Tampere University

Jere Miettinen

HIGH-LEVEL SYNTHESIS IMPLEMENTATION OF HEVC MOTION ESTIMATION ON FPGA

Master of Science Thesis
Information Technology and
Communication Sciences
Ass. Prof. Jarno Vanne
Postdoc. Alexandre Mercat
April 2020

ABSTRACT

Jere Miettinen: High-Level Synthesis Implementation of HEVC Motion Estimation on FPGA
Master of Science Thesis
Tampere University
Master's Degree Programme in Electrical Engineering
April 2020

The need for transmitting high quality videos fast and effectively has increased in the recent years. Main reason for that is the increase in resolutions and frame rates, and the growing use of mobile devices and streaming. High Efficiency Video Coding (HEVC) is the latest video coding standard designed to respond those needs. HEVC achieves better compression compared to previous standards without compromising the video quality.

High-Level Synthesis (HLS) tools bring automation to the complex design processes and the designer can focus more on the algorithm functionality. The HLS design flow is on a higher abstraction layer compared to the traditional hardware design flows. Programming language such as C can be used instead of one of the hardware description languages (HDL) such as VHDL or Verilog. HLS was chosen for this Thesis, instead of traditional register transfer level (RTL) design, for faster and easier development.

Kvazaar is an open source HEVC video encoder developed at Tampere University. The encoding is done by removing temporal or spatial data redundancy. Motion estimation (ME) aims to reduce the temporal data redundancy. ME can be done using one of the various block matching algorithms (BMA), such as full search (FS) or hexagon-based search (HEXBS). The main goal of this Thesis was to evaluate Kvazaar's ME algorithms and then implement a ME accelerator on hardware using HLS. The accelerator was aimed for a Field Programmable Gate Array (FPGA) circuit.

ME is one of the most complex parts of the encoding and takes a significant amount of time of the whole encoding process and it is a good candidate for HW acceleration. FS algorithm was chosen for hardware acceleration in this Thesis and the hardware implementation was done using Catapult-C HLS tool. The accelerated algorithm was synthesized to Arria 10 FPGA platform and integrated as a part of Kvazaar's encoding process.

The synthesized Accelerator works on 150 MHz frequency and takes 18% of the available logic resources on Arria 10. It uses 6% of the available M20K memory elements and 6% of the platform's registers. The Accelerator achieved ×66.26 speedup compared to the software only algorithm. Once integrated to the Kvazaar's encoding process the speedup was still ×1.94. The drop in the speedup can be explained with the throughput limitations of the PCIe bus used in the communication between Kvazaar and the Arria 10 platform.

Keywords: High Efficiency Video Coding (HEVC), motion estimation (ME), High-Level Synthesis (HLS), Kvazaar, Field Programmable Gate Array (FPGA)

TIIVISTELMÄ

Jere Miettinen: HEVC-videokoodekin liikkeentunnistuksen toteutus FPGA-piirille C-kielestä syntetisoimalla
Diplomityö
Tampereen yliopisto
Sähkötekniikan diplomi-insinöörin tutkinto-ohjelma
Huhtikuu 2020

Tarve siirtää hyvälaatuista videokuvaa nopeasti ja tehokkaasti on lisääntynyt viime vuosina. Suurimmat syyt tähän ovat kasvaneet resoluutiot ja kuvataajuudet sekä lisääntynyt mobiililaitteiden käyttö ja striimauksen tarve. High Efficiency Video Coding (HEVC) on viimeisin videonpakkausstandardi, joka on suunniteltu vastaamaan edellä mainittuihin tarpeisiin. HEVC vähentää pakkauskompleksisuutta verrattuna aiempiin standardeihin tinkimättä kuitenkaan laadusta.

High-Level Synthesis (HLS) työkalut tuovat automaatiota monimutkaiseen suunnitteluprosessiin ja suunnittelija voi keskittyä paremmin algoritmin toiminnallisuuteen. HLS suunnitteluvuo on korkeammalla tasolla verrattuna perinteiseen RTL (register transfer level) suunnitteluun. Korkeamman tason ohjelmointikieltä, kuten C:tä, voidaan käyttää laitteistoläheisten kielten, kuten VHDL:n ja Verilogin, sijaan. Tässä työssä käytetään HLS työkaluja RTL työkalujen sijaan nopeamman ja helpomman kehityksen vuoksi.

Kvazaar on avoimen lähdekoodin HEVC videokoodekki, joka on kehitetty Tampereen Yliopistossa. Videon pakkaaminen perustuu joko ajallisen tai spatiaalisen dataylimäärän vähentämiseen. Liikkeenestimointi on ajallinen menetelmä ja sitä tehdään käyttämällä yhtä monista *block matching* algoritmeista (BMA), kuten *full search* (FS) tai *hexagon-based search* (HEXBS). Tämän työn päätavoitteena oli valita liikkeenestimointialgoritmi kiihdytettäväksi ja suunnitella ja toteuttaa kiihdytin ohjelmoitavalle logiikka piirille (FPGA) käyttäen HLS työkaluja.

Liikkeenestimointi on yksi pakkauksen laskennallisesti työläimmistä osista ja vie huomattavan osan koko pakkausajasta ja siksi se vaatii kiihdyttämistä. Kiihdytettäväksi algoritmiksi valittiin FS ja se toteutettiin käyttäen Catapult-C HLS työkalua. Kiihdytetty algoritmi syntetisoitiin Arria 10 piirille ja liitettiin osaksi Kvazaarin pakkausprosessia.

Syntetisoitu kiihdytin toimii 150 MHz taajuudella ja vie 18% käytettävissä olevasta logiikasta Arria 10 piirillä. Se käyttää 6% M20K muistilohkoista ja 6% käytettävissä olevista rekistereistä. Kiihdytin saavutti 66.26 kertaisen suhteellisen nopeuden verrattuna puhtaasti CPU (central processing unit) pohjaiseen algoritmiin. Kun kiihdytin oli liitetty Kvazaariin, suhteellinen nopeutus oli edelleen 1.94 kertainen. Pudotusta suhteellisessa nopeutuksessa selittää Kvazaarin ja kiihdyttimen kommunikointiin käytetyn PCIe väylän suorituskykyrajoitukset.

Avainsanat: High Efficiency Video Coding (HEVC), liikkeenestimointi, High-Level Synthesis (HLS), Kvazaar, Field Programmable Gate Array (FPGA)

PREFACE

This Master of Science Thesis was written as a part of research in the Faculty of Information Technology and Communication Sciences at Tampere University.

I would like to thank my examiners, Jarno Vanne and Alexandre Mercat for their input and guidance to write this Thesis. I want also to thank Panu Sjövall and my other co-workers who helped me during my work.

Finally, I would like to thank my family and Sabrina for their endless support during my work and writing.

Tampere, 27th April 2020

Jere Miettinen

CONTENTS

LIST OF SYMBOLS AND ABBREVIATIONS

AI	All Intra
ALM	Adaptive Logic Module
Avalon-MM	Avalon Memory-Mapped
AVC	Advanced Video Coding
BD-BR	Bjøntegaard-Delta Bit Rate
BMA	Block Matching Algorithm
CMOS	Complementary Metal Oxide Semiconductor
CPU	Central Processing Unit
CTCs	Common Test Conditions
CTU	Coding Tree Unit
CU	Coding Unit
DMA	Direct Memory Access
DSP	Digital Signal Processing
EPROM	Erasable Programmable Read Only Memory
FIFO	First-In-First-Out
FME	Fractional Motion Estimation
FPGA	Field Programmable Gate Array
FPS	Frames Per Second
FS	Full Search
full HD	full High Definition
HDL	Hardware Description Language
HEVC	High Efficiency Video Coding
HEXBS	Hexagon-Based Search
HDR	High Dynamic Range
HLS	High-Level Synthesis
HM	HEVC test Model
HSSI	High-Speed Serial Interface
II	Initiation Interval
IME	Integer Motion Estimation
JCT-VC	Joint Collaborative Team on Video Coding
LAB	Logic Array Block
LUT	Lookup Table
MAD	Mean Absolute Difference
MC	Motion Compensation
ME	Motion Estimation
MPEG	Moving Picture Experts Group
MSE	Mean Square Error
MV	Motion Vector
PCIe	Peripheral Component Interconnect express
PLL	Phased Locked Loop
PU	Prediction Unit
RAM	Random Access Memory
SAD	Sum of Absolute Differences
SoC	System-on-Chip
SRAM	Static Random Access Memory
TZ	Test Zone
VCEG	Video Coding Experts Group
VHDL	VHSIC Hardware Description Language

VHSIC	Very High Speed Integrated Circuit

1. INTRODUCTION

The amount of video in the global networking has been growing fast in the recent years and will take approximately 82% of the whole traffic by the end of 2021 [1]. This is mainly because of the increased resolutions and frame rates, the growth of mobile users, the increased use of *high dynamic range* (HDR) and 360˚ videos. The streaming has become popular and video calls are established as a part of everyday life. This generates need for transmitting videos as effectively as possible without compromising on the quality.

High Efficiency Video Coding (HEVC) is the newest video coding standard [2] and is designed to address those needs. HEVC encoders are complex and thus considerable candidates for hardware acceleration. Kvazaar is an open source HEVC video encoder developed as a part of the research in Ultra Video Group at Tampere University [3]. The encoding is done reducing either temporal or spatial data redundancy. *Motion estimation* (ME) aims to reduce the temporal redundancy. It is one of the most computationally heavy parts of Kvazaar encoder and it stands as a starting point for the *Field Programmable Gate Array* (FPGA) acceleration in this Thesis.

High-Level Synthesis (HLS) is an alternative to the traditional hardware design. It brings automation to the hardware design flow. The design process is done on a higher abstraction level and the implementation and verification times can be greatly reduced compared to the traditional *register transfer level* (RTL) design. HLS is used in this Thesis to implement a ME accelerator for Kvazaar. The main goals of this Thesis are to choose a suitable ME algorithm to accelerate, design an FPGA accelerator using HLS tools for the chosen algorithm and finally integrate the designed accelerator as a part of Kvazaar's encoding process.

The Thesis is organized as follows: HLS, FPGAs, HEVC, Kvazaar, and related work are introduced in Chapter 2. In Chapter 3, the inter picture prediction and ME in HEVC are discussed. Chapter 4 presents the methodology, such as evaluating tools and design method. Also test materials are discussed. Chapters 5 and 6 are reserved for the hardware specification and implementation, respectively. The results are presented and discussed in Chapter 7. Chapter 8 concludes the Thesis.

2. BACKGROUND

This chapter introduces the main topics of this Thesis. Basic background information of High-Level Synthesis (HLS) and its tools, Field Programmable Gate Array (FPGA) circuits, Kvazaar High Efficiency Video Coding (HEVC) encoder and its accelerated intra encoder are given. Related work is also discussed.

2.1 High-Level Synthesis (HLS)

The concept of High-Level Synthesis (HLS) brings more automation to hardware design flow. The main purpose is to generate functioning register transfer level (RTL) description from higher abstraction programming languages [4]. HLS synthesis tools generate synthesizable *hardware description language* (HDL), such as *Very High Speed Integrated Circuit* (VHSIC) *Hardware Description Language* (VHDL) [5] or Verilog, that allows hardware designers to focus on the functionality of the design.

As hardware systems and applications grow bigger and more complex, the design and verification times increase significantly. The cost of using standard RTL development tools becomes too high. HLS promises to reduce design and verification times by needing less detail for the design specifics, e.g. the need for specifying a clock, and automized generation of RTL structures based on the targeted technology [4]. Using HLS, the designs are more generic because of the higher abstraction level. This means that the same algorithm could be easier used on different platforms as HLS takes care of the platform specific design constraints.

An important thing to consider when using HLS is bit accurate data types. This becomes important when modeling hardware directly from C or C++, as they only offer data types with limited widths [4]. Mentor Graphics has developed their own standardized bit accurate data type, which is called Algorithmic-C data types. Another option for the bit accurate data types is SystemC types. However, it has various limitations compared to Algorithmic-C data types, such as slow execution times. Therefore, the Algorithmic-C data types are used in this Thesis.

The most commonly used Algorithmic-C data types are signed and unsigned integers [4]. They allow the designer to model bit vectors with a constant bit width and a sign. Algorithmic-C allows also the use of fixed-point arithmetic with its fixed-point data type. This is something what cannot be done automatically with regular RTL design methods and is a big advantage of HLS. All the basic logical and arithmetical operators are included in the Algorithmic-C standard. The Algorithmic-C offers also a set of built-in methods, such

as slice read, and slice write. These are used to get access to a specific set of bits within a variable. In addition, the bitwise operators are designed so that there is no loss of precision.

The first step of HLS is the analysis of the written algorithm [4]. This step forms the dependencies within the algorithm, and in which order the operations must be executed. The results are illustrated as data flow graphs. The next step is resource allocation where the operators are mapped to the hardware components of the targeted platform. Already at this point, the required area and latency are known. The timing of the designed system is determined in the final step called scheduling. During scheduling, the exact timing of the system is decided. This means deciding which operation to execute in what clock cycle. In practice this means that the HLS tool adds registers to the design between processes according to the assigned clock frequency. The operations are assigned to a specific clock cycle based on the critical path of the design and registers are put in between to reduce delay when needed.

HLS introduces also the concept of loop optimization, which brings parallelism to the design. There are two ways of implementing it, loop pipelining and loop unrolling. Loop pipelining means that a new iteration of a loop can be started before the previous one has finished. Loop pipelining is controlled with *Initiation Interval* (II), which determines how many clock cycles are waited before the next cycle is started. Loop unrolling, on the other hand, means how many parallel duplications there are for the loop. The loops can be fully or partially unrolled. Partial loop unrolling means, that specified number of iterations are started at once and full loop unrolling means that all iterations are started at once.

2.1.1 HLS design flow with Catapult-C

Catapult-C is an HLS platform offered by Mentor Graphics. It makes possible to get RTL logic description automatically from industry-standard C/C++ and SystemC languages [6]. The key features for Catapult-C are functioning RTL, fast verification time and power optimization. It promises 80% less code writing, easier debugging and 80% less verification cost compared to the traditional RTL design. The output from Catapult-C is VHDL and/or Verilog code.

