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: Em. Prof. Dr. Ir. Erik D'Hollander
Prof. Dr. An Braeken

Advisors: In. Bruno Tiago Da Silva Gomes

January 2014

Abstract

acknowledgements

Contents

1 Introduction **7**
- 1.1 Computing Components . . . 7
  - 1.1.1 Multicore Processor . . . 7
  - 1.1.2 Graphics Processing Units . . . 8
  - 1.1.3 Field Programmable Gate Array . . . 8
- 1.2 Berkeley Dwarves . . . 8
- 1.3 GUDI Project . . . 10

2 Platform Overview **11**
- 2.1 Zynq-7000 . . . 11
  - 2.1.1 Processing System . . . 11
  - 2.1.2 Programmable Logic . . . 12
  - 2.1.3 Interconnect . . . 13
- 2.2 Toolchain . . . 16
- 2.3 Base Targeted Reference Design 14.5 . . . 18
  - 2.3.1 Hardware . . . 18
  - 2.3.2 Software . . . 19

3 High Level Synthesis **21**
- 3.1 Riverside Optimising Compiler for Configurable Computing . . . 21
- 3.2 Xilinx Vivado High Level Synthesis Tool . . . 22

4 Performance Analysis **23**
- 4.1 Roofline Model . . . 23
- 4.2 Factors influencing performance in a Vivado HLS core . . . 26
  - 4.2.1 Pragma's influencing the memory architecture implementation . . . 26
  - 4.2.2 `ap_linebuffer` Class . . . 26
  - 4.2.3 `ap_window` Class . . . 27
  - 4.2.4 influence of memory architecture on the computational intensity . . . 28

4.3 Pramga's influencing throughput 29
4.3.1 Orignal Pragma's 29
4.3.2 No pragma's 30
4.3.3 Other combinations of pragma's 31

List of Figures

1.1 Analysis of different hardware accelerators in regards to performance on a certain dwarf [11] 10

2.1 Zynq -7000 SoC overview [3] 13
2.2 Zynq Interconnect System Block Diagram [3] 15
2.3 Diagram of the Zynq toolchain 17

4.1 Example of a roofline model 24
4.2 paritioning of an array in multiple block RAM instances 27

List of Tables

4.1 Utilization Estimates 34
4.2 Analysis Data 35

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 which would make the research obsolete.To perpetuate the sequential programming paradigm several innovations such as multiple issue, deep pipelines and out of order execution were introduced into processors which were inefficient in both the use of transistors and power. Eventually though 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 go over 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 perfor-

mance 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 in in traditional computers meant to process images for output to the display. Recently however there has been increased interest in the GPGPU, the 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[12]. GPU's, due to their SIMD 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 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 aren't 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 sequential effectively 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 Dwarves

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 however 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 dwarves 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. More information can be found on the Berkeley View Wiki [1] and an updated view can be found in [5]. Finding out which dwarf is most suited for which platform is a very labour-intensive task. In [11] a theoretical analysis of dwarf performance on different accelerators in heterogeneous systems is given. A first point to note is that for floating point operations GPU's are hard to beat. 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 ˆ denotes floating point operations.

