TAMPEREEN TEKNILLINEN YLIOPISTO
TAMPERE UNIVERSITY OF TECHNOLOGY

VILI VIITAMÄKI

HIGH-LEVEL SYNTHESIS IMPLEMENTATION OF HEVC INTRA ENCODER

Master of Science Thesis

Examiner: Ass. Prof. Jarno Vanne
Examiner and topic approved by the Council of the Faculty of Computing and Electrical Engineering on 28th March 2018

ABSTRACT

VILI VIITAMÄKI: High-Level Synthesis Implementation of HEVC Intra Encoder
Tampere University of Technology
Master of Science Thesis, pages 45
October 2018
Master's Degree Programme in Information Technology
Major: Embedded Systems
Examiner: Assistant Professor Jarno Vanne

Keywords: High Efficiency Video Coding (HEVC), High-Level Synthesis (HLS), Intra coding, Video encoder, Field programmable gate array (FPGA)

High Efficiency Video Coding (HEVC) is the latest video coding standard that aims to alleviate the increasing transmission and storage needs of modern video applications. Compared with its predecessor, HEVC is able to halve the bit rate required for high quality video, but at the cost of increased complexity. High complexity makes HEVC video encoding slow and resource intensive but also ideal for hardware acceleration.

With increasingly more complex designs, the effort required for traditional hardware development at register-transfer level (RTL) grows substantially. High-Level Synthesis (HLS) aims to solve this by raising the abstraction level through automatic tools that generate RTL-level code from general programming languages like C or C++.

In this Thesis, we made use of Catapult-C HLS tool to create an intra coding accelerator for an HEVC encoder on a Field Programmable Gate Array (FPGA). We used the C source code of Kvazaar open-source HEVC encoder as a reference model for accelerator implementation. Over 90 % of the implementation including all major intra coding tools were implemented with HLS, with the rest being ready made IP blocks and hand-written RTL components.

The accelerator was synthesized into an Arria 10 FPGA chip that was able to accommodate three accelerators and associated interface components. With two FPGAs connected to a high-end PC, our encoder was able to encode 2160p Ultra-High definition (UHD) video at 123 fps. Total FPGA resource usage was around 80 % with 346k Adaptive logic modules (ALMs) and 1227 Digital signal processors (DSPs).

TIIVISTELMÄ

Vili VIITAMÄKI: HEVC intra kooderin toteuttaminen korkean tason synteesillä
Tampereen teknillinen yliopisto
Diplomityö, 45 sivua
Lokakuu 2018
Tietotekniikan DI-tutkinto-ohjelma
Pääaine: Sulautetut Järjestelmät
Tarkastaja: Apulaisprofessori Jarno Vanne

Avainsanat: High Efficiency Video Coding (HEVC), High-Level Synthesis (HLS), Intra koodaus, Videonpakkaus, Field programmable gate array (FPGA)

High Efficiency Video Coding (HEVC) on viimeisin videonpakkausstandardi, joka on kehitetty uusien multimediasovellusten kasvaviin tarpeisiin. HEVC:n tavoitteena on puolittaa videon tarvitsema bittinopeus edeltävään videonpakkausstandardiin verrattuna heikentämättä kuvanlaatua. HEVC:n suuri kompleksisuus tekee HEVC videon pakkaamisesta käytännön sovelluksissa hidasta ja resursseja vaativaa. Tästä johtuen HEVC on hyvä kandidaatti rautakiihdytettäväksi.

Entistä monimutkaisempien ja kasvavien mallien vuoksi perinteinen RTL (Register-Transfer Level) tason laitteistokehitys on yhä vaativampaa. High-Level Synthesis (HLS) -työkalut yrittävät ratkaista tämän nostamalla laitteistokehityksen abstraktiotasoa ja automatisoimalla RTL-tason koodin generoinnin korkeamman tason mallista, kuten esimerkiksi C ja C++ koodista.

Tässä työssä käytettiin Catapult-C HLS -työkalua luomaan ohjelmoitavalle logiikkapiirille (FPGA) kiihdytin, joka nopeuttaa HEVC:n intra pakkausta. Lähtökohtana työssä käytettiin Kvazaar HEVC kooderin avointa C kielistä lähdekoodia. Kaikki merkittävimmät HEVC:n koodaustyökalut toteutettiin HLS-työkalua käyttäen niin, että FPGA:lla kyetään suorittamaan kaikki raskas laskenta.

Toteutettu kiihdytin syntesoitiin Arria 10 FPGA piirille. Tälle piirille oli mahdollista sisällyttää kolme kiihdytintä rajapintakomponenttien lisäksi. Järjestelmää testattiin käyttämällä kahta FPGA-korttia ja yhteensä kuutta kiihdytintä tehokkaassa testikoneessa. Tällä kokoonpanoilla saavutimme pakkausnopeudeksi 123 kuvaa sekunnissa, joka on melkein seitsemän kertaa nopeampi kuin kiihdyttämätön järjestelmä. Kolmella kiihdyttimellä yhden FPGA:n resurssikäyttö oli noin 80 %, sisältäen 346k logiikkaelementtiä ja 1227 signaaliprosessoria.

PREFACE

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

I want to thank my examiner Jarno Vanne for providing me the opportunity to work in the university and for guidance during and before this Thesis. I would also like to thank Panu Sjövall and all others who help me during this Thesis.

Tampere, 22.10.2018

Vili Viitamäki

TABLE OF CONTENTS

LIST OF ABBREVIATIONS AND SYMBOLS

ALM	Adaptive logic module
ALUT	Adaptive look-up table
ANSI	American National Standards Institute
ASIC	Application specific integrated circuit
AVC/H.264	Advanced Video Coding
BAR	Base address register
CABAC	Context-adaptive binary arithmetic coding
CB	Coding block
CTB	Coding tree block
CTU	Coding tree unit
CU	Coding unit
DCT	Discrete cosine transform
DMA	Direct memory access
DSP	Digital signal processor
DST	Discrete sine transform
FDQ	Frequency dependent quantization
FIFO	First in, first out
FPGA	Field-programmable gate array
FPS	Frames per second
HDL	Hardware description language
HEVC/H.265	High Efficiency Video Coding
HLS	High-level synthesis
HM	HEVC Test Model
HSSI	High-speed serial interface
IDCT	Inverse discrete cosine transform
IDST	Inverse discrete sine transform
IO	Input/output
JCT-VC	Joint Collaborative Team on Video Coding
LE	Logic element
LUT	Look-up table
MPEG	Moving Picture Experts Group
PB	Prediction block
PCIe	PCI Express (Peripheral Component Interconnect Express)
PLL	Phase-locked loop
PU	Prediction unit
QP	Quantization parameter
RTL	Register-transfer level
SAD	Sum of absolute differences
SSD	Sum of squared differences
TB	Transform block
TU	Transform unit
UHD	Ultra-high-definition
VCEG	ITU-T Video Coding Experts Group
VHDL	Very high speed integrated circuit hardware description language
WPP	Wave-front parallel processing

1. INTRODUCTION

Internet video traffic is forecast to grow threefold in five years from that of 2015 and video is estimated to account for 82% of all global consumer Internet traffic by 2020 [1]. This growth comes from new end users and multimedia applications entering the market but also from higher video dimensions, resolutions, frame rates, and color depths. Despite the fast progress of network and storage capacities, rapidly increasing volume of video makes more efficient video compression inevitable.

The latest international video coding standard, *High Efficiency Video Coding (HEVC/H.265)* [2] is developed to address the increasing transmission and storage needs. It aims to halve the bit rate compared with the current mainstream *Advanced Video Coding (AVC/H.264)* [3] standard without sacrificing video quality. However, higher compression rate comes at a cost of increased complexity in encoders and decoders. In practice, the complexity of encoder tends to be at least doubled [4].

Due to complexity of HEVC, encoding a high-quality video is slow and requires a lot of processing power. Therefore, HEVC encoder is an ideal candidate for hardware acceleration. The target of this Thesis is to speed up Kvazaar HEVC encoder [5] with a hardware accelerator implemented on *Field Programmable Gate Array (FPGA)*. Kvazaar is an award-winning open-source HEVC encoder. It is written in low-level C language which acts as a good starting point for the implementation work.

High-Level Synthesis (HLS) will be used to accelerate hardware development. Hand-written *Register-Transfer Level (RTL)* could offer better results, but it could easily increase tenfold the development and verification effort required. Implementing a full hardware encoder is the ultimate goal of the work described in this Thesis, but due to time constraints the scope of this Thesis is limited to acceleration of HEVC intra encoding.

The structure of this Thesis is as follows: Chapter 2 introduces HLS, Kvazaar, and applied design flow and verification strategies. Chapter 3 presents intra encoding in HEVC. Chapter 4 introduces the structure of the accelerator system and Chapter 5 shows the Intra coding accelerator. Chapter 6 presents the results of the work, including synthesis results and encoding performance. Finally, Chapter 7 concludes the Thesis.

2. BACKGROUND

This chapter introduces *High-Level Synthesis* (*HLS*), Kvazaar, and the used design and verification flows.

2.1 High-Level Synthesis (HLS)

HLS is an automated design process where RTL *hardware description language (HDL)* code is automatically generated from a high-level program code like C or C++ by HLS tools. This has many advantages compared with traditional HDL design. Most notably HLS offers faster design and verification times and higher design reusability [6].

A traditional RTL based hardware design flow usually begins by creating executable software models from a specification. This model is used to validate and fine-tune the desired behavior. Once tested, the design of the actual hardware implementation can begin. In an implementation phase, the used architecture is defined, and the implementation is hand-written in HDL code for that specific architecture [6]. However, finding a suitable architecture is not a simple task and hand writing the implementation is error-prone that requires a lot of testing, leading into a cycle of bug reporting and fixing.

With HLS, an executable model can be used to generate a hardware model. If the executable model is built with HLS, it can also function as source for RTL synthesis. No separate hardware implementation is needed. Although this will require model to be written with hardware and tool limitations in mind and will most like require specific hardware related optimizations. In addition, the same tools can be used to verify design functionality and the generated RTL code with software test benches. If the underlying architecture changes later, new RTL can be easily regenerated making the HLS implementation architecture independent. Even during this work, the design has moved from one FPGA to another a handful of times and without any changes in code.

In this Thesis, the HLS tool used is Catapult-C version 10.0a [7]. Catapult-C is developed by Mentor Graphics and it can generate both *Very High Speed Integrated Circuit Hardware Description Language (VHDL)* and Verilog code from *American National Standards Institute (ANSI)* C, C++ and SystemC codes. The other notable HLS tools competing in the market include Synphony C Compiler from Synopsys [8], Vivado High-Level Synthesis from Xilinx [9] and Intel's HLS Compiler [10]. Catapult-C was selected by the project management and the comparison of the HLS tools is outside of the scope of this Thesis.

2.2 Kvazaar

Kvazaar [5] is an academic open-source HEVC encoder. It has been developed from scratch in C by Ultra Video Group at Tampere University of Technology. Kvazaar has five main development goals: 1) coding efficiency close to *HEVC Test Model (HM)* reference encoder [11]; 2) easy portability to various platforms; 3) real-time coding speed; 4) optimized computation and memory resources; and 5) well-documented source code [5].

Since Kvazaar is written in C, a language supported by HLS, it works well as a reference implementation for the hardware development. For this reason, Kvazaar functions are used as starting points for hardware blocks and as reference models for verification. During the work done in this Thesis, Kvazaar has been updated multiple times and new updates were periodically included to hardware design as well. The final version used in this Thesis was v1.2.0.

2.3 Verification

Two kinds of verification strategies were used in this work. Individual modules were verified against the reference software implementation with Catapult-C and the whole FPGA system was tested against Kvazaar software implementation.

Catapult-C supports both functional verification and RTL simulation. Functional verification is faster, and it is run before synthesis, but it is untimed. The RTL simulation requires full synthesis and system simulation where each signal can be examined with a wave viewer. The functional verification was found to be sufficient in most cases as the RTL simulation could not find any new functional errors. RTL simulation was only used when timing problems, like wrong transaction orders, were discovered.

With Catapult-C, the module-level verification effort becomes minimal since most blocks required only a small C testbench. The testbenches were mostly wrappers that feed data to the hardware and software blocks and compare the outputs. The input data was dependent on the module, either values were randomly generated or dumped from Kvazaar. To further decrease the verification effort, the same testbench were used for both, the functional and RTL level testing.

In the full system testing, the design was synthesized and flashed to an FPGA chip. At first, Arria V [12] development board was used. With ARM System on Chip (SoC) processor, the testing was fast due to a rapid system bootup. *Peripheral Component Interconnect Express (PCIe)* interface and parallel processing could not be tested efficiently on ARM, so later the testing moved to Arria 10 PCIe card. In both cases, the accelerated Kvazaar was tested with the same input data as the reference software and differences

were reported, as the results should be identical. During this Quartus Signal Tap tool were used to inspect internal signals if error were discovered.

2.4 Design flow

***Figure 1**. Tools in different steps of design flow*

In this work HLS design flow from software to hardware, started from Kvazaar source code that synthesized to RTL with Catapult-C and to the hardware in Quartus, pictured in Figure 1. Implementation of a hardware block begins by separating the desired functionality from the source code. This is usually almost done as different coding tools are usually written as separate functions or in some cases a combination of them. First step to this is to modify the code to work with Catapult-C. This requires rewriting the code to work as a continuous loop with streaming input and outputs, removing unsupported code functionality, and optimizing basic operations. Code functionality that cannot be synthesized, like dynamically allocated memory, unbounded function calls, and recursion, are removed or rewritten to use their static counterparts. Basic optimizations techniques are used with all blocks. The simplest and usually very efficient optimization was to change variables to bit-accurate data types. Using this approach, Catapult-C can use smaller operators instead of defaulting to the worst-case scenario. Usually, more innovative optimization methods were required for the best results, like algorithmic changes or reusing existing logic.

Next, the modified code is synthesized to RTL code. During this step, the used architecture is specified so that Catapult-C can map logical operations to physical resources. Arrays are mapped to registers or on-chip memories and arithmetic operators to adders, multipliers or any other resources assigned by the tool. With FPGAs, other more advanced operations can be also mapped to dedicated hardware resources if available on the chip.

In the third step, the synthesized RTL code is cleaned with a custom Python script. A code synthesized by Catapult-C uses a lot of generic sub-components with generic names. When multiple synthesized modules are added to a single project, the generic names can collide and cause errors. A small custom Python script is used to rename these modules to avoid inconsistencies with the following tools in the flow.

Along with the HLS generated blocks, readymade IP components from the FPGA vendor are used to utilize other FPGA resources. These blocks are instantiated and configured in Quartus Platform Designer. The tools will then automatically handle internal synthesis and configurations options for these components. The blocks are connected together with a hand-written VHDL top-level module. As a final step, the system is compiled in Quartus that generates a bit-file that can be used to program the FPGAs.

