UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
INSTITUTO DE INFORMÁTICA
CURSO DE ENGENHARIA DE COMPUTAÇÃO

BRUNNO ALVES DE ABREU

Power-Aware Integer Motion Estimation Architecture for HEVC Video Encoding

Work presented in partial fulfillment
of the requirements for the degree of
Bachelor in Computer Engineering

Advisor: Prof. Dr. Sergio Bampi
Coadvisors: Eng. Guilherme Paim and MSc.
Mateus Grellert

Porto Alegre
January 2018

“Take the end of every day,
tie it up to every morning
and sail away.”
— JOHN MAYER

ACKNOWLEDGEMENTS

First of all, I would like to thank my parents, Fernando César Ávila de Abreu and Jaqueline Alves de Abreu, for fully supporting me in my choices and giving me advices when needed.

To all my professors, who have made their best to teach me all they could and helped me to find outside sources to learn even more.

To all my classmates, with whom I have shared the difficulties, stressful and happy moments we all have been through during our graduation years.

To my research colleagues, who have helped me, directly or indirectly, in my maturation as a researcher, and also in the completion of this work.

Special thanks to my advisor Sergio Bampi, and my co-advisors Mateus Grellert and Guilherme Paim, who have dedicated countless hours in helping me with doubts I had, to work with me in many other side projects, and to encourage me when I lacked the motivation.

ABSTRACT

The multimedia content traffic over the Internet is increasingly being represented by battery-powered devices, such as smartphones, tablets, etc. On the other hand, the increase in the density of components in a chip and, consequently, in the power dissipation, poses a problem for such devices. Despite the considerable increase in computational capacity in the last decades, the same growth has not been observed in battery life for such systems. Due to the fact that the multimedia content represents the most part of the Internet traffic, there is a need for optimizing these kind of applications, in order to compensate for the short battery life. The research in digital video coding is one of the areas focused on this kind of optimization, and one of its goals is to find solutions for reducing the dissipated power of the encoders. Motion Estimation is a key component in current video encoders, as it exploits the temporal redundancies of video sequences, through intensive searches for similarities in previously encoded blocks. It is one of the most critical and time-consuming tasks of the latest video coding standard HEVC, being responsible for more than 60% of the total encoding time on average. This work proposes the design and implementation of a power-aware hardware architecture for the Integer Motion Estimation stage. The architecture was synthesized for ASIC with 65 nm standard cells library. Power analysis are performed in some of its components using real input vectors, in order to decide the best architectural versions of the modules to be optimized. A power-aware cache memory hierarchy is also proposed, interfacing the off-chip DRAM (containing data from reference frames) and the Integer Motion Estimation architecture, with hit-rate results of up to 96.47%. We were able to decrease the off-chip bandwidth from 5.22 GB/s - considering that every access was requested directly to the DRAM - down to 0.18 GB/s, without considering any buffering mechanisms. Considering the whole system, we obtained an energy reduction of 94.5% when compared to the version without using any cache mechanisms.

Keywords: Video Coding. HEVC. Integer Motion Estimation. Cache Memory.

Arquitetura Eficiente em Potência da Estimação de Movimento Inteira para Codificação de Vídeo no padrão HEVC

RESUMO

O tráfego de conteúdo multimídia pela Internet vem cada vez mais sendo representado por dispositivos alimentados por bateria, como *smartphones*, *tablets*, etc. Por outro lado, o aumento na densidade de componentes em um *chip* e, consequentemente, na dissipação de potência, representa um problema para tais dispositivos. Apesar da capacidade computacional ter crescido consideravelmente nas últimas décadas, o mesmo crescimento não foi observado no tempo de vida das baterias para tais sistemas. Devido ao fato do conteúdo multimídia representar a maior parte da taxa do tráfego na Internet, há uma necessidade de otimizar esses tipos de aplicações, para compensar o curto tempo de vida das baterias. A pesquisa em codificação de vídeo digital é uma das áreas focadas neste tipo de otimização, e um dos seus objetivos é a busca de soluções para reduzir a potência dissipada dos codificadores. A Estimação de Movimento é um componente chave nos codificadores de vídeo atuais, devido ao fato de explorar redundâncias temporais de sequências de vídeo, através de intensas buscas por similaridades em blocos anteriormente codificados. Consequentemente, esse estágio da codificação é um dos mais críticos e demorados do último padrão de codificação de vídeo HEVC, sendo responsável por mais de 60% do tempo total de codificação, em média. Este trabalho propõe o projeto e implementação de uma arquitetura eficiente em potência para o estágio da Estimação de Movimento Inteira. A arquitetura foi sintetizada para ASIC, com uma biblioteca de *standard cells* de 65 nm. Análises de potência são feitas em alguns dos componentes, para a decisão das melhores versões arquiteturais dos módulos a serem otimizados. Uma hierarquia de memória cache focada em eficiência de potência também é proposta, interfaceando a memória DRAM *off-chip* (que contém dados relativos aos quadros de referência) e a arquitetura da Estimação de Movimento Inteira, com resultados de *hit-rate* de até 96.47%. A solução proposta reduz a banda off-chip de 5.22 GB/s - obtido considerando que todo acesso é requisitado diretamente para a DRAM - para 0.18 GB/s, sem considerar mecanismos de bufferização. Considerando o sistema como um todo, foi obtida uma redução energética de 94.5% quando comparado com a versão que não utiliza mecanismos de cache.

Palavras-chave: Codificação de Vídeo, HEVC, Estimação de Movimento Inteira, Memória Cache.

LIST OF ABBREVIATIONS AND ACRONYMS

ASIC	Application Specific Integrated Circuits
CTU	Coding Tree Unit
CU	Coding Unit
DM	Direct Mapped
DS	Diamond Search
FA	Fully Associative
FME	Fractional Motion Estimation
FPGA	Field-Programmable Gate Arrays
FS	Full Search
GPP	General Purpose Processor
HD	High-Definition
HEVC	High Efficiency Video Coding
HS	Hexagon Search
IME	Integer Motion Estimation
ME	Motion Estimation
SS	Star Search
TZS	Test Zone Search
UHD	Ultra High-Definition

LIST OF FIGURES

LIST OF TABLES

CONTENTS

1 INTRODUCTION

In the recent years, advances in technology have allowed for better quality applications, leading to an increasing demand for more sophisticated services. Digital video applications are one of the main categories affected by the technology advances, given that they allowed the requirements for higher resolution video services to be met. Real-time video content has also benefited from technology improvements, allowing for personal video broadcasting on a worldwide scale, through streaming services such as Twitch or Youtube. The growing popularity of these services allied with the need for better quality content has led to current predictions showing that videos will take a bandwidth share of 82% of the whole Internet traffic by 2021 (Cisco, 2017).

Digital videos are usually not handled in their uncompressed form because this requires a huge amount of resources to store and transmit this information. To exemplify, a 10-minute Full High Definition (Full HD) 1920×1080 video, recorded at 30 frames per second, with each pixel being represented by 3 bytes, would require more than 104GB to be stored. In order to transmit this video as a real-time service, a bit-rate of more than 177 MB/s would be required. The video size and bandwidth becomes even worse when higher resolutions are considered, such as the increasingly popular Ultra High-Definition 4K (UHD 4K) (3840×2160 pixels). The International Telecommunication Union Radiocommunication Sector (ITU-R) recommendation for UHD Television (UHDTV) states that resolution should be increased in both spatial and temporal axes (ITU-R, 2015), so higher frame rates of up to 120 fps will have to be supported. Table 1.1 shows storage and bit-rate requirements for a 10-minute sequence with common spatial and temporal resolutions, considering 3 bytes per pixel. The size and bit-rate values are given by equations 1.1 and 1.2, in which W and H respectively refer to the width and height of the video, N refers to the representation of each pixel, in bytes, F denotes the frame-rate per second, and t refers to the time duration of the video sequence, in seconds.

$$Size[bytes] = W \times H \times N \times F \times t \tag{1.1}$$

$$BitRate[bytes/s] = Size/t \tag{1.2}$$

Table 1.1 shows that the required transmission rates for uncompressed data are prohibitive on current communication technology, so there is a real need for compressing (or encoding) this information before transmission.

Video compression is based in the principle of finding redundant information

Table 1.1: Raw video sizes and required transmission bit-rate for different spatial and temporal resolutions (sequence length: 10 minutes).

Resolution	FPS	Storage Requirement (GB)	Transm. Bit-rate (MB/s)
832×480	30	20.08	34.27
1920×1080	30	104.28	177.98
3840×2160	30	417.14	711.91
3840×2160	120	1668.55	2847.66
7680×4320	30	1668.55	2847.66
7680×4320	120	6674.19	11390.63

within frames of a video and suppressing most of these redundancies to minimize the number of bits needed to represent the video sequence and to make the required storage size and transmission rate more feasible.

While encoding reduces the amount of information used to represent videos, it also introduces a new problem regarding computation. Modern video encoders perform many time-consuming operations to compress data efficiently, increasing the time and energy required for this task. The demands for higher resolutions aggravate this issue because the amount of operations needed to encode each sequence is proportional to the input size. Real-time systems are particularly affected because in this case data has to be encoded at a minimum frame-rate to optimize the user experience (typically more than 25 frames per second).

The increased computational effort of video encoders leads to an additional issue on battery-powered devices, such as smartphones and camcorders. Due to limited battery resources, the encoding task needs to be executed as efficiently as possible. Using general purpose processors (GPPs) for video applications is inefficient, since the arithmetic units of these devices are not designed to compute video-coding operations efficiently.

Designing dedicated hardware architectures is one of the main solutions to tackle the power and energy issues, since they eliminate the overhead of GPPs, given that they are designed solely for a chosen application. Application Specific Integrated Circuits (ASICs) are completely designed for specific domains, contrasting with GPPs. However, cost and manufacturing time become an issue when considering the use of ASICs. Field-Programmable Gate Arrays (FPGAs) are a good balance between GPPs and ASICs. FPGAs achieve better performance when compared to GPPs for the same application and, despite not being as dedicated as an ASIC, their time-to-market is smaller, since no chip layout and subsequent manufacturing steps are needed. However, when power-efficiency is the main design restriction to take into account, the dedicated ASIC remains as the best option.

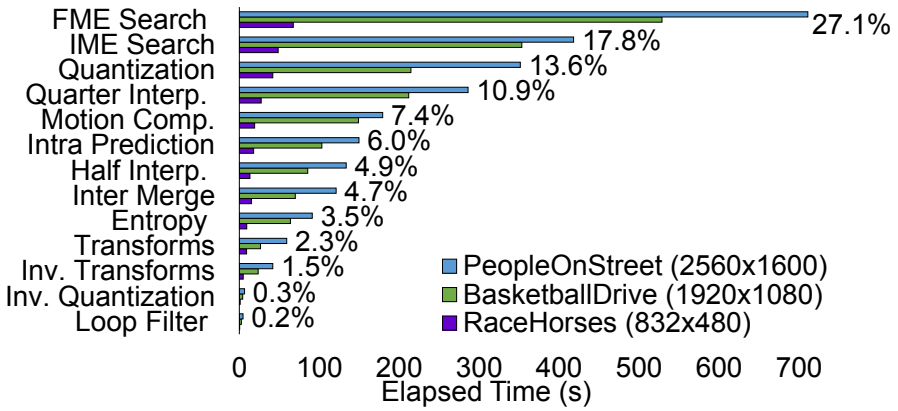
High Efficiency Video Coding (HEVC) (ITU-T and ISO/IEC, 2013) is the state-of-the-art video coding standard, and it was proposed focused on the increasing demands for higher resolution videos. When compared to its predecessor, the H.264/AVC (ITU-T and ISO/IEC JCT, 2011), HEVC achieves up to 50% bit savings for the same video quality (SULLIVAN et al., 2012), by applying more sophisticated techniques and algorithms.

Motion Estimation (ME) is part of the HEVC standard and refers to one of the most time-consuming processes in video encoding. ME is responsible for finding most of the redundancies in videos, which is the reason why it is repeatedly executed several times for each frame of the video sequence. Fig. 1.1 presents an analysis using GProf (GProf, 1997) for three different video sequences and shows that the Integer and Fractional stages of the ME (IME and FME) are responsible for most of the execution time in the encoding process.

The proposal of this work consists in analyzing and implementing modules related to the Integer Motion Estimation while focusing on a power-aware design. Three main parts are presented in this report. First, a cache memory hierarchy is proposed, interfacing the IME architecture and an off-chip memory, given that most of the works in the ME literature tend to abstract the memory communication required. A design space exploration of the absolute differences operator in the Sum of Absolute Differences (SAD) module, used by the IME, is also performed in this work. Lastly, a Finite-State Machine (FSM) for the IME is implemented. The modules are synthesized for ASIC using a commercial tool (Cadence RTL Compiler), targeting the ST 65 nm CMOS standard cells library.

This document is structured as follows: section 2 gives a background on the main concepts regarding this work; section 3 presents related works found in the literature that address similar topics in ME implementation; section 4 details the methodology used, in-

Figure 1.1: Time percentage of each stage in the video encoding process.



Source: (GRELLERT; BAMPI; ZATT, 2016)

cluding the main design decisions and methods for obtaining the results; section 5 presents the results regarding power, energy, area and throughput values for the designed architectures; and section 6 presents the conclusions, briefly summarizing the contributions of this work, highlighting possible future paths for further research.

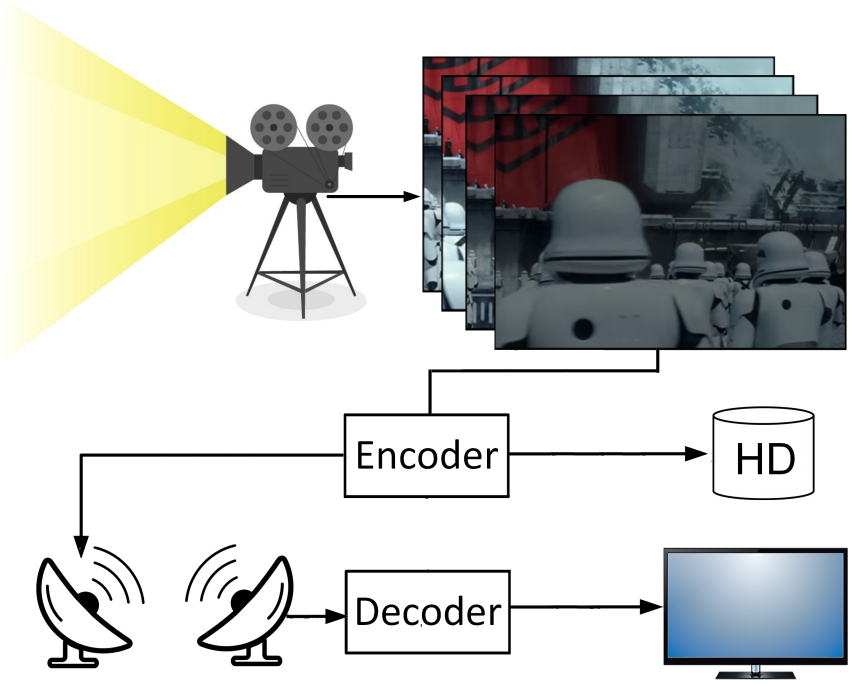
2 BACKGROUND

In order to understand the terminology used in the remaining sections, the presentation of some of the basic related concepts is essential. This chapter details the topics connected to the scope of this work, including terms related to video coding, motion estimation and cache memories.

2.1 Video Coding

The video coding process entails the operation of a video encoder and of a video decoder. The process starts with the capture of a real-life scene by a filming device, which generates a set of discrete scenes, i.e. frames, which corresponds to the raw video. The video encoder is responsible for applying compression techniques and transforming a raw video in a sequence of bits (video bitstream) according to a given standard. Next, the video is combined with other syntax elements at the transport layer (to be sent by a transmitter), so that a station can receive it, then extracts the original video source bitstream from the transport stream, and processes it with a decoder which supports the standard for which that bitstream was generated. The video can also be stored for future use. This process is depicted in a simplified way in Fig. 2.1.

Figure 2.1: Simplified scheme of encoding and decoding in video transmission.



Source: The Author

The state-of-the-art video coding standard is the High Efficiency Video Coding (HEVC/H.265), which is a more sophisticated version of the previous standard H.264/AVC. HEVC doubles the compression for the same video quality when compared to H.264, and it does that by processing more data and applying more sophisticated operations than its predecessor. However, these video compression gains incurred in 1.2-3.2× increase in computational effort on the encoder side, in comparison with the previous standard (SULLIVAN et al., 2012).

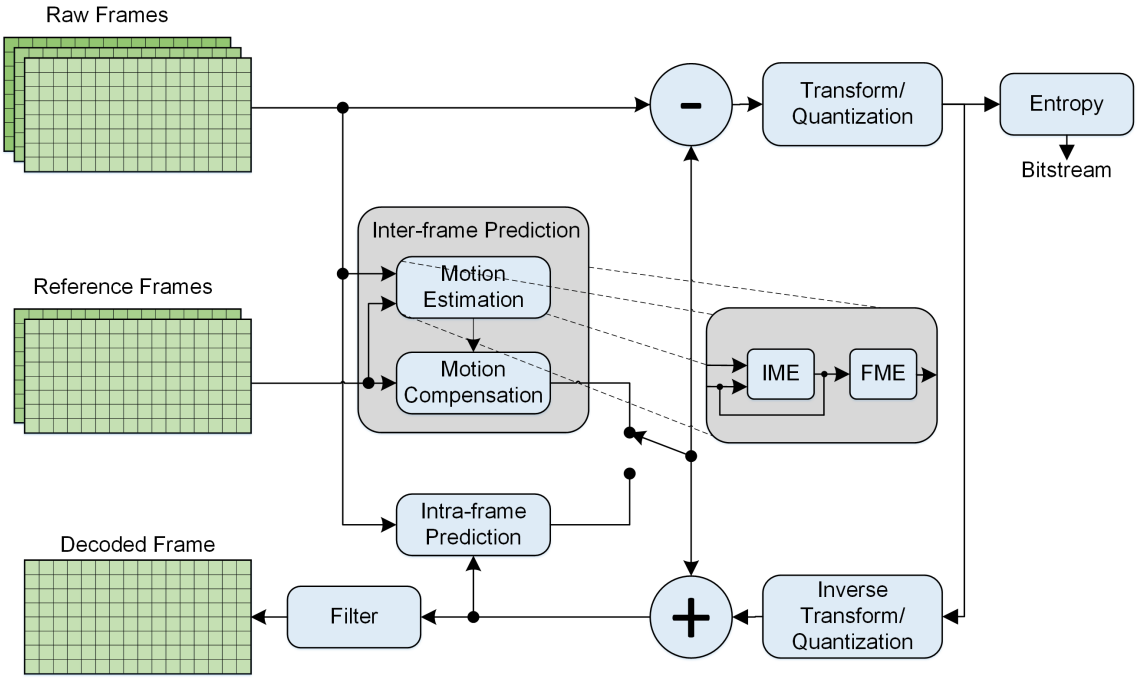
The video coding standard defines only the specifications for which the decoder has to comply. The encoder can be implemented in different ways, with different algorithms and in various hardware platforms, as long as the output bitstream generated by the encoder complies to the standard, namely the encoded bitstream can be processed by any standard-compliant decoder. The reference software for the HEVC standard is the HEVC Test Model (HM) (JCT-VC, 2016), which is freely distributed to document all the features and tools of the standard. The x265 (x265, 2016) software is also a well-known encoder compatible with the HEVC standard, with support for executing in multi-cores with thread-level parallelism. The x265 executes the encoding much faster than the reference software.

2.1.1 Video Encoder Diagram

The diagram of a general video encoder is shown in Fig. 2.2. The process starts by splitting each frame of the video sequence in blocks called Coding Tree Units (CTUs). HEVC defines 64×64 as the default and also the largest CTU size. Each CTU of the frame to be encoded is applied in the stages of the presented diagram.

CTUs are split into smaller squared blocks, called Coding Units (CUs), in a quad-tree partitioning scheme, based on heuristic decisions that are taken by the encoder. HEVC supports CU sizes of 8×8, 16×16, 32×32 or 64×64. The intra and inter-frame prediction executions are responsible for finding and resolving spatial and temporal redundancies, searching for similar information compared to the block being encoded, so that only residual information need to be sent to the encoder output, along with additional coding information for the CU that allows the decoder to recover the original block. In order to find the best possible redundant blocks, CUs are split into several possible partitions, called Prediction Units (PUs), for which the prediction algorithms are applied. The PUs can be categorized into Symmetric Motion Partitions (SMP) and Asymmetric Mo-

Figure 2.2: Video encoder diagram.



