Tampere University

PANU SJÖVALL

Feasibility Study of High-Level Synthesis

Implementation of a Real-Time HEVC Intra Encoder on FPGA

Tampere University Dissertations 751

Tampere University Dissertations 751

PANU SJÖVALL

Feasibility Study of High-Level Synthesis

Implementation of a Real-Time HEVC Intra Encoder on FPGA

ACADEMIC DISSERTATION
To be presented, with the permission of
the Faculty of Information Technology and Communication Sciences
of Tampere University,
for public discussion in the auditorium TB109
of Tietotalo, Korkeakoulunkatu 1, Tampere,
on 17 March 2023, at 12 o'clock.

ACADEMIC DISSERTATION
Tampere University, Faculty of Information Technology and Communication Sciences
Finland

Responsible supervisor and Custos	Associate Professor Jarno Vanne Tampere University Finland	
Supervisor	Professor Timo Hämäläinen Tampere University Finland	
Pre-examiners	Associate Professor Maxime Pelcat INSA Rennes, IETR France	Dr. Ercan Kalali Eindhoven University of Technology Netherlands
Opponents	Associate Professor Maxime Pelcat INSA Rennes, IETR France	Dr.-Ing. Christian Herglotz Friedrich-Alexander University Erlangen-Nürnberg Germany

Cover design: Roihu Inc.

ISBN 978-952-03-2775-0 (print)
ISBN 978-952-03-2776-7 (pdf)
ISSN 2489-9860 (print)
ISSN 2490-0028 (pdf)
http://urn.fi/URN:ISBN:978-952-03-2776-7

HILJAISUUTTAALI PAINOTUOTE
ClimateCalc CC-000025/FI
PunaMusta Printing

Carbon dioxide emissions from printing Tampere University dissertations have been compensated.

PunaMusta Oy – Yliopistopaino
Joensuu 2023

PREFACE

The research for this thesis was conducted in *Tampere University of Technology* (*TUT*) Department of Pervasive Computing and Laboratory of Pervasive Computing during 2014-2018 and in *Tampere University* (*TAU*) Computing Sciences during 2019-2021. Although the name of the university and the unit has changed during this journey, I have had the pleasure to be a part of the *Ultra Video Group* (*UVG*), led by Assoc. Prof. Jarno Vanne, the whole time. I received my M.Sc, degree in December 2015 and started pursuing for a PhD in August 2016. I was lucky enough that I could continue the same research during my doctoral studies that I started with my M.Sc. The research was thus a direct continuation of my master's thesis titled "High-Level Synthesis of HEVC Intra Prediction on FPG". This thesis describes the fully HLS implemented HEVC intra video encoder on FPGA.

I would like to express my deepest gratitude to Assoc. Prof. Jarno Vanne and Prof. Timo D. Hämäläinen for giving me the opportunity to work at the university, and for the guidance and collaboration with practically all of the publications. Also, a big thanks to all other co-authors collaborating to the research and publications, including Vili Viitamäki, Arto Oinonen, Mikko Teuho, Ari Lemmetti, Sakari Lahti, Janne Virtanen, and Ari Kulmala. A special thanks goes to my supervisor Assoc. Prof. Jarno Vanne for the supervision of my thesis and for all the help during my doctoral studies.

I would also like to thank my colleagues working in the same office room as I did, Esko Pekkarinen, Mikko Teuho, and Vili Viitamäki. Thank you for all the discussions (random and work related), for the company during lunchtime and coffee breaks, and for the occasional game of chess. These have been an essential part of work happiness and provided a small break from work when needed. It helped keeping the motivation high and reaching this final goal.

Finally, I would like to thank my family for all the support during this process.

I would have never gotten this far with my academic career without you. It is a privilege to come home from work to a loving family. Even though I had some doubts about working remotely from home during the COVID-19 situation at first, it proved to be very easy and pleasant. Even the writing of this thesis was done fully at home. Working from home has also made it possible to spend even more time with you.

Panu Sjövall
11.11.2022 Ylöjärvi

ABSTRACT

High-Level Synthesis (*HLS*) is an automated design process that seeks to improve productivity over traditional design methods by increasing design abstraction from *register transfer level* (*RTL*) to behavioural level. Various commercial HLS tools have been available on the market since the 1990s, but only recently they have started to gain adoption across industry and academia. The slow adoption rate has mainly stemmed from lower *quality of results* (*QoR*) than obtained with conventional *hardware description languages* (*HDLs*). However, the latest HLS tool generations have substantially narrowed the QoR gap.

This thesis studies the feasibility of HLS in video codec development. It introduces several HLS implementations for *High Efficiency Video Coding* (*HEVC*), that is the key enabling technology for numerous modern media applications. HEVC doubles the coding efficiency over its predecessor *Advanced Video Coding* (*AVC*) standard for the same subjective visual quality, but typically at the cost of considerably higher computational complexity. Therefore, real-time HEVC calls for automated design methodologies that can be used to minimize the HW implementation and verification effort.

This thesis proposes to use HLS throughout the whole encoder design process. From data-intensive coding tools, like intra prediction and discrete transforms, to more control-oriented tools, such as entropy coding. The C source code of the open-source Kvazaar HEVC encoder serves as a design entry point for the HLS flow, and it is also utilized in design verification. The performance results are gathered with and reported for *field programmable gate array* (*FPGA*).

The main contribution of this thesis is an HEVC intra encoder prototype that is built on a Nokia AirFrame Cloud Server equipped with 2.4 GHz dual 14-core Intel Xeon processors and two Intel Arria 10 GX FPGA Development Kits, that can be connected to the server via *peripheral component interconnect express* (*PCIe*) generation 3 or 40 Gigabit Ethernet. The proof-of-concept system achieves real-time

4K coding speed up to 120 fps, which can be further scaled up by adding practically any number of network-connected FPGA cards.

Overcoming the complexity of HEVC and customizing its rich features for a real-time HEVC encoder implementation on hardware is not a trivial task, as hardware development has traditionally turned out to be very time-consuming. This thesis shows that HLS is able to boost the development time, provide previously unseen design scalability, and still result in competitive performance and QoR over state-of-the-art hardware implementations.

TIIVISTELMÄ

High-Level Synthesis (*HLS*) on automatisoitu suunnitteluprosessi, joka pyrkii parantamaan tuottavuutta perinteisiin suunnittelumenetelmiin verrattuna, nostamalla suunnittelun abstraktiota *rekisterisiirtotasolta* (*RTL*) käyttäytymistasolle. Erilaisia kaupallisia HLS-työkaluja on ollut markkinoilla aina 1990-luvulta lähtien, mutta vasta äskettäin ne ovat alkaneet saada hyväksyntää teollisuudessa sekä akateemisessa maailmassa. Hidas käyttöönottoaste on johtunut pääasiassa huonommasta *tulosten laadusta* (*QoR*) kuin mitä on ollut mahdollista tavanomaisilla *laitteistokuvauskielillä* (*HDL*). Uusimmat HLS-työkalusukupolvet ovat kuitenkin kaventaneet QoR-aukkoa huomattavasti.

Tämä väitöskirja tutkii HLS:n soveltuvuutta videokoodekkien kehittämiseen. Se esittelee useita HLS-toteutuksia *High Efficiency Video Coding* (*HEVC*) -koodaukselle, joka on keskeinen mahdollistava tekniikka lukuisille nykyaikaisille mediasovelluksille. HEVC kaksinkertaistaa koodaustehokkuuden edeltäjäänsä *Advanced Video Coding* (*AVC*) -standardiin verrattuna, saavuttaen silti saman subjektiivisen visuaalisen laadun. Tämä tyypillisesti saavutetaan huomattavalla laskennallisella lisäkustannuksella. Siksi reaaliaikainen HEVC vaatii automatisoituja suunnittelumenetelmiä, joita voidaan käyttää *rautatoteutus-* (*HW*) ja varmennustyön minimoimiseen.

Tässä väitöskirjassa ehdotetaan HLS:n käyttöä koko enkooderin suunnitteluprosessissa. Dataintensiivisistä koodaustyökaluista, kuten intra-ennustus ja diskreetit muunnokset, myös enemmän kontrollia vaativiin kokonaisuuksiin, kuten entropiakoodaukseen. Avoimen lähdekoodin Kvazaar HEVC -enkooderin C-lähdekoodia hyödynnetään tässä työssä referenssinä HLS-suunnittelulle sekä toteutuksen varmentamisessa. Suorituskykytulokset saadaan ja raportoidaan *ohjelmoitavalla porttimatriisilla* (*FPGA*).

Tämän väitöskirjan tärkein tuotos on HEVC intra enkooderin prototyyppi. Prototyyppi koostuu Nokia AirFrame Cloud Server palvelimesta, varustettuna

kahdella 2.4 GHz:n 14-ytiminen Intel Xeon prosessorilla, sekä kahdesta Intel Arria 10 GX FPGA kiihdytinkortista, jotka voidaan kytkeä serveriin käyttäen joko *peripheral component interconnect express* (*PCIe*) liitäntää tai 40 gigabitin Ethernettiä. Prototyyppijärjestelmä saavuttaa reaaliaikaisen 4K enkoodausnopeuden, jopa 120 kuvaa sekunnissa. Lisäksi järjestelmän suorituskykyä on helppo skaalata paremmaksi lisäämällä järjestelmään käytännössä minkä tahansa määrän verkkoon kytkettäviä FPGA-kortteja.

Monimutkaisen HEVC:n tehokas mallinnus ja sen monipuolisten ominaisuuksien mukauttaminen reaaliaikaiselle HW HEVC enkooderille ei ole triviaali tehtävä, koska HW-toteutukset ovat perinteisesti erittäin aikaa vieviä. Tämä väitöskirja osoittaa, että HLS:n avulla pystytään nopeuttamaan kehitysaikaa, tarjoamaan ennen näkemätöntä suunnittelun skaalautuvuutta, ja silti osoittamaan kilpailukykyisiä QoR-arvoja ja absoluuttista suorituskykyä verrattuna olemassa oleviin toteutuksiin.

CONTENTS

List of Figures

List of Tables

List of Programs and Algorithms

ABBREVIATIONS

AI	All-Intra
ALM	Adaptive logic module
ALUT	Adaptive look-up table
AMP	Asymmetric motion partition
ASIC	Application-specific integrated circuit
AVC	Advanced Video Coding
AVX2	Advanced vector extensions 2
BD-rate	Bjøntegaard delta bitrate
CABAC	Context-adaptive binary arithmetic coding
CB	Coding block
CPB	Current picture buffer
CPU	Central processing unit
CTB	Coding tree block
CTU	Coding tree unit
CU	Coding unit
D	Residual
D’	Inversed residual
D_{rec}	Reconstructed picture
D_{ref}	Reference picture
DCT	Discrete cosine transform
DF	Deblocking filter

DMA	Direct memory access
DPB	Decoded picture buffer
DSE	Design space exploration
DSP	Digital signal processing
DST	Discrete sine transform
EC	Entropy coding
EDA	Electronic design automation
FIFO	First in, first out
FME	Fractional motion estimation
FPGA	Field-programmable gate array
GbE	Gigabit Ethernet
HDL	Hardware description language
HDMI	High-definition multimedia interface
HEVC	High Efficiency Video Coding
HLS	High-level synthesis
HM	HEVC Test Model
HW	Hardware
IDCT	Inverse discrete cosine transform
IDST	Inverse discrete sine transform
idx	Reference picture index
IME	Integer motion estimation
IP	Internet protocol
IP	Intra prediction
IPOL	Interpolation
IQ	Inverse quantization
IT	Inverse transform
LE	Logic element

LF	Loop filtering
LUT	Look-up table
MC	Motion compensation
MD	Mode decision
ME	Motion estimation
MV	Motion vector
OWF	Overlapped wavefront
P_{inter}	Inter prediction
P_{intra}	Intra prediction
PB	Prediction block
PC	Personal computer
PCIe	Peripheral component interconnect express
PMI	Prediction mode interlaced
PSNR	Peak signal-to-noise ratio
PU	Prediction unit
Q	Quantization
QoR	Quality of Results
QP	Quantization parameter
QTCOEFF	Quantized transform coefficient
RA	Random access
RD	Rate-distortion
RDO	Rate-distortion optimization
RMD	Rough mode decision
RTL	Register transfer level
RTP	Real-time transport protocol
SAD	Sum of absolute differences
SAO	Sample-adaptive offset

SATD	Sum of absolute transformed differences
SDN	Software defined networking
SoC	System on Chip
SW	Software
T	Transform
TB	Transform block
TCOEFF	Transform coefficient
TCOEFF'	Inversed transform coefficient
TU	Transform unit
USB	Universal serial bus
VHDL	Very high-speed integrated circuit hardware description language
WPP	Wavefront parallel processing

ORIGINAL PUBLICATIONS

Publication I — P. Sjövall, J. Virtanen, J. Vanne, and T. D. Hämäläinen, "High-level synthesis design flow for HEVC intra encoder on SoC-FPGA," in *Proceedings of Euromicro Conference on Digital System Design*, Funchal, Madeira, Portugal, Aug. 2015. DOI: 10.1109/DSD.2015.67.

Publication II — P. Sjövall, V. Viitamäki, J. Vanne, and T. D. Hämäläinen, "High-level synthesis implementation of HEVC 2-D DCT/DST on FPGA," in *Proceedings of IEEE International Conference on Acoustics, Speech and Signal Processing*, New Orleans, Louisiana, USA, Mar. 2017. DOI: 10.1109/ICASSP.2017.7952416.

Publication III — V. Viitamäki, P. Sjövall, J. Vanne, and T. D. Hämäläinen, "High-level synthesized 2-D IDCT/IDST implementation for HEVC codecs on FPGA," in *Proceedings of IEEE International Symposium on Circuits and Systems*, Baltimore, Maryland, USA, May 2017. DOI: 10.1109/ISCAS.2017.8050323.

Publication IV — S. Lahti, P. Sjövall, J. Vanne, and T. D. Hämäläinen, "Are we there yet? A study on the state of high-level synthesis," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 38, no. 5, pp. 898–911, May 2019. DOI: 10.1109/TCAD.2018.2834439.

Publication V — P. Sjövall, V. Viitamäki, A. Oinonen, J. Vanne, T. D. Hämäläinen, and A. Kulmala, "Kvazaar 4K HEVC intra encoder on FPGA accelerated air-frame server," in *Proceedings of IEEE International Workshop on Signal Processing Systems*, Lorient, France, Oct. 2017. DOI: 10.1109/SiPS.2017.8109999.

Publication VI P. Sjövall, V. Viitamäki, J. Vanne, T. D. Hämäläinen, and A. Kulmala, "FPGA-powered 4K120p HEVC intra encoder," in *Proceedings of IEEE International Symposium on Circuits and Systems*, Florence, Italy, May 2018. DOI: 10.1109/ISCAS.2018.8351873.

Publication VII V. Viitamäki, P. Sjövall, J. Vanne, and T. D. Hämäläinen, "Live demonstration: 4K100p HEVC intra encoder," in *Proceedings of International Symposium on Circuits and Systems*, Florence, Italy, May 2018. DOI: 10.1109/ISCAS.2018.8351770.

Publication VIII P. Sjövall, A. Oinonen, M. Teuho, J. Vanne, and T. D. Hämäläinen, "Dynamic resource allocation for HEVC encoding in FPGA-accelerated SDN cloud," in *Proceedings of IEEE Nordic Circuits and Systems Conference*, Helsinki, Finland, Oct. 2019. DOI: 10.1109/NORCHIP.2019.8906940.

Publication IX P. Sjövall, M. Teuho, A. Oinonen, J. Vanne, and T. D. Hämäläinen, "Visualization of dynamic resource allocation for HEVC encoding in FPGA-accelerated SDN cloud," in *Proceedings of IEEE Visual Communications and Image Processing*, Sydney, New South Wales, Australia, Dec. 2019. DOI: 10.1109/VCIP47243.2019.8966042.

Publication X P. Sjövall, A. Lemmetti, J. Vanne, S. Lahti, and T. D. Hämäläinen, "High-level synthesis implementation of an embedded real-time HEVC intra encoder on FPGA for media applications," *ACM Transactions on Design Automation of Electronic Systems*, vol. 27, no. 7, May 2022. DOI: 10.1145/3491215.

1 INTRODUCTION

High-level synthesis (*HLS*) is an intriguing subject in the field of *hardware* (*HW*) design. Even though various commercial HLS tools have been available on the market since the 1990s, they have only recently started to gain adoption across industry and academia [1]. One can argue that the slow adoption rate has mainly stemmed from lower *quality of results* (*QoR*) than obtained with conventional *hardware description language* (*HDL*) approaches. However, the latest HLS tool generations have substantially narrowed the QoR gap. HLS has also traditionally worked well with data-intensive designs, whereas implementing clock accurate control structures has been more challenging due to the lack of explicit time information in behavioural source code [2], [3].

Using HLS to fully implement a complex *High Efficiency Video Coding* (*HEVC*) [4], [5] video encoder on HW should give good insight on how HLS performs over traditional HDLs. HEVC is the key enabling technology for numerous modern media applications. HEVC doubles the coding efficiency over its predecessor *Advanced Video Coding* (*AVC*) standard for the same subjective visual quality, but typically at the cost of considerable computational complexity. Overcoming the complexity of HEVC and customizing its rich features for a real-time HEVC encoder implementation on HW is not a trivial task, as HW designs are traditionally very time-consuming. Thus, the development of modern video encoders can make use of automated design methodologies that can be used to minimize the HW implementation and verification effort.

This chapter gives the introduction for the thesis, including the motivation, research questions and methods, and the summary of publications with author's main scientific contribution in them.

1.1 The motivation and objectives of the research

The main motivation for this thesis is to evaluate whether HLS is a feasible implementation approach for computation-intensive multimedia processing. In practise, the analysis is carried out by implementing HLS solutions for HEVC encoding and evaluating their QoR, performance, and development time on *field-programmable gate array* (*FPGA*). The motivation is also to see if HLS is suitable for implementing designs that require more accurate control structures. This would enable the use of HLS for the whole system, and not just for traditionally suitable data-intensive algorithms. HEVC [4], [5] video encoder is an ideal application for the following reasons:

1. HEVC has gained a lot of traction in academia, so the obtained results can be compared with many other scientific works;
2. it is a very complex application, which will minimize the effect of normal variance in results (difference between HLS tools, synthesis results, etc.);
3. the complexity of HEVC is still reasonable for the scope this thesis;
4. HEVC has several independent units (coding tools, entropy coding, control structures) making it possible to showcase potential productivity increase and comparable QoR with HLS for single units and the whole system;
5. it also has characteristics that can be considered both strengths and weaknesses of HLS, i.e., functionality that can be considered data-intensive or something that needs of an accurate control structure; and finally,
6. HEVC is suitable for FPGA implementation, in order to enable performance measurements outside of simulations.

HEVC is a widely deployed and researched topic in industry and academia. The complexity of the HEVC [6] is very high, but HLS holds promise for better design management over traditional design methods. HEVC adopts the conventional hybrid video coding scheme (intra/inter prediction, transform coding, and entropy coding) [5] from the prior MPEG/ITU-T video coding standards. It offers a great variety of functionality, ranging from data-intensive algorithms, like intra prediction and discrete sine/cosine transform, to more control-intensive tools, such as intra search control and *context-adaptive binary*

arithmetic coding (*CABAC*). The latest FPGAs are able to meet the capacity and performance need for HEVC intra encoding, so they can be used as a real-life proof-of-concept test platforms, in addition to simulation.

The objectives of the thesis are the following:

1. To implement single algorithms/coding tools of HEVC on FPGA, by using Catapult HLS tool [7];
2. to integrate the implemented coding tools together and to add the associated control logic, to enable a fully-fledged real-time 4K HEVC intra encoder on FPGA; and
3. to make the encoder easily customizable for different media applications, scalable for different performance requirements, and portable to different platforms, from embedded devices to cloud environments.

The results of all these objectives are compared with prior art in existing scientific publications.

The entire implementation process relies on the open-source HEVC encoder called Kvazaar [8]–[10], that is used as a design entry point and as the reference *software* (*SW*) encoder. Although the implementation needs additional work to be fully optimized for HW, the original source code can still be used as is for testing purposes. Utilizing the existing Kvazaar C-code and C/C++ for the HW implementations, removes the need for re-implementing the reference algorithm. The automatic generation of *register transfer level* (*RTL*)-code with an HLS tool also removes the need for manual writing of traditional HDL, like *very high-speed integrated circuit hardware description language* (*VHDL*) and Verilog. This way, the focus can be directed more on the behavioural code.

Even though this work focuses on the *All-Intra* (*AI*) [11] coding configuration of HEVC Main Profile, the proposed design approach can be applied to other HEVC profiles or video codecs as well.

1.2 Research questions and methods

This thesis aims to answer the following research questions:

1. Is HLS feasible for the implementation of a complex multimedia application, like HEVC, that consists of multiple parallel units, and both control and data

intensive algorithms?

2. Do the developed HLS implementations for HEVC intra coding show comparable area and performance results with existing HW designs?
3. By combining and generalizing the first two questions, does HLS offer overall improvement over traditional design methodologies in terms of development time and QoR?

To answer these questions, the research in this thesis follows the principles of the design science methodology [12], [13] that has established well-known guidelines and evaluation methods for an iterative design process. The implementation of the whole encoding system is constructed of individual HEVC coding tools and control structures that are finally integrated together. The iterative design process for each coding tool starts from the reference algorithm, which is used for the first HW implementation and test bench. The design process then continues with *design space exploration* (*DSE*) and code restructuring, in order to optimize the area and performance of the HW design.

During the development, the individual units and the encoding system are iterated, improved, and compared with prior art according to the following criteria:

1. **Area**. The area figures reported for proposed implementations include the number of *logic elements* (*LE*), *look-up table* (*LUT*), *adaptive look-up table* (*ALUT*), or *adaptive logic module* (*ALM*) [14], the amount of on-chip memory, and the number of *digital signal processing* (*DSP*) units used for the specified FPGA. For fair comparison, effort has been made to unify the area figures with previous work, e.g., by using coefficients between the LEs according to the functionality of the used LE, or by re-generating the results for the same device using HLS and an appropriate synthesis tool.
2. **Performance**. The performance is measured for the developed individual coding tools by applying the worst-case scenario. For a complete encoder, several open-source 4K sequences [15] were used to obtain reliable and consistent results. Again, effort has been made to calculate or estimate missing values of the related work by extrapolating them from the published information.
3. **Coding quality**. For proposed individual coding tools where the implementation is one-to-one to Kvazaar [8]–[10], the need for such

measurements can be omitted. If the Kvazaar reference code had to be optimized in a way that it changed the original algorithm, this modification was also implemented in the HEVC reference encoder, *HEVC Test Model* (*HM*) [16], in order to report coding quality changes. *Bjøntegaard delta bitrate* (*BD-rate*) [17] can be used as direct value for comparing the coding quality between two codecs. It allows the measurement of bitrate reduction by a codec or a codec feature while maintaining the same quality as measured by objective metrics. In practise, the average *peak signal-to-noise ratio* (*PSNR*) is measured for the anchor and the one being evaluated using same four quantizers. BD-rate is then calculated for both curves [17]. Furthermore, HM, using AI configuration [18], is used as an anchor for the proposed full encoding system. The BD-rate gain, or loss, is reported for the test sequences from [15]. Coding quality is also considered when comparisons are made to related work.

4. **Supported HEVC features**. The supported features of individual coding tools and the supported encoding configuration of the full encoding system is always specified. The support or absence of features is listed for the related work. Supporting all features will always produce the best coding quality, but by removing some features, it can have a more positive affect on the area and performance when compared to the negative effect on coding quality.

This thesis contributes to the debate for the feasibility of HLS and the promise of overall improvement over traditional design methods by implementing a complete HEVC intra encoder on FPGA, using solely HLS. The individual publications included in this thesis illustrate the steps taken to achieve the goal and give a detailed description of how HLS was used to implement individual units and the final encoder.

1.3 Summary of publications

This section introduces all the publications included in this thesis. Figure 1.1 summarizes the main outcomes of these publications and the interconnections. The figure is drawn from the perspective of the developed HEVC encoder.

Publication I describe the developed HLS flow and how it can be used to develop HW- accelerated functions. The introduced HLS design flow is applied for

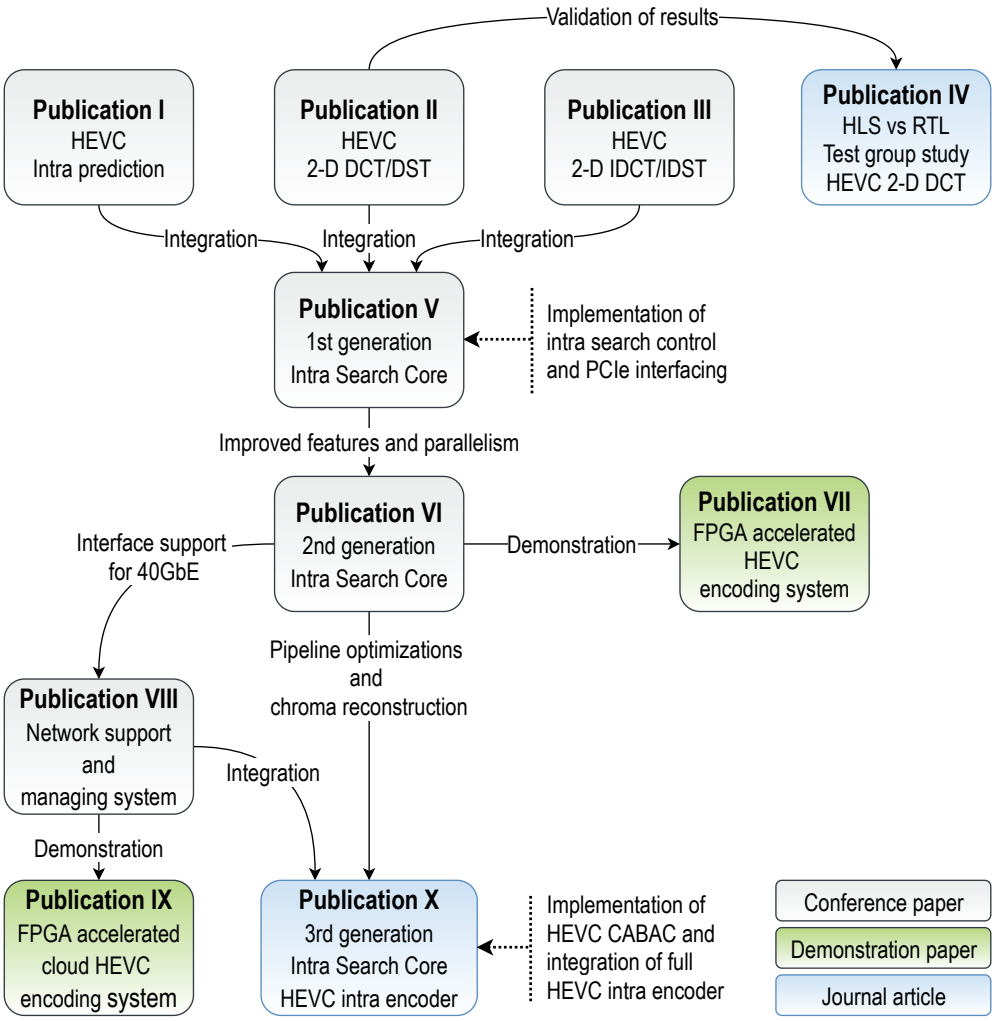


Figure 1.1 Summary and connection of publications.

HEVC intra prediction to produce a HW implementation for FPGA. The publication also presents a complete design of a real-time HEVC intra encoder on *system on chip* (*SoC*)-FPGA, utilizing the implemented HW intra prediction for encoding acceleration.

Publication II describes an HLS implementation of HEVC 2-D *discrete cosine transform* (*DCT*)/*discrete sine transform* (*DST*) on FPGA. The work reported in **Publication III** stems from that of Publication II, as it describes the inverse version of the same algorithm, 2-D *inverse discrete cosine transform* (*IDCT*)/*inverse discrete sine transform* (*IDST*). Both approaches implement the 2-D transforms by two

successive 1-D transforms using a well-known row-column and even-odd decomposition techniques. These two publications showed that the benefits of HLS do not come at the cost of implementation overhead, as the HLS solutions outperformed existing works in terms of performance and cost.

Publication IV is a survey of the state of HLS, based on the scientific literature published since 2010. The literature survey was conducted by Sakari Lahti. Sakari Lahti also wrote the majority of the text. The author contributed to this publication with the planning, organization, analysis, and text for the case study, in which a test group was given an assignment to implement HEVC 2-D DCT algorithm for 8×8 residual blocks with both HLS and traditional HDLs.

Publication V describes a 4K HEVC intra encoder partitioned between a processor and a *peripheral component interconnect express* (*PCIe*)–connected FPGA. It introduces the implementation and integration of the following data-intensive Kvazaar coding tools: intra prediction, DCT, IDCT, quantization, and inverse quantization. It also describes the HW implementation of the control-oriented intra search process using HLS. CABAC and other control-intensive coding tools are executed on SW. The publication also describes the implementation of the HW/SW partitioning scheme between a *central processing unit* (*CPU*) and an FPGA.

Publication VI describes the 2nd version of the Intra Search Core that has been improved upon the 1st version presented in Publication V. The proposed speedup techniques include optimizing the use of DSP-units, increasing the support of parallel *coding tree units* (*CTU*) on HW, and duplicating time-sensitive resources. Furthermore, the Linux kernel driver is upgraded to maximize the utilization of both HW and SW and to support multiple PCIe-FPGA cards.

Publication VII is a demonstration publication. It showcases how the encoding system presented in Publication VI can be utilized for real-life purposes. The presented demonstration setup enables real-time HEVC encoding of three 4K cameras simultaneously. The cameras send the RAW video data through *high-definition multimedia interface* (*HDMI*) and the HDMI-capture cards transmit the data to the encoder via *universal serial bus* (*USB*). Finally, the encoded video is sent via Ethernet for live view on three laptops.

Publication VIII describes an approach to accelerate, distribute, and manage video encoding services in large-scale cloud systems. The Intra Search Core

presented in Publication VI is used as a proof-of-concept application. In the case of cloud encoding, the usage of PCIe-FPGAs from the previous publications would have limited the flexibility of the encoding system, as the FPGAs are bound to servers and only 1-2 boards could be attached per cloud server. The given solution is an advanced partitioning scheme for sharing execution between servers, FPGAs, and *software defined networking* (*SDN*) switches. This combination allows the deployment of practically any number of heterogeneous FPGAs and servers. The implemented resource manager, which controls the SDN switches, is responsible for allocation, deallocation, and load balancing of software and hardware resources upon service requests, or changes in network infrastructure. The research also includes HLS implementations for the Ethernet packet parsing and generation, which was integrated to the HEVC accelerator logic presented in Publication VI.

Publication IX is another demonstration publication. It showcases how the system presented in Publication VIII works in practice. The demonstration setup includes a laptop that is connected to the cloud system. Through this connection, several encoding services can be invoked with requests to the resource manager. A run-time visualizer on the laptop illustrates in real-time the data provided by the resource manager, such as the physical network structure, running services, and performance of the network elements. The live encoded video stream can also be viewed on the laptop screen.

Finally, **Publication X** describes the HLS development framework improved upon Publication I, the 3rd version of Intra Search Core optimized from the version described in Publication VI, and the HLS implementation of the CABAC Core. The integration of intra search and CABAC cores is also presented, and the needed changes in the kernel driver. This finally creates a complete HEVC intra encoder on FPGA. This is the key publication of the thesis, and it shows that the HLS proposal not only boosts development time, but also provides previously unseen design scalability with competitive performance over the existing encoder implementations.

The author served as the 1st author in publications Publication I, Publication II, Publication IV, Publication V, Publication VIII, Publication IX, and Publication X, and as the 2nd author in Publication III, Publication VI, Publication VII. The author's contribution in each publication is listed in Table 1.2. The table also lists the work done in collaboration with co-authors.

Pub.	Author's own work	Work done in collaboration or by co-authors
I	• Presented HLS design flow • HEVC intra prediction (IP) HW implementation with HLS • Implementation of DMAs on FPGA • Integration of the IP accelerator into the SoC CPU • Linux kernel driver for interfacing the IP accelerator • Supporting HW IP acceleration in Kvazaar • Performance analysis	• HW components related to camera and display • FPGA – CPU interfacing related to camera and display
II	• Initial versions of 2-D DCT and 2-D DST • Implementation of the low-cost 2-D DCT variant • Implementation of the transpose memory • Performance analysis	• Implementation of the high-speed 2-D DCT variant • Implementation of the 4×4 2-D DCT/DST variant
III	• Initial versions of 2-D IDCT and 2-D IDST • Implementation of the transpose memory • Performance analysis	• Implementation of the final 2-D IDCT/IDST unit
IV	• Planning, development, organization, analysis, and text for the test group study	• The conducted survey and the writing of rest of the publication was done fully by the co-author Sakari Lahti
V	• Idea and design of the CTU based intra coding accelerator • FPGA – CPU interfacing using PCIe • Implementation of DMAs on FPGA • CTU-based memory indexing on FPGA • Linux kernel driver for interfacing the Intra Search Core on the PCIe FPGA from a server CPU • Supporting HW intra coding acceleration in Kvazaar • Performance analysis	• Design of the CTU based intra coding accelerator • Implementation of Intra Coding Control, Get Reference, Border, Coefficient Cost, and Reconstruction for the Intra Search Core
VI	• Linux kernel driver support for multiple PCIe FPGAs • Improved CTU load balancing • Interrupt support on FPGA and Linux kernel driver for signalling the finished coding of a CTU	• Optimization of HW units in the Intra Search Core
VII	• Setup and demonstration	
VIII	• Implementation of the Resource Manager application • HLS implementations of Ethernet packet parsing and generation on FPGA • FPGA – CPU interfacing using Ethernet • Supporting cloud HW intra coding acceleration in Kvazaar	• Initial idea of using FPGAs together with SDN • Initial network – FPGA tests • Orchestration of the prototype cloud system
IX	• Gathering data in the Resource Manager application and providing it to the run-time visualizer • Setup and demonstration	• Implementation of the run-time visualizer
X	• Improved HLS development framework from Pub. I • Optimized version of Intra Search Core from Pub. V • Full HW implementation of the CABAC Core using HLS • Integration of CABAC Core into Intra Search Core • Support for HW CABAC in the Linux kernel driver and Kvazaar • Performance analysis	

Figure 1.2 Author's main scientific contributions.

1.4 Outline of the thesis

The introductory part of this thesis is organized as follows. Chapter 2 describes the background of the thesis. It introduces the HLS design flow, the basics of HEVC encoding, and the basic concept of an FPGA. Chapter 3 considers the related work that is limited to HW HEVC approaches in academia. The related work consists of HLS implementations for single HEVC coding tools, as well as dedicated CABAC and entire encoder implementations done with handwritten HDLs. Chapter 4 summarizes the main results of the publications and gives the rationale for design choices and implementation process. Finally, Chapter 5 concludes the thesis.

1.5 Acknowledgments

This work is supported in part by Nokia, the European Celtic-Plus projects 4KREPROSYS and VIRTUOSE, the European ECSEL project ADACORSA (under the grant agreement 876019), the Tuula and Yrjö Neuvo Fund, Nokia Foundation, and the Finnish Foundation for Technology Promotion.

2 BACKGROUND

This chapter introduces the automated design process, HLS design flow, and their advantages over traditional design methods. Secondly, the basics of HEVC encoding are presented with common terminology used in the introductory part and publications. Finally, the use of FPGA as the target technology is rationalized and the basic concept of an FPGA is explained.

2.1 Automated design process: High-level synthesis (HLS)

HLS is an automated design process, where an HLS tool is used to automatically generate the RTL HDL-code from a high-level program code. The higher abstraction level offers technology-independent implementations, as the HLS tool can be used to configure and optimize the design for different target technologies, e.g., *application-specific integrated circuits* (*ASIC*) or FPGAs, without the need to rewrite the code. The RTL-code generation is based on the behavioural code and constraints specified by an HLS tool. Furthermore, to minimize the verification effort, HLS tools can be used to verify both the high-level code and the generated RTL-code with a shared high-level testbench.

There are numerous commercial and academic HLS tools [19] that support the HLS design flow. According to the survey done in Publication IV, the most popular HLS tool in academia was the Xilinx Vitis HLS [20] (previously Vivado HLS/Autopilot). However, the range of different HLS tools used in the scientific literature is wide, which argues that no single HLS tool is the best choice for every situation.

2.1.1 HLS flow

Figure 2.1 depicts the conceptual block diagram of the HLS design flow. The HLS tool is used for compiling the algorithmic specification of the system, most often

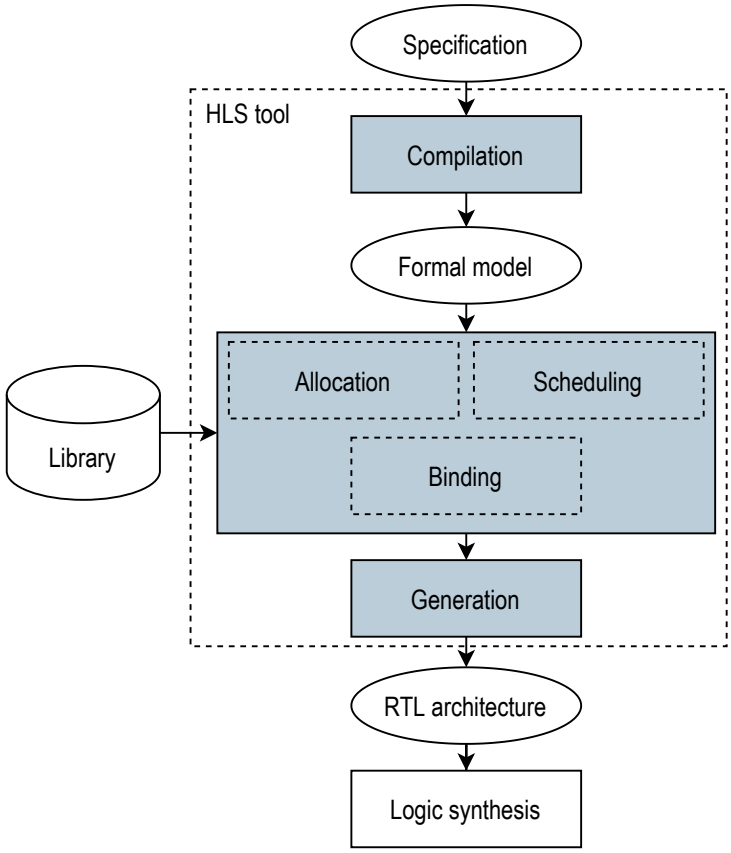


Figure 2.1 HLS design flow [Publication X].

written in C, C++, or SystemC. The user can specify the target technology and provide micro-architectural constraints, such as directives for loop pipelining/unrolling, desired clock frequency, number of clocks, clock-crossing, the usage and depth of *first in, first out* (*FIFO*) memory components, and mapping of arrays to registers or memories. The HLS tool then automatically allocates the HW resources required by the specification, creates state machines, schedules the operations, and binds the operations to physical resources specified in the target technology library. The clock(s) and reset(s) in the generated RTL code are completely inserted by the HLS tool, according to the architectural constraints. The generated RTL HDL-code can then be used in the downstream logic synthesis SW, for both FPGA and ASIC designs.

2.1.2 Advantages of HLS over traditional hardware design methods

The main strength of HLS comes from the productivity increase, when compared to traditional HW design methods. This is achieved by increasing design abstraction from RTL to behavioural level [2], [3], [21]. HLS has been reported to provide 4-6 times increase in productivity [Publication IV], mainly because the behavioral code is more readable, design and verification times are shorter, and the design reusability is far better than that of handwritten HDL.

In particular, HLS offers advantages over manual RTL coding in the following cases:

1. **Data-intensive designs.** HLS has traditionally worked well with data-intensive designs, whereas implementing clock accurate control structures has been more challenging due to the lack of explicit time information in behavioural source code [3], [21]. However, a recent work [22] has showed that even more demanding control structures could be described with the latest HLS tools.
2. **Algorithm and system architecture optimizations outperform micro-architectural optimizations.** Adopting HLS and high-level coding techniques allows the designer to shift focus from fine-tuning single algorithms to the whole system architecture, which tends to provide higher performance gains than optimizing micro-architectures.
3. **DSE.** Although the system architecture optimizations can also be considered as part of DSE, HLS tools also enable DSE for single units in the form of architectural constraints. These can be applied via configuration scripts or by embedded pragmas, like loop unrolling/pipelining options, in the code. The process of finding the optimal trade-off between performance and area is significantly faster with HLS than with hand-written RTL. In practice, a comprehensive DSE cannot be conducted with conventional HDL approaches, as each solution would require extensive rewriting of the code.
4. **Reduced verification effort.** Reducing the verification effort is an important aspect of any design. Especially for digital system projects, it has become one of the most time-consuming phase [23]. The verification process can be significantly faster with HLS than verifying manually written RTL, as the functional correctness of the high-level code and the automatically generated

RTL code can be done using the same high-level testbench. The output of the generated RTL can also be validated automatically against the behavioural code. Most often, only the algorithm level verification is needed. In addition, some corner cases, e.g., type casting or vector overflows, may require additional effort. The testbench also synchronizes with the input and output of the design under verification, which makes it tolerant of architectural changes and removes the need for any code rewriting during the DSE.

5. **Technology independency.** HLS can offer better design reusability over traditional design approaches. A technology-independent behavioral code releases the designers from addressing the implementation details of the target technology, such as timing, interfaces, and memory elements. In principle, the same holds for the handwritten RTL code, but the design is usually implemented with a specific technology and performance in mind. In practise, when a new platform is selected with an HLS tool, a platform-optimized RTL code is generated using the same existing source code. In contrast, with manual RTL, time-consuming code restructuring is typically needed because of new clock constraints or additional resource sharing, caused by limited capacity of the new platform. Full technology independency might not be achieved with every HLS tool, as different HLS tools support a different range of target technologies.

6. **Increased productivity.** All previous advantages of HLS add to the compelling productivity increase over traditional RTL design flows. Even though custom RTL approaches tend to achieve better performance with less resources, the literature survey in Publication IV indicates that the average development time of an HLS project is only a third of that of the manual HDL project. The average productivity of HLS is also reported to be more than 4× as high in terms of the system performance with respect to the development time.

2.1.3 Catapult HLS tool

In this work, Catapult HLS [7] from Siemens (previously owned by Calypto and Mentor) was used for its availability at the university where the research was carried out. In addition, Catapult HLS was preferred over another available tool, Xilinx Vitis, because Catapult HLS supports Intel FPGAs that were found

appropriate for this work. Several versions of the Catapult HLS tool were used during the research and the version was updated upon availability of newer version. Before the version 10.3 was released, the university only had access to a university license (some limitations in features), after that a full license was available.

Catapult HLS provides a wide support for the features described in Section 2.1.2. These features include:

- language support for C/C++/SystemC;
- support for both algorithmic and control logic synthesis;
- configuration scripts and embedded pragmas for architectural constraints;
- support for a unified testbench for the verification of both algorithmic code and RTL, with automated verification of the generated RTL against the algorithmic code;
- variety of target technologies ranging from ASIC technologies to different FPGA vendors, including automated synthesis tool integration;
- possibility to have multiple clock domains in a hierarchical designs and automatically generated clock crossing components;
- initial area and performance results without actual RTL synthesis; and
- a cycle accurate Gantt Chart of the scheduled operations for a visual representation of the generated RTL.

2.1.4 HLS design example

Listing 2.1 exemplifies the basic HLS coding features supported by the Catapult HLS tool. It shows how hierarchical designs, arrays, loops, pragmas, functions, channels, look-up-tables, type definitions, bit accurate types, macros, and compilation time calculated bit widths (*nbits*) can be used in HLS code. In this producer-consumer implementation, the *producer* (chef) is hierarchically connected to the *consumer* (tables) on the top-level function *restaurant*. These functions are marked with pragmas *hls_design* and *hls_desgin top*. The top-level interface consists of two parameters, orders and income. Both of these are defined as algorithmic C (*ac*), datatypes, and more specifically channels. In Catapult, the channels can be used to implement data transfers without, or with one or two handshake signals, buffered input/output for a more pipelined behaviour, FIFOs with automatic or

manually adjusted depth, and even clock crossing support between hierarchical blocks. Bit accurate types (*ac_int*) are also used and defined with type definitions.

The implementation of the *producer* has three separate parts. First, a *Tables* structure is read from the channel *orders*. The helper structure is necessary for reading an array of data with one call. Secondly, a separate function called *calculate_number_of_orders()* is used for calculating the number of non-zero orders in a loop that is fully unrolled with pragma *hlr_unroll*. Thirdly, the chef starts preparing the food in a loop for each table that has a non-zero *recipe* and writes the correct price from a look-up-table *price_of_food* to the correct index in the array of channels. This array will result in separate data paths for each index in the generated RTL. The for loop is pipelined with pragma *hls_pipeline_init_interval 1*, stating that a new iteration of the loop is started after one cycle. The possible interval depends on the implementation and data dependencies. The implementation of the for loop also shows how a *break* statement is used for an optimized loop control, as a known number of loops is always better than an unknown one. Breaking the loop when remaining iterations have no effect improves the responsiveness. It also shows the usage of *continue*.

The implementation of the *consumer* has to parts. First part is the loop, which is fully unrolled. All tables that get food, eat and pay for it. As the *food* parameter of the function is an array of channels and all indexes need to be iterated, the loop utilizes *size()* function of the channel to implement non-blocking reading. The other alternative would be using *nb_read()*, but size() is used here for a more optimized implementation. If the table is getting food (*size()* is not zero), the price for the food is read from the channel and stored to *cash*. The *cash* is incremented to the *cash_register* as zero or the read value. This loop shows the importance of data dependency in a fully unrolled loop, as incrementing *cash_register* inside the if statement would have resulted in a lower maximum frequency due to the dependency to *cash_register* between iterations. After the loop the *cash_register* is given to the owner.

On top of the code, the compilation can be affected with a set of directives (Tcl script), that can be modified directly in the directives file or via GUI. These directives affect the outcome, i.e., area, latency, and slack. Pragmas are also interpreted from the HLS code automatically as directives. Other important directives include design goal latency/area, clocks, clock overhead, resource binding (channels, wires, memories, registers), FIFO depths, and DSP extraction options.

```
#include "ac_int.h"
#include "ac_channel.h"

// Macro for amount of tables in the restaurant
#define TABLES 16

// Typedefs for bit accurate types
typedef ac_int<1, false> uint_1;
typedef ac_int<2, false> uint_2;
typedef ac_int<4, false> uint_4;
// Just enough bits (nbits<>) for iterating tables
typedef ac_int<ac::nbits<TABLES>::val, false> loop_t;
// Just enough bits (nbits<>) for maximum amount of money
typedef ac_int<ac::nbits<3 * TABLES>::val, false> money_t;

struct Tables{ // Helper struct for an array of orders
uint_2 order[TABLES];
};

static uint_2 price_of_food[4] = {0,1,2,3}; // Look-up-table for prices

static loop_t calculate_number_of_orders(Tables &cook) { // Inlined function
    loop_t num_of_orders = 0;
    #pragma hls_unroll // Calculate number of orders
    for(loop_t i = 0; i < TABLES; i++) {
        num_of_orders += cook.order[i] != 0 ? 1 : 0; // Increment if order is non zero
    }
    return num_of_orders;
}

#pragma hls_design // Pragma for hierarchical design
void producer(ac_channel<Tables> &orders, ac_channel<uint_2> food[TABLES]) {
    Tables cook = orders.read();
    loop_t num_of_orders = calculate_number_of_orders(cook);

    // The chef can pipeline the making of food and start next iterations every cycle
    #pragma hls_pipeline_init_interval 1
    for(loop_t i = 0; i < TABLES; i++) {
        uint_2 recipe = cook.order[i];
        if(recipe == 0) continue; // No order? Continue to next table
        food[i].write(price_of_food[recipe]); // Send food with price
        if(num_of_orders-1 == 0) break;
        num_of_orders--;

    }
}

#pragma hls_design // Pragma for hierarchical design
void consumer(ac_channel<uint_2> food[16], ac_channel<money_t> &owner) {
    money_t cash_register = 0; // No money yet
    #pragma hls_unroll // One waiter per table, everyone eats/pays at the same time
    for(loop_t i = 0; i < TABLES; i++) {
        uint_1 available = food[i].size(); // Check if food is serverd to table
        uint_2  cash = 0;
        if(food[i].size()) cash = food[i].read(); // Get the cash from the table
        cash_register += cash;
    }
    owner.write(cash_register); // Deposit all money
}

#pragma hls_design top // Pragma for top level
void restaurant(ac_channel<Tables> &orders, ac_channel<money_t> &income) {
    static ac_channel<uint_2> waiter[TABLES]; // Waiters serve food from chef to tables
    producer(orders, waiter); // Producer, chef
    consumer(waiter, income); // Consumer, customer at the table
}

```

Listing 2.1 HLS example

2.2 Application of Interest: High Efficiency Video Coding (HEVC)

Due to the increased usage of numerous modern media applications, i.e., Internet video, video-streamed gaming, and video conferencing, video traffic has been estimated to account for 82% of all global *internet protocol* (*IP*) traffic by 2022 [24]. HEVC [4], [5] is the latest widespread MPEG/ITU-T video coding standard. It doubles the coding efficiency over its predecessor AVC [25] standard for the same subjective visual quality, but typically at the cost of considerable computational complexity overhead in the reference encoder HM [6] and practical encoders.

HEVC adopts the conventional hybrid video coding scheme [5] from the prior MPEG/ITU-T video coding standards. The encoding model of HEVC is shown in Figure 2.2. It can be divided into intra/inter prediction, transform coding, and entropy coding. They are covered in the following sections. Although the focus of this thesis is in HEVC intra encoding [11], inter encoding is also briefly covered.

2.2.1 Block partitioning structure

The coding structure of HEVC has been extended from the traditional macroblock structure. It is an analogous block partitioning scheme with four different logical units: CTU, *coding unit* (*CU*), *prediction unit* (*PU*), and *transform unit* (*TU*). For 4:2:0 color format, each of these consists of one luma and two chroma blocks that cover the corresponding block areas: *coding tree block* (*CTB*), *coding block* (*CB*), *prediction block* (*PB*), and *transform block* (*TB*). This new HEVC coding structure is the primary factor for the improved coding gain over AVC. As a downside, it also introduces majority of the computational overhead for both intra and inter encoding.

In HEVC, each raw input video frame is partitioned into CTU [26] quadtree structures. The size of the CTU can be defined as $2N_{MAX} \times 2N_{MAX}$, where $N_{MAX} \in \{8, 16, 32\}$. The CTU can be recursively split into four smaller square CUs until the min CU size of 8×8 is reached. The size of the CU can be defined as $2N \times 2N$, where $N \in \{4, 8, 16, 32\}$. Each CU in the CTU is predicted and transformed individually.

2.2.2 Intra prediction

Intra prediction utilizes spatial redundancy in compression. HEVC intra prediction supports CBs from 32 × 32 pixels down to 8 × 8 pixels. In intra prediction, the PBs

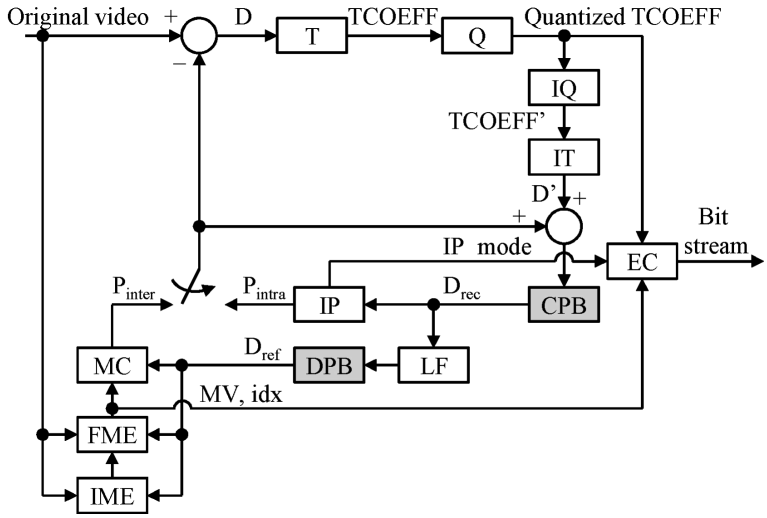


Figure 2.2 HEVC encoder model [6].

can be of the same size as the processed CB. In addition, HEVC supports partitioning the CB into four equal sized PBs when the CB size equals the smallest allowed CB size.

The *intra prediction* (*IP*) algorithm utilizes predefined prediction methods [5] to estimate the current CU. HEVC supports 35 different IP modes (DC, planar, and 33 angular modes) for each PB size. The *intra prediction* (P_{intra}) refers to the *reconstructed picture* (D_{rec}) from the *current picture buffer* (*CPB*), which is a storage for the previously reconstructed CBs. In intra mode, the *residual* (*D*) is calculated by subtracting P_{intra} from the original CB.

2.2.3 Inter prediction

Inter prediction utilizes temporal redundancy between different video frames. HEVC inter prediction supports CBs of 64 × 64 pixels down to 8 × 8 pixels. In addition to PBs of the same size as the processed CB, HEVC supports horizontal and vertical partitioning of CBs into two equal size PBs, or into four equal sized PBs when the CB size equals the smallest allowed CB size. Furthermore, by utilizing *asymmetric motion partitions* (*AMP*), HEVC supports partitioning of CBs into two asymmetric PBs [4], [5].

Inter prediction consists of *motion compensation* (*MC*), *integer motion estimation* (*IME*), and *fractional motion estimation* (*FME*). MC produces *inter*

predictions (P_{inter}) for PBs based on the result of *motion estimation* (*ME*) (IME, FME). First, IME is used to search the best candidate for the PB of interest from the *decoded picture buffer* (*DPB*), which is a storage for previously reconstructed *reference pictures* (D_{ref}). The *motion vector* (*MV*) and *reference picture index* (*idx*) produced by the IME is then forwarded to FME, which uses an 8-tap/4-tap separable *interpolation* (*IPOL*) filter to produce 1/4-pixel luma samples and 1/8-pixel chroma samples. The initial MV generated by the IME can then be refined to a sub-pixel accuracy.

When the encoder operates in inter mode, the P_{inter} produced by the MC is used to compute the D, by subtracting P_{inter} from the original CB. If the CU is encoded as skip mode, no D is computed, only PBs of size $2N \times 2N$ are allowed, and motion parameters are derived with merge mode [5]. The merge mode is used to derive the motion parameters, including MV and one or two *reference picture indices* (*idx*), from spatially or temporally neighboring blocks.

2.2.4 Transform coding

In HEVC, the *transform* (*T*) stage utilizes 2-D DCT (TB sizes from 32×32 to 8×8) and 2-D DST (4×4 TBs) to transform D from spatial domain into frequency domain *transform coefficients* (*TCOEFF*) [27]. The transform coding supports CB sizes of 32×32 to 8×8 and further partitioning of square CBs into square sized TBs until the size of 4×4 is reached. In frequency domain, high-frequency components of the video can be removed in the *quantization* (*Q*) stage to produce *quantized transform coefficients* (*QTCOEFF*) without significant quality loss, since human eye is less sensitive to the high-frequency components.

The encoding loop also includes decoder-side functionality such as *inverse quantization* (*IQ*) and *inverse transform* (*IT*) stages, where QTCOEFFs are dequantized into *inversed transform coefficients* (*TCOEFF'*) and then transformed back into *inversed residuals* (*D'*), i.e., back from frequency domain to spatial domain. The IT stage uses the corresponding inverse functions of the T stage, i.e., 2-D IDCT and 2-D IDST [27]. The reconstructed D is then added to the P_{intra}/P_{inter} to generate the final D', which is stored in CPB. For example, in intra mode, reconstructed CBs are needed in the intra prediction phase, where spatially adjacent pixels are used as a reference for generating predictions for neighbouring CBs. Furthermore, as the reconstructed pictures correspond to the images

generated and displayed by the decoder, they can also be used to measure the error introduced by compression.

In addition, HEVC supports sequential in-loop filters, *sample-adaptive offset* (*SAO*) and *deblocking filter* (*DF*), in the *loop filtering* (*LF*) stage. This stage is used for filtering distortions and visible borders of blocks. Unlike intra mode, which utilizes the D_{rec} directly, inter mode utilizes the filtered D_{rec} from DPB to find the best temporal candidate for the processed PU and to generate the inter prediction.

2.2.5 Entropy coding

In the last encoding phase, the QTCOEFFs, IP mode, and MV are processed by the *entropy coding* (*EC*) stage to generate the final encoded bitstream. In this step, the video signal is reduced to a series of syntax elements that contain properties of the blocks, including prediction modes, quantization parameters, transform coefficients, filter modes, and all other parameters required to describe how the video signal should be reconstructed by the decoder.

These elements are ordered and compressed to generate an encoded video bitstream. Entropy coding method in HEVC is called CABAC [28], which is a lossless compression technique based on arithmetic coding. The compression is achieved by utilizing statistical properties of symbols, i.e., more frequent symbols are coded with less bits and less frequent symbols with more bits.

2.2.6 Techniques for HEVC encoder parallelization

HEVC supports parallel processing of multiple CTUs [29] in a single frame, as well as parallelism between frames. As shown in Figure 2.3, the specified strategies include (a) slices, (b) tiles [30], (c) *wavefront parallel processing* (*WPP*) [30], [31], and (d) *overlapped wavefront* (*OWF*) [30], [32].

In its simplest, a slice in HEVC can contain the whole frame. For added parallelism, a single frame can be encoded using several independent slices. The minimum size of a slice is a single CTU. Each slice can consist of varying number of CTUs per frame, and each frame can consist of varying number of slices.

With tiles, the frame can be divided into independently encoded rectangular regions. This increases the capability of parallel processing of CTUs in a single

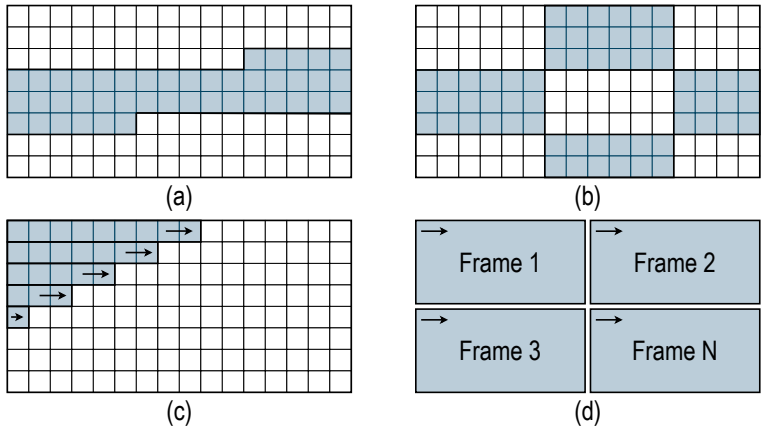


Figure 2.3 Parallelization approaches supported by HEVC: (a) slices, (b) tiles, (c) wavefront parallel processing, and (d) overlapped wavefront.

frame without the need for complex synchronizations.

When WPP is enabled, processing of adjacent CTU rows can always be started when two CTUs from the preceding row have been encoded. Thus, the level of parallelism with WPP increases row by row. Furthermore, WPP may provide higher coding performance than tiles by utilizing CABAC better, i.e., the CABAC state of the previous row is always propagated to the next row with a delay of two CTUs. This dependency does require better synchronization methods than with tiles.

OWF is the process of encoding multiple frames in parallel. This is straightforward in all-intra mode, as the frames have no temporal dependencies, but when using inter mode, additional synchronization is required for reference pictures.

2.2.7 Open-source implementations for HEVC encoding

The most notable open-source HEVC encoder is the HEVC reference software called HM [16]. The main purpose of HM is to implement and provide a reference for all coding tools in the standard. On the other hand, HM has not been optimized for speed, e.g., with multithreading, and thus it has a considerable computational complexity, which severely limits its practical usage outside research.

There are also two notable open-source HEVC encoder implementations addressing practical encoding, namely x265 [33] and Kvazaar [8]–[10]. Both of them are highly optimized and support HEVC specified parallelism to achieve

real-time SW encoding. The x265 encoder has primarily been written in C++, just like HM. The development of x265 is coordinated by MulticoreWare. Kvazaar encoder is presented in more detail in Section 2.2.8.

2.2.8 Kvazaar HEVC encoder

Kvazaar [8]–[10] is an academic open-source SW HEVC encoder developed by Ultra Video Group at Tampere University. The open development of the encoder was started in January 2014 and still continues actively. The encoder has completely been implemented from scratch using C with additional *advanced vector extensions 2* (*AVX2*) optimizations. Kvazaar supports HEVC Main profile [18] with ten presets from *ultrafast* to *placebo* [10]. Kvazaar architecture offers multi-threaded coding scheme with *rate-distortion optimized* (*RDO*) mode decision and HEVC parallelization strategies to achieve high *rate-distortion* (*RD*) performance with reasonable coding time.

In almost all publications included in this thesis, Kvazaar is used as a reference software for HEVC, i.e., as a design entry point for HLS implementations, for testing purposes, and as the SW application running on CPU where applicable. The wide usage of Kvazaar in this thesis is not only because Kvazaar has been developed in the same research group as this thesis is being carried out, but also because Kvazaar is a fully-fledged open-source HEVC encoder, which is implemented completely in a hardware-friendly C-code, and has shown promising results when compared with HM [16] or x265 [33].

2.3 Target device: Field-programmable gate array (FPGA)

FPGAs are re-programmable logic circuits that enable fast development and real-time emulations for HW implementations. A simplified example of an FPGA is illustrated in Figure 2.4. In the basic case, an FPGA is made up of an array of programmable LEs, consisting of a classic 4-input LUT, carry logic, and a single register. The FPGA also has programmable interconnections for connecting the LEs. These interconnected LEs can be used to form designs from simple gates to highly complex functions. In reality, the basic building block [14] of an FPGA device depends highly on the FPGA vendor. These building blocks can consist of several classic LEs, larger LUTs, multiple registers, or additional hard-coded

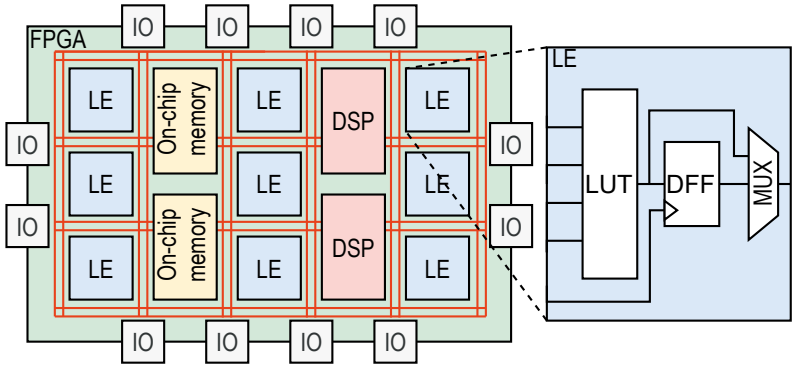


Figure 2.4 Simplified example of an FPGA.

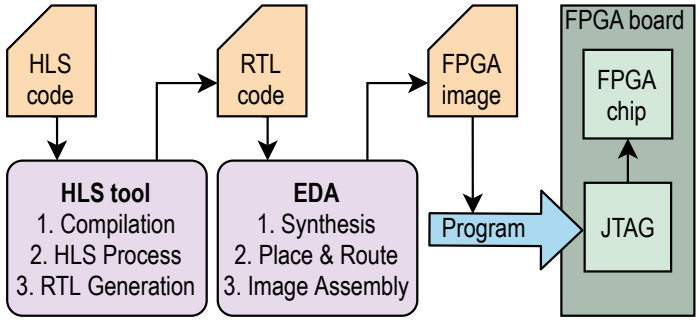


Figure 2.5 High-level overview of the design flow from HLS code to FPGA programming.

functionality such as adders. The FPGA chips also usually contain on-chip memory blocks and DSPs. The memory blocks are typically allocated for FIFOs and single or dual-port memories. DSPs are mostly used for performing complex mathematical functions. They can be used together with LEs and minimize the LE usage in memory-intensive or arithmetic applications.