2.5 Field Programmable Gate Arrays (FPGAs)

FPGAs are re-programmable logic circuits that are used in hardware development, prototyping and in commercial products. They offer fast development cycles and exhaustive support for hardware debugging. In this Thesis, the results are based on real FPGA executions instead of calculations or simulations.

In this Thesis, the FPGA designs are implemented on readymade FPGA development boards manufactured by Intel (formerly Altera) and its partners. The development begun with Cyclone II FPGA in DE2 development board. When more logic elements (LEs) were needed the development moved to Arria II GX FPGA Development Kit. When the development progressed, the lack of an integrated CPU became a limiting factor. Then a Cyclone V FPGA on a VEEK board with an ARM Cortex-A9 processor was took to use. When again more LEs were needed the development moved to Arria V FPGA with the same ARM processor but with more logic resources.

In the final step, Arria 10 GX FPGA Development Kit was taken in use. The software processing was moved from onboard CPU to a host PC and the integrated processor was replaced with a PCIe interface. The final executions were run with commercial Arria 10 PCIe cards.

3. HIGH EFFICIENCY VIDEO CODING

This chapter presents an overview of HEVC and its intra encoding loop.

3.1 Overview

High Efficiency Video Coding (HEVC) is the latest video coding standard developed by *Joint Collaborative Team on Video Coding (JCT-VC)* as a collaboration of ISO/IEC *Moving Picture Experts Group (MPEG)* and *ITU-T Video Coding Experts Group (VCEG)* [13]. HEVC was published in January 2013 and it is designed to meet the transmission and storage needs of modern video applications. It aims to halve the bit rate compared with its predecessor Advanced Video Coding (AVC/H.264) while still sustaining the same subjective quality. The efficiency comes at a cost of coding complexity. HEVC encoding and decoding complexities tend to be at least 1.5 times higher than those of AVC [14].

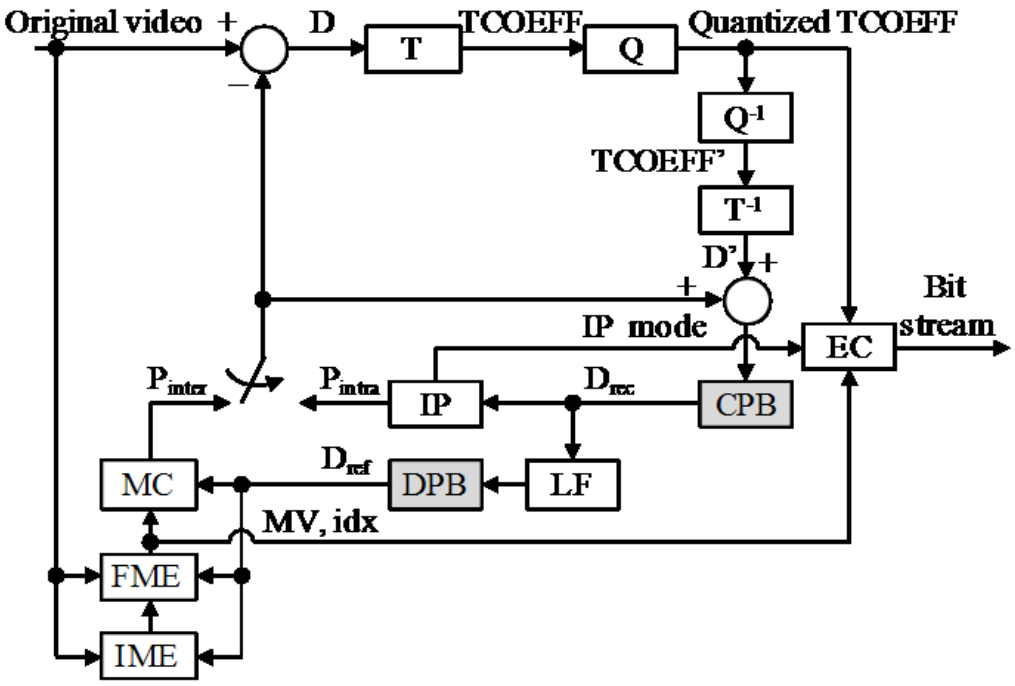

Figure 2. *HEVC encoder model [14].*

Like prior video coding standards, HEVC uses hybrid video codec approach pictured in Figure 2. This model originates from H.261 [15], the first member of the H.26x family of video coding standards. The hybrid video codec scheme is composed of intra/inter prediction, transform coding and entropy coding.

Encoding operation starts by splitting an image/frame to square blocks that are individually coded and added to the encoded bit-stream. Actual block coding starts with the prediction phase, where an estimation of the image is generated by using pre-defined prediction methods.

Prediction can be done either by using intra prediction (P_{intra}) or inter prediction (P_{inter}) modes. Intra prediction [16] uses spatial redundancy to compress blocks in individual pictures. Inter prediction exploits temporal redundancy in video to compress similarities between different video frames.

The prediction is subtracted from the original source image to generate a residual image (D) that contains the differences between prediction and the original image. Transform block (T) transforms the residual image from spatial domain to frequency domain coefficients (TCOEFFs). In frequency domain, high-frequency components of the video can be removed in Quantization block (Q) without significant quality loss, since human eye is less sensitive to the high-frequency components. In the last phase, the Quantized TCOEFFs and prediction modes are entropy coded (EC) to generate an encoded bit-stream.

The encoding loop continues with Inverse Quantization (IQ) and Inverse Transform (IT) phases where quantized TCOEFFs are dequantized and transformed back to spatial domain. This generates a reconstructed version of the residual image that is added to the prediction to generate the final reconstruction image. This corresponds to the image generated and displayed by the decoder in playback. In encoders, reconstructed image is generated because it is needed in the following intra and inter prediction phases, so that the reference picture stays the same in the encoder and the decoder.

In HEVC, a traditional macroblock structure of AVC is replaced with a more flexible block partitioning scheme with four different coding units: *Coding tree unit (CTU), Coding unit (CU)*, *Prediction unit (PU)* and *Transform unit (TU).* Each of these consists of one luminance and two chrominance blocks that cover the corresponding block areas: *Coding tree blocks (CTB)*, *Coding blocks (CB)*, *Prediction blocks (PB)* and *Transform blocks (TB)*.

Each video frame is partitioned into CTUs that can be 64 × 64, 32 × 32 or 16 × 16 pixels in size. Larger sizes usually provide better compression rates [13]. CTU is a quadtree structure where the root CU can be recursively split into four smaller CUs down to an 8 × 8 -pixel CU. Each CU in the CTU is predicted and transformed individually.

CUs can also be further split into different sized and shaped PUs and TUs for prediction and transformation phases. In all-intra coding, PUs and TUs are always the same size as the corresponding CU, despite that the smallest 8 × 8 CU can be further split to four 4 × 4 -pixel PUs and TUs. Inter coded CUs provide PUs and TUs with more split options, including non-square shapes, but inter coding is out of the scope of this Thesis.

3.2 Intra Prediction

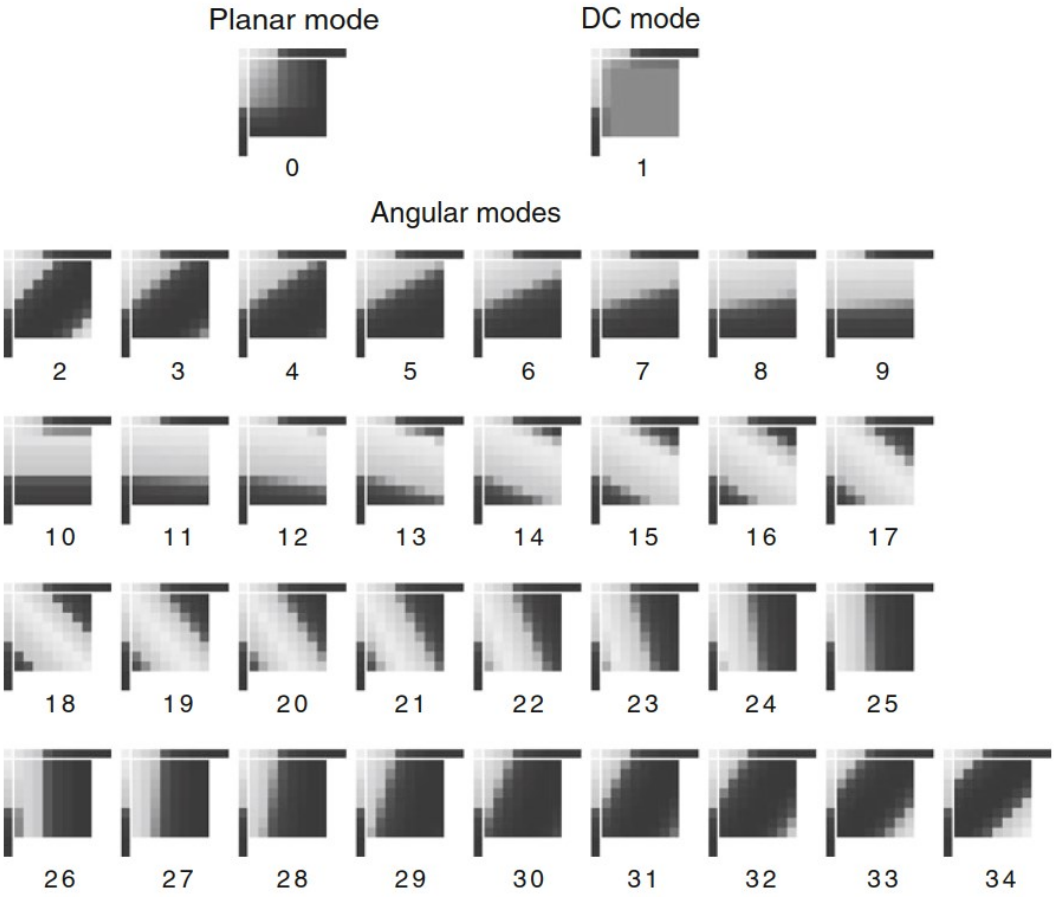

Figure 3. *HEVC intra prediction modes for 8 × 8 PU [13].*

In intra coding, the prediction phase exploits a spatial redundancy, i.e., the correlation of adjacent pixels within one frame. A prediction is calculated by extrapolating neighboring pixels. In this way, only the prediction mode needs to be encoded in bit-stream as the prediction can be calculated from previously decoded pixels.

The HEVC has three intra prediction modes: planar, DC and angular. Angular modes support 33 different angles of prediction, so the total number of different prediction modes is 35. All prediction modes can be used with all PU sizes, from 4 × 4 to 32 × 32. Figure 3 displays how different predictions are formed for a sample 8 × 8 PU.

HEVC planar prediction was specified to estimate smooth areas and preserve continuities along the block edges. It is obtained averaging a horizontal and vertical linear prediction on sample basis [13]. DC prediction is for a uniform area and it is defined as an average of the reference samples.

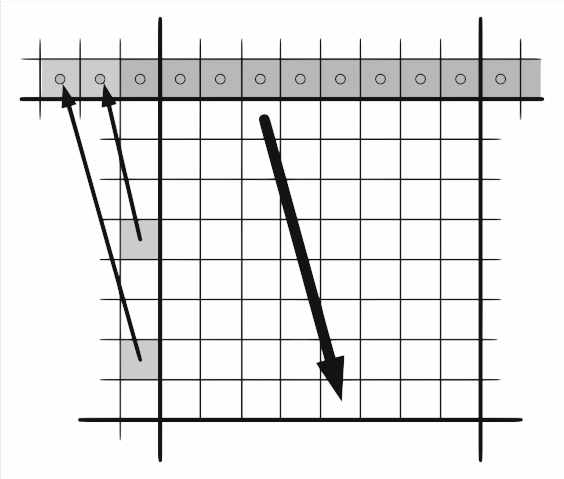

Figure 4. *Generation of angular intra prediction [13].*

Angular prediction modes in HEVC are designed to efficiently model different directional structures typically present in video and image content [13]. In order to calculate the prediction, reference samples from adjacent border are extrapolated to extend to reference samples from direction of the prediction, pictured in Figure 4 with thin arrows. Prediction is then calculated by projecting extended border samples to the PU area, as displayed in Figure 4 with a bold arrow.

As reference-samples, intra prediction uses pixels from the borders from top and left edges of the PU. If all needed samples are not available, as they could be outside of the frame or not yet coded due to coding order of the CUs, last valid sample is repeated.

To improve prediction quality, reference samples can also be filtered to smooth out large steps in sample values. Those could generate undesired directional edges in the prediction blocks. Filtering is done with a smoothing filter that averages samples with its neighbors. The decision to use filtering is dependent on PU size, prediction mode and the direction.

3.3 Transform

In the transform phase, the spatial domain residuals are converted to transform domain frequency coefficients. The transform outputs a transform coefficient matrix, where each coefficient represents the amplitude of its predefined base frequency. Amplitudes of lower frequencies are stored in the top-left area of the matrix and higher frequencies at the bottom right. Typically, images have no or very little high frequency components, especially after the quantization phase. Meaning that the coefficient matrix usually contains a lot of easily compressible zeroes.

In most video coding standards, including HEVC, only the inverse transforms are specified by the standard and the exact details of the forward transforms are left to the implementer [13]. Typically, the used forward transforms are transposes of the inverse transforms to achieve near lossless transform coding [17].

HEVC uses two-dimensional (2-D) finite precision integer approximations of two common transforms: 1) *Discrete sine transform (DST)* for intra-coded luminance TBs of size 4 × 4 and 2) *Discrete cosine transform (DCT)* for all other TBs. Both transforms are separable, they can be computed by applying two 1-D transforms first row-wise and then column-wise [18]. Increasing transform size from 8 × 8 used in AVC up to 32 × 32 improves coding gain by around 5-7 % but it also introduces the majority of complexity overhead in HEVC transform coding [18].

3.4 Quantization

In quantization, high frequency components are reduced by dividing transform coefficients by a Quantization step (Q_{step}) as

$$\boldsymbol{Q_{step}(QP) = (2^{1/6})^{QP-4}}. \quad (1)$$

Quantization parameter (QP) can have values in the range of 0-51 for a standard 8-bit video. Increasing QP by one means an increase of the quantization step size by approximately 12 %. An increase of six doubles the quantization step size. The QP value can be specified by the user or it can be automatically decided and adjusted by the encoder. The QP value directly relates to the compression ratio and image quality of the encoded video. Higher QP increases the compression rate and decreases video quality [13].

HEVC also supports f*requency dependent quantization (FDQ)* where different frequencies are quantized differently according to sensitivity of human vision system (HVS). Varying quantization is done by using predefined quantization matrixes, where low frequency coefficients are quantized with a finer Q_{step} and higher frequencies with a greater Q_{step}. As a result, FDQ has a negligible effect on the perceived video quality while still improving the coding performance [19].