The HLS design flow with Catapult-C is simpler than with traditional RTL design tools. Figure 2.1 illustrates the design flow with Catapult-C tool from the specification of the system to the synthesized hardware on FPGA. The flow is simpler and faster to execute

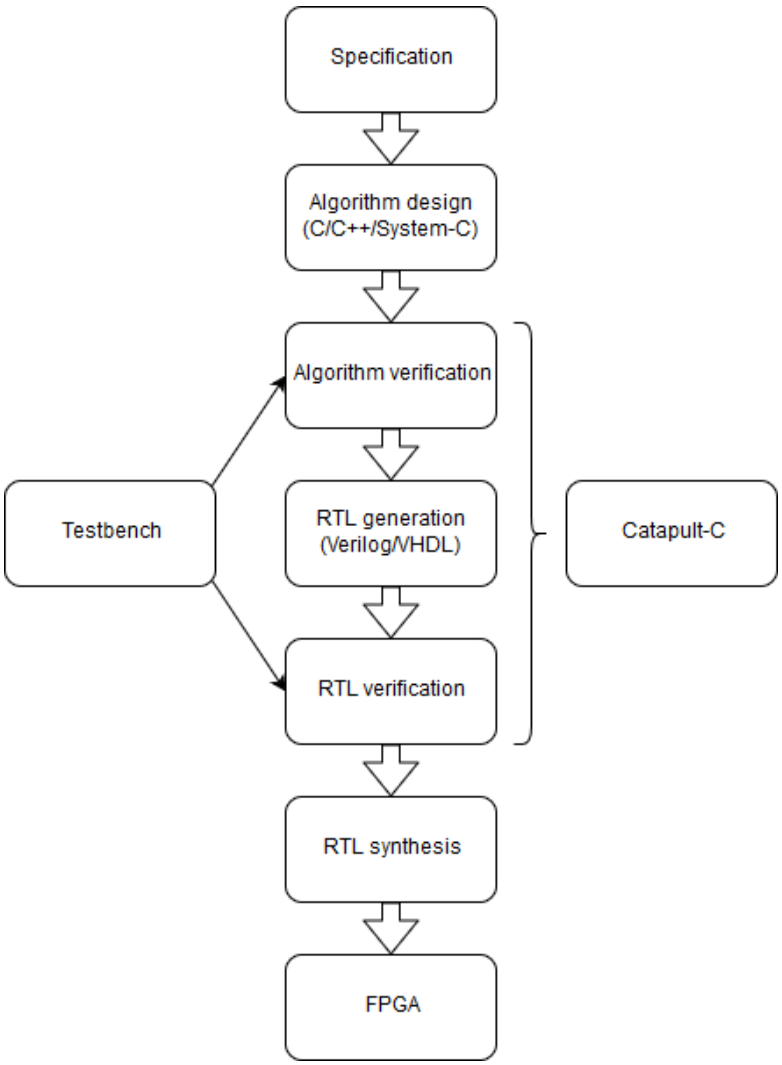

Figure 2.1 *HLS design flow.*

than with traditional hardware design flow. The same results as with the traditional hardware design flow can be achieved with HLS in a matter of hours instead of days or weeks.

The flow starts from the specification of the algorithm and its functionality. The untimed algorithm is then written with the chosen high abstraction level programming language. At this stage there are no specifications for clocks and reset signals. The functionality of the design is tested with a test bench, which is written in the same language as the algorithm. Once the functionality is tested, the RTL is generated.

Catapult-C also offers automation for RTL verification [6]. The same test bench used for functionality testing is used also for generating the stimulus and testing the RTL. During the RTL simulation, the RTL functionality is verified against the untimed algorithm. After the verified RTL description is obtained, the VHDL or Verilog code is transferred to the platform specific synthesis tool, i.e. Intel Quartus, and synthesized to the target platform.

```
void accumulator(ac_int<32, true> input[4], ac_int<32, true> &output)
{
  ac_int<32, true> accumulate = 0;
  for(ac_int<32, true> i = 0; i < 4; i++){
    accumulate += input[i];
  }
  output = accumulate;
}
```

Listing 2.1 *Accumulator example.*

Listing 2.1 shows an example of hardware design using Catapult-C. It accumulates the values from the input table and assigns the result to the output variable. The example does not contain control signals, such as clocks or resets, as Catapult-C adds them during the workflow.

Catapult-C also offers tools to manipulate the memories used in the design [4]. There are three important parameters which are used to simplify the memory architecture, and in the best case, greatly decrease the needed memory accesses. These parameters are word width, block size, and interleave.

Word width sets the width of each memory location, allowing data to be stored in parallel to the memories [4]. The parameter allows Catapult-C to automatically combine multiple reads or writes to the memory, without any manual coding. Left image on Figure 2.2 illustrates the block size parameter, which determines the number of blocks the memory is divided into. In the example, the block size is set three, which then divides the nine memory locations into three smaller memories, with three memory locations in each.

Right side of Figure 2.2 demonstrates how memory interleaving works. The same memory is now interleaved with three. The memory elements are stored again into three different memories, but this time the order is changed so that every third element is stored into the same memory. Memory interleaving and block size changing gives the designer

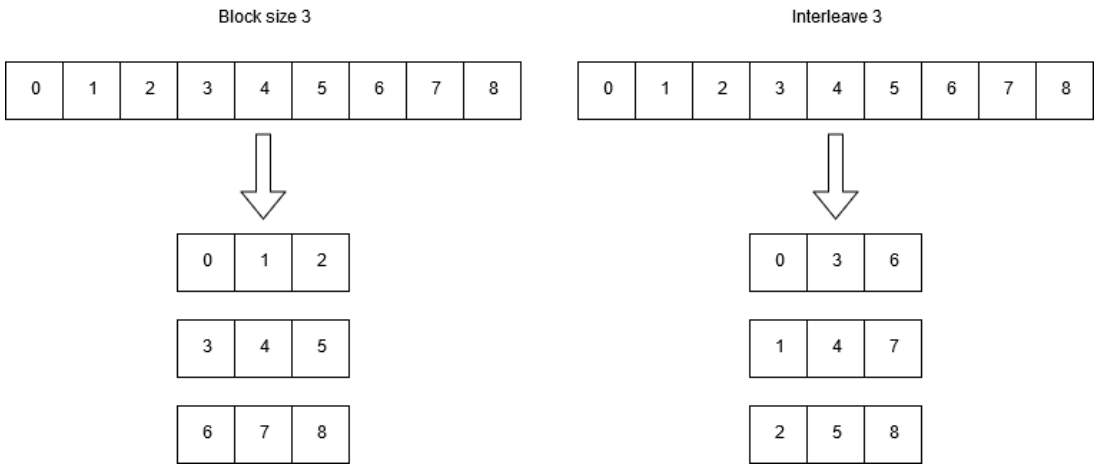

Figure 2.2 *Block size and interleaving.*

possibility to modify easily the designs memory interface. With these tools, parallel memory architectures are possible to implement in reasonable amount of time.

2.1.2 Field Programmable Gate Arrays (FPGAs)

Field Programmable Gate Arrays (FPGAs) are programmable silicon circuits. They are used as a part of various kind of systems and applications because of their flexibility, configurability and rather cheap prize [7]. FPGA based systems have also very short development times. These are the main advantages compared to *Application Specific Integrated Circuits* (ASICs). However, re-programmability brings also disadvantages. FPGAs have usually more delay and their power consumption is higher as well as resource usage.

Modern FPGAs are based on three basic programming technologies. These are *static random access memory* (SRAM), anti-fuse and flash [8]. Programming an anti-fuse-based FPGA is done by applying current and high voltage pulse to the device. The biggest disadvantage with anti-fuse technology is that it cannot be reprogrammed but the FPGAs using that technology have usually lower power consumption and are faster.

SRAM technology, on the other hand, uses static memory cells to store the data [7]. However, the data is lost from them when the FPGA is powered down. Another drawback is the needed size of the SRAM cell. One SRAM cell can require up to six transistors while no transistors are needed to use anti-fuse technology and only one transistor is required for flash technology.

Flash technology is relatively new and tries to overcome the problems of the SRAM technology. It is based on *erasable programmable read-only memories* (EPROM). Its biggest advantages compared to the SRAM are the resource usage and that the data is not lost once powered down.

The most widely used programming technology is the SRAM mainly because of the reprogrammability and that it uses the standard *Complementary Metal Oxide Semiconductor* (CMOS) manufacturing process [7]. Thus, there is no need for special development regarding to the programming as it would be the case with flash technology. The major FPGS producers, such as Intel, use SRAM based technologies in most of their FPGAs.

The FPGA circuits are built from logic blocks. These blocks are used for implementing the logical operations, routing, storage and off-chip connections [7]. The architecture of FPGAs is based on a *lookup table* (LUT) or a multiplexer. Each vendor has their proper names and definitions for their technologies, but the basic principle stays the same.

2.1.3 Arria 10 FPGA platform

The FPGA platform used in this Thesis is from Intel's Arria 10 device family. The FPGA device consists of *Logic Array Blocks* (LABs), which are its basic building blocks, memory blocks and various *digital signal processing* (DSP) blocks [9]. The LAB blocks are formed with several *Adaptive Logic Modules* (ALM) which perform logical functions and calculations. Fundamentally ALMs are based on LUTs. In addition, up to the quarter of the LAB blocks can be used as a memory (MLAB). The external communication with PC is done via *Peripheral Component Interconnect express* (PCIe) [10] bus.

Table 2.1 *Arria 10 FPGA overview.*

Device	Arria 10
ALMs	427200
DSP	1518
LAB	42720
M20K	2713
PCie	Gen2:x8
PLL	144

The specific board used in this Thesis is the Arria 10 GX FPGA Development Kit. Table 2.1 lists the main characteristics of the platform. Arria 10 is Intel's midrange FPGA platform and has a large amount of ALMs LABs and DSPs in use. It supports the 1st, 2nd and 3rd generation PCIe busses. The characteristics make the platform suitable for heavy calculations which are needed to design an accelerator.

The Arria 10 platform in this Thesis is configured to use Gen 2 PCIe bus with 8 lanes for data transfer between FPGA and the *central processing unit* (CPU) of the computer. The Intel PCIe IP on the FPGA runs at 250 MHz and uses a 128-bit Avalon bus as its interface. The theoretical maximum throughput for the PCIe bus with the presented settings is 32 Gbit/s.

The FPGA platforms have two types of *Random Access Memories* (RAM), single port and dual port. This applies also to Arria 10 platform. The memories in the platform support bit width up to 1024 bits for each memory location which is equal to 128 bytes. The memories in Arria 10 are represented as M20K blocks.

Direct Memory Access (DMA) blocks are used to transfer data between CPU and FPGA as independently as possible. The idea is that CPU writes data to an allocated table from where DMA reads it and stores to FPGA memories. Meanwhile CPU can occupy with other tasks as no more information is needed once the data is written to the table. DMA blocks are used also in this Thesis to transfer data between CPU and the FPGA platform.

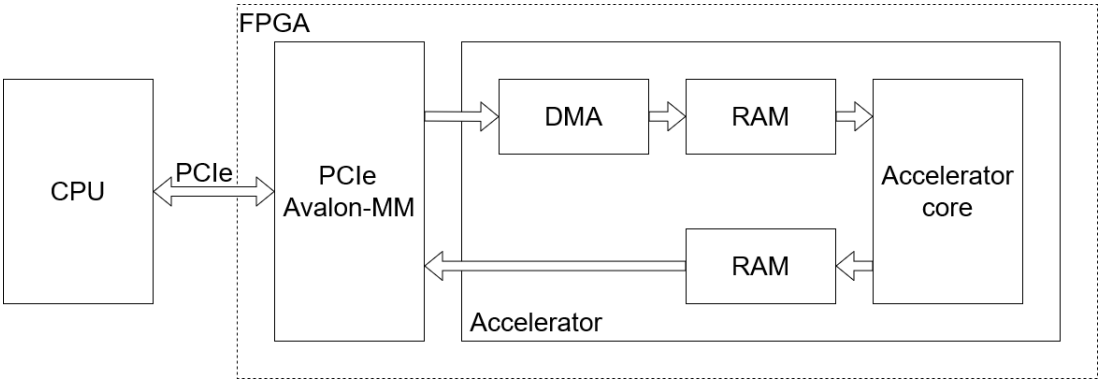

Figure 2.3 *Accelerator system architecture overview.*

The simplified system architecture overview is illustrated in Figure 2.3. The PCie bus utilizes *Avalon Memory-Mapped* (Avalon-MM) interface to access the FPGA memories with or without utilizing DMAs. The Avalon-MM interface is a connection based on addresses and suitable for read/write operations [11]. The interface can be used for master/slave type connection, which is also used in this Thesis. The DMAs and the RAMs form the memory interface between the Accelerator and the CPU. Figure 2.4 illustrates

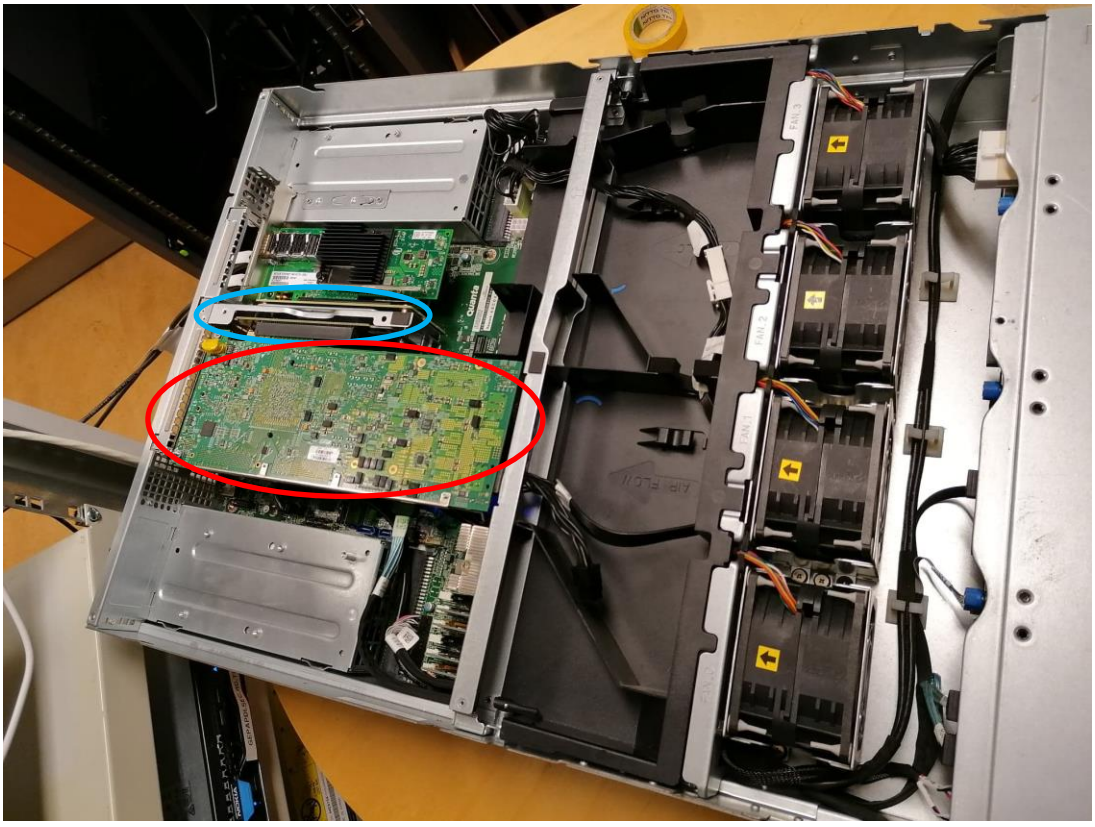

Figure 2.4 *FPGA (red) connected to the CPU via PCIe (blue).*

the physical platform which is used in this Thesis. From there it can be seen how the Arria 10 FPGA platform is connected to CPU via PCIe bus.

2.2 High efficiency video coding (HEVC)

The concept of video coding consists of reducing the amount of data used to represent a digital video signal. Video coding is based on the trade-off of visual degradation and compression. High Efficiency Video Coding (HEVC) is the latest video coding standard [12]. ISO/IEC *Moving Picture Experts Group* (MPEG) and ITU-T *Video Coding Experts Group* (VCEG) forming the *Joint Collaborative Team on Video Coding* (JCT-VC) produce the standard in collaboration. The standard follows the same principles as the previous standard H.264/MPEG-4 *Advanced Video Coding* (AVC), done by the same collaboration.

