ADDIS ABABA UNIVERSITY

College of Technology and Built Environment

SCHOOL OF ELECTRICAL AND COMPUTER ENGINEERING

ACCELERATION OF H.266 ENCODING USING OPENCL AND VECTORIZATION WITH BLOCK SIZE VARIATION

BY

MICHAEL GIRMA

ADVISOR

Dr. FITSUM ASSAMNEW

A thesis submitted to the School of Electrical and Computer Engineering

in partial fulfillment of the requirements for the Degree of Master of Science in Computer Engineering

JUNE, 2025

ADDIS ABABA, ETHIOPIA

ADDIS ABABA UNIVERSITY
College of Technology and Built Environment
SCHOOL OF ELECTRICAL AND COMPUTER ENGINEERING

ACCELERATION OF H.266 ENCODING USING OPENCL AND VECTORIZATION WITH BLOCK SIZE VARIATION

BY

MICHAEL GIRMA

APPROVED BY BOARD OF EXAMINERS

____

Dean, SECE, AAiT (Name and Signature)

____

Chairperson, (Name and Signature)

____

Advisor, (Name and Signature)

____

Examiner, (Name and Signature)

____

Examiner, (Name and Signature)

Acknowledgement

First and foremost, I would like to express my deepest gratitude to my advisor, Dr. Fitsum Assamnew, whose guidance, insight, and unwavering support have been invaluable throughout this research. Dr. Assamnew's expertise and encouragement have been the cornerstone of this work, helping me navigate challenges and refine my ideas to achieve this thesis's objectives.

My heartfelt thanks extend to the School of Electrical and Computer Engineering at AAiT for providing an enriching academic environment and the resources necessary to pursue this research. The faculty and staff have been instrumental in fostering a culture of inquiry and innovation.

I am also profoundly grateful to my colleagues and friends, especially members of the Parallel Computing Lab, for their constant support, brainstorming sessions, and constructive feedback. Their camaraderie and shared passion for research greatly enhanced my graduate experience.

To my family and friends, I owe an immeasurable debt of gratitude for their unconditional love, encouragement, and steadfast support. My family, in particular, has shaped my character and aspirations through their sacrifices and belief in my potential. For this, I am forever grateful.

Abstract

Versatile Video Coding (H.266) achieves approximately a 50% reduction in bitrate compared to its predecessor. However, this improvement in compression efficiency comes with a significant increase in computational complexity, presenting major challenges for real-time encoding on general-purpose processors. Most existing H.266 (VVC) implementations rely heavily on CPU-only processing or on vendor specific GPU solutions such as CUDA, which limits portability and cross platform compatibility. Moreover, these approaches often fail to fully utilize modern heterogeneous CPU-GPU architectures, leaving substantial performance potential unexploited. This work proposes an OpenCL-based H.266 encoding solution aimed at delivering high performance, broad cross-platform support, and efficient hardware utilization. Key encoding modules including block partitioning, prediction, transform and quantization, loop filtering, and entropy coding—are implemented as OpenCL kernels to leverage task-level parallelism across both CPUs and GPUs. Additionally, AVX and SSE vectorization techniques are applied on the CPU side to enhance per-core throughput, particularly in compute intensive operations such as transform and quantization. Experimental results across various platforms demonstrate significant performance improvements. On an NVIDIA V100 GPU, the OpenCL-accelerated encoder achieves speedups of up to 7500× compared to a sequential implementation running on an Intel Xeon E5-2698 v4, with peak efficiency observed at a block size of 512×512. Tests conducted on an Intel UHD 620 GPU and an Intel i5-8265U CPU reveal speedups ranging from 15.5× to 370×, depending on the block size. The findings suggest that medium block sizes (64×64 to 256×256) strike the best balance between computational efficiency and workload distribution. While AVX provides only modest gains over SSE, the primary performance bottleneck lies in memory access speed rather than computational power. Overall, the proposed OpenCL-based implementation significantly accelerates H.266 encoding while maintaining high compression quality.

Keywords: H.266/VVC, OpenCL, GPU Acceleration, CPU Optimization, Video Encoding

Contents

Acknowledgement ii

Abstract iii

List of Figures vii

List of Tables ix

List of Acronyms x

1 Introduction **1**

1.1 Background 1

1.2 Statement of the Problem 3

1.3 Objectives 4

1.3.1 General Objective 4

1.3.2 Specific Objectives 4

1.4 Contribution of the Study 5

1.5 Scope and Limitations 5

2 Theoretical Background **6**

2.1 Video Encoding 6

2.2 OpenCL Framework 7

2.3 VVC Encoding 10

2.3.1 Block Partitioning 10

2.3.2 Intra Prediction 11

2.3.3 Inter Prediction 11

2.3.4 Transform and Quantization 12

2.3.5 Loop Filtering 12

2.3.6 Entropy Coding 13

2.4 Related Works 13

3 Literature Review **16**
3.1 H.266/VVC Encoding Complexity 16
3.2 Parallel Computing in Video Coding 17
3.3 OpenCL for General-Purpose Video Processing 19
3.4 VVC Encoding and Heterogeneous Computing 20
3.5 Challenges in OpenCL Acceleration for VVC 21

4 Methodology **23**
4.1 Data Preparation and Preprocessing 23
4.1.1 Data Preparation 23
4.1.2 Data Preprocessing 24
4.2 System Overview and Workflow of Proposed Solution 26
4.2.1 System Architecture 26
4.2.2 H.266 (VVC) Encoding Workflow Using OpenCL 28
4.3 H.266 (VVC) Encoding Optimization for Transform and Quantization 31
4.3.1 AVX and SSE SIMD Vectorization for Transform and Quantization 31
4.3.2 OpenCL with AVX Vectorization for Transform and Quantization 33
4.4 Evaluation Metrics 35
4.4.1 Performance Measurement Metrics 36

5 Results and Discussion **38**
5.1 Experimentation Setup 38
5.2 Existing System 39
5.2.1 Performance Analysis Using perf 39
5.2.2 Existing H.266 Encoding Without OpenCL Acceleration . . 41
5.3 Optimization of Transformation and Quantization USing AVX and SSE Vectorization 42
5.4 Performance Evaluation to OpenCL and Opencl with AVX Implementation 45
5.4.1 Encoding Time for Opencl Implementation 45
5.4.2 Encoding Time for Opencl+AVX implementation 48

5.4.3 Speedup Gain 50
5.4.3.1 Speedup Gain of i5 OpenCL Implementation over i5 Sequential 51
5.4.3.2 Speedup Gain of Xeon OpenCL Implementations over Xeon Sequential 52
5.4.3.3 Speedup Gain of i5 OpenCL Implementation over i5 Sequential+AVX 53
5.4.3.4 Speedup Gain of Xeon OpenCL Implementations over Xeon Sequential+AVX 54
5.4.3.5 Speedup Gain of i5 OpenCL+AVX Implementation over i5 Sequential 55
5.4.3.6 Speedup Gain of Xeon OpenCL+AVX Implementations over Xeon Sequential 56
5.4.3.7 Speedup Gain of i5 OpenCL+AVX Implementation over i5 Sequential+AVX 57
5.4.3.8 Speedup Gain of Xeon OpenCL+AVX Implementations over Xeon Sequential+AVX 58
5.4.3.9 Compare Speedup Gain On OpenCL Xeon and i5 over sequential Implementations 59
5.4.3.10 Compare the speedup gain of OpenCL+AVX Xeon and i5 in sequential Implementations 62
5.4.3.11 Compare the Speedup gain Of OpenCL and OpenCL+AVX Xeon and i5 in sequential and AVX Implementations 63
5.5 Discussion 66

6 Conclusion and Recommendations 69

6.1 Conclusion 69
6.2 Recommendations 71
Bibliography 72

List of Figures

2.1 OpenCL Platform Model and Execution Hierarchy. 9

2.2 VVC Encoding Block Diagram. 13

4.1 Structure of a raw YUV 4:2:0 video frame. 24

4.2 Overview of preprocessing steps applied to raw video input. 25

4.3 System architecture for OpenCL-based H.266/VVC encoder implementation. 27

4.4 OpenCL-based H.266/VVC encoding pipeline architecture. 30

4.5 SSE Vectorization for Transform and Quantization. 32

4.6 AVX Vectorization for Transform and Quantization. 32

4.7 OpenCL-AVX Integration for Transform and Quantization. 34

4.8 Flowchart for Parallel Block Size Validation in OpenCL. 35

5.1 Performance profiling of H.266 encoder using `perf`. 40

5.2 OpenCL encoding time for Intel i5-8265U CPU vs. Intel UHD Graphics 620 GPU. 46

5.3 OpenCL Implementation Encoding Time: Xeon E5-2698 v4 CPU vs NVIDIA V100 GPU 47

5.4 OpenCL+AVX Implementation Encoding Time: i5-8265U CPU vs UHD 620 GPU 49

5.5 OpenCL+AVX Implementation Encoding Time: Xeon E5-2698 v4 CPU vs NVIDIA V100 GPU 49

5.6 Speedup gain UHD 620 GPU and i5-8265U CPU over i5 Sequential 51

5.7 Speedup gain of Xeon E5-2698 v4 CPU and NVIDIA V100 GPU over Xeon Sequential CPU 53

5.8 Speedup gain of OPenCL UHD 620 GPU and i5-8265U CPU over i5 Sequential+AVX CPU 54

5.9 Speedup gain of OpenCL Xeon E5-2698 v4 CPU and NVIDIA V100 GPU over Xeon Sequential+AVX CPU 55
5.10 Speedup gain of i5 OpenCL+AVX UHD 620 GPU and i5-8265U CPU over i5 Sequential CPU 56
5.11 Speedup gain of Xeon OpenCL+AVX E5-2698 v4 CPU and NVIDIA V100 GPU over Xeon Sequential CPU 57
5.12 Speedup gain of i5 OenCL+AVX UHD 620 GPU and i5-8265U CPU over Xeon Sequential+AVX CPU 58
5.13 Speedup gain of Xeon OpenCL+AVX E5-2698 v4 CPU and NVIDIA V100 GPU over Xeon Sequential+AVX CPU 59
5.14 Speedup gain: Intel UHD 620 GPU vs. Intel i5-8265U CPU using OpenCL. 60
5.15 Speedup gain: NVIDIA V100 GPU vs. Xeon E5-2698 v4 CPU using OpenCL. 61
5.16 Compare Speedup gain i5 OpenCL+AVX i5-8265U CPU over UHD 620 GPU in i5 Sequential CPU 62
5.17 Compere Speedup gain of Xeon OpenCL+AVX E5-2698 v4 CPU over NVIDIA V100 GPU in Xeon Sequential CPU 63
5.18 Compare the Speedup gain Of OpenCL over OpenCL+AVX i5 in sequential and AVX Implementations 64
5.19 Compare the Speedup gain Of OpenCL over OpenCL+AVX: Xeon in sequential and AVX Implementations 64

List of Tables

5.1 Baseline Encoding Times for i5 CPU 41

5.2 Baseline Encoding Times for Xeon CPU 42

5.3 AVX vs SSE SIMD Vectorization Encoding Time for i5 CPU 43

5.4 AVX vs SSE SIMD Vectorization Encoding Time for Xeon CPU) . 43

List of Acronyms

ALF Adaptive Loop Filter

AVC Advanced Video Coding

AVX Advanced Vector Extensions

CABAC Context-Adaptive Binary Arithmetic Coding

CCALF Cross-Component Adaptive Loop Filter

CPU Central Processing Unit

CTU Coding Tree Unit

DBF Deblocking Filter

DPB Decoded Picture Buffer

FPGA Field-Programmable Gate Array

GPU Graphics Processing Unit

HEVC High Efficiency Video Coding

JVET Joint Video Experts Team

MIP Matrix-based Intra Prediction

MTS Multiple Transform Selection

OpenCL Open Computing Language

PE Processing Element

QTMT Quad-Tree with Multi-Type Tree

RDO Rate-Distortion Optimization

SAO Sample Adaptive Offset

SIMD Single Instruction Multiple Data

SSE Streaming SIMD Extensions

VVC Versatile Video Coding

Chapter 1
Introduction

1.1 Background

Video encoding is the backbone of modern digital media distribution, enabling the efficient storage and transmission of video content across platforms such as streaming services, video conferencing, and broadcast television. As video resolutions advance from Full HD (1080p) to 4K and 8K, the need for more efficient compression standards becomes critical. The High Efficiency Video Coding (HEVC/H.265) standard has been widely adopted for its ability to reduce bitrates by roughly 50% compared to its predecessor, H.264/AVC [1]. However, the push for higher resolutions and more immersive media experiences has led to the development of a new standard, H.266/Versatile Video Coding (VVC), which achieves about a 50% bitrate reduction over H.265 at the same visual quality [2].

Despite its superior compression efficiency, H.266 comes with significantly higher computational complexity than H.265, particularly in core encoding operations such as motion estimation, transform coding, and entropy coding [3]. For instance, H.266's flexible block partitioning and advanced prediction techniques can require up to 10× longer encoding time than H.265 for 4K video content [4]. This computational burden makes real-time encoding impractical on conventional CPU-based systems, limiting H.266's use in latency-sensitive applications like live streaming and cloud gaming.

To address this challenge, heterogeneous computing leveraging both CPUs and GPUs—has emerged as a promising solution. OpenCL (Open Computing Language) provides a unified framework for parallel programming across diverse hardware architectures, enabling developers to tap into the combined processing power

of CPUs, GPUs, and other accelerators [5]. While prior research has demonstrated significant speedups using GPUs for H.264/AVC and H.265/HEVC encoding [6], OpenCL-based optimization of H.266 is still largely unexplored. This study aims to bridge that gap by investigating OpenCL acceleration techniques for H.266 encoding, focusing on improving encoding speed without sacrificing compression efficiency.

Key Challenges in H.266 Encoding:

- **Increased Algorithmic Complexity:** New coding tools in H.266 (such as affine motion estimation and multi-type tree block partitioning) impose much higher computational demands than those in previous standards [18].
- **Memory Bottlenecks:** Data dependencies can lead to inefficient (non-coalesced) memory access patterns, especially in components like motion compensation and transform coding [10].
- **Lack of CPU and GPU Optimization:** Most existing H.266 implementations rely on CPU-based processing, missing opportunities for parallel acceleration on CPUs and GPUs [6].

This research explores OpenCL-based optimizations to overcome these challenges and some other optimization with the goal of enabling real-time H.266 encoding for high-resolution video.

Single Instruction, Multiple Data (SIMD) vectorization is a technique that enhances computational performance by allowing a single instruction to process multiple data elements simultaneously. Instead of performing operations one element at a time (as in scalar execution), SIMD enables parallel execution across data vectors, significantly reducing the number of instructions and improving throughput. This method is particularly effective in applications like video encoding, where similar operations (matrix multiplications or filtering) are repeatedly applied to large blocks of pixel data.

Modern processors support SIMD through instruction sets like Intel's SSE (Streaming SIMD Extensions) and AVX (Advanced Vector Extensions). These allow processors to execute operations on 128-bit or 256-bit registers, handling multiple

pixels or coefficients in a single CPU cycle. In the context of H.266/VVC encoding, SIMD vectorization is especially beneficial during the transform and quantization stages, which involve intensive mathematical operations. By leveraging SIMD, these steps can be completed much faster, contributing to overall encoding speedup and efficiency improvements[78].

1.2 Statement of the Problem

While H.266 offers substantial improvements in compression efficiency, its high computational complexity makes real-time encoding difficult, especially for 4K and 8K video. Existing implementations are predominantly CPU-based and operate sequentially, which makes it difficult to meet the performance requirements of latency-sensitive applications. Moreover, very limited research has been conducted on using OpenCL to accelerate H.266, leaving many potential optimizations unexplored.

Key Research Gaps:

- **Limited OpenCL Optimization:** Most prior work focuses on CUDA or specialized hardware rather than OpenCL, which offers broader hardware compatibility [3].
- **Inefficient Memory Utilization:** Poor memory coalescing in GPU and CPU-based H.266 encoders leads to suboptimal performance [9].
- **Partial Pipeline Optimization:** Many studies accelerate only specific encoder modules (like motion estimation) rather than the entire encoding pipeline [9].

Proposed Solution: To address these gaps, we design an OpenCL-accelerated H.266 encoder with the following features:

- Implement the entire VVC encoding pipeline using OpenCL and Optimize memory access (coalescing) to reduce data transfer overhead.
- Apply SIMD and OpenCL+SIMD vectorization to compute-intensive kernels for transformation and quantization.

- Evaluate performance at 2K, 4K, and 8K video resolutions.

Research Questions:

RQ1: How does block size affect H.266 encoding time on CPUs and GPUs in OpenCL-based implementations?

RQ2: What performance improvements can be achieved in H.266 encoding through AVX, AVX combined with OpenCL, and block size variation as optimization techniques?

1.3 Objectives

1.3.1 General Objective

OpenCL-based acceleration of H.266 encoding across heterogeneous CPU–GPU architectures with preserved coding efficiency.

1.3.2 Specific Objectives

- Identify the most computationally intensive kernels in the H.266 encoding process and then use SSE and AVX optimization techniques to reduce encoding time.
- Evaluate the effect of block size variation on encoding speed.
- Implement the entire VVC encoding pipeline using OpenCL and Optimize memory access (coalescing) to reduce data transfer overhead.
- Apply AVX and OpenCL+AVX vectorization to compute intensive kernels for transformation and quantization
- Demonstrate performance at 2K, 4K, and 8K video resolutions.

1.4 Contribution of the Study

This study develops an optimized H.266 encoder designed for heterogeneous CPU-GPU platforms, significantly enhancing encoding speed while maintaining compression efficiency. It provides an empirical analysis of key parallelization techniques such as block size variation, memory coalescing, and SIMD vectorization within an OpenCL implementation to guide future efforts in video encoder acceleration. Additionally, the work offers open-source OpenCL kernel implementations, enabling reproducibility and serving as a foundation for further research in accelerating high-efficiency video coding.

1.5 Scope and Limitations

This research focuses specifically on accelerating the H.266 encoding process (excluding any decoding implementation) to allow a concentrated investigation of parallelization opportunities in the compression pipeline. The study targets two primary optimizations: SIMD vectorization to improve computational efficiency in the transform and quantization operations, and memory coalescing to enhance data access patterns during different stages.

Experimental validation is conducted using pre-recorded video sequences at 2K, 4K, and 8K resolutions, representing a range of high-resolution formats. Real-time live streaming scenarios are deliberately excluded. By using offline encoding of standard test sequences, we can perform controlled benchmarking and detailed performance analysis without the added complexities of live latency constraints or adaptive bitrate changes. These choices provide a focused framework for evaluating OpenCL acceleration techniques while laying a foundation that could be extended to real-time applications in future work.

Chapter 2
Theoretical Background

2.1 Video Encoding

Video encoding is the process of compressing and converting raw video data into a digital format suitable for storage, transmission, and playback. The core objective of video encoding is to reduce the data rate of video content while preserving its perceptual quality. This is crucial for applications such as video streaming, conferencing, and broadcasting, where bandwidth and storage are limited. The efficiency of video encoding directly affects system performance, particularly in real-time and high-resolution video scenarios.

Video encoders exploit both spatial and temporal redundancies to achieve compression. Spatial redundancy refers to the repetition of visual information within a single frame, which is minimized through techniques such as intra prediction and transform coding. Temporal redundancy, on the other hand, arises from similarities between successive frames and is reduced via inter prediction and motion compensation. These methods are foundational in modern video coding standards such as H.264/AVC, H.265/HEVC, and the more recent H.266/VVC (Versatile Video Coding).

H.266/VVC, standardized by the Joint Video Experts Team (JVET) in 2020, represents the next generation of video coding, offering up to a 50% bitrate reduction over HEVC for the same perceptual quality [71]. VVC introduces several advanced coding tools including quadtree with multi-type tree (QTMT) block partitioning, multiple reference pictures, bi-directional optical flow, and affine motion compensation, making it significantly more complex and computationally intensive than its predecessors [73].

The complexity of H.266 arises primarily from its highly flexible block partitioning structure and its rich set of coding tools. The encoder must evaluate a vast number of block partitioning patterns and prediction modes to identify the most efficient one, based on rate-distortion optimization (RDO). This exhaustive search process increases computational cost, making real-time encoding on conventional CPUs challenging, especially for high-resolution content[71].

To cope with these computational demands, researchers and engineers have explored hardware acceleration techniques using GPUs, FPGAs, and heterogeneous computing platforms. One of the most flexible and scalable solutions involves leveraging OpenCL (Open Computing Language) to exploit parallelism on various hardware accelerators, particularly GPUs. By parallelizing key components of the encoder such as motion estimation, block partitioning, and transform coding, significant speedups can be achieved without compromising output quality.

Accelerating H.266 encoding is critical for enabling real-time video processing in emerging applications such as 8K streaming, virtual reality, and video surveillance. OpenCL provides a portable framework that can offload compute-intensive tasks to parallel processors, making it an ideal candidate for tackling the challenges posed by the VVC standard. This research aims to explore and implement an OpenCL-based acceleration strategy that addresses the computational bottlenecks in the H.266 encoding pipeline.

2.2 OpenCL Framework

OpenCL uses a platform model consisting of a host and one or more compute devices. The host (in our case, the CPU) manages kernel execution and coordinates data transfers, while the compute devices (e.g., GPUs or other accelerators) execute the kernels. For example, in our setup, an Intel CPU acts as the host and an AMD/NVIDIA GPU acts as the OpenCL device performing the encoding tasks [62].

