Vrije Universiteit Brussel

Faculty of Applied Sciences and Engineering
INDI Department

Performance Analysis of a Real-Time Video Processing System

Graduation thesis submitted in partial fulfillment of the requirements for the degree of Master of Science in Applied Engineering: Electronics-ICT

Frank Vanbever

Promotors: Dr. Ir. Erik D'Hollander
Dr. An Braeken

Advisors: In. Bruno Tiago Da Silva Gomes

January 2014

Abstract

Preface

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Up until halfway the first decade of the new millennium it was possible to gain computing performance whilst also being able to maintain the sequential programming paradigm. This was due to Moore's law, stating that the number of transistors on integrated circuits double approximately every two years. There was no need for research into explicit parallelism because the next generation of computing devices was just around the corner. This new generation would increase the performance with zero effort required from the programmer.To perpetuate the sequential programming paradigm several innovations such as multiple issue, deep pipelines and out of order execution were introduced. These optimizations were inefficient in both the use of transistors and power. Eventually it became impossible to progress any further whilst still supporting the sequential paradigm. The integrated circuit industry was unable to continue decreasing the size of MOSFETs whilst continuing to increase the clock frequency. The industry had hit what is called the power wall. The solution to this problem was to turn to parallel processors, meaning that there is more than one processing unit working at a time. A lot of real world applications are parallel, and hardware can be made parallel with relative ease. The problem lies in the programming model, how to exploit this parallelism and make programming for these parallel architectures easier and transparent for the programmer.

1.1 Computing Components

1.1.1 Multicore Processor

The multicore processor is the solution presented for the aforementioned problems by the traditional CPU manufacturers such as Intel and AMD. The idea behind this type of processor is to place a number of cores (currently up to eight) on

the same die. This presents a compromise between maintaining sequential performance whilst also providing a certain advantage of parallel processing. Parallel programming for these processors presents certain challenges, whilst their modest parallelism cannot provide a dramatic improvement in power performance. Multicore processors are unlikely to be a one-size-fits-all solution to the parallel problem.[4]

1.1.2 Graphics Processing Units

Graphics processing units are a type of coprocessor used in traditional computers to process images for output to the display. However recently there has been an increased interest in the GPGPU, a general purpose graphics processing unit. These processors implement a different paradigm, namely the manycore paradigm. A GPU is a processor with hundreds single instruction multiple data cores, each of which is heavily multi-threaded. Because of this large amount of cores the FLOPS (floating point operations per second) is unrivaled[14]. GPU's, due to their SIMD (single instruction multiple data) nature present some problems, conditional execution paths for example, present a serious overhead on the GPU. GPU's are programmed with either OpenCL (open standard) or CUDA (proprietary to Nvidia)

1.1.3 Field Programmable Gate Array

FPGAs are devices containing a vast amount of configurable logic linked by programmable connections. This logic is comprised of lookup tables (LUTs) grouped together into configurable logic blocks. Any combinatorial function can be programmed into these LUT's. Next to these uncommitted logic blocks a typical FPGA also contains several blocks with a specific function such as block ram and DSP multipliers. FPGAs are an interesting competitor in the parallel processing field because they are not constrained by the Von Neuman architecture. FPGAs follow the dataflow paradigm in which the data *flows* through the logic. Implementing a data-flow is inherently parallel. The different stages in the datapath can also be made effectively sequential making the datapath a pipeline. The fine grained nature of FPGAs also means that the bitwidth can be adapted to the application.

1.2 Berkeley dwarfs

Image processing algorithms are very compute intensive. These makes them prime targets for exploiting parallelism and implementing them on parallel architectures. Which platform is the best fit is dependent on both the algorithm and

the data. A common method to subdivide parallel algorithms is presented in , the so called *dwarfs*. These 13 dwarfs are classes of algorithms in which the membership is defined by a similarity in computation and data movement.These 13 dwarfs are classes of algorithms in which the membership is defined by a similarity in computation and data movement. The dwarfs are:

1. Dense Linear Algebra
2. Sparse Linear Algebra
3. Spectral Methods
4. N-Body Methods
5. Structured Grids
6. Unstructured Grids
7. MapReduce
8. Combinational Logic
9. Graph Traversal
10. Dynamic Programming
11. Backtrack and Branch-and-Bound
12. Graphical Models
13. Finite State Machines

A thorough review of these dwarfs and what kind of computation and communication they entail goes beyond the scope of this document. An updated view can be found in [5]. Finding out which platform is most suited for a dwarf is a labor intensive task. In [13] a theoretical analysis of dwarf performance on different accelerators in heterogeneous systems is given. A first point of note is that GPGPU's are unsurpassed in floating point arithmetic. Fixed point numbers are a way to overcome this problem. Another point to note is that conditional elements and costly communication can wreak havoc on the accelerator's performance. In Figure 1.1 the analysis is represented by a Venn diagram. In this diagram '*' denotes fixed point operations whilst '^' denotes floating point operations.

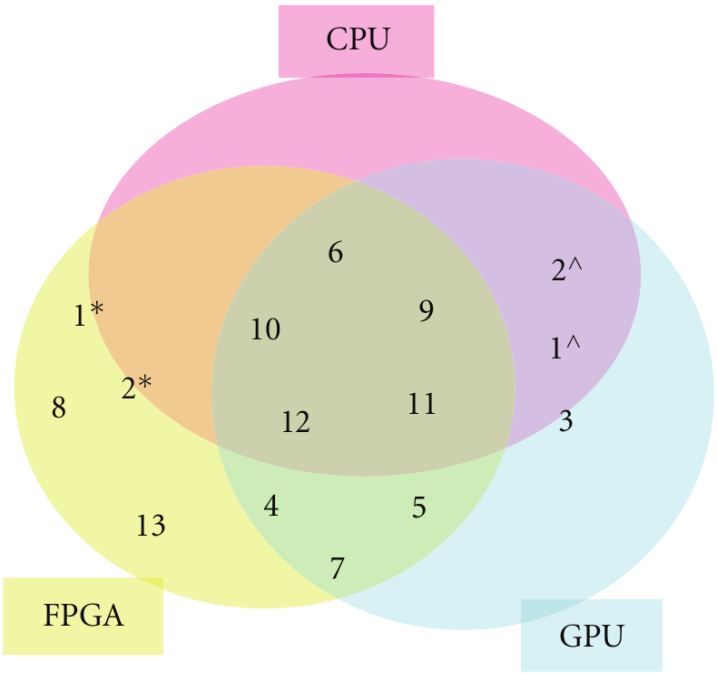

Figure 1.1: Analysis of different hardware accelerators in regards to performance on a certain dwarf [13]

1.3 GUDI Project

This thesis is inspired by the work of the GUDI project. GUDI is an acronym for "A Combined **G**P-GP**U**/FPGA **D**esktop for accelerating **I**mage processing applications". The research starts from the observation that there is a large need for computing power to process data using computationally intensive image processing algorithms. Conventional Off-the-shelf Desktop computers don't have the necessary processing power to satisfy this demand. A lot of image processing algorithms exhibit parallelism which can be exploited by the right architecture. Two such massively-parallel architectures are GP-GPU's and FPGA's. The GUDI project's goal is to investigate the possibilities and limitations of a computer with such a heterogeneous architecture. It is an investigation into the technologies and development tools and their performance in different situations. The means through which this is done is through the implementation and performance measurement of algorithms. The ultimate goal is to split an algorithm into several parts, executed on the technology (CPU,GPU,FPGA) most fit for the job so as to ensure optimal speed-ups.

Chapter 2

Platform Overview

2.1 Zynq-7000

The Zynq-7000 System on Chip combines a dual core ARM Cortex-A9 with Xilinx programmable logic in a single device. This combination of a CPU and an FPGA on the same device is not a new phenomenon, with examples of previous generations being the PowerPC based Xilinx Virtex-II Pro and some models of the Virtex 4 and Virtex 5 series FPGA's. A notable differences between these generations is the shift from PowerPC based architectures to ARM based architectures. Also, a notable shift in emphasis from HDL centered design to a more programmer centric view with an emphasis on high level languages can be seen.

2.1.1 Processing System

The Zynq-7000 series SoC is split into two parts: The processing system (PS) and the programmable logic (PL). The Processing system contains an Application Processor Unit (APU), memory interfaces and I/O peripherals.