HEVC standard is developed for the growing need for encoding the higher resolutions [12]. The standard is used in various application ranging from TV broadcasting to streaming and video conferencing. HEVC promises to improve the coding efficiency by 50% compared to AVC. This, however, increases the complexity of the encoders which leads to the need of software or hardware acceleration of the encoders.

In addition, the increased need for efficient encoding for mobile devices and video on demand services are motivations of HEVC standard. Thus, focus of the HEVC standard is on parallel processing and increased video resolutions. JCT-VC also introduces the reference encoder, called *HEVC test Model* (HM), which is used as a base for encoding tools development. The main working principle of HEVC is discussed in the next chapter from the point of view of practical HEVC encoder, Kvazaar.

2.2.1 Kvazaar HEVC Encoder

Kvazaar is an academic open source HEVC encoder developed as a part of research in Ultra Video Group in the Computing Sciences Unit at Tampere University [13]. The encoder is designed from scratch using C and Assembly and is available in GitHub [3]. The main design goals for Kvazaar are to get the coding efficiency close to the HEVC reference encoder and to achieve real time coding speed [13]. Other goals are to optimize the computation and the memory usage and to be easily portable to other platforms. Kvazaar is also used as a part of education at Tampere University.

Kvazaar, as HEVC encoder, is composed of several encoding tools, some of which can be enabled or disabled. Figure 2.5 shows an overview of Kvazaar HEVC encoder. The encoding process starts with input video signal entering to the encoder [12]. Each frame of the video signal is split into smaller blocks. Then the prediction mode is decided. Each block is encoded using either intra or inter prediction. Intra picture predictions aims to

reduce spatial data redundancy. Intra prediction predicts the pixels of a current block according to the neighboring blocks of the same frame. On the other hand, inter picture prediction is a temporal method and uses one or more of the previous frames as a reference for the prediction of the current block. This Thesis focuses only on inter picture prediction, as it is the part of the encoder needed to be accelerated. Inter prediction is discussed in more detail on Chapter 3. The first frame is always encoded using intra prediction because there are not yet available reference frames to perform inter search. The remaining frames are then encoded mostly using inter prediction.

Once the prediction is done, a residual signal is calculated which is the difference between predicted block and the original [12]. The residual goes through a spatial transformation forming prediction coefficients. These coefficients go through scaling and quantization before they are sent to the decoder which performs inverse scaling and transformation. The inverse transform also duplicates the decoded approximation of the residual because it is needed to be the same in the encoder and in the decoder. After that, the residual is summed with the prediction. At this point, some in-loop filters such as deblocking and sample adaptive offset might be used to remove some errors from the previous processing. Finally, the decoded picture is stored into a buffer for further use.

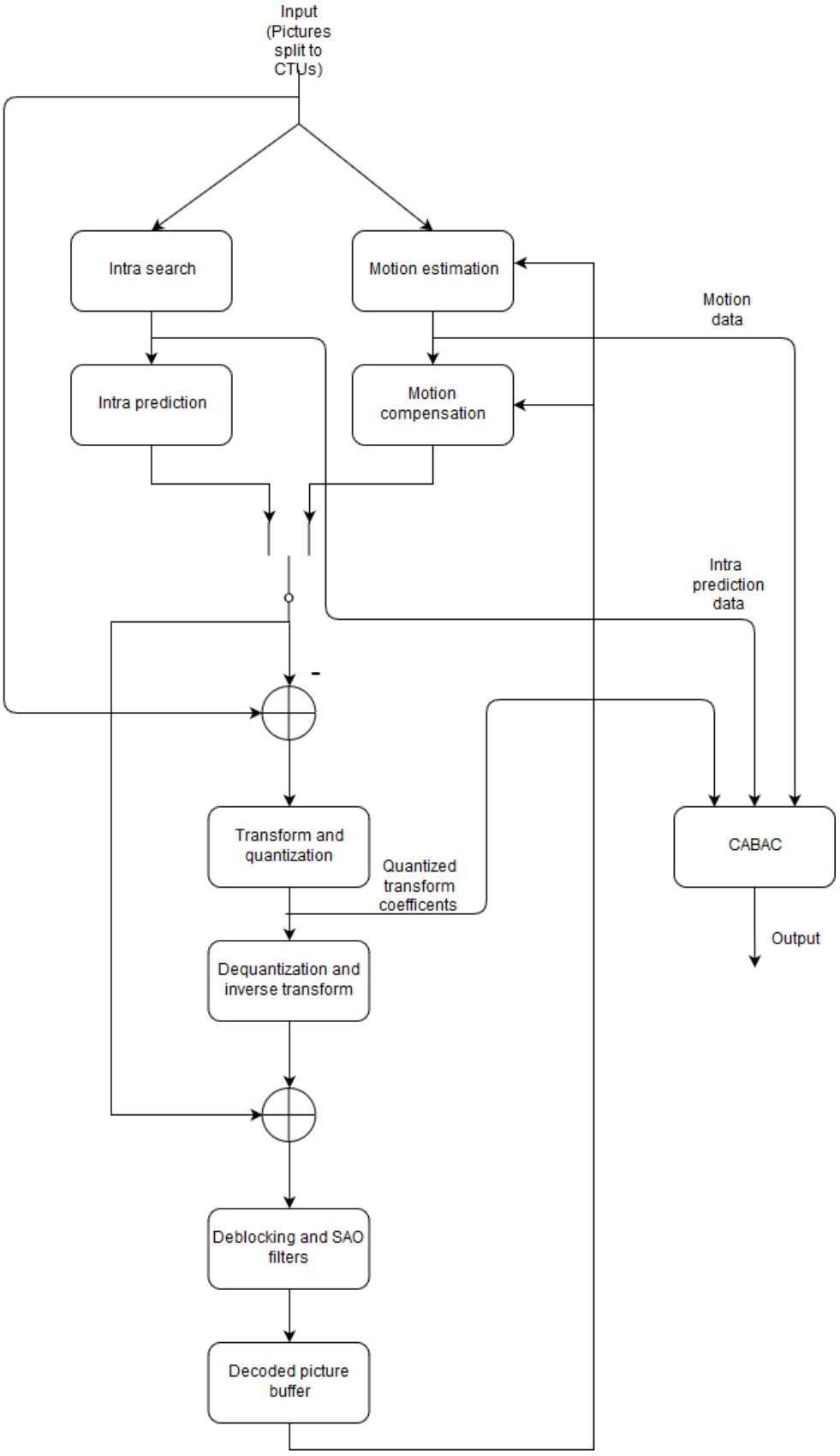

Figure 2.5 *Block diagram of Kvazaar HEVC encoder.*

2.2.2 FPGA Acceleration of Kvazaar Intra Encoder

Ultra Video Group at Tampere University has already developed an accelerator on *System-on-Chip* (SoC)-FPGA platform for the intra picture prediction of Kvazaar using HLS [14]. The accelerator uses Kvazaar's *All Intra* (AI) configuration, meaning that only intra prediction is used. The intra accelerator is designed for live video streaming. The first accelerator uses Intel Cyclone V on Terasic's VEEK development board and the later Arria V and Arria 10 development boards. The design was modeled with untimed and timed SystemC models before implementing on SoC-FPGA platform, also platform modeling and performance estimation was done.

One of the main design approaches is to use HLS instead of traditional RTL design principles to boost the design process and to keep up with Kvazaar's fast development process [14]. Catapult-C implementation was derived from the SystemC model making necessary changes to the coding style.

The intra accelerator on Arria 10 is the starting point for the accelerator interface in this Thesis. The basic communication between FPGA and CPU were already done and only minor modifications are needed to implement an accelerator for ME. Memory architectures need to be changed to respond the MEs needs as well as configuration data.

2.3 Related work

There are not many academic HLS implementations of ME [15]. Most of the other works that were found were either implemented using traditional hardware development methods such as VHDL [16], [17] or implemented on ASIC VLSI [18], not on FPGA. Related work research was limited to the academic publications.

A field report [15] presents a HLS based ME design implemented on FPGA. This field report is a case study that focuses on a predictive block-based ME implemented with HLS tools on Xilinx Virtex-7 FPGA board. Authors use parallel predictive block matching as a motion estimator in their case study. The focus of the report is more on the implementation tools and the comparison between HLS and traditional hardware design methods rather than the performance. The conclusion of the field report is that complex algorithms can be implemented on hardware with HLS in low amount of time compared to the traditional hardware design methods.

Other solutions, without HLS tools, focus on parallelizing the ME block to a very high extent. Authors in [17] propose a highly parallel *Sum of Absolute Differences* (SAD) architecture implemented with VHDL also to Xilinx Virtex-7 FPGA board. Authors developed a fast SAD calculation block to obtain 30 *frames per second* (FPS) real time encoding for 2K videos. The approach was to parallelize the design and the proposed block has

64 processing units to perform the calculations. Authors were able to develop a system that calculates 64 × 64 block size's SAD values in 16 clock cycles at 458 MHz clock frequency.

Authors in [18] have continued the work presented previously and designed ASIC accelerator for *full search* (FS) algorithm. They use 65nm TSMC CMOS technology and were able to achieve 30 FPS encoding for *full high definition* (full HD) (1920 × 1080) sequences using 720 MHz frequency.

Authors in [16] present a variable block size ME using FPGA. Their approach is also highly parallelized, including up to 16 SAD calculation units. Authors synthesized the design on Xilinx Virtex-5 FPGA board, and they were able to reach real time 31 FPS encoding for DVD frames and claim that with more hardware resources full HD frames are also possible.

Only the authors of the field report [15] used HLS tools for their design, but their focus was not on the performance. The rest of related works used traditional hardware design methods. None of them had any application such as video encoder presented with their works, but the architectures worked independently on the FPGAs. The hardware implementation in this Thesis uses HLS tools and is integrated as a part of real-life video encoder.

3. INTER PICTURE PREDICTION IN HEVC

This chapter goes through the principles of HEVC inter picture prediction. That includes block partitioning, motion compensation (MC), motion estimation (ME), calculation of sum of absolute differences (SAD) and block matching algorithms (BMA) including full search (FS) and fast BMAs. Prediction modes in inter picture prediction are also discussed.

The main idea of inter picture prediction in the HEVC standard is to reduce the content redundancy of the subsequent frames [19]. The content redundancy of the successive frames is notable particularly when the frame rate of the input video is high. Thus, HEVC presents an idea of temporal prediction meaning that prediction is based on the consecutive frames.

The prediction of the current frame is done from the previous frame or frames. This is known as forward motion prediction [20]. The other option is to use the following frame or frames as a reference leading to backward motion prediction. If the average of forward and backward prediction is used the resulting frame is known as bi-predicted frame. The reference frames must have been already coded to perform the prediction and they can be either intra or inter coded.

3.1 Block partitioning

In HEVC, the input frame is partitioned into smaller blocks to perform the encoding process. In this standard, the basic calculation block is called *coding tree unit* (CTU) [12]. This block contains $L \times L$ amount of luminance samples and according chroma samples. The HEVC standard allows CTU size L to be 16, 32 or 64. Kvazaar uses only 64×64 sized CTUs so the same size is used in this Thesis also.

CTU is then divided into *coding units* (CU) following recursive quadtree structure [21]. The CTU is divided following a treelike structure into four same sized square blocks on each level until the smallest supported block size, 8×8, is reached. Figure 3.1 illustrates an example of block partitioning of one CTU in HEVC. The coding order of the CUs, called Z-scan is also illustrated in Figure 3.1. The CTU decomposition into CUs depends on CTU's pixels. The same structure is illustrated in Figure 3.2 as a coding tree structure. If the pixels of CTU are homogenous it is not needed to partition until the smallest available block size. This can greatly improve the coding efficiency. However, the smaller CUs are still needed as the bigger ones are not specific enough to spot all the movements in the video signal. Kvazaar supports the CTU partitioning up to 8×8 sized CU blocks.

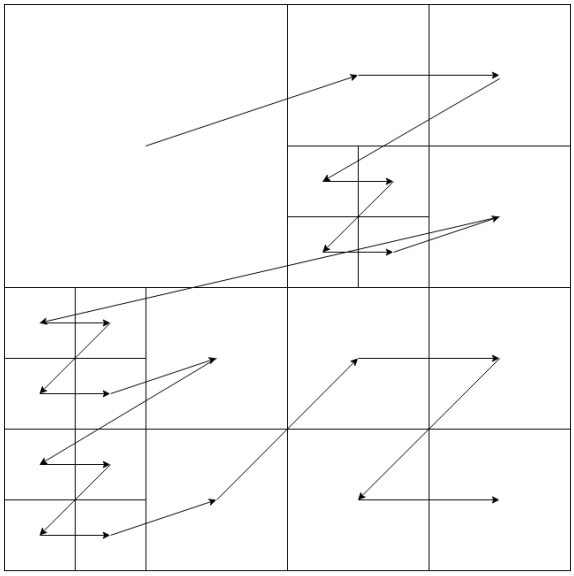

Figure 3.1 *Block partitioning example in HEVC.*

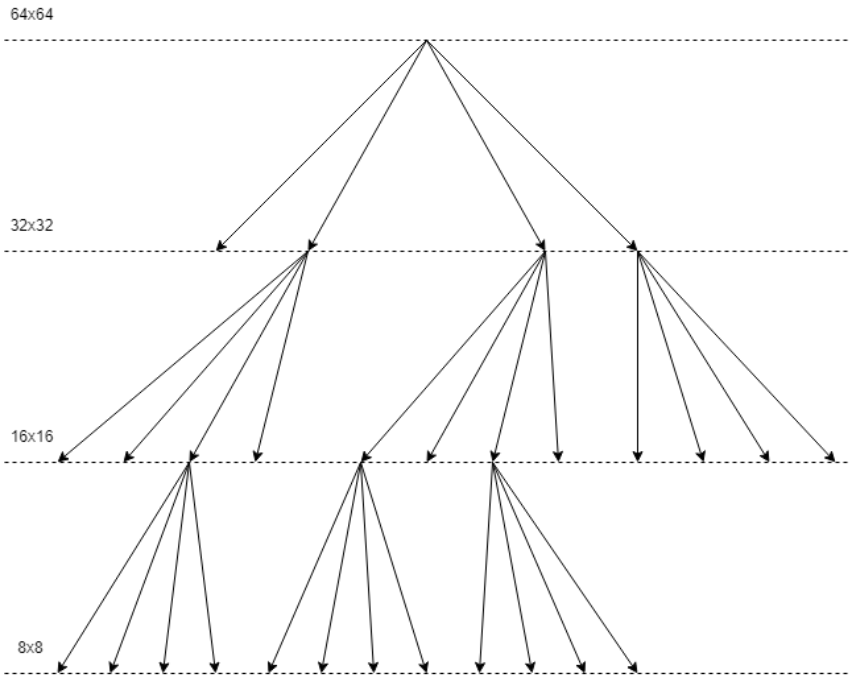

Figure 3.2 *Coding tree structure of HEVC.*

At the CU level is also decided whether the block is encoded using intra or inter prediction [12]. The CUs are divided into *prediction units* (PU). Each CU is split into one, two or four PUs. For inter prediction there are four symmetric and four asymmetric PU splitting types.