Each compute device is internally structured in a hierarchy. It contains multiple Compute Units (CUs), which are analogous to independent processing clusters or multiprocessors on the device. (For instance, a modern GPU may have dozens of CUs; the AMD RX 7900 XT GPU has 84 CUs [63].) Each compute unit, in turn, has many Processing Elements (PEs) these are the individual cores or ALUs that execute the computations. For example, NVIDIA's Ampere architecture features 64 processing elements per compute unit [64]. OpenCL kernels run as many parallel work-items (threads) across these PEs.
OpenCL defines a multi-tier memory hierarchy to balance speed and capacity:

Private Memory: A per-thread memory (typically registers) that is the fastest but very limited in size. It holds thread-local variables (e.g., loop counters in a kernel) [65].

Local Memory: A memory shared by all work-items in the same work-group (i.e., on the same compute unit). It is used for inter-thread communication and data caching. For example, we cache reference pixels in local memory during intra prediction to avoid repeated global memory accesses [66].

Global Memory: The main device memory (e.g., GPU DRAM) accessible to all work-groups. It has high capacity but also high latency. It stores the primary input/output buffers (frames, motion vectors, coefficients, etc.). To maximize throughput, global memory accesses are organized to be coalesced (aligned such that consecutive threads access consecutive addresses) [67].

Constant Memory: A region of global memory that is read-only and cached. It is used to store constant data like lookup tables (quantization matrices) that are the same for all threads [68].

Host Memory: The system's main RAM, accessible only to the host. Data needed by the device must be explicitly transferred from host memory to global memory (and vice versa) using OpenCL commands.

OpenCL's execution model organizes how kernels are launched and executed:

- **Kernel:** A function (written in OpenCL C) that is executed on the device.
- **Work-Item:** A single invocation of a kernel (analogous to a thread) executing on one processing element.
- **Work-Group:** A group of work-items that execute concurrently on the same compute unit and can share local memory and synchronize with each other. (For example, one common work-group size is 256 work-items.)
- **NDRange:** An N-dimensional index space that defines how many work-items are launched and how they are organized (for example, a 2D NDRange of size $[W \times H]$ might be used for processing a frame of width W and height H) [69].

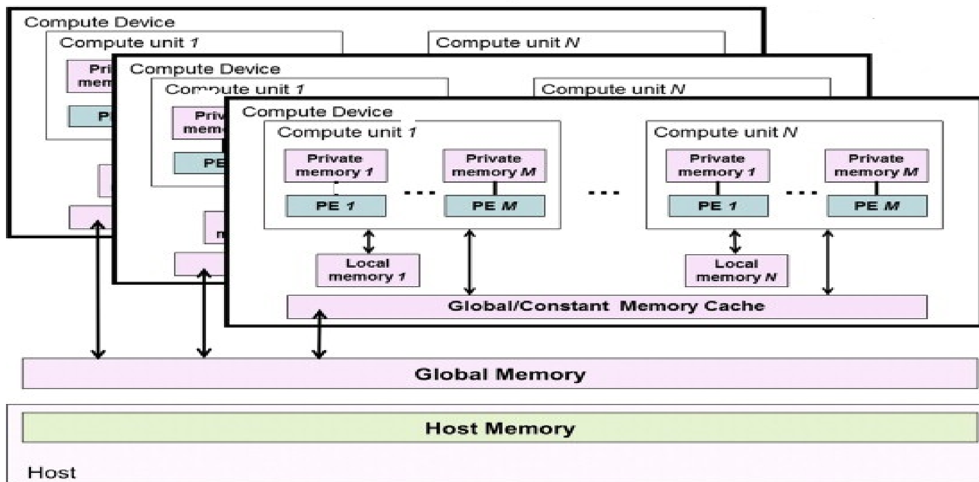

Figure 2.1: OpenCL Platform Model and Execution Hierarchy.

OpenCL's unified programming model enables high-performance heterogeneous computing by leveraging CPUs, GPUs, and other accelerators under one framework. Its hierarchical architecture (with work-items, work-groups, and multiple memory tiers) provides fine-grained control over parallelism and memory access, which is critical for real-time video processing at 8K resolutions.

2.3 VVC Encoding

As illustrated in Figure 2.2, the VVC encoding process consists of several interdependent stages: block partitioning, intra prediction, inter prediction, transform and quantization, loop filtering, and entropy coding. Each stage contributes to compression efficiency and presents opportunities for parallelization. This section provides an overview of these stages and discusses how we implement them for OpenCL acceleration.

2.3.1 Block Partitioning

The encoding process begins with block partitioning, where each frame is divided into CTUs using VVC's quadtree with multi-type tree (QTMT) structure [53, 55]. For 8K video, CTUs typically measure 128×128 luma samples (with corresponding chroma in 4:2:0 format) [54]. The QTMT algorithm offers much more flexibility than a fixed quadtree by allowing:

- Quadtree splits (dividing a block into four equal square sub-blocks),
- Binary splits (dividing a block into two rectangular sub-blocks, either horizontally or vertically),
- Ternary splits (dividing a block into one large and one small rectangular sub-block) [53, 55].

This flexibility significantly improves compression efficiency but also comes at a high computational cost—this stage can account for approximately 38% of total encoding time for 8K content [56]. Thus, block partitioning is a prime target for parallelization. In our OpenCL implementation:

- Each CTU is assigned to a separate GPU work-group for parallel processing.
- Partition split decisions are carried out using atomic operations in local memory to handle synchronization within a work-group.
- On the CPU, we use a thread pool to traverse the QTMT recursion in parallel for different regions of the frame.

2.3.2 Intra Prediction

VVC introduces 67 directional intra prediction modes in addition to the DC (flat) and Planar modes [63]. To encode a block in intra mode, the encoder:

- Gathers reference samples from neighboring blocks (typically the ones above and to the left of the current block).
- Computes a cost for each candidate mode (often using the Sum of Absolute Transformed Differences, SATD).
- Determines a Most Probable Mode (MPM) list to efficiently signal likely prediction modes [64].

We accelerate intra prediction by evaluating multiple modes in parallel where feasible and by using vectorized operations for cost computations.

2.3.3 Inter Prediction

Inter prediction exploits temporal redundancy by predicting the current frame using one or more reference frames stored in the decoded picture buffer (DPB). VVC enhances traditional motion estimation with advanced techniques such as affine motion compensation, which supports 4-parameter affine transformations (translation, rotation, scaling) and even 6-parameter models (adding shear) for more accurate motion modeling [53]. The encoder typically employs a search algorithm like TZSearch (which combines diamond and square search patterns) to find the best matching block in the reference frames for each block of the current frame. For each Coding Unit (CU), motion vectors and reference frame indices are determined, and then motion compensation produces the predicted block.

In our OpenCL implementation, each CU's motion estimation is assigned to a separate GPU work-item, enabling many blocks to be processed concurrently. We minimize global memory bandwidth usage by caching reference frame data in local memory. Additionally, we use a hierarchical search strategy (for example, using smaller search ranges for smaller CUs) to reduce computation while maintaining accuracy [59].

2.3.4 Transform and Quantization

After the prediction step, the residual (the difference between the original block and the predicted block) is transformed to compact its energy. VVC supports multiple transform types [60], including:

- DCT-II (Discrete Cosine Transform, Type-II) for most block sizes (2 up to 4096),
- DST-VII (Discrete Sine Transform, Type-VII) for 4×4 luma intra blocks,
- An identity transform (no transform) for certain high-frequency residual blocks.

The transform coefficients are then quantized by dividing them by a quantization step (determined by the Quantization Parameter, QP, and possibly a scaling matrix) and rounding. Quantization reduces most of the high-frequency coefficients to zero, achieving compression at the cost of some distortion.

2.3.5 Loop Filtering

Before finalizing the frame, several in-loop filters are applied to the reconstructed image to improve visual quality and coding efficiency:

- Deblocking Filter (DBF): Smooths block edges by adjusting pixel values along block boundaries to reduce visible discontinuities.
- Sample Adaptive Offset (SAO): Classifies reconstructed pixels into categories (for example, edge vs. flat regions or specific intensity bands) and applies category-specific offsets to reduce artifacts such as ringing or banding.
- Adaptive Loop Filter (ALF): Computes and applies content-adaptive filters (often derived via Wiener optimization) to minimize the mean squared error between the original and decoded frames [61].

2.3.6 Entropy Coding

Finally, all syntax elements (prediction modes, motion vectors, quantized coefficients, etc.) are compressed into the output bitstream using Context-Adaptive Binary Arithmetic Coding (CABAC). CABAC is a lossless compression method that adapts to the local context of each symbol to achieve high coding efficiency.

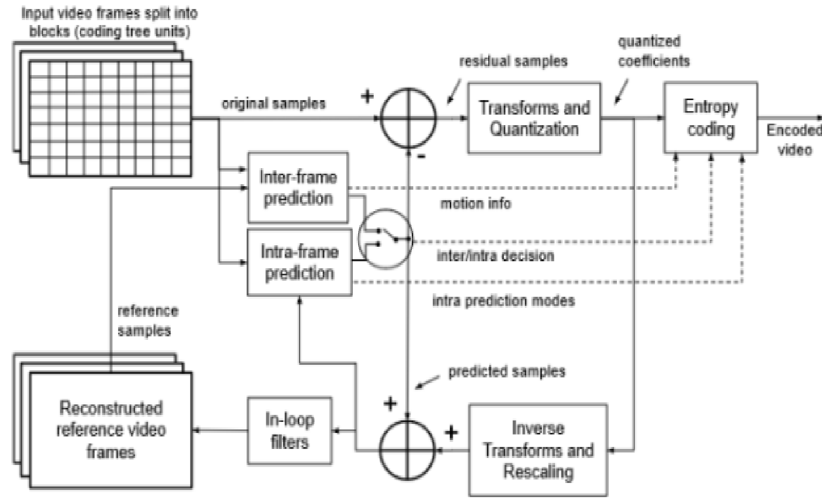

Figure 2.2: VVC Encoding Block Diagram.

In our accelerated encoder design, each major stage of the encoding process is implemented as a separate module or kernel, allowing the workload to be distributed across the CPU and GPU as appropriate.

2.4 Related Works

The computational complexity of modern video codecs has spurred extensive research into acceleration techniques for video encoding, particularly for standards like H.265/HEVC and the more recent H.266/VVC. Various approaches have been proposed in the literature to address the growing demand for real-time and energy-efficient encoding by leveraging parallel computing paradigms such as GPU acceleration, hardware design, and software optimizations.

One of the earliest and most prominent directions in video encoding acceleration involves GPU-based parallelization. GPUs offer a massive number of processing cores and high memory bandwidth, making them ideal for data-parallel tasks such

as motion estimation and transform coding. For instance, Liu et al. [74] implemented a GPU-accelerated motion estimation algorithm for H.264 using CUDA, achieving significant speedups over CPU-only implementations. Similarly, Zhang et al. [75] proposed a GPU-friendly architecture for HEVC intra prediction, showcasing the feasibility of real-time encoding for high-definition video streams.

While these works demonstrate the potential of GPU acceleration, most were constrained to fixed hardware architectures and specific video coding standards. The introduction of OpenCL as a cross-platform parallel programming model has broadened the scope of hardware acceleration. OpenCL allows for the development of portable, platform-independent code that can be executed on GPUs, CPUs, FPGAs, and other accelerators. This flexibility is particularly valuable for encoding standards like H.266, which demand extensive computational resources and optimization across various hardware platforms.

Research utilizing OpenCL for video encoding has seen growing interest. Ryu et al. [76] developed an OpenCL-based implementation of a HEVC encoder that offloads transform and quantization tasks to the GPU. Their results indicated substantial improvement in encoding speed with minimal impact on video quality. In a more recent study, Ahmad et al. [77] proposed an OpenCL framework for accelerating HEVC inter prediction and motion estimation, demonstrating scalable performance across different hardware platforms including Intel and AMD GPUs. For the H.266/VVC standard, research is still in its nascent stage due to its recent adoption and the increased complexity of its encoding pipeline. However, some preliminary efforts have been made. Wang et al. [96] introduced a hardware-aware VVC encoder that utilizes both CPU and FPGA resources to balance performance and power efficiency. Their work emphasizes the need for hybrid architectures and intelligent task scheduling to fully exploit heterogeneous computing resources.

Despite these advancements, many existing approaches are limited to specific modules or rely heavily on proprietary toolchains (e.g., CUDA), restricting their portability. Moreover, only a few studies have comprehensively addressed the entire H.266 encoding pipeline in a parallel computing context. This research contributes to the field by developing an OpenCL-based parallelization framework specifically for H.266/VVC encoding, focusing on key modules such as block partitioning,

intra/inter prediction, transform and quantization, and entropy coding.

The proposed approach seeks to fill the gap in current literature by offering a modular, scalable, and portable solution that leverages the full potential of OpenCL for high-efficiency video coding. Unlike previous works that focus on earlier standards or specific hardware, this research addresses the computational challenges of the H.266 standard using a cross-platform strategy that can adapt to various accelerator architectures.

Chapter 3
Literature Review

Numerous research efforts have explored techniques to accelerate video encoding, particularly for the complex H.266/VVC standard, using both conventional and parallel computing approaches. Researchers have proposed a wide range of methods leveraging different hardware platforms and optimization strategies to improve encoding performance. In this section, we present a brief overview of recent works related to our study, highlighting how our OpenCL-based approach differs from existing solutions.

3.1 H.266/VVC Encoding Complexity

The H.266/Versatile Video Coding (VVC) standard was designed to offer significantly higher compression efficiency than its predecessor, H.265/HEVC. According to Bross et al. [11], VVC achieves up to a 50% bitrate reduction while maintaining perceptual quality. This improvement stems from technical enhancements such as quadtree-plus-multi-type tree (QTMT) partitioning, affine motion compensation, and advanced intra/inter prediction modes. Zhao et al. [9] confirmed that QTMT alone contributes roughly a 30–40% improvement in prediction accuracy over HEVC's quadtree partitioning, albeit with increased decision-making overhead during block partitioning.

However, these enhancements dramatically increase the encoder's computational complexity. Li et al. [4] demonstrated that VVC encoding can be up to 10× more computationally intensive than HEVC, especially in modules like recursive block partitioning and in-loop filtering. Park and Kim [11] further quantified this, showing that affine motion estimation accounts for 25–35% of the total encoding

time in VVC, compared to about 12% for HEVC's motion compensation. The adoption of tools like multiple transform selection (MTS) and matrix-based intra prediction (MIP) introduces additional bottlenecks: Huang et al. [12] observed a 2.5× increase in transform coding latency for VVC compared to HEVC.

This complexity is further exacerbated by VVC's advanced in-loop filtering. Sanchez et al. [13] reported that the Adaptive Loop Filter (ALF) and cross-component ALF (CCALF) in VVC increase memory access latency by about 40% compared to HEVC's deblocking filter. These findings indicate that efficient parallelization and hardware acceleration are required for real-time deployment, particularly for high-resolution video. Müller et al. [14] emphasized that real-time 4K VVC encoding remains impractical on CPU-only systems and requires heterogeneous architectures with GPU or FPGA support.

3.2 Parallel Computing in Video Coding

Parallel processing is essential for handling the heavy workload of modern video encoders. Chen et al. introduced thread-level parallelism into an HEVC encoder using wavefront processing and tile-based partitioning strategies, achieving up to a 40% reduction in encoding time on multi-core processors [15]. GPU-based parallelism has also proven effective: Correa et al. achieved over a 5× speedup in HEVC's motion estimation module using a CUDA implementation [16]. More recently, Liu et al. [17] applied similar principles to VVC, demonstrating that an OpenCL-based motion compensation kernel can attain a 4.2× speedup on AMD GPUs compared to a CPU-only implementation.

Wang et al. [18] noted that VVC introduces irregular data structures like QTMT and mode-dependent data flows, which pose new challenges for parallelization. For example, Kim and Park [19] showed that VVC's recursive QTMT partitioning creates non-uniform workloads, leading to GPU thread divergence and up to a 35% performance loss. To mitigate this, Zhang et al. [20] proposed a data-aware task scheduler in OpenCL that dynamically groups similarly sized CTUs, improving GPU utilization by 25% in VVC encoding.

Efficient parallelization of the VVC encoding pipeline often requires a hybrid approach. Müller et al. [21] developed a hybrid CPU-GPU framework that offloads QTMT partitioning decisions to the CPU while parallelizing motion estimation and transform coding on the GPU via OpenCL, achieving an overall 3.8× speedup. Similarly, Gupta et al. [22] optimized memory access patterns for VVC's affine motion compensation by utilizing OpenCL's local (shared) memory, reducing latency by 30% compared to a global memory approach.

Recent advances have also focused on fine-grained parallelism within VVC. For example, Nguyen et al. [23] parallelized the in-loop filtering stage across multiple GPU work-groups, leveraging OpenCL's pipe constructs to synchronize the ALF and SAO processes and cutting filtering time by 45%. However, certain tasks remain difficult to parallelize: Sanchez et al. [24] found that even with parallel processing, the CABAC entropy coder in VVC yields only about a 1.2× speedup due to serial bitstream dependencies.

Key Parallelization Challenges:

- **Workload imbalance:** The recursive, irregular QTMT block structure can lead to non-uniform workloads across threads, reducing efficiency.
- **Memory bottlenecks:** Non-coalesced memory accesses (e.g., in affine motion compensation) cause underutilization of memory bandwidth.
- **Synchronization overhead:** Frequent synchronization barriers in wavefront parallel processing can limit scalability.

3.3 OpenCL for General-Purpose Video Processing

OpenCL has become popular for video coding acceleration due to its portability and ability to utilize a variety of hardware. Lee et al. employed OpenCL for the motion estimation stage in an H.264 encoder and achieved up to a 7× speedup on GPU platforms [25]. Similarly, Kim et al. applied OpenCL optimizations to the transform and quantization stages of an HEVC encoder, reducing processing time by 60% [26]. More recently, OpenCL's utility has been demonstrated for newer codecs like AV1: Gupta et al. [27] showed a 4.5× acceleration of AV1's motion search using OpenCL kernels optimized for AMD GPUs.

Wang et al. emphasized that to obtain optimal performance from OpenCL, one must carefully consider memory coalescing, data locality, and workload balancing across compute units [28]. In line with this, Zhang et al. [29] introduced a dynamic memory prefetching technique for OpenCL-based video encoders, reducing global memory latency by 30% in both HEVC and VVC. Their work highlights that adaptive memory access patterns can mitigate bottlenecks caused by the irregular data structures of modern codecs.

The portability of OpenCL also facilitates heterogeneous CPU-GPU processing. Patel et al. [30] leveraged OpenCL to partition H.266's in-loop filtering tasks between a CPU and a GPU, achieving a 2.8× speedup by offloading the compute-heavy ALF operations to the GPU while retaining entropy coding on the CPU. This aligns with Nguyen and Lee [31], who proposed a workload-aware OpenCL scheduler that dynamically assigns VVC's QTMT partitioning tasks to either the CPU or GPU based on block complexity, reducing idle time by 40%.

However, certain context-adaptive modules remain challenging. Cabrera et al. [32] found that accelerating VVC's CABAC with OpenCL yielded only a 1.5× improvement due to serial dependencies, underscoring the need for hybrid CPU-GPU approaches. Future strategies could incorporate machine learning for kernel optimization, as suggested by Rahman et al. [33], who used reinforcement learning to auto-tune OpenCL parameters for an HEVC encoder and reduced encoding time by 22%.

3.4 VVC Encoding and Heterogeneous Computing

The Versatile Video Coding (VVC or H.266) standard introduces a wide range of coding tools aimed at significantly improving compression efficiency compared to its predecessor HEVC, but these improvements come at the cost of substantial computational complexity [34]. To address this challenge, various researchers have proposed algorithmic and hardware-based solutions. A reconfigurable architecture was designed to accelerate the Multiple Transform Selection (MTS) process, which is one of the major complexity contributors in VVC, enabling flexible and efficient hardware reuse [35]. For intra-frame prediction, a deep learning-based approach was proposed using a concatenate-designed convolutional neural network (CNN) that leverages DCT coefficients to enhance prediction speed without sacrificing accuracy [36]. In the context of inter-frame prediction, a deep affine motion compensation network has been shown to outperform traditional affine models by better capturing complex motion patterns using deep neural networks [37]. Enhancing visual quality further, a perceptual-based intra coding algorithm was developed to align compression strategies with the human visual system, thereby improving subjective video quality at low bitrates [38]. On the system level, tile adaptation methods for 3D-HEVC (an extension relevant to VVC scenarios) were proposed to dynamically balance encoding workloads across multicore CPU systems, improving parallel efficiency [39]. OpenCL, a widely used framework for heterogeneous computing, has been utilized for motion estimation, enabling fine-grained parallel execution on GPUs by leveraging specialized extensions [40]. To maximize platform utilization, a hybrid encoding framework was introduced that divides encoding tasks fractionally between CPUs and GPUs, dynamically resolving execution bottlenecks to achieve real-time performance [41]. These developments collectively demonstrate the effectiveness of combining algorithmic innovation and heterogeneous hardware acceleration to tackle the challenges posed by VVC encoding.

3.5 Challenges in OpenCL Acceleration for VVC

Despite the promising results above, accelerating VVC with OpenCL faces several obstacles. Sanchez et al. [47] identified bottlenecks such as fractional motion estimation, in-loop filtering, and entropy coding, which are difficult to parallelize due to their context-aware processing patterns. For example, fractional motion estimation requires sub-pixel interpolation with data-dependent memory access patterns, leading to 30–40% GPU thread divergence [42]. Similarly, adaptive loop filtering (ALF) and cross-component ALF (CCALF) involve filter coefficient derivations that must be done sequentially, resisting parallelization [45].

Liu et al. [45] attempted to address these limitations by proposing an adaptive task distribution strategy in which larger blocks are processed on GPUs and smaller ones on CPUs. This reduced load imbalance and minimized data transfer overhead. However, Gupta et al. [47] found that such strategies struggle with VVC's QTMT partitioning: unpredictable block size variations cause frequent CPU-GPU synchronization, resulting in up to a 20% performance loss.