Source: The Author

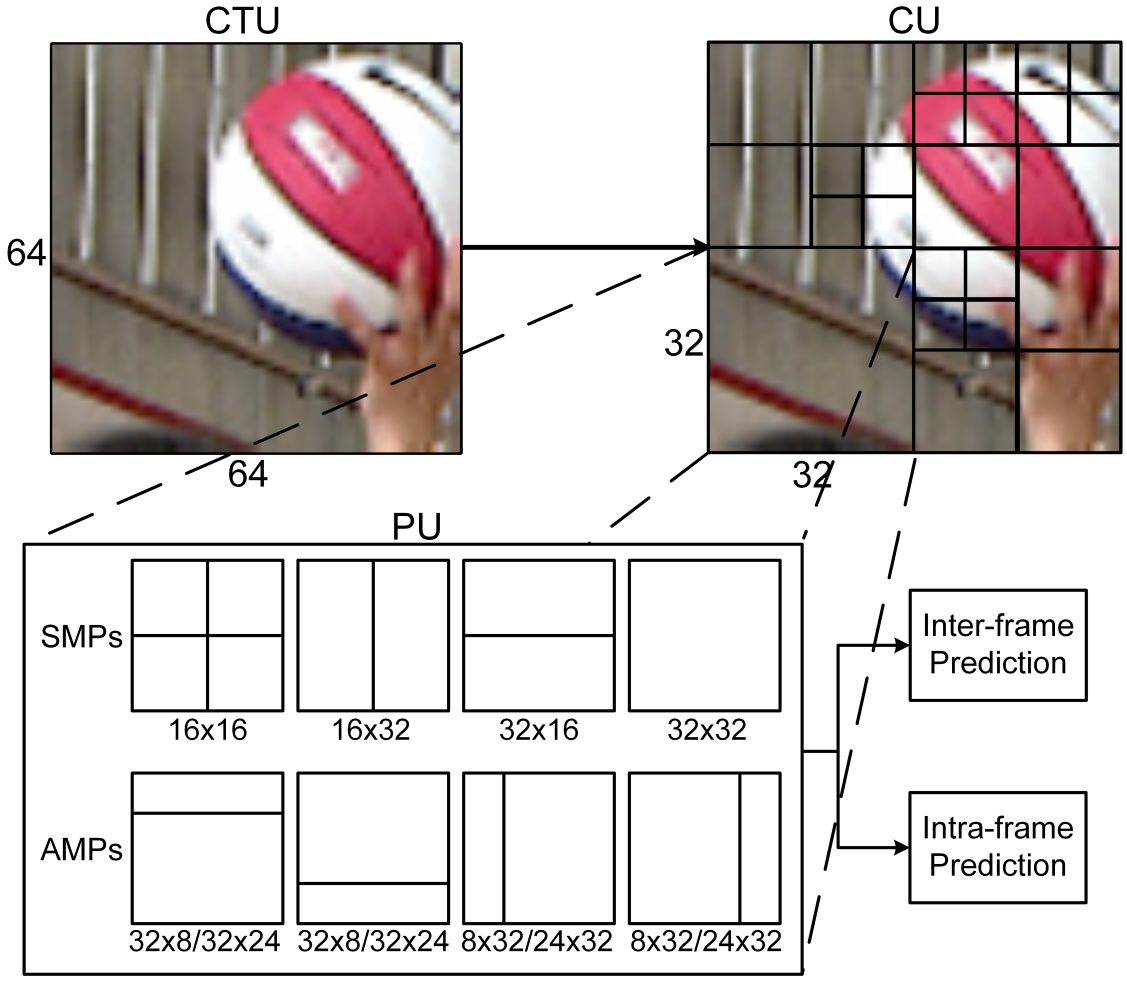
tion Partitions (AMP), for which the inter-frame redundancies are searched for. Fig. 2.3 shows the split from CTU to PU level, highlighting the PUs of type SMP and type AMP for a 32×32 CU size example.

Ideally, encoders would need to apply intra and inter-frame partitioning algorithms for every possible PU partition in a given Coding Unit. However, given the intense memory accesses and the demanding computations that would be required - hence increasing the decoding time and energy spent per frame to be encoded - encoders usually employ heuristics for some of the partitions not to be evaluated in some iterations, while still obtaining acceptable compression results. These heuristic decisions are left to the encoder developer, to determine whether the PU will be using an intra or an inter-frame partition, and of which type and size of CTU partitions, of CUs and PUs will be employed.

After applying algorithms to find the best redundancies (best candidate blocks) from either intra or inter predictions, a residual block is calculated. Transform and quantization stages are applied to that residual block, allowing for the introduction of lossy compression in the encoding of these residuals. Before being sent to the bitstream output, an entropy algorithm is applied to the block, to exploit entropic redundancies in the data.

The coded block also needs to be decoded in this encoding scheme, by applying inverse quantization and inverse transform functions in the encoder also, so that the en-

Figure 2.3: CTU partitioning scheme.



Source: The Author

coded frame can be recovered and stored to be used as reference frame for encoding other frames. These inverse operations are needed because the inter-frame prediction uses information of previously coded frames to find temporal redundancies in the current frames, so these previously encoded frames need to be ready to be analyzed.

2.1.2 Motion Estimation

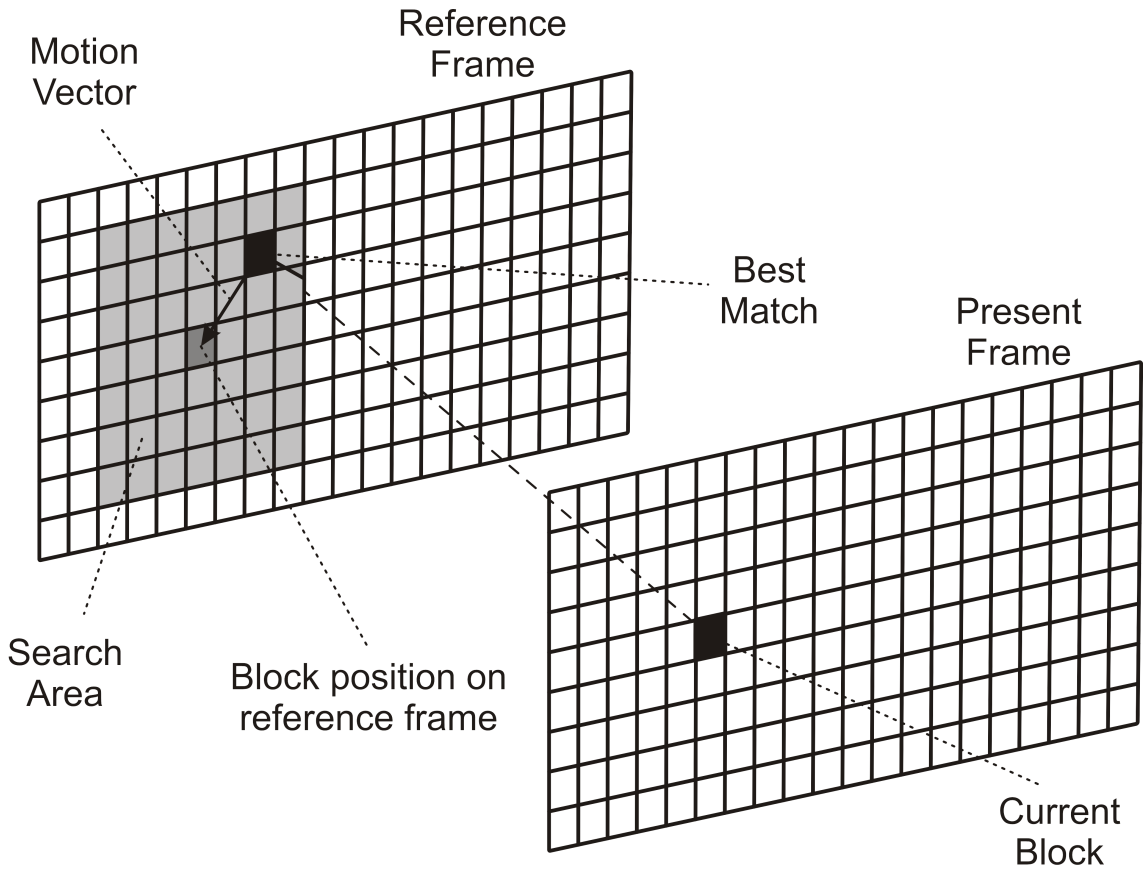
ME is a stage inside the inter-frame prediction in video encoding responsible for resolving temporal redundancies in a video. ME is applied to each block of the video frame and finds the most similar block compared to the one being encoded. Thus, the information needed to be sent to the output of an encoder is just the difference between both blocks (namely, the block of residuals) and a vector pointing to the best matching block, so that the decoder has sufficient information, along with the previously decoded

frames, to recover the original block. The ME is split into two subsequent stages: Integer Motion Estimation (IME) and Fractional Motion Estimation (FME). These two modules are briefly explained in the following subsections, focusing on the IME module, given that this is the main encoding stage which is dealt with in this work.

2.1.2.1 Integer Motion Estimation

IME finds the best matching block between PUs in two different frames, using only integer-pixel displacement vectors to compare blocks, by applying a search algorithm for the blocks being encoded in the present frame. This search algorithm defines a pattern of positions - represented by motion vectors with integer-pixel x- and y-components - in which the most similar block will be searched in the reference frame. The search algorithm is applied in a given search area of the frame, which consists of a window smaller than the frame itself, because image patterns tend to slightly displace from the area where they were in a previous (past) frame. Fig. 2.4 generically shows the concepts involved in the IME.

Figure 2.4: Generic search in a previously coded frame.



Source: (Roger Endrigo Carvalho Porto, 2008)

The following subsections present some of the main search patterns and algorithms mentioned in the literature.

2.1.2.2 Full Search

The Full Search (FS) is the most basic algorithm found in the literature. FS applies the search for the most similar block by displacing to every pixel in the search window, and it always finds the best possible matching block contained in the search window. For that reason, FS maximizes the temporal redundancy, resulting in a smaller bitstream at the encoder output.

Due to a large number of candidates being tested, FS requires the highest number of memory accesses and calculations among all the search algorithms. For that reason, real-time implementations employ other solutions and search heuristics, so that the search is performed for a smaller number of motion vector displacement candidates, while still attempting to find residual blocks very similar to the best possible.

2.1.2.3 Test Zone Search

Test Zone Search (TZS) is the algorithm used in the latest versions of the HEVC HM reference software. TZS applies more than one search pattern and has a more sophisticated search flow, so that very similar matching blocks can be found with fewer calculations. The algorithm is divided into four subsequent stages (Cássio Rodrigo Cristani, 2014), each of which is dependent on the previous one. For that reason, pipeline-based architectural implementations waste a high amount of additional cycles, given that the pipeline needs to be totally emptied so that all the processing of a previous stage can be performed before starting the next one. These stages are explained in the following items.

a) Vector Prediction: this stage is responsible for examining resulting motion vectors of the neighboring blocks of the PU being encoded, generated from their previous ME execution. Then, the block being encoded is compared to each block pointed by these vectors, and the position of the most similar one is defined as the initial center of the search.

b) First Search: a diamond-shaped search is performed, starting from the position defined by the Vector Prediction. The diamond pattern tests four, eight or sixteen candidates around the center, depending on the distance between samples, which is incremented after each iteration. If the search finds a block more similar than the one in

the current center in an iteration, the center is redefined as the position of that block. The search stops when the best candidate remains as the center after three iterations. Fig. 2.5 shows the shapes of the first three iterations.

c) Raster Search: this stage searches through the entire search area, starting from the top-left corner, and skips some neighboring candidates by defining a raster step constant. This stage is illustrated in Fig. 2.5 for a constant of 2.

d) Refinement Search: the last stage works the same way as the First Search. The main difference is that the default tolerance in the HM software for this level is two instead of three iterations.

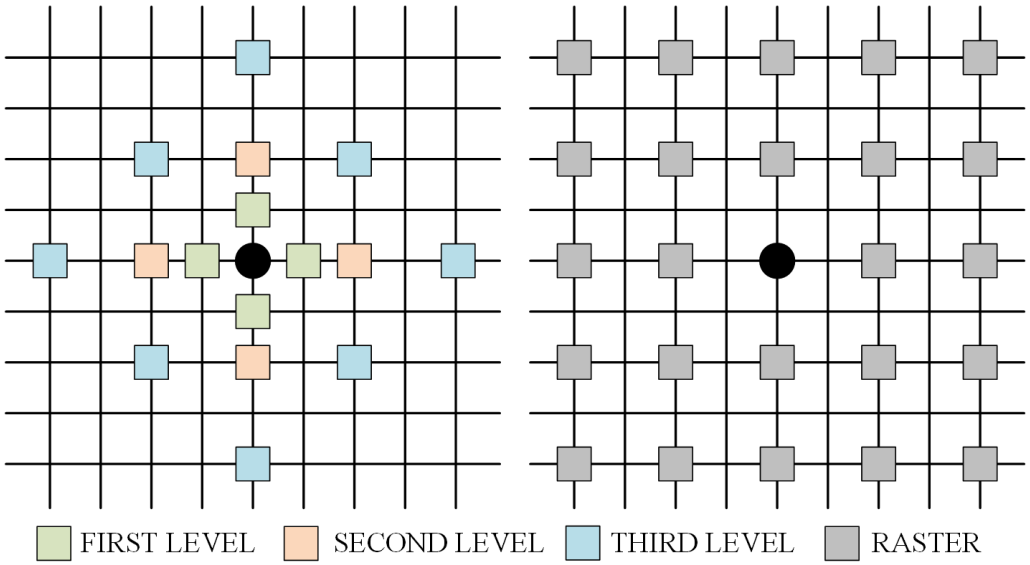
2.1.2.4 Hexagon Search

The Hexagon Search (HS) is the search algorithm implemented by the x265 software. HS is split into three main steps, with dependencies between them, just like in the TZS. The following items detail the HS.

a) Motion Prediction: this stage works similarly to the Vector Prediction in the TZS algorithm, only differing in the number of candidates evaluated to decide the initial center of the search.

b) Hexagon: this step performs a six-point search in a hexagon-shaped format around the center of the search, considering the initial center is defined by the previous step. Whenever there is a candidate more similar to the block being encoded than the one

Figure 2.5: Search shapes for the TZS.



Source: The Author

in the center, the new center is set as the vector associated with that new candidate. Then, the iteration is applied again, starting from the new center of the search. This stage stops when none of the candidates are better than the current center.

The x265 software applies an optimization to this stage: starting from the second iteration of the hexagon search, only three candidates need to be evaluated instead of six, since the three remaining points will always have been evaluated in the previous iteration.

c) Square Refinement: after the Hexagon step defines the best candidate, an 8-point square refinement is applied around that point. The final vector value is defined by the best candidate evaluated at this stage. If none of the candidates are better than the current center, then the center defined by the previous step is the best vector.

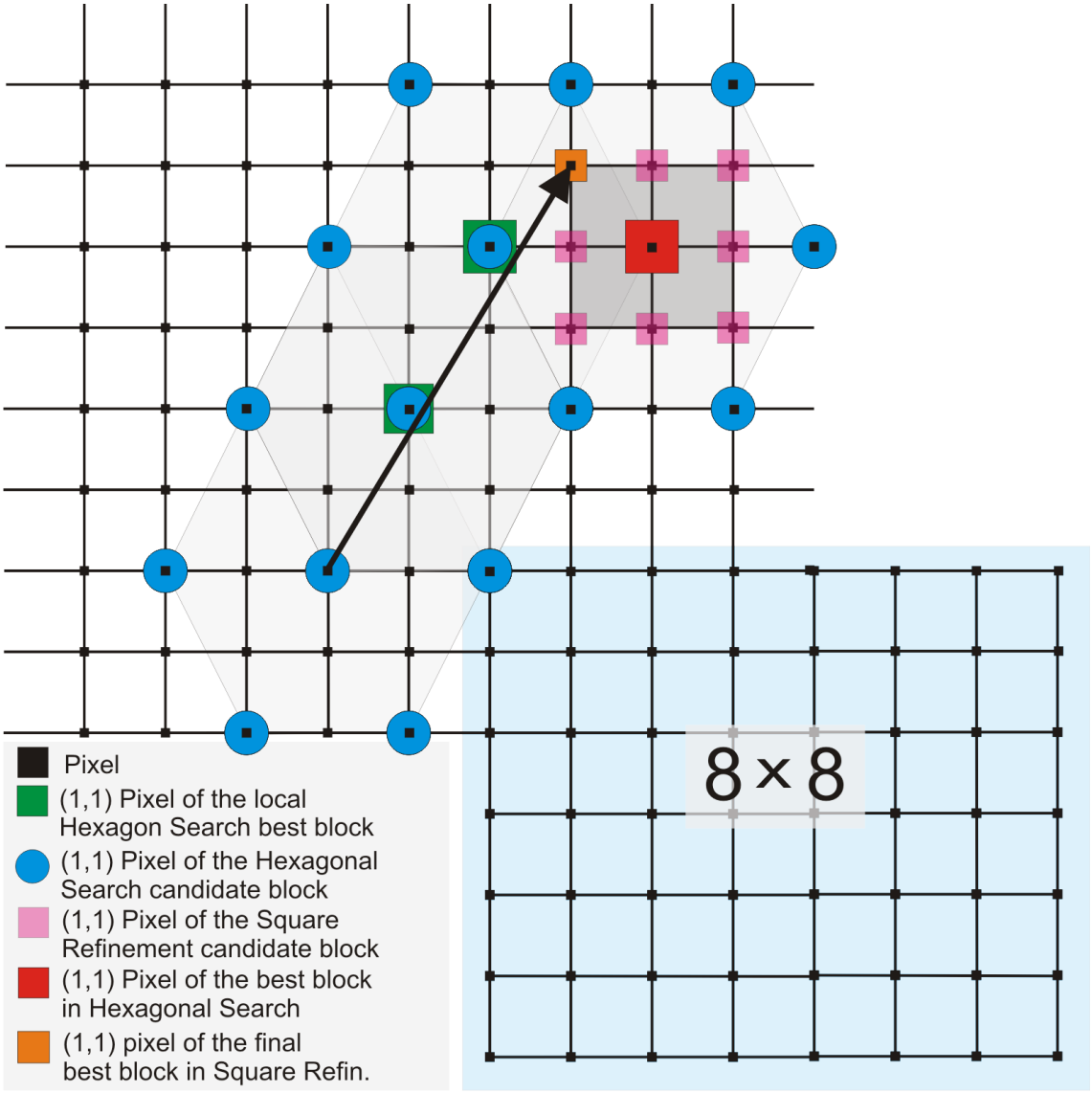
Fig. 2.6 illustrates the whole process of the HS algorithm, considering an 8×8 PU, in which four iterations of the Hexagon are performed. In the last one, no candidates were more similar to the block being encoded than the center, so the square refinement was applied, resulting in a better block found in one of its eight candidates.

2.1.2.5 Fractional Motion Estimation

Fractional Motion Estimation (FME) is applied after the IME and it is responsible for finding the best match at the fractional-pixel level, starting from the most similar block found in the IME. Since the frames are formed only by integer pixels, FME requires the use of an interpolator to estimate fractional pixels positioned between the integer pixels of the image. FME is responsible for increasing image quality in video sequences due to the fact that real-life patterns most frequently move at a rate that is not a good match to the integer-pixel displacement between two simultaneous video captures.

HEVC defines 48 possible candidates to be compared in the FME: 8 half-precision and 40 quarter-precision candidates. Fig. 2.7 presents the set of fractional points needed to gather all the information regarding the 48 fractional blocks. In this figure, green positions correspond to integer pixels, blue positions correspond to half-precision pixels and white positions correspond to quarter-precision pixels. Half and quarter-precision pixels are generated by interpolation using 7-taps and 8-taps FIR filters, depending on the specific point. The highlighted partition shows the 48 possible points to which comparisons will be performed.

Figure 2.6: Search shapes for the HS.

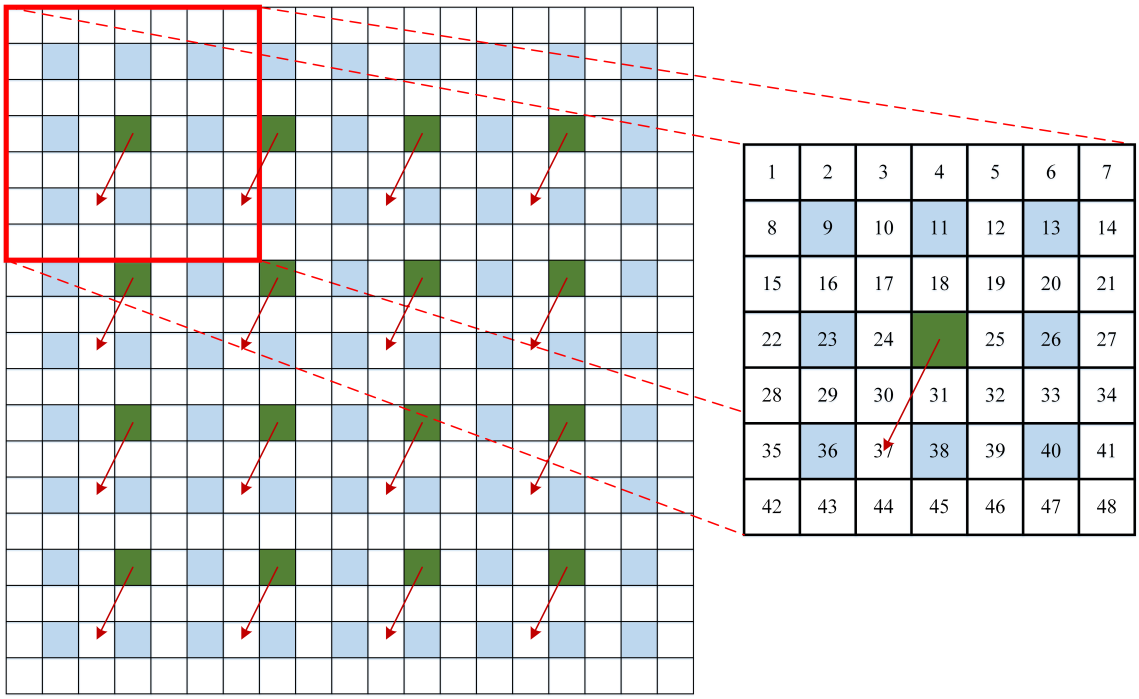


Source: (SILVEIRA et al., 2017)

2.1.3 Metrics for Block Similarity

Several metrics can be applied to determine the degree of similarity between two blocks. They differ from each other in ease of implementation, efficiency and result accuracy, i.e., how precisely they can define the similarity between the blocks. The main metrics used to estimate block similarity in video codecs are shown in the next subsections.