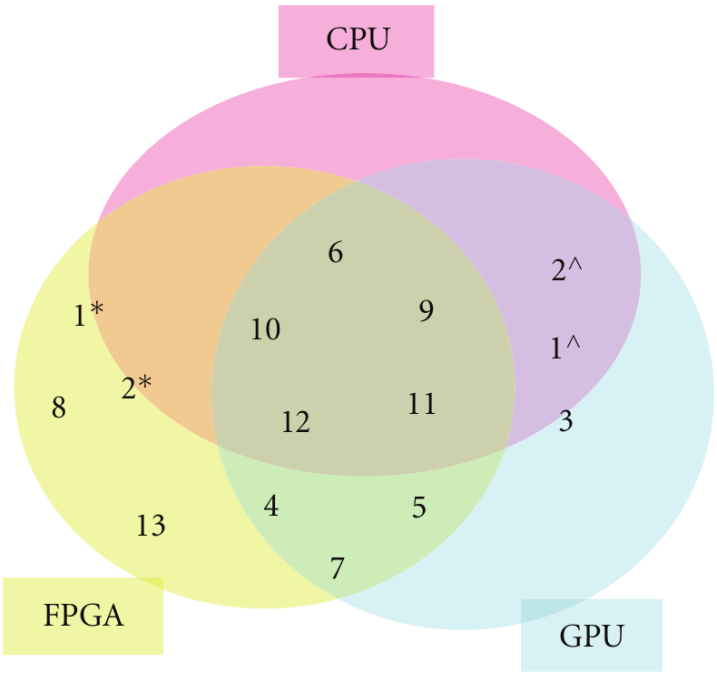

Figure 1.1: Analysis of different hardware accelerators in regards to performance on a certain dwarf [11]

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 has for goal to investigate the possibilities and limitations of a computer with such a heterogeneous architecture. It is an investigation into which technologies and which development tools perform best 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 which are 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. The two most notable differences between these generations is the shift from PowerPC based architectures to ARM based architectures, and a notable shift in emphasis from HDL centered design to a more programmer centric view with an emphasis on high level languages.

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 (PS) 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 from the programmable logic, which performs the role of master, to the processing system which serves as slave. This allows direct communication between the

PS and the PL through the L2 caches or on chip memory with guaranteed coherency. The 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. Finally 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.

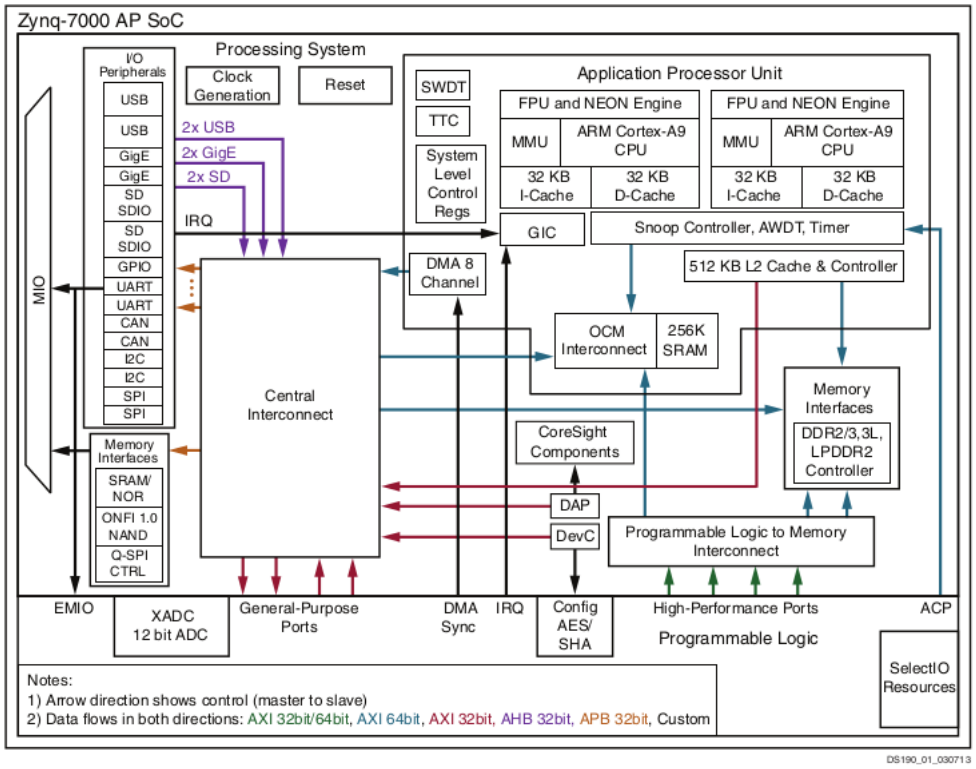

Figure 2.1: Zynq -7000 SoC overview [3]

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 done 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 comming from the central interconnect to AXI_GP, I/O peripherals, APB connections, etc.