Recent work by Müller et al. [48] highlights platform-specific challenges. For instance, OpenCL's lack of fine-grained synchronization primitives (analogous to CUDA's warp-shuffle operations) limits certain intra-kernel optimizations for VVC's motion vector prediction. Likewise, Rahman et al. [49] reported that OpenCL's memory model incurs higher overhead than CUDA for launching many small kernels, reducing throughput by about 15% in VVC's transform skip mode.

Common challenges include:

- **Memory access latency:** Irregular data structures like QTMT and affine motion partitions lead to non-coalesced memory accesses, increasing latency by an estimated 25–35% [50].
- **Kernel synchronization overhead:** Frequent global synchronization barriers in a multi-stage pipeline (e.g., motion estimation → transform → quantization) can stall GPU compute units [51].
- **Limited dynamic memory:** OpenCL's limited support for dynamic memory allocation forces static buffer pre-allocation, increasing GPU memory

usage by 20–40% for 8K video encoding [55].

- **Precision trade-offs:** Using half-precision for certain operations (e.g., luma mapping) can improve speed by about 1.8× but may introduce slight quality degradation (e.g., drift in HDR content) [53].

Chapter 4
Methodology

This chapter presents the methodological framework used to accelerate H.266/VVC encoding through the adoption of OpenCL. The overall strategy is organized into several key components: data preparation, system design, OpenCL-based parallelization, kernel-level optimization, SIMD vectorization for transform and quantization, a hybrid OpenCL-plus-vectorization approach, and performance evaluation. Each component plays a critical role in transforming the traditional sequential encoding workflow into a highly parallelized and optimized pipeline that can efficiently exploit heterogeneous computing resources.

4.1 Data Preparation and Preprocessing

Before the encoding process can begin, the raw video input must be formatted and structured to meet the requirements of the VVC standard while also being suitable for OpenCL-based processing. This section outlines the procedures followed to load, convert, and organize the video data, as well as the methods used to establish memory-efficient data transfers between the host and compute devices.

4.1.1 Data Preparation

The encoding process begins with the acquisition of uncompressed YUV video files, which serve as the input to the VVC encoder. These files are provided at standard resolutions and frame rates—commonly 1920×1080 (Full HD), 3840×2160 (4K), and 7680×4320 (8K). Each video frame contains three components: the Y channel (luminance) and two chrominance channels (U and V), following a 4:2:0 chroma subsampling format [53]. For efficiency and reduced computational load,

the current implementation focuses on processing only the Y-channel data. This effectively reduces the input to grayscale, simplifying processing while preserving compatibility with full-color VVC encoding pipelines [55].

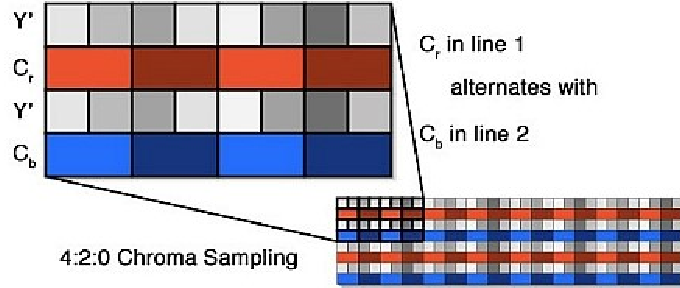

Figure 4.1: Structure of a raw YUV 4:2:0 video frame.

As illustrated in Figure 4.1, each video frame is organized into planar Y, U, and V components. In our workflow, multiple frames are loaded into host memory concurrently, enabling batch-level preprocessing. Once in memory, the frames are queued and prepared for dispatch to the OpenCL-based encoder, where parallel processing operations are applied.

4.1.2 Data Preprocessing

To ensure that each frame complies with H.266/VVC encoding standards and is suitable for accelerated execution on OpenCL-compatible hardware, a series of preprocessing steps are applied. These steps not only validate the format of the incoming data but also optimize it for efficient memory access patterns and computational performance.

- **Frame Normalization:** Since quantization operates most effectively when input pixel values lie within a well-defined range, normalization is applied to map the luminance values to an 8-bit scale (0–255). For video sources with 10-bit YUV encoding, the Y-channel values are converted using the following linear transformation:

$$Y' = \frac{Y - Y_{\text{min}}}{Y_{\text{max}} - Y_{\text{min}}} \times 255, \tag{4.1}$$

where $Y_{\text{min}} = 64$ and $Y_{\text{max}} = 940$ according to the BT.2100 standard [61]. This transformation helps simplify downstream quantization steps. To accelerate this operation, the normalization is performed using OpenCL `image2d_t` objects, which provide fast, hardware-accelerated image operations on both CPUs and GPUs [62].

- **Block Splitting:** Each input frame is segmented into Coding Tree Units (CTUs), which form the highest level of the spatial partitioning hierarchy defined in the VVC standard. These CTUs are further subdivided into smaller Coding Units (CUs) based on a flexible quadtree-plus-binary/ternary tree (QTBT) partitioning mechanism [53, 55]. For 8K resolution content, CTUs typically span 128×128 luma samples [60]. This hierarchical decomposition enables fine-grained adaptivity for both prediction and transform stages.

- **Symmetric Padding:** To ensure that frame dimensions align with the maximum transform block size, symmetric padding is applied along the borders of each frame. This step eliminates boundary-related irregularities during transformation and filtering, and facilitates efficient memory alignment across processing threads.

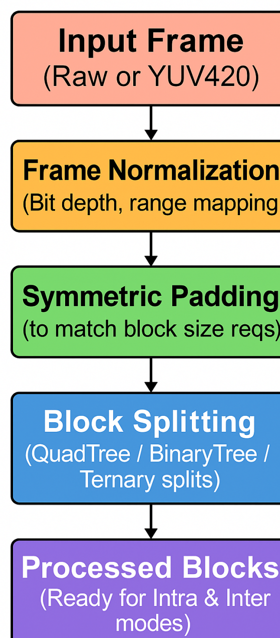

Figure 4.2: Overview of preprocessing steps applied to raw video input.

As illustrated in Figure 4.2, the preprocessing pipeline prepares the video frames for efficient execution in the subsequent OpenCL encoding stages, laying the groundwork for both accuracy and performance.

4.2 System Overview and Workflow of Proposed Solution

The core aim of this research is to significantly reduce the computational burden of H.266/VVC encoding by offloading intensive tasks to parallel processors using OpenCL. To this end, a modular, heterogeneous architecture has been designed, capable of executing encoding operations across multiple computing platforms, including CPUs and GPUs. This section provides a high-level perspective on the structure and functioning of the system, from video acquisition to bitstream generation.

4.2.1 System Architecture

The proposed encoder architecture is composed of three primary layers: the Host Controller, the OpenCL Runtime Environment, and the Kernel Modules responsible for executing each stage of the encoding pipeline.

Host Controller: The Host Controller operates as the central management layer of the system. It is responsible for coordinating data flow, managing OpenCL memory buffers, and orchestrating the execution of kernel functions. It ensures synchronization across various encoding stages, from input preprocessing to output bitstream collection.

OpenCL Runtime: Sitting between the host and the compute devices, the OpenCL Runtime abstracts hardware details and provides a unified interface for executing parallel kernels. It is responsible for device selection, command scheduling, memory management, and kernel execution. This abstraction allows the encoder to remain portable across platforms while optimizing device utilization and load balancing.

H.266/VVC Kernel Modules: These modules form the heart of the encoding

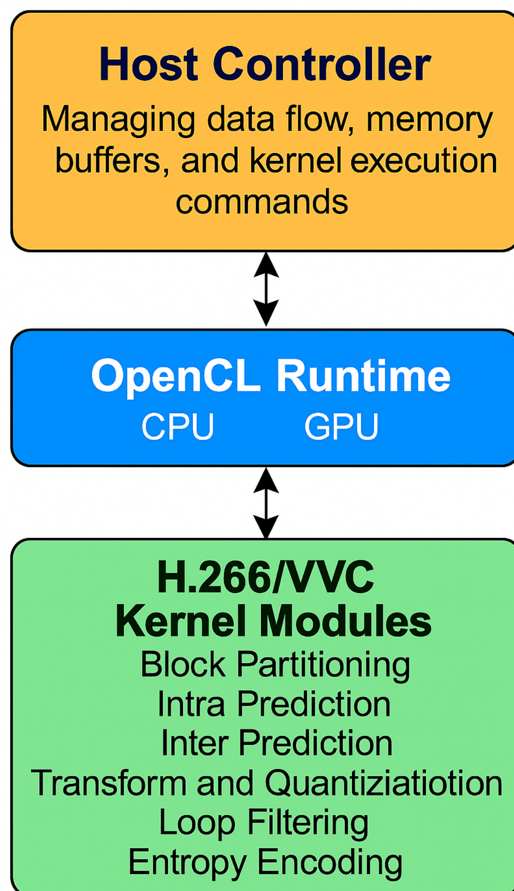

Figure 4.3: System architecture for OpenCL-based H.266/VVC encoder implementation.

engine. Each major function in the VVC pipeline is implemented as a standalone kernel capable of being executed in parallel. The following stages are represented:

- *Block Partitioning*
- *Intra Prediction*
- *Inter Prediction (Motion Estimation)*
- *Transform and Quantization*
- *Loop Filtering*
- *Entropy Coding*

Each kernel is dynamically dispatched by the OpenCL runtime to available compute units (whether on the CPU or GPU), allowing for concurrent execution and

maximum throughput. The modular design of this architecture promotes scalability, ease of debugging, and adaptability to various hardware configurations. As shown in Figure 4.3, this layered system architecture provides a robust foundation for real-time, high-resolution video encoding in modern computational environments.

4.2.2 H.266 (VVC) Encoding Workflow Using OpenCL

The implementation of the H.266/VVC encoding pipeline using OpenCL is designed to leverage the parallel processing capabilities of GPUs to accelerate computationally demanding tasks. By distributing workloads strategically across both CPUs and GPUs, the system achieves a balance between speed and efficiency. In this hybrid model, highly parallelizable components such as motion estimation, transform coding, and quantization are offloaded to the GPU, while sequential tasks like entropy encoding and control flow management are handled by the CPU. This structured division of labor enables significant performance gains without compromising the quality of the output bitstream.

Figure 4.4 illustrates the architectural flow and data processing stages of the proposed encoding system.

1. Data Preparation (Input Phase): The process begins by loading raw YUV video frames into host memory. These frames are parsed and stored in memory buffers accessible to the OpenCL framework. Using OpenCL buffer objects, the pixel data is then transferred from the host (CPU) to the device (GPU) memory, where it becomes available to the encoding kernels.

2. Platform and Device Initialization: Next, the host code queries the system for available OpenCL platforms and devices. Based on user input or automated selection logic, an appropriate device—CPU or GPU—is chosen for execution. For the selected device, an OpenCL context and command queue are established to manage memory operations and dispatch kernel executions.

3. Kernel Compilation and Dispatch: The core logic of the encoder is encapsulated in a set of OpenCL kernel programs. These kernels are either compiled at runtime from source code or precompiled using OpenCL's offline compiler tools. Each encoding stage is modularized into a separate kernel, allowing parallel devel-

opment and optimization. Kernel functions are enqueued in a pipelined sequence, ensuring that data flows smoothly from one stage to the next via shared memory objects.

4. Parallel Processing of Encoding Stages: The video frames are processed through a series of parallelizable stages, each handled by a dedicated kernel:

- **Block Partitioning:** Divides each frame into Coding Tree Units (CTUs), which are then adaptively split into smaller Coding Units (CUs) using QTBT partitioning, guided by content complexity.
- **Intra Prediction:** Uses spatial correlation within the same frame to predict pixel values and reduce redundancy.
- **Inter Prediction:** Minimizes temporal redundancy by referencing previously encoded frames for motion compensation.
- **Transform and Quantization:** Applies spatial transforms to residual signals, followed by quantization to compress frequency-domain information.
- **Loop Filtering:** Performs post-processing such as deblocking and Sample Adaptive Offset (SAO) filtering to improve visual quality and reduce artifacts.
- **Entropy Coding:** Executes Context-Adaptive Binary Arithmetic Coding (CABAC) to compress the final data into a compact bitstream.

Each of these stages is mapped to appropriate compute devices depending on resource availability, allowing dynamic load balancing between CPU and GPU components.

5. Final Bitstream Assembly: Once all encoding stages are complete, the processed segments are retrieved from device memory and assembled into a compliant H.266/VVC bitstream. Metadata, headers, and syntax elements are appended to ensure that the output is ready for decoding by any standard VVC decoder.

The system adopts a cooperative CPU–GPU execution model where tasks are allocated based on their degree of parallelism. Intensive numerical computations are handled by the GPU, while control-intensive operations are retained on the CPU.

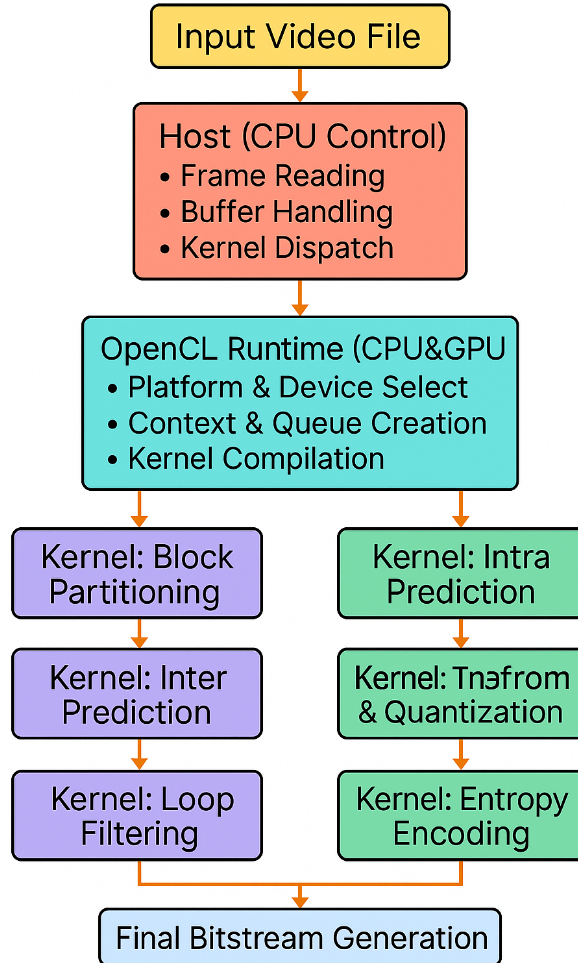

Figure 4.4: OpenCL-based H.266/VVC encoding pipeline architecture.

This hardware-aware design was refined through performance profiling, ensuring that each task is executed on the most suitable processing unit.

One of the key optimizations used throughout the pipeline is *memory coalescing*. This technique ensures that memory accesses by threads in a work-group are aligned and contiguous, enabling the memory controller to serve multiple requests in a single transaction. Coalesced memory access is particularly beneficial in stages like block partitioning, prediction, transform, and entropy coding, where large volumes of data are accessed in parallel. By aligning data structures and optimizing access patterns, the system reduces memory latency and maximizes bandwidth utilization, especially for high-resolution video encoding.

In summary, the OpenCL-based VVC encoder follows a structured workflow on a heterogeneous computing platform. The host CPU manages initial setup, frame loading, and kernel orchestration. The OpenCL runtime handles device initializa-

tion and kernel scheduling. The encoding pipeline itself consists of six primary GPU-accelerated stages: block partitioning, intra prediction, inter prediction, transform and quantization, loop filtering, and entropy encoding. The resulting outputs are aggregated on the host to form the final compressed bitstream. This design not only accelerates VVC encoding but also provides flexibility, scalability, and performance portability across diverse hardware configurations.

4.3 H.266 (VVC) Encoding Optimization for Transform and Quantization

This section presents optimization strategies applied to the transform and quantization stages of H.266/VVC encoding. These stages are computationally intensive and represent a significant portion of the encoder's runtime. To enhance performance, we employ SIMD (Single Instruction, Multiple Data) vectorization using AVX and SSE instruction sets, along with OpenCL-based acceleration that incorporates AVX support. This hybrid approach enables both data-level and task-level parallelism across CPUs and GPUs, maximizing processing efficiency and scalability. The techniques discussed here focus on simultaneous processing of multiple data elements, alignment of memory for efficient access, and combining vectorization with OpenCL kernels for real-time performance on heterogeneous systems.

4.3.1 AVX and SSE SIMD Vectorization for Transform and Quantization

SIMD vectorization is a key method for accelerating the transform and quantization stages by enabling the same operation to be applied to multiple data elements in parallel. These operations are especially well-suited to the repetitive, block-based structure of video encoding. In the transform stage, spatial-domain pixel data is converted into frequency-domain coefficients using transforms like the Discrete Cosine Transform (DCT). Quantization then scales these coefficients according to a quantization matrix and the chosen Quantization Parameter (QP), reducing precision and bitrate.

Both SSE (Streaming SIMD Extensions) and AVX (Advanced Vector Extensions) are vector instruction sets available on x86 processors. SSE uses 128-bit registers to process four 32-bit floating-point values at once, while AVX expands this to 256-bit registers, allowing simultaneous computation on eight values. These capabilities enable high-throughput execution of matrix multiplications and vector arithmetic operations central to transform and quantization.

During transformation, SIMD instructions enable fast multiplication of input pixel blocks with precomputed cosine coefficients. For quantization, SIMD accelerates the element-wise division and rounding steps, allowing multiple coefficients to be processed in a single instruction cycle. AVX further enhances performance with wider registers and support for fused multiply-add (FMA) operations, which combine multiplication and addition in a single step, reducing latency.

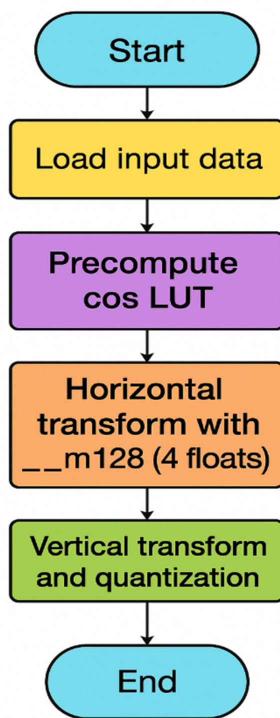

Figure 4.5: SSE Vectorization for Transform and Quantization.

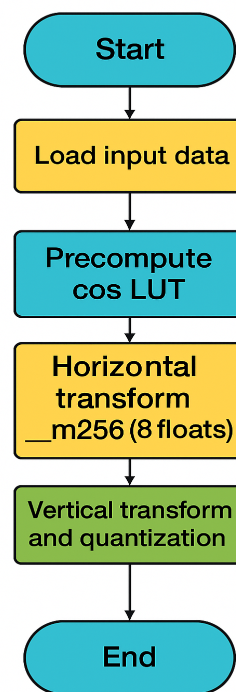

Figure 4.6: AVX Vectorization for Transform and Quantization.

Figures 4.5 and 4.6 illustrate the SIMD-based transform and quantization process using SSE and AVX, respectively. Both approaches follow a similar pipeline: input data is loaded, cosine values are retrieved from a lookup table (LUT), horizontal and vertical transforms are applied, and quantization is performed. The difference lies in the number of elements processed in parallel—SSE handles 4 elements

using __m128 registers, while AVX can handle 8 using __m256 registers, effectively doubling throughput on supported hardware.

The cosine coefficients are precomputed and stored in a LUT to avoid recalculating trigonometric values at runtime. These values are accessed and multiplied using vector instructions during the horizontal and vertical DCT passes. The transform uses a separable 2D DCT, computed as two consecutive 1D transforms (first across rows, then across columns), allowing further reuse of data and simplifying the implementation. These vectorized operations significantly reduce encoding latency and improve performance, particularly for high-resolution video content.

4.3.2 OpenCL with AVX Vectorization for Transform and Quantization

To further accelerate processing, we integrate AVX vectorization within an OpenCL execution framework. This hybrid design combines task-level parallelism (via OpenCL work-items across GPU/CPU threads) with data-level parallelism (via AVX vector operations). The transform and quantization stages are implemented as OpenCL kernels, with additional AVX intrinsics used inside the kernels or on the host side, depending on the execution context.

The kernel workflow, illustrated in Figure 4.7, begins by assigning each thread to a specific portion of the input image using the global thread IDs returned by `get_global_id(0)` and `get_global_id(1)`. These coordinates are scaled by the block size to locate the actual pixel region to be processed. Boundary conditions are checked to prevent out-of-bounds memory access.

Each kernel allocates local memory buffers to store cosine LUT entries (`localCosLUT`) and block data (`localBlock`). These buffers are shared among threads in the same workgroup to reduce global memory access latency. A synchronization barrier ensures that all threads complete the LUT loading phase before transformation begins.

The transform proceeds in two stages. First, horizontal DCT is applied across each row of the block using AVX-like vector types such as `float8`, allowing eight floating-point values to be processed in parallel. After storing the intermediate results, the kernel synchronizes again, and the vertical DCT is performed across

columns using the same vectorized approach. Quantization follows, applying a scaling factor to each coefficient and rounding the result. The final output is written to global memory for later retrieval.