Inverse quantization, also called dequantization is performed by multiplying transform coefficients by a Q_{step}. This returns quantized coefficients back to transform coefficients that can be used to generate a reconstructed picture. As with the transform, only inverse quantization is specified by the HEVC standard and the details of the forward quantization are left to the implementer.

3.5 Inverse Transform

The inverse transforms specified by HEVC are similar to their forward transform counterparts: *Inverse Discrete Sine Transform (IDST)* for intra-coded luminance TBs of size 4 × 4 pixels and *Inverse Discrete Cosine Transform (IDCT)* for all other TBs.

The core transform in HEVC was designed to be close to the integer approximation of the base IDCT function but still being efficient on both software and hardware. The effi-

ciency goal was achieved by limiting bit depths and using symmetrical transform matrices. With limited bit depths, multiplications can be performed with 16-bit multipliers common in hardware. In addition, symmetrical transform matrices share values for different TU sizes so only one transform matrix is needed.

DST has been found to reduce bit-rate by 1 % when coding small intra-coded TBs. The quality of intra prediction is better near top and left borders [13]. In residual picture, the amplitude of those samples tends to be lower and higher for samples further from the edges. The base functions of DST are found to model this effect better and therefore reduce bit-rate. For larger block sizes, DCT is shown to outweigh DST and therefore DST is only used for small luminance TBs [20].

3.6 Entropy Coding

Entropy coding is the last step of the video encoder. In this step, video signal is reduced to a series of syntax elements that contain properties of the CTUs, CUs, PUs and TUs, including prediction modes, quantization parameters, transform coefficients, filter modes and all other parameters required to describe how the video signal can be reconstructed by the decoder [13]. These elements are ordered and compressed to generate an encoded video bit-stream.

Entropy coding method used in HEVC is *Context-based adaptive binary arithmetic coding (CABAC)*, as in AVC. CABAC is a lossless compression technique based on arithmetic coding. Compression comes from statistical properties of symbols; more frequent symbols can be coded with less bits and more rare symbols with more bits. The number of bits is logarithmically proportional to the probability of the symbols [13].

4. SYSTEM OVERVIEW

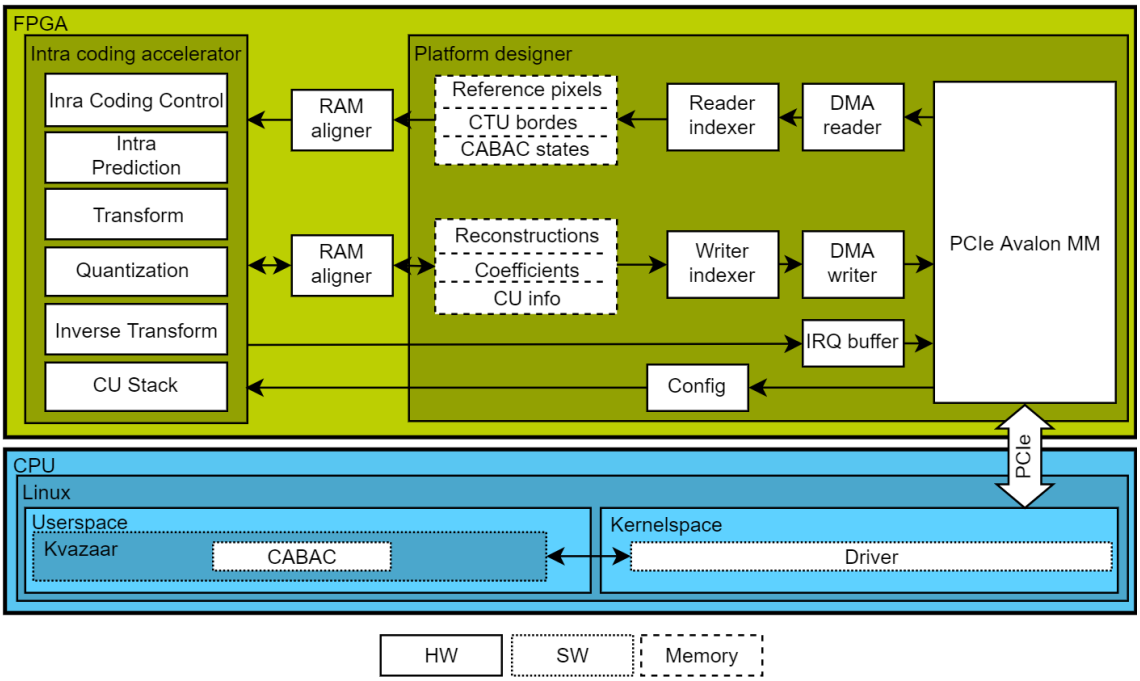

Figure 5. *System overview*

Figure 5 shows the partitioning of the encoder. Kvazaar running on a Linux PC handles CABAC coding, bit-stream generation and other control-intensive coding tools such as *Wave-front Parallel Processing (WPP)* and picture-level parallelism. FPGA accelerator operates at the CTU level and handles the most computationally intensive intra coding tools including intra prediction, transform, quantization, inverse quantization and inverse transform.

An FPGA accelerator consists of a PCIe interface, two *Direct memory access (DMA)* blocks, on-chip memories with indexers and aligners, Intra coding accelerators and an IRQ buffer. One FPGA can have multiple accelerators. In those cases, PCIe interface and IRQ buffer are shared between them but all other components are accelerator specific.

This chapter covers the structure of the FPGA accelerator, including software, drivers and the hardware interface. Intra coding accelerator, the main unit on FPGA, is explored in Chapter 5.

4.1 User Space

On the CPU, a slightly modified version of Kvazaar runs as a Linux user space program that handles input/output (IO), CTU level parallelism and CABAC coding. As input, Kvazaar can take offline files or live stream directly, e.g., from camera or the network.

Likewise, the output can be written to a file or piped elsewhere. Kvazaar has a few built-in options for CTU level parallelism, most notably WPP, tiles and processing multiple frames in parallel. In all cases, parallelization is implemented with a CPU thread pool. On system startup, Kvazaar generates a pool of worker threads and a separate scheduler thread that allocates tasks to available workers. A CTU represents the smallest work unit for a single thread as the CTUs have very little dependencies with each other. Hence, a numerous CTUs can be processed simultaneously.

Kvazaar was modified to offload a majority of coding tasks to the hardware accelerators on FPGAs. Instead of processing CTUs in CPU threads, when a worker arrives in the processing function, it sends its data to the hardware accelerator and sleeps until the accelerator notifies it that the CTU processing is completed. Then, the worker thread continues as without the FPGA acceleration. As the worker threads sleep while waiting for then accelerator, a lot of threads can be used effectively.

4.2 Kernel Space

Kvazaar uses a custom Linux kernel module, i.e., a driver to transact with PCIe FPGA card. The driver hides all complex hardware protocols and registers. It only displays a simple generic software interface so that the software can operate without knowing about the underlying hardware. Hardware could be composed of one or more PCIe FPGA cards, or there could be FPGAs with different interfaces and different number of accelerators per chip.

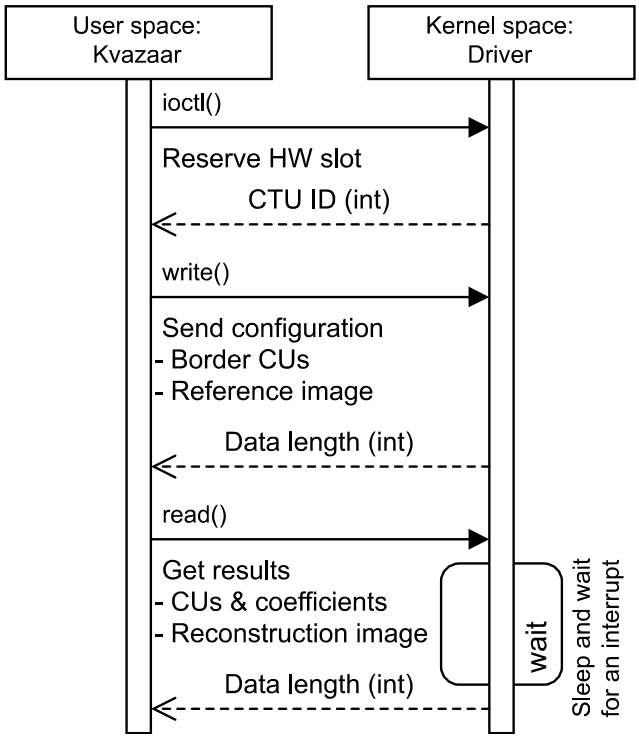

Figure 6. *Sequence chart of system calls between Kvazaar and a kernel driver*

Figure 6 shows the sequence chart of system calls between Kvazaar and the kernel driver. At first, Kvazaar calls ioctl function to request a free hardware slot from the driver. If there are hardware slots available, the driver returns an id number of the reserved slot. If no slot is available, a negative number is returned to indicate an error. In these cases, the slot reservation can be either tried again later or it can be decided to process the CTU in software. If the reservation is successful, processing can be unloaded to the hardware. The reservation is valid for a single CTU computation and is automatically freed after its completion.

After the slot is reserved, Kvazaar calls a write function of the driver to copy all necessary data of the processed CTU to the FPGA. The driver combines different data slots into a single chunk and writes it to consecutive virtual memory addresses in the kernel memory space. The actual data transfer to the FPGA is done by hardware DMAs in the FPGA in order to speed up the transfer and save cycles on the CPU. After the data is copied to the kernel, the driver sends the start address and the length of the data to the DMA through PCIe *Base address registers (BAR)* and the DMA starts the transfer. The driver is halted during the data transfer after which hardware processing can be started.

After the data is written, Kvazaar can call a read function of the driver. The driver checks whether the data is available. If not, the thread is put to sleep and it waits until the data is ready. After the CTU processing has completed, the DMA transfer from hardware to kernel memory is automatically started. When all data is transferred back to kernel, the hardware raises a CPU interrupt. The driver catches the interrupt, identifies the task, and wakes up the sleeping thread. The thread then copies all the data back to Kvazaar memory structures in the user space memory. The thread is returned from the kernel and the Kvazaar encoding continues as usual.

4.3 PCIe and DMAs

The CPU - FPGA interface is implemented with PCIe bus. On the FPGA side, the PCIe bus protocol is handled by Intel's Arria 10 Hard IP for PCI Express IP Core [21]. The IP Core is instantiated in Platform Designer where it offers a single Memory Mapped Avalon (Avalon-MM) interface [22] for accessing the CPU memory.

The PCI Express IP Core is configured to the second fastest configuration option: a PCIe generation 3.0 with x4 lanes and a 128-bit interface with a 250 MHz application clock. This configuration gives 31.52 gigabits per second (Gbit/s) bidirectional theoretical throughput. In our test, we achieved 29.87 Gbit/s throughput with two DMAs reading and writing simultaneously. That is sufficient for transferring multiple streams of raw *Ultra-high-definition (UHD)* video in and out of the accelerator, as one raw 30 *frames per second (fps)* UHD 8-bit 4:2:0 video requires a bandwidth of 3 Gbit/s. The fastest option would have been a generation 3.0 with x8 lanes. It would increase the interface width to

256 bits and double the bandwidth, but this configuration is not recommended for our chip by the manufacturer and it would only support Avalon Streaming interface.

Two custom DMA blocks are connected to the PCI Express IP Core, one for both directions. DMAs are used to reduce CPU load when transferring data through PCIe bus. On system startup, the kernel module configures the DMAs with base addresses of the allocated memory regions and transfer data lengths over BAR registers on the PCIe bus. The CPU → FPGA transfer is started by the CPU that writes a CTU id to a DMA reader block. The DMA reads data from the CPU memory using the configured base address with offset calculated from the CTU id. Data is then written trough read indexer to the FPGA on-chip memories. In the opposite direction, the FPGA → CPU transfer is automatically started by the Intra coding accelerator when CTU processing is completed. After the DMA writer is ready, an interrupt request is sent to the IRQ buffer. The IRQ buffer raises a CPU interruption and holds it until it is marked as processed. In the meantime, other interrupt requests are buffered so none of them is lost.

4.4 FPGA Memories

As an input, the Intra coding accelerator requires original source pixels of a CTU and previously calculated reconstruction pixels from the CTU borders, along with prediction modes from bordering CUs and CABAC states. Inputs for the accelerator sums to over 6 kB data, which needs to be stored on-chip for fast access. The data is stored to multiple memory modules with different data widths as required by the different phases of the coding pipeline.

The accelerator outputs a reconstructed image, transform coefficients, and CU configurations within the CTU. The accelerator produces close to 19 kB of data which is stored on-chip before it is written to CPU's memory.

Read- and Write Indexer are located between the memory modules and DMAs. These indexers handle address translation and separate data stream to different memory modules. Address translation is needed as the DMAs can only read and write from consecutive memory addresses but each of the memory modules have their own address spaces. When the indexers see a read or write request from a DMA they fetch the base address of the target memory from an internal look-up table and add it to the DMA write- or read address. With the corrected address, accessing the memory modules are automatically handled by the Avalon-MM interconnect.

RAM aligners are located between the memory modules and the Intra coding accelerator. They are used to change data widths of the memories to match the widths needed by the accelerator. The hardware supports mixed width memory modules with read and write ports of different size, but the memory controllers are limited to a maximum data bus width ratios of one to four. On the DMA side, the memories have a 128-bit wide data bus

required by the PCIe interface. On the accelerator side, when the required data width is within the supported range i.e. 128, 64 or 32-bits, normal memories with mixed width ports were used. Otherwise, additional RAM aligners were added to convert bus widths down to a required size.

