorithmic software implementation and the final hardware implementation. For this department then a research for an optimized, cost-reduced software-to-hardware translation is very valuable.

1.1.3 Almarvi

To reduce overall system design cost and time-to-market, and to enable low-cost solutions for high volume markets, the Almarvi project was conceived. [10] Almarvi stands for: Algorithms, Design Methods, and Many-core Execution Platforms for Low-Power Massive Data-Rate Video and Image Processing. Specifically, Almarvi wants to: enable cross-domain re-use and interoperability for different product categories and application domains; facilitate predictable system and product properties; and develop joint hardware-software techniques for resource and power-management. Essentially, the project aims to ease the programming of new technologies and cut production costs.

Almarvi is a European project with many different parties contributing deliverables worldwide. Philips Healthcare is also working on the Almarvi project, together with

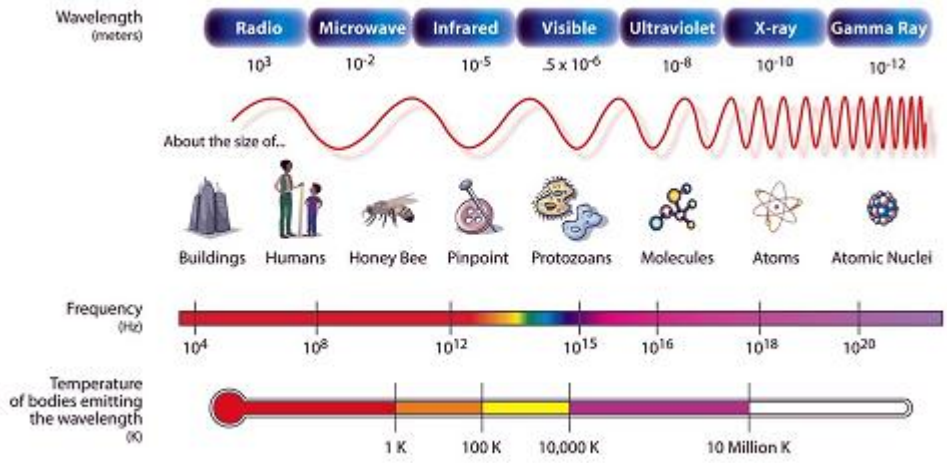

Figure 1.1: The electromagnetic spectrum.
Source: [1]

TUDelft. Many of the knowledge gathered by working on this project will also be shared with the Almarvi project. Providing a fast hardware-to-software translation combines perfectly with the Almarvi mission statement: to ease the programming of new technologies and cut production costs by reducing design time.

1.2 X-ray Image Processing

X-radiation, which is composed of X-rays, is a form of electromagnetic radiation. The electromagnetic spectrum chain can be seen in Figure 1.1. X-rays below 0.2 - 0.1 nanometers wavelengths are called *hard* X-rays. These hard X-rays can penetrate skin and are most often used in medical radiography, like in the iXR medical X-ray systems, but also for other applications, for example airport security. The X-rays can be adjusted for a specific application; this means that by tweaking this energy the X-ray will change penetration depth. This allows X-rays to be absorbed or scattered by bones, veins, or other tissues, providing a means to 'see' different layers in the same object.

The patient to be treated will be placed underneath the X-ray tube to absorb or pass the produced X-rays. Under the patient, an X-ray detector will be placed. An X-ray detector 'catches' the X-ray which have passed through the subject, and will make up the basic image received by the system. The detector of the X-ray machine is relatively large, 20 centimeters by 30 centimeters. A diagram of it can be seen in Figure 1.2. It can be seen that the detector is divided up in squares; these squares are the X-ray sensors which will pick up the X-rays passing through the subject. These will, after refinement, later represent pixels in the final image.

The final setup in a general X-ray system can be seen in Figure 1.3. This setup will be detailed further in this section.

First, the detector takes the so-called 'shadow image' containing the X-ray image created by the X-ray machine. Some imperfections may exist during the capture of a

