TAMPEREEN TEKNILLINEN YLIOPISTO

PANU SJÖVALL

HIGH-LEVEL SYNTHESIS OF HEVC INTRA PREDICTION ON FPGA

Master of Science Thesis

Examiners:
Dr Jarno Vanne and
Prof. Timo D. Hämäläinen
Examiners and topic approved by the
Council of the Faculty of Engineering
Sciences on 9th September 2015

TIIVISTELMÄ

Tampereen Teknillinen Yliopisto
Automaatiotekniikan koulutusohjelma
Panu Sjövall : HEVC-videokoodekin intra-ennustuksen toteutus FPGA-piireille C-kielestä syntesoimalla
Diplomityö, 49 sivua
Marraskuu 2015
Pääaine: Digitaalisten piirien suunnittelu
Tarkastajat: TkT Jarno Vanne, Prof. Timo D. Hämäläinen
Avainsanat: HEVC, Kvazaar, Intra Prediction, HLS, Catapult-C, C-Synthesis, FPGA

High Efficiency Video Coding (HEVC) on uusin standardi videon pakkauksessa ja purussa. HEVC:n avulla videota pystytään pakkaamaan puolella bittivirralla, verrattuna aiempaan standardiin, AVC:hen (Advanced Video Coding), saavuttaen silti saman laadun. Tämä kuitenkin lisää enkooderin laskentavaatimuksia.

Kun järjestelmien kompleksisuus kasvaa, nykyisillä laitteistonkuvauskielillä (Hardware Descrpition Language, HDL), kuten VHDL tai Verilog, ei enää pystytä kuvamaan järjestelmää vaivattomasti. Ratkaisu on käyttää korkeamman tason kuvauskieli. Korkeamman tason synteesissä (High-Level Syntehesis, HLS) laitteisto kuvataan käyttäen ohjelmointikieltä kuten C tai C++, ja HDL kuvaus luodaan automaattisesti. HLS:n avulla koodi on helpompi lukea ja ymmärtää, ja siksi toteutukseen käytetty aika pienenee.

Tässä työssä käytettään Catapult-C:tä, jonka avulla luodaan HLS toteutus HEVC video koodekin intra-ennustuksesta FPGA:lle (Field Programmable Gate Array). HEVC enkoodeerina käytetään avoimen lähdekoodin Kvazaaria, joka kehtitetty TTY:llä. Työn tavoitteena on toteuttaa kiihdytin intra-ennustukseen, nopeammin kuin se olisi mahdollista rekisteritason HDL kuvaksella (Register Transfer Level, RTL) ja silti saavuttaa vertailukelpoisia tuloksia.

Tämä työ esittää kuusi kehitysversiota intra-ennustuksen kiihdyttimestä. Kiihdyttimen kompleksisuus kasvoi työn edetessä, sitä kun uusia ominaisuuksia lisättiin. Lopullinen versio pystyi suorittamaan intra ennustuksen, moodin kustannuksen laskennan sekä moodin valinnan teräväpiirto videolle 24.5 kuvaa sekunnissa käyttäen 11 662 ALM:ia (Adaptive Logic Modules) Altera Cyclone V FPGA:sta.

Tässä työssä tuodaan esille Catapult-C:n sekä HLS:n edut. Toteutuksen tulokset olivat laadullisesti vertailukelpoisia käsin tehtyyn RTL-koodiin. Karkeasti arvioiden VHDL-toteutus vie kuukauden, mutta saman tekeminen HLS:llä vie vain viikon. Suurin hyöty on muutosten tekemisen nopetumuinen sillä vain C-kielistä kuvausta on muutettava. Testipenkit ja RTL-koodi luodaan sen jälkeen automaattisesti.

ABSTRACT

Tampere University of Technology
Master's Degree Programme in Automation Technology
Panu Sjövall : High-Level Synthesis of HEVC Intra Prediction on FPGA
Master of Science Thesis, 49 pages
November 2015
Major: Digital circuits design
Examiners: Dr. Jarno Vanne, Prof. Timo D. Hämäläinen
Keywords: Kvazaar, HLS, HEVC, Catapult-C, C-Synthesis, intra prediction

High Efficiency Video Coding (HEVC) is the latest video coding standard in video compression. With HEVC, it is possible to compress the video with half the bitrate compared to the previous video coding standard, Advanced Video Coding (AVC), with the same video quality. Now even, the complexity of the encoder is significantly larger.

As designs become more and more complex, traditional hardware (HW) description languages (HDLs), such as Very High Speed Integrated Circuit Hardware Description Language (VHDL) or Verilog, can not be used to present the designs without increasing effort. The solution for this is a higher abstraction language for describing HW. High-Level Synthesis (HLS) is a way of using a programming language like C or C++ to describe the HW and automatically generating the HDL from it. This makes the code easier to understand and decreases the time used for implementing the design.

This Thesis uses Catapult-C to create an HLS-based implementation of HEVC intra prediction for a Field Programmable Gate Array (FPGA). The HEVC encoder used in this Thesis is open source Kvazaar which has been developed at Tampere University of Technology. The objective is to implement an intra prediction accelerator faster than implementing it with register-transfer level (RTL) using VHDL or Verilog and still get comparable area and performance.

This Thesis presents six development versions of the intra prediction accelerator. The complexity of the accelerator grows gradually, as more features were added to it. The final version is able to perform the intra prediction, mode cost computation and mode decision for Full HD video at 24.5 fps using 11 662 adaptive logic modules (ALMs) on an Altera Cyclone V FPGA.

This Thesis presents the benefits of Catapult-C and HLS. The implementation results were comparable to hand coded RTL but achieved with a fraction of the estimated time for a VHDL implementation. As a rough estimate, if something takes a month to implement in VHDL, it takes a week with HLS. The biggest gain with HLS is the fast process of changes. Only the C implementation needs to change. The testbench and the RTL-code are generated automatically.

PREFACE

This Master of Science Thesis was written in the Department of Pervasive Computing at Tampere University of Technology as part of research.

I want to thank my examiners Jarno Vanne and Timo D. Hämäläinen for giving me the opportunity to work in the university and for guidance during the work for this Thesis. I would also like to thank my co-workers Esko Pekkarinen, Marko Viitanen and Ari Koivula for all the help.

My deepest gratitude to my family, and especially for Mari, for all the support.

Tampere, 19th November 2015.

Panu Sjövall

CONTENTS

LIST OF TERMS AND ABBREVIATIONS

ALM	Adaptive Logic Module
ALUT	Adaptive Look-Up Table
AVC	Advanced Video Coding
AXI	Advanced eXtensible Interface
CTU	Coding tree unit
DMA	Direct Memory Access
FPGA	Field programmable gate array
FPS	Frames per second
HD	High-definition
HDL	Hardware description language
HEVC	High Efficiency Video Coding
HLS	High-Level synthesis
HW	Hardware
IP	Intellectual property
LE	Logic element
RDO	Rate-distortion optimization
RDOQ	Rate-distortion optimized quantization
RISC	Reduced instruction set computing
RTL	Register-transfer level
SAD	Sum of absolute difference
SAO	Sample adaptive offset
SATD	Sum of absolute transformed differences
VHDL	Very high speed integrated circuit hardware description language

1. INTRODUCTION

High Efficiency Video Coding (HEVC) standard is the latest milestone in the progress of video compression. With HEVC, it is possible to compress the video with half the bitrate compared to the current mainstream Advanced Video Coding (AVC) standard without sacrificing video quality, but at a cost of increased encoder complexity. In *all-intra coding*, HEVC reduces the bitrate by 23% compared to AVC with the same quality, but at about 3.2x encoding complexity [1].

Designing, implementing, and verifying new hardware (HW) takes more and more time as the complexity increases. Traditional hardware description languages (HDLs) including Very High Speed Integrated Circuit Hardware Description Language (VHDL) and Verilog are laborious and error-prone in large projects. Nowadays, most of the time is used for finding, fixing, and minimizing errors. With a higher abstraction level language, the focus is on the algorithm and not on the register-transfer level (RTL) and timing. High-Level Synthesis (HLS) tools promise to generate high-quality RTL and to greatly accelerate the design time. HLS is able to automate the process from a high level model, usually done in C, to RTL and thus is able to eliminate the source of many errors that could come from implementing the RTL manually. This also reduces the overall verification effort. The HLS tool used in this Thesis is Catapult-C that supports hardware description with C, C++, and SystemC.

As HEVC is a very complex encoder requiring a lot of processing power, it is a perfect candidate for Field Programmable Gate Array (FPGA) acceleration. Implementing a full HEVC encoder by hand for an FPGA would be a on year task. Writing RTL for an FPGA is comparable to writing assembly for a CPU. The most optimum result is obtained in this way, but it will need a lot of effort.

The main purpose of this Thesis is to use HLS design and implement an intra prediction accelerator for Kvazaar on an FPGA. Kvazaar is the leading open source HEVC implementation at the moment. The goal is to show how fast a complex hardware accelerator can be designed and implemented using HLS, and still get result that are comparable to hand-made VHDL or Verilog.

The work was done by first getting familiar with HLS and Catapult-C by implementing an H.263 encoder as a proof of concept. After getting the example implementation working, the work on HEVC and Kvazaar was started by first implementing small portions of the system and incrementally adding more functionality. This Thesis shows and explains the designs and results of each development version.

The structure of the Thesis is as follows: Chapter 2 briefly introduces HEVC, HLS, Catapult-C and FPGAs. This chapter also discusses and presents related work in the field. Chapter 3 discusses the HLS design flow and coding style with Catapult-C. In Chapter 4, first runs with HEVC and Kvazaar are done together with profiling. Chapter 5 introduces intra search and related algorithms more precisely. Chapter 6 presents the work done and different development versions of the intra prediction accelerator. Finally, Chapter 7 concludes the Thesis.

2. BACKGROUND

This chapter briefly introduces the main topics of this Thesis: High Efficiency Video Coding (HEVC), High-Level Synthesis (HLS), as well as the used HLS tools, Field programmable gate array(FPGA) chips and boards. This chapter also presents the related work.

2.1 High Efficiency Video Coding (HEVC)

The standardization of High Efficiency Video Coding (HEVC) was formally launched in January 2010. It is the latest international video coding standard in the progress of video compression. It is developed by Joint Collaborative Team on Video Coding (JCT-VC) as a joint activity of ITU-T Video Coding Experts Group (VCEG) and ISO/IEC Moving Picture Experts Group (MPEG) The first version of HEVC was completed in January 2013.

With HEVC it is possible to compress video with half of the bits compared to Advanced Video Coding (AVC) without sacrificing the quality of the video. HEVC can also be used to deliver higher resolutions and higher frame rates [2, p.1-11].

Currently, there are three noteworthy open- source HEVC encoders: x265 [3], Kvazaar [4], and f265 [5], out of which only x265 and Kvazaar are currently under active development. Compared to x265, Kvazaar is more hardware-friendly being implemented in C from scratch. Therefore, Kvazaar is the HEVC encoder used in this Thesis.

This thesis focuses only in Kvazaar all-intra coding due to its appropriate complexity and design time. In intra coding, each frame is encoded individually, i.e. no temporal processing is performed outside of the current frame.

2.2 High-Level Synthesis (HLS)

HLS tools are able to generate high RTL implementations using high abstraction languages. The main point of HLS is to be able to automate the process from a high level model, usually done in C, to RTL. The use of HLS eliminates many errors that come from implementing the RTL manually. This greatly accelerates the design time while also reducing the overall verification effort [6, p. 1-4].

Figure 2.1 shows the traditional RTL design flow compared to the HLS design flow. It can be seen that the HLS design flow is much simpler than the RTL design

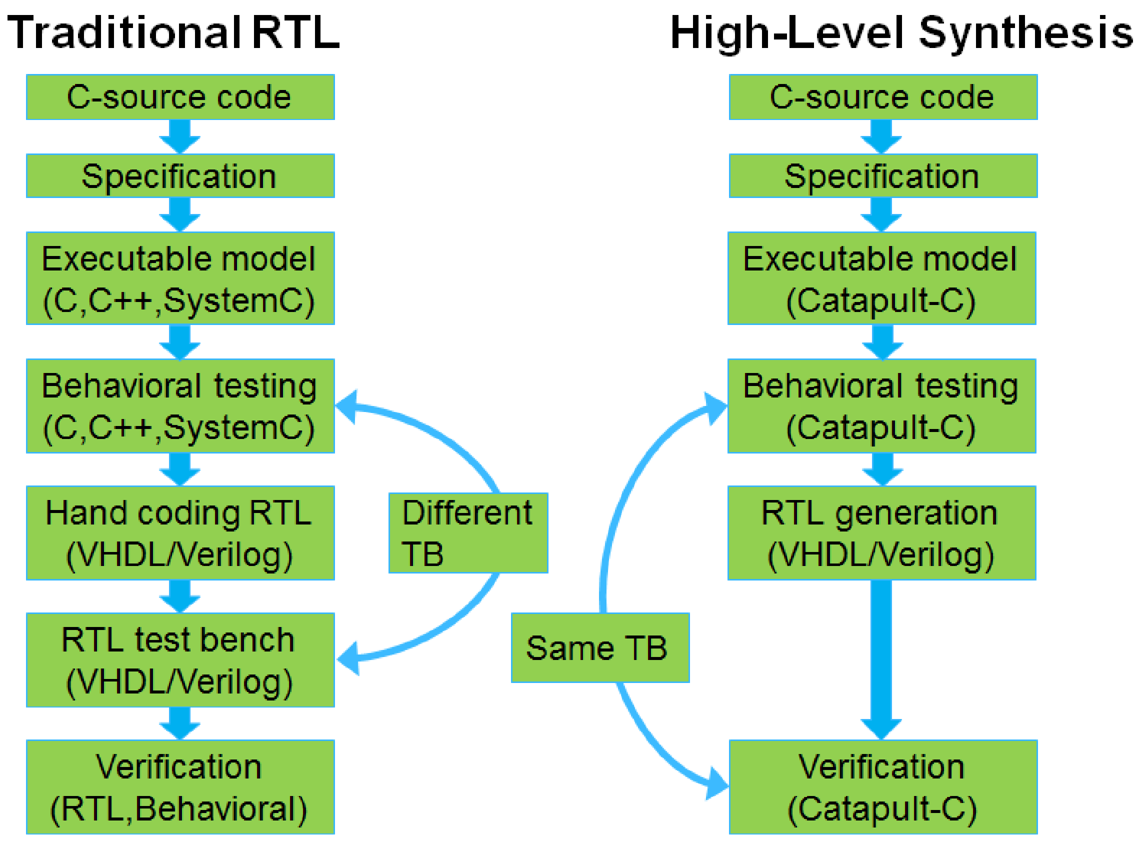
Traditional RTL
C-source code
Specification
Executable model (C,C++,SystemC)
Behavioral testing (C,C++,SystemC)
Hand coding RTL (VHDL/Verilog)
RTL test bench (VHDL/Verilog)
Verification (RTL,Behavioral)
Different TB
High-Level Synthesis
C-source code
Specification
Executable model (Catapult-C)
Behavioral testing (Catapult-C)
RTL generation (VHDL/Verilog)
Verification (Catapult-C)
Same TB

Figure 2.1: Traditional RTL design flow versus an HLS design flow

flow. In both flows, a specification is made from the existing C-source code, which is followed by an implementation of an executable model, which is commonly written in C,C++ or SystemC. The executable models produce the same output in both design flows, but the executable model in the HLS design flow might take more time to implement. The traditional executable model is just a representation of the structure of the HW with certain inefficiency in the code, but with the HLS executable model the code must be optimized for HW generation and done by following HLS coding rules. In behavioral testing, both executable models are tested against the specification, and should produce identical results.

After verifying that the executable model fulfills the specifications, the RTL can be generated automatically in HLS, versus hand writing the code in the RTL flow. This is one of the huge time saving techniques that HLS offers. The other one comes from testing, as HLS tools can reuse the same testbench for both RTL verification and behavioral testing. In the RTL flow, both testbenches must be created and updated separately.

There are several HLS tools in the market, e.g. Catapult-C from Calypto [7], Cynthesizer [8] and C-to-Silicon from Cadence [9], Vivado High-Level Synthesis from Xilinx [10], and Synphony C Compiler from Synopsys [11]. This Thesis does not compare the differences between different HLS tools but instead only relies on the results of Catapult-C, available through a university license.

2.2.1 Catapult-C

Catapult-C is developed by Mentor Graphics that is also known for software like Modelsim [12] simulator and Precision synthesis [13], which offers advanced RTL and physical synthesis for FPGAs. Catapult-C was acquired by Calypto Design Systems in 2011. Catapult-C is an HLS tool that allows generating RTL code using a higher abstraction language compared to VHDL or Verilog. Catapult-C supports C-to-RTL. It can generate RTL using ANSI C, C++ or SystemC [7].