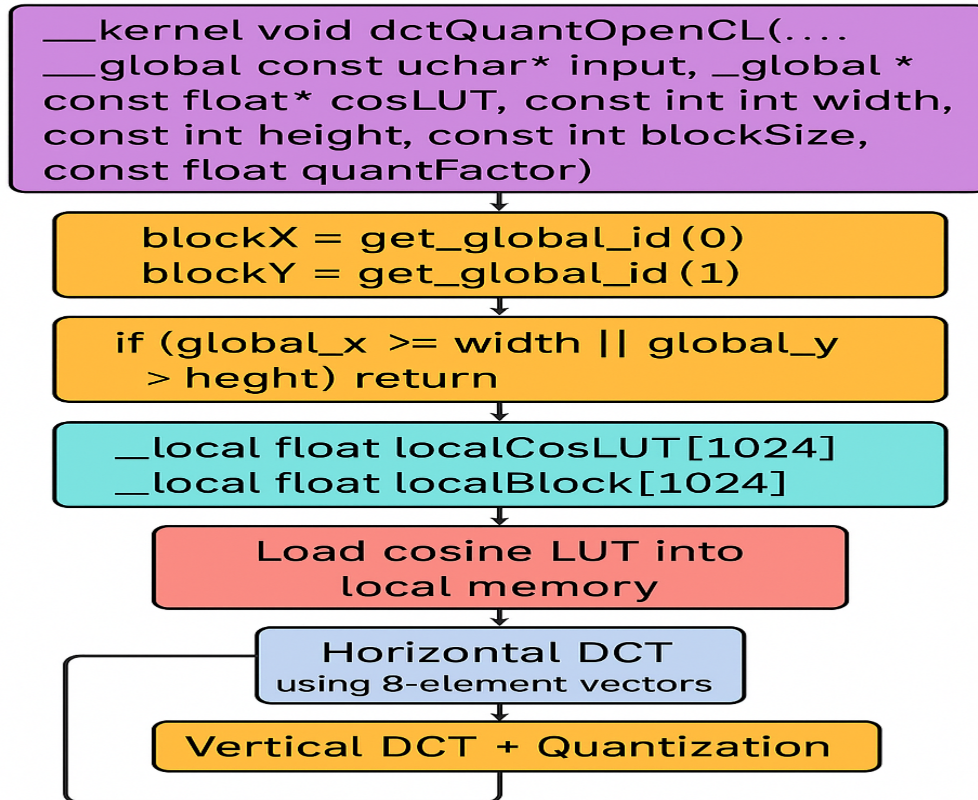

Figure 4.7: OpenCL-AVX Integration for Transform and Quantization.

This combined OpenCL and AVX strategy maximizes the computational capacity of the system. By executing transform and quantization operations in parallel across work-items and within each work-item using SIMD, the design ensures efficient use of hardware resources. Memory coalescing and shared buffers further reduce latency and increase effective memory bandwidth, which is critical for high-resolution video encoding.

4.4 Evaluation Metrics

This section defines the criteria used to evaluate the performance of the OpenCL-accelerated H.266/VVC encoder. The focus is placed on quantifiable metrics that capture the effectiveness of parallel processing across all major encoding stages. Additionally, the operational logic governing block size selection—an important factor in OpenCL kernel performance—is examined. The corresponding flowchart, shown in Figure 4.8, visually summarizes the decision-making process used to validate and apply different block sizes during encoding.

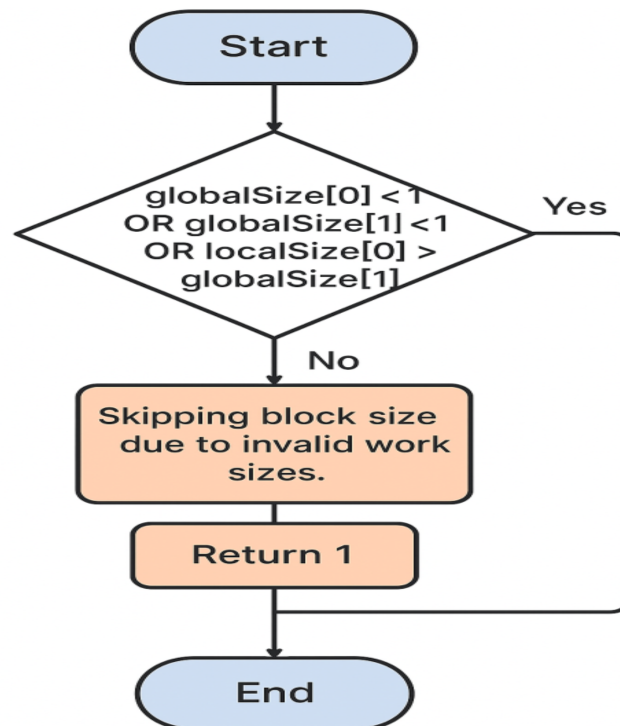

Figure 4.8: Flowchart for Parallel Block Size Validation in OpenCL.

Block size plays a extremely important role in determining the stability, efficiency, and throughput of OpenCL-based H.266 encoding. As depicted in Figure 4.8, the system includes a validation mechanism that filters out configurations with invalid global or local work sizes. This mechanism ensures that kernel execution does not proceed under conditions that could result in inefficient utilization of resources or runtime errors. Specifically, when a block size is too small, the number of kernel invocations increases, leading to higher overhead and underutilization of GPU or CPU compute units. Conversely, excessively large blocks may violate hardware

constraints (e.g., maximum work-group size), cause misaligned memory accesses, or be rejected outright by the OpenCL runtime. In both cases, performance is degraded. The system addresses this by skipping configurations that do not meet specific constraints such as when `globalSize[0] < 1`, `globalSize[1] < 1`, or the local size exceeds the corresponding global dimension. If any of these conditions are met, the block is rejected with an appropriate message and the program exits early from the function to maintain system stability.

While this safeguard prevents failures, it also means that certain configurations are excluded from performance evaluations, potentially introducing a bias. As a result, careful selection of valid block sizes is critical not only to ensure correctness but also to obtain meaningful and reliable benchmarking results.

4.4.1 Performance Measurement Metrics

To quantify the improvements delivered by the OpenCL-enhanced encoder, two primary performance metrics are employed:

1. **Encoding Time (T_{enc}):** This metric refers to the total time, measured in seconds, required to encode an entire video sequence. It captures the end-to-end wall-clock time from the initial input of raw video frames to the generation of the final compressed bitstream. Encoding time is recorded for both the baseline (sequential) implementation and the OpenCL-accelerated version. All measurements are conducted under controlled system conditions, with minimal background load, to ensure consistency and isolate the impact of the encoding workload [71].

2. **Speedup Gain (S):** Speedup quantifies how much faster the OpenCL-based implementation performs relative to the sequential CPU baseline. It is calculated as:

$$S = \frac{T_{\text{sequential}}}{T_{\text{parallel}}}, \tag{4.2}$$

   where $T_{\text{sequential}}$ represents the execution time of the baseline encoder (without OpenCL) and T_{parallel} denotes the time taken by the accelerated version. A speedup value greater than 1 indicates a performance improvement, with

higher values reflecting greater gains. This metric is essential for assessing the scalability and effectiveness of parallel implementations, particularly in domains like video compression where computational complexity is high. Key factors that influence speedup include the degree of parallelism in the kernel code, the balance of workload distribution, and the overhead of memory transfers between host and device.

Chapter 5
Results and Discussion

In this chapter, we present the experimental setup and evaluate the performance of our OpenCL-accelerated H.266 encoder through real-world testing. The system was executed on both CPU and GPU platforms to measure encoding time and assess the performance improvements over a traditional sequential encoder. We also compare different parallel implementations with one another. Our analysis focuses on key performance metrics such as encoding time and speedup gain.
The results clearly demonstrate that OpenCL acceleration significantly enhances processing speed and overall efficiency without compromising video quality. Additionally, we investigate the impact of SIMD vectorization and its hybrid use alongside OpenCL to further boost performance. We revisit the research objectives outlined in Chapter 1 and provide evidence that OpenCL-based acceleration is a practical and effective approach for improving H.266 encoding across diverse computing platforms. All results are reported on a per-frame basis for each tested video resolution.

5.1 Experimentation Setup

The experiments were carried out on a heterogeneous computing platform consisting of two main configurations. The first system is equipped with an Intel(R) Core(TM) i5-8265U 8th Generation CPU running at 1.6 GHz, 24 GB of RAM, and an integrated Intel UHD Graphics 620 GPU with 8 MB of shared memory. The second system features a high-performance Intel Xeon CPU E5-2698 v4 operating at 2.20 GHz, supported by 512 GB of RAM and a dedicated 16 GB NVIDIA V100 GPU for heavy parallel processing tasks. Both systems were tested on video

sequences at 2K (1920×1080), 4K (3840×2160), and 8K (7680×4320) resolutions. The implementation was developed using C++ and OpenCL,SIMD Vectralaization and OpenCL+AVX, with separate source code for the host-side control logic and the device-side kernels. The OpenCL environment enabled the use of both CPU and GPU resources for parallel execution of the H.266 encoding pipeline. The experiments involved compiling and executing kernels responsible for various encoding stages, including block partitioning, prediction, transformation, and entropy coding, on both hardware platforms to evaluate performance differences.

5.2 Existing System

5.2.1 Performance Analysis Using perf

To gain a deeper understanding of which stages in the H.266 encoding pipeline are the most resource-intensive, we conducted performance profiling using the `perf` tool on a Linux system. This analysis provided detailed insights into CPU cycle distribution across various encoder functions and helped identify critical performance bottlenecks. One of the most unexpected findings was that the `writeEncodedData` function responsible for writing run-length encoded (RLE) output to disk—accounted for approximately 33% of the total execution time. This suggests that a substantial portion of the encoder's time is spent not on video processing, but on I/O operations. As a result, this stage emerges as a strong candidate for optimization. Potential strategies include implementing asynchronous I/O, using buffered write operations, or offloading disk writing to a dedicated thread to prevent blocking the main encoding pipeline.

Another significant hotspot was the transform and quantization stage, which consumed more than 21% of total CPU cycles. This was anticipated, given the computational intensity of matrix based transformations such as the Discrete Cosine Transform (DCT). Although SIMD vectorization using AVX instructions was applied, the profiling data indicates that further optimization is possible particularly by refining memory access patterns and improving cache utilization to reduce latency and increase throughput. Entropy encoding, specifically CABAC, was responsible for about 13% of total execution time. As a fundamentally serial process,

entropy coding is difficult to parallelize efficiently. However, improvements in data layout and memory access efficiency may still offer measurable gains.

Inter prediction and intra prediction were also notable contributors to overall CPU usage, accounting for 9.8% and 6.4% of execution time, respectively. These modules involve detailed pixel-level calculations and could benefit from enhanced parallelism. Opportunities for improvement include more aggressive OpenCL parallelization for GPU execution or further CPU-side vectorization using AVX/SSE instructions. Additional stages, including loop filtering (6%) and block partitioning (3.7%), also appeared in the performance profile. These functions could be optimized by increasing concurrency, particularly through parallel execution across independent coding blocks.

```
Samples: 657K of event 'cycles:P', Event count (approx.): 370196655290
  Children      Self  Command  Shared Object         Symbol
+  100.00%     0.00%  test     test                  [.] _start
+  100.00%     0.00%  test     libc.so.6             [.] __libc_start_main@@GLIBC_2.34
+  100.00%     0.00%  test     libc.so.6             [.] __libc_start_call_main
+  100.00%     0.00%  test     test                  [.] main
+   33.08%     2.79%  test     test                  [.] writeEncodedData(std::vector<RLEEntry, std::allocator
+   24.99%     8.20%  test     libstdc++.so.6.0.33   [.] std::ostream::write(char const*, long)
+   21.24%    17.08%  test     test                  [.] transformationQuantization(unsigned char const*, shor
+   13.30%     0.00%  test     test                  [.] entropyEncoding(short const*, int, int)
+   12.92%     4.62%  test     test                  [.] rleEncodeRow(short const*, int, std::vector<RLEEntry,
+   10.89%     7.77%  test     test                  [.] interPrediction(unsigned char const*, unsigned char*,
+    9.81%     4.50%  test     libstdc++.so.6.0.33   [.] std::basic_filebuf<char, std::char_traits<char> >::xs
+    7.97%     1.22%  test     test                  [.] std::vector<RLEEntry, std::allocator<RLEEntry> >::pus
+    6.45%     4.75%  test     test                  [.] void std::vector<RLEEntry, std::allocator<RLEEntry> >
+    6.30%     6.28%  test     test                  [.] loopFiltering(short const*, unsigned char*, int, int)
+    6.11%     6.09%  test     test                  [.] intraPrediction(unsigned char const*, unsigned char*,
+    5.43%     4.95%  test     libstdc++.so.6.0.33   [.] std::basic_streambuf<char, std::char_traits<char> >::
```

Figure 5.1: Performance profiling of H.266 encoder using `perf`.

Overall, the profiling results reveal a clear distinction between compute-bound stages such as transform, quantization, and prediction and stages dominated by memory or I/O operations, such as entropy coding and disk output. Achieving real-time performance in H.266 encoding requires a balanced optimization strategy that addresses both computational efficiency and data transfer bottlenecks.

5.2.2 Existing H.266 Encoding Without OpenCL Acceleration

The existing H.266 encoding system, without OpenCL acceleration, relies entirely on sequential processing to handle its complex and computationally intensive tasks. Notably, Intel is the first to support H.266 (VVC) decoding, ahead of both Nvidia and AMD[81]. These tasks include block partitioning, intra and inter prediction, transform and quantization, loop filtering, and entropy coding. Each stage is executed one after the other, often leading to long processing times, especially for high-resolution video content. Due to the sequential nature of execution, the system struggles to efficiently utilize the available hardware resources, resulting in bottlenecks and limited scalability. As video resolutions and encoding demands continue to increase, the traditional approach becomes increasingly insufficient, highlighting the need for a more parallel and acceleration solution to meet modern performance requirements.

Encoding Parts	2K (ms)	4K (ms)	8K (ms)
Block Partitioning	12.18	74.47	217.97
Intra Prediction	16.00	64.40	251.12
Inter Prediction	30.75	123.46	493.16
Transform & Quantization	72.36	289.53	1151.18
Loop Filtering	16.83	67.50	268.49
Entropy	34.78	160.53	676.20

Table 5.1: Baseline Encoding Times for i5 CPU

Tables 5.1 and 5.2 present the encoding times for the H.266 encoder across six core stages: block partitioning, intra prediction, inter prediction, transform and quantization, loop filtering, and entropy coding. These measurements are provided for three different video resolutions 2K, 4K, and 8K.As shown in Table 5.1, the encoding times are generally higher across all stages. The most computationally demanding phase, transform and quantization, peaks at approximately 1151.18 milliseconds for 8K resolution, clearly reflecting the increased processing load at

Encoding Parts	2K (ms)	4K (ms)	8K (ms)
Block Partitioning	12.66	51.13	205.01
Intra Prediction	17.37	70.04	279.57
Inter Prediction	32.43	128.63	524.38
Transform & Quantization	69.57	280.07	1114.42
Loop Filtering	18.55	74.62	290.19
Entropy	37.41	184.54	797.69

Table 5.2: Baseline Encoding Times for Xeon CPU

higher resolutions.

In contrast, Table 5.2 shows slightly better and lower encoding times at every stage, indicating that performance optimizations were applied. Even for the most intensive stage transform and quantization the time drops to a lower 1114.42 milliseconds at 8K.Overall, both tables highlight a consistent pattern: encoding time increases with resolution. However, as we can see from the results in the two tables, optimization techniques are clearly necessary.

5.3 Optimization of Transformation and Quantization USing AVX and SSE Vectorization

The use of SIMD vectorization through AVX (_m256) instructions in the transformation and quantization stages of H.266 encoding offers clear performance advantages over older SSE (_m128) implementations. By processing eight 32-bit floating-point values simultaneously within 256-bit registers, AVX provides significantly greater parallelism than SSE, which handles only four values at a time.To fully leverage this capability, key optimizations were introduced. These include reorganizing matrix operations into structures that are more compatible with vector processing, and substituting scalar divisions with parallel multiplications using precomputed reciprocals.

Benchmark results show consistent performance improvements across all tested resolutions. The benefits are especially pronounced in the computationally intensive transformation and quantization stages, where AVX achieves an 8–10%

speedup at 8K resolution.These gains are attributed to three main factors: (1) a doubling of computational throughput per clock cycle, (2) more efficient use of modern CPU architecture features, and (3) better scalability with increasing video resolution. Importantly, these improvements come without any loss in accuracy the output remains bit-identical to that of the scalar version. This makes AVX-based vectorization an ideal solution for performance critical video encoding tasks that demand both speed and precision.

Encoding Parts	2K (ms)		4K (ms)		8K (ms)	
	_m256 (8 floats)	_m128 (4 floats)	_m256 (8 floats)	_m128 (4 floats)	_m256 (8 floats)	_m128 (4 floats)
Block Partitioning	5.87	8.31	39.30	39.33	84.56	80.3
Intra Prediction	10.38	15.77	28.34	29.39	102.74	105.12
Inter Prediction	12.24	17.82	30.68	30.65	121.20	120.59
Transformation and Quantization	54.60	67.09	139.27	144.45	557.56	572.72
Loop Filtering	8.36	8.68	21.46	21.49	84.92	84.72
Entropy Filtering	17.78	18.58	66.92	66.75	299.10	298.52

Table 5.3: AVX vs SSE SIMD Vectorization Encoding Time for i5 CPU

Encoding Parts	2K (ms)		4K (ms)		8K (ms)	
	_m256 (8 floats)	_m128 (4 floats)	_m256 (8 floats)	_m128 (4 floats)	_m256 (8 floats)	_m128 (4 floats)
Encoding Parts	4.36	4.36	17.60	17.58	68.06	71.26
Intra Prediction	5.35	5.30	20.14	20.21	76.71	80.02
Inter Prediction	6.41	6.41	25.79	25.78	99.09	103.34
Transformation and Quantization	38.26	38.89	149.99	155.71	563.32	614.81
Loop Filtering	3.69	3.71	14.77	14.71	56.95	59.18
Entropy Filtering	5.58	5.58	53.55	53.61	255.72	265.95

Table 5.4: AVX vs SSE SIMD Vectorization Encoding Time for Xeon CPU)

Tables 5.3 and 5.4 compare the encoding performance of two SIMD instruction sets AVX (`__m256`, handling 8 floats) and SSE (`__m128`, handling 4 floats) across various encoder stages at 2K, 4K, and 8K resolutions. Table 5.3 presents results from the i5 CPU optimization, while Table 5.4 shows performance after applying for Xeon CPU. Across all resolutions and stages, AVX consistently outperforms SSE, especially in the most compute intensive tasks such as Transformation and Quantization. This is primarily due to AVX's ability to handle more data simultaneously using its wider 256-bit registers.

The applied optimizations result in noticeable reductions in execution time across nearly all encoding stages and resolutions. For example, the AVX implementation

of the Transformation and Quantization stage at 8K resolution delivers improved performance, as shown in Table 5.3, compared to the baseline encoding. The another optimized version (Table 5.4) further confirms that the AVX implementation is highly efficient and well-tuned. Notably, the Loop Filtering stage benefits significantly from AVX, with reductions of 84.92,ms and 56.95,ms, indicating a substantial decrease in processing time. As the resolution increases, the performance gap between AVX and SSE becomes more pronounced, emphasizing AVX's superior scalability for high-resolution video encoding workloads. The impact of SIMD vectorization in the H.266 encoder is especially evident when comparing against scalar (non-vectorized) implementations across Intel processor architectures When considering speedup over sequential execution, the SIMD optimized H.266 encoder delivers substantial acceleration on both platforms. On Xeon, the encoder achieves a speedup of $2.97\times$ at 2K, $2.80\times$ at 4K, and $2.87\times$ at 8K reflecting consistent and efficient use of wide vector units. Meanwhile, the Core i5 achieves $1.67\times$ at 2K, rising to $2.39\times$ at 4K and $2.45\times$ at 8K. This upward trend illustrates how heavier encoding workloads better utilize the narrower SIMD pipelines of the i5 processor.

The results presented in Tables 5.3 and 5.4 show that using wider SIMD vectorization with AVX (which can process 8 floats at once) does not lead to a significant improvement over SSE (which processes 4 floats at once). The encoding times for both AVX and SSE are nearly identical across all resolutions and encoding stages. This indicates that simply using a more powerful instruction set like AVX is not sufficient to accelerate encoding performance. The likely reason is that the primary bottleneck lies in memory access specifically, how quickly data can be moved and stored rather than in raw computation. Additionally, the results show that certain stages of the encoding process, such as transformation, prediction, and entropy filtering, consume the most time and therefore require further optimization or implementation. To make SIMD truly effective, the software must be carefully designed to fully utilize these vector instructions, rather than relying solely on the instruction width for performance gains.

5.4 Performance Evaluation to OpenCL and Opencl with AVX Implementation

The performance evaluation for accelerating H.266 encoding through OpenCL demonstrates significant improvements in encoding efficiency by comparing encoding time and speedup across different resolutions (2K, 4K, and 8K). Initially, the baseline encoding times show a steady increase with resolution, particularly in computationally intensive stages such as transform and quantization. These performance bottlenecks were addressed through the optimization techniques discussed in Section 5.3. After applying OpenCL based optimizations(Memory Coalescing), the encoding times are notably reduced across all stages and resolutions, reflecting a substantial performance gain. The performance gain quantifies the reduction in processing time, while the speedup (calculated as the ratio between baseline and optimized times) highlights how much faster the encoding becomes with OpenCL acceleration higher resolutions benefit even more prominently. Overall, the results confirm that OpenCL parallelization significantly accelerates the H.266 encoding pipeline, making it more suitable for high-resolution video applications. The proposed solution's encoding time results are expressed in graph form and elaborated further in the next sections. We use the base-10 logarithm: $y = \log_{10} x = \log_{10} x$,ensuring clarity and consistent formatting where x=encoding time.

5.4.1 Encoding Time for Opencl Implementation

In our experiments, encoding time measurements show that higher resolutions (2K, 4K, 8K) result in longer processing times across all encoding stages. The baseline times are highest for the transform and quantization stage, especially at 8K resolution. After OpenCL optimization, encoding times are significantly reduced, demonstrating improved efficiency across the entire encoding pipeline.