Figure 2.5 gives a high-level overview of the design flow from HLS code to FPGA programming. The initial state of the LEs, memory blocks, DSPs, and the connections between them are generated by an *electronic design automation* (*EDA*) software. The EDA usually takes an HDL description of the design as an input and the HDL code goes through synthesis, place & route, and finally bitstream generation. The bitstream image can be used for programming the specific FPGA. Because the code generated by the HLS tool is in HDL format (VHDL or Verilog), the use of HLS should not affect the EDA tools used for FPGA synthesis.

FPGA is selected as the target device in almost every publication included in this thesis, because it enables the construction of proof-of-concept systems, which

is a step forward from calculations or simulations. Achieving realistic performance evaluations is somewhat limited in simulations, as it might not consider all real-life aspects and dependencies. Furthermore, FPGAs were also preferred to ASICs, because the back-end flow for ASICs would have been too demanding for the scope of this thesis, without the possibility to actually have the chip physically manufactured.

From the scope of the HLS flow and the generated RTL, the target technology is not that important, as HLS offers technology-independent designs. This made it possible to prioritize the device selection according to device and EDA software availability/support. Hence, the chosen setup was Catapult HLS, Intel Quartus Prime (previously Altera Quartus II) FPGA synthesis tool, and Intel (previously Altera) FPGAs (Cyclone V, Arria V, and Arria 10 device families)

3 RELATED WORK

Since the advent of HEVC, a great number of individual HW components or complete HW encoders have been designed for it on FPGAs and ASICs, both in academia and industry. However, to the best of knowledge, none of the existing HLS approaches, at least in literature [34]–[45], have proposed a complete HEVC encoder but only individual HEVC coding tools.

All these existing HLS approaches considered only data-intensive coding tools and skipped the control-intensive parts of HEVC, e.g., CABAC. This is because data-intensive algorithms have traditionally been considered more suitable for HLS. CABAC is part of the work presented in this thesis, and due to the lack of HLS implementations of CABAC in literature, independent CABAC units implemented with traditional RTL [46]–[52] are included in the related work for comparison. Correspondingly, industrial and academic HEVC encoders developed with manual RTL are considered as related work [53]–[67].

3.1 Existing HLS implementations for HEVC

As listed in Table 3.1, HLS implementations have been presented for different HEVC encoding tools, including 1) IP [34]–[36]; 2) DCT/IDCT [37], [38]; 3) interpolation [39]–[41]; and 4) *sum of absolute transformed differences* (*SATD*) [42]. Correspondingly, Table 3.2 lists the works for HEVC decoding tools [43]–[45].

3.1.1 Intra prediction

The authors in [34] used Xilinx Vivado HLS (now Xilinx Vitis HLS [20]) to implement HEVC IP for Xilinx Virtex 6. It supported all PU sizes and was able to process predictions for 1080p video at 35 fps (1080p@35fps) in the worst case. Three prediction units are used to calculate angular, planar, and DC predictions at 32, 4, and 1 pixel(s) per clock cycle, respectively. The authors performed manual

Table 3.1 Existing HLS approaches for HEVC encoding

	Intra	Inter	Coding tool
Kalali [34]	×		Intra prediction
Cui [35]	×		Intra prediction
Chen [36]	×		Intra prediction
Mohamed [37]	×	×	DCT
Kalali [38]	×	×	IDCT
Ghani [39]		×	Interpolation
Pelcat [40]		×	Interpolation
Sjövall [41]		×	Interpolation
Partanen [42]	×	×	SATD

Table 3.2 Existing HLS approaches for HEVC decoding

	Intra	Inter	Coding tool
Cichoń [43]	×	×	IDCT
Kammoun [44]	×	×	Intra prediction, dequantization, and IDCT
Atitallah [45]	×		Intra prediction

loop unrolling and several Vivado HLS specific optimizations to increase performance.

The work in [35] addressed the *mode decision* (*MD*) process of intra predicted PUs. It proposed to calculate the *sum of absolute differences* (*SAD*) [68], [69] only for 8×8 PUs and construct the larger CUs from them. The reconstruction dependencies have also been removed by utilizing original pixels as reference for adjacent PUs. This increased parallelism but decreased the MD accuracy. The presented HEVC intra encoder was able to encode 1080p@29.7fps on the ARM platform.

Another HLS IP implementation was presented in [36]. The design included six groups of 35 prediction modes and supported 8×8 PUs only. The design was parallelized by removing the recursive search, accumulating the SADs of 8×8 PUs for the larger PUs, and utilizing original pixels as reference for adjacent PUs instead of reconstructions. The presented SoC FPGA (Xilinx ZCU102) solution was able to encode 1080p@29fps. Both [35] and [36] utilized Kvazaar [8]–[10] as a reference for their HLS designs.

3.1.2 DCT and IDCT

The authors in [37] utilized HLS, HW/SW co-design, and a SoC FPGA (Xilinx Zynq ZC702) to implement separate 2-D DCT/IDCT cores for 4×4 and 32×32 TUs for 1080p@30fps video coding. They also ported Kvazaar to the processor of the SoC FPGA in order to verify the HLS-implemented HW in real-time testing. Kvazaar was also used as reference for the implementation. The full 2-D HEVC transform phase consisted of two 1-D DCT and 1-D IDCT units and a transpose circuit between them. The authors also implemented the corresponding HDL design for comparison with the HLS implementation.

The work in [38] presented three HLS implementations for 2-D IDCT that supported all TU sizes. The used HLS tools were Vivado HLS, LegUp [70], and MATLAB Simulink HDL Coder [71]. In addition, the authors have previously implemented the same design using handwritten HDL supporting 2160p48fps, which is compared with the HLS designs supporting 1080p@35fps to 1080p@55fps. The HLS designs consisted of two 1-D transforms for each TU size and a transpose memory between the transforms. The results showed that real-time performance is achievable for IDCT with several different HLS tools, with significantly reduced design time.

3.1.3 Interpolation

Even though this thesis only considers the implementation of an intra HEVC encoder, literature shows that HLS has also been applied for HEVC inter encoding, i.e., interpolation filter for FME and MC. The authors in [39] used Vivado HLS to create multiple versions for the interpolation HW. Each version utilized different optimization techniques, ranging from Vivado HLS specific optimizations and manual loop unrolling to different multiplierless designs. The best design was able to process interpolations for 2160p@45fps video.

HEVC interpolation was also used in [40] as a proof-of-concept algorithm to demonstrate the proposed Design Productivity evaluation when the same design was implemented with VHDL and CAPH HLS [72].

The work in [41] considered fully accurate interpolation filters that were implemented with Catapult HLS [7] and optimized for FME. The design was profiled to be able to filter an adequate number of samples for 2160p@99fps video

on Virtex 6.

3.1.4 Hadamard SATD

The work in [42] considered an HLS implementation for SATD [68], which is used to calculate similarity between pixel blocks. The work combines previously introduces techniques with HLS and introduced implementations for 4×4, 8×8, and 16×16 blocks. The designs were profiled to provide adequate SATD throughput for 2160p@60fps FME.

3.1.5 Decoding tools

Even though this thesis only considers HEVC encoding, HLS implementations for HEVC decoders can also be found in literature. The authors in [43] used Impulse C [73] to implement 2-D IDCT/IDST for all TU sizes. After the design was optimizing using loop unrolling and pipelining, it supported decoding of 1080p@30fps.

The HEVC decoder has also been used as a case study for HLS in [44]. The work proposed HW implementations for IP, dequantization, and IDCT using Vivado HLS. These units are deployed on a SoC FPGA (Xilinx Zynq ZC702), and the SW/HW design is profiled to be able to decode 1080p@17fps video. The same authors [44] also published a dedicated HLS implementation for IP [45]. In this work, they have utilized Vivado HLS and created multiple solutions by adding optimized pragmas incrementally, finally achieving 1080p@51fps performance for the standalone unit, and 1080p@15fps decoding speed with the entire SW/HW design.

3.2 Existing hardware implementations for HEVC entropy coding

The related works for CABAC are listed in Table 3.3. The authors in [46]–[49] presented the whole entropy encoder (both binarization and arithmetic encoding), whereas the remaining works focused only on arithmetic encoding [50], [51] or binarization [52]. All these works included various optimization techniques for removing data dependencies, minimizing critical paths, multi-symbol processing, and parallel processing of bypass-mode.

3.2.1 Whole entropy encoder

The work in [46] described an architecture for the whole CABAC encoder. It detailed how the context modeler and binarizer work in parallel with a connection to the arithmetic encoder via a parallel-in/serial-out unit. The overall process was controlled by a CABAC controller unit. The design aimed to reduce HW resources by applying optimizations via adaptive binarization and memory reduction in context selection. Separate area figures were reported for Virtex 6 FPGA and ASIC for 1600p@60fps video.

The authors in [47] also presented an architecture for the whole CABAC encoder, including binarization, separated paths for regular bins and bypass bin, and a multi-stage arithmetic encoding. Optimizations for the arithmetic encoding were carried out by utilizing incomplete data dependencies considering range updating and less probable symbol bins, shortening critical paths. The work also presented a new architecture for context modeling and binarization, that were developed to ensure an adequate output speed for arithmetic encoding.

In [48], the authors analyzed the challenges of CABAC and proposed parallelism for binary coding and the renormalization of the least probable symbol. The optimization of binarization is also considered by using eight heterogeneous functional units. The design has been developed to support 4K content.

A full CABAC encoder was presented in [49], that supported the throughput requirement of real-time 8K encoding. The architecture showed how the CABAC processing was controlled by a top controlling unit with connections to context modelling, binarization, and arithmetic encoding. The work described optimizations for binarization in the form of pre-allocated context modelling, which simplified the actual binarization of syntax elements. Optimizations for arithmetic encoding included efficient utilization of bypass bins, by splitting them from the multistage arithmetic encoding pipeline and merging them to a later stage.

3.2.2 Separate implementations for arithmetic encoding or binarization

The work in [50] presented an arithmetic encoding part of the full CABAC process. It was designed for a low-power ASIC architecture that supported real-time processing of 8K content. The low-power techniques applied for the design included clock gating and operand isolation. They were used together with

an analysis of the statistical behaviour of inputs.

An arithmetic encoding unit was also presented in [51]. The proposed optimizations included: 1) a parallel four-path data flow for range evaluation supporting high operating frequencies, 2) hardware usage of the four-path range evaluation and actual range calculation is optimized to reduce complexity, 3) modified processing order for updating low register, and 4) increased number of symbols in bypass mode.

The same authors [51] also presented a corresponding binarization unit in [52]. It was designed to meet the high throughput of their arithmetic encoder via fast implementations of binarization, context modeling, and probability model. The work described the decomposition of the processing path into many parallel ones. The proposal was targeted for real-time processing of high-quality 8K content.

3.3 Existing hardware implementations for complete HEVC encoders

As listed in Table 3.4, commercial HW encoders have been unveiled for HEVC, e.g., by NVIDIA (NVENC) [53], Xilinx (LogiCORE IP H.264/H.265 Video Codec Unit) [54], VITEC (e.g. MGW Diamond) [55], ORIVISION (e.g. ZY-EH901) [56], and AJA (Corvid HEVC) [57]. However, the publicly available information of these confidential solutions tends to be limited, so only academic works are considered in this thesis.

The existing academic HW HEVC encoders are listed in Table 3.5. They can be further categorized as: 1) FPGA implementations [58]–[60]; 2) FPGA/ASIC implementations [61]–[63]; and 3) ASIC implementations [64]–[67].

3.3.1 Academic HEVC encoders on FPGA

The authors in [58] presented an implementation capable of supporting both intra and inter encoding for 1080p@60fps video. In addition, the used FPGA can be mounted in a rack supporting as many as 17 FPGAs and 8K@60fps encoding.

An intra encoder for 1080p@30fps video was presented in [59]. It proposed to reduce dependencies in the RDO loop and use Hadamard-based early decision method for higher parallelism.

Table 3.3 Existing CABAC implementations for HEVC

	ASIC	FPGA	Design
Peng [46]	×	×	Full entropy encoder
Zhou [47]	×		Full entropy encoder
Vizzotto [48]	×		Full entropy encoder
Li [49]	×		Full entropy encoder
Ramos [50]	×		Arithmetic encoding
Pastuszak [51]	×		Arithmetic encoding
Pastuszak [52]	×		Binarization

Table 3.4 Existing commercial HEVC encoders on HW

	Intra	Inter	ASIC	FPGA	Performance
NVIDIA [53]	×	×	×		Up to 8K
Xilinx [54]	×	×	×	×	4K@60fps
VITEC [55]	–	–	–	–	4K@60fps
ORIVISION [56]	–	–	–	–	1080p@60fps
AJA [57]	–	–	–	–	4K@59.94fps

Table 3.5 Existing academic HEVC encoders on HW

	Intra	Inter	ASIC	FPGA	Performance
Miyazawa [58]	×	×		×	8K@60fps
Atapattu [59]	×			×	1080p@30fps
Ding [60]	×			×	1080p@60fps
Pastuszak [61]	×		×	×	4K@30fps / 1080p@60fps
Zhang [62]	×		×	×	4K@30fps / 1080p@45fps
Zhang [63]	×		×	×	4K@30fps / 1080p@45fps
Tsai [64]	×	×	×		8K@30fps
Zhu [65]	×		×		1080p@44fps
Huang [66]	×		×		1080p@60fps
Xu [67]	×	×	×		4K@30fps

An intra HEVC encoder capable of 1080p@60fps was presented in [60]. The work introduced basic pixel-level processing elements for various fundamental algorithm modules, from which the encoder was constructed.

3.3.2 Academic HEVC encoders on FPGA/ASIC

The authors in [61] presented an HEVC intra encoder for 1080p@60fps video. The architecture takes advantage of a simplified RDO process, a separate 4×4 reconstruction loop, and an interleaved mode processing.

The same authors in [62], [63] presented two intra encoders for 1080p@45fps video with small differences. They proposed chroma reconstruction based on luma, luma mode preselection, and simplified CABAC rate estimation.

In addition, the works in [61]–[63] also included ASIC results for 4K@30fps video.

3.3.3 Academic HEVC encoders on ASIC

The authors in [64] presented an implementation for 8K@30fps video supporting both intra and inter encoding. It was implemented without the complex 8×8 CUs and 4×4 PUs. It also supported fully parallelized encoding of 64×64, 32×32, and 16×16 intra CUs to meet high throughput.

The work in [65] described an intra encoder capable of 1080p@44fps. The work utilized CU/PU pre-decision to reduce the complexity, but the sequential processing caused throughput degradation.

The ASIC HEVC encoder presented in [66] supported intra encoding of 1080p@60fps video. The work proposed multiple fast algorithms to remove data dependencies and to reduce computational complexity. These fast algorithms included fast *rough mode decision* (*RMD*), *prediction mode interlaced* (*PMI*) RDO mode decision, parallelized context adaption, and chroma-free CU/PU decision.

The intra/inter HEVC encoder presented in [67] supported 4K@30fps video encoding. The work utilized a pyramid motion estimation to reduce search complexity, original pixels for intra mode decision to reduce pipeline stall, and various low-power design techniques.

3.4 How to improve upon prior art

The solutions described in this thesis aim to improve upon the existing works with the following means:

- The use of HLS not only for the data-intensive parts of HEVC (intra prediction, discrete sine/cosine transform, quantization, inverse quantization, inverse discrete sine/cosine transform , and reconstruction), but also for more control-oriented tools, such as intra search control and CABAC. Using HLS for the entire implementation is beneficial from the

perspectives of design effort, complexity, re-usability, ease of modification, and verification time.

- Support for parallel processing of multiple independent CTUs, in order to efficiently fill the pipeline with independent CUs from different CTUs. This approach maximises the utilization of the HW pipeline without breaking any dependencies between adjacent CTUs in a frame or adjacent CUs in a CTU. This way, it improves encoding speed without any negative effects on the coding quality.
- This work is not limited to simulations or assessments of the potential encoding performance. Most of the existing HEVC encoders on HW have been implemented with ASIC technologies, but a proof-of-concept system for real-life HEVC encoding is more easily achievable with FPGA. This also makes it possible to benchmark the coding quality of the implemented HW, instead of estimating the coding quality by introducing possible optimizations or limitations to HM [16].

4 RESULTS OF THE RESEARCH

The main results of this thesis are summarized in the following sections. Each publication is addressed separately, except that two technical papers (Publications V and VIII) are accompanied with associated demonstration papers (Publications VI and IX). The purpose of these two demonstration papers is to describe the technical setup used to validate the proposed functionality in practice.

Section 4.1 presents the results for single HEVC encoding tools. It includes the implementations for intra prediction (Publication I), 2-D DCT/DST (Publication II), and 2-D IDCT/IDST (Publication III). It also describes the results of the conducted test group study (Publication IV). Section 4.2 presents the results for the 1st and 2nd generation versions of the Intra Search Core on an FPGA (Publications V, VI, VII). Section 4.3 presents the results for the 2nd generation Intra Search accelerator in a cloud environment (Publications VIII, IX). Finally, Section 4.4 presents the results for the full HEVC intra encoder, which includes the 3rd generation Intra Search Core and the HLS implementation of CABAC (Publication X). Comparisons to prior art that are not covered in this chapter are included in respective publications.

4.1 HLS implementations of single HEVC intra encoding tools

4.1.1 Intra prediction

The HEVC encoder implementation was started with the HLS implementation of IP described in Publication I. The area and performance results of the implemented IP units are listed in Table 4.1. IP was chosen as the first coding tool for HLS implementation according to the profiling results of Publication I, where IP was shown to account for almost 68% of the whole intra encoding process when the encoding was performed on the CPU.

Table 4.1 HLS implemented HEVC intra prediction units with area and performance figures

Pub.	Coding tool	Supported PBs/TBs	FPGA Device	Area (ALMs)	DSPs	Freq. (MHz)	Performance
I	Intra Prediction + Intra MD (1 pixel per cycle)	All luma and chroma PBs	Cyclone V	8 345	0	150	1080p@6fps
I	Intra Prediction + Intra MD (2 pixels per cycle)	All luma and chroma PBs	Cyclone V	10 815	0	150	1080p@9fps

The work included two versions of the unit. One capable of performing IP and intra MD at a rate of one pixel per cycle, and a second one able to perform two pixels per cycle. These versions could perform intra prediction for 1080p resolution at 6 fps and 9 fps, respectively. The work also included a proof-of-concept HEVC streaming system on a SoC-FPGA, where the implemented HW Intra Prediction unit was used to accelerate the HEVC encoding on CPU. Even though the performance of the coding tool did not achieve real-time performance for 1080p resolution, the design served as a promising starting point for utilizing HLS in fast development and accelerating SW HEVC encoding. In hindsight, the speed was limited by not fully utilizing the internal encoding pipeline for successive CUs, and the limited potential of the low-end Cyclone V FPGA. Nevertheless, the work gave a positive outlook for continuing the development of an embedded HW HEVC encoder.

4.1.2 Transform coding

The next coding tools after IP were DCT and IDCT, described in Publication II and Publication III, respectively. Quantization/Inverse Quantization could have also been selected for implementation, but as the quantization process is more straightforward than DCT/IDCT, the added value of reporting such work would have been limited. The area and performance results of the implemented HW DCT and IDCT units are listed in Table 4.2 and Table 4.3, respectively. The target device for these two works was upgraded from Cyclone V to a mid-end Arria II FPGA device, removing the device performance limitation in Publication I.

The work in Publication II included two separate versions of 2-D DCT/DST units that support all luma and chroma TBs. The individual units presented were 1) a separate 2-D DCT/DST unit dedicated for luma and chroma 4×4 TBs, 2) a low-cost variant of 2-D DCT for larger luma and chroma TBs, and 3) a high-speed

Table 4.2 HLS-implemented HEVC DCT/DST units with area and performance figures

Pub.	Coding tool	Supported PBs/TBs	FPGA Device	Area (ALMs)	DSPs	Freq. (MHz)	Performance
II	2-D DCT/DST (4×4 TB variant)	Luma and chroma 4×4 TBs	Arria II	5 775	0	160	2160p@30fps
II	2-D DCT (low-cost variant)	Luma and chroma 32×32, 16×16, 8×8 TBs	Arria II	4 263	216	100	1080p@60fps
II	2-D DCT/DST (low-cost w/ 4×4 TBs)	All luma and chroma TBs	Arria II	10 038	216	100/160	1080p@60fps
II	2-D DCT (high-speed variant)	Luma and chroma 32×32, 16×16, 8×8 TBs	Arria II	8 114	344	160	2160p@30fps
II	2-D DCT/DST (high-speed w/ 4×4 TBs)	All luma and chroma TBs	Arria II	13 889	344	160	2160p@30fps

variant of 2-D DCT for larger luma and chroma TBs. The individual units support HEVC transform in the worst case at the rate of 2160p@30fps, 1080p@60fps, and 2160p@30fps, respectively. Table 4.2 also lists two combinations of these individual units, creating a complete 2-D DCT/DST unit for all luma and chroma TBs. A comparison to the existing manual RTL designs showed that the HLS implementations were able to produce transformed coefficients with a much higher rate and with better area to performance ratio.

The work in Publication III included two separate units for performing 2-D IDCT/IDST, which are listed in Table 4.3. The first one is dedicated for 4×4 luma and chroma TBs and the second one for the rest of the larger TBs. These two units were capable of performing HEVC inverse transform 2160p@68fps and 2160p@96fps, respectively. Again, the complete 2-D IDCT/IDST unit was able to outperform and achieve better area to performance ratio than the prior art.

4.1.3 User study: HLS vs. manual RTL

To validate the findings of Publication II, a case study was organized in Publication IV to better evaluate the efficiency of HLS and manual RTL design flows. The test subjects were instructed to implement a specified algorithm with both manual RTL and HLS. The coding tool of interest was HEVC 2-DDCT for 8×8 TBs and the subjects were given the opportunity to start with either HLS or manual RTL. The results of the study are summarized in Table 4.4, which shows the area, speed, and hours used by each subject. The bolded values indicate the subjects that started with the specific design flow.

Table 4.3 HLS-implemented HEVC IDCT/IDST units with area and performance figures

Pub.	Coding tool	Supported PBs/TBs	FPGA Device	Area (ALMs)	DSPs	Freq. (MHz)	Performance
III	2-D IDCT/IDST	Luma and chroma 4×4 TBs	Arria II	5 559	0	150	2160p@96fps
III	2-D IDCT	Luma and chroma 32×32, 16×16, 8×8 TBs	Arria II	6 859	344	150	2160p@68fps
III	2-D IDCT/IDST	All luma and chroma TBs	Arria II	12 418	344	150	2160p@68fps

Table 4.4 Area and performance figures from the test group study for HLS and RTL designs, with quality and productivity comparison

	HLS					HLS/RTL Ratio		RTL				
Person #	Area (LUTs)	Speed*	Hours	Quality**	Quality**/ Hours	Quality**	Quality**/ Hours	Area (LUTs)	Speed*	Hours	Quality**	Quality**/ Hours
1	1 860	258	1.7	139	80.2	2.8×	5.9×	**4 000**	**197**	**3.7**	**49**	**13.5**
2	**3 161**	**588**	**4.3**	**186**	**43.7**	2.0×	4.1×	2 068	192	8.6	93	10.8
3	2 814	675	12.3	240	19.5	0.7×	1.1×	**1 292**	**458**	**20.7**	**355**	**17.2**
4	**2 273**	**972**	**9.0**	**427**	**47.5**	3.8×	7.6×	4 431	499	18.0	113	6.3
5	2 768	797	4.8	288	60.0	4.9×	9.4×	**2 066**	**122**	**9.3**	**59**	**6.4**
6	2 463	750	20.0	305	15.2	6.9×	9.0×	**2 722**	**121**	**26.2**	**44**	**1.7**
Average	2 557	673	8.7	264	44.3	2.2×	6.2×	2763	265	14.4	119	9.3

*Mtcoeffs/s **Mtcoeffs/kLUTs

This case study justified that the quality improvement achieved with HLS for HEVC DCT (in Publication II) over the manual RTL implementations was not a random occurrence. Table 4.4 shows that all subjects were able to produce a design with better performance using HLS than with manual RTL. Only one subject got better performance area ratio with manual RTL, but HLS still improved the productivity in that case too. These results are well in line with the experiences of the author and the results of the previously implemented HEVC coding tools.

4.2 FPGA-accelerated HEVC intra search on a compute server

The implementation of the HEVC encoder was continued by moving the focus from single data-intensive coding tools to a complete intra search on FPGA. As the most demanding coding tools of the intra search pipeline were already implemented, Publication V and Publication VI addressed 1) the integration aspects of the IP, DCT, and IDCT, 2) implementation of the missing quantization/dequantization and reconstruction units, and 3) the whole control

logic of intra search. This also shifted the focus from data-intensive algorithms to more control-oriented implementations.

As intra search alone cannot produce the final HEVC bitstream from a direct video input, a CPU of a server/PC was used for input processing, parallelization, chroma coding (1st generation only), CABAC, and HEVC output bitstream processing. Furthermore, in order to access the implemented Intra Search Core on FPGA from CPU, a Linux kernel driver was implemented for sending and receiving data to/from FPGA via PCIe 3.0 x4 and dedicated *direct memory access* (*DMA*) units that are connected to FPGA memory blocks.

Table 4.5 tabulates the base encoding configuration used in benchmarking. In addition, an intra depth range can be configured at run time as specified in Table 4.6. The intra depth range can affect the encoding speed, i.e., larger blocks are more data intensive, the number of smaller blocks is larger, and a larger range affects the number of possible configurations.

4.2.1 1st generation Intra Search Core

The work carried out for the 1st generation Intra Search Core is described in Publication V. This first complete HLS implementation of the Intra Search Core was designed to process data at CTU level to maximize the internal parallelization of CUs within a single core. This parallelization method was also chosen because it does not break any adjacent CTU dependencies and thus does not degrade the coding quality. The 1st generation Intra Search Core was able to perform intra search for eighth individual CTUs in parallel. The number of parallel CTUs was limited by the intra search control and the size of intra search memories. The number of FPGAs per CPU was also limited to a single board by the initial Linux kernel driver.

The area and performance figures of the 1st generation Intra Search Core are given with the specified configurations in Table 4.7 and Table 4.8, respectively. A single Arria 10 FPGA chip can accommodate two parallel Intra Search cores, and the area is given for the whole design. The performance was separately benchmarked with a single and two cores per FPGA using sequences from [15]. With the wider intra depth range of 1-3 and *quantization parameter* (*QP*) of 32, the two cores were able to perform 1080p encoding at 109 fps. Using the narrower intra depth range of 2-3, the two cores were able to perform 2160p encoding at 41 fps. Although

Table 4.5 Supported base configuration for the proposed HW HEVC intra encoder

Feature	Base configuration
Profile	Main
Internal bit depth	8
Color format	4:2:0
Coding mode	Intra
IP modes	DC, planar, 33 angular
Intra Search	Full
Transform	Integer DCT
Mode decision	SAD
Parallelization	WPP, OWF, Tiles
Sign bit hiding	Disabled
Rate Control	Disabled
RD optimized quantization	Disabled

Table 4.6 Corresponding intra depth ranges and luma PU and TU sizes

Intra depth range	Enabled PUs & TUs
1-1	32×32
1-2	32×32, 16×16
1-3	32×32, 16×16, 8×8
1-4	32×32, 16×16, 8×8, 4×4
2-2	16×16
2-3	16×16, 8×8
2-4	16×16, 8×8, 4×4
3-3	8×8
3-4	8×8, 4×4
4-4	4×4

some HEVC encoding functionalities were implemented on CPU, the 1st generation Intra Search Core was able to more than double the encoding speed over CPU only encoding. Furthermore, the implementation showed competitive performance over existing FPGA and ASIC implementations.

4.2.2 2nd generation Intra Search Core

The work for the 2nd generation Intra Search Core in Publication VI was motivated by the bottlenecks found in the 1st generation core. The work thus described the solutions for the encountered limitations, added support for HW chroma coding, and proposed optimizations for faster encoding speeds.

The encoding system in Publication V utilized available CPU processing power to maximize the coding speed because the 1st generation Intra Search Core only

Table 4.7 The resource consumption of the proposed 1st and 2nd generation Intra Search Cores

Pub.	Design	Supported CBs	FPGA Device	Area (ALUTs)	DSPs	Freq. (MHz)
IV	1st generation Intra Search Core **×2**	All luma CBs	Arria 10	308k	862	125
V	2nd generation Intra Search Core **×3**	All luma and chroma CBs	Arria 10	552k	1 227	175

Table 4.8 The performance of the proposed 1st and 2nd generation Intra Search Cores

Pub.	Intra depth range	QP	# FPGAs	# Intra Search Cores	Performance
IV	1-3	32	1	1st gen ×1	1080p@81fps
IV	1-3	32	1	1st gen ×2	1080p@109fps
IV	2-3	32	1	1st gen ×1	2160p@31fps
IV	2-3	32	1	1st gen ×2	2160p@41fps
V	1-3	32	1	2nd gen ×1	2160p@26fps
V	1-3	32	1	2nd gen ×2	2160p@50fps
V	1-3	32	1	2nd gen ×3	2160p@64fps
V	1-3	32	2	2nd gen ×2	2160p@52fps
V	1-3	32	2	2nd gen ×4	2160p@97fps
V	1-3	32	2	2nd gen ×6	2160p@123fps

came with limited parallelism and thus utilized only a portion of the CPUs threading performance. To that end, the support of parallel CTUs in a single core was increased from 8 to 16. In addition, the processing time in different parts of the HW pipeline was equalized by separating the 2-D process of DCT and IDCT units into individual pipeline stages. The operation of the Linux kernel driver was also upgraded to enable multiple PCIe boards and interrupt handling. In addition, the overall optimizations improved the HW utilization and enabled a higher clock frequency.

The area and performance figures of the 2nd generation core are also given in Table 4.7 and Table 4.8, respectively. The duplication of the DCT and IDCT units increased the number of ALUTs per core only minimally and decreased the number of DSPs per core. The maximum clock frequency increased from 125 MHz to 175 MHz, which improved the coding speed by 40%. The performance of the 2nd generation core was benchmarked using the base encoding configuration with 1-3 intra depth range only. As three 2nd generation cores could now be accommodated per FPGA, the maximum number of parallel Intra Search Cores in the encoding system with two FPGAs increased from 2 to 6. This increased the maximum number of parallel processed CTUs from 16 to 96. With the sequences from [15], this system was able to encode 2160p video at 123 fps on average.

4.2.3 Live demonstration of the Intra Search Core

The proof-of-concept encoding system proposed in Publication VI was also validated with a live demonstration described in Publication VII. The demonstration showcased how a single *personal computer* (*PC*) with two PCIe FPGAs can encode three 4K streams in real time. The applied 4K cameras supported raw video output via HDMI. The outputs were captured by HDMI capture cards and transmitted to the PC via USB. The encoded videos were finally streamed to three separate laptops for playback over a network using *real-time transport protocol* (*RTP*).

4.3 FPGA-accelerated HEVC intra search in a cloud environment

Publication VIII is about adding support for a 40 *Gigabit Ethernet* (*GbE*) connection for the 2nd generation Intra Search Core whereas publication IX describes the live demonstration of the developed cloud encoding system. These works were motivated by the objective of adding even more cores to the encoding system, without a specialized motherboard that can fit and support multiple PCIe boards. The combination of an Ethernet connection and a network switch enables practically any number of heterogeneous FPGAs to be added to the system. The challenge with the network connection resides in implementing a full protocol stack for communication and application abstraction on FPGA. This effort was minimized by utilizing SDN switches, in which the connections and data flows are programmable by a separate controller. This allows the network interface on FPGA to be at very low level, which in turn saved a major portion of the FPGA resources and implementation time. The publication also describes the implementation of a resource manager and how it controls the SDN flows. The advanced utilization of these SDN flows also enabled the manager to modify the used resources at run-time, i.e., allocate varying number of FPGAs per server, switch FPGAs on the fly without disrupting the HEVC streaming, and prioritizing different streams being encoded.

Table 4.9 shows the performance of the same 2nd generation Intra Search Core when using either PCIe 3.0 x4 or 40 GbE. The benchmarking was carried out with a single FPGA board, three cores, and the same server CPU. The intra depth range

Table 4.9 The performance of proposed Intra Search Cores using PCIe and 40GbE

Pub.	Intra depth range	QP	Interface	# FPGAs	# Intra Search Cores	Performance	Avg. Frame Latency (ms)	Avg. CPU Utilization (%)
V	2-3	22	PCIe 3.0 x4	1	3	2160@70fps	20	46
VIII	2-3	22	40GbE	1	3	2160@61fps	28	48

of 2-3 and the QP of 22 were used. The results show that the system with a 40 GbE was also able to encode 2160p video at 61fps, with similar CPU utilization. The performance of the network connected FPGA was mainly limited by the utilization of $2 \times$ 10 GbE connections on the server, in comparison with the 32Gbps PCIe connection. Having access to a 40 GbE network card on the server would have given identical performance.

4.4 Complete HEVC intra encoder on FPGA

The final publication, Publication X, describes the 3rd generation Intra Search Core and the implementation of the CABAC coding tool. In all previous systems, CABAC was performed on CPU, but in this work, CABAC was implemented as a dedicated core that operates in parallel with the Intra Search Core. The implementation of the intra search control logic and CABAC gave a thorough understanding of HLS competitiveness with sequential and control-intensive implementations.

The system architecture of the implemented HEVC Intra encoder is presented in Figure 4.1. It depicts the interconnectivity of the CPU and FPGAs. Each Intra Encoder instance consists of a single Intra Search Core and two parallel CABAC Cores (*TOP* and *BTM*) that divide the CTU processing.

The optimizations of the 3rd generation Intra Search Core include the improved pipelining of the IP unit to remove initial latencies of adjacent PUs and optimized area and memory usage of the prediction buffer. The 2nd generation Intra Search Core supported chroma coding, but only together with luma CBs. To skip the processing of unused chroma CBs, the 3rd generation core reconstructs the corresponding chroma results from the luma results. The resource utilization of the 3rd generation core is presented in Table 4.10. A single 3rd generation core now utilizes 180k ALUTs (1 ALM $\simeq$ 2 ALUTs) and 523 DSPs. For comparison, previously a single 2nd generation core 4.7 utilized 184k ALUTs and 409 DSPs.

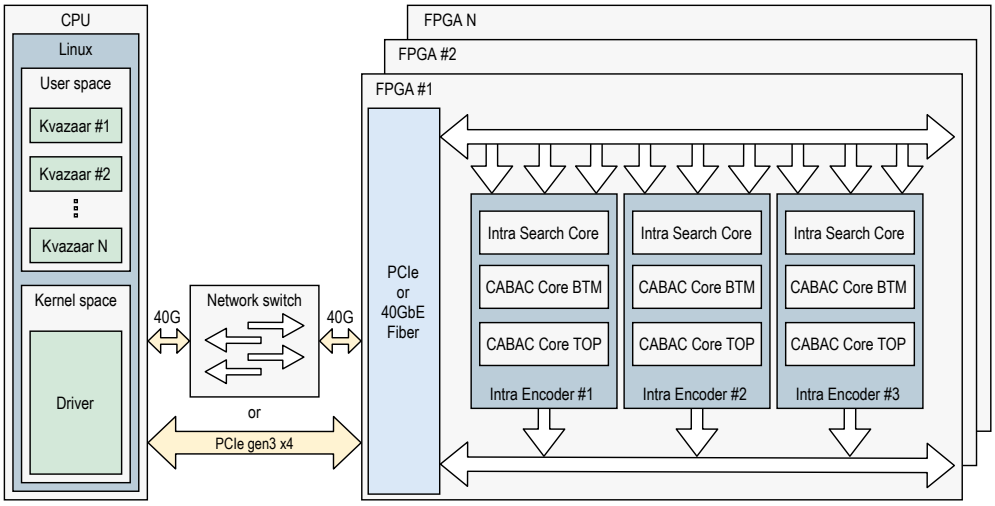


Figure 4.1 System architecture of the proposed HEVC intra encoder on FPGA [Publication X].

Table 4.10 The area figures of the proposed fully HLS implemented HEVC intra encoder on FPGA

Pub.	Design	Supported CBs	FPGA Device	Area (ALMs)	DSPs	Freq. (MHz)
X	3rd generation Intra Search Core **×1**	All luma and chroma CBs	Arria 10	90k	523	190
X	CABAC Core **×1**	All luma and chroma CBs	Arria 10	11k	3	190/266
X	3rd generation Intra Search Core **×3**, CABAC Core **×6**, Surrounding connectivity	All luma and chroma CBs	Arria 10	378k	1 468	190/266

Table 4.11 Performance comparison of the proposed standalone CABAC Core [Publication X] to related work

	Peng [46]	Zhou [47]	Vizzotto [48]	Li [49]	Ramos [50]	Pastuszak [51]	Pastuszak [52]	Publication X
Technology	FPGA/ASIC	ASIC	ASIC	ASIC	ASIC	ASIC	ASIC	FPGA
Design	Full entropy encoder	Full entropy encoder	Full entropy encoder	Full entropy encoder	Arithmetic encoding	Arithmetic encoding	Binarization	Full entropy encoder
Throughput Mbins/s	439	1836	900	2410	1120	4190	6755	4152
Bins/cycle	1.18	4.37	2.34	4.67	4	7.35	9.65	15.6

The maximum clock frequency also increased from 175 MHz to 190 MHz, which further improved the performance by almost 9%.

The work describes in detail how both CABAC binarization and CABAC

Table 4.12 The 2160p performance of the proposed prototype HEVC intra encoding system [Publication X]

Quantization parameter	Intra depth range									
	1-1	1-2	1-3	1-4	2-2	2-3	2-4	3-3	3-4	4-4
QP22	147	135	101	50	139	115	56	98	53	51
QP27	174	160	114	69	168	144	81	125	68	59
QP32	178	168	130	98	174	159	119	131	87	60
QP37	184	171	143	126	173	163	144	132	103	60
Average fps	171	159	122	86	164	145	100	122	78	58
			Fast			Ultrafast				

Table 4.13 Performance comparison with related work [Publication X]

	Technology	Intra/Inter	FPGA Performance	System/ASIC performance	Cells	DSPs
Miyazawa [58]	FPGA/System	x/x	1080p@60fps	8K@60fps	-	-
Atapattu [59]	FPGA	x/-	1080p@30fps	-	-	-
Ding [60]	FPGA	x/-	1080p@60fps	-	63450 LUTs	721
Pastuszak [61]	FPGA/ASIC	x/-	1080p@60fps	4K@30fps	93 184 ALUTs	481
Zhang [62]	FPGA/ASIC	x/-	1080p@45fps	4K@30fps	201 823 ALUTs	1253
Zhang [63]	FPGA/ASIC	x/-	1080p@45fps	4K@30fps	195 883 ALUTs	1244
Tsai [64]	ASIC	x/x	-	8K@30fps	-	-
Zhu [65]	ASIC	x/-	-	1080p@44fps	-	-
Huang [66]	ASIC	x/-	-	1080p@60fps	-	-
Xu [67]	ASIC	x/x	-	4K@30fps	-	-
Proposed	FPGA/System	x/-	4K@60fps	8K@30fps	377 649 ALMs	1468

arithmetic encoding have entirely been implemented with HLS. This was the first HLS implementation for CABAC in literature. The area figures for a single CABAC Core are listed in Table 4.10. The performance of the CABAC Core was reported in the publication for different configurations, but individual comparison with related work was omitted since the focus was on the overall HEVC encoding performance. However, the omitted comparison is given here in Table 4.11. The related work consists of only manual RTL implementations and a collection of designs implementing either CABAC arithmetic encoding, binarization, or a full CABAC entropy encoder. The whole CABAC processing power of the encoding system (12 $\times$ CABAC Cores in total) presented in Publication X, benchmarked by using the worst-case sequence (Beauty [15]), and averaged over every intra depth range, had a throughput of 4152 Mbins/s and 15.6 Bins/cycle. The design shows equal or higher performance in comparison with the related work.

As can be seen in Figure 4.1, two CABAC Cores are dedicated per a single Intra

Search Core. This configuration was found to be ideal for the proposed system. The number of CABAC cores per Intra Search Core can was made configurable in the RTL synthesis tool, allowing optimization of area or performance for future designs. Table 4.10 lists the area usage for three Intra Search Cores and six CABAC Cores when fitted on a single FPGA.

Tables 4.12 and 4.13 report the performance of the entire encoding system, and comparison to related work, of the work presented in Publication X. It consists of two FPGAs, a Nokia AirFrame Cloud Server equipped with 2.4 GHz dual 14-core Intel Xeon processors, 6× Intra Search Cores, and 12× CABAC Cores. The average coding speed is given with all 4K (3840 × 2160) test sequences [15], for four different QP values and for each intra depth range. The base configuration is the same as before (Table 4.5). The highlighted *Fast* and *Ultrafast* presets comply with the original Kvazaar HEVC encoder [9], [10]. With these presets, the average coding speed across all QP values is 2160p@122fps and 2160p@145fps, respectively.

5 CONCLUSION

The main objective of this thesis was to examine the feasibility of HLS when implementing an embedded HEVC intra encoder on FPGA. Real-time HEVC encoding was selected as an application, because of its data and control-intensive characteristics. HLS was shown to provide short implementation and verification times, easy portability between FPGAs, increased design reusability and customization, and competitive performance with prior art.

5.1 Discussion about lessons learned

In the beginning, the author was already familiar with programming languages like C/C++, but also with traditional RTL design methods for FPGAs, which may influence the views presented in this section. On the other hand, the author also obtained some second-hand knowledge from designers with different backgrounds and also from the group study organized in Publication IV. However, the actual learning process of HLS was started from zero for this thesis. It did take some time to learn the best practises to achieve good and consistent results, but even the first designs created with somewhat limited HLS knowledge showed comparable results against the related work. In the end, the time used for learning HLS was paid off, as the design times of later designs were shortened significantly. Although HLS tends to improve the QoR more than absolute performance, the results listed in this thesis also showed competitive video coding speed over related work.

The HLS code for this work was written mostly using C for algorithms taken from Kvazaar [10]. C++ was used for templates, template recursion, and classes, instead of structures, where additional functionality is closely related to the structured data, e.g., reading a configuration bit vector and parsing the data to class variables. Furthermore, SystemC was utilized in Publication I, but for the purpose of early system simulations of the first HEVC coding tool. The RTL of this coding

tool was still generated using untimed C/C++. A more accurate control of the outcome and introduction of clock in the design itself with SystemC was not found necessary. The coding overhead (actual lines of code) with SystemC would have also been larger than that of the purely untimed C/C++.

The original reference code was used as is in almost every test bench for generating golden reference data with a random input. On the other hand, the amount of modifications needed for the HLS implementations was dependent on the coding tool. SAD, quantization, dequantization, and even non-time critical parts of CABAC did not need a lot of effort, but some highly vectorized coding tools like IP, DCT/IDCT, intra coding control, and time critical parts of CABAC, needed substantial re-writing. The necessary modification for all units, moving from the reference to the implementation, was the interface, integration to the pipeline, and passing of necessary data for operation, i.e., data pointers to big data structures cannot be reused from the original reference code as such. Introducing parallelism to a single coding tool turned out to be the most time-consuming part, especially when applying loop unroll for a loop was not enough, e.g., IP predicting all modes and multiple pixels in parallel, DCT/IDCT having parallel 1D transforms with a constant 32 pixel transform independent of TB size, and CABAC with efficient coefficient binarization and arithmetic encoding. As a downside, these modifications separate the HLS code from the original reference code. Although the entire HW implementation is still executable on CPU, it is no longer optimized for it.

For this thesis, the verification effort was minimized with the combination of a ready-made reference code and Catapult HLS tool. Catapult supports the utilization of a high-level C/C++ test bench that can be used for verification of both the behavioural code and the generated RTL. In practice, only the verification of the behavioural model was necessary, as the generated RTL preserves exactly the same functionality. Some corner cases were found, e.g., issues with type casting and overflows, but they can be removed with careful and proper coding style. Furthermore, for the system-level verification, the HW and SW reference encoders were run in parallel and validated for functionality.

The author sees that HLS code and tools provide a shorter path to HW design than learning RTL design with traditional HDLs. However, experiences with HLS might vary between HLS tools because they offer different features and target

technologies. This is also true for downstream synthesis tools generating netlist code from HDLs. The ease of verification also advocates the usage of HLS.

5.2 Research question 1: Feasibility of HLS for implementing the HEVC encoder

It is notable that the main novelty of this thesis comes from the reported feasibility and usability of HLS in implementing the HEVC intra encoder. To the best of knowledge, this thesis presents the first known HEVC intra encoder that is fully implemented through HLS, even though there are multiple prior HEVC intra encoder implementations on ASIC and FPGA. Furthermore, when focusing solely on the implemented HEVC intra encoder, the presented design offers unseen scalability via number of server CPUs, accelerator FPGA boards, and HW encoder instances per FPGA. In addition, it offers connectivity via PCIe or 40 GbE port and flexibility to switch execution between SW and HW. This was also very beneficial during design time, from component and connectivity/protocol verification to more complex system verification.

The implemented encoding system shows competitive performance over the existing FPGA and ASIC encoder implementations. Finally, the support for network connectivity in the design together with SDN switches and a resource manager, the coding performance can be easily scaled up by adding practically any number of network-connected FPGA cards and servers to the system. This thesis showed that the designer writing code for an HLS tool can translate behavioural source code to structural RTL and optimize it efficiently.

5.3 Research question 2: Area and performance of the HLS implementations against existing work

In the end, HLS was very feasible for the implementation of an HEVC intra encoder on an FPGA and showed comparable or better results when compared to prior art. The justification for this conclusion gathered from the included publications is covered in the following paragraph. The area and performance of the HLS-implemented coding tools and larger coding entities were reported and compared with prior art in respective publications.

The Publications I- III showed that HLS is very suitable for data-intensive HEVC coding tools. The group study organized in Publication IV validated that HLS can provide results for HW designs faster. Publications V - VII showcased that HLS can also be used for more control-oriented designs. These publications also stated that HLS does not only provide better QoR against manual RTL, but also competitive performance. Publications VIII and IX demonstrated the possibility to customize the interfacing logic for the FPGA and the HW encoding architecture. In these works, HLS was specifically used for implementing the Ethernet packet parser and generator. Finally, Publication X again showed the feasibility of HLS for a control-oriented algorithm with the full implementation of CABAC encoding and its integration with the existing Intra Search Core. This publication also further showed how HLS produced better or comparable QoR and performance against existing manual RTL works.

5.4 Research question 3: Final conclusion

The HEVC encoder, with its parallel HW instances, is a very complex design as a whole and manually controlling all task allocations and scheduling would have been very laborious. This thesis proves that with HLS, the shorter development time and better complexity control does not come at the cost of coding performance or increased logic area.

REFERENCES

[1] G. Martin and G. Smith, "High-level synthesis: Past present and future," *IEEE Des. Test. Comput.*, vol. 26, no. 4, pp. 18–25, Jul. 2009.

[2] P. Coussy, D. D. Gajski, M. Meredith, and A. Takach, "An introduction to high-level synthesis," *IEEE Des. Test. Comput.*, vol. 26, no. 4, pp. 8–17, Jul. 2009.

[3] H. Ren, "A brief introduction on contemporary high-level synthesis," in *Proc. Int. Conf. IC Design Technol.*, Austin, Texas, USA, Jun. 2014.

[4] ITU-T and ISO/IEC, *High Efficiency Video Coding*, document ITU-T Rec. H.265 and ISO/IEC 23008-2 (HEVC), Nov. 2019.

[5] G. J. Sullivan, J. R. Ohm, W. J. Han, and T. Wiegand, "Overview of the high efficiency video coding (HEVC) standard," *IEEE Trans. Circuits Syst. Video Technol.*, vol. 22, no. 12, pp. 1649–1668, Dec. 2012.

[6] 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, pp. 1885–1898, Dec. 2012.

[7] Siemens. "Catapult high-level synthesis and verification," [Online]. Available: https://eda.sw.siemens.com/en-US/ic/catapult-high-level-synthesis/ (visited on Nov. 4, 2021).

[8] A. Lemmetti, M. Viitanen, A. Mercat, and J. Vanne, "Kvazaar 2.0: Fast and efficient open-source HEVC inter encoder," in *Proc. ACM Multimedia Syst. Conf.*, Istanbul, Turkey, Jun. 2020.

[9] A. Lemmetti, A. Koivula, M. Viitanen, J. Vanne, and T. D. Hämäläinen, "AVX2–optimized Kvazaar HEVC intra encoder," in *Proc. IEEE Int. Conf. Image Processing*, Phoenix, Arizona, USA, Sept. 2016.

[10] Ultra Video Group. "Kvazaar HEVC encoder," [Online]. Available: https://github.com/ultravideo/kvazaar (visited on Nov. 5, 2021).

[11] 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, pp. 1792–1801, Dec. 2012.

[12] S. T. March and G. F. Smith, "Design and natural science research on information technology," *Decision Support Systems*, vol. 15, no. 4, pp. 251–266, Dec. 1995.

[13] A. Hevner, A. R, S. March, S. T, Park, J. Park, Ram, and Sudha, "Design science in information systems research," *Management Information Systems Quarterly*, vol. 28, no. 1, pp. 75–106, Mar. 2004.

[14] Altera. "FPGA architecture white paper," [Online]. Available: https://www.intel.com/content/dam/www/programmable/us/en/pdfs/literature/wp/wp-01003.pdf (visited on Feb. 23, 2022).

[15] A. Mercat, M. Viitanen, and J. Vanne, "UVG dataset: 50/120fps 4K sequences for video codec analysis and development," in *ACM Multimedia Syst. Conf.*, Istanbul, Turkey, Jun. 2020.

[16] Joint Collaborative Team on Video Coding Reference Software. "Ver. HM 16.24," [Online]. Available: http://hevc.hhi.fraunhofer.de/ (visited on Dec. 3, 2021).

[17] G. Bjøntegaard, *Calculation of average PSNR differences between RD curves*, Document VCEG-M33, Austin, Texas, USA, Apr. 2001, pp. 1–4.

[18] F. Bossen, *Common test conditions and software reference configurations*, document JCTVC-L1100, Geneva, Switzerland, Jan. 2013.

[19] R. Nane, V.-M. Sima, C. Pilato, J. Choi, B. Fort, A. Canis, Y. T. Chen, H. Hsiao, S. Brown, F. Ferrandi, J. Anderson, and K. Bertels, "A survey and evaluation of FPGA high-level synthesis tools," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 35, no. 10, pp. 1591–1604, Oct. 2016.

[20] Xilinx. "Vitis Platform," [Online]. Available: https://www.xilinx.com/products/design-tools/vitis/vitis-platform.html (visited on Nov. 17, 2021).

[21] J. Cong, B. Liu, S. Neuendorffer, J. Noguera, K. Vissers, and Z. Zhang, "High-level synthesis for FPGAs: From prototyping to deployment," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 30, no. 4, pp. 473–491, Apr. 2011.

[22] S. Lahti, J. Vanne, and T. D. Hämäläinen, "Designing a clock cycle accurate application with high-level synthesis," in *Proc. Annu. Conf. IEEE Ind. Electronics Soc*, Florence, Italy, Dec. 2016.

[23] H. Foster. "The 2018 wilson research group functional verification study," [Online]. Available: https://blogs.sw.siemens.com/verificationhorizons/2018/12/04/part-3-the-2018-wilson-research-group-functional-verification-study/ (visited on Nov. 18, 2021).

[24] C. Systems. "Cisco visual networking index: Forecast and trends 2017-2022," [Online]. Available: http://web.archive.org/web/20181213105003/https://www.cisco.com/c/en/us/solutions/collateral/service-provider/visual-networking-index-vni/white-paper-c11-741490.pdf (visited on Nov. 23, 2021).

[25] ITU-T and ISO/IEC, *Advanced video coding for generic audiovisual services*, document ITU-T Rec. H.264 and ISO/IEC 14496-10 (AVC), Mar. 2009.

[26] I. K. Kima, J. Min, T. Lee, W. J. Han, and J. Park, "Block partitioning structure in the HEVC standard," *IEEE Trans. Circuits Syst. Video Technol.*, vol. 22, no. 12, pp. 1697–1706, Dec. 2012.

[27] 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, pp. 1029–1041, Dec. 2013.

[28] V. Sze and M. Detlev, "Entropy coding in HEVC," *High Efficiency Video Coding (HEVC)*. Springer, 2014, pp. 209–274.

[29] H. Schwarz, T. Schierl, and D. Marpe, "Block structures and parallelism features in HEVC," *High Efficiency Video Coding (HEVC)*. Springer, 2014, pp. 49–90.

[30] C. C. Chi, M. Alvarez-Mesa, B. Juurlink, G. Clare, F. Henry, S. Pateux, and T. Schierl, "Parallel scalability and efficiency of HEVC parallelization

approaches," *IEEE Trans. Circuits Syst. Video Technol.*, vol. 22, no. 12, pp. 1827–1838, Dec. 2012.

[31] G. Clare, F. Henry, and S. Pateux, *Wavefront parallel processing for hevc encoding, and decoding*, Document JCTVC-F274, Torino, Italy, Jul. 2011.

[32] C. C. Chi, M. Alvarez-Mesa, B. Juurlink, V. George, and T. Schierl, "Improving the parallelization efficiency of HEVC decoding," in *Proc. IEEE Int. Conf. Image Process.,* Orlando, Florida, USA, Sept. 2012.

[33] M. Inc. "X265 HEVC Encoder / H.265 Video Codec.," [Online]. Available: https://bitbucket.org/multicoreware/x265_git/downloads/ (visited on Dec. 3, 2021).

[34] E. Kalali and I. Hamzaoglu, "FPGA implementation of HEVC intra prediction using high-level synthesis," in *Proc. Int. Conf. Consum. Electronics - Berlin*, Berlin, Germany, Sept. 2016.

[35] Z. Cui, J. Xia, Y. Wang, G. Shi, and W. Yan, "Design of HEVC intra model decision based on Zynq," in *Proc. Int. Conf. Real-time Comput. Robot.,* Irkutsk, Russia, Aug. 2019.

[36] W. Chen, Q. He, S. Li, B. Xiao, M. Chen, and Z. Chai, "Parallel implementation of H.265 intra-frame coding based on FPGA heterogeneous platform," in *Proc. Int. Conf. High Perform. Comput. Commun.,* Yanuca Island, Cuvu, Dec. 2020.

[37] B. Mohamed, A. Elsayed, O. Amin, E. Khafagy, M. Abdelrasoul, A. Shalaby, and M. S. Sayed, "High-level synthesis hardware implementation and verification of HEVC DCT on SoC-FPGA," in *Proc. Int. Comput. Eng. Conf.,* Cairo, Egypt, Dec. 2017.

[38] E. Kalali and I. Hamzaoglu, "FPGA implementations of HEVC inverse DCT using high-level synthesis," in *Proc. Conf. Des. Architectures Signal Image Process.,* Krakow, Poland, Sept. 2015.

[39] F. A. Ghani, E. Kalali, and I. Hamzaoglu, "FPGA implementations of HEVC sub-pixel interpolation using high-level synthesis," in *Proc. Int. Conf. Des. Technol. Integr. Syst. Nanoscale Era*, Istanbul, Turkey, Apr. 2016.

[40] M. Pelcat, C. Bourrasset, L. Maggiani, and F. Berry, "Design productivity of a high level synthesis compiler versus HDL," in *Proc. Int. Conf. Embedded Comput. Syst.: Architectures, Modeling and Simul.*, Agios Konstantinos, Greece, Jul. 2017.

[41] P. Sjövall, M. Rasinen, A. Lemmetti, and J. Vanne, "High-level synthesis implementation of an accurate HEVC interpolation filter on an FPGA," in *Proc. IEEE Nordic Circuits Syst. Conf.*, Oslo, Norway, Oct. 2021.

[42] T. Partanen, A. Lemmetti, P. Sjövall, and J. Vanne, "High-level synthesis implementation of transform-exempted SATD architectures for low-power video coding," in *Proc. IEEE Int. Symp. Circ. Syst.*, Daegu, Korea, May 2021.

[43] S. Cichoń and M. Gorgoń, "H.265 inverse transform FPGA implementation in Impulse C," in *Proc. Federated Conf. Comput. Sci. Inf. Syst.*, Prague, Czech Republic, Sept. 2017.

[44] M. Kammoun, A. Ahmed, A. Karim, and A. Rabie, "Case study of an HEVC decoder application using high-level synthesis: Intraprediction, dequantization, and inverse transform blocks," *J. Electronic Imaging*, vol. 28, no. 3, pp. 1–11, May 2019.

[45] A. B. Atitallah and M. Kammoun, "High-level design of HEVC intra prediction algorithm," in *Proc. Int. Conf. Adv. Technologies Signal Image Process.*, Sousse, Tunisia, Sept. 2020.

[46] B. Peng, D. Ding, X. Zhu, and L. Yu, "A hardware CABAC encoder for HEVC," in *Proc. Int. Symp. Circuits Syst.*, Beijing, China, May 2013.

[47] D. Zhou, J. Zhou, W. Fei, and S. Goto, "Ultra-high-throughput VLSI architecture of H.265/HEVC CABAC encoder for UHDTV applications," *IEEE Trans. Circuits Syst. Video Technol.*, vol. 25, no. 3, pp. 497–507, Mar. 2015.

[48] B. Vizzotto, V. Mazui, and S. Bampi, "Area efficient and high throughput CABAC encoder architecture for HEVC," in *Proc. Int. Conf. Electronics, Circuits, Syst.*, Cairo, Egypt, Dec. 2015.

[49] W. Li, X. Yin, X. Zeng, X. Yu, W. Wang, and Y. Fan, "A VLSI implement of CABAC encoder for H.265/HEVC," in *Proc. Int. Conf. Solid-State Integr. Circuit Technol.*, Qingdao, China, Oct. 2018.

[50] F. L. L. Ramos, J. Goebel, B. Zatt, M. Porto, and S. Bampi, "Low-power hardware design for the HEVC binary arithmetic encoder targeting 8K videos," in *Proc. Symp. Integr. Circuits Syst. Des.*, Belo Horizonte, Brazil, Aug. 2016.

[51] G. Pastuszak, "Multisymbol architecture of the entropy coder for H.265/HEVC video encoders," *IEEE Trans. Very Large Scale Integration Syst.*, vol. 28, no. 12, pp. 2573–2583, Dec. 2020.

[52] G. Pastuszak, "Generative multi-symbol architecture of the binary arithmetic coder for UHDTV video encoders," *IEEE Trans. Circuits Syst. I: Regular Papers*, vol. 67, no. 3, pp. 891–902, Mar. 2020.

[53] NVIDIA. "Video Codec SDK," [Online]. Available: https://developer.nvidia.com/nvidia-video-codec-sdk (visited on Dec. 8, 2021).

[54] Xilinx. "Video Processing Subsystem," [Online]. Available: https://www.xilinx.com/products/intellectual-property/v-vcu.html#overview (visited on Dec. 8, 2021).

[55] VITEC. "MGW Diamond," [Online]. Available: https://www.vitec.com/product/mgw-diamond (visited on Mar. 4, 2022).

[56] ORIVISION. "HDMI Video Encoder," [Online]. Available: https://www.orivision.com.cn/collections/hdmi-video-encoder (visited on Dec. 8, 2021).

[57] AJA. "Corvid HEVC," [Online]. Available: https://www.aja.com/products/corvid-hevc (visited on Dec. 8, 2021).

[58] 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 Symp.*, San Jose, California, USA, Dec. 2013.

[59] 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. Appl.-specific Syst. Architectures Processors*, London, United Kingdom, Jul. 2016.

[60] D. Ding, S. Wang, Z. Liu, and Q. Yuan, "Real-time H.265/HEVC intra encoding with a configurable architecture on FPGA platform," *Chinese J. Electron.*, vol. 28, no. 5, pp. 1008–1017, Sept. 2019.

[61] 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, pp. 210–222, Jan. 2016.

[62] Y. Zhang and C. Lu, "High-performance algorithm adaptations and hardware architecture for HEVC intra encoders," *IEEE Trans. Circuits Syst. Video Technol.*, vol. 29, no. 7, pp. 2138–2145, Jul. 2019.

[63] Y. Zhang and C. Lu, "Efficient algorithm adaptations and fully parallel hardware architecture of H.265/HEVC intra encoder," *IEEE Trans. Circuits Syst. Video Technol.*, vol. 29, no. 11, pp. 3415–3429, Nov. 2019.

[64] S.-F. Tsai, C.-H. Tsai, and L.-G. Chen, "Encoder hardware architecture for HEVC," *High Efficiency Video Coding (HEVC)*. Springer, 2014, pp. 209–274.

[65] 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 South Pacific Des. Automat. Conf.*, Singapore, Jan. 2014.

[66] X. Huang, H. Jia, B. Cai, C. Zhu, J. Liu, M. Yang, D. Xie, and W. Gao, "Fast algorithms and VLSI architecture design for HEVC intra-mode decision," *J. Real-Time Image Process.*, vol. 12, no. 6, pp. 285–302, Aug. 2016.

[67] K. Xu, Y. Li, B. Huang, X. Liu, H. Wang, Z. Wu, Z. Yan, X. Tu, T. Wu, and D. Zeng, "A low-power 4096x2160@30fps H.265/HEVC video encoder for smart video surveillance," in *Proc. Int. Symp. Low Power Electronics Des.*, New York, New York, USA, Jul. 2018.

[68] I. E. Richardson, *The H.264 advanced video compression standard*. John Wiley & Sons Ltd, 2011.

[69] G. J. Sullivan and T. Wiegand, "Rate-distortion optimization for video compression," *IEEE Signal Processing Mag.*, vol. 15, no. 6, pp. 74–90, Nov. 1998.

[70] LegUp. "Legup high-level synthesis software," [Online]. Available: https://www.legupcomputing.com/ (visited on Jan. 18, 2022).

[71] MathWorks. "HDL coder," [Online]. Available: https://www.mathworks.com/products/hdl-coder.html (visited on Jan. 18, 2022).

[72] J. Sérot, F. Berry, and S. Ahmed, "Caph: A language for implementing stream-processing applications on FPGAs," in *Embedded Systems Design with FPGAs*, Springer, 2013, pp. 201–224.

[73] I. A. Technologies. "Impulse c," [Online]. Available: https://web.archive.org/web/20180901184426/http://www.impulsec.com/ (visited on Jan. 18, 2022).

PUBLICATIONS

PUBLICATION
I

High-level synthesis design flow for HEVC intra encoder on SoC-FPGA

P. Sjövall, J. Virtanen, J. Vanne, and T. D. Hämäläinen

In *Proceedings of Euromicro Conference on Digital System Design*, Funchal, Madeira, Portugal, Aug. 2015
DOI: 10.1109/DSD.2015.67

High-Level Synthesis Design Flow for HEVC Intra Encoder on SoC-FPGA

Panu Sjövall, Janne Virtanen, Jarno Vanne, Timo D. Hämäläinen
Department of Pervasive Computing
Tampere University of Technology, Tampere, Finland
panu.sjovall@tut.fi, janne.m.virtanen@tut.fi, jarno.vanne@tut.fi, timo.d.hamalainen@tut.fi

***Abstract*—This paper presents a High-Level Synthesis (HLS) flow for mapping a software HEVC encoder into Altera CycloneV SoC-FPGA. The starting point is a C implementation of an open-source Kvazaar HEVC intra encoder, which is minimally refined for SystemC design space exploration and automatic Catapult-C RTL generation. The final implementation involves Kvazaar encoder executed in Linux on dual-core ARM, and HW accelerated intra prediction on FPGA. Changing the SW/HW partitioning or modifying the implementation takes hours instead of weeks with Catapult-C HLS. In addition, the design is portable to other platforms without major manual re-writing. We obtained 9 fps full-HD intra prediction speed with a single accelerator on Altera Cyclone V SX on Terasic VEEK-MT-C5SoC board including video capture and HEVC video streaming via Ethernet. To the best of our knowledge, this is the first reported HLS assisted implementation of HEVC encoder on SoC-FPGA.**

Keywords—High-Level Synthesis, C to RTL, Catapult-C, HEVC, Kvazaar, intra coding, SoC-FPGA, SystemC, Cyclone V

I. INTRODUCTION

The latest video coding standard, *HEVC* (*High Efficiency Video Coding*) [4], has been developed for the transmission and storage of next-generation video. Compared with its predecessor standard AVC [8], HEVC is able to halve the bit rate for the same subjective quality, but its encoding complexity tends to be at least doubled in practical encoders. Furthermore, HEVC coding is at the same time both control and data dependent, which makes its modeling and implementation difficult. A trade-off would be a combination of state-based and dataflow-oriented *models of computation* (*MoC*). In this approach, the most complex parts of the encoder are tackled by hardware and control-intensive tasks are mapped to processor cores.