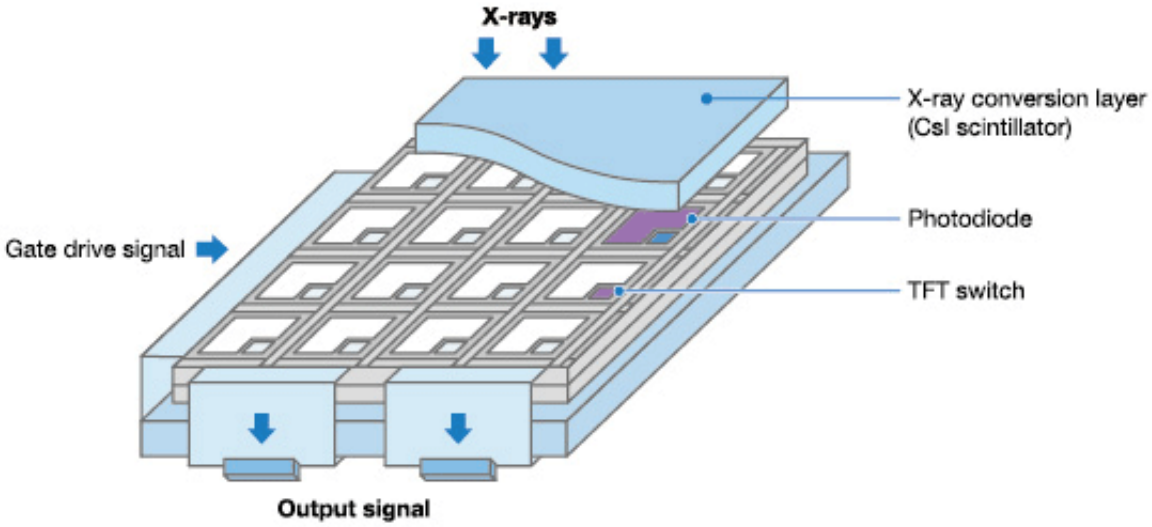

Figure 1.2: An X-ray detector.
Source: [2]

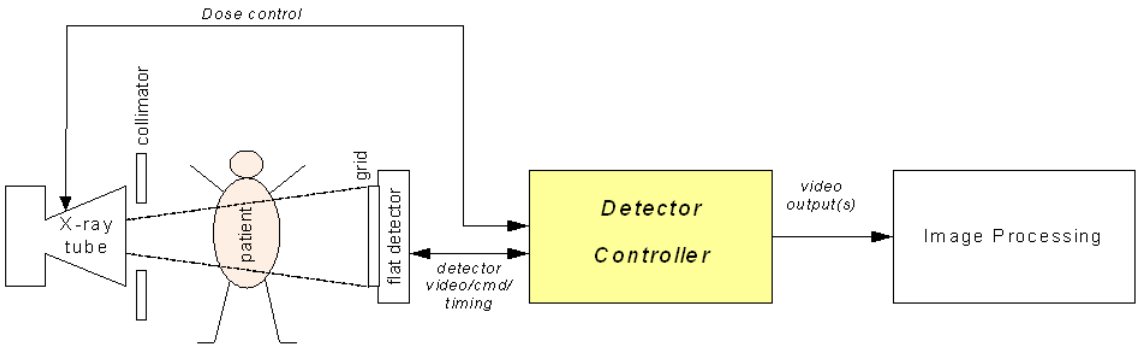

Figure 1.3: The general X-ray system setup.
Source: [3]

raw X-ray image. These imperfections can be hidden by imaging algorithms, effectively cleaning the image. Most of this is done by the Detector Controller (DC). By adjusting the dose produced by the X-ray tube, based on the input by the Detector Controller (DC), the raw images of the patient can be tweaked. The Detector Controller also provides minimal clean-up of the raw images.

After the detector controller has captured the raw X-ray image and has cleaned it, the image is far from clear. To get an acceptable image quality for the physician, this image needs to be processed in several different stages. This is done in an image processing chain. Every image must be as clear as possible. This is a fuzzy result; there

are no exact guidelines what constitutes a clear image. The image processing is done with more advanced image algorithms implementing image correction, noise reduction and measuring specific attributes of each image.

After the image has been processed in the image processing chain, it is streamed to the output to be viewed directly by the physician treating the patient. However, the output of an X-ray system is often not a single image: it is a continuous stream of images constituting a video stream. This video stream is shown either directly to an external monitor or to another PC which distributes this signal to different monitors. This distribution PC also takes video input from other systems which can be used in the final output to other monitors.

One can imagine that not only functional correctness, in the sense of a clear picture, but also latency can be an important requirement of such an X-ray system. A physician who is taking images of a patient in real-time, needs to have timely feedback to allow synchronous hand-eye coordination. This means that a certain maximum latency has to be defined. With a video stream at 15 Frames-Per-Second (FPS), the operating physician would only notice a very slight, non-obstructing delay.

1.3 Problem Definition

Currently, the important noise reduction algorithm as described in Section 1.2 is implemented on a general-purpose computer – also called a Personal Computer (PC). This PC runs the image processing chain inside an X-ray machine; more complex X-ray systems may even use more than 20 PCs. A PC generally breaks down after around three to five years, yet the X-ray machine has to be maintained for at least 25 years. This means that these PCs have to be replaced many times during the maintenance period of the X-ray system, which yields high maintenance costs. Also, PCs used to implement this processing chain algorithm are generally not commercially available for this time. This necessitates porting the software that is run on the PC to a new target platform, while ensuring the exact same performance and functional correctness of the algorithm, which are essential to a medical system and required in an embedded solution. This results in high development costs.

