UNIVERSITY OF THESSALY

Video Coding Optimization Techniques

Author:

Georgios I. Papaioannou

Supervisor:

Dr. Athanasios Loukopoulos

A thesis submitted in fulfillment of the requirements for the degree of

Doctor of Philosophy

in the

Department of Computer Science and Biomedical Informatics

March 2024

Η εργασία αυτή αφιερώνεται στον Θεό μου, που μου έδωσε τη δύναμη να την ολοκληρώσω. Επίσης, στους ανθρώπους που στάθηκαν δίπλα μου όλα αυτά τα χρόνια της διατριβής μου, με υπομονή και συνεχή παρότρυνση. Αυτοί είναι, κυρίως, ο Θανάσης και η Νατάσα. Ο Θανάσης, όχι μόνο ως επιβλέπων, αλλά πάνω απ' όλα ως φίλος και πιστός συνεργάτης. Διευκόλυνε το έργο μου με πολλούς και διαφόρους τρόπους από την πρώτη ημέρα, που τον συνάντησα. Νιώθω πραγματικά τυχερός, που οι δρόμοι μας συναντήθηκαν με έναν εντελώς απροσδόκητο τρόπο και μπόρεσα να τον γνωρίσω ως άνθρωπο. Θα του είμαι για πάντα ευγνώμων. Και η Νατάσα, η σύντροφος της ζωής μου, για την ανεξάντλητη υπομονή της και το ήρεμο παρηγορητικό της ύφος σε όλες εκείνες τις δύσκολες νύχτες, που τα προβλήματα της εργασίας μού φαίνονταν ανυπέρβλητα. Όλοι μαζί σαν ομάδα φτάσαμε σε αυτό το αποτέλεσμα. Τέλος, δεν πρέπει να παραλείψω να ευχαριστήσω θερμά, δείχνοντας έτσι την ευγνωμοσύνη μου, τους γονείς μου, για την αμέριστη στήριξή τους και αγάπη, από τότε που θυμάμαι τον εαυτό μου, αλλά και στο ακαταμάχητο δίδυμο, που συμπληρώνει τα τελευταία χρόνια την ευτυχία μου, την Όλια και τον Άγγελο. Οι δύο τελευταίοι αποτελούν πλέον αναπόσπαστο κομμάτι της ζωής και της καθημερινότητάς μου, παρά τις συγκρούσεις και τα μεγάλα διαστήματα απουσίας μου από δίπλα τους - έστω και νοερά πολλές φορές - λόγω των επαγγελματικών υποχρεώσεων και των βαθύτερων σκέψεων/προβληματισμών μου. Εύχομαι τα καλύτερα για όλους τους, γιατί ο καθένας με τον τρόπο του συμπλήρωσε επάνω μου - σαν ένα μικρό κομμάτι παζλ - όλα εκείνα τα θετικά στοιχεία, που με ολοκληρώνουν και με κάνουν καλύτερο ως άνθρωπο.

Τόσο λίγες λέξεις για τόσο πολλά συναισθήματα συνιστούν μια εναλλακτική μορφή «απωλεστικής συμπίεσης δεδομένων», που πιθανότατα θα μείνουν για πάντα σε μια ψηφιακή βιβλιοθήκη, ως απόδειξη της ύπαρξης όλων αυτών των παράλληλων σχέσεών μας. Ωστόσο, είναι πολύ πιθανό, ο μελλοντικός αναγνώστης να «αποσυμπιέσει» νοερά όλες τις παραπάνω περιγραφόμενες καταστάσεις, εκτυλίσσοντας στο μυαλό του μια μικρή υποθετική ιστορία με τη μορφή των διαδοχικών εικόνων-καρέ, δηλαδή ενός βίντεο.

UNIVERSITY OF THESSALY

Abstract

Department of Computer Science and Biomedical Informatics

Doctor of Philosophy

Video Coding Optimization Techniques

by Georgios I. Papaioannou

Video encoding techniques are getting increasingly important due to the fast-growing video application demands worldwide, which require high-quality videos coupled with low bit rates for storage and streaming efficiency. Silky-smooth video streaming with high frame rates at high resolutions is an evolving research area for the video encoding ecosystem. As the necessity of supporting ever-increasing demands in video resolution leads to new video coding standards, the challenge of harnessing their computational overhead becomes important. Such overhead stems not only from the increased image data due to higher resolutions but also from the coding techniques per se, that are introduced by each standard to improve compression. All modern standards in the field of video coding offer high compression efficiency but this is achieved as aforementioned, by increasing the computational complexity of both the encoding and decoding parts. Ultra-High-Definition (UHD) videos, bring new encoding implementation patterns, that are being recommended for parallelization schemes. Many popular video standards, are sharing different parallelization approaches. In this thesis research, we developed algorithms to tackle the aforementioned encoding complexity through CPU parallelism and/or GP-GPU (General Purpose-Graphics Processing Units) exploitation. We focus on a multithreaded approach for coarse-grained parallel video encoding, that is based on an entirely new encoding scheme and evaluate the performance gains as well as the visual quality of the algorithms under various scenarios to effectively minimize the computational load. As an extension of the previous efforts, we engaged the video graphics card, to investigate the mapping of a novel GPU-specific fast search motion estimation algorithm, the resulting challenges and benefits therein. Finally, we added a fraction execution resolver to avoid unnecessary accuracy calculations for the motion estimation process and to further improve the computational efficiency.

ΠΑΝΕΠΙΣΤΗΜΙΟ ΘΕΣΣΑΛΙΑΣ

Περίληψη

Τμήμα Πληροφορικής με Εφαρμογές στη Βιοϊατρική

Διδακτορική Διατριβή

Τεχνικές Βελτιστοποίησης Κωδικοποίησης Βίντεο

από Γεώργιος Ι. Παπαϊωάννου

Οι τεχνικές κωδικοποίησης βίντεο αποδεικνύονται ιδιαιτέρως σημαντικές τα τελευταία χρόνια, λόγω της δραματικής αύξησης της χρήσης βίντεο παγκοσμίως, που απαιτεί διαρκώς υψηλότερη ποιότητα εικόνας, με όσο το δυνατό χαμηλότερη ροή δεδομένων και μεγέθη αποθηκευτικού χώρου. Η έρευνα για μεγαλύτερες αναλύσεις και ρυθμούς προβολής ανά δευτερόλεπτο (frame rate) είναι μια διαρκώς εξελισσόμενη διαδικασία μέσα στην επιστημονική κοινότητα. Η ανάγκη υποστήριξης υψηλών αναλύσεων στο πεδίο της εικόνας, και κατ' επέκταση των ροών βίντεο, οδηγεί τη βιομηχανία στη δημιουργία νέων προτύπων κωδικοποίησης βίντεο με την αύξηση της υπολογιστικής πολυπλοκότητας κάθε φορά, να αποτελεί έναν από τους πιο κρίσιμους παράγοντες αντιμετώπισης για την ελαχιστοποίησή της. Σε αυτή συμβάλλει, όχι μόνο η αύξηση των αναλύσεων της εικόνας, αλλά και οι νέες τεχνικές κωδικοποίησης, που παρουσιάζονται, για τη μεγιστοποίηση της συμπίεσης. Όλα τα νέα πρότυπα στον χώρο του βίντεο προσφέρουν μεγάλη αποτελεσματικότητα ως προς τη συμπίεση των δεδομένων, αλλά με αντάλλαγμα, όπως προαναφέρθηκε, την αύξηση της υπολογιστικής πολυπλοκότητας, τόσο από την πλευρά του κωδικοποιητή, όσο και από την πλευρά του αποκωδικοποιητή. Οι υπερύψηλες αναλύσεις (UHD) φέρνουν υποχρεωτικά νέες τεχνικές κωδικοποίησης, που περιλαμβάνουν και την παραλληλοποίηση της διαδικασίας. Σε αυτήν τη διατριβή αναπτύξαμε αλγορίθμους, ώστε να αντιμετωπιστεί η υπολογιστική πολυπλοκότητα κατά την κωδικοποίηση, με χρήση πολλών επεξεργαστών (CPUs) και την αξιοποίηση της κάρτας γραφικών (GPU). Στην ουσία, επικεντρωθήκαμε στην ανάπτυξη ενός πολυδιεργασικού (multithreaded) περιβάλλοντος κωδικοποίησης, δημιουργώντας ένα εντελώς νέο σχήμα παράλληλης κωδικοποίησης και αξιολογήσαμε τον αλγόριθμο σε επίπεδο απόδοσης και ποιότητας με τη χρήση διαφόρων σεναρίων λειτουργίας. Ως επέκταση αυτής της προσπάθειας, εξερευνήσαμε και τις δυνατότητες της κάρτας γραφικών, αναθέτοντάς της τους υπολογισμούς της εκτίμησης της κίνησης (motion estimation). Αυτό έγινε με την εφαρμογή ενός νέου καινοτόμου αλγορίθμου γρήγορης αναζήτησης (fast search), που αξιοποιεί την εν γένει χρήση πολλών πυρήνων της κάρτας. Τέλος,

προσθέσαμε έναν αλγόριθμο εκτίμησης για την αποφυγή των μη απαραίτητων υπολογισμών, που αφορούν την ακρίβεια της εκτιμώμενης κίνησης (motion estimation), ώστε να αυξήσουμε ακόμη περισσότερο την υπολογιστική αποτελεσματικότητα.

Contents

1. **Digital Video Introduction 14**
   1.1. Overview of digital video 14
   1.2. Video encoding 15
   1.3. Motivation and Thesis Contributions 16
   1.4. Thesis Structure 18
2. **Video Codec Basic Structures & Conceptual Framework 20**
   2.1. Overview 20
   2.2. Color Space 20
   2.3. Video Compression 27
   2.4. Block Partitioning for Prediction 35
   2.5. Prediction 37
   2.6. Block Matching Algorithm (BMA) 43
   2.7. Fraction Refining 45
   2.8. Quality 46
   2.9. Entropy Coding 48
   2.10. Parallelization 49
      Slices 49
      Tiles 50
      Wavefront Parallel Processing 51
3. **Tile-Based Wavefront Parallelism in HEVC 53**
   3.1. Overview 53
   3.2. Related Work 55
   3.3. Speedup Performance 56
   3.4. Experiments 58
      Setup 58
      Speedup Results 59
      Coding efficiency results 60
   3.5. Summary and Discussion 62

4. **On Combining Wavefront and Tile Parallelism with a Novel GPU-friendly Fast Search 63**
4.1. Overview 63
4.2. Related Work 67
4.3. Implementation 69
4.4. Experiments 77
Implementation Details 77
Setup 77
Coding Efficiency Results 78
4.5. Summary and Discussion 82
5. **Fractional execution resolver using a hybrid multi-CPU/GPU encoding scheme 84**
5.1. Overview 84
5.2. Related Work 89
5.3. Implementation 92
5.4. Experiments 97
Implementation Details 97
Setup 97
Coding Efficiency Results 98
5.5. Summary and Discussion 103
6. **Conclusion and Future Work 104**
6.1. Conclusions 104
6.2. Future Work 105
Deep Learning AI Methods 105
Distributed Video Encoding With Cloud Computing 106
7. **Bibliography 107**
8. **List of Acronyms and Abbreviations 116**

List of Figures

Figure 1. Chroma subsampling examples. 25
Figure 2. Chroma subsampling comparison. From top to bottom (a) 4:4:4 (b) 4:2:2 (c) 4:2:0 26
Figure 3. (a) Original uncompressed image (left). (b) Visual artifacts and image degradation, after aggressive video compression (right). 27
Figure 4. Image reconstruction discarding coefficients lower than 50 using Matlab..... 28
Figure 5. Matlab commands to apply DCT transformation of the grayscale image above..... 28
Figure 6. Displays the transformed grayscale image above using a logarithmic scale. Most of the energy is in the upper left corner (DC coefficient)..... 28
Figure 7. (a) Uncompressed array (left) (b) DCT transformed array 29
Figure 8 Basic DCT steps 30
Figure 9. Zig-Zag scan order. 31
Figure 10. The design of CABAC involves the key elements of binarization, context modeling, and binary arithmetic coding. These elements are illustrated as the main algorithmic building blocks of the CABAC encoding block diagram, as shown above. 32
Figure 11. DCT steps. 33
Figure 12. HEVC partition blocks are up to 64x64 34
Figure 13. Video standards partition sizes. The bigger, the better [6]. 34
Figure 14. HEVC's highly variable quadtree-based structure. 35
Figure 15. Block diagram of an HEVC encoder with built-in decoder (gray shaded)..... 35
Figure 16. CTU – Coding Tree Unit is a logical unit. 36
Figure 17. Each CTU can be differently split into multiple CUs (Coding Units) and each CU becomes the decision-making point of prediction type. 36
Figure 18. Block partition of HEVC 37
Figure 19. Temporal redundancy to the best match position in consecutive frames over time .. 38
Figure 20. Spatial redundancy takes advantage of similarity among most neighboring pixels 38
Figure 21. Intra-prediction modes of HEVC 39

Figure 22. HEVC's reference samples for a block NxN 40
Figure 23. Motion Vector displacement Δx=x2-x1 and Δy=y2-y1 [9]..... 41
Figure 24. Uni-prediction example. Predicted Frame (P-frames) use frames that have been previously encoded [9]. 42
Figure 25. Bi-prediction example. Bi-prediction (B-frames) use frames that have been previously encoded or frames that are going to be encoded in the future [9]..... 42
Figure 26. Mechanism of block-matching algorithm (Wikipedia) 44
Figure 27. Fraction Estimation prediction. Square samples are integer pixel points. Circle samples are half-interpolated samples and triangles are quarter-pixel interpolated samples..... 46
Figure 28. PSNR showcases..... 47
Figure 29. Matlab commands to apply PSNR calculations of the grayscale image above..... 47
Figure 30. Slice encoding 50
Figure 31. Tile encoding..... 51
Figure 32. Wavefront encoding..... 52
Figure 33. Example of wavefront parallelism (WPP), 8 threads. 54
Figure 34. Example of proposed Wavefront per Tile Parallelism (WTP), 2×2 tiles, 16 threads. ... 55
Figure 35. Average speedup in Class A sequences for QP=32 (Traffic, PeopleOnStreet). 59
Figure 36. Average speedup in Class B sequences for QP=32 (BasketballDrive, BQTerrace, Cactus, Kimono, ParkScene)..... 60
Figure 37. BD-rate percentage of increase versus WPP (24 threads, QP=22, 27, 32, 37). 61
Figure 38. BD-PSNR difference versus WPP (24 threads, QP=22, 27, 32, 37, negative values show a quality drop). 61
Figure 39. Principle of WPP and CTU dependencies..... 64
Figure 40. Principle of Tiles parallelism. 64
Figure 41. Principle of Slices parallelism..... 65
Figure 42. Example of wavefront per Tile parallelism (WTP), 2 × 2 Tiles, 16 Threads..... 65
Figure 43. Wavefront thread-safe structures for each tile..... 66
Figure 44. Wavefront encoding inside Tiles..... 67

Figure 45. A parallel reduction scheme is used to calculate all blocks from the bottom up. Thirty-six PU block reduction calculations are executed in total, excluding the initial 4 × 4. 71
Figure 46. Enumeration of all CU and PU modes in a CTU. 71
Figure 47. Visualization of the four phases. Steps leaps are expressed to video pixels either horizontally or vertically inside the searching area. 72
Figure 48. (a) The simplified GPU Kernel program with major steps executed. (b) The way the host calls the GPU kernel and gets back the MVs and ruiCost for further processing. 74
Figure 49. WTP encoding with GPU 76
Figure 50. WTP + 4PhaseME average speed up in Class A sequences for QP = 32 (Traffic, PeopleOnStreet) versus default TZSearch. 79
Figure 51. WTP + 4PhaseME average speed up in Class B sequences for QP = 32 (BasketballDrive, BQTerrace, Cactus, Kimono, ParkScene) versus default TZSearch. 80
Figure 52. Average speedup increase using WTP + 4PhaseME versus original WTP (QP = 32, AMP = true, Modified CPU threads scheduling) 80
Figure 53. WTP + 4PhaseME average speedup versus original WTP, using the same GPU but with different CPU frequencies for QP = 37 (average of WTP_2 × 2, WTP_3 × 3, and WTP_6 × 4, CPU-A = 1.2 Ghz and CPU-B = @2.9 Ghz). 81
Figure 54. BD-rate percentage increase for 4PhaseMe (24 threads) versus default TZSearch (QP = 22, 27, 32, 37). 81
Figure 55. BD-PSNR difference for 4PhaseMe (24 threads) versus default TZSearch (QP = 22, 27, 32, 37, negative values show a quality drop). 82
Figure 56. The HEVC FME process. Square symbols represent the integer pixel search points; the circles denote the interpolated half-pixel search points (1st step of the FME) and triangles the interpolated quarter-pixel search points (2nd step of the FME). 85
Figure 57. Combining WPP (upper left) and tiles (upper right) to create a new parallel scheme (WTP) (bottom). 87
Figure 58. Encoding time with different setups. Fraction execution resolver (FER) achieves 42% encoding time savings compared to the same optimized setup without the FER extension ([WTP] 24 CPU threads, GPU 4PhaseME). Where [WTP] is used, bitrate and PSNR were re 89

Figure 59. The fraction execution resolver (FER) flow chart. 95
Figure 60. The preliminary fast test decision point (pFTDP) algorithm. 96
Figure 61. Class A sequences using FER with WTP speedup for QP = 32 and 24 threads versus default. 99
Figure 62. Class B sequences using FER with WTP speedup for QP = 32 and 24 threads versus default. 100
Figure 63. Increase of BD−Rate using FER with WTP for QP = 22, 27, 32, 37, and 24 threads versus default (positive values indicate bigger files). 100
Figure 64. Decrease of BD−PSNR using FER with WTP for QP = 22, 27, 32, 37, and 24 threads versus default (negative values indicate a quality drop). 101
Figure 65. The absolute speedup increments when FER is enabled/disabled in our GPU-IME accelerated WTP encoder compared with the default HEVC encoder benchmark results, i.e., Diff (FER enabled–FER disabled). 101
Figure 66. The PSNR overhead when FER is enabled in our GPU−IME accelerated WTP encoder. Positive values indicate a quality improvement, while negative values indicate a quality drop. 102
Figure 67. The bitrate overhead when FER is enabled in our GPU−IME accelerated WTP encoder. Positive values indicate bigger files. 102

List of Tables

Table 1. Speedup in all sequences (TP, WTP, 24 THREADS) 62
Table 2. Index array of the SAD. 75
Table 3. Test video sequences. 78
Table 4. Test video sequences. 98

Chapter 1

Introduction

1.1. Overview of digital video

Digital video is an electronic representation of consecutive images that are encoded using a digital video signal rather than an analog one. Digital video was first introduced in 1986 by Sony and D1 format. The recorded video was an uncompressed standard definition video signal in digital form, encoded at YCbCr 4:2:2 using BT.601 raster format with 8 bits. Later it was replaced by cheaper systems using a compressed video format, most notably Sony's Betacam. Before that, research departments of the British Broadcasting Corporation (BBC) and Bell Labs performed early experiments during the 1960s with digital video. Their goal was to minimize the distortion and noise of their television video feeds, sent over coaxial cable circuits or terrestrial microwave signals. Later, in the 1970s, many manufacturers of professional video broadcasting equipment such as RCA, Ampex, and Bosch, developed prototypes of digital videotape recorders in their research labs, but they were never released for commercial use. In 1988, Sony and Ampex co-developed and released the D2 digital videocassette format. This advanced video format recorded uncompressed video much like D1. The major difference was that it used a single-cable composite video connection which made it a perfect fit for the majority of television facilities at the time. Other examples of digital video formats were Ampex's DCT, the industry-standard DV and its professional variations, Panasonic's DVCPro, Sony's DVCAM, and Betacam SX using MPEG2 compression.

At the same time, commercial digital video products were expanded to adapt cheap non-professional equipment. The PICS Animation Compiler (PACo) from the Company of Science and Art in Providence, was one of the first digital video applications to run on personal computers. It was shipped in 1990 for Mac computers and the playback was possible for many platforms like Macs, Sun SPARCstations, and IBM-compatible PCs. Next year, Apple Computer released QuickTime followed by Microsoft with Audio Video Interleave (AVI) in 1992. The consumer digital

video increased rapidly with the introduction of the MPEG1 and MPEG2 playback standards and the widespread adoption of the DV format. The recorded footage could directly be transferred from the camera to the computer through a standard FireWire port. No external playback or recording equipment is needed. This simplified process, allowed non-linear editing systems (NLE) to be deployed widely on desktop computers with very low costs. The ever-growing eagerness by both home users and professionals to create video footage results in increased demands for storage capacity. High-resolution and video streaming, exacerbated this problem. Video Compression is the term used to define an algorithm for reducing the storage requirements of digital video content.

1.2. Video encoding

Video encoding is a term used to convert a series of individual images to a fluid video. Nonetheless, this same umbrella term is also used to describe processes relating to video file size reduction. This reduction in the video translates to benefits such as lower storage and transmission bandwidth requirements. Nowadays, video encoding term is interweaved with video compression. For a vast majority of use cases, compression is the most time-consuming part of processing video. For filmmaking and production reasons a video encoding process could also be used with uncompressed (raw) or semi-compressed (lossless compression) video data just to comprise a series of digital images in a known content representation format for storage or transmission. Uncompressed video requires a very high data rate which means high storage and network bandwidth capacity, but preserves the highest possible quality without loss of data or artifacts.

For common use, the video footage is always compressed. It uses a standardized video compression algorithm and a detailed technical specification document -known as a video coding specification- is used for compression or decompression (co/dec). A video application can use a codec to encode or decode the digital video material. This video material can be hosted in file containers. Some of the most popular file containers, include Mp4, FLV, MOV, MKV, AVI, and QuickTime. These video files are essentially a container that contains the video data and the instructions on how that video data should be processed. Thus, a video file container is just the layout shell for data produced or consumed by a codec. There is a clear conceptual difference between them. The codec compresses raw video and audio files between analog and digital formats and makes them smaller while the container hosts that data. A very efficient and common codec today is Advanced Video Coding (AVC) known as H.264 or MPEG-4 AVC. It is commonly associated with the distribution of high-definition content and is the compression schema used for Blu-ray discs as well as many streaming video applications on the internet.

MP4 mentioned before and AVC/H.264, are two different technologies that are related to video terminology. MP4 is the encoding format and AVC/H.264 is the video codec. AVC supports ultra-high-resolution video streaming (4K) but newer codecs like High-Efficiency Video Coding (HEVC) or Versatile Video Coding (VVC) are proposed as successors to this widely used codec. The compression concept remained the same, but the compression efficiency and image quality increased. HEVC, also known as H.265 and MPEG-H Part 2, offers considerably improved video quality at the same bit rate or from 24% to 50% higher data compression with the same video quality. It also supports resolutions up to 8192×4320, including 8K UHD, and 10-bit depth for higher color representation (1 billion colors unlike the primarily 8-bit AVC which has 16.7 million). The successor to HEVC is VVC also known as H.266 and MPEG-I Part 3. It was finalized on 6 July 2020 by the JVET (Joint Video Experts Group), VCEG working group of ITU-T, and MPEG working group of ISO/IEC JTC 1/SC29. VVC standard has about 50% better compression rate for the same video quality compared with HEVC. It supports resolutions up to 16K as well as 360^{o} video views. The expected encoding complexity is higher (up to ten times) in comparison with its predecessor. The decoding complexity is twice that of HEVC.

1.3. Motivation and Thesis Contributions

As the necessity of supporting ever-increasing demands in video resolution leads to new video coding standards, the challenge of harnessing their computational overhead becomes important. Such overhead stems not only from the increased image data due to higher resolutions but also from the coding techniques per se, that are introduced by each standard to improve compression. All modern standards in the field of video coding offer high compression efficiency but this is achieved by increasing the computational complexity of the encoding part. Ultra-High-Definition (UHD) videos, bring new encoding implementation schemes that are being recommended for CPU and GPUs parallelization. For example, High-Efficiency Video Coding (HEVC) was developed to tackle the 4K resolution era, achieving up to 50% more compression efficiency (for the same video quality) compared to its predecessor H.264/AVC. However, this performance improvement comes at the expense of increased coding time. To ameliorate the situation, the video coding standard provides coarse-grained parallelization primitives. An optimal solution requires a trade-off between quality and processing time. Today, most video standards focus on two goals: to improve compression performance by implementing parallel architectures to minimize encoding time and at the same time, achieve better quality for higher video resolutions.

In this thesis, we tackle both of the aforementioned aspects, namely, harnessing computational overhead through parallelism and exploiting the potential usage of side-channels hardware equipment like Graphic Processor Units (GPU) to keep video quality at high levels. Leveraging the existing GPU framework, enables efficient low-latency, low-overhead use of

processing resources. Graphic Processor Units have been at the forefront of high-performance computing due to their high peak performance, high-speed bandwidth, and efficient programming environments. So, as far as the first aspect (parallelism) is concerned, we focus on a new novel block-based parallelization pattern, that combines two existing parallelization primitives: tiles and wave front. Through reorganizing the execution order and optimizing the data resources in both multi CPUs and GPU cores, we proposed an efficient parallel environment that is fully compatible with the HEVC video standard output stream. In this way, the intensive components of the HEVC/H.265 encoder are vastly parallelized keeping at the same time the quality at very high levels, fulfilling this way and the second aspect of the video standards goal. In addition, we proposed an extra parallel optimization method, that skips the calculation complexity overhead of the motion estimation fraction part, when static blocks are met, with negligible video quality deterioration.

To summarize, this thesis was motivated by the following questions:

Q1) Is possible to combine multiple parallelization schemes to increase encoding speed?

Q2) Can a modern multicore GPU kernel algorithm reduce the encoding time and speed up the whole process?

Q3) Can we avoid unnecessary computations without significant quality loss to speed up the encoding process?

This thesis aims to propose algorithms and methodologies that contribute to Q1-Q3. Our focus is on the scenario where a multicore server is used with thread-level parallelism. Specifically, in Q1, we introduced a new parallel encoding scheme to effectively combine the best of two very well-known parallelization schemes, i.e., tile and wavefront parallelism. The primary encoding scheme used is Tiles, but internally our modified encoder enables a custom multi-threading wavefront encoding for each tile separately. This new encoding scheme is designed to achieve load balancing in two scenarios: (a) when the number of tiles and CPU cores matches by keeping all CPU cores busy until the end of the encoding process but without exploiting the internal wavefront parallelization due the CPU core exhaustion, and (b) when the number of available physical CPU cores is higher than the number of tiles, and engaging progressively the rest of the CPU cores when multiple block lines are invoked in the wavefront scheme sub-pattern inside each tile. Results with this upgraded encoding scheme and common test settings and benchmarks show a significant speedup over using the default parallelization schemes of the HEVC standard. The parallelization degree achieved by tile parallelism (TP) is bounded by the number of tiles used, while our new parallelization scheme (Wavefront per Tile Parallelism - WTP), can have more than one working thread per tile. Under a restricted number of processors, the maximum achievable speedup of TP is dictated by using a number of threads equaling the available processors. WTP can exploit a trade-off between compression rate and speedup by using a smaller scale tile grid compared to the one of TP and allocating more than one processor per tile. Finally, compared to

wavefront parallel processing (WPP), the speedup is expected to be higher in WTP since threads (equaling the number of tiles) can commence at once.

in Q2 we investigate the possibility of using GPU to accelerate some parts of the video encoder. After carefully reviewing and profiling the program, we proposed a fully parallel algorithm for the motion vector calculations that is proven to be the most intensive part of any modern video encoder. For this extension, we used the OpenCL framework to set concurrent GPU kernels to compute the calculations required for the integer motion estimation part (IME). Our novel GPU fast search algorithm manifested as “4PhaseFS”, is implemented as a high-parallel alternative to other non-parallel search methods. Despite the heavy parallelization in Q1, only one CPU thread can be used at a time. Mixing our high parallel WTP encoding scheme, which is CPU-based, with GPU for the IME part, we tried to balance the encoding load between CPUs and GPU. Results demonstrate that the GPU contribution in the original WTP can increase the speed by an average of 18%.

Finally, concerning Q3 we developed an algorithm that lets the encoder skip the fraction part of the motion estimation calculations when specific criteria are met by introducing a preliminary Fast Test Decision Point (pFTDP) function before GPU kernel execution. Our evaluations provide a greater than 1600% encoding time-saving at its peak in comparison with the default HEVC sequential mode and ideally a saving of greater than 2286% for still video frame sequences. The total average speedup for both Class A and Class B video sequences is x13.45. The gain of the algorithm itself is more than x3.9 compared with the same multithreaded setup environment in Q2.

1.4. Thesis Structure

This effort has a continuous approach with the goal of speeding up the encoding process while minimizing quality loss. All recommended approaches are implemented with the HEVC standard, and the HM 16.17 as a reference program. All further optimizations are built using C++ on top of the reference program.

In Chapter 2, the fundamental structures of a modern video codec are presented for understanding the different compression techniques, block partitioning, prediction-based algorithms, and the parallel mechanisms applied to improve the encoding times.

In Chapter 3, we introduce a groundbreaking parallelization pattern that combines two distinct and incompatible schemes from HEVC's video standard design. Specifically, we suggest and evaluate a multi-threaded approach for coarse-grained parallel video encoding, which is based on tile partitioning and per-tile wavefront parallelism. The Wavefront per Tile Parallelism

(WTP) we propose offers a valid trade-off between video coding efficiency and speedup by efficiently merging the best of both worlds, i.e., tile and wavefront parallelism.

The authors of the research paper, which was published in the Proceedings of the 15th International Workshop on Semantic and Social Media Adaptation and Personalization in Zakynthos, Greece in 2020 [1] recommend the use of the mentioned proposed parallelism scheme as an effective approach in assessing the conclusions drawn from the research.

In Chapter 4, we introduce a novel GPU-friendly Fast Search motion estimation algorithm, that can further improve the encoding performance, especially in a multi-threaded environment. We evaluated this algorithm with our proven WTP parallelization encoding scheme and is proven that its contribution to the encoding processing time is considerable. Results demonstrated, that the 4Phase GPU fast search motion estimation algorithm can achieve an average speedup of x8.84 for Class B videos and x9.60 for Class A compared with the default HEVC's sequential fast search algorithm. Also, results show that the GPU contribution in the original WTP introduced in Chapter 3, can increase the speed by an average of 18%. The mentioned speedups can further increase when low-frequency CPUs are used in conjunction with a powerful GPU without any significant quality loss.

The authors of the research paper, which was published in the MDPI Electronics, vol. 12, 2023 [2] recommend the use of the mentioned proposed Fast Search motion estimation algorithm as an effective approach in assessing the conclusions drawn from the research.

In Chapter 5, we introduce an optimization technique, that helps to avoid unnecessary calculations for the fraction part of the motion estimation, to boost the speedup times without any significant quality loss. In this work, we propose a fraction execution resolver (FER) algorithm that lets the encoder skip the fraction part when specific criteria are met by introducing a preliminary fast test decision point (pFTDP) function for the IME part. Through experiments with common test sequences, the algorithm is shown to provide a greater than 1600% encoding time-saving at its peak in comparison with the default HEVC sequential mode and ideally a saving of greater than 2286% for still video frame sequences. The total average speed-up for both Class A and Class B video sequences is x13.45. The gain of the FER itself is more than x3.9 compared with the same multithreaded setup environment that is presented in Chapter 4.

The authors of the research paper, which was published in the MDPI Electronics, vol. 12, 2023 [3] recommend the use of the mentioned proposed Fraction Execution Resolver algorithm as an effective approach in assessing the conclusions drawn from the research.

Finally, Chapter 6, presents our concluding remarks and avenues for future work.

Chapter 2

Video Codec Basic Structures & Conceptual Framework

2.1. Overview

Video codecs can be either software or hardware that compress and decompress digital video data. The term "codec" is short for "coder-decoder." These technologies use algorithms and mathematical transforms, such as the Discrete Cosine Transform, to eliminate redundant and unnecessary information while preserving essential visual details. This compression allows for storing, transmitting, and playing back video data on various devices with differing computing power and storage capacity. Many video codecs are available, including H.264/AVC, H.265/HEVC, VP9, LCEVC, AV1, VVC, and others still in development. In this Chapter, the essential background material that is necessary for understanding the basic structures of a video codec is covered. Furthermore, the conceptual framework of the encoding process is analyzed, including the parallelization features of the well-known HEVC and VVC video standards.

2.2. Color Space

Video color space refers to the way colors are represented and encoded in a video signal or file. It is a critical aspect of video technology, as it defines how colors are captured, stored, transmitted, and displayed in video systems. There are several common video color spaces used

in the industry, each with its own characteristics and applications. Some of the most notable video color spaces include:

1. RGB (Red, Green, Blue):

- RGB is an additive color model in which colors are represented by combining various intensities of red, green, and blue primary colors.
- It is widely used in computer graphics, displays (such as computer monitors and TVs), and digital cameras.
- Each pixel in an RGB image contains three color channels: one for red, one for green, and one for blue.

2. YUV (Luma, Chroma):

- YUV separates the image into two components: Y (luma) and UV (chroma).
- Y represents the brightness or grayscale information of the image, while UV represents the color information.
- YUV is commonly used in video compression formats like MPEG and H.264, as it allows for efficient compression of color information.

3. YCbCr:

- YCbCr is a digital representation of the analog YUV color space, commonly used in video compression and transmission.
- Y represents luminance (brightness), while Cb and Cr represent chrominance (color difference) components.
- YCbCr is used in video standards like JPEG, MPEG, and digital television (e.g., PAL and NTSC).

4. HSL (Hue, Saturation, Lightness) and HSV (Hue, Saturation, Value):

- HSL and HSV are alternative color models often used in image and video editing applications.
- Hue represents the dominant wavelength of color, saturation represents the intensity of the color, and lightness or value represents the brightness.
- These color spaces are intuitive for adjusting colors in post-production.

5. CMY and CMYK (Cyan, Magenta, Yellow, Key/Black):

- CMY(K) color spaces are subtractive color models used in color printing.
- They are based on the idea that colors are created by subtracting certain wavelengths of light from white light.
- CMYK includes a key (black) channel for better control over the printing process.

6. HSB (Hue, Saturation, Brightness):

- HSB is similar to HSL and HSV but uses brightness instead of lightness or value.
- It's used in some graphic design and image editing software for color manipulation.

The choice of color space depends on the specific application and requirements. For instance, RGB is common for display and editing, YUV for video compression, and CMYK for print. Converting between these color spaces is essential for compatibility between different devices and media. Modern video coding standards like HEVC or VVC take advantage of the YCbCr color space (luma, chroma: blue, chroma: red) to efficiently compress and encode video data while maintaining high video quality. The YCbCr color space, also known as YCC or YUV, is a color representation commonly used in digital video and image processing. It separates color information from luminance (brightness) information, making it particularly useful for video compression, transmission, and storage. The original RGB pixel data recorded by a camera or created by a video encoder is "subsampled" in a non-linear fashion corresponding more closely to the color and luminance sensitivity of the human eye (Equation 1).

$$Y' = 0.2627R' + 0.6780G' + 0.0593B'$$

$$C_b' = \frac{B' - Y'}{1.8814}$$

$$C_r' = \frac{R' - Y'}{1.4746}$$

R’ = Red
G’ = Green
B’ = Blue

Equation 1. Y'C'bC' definitio. The BT.2100 standard definition describes how the values for Y'C'bC'r are calculated from the original R'G'R' values. The luma component Y' contains mostly data derived from the Green component.

YCbCr components are described below with more details than previously aforementioned.

Y (Luma):

- Y represents the luminance or brightness component of the image.
- It carries the grayscale information, indicating the intensity of the image without color.
- The Y component is responsible for defining the overall image clarity and detail. Since human vision is more sensitive to changes in brightness than color, the Y component is typically encoded at higher precision and resolution compared to the chroma components.

Cb (Chrominance Blue):

- Cb represents the blue-difference chrominance component.
- It carries information about the blue color difference between the luminance (Y) and the blue channel.
- Cb helps represent the color and detail in the blue and yellow color range. This chroma component as already mentioned, is subsampled, meaning that it has a lower resolution than the Y component. This is because the human eye is less sensitive to small changes in color compared to changes in brightness.

Cr (Chrominance Red):

- Cr represents the red-difference chrominance component.
- It carries information about the red color difference between the luminance (Y) and the red channel.
- Cr helps represent the color and detail in the red and cyan color range. This chroma component is also subsampled like Cb for the same reason.