The version of Catapult-C was updated twice during the work. Each update brought support to newer FPGA chips, new synthesis tools, and minor improvements to the resulting RTL. The first version used was 2011a.126, the next update was 2011a.200, and the latest one is 8.0. All these versions were used with the university license, which limits some of the features. For example, it does not allow the use of SystemC and hierarchical blocks in one project.

2.3 Field Programmable Gate Arrays (FPGAs)

FPGAs [14] are re-programmable logic circuits that allow fast development and real-time emulations. The results of this thesis are based on an FPGA execution instead of calculations or simulations.

The FPGA chips used in this Thesis are manufactured by Altera. At first a DE2 development and education board [15] was used since it was available immediately and it was supported by Catapult-C. Arria II GX FPGA Development Kit[16] was taken to use when more logic elements (LEs) were needed. The DE2 board has a Cyclone II FPGA chip with 33k LEs and the Arria II has 124k LEs. The final board in use was a Cyclone V VEEK board which is a System on Chip (SoC) FPGA board. Cyclone V has a dual-core ARM Cortex-A9 processor running at 900 MHz and 110K LEs in one chip [17]. A picture of the used Cyclone V VEEK board can be seen in 2.2.

2.3.1 ARM

ARM is the leading supplier of semiconductor Intellectual Property (IP). ARM doesn't manufacture their own semiconductor chips, but designs and licenses IPs. This way, other companies buy the licenses for IPs (e.g. ARM based CPUs) and use them in their own products.

ARM offers a wide range of microprocessors cores varying in performance, power, and cost. ARM processors are reduced instruction set computing (RISC) based

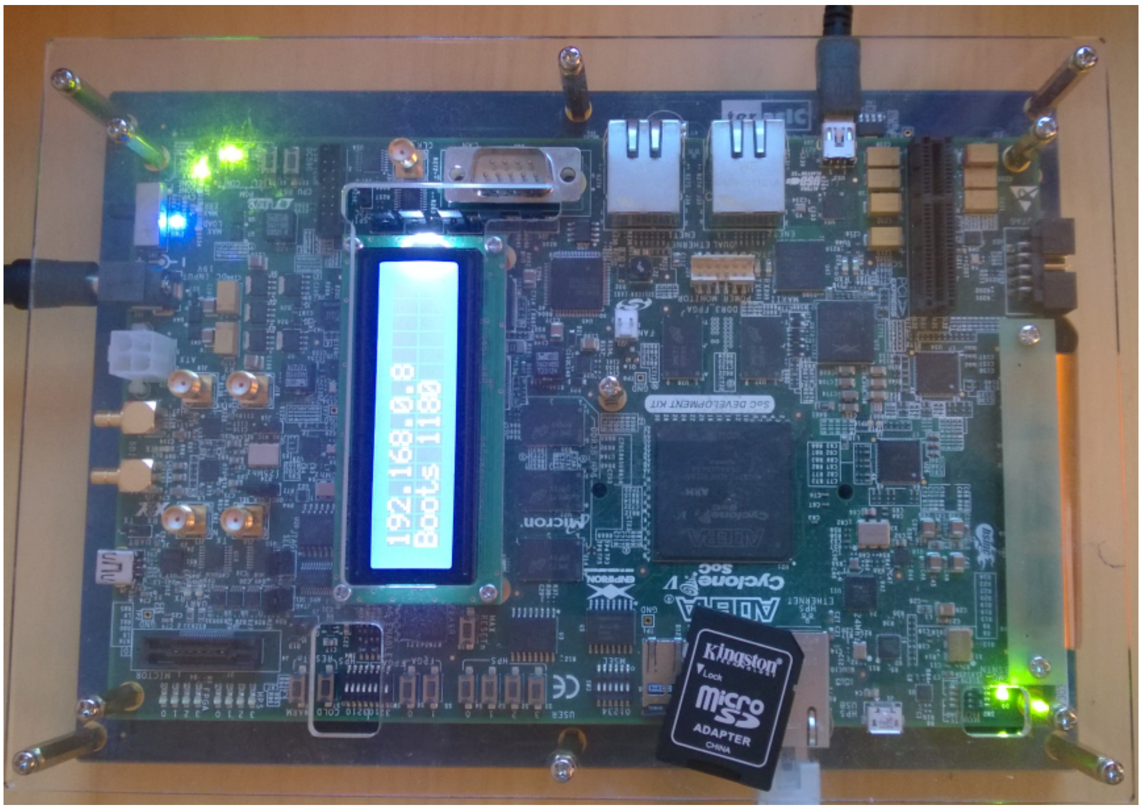

Figure 2.2: The Cyclone V VEEK board used in the final implementation

CPUs, i.e. they require significantly less transistors than typical x86 processors. Their reduced power, heat dissipation, and cost makes them ideal for portable devices. [18]

2.3.2 Altera SoCs

Altera System on Chip (SoC) FPGAs have an integrated ARM-based hard processor system (HPS) that consists of a processor, peripherals, interconnections to peripherals, the FPGA area, and a SDRAM Controller Subsystem as depicted in Figure 2.3. An FPGA integrated hard processor enables higher CPU clock frequencies compared to a soft processor synthesized on a FPGA and still have a fast and simple interconnection to the FPGA fabric [20].

The Microprocessor Unit (MPU) Subsystem is connected to the L3 Interconnect as shown in the Figure 2.3. The L3 Interconnect is an Advanced eXtensible Interface (AXI) bus structure. The AXI bus offers high performance and supports high clock frequency system designs and high speed interconnections. The ARM processor can access the FPGA Portion through the L3 Interconnect, using the HPS to FPGA and the Lightweight HPS to FPGA bridge. The processor has a dedicated line to the SDRAM Controller Subsystem through the L2 Cache. The SDRAM Controller Subsystem can also be accessed via the L3 Interconnect and the FPGA portion

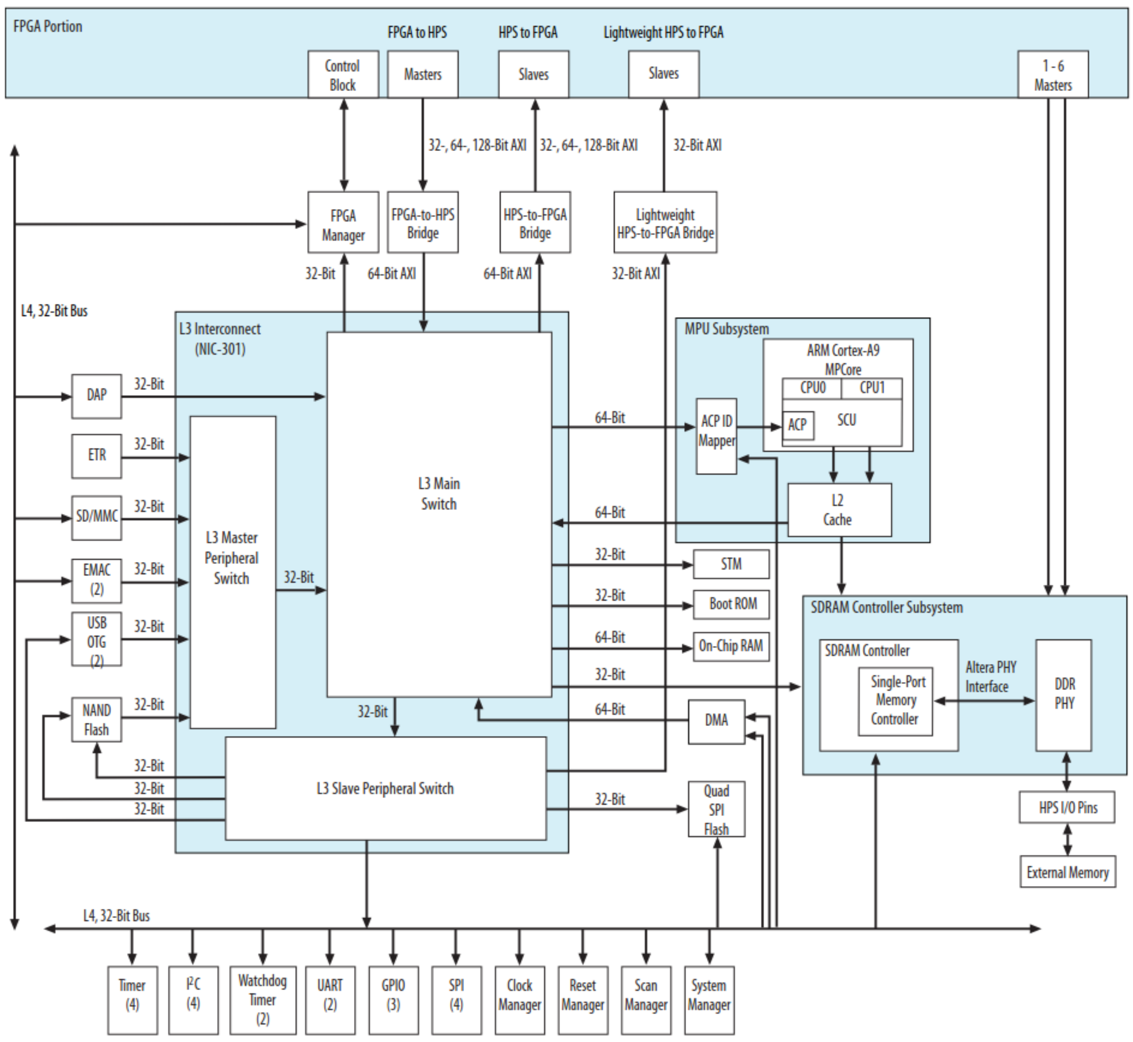

Figure 2.3: Block diagram of the Hard Processor System in Altera SoCs[19, p. 499]

directly, allowing fast access to the HPS External Memory from the FPGA portion.

2.4 Related work

A master's Thesis [21] by Ayla Chabouk and Carlos Gómez also utilizes Catapult-C. Their purpose was to study, analyze and test Catapult-C with reference models provided by ARM Sweden. They compared the correctness and quality of the RTL code generated by Catapult-C against handwritten RTL description of the models. They found the following advantages in using HLS: 1) A well known programming language like C in HLS makes the code easier to write and easier to understand; 2) HLS tools include a useful way to verify the C code and the RTL code with one testbench; 3) HLS tools save time. They also found some disadvantages in using HLS, which are: 1) The control of the design in C code is not as detailed as in the RTL description, so it is not possible to write cycle-accurate descriptions; 2) HLS has problems with complex blocks.

At the moment, the only published HLS-assisted HEVC intra encoder implementation is Author's previous work [22], which is an older version of the same accelerator implemented in this Thesis. There are some non-HLS implementations for the core HEVC functions, like intra-prediction. One of the works presents an intra prediction FPGA accelerator that can predict 17.5 Full HD frames per second. The implementation supports all intra block sizes from 4x4 to 32x32 and uses 15 589 adaptive logic modules (ALMs) on Altera Arria II [23].

An FPGA implementation capable of real-time HEVC encoding of 8k video is presented in [24]. It utilizes 17 boards, each having 3 FPGA chips. Each board is capable of encoding full-HD at 60 fps. Comparison with this work is challenging due to lack of specifics on algorithm speeds, FPGA chips, and the used area.

There also exits HLS implementations for AVC. One paper presents a complete design of an H.264 encoder with Catapult-C as the HLS tool [25]. The Authors conclude that using an HLS design flow did not make the design and implementation process faster compared to a traditional RTL design flow. This was mainly because it takes some time to learn how to use the HLS tool. They do say that coding and simulation times are reduced, because high level description of HW with C is easier than writing RTL descriptions. They state that there is a huge benefit with the reusability of HW C descriptions, as the target RTL is generated depending on constraints like clock frequency. So if they started to work e.g. on a HD encoder after the SD encoder, they could reuse a lot of code from the SD encoder project.

The work in [26] presents an HLS design flow and implementation of H.264 Deblocking filter, using Catapult-C as the HLS tool. The obtained results are even better than those of some state-of-the-art architectures with the same operating frequency. There was only one implementation that was better, but it was a highly hand optimized, and with a long development time.

3. HLS DESIGN FLOW WITH CATAPULT-C

This chapter shows the proof of concept work done with Catapult-C and presents the design flow and verification process used with HLS.

3.1 Proof of concept

Starting to work with Catapult-C and implementing the first design does not take much time. However, really understanding the functionality of the code does take some time without any help from more experienced users. Because Catapult-C is a licensed commercial software, there are practically no online discussions, that tend to be a good source for help with software like this. Catapult-C comes with few tutorials to get started with, but these tutorials are minimal. Catapult-C includes an HLS Blue Book [6]. The book is very comprehensive on what kind of code should be written to get the desired results.

The first tests with HLS and Catapult-C were made with the H.263 encoder that is a predecessor of HEVC. Because the main task of this Thesis was to accelerate HEVC with HLS, getting comfortable with Catapult-C even before taking a look at HEVC was the first priority. Accelerating H.263 first acted as a proof of concept

A ready made H.263 encoder running on a NiosII [27] soft processor synthesized on a DE2 FPGA board was available. This software version of the encoder later worked as a reference output when testing. The work started by taking the code and trying to generate RTL from it. Creating the top level interface for the accelerator to get the required data in and the calculated data is not hard. When trying to generate the RTL from the C code with minimal changes, a common problem is that Catapult-C optimizes everything away as it examines that nothing is happening. Catapult-C usually concludes this if there are no outputs or some conditions for calculations are not activated. Catapult-C gives very minimal information on the matter other than just informing everything is optimized away.

Catapult-C treats the top level function as a loop. This creates few challenges on how to write the top level function. The code listed in 3.1 shows a few of the problems mentioned before. Now, the function is executed in a loop so the integer a is always zero The same happens with the table. Static makes the values stay the same as they were after the functions execution ended. Now because a is always zero the code never reaches the part where data is written out, and therefore optimizes the output port away and possibly the whole design. The code also has a 1-bit port *irq*. If all other problems with the code would be solved *irq* would still never be high even for a cycle. Because *irq* is set low at the end and never used after it's set high, Catapult-C optimizes the port to being always low. This is a simple example with obvious errors, implying that with more complex designs finding similar errors may take considerable time.

Without any major coding related problems and with some experience with Catapult-C, creating the accelerator for the H.263 encoder took less than a week. In retrospect there are many small things that could have been done in order to increase the speed and lower the area cost of the design. The end result was more than encouraging, as the resulting frames per second (FPS) performance was better compared to a reference work that had the encoder running on a NiosII and a hand written VHDL quantization acceleration system.

After generating the RTL with Catapult-C, the next phase is to synthesize the RTL for the FPGA. This phase has some problems, related to the Catapult-C project settings, and took some time to solve. Because all arrays in C code become either registers or on-chip memory after RTL generation, it causes some tool specific problems. If only registers are used for arrays the design is usually faster, but the registers and the resulting muxes take a lot of area, resulting in that they are only useful with small arrays and when high speed is essential. The solution for saving area is to map the arrays to an on-chip memory, that exists as a dedicated memory on the FPGA

```
void top_level(ac_channel<ac_int<8,false> > in,
               ac_channel<ac_int<8,false> > out,
               ac_int<1,false> *irq)
{
  int a = 0;
  int table[10];
  table[a] = in.read();
  if(a == 9) {
    int b = 0;
    int sum = 0;
    for(b = 0; b < 10;b++) {
      sum += b;
      out.write(sum);
    }
    *irq = 1;
  }
  else{
    a++;
  }
  *irq = 0;
}
```

Listing 3.1: Catapult-C example

chip. The on-chip memory takes 1 cycle to read and write per address, so accessing a single value is fast but accessing multiple values sequentially is slow compared to parallel access to all register values. The synthesis tool for the RTL is set from the Catapult-C project settings. In this Thesis the synthesis tool used was Precision RTL Synthesis 2014b (with Catapult-C 8.0) and 2013b (with previous Catapult-C versions). The synthesis tool is important when using a library component like a single-port or dual-port on-chip memory. When an array is mapped to a single-port or a dual-port memory, Catapult-C adds a memory model to the generated RTL for simulation purposes. If the generated RTL would be synthesized directly with Quartus II [28], it would end in an error because Quartus II does not know what to do with the memory model. This is why Precision Synthesis is needed in the middle to switch the memory model to an FPGA chip specific on-chip memory format, by synthesizing a netlist of the RTL. The generated netlist can then be place & routed with Quartus II without errors.

The most specific tool related problem with Catapult-C and Precision Synthesis is that the only way to get everything working is to do the following: The Catpult-C generated Verilog RTL should be used for synthesis, instead of VHDL. After Precision Synthesis, Verilog Quartus Mapping File netlist should be used for place & route, instead of Electronic Design Interchange Format, VHDL or Verilog netlist. This seemed to be the case at least with these specific tools, otherwise an error free compilation was not guaranteed.