System-on-Chip FPGAs (*SoC-FPGAs*) integrate hard processor cores and programmable logic on a single chip, which makes them attractive to high-performance computing and application evaluation for HW/SW partitioning. The current design tools for SoC-FPGAs integrate traditionally separated SW development tools for processors and FPGA design tools. One challenge is the interface between these two domains. HW abstraction layers were written practically once for the traditional processor platforms, but this would be a weekly practice for SoC-FPGAs as the HW can easily be modified. Another challenge is the way the current tools operate. They follow the waterfall approach, and force starting over from the beginning for changes in the middle. This is very laborious for prototyping with HW/SW partitioning on SoC-FPGAs. For these reasons, we chose *High-Level Synthesis* (*HLS*) to speed up the design exploration, verification, and implementation [1][2].

The proposed HLS experiments have been carried out with an open-source Kvazaar HEVC encoder [6]. In this paper, the focus is on *all-intra* (*AI*) coding configuration [5] of Kvazaar. Our first platform is a low-range Altera Cyclone V chip on Terasic's VEEK development board [11].

Fig. 1 shows an overview of the prototype. The live video is captured with VEEK's camera, displayed on a touchscreen, encoded on Cyclone V, streamed to PC, and decoded on PC. Kvazaar HEVC intra encoder runs on a dual-core ARM@Linux with synthesized HW accelerators on FPGA. The overall design mission is very complex, and this paper reports how the implementation is accomplished. The new contributions of this paper are the following:

1. The first reported streaming HEVC intra encoder implementation on SoC-FPGA
2. HLS of HW accelerated HEVC intra encoding functions
3. HW/SW interface accepting pointers in SW code and DMA transfers in HW for HLS

The rest of this paper is organized as follows. Section 2 presents Kvazaar HEVC encoder and prior-art FPGA implementations of HEVC intra encoder functions. The used HLS design flow is introduced step by step in Section 3. Section 4 and Section 5 illustrate the implemented HW and SW architectures. Section 6 compares the design time and obtained performance between the proposed and contemporary approaches. Section 7 concludes the paper.

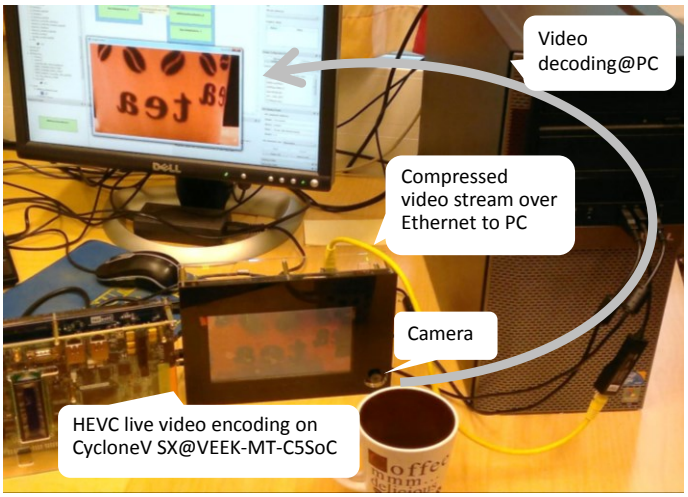


Fig. 1. Prototype platform for SoC-FPGA based HEVC encoder.

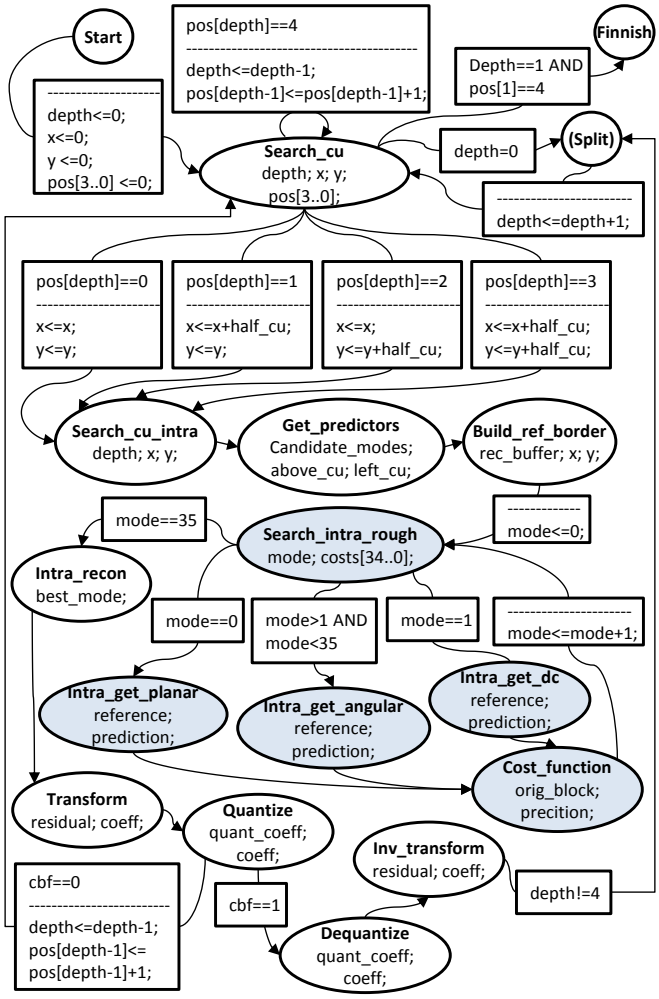


Fig. 2 Kvazaar HEVC intra encoder modelled as a state machine.

II. Related work

Currently, there exist three noteworthy practical open-source HEVC encoders: x265 [12], Kvazaar, and f265 [13], out of which only x265 and Kvazaar are currently under active development. Compared to x265 written in C++, Kvazaar is more hardware-friendly being implemented in C from scratch. Therefore, Kvazaar is used in our experiments.

A. Kvazaar HEVC intra encoder

Kvazaar intra encoder supports HEVC AI coding of 8-bit video with 4:2:0 chroma sampling. Currently, Kvazaar includes two presets: RD1 for high-speed encoding and RD2 for high-quality encoding [6]. The RD1 preset with parameters listed in Table 1 has been selected for the HLS flow. The involved encoder features are detailed in [6] and the source codes for Kvazaar can be found on its GitHub page [7]. Kvazaar version 0.24 is used in our experiments.

Fig. 2 depicts a state-machine model of Kvazaar HEVC intra encoder to illustrate its computational complexity. Here, the focus is on a rapid implementation of the HEVC encoder through a HLS flow, which later enables fast implementations of more optimized designs.

Table 1 Kvazaar HEVC coding parameters used in this work.

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

B. Existing HEVC implementations on FPGA

To the best of our knowledge, this is the first reported HLS assisted implementation of a complete HEVC encoder on SoC-FPGAs. However, there exits some non HLS implementations for core functions like intra-prediction [9] that supports all block sizes from 4x4 to 32x32 and achieves 17 frames per second on Altera Aria II. One of the existing FPGA implementations [14] is capable of real-time HEVC encoding of 8k video, but it has 17 boards, each having 3 FPGA chips. One board is capable of encoding full-HD at 60fps. Comparing our work to [14] would be difficult, due to lack of specifics on algorithm speeds, FPGA chips, and used area.

There is also a couple of HLS assisted HEVC decoder implementations on FPGA such as [15]. In addition Verisilicon has created a WebM (VP9) video decoder for Google. They report less than 6 months of the development time, compared to a one year estimate for a traditional RTL approach [10]. The project includes 69k lines of C++ source code, which is much smaller compared to 300k lines of RTL source code. We can confirm similar order of speed-up in our development work.

III. Design flow and tools

Fig. 3 depicts the main design steps and tools. The first phase, functional verification, is done on a PC using ready-made *make* for Linux GCC compiler. The next step is profiling for early performance estimation, in which we use Gprof, gprof2dot, and Graphwitz. Potential functions for HW acceleration are selected by examining the Gprof results.

According to our profiling with Cactus 1080p test sequence, the most time-consuming encoding functions are intra prediction, quantization, dst/dct, inverse dst/dct, and dequantization, whose respective shares of the encoding time are 67,74%, 8,54%, 4,69%, 3,78%, and 0,95%. Furthermore in Kvazaar intra prediction (*search_intra_rough*) the most time consuming function is intra_get angular with 35,75% of whole encoding process.

Search_intra_rough function calls *intra_get_pred* function to calculate the prediction for all 35 modes, then calculates the *Sum of Absolute Difference* (*SAD*) for all these modes, and finally returns the costs for all modes through a pointer passed to the function (Fig. 2). These functions are the most potential candidates for HW acceleration.

To implement intra predictions and SAD calculations on HW, we need to modify *search_intra_rough* to transfer data to and from the HW accelerator. This has to be done with minimal changes to the original Kvazaar code to maintain good portability.

A. High-level synthesis for estimation

Kvazaar is primarily developed for general-purpose processors, so inputting Kvazaar C code as such to Catapult-C [3] HLS would result in poor area and performance results. However, it is reasonable to utilize HLS of single functions for early area estimates, and use that information to partition Kvazaar for HW/SW implementations.

Catapult-C offers good support for interfacing SW and HW. To prepare a single C function for HLS, the input arguments and return values are replaced by explicit communication channels. On the other hand, pointer data can be retained in the SW code and the actual memory region can be accessed by a HW module. Based on Gprof profiling results and HLS trials of Kvazaar functions, SystemC models are created for design space exploration and HW/SW partitioning.

B. Untimed SystemC modeling

The next design step is separation of communication and computation in Kvazaar. The work is started with untimed SystemC model in which the intra prediction functions are

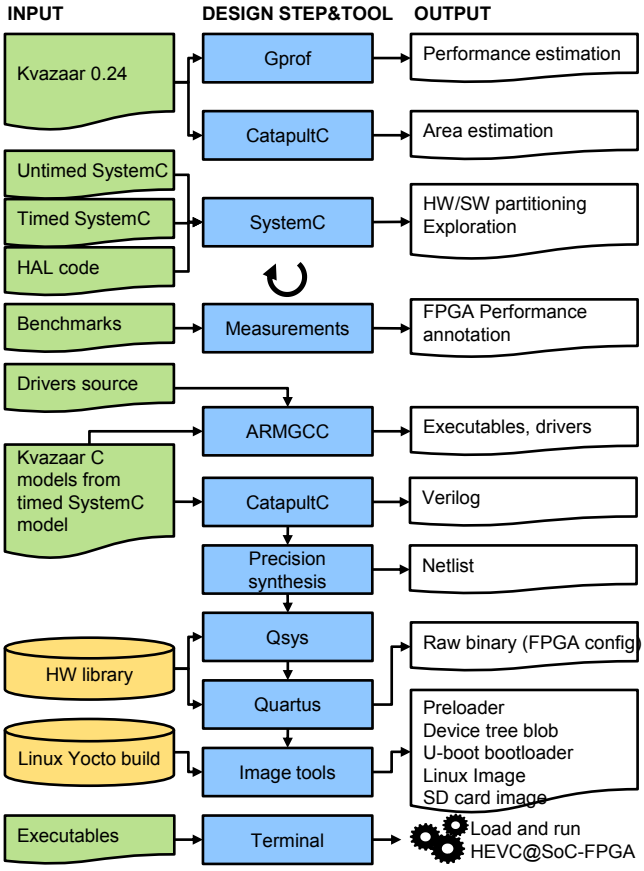


Fig. 3 HLS-based design flow for HEVC on SoC-FPGA

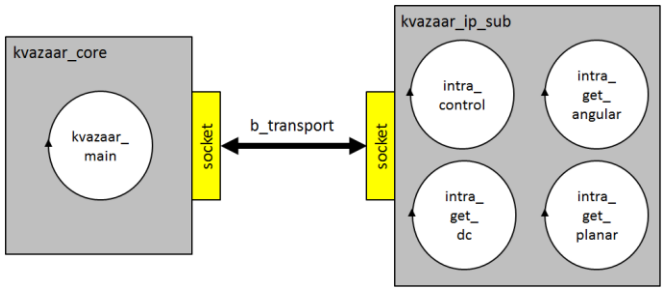


Fig. 4 Untimed SystemC model of Kvazaar.

divided into two main modules (*kvazaar_core* and *kvazaar_ip_sub*) as depicted in Fig. 4. We separated only the neeededdata used by the HW accelerator from the PC friendly data structure to minimize the data transfers.

At this point, a HW abstraction layer is prepared between Kvazaar and the SystemC modules for later implementation models. The Kvazaar main function is called in a SystemC thread, and interaction with other SystemC modules occurs through class member functions called mmap, *ioctl, read*, and *write*. By using these functions, moving from the SystemC model to C with Linux drivers is possible without remarkable modifications to the Kvazaar code.

The HW accelerator (*kvazaar_ip_sub*) consists of four threads. The control thread waits until data is valid and then starts the prediction threads *angular, dc*, and *planar* which have the same implementations as in Kvazaar. Each prediction thread- calculates the prediction- and the corresponding SAD for it after which the control thread sorts the results and returns the best one to the Kvazaar module. Data is passed between Kvazaar module and the HW accelerator using TLM-2.0 transaction level modeling.

C. Timed SystemC modeling

The untimed model proved that the functionality did not change after separation of communication and introducing the *hardware abstraction layer* (*HAL*). The next step is to create a timed SystemC model to explore parallelization of the HW model. This is carried out by re-fining kvazaar_ip_sub, adding more modules, and by leaving the kvazaar_core intact.

Actual functions calculating intra prediction pixels in Kvazaar are divided into *intra_get_dc, intra_get_planar* and *intra_get_angular*. In the untimed model, we created different processes for those, but still executed all 33 angular modes sequentially as in the original Kvazaar. However, none of the 35 intra prediction modes have data dependencies and can run in parallel. We also further divided the *intra_get_angular* to three separate functions *get_ang_pos, get_ang_neg*, and *get_ang_zero* to remove overlapping computation. This decision was based on Catapult-C HLS trials and area results.

The complete timed SystemC model is depicted in Fig. 5. It includes reference pixel filtering in *ip_ctrl*, all 35 modes predicted in parallel with *get* blocks, and SAD for all modes in *sad_parallel*. The kvazaar_ip_sub is now different compared to the untimed model, also the naming for the IP

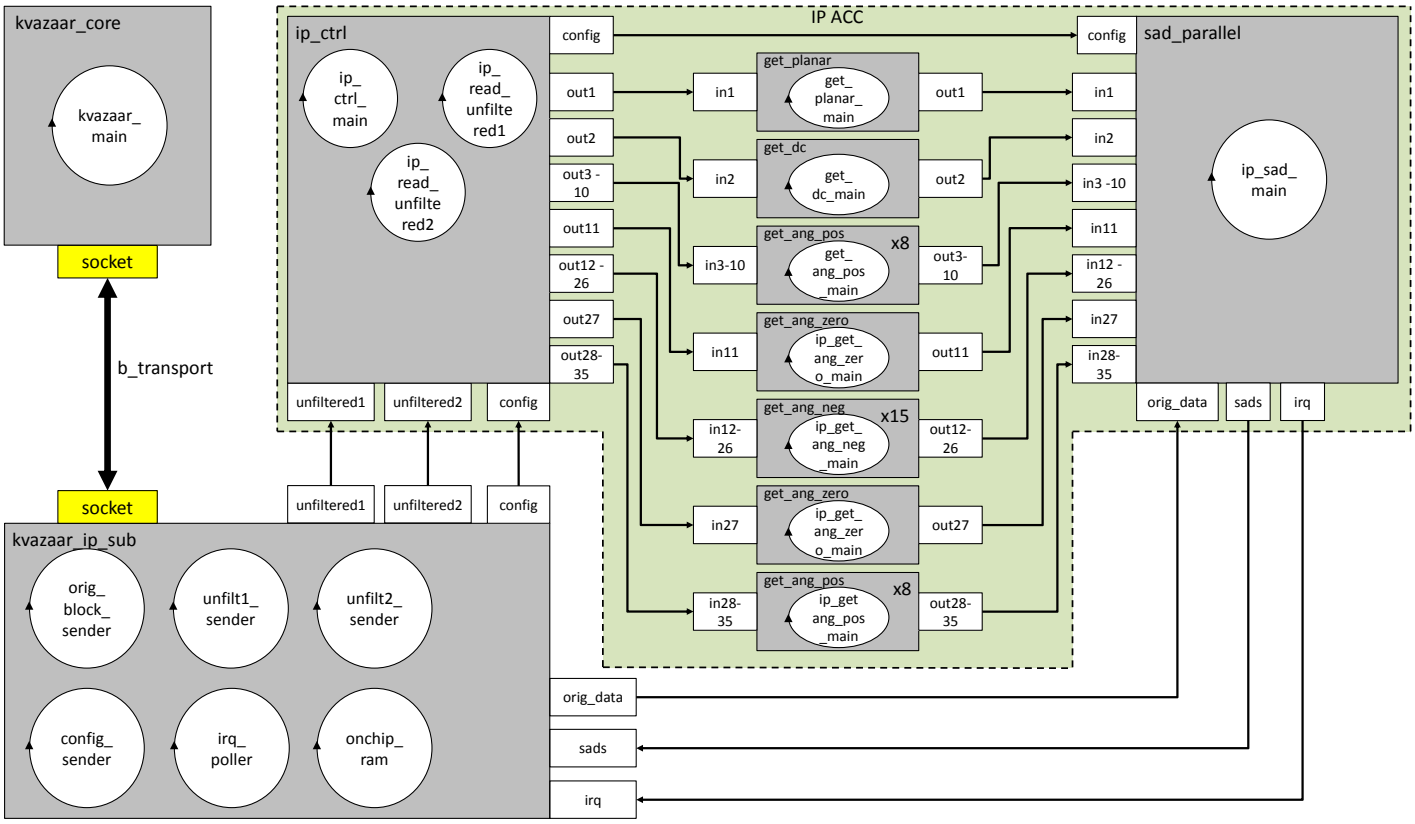


Fig. 5 Timed SystemC model of Kvazaar.

ACC blocks has been changed to differ from original Kvazaar functions. In the untimed model, kvazaar_ip_sub did the intra prediction, but it is now substituted by a block that has the same interface to the kvazaar_core as before but is used as an abstraction between the IP ACC blocks and TLM2.0 transactions. SystemC test benches are created for all intra prediction SystemC modules and a system test for the full model.

D. Platform modeling and performance estimation

The next step is the FPGA platform modeling with measured benchmark values. This is used to annotate the timed SystemC model for more accurate performance estimation results. Benchmarking programs were written for Cyclone V to measure memory bandwidths and latency between HPS and the FPGA. In addition, we measured the performance of Kvazaar on ARM with Linux to get realistic annotations to the timed SystemC model. We compiled Kvazaar using the same makefile we used to compile it for Linux PC, only changing the compiler to ARMGCC.

Time consumption estimates of different functions used SystemC function wait(), which suspends the thread or clocked thread process instance from which it is called. This turned out to increase the simulation time 15x in some cases. We solved this by using nanosecond counters for the functions, and using wait() only in the timed HW model.

The timed SystemC model was annotated with the measured values from Kvazaar executed on ARM without any HW acceleration. We used the frame rate got from ARM and the distribution of time got from Gprof. In the non-HW accelerated SystemC model, the time usage of the *intra_rough_search* is adjusted to match to the measured.

In the HW accelerated version, the time used for *intra_rough_search* is the clock accurate simulation time of the SystemC hardware model. Other functions run on ARM stay the same, thus giving the frame rate improvement. The simulation model could be easily modified for different FPGA boards by changing the benchmarked or estimated values.

E. Implementation

Next, we implemented the HW for our encoder. Generating the RTL for the HW accelerator only involves creating the top-level function required by Catapult-C and modifying the SystemC code to match Catapult-C code style. This phase could be omitted if there were a license for SystemC synthesis in Catapult-C. Since the created Kvazaar model in SystemC is close to the C in Catapult-C- it was very easy to change between the two without changes in functionality. Changes included differences in defining ports, writing to ports or Catapult-C channels, wait() calls removed and bit accurate types *sc_int* changed to *ac_int*. All of that could be automated.

Catapult-C generates Verilog code, which was synthesized using Precision that generated netlist. The rest of the design flow follows a conventional SoC-FPGA

development project. Qsys was used to include *hard processor system* (*HPS*) and components connected to the AXI3 bus, and Quartus to assemble HLS generated parts and library components. A ready-made Linux image was utilized without re-building, because the generated HW accelerator uses a standard HPS to FPGA interface.

IV. HW Architecture

The final HW architecture on SoC-FPGA is depicted in Fig. 6. It consists of the fixed HPS with ARM cores, ready-made Terasic camera subsystem with our own modifications, the HLS generated HW accelerator, and our custom interface blocks for DMA and configuration.

A. HPS/FPGA interface

The HPS has two ways to interface with the FPGA. Shared AMBA3 (AXI) interconnection is suitable for small data amounts like signaling. The second way is to use multi-port DDR SDRAM controller and off-chip DDR3 memory.

An important issue is to use non-cached memory in ARM and Linux to ensure data coherency between HW/SW. This way, we can use data through the original memory pointers in Kvazaar source code by mapping the memory to a physical location of our choosing. Transferring data to the HW accelerator only requires that the DMA is configured to read the right amount of data from DDR3. DMA is synthesized on FPGA and connected to the memory controller through the AXI bus.

HPS has fixed base addresses for generic interfaces for FPGA. Each IP-block using them gets address offset on top of the base. The resulting physical FPGA addresses are above the reserved Linux memory space in DDR3, which ensures Linux is not interfering with data. However, physical address is mapped to a Linux virtual address by a driver calling mem_map. This way, the Linux Device Tree is not needed to re-create every time the FPGA logic is modified.

V. SW Architecture

CycloneV boots by first starting the HPS on boot ROM. It reads a preloader from the SD card that configures the FPGA part and starts the U-boot bootloader, which in turn starts Linux OS. The Linux image is built using Yocto project, in which design specific Linux Device Tree Source is obtained from the handoff files from Quartus. In our case, the interface between HPS and FPGA remains the same (FPGA bus base addresses are intact) and there is no need to rebuild Linux after redesigning the HW accelerator part (Fig. 6).

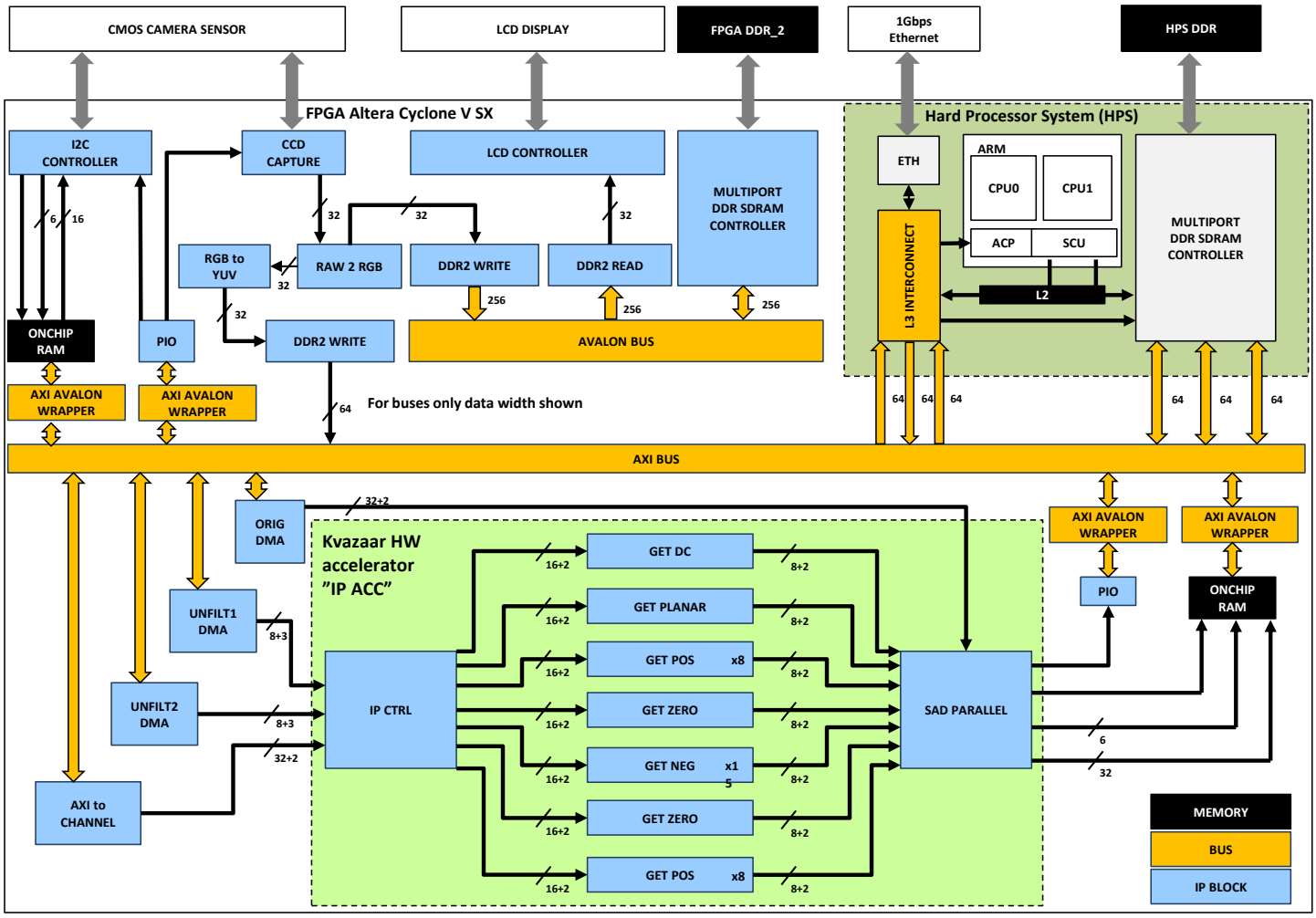


Fig. 6. HEVC intra encoder architecture on CycloneV SoC-FPGA.

A. HW abstraction

We use run time loadable kernel driver modules for the HW abstraction. Two drivers are needed for FPGA: The camera and the Kvazaar HW accelerator (Fig. 6). Both consist mostly of functions mapped to system calls (SCs), and have an initialization and a clean up call for loading and removing the kernel module. Both do memory mapping and access HW via such memory regions. Both include about 1k LOC. For debugging, Altera Signal Tap was used to see how the interaction functioned from the HW side.

The kernel modules are used in SW via file descriptors given as a parameter to system calls, such as open() and ioctl(). The other parameters are predefined when designing and implementing the driver.

The drivers implement a function for each supported system call. Most important is mmap(), which maps a region of physical memory for the application utilizing the driver. This allows the application to access data directly in the DDR memory. This saves CPU time, as the alternative would be using a driver to copy data to kernel space, and then copy it again to the user space. Within the kernel module, remap_pfn_range is called to execute the mapping. ACU was not used, because we did not want to affect the L2 cache, and to get parallel access to the memory controller.

The memory mapping is applied only to large transfers such as pixel data, while ioctl is used for configuration and control signaling. Both may require some HW specific handling, such as acknowledging that data is read. This is done by an ioctl call before or after accessing a mapped region. The regions used for signaling are mapped for the kernel module by calling ioremap_nocache.

For both mapping types, the physical HW specific addresses are defined in the kernel module. This way the application remains fully portable. An important issue is to mark the mapped regions non-cached. The transfers between HW/SW fail if the data is not truly in DDR at all times.

B. HW/SW interaction

Fig. 7 illustrates the message sequence chart of the complete system from camera to ethernet. Kvazaar supports multi-threading, and there can be multiple instances of the HW accelerator. For clarity, the threads are omitted and only one vertical line is displayed for each instance. On the top of the chart, the blue boxes are HW components and green are SW.

The major components are Camera User, Kvazaar, IP Acc and Stream server. The rest are mostly used for transfers between other components. DDR is used for large transfers and on-chip memory and parallel io (PIO) are used for smaller signals and configuration data. DMA modules are used to control the transfers using DDR. Software accesses the HW via drivers.

The execution sequence begins with initialization. System calls are used for user space software to access the kernel module drivers. Furthermore, Camera User, Kvazaar, and the Stream server are configurable with command line parameters.

The camera feeds frames simultaneously for both LCD controller in RGB format and YUV format for the rest. Kvazaar expects 4:2:0 YUV, which is not HW accelerated but the Camera User must convert each frame before passing it to stdout. This is currently a known bottleneck with larger resolutions. The camera operates in a continuous mode, feeding frames as fast as it can. Also an interrupt driven mode is supported (not shown).

Kvazaar reads a frame from stdin. For each LCU (Large Coding Unit), the HW accelerator is used to compute its SAD values. Each LCU is configured by sending the original block to the accelerator. The original block is first written to the HPS DDR and the Original block is signaled the address and length of the block in the DDR. After that, DMA proceeds to read the original block and pass it to the accelerator.

The LCU contains multiple CUs. Each contains two unfiltered blocks, which are passed to the accelerator. However, each of these is preceded by additional configuration data passed directly to the accelerator, saved for a bridge between it and the AXI bus channel. For each LCU, a SAD value is computed in the accelerator. When done, the values are passed via on-chip memory. The driver polls the HW for a signal notifying that the values are available. The system also supports interrupts. After SADs are passed to Kvazaar, it proceeds with encoding.

Output of Kvazaar in this system is passed via stdio to a streaming server application, which sends it over UDP to a stream client. A standard PC and Classic Media Player is used to receive and decode the video (not shown in chart).

VI. Evaluation

Altogether, Kvazaar 0.24 intra encoder has 19k lines of code, 300 functions, and 33 modules. Table 2 shows the lines

Table 2 Lines of code in SystemC and generated Verilog RTL code.

Module	SystemC	Verilog
kvazaar_core (untimed/timed)	289	-
kvazaar_ip_sub (untimed)	357	-
kvazaar_ip_sub(timed)	369	-
ip_ctrl	553	9423
get_planar	170	1824
get_dc	186	1658
get_ang_pos	150	1422
get_ang_zero	149	1301
get_ang_neg	178	2060
sad_parallel	404	10976

Table 3 Cycle counts of intra prediction on FPGA (version 1).

Search Mode	Execution count	HW cycles/mode	Total cycles
4	74 074	159	11 777 766
8	27 568	243	6 699 024
16	7876	503	3 961 628
32	1980	1403	2 777 940
Total	**111 498**		**25 216 358**

Table 4 Cycle counts of intra prediction on FPGA (version 2).

Search Mode	Execution count	HW cycles/mode	Total cycles
4	74 074	110	8 148 140
8	27 568	159	4 383 312
16	7876	300	2 362 800
32	1980	776	1 536 480
Total	**111 498**		**16 430 732**

of handwritten SystemC code and the lines of RTL Verilog code generated by Catapult-C for the implemented modules. The rest of Kvazaar functions stay unchanged so they are excluded.

The untimed and timed SystemC models use the same *kvazaar_core* module, but the *kvazaar_ip_sub* modules have different timing abstractions. The timed modules include Kvazaar source code algorithms as such, only major difference being the cycle accurate interfaces between the blocks.

Changes in Kvazaar C source code were transferred to the HW implementation with only minor effort. For example, reorganizing the DC filtering in Kvazaar from reconstruction to be also part of the intra prediction HW took about 4 hours. To create a single DCT or a quantization

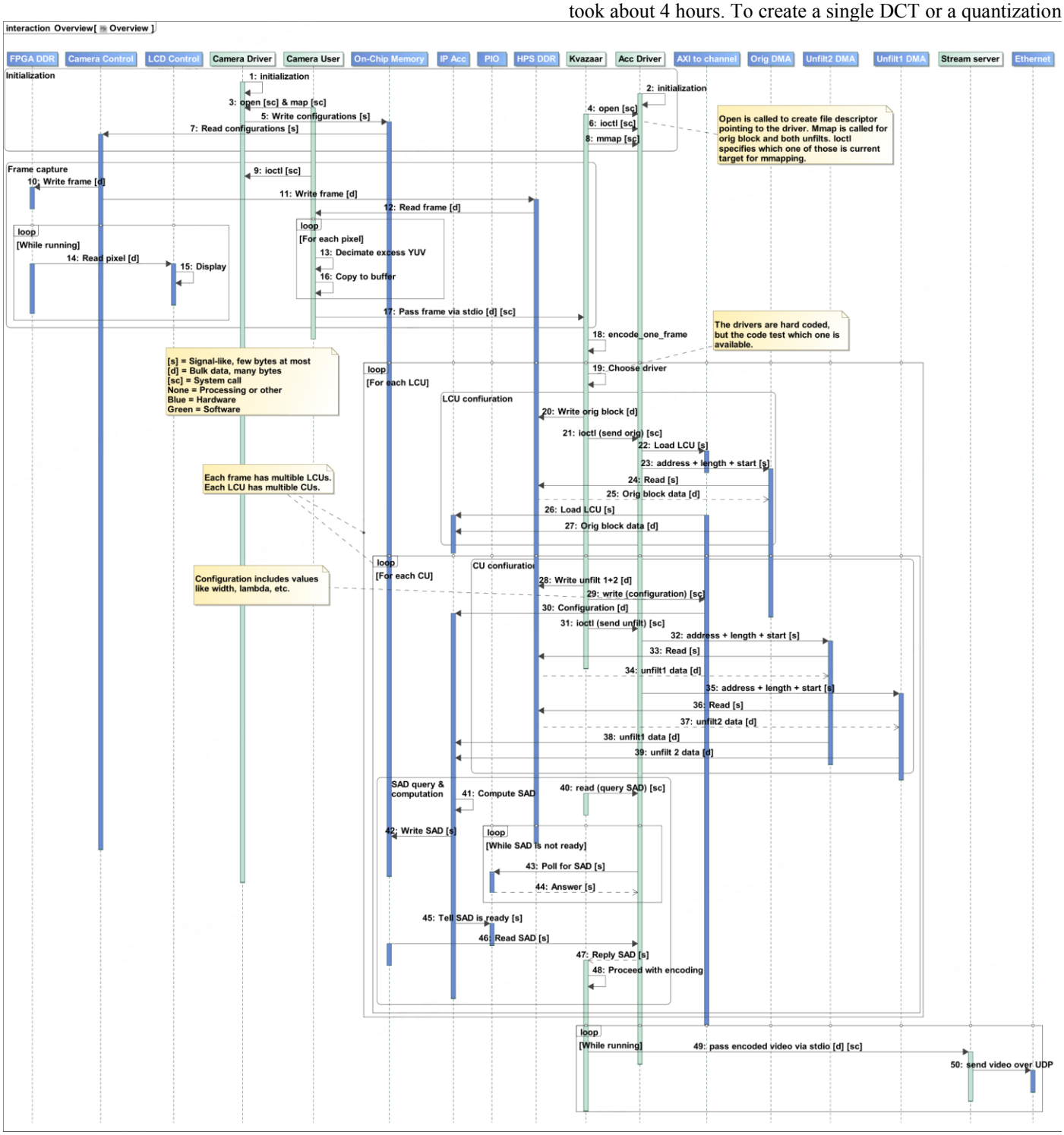


Fig. 7. HEVC intra encoder HW/SW message sequence chart.

accelerator is estimated to take less than a day for the first implementation and another day for speed and area optimizations.

A. Intra prediction performance

Tables 3 and 4 report the cycle counts for two different versions of the implemented intra prediction on FPGA. The test sequence is Cactus 1920×1080.

In version 1, one pixel is predicted at a time. The HW accelerator calculates intra predictions for all prediction modes and block sizes, and SAD for every prediction mode. The encoding speed of this design is 5.9 fps and the FPGA area occupied is 8345 ALMs.

In version 2, two pixels are predicted simultaneously. This changes the 8+2 outputs of GET blocks in Fig. 6 to 16+2 outputs, increases the area of the GET blocks and SAD PARALLEL block.In this case, SAD is calculated for all modes two pixels at a time. The encoding speed increased to 9.1 fps and the area to 10815 ALMs, which is 25.8% of the total. CycloneV SX can accommodate two HW accelerators. The time to make such a change in the design took only about ten hours from C to verified execution on board.

B. Discussion

For comparison, the design presented in [9], uses 31.179 ALUTs (15.589 ALMs) and achieves 17.52 fps. Contrary to our design, the presented result does not include SAD computation. The similar performance could be achieved with our design by, e.g., calculating four pixels at a time, decreasing time sending configurations, and sending more reference pixels from ip_ctrl to GET blocks at a time (Fig. 7). More HW acceleration can also be included in the camera side, e.g., by formatting raw frames from 4:4:4 to 4:2:0 that is inefficient with SW. Furthermore, passing frames via stdio from Camera User to Kvazaar involves extra copying in SW. As the HPS side limits the performance, more functions could be HW accelerated in the remaining available area.

Based on the annotated SystemC simulations, we can estimate the requirements for a Full-HD 30 fps performance. According to timed SystemC model, one alternative is to run the ARM at 3GHz and the HW accelerator at 500MHz with the design version 2.

Our benchmark measurements for SystemC simulation annotations revealed some bottlenecks on the platform. For example, having two user space Linux threads reading and writing via L3 interconnect (without cache, forced DDR3) resulted in 28.81 MiB/s speed. For comparison, the read-write speed between FPGA on-chip memories is 200 MiB/s with the FPGA side AXI bus running at 50MHz. The external DDR3 memory speed is dependent on the board design and thus out of scope of our research.

VII. Conclusions

We presented the design of a live streaming HEVC intra encoder on SoC-FPGA using High-Level Synthesis for HW accelerated functions. The design time and flexibility is significantly improved over traditional VHDL-based implementations. As a rough estimate, the HLS reduces the development time from weeks to days, especially for changes and modifications in the original source code. Since Kvazaar is under intensive development, it is essential to have a rapid design flow to carry out the algorithmic changes to HW implementations. In addition, fast porting of the encoder functionality onto different embedded platforms or for different HW/SW partitions is one of our main goals.

Compared to reported HEVC implementations, our HLS-based design flow produces comparable performance and area for the CycloneV device which has enough capacity for two intra search HW accelerators Our future work includes an implementation on Altera Arria SoC-FPGAs that can accommodate more encoder functionality on HW.

References

[1] L. Daoud, D. Zydek, and H. Selvaraj, “A survey of high level synthesis languages, tools, and compilers for reconfigurable high performance computing,” in *Advances in Systems Science*, Springer, 2014, pp. 483-492.

[2] W. Meeus, K. Van Beeck, T. Goedemé, J. Meel, and D. Stroobandt, “An Overview of Todays High-Level Synthesis Tools,” *Design Automation for Embedded Systems*, pp. 1-21, Aug. 2012.

[3] *Calypto's Catapult 8 HLS: C-Based HWHardware Design Matures*, Berkeley Design Technology, Inc. [Online] Available: http://www.bdti.com/InsideDSP/2014/11/18/Calypto

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

[5] 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, pp. 1792-1801, Dec. 2012.

[6] M. Viitanen, A. Koivula, A. Lemmetti, J. Vanne, and T. D. Hämäläinen, “Kvazaar HEVC encoder for efficient intra coding,” *in Proc. IEEE Int. Symp. Circuits Syst.*, Lisbon, Portugal, May 2015.

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

[8] *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.

[9] A. Abramowski and G. Pastuszak, “A double-path intra prediction architecture for the hardware H.265/HEVC encoder,” *in Proc. IEEE Symp. Des. Diagnost. Electron. Circuits Syst.*, Warsaw, Poland, Apr. 2014, pp. 27 - 32.

[10] Calypto's Catapult 8 HLS: C-Based Hardware Design Matures [Online] Available: http://www.bdti.com/InsideDSP/2014/11/18/Calypto

[11] VEEK-MT-C5SoC [Online] Available: http://www.terasic.com.tw/cgi-bin/page/archive.pl?Language=English&CategoryNo=&No=828

[12] *x265* [Online]. Available: http://x265.org/

[13] f265 [Online]. Available: http://f265.org/

[14] Miyazawa, K.; Sakate, H.; Sekiguchi, S.-I.; Motoyama, N.; Sugito, Y.; Iguchi, K.; Ichigaya, A.; Sakaida, S.-I., "Real-time hardware implementation of HEVC video encoder for 1080p HD video," *Picture Coding Symposium (PCS), 2013* , vol., no., pp.225,228, 8-11 Dec. 2013,

[15] Abid, M.; Jerbi, K.; Raulet, M.; Deforges, O.; Abid, M., "System level synthesis of dataflow programs: HEVC decoder case study," *Electronic System Level Synthesis Conference (ESLsyn), 2013* , vol., no., pp.1,6, May 31 2013-June 1 2013

PUBLICATION
II

High-level synthesis implementation of HEVC 2-D DCT/DST on FPGA

P. Sjövall, V. Viitamäki, J. Vanne, and T. D. Hämäläinen

In *Proceedings of IEEE International Conference on Acoustics, Speech and Signal Processing*,
New Orleans, Louisiana, USA, Mar. 2017
DOI: 10.1109/ICASSP.2017.7952416

HIGH-LEVEL SYNTHESIS IMPLEMENTATION OF HEVC 2-D DCT/DST ON FPGA

Panu Sjövall, Vili Viitamäki, Jarno Vanne, Timo D. Hämäläinen

Laboratory of Pervasive Computing
Tampere University of Technology, Finland
{panu.sjovall, vili.viitamaki, jarno.vanne, timo.d.hamalainen}@tut.fi

ABSTRACT

This paper presents the first known high-level synthesis (HLS) implementation of integer discrete cosine transform (DCT) and discrete sine transform (DST) for High Efficiency Video Coding (HEVC). The proposed approach implements these 2-D transforms by two successive 1-D transforms using a well-known row-column and Even-Odd decomposition techniques. Altogether, the proposed architecture is composed of a 4-point DCT/DST unit for the smallest transform blocks (TBs), an 8/16/32-point DCT unit for the other TBs, and a transpose memory for intermediate results. On Arria II FPGA, the low-cost variant of the proposed architecture is able to support encoding of 1080p format at 60 fps and at the cost of 10.0 kALUTs and 216 DSP blocks. The respective figures for the proposed high-speed variant are 2160p at 30 fps with 13.9 kALUTs and 344 DSP blocks. These cost-performance characteristics outperform respective non-HLS approaches on FPGA.

Index Terms— High Efficiency Video Coding (HEVC), Discrete cosine transform (DCT), Discrete sine transform (DST), High-level synthesis (HLS), Catapult-C, Field-programmable gate array (FPGA)

1. INTRODUCTION

The latest video coding standard, *High Efficiency Video Coding* (*HEVC*) [1], has been developed to meet the transmission and storage needs of modern video applications. Compared with its predecessor standard AVC [2], HEVC is able to halve the bit rate for the same subjective quality, but its encoding complexity tends to be at least doubled in practical encoders.

HEVC adopts the conventional hybrid video coding scheme (inter/intra prediction, transform coding, and entropy coding) [3] from the prior MPEG/ITU-T video coding standards. As a new feature, the coding structure of HEVC has been extended from a traditional macroblock concept to an analogous block partitioning scheme that supports *coding tree units* (*CTUs*) of up to 64 × 64 pixels [4].

This paper focuses on HEVC transform coding for which the sizes of *transform blocks* (*TBs*) and associated core transform matrices can be defined as $N \times N$, where $N \in \{4, 8, 16, 32\}$. Extending the sizes of transform matrices from that of AVC to $N > 8$ improves coding gain by around 5-7% but it also introduces the majority of complexity overhead in HEVC transform coding [5].

HEVC specifies *two-dimensional* (*2-D*) integer *discrete sine transform* (*DST*) for intra coded luminance TBs of size 4 × 4 pixels [6] and 2-D integer *discrete cosine transform* (*DCT*) for all other TBs [7]. Both of these 2-D transforms are separable so they can be computed by applying two *N*-point 1-D transforms first row-wise and then column-wise [5]. This indirect approach is called a row-column decomposition technique and it is typically utilized by software [8]-[9] and hardware implementations [10]-[16] of HEVC DCT/DST.

This work focuses on HEVC DCT/DST implementations on FPGA. Contrary to previous works [12]-[16], our proposal does not use traditional *hardware* (*HW*) *description languages* (*HDLs*), but *High-Level Synthesis* (*HLS*) [17] which is an emerging approach for raising the abstraction level in HW description. HLS is a way of using well-known programming languages such as C and C++ to describe the designs at behavioral level and automatically generating the HDL from it. This way, the code is more readable, design and verification times are shorter, and the design reusability is far better than with handwritten HDL equivalents.

To the best of our knowledge, this is the first paper to describe an HLS implementation for HEVC DCT/DST. The proposed designs include low-cost and high-speed variants of the 8/16/32-point DCT unit for $N \in \{8, 16, 32\}$ and a separate 4-point DCT/DST unit for $N = 4$. They are all implemented on Arria II FPGA using Catapult C [18] HLS tool.

The rest of this paper is organized as follows. Section 2 describes the hardware-oriented DCT/DST algorithm implemented in this work. Section 3 proposes our HLS implementations for low-cost and high-speed DCT/DST computation. In Section 4, the proposed HLS implementations are compared with handcrafted prior-art. Section 5 concludes the paper.

2. 2-D INTEGER DCT/DST ALGORITHMS IN HEVC

In this work, the C implementations of DCT and DST algorithms are obtained from the open source Kvazaar HEVC encoder [8]. Basically, Kvazaar implements the same DCT/DST functionality than *HEVC reference encoder* (*HM*) [9] but the hardware-oriented C source code of Kvazaar provides a better starting point for HLS.

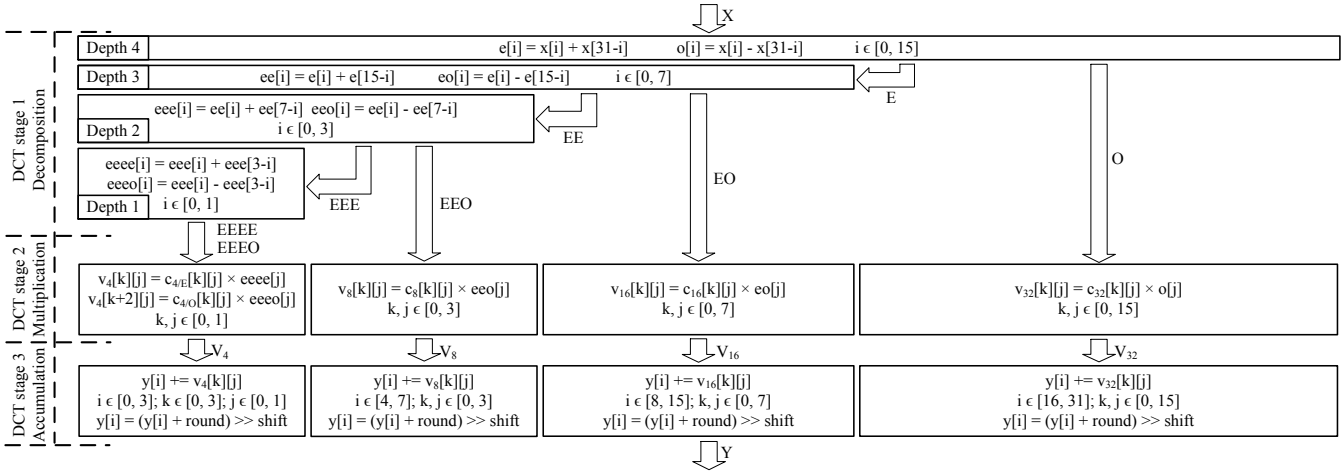


Figure 1. Even-Odd decomposition algorithm (N = 32).

2.1 Even-Odd decomposition algorithm

In HEVC encoder, DCT and DST are used to convert spatial-domain residual blocks into transform-domain coefficient matrices. A well-known row-column algorithm [5] executes these 2-D transforms with separable 1-D transforms in two consecutive stages. An N-point transform is first applied 1) to each row of a residual block of size $N \times N$ to generate an intermediate matrix of size $N \times N$; and then 2) to each column of the intermediate matrix to generate a final transform coefficient matrix of size $N \times N$.

The number of arithmetic operations can be further reduced by implementing these 1-D transforms with Even-Odd decomposition algorithm, a.k.a., Partial Butterfly algorithm [5]. It decomposes an input and core transform matrices of size $N \times N$ into two matrices of size $N/2 \times N/2$ according to even and odd rows/columns, respectively. The core transform matrices for each N ($\mathbf{C}_N$) are specified in [7]. Now, an N-point transform can be computed for even and odd cases separately with two N/2-point transforms.

For a residual vector $\mathbf{X} = [x(0), x(1), \ldots, x(N\text{-}1)]$, the even and odd vectors, $\mathbf{E} = [e(0), e(1), \ldots, e(N/2\text{-}1)]$ and $\mathbf{O} = [o(0), o(1), \ldots, o(N/2\text{-}1)]$, can be computed as

$$e(i) = x(i) + x(N - 1 - i) \quad (1)$$
$$o(i) = x(i) - x(N - 1 - i) \quad (2)$$

where i = 0, 1, …, N/2 - 1. The output vector $\mathbf{Y} = [y(0), y(1), \ldots, y(N\text{-}1)]$ of 1-D transform coefficients could be directly obtained by multiplying the vectors **E** and **O** by the associated transform matrices at this stage. However, the arithmetic operations can be further reduced by applying decomposition recursively. In this approach, the largest transform matrix also embeds the smaller transform matrices.

Fig. 1 depicts the phases of Even-Odd decomposition for N = 32. First, the vectors **E** and **O** of size 16 are computed according to (1) and (2). The latter is an input to $\mathbf{C_{32}} \times \mathbf{O}$ multiplication and the former is recursively decomposed into smaller even and odd vectors as in (1) and (2), i.e., the vector **E** is divided into **EE** and **EO** vectors of size 8. The vector **EO** is multiplied by $\mathbf{C_{16}}$ whereas **EE** is decomposed into **EEE** and **EEO** vectors of size 4, and **EEE** to **EEEE** and **EEEO** vectors of size 2. **EEO** is multiplied by $\mathbf{C_8}$, **EEEO** by $\mathbf{C_{4/O}}$, and **EEEE** by $\mathbf{C_{4/E}}$. The corresponding structure can be used for all N by starting at depth $(\log_2 N) - 1$.

2.2 Proposed hardware-oriented algorithm optimization

In the case of 8-bit video, the residual vector **X** contains 9-bit signed integers for which the original Even-Odd decomposition algorithm produces $9 + (\log_2 N + 6)$ –bit signed results [5] without any truncations. Our motivation is to optimize the algorithm for 18 × 18 multipliers on Arria II FPGA due to which 19-bit (N = 16) and 20-bit (N = 32) odd and even values are saturated to 18-bit signed values.

The impact of this modification was tested with HM 16.12 using test sequences from HEVC common test conditions (classes A-F) [19] and the average BD-rate overhead is 0.002%. This negligible loss is preferred to using 20 × 20 – bit multipliers that would increment the number of needed DSP blocks fourfold.

3. PROPOSED DCT/DST ARCHITECTURE

The proposed DCT/DST architecture is composed of 1) an 8/16/32-point DCT unit for TBs of size 8 × 8, 16 × 16, and 32 × 32; 2) a separate 4-point DCT/DST unit for TBs of size 4 × 4; and 3) a transpose memory for intermediate results.

3.1 8/16/32-point DCT unit

Fig. 2 shows the block diagram of the 8/16/32-point DCT unit. It contains a control block ($\text{Ctrl}_{8/16/32}$), 3-stage pipeline for DCT computation, and a transpose memory.

A 288-bit input to the $\text{Ctrl}_{8/16/32}$ block is for up to 32 9-bit signed residuals. The $\text{Ctrl}_{8/16/32}$ block sign extends each 9-bit residual to 16 bits and passes them through the 3-stage DCT computation via a 512-bit connection. The mapping of the

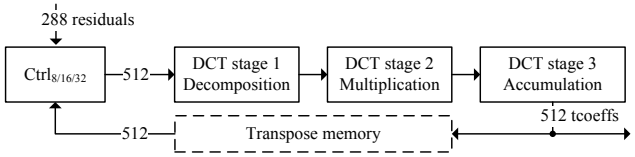


Figure 2. Block diagram of the 8/16/32-point DCT unit.

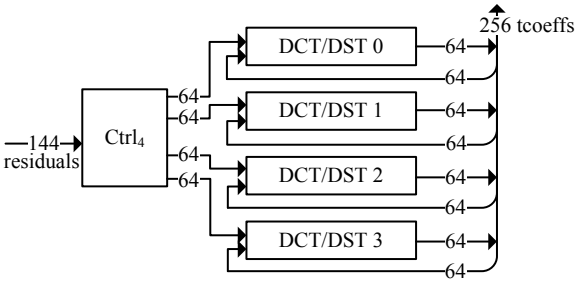


Figure 3. Block diagram of the 4-point DCT/DST unit.

Even-Odd decomposition algorithm to three DCT stages is illustrated in Fig. 1.

The DCT stage 1 performs the recursive Even-Odd decomposition for the 16-bit residuals and computes all even and odd vectors (**E/O, EE/EO, EEE/EEO,** and **EEEE/EEEO**). It is implemented in C code as a recursive template function which is synthesized by Catapult-C to an adder tree

The DCT stage 2 is for multiplication between transform matrices and odd vectors ($\mathbf{C_{32}} \times \mathbf{O}$, $\mathbf{C_{16}} \times \mathbf{EO}$, $\mathbf{C_8} \times \mathbf{EEO}$, $\mathbf{C_{4/O}} \times \mathbf{EEEO}$, and $\mathbf{C_{4/E}} \times \mathbf{EEEE}$). On FPGA, this functionality is mapped to multipliers of DSP blocks to save logic cells. Catapult-C facilitates instantiation of DSP blocks in C code by providing a library for DSP blocks as C++ templates for different FPGA architectures.

The DCT stage 3 finalizes the 1-D transform by accumulating the individual products of multiplication and scales the coefficients to 16 bits.

The 8/16/32-point DCT unit performs the 2-D DCT in two successive passes and the intermediate data is stored in the transpose memory. The latency for both passes is 3 cycles because of the DCT pipeline. Finally, the 2-D 16-bit *transform coefficients* (*tcoeffs*) are sent via 512-bit output.

This work proposes two alternate 8/16/32-point DCT units with different parallelization strategies:

1) A low-cost unit processes N residuals (one row/column of a TB) in parallel. In this unit, the residuals enter the DCT stage 1 at depth $(\log_2 N) - 1$. In addition, the DCT stages 2 and 3 operate at double clock frequency to be able to compute the largest TB in two phases with the reduced number of DSP blocks. This approach halves the width of the largest multiplier array, without increasing latency.
2) A high-speed unit processes 32 residuals ($32/N$ rows/columns of a TB) in parallel so that a constant data rate with full hardware utilization is achieved. In this unit, the residuals enter the DCT stage 1 at depth 4. In addition, all DCT stages operate at the same frequency and the DCT stage 2 contains a full-width multiplier array.

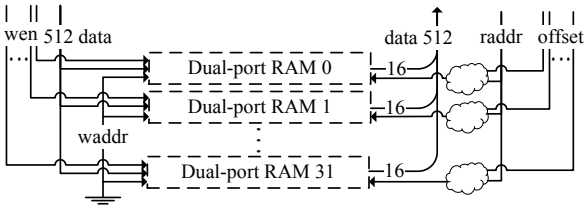


Figure 4. Block diagram of the transpose memory.

3.2 4-point DCT/DST unit

Fig. 3 depicts a 4-point DCT/DST unit that operates in parallel with the 8/16/32-point DCT unit. A 144-bit input to the Ctrl_4 block accepts a single 4×4 residual block at a time. The 9-bit residuals are sign extended to 16-bits and passed row-wise to the respective four DCT/DST blocks. The intermediate matrix is ready in one cycle after which it is sent back to the same DCT/DST blocks by picking the intermediate values from the registers in a transposed order. After these two passes, the unit outputs 16 16-bit coeffs.

A separate 4-point DCT/DST unit increases the occupied resources on FPGA. However, this overhead is compensated by better load balancing since the share of 4×4 TBs is relatively high compared to the other TBs.

3.3 Transpose memory

Fig. 4 depicts the structure of the transpose memory used in the 8/16/32-point DCT unit. On FPGA, it is made of 32 dual-port on-chip memory modules without registers. Each memory module has a 512-bit write (N coefficients) and a 16-bit (1 coefficient) read port. The structure supports block transpose for $N \in \{8, 16, 32\}$.

The memory utilization of the low-cost 8/16/32-point DCT unit depends on N. The intermediate matrix is written to the memory modules row by row and the module number is incremented from 0 to N accordingly. The right module is identified by a one-hot write enable (wen) signal. A matrix is read from the memory column by column by accessing a single coefficient per each module and incrementing the *read address* (*raddr*) by one after each read (from 0 to N).

The high-speed 8/16/32-point DCT unit utilizes the whole memory for each N. To enable simultaneous reading of $32/N$ columns of the matrix without any access conflicts, the same rows are written to $(32/N)^2$ modules. Let us use $N = 8$ as an example. The first four rows are written in the modules 0-3, 8-11, 16-19, 24-27 after which the last four rows are written to the remaining modules respectively. Eight columns can now be read in two cycles by using raddr and offset.

4. PERFORMANCE ANALYSIS

Table 1 reports the cost-performance characteristics of the proposed and the most competitive prior-art FPGA implementations. The comparison is simplified by deriving

Table 1. Comparison of the proposed and related work.

Architecture	Transform	N	FPGA	Logic cells	DSPs	Freq.	Mtcoeffs/s	Cells/(Mtcoeffs/s)
Proposed (low-cost)	2-D DCT	8/16/32	Arria II	4 263 ALUTs	216	100 MHz	515	8.3
Proposed (high-speed)	2-D DCT	8/16/32	Arria II	8 114 ALUTs	344	160 MHz	1 224	6.6
Proposed (4×4)	2-D DCT/DST	4	Arria II	5 775 ALUTs	0	160 MHz	1 280	4.5
Jeske et al. [12]	1-D DCT	16	Stratix III	5 168 ALUTs	0	88 MHz	*701	7.4
Darji et al. [13]	1-D DCT	16	Spartan 3E	3 419 LEs	0	48 MHz	*384	**7.1
Zhao et al. [14]	2-D DCT	4/8/16/32	Cyclone IV	40 541 LEs	0	125 MHz	238	**136.3
Arayacheeppreecha et al. [15]	1-D DCT	4/8/16/32	Spartan 3A	15 677 LEs	77	205 MHz	*820	**15.3
Pastuszak et al. [16]	2-D DCT	8/16/32	Arria II	6 928 ALUTs	256	100 MHz	512	13.5
Pastuszak et al. [16]	2-D DCT/DST	4	Arria II	4 256 ALUTs	0	100 MHz	400	10.6

*Scaled sample rate (divided by two) **1.25 × LE = ALUT

Table 2. Logic cells (DSP blocks replaced by logic).

Architecture	Cells w/o DSPs	Cells/(Mtcoeffs/s)
Proposed (low-cost)	25 698	36.3
Proposed (high-speed)	38 829	23.0
Arayacheeppreecha et al. [15]	18 124	22.1
Pastuszak et al. [16]	29 744	42.3

normalized performance and cost figures for the architectures: sample rate as million tcoeffs processed per second (Mtcoeffs/s) and performance-cost ratio as logic cells per sample rate (cells/(Mtcoeffs/s)). The works in [12], [13], [15] only implement the 1-D transform. For fair comparison, their sample rates have been scaled (divided by two) to correspond to those of the 2-D transform architectures.

4.1 Proposed architecture

Table 1 tabulates the results for the proposed low-cost and high-speed variants of 8/16/32-point DCT units and for the 4-point DCT/DST unit separately. Altogether, the combined resource usage of our proposal is (4.2 + 5.8) kALUTs = 10.0 kALUTs and 216 DSP blocks in the low-cost case and (8.1 + 5.8) kALUTs = 13.9 kALUTs and 344 DSP blocks in the high-speed case. The low-cost approach uses 28% less ALUTs and 37% less DSP blocks than the high-speed one which has, on the other hand, almost 2.4× better sample rate.

The sample rate of our low-cost solution is adequate for transform coding of 4:2:0 1080p (1920 × 1080) video at 60 fps. The speed is computed for the worst case where the DCT/DST is needed once for all TBs in a CTU. It is also assumed here that there are always residual blocks available for the architecture. A practical HEVC intra/inter encoder can meet these conditions by coding successive CTUs in parallel without rate-distortion optimization. Respectively, the high-speed case is for 4:2:0 2160p (3840 × 2160) video at 30 fps.

On FPGA, the functionality of the proposed design was validated as a part of Kvazaar HEVC intra encoder.

4.2 Comparison with prior-art

The architecture proposed by Jeske et al. [12] is limited to $N = 16$ whereas the work of Darji et al. [13] supports all TBs but results are given for $N = 16$ only. Hence, the features of these two works are not directly comparable with our proposal. Furthermore, Zhao et al. [14] support all TB sizes but with non-competitive cost-performance figures.

The remaining approaches make also use of DSP blocks whose impact on the overall logic cell count is taken into account in Table 2. For the proposed designs, the total cell count is obtained by synthesizing them without the DSP blocks. The same cost per DSP block (72.5 ALUTs) is used when estimating the respective count for the related works.

The fastest prior-art solution is presented by Arayacheeppreecha et al. [15] whose overall cell count is also the smallest. However, including the missing DST unit and transpose memory would add overhead in their cost-performance figures. Furthermore, the cell counts of our both architectures are smaller if DSP blocks are available. Our high-speed architecture is also almost 1.5× faster.

Pastuszak et al. [16] present an approach similar to ours by implementing separate units for $N = 4$ and $N \in \{8, 16, 32\}$. However, our low-cost architecture is slightly faster and consumes still 14% less resources. Our high speed approach needs around 31% more resources but is around 2.4× faster.

5. CONCLUSIONS

This paper presented the first known HLS implementation for HEVC 2-D DCT/DST on FPGA. The presented architecture implements a hardware-oriented even-odd decomposition algorithm whose C code is synthesized to HDL with HLS. A low-cost variant of the architecture is able to support 1080p video up to 60 fps and a high-speed variant 2160p video up to 30 fps. HLS reduces design and verification times over traditional handwritten approaches. This work shows that these benefits do not come at the cost of implementation overhead but our HLS solution outperforms the prior-art approaches in terms of performance and cost.

6. ACKNOWLEDGMENT

This work was supported in part by the European Celtic-Plus Project 4KREPROSYS and the Academy of Finland (decision no. 301820).

7. REFERENCES

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

[2] *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.

[3] G. J. Sullivan, J. R. Ohm, W. J. Han, and T. Wiegand, "Overview of the high efficiency video coding (HEVC) standard," *IEEE Trans. Circuits Syst. Video Technol.*, vol. 22, no. 12, pp. 1649-1668, Dec. 2012.

[4] I. K. Kim, J. Min, T. Lee, W. J. Han, and J. Park, "Block partitioning structure in the HEVC standard," *IEEE Trans. Circuits Syst. Video Technol.*, vol. 22, no. 12, pp. 1697-1706, Dec. 2012.

[5] 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, pp. 1029-1041, Dec. 2013.

[6] A. Saxena and F. C. Fernandes, "CE7: Mode-dependent DCT/DST without 4×4 full matrix multiplication for intra prediction," *Document JCTVC-E125*, Geneva, Switzerland, Mar. 2011.