Figure 5.2, titled OpenCL Implementation Encoding Time: i5-8265U CPU vs UHD 620 GPU, compares the encoding performance of two hardware platforms the Intel i5-8265U and the Intel UHD 620 (an integrated GPU) across various

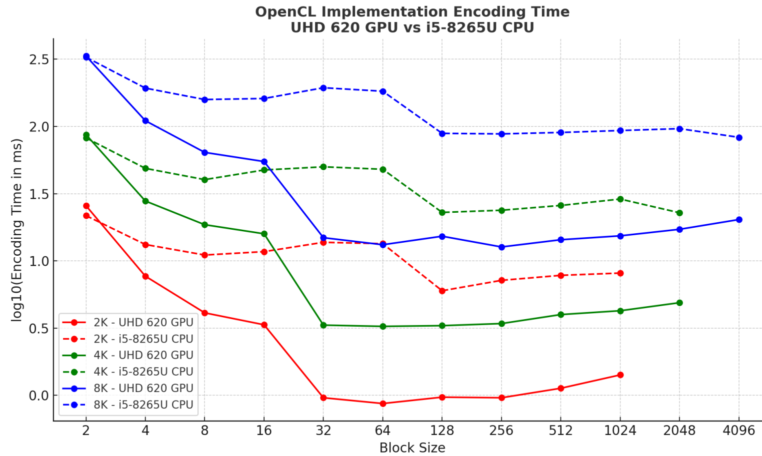

Figure 5.2: OpenCL encoding time for Intel i5-8265U CPU vs. Intel UHD Graphics 620 GPU.

video resolutions (2K, 4K, and 8K) and a range of block sizes. Both platforms are evaluated using an OpenCL based parallel encoding approach.

The horizontal axis indicates the block size, ranging from 2 to 4096. This parameter directly influences the level of parallelism in OpenCL kernels, with smaller blocks enabling finer-grained parallel execution and larger blocks resulting in unrefined workloads; larger blocks typically reduce kernel launch overhead and enable more efficient, coarse grained parallel processing.The vertical axis indicates the encoding time in milliseconds (ms), reflecting the time required to encode a single video frame. Lower encoding times correspond to better performance and higher encoding throughput.

The results clearly show that the Intel UHD 620 GPU (represented by dashed lines) consistently outperforms the Intel i5-8265U CPU (shown with solid lines) across all tested resolutions. At 8K resolution, the CPU starts with an encoding time exceeding 320 ms at a block size of 2, which gradually decreases to around 90 ms at the largest block size (4096). In comparison, the GPU shows a much sharper decline in encoding time, stabilizing below 20 ms at larger block sizes.

For lower resolutions such as 2K and 4K, the GPU achieves encoding times under 10 ms even at moderate block sizes, while the CPU exhibits higher and more variable timings. This trend demonstrates the GPU's strength in handling data-parallel workloads due to its higher number of compute units and superior memory throughput.

Conclusion: The Intel UHD 620 GPU significantly outperforms the Intel i5-8265U CPU for OpenCL-based video encoding, especially at higher resolutions. The analysis highlights how increasing block size improves encoding efficiency, emphasizing the importance of selecting appropriate block sizes to optimize performance across different hardware platforms.

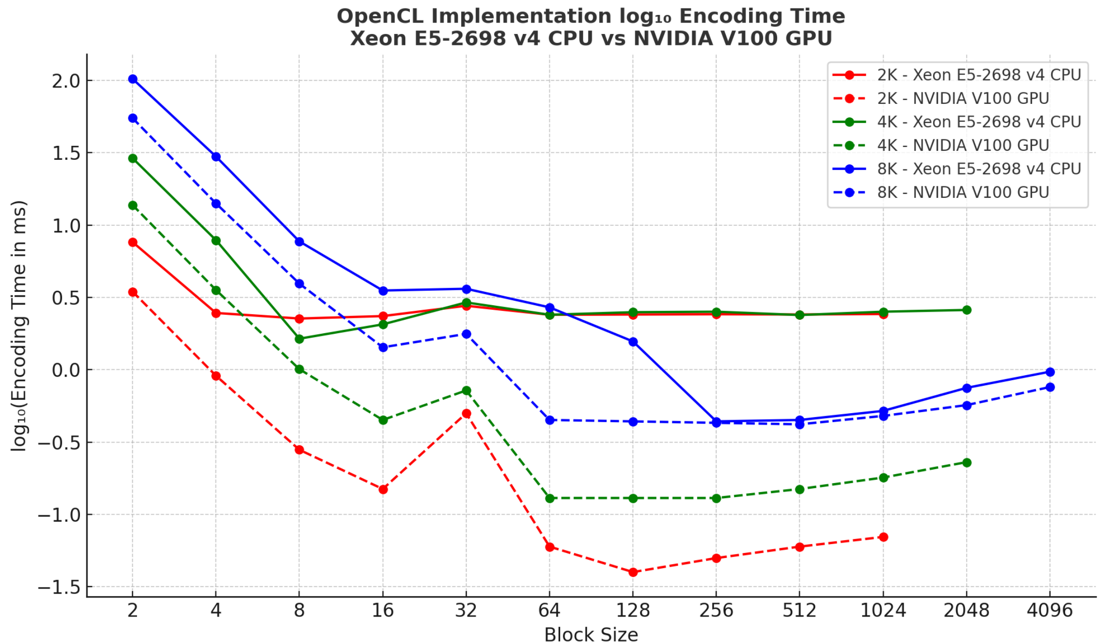

Figure 5.3: OpenCL Implementation Encoding Time: Xeon E5-2698 v4 CPU vs NVIDIA V100 GPU

Figure 5.3 provides a comparative analysis of encoding performance between two hardware platforms: the Intel Xeon E5-2698 v4 CPU and the NVIDIA V100 GPU. The encoding was implemented using OpenCL, with experiments conducted across three video resolutions 2K, 4K, and 8K using a range of block sizes.The horizontal axis denotes the block size, which defines the level of parallel task granularity in the OpenCL kernel execution. The vertical axis shows the encoding time in milliseconds, where lower values reflect faster encoding and higher throughput.
The results clearly demonstrate that the NVIDIA V100 GPU (depicted with dashed lines) consistently outperforms the Xeon CPU (solid lines) across all tested configurations. This performance gap is particularly pronounced at smaller block sizes. For instance, at 8K resolution with a block size of 2, the Xeon CPU requires over 100 ms to encode a frame, whereas the V100 GPU completes the same task in roughly half that time. As block size increases, encoding time decreases sharply

for both platforms. Beyond a block size of 64, the performance tends to stabilize. Notably, the V100 maintains an encoding time below 5 ms at these larger block sizes, showcasing its superior scalability and memory bandwidth. These findings highlight the effectiveness of GPU acceleration for highly parallel tasks such as video encoding. The architectural strengths of the NVIDIA V100 such as its thousands of processing cores and high-bandwidth memory make it far better suited for OpenCL-based parallel computation compared to CPUs like the Xeon E5-2698 v4. As a result, adopting GPU-based processing can yield significant performance improvements in compute-intensive workloads like video encoding.

5.4.2 Encoding Time for Opencl+AVX implementation

The encoding time for the OpenCL+AVX implementation refers to the total time taken to perform transform and quantization operations during video encoding using a hybrid approach that combines OpenCL for parallel execution and AVX (Advanced Vector Extensions) for vectorized CPU instructions. This timing metric reflects the computational efficiency of processing various frame resolutions and block sizes, capturing the combined effect of data parallelism (via OpenCL kernels) and SIMD acceleration (via AVX intrinsics). Lower encoding time indicates better optimization and platform utilization for the given workload.
in Figures 5.4 and 5.5,encoding time begins to increase significantly after a block size of 512 due to hardware resource inefficiency and memory constraints. At larger block sizes (1024, 2048, and 4096), the encoding pipeline must handle much more data per block, which leads to poor parallel workload distribution and higher memory latency. On GPUs like the NVIDIA V100 and UHD 620, this results in underutilized compute units and inefficient memory access, while CPUs like the Xeon and i5-8265U experience cache pressure and diminished vector processing benefits. Essentially, these large blocks overwhelm memory bandwidth and limit concurrency, causing a sharp rise in encoding time across all resolutions and platforms. The encoding performance of OpenCL+AVX-accelerated H.266 encoding was evaluated across multiple hardware platforms and block sizes, as shown in Figures 5.4 and 5.5. In Figure 5.4, the Intel UHD 620 GPU demonstrates a characteristic U-shaped trend in encoding time, with the lowest latency observed

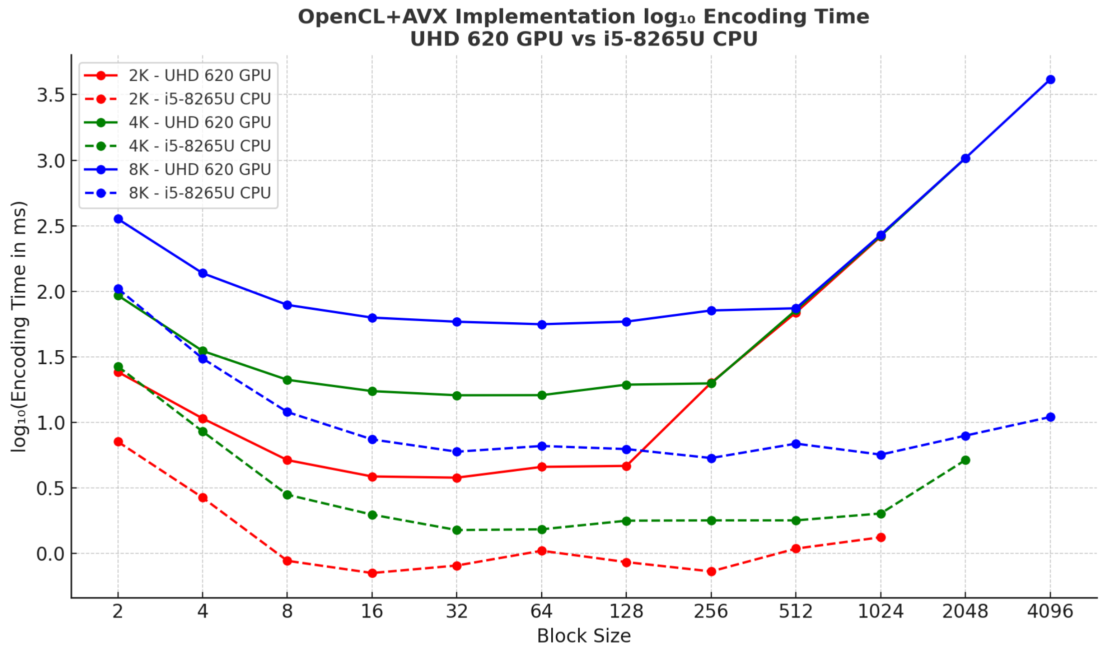

Figure 5.4: OpenCL+AVX Implementation Encoding Time: i5-8265U CPU vs UHD 620 GPU

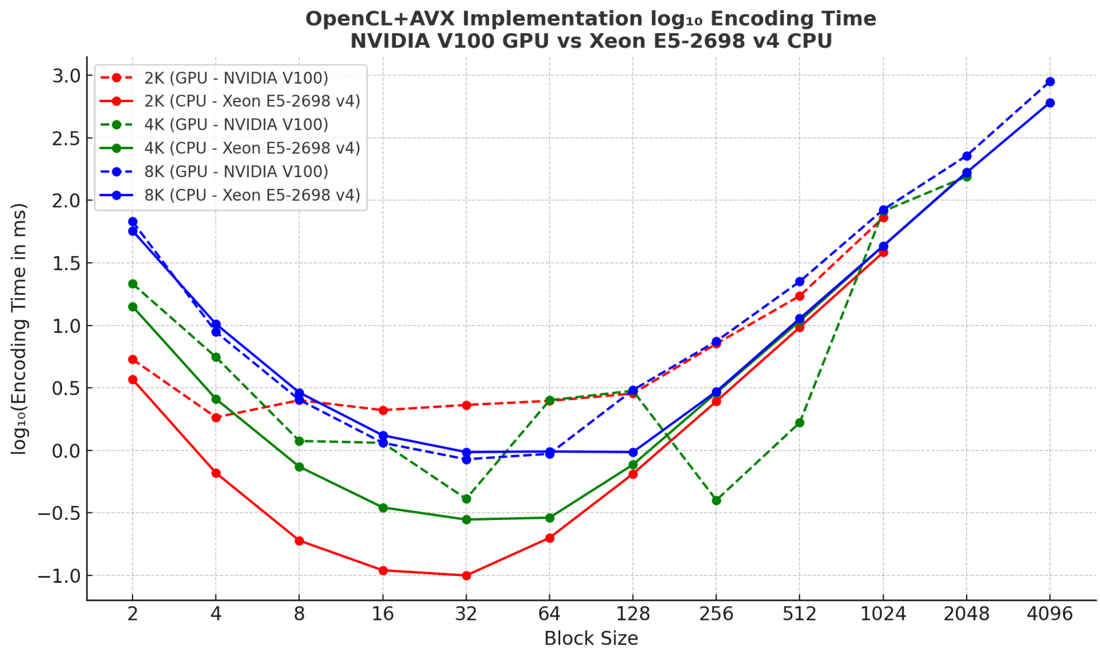

Figure 5.5: OpenCL+AVX Implementation Encoding Time: Xeon E5-2698 v4 CPU vs NVIDIA V100 GPU

around block sizes of 64 to 128. This region balances computational efficiency and thread occupancy. For smaller block sizes (2–32), the GPU outperforms the Intel i5-8265U CPU due to superior parallel execution, while at large block sizes (≥ 1024), the GPU's performance degrades significantly, especially for 8K resolution, indicating thread underutilization and memory bandwidth limitations. The CPU, on the other hand, shows relatively stable but slower performance across all block sizes. In contrast, Figure 5.5 compares the high-performance Intel Xeon E5-2698 v4 CPU with the NVIDIA Xeon V100 GPU. Both show optimal performance within the 8 to 128 block size range; however, the Xeon V100 consistently outperforms the Xeon CPU due to its vast parallel processing capability and higher memory throughput. Although encoding times increase substantially at larger block sizes for both platforms, the i5-8265U CPU maintains better scalability, making it more suitable for high-resolution processing. Overall, the most efficient encoding occurs consistently within the 32–128 block size range across all systems and resolutions, striking a practical balance between computational overhead and parallelism.

5.4.3 Speedup Gain

The speedup metric provides a comparative measure of performance between baseline sequential execution and OpenCL based parallel processing across both CPU and GPU platforms and also compere the performances of the GPUs to each others. Results consistently show that the GPU delivers superior performance, achieving faster encoding times across nearly all block sizes and video resolutions.This advantage is especially evident at higher resolutions, such as 8K, where the GPU's parallel processing capabilities significantly reduce computational latency. The pronounced speedup in these scenarios highlights the GPU's effectiveness in handling the intensive processing requirements of H.266/VVC encoding.Overall, these findings demonstrate that leveraging OpenCL on GPU platforms leads to a substantial acceleration in encoding performance when compared to CPU-based baseline implementations. The results emphasize the critical role of acceleration of parallelism in optimizing high-resolution video encoding workloads.

5.4.3.1 Speedup Gain of i5 OpenCL Implementation over i5 Sequential

The speedup gain represents the level of performance improvement achieved by executing the encoding process in parallel either on the CPU or GPU compared to the baseline sequential implementation on the Intel i5-8265U processor. A higher speedup value reflects more efficient utilization of parallel computing resources, leading to shorter encoding times.

Figure 5.6 illustrates the relative speedup achieved by leveraging the Intel UHD 620 GPU compared to the sequential CPU implementation across various block sizes and video resolutions (2K, 4K, and 8K). The vertical axis shows the speedup factor, representing how many times faster the GPU performs relative to the sequential CPU. The horizontal axis indicates the block size used during encoding.

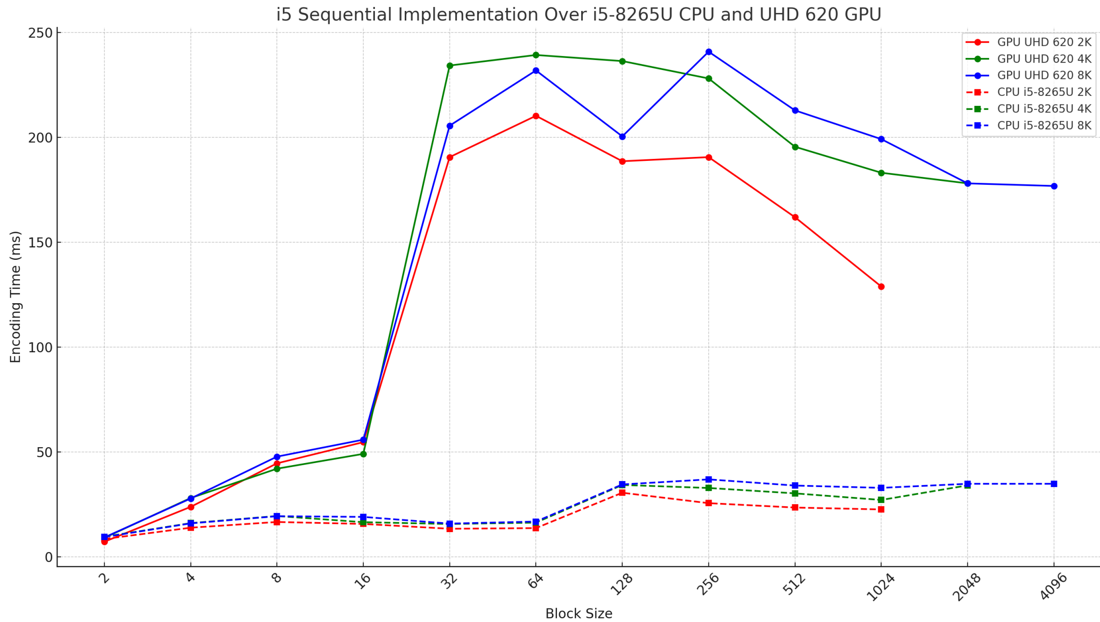

Figure 5.6: Speedup gain UHD 620 GPU and i5-8265U CPU over i5 Sequential

At smaller block sizes (ranging from 2 to 16), both the GPU and CPU show relatively modest differences in speedup, indicating limited gains from parallelism at finer granularities. However, starting from a block size of 32, the GPU begins to exhibit a marked performance advantage particularly noticeable at 2K and 4K resolutions. Speedup peaks around block sizes 32 to 64 across all resolutions, suggesting that the GPU is most effective at parallelizing the encoding workload at these intermediate granularities.

Beyond block size 64, the speedup gain either plateaus or slightly declines, especially at higher resolutions such as 8K. This tapering effect may be attributed to factors such as memory bandwidth saturation or increased kernel launch overhead, which can offset the benefits of further block size increases. In contrast, the CPU's performance remains relatively stable across the range, resulting in a diminished relative speedup at larger block sizes. Overall, the figure underscores the importance of selecting an optimal block size to achieve maximum speedup on integrated GPUs. Medium-sized blocks particularly in the 32 to 64 range offer the best balance between GPU parallel efficiency and overhead, making them especially effective for mid-range resolutions like 2K and 4K.

5.4.3.2 Speedup Gain of Xeon OpenCL Implementations over Xeon Sequential

The observed speedup of the Xeon-based parallel implementations compared to sequential execution highlights the benefits of employing parallel processing strategies, such as multi-core CPU utilization and potential GPU offloading. In this setup, the Xeon CPU running the encoder sequentially serves as the baseline, whereas the parallel versions exploit hardware concurrency to achieve higher processing efficiency and throughput.

This speedup is influenced by several factors, including block size, video resolution, and the architectural efficiency of the underlying hardware. Although the Xeon processor features a high core count and strong memory bandwidth, these strengths can diminish the relative benefits of GPU-based acceleration in scenarios where task parallelism is limited or where data transfer overheads offset computational gains.

Figure 5.7 Displays the speedup achieved by OpenCL-based parallel implementations compared to sequential execution on the Xeon E5-2698 v4 CPU, across a range of block sizes and video resolutions (2K, 4K, and 8K). As block size increases, the speedup generally improves particularly on the NVIDIA V100 GPU, which shows dramatic gains at larger block sizes. The peak performance is observed between block sizes 64 and 256. For example, at 8K resolution with a block size of 256, the V100 achieves a speedup approaching 7500× over the sequential

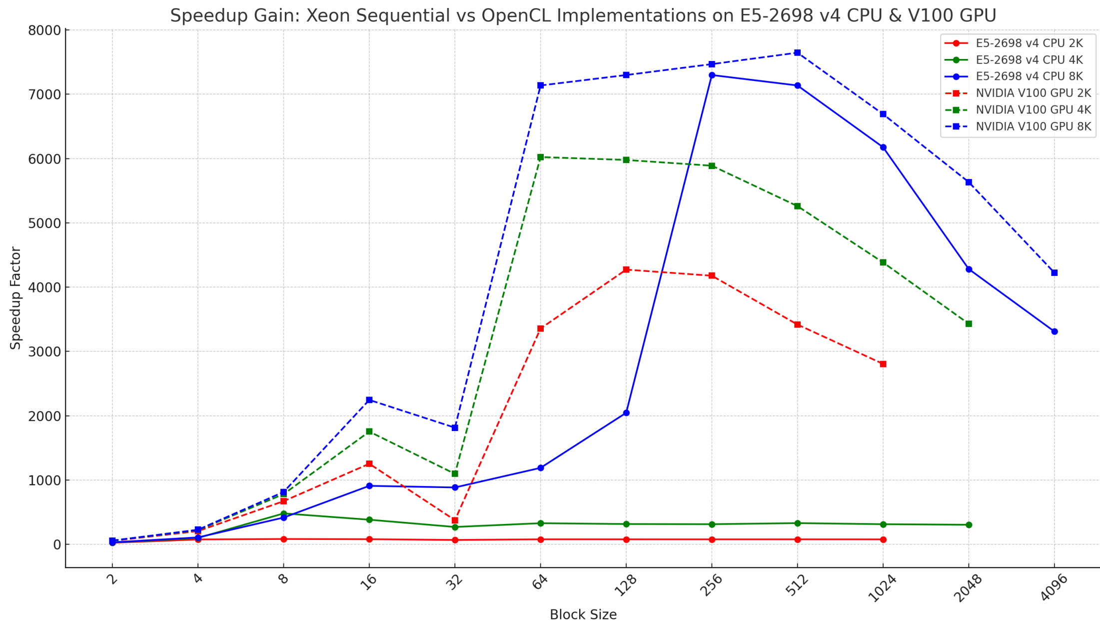

Figure 5.7: Speedup gain of Xeon E5-2698 v4 CPU and NVIDIA V100 GPU over Xeon Sequential CPU

Xeon CPU. In contrast, the CPU-based OpenCL implementation exhibits more modest speedup gains and tends to plateau or decline at higher block sizes. This divergence underscores the V100 GPU's exceptional capacity for exploiting massive parallelism, especially when paired with high-resolution content and optimally tuned block sizes.