Figure 2.7: FME fractional pixels window for a 4×4 PU.



Source: The Author

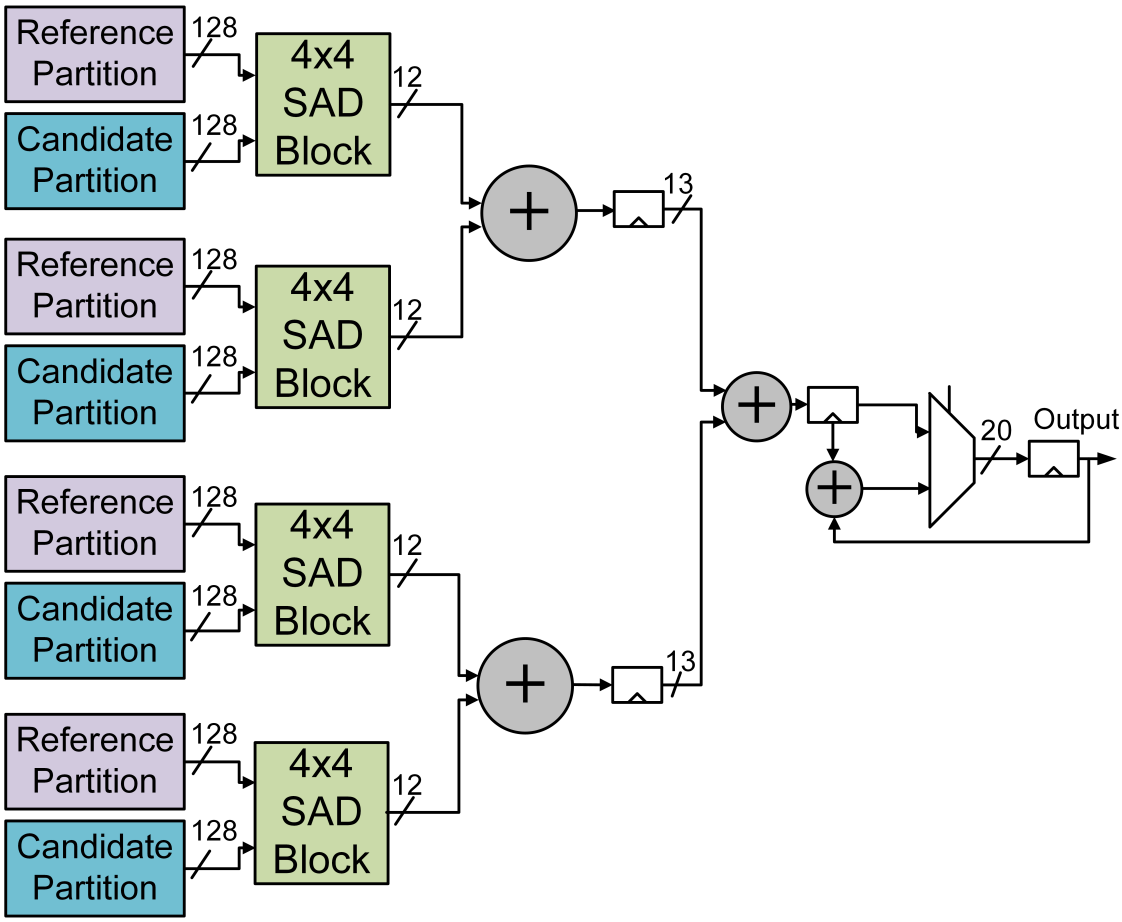
2.1.3.1 Sum of Absolute Differences

Sum of Absolute Differences (SAD) is the simplest metric used in the video encoding process and it is applied by calculating the differences between the co-localized pixels of a current and a candidate block, performing an absolute operation in these differences, and then adding the values. SAD is employed in the IME and is also one of the most used metrics in the video encoding process, representing, on average, 22.4% of the encoding time in the HM reference software (ABREU et al., 2017). The complete formula is given by 2.1, in which O and R denote the current and the candidate blocks, respectively; m and n refer to the width and height of the blocks (which have the shape of the PU being considered).

$$SAD = \sum_{i=0}^{m-1} \sum_{j=0}^{n-1} |O_{i,j} - R_{i,j}| \tag{2.1}$$

SAD architectures are mostly implemented using subtractors, absolute operators, and an adder tree, with an accumulator on the output so that SAD for bigger blocks can also be calculated. Most architectures use pipeline schemes so that several sum stages can be done concurrently and the resulting critical path is shortened. An 8×8 SAD architecture is shown in Fig. 2.8, in which the sizes are presented in bits.

Figure 2.8: 8×8 SAD architecture.



Source: The Author

The presented architecture calculates SAD for an 8×8 block in each cycle after the pipeline is filled. The 4×4 SAD blocks correspond to simple adder trees, including the subtractors and absolute operators. The 2-1 multiplexer is used to extend the block sizes possibilities so that receiving 8×8 blocks several times would allow the architecture to calculate SAD for larger block sizes. Considering a generic SAD architecture, the number of cycles required to calculate an $H \times W$ block is given by 2.2. In this formula, $input_W$ is the input width, in bytes, of all the candidate partitions; H and W refer to the height and the width of the PU to be calculated, respectively; and P_{depth} corresponds to the number of pipeline levels in the architecture.

$$cycles = P_{depth} + (H \times W)/input_W(bytes) \tag{2.2}$$

It should be noted that an increase in the minimum block size of the architecture implies in more area for the dedicated parallel hardware operators, resulting in power and energy increases. However, small architectures spend more time calculating SAD for

bigger blocks, since the number of accumulations needed is higher.

2.1.3.2 Sum of Absolute Transformed Differences

The Sum of Absolute Transformed Differences (SATD) is a more complex metric for similarity evaluation during the inter-prediction process. SATD is calculated by taking the frequency transform of the differences between pixels of current and candidate blocks. Equation 2.3 presents the formula to calculate SATD.

$$SATD = \sum_{i=0}^{m-1} \sum_{j=0}^{n-1} |T_{i,j}| \tag{2.3}$$

In HEVC, SATD is the metric used by default during the FME, and Hadamard is the transform function used for that space-to-frequency transformation. Although SATD is a more accurate metric to determine the most efficient block match, its hardware architectures have a more complex implementation when compared to SAD architectures.

2.2 Cache Memory

Cache memories are responsible for decreasing the latency of the accesses to main memories, by taking advantage of temporal and spatial localities and store previously used data in faster and smaller memory modules, since there is a high possibility that these data can be reaccessed in the future (PATTERSON; HENNESSY, 2007).

For the most of this work, simulations related to cache memories were performed, and a caching methodology for the IME was proposed. For that reason, it is essential to be familiar with the basic concepts regarding such memories, in order to fully understand the concepts presented in the methodology. The following subsections briefly present the concepts regarding the three types of classic cache memories: Direct Mapped (DM), Set-Associative and Fully Associative (FA) cache. Generically speaking, the DM and the FA caches are a subset of the set-associative cache. For clarity purposes, usually these cache address mapping cases are explained separately.

2.2.1 Direct Mapped Cache

The direct mapped cache is the simplest cache model, due to the fact that each one of the memory blocks can only be mapped to a single cache block. The index and tag bits used are shown below, in 2.4 and 2.5. In these formulas, $nrlines$ refers to the number of lines of the cache, $addr$ refers to the address requested by the processing module, and $sizeAddr$ denotes the size of the address, in bits.

$$index = addr[(\log_2 nrlines) - 1 : 0] \tag{2.4}$$

$$tag = addr[(sizeAddr - 1) : \log_2 nrlines] \tag{2.5}$$

2.2.2 Set-Associative Cache

The set-associative cache is the generic model for cache memories. In this case, each memory block can be addressed to n cache blocks, where n is the associativity. In this case, when searching for some data with a given tag, every block of the respective set must be searched. For that reason, more comparators are needed when compared to DM caches. In set-associative caches, the replacement policies to be implemented when writing data to the cache must also be considered. For example, if a First-In-First-Out (FIFO) replacement policy is used for a 2-way cache, when writing to a given set, the position that has been written first out of the two positions will be overwritten. In the case of a Least Recently Used (LRU) policy, as the name suggests, the position that has the oldest usage out of the possible positions is replaced. The tag and index bits for a set-associative cache is given by 2.6 and 2.7, where $assoc$ refers to the cache associativity.

$$index = addr[(\log_2(nrlines/assoc)) - 1 : 0] \tag{2.6}$$

$$tag = addr[(sizeAddr - 1) : \log_2(nrlines/assoc)] \tag{2.7}$$

2.2.3 Fully Associative Cache

The fully associative (FA) cache is the extreme or ideal case of a set-associative cache, in which the number of sets is maximum, i.e. with a size equal to the number of cache lines. In this case, for a given memory address, every tag of every line of the cache is compared, as in a table search, in order to determine if the requested address results in a miss or a hit. FA caches in hardware are generally not used as much, due to the high number of comparators required (as many as the cache lines) and, consequently, hardware area usage. However, they are used to determine and estimate an upper level to the hit-rate. In this FA cache, none of the bits are used as indices, and all the bits of the address are used as the tag. FA caches are used to store in-processor the most recent memory page addresses translated from virtual address to physical address in virtual memory support in hardware; this FA cache is called "translation look-aside buffer" in processor architecture literature.

3 RELATED WORK

ME modules can be found in the literature frequently divided into the IME and the FME, due to the focus of each work in optimizing specific parts of each of these modules.

Sanchez et al. (2015) employ two algorithms and their respective hardware implementations for an IME module. The architectures are based on the use of parallel instances of a diamond-shaped search. The architectures achieve real-time throughput for 1080p sequences. Even though the architecture presented some quality loss when compared to other solutions, the proposed architecture has fewer data dependencies, more regular memory accesses and regular cycles to encode a single block. Results show gate count of 150k with a 42.3MHz frequency and power dissipation of 12.5 mW and 13.5 mW (for each presented architecture), for a 90nm cell library. However, even though this work considers some generic memory aspects, the analysis does not present specific power results for the memory hierarchy proposed.

Porto et al. (2011) propose an architecture for the IME and, even though the proposal includes an internal memory, the implementation and power results for that specific memory module are not presented.

The work by Yuan et al. (2013) also implements an architecture for the IME, achieving 30fps real-time coding for 1080p video sequences, using 19.7k slice registers considering a Xilinx Virtex-6 device. However, the architecture considers 32×32 CTUs instead of 64×64. Moreover, the architecture was synthesized only for FPGA devices, and power results are not presented. Also, the work does not consider how the data from the IME will be obtained, abstracting the transfer between the DRAM and the architecture. In the work by Leon, Cardenas and Castillo (2016), an architecture for the FME using SAD is implemented. The architecture was synthesized only for FPGA devices, and is able to achieve 4K@30fps real-time processing considering a frequency of 258MHz, for Altera Cyclone IV E. Power results are not presented in this work.

Complete ME modules can also be found. Pastuszak and Trochimiuk (2016) propose an IME+FME algorithm and its respective architecture. It processes real-time 2160p@30fps videos, and results for 90nm technology present 400MHz frequency. The work implements the TZS algorithm in the IME stage, but it only considers 8×8 blocks - the bigger squared blocks are considered by reusing results from the 8×8 search, which makes it lose quality when compared to the default implementation. The implementation only considers rectangular PUs in the FME. Even though the work considers the memory

usage of the modules, it uses a dedicated memory module that contains the whole search window for the search to be done efficiently, implying that more silicon area is needed.

Separate SAD units were also proposed and implemented, aside from the ME module. A generic SAD architecture focused on low-power is proposed by Silveira et al. (2016), and an analysis of the use of the architecture with the Hexagon Search is also provided by Silveira et al. (2017). However, both works do not provide any analysis and information about how to solve the memory latency issue, and do not consider the interface between their proposed architecture and the gathering of block data. Nalluri, Alves and Navarro (2014) propose SAD architectures for the ME module. The architectures compute every possible PU size defined in the HEVC standard, including the AMPs. Three models were proposed: sequential, 1-stage parallel and 2-stage parallel architecture. The architectures were only synthesized for FPGA, and obtained power results of 91.3, 136.18 and 320.86 mW, respectively.

Sinangil et al. (2012) and Sinangil et al. (2013) present some considerations on memory projects for the ME, but they do not consider any power or energy estimative and do not mention any possibility of using cache memories. Jou, Chang and Chang (2015) propose an architecture for the ME in HEVC, including a memory analysis, but no power results are given, and there is no exploration of different memory modules configurations.

Even though many works propose and implement hardware architectures for the IME and SAD, most of them tend to disregard the power dissipation and therefore do not present any power or energy results, especially regarding the absolute differences operator of the SAD architectures. Also, a representative portion of the works found in the literature only present results for FPGA devices, which are not as power-efficient as ASICs. Moreover, to the best of our knowledge, none of the related works included an extensive analysis on using cache memories, varying parameters known in the study of such memories, and including power/energy results.

4 METHODOLOGY

In order to define the best configurations for both the block-matching algorithm and the cache model to be used in this work, we made several decisions based on the parameters that this work is most interested in optimizing. In this study, this is the reduction of the power/energy of the full architecture, while attempting at the same time to achieve real-time encoding for acceptable frame-rates. The following subsections detail how the decisions regarding every part of our design were taken.

4.1 Choice of Search Algorithm

The hexagon search (HS) is used as a case study in this work, due to the fact that this is one of the most optimized algorithms in terms of throughput. We performed some simulations in order to analyze the number of block access requests (PU requests in a reference frame) made by some of the IME algorithms in the literature. Each PU request may represent a variable number of bytes but, due to the similar sizes of PUs considered in the encoders, the number of bytes in each request is also similar, on average. We found that, for 1 second of five Full HD video sequences, the TZS makes an average of 178.72×10^7 access requests, while the HS makes about 5.92×10^7 of such access requests. The simulations were performed for the default presets of HM (for the TZS) and x265 (for the HS). More results for different video sequences are shown in Table 4.1.

This analysis shows that HS is about $30.19\times$ less costly in terms of access requests than TZS. This difference is due to the fact that the HM software is not focused on achieving real-time encoding performance, as it just serves as a reference document for the HEVC standard. On the other hand, the x265 software, from where the HS comes from,

Table 4.1: Access requests for different search algorithms.

	# Block (PU) Access Requests (1s)	
Video	TZS (10^7)	HS (10^7)
BBDrive	306.7	8.27
BQTerr	165.7	7.44
Cactus	207.7	6.93
Kimono	144.5	4.14
ParkScene	69	2.81
Average	178.72	5.92

achieves faster encoding, as it applies more sophisticated search heuristics and it supports parallel execution in multi-threaded multi-core architectures, which prevail nowadays in general-purpose computing platforms.

Due to the fact that the x265 is a widely used encoder and, given that this work aims at achieving a real-time frame-rate which would be difficult to achieve with HM, we decided to use the HS from x265 as a case study.

4.2 Input Data Extraction

In order to obtain results for the cache analysis, we needed to extract the access requests (vectors) of the HS from the x265 software, to use them as input for the cache simulator (which will be explained later in this section). To do that, we inserted a new C++ class in the x265 encoder, responsible for communicating with the inter-prediction modules and writing the vectors to files. We created the files according to the model presented in Table 4.2. The values presented are from the BQTerrace (1080p) video, for illustration purposes only, as they can vary depending on the video and on the frame considered.

In this model, the current frame whose vector was extracted from is presented in the first column, the next two columns contain the (x, y) vectors, and the last two columns contain the current PU size (height and width) that was being analyzed at that moment, i.e., the PU size that the (x, y) vector was pointing at. For this analysis, we considered that the video was running with only one reference frame for each inter-prediction execution. This optimization is due to the fact that a (x, y) vector in a particular frame from two different lines in the extracted files could be referring to a different reference frame. When using the cache simulator for these inputs, the reference frame for that data would have to be stored in the tag as well; otherwise, wrong hits would be counted. Thus, we decided

Table 4.2: Vector model used as input.

	Vector		PU Size	
Frame #	x	y	Height	Width
10	0	2	32	32
10	2	2	32	32
10	3	0	32	32
⋮	⋮	⋮	⋮	⋮
10	1910	1074	8	8

that one reference frame would be used.

Table 4.3 shows the coding efficiency loss (BD-Rate) when comparing a default preset x265 execution, with 3 reference frames, to a preset using only one reference frame, for different videos. For these results, we used the Bjontegaard model (BJONTEGAARD, 2001) to calculate the percentages. This model is widely used in the video coding community to calculate quality variances, measuring the bit-rate increase for the same video quality (using Peak Signal-to-Noise Ratio - PSNR as a quantitative metrics for video quality). For this metric, higher percentages mean higher losses.

In the input extraction, we decided to gather data from the execution of one frame of each video. We developed a Python script in order to automatically run all the x265 simulations for five Full HD video sequences. We could not obtain data from frame 0, due to the fact that there is no inter-prediction occurring in that frame, given that there are no previously coded frames to get temporal redundancies from. Because of that and due to any difference of initializations from the first frames, we decided that data from the tenth frame of every video would be gathered. Moreover, since the simulations were time-consuming and the number of possible configurations was high, we estimated bandwidths and hit-rates for one second of the video based on the data gathered from one frame.

It is important to mention that, given that the SMPs and AMPs (defined in the Background section) are disabled by default in the x265 reference software, none of the vectors from any of the files contain these PU models; thus, this analysis only uses the square size of the PUs. Lastly, vectors with negative values were removed, which happens due to padding (extension of the image in the left and upper side of the frame) performed by x265, given that not every encoder performs padding techniques, and we wanted to keep the analysis as generic as possible. Table 4.4 shows the amount of access requests for the tenth frame of each video, for the tested video sequences.

Table 4.3: Coding efficiency loss of using 1 reference frame.

	Coding Efficiency Loss (BD-Rate)
Video	1 ref.
BBDrive	1.49%
BQTerr	18.35%
Cactus	4.7%
Kimono	0%
ParkScene	1.84%
Average	5.28%

Table 4.4: Amount of access requests for the tenth frame.

Video	# Access Requests (10^3)
BBDrive	391.2
Cactus	251.8
Kimono	621.6
BQTerr	269.2
ParkScene	254.3

4.3 Cache Memory Analysis

After obtaining the input vectors, we designed the cache model. The generic model is shown below. The main aspect to be noticed here is in the formation of the tag and the index. Since the only information regarding the PU block is the (x, y) vector coordinates, we decided that the bits of these vectors would be used to form both the tags and indices. In the case of a Full HD (1920×1080) video, the number of bits required for each of the coordinates is 11, totaling 22 bits of information for each vector. This value is obtained from the maximum x and y values that a vector can point at (11 bits for addressing the values 1920 and 1080). These bits are split into the tag and index for the cache memory. We decided that we would use the least significant bits (LSBs) of the x and y coordinates to form the index, and the remaining bits would be used to form the tag.

The formula for obtaining the index is shown below in 4.4, along with other auxiliary equations in 4.1 - 4.3, for a generic cache. In these equations, the associativity is denoted by $assoc$, and it equals 1 for the DM and equals the total number of lines when considering an FA cache; $nrlines$ denotes the number of lines in the cache; $nBits_X$ and $nBits_Y$ correspond to the number of bits from each x and y coordinate used to form the index, such that the sum of these values equals $\log_2 nrlines$ (4.1). The $\{\}$ operator used in the equations corresponds to the concatenation operation.

$$nBits_X + nBits_Y = \log_2 nrlines \tag{4.1}$$

$$splitVec = \{X[nBits_X - 1 : 0], Y[nBits_Y - 1 : 0]\} \tag{4.2}$$

$$cutVec = \log_2 \frac{nrlines}{assoc} \tag{4.3}$$

$$index = splitVec[cutVec - 1 : 0] \tag{4.4}$$

The tag corresponds to the remaining bits, in any given order. The only needed condition is that the same order is used for the rest of the considerations, so that the comparison between tags is accurate. The formula used is given by 4.5, in which max_B is the maximum number of bits and it equals 11.