3.2 Design Flow

Figure 3.1 shows the whole general design flow based on the proof of concept work. The work flow depicts the process from C source code to FPGA, and this work flow was followed during the work on this Thesis. First the source codes are taken form an existing implementation, e.g. function or an algorithm. Next the source code is modified to work in Catapult-C. Only a single testbench is created, because it can test both the software implementation and the RTL generated code with Catapult-C. The software implementation is tested before the RTL generation. Project specific settings are applied to Catapult-C e.g. FPGA chip and clock frequency. After generating the RTL, it is tested with the same testbench as the software version. At this point the project settings can be re-evaluated for better results, e.g. higher frequency, or loop unrolling for more parallelism. Once satisfied with the results, the RTL code is taken to Precision Synthesis for netlist generation, after which the netlist is taken to Quartus II for place & route. Quartus II generates the FPGA image with which the FPGA is programmed.

3.3 Verification

With HLS and Catapult-C, the verification effort is minimal. The verification process consists of testing the executable model and the RTL with the same testbench. The nature of the testbench is to only test the functionality of the design. As the RTL generation phase is automated, the resulting RTL can be assumed to be valid. This means that the verification is done by testing the output with certain stimulus.

Being able to test the HW design in software first, gives the advantage of faster simulation times and better coverage. The RTL verification is required, but it is only done to reveal minor problems with the difference in software code and RTL code. For example, problems caused by typecasts and bit accurate types. These problems can still be minimized with a proper coding style.

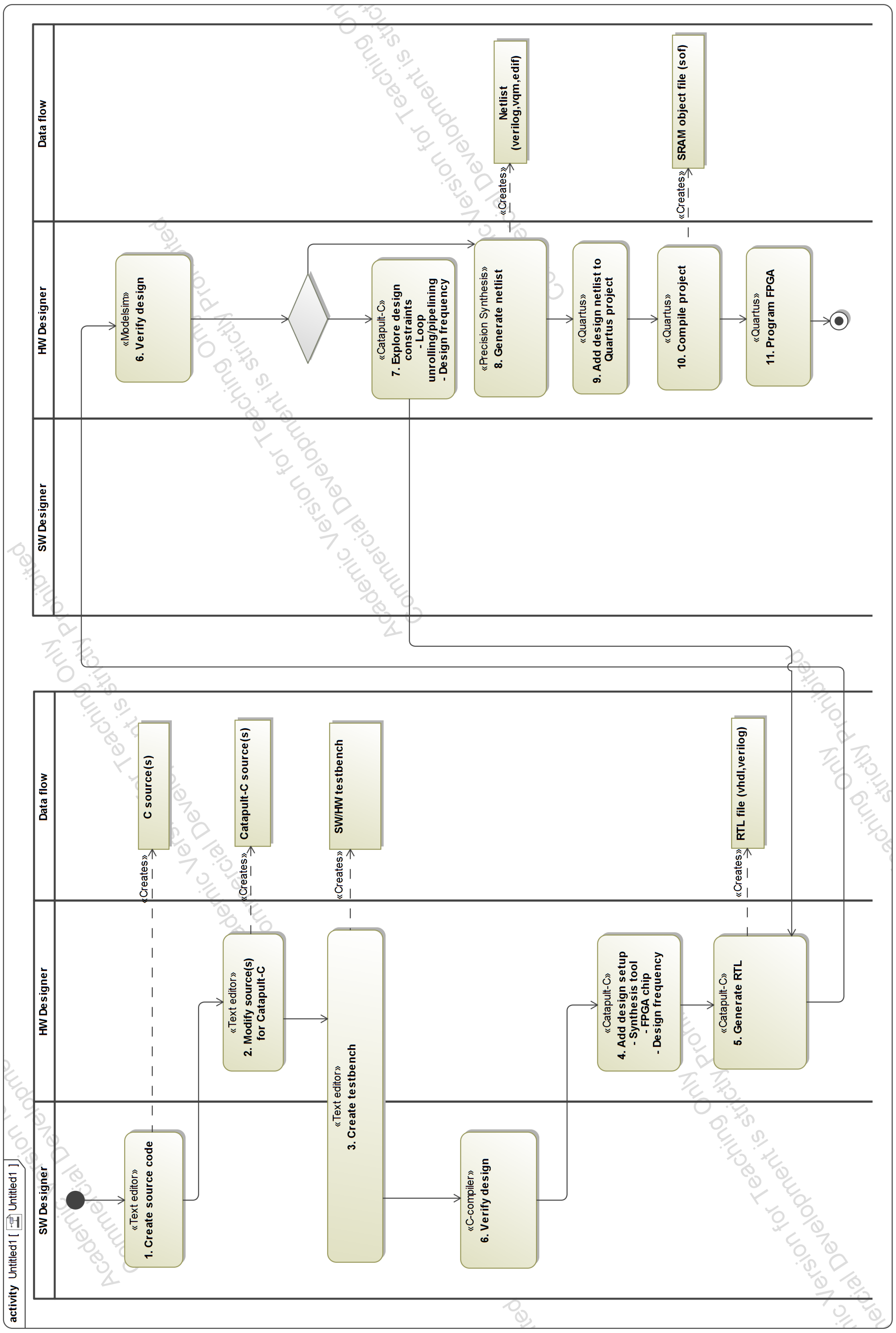
activity Untitled1 [Untitled1]
SW Designer
HW Designer
Data flow
«Text editor» 1. Create source code
«Creates»
C source(s)
«Text editor» 2. Modify source(s) for Catapult-C
«Creates»
Catapult-C source(s)
«Text editor» 3. Create testbench
«Creates»
SW/HW testbench
«C-compiler» 6. Verify design
«Catapult-C» 4. Add design setup - Synthesis tool - FPGA chip - Design frequency
«Catapult-C» 5. Generate RTL
«Creates»
RTL file (vhdl,verilog)
«Modelsim» 6. Verify design
«Catapult-C» 7. Explore design constraints - Loop unrolling/pipelining - Design frequency
«Precision Synthesis» 8. Generate netlist
«Creates»
Netlist (verilog,vqm,edif)
«Quartus» 9. Add design netlist to Quartus project
«Quartus» 10. Compile project
«Creates»
SRAM object file (sof)
«Quartus» 11. Program FPGA

Figure 3.1: The full design flow with Catapult-C from source code to programming to an FPGA

4. KVAZAAR HEVC INTRA ENCODER

Kvazaar [4] is an open-source HEVC encoder that is being developed from scratch in C by Ultra Video Group in the Department of Pervasive Computing at Tampere University of Technology. Kvazaar has a modular and portable structure that attains high coding efficiency with optimized speed and resources. Kvazaar is currently the leading open source intra encoder. Table 4.2 summarizes the basics of Kvazaar. The source codes of Kvazaar [4] are on GitHub, which is a web-based Git repository hosting service.

Table 4.1: Kvazaar HEVC coding parameters used in this work

Feature	**Kvazaar HEVC intra encoder**
Profile	Main
Internal bit depth, color format	8, 4:2:0
Coding modes	Intra
Sizes of luma coding blocks	64x64, 32x32, 16x16, 8x8
Sizes of luma transform blocks	32x32, 16x16, 8x8, 4x4
Sizes of luma prediction blocks	64x64, 32x32, 16x16, 8x8, 4x4
Intra prediction modes	DC, planar, 33 angular
Mode decision metric	SAD
RDO	1
RDOQ	Disabled
Transform	Integer DCT (integer DST for luma 4×4)
4x4 transform skip	Enabled
Loop filtering	DF, SAO

Table 4.2: Kvazaar digest

Main developer	Tampere University of Technology
Source codes	github.com/ultravideo
License	GNU LGPLv2.1
Contributors	7 at TUT + 6 external
Language	C with intrinsics/ASM
Operating systems	x86, x64, PowerPC, ARM
Processors	DC, planar, 33 angular
Presets	RD1 for high-speed encoding, RD2 for high-quality encoding

Table 4.1 shows the coding parameters used with Kvazaar in this thesis. The most important values from the table are that only intra coding is used, all block

sizes and prediction modes are supported, Sum of Absolute Differences (SAD) is used as the mode decision metric, Rate-Distortion Optimization (RDO) is 1, Rate-Distortion Optimized Quantization (RDOQ) disabled, and transform skip enabled. These settings are chosen for reducing the complexity of the encoder.

This thesis does not cover all aspects of the encoder. As the main purpose of this thesis was to accelerate only a part of Kvazaar, a limited knowledge of the whole HEVC encoding process, but deep understanding of the functions to be accelerated, is sufficient. First runs with Kvazaar were done on a PC. These runs were done to get familiar with Kvazaar parameters and doing some step-by-step debug runs to better learn the encoding flow of Kvazaar.

When the work on this Thesis started, Kvazaar intra encoder was still greatly under development. It was still missing some encoding tools for intra encoding and was just getting parallelization tools added. The changes in Kvazaar during the process of this Thesis had minimal effects to the work done. Kvazaar was easy to get working on a soft processor synthesized to an FPGA chip on a DE2 board. The software development environment for NiosII was *NiosII 12.1 Software Build Tools for Eclipse*, which was able to compile the source codes with minimal changes. Changes included removing and replacing unsupported code and writing a new method to read video input from memory as there is no trivial way to read data from an external storage with NiosII. This was the case with the earlier Kvazaar version that did not yet implement threads as a major part of the encoder. Compiling for NiosII with the later versions of Kvazaar would require major changes to the code. To get an understanding how demanding HEVC and Kvazaar is, the encoding speed with a QCIF resolution (176x144) video (Carphone [29]) was only 0,065 fps, with the NiosII processor running at 50 MHz.

4.1 Profiling Kvazaar

After the runs on NiosII, it was very clear that Kvazaar would need a major speed boost to get acceptable results. These improvements could be achieved by creating an HW accelerator for Kvazaar. Choosing what parts or functions to be accelerated can be hard, because not all functions can be accelerated depending on the structure and the functionality. By profiling the encoder and getting accurate time usages of all functions helps to narrow the search.

Gprof [30] is a tool for profiling programs and it is part of the GCC compiler. To compile a source file for profiling, the only thing needed to do is to specify a *-pg* flag when the compilation is done. When the gprof compiled application is run it produces a "gmon.out" file that can then be processed with gprof, which in turn outputs tables of processing times for all functions and also the cumulative time for all functions. This table is useful as-is, but from it a visual graph representation

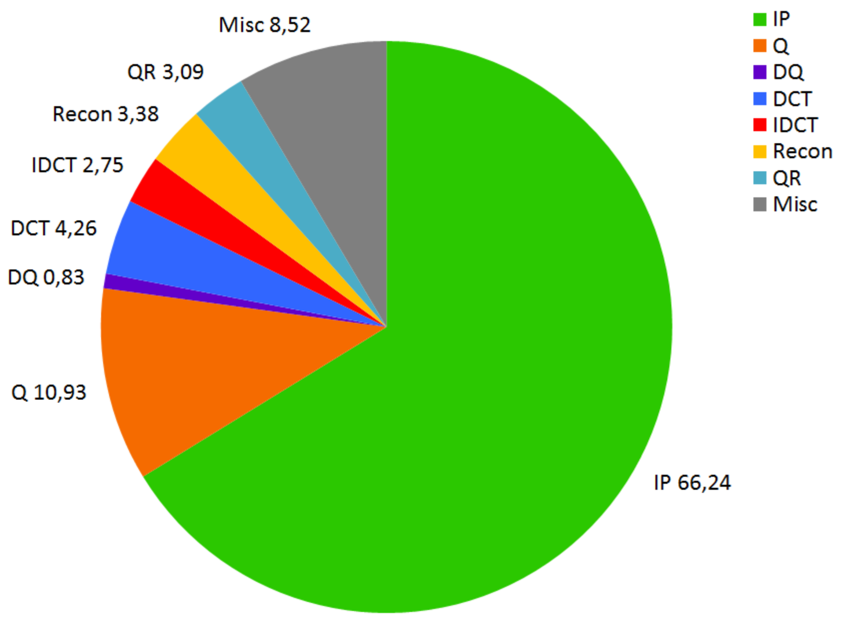

Figure 4.1: Kvazaar time usage diagram.

can be generated with gprof2dot [31].

Table 4.3 and Figure 4.1 shows the most time consuming parts of Kvazaar intra encoding. The video sequence (Kristen And Sara [32]) used to get these results had a 1280x720 resolution. From the Table 4.3 it can be seen that the most time consuming function in Kvazaar with quantization value 32 is *intra_get_angular_pred*. The function takes 39.23% of the overall encoding time when using full-intra search and 22.50% when using rough search.

Rough search implements a coarser version of the full-intra search. First it calculates the SAD for evenly spaced modes to select the starting point for a more refined search around the starting point.

Although the search_intra_rough function only takes 2.17% of the overall encoding time when using full intra search and 2.75% when using rough search, the cumulative time is much higher for both. For full intra search it is 66.24% and for rough search it is 41.83%. The cumulative time usage of *search_intra_rough* consists of all the functions marked purple.

The rest of the Thesis will focus on full intra search only, rather than rough search, as it will produce better picture quality and a better insight to accelerating algorithms by using an HLS tool.

Table 4.3: Most time consuming functions of Kvazaar in percentages

Full intra search (CPU only)		Rough intra search (CPU only)	
%	Functions	%	Functions
39.23	intra_get_angular_pred	22.50	intra_get_angular_pred
10.93	quant	15.63	quant
8.01	sort_modes	5.79	quantize_residual
4.26	sad_8bit_32x32_generic	5.79	sort_modes
4.17	sad_8bit_16x16_generic	2.89	intra_get_planar_pred
4.01	sad_8bit_8x8_generic	2.75	search_intra_rough
3.09	quantize_residual	2.60	partial_butterfly_32
2.84	intra_get_pred	2.46	sad_8bit_8x8_generic
2.17	search_intra_rough	2.32	partial_butterfly_inverse_16
1.59	partial_butterfly_16	2.03	intra_build_reference_border
1.42	intra_build_reference_border	1.88	sad_8bit_4x4_generic
1.34	intra_get_planar_pred	1.88	dequant
1.25	sad_8bit_4x4_generic	1.88	sad_8bit_16x16_generic
1.25	partial_butterfly_32	1.59	partial_butterfly_16
1.25	partial_butterfly_inverse_32	1.45	partial_butterfly_inverse_8
0.83	dequant	1.30	partial_butterfly_8
0.83	partial_butterfly_inverse_16	1.30	sad_8bit_32x32_generic
0.75	intra_pred_ratecost	1.16	intra_get_pred
0.75	partial_butterfly_8	1.01	intra_pred_ratecost
0.67	intra_recon	1.01	partial_butterfly_inverse_32
0.42	intra_filter	0.87	search_cu_intra
0.42	intra_recon_lcu_luma	0.87	intra_filter
0.33	search_cu_intra	0.87	intra_recon
0.33	fast_forward_dst	0.72	fast_inverse_dst
0.33	transformskip	0.43	fast_forward_dst
0.25	partial_butterfly_4	0.43	partial_butterfly_inverse_4
0.25	partial_butterfly_inverse_8	0.29	partial_butterfly_4
0.25	partial_butterfly_inverse_4	0.14	intra_recon_lcu_luma
0.17	intra_get_dc_pred	0.14	intra_recon_lcu_chroma
0.17	fast_inverse_dst	0.14	transform2d
0.08	transform2d	0.14	itransform2d
0.00	itransform2d	0.14	transformskip
0.00	intra_recon_lcu_chroma	0.00	intra_get_dc_pred

5. INTRA SEARCH

Intra search is a process of conducting a series of intra predictions and reconstructions in order to partition a Coding Tree Unit (CTU) into different modes and sized coding blocks. The intra search is done for every CTU, which can have a size of up to 64x64 pixels. The CTU can be divided into 64x64, 32x32, 16x16, 8x8 and 4x4 sized coding blocks.

Figure 5.1 shows the search order of each block in a CTU. Intra predictions for different blocks is done in the numerical order as seen in the Figure 5.1. This figure shows the worst case situation where every block is searched, but in a real scenario that might not be the case. After predicting the best mode for a specific block, reconstruction is done to get the actual coded pixels. These pixels are necessary for the adjacent blocks, as these pixels are used as the reference pixels for the next prediction. Using the actual coded pixels lowers the bitrate compared to using the original pixels.

5.1 Intra prediction

The HEVC intra prediction has three distinctive methods: planar, dc, and angular. The total number of intra prediction modes supported by HEVC is 35. The set of defined prediction modes consists of methods modeling various types of content typically present in video and still images [2, p. 91-93].

Figure 5.2 shows how the reference samples from the adjacent reconstructed blocks are utilized by the HEVC intra prediction modes. For example, when predicting a 8x8 block, the coordinate for the upper left pixel for the predicted block is (0,0), the needed above reference pixels go from (-1,-1) to (15,-1) and the left reference pixels go from (-1,-1) to (-1,15). All modes do not need all reference pixels to predict the block. Figure 5.3 shows an example of intra prediction in HEVC for 8x8 blocks for different modes and angles.