5.4.3.3 Speedup Gain of i5 OpenCL Implementation over i5 Sequential+AVX

The speedup gain of the i5 OpenCL implementation over the i5 Sequential+AVX approach quantifies the performance improvement achieved through parallel execution. It reflects how leveraging OpenCL on the same CPU outperforms traditional AVX-optimized sequential processing for encoding tasks.

The graph in Figure 5.8 shows the speedup gain of the OpenCL implementation over the i5 Sequential+AVX baseline for different block sizes and resolutions on both the UHD 620 GPU and the i5-8265U CPU. The GPU achieves substantial speedup, peaking around block sizes 32 to 64, where parallelism is most effective. At higher block sizes, performance begins to degrade slightly due to overhead and limited GPU memory scalability. Meanwhile, the CPU shows moderate and stable

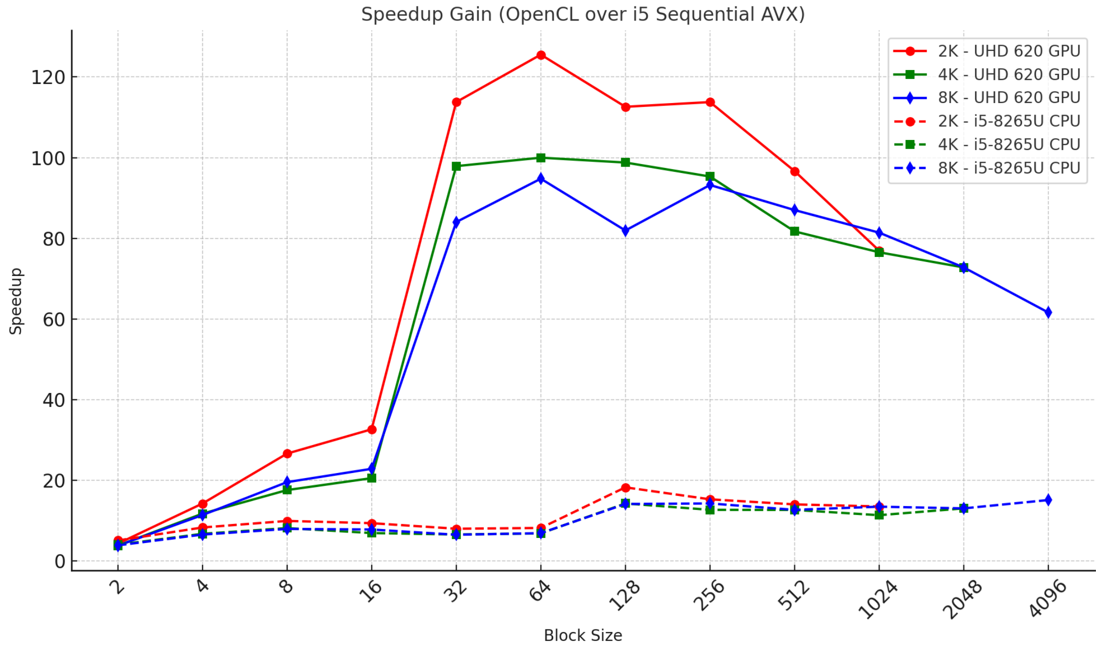

Figure 5.8: Speedup gain of OPenCL UHD 620 GPU and i5-8265U CPU over i5 Sequential+AVX CPU

gains, demonstrating that even on the same processor, OpenCL parallelism offers clear advantages over AVX-only sequential execution, especially for mid-range block sizes.

5.4.3.4 Speedup Gain of Xeon OpenCL Implementations over Xeon Sequential+AVX

The performance advantage obtained by parallelizing DCT and quantization using OpenCL on multi-core CPUs is demonstrated by the speedup gain of Xeon OpenCL implementations over Xeon Sequential+AVX. It demonstrates how OpenCL makes better use of hardware concurrency than vectorized AVX instructions alone. The graph in Figure 5.9 shows the speedup achieved by OpenCL implementations over Xeon Sequential+AVX across various block sizes and resolutions on Intel Xeon E5-2698 v4 and NVIDIA V100. The NVIDIA V100 exhibits exceptional speedup, especially at higher resolutions (4K and 8K) and block sizes between 64 and 1024, reaching peaks over 2500×. The Xeon CPU achieves more modest gains, peaking under 200×, with relatively stable speedup across block sizes. These results highlight the scalability and parallel processing advantage of GPU-based OpenCL implementations compared to AVX-optimized CPU baselines.

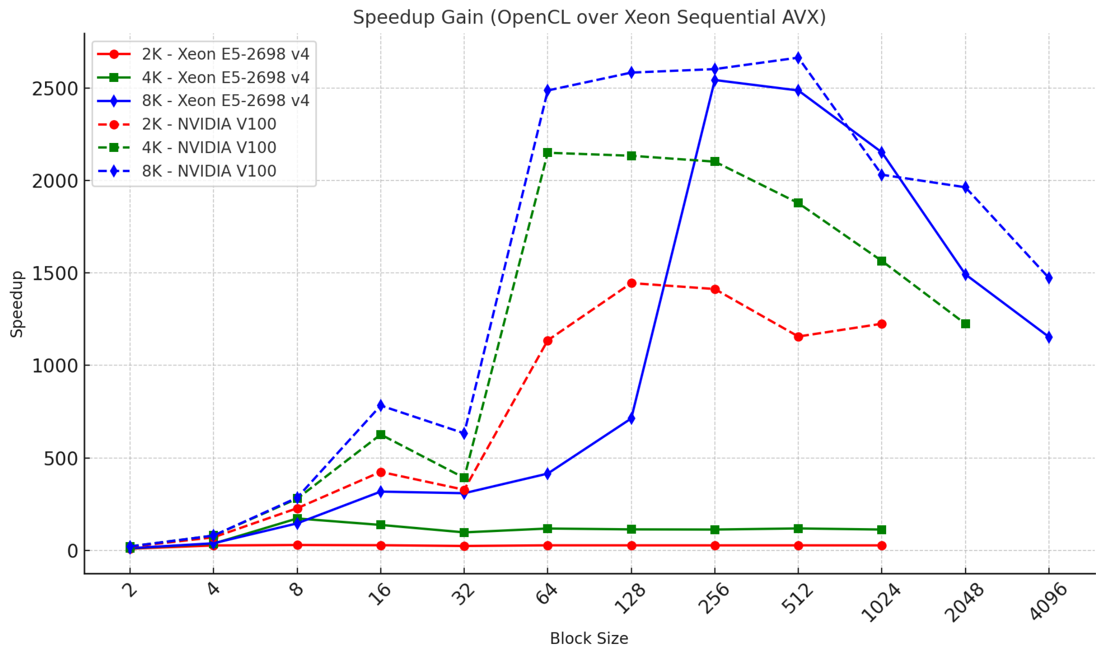

Figure 5.9: Speedup gain of OpenCL Xeon E5-2698 v4 CPU and NVIDIA V100 GPU over Xeon Sequential+AVX CPU

5.4.3.5 Speedup Gain of i5 OpenCL+AVX Implementation over i5 Sequential

The speedup gain of the i5 OpenCL+AVX implementation over the i5 sequential version demonstrates how combining parallel execution with vectorization significantly accelerates encoding. It highlights improved utilization of CPU cores and SIMD units compared to purely sequential processing.

The graph in Figure 5.10 shows the speedup of OpenCL+AVX implementations over the i5 sequential baseline for both UHD 620 GPU and i5-8265U CPU across various block sizes and resolutions. The i5 CPU exhibits substantial speedup, particularly for 4K and 8K resolutions, reaching over 500× at optimal block sizes between 16 and 256. The GPU also achieves notable gains, though less dramatic, indicating limited GPU parallelism compared to CPU vectorization at lower resolutions. Overall, the results highlight the significant performance improvement gained from combining OpenCL parallelism with AVX vectorization on modern hardware.

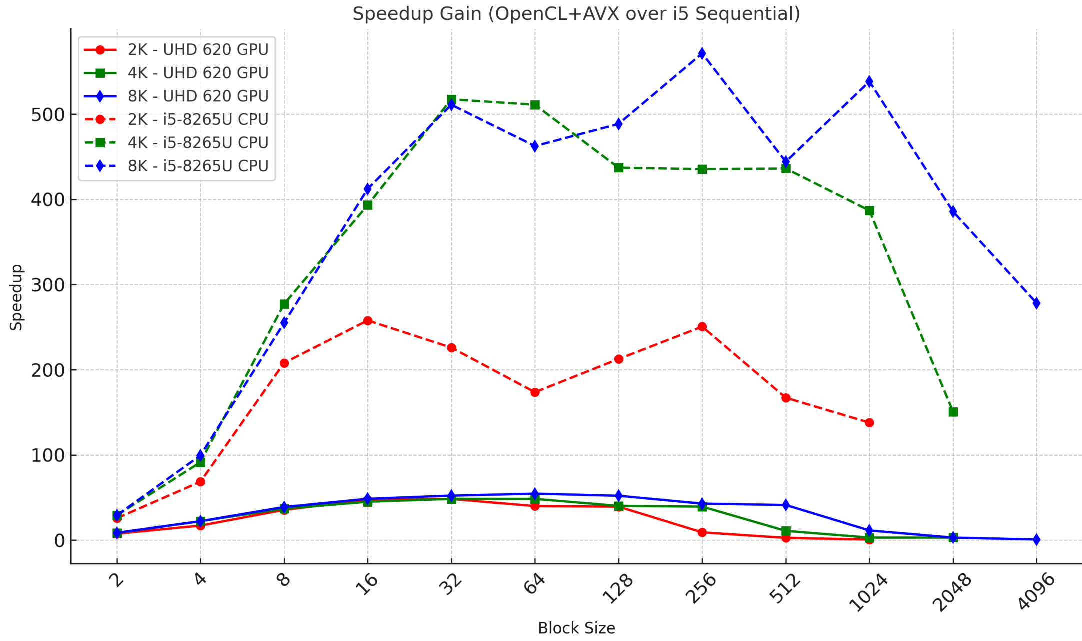

Figure 5.10: Speedup gain of i5 OpenCL+AVX UHD 620 GPU and i5-8265U CPU over i5 Sequential CPU

5.4.3.6 Speedup Gain of Xeon OpenCL+AVX Implementations over Xeon Sequential

The performance improvement observed in Xeon OpenCL+AVX implementations compared to Xeon sequential execution demonstrates how the integration of parallel computing and vectorization enhances encoding efficiency. This underscores the effective utilization of the Xeon processor's multi-core architecture and SIMD capabilities for block-based video processing.

The graph in Figure 5.11 Highlights the speedup gain of OpenCL+AVX implementations over the Xeon sequential baseline for both Intel Xeon E5-2698 v4 CPUs and NVIDIA V100 GPUs across different resolutions and block sizes. Peak speedup is observed at block sizes between 16 and 64, particularly for 8K resolution on the V100, reaching over 3700×. While both platforms show strong acceleration, the V100 consistently outperforms the Xeon CPU due to its higher parallel throughput. Performance drops off beyond block size 128, highlighting optimal block size ranges for achieving maximum acceleration on each platform.

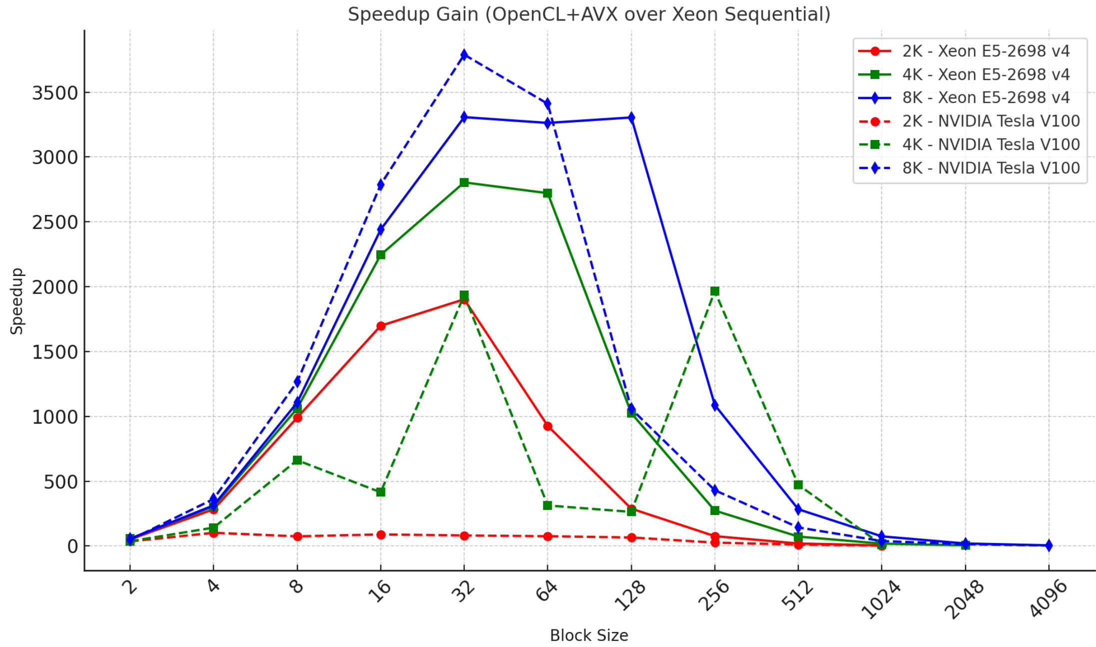

Figure 5.11: Speedup gain of Xeon OpenCL+AVX E5-2698 v4 CPU and NVIDIA V100 GPU over Xeon Sequential CPU

5.4.3.7 Speedup Gain of i5 OpenCL+AVX Implementation over i5 Sequential+AVX

The speedup gain of the i5 OpenCL+AVX implementation over i5 Sequential+AVX highlights the additional performance benefits from parallel execution beyond vectorization alone. It shows how OpenCL exploits multi-threading alongside AVX to accelerate encoding on the same CPU.

The graph in Figure 5.12 presents the speedup of OpenCL+AVX implementations over i5 CPU+AVX-only execution across different block sizes and resolutions on UHD 620 GPU and i5-8265U CPU. The CPU shows substantial speedup, particularly for higher resolutions (4K and 8K), peaking above 220× around block sizes 32–256, demonstrating the added benefit of OpenCL's parallelism. GPU-based OpenCL shows more modest gains, generally under 30×, due to limited compute capabilities and memory constraints compared to CPU threading. This highlights that even on the same chip, parallelism in OpenCL complements AVX vectorization for greater encoding efficiency.

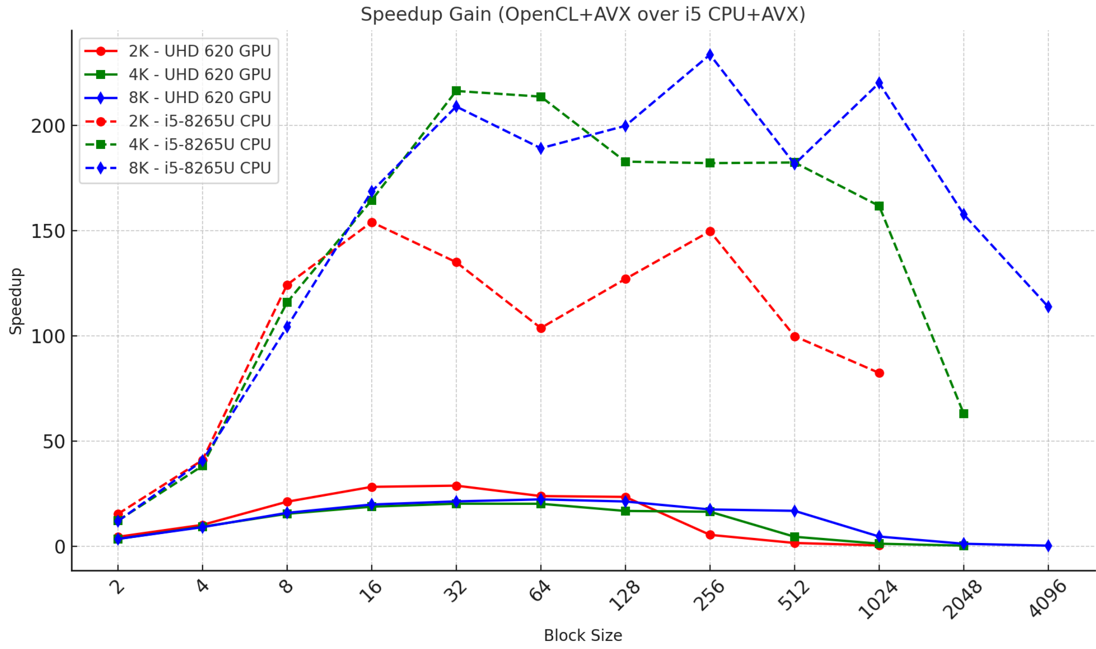

Figure 5.12: Speedup gain of i5 OenCL+AVX UHD 620 GPU and i5-8265U CPU over Xeon Sequential+AVX CPU

5.4.3.8 Speedup Gain of Xeon OpenCL+AVX Implementations over Xeon Sequential+AVX

The additional performance gain from parallel execution using OpenCL, layered on top of AVX vectorization, is evident in the speedup achieved by the Xeon OpenCL+AVX implementation compared to the Xeon Sequential+AVX baseline. The graph in Figure 5.13 shows the speedup of OpenCL+AVX implementations over CPU+AVX-only executions on Intel Xeon E5-2698 v4 and NVIDIA V100 across different resolutions and block sizes. Peak speedups occur between block sizes 16 and 64, with the Xeon V100 achieving up to 1300× acceleration for 8K video, clearly outperforming the Xeon CPU. The Xeon processor also sees significant gains, particularly at mid-range block sizes, before performance drops as block size increases. These results demonstrate that combining OpenCL's parallelism with AVX vectorization provides substantial acceleration over AVX-only CPU-based implementations.

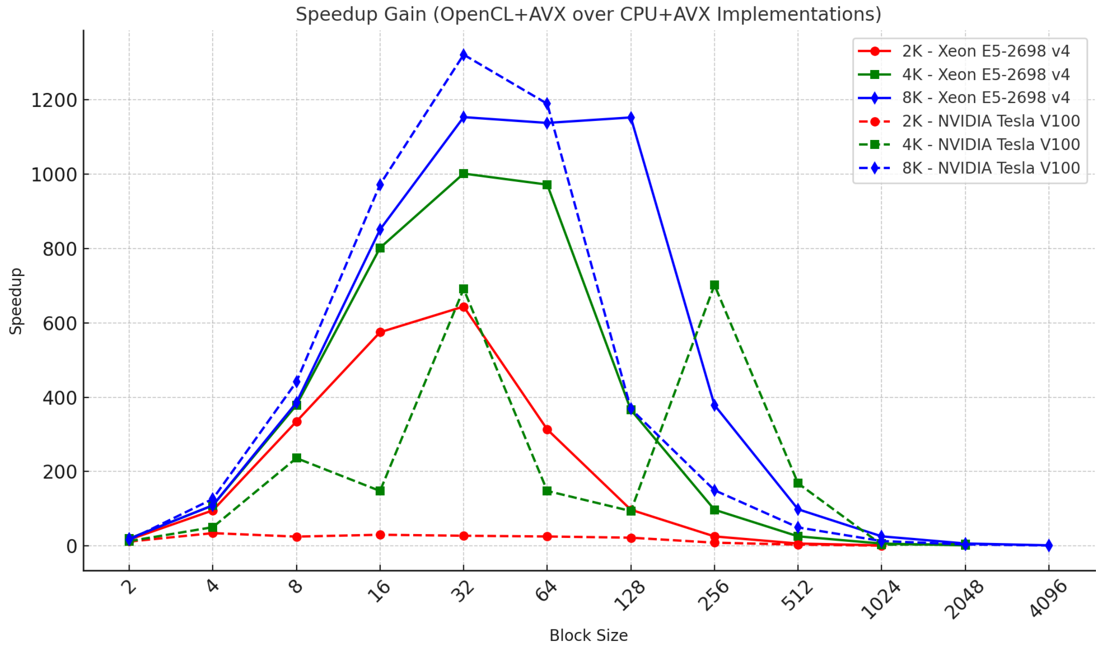

Figure 5.13: Speedup gain of Xeon OpenCL+AVX E5-2698 v4 CPU and NVIDIA V100 GPU over Xeon Sequential+AVX CPU

5.4.3.9 Compare Speedup Gain On OpenCL Xeon and i5 over sequential Implementations

While the UHD 620 GPU consistently outperforms the i5-8265U, especially at higher resolutions and larger block sizes, the speedup achieved through parallel implementations on the Xeon and i5 processors indicates that the Intel Xeon, when paired with the NVIDIA V100 GPU, delivers better performance than the Xeon E5-2698 v4.

Figures 5.14 and 5.15 compare the speedup achieved by GPU acceleration over CPU execution for a video encoding pipeline using OpenCL. Both graphs explore performance across various block sizes (logarithmic scale on the x-axis) and three video resolutions: 2K, 4K, and 8K. The y-axis represents the resulting speedup factor relative to the respective CPU baseline.