The separation of luminance and chrominance information in the YCbCr color space for video encoding is advantageous for several reasons:

i. Compression: Since the human visual system is more sensitive to changes in brightness (luminance) than to changes in color, video compression algorithms can allocate more bits to the luminance channel (Y) while reducing the chrominance channels (Cb and Cr) to achieve higher compression ratios without a significant loss in perceived image quality.
ii. Compatibility: YCbCr is used in many video standards, including MPEG, H.264, and JPEG, making it a common format for video transmission and storage. It allows efficient encoding and decoding of video data.
iii. Color Adjustments: In video editing and post-production, color adjustments can be made separately on the chrominance channels (Cb and Cr) without affecting the luminance (Y), providing fine control over color correction and enhancement.
iv. Broadcast and Display: YCbCr is used in broadcast and display technologies like PAL and NTSC, where it separates luminance and chrominance information for compatibility with older analog television systems.

When working with digital video, understanding and manipulating the YCbCr color space is crucial for tasks such as video editing, color correction, and video compression, as it offers efficient ways to manage both image quality and file size. HEVC and other modern video standards employ various techniques to efficiently encode and compress the YCbCr data, including intra-prediction, inter-prediction, and transform coding. By taking advantage of the separate YCbCr color channels and the differences in perceptual sensitivity to luminance and chrominance, HEVC can achieve higher compression ratios using chroma subsampling, while maintaining video quality compared to its predecessor, H.264 (AVC).

Chroma subsampling, also known as chroma sampling or chrominance subsampling, is a technique used in digital image and video compression to reduce the amount of data needed to represent color information accurately while preserving the perceived quality of the image or video. It is commonly denoted using a notation like "4:4:4," "4:2:2," or "4:2:0," where the numbers represent the ratio of chroma (color) samples to luma (brightness) samples (Figure 1 – top). The first number represents the number of luma (Y) samples while the two following numbers both refer to chroma. They are both relative to the first number and define the horizontal and vertical sampling respectively. For example, in a 4:2:2 format, there are four luma samples per block row. The second number (2) represents the number of chroma samples for the first chroma row. The third number (also 2) represents the number of chroma samples for the second chroma row of the block (Figure 1 - bottom). In a 4:2:2 format, there are two chroma samples for every four luma samples. In practical terms, this means that in a 4:2:2 chroma subsampling format, the chroma information is sampled at half the rate horizontally but keeps full vertical resolutions compared to the luma information. This results in a reduced amount of color information compared to luma information but still retains relatively high color fidelity, making it a compromise between full color (4:4:4 - uncompressed) and more aggressive chroma subsampling like 4:2:0. 4:2:2 is commonly used in professional video production and broadcasting because it strikes a balance between maintaining good color quality while reducing data rates compared to 4:4:4, which would require significantly more bandwidth and storage.

It is considered the highest quality level of subsampling and is particularly useful for applications where color accuracy and quality are crucial, such as video editing and broadcast video transmission. A signal with chroma 4:4:4 has no compression (so it is not subsampled meaning for every luma sample we have also a chroma sample) and transports both luminance and color data entirely. In a four-by-two array of pixels, 4:2:2 has half the chroma of 4:4:4, and 4:2:0 has a quarter of the color information available. The 4:2:2 signal will have half the sampling rate horizontally but will maintain full sampling vertically. 4:2:0, on the other hand, will only sample colors out of half the pixels on the first row and ignore the second row of the sample completely. In the 4:4:4 format, there is no subsampling. Each pixel has a full set of chroma samples (Cb and Cr), which means color information is sampled at the same rate as brightness. This provides the highest color fidelity but requires a lot of data, making it suitable for high-quality applications like professional video editing.

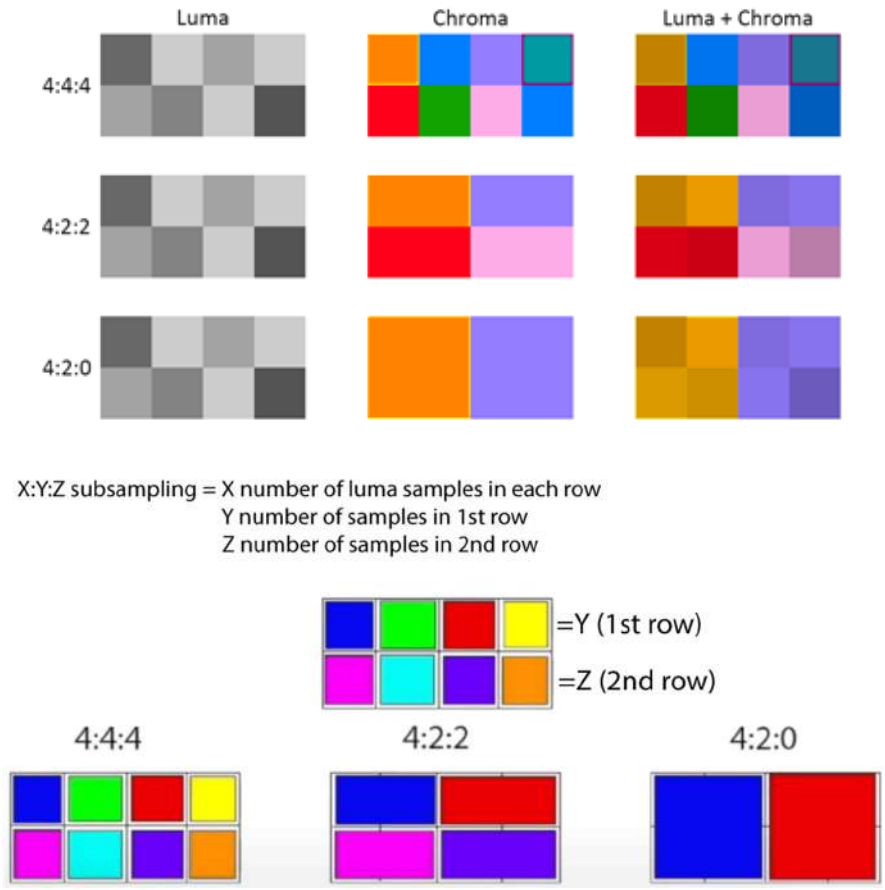


Figure 1. Chroma subsampling examples.

The choice of chroma subsampling depends on factors like the target application, available bandwidth, and storage requirements. Higher subsampling ratios like 4:2:2 and 4:4:4 are preferred in professional settings where color accuracy is critical. Lower subsampling ratios like 4:2:0 are common in consumer video and broadcasting due to their data compression benefits (Figure 2).

Figure 2. Chroma subsampling comparison. From top to bottom (a) 4:4:4 (b) 4:2:2 (c) 4:2:0

Chroma subsampling is an important consideration in video compression standards like MPEG, H.264, H.265 (HEVC), and others, as it can significantly impact video quality and data rates.

2.3. Video Compression

Video compression is a video encoding subprocess that aims to reduce the size of the video file while retaining as much high, video quality. The compression part of the encoder analyzes a group of video frames using a search algorithm trying to identify matching block areas. The redundancy block areas can be dropped or squeezed as much as possible. The challenge is to minimize the video file size while maintaining an excellent video experience. A compression algorithm typically involves an elision of carefully selected video data that are not considered critical to the viewers, and at the same time, avoids significant degradation of the image quality *[1]*. Modern video compression codecs offer various levels of compression profiles. The more aggressive the compression scheme, the smaller the file size, but the higher the image quality degradation, and vice versa (Figure 3). Compression artifact is a noticeable visual distortion, caused by high compression parameters manifested also with the terms, visual artifacts, pixelated edges, blurring, or blockiness [2]. The most common compression artifacts are DCT (Discrete Cosine Transform) blocks (Figure 4 and Figure 5). DCT is a lossy block compression scheme, that was first proposed by Nasir Ahmed in 1972. It is a hybrid coding algorithm because it combines two different compression techniques. The first one is DCT for spatial dimension i.e. in the same video frame and motion compensation in the temporal dimension i.e. references in the next or previous frames. In simple terms, the DCT coding takes a set of V_c correlated values (minimum arithmetic deviation) and returns V_d de-correlated values

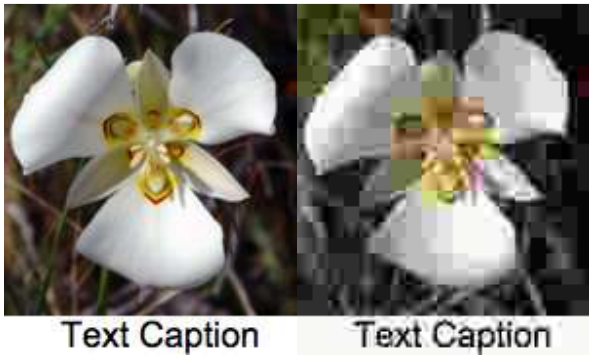


Figure 3. (a) Original uncompressed image (left). (b) Visual artifacts and image degradation, after aggressive video compression (right).

manifested as coefficients, in a such way that energy (a group area with similar arithmetic values) is compacted in only a few coefficients E_c where

$$E_c < V_c$$

In other words, DCT's primary job is to output (de-correlated) similar values in a compacted form (high energy areas - Figure 6) and minimize or even zero all others (discard values under a

threshold value). Thus, most of the input values are concentrated in only a few of the output values.

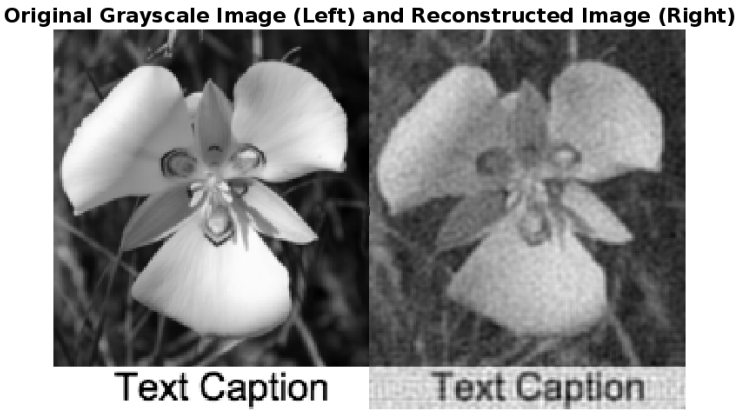


Figure 4. Image reconstruction discarding coefficients lower than 50 using Matlab.

```
>> Sego_lily_cm_150 = imread("Sego_lily_cm-150.png"); % import image
>> I = rgb2gray(Sego_lily_cm_150); % convert to grayscale
>> J = dct2(I); % compute the 2D DCT
>> J(abs(J) < 50) = 0; % discard certain coefficients (set to zero) < 50
>> K = idct2(J); % recover the pixels using the inverse 2D DCT
>> imshowpair(I,K,'montage'); % display the original and reconstructed image
>> title('Original Grayscale Image (Left) and Reconstructed Image (Right)'); % display captions
>> imshow(log(abs(J)),[]) % display image using a logarithmic scale
>> colormap parula % select color map
>> colorbar % display color bar
```

Figure 5. Matlab commands to apply DCT transformation of the grayscale image above.

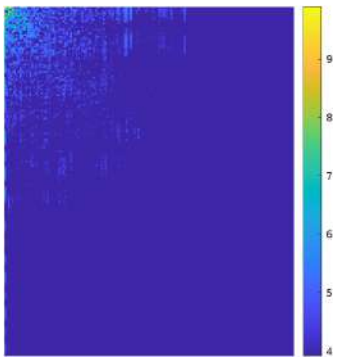


Figure 6. Displays the transformed grayscale image above using a logarithmic scale. Most of the energy is in the upper left corner (DC coefficient).

The most important coefficient is called DC and the rest of them -less important- that contain the image's finer details are called AC coefficients. In the example below (Figure 7) the

correlated (in this case, are more than correlated because they are identical) input values of 255 in the array on the left side, are energy-compacted in the top left element [0,0] of the right array after the DCT transformation. So, in the left array case, instead of saving 512 bits (8 bits x 64 elements), we save a value that only needs 12 bits (binary representation of the value 2040) plus some extra bits for the zero-value repetition (like Run-Length Encoding concept). In the transformed array, the DC coefficient is the 1st array element [0,0] and is considered the most important, while all others are considered AC coefficients.

The DCT transform is closely related to the discrete Fourier transform. The two-dimensional transform is equivalent to a one-dimensional DCT performed along a single dimension followed by a one-dimensional DCT in the other dimension [3]. The definition of the two-dimensional DCT for input image A and output image B is shown in Equation 2. The basic DCT encoding steps are depicted in Figure 8. The first step is usually to change color space from RGB to YCbCr because of the high correlation among the three components of the RGB model if this is the case. Usually, this applies mostly to images rather than video sequences that were originally recorded using the YCbCr color space. Source pixel samples are grouped into blocks and shifted from unsigned integers to signed integers. Then blocks are fed to DCT.

	1	2	3	4	5	6	7	8
1	255	255	255	255	255	255	255	255
2	255	255	255	255	255	255	255	255
3	255	255	255	255	255	255	255	255
4	255	255	255	255	255	255	255	255
5	255	255	255	255	255	255	255	255
6	255	255	255	255	255	255	255	255
7	255	255	255	255	255	255	255	255
8	255	255	255	255	255	255	255	255

	1	2	3	4	5	6	7	8
1	2040	0	0	0	0	0	0	0
2	0	0	0	0	0	0	0	0
3	0	0	0	0	0	0	0	0
4	0	0	0	0	0	0	0	0
5	0	0	0	0	0	0	0	0
6	0	0	0	0	0	0	0	0
7	0	0	0	0	0	0	0	0
8	0	0	0	0	0	0	0	0

Figure 7. (a) Uncompressed array (left) (b) DCT transformed array

$$B_{pq} = a_p a_q \sum_{m=0}^{M-1} \sum_{n=0}^{N-1} A_{mn} cos \frac{\pi(2m+1)p}{2M} \; cos \frac{\pi(2n+1)q}{2N}, \begin{cases} 0 \le p \le M-1 \\ 0 \le q \le N-1 \end{cases}$$

where

$$a_p = \begin{cases} \frac{1}{\sqrt{M}}, & p = 0 \\ \sqrt{\frac{2}{M}}, & 1 \le p \le M-1 \end{cases}$$

and

$$a_q = \begin{cases} \frac{1}{\sqrt{N}}, & p = 0 \\ \sqrt{\frac{2}{N}}, & 1 \le q \le N-1 \end{cases}$$

N and M are the column and the row size of A, respectively.

Equation 2. The definition of the two-dimensional DCT for input and output images

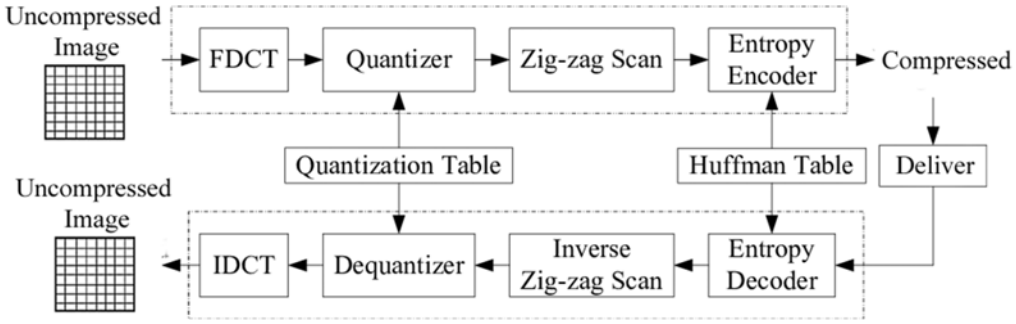


Figure 8 Basic DCT steps

DCT decomposes the discrete spatial frequencies input and outputs signal amplitudes (DCT coefficients) with the high energy signals to be concentrated at the upper left corner (important DC coefficient). DC coefficient has zero frequency in both dimensions. Most AC coefficients have zero or near-zero amplitude and need not be encoded. To achieve further compression all DCT coefficients are uniformly quantized using a quantization table which is specified by the application. Quantizer discards information that is not visually significant. Quantization is also a lossy process. Quantization is defined as the division of each DCT coefficient by its corresponding quantizer table reference, followed by rounding to the nearest integer (Equation 3). The output is normalized by the quantizer step size.

$$F^Q(u,v) = IntRound\left(\frac{F(u,v)}{Q(u,v)}\right)$$

Equation 3. Quantization equation.

Every encoder's final step is entropy coding. Entropy encoding is an extra compression scheme -lossless this time-, which creates an optimized lookup table of the actual data. Lengthy bit data streams with high-frequency occurrence inside the video frame, are replaced with symbols that have a smaller number of bits. But before the actual entropy coding takes place, a 2-substep coefficient preparation is involved. The first is the DC value assignment. The average value of all AC image samples is assigned to the DC coefficient. Thus, a strong correlation is usually built between the DC coefficients of adjacent blocks. Based on this concept, the new quantized DC coefficient is calculated as a difference from the previous block DC coefficient. This is known as Differential Pulse Code Modulation (DPCM), and the function for N=64 (8x8 block) is as depicted in Equation 4. DPCM can achieve further compression due to the smaller range of the coefficient values. The second subprocess is a coefficient ordering sequence. The AC coefficients are ordered into the "zig-zag" sequence (Figure 9), to help entropy coding to place low-frequency coefficients before high-frequency coefficients.

$$Diff\ (DC_i) = DC_i - DC_{i-1}, \begin{cases} DC_0 = 0, N = 64 \\ 0 \le i \le N - 1 \end{cases}$$

Equation 4. Differential Pulse Code Modulation (DPCM)

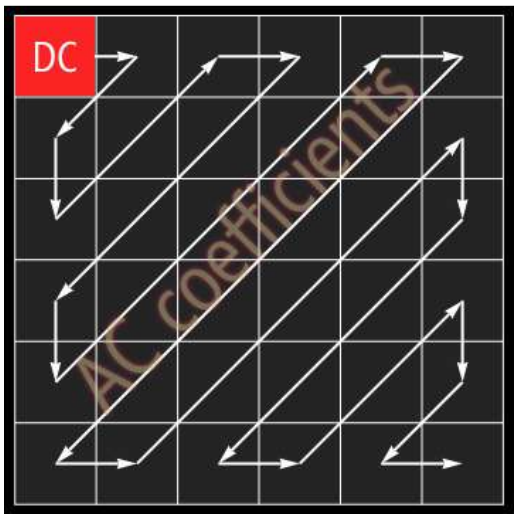


Figure 9. Zig-Zag scan order.

Finally as mentioned above, entropy coding is applied. The very basic concept of entropy coding schemes is based on codewords that are an integral number of bits like Huffman Encoding or Variable Length Encoding (UVLC) but is much more complicated with many key elements (Figure 10). Encoding involves organizing quantized transform coefficients in a two-dimensional order based on their expected efficiency, with the most efficient placed first. Next, an encoding

algorithm is applied to assign indexes and track the number of trailing zeros. The effectiveness of this method relies on the concentration of non-zero values and the sequence's structure.

Context-based adaptive binary arithmetic coding (CABAC) has been adopted as a normative part of the H.26 4/AVC standard; it is one of two alternative methods of entropy coding in this new video coding standard. The other one is the run length coder CAVLC (Context Adaptive Variable Length Coding) which has a lower complexity but is less efficient than CABAC. The latter is only available in main and high profiles. HEVC standard [4], supports only CABAC despite the higher computational cost. VCC standard [5] also supports CABAC with some improvements to increase efficiency such as better initializations without the need for a look-up table (LUT), and increased flexibility in Coefficient Group sizes. CABAC utilizes statistical data from previously coded binary symbols (bins) to efficiently code them. By estimating conditional probabilities of current bins based on this data, probability models are switched and intersymbol dependencies are exploited according to the frequency statistics of dominant bins in neighboring blocks. This adaptability allows the encoder to adjust to changes in symbol statistics. The bins are transmitted via arithmetic coding, which enables fractional bit allocation per symbol.

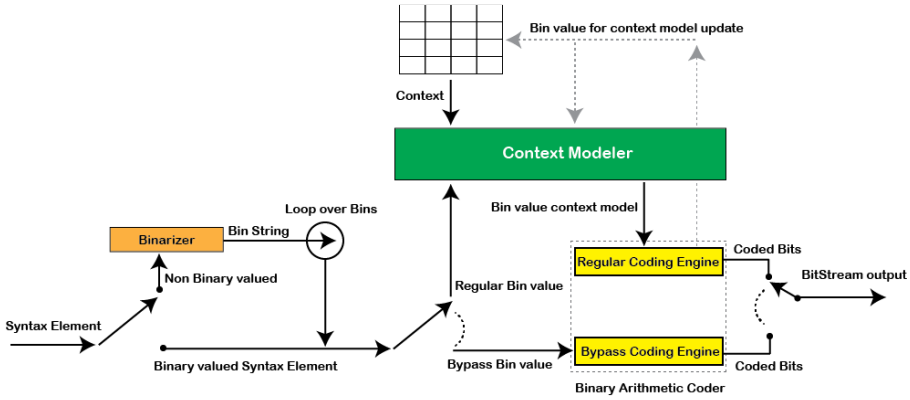


Figure 10. The design of CABAC involves the key elements of binarization, context modeling, and binary arithmetic coding. These elements are illustrated as the main algorithmic building blocks of the CABAC encoding block diagram, as shown above.

Figure 11 summarizes the DCT compression process steps in 9 steps. Step 0 is the input of the uncompressed image to be compressed. In step 1 each frame is separated into small blocks ie. 8x8 and is fed to DCT. Each color component of the YCbCr color space is separated to create each DCT block input. For simplicity's sake in this example, we only use a grayscale image. For a grayscale image, each pixel may have values from 0 to 255, where 0 is pure black and 255 is pure white. Step 3 depicts an 8x8 grayscale block with the actual pixel values. In step 4 we shift the image by -128 to center pixel values around zero. Then in step 5, a 2-D DCT energy compaction is applied. Step 6 is the sample quantization reference table -for this specific example- that is applied to the DCT table. The quantized DCT coefficients are depicted in step 7. At this step, DCT

successfully compacted the energy in a few coefficients only. Finally, all coefficients are ordered in a zig-zag order (step 8) before the entropy encoding takes place in the final step 9.

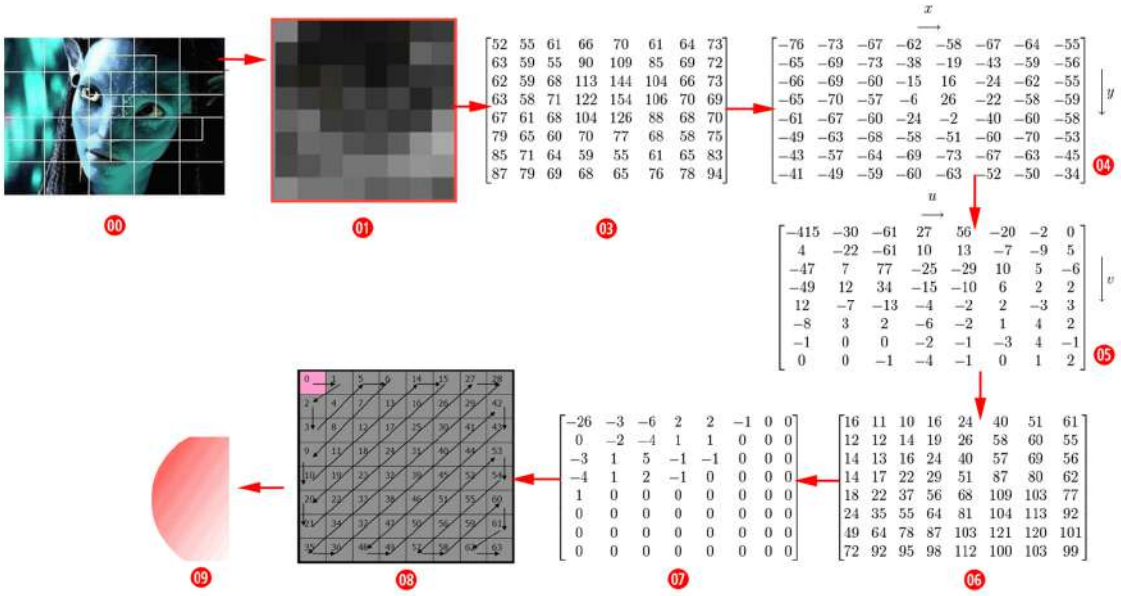


Figure 11. DCT steps.

An efficiency compression that uses DCT is highly dependent on the input block. Multi-video frame partition schemes are supported. In AVC video standard the partition blocks are always 16x16 and in HEVC and EVC-baseline (MPEG-5 Essential Video Coding Standard) are up to 64x64 (Figure 14). For EVC-main, VVC, and AV1 are up to 128x128. In video standards that support ultra-high resolutions, the block sizes are getting larger because they enable efficient encoding with smooth textures (Figure 13). While AVC had fixed block sizes (manifested as macroblocks), in HEVC the quad-tree was introduced (Figure 14). Quad-trees allow the Coding-Tree-Unit (CTU) to be recursively partitioned into four additional sub-blocks. EVC-Baseline implements the same scheme. VVC added Binary Tree (2-way) and Ternary Tree (3-way) splits to the Quad-Tree, thus increasing the partitioning flexibility. An evolving aspect of partitions is their shape flexibility. The blocks can split both symmetrically and asymmetrically allowing efficient and accurate partitions. Thus, the need for fine granularity of a sub-partitioning can be avoided and the "staircase" effect is diminished. The geometric partitioning of VVC and the wedges partitioning introduced in AV1 support diagonal partitions between two prediction areas, thus enabling very accurate partitioning.

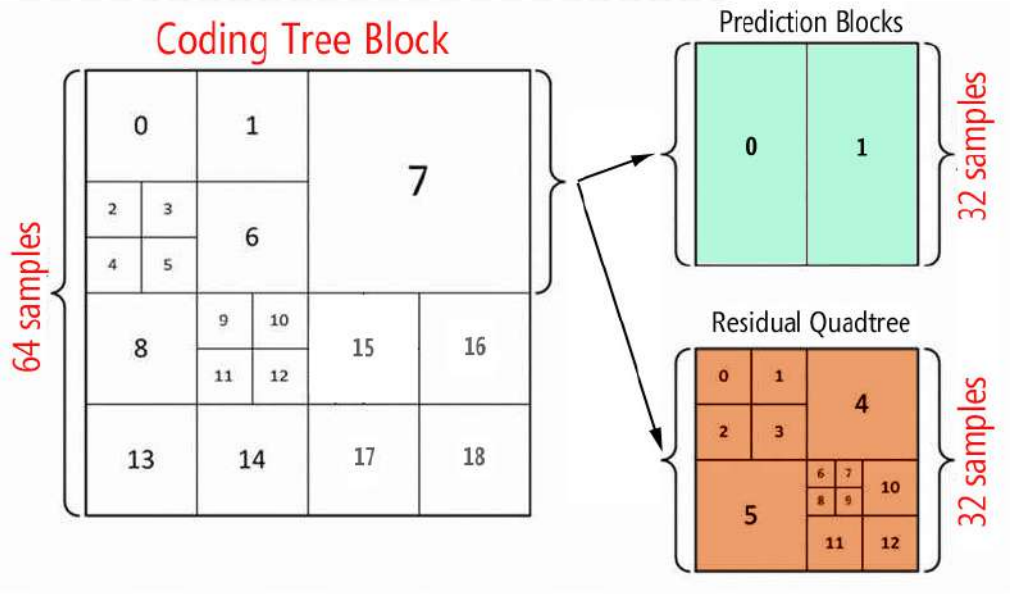


Figure 12. HEVC partition blocks are up to 64x64

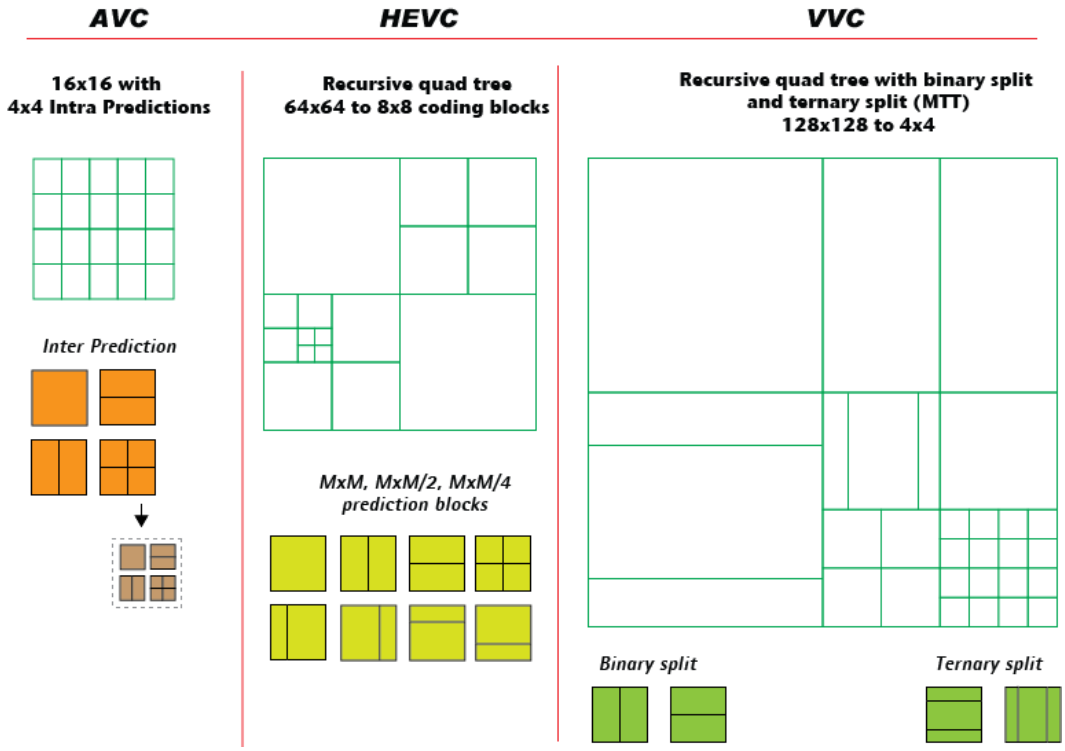


Figure 13. Video standards partition sizes. The bigger, the better [6].

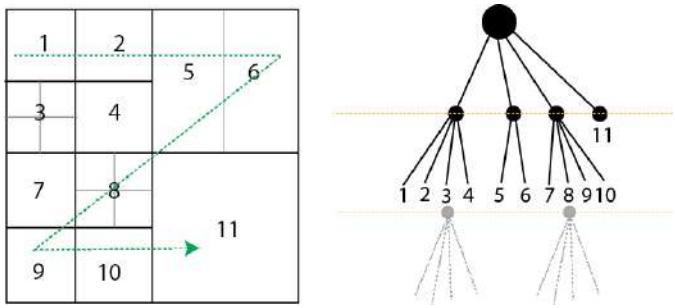


Figure 14. HEVC's highly variable quadtree-based structure.

2.4. Block Partitioning for Prediction

All ITU-T and ISO/IEC video coding standards since H.261 [6] follow the approach of block-based hybrid video coding, illustrated in Figure 15. In the first step, each picture in HEVC is subdivided into disjunct square blocks of the same size, each of which serves as the root of a first block partitioning quadtree structure. Prior video standards used macroblocks; HEVC employs Coding Tree Units (CTUs) instead, which are better at sub-dividing the image frame, into different-sized structures and can use larger block structures up to 64x64 pixels [7, 8]. Each CTU contains one luma Coding Tree Block (CTB) and two corresponding chroma CTBs (Figure 16).

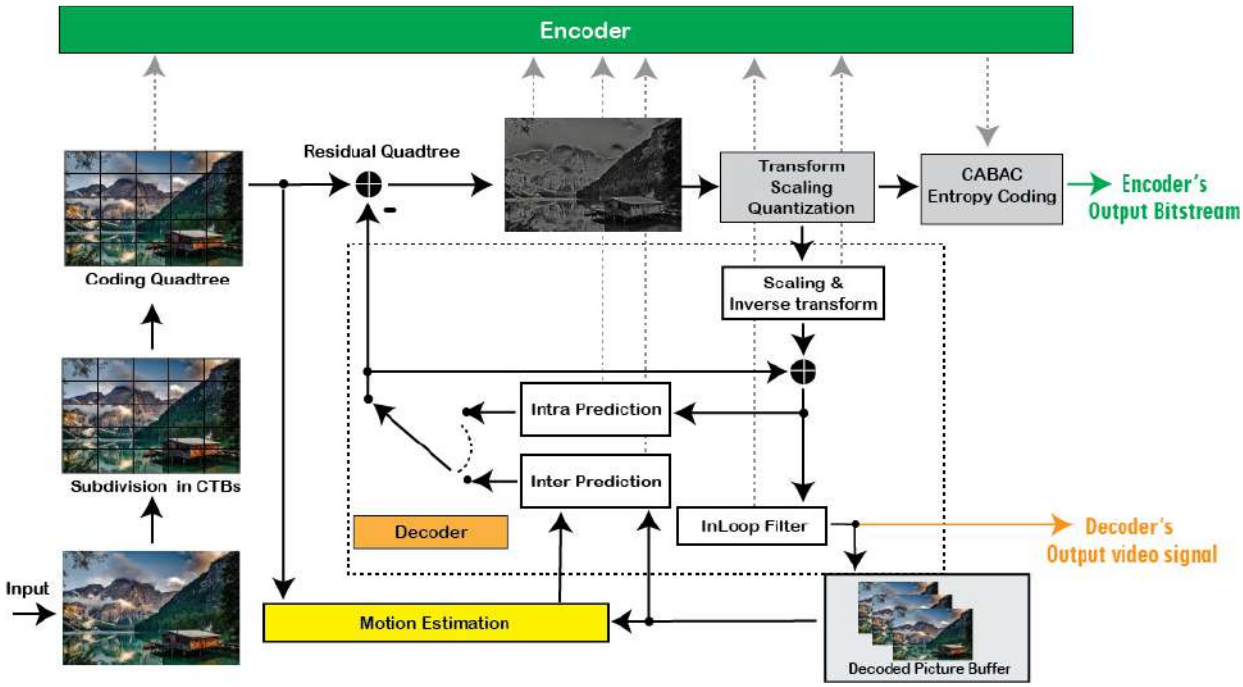


Figure 15. Block diagram of an HEVC encoder with built-in decoder (gray shaded)

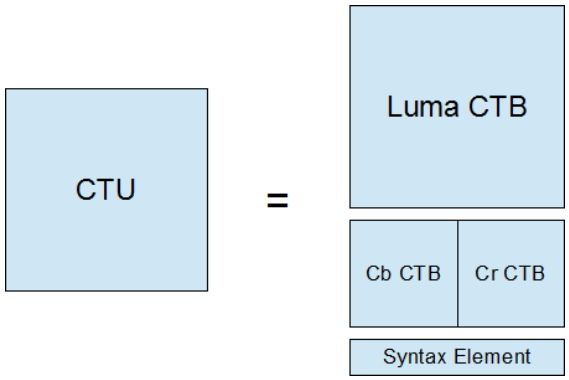


Figure 16. CTU – Coding Tree Unit is a logical unit.

The HEVC standard utilizes the term "Block" to denote an intended process for a particular region of the video frame buffer. On the other hand, if "Unit" is employed, it signifies a coding logical unit that will be encoded into an HEVC bitstream. Logical units, in addition, contain supplementary information i.e. Syntax Element which is crucial for the decoder. The CTUs can be further subdivided along the Coding Tree Structure into Coding Units (CUs) (Figure 17). CUs are the entities for which an encoder has to decide between intra-picture and motion-compensated prediction (Figure 18). A CTU in HEVC can have a pixel block size of 64x64, 32x32, or 16x16, with the coding efficiency often rising with larger pixel blocks. CTU size is the Largest Coding Unit (LCU) size. The LCU in other newer video standards (128x128 in VVC) is even bigger allowing better performance for higher spatial resolution videos. The concept of CTUs can be regarded as a concept of extended macroblocks -regarding the older AVC video standard- when the CTU size is not 16x16.

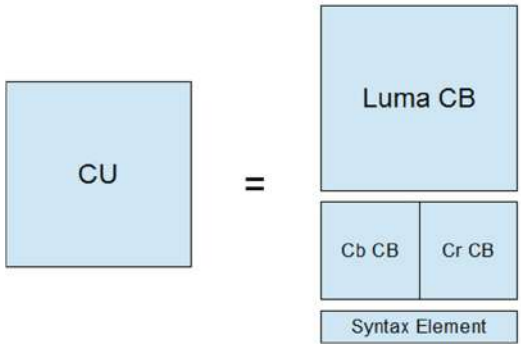


Figure 17. Each CTU can be differently split into multiple CUs (Coding Units) and each CU becomes the decision-making point of prediction type.