Memory Interconnect The memory interconnect switches high speed traffic comming 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 the way these interconnects are organized can be found in figure 2.2

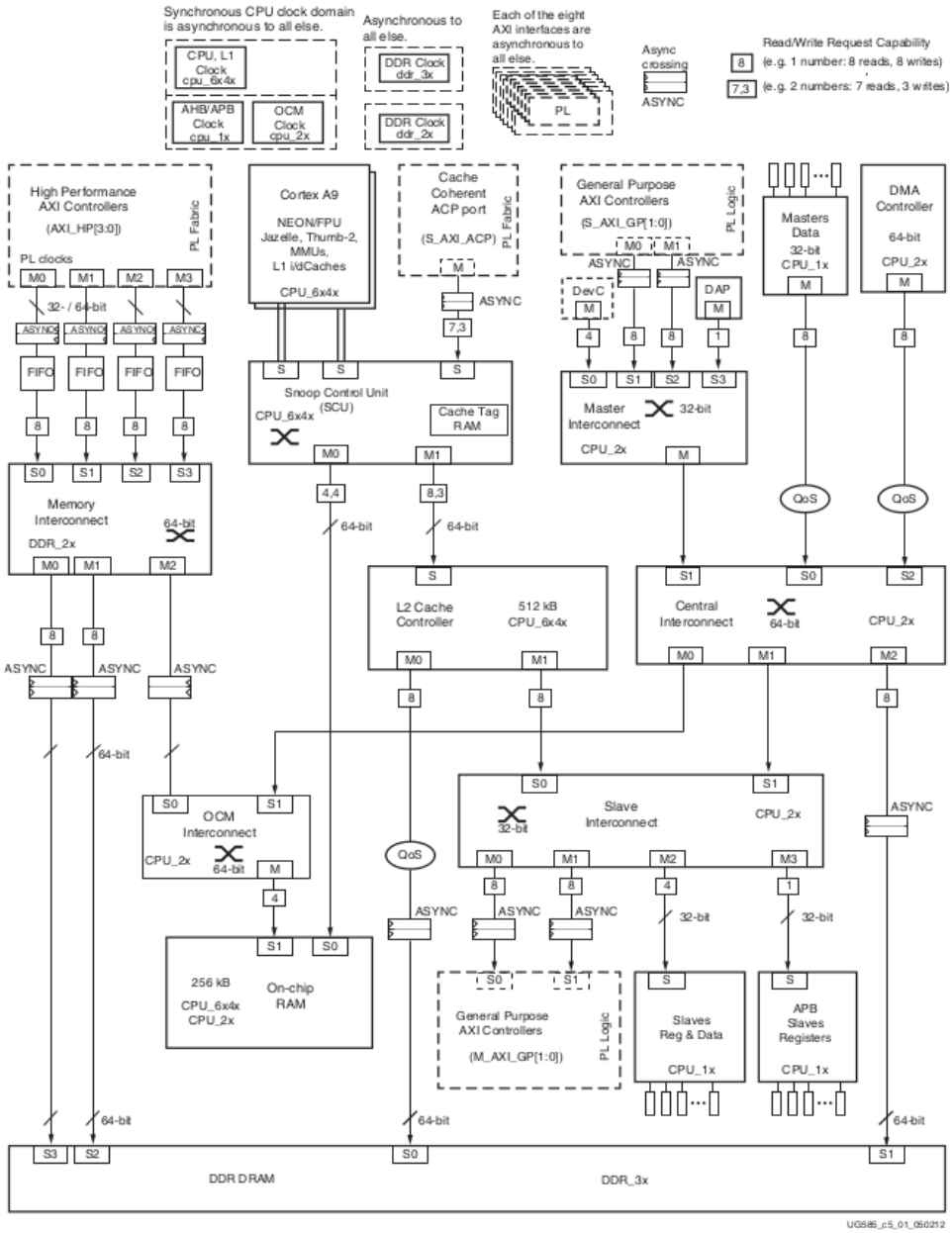

Figure 2.2: Zynq Interconnect System Block Diagram [3]

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. Connecting peripherals in the PL to the PS of the Zynq is done through 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, zhich also incorporates debugging and profiling tools. The SDK also has plugins which make it aware of the peripherals placed in the PL. The SDK also has a library of drivers for Xilinx 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 statisfies 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 necessary for the functioning of the system. 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. In SDK, develop the embedded software 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