5.2 Angular prediction modes

Angular intra prediction is specified in HEVC to model different directional structures, which are usually present in image content [2, p. 97]. The angular intra prediction has 33 different prediction angles that can be seen in Figure 5.3 (examples 2 to 34). These directions are selected to provide a good trade-off between encoder

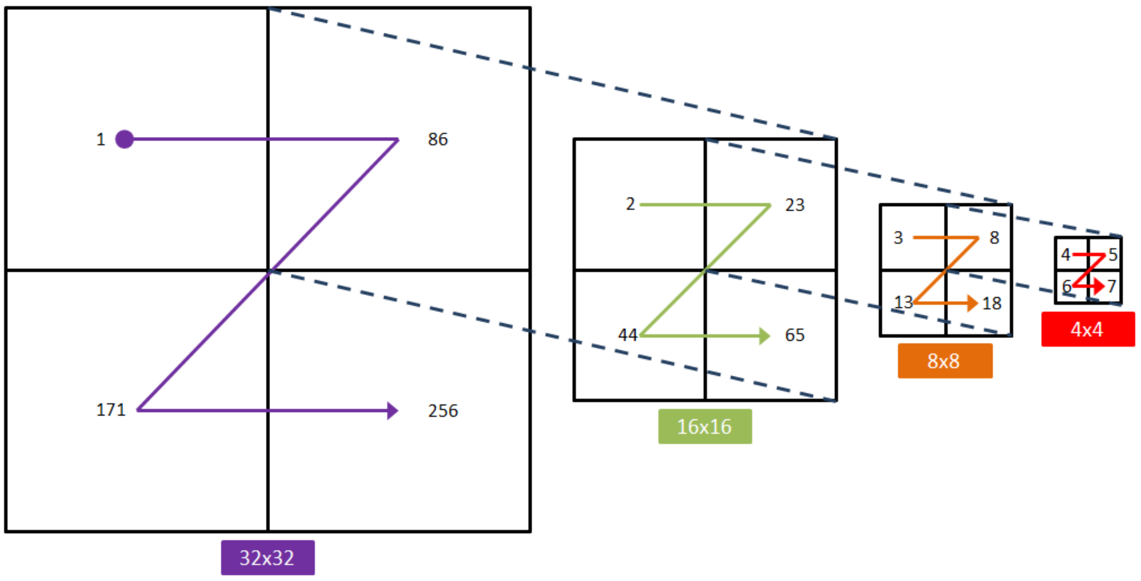

Figure 5.1: HEVC CTU search order

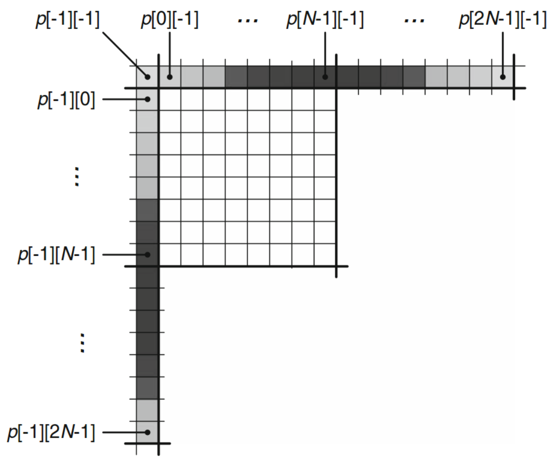

Figure 5.2: Example of reference pixels [2, p. 93]

complexity and coding efficiency [2, p. 97]. The number of prediction directions in addition to the supported block sizes of HEVC offer more compression capabilities than the AVC standard. Angular prediction is performed by *intra_get_angular* function in Figure 5.4.

5.3 DC prediction mode

With DC prediction, the predicted block is filled with values representing the average of above and left reference pixels. With block sizes of 4x4, 8x8, and 16x16, the predicted block is further filtered to soften the left and above edges as seen in Figure 5.3 with example 1 [2, p. 101]. DC prediction is performed by *intra_get_dc* function in Figure 5.4

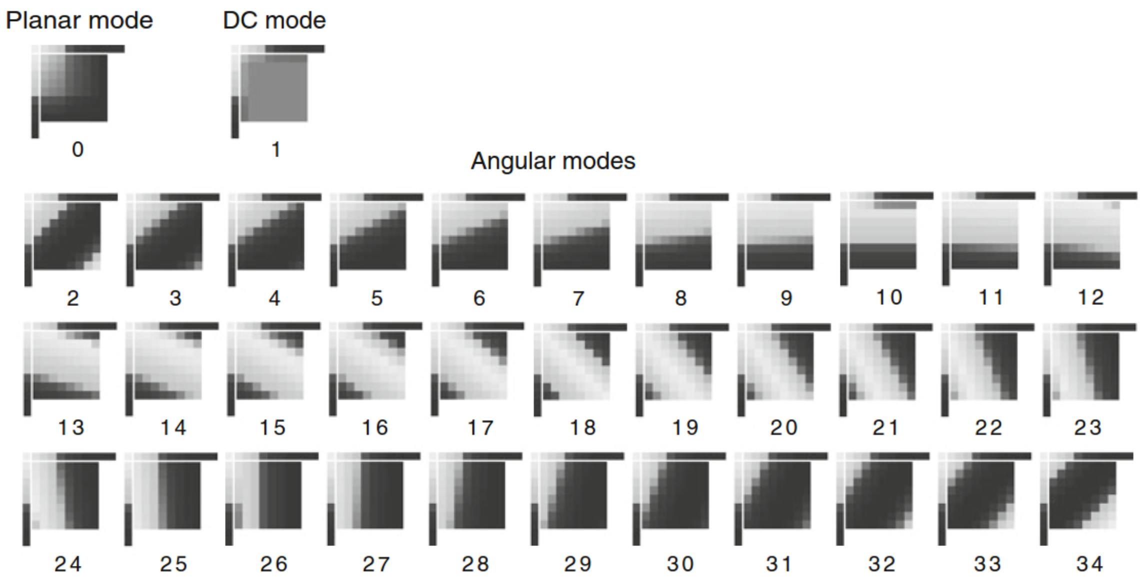

Figure 5.3: Intra prediction examples for 8x8 luma blocks [2, p. 92]

5.4 Planar prediction mode

Although angular prediction provides good approximations for structures with edges, it can create visible contouring in picture areas. Some blockiness can also be observed in smooth image areas when DC prediction is applied at low bitrates. The purpose of planar prediction is to generate a prediction surface without discontinuities on the block boundaries, as seen in Figure 5.3 with example 0, this way it overcomes some of the issues of predictions done with Angular or DC [2, p. 101]. Planar prediction is performed by *intra_get_planar* function in Figure 5.4

5.5 Mode cost computation

In digital imaging, it is useful to have a simple criterion for block similarities. In HEVC this criterion is used to select the best possible prediction mode. Calculating the SAD is one way to measure the differences between two picture blocks. The SAD is computed between the corresponding pixels from the original block and the block being compared to.

The other algorithm used to measure differences between two image blocks is the sum of absolute transformed differences (SATD). In SATD, a frequency transform is taken from the differences between the original block and the block being compared to. Therefore SATD is more complex and slower than SAD. Only SAD is used in this Thesis, as SATD was not implemented in Kvazaar until the accelerator was already finished.

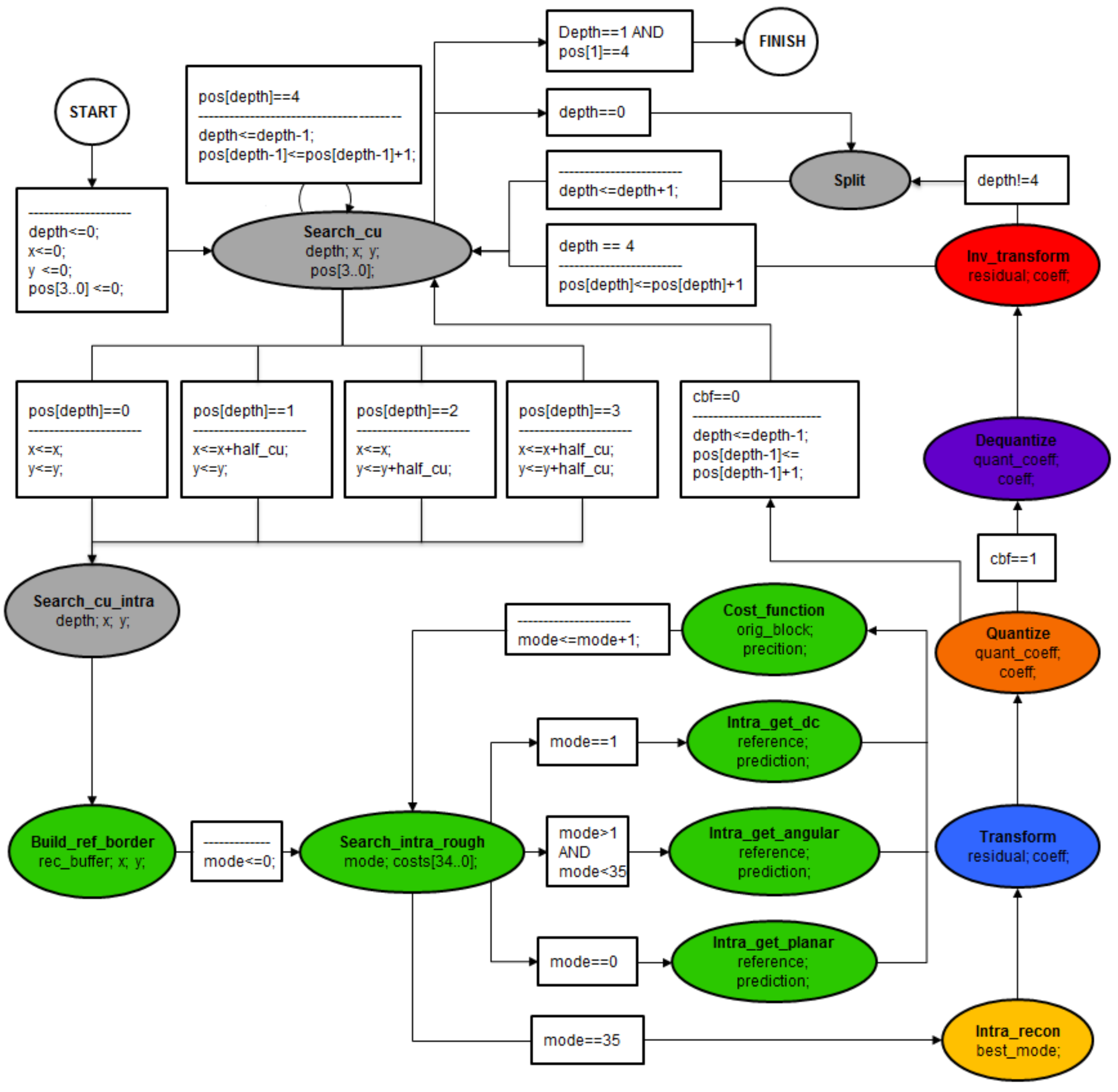

Figure 5.4: Kvazaar intra search flow diagram

5.6 Kvazaar intra search flow

Figure 5.4 shows the intra search flow in Kvazaar. The intra search starts at depth 0. The block size at depth 0 is 64x64 and 4x4 at depth 4. At depth 0 the 64x64 block is immediately split into four 32x32 blocks. The left upper 32x32 block is the first coding block to be predicted. The build_ref_border builds the reference pixels for the block. Search_intra_rough calls the prediction functions and chooses the best mode. The predicted block is reconstructed in order to have the reference pixels for adjacent blocks. During reconstruction, it is possible that all quantized pixels are zero and the coded block flag (cbf) is set to zero. This means that splitting the block does not necessarily give better results, reducing the number of blocks to be predicted. Otherwise the block is further split into smaller blocks. Search_cu determines, into which block sizes the CTU is parted.

6. HARDWARE DESIGNS

This chapter presents the verification method and the process of creating an HW accelerator for Kvazaar using HLS. All the measured results are for a QCIF (176x144) resolution video sequence (Carphone). The resolution was mainly limited by the speed and the memory of the first board. CycloneII DE2, that had only 8 MB of SDRAM and 50 MHz operating frequency. Although the other boards used have better performance and more memory, the same test sequence was used to have directly comparable results between different designs. The tables with profiling values were generated with an HD (1280x720) resolution video sequence (Kristen And Sara) as the PC version used the same sequence.

6.1 Verification

As discussed in Section 3.3, the verification with Catapult-C is easy. The presented HW blocks generated by Catapult-C are tested in software and in RTL with simulators. The system testing is done by running the system on the FPGA. The HW accelerator and the original source code are run in series and the results are compared. The results are expected to be identical.

The golden reference data for the HW blocks is generated with Kvazaar. Kvazaar code was modified to output real data input and output for each accelerated function for various test cases. The golden input data is then passed to the design under testing and output is verified against the golden output data. These test cases are done with a simulator to clear the most obvious errors, and the system test is used for more exhaustive testing. Errors in the code are solved by running the original function with debug prints against the debug prints in the HW design.

6.2 Accelerator I: Angular prediction modes

As seen in Table 4.3, *intra_get_angular_pred* is the most demanding function taking over 39% of the overall processing time. It was therefore a perfect candidate to start the accelerating process from.

The first step was to take the *intra_get_angular_pred* function to Catapult-C and generate RTL for it. Modifying the C implementation of the function to get functional RTL was fairly straightforward. As the proof of concept H.263 encoder

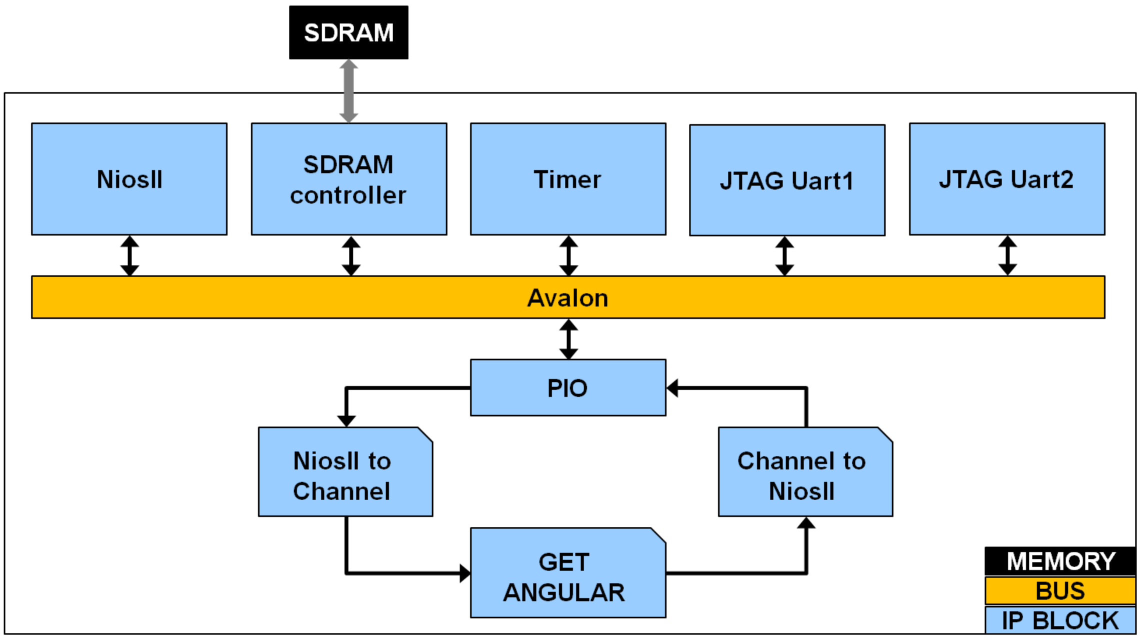

Figure 6.1: First implementation for intra prediction accelerator

was successful, the same design flow was used with *intra_get_angular_pred* function. The work was started by creating the top level function, that handles all the data communications from NiosII to the accelerated function.

Only small modifications were made to the code of the function to minimize the resulting HW area. For example, the original function contained a secondary array of 129 8-bit values both for the above reference pixels and for he left reference pixels. The arrays were oversized even for the largest 32x32 block. The number of pixels needed for the above and left reference pixels is $cu\_width * 2 + 1$. Another modification addressed the indexing of the reference pixels table inputed to the original function. The original function got a pointer to a two-dimensional table that had useful data only on the first row and in the first value of every row. This was changed to use two separate arrays, one with the above reference pixels and a second one for the left reference pixels. Other smaller optimizations included creating limits to loops. It is not important to know the limits of loops in C when compiling to CPUs, but it is when generating RTL. The loop limits are usually other variables, which means that Catapult-C cannot specify how many iterations a specific loop takes and thus cannot optimize or unroll the loop. For example, considering the following loop, `for(int a = 0; a < cu_width; a++)`, where cu_width is a 16-bit value, Catapult-C is unaware that the loop can only run for a maximum of 32 iterations. Instead it will expect the worst case, i.e., 65536. The way to avoid this is to specify the maximum limit as in `for(int a = 0; a < 32; a++)` and then break the loop with `if(a == cu_width-1) break;` inside the loop, as is also seen in Listing 6.1.