To combat this problem, the image processing techniques these conventional computers employ can instead be implemented on reconfigurable computing technologies, like Field-Programmable Gate Arrays (FPGAs) devices. FPGAs are integrated circuits that can be programmed by engineers after manufacturing, by creating a Register-Transfer Level (RTL) design written in for example in VHDL or Verilog, and loading this onto the FPGA board. An FPGA therefore fills an important niche between ASIC design and Software Engineering: being able to design application-specific hardware without the steep design time and cost of ASIC designs, and having a relatively high performance compared to standard hardware.[11] Because of their reconfigurability, FPGA technology has been used in a great amount of co-design implementations, especially in image processing with speeds comparable to GPU implementations. [12] [13] [14]

It is important to note that certain types of FPGA boards, like those intended for the automotive, defense, and aerospace industries, have a lifetime of fifteen years or more, greatly exceeding the lifetime of an average PC.[15] [16] Used in an X-ray system,

this means that these FPGA boards need not be replaced as often, which decreases the total maintenance costs. An added functionality of reconfigurable technology is that once an RTL design has been made, this Intellectual Property (IP) can be synthesized on a great many different FPGA devices if its compatible. This functionality allows designs to be synthesized on newer FPGA devices with relative ease, with the same original optimizations and functional correctness. In case of a medical device, the same hardware can be replicated with minimal added development time.

Thus there exists an added benefit of an FPGA hardware design over a software design executed on a PC. Yet to produce a hardware design, usable in an FPGA, the traditional way would be to create a fine-tuned, manually created RTL design. Although the same general design methodology would be applied, like algorithmic analysis, profiling and parameterizing, such a hand-made design development takes much development time and expertise. [17] Also, the validation of such a design is not trivial either, as custom testbenches have to be built as well.

In conclusion, the problem is that the original software implementation of the noise reduction algorithm is costly to maintain because of the maintenance and development costs that arise from it. To decrease costs, a custom hardware design for FPGA has to be created, as an FPGA board generally has a longer lifetime and its RTL design can be more easily ported to newer board. Yet the development costs for such an RTL design are relatively high because of its complexity, leading up to long development times. This leads to the following research question:

Is it possible to reduce the development time of software-to-hardware conversion of image processing algorithms for medical imaging while maintaining optimized performance and functional correctness?

This thesis aims to answer this question in form of a reproducible workflow for the creation of the noise reduction algorithm implemented in the interventional X-ray system used by Philips Healthcare, backed up by the reasoning of choices made during this process and eventual results.

In conclusion, several requirements for this thesis can be derived.

- A new workflow, defined in tools and strategies, must be produced. This workflow must be able to produce an embedded solution based on a software design. The focus of this workflow is reduced development time. This requirement ensures that a working embedded solution can be produced.

- The hardware implementation produced by this workflow must have output as close to to the results of the original software implementation. When approximating, a certain fuzzy error should be allowed; if the error is not noticeable by the human eye of the operating physician, the error is not significant. This requirement ensures functional correctness.

- The hardware implementation produced by this workflow must have a throughput which will not disturb the work of the operating physician. This corresponds with a throughput of at least 15 Frames-Per-Pixel (FPS) for 1K squared images,

and at least 7,5 FPS for 2K squared images. This requirement ensures optimized performance.

- The design effort required to execute this workflow, measured in time, has to be lower than manually designing the entire RTL design. This requirement provides proof of the possibility of reduced development costs.

In the end, the final implementation of the RTL design produced by this workflow is a proof of concept. It merely serves to answer the research question, in the sense that is should provide proof of a reduced development time, and the advantages and disadvantages of this workflow. The final conclusion of this thesis then must relate primarily to the final workflow produced.

1.4 Thesis Outline

The rest of this thesis is outlined as follows. In Chapter 2, the background of the thesis is investigated more thorougly, though this chapter has been removed for sake of confidentiality. In Chapter 3, the analysis of the workflow and the hardware implementation will be detailed. In Chapter 4, the results of this thesis will be published. In Chapter 5, these results will be discussed. In Chapter 6, conclusions regarding this thesis will be drawn and future work will be reflected upon.

Background (Removed for Confidentiality) 2

Analysis & Implementation (Summary) 3

3.1 Toolflow

For the workflow to reduce the development time, Computer-Aided Design (CAD) tools can be used which automatically generate an RTL design out of programming languages of higher abstraction. Prior research has selected the Vivado High-Level Synthesis (HLS) tool from Xilinx over the HDL Coder from Mathworks because of its relative ease of use and its performance results. Vivado HLS is CAD tool which can take a C, C++ and System C specification, and target this specification onto a Xilinx programmable device (like an FPGA) without the need to manually create a RTL design. This reduces the development time greatly as opposed to developing an entire RTL design from scratch and can be very useful for fast design space exploration, while maintaining some level of performance. [8] [9]

The original algorithm was defined in MATLAB. This implementation needed to be translated to a lower abstraction using either C, C++ or SystemC, because Vivado HLS can only accept these as an input to generate an RTL design. In the end, C++ was chosen, as many libraries used by Vivado HLS are also written in C++. Not all programming constructs allowed in C++ are allowed in Vivado HLS: object-oriented programming and dynamic allocation are two of these examples, as there is no RTL equivalent of these concepts.

However, this C++ implementation used floating-point type definitions, which use many resources in an FPGA design. For this reason, a fixed-point implementation had to be created in which no floating-point type definitions should occur. This was done by the internally constructed Fixed-point Analyzer & Scaler Tool (FAST) developed for this project, which could analyze the range of values used and generate new fixed-point type definitions based on this analysis. After this, the fixed-point implementation can be used as input for Vivado HLS, after which it can generate a packaged IP to be loaded on an FPGA.