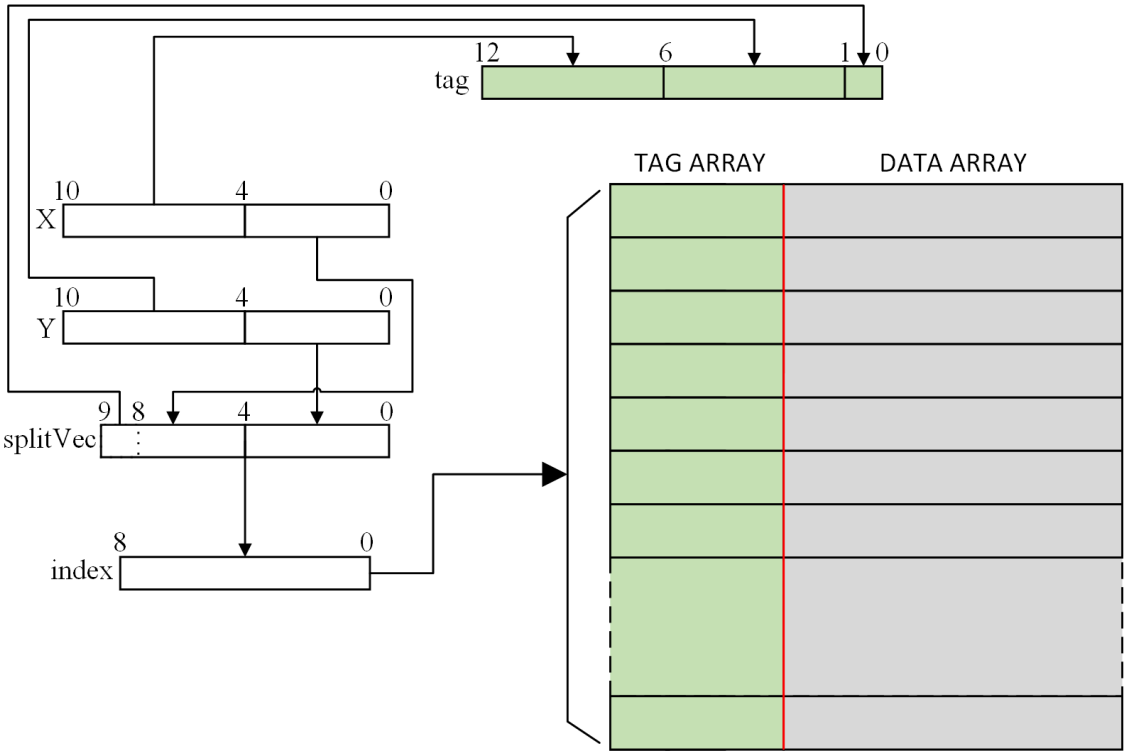
$$tag = \{X[max_B - 1 : nBits_X], Y[max_B - 1 : nBits_Y], splitVec[(nBits_X + nBits_Y - 1) : cutVec]\} \tag{4.5}$$

As a case of example, using a 2-way associative cache ($assoc = 2$) of 1024 lines ($nrlines = 1024$), for a Full HD video, considering vector $(355, 270)$, the binaries would be equivalent to $(00101100011, 00100001110)$. Considering an equal distribution of bits x and y to form the index, $nBits_X = 5$ and $nBits_Y = 5$. Using 4.2, $splitVec = 0001101110$, since this is the concatenation of a certain number of bits (5 for each, in this case) from each of the x and y coordinates. Considering only what we did so far, we would already have the index for the DM cache associated to that vector. However, when taking the associativity into account, 4.3 must be applied to determine the number of bits of $splitVec$ that will be removed. Using 4.3, $cutVec = 9$, leading to $index = 001101110$, since the most significant bit (MSB) was removed. For the tag, the remaining bits are used in the order determined by 4.5. Therefore, $tag = 0010110010000$ in this case. An example applying the equations is also presented in Fig. 4.1, for a 2-way associative cache of 1024 lines, as well.

After obtaining the access requests files and designing how the tag/index formation would work, the next step was the implementation of the cache memory simulator. We decided that a custom cache simulator would be implemented, due to the fact that having full control of the development and being familiar with the code was desired, in order to add more features in the future, e.g., adding support for more replacement policies, etc. Moreover, the values obtained in the simulator do not depend on high accuracy calculations - viabilizing the implementation -, since the number of hits and misses depends only on basic cache parameters and on the data input, which was obtained earlier. Therefore, implementing our own cache simulator would not be time-consuming.

In order to apply this cache analysis to the IME, the best input width, in bytes, for the SAD module is for it to be a multiple of 8 (considering only the reference blocks), since that is divisible by the width and height of every considered PU. This is also ideal due to the fact that the AMPs, which are the only PUs whose size parameters are not multiples of 8, are being disregarded. Moreover, in order to use all of the available hardware

Figure 4.1: Index and tag formation based on the proposed model.



Source: The Author

described in the next steps, we decided that the input width of the SAD architecture would also not be larger than the word size of the cache.

Initially, we only considered word sizes of 8 bytes for the data array of the cache, using 8 bytes for the input width of the IME as well. However, in order to make the cache even more generic, larger word sizes were considered. In these cases, when the size of the input width is not the same as the word size, we would have to align the data in order to gather exactly the requested bytes. In the cache level, the only possible word sizes to be used are when $\log_2(size - n + 1)$ results in a natural number, where n is the input width of the architecture, in bytes (reference blocks only). For the case of an input width of 8 bytes, the possible sizes include 8, 9, 11, 15, etc. This happens because the remaining bits (starting from 8) have to be a power of two, so that we can address the correct vector by taking the LSBs of the index. When vector (x, y), for example, is stored in a line of the cache, and the word size is 9 bytes, we would have bytes (x, y) to $(x + 8, y)$ in that same line. In this case, we would be able to gather two possible 8-byte data from this line: from (x, y) to $(x + 7, y)$ and from $(x + 1, y)$ to $(x + 8, y)$, and the LSB of the index would be responsible for deciding which one of the possible data we would retrieve. In the circuit level, the LSB would be used to align the data using a scheme similar to a barrel shifter, through a sequence of multiplexers to decide whether the 0-7 bytes or the 1-8 bytes would

be retrieved.

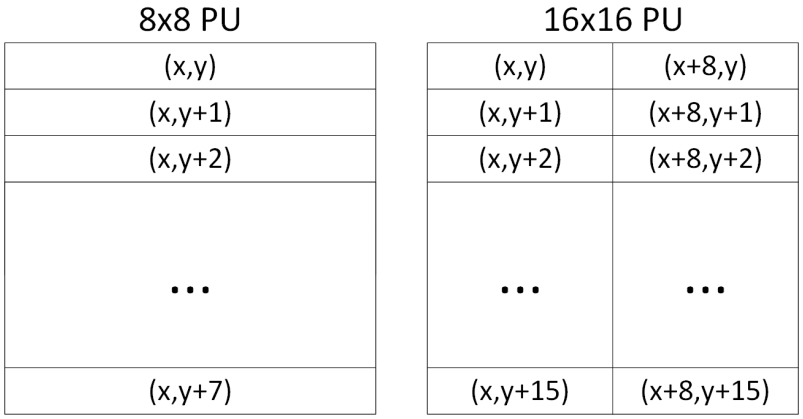
When reading data from the cache in this case, the LSBs of the index would be filled with zeroes, and the values of the bits before the fill would be used in the aligner. For example, for a 9-byte word size cache, if the x coordinate of the (x, y) vector is an odd number, we would request the value $(x - 1, y)$ when reading. However, the multiplexers used in the barrel shifter would receive a $'1'$ signal in the selectors (since the LSB is $'1'$ for binary odd numbers), shifting the data so that the architecture receives the correct sequence of bytes.

The developed cache scripts work by receiving an input file, in the same model as shown in Table 4.2 from the previous subsection. The following parameters can be changed in the simulator: word size, number of lines, associativity, replacement policy, the order of the bits from (x, y) coordinates to form the index and the number of bits extracted from each of the x and y values in order to form the full address ($nBits_X$ and $nBits_Y$ from 4.1).

For this work, we consider that the data in the main memory is organized in such a way that each of the PUs have to request each of its lines separately. For example, when considering an 8×8 PU pointed by vector (x, y), we would have to request 8 bytes eight times, since the next lines of the same PU would be in other lines of the main memory. For the mentioned case, we would have to request data for (x, y), $(x, y+1)$, ..., $(x, y+7)$, each of these requests with 8 bytes. Moreover, if the width of the PU is larger than the word size considered, we need to perform more than one request for each of the PUs lines. Considering a 16×16 PU, for a word size of 8 bytes, we would have to request (x, y), $(x+8, y)$, $(x, y+1)$, $(x+8, y+1)$, ..., $(x+8, y+15)$, each of these with 8 bytes as well. This is shown in Fig. 4.2. Due to that, the number of requests to the memory is considerably higher than the number of lines in each of the input files, since the size of the PUs corresponding to those vectors are, at least, 8×8.

Cache analysis for three different algorithms from the x265 software were performed, in order to decide whether the HS is good enough in terms of cache access. The HS and two other algorithms supported by the x265 software were analyzed: the Diamond Search (DS) and the Star Search (SS), both of which are used in other presets of the software. This was only done for checking purposes - i.e., to check if the HS has reasonable hit-rate values - given that DS runs in a x265 preset that decreases the coding efficiency significantly (and we had already made decisions that decreased the quality, so we did not want to decrease it with any other optimizations), and SS runs in a slower x265

Figure 4.2: Requests for 8×8 and 16×16 PUs considering a word size of 8 bytes.



Source: The Author

preset, so it would be harder to achieve real-time. Input data were gathered from these algorithms using the Python script previously described, and the cache simulator was run for these algorithms. The hit-rate values were similar for the search algorithms tested. For that reason, and due also to the fact that the HS is our main case study, we only present the results for the HS in the results section.

After obtaining cache hit/miss-rates, on-chip and off-chip bandwidth for every configuration (using the approximations performed by applying the values found for one frame and extending it to the desired frame-rate, in order to estimate the values for one second of the video), we decided to estimate the leakage/dynamic power and energy for each of the cache models. To do that, we used Cacti-P (LI et al., 2011), which is a tool that analyzes cache configurations, generating power/energy values based on the cache parameters. The parameters include associativity, block size, cache size, and some others. A better analysis on which parameters were used for this simulations is presented in the results section.

We also estimated the energy read of an LPDDR2 memory in order to perform comparisons between our model using the cache memory and a model without the use of any cache mechanisms. The energy values considering the off-chip DRAM and the cache memory are shown in the results section.

4.4 Absolute Differences Operator and SAD Architecture

After the analysis of the cache memory, the next step was to develop an SAD architecture that would receive the data from the memory hierarchy (either from the cache

or the main memory) and calculate the SAD, with the values from reference and current blocks. SAD performs an absolute differences operation before applying the sum of these values. For that reason, we focused on the hardware model of that specific absolute differences operator, focusing on obtaining low-power. We decided to perform this analysis due to the fact that none of the works in the literature made a research on finding the best absolute differences operator in terms of power in the context of SAD (most works use the abs operator from the synthesis tool).

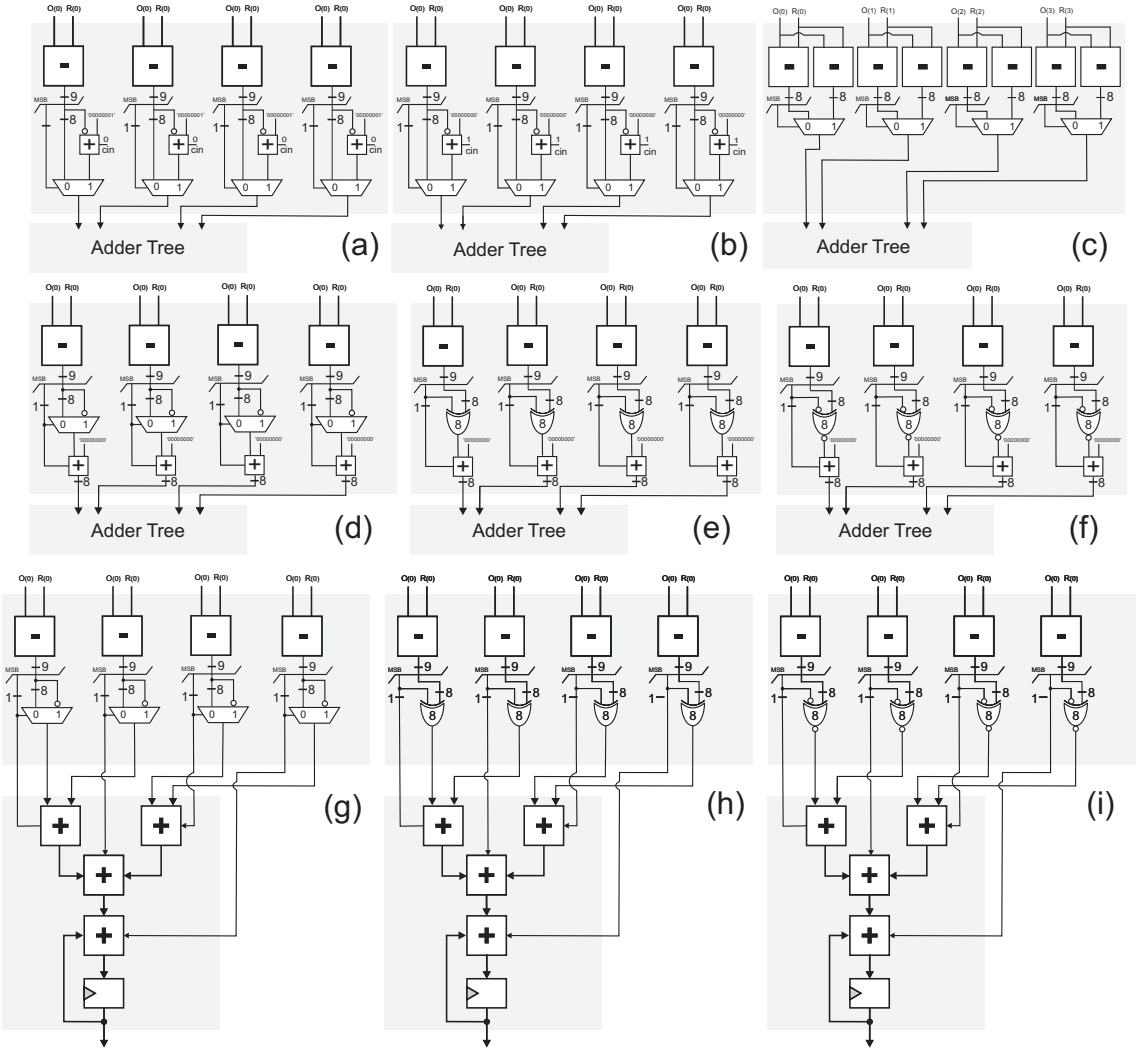
Ten different absolute differences architectures are considered in this work in order to analyze the best configuration for a power-efficient module. The architectures are shown in Fig. 4.3. The architectures (a-f) consist of independent absolute difference calculation, i.e. do not require the adder tree to work, which means they can be used in a wider variety of applications. The remaining proposals (g-i) work by propagating the MSB of the subtraction operation to the rest of the adder tree. These signals will be connected to the carry-in inputs of each of the tree adders, including the accumulation adder. It is worth mentioning that the (g-i) architectures cannot be coupled with every SAD unit, since they require the tree to be composed by an adder implementation with enough carry-in inputs for the operation to work properly. In other words, the total number of carry-ins must be equal to the number of extra bits sent to the adder tree.

Architectures (a) and (b) differ from each other in the decision of using the '$+1$' bit in the normal input of an adder or in the carry-in of the same adder. Architecture (c) uses two subtractors to calculate $O_{i,j} - R_{i,j}$ and $R_{i,j} - O_{i,j}$ in parallel, using the MSB of one of them to decide the selected result. The pairs (e-f) and (h-i) vary by implementing an Exclusive-Or (XOR) or an Exclusive-Nor (XNOR) with a NOT gate, because the synthesis may achieve better results by optimizing the XNOR gate, compensating the additional use of an inverter. Both pairs differ because (h-i) use the MSB as the carry-in of the adder tree, without needing an additional adder. On the other hand, (d) and (g) differ from each other for the same reason, but using a multiplexer instead of XOR/XNOR gates.

The best version was chosen based on the power results. The results for the absolute differences operator will be presented in the next section.

We used the results of the best absolute differences operator in order to choose which of the models we would use when implementing the rest of the SAD architecture. The SAD architecture was modeled just like presented in the background section: as a tree of adders. The adder tree has a number of levels equal to $\log_2 input_{BW}$. In the last

Figure 4.3: Absolute differences operation architectures.



Source: The Author

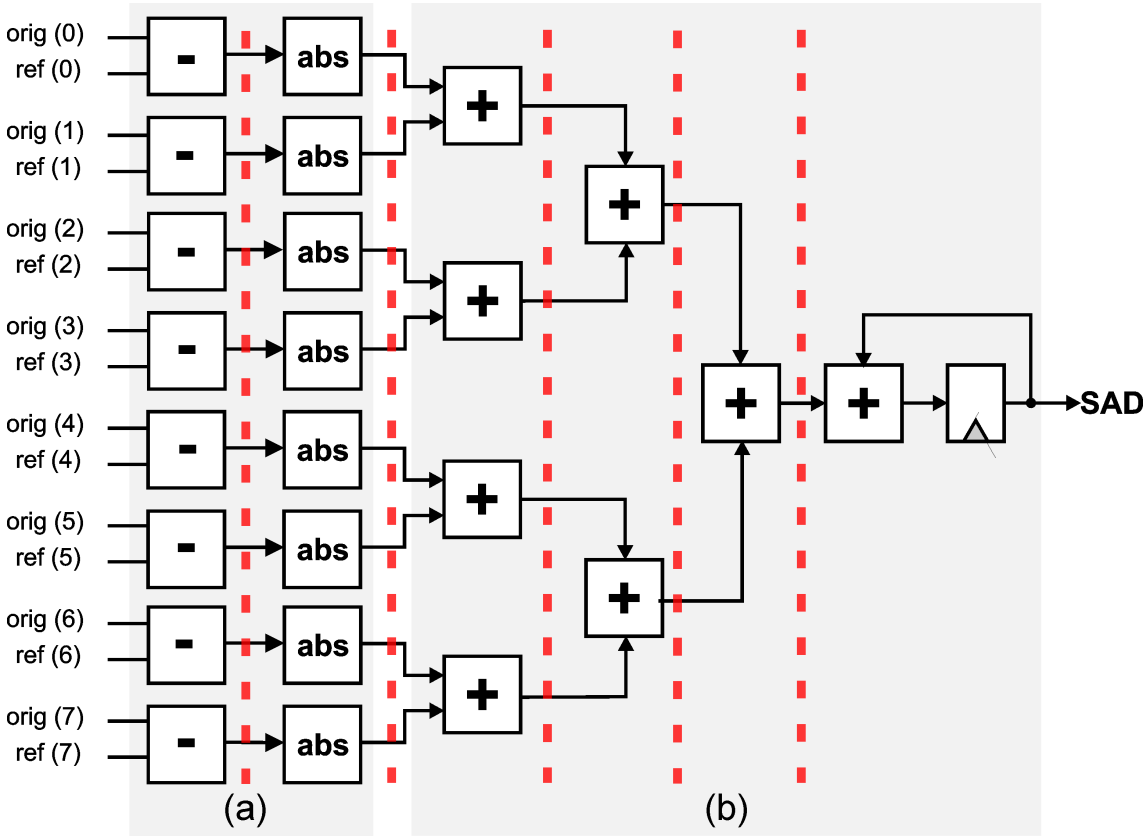
level of the tree, an accumulator is needed, to store the partial values of SAD, in case the SAD input width is not large enough for its adder tree to calculate the full SAD in only one requisition.

In the worst case scenario, we would need to calculate SAD for the largest possible PU, which is 64×64. For an accumulator not to be needed in this case, the input width would have to be of 8192B (4096B from each current and reference blocks). This size, however, would require an absurd amount of adders in the adder tree. For that reason, we decided to use a smaller input width size, and, in order to save hardware area and power, we decided to use an amount that allows it to properly work with the smallest cache line without having unused hardware (8 bytes). In other words, the SAD architecture has an input size of 16 bytes, 8 of them from the reference blocks (by using the cache model proposed) and 8 for the current block to be encoded.

The full SAD architecture is presented in Fig. 4.4. The architecture was developed with and without pipelines between each level. Near the output, the accumulator has three possible operations: adding its current value to the new value coming from the adder tree, in the case that the PU is still being calculated in blocks bigger than the input width; setting the value of the accumulator to the value coming from the adder tree, in case that is the first accumulation of a given PU; and maintaining the same value as before, in case no new values have arrived in a given cycle. The last case would happen when a cache miss occurs and the architecture needs to wait a certain number of cycles for the data to arrive.

The barrel shifter was also designed and implemented, to serve as an interface between the cache and the SAD input, for when the cache size is larger than the input width. This circuit was designed as a sequence of parallel multiplexers, and the choice of how many bytes it shifts depends on the last bits of the vector requested. For example, for a 9-byte word size cache, if vector (x, y) - with x having an LSB of $'1'$ - is requested (whose size is 8 bytes), and the cache contains the vector $(x - 1, y)$, we would need to

Figure 4.4: SAD Architecture; (a) Absolute Difference and (b) Adder Tree with accumulation; and the possible pipeline levels in red.