5. INTRA CODING ACCELERATOR

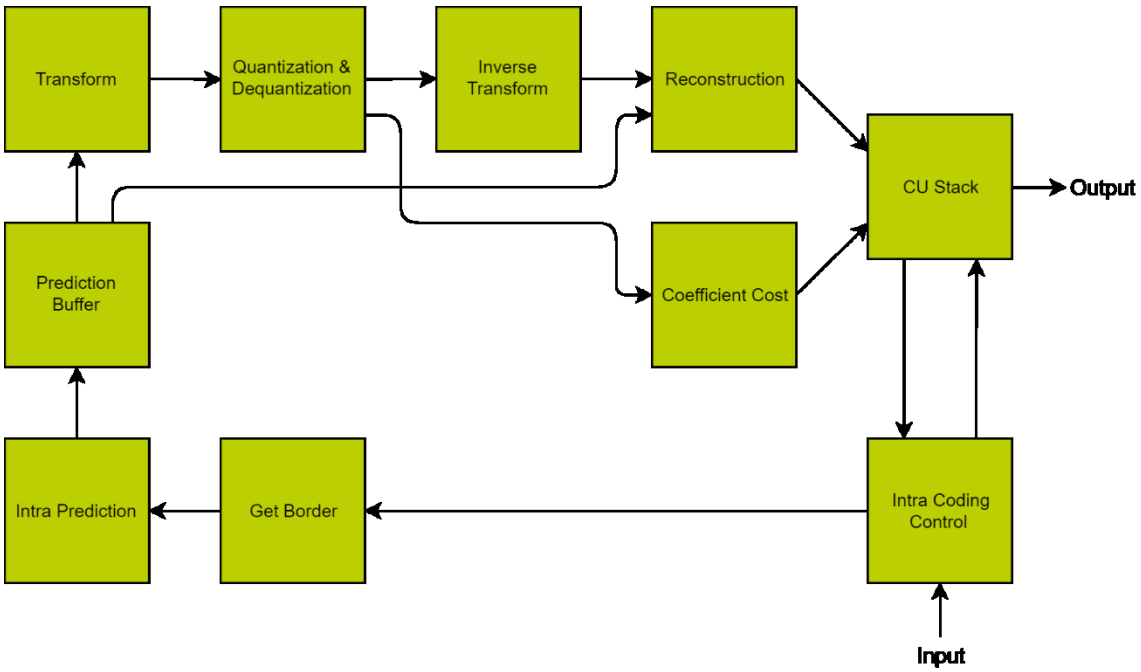

Figure 7. *Block diagram of Intra coding accelerator*

The Intra coding accelerator implements the intra search, generates transform coefficients and reconstruction images for CBs, and performs mode decisions on CTU level. It is configured with the following search parameters: 1) the CTU reference pixels and 2) reconstruction pixels and predictions modes from its left and top borders. After the search is completed, the accelerator sends an interrupt signal to the DMA block that starts data transfer and notifies the CPU. Figure 7 shows the structure of the Intra coding accelerator.

This chapter explores the Intra coding accelerator and its sub-blocks one by one. The main sub-blocks are a Intra coding control block, IP block, transform, quantization/dequantization, inverse transform, reconstruction, coefficients cost, CU stack, and finally a transpose block that is used in multiple locations within the coding pipeline.

5.1 Intra coding control

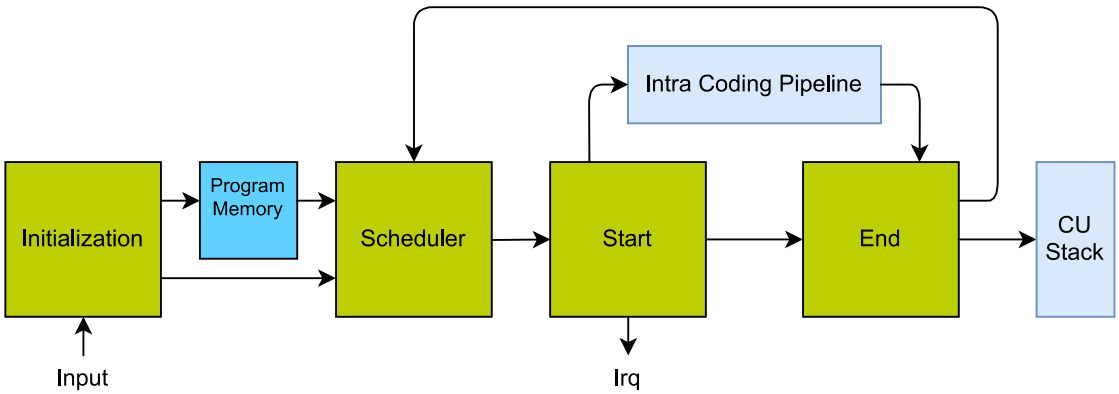

Figure 8. *Block diagram of Intra coding control block*

The CTU level coding control in Kvazaar combines intra coding with inter coding and many other coding tools not supported in hardware. Kvazaar software also uses software tools like recursion, which makes the software implementation unusable in hardware. Therefore, a new coding control was written for the hardware accelerator. A new control block uses the tools from the software implementation, but it was written from scratch with hardware limitations and parallel processing in mind.

The structure of the new Intra coding control block is pictured in Figure 8. It consists of four sub-blocks: Initialization, Scheduler, Start and End blocks and a program memory. The control block is built like a CPU, that is, CTU encoding is split into smaller steps that are preformed one after another. Different steps are described with different instructions with which we can write programs to encode CTUs.

STR	Initialize CTU and start the program
IP	Perform intra search, build reconstruction for a PU and store it on a stack
CMP	Compare cost values of CUs in stack and select the best
END	End current program and send CPU interruption

Table 1. *Intra Coding instruction set*

The control block was implemented as an instruction-based system to offer improved configurability and scalability. The basic operations of the four available instructions are covered in Table 1. STR instruction starts a new program, IP instruction calculates a predictions for a given CU, CMP instruction compares CUs in the stack, and END instruction ends the current program.

Execution of these instruction are split into two parts. Some of the operations are executed in the Start block before the intra coding pipeline and the rest after the pipeline in the End

block. Along with the type, each instruction contains other operation parameters and a skip address. The parameters carry other processing information like block size and coordinates. The skip address is used to tell where to move in case of processing of unnecessary instructions is skipped.

The Initialization block is the first part of the control block . It receives CTU configuration from the CPU and selects or generates the program needed to process the CTU. To avoid generating a new program for every CTU and thus increasing latency, the control block has a few prebuild programs saved on read only memory for most common CTU configurations. If a configuration does not match a prebuild program, the Initialization block generates a new program for the specified configuration dynamically and saves it in the program memory. Once the program, prebuild or dynamically generated is ready, the starting address of the selected program along with its configuration is sent to the Scheduler block.

The main task of the Scheduler block is to schedule instructions when running multiple programs in parallel. After receiving the configuration, the scheduler reads the instruction from the starting address and stores it into an internal cache. The cache contains latest instructions from all running programs which can be either in active or inactive state. For all active instructions, the scheduler calculates a priority numbers and selects the one with the highest priority. The selected instruction is sent to the Start block for processing and a cached copy is changed to an inactive state.

The instructions are prioritized to maximize the efficiency of the coding pipeline. Due to the structure of the pipeline, the CBs of different size will move at different speeds. Larger CBs will create congestion behind them and reduce efficiency. To minimize this, the CBs of the same size are processed in sets. This method is used for the IP instructions. Other instructions have very little effect on the pipeline and thus have a small fixed priority.

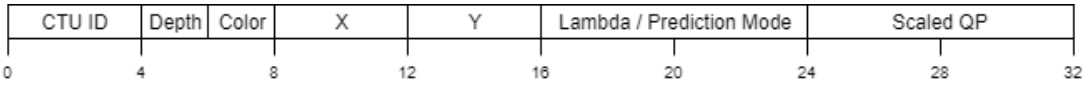

Figure 9. *Configuration vector used in intra prediction pipeline*

The Start block is at the beginning of the intra coding pipeline. Its main purpose is to start processing of the CBs in the pipeline with the IP instructions received from the Scheduler. For the IP instructions, the Start block builds a configuration vector pictured in Figure 9 and sends it to the Get Border block. This vector contains all parameters required throughout the pipeline until the End block. The CTU id, depth, color and the coordinates are generic parameters used by most of the blocks in the coding pipeline. Lambda is utilized by the SAD block where it is replaced by the selected prediction mode. The scaled QP

used in the quantization phase. After the instruction is processed, it is forwarded to the End block for further processing.

The STR and CMP instructions require no processing in the Start block and like IP instructions they are forwarded to the End block. The END instruction ends the program. With it, the Start block sends a completion signal to the DMA writer, which then starts data transfer and notifies the CPU. The END instructions are not forwarded to the End block.

The End block is at the end of the coding pipeline. Its two main tasks are calculating cost values for CBs and comparing the cost values to select CUs for the final CTU configuration. The first instruction to arrive in the End block is the STR instruction, which is used to clear old cost values from the cache and reset the CTU state. With IP instructions, the End block first waits until it receives results from the pipeline. Then, it calculates a cost value for a CB and stores it to an internal cache. If the CB is of the smallest size configured or the CB is deemed good enough not to search alternatives, the CU Stack is notified to write the CB to the output memory. As the END instructions are already dropped in the Start block, the third and final instruction type is the CMP instruction. With it, the stored cost values are compared and the best CU configuration is selected. If the selected configuration is not already written in the output memory, the CU Stack is notified again to overwrite it with the selected configuration.

After each instruction, the End block sends an instruction completed signal with a skip-flag to the scheduler. The flag is used to indicate if the CU was deemed good enough and the scheduler should skip the search for alternative CUs. If this flag is set, the scheduler reads the skip address field from the cached instruction and moves to that address accordingly. Otherwise, the next instruction is read from the following address, stored into the cache, and marked as active. The process continues until the program is terminated with the END instruction.

5.2 Get Border

The Get Border block is the first block in the intra coding pipeline. It generates reference pixels for the intra prediction from CTU reference borders and previously calculated CUs. The block receives the configuration information (the block size and CU coordinates) from the control block. Get Border block then generates reference borders by using pixels from the CTU borders or from the previously calculated CUs. If no border pixels are available the last available pixel or a constant value is used instead. The referenced borders are built and sent to IP block at the rate of four pixels per cycle. The bandwidth is limited by the used memory widths.

In Kvazaar, the reference pixels are fetched from the CTU borders and the output reconstruction memory array. In hardware, the CTU borders are stored into two separate memories. The border pixels are written by the driver and they could be accessed as in software. Fetching the reference pixels coming from the reconstruction picture from the full output memory is inefficient. The output memory is implemented with a dual port memory, where one port is used by the CU Stack for writing and the other port by the DMA for moving data to CPU memory. The memory modules support two ports at maximum so adding more ports would require duplicating the whole memory.

Although possible, reading multiple reference pixels from the output memory is inefficient. It either limits throughput or consumes excess memory and logic cells. Reading multiple pixels from the top border can easily be done with a wider memory interface as the pixels are next to each other. Instead, the pixels from the left border are in different rows and can only be read one at a time. To read left border as fast as the top border, the output memory would need to be duplicated in transposed order.

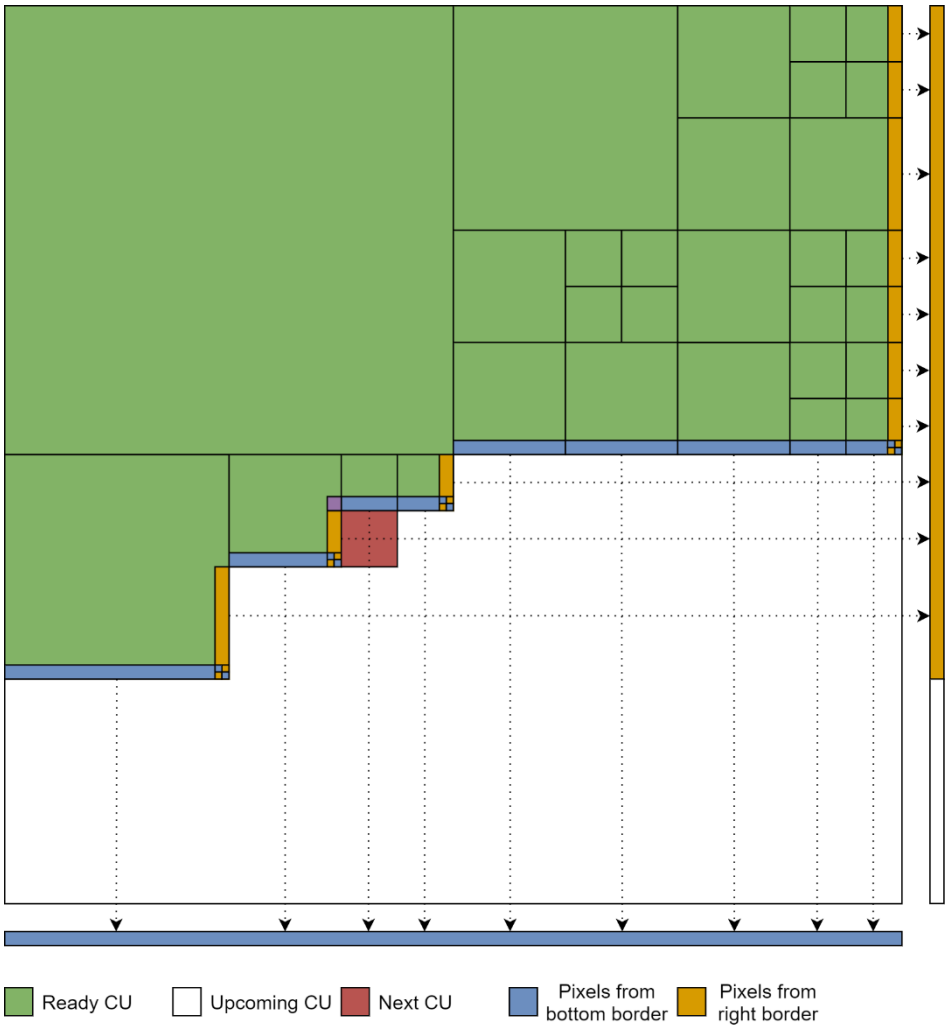

Figure 10. *Content of three buffers memories.*

To reach an adequate throughput, a new method to save and read the top and left borders was developed. Three buffer memories were added between the Get Border and CU Stack blocks. Two 64 pixels per CTU deep memories for the last calculated pixels from bottom and right borders, and one extra memory to store bottom right pixels from all 4 × 4 regions.

In the buffers, the bottom and right border pixels of each CU are stored to their x and y coordinates. Due to the CU coding order, no longer needed pixels are overwritten by those of the following CUs and only the latest pixels are stored. The only exceptions for this are the bottom right pixels from the 4 × 4 regions that are occasionally overwritten before they are used. Example of this case is highlighted in Figure 10 with a purple pixel. To solve this the bottom right pixels are stored separately in a third memory.