APU The APU is a Dual ARM Cortex-A9 CPU which implements version 7 of the ARM ISA as well as Thumb and Jazelle instruction sets. Each core has a NEON Media Processing Engine supporting SIMD vector and scalar single-precision floating-point and integer computation and scalar double-precision floating-point computation. Each core has 32 KB instruction and 32 KB data caches and there is 512 KB shared L2 cache and 256 KB of on-chip SRAM memory. The APU also has a snoop control unit to maintain L1 and L2 coherency. This snoop control unit also controls the Accelerator Coherency Port, a 64-bit AXI slave port connected to the programmable logic. The PL acts as the master and the PS as slave. This allows direct communication between the PS and the PL through the

L2 caches or on chip memory with guaranteed coherency. The APU also has an on-board 8-channel DMA controller with 4-channels reserved for PS to/from memory and 4 for PL to/from memory transfers. The Processing system also contains an interrupt controller.

Memory Controller The Memory controller supports a number of memory technologies. The system has a DDR controller which supports DDR2 and DDR3 memory, a Quad-SPI controller which converts normal memory read operations to SPI and vice versa, and a Static Memory Controller which supports NAND and SRAM/NOR type memory.

I/O Peripherals The Processing system contains quite a lot of industry standard I/O peripherals for external data communication.

- GPIO
- 2 Gigabit Ethernet Controllers
- 2 USB controllers
- 2 SD/SDIO controllers
- 2 SPI controllers
- 2 CAN controllers
- 2 UART controllers
- 2 I^2C controllers

These peripherals are connected to multiplexed I/O buffers which enable to externalize these signals to up to 54 pins. If there is a need for more I/O pins the signals can be routed into the PL through the extended MIO, where they can be routed directly to package pins or peripherals in the PL.

2.1.2 Programmable Logic

The programmable logic provides the same functionality that can be expected from a Xilinx FPGA. The PL in 7z010 and 7z020 Zynq SoCs is based on Artix-7 FPGAs whereas the PL in 7z030, 7z045 and 7z100 SoCs is based on Kintex-7 FPGA logic. This PL can be coupled through a couple of different interconnects, with varying degrees of interconnectedness between the PL and the PS. Of note here is that the PS has to be booted first and the PL logic has to be configured from the PL at boot or at a later time. This is another example of the shift to a more software centered view. The system has all the features one can expect from an FPGA: configurable logic blocks with look-up tables, a number of 36 KB block RAMs, DSP348E slices and configurable IO. The PL side also contains an Analog to Digital converter, and in the larger varieties of the Zynq SoC an integrated PCI Express block.

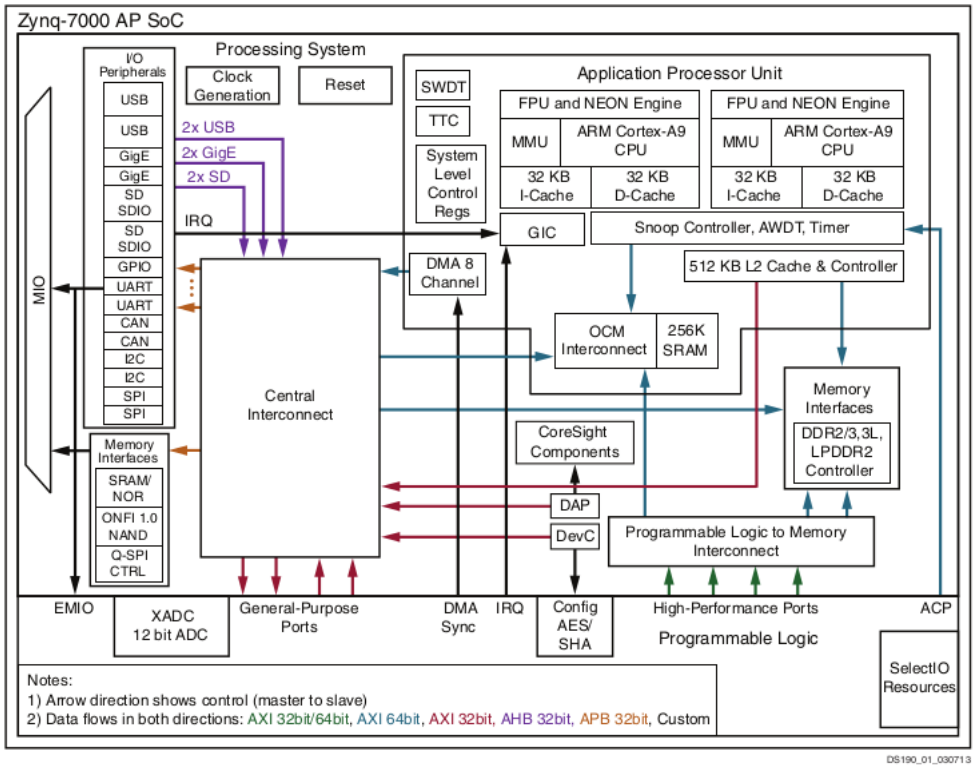

Figure 2.1: Zynq -7000 SoC overview [3]

2.1.3 Interconnect

The interconnect system is located in the PS but because of it's influence on performance it warrants its own section. The interconnect system is comprised of a number of switches to connect the different parts of the system using the AXI point-to-point protocol. The AXI protocol is part of the ARM Advanced Microcontroller Bus Architecture version 3.0. These AXI interconnects are the primary means of communication between the PS and the PL. There are a number of different interface ports between the PS and the PL:

AXI_HP There are four AXI_HP interfaces, connecting PL masters with high bandwidth datapaths to the DDR and OCM memories. Each interface is buffered with 2 FIFOs and is configurable to be 32 or 64 bits wide.

AXI_GP The four general purpose AXI_GP ports are divided into 2 master ports and 2 slave ports. These ports don't have FIFO buffering which makes them less suitable for high performance use.

AXI_ACP The Accelerator coherency port is a 64-bit AXI slave interface that directly connects the PL to the APU caches. This is done through the snoop control unit and can enforce coherency if requested.

The actual interconnection is implemented through a number of switches. Amongst these are the snoop control unit, the L2 cache controller and a couple of ARM NIC-301 based interconnect switches.

Snoop Control Unit Although the SCU is in essence not a switch, its behavior in regards to the transfer of data from its AXI slave ports to its AXI master ports makes it function as a switch.

Central Interconnect The central interconnect is the core of the interconnect network in the Zynq SoC.

Master Interconnect The master interconnect connects the Master switches the traffic from the AXI_GP ports as well as traffic coming from the device configuration core and the debug access port.

Slave Interconnect The slave interconnect switches traffic coming from the central interconnect to AXI_GP, I/O peripherals, APB connections, etc.

Memory Interconnect The memory interconnect switches high speed traffic coming from the AXI_HP ports to DDR and on-chip RAM.

OCM Interconnect The on-chip memory connect switches the traffic from the central interconnect and the memory interconnect.