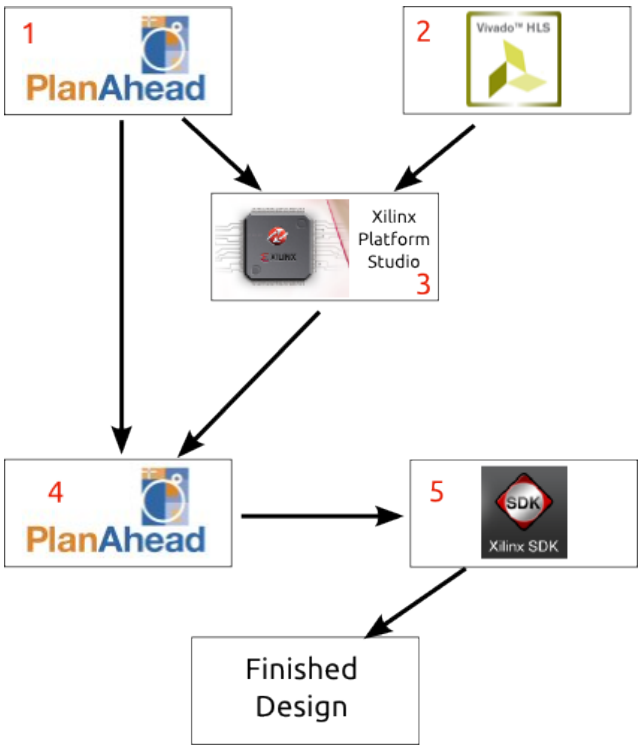

Figure 2.3: Diagram of the Zynq toolchain

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]. Xilinx

2.3 Base Targeted Reference Design 14.5

Zynq is a powerful but complex architecture requiring that a designer has different skills. Building a real-time video processing from scratch would involve connecting multiple IP-cores to each other and to the PL, as well as writing efficient applications that use the implemented hardware. Because this falls beyond the scope of this thesis an existing design was selected to do a performance analysis on. The Base Targeted reference Design version 14.5 was chosen because it utilized a Vivado HLS accelerator as well as some features of the Zynq architecture promising high performance and it was especially developed for the available hardware.

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. 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. This core 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 AXI form the 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 is the PS 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 to boot form. 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, which 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 initialisation is done. 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 that provide drivers for many Xilinx IP-cores. 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.

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. This read returns 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 between 2 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 throughput of the system.

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/compiling for more traditional architectures [11, 16]. High level synthesis tools enable a designer to implement an algorithm in a high level language and to have it compiled and synthesized into hardware. These tools enable faster prototyping and implementation[8]. The latest generation of these tools uses C or variants of this language to 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 existing HDL designers HLS tools these tools present a reduction in the number of lines of code that are needed to describe the design[7]. 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[17]. 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[9]. Two notable tools are the Riverside Optimizing Compiler for Configurable Computing and Vivado HLS.

3.1 Riverside Optimising Compiler for Configurable Computing

ROCCC is a C-to-vhdl compiler which focusses 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[13]. 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 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[6].
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 done by utilizing intelligent data reuse to minimize the number of off-chip memory acceses.

3.2 Xilinx Vivado High Level Synthesis Tool

Vivado High-Level Synthesis is part of the Xilinx Vivado design suite and 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". It allows C, C++ and SystemC code to be synthesized into VHDL or Verilog code. Functional simulation can be done in C, which is a great improvement over the typical VHDL or Verilog simulation. The Vivado tool is based on the Eclipse platform and incorporates the C Development tool (CDT).[15] Due to the recent release of Vivado there is not much academic information to be found about this tool.

Chapter 4

Performance Analysis

4.1 Roofline Model

The roofline model as proposed in [18] provides a visual model to gain insight into the factors influencing the performance of multicore CPU's. It is based on the observation that the off-chip memory bandwidth is the constraining resource on the performance of a system.[14] 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. 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 the performance can 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.

