COMPLEXITY REDUCTION IN H.264 ENCODER USING OPEN MULTIPROCESSING

by

TEJAS PRAVIN SATHE

Presented to the Faculty of the Graduate School of

The University of Texas at Arlington in Partial Fulfillment

of the Requirements

for the Degree of

MASTER OF SCIENCE IN ELECTRICAL ENGINEERING

THE UNIVERSITY OF TEXAS AT ARLINGTON

May 2012

ACKNOWLEDGEMENTS

It would not have been possible to complete this thesis without the guidance and the help of several individuals who in one way or another contributed and extended their valuable assistance

First and foremost, my utmost gratitude to my thesis advisor, Dr. K. R. Rao, whose sincerity and encouragement I will never forget. I really appreciate his helpful nature and care he takes for the students. I feel really proud to be Dr. Rao's student and a member of Multimedia Processing Lab.

I would like to thank Dr. Saibun Tjuatja and Dr. Ioannis D. Schizas for their valuable time and taking interest to review my thesis work. I am also very grateful to Dr. Dongil Han for his technical guidance and financial support.

I want to thank Dr. Roger Walker for offering the course 'Real Time Data Acquisition and Control', in which he taught parallel programming using OpenMP, which this thesis is based on.

This thesis would never have been completed without the encouragement and devotion of my family and friends. I express my sincere gratitude towards them.

April 20, 2012

ABSTRACT

COMPLEXITY REDUCTION IN H.264 ENCODER USING

OPEN MULTIPROCESSING

Tejas Sathe, (M.S.)

The University of Texas at Arlington, 2012

Supervising Professor: K. R. Rao

H.264 video standard developed by Joint Video Team has proven dramatic improvements in bit-rate efficiency, compression ratio, video quality and error resilience. But, all this is achieved at the expense of more than four times of the computational complexity due to various new features including quarter-pixel motion estimation with variable block sizes and multiple reference frames, adaptive directional intra-prediction, integer transformation based on discrete cosine transform, alternative entropy coding mode, Context-based Adaptive Variable Length Coding (CAVLC) or Context-Based Adaptive Binary Arithmetic Coding (CABAC), in loop de-blocking filter etc.

This thesis aims at reducing the encoding time for the video sequences while maintaining the same quality and compression efficiency using parallel processing approach. Entire video sequence is divided into four groups having one I frame each. In this thesis, the original JM software code, written in serial manner is enhanced in such a way that the encoding of all individual parts of the original video is implemented in parallel using the threads that are managed by Open Multiprocessing runtime system which shows more than 60% encoding time reduction.

JM 18.0 reference software is used in this thesis for implementing H.264 video encoder. The reference software along with manual is available at http://iphome.hhi.de/suehring/tml. The software manual includes information about the H.264 encoder and decoder input parameters, syntax, compilation issues, and additional information with regards to the best usage and configuration of the software.

TABLE OF CONTENTS

LIST OF ILLUSTRATIONS

LIST OF TABLES

LIST OF ACRONYMS

AVC	Advanced Video Coding
B slice	Bi-directionally predictive slice
CABAC	Context-Based Adaptive Binary Arithmetic Coding
CAVLC	Context Adaptive Variable Length Coding
CD-ROM	Compact Disc- Read Only Memory
CIF	Common Intermediate Format
CMP	Chip Multi Processing
CPU	Central Processing Unit
CUDA	Compute Unified Device Architecture
DCT	Discrete Cosine Transform
DVD	Digital Video Disk
FIR	Finite Impulse Response
FMO	Flexible Macroblock Order
GPU	Graphical Processing Unit
GPGPU	General Purpose Computation on Graphical Processing Unit
HD	High Definition
I slice	Intra slice
ISO	International Organization for standardization
ITU-T	International Telecommunication Union- Transmission standardization sector
JVT	Joint Video Team
JM	Joint Model
MPEG	Moving Picture Experts Group
MV	Motion Vector

NAL – Network Abstraction Layer

NTSC – National Television System Committee

P slice – Predictive slice

PSNR – Peak Signal to Noise Ratio

PCM – Pulse Code Modulation

QCIF – Quarter Common Intermediate Format

QP – Quantization Parameter

RAM – Random Access Memory

RDO – Rate Distortion Optimization

RISC – Reduced Instruction Set Computing

RS – Redundant Slices

SAD – Sum of Absolute Differences

SATD – Sum of Absolute Transformed Differences

SSIM – Structural Similarity Index Metric

TV – Television

URQ – Uniform Reconstruction Quantizers

VCEG – Video Coding Experts Group

VCL – Video Coding Layer

VCE – Video Codec Engine

VLE – Variable Length Encoding

CHAPTER 1

INTRODUCTION

1.1 Introduction to video coding standards

Over past forty years, efficient digital representation of image and video signals has been the subject of considerable research. The rapid growth of digital video coding technology has resulted in increased commercial interest in video communications. This arose a need for international image and video coding standards, which is the basis of large markets for video communication equipment, digital video broadcasting.