Source: The Author

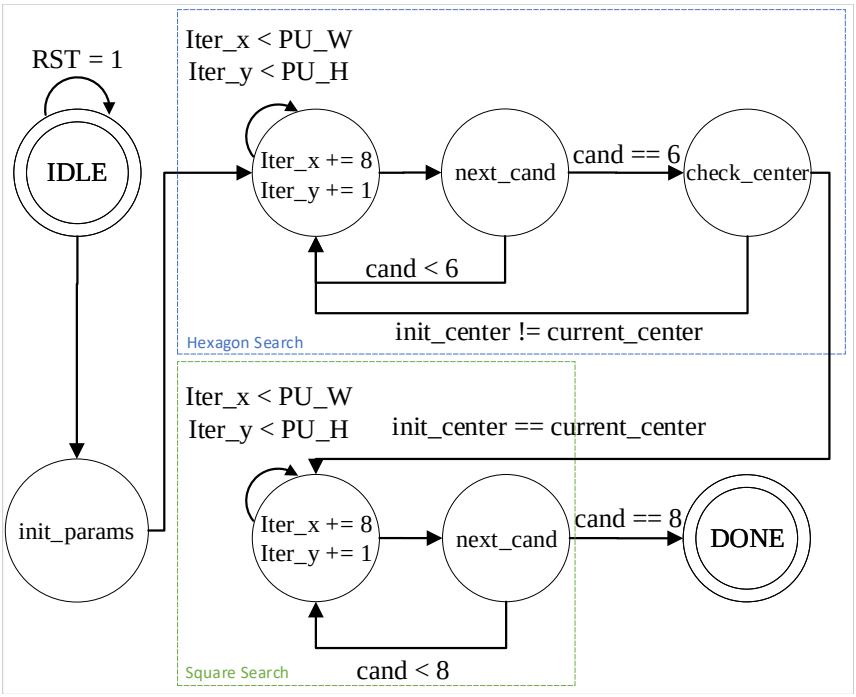
shift one byte of the cache line, so that the output of the barrel shifter contains exactly the data we want. Therefore, we use the LSBs of the x coordinate of the requested vector as a selector for the multiplexers in order to decide the number of shifts.

4.5 Hexagon Search Control Unit

After the design of the barrel shifter, we designed the Control Unit that would request the vectors and control the accumulations in the SAD, set the best SAD registers, current center registers, etc, based on the HS algorithm. The Control Unit consists of an FSM, described in VHDL. The operation basically begins by setting the center of the search, received as input, to the center registers (x and y), since the decision of the center is part of a previous stage out of the scope of this module. Also, the operation receives data regarding the size of the current PU for which the HS will be applied, in order to determine the number of accumulations needed to be done in each of the SAD calculations. The vector blocks are requested by the FSM based on the current center of the search. This center is updated when one iteration of the Hexagon Search is performed, and it looks for the other candidates around the new center, until none of the candidates are better than the current one. The last iteration performs the square search, just like described in the Background section. Fig. 4.5 shows a brief version of the FSM implemented for this work, focusing on the access requests (vectors) generation for the memory. Many of the auxiliary signals were omitted, for simplicity.

In Fig. 4.5, the $init\_params$ state sets the registers mentioned before with their initial values; the $Iter\_x$ and $Iter\_y$ variables calculate the current position inside the PU to be requested, so that the requested vector coordinates are given by the sum between the respective $Iter$ value, the current center and the displacement of the current candidate. The logic of the incrementation of the current vector (which is a function dependent on $Iter\_x$ and $Iter\_y$) to be requested was simplified for illustration purposes, since we have to check if the end of the width (for $Iter\_x$) has finished before incrementing $Iter\_y$. Therefore, both variables are not added in the same cycle. The next state changes the candidate and checks whether the number of checked candidates is equal to six, in which case it checks if the current center has changed from the initial center, in the next state. If this is the case, the algorithm performs the eight-point search around the current center in the square search, ending the algorithm. Otherwise, the six-point step is performed once again.

Figure 4.5: Brief version of the HS Control Unit.



Lastly, we estimated the throughput required for the IME architecture to work in real-time. We obtained the number of 8-byte requests from the x265, for the same 5 Full HD videos of the previous analysis, for 30 frames of each of the videos (equivalent to 1s of a 30fps video). In our analysis, the number of 8-byte requests of the SAD operation in the HS is equivalent to the frequency required for the architecture to work in real-time for that fps. Table 4.5 shows the requests for the 5 videos, as well as the average case, which corresponds to our frequency goal.

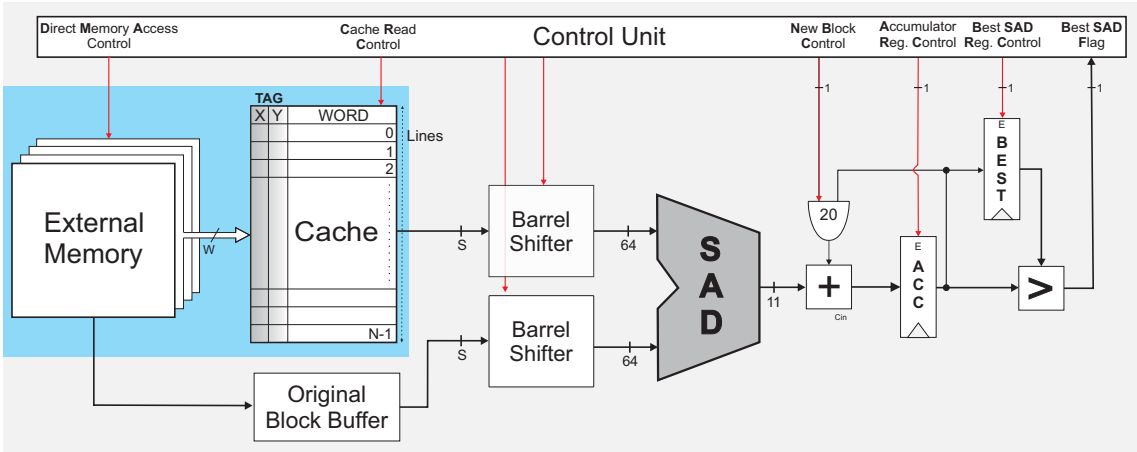
With the cache memory analysis, the absolute differences operator power evaluation, the SAD architectural decisions and the FSM designed, a complete system was developed. This system is presented in Fig. 4.6.

The system consists of the memory part (external + cache), which sends data for

Table 4.5: Frequency estimation (for 1080p@30fps) for different video sequences.

Video	Estimated Frequency (MHz)
BBDrive	561.61
BQTerr	402.926
Cactus	441.188
Kimono	641.613
ParkScene	388.072
Average	487

Figure 4.6: Top level IME system.



Source: The Author

the barrel shifter - whose size is decided from the best results in the next section -, responsible for aligning the data of the reference block for the SAD to be calculated correctly. The output of the SAD kernel consists of an Accumulator register, to calculate the SAD for bigger blocks. A best SAD register is also defined, which updates based on signals sent by the Control Unit. Other auxiliary signals are sent by the Control Unit for the accumulations to work, and to request data from the cache/external memory.

5 RESULTS

5.1 Cache Memory

Several parameters were varied in the cache simulations: five Full HD videos (BasketballDrive, BQTerrace, Cactus, Kimono and Park Scene), four associativities (DM, 2-way, 4-way and 8-way), three word sizes (8, 9 and 11 bytes), five number of lines (512, 1024, 2048, 4096 and 8192) and two replacement policies (FIFO and LRU). A random replacement policy was also implemented in the simulator, but not used due to the difficulty in applying that random logic in a future hardware model. We also varied the number of bits retrieved from the x and y coordinates, testing every possible sequence such that the sum of these values equals $\log_2 nLines$ (4.1) (the bits removed because of higher associativities were taken out after that division).

We found out that we generally obtain better hit-rate values when we equally split the weights of the x and y coordinates to form the address. Moreover, when the number of lines is an odd power of 2 - for lines 512, 2048 and 8192, which are equal to 2^9, 2^{11} and 2^{13}, respectively -, the best hit-rate results are usually in the most balanced division (when using the extra bit for the x or the y coordinates). For example, for a cache of 8192 lines, the best hit-rate values, for one given configuration of associativities, replacement policies and word size, were usually obtained when taking 7 bits from the x coordinate and 6 bits from the y coordinate to form the index.

Table 5.1 shows, for a 8192 lines cache with 8-byte word size, that the best hit-rate values were, in fact, obtained in balanced cases. For this analysis, LRU was used as the replacement policy, and it was simulated for 1 Full HD video (BasketballDrive).

The values were usually the highest when considering the preference for the x coordinate for gathering the "extra" bit of the balanced division. Therefore, due to that behavior, and due to the fact that the executions were time-consuming, we only tested that distribution when testing for number of lines which are odd powers of 2, for the rest of the simulations. For the even cases, we equally divided the bits.

Due to the high number of requests, our goal was to obtain a hit-rate of more than 90%, given that even relatively high hit-rates would lead to a high off-chip BW. Based on these results, we did not achieve that threshold for 512 and 1024 lines cache, for any of the word sizes. Therefore, they are not presented.

We also did not achieve that hit-rate threshold for caches with 8 bytes as the word

Table 5.1: Hit-rate values when varying the index formation.

	Hit-rate			
(x, y) **distrib.**	**DM**	**2-way**	**4-way**	**8-way**
(0,13)	1.15%	4.97%	14.91%	32.55%
(1,12)	2.04%	4.97%	14.91%	32.55%
(2,11)	4.57%	8.76%	14.91%	32.55%
(3,10)	5.4%	14.06%	23.73%	32.55%
(4,9)	17.19%	16.14%	36.27%	47.39%
(5,8)	45.04%	43.56%	42.42%	65.5%
(6,7)	78.8%	77.75%	76.12%	75.03%
(7,6)	85.33%	85.63%	85.23%	84.64%
(8,5)	47.06%	84.22%	85.01%	84.87%
(9,4)	18.14%	44.82%	82.39%	84.49%
(10,3)	5.61%	15.89%	42.34%	79.22%
(11,2)	1.12%	3.74%	13.98%	38.13%
(12,1)	0.15%	0.48%	3.28%	13.09%
(13,0)	0.04%	0.13%	0.45%	2.65%

size, which indicates that, for our method of forming addresses, we need to use the barrel shifter to align the data, considering our decision to use an input width of 8 bytes in the SAD architecture.

Figs. 5.1, 5.2 and 5.3 show the final hit-rate results for the configurations tested, for 8, 9 and 11 bytes of word size, respectively, for the HS algorithm. The results represent the average of the 5 Full HD videos.

Figure 5.1: Hit-Rate results for different associativities (8-byte word size).

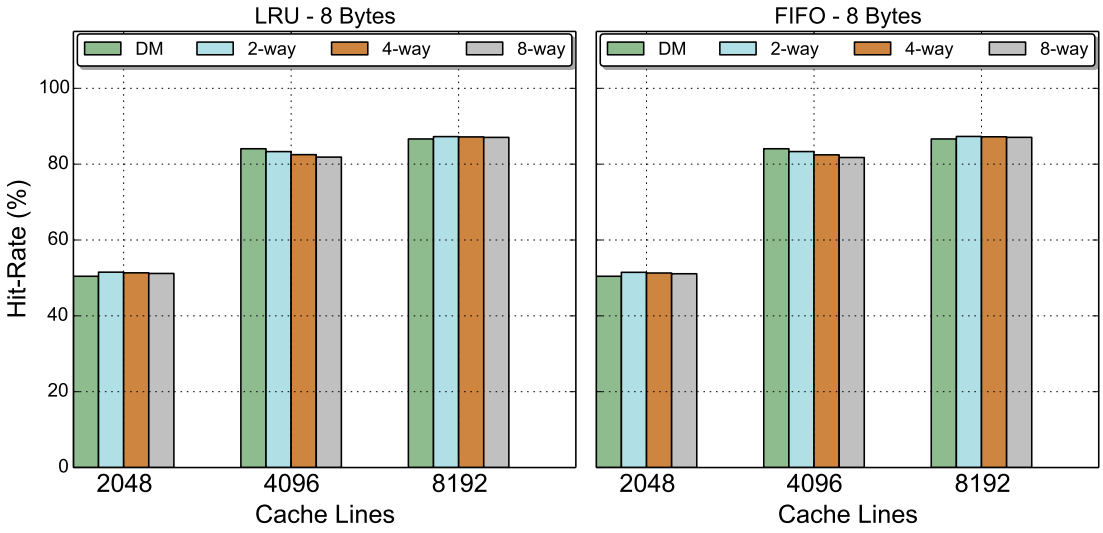


Figure 5.2: Hit-Rate results for different associativities (9-byte word size).

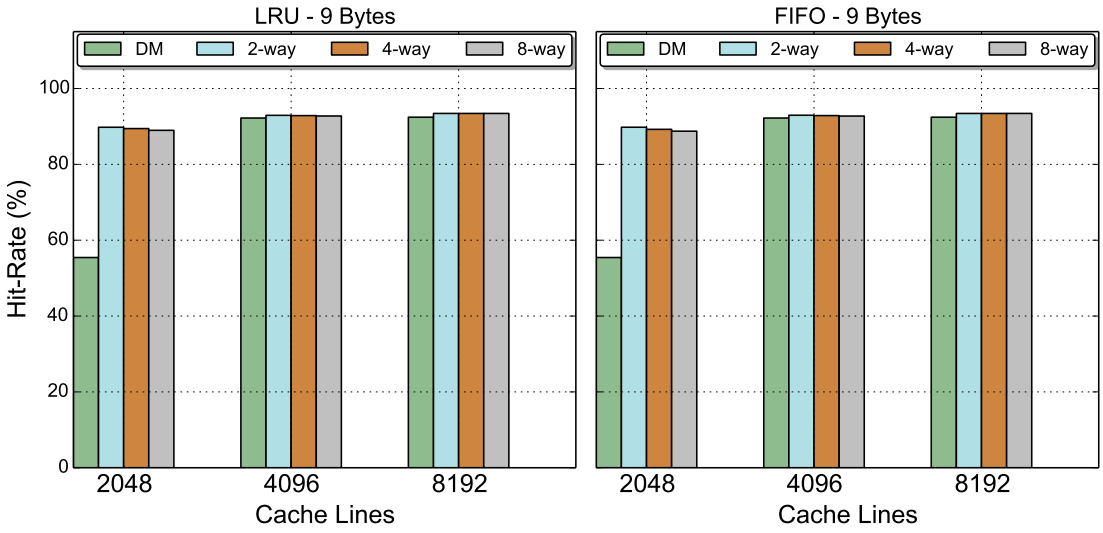
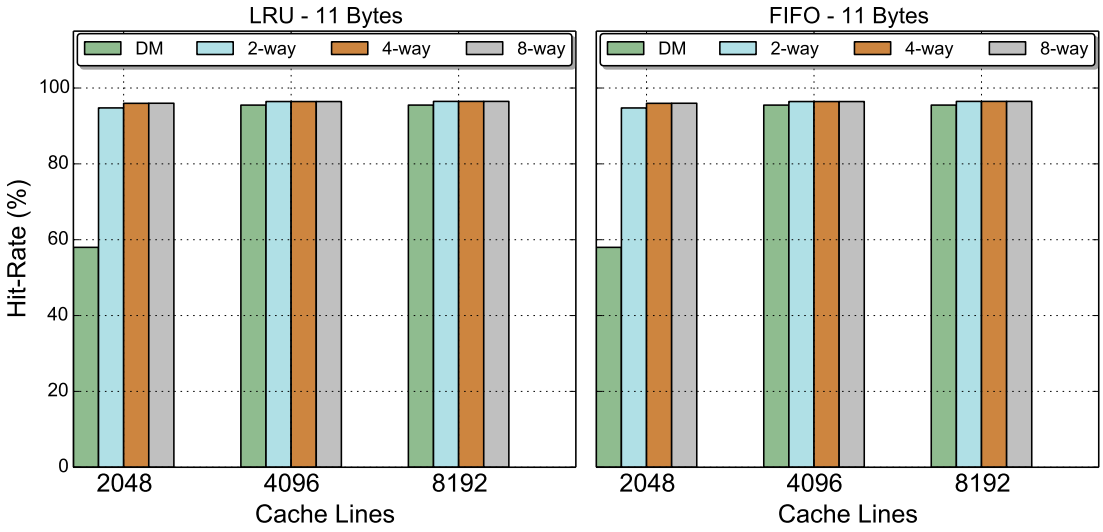


Figure 5.3: Hit-Rate results for different associativities (11-byte word size).



Some of the results presented higher hit-rate values for smaller associativity caches. The reasons for that may be challenging to notice at first, but debugging the access orders for cases that this actually happened made it clear. The reason is that a higher associativity cache may remove, due to the replacement policy, data that would not be removed when using a smaller associativity cache, since the address for the data that removed the cache would not point to the same set in the smaller associativity. Due to the HS algorithm, this pattern was quite frequent, enough for it to be noticed in slightly higher hit-rates for smaller associativity caches. For these cases, there is no trade-off between both caches,

as a smaller associativity, for the same size, will also have a smaller dissipated power, due to the use of less comparators to search in the set.

We obtained hit-rate results of up to 96.47% and an off-chip bandwidth of 0.18 GB/s, for a cache of 8192 lines, 11-byte word size, 2-way associative, for both FIFO and LRU replacement policies. Without our cache solution, the off-chip BW would be of about 5.22 GB/s, by taking the sum of the average on-chip and off-chip BW from the videos.

The variation of the replacement policy did not lead to a significant change in the results: LRU was slightly better, but in most cases the hit-rate increase was insignificant. Due to that reason, we only present the energy analysis below for the LRU, as the analysis for the FIFO is similar, for this dataset.

The cache energy results were simulated with Cacti-P, using a 65nm technology. We simulated data considering one exclusive read port - due to the fact that this is a read-only cache -, with one bank, and an operating temperature of 360K. Moreover, the design was set to equally focus on weight delay, dynamic power, leakage power, cycle time and area. The parameters for cell and peripheral types for the data and tag arrays were all set to types focused on low-power.

In order to estimate the energy from reading in the off-chip memory, we used a calculator obtained from Micron, for LPDDR2 memories (Micron, 2013). We chose a 4Gb LPDDR2-S4 to be used as a case study in this work. This off-chip analysis was made only to estimate how much energy is saved when using our cache memory proposal and when using no cache at all (only the off-chip); therefore, the off-chip values, as stated by the Micron calculator, may not be exact. We considered the memory frequency to be set as 533 MHz (obtained as the inverse of the CK cycle rate of 1.875 ns for a Speed Grade of -18, in the Micron calculator). We also considered that 32 cycles are needed to perform one read, with 2 bursts of 8 bytes each (since 16 bytes are the minimum multiple of 8 bytes large enough to gather the 11 bytes for the cache, which is the maximum width we were analyzing). The energy for a single read in the off-chip memory is determined by 5.1.

$$Energy_{Read} = Power_{Read} \times nrCycles \times time_{CK} \tag{5.1}$$

According to Micron reports (Micron, 2013) (Micron, 2005), we found out that the Power of a read for the LPDDR2 used is 53.8 mW. Applying 5.1, considering $nrCycles = 32$ and $time_{CK} = 1.875ns$, we obtain an energy value of 3.23 nJ for each of the reads.

In order to calculate the energy of the memory system without cache mechanisms - considering a miss-rate of 100% in the cache -, we consider that every access is requested to the off-chip. Therefore, we take the product between the total number of requests and the energy per read (E_{DRAM}) of the LPDDR2 analyzed. This formula is shown in 5.2.

$$TotalEnergy = nRequests_{Total} \times E_{DRAM} \tag{5.2}$$

When using our cache solution, we calculate the total energy based on 5.3, which is a function of the read energy cost of the DRAM (obtained from Micron), read and write energy cost of the respective cache configuration (obtained from Cacti-P), leakage energy (product between the leakage power - also obtained from Cacti-P - and the time considered), and the number of requests from both the cache and the DRAM, which were obtained from the cache simulator implemented.

$$TotalEnergy = E_{Leak} + nRequests_{DRAM} \times (E_{Cache_W} + E_{DRAM}) + nRequests_{Cache} \times E_{Cache_R} \tag{5.3}$$

The energy result without the cache was calculated as presented in 5.2. We obtained an average value of 509079840 requests, for the 5 videos. Therefore, by applying 5.2, $TotalEnergy = 509079840 \times 3.23 \times 10^{-9} J$, which equals 1.64 J. It is important to notice that the high energy value was due to the million of requests performed by the videos.

Figs. 5.4, 5.5 and 5.6 show the energy results with and without the cache module considering the average number of requests performed in 1 second of the 5 Full HD videos, for the word sizes of 8, 9 and 11 bytes, considering the LRU replacement policy. The results are presented for 2048, 4096 and 8192 cache lines.

Table 5.2 presents the energy decrease percentage of each cache configuration when compared to the model which only uses the DRAM, for the LRU replacement policy.

We chose the best configuration as the one with the best energy decrease among the configurations tested. Therefore, the best cache model using this heuristic is the 4096 2-way cache, with 11 bytes per line, which has a total energy result (cache + DRAM) of 83.22 mJ, corresponding to an energy decrease of 94.94% when compared to the energy value of 1.64 J which does not consider the use of the cache.