In Figure 5.15, the NVIDIA V100 GPU exhibits modest speedup at smaller block sizes (2 to 16), likely due to overhead from frequent kernel launches and limited concurrency at fine granularity. However, performance improves significantly with mid-sized blocks especially around block size 128 where the GPU achieves a peak speedup of approximately 55× for 2K resolution. This peak reflects an optimal balance between computational load distribution and memory efficiency.

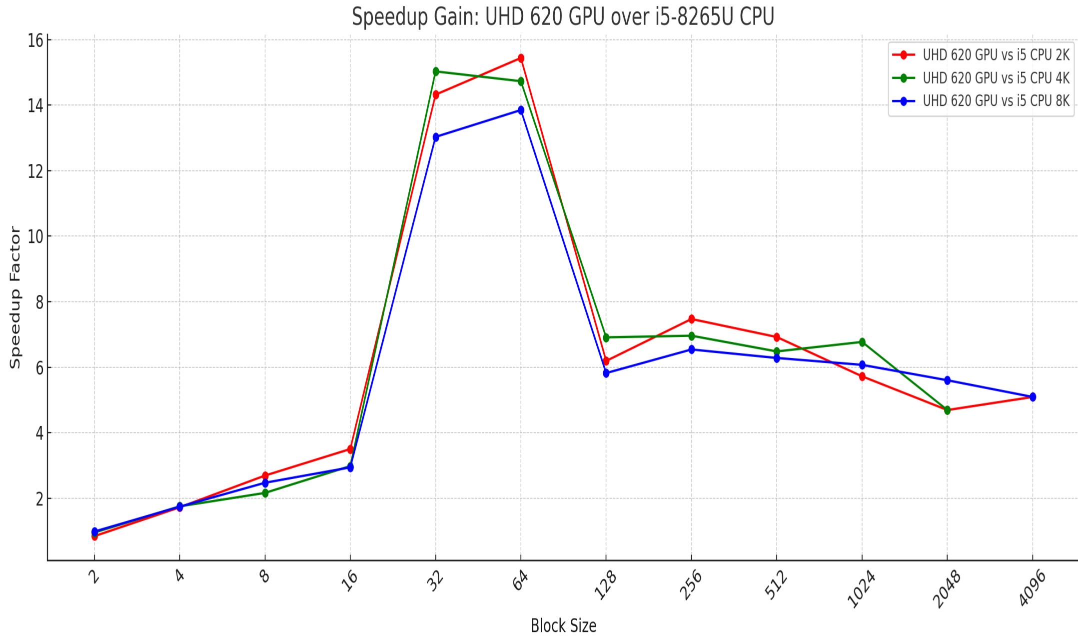

Figure 5.14: Speedup gain: Intel UHD 620 GPU vs. Intel i5-8265U CPU using OpenCL.

As the block size increases beyond 256, the speedup begins to decline, attributed to reduced parallel block count and underutilization of GPU resources. Moreover, speedup factors for 8K resolution remain consistently lower than those for 2K and 4K, suggesting possible limitations due to memory bandwidth or reduced occupancy.

In contrast, Figure 5.14 presents the performance of the Intel UHD 620 integrated GPU against the i5-8265U CPU. Although the absolute speedup values are lower compared to the V100, the GPU still demonstrates a clear performance peak around block size 64, reaching up to 15.5× speedup across all tested resolutions. This suggests that mid-sized blocks provide an ideal balance between thread scheduling and memory access efficiency on integrated GPUs. Beyond block size 128, the speedup steadily declines and stabilizes between 4× and 7×, likely due to diminishing parallel workload distribution. Notably, the UHD 620 shows less variation across resolutions, highlighting its limited capacity to scale with increasing resolution and complexity.

Overall, both figures highlight the importance of tuning block size for maximizing GPU acceleration. The NVIDIA V100 demonstrates impressive gains at optimal block sizes particularly for lower resolutions while the UHD 620, despite its

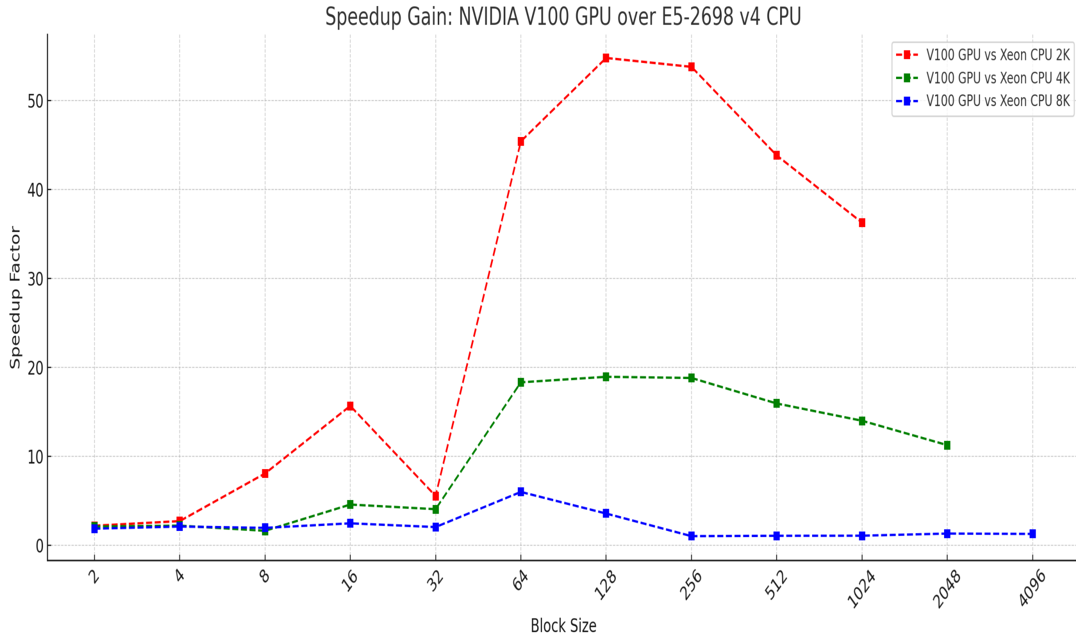

Figure 5.15: Speedup gain: NVIDIA V100 GPU vs. Xeon E5-2698 v4 CPU using OpenCL.

architectural limitations, still achieves respectable speedup when configured appropriately. In both cases, extremely small or very large block sizes result in suboptimal performance, underscoring the need for balanced granularity to best utilize available compute resources.

The comparison between the speedup gain of OpenCL on Xeon and Core i5 in sequential + AVX implementations shows similar results. Both processors benefit from AVX optimizations in sequential workloads, and when using OpenCL, their performance improvements follow a comparable trend. Xeon processors generally offer higher memory bandwidth and additional cores, which can influence performance depending on the workload, but in cases where AVX instructions dominate, the speedup gains between the two tend to converge.

The sharp rise in speedup for 2K resolution between block sizes 32 and 64, as shown in Figure 5.15, can be attributed to a transition from inefficient fine-grained task execution to an optimal block configuration that fully leverages the GPU's architecture. At smaller block sizes, the kernel launch overhead and limited workload per thread hinder performance. However, at block size 64, the computation becomes sufficiently coarse-grained to align well with the V100 GPU's execution model, resulting in a substantial gain in speedup. In contrast, the Xeon CPU

experiences minimal benefit from increasing block size in this range, amplifying the relative performance difference.

5.4.3.10 Compare the speedup gain of OpenCL+AVX Xeon and i5 in sequential Implementations

The Intel UHD 620 GPU achieves considerable speedup over its sequential implementation, but it is consistently outperformed by the NVIDIA V100 GPU, which offers significantly higher parallel compute capability. Similarly, the Xeon E5-2698 v4 CPU shows noticeable improvement over the i5-8265U CPU, though the performance gap is smaller due to limited GPU cores and memory bandwidth, reflecting the architectural differences between general-purpose CPUs and parallel-optimized GPUs.

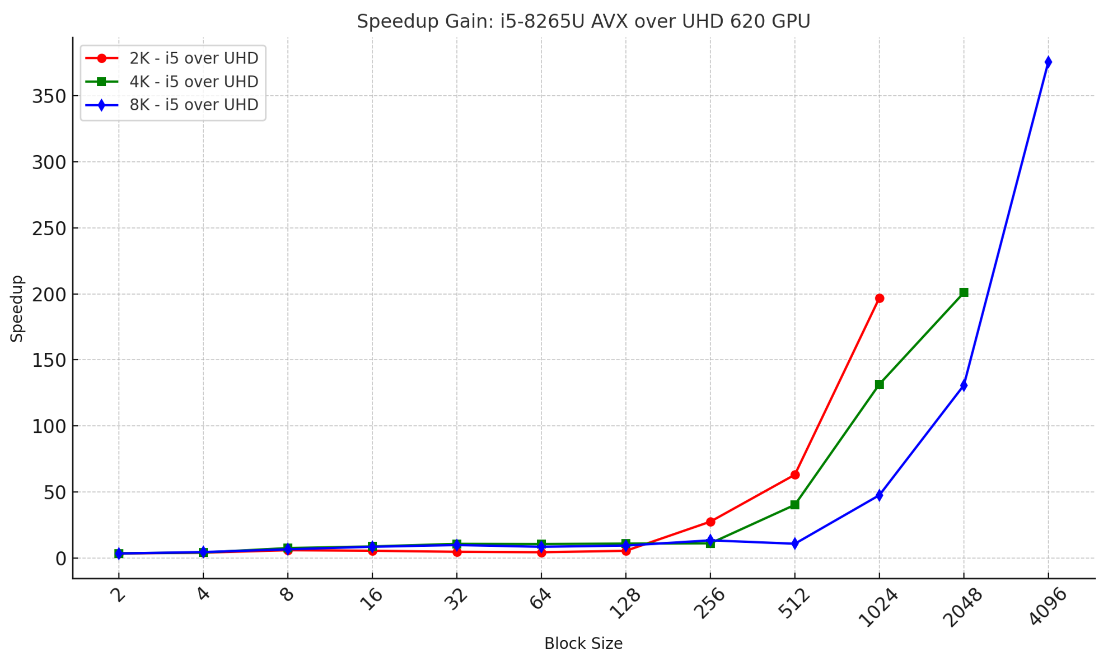

Figure 5.16: Compare Speedup gain i5 OpenCL+AVX i5-8265U CPU over UHD 620 GPU in i5 Sequential CPU

The two graphs in Figures 5.16 and 5.17 compare CPU versus GPU performance for different architectures. In Figure 5.17, the Intel Xeon E5-2698 v4 shows a speedup gain over the NVIDIA V100 primarily at low resolutions (2K) and mid-sized blocks (16–64), while at higher resolutions and larger block sizes the GPU becomes more efficient. In contrast, Figure 5.16 shows the i5-8265U CPU with AVX significantly outperforming the integrated UHD 620 GPU, especially as reso-

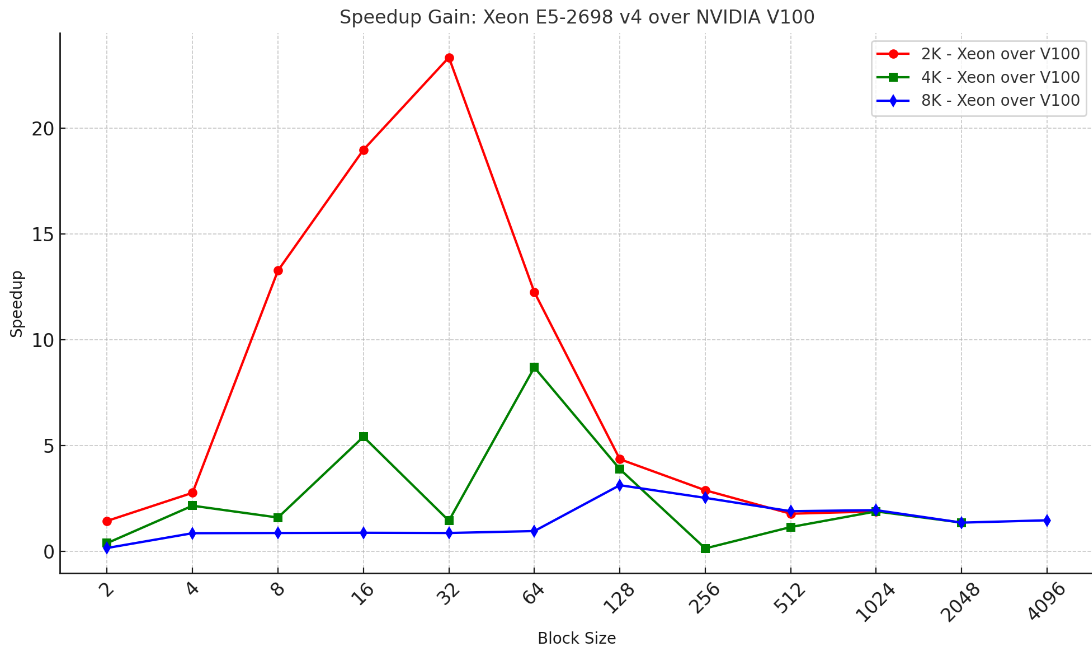

Figure 5.17: Compere Speedup gain of Xeon OpenCL+AVX E5-2698 v4 CPU over NVIDIA V100 GPU in Xeon Sequential CPU

lution and block size increase, with speedups exceeding 300× for 8K. These results emphasize that while discrete GPUs outperform CPUs at high workloads, integrated GPUs are quickly outpaced by even moderately capable CPUs with AVX support.Compare the Speedup gain Of OpenCL+AVX Xeon and i5 in sequential+AVX Implementations with Compare the speedup gain of OpenCL+AVX Xeon and i5 in sequential Implementations are the same result.

5.4.3.11 Compare the Speedup gain Of OpenCL and OpenCL+AVX Xeon and i5 in sequential and AVX Implementations

Comparing OpenCL and OpenCL+AVX implementations on Xeon and Core i5 reveals that OpenCL achieves a higher speedup on Xeon, while Core i5 benefits more from OpenCL combined with AVX optimizations. Xeon's architecture leverages OpenCL effectively for parallel computing, whereas the AVX instruction set enhances Core i5's performance in computationally intensive workloads.
Figures 5.18 and 5.19 collectively demonstrate the performance characteristics of OpenCL-based H.266 encoding across heterogeneous platforms and optimization levels. In Figure 5.18, the speedup gain of the Intel UHD 620 GPU (integrated GPU) using OpenCL is compared against the Intel Core i5-8265U CPU implemen-

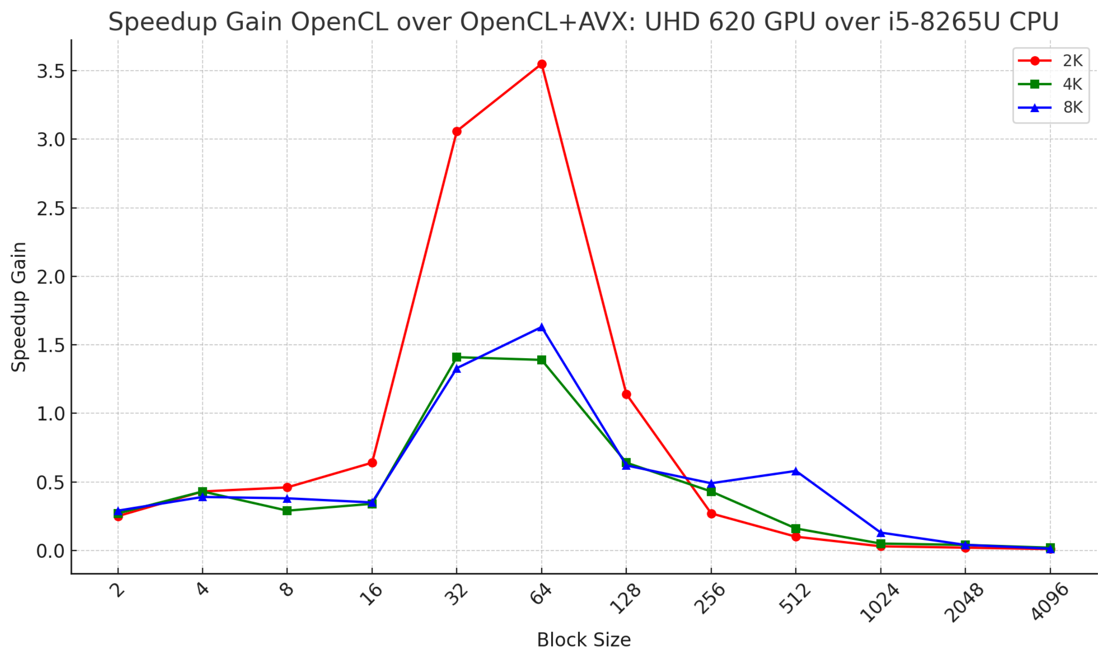

Figure 5.18: Compare the Speedup gain Of OpenCL over OpenCL+AVX i5 in sequential and AVX Implementations

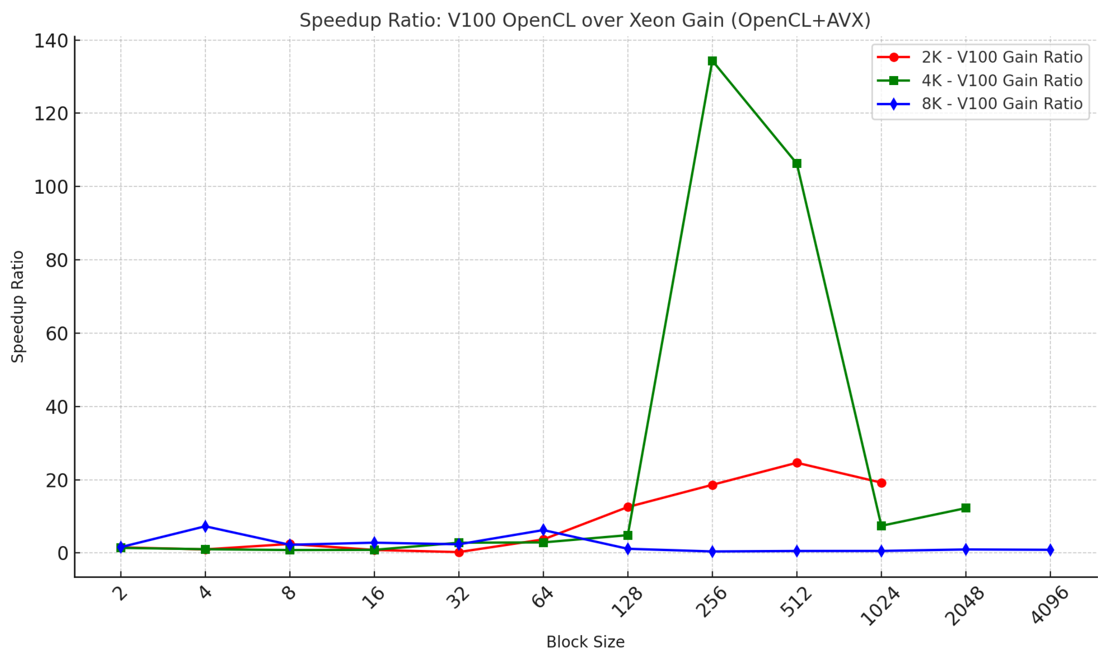

Figure 5.19: Compare the Speedup gain Of OpenCL over OpenCL+AVX: Xeon in sequential and AVX Implementations

tation that combines OpenCL with AVX vectorization. The results span various block sizes for 2K, 4K, and 8K resolutions. Notably, for smaller block sizes (2–16), both GPU and CPU are underutilized, resulting in modest gains due to kernel launch overheads and memory access inefficiencies. However, for mid-sized blocks (32–128), the GPU demonstrates significant acceleration—up to 3.55× speedup at 64×64 block size in 2K resolution due to better thread utilization and memory coalescing in GPU execution. Beyond 256, speedup gains sharply decline as the large block sizes reduce available parallelism, leading to suboptimal occupancy on the GPU's execution units. In contrast, Figure 5.19 shows the speedup ratio of a high-end NVIDIA V100 GPU over a Xeon E5-2698 v4 CPU, both using OpenCL with SIMD optimizations. This comparison highlights the raw parallel processing advantage of a dedicated GPU architecture. The V100 outperforms the Xeon CPU significantly, particularly at 4K resolution and block sizes between 128 and 512, where the speedup ratio spikes beyond 130×. This is attributed to the V100's thousands of cores efficiently handling massive data-parallel workloads, which are prevalent in video encoding operations such as transform and quantization. However, the performance benefit diminishes at very large block sizes (1024–4096), where the GPU's fine-grained parallelism is underutilized, and kernel execution becomes memory-bound. Similarly, smaller block sizes incur overhead due to excessive kernel launches and synchronization. These findings underscore that optimal performance in H.266 encoding depends not just on hardware capabilities but also on carefully chosen block sizes that match the execution model of the target device. Together, the figures validate the importance of adaptive block size selection and heterogeneous acceleration strategies in real-time VVC encoders.
The sharp peak in speedup at 4K resolution, particularly for block sizes between 128 and 512, is due to an optimal balance between data size and hardware utilization. At this resolution, the workload is large enough to fully exploit the parallel architecture of the V100 GPU without overwhelming memory resources, while the Xeon CPU reaches saturation in performance with AVX vectorization. In contrast, 2K workloads are too small to fully leverage the GPU, and 8K introduces memory related performance penalties, reducing the relative speedup.