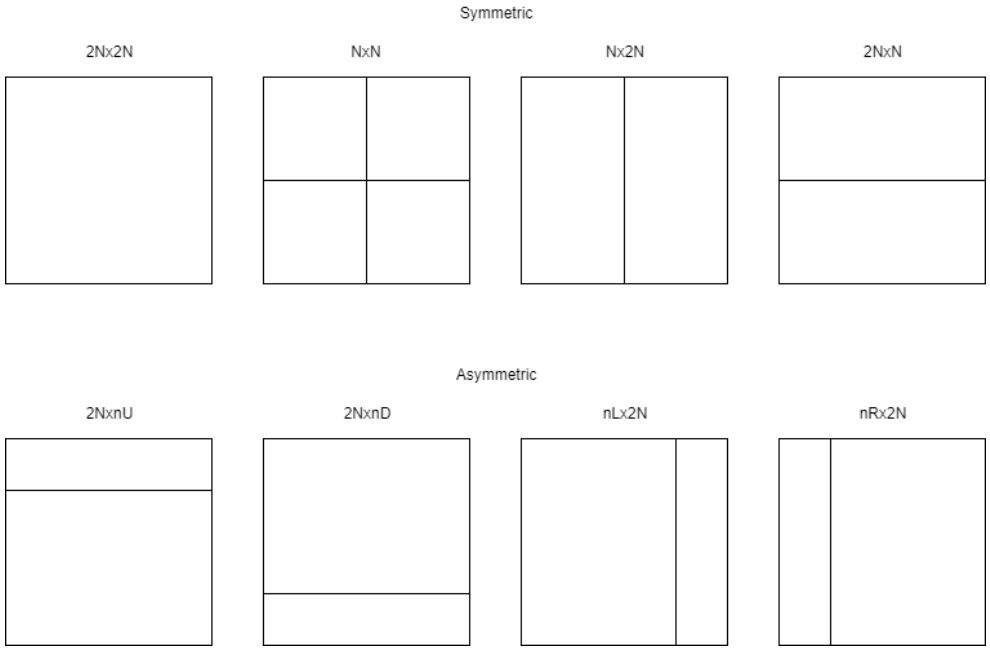

Figure 3.3 *PU splitting in HEVC, modified from [21].*

All the symmetric and asymmetric PU splitting modes are illustrated in Figure 3.3. The symmetric partition modes for PUs are $2N \times 2N$, $N \times N$, $N \times 2N$ and $2N \times N$. Similarly, for the asymmetric modes the partitions are $2N \times nU$, $2N \times nD$, $nL \times 2N$ and $nR \times 2N$. Only the symmetric $2N \times 2N$ mode is used in this Thesis because of its uniformity.

3.2 Motion estimation and compensation

Motion estimation (ME) is used to determine the movement of the objects between the current frame and reference frames and therefore to reduce the residual needed to encode [20]. The result of the ME is called a *motion vector* (MV) which is then used in *motion compensation* (MC) to encode the current frame. MC is the main technique to reduce the coding cost of the inter coded frames in HEVC standard. It is especially effective when there are only small changes in the successive frames in the input video stream. Using MC leads easily to high compression ratio.

To achieve the MVs, ME is needed which is the first thing to perform in inter picture prediction. ME is usually the part of the encoder which requires the most calculations as the most of the frames are usually encoded using inter picture prediction [12]. ME is performed using one of the various BMAs [20]. To perform a BMA, the current frame is divided into non-overlapping PUs which are compared to the reference frame.

The resulting frame from MC is known as motion compensated prediction [20]. MC uses one or various reference frames. When the number of reference frames is increased also the complexity of the encoder increases significantly as well as the need for more memory

to store all the reference data until the prediction is done. MC produces two outputs, one being residual prediction error which is the difference between the motion compensated prediction and the current frame and the second is the already calculated MVs.

ME consists of *integer motion estimation* (IME) and *fractional motion estimation* (FME). IME focuses only on integer pixels and FME divides the pixels still into smaller parts. FME is used to get even more precise results within the pixel. The following BMAs are used for IME. FME uses different methods to adjust the results from IME. Only IME is discussed in detail in this Thesis as it forms its own complete entity. Introducing also the FME would make the Thesis too large. However, FME is an important part of the whole ME process from the quality and performance point of view and is considered in the future work.

3.3 Block matching algorithms (BMA)

As described previously, the current frame is divided into PUs according to the luminance samples. In addition, the reference frame is also divided into blocks called search area. The size of the search area is defined by search range. The search range defines the amount of pixels a PU expands from its corners to form the search area. Kvazaar supports the search range of 8, 16, 32 and 64 pixels. The chosen *block matching algorithm* (BMA) is executed within that search area [20].

The basic idea of the block matching is to find the best PU from the reference frame's search area corresponding to the current PU of the frame to be predicted [20]. When the best PU is found from the reference frame's search area, the displacement between the current PU and the reference PU is determined to achieve the MV. MVs have two components, vertical and horizontal representing the displacement in the frame on Y-axis and X-axis respectively.

Block distortion measures are used to determine the best matching PU from the reference frame. The most commonly used distortion measures are sum of absolute differences (SAD), *mean absolute difference* (MAD) and *mean square error* (MSE) [20]. Kvazaar uses mostly SAD for block distortion measure in inter prediction and therefore it is investigated in more detail.

Calculating SAD means calculating the absolute difference of each pixel's luminance sample within the PU and adding them together [20]. The same process is repeated for each search location within the search area. All the calculated SAD values are then compared, and the smallest SAD determines the best matching block. The following formula is used to calculate SAD for one $N \times N$ sized PU:

$$SAD = \sum_{i=1}^{N} \sum_{j=1}^{N} \left| C(i,j) - R(i + v_x, j + v_y) \right| \quad (3.1)$$

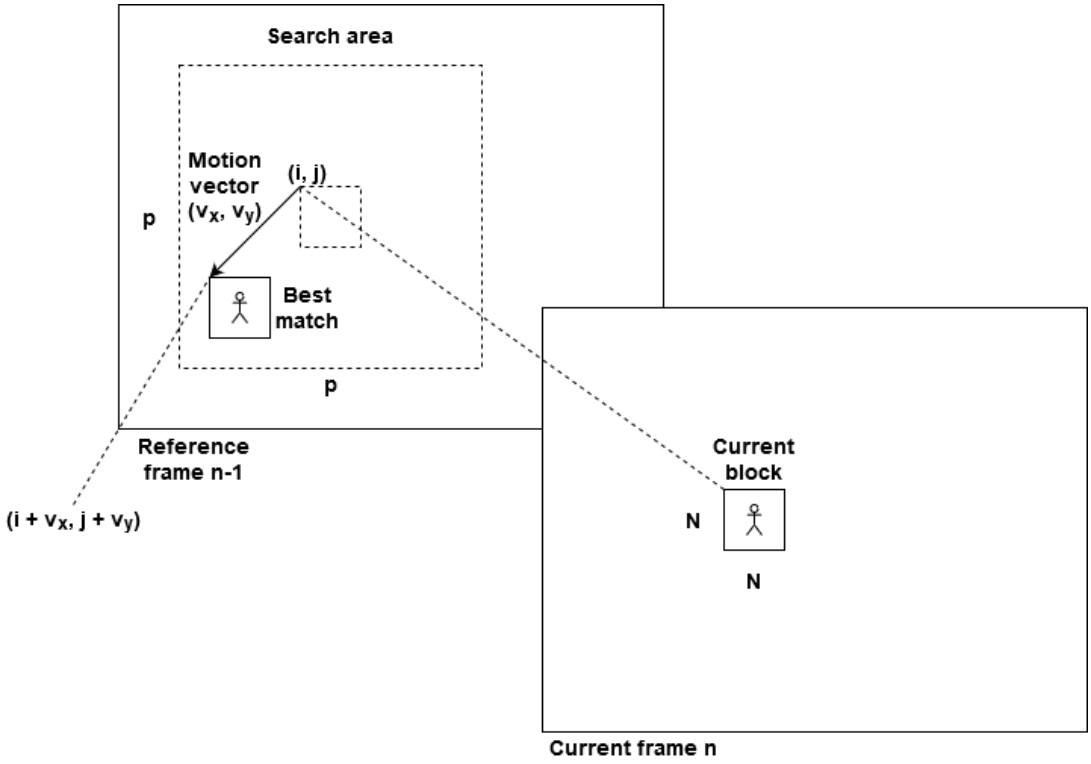

Figure 3.4 *Block matching example.*

where N is the width and the height of the macroblock. $C(i, j)$ is the value of the luminance sample of the pixel at the location (i, j) of the current frame. $R(i + v_x, j + v_y)$ is the luminance value of the reference block at the location $(i + v_x, j + v_y)$ and v_x and v_y are the MV coordinates [20]. Figure 3.4 illustrates an example of block matching. The best matching PU's location is illustrated in the reference frame's search area and the displacement from the original location in the current frame is the MV.

There are various BMAs to find the best matching PU from the reference frame within the search area [20]. The most straightforward is FS, also known as exhaustive search. Other, less computationally heavy, algorithms are for example hexagon-based search and test zone search. They are commonly known as fast BMAs.

3.3.1 Full search (FS)

Full search (FS) algorithm calculates SAD in each possible PU location within the search area and determines where the SAD value is lowest. Calculating SAD and MV in FS for search range of $[-p, p]$ is done as follows:

$$SAD(m,n) = \sum_{i=1}^{N}\sum_{j=1}^{N}|c(i,j) - s(i + m, j + n)|\ ;\ -p \le m,n \le p \tag{3.2}$$

$$MV = \{(u,v)|\ SAD(u,v) \le SAD(m,n); -p \le m,n \le p\} \tag{3.3}$$

Table 3.1 *The amount of search locations with different search ranges.*

Search range	Amount of search locations
8	289
16	1 089
32	4 225
64	16 641

where *SAD (m, n)* is the current PU distortion at the location *(m, n), c (i, j)* is the current PU data at the location *(i, j)* and *s(i + m, j + n)* is the reference PU data at the location *(i + m, j + n)*. MV is the motion vector where the SAD value is the lowest within the search range *[-p, p]* [20].

The complexity of the FS algorithm can be estimated by calculating all the possible unique search locations needed to determine the lowest SAD within the search range. The following equation gives the number of locations tested in a search range of *[-p, p]* [20].

$$LOC = (2p + 1)^2 \tag{3.4}$$

Table 3.1 shows the amount of search locations with different search ranges. As the search locations correlate directly to the amount of calculations needed to perform the FS for one whole full HD video frame, it can be seen how computationally heavy the algorithm is.

3.3.2 Fast block matching algorithms

Several solutions, called fast BMAs, have been proposed to accelerate the FS algorithm [20], [22], [23]. The fast BMAs introduce encoding degradation compared to the FS algorithm as the whole search area is not utilized for the search. This is also the reason why they execute faster than FS. The most popular fast BMAs are *test zone* (TZ) search [23] and *hexagon-based search* (HEXBS) [22]. The fast BMAs allow to adjust the trade-off between computational complexity and encoding degradation. They contain more complex data access patterns compared to very straightforward FS.

TZ search consists of four steps to execute the MV search [23]. First step is to set up the centre of the search which is done using the MV from the previous frame [24]. The previously predicted MV is used as the starting point for the TZ search. Once the starting point is chosen, square or diamond search, illustrated in Figure 3.5 and Figure 3.6, respectively, is performed within the chosen search range. This part is generally called zonal search. The third step of the TZ search is called raster search. This is done only if the result from the current MV search is far from the search centre. Raster search means a small FS around the obtained result from the previous step. The fourth and the last step

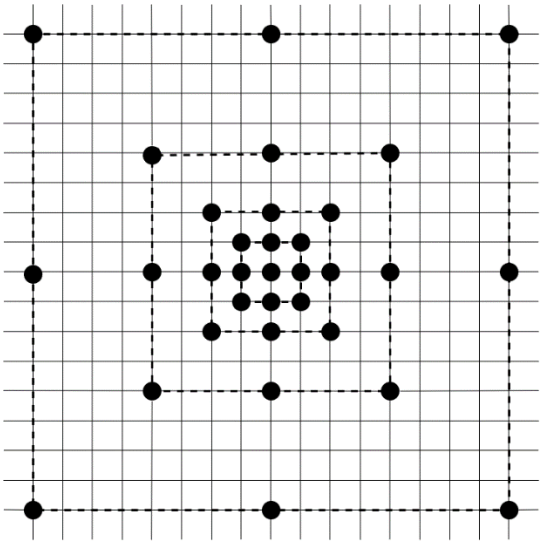

Figure 3.5 *Square search pattern.*

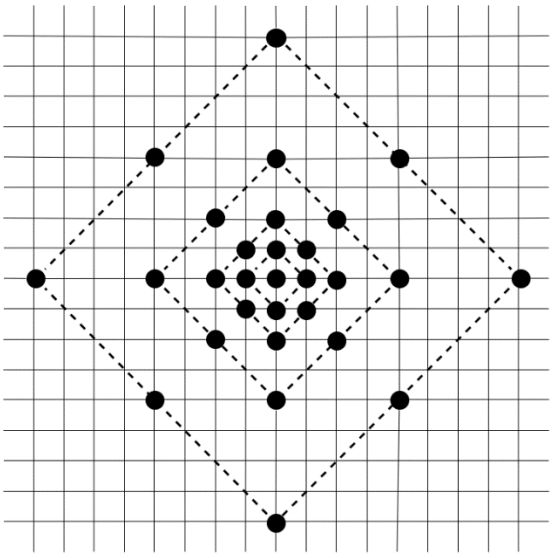

Figure 3.6 *Diamond search pattern.*

of the TZ search is called refinement. On this step, the diamond search pattern is used to adjust the results from the previous steps.

Even though the TZ search can be as much as 60% faster than FS, it still has optimization issues and it does not reach the efficiency of the other fast algorithms [24]. The TZ search can have also bad video compression efficiency as IME has only limited search range, which can lead to the best match being out of scope.

HEXBS has improved search efficiency and performance compared to the TZ search [23]. The first step of HEXBS is the same as on TZ search. Then large hexagon pattern is used to perform the search [22]. Traditional hexagon pattern, illustrated in Figure 3.7, includes

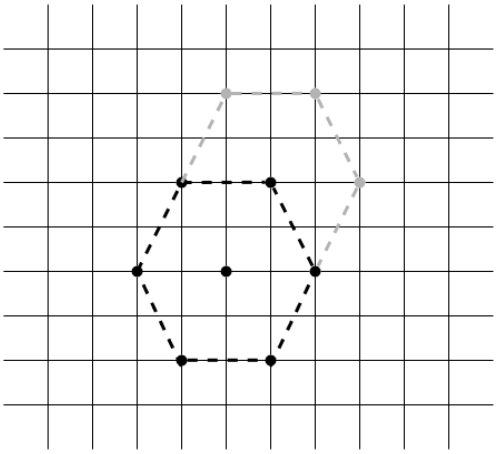

Figure 3.7 *Hexagon-based search.*

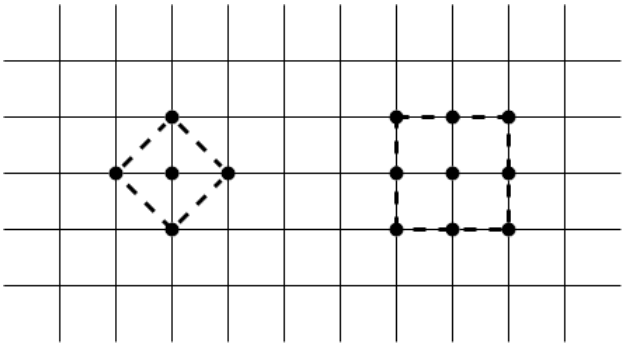

Figure 3.8 *Small pattern of HEXBS.*

six checking points around the centre. As the search continues, the new centre for the search is determined from the previous best match. When the best result is found in the centre of the pattern, small pattern is used. Small pattern, illustrated on the left side of Figure 3.8, has only four search points. However, small pattern excludes the corners so refinement, on the right side of Figure 3.8, is often used instead.