A diagram of how these interconnects are organized can be found in figure 2.2

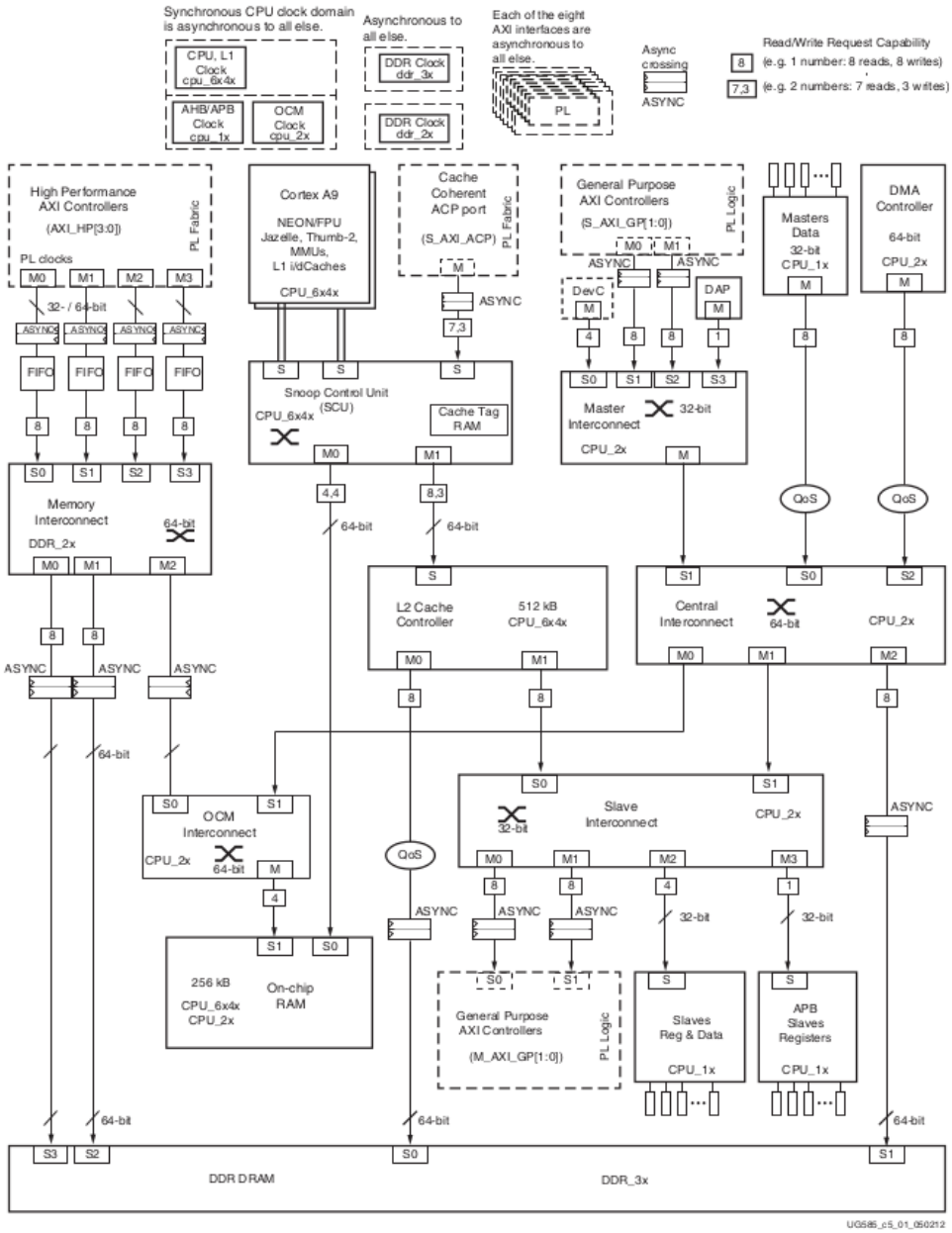

Figure 2.2: Zynq Interconnect System Block Diagram [3]

2.2 Toolchain

Because of the heterogeneous nature of the Zynq Soc there are a number of tools necessary to implement a design. In this section the tools used for this thesis are discussed.

PlanAhead PlanAhead is the main interface for the hardware side of a project. It allows a designer to bring together VHDL or Verilog code, IP-cores, embedded designs from Xilinx Platform studio and DSP designs from System generator. It also integrates with ISE Simulator to allow the functional verification of HDL code and IP. PlanAhead also allows the insertion of Chipscope cores to debug RTL designs.

Xilinx Platform Studio (XPS) Xilinx platform studio is a graphical tool that allows a designer to build embedded processor systems including IP-cores. The connections between peripherals in the PL and the PS are made using this application.

Vivado HLS Vivado HLS is a high-level synthesis tool that converts C, C++ and SystemC to synthesizable hardware. It can export this hardware into a number of formats, among which the PCore format for XPS. More on Vivado HLS can be found in section 3.2

Xilinx SDK Xilinx' Software Development Kit is an eclipse based integrated development environment targeting the ARM core in Zynq or the Microblaze soft-core processor. It includes a complete GNU based compiler toolchain in the form of the Mentor Sourcery Codebench Lite - Xilinx edition, which also incorporates debugging and profiling tools. The SDK also has plug-ins which make it aware of the peripherals placed in the PL and drivers for Xilinx supplied IP-cores.
The necessary steps for developing a design for the Zynq SoC are as follows:

1. Start the project in PlanAhead, specify the parameters of the hardware you're developing for and create a new embedded design
2. Independent of the PlanAhead project, implement the algorithm using Vivado HLS so it satisfies all design constraints. Export the implementation as an IP-core suitable for use in XPS.
3. In XPS, add the Vivado HLS generated IP core to the system along with other IP-cores. Make all the necessary interconnections.

4. Add the necessary floor-planning constraints in PlanAhead. Synthesize the system, perform the implementation step on the system and generate a bitstream. Correct any errors that show up during these steps until the system is error-free.

5. Export the system to SDK. Develop the embedded software in the IDE using the available drivers. Create a boot image combining the software and the hardware and launch the application on the hardware.

These steps are visually represented in figure 2.3

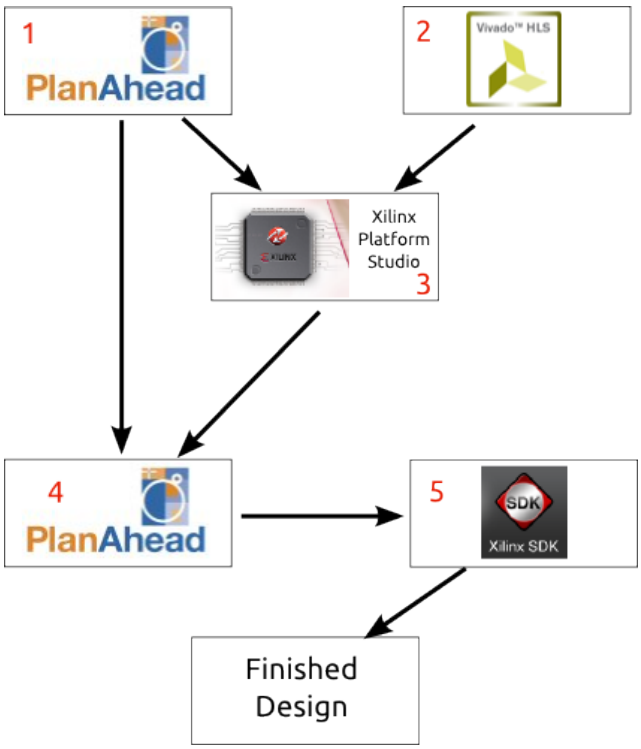

Figure 2.3: Diagram of the Zynq toolchain

Xilinx provides an alternative toolchain in its Vivado Design Suite. It supports the Zynq-7000 series since the release of version 2013.3 in October 2013. This software combines the functions of PlanAhead and XPS into one application. For new designs using 7-series and Zynq devices Xilinx recommends using Vivado Toolchain, With the old ISE toolchain still being available for backward compatibility reasons.[2].

2.3 Base Targeted Reference Design 14.5

Zynq is a powerful but complex architecture requiring a designer with a diverse skill set. Building a real-time video processing system from scratch would involve connecting multiple IP-cores to each other and to the PS, as well as writing efficient applications that use the implemented hardware. Because this falls beyond the scope of this thesis an existing design, Base Targeted reference Design version 14.5, was selected to do a performance analysis on.

2.3.1 Hardware

The hardware side can be split into 3 stages:

The first stage starts with the *fmc_imageon_hdmi_in* IP-core. This core extracts the horizontal and vertical blanking signals from the YCrCb 4:2:2 input it receives from the FMC-Imageon Module. In turn this core is connected to the *Video In to AXI4-Stream* IP-core. This core handles the clock boundary between the video clokc domain and the AXI4-Stream clock domain. The preservation of timing information is guaranteed by a Video Timing Core. The AXI4-stream is routed into a *Test Pattern Generator* IP-core which can generate a test-pattern or act as a pass-through for the video signal, depending on the configuration. The data coming from the TPG is fed into a *Chroma Resampler* IP-core which converts the YCrCb 4:2:2 formatted signal into a YCrCb 4:4:4 signal. HDMI uses the chroma subsampling to reduce the amount of data that needs to be transmitted. This can be done because the human eye is not as sensitive for the Chroma components as it is for the Luminance component. The upsampling is necessary for the following step, which is the colorspace conversion performed by the *YCrCb to RGB Color-Space Converter* IP-core, which converts the YCrCb signal to RGB video as this is the format required by the following steps in the video processing pipeline. The video signal gets routed into a *Video DMA*controller which writes the data to memory using an *AXI Interconnect* IP-core connected to the PS' S_AXI_HP0 port.