Research in signal processing and image compression, VLSI technology, visual communications, digital video coding technology and growing availability of digital transmission links, is dramatically becoming more feasible. Interoperability of implementations from different vendors enables the consumer to access video from a wider range of services and VLSI implementations of coding algorithms conforming to international standards can be manufactured at considerably reduced costs. Modern data compression techniques today offer the possibility to store or transmit the vast amount of data necessary to represent digital images and video in an efficient and robust way. Digital video technology is enabling and generating ever new applications with a broadening range of requirements regarding basic video characteristics such as spatiotemporal resolution, chroma format, and sample accuracy. Various application areas, now a days, range from videoconferencing over mobile TV and broadcasting of standard and high-definition TV content up to very high quality applications such as professional digital video recording or digital cinema/large-screen digital imagery.

Prior video coding standards such as MPEG2/H.262 [1], H.263 [2], and MPEG4 Part 2 [3] are already established in parts of those application domains. But with the proliferation of digital video into new application areas such as mobile TV or high-definition TV broadcasting, the requirements for efficient representation of video have increased up to operation points where previously standardized video coding technology can hardly keep pace. Furthermore, more cost-efficient solutions in terms of bit rate vs. end-to-end reproduction quality are increasingly sought in traditional application areas of digital video as well. If your work contains more than four chapters, add additional template chapters to your template now before continuing. To add additional chapters to those that come with your template you will need to use the text selection and copy operations.

A diversity of products has been developed targeted for a wide range of emerging applications, such as video on demand, digital TV/HDTV broadcasting, and multimedia image/video database services.

1.2 The role and standardization procedure of video coding standards [11], [24]

Over last two decades, various video coding techniques have been proposed. Although there have been a wide range of innovations in the domain of video encoding, commercial video coding applications tend to use a limited number of standardized techniques for video compression. In fact, standardized video coding formats always have upper hand, as far as potential benefits are concerned, compared with non-standard, proprietary formats.

Standards attempt to simplify the inter-operability between encoders and decoders from different manufacturers. In addition, standards make it possible to build platforms that incorporate video, in which many different applications such as video codecs, audio codecs, transport protocols, security and rights management, interact in well-defined and consistent ways.

Almost every video coding technique used commercially is patented. This fact always poses a risk in which some other video codec implementation may infringe patent(s). The techniques and algorithms required to implement a standard are well-defined and the cost of licensing patents that cover these techniques, i.e. licensing the right to use the technology exemplified in the patents, can be clearly defined.

The main steps towards the finalization of a standard can roughly be described as shown in Figure 1.1, although there are slight differences in standardization procedures among the different standardization bodies.

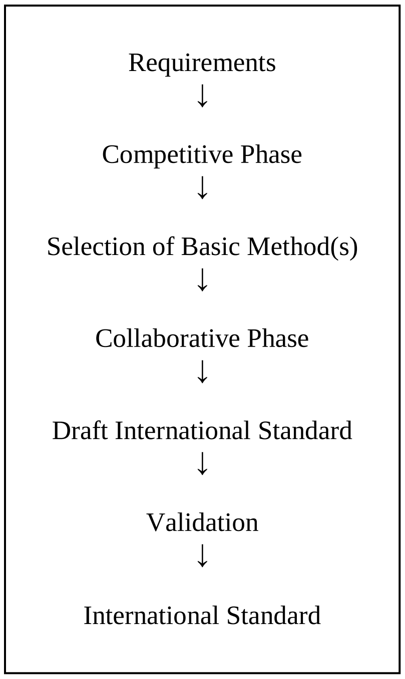


Figure 1.1 Steps in international standardization

Starting with the requirements phase, the requirements for a particular application or for a field of applications are identified. Different algorithms are developed next by various laboratories and then compared. A single basic technique, as a result of this comparison, is identified which is then sophisticated in a joint effort during the next phase, called collaborative phase. At the end of this phase a draft standard is issued, which has to be validated by compliance testing based on computer simulations or hardware testing. Once successful validation and eventual refinements are done the final standard is issued.

1.3 Importance of H.264 Advanced Video Coding [11]

The H.264 video coding standard was jointly published by the International Telecommunication Union (ITU) and the International Standards Organization (ISO). The standard is known by several other names like ‘MPEG-4 Part 10’ and ‘Advanced Video Coding’. Over 550 pages long and filled with highly technical definitions and descriptions, the standard’s document was developed by a team consisting of hundreds of video compression experts from the Joint Video Team (JVT). JVT is a collaboration of the Moving Picture Experts Group (MPEG) and the Video Coding Experts Group (VCEG).

H.264/AVC standard has enormous significance in the broadcast, internet, consumer electronics, mobile and security industries, amongst others. It describes and defines a method of coding video that can give better performance than any of the preceding standards.

The H.264 video codec attempts to compress the video so that, the compressed video clip takes up less bandwidth as far as transmission is concerned and takes up less memory space compared to older codecs, as far as storage on secondary memory is concerned.

A combination of market expansion, technology advances and increased user expectation is driving demand for better, higher quality digital video. Various examples in daily life include High Definition (HD) content delivery, uploading and downloading videos using websites such as YouTube, recording and sharing videos using mobile handsets, internet video calls and so on. In each of these examples, the most important expectation from a codec is better video compression for delivering more, higher-quality video in a frugal way. It is the H.264 standard that makes it possible to transmit HD content over a broadcast channel having limited-capacity, to record hours of video on a Flash memory card and to deliver massive numbers of video streams over an already busy internet.