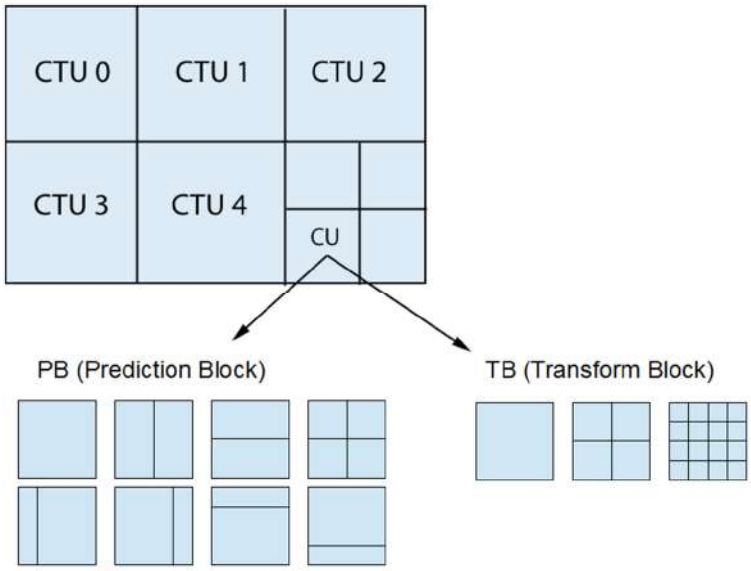


Figure 18. Block partition of HEVC

While CU may suffice for spatial or temporal predictions, it may not be optimal for storing motion vectors in intra- or inter-prediction modes. For instance, when a small object like a snowflake moves within an 8 by 8 CU, determining the appropriate MVs can be challenging depending on its location within the CU. To enhance compression efficiency, the Prediction Block (PB) was introduced. CUs can be broken down into smaller PBs depending on their predictability in temporal and/or spatial space. Once a prediction is made, the residual must be coded using a DCT-like transformation to calculate the disparity between the predicted and actual image. However, CUs may contain both low and high frequencies, resulting in an oversized block. Hence, each CU can be separated into smaller Transform Blocks (TB), which do not need to correspond with PBs. It is feasible, and often beneficial, to apply a single transformation to residuals from different PBs.

2.5. Prediction

The process of data compression involves removing unnecessary information that does not impact the accuracy of the data. Lossless compression works well for many types of data that contain statistical redundancy, as it results in a perfect replication of the original data upon decoding. However, when it comes to compressing images and videos, lossless compression only provides modest compression results. To achieve higher compression ratios, lossy compression is necessary. In a lossy compression system, the decompressed data will not be identical to the

source data, resulting in a loss of visual quality. When it comes to video compression, lossy systems remove subjective redundancy, which are elements that can be removed without significantly affecting the visual quality. Video coding methods make use of both temporal and spatial redundancy to achieve compression. Temporal redundancy is high between frames captured over time, particularly when the frame rate is high (Figure 19). Spatial redundancy is high between adjacent pixels in the same frame (Figure 20).

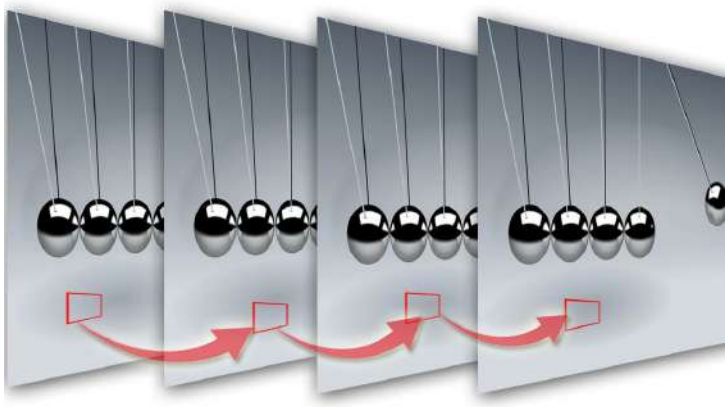


Figure 19. Temporal redundancy to the best match position in consecutive frames over time

Figure 20. Spatial redundancy takes advantage of similarity among most neighboring pixels

Spatial redundancy refers to the repetitive or duplicate information present in neighboring data elements within an image, video, or any other form of spatial data. It occurs when adjacent pixels contain similar or identical values or patterns. Intra-prediction is one of the key components of spatial compression, which aims to reduce data redundancy within a frame

without reference to other frames in the video sequence. The first step in intra-prediction involves dividing the frame into smaller blocks, typically square or rectangular, called macroblocks or coding units. These blocks are usually 4x4, 8x8, 16x16, or other sizes, depending on the video coding standard and the specific encoding settings. For each block, intra-prediction uses a set of prediction modes. These modes define how the pixels inside the block should be predicted based on the surrounding pixels. Common prediction modes include (a) *DC Mode*: In this mode, all the pixels in the block are predicted to have the same value, which is an average of the neighboring pixels. (b) *Vertical Mode*: The pixels are predicted by copying the values of the pixels in the column directly above the block. (c) *Horizontal Mode*: The pixels are predicted by copying the values of the pixels in the row directly to the left of the block and (d) *Diagonal Modes*: These modes involve predicting pixels in a diagonal direction based on the surrounding pixels. For each block, the encoder evaluates different prediction modes and selects the one that minimizes the prediction error. The prediction error is the difference between the predicted values and the actual values in the block. Once the prediction mode is selected, the encoder encodes the residual data, which is the difference between the original pixel values and the predicted values. This residual data is typically smaller in magnitude than the original pixel values, especially if the prediction mode is effective.

HEVC intra-prediction methods fall into two categories: angular prediction and planar prediction/DC prediction [7]. Angular prediction is ideal for modeling directional edges with precision, whereas planar prediction/DC prediction provides predictors that can estimate smooth image content. HEVC offers support for a total of 35 intra-prediction modes, as outlined in Figure 21. All of these modes rely on reference samples from adjacent reconstructed blocks, as demonstrated in Figure 22. Intra-prediction is executed at the chosen transform block size, which is consistent with the reconstruction process and ranges from 4x4 to 32x32 samples.

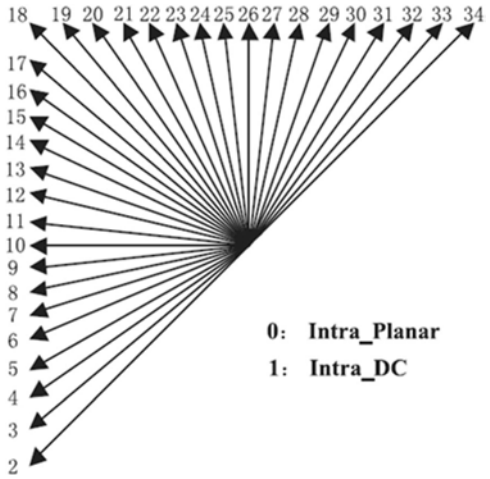


Figure 21. Intra-prediction modes of HEVC

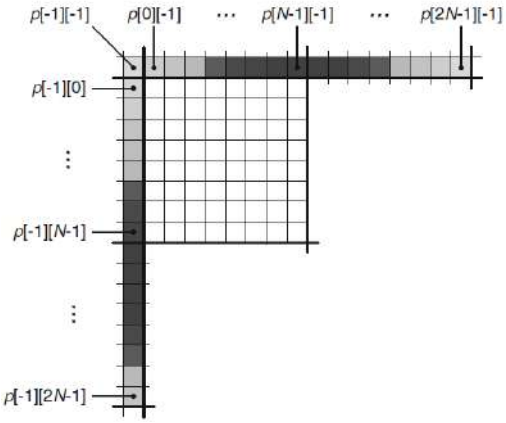


Figure 22. HEVC's reference samples for a block NxN

To improve the accuracy of prediction candidates, HEVC provides a range of filtering options to prepare reference samples before their utilization in the intra-prediction process. Moreover, specific prediction modes involve a post-processing step that enhances the smoothness of the sample surface at block boundaries. In DC prediction, the anticipated sample values are populated with a constant that signifies the mean of the reference samples to the left and above the block being predicted. The predicted luminance blocks, which are 16x16 or smaller, are subject to gentle filtering to create a smoother appearance on the edges to the left and above the block. Whereas, working with structures that have edges, angular prediction is an effective method, it may produce noticeable contouring in areas of smooth images. Additionally, utilizing DC prediction at low or medium bit rates can create blockiness in smooth image areas. To solve these problems, HEVC's planar prediction can produce a seamless prediction surface that doesn't have any abrupt changes on block boundaries. This is achieved by averaging a horizontal and vertical linear prediction on a sample basis (Equation 5).

$$p[x][y] = (p_{hor}[x][y] + p_{ver}[x][y] + N) \gg (log_2(N) + 1)$$
$$\begin{cases} p_{hor}[x][y] = (N-1-x) * p[-1][y] + (x+1) * p[N][-1] \\ p_{ver}[x][y] = (N-1-y) * p[x][-1] + (y+1) * p[-1][N] \end{cases}$$

Equation 5. Planar prediction calculation

On the other hand, the *temporal* model is designed to minimize redundant information by leveraging the similarities found in neighboring video frames. This is accomplished by predicting the current frame by analyzing one or more adjacent frames and then refining it by accounting for any disparities between them (motion-compensated prediction). The outcome of this approach is a residual frame (obtained by subtracting the prediction from the current frame)

and a collection of model parameters, most commonly represented by motion vectors that describe the nature of the compensation applied. This kind of prediction is called Inter-picture prediction and can be seen as an improvement from previous video coding standards, such as H.264/AVC. Inter-picture prediction utilizes temporal correlation between images to generate a motion-compensated prediction (MCP) for image sample blocks. MCP involves dividing a video image into rectangular blocks. Assuming uniform motion within each block and larger moving objects consisting of multiple blocks, a corresponding block in a previously decoded picture can be utilized as a predictor for each block. The MCP utilizes a translational motion model, as depicted in Figure 5.1, where the motion of the block in the previously decoded picture is denoted by a motion vector (MV) (Δ_x, Δ_y) relative (displacements) to the position of the current block (Figure 23).

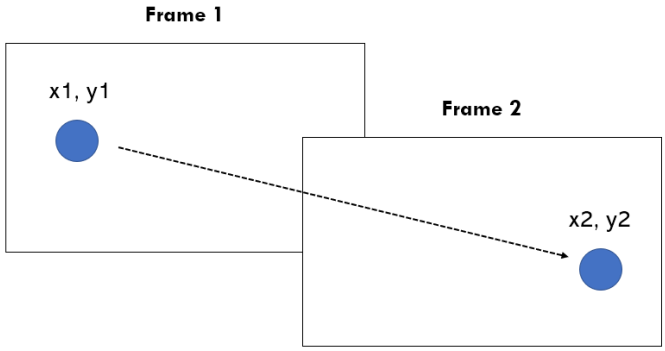


Figure 23. Motion Vector displacement Δx=x2-x1 and Δy=y2-y1 [9]

For greater precision in capturing an object's movement, these motion vectors can have fractional sample accuracy. When corresponding motion vectors have fractional sample accuracy, an interpolation algorithm is applied to the reference pictures to derive the prediction signal. The previously decoded picture is referred to as the reference picture and is identified by a reference index Δ_t to a reference picture list. These motion data parameters, including motion vectors and reference indices, are known as motion data. In modern video coding standards, two types of inter-picture prediction are allowed: uni-prediction and bi-prediction.

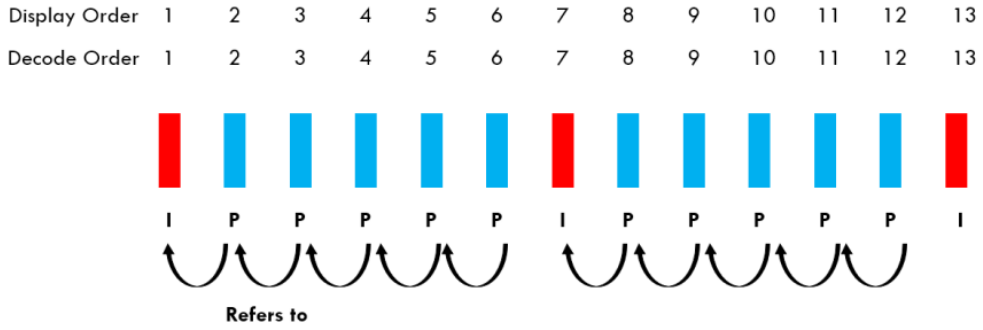


Figure 24. Uni-prediction example. Predicted Frame (P-frames) use frames that have been previously encoded [9].

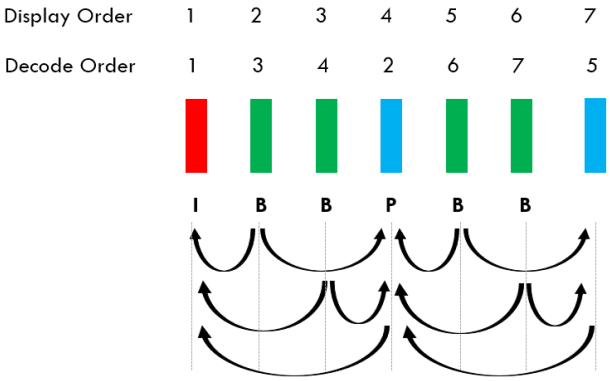


Figure 25. Bi-prediction example. Bi-prediction (B-frames) use frames that have been previously encoded or frames that are going to be encoded in the future [9].

Utilizing bi-prediction can significantly decrease the size of video files without compromising their quality. These frames can reference and interpolate from various frames that precede and follow them, making them incredibly efficient for video compression. By leveraging both spatial and temporal repetition, they effectively reduce redundancy in past and future frames. However, it's important to note that bi-prediction can increase computational complexity on both the encoder and decoder's resources. The High-Efficiency Video Coding (HEVC) standard employs an innovative motion vector prediction (AMVP) technique to enhance the accuracy of motion vector (MV) coding. This approach identifies the optimal predictor for each motion block and transmits that information to the decoder. HEVC also utilizes inter-prediction block merging, which gathers all motion data for a block from adjoining blocks, replacing the direct and skip modes utilized in H.264/AVC. As already mentioned, the motion vectors for HEVC are encoded in terms of horizontal (x) and vertical (y) components, as a difference from the motion vector

predictor (MVP). The equation Equation 6 illustrates the calculation of both motion vector differences (MVD).

$$\begin{cases} MVD_x = \Delta_x - MVP_x \\ MVD_y = \Delta_y - MVP_y \end{cases}$$

Equation 6. Motion Vector Difference (MVD) calculations for both x and y displacements.

The MVPs are typically obtained from decoded motion vectors in spatial or temporal neighboring blocks in the co-located picture. In the H.264/AVC video compression standard, the process of determining MVPs is achieved through the calculation of a component-wise median of three neighboring motion vectors. This method eliminates the need for signaling of the predictor. In addition, only temporal MVPs from a co-located picture are used in the temporal direct mode of H.264/AVC. This model is also utilized to derive other types of motion data beyond just the motion vectors. In High-Efficiency Video Coding (HEVC), the method of implicitly obtaining the Motion Vector Predictor (MVP) was replaced with the Motion Vector Competition technique. This new technique explicitly indicates which MVP from a list of MVPs is used for motion vector derivation [10]. The structure of coding quadtree blocks can sometimes lead to a situation where a block may have multiple neighbors that could be considered possible candidates for motion vector prediction (MVP). To address this issue, Advanced Motion Vector Prediction (AMVP) was introduced to adjust motion vector competition to accommodate the flexible block structure [11].

2.6. Block Matching Algorithm (BMA)

The Block Matching Algorithm (BMA) is a simple and widely used technique for motion estimation in video processing and computer vision [12]. It is used to estimate the motion vector between two consecutive frames in a video sequence. The primary idea behind the Block Matching Algorithm is to divide the frames into blocks and search for the best matching block in the reference frame to each block in the current frame. The displacement between the matched blocks provides the motion vector. Both the current frame (frame to be encoded) and the reference frame (the frame to which a comparison is applied) are divided into non-overlapping blocks of equal size. Common block sizes are 8x8 or 16x16 pixels, but this can vary depending on the specific application. For each block in the current frame, a search is performed in the reference frame to find the block that most closely resembles it. The goal is to find the block in the reference frame that has the minimum difference (often computed using a metric like Mean Squared Error or Sum of Absolute Differences) with the corresponding block in the current frame. Once the best matching block in the reference frame is found, the displacement between the centers of the current frame block and the matching reference frame block is calculated. This displacement is represented by a motion vector, which indicates how much the block has moved between frames (Figure 26). The above steps are repeated for all the blocks in the current frame,

resulting in a set of motion vectors that describe the motion between the two frames. The Block Matching Algorithm is relatively simple but computationally intense, making it inappropriate for real-time applications without extra hardware equipment or efficient algorithms. The Block Matching Algorithm (BMA) can be implemented with different search strategies, and one of the key distinctions is between the "full search" and "fast search" algorithms.

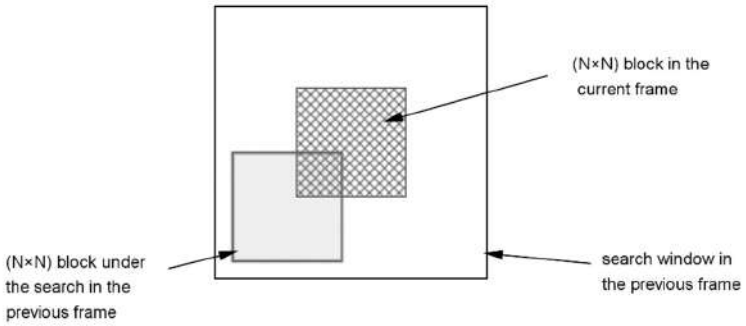


Figure 26. Mechanism of block-matching algorithm (Wikipedia)

$$SAD = \sum_{i=0}^{n-1}\sum_{j=0}^{n-1}\left|C_{ij} - R_{ij}\right| \;\; where \begin{cases} m,n = sizes\ of\ block \\ C_{ij}, R_{ij} = Current\ and\ Refference\ block \end{cases}$$

$$MSE = \frac{1}{N^2}\sum_{i=0}^{n-1}\sum_{j=0}^{n-1}\left|C_{ij} - R_{ij}\right| \quad where \begin{cases} m,n = sizes\ of\ the\ block \\ N = mxn \\ C_{ij}, R_{ij} = Current\ and\ Refference\ block\ position \end{cases}$$

Equation 7. The Sum of Absolute Difference (SAD) and Mean Squared Error calculations (MSE)

In the full search method, every possible displacement within a predefined search range is considered for each block in the current frame. This means examining all candidate blocks in the reference frame to find the best match for each block in the current frame. Full search tends to provide highly accurate motion vectors because it explores all possible displacements and finds the best match based on a chosen similarity metric (e.g., Mean Squared Error or Sum of Absolute Differences). The primary drawback of the full search method is its computational cost. It involves a large number of block comparisons for each block in the current frame, making it computationally intensive and practically not applicable for real-time applications or low-power devices.

On the other hand, the fast search method is designed to reduce the computational complexity by employing more efficient search strategies. Instead of examining every possible displacement, it focuses on a limited set of candidate blocks in the reference frame. While fast

search methods significantly reduce the computational burden, they may sacrifice some accuracy compared to full search. However, they often strike a balance between accuracy and computational efficiency. There are several fast search techniques, such as TZ Search, Three-Step Search (TSS), Diamond Search (DS), and Block-Based Gradient Descent Search (BBGDS), among others. These methods determine a smaller set of candidate locations to search, which helps in speeding up the motion estimation process.

In practice, all modern motion estimation algorithms use variations of fast search methods that incorporate heuristics and optimizations to further improve efficiency while maintaining acceptable accuracy. These methods are often more practical for real-world applications where both speed and quality are important. In the HEVC/H.265 video standard, a Test Zone Search (TZ) is adopted as a default fast search matching block which is proven to be very efficient in terms of speed and accuracy. Coding blocks (CBs) may be split into smaller prediction blocks (PBs), which are then inter-predicted. Each PB comes equipped with motion parameters such as motion vector (MV), reference frame index, and reference list flag. These partitions may be square or non-square, with non-square partitioning offering a more precise motion prediction by dividing a CB into two non-square PBs. Partitions for inter-coded CBs may be symmetrical or asymmetrical, and an asymmetric motion partition (AMP) can be used to improve the coding efficiency of irregular object boundaries without further division. The non-square quadtree transform (NSQT) combines square and non-square transform blocks (TBs) in a unified transform, supporting AMP albeit requiring additional decoder logic for non-square shapes. For each PB, one or two MVs can be transmitted, resulting in either uni-prediction or bi-prediction. The PB samples of an inter-predicted CB are obtained from a corresponding block region of the reference frame, which is offset horizontally and vertically by the MV.

2.7. Fraction Refining

When it comes to motion estimation, a "fractional motion estimation" or "sub-pixel motion estimation" is applied as a subsequent job of the integer motion estimation block matching algorithm. This process aims to refine the estimation by considering sub-pixel accuracy. The integer MVs represent the estimated displacement of an object or features from one frame to the next and this step increases even more matching accuracy. Instead of assuming that objects move only by integer pixel values, fractional motion estimation tries to find the precise fraction of a pixel by interpolating values between neighboring pixels (Figure 27). The MVs with fractional samples interpolated, produce prediction samples with non-integer sampling locations. Similar to AVC/H.264, HEVC/H.265 employs MVs with half and quarter sample precision between luma samples of the initial integer MVs. However, despite in AVC, in HEVC, quarter samples are generated from the integer luma samples via a longer filter, as opposed to relying on bilinear

interpolation based on the neighboring half and integer samples i.e. meaning extensions in H.265 allow high-precision weighted prediction that consequently increases the coding efficiency for at higher bit depths.

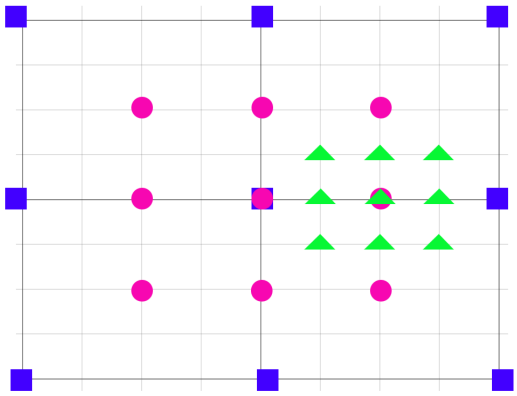


Figure 27. Fraction Estimation prediction. Square samples are integer pixel points. Circle samples are half-interpolated samples and triangles are quarter-pixel interpolated samples.

The use of a separable 8-tap filter for luma subsample locations simplifies the architecture and improves the accuracy of fractional sample interpolation. This, however, increases memory bandwidth and the number of multiply-accumulate operations for motion compensation. To minimize hardware cost, filter coefficients are limited to the 7-bit signed range. Unlike H.264 which supported both temporally implicit and explicit weighted prediction, only explicit weighted prediction is applied in H.265. The prediction is scaled and offset with values sent explicitly by the encoder. The bit depth of the prediction is then adjusted to the original bit depth of the reference samples. Uni-prediction involves rounding, right-shifting, and clipping the interpolated (and possibly weighted) prediction value to have the original bit depth. Bi-prediction, on the other hand, involves adding the interpolated (and possibly weighted) prediction values from two PBs, rounding, right-shifting, and clipping the result. Motion compensation in H.265 can be performed using only 16-bit storage elements, just like in H.264.

2.8. Quality

To evaluate and compare video communication systems with precision, it is vital to determine the quality of the images that viewers perceive. However, measuring visual quality can be a complex task due to the many variables that can impact the results, making it a subjective and imprecise process. Given the multitude of factors at play, achieving a completely accurate measure of quality can be quite challenging. Developers of video compression and processing rely

heavily on algorithmic quality measures, such as objective measures. While the Peak Signal to Noise Ratio (PSNR) is the most commonly used measure, its limitations have prompted numerous attempts to create more sophisticated measures that emulate the perception of human observers. The Peak Signal to Noise Ratio (PSNR) is a logarithmic measure, represented by Equation 8, which calculates the degree of distortion between the original image or video frame and an impaired one. The PSNR depends on the Mean Squared Error (MSE) between the measured images (original and impaired) and is relative to $(2n-1)^2$, where 'n' denotes the number of bits per image sample and signifies the square of the highest attainable signal value in the image.

$$PSNR_{db} = 10log_{10}\frac{(2n-1)^2}{MSE}$$

Equation 8. PSNR calculation

PSNR is a widely used quality measure for evaluating compressed and uncompressed video images. Its popularity can be attributed to its simplicity and speed of calculation. Figure 28-Figure 29 showcases five images - the original photo and four degraded (noisy) versions of it. The PSNR value for image (b) is 29.00 dB, indicating comparatively lower image quality than image (e) which has a measured PSNR of 5.29 dB.

Figure 28. PSNR showcases

```
originalImage=imread("originalImage.jpg");  % import original image
originalImage = rgb2gray(originalImage); % convert to grayscale
noisyImage=imread("noisyImage.jpg"); % import degraded image
noisyImage = rgb2gray(noisyImage); % convert to grayscale
[rows columns] = size(originalImage); %get dimensions
squaredErrorImage = (double(originalImage) - double(noisyImage)) .^ 2;
mse = sum(sum(squaredErrorImage)) / (rows * columns); % calculate MSE
PSNR = 10 * log10( 256^2 / mse); % calculate PSNR
fprintf('The mean square error is %.2f. The PSNR = %.2f', mse, PSNR); % display MSE and PSNR
```

Figure 29. Matlab commands to apply PSNR calculations of the grayscale image above.

Although the PSNR measure is a useful tool, it does have its limitations. To make comparisons, an unimpaired original image is required, which may not always be readily available

or easily verifiable. Additionally, PSNR does not always correspond well with subjective measures of video quality, such as those defined in ITU-R 500 [13]. While high PSNR values generally indicate high quality and low PSNR values usually indicate low quality for a given image or sequence, this doesn't necessarily reflect an absolute subjective quality. For example, a blurred background version of the original image may be introduced after the compression resulting in a lower PSNR relative to the original. However, most viewers would consider this image to be significantly better than the original image because the face is clearer, which contradicts the PSNR rating. This example illustrates that PSNR ratings don't always correlate with true subjective quality, as human observers may be particularly sensitive to distortion in specific areas and give them greater importance [14]. Unless there is a significant breakthrough in automatic quality assessment, accurate objective quality measurement is likely to remain a problem for some time.

2.9. Entropy Coding

The compression of data through entropy coding utilizes statistical properties to ensure that the number of bits used to represent data is proportional to its probability. Frequently used characters are represented by a few bits, while infrequently used characters are represented by many bits. Shannon's information theory [15] provides the optimal average code length for a character with probability p, which is $-log_2 p$. Entropy coding is performed in the final stage of video encoding and the first stage of video decoding. Syntax elements are used to describe the video signal's reconstruction process at the decoder, including prediction method, prediction parameters, and prediction error signal. In HEVC, syntax elements describe the coding tree unit (CTU), prediction unit (PU), and transform unit (TU), while H.264/AVC groups equivalent syntax elements together. Syntax elements for a CTU describe its block partitioning into coding units (CU), intra-picture or inter-picture prediction, quantization parameters for the CU, and sample adaptive offset (SAO) in-loop filtering type and offsets. Syntax elements for a PU describe its intra-prediction mode or motion data, while those for a TU describe the residual signal in terms of frequency position, sign, and magnitude of quantized transform coefficients.

Modern video standards, such as VVC and HEVC, commonly use Context-Based Adaptive Binary Arithmetic Coding (CABAC) [16] as an advanced entropy coding method. However, for slower devices, a simpler alternative manifested in AVC/H.264 standard as Context-Adaptive Variable-Length Coding (CAVLC) may be used to improve performance. Although CAVLC offers lower implementation and complexity costs, it sacrifices compression efficiency compared to CABAC. In earlier versions of the HEVC test model HM 1.0 [17], a low-complexity entropy coding method called LCEC was introduced to upgrade CAVLC. However, CABAC proved to be superior in compression efficiency, and LCEC was abandoned. The CABAC design involves three main components: binarization, context modeling, and binary arithmetic coding. In the binarization

stage, a binary string is generated for each non-binary valued syntax element. Context modeling estimates the probability of each non-bypassed bin based on specific context, such as MVs and transform coefficients which may employ different models. Finally, arithmetic coding compresses the bins to bits based on the estimated probability, which was updated during the previous context-modeling stage. Although CABAC requires more computational power than other entropy encoding schemes, it achieves better compression efficiency. The strategy behind the CABAC coding technique involves effectively coding non-binary syntax element values in a block-based video coder. This includes components such as motion vector differences and transform coefficient level values. The process involves a binarization scheme that maps syntax element values to binary symbols, also known as "bins." This scheme is used as a preprocessing unit for subsequent stages of context modeling and arithmetic coding. The design of binarization schemes in CABAC for all modern video standards is based on a few elementary prototypes that enable fast implementation and represent suitable model probability distributions.

2.10. Parallelization

Parallelization in video encoding refers to the process of dividing and processing video encoding tasks across multiple processing units (e.g., CPU cores or GPU cores) simultaneously. This technique is employed to improve the speed and efficiency of video encoding, especially when dealing with high-resolution videos and real-time encoding requirements. The choice of parallelization method depends on the specific video encoding software, hardware, and requirements. It's essential to balance the trade-off between encoding speed and compression efficiency, as some parallelization methods may sacrifice compression performance to achieve faster encoding times. Suggested parallelization video encoding techniques for HEVC include (a) slice parallelization (b) tiling and (c) wavefront using either multiple-threading CPUs or Graphics Processing Units (GPUs)

Slices

A single frame can be divided into multiple segments, each containing either the entire frame or a portion of it. A slice is made up of a sequence of CTUs that are decoded in a specific order (Figure 30). Each slice is self-contained and can be transported in a separate unit. This ensures that prediction data, residual data, and entropy coding are not performed across slice boundaries, enabling independent parsing and decoding of slices within a frame. However, when in-loop filtering is applied to a frame, information across slice boundaries may be necessary. Variable length codes can cause synchronization issues in the event of data loss in the bitstream.

While H.264 utilizes a unique resynchronization marker, H.265 employs slices as data structures to aid in resynchronization and error resilience. The number of CTUs in a slice can vary depending on the slice, and a slice breaks the prediction dependency at its boundary, potentially leading to a loss of coding efficiency and visible artifacts at the edges.

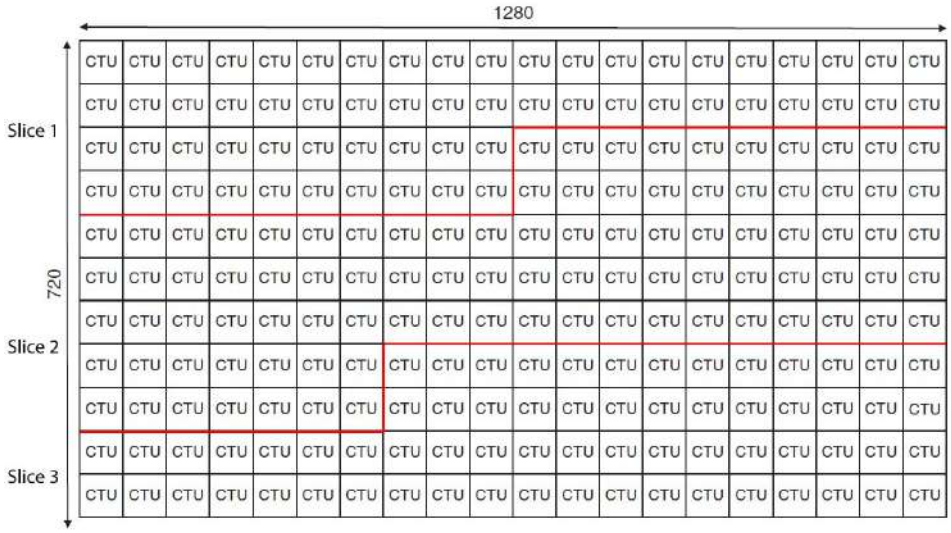


Figure 30. Slice encoding

Tiles

Tiles offer greater flexibility than slices when it comes to supporting parallel processing. By dividing a frame into a grid of rectangular regions separated by vertical and horizontal boundaries, each containing a group of CTUs, tiles can be used to encode and decode multiple regions simultaneously (Figure 31). The number of tiles is limited by a specific level. When multiple tiles are encoded in the same slice, they can be independently processed, allowing for faster encoding and decoding. In addition, tiles can be used to reduce the encoder buffer size by operating on smaller sections of the frame. They also enable the use of multiple rectangular sources (e.g., picture-in-picture applications) within a frame, which can be encoded independently. Tiles provide random access to specific regions of a video frame, and their primary purpose is to facilitate parallel processing rather than provide error resilience. They address hardware bandwidth restrictions between the codec's processors or cores and software-based architectures where processors or cores are difficult to synchronize at a CTU level without incurring delay or processor underutilization. By enabling parallel-processing architectures for encoding and decoding without the use of complex synchronization threads, tiles represent an important tool for video processing.

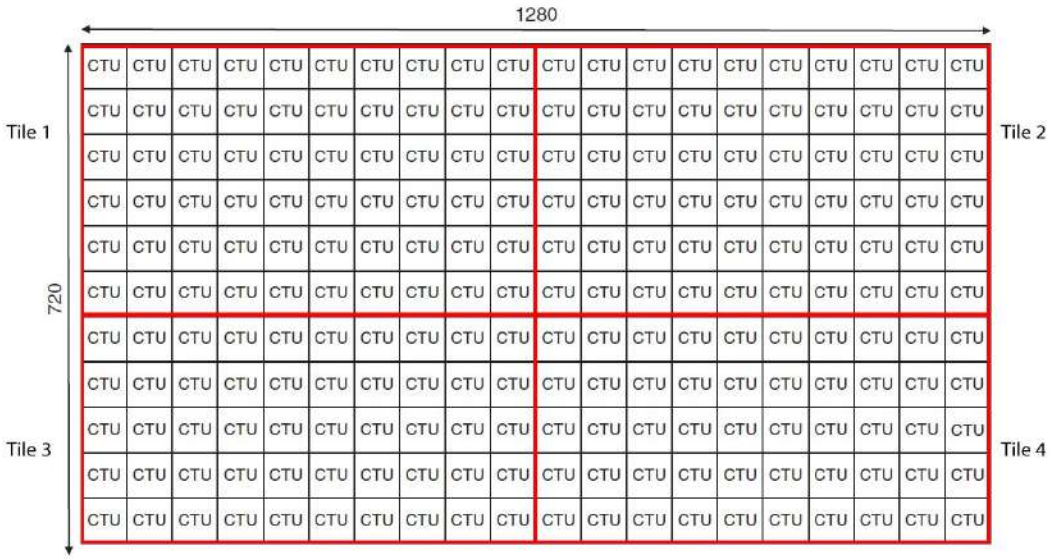


Figure 31. Tile encoding

Wavefront Parallel Processing

Both slices and tiles can impact coding performance due to prediction dependencies that may cross boundaries, and statistics used in context-adaptive entropy coding must be initialized for every slice or tile. However, Wavefront Parallel Processing (WPP) overcomes these issues by enabling parallel encoding or decoding at the slice level while maintaining prediction dependencies and leveraging as much context as possible in entropy encoding. Wavefronts divide a frame into rows of CTUs, allowing each row to be processed by a different thread as soon as two consecutive CTUs have been processed in the row above. This preserves coding dependencies between rows, except for the CABAC context state, which is reinitialized at the beginning of each CTU row. A shift of two CTUs is enforced between consecutive rows of CTUs that are processed in parallel, so the intra and inter-prediction of a CTU may require the top and top-right CTU to be available (Figure 32). Therefore, WPP requires additional inter-core communication compared to tiles. The first row of CTUs is processed conventionally, but subsequent rows require the entropy encoder to use data from the preceding row of CTUs. The two-CTU processing delay allows the entropy coder in each row to infer context models from CTUs in the preceding row. This enhances compression efficiency because the context state is inherited from the second CTU of the previous row instead of performing a CABAC reinitialization. Each row of CTUs can be assigned to a processing thread in the encoder or decoder, and multiple threads can be activated to enable parallel processing. WPP may achieve better compression than tiles avoiding visual artifacts that may be induced by using tiles because preceding data is used for entropy encoding. WPP does not affect single-step processing functions, such as entropy coding, predictive coding, and in-loop

filtering. Wavefronts cannot operate in conjunction with tiles for design simplicity, but tiles or wavefronts may use slices.