During these implementations, functional correctness is ensured using a testbench of which the testcases are derived from the original algorithm. This testbench can be reused at every stage. The total toolflow is also shown in Figure 3.1.

3.2 Hardware Implementation

Vivado HLS generates an IP package to be loaded on an FPGA. To shorten development time and ensure functional correctness, the original algorithm is divided up into hardware blocks which can be implemented individually using Vivado HLS. The final hardware implementation is implemented using the DYnamic Process LOader (Dyplo) tool developed by Topic Embedded Systems.[18] Dyplo is a middleware solution which

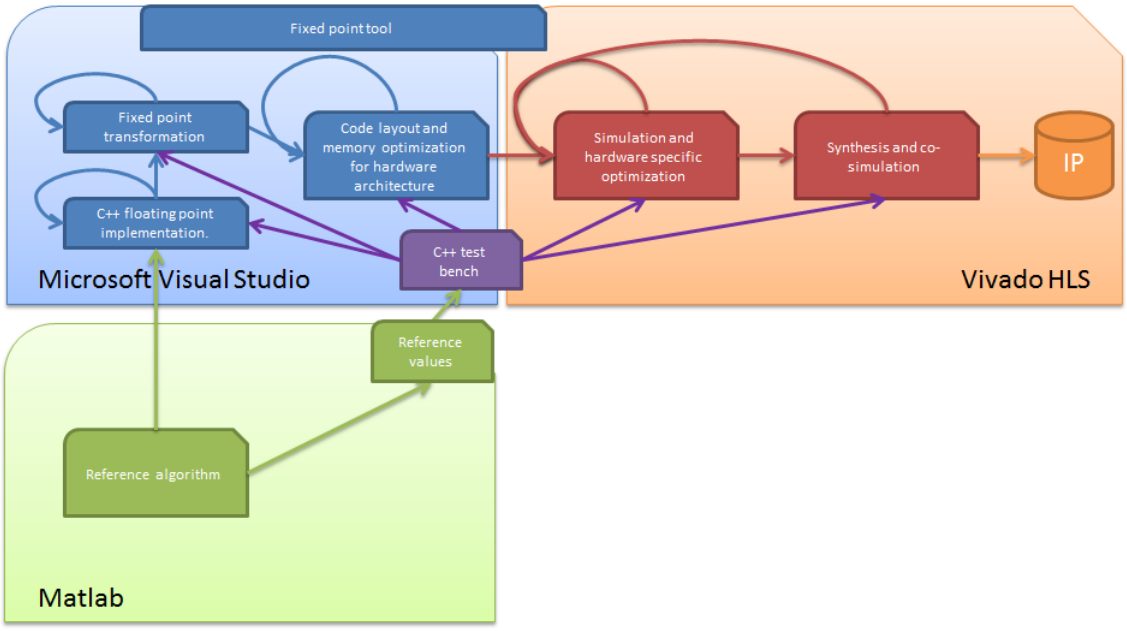

Figure 3.1: Toolflow illustration.
Source:[4]

provides seamless integration of FPGA and software processes in applications. It manages this by viewing the system as a network of process nodes, connected to each other using data queues. These Dyplo nodes can either be implemented in software (CPU nodes) or hardware (RTL nodes). Using this property, a full software-driven hardware development approach is made possible, allowing a piece-by-piece hardware implementation, slowly connecting software nodes to more hardware nodes. Together with using the simulation and co-simulation features in Vivado HLS and previously constructed testbenches, a Hardware-In-the-Loop (HIL) test strategy is used.

For this project, a Florida board incorporated on a Miami System-On-Module (SOM) will be used, which are both developed by Topic as well. The Miami SOM is based on the Xilinx Zynq 7z030 System On Chip (SoC). The Zync 7z030 uses an ARM-based application micro-processor with FPGA logic in a single chip. It integrates all system components required, for example peripherals like an SD-card reader, in one board, which make it very useful for fast development. It can be connected directly to a display, has 1GB of DDR RAM, and supports the Linux OS, so development can be done on the device itself. The Zynq 7z030 contains 530 BRAM_18K blocks, 400 DSP48E blocks, 157200 FFs, and 78600 LUTs.

The software implementation of the original algorithm is frame-based: the entire input frame is stored in memory and can be addressed by coordinates. This means that functions can access every pixel in a frame immediately as the frame is available as soon as the function is called. This can be done as in a general PC enough RAM is available to store these frames completely. Also, such an implementation is not desirable, as entire frames have to be buffered, stored and retrieved, leading up to many clock cycles used

for memory retrieval.

Instead, a stream-based implementation can be used in the RTL design using the stream libraries available in Vivado HLS. A stream-based implementation does not buffer the entire image; it reads in one pixel, processes it, and then streams it to output. This decreases RAM usage significantly and promotes higher throughput. Yet a problem of a stream-based implementation is that in many image processing filters kernels are used which need pixels both horizontally and vertically: these are not available at all times and need to be streamed in before processing. Another similar problem is created because of boundary conditions. These problems necessitate restructuring the implementation of the hardware blocks. This results in more complex code and relatively long development time.