1.4 H.264 CODEC standard [1, 2]

The aim of H.264 is high-quality coding of video contents at low bit-rates, having significant improvements in coding efficiency and error robustness in comparison with previous video coding standards [1]. There are a number of new features and capabilities that have been added in H.264 to improve its coding performance. As a result, the H.264 encoding process is more computationally intensive than existing standards.

Some important applications of H.264 codec include:

1. Broadcast over cable, satellite, cable modem, DSL, terrestrial, etc.
2. Interactive or serial storage on optical and magnetic devices, DVD, etc.
3. Conversational services over ISDN, Ethernet, LAN, DSL, wireless and mobile networks, modems, etc. or mixtures of these.
4. Video-on-demand or multimedia streaming services over ISDN, cable modem, DSL, LAN, wireless networks, etc.
5. Multimedia messaging services (MMS) over ISDN, DSL, Ethernet, LAN, wireless and mobile networks, etc.

1.5 Key features of H.264 video codec

Following are some highlighted features of H.264 video codec:

1. Variable block-size motion compensation with small block sizes
2. Quarter-sample-accurate motion compensation
3. Multiple reference picture motion compensation
4. Weighted prediction
5. Improved “skipped” and “direct” motion inference
6. Directional spatial prediction for intra coding
7. In-the-loop deblocking filtering
8. Context-adaptive entropy coding
9. Flexible slice size
10. Flexible macroblock ordering (FMO)

1.6 H.264 Profiles [2]

The JVT involved in defining H.264 focused on creating a simple and clean solution, limiting options and features to a minimum. An important aspect of the standard, as with other video standards, is providing the capabilities in profiles (sets of algorithmic features) and levels (performance classes) that optimally support popular productions and common formats. Following sets of capabilities, known as profiles, as shown in Figure1.2, are defined in H.264/AVC standard. These profiles target specific classes of applications.

1.6.1 Baseline Profile

1. Flexible macroblock order: macroblocks may not necessarily be in the raster scan order. The map assigns macroblocks to a slice group.
2. Arbitrary slice order: the macroblock address of the first macroblock of a slice of a picture may be smaller than the macroblock address of the first macroblock of some other preceding slice of the same coded picture.
3. Redundant slice: this slice belongs to the redundant coded data obtained by same or different coding rate, in comparison with previous coded data of same slice.

1.6.2 Main Profile

1. B slice (Bi-directionally predictive-coded slice): the coded slice by using inter-prediction from previously decoded reference pictures, using at most two motion vectors and reference indices to predict the sample values of each block.
2. Weighted prediction: scaling operation by applying a weighting factor to the samples of motion-compensated prediction data in P or B slice.
3. CABAC (Context-based Adaptive Binary Arithmetic Coding) for entropy coding.

1.6.3 Extended Profile

1. Includes all parts of Baseline Profile: flexible macroblock order, arbitrary slice order and redundant slice
2. SP slice: the specially coded slice for efficient switching between video streams, similar to

coding of a P slice.

3. SI slice: the switched slice, similar to coding of an I slice.
4. Data partition: the coded data is placed in separate data partitions, each partition can be placed in different layer unit.
5. B slice
6. Weighted prediction

1.6.4 High Profiles

1. Includes all parts of Main Profile: B slice, weighted prediction and CABAC
2. Adaptive transform block size: 4x4 or 8x8 integer transform for luma samples
3. Quantization scaling matrices: different scaling according to specific frequency associated with the transform coefficients in the quantization process to optimize the subjective quality

This thesis implements the video algorithm using baseline profile.

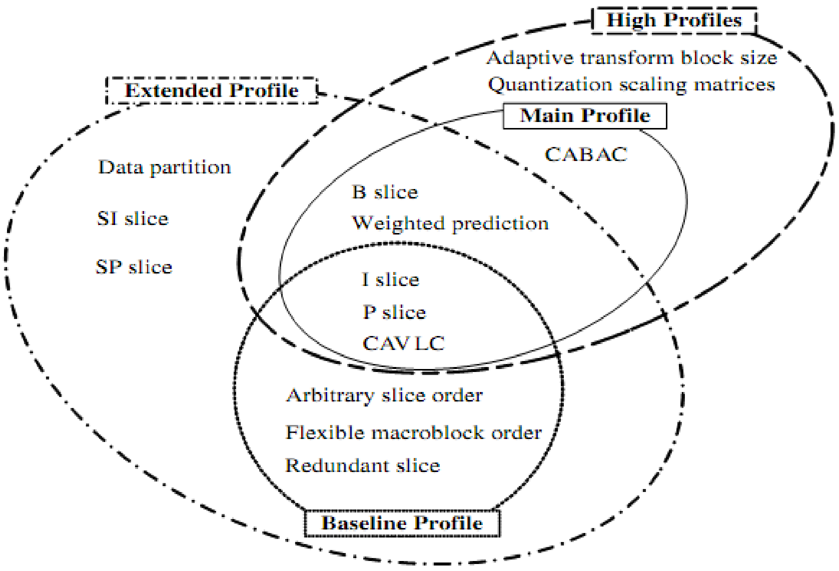


Figure 1.2 H.264 Profiles [1]