From these fast BMAs, HEXBS is often consider the fastest and the most effective. This is because it uses fewer search points as the others without having differences in the distortion performance [22].

3.4 Inter picture prediction modes

In HEVC standard, the ME is done using one of the three prediction modes [25]. The modes are *advanced motion vector prediction* (AMVP) (sometimes called inter mode), merge mode and skip mode. One of these prediction modes is used every time a MV is predicted.

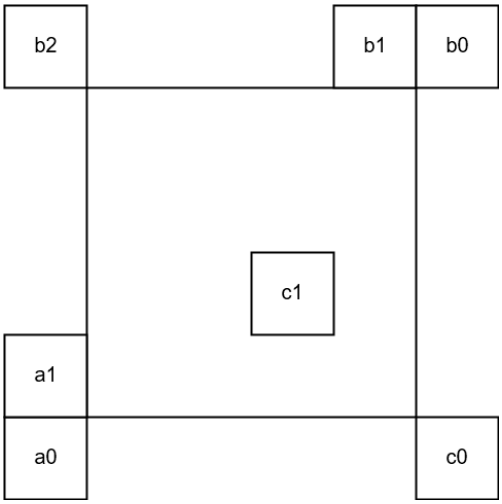

Figure 3.9 *Temporal and spatial MV candidates.*

In the AMVP, a specific candidate list of MVs is checked to obtain the best MV candidate [12]. The candidates are divided to temporal and spatial candidates and are illustrated in Figure 3.9. They illustrate the possible starting locations for defining the best MV. The HEVC standard defines the spatial candidates that are a1, b1, b0, a0 and b2 and they are also checked in that order. The temporal candidates are c0 and c1. The spatial candidates are located on the top or on the left side because those are already processed and available as they are within the current frame. Temporal candidates use the information from the reference frame and therefore the PUs on the right and below are also available. In addition to those, the candidates can include also generated candidates. In Kvazaar, these are called extra candidates.

The candidate list is evaluated in the specified order and up to two best candidate locations are used to obtain the best MV. The MV location can be unavailable if for example the PU in the location is not coded yet or if it is outside the frame.

HEVC standard introduces also a merge mode that can inherit the motion information from temporally or spatially neighbouring blocks [25]. This means that the MV does not need to be calculated but the information is copied directly from the candidate location. The merge candidate positions are the same as in the AMVP. Finally, the skip mode is a special case of the merge mode. If the skip mode is used, all the candidates equal to zero and only the index of the candidate is transmitted.

4. METHODOLOGY

This chapter introduces the design method for designing an accelerator. The complete hardware design flow is discussed as well as tools for defining video coding efficiency. Test materials for testing Kvazaar are also provided.

4.1 Coding efficiency

The most common way to measure video quality degradation is to measure its distortion with *peak signal-to-noise ratio* (PSNR) [26]. It is an objective quality evaluation and compares the maximum power of the signal of the original image to the power of the compressed signal. PSNR is based on *mean square error* (MSE) of the decoded image. The MSE is defined by following equation:

$$MSE = \frac{\sum_{i=0}^{M-1}\sum_{j=0}^{N-1}(I(i,j) - I_d(i,j))^2}{M \cdot N}, \tag{4.1}$$

where N and M are dimensions of the current block which is to be calculated and I is the original image component and I_d is the decoded image component. Then PSNR is calculated from the MSE using the following equation:

$$PSNR = 10 \cdot log\frac{(2^B-1)^2}{MSE}, \tag{4.2}$$

where B represents the bit depth. This gives PSNR value for one of the three color components (luminance and two chrominance) of one block. Weighted average is calculated using all the three components or only PSNR from luminance sample. Weighted average for 4:2:0 sampling is calculated as follows.

$$PSNR_W = \frac{6 \cdot PSNR_Y + PSNR_{C_B} + PSNR_{C_R}}{8}, \tag{4.3}$$

where $PSNR_Y$, $PSNR_{C_B}$ and $PSNR_{C_R}$ are the three different PSNR values from the color components.

In addition to the PSNR, bit rate of the encoder is used to estimate coding efficiency. It means the total amount of bits the encoder produces. To compare and evaluate the performance of encoders and encoding tools, the most widely used metrics is the *Bjøntegaard-delta bit rate* (BD-BR). It makes the comparison of the coding efficiency of two encoders possible [27]. Generally, the BD-BR reports the average bit rate difference percent for two encodings at the same quality level measured with PSNR. When determining the BD-BR, the bit rate is usually represented in a logarithmic scale because on the linear

scale the higher bit rates would be dominating. The curve in the bit rate-PSNR graph is based on four measure points. Those points are the different *Quantization Parameter* (QP) values, 22, 27, 32, 37, defined on the JCT-VCs *common test conditions* (CTCs) [28]. The interpolation is done drawing a third order polynomial on the graph [27]. Then the BD-BR is obtained integrating the both curves and calculating the difference in the area between them.

The QP values control the ratio between compression and visual quality degradation of the encoding process. As the QP represents the quantization levels, on lower QP values there is less distortion and the encoding is slower. Vice versa, using higher QP, quantization levels are less, encoding is faster and there is more distortion on the bit stream.

A common way to measure encoding speed is to determine the frame rate of the encoder. Frame rate is measured in frames per second (FPS). FPS is defined as the inverse of encoding time as described by the following equation

$$FPS = \frac{1}{t}, \tag{4.4}$$

where t is the time needed to process one frame. Variable t is obtained as

$$t = \frac{c}{f}, \tag{4.5}$$

where c is the throughput cycles of an accelerator to process one frame and f is the operating frequency where an accelerator is designed to work. Combining equations (4.4) and (4.5) the FPS calculation is expressed as:

$$FPS = \frac{f}{c}. \tag{4.6}$$

On the other hand, the maximum amount of throughput cycles to achieve certain FPS is further derived from the equation (4.6) as

$$c = \frac{f}{FPS}. \tag{4.7}$$

The throughput cycles for the whole frame are approximated as

$$c \approx c_s \cdot x, \tag{4.8}$$

where c_s is the throughput cycles for one PU and x is the total amount of PUs in one frame. Combining the equations (4.7) and (4.8), the maximum throughput cycles to calculate one frame is obtained the as

$$c_s \approx \frac{f}{x \cdot FPS}. \tag{4.9}$$

4.2 Test materials

As mentioned, JCT-VC has specified CTCs for testing HEVC encoders. Table 4.1 lists the test materials used for testing Kvazaar. The test sequences are separated to different classes, A – E, according to the resolution. Classes F and X defined in the CTCs are omitted from the tests as their sequences do not contain natural motion and the results would be misrepresented.

The tests are automated with a testing tool developed as a part of Ultra Video Group's research, called Venctester. Each sequence is tested with the four earlier mentioned QP values defined in the CTCs: 22, 27, 32, 37. Venctester compares encoders or encoder versions by calculating the BD-BR and encoding speedup. On the tests run for this Thesis, the anchor is the Kvazaar v1.3.0 using HEXBS algorithm for ME. Kvazaar is set to use its ultrafast preset which is the fastest one in terms of encoding speeds.

Table 4.1 *Test sequences.*

Class	Sequence	Resolution	Frame rate (Hz)	Length (s)
A	PeopleOnStreet	2560 × 1600	30	5
	Traffic	2560 × 1600	30	5
B	BasketballDrive	1920 × 1080	50	10
	BQTerrace	1920 × 1080	60	10
	Cactus	1920 × 1080	50	10
	Kimono	1920 × 1080	24	10
	ParkScene	1920 × 1080	24	10
C	BasketballDrill	832 × 480	50	10
	BQMall	832 × 480	60	10
	PartyScene	832 × 480	50	10
	RaceHorces	832 × 480	30	10
D	BasketballPass	416 × 240	50	10
	BlowingBubbles	416 × 240	50	10
	BQSquare	416 × 240	60	10
	RaceHorces	416 × 240	30	10
E	FourPeople	1280 × 720	60	10
	Johnny	1280 × 720	60	10
	KristenAndSara	1280 × 720	60	10

4.3 Design method

The whole accelerator design process is illustrated in the flowchart in Figure 4.1. The design flow starts from inspecting Kvazaar's C source code and writing a modified, untimed version of the algorithm for the hardware platform. This means the variable sizes must be considered as well as the use of the operators such as division and modulus. Next step is to create a test bench for the Catapult-C source code and verify the functionality of the untimed algorithm.

Once the untimed algorithm works correctly, the hardware specific design constraints are specified. In this phase, the used FPGA platform and the operating frequency are chosen. In addition, the RTL synthesis tool is chosen. Next, the architecture specific design constraints are set up. These include how the arrays and look-up tables are mapped to memories and whether memory partitioning or interleaving is used. Arrays could also be mapped to registers, if needed.

In the same phase, the loop unrolling, and pipelining are decided. Once the design constraints are set up, the RTL description of the design is created. Practically, Catapult-C generates Verilog and VHDL files that contain the RTL description of the algorithm. RTL functionality is verified with Modelsim simulation tool. When launched directly from Catapult-C the same test bench is used to verify the functionality of the created RTL hardware.

In this point, architecture constraint exploration is done if the results are not satisfying or there are timing violations. The targeted frequency and pipelining and loop unrolling settings are changed if needed as well as the memory mappings to fulfill the requirements. Then, the RTL description is generated again, and the functionality verified with Modelsim.

When the design works correctly, the generated Verilog (or VHDL) file is moved to Intel Quartus Prime Standard Edition design tool and integrated there to the top-level module of the hardware system. The top-level module is implemented in Verilog and the PCIe interface with possible memory connections is implemented in Intel Qsys Platform Designer. The whole design is compiled, and the FPGA chip is programmed using the Quartus Programmer.

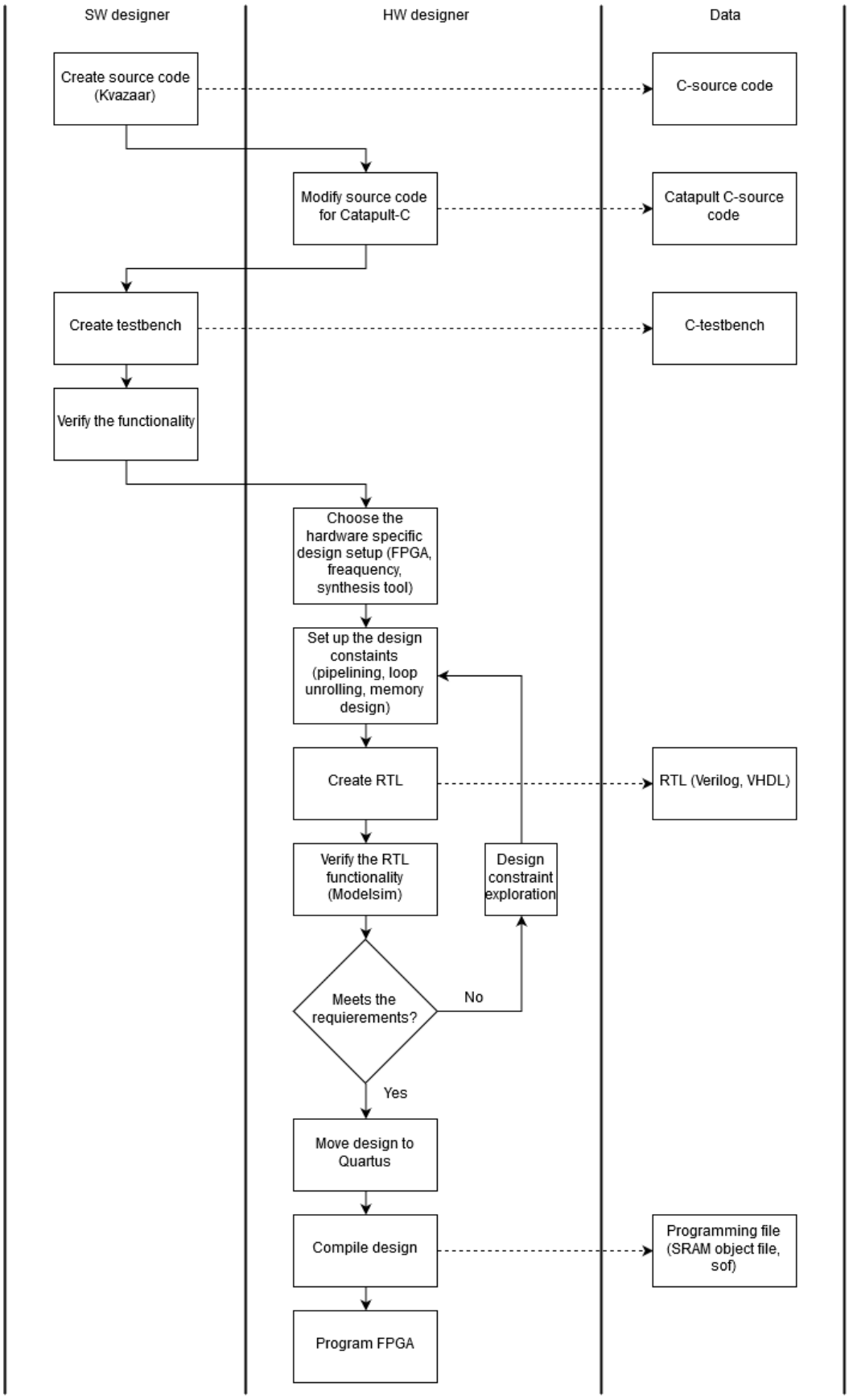

Figure 4.1 *Flowchart of the accelerator design process.*

5. HARDWARE SPECIFICATION

This chapter focuses on the study to choose and limit the ME algorithm for hardware acceleration. The main goal was to find a hardware friendly ME algorithm to accelerate and limit it to be suitable for transferring to the FPGA. Maximum cycle requirements and memory limitations are also discussed.

5.1 Algorithm selection for hardware acceleration

The goal of this Thesis is to accelerate one of the BMAs of ME. The main problem when accelerating the fast BMAs is the complex non-linear memory accesses and the communication between the CPU and the hardware platform. Transferring them to the FPGA platform, would most likely only give very minimal boost to the performance, if any. In addition, the performance boost gained accelerating the fast BMAs might be lost due to the communication as they are already well optimized and fast on CPU.

One advantage for implementing the fast BMAs on software is the use of CPU's threads. They are used to add parallelism to the calculations and that way speed up the encoding. Fast mode decisions are also often used with fast BMAs, which greatly reduces their computational complexity without a major impact on the quality. In conclusion fast BMAs are CPU friendly and they benefit more when executed on software.