An RTL design, defined for example in VHDL or Verilog, is a description of hardware modules. Designing a hardware architecture is therefore essential to ensure a functional device conforming to all requirements. Vivado HLS can generate a rudimentary architecture but is often not optimized. Looking back at the original requirements, it can be seen that one of the requirements for the RTL design is throughput. Because of this reason, a pipelined architecture seems like the best choice for the final hardware architecture, as it has the promise of the highest throughput, using the same structure as a stream-based implementation. It has the added advantage of a decreased amount of memory blocks, which will make it easier to fit the final design on the FPGA board.

Although the generated RTL designs of each hardware block may be verified to be correct using testbenches constructed earlier, the implemented hardware blocks have to adhere to two additional requirements.

- Area usage. One of these limitations of the implementation of a Dyplo node is that its resource usage can't exceed an eighth of the total FPGA. This means that the usage of all individual parts available on the FPGA, like LUTs or BRAMs, may not exceed 12,5%. In some cases this may lead to increased latency for the hardware block to fulfill this area usage requirement.
- Throughput. Measuring the throughput of a pipelined design is most effectively done using the Initiation Interval (II). Previous, a throughput requirement of the total hardware implementation was defined as 15 FPS. The target frequency for the Miami board is 150MHz. The frame dimensions used are 960 by 960 pixels. The target throughput is $15 fps * 960 * 960 = 13824000$ pixels per second. When dividing clock speed over the expected throughput, the required amount of clock cycles per pixel can be calculated. This amounts to $149925037,48/13824000 \approx 10.9$ clocks per pixel. This means that to achieve 15 FPS in a fully pipelined design, every pixel must take no more than 10,9 clock cycles to execute. So the maximum II the final hardware implementation must have is 10, from which follows that every hardware block should have an II of 10 or lower, as the highest II in a pipelined design dictates the total II.

These requirements may not be fulfilled by the RTL designs generated by Vivado HLS. In order to meet these requirements, certain optimizations can be applied to optimize the designs. Examples include: pipelining; arithmetic reduction or removal; loop directives; resolving memory dependencies; restructuring code; tighter fixed-point bit-widths; replacing functions with Look-Up Tables (LUTs); using the Vivado HLs libraries

for certain standard functions; using arbitrary precisions for integer values; in-lining functions; or using alternative options associated with certain Vivado HLS directives.

A summary of the optimized synthesis results for a selection of the hardware block can be found in Figure 3.2. It can be seen that in general, optimization results have been largely successful, and the area usage, latency and II have been decreased compared to their original implementations.

Due to time restrictions the only a part of the filter hass been implemented. Yet four hardware blocks have been converted to Dyplo nodes and have been executed on the Zynq board. Using Direct Memory Access (DMA), data was transferred to each node's input streams and their function was started. The execution was then measured by the software side: the clock was started when the first value entered the node, and then stopped when the last value exited the output stream. The output is then verified to be correct using the same testbenches used with Vivado HLS.

Table 3.1 shows all timings of every node. An Initiation Interval (II) close to 1 has been achieved for the blocks A and B, and an II close to 4 for Block D. Block C consisted of very complex code and was difficult to optimize. Any differences between expected and achieved latency can be explained due to the latency overhead of the Dyplo backplane. Note for example that for Block C and Block D a large difference exists; this is caused by a buffering in the input data queue of the Dyplo nodal framework. Unfortunately this could not be resolved at run-time, yet this only influences the input to the RTL block designs – the designs themselves are correct.

Table 3.1: Node timings of Dyplo node implementations running on the Florida FPGA board, using a software interface to transfer values and calculate timings.

	Block A	**Block B**	**Block C**	**Block D**
ms	11.590	11.575	1627.173	37.03
pixels	921600	921600	921600	921600
per pixel [ns]	12.57595486	12.55967882	1765.595703	40.18012153
clock period [ns]	10	10	10	10
clk per pixel	1.257595486	1.255967882	176.5595703	4.018012153
clk in total	1159000	1157500	162717300	3703000
expected latency	1154883	1154878	161785827	927360

First Implementation							
Max Latency	1846083	2349045	1170229	215	669	446925262	
Main Loop Latency	1844157	2341660	1165364	61	385	446919484	
Main Loop Iteration Latency	1923	2452	2438	38	77	467489	
Max II	2	2	1	1	1	-	
BRAM_18K blocks	6	11	6	0	12	47	
DSP48E blocks	0	114	68	270	143	1492	
FF blocks	798	10775	5301	63044	13903	271897	
LUT blocks	1558	14861	10087	64830	18955	298229	
BRAM_18K usage	1.13%	2.08%	1.13%	0.00%	2.26%	8.87%	
DSP48E usage	0.00%	28.50%	17.00%	67.50%	35.75%	373.00%	
FF usage	0.51%	6.85%	3.37%	40.10%	8.84%	172.96%	
LUT usage	1.98%	18.91%	12.83%	82.48%	24.12%	379.43%	
Final Implementation							
Max Latency	927360	1154883	1154879	38	125	161785827	
Main Loop Latency	925434	1147662	1150068	37	45	161780055	
Main Loop Iteration Latency	965	2406	2406	14	22	169226	
Max II	1	2	1	1	1	-	
BRAM_18K blocks	8	7	4	0	4	22	
DSP48E blocks	0	0	0	12	8	24	
FF blocks	1443	2066	1041	604	1474	2660	
LUT blocks	1991	3395	2414	1206	2458	4841	
BRAM_18K usage	1.51%	1.32%	0.75%	0.00%	0.75%	4.15%	
DSP48E usage	0.00%	0.00%	0.00%	3.00%	2.00%	6.00%	
FF usage	0.92%	1.31%	0.66%	0.38%	0.94%	1.69%	
LUT usage	2.53%	4.32%	3.07%	1.53%	3.13%	6.16%	
Improvement							
Max Latency Change	-49.77%	-50.84%	-1.31%	-82.33%	-81.32%	-63.80%	
Average Area Change	45.72%	-78.77%	-77.63%	-98.58%	-88.05%	-98.68%	