Figure 5.4: Energy results of the memory hierarchy (8-byte word size).

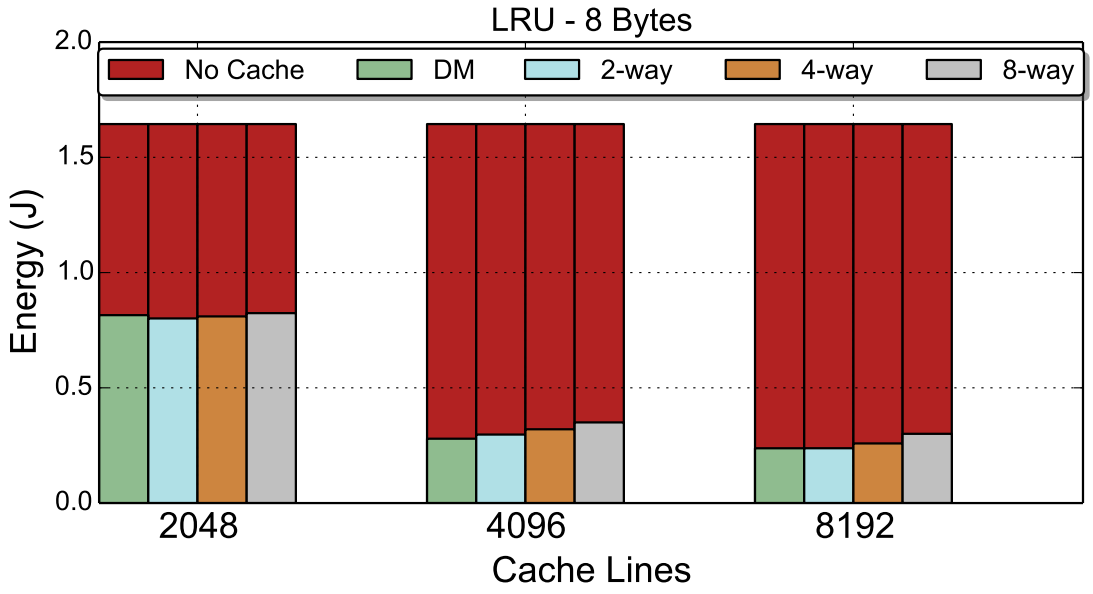
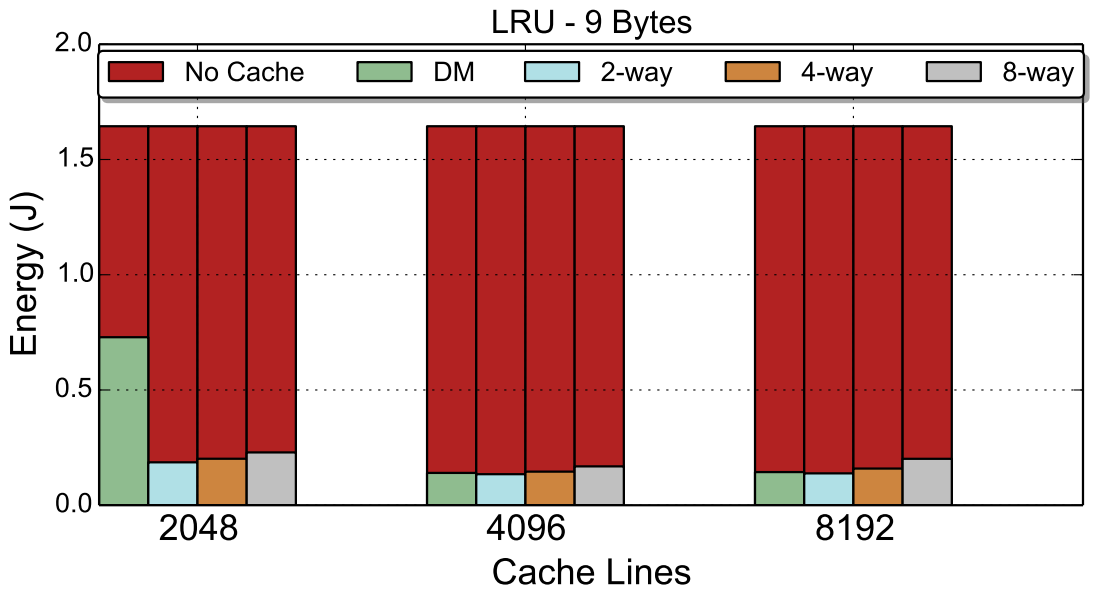


Figure 5.5: Energy results of the memory hierarchy (9-byte word size).



5.2 Absolute Differences Operator and SAD

Table 5.3 presents the power results for an 8-input SAD (8-byte original pixels and 8-byte reference pixels), using real input vectors (from BQTerrace 1080p) in the netlist level simulation, considering gate and interconnection delays. These results include leak-

Figure 5.6: Energy results of the memory hierarchy (11-byte word size).

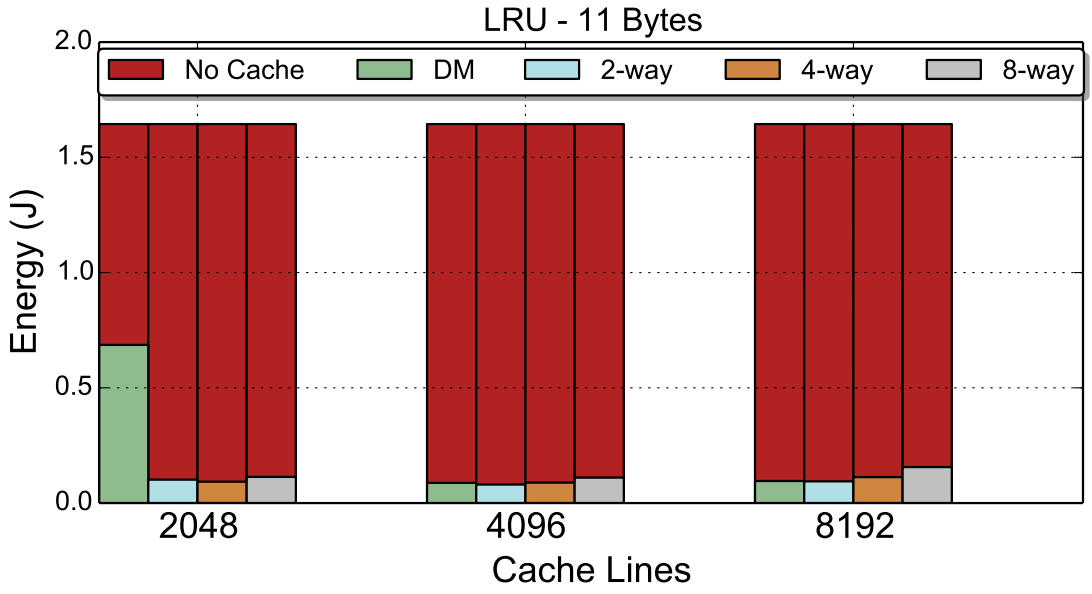


Table 5.2: Energy Reduction.

		Energy Reduction		
#Lines	**Assoc**	**8B**	**9B**	**11B**
2048	DM	50.16%	55.43%	58.01%
	2-way	50.9%	88.54%	93.71%
	4-way	50.05%	87.52%	94.22%
	8-way	48.6%	85.7%	92.9%
4096	DM	82.84%	91.32%	94.5%
	2-way	81.65%	91.62%	94.94%
	4-way	80.1%	90.88%	94.39%
	8-way	77.95%	89.35%	92.98%
8192	DM	85.25%	90.97%	93.87%
	2-way	85.19%	91.28%	93.95%
	4-way	83.81%	89.95%	92.79%
	8-way	81.05%	87.26%	90.05%

age and dynamic power, as well as glitching power percentage in the dynamic power, and total power reduction with respect to the baseline circuit (with "abs" macrofunction from Cadence synthesis tool), for full pipeline and no pipeline implementations. Table 5.4 shows the results in terms of absolute circuit area, the relative gate count (2-input NANDX1), and the total area reduction. The frequency used for the synthesis in this analysis was based on throughput estimations of Silveira et al. (2017). We estimated that we would have similar best results of absolute differences operators in the frequency of

the full IME architecture, whose results are shown in the next subsection; therefore, we decided the absolute differences operator to be used based on these results.

Table 5.3: Power Dissipation Synthesis Results @ 133MHz (μW)

	No Pipeline				
Version	**Leakage**	**Dynamic**	**Glitching**	**Total**	**Total Reduction**
Baseline	3.42	3340.2	55.7%	3343.6	-
a,b	3.11	3038.4	56.9%	3041.5	9.0%
c	2.90	2658.0	55.2%	2660.9	20.4%
d,e	4.74	5574.8	52.0%	5579.5	-66.9%
f	4.73	5575.8	54.4%	5580.5	-66.9%
g,h	3.13	3706.5	52.5%	3709.7	-10.9%
i	3.50	4030.9	52.6%	4034.4	-20.7%
	Full Pipeline				
Baseline	1.73	557.1	16.8%	558.8	-
a,b	1.74	559.2	16.9%	560.9	-0.38%
c	1.84	505.1	14.7%	507.0	9.28%
d,e	1.83	566.2	15.7%	568.0	-1.65%
f	1.86	639.2	11.2%	641.1	-14.72%
g,h	1.77	488.9	12.6%	490.6	12.20%
i	1.77	486.5	12.6%	488.3	12.62%

Table 5.4: Circuit area, Gate Count Results

	No Pipeline			Full Pipeline		
Version	(μm^2)	**kGates**	**Red.**	(μm^2)	**kGates**	**Red.**
Baseline	4143	1.99	-	3642	1.75	-
a,b	3980	1.91	4.0%	3662	1.76	-0.6%
c	3820	1.84	7.8%	3879	1.86	-6.5%
d,e	5184	2.49	-25%	3845	1.85	-5.6%
f	5145	2.47	-24%	3899	1.87	-7.1%
g,h	3865	1.86	6.7%	3673	1.77	-0.9%
i	4085	1.96	1.4%	3665	1.76	-0.6%

As shown in Table 5.3, the architectures are organized in pairs, i.e. (a,b), (d,e) and (g,h), since they have achieved the same results between each pair, due to the optimizations performed by the synthesis tool. In the pair (a,b), the difference in using the '1' logic value in the normal input or in the carry-in was ignored by the tool. In the pairs (d,e) and (g,h), the difference between using a 2-1 multiplexer and an XOR gate with a NOT gate in one of its inputs has been optimized by the tool to be implemented in the same way.

According to Tables 5.3 and 5.4, architecture (c) has achieved a power reduction of 20.4% and 9.28% for the versions without and with full pipeline, respectively, and an area reduction of 7.8% in the version without pipeline, when compared to that of the synthesis tool for the target frequency. The pair (a,b) has also achieved power and area reductions when compared to the implementation of the synthesis tool, without pipeline. Versions (g,h,i) have achieved the highest power reductions when using full pipeline, while having only a slight increase in area.

The low-power behavior of architecture (c) without pipeline stands out because the propagation in the data-path was more balanced, reducing power dissipation due to glitching in its adder tree. The results show that the pipeline reduce the glitching power since the changes in the input of each adder tree level occurs almost at the same time, reducing the dynamic power considerably. On the other hand, the full pipeline circuit architectures also reduce the drive strength of the cells (and the size of transistors), as it can observed in the circuit area and leakage power reduction even with the inclusion of registers in the circuits. With the glitching reduced in the pipeline versions, the best architectures are the (g,h,i), since they do not require the "+1" adder, as they propagate the MSB into the carry-in inputs of the adders in the tree.

5.3 IME Architecture

The architecture for IME was implemented with the 11-byte shift barrel, based on the favorable cache energy results, and without pipeline, using the best respective absolute differences operator (alternative c) from the previous section. The decision for not using pipeline was based on the fact that, just like the TZS - described in the Background section -, the HS contains several data dependencies between each of the Hexagon iterations, which would generate pipeline bubbles, since the whole SAD of the last candidate has to be entirely calculated before deciding whether the center would be changed in the next iteration. The FSM was implemented according to the model shown in the methodology section.

The results for the full architecture of the SAD, including the FSM control, is shown below. To obtain these values, we have synthesized the developed architecture for ASIC, with ST 65 nm CMOS standard cells library. The architecture was synthesized for the maximum frequency of 555 MHz. Due to the fact that none of the architectures found in the literature implement the HS algorithm, no fair comparisons could be performed.

The power and area results are presented in Table 5.5.

Table 5.5: Results for the IME architecture @ 555 MHz

Circuit Area		**Power** (μW)		
(μm^2)	**kGates**	**Leakage**	**Dynamic**	**Total**
20367.9	11.25	15.936	6950.626	6966.562

This power analysis was also performed applying real input vectors extracted from the SAD operation in the x265 software, similar to the data used in the absolute differences operator analysis, to estimate the dynamic power more accurately. Considering the previous cache and off-chip analysis, we obtained a best energy configuration of 80.31 mJ, for one second of access requests. In the IME architecture, for one second of operation, we obtained an energy value of 6966.562 μJ, which is equivalent to about 6.97 mJ. Therefore, the total energy of the whole system for that number of access requests (considering the off-chip, the cache and the IME architecture) is 90.19 mJ, showing that the IME module represents a small percentage of the total energy spent, and that the most part of energy is consumed in memory accesses, which include off-chip memory communication. The system without the cache would require for the same video duration an energy of 1.65 J. Therefore, our solution decreases the total energy of the IME by 94.5%, namely by a factor of 18.3$\times$.

Based on the design methodology used in this work, considering the maximum clock frequency of 555 MHz achieved by this architecture, and considering that a throughput analysis estimated 487 MHz as the target frequency needed, our architecture is able to achieve real-time throughput for 1080p videos at 30fps.

6 CONCLUSIONS

This work presented several analysis of components related to one of the most time and energy-consuming tasks in the HEVC video coding process: the Integer Motion Estimation. In this work, a cache analysis interfacing the IME and the off-chip DRAM was performed, due to the fact that this is one of the communication bottlenecks of the video encoders. Input vectors were extracted from the x265 encoder software, in order to determine the access order for a selection of search algorithms. The motivations given in this work and the analysis presented have resulted in the decision of using the Hexagon Search as a case study for this work.

A Python cache simulator script was implemented, in order to obtain hit-rate and off-chip BW demand values for different cache configurations. In this cache analysis and exploration, this work explored five different number of lines, four associativities, three word size, and two replacement algorithms for the caches considered. The execution was simulated for 5 full HD video sequences. We obtained hit-rate results of up to 96.47%, and a minimum off-chip BW of 0.18 GB/s, for a particular cache configuration, which represents an improvement when compared to the off-chip required BW of 5.22 GB/s, without our dedicated cache solution.

Power analysis for the same configurations were performed, obtaining the cache read power values using Cacti-P. Also, a power comparison using cache memories and using only the off-chip DRAM was performed, obtaining power parameters from Micron power calculator, for a LPDDR2. In this comparison, we obtained an improvement of up to 94.94% when using our cache mechanism.

This work also presented an analysis for the logic design of absolute differences operators in SAD architectures, to be used later on the final IME architecture. Ten different versions of the architecture were synthesized in 65nm CMOS standard cell technology, with an input width of 8 bytes per clock cycle. The results show the relevance of the absolute differences operator in the overall SAD power dissipation, and demonstrate that a small circuit can interfere in the adder tree behavior. In the main results of these designs, without using pipeline and for the technology used in this analysis, the version with two parallel operators performing $A-B$ and $B-A$ achieved the best power and area results among the tested versions, including the baseline version from the synthesis tool. The fully pipelined versions (g,h,i) have achieved the best power results, with a power reduction of up to 12.62% when compared to the baseline version. The best design without

pipeline achieved a reduction of 20.4% in total power and a 7.8% reduction in area.

Lastly, we developed the FSM for the Hexagon Search algorithm control, in order to obtain power results for the SAD + FSM system. A power analysis was performed using 65nm standard cell technology, for the maximum frequency of 555 MHz. The power analysis was performed using real input vectors, and we obtained a total power of 6.97 mW. For 1 second of operation, including our cache analysis, the total energy of 90.19 mJ for the whole hardware system was estimated.

As possible future works, we will extend the analysis of this module, implementing versions with varying number of pipeline levels as well, to verify the power and energy behavior, and to check whether the data dependencies of the search algorithms - leading to pipeline bubbles - can be alleviated by the higher frequency we may achieve.

Moreover, for future work is left the implementation of the FME, leading to a full ME architecture, attempting to achieve real-time throughput for even higher resolution videos and higher fps. We also plan on including approximate operations in these modules, to verify if the balance between the coding efficiency decrease, the higher throughput and the lower power are viable. A more accurate off-chip analysis may also be performed, by considering other power parameters for the DRAM, considering other genres of off-chip memory standards, like DDR4, wide-I/O, and others.

REFERENCES

ABREU, B. et al. Exploiting Absolute Arithmetic for Power-Efficient Sum of Absolute Differences. **IEEE International Conference on Electronics, Circuits and Systems (ICECS)**, Batumi, Georgia, 2017.

BJONTEGAARD, G. Calculation of average PSNR differences between RD-Curves. In: **Proceedings of the ITU-T Video Coding Experts Group (VCEG) Thirteenth Meeting**. Austin, Texas, USA: [s.n.], 2001.

Cássio Rodrigo Cristani. **Investigação da Estimação de Movimento para o Novo Codificador de Vídeo HEVC em Vídeos de Ultra Alta Definição**. Trabalho de Bacharelado — UFPEL, Pelotas/RS, 2014.

Cisco. **Cisco Visual Networking Index: Forecast and Methodology, 2016-2021**. 2017. Available from Internet: <https://www.cisco.com/>.

GProf. **GProf Profiler**. 1997. Available from Internet: <https://ftp.gnu.org/old-gnu/Manuals/gprof-2.9.1/html_mono/gprof.html>.

GRELLERT, M.; BAMPI, S.; ZATT, B. Complexity-scalable HEVC encoding. In: **2016 Picture Coding Symposium (PCS)**. [S.l.: s.n.], 2016. p. 1–5.

ITU-R. Parameter values for ultra-high definition television systems for production and international programme exchange. **Recommendation ITU-R BT.2020**, Oct 2015.

ITU-T and ISO/IEC. High Efficiency Video Coding. **ITU-T Recommendation H.265 and ISO/IEC 23008-2**, 2013.

ITU-T and ISO/IEC JCT. Advanced Video Coding for Generic Audiovisual Services. **ITU-T Recommendation H.264 and ISO/IEC 14496-10 (MPEG-4 AVC)**, 2011.

JCT-VC. **HEVC Test Model (HM) v. 16.7**. 2016. Available from Internet: <https://hevc.hhi.fraunhofer.de/>.

JOU, S. Y.; CHANG, S. J.; CHANG, T. S. Fast Motion Estimation Algorithm and Design for Real Time QFHD High Efficiency Video Coding. **IEEE Transactions on Circuits and Systems for Video Technology**, v. 25, n. 9, p. 1533–1544, Sept 2015. ISSN 1051-8215.

LEON, J. S.; CARDENAS, C. S.; CASTILLO, E. C. V. A highly parallel 4K real-time HEVC fractional motion estimation architecture for FPGA implementation. In: **2016 IEEE International Conference on Electronics, Circuits and Systems (ICECS)**. [S.l.: s.n.], 2016. p. 708–711.

LI, S. et al. CACTI-P: Architecture-level modeling for SRAM-based structures with advanced leakage reduction techniques. In: **2011 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)**. [S.l.: s.n.], 2011. p. 694–701. ISSN 1092-3152.

Micron. **Mobile DRAM Power-Saving Features and Power Calculations**. 2005. Available from Internet: <https://www.micron.com/support/tools-and-utilities/power-calc>.

Micron. **TN4201 - LPDDR2 System Power Calculator**. 2013. Available from Internet: <https://www.micron.com/support/tools-and-utilities/power-calc>.

NALLURI, P.; ALVES, L. N.; NAVARRO, A. High speed SAD architectures for variable block size motion estimation in HEVC video coding. In: **IEEE International Conference on Image Processing (ICIP)**. [S.l.: s.n.], 2014. p. 1233–1237. ISSN 1522-4880.

PASTUSZAK, G.; TROCHIMIUK, M. Algorithm and Architecture Design of the Motion Estimation for the H.265/HEVC 4K-UHD Encoder. **Journal of Real-Time Image Processing**, v. 12, n. 2, p. 517–529, 2016.

PATTERSON, D. A.; HENNESSY, J. L. **Computer Organization and Design: The Hardware/Software Interface**. 3rd. ed. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2007. ISBN 0123706068, 9780123706065.

PORTO, M. et al. A real time and power efficient hdtv motion estimation architecture using adder-compressors. In: **2011 IEEE Second Latin American Symposium on Circuits and Systems (LASCAS)**. [S.l.: s.n.], 2011. p. 1–4.

Roger Endrigo Carvalho Porto. **Desenvolvimento Arquitetural para Estimação de Movimento de Blocos de Tamanhos Variáveis Segundo o Padrão H.264/AVC de Compressão de Vídeo Digital**. Dissertation (Dissertação de Mestrado) — PPGC/UFRGS, Porto Alegre/RS, 2008.