6.2.1 Design

Figure 6.1 shows the block diagram of the resulting design. This design was implemented for a Cyclone II FPGA chip on DE2 board. NiosII runs the whole encoding process excluding the angular prediction which is offloaded to the FPGA. NiosII is connected to the peripherals through Avalon bus. Peripherals include a SDRAM controller, Timer for the processor, and two JTAG Uarts for debug prints and data transfers. The accelerator is connected to Avalon through a Parallel input/output (PIO).

NiosII calculates the right reference pixels and the filtered ones and sends them through the PIO to the accelerator. Data amount sent to the GET ANGULAR block is $(2 * cu\_width + 1) * 4 + 2$ bytes. Filtered pixels are not sent when the block size is 4x4, in which case the data amount is only $(cu\_width + 1) * 4 + 2$. The GET ANGULAR block utilizes the mode and block size to decide whether to use the filtered or unfiltered reference pixels. Then it calculates the prediction for modes 2 to 35 and sends the predicted data back to the NiosII, through the PIO. GET ANGULAR block is able to calculate the prediction for all block sizes, so the amount of data generated per block is $33 * cu\_width * cu\_width$ bytes. Code for the GET ANGULAR block can be seen in Listing 6.1.

6.2.2 Performance

The presented design, with the *intra_get_angular_pred* function on the FPGA, is able to encode the test QCIF video at **0.13** fps. The GET ANGULAR block takes 2 449 LEs on the Cyclone II. The design is able to encode the video **1.7x** faster compared to the CPU only version which was able to encode the video at **0.07** fps. By accelerating the *intra_get_angular_pred* function, the overall time used in the *intra_search_rough* function decreases from 66.24% to 43.62% over the CPU only version. NiosII and the accelerator were both running at 50 MHz.

6.3 Accelerator II: Angular prediction modes with mode cost computation

After the *intra_get_angular_pred* function was offloaded to the FPGA the next phase was to offload more functionality to the FPGA. According to Table 4.3, quantization is the second most demanding function, but its acceleration would not give much better results. Implementing quantization on FPGA would need data to be transfered between the FPGA and the CPU multiple times, hindering the acceleration because of data transfer times. So the logical choice was to implement SAD calculation for the modes using *intra_get_angular_pred*. Altogether, the SAD calculation functions account for 13.69% of the time.

In order to calculate the SAD value, the predicted pixels and the original luma pixels of the same block are needed. The predicted pixels are generated by the GET ANGULAR block, so only the original luma pixels of the right block have to be sent to the FPGA. A new project was created with Catapult-C in order to have

```
#pragma hls_design top
void get_angular(ac_channel<uint_8> &data_in,
                 ac_channel<uint_8> &data_out)
{
  uint_8 width=0,threshold=0,distance=0,a=0,mode=0;
  pixel unfiltered1[65],unfiltered2[65],filtered1[65],filtered2[65];
  pixel* src1,src2;
  width = data_in.read(); threshold = data_in.read();
  // Reading all reference pixels
  for(a = 0; a < 65;a++){
    unfiltered1[a] = data_in.read();
    if(a == 2*width){break;}
  }
  for(a = 0; a < 65;a++){
    unfiltered2[a] = data_in.read();
    if(a == 2*width){break;}
  }
  if(width != 4){
    for(a = 0; a < 65;a++){
      filtered1[a] = data_in.read();
      if(a == 2*width){break;}
    }
    for(a = 0; a < 65;a++){
      filtered2[a] = data_in.read();
      if(a == 2*width){break;}
      }
  }
  // Calculate angular predictions
  for(mode = 2; mode < 35;mode++){
    if(width == 4){
      src1 = unfiltered1; src2 = unfiltered2;
    }
    else{
      distance = MIN(abs(mode - 26),abs(mode - 10));
      if(distance > threshold){
        src1 = filtered1; src2 = filtered2;
      }
      else {
        src1 = unfiltered1; src2 = unfiltered2;
      }
    }
    angular_pred(src1,src2,data_out,width,mode);
  }
}
```

Listing 6.1: Catapult-C code for calculating angular predictions

an HW-block that works in parallel with the GET ANGULAR block. The newly created SAD block gets the predicted pixels from GET ANGULAR one pixel at a time, and calculates the SAD. The SAD block has an interface to an on-chip RAM that holds the original luma pixels. The right pixels are read from the RAM as the predicted pixels arrive. The code for the SAD block is illustrated in Listing 6.2, where *orig_block* and *sads* are parameters for the function. Catapult-C can map a table to an single port on-chip RAM interface and use it as a normal array in C. The single port on-chip RAM interface is generated by Catapult-C. So, after the RTL is generated the interface can be connected to an external single port on-chip RAM, or in this case to the second interface of a dualport memory.

If the prediction mode is higher than 17, the pixels are predicted in transpose. The original source code flips the block before continuing, but it takes time. In order to minimize the area cost and the computation time in HW the SAD block calculates the SAD in transpose for those modes, as illustrated in Listing 6.2.

```
#pragma hls_design top
void sad(uint_8 orig_block[1024],ac_channel<uint_8> &data_in,
         uint_32 sads[34], ac_int<1,false> *irq)
{
...
  for(y = 0; y < 32;y++){
    for(x = 0; x < 32;x++){
    pred = data_in.read();
    // Vertical
      if((a > 17)){
        temp1 = orig_block[x*width+y] - pred;
      }
    // Horizontal
      else{
        temp1 = orig_block[y*width+x] - pred;
      }
      sad[a] += (abs(temp1));
    }
  if(x == cu_width-1) break;
  }
  if(y == cu_width-1) break;
...
}
```

Listing 6.2: Catapult-C code for calculating SAD

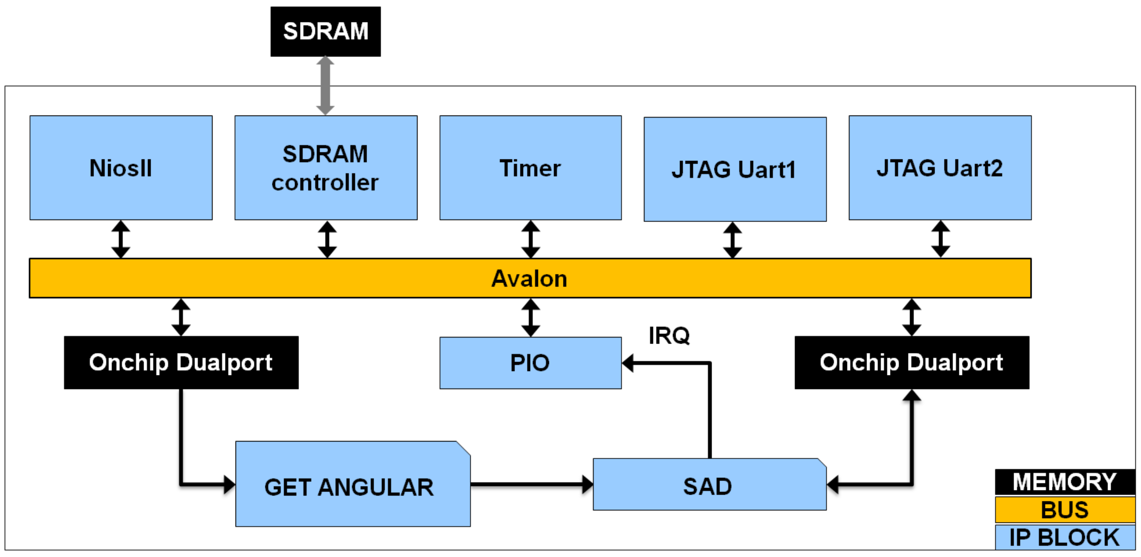

Figure 6.2: Adding SAD block

6.3.1 Design

Figure 6.2 shows the the block diagram of the resulting design. Here, the differences over the Accelerator I are the use of on-chip memories. The memories have one port connected to the Avalon bus and the other port to the GET ANGULAR and SAD blocks. Now, the PIO is only used to create an IRQ signal to the NiosII.

Here, the filtered and unfiltered reference pixels are sent to the GET ANGULAR block through an on-chip RAM. The RAM is sized $(max\_cu\_width * 2 + 1) * 4 + 1$ bytes to have enough space for a ready flag and for all reference pixels. The ready flag is in the first index indicating that the reference pixels are all written to the memory before GET ANGULAR starts to read the data and process it. The same concept is used for the SAD block, where size of the the on-chip memory is $max\_cu\_width * max\_cu\_width$ and 140. The $max\_cu\_width * max\_cu\_width$ bytes is needed for the original CTU pixels, so that the SAD can be calculated as explained before. The $132 + 4$ bytes is needed for the ready flag and for 33 32bit SAD values. As in the Accelerator I, the GET ANGULAR block calculates the prediction for modes 2 to 35, but this time sends the predicted data to the SAD block which calculates the SAD value for all 33 modes and saves all the SAD values to the on-chip memory. After all 33 SADs have been calculated the SAD block signals NiosII with an IRQ.

6.3.2 Performance

Accelerator II is able to encode the QCIF video at **0.13** fps. The GET ANGULAR block needs 2 449 LEs and the SAD block 854 LEs on the Cyclone II. The design is able to encode the video **2.0x** faster compared to the CPU only version.

Compared to the Accelerator I the improvement is **1.2x** and the time used in the *intra_search_rough* function decreases from 43.62% to 32.24%. NiosII and the accelerator were both running at 50 MHz.

6.4 Accelerator III: All prediction modes with mode cost computation and selection

After the GET ANGULAR block and the SAD block working successfully on HW, a more complete intra prediction accelerator (IP ACC) was designed, by including the prediction for modes 0 (planar) and 1 (DC). Listing 6.3 shows the updated *get_angular* function. The only differences are the name of the top level and the last for-loop, that now includes modes 0 and 1, as well as the function calls for *planar_pred* and *dc_pred.* The algorithms for planar and DC are much simpler compared to angular prediction, making it fast to get the first version working in Catapult-C after adding them to the existing GET ANGULAR code. The same optimization techniques were used for the new code, as covered in section 6.2

6.4.1 Design

Figure 6.3 shows the first complete version of the IP ACC. The only difference over the Accelerator II is that the GET ANGULAR block is now a complete INTRA PREDICTION block that performs the prediction for all modes sequentially. The data sent to the INTRA PREDICTION block is equal to that sent to the GET ANGULAR block before. The SAD block calculates the SAD value for all modes as the predicted data arrives. The on-chip memory connected to the SAD block contains 8 bytes more data for two extra 32bit SAD values. Since the SAD block calculates values for all modes, it can also sort them accordingly. This means that *sort_modes* function, which uses the third most time in the encoder, is offloaded to the FPGA.

6.4.2 Performance

The Accelerator III is able to encode the QCIF video at **0.17** fps. The IP block consumes 5 454 LEs and the SAD block 854 LEs on the Cyclone II. The design is able to encode the video **2.6x** faster compared to the CPU only version. Compared with the Accelerator II, the improvement is **1.3x** and the time used in the *intra_search_rough* function decreases from 32.24% to 22.09%. NiosII and the accelerator were both running at 50 MHz.

6.5 Accelerator IV: Parallel implementation of Accelerator III

All three prediction functions, *intra_get_planar_pred*, *intra_get_dc_pred* and *intra_get_angular_pred*, use the same input data to calculate the prediction for all modes. In addition they have no dependencies between each other. Therefore it is possible to run the prediction and calculate the SAD for all modes in parallel. Making parallel prediction blocks means separate Catapult-C projects for all functions in order get them working in parallel.

As the INTRA PREDICTION block in the Accelerator III was able to calculate all modes, the reference pixels were only sent there. Multiple prediction blocks using the same data need a structure that writes the same data to all of them. Rather than instantiating 35 on-chip memories and writing the data to all of them with NiosII, an IP CTRL block was created that reads the data from the same on-chip memory as before and distributes the data to the prediction blocks in parallel. Creating a

```
#pragma hls_design top
void intra_prediction(ac_channel<uint_8> &data_in,
                      ac_channel<uint_8> &data_out)
{
...
  for(mode = 0; mode < 35;mode++){
    if(width == 4 || mode == 1){
      src1 = unfiltered1;src2 = unfiltered2;
    }
    else if(mode == 0){
    src1 = filtered1;src2 = filtered2;
    }
    else{
      distance = MIN(abs(mode - 26),abs(mode - 10));
      if(distance > threshold){
        src1 = filtered1;src2 = filtered2;
      }
      else{
        src1 = unfiltered1;src2 = unfiltered2;
      }
    }
    if(mode == 0){
      planar_pred(src1,src2,data_out,width);
    }
    else if(mode == 1){
      dc_pred(unfiltered1,unfiltered2,data_out, width);
    }
    else{
      angular_pred(src1,src2,data_out,width,mode);
    }
  }
}
```
Listing 6.3: Catapult-C code for calculating all prediction modes

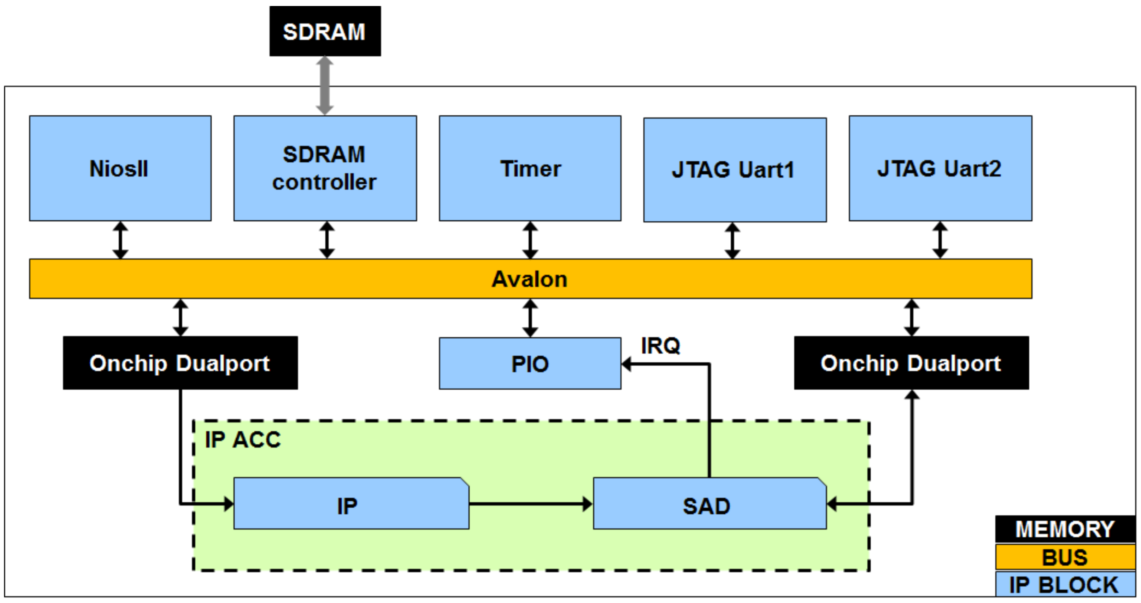

Figure 6.3: Support for all modes

```
#pragma hls_design top
void sad_parallel(uint_8 orig_block[1024],port *in[35],
                  uint_32 sads[37],ac_channel<uint_32> &config,
                  one_bit *irq)
{
  ...
  for(y = 0; y < 32;y++){
    for(x = 0; x < 32;x++){
      ...
      // Loop for calculating 35 SADs for 35 modes
      for(a = 0;a < 35;a++){
        ac_int<9,true> sad_temp = 0;
        input_temp = in[a]->read();
        // Vertical
        if((a > 17) || (a == 0) || (a == 1)){
          sad_temp = orig_block[x*width+y] - input_temp;
        }
        // Horizontal
        else{
          sad_temp = orig_block[y*width+y] - input_temp;
        }
        sad[a] += abs(sad_temp);
      }
      if(x == cu_width-1) break;
    }
    if(y == cu_width-1) break;
    ...
  }
}
```
Listing 6.4: Catapult-C code for SAD PARALLEL