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

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

[9] *Joint Collaborative Team on Video Coding Reference Software, ver. HM 16.3* [Online]. Available: http://hevc.hhi.fraunhofer.de/

[10] P. K. Meher, S. Y. Park, B. K. Mohanty, K. S. Lim, and C. Yeo, "Efficient integer DCT architectures for HEVC," *IEEE Trans. Circuits Syst. Video Technol.*, vol. 24, no. 1, pp. 168-178, Jan. 2014.

[11] G. Pastuszak, "Hardware architectures for the H.265/HEVC discrete cosine transform," *IET Image Process.*, vol. 9, no. 6, pp. 468-477, 2015.

[12] R. Jeske, J. C. de Souza, G. Wrege, R. Conceição, M. Grellert, J. Mattos, and L. Agostini, "Low cost and high throughput multiplierless design of a 16 point 1-D DCT of the new HEVC video coding standard," *in Proc. Southern Conf. Programmable Logic*, Bento Goncalves, Spain, Mar. 2012.

[13] A. D. Darji and R. P. Makwana, "High-performance multiplierless DCT architecture for HEVC," *in Proc. Int. Symp. VLSI Design and Test*, Ahmedabad, India, Jun. 2015.

[14] W. Zhao, T. Onoye, and T. Song, "High-performance multiplierless transform architecture for HEVC," *in Proc. IEEE Int. Symp. Circuits Syst.*, Beijing, China, May 2013, pp. 1668-1671.

[15] P. Arayacheeppreecha, S. Pumrin, and B. Supmonchai, "Flexible input transform architecture for HEVC encoder on FPGA," *in Proc. Int. Conf. Electrical Engineering/ Electronics, Computer, Telecommunications and Information Tech.*, Hua Hin, Thailand, Jun. 2015.

[16] 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, pp. 210-222, Jan. 2016.

[17] P. Coussy, D. Gajski, M. Meredith, and A. Takach, "An introduction to high-level synthesis," *IEEE Des. Test. Comput.*, vol. 26, no. 4, pp. 8-17, Jul.-Aug. 2009.

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

[19] F. Bossen, "Common test conditions and software reference configurations," *Document JCTVC-J1100*, Stockholm, Sweden, Jul. 2012

PUBLICATION III

High-level synthesized 2-D IDCT/IDST implementation for HEVC codecs on FPGA

V. Viitamäki, P. Sjövall, J. Vanne, and T. D. Hämäläinen

In *Proceedings of IEEE International Symposium on Circuits and Systems*, Baltimore, Maryland, USA, May 2017
DOI: 10.1109/ISCAS.2017.8050323

High-level Synthesized 2-D IDCT/IDST Implementation for HEVC Codecs on FPGA

Vili Viitamäki, Panu Sjövall, Jarno Vanne, Timo D. Hämäläinen
Laboratory of Pervasive Computing
Tampere University of Technology
Tampere, Finland
{vili.viitamaki, panu.sjovall, jarno.vanne, timo.d.hamalainen}@tut.fi

***Abstract*— This paper presents efficient inverse discrete cosine transform (IDCT) and inverse discrete sine transform (IDST) implementations for High Efficiency Video Coding (HEVC). The proposal makes use of high-level synthesis (HLS) to implement a complete HEVC 2-D IDCT/IDST architecture directly from the C code of a well-known Even-Odd decomposition algorithm. The final architecture includes a 4-point IDCT/IDST unit for the smallest transform blocks (TB), an 8/16/32-point IDCT unit for the other TBs, and a transpose memory for intermediate results. On Arria II FPGA, it supports real-time (60 fps) HEVC decoding of up to 2160p format with 12.4 kALUTs and 344 DSP blocks. Compared with the other existing HLS approach, the proposed solution is almost 5 times faster and is able to utilize available FPGA resources better.**

Keywords*— High Efficiency Video Coding (HEVC); Inverse discrete cosine transform (DCT); Inverse discrete sine transform (DST); High-level synthesis (HLS); Field-programmable gate array (FPGA)

I. INTRODUCTION

High Efficiency Video Coding (*HEVC/H.265*) [1] is the newest international video coding standard published as twin text by ITU, ISO, and IEC as ITU-T H.265 | ISO/IEC 23008-2. It has been developed to address the increasing transmission and storage needs of modern video applications. HEVC is able to reduce the bit rate by almost 40% over the current mainstream standard AVC [2] for the same objective quality, but the respective encoding and decoding complexities tend to be at least 1.5 times higher [3]. The main reason for HEVC coding gain and complexity increase is a new HEVC coding structure that extends a traditional macroblock concept to an analogous block partitioning scheme with *coding tree units* (*CTUs*) of up to 64 × 64 pixels.

This paper addresses HEVC transform coding [4] for which the sizes of *transform blocks* (*TBs*) and associated core transform matrices [5] can be defined as $N \times N$, where $N \in \{4, 8, 16, 32\}$. Increasing the sizes of transform matrices from that of AVC to $N > 8$ improves *rate-distortion* (*RD*) performance by around 5-7% but it also introduces the majority of complexity overhead in HEVC transform coding [4].

HEVC standard specifies *two-dimensional* (*2-D*) integer *inverse discrete sine transform* (*IDST*) for intra coded luminance TBs of size 4 × 4 pixels and 2-D integer *inverse discrete cosine transform* (*IDCT*) for all other TBs [1]. Both of these separable 2-D transforms can be computed by two successive N-point 1-D transforms, first column-wise and then row-wise [5]. This indirect approach is called row-column decomposition and it is a widely used technique in software [6]-[7] and hardware implementations [8]-[11] of HEVC IDCT/IDST.

This work deals with *field-programmable gate array* (*FPGA*) implementations of HEVC IDCT/IDST architectures. However, our design is not written in traditional *hardware* (*HW*) *description languages* (*HDLs*), but the abstraction level in HW description is raised by *High-Level Synthesis* (*HLS*) [12]. HLS tools support well-known programming languages such as C and C++ in design description from which they can automatically generate the HDL. This approach makes the code more readable, shortens design and verification times, and increases the design reusability over those of handwritten HDL equivalents.

In our recent work [13], we proposed to use HLS for HEVC DCT/DST. This work utilizes the same HLS flow than in [13] and applies it to IDCT and IDST algorithms. The created 2-D IDCT/IDST architecture includes an 8/16/32-point IDCT unit for $N \in \{8, 16, 32\}$, a separate 4-point IDCT/IDST unit for $N = 4$, and a transpose memory for intermediate results. The architecture is implemented on Arria II FPGA using Catapult C [14] HLS tool. For the time being, only a single HLS implementation has been presented for HEVC IDCT [8] and none for HEVC IDST. Thus, this paper presents the first HLS implementation for a complete HEVC 2-D IDCT/IDST.

The remainder of the paper is organized as follows. Section 2 describes the adopted hardware-oriented IDCT and IDST algorithms. Section 3 proposes our HLS implementation for HEVC 2-D IDCT/IDST. In Section 4, performance characteristics of our proposal are reported and compared with the prior-art. Section 5 concludes the paper.

II. 2-D INTEGER IDCT/IDST ALGORITHMS IN HEVC

In this work, the C implementations of IDCT and IDST algorithms are taken from the open-source Kvazaar HEVC encoder [6]. Basically, Kvazaar implements the same IDCT/IDST functionality than *HEVC reference encoder* (*HM*) [7] but the hardware-oriented C source code of Kvazaar provides a better starting point for HLS.

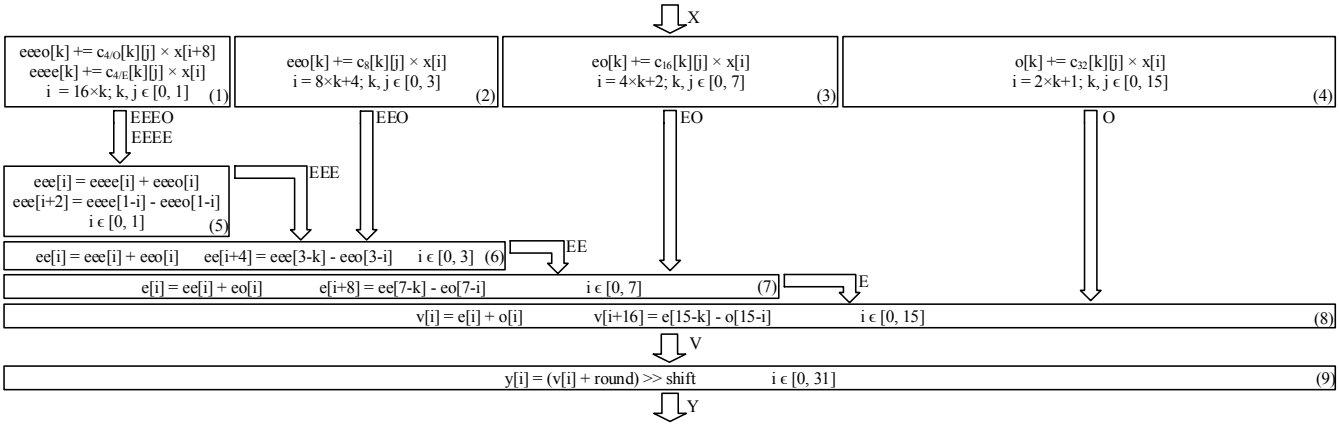


Fig. 1. Even-Odd decomposition algorithm for IDCT (N = 32).

A. Even-Odd decomposition algorithm

In HEVC codec, IDCT and IDST are used to convert transform-domain coefficient matrices back into spatial-domain residual blocks. A well-known row-column algorithm [4] executes these 2-D inverse transforms with separable 1-D inverse transforms in two consecutive stages. An N-point inverse transform is first applied 1) to each column of a transform-domain coefficient matrix of size $N \times N$ to generate an intermediate matrix of size $N \times N$; and then 2) to each row of the intermediate matrix to generate a final spatial-domain residual block of size $N \times N$.

The number of arithmetic operations can be further reduced by implementing these 1-D inverse transforms with Even-Odd decomposition algorithm, a.k.a., Partial Butterfly algorithm [5]. It decomposes an input and core transform matrices of size $N \times N$ into two matrices of size $N/2 \times N/2$ according to even and odd columns/rows, respectively. The core transform matrices for each N (C_N) are specified in [5]. Now, an N-point inverse transform can be computed with two N/2-point transforms so that the matrix multiplication is done separately for even and odd cases after which the result is yielded with basic add and subtract operations. The respective decomposition can be applied recursively down to N = 4 to reduce arithmetic operations further. In this approach, the largest transform matrix also embeds the smaller transform matrices.

B. Example: 32-point IDCT

Fig. 1 depicts the Even-Odd decomposition of the 1-D inverse transform for N = 32. The transform coefficients of the input vector $\mathbf{Y} = [y(0), y(1), \ldots, y(31)]$ are recursively decomposed into five parts which can be multiplied in parallel by transform matrices as shown by (1), (2), (3), and (4) in Fig. 1. That is, $\{y(8), y(24)\}$ are multiplied by $\mathbf{C}_{4/O}$, $\{y(0), y(16)\}$ by $\mathbf{C}_{4/E}$, $\{y(4), y(12), y(20), y(28)\}$ by $\mathbf{C}_8$, $\{y(2), y(6), \ldots y(30)\}$ by $\mathbf{C}_{16}$, and $\{y(1), y(3),\ldots y(31)\}$ by $\mathbf{C}_{32}$ to yield 2-point vectors **EEEO** and **EEEE,** a 4-point vector **EEO**, an 8-point vector **EO**, and a 16-point vector **O**, respectively.

In the next stage, a 4-point vector **EEE** is computed from the vectors **EEEE** and **EEEO** with add and subtract operations as in (5). Correspondingly, an 8-point vector **EE** is derived from the vectors **EEE** and **EEO** as in (6), a 16-point vector **E** from the vectors **EE** and **EO** as in (7), and finally a 32-point vector **V** from the vectors **E** and **O** as in (8). The residual output vector $\mathbf{X} = [x(0), x(1), \ldots, x(31)]$ is then formed by scaling the vector **V**. The scaling factor depends on the transform stage (column or row) and on the video bit depth (8 bits in our case) [5].

III. PROPOSED IDCT/IDST ARCHITECTURE

The proposed IDCT/IDST architecture is composed of 1) an 8/16/32-point IDCT unit for TBs of size 8 × 8, 16 × 16, and 32 × 32; 2) a transpose memory for intermediate results; and 3) a separate 4-point IDCT/IDST unit for TBs of size 4 × 4.

A. 8/16/32-point IDCT unit

Fig. 2 depicts a block diagram of the 8/16/32-point IDCT unit. It can be divided into three parts: 1) a control block ($\text{Ctrl}_{8/16/32}$); 2) a 3-state IDCT computation pipeline; and 3) a transpose memory.

The $\text{Ctrl}_{8/16/32}$ block has a 512-bit input for 32 16-bit signed *transform coefficients* (tcoeffs). It extends the coefficients with configuration bits and passes them to the IDCT computation. The configuration bits are used to define the block size and the scaling factor. After the first pass, the $\text{Ctrl}_{8/16/32}$ block extends the transposed results with new configuration bits and sends them back to the IDCT computation.

The IDCT stage 1 performs multiplications between the transform matrices ($\mathbf{C}_{32}$, $\mathbf{C}_{16}$, $\mathbf{C}_8$, $\mathbf{C}_{4/O}$, and $\mathbf{C}_{4/E}$) and input (**Y**). To save logic cells on an FPGA, multiplications are mapped to DSP blocks. Catapult-C facilitates instantiation of DSP blocks in C code by providing a library for DSP blocks as C++ templates for different FPGA architectures.

The IDCT stage 2 includes three addition/subtraction levels to compose the final even vector (**E**) from the decomposed even and odd vectors (**EEEO**, **EEEE**, **EEO**, **EEE**, **EO**, and **EE**). Fig. 1 depicts this for N = 32 in (5) – (7). For N < 32, all these levels are used to calculate 32/N rows or columns at a time. For example, for N = 8, four rows or columns are decomposed into the **EEEO**, **EEEE**, **EEO**, **EO**, and **O** vectors in parallel to take full advantage of the available adding and subtracting levels.

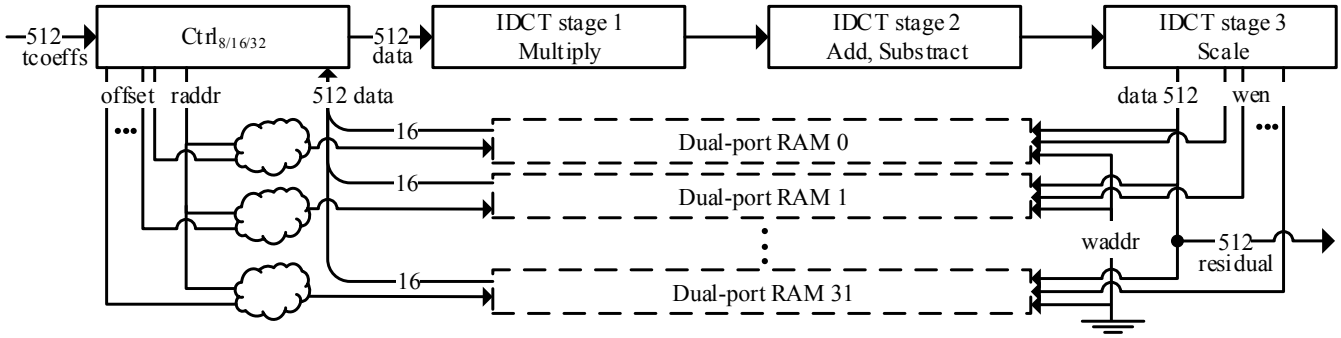


Fig. 2. Block diagram of the pipelined 8/16/32-point IDCT unit and a transpose memory.

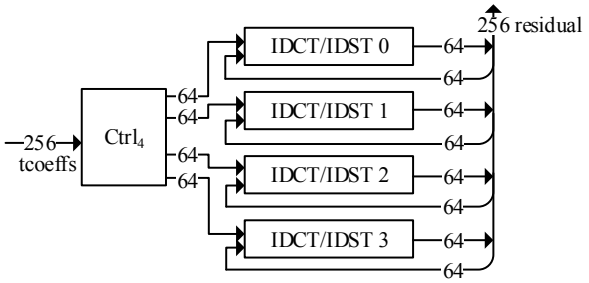


Fig. 3. Block diagram of the separate 4-point 2-D IDCT/IDST unit.

The IDCT stage 3 finalizes the 1-D transform by combining the **E** and **O** vectors and by scaling the final result (**V**) to 16-bit signed residuals (**X**). Before outputting the final residual vector, it is permuted back to its original order.

The 8/16/32-point IDCT unit performs the 2-D IDCT in two successive passes and the intermediate data is stored in the transpose memory. 32 coefficients (32/*N* rows or columns of a TB) are processed in parallel to ensure a more constant hardware utilization. The latency for both passes is three clock cycles. Finally, the 32 16-bit residuals are sent via the 512-bit output. The same output is connected to the transpose memory.

B. Transpose memory

Fig. 2 also shows the structure of the transpose memory used with the 8/16/32-point IDCT unit. On FPGA, it is made of 32 dual-port on-chip memory modules without registers. Each memory module has a 512-bit write (32 coefficients) and a 16-bit (1 coefficient) read port. The structure supports a block transpose for $N \in \{8, 16, 32\}$.

The 8/16/32-point IDCT unit utilizes the whole memory for each *N*. To enable simultaneous reading of 32/*N* rows of the matrix without any access conflicts, the same columns of the matrix are written to $(32/N)^2$ modules. The right modules are identified by a *write enable* (*wen*) signal. Let us use $N = 8$ as an example. The first four columns are written in the modules 0-3, 8-11, 16-19, 24-27 after which the last four columns are written to the remaining modules respectively. Eight rows can now be read in two clock cycles by using raddr and offset.

C. 4-point IDCT/IDST unit

Fig. 3 depicts a 4-point IDCT/IDST unit that can operate in parallel with the 8/16/32-point IDCT unit. The 256-bit input to the $Ctrl_4$ block accepts one 4 × 4 coefficient block at a time. The 16 16-bit coefficients are passed column-wise to the respective four IDCT/IDST blocks. The intermediate matrix is ready in one clock cycle after which it is sent back to the same IDCT/IDST blocks by picking the intermediate values from the registers in a transposed order. After these two passes, the unit outputs 16 16-bit residuals.

A separate 4-point IDCT/IDST unit increases the occupied resources on FPGA. However, this overhead is compensated by better load balancing since the share of 4 × 4 TBs is relatively high compared to the other TBs.

IV. Performance Analysis

Table 1 reports the cost-performance characteristics of the proposed and the most competitive prior-art FPGA implementations.

A. Proposed architecture

Table 1 tabulates results for the proposed 8/16/32-point IDCT unit and for the 4-point IDCT/IDST unit separately. Altogether, the combined resource usage of our proposal is (6.9 + 5.6) kALUTs = 12.4 kALUTs and 344 DSP blocks. The total cell count rises to 49.9 kALUTs if the DSP blocks are not used. It is assumed here that one DSP block (DSP18×18) equals to 109 ALUTs. This relationship was obtained by synthesizing an equivalent DSP functionality using only logic cells.

The proposed design is capable of supporting 4:2:0 Ultra HD (3840 × 2160) video decoding at 68 *frames per second* (*fps*) and 4:2:0 Ultra HD video encoding at 35 fps. The reported speeds are for the worst case scenarios: a decoded bit stream is assumed to contain TBs of size 8 × 8 pixels only and an encoder is assumed to encode all TBs in a CTU when searching for the best block partitioning.

The functionality of the proposed design was validated on FPGA, as part of Kvazaar HEVC intra encoder.

B. Comparison with prior-art

Kalali et al. [8] present the first HSL implementations for HEVC IDCT. Altogether, they proposed three implementations

TABLE 1. COMPARISON OF THE PROPOSED AND RELATED IDCT/IDST ARCHITECTURES ON FPGA

Architecture	HLS	Transform	N	FPGA	Logic cells	DSPs	Cells w/o DSPs	Freq.	Speed (worst case)
Proposed	YES	2-D IDCT	8/16/32	Arria II	6 859 ALUTs	344	44355 ALUTs	150	2160@68fps
Proposed (4x4)	YES	2-D IDCT/IDST	4	Arria II	5 559 ALUTs	0	5559 ALUTs	150	2160@96fps
Kalali et al. [8]	YES	2-D IDCT	4/8/16/32	Virtex 6	13 669 4-LUTs	0	*13669 ALUTs	110	1080@55fps
Pastuszak et al. [9]	NO	2-D IDCT	8/16/32	Arria II	3 079 ALUTs	512	58887 ALUTs	200	2160@64fps
Pastuszak et al. [9]	NO	2-D IDCT/IDST	4	Arria II	3 554 ALUTs	0	3554 ALUTs	100	2160@32fps
Kalali et al. [10]	NO	2-D IDCT/IDST	4/8/16/32	Virtex 6	38 790 4-LUTs	0	*38790 ALUTs	150	2160@48fps
Conceição et al. [11]	NO	2-D IDCT	4/8/16/32	Stratix V	17 340 ALMs	0	**34680 ALUTs	63	2160@20fps

*4-LUT = ALUT **ALM = 2×ALUT

done with three different HLS tools. The best performance characteristics were obtained with MATLAB Simulink HDL Coder. It supports IDCT for all TB sizes but is missing the IDST unit. In addition, it is only capable of decoding 1080p video at 55 fps. Our solution is 3.7 times larger when DSPs are normalized to logic cells, but also 5.0 times faster.

Pastuszak et al. [9] introduce an approach similar to ours by implementing separate units for $N = 4$ and $N \in \{8, 16, 32\}$. However, our approach is slightly faster and consumes around 25% less resources when DSPs are normalized to logic cells. Our 4-point IDCT/IDST unit, compared with their counterpart, consumes 1.6 times more resources, but is three times faster.

Kalali et al. [10] also propose a handwritten Verilog RTL implementation, that is notably faster but also larger than their HLS implementation. Compared with our solution, the cell count is much higher as it does not utilize DSP blocks. With DSPs normalized to logic cells, our architecture is 1.3 times larger, but also 1.4 times faster.

Conceição et al. [11] present an implementation that supports IDCT for all TB sizes but not IDST. Their implementation does not utilize DSP blocks. When DSPs are normalized to logic cells, our proposal requires 1.4 times more logic cells, but is also 3.4 times faster.

V. CONCLUSION

This paper presented the first complete HLS design for HEVC 2-D IDCT/IDST on FPGA. The proposed design implements a hardware-oriented Even-Odd decomposition algorithm whose C code is directly synthesized to HDL with HLS. The architecture supports 2160p video decoding up to 68 fps at a cost of 12.4 kALUTs and 344 DSP blocks. It is almost 5 times faster than the existing HLS implementation for IDCT and consumes less logic cells on an FPGA due to efficient mapping of computation to the available DSPs.

Compared with traditional approaches, HLS design techniques are known to increase design productivity, code readability, and design reusability. The presented results also show that our HLS solution is very competitive with the existing non-HLS IDCT/IDST solutions in terms of performance and cost. Hence, the main conclusion of this paper is that the manifold benefits of HLS do not come at the cost of implementation overhead.

ACKNOWLEDGMENT

This work was supported in part by the European Celtic-Plus Project 4KREPROSYS and Academy of Finland (decision number 301820).

REFERENCES

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

[2] *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.

[3] 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 pp. 1885-1898, Dec. 2012.

[4] 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, pp. 1029-1041, Dec. 2013.

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

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

[7] *Joint Collaborative Team on Video Coding Reference Software, ver. HM 16.3* [Online]. Available: http://hevc.hhi.fraunhofer.de/

[8] E. Kalali and I. Hamzaoglu, "FPGA implementations of HEVC inverse DCT using high-level synthesis," *in Proc. Conf. Design and Architectures for Signal and Image Processing*, Krakow, Poland, Sep. 2015.

[9] 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, pp. 210-222, Jan. 2016.

[10] E. Kalali, E. Ozcan, O. M. Yalcinkaya, and I. Hamzaoglu, "A low energy HEVC inverse transform hardware," *IEEE Trans. Consumer Electron.*, vol. 60, no. 4, pp. 754-761, Nov. 2014.

[11] R. Conceição, J. C. de Souza, R. Jeske, M. Porto, B. Zatt, and L. Agostini, "Power efficient and high troughtput multi-size IDCT targeting UHD HEVC decoders," *in Proc. IEEE Int. Symp. Circuits Syst.*, Melbourne, Australia, Jun. 2014.

[12] P. Coussy, D. Gajski, M. Meredith, and A. Takach, "An introduction to high-level synthesis," *IEEE Des. Test. Comput.*, vol. 26, no. 4, pp. 8-17, Jul.-Aug. 2009.

[13] P. Sjövall, V. Viitamäki, J. Vanne, and T. D. Hämäläinen, "High-level synthesis implementation of HEVC 2-D DCT/DST on FPGA," *in Proc. IEEE Int. Conf. Acoustics, Speech, Signal Process.*, New Orleans, Louisiana, USA, Mar. 2017.

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

PUBLICATION
IV

Are we there yet? A study on the state of high-level synthesis

S. Lahti, P. Sjövall, J. Vanne, and T. D. Hämäläinen

IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, vol. 38, no. 5, pp. 898–911
DOI: 10.1109/TCAD.2018.2834439

Are We There Yet? A Study on the State of High-level Synthesis

Sakari Lahti, *Graduate Student Member, IEEE*, Panu Sjövall, *Graduate Student Member, IEEE*, Jarno Vanne, *Member, IEEE,* and Timo D. Hämäläinen, *Member, IEEE*

***Abstract*—To increase productivity in designing digital hardware components, high-level synthesis (HLS) is seen as the next step in raising the design abstraction level. However, the quality of results (QoR) of HLS tools has tended to be behind those of manual register-transfer level (RTL) flows. In this paper, we survey the scientific literature published since 2010 about the QoR and productivity differences between the HLS and RTL design flows. Altogether, our survey spans 46 papers and 118 associated applications. Our results show that on average, the QoR of RTL flow is still better than that of the state-of-the-art HLS tools. However, the average development time with HLS tools is only a third of that of the RTL flow, and a designer obtains over four times as high productivity with HLS. Based on our findings, we also present a model case study to sum up the best practices in comparative studies between HLS and RTL. The outcome of our case study is also in line with the survey results, as using an HLS tool is seen to increase the productivity by a factor of six. In addition, to help close the QoR gap, we present a survey of literature focused on improving HLS. Our results let us conclude that HLS is currently a viable option for fast prototyping and for designs with short time to market.**

***Index Terms*—Electronic design automation (EDA) and methodology, field programmable gate array (FPGA), hardware description languages (HDL), high level synthesis (HLS), reconfigurable logic**

I. Introduction

For decades now, *register-transfer level* (RTL) has been the dominant method to describe *very large scale integration* (VLSI) systems and their constituent intellectual property blocks. Whereas the RTL tools have developed only incrementally, the complexity of the VLSI systems has raised exponentially, which has made the design and verification process a bottleneck for productivity [1].

High-level synthesis (HLS) promises to alleviate this problem by a variety of ways [2]–[5]. In HLS, the application is described on a behavioral level, omitting implementation details such as timing and the nature of interface and memory elements. These details are determined using an HLS tool that takes the behavioral description as an input. The designer can select the target technology in the tool and map the interface and memory variables to specified technology-dependent elements. The HLS tool then produces an RTL description based on the target technology and microarchitectural choices.

The promises of HLS are many.

1) Initial design effort is reduced by raising the abstraction level. The designer can concentrate on describing the behavior of the system without having to spend time implementing the microarchitectural details. Introduction of bugs in the code is also less likely on a higher level of abstraction.
2) Verification is accelerated. The behavior of the design can often be verified using software verification tools that are faster and simpler to use than RTL simulation tools. Furthermore, the RTL output of the HLS tool can be verified by using the original behavioral test bench, as the tool can check that the results of both models are identical.
3) Design space exploration (DSE) is faster. The microarchitecture can be explored by making choices in the HLS tool, which require little or no modifications to the code. Thus, several transformations such as pipelining and various loop unrolling factors can be explored in a matter of hours. This is a tremendous improvement upon RTL methodology, where these kind of changes would require significant modifications to the source code.
4) Targeting new platforms is straightforward. If the target platform changes, the HLS tool is able to modify the RTL output accordingly. For example, if the new platform has a different clock frequency, the HLS tool reschedules operations according to the new frequency.
5) HLS is accessible to software engineers. Whereas RTL design requires knowledge of languages such as VHDL and Verilog, HLS tools usually use familiar languages such as C/C++. The HLS tool can take care of most of the hardware specific implementation details, so the threshold of software engineers to tackle hardware projects is greatly reduced. That said, to obtain optimal results, hardware expertise is still useful when employing HLS.

Together, these benefits reduce the design and verification time, push down the development costs, and lower the bar for tackling hardware projects. Consequently, the time to market is shortened, and using hardware acceleration on heterogeneous systems becomes a more attractive option.

The rise of *field-programmable gate arrays* (FPGAs) is also an enabling factor for HLS. FPGAs are ideal platforms for HLS

TABLE I
NUMBER OF PAPERS PUBLISHED BY YEAR

Year	Papers
2010	4
2011	5
2012	3
2013	8
2014	10
2015	8
2016	8

designs, as they allow quick prototyping, have rapid design cycle, and are inherently reprogrammable. Modern HLS tools usually contain a wide library of FPGA technologies for design targeting.

The history of HLS dates back to the 1970s and 1980s, but it was not until the turn of the century that it became a viable option for the industry [2]. One of the reasons for the slow adoption is that the *quality of results* (QoR), such as resource usage and performance, was initially poor compared with the RTL approach. The QoR has improved with the newest generation of HLS tools, but the results reported in individual studies still vary, and it is unclear whether the QoR gap has been closed yet.

The goal of this paper is to answer this question by a literature review. We examine 46 recent papers that compare the QoR and development effort of HLS and RTL approaches for the same applications. Our work has four main contributions:

1) a comparative analysis of the QoR and design effort of HLS and RTL reported in scientific articles;
2) a case study presenting the best practices for comparing HLS and RTL approaches with a test group that uses both flows to implement a part of a HEVC/H.265 video encoder;
3) a survey of the literature suggesting research directions and ways to improve HLS;
4) conclusions on the current state of the art in HLS.

To the best of our knowledge, this is the first comprehensive quantitative study that uses a wide variety of sources to compare the QoR and design effort of HLS and RTL flows. Previous works have instead focused on comparing different HLS tools to each other [6], [7]. Other papers have provided insights on how to close the QoR gap against RTL or otherwise improve HLS tools [5], [8]. A thorough quantitative analysis on the current state of HLS has been missing, however, which this paper amends.

The rest of the paper is structured as follows. Section 2 describes our criteria for selecting papers for this study. Section 3 contains a meta-analysis of the reviewed papers, summarizing what kind of information was reported in them. In Section 4, we show and analyze the results from the literature study, and Section 5 describes our test group study with its results. Section 6 reviews papers that propose improvements to HLS, and finally section 7 concludes the paper with some discussion of the results.

II. QUALIFYING PAPERS

For this study, we examined papers published in 2010 or later to get a comprehensive view of the latest HLS works. Altogether, we found over a thousand candidate papers and selected those articles for further study whose abstracts stated that: 1) one or more applications were implemented using HLS; and 2) the obtained results were compared with equivalent self-made or referenced RTL applications.

We also required the qualifying papers to list one or more of the following metrics for *both* the HLS and RTL versions of the applications:

1) performance with an application specific metric;
2) execution time and/or latency;
3) maximum achievable clock frequency on target platform;
4) area on application-specific integrated circuit (ASIC);
5) resource usage on FPGA;
6) power consumption;
7) development time;
8) lines of input source code (LoC).

In total, we found 46 qualifying papers out of which 39 were from IEEE Xplore, two from Springer Link, one from ACM Digital Library, two from arXiv.org, one from EBSCOhost, and one from Science Direct. Basic information on all the reviewed papers is given in Table IX in the Appendix. As can be seen from the table, the range of applications is very diverse. This makes it impractical to analyze the QoR results by the type of application, which would otherwise give interesting insight on the strengths and weaknesses of HLS. A qualitative analysis like that would also benefit from access to the implementations' source codes, which are seldom available.

Table I shows a breakdown of the number of qualifying papers published each year. Because the number of papers from each year is low, it is not feasible to use our data to check for a possible trend in the QoR of HLS during these years. A longer year range would also be preferable for that kind of study.

III. META-ANALYSIS

Table II gathers a summary of the metrics of interest and their frequency of occurrence in the reviewed papers. In general, the reviewed works have much variance in the reported details about the experimental setup and results. The table counts only those papers that report the results in exact terms either in absolute values or in percentages. Inexact values, such as "the execution time was less than 100 ms," were excluded from our quantitative analysis.

Twenty-two articles report results for more than one application or experimental setup. In many works, multiple different applications were implemented, often related to each other (for example, [9]–[11]). Some authors compared different HLS tools [12]–[14], whereas others compared various micro-architectural optimizations, such as loop unrolling and pipelining [15], [16], or different FPGA chips [17], [18]. The data set is thus larger than the mere number of qualifying papers

TABLE II
THE METRICS AND THE FREQUENCE OF THEIR OCCURRENCE IN THE REVIEWED PAPERS

Metric	Number of papers reporting (percentage of total)	Number of applications for which the metric was reported
HLS tool	42 (91%)	-
HLS input language	46 (100%)	-
Lines of input code	13 (28%)	36
Development time	15 (33%)	25
Maximum frequency	24 (52%)	74
Latency	10 (22%)	17
Execution time	8 (17%)	14
Performance	15 (33%)	46
FPGA LUTs/LCs/LEs/Slices	36 (78%)	92
FPGA Flip-flops	23 (50%)	63
FPGA DSP blocks	22 (48%)	50
FPGA BRAM	22 (48%)	55
ASIC area	4 (9%)	8
Power consumption	3 (7%)	7
Total papers	46	

TABLE III
HLS TOOL USAGE BY PAPERS

HLS Tool	N	Tool language
AccelDSP	1	MATLAB
Altera OpenCL	3	OpenCL
Bluespec	2	Bluespec language
C2RTL	1	C
Cadence Stratus	1	C/C++/SystemC
CAPH Toolset	2	CAPH language
Catapult-C	2	C/C++/SystemC
Chisel	2	Scala
Cadence C-to-Silicon	2	C/C++/SystemC
Convey Hybrid-Threading	1	HT language
HCE	2	C
HIPAcc	2	HIPAcc language
Impulse C	1	C
LegUp	3	C
MATLAB Simulink HDL Coder	1	MATLAB functions, Simulink model
Maxeler MaxCompiler	1	Java
ROCCC	1	C
Xilinx System Generator for DSP	2	MATLAB Simulink
Xilinx Vivado HLS/Autopilot	18	C/C++/SystemC
Undisclosed	4	N/A

would suggest. For brevity, we shall call each of these individual results *applications*, regardless of whether they are based on actual different applications, HLS tools, FPGA chips, or other variations. The third column of Table II shows the total number of applications for which the given metric is reported.

Development time is of great interest when comparing the HLS and RTL methodologies. However, only a third of the papers report the development time, which can be seen as a flaw in the articles omitting it. Of the various QoR metrics, FPGA resource usage is reported more often than the performance values. Only four papers target ASIC implementation (instead of FPGA), and thus there is not enough data to compare ASIC area results. The same is true for power consumption.

Almost all papers report the used HLS tool. The remaining works give no reason for not revealing the information, but license agreements may have been the cause. However, even those articles mention the HLS input language.

Table III shows a summary of the used HLS tools. The second column tells the number of occurrences of each tool and the third column their input languages. The table suggests that Vivado HLS (formerly known as Autopilot) is the most popular HLS tool, at least in academia. All the other tools gain only scattered usage. Vivado's popularity is probably due to Xilinx being the leading FPGA vendor, whose design suite for FPGAs includes Vivado HLS. The large number of used HLS tools also speaks of the relative immaturity of the field.

Of the 46 qualifying works, 39 used self-made RTL implementations for comparison with HLS and seven cited RTL results from other research groups. There are additional papers that would have qualified for this study, but they cite papers with incompatible RTL implementations, which resulted in their preclusion. For example, the FPGA chip used for RTL was from a different family, which prevented fair resource usage comparison.

IV. COMPARATIVE STUDY RESULTS AND ANALYSIS

A. On the QoR Metrics

The fundamental building block of FPGAs is a *configurable logic block* (CLB) or a *logic array block* (LAB), depending on the FPGA vendor and device. CLB/LAB consists of a few logical cells that may be called *logic cells* (LCs), *logic elements* (LEs), or *adaptive logic modules* (ALMs). These logical cells are made of look-up tables and flip-flops. The reviewed papers usually report one of these figures when synthesizing an application for FPGA. For the purposes of this study, it is irrelevant which figure was reported, since we are interested in the *ratio* of resource usage between HLS and RTL. Thus, we have grouped all of these resource metrics under the same term called *basic FPGA resources*.

FPGAs also contain other resources such as DSP blocks and on-chip block RAM (BRAM) memories, which cannot be converted to CLB equivalents without sufficient data from the FPGA vendors. This would require knowing the exact FPGA chip type, but only about 60% of the reviewed papers report it, and the others merely state the used FPGA family. Therefore, we had no universal way to combine all the resource metrics into a single resource usage value, which could be compared across applications. Thus, we discarded this approach and chose CLBs or its constituents as the basis for resource usage comparisons.

The reviewed papers also use various different performance metrics depending on the implemented application. These can be divided into four categories: 1) performance, 2) execution time, 3) latency, and 4) maximum frequency. In this context, performance can be interpreted in several ways depending on

TABLE IV
SUMMARY OF THE NUMERICAL DATA FROM THE PAPERS

Metric	N	HLS/RTL mean	Geometric std. dev.	HLS better or equal to RTL
Lines of code	36	**0.52**	2.26	75 %
Development time	25	**0.32**	2.59	88 %
Performance	46	0.47	5.50	39 %
Execution time	14	1.70	2.21	39 %
Latency	17	1.05	2.07	35 %
Maximum frequency	74	0.88	1.48	42 %
Basic FPGA resources	92	1.41	3.76	33 %
DSP blocks	50	1.11	-	68 %
BRAM blocks	29	**0.49**	-	45 %
BRAM (kB)	27	1.47	-	33 %

the application. For example, for a video encoder, it would mean frames per second, and for a cryptography module, it would mean encrypted bits per second. For applications with a clear start and finish, execution time is often reported, and some papers report latency, i.e. the number of clock cycles for processing a sample. The most often reported performance metric is the maximum frequency for which the application can be scheduled on the target FPGA.

We wanted to include as many performance metrics as possible so all of them are used in our study. For papers that report more than one metric, we prioritize performance over execution time, execution time over latency, and latency over maximum frequency. Thus, we use only one of these values per application rather than try to create an arbitrary aggregate performance metric. In the figures of the following subsections, we shall call the selected value just *performance*. We have also inverted execution time and latency values in calculations for the figures so that a larger value is always better. The way to examine the various data cloud figures in the following subsections is not to compare individual data points to each other but to concentrate on the center of gravity and dispersion of the data.

B. Numerical Analysis

Table IV gathers the numerical aggregate data of our findings. *N* denotes the number of applications for which the corresponding data were reported. The third column reports the mean of the ratios between HLS and RTL results. For all the values except DSP blocks and BRAM, we used the geometric mean rather than the arithmetic one, since the values in each category can differ by orders of magnitude because of the wide variety of applications. For DSP blocks and BRAM, the geometric mean could not be calculated because of the zeros in the data set, so arithmetic mean was used instead. Bolded mean values favor HLS while unbolded values favor RTL. The fourth column shows the geometric standard deviation (GSD). Note that it is a multiplicative value: The lower bound is obtained by dividing by the GSD and the upper bound is obtained by multiplying by the GSD. The last column shows the percentage of results for which the HLS application performed as well or better than the corresponding RTL version.

As expected, HLS outperforms RTL in both development time and lines of source code. The average development time is only about a third of a corresponding RTL application. We also examined the HLS to RTL *development time ratio* as a function of the *absolute development time* to see if the scale of the project had an effect on the ratio, but found no correlation. Thus, it seems that for both large-scale and small-scale applications the reduction in development time is the same. On the other hand, the respective comparison with code size shows that for larger applications (1,000 LoC or more), HLS code seems to be more compact compared with RTL code. In fact, in all the cases where there was more HLS LoC than RTL LoC, the code size was less than 250 LoC. With smaller code size, non-behavioral code takes a relatively larger part of the total, which seems to favor RTL.

In performance and execution time, the HLS design is on average clearly inferior, but in latency and maximum frequency the difference is less prominent. The HLS approach also loses in basic resource usage: On average, HLS uses 41% more basic FPGA resources than RTL. With BRAM and DSP blocks, the results are ambivalent. Based on papers, which report the number of used BRAM blocks, HLS seems to use them more efficiently, but with papers, which report BRAM usage in kilobytes, RTL wins. In DSP block usage, HLS and RTL seem similar. .

We also examined how the HLS input language affects the QoR. In [19], the HLS tools are divided into five categories based on their style of describing the input: hardware description language (HDL) like frameworks, C based frameworks, high-level language (HLL) based frameworks (these are highly abstract, usually object-oriented languages), model based frameworks (using executable specification, e.g. NI LabView and Matlab HDL Coder), and CUDA/OpenCL based frameworks. In our study, we found five applications implemented with HDL like, 77 with C based, 10 with HLL based, six with model based, and 11 with CUDA/OpenCL based frameworks. Since other than C based frameworks receive only scattered usage, it is not prudent to compare all the categories with each other. Instead, we compare the QoR of C based frameworks and all the others. The results are shown in Table V, where *N* denotes the number of comparable results. It seems that C based frameworks produce designs with worse performance than the other frameworks but save in basic resource usage. Looking further into the data, we noticed that

TABLE V
COMPARISON OF QOR BY FRAMEWORK TYPE

	N	HLS/RTL performance ratio	N	HLS/RTL Basic resouce usage ratio
C based framework	100	0.64	71	1.26
Other frameworks	51	0.84	36	1.50

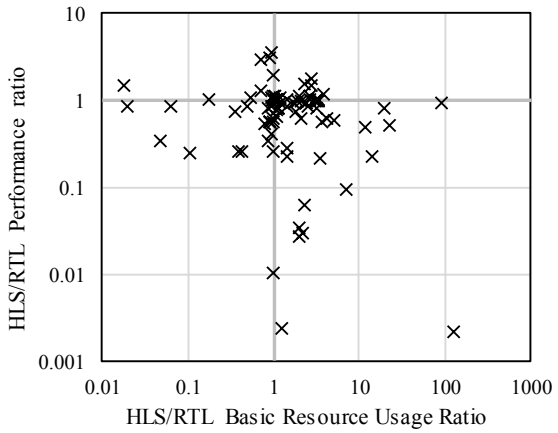


Fig. 1. Scatter graph of the HLS to RTL ratio between performance and basic resource usage for different applications.

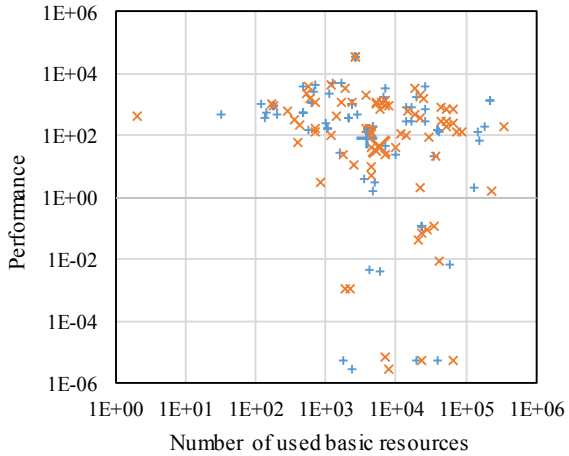


Fig. 2. HLS (orange "x") and RTL (blue "+") performance and basic resource usage for each application. Note that the performance does not have a listed unit as it varies from application to application.

the CUDA/OpenCL based frameworks were especially resource consuming (3.56×) and produced the worst performance (0.56×).

C. Comparisons between Resource Usage and Performance

To better illustrate the QoR differences, Fig. 1 shows the relative HLS/RTL performance against the relative HLS/RTL basic resource usage for each application. Each "X" in the figure represents a single application. The wider horizontal and vertical lines denote break-even lines where the performance and basic resource usage are the same for both HLS and RTL, respectively. Most of the marks are clustered around the intersection of the break-even lines, indicating that in the great

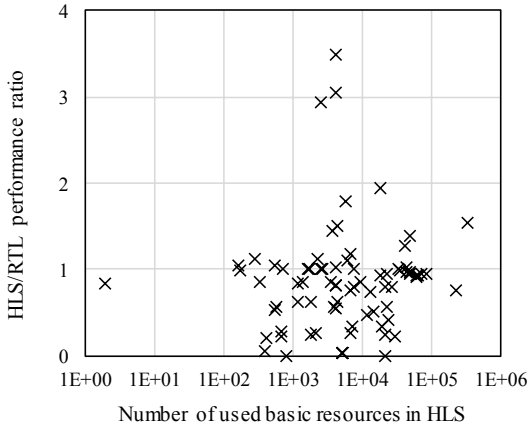


Fig. 3. HLS/RTL performance vs. HLS basic resource usage.

majority of cases the performance and basic resource usage difference between HLS and RTL is relatively small. Nevertheless, there are more marks towards the right and bottom of the figure than in the opposite directions, showing that RTL tends to outmatch HLS in both regards.

Another way to look at the same data is depicted in Fig. 2, which shows the absolute performance and basic area usage values for both HLS applications ("+") and RTL applications ("x"). The large, partially overlapping symbols show the centers of gravity based on geometric means for both metrics correspondingly. The data point clouds are largely overlapping, and the centers of gravity lie close to each other. Thus, on average there is no radical difference between the HLS and RTL QoRs, but RTL fares somewhat better.

We also wanted to see, whether there exists any correlation between the relative HLS/RTL performance and the absolute numbers of basic resource usage. That is, does the relative performance between HLS and RTL designs change as a function of consumed FPGA resources. Our hypothesis was that the HLS tools' ability to optimize data path and control logic might be more limited with larger applications. The results are plotted in Fig. 3, which shows that there is no clear correlation, and indeed, the Pearson correlation coefficient is only 0.10 for this data set. Thus, the size of the design does not seem to affect the HLS tools' ability to optimize performance.

D. Comparisons Based on Design Effort

Fig. 4 shows the HLS/RTL development time ratio for applications for which the development time was reported. In all but three cases, the ratio is less than one, and in 72% of cases, it is less than 0.5. The three applications, where the HLS development time is larger than that of RTL, are from the same work [13]. The authors stated that the difference in development time was due to the time spent to learn to use the HLS tool and the need to modify the reference C++ source code to reach the required throughput.

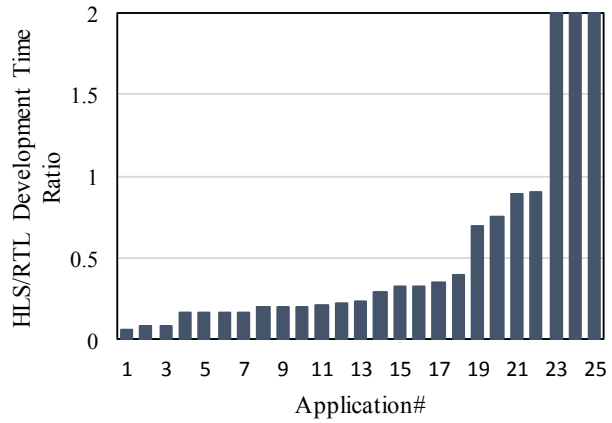


Fig. 4. HLS/RTL development time ratio for different applications.

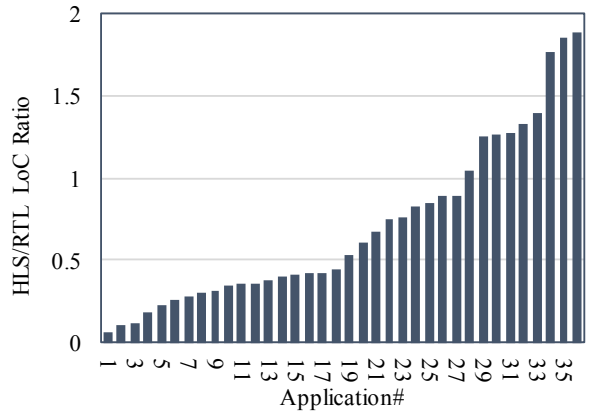
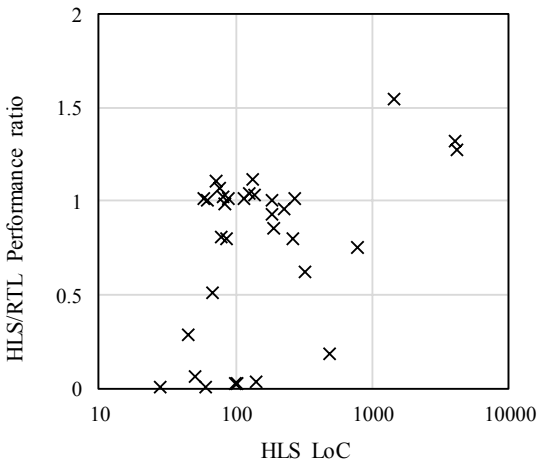


Fig. 5. HLS/RTL LoC ratio for different applications.

Similarly, Fig. 5 depicts the LoC ratio between HLS and RTL designs. Here, the HLS dominance is less prominent but still significant. In 75% of cases, the HLS LoC is smaller than RTL LoC.

We also investigated a possible correlation between the size of the application in LoC and HLS/RTL performance. Fig. 6(a) shows the data. When the three outliers in the top right corner are eliminated from calculations, the Pearson correlation coefficient is only 0.04. Thus, it seems that the size of the code is no indication for the relative HLS/RTL performance. Fig. 6(b) shows the same data for the relative HLS/RTL basic resource usage. The correlation is -0.08, so the code size does not correlate with the basic resource usage ratio either. Taken together, Figs. 3 and 6 indicate that the complexity of the application has no effect on the relative HLS to RTL performance or basic resource usage. However, as Fig. 6 shows, the majority of the applications presented in the papers are rather small in terms of LoC. Studying the respective behavior with larger applications is omitted due to the absence of data.

One way to look at the usefulness of HLS relative to RTL is to examine the performance obtained per design hour as discussed in [11]. Fig. 7 shows the relative productivity for all applications for which both performance and development time is reported, by dividing the HLS/RTL performance by the development time ratio. A value larger than one indicates that the HLS approach gives more performance per design hour than RTL. The average value is 4.4. RTL approach clearly wins in cases 1 to 4. The methodologies are about equal in cases 5 and 6, and HLS is the better approach in the remaining cases. For application 1, the bar is almost invisible, as the ratio is 0.05.



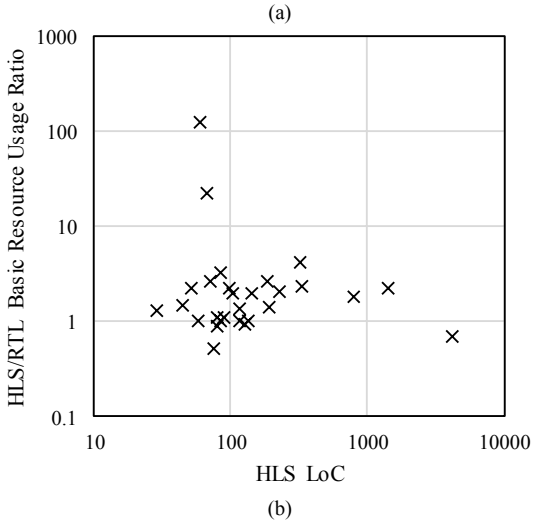


Fig. 6. HLS/RTL (a) performance ratio (b) basic resource usage ratio vs. HLS LoC.

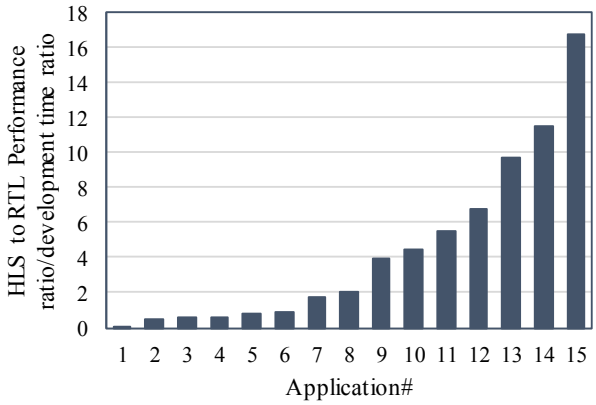


Fig. 7. Relative HLS to RTL productivity for different applications.

TABLE VI
BACKGROUND EXPERIENCE OF THE TEST GROUP

Person #	LoC (SW)	SW experience (months)	LoC (HW)	HW experience (months)
1	1k	18	1k	10
2	10k	3	1k	3
3	10k	18	1k	3
4	100k	50	1k	0
5	10k	3	1k	3
6	1k	1	1k	1

This application is a sparse algorithm matrix multiplication [11] with dynamic loop bounds, which are unsuitable for the automatic optimizations that HLS tools perform to speed-up computation. Despite that, the figure indicates that on average, a designer gets more performance per design hour with HLS tools.

V. TEST GROUP STUDY

This survey demonstrates that many prior works report the results of HLS to RTL comparisons rather inadequately, which also complicated our data collection. Therefore, we organized a case study to demonstrate the best practices in setting up appropriate tests for such comparisons and reporting the results. A secondary purpose of the case study was to examine how HLS and RTL flows differ from the user's perspective and what is the relative productivity of the flows. Most previous studies have focused only on the QoR differences instead.

The case study was to implement a 2-dimensional *discrete cosine transform* (DCT) [20] algorithm for 8 × 8 residual blocks used in the *High Efficiency Video Coding* (HEVC) [21] encoder. DCT was chosen because it is well known and of suitable complexity for this study.

A. Test Group

The test group consisted of six participants having basic knowledge on digital design and programming. As shown in Table VI, they had written 1k to 100k lines of C or C++ previously. On average, they had about 15 months of programming experience in work or hobby projects. The participants were much less experienced in hardware design with an average of 1k lines of VHDL or Verilog code and three months of experience in such projects. Only one of the participants had done a small tutorial with HLS before this study, making this experiment the first introduction to HLS for the rest.

We selected participants with limited hardware experience but moderate software experience, as HLS promises to hide away the hardware-specific implementation details. Thus, programmers who are used to writing behavioral descriptions in software projects are an ideal audience for HLS. Indeed, the litmus test of HLS is that such users reasonably effortlessly can produce acceptable results when designing relatively simple hardware blocks.

To acquire sufficient background knowledge of HLS, the participants self-studied the HLS basics and carried out five small exercises implementing parts of an audio codec for FPGA. Previously, they had done the same exercises using VHDL RTL.

B. Test Case

In an HEVC encoder, DCT is used to convert 8 × 8 spatial-domain residual blocks into 8 × 8 *transform-domain coefficient* (tcoeff) matrices. A well-known row-column algorithm [20] executes these 2-D transforms with separable 1-D transforms in two consecutive stages. The transform is first applied to each row of the residual block to generate an intermediate matrix and then to each column of the intermediate matrix to generate the final transform coefficient matrix.

The participants were assigned to implement this 2-D DCT hardware unit for 8 × 8 residual blocks with RTL (VHDL or Verilog) and HLS (C/C++ with Mentor Graphics' Catapult-C version 8.2m UV). Catapult-C supports the whole design flow from writing the original source code to generating and verifying the RTL code. In this study, no physical FPGA implementation was made, but only the synthesis results were used to obtain the QoR data. Performing place & route (P&R) was omitted as we were interested in the relative HLS to RTL results, and P&R should not affect the ratio significantly.

The provided DCT references included the HEVC specification and its implementation in the HEVC reference encoder (HM) [22]. The participants were also given a ready-made SystemC test bench and requirements for the interfaces to make the test bench work without modifications. Interface requirements included the widths of the input and output data buses and related control signals. The same test bench was used for the RTL and HLS versions. It generated random residual values for the first pass and performed the necessary transpose for the second pass. The condition for successful implementation was to pass the test bench validation.

The participants were also instructed to allocate their working hours to five categories: designing, implementing, searching information, simulating, and debugging. They were allowed to choose whether to implement the HLS or RTL version first or both simultaneously.

C. Results

Table VII shows the area and speed figures of the RTL and HLS implementations for the individual test persons. The HLS/RTL ratio shows the ratio between the results for HLS and RTL. The bolding indicates when the HLS flow achieved better results. The speed was calculated as million transform coefficients per second (Mtcoeff/s) using the output coefficients, latency, throughput, and frequency reported by the participants.

Four test persons started the work with the RTL implementation. All participants wrote the RTL code with VHDL rather than Verilog. Even though the smallest area and the highest frequency were achieved with RTL, the overall trend was that the participants were able to get slightly smaller area and slightly higher clock frequencies with the HLS tool. Furthermore, the HLS designs are over 2.5× as fast as the RTL designs, which also affected the speed to area ratios. For example, person #4 achieved the best speed to area ratio of all designs with HLS. On the other hand, person #3 was the only one who got better speed to area ratio with RTL. All test persons

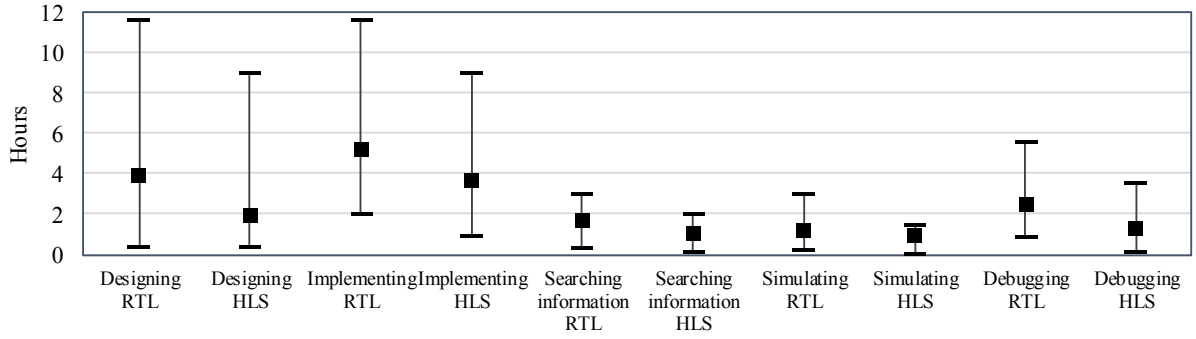


Fig. 8. Maximum, minimum, and average time usage for different categories with RTL and HLS.

TABLE VII
AREA AND PERFORMANCE FIGURES OF RTL AND HLS DESIGNS

HLS Person #	Area (LUTs)	Freq. (MHz)	Speed*	Speed/Area**	Started
1	1 860	214	258	139	
2	3 161	101	588	186	x
3	2 814	211	675	240	
4	2 273	167	972	427	x
5	2 768	137	797	288	
6	2 463	211	750	305	
Avg.	2 557	174	673	264	
RTL					
1	4 000	145	197	49	x
2	2 068	108	192	93	
3	1 292	308	458	355	x
4	4 431	148	499	113	
5	2 066	137	122	59	x
6	2 722	149	121	44	x
Avg.	2 763	166	265	119	
HLS/RTL ratio					
1	**0.47×**	**1.48×**	**1.31×**	**2.81×**	
2	1.53×	0.94×	**3.06×**	**2.00×**	
3	2.18×	0.69×	**1.47×**	0.68×	
4	**0.51×**	**1.13×**	**1.95×**	**3.80×**	
5	1.34×	1.00×	**6.55×**	**4.89×**	
6	**0.90×**	**1.42×**	**6.22×**	**6.87×**	
Avg.	**0.93×**	**1.05×**	**2.54×**	**2.22×**	

*Mtcoeffs/s **Mtcoeffs/kLUTs

TABLE VIII
HLS AND RTL PRODUCTIVITY

	HLS		RTL		HLS/RTL
Person #	Hours	Quality*/Hours	Hours	Quality*/Hours	Quality Ratio
1	2	80	4	14	5.9×
2	4	44	9	11	4.1×
3	12	19	21	17	1.1×
4	9	47	18	6	7.6×
5	5	60	9	6	9.4×
6	20	15	26	2	9.0×
Avg.	9	44	14	9	6.2×

*Mtcoeffs/kLUTs

used a multi-stage structure to calculate the DCT in RTL code, but none of them implemented a more complex state machine to use stage pipelining for consecutive inputs. The lack of pipelining lowered throughputs in the RTL case. In comparison, all persons were able to use loop unrolling and pipelining in HLS to achieve much better throughput values than with RTL.

Table VIII tabulates productivity values for HLS and RTL approaches. The productivity of all participants was clearly better with the HLS tool, and the average productivity of HLS was up to 6.0 times that of RTL. Hence, it is even higher than that found in the survey results. We can speculate how the productivity would have changed if the persons had implemented stage pipelining in their RTL implementations. It is still unlikely that the productivity levels had shifted to support RTL over HLS, as the time usage would have increased along with the throughput.

Fig. 8 shows the time usage of the participants in five categories. On average, the persons used less time within all categories when working with HLS. The grand total for maximum, average, and minimum time usages with the RTL flow was 37.7, 15.1 and 3.7 hours, respectively, whereas the respective values for the HLS flow were 25.0, 10.1, and 1.6 hours.

As a conclusion, all participants had better productivity with HLS than with RTL. Although the group size was small, and the hardware background of the persons was very similar, this study shows that it is easier to adopt HLS than RTL and receive better results faster for people who have most of their experience in software design. This result underlines the fact that HLS is a useful tool for software engineers who want to implement, for example, hardware accelerators.

It should be noted that our result differs from the typical surveyed study, where the QoR of RTL was better than that of HLS. The likely explanation for this is that in the surveyed works, the designers had significantly more previous hardware expertise than our test persons. On the other hand, our case study is in line with the surveyed literature concerning productivity, which favors HLS.

D. Feedback from the test persons

After completing the test assignments, the participants were asked about the pros and cons of HLS and RTL design flows, out of which they finally had to select their favorite. The answers were split evenly (3-3) between HLS and RTL flows.

The persons favoring RTL over HLS hoped for more open source support for HLS tools, as the flow is highly tool dependent. This would allow more hobbyists to use HLS tools.

Some test persons also longed for more control in the HLS tool over the resulting RTL in terms of cycle accuracy. For them, RTL was easier to fine tune and it gave them a better understanding of the problem at hand.

The persons favoring HLS over RTL liked the ease of HLS, where unnecessary details such as automatic I/O handshaking and pipelining support can be left as the responsibility of the HLS tool. This let the participants to focus on defining the behavioral description. They also felt that RTL was more time consuming, required more planning, and would have been harder to redesign.

The overall conclusion from the test persons was that HLS vs. RTL compares to C vs. assembly languages in embedded programming. They expressed the view that, a designer would rather use HLS, the highest level of abstraction available, and the lower level RTL should only be used in cycle critical applications or if it is able to provide a noticeable increase in performance.

E. Best Practices

We use our literature survey and this case study to sum up the following best practices for conducting comparative studies with RTL and HLS work flows:

1) A group of individuals should be used to implement the same design to lessen the impact of designer experience with the two flows.
2) The used HLS tools and languages should be reported unless license agreements prevent that. These choices have been shown to affect the QoR [12]-[14].
3) The same microarchitectures should be used in both RTL and HLS designs when conducting a study that concentrates on the QoR differences. If the emphasis is on productivity or the usability of HLS, however, then this restriction can be lifted.
4) For FPGA implementation, the exact FPGA chip model and version should be reported to allow replication of the results.
5) The time usage by each designer should be reported. Additionally, the time spent in each work phase should be reported to allow more insight into what parts are the most time consuming with HLS and RTL versions.
6) Lines of input code should be reported to show the size and complexity of the applications.
7) In addition to the basic QoR results, performance per design time should be reported to show the difference in productivity between the HLS and RTL flows.