The second stage has a second *AXI Interconnect* IP-core connected to the PS' S_AXI_HP2 port. A *Video DMA* controller reads data from memory through this interconnect and converts the date from the AXI memory mapped format to an AXI stream format. This gets sent to the Vivado HLS generated Sobel core. This core sends the data back to the Video DMA controller which writes it back to memory through the AXI Interconnect.
The Third and last stage consists of the Xylon logiCVC-ML which is connected to the first AXI Interconnect. This is a video display controller which reads the data from the video memory and converts it into a format suitable for output. It

also generates the control signals for the output.
The IP-cores that use AXI-Lite for their configuration are also connected to a third AXI Interconnect IP-core. This IP-core is connected to the M_AXI_GP0. Through this connection the PS is able to control the functioning of the video processing pipeline. Finally There is an AXI Performance Monitor IP-core also connected to this third interconnect. The Performance monitor monitors the througput on the S_AXI_HP0 and S_AXI_HP2 ports.

2.3.2 Software

On the software side the TRD uses three major components:

- Boot Loader
- Xilinx Linux Kernel
- Application

Boot Loader

The TRD uses the SD card as the boot device. The boot loader is stored in the BOOT.BIN file and performs a number of functions. The system uses a two-stage boot loader. On boot the first stage boot loader gets executed. This performs the necessary initializations that enable the system to load the bitstream into the PL and execute the U-Boot bootloader. The bitstream is also contained in the BOOT.BIN file and is loaded into the PL after the initialization has finished. After the FSBL has finished it executes the second stage, the U-Boot boot loader which loads the kernel image into the DDR memory.

Linux Kernel

The TRD uses a Linux kernel that is based on the mainline Linux kernel but maintained by Xilinx. This kernel incorporates many patches not found in the mainline kernel. These patches provide support for Xilinx specific features. Supported devices include the Xylon logiCVC-ML which gets abstracted to a generic frame buffer drive, Xilinx VDMA controllers which controls the transfers to and from the DDR memory, PS-GPIO giving access to GPIO pins to the operating system, and the ADV7511 HDMI transmitter gets a Video4Linux v2 Driver. Some IP cores have limited functionality and don't warrant a complete Kernel Driver being written to support the driver. In this case the developer can use the *Userspace IO* framework present in the Linux kernel.
The userspace IO (or UIO) is a framework present in the Linux kernel that allows

a developer to write a minimal kernel driver for a piece of hardware and perform most of the necessary functions from userspace. This system reduces the complexity of driver development and reduces the risk for bugs in a kernel module. UIO is a good fit for a device if it has one or more of following properties:

- The device has memory that can be mapped onto virtual memory and can be completely controlled using this memory.
- The device generates interrupts.
- The device doesn't fit in one of the standard kernel subsystems.

These UIO device drivers are made available to the user through the device node and the Sysfs, which is a virtual file system that exports information on devices and their drivers. The device file typically has the form /dev/UIOX with X being a number starting on 0. This file represents the memory of the system and can be opened using the MMAP() system call. Interrupts are handled by performing a blocking read on the device file. When the read returns an interrupt has happened. The return value is an integer which contains the number of interrupts that have occurred. By comparing this value to the previous value the user can check whether any interrupts were missed. Vivado HLS generates the UIO driver, leaving only the development of a kernel module to be done by the developer.

Application

There are 2 versions of the application available: a version with a Qt based GUI and a commandline based application. The application lets the user to select one of two video sources, the TPG or the HDMI input, and apply 2 implementations of the Sobel filter on these inputs. One implementation is done in software and runs on ARM processor, the other one is a vivado HLS generated core. The Qt application also has a readout of the system throughput.

2.3.3 Sobel Operator

The described system performsedge detection using the sobel operator on the video data. The sobel operator computes an approximation of the gradient. The computation is done by convolving the image with 2 3×3 kernels.

$$G_x = \begin{bmatrix} +1 & 0 & -1 \\ +2 & 0 & -2 \\ +1 & 0 & -1 \end{bmatrix} \qquad G_y = \begin{bmatrix} +1 & +2 & +1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{bmatrix} \tag{2.1}$$

The combination of these operators give an approximation of the gradient in both the x and the y direction. The sum of the absolute values of these two approximations gives the edge weight. It shows whether a location is close to an abrupt change in intensity in an image, an edge. By thresholding the edges can be separated from the rest of the image. The Sobel operator falls under dwarf number 6: structured grid which puts it in the cross section of the CPU, GPGPU and FPGA categories. The sobel oparator is thus a good fit for the Zynq platform.

Chapter 3

High Level Synthesis

A recurring theme in the literature is the relative difficulty of implementing an algorithm on an FPGA compared to conventional implementation techniques on CPU's and GPU's. Both development time and place-and-route take considerably more time compared to programming and compiling for more traditional architectures [13, 18]. With an increase in the complexity required to perform a task also comes an increase in the difficulty of designing and debugging such a system. High level synthesis tools enable a user to specify the behavior of a system in a high level programming language and convert this description into usable hardware. Most tools use the C, C++ or sytstemC programming languages, however other programming languages such as the functional programming language Haskell [6], the scripting language Python[12] or Matlab M-code[1] have been used. These tools enable faster prototyping and implementation[9]. These tools also enable programmers without a background in HDL design to benefit from the advantages of FPGA accelerators without facing the steep learning curve of learning a HDL such as VHDL or Verilog. For HDL designers these tools increase productivity by allowing the designer to describe the desired behavior using less code.[8]. These tools enable to shift the focus from low-level implementation details to the development and improvement of the algorithm in a rapid prototyping fashion[19]. HLS tools have a long history dating back to the 1970's, but only recently have these tools matured enough to become adopted by industry. These tools present an interesting evolution and a possible paradigm shift in hardware design and prototyping[10].

3.1 Riverside Optimizing Compiler for Configurable Computing

ROCCC is a C-to-VHDL compiler which focuses on FPGA based code acceleration. It implements a subset of the C language on which it performs loop analysis techniques to provide increasing throughput with less usage of area[15]. The generated VHDL is independent from FPGA platforms and supports code reuse through the use of modules. ROCCC uses the streaming paradigm, in which data is represented by streams, a data format similar to the way arrays are stored in memory. These streams pass through a set of operations called kernels. This particular way of representing data makes it possible to express parallelism and is relatively easy mapped to the FPGA hardware. This paradigm removes the need for area-costly soft-core processors[7].
This streaming paradigm is also what enables the platform independence of the ROCCC hardware. As long as the data is delivered to the system in the form of a stream it can be used. Another important feature of ROCCC are the so-called smart buffers. These attempt to utilize the data-locality of certain applications to increase the performance. This is achieved through intelligent data reuse which minimizes the number of off-chip memory accesses.

3.2 Xilinx Vivado High Level Synthesis Tool

Vivado High-Level Synthesis is part of the Xilinx Vivado design suite. It is the product of the acquisition of AutoESL and the re-branding of their AutoPilot High-Level Synthesis tool. Vivado represents the next evolution of Xilinx tools fitting in their vision of an "all programmable world".
Vivado HLS translates a subset of C, C++ or SystemC into VHDL or Verilog code. This is done through 2 types of synthesis: *functional synthesis* and *interface synthesis*. The functional synthesis synthesizes the functional statements into RTL statements spread over different clockcycles. The interface synthesis synthesizes the function parameters into ports with specific timing protocols. To do this the HLS tool extracts the control behavior and the datapath. The control is specified in the high level language by loops and conditional statements in the code. These get translated into state transitions in a finite state machine. The datapath is determined by unrolling all the loops and evaluating all the conditional statements. After the control behavior and datapath have been extracted, the actions that need to be taken are identified and scheduled to occur during in a state. This process takes into account the timing information and optimization directives to ensure optimal performance. After all the necessary actions are scheduled, the operations are bound to a specific hardware resource or core. The binding process influences

the scheduling process so binding decisions are considered during scheduling. The directives allow the programmer to optimize for different parameters such as latency, area or initiation interval. The area is the number of resources a specific implementation uses. Latency is the number of clockcycles necessary to produce an output, and the initiation interval is the number of clockcycles that need to pass before a new input can be accepted. there are also directives that allow to constrain the latency of functions or loops and the area used, as well as specify of override dependencies in the code.
Vivado HLS supports a large subset of the C, C++ and SytemC languages, some constructs are not supported. Most notable are calls to the operating system, dynamic memory allocation and the C++ Standard Template Library. Vivado HLS has a number of features that can improve the productivity of a developer:

Verification Verifying the functionality of an algorithm and the correct functioning of its implementation in hardware are important steps in the development process. Manually verifying the implementation through RTL simulation is a tedious and error-prone process. For this reason it is interesting to be able to automate these tests. Vivado HLS separates the verification into two discrete processes.
The first is the pre-synthesis validation, which does a functional validation of the high level code. The developer needs to develop a testbench, which provides an input for the function being verified, and compares the output of this function to a so called *gold standard* which is the expected output.
The second verification process is the post synthesis verification. Vivado HLS provides an RTL implementation in VHDL of Verilog for which a testbench can be written. Vivado HLS also has the `cosim_design` function which converts the high level testbench into a SystemC testbench. This low level testbench provides stimulus to the RTL code and compare the output to the *gold standard*. This automation is possible when some requirements are satisfied:

- The top-level system needs to be synthesized with a supported control interface and the output ports need to have a valid signal.
- Certain optimizations or combination of optimization on arrays in the function interface, either as an explicit array or hidden in a struct, are not supported.
- The testbench needs to be self-checking, return 0 if the output is correct and a non-zero value when the output is incorrect.