Figure 3.2: Hardware block results.

4 Results (Summary)

This section details the results found during the research involved in this thesis. The focus of this thesis is to answer the following research question defined in Section 1.3.
Is it possible to reduce the development time of software-to-hardware conversion of image processing algorithms for medical imaging while maintaining optimized performance and functional correctness?
To answer this question, a reproducible workflow has to be produced that adheres to the following four requirements defined in Section 1.3.

- This workflow must be able produce an embedded solution based on a software design.
- The hardware implementation produced by this workflow must have output identical to the results of the original software implementation.
- The hardware implementation produced by this workflow must have a throughput which will not disturb the work of the operating physician.
- The design effort required to execute this workflow, measured in time, has to be lower than manually designing the entire RTL design.

The result of this research is then a reproducible workflow, defined in tools and strategies. To prove the validity of such a workflow, a proof of concept has to be constructed that implements a software to hardware translation. This concept implementation is chosen to be a noise reduction and edge enhancement algorithm used in X-ray systems in Philips Healthcare, as seen in Section 2. This algorithm is originally implemented in MATLAB and needs to be converted to an RTL design implementable on FPGA technology, as defined in Section 1.3.

Over the course of this research, the result of this research can then be defined as the following workflow, represented as the following list of steps to be taken in order.

1. Inspect the MATLAB model using MATLAB.
   1.1. Study its code structure, and divide functionality into stages and functions.
   1.2. Check input and output types.
2. Set up inter-implementational test-bench to ensure functional correctness.
   2.1. Define test cases using inter-implementational data format.
3. Translate the MATLAB model into a floating-point C++ implementation manually using Visual Studio.
   3.1. Define the input and output environment.

3.2. Construct test-benches following the general test-bench setup. Construct test-cases.

3.3. Examine implementation-specific considerations.

4. Create fixed-point C++ implementation using Visual Studio.

4.1. Implement fixed-point data types using Vivado HLS fixed-point libraries. Start with large bit-widths. Verify functional correctness using test-bench.

4.2. Use the FAST tool to ascertain lowest fixed-point bit-widths.

4.3. Implement new bit-widths while verifying functional correctness.

5. Investigate design space of the hardware implementation using Visual Studio in combination with Vivado HLS libraries.

5.1. Define input/output environment of RTL design.

5.2. Divide the C++ fixed-point design into hardware blocks, each representing a different filter. These hardware blocks are defined by the functions in the original code.

5.3. Convert hardware blocks from frame-based data structure to stream-based data structure.

5.4. Determine a system architecture by a selection based on performance characteristics.

5.5. Construct testbenches based on the fixed-point C++ testbenches to verify the functional correctness of each hardware block.

6. Create RTL design using Vivado HLS. Implement and run it on the Xilinx Zync 7z030 using the Dyplo framework.

6.1. Define requirements for each block requirement.

6.2. Optimize hardware blocks to adhere to requirements.

6.3. Convert hardware blocks into Dyplo nodes and link them together.

6.4. Run RTL design on Xilinx Zync 7z030 board by sending input data through DMA to individual Dyplo nodes.

5 Discussion (Summary)

This section discusses the results during the research of this thesis, which can be found in Section 4. First, it will be verified if the original research question has been answered. Next, each step in the workflow itself, defined in the results, is discussed.

5.1 Research Question

As a proof of concept, using the workflow seen in Section 4, an RTL design of a noise reduction and edge enhancement filter originally defined in MATLAB has been implemented. This workflow is subject to four different requirements which are derived from the research question as seen in Section 1.3. The first requirement is that an embedded solution should be able to be produced using this workflow. According to the results found by implementing the final hardware implementation, an RTL design can be created using this workflow, and thus the workflow adheres to the first requirement.

The second requirement indicates that the workflow must be able to produce an RTL implementation which generates an output which is as close to the output of the original algorithm as possible. As the final RTL design uses fixed-point values as opposed to floating-point values used in the original algorithm, the exact same results require very high precision to be reproduced. Instead, the final RTL design is stated to be functionally correct if every pixel of the output image of the final implementation is accurate up to 0.5%. This percentage is achieved using an inter-implementational testbench for every individual hardware block, using test-cases constructed from the original algorithm. This results in the workflow being able to adhere to this requirement as well.