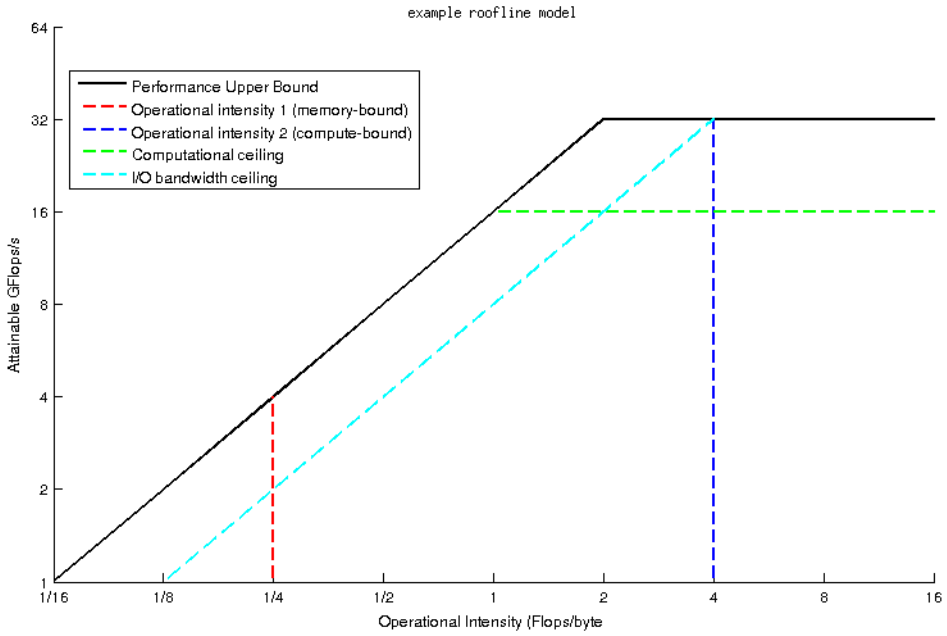

Figure 4.1: Example of a roofline model

Though 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[15] and FPGA's[15, 10]. 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 [10] 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 Consider a processing element (PE) 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 both are 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 detemining 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 preffered 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

4.2.1 Pragma's influencing the memory architecture implementation

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 but can be influenced by using directives or pragma's. Especially the way arrays are translated into hardware is of importance. For this purpose a couple of directives are available.

Resource lets the programmer determine which component will be used to map a certain array to.

Array_Map Maps several smaller arrays to the same memory to decrease the resource consumption.

Array_Partition Determines how a certain array will be partitioned into smaller arrays each using their own memory to avoid the bottleneck of having to perform multiple consecutive reads. This directive also allows to partition an array completely into registers.

Array_Reshape Will rearrange an array so the elements have a larger word width. This improves the performance of the memory while maintaining the same resource consumption.

In systems doing video processing buffers are usually employed to exploit the spatial and temporal data locality. In the example of the TRD there are 2 abstractions implemented: `ap_linebuffer` and `ap_window`.

4.2.2 `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. This is done in the `sobel.h` file by the following line:

```
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 get 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: paritioning of an array in multiple block RAM instances

4.2.3 `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 paritioned 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.

4.2.4 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/written to/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 Pramga's influencing throughput

4.3.1 Orignal Pragma's

The original TRD has 3 pragma's 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"`

These all influence the system in a distinct way.

Loop Flattening This directive combines nested loops. This removes the need for the clockcycle needed to enter and leave the loop. It needs to be applied to the inner loop of a set of nested loops. In the TRD loop flattening is explicitly disabled.

Dependence The compiler tries to identify dependencies between calculations or resources. Sometimes this automatic identification of dependencies is too conservative because the compiler doesn't have some 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 this will allow the loop to be 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.

In the case of the TRD the inter-dependence of the variable buff_A is set to false.

Pipeline The directive set_directive_pipeline is used to control the pipelining of loops and functions. Each function or loop on which this directive is used can read a new input every N clockcycles. This variable N is called the *Initiation Interval* or II for short. In the case of the TRD pipelining is applied to the inner loop, with an Initiation Interval equal to 0.

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 pragma's

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 done 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 pragma's

It is clear that these pragma's 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 pragma's can thus be represented as a ceiling in the roofline model. All sensible combinations of these pragma's 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 performance of the system. 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 statisfy 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

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

	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. Dwarf mine - view.

[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] 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.

[7] 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.

[8] 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.

[9] 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.

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

[11] 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.

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

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

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

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

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

[17] 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.

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