Vivado HLS incorporates a simulation tool but also supports third-party simulators.

Data type conversions Whilst C-based data types have widths limited by byte boundaries, RTL buses don't have this constraint and can have an arbitrary width. Vivado HLS provides libraries for the supported programming languages that allow to specify the desired width between 1 and 1024 for integers in C and C++. The `__SYNTHESIS__` macro allows to retain the original declarations for backwards compatibility and reference purposes.
Vivado HLS also has libraries for fixed-point types. Fixed point operations are often preferred by HDL designers because of the reduced resource consumption. These libraries also ensure that the high-level and RTL behavior remain the same. This allows the designer to use the high-level simulations to study the effects of the reduced precision compared to floating-point representations. These fixed point data types are available in C++ and SystemC.
Floating point arithmetic is supported, however not all operations are supported.

Interface Synthesis Vivado HLS supports a number of different interfaces that can be used to convert the function arguments into RTL ports. These standard interfaces can range from control signals to elaborate handshaking protocols and memory interfaces. Vivado HLS also supports a number of bus interfaces. These differ form the standard interfaces in that they only get added to the system during the export to RTL. The supported bus interfaces are:

- AXI4-Lite Slave
- AXI4 Master
- AXI4 Stream

These bus interfaces sit "on top of" the RTL interfaces, and only certain RTL interfaces can be connected to certain bus interfaces.

Optimizations Vivado HLS supports a large number of optimizations on the function, loop and array level.
The available function optimizations are

- **Inline** The *inline* directive places the function in-line, removing all hierarchy. This removes the cycles needed to enter and leave the function.
- **Instantiate** By default functions remain separate hierarchical blocks in the RTL and all the instantiations use the same RTL implementation. The *instantiate* directive lets the compiler create a separate, unique and optimized implementation for each instance of the function.

- **Dataflow** The *dataflow* directive will have the compiler attempt to execute each RTL function block at the same time. If data dependencies prevent this the interval will be modified until the dependencies are satisfied. This effectively pipelines the communication between the functions. The channels can be configured to be ping-pong or FIFO buffers.

- **Pipeline** Whereas the *dataflow* directive pipelines the communication between different functions, the *pipeline* directive pipelines the operations within a function. This allows these operations to happen concurrently. As with dataflow pipelining data dependencies can prevent pipelining. The solution is to increase the initiation interval until the dependencies are satisfied.

- **Latency** The latency of a function can be specified using the *latency* directive. The synthesis process tries to stay within the constraints set by the directive and allows timing violations in doing so. If the system cannot satisfy the constraints they are automatically relaxed. After synthesis the constraint report details whether all constraints have been satisfied.

On the loop level a number of optimizations are available. The *dataflow*, *pipeline* and *latency* optimizations are also available for loops and have functionality comparable to the aforementioned optimizations for functions. The other ones are:

- **Unrolling** The default behavior for Vivado HLS is that all loops remain rolled. This means that every iteration of the loop uses the same hardware. The *unroll* directive allows the designer to partially or completely split the iterations of a loop into separate entities. Multiple iterations can then be executed concurrently. Unrolling a loop presents a trade-off between resource consumption and performance.

- **Merging** A loop takes one cycle to enter and one cycle to exit from. If there are consecutive loops in the design these states are unnecessary. The *merge* optimization combines these consecutive loops into one loop, no longer requiring multiple cycles to enter and exit a loop.

- **Flattening** This optimization is similar to the *merging* optimization, but applied to nested loops instead of consecutive loops. The nested loops are combined into one loop removing the need for extra clockcycles to enter and exit the innermost loop. When loops are nested the outermost loop cannot be pipelined. *Flattening* prevents this.

- **Dependence** The compiler tries to identify dependencies between calculations or resources. Sometimes this automatic identification of dependencies

is too conservative because the compiler does not have the necessary information. For this reason the *dependence* directive exists, allowing the programmer to explicitly state that there are or aren't dependencies for a certain variable. There are two types of dependence:

Inter The dependence is between different iterations of the same loop. If the dependence is set to 'false', the compiler will not prevent the loop from being unrolled.

Intra The dependence is inside the iteration. If the dependence is set to be 'false' the compiler will attempt to reorder the operations for the most optimal performance.

- **Tripcount** HLS tries to determine the maximum possible number of iterations that a loop can perform during the execution. The compiler cannot determine the actual maximum iteration of a loop. The *tripcount* optimization provides this data to the compiler, so as to ensures that the synthesis report will contain valid figures for throughput and latency. This optimization does not influence the synthesis process and only exists to provide metadata to the compiler.

Finally there are also optimizations that determine the memory architecture of the design. The way memory accesses are implemented are an important factor influencing the performance of an IP-Core. Using buffers is a way to increase the computational intensity, thereby decreasing the load on the memory In *High Level* programming languages memory gets abstracted to variables and array's. These abstractions need to be translated into something that can be implemented in hardware. For FPGA's this means choosing between *block ram* or registers. The process of translating the memory constructs into the most fitting type of physical memory is controlled by the HLS compiler. This process can be influenced by using directives or pragma's. Especially the manner in which arrays are translated into hardware is of importance. For this purpose a couple of directives are available.

- **Resource** The default behavior of Vivado HLS is to select the memory to be used. The *resource* optimization allows the designer to override this decision and specify which type of memory needs to be used to implement an array.

- **Array Mapping** The RAMs available in an FPGA have pre-defined sizes. Having to store many arrays each in their own RAM increases the resource consumption. The *array map* directive allows the designer to map multiple

arrays onto one RAM reducing the resource consumption. There are two types of mapping. Horizontal mapping concatenates the original arrays to create an implementation using a single array with more elements. Vertical mapping concatenates the words in the array to create an implementation that uses a single array with a larger bit-width.

- **Array_Partition** The *array partition* optimization performs the opposite of the *array mapping* optimization. It splits a large array into a number of smaller arrays which can each be mapped onto a different RAM. This allows for multiple consecutive reads or writes to memory. There are three possible ways to partition. Block partitions separate the array into a number equally sized blocks containing consecutive elements of the original array. Cyclic partitions split the array into a number of equally sized blocks with the elements of the original array interleaved. Complete partitioning separates the array and stores the individual elements in registers. The number of blocks in which the arrays are split can be specified by a factor.

- **Array_Reshape** The *array reshape* optimization combines array partitioning with vertical mapping. It takes elements from one dimension in the original array and maps them to a single element with a larger bitwidth in the reshaped array.

Pragmas and directives The aforementioned optimizations can be applied to a design using 2 methods. The first one is by embedding pragmas in the source code. In C-style languages pragmas are typically specified as preprocessor directives. These directives provide the compiler with meta-information concerning the way the code should be converted to machine code. Vivado HLS has its specific pragmas who serve the same function. The second method is by using the directives file, a TCL script that specifies the optimizations that need to be applied. The optimizations that can be applied are the same for both methods, which makes the two possibilities seem redundant. There are however subtle differences in the usage.
Vivado HLS has a feature called a *solution*. These solutions allow a designer to quickly do design space exploration. This is achieved by applying many different combinations of directives in each solution. The designer can then measure the impact and compare the performance of different solutions. This can only be achieved by using the directives file as the code is shared between solutions. Pragmas are useful when the directives need to remain the same irrespective of the solution. This is for example the case when using libraries.