The third requirement states that the workflow must be able to produce an RTL implementation which has a throughput which does not hinder the operating physician. This throughput is defined as 15 FPS at 1K images, and 7,5 FPS at 2K images. Using for the hardware environment the Dyplo framework and optimizations in Vivado HLS, it is estimated that this throughput can be achieved for the final RTL implementation. The workflow also adheres to this requirement.

The final requirement needs the overall design time used for executing this workflow to be lower than manually coding a hardware design. Most of the investigation and preparation steps used for the software simulation would not be significantly different for manually creating an RTL design. Yet, using the CAD tool Vivado HLS means that automatic RTL generation can be applied. This is significantly faster than manually creating an RTL design.[19] [20] Using Vivado HLS, a considerable part of the original algorithm has been implemented in an RTL design in the space of four months by a single student. A very rough estimate in development time decrease can be made by comparing the workflow to manually creating an RTL design, although it is very difficult to compare these two development methods. For example, a similar project is estimated

to take approximately 2 years in a team of at least two senior FPGA engineers. The development time difference can then be quantified as $((4-(24*2))/(24*2))*100 \approx$-92%. This leads to a workflow which can produce an implementation approximately ten times faster. In addition, RTL designs created using Vivado HLS are portable and easily modified. These characteristics indicate that the development time of this workflow is significantly lower, fulfilling this requirement.

Because the workflow adheres to these requirements, the workflow found in this thesis can perform a hardware-to-software translation, in which the performance of the final hardware implementation can be optimized, the output can be verified to be functionally correct, and the development time has been reduced. So, it can be concluded that the answer to the research question has been found: **it is possible to reduce the development time of software-to-hardware conversion of image processing algorithms for medical imaging while maintaining optimized performance and functional correctness**.

5.2 Discussion Results

The results of this thesis, in form of a reproducible workflow, will be discussed here. Most of the discussion relates to development time which ma have been decreased in each step, as this is the primary characterstic of the workflow. This is done by discussing every general step of the workflow individually.

Step 1 *Inspect the MATLAB model using MATLAB.* In hindsight, a large part of the development time could be shortened by a more in-depth study of the original algorithm before starting with the initial conversion. This is not limited only to MATLAB models. Although the mathematical refinement of the algorithm was not the focus of this research, both performance would have been improved and development time decreased if certain parts of the original algorithm had been studied more and replaced by faster counterparts. Early on for example, the bottleneck function of the algorithm had been discovered. Optimizing this function lead to the most significant speed-up. Other optimizations would include the replacement certain complex functions in a relatively simple Look-Up Table.

Step 2 *Set up inter-implementational test-bench to ensure functional correctness.* Over the course of the entire project, creating the testbenches for each implementation has taken up much of the development time as well. This includes adjusting and adding individual testcases to provide for more and better testing. Although the Test-Driven Development approach, combined with the Agile methodology, quickly led to intermediate demos, this step in the workflow had to be updated consistently over the entire project, requiring a lot of development time. In the end, though, it saved a lot of development time as it made debugging faster.

Step 3 *Translate the MATLAB model into a floating-point C++ implementation manually using Visual Studio.* The C++ implementation that is used by Vivado HLS is bound by certain restrictions. These restrictions mostly relate to data structure and input/output consideration. It is advised that a software design capable of handling such restriction is created in the earliest stage as possible. This may save a lot of development time.

Step 4 *Create fixed-point C++ implementation using Visual Studio.* Transitioning from floating-point types to fixed-point types can be very difficult. The use of predefined testcases greatly reduced the workload in this effort. In this project, an internal tool has been developed for the express purpose to investigate ranges and values of certain parameters. Other investigative tools exist for this purpose and may suit another individual project even better.

Step 5 *Investigate design space of the hardware implementation using Visual Studio in combination with Vivado HLS libraries.* Most of the development time concerned with converting the C++ code to RTL design was spent here, as restructuring the code in such a way that it would fit in the rest of the hardware environment took much development effort. This includes the frame-based implementation to stream-based implementation. It depends on the complexity of the project, but checking the hardware environment earlier on in the project may decrease overall development, as previous implementations may be designed differently.

Step 6 *Create RTL design using Vivado HLS. Implement and run it on the Xilinx Zync 7z030 using the Dyplo framework.* Vivado HLS is a very powerful and intuitive tool, yet the best results using it were found in generating RTL designs for smaller hardware blocks. The best option for this project was to decide on an architecture outside of Vivado HLS, instead of letting Vivado HLS decide one. Although initial RTL designs are easily generated, a certain amount of experience is needed to fine-tune the performance of these designs, which takes much development effort. Much development time was saved however by an in-built function of Vivado HLS called co-simulation, which takes a C testbench and uses this to test the RTL design automatically. This makes Vivado HLS more flexible than a pure RTL design: making changes to the algorithm is easier and less time-consuming, thus reducing the maintenance effort.