1.6.5 Applications of H.264 profiles [1]

Table 1.1 H.264 Profiles and Applications

Application	**Requirements**	**H.264 Profiles**
Broadcast television	Coding efficiency, reliability (over a controlled distribution channel), interlace, low-complexity decoder	Main
Streaming video	Coding efficiency, reliability (over a uncontrolled packet-based network channel), scalability	Extended
Video storage and Playback	Coding efficiency, interlace, low-complexity encoder and decoder	Main
Videoconferencing	Coding efficiency, reliability, low latency, low-complexity encoder and decoder	Baseline
Mobile video	Coding efficiency, reliability, low latency, low-complexity encoder and decoder, low power consumption	Baseline
Studio distribution	Lossless or near-lossless, interlace, efficient transcoding	Main and High Profiles

1.7 Summary

This chapter explains video coding standardization, especially, the H.264/AVC standard along with key features in it.

Next chapter illustrates working of the H.264 codec, which gives a gist of additional complexity involved in the standard. The chapter finally concludes with need for time complexity reduction in H.264.

CHAPTER 2

NEED FOR TIME COMPLEXITY REDUCTION IN H.264

2.1 Key concepts involved in a video codec [11]

Basically, video coding is the process of compressing and decompressing a digital video signal. Digital video is nothing but, a representation of a natural or real-world visual scene, sampled spatially and temporally. A frame is produced by sampling a scene temporally at a point in time. The frame represents the complete visual scene at that point in time, or a field, which typically consists of odd- or even-numbered lines of spatial samples. Sampling is repeated at certain intervals such as 1/25 or 1/30 second to finally produce a moving video signal. Three components or sets of samples are typically required to represent a scene in color. In order to determine the performance of a visual communication system, which is a difficult and inexact process, the accuracy of a reproduction of a visual scene has to be measured.

2.1.1 CODEC and types of compression [11]

Any compression of audio, video and data involves a complementary pair of systems, an encoder, which acts as a compressor and a decoder which acts as a decompressor. The role of compression is to compact the data into a smaller number of bits. The encoder converts the source data into a compressed form occupying a reduced number of bits, prior to transmission or storage. On the other hand, the decoder converts the compressed form back into a representation of the original video data. The encoder/decoder pair is nothing but a CODEC (enCOder/ DECoder).

There are, typically two kinds of compression types defined, lossless and lossy compression. In case of lossless compression, the reconstructed data at the output of the decoder is a perfect copy of the original data. But, lossless compression of image and video information gives only a moderate amount of compression.

Lossy compression, on the other hand, is necessary to achieve higher compression. Lossy compression helps achieve much higher compression ratios, but, at the expense of a loss of visual quality. Most of the video coding methods exploit both temporal and spatial redundancies to achieve compression as shown in Figure 2.1.

In the spatial domain, there exists high correlation between pixels (samples) that are close to each other, i.e. the values of neighboring samples are often very similar.

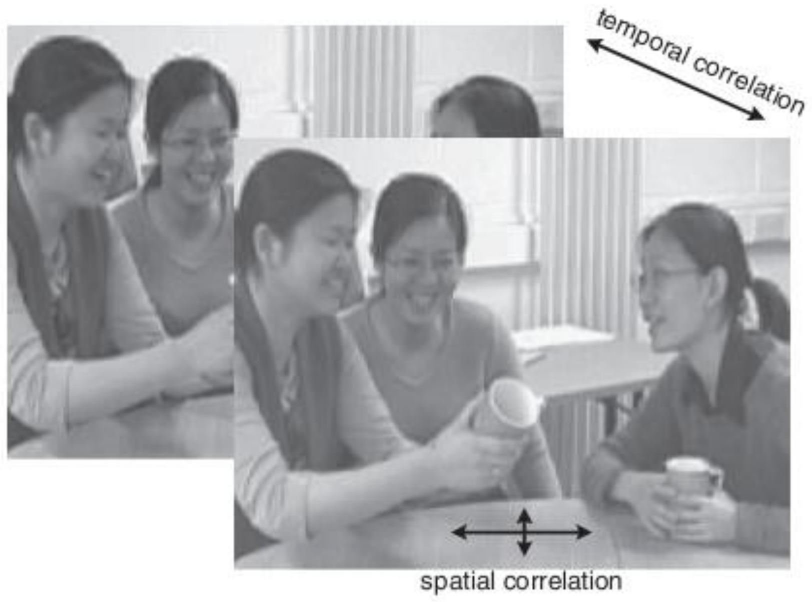


Figure 2.1 Spatial and temporal correlation in a video [11]

There is a high correlation or similarity, as shown in Figure 2.1, between temporally adjacent frames, i.e. successive frames in time order, especially, if the temporal sampling rate or frame rate is high.