Synthesis report & Analysis perspective Vivado HLS generates a report after the synthesis has finished. This *synthesis report* contains, next to some general information, performance estimates for the timing, latency an resource consumption. The timing information comprises an estimate of the smallest attainable clock period, the uncertainty and whether this estimate satisfies the target clock period. The latency information comprises an estimate of the function's latency as well as the initiation interval for the block and all sub-blocks. the utilization estimates detail the expected utilization of resources present in the FPGA such as LUTs or DSP48s. A more detailed view is also given, showing the estimates of the resource consumption of different cores.
Another feature of Vivado HLS that can help in estimating the performance of an implementation is the *analysis perspective*. The analysis perspective provides the designer with a graphical representation of the design's performance and resource usage, as well as the scheduling of the different operations. An example of the schedule viewer pane is given in figure 3.1.

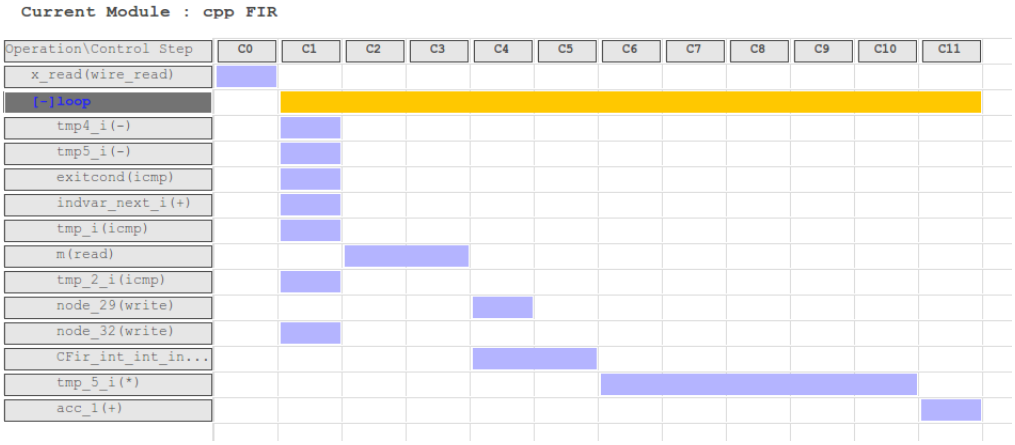

Figure 3.1: Example of the Schedule Viewer Pane

The left column details the resources. Loops are marked in yellow and operations are marked in purple. The numbered columns each represent a state in the FSM. This allows the designer to see where resources are being used concurrently and where there are dependency issues preventing this. The schedule viewer also has a "Goto Source" command, which jumps form the selected operation to the corresponding line in the high level source code. Using this functionality the designer can link the high level code to the execution of the hardware.

Exporting Vivado HLS is part of Xilinx' Vivado toolchain and is backwards compatible with its ISE toolchain. This tight integration is most visible in the

available export options in Vivado HLS. Dependent on the target technology and available license it can export to formats supported by the Vivado IP-catalog, Xilinx System Generator and EDK PCore. If the core is configured to use the AXI protocol the wrappers will be added in this step. The AXI wrappers are not part of the VHDL or Verilog implementation.

Vivado HLS version 2013.2 was used for this thesis.

Chapter 4

Performance Analysis

4.1 Roofline Model

The roofline model as proposed provides a visual model to gain insight into the factors influencing the performance of multicore CPU's.[20] It is based on the observation that the off-chip memory bandwidth is the constraining resource on the performance of a system.[16] The roofline model is a plot of the attainable floating point operations per second in function of the computational intensity. The computational intensity (CI) is the number of floating-point operations per byte fetched from the off-chip RAM memory. The roofline model thus relates the demands of an application on the memory system to the maximum attainable performance. An example plot of the roofline model is given in figure 4.1.

There are two main factors influencing the upper bound on the performance. The first one is the peak memory bandwidth (BW). The peak memory bandwidth is represented by the sloped black line on the left side of the graph in figure 4.1. An application that hits the roof in this area is called memory bound. An example of an computational intensity resulting in memory-bound operation is given by the red line in 4.1 With increasing computational intensity the performance increases up to the ridge point. The x-coordinate of the ridge point represents the minimal computational intensity an application needs to reach to get maximum performance from the architecture. This maximum performance is the maximum computational performance (CP) and is represented by the flat line on the right side of figure 4.1. The dashed blue line shows an application of which the performance hits the roof in the computational performance area. The relation between CI, BW and CP are given by the following formula.

$$Attainable\ GFlops/sec = min\,(BW \times CI\ ,\ CP) \tag{4.1}$$

The roofline model provides an upper bound to the performance of a certain architecture, but an application is not guaranteed to perform at this upper bound. Only if sufficient use is made of the available resources and optimizations, can the performance reach the roof. The effect on the maximum attainable performance on the performance of an architecture can be represented by what is called a *ceiling*. In figure 4.1 there are 2 examples of a ceiling: an I/O bandwidth ceiling represented by the dashed cyan line and a computational ceiling represented by the dashed green line.

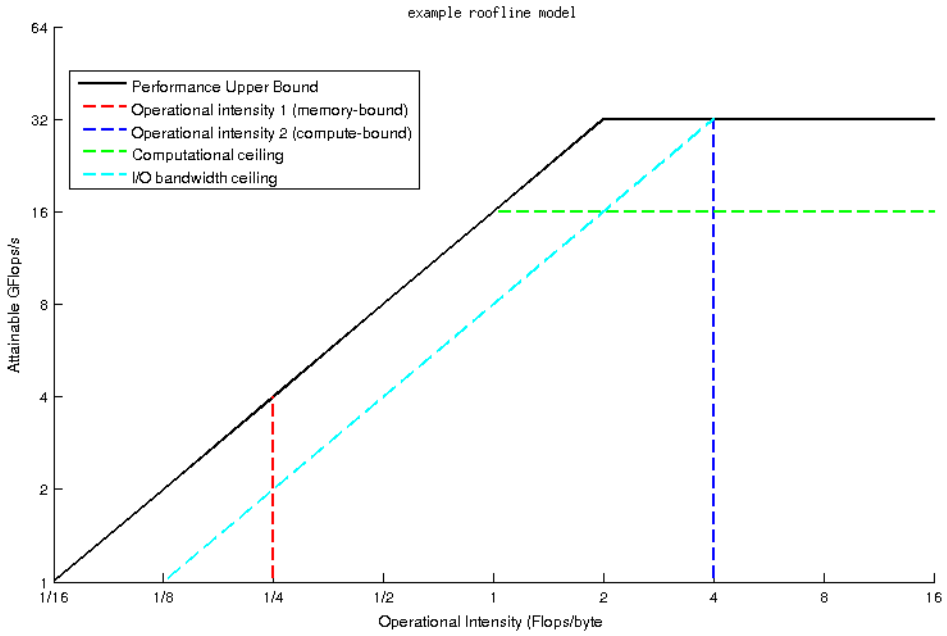

Figure 4.1: Example of a roofline model

Although the roofline model has been conceived as a tool for the estimation of performance of multicore CPU architectures, it has been adapted for use with other architectures such as GPU's[17] and FPGA's[17, 11]. The roofline model in it's default form is inadequate to describe the maximum attainable performance of an FPGA based system because of the flexibility of FPGA's. In [11] a number of extensions are proposed to make the roofline model more suitable for FPGA's:

Operations Because floating-point operations are prohibitively expensive area-wise, they are often replaced by alternatives such as fixed-point operations. For this reason byte-operations [Bops] are proposed as a more general alternative to the more CPU/GPU specific floating-point operations.

Scalability A processing element (PE) is defined as the hardware that contains all the necessary resources to perform the functionality of the algorithm. If the FPGA has enough resources this allows for multiple instantiations of the PE. The scalability factor is given by following formula:

$$SC = \left[\frac{Available\ Resources}{Resource\ consumption\ per\ PE} \right] \tag{4.2}$$

The attainable performance of one PE needs to be multiplied by the scalability factor.

$$Attainable\ Performance = min(CP_{PE} \times SC, CI \times BW) \tag{4.3}$$

This relates computational performance to resource consumption.