5.3 Intra Prediction

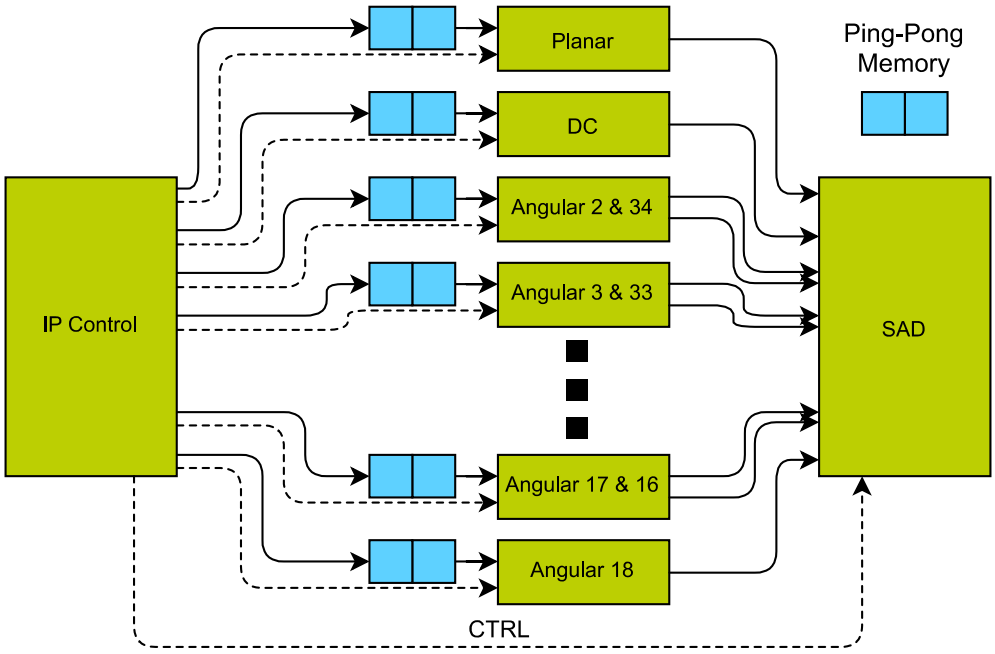

Figure 11. *Block diagram of Intra Prediction block.*

The IP block calculates and selects the best prediction for a given CB. It does this by generating prediction images for all modes and selecting the mode closest to the reference image. Figure 11 depicts the structure of IP block. The IP block is composed of IP Control block, SAD block and 18 parallel prediction blocks with ping-pong input buffers used to compute all 35 prediction modes in parallel. The prediction modes are calculated with five different prediction modules: planar (mode 0), DC (mode 1), positive angular (modes 2-9, 27-34), negative angular (modes 11-25), and zero angle (modes 10, 26). This IP block is an improved version of the IP block presented in [23] and [24].

5.3.1 Control

The IP control block receives PB configuration data and the reference pixels from the Get Border block. Before sending the reference pixels to the prediction blocks, the control block filters them. As doing it once on control block saves resources compared with performing filtering individually in prediction blocks. Filtering is done in real time while receiving new data at the same speed, four pixels at a time. Depending on the mode and the CB, either the filtered or the unfiltered pixels are written to prediction blocks ping-pong memories. After all pixels are read, filtered, and written to memories, the IP control block wakes up the prediction blocks and the SAD block by sending them their configuration data. Along with the common parameters of the PB size and an CTU id, each prediction block has its own configuration parameters, which are dependent on prediction mode.

The ping-pong memories in the prediction blocks are used to pipeline the IP block so that the control block can filter reference pixels from the next PB while prediction blocks are still calculating predictions from the previous PBs. Synchronization and overflow protection between the control block and prediction block are handled with handshaking signals in configuration channels. The channels are implemented by *first in, first out (FIFO)* buffers. The ping-pong memories are configured to allow for two PBs more than the FIFO's depth, one reserved for reading (by the prediction blocks) and the other for writing (by the control block). If the FIFO is full, the control block cannot write configuration data and is blocked until prediction blocks finish and start the next PB by reading a new configuration from the FIFO.

5.3.2 Predictions

All prediction blocks operate in parallel and predict four pixels per clock cycle, i.e., 32 × 32 block is predicted in 256 cycles and 16 × 16 block in 64 cycles. The Planar and DC blocks have the same structure. They both have a duplicated ping-pong memory to support two read ports with a data width of four pixels. With two read ports, both blocks can predict four pixels in parallel. The configuration vector for planar prediction contains last pixels from top and left borders. They are used to calculate the prediction. The DC prediction block has a dc value in its configuration vector and it only performs post-processing for the borders. The dc value is calculated in the control block to reduce latency as it can be effectively calculated in parallel while filtering the reference pixels.

The angular predictions are usually interpreted as a single prediction mode. However, the hardware implementation was split into three different modules according to direction of the prediction angle: positive angles, negative angles and zero angles. This way, the extra complexity could be removed from the simpler modes.

In Kvazaar, horizontal angular predictions (numbered from 19 to 34) are calculated as their vertical counterparts but with their left and top borders swapped. This generates correct pixels but in transposed order. In software, the outputs of the horizontal modes are flipped, but adding a transpose block at the end of all horizontal modes in hardware would require a lot extra logic and memory. It would also notably increase the latency of the prediction phase. To avoid the transpose operation, the algorithm for horizontal modes was redesigned to calculate pixels in transposed order. The new design did require more logic than that for the vertical modes as the algorithm cannot take benefit of relations of pixels in the same rows. The increase in logic was still notably less than adding an extra transpose block and it had no effect on latency.

Together with the redesigned logic for horizontal predictions, the corresponding vertical prediction modes were implemented in the same design as one double prediction block. These predictions make use of the same borders and have the same but opposite prediction angles so they can share the same control logic. This way, all angular blocks, except mode 18 which does not have a pair, predict two modes simultaneously. For example, modes 2 and 34 are equal distance from the middle, i.e., 18 - 2 = 34 – 18 so they are predicted parallel in one block.

The zero angle prediction block is the simplest one of the three implementations. It has the same duplicated ping-pong memory interface as the Planar and DC blocks. It calculates the prediction symmetrically for modes 10 and 26 by projecting the reference samples to the CB area and applying a smoothing filter to boundary samples to reduce discontinuities along the block boundaries.

The positive angle prediction block is used to calculate modes 2-9 and 27-34. It requires thirteen single-pixel ports to the ping-pong memory, 8 for vertical prediction and 5 for horizontal prediction. With dual-port memories, where one port is used for writing, thirteen ports are generated by instantiating thirteen copies of the ping-pong memory with a shared write port. The prediction is calculated with a weighted average of two reference pixels and it is implemented with a dual-multiplication *digital signal processor (DSP)* from HLS DSP library. As the prediction is a projection from the border, the weights and the used reference pixels depend on the projection angle and the location of the pixels. The total weights of the weighted average always sum to 32 so the division operation is implemented with a shift of five.

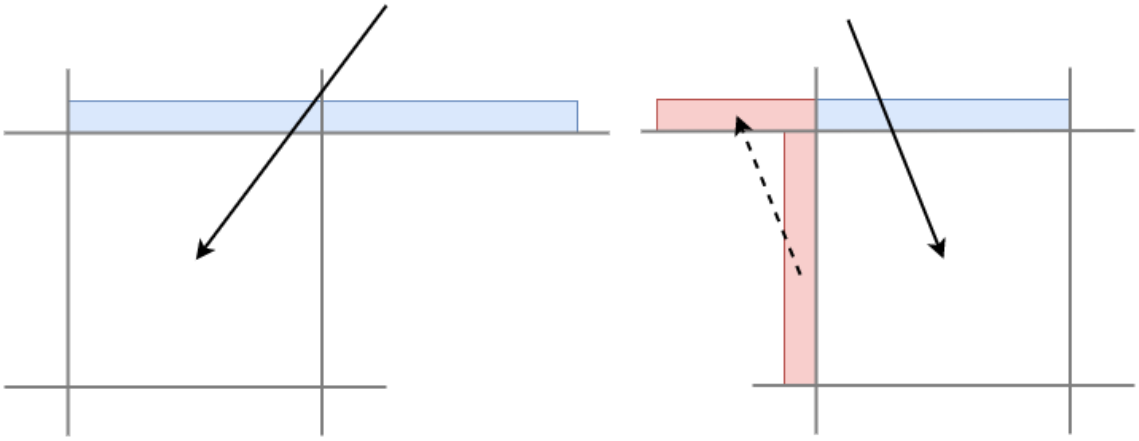

Figure 12. *Projection. (a) Positive edges. (b) Negative angles with border extension.*

A prediction block for the negative angles is the most complex of the three. The block requires the same thirteen single-pixel read ports and the prediction is calculated as in the block for positive angles. The difference comes from the used pixels. The positive angles only use pixels from one border whereas the negative angles use both borders. With negative angles, the side border is extended to the main border before the projection, as pictured in Figure 12. This mapping is not one to one, as the moved pixels depend on the prediction angle. In hardware, pixels are not moved but cross referenced. If the position of pixels is negative a new index number is read from a lookup table and pixel is read from that index in the other border.

5.3.3 Mode selection

The SAD block selects the used prediction mode for luminance PBs by calculating and comparing *Sum of Absolute Differences (SAD)* and entropy coding cost values of all modes. The SAD is used as a measure of image quality whereas the entropy cost values estimate how many bits are needed to code the prediction mode into bitstream. The entropy coding values are usually insignificant compared with SAD values and they only affect the mode decision when two predictions are very close to each other.

The entropy costs are calculated first since there is a small latency in pixel generation. Three entropy coding values are used for each PB. They are calculated by multiplying a fixed entropy cost by a software-defined lambda value, which is derived from the QP. The cost values are then added to SAD calculations. With each prediction mode, one of three entropy coding selected as base cost depending if the mode is one of the preferred modes or not.

For SAD calculations, the block receives four pixels per cycle and it simultaneously reads the reference pixels from an on-chip memory. All calculations are performed in parallel after which predicted and reference pixels are sent to the Prediction Buffer. A mode with the smallest cost is selected and the Prediction Buffer is notified. The comparisons are

implemented as recursive template functions, which are adopted from Catapult-C design examples. It is synthesized to a comparison tree.

The above mode decision is only used for luminance PBs. The chrominance PBs inherit the luminance prediction mode saved in the internal registers. The same prediction mode is also used in hardware implementation as it is default behavior of Kvazaar in all presets.

5.3.4 Prediction Buffer

A prediction Buffer has three purposes: 1) buffer all predictions while SAD block is calculating the cost values and selecting the prediction mode; 2) generate the residual image from the selected prediction and the original reference picture; and 3) change the pipeline data width from four pixels to 32 pixels.

To achieve these goals, the Prediction Buffer is composed of two blocks and an intermediate memory array between them. Input side of the buffer writes data to the memory array while output side reads data from the memory array and generates residual image. The memory array is a set of 36 mixed-width memories between the two blocks. The memories have a 32-bit write port and a 256-bit read port. The input side receives four pixels from all predictions and the reference image in parallel from a 1152-bit wide data bus coming from the SAD block. The data is written to memory array with a matching data rate of four pixels pre-cycle in parallel to all 36 memories.

After selecting the used prediction mode, the SAD block notifies the Prediction Buffer. The output side of the buffer starts reading the pixels of the selected mode and the reference picture. It calculates the residual image by subtracting the prediction from the original image. The mixed-width memories operate on a larger data bus so 32 pixels can be read and computed in parallel. The calculated residuals are sent to the Transform block for further processing. The prediction and reference images are forwarded to the reconstruction block as they are needed in calculating the final reconstruction image.

5.4 Transform

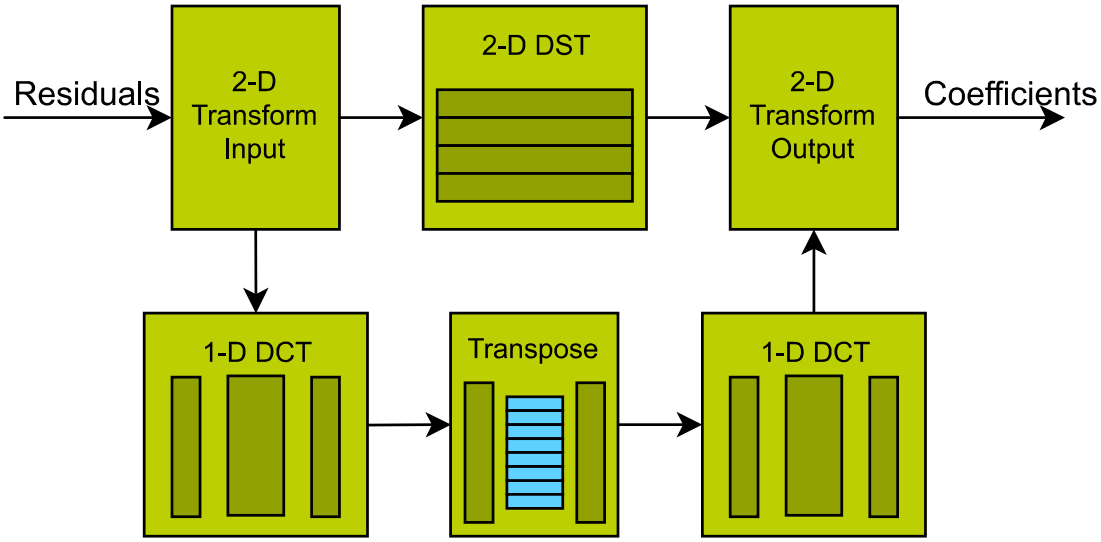

Figure 13. *Block diagram of Transform block*

A transform architecture is pictured in Figure 13. It is composed of three main parts: 1) two 32-point DCT blocks; 2) a separate 4-point DST block for 4 × 4 luminance TBs; and 3) a transpose block for row-column transpositions between the DCT blocks. In addition, the design contains small control blocks in input and output. The transpose block is covered in Chapter 5.10.

The input block reads 32 9-bit residual values from 288-bit input channel. Luminance 4 × 4 TBs are redirected to the 4-point DST block whereas all other inputs are sent to the 32-point DCT block. Output values from the first DCT are row-column transposed in the Transpose block and sent to the second DCT block for a complete 2-D transformation. Output block collects the results from the DST or DCT block (depending on the TB) and outputs the transform coefficients to Quantization phase.

In the earlier stages of the work, a single DCT block was used and data was looped back to the same DCT block through the Transpose block. This approach reduces logic and DSPs, but the loopback would force transform phase to block the coding pipeline during the second round of the DCT. This resulted in a noticeable drop in throughput.

5.4.1 32-point DCT block

The 32-point DCT block performs 1-D transform in a three-step pipeline. First recursive even-odd decomposition, secondly multiplication between the transform matrices and odd vectors, and finally accumulation and scaling of the individual multiplication products to 16-bit coefficients.

The algorithm used for the 1-D transform is a well-known even-odd decomposition algorithm, a.k.a., Partial Butterfly algorithm [18]. It decomposes the input and core transform matrices to half of their sizes according to even and odd rows/columns, respectively. The algorithm allows an N-point transform to be computed for even and odd cases separately with two $N/2$-point transforms that reduce the number of arithmetic operations needed for the full transform. The same algorithm is also used in Kvazaar and it implements the same functionality as in HEVC reference encoder (HM) [11].

Four Partial Butterfly algorithms were combined to a single function with selectable block size $N \times N$, where $N \in \{4, 8, 16, 32\}$. To improve performance in hardware, the algorithm was modified to transform multiple rows/columns in parallel if the block size is smaller than 32×32. This improves performance as $32/N$ rows/columns are transformed in parallel, e.g., 8×8 TBs can be processed in only two parts.

The recursive even-odd decomposition in the first stage of the DCT is implemented as a recursive template function that is synthesized to an adder tree. The adder tree was built for the largest block size. In order to reuse it for smaller block sizes, input vectors need to be reordered to match the structure of the adder tree. The order only depends on the block size, so the reordering can be implemented with a four-port mux.

In the second stage, the odd vectors are multiplied with the transform matrices. The multiplication stage was also built for 32×32 TBs. To reuse it with smaller TBs, input vectors need to be reordered again to match the structure of the multiplication stage.