VI. Closing the Quality Gap

Our survey shows that more often than not, there still is a QoR gap between the HLS and RTL methods for any given application, usually favoring RTL. A large amount of literature exists that has recognized the gap and proposes ways to close it. In this section, we present a survey of that literature to highlight it for the HLS researchers and developers. In addition, we review papers that introduce novel improvements to the existing HLS flows.

A. Research Directions for Tool Developers

The authors in [8] have several suggestions for the HLS tool developers, for where to focus their efforts. They note that resource sharing and scheduling are two major features in HLS techniques that the current HLS tools still struggle with. For example, they demonstrate that a HLS tool instantiates 31 hardware operators of a certain type when only 13 would be needed with optimal sharing. They also note that the HLS tools obfuscate the relationship between the source code and the generated hardware, which in turn makes it hard to identify the sub-optimal parts of the code. Furthermore, the authors call for the industry to agree on a standard C-based input language for HLS. This would allow an unambiguous way for the tool users and the tools themselves to interpret the source code.

In [57], the authors recognize room for development in both the usability and the QoR of HLS. Their study uses AutoPilot (now Vivado HLS), but the advice is generalizable. The authors propose automatic tradeoff analysis of loop pipelining and unrolling to make the DSE faster. With complicated loop structures, the number of possible optimization combinations can be very high. In addition, the authors call for support for BRAM port duplication directives, more robustness for dataflow transformations, and support for streaming computation for 2D access patterns. To improve the QoR, they suggest that the tools should detect memory level dependence between separate loops and functions, and automatically re-order memory access to allow partitioning, streaming, and better pipelining. The tools should also automatically create buffers to improve memory access reuse.

The importance of optimized memory accesses in high-quality designs is also recognized in [5]. The authors point out that the HLS tools usually do not have support for memory hierarchy nor do they abstract external memory accesses. Therefore, the designer is required to pay attention to the details of bus interfaces and memory controllers, which does not sit well with the idea of behavioral design paradigm. The HLS tools should hide external memory transfers from the designers to fix this problem. The paper also notices the difficulty of obtaining task-level parallelism from sequential C/C++ specifications, for which the authors suggest developing an appropriate device-neutral programming model.

In [32], the lack of support for dynamic data structures in HLS tools is brought forward. The authors implement the same algorithm with a data-flow centric way and by using recursive tree traversal, which uses dynamic memory allocation, and observe a significant performance reduction using the latter method. By applying several manual code transformations, the authors can increase the performance, and conclude that the HLS tools should automatically perform similar optimizations with dynamic data structures.

B. Improvements to the HLS Flow

Since the writers of research articles typically have no access to the source code of commercial HLS tools, most papers that have improved upon the HLS results do so by introducing new optimization steps to the design flow. Some promising results falling in this category are reviewed in this sub-section.

In [58], the authors propose using parallel pattern templates to scale module implementation according to the properties of the target device, exceeding the capability of the HLS tool to do so. The authors show up to 2.8× speed-up over a standard HLS tool flow. A template-based approach is also used in [59], where composable and parameterizable templates of common computation patterns optimized for hardware are used to improve performance. These kind of templates could be included in HLS tools for the users' convenience.

In [60], the problem of memory access bottleneck in massively parallel algorithms is discussed. The authors propose an algorithm that schedules the memory accessed during different pipeline stages reducing the simultaneous access pressure. Their approach improves the pipelining performance by 43% on average and reduces memory bank usage by 55%. Another way to reduce memory access overhead is discussed in [61]. The paper presents a novel algorithm to scalarize arrays selectively to on-chip registers within certain area constraints. The results indicate significant performance improvements.

One method to enable more efficient HLS is by identifying custom operations that are merged from sequential basic operations. This reduces the complexity of the data flow graph of the synthesized algorithm, which in turn reduces synthesis runtime and improves the QoR. This method has been studied in [62], achieving significant improvements in area consumption, performance, and code size. Therefore, HLS tools should include custom operation identification as a pre-processing step.

Resource allocation and operation binding are two of the basic steps in HLS. Thus, their efficient implementation is of critical importance in achieving good QoR. In [63], the effect of register allocation has been investigated. The paper shows that in most cases a naïve resource allocation strategy, i.e. one register per variable without register sharing leads to the best QoR results.

HLS tools use a software compiler to create an intermediary representation (IR) of the input program. The IR is then used in the HLS optimization steps. It is not surprising that the IR and thus the compiler options affect the QoR. The authors in [64] have studied the effect of different compiler options on the QoR and developed a method to automatically select only those options that improve the QoR, achieving on average a 16% better performance compared to the usual –O3 optimization level.

In [65], it is observed that significant area savings can be achieved by merging different behavioral descriptions instead of performing HLS for each of them separately. This is due to allowing better resource sharing of functional units on FPGA when the HLS tool can share them between descriptions. The paper presents an algorithm for searching for optimal mergings within given latency constraints.

C. Design Space Exploration

The HLS tools contain various directives that can guide the hardware synthesis to generate designs that are more efficient. These directives include pipelining and unrolling of loops and array partitioning among others. Since most algorithms contain numerous loops and data arrays, finding the group of Pareto optimal directive settings can be a daunting task, yet it is essential for good QoR. Exploring the design space for optimal settings should therefore be automated, but currently the leading HLS tools do not help the user in DSE. On the other hand, there are a few academic papers that have studied the DSE automation in HLS.

A straightforward automated iterative DSE methodology is presented in [66]. The method, which focuses on area reduction, achieves up to 50% improvement in the QoR when compared to non-guided HLS flow. A more complicated DSE algorithm, based on an adaptive windowing method, is shown in [67]. This algorithm is shown to offer a good trade-off between running time and finding the best QoR. A similar approach specializing on applications with nested loops has demonstrated up to 235× speed-up compared to exhaustive DSE, while achieving similar results [68].

A sequential model-based optimization has been applied to the DSE problem in [69]. The paper shows that the method can find globally optimal points from a space of tens of thousands of possible designs in reasonable time. In [70], a lightweight pre-processing step has been added before HLS to perform dynamic dependence analysis of the target algorithm. The method can expose resource sharing opportunities that result in better QoR when they are given as constraints to the HLS tool.

The specific but important question of finding the optimal loop unrolling factor has been discussed in [71]. The authors have developed an algorithm for finding the optimal unrolling factor within given area constraint and show that it can provide the best performance compared to other possible solutions.

D. Verification

Verification remains a time-consuming part of any design project. Therefore, it is crucial that the HLS tools support the verification flow on all stages. While the HLS flow allows for efficient behavioral verification of single modules, the generated RTL must still be verified for non-behavioral aspects such as interface synthesis results and successful component integration. Traditional RTL verification after HLS is difficult since there is no direct relationship to the input source code [4], [5], [8]. Nevertheless, verification time has been halved in many cases using HLS [72].

The verification aspects of HLS have been extensively discussed in a recent paper [72]. The author points out that logic redundancy, which lowers test coverage, is a major problem with HLS. Logic redundancies may be present in the source specification but also introduced by the HLS tool in the RTL generation. Thus, the developers of the HLS tools should strive to eliminate the tendency to generate logic redundancy. Besides that, formal tools can be used to identify the redundancies during verification. The paper also promotes source linting as a way to improve HLS. Not only can it be used to check for error sources, but also to help with the design optimization by proving properties such as FIFO sizes.

The authors in [5] present three noteworthy items to enable most of the debugging to occur on the behavioral input language level for on-chip validation: 1) the ability to add

debugging logic with small overhead, 2) the ability to observe critical buffers such as FIFOs, and 3) the ability to observe the internal states of hardware blocks using breakpoints in the source code. These important debugging features cannot be implemented on the RTL level after performing HLS because of the machine-generated RTL code.

Besides verification, engineering change orders (ECOs) present difficulties with HLS [4]. When an ECO is issued, only some small incremental changes are required, which are typically not captured by the high-level behavioral description. On the other hand, it has been noted that since the behavioral source code can be extensively verified and the HLS tools ensure that the generated RTL is correct, ECOs are uncommon in HLS flows [66].

VII. Conclusion

In this paper, we examined 46 recent articles about comparisons on the QoR and design effort between HLS and RTL design flows. As HLS promises great productivity gains over RTL, our aim was to see whether the contemporary HLS tools are also able to produce results that can compete with hand tuned RTL designs.

Our survey indicates that even the newest generation of HLS tools does not provide as good performance and resource usage as manual RTL does. However, there is a great variance in the results and HLS is shown to equal or outperform RTL approach in about 40% of the evaluated cases. Our own case study demonstrates that a designer with limited hardware experience can obtain better results with HLS, with 2.5 times more performance and slightly lower FPGA resource usage. We also examined whether the size of the design affected the relative QoR between HLS and RTL, but found no correlation. Thus, HLS seems as suited for small as large designs.

In design effort, the survey showed that HLS was clearly the frontrunner as expected. On average, the HLS design time was only a third of the corresponding RTL design time. In addition, the size of the HLS input code was almost halved, being 52% of the RTL code size on average. When taking into account both the QoR and the design effort, we found out that a designer gets on average 4.4 times as high performance per design hour using HLS than RTL. Our own case study supported this argument by reporting 6.0 times increase in productivity. Thus, HLS is a particularly good choice when time to market is a dominant issue and there is no compelling need to gain the ultimate performance or smallest resource usage for the product. HLS also offers tremendous time savings when architectural changes are made to an existing design.

In our reference literature, there was often lacking information, which made the HLS to RTL comparisons more challenging. Therefore, our case study also demonstrated the best practices in reporting HLS and RTL results for the same application. Preferably, the test group should be larger than we had at our disposal, but our test case still shows the essential details that we recommend reporting in this kind of research. In the future, a similar case study could be carried out with a test group with more hardware expertise. While our study shows that people with limited hardware experience can easily adopt HLS and produce good results, it would also be interesting to see how the productivity and QoR differences behaved with hardware engineers as test persons.

Verification effort comparisons were also often missing from the surveyed papers. Most often, there was only a brief note on how HLS tools allow convenient use of behavioral test bench in RTL verification. As verification is a major part of any hardware project, this is a significant oversight in the state of HLS research. Therefore, in the future, more quantitative studies should be carried out on HLS vs. RTL verification flows.

We also surveyed the literature for both suggestions and completed research for improving the QoR and verification flow of HLS. We found numerous papers that showed methods to improve the QoR significantly by adding new steps to the HLS design flow or by automating the design space exploration.

With the progress achieved in HLS tools during the last decade, we can conclude that the methodology is ready for adoption by the industry in prototyping and fast product development. If the next generation of HLS tools can close the QoR gap entirely, then HLS will become the new standard design method, and RTL can be targeted at similar limited fine-tuning as assembly languages in software development today.

Appendix

TABLE IX
Summary of the Reviewed Papers

[#]	Year	HLS tools	Modules or algorithms	Number of applications	LoC	Dev. time	Performance	Basic FPGA Resources
[5]	2011	AutoPilot	Multi-I/O sphere decoder	1		x		x
[8]	2016	Undisclosed	AES encryption	1	x		x	
[9]	2014	Catapult-C	K-means accelerator, histogram map/reduce, matrix mult., word count	5			x	x
[10]	2016	Vivado	Data pinning, step row filter, Sobel filter	3		x	x	x
[11]	2013	Vivado	Matrix multiplication	3		x	x	x
[12]	2015	Vivado, LegUp, Simulink HDL	HEVC 2D IDCT	3			x	x

TABLE IX CONT.

[13]	2014	Vivado, Catapult-C	Predictive block-based motion estimation	8	x	x	x	x
[14]	2014	Altera OpenCL, BlueSpec, Chisel, LegUp	Bitonic sorter, spatial sorter, median operator, hash join	16	x		x	x
[15]	2016	Cadence C-to-Silicon	Semi-global matching algorithm for stereo vision	4			x	x
[16]	2011	Xilinx System Generator for DSP, Simulink HDL	Hybrid RAKE receiver for DS-UWB communication	2				x
[17]	2011	Xilinx System Generator for DSP, Simulink HDL	Adaptive impulsive noise filtering	3			x	x
[18]	2014	Vivado	128-bit key AES-CTR cryptography	6			x	x
[23]	2010	BlueSpec, unspecified	Reed-Solomon decoder for 802.16 protocol receiver	2			x	x
[24]	2010	ROCCC 2.0	Ten small separate applications	10	x	x	x	x
[25]	2010	Impulse-C	Computed tomography filtered backprojection	2		x	x	
[26]	2010	AccelDSP	Image preprocessing coprocessor	1		x	x	x
[27]	2011	HCE	Reverse time migration for solving a wave equation	1	x	x	x	
[28]	2011	AutoESL AutoPilot	Sphere detector for spatial multiplexing MIMO	1		x		x
[29]	2012	Cadence C-to-Silicon	Hardware-based run-time monitors	4	x		x	
[30]	2012	HCE	GRN generator for Brownian motion simulation	1		x	x	x
[31]	2012	Chisel	3-stage 32-bit RISC processor	2	x			
[32]	2013	Vivado	K means clustering	2			x	x
[33]	2013	Vivado	Operating system scheduler	1			x	x
[34]	2013	Vivado	Buck converter closed-loop control	1			x	x
[35]	2013	Catapult-C	IEEE 802.15.4 physical layer for SW defined radio	1				x
[36]	2013	MaxCompiler	LDPC decoder	1			x	
[37]	2013	Vivado	Skin detection image processing system	1				x
[38]	2013	Vivado	Convolution, background subtraction	2		x	x	x
[39]	2014	Altera OpenCL, Vivado	Tri-diagonal linear system solver	2			x	x
[40]	2014	Vivado	10 Gb/s network flow monitor	1	x	x		x
[41]	2014	Undisclosed	Floating-point unit co-processor	3			x	
[42]	2014	CAPH Toolset	MPEG encoder core	1	x		x	x
[43]	2014	Vivado	Memcached key-value store	1	x		x	x
[44]	2014	CAPH Toolset	Histograms of oriented gradients	1	x			x
[45]	2015	Undisclosed	Microwave imaging via space-time beamforming	1			x	
[46]	2015	LegUp	MIPS ISA processor	2			x	x
[47]	2015	Convey Hybrid-Threading	Sobel edge detector, breadth-first search, Smith-Waterman sequence alignment	3	x	x	x	x
[48]	2015	Altera OpenCL	Canny edge detector, Sobel edge detector, SURF feature extraction	6		x	x	x
[49]	2015	HIPAcc & Vivado	Stereo vision block matching	1			x	x
[50]	2015	HIPAcc & Vivado	Stereo block matching	2			x	x
[51]	2015	Vivado	Non-binary LDPC decoder	1			x	
[52]	2016	Cadence Stratus HLS	Direct memory access controller	1	x	x	x	
[53]	2016	Vivado	HEVC sub-pixel interpolation	1			x	x
[54]	2016	Vivado	HEVC intra prediction	1			x	x
[55]	2016	C2RTL	Bilateral filter for image denoising	1			x	x
[56]	2016	Vivado	SURF algorithm	1			x	x

The "x" marks denote what information was given in the papers

REFERENCES

[1] "International technology roadmap for semiconductors, 2011 edition, design," ITRS. 2011. [Online] Available: https://www.dropbox.com/sh/r51qrus06k6ehrc/AACQYSRnTdLGUCDZFhB6_iXua/2011Chapters?dl=0&preview=2011Design.pdf

[2] G. Martin and G. Smith, "High-level synthesis: past, present, and future," *IEEE Des. Test. Comput.*, vol. 26, no. 4, pp. 18-25, Jul. 2009.

[3] P. Coussy, D. D. Gajski, M. Meredith, and A. Takach, "An introduction to high-level synthesis," *IEEE Des. Test. Comput.*, vol. 26, no. 4, pp. 8-17, Jul. 2009.

[4] H. Ren, "A brief introduction on contemporary high-level synthesis," in *2014 IEEE Int. Conf. IC Design & Technology*, Austin, TX, pp. 1-4.

[5] J. Cong *et al.*, "High-Level Synthesis for FPGAs: From Prototyping to Deployment," *IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst.*, vol. 30, no. 4, pp. 473-491, Apr. 2011.

[6] S. Windh *et al.*, "High-level language tools for reconfigurable computing," *Proc. IEEE*, vol. 103, no. 3, pp. 390-408, Mar. 2015.

[7] R. Nane *et al.*, "A survey and evaluation of FPGA high-level synthesis tools," *IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst.*, vol. 35, no. 10, pp. 1591-1604, Oct. 2016.

[8] Z. Sun, K. Campbell, and W. Zuo, "Designing high-quality hardware on a development effort budget: A study of the current state of high-level synthesis," in *2016 21st Asia and South Pacific Design Automation Conf.*, Macao, pp. 218-225.

[9] M. Sharafeddin *et al.*, "On the efficiency of automatically generated accelerators for reconfigurable active SSDs," in *2014 26th Int. Conf. Microelectronics*, Doha, pp. 124-127.

[10] A. Cortes, I. Velez, and A. Irizar, "High level synthesis using Vivado HLS for Zynq SoC: Image processing case studies," in *2016 Conf. Design Circuits and Integrated Systems*, Granada, pp. 1-6.

[11] S. Skalicky, C. Wood, M. Lukowiak, and M. Ryan, "High level synthesis: Where are we? A case study on matrix multiplication," in *2013 Int. Conf. Reconfigurable Computing and FPGAs*, Cancun, pp. 1-7.

[12] E. Kalali and I. Hamzaoglu, "FPGA implementations of HEVC inverse DCT using high-level synthesis," in *2015 Conf. Design and Architectures Signal and Image Processing*, Krakow.

[13] G. Schewior, C. Zahl, and H. Blume, "HLS-based FPGA implementation of a predictive block-based motion estimation algorithm — A field report," 2014 Conf. Design and Architectures Signal and Image Processing, Madrid, pp. 1-8.

[14] O. Arcas-Abella *et al.*, "An empirical evaluation of High-Level Synthesis languages and tools for database acceleration," in *2014 24th Int. Conf. Field Programmable Logic and Applications*, Munich, pp. 1-8.

[15] A. Qamar, F. Muslim, F. Gregoretti, L. Lavagno, and M. T. Lazarescu, "High-level Synthesis for Semi-global Matching: Is the juice worth the squeeze?," *IEEE Access*, vol. 4, no. 99, Dec. 2016.

[16] C. Thomos and G. Kalivas, "FPGA-based architecture and implementation techniques of a low-complexity hybrid RAKE receiver for a DS-UWB communication system," *Telecommunication Systems*, vol. 52, no. 4, pp. 2115-2132, Apr. 2013.

[17] A. Rosado-Munoz, M. Bataller-Mompean, E. Soria-Olivas, C. Scarante, and J. F. Guerrero-Martinez, "FPGA Implementation of an Adaptive Filter Robust to Impulsive Noise: Two Approaches," *IEEE Trans. Ind. Electron.*, vol. 58, no. 3, pp. 860-870, Mar. 2011.

[18] E. Homsirikamol and K. Gaj, "Can high-level synthesis compete against a hand-written code in the cryptographic domain? A case study," in *2014 Int. Conf. Reconfigurable Computing and FPGAs*, Cancun, pp. 1-8.

[19] D. F. Bacon, R. Rabbah, S. Shukla, "FPGA Programming for the masses," *ACM Queue*, vol. 11, no. 2, Feb. 2013.

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

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

[22] Joint Collaborative Team on Video Coding Reference Software, ver. HM 16.3 [Online]. Available: http://hevc.hhi.fraunhofer.de/

[23] A. Agarwal, M. C. Ng, and Arvind, "A Comparative Evaluation of High-Level Hardware Synthesis Using Reed–Solomon Decoder," *IEEE Embedded Systems Letters*, vol. 2, no. 3, pp. 72-76, Sep. 2010.

[24] J. Villarreal, A. Park, W. Najjar, and R. Halstead, "Designing Modular Hardware Accelerators in C with ROCCC 2.0," in *2010 18th IEEE Annu. Int. Symp. Field-Programmable Custom Computing Machines*, Charlotte, NC, pp. 127-134.

[25] J. Xu, N. Subramanian, A. Alessio, and S. Hauck, "Impulse C vs. VHDL for Accelerating Tomographic Reconstruction," in *2010 18th IEEE Annu. Int. Symp. Field-Programmable Custom Computing Machines*, Charlotte, NC, pp. 171-174.

[26] M. Samarawickrama, R. Rodrigo, and A. Pasqual, "HLS approach in designing FPGA-based custom coprocessor for image preprocessing," in *2010 5th Int. Conf. Information and Automation Sustainability*, Colombo, pp. 167-171.

[27] T. Hussain, M. Pericás, N. Navarro, and E. Ayguadé, "Implementation of a Reverse Time Migration kernel using the HCE High Level Synthesis tool," in *2011 Int. Conf. Field-Programmable Technology, New Delhi*, pp. 1-8.

[28] J. Noguera *et al.*, "Implementation of sphere decoder for MIMO-OFDM on FPGAs using high-level synthesis tools," *Analog Integrated Circuits and Signal Processing*, vol. 69, no. 2-3, Sep. 2011.

[29] M. Ismail and G. E. Suh, "Fast development of hardware-based run-time monitors through architecture framework and high-level synthesis," in *2012 IEEE 30th Int. Conf. Computer Design*, Montreal, QB, pp. 393-400.

[30] J. S. Malik, P. Palazzari, A. Hemani, "Effort, resources, and abstraction vs performance in high-level synthesis, finding new answers to an old question," *ACM SIGARCH Computer Architecture News*, vol. 40, no. 5, pp. 64-69, Mar. 2012.

[31] J. Bachrach, H. Vo, and B. Richards, "Chisel: Constructing hardware in a Scala embedded language," in *2012 49th ACM/EDA/IEEE Design Automation Conf.*, San Francisco, CA, pp. 1212-1221.

[32] F. Winterstein, S. Bayliss, and G. A. Constantinides, "High-level synthesis of dynamic data structures: A case study using Vivado HLS," in *2013 Int. Conf. Field-Programmable Technology*, Kyoto, pp. 362-365.

[33] J. Dahlstrom, and S. Taylor, "Migrating an OS Scheduler into Tightly Coupled FPGA Logic to Increase Attacker Workload," in *2013 IEEE Military Communications Conf.*, San Diego, CA, pp. 986-991.

[34] D. Navarro, O. Lucia, L.A. Barragan, I. Urriza, and J. I. Artigas, "Teaching digital electronics courses using high-level synthesis tools," in *2013 7th IEEE Int. Conf. e-Learning Industrial Electronics*, Vienna, pp. 43-47.

[35] V. Bhatnagar, G. S. Ouedraogo, M. Gautier, A. Carer, and O. Sentieys, "An FPGA Software Defined Radio Platform with a High-Level Synthesis Design Flow," in *2013 IEEE 77th Vehicular Technology Conf.*, Dresden, pp. 1-5.

[36] F. Pratas, J. Andrade, G. Falcao, V. Silva, and L. Sousa, "Open the Gates: Using High-level Synthesis towards programmable LDPC decoders on FPGAs," in *2013 IEEE Global Conf. Signal and Information Processing*, Austin, TX, pp. 1274-1277.

[37] D. O'Loughlin, A. Coffey, F. Callaly, D. Lyons, and F. Morgan, "Xilinx Vivado High Level Synthesis: Case studies," in *25th IET Irish Signals & Systems Conf. 2014 and 2014 China-Ireland Int. Conf. Information and Communications Technologies*, Limerick, pp. 352-356.

[38] J. Hiraiwa and H. Amano, "An FPGA Implementation of Reconfigurable Real-Time Vision Architecture," in *2013 27th Int. Conf. Advanced Information Networking and Applications Workshops*, Barcelona, pp. 150-155.

[39] D.J. Warne, N.A. Kelson, and R.F. Hayward, "Comparison of High Level FPGA Hardware Design for Solving Tri-diagonal Linear Systems," *Procedia Computer Science*, vol. 29, pp. 95-101, 2014.

[40] M. Forconesi, G. Sutter, S. Lopez-Buedo, J. E. Lopez de Vergara, and J. Aracil, "Bridging the gap between hardware and software open source network developments," *IEEE Network*, vol. 28, no. 5, pp. 13-19, Sep. 2014.

[41] C-I. Chen, C-Y. Yu, Y-J. Lu, and C-F. Wu, "Apply high-level synthesis design and verification methodology on floating-point unit implementation," in *2014 Int. Symp. VLSI Design, Automation and Test*, Hsinchu.

[42] J. Sérot and F. Berry, "High-Level Dataflow Programming for Reconfigurable Computing," in *2014 Int. Symp. Computer Architecture and High Performance Computing Workshop*, Paris, pp. 72-77.

[43] K. Karras, M. Blott, and K. Vissers, "High-Level Synthesis Case Study: Implementation of a Memcached Server," in *1st Int. Workshop FPGAs Software Programmers*, Munich, 2014, pp. 77-82.

[44] J. Sérot, F. Berry, and C. Bourrasset, "High-level dataflow programming for real-time image processing on smart cameras," *Journal of Real-Time Image Processing*, vol. 12, no. 4, pp. 635-647, Dec. 2016.

[45] D. J. Pagliari, M. R. Casu, and L. P. Carloni, "Acceleration of microwave imaging algorithms for breast cancer detection via High-Level Synthesis," in *2015 33rd IEEE Int. Conf. Computer Design*, New York, NY, pp. 475-478.

[46] T. Ahmed, N. Sakamoto, and J. Anderson, "Synthesizable-from-C Embedded Processor Based on MIPS-ISA and OISC," in *2015 IEEE 13th Int. Conf. Embedded and Ubiquitous Computing*, Porto, pp. 114-123.

[47] G. Wang, H. Lam, and A George, "Performance and productivity evaluation of hybrid-threading HLS versus HDLs," in *2015 IEEE High Performance Extreme Computing Conf.*, Waltham, MA.

[48] K. Hill, S. Craciun, A. George, and H. Lam, "Comparative analysis of OpenCL vs. HDL with image-processing kernels on Stratix-V FPGA," in *2015 IEEE 26th Int. Conf. Application-specific Systems, Architectures and Processors*, Toronto, ON, pp. 189-193.

[49] M. Schmid, O. Reiche, F. Hannig, and J. Teich, "Loop coarsening in C-based High-Level Synthesis," in *2015 IEEE 26th Int Conf. Application-Specific Systems, Architectures and Processors*, Toronto, ON, pp. 166-173.

[50] O. Reiche *et al.*, "Automatic Optimization of Hardware Accelerators for Image Processing," in *DATE Friday Workshop Heterogeneous Architectures and Design Methods for Embedded Image Systems*, Grenoble, 2015, pp. 10-15.

[51] J. Andrade, N. George, and K. Karras, "From low-architectural expertise up to high-throughput non-binary LDPC decoders: Optimization guidelines using high-level synthesis," in *2015 25th Int. Conf. Field Programmable Logic and Applications*, London, pp. 1-8.

[52] Q. Zhu and M. Tatsuoka, "High quality IP design using high-level synthesis design flow," in *2016 21st Asia and South Pacific Design Automation Conf.*, Macao, pp. 212-217.

[53] F.A Ghani, E. Kalali, and I. Hamzaoglu, "FPGA implementations of HEVC sub-pixel interpolation using high-level synthesis," in *2016 Int. Conf. Design and Technology of Integrated Systems Nanoscale Era*, Istanbul.

[54] E. Kalali, I. Hamzaoglu, "FPGA implementation of HEVC intra prediction using high-level synthesis," in *2016 IEEE 6th Int. Conf. Consumer Electronics*, Berlin, pp. 163-166.
[55] I. Endo, T. Isshiki, D. Li, and H. Kunieda, "A design method for real-time image denoising circuit using High-Level Synthesis," in *2016 7th Int. Conf. Information and Communication Technology Embedded Systems*, Bangkok, pp. 30-35.
[56] W. Chen *et al.*, "FPGA-Based Parallel Implementation of SURF Algorithm," in *2016 IEEE 22nd Int. Conf. Parallel and Distributed Systems*, Wuhan, pp. 308-315.
[57] K. Rupnow, Y. Liang, Y. Li, and D. Chen, "A study of high-level synthesis: Promises and challenges," in *2011 9th IEEE Int. Conf. ASIC*, Xiamen, pp. 1102-1105.
[58] L. Josipovic, N. George, and P. Ienne, "Enriching C-based High-Level Synthesis with parallel pattern templates," in *2016 Int. Conf. Field-Programmable Technology*, Xi'an, 2016, pp. 177-180.
[59] J. Matai, D. Lee, A. Althoff, and R. Kastner, "Composable, parameterizable templates for high-level synthesis," in *2016 Design, Automation & Test Europe Conference & Exhibition*, Dresden, pp. 744-749.
[60] T. Lu, S. Yin, X. Yao, Z. Xie, L. Liu, and S. Wei, "Memory fartitioning-based modulo scheduling for high-level synthesis," in *2017 IEEE Int. Symp. Circuits and Systems (ISCAS)*, Baltimore, MD, pp. 1-4.
[61] P. R. Panda, N. Sharma, A. K. Pilania, G. Krishnaiah, S. Subramoney, and A. Jagannathan, "Array scalarization in high level synthesis," in *2014 19th Asia and South Pacific Design Automation Conf.*, Singapore, pp. 622-627.
[62] C. Xiao and E. Casseau, "Improving high-level synthesis effectiveness through custom operator identification," in *2014 IEEE Int. Symp. Circuits and Systems*, Melbourne VIC, pp. 161-164.
[63] G. Hempel, J. Hoyer, T. Pionteck, and C. Hochberger, "Register allocation for high-level synthesis of hardware accelerators targeting FPGAs," in *2013 8th Int. Workshop Reconfigurable and Communication-Centric Systems-on-Chip*, Darmstadt, 2013, pp. 1-6.
[64] Q. Huang *et al*., "The Effect of Compiler Optimizations on High-Level Synthesis for FPGAs," in *2013 IEEE 21st Annual Int. Symp. Field-Programmable Custom Computing Machines*, Seattle, WA, pp. 89-96.
[65] B. Carrion Schafer, "Process selection for maximum resource sharing in High-Level Synthesis," in *2015 Electronic System Level Synthesis Conf.*, San Francisco, CA, pp. 35-40.
[66] J. S. da Silva and S. Bampi, "Area-oriented iterative method for Design Space Exploration with High-Level Synthesis," in *2015 IEEE 6th Latin American Symp. Circuits & Systems*, Montevideo, pp. 1-4.
[67] D. Liu and B. C. Schafer, "Efficient and reliable High-Level Synthesis Design Space Explorer for FPGAs," in *2016 26th Int. Conf. Field Programmable Logic and Applications*, Lausanne, pp. 1-8.
[68] G. Zhong, V. Venkataramani, Y. Liang, T. Mitra, and S. Niar, "Design space exploration of multiple loops on FPGAs using high level synthesis," in *2014 IEEE 32nd Int. Conf. Computer Design*, Seoul, pp. 456-463.
[69] C. Lo and P. Chow, "Model-based optimization of High Level Synthesis directives," in *2016 26th Int. Conf. Field Programmable Logic and Applications*, Lausanne, pp. 1-10.
[70] R. Garibotti, B. Reagen, Y. S. Shao, G. Y. Wei, and D. Brooks, "Using dynamic dependence analysis to improve the quality of high-level synthesis designs," in *2017 IEEE Int. Symp. Circuits and Systems*, Baltimore, MD, pp. 1-4.
[71] R. Nane, V. M. Sima, and K. Bertels, "Area constraint propagation in high level synthesis," in *2012 Int. Conf. Field-Programmable Technology*, Seoul, pp. 247-252.
[72] A. Takach, "High-level Synthesis: Status, Trends, and Future Directions," *IEEE Design & Test*, vol. 33, no. 3, pp. 116-124, Jun. 2016.

Sakari Lahti (S'16) received the M.Sc. degree in engineering physics from the Tampere University of Technology (TUT), Tampere, Finland in 2002 and in computer engineering in 2014.

From 2000 to 2002, he was a Research Assistant with the Department of Physics, TUT, and from 2012 to 2014 with the Department of Pervasive Computing, TUT. Since 2015, he has been a doctoral student at the Laboratory of Pervasive Computing, TUT, where he works as a university teacher. His research interests include high-level synthesis on FPGA, and hardware and system-on-chip designing.

Panu Sjövall (S'17) received the M.Sc degree in automation engineering from the Tampere University of Technology (TUT), Tampere, Finland in 2015.

He was a Research Assistant with the Department of Pervasive Computing in TUT from 2014 to 2016, and briefly a Project Researcher with the Department of Pervasive Computing in 2016. Since 2016, he has been a doctoral student at the Laboratory of Pervasive Computing, TUT. His research interests include hardware and system-on-a-chip designing, high-level synthesis, FPGAs, Linux kernel driver development and video encoding.

Jarno Vanne (M'02) received the M.Sc. degree in information technology and the Ph.D. degree in computing and electrical engineering from the Tampere University of Technology (TUT), Tampere, Finland, in 2002 and 2011, respectively.

He is currently an Assistant Professor in the Laboratory of Pervasive Computing, TUT. He leads the Ultra Video Group that develops open-source Kvazaar HEVC encoder and related multimedia applications on various computing platforms ranging from low-power embedded devices to highly distributed cloud environments. His special research interests include real-time HEVC coding, 3D/360 video coding for virtual and augmented reality, future video coding standards, intelligent video compression, and high-level synthesis.

Timo D. Hämäläinen (M'95) received the M.Sc. and Ph.D. degrees in electrical engineering from the Tampere University of Technology (TUT), Tampere, Finland, in 1993 and 1997, respectively.

He is currently a full professor and head of the Laboratory of Pervasive Computing in TUT. He is an author of more than 70 journals and 240 conference publications. He holds several patents. His current research interests include model based design methods for multiprocessor systems-on-chip, Kactus2 open source IP-XACT based system design tool with new visualization paradigms and FPGA accelerated cloud computing for video processing.

PUBLICATION V

Kvazaar 4K HEVC intra encoder on FPGA accelerated air-frame server

P. Sjövall, V. Viitamäki, A. Oinonen, J. Vanne, T. D. Hämäläinen, and A. Kulmala

In *Proceedings of IEEE International Workshop on Signal Processing Systems*, Lorient, France, Oct. 2017
DOI: 10.1109/SiPS.2017.8109999

Kvazaar 4K HEVC Intra Encoder on FPGA Accelerated Airframe Server

Panu Sjövall, Vili Viitamäki, Arto Oinonen, Jarno Vanne, Timo D. Hämäläinen
Laboratory of Pervasive Computing
Tampere University of Technology
Tampere, Finland

Ari Kulmala
Accelerator SoC R&D
Nokia
Tampere, Finland

***Abstract*— This paper presents a real-time Kvazaar HEVC intra encoder for 4K Ultra HD video streaming. The encoder is implemented on Nokia AirFrame Cloud Server featuring a 2.4 GHz dual 14-core Intel Xeon processor and Arria 10 PCI Express FPGA accelerator card. In our HW/SW partitioning scheme, the data-intensive Kvazaar coding tools including intra prediction, DCT, inverse DCT, quantization, and inverse quantization are offloaded to Arria 10 whereas CABAC coding and other control-intensive coding tools are executed on Xeon processors. Arria 10 has enough capacity for up to two instances of our intra coding accelerator. The results show that the proposed system is able to encode 4K video at 30 fps with a single intra coding accelerator and at 40 fps with two accelerators. The respective speed-up factors are 1.6 and 2.1 over the pure Xeon implementation. To the best of our knowledge, this is the first work dealing with HEVC intra encoder partitioned between CPU and FPGA. It achieves the same coding speed as HEVC intra encoders on ASIC and it is at least 4 times faster than existing HEVC intra encoders on FPGA.**

Keywords*— High Efficiency Video Coding (HEVC), Kvazaar, Intra coding, Field-programmable gate array (FPGA), PCI Express (PCIe), Real-time

I. 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 capacities, the holistic increase of video volume makes more efficient video compression inevitable.

High Efficiency Video Coding (*HEVC*/*H.265*) [2], [3] is the latest international video coding standard developed to meet video storage and transmission needs of modern multimedia applications. HEVC is published as twin text by ITU, ISO, and IEC as ITU-T H.265 | ISO/IEC 23008-2. This paper addresses *all-intra* (*AI*) coding configuration [4] of HEVC Main Profile. HEVC is shown to improve intra coding efficiency by 23% over that of the preceding state-of-the-art standard AVC/H.264 [5] for the same objective quality but at a cost of over 3 × encoding complexity [6]. Therefore, implementing a real-time HEVC intra encoder with a reasonable coding efficiency, implementation cost, and power budget requires efficient encoder optimizations and powerful computing platforms.

The complexity of *software* (*SW*) HEVC encoders can be primarily tackled by two techniques: multithreading through data-level parallelism [7], [8] and *single instruction multiple data* (*SIMD*) optimizations [9], [10]. Further speedup and lower power dissipation can be obtained by offloading the compute-intensive coding tools to *hardware (HW)* accelerators or implementing the entire HEVC encoder on HW [11]-[14]. Existing HW encoders include both *application specific integrated circuit (ASIC)* [11], [12] and *field-programmable gate array* (*FPGA*) implementations [12]-[14].

The main motivation of this work was to optimize our Kvazaar HEVC intra encoder [15], [16], for real-time 4K *Ultra High Definition* (*UHD*) coding on Nokia AirFrame Cloud Server. Airframe includes a 2.4 GHz dual 14-core Xeon processor an Arria 10 *PCI Express* (*PCIe*) FPGA accelerator card. Airframe rackmount server is easily expandable to large server farms and an accompanied FPGA brings lots of additional computing power for a single server. Cloud video encoding on servers like AirFrame has gained a lot of traction in the recent years because of the advent of cloud gaming, telco clouds, and edge computation in general.

Our previous works have already investigated parallelization of Kvazaar intra encoder on multi-core processors [8] and SIMD optimizations of Kvazaar [10], so the main emphasis here is on 1) HW/SW partitioning of Kvazaar; and 2) HW acceleration of Kvazaar on FPGA. The HW-oriented C source code of Kvazaar enables more straightforward HW/SW partitioning than other eligible open-source HEVC encoders [17], [18]. Kvazaar code is also written at a suitable abstraction level for *high-level synthesis* (*HLS*) [19] that enables automatic *hardware description language* (*HDL*) generation from C. In this work, our intra coding accelerator is implemented using Catapult C [20] HSL tool. Through HLS, the code is more readable, design and verification times are shorter, and the design reusability is far better than with handwritten HDL equivalents.

The rest of this paper is organized as follows. Section 2 gives an overview of the adopted CPU + FPGA platform and the proposed SW/HW partitioning of Kvazaar on it. Section 3 describes the Kvazaar functionality on CPU, Section 4 the communication between CPU and FPGA, and Section 5 the implemented intra coding accelerator on FPGA. In Section 5, the speedup of HW acceleration is benchmarked against SW only encoding using 2160p (3840 × 2160) and 1080p (1920 × 1080) test videos. Section 6 concludes the paper.

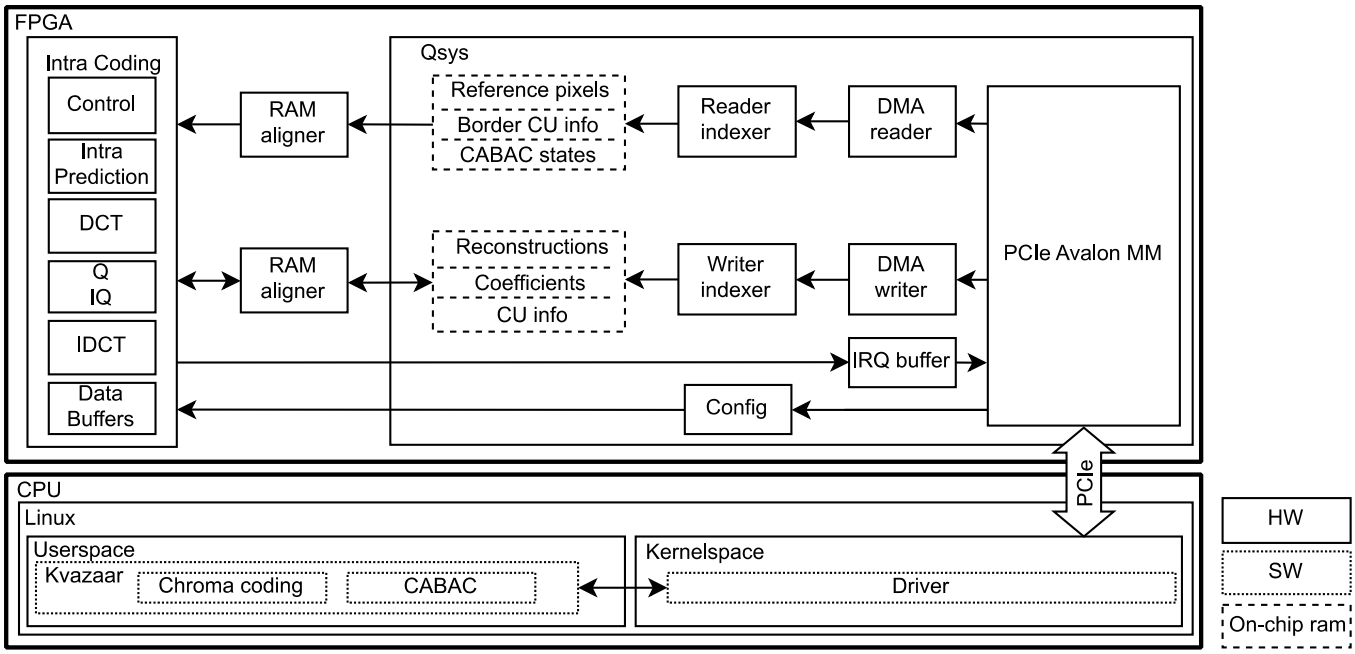


Fig. 1. Block diagram of the proposed encoder system with a single intra coding accelerator.

II. System Overview

Fig. 1 shows the block diagram of the underlying CPU + FPGA platform on which Kvazaar encoder is implemented. The backbone of the system is Nokia AirFrame server [21] with two Xeon E5-2680 v4 processors and 256 GB of memory. Arria 10 FPGA accelerator card is connected to the CPU via a PCIe bus. The operating system is CentOS 6.8.

A. Kvazaar HEVC Intra Encoder

Kvazaar [15] is an academic cross-platform open-source HEVC encoder. It contains all integral coding tools of HEVC and its modular code facilitates parallelization on multi and manycore processors as well as algorithm acceleration on HW.

Kvazaar intra encoder supports HEVC Main profile for 8-bit 4:2:0 video with ten presets out of which *fast* and *medium* presets are used in this work for their favorable cost-performance characteristics. Table I tabulates the settings of these presets. The medium preset is utilized without *rate-distortion optimized quantization* (*RDOQ*). Kvazaar implements a basic HEVC block partitioning structure in which the pictures are partitioned into *coding tree units* (*CTUs*) of size 64 × 64. CTUs can be optionally divided into four equal-sized blocks and the division can be recursively continued until the maximum hierarchical depth of the HEVC quadtree is reached. The leaf nodes of the quadtree are called *coding units* (*CUs*).

The proposed implementation of Kvazaar offers two schemes for parallel CTU coding: 1) *Wavefront Parallel Processing* (*WPP*); and 2) picture-level parallel processing. These schemes can be enabled concurrently.

B. Kvazaar Partitioning

Kvazaar is run on the platform under AI coding configuration in which the main coding tools are *intra prediction* (*IP*), *discrete cosine transform* (*DCT*), *quantization* (*Q*), *inverse Q* (*IQ*), *inverse DCT* (*IDCT*), and *context-adaptive binary arithmetic coding* (*CABAC*). In this work, the most computationally intensive coding tools including IP, DCT, Q, IQ, and IDCT are implemented with HLS and synthesized to FPGA. CABAC and other control-intensive coding tools such a control for WPP and for picture-level parallelism are executed on CPU. In addition, CPU takes care of raw input video reading, chrominance coding, and outputting the encoded bit stream.

TABLE I. Implemented Coding tools of Kvazaar Intra Encoder

Feature	Fast	Medium (wo RDOQ)
Profile	Main	Main
Internal bit depth	8	8
Color format	4:2:0	4:2:0
Coding mode	Intra	Intra
Coding units	16×16, 8×8	64×64, 32×32, 16×16, 8×8
Prediction units	16×16, 8×8	32×32, 16×16, 8×8
Transform units	16×16, 8×8	32×32, 16×16, 8×8
IP modes	35 (DC, planar, 33 angular)	35 (DC, planar, 33 angular)
Intra Search	Full	Full
Transform	Integer DCT	Integer DCT
Mode decision	Sum of absolute difference	Sum of absolute difference
Parallelization	WPP, Picture level	WPP, Picture level
SAO	Enabled	Enabled
Signhide	Disabled	Disabled
Rate Control	Disabled	Disabled
RDO	Disabled	Disabled
RDOQ	Disabled	Disabled

Arria 10 FPGA has enough resources for two instances of our intra coding accelerator including the needed peripherals and on-chip memories. Mapping a major share of CTU coding to FPGA could be utilized to decrease power dissipation through lower CPU usage. However, we are aiming at the maximum HEVC coding speed, so encoding parallelism is increased by coding additional CTUs entirely in SW with released CPU resources.

III. Functionality on Xeon

On Xeon processors, Kvazaar is run in the user space and the Linux driver in the kernel space. The Linux driver is used for the CPU-PCIe-FPGA interfacing. It is accessed by Kvazaar via ioctl, write, and read system calls.

A. User Space: Kvazaar

Kvazaar parallelization is implemented using a CPU thread pool with a single CTU as the smallest work unit. The CTUs are put in a queue in the order they would be processed in a single threaded case, and the free worker threads start processing the first CTU with no dependencies. In this work, a CTU search function of Kvazaar is modified to offload a majority of coding tasks to the HW accelerator on FPGA. Offloading is performed through system calls to the kernel driver. A worker thread sends its CTU data to the HW accelerator and sleeps until the accelerator notifies that the CTU coding on FPGA is completed. Then, the worker thread performs chrominance coding and CABAC coding for the CTU according to the results from FPGA. The threads not being able to be served by FPGA are encoded on CPU. Intra coding on FPGA has the highest priority for new CTUs and the CPU is used only when the pipeline of the HW accelerator is full.

B. Kernel Space: Driver

Fig. 2 shows the sequence chart of system calls between Kvazaar and the kernel driver. At first, Kvazaar calls the ioctl function to request a free index from the driver, which returns a nonnegative index if the HW accelerator can accept a new CTU for encoding. The driver uses semaphores initialized to the maximum CTU count supported by the accelerator. In the next step, Kvazaar calls the write function to copy all necessary data of the processed CTU to FPGA. The data being sent to FPGA is aligned in consecutive virtual memory addresses in the user space and in consecutive physical memory addresses in the kernel space. A worker thread uses the read function to request intra coding results for the CTU of interest. The thread will sleep in the kernel space until the CTU of interest has finished and the accelerator sends an interrupt signal. Both the write and read system calls return the amount of bytes (*length*) read or written successfully.

IV. Interface between Xeon and FPGA

Fig. 1 shows the FPGA interface made of the Avalon-MM Hard IP for PCIe, separate *Direct Memory Access* (*DMA*) blocks for reading and writing, and the on-chip memories of the intra coding accelerator.

A. PCIe Interface

The CPU communicates with the FPGA via the PCIe bus. The PCIe IP is configured to PCIe generation 3 × 4 with 128-bit interface and 250 MHz application clock. The IRQ Buffer block is used for generating the interrupt through the PCIe IP. The IRQ buffer detects the rising edges of the CTU ready signals from the intra coding accelerator and buffers them. The interrupt is delayed until the CPU acknowledges the previous interrupt. This is done in order to prevent two interrupts from happening in consecutive cycles, which is a limitation of the PCIe IP.

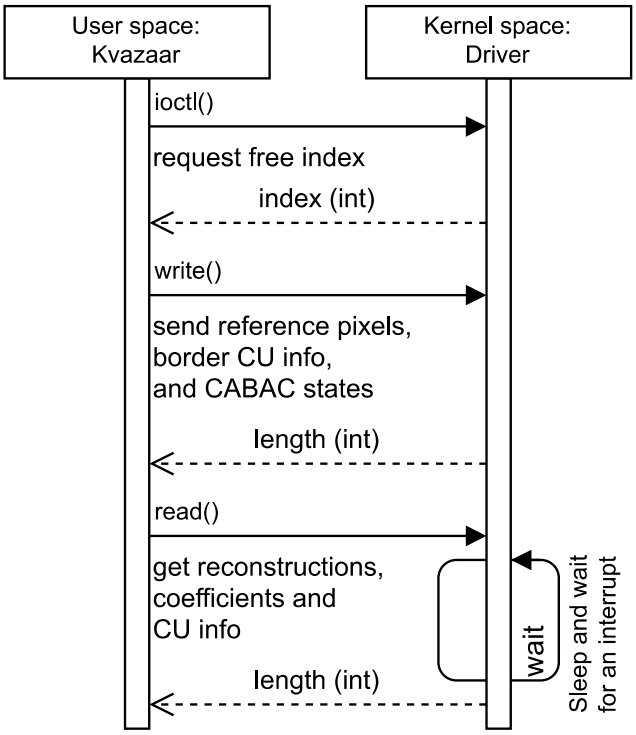


Fig. 2. Message sequence chart between Kvazaar and the kernel driver.

B. DMAs

A single intra coding accelerator consists of two DMA blocks. One DMA block is used for reading data from the shared memory and the other one is for writing data to the shared memory. This separation allows the DMA blocks to better utilize the bandwidth of the PCIe interface to the CPU memory. Our tests showed that this scheme increases the data transfer speed by 54% compared with sequential reading and writing.

The accelerator utilizes Reader and Writer indexer blocks for address translation. The blocks are configured with the CTU index before the DMA transfers are started. The DMA blocks read and write to consecutive memory addresses, but the memory structure of the on-chip memories on FPGA requires non-consecutive addresses depending on the index of the CTU.

C. On-Chip Memories

For each CTU, the HW accelerator requires the corresponding reference pixels, information about the CU borders (reconstructed pixels and modes), as well as the CTU CABAC states. The reference pixels are used to calculate *Sum of Absolute Difference* (*SAD*) values for intra mode selection and *Sum of Squared Differences* (*SSD*) values for final mode decision. CTU border pixels are used to calculate intra predictions for the CUs on the CTU borders whereas border modes are used as candidate modes when selecting the best intra mode. The CABAC states are used for *mode decision* (*MD*).

There are also on-chip memories for the final reconstructed pixels and quantized coefficients, which are flushed from the intermediate buffer. The CU info contains the resulting modes and depths from the accelerator. The RAM aligners are used as wrappers with the on-chip memories because the PCIe interface and the HW accelerator have different memory access widths.

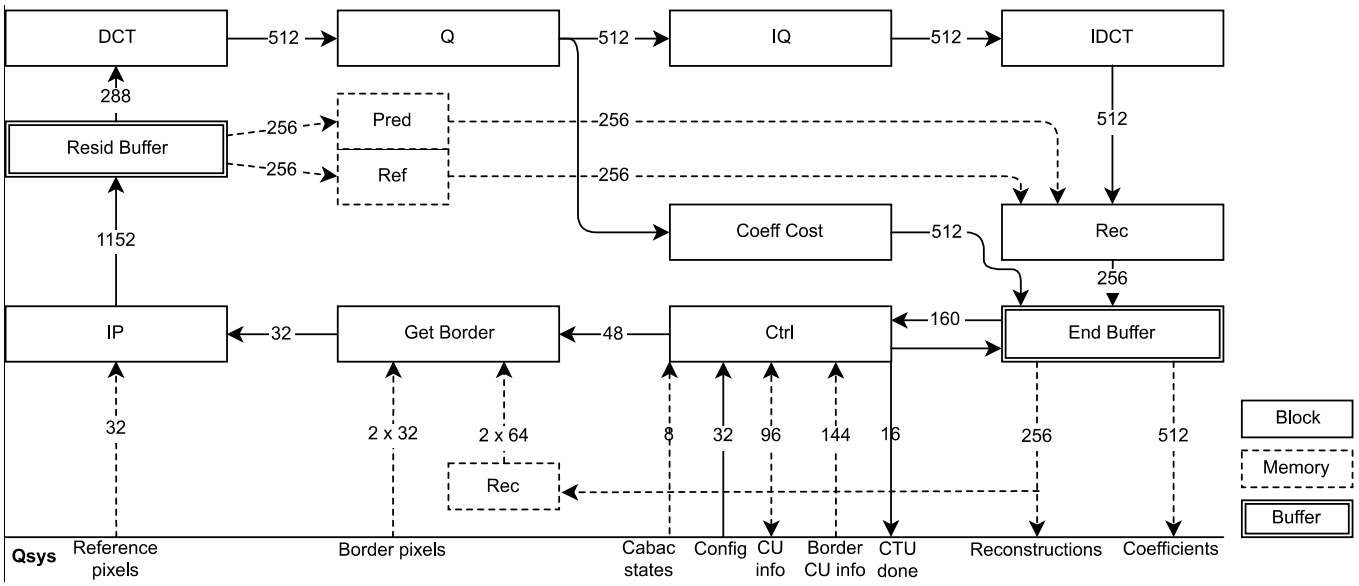


Fig. 3. The block diagram of the intra coding accelerator on FPGA.

V. Intra Coding on FPGA

Fig. 3 shows a block diagram of the intra coding accelerator. It consists of the following units implemented with HLS.

A. Intra Coding Control (Ctrl)

The Ctrl unit receives instructions from the CPU. It is split into Initialization, Scheduler, Start, and End blocks.

The Initialization block generates a full instruction set for processing a CTU. The instruction set contains operations for calculating IPs with different configurations and MD operations for selecting a CTU configuration. The HW generates the instruction set for each CTU individually.

The Scheduler block is responsible for the CTU parallelization in the HW accelerator. It loads the valid instructions for each CTU and selects the ones with the highest priority for processing. The priority for each instruction is determined so that there will be a minimal delay on the intra coding pipeline.

The Start block processes instructions from the Scheduler in order. It initializes the IP configuration for the CU according to the input instruction and sends CU information to the Get Border unit. It also notifies the CPU about finishing the CTU search if it receives the instruction for terminating the search.

The End block is at the end of the intra coding pipeline, from where it receives the search results. The results include the selected intra mode, SSD, and the estimated coding cost of the CU coefficients. The End block uses the results to calculate the MD cost for every CU configuration and stores them to the internal memory. With the MD instructions, the End block determines the best CU partitioning for the CTU according to stored cost values and flushes the pixels and the coefficients for that configuration from the buffer.

B. Get Reference Border (Get Border)

The *Get Border* unit reads the reconstructed reference pixels and sends them to the IP unit. It operates according to the configuration data consisting of CU block size and coordinates of the CU in the CTU. The coordinates are utilized when reading reconstructed pixels, i.e., either the neighboring column on the left to the CU or the neighboring row above the CU. The reconstructed pixels are read from either the CTU memory or the CTU borders memory, depending on the location of the CU within the CTU.

C. Intra Prediction (IP)

The IP unit is composed of an IP control block, SAD block, and the following IP blocks that predict 35 IP modes in parallel: DC IP (mode 0), Planar IP (mode 1), Positive Angular IP (modes 2-9, 27-34), Negative Angular IP (modes 11-25), and Zero Angular IP (modes 10, 26). All these IP blocks predict four pixels at a time, i.e., 32 × 32 block is predicted in 256 cycles, 16 × 16 block in 64 cycles, etc. The IP unit used here is an improved version of our previous IP accelerator presented in [22].

The IP unit operates according to the configuration data consisting of the CU block size and the corresponding reference pixels from the Get Border block. The IP control block filters reference pixels if needed and configures all the IP blocks that perform the prediction algorithm for a proper CU size, and all angular IP blocks for the right angle. This configurability makes the IP blocks more generic and easy to instantiate.

All angular IP blocks calculate the predicted pixels in original order, so additional transposing is not needed. The blocks also have a common control. Furthermore, IP modes with an equal distance to the horizontal (mode 10) and vertical (mode 26) modes are computed by the same IP block. For example, modes 2 and 34 are calculated in the same Positive Angular IP block since 10 - 2 = 34 - 26. This allows a reduced number of intra prediction IPs and saves area.

TABLE II. COMPARISON OF THE PROPOSED AND RELATED HEVC INTRA ENCODERS

Architecture	Technology	Board	HW Lang.	Frequency	Resolution	Coding mode	Cells	DSPs
Proposed	CPU + FPGA	Arria 10	C/C++	125 MHz	2160p@30fps	Intra	308k ALUTs	862
Zhu et al. [11]	ASIC	-	Verilog	357 MHz	1080p@44fps	Intra	2296k gates	-
Pastuszak et al. [12]	ASIC	-	VHDL	200/400 MHz	2160p@30fps	Intra	1086k gates	-
Pastuszak et al. [12]	FPGA	Arria II	VHDL	100/200 MHz	1080p@30fps	Intra	93k ALUTs	481
Atapattu et al. [13]	FPGA	Zyng ZC706	Verilog	140 MHz	1080p@30fps	Intra	84k LUTs	34
Miyazawa et al. [14]	FPGA	Custom 3xFPGA	N.A.	N.A.	1080p@60fps	Intra/Inter	N.A.	-

TABLE III. CODING SPEED WITH 2160P VIDEO (FAST PRESET)

Sequence (2160p)	No acceleration	1 accelerator		2 accelerators	
	Speed (fps)	Speed (fps)	Speedup	Speed (fps)	Speedup
Beauty	20	31	1.6×	40	2.0×
Bosphorus	21	32	1.6×	42	2.1×
HoneyBee	19	31	1.6×	41	2.1×
Jockey	22	35	1.6×	47	2.1×
ReadySetGo	20	31	1.6×	41	2.0×
ShakeNDry	17	26	1.6×	35	2.1×
YachtRide	19	30	1.6×	40	2.1×
Average	20	31	1.6×	41	2.1×

TABLE IV. CODING SPEED WITH 1080P VIDEO (MEDIUM PRESET)

Sequence (1080p)	No acceleration	1 accelerator		2 accelerators	
	Speed (fps)	Speed (fps)	Speedup	Speed (fps)	Speedup
Beauty	63	102	1.6×	136	2.2×
Bosphorus	51	82	1.6×	110	2.2×
HoneyBee	46	73	1.6×	98	2.2×
Jockey	52	84	1.6×	113	2.2×
ReadySetGo	51	79	1.6×	107	2.1×
ShakeNDry	44	70	1.6×	94	2.2×
YachtRide	49	78	1.6×	105	2.1×
Average	51	81	1.6×	109	2.2×

The SAD block reads the reference pixels of the processed CU from the corresponding on-chip memory. It also receives predicted pixels from the IP blocks and calculates the SAD in parallel for all modes. The SAD block sends all the predicted pixels and the reference pixels to a buffer, four pixel at a time. After the SAD calculation is done and the best mode is determined, SAD block notifies the buffer. The buffer recalculates the residual vector and reference pixels for the best mode and sends them to the DCT unit.

D. Discrete Cosine Transform (DCT)

The DCT unit equals the high-speed variant of our 8/16/32-point DCT unit presented in-depth in [23]. The unit performs the 2-D DCT in two successive passes and the intermediate data is stored in a transpose memory. It can process 32 residuals in parallel so that a constant data rate with full HW utilization is achieved. The latency for both passes is three cycles because of the DCT pipeline. After the 2-D transform, the 16-bit *transform coefficients* (*tcoeffs*) are passed to the Q unit.

E. Quantization (Q)

The Q unit operates according to the configuration data consisting of the block size and the quantization parameter. The unit receives one or several columns of tcoeffs from the DCT unit per write, depending on the block size. Then it performs the quantization to all tcoeffs in parallel and outputs the quantized tcoeffs to the IQ unit and the Coeff Cost unit.

F. Inverse Quantization (IQ)

The IQ unit operates according to the configuration data consisting of the block size and the quantization parameter. The unit receives one or several columns of quantized tcoeffs from the Q unit per write, depending on the block size. Then it performs the inverse quantization to all quantized tcoeffs in parallel and outputs them to the IDCT unit.

G. Inverse Discrete Cosine Transform (IDCT)

The IDCT unit equals the 8/16/32-point IDCT unit presented by us in-depth in [24]. The unit performs the 2-D IDCT in two successive passes and the intermediate data is stored in a transpose memory. The IDCT unit can process 32 tcoeffs in parallel to ensure a more constant HW utilization. The latency for both passes is three clock cycles. After the 2-D inverse transform, the 16-bit residuals are passed to the Rec unit.

H. Coefficient Cost (Coeff Cost)

The Coeff Cost unit operates according to the configuration data consisting of the block size. The unit reads all the columns of the quantized tcoeffs, which are transposed back to the original order. After the transpose, the block calculates the approximate coding cost for the CU coefficients, processing 32 coefficients in parallel.

I. Reconstruction (Rec)

The Rec module reads the reconstructed residuals from the IDCT unit and the original and predicted pixels from the memory in parallel. It generates the final reconstructed pixels and calculates the SSD for the processed CU. The reconstructed pixels are stored to memory through a buffer in order to store the right CU in the CTU sized memory.

VI. EXPERIMENTAL RESULTS

Table II tabulates the characteristics of the proposed and other existing HEVC intra encoders on ASIC and FPGA. The real-time coding speed of the ASIC-based HEVC intra encoder in [11] is limited to 1080p video. The HEVC intra encoder in [12] 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 [13] is restricted to 1080p video coding. The intra/inter HEVC encoder in [14] 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. To sum up, our proposal is the only FPGA-based implementation that supports real-time HEVC encoding up to 2160p resolution with a single board.

Table III and Table IV report HEVC coding speed of the proposed system with fast and medium presets (Table I) using 2160p and 1080p test videos, respectively. The 8-bit 4:2:0 2160p test sequences were taken from Ultra Video Group test sequence set [25] and scaled down to 1080p resolution for our tests. In both cases, the results are given for our system with 0, 1, and 2 intra coding accelerators.

The average results with the fast preset show that our implementation is able to encode 2160p video at 20 fps without HW acceleration, at 30 fps with a single accelerator, and at 40 fps with two accelerators. The speedups obtained with one and two accelerators are 1.6× and 2.1× over the pure SW implementation, respectively. In 2160p case, real-time coding speed (30 fps) requires at least one accelerator. The coding speeds of 1080p test videos are approximately 2.6 times as high as those of 2160p sequences even though a more complex medium preset (without RDOQ) is used. Hence, real-time coding speed is attainable without any HW acceleration in 1080p case. Our implementation would also be able to encode three separate real-time 1080p sequences in parallel.

VII. Conclusion

This paper presented the first known 4K HEVC intra encoder partitioned between a processor and a PCIe-connected FPGA. The encoder functionality is based on C source code of Kvazaar HEVC intra encoder and HLS was used to implement the most compute-intensive Kvazaar coding tools on FPGA. For the first time, HLS was applied to the whole intra coding chain from intra prediction to block reconstruction. HLS is generally known to reduce design and verification times over a traditional HDL workflow. This work shows that these benefits do not come at a cost of coding performance.

The proposed encoder implementation was prototyped on Nokia AirFrame Cloud Server composed of dual 14-core Intel Xeon processor and Arria 10 FPGA. On AirFrame, our solution is able to encode one 2160p video in real-time. The introduced HW acceleration roughly doubles coding speed over that of a pure SW encoder. Further performance boost could be easily obtained by inserting another FPGA card in the available slot in the server and replacing the current FPGAs with larger ones. This way, up to four times as high coding speed is anticipated.

Acknowledgment

This work was supported in part by Nokia, the European Celtic-Plus Project 4KREPROSYS, and the Academy of Finland (decision no. 301820).

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] G. J. Sullivan, J. R. Ohm, W. J. Han, and T. Wiegand, "Overview of the High Efficiency Video Coding (HEVC) standard," *IEEE Trans. Circuits Syst. Video Technol.*, vol. 22, no. 12, Dec. 2012, pp. 1649-1668.

[4] 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.