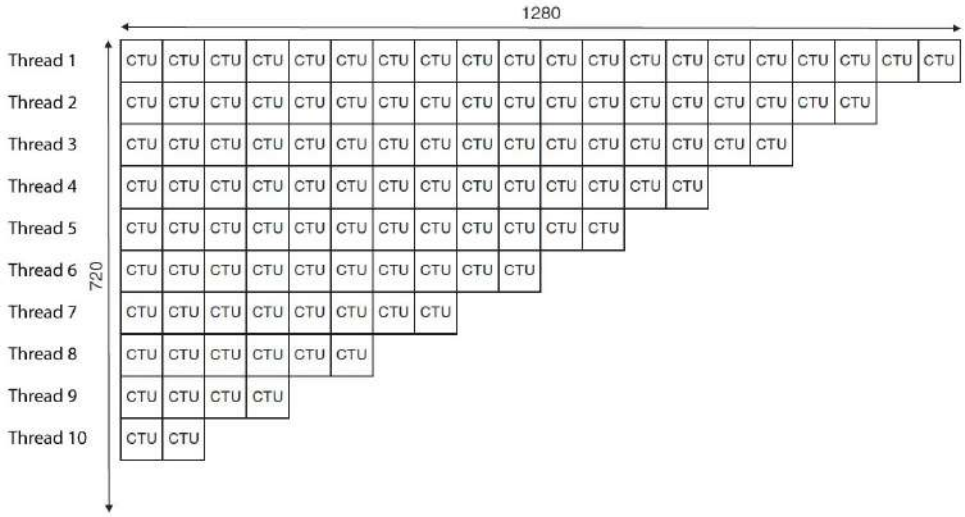


Figure 32. Wavefront encoding

Video encoding or decoding can be accomplished on devices with limited processing capabilities such as smartphones and tablets without the need for parallel processing. However, parallel processing through the use of tiles and wavefronts is also a viable option. With tiles and WPP, encoders or decoders can allocate multiple cores or processors to specific spatial parts of the frame for simultaneous processing. This can be achieved by using a table of offsets to identify the starting position of each tile or slice in the bitstream, resulting in minimal overhead. When implementing tiles in a single-core setup, there is additional overhead due to more complex boundary condition checking, resetting CABAC for each tile, and performing optional filtering of tile boundaries. Operating on a subregion of the frame enables improved data access. In multicore wavefront implementation, additional storage is needed to save the CABAC context state between CTU rows and perform a CABAC reset at the beginning of each row using this saved state. The overhead in multicore implementation is linked to memory bandwidth. In both tile and wavefront implementations, each processing core can decode any tile with minimal inter-core communication or synchronization. However, performing in-loop filtering across tile boundaries presents a challenge. Wavefront implementation requires greater communication and synchronization between cores, making it less efficient when compared to tile-based alternatives. When employing a wavefront implementation, the greatest potential for parallel enhancement is hindered by the duration it takes for all cores to be optimally utilized and an escalated possibility of dependency-related delays between CTU rows. For UHD/HD videos, the utilization of high-level parallelization tools proves to be more effective, and mandating a minimum number of frame partitions for larger frame sizes ensures a baseline level of parallelism for the decoder.

Chapter 3

Tile-Based Wavefront Parallelism in HEVC

3.1. Overview

Video encoding is arguably a computationally demanding process and for this reason, parallelization techniques have been considered since the 90s [19]. In HEVC [20] a frame is partitioned into pixel blocks of maximum size 64×64 referred to as Coding Tree Units (CTUs). Both tiles and wavefront were introduced in HEVC as novel block-based parallelization primitives to supplement slices that existed in H.264/AVC [21]. The three primitives jointly offer different levels of block parallelism. Since slices also act as network transmission granules (involving high bitrate overhead) most research on block-based parallelism in HEVC focused on wavefront and tile parallelism.

Tiles are defined as groups of CTUs after splitting the frame into M horizontal zones, each comprised of consecutive CTU rows, and N vertical zones comprised of consecutive CTU columns. Within a tile, the coding process follows a raster order on the contained CTUs. Dependencies are broken on tile boundaries; thus, each tile can be processed independently. Therefore, the ideally achievable parallelization level assuming a one-on-one tile-thread-processor assignment equals the number of tiles (MN upon which the frame is partitioned. HEVC places no restrictions on the tile size, other than it should comprise at least 4 CTU sub-columns. For a 1920×1080 resolution, this means that 119 tiles can be defined and processed by separate threads, a parallelization potential that meets and even exceeds the number of physical processing cores currently found in most CPUs or medium-sized servers. Nevertheless, the fact that dependencies are broken on tile boundaries results in coding losses which cannot be ignored.

In Wave Front Parallel Processing (WPP) [22] prediction dependencies are maintained. These dependencies dictate that to compress a CTU resting at row *i* and column *j* (let CTU_{ij}) the coding

process of its left CTU ($CTU_{i(j-1)}$), top ($CTU_{(i-1)j}$) and top right ($CTU_{(i-1)(j+1)}$) must have finished. By respecting such precedence constraints, per CTU row parallelization is possible as shown in Figure 33 (processed CTUs shown in grey). Since WPP respects prediction dependencies, any compression loss primarily stems from the maintenance (and possibly resetting) of different CABAC instances per CTU row. This loss was shown to be small [23] making wavefront the preferred parallelization method when losses should be minimized. However, the precedence constraints in WPP limit the achievable speedup. Furthermore, it is straightforward that the degree of parallelization (i.e., the maximum number of threads that can work concurrently) is dictated by the aspect ratio and in any case cannot exceed the number of CTU rows. For the example of Figure 33 the maximum parallelization degree achievable by WPP is 15, whereas by comparison with Tile Parallelism (TP), it can go as high as 119. The different trade-offs between compression efficiency and parallelization degree achieved by WPP and TP provide the motivation for this work. Namely, by defining a comparatively small number of tiles and processing each tile using wavefront parallelism, we can overcome the speedup limitations from WPP without suffering the level of compression losses of an equivalent TP scheme. The resulting method called Wavefront per Tile Parallelism (WTP) is graphically depicted in Figure 34.

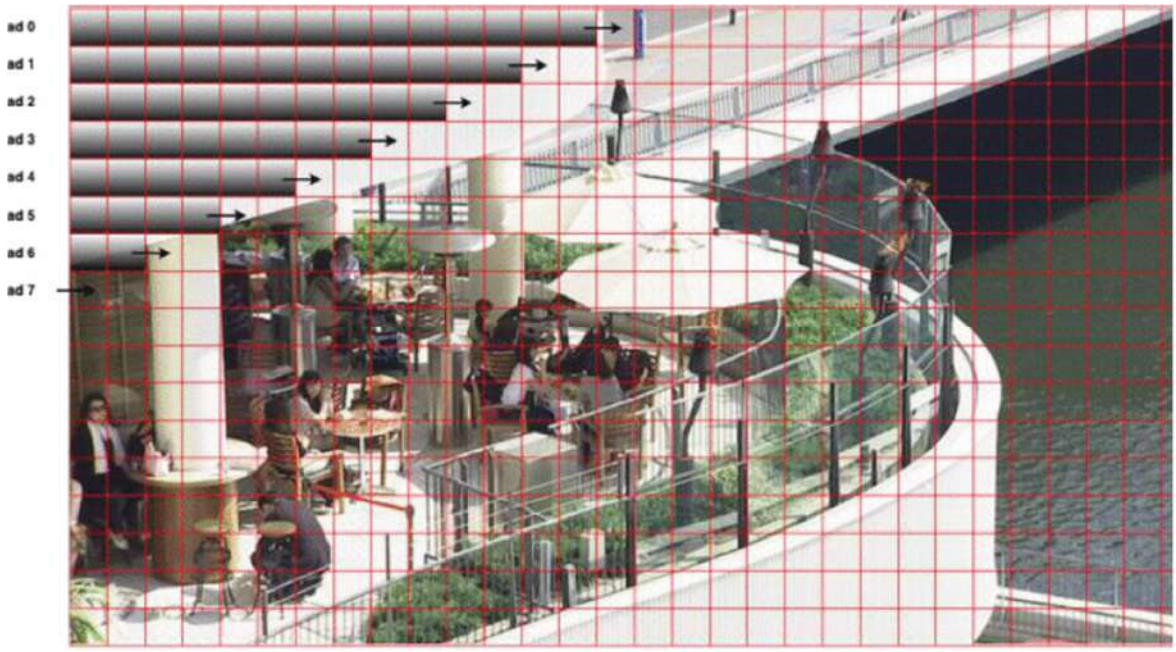


Figure 33. Example of wavefront parallelism (WPP), 8 threads.

Although HEVC does not allow signaling both tiles and wavefront to the decoder (a related patent can be found in [24]), the potential benefits of using WTP at the encoder side (signaling only tiles) cannot be overlooked. To the best of our knowledge, this is the first work characterizing

the potential of WTP as a technique for parallel video encoding and this is the primary contribution of the chapter. The rest of the chapter is organized as follows. Section 3.2 discusses the related work. Section 3.3 discusses the theoretical speedup of WTP. Performance evaluation is done in Section 3.4. Finally, Section 3.5 concludes this chapter.

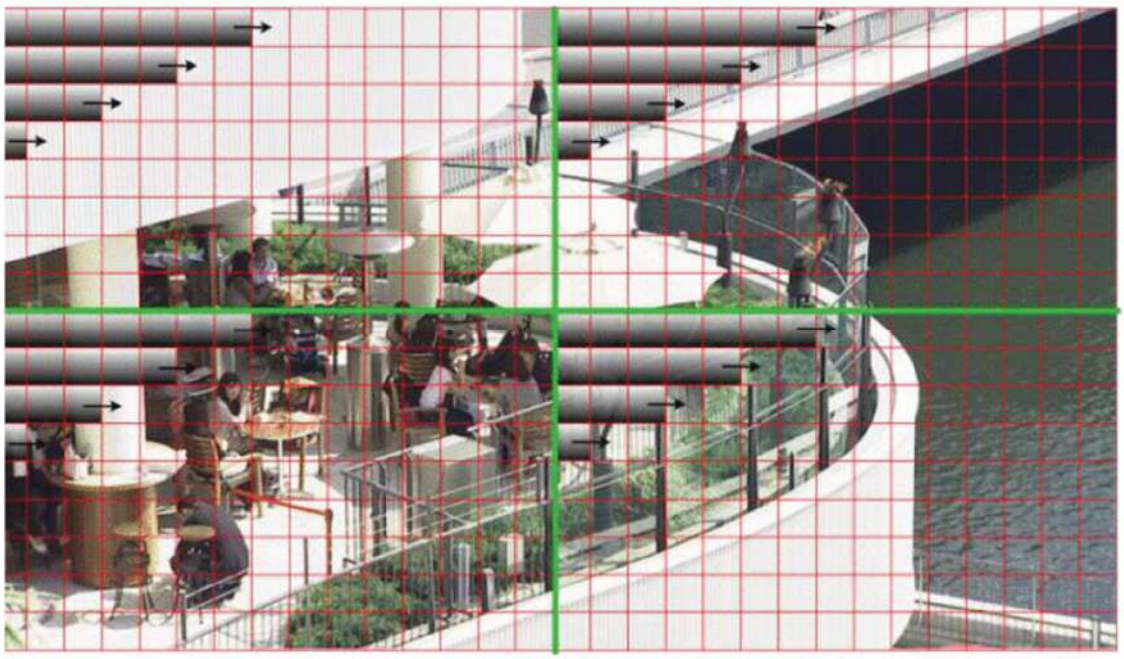


Figure 34. Example of proposed Wavefront per Tile Parallelism (WTP), 2×2 tiles, 16 threads.

3.2. Related Work

The basic WPP scheme was proposed and implemented in H.264 by the authors of [22]. In [25] the authors proposed a parallel approach for intra-encoding in HEVC. The approach was based on the wavefront processing of CTUs. Instead of maintaining different entropy coders for each CTU line (as WPP does) they advocated the use of a single entropy coder. The entropy coder itself is launched as a separate thread and works concurrently with the wavefront processing threads in raster order, producing the final bit stream. Thread assignment issues in WPP were discussed in [26], whereby the authors evaluated two different strategies. The first was to assign a complete block row at each thread while the second one was to assign single blocks. In [27] the authors detailed implementation details for parallelizing WPP in HM 11.0. They advocated that to reduce thread synchronization overhead, each thread should keep separate deep copies of data structures related to CU encoding.

Aside from implementation issues, another research direction on wavefront parallelism was the exploitation of inter-frame CTU prediction dependencies. In [23] OWF (Overlapping Wave Front) was proposed and tested with a focus on the decoding side. In OWF, whenever a thread finishes its assigned row and all remaining rows of the frame are already assigned, the thread does not terminate (as in WPP) but proceeds with processing the first row of the subsequent frame. This concept was further expanded by WHP [28] and 3D-WPP [29] algorithms that capitalize on dependencies existing in all the frames of the reference list. We view these approaches as complementary to ours since the parallelization degree (at the encoder side) obtained by capitalizing on inter-frame CTU prediction dependencies can also be exploited by WTP (in a much faster and more efficient way due to the presence of tiles).

As far as tile parallelism is concerned, in [30] the benefits of tile parallelism using uniform sizing were exploited. The motivation for the tile partitioning algorithm in [31] was to use more tiles compared to the available cores to facilitate load balancing. The method was based on deriving a static tile partition based (among others) on pixel variance and the required throughput. In [32] an adaptive content tile partitioning algorithm was proposed. The size of the tiles was decided to reduce the losses in coding efficiency incurred by tiles.

Many works on tile parallelism have focused on resizing tiles to reduce potential load imbalances and improve the speedup of TP. In [33] tile boundaries are adjusted on a per-frame basis. The scheme first defines vertical tile grid zones and then the horizontal boundaries by evenly balancing the expected workload at the vertical/horizontal zones induced. Workload estimates are based on average (from all previous frames) encoding time for each CTU. A similar approach but with static CTU weights was studied in [34], while [35] examines resizing heuristics based on the array partitioning problem with CTU load estimates tailored for the Low Delay (LD) Group of Pictures (GOP) structure. In [36] two CTU cost functions were proposed, one for the CTUs located in the right tile boundary and another one for the rest. The first one characterized coding losses while the second one was compression time. In [37] the authors defined in an offline manner optimal tile grids to be used by the encoder on a per-frame basis. Finally, in [38] the aim was to define tile resizing and tile-processor assignment in the case where the number of processors is smaller compared to the number of tiles. The algorithm iteratively explored different tile cuts each time performing Max Min scheduling. We view the aforementioned tile resizing approaches as complementary to ours, since they can be tentatively incorporated in the WTP scheme proposed in this chapter.

3.3. Speedup Performance

The performance of a parallelization scheme can be measured in terms of speedup, i.e., as the ratio between sequential and parallel processing time:

$$speedup = \frac{time_{seq}}{time_{par}} \quad (1)$$

Consider a frame with X CTU rows and Y CTU columns. Let t_{ij} denote the compression time of CTU_{ij}. The processing time for the frame using one thread (sequential) is dominated by the compression of each CTU (additional overheads are tiny) and can thus, be expressed as:

$$time_{seq} = \sum_{i=0}^{X-1} \sum_{j=0}^{Y-1} t_{ij} \quad (2)$$

The ideal speedup for WPP [12] comes when all CTUs incur the same processing time, i.e., $t_{ij}=c \ \forall \ i, j$, and the number of processors is not limited. Let EST_i denote the earliest start time for commencing the process of CTU row *i*. Due to precedence constraints in WPP, the following holds true:

$$EST_i = \begin{cases} 0, & i = 0 \\ EST_{i-1} + 2c, & i > 0 \end{cases} \quad (3)$$

By observing that the finish time of WPP equals the finish time of the last CTU row we get:

$$time_{WPP} = 2(X-1)c + Yc \quad (4)$$

which gives a speedup of:

$$speedup_{WPP} = \frac{XY}{2X + Y - 2} \quad (5)$$

Consider an *N×M* frame partitioning into uniformly sized tiles (*M* horizontal and *N* vertical zones). In TP the running time is dominated by the size of the largest tile. Depending on whether *X mod M = 0* or not, the maximum rows per tile are given by:

$$\max(TileRows) = \begin{cases} X/M, & XmodM = 0 \\ \lfloor X/M \rfloor + 1, & XmodM \neq 0 \end{cases} \quad (6)$$

Similarly, for columns:

$$\max(TileCols) = \begin{cases} Y/N, & YmodN = 0 \\ \lfloor Y/N \rfloor + 1, & YmodN \neq 0 \end{cases} \quad (7)$$

Thus, the speedup for TP can be given by:

$$speedup_{TP} = \frac{XY}{\max(TileRows)\max(TileCols)} \quad (8)$$

The running time of WTP is dominated by the largest tile for which (6) and (7) hold true. By treating this tile as a frame in which wavefront parallel processing is performed, (5) can give the speedup of WTP as follows:

$$speedup_{WTP} = \frac{XY}{2\max(TileRows) + \max(TileCols) - 2} \quad (9)$$

Comparing WTP against WPP it is straightforward that

$$speedup_{WPP} < speedup_{WTP} \ \ iff \ \ M > 1 \vee N > 1 \quad (10)$$

As far as the comparison of WTP and TP is concerned, we can get:

$$speedup_{TP} < speedup_{WTP} \ \ iff \ \ M < X \wedge N < Y/2 \qquad (11)$$

Notice, that the constraint posed by HEVC that a tile should consist of at least 4 CTU columns implies that for all eligible tile partitions, (11) holds true if there exists a tile with at least 2 rows. As a numeric example, consider a 1080p sequence (X=17, Y=30) and a 2×2 tile partition in WTP. Assuming that all CTUs incur the same computational cost then the maximum tile has size 9×15. By substituting in (5), (8) and (9) we get that: $speedup_{WPP}$ =8.2, $speedup_{TP}$ =3.92, and $speedup_{WTP}$ =16.45, almost double the one of WPP.

The above analysis demonstrates that the parallelization potential of WTP is higher than both WPP and TP when processors are unlimited and WTP uses the same number of tiles as TP. For the practical case where the number of processors is limited and WTP uses fewer tiles than TP (to obtain a better compression rate), TP is expected to exhibit higher speedup due to the latency experienced by WTP in fulfilling wavefront precedence constraints. Against WPP however, the speedup is expected to be higher in WTP since threads (equaling the number of tiles) can commence at once.

3.4. Experiments

Setup

We implemented WPP, TP, and WTP in HM 16.17 reference software using *p threads* for multi-threaded parallelization. Tile level parallelism was implemented by selecting the uniform tile option in the encoder (deactivating wavefront). Local, thread-safe copies were created of the main coding tools, i.e., search, transform and quantization, entropy coder, RD, etc. All algorithms were implemented using a thread pool with a configurable number of threads. In TP threads equaled the number of tiles and they were assigned on a one-on-one basis. WPP and WTP were implemented using row assignment whereby each thread was assigned a complete CTU row.

Experiments were conducted on a Linux Server with two 12-core Intel Xeon E5-2650 running at 2.20GHz. To make experimentation time manageable we used the first 150 frames of Class A and B test sequences for our evaluation. Results were obtained from a Low Delay (LD) setup with an initial I frame followed by B frames and a GOP size of 4. Bit depth was 8, CTU size was 64×64, max partitioning depth was 4, and search mode was TZ. One slice was used. Unless otherwise stated QP was set to 32.

Experiments with WPP and TP involved 4, 8, 16, and 24 threads (the number of physical cores available in our server), with tile partitions for TP of 2×2,4×2,4×4 and 6×4 respectively.

Various configurations were used for the evaluation of WTP including 2×2,4×2 and 3×3 tile partitions and a number of threads (8, 16, or 24). Performance was measured for the cases where the number of threads exceeded the number of tiles since with equal numbers WTP would behave as TP.

Speedup Results

In the first experiment, we measured the speedup of WPP, TP, and WTP. Figure 35 and Figure 36 plot the average speedups respectively for Class A and B sequences. First, we can observe that WPP offers the worst scalability, reaching a performance plateau at 16 threads in Class A and at 8 threads in Class B sequences. WTP on the other hand overcomes this limitation, further increasing speedup. For Class B sequences it achieves more than double speedup the speedup of WPP with 24 threads. As expected, TP achieves the best speedup since it uses a number of tiles equaling the number of threads/CPU cores. Nevertheless, it is noticeable that the differences against WTP are rather small, particularly in the case of 24 threads and 3×3,4×2 tile partitioning.

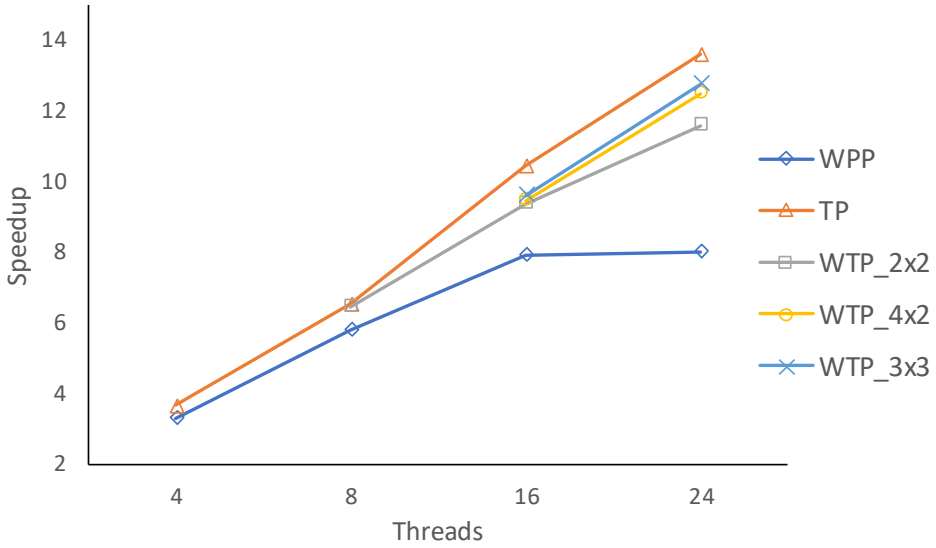


Figure 35. Average speedup in Class A sequences for QP=32 (Traffic, PeopleOnStreet).

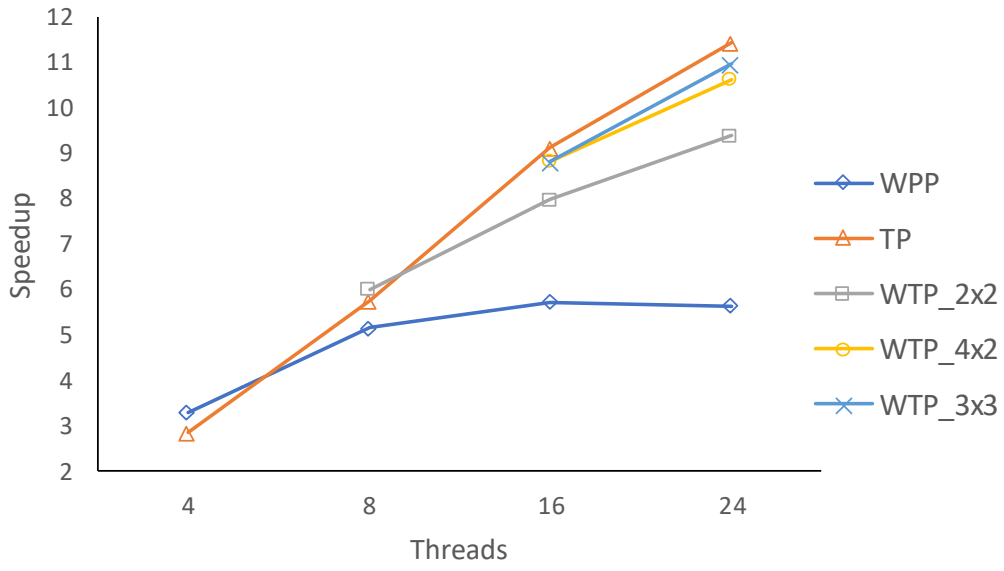


Figure 36. Average speedup in Class B sequences for QP=32 (BasketballDrive, BQTerrace, Cactus, Kimono, ParkScene).

Coding efficiency results

Next, we evaluate the compression efficiency and video quality of the 3 algorithms. We consider two tile partitions for WTP, namely 2×2,3×3 and a 6×4 partition in TP. All algorithms run with 24 threads for QP=22,27,32,37. Figure 37 shows the BD-rate [39] increase percentage and Figure 38 the BD-PSNR difference between WTP, TP, and WPP. For TP to capitalize on the available CPU cores, the resulting 24-tile partition leads to a high bitrate increase (in Kimono it reaches 6%) and a non-negligible decrease in PSNR (more than 0.2 in Kimono). On the other hand, WTP reduces negative effects with the smallest decrease being achievable for the smaller tile partitioning case of 2×2. For comparison, reasons Table 1 summarizes the achievable speedups in all QPs for 24 threads. It can be noticed that the improved coding efficiency of WTP comes at rather affordable drop-in speedup terms.

Cumulatively, the experimental results confirm the validity of our motivation, namely that we can overcome the speedup limitations of WPP and efficiency limitations of TP by using a combination of the two.

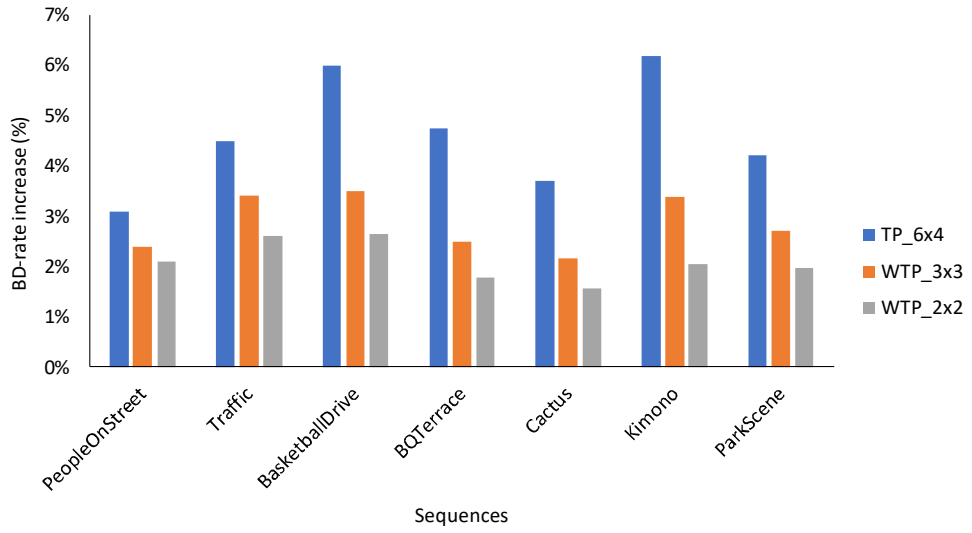


Figure 37. BD-rate percentage of increase versus WPP (24 threads, QP=22, 27, 32, 37).

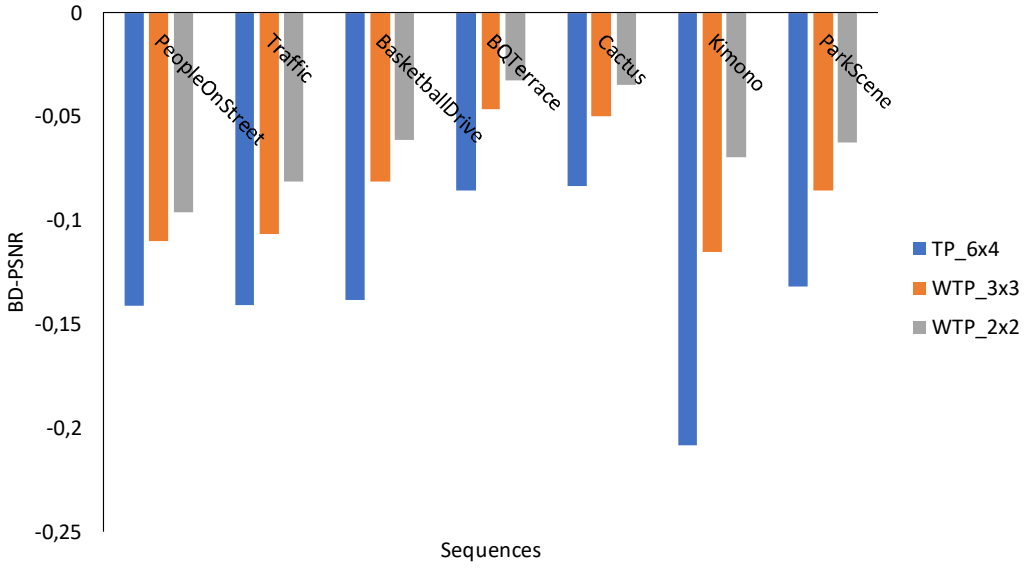


Figure 38. BD-PSNR difference versus WPP (24 threads, QP=22, 27, 32, 37, negative values show a quality drop).

	TP				WTP_2×2			
Sequence	**22**	**27**	**32**	**37**	**22**	**27**	**32**	**37**
PeopleOnStreet	15.0	14.6	14.4	11.8	13.1	11.8	12.7	11.7
Traffic	14.2	13.4	12.8	10.1	11.9	12.2	10.5	10.8

	TP				WTP_2×2			
Sequence	**22**	**27**	**32**	**37**	**22**	**27**	**32**	**37**
BasketballDrive	13.0	12.5	11.5	10.7	10.3	10.0	9.4	9.8
BQTerrace	14.3	12.3	11.3	10.1	9.8	9.5	9.4	8.4
Cactus	12.9	10.8	11.0	10.7	10.4	9.3	9.2	9.0
Kimono	15.9	14.3	14.0	11.6	11.5	11.2	10.7	9.7
ParkScene	14.4	13.0	12.3	12.4	10.9	9.8	9.8	9.7

Table 1. Speedup in all sequences (TP, WTP, 24 THREADS)

3.5. Summary and Discussion

In this chapter, we presented a novel parallelization scheme Wave front per Tile Parallelism (WTP) that aims at overcoming standalone deficiencies of tile and wavefront parallelization. Results demonstrated that WTP can achieve valid trade-offs between speedup and coding efficiency while overcoming the natural time performance limitations of the wavefront.

Chapter 4

On Combining Wavefront and Tile Parallelism with a Novel GPU-friendly Fast Search

4.1. Overview

Video encoding is a computationally demanding process that requires a well-designed and developed algorithm. An optimal solution requires a trade-off between quality and processing time. Today, most video standards focus on two goals: to improve compression performance and implement parallel architectures to minimize encoding time and at the same time, achieve better quality for higher video resolutions. Two of the most popular video standards, HEVC and the most recent VVC share three parallelization approaches. The first one included is Wavefront Parallel Processing (WPP). WPP allows for the creation of picture partitions that can be processed in parallel without incurring high coding losses. Rows of coding tree units (CTUs) are processed in parallel while preserving all coding dependencies. The processing pattern begins from the top left corner and progressively goes to the right bottom. In any case, the current CTU requires that top-left, left, top, and top-right CTUs are available to continue their encoding. For that reason, a shift of at least two CTUs is enforced between consecutive rows processed in parallel Figure 39.

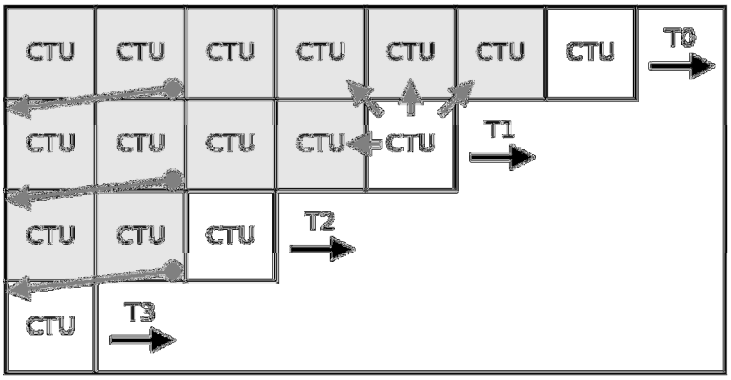


Figure 39. Principle of WPP and CTU dependencies.

Tiles on the other side divide a picture into independent, rectangular regions, enabling improved coding efficiency for parallel architectures. Tiles can be thought of as rectangular regions formed by the intersection of rows and columns (Figure 40). Tiles can have uniform spacing in a row and column boundary specification, or not. Moreover, Tiles share the same boundaries with CTUs and they yield a high pixel correlation. A similar partition scheme to parallelize the encoding or decoding process is Slices. Similar to Tiles, they are also independent pixel blocks so they can be coded separately (Figure 41).

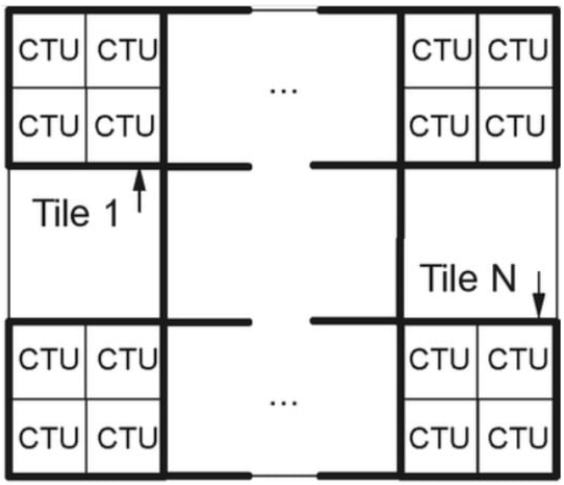


Figure 40. Principle of Tiles parallelism.

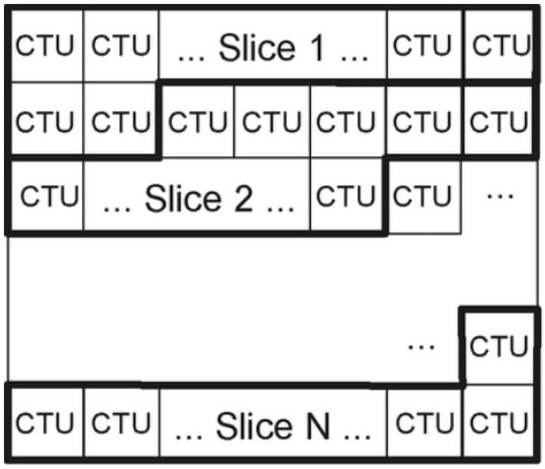


Figure 41. Principle of Slices parallelism.

Generally, tile partitioning is considered to be more flexible than slicing because Slices include headers that produce extra overhead that harms the coding efficiency. Therefore, in the case of multiple CPU threads that serve multiple slices (one thread corresponds to one slice), it may lead to non-negligible coding losses. On the other side, if we decrease the number of slices, we also decrease the performance. Furthermore, both video standards do not allow the co-existence of Tiles and WPP parallelization scheme at the same time, by design [20]. We achieved a bypass of this restriction with this work [40] proposing a new multi-threaded encoding scheme that is based on Tile partitioning and per-tile wavefront parallelism (WTP) (Figure 42).

Figure 42. Example of wavefront per Tile parallelism (WTP), 2 × 2 Tiles, 16 Threads.

In this case, the primary encoding scheme used is Tiles, but internally, our modified encoder enables a custom multi-threading wavefront encoding each tile separately. To comply with the original output-encoded streaming, we had to propagate the original HEVC structures in a thread-safe environment. The frame is divided into independent Tiles that have the same size. Each tile has its own thread-safe WPP data structures (entropy coders, search & prediction, quantization & transform) (Figure 43), and they are treated as separate frames but with lower resolution in CTU metrics. Tiles essentially allow for a typical multithreaded WPP-encoding scheme to be used. For each one of those Tiles, an independent WPP is used for the encoding procedure (Figure 44) consisting of a new encoding parallelism scheme (WTP).

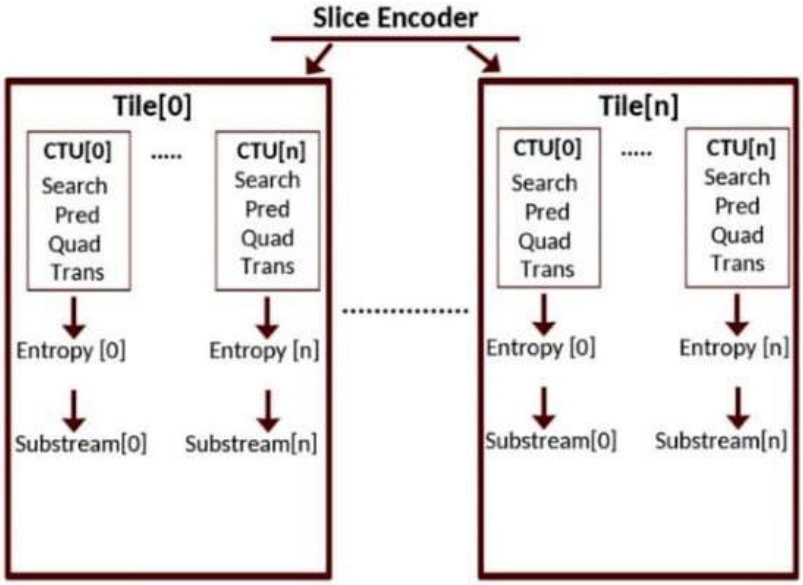


Figure 43. Wavefront thread-safe structures for each tile.