6 Conclusions & Future Work

Manually designing a complete RTL design for a complex algorithm takes up much development time. This development can be significantly reduced by using Vivado HLS tool to automatically generate an RTL design. However, certain complications arise in the resulting performance of such a generated design. Although functional correctness can be guaranteed using test-benches, a manual RTL design proved to outperform the generated RTL design in many cases. Yet, using optimizations and restructuring the code significantly, a high performance close to a manual design could be reached, at least for individual, less complex filters. These designs could be ported to several different FPGA boards without comprimising performance, ensuring portability.

This thesis has created a workflow in which an optimized, functionally correct RTL design can be generated from a software implementation. Using this workflow, development time can be decreased by roughly ten times compared to manually creating an RTL design, while also decreasing the difficulty of modifying the RTL design. A proof-of-concept is shown implementing a noise reduction and edge enhancement algorithm found in X-ray systems in use by Philips Healthcare. The algorithm was originally defined in a MATLAB model, and using various tools like Visual Studio, an internally developed fixed-point analysis tool and Vivado HLS, a final implementation was produced running on the Zync 7z030 SoM using the Dyplo framework. This implementation did not feature the entire original algorithm, yet it has been proven that this could be implemented on basis of the intermediate results found.

Future work should be done on implementing the entire noise reduction and edge enhancement filter to produce final performance results. Another implementation used in a workflow to reduce development time, and ensuring portability and performance, would be an OpenCL implementation. An OpenCL implementation can also be used with another Xilinx tool called SDAccel [21], which incorporates Vivado HLS and OpenCL implementation results. SDAccel uses OpenCL kernels, but combines these with generated RTL designs to executed different functions of algorithms. This approach could generate lower development time and increase performance even further.

Bibliography

[1] [Online]. Available: /www.onslownet.school.nz/

[2] [Online]. Available: http://www.toshiba-tetd.co.jp

[3] P. Healthcare, "D5.1 healthcare demonstrator early prototype [confidential]," Philips Healthcare, Tech. Rep., 2015.

[4] ——, "D4.1 application framework control [confidential]," Philips Healthcare, Tech. Rep., 2016.

[5] G. D. Michell and R. K., "Hardware/software co-design," *Proceedings of the IEEE*, vol. 85, no. 3, pp. 349–365, Mar 1997.

[6] Y. Li, T. Callahan, E. Darnell, R. Harr, U. Kurkure, and J. Stockwood, "Hardware-software co-design of embedded reconfigurable architectures," in *Proceedings of the 37th Annual Design Automation Conference*, ser. DAC '00. New York, NY, USA: ACM, 2000, pp. 507–512. [Online]. Available: http://doi.acm.org/10.1145/337292.337559

[7] J. R. Hauser and J. Wawrzynek, "Garp: A mips processor with a reconfigurable coprocessor," in *Field-Programmable Custom Computing Machines, 1997. Proceedings., The 5th Annual IEEE Symposium on*. IEEE, 1997, pp. 12–21.

[8] A. Darabiha, J. Rose, and J. W. Maclean, "Video-rate stereo depth measurement on programmable hardware," in *Computer Vision and Pattern Recognition, 2003. Proceedings. 2003 IEEE Computer Society Conference on*, vol. 1, June 2003, pp. I–203–I–210 vol.1.

[9] R. K. Gupta, *Co-synthesis of hardware and software for digital embedded systems*. Springer Science & Business Media, 2012, vol. 329.

[10] [Online]. Available: http://www.almarvi.eu/

[11] S. Asano, T. Maruyama, and Y. Yamaguchi, "Performance comparison of fpga, gpu and cpu in image processing," in *2009 international conference on field programmable logic and applications*. IEEE, 2009, pp. 126–131.

[12] R. K. Gupta, C. N. Coelho Jr, and G. De Micheli, "Synthesis and simulation of digital systems containing interacting hardware and software components," in *Proceedings of the 29th ACM/IEEE Design Automation Conference*. IEEE Computer Society Press, 1992, pp. 225–230.

[13] M. V. G. Rao, P. R. Kumar, and A. M. Prasad, "Implementation of real time image processing system with fpga and dsp," in *2016 International Conference on Microelectronics, Computing and Communications (MicroCom)*, Jan 2016, pp. 1–4.

[14] P. Wang and J. McAllister, "Streaming elements for fpga signal and image processing accelerators," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 24, no. 6, pp. 2262–2274, June 2016.

[15] Altera, "Reliability report 60 2h 2015," Altera, Tech. Rep., 2016.

[16] Xilinx, "Device reliability report second half 2015 ug116 (v10.4)," Xilinx, Tech. Rep., 2016.

[17] L. Li and A. M. Wyrwicz, "Design of an mr image processing module on an fpga chip," *Journal of Magnetic Resonance*, vol. 255, pp. 51–58, 2015.

[18] [Online]. Available: http://topicembeddedproducts.com/products/dyplo/

[19] Xilinx, "Introduction to fpga design with vivado high-level synthesis ug998 (v1.0)," Xilinx, Tech. Rep., 2013.

[20] [Online]. Available: http://www.xilinx.com/support/documentation/sw_manuals/ug998-vivado-intro-fpga-design-hls.pdf

[21] [Online]. Available: http://www.xilinx.com/products/design-tools/software-zone/sdaccel.html