2.1.2 Working of H.264 video codec [1], [11]

The codec, consisting of an encoder and a decoder is depicted in Figures 2.2 and 2.4. The encoder may select between intra and inter coding for block-shaped regions of each picture to exploit either spatial or temporal redundancy. Intra-coding uses various spatial prediction modes to reduce spatial redundancy in the source signal for a single picture. Inter - coding, which could be predictive or bi-predictive is more efficient using inter-prediction of each block of sample values from some previously decoded pictures. In order to reduce temporal redundancy among different pictures, inter-coding uses motion vectors for block-based inter-prediction. Prediction is obtained from deblocking filtered signal of previous reconstructed pictures.

The deblocking filter is to reduce the blocking artifacts at the block boundaries. Motion vectors and intra-prediction modes may be specified for a variety of block sizes in the picture. The prediction residual is then further compressed using a transform to remove spatial correlation in the block before it is quantized.

Finally, the motion vectors or intra-prediction modes are combined with the quantized transform coefficient information and dencoded using entropy code such as context-adaptive variable length codes (CAVLC) or context adaptive binary arithmetic coding (CABAC).

2.1.3 Encoder [2], [7], [11]

The H.264 video encoder block diagram is shown in Figure 2.2. The working of the encoder can be roughly divided into two parts: forward and reverse paths.

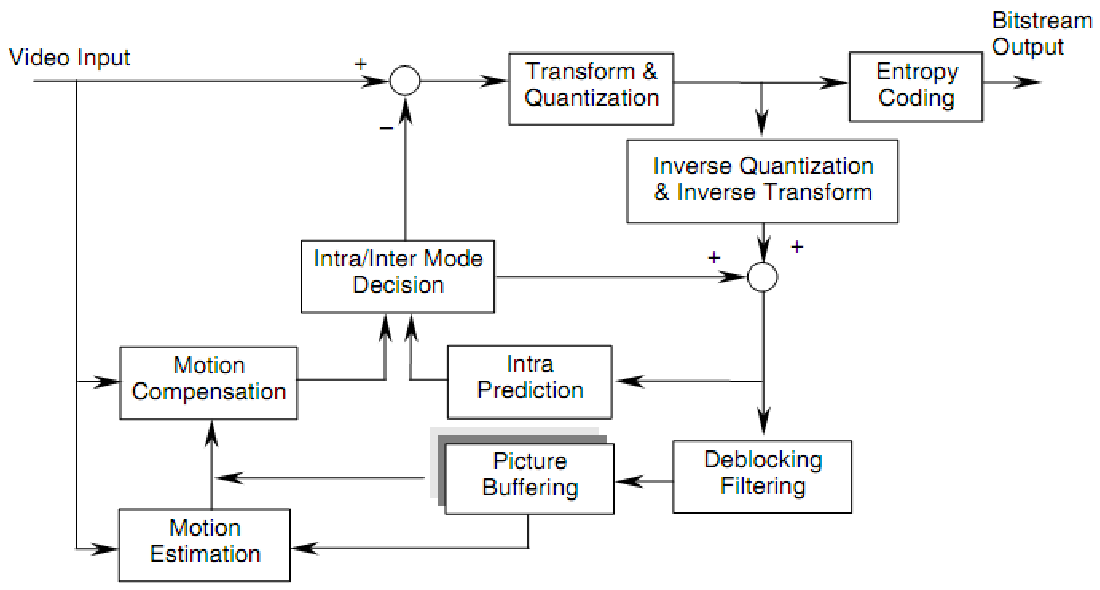


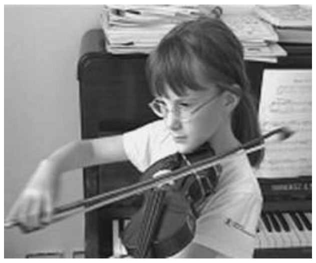
Figure 2.2 Block diagram of H.264 encoder [1]

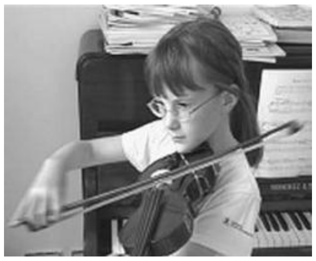
2.1.3.1 Encoder (Forward Path)

An H.264 video encoder carries out prediction, transform and encoding process to produce a compressed H.264 bit stream. A frame to be encoded is processed by an H.264 compatible video encoder. In addition to coding and sending the frame as a part of the coded bit stream, the encoder reconstructs the frame i.e. imitates the decoder and the reconstructed frame is stored in a coded picture buffer, and used during the encoding of further frames.

An input frame is presented for encoding as shown in Figure 2.2. The frame is processed in units of a macroblock corresponding to 16x16 pixels in the original image. Each macroblock is encoded in intra or inter mode. Based on a reconstructed frame, a predicted macroblock is formed. In intra mode, predicted macroblock is formed from samples in the current frame that have been previously encoded, decoded and reconstructed. The unfiltered samples are used to form P. In inter mode, P is formed by inter or motion-compensated prediction from one or more reference frame(s). The prediction for each macroblock may be formed from one or more past or future frames (in time order) that have already been encoded and reconstructed.