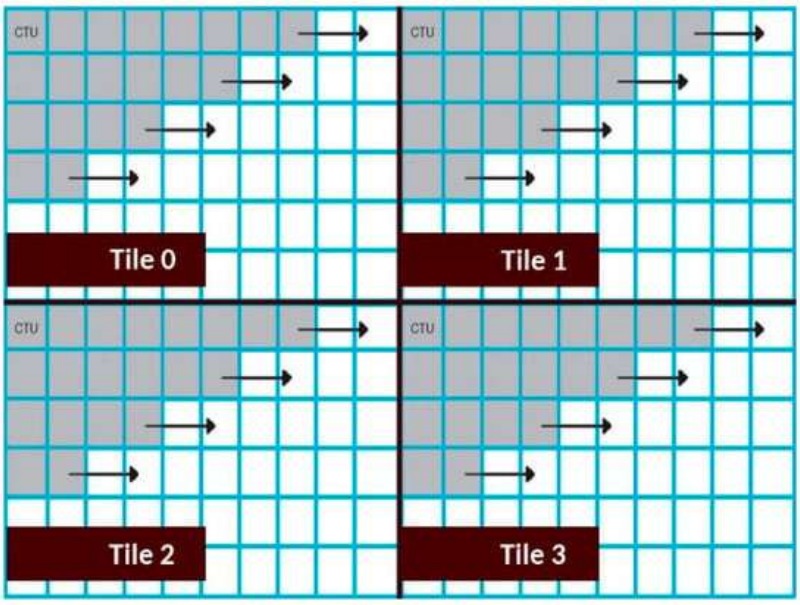


Figure 44. Wavefront encoding inside Tiles.

Furthermore, besides the multi-threading role contribution, in any modern video standard, coding efficiency has a huge impact that relies on the most critical part of the encoding process, which is the search for the best motion vectors and matching blocks. In this chapter, we propose a novel parametrized GPU-friendly Fast Search algorithm that can further improve the performance in such an environment. We evaluated this algorithm with our proven WTP multi-threaded encoding scheme that is based on the HEVC video standard. It seems that its contribution to the encoding processing time is considerable and it can be maximized when the system has weaker CPU process power than the GPU itself. In addition, this fast search algorithm is parameterized for its leaping search steps, which means that we can feed more steps when more powerful GPUs are installed in the system, to achieve even more accurate results.

4.2. Related Work

Motion vector calculations are the most intensive part of any modern video encoder. Therefore, several works are published to minimize that cost, including software and hardware implementations. Software implementations include CPU-only powered processors. Small diamond pattern search, large diamond pattern search, and horizontal diamond search [41, 42] are some of them that are proposed to replace the default TZ search algorithm. Experiments show that 49%-time saving could be achieved with a minor decrease in the PSNR ratio. Another implementation is the 6-point hexagon that reduces the motion estimation encoding time

compared with the TZ search, by about 50% [43, 44]. Xiong et al. [45] proposed a fast-inter-CU decision to reduce the encoding time, and Khemiri et al. [46] proposed a fast ME algorithm that is based on a diamond search pattern but introduced three mode decision levels. The time saving was about 57% with a minor PSNR decrease.

Hardware implementations include non-CPU integrated circuits, such as Field Programmable Gate Arrays (FPGAs), Application Specific Integrated Circuits (ASICs), and video cards. The most popular non-CPU hardware is video cards, most of them nowadays include graphics processor units (GPUs) that are programmable for general-purpose calculations. Parallel SAD calculations using an FPGA Xilinx [47] are used to achieve ×3.9 speedup in comparison to the baseline (single core or non-parallel) HEVC algorithm. Moreover, F.H. Shajin et al. [48] proposed a fast search quadrant-based algorithm that obtains better block matching, using an FPGA hardware platform. According to their experiments, they had 8.3562% better PSNR compared with the HEVC diamond search and 0.131% better performance for single-core executions. In addition, hybrid or heterogeneous algorithms were introduced that use both CPUs and the extra hardware (FPGAs, ASICs, or GPUs) [[49], [50], [51], [52], [53]], i.e., CPUs for coarse searching and GPUs for fine searching with analogous results. An average time saving for the latest is about 19% for single cores and ×10.75 when multi-cores are used.

On the other hand, video cards are far more common hardware among users. A high parallel 5-step GPU ME algorithm presented by Chen and Hang [54] reduced the encoding time for the motion estimation by about ×12 compared with the baseline alternative, but it concerns the previous standard H.264/AVC, as well as the work by Rodriguez and Martinez [55], where they achieved a speedup of ×2 compared with the baseline without any significant PSNR loss. An interesting GPU-based fast motion implementation is the adaptive search range (ASR), proposed by Kim et al. [56] for the HEVC. The idea was to adaptively change the search area every time, to reduce the number of parallel calculations inside the GPU. The results showed a reduction in the encoding time of about 40.3% and an RD loss of 1.2% compared with the single core competitive. Jiang et al. [57] implemented the integer ME and fractional ME on the GPU using a full search algorithm and obtained a time reduction of up to 53% compared with non-parallel baseline competition. The new NVIDIA's Fermi GPU architecture was used by Klemiri et al. [58] to improve the SAD calculations, reducing the encoding time by about 56.17% without any significant loss. Gogoi et al. [59] applied a low-complexity IME algorithm using two different pattern structures (PS) with 38 search points, resulting in 8.725%- and 9.072 %-time savings, respectively, in HM 16.8. A novel GPU full search multilevel resolution sample area (MLRME) was proposed by Xue et al. [60] that creates several down-sampling copies of the searching area to reduce the number of calculations. Experiments showed that the encoding quality declined by less than 1.5%, and a 30–60× speedup was achieved compared with on single-core CPU. A parallel motion estimation implementation was proposed by Zhang et al. [61], which includes pre-motion estimation, integer motion estimation, and fractional motion estimation.

A rapid mapping table algorithm was introduced to improve the efficiency of data access, in addition to a quasi-integral-graph algorithm, which is designed to calculate SAD or SATD efficiently for blocks of different sizes through GPU. A maximum of 12.7% was the encoding gain for this scheme compared with the single-core sequential code. Finally, a GPU-based low delay ME parallelization scheme was proposed by Luo et al. [62] that hierarchically uses a GPU parallelization by optimizing the ME process in a coding tree unit (CTU), prediction unit (PU), and motion vector (MV) layers. Experimental results demonstrate that the proposed scheme can achieve 41% encoding time savings with ME acceleration up to 12.7 times, and the incurred BD-BR loss is only 0.52% on average. This work is different from the latter approach in many ways. (a) Best MVs have zero assumptions, so there are no dependencies, and are all calculated at the runtime, using a z-order scheme using the full search range at the beginning (1st phase) and progressively reducing the search area, but leaving constant the searching points for all phases. (b) Our parallelization scheme optimizes only the IME part as a solid GPU block, giving better absolute speedup times and leaving space for further optimizations for the FME part. (c) Our core searching algorithm is versatile and easily parametrized to use any number of phases, search area boundaries, and searching points for each phase, according to the power of the GPU and the accuracy we prefer. These parameters are passed from the Host to Kernel at the runtime.

The main contribution of this chapter is to engage the existing hardware without any extra cost and to maximize the overall performance, especially when the system has weaker CPU process power than the GPU itself on specific topics. This is accomplished using an ultra-fast parametrized search algorithm in a parallel pattern, inside the video card hardware. Furthermore, by combining the GPU contribution to this work, with our previous parallelization scheme (WTP) running exclusively on CPUs, we propose an all-in-one encoding scheme with optimistic experiment results. Taking into account that exceptional video cards are nowadays very cost-effective and the fact that our fast search algorithm can be parametrized to delegate with more fine-grained computations in ultra-high definition resolutions, it is imperative to exploit the existing hardware to improve the computational complexity of the encoding part.

4.3. Implementation

For our WTP [40] encoding scheme, a custom thread pool is used to manage an arbitrary number of threads. Threads are typically set according to the number of available cores, but any other number can be used. The thread pool examines, in a loop, a reference table map of the available jobs for each virtual tile created, either sequentially or randomly according to an inline parameter. The first free thread is assigned to start the encoding process for the next available CTU for the tile selected. The way each thread is administrated from the application is also an important factor, and two main options are available to be selected: (a) to tie every single thread

to each CTU row, even keeping them in a wait state cycle when they have to wait for dependency CTUs and release them back to the thread pool only when the row is fully encoded, or (b) to release them immediately when the current job is completed (for example it cannot encode any more CTUs for the current row because neighbor dependencies are not ready) and re-schedule them for the next available job. While in [40] the first option was selected, in this work, threads are released as soon as the current job is completed and re-scheduled to keep GPU cores busy as much as possible [63].

In the proposed algorithm, threads are selected to comply with physical CPU cores to achieve maximum performance. If more CPUs are available, the speedup will automatically scale [64] concerning the available video resolution and the GPU capabilities [65]. Extending the WTP encoding scheme, we added the power of the GPUs in the encoding process [66] [67], exclusively for the motion estimation part, which is the most time-consuming task in a video encoder. In the HEVC case, motion estimation requires about 70% of the total computation time [20]. For this extension, we used the OpenCL framework [68], [69] to set concurrent GPU kernels to compute the calculations required for the integer motion estimation part (IME). OpenCL was selected because it is a universal framework for GPGPU programming that supports most of the major GPU manufacturers and because it achieves better performance compared with other frameworks [70]. Our novel GPU fast search algorithm, named “4PhaseFS”, is implemented as a high-parallel alternative to other non-parallel search methods. Despite the heavy parallelization, only one CPU thread can be used at a time. Mixing our high parallel WTP encoding scheme, which is CPU-based, with GPU for the IME part [71], we tried to balance the encoding load between CPUs and GPU. The GPU is charged to calculate the distortion ratio using a bottom-up accumulative scheme starting with 4 × 4 blocks (256 in total) as a base for all successive SAD (Sum of Absolute Difference) calculations that progressively reach the final 64 × 64 blocks. A parallel reduction method [72] [73], as shown in Figure 45, is used to speed up this process using 16 × 16 work groups, which means 256 threads in parallel are executing the GPU program (device kernel). Based on the previous step, we calculate the SADs of all enumerated CU and PU possible modes (i.e., 4 × 8, 8 × 4, 16 × 4, 4 × 16, etc.) inside a CTU, as shown in Figure 46, building, this way, an index array of 593 SAD elements in total, as shown in Table 2. The minimum distortion ratios from the previous processing step, which concerns the whole search area, are the motion vectors selected for the next step in the Host program (CPU). Memory optimization strategies are used to confine the data transfers between the host and the kernel [74].

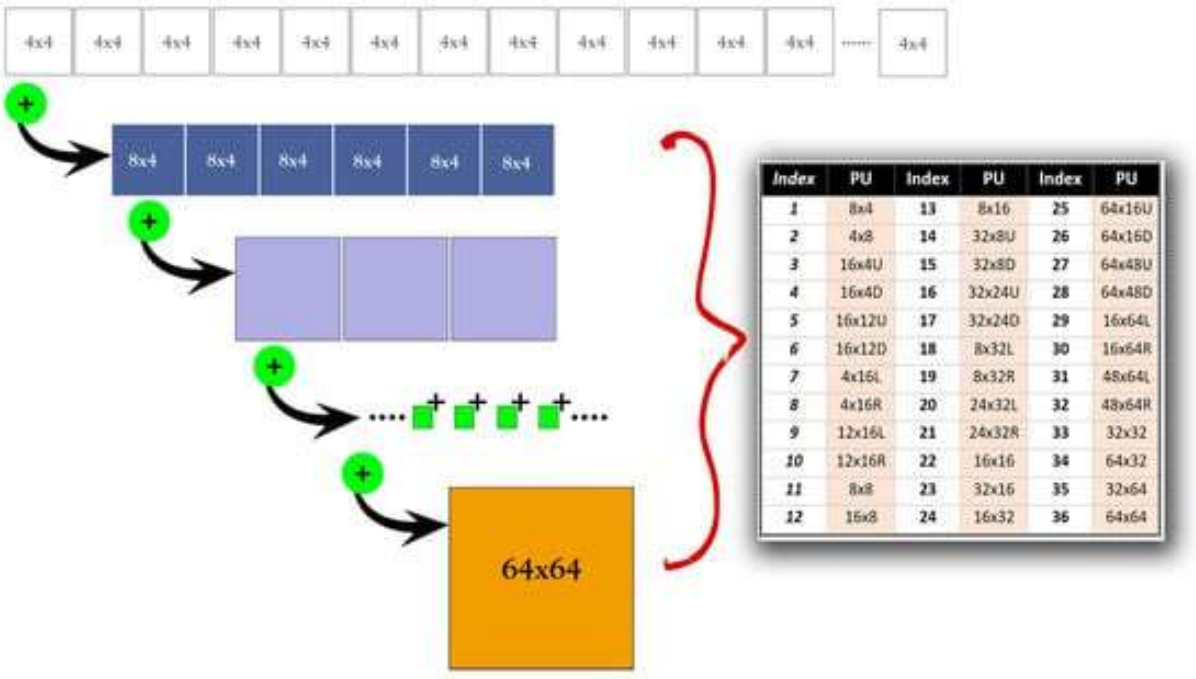


Figure 45. A parallel reduction scheme is used to calculate all blocks from the bottom up. Thirty-six PU block reduction calculations are executed in total, excluding the initial 4 × 4.

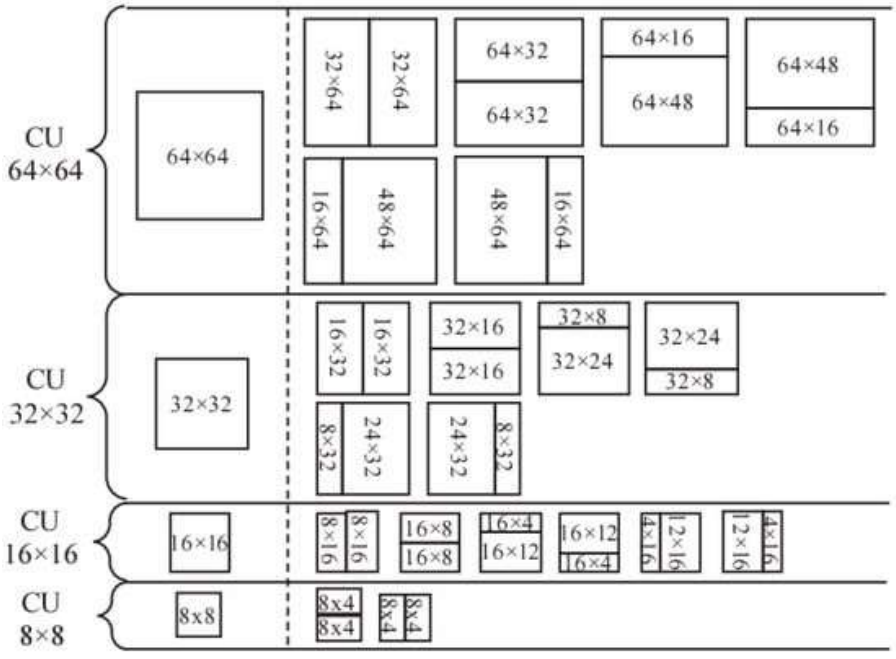


Figure 46. Enumeration of all CU and PU modes in a CTU.

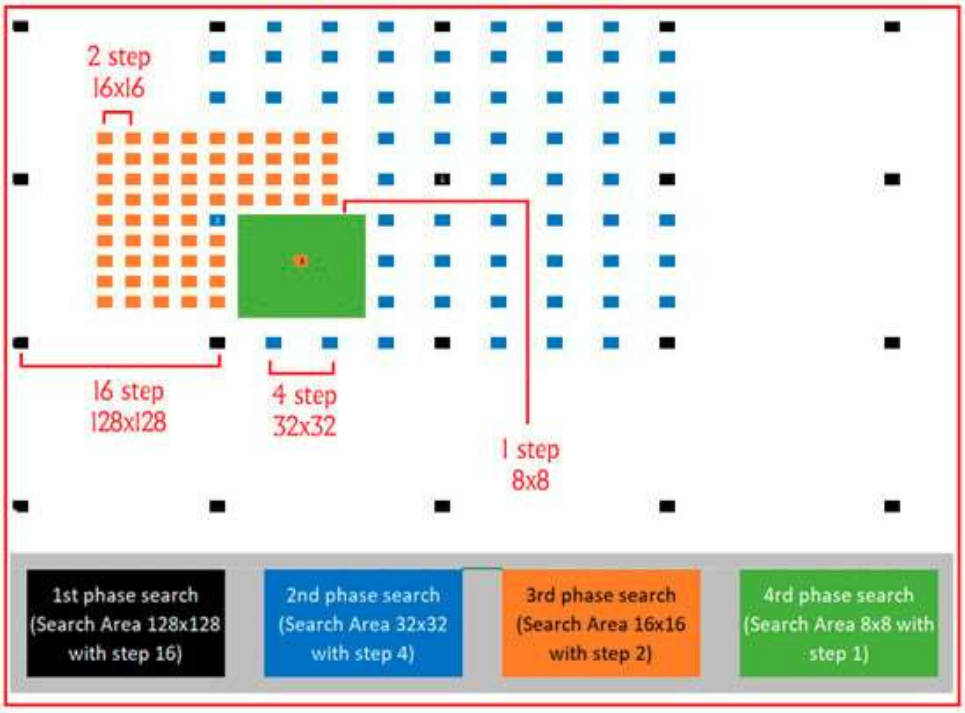


Figure 47. Visualization of the four phases. Steps leaps are expressed to video pixels either horizontally or vertically inside the searching area.

The minimum estimated RD-ratio is defined from Equation (1), which is also computed inside the kernel (GPU program created with a modified C programming language).

$$J_{MV} = SAD_{MV} + \lambda R_{MV} \qquad (1)$$

Where J_{MV} is the total cost, SAD_{MV} is the SAD of the specific block inside the index array as described in Figure 8, λ is the *Lagrange* multiplier, and R_{MV} is the total number of bits required to encode the specific motion vector. "4Phase" fast search is executed completely inside GPU cores, calculating the neighboring area of the current encoding CTU with a reduced step in every cycle (phases). That is true for both the searching area and the number of selected CTUs (leap step) for the calculations described above.

The 1st phase is the initial coarse search, starting from the top left side of the initial 128 × 128 searching area, and uses 16-leap steps (meaning how many CTUs to skip before the next CTU can proceed) in a top-down z-order pattern. The CTU block that has the minimum SAD (sum of absolute differences) from each phase becomes the center for the next phase. From the 3rd phase and after, every new search area, as well as every leaping step, becomes half of the previous phase, meaning that a progressively fine-grained search is made. Following this pattern, starting

the fine-grained search from the 2nd phase, it covers an area of 32 × 32 CTUs around the center found at phase 1, with 4 leap steps, 16 × 16 for the 3rd phase with 2 leap steps, and finally, an 8 × 8 area for the 4th phase with 1-leap-step (tests every CTU around the center), as indicated in Figure 47. The CTU was chosen from phase 4 of the GPU Kernel as the best candidate for the Host to get back to continue its process. The comparisons that are needed for the 4Phase fast search algorithm are always 256 in total (64 comparisons for each phase). Even though the 4Phase fast search always performs 256 comparisons (with the appropriate calculations) to obtain the motion vectors for the minimum SAD, this job is done ultra-fast by the GPU and it takes about 40% of the total time that the algorithm needs. The remaining 60% of the time is the data bus transfers overhead from the CPU to the GPU and vice versa.

This means that an integrated GPU, or the next PCIe generations, can achieve better speedups, as well as better quality since the 4PhaseFS algorithm can be adapted to be more accurate in its computations since there will be more headroom time to increase the searching area with detailer leap steps for each phase.

To summarize, the GPU program, called Kernel, executes two main functions for each position in the search area, which normally is double the CTU size. Function A calculates the SAD for each possible CU/PU block in the CTU to construct a new SAD Array and Function B calculates the rate-distortion ratio (RD-ratio) for each block and selects every time, the minimum cost, according to the SAD calculation history of Function A inside 593 data array elements. For the minimum cost calculated before, we also calculate and store in this function, the counterpart motion vector coordinates, which are 593 integer data arrays for X and Y parts accordingly. The resulting arrays of a) *ruiCost* and b) motion vectors are sent back to the Host (kernel caller) for further processing. The simplified algorithm of the kernel program (Figure 48a) and the way the host gets back the calculation results are shown below (Figure 48b).

```
1: Initialize AreaStartX, AreaStartY with 0
2: Initialize AreaEndX, AreaEndY with 128
3: Initialize AreaStepX, AreaStepY with 16
4: for Phase → 1 to 4
5:  for y → AreaStartY to AreaEndY with step AreaStepY
6:    for x →  AreaStartX to AreaEndX with step AreaStepX
7:      Call function Calculate_IndexArray_SADs(GPU_Thread)
8:      Call function GPU_Threads_Sync()
9:      Call function Calculate_RD_Cost(GPU_Thread)
10:     Call function GPU_Threads_Sync()
11:     Call function Save_BestRD_Cost_MV(GPU_Thread)
12:     Call function GPU_Threads_Sync()
13:   end for
14:  end for
15:  RngStep  ← 32 ShiftRight phase
16:  AreaStartX ← BoundariesCheck (BestSAD_X – RngStep)
17:  AreaEndX  ← BoundariesCheck (BestSAD_X + RngStep)
18:  AreaStartY ← BoundariesCheck (BestSAD_Y – RngStep)
19:  AreaEndY  ← BoundariesCheck (BestSAD_Y + RgnStep)
20:  AreaStepX ← 8 ShiftRight phase
21:  AreaStepY ← 8 ShiftRight phase
22: end for
```

(a)

```
1: if pcCUPartitionSize(0) is equal with SIZE_2Nx2N
2:    Call function setGPUKernelArguments(initialize)
3:    Call function setGPUKernelArgument(kernel_C_Program, memory address of xMVBuffer[cpuThread])
4:    Call function setGPUKernelArgument(Kernel_C_Program, memory address of yMVBuffer[cpuThread])
5:    Call function setGPUKernelArgument(kernel_C_Program, memory address of ruiCostBuffer[cpuThread])
6:    Call function runGPUKernelAndWaitToFinish(cpuThread, Kernel_C_Program)
7:
8:    for PU_mode → 0 to 592
9:       xMVArray   ← getMVx(cpuThread ,xMVBuffer[PU_mode]);
10:      yMVArray   ← getMVy(cpuThread, yMVBuffer[PU_mode]);
11:      ruiCostArray ← getRuiCost(cpuThread, ruiCostBuffer[PU_mode]);
12:   end for
13: end if
14:
15: indexBlock ← getCUBlockIndex(PUIndex);
16: rcMV       ← getMVs(indexBlock,xMVArray,yMVArray, cpuThread)
17: ruiCost    ← getCost(indexBlock,ruiCostArray, cpuThread)
18: Call function FractionMVSearch(rcMV,ruiCost)
```

(b)

Figure 48. (a) The simplified GPU Kernel program with major steps executed. (b) The way the host calls the GPU kernel and gets back the MVs and ruiCost for further processing.

The first part of Figure 48, Lines 1–3, is the initialization of the variables that are used in the inner loops below. We need AreaStartX, AreaStartY, AreaEndX, and AreaEndY to avoid boundary errors (below zero or out of the 128 × 128 search area). Line 4 is the counter of the

phases and lines 5–14 calculate the SAD and the Rate-Distortion Cost from the current CTU pointed from the x and y loop variables. The Calculate_IndexArray_SADs function fills the index array described in Table 2. This is performed in parallel from the available GPU cores and independently from each other. To calculate the RD costs for all 593 SAD elements, we must synchronize all GPU cores, which means that the Kernel should wait until all of them complete their calculations. At line 11, we save the best SAD Motion Vector for each one of the indexes, as well as the Cost. Lines 15–21 set the new search area, as well as the searching leap steps. The second part of Figure 48, Lines 1–15, is the GPU Kernel calling process from the Host side. This part is called only once, i.e., every new 64 × 64 CTU block is met. When the Kernel returns, we copy all calculated MVs and the ruiCost from the GPU memory back to Host data structures. From now on, for every PU mode pattern we receive from the encoder, we only search inside the Host's data structures to obtain the pre-calculated MV and ruiCost for further processing (lines 17–20).

Index	PU	Index	PU	Index	PU
0–127	8 × 4	480–511	8 × 16	576	64 × 16U
128–255	4 × 8	512–515	32 × 8U	577	64 × 16D
256–271	16 × 4U	516–519	32 × 8D	578	64 × 48U
272–287	16 × 4D	520–523	32 × 24U	579	64 × 48D
288–303	16 × 12U	524–527	32 × 24D	580	16 × 64L
304–319	16 × 12D	528–531	8 × 32L	581	16 × 64R
320–335	4 × 16L	532–535	8 × 32R	582	48 × 64L
336–351	4 × 16R	536–539	24 × 32L	583	48 × 64R
352–367	12 × 16L	540–543	24 × 32R	584–587	32 × 32
368–383	12 × 16R	544–559	16 × 16	588–589	64 × 32
384–447	8 × 8	560–567	32 × 16	590–591	32 × 64
448–479	16 × 8	568–575	16 × 32	592	64 × 64

Table 2. Index array of the SAD.

The costliest part for the GPU calculations is the data transfer from the Device (Video Card) to the Host (CPU) and vice versa because of the PCIe bus, which is much slower than CPU registers and the Host main memory data bus. For that reason, to minimize the transfer data costs, the kernel (GPU program) is implemented as a single program that executes both the A and B functions described above, at once, without having to return partial results to the host. When the 4-phase search cycle is completed, then and only then, the host receives back the final minimum RD cost array with respective MVs [74]. From this point so far, the CPU thread continues with the next step.

Each CPU thread is called its Device kernel (GPU), which means that the Device executes multiple kernels concurrently, as shown in Figure 49. This is a new feature for the Video Cards

manufactured in 2013 and after i.e., Nvidia's Fermi architecture (CUDA API version 3.0) [58]. The workhorse of the GPU is the streaming multiprocessor, or simply SM in the Nvidia world. In the OpenCL jargon, they are called Compute Units. The number of Compute Units may range from two to several dozen, and each Compute Unit contains the following: (1) execution units to perform 32-bit integer and single or double-precision floating-point arithmetic; (2) special function units (SFUs) to compute single-precision approximations of log/exp, sin/cos, and rcp/rsqrt; (3) a work-group scheduler to coordinate instruction dispatch to the execution units; (4) a constant cache to broadcast data to the Compute Units; (5) shared memory for data interchange between threads; and (6) dedicated hardware for texture mapping.

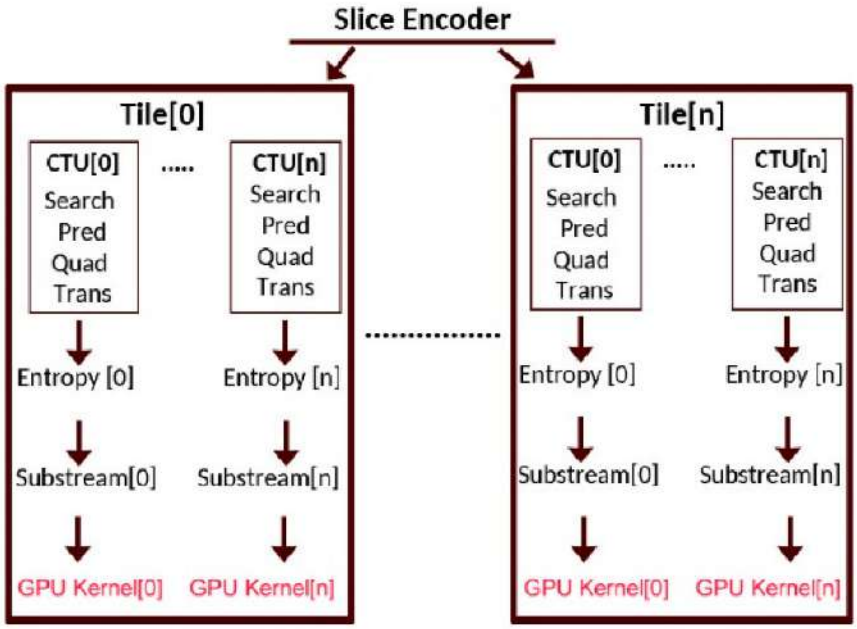


Figure 49. WTP encoding with GPU.

CUDA streams are used to manage concurrency between execution units with coarse granularity. This feature is implemented in the OpenCL framework with multiple queues. Therefore, each CPU thread uses a different queue to execute the same kernel but with different data. If the Compute Units are all busy, then the execution of the next kernel takes on a wait state until one becomes free. Thus, separate GPU streams are executed concurrently, and the operations requested in each queue are performed sequentially; in a sense, OpenCL queues are like CPU threads where operations are performed in order. Even when multiple queues are not used, keeping all operations asynchronous (i.e., out-of-order execution with only one queue) improves performance by enabling the CPU and GPU to run concurrently by letting the video card drivers automatically decide the best strategy for the load distribution. This method works asynchronously, which means that a later encoding CPU thread may be serviced sooner than a previous one, so a synchronization engine should be used to keep track of the encoding process

of the WTP. An arbitrary number of CPU threads request the same number of Video Card Compute Units to calculate the RD cost of the IME part. According to the capabilities of the Video Card being used, i.e., number of concurrent Compute Units, local memory size, L1 and L2 cache sizes, etc., the right balance must be chosen between the CPU's physical cores and the video card's Compute Units, to get the best results from each world combined. Increasing CPU threads to a number that exceeds the maximum Compute Units of the Video card does not offer any important gain to the encoding speedup since the CPU threads are obtaining a wait state. In addition, kernels with a very large number of threads or increased demands of local memory size (most GPUs today give about 64 KB of local memory for each Compute Unit) may need two or more Compute Units to execute the same code, which means that the concurrency level is dropping down. The Kernel for this extension is written carefully to avoid data waste and to be as compact as possible. Moreover, the code is optimized to avoid raw multiplications and division program statements and is replaced with left and right shifting statement commands, respectively, which are far faster.

4.4. Experiments

Implementation Details

We implemented GPU Fast Search in HM 16.17 reference software using a custom C++ code for multi-threaded parallelization and with AMP (Asymmetric Motion Partitioning) enabled, a feature that was not used in our previous benchmark publication [40]. All algorithms were implemented using a thread pool with a configurable number of threads. In addition, in this implementation, we have changed the thread assignment strategy and routing. Instead of preserving the CPU thread until it finishes the whole encoding CTU line, obtaining a wait state cycle in case the neighbor dependencies are not ready—this is a WPP restriction; in this implementation, we free the CPU thread immediately as soon as it meets the above condition paying, of course, the thread scheduling initialization cost. We selected this approach to keep the GPU's core load as high as possible.

Setup

We conducted experiments on a Linux Server with two 12-core Intel Xeon E5-2650s running at 2.20 GHz as the base frequency 2.90 GHz for turbo mode and 1.20 GHz for power save mode. The Graphics Card installed on our chain was Nvidia Quadro P4000 with 8 GB of GPU

Memory that combines 1792 CUDA Cores (driver version 470.141.03). The Pascal architecture of the Video Card was introduced in April 2016 and uses the PCI Express 3.0 × 16 system interface.

To make experimentation time manageable we used, for our evaluation, the first 150 frames of Class A and B common test sequences [37] listed in Table 3. Results were obtained from a Low Delay (LD) setup with an initial I frame followed by B frames and a GOP size of 4. Bit depth was 8, CTU size was 64 × 64, max partitioning depth was 4, and the search mode was TZ. As mentioned above, AMP (Asymmetric Motion Partitioning) was enabled. One slice was used. Experiments with WTP involved 24 threads (the number of physical cores available in our server), with tile partitions of 2 × 2, 3 × 3, and 6 × 4, respectively.

Class	Sequence Name	Resolution
B	Basketballdrive	1920 × 1080 50 fps
B	Bqterrace	1920 × 1080 60 fps
B	Cactus	1920 × 1080 50 fps
B	Kimono	1920 × 1080 24 fps
B	Parkscene	1920 × 1080 24 fps
A	Peopleonstreet	2560 × 1600 30 fps
A	Traffic	2560 × 1600 30 fps

Table 3. Test video sequences.

Coding Efficiency Results

Next, we evaluated the compression efficiency and video quality of the GPU-accelerated WTP with the original WTP and the default HEVC TZSearch. We considered three tile partitions for WTPs, namely 2 × 2, 3 × 3, 6 × 4, and a 1 × 1 partition for the default. All algorithms run with 24 threads for QP = 22, 27, 32, and 37, except for the original TZSearch which runs with 1 thread (sequential). Figure 50 and Figure 51 plot the average speedup for Class A and B sequences with constant QP = 32 versus the default sequential TZSearch (1 × 1 Tile partition and 1 thread). In every case, we can observe speedups with an average rate of ×8.8 for Class B videos and ×9.6 for Class A videos. Figure 52 plots the average speedups, respectively, for Class A and B sequences with constant QP = 32 versus the original WTP (the one without GPU acceleration). In this case, the average speedup is ×1.18 or 18% of time-saving. Figure 53 plots the gain difference between the same setup but with different CPU frequency clocks, i.e., turbo mode @2.9 Ghz versus low-power mode @1.2 Ghz. Using a constant QP = 37, we can observe that for the relative average speedup between the original WTP and the GPU accelerated one, the gain is a little bit higher for the weaker CPUs than the stronger ones. These are relative comparison benchmark pairs, between the original WTP and the GPU-accelerated fast search WTP version, for each CPU

frequency mode separately. Thus, while the absolute encoding time is significantly smaller for the high-frequency CPUs, the gain is smaller than the low-frequency competitive ones.