On the other hand, the data access in the FS algorithm is straightforward and there are not a lot of dependencies within the algorithm. Also, FS is computationally heavy and executes slower than the fast BMAs on CPU making it ideal for acceleration. FS has also good coding efficiency. In conclusion, FS algorithm is very hardware friendly compared to the fast BMAs. FS algorithm is therefore chosen for hardware acceleration in this Thesis.

5.2 Algorithm limiting

By default, Kvazaar uses all the available PU sizes for ME. To simplify the hardware design process, the algorithm is limited to use only one size. The smallest PU size, 8×8, is used because it results to the best coding efficiency. The bigger the PUs used for ME are, the harder it is to detect motion from the sequence. Other limitation comes from the search range. Kvazaar supports search ranges 8, 16, 32 and 64 and one of them must be selected for the hardware design also.

Table 5.1 *Test results comparing FS and HEBS.*

	normal FS	no spatial candidates	
Range	**BD-BR (%)**	**BD-BR (%)**	**Amount of PUs**
8	-1.04	1.40	289
16	-1.12	-0.96	1089
32	-1.08	-1.09	4225
64	-1.00	-1.01	16641

Extensive tests are run to software only Kvazaar to find out the best trade-off between the coding efficiency and coding complexity. The focus of this experiment is to compare FS algorithm with different search ranges to HEXBS using the BD-BR. HEXBS works as an anchor where the different FS settings are compared to. The HEXBS algorithm is chosen because it is well optimized in Kvazaar and has good coding efficiency. The aim is to find out whether the search range can be reduced from the default search range and is it necessary to use the spatial candidates as described in Chapter 3.4.

The test results comparing normal FS to the HEXBS are shown in the first column of Table 5.1. Results show that the FS algorithm is superior to HEXBS algorithm in terms of coding efficiency no matter which search range is set. Thus, according to the results the search range could be reduced to 8.

However, the first comparison is with complete FS, meaning that the search is done also around the spatial candidates. Searching around them makes the algorithm more complex because they are not part of the regular search area which generates extra data to be sent to the FPGA. The same tests are run to see how omitting the spatial candidates and searching only around the temporal candidate c1 affects to the coding efficiency.

The middle column of Table 5.1 presents the test results without searching the spatial candidates. The average BD-BR using the search range of 16 for FS is slightly better compared to HEXBS algorithm. The amount of PUs, seen in the right column of Table 5.1, is roughly ¼ compared to the default search range 32. According to the results, the search range 16 searching only around the temporal candidate c1 is chosen for the Accelerator. It is a good choice also because of its simplicity for the memory architecture as the needed memory access is straightforward and just one search area needs to be transmitted to the FPGA for each PU. Implementing complicated memory structures to the hardware is both inefficient and time consuming.

5.3 Design limitations

To get a general idea how fast the designed Accelerator should be, maximum throughput cycles to achieve 30 FPS encoding is calculated. The theoretical maximum throughput

Table 5.2 *Maximum throughput cycles to achieve 30 FPS.*

	125 MHz	150 MHz	175 MHz	200 MHz
full HD	128	154	180	205
4k	32	38	45	51

cycles with different frequencies for full HD and 4k resolutions are presented in Table 5.2.

The maximum amount of throughput cycles to process one PU is calculated from (4.9) knowing the targeted frequency, wanted FPS and the total amount of PUs in one frame. Each full HD frame consists of (1 920 / 8) × (1 080 / 8) = 32 400 PUs and 4k frame (3 840 / 8) × (2 160 / 8) = 129 600 PUs. The results are used as a guideline to achieve the targeted 30 FPS encoding. The numbers do not include the communication delay between CPU and FPGA and are purely limitations for the Accelerator.

Kvazaar uses also an encoding tool called early skip. The purpose is to reduce bit stream and make the encoding faster skipping PUs if the lowest SAD would be almost the same as before. In practice, this means that some of the PUs, depending on the sequence, are not processed. This is not considered on the calculations in Table 5.2 and the required cycles represent the situation when every PU of the frame is encoded, leading to the worst-case scenario.

5.4 Memory limitations

The type of memory architecture is important to consider when designing a computationally heavy algorithm. These algorithms require a lot of data to be read from on-chip RAMs. When using one RAM block, only two memory reads, or writes can be done in parallel. This is a huge restriction when trying to add more parallelism to the architecture.

Each pixel is composed of 8 bits. In this case, it would not be efficient to save each pixel to a separate memory location. The memories in Arria 10 support different bit widths. Utilizing this, one memory location could hold more than one pixel. This already reduces the required reads significantly as various pixels are read in parallel from one memory location.

One PU has 1 089 unique search locations within the search area when using the search range of 16. Considering 8 × 8 PUs and supposing each pixel and reference pixel is stored in one separate memory location, for each search locations 64 memory reads are needed. This leads to a total of 1 089 × 64 + 64 = 69 760 memory reads to calculate the SAD values for the whole search area. The extra 64 reads are to get the current PU, which needs to be read just once as it stays the same for each search location.

As one full HD frame consists of 32 400 PUs, accessing the needed pixels to process one whole frame uses $32\,400 \times 69\,760 = 2.26 \times 10^9$ reads from the on-chip memories. This example aims to demonstrate that to perform FS for one full HD frame in reasonable amount of time an optimized memory structure is needed.

6. ACCELERATOR DESIGN

This chapter presents the complete practical design flow of using HLS to create an FPGA based ME Accelerator for Kvazaar. The design process follows the flow presented in Chapter 4 and the hardware specifications from the previous chapter are also considered.

6.1 Architecture overview

Figure 6.1 illustrates the block diagram of the Accelerator design. It is divided into Accelerator core, for calculations, and the Accelerator which works as an interface for connecting Kvazaar to the CPU. The Accelerator is connected to the CPU via PCIe bus, which is used for sending the current PUs luminance pixels and the according search area's luminance pixels to the FPGA. Once calculations are done, the results can be read from the on-chip memories also via PCIe bus. A Linux driver is created to handle the communication between Kvazaar and PCIe device. The Accelerator and the Accelerator core designs are generic and can be multiplied on the FPGA platform. All the designed blocks are discussed in greater detail as well as the required memory architecture in the following sections.

The current PU and the according search area are transferred to the FPGA memory using a DMA block, a *first-in-first-out* (FIFO) component and a *memory indexer* block. The *memory indexer* saves the calculation data to the memories from where it is read to be processed. The DMA block is capable of processing data independently from the processor, so when the needed data is sent to the DMA buffer, the processor can occupy with other tasks meanwhile. The DMA writes the data to the FIFO memory which forwards it to the *memory indexer*. The reason to use a FIFO between the DMA and the *memory indexer* is to have a temporal place for the data as the DMA block processes the whole data at once when started.

The Accelerator core is separated into three different blocks named *write*, *read* and *calculation*. The *write* block gets the calculation data from the on-chip memories and writes it to the *calculation* block. The *calculation* block determines the SAD values and the according MVs from the data. Finally, the *read* block gets the calculated results and stores them to the on-chip memory.

Keeping in mind the cycle limitations and the memory access limitations, the calculations must be pipelined and parallelized. Easiest way to reduce the required cycles is to reduce the amount of data sent to the *calculation* block and then reuse it. This already reduces

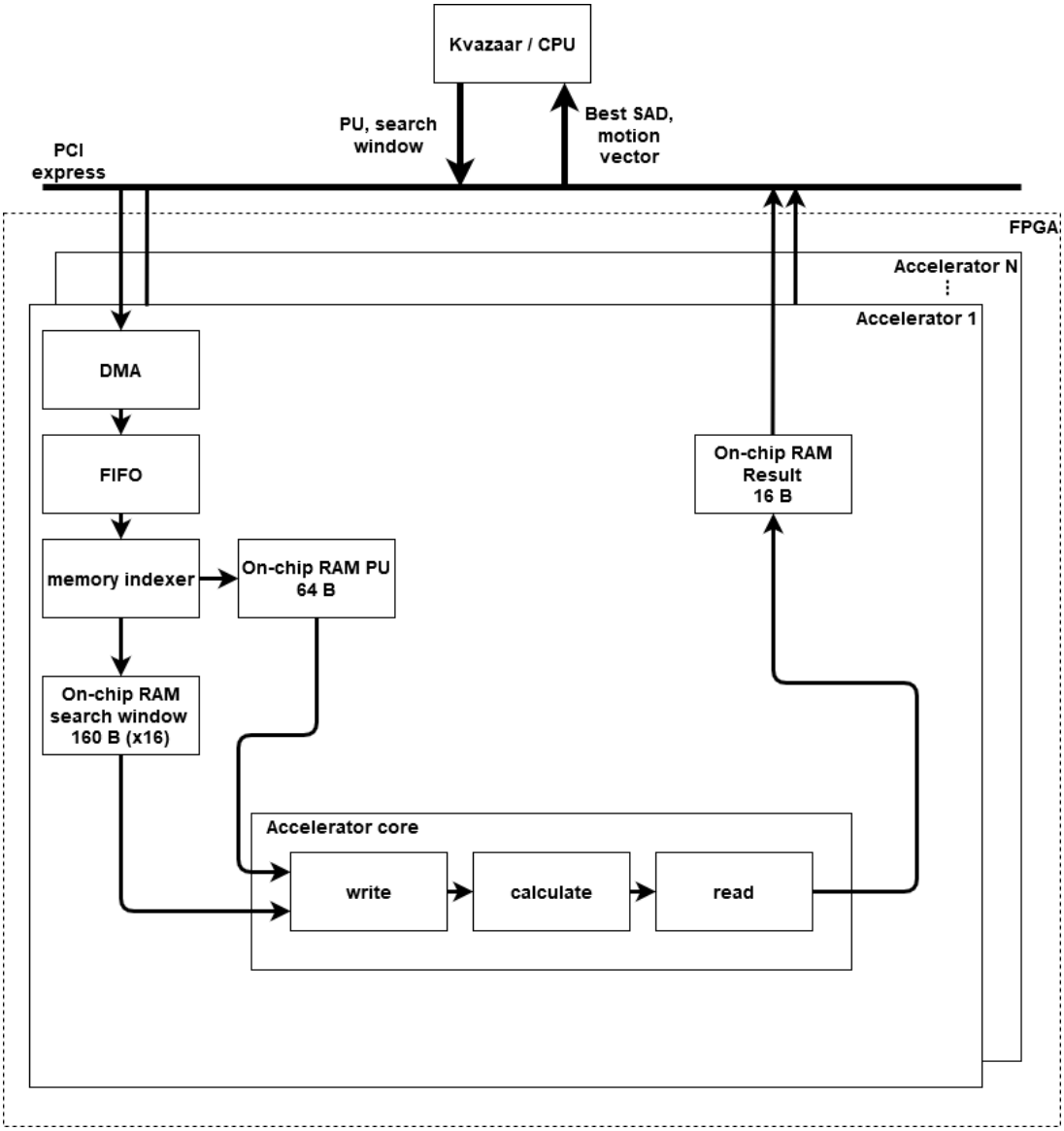

Figure 6.1 *Accelerator block diagram.*

the memory access greatly. To make the reuse easier, the search area is divided, and the calculation is done in smaller parts. In addition, the *calculation* block can be reused.

6.2 Read and write blocks

Read and *write* blocks are needed for getting the data for calculation and storing the results to the on-chip memories. The *write* block is responsible of writing the correct pixels to the *calculation* block and *read* block reads the results from there. Figure 6.2 illustrates search area division into smaller search windows. It shows the computed part of one *calculation* block call. Figure 6.2 shows the first five search windows. The search area is divided into a total of 16 search windows, reflecting also to the amount of times the *calculation* block is called.

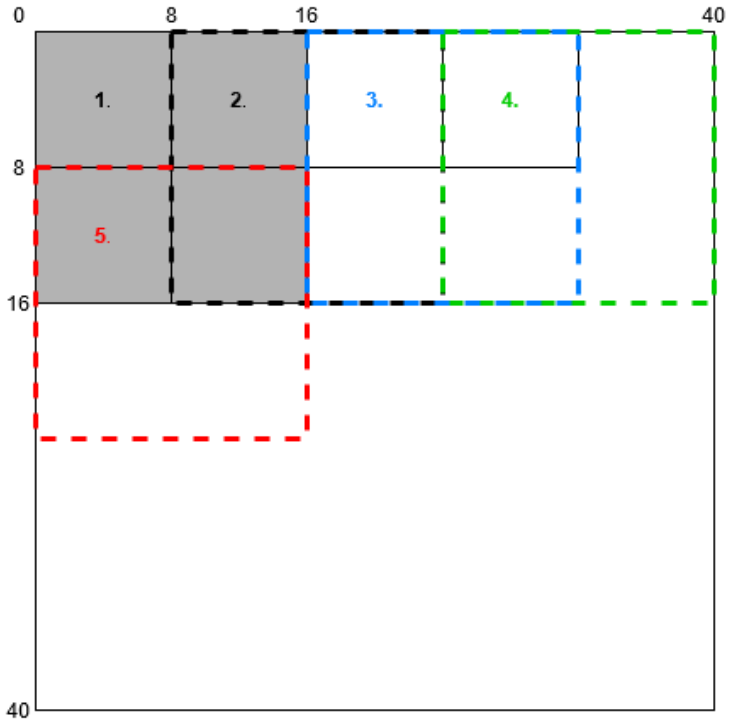

Figure 6.2 *Search area partition into search windows.*

The grey area shows the first part of the calculation, 16 × 16 = 256 pixels, that are sent to *calculation* block. The right half of the grey marked pixels, 128 pixels in total, are reused for the second time the *calculation* block is called, illustrated as black dashed line. This means that only the right half of the black dashed line pixels must be read from the memory. The same happens again for the third and fourth search windows, marked with blue and green dashed lines respectively. When the search window row changes, illustrated with red dashed line, within the search area, the reuse is not used. Instead, all the pixels are read from the memory and registers and the reuse cycle is started over.

One way to gain access to the new 128 pixels from the search area is to access 64 bits (8 pixels) from 16 different memories. Though, the current 40 × 40 search area cannot be divided easily into 16 equal sized memories. The problem is overcome by duplicating some parts of the search area. That way the Catapult-C's design constraints can be used effectively, and pixels are read always from the same index from each of the memories. The duplication is shown in Figure 6.3(a). When starting the new search window row, illustrated as red dashed line, the upper part of the pixels is duplicated. This leads to total of 3 × 8 × 40 = 960 duplicated pixels. With the duplication the total amount of search area memory is increased to 1 600 B + 960 B = 2 560 B. The numbers 1 – 4 in Figure 6.3(a) illustrate the separated search window rows.