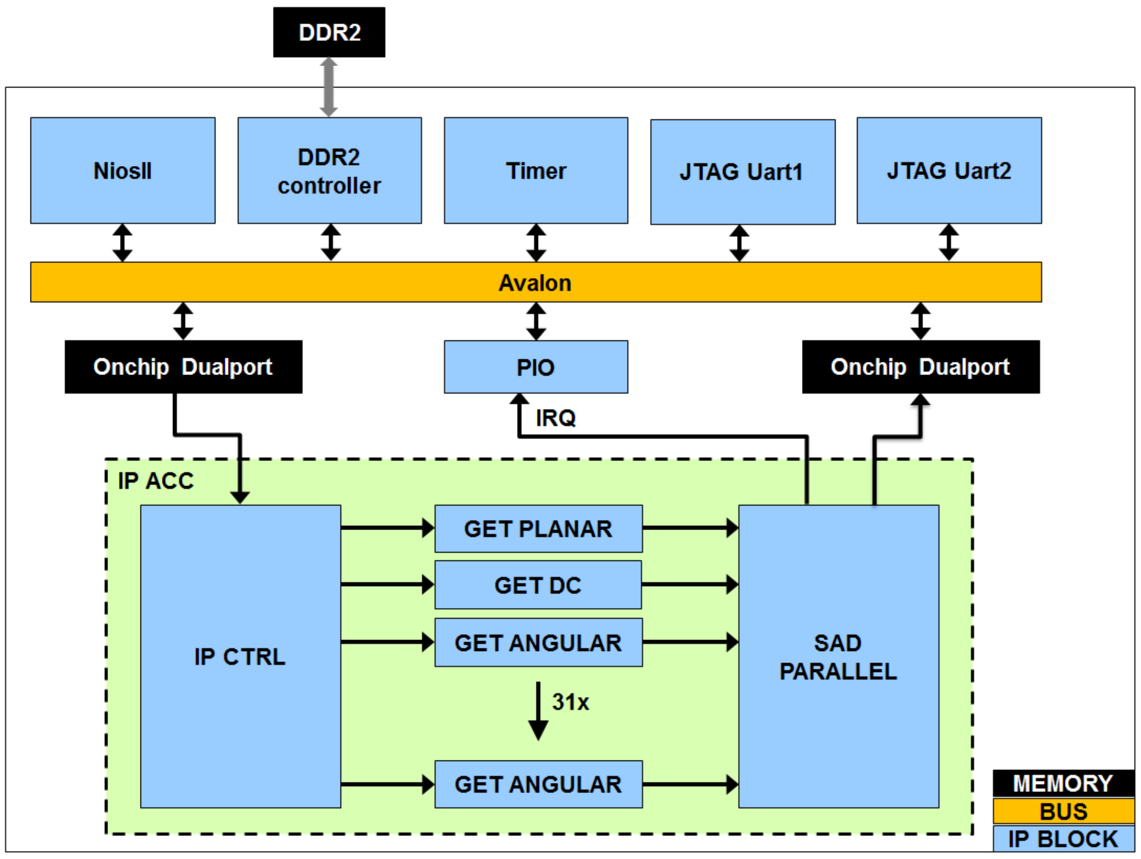

Figure 6.4: Parallel intra prediction

SAD PARALLEL block that calculates the SAD value in parallel was implemented through loop unrolling. A similar loop structure can be seen in Listing 6.4 and in Listing 6.2, except that there is a third inner loop `for(a = 0;a < 35;a++)` in Listing 6.4. This loop can be unrolled in Catapult-C project settings. This helps the coding process as the loops do not need to be unrolled manually, which would lead into code that is difficult to read and manage.

6.5.1 Design

Figure 6.4 shows the new parallel IP ACC as part of the entire system. The IP components above the Avalon bus are the same as in previous versions. The IP ACC now has an IP CTRL block, separate GET PLANAR, GET DC, and 33 GET ANGULAR blocks that work in parallel, and a SAD PARALLEL block. The size of the on-chip memory connected to the IP CTRL block is now half the size compared to the one in Figure 6.3.

In this design, only the unfiltered reference pixels are sent to the accelerator. The IP CTRL block selects the modes for the unfiltered and filtered reference pixels and calculates the filtered pixels in real-time as they are sent to the prediction blocks. All the prediction blocks start the prediction after getting the respective reference pixels.

The SAD PARALLEL block calculates the SAD value for all modes in parallel. If some mode is predicted slower than the others, the SAD PARALLEL stalls the other modes. For example, some modes can start the prediction faster, because not all modes need the same amount of reference pixels. In addition there are one cycle delays in some modes because of state changes. Otherwise, the prediction blocks can calculate the prediction one pixel per cycle in average.

The resulting IP ACC was so large that it did not fit in the low level CycloneII FPGA chip on the DE2, so the Arria II GX FPGA Development Kit with a larger entry-level FPGA chip was taken into use. NiosII and the accelerator were both running at 125 MHz.

6.5.2 Performance

The Accelerator IV is able to encode the QCIF video at **1.4** fps. The IP CTRL block takes **3 742** LEs, the GET blocks **21 627** LEs, and the SAD PARALLEL block **2 871** LEs on the Arria II. The combined area was **28 240** LEs which equals to **10 656** ALMs. The design is able to encode the video **3.0x** faster compared to the CPU only version, which is able to encode the video at **0.47** fps on the Arria II.

These results are not entirely comparable to the previous ones. When taking the operating frequencies into account, the frame rate of 0.07 fps obtained with NiosII on the CycloneII should be scaled on ArriaII as follows: $(125Mhz/50MHz)*0.07fps = 0.175$. The difference may be caused by memory speed and NiosII that takes different number of cycles per instructions. The improvement with the IP ACC is still **3.0x**.

6.6 Accelerator V: Integrating the Accelerator IV to ARM

Even though the NiosII is a good soft processor, it is not made for heavy calculations. The speed is mainly limited by the FPGA chip in use. So, an ARM hard processor is suggested to get a speed boost for the processor side. Altera SoC device has an integrated ARM processor and a CycloneV FPGA chip. Although the CycloneV is a newer chip than the ArriaII they both have almost the same amount of LEs. They are also about the same speed grade despite that CycloneV is a lower level FPGA. Linux operating system is run on the ARM to ease the use of a file system, a network connection, and threading.

The interface to the ARM uses an AXI bus whereas an Avalon bus is used on Arria II. Therefore, switching from NiosII to ARM requires some changes in the surrounding components. The most significant change is the way the data is sent to the accelerator. A VHDL implementation of a Direct Memory Access (DMA) was created for reading the data from the CPU data memory directly, using dedicated interfaces to the memory controller. Altera provided IPs PIO and on-chip memories,

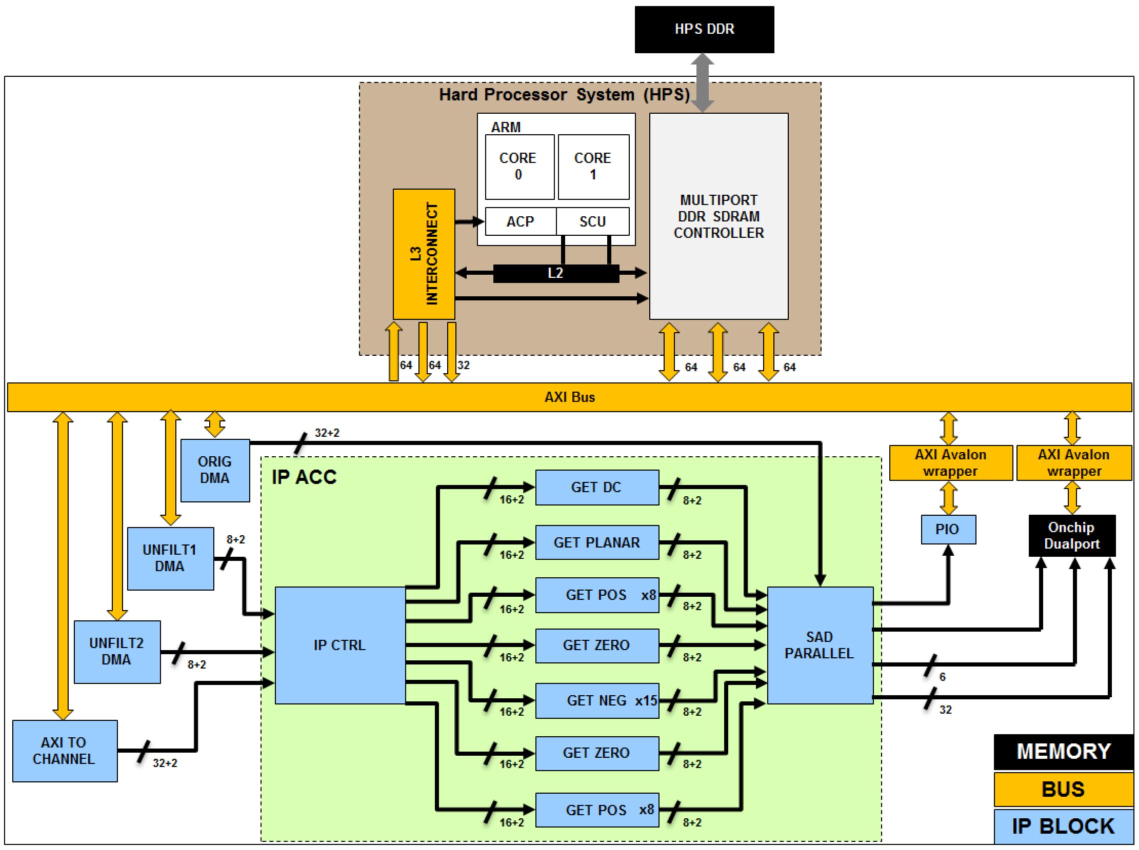

Figure 6.5: Final system on CycloneV

are still used, but as they are not AXI native, QuartusII generates a wrapper between them and the AXI bus. AXI bus and Avalon bus are similar enough for this to be possible.

Changes in the IP ACC include optimizations in all blocks. IP CTRL now has channels as inputs for the reference pixels compared to the memory interfaces seen in Accelerator IV. The GET ANGULAR block from Accelerator IV is further divided into three separate blocks GET POS, GET ZERO, and GET NEG according to the angle of the mode. As the GET ANGULAR in Accelerator IV had slightly different operations depending on the mode it was useless to have the same functionality in all modes. GET ANGULAR was a more generic block, compared to the three new ones. Doing this saved LEs on the FPGA and made the code more readable. The delivery of the original CTU pixels is also optimized for the SAD PARALLEL block. Before, the memory for the original pixels was updated every time for each coding block. This caused some duplicate data to be transfered for different sized coding blocks. Now, the whole CTU of original pixels is sent at once and only the coordinates are sent among the configuration data through the AXI TO CHANNEL block, which is a wrapper between the AXI bus and the Catapult-C generated channel.

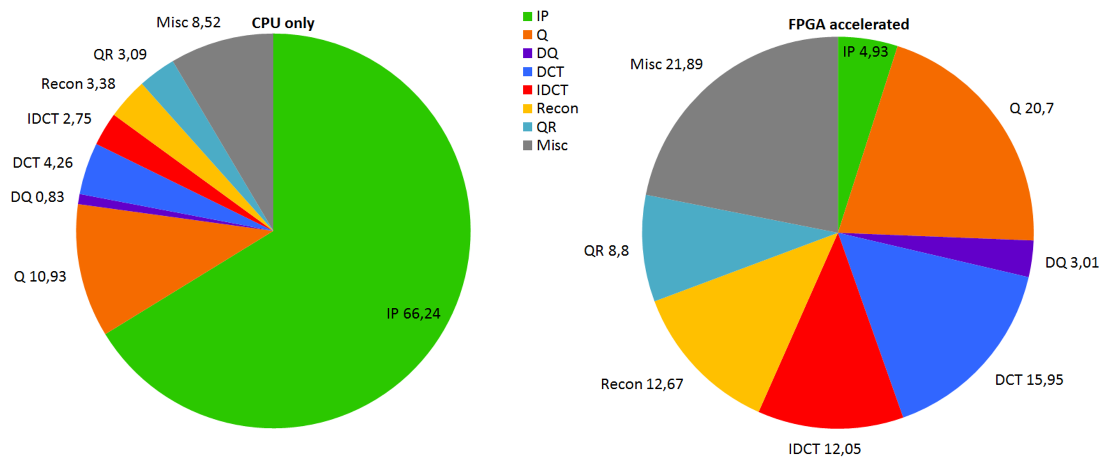

Figure 6.6: CPU only Kvazaar compared to FPGA accelerated Kvazaar

6.6.1 Design

Figure 6.5 shows the design on the SoC CycloneV. All the data sent to the IP ACC is read from the HPS DDR with ORIG DMA, UNFILT1 DMA, and UNFILT2 DMA blocks. The data is written to a specific address in the memory by a kernel driver. The encoder uses system calls, e.g., `ioctl()`,`write()`, and `read()` to interact with the FPGA. The encoder gives the pointer to the data as a parameter to the driver, and the driver copies the data to memory location reserved by the driver. After the data is copied to continuous memory locations, the DMA can start reading the data from the start address configured to the DMA beforehand. The IP ACC works pretty much the same way as in Accelerator IV, except for the changes explained in Section 6.6. The ARM is running at 900 MHz and the accelerator at 100 MHz.

6.6.2 Performance

With the Accelerator V on the SoC FPGA Cyclone V, the design was able to encode the QCIF video at **16.5** fps. The IP CTRL block needs **645** ALMs, the GET blocks **5 363** ALMs, and the SAD PARALLEL block **2 256** ALMs on the Cyclone V. The combined area is **8 264** ALMs. Compared to the area of the Accelerator IV, the area for this design is 2 392 ALMs less. The design is able to encode the video **2.5x** faster compared to the CPU only version which was able to encode the video at **6.5** fps. Although the Accelerator IV is reported to improve the performance by 3.0x, the improvement with the Accelerator V is still better, as the ARM CPU and the memory on the CycloneV SoC are much faster compared to the CPU and memory speed on the ArriaII board.

Table 6.1 and Figure 6.6 show the improved results of the design seen in Figure 6.5. Table 6.1 tabulates the CPU only results on the left side and the acceler-

Table 6.1: Most time consuming functions of Kvazaar in percentages

Full intra search (CPU only)		Full intra search (Accelerator V)	
%	Functions	%	Functions
39.23	intra_get_angular_pred	4.25	intra_get_angular_pred
10.93	quant	20.70	quant
8.01	sort_modes	-	HW Accelerated
4.26	sad_8bit_32x32_generic	-	HW Accelerated
4.17	sad_8bit_16x16_generic	-	HW Accelerated
4.01	sad_8bit_8x8_generic	-	HW Accelerated
3.09	quantize_residual	8.83	quantize_residual
2.84	intra_get_pred	0.30	intra_get_pred
2.17	search_intra_rough	4.93	search_intra_rough
1.59	partial_butterfly_16	4.96	partial_butterfly_16
1.42	intra_build_reference_border	3.22	intra_build_reference_border
1.34	intra_get_planar_pred	0.57	intra_get_planar_pred
1.25	sad_8bit_4x4_generic	-	HW Accelerated
1.25	partial_butterfly_32	7.09	partial_butterfly_32
1.25	partial_butterfly_inverse_32	5.46	partial_butterfly_inverse_32
0.83	dequant	3.01	dequant
0.83	partial_butterfly_inverse_16	4.71	partial_butterfly_inverse_16
0.75	intra_pred_ratecost	-	HW Accelerated
0.75	partial_butterfly_8	1.63	partial_butterfly_8
0.67	intra_recon	1.75	intra_recon
0.42	intra_filter	1.70	intra_filter
0.42	intra_recon_lcu_luma	0.46	intra_recon_lcu_luma
0.33	search_cu_intra	1.74	search_cu_intra
0.33	fast_forward_dst	0.60	fast_forward_dst
0.33	transformskip	0.46	transformskip
0.25	partial_butterfly_4	0.64	partial_butterfly_4
0.25	partial_butterfly_inverse_8	1.42	partial_butterfly_inverse_8
0.25	partial_butterfly_inverse_4	0.04	partial_butterfly_inverse_4
0.17	intra_get_dc_pred	0.07	intra_get_dc_pred
0.17	fast_inverse_dst	0.28	fast_inverse_dst
0.08	transform2d	0.57	transform2d
0.00	itransform2d	0.14	itransform2d
0.00	intra_recon_lcu_chroma	0.62	intra_recon_lcu_chroma

ated results on the right side. CPU only functions that are colored with two colors are used by both intra prediction and reconstruction. However, functions like *intra_get_angular_pred* on the right, are mono colored as the whole intra prediction is offloaded to the FPGA and these functions are only used by reconstruction.