Finally, in Figure 54, the BD-Rate increase is proportional to the tiling scheme used, with 2 × 2 partitioning, obtaining the lower increase from all video sequences. To capitalize on the usage of the available CPU cores, a 6 × 4 tile pattern scheme results in a 24-tile partition that leads to a relatively high bitrate increase (in Cactus it reaches 13%). This is expected since Tiles do not use the Coding Units from neighboring ones for prediction, and this results in an increase of the BD-Rate and simultaneously a PSNR decrease since this affects the CABAC (Context-Based Adaptive Binary Arithmetic Coding) learning process, as the context variables are reset at the beginning of every Tile. The BD-PSNR [39] decrease in Figure 55 is also proportional to the tiling scheme. In the worst case, we can observe a 0.24 quality drop for the kimono video and a 6 × 4 tiling scheme. BD-PSNR values represented in the plot are the average absolute PSNR experiment results for QP = 22, 27, 32, and 37 with 24 threads

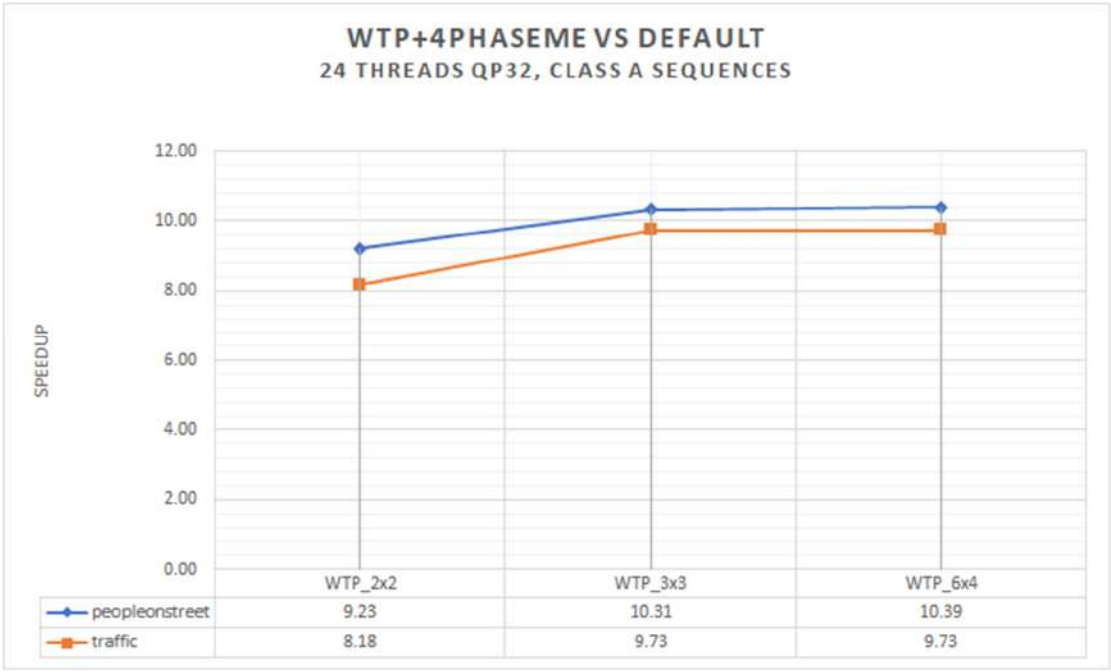
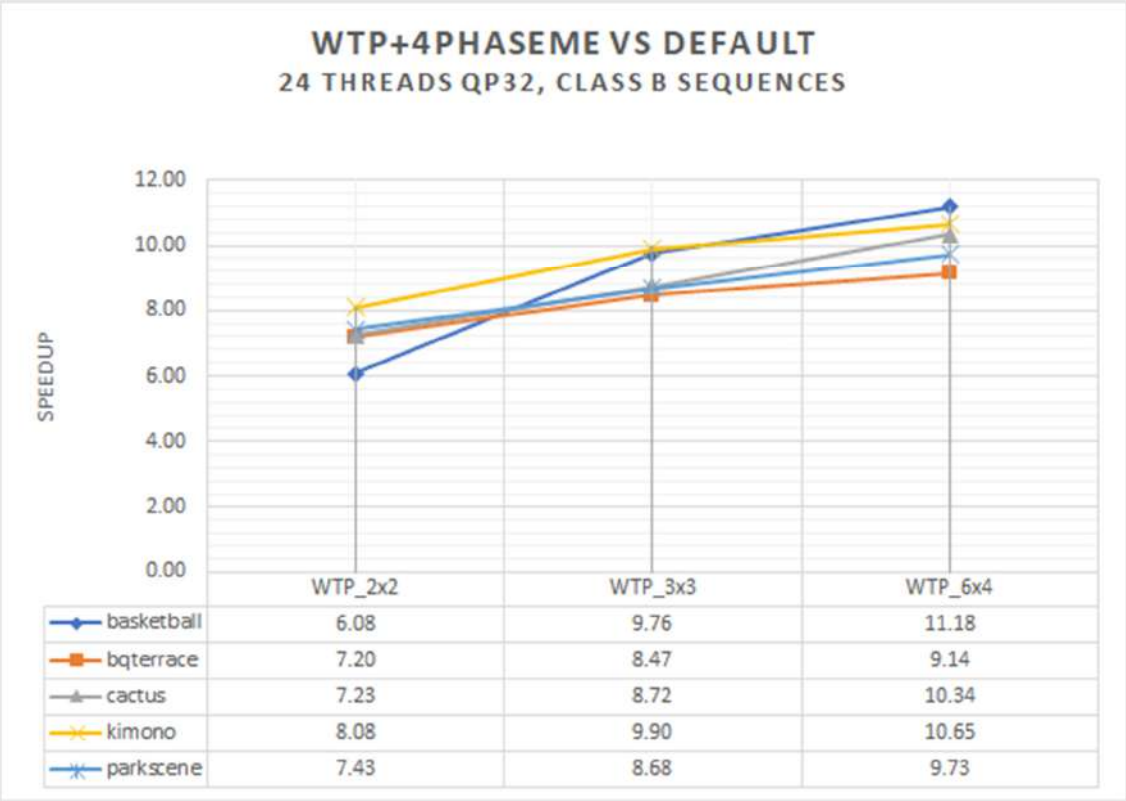
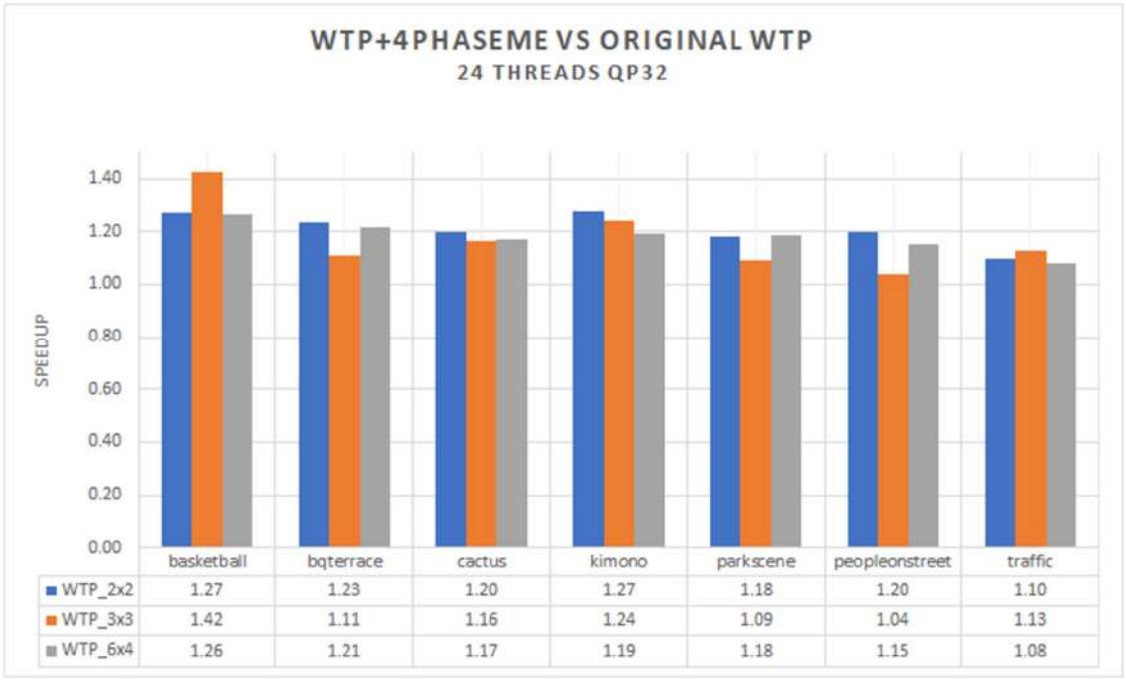


Figure 50. WTP + 4PhaseME average speed up in Class A sequences for QP = 32 (Traffic, PeopleOnStreet) versus default TZSearch.



	WTP_2x2	WTP_3x3	WTP_6x4
basketball	6.08	9.76	11.18
bqterrace	7.20	8.47	9.14
cactus	7.23	8.72	10.34
kimono	8.08	9.90	10.65
parkscene	7.43	8.68	9.73

Figure 51. WTP + 4PhaseME average speed up in Class B sequences for QP = 32 (BasketballDrive, BQTerrace, Cactus, Kimono, ParkScene) versus default TZSearch.



	basketball	bqterrace	cactus	kimono	parkscene	peopleonstreet	traffic
WTP_2x2	1.27	1.23	1.20	1.27	1.18	1.20	1.10
WTP_3x3	1.42	1.11	1.16	1.24	1.09	1.04	1.13
WTP_6x4	1.26	1.21	1.17	1.19	1.18	1.15	1.08

Figure 52. Average speedup increase using WTP + 4PhaseME versus original WTP (QP = 32, AMP = true, Modified CPU threads scheduling).

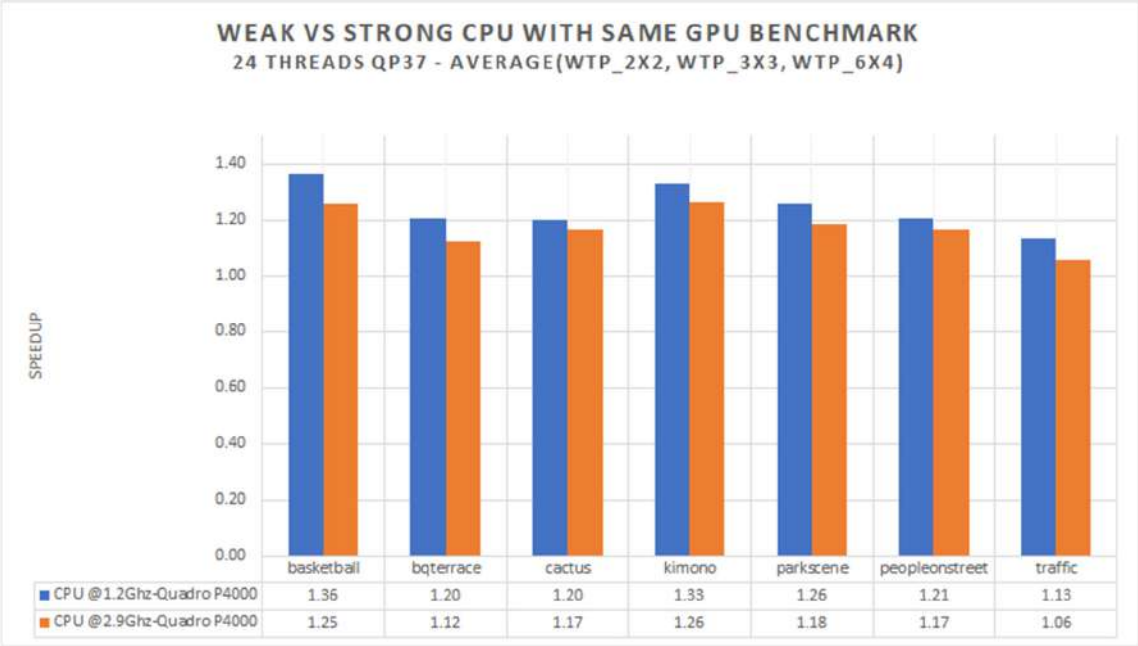


Figure 53. WTP + 4PhaseME average speedup versus original WTP, using the same GPU but with different CPU frequencies for QP = 37 (average of WTP_2 × 2, WTP_3 × 3, and WTP_6 × 4, CPU-A = 1.2 Ghz and CPU-B = @2.9 Ghz).

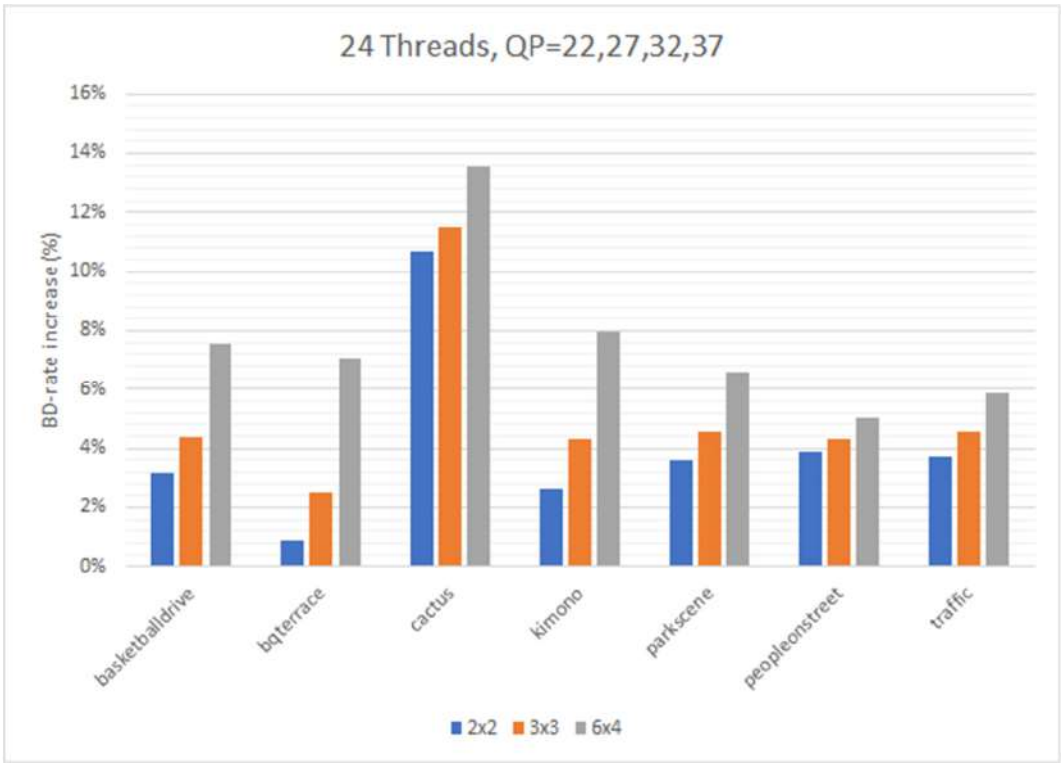


Figure 54. BD-rate percentage increase for 4PhaseMe (24 threads) versus default TZSearch (QP = 22, 27, 32, 37).

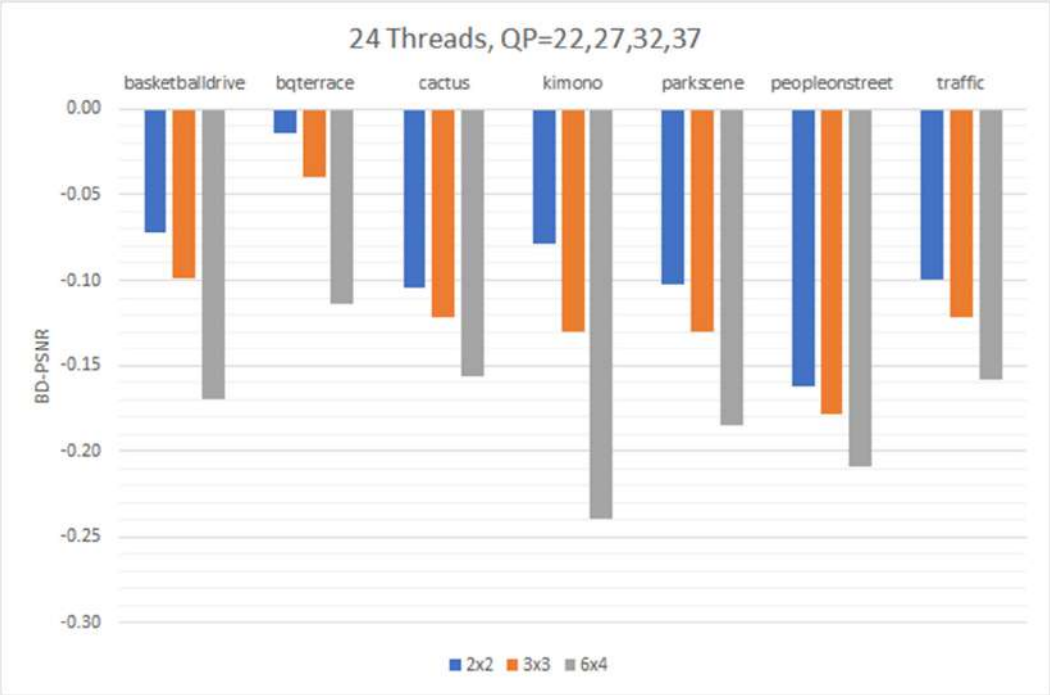


Figure 55. BD-PSNR difference for 4PhaseMe (24 threads) versus default TZSearch (QP = 22, 27, 32, 37, negative values show a quality drop).

4.5. Summary and Discussion

In this chapter, we present a novel GPU fast search motion estimation algorithm that works in parallel with a CPU-based multi-core encoding scheme WTP, which aims to minimize the encoding time. Results demonstrated that the 4Phase GPU fast search motion estimation algorithm can achieve an average speedup of ×8.84 for Class B videos and ×9.60 for Class A compared with the original sequential TZSearch. Moreover, the GPU contribution in the original WTP can increase the speed by an average of 18%. According to the results quoted above, undoubtedly, the GPU contribution is considered adequate to exploit any improvements. Using our already optimized and parallelized WTP encoding scheme, the GPU addition improved the encoding times by more than 25% in some cases, reaching 42% at its peak, when a WTP 3 × 3 pattern is used on a “basketball” video sequence. Class A video sequences seem to have a better average speedup than Class B sequences. Regarding the evolving GPU in the encoding process, many factors may have an impact on the final system performance, for example, PCI performance, video card resources, GPU performance when thermal throttles are met, operating system scheduling priorities, etc. It is worth mentioning from our experiments that different official video card driver versions can perform differently on identical setups.

According to the results demonstrated, it is worth highlighting that the average speedup gain is higher when CPU core frequencies are lower, enforcing the same setup but changing the scaling governor in our Linux server from performance to power saving. Scaling governors implement the algorithms to compute the desired CPU frequency, potentially based on the system's needs. Scaling drivers interact with the CPU directly, enacting the desired frequencies that the current governor is requesting. All experiments except the last mentioned in Figure 16 were executed with the default scaling governor (OnDemand), which scales the frequency dynamically according to the current load, i.e., jumps to the highest frequency and then possibly back off as the idle time increases. Despite the technical details and the way we implemented them in this work, the general idea of the searching algorithm can easily be applied to any platform, taking advantage of the newer video card innovations and add-ons, such as extensive and explicit memory (i.e., fewer data transfers from host to the kernel and vice versa) with higher bandwidth and speed clocks, more GPU cores and streaming processors (meaning more kernels are executed concurrently), and better support for concurrent kernel execution by the driver itself in newer architectures, etc. The negative effects of the BD-Rate increase and BD-PSNR decrease that are observed are more related to tiling since it affects the CABAC learning process. To reduce these effects, a balancing tile scheme is suggested, preferably a 2 × 2 pattern, to obtain the best of both worlds, combining speed and quality.

Cumulatively, the experimental results confirm the validity of our motivation, namely that we can benefit from the GPU computational power, especially when massive calculations take place in parallel for any video encoder that makes use of the motion estimation concept. As supreme video cards are becoming more and more cost-effective and our proposed fast search algorithm can easily be adapted and parametrized to delegate fine-grained computations in ultra-high definition resolutions, it is more than imperative to exploit the new possibilities and to improve the computational complexity. The gain is further increased when entry-level systems come into play operated by low-end CPUs. It is believed that the new-generation GPUs will boost the whole process and maximize the gain of the proposed GPU-friendly Fast Search without any extra costs.

Chapter 5

Fractional execution resolver using a hybrid multi-CPU/GPU encoding scheme

5.1. Overview

Video coding is a technique for compressing video data to minimize storage capacity and network bandwidth. As is known, the most intensive parts of any modern video encoder are the motion estimation (ME) calculations. Motion estimation includes integer-pixel ME and fraction-pixel ME. Typically, integer motion estimation (IME) is initially performed, and the best motion vectors from this stage are passed to the fraction motion estimation (FME) to achieve greater accuracy [20]. Fraction motion estimation is a two-step process. It first achieves half-pixel precision over integer pixel estimation by testing eight neighboring search points and then achieves quarter-pixel precision (Figure 56). A total of 16 search points are tested during the FME process, and the point with the minimum rate-distortion (RD) is selected as the best candidate. The fraction calculations (Equation (1)) are undeniably another CPU-intensive job for the motion estimation part, consuming more than 16% of the total time motion estimation (ME) needs according to our benchmark profiler.

$$\begin{cases} f\left(\overrightarrow{MV_{pre}}, \lambda_m\right) = SATD\left(\overrightarrow{MV_{cur}}, \quad \overrightarrow{MV_{pre}}\right) + \lambda_m \cdot R(\overrightarrow{MV_{pre}} - \overrightarrow{MV_{cur}}) \\ \overrightarrow{MV_{best}} = \arg\min_{\overrightarrow{MV_n}} f\left(\overrightarrow{MV_n}, \lambda_m\right), \quad \overrightarrow{MV_n} \in \delta \end{cases} \tag{1}$$

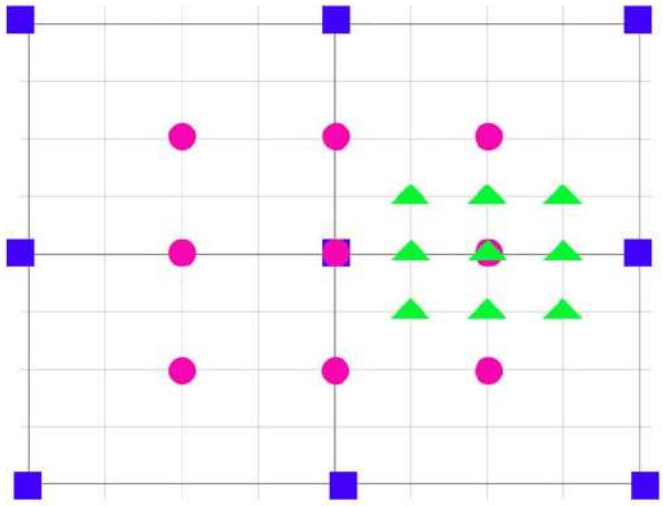


Figure 56. The HEVC FME process. Square symbols represent the integer pixel search points; the circles denote the interpolated half-pixel search points (1st step of the FME) and triangles the interpolated quarter-pixel search points (2nd step of the FME).

The SATD (Sum of the Transformed Differences) measures the distortion when encoding the current PU $(\overrightarrow{MV_{cur}})$ and with predictive motion vector c($\overrightarrow{MV_{pre}}$); λ_m is the Lagrange multiplier and $R(\overrightarrow{MV_{pre}} - \overrightarrow{MV_{cur}})$ is the code bits required for the encoding; δ is the motion vector candidates set for the FPME search points. FME is searching among 16 neighboring candidate points (δ) around the best IME motion vectors. The best candidate is the point with the minimum RD cost.

Fraction motion estimation typically follows integer motion estimation even though, in some cases, the extra computational efforts do not contribute extra accuracy to the already calculated motion vectors from the IME part. Hence, several software and hardware solutions have been published to minimize the FME computational load and thus minimize the total encoding times. Many of them include specialized hardware such as field-programmable gate arrays (FPGAs), while others are focused only on optimized algorithms at the software level. FPGAs are expensive and oriented toward companies and professionals that mostly need real-time video-encoding hardware [75], [76], [77], [78]. In contrast, optimized software algorithms have been introduced to reduce the computational load [79], [80] [81] in variants of existing software video encoders without any additional costs. The proposed work in this chapter is also a software algorithm that extends our proven hybrid encoder [40], [82] to increase the speedup times by minimizing the calculations of the fraction motion estimation (FME) part by skipping them when specific criteria are met inside a fraction execution resolver (FER) function. Our encoder already has a GPU-friendly fast integer motion estimation algorithm [82], so this extension focuses only on the fraction optimization part. As we target mostly the end users, we only exploit and capitalize common hardware setups such as video cards. Video cards with powerful graphics processor units (GPUs) embedded in them cannot be considered an extra addition to these common setups; so,

as nowadays GPUs are a standard hardware component for every video card unit, they cannot be considered as external hardware add-ons such as FPGAs, which have custom design requirements or costs for the end user, but rather as already built-in components. GPUs can be used for fast general-purpose calculations, so many researchers have exploited their capabilities. GPUs are bandwidth optimized but have low latency—unlike CPUs—so massive parallelism is used to hide latency. Although these programming considerations have been mentioned before, GPUs are continuously evolving to deliver new capabilities to all workloads even though they were initially intended for gaming.

One of the key technologies that have appeared recently in the latest GPU generations is tensor processing units (TPUs), which comprise an application-specific integrated circuit (ASIC) that charges for artificial intelligence (AI) calculation acceleration [83], [84]. AI workloads are specifically designed to handle computational demands using matrix processor technology. Using TPUs as matrix processors instead of general-purpose processors, we can overcome the memory access restrictions that slow down both GPUs and CPUs. So, the primary task for TPUs is matrix processing, which is a combination of multiply and accumulate operations. In our proposed work, neither a GPU nor TPUs can be applied for the FME part for many reasons. The most important reason for the GPU's inadequacy is the problematic parallelization of the searching point dependencies (Figure 56), while GPU parallelization is a constraint-based scenario for sequential vectorized data.

Alternatively, as a suggested solution to this problem, a new GPU-friendly FME algorithm could be introduced to overcome the dependencies problem; however, this suggestion extends our proposed FER rather than mutually excludes it. Furthermore, TPU utilization was also dropped as an alternative acceleration proposition because the default FME algorithm generally requires high-precision arithmetic calculations to which TPUs are not suited [85]. In addition, since TPUs are designed for AI applications, they do not fetch instructions to execute directly; instead, the host loads the matrix processor for specific operations only [86], [87]. So, currently, TPUs are inappropriate for general-purpose calculations, but, in the near future, TPUs can be developed in such a way as to outperform non-AI applications also. So, for the sake of (a) easy adaptation, i.e., no extra hardware is needed, and (b) easy implementation, i.e., intervention only at the software level with a well-known TSZ algorithm, we focused on skipping the encoder's default fraction ME when specific criteria are met.

We extended our previous hybrid encoding model [40], [82] to support the proposed fraction execution resolver (FER) algorithm, which has been proven to be fast, easy to implement, and cost-effective, minimizing in this way the computational load and thus the encoding times even further. This encoding model effectively utilizes both CPUs and GPUs in a flexible tiling selection scheme to balance speed and quality. Under specific circumstances where a balance between quality and speed is important for the end user, it provides a valid trade-off between speedup and video coding efficiency. We implemented a highly parallel environment using an innovative technique named Wavefront per Tile Parallelism (WTP). With this technique, we combined two

parallelization schemes into one, maximizing in this way the parallelization workload of the encoder by the co-existence of WPP and tiles simultaneously (Figure 57), which is forbidden by design in HEVC and VVC specification standards. The primary encoding scheme of WTP is tiles, which are fully compatible with the mentioned standards. Furthermore, by extending our modified encoder, we enabled internally the wavefront parallelization pattern for each tile separately, applying to the latter our multithreading environment setup. We propagated also the original structures to comply with the HEVC video standard, i.e., the encoder we selected to run our experiments [40]. Following the mentioned video standard, tiles divide each frame into independent rectangular regions. In our case, all tile columns and row boundaries are distributed uniformly, i.e., they all have the same size, and, essentially, they are treated as low-resolution separate frames—in CTU metrics—and they are encoded separately in a parallel pattern. So, each tile uses the standard WPP encoding pattern.

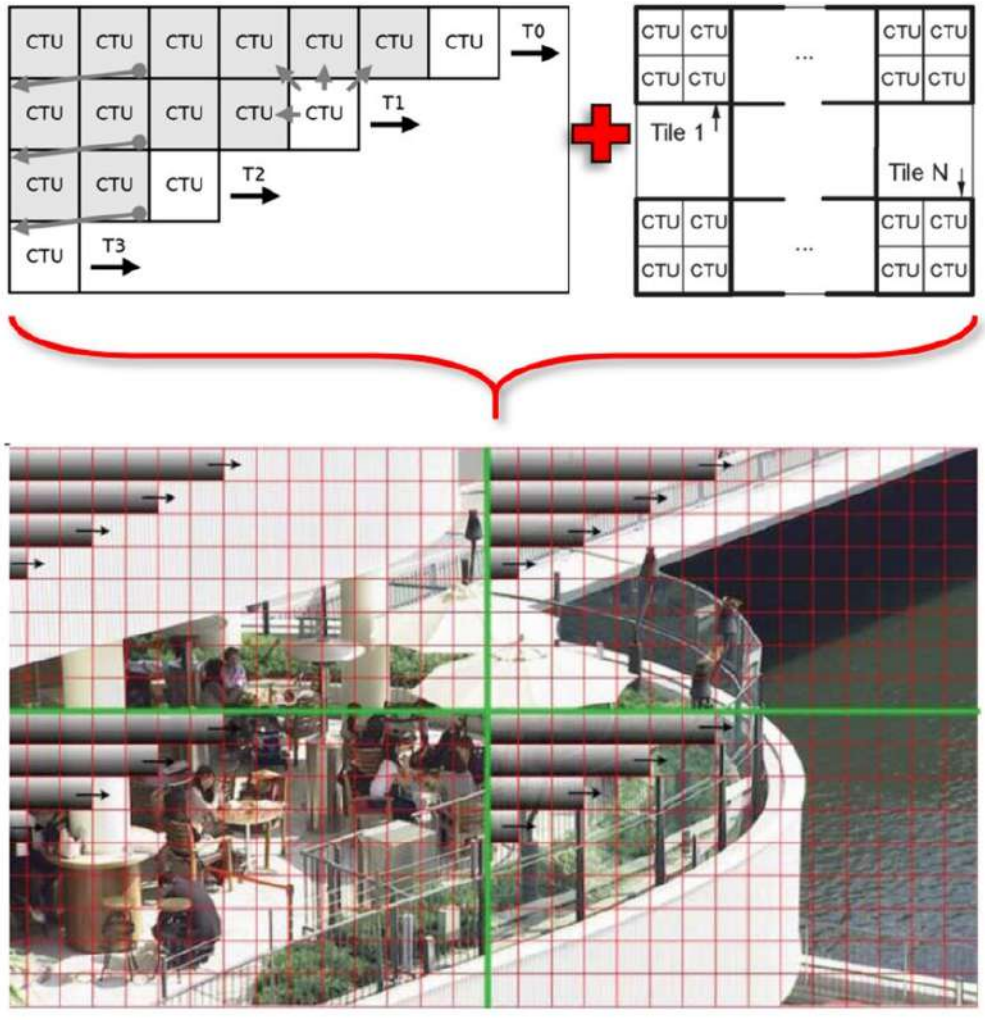


Figure 57. Combining WPP (upper left) and tiles (upper right) to create a new parallel scheme (WTP) (bottom).

GPU power is engaged [82] to improve the coding efficiency for each tile separately only for the integer motion estimation (IME) part and falls back to the CPU for the fraction refinement as a subsequent and mandatory job. The fraction part always follows the IME part, but it is very difficult to parallelize the default algorithm because of the dependencies it has. Vectorization is the particular way parallelism is achieved in this case. In particular, vectorization mostly uses dedicated SIMD execution hardware units in processors using specialized instructions such as ×86 SSE/AVX, ARM Neon, and GPU compute units. So, since we cannot achieve vectorized data in the FME case and the data size is not big enough for bulk calculations (non-bandwidth optimized), dedicated hardware may be the solution for this particular problem, which is costly, or a GPU-friendly FME algorithm that directly follows the IME calculations in the same kernel. Although there are parallelization implementation difficulties in the FME part, the fractional refinement is very important because it can increase the accuracy of the IME step using an interpolation pattern for the half and quarter samples of the testing area. While our high parallel CPU/GPU encoding scheme, algorithm reduces significantly the encoding time, the fraction part (FME) that is always executed as a subsequent job does not contribute to the final encoding time since it is not optimized in any way.

Common video sequences are known that contain many static blocks or non-moving parts in each frame. In some special cases, there are even a number of whole consecutive frames that are static. Based on this idea, we optimized our encoder to invoke the fraction part if, and only if, there are non-static blocks inside the current frame and skip it completely when this is not the case, which means that the fraction part is ignored if static blocks are met. We implemented an algorithm that applies or skips the FME if specific criteria are met. The fraction execution resolver (FER) is a software solution algorithm that minimizes the encoding time by avoiding unnecessary calculations. In the same multithreaded environment, with an identical setup, our proposed FER algorithm can achieve a greater than 42% encoding time saving when enabled at its peak, i.e., a sequence of still images (150 frames with a repeated photo for all frames) (Figure 58). It is worth mentioning that the PSNR and bitrate remain unchanged, i.e., there is no loss or image degradation compared to when the counterpart with the same setup but without FER is used. Regarding the total encoding time saving when compared to the setup without GPU-IME acceleration and FER disabled (original WTP), the gain is more than 52%, and, finally, compared with the setup with the default (single-threaded) original HEVC encoder, the gain is more than 2285%. These experimental results were the best we could obtain for the specific hardware setup in the identical case where all frames repeat the same stationary image. With common video sequences, the gain is more than 30%, since most of them have more or fewer moving blocks. So, our proposed work focused exclusively on the fraction execution resolver (FER) algorithm and the criteria that should be met to decide when to execute or discard the fraction part. The main criteria behind this decision are (a) zero motion vectors (MVs) for both XY vectors at the IME part and (b) zero distance displacement from the predicted search area center. This test is executed only once for the initial 64 × 64 (2N × 2N) block. For that purpose, we created a preliminary fast

test decision point (pFTDP) function which is a modified, cut-down version of the original diamond test zone search (TZ) algorithm [88].

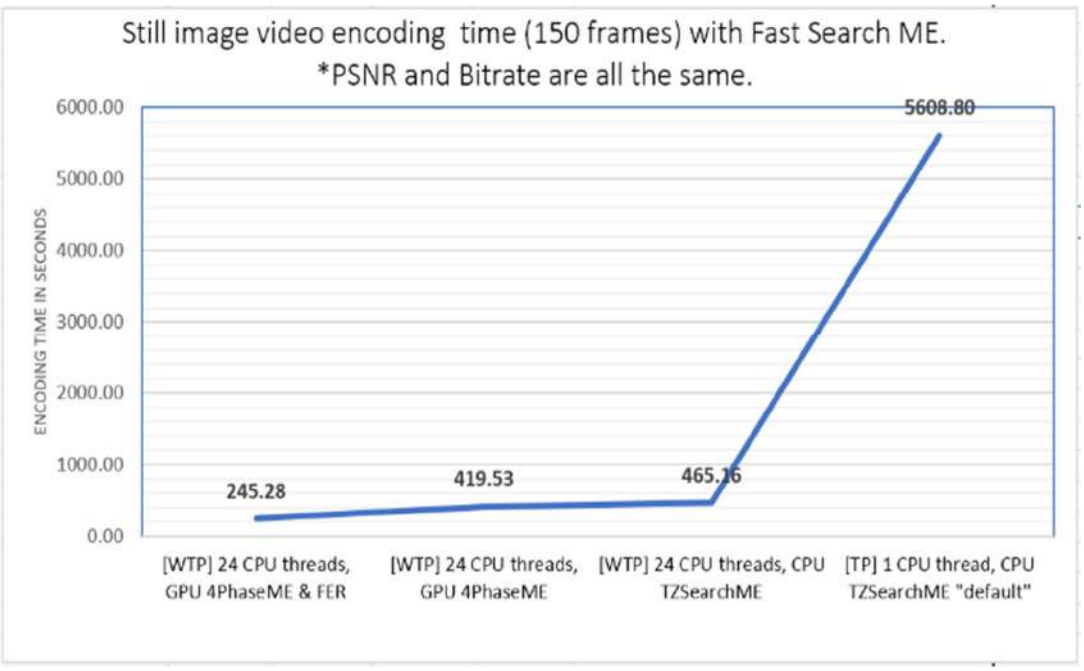


Figure 58. Encoding time with different setups. Fraction execution resolver (FER) achieves 42% encoding time savings compared to the same optimized setup without the FER extension ([WTP] 24 CPU threads, GPU 4PhaseME). Where [WTP] is used, bitrate and PSNR were re

5.2. Related Work

Many solutions have been presented in the past to accelerate FME computations and motion estimation in general. They are based on two categories: (a) software-optimized algorithms and (b) hardware implementations. Hardware implementations include non-CPU integrated circuits such as FPGAs (field-programmable gate arrays), ASICs (application-specific integrated circuits), and video card GPUs. The video card is an exception because it is compulsory for the system functionality itself. Thus, it should be treated as a de facto piece of hardware installed in the system, like CPUs, RAM, or any other important component, and not as extra hardware with a custom design or requirements.

A high-throughput hardware architecture for the FME interpolator was proposed [89], [90] using specialized hardware that always uses the FME part but in an optimized way. The results showed that a design architecture targeting an Altera Stratix III FPGA device is capable of processing real-time QFHD (3840 × 2160) resolutions at 60 fps with a 353.8 Mhz clock frequency

and 995 Mpixels/s for 7680 × 4320@30 fps at 188 MHz on a 64 nm VLSI CMOS chip. To the best of our knowledge, most hardware implementations focused only on FME optimization algorithms rather than skipping that part. An exception to this rule, with a similar approach to our basic skipping concept, was implemented on an FPGA hardware platform by F. H. Shajin [48] using a quadrant-based search algorithm. A zero-movement prejudgment technique (ZMP) is used to find out whether a block is static or not. This decision is used to decrease the computational complexity of the search method since ZMP can identify stationary blocks (distortion of the block is under threshold (T) forecast) even before the search procedure begins. The latter method uses a custom search algorithm (quadrant-based) that is tailored to work on a Verilog HDL with Virtex-5 technology and integrated with the Xilinx ISE Design Suite 14.5 hardware platform. It achieves lower search points compared with the hexagon, adaptive root pattern, and diamond search algorithms. Using lower search points, the quadrant-based search algorithm provides a PSNR of 10.906%, 2.03%, and 8.3562% and motion estimation time of 1.89%, 1.4693%, and 0.131% compared with the aforementioned algorithms.