[5] *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.

[6] 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.

[7] C. C. Chi, M. Alvarez-Mesa, B. Juurlink, G. Clare, F. Henry, S. Pateux, and T. Schierl, "Parallel scalability and efficiency of HEVC parallelization approaches," *IEEE Trans. Circuits Syst. Video Technol.*, vol. 22, no. 12, Dec. 2012, pp. 1827-1838.

[8] A. Koivula, M. Viitanen, J. Vanne, T. D. Hämäläinen, and L. Fasnacht, "Parallelization of Kvazaar HEVC intra encoder for multi-core processors," *in Proc. IEEE Workshop Signal Process. Syst.*, Hangzhou, China, Oct. 2015, pp. 1-6.

[9] Y. J. Ahn, T. J. Hwang, D. G. Sim, and W. J. Han, "Implementation of fast HEVC encoder based on SIMD and data-level parallelism," *EURASIP J. Image Video Process.*, vol. 16, Dec. 2014, pp. 1-19.

[10] A. Lemmetti, A. Koivula, M. Viitanen, J. Vanne, and T. D. Hämäläinen, "AVX2-optimized Kvazaar HEVC intra encoder," *in Proc. IEEE Int. Conf. Image Processing*, Phoenix, Arizona, USA, Sep. 2016, pp. 549-553.

[11] J. Zhu, Z. Liu, D. Wang, Q. Han, and Y. Song, "HDTV1080p HEVC Intra encoder with source texture based CU/PU mode pre-decision," *2014 19th Asia and South Pacific Design Automation Conf.*, Singapore, 2014.

[12] 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.

[13] 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.

[14] 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.

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

[16] M. Viitanen, A. Koivula, A. Lemmetti, A. Ylä-Outinen, J. Vanne, and T. D. Hämäläinen, "Kvazaar: open-source HEVC/H.265 encoder," *in Proc. ACM Int. Conf. Multimedia*, Amsterdam, The Netherlands, Oct. 2016, pp. 1179-1182.

[17] *x265* [Online]. Available: http://x265.org/

[18] *Turing codec* [Online]. Available: https://github.com/bbc/turingcodec

[19] P. Coussy, D. D. Gajski, M. Meredith, and A. Takach, "An introduction to high-level synthesis," *IEEE Des. Test Comput.*, vol. 26, no. 4, Jul.-Aug. 2009, pp. 8-17.

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

[21] *AirFrame data center solution* [Online]. Available: https://networks.nokia.com/solutions/airframe-data-center-solution

[22] 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, pp. 49-56.

[23] P. Sjövall, V. Viitamäki, J. Vanne, and T. D. Hämäläinen, "High-level synthesis implementation of HEVC 2-D DCT/DST on FPGA," *in Proc. IEEE Int. Conf. Acoustics, Speech, Signal Process.*, New Orleans, Louisiana, USA, Mar. 2017, pp. 1547-1551.

[24] V. Viitamäki, P. Sjövall, J. Vanne, and T. D. Hämäläinen, "High-level synthesized 2-D IDCT/IDST implementation for HEVC codecs on FPGA," *in Proc. IEEE Int. Symp. Circuits Syst.*, Baltimore, Maryland, USA, May 2017.

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

PUBLICATION VI

FPGA-powered 4K120p HEVC intra encoder

P. Sjövall, V. Viitamäki, J. Vanne, T. D. Hämäläinen, and A. Kulmala

In *Proceedings of IEEE International Symposium on Circuits and Systems*, Florence, Italy, May 2018
DOI: 10.1109/ISCAS.2018.8351873

FPGA-Powered 4K120p HEVC Intra Encoder

Panu Sjövall, Vili Viitamäki, Jarno Vanne, Timo D. Hämäläinen, Ari Kulmala*
Laboratory of Pervasive Computing, Tampere University of Technology, Tampere, Finland
*Datacenter Infrastructure Modules, Nokia, Tampere, Finland

***Abstract*— This paper presents a hardware-accelerated Kvazaar HEVC intra encoder for 4K real-time video coding at up to 120 fps. The encoder is implemented on a Nokia AirFrame Cloud Server featuring a 2.4 GHz dual 14-core Intel Xeon processor and two Arria 10 PCI Express FPGA accelerator cards. The presented encoder is a speed-optimized version of our 1st generation 4K40p HEVC intra encoder. The proposed speedup techniques include 1) Increasing the number of FPGA cards to two; 2) Remapping the simplest multiplications from DSP blocks to logic for better FPGA utilization; 3) Making task scheduling more flexible to improve utilization rate of hardware accelerators; and 4) Increasing the pipeline depth and duplicating time-sensitive resources in the hardware accelerator. As a result, up to three hardware accelerator instances can be accommodated in a single Arria 10 so the encoder is able to make use of six accelerators. According to our experiments, the proposed encoder obtains threefold speedup over our 1st generation encoder. Our proposal is also shown to outperform all other encountered FPGA and ASIC implementations.**

Keywords*— *High Efficiency Video Coding (HEVC); Ultra High Definition Television (UHDTV); Kvazaar Intra coding; field-programmable gate array (FPGA); real-time

I. INTRODUCTION

Live Internet video is forecast to grow 15-fold in five years, accounting for 13% of all Internet video traffic by 2021 [1]. This growth comes from a plurality of new end users and multimedia applications but also from higher spatial and temporal video resolutions that are rapidly gaining ground. For example, 4K *Ultra High Definition Television* (*UHDTV*) format features 3840 × 2160 pixels (2160p) and frame rates up to 120 *frames per second* (*fps*) [2].

Despite the fast progress of transmission and storage technologies, the holistic growth of video volume makes more efficient video coding inevitable. The latest international video coding standard, *High Efficiency Video Coding* (*HEVC/H.265*) [3], [4], is developed to address these needs. This work deals with *all-intra* (*AI*) coding configuration [5] of HEVC Main Profile. It is shown to improve intra coding efficiency by 23% over that of the preceding standard AVC/H.264 [6] for the same objective quality, but at a cost of over threefold increase in coding complexity [7]. Therefore, implementing a real-time HEVC intra encoder for UHDTV format with a reasonable coding efficiency, implementation cost, and power budget requires efficient encoder optimizations and powerful computing platforms.

Multithreading [8] and *single instruction multiple data* (*SIMD*) optimizations [9] are primary design techniques for complexity reduction in *software* (*SW*) HEVC encoders. Further speedup and lower power dissipation is typically sought by offloading the compute-intensive coding tools to *hardware* (*HW*) accelerators or implementing the entire HEVC encoder on HW [10]-[13].

Our recent work [14] shows that a pure SW implementation of HEVC intra encoder is able to attain real-time coding speed for 4K30p format and formats up to 4K60p can be supported by using several software encoder instances in parallel [15]. The respective speeds are also reported for HW accelerated intra encoders [11], [16] and high-end frame rates of 4K UHDTV format are only reached with several HW encoder instances [13].

The main motivation of this work was to implement a real-time HEVC intra encoder for up to 4K120p format. The presented solution is a direct continuation to our previous work [16] where Kvazaar open-source HEVC encoder [17] is accelerated to encode 4K video at 40 fps on Nokia AirFrame Cloud Server [18]. The adopted server setup included a 2.4 GHz dual 14-core Xeon processor and an Arria 10 *PCI Express* (*PCIe*) FPGA accelerator card. Servers like AirFrame have gained a lot of traction in the recent years due to the advent of cloud gaming, telco clouds, and edge computation.

In this work, the same AirFrame server is equipped with two Arria 10 PCIe cards. In addition, up to three HW accelerator instances can be accommodated on a single FPGA by remapping the simplest multiplications to logic blocks and only allocating DSP blocks to the most compute-intensive multiplications. Individual HW accelerator instances are also boosted by using a higher pipeline depth and duplicated resources, whereas a proposed task scheduling improves the utilization rate of the instances. Together, the proposed techniques result in around threefold encoding speed over that of [16].

The original HW accelerator is implemented in [16] with Catapult C [19] *high-level synthesis* (*HLS*) [20] tool that enables automatic hardware description language generation from C source code of Kvazaar. The same approach is applied in this work since HLS offers much shorter design and verification times than manual design approaches. This is particularly true in resource remapping and pipeline modifications.

The remainder of this paper is structured as follows. Section 2 describes the applied platform and the selected SW/HW partitioning of Kvazaar on it. Section 3 presents the pipeline optimizations made for the HW accelerator instances. Section 4 introduces the proposed task-scheduling scheme among the accelerator instances. In Section 5, 4K performance of the proposed encoder is benchmarked against our earlier solution and other prior-art. Section 6 concludes the paper.

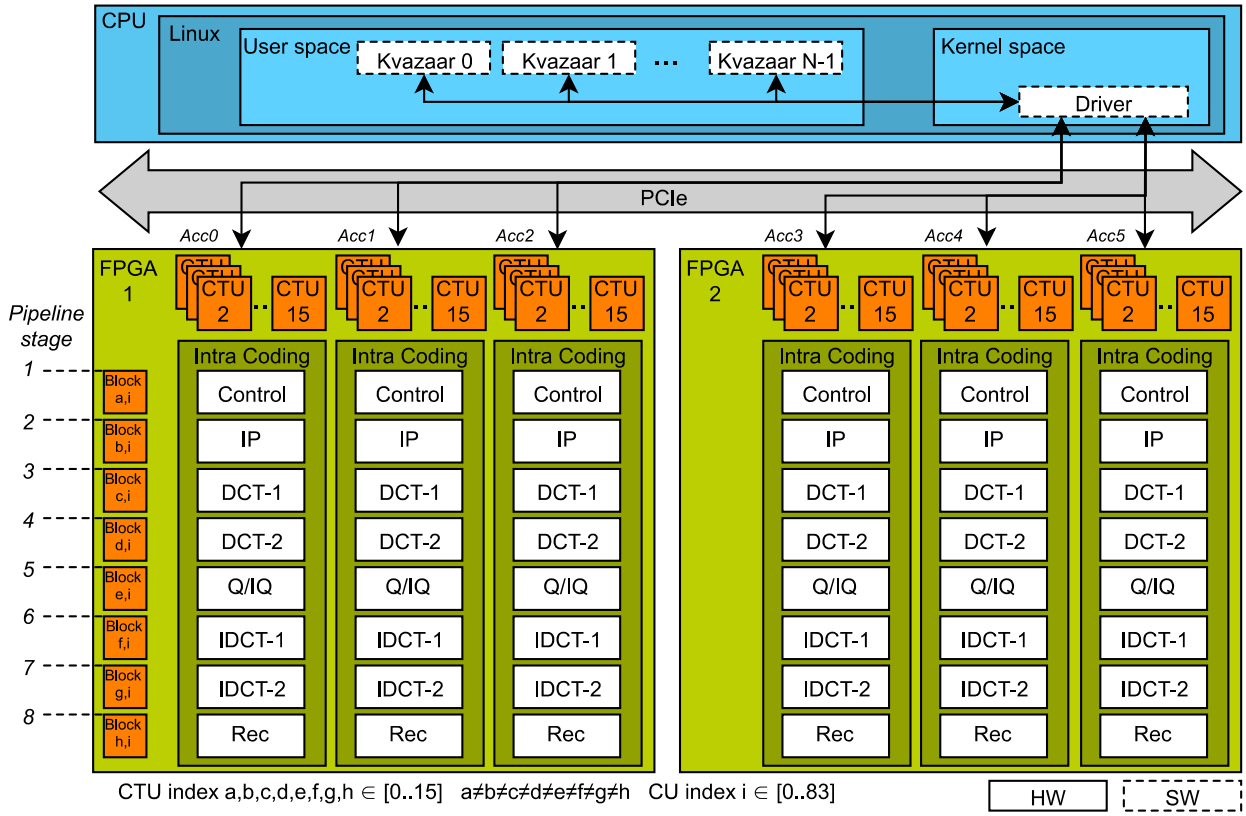


Fig. 1. Block diagram of the proposed encoder and a processing flow of CTUs in Intra Coding accelerators.

II. Overview of the Proposed System

Fig. 1 shows the block diagram of the underlying SW/HW platform. The backbone of the system is a Nokia AirFrame Cloud server [18] with two Xeon E5-2680 v4 processors and 256 GB of memory. Two Arria 10 FPGA cards are connected to the CPU via a PCIe bus. The operating system is CentOS 6.8.

A. Kvazaar Partitioning

On Xeon processors, Kvazaar [17] is run in the user space and the Linux driver in the kernel space. The Linux driver is used for the CPU-PCIe-FPGA interfacing. A single Arria 10 FPGA has enough resources for three Intra Coding accelerator instances including the needed peripherals and on-chip memories. The FPGA interface is made of the Avalon-MM Hard IP for PCIe, separate *Direct Memory Access* (*DMA*) blocks for reading and writing, and the on-chip memories of the Intra Coding accelerator. A more detailed functionality of the platform is described in our previous work [16].

Kvazaar implements a basic HEVC block partitioning in which the pictures are partitioned into *coding tree units* (*CTUs*) of size 64 × 64. CTUs can be optionally divided into four equal-sized *coding units* (*CUs*) and the division can be recursively continued until the maximum hierarchical depth of the HEVC quadtree is reached. The proposed encoder supports Kvazaar ultrafast preset [17] with extended coding tree depth so that CUs of size 32 × 32, 16 × 16, and 8 × 8 are supported. It also implements *Wavefront Parallel Processing* (*WPP*) and picture-level parallel processing for parallel CTU coding. These schemes can be enabled concurrently.

The most computationally intensive Kvazaar coding tools including *intra prediction* (*IP*), *discrete cosine transform* (*DCT*), *quantization* (*Q*), *inverse Q* (*IQ*), *inverse DCT (IDCT*), and *reconstruction* (*Rec*) are implemented with HLS and synthesized to FPGA. *Context-adaptive binary arithmetic coding* (*CABAC*) and other control-intensive coding tools are executed on CPU. In addition, the CPU takes care of raw input video reading and outputting the encoded bit stream. Mapping the major share of CTU coding to FPGA decreases the power dissipation through lower CPU usage and accelerates the whole encoding process.

B. System Configuration

In this work, the FPGA driver is upgraded to support practically any number of FPGAs, but the FPGA count is here limited to two by the available PCIe slots. Therefore, the system can contain six accelerator instances (Acc_0 - Acc_5) at maximum.

The proposed system is also configurable at run time to the chosen number of Kvazaar instances without any performance compromises. This way, the user can choose whether to encode a single video with the maximum speed or several videos in parallel. Different Kvazaar instances can also encode input videos with different encoding parameters and resolutions at the same time. This is made possible by processing each CTU individually in the Intra Coding accelerators.

III. Proposed Hardware Pipeline

Fig. 1 illustrates the processing flow of CTUs in Intra Coding accelerators. Each accelerator is able to take care of 16 CTUs

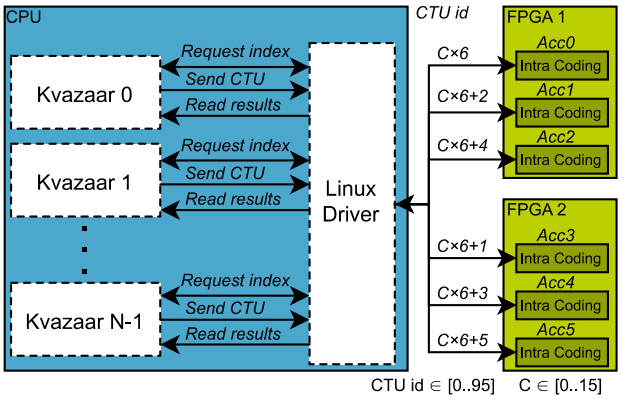


Fig. 2. CTU load balancing between Kvazaar instances and accelerators.

(0..15) simultaneously, so up to 96 CTUs can be under way in parallel with six accelerators. An eight-stage pipeline of a single accelerator can process eight blocks of separate CTUs at a time and the remaining eight CTUs are buffered for faster access. The processed blocks move sequentially through HEVC encoding stages. Altogether, each CTU can contain 4 + 16 + 64 = 84 separate CUs at maximum when CUs of size 32×32, 16×16, and 8×8 are supported.

A. Intra Prediction Pipelining

In our 1st generation encoder, IP and the creation of reference pixels were done in the same pipeline stage. Generating the reference pixels from the border pixels caused an overhead, which almost doubled the delay of the IP stage with 8×8 blocks. Therefore, the reference pixel generation was moved from the IP stage to the control stage. Now, the reference pixels of successive CUs from different CTUs are generated and buffered in advance. This way, the control stage is not blocked by the IP and the IP has an adequate small delay between predictions.

B. DCT / IDCT Pipelining

Our 1st generation encoder used only a single transform unit for the DCT and another unit for the IDCT, i.e., both algorithms ran the transform twice with the same transform unit. First from the input and second from the transpose memory. Although this design had sufficient speed for smaller number of parallel CTUs in a single Intra Coding accelerator, it caused a bottleneck when aiming higher CTU parallelism.

In the proposed work, there are two transform units for both the DCT and IDCT. In addition, the memory size of the transpose memories was doubled, allowing all transform units to run at the same time and enabling successive block pipelining. This modification practically doubled the processing speed of DCT [21] and IDCT [22] and increased the overall hardware pipeline by two stages. Although this modification increased the area of the whole Intra Coding accelerator, the speed improvement was more significant.

C. Remapping Multiplications from DSP to Logic

Our prior encoder implementations relied heavily on DSPs, mostly because they were implemented on FPGAs having half the logic area but still ~75% of the DSPs of Arria 10. Hence,

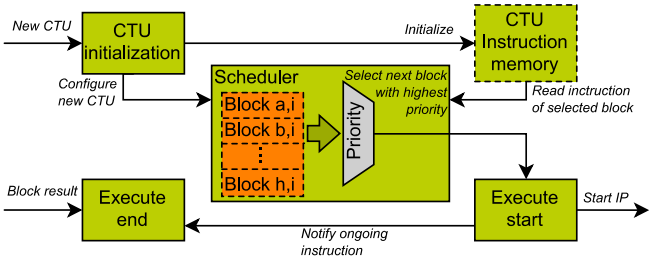


Fig. 3. Block scheduling in Intra Coding accelerator.

adding a third Intra Coding accelerator would have caused Arria 10 to run out of DSPs.

Even though the DCT and IDCT transform units were doubled in this work, we were able to fit a third Intra Coding accelerator in a single Arria 10 FPGA. This was achieved by replacing all DSPs in IP and DST transform as well as constant multiplications in DCT and IDCT with logic elements. More economic utilization of DSPs and other HLS code optimizations allowed for better routing of our design on FPGA and made it possible to increase the maximum frequency from 125 MHz to 175 MHz.

D. Other Optimizations

In our 1st generation encoder, a single Intra Coding accelerator supported eight parallel CTUs and the CPU was used to encode CTUs whenever the accelerators had no space for a new CTU. In this work, the additional CPU encoding was not used anymore since the proposed improvements have made the accelerator much faster at processing a CTU than the CPU. In addition, the increase of parallel CTUs supported on a single Intra Coding accelerator from eight to 16, caused encoding even a single CTU with the CPU to bottleneck the system. Waiting for available processing time from the accelerators and waiting for the result is faster than encoding a CTU with SW.

Performing the CTU encoding solely on the FPGA reduced the overall CPU usage and the CPU is now mostly waiting for results from the FPGA. This allows the CPU to perform other processing, even while encoding HEVC 4K120p. Further improving the CPU utilization and maximizing thread usage, the DMAs in the FPGA now generate interrupts when ready. Previously, the kernel driver polled the DMAs, but the increase in FPGAs and accelerators caused the locking mechanism in the kernel to use a major part of the processing time of a thread. With interrupts and semaphores, the thread can now sleep while waiting for the DMA completion and yield processing time for other threads.

IV. Task Scheduling and Resource Management

Scheduling of intra coding tasks is also improved to make the most of Kvazaar SW instances on a CPU and Intra Coding accelerators on FPGA.

A. CTU Load Balansing

Fig. 2 shows the process of scheduling processing time for different Kvaazar instances and choosing the best available Intra

TABLE I. CODING SPEED OF 4K VIDEO WITH DIFFERENT NUMBER OF INTRA CODING ACCELERATORS

Sequence (2160p)	Software	Single FPGA			Two FPGAs		
	No acceleration Speed (fps)	1 accelerator Speed (fps)	2 accelerators Speed (fps)	3 accelerators Speed (fps)	2 accelerators Speed (fps)	4 accelerators Speed (fps)	6 accelerator 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

TABLE II. COMPARISON OF THE PROPOSED AND RELATED INTRA ENCODERS

Architecture	Technology	Frequency	Resolution	Cells	DSPs
[10]	ASIC	357 MHz	1080@44fps	2296k gates	-
[11]	ASIC	200/400 MHz	2160@30fps	1086k gates	-
[11]	Arria II	100/200 MHz	1080@60fps	93k ALUTs	481
[12]	Zyng ZC706	140 MHz	1080@30fps	84k LUTs	34
[13]	Custom 3x FPGA	N.A.	1080@60fps	N.A.	-
[16]	CPU + Arria 10	125 MHz	2160@40fps	308k ALUTs	862
Proposed	CPU + Arria 10	175 MHz	2160@60fps	552k ALUTs	1227

Coding accelerator for a new CTU. The Linux driver is accessible by all Kvazaar instances, which request processing time on the FPGA from the driver. If there are no available resources, Kvazaar instances need to wait. Waiting instances are served in request order. The driver assigns new CTUs to different Intra Coding accelerators according to the CTU id provided by the driver. The CTU id is a running number limited by available resources, i.e., the number of Intra Coding accelerators and the number of CTUs per accelerator.

B. *Block Scheduling in Intra Coding Accelerator*

Fig. 3 shows how the Intra Coding accelerator determines the next block to the pipeline. For each CTU, a set of instructions are generated to signal the scheduler the encoding order of the blocks in a CTU. The next block of a CTU is valid for processing if the previous block of the same CTU is done. The scheduler assigns priorities to the valid blocks and chooses the one with the highest priority. The priority is higher when the next block in line is of equal size or larger than the previous one. This policy aims to keep the pipeline utilization high and it prevents larger blocks from bottlenecking smaller ones.

V. CODING SPEED ANALYSIS

Table I tabulates the obtained encoding speeds with different number of Intra Coding accelerators using the 8-bit 4:2:0 4K120p test video sequences from [23]. The average results show that our implementation is able to reach 4K30p with two accelerators, 4K60p with three accelerators, and 4K120p with six accelerators. The maximum speed of the accelerated system is 6.8 times as high as that of the pure software version. Coarsely speaking, doubling the number of accelerators doubles the encoding speed.

Our 1st generation encoder was able to encode 4K30p with a single Intra Coding accelerator but it was limited to use CU sizes of 8×8 and 16×16. In addition, it utilized the remaining CPU power for CTU encoding. Disabling 32×32 blocks in the current version would also increase its 4K coding speed to 30 fps with a single accelerator even without utilizing the CPU. With medium preset [17] and *rate-distortion-optimized quantization* (*RDOQ*) disabled, our proposal is able to encode 4K60p with six Intra Coding accelerators.

Table II tabulates the performance figures of the proposed and existing HEVC intra encoders on ASIC and FPGA. To make comparison more straightforward, our proposal is configured to use only a single FPGA with which 4K format can be encoded up to 60 fps (Table I). Our 1st generation encoder was already able to outperform related FPGA implementations and compete equally with the existing ASIC implementations. The proposed 2nd generation encoder even beats these ASIC approaches.

VI. CONCLUSION

This paper presented our 2nd generation HEVC encoder for real-time 4K intra coding. The proposed encoder was prototyped on Nokia AirFrame Cloud Server composed of a dual 14-core Intel Xeon processor and two Arria 10 FPGAs. On AirFrame, our solution is able to encode 4K video at 120 fps or four 4K videos at 30 fps.

The implemented HW acceleration speeds up the encoder by 6.8 times over the pure SW implementation and the obtained performance is three times as high as that of our 1st generation encoder. The speedup was achieved by increasing the number of FPGAs to two, improving FPGA utilization by allocating the simplest multiplications to logic, increasing the efficiency of pipeline in Intra Coding accelerator, and improving the utilization rate of the accelerators by better task scheduling.

The Intra Coding accelerators of the encoder are entirely implemented with HLS tools from C source code of Kvazaar HEVC intra encoder. HLS is generally known to reduce design and verification times over traditional design flows. This work further shows that the shorter development time does not come at a cost of coding performance.

ACKNOWLEDGMENT

This work was supported in part by Nokia, the European Celtic-Plus Project 4KREPROSYS, and the Academy of Finland (decision no. 301820).

REFERENCES

[1] Cisco, *Cisco Visual Networking Index: Forecast and Methodology*, 2016-2021, Jun. 2017.

[2] *Parameter values for ultra-high definition television systems for production and international programme exchange*, document ITU-R Rec. BT.2020-2, ITU-R, Oct 2015.

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

[4] G. J. Sullivan, J. R. Ohm, W. J. Han, and T. Wiegand, "Overview of the High Efficiency Video Coding (HEVC) standard," *IEEE Trans. Circuits Syst. Video Technol.*, vol. 22, no. 12, Dec. 2012, pp. 1649-1668.

[5] 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.

[6] *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.

[7] 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.

[8] C. C. Chi, M. Alvarez-Mesa, B. Juurlink, G. Clare, F. Henry, S. Pateux, and T. Schierl, "Parallel scalability and efficiency of HEVC parallelization approaches," *IEEE Trans. Circuits Syst. Video Technol.*, vol. 22, no. 12, Dec. 2012, pp. 1827-1838.

[9] Y. J. Ahn, T. J. Hwang, D. G. Sim, and W. J. Han, "Implementation of fast HEVC encoder based on SIMD and data-level parallelism," *EURASIP J. Image Video Process.*, vol. 16, Dec. 2014, pp. 1-19.

[10] 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.

[11] 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.

[12] 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. Application-specific Syst., Architectures and Processors*, London, United Kingdom, Jul. 2016.

[13] 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 Symp.*, San Jose, California, USA, Dec. 2013.

[14] A. Ylä-Outinen, A. Lemmetti, M. Viitanen, J. Vanne, and T. D. Hämäläinen, "Kvazaar: HEVC/H.265 4K30p intra encoder," *in Proc. IEEE Int. Symp. Multimedia*, Taichung, Taiwan, Dec. 2017.

[15] T. K. Heng, W. Asano, T. Itoh, A. Tanizawa, J. Yamaguchi, T. Matsuo, and T. Kodama, "A highly parallelized H.265/HEVC real-time UHD software encoder," *in Proc. IEEE Int. Conf. Image Processing*, Paris, France, Oct. 2014.

[16] 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.

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

[18] *AirFrame data center solution* [Online]. Available: https://networks.nokia.com/solutions/airframe-data-center-solution

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

[20] P. Coussy, D. D. Gajski, M. Meredith, and A. Takach, "An introduction to high-level synthesis," *IEEE Des. Test Comput.*, vol. 26, no. 4, Jul.-Aug. 2009, pp. 8-17.

[21] P. Sjövall, V. Viitamäki, J. Vanne, and T. D. Hämäläinen, "High-level synthesis implementation of HEVC 2-D DCT/DST on FPGA," *in Proc. IEEE Int. Conf. Acoustics, Speech, Signal Process.*, New Orleans, Louisiana, USA, Mar. 2017.

[22] V. Viitamäki, P. Sjövall, J. Vanne, and T. D. Hämäläinen, "High-level synthesized 2-D IDCT/IDST implementation for HEVC codecs on FPGA," *in Proc. IEEE Int. Symp. Circuits Syst.*, Baltimore, Maryland, USA, May 2017.

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

PUBLICATION VII

Live demonstration: 4K100p HEVC intra encoder

V. Viitamäki, P. Sjövall, J. Vanne, and T. D. Hämäläinen

In *Proceedings of International Symposium on Circuits and Systems*, Florence, Italy, May 2018
DOI: 10.1109/ISCAS.2018.8351770

Live Demonstration: 4K100p HEVC Intra Encoder

Vili Viitamäki, Panu Sjövall, Jarno Vanne, Timo D. Hämäläinen, Ari Kulmala*
Laboratory of Pervasive Computing, Tampere University of Technology, Tampere, Finland
*Datacenter Infrastructure Modules, Nokia, Tampere, Finland

***Abstract*— This paper describes a demonstration setup for real-time 4K HEVC intra coding. The system is built on Kvazaar open-source HEVC encoder partitioned between 22-core Xeon processor and two Arria 10 FPGAs. The demonstrator supports 1) live streaming of up to three 4K30p videos; or 2) offline video streaming up to 4K100p format. Live feeds are shot by three cameras whereas offline video is accessed from a local hard drive. In both cases, encoded bit stream is sent over a wired connection and played back by laptop(s). The demonstrated HEVC coding speed is over three times as fast as that of a pure software solution.**

Keywords*— *High Efficiency Video Coding (HEVC); real-time intra coding; 4K; Kvazaar; field-programmable gate array (FPGA)

I. Introduction

The explosive growth of live Internet video arouses a need for efficient real-time video compression. The latest video coding standard, *High Efficiency Video Coding* (*HEVC/H.265*) [1], brings about significantly higher coding efficiency but at the cost of substantially increased coding complexity over that of earlier standards. Therefore, implementing a real-time HEVC encoder with a reasonable coding efficiency requires efficient encoder optimizations and powerful computing platforms.

This work focuses on the *all-intra* (*AI*) coding configuration of HEVC Main Profile. The setup is built on Kvazaar HEVC encoder [2] that is shown to be the fastest fully-fledged open-source implementation for AI coding [3]. Our recent work [4] shows that a pure software implementation of Kvazaar is able to attain 4K30p coding speed on a 22-core 2.2 GHz Intel Xeon E5-2699 v4 processor. This demonstrator setup more than triples the coding speed attained in [4] by accelerating the same processor with two Altera Arria 10 FPGA cards connected via PCIe buses.

II. Setup for Kvazaar 4K100p HEVC Encoding

Fig. 1 depicts the demonstrator equipment showcased to the visitors. The implementation details of Kvazaar encoder are given in [5] on Nokia AirFrame Cloud Server which is, however, replaced by a more compact workstation in this demonstrator setup.

In the case of live streaming, three Sony FDR X1000V 4K action cameras are used to shoot three 4K (3840×2160) streams at 30 *frames per second* (*fps*). These raw feeds are captured by Epiphan AV.io HDMI capture cards and converted by three FFmpeg instances from RGB to YUV 4:2:0 format. Three Kvazaar instances encode the converted YUV streams in real-time on a FPGA-accelerated Xeon E5-2699 v4 processor. The encoded HEVC bit streams are then encapsulated by three FFmpeg instances to MPEG-2 TS format and sent over the Ethernet cables to three Asus VivoBook Pro 15 laptops for 4K playback. The average bit rate is ca. 21 Mb/s per stream.

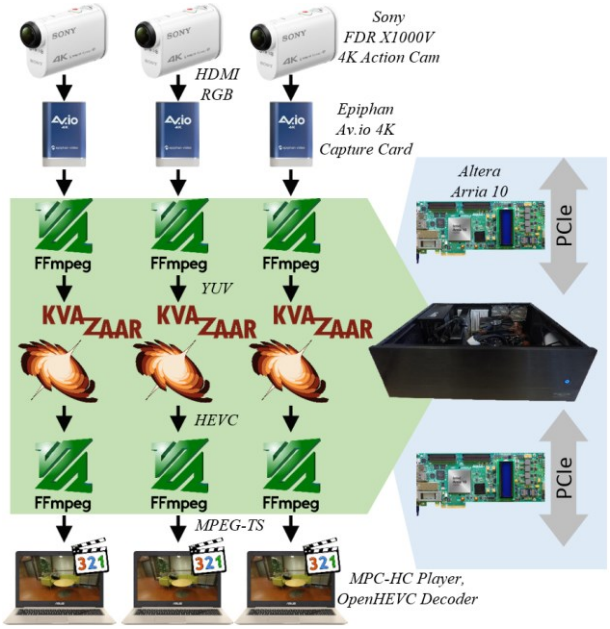


Fig. 1. Demonstration setup for Kvazaar live 3×4K30p HEVC intra coding.

In the offline case, a single YUV 4K100p video is read from a local hard drive, encoded by a single Kvazaar instance at 100 fps, converted to TS, and sent to Asus laptop for playback (with the frame rate limited to 60 fps).

The demonstrator seeks to make the visitors understand the stringent requirements of live 4K HEVC encoding. The visitors can monitor Xeon CPU usage and Kvazaar coding statistics in real time. They can also move cameras to see how the texture of the video affects the bit rate and CPU load.

References

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

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

[3] A. Lemmetti, A. Koivula, M. Viitanen, J. Vanne, and T. D. Hämäläinen, "AVX2–optimized Kvazaar HEVC intra encoder," *in Proc. IEEE Int. Conf. Image Processing,* Phoenix, Arizona, USA, Sep. 2016.

[4] A. Ylä-Outinen, A. Lemmetti, M. Viitanen, J. Vanne, and T. D. Hämäläinen, "Kvazaar: HEVC/H.265 4K30p intra encoder," *in Proc. IEEE Int. Symp. Multimedia*, Taichung, Taiwan, Dec. 2017.

[5] P. Sjövall, V. Viitamäki, J. Vanne, T. D. Hämäläinen, and Ari Kulmala, "FPGA-powered 4K120p HEVC intra encoder," *in Proc. IEEE Int. Symp. Circuits Syst*., Florence, Italy, May 2018.

PUBLICATION VIII

Dynamic resource allocation for HEVC encoding in FPGA-accelerated SDN cloud

P. Sjövall, A. Oinonen, M. Teuho, J. Vanne, and T. D. Hämäläinen

In *Proceedings of IEEE Nordic Circuits and Systems Conference*, Helsinki, Finland, Oct. 2019
DOI: 10.1109/NORCHIP.2019.8906940

Dynamic Resource Allocation for HEVC Encoding in FPGA-Accelerated SDN Cloud

Panu Sjövall, Arto Oinonen, Mikko Teuho, Jarno Vanne, Timo D. Hämäläinen
Computing Sciences, Tampere University, Finland
{panu.sjovall, arto.oinonen, mikko.teuho, jarno.vanne, timo.hamalainen}@tuni.fi

***Abstract*—This paper presents a novel approach to accelerate, distribute, and manage video encoding services in large-scale cloud systems. A proof-of-concept application is Kvazaar HEVC intra encoder, whose functionality is partitioned between FPGAs and processors. Typically, only 1-2 FPGA boards can be attached per cloud server, which severely limits the flexibility of the cloud systems. Our solution is based on Software Defined Networking (SDN), in which practically any number of FPGAs and servers can be deployed. The system features a resource manager that is responsible for allocation, deallocation, and load balancing of resources upon service requests or changes in network infrastructure. Our prototype cloud system is composed of three Intel Xeon servers, two HP SDN switches, and two Intel Arria 10 FPGAs. The servers and FPGAs have 20GbE and 40GbE connections to the SDN switches, respectively. The prototype system can encode two 4K HEVC streams at 60 fps and the performance is predicted to scale almost linearly with the number of servers and FPGAs.**

Keywords— High Efficiency Video Coding (HEVC), Kvazaar HEVC encoder, field-programmable gate array (FPGA), Software-defined networking (SDN), High-level synthesis (HLS)

I. Introduction

Video coding, deep neural networks, and data analytics are the main drivers of hardware acceleration in cloud computing. Kvazaar HEVC intra encoder has been previously accelerated on a *field-programmable gate array* (*FPGA*) [1], [2] using *High Level Synthesis* (*HLS*) [3], [4] and 4× increase in coding speed was obtained with two FPGAs. However, the FPGA boards were connected to the server via the PCIe bus, which limits the number of FPGAs per server. Moreover, PCIe FPGA cards cannot fully act as independent computing nodes.

In this work, our ultimate goal is to deploy flexible combinations of servers and FPGAs, so that the same FPGA can be shared by many servers and vice versa. In addition, FPGA acceleration should act as a microservice for as easy deployment as software resources in cloud computing. The challenge is that implementing a full protocol stack for communication and application abstraction on an FPGA takes a major portion of the FPGA resources. Hence, it introduces too much overhead for an application. Our solution is to offload most of the network functionality from the FPGA by using *software-defined networking* (*SDN*), in which any data flow is programmable and the network interface can be at very low level [5]. Our main contributions are listed below:

- Dynamic resource allocation for HEVC encoding services on a changing setup of software and hardware resources
- Usage of SDN for offloading most network functions from the FPGA
- An advanced partitioning scheme for sharing execution between servers, FPGAs, and SDN switches
- HLS implementations for the network interface, control, and HEVC accelerator logic

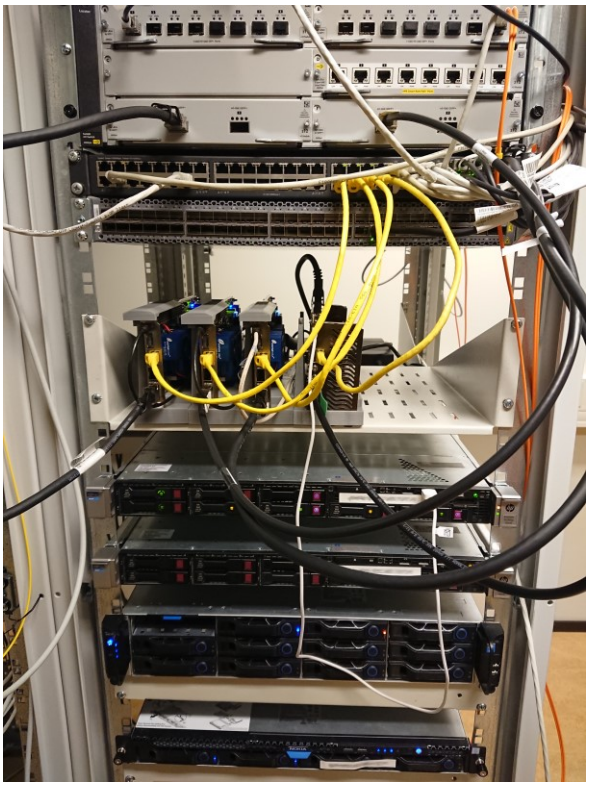


Fig. 1. Snapshot of the server rack.

- A prototype system implementing real-time 4K HEVC intra encoder

This paper is organized as follows. Section II considers the related work. Section III presents the proposed system consisting of Kvazaar FPGA accelerators, servers, and SDN switches. It also describes how the proposed resource manager and SDN are used to dynamically distribute HEVC encoding services between software and hardware resources. Section IV analyses the performance of the proposed system. Section V concludes the paper.

II. Related Work

The mainstream approaches for cloud FPGA acceleration are based on PCIe boards that are attached to the host server [6]-[9] or connected via Ethernet [10]-[12]. However, the host PCIe was still needed in [10] and all server traffic was routed through an FPGA making it even more tightly coupled to a server than in the other proposals. In [11] and [12], a complete network interface was implemented on an FPGA, so the full FPGA independency was attained at the cost of FPGA area.

For the time being, several HEVC encoders have been implemented on an FPGA [1], [2], [13]-[16], but none of them have utilized network interfaces between the processor and FPGA logic. To the best of our knowledge, this is the first paper that addresses fully independent FPGAs and servers in video encoding acceleration.

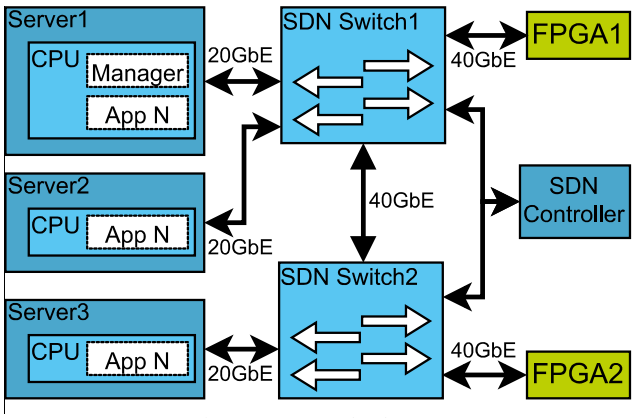


Fig. 2. Prototype cloud system.

TABLE I. CLOUD SYSTEM SPECIFICATIONS

Device	Type	CPU	Memory
Server1	HP Server	Xeon E5-2630	96GB
Server2	Nokia Airframe Cloud server [17]	Xeon E5-2680v4	256GB
Server3	Nokia Airframe Cloud server [17]	Xeon E5-2680v3	256GB
Switch1	HPE FlexFabric 5900AF 48G 4XG 2QSFP+	-	-
Switch2	HP Switch 5406Rzl2	-	-
FPGA1	Intel Arria 10 GX FPGA Dev Kit [18]	-	-
FPGA2	Intel Arria 10 GX FPGA Dev Kit [18]	-	-
Controller	HP VAN SDN Controller [19]	-	-

III. PROPOSED SYSTEM PARTITIONING

Fig. 1 and 2 show a snapshot and the corresponding network structure of our prototype cloud system, respectively. Table I tabulates the component specifications for Fig. 2. The SDN controller manages connections between the servers and FPGAs by modifying data flows in SDN switches. Each FPGA is connected to an SDN switch via one *40 Gigabit Ethernet* (*40GbE*) link and each server with 2×10GbE links.

A. Server Interfacing

In the proposed system, the servers use Linux operating system, e.g., CentOS or Ubuntu. Each server has two 10GbE SFP interfaces, which are configured to use IEEE 802.3ad Dynamic link aggregation (802.3ad, LACP) that combines the interfaces into a single load-balanced logical link with an effective bandwidth of 20GbE. As the proposed system operates on Ethernet frames, the criteria for load balancing are derived from the source and destination MAC addresses and the Ethernet type.

Because the proposed system utilizes the data link layer (layer 2), there are no built-in reliability mechanisms available with Ethernet frames. Therefore, the CPU and FPGA keep track on how many packets need to be received and sent for each *coding tree unit* (*CTU*) [20] in HEVC encoding. A lost packet causes a timeout and the same data is then re-sent to the FPGA for re-encoding. Using the IPv4 and UDP protocols from the network and transport layers (layer 3-4) would add some overhead in data rates and FPGA design complexity, but it would make it possible to send packets over different LANs. Wrapping the CTU payload inside the UDP packets would allow inclusion of UDP ports in the criteria for load balancing. These aspects will be addressed in the future.

B. FPGA Interfacing

Fig. 3 depicts the network interface on the FPGA. It includes Intel 40G Ethernet IP block and our own implementations of RX/TX Parsers and ETH Writer modules. The RX Parser decodes the Ethernet frames, ensures that the incoming frame is valid, and configures the correct accelerator instance. The TX Parser is responsible for generating the

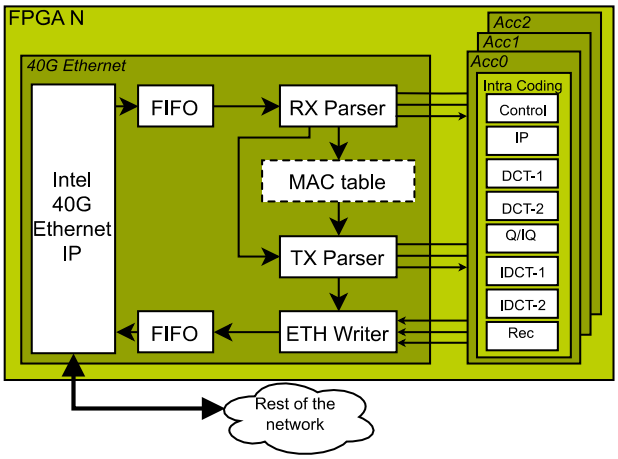


Fig. 3. Proposed FPGA accelerator architecture.

TABLE II. KVAZAAR INTRA CODING SETTINGS

Feature [20]	Kvazaar parametrization
Coding unit sizes	16×16, 8×8
Prediction unit sizes	16×16, 8×8
Transform unit sizes	16×16, 8×8
Intra prediction modes	35 (DC, planar, 33 angular)
Parallelization	Wavefront parallel processing Picture-level
Sample adaptive offset	Disabled
Sign bit hiding	Disabled
RD optimized quantization	Disabled
Transform skip	Disabled
Quantization parameter	22

Ethernet headers, gathering the payload, and controlling the ETH Writer. The frame size and the number of Ethernet frames generated per CTU are configurable. Implementing these three modules in C and using Catapult-C HLS tool [21] simplified the design process on FPGA and lowered the bar for design iterations over the corresponding approaches with VHDL or Verilog. With HLS, these blocks could also be easily modified for any accelerator usage.

The RX and TX parsers utilize a look-up-table for MAC addresses to identify which server sent the CTU for encoding. This way, the server MAC address can be translated and the results are sent back to the correct server. This approach also allows multiple servers to use the same FPGA at the same time and all results are forwarded back correctly. Fast FIFO memories compensate for differences in data widths and rates between the physical 40G Ethernet IP and our FPGA logic.

C. Kvazaar Cloud FPGA Accelerator

The execution of Kvazaar encoding is partitioned between CPUs and FPGA accelerators. First, the CTU structures of a video frame are initialized by the CPU and sent to the accelerator which implements most of the coding tools. Only the final steps, *context-adaptive binary arithmetic coding* (*CABAC*) and video stream construction, are left for the CPU. The implementation supports Kvazaar ultrafast preset [22] detailed in Table II.

A block diagram of the accelerator architecture is also shown in Fig. 3. The core component is the Kvazaar HEVC intra coding unit [1], [2], which was implemented with Catapult-C HLS tool from Kvazaar [23] open-source C code. Altogether, three accelerator instances (*Acc0*, *Acc1*, and *Acc2*) can be placed on a single Arria 10 FPGA. Each accelerator is able to process up to 16 CTUs in parallel. Software parts of the encoder can be executed on any server running Linux.

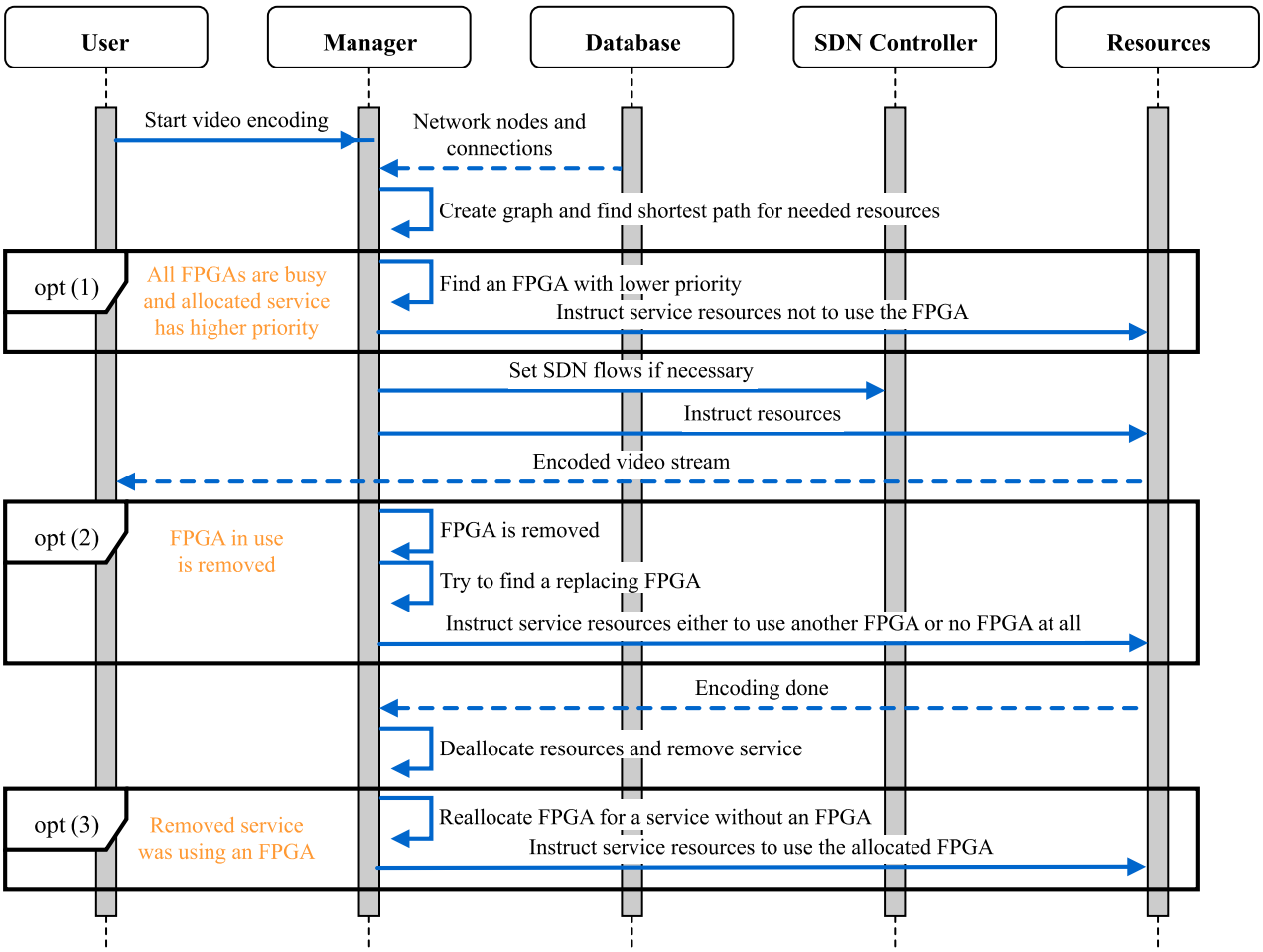


Fig. 4. Message sequence chart of dynamic resource allocation for a HEVC encoding service.

D. SDN

The SDN-controlled switches make it easy to connect the FPGAs to the network. As the data flows are automatically set by the resource manager, the FPGA does not need to support a full set of Internet protocols. Different FPGAs are identified by their MAC addresses, which is sufficient for routing the packets correctly. For example, when the SDN controller sees a MAC address assigned to a certain FPGA, it routes all associated packets to it. The data from the FPGA is routed back to the server when the source refers to the FPGA and the destination is the MAC address of the server.

E. Dynamic Resource Allocation Manager

Fig. 4 shows a message sequence chart of how our dynamic resource allocation is used for an HEVC encoding service. First, a Kvazaar HEVC encoding service is started by a user request. The resource manager collects all the needed network components including devices, switches, and connections from the database. Then, it creates a network graph and defines the most economical paths for the components, e.g., the shortest paths from a video source to a server and from the server to an FPGA. The same server and FPGA can be allocated multiple times to different services, but by monitoring the resource usage, the manager tries to optimize resource utilization for the best performance.

The manager also supports prioritization of services. Encoding speeds can thus be balanced by giving higher resolution videos a higher priority in FPGA acceleration. When a higher priority service is invoked and no FPGAs are available, the manager moves the execution of a lower priority service from the FPGA to the CPU, as shown in Fig. 4 with the option (1).

After the manager has allocated the needed resources, it sets the necessary SDN flows by using the API of the SDN controller. For example, in Fig. 2, *Server1* can access the *FPGA2* connected to a different switch by using their original MAC addresses, without any *Address Resolution Protocol* (*ARP*) messages. After the setup is ready, the manager uses POST messages to inform the resources to start the service, maybe with some additional configuration information (e.g., IDs and encoding parameters).

The manager brings robustness to the encoding process, e.g., the system is able to recover from FPGA removal. When an FPGA is switched off, the manager automatically switches the services from the removed FPGA to a CPU and starts checking equivalent replacements for the removed FPGA. This is illustrated in Fig. 4 by the option (2). Switching an encoding service from FPGA to CPU takes around one second due to the implemented encoder timeouts. Instead, switching between FPGAs and from CPU to FPGA take place instantaneously.

After a service is completed, the manager deallocates the resources in use. If an FPGA is deallocated, it is automatically assigned to the next service having no assigned FPGA, as described in Fig. 4 with the option (3). The reallocation favors services with the highest priority and the longest running time on a CPU.

TABLE III. PERFORMANCE OF KVAZAAR HEVC INTRA ENCODER ON CPU, CPU + PCIE FPGA CARD [2], AND THE PROPOSED SYSTEM (CPU + FPGA)

Sequence [24] (2160p)	Speed (fps)			Avg. frame latency (ms)			CPU utilization (%)		
	CPU only	PCIe	Proposed	CPU only	PCIe	Proposed	CPU only	PCIe	Proposed
Beauty	31	70	60	49	24	33	94	56	60
Bosphorus	43	70	62	33	18	25	94	33	35
HoneyBee	34	70	61	41	19	27	96	49	48
Average	36	70	61	41	20	28	95	46	48

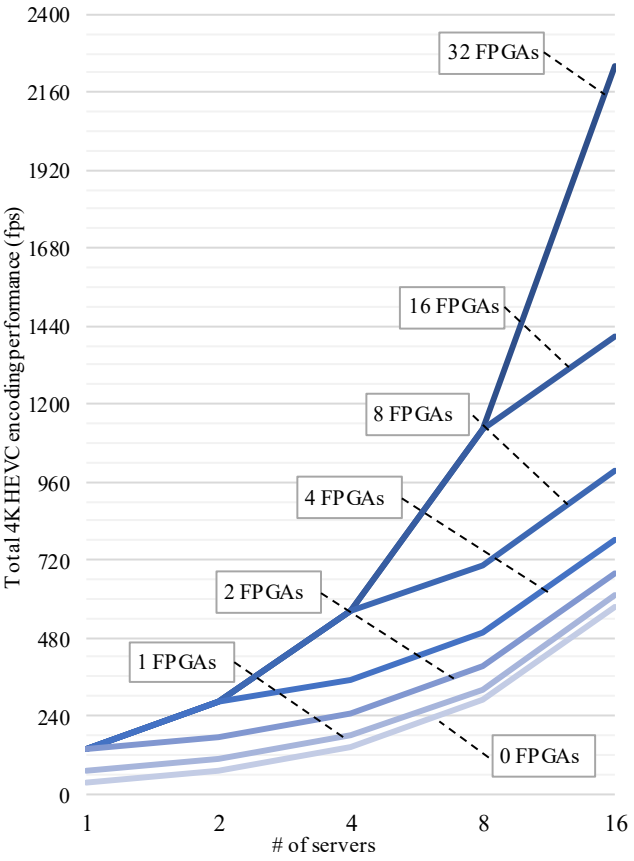


Fig. 5. Predicted encoding performance with differently scaled setups.

IV. PERFORMANCE ANALYSIS

Table III reports the performance results for Kvazaar HEVC encoder on three different platforms: 1) CPU-only; 2) CPU with a single PCIe FPGA card [2]; and 3) the proposed prototype cloud system containing a single CPU and FPGA. For fair comparison, the resources of the cloud setup are unified with that of the PCIe approach. Furthermore, all these setups use an equivalent Intel Xeon CPU and the latter two equivalent Arria 10 FPGA for acceleration.

According to our results, the proposed cloud approach speeds up HEVC encoding by 1.5-2 fold over that of the CPU-only case. However, the average coding speed of our proposal is around 9 fps slower than that of the PCIe approach. There are three reasons for the slowdown: the usage of a 20Gbps Ethernet link in place of a 32Gbps PCIe bus, the overhead of Ethernet packets, and reduced parallelism due to longer waiting times of Ethernet frames. Nevertheless, our proposal is still able to encode 4K resolution test videos at 60 fps.

It is notable that the average frame latency with the fiber connection is around 40% higher than that of the PCIe bus. On the other hand, the Ethernet interfacing still has 31% smaller average frame latency than with CPU-only encoding. The CPU utilization is nearly the same in the cloud and PCIe approaches. Both solutions accelerate HEVC encoding and still use around 50% less CPU resources than the CPU-only case.

The proposed system is able to encode 4K video at 90 fps with two FPGAs and a single server (Fig. 2). In this case, the maximum speed is limited by the 20GbE connection. Alternatively, our system can encode two 4K sequences at 60 fps with two servers and two FPGAs. Using 40Gbps network cards on the servers in place of 2×10GbE would remove this limitation and provide smaller latency as well as faster encoding speed. However, acquiring these network cards is left for the future.

The system also appears to be robust, as Kvazaar execution can be switched between CPUs and FPGAs on the fly depending on the resource availability. This means, in practice, that the system can switch an encoding process between FPGA accelerators and recover from an FPGA removal, all without interrupting the encoding process. These features are visualized in [25].

Despite the minor performance penalty, the fiber connected FPGAs allow much better scalability than the dedicated PCIe-based approach. For example, both approaches would need three Kvazaar accelerator instances on FPGA for 4K60p encoding, but only the cloud approach is able to attain the same speed with three smaller FPGAs, each having a single accelerator instance.

The proposed dynamic resource allocation and partitioning scheme leaves lots of room for further performance scaling. Fig. 5 predicts the total encoding performance of differently scaled up systems with the 40GbE links in servers. For example, the graphs show equal performance for the systems composed of 16 servers or 4 servers with 8 FPGAs. A high-end system with 16 servers and 32 FPGAs has potential to encode 64 HEVC streams at 4K30p or 16 streams at 4K120p simultaneously.

V. CONCLUSION

This paper presented an automated approach for managing services with a lightweight framework that connects multiple servers and FPGAs in an SDN based cloud. The combination of a dedicated resource manager and SDN makes it possible to have practically any number of independent FPGAs on the network without wasting FPGA resources for communication and application abstraction.

Instead of using complicated network protocols, the proposed system uses the SDN controller and SDN switches for routing data. A dedicated SDN controller allows scaling the network to a large-scale cloud infrastructure without losing the speed and connectivity of a small network.

The proposed system was also validated in practice with a proof-of-concept real-time 4K HEVC encoder implementation. It was shown to attain near the same speed as the previous PCIe equivalent implementation but with much better scalability and robustness.

ACKNOWLEDGMENT

This work was supported in part by the European Celtic-Plus project VIRTUOSE, the Academy of Finland (decision no. 301820), Nokia Foundation, and the Finnish Foundation for Technology Promotion.

REFERENCES

[1] 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.

[2] P. Sjövall, V. Viitamäki, J. Vanne, T. D. Hämäläinen, and A. Kulmala, "FPGA-powered 4K120p HEVC intra encoder," *in Proc. IEEE Int. Symp. Circuits Syst.*, Florence, Italy, May 2018.

[3] S. Lahti, P. Sjövall, J. Vanne, and T. D. Hämäläinen, "Are we there yet? A study on the state of high-level synthesis," *IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst*., vol. 38, no. 5, May 2019, pp. 898-911.

[4] P. Coussy, D. D. Gajski, M. Meredith, and A. Takach, "An introduction to high-level synthesis," *IEEE Des. Test Comput.*, vol. 26, no. 4, Jul.-Aug. 2009, pp. 8-17.

[5] M. Vajaranta, V. Viitamäki, A. Oinonen, T. D. Hämäläinen, A. Kulmala, and J. Markunmäki, "Feasibility of FPGA accelerated IPsec on cloud," *in Proc. Euromicro Symp. Digit. Syst. Des.*, Prague, Czech Republic, Aug. 2018.

[6] A. Putnam *et al*., "A reconfigurable fabric for accelerating large-scale datacenter services," *IEEE Micro*, vol. 35, no. 3, May-June 2015, pp. 10-22.

[7] S. A. Fahmy, K. Vipin, and S. Shreejith, "Virtualized FPGA accelerators for efficient cloud computing," *IEEE Int. Conf. Cloud Comput. Technol. Sci*, Vancouver, British Columbia, Canada, Nov.-Dec. 2015.

[8] Z. Zhu, A. X. Liu, F. Zhang, and F. Chen, "FPGA resource pooling in cloud computing," *IEEE Trans. Cloud Comput*., Early Access.

[9] J. Lallet, A. Enrici, and A. Saffar, "FPGA-based system for the acceleration of cloud microservices," *in Proc. IEEE Int. Symp. Broadband Multimedia Syst. Broadcast.*, Valencia, Spain, Jun. 2018.

[10] A. M. Caulfield *et al*., "A cloud-scale acceleration architecture," *in Proc. Annual IEEE/ACM Int. Symp. Microarchitecture*, Taipei, Taiwan, Oct. 2016.

[11] J. Weerasinghe, F. Abel, C. Hagleitner, and A. Herkersdorf, "Enabling FPGAs in hyperscale data centers," *in Proc. IEEE Int. Conf. Ubiquitous Intell.,* Beijing, China, Aug. 2015.

[12] J. Weerasinghe, R. Polig, F. Abel, and C. Hagleitner, "Network-attached FPGAs for data center applications," *in Proc. Int. Conf. Field-Programmable Technol.*, Xi'an, China, Dec. 2016.

[13] 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.

[14] 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.

[15] 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. Application-specific Syst., Architectures and Processors*, London, United Kingdom, Jul. 2016.

[16] 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 Symp.*, San Jose, California, USA, Dec. 2013.

[17] *AirFrame data center solution.* Accessed on: Sep. 20, 2019. [Online]. Available: https://www.nokia.com/networks/solutions/airframe-data-center-solution/

[18] *Arria 10.* Accessed on: Sep. 20, 2019. [Online]. Available: https://www.intel.com/content/www/us/en/programmable/products/boards_and_kits/dev-kits/altera/kit-a10-gx-fpga.html

[19] *HP Virtual Application Networks SDN Controller.* Accessed on: Sep. 20, 2019. [Online] Available: https://support.hpe.com/hpsc/doc/public/display?docId=emr_na-c03967699

[20] G. J. Sullivan, J. R. Ohm, W. J. Han, and T. Wiegand, "Overview of the High Efficiency Video Coding (HEVC) standard," *IEEE Trans. Circuits Syst. Video Technol.*, vol. 22, no. 12, Dec. 2012, pp. 1649-1668.

[21] *Catapult High-Level Synthesis.* Accessed on: Sep. 20, 2019. [Online]. Available: https://www.mentor.com/hls-lp/catapult-high-level-synthesis/

[22] *Kvazaar HEVC encoder.* Accessed on: Sep. 20, 2019. [Online]. Available: https://github.com/ultravideo/kvazaar

[23] M. Viitanen, A. Koivula, A. Lemmetti, A. Ylä-Outinen, J. Vanne, and T. D. Hämäläinen, "Kvazaar: open-source HEVC/H.265 encoder," *in Proc. ACM Int. Conf. Multimedia*, Amsterdam, The Netherlands, Oct. 2016.

[24] *Test Sequences.* Accessed on: Sep. 20, 2019. [Online]. Available: http://ultravideo.cs.tut.fi/#testsequences

[25] P. Sjövall, M. Teuho, A. Oinonen, J. Vanne, and T. D. Hämäläinen, "Visualization of dynamic resource allocation for HEVC encoding in FPGA-accelerated SDN cloud," *in Proc. IEEE Int. Conf. Visual Commun. Image Process.*, Sydney, Australia, Dec. 2019.

PUBLICATION IX

Visualization of dynamic resource allocation for HEVC encoding in FPGA-accelerated SDN cloud

P. Sjövall, M. Teuho, A. Oinonen, J. Vanne, and T. D. Hämäläinen

In *Proceedings of IEEE Visual Communications and Image Processing*, Sydney, New South Wales, Australia, Dec. 2019
DOI: 10.1109/VCIP47243.2019.8966042

Visualization of Dynamic Resource Allocation for HEVC Encoding in FPGA-Accelerated SDN Cloud

Panu Sjövall, Mikko Teuho, Arto Oinonen, Jarno Vanne, Timo D. Hämäläinen
Computing Sciences, Tampere University, Finland
{panu.sjovall, mikko.teuho, arto.oinonen jarno.vanne, timo.hamalainen}@tuni.fi

***Abstract*—This paper describes a demonstration setup to visualize dynamic resource allocation for real-time HEVC encoding services in FPGA-accelerated cloud. The demonstrated application is Kvazaar HEVC intra encoder, whose functionality is partitioned between FPGAs and processors. During the demonstration, several encoding services can be invoked with requests to the resource manager, which is responsible for allocation, deallocation, and load balancing of resources in the network. The manager provides JSON data to the visualizer, which uses D3 JavaScript library to visualize 1) the physical network structure; 2) running services; and 3) performance of the network elements. This interactive demonstration allows users to request new video streams, view the encoded streams, observe the visualization of the network and services, and manually turn on/off resources to test the robustness of the system.**

Keywords— Data Center processing, High Efficiency Video Coding (HEVC), Kvazaar HEVC encoder, Field-programmable gate array (FPGA), Software-defined networking (SDN)

I. Introduction

The rapidly increasing popularity and complexity of video coding, deep neural networks, and data analytics call for hardware acceleration in cloud computing. In the mainstream cloud computing systems, *field-programmable gate array* (*FPGA*) acceleration is typically implemented by PCIe cards attached to host servers [1], [2]. However, this approach ties the number of FPGAs to the server counts.