Assuming the pixels are duplicated, the memory interface of the *write* block is reorganized using the Catapult-C's design constraints. The memory width in Catapult-C is set

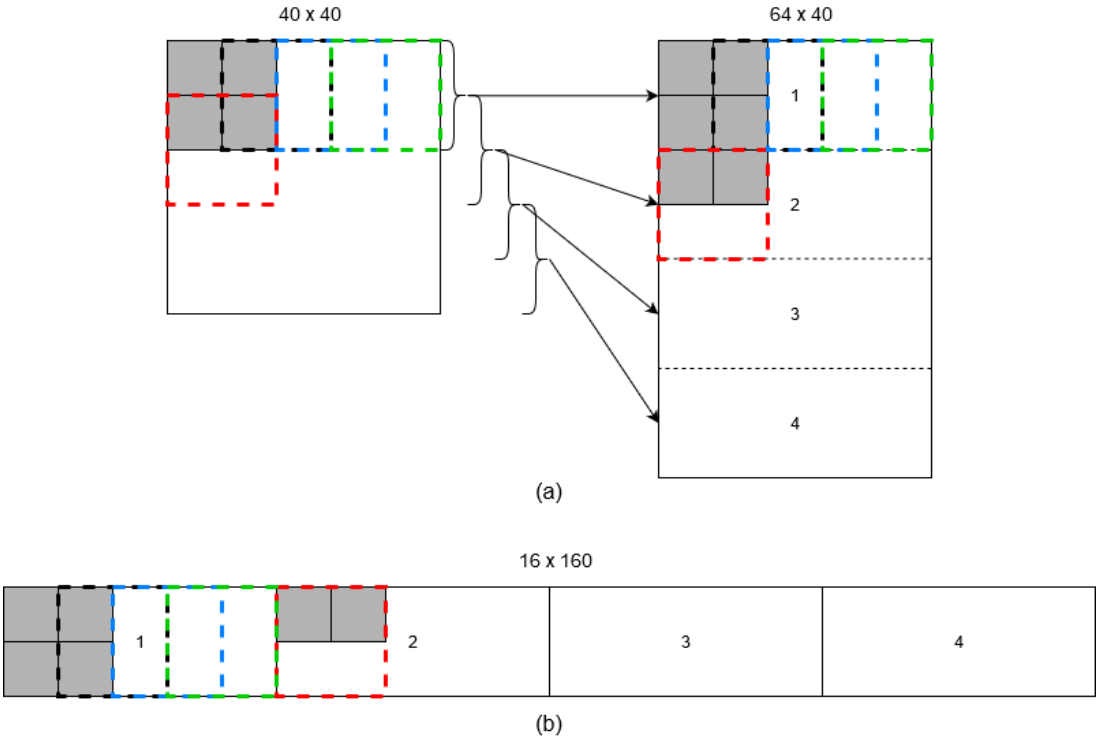

Figure 6.3 *Memory organization.*

to 64 bits meaning that one memory location contains eight pixels. The block size is set to 20 which leads to the wanted 16 separate memories, illustrated in Figure 6.3(b). Eight new pixels are read from each one of the 16 memories granting access to the wanted 128 new pixels each clock cycle. As the duplicated search area is stored evenly on the on-chip memories, each of them contains 2 560B / 16 = 160 B. The *memory indexer* block takes care of the duplication and is discussed into greater detail later.

The *write* block sets up constant registers reading the first part (4 × 16 × 8 = 512 pixels) of each row and the PU pixels. Then, the *write* block reads 8 × 16 = 128 pixels each clock cycle from the search area memories and stores them into registers. Each time the *calculation* block is called, the corresponding search window data is created combining the newly read data and the data stored into the registers. That way the same pixels do not need to be read twice from the on-chip memories.

The main data write loop is illustrated as a simplified Catapult-C code on the Listing 6.1. The writing loop is pipelined with II = 1 so that the pixels are written to the data channel of *calculation* block every clock cycle. The *allData* variable contains the PU pixels and the search window pixels. The *swSliceFirst* contains each search window rows first pixels. The two inner loops are designed to recycle the pixels and the minimum amount of them is read from *searchArea* memory each cycle. The pixels are moved to and from the registers *swSliceBig* and *swSliceRowBack* using *slc* and *set_slc* methods respectively,

```
    #pragma hls_pipeline_init_interval 1
2   for(memLoc = 0; memLoc < 160; memLoc+=40){
      for(widthLoc = 0; widthLoc < 32; widthLoc+=8){
4       #pragma hls_unroll yes
        for(row = 0; row < 16; row++){
6         if(widthLoc != 0){
            swSliceBig.set_slc(row*128,
8           swSliceBig.slc<64>((row*128)+64));
          }
10        else{
            swSliceBig.set_slc(row*128,
12          swSliceFirst.slc<64>((((heightLoc*2)*64)+(row*64))));
          }
14        #pragma hls_unroll yes
          for(pix = 0; pix < 8; pix++){
16          swSliceRowBack.set_slc(pix*8,
            searchArea[row*160+(memLoc+widthLoc+pix+8)]);
18        }
          swSliceBig.set_slc((row*128)+64 ,swSliceRowBack);
20      }
        allData.set_slc(512, swSliceBig);
22      write(allData);
      }
24    heightLoc+=8;
    }
```

Listing 6.1 *Data write loop as a simplified Catapult-C code.*

provided by Algorithmic-C. The inner loops are fully unrolled. The main loop in the *read* block is similar, but it does not contain the two most inner loops and instead of writing the pixels the results are read from the channel.

Once the *calculation* block is ready, the *read* block gets the results from it and stores them into registers. When the whole search area is calculated, the *read* block iterates each value of the register where the results are stored, chooses the best SAD and MV pair from there, and stores it to the result memory. Then the results data is transferred back to CPU via PCIe bus.

6.3 Calculation block

The *calculation* block is composed of the SAD value and corresponding motion vector computations. The idea was to parallelize as much as possible to achieve lowest throughput possible even though it means using more resources on the FPGA. The *calculation* block is connected to the *write* and *read* blocks using the Catapult-C's channels. The current PU pixels and its 16 × 16 sized search window is transmitted through the input channel.

The *calculation* block is entirely register based and therefore notably the biggest part of the Accelerator. Listing 6.2 shows the main calculation loop as a simplified Catapult-C code, where the SAD values are calculated and how the new calculation data is obtained.

```
    #pragma hls_unroll yes
2   for(row = 0; row < 8+1; row++){
      #pragma hls_unroll yes
4     for(pix = 0; pix < 16; pix++){
        // Locations 0-8 on a row, calculate SAD
6       if(pix < 8+1){
          cost = calcSAD(pu, refpu);
8         costTable[costLoc] = cost;
          mvXtable[costLoc] = pix;
10        mvYtable[costLoc] = row;
          costLoc++;
12        cost = 0;
        }
14      // Move the pixels in refpu and get 8 new pixels from searchWindow
        #pragma hls_unroll yes
16      for(loc = 0; loc < 8; loc++){
          refpu[loc*8+0] = refpu[loc*8+1];
18        refpu[loc*8+1] = refpu[loc*8+2];
          refpu[loc*8+2] = refpu[loc*8+3];
20        refpu[loc*8+3] = refpu[loc*8+4];
          refpu[loc*8+4] = refpu[loc*8+5];
22        refpu[loc*8+5] = refpu[loc*8+6];
          refpu[loc*8+6] = refpu[loc*8+7];
24        refpu[loc*8+7] = searchWindow[row+loc][8+pix];
        }
26    }
    }
```

Listing 6.2 *The main calculation loop as simplified Catapult-C code.*

Before the SAD calculations, the PU and search window pixels are read from the channel and stored to registers. The first reference PU is also separated to its own register from the search window outside the main calculation loop.

The SAD value for one search location is calculated recursively. The calculation is separated on its own function called *calcSAD* on Listing 6.2. As shown in row 7, the *calcSAD* function is called which takes two 64 B array registers, *pu* and *refpu*, as an argument. Those registers contain the current PUs pixels and the corresponding reference pixels from the search window, respectively. The function iterates through the registers recursively, subtracts the pixel values and determines its absolute value. Then, each absolute value is added together resulting to a SAD value for one search location. All the loops are fully unrolled resulting to the maximum parallelization. Also, the whole *calculation* block is pipelined with II=1 leading to effective data flow.

Once the SAD value is determined for one register pair, it is stored to *costTable* register and the search window location is stored into *mvXtable* and *mvYtable* registers representing the x and y components of the MV. Then, the pixels inside *refpu* register are shifted according to Figure 6.4. The vertical pixels on the left side are discarded and the rest are moved vertically to start from the beginning without changing the order. Finally, eight

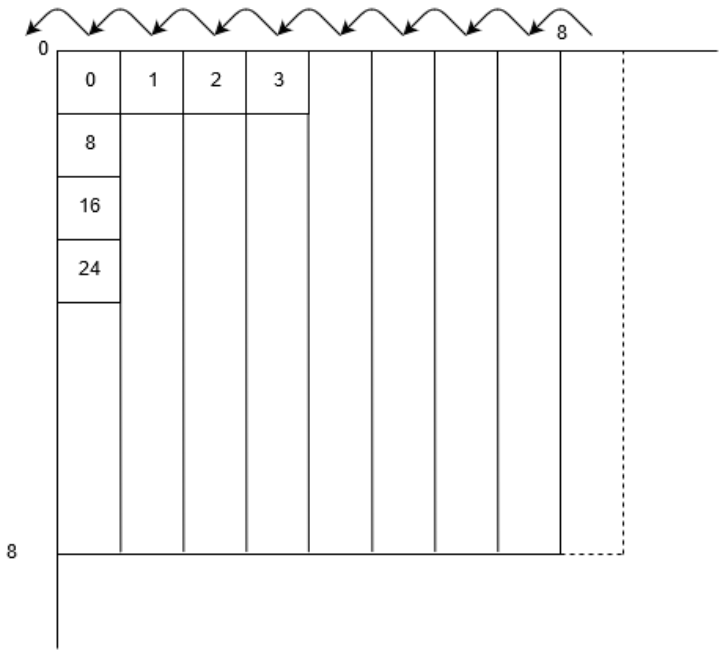

Figure 6.4 *Pixel shifting in refpu register.*

new pixels, illustrated with dashed line in Figure 6.4, are obtained from the *searchWindow* register. The pixels are stored to the last vertical position of the *refpu* register and the calculations for the next location start.

The SAD and MVs are calculated for all the unique locations on the first row of the search window. To get the right pixels to the *refpu* when changing the row, the shifting loop is executed eight times. In practice, this means reading eight new pixels from the *searchWindow* register and shifting them each time the shifting loop is executed. Finally, after executing the loop eight times the *refpu* register has the correct pixels to start the calculation again. The same process is repeated for the whole search window and the SAD and MV pairs are stored to the according registers. Finally, the smallest SAD is determined using a recursive function and it is written to an output channel with the corresponding MV.

6.4 Memory indexer

The *memory indexer* performs the pixel organization to the on-chip memories. The pixels organization requirement is led by the *write* block which needs to access them in a specific way to be pipelined and parallelized effectively. It will not be beneficial to separate the pixels to 16 separate memories directly from Kvazaar. That is why the whole search area and the PU pixels are sent at the same time through the DMA and FIFO to the *memory indexer* which takes care of storing the pixels to the correct memories in correct order.

The total amount of search area pixels needed to calculate the whole PU is $(16 + 8 + 16)^2 = 1\ 600$. In addition, the 64 PU pixels must be transferred leading to total of 1 664 pixels. Those pixels are obtained from Kvazaar and sent to the DMA block which writes them to the FIFO. The *memory indexer* is connected to the FIFO with Catapult-C channel. It obtains 64 pixels at the time from the FIFO and stores them into the on-chip memories. This results to 1 664 / 64 = 26 reads from the FIFO. The *memory indexer* also duplicates the necessary pixels while storing them.

Figure 6.5 illustrates how the search window pixels are divided and duplicated into 16 on-chip memories. Each smaller memory SW0 – SW15 has 20 memory locations, each holding 8 pixels. This solution leads to store 160 pixels to each smaller memory. The colors in Figure 6.5 show which part of the pixels are the same in the memories. For example, the memory SW0 has the same pixels in the location 5 – 9 as the memory SW8 in the locations 0 – 4. The main disadvantage of this approach is that the search window memories have quite a lot of overlapping pixels.

Figure 6.5 *Memory structure of the Accelerator.*

7. PERFORMANCE

This chapter focuses on the performance analysis of the designed Accelerator. The performance of the Accelerator is analyzed from different points of view. HLS results are presented, and theoretical maximum FPS is calculated. Another aspect is to observe the used resources in the Arria 10 FPGA platform. Moreover, a relative speedup is calculated compared to the pure software implementations. The results are finally compared to the existing hardware ME implementations.

7.1 Implementation results

The implementation results are presented in the same order as the design flow goes. First the HLS results from Catapult-C, then the synthesis results from Quartus and finally the results of the whole Accelerator ran with Kvazaar. Version v1.3.0 of Kvazaar is run using the fastest preset, known as ultrafast. In addition, Kvazaar is limited to use only 8 × 8 PUs and the ME algorithm is chosen to be FS with the search range of 16. HEXBS algorithm is also run on software for comparison purposes.

Table 7.1 *Test PC setup.*

Test PC	
CPU	Intel E5-2680 v3, 2.50 GHz
Memory	32 GB DIMM, 2.1 GHz
Storage	3 TB SATA HDD
OS	Ubuntu 18.4.1
FPGA	Arria 10
Encoder	Kvazaar v1.3.0

The test PC setup is listed on Table 7.1. The Accelerator is synthesized on Arria 10 FPGA and Kvazaar is running on Intel E5-2680 v3 Xeon CPU at 2.50 GHz. In addition to the CPU, the Test PC consists of 32 GB DIMM RAM running at 2.1 GHz and 3 TB hard disk drive. The operating system is Ubuntu 18.4.1.

7.1.1 Catapult-C

After going through the HLS flow, Catapult-C outputs the latency and the throughput of the design in cycles. Catapult-C also outputs estimations for the required logic usage on the FPGA. However, it is not presented in this chapter because it usually does not correspond to the real logic usage after the synthesis.

Table 7.2 *Catapult-C results.*

Design unit	Latency	Throughput
Accelerator core	54	46
Write and read	48	30
Calculation	6	16
Memory indexer	26	28
Accelerator total	**80**	**74**

Table 7.2 shows the latency and the throughput cycles for the designed blocks for processing one PU. They do not include data transfer from CPU to the FPGA. The results are separated to the *read*, *write*, *calculation* and *memory indexer* blocks and the total amount of the cycles of the whole Accelerator is calculated. Results of Table 7.2 show that the Accelerator core, including *write*, *read*, and *calculation*, has the total throughput of 46 cycles and the latency of 54 cycles. These results represent the total amount of cycles required for calculating the SAD and MVs because the *memory indexer* is needed only for data organization before the calculations and is not directly part of the FS algorithm. The actual throughput of the *calculation* block is 1, but it is executed 16 times as 16 iterations are needed to calculate the whole search area for one PU. This results to the throughput of 16. Storing the pixels to the FPGA memory with *memory indexer* takes 28 cycles. Therefore, the total amount of throughput cycles and latency cycles of the whole Accelerator is 74 and 80 respectively.

From the presented results, the theoretical maximum FPS of the proposed Accelerator core is calculated. The Accelerator is tested with various full HD sequences provided by Ultra Video Group [29]. First, the average amount PUs in a frame is calculated by running Kvazaar and calculating how many PUs are processed in one encoding run. Then it is

Table 7.3 *Theoretical maximum FPS and speedup.*

Sequence	Amount of PUs per frame	FS speed on SW (FPS)	Accelerator core (FPS)	Speedup
Beauty	25 530	1.93	127.73	66.23
Bosphorus	25 104	1.96	129.90	66.14
HoneyBee	29 951	1.64	108.87	66.36
Jockey	24 944	1.97	130.73	66.22
ReadySetGo	26 308	1.87	123.95	66.27
ShakeNDry	31 117	1.58	104.79	66.25
YachtRide	25 308	1.94	128.85	66.36
Average	**26 894**	**1.84**	**122.12**	**66.26**

divided with the number of processed frames. Then, from the required cycles to process one PU and the used FPGA frequency, 150 MHz, the maximum theoretical FPS for the Accelerator core is calculated. Also, the software only FS algorithm FPS is calculated determining the time used in the algorithm only, excluding the rest of the encoding process.