Approximate unified FME filters (AUFF) are another hardware implementation [91] for interpolation filters that use two hardware designs. Using a 40 nm standard-cell library, with a power dissipation ranging from 22.04 to 62.06 mW, the hardware is capable of real-time interpolation of UHD videos of 2160p@60 fps and 4320p@60 fps when synthesized. Gogoi et al. [59], [53], using special hardware in HM 16.8, achieved an 8.725% and 9.072% time saving using a low-complexity algorithm with 38 search points in two different pattern structures (PS). It provides real-time encoding of 4096 × 2160@60 fps operated at the maximum frequency of 353 MHz. Although all mentioned solutions reduce the computational complexity, affecting the coding efficiency more or less significantly, undoubtedly they are specialized hardware implementations that do not focus on the end users but rather target companies and professionals that need real-time live streaming or broadcasting encoders. Much hardware equipment that everyone has broad access to, such as modern superpower GPGPUs (general-purpose graphics processing units), comes with next-generation, built-in encoding-dedicated hardware such as Nvidia's NVENC, AMD's AMF, and Intel's Quick Sync. They offer real-time streaming characteristics but with predefined codecs (i.e., h.264 or AV1) and image quality presets. Custom optimization tampering is not possible since the encoder is closely connected to the hardware.

Software solutions are easy to implement, have low costs, do not need extra hardware, and mostly target minimization of the computational complexity for low-end users. Fast two-step sub-pixel motion estimation is proposed in [81] which can reduce the sub-pixel search points significantly with negligible quality degradation. This algorithm uses six-pixel integer points for the fraction part. The half pixel is obtained from a sub-pixel error model surface that is first applied. Using the information resulting from the latter step and an additional integer point, a new error model surface is applied in a smaller area. Thus, the quarter-pixel point is obtained by minimizing the function. The best MV candidate is the one with the minimum RD cost among integer, half-pixel, and quarter-pixel points. For the standard Class B ParkScene video sequence,

the encoding time saving is 46.25% with a −0.02 PSNR quality drop compared with the hierarchical search (HS) algorithm.

In [6], an adaptive fractional pixel ME skipped scheme was proposed for low-complexity ME. In this scheme, all children's PU modes are classified into their root-type PU (2N × 2N) during the encoding. Based on the ME earning result of the root type, the FPME of all sub-pattern PU modes is skipped. The proposed algorithm reduces the encoding complexity and maintains a comparable encoding efficiency with the original HM video standard. Even though the basic idea is similar to our proposed work, the criteria to skip the FPME part and the computational load before that decision are very different. The root-type PU mode (Inter_2Nx2N) performs both integer and fraction ME parts. If the best motion vector of both IME and total ME (IME + FME) is equal, then the FME can be skipped. The fast search algorithm used in the IME part in HEVC is the test zone search (TZS), which, by default, has three different searching modes [88]: (a) the zonal search, (b) the raster search, and (c) the refinement. If the zonal search fails (best MV candidates are far away from the center of the search area), then a raster search is applied, which is essentially a full search in a down-sampled search area, a process that slows down the ME efficiency. Thousands or even millions of Inter_2Nx2N CTUs may exist in a video sequence to which a raster search may be applied, minimizing in this way the coding efficiency.

In this chapter, the pFTDP function is a lightweight zonal search only. This means no raster search is performed by default; thus, a huge time saving can be achieved for the encoder itself. Another major difference is that our proposed pFTDP does not even need to execute the fraction motion estimation part for the root-type PU mode to make its decision, which also adds extra computational overhead. Our skipping criteria are decided inside a lightweight preliminary function that calculates at the same time both the best MVs and minimum RD cost for all static blocks. Another adaptive fraction motion skipping process was proposed by Lee et al. [80] for multiview extension (video data captured from different viewpoints) of the standard HEVC encoder, called multiview HEVC (MV-HEVC). The decision-skipping condition, in this case, takes into account the interview frame encoding that performs a disparity estimation (DE) to reduce redundancies between neighboring views. In this case, the algorithm is designed to reduce the computational complexity of both ME and DE. When the result of 2N × 2X PU is $Inter_{2N}^{T} - I$, the faction and disparity part may be skipped. If the result is $Inter_{2N}^{IV} - I$, then only the fraction part is skipped. This FME skipping method uses a different video coding structure that performs the inter- and interview coding. The encoding time saving is 27.71% with a coding loss of 0.07% on average.

A parallel pre-motion estimation with IME and FME was proposed by Zhang et al. [61]. The efficiency of data access is achieved using a rapid mapping table algorithm. The GPU power is used to calculate SAD or SATD efficiently for blocks of different sizes for both IME and FME parts. A maximum of ×12.13 is the average speedup for IME and FME parts—both implemented in the GPU kernel—for the Class B sequence *Basketballdrive* (the only video sequence that can be compared with our experiments), which is much lower than our ×15.54 speedup experiment

result without the need to implement a GPU-dependent FME algorithm. Finally, another GPU-based parallel algorithm was proposed by Luo et al. [62]. They optimized the whole ME process by separately optimizing hierarchically the CTUs (coding tree units), the PUs (prediction units), and, finally, the MVs (motion vectors). The latter proposed scheme accelerates the motion estimation by up to 12.7 times with a BD-BR loss of 0.52% on average. Other implementations [51], [60], [52] made use of the GPU parallelization power either using only the IME part or the FME part, but many concessions were made to achieve the best parallelization results. All of the referenced works tried to reduce the FME computational complexity, either by using specialized hardware or by applying an optimized FME algorithm. So, our efforts in this work were focused mostly on software solutions, emphasizing the popular TZS algorithm, which is considered to be one of the fastest ME algorithms [92]. Our proposed fraction execution resolver approach differentiates mainly in three ways: (a) it does not require any special hardware equipment; (b) it is super-fast since CPU cores are latency optimized for a lightweight job, i.e., execution of the pFTDP function; and (c) it is easy to implement as the pFTDP decision-maker function is just a modified version of the well-known TZS algorithm. To sum up, we propose a software decision-maker algorithm to decide the invocation of the FME part based on the encoding process without any significant losses. The negative effects of a maximized tiling pattern concern the WTP encoding scheme itself more than the FER algorithm. These effects can be improved when a lower tiling pattern is selected. With super-high-resolution video sequences, the gain of the WTP/FER was expected to be higher even with lower tiling scheme patterns since the tile's internal parallel Wavefront progressively engages more CPU cores than low-resolution videos. Finally, our FER approach can be extended to include a multicore GPU/TPU-friendly fraction estimation algorithm or even a GPU-kernel-based pFTDP function to further improve the coding efficiency. Emerging deep learning methods can also be applied, such as implicit functions and attention mechanisms, to predict the motion vectors from the past GOP (group of pictures) and thus speed up the encoding process. Nevertheless, such extensions exceed the scope of this chapter, which proposes a fraction execution resolver and evaluates its potential merits, leaving further optimizations for future work.

5.3. Implementation

To evaluate our proposed algorithm, we used our proven multithreaded Wavefront per Tile Parallelism (WTP) encoding scheme [40]. This modified encoder uses tiles as a primary parallelism scheme but internally uses WPP for each tile separately with multiple CPU cores working in a parallel pattern. Extending the original WTP encoder, the GPU's fast search algorithm has been demonstrated (4PhaseFS) [82] to speed up the motion estimation calculations, which has been proven to be one of the most intensive tasks in a video encoder. As in our previous works, to achieve the best performance from our experiments, the application thread pool was fed with a

pre-selected number of threads that matched exactly with the physical CPU cores we had available for our setup. With respect to the video sequence's available resolution, the speedup automatically scales [64] depending on the CPU and GPU capabilities [65]. The GPU contributes significantly to the minimization of the motion estimation computation cost [66], [50], [58], [44], [46] since many ME vectorized data can take advantage of this powerful piece of hardware. The OpenCL framework [68] was the ideal candidate to set up the GPU kernel and schedule its cores to compute the integer estimation part (IME) in a parallel pattern. OpenCL is a universal general-purpose graphics processing unit programming framework that major GPU manufacturers support; it has an easy learning curve, and it seems to have higher performance rates in comparison with other competitive frameworks [70]. In the WTP parallelization scheme, each CPU thread executes its own copy of the code in a sequential pattern. When a thread meets the beginning of the motion estimation's IME part [71], it automatically reverts to the GPU kernel, trying in this way to distribute the encoding resource load among different hardware parts of the system. Even though the thread waits for the GPU to send back its calculation results, the OS frees some of the thread's resources, allowing in this way the normalization of the load balance of the encoder. The GPU exceeds the vectorized data calculations in comparison with the CPU because it encapsulates many thousands of cores that work in parallel. The GPU kernel program (a C-like program) uses a parallel reduction method [72] to calculate the distortion ratios. A bottom-up accumulative pattern is used, starting from 4 × 4 blocks as a base for the next SAD (sum of absolute differences) calculations, and progressively builds bigger PU blocks until it reaches a maximum of 64 × 64 root-type PU. An OpenCL work group of 16 × 16 is selected (256 cores in total) to comply with the initial 4 × 4 blocks, which consist of 256 pixels. Next, in the SAD calculation kernel steps, each core may apply to more than one pixel calculation offset to optimize the GPU resources. This means that each CPU thread uses a set of 256 GPU cores at the current time, so a total of 6.144 GPU cores may be needed for 24 CPU threads if all of them concurrently request resources from the GPU. This is an extreme possibility, but it is possible in UHD video sequences. In such a case, the GPU driver sets the extra demands in a wait queue. Finally, the kernel (GPU) returns to the caller (CPU) a set of indexed arrays—concerning the whole search area [67]—with the minimum distortion ratios as well as the corresponding motion vector. To minimize the data transfer costs, memory optimization strategies are used between the kernel and the host [74], [63].

The GPU kernel, as described above, takes into account only the integer part (IME), while the fraction part (FME) is left to the CPU since the parallelism of the fraction part inside the GPU is inefficient due to its dependencies. The data transfer between the CPU and GPU and vice versa is costly, and it is worthwhile only if heavy calculations in parallel are to be made. For light calculations such as the one made in the pFTDP function, the CPU is faster, without slowing down the whole process by sending the preliminary calculation directly to the GPU; thus, costly data transfers are avoided. Based on this idea, we developed a preliminary fast test decision point (pFTDP) using a modified but optimized version of the original TZSearch. The pFTDP function takes into account only the first part (zonal search) of the original TZ searching algorithm with a

maximum distance of two (2), i.e., checks only neighbor areas of the 64 × 64 block with distances of 1 and 2 according to the predicted starting position. After the end of this preliminary stage, if the MVs are still zero and the best distance is also zero, we can safely conclude that the current block is a static block. In this case, we can use the already calculated motion vectors and RD from this lightweight function to return the results back to the caller, discarding the fraction part. Otherwise, i.e., we have a non-static block, we call the GPU kernel to precalculate MVs and RDs for all sub-blocks [82] at once and use them as an input parameter for the fraction part (Figure 59) in the next stage.

The decision-making is executed only for the initial 2N × 2N of each block pattern, while the subsequent block patterns follow the flow taken from this initial decision, i.e., they either (a) obtain the MVs and RD from the lightweight pFTDP function while FME calculations are skipped (static blocks), or (b) use the precalculated GPU MVs and RD in case the pFTDP fails, and, consequently, FME calculations are executed for each one of them (non-static blocks) from the CPU. So, if the pFTDP succeeds, i.e., we have a static block, then there is no need to call the heavier GPU kernel since the gain would be negligible if not zero, and FME calculations are also bypassed since there is no need to obtain more accuracy from the specific static CTU block. The already calculated MVs and RD from this stage are sent directly to the caller function since they are the best candidates we can obtain so far, and the GPU is not involved. The extra optimized method applied here—as already mentioned—is the one-time execution of the pFTDP, i.e., when a 2N × 2N (64 × 64) block pattern is met.

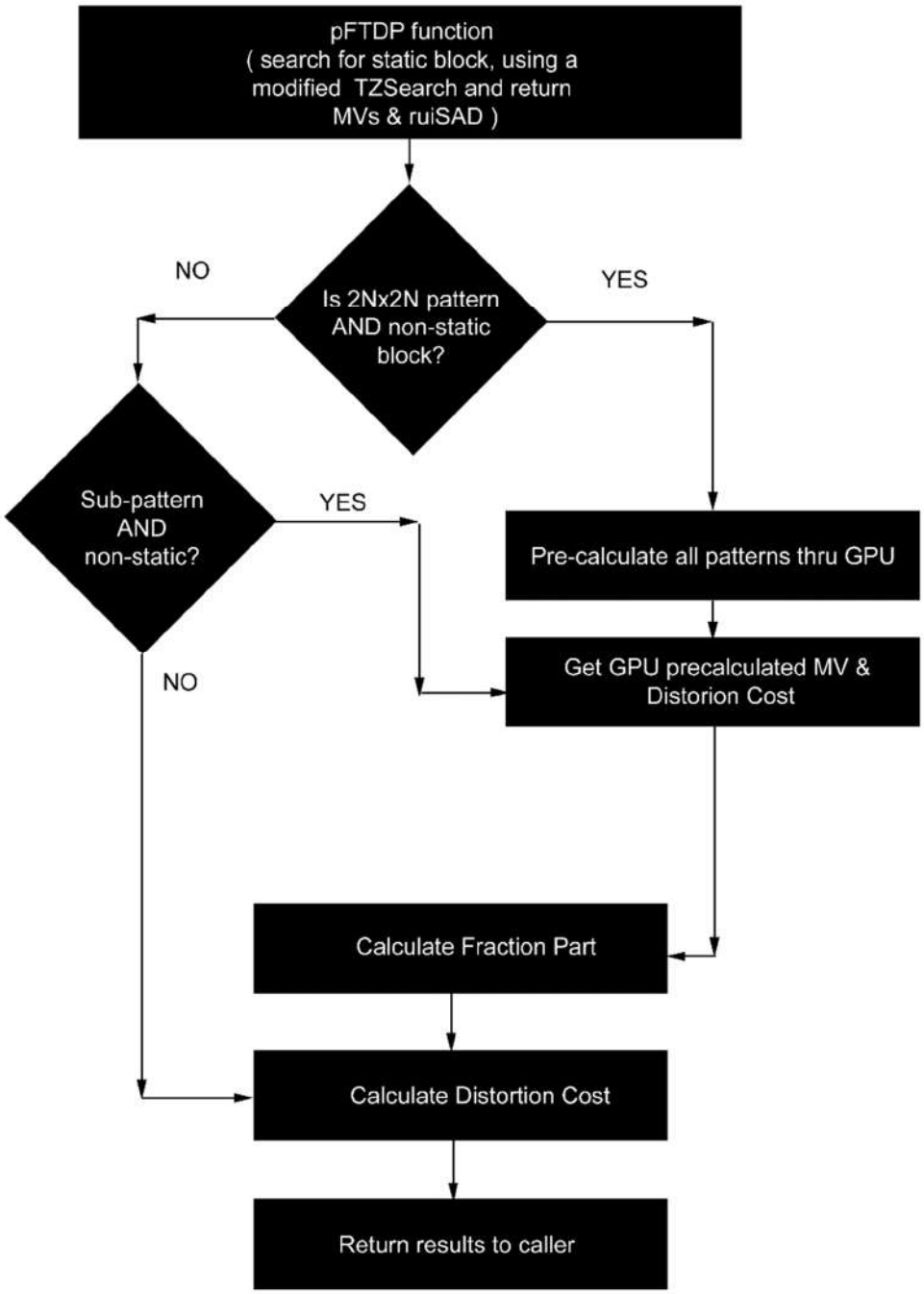


Figure 59. The fraction execution resolver (FER) flow chart.

According to the strategy described above, it is obvious that we are using a hybrid computation encoding scheme pattern for the motion estimation part; (a) non-static integer CTU

blocks are calculated in parallel using the GPU kernel and FME from the CPU, and (b) static CTU blocks, also in parallel but exclusively from the CPU, at the same time skip the FME part. Internally, our pFTDP algorithm still uses the eight-point diamond search around the starting point, but the only distances compared across the reference search area are those of step 1 and step 2. The *pFTDP* function always returns the evaluated results of *BestX*, *BestY*, and *BestDistance*. If the block is 2N × 2N, we immediately sign our code from the very beginning, either as a static or a non-static block, and, consequently, the fraction execution (FME) is invoked. This is valid also with all subsequent block patterns that follow the initial sign of the *pFTDP* and consequently execute or skip the FME calculations.

Lines 1–9 of Figure 60 are the initialization part of the algorithm. Lines 10–21 are the main loop where the TZ eight-point diamond search is applied. If the points around the center return zero for both X and Y motion vectors and the distance is also zero, we break the loop with true for the flag *isStillBlock*, which is used by the caller as a decision point for further processing (Figure 59). If the distance is higher than 2, then the *isStillBlock* flag returns to the caller with false. Lines 22–23 calculate the *ruiCost* and return back to the caller the memory address of the *BestX* and *BestY* candidates as well as the *isStillBlock* decision flag.

```
Input   : SearchRange passed from xMotionEstimation function
Output : IsStillBlock, BestX, BestY, ruiCost
Notes   : ptr denotes memory address of X variable. It is used for both input and output.

1:function pFTDP(ptr SearchRange, ptr iBestX, ptr iBestY, ptr ruiCost)
2:  IsStillBlock ← false;
3:  iBestX = iBestY ← 0;
4:  call function getPredictedXY(ptr iBestX, ptr iBestY)
5:  iStartX ← iBestX
6:  iStartY ← iBestY
7:  iDist ← 0
8:  uiBestRound ← 0
9:  uiBestSAD ← 0
10: while iDist < SearchRange
11:          call function xTZ8PointDiamondSearch(ptr iStartX, ptr iStartY, in iDist, ptr iBestX, ptr iBestY, ptr uiBestSAD, ptr uiBestRound)
12:          if  uiBestRound >= 3
13:          begin
14:            if uiBestDistance = 0  AND iBestX = 0 AND iBestY = 0
15:            begin
16:                IsStillBlock ← true;
17:            end if
18:            exit from loop
19:          end if
20:          iDist ← iDist mul 2
21: end while
22: ruiCost ← call function calculateCost(ptr iBestX, ptr iBestY, ptr uiBestSAD)
23: return IsStillBlock
24:end function
```

Figure 60. The preliminary fast test decision point (pFTDP) algorithm.

5.4. Experiments

Implementation Details

Our code was implemented in HM 16.17 reference software. The programming language used to optimize the encoder was C++. Furthermore, the AMP (asymmetric motion partitioning) inside the configuration file was enabled to increase the accuracy of the motion estimation part when it is needed and, consequently, increase the computational load for comparison reasons between CPU and GPU setups. Finally, the thread pool was configured with 24 CPU threads to comply with the physical CPU cores of our setup. We kept the CPU thread assignment strategy and routing used in our previous work [82] for comparison reasons, meaning that, if any CTU encoding dependency constraints are met, the active CPU thread is dismissed immediately back to the threading pool to avoid wait state cycles and to free up unused CPU/GPU resources.

Setup

Experiments were conducted on a two 12-core 2.20 GHz Intel Xeon E5-2650 Linux server for base frequency, with 2.90 GHz for turbo mode and 1.20 GHz for power-save mode. The GPU computations for the integer motion estimation part were made with a PCI Express 3.0 × 16 Nvidia Quadro P4000 video card. Quadro P4000 was introduced in April 2016 by Nvidia, implementing at the time the new Pascal architecture that allows multi-kernel execution as independent units, and it combines 1792 CUDA cores with 8 GB of memory.

As shown in Table 4, common Class B and A video sequences [93] were used in our experiments. Due to the number of computationally intensive tasks and time restrictions, we used the first 150 frames of each video sequence to evaluate the proposed FER algorithm. A GOP size of 4 was used with a low-delay (LD) setup, meaning a <I> frame was initially used, followed by consecutive <B> frames. The following parameters were used unless otherwise stated: 64 × 64 CTU size, TZ fast search mode, one slice, max partitioning depth of 4, 24 CPU threads, and default QP = 32. Tile partitions 2 × 2, 3 × 3, and 6 × 4 remained the same as in our previous work [10] for comparison reasons.

Class	Sequence Name	Resolution
B	Basketballdrive	1920X1080 50fps
B	Bqterrace	1920X1080 60fps

B	Cactus	1920X1080 50fps
B	Kimono	1920X1080 24fps
B	Parkscene	1920X1080 24fps
A	Peopleonstreet	2560X1600 30fps
A	Traffic	2560X1600 30fps

Table 4. Test video sequences.

Coding Efficiency Results

In most of our experiments, we measured the speedup using the optimized WTP (GPU-IME accelerated) with FER over the default TZ search (one thread, sequential) for the same settings with different QPs but the same number of threads. The average speedups for Class A and B sequences are shown respectively in Figure 61 and Figure 62. In both plots, the average rate speedups are more than ×12 for Class B videos and more than ×14 for Class A videos. In Figure 63, it can be observed that the BD-Rate is proportional to the tiling scheme used, i.e., a partitioning scheme with a smaller number of tiles has lower quality loss and vice versa. This was expected because tiles reset the context variables of the CABAC (context-based adaptive binary arithmetic coding) at their initialization step. This means that the CABAC learning process starts over again with zero coding units being used from the neighboring prediction encoding history. Thus, a PSNR decrease and BD-Rate increase are expected. This means that a bigger tile pattern scheme, i.e., 6 × 4, capitalizes the CPU usage in our setup (all 24 CPU physical cores are involved), but, at the same time, results in a relatively higher BD-Rate.

The same BD-PSNR [39] decrease effect can also be observed in Figure 64, where a −0.22 quality drop is plotted for the *PeopleOnStreet* video sequence. For this plot, the summed average PSNR values for all QPs (= 22, 27, 32, 37) were used to calculate and plot the final BD-PSNR. In Figure 65, we show the absolute speedup increase compared with the default HEVC encoder benchmarks results when FER is enabled in our GPU-IME accelerated WTP [82] and without this. Finally, the last plots (Figure 66 and Figure 67) show the PSNR (image quality) and bitrate overhead when FER is enabled in our WTP (GPU-IME accelerated) encoder. The PSNR appears to be slightly improved (positive values in Figure 66), while, in other cases (negative values), the image quality drops but is proven to be negligible; −0.05 in the worst-case scenario. The explanation for the slight image quality improvement is that the *pFTDP* algorithm is more accurate than the GPU-IME 4Phase fast search counterpart in some cases, i.e., in static blocks. The bitrate overhead shown in Figure 67 seems to be almost consistent in both experiments (FER

enabled/disabled). The slight improvements or deteriorations observed are more related to the IME algorithms themselves, i.e., CPU-based *pFTDP* versus GPU-based 4PhaseME counterpart.

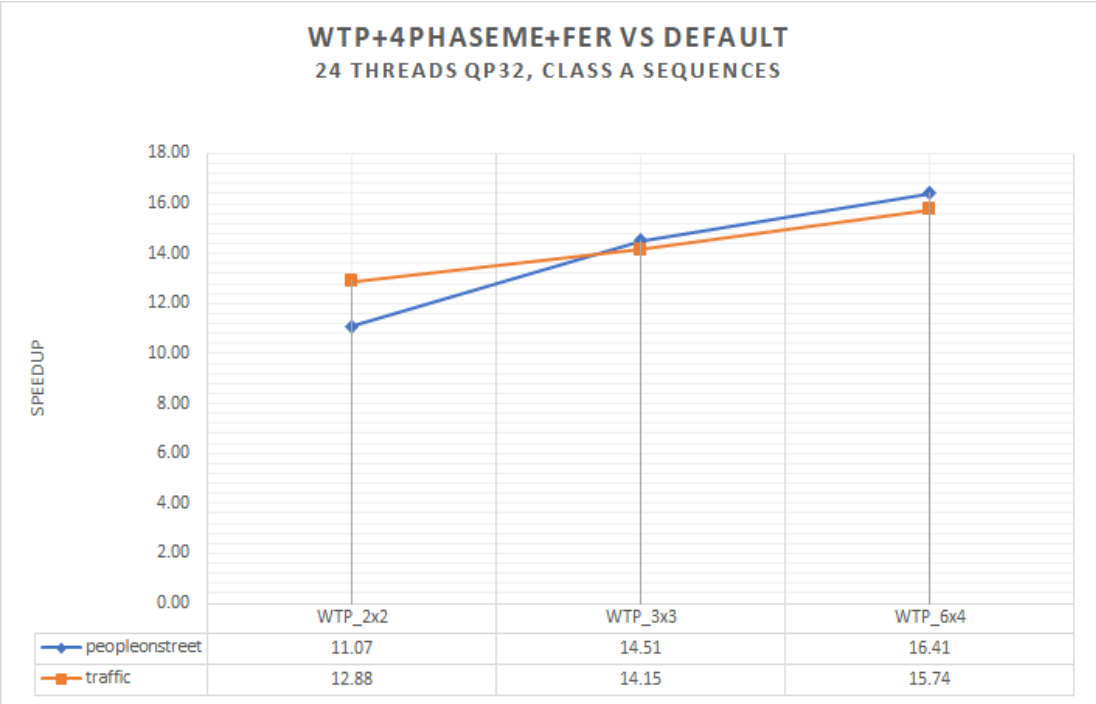
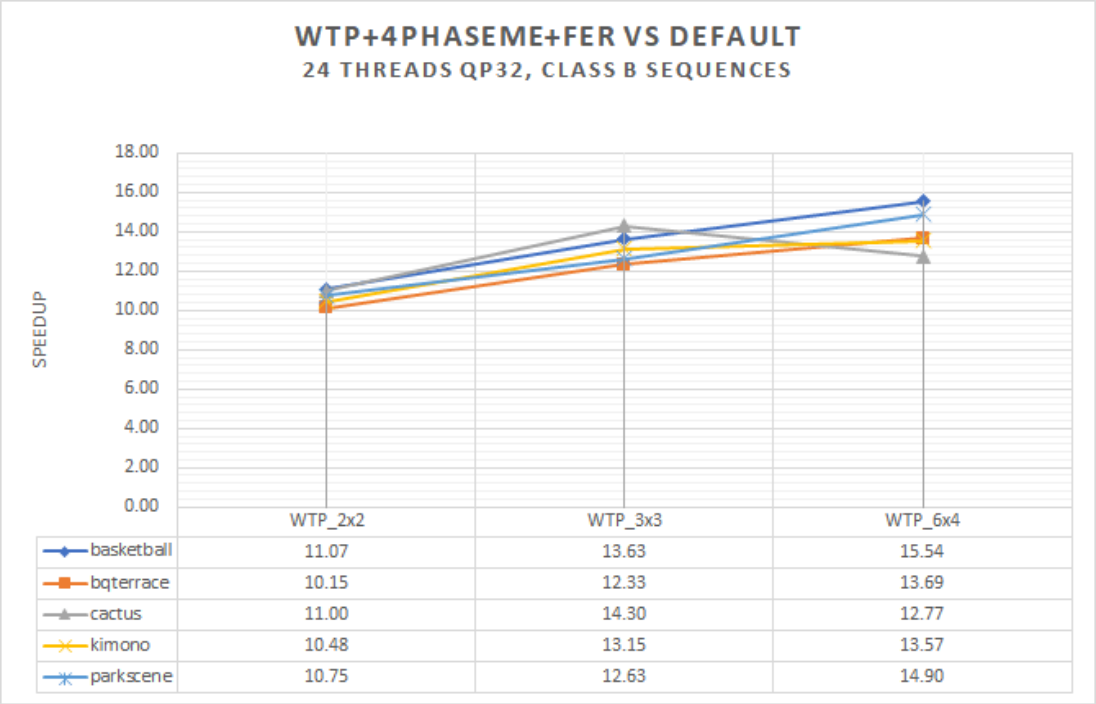


Figure 61. Class A sequences using FER with WTP speedup for QP = 32 and 24 threads versus default.



	WTP_2x2	WTP_3x3	WTP_6x4
basketball	11.07	13.63	15.54
bqterrace	10.15	12.33	13.69
cactus	11.00	14.30	12.77
kimono	10.48	13.15	13.57
parkscene	10.75	12.63	14.90

Figure 62. Class B sequences using FER with WTP speedup for QP = 32 and 24 threads versus default.

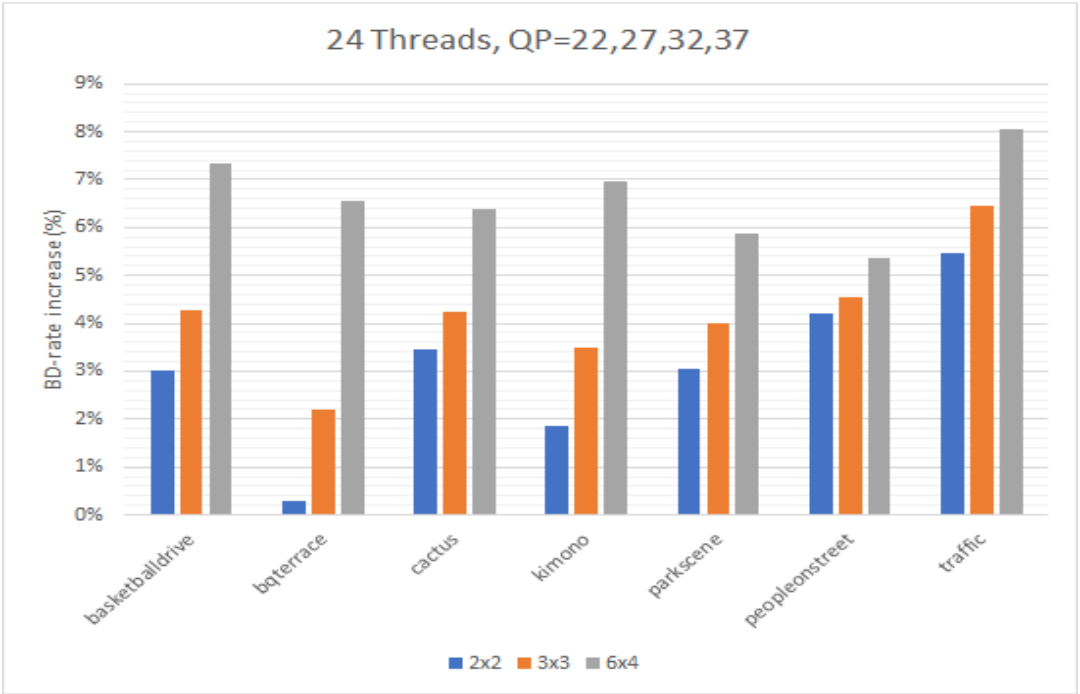


Figure 63. Increase of BD−Rate using FER with WTP for QP = 22, 27, 32, 37, and 24 threads versus default (positive values indicate bigger files).

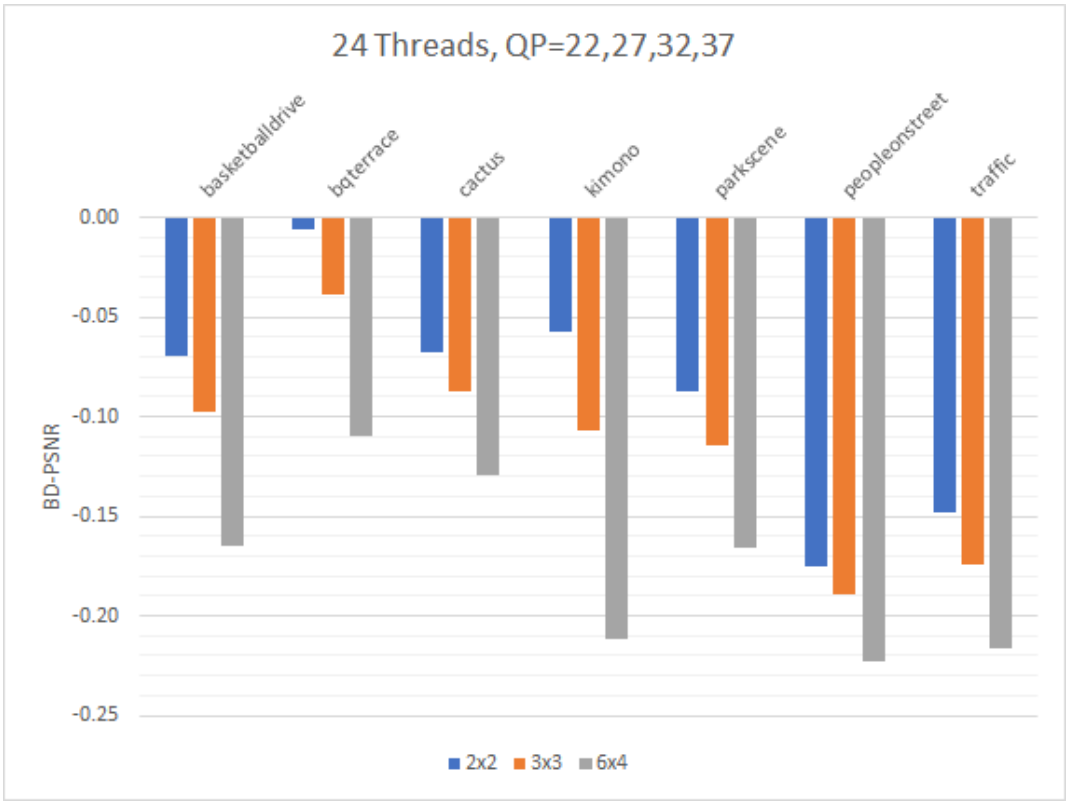


Figure 64. Decrease of BD–PSNR using FER with WTP for QP = 22, 27, 32, 37, and 24 threads versus default (negative values indicate a quality drop).

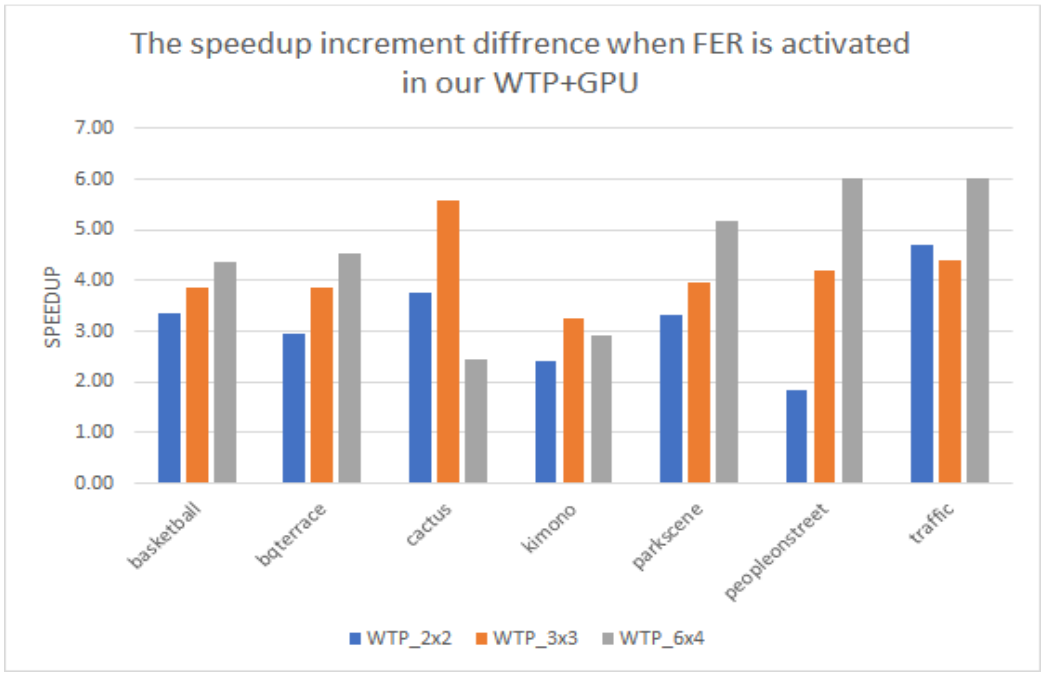


Figure 65. The absolute speedup increments when FER is enabled/disabled in our GPU-IME accelerated WTP encoder compared with the default HEVC encoder benchmark results, i.e., Diff (FER enabled–FER disabled).