I/O bandwidth In the original roofline model the off-chip memory is considered as the only bound on the performance of a system. Because of the nature of FPGA's where multiple means of I/O are available at the same time these all have to be considered. The roofline model should all take these into consideration and thus have an I/O bandwidth roof and possibly multiple I/O bandwidth ceilings.

Roofs and ceilings Due to the fact that the computational intensity and the computational performance are both dependent on the implementation of the algorithm, the computational performance roofline will no longer be a constant. Also a number of ceilings have to be included to represent the different optimization and I/O possibilities.

Computational intensity Because the CI influences both the SC and the CP_{PE}, it is the key to determining the roofline model. In the original roofline model the CI is modified through the use of code adjustments or optimizations. In the roofline model for FPGA the preferred methods of modifying the CI is through optimizations or through increasing the data locality of the implementation. An example of an optimization that influences the CI is loop unrolling.

4.2 Factors influencing performance in a Vivado HLS core

Video processing systems usually employ buffers to benefit from spatial and temporal locality of reference. In the example of the TRD there are 2 abstractions

implemented: `ap_linebuffer` and `ap_window`.

4.2.1 `ap_linebuffer` Class

The class `ap_linebuffer` is a generic C++ implementation of the linebuffer described in XAPP793. A linebuffer is described as a multi-dimensional shift-register. A linebuffer needs to be able to be read and written to in the same cycle to maximize performance. The dual port nature of block RAM makes it the ideal component for this abstraction. Because the `ap_linebuffer` class is generic its behavior needs to be defined in the application. The template for the `ap_linebuffer` class is `<typename T, int LROW, int LCOL>`. A type, the number of rows and the number of columns need to be specified. These definitions are in the `sobel.h` file:

```
typedef ap_linebuffer<unsigned char, 3, MAX_WIDTH> Y_BUFFER;
```

The parameters of the template are used to determine the size of the only variable of the class, -the array M of type T with LROW rows and LCOL columns.
This array gets partitioned by the following directive:

```
#pragma AP ARRAY_PARTITION variable=M dim=1 complete
```

This means that the first dimension, the number of rows, get partitioned into different block RAMs. This is also reported by the Vivado HLS tool. buff_A is the line buffer used throughout the implementation.

Memory

Memory	Module	BRAM_18K	Words	Bits	Banks	W*Bits*Banks
buff_A_M_0_U	sobel_filter_buff_A_M_0	1	1920	8	1	15360
buff_A_M_1_U	sobel_filter_buff_A_M_0	1	1920	8	1	15360
buff_A_M_2_U	sobel_filter_buff_A_M_2	1	1920	8	1	15360
Total		3	3	5760	24	3	46080

Figure 4.2: Partitioning of an array in multiple block RAM instances

4.2.2 `ap_window` Class

The second class used in the application is a generic implementation of the memory window described in XAPP793. It is a combination of shift-registers forming a 2-dimensional data storage element of N pixels centered on a pixel P. Usually these are implemented as flip-flops because they contain relatively few elements who need to be simultaneously available for a calculation. This is achieved

through completely partitioning the memory into registers, preventing it from being implemented by a block RAM. The template of `ap_window` is `<typename T, int LROW, int LCOL>`. These parameters are used for the only variable in the class, an array M of type T with LROW rows and LCOL cols. Analog to the linebuffer class the programmer here also needs to define the type, number of rows and number of columns of array M. The array gets partitioned into registers by the following directive

```
#pragma AP ARRAY_PARTITION variable=M dim=0 complete
```

The `dim=0` means that all dimensions should be partitioned. The `complete` keyword signifies that this partitioning should be done for the whole array.

4.2.3 Influence of memory architecture on the computational intensity

This hierarchical structure of the memory influences the computational intensity of the algorithm. The numerator is determined by the number of bytes being processed by the core. Every iteration one value gets read from the external memory and one value gets written. There are 32 bits per pixel, so there are 4 bytes per pixel. This means that with height H and width W there are:

$$4 * 2 * (H * W) \tag{4.4}$$

bytes being read or written to or from the memory. This is the denominator in the expression of the computational intensity.
The numerator is dependent on the number of pixels being calculated by the core. The implementation of the Sobel core doesn't calculate the values on the pixels of the outer rim, as can be seen in code listing 4.1

```
if( row <= 1 || col <= 1 || row > (rows-1) || col > (cols-1)){

                edge.R = edge.G = edge.B = 0;

}
//Sobel operation on the inner portion of the image
else{

                edge = sobel_operator( ... );

}
```

Listing 4.1: Sobel Code Snippet

This however doesn't influence the computational intensity if one interprets a processing element as being all the necessary resources to perform the functionality

of the algorithm. Even more so, these pixels also have to be read from memory and are already present in the denominator part of the expression.

$$(H * W) \tag{4.5}$$

Combining Equations 4.4 and 4.5 gives us

$$CI = \frac{H \times W}{4 \times 2 \times (H \times W)} \tag{4.6}$$

4.3 Directives influencing throughput

4.3.1 Original Directives

The original TRD has 3 directives applied to it:

- `set_directive_loop_flatten -off "sobel_filter/sobel_filter_label0"`
- `set_directive_dependence -variable &buff_A -type inter -dependent false "sobel_filter/sobel_filter_label0"`
- `set_directive_pipeline -II 1 "sobel_filter/sobel_filter_label0"`

Throughput Given the analysis generated by Vivado HLS, presented in table 4.2, it is possible to calculate the throughput of the system. First of all it needs to be noted whether the system satisfies the timing requirements. The analysis gives us an estimated clock period of 4.2 ns with an uncertainty of 0.62 ns placing it well within the bounds of the required 5 ns clock period. The system needs 22 cycles to complete. Of these, there are 2 initialization cycles, 20 cycles to finish the outer loop, of which 19 cycles are the inner loop. The system employs pipelining on the innermost loop, which results in an initiation interval of 1.
I denotes a number of iterations, n a number of cycles. N is the number of cycles necessary to calculate one frame:

$$N_{frame} = n_{init} + I_{outer\ loop} \times (n_{inner\ loop} + I_{inner\ loop} - 1) \tag{4.7}$$

These cycles all take one clock period to complete:

$$t_{frame} = N_{frame} * T_{clock} \tag{4.8}$$

The number of frames per second is then given by:

$$FPS = \frac{1}{t_{frame}} \tag{4.9}$$

Entering the numbers found in table 4.2 gives us a value of 95.37 frames per second. Given that HDMI has a 60 Hz refresh rate this system satisfies that constraint.

4.3.2 No directives

The original system performance satisfies the real time constraint placed on the system. To study the impact the directives have on the performance of the system, all directives were removed from the system. The analysis generated by Vivado HLS is presented in the second column of table 4.2. The first observation that can be made is that the system performs the same operation in only 16 clock cycles instead of 22, a decrease of 27%. Because the system has no pipelining, the initiation interval is 14 cycles. A new value gets read each iteration.
The number of cycles needed to calculate one frame is given by:

$$N = n_{init} + I_{outer\ loop} \times (I_{inner\ loop} \times n_{inner\ loop}) \tag{4.10}$$

Knowing the number of cycles the throughput can be calculated using formulas 4.8 and 4.9 and the information found in table 4.2 . This gives us a value of 0.47 frames per second, a 203 times decrease in performance.

4.3.3 Other combinations of directives

It is clear that these directives have a profound effect on the performance of a system. Also noteworthy is that the directives presented in section 4.3.1 in no way influence the computational intensity of the implementation. Each combination of these directives can thus be represented as a ceiling in the roofline model. All sensible combinations of these directives have been tested and the results are represented in table 4.2. The corresponding resource utilization estimates are presented in table 4.1.

Only Dependence Using only the following directive

```
set_directive_dependence -variable &buff_A -type inter
-dependent false "sobel_filter/sobel_filter_label0"
```

doesn't have any measurable effect on the performance on the system compared to the one presented in section 4.3.2. The calculations for the performance stay the same at 0.47 FPS.

Only Loop Flattening Using only the following directive

```
set_directive_loop_flatten "sobel_filter/sobel_filter_label0"
```

combines the 2 loops iterating over the rows and the columns into 1 loop. Because this means that there is only one iteration to take into account, this changes the throughput of the system. The number of cycles necessary to process a frame is given by:

$$N = n_{init} + (I_{outer\ loop} \times I_{inner\ loop}) \times n_{loop} \tag{4.11}$$

Using formulas 4.8 and 4.9 and the values found in 4.2, a performance of 6,88 FPS can be calculated.