From Table 6.1 it can also be seen that *search_intra_rough* is the only intra prediction function run on software, as intra prediction, result sorting, and SAD calculation are offloaded to the FPGA. The overall improvement to intra prediction

Table 6.2: Comparing search_intra_rough with different block sizes fully on CPU and with Accelerator V @100 MHz

Block size	Count	CPU (s)	Accelerator V (s)	Improvement
4x4	356820	11.190	1.519	7.37x
8x8	161880	12.550	0.831	15.10x
16x16	57960	14.050	0.453	31.02x
32x32	17600	15.370	0.327	47.00x
TOT	594260	53.160	3.130	16.98x

can be seen in Figure 6.6. It shows the time usage diagram of both CPU only and FPGA accelerated Kvazaar. In the CPU only Kvazaar, the intra prediction accounts for 66,24% and in the FPGA accelerated Kvazaar the respective percentage is only 4.93%. Hence, the improvement is 13x. Table 6.2 shows the actual time used in *search_intra_rough* in both CPU only and FPGA accelerated.

6.7 Accelerator VI: Multiple pixel prediction

Although intra prediction is no longer the bottleneck, accelerating it further is still necessary. As the next step would be to offload reconstruction functions (dct, inverse dct, quantization, dequantization) and actual intra search to the FPGA, intra prediction is foreseen to again become the bottleneck of the system. To make intra prediction faster, it is possible to predict multiple pixels at a time with minimal changes to the code.

6.7.1 Design

The faster Catapult-C top level function for GET POS is illustrated in Listing 6.5. The difference between this version and the one predicting only one pixel at a time is the number of *src* arrays. The *src* arrays are mapped to on-chip memories to save area. If they were registers the resulting area would at least double. Reading and writing to on-chip memories takes one cycle each. For this reason, there are two separate arrays for the reference pixels located above (*src1* and *src3*) and left (*src2* and *src4*). This way the accelerator has enough memory bandwidth without the increase in latency or area.

The Catapult-C code for predicting two pixels at a time with GET POS blocks is illustrated in Listing 6.6. The *get_ang_pos* function is called from the top level, that passes two sets of reference pixel arrays to the function. The function produces two predicted pixels in parallel, and writes them to the output, which is twice the size from the previous version. This halves the time used in the inner loop, and thus almost halves the entire time of the prediction.

The block diagram is almost the same as in 6.5. The only difference being the data width from GET blocks to the SAD is increased in this version from $8+2$ bits to $16+2$ bits.

6.7.2 Performance

Table 6.3: Comparing intra prediction with different block sizes fully on CPU and with Accelerator VI @100 MHz

Block size	Count	CPU (s)	Accelerator VI (s)	Improvement
4x4	356820	11.190	1.416	7.90x
8x8	161880	12.550	0.691	18.16x
16x16	57960	14.050	0.318	44.18x
32x32	17600	15.370	0.182	84.45x
TOT	594260	53.160	2.607	20.39x

With the Accelerator VI, the design is able to encode the QCIF video at **16.8** fps and a HD video at **0.7** fps. The IP CTRL block takes **645** ALMs, the GET blocks **7 229** ALMs, and the SAD PARALLEL block **2 941** ALMs on the Cyclone V. The combined area is **10 815** ALMs. The frame rate improvement with the Accelerator VI only produces a minimal increase in speed compared to the result of Accelerator V, but that was expected as explained in Section 6.7.

As is shown in Table 6.3, halving the intra prediction and SAD calculation time does not double the performance in all block sizes. The time used for the 4x4 blocks with Accelerator VI is almost identical to Accelerator V seen in Table 6.2 (only

```
#pragma hls_design top
void get_ang_pos(ac_channel<uint_16> &data_in,
                 ac_channel<uint_16> &data_out)
{
  uint_8 width = 0,a = 0,mode = 0,bytes = 0;
  one_bit mode_ver;
  pixel  src1[65],src2[65],src3[65],src4[65];
  width = data_in.read();
  mode = data_in.read();
  mode_ver = data_in.read();
  bytes = 2*width;
  for(a = 0; a < 65;a++){
    uint_16 temp = data_in.read();
    src1[a] = temp.slc<8>(0);
    src2[a] = temp.slc<8>(8);
    src3[a] = temp.slc<8>(0);
    src4[a] = temp.slc<8>(8);
    if(a == bytes)break;
  }
  ang_pos_pred(src1,src2,src3,src4,data_out,width,mode,mode_ver);
}
```

Listing 6.5: Catapult-C code for GET POS top level

1.07x improvement). In the case of 32x32 blocks, the respective improvement was 1.8x. The suggested reasons for this are the sheer number of 4x4 calculations and the time used for 4x4 predictions. The overhead from the function call, the system calls, sending of data, and actually starting the HW prediction hinders the improvement got in the 4x4 predictions with the Accelerator VI. In the case of 32x32 blocks, the

```
void ang_pos_pred(pixel* src1,pixel* src2,pixel* src3,pixel* src4,
                  ac_channel<uint_16> &data_out,uint_8 cu_width,
                  uint_8 dir_mode,one_bit mode_ver)
{
...
  ref_main = mode_ver ? src1 : src2;
  ref_main2 = mode_ver ? src3 : src4;
  data_out.write(mode_ver);
  ac_int<12,true> delta_pos=0;
  ac_int<7,true> delta_int,delta_fract,minus_delta_fract,main_index;
  for (y = 0; y < 32; y++){
    delta_pos  += abs_ang;
    delta_int   = delta_pos >> 5;
    delta_fract = delta_pos & (32 - 1);
    minus_delta_fract = (32 - delta_fract);
    for (x = 0; x < 32; x++){
      if(delta_fract){
        // Pixel one
        main_index = x + delta_int + 1;
        pred = (minus_delta_fract*ref_main[main_index] +
                  delta_fract*ref_main[main_index+1] + 16) >> 5;
        output_temp.set_slc(0,pred);
        // Pixel two
        x++;
        main_index = x + delta_int + 1;
        pred = (minus_delta_fract*ref_main2[main_index] +
              delta_fract*ref_main2[main_index+1]+16) >> 5;
        output_temp.set_slc(8,pred);
      }
      else{
        // Pixel one
        pred = ref_main[x + delta_int + 1];
        output_temp.set_slc(0,pred);
        x++;
        // Pixel two
        pred = ref_main2[x + delta_int + 1];
        output_temp.set_slc(8,pred);
      }
      // Write pixels out
      data_out.write(output_temp);
      if(x == cu_width-1) break;
    }
    if(y == cu_width-1) break;
  }
}
```

Listing 6.6: Catapult-C code for GET POS fuction

time used in the HW is clearly more than the communication time and the related overhead.

6.8 Accelerator VII: Optimized implementation of Accelerator VI

According to Table 6.3, the biggest problems with Accelerator VI are the 4x4 blocks, so further analysis was done to identify the issues and solving them. First, a series of simulations were made to identify possible bottlenecks in the HW. Most of the slowness at this point is most likely caused by the software overhead, but it does not explain the lack of improvement in the smaller block sizes. From previous results it is known that the time used in the predicting blocks is minimal. So IP CTRL block and the SAD PARALLEL block are taken into closer observation.

6.8.1 Design

The simulation results of the SAD PARALLEL block show that the search for the minimum SAD value takes 35 cycles for every block size. That is a huge part of the

```
//CONFIG
...
//SAD CALCULATIONS
...
for(y = 0; y < 35;y++){
  uint_8 ratecost = 5;
  ac_int<18,false> cost_temp = 0;
  if(candidates[0] == -1){
    ratecost = 0;
  }
  if(candidates[0] == y){
    ratecost = 1;
  }
  else if(candidates[1] == y || candidates[2] == y){
    ratecost = 2;
  }
  cost_temp = sad[y]+ratecost*lambda;
  if(cost_temp < best_sad){
    best_sad = cost_temp;
    best_modecost = ratecost;
    sad_index = y;
  }
  sads[y] = cost_temp;
}
sads[35] = sad_index;
sads[36] = best_modecost;
*irq = 1;
```

Listing 6.7: Calculating the cost in SAD PARALLEL

overall time of 4x4 blocks. In comparison, predicting 16 pixels (two at a time) takes 8 cycles. Listing 6.7 describes the process of finding the minimum SAD value and calculating the cost of that mode. The cost is calculated by using a *lambda* value and by the surrounding modes of the current predicted block. The lambda value is obtained from the quantization parameter and the surrounding modes from the *candidates* array. The surrounding predictions affect the choosing of the best mode in cases where there are minimal differences between the SADs. Encoding the block with a same mode as the surrounding CUs saves bits and thus lowers the bitrate.

In Listing 6.7 the whole process of finding the best SAD is done after the prediction and SAD calculations. The for-loop could be unrolled, but that would lead to a significant increase in area, because there would be a need for 35 separate multipliers. Hence this part of the code is impossible to make faster by exploring Catapult-C project settings. The only solution is to change the structure of the code.

```
template<int N> struct min_s {
template<typename T> static T min(T *a,ac_int<6,false> index,
                                  ac_int<6,false> *best_index)
{
  ac_int<6,false> i0,i1;
  T m0 = min_s<N/2>::min(a,index,&i0);
  T m1 = min_s<N-N/2>::min(a + N/2,index+N/2,&i1);
  if(m0 <= m1){
    *best_index = i0;
    return m0;
  }
  else{
    *best_index = i1;
    return m1;
  }
}
};

template<> struct min_s<1> {
template<typename T> static T min(T *a,ac_int<6,false> index,
                                  ac_int<6,false> *best_index)
{
    *best_index = index;
    return a[0];
}
};

template<int N, typename T> T min(T *a,ac_int<6,false> *best_index){
return min_s<N>::min(a,0,best_index);
}
```

Listing 6.8: Template recursion code used to generate a balanced comparison tree

Listing 6.8 shows a template recursion [6, p 138] that implements the same search for the best SAD as seen in Listing 6.7. The template recursion is inlined during the compilation and it results into a balanced comparison tree. The for-loop in Listing 6.7 has a comparison dependency to the previous *best_sad* value. It results in a long chain of operations and to a multi-cycle for-loop with even small iterations. In Listing 6.8, the `min` function is a template function that calls the template function. A series of recursion calls start from the value N, according to the first template call to `min`. N is halved every recursion call until N=1, after which the default `template<> struct min_s<1>` is called.

Listing 6.9 illustrates the changes to the structure of the best mode search. The *sad* array is now initialized with the *ratecost* compared to calculating the *ratecost* in real-time in Listing 6.7. This way the array initialization loop can be unrolled without a huge increase in area, as the multiplication is done outside the loop and there is no need for 35 separate multipliers. The ratecost for modes affected by the surrounding CUs are calculated after the loop. The SAD PARALLEL block is able to initialize the *sad* array after the configuration from the IP CTRL block and before the prediction blocks start sending data to the SAD PARALLEL block.

Changes were also made to the retrieval of the reference pixels in the IP CTRL block. Instead of loading the reference pixels to an internal memory structure, and filtering and sending the data afterwards, it now loads the reference pixels and

```
//CONFIG
...
lambda = lambda*5;
for(y = 0; y < 35;y++){
  sad[y] = lambda;
}
sad[candidates[0]] = lambda;
sad[candidates[1]] = 2*lambda;
sad[candidates[2]] = 2*lambda;
...
//SAD CALCULATIONS
..
best_sad = min<35>(sad,&best_index);
ac_int<3,false> ratecost = 5;
if(candidates[0] == -1){
  ratecost = 0;
}
if(candidates[0] == best_index){
  ratecost = 1;
}
else if(candidates[1] == best_index || candidates[2] == best_index){
  ratecost = 2;
}
sads[best_index] = best_sad;
sads[35] = best_index;
sads[36] = ratecost;
```
Listing 6.9: Optimized calculation of cost in SAD PARALLEL

Table 6.4: Comparing search_intra_rough with different block sizes fully on CPU and with Accelerator VII @100 MHz

Block size	Count	CPU (s)	Accelerator VII (s)	Improvement
4x4	356820	11.190	0.973	11.50x
8x8	161880	12.550	0.489	25.66x
16x16	57960	14.050	0.237	59.28x
32x32	17600	15.370	0.161	95.47x
TOT	594260	53.160	1.860	28.58x

Table 6.5: Comparing the cycles used in Accelerator V and Accelerator VII

Block size	Accelerator V (cycles)	Accelerator VII (cycles)	Improvement
4x4	159	40	3.98x
8x8	243	68	3.57x
16x16	503	172	2.92x
32x32	1403	572	2.45x

calculates the filtered pixels at the same time without unnecessary temporary data structures. The data width from IP CTRL to GET blocks was also increased from 16+2 to 32+2 in order to send more reference pixels per cycle. Reason why this was not made before, was to first get a working version with readable code. HLS suits for this kind of work, getting a working version fast and with little ease, and then modifying the code afterwards for more functionality or improved performance and just regenerating the RTL again. And as long as the interfacing works the same way nothing else needs changes e.g. software code or other blocks.

6.8.2 Performance

After the optimizations, SignalTapII, which is part of QuartusII FPGA design software, was used to get cycle accurate profiling of the accelerator. The IP ACC time consumption for 4x4 blocks is divided into following parts: IP CTRL and SAD PRALLEL configuration (14 cycles); Receiving, filtering and sending the reference pixels to the prediction blocks (7 cycles); Actual prediction and sad calculation (13 cycles); The search for the lowest mode cost and saving the results to the on-chip memory (6 cycles). In the Accelerator VI, reading and sending the reference pixels in the IP CTRL block takes 9+9=18 cycles, resulting in **2.6x** improvement over the Accelerator VII. Finding the minimum cost in the SAD PARALLEL block in the Accelerator VI takes 37 cycles and in Accelerator VII 6 cycles, resulting in a **6.17x** improvement. The whole process for the 4x4 blocks takes 40 cycles with the Accelerator VII resulting in **2.05x** improvement over the Accelerator VI.

As reported in Table 6.4, Accelerator VII processed 4x4 blocks **1.56x** faster than the Accelerator V and **1.46x** faster than the Accelerator VI. In conclusion, the

Table 6.6: Comparing the time usage of Accelerator V and Accelerator VII for one HD frame @125 MHz

Block size	Count	Accelerator V (s)	Accelerator VII (s)	Improvement
4x4	356820	0.454	0.114	3.98x
8x8	161880	0.315	0.088	3.57x
16x16	57690	0.233	0.080	2.92x
32x32	17600	0.198	0.080	2.45x
Total		1.2 (16.7 fps)	0.362 (55.2 fps)	3.31x

optimizations really speed up the processing of 4x4 blocks. With 32x32 blocks, the improvement is **2.03x** compared to Accelerator V. With the Accelerator VII, the design is able to encode the QCIF video at **18.0** fps and a HD video at **0.74** fps. The IP CTRL block takes **919** ALMs, the GET blocks **7 581** ALMs, and the SAD PARALLEL **3 162** ALMs on Cyclone V. The combined area is **11 662** ALMs. According to Tables 6.5 and 6.6 the average improvement from the Accelerator VI to the Accelerator VII is **3.31x**. The Accelerator VII can perform the prediction and mode selection for HD video at **55.2** fps.

7. ANALYSIS

Table 7.1 summarizes the results of the accelerators. This chapter presents analyzes of the different accelerator versions and compares the time usage in the HLS design flow to traditional RTL flow.

Table 7.1: Results of all development versions

Accelerator	Features	Board	ALMs	Area % of total	QCIF fps
I	Angular prediction modes	Cyclone II	924	7.4	0.109
II	Angular prediction modes with mode cost computation	Cyclone II	1246	10.0	0.131
III	All prediction modes with mode cost computation and selection	Cyclone II	2380	19.1	0.170
IV	Parallel implementation of Accelerator III	Arria II	10 656	22.8	0.472
V	Integrating the Accelerator IV to ARM	Cyclone V	8 264	19.9	16.50
VI	Multiple pixel prediction	Cyclone V	10 815	26.1	16.77
VII	Optimized implementation of Accelerator VI	Cyclone V	11 662	28.1	18.03

7.1 Performance

First the Accelerator I with angular prediction modes was created. The angular prediction is the most demanding function of Kvazaar. Before any acceleration it takes over 39% of the overall encoding time. With Accelerator I the overall time consumption of intra prediction decreased from 66.24% to 43.62%. The NiosII processor alone was able to encode the video 0.065 fps on the Cyclone II FPGA.

The Accelerator II has mode cost computation or SAD calculation added to the Accelerator I. None of the SAD calculation function are the next most demanding functions in terms of time usage by them selves, but combined they took 13.69%. Adding the SAD calculation is more natural than e.g. quantization to get a more coherent implementation. With Accelerator II the overall time consumption of intra prediction further decreased to 32.24%.

Next the rest of the prediction algorithms, planar and DC, and mode selection were added to the Accelerator III, creating an accelerator that is able to perform the same function as the *intra_rough_search*. This meant that *sort_modes* function, which used the third most overall time of the encoder, was also offloaded to the FPGA. With Accelerator III the overall time consumption of intra prediction further decreased to 22.09%.