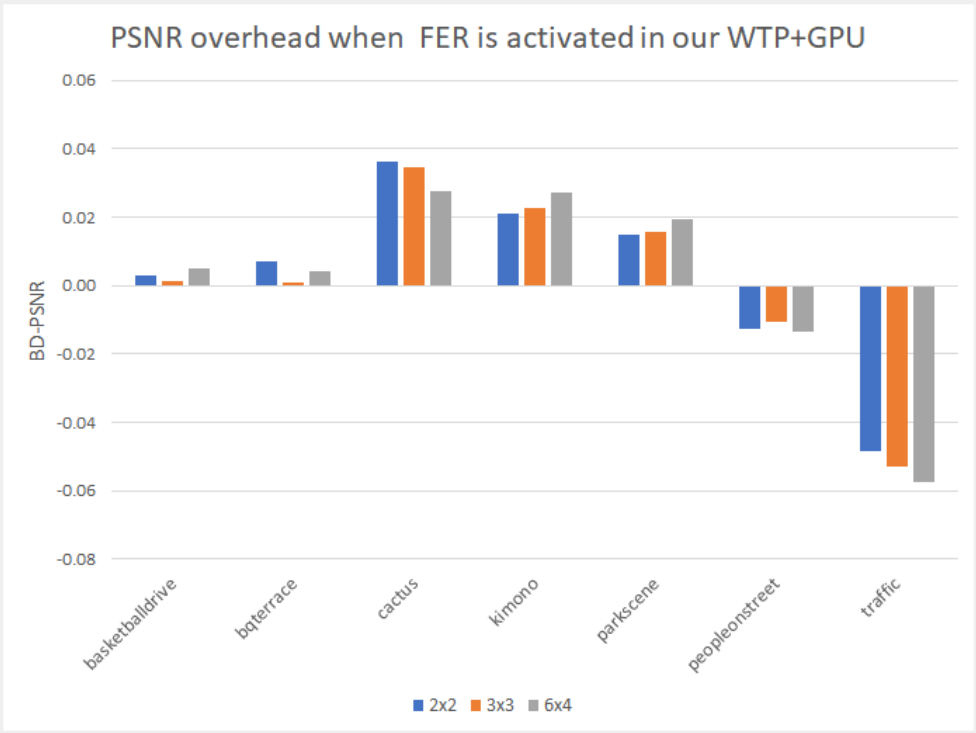


Figure 66. The PSNR overhead when FER is enabled in our GPU–IME accelerated WTP encoder. Positive values indicate a quality improvement, while negative values indicate a quality drop.

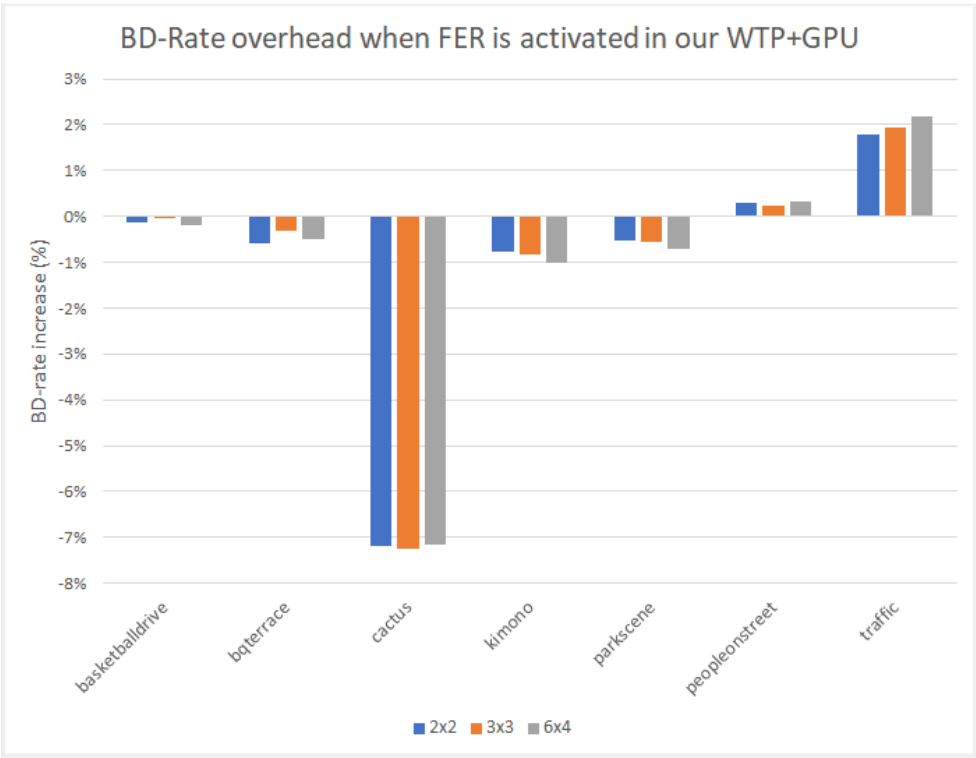


Figure 67. The bitrate overhead when FER is enabled in our GPU–IME accelerated WTP encoder. Positive values indicate bigger files.

5.5. Summary and Discussion

In this chapter, we presented a fraction execution resolver (FER) algorithm that works in parallel and non-parallel encoders and aims to minimize the encoding time and overcome the standalone deficiencies of complex computations and specialized hardware. Results demonstrated that the FER can achieve an average speedup of ×14.3 for Class A video sequences and ×12.6 for Class B video sequences with an average PSNR decrease for both Class A and B video sequences of −0.12 and an approximate 4.6% increase for the bitrate. The average speedup enabled by the FER, in comparison with the same sequences and identical setup without the FER, is about ×3.95. Video sequences that have many static blocks gain higher time-saving encoding times using the proposed FER algorithm. It is worth highlighting that the FER concept can easily be adapted with many other video encoding standards, such as newer VVC since it is not hardware dependent, and similar decision criteria can be applied. According to the results demonstrated, a PSNR and bitrate overhead occurred. These negative effects are more related to the tile encoding scheme itself and the way CABAC works internally. The FER itself has been proven to add negligible overhead in PSNR and bitrate metrics. To combine speed and quality, an encoding balance scheme should be used via a lower tiling pattern, preferably a 2 × 2 scheme, to obtain the best of both worlds. Cumulatively, the experimental results confirmed the validity of our motivation, namely, that we can benefit from a software fraction execution resolver without any extra hardware costs or considerable quality loss. The gain is further increased when video sequences used for encoding have more static blocks than others; however, practically, static blocks always exist, more or less, in common video material.

Chapter 6

Conclusion and Future Work

6.1. Conclusions

In this thesis, we proposed algorithms and techniques for accelerating video encoding through parallel processing.

In Chapter 3, we introduced a new parallel encoding scheme to effectively combine the best of two very well-known parallelization schemes, i.e., tile and wavefront parallelism. Results demonstrated that WTP can achieve valid trade-offs between speedup and coding efficiency while overcoming the natural time performance limitations of the wavefront. The average speedup, in comparison with the same sequences and identical setup without any parallelization scheme, is more than x6.30 for both Class A and Class B videos.

In Chapter 4, we presented a novel GPU fast search motion estimation algorithm that works in parallel with a CPU-based multi-core encoding scheme WTP, that aims to minimize the encoding time. Results demonstrated, that the 4Phase GPU fast search motion estimation algorithm can achieve an average speedup of x8.84 for Class B videos and x9.60 for Class A compared with the original sequential TZSearch. Also, the GPU contribution in the original WTP can increase the speed by an average of 18%. To reduce the negative effects of BD-Rate increase and BD-PSNR decrease, a 2×2 tile scheme is suggested. Finally, weaker CPUs with lower core frequencies can benefit more from the GPU addition in the encoding chain than CPUs that run at higher frequencies. Cumulatively, the experimental results confirm the validity of our motivation, namely that we can benefit from the GPU computational power, especially when massive calculations take place in parallel. The mentioned speedups can further increase when low-frequency CPUs are used in conjunction with a powerful GPU without any significant quality loss.

Since the Achilles heel of all GPU-based algorithms is the time consumed for the data transfers using slower channels than CPU registers, it is believed that powerful integrated GPUs that lie onto the same die as the CPUs or new versions of PCIe will boost that process without any extra costs

In Chapter 5, we presented a fraction execution resolver (FER) algorithm that works in parallel and non-parallel encoders and aims to minimize the encoding time and overcome the standalone deficiencies of complex computations and specialized hardware. Results demonstrated that the FER can achieve an average speedup of x14.3 for Class A video sequences and x12.6 for Class B video sequences with an average PSNR decrease for both Class A and B video sequences of -0.12 and an approximate 4.6% increase for the bitrate. The average speedup enabled by the FER, in comparison with the same sequences and identical setup without the FER, is about x3.95. Video sequences that have many static blocks gain higher time-saving encoding times using the proposed FER algorithm. It is worth highlighting that the FER concept can easily be adapted with many other video encoding standards, such as newer VVC since it is not hardware dependent, and similar decision criteria can be applied. According to the results demonstrated, a PSNR and bitrate overhead occurred. These negative effects are more related to the tile encoding scheme itself and the way CABAC works internally. The FER itself has been proven to add negligible overhead in PSNR and bitrate metrics. To combine speed and quality, an encoding balance scheme should be used via a lower tiling pattern, preferably a 2x2 scheme, to obtain the best of both worlds. Cumulatively, the experimental results confirmed the validity of our motivation, namely, that we can benefit from a software fraction execution resolver without any extra hardware costs or considerable quality loss. The gain is further increased when video sequences used for encoding have more static blocks than others; however, practically, static blocks always exist, more or less, in common video material.

6.2. Future Work

As video resolutions increase, so does encoding complexity to achieve higher compression. Future challenges may include:

Deep Learning AI Methods

Deep learning is an AI method that enables computers to process data similarly to how the human brain works. With this approach, deep learning models can identify intricate patterns in images, text, sounds, and other forms of data to generate precise predictions and insights. Emerging deep learning methods can be applied, such as implicit functions and attention

mechanisms, to predict the motion vectors from the past GOP (group of pictures) and thus speed up the encoding process.

Distributed Video Encoding With Cloud Computing

Cloud computing provides numerous benefits, including the ability to allocate resources on demand, accommodate multiple requests, and offer elasticity and resiliency. Video encoding can be effectively deployed on such infrastructures as they can adapt resources to the workload, provide high availability, and enable ubiquitous access. They utilize an elastic pool of workers and provide all the required hardware to accelerate the encoding procedure. A distributed video encoder can be deployed on top of a cloud architecture to divide the video encoding process into multiple jobs that can be dynamically assigned to the elastic pool of workers.

Bibliography

[1] G. Papaioannou, M. Koziri, P. Papadopoulos, T. Loukopoulos and I. Anagnostopoulos, "Tile Based Wavefront Parallelism in HEVC," in *Proceedings of the 15th International Workshop on Semantic and Social Media Adaptation and Personalization*, Zakynthos, Greece, 2020.

[2] G. Papaioannou, M. Koziri and I. Anagnostopoulos, "On Combining Wavefront and Tile Parallelism with a Novel GPU-friendly Fast Search," *Electronics,* vol. 12, 2023.

[3] G. I. Papaioannou, M. Koziri, T. Loukopoulos and I. Anagnostopoulos, "Fraction Execution Resolver Using a Hybrid Multi-CPU/GPU Encoding Scheme," *Electronics,* vol. 12, no. 17, 2023.

[4] A. K. Jain, Fundamentals of Digital Image Processing, NJ: Prentice Hall, 1989, pp. 150-153.

[5] "WikiPedia," [Online]. Available: https://en.wikipedia.org/wiki/Compression_artifact. [Accessed 1 Sep 2023].

[6] W. B. Pennebaker and J. L. Mitchell, JPEG: Still Image Data Compression Standard (Digital Multimedia Standards S), Springer, 1993.

[7] J. Ohm and G. Sullivan, "High efficiency video coding: the next frontier in video compression [standards in a nutshell," *Signal Processing Magazine,* vol. 30, p. 152–158, Jan 2013.

[8] B. Bross, "Overview of the Versatile Video Coding (VVC) Standard and its Applications," *IEEE Transactions on Circuits and Systems for Video Technology,* vol. 31, pp. 3736-3764, Oct. 2021.

[9] R. Trafford-Jones, "Video: The New Video Codec Landscape – VVC, EVC, HEVC, LC-EVC, AV1 and more," 25 Nov 2020. [Online]. Available: https://thebroadcastknowledge.com/tag/vc6/. [Accessed Sep 1 2023].

[10] ITU-T, "H.261 : Video codec for audiovisual services at p x 64 kbit/s," in *ITU-T Recommendation H.261*, 1993.

[11] G. J. Sullivan, J.-R. Ohm, W.-J. Han and T. Wiegand, "Overview of the High Efficiency Video Coding," *IEEE Transactions on Circuits and Systems for Video Technology,* vol. 22.

[12] J.-R. Ohm, G. J. Sullivan, H. Schwarz, T. K. Tan and T. Wiegand, "Comparison of the Coding Efficiency of Video Coding Standards—Including High Efficiency Video Coding (HEVC)," *IEEE Transactions on Circuits and Systems for Video Technology,* vol. 22, Dec 2012.

[13] K. R. Vijayanagar, "OTT Verse," Dec 2020. [Online]. Available: https://ottverse.com/i-p-b-frames-idr-keyframes-differences-usecases/. [Accessed Aug 2023].

[14] G. Laroche, J. Jung and B. Pesquet-Popescu, "RD Optimized Coding for Motion Vector Predictor Selection," *IEEE Transactions on Circuits and Systems for Video Technology,* vol. 18, no. 9, p. 1247–1257, Sep 2008.

[15] W. -J. Han et al., "Improved video compression efficiency through flexible unit representation and corresponding extension of coding tools," *IEEE Transactions on Circuits and Systems for Video Technology,* vol. 20, pp. 1709-1720, Dec 2010.

[16] "Wikipedia," [Online]. Available: https://en.wikipedia.org/wiki/Block-matching_algorithm. [Accessed 18 Aug 2023].

[17] Recommendation ITU-T BT.500-11, "Methodology for the subjective assessment of the quality of television pictures," in *ITU-T*, 2002.

[18] I. E. G. Richardson, H.264 and MPEG-4 Video Compression, West Sussex, England: John Wiley & Sons Ltd, 2003.

[19] C. Shannon, A mathematical theory of communications, Bell Syst Tech, 1948, p. 379–423.

[20] D. Marpe, H. Schwarz and T. Wiegand, "Context-based adaptive binary arithmetic coding in the H.264/AVC video compression standard," *IEEE Trans CSVT,* p. 620–636, 2003.

[21] K. McCann, B. Bross, S. Sekiguchi and HanWJ, "HEVC test model 1 (HM 1) encoder description," in *Document JCTVC-C402, Joint Collaborative Team on Video Coding (JCT-VC)*, Guangzhou, 2010.

[22] S. Akramullah, I. Ahmad and M. L. Liou, "A Data-Parallel Approach for Real-Time MPEG-2 Video Encoding," *Journal of Parallel and Distributed Computing,* vol. 30, pp. 129-146, 1995.

[23] G. J. Sullivan, J.-R. Ohm, W.-J. Han and T. Wiegand, "Overview of the High Efficiency Video Coding (HEVC) standard," *IEEE Trans. Circuits Syst. Video Technol,* vol. 22, pp. 1649-1668, Dec. 2012.

[24] T. Wiegand, G. J. Sullivan, G. Bjontegaard and A. Luthra, "Overview of the H.264/AVC video coding standard," *IEEE Transactions on Circuits and Systems for Video Technology,* vol. 13, pp. 560-576, 2003.

[25] Z. Zhao and P. Liang, "Data partition for wave front parallelization of H. 264 video encoder," *Proc. 2006 IEEE International Symposium on Circuits and Systems (ISCAS),* p. 4, 2006.

[26] C. C. Chi, M. Alvarez-Mesa, B. Juurlink, G. Clare, F. Henry, S. Pateux and T. Schierl, "Parallel Scalability and Efficiency of HEVC Parallelization Approaches," *IEEE Transactions on Circuits and Systems for Video Technology,* vol. 22, pp. 1827-1838, 2012.

[27] M. Coban and Y. Wang, "Tiles and Wave Front Parallel Processing". United States Patent US20140003531A1, 2 Jan 2014.

[28] Y. Zhao, L. Song, X. Wang, M. Chen and J. Wang, "Efficient realization of parallel HEVC intra encoding," *Proc. 2013 IEEE International Conference Multimedia and Expo Workshops (ICMEW),* pp. 1-6, 2013.

[29] Z. Y. Wang, S. F. Dong, R. G. Wang, W. M. Wang and W. Gao, "Dynamic macroblock wave front parallelism for parallel video coding," *J.Vis. Commun. Image Represent,* vol. 28, pp. 36-43, Apr 2015.

[30] S. Radicke, J.-U. Hahn, C. Grecos and Q. Wang, "A multi-threaded full-feature HEVC encoder based on wave front parallel processing," *Proc. Int. Conf. Signal Process. Multimedia Appl. (SIGMAP),* pp. 90-98, Aug 2014.

[31] K. Chen, J. Sun, Y. Duan and Z. Guo, "A Novel Wavefront-Based High Parallel Solution for HEVC Encoding," *IEEE Transactions on Circuits and Systems for Video Technology,* vol. 26, pp. 181-194, 2016.

[32] Z. Wen, B. Quo, J. Liu, J. Li, Y. Lu and J. Wen, "Novel 3D-WPP algorithms for parallel HEVC encoding," in *2016 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, 2016.

[33] K. Misra, A. Segall, M. Horowitz, S. Xu, A. Fuldseth and M. Zhou, "An Overview of Tiles in HEVC," *IEEE Journal of Selected Topics in Signal Processing,* vol. 7, pp. 969-977, 2013.

[34] M. Shafique, M. U. K. Khan and J. Henkel, "Power efficient and workload balanced tiling for parallelized high efficiency video coding," in *2014 IEEE International Conference on Image Processing (ICIP)*, 2014.

[35] C. Blumenberg, D. Palomino, S. Bampi and B. Zatt, "Adaptive content-based Tile partitioning algorithm for the HEVC standard," in *2013 Picture Coding Symposium (PCS)*, 2013.

[36] I. Storch, D. Palomino, B. Zatt and L. Agostini, "Speedup-aware history-based tiling algorithm for the HEVC standard," in *2016 IEEE International Conference on Image Processing (ICIP)*, 2016.

[37] Y.-J. Ahn, T.-J. Hwang, D.-G. Sim and W.-J. Han, "Implementation of fast HEVC encoder based on SIMD and data-level parallelism," *EURASIP J. Image and Video Processing,* vol. 14, 2014.

[38] M. Koziri, P. K. Papadopoulos, N. Tziritas, N. Giachoudis, T. Loukopoulos, S. U. Khan and G. I. Stamoulis, "Heuristics for tile parallelism in HEVC," in *2017 25th European Signal Processing Conference (EUSIPCO)*, 2017.

[39] C.-H. Chan, C.-C. Tu and W.-J. Tsai, "Improve load balancing and coding efficiency of tiles in high efficiency video coding by adaptive tile boundary," *Journal of Electronic Imaging,* vol. 26, pp. 26 – 26-10, 2017.

[40] I. Storch, D. Palomino, B. Zatt and L. Agostini, "Speedup evaluation of HEVC parallel video coding using Tiles," *Journal of Real-Time Image Processing,* pp. 1-18, 2019.

[41] P. K. Papadopoulos, M. Koziri and T. Loukopoulos, "A Fast Heuristic for Tile Partitioning and Processor Assignment in Hevc," in *2018 25th IEEE International Conference on Image Processing (ICIP)*, 2018.

[42] G. Bjøntegaard, "Calculation of average PSNR Differences between RD-curves," in *VCEG-M33, Tech. Rep. Proceedings of the ITU-T Video Coding Experts Group (VCEG) Meeting*, Austin, TX, USA, 2–4 April 2001.

[43] F. Belghith, H. Kibeya, H. Loukil, M. Ben Ayed and N. Masmoudi, "A new fast motion estimationalgorithm using fast mode decision for high-efficiency video codingstandard," *J. Real-Time Image Process.,* vol. 11, p. 675–691, 2014.

[44] H. Kibeya, F. Belghith, M. Ben Ayed and N. Masmoudi, "Fast coding unit selectionand motion estimation algorithm based on early detection of zero blockquantified transform coefficients for high-efficiency video coding standard.," *IET Image Process.,* vol. 10, p. 371–380, 2016.

[45] P. Nalluri, L. Alves and A. Navarro, "Improvements to TZ search motionestimation algorithm for multiview video coding," *Proceedings of the 2012 19th International Conference on Systems, Signals and Image Processing (IWSSIP),* pp. 11-13, 2012.

[46] P. Nalluri, L. Alves and A. Navarro, "Complexity reduction methods for fastmotion estimation in HEVC.," *Signal Process. Image Commun.,* vol. 39, p. 280–292, 2015.

[47] J. Xiong, H. Li, F. Meng, Q. Wu and K. Ngan, "Fast HEVC inter CU decision based onlatent SAD estimation," *IEEE Trans. Multimed.,* vol. 17, p. 2147–2159, 2015.

[48] R. Khemiri, N. Bahri, F. Belghith, F. Sayadi, M. Atri and N. Masmoudi, "Fast motion estimation for HEVCvideo coding.," in *Proceedings of the 2016 International Image Processing, Applications and Systems (IPAS)*, Hammamet, Tunisia, 5–7 November 2016.

[49] P. Nalluri, L. Alves and A. Navarro, "A novel SAD architecture for variable block size motion estimation in HEVC video coding," in *Proceedings of the 2013 International Symposium on System on chip (SoC)*, Tampere, Finland, 23–24 October 2013.

[50] F. Shajin, P. Rajesh and M. Raja, "An Efficient VLSI Architecture for Fast Motion Estimation Exploiting Zero Motion Prejudgment Technique and a New Quadrant-Based Search Algorithm in HEVC," *Circuits Syst. Signal Process,* vol. 41, p. 1751–1774, 2021.

[51] Y. Qiu, J. Jiao, Y. Tang, Y. Liu, J. Ren and X. Zeng, "A Heterogeneous HEVC Video Encoder System Based on Two-Level CPU-FPGA Computing Architecture," in *Proceedings of the International Conference on ASIC (ASICON)*, Kunming, China, 26–29 October 2021.

[52] D. Lee, D. Sim, K. Cho and S. Oh, "Fast motion estimation for HEVC on the graphics processing unit (GPU)," *Real-Time Image Process,* vol. 12, p. 549–562, 2015.

[53] G. Cebrian-Marquez, V. Galiano, H. Migallon, J. Martinez, P. Cuenca and O. Granado, "Heterogeneous CPU plus GPU approaches for HEVC," *J. Supercomput.,* vol. 75, p. 1215–1226, 2019.

[54] R. Zamanshoar and A. Mansouri, "Parallel Motion Estimation Based on GPU and Combined GPU-CPU," *Adv. Parallel Comput.,* vol. 33, p. 210–220, 2018.

[55] S. Gogoi and R. Peesapati, "A hybrid hardware oriented motion estimation algorithm for HEVC/H.265," in *Proceedings of the IEEE International Symposium on Smart Electronic Systems (iSES)*, Rourkela, India, 18–22 December 2019.

[56] W. Chen and H. Hang, "264/AVC motion estimation implementation on compute unified device architecture (CUDA)," in *Proceedings of the 2008 IEEE International Conference on Multimedia and Expo*, Hannover, Germany, 23–26 April 2008.

[57] R. Rodriguez and J. Martinez, "Reducing complexity in H.264/AVC motion estimation by using a GPU," in *Proceedings of the 2011 IEEE 13th International Workshop on Multimedia Signal Processing (MMSP)*, Hangzhou, China, 17–19 October 2011.

[58] S. Kim, D. Lee, C. Sohn and S. Oh, "Fast motion estimation for HEVC with adaptive search range decision for CPU and range," in *Proceedings of the 2014 IEEE China Summit &*

International Conference on Signal and Information Processing (ChinaSIP), Xi'an, China, 9–13 July 2014.

[59] X. Jiang, T. Song, T. Shimamoto and L. Wang, "High efficiency video coding (HEVC) motion estimation parallel algorithms on GPU," in *Proceedings of the 2014 IEEE International Conference on Consumer Electronics-Taiwan*, Taipei, Taiwan, 26–28 May 2014.

[60] R. Khemiri, H. Kibeya, F. Sayadi and N. Bahri, "Optimisation of HEVC motion estimation exploiting SAD and SSD GPU-based implementation," *IET Image Process,* vol. 12, p. 243–253, 2018.

[61] S. Gogoi and R. Peesapati, "Design and Implementation of an Efficient Multi-Pattern Motion Estimation Search Algorithm for HEVC/H.265," *IEEE Trans. Consum. Electron.,* vol. 67, p. 319–328, 2021.

[62] Y. Xue, C. Zhang, H. Su, J. Ren and L. Xiao, "A Highly Parallel and Scalable Motion Estimation Algorithm with GPU for HEVC," *Sci. Program.,* 2017.

[63] T. Zhang, X. An, X. Zhao and X. Gao, "A Novel Parallel Motion Estimation Design and Implementation on GPU," *IEEE Access,* vol. 7, p. 11747–11753, 2019.

[64] F. Luo, S. Wang, S. Wang and X. Zhang, "GPU-Based Hierarchical Motion Estimation for High-Efficiency Video Coding," *IEEE Trans. Multimed.,* vol. 21, p. 851–862, 2019.

[65] NVIDIA, CUDA C Programming Guide, Santa Clara, CA, USA: NVIDIA Corp, 2014.

[66] C. Yan, Y. Zhang, J. Xu, F. Dai, J. Zhang, Q. Dai and F. Wu, "Efficient parallel framework for HEVC motion estimation on many-core processors," *IEEE Trans. Circuits Syst. Video,* vol. 24, p. 2077–2089, 2014.

[67] A. Obukhov, "GPU-accelerated video encoding," in *Proceedings of the NVIDIA GPU Technology Conference*, San Jose, CA, USA, 20–23 September 2010.

[68] S. Radicke, S. Hahn, J. Grecos and Q. Wang, "Highly-parallel HVEC motion estimation with CUDA," in *Proceedings of the European Workshop on Visual Information Processing (EUVIP)*, Paris, France, 10–12 June 2013.

[69] F. Ezzahra, M. Chouchene, H. Bahri, R. Khemiri and M. Atri, "Parallel Full Search Algorithm For Motion Estimation On GPU," *Recent Adv. Electr. Electron. Eng.,* vol. 12, p. 317–323, 2019.

[70] A. Munshi, B. Gaster, T. Mattson and D. Ginsburg, OpenCL Programming Guide, London, UK: Pearson Education, 2011.

[71] F. Wang, D. Zhou and S. Goto, "OpenCL based high-quality HEVC motion estimation on GPU," in *Proceedings of the 2014 IEEE International Conference*, Paris, France, 27–30 October 2014.

[72] R. Khemiri, S. Bouaafia, A. Bahba, M. Nasr and F. Sayadi, Performance Analysis of OpenCL and CUDA Programming Models for the High-Efficiency Video Coding. In Digital Image Processing Applications, London, UK: IntechOpen, 2021.

[73] A. Gomez, J. Perea and M. Trujillo, "Parallel Integer Motion Estimation for High Efficiency Video Coding (HEVC) Using OpenCL," in *Proceedings of the Iberoamerican Congress on Pattern Recognition (CIARP)*, Lima, Peru, 8–11 November 2016.

[74] M. Harris, "Optimizing parallel reduction in CUDA," *NVIDIA Dev. Technol.,* vol. 2, p. 70, 2007.

[75] B. Catanzaro, OpenCL Optimization Case Study: Simple Reductions, Hyderabad, India, 2010: AMD Developer Center, 2010.

[76] F. Sayadi and M. Chouchene, "CUDA Memory Optimization Strategies for Motion Estimation," *IET Comput. Digit. Tech.,* vol. 13, p. 20–27, 2019.

[77] S. Atapattu, N. Liyanage, N. Menuka, I. Perera and A. Pasqual, "Real time all intra HEVC HD encoder on FPGA," in *Proceedings of the IEEE 27th International Conference on Application-specific Systems, Architectures and Processors (ASAP*, London, UK, 6–8 July 2016.

[78] P. Sjövall, A. Lemmetti, J. Vanne, S. Lahti and T. Hämäläinen, "High-Level Synthesis Implementation of an Embedded Real-Time HEVC Intra Encoder on FPGA for Media Applications," *ACM Trans. Des. Autom. Electron. Syst.,* vol. 27, no. 4, 2022.

[79] E. Ma, Z. Zhao and H. Qi, "Hardware-friendly Integer Motion Estimation with Weighted Search For AVS3," in *Proceedings of the 2023 IEEE International Symposium on Circuits and Systems (ISCAS)*, Monterey, CA, USA, 21–25 May 2023.

[80] H. Loukil and A. Mayet, "Hardware implementation and validation of the fast variable block size motion estimation architecture for HEVC Standard," *Multimed. Tools Appl.,* vol. 66, p. 701–710, 2023.

[81] Z. Pan, J. Lei, Y. Zhang and F. Wang, "Adaptive Fractional-Pixel Motion Estimation Skipped Algorithm for Efficient HEVC Motion Estimation," *ACM Trans Multimed. Comput.,* vol. 14, p. 1–19, 2018.

[82] J. Lee and S. Park, "Adaptive fractional motion and disparity estimation skipping in MV-HEVC," *J. Vis. Commun. Image Represent.,* vol. 79, 2021.

[83] W. Dai, O. Au, C. Pang, L. Sun, R. Zou and S. Li, "A novel fast two step sub-pixel motion estimation algorithm in HEVC," in *Proceedings of the 2012 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, Kyoto, Japan, 25–30 March 2012.

[84] G. Wei, X. Wang and T. Zhang, "Massively Accelerating VVC Intra Encoding Through Decoder-side Neural Quality Enhancement," in *Proceedings of the 4th Information Communication Technologies Conference (ICTC)*, Nanjing, China, 17–19 May 2023.

[85] D. Paper, Introduction to Tensor Processing Units. In State-of-the-Art Deep Learning Models in TensorFlow, Berkeley, CA, USA: Apress, 2021.

[86] "Introduction to Cloud TPU," [Online]. Available: https://cloud.google.com/tpu/docs/intro-to-tpu. [Accessed 6 July 2023].

[87] M. Elbtity, P. Chandarana, B. Reidy, J. Eshraghian and R. Zand, "APTPU: Approximate Computing Based Tensor Processing Unit," *IEEE Trans. Circuits Syst. I Regul. Pap.,* vol. 69, p. 5135–5146, 2022.

[88] N. Jouppi, C. Young, N. Patil and D. Patterson, "Motivation for and evaluation of the first tensor processing unit," *IEEE Micro.,* vol. 38, p. 10–19, 2018.

[89] H. Maich, V. Afonso, D. Franco, B. Zatt, M. Porto and L. Agostini, "A hardware-oriented concurrent TZ search algorithm for High-Efficiency Video Coding," *EURASIP J. Adv. Signal Process.,* vol. 78, 2017.

[90] H. Maich, V. Afonso, D. Franco, B. Zatt, M. Porto and L. Agostini, "High throughput hardware design for the HEVC Fractional Motion Estimation Interpolation Unit," in *Proceedings of the 2013 IEEE 20th International Conference on Electronics, Circuits, and Systems (ICECS)*, Abu Dhabi, United Arab Emirates, 8–11 December 2013.

[91] G. He, D. Zhou, Y. Li, Z. Chen, T. Zhang and S. Goto, "High-Throughput Power-Efficient VLSI Architecture of Fractional Motion Estimation for Ultra-HD HEVC Video Encoding," *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.,* vol. 23, p. 3138–3142, 2015.

[92] W. Penny, G. Correa, L. Agostini, D. Palomino, M. Porto, G. Nazar and B. Zatt, "Low-Power and Memory-Aware Approximate Hardware Architecture for Fractional Motion Estimation Interpolation on HEVC," in *Proceedings of the 2020 IEEE International Symposium on Circuits and Systems (ISCAS)*, Seville, Spain, 12–14 October 2020.

[93] H. Kibeya, F. Belghith, H. Loukil, M. Ayed and N. Masmoudi, "TZSearch pattern search improvement for HEVC motion estimation modules," in *Proceedings of the 2014 1st International Conference on Advanced Technologies for Signal and Image Processing (ATSIP)*, Sousse, Tunisia, 17–19 March 2014.

[94] F. Bossen, "Common Test Conditions and Software Reference Configurations," in *Proceedings of the 9th Meeting of the JCT-VC*, Geneva, Switzerland, 27 April–7 May 2012.

[95] L. Zhao, L. Zhang, S. Ma and D. Zhao, "Fast mode decision algorithm for intra prediction in HEVC," *Proceedings of visual communications and image processing (VCIP),* Vols. O-02-4, pp. 1-4, Nov 2011.

[96] F. Shajin, P. Rajesh and M. Raja, "An Efficient VLSI Architecture for Fast Motion Estimation Exploiting Zero Motion Prejudgment Technique and a New Quadrant-Based Search Algorithm in HEVC," *Circuits Syst. Signal Process,* vol. 41, p. 1751–1774, 2022.

[97] S. Gogoi and R. Peesapati, "Design and Implementation of an Efficient Multi-Pattern Motion Estimation Search Algorithm for HEVC/H.265," *IEEE Trans. Consum. Electron.,* vol. 67, p. 319–328, 2021.

[98] C. Yan, Y. Zhang, J. Xu, F. Dai, J. Zhang, Q. Dai and F. Wu, "Efficient parallel framework for HEVC motion estimation on many-core processors," *IEEE Trans Circuits Syst Video,* vol. 24, p. 2077–2089, 2014.

[99] G. Bjøntegaard, "Calculation of average PSNR Differences between RD-curves," in *Proceedings of the Tech. Rep. VCEG-M33, ITU-T Video Coding Experts Group (VCEG)*, Austin, TX, USA, 2–4 April 2001.

[100] P. Jövall, A. Lemmetti, J. Vanne, S. Lahti and T. Hämäläinen, "High-Level Synthesis Implementation of an Embedded Real-Time HEVC Intra Encoder on FPGA for Media Applications," *ACM Trans. Des. Autom. Electron. Syst.,* vol. 27, no. 4, 2022.

List of Acronyms and Abbreviations

A

adaptive offset: (SAO), 48
adaptive search range: (ASR), 68
Advanced Motion Vector Prediction: (AMVP), 43
Advanced Video Coding: (AVC), 15
AMP: (Asymmetric Motion Partitioning), 77
Application Specific Integrated Circuits: (ASICs), 68
artificial intelligence: (AI), 86
Audio Video Interleave: (AVI), 14

B

Block Matching Algorithm: (BMA), 43
British Broadcasting Corporation: (BBC), 14

C

CAVLC: (Context Adaptive Variable Length Coding), 32
CMYK: (Cyan, Magenta, Yellow, Key/Black), 21
Coding Tree Block: (CTB), 35
Coding Tree Units: (CTUs), 35
Coding Units: (CUs), 36
compression or decompression: (co/dec), 15
Context-based adaptive binary arithmetic coding: (CABAC), 32
Context-based adaptive binary arithmetic coding (CABAC), 32

D

DCT: (Discrete Cosine Transform), 27
Differential Pulse Code Modulation: (DPCM), 31

F

Field Programmable Gate Arrays: (FPGAs), 68
Four Phase Fast Search: (4PhaseFS), 92
fraction execution resolver: (FER), 19, 85
fraction motion estimation: (FME), 84

G

Graphic Processor Units: (GPU), 16
Group of Pictures: (GOP), 56

H

High-Efficiency Video Coding: (HEVC), 16
HSB: (Hue, Saturation, Brightness), 22
HSL: (Hue, Saturation, Lightness), 21
HSV: (Hue, Saturation, Value), 21

I

integer motion estimation: (IME), 84
integer motion estimation part: (IME), 18

J

Joint Video Experts Group: (JVET), 16

L

Largest Coding Unit: (LCU), 36
look-up table: (LUT), 32
Low Delay: (LD), 56

M

Mean Squared Error: (MSE), 44
motion estimation: (ME), 84
motion vector: (MV), 41
motion vector differences: (MVD), 43
motion vector prediction: (AMVP), 42
motion vector predictor: (MVP), 43
motion-compensated prediction: (MCP), 41
multilevel resolution sample area: (MLRME), 68
multiview HEVC: (MV-HEVC), 91

N

non-linear editing systems: (NLE), 15

O

OWF: (Overlapping Wave Front), 56

P

pattern structures: (PS), 90
Peak Signal to Noise Ratio: (PSNR), 47
PICS Animation Compiler: (PACo), 14
Prediction Block: (PB), 37
preliminary Fast Test Decision Point: (pFTDP), 18

R

rate distortion: (RD), 84
RGB: (Red, Green, Blue), 21

S

special function units: (SFUs), 76
Sum of Absolute Difference: (SAD), 44

T

tensor processing units: (TPUs), 86
test zone search: (TZS), 91
Tile Parallelism: (TP), 17
Transform Blocks: (TB), 37

V

Variable Length Encoding: (UVLC), 31
Versatile Video Coding: (VVC), 16

W

Wavefront Parallel Processing: (WPP), 17
Wavefront per Tile Parallelism: (WTP), 17

Y

YCbCr: (Luma, Color Blue diff relative to Green, Color Red Diff relative to Green), 21
YUV: (Luma, Chroma), 21

Z

zero-movement prejudgment technique: (ZMP), 90