Only pipelining Using the following directive

```
set_directive_pipeline -II 1 "sobel_filter/sobel_filter_label0"
```

Flattens the loops and pipelines them. This has a profound effect on the system performance. II is the initiation interval.

$$N = n_{init} + n_{outer\ loop} + ((I_{inner\ loop} \times I_{outer\ loop}) - 1) \times II \tag{4.12}$$

Using the values found in table 4.2 gives a performance of 48.15 FPS. This is a considerable improvement but doesn't reach the required 60 FPS to satisfy the requirements placed on the system by the HDMI protocol. Noteworthy in this case is that the system has an initiation interval of 2. This causes a pixel to be output only every 2 cycles instead of every cycle after the first iteration. The compiler defaults to the lowest II it can use. As an experiment the initiation interval was set to 2 in the directive. This didn't have any measurable influence on the system.

Loop flattening on Using following directives:

- `set_directive_dependence -variable &buff_A -type inter -dependent false "sobel_filter/sobel_filter_label0"`
- `set_directive_pipeline -II 1 "sobel_filter/sobel_filter_label0"`
- `set_directive_loop_flatten "sobel_filter/sobel_filter_label0"`

This flattens the loops, pipelines them and takes into consideration that there is no inter-iteration dependence conflict for the buff_A. The expression for the number of cycles necessary is the same as in equation 4.12. The most notable difference is that the initiation interval is equal to 1. This gives a performance of 91.72 FPS. Vivado HLS predicts in it's synthesis report that the system will have a minimum clock period 5.25 ns. This leaves the risk that the core will cause glitches and not satisfy the requirements. Increasing the II to 2 makes the system respect the timing constraints again but lowers the performance to 48.15 FPS.

No dependence Using following directives:

- `set_directive_loop_flatten -false "sobel_filter/sobel_filter_label0"`
- `set_directive_pipeline -II 1 "sobel_filter/sobel_filter_label0"`

The only difference with the directives of the original TRD is that the compiler isn't notified of the fact that there is no inter-dependence for the `buff_A` variable. The same equation for the performance is given by eq. 4.7. Using the values found in table 4.2 a performance of 95.31 FPS cam be calculated, only marginally worse than the performance of the original configuration.

4.3.4 Resource consumption

	BRAM_18K	DSP48E	FF	LUT
Original directives	3	19	1487	1412
No pragma	5	4	802	1475
loop_flatten_on	3	23	1764	1668
loop_flatten_II_2	3	23	1777	1875
no_dependence	3	19	1487	1412
no_pipeline	5	4	802	1475
only dependence	5	4	786	1534
only loop flattening	5	8	1050	1783
only pipelining	3	23	1851	1849

Table 4.1: Utilization Estimates

	Original directives	No pragma	loop_flatten_on	loop_flatten_II_2	no_dependence
Estimated Clock (ns)	4,2	4,35	5,25	4,2	4,2
Uncertainty (ns)	0,62	0,62	0,62	0,62	0,62
cycle time (ns)	5	5	5,25	5	5
total cycles	22	16	28	29	23
init	2	2	8	8	2
outer loop	20	14	20	21	21
inner loop	19	13	N/A	N/A	20
pipelining outer	no	no	yes	yes	no
pipelining inner	yes	no	N/A	N/A	yes
Initiation Interval	1	14	1	2	1

	no_pipeline	only dependence	only loop flattening	only pipelining	only pipelining II 2
Estimated Clock (ns)	4,35	4,35	4,35	4,2	4,2
Uncertainty (ns)	0,62	0,62	0,62	0,62	0,62
cycle time (ns)	5	5	5	5	5
total cycles	17	16	22	31	31
init	2	2	8	8	8
outer loop	15	14	14	23	23
inner loop	14	13	N/A	N/A	N/A
pipelining outer	no	no	no	yes	yes
pipelining inner	no	no	no	no	no
Initiation Interval	15	14	14	2	2

Table 4.2: Analysis Data

Bibliography

[1] Anon. Hdl coder.

[2] Anon. *ISE Design Suite 14: Release Notes, Installation, and Licensing*. Xilinx, 14.7 edition, October 2013.

[3] Anon. Zynq-7000 all programmable SoC technical reference manual, March 2013.

[4] K. Asanovic, R. Bodik, B. C. Catanzaro, J. J. Gebis, P. Husbands, K. Keutzer, D. A. Patterson, W. L. Plishker, J. Shalf, and S. W. Williams. The landscape of parallel computing research: A view from berkeley (2006). *Electrical Engineering and Computer Sciences University of California at Berkeley. http://www. eecs. berkeley. edu/Pubs/TechRpts/2006/EECS-2006-183. pdf*.

[5] Krste Asanovic, Rastislav Bodik, James Demmel, Tony Keaveny, Kurt Keutzer, John Kubiatowicz, Nelson Morgan, David Patterson, Koushik Sen, John Wawrzynek, David Wessel, and Katherine Yelick. A view of the parallel computing landscape. *Commun. ACM*, 52(10):5667, October 2009.

[6] Christiaan Baaij, Matthijs Kooijman, Jan Kuper, Arjan Boeijink, and Marco Gerards. Cλash: structural descriptions of synchronous hardware using haskell. In *Digital System Design: Architectures, Methods and Tools (DSD), 2010 13th Euromicro Conference on*, pages 714–721. IEEE, 2010.

[7] Betul Buyukkurt, Zhi Guo, and Walid A. Najjar. Impact of loop unrolling on area, throughput and clock frequency in ROCCC: c to VHDL compiler for FPGAs. In Koen Bertels, Joo M. P. Cardoso, and Stamatis Vassiliadis, editors, *Reconfigurable Computing: Architectures and Applications*, number 3985 in Lecture Notes in Computer Science, pages 401–412. Springer Berlin Heidelberg, January 2006.

[8] Emmanuel Casseau, Bertrand Le Gal, Pierre Bomel, Christophe Jego, Sylvain Huet, and Eric Martin. C- based rapid prototyping for digital signal

processing. In *Proceedings of the European Signal Processing Conference*, pages 1–4, Turquie, 2005. EUSIPCO.

[9] Shuai Che, Jie Li, J.W. Sheaffer, K. Skadron, and J. Lach. Accelerating compute-intensive applications with GPUs and FPGAs. pages 101 –107, June 2008.

[10] J. Cong, Bin Liu, S. Neuendorffer, J. Noguera, K. Vissers, and Zhiru Zhang. High-level synthesis for FPGAs: from prototyping to deployment. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 30(4):473 –491, April 2011.

[11] Bruno da Silva, An Braeken, Erik H DHollander, and Abdellah Touhafi. Performance modeling for fpgas: Extending the roofline model with high-level synthesis tools.

[12] Jan Decaluwe. Myhdl: a python-based hardware description language. *Linux journal*, 2004(127):5, 2004.

[13] Ra Inta, David J. Bowman, and Susan M. Scott. The Chimera: an off-the-shelf CPU/GPGPU/FPGA hybrid computing platform. *International Journal of Reconfigurable Computing*, 2012:1–10, 2012.

[14] David B. Kirk and Wen-mei W. Hwu. *Programming Massively Parallel Processors: A Hands-on Approach*. Elsevier, February 2010.

[15] G. Martin and G. Smith. High-level synthesis: Past, present, and future. *IEEE Design Test of Computers*, 26(4):18 –25, August 2009.

[16] David A Patterson. Latency lags bandwith. *Communications of the ACM*, 47(10):71–75, 2004.

[17] Martien Spierings, Rob van de Voort, Henk Corporaal, Cedric Nugteren, and Tom Goossens. Embedded platform selection based on the roofline model.

[18] Kuen Hung Tsoi and Wayne Luk. Axel: a heterogeneous cluster with FPGAs and GPUs. FPGA '10, page 115124, New York, NY, USA, 2010. ACM.

[19] Kazutoshi Wakabayashi. C-based behavioral synthesis and verification analysis on industrial design examples. In *Proceedings of the 2004 Asia and South Pacific Design Automation Conference*, ASP-DAC '04, page 344348, Piscataway, NJ, USA, 2004. IEEE Press.

[20] Samuel Williams, Andrew Waterman, and David Patterson. Roofline: An insightful visual performance model for multicore architectures. *Commun. ACM*, 52(4):65–76, April 2009.