In the encoder, the prediction macroblock P is subtracted from the current macroblock. This produces a residual macroblock, also known as difference macroblock. Figure 2.3 shows how a difference frame looks like. Using a block transform, the difference macroblock is transformed and quantized to give a set of quantized transform coefficients. These coefficients are rearranged and encoded using entropy encoder. The entropy encoded coefficients, and the other information such as the macroblock prediction mode, quantizer step size, motion vector information etc. required to decode the macroblock form the compressed bitstream. This is passed to network abstraction layer (NAL) for transmission or storage.


a)


b)

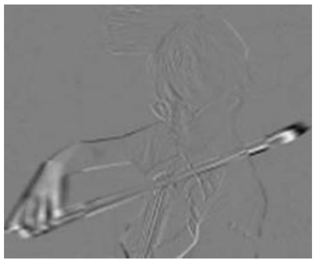

c)

Figure 2.3 Difference between adjacent frames. a) Frame 1, b) Frame 2, c) Difference frame

2.1.3.2 Encoder (Reconstruction path)

In the reconstruction path, quantized macroblock coefficients are dequantized and are re-scaled and inverse transformed to produce a difference macroblock. This is not identical to the original difference macroblock, since quantization is a lossy process. The predicted macroblock P is added to the difference macroblock to create a reconstructed macroblock a distorted version of the original macroblock. To reduce the effects of blocking distortion, a de-blocking filter is applied and from a series of macroblocks, reconstructed reference frame is created.

2.1.4 Decoder [1], [2], [11]

The decoder block diagram of H.264 is shown in Figure 2.4. The decoder carries out the complementary process of decoding, inverse transform and reconstruction to produce a decoded video sequence.

The decoder receives a compressed bitstream from the NAL. The data elements are entropy decoded and rearranged to produce a set of quantized coefficients. These are rescaled and inverse transformed to give a difference macroblock. Using the other information such as the macroblock prediction mode, quantizer step size, motion vector information etc. decoded from the bit stream, the decoder creates a prediction macroblock P, identical to the original prediction P formed in the encoder. P is added to the difference macroblock and this result is given to the deblocking filter to create the decoded macroblock.

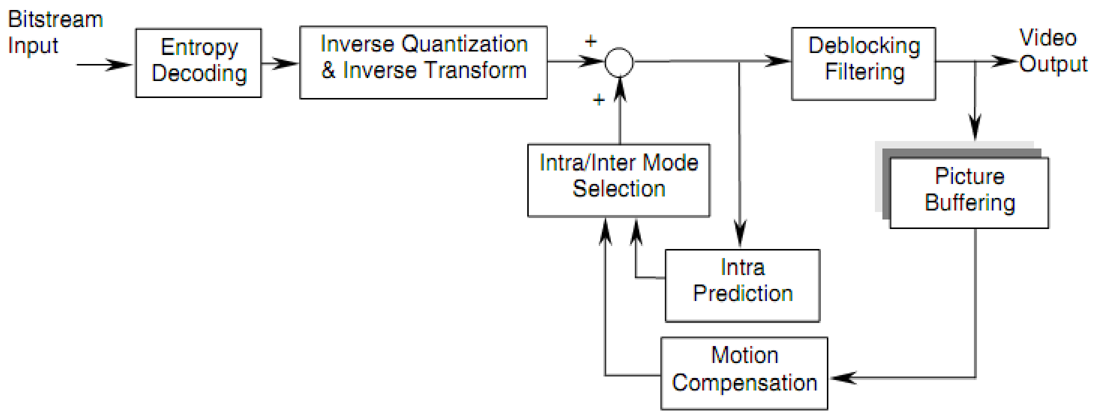


Figure 2.4 Decoder block diagram of H.264 [1].

The reconstruction path in the encoder ensures that both encoder and decoder use identical reference frames to create the prediction P. If this is not the case, then the predictions P in encoder and decoder will not be identical, leading to an increasing error or drift between the encoder and decoder.

2.1.5 Intra Prediction [2], [10], [11]

Adaptive intra directional prediction modes for (4x4) and (16x16) blocks are shown in Figures. 2.5 and 2.6.

In order to exploit the spatial redundancy between adjacent macroblocks within a frame, technique used in H.264 encoder is intra-prediction. From adjacent edges of neighboring macroblocks that are decoded before the current macroblock, it predicts the pixel values as linear interpolation of pixels. Directional in nature, these interpolations are with multiple modes. Each mode implies a spatial direction of prediction. There are 9 prediction modes defined for a 4x4 block and 4 prediction modes defined for a 16x16 block.

2.1.5.1 4x4 luma prediction modes [4], [14]

Figure 2.5 shows a luminance macroblock and a 4x4 luma block that is required to be predicted and various modes of prediction are shown. The samples above and to the left have previously been encoded and reconstructed and are available both in the encoder and decoder to form a prediction reference. The prediction block is calculated based on the samples labeled A-M in Figure 2.5. However, in some cases, not all of the samples A-M are available within the current slice: in order to preserve independent decoding of slices, only samples within the current slice are available for prediction. DC prediction (mode 0) is modified depending on which samples out of A-M are available; the other modes (1-8) may only be used if all of the required prediction samples are available (except that, if E, F, G and H are not available, their value is copied from sample D).