At first, all multiplications were mapped to DSP blocks to save logic cells. Catapult-C and Quartus can automatically map multiplications to DSP blocks, but they cannot take the most out of the DSPs. For example, a single DSP in Arria 10 can calculate two 18-bit multiplications and add them together [25]. The automatic mapping was unable to detect this and therefore the number of DSPs was doubled.

In Catapult-C, DSP usage could be manually improved by using a library developed by Altera and Mentor graphics [26]. The library contains functions for the most common use cases for DSPs together with some extra DSP features. In DCT, sum of two multiplications mode was used to reduce the number of DSPs. The logic usage could be further reduced by using coefficient banks in the DSPs, but there was no a ready-made library implementation for that and it was not studied further.

In the final stage of the DCT, the individual products of matrix multiplication are added together and scaled to 16-bits. In addition, the output vector has to be reordered back to the original order.

Mapping all multiplications to DSPs allowed two Intra coding accelerators to fit into a single Arria 10 FPGA. Even with two accelerators, there were a surplus of unused logic and DSP usage became a limiting factor. Transform and Inverse Transform took the most

of the DSPs, with a single DCT/IDCT block consuming 172 DSPs. With two accelerators and two DCT/IDCT blocks per transform, 1376 out of 1518 DSPs were used for transforms only.

To fit a third accelerator to a single FPGA, reduction of DSP usage was required. Values of the transform matrices are constant, but matrix itself is selected by the block size. Hence, the multiplication is performed by multiplying input value by one of the four constant coefficients. Smaller block sizes have many zeroes in their matrices as the number of multiplication operations performed for a given block size is $(N/2)^2$. For example, TB of size 32 × 32 requires 256 multiplications whereas four parallel 8 × 8 rows/columns require only 64.

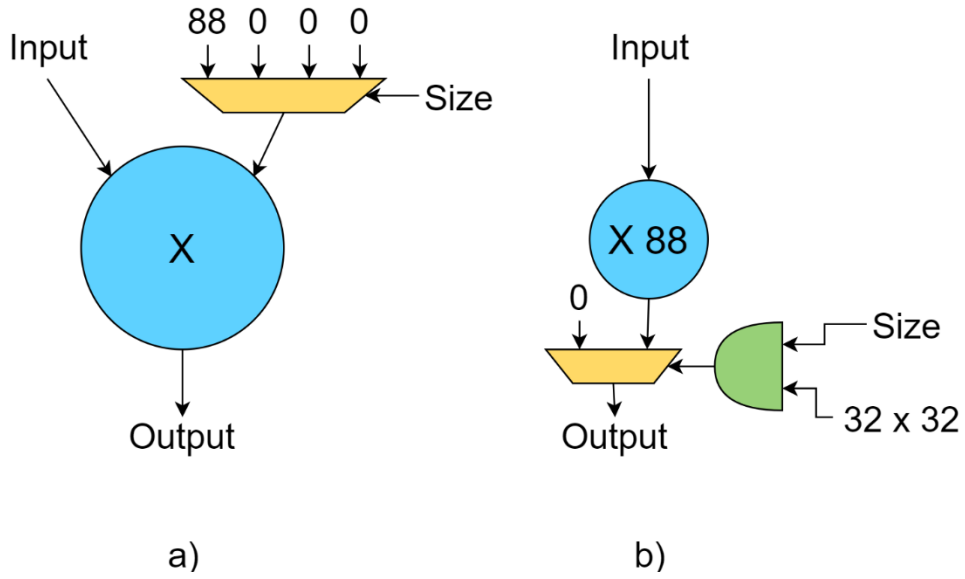

Figure 14. *Example of optimization methods used to reduce the number of DSPs. On case b) DSP is replaced by constant multiplier and mux*

In this work, we have only exploited this approach when three of the four coefficients are zero. Multiplication by a constant is significantly smaller than a full multiplier and it can be implemented in hardware with adders and shifters. This way, we were able to replace 100 DSPs per DCT block with a simple constant multiplier and a mux, like pictured in Figure 14. Alternatively, more DSPs could be saved by using them when there are two or three no-zero confidents, but it would offer notably diminishing returns with even larger logic usage.

5.4.2 4-point DST block

The 4 × 4 luminance TBs are transformed in 4-point DST block that operates in parallel with the 32-point DCT block. The DST block is built of four parallel 1-D row-transform blocks that are connected back to each other in transposed order for a second transform.

The 4 × 4 transpose requires no external components as it is possible to cross-wire the outputs and inputs of the block.

The DST is performed by a fast DST algorithm instead of full matrix multiplication. The same algorithm is used in Kvazaar encoder. The algorithm produces the same result but with less arithmetic operations. Since only one TB size is supported, all multiplications in DST are with constant values. These multiplications are not mapped to DSPs but only logic cells are used instead.

As a small extra coding tool, the block also supports transform skip. As the name suggest, in this mode the transform phase is skipped. This is implemented by forwarding the residual pixels without any operations in the upper half of the 32-coefficient wide output vector.

5.5 Quantization & Dequantization

Quantization block performs both quantization and dequantization of transform coefficients. Although they are different operations, they were implemented in one block because they share the same overall structure and can have a shared control. The Quantization block receives data from the Transform block. It outputs quantized coefficients to the Coefficient Cost block and dequantized coefficients to the Inverse Transform block. The input configuration vector contains scaled QP value used to define the quantization level.

Both quantization and dequantization are done by multiplying coefficients by a scaler value, that is derived from scaled QP values and rounding the output. DSPs were used for multiplications. Even though each DSP has two multipliers, the DSP libraries do not support the use of two multipliers individually. Therefore, the second multiplier was used for rounding. In hardware, rounding is performed by adding one and chopping the result. Rounding with the DSPs was done by adding the rounding factor to the second input of the DSP and one for its multiplier. This way, the end adder in the DSPs is used to add the rounding factor to output of the first multiplier.

The quantization and dequantization operate with individual coefficients so the system can be scaled up to process as many coefficients as needed. With 32 parallel processing units, 32 DSPs were used by the Quantization block and the other 32 by the Dequantization block.

5.6 Inverse Transform

Inverse transform shares the same top-level architecture as forward transform. It is built of three main parts: 1) two 32-point IDCT blocks; 2) a separate 4-point IDST block for 4 × 4 luminance TBs; and 3) the same Transpose block, which is used in forward transform for row-column transpositions between the 1-D IDCT blocks.

The Inverse Transform has been developed along with the forward transform through the same development steps. First, only one 1-D IDCT block was used, but a second one was added later for better performance. The DSP usage in IDCT was also reduced by mapping constant multiplications to logic cells. Due to the differences in transform matrixes, DSP usage could only be reduced to 90 DSP per IDCT compared with 72 in DCT.

The 32-point IDCT block uses the Partial Butterfly algorithm [18] implemented in three-stage pipeline as in the 32-point DCT block. In the first stage, DSPs are used to multiply the input with transform matrices as described in Chapter 5.4.1. A second stage includes three addition/subtraction levels to compose the final even vector from the decomposed even and odd vectors. Lastly, the third stage finalizes the 1-D transform by combining the even and odd vectors and by scaling the final result to 16-bit signed residuals. To support multiple parallel rows with smaller block sizes, the inputs in each stage are reordered to match the structure of the stage. After the first 1-D transform, the intermediate data is transposed in the Transpose block and sent to the second IDCT block to complete the 2-D transform.

In parallel with the 32-point IDCT block, a separate 4-point 2-D IDST block is used for 4×4 luminance CBs. The IDST block performs the full 2-D transform internally without any external transpose. As with the 4-point DST block, IDST uses the fast DST algorithm instead of full matrix multiplication to reduce the number of arithmetic operations needed. The support for transform skip was also added by forwarding the residuals without any operations in the upper half of the output vector.

5.7 Reconstruction

The Reconstruction block receives the reconstructed residual pixels from the Inverse Transform block. It also accesses the original and predicted pixels from the FIFOs from the Prediction Buffer. The block uses residual pixels and a prediction to generate the final reconstruction image as on the decoder side. Original pixels are used to simultaneously calculate the *Sum of Squared Differences (SSD)* value between the reconstruction and the original image.

A reconstruction is calculated by adding the residual to the prediction pixel by pixel. In the case of overflow, the output is clipped to the maximum or minimum value. Pixels have no dependencies with each other, so as many pixels as needed can be calculated in parallel. The input from the Inverse Transform contains 32 coefficients and the Reconstruction block was built for that.

With the smallest 4×4 luminance CBs, an input vector from the Inverse Transform block contains two CBs. The lowest half contains a normal CB and the top half contains the

respective transform skip candidate for it. In reconstruction calculations, an ability to duplicate the lower half of the prediction to the upper half was added to cover this special case.

SSD is used as an image quality metric in the Intra coding control block since the encoder selects the best configuration as a function of the image quality and the number of consumed bits. Estimates for the coding cost are calculated later in the Coefficient Cost block, which is covered in the next section.

SSD is calculated by deducting reconstruction from the original image, squaring the differences in pixel values, and adding them all together. The 4×4 luminance CBs require SSD to be calculated in two halves, as separate SSD values are needed for both CBs. For other CBs, the full SSD is produced by adding the two halves together. As an output, all three values, the two halves and the combined sum, with the reconstruction image are sent to the CU Stack.

Squaring operation in SSD calculations requires multipliers that were manually mapped to DSPs in the DSP library. In this case, multiply-accumulate mode was used. With two input ports, the DSPs calculate SSD in sixteen parts which are combined to the required three SSD values by summing DSP outputs together.

5.8 Coefficient Cost

The Coefficient Cost block calculates the coding cost of the CB in the encoded bitstream. Kvazaar software uses actual CABAC coding method to calculate the exact number of consumed bits. However, the CABAC is a complex operation with numerous steps. It requires a lot of extra information about the states of the CTU and CB being coded. Using CABAC in hardware is impractical as it would require all this extra data to be transferred. In addition, the operations in CABAC are serial in nature and it would be hard to accelerate it in hardware. For these reasons, a more hardware-friendly algorithm was needed to estimate the cost of coding CBs.

A new, more hardware-friendly linear algorithm was provided by the Kvazaar software team. The algorithm uses five different parameters found from quantized transform coefficients: total sum of coefficients, number of non-zero coefficient groups, number of coefficients with value of zero or one, and the index number of the last non-zero coefficient group. Different weights are predefined for each parameter and for each CB size. The final cost estimation is calculated by multiplying each parameter with its weight and added together. Analysis of the algorithm is out of the scope of this Thesis, but it is known to produce slightly worse results than CABAC as it only estimates the cost coding.

Although the new algorithm was more hardware friendly, it still had a few impracticalities for hardware implementation. For that, the algorithm was converted from floating point

to fixed point format. With fourteen decimal bits, errors were limited to occasional off-by-one errors. As the floating point model was already an estimate, the small errors were acceptable.

Another issue is data availability. In Kvazaar software, all data is ready at the beginning of the algorithm and easily accessible in any order. To perform better in hardware, the algorithm was updated to operate in streaming mode. Instead of all data being available all the time, data is processed in slices of fixed size. As the data was originally processed in order of the coefficient groups, extra operations were required to find order and group numbers for all coefficients. These were implemented with lookup tables mapped to read only memories in hardware.

The input coming from the Quantization block is in transposed order due to the transpose in the Transform phase. The order had no effect on the quantization phase and was ignored there. However, the Coefficient Cost block requires data in natural order. To transpose the input, an extra Transform block was added between the Quantization block and the Coefficient block. The lookup tables in the algorithm could also be modified to support transpose order but the transpose is still required sooner or later.

Like in software, the hardware implementation of the Reconstruction block processes the smallest 4×4 luminance CBs in pairs: normal and transform skip. For this reason, the Coefficient Cost block calculates the cost values for the two halves independently. Unlike the SSD computation on the Reconstruct block, combining two cost values was non-trivial and required extra operations as the group numbers are mixed.

5.9 CU Stack

The CU Stack is an output buffer where CUs are stored temporarily before writing them to output memory. Buffering is needed because the final CU configuration can only be determined in the end after comparing all options. The other option would be to regenerate the selected CUs, but it would introduce extra overhead on the coding pipeline and reduce the system overall throughput.

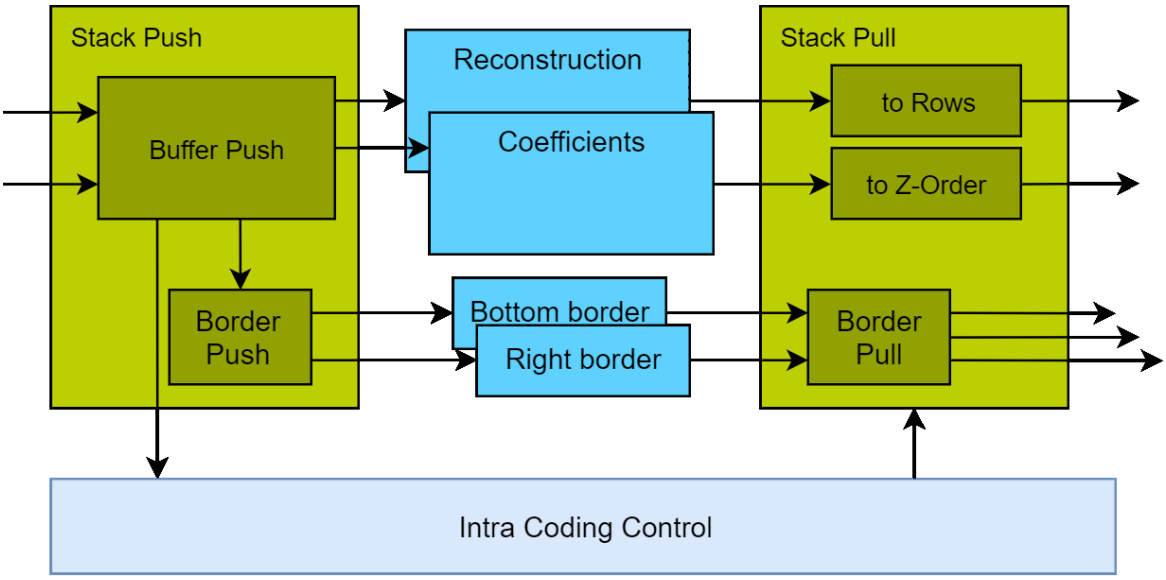

Figure 15. *Block diagram of CU Stack*

The CU Stack is pictured in Figure 15. It is built from three sub-blocks: Stack Push, Stack Pull and intermediate memory modules between them. The push and pull are hierarchical HLS blocks with their own sub-blocks. The intermediate memories are instantiated in top level RTL design. The hierarchical structure is used to make both parts operate in parallel without either one blocking the other one.

The buffers store one CU of each size for 16 CTUs. This way, all CUs have a reserved buffer slot assigned by their size and CTU id. The next co-located CU of the same size in the same CTU will overwrite the old one in the buffer. This policy follows the computational order of the CTU quadtree. The CUs are either flushed or discarded before moving to the next one of the same size.