SANCHEZ, G. et al. Hardware-friendly HEVC Motion Estimation: New Algorithms and Efficient VLSI Designs Targeting High Definition Videos. **Analog Integrated Circuits and Signal Processing**, v. 82, n. 1, p. 135–146, 2015.

SILVEIRA, B. et al. Power-Efficient Sum of Absolute Differences Hardware Architecture Using Adder Compressors for Integer Motion Estimation Design. **IEEE Transactions on Circuits and Systems I: Regular Papers**, PP, n. 99, p. 1–12, 2017. ISSN 1549-8328.

SILVEIRA, B. et al. Power-efficient sum of absolute differences architecture using adder compressors. In: **2016 IEEE International Conference on Electronics, Circuits and Systems (ICECS)**. [S.l.: s.n.], 2016. p. 340–343.

SINANGIL, M. E. et al. Memory cost vs. coding efficiency trade-offs for HEVC motion estimation engine. In: **IEEE International Conference on Image Processing (ICIP)**. [S.l.: s.n.], 2012. p. 1533–1536. ISSN 1522-4880.

SINANGIL, M. E. et al. Cost and Coding Efficient Motion Estimation Design Considerations for High Efficiency Video Coding (HEVC) Standard. **IEEE Journal of Selected Topics in Signal Processing**, v. 7, n. 6, p. 1017–1028, Dec 2013. ISSN 1932-4553.

SULLIVAN, G. J. et al. Overview of the High Efficiency Video Coding (HEVC) Standard. **IEEE Transactions Circuits Systems Video Technol.**, v. 22, n. 12, p. 1649–1668, Dec 2012. ISSN 1051-8215.

x265. **HEVC x265 Encoder**. 2016. Available from Internet: <https://bitbucket.org/multicoreware/x265/>.

YUAN, X. et al. A High Performance VLSI Architecture for Integer Motion Estimation in HEVC. In: **2013 IEEE 10th International Conference on ASIC**. [S.l.: s.n.], 2013. p. 1–4. ISSN 2162-7541.

Appendices

APPENDIX A — TG1

Power-Efficient Motion Estimation Architecture for HEVC Encoding

Brunno Alves de Abreu

[1]Instituto de Informática – Universidade Federal do Rio Grande do Sul (UFRGS)

baabreu@inf.ufrgs.br

Abstract. *Motion Estimation is one of the most critical and time-consuming tasks of the latest video coding standard HEVC, consuming more than 60% of the total encoding time on average. This research proposes the design and implementation of a power-efficient architecture for Motion Estimation, as well as a memory module to reduce the off-chip DRAM bandwidth. Different architectural versions with varying input bandwidth and throughput will be implemented and assessed in terms of power, clock frequency, and latency, aiming at finding the best trade-off between computation, memory and energy performance.*

1. Introduction

In the recent years, advances in technology have allowed for better quality applications, leading to an increasing demand for more sophisticated services. Digital video applications are one of the main categories affected by the technology advances, given that they allowed the requirements for higher resolution video services to be met. Real-time video content have also benefited from the technology improvement, allowing for personal video broadcasting in a worldwide scale, through streaming services such as Twitch or Youtube. The growing popularity of these services allied with the need for better quality content has resulted in predictions showing that videos will take a bandwidth share of 82% of the whole Internet traffic by 2021 [Cisco 2017].

Digital videos are usually not handled in their uncompressed form, because this requires a huge amount of resources to store and transmit this information. To exemplify, a 10-minute Full High Definition (Full-HD) 1920×1080 video, recorded at 30 frames per second, with each pixel being represented by 3 bytes, would require more than 104GB to be stored. In order to transmit this video as a real-time service, a bit-rate of more than 177 MB/s would be required. This becomes even worse when higher resolutions are considered, such as the increasingly popular Ultra High Definition 4k (UHD-4k) (3840×2160 pixels). The ITU-R recommendation for UHD Television (UHDTV) states that resolution should be increased in both spatial and temporal axes [ITU-R 2015], so higher frame rates of up to 120 fps will have to be supported. Table 1 shows storage and bit-rate requirements for a 10-minute sequence with common spatial and temporal resolutions. The size and bit-rate values are given by equations 1 and 2, in which W and H respectively refer to the width and height of the video, N refers to the representation of each pixel, in bytes, F denotes the frame-rate per second, and t refers to the time duration of the video sequence, in seconds.

$$Size(bytes) = W \times H \times N \times F \times t \tag{1}$$

Table 1. Raw video sizes and required transmission bit-rate for different spatial and temporal resolutions (sequence length: 10 minutes).

Resolution	FPS	Storage Requirement (GB)	Transm. Bit-rate (MB/s)
832×480	30	20.08	34.27
1920×1080	30	104.28	177.97
3840×2160	30	417.13	711.91
3840×2160	120	1668.54	2847.65
7680×4320	30	1668.54	2847.65
7680×4320	120	6674.19	11390.62

$$BitRate(Bytes/s) = Size/t \tag{2}$$

As seen in Table 1, the required transmission rates for uncompressed data is prohibitive on current communication technology, so there is a real need for compressing (or encoding) this information before transmission.

Video compression is based in the principle of finding redundant information within frames of a video and suppressing most of these redundancies in order to minimize the number of bits needed to represent the video sequence, and to make the required storage size and transmission rate more feasible.

While encoding reduces the amount of information used to represent videos, it also introduces a new problem in terms of computation. Modern video encoders perform many time-consuming operations to compress data efficiently, increasing the time and energy required for this task. The demands for higher resolutions aggravate this issue, because the amount of operations needed to encode each sequence is proportional to the input size. Real-time systems are particularly affected, because in this case data has to be encoded at a minimum frame-rate to optimize the user experience (typically more than 25 frames per second).

The increased computational effort of video encoders leads to an additional issue on battery-powered devices, such as smart-phones and camcorders. Due to limited battery resources, the encoding task needs to be executed as efficiently as possible. The use of general purpose processors (GPPs) for video applications is inefficient, because the arithmetic units of these devices are not designed to compute video-coding operations efficiently. For instance, many encoding steps involve performing the same operation on several pixels. In GPPs, this is usually translated to many lines of assembly code that must undergo the execution pipeline, wasting time and energy with unnecessary fetch and decode cycles.

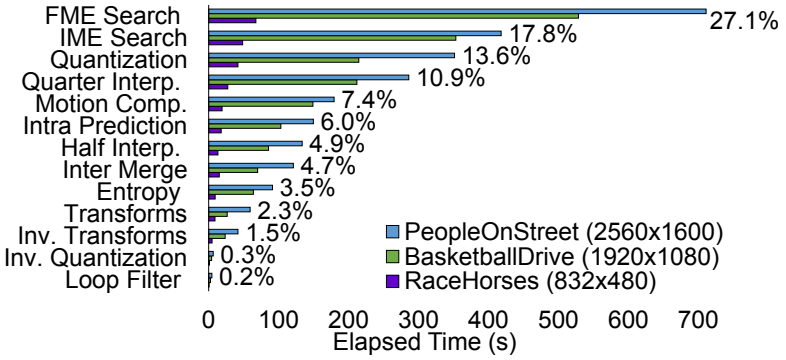
Designing dedicated hardware architectures is one of the main solutions to tackle the power and energy issues, since they eliminate the overhead of GPPs, given that they are designed solely for a chosen application. Application Specific Integrated Circuits (ASICs) are completely designed for specific domains, contrasting with GPPs. However, cost and manufacturing time become an issue when considering the use of ASICs.

Field-Programmable Gate Arrays (FPGAs) are a good balance between GPPs and ASICs. FPGAs achieve better performance when compared to GPPs for the same applica-

tion and, despite not being as dedicated as an ASIC, their time-to-market is smaller, since no layouts and other manufacturing stages are needed. However, when power-efficiency is highly taken into account, ASIC still remains as the best option.

High Efficiency Video Coding (HEVC) [ITU-T and ISO/IEC 2013] is the state-of-the-art video coding standard, and it was proposed focused on the increasing demands for higher resolution videos. When compared to its predecessor, the H.264/AVC [ITU-T and ISO/IEC JCT 2011], HEVC achieves up to 50% bit savings for the same video quality [Sullivan et al. 2012], by applying more sophisticated techniques and algorithms.

Motion Estimation (ME) is part of the HEVC standard and refers to one of the most time-consuming processes in video encoding. ME is responsible for finding most of the redundancies in videos, which is the reason why it is called several times for each frame of the sequence. Fig. 1 presents an analysis using GProf [GProf 1998] for three different video sequences, and shows that the Integer and Fractional stages of the ME (IME and FME) are responsible for most of the time in the encoding process.



SOURCE: [Grellert et al. 2016].

Figure 1. Time percentage of each stage in the video encoding process.

The proposal of this work consists in implementing a dedicated hardware architecture targeting the ME module, focusing on a power-efficient implementation and aiming to achieve a sufficient frame-rate for high resolution video sequences. Moreover, memory analysis will be performed, showing that an on-chip cache memory (SRAM) is needed in order for the ME to work when the context of the rest of the video encoder is considered. The memory studies are planned to become part of the implementation of the ME, given that most of the works in the literature tend to abstract the memory communication.

The rest of this document is presented as follows. Section 2 gives a general overview of concepts regarding video coding and points state-of-the-art proposals for modules related to the ME. Section 3 presents the proposal of this work, which will be implemented in a later stage, and the methodology that will be applied. Section IV concludes this work, encapsulating the topics discussed and highlighting the importance of the proposed work.

2. Overview

2.1. Video Coding

The video coding process is formed by a video encoder and a decoder. The process start with the capture of a real-life scene by a filming device, which generates a set of discrete scenes, i.e. frames, which correspond to the raw video. The video encoder is responsible for transforming a raw video in a sequence of bits according to a given standard, by applying compression techniques. Next, the video is sent by a transmitter, e.g. an antenna, so that it can be received by a station with a decoder which supports the standard of that bitstream. The video can also be stored for future uses. This process is simplistically shown in Fig. 2.

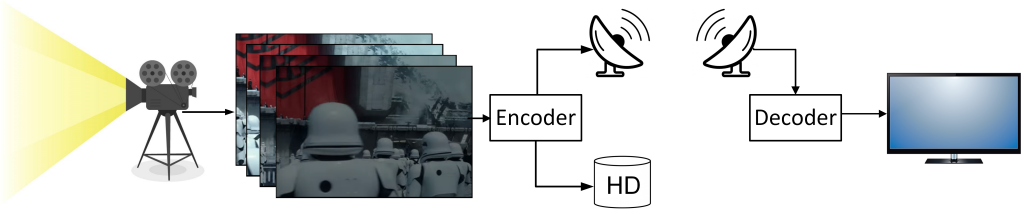


Figure 2. Simple scheme of encoding and decoding in video transmission.

The state-of-the-art video coding standard is the High Efficiency Video Coding (HEVC/H.265), which is a more sophisticated version of the previous standard H.264/AVC. HEVC doubles the compression for the same video quality when compared to H.264, and it does that by processing more data and applying more sophisticated solutions than its predecessor. However, these compression gains incurred in 1.2-3.2× increase in computational effort in the encoder side in comparison with the previous standard [Sullivan et al. 2012].

The video coding standard only defines specifications for the decoder. The encoder can be freely implemented, as long as the output bitstream conforms to the standard that can be processed by the decoder. The reference software for the HEVC standard is the HEVC Test Model (HM) [JCT-VC 2016]. x265 [x265 2016] is also a well-known encoder software compatible with the HEVC standard.

2.2. Video Encoder Diagram

The diagram of a general video encoder is shown in Fig. 3. The process starts by splitting each frame of the video sequence in blocks called Coding Tree Units (CTUs). HEVC defines 64×64 as the default CTU size. Each CTU of the frame to be encoded is applied in the stages of the presented diagram.

CTUs are split into smaller squared blocks, called Coding Units (CUs), in a quadtree partitioning scheme, based on heuristic decisions. HEVC supports CU sizes of 8×8, 16×16, 32×32 or 64×64. The intra and inter-frame are responsible for resolving spatial and temporal redundancies, searching for similar information compared to the block being encoded, so that only residual information need to be sent to the encoder output. In order to find optimal redundant blocks, CUs are split into several possible partitions, called Prediction Units (PUs), for which the prediction algorithms are applied. The PUs can be categorized in Symmetric Motion Partitions (SMP) and Asymmetric Motion Partitions

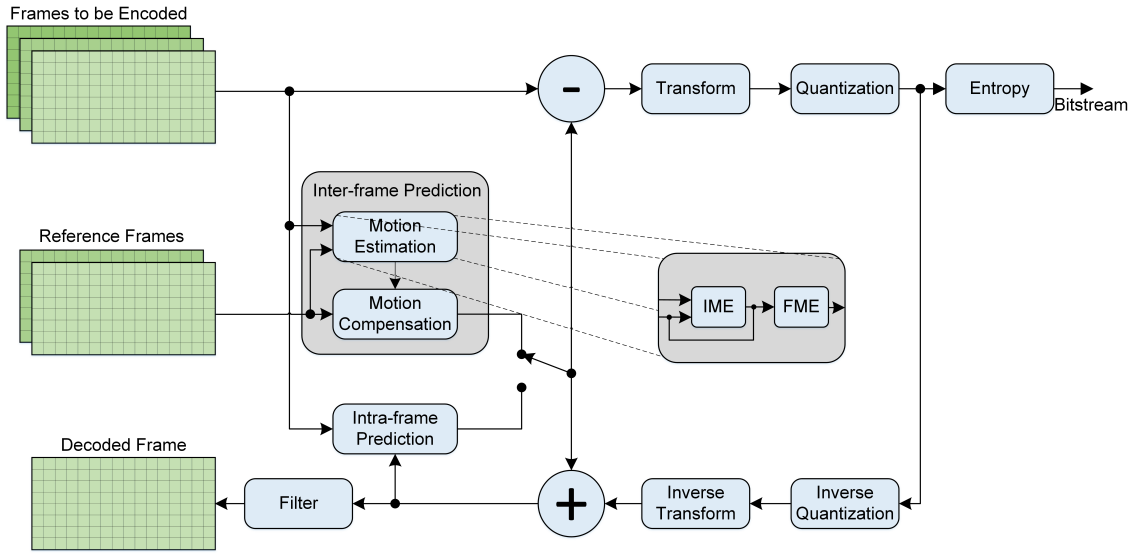


Figure 3. Video encoder diagram.

(AMP). Fig. 4 shows the split from CTU to PU level, highlighting the PUs for a 32×32 CU.

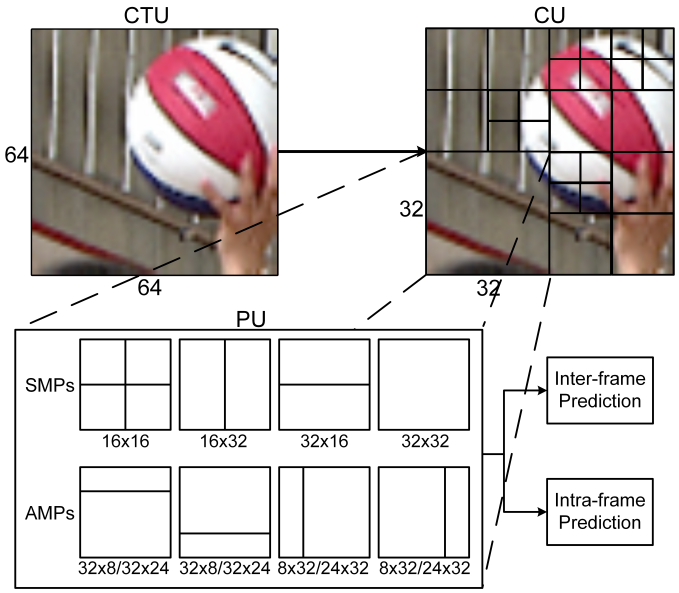


Figure 4. CTU partitioning scheme.

After generating the best candidate from either intra or inter-frame prediction, a residual block is calculated. Transform and quantization stages are applied to that block, allowing for the introduction of lossy compression. Before being sent to the bitstream output, an entropy algorithm is applied to the block, to exploit entropic redundancies.

The coded block also needs to be decoded in this encoding scheme, by applying inverse quantization and inverse transform functions so that it can be recovered. This is needed because the inter-frame prediction uses information of previously coded frames

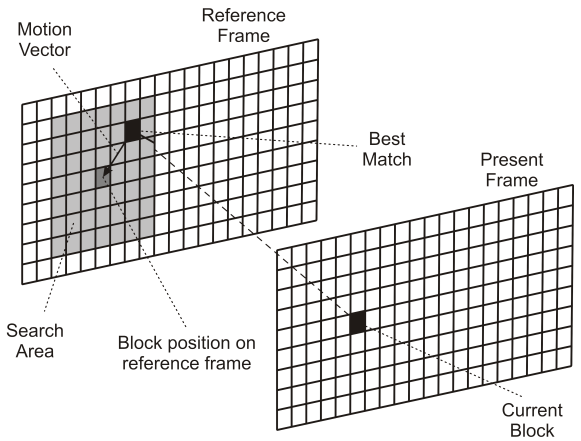
to find temporal redundancies in the current frames, so these previous frames need to be ready to be analyzed.

2.3. Motion Estimation

ME is a stage inside the inter-frame prediction in video encoding responsible for resolving temporal redundancy in a video. ME is applied for each block of the video frame and finds the most similar block compared to the one being encoded. Thus, the only information needed to be sent to the output of an encoder is the difference between both blocks (residual block) and a vector pointing to the best matching block, so that the decoder can use as much information from previously decoded frames and their blocks, to obtain the original block to be decoded. ME is split into two subsequent stages: Integer Motion Estimation (IME) and Fractional Motion Estimation (FME).

2.3.1. Integer Motion Estimation

IME finds the best matching block in an integer-pixel level, by applying a search algorithm in the blocks being encoded. This search algorithm defines a pattern of positions in which the most similar block will be searched. The search algorithm is applied in a given search window of the frame, which consists of an area smaller than the frame itself, due to the fact that image patterns tend to slightly move from the area where they were in a previous frame. Fig. 5 generically shows the concepts involved in the IME.



SOURCE: [Roger Endrigo Carvalho Porto 2008].

Figure 5. Generic search in a previously coded frame.

Some of the main search patterns and algorithms known in the literature are presented in the following subsections.

2.3.2. Full Search

The Full Search (FS) is the most basic algorithm found in the literature. FS applies the search of the most similar block in every pixel in the search window, always finding the best possible matching block, as long as it is contained in the window. For that reason, FS

maximizes the temporal redundancy found, resulting in a smaller bitstream in the encoder output.

Due to the large number of candidates being tested, FS requires the highest number of memory accesses and calculations, among all the other search algorithms. For that reason, real-time implementations apply other solutions so that the search is performed for a smaller number of candidates, while still attempting to find residual blocks very similar to the best possible.

2.3.3. Test Zone Search

Test Zone Search (TZSearch) is the algorithm used in the latest versions of the HM reference software. TZSearch applies more than one search pattern and has a more sophisticated flow, so that very similar matching blocks can be found with less calculations. The algorithm is divided in 4 subsequent stages, each of which are dependent from the previous one. For that reason, pipeline-based architectural implementations waste a good amount of unnecessary cycles, given that the pipeline needs to be totally emptied so that all the processing of a previous stage can be done before starting the next one. These stages are explained in the following subsections.

a) Vector Prediction: this stage is responsible for examining resulting vectors of the neighbor blocks of the PU being encoded, generated from their previous ME execution. Then, the block being encoded is compared to each block pointed by these vectors, and the position of the most similar one is defined as the initial center of the search.

b) First Search: a diamond-shaped search is performed, starting from the position defined by the VP. The diamond pattern tests four, eight or sixteen candidates around the center, depending on the distance between samples, which is incremented after each iteration. If the search finds a block more similar than the one in the current center in an iteration, the center is redefined as the position of that block. The search stops when the best candidate remains being the center after three iterations. Fig. 6 shows the shapes of the first three iterations.

c) Raster Search: this stage searches through the entire search area, starting from the top-left corner, and skips some neighboring candidates by defining a raster step constant. This is illustrated in Fig. 6 for a constant of 2.

d) Refinement Search: the last stage works the same way as the First Search. The main difference is that the default tolerance in the HM software for this level is two instead of three iterations.

2.3.4. Hexagon Search

This is the search algorithm implemented by the x265 software. Hexagon Search is split into three main step, and the dependencies between them also makes it important to analyze the number of pipeline levels to be used in an architecture, just like in the TZSearch. The following items detail the Hexagon Search.

a) Motion Prediction: this stage works similarly to the VP in the TZSearch algo-

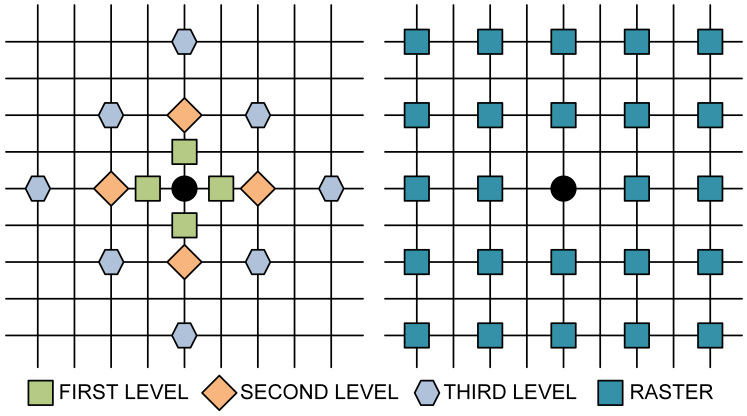


Figure 6. Search shapes for the TZSearch.

rithm, only differing in the number of candidates evaluated to decide the initial center of the search.

b) Hexagon: this step performs a six-point search in a hexagon-shaped format around the center of the search, considering the initial center is defined by the previous step. Whenever there is a candidate more similar to the block being encoded than the one in the center, the new center is defined as the vector associated to that new candidate. Then, the hexagon search is applied again, starting from the new center of the search. This stage stops when none of the candidates are better than the current center.