In some cases, OpenCL-only implementations outperform OpenCL+AVX vectorized versions on CPUs due to several factors. While AVX enables data-level parallelism through SIMD instructions, OpenCL leverages thread-level parallelism across multiple cores. If the workload is inherently parallel and well-balanced across threads, the benefits of AVX vectorization may be marginal or even counterproductive due to added synchronization, memory alignment constraints, and cache pressure. Moreover, the overhead of preparing data in SIMD-friendly formats and managing register usage can offset the gains of vectorization. Studies such as [79] and [80] have shown that SIMD optimizations are highly sensitive to data layout, memory access patterns, and kernel structure, and may not always lead to significant performance improvement when combined with task-level parallelism.

5.5 Discussion

In this section, we discuss how block size affects H.266 encoding time on CPUs and GPUs in OpenCL-based implementations and AVX Vectorilazation, as well as the performance improvements that can be achieved through optimization techniques. This study presents an in-depth analysis of how both block size and software-level optimization strategies influence the overall performance of H.266 encoding. By examining the interaction between hardware characteristics and encoding parameters, the analysis aims to identify optimal configurations that maximize efficiency and reduce computational overhead in OpenCL-based H.266 implementations.
RQ1: How does block size affect H.266 encoding time on CPUs and GPUs in OpenCL-based implementations?
Block size plays a crucial role in the performance of H.266 encoding when using OpenCL-based implementations. Smaller block sizes, such as 2 and 4 are generally better result than sequential implementation across both CPU and GPU architectures. This is primarily because processing a large number of small blocks leads to excessive kernel invocations, increased overhead, and inefficient memory utilization[22][82]. On CPUs, particularly those like the Intel i5-8265U, this results in underutilization of SIMD vector units such as AVX. Rather than executing oper-

ations on large data segments in parallel, the processor is burdened with repeated operations on minimal data, causing throughput to degrade.

As block size increases to moderate levels typically in the range of 16 to 128 encoding performance improves dramatically. This improvement is attributed to a better balance between parallel thread execution and memory access efficiency. Larger blocks enable more data to be processed per kernel launch, maximizing the use of vector instructions and allowing greater thread level parallelism. CPUs with higher core counts, such as the Intel Xeon E5-2698 v4, particularly benefit from this range, as workloads can be distributed evenly across threads, making full use of AVX instructions. GPUs, especially the NVIDIA V100, also achieve their peak efficiency in this region due to their massive parallel processing capabilities and high memory bandwidth.

In summary, block size must be carefully chosen to align with the target hardware architecture. Small blocks lead to excessive overhead and poor vector utilization, while very large blocks reduce parallelism and introduce memory inefficiencies. Medium block sizes, typically between 32 and 128, offer the best performance for both CPUs and GPUs by balancing vector execution, memory access patterns, and thread scheduling.

RQ2: What performance improvements can be achieved in H.266 encoding through AVX, AVX combined with OpenCL and block size variation as optimization techniques?

One of the key optimization techniques we applied to H.266 encoding is AVX vectorization both as a standalone method and in conjunction with OpenCL. This approach led to substantial performance improvements by exploiting both data-level and task-level parallelism. However, the AVX-based implementation alone showed only a slight improvement compared to the baseline. Therefore, additional optimization techniques are necessary to achieve more significant gains. When OpenCL is paired with AVX (Advanced Vector Extensions), performance improves even further particularly during compute intensive stages like transform and quantization. In some cases, the improvements are striking. For example, CPUs like the Intel Xeon equipped with AVX can see speedups of over 3000× compared to their non-AVX (scalar) counterparts. This powerful combination of

OpenCL and AVX allows for the simultaneous processing of many data elements per instruction cycle, significantly increasing throughput. Even more interestingly,when AVX vectorization is enabled on the i5-8265U, its performance jumps more than 370×, even surpassing the GPU. This shows that, in certain scenarios especially when compute-heavy operations are carefully optimized well-tuned CPU-based SIMD implementations can outperform GPUs.

Additionally, block size tuning plays a key role; optimal performance is generally achieved between block sizes 64 and 256, where hardware resources are best utilized and memory throughput is maximized. Overall,these results underscore that architectural-aware optimization combining OpenCL, AVX, and block level graininess can unlock massive gains in encoding throughput. additionally, the performance improvements achieved through optimization techniques in H.266 encoding are significant and vary across hardware platforms and resolutions. GPU-based OpenCL implementations demonstrate substantial speedup over CPU-based baselines, with the NVIDIA V100 achieving up to 55× speedup compared to a Xeon CPU at optimal block sizes. Overall, these optimization techniques dramatically reduce encoding time and improve efficiency across various hardware setups and resolutions. They make high efficiency video coding like H.266 not just faster, but also more accessible even on devices with limited computational power.

Chapter 6
Conclusion and Recommendations

6.1 Conclusion

This study underscores the pivotal role of block size selection and software-level optimization techniques in enhancing the performance of H.266 encoding using OpenCL. Through extensive evaluations across a range of hardware platforms including high-performance CPUs like the Intel Xeon E5-2698 v4 and mainstream processors such as the Intel i5-8265U, our evaluations show that encoding times consistently decrease as block sizes increase from small configurations (2×2) to medium sizes (64×64 or 128×128). This improvement is primarily attributed to the reduced number of blocks, which lowers kernel launch overhead and improves computational throughput.

However, performance gains tend to plateau or even decline at block sizes beyond 256×256. This degradation arises from reduced parallelism, inefficient memory access patterns, and workload imbalance factors that are particularly impactful on systems with limited computational resources or memory bandwidth.

The application of OpenCL-based parallelism significantly accelerates encoding by leveraging the high concurrency of modern CPUs and GPUs. When combined with SIMD vectorization techniques specifically using AVX and SSE instruction sets additional speedups are realized, particularly in computation-intensive stages such as transformation and quantization. These vectorized operations allow multiple data elements to be processed in a single CPU cycle, alleviating bottlenecks common in high-resolution encoding workflows. Notably, all optimizations explored in this study are software-based and require no hardware modifications, making them both portable and scalable.

Our findings emphasize that the effectiveness of these optimization strategies is highly dependent on the underlying hardware architecture. High-end systems exhibit the most substantial gains with mid-sized block configurations, which best match their extensive parallel execution capabilities. In contrast, lower-power platforms benefit from block sizes that carefully balance workload distribution and overhead management.

In summary, block size must be carefully chosen to align with the target hardware architecture. Small blocks lead to excessive overhead and poor vector utilization, while very large blocks reduce parallelism and introduce memory inefficiencies. Medium block sizes, typically between 32 and 128, offer the best performance for both CPUs and GPUs by balancing vector execution, memory access patterns, and thread scheduling. For optimal encoding efficiency in OpenCL-based H.266 implementations, tuning the block size according to the resolution and hardware configuration is crucial.

Furthermore, the integration of optimization techniques such as OpenCL and AVX vectorization leads to a better improvements in encoding efficiency. OpenCL alone significantly accelerates execution by leveraging massive GPU parallelism, while the addition of AVX instructions on CPUs further enhances data throughput through SIMD execution. The combination of these techniques when paired with carefully tuned block sizes enables speedup factors reaching thousands over sequential baselines. Overall, these results affirm that performance in H.266 encoding can be greatly enhanced through architecture-aware optimization strategies, underscoring the importance of both algorithmic tuning and hardware-specific enhancements in modern video encoding pipelines.

6.2 Recommendations

Furthermore, the integration of optimization techniques such as OpenCL and AVX vectorization yields substantial improvements in encoding efficiency. OpenCL significantly accelerates execution by leveraging the massive parallelism of GPUs, while AVX instructions on CPUs enhance data throughput through SIMD-based processing. When combined with carefully tuned block sizes, these techniques enable speedup factors that can reach several thousand times over traditional sequential baselines. These results clearly demonstrate that performance in H.266 encoding can be greatly enhanced through architecture-aware optimization strategies, underscoring the importance of both algorithmic refinement and hardware-specific enhancements in modern video encoding pipelines.

Based on the findings of this study, several recommendations are proposed to further advance the performance of H.266 encoding using OpenCL. First, adaptive block size tuning should be investigated to dynamically select optimal block sizes during encoding. This approach should consider both the resolution and the content complexity of video frames to maximize processing efficiency and maintain compression quality. Second, more advanced OpenCL optimizations such as asynchronous data transfers, memory tiling, and effective utilization of shared local memory should be explored to further exploit GPU computational capabilities.

Additionally, it is recommended to tailor encoding strategies to the characteristics of the underlying hardware. For example, different optimization techniques should be applied for multi-core CPUs versus massively parallel GPUs to leverage their respective strengths. Future work should also incorporate an analysis of energy consumption to ensure that the achieved performance gains do not come at the cost of excessive power usage. This is especially critical for mobile and embedded platforms, where power efficiency is a primary constraint. Finally, scaling the encoding workload across multi-GPU systems should be considered to support real-time and ultra-high-resolution video encoding, thereby enabling greater scalability and performance for future video processing demands.

Bibliography

[1] V. Sze et al. *High Efficiency Video Coding (HEVC)*. Springer, 2014. https://doi.org/10.1007/978-3-319-06895-4

[2] B. Bross et al., "Overview of the Versatile Video Coding (VVC) Standard," *IEEE Trans. Circuits Syst. Video Technol.*, 2021.

[3] J. Chen et al. "Algorithm Description for Versatile Video Coding and Test Model 5 (VTM 5)". *JVET Document*, Oct. 2018.

[4] X. Li et al., "Complexity Analysis of VVC Encoding," *Proc. IEEE Int. Conf. Image Process.*, 2020.

[5] X. Li et al. "Complexity Analysis of VVC Encoding." *IEEE Access*, 2021.

[6] Hasitha Muthumala Waidyasooriya,Masanori Hariyama and Hiroe Iwasaki. "OpenCL-Based Design of an FPGA Accelerator for H.266/VVC Transform and Quantization.2022,10.1109/MWSCAS54063.2022.9859281.

[7] M. Karrenbauer et al. "OpenCL vs CUDA for Video Processing." ACM SIGARCH, 2021.

[8] S. Park et al. "Memory Coalescing for GPU Video Encoding." *IEEE Trans. Parallel Distrib. Syst.*, 2022.

[9] J. Kim et al. "Wavefront Parallel Processing for VVC." *IEEE Trans. Circuits Syst. Video Technol.*, 2022. https://doi.org/10.1109/TCSVT.2021.3101953

[10] L. Zhao et al., "QTMT Partitioning Complexity in VVC: Analysis and Optimization," *IEEE Trans. Multimedia*, 2023.

[11] J. Park and S. Kim, "Affine Motion Estimation in VVC: Computational Overhead and Speed-Accuracy Tradeoffs," *IEEE Access*, 2022.

[12] Y. Huang et al., "Multiple Transform Selection in VVC: Impact on Encoding Efficiency and Complexity," *Proc. IEEE Int. Conf. Image Process.*, 2021.

[13] V. Sanchez et al., "In-Loop Filtering Complexity in VVC: A Comparative Study," *IEEE Trans. Consumer Electron.*, 2022.

[14] K. Müller et al., "Real-Time VVC Encoding: Challenges and Opportunities," *J. Vis. Commun. Image Represent.*, 2023.

[15] J. Chen et al., "Parallel HEVC Encoding on CPU-GPU Systems," *IEEE Trans. Multimedia*, 2015.

[16] G. Correa et al., "GPU Acceleration of HEVC Motion Estimation," *IEEE Trans. Consumer Electron.*, 2018.

[17] Z. Liu et al., "OpenCL-Accelerated Motion Compensation for VVC," *IEEE Trans. Circuits Syst. Video Technol.*, 2023.

[18] H. Wang et al., "Parallel VVC Encoding Challenges," *IEEE Trans. Computers*, 2022.

[19] S. Kim and J. Park, "QTMT Partitioning in VVC: GPU Workload Analysis and Optimization," *IEEE Access*, 2022.

[20] Y. Zhang et al., "Data-Aware Task Scheduling for VVC Parallelization," *Proc. IEEE Int. Conf. Multimedia and Expo*, 2023.

[21] K. Müller et al., "Hybrid CPU/GPU VVC Encoding Framework," *IEEE Trans. Circuits Syst. Video Technol.*, 2021.

[22] P. Gupta et al., "OpenCL Memory Optimization for VVC Affine Motion," *Proc. IEEE Int. Symp. Circuits and Systems*, 2022.

[23] T. Nguyen et al., "Parallel In-Loop Filtering in VVC via OpenCL Pipes," *Proc. IEEE Data Compression Conf.*, 2023.

[24] V. Sanchez et al., “CABAC Parallelization Limits in VVC Encoding,” *IEEE Trans. Broadcasting*, 2021.

[25] Y. Lee et al. “OpenCL-Accelerated Motion Estimation for H.264,” *Proc. ACM Multimedia*, 2014.

[26] C. Kim et al. “OpenCL Optimization of HEVC Transform and Quantization,” *Proc. IEEE Int. Conf. Image Process.*, 2016.

[27] P. Gupta et al. “AV1 Motion Search Acceleration on GPUs using OpenCL,” *Proc. ACM Multimedia Syst. Conf.*, 2020.

[28] H. Wang et al. “Optimizing OpenCL Performance: Memory and Workload Considerations,” *IEEE Int. Parallel Distrib. Process. Symp.*, 2019.

[29] Y. Zhang et al. “Dynamic Prefetching for OpenCL Video Encoders,” *Proc. IEEE GlobalSIP*, 2020.

[30] R. Patel et al. “Partitioning VVC In-Loop Filters on CPU+GPU via OpenCL,” *Proc. Picture Coding Symp.*, 2021.

[31] T. Nguyen and S. Lee, “Workload-Aware Scheduling for VVC on Heterogeneous Platforms,” *IEEE Trans. Multimedia*, 2022.

[32] D. Cabrera et al. “Accelerating CABAC in VVC with OpenCL,” *Proc. IEEE Int. Conf. Image Process.*, 2022.

[33] A. Rahman et al. “Reinforcement Learning for OpenCL Parameter Auto-Tuning in HEVC,” *Proc. IEEE Int. Conf. Image Process.*, 2021.

[34] J. Bankoski et al. “Technical Overview of VVC and Coding Efficiency Gains,” *IEEE Trans. Circuits Syst. Video Technol.*, 2022.

[35] Y. Lin et al. “Hardware Acceleration of VVC’s Multiple Transform Selection,” *Proc. IEEE Int. Symp. Circuits and Systems*, 2021.

[36] W. Zhao et al. “CNN-Accelerated Intra Prediction for VVC,” *IEEE Trans. Multimedia*, 2022.

[37] M. Li et al. "Deep Affine Motion Compensation for Video Coding," *IEEE Trans. Image Process.*, 2023.

[38] A. Ahmed et al. "Perceptual Intra Coding for VVC at Low Bitrates," *Proc. IEEE Int. Conf. Acoust. Speech Signal Process.*, 2022.

[39] N. Ma et al. "Tile Adaptation for 3D-HEVC on Multicore Systems," *IEEE Trans. Multimedia*, 2020.

[40] L. Gao et al. "OpenCL-Based Motion Estimation for 8K Video," *Proc. IEEE Int. Conf. Consumer Electron.*, 2021.

[41] F. Smith et al. "Hybrid CPU-GPU Encoding Framework for Next-Gen Codecs," *Proc. SPIE Optical Eng. + Applicat.*, 2021.

[42] R. Sanchez et al. "Bottlenecks in OpenCL VVC Encoding," *Proc. IEEE Int. Conf. Image Process.*, 2023.

[43] L. Chen et al. "Fractional ME Challenges in GPU Video Coding," *Proc. IEEE Int. Conf. Image Process.*, 2021.

[44] Y. Chen et al. "Parallelizing ALF/CCALF in VVC," *Proc. IEEE Int. Conf. Multimedia and Expo*, 2022.

[45] Z. Liu et al. "Adaptive Task Allocation in VVC CPU+GPU Encoding," *Proc. IEEE Int. Conf. Image Process.*, 2023.

[46] P. Gupta et al. "Limitations of Block-Adaptive Task Scheduling in VVC," *IEEE Access*, 2023.

[47] K. Müller et al. "Synchronization Primitives in OpenCL vs CUDA for VVC," *Proc. ACM Multimedia*, 2022.

[48] A. Rahman et al. "OpenCL Kernel Launch Overhead for Small Tasks in VVC," *Proc. IEEE Int. Conf. Image Process.*, 2022.

[49] J. Lee et al. "Non-Coalesced Access Penalties in VVC GPU Encoding," *Proc. IEEE Int. Symp. Circuits and Systems*, 2021.

[50] T. Zhang et al. “Synchronization Overhead in Pipeline Parallelism for Video Coding,” *Proc. IEEE Int. Conf. Image Process.*, 2020.

[51] M. Ito et al. “Memory Allocation Strategies for 8K OpenCL Encoding,” *Proc. IEEE Int. Conf. Consumer Electron.*, 2022.

[52] D. Nguyen et al. “Half-Precision Computation in Video Encoding: Trade-offs,” *Proc. IEEE GlobalSIP*, 2021.

[53] B. Bross et al. *Versatile Video Coding (VVC) Draft 8.* JVET-P2001, 2020.

[54] F. Bossen et al. “VVC Test Model and Encoder Description VTM 8.0.” *JVET Document*, 2020.

[55] D. Flynn et al. “Overview of Tools in Versatile Video Coding (Draft 10).” *JVET Document*, 2020.

[56] J. Chen et al. “JVET Common Test Conditions and Software Reference Configurations for SDR Video.” *JVET Document*, Oct. 2018.

[57] F. Schwarz et al. “Block Merging for Quadtree-Based Partitioning in HEVC.” *Proc. IEEE Int. Conf. Image Process.*, 2012.

[58] B. Li et al. “Intra Prediction Modes in VVC.” *JVET Document*, 2018.

[59] G. Choi et al. “Hierarchical Motion Estimation for High Efficiency Video Coding,” *IEEE Trans. Circuits Syst. Video Technol.*, 2016.

[60] G. J. Sullivan et al. “Standardized Extensions of HEVC and VVC for 360-Degree Video and Screen Content,” *IEEE Proc.*, 2021.

[61] S. Choi et al. “Adaptive Loop Filter in VVC: Design and Performance,” *Proc. IEEE Int. Conf. Image Process.*, 2019.

[62] Intel Corp. “Intel SDK for OpenCL – Optimization Guide.” 2019.

[63] AMD Inc. “AMD RDNA 2 Architecture Overview (7900 XT).” 2022.

[64] NVIDIA Corp. “NVIDIA Ampere GPU Architecture Whitepaper.” 2020.

[65] Khronos OpenCL Working Group. *The OpenCL Specification Version 3.0.* 2020.

[66] A. Munshi. "The OpenCL Specification," *IEEE Hot Chips*, 2009.

[67] W. J. Dally and B. Towles. *Principles and Practices of Interconnection Networks*. Morgan Kaufmann, 2004.

[68] J. Nickolls et al. "Scalable Parallel Programming with CUDA." *ACM Queue*, 2008.

[69] Khronos OpenCL Working Group. *OpenCL Performance Optimization Guide*. 2021.

[70] G. Marin and J. Mellor-Crummey. "Strategies for Realistic Measurement of End-to-End Application Performance." *Proc. ACM/IEEE Supercomputing*, 2015.

[71] B. Bross, J. Chen, S. Liu, & Y. Wang, "Versatile Video Coding (VVC) Standard – Draft 10," JVET-N1001, Oct. 2020.

[72] J. Chen, Y. Ye, & S. Kim, "Algorithm Description for Versatile Video Coding and Test Model 10 (VTM 10)," JVET-P2002, Jan. 2020.

[73] S. Vanne, M. Viitanen, T. D. Hämäläinen, and A. Hallapuro, "Comparative Rate–Distortion–Complexity Analysis of HEVC and AVC Video Codecs," IEEE Transactions on Circuits and Systems for Video Technology, vol. 22, no. 12, pp. 1885–1898, Dec. 2012. DOI: 10.1109/TCSVT.2012.2221191

[74] Y. Chen, B. Li, and J. Xu, "SIMD-Based Optimization for Transform and Quantization in HEVC Encoder," in Proceedings of the IEEE International Conference on Image Processing (ICIP), Paris, France, Oct. 2014, pp. 2624–2628. DOI: 10.1109/ICIP.2014.7025460

[75] J. Ryu, D. Kim, & H. Lee, "OpenCL-based hardware acceleration of HEVC encoder on heterogeneous platforms," *Proc. IEEE Int. Conf. Consumer Electronics (ICCE)*, 2016, pp. 273–274.

[76] I. Ahmad, X. Chen, & Y. Zhang, "OpenCL framework for real-time HEVC motion estimation," *J. Real-Time Image Process.*, vol. 15, pp. 573–585, 2018.

[77] J. Wang, S. Li, & Y. Zhang, "A low-power hardware-efficient encoder for VVC," *IEEE Access*, vol. 9, pp. 85347–85359, 2021.

[78] SIMD Acceleration for Video Processing: Techniques and Case Studies, author=Baker, David and Smith, Alan and Zhou, Mei, booktitle=2020 IEEE International Conference on Multimedia and Expo (ICME), pages=1–6, year=2020, organization=IEEE

[79] S. Vanne, M. Viitanen, T. D. Hämäläinen, and A. Hallapuro, "Comparative Rate–Distortion–Complexity Analysis of HEVC and AVC Video Codecs," IEEE Transactions on Circuits and Systems for Video Technology, vol. 22, no. 12, pp. 1885–1898, Dec. 2012. DOI: 10.1109/TCSVT.2012.2221191

[80] Y. Chen, B. Li, and J. Xu, "SIMD-Based Optimization for Transform and Quantization in HEVC Encoder," Proc. IEEE Int. Conf. Image Process. (ICIP), 2014, pp. 2624–2628. DOI: 10.1109/ICIP.2014.7025460

[81] https://www.guru3d.com/story/intel-is-the-first-to-support-h266-vvc-decoding-ahead-of-nvidia-and-amd

[82] Jakubowski,Pastuszak,"Block-based motion estimation algorithms a survey". Opto-Electron. Rev. 21, 86–102 (2013). https://doi.org/10.2478/s11772-013-0071-0