We solved this limitation by connecting FPGAs to servers via fiber and letting FPGAs act as independent nodes. We also replaced the full protocol stack implementations on the FPGAs with *Software-Defined Networking* (*SDN*). The SDN approach enables sharing any FPGA with any server through programmable data flows. The proposed system makes use of a proactive resource manager that dynamically switches between available software and hardware resources, without breaking up the live video stream.

II. Demonstration Setup

Fig. 1 shows our cloud architecture. It consists of three Xeon servers, two Intel Arria 10 FPGAs, and two HP SDN switches with HP VAN SDN Controller. The network components are specified in Table I.

Fig. 2 illustrates the demonstration setup. The prototype cloud is physically located at Tampere University (Fig. 2 (a)) and it is accessed over the network via VPN in the demonstration. A laptop (Fig. 2 (b)) is used for displaying the visualization interface, user interaction, and video playback.

The demonstrated application is Kvazaar HEVC encoder [3] which can be executed as a CPU-only service or it can be partitioned between CPUs and FPGA accelerators [4], [5]. Kvazaar is a standard software encoder [6] written in C and its hardware accelerator is implemented with Catapult-C high-level synthesis tool [7] from the C code. The inputs for all

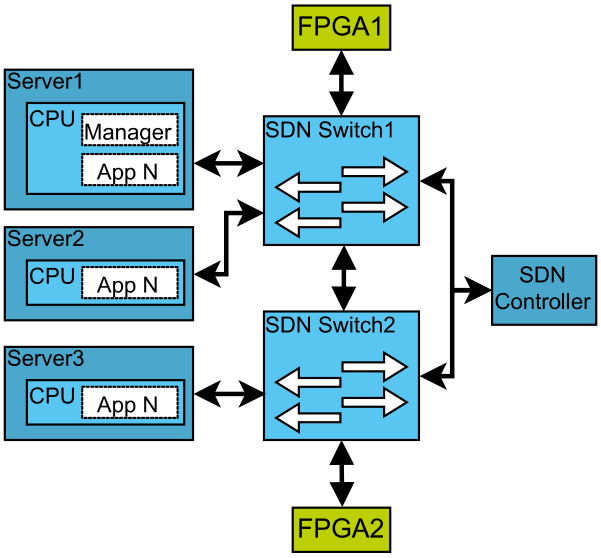


Fig. 1. Prototype cloud system

TABLE I. Cloud Component Specifications

Device	Type	CPU	Memory
Server1	HP Server	Xeon E5-2630	96GB
Server2	Nokia Airframe Cloud server [8]	Xeon E5-2680v4	256GB
Server3	Nokia Airframe Cloud server [8]	Xeon E5-2680v3	256GB
Switch1	HPE FlexFabric 5900AF 48G 4XG 2QSFP+	-	-
Switch2	HP Switch 5406Rzl2	-	-
FPGA1	Intel Arria 10 GX FPGA Dev Kit [9]	-	-
FPGA2	Intel Arria 10 GX FPGA Dev Kit [9]	-	-
Controller	HP VAN SDN Controller [10]	-	-

demonstrated encoding services are raw video files with different resolution. The output is in *HTTP Live Streaming* (*HLS*) format, which is decoded in live playback.

III. Run-Time Visualizer

The visualizer is written in JavaScript using D3 library. It gets input data in real time in JSON format from the manager. The visualizer illustrates dynamic deployment of HEVC encoding tasks in run time. It can work with an arbitrary set of resources and with varying number of encoding services.

The physical view of the network is shown in Fig. 2 (c). The symbols correspond to the device type and the server symbol size to the number of CPU cores. The physical view also shows 1) the CPU load graph inside the server symbol; 2) details as tooltips; and 3) connection bandwidth with changing line width. The services are shown in Fig. 2 (d) by dividing them as input sources, software (Kvazaar_HEVC) and hardware (Kvazaar_HEVC_acc) encoding services, and output destinations. Encoding speeds and bitrates are also shown for every service.

During the demonstration, existing computing resources can also be manually removed to see how the network self-organizes without breaking up video streaming. This can be seen from the visualization in real time.

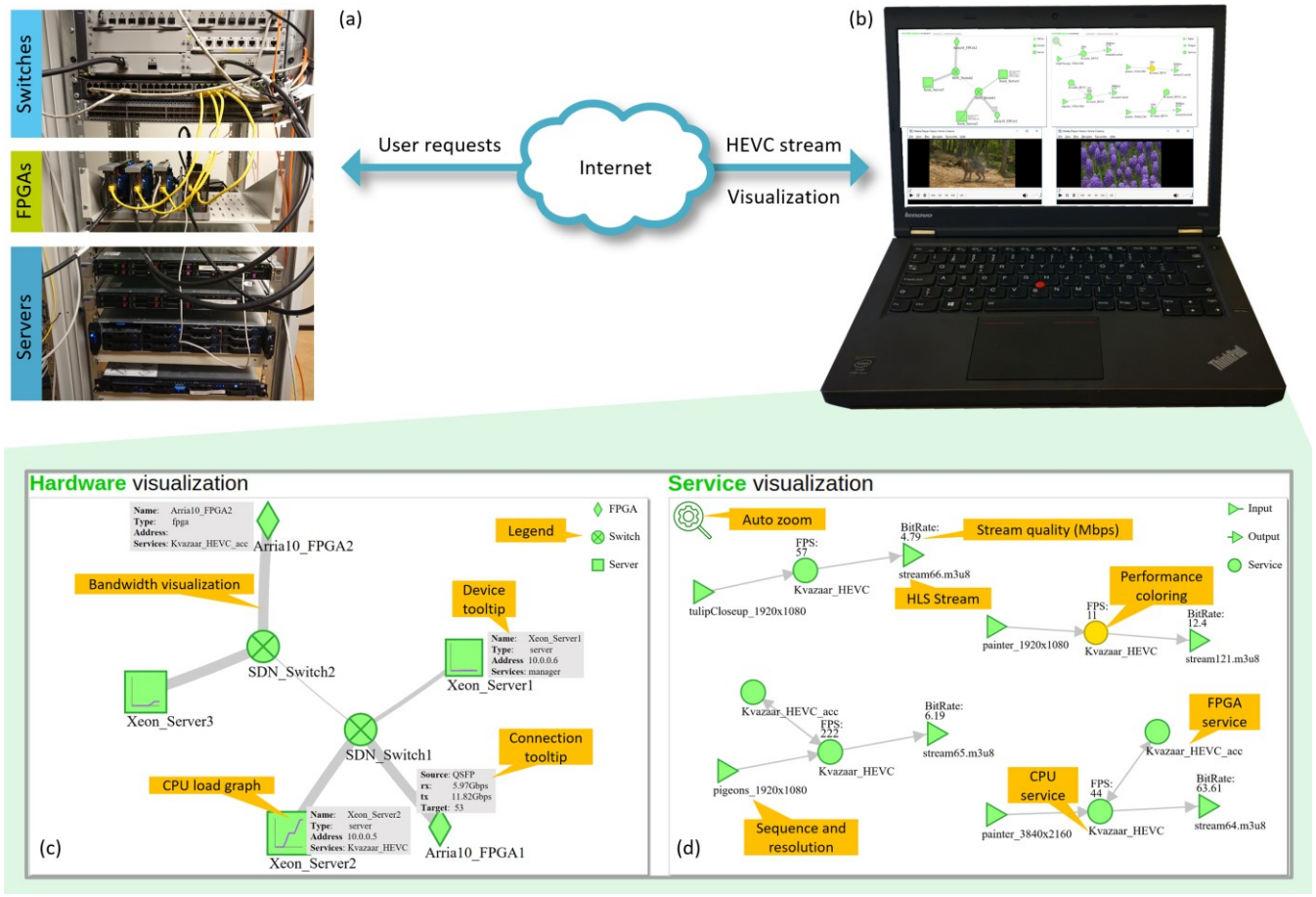


Fig. 2. The demonstration setup. (a) Remote server. (b) Laptop at the venue. (c) Physical view of the visualizer. (d) Service view of the visualizer.

IV. Conclusion

Our system with fiber connected FPGAs provides a new microservice based approach for hardware accelerated video encoding services in the cloud. This paper described a setup to demonstrate the basic operating principles of our proposal. In the demonstration, a special attention is paid to dynamic resource allocation for HEVC encoding services, switching service execution between CPUs and FPGAs, recovering from changes, and scalability of our architecture. The visualizer offers a real-time view of the available resources, running services, and performance.

Acknowledgment

This work was supported in part by the European Celtic-Plus project VIRTUOSE, the Academy of Finland (decision no. 301820), Nokia Foundation, and the Finnish Foundation for Technology Promotion.

References

[1] A. Putnam *et al.*, "A reconfigurable fabric for accelerating large-scale datacenter services," *IEEE Micro*, vol. 35, no. 3, May-June 2015, pp. 10-22.

[2] A. M. Caulfield *et al.*, "A cloud-scale acceleration architecture," *in Proc. Annual IEEE/ACM Int. Symp. Microarchitecture*, Taipei, Taiwan, Oct. 2016.

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

[4] 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.

[5] P. Sjövall, V. Viitamäki, J. Vanne, T. D. Hämäläinen, and A. Kulmala, "FPGA-powered 4K120p HEVC intra encoder," *in Proc. IEEE Int. Symp. Circuits Syst.*, Florence, Italy, May 2018.

[6] M. Viitanen, A. Koivula, A. Lemmetti, A. Ylä-Outinen, J. Vanne, and T. D. Hämäläinen, "Kvazaar: open-source HEVC/H.265 encoder," *in Proc. ACM Int. Conf. Multimedia*, Amsterdam, The Netherlands, Oct. 2016.

[7] *Catapult High-Level Synthesis* [Online]. Available: https://www.mentor.com/hls-lp/catapult-high-level-synthesis/

[8] *AirFrame data center solution* [Online]. Available: https://networks.nokia.com/solutions/airframe-data-center-solution

[9] *Arria 10* [Online]. Available: https://www.intel.com/content/www/us/en/programmable/products/boards_and_kits/dev-kits/altera/kit-a10-gx-fpga.html

[10] *HP Virtual Application Networks SDN Controller* [Online]. Available: http://h17007.www1.hpe.com/docs/networking/solutions/sdn/4AA4-8807ENW.pdf

PUBLICATION X

High-level synthesis implementation of an embedded real-time HEVC intra encoder on FPGA for media applications

P. Sjövall, A. Lemmetti, J. Vanne, S. Lahti, and T. D. Hämäläinen

ACM Transactions on Design Automation of Electronic Systems, vol. 27, no. 7
DOI: 10.1145/3491215

High-Level Synthesis Implementation of an Embedded Real-Time HEVC Intra Encoder on FPGA for Media Applications

Panu Sjövall, Ari Lemmetti, and Jarno Vanne

Ultra Video Group, Computing Sciences, Tampere University, Finland

Sakari Lahti and Timo D. Hämäläinen

SoC Design Group, Computing Sciences, Tampere University, Finland

High Efficiency Video Coding (HEVC) is the key enabling technology for numerous modern media applications. Overcoming its computational complexity and customizing its rich features for real-time HEVC encoder implementations, calls for automated design methodologies. This paper introduces the first complete High-Level Synthesis (HLS) implementation for HEVC intra encoder on FPGA. The C source code of our open-source Kvazaar HEVC encoder is used as a design entry point for HLS that is applied throughout the whole encoder design process, from data-intensive coding tools like intra prediction and discrete transforms to more control-oriented tools such as context-adaptive binary arithmetic coding (CABAC). Our prototype is run on Nokia AirFrame Cloud Server equipped with 2.4 GHz dual 14-core Intel Xeon processors and two Intel Arria 10 PCIe FPGA accelerator cards with 40 Gigabit Ethernet. This proof-of-concept system is designed for hardware-accelerated HEVC encoding and it achieves real-time 4K coding speed up to 120 fps. The coding performance can be easily scaled up by adding practically any number of network-connected FPGA cards to the system. These results indicate that our HLS proposal not only boosts development time, but also provides previously unseen design scalability with competitive performance over the existing FPGA and ASIC encoder implementations.

CCS CONCEPTS •Hardware~Electronic design automation~High-level and register-transfer level synthesis~Hardware-software codesign •Hardware~Integrated circuits~Reconfigurable logic and FPGAs~Hardware accelerators •Computing methodologies~Computer graphics~Image compression

Additional Keywords and Phrases: High-level synthesis (HLS), High Efficiency Video Coding (HEVC), Field-programmable gate array (FPGA), video encoding, Kvazaar intra encoder

ACM Reference Format:

This work is part of the ADACORSA project that has received funding within the ECSEL JU in collaboration with the European Union's H2020 Framework Programme (H2020/2014-2020) and National Authorities, under grant agreement 876019. Other supporters include Nokia Foundation and the Finnish Foundation for Technology Promotion.
The authors are with Tampere University, Korkeakoulunkatu 7, 33720 Tampere, Finland; emails: {panu.sjovall, ari.lemmetti, jarno.vanne, sakari.lahti, timo.hamalainen}@tuni.fi

1 INTRODUCTION

Proliferation of media applications, omnipresent connectivity, and immersive *extended reality* (*XR*) technologies foster the phenomenal growth of video traffic, which is estimated to account for 82% of all global IP traffic by 2022 [1]. The latest widespread MPEG/ITU-T video coding standard, *High Efficiency Video Coding* (*HEVC/H.265*) [2], [3], mitigates this growth by reducing the transmission and storage needs of modern video applications. HEVC halves the bit rate over the preceding *Advanced Video Coding* (*AVC/H.264*) [4] standard for the same subjective visual quality, but typically at the cost of considerable computational complexity overhead in practical encoders. Therefore, the deployment of HEVC calls for powerful implementations, which are able to tackle its computational complexity with acceptable coding efficiency and power budget.

Multithreading and *single instruction multiple data* (*SIMD*) are commonly used optimization techniques in software (SW) HEVC encoders [5]-[7]. Further speedup and lower power dissipation are typically sought by offloading compute-intensive coding tools to *hardware* (*HW*) accelerators or implementing the entire HEVC encoder on HW [8]-[37]. However, HW design is traditionally very time-consuming, so the efficient development of modern video encoders calls for automated and agile design methodologies, efficient encoder optimization techniques, and specialized high-performance computing platforms. Our work addresses these requirements by using: 1) *high-level synthesis* (*HLS*) [38] as a design methodology, 2) a fully-fledged practical Kvazaar SW HEVC encoder [5] as a design entry point, and 3) a heterogeneous combination of general-purpose CPUs and *field-programmable gate array* (*FPGA*) accelerator cards as an underlying HW platform.

Our primary motivation is to implement a real-time 4K HEVC intra encoder that is easily customizable for different media applications and scalable for different performance requirements. To this end, we propose to use Catapult HLS tool [39] that can automatically generate *register-transfer level* (*RTL*) code from C/C++ code. Thus, there is no need to manually rewrite the existing source code of Kvazaar to traditional *hardware description languages* (*HDLs*) like VHDL and Verilog. HLS has been reported to provide 4-6 times increase in productivity [40], mainly because the behavioral code is more readable, design and verification times are shorter, and the design reusability is far better over that of handwritten HDL. This work focuses on the *all-intra* (*AI*) [41] coding configuration of HEVC Main Profile but the proposed design approach can be applied to other HEVC profiles or video codecs as well.

Unlike prior art, our HEVC encoder is completely implemented with HLS, i.e., the use of HLS is not only limited to data-intensive algorithms like HEVC *intra prediction* (*IP*), *discrete sine/cosine transform* (*DST*/*DCT*), *quantization* (*Q*), *inverse Q* (*IQ*), *inverse DST/DCT* (*IDST/IDCT*), and reconstruction, but it is also applied to control-intensive tools such as intra search control and *context-adaptive binary arithmetic coding* (*CABAC*). Even though all design decisions in this work have been taken from the perspective of using HLS for efficient FPGA implementations, HLS would also allow us to use the same Kvazaar source code to generate optimized RTL for *application specific integrated circuit* (*ASIC*) implementations, but this is beyond the scope of this paper.

The rest of the paper is organized as follows. Sections 2 and 3 provide an overview of HEVC intra coding and the related work. Section 4 gives the motivation and rationale for selecting HLS as the proposed design methodology for fast HEVC encoder development and prototyping on FPGA. Section 5 presents the system-level architecture and HW/SW partitioning scheme for the proposed intra HEVC encoder. The main HW components, the *Intra Search Core* and *CABAC Core*, are detailed in Section 6 and Section 7, respectively. Section 8 evaluates the performance of our proof-of-concept prototype system and compares it with prior art. Finally, Section 9 concludes the paper.

2 OVERVIEW OF HEVC INTRA CODING

HEVC adopts the conventional hybrid video coding scheme (inter/intra prediction, transform coding, and entropy coding) [3] from the prior MPEG/ITU-T video coding standards. As a new feature, the coding structure of HEVC has been extended from the traditional macroblock concept to an analogous block partitioning scheme with four different logical units: *coding tree unit* (*CTU*), *coding unit* (*CU*), *prediction unit* (*PU*), and *transform unit* (*TU*). For 4:2:0 color format, each of these consists of one luma and two chroma blocks that cover the corresponding block areas: *coding tree blocks* (*CTB*), *coding blocks* (*CB*), *prediction blocks* (*PB*), and *transform blocks* (*TB*). This new coding structure is the primary factor for the HEVC coding gain, but it also introduces majority of the computational overhead over its predecessors.

Each raw input video frame is partitioned into CTUs [42]. A CTU represents a root node of the quadtree and it can be up to 64 × 64 pixels at quadtree depth 0 ($h = 0$). It can be recursively split into four smaller square CUs until the maximum hierarchical depth (h_{MAX}) of the quadtree is reached. The size of the CTU can be defined as $2N_{MAX} \times 2N_{MAX}$, where $N_{MAX} \in \{8, 16, 32\}$ and the size of a CU as $2N \times 2N$, where $N \leq N_{MAX}$ and $N \in \{4, 8, 16, 32\}$, so $N_{MIN} = 4$ and $h_{MAX} = 4$. Each CU in the CTU is predicted and transformed individually. In intra coding, PUs and TUs are the same size as the parent CU, unless they are split further. For example, the smallest 8 × 8 CU can be split once into four 4 × 4 -pixel PUs whereas TUs can be split recursively until the minimum size of 4 × 4.

Actual block coding starts with a prediction phase, where an estimate of an image is generated by using predefined prediction methods. Intra prediction compresses blocks of a picture by exploiting its spatial redundancy. The prediction is subtracted from the original source image to generate a residual image.

Transform coding transforms the residual image from spatial domain to frequency domain coefficients. In frequency domain, high-frequency components of the video can be removed with quantization without significant quality loss since human eye is less sensitive to the high-frequency components.

In the last phase, the quantized transform coefficients and prediction modes are entropy coded to generate an encoded bitstream. In this step, the video signal is reduced to a series of syntax elements that contain properties of the blocks, including prediction modes, quantization parameters, transform coefficients, filter modes, and all other parameters required to describe how the video signal should be reconstructed by the decoder. These elements are ordered and compressed to generate an encoded video bitstream. Entropy coding method in HEVC is called CABAC, which is a lossless compression technique based on arithmetic coding. The compression is achieved by utilizing statistical properties of symbols, i.e., more frequent symbols are coded with less bits and less frequent symbols with more bits.

The encoding loop also includes decoder-side functionality such as IQ and IDCT phases, where quantized transform coefficients are dequantized and transformed back to the spatial domain. This generates a reconstructed version of the residual image that is added to the prediction to generate the final reconstructed image. In Intra HEVC encoders, reconstructed images are needed in the intra prediction phase, where spatially adjacent pixels are used to generate the predictions. Furthermore, reconstructed pictures correspond to the images generated and displayed by the decoder so they can also be used to measure the error introduced by compression.

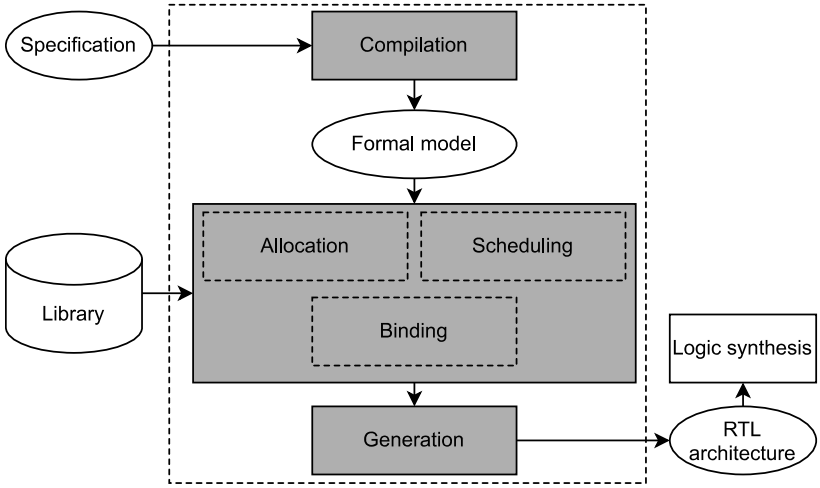


Figure 1: HLS design flow.

3 RELATED WORK

Since the advent of HEVC, a plethora of HW accelerators or complete HW encoders have been designed for it on FPGAs and ASICs. However, to the best of our knowledge, none of the existing HLS approaches [8]-[22] implemented a complete HEVC encoder but only individual HEVC coding tools. Furthermore, all of them only addressed data-intensive coding tools and passed over CABAC and other control-intensive parts.

3.1 Existing high-level synthesis approaches for HEVC

In the literature, HLS implementations have been presented for HEVC intra prediction [8]-[11] , DCT [12], IDCT [13], and interpolation [14]-[16]. HLS has also been applied in HEVC decoding [17] for intra prediction, dequantization, and inverse transformation. Our own HLS implementations for HEVC encoding are presented in [18]-[22]. These include intra prediction, DCT/DST, IDC/IDST, and two earlier versions for intra search acceleration, respectively.

3.2 Existing HEVC encoders on HW

Commercial HW encoders have been unveiled for HEVC, e.g., by NVIDIA (NVENC) [23], Xilinx (LogiCORE IP H.264/H.265 Video Codec Unit) [24], VITEC (e.g. MGW Ace Encoder) [25], ORIVISION (e.g. ZY-EH901) [26], and AJA (Corvid HEVC) [27]. However, the publicly available information of these confidential solutions tends to be limited so only academic works are considered in this paper. The existing academic HW HEVC encoders can be found in [28]-[37]. These can be categorized as: 1) FPGA implementations [28], [29], [32]; 2) FPGA/ASIC implementations [30], [31], [36]; and 3) ASIC implementations in [33]-[35] and [37]. All these implementations are characterized in detail in Section 8, where the performance of our proposal is compared with them.

4 METHODOLOGY

HLS seeks to improve productivity over traditional design methods by increasing design abstraction from RTL to behavioural level [38], [44], [45]. Various commercial HLS tools have been available on the market since the 1990s, but only recently they have started to gain adoption in the industry and academia [43]. The slow adoption rate has mainly stemmed from lower *quality of results* (*QoR*) than obtained with conventional HDL approaches. However, the latest HLS tool generations have substantially narrowed the QoR gap.

4.1 Motivation for high-level synthesis

Figure 1 depicts the conceptual diagram of the HLS design flow. The HLS tool accepts and compiles the algorithmic (behavioural) specification of the system, which is most often written in C/C++ or SystemC. The user specifies the target technology and provides micro-architectural constraints, such as directives for loop pipelining/unrolling and mapping of arrays to registers or memories. The HLS tool allocates the HW resources required by the specification, creates state machines, schedules the operations, and binds the operations to physical resources specified in the target technology library. Clock and reset are inserted by the HLS tool as per the designer's choice for the target clock frequency and type of reset. The generated structural RTL architecture description in VHDL or Verilog can then be used in the downstream logic synthesis SW, both for FPGA and ASIC designs.

In this work, we selected HLS over manual RTL coding for the following reasons:

1) **Application suitability**. HEVC coding is mostly a data-intensive process with relatively simple control structures. HLS has traditionally worked well with data-intensive designs, whereas implementing clock accurate control structures has been more challenging due to the lack of explicit time information in behavioural source code [44], [45]. However, our previous work [46] showed that even more demanding control structures can be described with the latest HLS tools. This motivated us to implement HEVC entropy encoding and all other control-intensive coding tools of HEVC with HLS. Moreover, even the recursive HEVC quad-tree coding structure can be implemented with HLS because the level of recursion is known at compile time.
2) **Algorithm and system architecture optimizations outperform micro-architectural optimizations.** Engineering hours should be spent where more gains can be reaped. Because of the immense complexity of HEVC, optimizing HEVC algorithm mapping to HW is encouraged. As the HEVC standard only defines the decoding process, there are several degrees of freedom to optimize nonnormative HEVC encoding tools. This leaves many design choices open at system level. By adopting HLS over RTL, most of the design effort can be concentrated on the system architecture, which tends to provide higher performance gains than optimizing the micro-architectures.
3) **Agile *design-space exploration* (*DSE*).** DSE refers to the systematic search of the pareto-optimal solutions with different performance-area trade-offs. In HLS, this can be as straightforward as choosing different loop unrolling/pipelining options in the graphical user interface or by embedding pragmas in the code. This is significantly faster than with hand-written RTL, where implementing each candidate solution requires extensive rewriting of the code. In practice, a comprehensive DSE cannot be even conducted with conventional HDL approaches, but the optimal micro-architecture needs to be calculated before actual implementation, which is a non-trivial task.

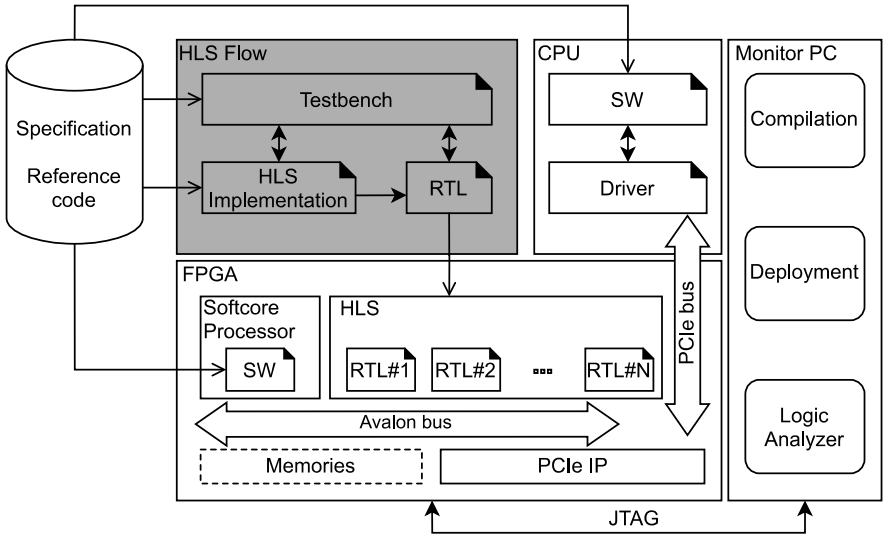


Figure 2: Proposed HLS development framework.

4) **Reduced verification effort.** Verification is one of the most time-consuming phases of any digital system project [47]. HLS boosts the verification process significantly as it allows verifying the automatically generated RTL code against the behavioural source code. In practise, the same testbench can be used to verify the functional correctness of the algorithm and the generated RTL code.
5) **Platform independency.** HLS also shines in the ease of adopting new target platforms for the system. When a new platform is selected with the HLS tool, a new RTL code for it is re-synthesized from the same source code. In contrast, with custom RTL, code restructuring is needed, e.g., if the state machine is changed to meet new clock constraints or if more resource sharing is required due to the limited capacity of the new platform.
6) **Increased productivity.** All previous advantages of HLS result in compelling productivity increase over custom RTL. Even though custom HDL approaches tend to achieve better performance with less resources, our recent literature survey [41] indicated that the average development time of an HLS project is only a third of that of the manual HDL project. The average productivity of HLS is also reported to be more than 4× as high in terms of the system performance with respect to the development time. In fact, our recent HLS implementations for HEVC algorithms [18]-[22] have achieved equivalent or even better performance than the respective works with hand-written RTL.

4.2 Proposed HLS development framework

Figure 2 illustrates the proposed development framework that is used to develop, verify, and deploy HLS implementations for HEVC coding. The Kvazaar C/C++ code is used as an input to the RTL code generation but also as a golden reference to verify the HLS implementation at both algorithmic and RTL levels. Constraints and features are defined in a separate specification document and they are used as input parameters for the HLS tool. The reference code can also be used as a fully functional SW implementation on an FPGA softcore CPU or on a server CPU.

The HLS generated RTLs are synthesized for FPGA. On-chip memories and an interface IP-block for the external CPU are instantiated manually. A monitor PC is used to compile the FPGA image, deploy the image and softcore CPU program, and analyze the internal FPGA signals while debugging.

The framework also allows both unit and system level testing. The unit testing is performed during the HLS flow by executing the testbench at algorithm and RTL levels. Most often only the algorithm level verification is needed, which is one of the largest benefits of HLS. Only some corner cases, e.g., type casting or vector overflows, might need verification between the RTL and algorithmic code. The same unit testing can also be performed on FPGA by generating the same test feed for the synthesized RTL on FPGA and the reference code on the softcore processor. In addition, a logic analyzer can be used to get a real-time view of signals on FPGA like in RTL simulation. The content of the memories, connected via Avalon bus, can also be validated with the softcore processor.

For system level verification, multiple independently verified RTL codes can be connected at top level. The process for verifying the created system is similar as that for a single unit. The corresponding reference codes are run on the softcore processor, a test feed is generated for the system, and the results are cross checked. The HW and SW co-processing also enables compilation time optimizations. As the system under verification becomes larger, the compilation time unavoidably increases. For testing purposes and faster compilation times, part of the HW system functionality can be replaced by executing the equivalent reference code on the softcore processor. This substitution allows the whole system to be executed on FPGA but does not necessary mean that the whole system is running on dedicated HW.

The PCIe connection between the FPGA and external CPU is not necessary for the FPGA development but it enables offloading processing from the CPU to the FPGA. The CPU driver development can be started even before the RTL synthesis, because the verification of CPU-FPGA interfacing can be conducted by running the whole system on the softcore processor.

The tools used in the design flow include:

1) Catapult Ultra Synthesis 10.5a for HLS;
2) ModelSim SE 10.6c for RTL simulation;
3) Quartus Prime 20.1.0 Standard edition including Signal Tap Logic Analyzer for FPGA synthesis (compilation), programming (deployment) and logic analyzing;
4) Eclipse IDE for C/C++ Developers 4.5.2 for programming the softcore CPU; and
5) Linux OS (Ubuntu) for developing the external CPU driver.

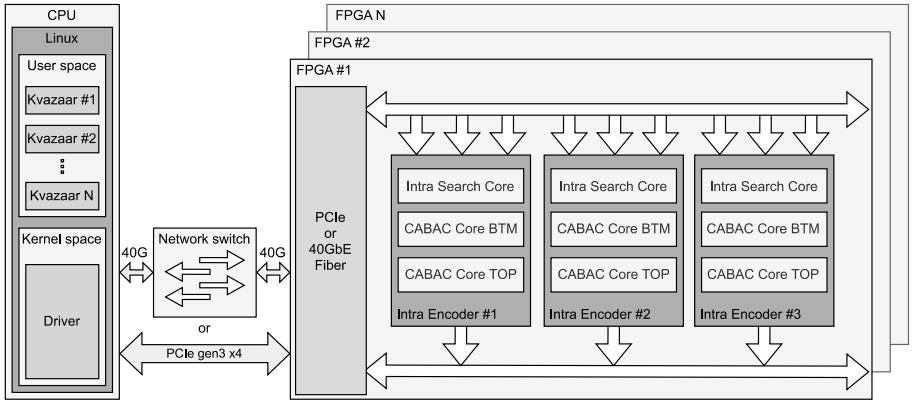


Figure 3: System architecture of the proposed HEVC Intra encoder.

5 PROPOSED ARCHITECTURE FOR INTRA HEVC ENCODER

Figure 3 presents the system architecture of the designed Intra HEVC encoder. The underlying SW/HW platform is composed of a server CPU and practically any number of FPGAs. Each FPGA is connected to a server over a network switch using *40 Gigabit Ethernet* (*40GbE*) link or directly via a *PCI Express* (*PCIe*) gen3 x4 bus.

The server runs Linux OS, e.g., CentOS or Ubuntu, and the processing is partitioned into user and kernel spaces. Kvazaar [5] is run in the user space, which can contain multiple Kvazaar SW encoder instances. A dedicated Linux-driver in the kernel space was developed to connect the Kvazaar SW instances to the FPGA. The driver can be shared between multiple Kvazaar instances, which allows parallel encoding of multiple video streams. The driver implements ioctl, read, and write system calls to provide Kvazaar with the data transfer functionality.

A single FPGA board may accommodate one or multiple HW Intra encoder instances, depending on its capacity. Each encoder instance is further divided into three independent units called *Intra Search Core* and *CABAC Core BTM*, and *CABAC Core TOP*.

The proposed system can be configured from SW-only encoding to pure HW encoding. On the server, encoding can be carried out with one or multiple Kvazaar SW instances in parallel. In HW encoding, functionality is partitioned between the server and FPGA(s) so that the server only takes care of 1) raw video input management; 2) HEVC stream initialization; 3) CTU parallelization; 4) offloading intra encoding task to FPGA(s); and; 5) reading the encoded CTU bitstream and related parameters from the FPGA.

In the following sections, the individual HW components are described in detail. All our design decisions were taken from the perspective of using HLS with the FPGA technology. The described functionality follows the HLS code almost directly and shows how everything was implemented with it. Catapult supports hierarchical HLS code, but it was not extensively used as the compilation time increases with design complexity. Instead,

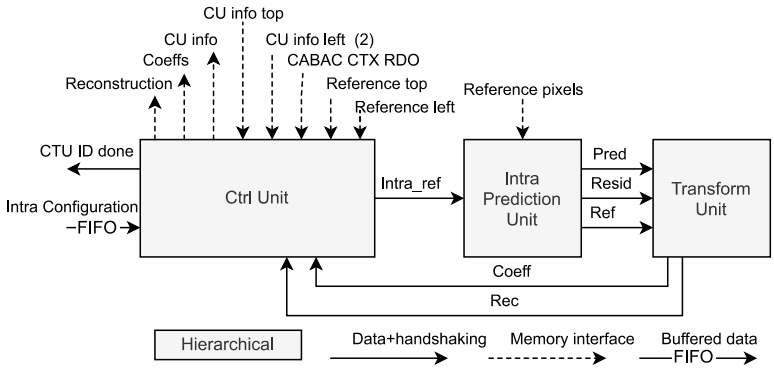


Figure 4: Top-level structure, interfacing, and connections of the Intra Search Core.

the architecture was split into logical parts that were compiled in separate projects and connected manually. Initial area, latency, and throughput reports of the HLS tool were used in DSE to find the design configuration (design partitioning, code structure, etc.) that meets area and performance requirements.

Supporting iterative design process is one of the major strengths of HLS. Furthermore, the flow of data is controlled by IO stalling and handshake signals, which are generated by the HLS tool automatically. This allows the parallel operation and synchronization of independent units. HLS also makes it easier to deploy these components as stand-alone IP-blocks in other system setups.

6 INTRA SEARCH CORE

Figure 4 depicts the *Intra Search Core* that performs HEVC intra search at CTU level. It consists of three sub-components: 1) *Ctrl Unit* for controlling the search, scheduling of parallel CTUs, and performing *mode decision* (*MD*) at CTU level; 2) *Intra Prediction Unit* for performing intra prediction and intra MD for PUs; and 3) *Transform Unit* for generating transform coefficients and reconstruction images for CBs.

A single *Intra Search Core* instance can cache 16 individual CTUs to on-chip memories for parallel processing. Every processing stage of the encoding pipeline works on the basis of CTU IDs, so the usage of the pipeline can be scheduled between 16 CTU IDs. The respective degree of parallelism would not be possible in a single CTU processing without breaking the dependencies between adjacent blocks [42]. The core interface includes memory interfaces to on-chip memories and direct data transmission with handshake signals. The *Intra Search Core* uses a 190 MHz clock in the proposed system.

The start signal and additional configuration data are provided for the *Intra Search Core* via the Intra Configuration channel. The configuration data consists of 1) CTU ID, with a value from 0 to 15; 2) depth limits for intra search, i.e., h = {1, 2, 3, 4}; 3) identification if the CTU is partially outside the frame; 4) *quantization parameter* (*QP*); and 5) a lambda value.

6.1 Memories and configuration for CTU intra search

The memories of the *Intra Search Core* are presented in Table 1. They are divided into 1) external memories, that are instantiated outside the Core, and are necessary for the CTU intra search process and search results;

Table 1: Memory name, location, size, and instances needed for a single Intra Search Core

			Name	Location	Bytes	Instances
Core			Reconstruction	External	98 304	1
			Coefficients (Coeffs)	External	196 608	1
			CU Info	External	16 384	1
			CU Info top	External	256	1
			CU Info left	External	256	2
			CABAC CTX RDO	External	256	1
			Reference top	External	4 096	1
			Reference left	External	4 096	1
			Reference pixels	External	98 304	1
	Ctrl		CU Info	Internal	4 096	2
			RDO Config	Internal	88	1
			Exec Config	Internal	32	1
			Instructions	Internal	23 552	1
			Instructions cache (Inst$)	Internal	14 720	1
			CTU Stack	Internal	840	1
		CU Stack	Coefficients (Coeffs)	Internal	131 072	1
			Reconstruction (Rec)	Internal	65 536	1
			Reconstruction left (Rec Left)	Internal	2 048	1
			Reconstruction top (Rec Top)	Internal	2 048	1
			Reconstruction top (Rec Top)	Internal	2 048	1
			Reconstruction left (Rec Left)	Internal	2 048	1
			Reconstruction top left (Rec TopLeft)	Internal	8 192	1
	Intra		Prediction references	Internal	76 544	1
	Transform	DCT	Transpose	Internal	8 192	1
		IDCT	Transpose	Internal	8 192	1
		Coeff Cost	Transpose	Internal	8 192	1
	Core				780 352	

and 2) internal memories, which are used for logic optimization and adjacent CU storage during the intra search. The table also shows the sizes and instantiation counts of these memories. If several units need access to a specific memory, multiple identical instances of it are generated. All memories are designed for the CTU ID based intra search and can accommodate 16 CTUs. For example, dividing the *Reconstruction* memory of size 98 304 bytes by 16 gives 6144 bytes per CTU, of which 64 × 64 = 4096 pixels are needed for the Y channel and ¼th of Y (1024 pixels) for both U and V channels in an 8-bit YUV420 format.

The external memories used for the intra search results are the following. The *Reconstruction* memory stores the final reconstructed CTU that is used as a reference when processing adjacent CTUs. The *Coeffs* memory is used to store the final entropy encoding coefficients of HEVC. The *CU info* memory is for the information of the final CTU structure, including the selected intra mode, chroma mode, depth, transform skip flag, and *coded block flag* (*CBF*) of each CU. It is also accessed when processing adjacent CTUs. The *CU info top* and *left* memories contain the bottom and right CU configurations of the neighbouring top and left CTU. The information contains the intra mode and CU depths. The CABAC *context* (*CTX*), needed for *rate–distortion optimization* (*RDO*), is stored in *CABAC CTX RDO* memory. This is not the full CABAC context, but it only contains the values needed by the *Ctrl Unit* for CU cost calculations when optimizing the CTU structuring.

Reference top and *left* memories store the reconstructed bottom and right pixels of neighbouring CTUs on top and left of the CTU in search. These pixels are used to generate the reference border pixels for intra prediction. The *Reference pixels* memory stores the original pixels of a CTU. It is used for calculating similarity between the original pixels and predictions. The internal memories are explained in the respective sections.

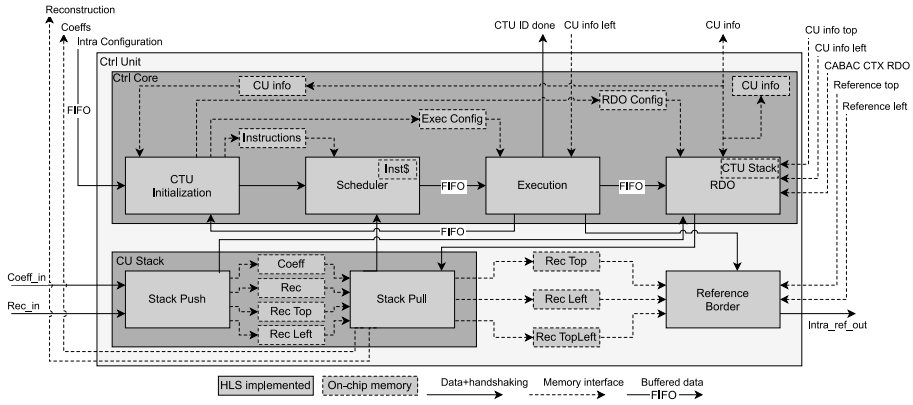


Figure 5: Internal structure of the hierarchical Ctrl Unit.

Table 2: Intra search instruction set

Instruction	Description
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

6.2 Intra search control (Ctrl Unit)

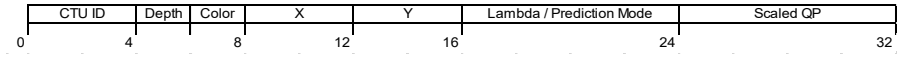
The structure of the *Ctrl Unit* is depicted in Figure 5. It consists of seven sub-units: 1) *CTU Initialization*, 2) *Scheduler*, 3) *Execution*, 4) *RDO*, 5) *Stack Push* and 6) *Pull*, and 7) *Reference Border* units, of which, units 1 to 4 belong to the *Ctrl Core*, and 5 and 6 to *CU Stack*.

The *Ctrl Core* is responsible for controlling the intra search. The HLS workflow made us implement the intra search control like in a CPU, i.e., the search is divided into smaller units that are performed sequentially. This approach also improves configurability and scalability. The basic operations of the implemented instructions are listed in Table 2.

The execution of these instructions is split into two parts. Some of the operations are performed in the *Execution* unit before the intra search pipeline and the rest after the pipeline in the *RDO* unit. Along with the type, each instruction contains operation parameters and a skip address. The parameters carry common processing information like block size and coordinates. The skip address is used for moving the program counter to the correct position if a quadtree search process is terminated earlier, i.e., the processing of the smaller CUs is skipped when all coefficients are zero.

The *CU Stack* is responsible for buffering the reconstructions, coefficients, and reconstructed borders of CUs for internal use. In addition, it updates the final reconstructions and coefficients in the external memories.

Table 3: Configuration bit vector used in intra prediction pipeline



6.2.1 CTU Search Initialization (CTU Initialization)

The *CTU Initialization* unit initializes two operations: 1) the luma search, which is started when the configuration information is received from the Intra Configuration channel; 2) the chroma reconstruction, which is started upon receiving the configuration information from the *Execution* unit after the whole luma search has been finished. The arbitration of these two configuration channels is achieved with HLS generated functionalities for channel stalling and non-blocking reading.

When the CTU is fully inside the frame, a pre-calculated program can be selected from the cache, which matches the intra search depth limits configuration. This reduces the start latency. Instead, when the CTU is only partially searched, the instructions for the CTU are generated during runtime and stored in the internal *Instructions* memory. Due to the tree structure of a CTU, the generation of these instructions was implemented in HLS code by using limited template recursion, which simplified the tracking of depth and CB coordinates in Z-order. Once the program initialization is ready, the starting address of the selected program is sent to the next unit and the CTU search configurations are stored into internal *Exec* and *RDO Config* memories.

With chroma reconstruction, the instructions are always generated during runtime based on CTU structure defined by the luma search. The chroma processing does not use the CMP instruction, because the reconstruction and coefficients of the chroma components are generated based on luma results.

6.2.2 Scheduler

The *Scheduler* unit schedules instructions when the *Intra Search Core* is processing multiple CTU IDs in parallel. After receiving the configuration, the *Scheduler* reads the instruction from the internal *Instructions* memory or from the predefined *Instructions cache (Inst$)* and stores it into an internal register. The register contains the latest instructions from all running processes, which can be either running or waiting. With this kind of mixed use of on-chip memories, look-up-tables, and registers, HLS improves code readability, as simple C-arrays can be used for each case, but the resource-dependent addressing and timing is generated based on the resource mapping. For all waiting instructions, the *Scheduler* calculates a priority number and selects the one with the highest priority. The selected instruction is sent to the *Execution* unit and the cached copy is changed to an running state. The state is changed back to waiting when the processing for the CB has finished. This procedure ensures that the processing of adjacent CBs complies with all data dependencies.

Due to the structure of the pipeline, CBs of different sizes move at different speeds. Larger CBs create congestion behind them and reduce efficiency. To minimize this, the *Scheduler* starts the processing of same size CBs with different CTU IDs in batches. This approach is only used for the IP instructions, because other instructions have very little effect on the pipeline and thus have a small, fixed priority.

6.2.3 Execution

The *Execution* unit starts processing CBs upon receiving IP instructions from the *Scheduler*. It builds a configuration vector, specified in Table 3, and sends it to the *Reference Border* unit. This vector contains all configuration parameters required throughout the pipeline until the *RDO* unit. The CTU ID, depth, color, and the

coordinates are generic parameters used in the coding pipeline. Lambda, which is derived from the QP value, is applied when the similarity of the reference and intra prediction is computed [41] and is then replaced by the selected prediction mode. The scaled QP is needed in the quantization phase and it is forwarded to the *RDO* unit for further processing.

The STR and CMP instructions require no processing in the *Execution* unit and are forwarded to the *RDO* unit. The END instructions are not forwarded, but when the program for luma search reaches the END instruction, the *Execution* unit sends a configuration for the *CTU Initialization* unit to start chroma reconstruction. When the program for chroma reconstruction reaches the END instruction, the unit sends a signal through the CTU ID done channel.

6.2.4 Rate–Distortion Optimization (RDO)

The *RDO* unit calculates the costs for CBs and compares them to select CUs for the final CTU configuration. The first instruction to arrive in the *RDO* unit is the STR instruction, which is used for resetting the CTU state of the specified CTU ID.

These results of an IP instruction include the selected intra mode, *Sum of Squared Differences* (*SSD*) calculated in the *Reconstruction* unit, and estimation of bits to code the coefficients calculated in the *Coefficient Cost* unit. The final cost for the CB is then calculated from the SSD, coefficient cost, current and previous CB configurations, and the content of the *CABAC RDO CTX* memory. The final cost is stored into the internal *CTU stack* memory. If the search process reaches the configured depth limit or the CB is an all-zero coefficient block, there is no need to continue search. The *CU Stack* unit is then notified to flush the coefficients and the reconstruction of the CB through the external interface. Chroma CBs are always flushed.

With a CMP instruction, the stored costs in the CTU stack are compared to achieve the best CU configuration. If a better configuration is found, the previously chosen configuration is overwritten via the *CU Stack* unit.

After each instruction, in addition to the flush-flag, the *RDO* unit sends an instruction completion signal with a skip-flag to the *Scheduler* via the *Stack Pull* unit. If this skip-flag is set in *RDO*, the *Scheduler* reads the skip address field from the cached instruction and moves the program counter to that address accordingly. Otherwise, the next instruction is read from the following address.

6.2.5 Build Reference Border (Reference Border)

The *Reference Border* unit generates the reference samples for the intra prediction from the external *Reference top* and *left* memories or from the internal *Rec Top*, *Left,* and *TopLeft* memories. If border pixels are not available, i.e., CU is located at the top and left border of the frame, the last available pixel or a constant value is used instead. The unit is configured with the block size and CU coordinates by the *Execution* unit. The referenced borders are built and sent to the *IP Ctrl* unit.

The three internal reconstruction memories are used to store the last reconstructed pixels from bottom and right borders, and one extra memory for all bottom right pixels of 4 × 4 blocks. Because the intra search of the coding tree works in Z-order from top to bottom, the reconstructed bottom and right border pixels of each CB can always overwrite previous pixels in the corresponding *x* and *y* coordinates in the *Rec Top* and *Left* memories. However, the corners need to be stored separately. This complies with all CB dependencies and minimizes the memory usage for the references.

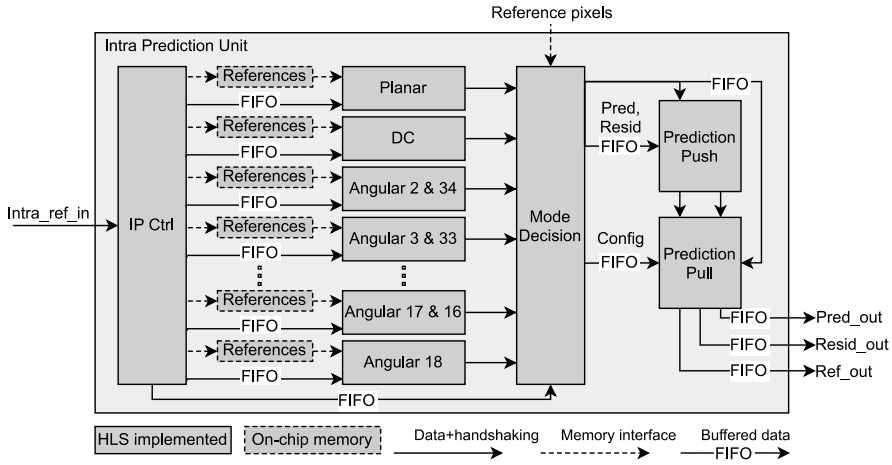


Figure 6: Internal structure of the hierarchical Intra Prediction Unit.

6.2.6 CU Stack

The *CU Stack* unit stores CUs temporarily before they are written into external memories. Buffering is needed because the final CU configuration can only be determined after all options have been compared. The *CU Stack* unit is built from *Stack Push* and *Stack Pull* units as well as from the internal *Coeffs*, *Rec*, and *Rec Top*, and *Left* memory modules between them.

The memories store one CU of each size for all CTU IDs. This way, all CUs have a reserved buffer slot assigned by their size and CTU ID. The adjacent CU of the same size in the same CTU overwrite the old one in the buffer. This policy follows the computational order of the CTU coding tree. The CUs are either flushed to the external memory or discarded before moving to the next one of the same size, according to instructions from the *RDO* unit.

6.2.7 Stack Push

The *Stack Push* unit receives the reconstruction pixels and the SSD values from the *Reconstruction* unit. In addition, it gets the quantized transform coefficients and coding cost estimation from the *Coefficient Cost* unit. It writes the pixels and coefficients to the reserved slots in the *Rec* and *Coeffs* memories and simultaneously collects pixels of the bottom and right borders to the corresponding *Rec Top* and *Left* memories. Lastly, the SSD and coding cost estimations are sent to the *RDO* unit to signal the completion of the CB process.

6.2.8 Stack Pull

The *Stack Pull* unit is on the other side of the memories. It has no direct connection to the *Stack Push* and it receives its configuration data from the *RDO* unit. Based on the flush-flag sent from the *RDO* unit, the *Stack Pull* unit starts reading the CU from the internal memories and writes it into the external memories. While writing

the CU, the *Stack Pull* unit performs two additional operations: transforming reconstruction from slices to rows and reordering coefficients to Z-order.

Reconstructions in the internal and external memories are stored as slices of 32 pixels. The external 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. The *Rec Top*, *Left* and *TopLeft* internal memories connected to the *Reference Border* unit are also updated during this process.

The coefficients are written in Z-order. 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 consecutive addresses.

6.3 Intra Prediction and Mode Decision (Intra Prediction Unit)

The *Intra Prediction Unit* calculates and selects the best prediction for a given CB. It does this by generating prediction images for all 35 modes and selecting the mode closest to the reference image. Figure 6 shows the internal structure of the *Intra Prediction Unit*. It consists of 23 sub-units: 1) *Intra Prediction Control (IP Ctrl)*; 2) *Mode Decision*; 3) *Prediction Push*; 4) *Prediction Pull*; and 5-23) 19 parallel prediction units for all 35 prediction modes (*Planar*, *DC,* and *Angular*). The reference samples for all prediction units are stored in the internal *References* memories that are located between the *IP Ctrl* and prediction units. The memories can hold reference data for up to 4 PBs at a time and allow pipelined processing of predictions.

6.3.1 Intra Prediction Control (IP Ctrl)

The *IP Ctrl* unit receives the PB configuration data and the reference samples from the *Reference Border* unit. A smoothing filter is used for the reference samples in the *IP Ctrl* unit to reduce contouring artifacts [41]. Depending on the mode and the CB, either filtered or unfiltered pixels are written to the *References* memories. Writing data to multiple memory instances is implemented in HLS code as an unrolled loop that iterates a static array of pointers and either filtered or unfiltered pixels are written.

After all pixels are read, filtered, and written to the memories, the *IP Ctrl* unit wakes up the prediction units and the *Mode Decision* unit by sending them their configuration data. Along with the common parameters of the PB size and an CTU ID, each prediction unit has own configuration parameters, which are dependent on the prediction mode, i.e., last pixels from top and left borders for planar prediction, a DC value for DC prediction, and an absolute angle for angular predictions.

The *References* memories for the prediction units are needed to pipeline the intra prediction so that the control unit can filter reference samples for the next PB, while the prediction units are still generating predictions for the previous PB. Synchronization and overflow protection between the *IP Ctrl* unit and prediction units are managed with handshaking signals in configuration channels.

6.3.2 Prediction units (Planar, DC, and Angular)

All prediction units operate in parallel and are configured to predict four pixels per clock cycle, i.e., 32 × 32 block is predicted in 256 cycles and 16 × 16 block in 64 cycles. HLS is used to generate a pipeline for the entire prediction process so that the unit starts outputting predictions for the next PB immediately after the previous one ends. The pipelining removes the initial latency from new predictions when configurations are received at

a constant rate. This is especially important with smaller CBs, as the latency without pipelining can exceed the number of cycles needed for outputting the actual prediction. Successful pipelining requires paying attention to data dependencies even with HLS. Furthermore, the performance could be easily increased by predicting more than four pixels in parallel, but the area/performance ratio of the selected approach was found sufficient for this work.

In addition, the angular predictions were split into the following three different modules according to direction of the prediction angle: positive angles 2-9 & 27-34, negative angles 11-25, and zero angles 10 & 26. This approach allowed removing unnecessary structures that are specifically needed for the specified angles. It also allowed reusing the units and operate configuration based. The corresponding horizontal and vertical prediction modes were also implemented in the same units. These two 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* units, except for mode 18, predict two modes simultaneously. For example, modes 2 and 34 are of equal distance from the middle, i.e., 18 - 2 = 34 - 18 so they are predicted simultaneously in one unit.

Planar, *DC*, and zero *Angular* prediction units use two memory instances each, whereas the rest of the *Angular* units use 13 parallel memory instances. The memories of the same prediction unit share a common write port that is controlled by the *IP Ctrl* unit. A single prediction unit utilizes all read ports in parallel to support predicting 4 pixels at a time.

6.3.3 Mode Decision

The *Mode Decision* unit selects the prediction mode for luma PBs by calculating and comparing *Sum of Absolute Differences (SAD)* and entropy coding costs of all candidate modes. SAD is used as a measure of image similarity whereas the entropy coding cost estimates the number of bits needed to code the prediction mode into bitstream. The entropy coding values are calculated by multiplying a fixed entropy cost with lambda. This offset tends to affect the MD when two predictions are close to each other [41].

For cost calculations, the unit receives four pixels from each prediction unit per cycle and it simultaneously reads the corresponding reference pixels from the external *Reference pixels* memory. The SAD is calculated in parallel for all modes, four pixels per cycle. The comparison of the 35 mode costs was implemented in HLS as a limited template recursion function that compares all mode costs in pairs and returns the best mode with its cost. The HLS code implements inputs from the prediction units with pointer arrays and reads them in unrolled loops, which helps with the code readability. In addition, loop unrolling does not break the synchronous reading of inputs during stalling. The unit is also fully pipelined with HLS, so that MD for next PB can start immediately after the previous one. The prediction data and corresponding reference pixels of each calculated mode are sent to the *Prediction Push* unit for buffering. In addition, the selection of the smallest mode cost is signaled to both *Prediction Push* and *Prediction Pull* units.

6.3.4 Prediction Buffering (Prediction Push & Prediction Pull)

The *Prediction Push* and *Prediction Pull* units 1) buffer all predictions while *Mode Decision* unit is selecting the prediction mode; 2) generate the residual image from the prediction and reference pictures; and 3) adjust the width of the pipeline data bus between four and 32 pixels. To implement them with HLS requires that all

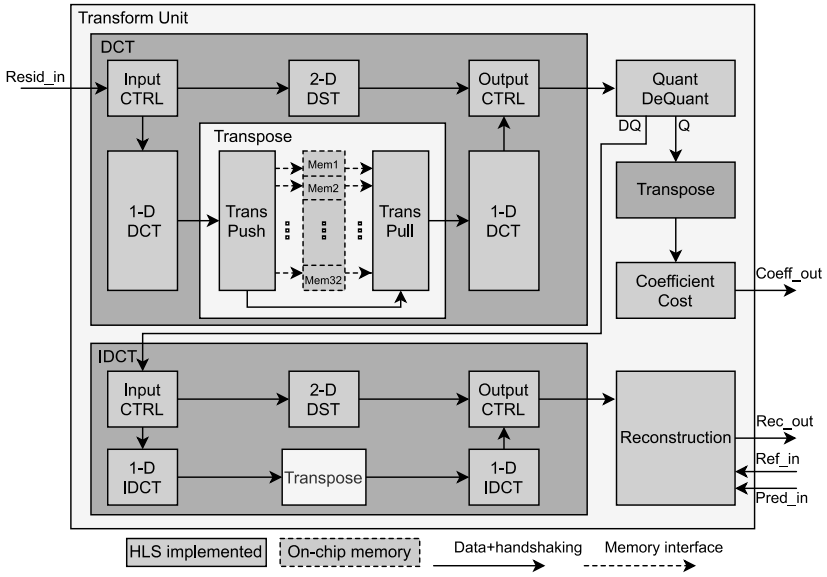


Figure 7: Internal structure of the hierarchical Transform Unit.

reference and the prediction pixels are stored in a FIFO, which is located between the *Mode Decision* unit and the *Prediction Push* unit. The size of the FIFO is 256 × 1153-bits (35 modes × 4 predicted pixels × 8-bits + 4 reference pixels × 8-bits + end bit = 1153-bits) to support the largest possible PBs. The main loop of the *Prediction Push* unit is pipelined to output data every cycle. It waits for the selected mode from the *Mode Decision* unit before it starts reading data from the FIFO and continues until the end bit of the current PB marks the completion. The mode is used as a parameter for shifting the input vector, so that correct prediction and reference pixels are forwarded to the *Prediction Pull* unit, which generates the residual and takes care of the output data width, i.e., a single line of 32 × 32 block, two lines of a 16 × 16 block, etc.

6.4 Transform Unit

The *Transform Unit* is depicted in Figure 7. It consists of 6 sub-units: 1) *Discrete Cosine Transform (DCT) unit*; 2) *Inverse Discrete Cosine Transform (IDCT) unit*; 3) merged *Quantization* and *DeQuantization (Quant DeQuant) unit*; 4) *Coefficient Cost unit*; 5) *Reconstruction unit*; and 6) *Transpose* unit. The *Transform Unit* has three main functions: 1) generate the quantized transform coefficients from the residual pixels provided by the *Intra Prediction Unit*; 2) create the reconstruction from the prediction and residual pixels, which are dequantized and transformed back to the spatial domain from the quantized transform coefficients; and 3) calculate the estimation for the coefficient coding cost and the similarity between the reference and reconstruction.

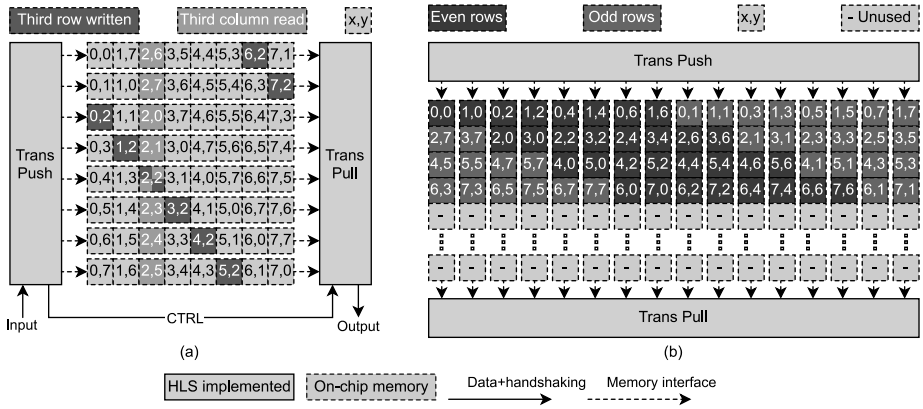


Figure 8: Visualization of pixel placement. (a) A slice containing a single row (example with an 8 × 8 variant of the transpose unit). (b) A slice containing multiple rows (example of 8 × 8 transpose with 16 × 16 variant of the transpose unit).

6.4.1 Transpose

The *Transpose* unit 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. It is used as a sub-unit in three different locations: between the 1-D transforms in *DCT* and *IDCT* units as well as between the *Quant DeQuant* and *Coefficient Cost* units.

Internally, it is composed of two parallel units, *Transpose Push* and *Transpose Pull*, and a storage array between them. The *Push* unit writes data to the memory array and the *Pull* unit reads it. The memory array is a collection of 32 parallel memory modules with individual read and write ports. For illustration purposes, an 8 × 8 variant of the *Transpose* unit is pictured in Figure 8 (a). It shows how the unit can transpose an 8 × 8 CB. The values in each cell represent the *x* and *y* coordinates of the 8 × 8 CB when all samples are written into the memory.

The actual input rate is 32 samples per cycle. Depending on the CB size, the slice contains from one to eight rows, as the slice can contain up to two 4 × 4 blocks. The correct memory instance is determined by rotationally shifting the bit slice left, according to the starting index *y* of the slice. In other words, the first slice of a row is not shifted at all, and the following slices are shifted depending on how many rows have been stored already. For example, in Figure 8 (a) the first index *y* of the third slice is two and in Figure 8 (b) the first index *y* of the third slice is four.

Simultaneously, the write address is determined by the number of rows in a slice and the corresponding index *x* of the sample. For example, in Figure 8 (a) each sample is written to the address *x*, and in Figure 8 (b) the samples are written to an address *x* divided by the number of rows written at once. The memory also operates as a circular buffer for a more pipelined operation. After the whole block is written to the memory, the *Pull* unit is notified that the memory is ready, and it can start reading the data.

The *Pull* unit is less complex, as it reads data in a consecutive order from all memory instances. As the slices were shifted left in the *Push* unit, the *Pull* unit reverses this operation. Furthermore, when a slice contains multiple rows, the rows are scattered as shown in Figure 8 (b). As the order only depends on the number of rows written at once, the reordering is simply implemented as a four-port multiplexer.

6.4.2 Discrete Cosine Transform (DCT)

The *DCT* unit is composed of three main parts: 1) two 32-point *DCT* units; 2) a separate 4-point *discrete sine transform* (*DST*) unit for 4 × 4 luma TBs; and 3) a *Transpose* unit for row-column transpositions between the *DCT* units. In addition, the design contains small control units for input and output. The *Input CTRL* unit reads the residual values and passes on the luma 4 × 4 TBs to the 4-point *2-D DST* unit, and all other TBs to the 32-point *DCT* unit. Output values from the first *1-D DCT* are row-column transposed in the *Transpose* unit and sent to the second *1-D DCT* unit for a complete 2-D transformation. The *Output CTRL* unit collects the results from either the *DST* or *DCT* unit.