The theoretical results are listed in Table 7.3. The Accelerator core is on average theoretically ×66 times faster than the software only FS algorithm. The amount of processed PUs correlates directly to the achieved FPS. With less PUs to process, the FPS is higher and vice versa. Therefore, there is a big gap between the smallest and highest achieved FPS.

7.1.2 Synthesis on Quartus

The Accelerator is synthesized using Intel's Quartus Prime to the Arria 10 FPGA. Table 7.4 presents the synthesis results. A bit less than a fifth of the available ALMs are used, most of them as registers. The design does not use any DPS blocks as there is no need for multiplications. The *High-Speed Serial Interface* (HSSI) channels as well as 9 *Phase Locked Loops* (PLLs) are used for PCIe communication. The last PLL is used in the Accelerator. The Accelerator functionality was verified with 150 MHz FPGA frequency and according to the synthesis 151.42 MHz maximum FPGA frequency is achieved.

Table 7.5 shows the synthesis results for each designed block. The *calculation* block is by far the largest part of the whole Accelerator. It takes 85% of the used ALMs. This is

Table 7.4 *Synthesis results.*

Synthesis summary	
Quartus Prime Version	17.1.1 Internal Build 593 SJ Standard Edition
Family	Arria 10
Device	10AX115S2F45I1SG
Logic utilization (total ALMs)	76871 / 427 200 (18%)
Total registers	54 849 / 854 400 (6%)
Total pins	35 / 960 (4%)
Total block memory bits	126 528 / 55 562 240 (< 1%)
Total DSP blocks	0 / 1 518 (0%)
Total HSSI RX channels	8 / 72 (11%)
Total HSSI TX channels	8 / 72 (11%)
Total PLLs	10 / 144 (7%)
Total M20K blocks	158 / 2 713 (6%)
ALMs used for memory	200
Fmax 100 ˚C	151.42 MHz
Fmax -40 ˚C	147.47 MHz

Table 7.5 *Block level synthesis results.*

	ALMs	Registers	Memory bits	M20K
Total	76 871	54 849	126 528	158
Accelerator core	68 546	46 146	0	0
Read and write	3 430	11 857	0	0
Calculation	65 116	34 289	0	0
Search area memory	0	0	20 480	112
PU memory	0	0	512	7
Results memory	0	0	128	2
Memory indexer	5 223	3 477	0	0
FIFO	19	64	16 384	13
DMA	324	1 039	16 384	4
PCIe control	2 759	4 123	72 640	20

mainly because it is entirely register based. *Read*, *write*, *memory indexer* and PCIe control blocks share the rest of the resources almost evenly. The memory usage is shared between search area memories, PU memory, FIFO and DMA blocks and the PCIe control.

7.1.3 Encoding speedup

The Accelerator is tested with different full HD sequences and the test results are listed on Table 7.6. The relative speedup is calculated from the test results and compared to the software only FS and fast HEXBS algorithm ran on the PC.

The encoding with the Accelerator is on average two times faster than the software only FS encoding. On the other hand, the Accelerator does not reach the speed of the optimized HEXBS algorithm on software.

Table 7.6 *Speedup comparison with Kvazaar.*

Sequence	**fs (FPS)**	**HEXBS (FPS)**	**Accelerator (FPS)**	**Speedup (fs)**	**Speedup (HEXBS)**
Beauty	1.39	5.22	2.65	1.91	0.51
Bosphorus	1.50	6.00	2.97	1.98	0.50
HoneyBee	1.29	5.79	2.61	2.02	0.45
Jockey	1.47	5.50	2.81	1.91	0.51
ReadySetGo	1.43	5.62	2.77	1.94	0.49
ShakeNDry	1.22	4.92	2.32	1.90	0.47
YachtRide	1.45	5.22	2.80	1.93	0.54
Average	**1.39**	**5.47**	**2.70**	**1.94**	**0.50**

The variation on the performance between the different sequences is caused by the different amount of motion and processed PUs. The tests are done without any software parallelization to measure the performance of the Accelerator compared to the software only algorithm.

7.2 Comparison with related work

Table 7.7 presents the comparison to the related work presented in Chapter 2.3. Only one HLS implementation was found in the literature, and the rest were designed using traditional hardware design tools.

The comparison with [15] is difficult as there are no clear results in their paper for the whole design but for the smaller parts separately. Their focus is more on comparing the HLS to the traditional RTL design with implementation time and code length and readability. The other related works did not use HLS tools for their designs nor are they integrated into the overall encoding process.

Authors in [16], [17] and [18] achieved encodings at 30 FPS with different settings and architectures. The Accelerator core proposed in this Thesis is four times faster than designs in related works. Though, when using it as a part of real encoding process the speed drops down to 2.70 FPS. Also, the proposed Accelerator uses only 8 × 8 block sizes making it less versatile.

However, all of designs in the related works use higher operating frequencies than the Accelerator proposed in this Thesis. They all also have wider search range compared to the proposed Accelerator. Further comparison with [18] is not meaningful as they use ASIC instead of FPGA as their platform. The area comparison is also difficult as the designs in the related works used Xilinx FPGAs and they have announced their areas in LUTs.

***Table 7.7** Comparison with related work.*

Architecture	Technology	Search range	Frequency	Area	Performance
Schewior et al. [15]	Virtex-7	-	-	-	-
Asano et al. [16]	Virtex-5	±32	269.3 MHz	66.4 kLUTs	1080@30 fps
Medhat et al. [17]	Virtex-7	±20	458 MHz	25 kLUTs	1080@30 fps
Medhat et al. [18]	65nm CMOS	±27	720 MHz	434 kgates	1080@30 fps
Software only	CPU	±16	-	-	1080@1.39 fps
Proposed Accelerator core	Arria 10	±16	150 MHz	76 871 ALMs	1080@122.12 fps
Proposed Accelerator	Arria 10 + CPU	±16	150 MHz	76 871 ALMs	1080@2.70 fps

In conclusion, the Accelerator core proposed in this Thesis is the fastest compared to the related work. It is also the only one to reach 30 FPS encoding with only 150 MHz operating frequency.

7.3 Discussion

The designed Accelerator core is capable of 122 FPS encoding for full HD sequences when tested without Kvazaar encoder. Its integration into the software encoding process makes it slower due to the communication limitations between the CPU and the FPGA. PCIe bus it not able to handle the high amount of data transfers needed to keep the Accelerator core at the Kvazaar's software encoding speed. This is mainly due to the number of pixels overlapping in the search area which is sent multiple times. Integrating the Accelerator to Kvazaar is complex as a lot of data must be transferred between the CPU and the FPGA.

As the designed Accelerator supports only 8 × 8 PUs, comparing it to the software optimized HEXBS algorithm using the same sized PUs is not fair. Using SAD reuse method to calculate the rest PU sizes in the Accelerator could give a significant speedup because most of the calculated SAD values could be reused. That is not possible with the HEXBS algorithm, and all the different PU sizes must be calculated separately. Implementing the rest of the PU sizes is one development path for the future work.

The basic idea of SAD reuse is to calculate all the SAD values for the smallest PUs and then combine them to achieve the SADs for the bigger PUs. For example, 64 × 64 PU can be divided into 64 small 8 × 8 PUs. Once all the 8 × 8 PUs are calculated and the values stored, they are used to compute the SAD values for the 16 × 16, 32 × 32 and 64 × 64 PUs recursively. However, with the current solution, implementing SAD reuse is not directly possible. The SAD values are not calculated in the same order as originally in Kvazaar and the SAD and the MVs are not stored to on-chip memories. There are various approaches how to implement SAD reuse mechanism. One way would be delaying or storing the calculated SAD values until they are needed for the bigger block. Another method would be calculating just the SAD values for the smaller blocks and storing them to a memory. Then going through all the stored SAD values and determining the best SAD and the MV for each required block size.

The communication bottleneck can be reduced designing an effective search area reuse method. Currently, many pixels are overlapping in the search area shared by adjacent PUs. Those pixels could be reused instead of sending them again when PU changes. One way of doing this is to send the biggest PU, 64 × 64, at once to the FPGA and distribute the needed pixels there for the smaller PU calculation. Combining effective search area

reuse with SAD reuse could significantly improve the Accelerator performance once integrated to Kvazaar.

Currently, the Accelerator takes less than 1/5 of the whole available logic of the used Arria 10 FPGA. As the Accelerator design is generic, up to 5 Accelerators fit to the FPGA. In theory, by duplicating the Accelerators in the FPGA, the performance should also become fivefold. However, the communication issue remains. With the current implementation, only IME part of ME is accelerated, and this makes the relative communication overhead between the platforms more significant. When implementing for example the FME stage of the ME process on the FPGA, the communication overhead reduces as most of the data already sent to the FPGA can be used again.

The Accelerator design process is not limited to this Thesis and the development continues in Ultra Video Group. The main goals in the future are to optimize the data transfer to the FPGA and modify the Accelerator to support all Kvazaar's PU sizes instead of only one.

8. CONCLUSION

In this Thesis a ME algorithm was chosen for hardware acceleration. Software exploration was done to Kvazaar to find the best settings for the ME algorithm. Then, an FPGA based accelerator was designed for the chosen FS algorithm using Catapult-C HLS design tool. HLS tools were used instead of traditional RTL tools because they offer more automation to the design process and make it faster, also verification is faster using HLS.

The Accelerator was synthesized on Arria 10 FPGA platform. The designed Accelerator was integrated to Kvazaar HEVC encoder. The Accelerator's memory architecture was also discussed in detail.

The designed Accelerator core was able to reach 122 FPS encoding speed for full HD video sequences before it was integrated as a part of Kvazaar's encoding process. The performance represents 66 times fold speedup compared to software only FS. Once integrated to Kvazaar the speedup was still almost two times fold. Compared to the literature solutions for hardware ME, the proposed Accelerator core is the fastest and works on the lowest frequency.

To reduce the role of communication, other ME tools, such as FME should be implemented also on hardware. Also, the search area reuse is an important feature to consider. In addition, to make the Accelerator more versatile, compatibility for the rest of the PU sizes should be implemented. These are the current development paths for the future work which continues in the Ultra Video Group.

REFERENCES

[1] Cisco, *Cisco Visual Networking Index: Forecast and Methodology*, 2016-2021, Jun 6, 2017.

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

[3] *Kvazaar HEVC encoder,* Available: https://github.com/ultravideo/kvazaar.

[4] M. Fingeroff, *High-Level Synthesis Blue Book.* Xlibris Corporation, 2010.

[5] *Vhdl*, in A Dictionary of Computer Science, 7th ed., A. Butterfield and E. N. Gerard, Eds. Oxford University Press, 2016.

[6] Mentor Graphics, *Catapult High-Level Synthesis datasheet*, pp. 4, 2017.

[7] I. Kuon, R. Tessier and J. Rose, *FPGA Architecture.* 2008, Available: http://ebook-central.proquest.com/lib/tut/detail.action?docID=3383629.

[8] C. M. Maxfield, *Chapter 1 - The Fundamentals, FPGAs: Instant Access,* pp. 1-12, 2008. Available: http://www.sciencedirect.com/science/article/pii/B9780750689748000016.

[9] Intel, *Intel® Arria® 10 Core Fabric and General Purpose I/Os Handbook*, 2019.

[10] *Pci*, in A Dictionary of Computer Science, 7th ed., A. Butterfield and E. N. Gerard, Eds. Oxford University Press, 2016.

[11] Intel, *Avalon® Interface Specifications*, MNL-AVABUSREF, 2019.

[12] G. J. Sullivan *et al*, *Overview of the High Efficiency Video Coding (HEVC) Standard*, IEEE Transactions on Circuits and Systems for Video Technology*,* vol. 22, pp. 1649-1668, 2012.

[13] M. Viitanen *et al*, *Kvazaar: Open-Source HEVC/H.265 Encoder*, Proceedings of the 24th ACM International Conference on Multimedia*,* pp. 1179-1182, 2016.

[14] P. Sjövall *et al*, *High-level synthesis design flow for HEVC intra encoder on SoC-FPGA*, in 2015, pp. 49-56.

[15] G. Schewior *et al*, *HLS-based FPGA implementation of a predictive block-based motion estimation algorithm — A field report*, Proceedings of the 2014 Conference on Design and Architectures for Signal and Image Processing*,* pp. 1-8, 2014.

[16] S. Asano, Z. Z. Shun and T. Maruyama, *An FPGA implementation of full-search variable block size motion estimation*, 2010 International Conference on Field-Programmable Technology*,* pp. 399-402, 2010.

[17] A. Medhat *et al*, *A highly parallel SAD architecture for motion estimation in HEVC encoder*, 2014 IEEE Asia Pacific Conference on Circuits and Systems (APCCAS), pp. 280-283, 2014.

[18] A. Medhat, A. Shalaby and M. S. Sayed, *High-throughput hardware implementation for motion estimation in HEVC encoder*, 2015 IEEE 58th International Midwest Symposium on Circuits and Systems (MWSCAS), pp. 1-4, 2015.

[19] B. Bross *et al*, *Inter-picture prediction in HEVC*, in High Efficiency Video Coding (HEVC): Algorithms and Architectures, V. Sze, M. Budagavi and G. J. Sullivan, Eds. 2014.

[20] A. J. Hussain and Z. Ahmed, *A Survey on Video Compression Fast Block Matching Algorithms*, Neurocomputing, 2018. Available: http://www.sciencedirect.com/science/article/pii/S092523121831261X.

[21] I. Kim *et al*, *Block Partitioning Structure in the HEVC Standard*, IEEE Transactions on Circuits and Systems for Video Technology, vol. 22, pp. 1697-1706, 2012.

[22] Ce Zhu, Xiao Lin and Lap-Pui Chau, *Hexagon-based search pattern for fast block motion estimation*, IEEE Transactions on Circuits and Systems for Video Technology, vol. 12, pp. 349-355, 2002.

[23] X. Li *et al*, *Fast motion estimation methods for HEVC*, 2014 IEEE International Symposium on Broadband Multimedia Systems and Broadcasting, pp. 1-4, 2014.

[24] N. Doan *et al*, *A hardware-oriented concurrent TZ search algorithm for High-Efficiency Video Coding*, EURASIP Journal on Advances in Signal Processing, vol. 2017, *(1),* pp. 78, 2017.

[25] J. Lin *et al*, *Motion Vector Coding in the HEVC Standard*, IEEE Journal of Selected Topics in Signal Processing, vol. 7, pp. 957-968, 2013.

[26] T. K. Tan *et al*, *Video Quality Evaluation Methodology and Verification Testing of HEVC Compression Performance*, IEEE Transactions on Circuits and Systems for Video Technology, vol. 26, pp. 76-90, 2016.

[27] G. Bjontegaard, *Calculation of average PSNR differences between RD-curves*, Apr 2-4, 2001.

[28] F. Bossen, *Common test conditions and software reference configurations*, Jctvc-L1100, 2013.

[29] A. Mercat, M. Viitanen and J. Vanne, *UVG dataset: 50/120fps 4K sequences for video codec analysis and development*, ACM Multimedia Systems Conference, 2020.