5.9.1 Stack Push

The Stack Push block is located in the intra coding pipeline. It receives reconstruction pixels and the SSD values from the Reconstruction block. In addition, it gets quantized transform coefficients and cost values from the Coefficient Cost block. It writes the pixels and coefficients to the reserved slots in the buffer memory and simultaneously collects pixels of the bottom and right borders to the separate vectors. After all pixels and coefficients are written, the border vectors are sent to a hierarchical sub-block that writes them to the corresponding buffers. Lastly, the SSD and cost values are sent to the Intra coding control block to signal the completion of the CB processing.

The Stack Pull block is in the other side of the buffer memories. It has no direct connection to the Stack Push and it receives its configuration data from the Intra coding control. The control block instructs the stack in two cases: 1) when processing in the CTU quadtree moves up and a larger CU is selected; and 2) after generating CBs of the smallest

configured size. The first case is issued by a CMP instruction and the control block selects a CU configuration from two or more possible combinations. The second case is about automatic flushing of the smallest CBs. It is required since the borders of the CB must be in the memory before processing of the next one can begin. Automatic flushing also offers a small performance boost in cases where the smallest CUs are selected since they are already in output. In other cases, they will be overwritten by larger CUs.

5.9.2 Stack Pull

After the control block has decided the CUs to the output, it sends the Stack Pull block the respective instructions. The pull block starts reading the CU from the main buffer and writing it into the output memory. While writing the CU, the pull block performs two additional operations: transforming reconstruction from slices to rows and reordering coefficients to Z-order.

Figure 16. *Slices to columns transform for 4 x 4 CB*

Reconstructions in the buffer are stored as slices of 32 pixels. The output memory is an array of 64 × 64 pixels. A 32-pixel wide data bus maximizes speed when writing the largest CUs. The large data bus requires smaller CUs to be shifted to a correct location and byte enables to assign writes to the correct pixels, as pictured in Figure 16. Catapult-C supports byte enabled memory interfaces, but it is limited to four enable signals. In a 32-pixel wide bus, a minimum of 8 byte-enable signals are needed with 4 × 4 CUs. A workaround for this is to add an extra output signal HLS model along with the normal memory interface, connect it to byte-enable port of the memory and manually time it to match memory writes.

The coefficients are written to the bitstream in z-order which is also used as the format of the output memory. Compared with slices-to-rows transform, z-ordering is a much simpler operation. The Z coordinate is calculated from x and y coordinates and coefficients

are written to the consecutive addresses. Only one byte enable signal is needed to mask the write of 4 × 4 CUs.

A separate Border Pull block is implemented to write right and bottom borders to the border memory which is used by the Get Border block. Extra buffers for the borders accelerate the transfer. They are needed by the Intra Prediction block when generating the following CUs. There is no handshaking between the buffer and the intra prediction phase since buffer pixels are written out as fast as possible. In addition, an extra memory is used to store every fourth pixel from both borders, i.e., the bottom right-most pixel from every 4 × 4 block. The bottom right-most pixels are duplicated to a third memory to cover special cases where they are overwritten in normal border memories.

5.10 Transpose

The Transpose block performs row-column transpose for a square $N \times N$ block, where $N \in \{4, 8, 16, 32\}$. It operates on a constant data rate of 32 samples per cycle. The Transpose is used as a sub-block in three different locations: between the 1-D transforms in Transform and Inverse Transform blocks as well as in between the Quantization and Coefficient Cost blocks.

The Transpose block is built of three sub-blocks: Transpose Push, Transpose Pull and a storage array between them. The push block writes data to the memory array and the pull block reads data from the array. The memory array is a collection of 32 parallel memory modules with individual read and write ports.

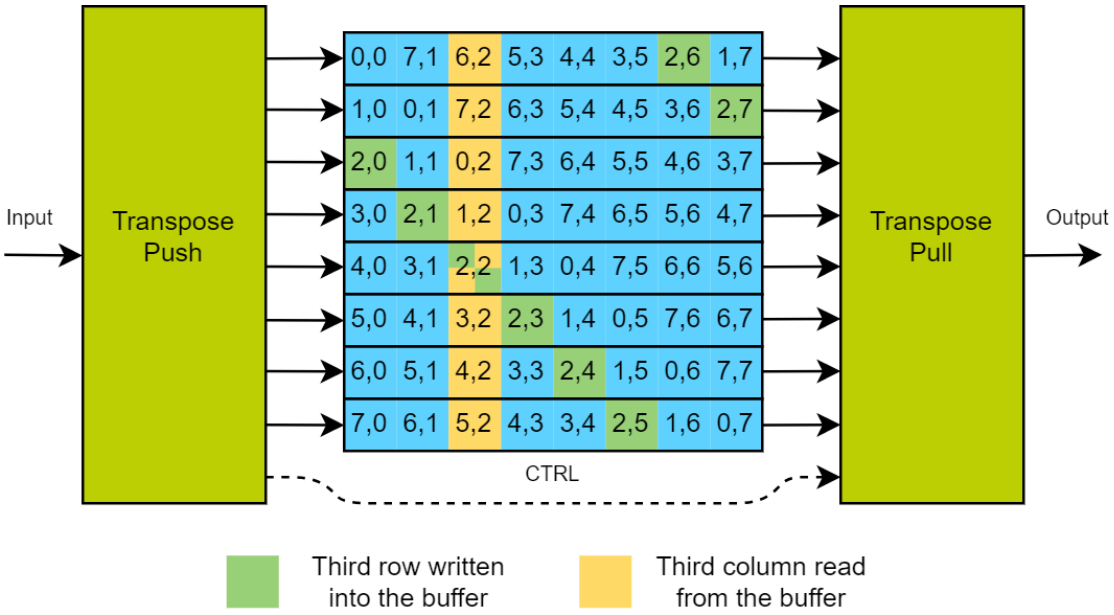

Figure 17. *Visualization of pixel placement in 8 × 8 variant of transpose block.*

A 8 × 8 sample of the used transpose block is pictured in Figure 17. It shows how samples are placed in the memory array. Numbers in each cell represent x and y coordinates of an 8 × 8 block when all samples are written into the memory.

5.10.1 Transpose Push

The input to the Transpose block goes to the push block that receives a slice of 32 samples per cycle. Depending on the block size, the slice contains from one to eight rows as the slice can contain up to two 4 × 4 blocks. Before writing data into the memory array, the push block has to move samples to correct memory modules and calculate the write addresses for each memory. To move samples, the input slice is rotationally shifted left by its index number, i.e., the first slice is not shifted at all, the second slice is shifted by one place, the third one by two places, and so on.

Simultaneously, the write addresses are calculated by deducting the slice numbers from the column numbers of each sample. Using the column number causes writes to happen diagonally and reducing the slice number shifts writes left. If the slice contains multiple columns the addresses are bit masked to keep values in range. Lastly, an offset value is added to each address. The offset value moves blocks forward and makes the memory to operate as a circular buffer. This allows the Transpose block to operate as a FIFO buffer. After the whole block is written to the memory, the pull block is notified that the memory is ready, and it can start reading the data.

5.10.2 Transpose Pull

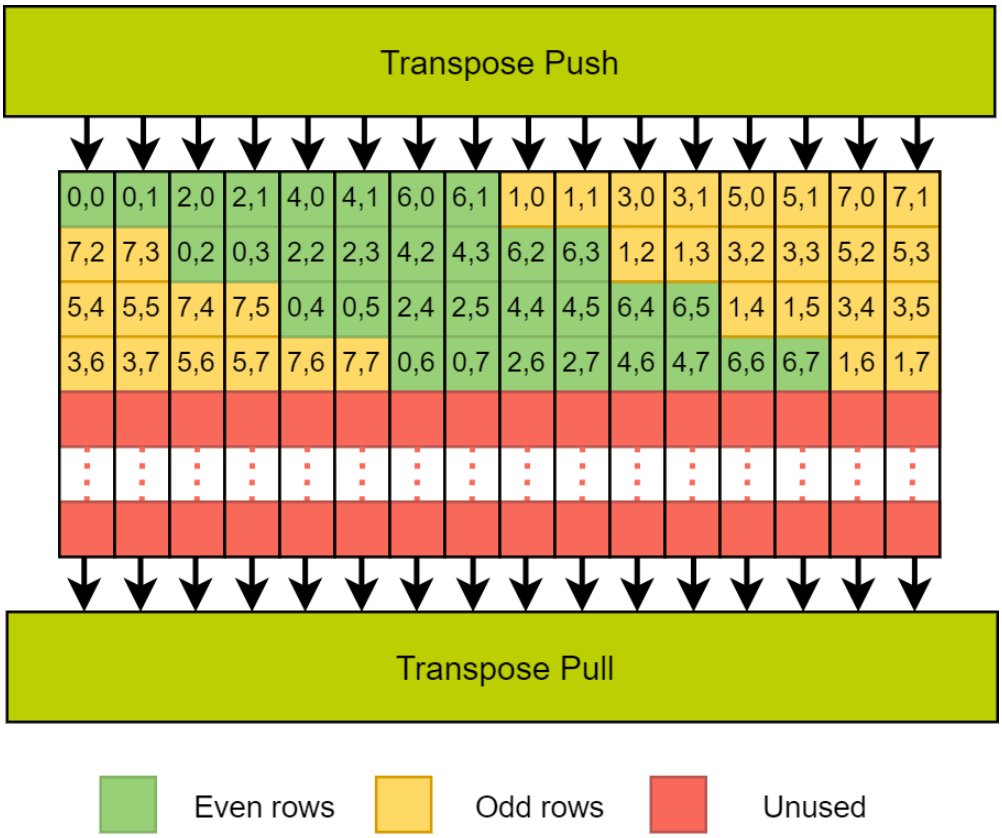

Figure 18. *Transposing a slice containing multiple rows (8 × 8 block in 16 × 16 transpose block)*

The pull block is less complex, and it does not require individual read addresses for each memory. Instead, one read address is enough for all memories since the reads are directed to the consecutive addresses. The slices were shifted left in the push block, so the pull block reverses this operation by shifting each slice right by its index number. The last step before outputting the transposed slice is sample reordering. When a slice contains only one row no reordering is required as seen in Figure 17. But, when a slice contains multiple rows the output is mixed and reordering is needed as seen in Figure 18. As the order only depends on the block size, the reordering is simply implemented as a four-port mux.

6. ANALYSIS

This chapter shows the synthesis results of the FPGA accelerator, performance numbers and compares them to related works.

6.1 Synthesis results

Table 2. *Synthesis summary of the accelerator with three Intra coding accelerators.*

Synthesis Summary	
Family	Arria 10
Device	10AX115N3F40E2SG
Logic utilization (ALMs)	346 342 / 427 200 (81 %)
Total registers	427 569 / 854 400 (51 %)
Total pins	35 / 826 (4 %)
Total block memory bits	21,566,288 / 55,562,240 (39 %)
Total M20K Blocks	2 713 / 2 713 (100 %)
Total DSP Blocks	1 227 / 1 518 (81 %)
Total HSSI RX channels	8 / 48 (17 %)
Total HSSI TX channels	8 / 48 (17 %)
Total PLLs	50 / 112 (45 %)
ALMs used for memory	14 180
Timing Analyzer Fmax 0 ℃	225,65 MHz
Timing Analyzer Fmax 100 ℃	186,22 MHz
Quartus Prime Version	17.1.0 Build 590 SJ Standard Edition

Our design with three Intra coding accelerators was synthesized into an GT 1150 class Arria 10 FPGA [27]. The summary of the synthesis results is displayed in Table 2. A complete design utilizes over 346k *Adaptive Logic Modules (ALMs)*, i.e., it consumes more than 80 % of the available logic. Likewise, over 80 % of the available DSPs and over half of the registers were used. Notably, all M20K memory modules [27] were utilized so some of the memories were mapped to ALMs. Quartus reports that extra 14k ALMs were used instead of memory bits due to the lack of available M20K controllers, but this number only covers ALMs used as memory. With the memory controllers, the actual consumption is close to 18k ALMs.

The *High-speed serial interface (HSSI)* channels and most of the *phase-locked loops (PLLs)* are used by the PCIe interface. The accelerator uses only one clock, so it needs only a single PLL. Quartus reports 186 MHz at 100 ℃ and up to 225 MHz at 0 ℃ for the maximum operating frequency. These numbers are higher than the 175 MHz target frequency, at which the system was verified to function properly.

Table 3. *Block-level synthesis results of the one Intra Coding Accelerator*

	ALMs	Registers	Memory Bits	M20K	DSPs
Total	108 802	133 635	3 476 096	663	409
Intra coding control	7 114	10 598	311 936	21	5
Initialization	219	264	0	0	0
Scheduler	1 032	1 014	0	0	0
Start	166	301	0	0	0
End	5 625	8 945	0	0	5
Intra Prediction	26 017	43 783	1 361 920	454	0
Get Border	536	703	0	0	0
Control	399	470	0	0	0
Planar	476	482	4 096	2	0
DC	137	232	4 096	2	0
Zero Angle	170	260	0	2	0
Negative Angle	938	1 136	26 624	13	0
Positive Angle	622	755	53 248	13	0
SAD	4 351	7 964	0	0	0
Prediction Buffer	8 337	19 595	589 824	231	0
Transform	26 107	31 761	65 536	32	144
32-point DCT	9 135	11 866	0	0	72
4-point DST	2 910	2 266	0	0	0
Transpose	3 617	2 256	65 536	32	0
Quantization	5 035	4 148	0	0	64
Inverse Transform	28 023	30 035	65 536	32	180
32-point IDCT	9 777	10 470	0	0	90
4-point IDST	3 154	2 355	0	0	0
Transpose	3 613	2 269	65 536	32	0
Reconstruction	1 315	2 052	0	0	16
Coefficient Cost	4 733	3 807	196 608	45	0
Transpose	3 681	2 398	196 608	45	0
Get Cost	1 029	1 370	0	0	0
CU Stack	3 430	5 819	1 605 632	79	0
Stack Push	1 619	3 588	0	0	0
Stack Pull	1 811	2 231	0	0	0

The shared PCIe interface consumes 4k ALMs and 5k registers. With each accelerator, input and output memories take 367 Kbits of memory and 291 M20K memory controllers. In total, 64 controllers are used for reference/reconstruction images and 104 for transform coefficients. High M20K block usage comes from the need to store 16 CTUs for parallel processing and wide data busses required for fast access. With three accelerators, the input and output memories consume in total 1.1 Mbits of memory and 883 M20Ks. Each Intra coding accelerator takes around 108k ALMs, 130k registers, 3.5 Mbits of memory, 663 M20K, and 409 DSPs in optimal conditions.

The distribution of resources in the Intra coding accelerator are displayed in Table 3. The largest blocks of the design are the intra prediction and both transforms, each one taking around a quarter of the used logic. The remaining 25 % of the logic is divided between the control, quantization, reconstruction, coefficient cost and CU stack.