The *1-D DCT* unit performs the transform in a three-step pipeline: 1) recursive even-odd decomposition, 2) multiplication between the transform matrices and odd vectors, and 3) 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 [48]. 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, where $N \in \{4, 8, 16, 32\}$, 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 implementation supports transform of 32/*N* rows/columns in parallel. For example, 8 × 8 TBs can be processed in only two parts. This is achieved by utilizing reordering of input and intermediate vectors and block size dependent look-up-tables for transform matrices [2].

Every stage is built to support max 32 × 32 TBs. The recursive even-odd decomposition can be reused for multiple rows/columns by reordering the reading of the input vector, so that the data flows in the recursive adder tree separately for each row/column. The resulting odd vectors are multiplied with the corresponding transform matrices. To reuse the multiplication stage with multiple rows/columns, the odd vectors are reordered to utilize the block size dependent look-up-tables. Individual products of matrix multiplication are finally added together and scaled to 16-bits, and the output vector is ordered back to the original order. HLS was vital for the implementation of the multi row/column functionality, from the perspective of parameterization and verification.

The 4 × 4 luma TBs are transformed in 4-point DST unit that operates in parallel with the 32-point *DCT* unit. The *DST* unit is composed of four parallel 1-D row-transform units 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 crosswire the outputs and inputs of the unit. The unit also supports transform skip, in which the transform phase is omitted. This is implemented by forwarding the residual pixels without any operations in the upper half of the 32-coefficient wide output vector.

6.4.3 Quantization (Quant) and Inverse Quantization (DeQuant)

The combined unit of *Quantization* and *DeQuantization* performs both quantization and dequantization of transform coefficients. Although they are different operations, they were implemented in one unit because they share the same overall structure and can have a shared control. This unit receives data from the *DCT* unit.

Quantized coefficients are forwarded to the *Coefficient Cost* unit and dequantized coefficients to the *IDCT* unit. The input configuration vector contains a 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.

6.4.4 Inverse Discrete Cosine Transform (IDCT)

The inverse transform of the *IDCT* unit shares the same top-level architecture as forward transform of the *DCT* unit. The *IDCT* unit was developed along with the *DCT* unit through the same HLS steps. The *1-D IDCT* unit also uses the Partial Butterfly algorithm implemented as a three-stage pipeline.

In the first stage, the input is multiplied with the transform matrices. The second stage has three addition/subtraction levels to compose the final even vector from the decomposed even and odd vectors. Lastly, the third stage combines the even and odd vectors and scales the final result to 16-bit signed residuals. To support the same multiple parallel rows with smaller block sizes as in the *DCT* unit, 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* unit and sent to the second *IDCT* unit to complete the 2-D transform.

In parallel with the *1-D IDCT* units, a separate 4-point *2-D IDST* unit is used for 4 × 4 luma CBs. The *IDST* unit performs the full 2-D transform internally without any external transpose. The support for transform skip was also added by forwarding the residuals without any operations in the upper half of the output vector.

6.4.5 Coefficient Cost

The *Coefficient Cost* unit calculates the estimated coding cost to encode the CB to the bitstream. The input coming from the *Quant DeQuant* unit is in transposed order, due to the transpose in the DCT unit. As the *Coefficient Cost* unit requires data in original order, an extra *Transform* unit was added between it and the *Quant DeQuant* unit.

Look-up-tables were used for the XY coordinates of the quantized transform coefficients to find the equivalent coefficient group and scan order index of the pixel. The estimation uses a linear model for the cost by utilizing five different parameters derived from the quantized transform coefficients: total sum of coefficients, number of nonzero coefficient groups, number of coefficients with value of zero or one, and the index number of the last nonzero coefficient. Different weights are predefined according to data gathered from CABAC for each parameter and for each CB size. The final cost estimation is calculated by multiplying each parameter with its weight and added together. This algorithm produces slightly worse results than CABAC, as it only estimates the cost of coding, which might reduce the encoding quality with certain CBs.

6.4.6 Reconstruction and SSD (Reconstruction)

The *Reconstruction* unit receives the reconstructed residual pixels from the *IDCT* unit. It also receives the reference and predicted pixels from the *Prediction Push* unit. The unit uses residual pixels and prediction pixels to generate the final reconstructed image as on the decoder side. The reference pixels are used to simultaneously calculate the 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 an overflow, the output is clipped to the maximum or minimum value. Pixels inside a PB have no dependencies

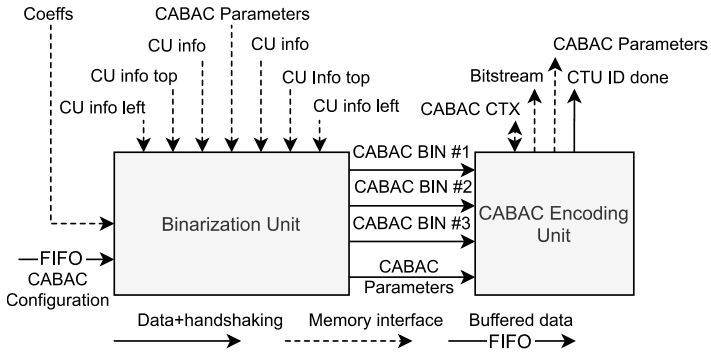


Figure 9: Top-level structure, interfacing, and connections of CABAC Core.

with each other, so any number of pixels can be calculated in parallel. As the output from the *IDCT* contains 32 coefficients, the *Reconstruction* unit was built to support that.

With the smallest 4 × 4 luma CBs, an output vector from the *IDCT* unit 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, the ability to duplicate the lower half of the prediction to the upper half was added to cover this special case.

SSD is calculated by deducting reconstruction from the original image, squaring the differences in pixel values, and adding them all together. The 4 × 4 luma CBs require SSD to be calculated in two halves, as separate SSD values are needed for both CBs. Utilizing the same structure 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*. The SSD is used as an image quality metric in intra coding, and the best CU configuration is selected as a function of the image quality and the number of consumed bits from the *Coefficient Cost* unit.

7 CABAC CORE

The *CABAC Core* is the top-level component for performing context-adaptive binary arithmetic coding of HEVC. A single *CABAC Core* can cache 16 individual CTUs to on-chip memories for pipelined processing. The coding of a CTU is started upon receiving CTU ID ready signal from the *Intra Search Core*. Contrary to the *Intra Search Core*, the *CABAC Core* does not process different CTU IDs in parallel in different pipeline stages, but it processes CTUs in first in first out order. This is because the CABAC process is serial in nature and the time used in binarization varies highly based on the contents of the coefficients, which in turn depends on the input video, quantization factor, and prediction accuracy of the intra search process. The core interface includes memory interfaces to on-chip memories and direct data transmission with handshake signals.

Table 4: Memory name, location, size, and instances needed for a single CABAC Core

		Name	Location	Bytes	Instances
Core		Coefficients (Coeffs)	External	196 608	1*
		CABAC Parameters	External	256	2
		CU Info	External	16 384	1+1*
		CU Info top	External	256	2
		CU Info left	External	256	2
		CABAC CTX	External	4 096	1
		Bitstream	External	32 768	1
	Binarization	Coefficient Groups (CGs)	Internal	4 096	1
	Core			58 880	

*Shared with Intra Search Core

The *CABAC Core* is presented in Figure 9. It consists of two components: 1) *Binarization Unit* for binarization of the CTU structure and the coefficients of each CB, and 2) *CABAC Encoding Unit* for binary arithmetic encoding of the binarized syntax elements to the bitstream and updating the parameters and CABAC states in corresponding tables.

The same iterative HLS process that was used for the Intra Search was also applied for CABAC. The initial design partitioning was done according to the CABAC functions in the Kvazaar reference code. The system-level architecture was further optimized during the implementation and DSE was used to improve area/performance figures.

The following sections present all HLS units implemented in the sub-hierarchical units with the required memories and configuration data for the CABAC process. The described functionality follows the HLS code almost directly and shows how the implementation of CABAC was made possible.

7.1 Memories and configuration for CTU CABAC process

The memories of the *CABAC Core* are presented in Table 4. They are divided into: 1) external memories, that are necessary for the binarization process, actual CABAC encoding, and the final HEVC bitstream; and 2) an internal memory, which is used to buffer the coefficient groups for the coefficient binarization. The table also shows the sizes and instantiation counts of these memories. If several units need access to a specific memory, multiple identical instances of it are generated. In the case of *Coeff* and *CU info* memories, the *CABAC Core* needs one identical instance in addition to the one instantiated in *Intra Search Core*. All memories are designed for caching 16 CTUs from the CTU ID based intra search. For example, dividing the *Coeffs* memory of size 196 608 bytes by 16 gives 12 228 bytes per CTU, and as the coefficients are 16-bit, divided by two bytes gives 6114 coefficients. Of these coefficients, 64 × 64 = 4096 are needed for the Y channel and ¼th of Y (1024 coefficients) for both U and V channels in an 8-bit YUV420 format..

The external interfaces used for the CABAC process are the following. The *Coeffs* memory stores the final coefficients of intra search and is read by the coefficient binarization process of CABAC. There are two *CABAC parameters* memories. One for receiving the current state and the other for storing the final state of the CABAC

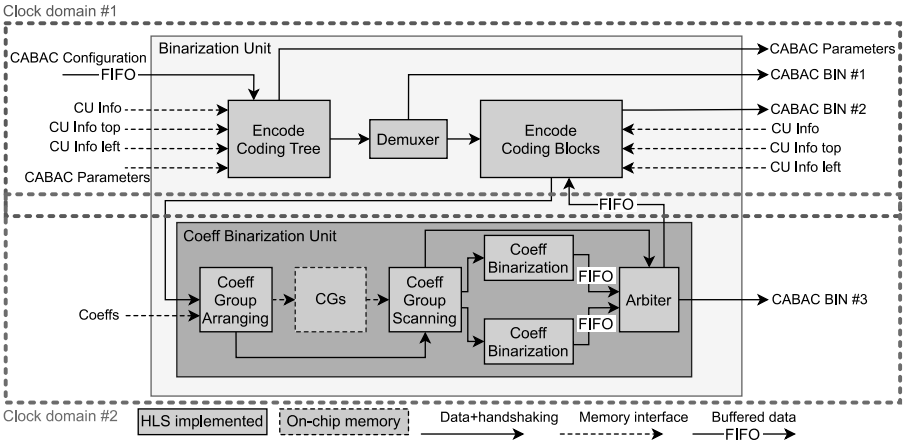


Figure 10: Internal structure of the hierarchical Binarization Unit.

variables bits_left, buffered_byte, num_buffered_bytes, low, and range, and bitstream variables current_data, current_bit, and zerocount, which are detailed later. *CU Info* contains the CTU configuration determined by intra search, and *CU Info top* and *CU Info left* the CU configuration of neighbouring CTUs on top and left. *CABAC CTX* memory contains the full state of all CABAC tables. A single syntax element is 7-bits, of which the lower 6-bits stores the *least probable symbol* (*LPS*) and the *most signifigant bit* (*MSB*) stores the *most probable symbol* (*MPS*). The memory uses 256 bytes per CTU ID for the 184 syntax elements [2]. The *Bitstream* memory stores the final HEVC bitstream. The internal memory is explained in the corresponding section later.

The start signal for the *CABAC Core* and additional configuration data is provided via the CABAC Configuration channel. The configuration data consists of 1) CTU ID, 2) frame size, 3) offset of tile, 4) *x* and *y* coordinates of the CTU, 5) current QP value, 6) initial QP value for delta QP, and 7) bottom right CU configuration of the top left neighbouring CTU.

7.2 Binarization Unit

The *Binarization Unit* is presented in Figure 10. It consists of seven sub-units: 1) *Encode Coding Tree*, 2) *Demuxer*, 3) *Encode Coding Blocks*, 4) *Coeff Group Arranging*, 5) *Coeff Group Scanning*, 6) *Coeff Binarization*, and 7) *Arbiter*. These units are responsible for the whole binarization process of a CTU.

The internal structure is divided into two clock domains: clock domain #1 is for non-time critical units; and clock domain #2 is for the more time critical process in the hierarchical *Coeff Binarization Unit*. By only overclocking the clock domain #2, the performance of the whole *Binarization Unit* increases without the need to route-optimize the clock domain #1. Clock crossing can be enabled manually via external dual clock FIFOs or automatically with internal clock crossing components generated by the HLS tool. The proposed system uses 190 MHz for clock domain #1 and 266MHz for clock domain #2.

The format of binarization commands sent via the CABAC BIN #NUM channels is presented in Table 5. The bit vector contains the necessary information to perform the bitstream encoding in the *CABAC Encoding Unit*.

Table 5: Bit vector used for binarization commands

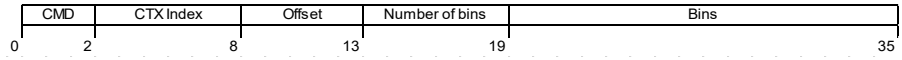


Table 6: CTX Index of CABAC Tables and the size of tables in bytes

CTX Index	CABAC Table	Bytes	CTX Index	CABAC Table	Bytes
0	sao_merge_flag_model	1	17	cu_ctx_last_x_chroma	15
1	sao_type_idx_model	1	18	cu_one_model_luma	16
2	split_flag_model	3	19	cu_one_model_chroma	18
3	intra_mode_model	1	20	cu_abs_model_luma	4
4	chroma_pred_model	2	21	cu_abs_model_chroma	2
5	inter_dir	5	22	cu_pred_mode_model	1
6	trans_subdiv_model	3	23	cu_skip_flag_model	3
7	qt_cbf_model_luma	4	24	cu_merge_idx_ext_model	1
8	qt_cbf_model_chroma	4	25	cu_merge_flag_ext_model	1
9	cu_qp_delta_abs	4	26	cu_transquant_bypass	1
10	part_size_model	4	27	cu_mvd_model	2
11	cu_sig_coeff_group_model	4	28	cu_ref_pic_model	2
12	cu_sig_model_luma	27	29	mvp_idx_model	2
13	cu_sig_model_chroma	15	30	cu_qt_root_cbf_model	1
14	cu_ctx_last_y_luma	15	31	transform_skip_model_luma	1
15	cu_ctx_last_y_chroma	15	32	transform_skip_model_chroma	1
16	cu_ctx_last_x_luma	15			

The CMD field is for initializing the CABAC encoding process with START command and CTX Index field is for the CTU ID. With STOP command in the CMD field the rest of the fields are not used, and the CMD initiates flushing of CABAC encoding results. For actual binarization commands, CMD is used to identify if the command is for a single BIN (binary symbol mapped to a syntax element), equiprobable (EP)_BIN or multiple EP_BINS (bypass-coded bins). The CTX Index field is used for identifying the correct CABAC table for the command and Offset field to identify the correct state value in the corresponding table. The last two fields are used for the number of bins and the actual bins value.

Table 6 presents the indexing order and size of each CABAC Table in the *CABAC CTX* memory. The binarization explained in the following sections complies with HEVC standard and is detailed in [2], [49].

7.2.1 Encode Coding Tree

The *Encode Coding Tree* unit is the main control unit of *CABAC Core*. It receives the configuration and the start signal for a CTU ID ready for binarization and CABAC encoding. When the process starts, the unit sends a start command to the CABAC BIN #1 channel through the *Demuxer* unit. This makes the *CABAC Encoding Unit* initialize the processing for a new CTU ID, which is used for indexing the correct *CABAC CTX* and *Bitstream* memories. In addition, the *Encode Coding Tree* unit reads the initial parameters from the *CABAC Parameters* memory according to the CTU ID and sends them to the *CABAC Encoding Unit* via a channel. The final write from the *Encode Coding Tree* unit, after the whole CTU is processed, is a stop command to the CABAC BIN #1 channel. This instructs the *CABAC Encoding Unit* to flush the results to external memories. To keep the commands in order during HLS scheduling, the data for the CABAC BIN #1 channel and configuration to *Encode Coding Blocks* are combined into a single channel. The *Demuxer* unit then directs the data to the correct unit marked by the least significant bit, which is only included in the intermediate channel between the two units.

The quadtree CTU configuration is processed using limited template recursion function. The depth in the tree is increased until it matches the configuration of the current CU or maximum depth is reached. The process then continues moving up in the tree and to the next location according to Z-order. During this process, all SplitFlags (split=1, no split=0)) are binarized as BIN commands using the split_flag_model as the CTX. The part mode is binarized for all CUs as part_mode $2N \times 2N$ (1), unless the 8 × 8 CU is split into four 4 × 4 blocks and part_mode $N \times N$ (0) is used. The part mode binarization is a BIN command with the part_size_model as the CTX. All bins here are written to CABAC BIN #1 channel. The unit is also responsible for configuring the *Encode Coding Blocks* when there is no longer a split or the max depth is reached. The configuration data contains the 1) CTU ID, 2) *x* and *y* coordinates of the CU in the frame, 3) the *x* and *y* coordinates of the CU in the CTU, 4) current depth in the coding tree, 5) current QP value, 6) initial QP value, 7) luma mode of the CU, 8) block size of the CU, and 9) the configuration of the neighbouring top left corner CU.

7.2.2 Encode Coding Blocks

The *Encode Coding Blocks* is responsible for binarizing intra coding units and transform units. Based on the configuration received from *Encode Coding Tree* unit, the intra prediction mode is first compared with three predictors to find if it is a *most probable mode* (*MPM*). The predictor list is constructed according to the left and top neighbouring CUs. Prev_intra_luma_pred_flag, mpm_idx, and rem_intra_luma_pred_mode are then binarized according to the predictor list using BIN command for prev_intra_luma_pred_flag, EP_BIN for mpm_idx, and EP_BINS for rem_intra_luma_pred_mode. All these binarizations use the intra_mode_model as the CTX. As chroma is reconstructed in the *Intra Search Core* by using the luma intra prediction mode, intra_chroma_pred_mode is simply binarized as 0 with a BIN command using chroma_pred_model as the CTX.

Next, the chroma CBFs cbf_cb and cbf_cr are binarized with BIN commands using qt_cbf_model_chroma as the CTX. The luma CBF, cbf_luma, is also binarized as a BIN command using qt_cbf_model_luma as the CTX. The CBF flags are read directly from the *CU Info* external memories.

If one of the CBFs indicates that there are coefficients to be coded, the QP delta is first binarized if needed. The absolute QP delta is calculated based on the current QP value and the reference QP value. The prefix is binarized with BIN commands and the suffix (QP delta > 4) is binarized an EP_BINS command. Finally, the qp_delta_sign_flag is binarized with an EP_BIN command. The CTX is used for QP delta is cu_qp_delta_abs.

During CBF binarization process, the *Coeff Binarization Unit* is configured according to the CBFs. The configuration consists of 1) CTU ID, 2) *x* and *y* coordinates of the CU in the CTU, 3) block size, and 4) color channel identifier. To keep the binarization commands in order with HLS between the *Encode Coding Blocks* and *Coeff Binarization Unit*, the *Encode Coding Blocks* waits for a feedback signal from the *Coeff Binarization Unit*. HLS stalls the internal pipeline with the blocking feedback read, until the *Arbiter* unit sends a ready signal. This is because all coefficients must be binarized for all color channels before the next CB can be started.

7.2.3 Coeff Group Arranging

Coeff Group Arranging unit is the first unit in the hierarchical *Coeff Binarization Unit*. This unit is responsible for reading the coefficients from the Z-order external memory and storing them in the internal *Coefficient Groups* (*CGs*) memory. The unit reads four coefficients from the external memory at a time and writes all 16 coefficients to the internal *CGs* memory at a time, so that the *Coeff Group Scanning* unit can read complete CGs at a time.

The reading of coefficients uses scan order specific look-up-tables for translating scan order indexes to XY-index. During reading, the XY-index of last nonzero CG and the XY-index and scan order index for the last nonzero coefficient are gathered. The scan mode depends on the block size and color component. The scan order is always diagonal, except for 4 × 4 and 8 × 8 luma and 4 × 4 chroma CBs. For these CBs, angular modes 6-14 cause the use of vertical scanning and angular modes 22-30 cause horizontal scanning. In addition, a flag is set for each CG if the CG has nonzero coefficients.

The coefficients of a CB are stored to the memory in two alternating locations to allow pipelined arranging of adjacent CBs. After the arranging, configuration data is sent to the *Coeff Group Scanning* unit via a channel. This configuration contains 1) the starting location of CGs in the internal memory, 2) block size, 3) color channel, 4) scan mode, 5) last nonzero coefficient position in scan order and raster order, 6) last nonzero CG in raster order, and 7) max 64-bit vector for identifying which CGs have coefficients.

7.2.4 Coeff Group Scanning

Before starting the CG scanning the last significant XY is first binarized in this unit. The binarization depends on the XY coordinates of the last nonzero coefficient. The prefixes are binarized first as BIN commands and the suffixes as EP_BINS if needed. The CTX for these depends on the coordinate and color channel and can be cu_ctx_last_y or _x for _luma and _chroma.

The *Coeff Group Scanning* unit is mainly responsible for reading the CGs from the internal memories in correct CG scan order by using a look-up-table according to the scan mode from the configuration. Diagonal is scanned in zigzag, horizontal from left to right and vertical from top to bottom. The scanning is performed in reverse CG and coefficient order starting from the last nonzero CG and last nonzero coefficient. The unit performs pre-processing during the scanning of each CG. The unit calculates the absolute value for each coefficient; counts the number of nonzero coefficients; generates a 16-bit vector for the coefficient signs; determines the index of the last *coefficient equal to 1* (*c1*) and the first index of a *coefficient equal or greater than 2* (*c2*); and counts the number of c1 values after the first c2 and the total number of c2 values. The configuration data is then sent to the correct *Coeff Binarization* unit. The configuration is a combination of the pre-processed data and the configuration from the *Coeff Group Arranging* unit.

An *Arbiter* unit is needed for the three different paths binarization commands can be sent from. After the last significant XY is binarized, the *Coeff Group Scanning* initializes the *Arbiter* unit to start reading data from the first *Coeff Binarization* unit, after which the unit is alternated.

7.2.5 Coeff Binarization

As the *Coeff Group Scanning* does most of the pre-processing, the structure of the *Coeff Binarization* is basically just comparing different values and coefficients to produce correct binarization. The loops that generate binarization commands for the coefficients go through all coefficients until they are binarized. The loops use the pre-processed values to determine if binarization is needed. The coefficient binarization is done in six different steps. Steps 1 to 4 use BIN commands and steps 5 and 6 used EP_BINS commands.

1) The coded_sub_block_flag is binarized as 0 or 1 depending on if the CG has coefficients, except the last nonzero CG and the first CG in scan order, which are known to be one. This step uses the cu_sig_coeff_group_model as the CTX.

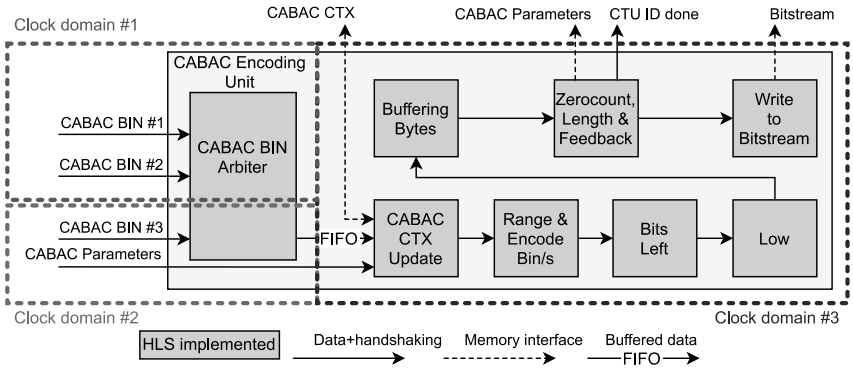


Figure 11: Internal structure of the hierarchical CABAC Encoding Unit.

2) Next, if the CG has coefficients, the sig_coeff_flag is signaled in a loop, and is binarized as 1 for all nonzero coefficients and 0 otherwise. This is signaled for all 16 coefficients of the first CG in scan order, skipping the first coefficient when rest of the coefficients are zero, or starting from index before the last nonzero coefficient with the last nonzero CG. This step uses either cu_sig_model_luma or cu_sig_model_chroma as the CTX.

3) Also, if the CG has coefficients, the coeff_abs_level_greater1_flag is signaled in a loop as 1 for all absolute coefficients that are greater than 1, 0 when they are one, and zero coefficients are not signaled. The binarization loop of coeff_abs_level_greater1_flag is ended if there are no longer nonzero coefficients in a CG or the maximum number of coeff_abs_level_greater1_flags reach the limit of eight. This step uses either cu_one_model_luma or cu_one_model_chroma as the CTX.

4) The coeff_abs_level_greater2_flag is signaled once for the first index of c2. It is binarized as 1 when the absolute coefficient is larger than 2 and as 0 if it is 1 or 2. This step uses either cu_abs_model_luma or cu_abs_model_chroma as the CTX.

5) The coeff_sign_flag is signaled for the coefficients if the CG contains nonzero coefficients. The bins sent is the vector pre-processed in the *Coeff Group Scanning* unit. This step uses the same CTX as step 3 or 4, depending on if coeff_abs_level_greater2_flag was signaled.

6) Finally, if the CG contains more than eight nonzero coefficients or has a coefficient larger than 1, the coeff_abs_level_remaining is signaled in a loop. The remaining coefficients are signaled according to the index of nonzero coefficients, absolute coefficient value, and Rice parameter. This step uses the same CTX as step 3 or 4, depending if coeff_abs_level_greater2_flag was signaled.

After all coefficient binarization steps are done for a CG, the unit sends a notification bit for the *Arbiter* unit to start forwarding the binarization commands from the second *Coeff Binarization* unit. The unit then immediately continues the processing of next CG if available. Because the time spent in *Coeff Binarization* depends on the encoded content and the number of coefficients to binarize, it can vary highly between different CGs. In a single unit there might be some latencies between consecutive binarization commands because the different loops are scheduled independently, and loop iterations without bin writes can exist. The loops are fully pipelined to process a single coefficient per cycle and break the loop depending on break conditions, but each

loop has initial latencies. Merging the loops with HLS at top level and having a common pipeline for each loop was not suitable because of data dependencies. To counter this, and to minimize the waiting of binarization commands in the *Arbiter* unit, the combination of pre-processing in the *Coeff Group Scanning* unit and the two parallel *Coeff Binarization* units was developed. This aims to keep the FIFOs before the *Arbiter* unit more evenly full and a steadier flow of binarization commands to the *CABAC Encoding Unit*.

7.3 CABAC Encoding Unit

The *CABAC Encoding Unit* is presented in Figure 11. It consists of eight sub-units: 1) *CABAC BIN Arbiter*, 2) *CABAC CTX Update*, 3) *Range & Encode Bin/s*, 4) *Bits Left*, 5) *Low*, 6) *Buffering Bytes*, 7) *Zerocount, Length & Feedback, and 8) Write to Bitstream*. These units are responsible for the whole CABAC encoding process of a CTU. The internal structure is divided into three clock domains. Clock domain #1 and #2 are the clock domains from the *Binarization Unit* and are used for reading the data from the corresponding CABAC BIN #NUM channel. The clock crossing between these two clock domains and the clock domain #3 is done in the *CABAC BIN Arbiter* automatically with internal clock crossing components with the HLS tool. Depending on the target device and routing results, the clock domain #3 can be set to a different frequency than the other two domains. In the proposed system, 266 MHz is used due to limitations in the device PLL.

The resulting design partitioning is the outcome of the iterative HLS development. The aim was to find best combination of performance and area usage. Excluding the *CABAC BIN Arbiter*, the partitioning is based on modifying specific CABAC variables per unit, including only necessary operations for modifying the corresponding variable. Each unit also forwards only the necessary data to the adjacent units. This helped simplifying data dependencies in each unit. Although not shown in the block diagram, all units send the needed data with a single write. If the write has multiple data sets or bytes for the next unit, this data is internally buffered and then sent in pieces. This helps the HLS tool to pipeline the top-level process in each unit better, as the there is no need to stall the pipeline for consecutive writes. This also keeps the whole CABAC pipeline fuller as some units might not write to output per read.

7.3.1 CABAC BIN Arbiter

The *CABAC BIN Arbiter* unit reads the CABAC BIN #NUM channels in reverse priority. For example, if there is data available in channel #2 and channel #3, the arbiter first reads all values from channel #3 before reading the values from #2. Furthermore, there is no internal buffering for the channels #1 and #2 during the clock crossing, which causes the writes from the corresponding unit to stall. This is to make sure the higher level binarization commands are made in order and no commands are missed before moving to the higher priority channel. The stalling of writes prevents the configurations to propagate, which in turn prevents the generation of new commands to higher priority channels before it is allowed. As the channel #3 generates the most commands and has the highest priority, there is also internal buffering during the clock crossing to compensate possible latencies in the data feed. HLS makes it easy to implement arbiter units that include prioritization, clock crossing, and internal buffering, as the complex functionality is generated by the HLS tool.

7.3.2 CABAC CTX Update

CABAC CTX update is the first unit in the actual CABAC encoding process. When the unit receives the START command, it reads the CABAC Parameters and initiates an internal initialization process where the CTU ID and

each parameter is updated to corresponding units in the CABAC pipeline. The vector with the CABAC parameters propagates through all units in the CABAC pipeline. Similarly, with the STOP command, the unit initiates a flushing process where each unit of the CABAC pipeline propagates the current corresponding CABAC parameter through all units in the CABAC pipeline.

The *CABAC CTX* is read and updated only when the unit receives a command for encoding a single BIN. This state is not altered with EP_BIN and EP_BINS commands. It first reads the state from the external *CABAC CTX* memory according to the CTX and offset field of the command. The state is used for reading the LPS and the MPS from next state look-up-tables. The state is updated to the external memory with the next LPS state if the bin is not equal to the first bit of the current state. Next MPS state is written otherwise. The current state is also forwarded in place of CTX and offset with the original command to the next unit.

7.3.3 Range & Encode Bin/s

The *Range & Encode Bin/s* unit is responsible for modifying the range parameter according to the state and command from the previous unit. The range is updated only when the CMD is for a single BIN. With CMDs EP_BIN and EP_BINS, the range is only used for forwarding data to the next unit.

With the BIN CMD, the range is updated by reading a value from the rangeLPS look-up-table [2] according to the LPS of the current state and the top two bits of the range. This table allows a multiplication-free approximation of the product range × LPS. This value is first subtracted from the current range value. If the bin is not equal to the MPS of the current state, the range is renormalized according to the top five bits of the value read from the rangeLPS table. This renormalization value is also read from a look-up-table that has 32 values. The value at index 0 is 6 and the value in the following indexes from 1 to 31 are $6 - (\log_2(\text{index}) + 1)$. The previously updated range is forwarded to the next unit with the renormalization value as the number of bits, after which the range is again updated to the value read from the rangeLPS table shifted left by the renormalization value. If the range was not renormalized and the previously updated range is less than 256, one bit zero is forwarded to the next unit. In this case the range is also shifted by one to the left. In addition, the order in which the low value is incremented and shifted in the next unit depends on if the range is renormalized. This is identified with the first bit in the forwarded data.

With the CMD EP_BIN, the current range is forwarded to the next unit when the bin is one with a bit size of one. One bit zero is sent when the bin is zero. With the CMD EP_BINS, the forwarding of data is divided into two sets according to the number of bins. If the number of bins is greater than eight, the first set of data forwarded is the 8-bits with the product of bins multiplied with the current range. The second set of data forwarded is the remaining number of bins with the product of remaining bins multiplied with the range.

7.3.4 Bits Left – Low – Buffering Bytes – Zerocount, Length & Feedback – Write to Bitstream

Bits Left unit is used for generating the bitmask for the *Low* unit and tracking the number of bits in the low value. The bits_left variable is updated according to number of bits received from the *Range & Encode Bin/s* unit. The bitmask is a 32-bit vector of all-ones, unless the bits_left value drops below 12-bits. In that case the bitmask is a 24-bit vector of all-ones, shifted right with the number of bits left. The bitmask is then forwarded to the *Low* unit, along with the data received from *Range & Encode Bin/s* unit.

The *Low* unit updates the low variable according to the data received from the previous units. The first bit of the data received is used to identify range renormalization. If the range was renormalized, the low value is first

incremented with it, after which the low value is shifted left according to number of bits in the data. If range was not renormalized, the low value is first shifted and then incremented. Finally, depending on the bitmask, the unit forwards the leading 9 bits of the low to the next unit and use the bitmask to zero them.

The *Buffering Bytes* unit is used for forwarding correct bytes according to the CABAC variables buffered_byte and num_buffered_bytes. The bytes written depends on the buffered_byte, the msb of the leading 9 bits, and the num_buffered_bytes. If the num_buffered_bytes is zero, the buffered_byte is set with the leading byte, and the num_buffered_bytes is set to 1 with no data sent. If the num_buffered_bytes is larger than zero, the first byte written is the buffered_byte + msb of leading 9 bits. The buffered_byte is then updated with the leading byte. The number of num_buffered_bytes is written next as 0 or 255, depending on the msb of the leading 9 bits. If the leading 9 bits represent a value of 255 only the num_buffered_bytes value is incremented.

The *Zerocount, Length & Feedback* unit tracks the number of consecutive zero bytes. If the leading 6 bits of the byte are zero, after two zero bytes, an emulation prevention 3-byte is written to the bitstream before the actual byte. The length variable is updated according to the number of bytes forwarded to the final bitstream. When the *CABAC Encoding Unit* receives the STOP CMD, the CABAC parameters from the previous units propagate to this unit. The unit stores the CABAC parameters to the external memory according to the CTU ID, which was specified during the START CMD. The unit also generates a done signal with the CTU ID, to inform the finished CABAC process.

The *Write to Bitstream* unit writes the received bytes into the external Bitstream memory one byte at a time. Because the 3-byte emulation prevention can cause two bytes to be written at once, the bytes are buffered in this unit and written one at a time. This way, the processing is not blocked in the previous unit by the memory write.

8 PERFORMANCE EVALUATION OF THE PROPOSED CLOUD ENCODING SYSTEM

Our proof-of-concept prototype was implemented on Nokia AirFrame Cloud Server equipped with 2.4 GHz dual 14-core Intel Xeon processors and two Intel PCIe FPGA accelerator cards. The applied FPGA chip was an Intel Arria 10 10AX115S2F45I1SG on Intel Arria 10 GX FPGA Development Kit, which supports both PCIe generation 3 x4 and 40GbE fiber connections. However, these two connections were not compiled together into a single project, but two compiled images exist for the same Intra Encoding unit, one with the DMA and the other with Ethernet blocks. The IP for either connection is provided by Intel Quartus Prime IP Catalog.

Figure 12 depicts the proposed system on FPGA. The Arria 10 FPGA can accommodate three Intra Encoding instances (#1-3). One encoder instance consists of a single *Intra Search Core* and two *CABAC Cores*, of which *CABAC Core BTM* is used for the bottom 0-7 CTU IDs and *CABAC Core TOP* for the top 8-15 CTU IDs. Other functional and memory instances, as well as their connections, are also drawn. Duplicate memories are illustrated with memory stacks, where the memory count indicates the number of parallel connections to the unit. Furthermore, some memories are also divided into btm and top instances, that can both store data for eight CTU IDs used by the respective *CABAC Core*.

The units implemented with VHDL include the Intra and CABAC config units, DMAs or Ethernet RX/TX for receiving and sending data, and CTU ID indexers. They are all directly connected to the Avalon bus.

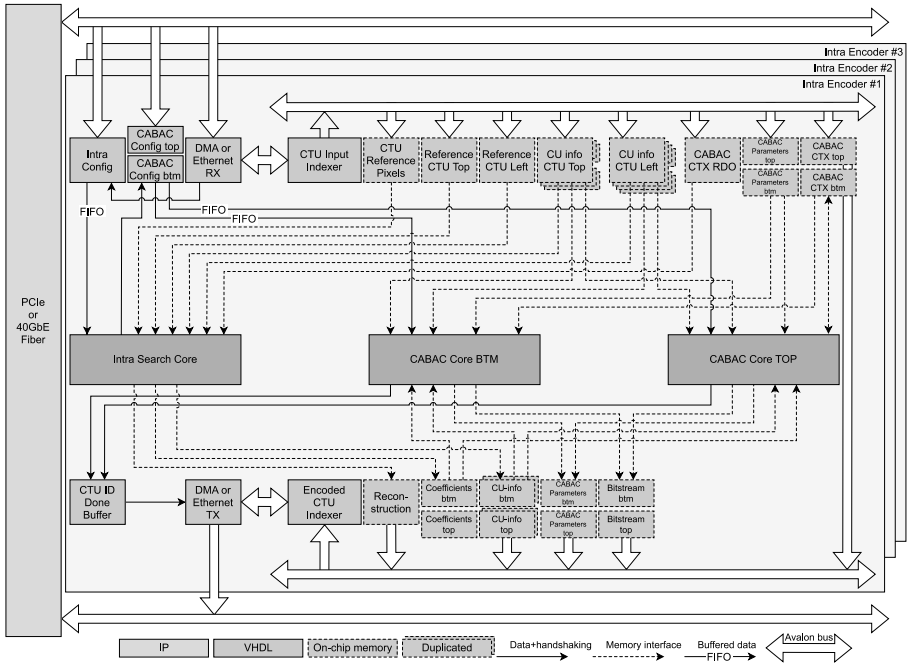


Figure 12: Proposed Intra Encoding System on FPGA.

8.1 FPGA Area Utilization

Table 7 reports the area utilization of the HW units on FPGA. The results are reported as *adaptive logic modules* (*ALMs*), which have the flexibility to implement 2.5 *logic elements* (LEs*)* of a classic 4-input LUT. A single DSP equals to two 18 × 19 multipliers or one 27 × 27 multiplier, which are automatically inferred by the Quartus Prime tool from the generated RTL.

A single Intra Encoding instance (see Figure 12) is made up of one *Intra Search Core* (90k ALMs with DSPs or 95k ALMs without DSPs in Intra Prediction), two *CABAC Cores* (22k ALMs), and Surrounding connectivity (13k ALMs). The area utilization of these modules is detailed in Table 7 (a) - (c), respectively. The sizes of internal buffers are not separately given, but are included in the total area of each hierarchical module.

In total, each Intra Encoding unit takes around 125k ALMs with DSPs or 130k ALMs without DSPs in Intra Prediction. The Arria 10 FPGA can include three Intra Encoding units of which one is implemented without DSPs in Intra Prediction, so the total area utilization is around (125k + 125k + 130k) ALMs = 380k ALMs as reported in Table 7 (d).

Table 7: Area utilization. (a) Intra Search Core. (b) CABAC Core. (c) Surrounding connectivity. (d) Intra Encoding System.

			ALMs	DSP Blocks
Single Intra Search Core			89 517	523
	Ctrl		7 806	
		CTU Initialization	339	
		Scheduler	871	
		Execution	233	
		RDO	1 767	
		Reference Border	617	
		Stack Push	1 607	
		Stack Pull	1 842	
	Intra Prediction		17 909	119
		IP Ctrl	469	
		Prediction blocks[1]	9 701	116
		Mode Decision[2]	5 132	3
		Prediction Push	1 791	
		Prediction Pull	612	
	Transform		63 407	404
		DCT	26 034	144
		IDCT	26 331	180
		Quant/DeQuant	5 021	64
		Transpose	3 675	
		Coefficient Cost	1 025	
		Reconstruction	1 298	16

[1]Without DSPs: 14 972 ALMs [2]Without DSPs: 5 408 ALMs

(a)

			ALMs	DSP Blocks
Single CABAC Core			10 899	3
	Binarization		8 435	2
		Encode Coding Tree	840	
		Demuxer	79	
		Encode Coding Blocks	1 751	2
		Coeff Group Arranging	1 232	
		Coeff Group Scanning	2 387	
		Coeff Binarization	752	
		Arbiter	211	
	CABAC Encoding		2 093	
		CABAC BIN Arbiter	439	
		CABAC CTX Update	100	
		Range & Encode Bin/s	221	1
		Bits Left	85	
		Low	351	
		Buffering Bytes	310	
		Zerocount, Length & Feedback	90	
		Write to Bitstream	74	

(b)

	ALMs	DSP Blocks
Surrounding connectivity	12719	

(c)

	ALMs	DSP Blocks
Proposed Intra Encoding System		
3× Intra Search Core*, 6× CABAC Core, 3× Surrounding connectivity	377 649	1468

**Intra Prediction DSPs are disabled for one Intra Search Core*

(d)

8.2 HEVC Coding Speed

Table 8 reports the coding speed of our Intra Encoding System over all 16 4K (3840×2160) test video sequences obtained from our UVG dataset [50]. The results are given for the base QP values of 22, 27, 32, and 37 as well as for ten different depth ranges of the HEVC quadtree. The depth ranges were denoted as "h_{min}-h_{max}", where h_{min} and h_{max} equal the minimum and maximum depths, respectively. For example, only 32 × 32 blocks (N = 32) are encoded with range "1-1" whereas $N \in \{4, 8, 16, 32\}$ with "1-4". The average frame rate values are color-coded for clarity: dark green denotes 120 fps or more, light green 60 - 120 fps, and orange below 60 fps.

The depth ranges of "1-3" and "2-3" highlighted in orange and blue comply with the fast and ultrafast presets of Kvazaar [5], [6]. With these presets, our system is able to encode 4K video over 80 fps in the worst case (QP=22) and over 100 fps on average. Respectively, the coding speed with the entire depth range ("1-4") exceeds 30 fps in each test case and is 85 fps on average.

The real-time presets of Kvazaar have intensively been optimized for speed [51]. In addition, the *sum of absolute transformed differences* (*SATD*) and accurate bin counting through CABAC were excluded from this proposal for simplicity. These optimizations add some overhead to coding efficiency, e.g., when compared with the HEVC reference encoder HM 16.23 [52] that implements practically all HEVC coding tools. With the UVG

Table 8: Encoding speed with the proposed Intra Encoding System (with the PCIe interface and two Arria 10 FPGAs)

	QP22										QP27									
4K Sequence [50]	1-1	1-2	1-3	1-4	2-2	2-3	2-4	3-3	3-4	4-4	1-1	1-2	1-3	1-4	2-2	2-3	2-4	3-3	3-4	4-4
Beauty	105	99	89	43	92	82	47	61	45	35	169	150	105	48	152	129	53	99	50	54
Bosphorus	149	155	110	56	171	140	63	126	58	59	184	173	120	73	169	155	86	133	72	60
CityAlley	171	129	105	51	160	132	58	120	54	59	170	162	122	85	180	155	102	133	80	60
FlowerFocus	171	156	105	46	161	130	51	102	48	56	188	163	123	89	184	158	111	133	83	60
FlowerKids	174	160	108	56	165	126	64	114	59	59	177	159	120	77	179	150	91	131	75	60
FlowerPan	135	115	86	41	113	88	46	73	46	42	170	156	98	48	160	125	54	109	52	60
HoneyBee	136	131	97	43	128	108	47	82	46	46	173	161	107	62	169	141	72	131	64	60
Jockey	170	153	103	46	144	125	50	96	48	52	178	174	114	76	182	149	91	133	74	60
Lips	108	101	93	43	93	84	47	63	46	35	173	164	106	46	156	132	51	101	49	56
RaceNight	115	108	91	42	104	93	46	79	46	40	166	164	107	55	176	134	62	129	57	59
ReadySteadyGo	158	155	103	52	156	125	58	112	55	58	181	170	115	65	183	146	76	128	66	60
RiverBank	142	122	94	49	129	102	55	91	54	47	159	137	111	58	155	137	65	123	59	59
ShakeNDry	120	111	90	41	112	96	45	77	45	43	163	134	101	49	157	129	55	118	52	59
SunBath	167	148	119	76	160	136	90	124	75	59	153	162	126	95	132	158	118	124	89	59
Twilight	176	156	107	53	167	133	60	124	56	60	184	162	134	97	174	160	119	132	87	60
YachtRide	149	161	102	52	164	129	59	119	56	59	182	161	114	66	176	146	77	132	68	60
Average fps	147	135	101	50	139	115	56	98	53	51	174	160	114	69	168	144	81	125	68	59

	QP32										QP37									
4K Sequence [50]	1-1	1-2	1-3	1-4	2-2	2-3	2-4	3-3	3-4	4-4	1-1	1-2	1-3	1-4	2-2	2-3	2-4	3-3	3-4	4-4
Beauty	155	167	121	91	178	148	112	131	84	60	188	177	175	167	181	175	162	133	116	60
Bosphorus	187	173	132	97	186	171	119	133	88	60	187	182	145	125	175	172	157	133	105	60
CityAlley	186	175	142	110	181	175	137	131	95	60	187	169	155	137	173	174	153	133	108	60
FlowerFocus	188	174	158	141	168	166	169	131	107	60	188	178	170	158	169	175	174	128	120	60
FlowerKids	181	175	133	94	171	160	114	133	85	60	190	170	148	114	170	168	135	131	97	60
FlowerPan	176	147	108	63	177	140	74	132	66	60	185	157	114	82	167	156	102	133	83	60
HoneyBee	184	165	124	95	167	133	119	133	87	60	181	177	122	118	180	175	149	132	102	60
Jockey	188	173	142	121	184	162	153	130	98	60	189	167	157	154	183	177	160	133	113	60
Lips	186	166	133	114	165	173	140	133	95	60	175	177	162	179	181	167	165	131	116	60
RaceNight	187	170	132	100	182	161	119	131	86	60	186	173	149	123	181	147	149	130	101	60
ReadySteadyGo	185	169	117	81	183	155	97	128	78	60	167	175	135	102	182	169	124	133	92	60
RiverBank	157	173	121	68	166	145	78	131	66	60	186	165	135	86	178	163	102	132	79	60
ShakeNDry	128	151	109	66	156	144	77	128	67	59	175	165	112	87	147	139	105	126	83	59
SunBath	174	166	132	116	168	161	145	125	100	59	169	153	138	134	166	151	158	130	110	59
Twilight	188	177	140	127	173	175	145	134	102	60	189	178	135	136	179	179	175	133	114	60
YachtRide	186	161	125	83	175	162	100	131	79	60	188	164	134	105	145	119	130	132	93	60
Average fps	178	168	130	98	174	159	119	131	87	60	184	171	143	126	173	163	144	132	103	60

Fast preset | Ultrafast preset | fps > 120 | 60 ≤ fps ≤ 120 | fps < 60

dataset, HM Intra encoder achieves 19% better coding efficiency (BD-rate [53]) over the proposed system with “1-4” depth range. However, our solution is also 5832× as fast as the single-threaded HM. Furthermore, the bit rate penalty does not stem from the HLS approach, but the coding tool optimizations performed in the source code.

Table 9 compares the average performance and resource consumption of our solution with the existing HEVC encoders on ASIC and FPGA. The results show that the proposed system consumes more resources than the respective FPGA approaches, but it is also able to attain higher performance with most depth ranges as shown in Table 8. The performance of our system could also be further scaled up by adding more FPGAs.

Table 9: Performance comparison with related work

	Miyazawa [28]	Atapattu [29]	Zhang [30]	Zhang [31]	Ding [32]	Tsai [33]	Zhu [34]	Huang [35]	Pastuszak [36]	Xu [37]	Proposed
Technology	FPGA/System	FPGA	FPGA/ASIC	FPGA/ASIC	FPGA	ASIC	ASIC	ASIC	FPGA/ASIC	ASIC	FPGA/System
Intra/Inter	x/x	x/-	x/-	x/-	x/-	x/x	x/-	x/-	x/-	x/x	x/-
FPGA performance	1080 @60fps	1080p @30fps	1080p @45fps	1080p @45fps	1080p @60fps	-	-	-	1080p @60fps	-	4K @60fps
System/ASIC performance	8K @60fps	-	4K @30fps	4K @30fps	-	8K @30fps	1080p @44fps	1080p @60fps	4K @30fps	4K @30fps	8K @30fps
Cells	-	-	-	195 883 ALUTs	63450 LUTs	-	-	-	93 184 ALUTs	-	377 649 ALMs
DSPs	-	-	-	1244 DSPs	721 DSPs	-	-	-	481 DSPs	-	1468 DSPs

Table 10: System, FPGA, and single CABAC Unit performance measured with the worst-case sequence

4K Sequence	Beauty	QP22									
		1-1	1-2	1-3	1-4	2-2	2-3	2-4	3-3	3-4	4-4
2× FPGA Encoding System	Mbins/s	4234	5240	5941	3371	4804	5189	3475	3745	3210	2316
	bins/c	15.9	19.7	22.3	12.7	18.1	19.5	13.1	14.1	12.1	8.7
1× FPGA Encoding System	Mbins/s	2117	2620	2971	1686	2402	2594	1738	1872	1605	1158
	bins/c	8.0	9.8	11.2	6.3	9.0	9.8	6.5	7.0	6.0	4.4
Single CABAC Core	Mbins/s	353	437	495	281	400	432	290	312	268	193
	bins/c	1.3	1.6	1.9	1.1	1.5	1.6	1.1	1.2	1.0	0.7

8.3 CABAC

To the best of our knowledge, this is the first work that has implemented HEVC CABAC with HLS. Therefore, the results are separately given for CABAC processing in Table 10 by reporting the achieved throughput in Mbins per second (Mbins/s) and bins per cycle (bin/c). The selected test sequence and QP represent the worst case in Table 8. The performance averages 4152 Mbins/s and 15.6 bins/c for the 2× FPGA Encoding System, 2076 Mbins/s and 7.8 bins/c for a single FPGA, and 346 Mbins/s and 1.3 bin/c for a single *CABAC Core*.

Our previous work [54] showed that the use of Ethernet connection decreased coding speed by 13% over an equivalent PCIe system. The reduction was mainly caused by the limited 20GbE connection on the host server. The work also used an earlier version of the *Intra Search Core*, and CABAC was still performed on CPU. Therefore, the coefficients (12 288 bytes) were transferred from FPGA, which accounted almost 2/3rd of the payload. A fully functional Intra HEVC encoder on FPGA and 40GbE enabled server would achieve the same performance as listed in Table 8, with scalability of adding virtually unlimited number of FPGAs to the network.

8.4 HLS Productivity

The HLS approach speeded up the HW implementation significantly over manual RTL coding and proved to work in such a highly complex system, from data-intensive coding tools like intra prediction, discrete transforms, and quantization to more control-oriented tools such as CABAC. The HSL part of the codebase includes 5 major versions, the total number of repository commits is 480, and the number of code lines exceeds 48k LoC of which C/C++ HLS code accounts for 41k LoC. Furthermore, the Verilog RTL generated from the HLS code exceeds 505k LoC, but it was not included in the repository. Manually written RTL takes up 7.5k LoC, which was only needed for instantiating and connecting the generated RTL (5.5k LoC) and VHDL units (2k LoC).

Altogether, we executed the RTL synthesis from Catapult over 10k times, i.e., the development included thousands of iterations and refinements. We estimate 21 person months effort was spent on the HLS implementation. Our conclusion is that without HLS it would have been very challenging to manage the project schedule and the overall design complexity.

9 CONCLUSIONS

This paper presented the architecture and HLS implementation of an embedded real-time 4K HEVC intra encoder. The HLS design approach made it possible to meet several design objectives at the same time: scalability as the number of server CPUs, accelerator FPGA boards, and HW encoder instances per FPGA as well as the flexibility to switch execution between SW and HW. The latter was found very beneficial at design time in protocol and interface verification as well as in developing the HLS synthesis and CPU SW simultaneously. The HLS synthesis took time to learn, but it speeded up the design iterations significantly. The productivity increase is challenging to justify, and HLS typically helps to improve the QoR more than absolute performance. However, our results show competitive video coding performance over related work, which indicate that the HLS tool was able to translate behavioural source code to structural RTL and optimize it efficiently. In particular, the implementation of CABAC is not trivial even with handwritten RTL. The HEVC encoder with its parallel HW instances is very complex as a whole and manually controlling all task allocations and scheduling would have been very laborious. This work proves that the shorter development time and better complexity control does not come at a cost of coding performance.

ACKNOWLEDGMENTS

This paper is part of the ADACORSA project that has received funding within the ECSEL JU in collaboration with the European Union's H2020 Framework Programme (H2020/2014-2020) and National Authorities, under grant agreement 876019. Other supporters include Nokia Foundation and the Finnish Foundation for Technology Promotion.

REFERENCES

[1] Cisco Systems. (Dec. 2018). *Cisco Visual Networking Index: Forecast and Trends 2017-2022*. Accessed on: June 15, 2021. [Online]. Available: http://web.archive.org/web/20181213105003/https:/www.cisco.com/c/en/us/solutions/collateral/service-provider/visual-networking-index-vni/white-paper-c11-741490.pdf

[2] *High Efficiency Video Coding*, document ITU-T Rec. H.265 and ISO/IEC 23008-2 (HEVC), ITU-T and ISO/IEC, Nov. 2019.

[3] G. J. Sullivan, J. R. Ohm, W. J. Han, and T. Wiegand, "Overview of the high efficiency video coding (HEVC) standard," *IEEE Trans. Circuits Syst. Video Technol.*, vol. 22, no. 12, pp. 1649-1668, Dec. 2012.

[4] *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.

[5] A. Lemmetti, M. Viitanen, A. Mercat, and J. Vanne, "Kvazaar 2.0: fast and efficient open-source HEVC inter encoder," *in Proc. ACM Multimedia Syst. Conf.*, Istanbul, Turkey, June 2020.

[6] Ultra Video Group. *Kvazaar HEVC encoder*. Accessed on: June 15, 2021. [Online]. Available: https://github.com/ultravideo/kvazaar

[7] MulticoreWare. *x265 HEVC Encoder / h.265 Video Codec*. Accessed on: June 15, 2021. [Online]. Available: https://bitbucket.org/multicoreware/x265/downloads

[8] E. Kalali and I. Hamzaoglu, "FPGA implementation of HEVC intra prediction using high-level synthesis," *in Proc. Int. Conf. Consum. Electronics - Berlin*, Berlin, Germany, Sept. 2016.

[9] Z. Cui, J. Xia, Y. Wang, G. Shi, and W. Yan, "Design of HEVC intra model decision based on Zynq," *in Proc. Int. Conf. Real-time Comput. Robot.*, Irkutsk, Russia, Aug. 2019.

[10] A. B. Atitallah and M. Kammoun, "High-level design of HEVC intra prediction algorithm," *in Proc. Int. Conf. Adv. Technol. Signal Image Process.*, Sousse, Tunisia, Sept. 2020.

[11] W. Chen, Q. He, S. Li, B. Xiao, M. Chen, and Z. Chai, "Parallel implementation of H.265 intra-frame coding based on FPGA heterogeneous platform," *in Proc. Int. Conf. High Perform. Comput. Commun.; Int. Conf. Smart City; Int. Conf. Data Sci. Syst.*, Yanuca Island, Cuvu, Fiji, Dec. 2020.

[12] B. Mohamed, A. Elsayed, O. Amin, E. Khafagy, M. Abdelrasoul, A. Shalaby, and M. S. Sayed, "High-level synthesis hardware implementation and verification of HEVC DCT on SoC-FPGA," *in Proc. Int. Comput. Eng. Conf.*, Cairo, Egypt, Dec. 2017.

[13] E. Kalali and I. Hamzaoglu, "FPGA implementations of HEVC Inverse DCT using high-level synthesis," *in Proc. Conf. Des. Architectures Signal Image Process.*, Krakow, Poland, Sept. 2015.

[14] F. A. Ghani, E. Kalali, and I. Hamzaoglu, “FPGA implementations of HEVC sub-pixel interpolation using high-level synthesis,” *in Proc. Int. Conf. Des. Technol. Integr. Syst. Nanoscale Era*, Istanbul, Turkey, Apr. 2016.

[15] W. Ahmad, J. Iqbal, M. Martina, and G. Masera, “High level synthesis based FPGA implementation of H.264/AVC sub-pixel luma interpolation filters,” *in Proc. Eur. Modelling Symp.*, Pisa, Italy, Nov. 2016.

[16] M. Pelcat, C. Bourrasset, L. Maggiani, and F. Berry, “Design productivity of a high level synthesis compiler versus HDL,” *in Proc. Int. Conf. Embedded Comput. Syst.: Architectures, Modeling and Simul.*, Agios Konstantinos, Greece, July 2017.

[17] M. Kammoun, A. Ahmed, A. Karim, and A. Rabie, “Case study of an HEVC decoder application using high-level synthesis: intraprediction, dequantization, and inverse transform blocks,” *J. Electronic Imaging*, vol. 28, no. 3, pp. 1-11, May 2019.

[18] 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.

[19] P. Sjövall, V. Viitamäki, J. Vanne, and T. D. Hämäläinen, “High-level synthesis implementation of HEVC 2-D DCT/DST on FPGA,” *in Proc. IEEE Int. Conf. Acoustics, Speech, Signal Process.*, New Orleans, Louisiana, USA, Mar. 2017.

[20] V. Viitamäki, P. Sjövall, J. Vanne, and T. D. Hämäläinen, “High-level synthesized 2-D IDCT/IDST implementation for HEVC codecs on FPGA,” *in Proc. IEEE Int. Symp. Circuits Syst.*, Baltimore, Maryland, USA, May 2017.

[21] P. Sjövall, V. Viitamäki, A. Oinonen J. Vanne, and T. D. Hämäläinen, “Kvazaar 4K HEVC intra encoder on FPGA accelerated air-frame server,” *in Proc. IEEE Int. Workshop Signal Process. Syst.*, Lorient, France, Oct. 2017.

[22] P. Sjövall, V. Viitamäki, J. Vanne, T. D. Hämäläinen, and A. Kulmala, “FPGA-powered 4K120p HEVC intra encoder,” *in Proc. IEEE Int. Symp. Circ. Syst.*, Florence, Italy, May 2018.

[23] NVIDIA: Video Codec SDK, Accessed on: Sept. 14, 2021. [Online]. Available: https://developer.nvidia.com/nvidia-video-codec-sdk

[24] Xilinx: Video Processing Subsystem, Accessed on: Sept. 14, 2021. [Online]. Available: https://www.xilinx.com/products/intellectual-property/v-vcu.html#overview

[25] VITEC: MGW Ace Encoder, Accessed on: Sept. 14, 2021. [Online]. Available: https://www.vitec.com/product/MGW-Ace

[26] ORIVISION: HDMI Video Encoder, Accessed on: Sept. 14, 2021. [Online]. Available: https://www.orivision.com.cn/collections/hdmi-video-encoder

[27] AJA: Corvid HEVC, Accessed on: Sept. 14, 2021. [Online]. Available:https://www.aja.com/products/corvid-hevc

[28] 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 Symp.*, San Jose, California, USA, Dec. 2013.

[29] 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. Appl.-specific Syst. Architectures Processors*, London, UK, July 2016.

[30] Y. Zhang and C. Lu, “Efficient algorithm adaptations and fully parallel hardware architecture of H.265/HEVC intra encoder,” *IEEE Trans. Circuits Syst. Video Technol.*, vol. 29, no. 11, Nov. 2019, pp. 3415-3429.

[31] Y. Zhang and C. Lu, “High-performance algorithm adaptations and hardware architecture for HEVC intra encoders,” *IEEE Trans. Circuits Syst. Video Technol.*, vol. 29, no. 7, July 2019, pp. 2138-2145.

[32] D. Ding, S. Wang, Z. Liu, and Q. Yuan, ”Real-time H.265/HEVC intra encoding with a configurable architecture on FPGA platform,” *Chinese J. Electron.*, vol. 28, no. 5, Sept. 2019, pp. 1008-1017.

[33] S.-F. Tsai, C.-H. Tsai, and L.-G. Chen, “Encoder Hardware Architecture for HEVC,” *High Efficiency Video Coding (HEVC)*, Springer, 2014, pp. 209-274

[34] 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 South Pacific Des. Automat. Conf.*, Singapore, Jan. 2014.

[35] X. Huang, H. Jia, B. Cai, C. Zhu, J. Liu, M. Yang, Don Xie, and W. Gao, “Fast algorithms and VLSI architecture design for HEVC intra-mode decision,” *J. Real-Time Image Process.*, vol. 12, 2015, pp. 285-302.

[36] 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.

[37] K. Xu, Yu. Li, B. Huang, X. Liu, H. Wang, Z. Wu, Z. Yan, X. Tu, T. Wu, and D. Zeng, ”A low-power 4096x2160@30fps H.265/HEVC video encoder for smart video surveillance,” *in Proc. Int. Symp. Low Power Electronics Des.*, New York, NY, USA, July 2018.

[38] P. Coussy, D. D. Gajski, M. Meredith, and A. Takach, “An introduction to high-level synthesis,” *IEEE Des. Test. Comput.*, vol. 26, no. 4, July 2009, pp. 8-17.

[39] Catapult: HLS-verification, Accessed on: June 20, 2021. [Online]. Available: https://www.mentor.com/hls-lp/catapult-high-level-synthesis/hls-verification

[40] S. Lahti, P. Sjövall, J. Vanne, and T. D. Hämäläinen, “Are we there yet? a study on the state of high-level synthesis,” *IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst.*, vol. 38, no. 5, May 2019, pp. 898-911.

[41] 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.

[42] I. K. Kim, J. Min, T. Lee, W. J. Han, and J. Park, “Block partitioning structure in the HEVC standard,” *IEEE Trans. Circuits Syst. Video Technol.*, vol. 22, no. 12, pp. 1697-1706, Dec. 2012.

[43] G. Martin and G. Smith, “High-level synthesis: Past present and future,” *in IEEE Des. Test. Comput.*, vol. 26, no. 4, July/Aug. 2009, pp.

18-25.

[44] H. Ren, “A brief introduction on contemporary high-level synthesis,” *in Proc. Int. Conf. IC Design Technol.*, Austin, Texas, USA, June 2014.

[45] J. Cong, B. Liu, S. Neuendorffer, J. Noguera, K. Vissers, and Z. Zhang, “High-level synthesis for FPGAs: from prototyping to deployment”, *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 30, no. 4, Apr. 2011, pp. 473-491.

[46] S. Lahti, J. Vanne, and T. D. Hämäläinen, “Designing a clock cycle accurate application with high-level synthesis,” *in Proc. Annu. Conf. IEEE Ind. Electronics Soc.*, Florence, Italy, Dec. 2016.

[47] H. Foster, *The 2018 Wilson Research Group Functional Verification Study*, Siemens, 2018, Accessed on: June 16, 2021. [Online]. Available : https://blogs.sw.siemens.com/verificationhorizons/2018/12/04/part-3-the-2018-wilson-research-group-functional-verification-study/

[48] 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, pp. 1029-1041, Dec. 2013.

[49] V. Sze and M. Detlev, “Entropy Coding in HEVC”, *High Efficiency Video Coding (HEVC)*, Springer, 2014, pp. 209-274

[50] A. Mercat, M. Viitanen, and J. Vanne, “UVG dataset: 50/120fps 4K sequences for video codec analysis and development,” *in Proc. ACM Multimedia Syst. Conf.*, Istanbul, Turkey, June 2020.

[51] A. Lemmetti, A. Koivula, M. Viitanen, J. Vanne, and T. D. Hämäläinen, “AVX2–optimized Kvazaar HEVC intra encoder,” in Proc. IEEE Int. Conf. Image Processing, Phoenix, Arizona, USA, Sept. 2016.

[52] Joint Collaborative Team on Video Coding Reference Software, ver. HM 16.23. Accessed on: Sept. 9, 2021. [Online]. Available: http://hevc.hhi.fraunhofer.de/

[53] G. Bjøntegaard, “Calculation of average PSNR differences between RD curves” *Document VCEG-M33*, Austin, TX, USA, pp. 1-4, Apr. 2001.

[54] P. Sjövall, A. Oinonen, M. Teuho, J. Vanne, and T. D. Hämäläinen, “Dynamic resource allocation for HEVC encoding in FPGA-accelerated SDN cloud,” *in Proc. IEEE Nordic Circuits Syst. Conf.*, Helsinki, Finland, Oct. 2019.