Direction of prediction in each mode is indicated by the arrows in Figure 2.5. For modes 3-8, the predicted samples are formed from a weighted average of the prediction samples A-M. The encoder may select the prediction mode for each block that minimizes the residual between P and the block to be encoded.

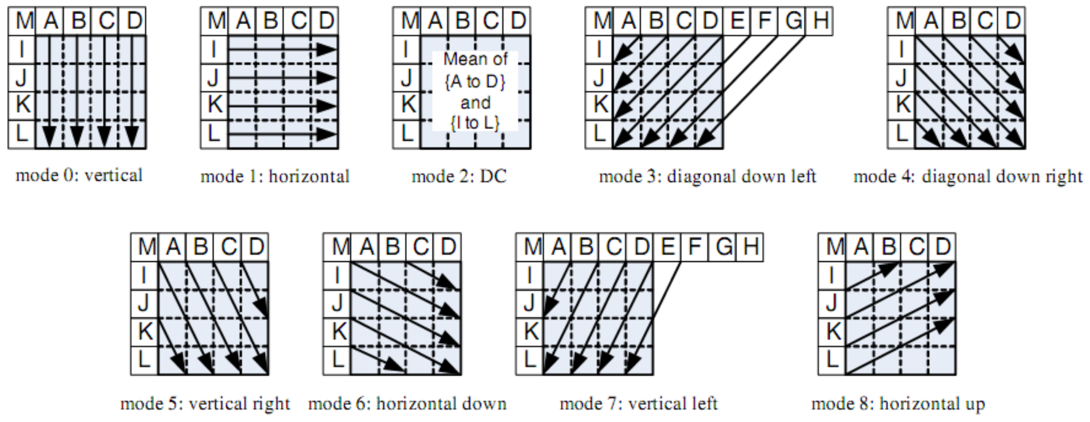


Figure 2.5 Intra 4x4 prediction modes and prediction directions [11]

The 9 prediction modes (0-8) are calculated for the 4x4 block shown in Figure 2.5. The prediction block is then created by each of the predictions. The Sum of Absolute Errors (SAE) for each prediction is calculated for each mode, which indicates the magnitude of the prediction error. The mode that finally gives the smallest SAE is finally selected for encoding.

2.1.5.2 16x16 luma prediction modes [14]

The entire 16x16 luma component of a macroblock can be predicted, as an alternative to the nine 4x4 luma modes. For 16x16 macroblock, four modes are available, shown in Figure 2.6:

1. Mode 0 (vertical): extrapolation from upper samples (H).
2. Mode 1 (horizontal): extrapolation from left samples (V).
3. Mode 2 (DC): mean of upper and left-hand samples (H+V).
4. Mode 4 (Plane): a linear “plane” function is fitted to the upper and left-hand samples H and V. This works well in areas of smoothly-varying luminance.

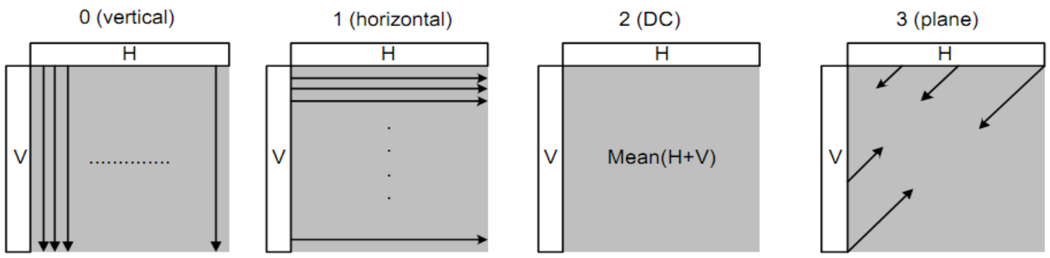


Figure 2.6 H.264 Intra 16x16 prediction modes [11] (all predicted from pixels H and V)

2.1.6 Inter Prediction [1], [11], [14]

To exploit temporal redundancy between the frames, inter prediction is performed in H.264 encoder. In this process, a block of luma and chroma samples is predicted from a picture that has previously been coded and transmitted, a reference picture. Selecting a prediction region, generating a prediction block and subtracting this from the original block of samples to form a residual are various steps involved. The residual is then transformed, coded and transmitted. The block size can range from 16x16 to 4x4 luma and corresponding chroma samples.

The luminance component of each macroblock (16x16 samples) maybe split up in 4 ways, 16x16, 16x8, 8x16 or 8x8 as shown in Figure 2.7. Each of the sub-divided regions is a macroblock partition. If the 8x8 mode is chosen, each of the four 8x8 macroblock partitions within the macroblock may be split in a further 4 ways, 8x8, 8x4, 4x8 or 4x4 (known as macroblock sub-partitions) as shown in Figure 2.8. These partitions and sub-partitions give rise to a large number of possible combinations within each macroblock. In general, a large partition size is appropriate for homogeneous areas of the frame and a small partition size may be beneficial for detailed areas.