At the next phase the Accelerator III was re-implemented to work in parallel. All the 35 prediction modes are calculated at the same time and the SAD value is also calculated in parallel. The NiosII processor alone was able to encode the video at 0.065 fps on the Arria II FPGA.

After getting the support from Catapult-C for more FPGA chips, the Cyclone V SoC FPGA was taken into use. This meant the integration of the Accelerator IV to the ARM interface, which included an implementation for a DMA and a kernel driver for the HW. With Accelerator V the overall time consumption of intra prediction was down to 4.93%. The ARM processor alone was able to encode the video 6.52 fps on the Cyclone V SoC FPGA.

To further show the ease of using Catapult-C, the Accelerator V was further accelerated. The Accelerator VI has modified predicting blocks that are able to predict two pixels at a time, supposedly halving the time used compared to Accelerator V. After further inspection, the acceleration time did not halve as first thought.

More work was done to optimize the Accelerator VI. Optimizations included receiving data from the DMAs faster, sending data to the prediction blocks faster and sorting the SAD values faster. These improvements lowered the overhead of data transfers and calculations compared to the prediction, more than doubling the speed of the Accelerator VII compared to the Accelerator VI. The final version was able to encode the QCIF video at 18.03 fps as seen in Table 7.1.

The Accelerator V was able to perform intra prediction and mode selection for HD video 16.7 fps using 8 265 ALMs and the Accelerator VII was able to do the same **55.2** fps using **11 662** ALMs. So the improvement was 3.31x but the area increase was only 1.41x.

7.2 Area

With all the accelerator versions, speed was the first main criterion before area. Area of the accelerator was optimized at the cost of speed if the speed decrease compared to the area saving was minimal. From Table 7.1 can be seen that the Accelerator I takes only 7.4% of the Cyclone II, due to it being only part of the whole intra prediction. The area percentage of the different accelerators vary depending on the different sized FPGA chips.

The reason why the Accelerator VII does not use the whole capacity of the FPGA chip is that the purpose of the final accelerator was to become as fast as possible, but still have minimal area cost. The future purpose of the Accelerator VII is to become a part of a bigger system, where rest of the area is needed for other components. The whole area of the Cyclone V, can still be utilized by adding more instances of Accelerator VII, or by increasing the number of pixels predicted in the prediction blocks.

7.3 Comparison to related work

The implementation in [23], which is able to predict 17.5 Full HD frames per second and takes 31 179 ALUTs (15 589 ALMs) or 33.3% of ArriaII. The final version of the accelerator done in this Thesis can predict **24.5** Full HD frames per second and takes **11 662** ALMs or 28.1% of Cyclone V. The 24.5 fps result for Full HD video is gotten by scaling the result for HD video, with resolution as the factor $(55.2fps/(1920 * 1080/1280 * 720))$. So in comparison the accelerator implemented in this Thesis takes less area and is faster than [23]. In addition the accelerator presented in this Thesis implements the SAD calculations, which [23] does not. The final optimized version is also **2.25x** faster compared to the same accelerator presented in [22].

The 24.5 fps result was achieved with a specific video sequence. Depending on the sequence the performance might vary. Because the time used for intra search varies between LCUs and frames, the maximum number of CUs searched in a CTU for Full HD can be calculated. A CTU has four 32x32 blocks, 16 16x16 blocks, 64 8x8 blocks and 256 4x4 blocks, so a Full HD frame has 506 CTUs. So the maximum number of intra predictions in a Full HD frame is $(4+16+64+256)*506*35 = 6021400$. With the worst case scenario the presented intra prediction accelerator can predict 12.5 fps. The sequence or testing environment for the accelerator in [23] is not known. But if the SAD calculation is taken off from the Accelerator VII and another accelerator is added to work in parallel with the other one, the accelerators combined can achieve **25** fps using **17 002** ALMs or 41% of Cyclone V and thus is still faster than in [23] and uses only slightly more ALMs. The absolute performance of the system is not

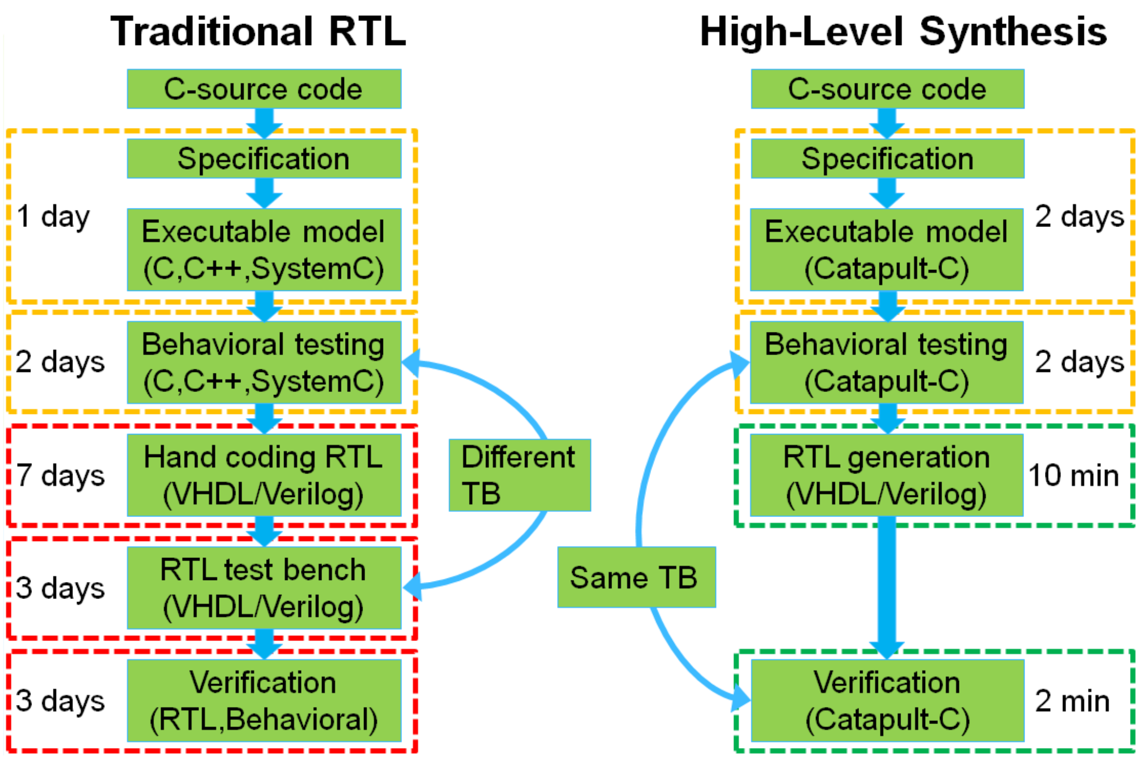
Traditional RTL
C-source code
Specification
1 day
Executable model (C,C++,SystemC)
2 days
Behavioral testing (C,C++,SystemC)
7 days
Hand coding RTL (VHDL/Verilog)
3 days
RTL test bench (VHDL/Verilog)
3 days
Verification (RTL,Behavioral)
Different TB
Same TB
High-Level Synthesis
C-source code
Specification
Executable model (Catapult-C)
2 days
Behavioral testing (Catapult-C)
2 days
RTL generation (VHDL/Verilog)
10 min
Verification (Catapult-C)
2 min

Figure 7.1: Comparison of the time usage in the traditional RTL design flow and the HLS design flow for the SAD PARALLEL block

near the performance of the accelerator, as the CPU is hindering the overall fps. The purpose of this Thesis was to research the HLS design flow and the scalability, and not to get the maximum performance for the whole encoding process.

7.4 Development time

HLS and Catapult-C has an reasonable learning curve compared to traditional RTL design. Learning an RTL language from scratch takes time and practice to perfect. With HLS, the the language is usually not the problem, as users are already familiar with C or C++. With HLS and Catapult-C, time is spent for learning the tool itself and the slightly different way of writing the HW oriented C-code.

The time to learn the basics of Catapult-C took **1 day** with the included finite impulse response filter tutorial. Using the H.263 proof of concept done in this Thesis as a reference, a first HLS implementation with some complexity, took **one month**. After some experience with Catapult-C, re-design with similar complexity is estimated to take **less than a week**. Most of the time used in the first implementation was learning the tool flow following the RTL generation.

Figure 7.1 presents the time used in the SAD PARALLEL block in Accelerator VII. The figure compares the estimated traditional RTL times to the times it took in HLS. The time used for the specification and the execution model based on the C-source code takes more time with the HLS design flow than with the traditional RTL. This is because the execution model in HLS is written more precisely and optimized for RTL generation. The difference between the two flows should still not be too significant, if both of the executable models have the same overall functionality. The time used in testing the executable models is the same. The testbenches should not differ too much. The major difference in time usage comes after the behavioral testing. Using the SAD PARALLEL block as an example, it takes 10 minutes to generate the RTL code for it. The time for manually writing the RTL code is estimated to take 7 days. HLS also saves time in the RTL verification, because the behavioral testbench is re-used in the RTL verification. With traditional RTL the testbench is usually done in the same language as the implementation, or for example in SystemVerilog, nevertheless, the testbench is re-written for the RTL. In HLS, the RTL verification usually passes with the first try, if the behavioral testing has passed. For example, errors that might happen in the HLS verification are due to the use of bit accurate types, but these are rare and easy to fix. With traditional RTL, both the implementation and the testbench can have several errors making the verification cumbersome.

To summarize, HLS was proofed to decrease the accelerator design and implementation time significantly compared to traditional RTL. As a rule of thumb, one month in RTL is decreased to one week in HLS.

8. CONCLUSION

The main goal of this Thesis was to use Catapult-C HLS tool for creating an HEVC intra prediction accelerator for an FPGA faster than could be done with traditional RTL coding. The accelerator was synthesized for an FPGA and run in real-time on an FPGA development board. Several boards were used during the Thesis, as the size of the accelerator grew during the process. The final FPGA board used was a Cyclone V SoC FPGA with a dual-core ARM processor integrated to the FPGA.

The power and the ease of HLS was exploited in this Thesis. A simple accelerator was created at first to get familiar with the Kvazaar and Catapult-C, after which more features were gradually added to the accelerator. The features of a new development version were selected after profiling Kvazaar with the previous development version.

The goal of using HLS to create RTL for an FPGA was achieved and the end results were relatively good. The resulting intra prediction accelerator for Kvazaar HEVC intra encoder achieved better results compared to the related work.

As future work, the Accelerator VII can still be further accelerated ,e.g. for 4k video resolution, by increasing the data width of receiving and sending of the reference pixels, and by predicting even more pixels at a time. Other work focusing on increasing the performance, would need offloading more functions to the FPGA, e.g. reconstruction functions *dct*, *inverse dct*, *quantization* and *dequantization*. The best results would be achieved by implementing the entire CTU search on the FPGA leaving only the *file IO*, *data control* and *entropy encoding* to the CPU.

REFERENCES

[1] J. Vanne, M. Viitanen, T.D. Hämäläinen, and A. Hallapuro, "Comparative rate-distortion-complexity analysis of hevc and avc video codecs," *Circuits and Systems for Video Technology, IEEE Transactions on*, vol. 22, pp. 1885–1898, Dec 2012.

[2] V. Sze, M. Budagavi, and G. J. Sullivan, *High Efficiency Video Coding (HEVC)*. Springer, 2014.

[3] x265, "x265." `http://x265.org/`, 2015. [WWW], Accessed 04.09.2015.

[4] Ultra Video Group, "Kvazaar HEVC Encoder." `https://github.com/ultravideo/kvazaar`, 2015. [WWW], Accessed 08.06.2015.

[5] f265, "f265." `http://f265.org/`, 2015. [WWW], Accessed 04.09.2015.

[6] M. Fingeroff, *High-Level Synthesis Blue Book*. xlibris corporation, 2010.

[7] Calypto, "Catapult: Product Family Overview." `http://calypto.com/en/products/catapult/overview/`, 2015. [WWW], Accessed 03.06.2015.

[8] Cadence, "Cynthesizer Solution." `http://www.cadence.com/products/sd/cynthesizer/pages/default.aspx?CMP=MOSS1/`, 2015. [WWW], Accessed 07.09.2015.

[9] Cadence, "C-to-Silicon Compiler." `http://www.cadence.com/products/sd/silicon_compiler/pages/default.aspx`, 2015. [WWW], Accessed 07.09.2015.

[10] Xilinx, "Vivado High-Level Synthesis." `http://www.xilinx.com/products/design-tools/vivado/integration/esl-design.html`, 2015. [WWW], Accessed 07.09.2015.

[11] Synopsys, "Synphony C Compiler." `http://www.synopsys.com/Tools/Implementation/RTLSynthesis/Pages/SynphonyC-Compiler.aspx`, 2015. [WWW], Accessed 07.09.2015.

[12] M. Graphics, "ModelSim." `https://www.mentor.com/products/fpga/model/`, 2015. [WWW], Accessed 19.10.2015.

[13] M. Graphics, "Precision RTL." `https://www.mentor.com/products/fpga/synthesis/precision_rtl_plus/`, 2015. [WWW], Accessed 19.10.2015.

[14] Xilinx, “What is an FPGA?.” http://www.xilinx.com/training/fpga/fpga-field-programmable-gate-array.htm, 2015. [WWW], Accessed 03.06.2015.

[15] Altera, “DE2 Development and Education Board.” http://wl.altera.com/education/univ/materials/boards/de2/unv-de2-board.html, 2015. [WWW], Accessed 03.06.2015.

[16] Altera, “Arria II GX FPGA Development Kit.” https://www.altera.com/products/boards_and_kits/dev-kits/altera/kit-aiigx-pcie.html, 2015. [WWW], Accessed 03.06.2015.

[17] Terasic, “VEEK-MT-C5SoC.” http://www.terasic.com.tw/cgi-bin/page/archive.pl?Language=English&No=828, 2015. [WWW], Accessed 03.06.2015.

[18] ARM, “ARM.” http://www.arm.com/, 2015. [WWW], Accessed 04.09.2015.

[19] Altera, “Cyclone V Device Handbook.” https://www.altera.com/content/dam/altera-www/global/en_US/pdfs/literature/hb/cyclone-v/cyclone5_handbook.pdf, 2015. [WWW], Accessed 15.06.2015.

[20] Altera, “Altera SoCs.” https://www.altera.com/products/soc/overview.html, 2015. [WWW], Accessed 12.06.2015.

[21] A. Chabouk Jokhadar and C. Gomez Gonzalez, “High level synthesis for design of video processing blocks,” 2015. MSc Thesis.

[22] P. Sjövall, J. Virtanen, J. Vanne, and T. D. Hämäläinen, “High level synthesis design flow for hevc intra encoder on soc-fpga,” in *Euromicro Conference on Digital System Design (DSD), 2015*, Aug 2015.

[23] A. Abramowski and G. Pastuszak, “A double-path intra prediction architecture for the hardware h.265/hevc encoder,” in *Design and Diagnostics of Electronic Circuits Systems, 17th International Symposium on*, pp. 27–32, April 2014.

[24] K. Miyazawa, H. Sakate, S.-I. Sekiguchi, N. Motoyama, Y. Sugito, K. Iguchi, A. Ichigaya, and S.-I. Sakaida, “Real-time hardware implementation of hevc video encoder for 1080p hd video,” in *Picture Coding Symposium (PCS), 2013*, pp. 225–228, Dec 2013.

[25] S. Kim, H. Kim, T. Chung, and J.-G. Kim, “Design of h.264 video encoder with c to rtl design tool,” in *SoC Design Conference (ISOCC), 2012 International*, pp. 171–174, Nov 2012.

[26] T. Damak, I. Werda, N. Masmoudi, and S. Bilavarn, “Fast prototyping h.264 deblocking filter using esl tools,” in *Systems, Signals and Devices (SSD), 2011 8th International Multi-Conference on*, pp. 1–4, March 2011.

[27] Altera, “Nios II Processor.” `https://www.altera.com/products/processors/overview.html`, 2015. [WWW], Accessed 16.06.2015.

[28] Altera, “Quartus II software.” `https://www.altera.com/products/design-software/fpga-design/quartus-ii/overview.html`, 2015. [WWW], Accessed 19.10.2015.

[29] Xiph, “Xiph.org Video Test Media.” `https://media.xiph.org/video/derf/`, 2015. [WWW], Accessed 24.11.2015.

[30] J. Fenlason, “GNU gprof manual.” `https://sourceware.org/binutils/docs/gprof/`, 2008. [WWW], Accessed 14.08.2015.

[31] J. Fonseca, “Gprof2dot.” `https://github.com/jrfonseca/gprof2dot`, 2015. [WWW], Accessed 17.08.2015.

[32] F. Bossen, “Common Test Conditions and Software Reference Configurations.” document JCTVC-H1100, JCT-VC, San Jose, CA, Feb. 2012.