The intra prediction also takes over two third of the used M20K controllers. Around half of them is used for parallel access in the prediction blocks and the other half in the Prediction Buffer. Out of the 16 angled prediction blocks, 15 requires 13 M20Ks each and last one for the mode 18 requires 8 M20Ks. This requirement results from the random-access pattern, which prevents exploiting larger data widths. The high controller usage in the Prediction Buffer results from the opposite problem. The M20K has a maximum data width of 40 bits [27] and memory buses larger than that are implemented with parallel controllers. The super wide 9216-bit (36 × 256-bit) memory interface of the Prediction Buffer requires 231 memory controllers. FIFOs for reference picture and prediction are also located within the intra prediction block requiring 7 M20Ks each. Aside from the intra prediction, memories are mainly used in the CU stack and transpose blocks.

Most of the used DSP blocks are taken by the double DCT and IDCT blocks in the transform blocks. After optimizations, one DCT takes 72 DSPs and one IDCT takes 90 DSPs. This leaves 291 DSPs unused in the device. They could be used in the DST or IDST blocks or in the prediction blocks. Using more DSPs could save some logic but could also negatively affect timing due to routing challenges.

6.2 Performance

Table 4. *Test-PC setup*

Test PC	
CPU	Intel E5-2699 v4 (22 × 2.2 GHz)
FPGA	2 × Arria 10 Nallatech 385A
Memory	64 GB DDR4 ECC 2400 MHz
Storage	512 GB SATA 3.0 SSD
OS	Ubuntu 16.4
Encoder	Kvazaar v.1.2.0

The synthesized FPGA image was flashed into two Arria 10 FPGA on Nallatech 385A PCIe cards that were added to test PC. The test-PC is powered by a 22-core Intel E5-2699 v4 Xeon CPU running at 2.2 GHz. The CPU is paired with 64 GB of EEC DDR4 RAM and a fast 512 GB solid-state drive. Ubuntu 16.04 LTS was used as an operating system.

Table 5. *Kvazaar coding configuration*

Feature	Ultrafast w/ 32 × 32
Profile	Main
Bit depth	8
Color format	4:2:0
Coding mode	Intra
Coding unit sizes	32 × 32, 16 × 16 and 8 × 8
Prediction unit sizes	32 × 32, 16 × 16 and 8 × 8
Transform unit sizes	32 × 32, 16 × 16 and 8 × 8
IP modes	All (Planar, DC and 33 Angular)
Intra search	Full
Transform	Integer DCT and DST
Mode decision	Sum of Absolute Differences
Parallelization	Wave-front, Tiles
SAO	Disabled
Sign hide	Disabled
Rate Control	Disabled
RDO	Disabled
RDOQ	Disabled

Encoding was benchmarked with the modified version of Kvazaar v1.2.0 using ultrafast preset with 32 × 32 CUs. The 32 × 32 CUs were enabled as their effect on coding speed on hardware is negligible. The other coding tools and settings are displayed in Table 5. In this configuration, the hardware accelerator handles most of the work and the software processing is limited to entropy coding and data handling with picture-level parallelism. Using other presets would introduce coding tools not implemented on hardware so they should be handled by the software. It would increase the CPU load and performance would be limited by the CPU. With ultrafast setting, the 22-core Xeon is capable of full hardware utilization with two FPGAs and performance is only limited by the hardware.

The performance of the accelerator was tested with 8-bit UHD 4:2:0 YUV test sequences shared by Ultra Video Group [28]. The sequences contain different video content from static content to slowly and fast moving objects and views.

The performance tests were run with six different hardware configurations including one or two FPGA cards with one to three accelerators per FPGA. For all configurations, each test sequence was encoded three times and the results were averaged to get rid of random variation.

Table 6. HEVC coding speed of UHD video with different number of Intra coding accelerators

Sequence (2160p)	Software	Single FPGA, accelerators			Two FPGAs, accelerators		
		1	2	3	2	4	6
	Speed (fps)	Speed (fps)	Speed (fps)	Speed (fps)	Speed (fps)	Speed (fps)	Speed (fps)
Beauty	17	25	49	64	50	96	125
Bosphorus	20	27	53	65	54	102	127
HoneyBee	17	26	50	64	51	98	124
Jockey	21	29	54	65	58	104	126
ReadySetGo	19	27	52	64	53	99	123
ShakeNDry	16	22	44	63	45	85	115
YachtRide	18	26	51	64	51	98	123
Average	18	26	50	64	52	97	123

Average coding speed of the software only encoding was 18 *frames per second (fps).* With one FPGA and one accelerator, the speed was increased by 45 % to 26 fps. The relatively low performance increase is caused by hardware limitation of 16 parallel CTUs. If there are less hardware slots than CPU processing threads, some threads stall while waiting for hardware. With less powerful CPU, the relative performance increase would be higher.

With two accelerators, coding performance was increased by 175 % to over 50 fps. There is no difference between two accelerators split into one or two FPGAs as the PCIe interface is capable of handling data transfers for 32 CTUs.

With more than two accelerators, the system scales somewhat linearly up to 585 % performance boost or 123 fps with 6 accelerators on two FPGAs. Performance increase slows down with 3 accelerators on a single FPGA as the data rate on the PCIe interface starts reaching the maximum supported data rate.

6.3 Comparison to Related Work

Table 7. *Comparison of the designed and prior-art HEVC intra encoders*

Architecture	Technology	Frequency	Performance	Cells	DSPs
Zhu et al. [29]	TSMC 90 nm	357 MHz	1080@44 fps	2 296k gates	-
Pastuszak et al. [30]	TSMC 90 nm	200/400 MHz	2160@30 fps	1 086k gates	-
Pastuszak et al. [30]	Arria II	100/200 MHz	1080@30 fps	93k ALUTs	481
Atapattu et al. [31]	Zyng ZC706	140 MHz	1080@30 fps	84k LUTs	34
Miyazawa et al. [32]	3x FPGA	N/A	1080@60 fps	N/A	N/A
This Thesis	CPU + Arria 10	175 MHz	2160@64 fps	552k ALUTs	1227

Table 7 tabulates the characteristics of the encoder developed in this work and existing HEVC intra encoders on *Application specific integrated circuits (ASICs)* and FPGAs. For more straightforward comparison, our accelerator is configured to use only one FPGA.

From the related HEVC encoders, the real-time coding speed of the ASIC-based HEVC intra encoder in [29] is limited to 1080p at 44 fps. The HEVC intra encoder in [30] supports real-time 2160p video encoding on ASIC but the respective FPGA implementation is limited to 1080p resolution. Similarly, the FPGA-based HEVC intra encoder in [31] is restricted to 1080p resolution. The intra/inter HEVC encoder in [32] is able to encode 1080p at 60 fps with a custom board of three separate FPGA chips. Higher resolutions are also supported but not without increasing the number of boards and chips.

The accelerator developed in this work is the fastest one of the compared encoders. It more than doubles the performance over that of the second fastest. Miyazawa et al. do not disclose the size of their encoder but compared with others, our accelerator is the largest one and uses more DSPs than the others. However, comparing *Look-up tables (LUTs)*, *Adaptive look-up tables (ALUTs)*, ALMs and ASIC gates is difficult because of different technologies. Our encoder also requires a high-performance CPU to function effectively.

7. CONCLUSION

The main objective of this Thesis was to accelerate HEVC encoding with an FPGA hardware accelerator. The accelerator was implemented using Catapult-C HLS design flow instead of a traditional RTL approach. HLS offered notably faster process from software model to the hardware implementation. This made design and verification times more acceptable for project of this size.

To implement the accelerator, multiple different coding tools from HEVC were implemented with HLS. The most notable blocks of the accelerator were the intra prediction and transforms blocks. A fully custom control structure was also developed for the accelerator to enable its parallelization. Although some parts of the accelerator were developed before the start of this Thesis, all parts were at least improved during this work. Either resource consumption was optimized or performance increased.

The hardware - software interface was also developed for the accelerator. On hardware, the interface was implemented with Intel PCIe block using custom DMAs and memory structure. On software side, the designed interface included a custom Linux kernel module and a modified Kvazaar software interacting with the accelerator.

The accelerator achieved an encoding speed of 64 fps with one FPGA and 123 fps with two FPGAs for 2160p video. Compared with related works, the designed accelerator achieved the fastest coding speeds, although also being the largest.

To increase encoding speed further, CABAC and others coding tools should also be implemented on FPGA. This way, the accelerator could be converted to a standalone encoder that does not require a separate CPU.

REFERENCES

[1] Cisco, *Cisco Visual Networking Index*: Forecast and Methodology, 2015-2020, Jun. 2016.

[2] *High Efficiency Video Coding*, document ITU-T Rec. H.265 and ISO/IEC 23008-2 (HEVC), ITU-T and ISO/IEC, Apr. 2013.

[3] *Advanced Video Coding for Generic Audiovisual Services*, document ITU-T Rec. H.264 and ISO/IEC 14496-10 (AVC), ITU-T and ISO/IEC, Mar. 2009.

[4] M. Viitanen, J. Vanne, T. D. Hämäläinen, M. Gabbouj, and J. Lainema, "Complexity analysis of next-generation HEVC decoder," in *Proc. IEEE Int. Symp. Circuits Syst*., May 2012.

[5] *Kvazaar HEVC encoder* [Online]. Available: https://github.com/ultravideo/kvazaar

[6] M. Fingeroff, *High-Level Synthesis Blue Book*. Mentor Graphics Corporation, 2010.

[7] *Catapult: Product Family Overview* [Online]. Available: http://calypto.com/en/products/catapult/overview

[8] *Synphony C Compiler* [Online]. Available: https://www.synopsys.com/implementation-and-signoff/rtl-synthesis-test/synphony-c-compiler.html

[9] *Vivado High-Level Synthesis* [Online]. Available: https://www.xilinx.com/products/design-tools/vivado/integration/esl-design.html

[10] *Intel® High Level Synthesis Compiler* [Online]. Available: https://www.intel.com/content/www/us/en/software/programmable/quartus-prime/hls-compiler.html

[11] *HM (HEVC Test Model)* [Online]. Available: https://hevc.hhi.fraunhofer.de/

[12] *Arria V Device Overview* [Online]. Available: https://www.intel.com/content/dam/www/programmable/us/en/pdfs/literature/hb/arria-v/av_51001.pdf

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

[14] J. Vanne, M. Viitanen, T. D. Hämäläinen, and A. Hallapuro, "Comparative rate-distortion-complexity analysis of HEVC and AVC video codecs," *IEEE Trans. Circuits Syst. Video Technol*., vol. 22, no. 12, Dec. 2012, pp. 1885-1898

[15] *H.261: Codec for audiovisual services at n x 384 kbit/s*, document ITU-T Rec. H.261, Nov. 1988.

[16] J. Lainema, F. Bossen, W. J. Han, J. Min, and K. Ugur, "Intra coding of the HEVC standard," *IEEE Trans. Circuits Syst. Video Technol*., vol. 22, no. 12, Dec. 2012, pp. 1792-1801.

[17] A. Fuldseth, G. Bjøntegaard, M. Budagavi, and V. Sze "Core transform design for HEVC," *Document JCTVC-G495*, Geneva, Switzerland, Nov. 2011.

[18] M. Budagavi, A. Fuldseth, G. Bjøntegaard, V. Sze, and M. Sadafale, "Core transform design in the High Efficiency Video Coding (HEVC) standard," *IEEE J. Select. Topics Signal Process*., vol. 7, no. 6, Dec. 2013, pp. 1029-1041

[19] L. Braatz, L. Agostini, B. Zatt, and M. Porto, "A multiplierless parallel HEVC quantization hardware for real-time UHD 8K video coding", *in Proc. IEEE International Symposium on Circuits and Systems*, Baltimore, Maryland, USA, May 2017.

[20] A. Saxena and F. C. Fernandes, "DCT/DST-Based transform coding for intra prediction in image/video coding," *IEEE Transactions on Image Processing*, vol. 22, no. 10, pp. 3974-3981, Oct 2013.

[21] *Arria 10 Hard IP for PCI Express IP Cores* [Online]. Available: https://www.intel.com/content/dam/www/programmable/us/en/pdfs/literature/ug/ug_a10_pcie.pdf

[22] *Avalon® Interface Specifications* [Online]. Available: https://www.intel.com/content/dam/www/programmable/us/en/pdfs/literature/manual/mnl_avalon_spec.pdf

[23] P. Sjövall, J. Virtanen, J. Vanne, and T. D. Hämäläinen, "High-level synthesis design flow for HEVC intra encoder on SoC-FPGA," *in Proc. Euromicro Symp. Digit. Syst. Des*., Funchal, Madeira, Portugal, Aug. 2015.

[24] P. Sjövall, V. Viitamäki, A. Oinonen, J. Vanne, T. D. Hämäläinen, and A. Kulmala, "Kvazaar 4K HEVC intra encoder on FPGA accelerated Airframe server," *in Proc. IEEE Workshop Signal Process. Syst*., Lorient, France, Oct. 2017.

[25] *Intel Arria Native Fixed Point DSP IP Core User Guide* [Online]. Available: https://www.intel.com/content/www/us/en/programmable/documentation/kly1418710866787.html

[26] *Designing High-Performance DSP Hardware - Using Catapult C Synthesis and the Altera Accelerated Libraries* [Online]. Available: https://www.intel.com/content/dam/www/programmable/us/en/pdfs/literature/wp/wp-01039.pdf

[27] *Intel® Arria® 10 Device Overview* [Online]. Available: https://www.intel.com/content/dam/www/programmable/us/en/pdfs/literature/hb/arria-10/a10_overview.pdf

[28] *Test Sequences* [Online]. Available: http://ultravideo.cs.tut.fi/#testsequences

[29] J. Zhu, Z. Liu, D. Wang, Q. Han, and Y. Song, "HDTV1080p HEVC Intra encoder with source texture based CU/PU mode pre-decision," *in Proc. Asia and South Pacific Design Automation Conf.*, Singapore, Jan. 2014.

[30] G. Pastuszak and A. Abramowski, "Algorithm and architecture design of the H.265/HEVC intra encoder," *IEEE Trans. Circuits Syst.* Video Technol., vol. 26, no. 1, Jan. 2016, pp. 210-222.

[31] S. Atapattu, N. Liyanage, N. Menuka, I. Perera, and A. Pasqual, "Real time all intra HEVC HD encoder on FPGA," *in Proc. IEEE Int. Conf. on Application-specific Syst., Architectures and Processors*, London, Jul. 2016, pp. 191-195.

[32] K. Miyazawa, H. Sakate, S. Sekiguchi, N. Motoyama, Y. Sugito, K. Iguchi, A. Ichigaya, and S. Sakaida, "Real-time hardware implementation of HEVC video encoder for 1080p HD video," *in Proc. Picture Coding Symposium*, San Jose, California, USA, Dec. 2013, pp. 225-228.