The x265 software applies an optimization to this stage: starting from the second iteration of the hexagon search, only three candidates need to be evaluated instead of six, since the three remaining points will always have been evaluated in the previous iteration. This is illustrated in Fig. 7.

c) Square Refinement: after the Hexagon step defines a best candidate, an 8-point square refinement is applied around that point. The final vector value is defined by the best candidate evaluated in this stage. If none of the candidates are better than the current center, then the center defined by the previous step is the best vector.

Fig. 7 illustrates the whole process of the Hexagon Search algorithm, considering an 8×8 PU, in which four iterations of the Hexagon are made. In the last one, no candidates were more similar to the block being encoded than the center, so the square refinement was applied, resulting in a better block found in one of its eight candidates.

2.3.5. Fractional Motion Estimation

Fractional Motion Estimation (FME) is applied after the IME and is responsible for finding the best match at the fractional-pixel level, starting from the most similar block found in the IME. Due to the fact that frames are formed only by integer pixels, FME requires the use of an interpolator to estimate fractional pixels positioned between the integer pixels of the image. FME is responsible for increasing quality in video sequences due to the fact that real-life patterns frequently move at a rate that is not a good match to the integer-pixel displacement between two simultaneous video captures.

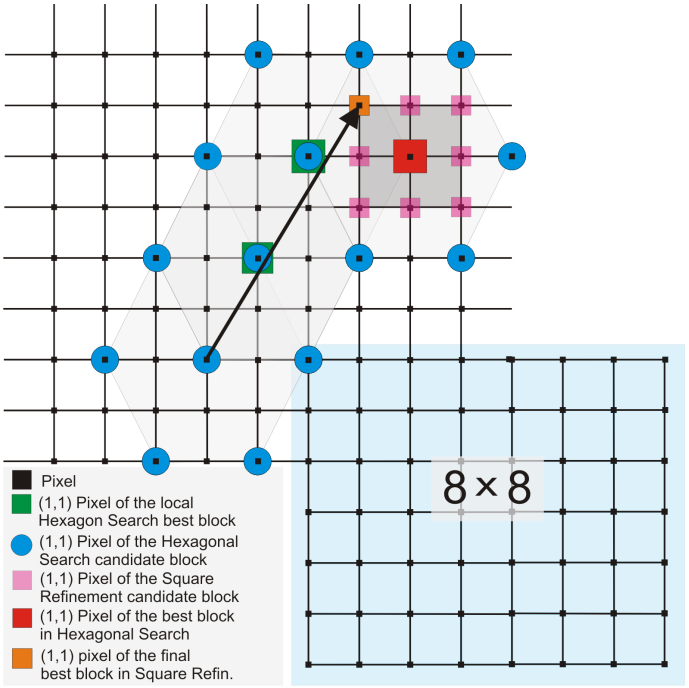


Figure 7. Search shapes for the Hexagon Search.

HEVC defines 48 possible candidates to be compared in the FME: 8 half-precision and 40 quarter-precision candidates. Fig. 8 presents the set of fractional points needed to gather all the information regarding the 48 fractional blocks. In this figure, green positions correspond to integer pixels, blue positions correspond to half-precision pixels and white positions correspond to quarter-precision pixels. Half and quarter-precision pixels are generated through interpolation using 7-taps and 8-taps FIR filters, depending on the specific point. The highlighted partition shows the 48 possible points to which comparisons will be performed.

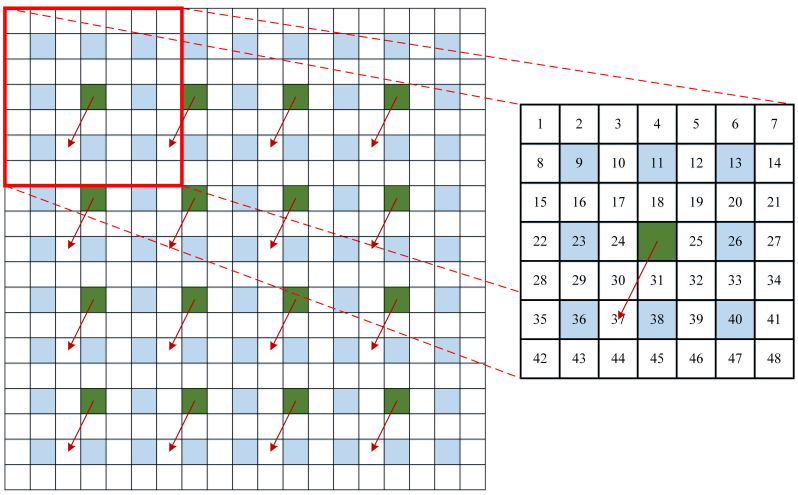


Figure 8. FME fractional pixels window for a 4×4 PU.

2.4. Metrics for Block Similarity

Several metrics can be applied to determine the degree of similarity between two blocks. They differ from each other in ease of implementation, efficiency and result accuracy, i.e. how precisely they can define the similarity between the blocks. The main metrics used in video codecs are shown in the next subsections.

2.4.1. Sum of Absolute Differences

Sum of Absolute Differences (SAD) is the simplest metric used in the video encoding process, and is applied by calculating the differences between the co-localized pixels of a current and a candidate block, performing an absolute operation in these differences and then adding the values. The complete formula is given by 3, in which O and R denote the current and the candidate blocks, respectively.

$$SAD = \sum_{i=0}^{m} \sum_{j=0}^{n} |O_{m,n} - R_{m,n}| \tag{3}$$

SAD architectures are mostly implemented using subtractors, absolute operators and an adder tree, with an accumulator on the output so that SAD for bigger blocks can also be calculated. Most architectures use pipeline schemes, so that several sum stages can be done concurrently and the resulting critical path is shortened. A 8×8 SAD architecture is shown in Fig. 9, in which the sizes are presented in bits.

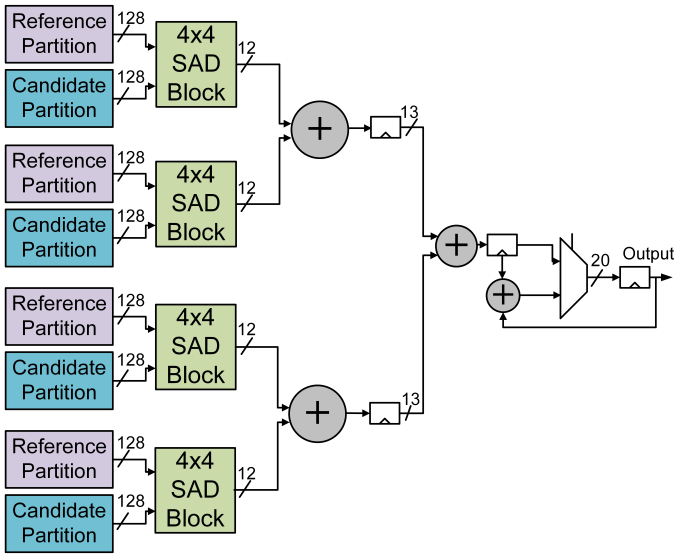


Figure 9. 8×8 SAD architecture.

The presented architecture calculates SAD for an 8×8 block in each cycle, after the pipeline is filled. The 4×4 SAD blocks correspond to simple adder trees, including the subtractors and absolute operators. The 2-1 multiplexer is used to extend the block sizes possibilities, so that receiving 4×4 blocks several times would allow the architecture to calculate SAD for larger block sizes. Considering a generic SAD architecture, the number of cycles required to calculate a N×N block is given by 4. In this formula, BW is the bandwidth of all the candidate partitions; H and W refer to the height and the width of

the PU to be calculated, respectively; and P_{depth} corresponds to the number of pipeline levels in the architecture.

$$cycles = P_{depth} + BW(bytes)/(H * W) \tag{4}$$

It should be noted that an increase in the minimum block size of the architecture implies in more hardware area, resulting in power and energy increases. However, small architectures spend more time calculating bigger blocks, since the number of accumulations needed is higher.

2.4.2. Sum of Absolute Transformed Differences

The Sum of Absolute Transformed Differences (SATD) is a more complex metric for video compression. SATD is calculated by taking the frequency transform of the differences between pixels of current and candidate blocks, and the formula is given by equation 5.

$$SATD = \sum_{i=0}^{m} \sum_{j=0}^{n} T\left(O_{m,n} - R_{m,n}\right) \tag{5}$$

In HEVC, this is the metric used by default during the FME, and Hadamard is the transform function used for that matter. Although SATD is a more accurate metric to determine the most efficient block match, hardware architectures have a more complex implementation when compared to SAD architectures.

2.5. Related Work

ME modules can be found in the literature frequently divided into the IME and the FME, due to the focus of each work in optimizing specific parts of each of these modules. [Sanchez et al. 2015] employs two algorithm and their respective hardware implementations for an IME module. The architectures are based in the use of parallel instances of a diamond-shaped search. The architectures achieve real-time throughput for 1080p sequences. Even though the architecture presented some quality loss when compared to other solutions, the proposed architecture has less data dependencies, more regular memory accesses and regular cycles to encode a single block. Results show gate count of 150k with a 42.3MHz frequency and power dissipation of 12.5 and 13.5 mW (for each presented architecture), for a 90nm cell library.

[Yuan et al. 2013] also implements an architecture for the IME, achieving 30fps real-time coding for 1080p video sequences, using 19.7k slice registers considering a Xilinx Virtex-6 device. However, the architecture considers 32×32 CTUs instead of 64×64. Moreover, the architecture was synthesized only for FPGA devices, and power results are not presented. In [Leon et al. 2016], an architecture for the FME using SAD is implemented, including the interpolation and the decision units for fractional pixels. The architecture was synthesized only for FPGA devices, and is able to achieve 4K@30fps real-time processing considering a frequency of 258MHz, for Altera Cyclone IV E. Power results are not presented in this work.

Complete ME modules can also be found. [Pastuszak and Trochimiuk 2016] proposes an IME+FME algorithm and its respective architecture. It processes real-time

2160p@30fps videos, and results for 90nm technology present 400MHz frequencies. The work implements the TZSearch algorithm in the IME stage, but it only considers 8×8 blocks - the bigger squared blocks are considered by reusing results from the 8×8 search, which makes it to loose quality when compared to the default implementation. The implementation only considers rectangular PUs in the FME. Even though the work considers the memory usage of the modules, it uses a dedicated memory module that contains the whole search window for the search to be done efficiently, implying that more silicon area is needed.

Separate SAD units have also been proposed and implemented, aside from the ME module. [Silveira et al. 2016] presented a power-efficient SAD architecture using adder compressor structures. These adder compressors are used based in the principle that SAD architecture can easily disregard partial sum values, so a normal SAD tree would be unnecessary. Therefore, the use of adder compressors for that matter is an efficient solution for architectures focused on power results. The architecture achieves an average of 60.8% of power reduction when compared to an architecture using the adder from the synthesis tool. [Nalluri et al. 2014] proposes SAD architectures for the ME module. The architectures compute every possible PU size defined in the HEVC standard, including the AMPs. Three models were proposed: sequential, 1-stage parallel and 2-stage parallel architecture. The architectures were synthesized for FPGA only, and obtained power results of 91.3, 136.18 and 320.86 mW, respectively.

Even though many works propose and implement hardware architectures for the IME, FME and SAD, most of them tend to disregard the power dissipation and therefore do not present any power or energy results. Also, a representative portion of the works found in the literature only present results for FPGA devices, which are not as power-efficient as ASICs.

3. Work Proposal

This work proposes the design and implementation of a dedicated hardware architecture for the ME stage of the video coding process, aiming to achieve power-efficient results. The work intends to include cache memory evaluations for the implementation of a specific search algorithm, and generic analysis to be used by the video coding community in other search algorithms. All the architectures will be described in VHDL, and the scripts and parsers needed to make simulators and analyze data will be mostly made with Python language. C++ will also be used to gather data from the encoder softwares, due to the fact that both the HM and x265 are written in that same language.

Vector data from the search algorithm will be gathered directly from the encoder software for benchmark video sequences, so that they can be used as input for a cache memory script simulator. This script will simulate the behavior of several cache memory configurations, by varying line size, cache size, associativity, number of banks, etc, and hit-ratio values will be gathered for each of these configurations, based on the input data extracted from the encoder software. Next, power and energy analysis of each cache configuration will be gathered using the CACTI software [N. Muralimanohar and R. Balasubramonian and N. P. Jouppi]. The cache memory will also have to take into consideration the limitations of the off-chip memory that will be connected to it, making certain cache line sizes unfeasible.

Considerations also will be made to estimate the average number of cycles that will be wasted whenever a miss happens in the cache memory, given the difference in the frequencies of the off-chip and the on-chip memory, write latency, memory bus, difference in line sizes, etc.

After hit-ratio and power/energy values are obtained for these configurations, an SAD module will be proposed, similar to the one presented in the SAD section. The bandwidth size will be decided based on the smallest size that can achieve real-time processing for high resolution videos, and such that the cache line size is large enough to fill the whole bandwidth of the module in a single cycle.

A script that generates generic SAD modules of various sizes and several pipeline configurations in VHDL language has already been developed. The script also generates the SAD module using different absolute operators. The chosen implementation of this operator will be decided based on power/energy results, to be gathered using the Cadence RTL Compiler (RC) [Cadence].

The IME control, which defines the search algorithm, will be implemented on top of the SAD module, so it can decide the best block using that metric. Further optimizations for that algorithm will possibly be made if higher frame-rate results are required. These optimizations will also be implemented in the corresponding software (HM or x265), in order to measure the quality loss compared to the standard configurations of these softwares. Fig. 10 shows the expected top-level module of the architecture, including the memory modules for the current and the candidate partition.

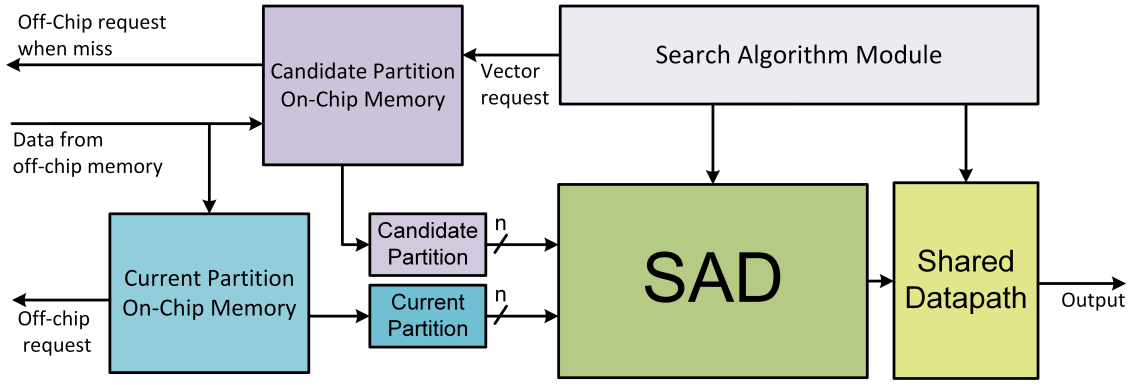


Figure 10. Top-Level module.

Initially, off-chip data will be requested for the current partition to be obtained, to which the search will be performed. Next, the search algorithm module, which defines the search pattern in the reference frame, will request a block through a vector value from the candidate partition on-chip memory. The cache module will be responsible to analyze whether the data is already in it (hit) or if it requires an off-chip request (miss) for the data to be delivered to the SAD core. The SAD module will calculate the similarity between both blocks and compare with the best previous SAD value. The shared datapath will contain registers to maintain the best SAD value, the best vectors related to the block that generated that best SAD, along with other auxiliary components. In the end of the search algorithm, the best vector value from the vector register will be assigned to the output.

Block data will be gathered from the software to be used in a testbench to deter-

mine the correctness of the architectures. The data and the architectures will also be used in RC in order to extract static and dynamic power, frequency and area results for 45 nm standard cell flux.

Table 2 shows a schedule of the activities planned for this work.

Table 2. Activities schedule

	Jul	Aug	Sep	Oct	Nov	Dec
Development of the cache memory simulator	■	■				
Extraction of best configuration results		■				
Implementation of the IME architecture		■	■			
Implementation of the cache memory interface			■	■		
Extraction of power/energy, area and throughput results				■	■	
Writing of the term paper			■	■	■	■

4. Conclusions

This document presented a general overview of concepts related to video coding, focused on the state-of-the-art HEVC standard. The main stages of a video encoder were briefly reviewed, and stages like the ME, which will be targeted in this work, have been detailed more precisely, regarding some of the search algorithms implemented by encoder softwares, and some metrics used by these algorithms.

The research plan herein proposed targets the implementation of a power-efficient architecture for the ME module considering a specific search algorithm. Due to the very intensive memory accessing required by the most important ME algorithms, cache memory analysis will be made in order to decrease the bottleneck between the ME module and the off-chip DRAM memory.

The results gathered from this research will be useful for the video coding community given that the architecture and the memory analysis will be generically implemented. For that reason, it will become much simpler to extend solutions for any other search algorithms, as our findings will also consider the memory communication requirement.

References

Cadence. Cadence EDA tools. http://www.cadence.com.

Cisco (2017). Cisco Visual Networking Index: Forecast and Methodology, 2016-2021.

GProf (1998). GProf Profiler. https://ftp.gnu.org/old-gnu/Manuals/gprof-2.9.1/html_mono/gprof.html.

Grellert, M., Bampi, S., and Zatt, B. (2016). Complexity-scalable HEVC encoding. In *2016 Picture Coding Symposium (PCS)*, pages 1–5.

ITU-R (2015). Parameter values for ultra-high definition television systems for production and international programme exchange.

ITU-T and ISO/IEC (2013). High Efficiency Video Coding. *ITU-T Recommendation H.265 and ISO/IEC 23008-2*.

ITU-T and ISO/IEC JCT (2011). Advanced Video Coding for Generic Audiovisual Services. *ITU-T Recommendation H.264 and ISO/IEC 14496-10 (MPEG-4 AVC)*.

JCT-VC (2016). HEVC Test Model (HM) v. 16.7. http://hevc.hhi.fraunhofer.de/.

Leon, J. S., Cardenas, C. S., and Castillo, E. C. V. (2016). A highly parallel 4K real-time HEVC fractional motion estimation architecture for FPGA implementation. In *2016 IEEE International Conference on Electronics, Circuits and Systems (ICECS)*, pages 708–711.

N. Muralimanohar and R. Balasubramonian and N. P. Jouppi. CACTI 6.0: A Tool to Model Large Caches.

Nalluri, P., Alves, L. N., and Navarro, A. (2014). High speed SAD architectures for variable block size motion estimation in HEVC video coding. In *2014 IEEE International Conference on Image Processing (ICIP)*, pages 1233–1237.

Pastuszak, G. and Trochimiuk, M. (2016). Algorithm and Architecture Design of the Motion Estimation for the H.265/HEVC 4K-UHD Encoder. *Journal of Real-Time Image Processing*, 12(2):517–529.

Roger Endrigo Carvalho Porto (2008). Desenvolvimento Arquitetural para Estimação de Movimento de Blocos de Tamanhos Variáveis Segundo o Padrão H.264/AVC de Compressão de Vídeo Digital. Dissertação de mestrado, PPGC/UFRGS, Porto Alegre/RS.

Sanchez, G., Zatt, B., Porto, M., and Agostini, L. (2015). Hardware-friendly HEVC Motion Estimation: New Algorithms and Efficient VLSI Designs Targeting High Definition Videos. *Analog Integrated Circuits and Signal Processing*, 82(1):135–146.

Silveira, B., Paim, G., Diniz, C. M., and da Costa, E. A. C. (2016). Power-efficient Sum of Absolute Differences Architecture using Adder Compressors. In *2016 IEEE International Conference on Electronics, Circuits and Systems (ICECS)*, pages 340–343.

Sullivan, G. J., Ohm, J. R., Han, W. J., and Wiegand, T. (2012). Overview of the High Efficiency Video Coding (HEVC) Standard. *IEEE Transactions Circuits Systems Video Technol.*, 22(12):1649–1668.

x265 (2016). HEVC x265 Encoder. https://bitbucket.org/multicoreware/x265/.

Yuan, X., Jinsong, L., Liwei, G., Zhi, Z., and Teng, R. K. F. (2013). A High Performance VLSI Architecture for Integer Motion Estimation in HEVC. In *2013 IEEE 10th International Conference on ASIC*, pages 1–4.