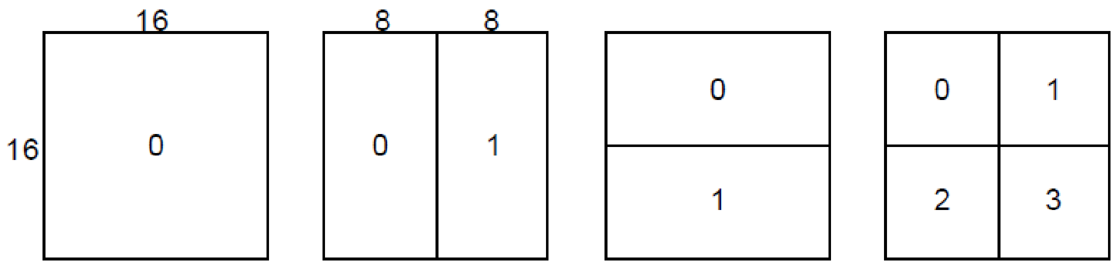


Figure 2.7 Macroblock partitions: 16x16, 8x16, 16x8, 8x8 [14]

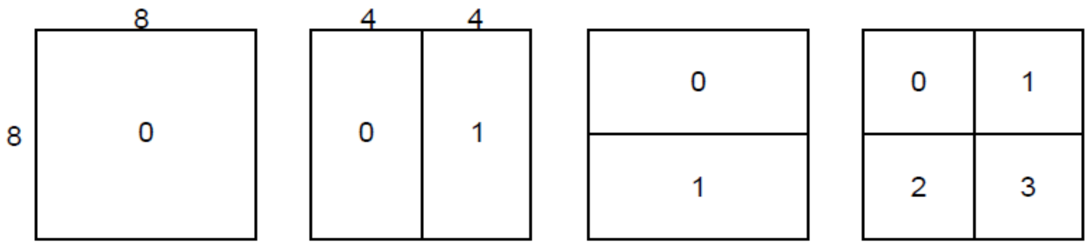


Figure 2.8 Macroblock sub partitions: 8x8, 4x8, 8x4, 4x4 [14]

Each macroblock partition or sub-partition requires a separate motion vector. Each motion vector must be coded and transmitted; in addition, the choice of partition(s) must be encoded in the compressed bitstream. If a large partition size (e.g. 16x16, 16x8, 8x16) is chosen, less number of bits are required to indicate the choice of motion vector(s) and the type of partition. The motion compensated residual may contain a significant amount of energy in frame areas with high detail. Choosing a small partition size (e.g. 8x4, 4x4, etc.) may give a lower-energy residual after motion compensation but requires a larger number of bits to signal the motion vectors and choice of partition(s). The choice of partition size therefore has a significant impact on compression performance. In general, a large partition size is appropriate for homogeneous areas of the frame and a small partition size may be beneficial for detailed areas. Figure 2.9 shows a residual frame (without motion compensation). The H.264 encoder chooses the best partition size for each part of the frame in such a way that the partition size minimizes the coded residual and motion vectors. If there is little difference between the frames (residual appears grey), a 16x16 partition is chosen; whereas in areas of detailed motion (residual appears black or white), smaller partitions are more efficient.

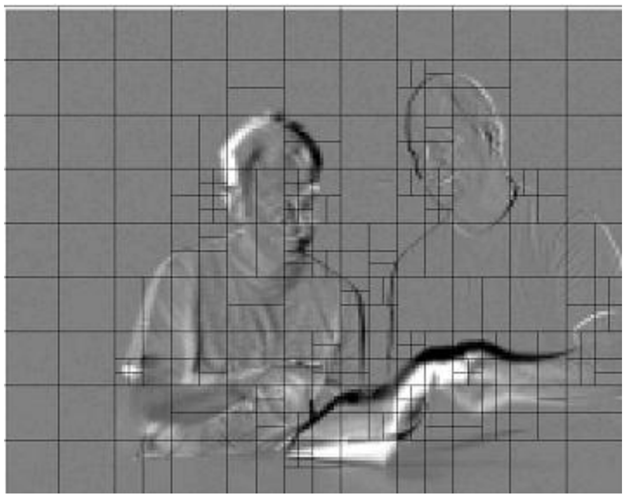


Figure 2.9 Optimum choice of partitions for residual without motion compensation [11]

AVC also defines sub-pel motion compensation, which can provide significantly better compression performance than integer-pel compensation, at the expense of increased complexity. The codec defines quarter- pel accuracy which, definitely, performs better than half-pel accuracy.

Three sampling patterns [11] for Y, Cr and Cb supported by H.264/AVC are shown in Figure 2.10. 4:4:4 sampling means that the three components (Y:Cr:Cb) have the same resolution and hence a sample of each component exists at every pixel position. 4:2:2 sampling, sometimes referred to as YUY2, consists of chrominance components having the same vertical resolution as the luma but half the horizontal resolution. In the popular 4:2:0 sampling format (‘YV12’), Cr and Cb each have half the horizontal and vertical resolution of Y. 4:2:0 sampling is widely used for consumer applications such as video conferencing, digital television and digital versatile disk (DVD) storage.

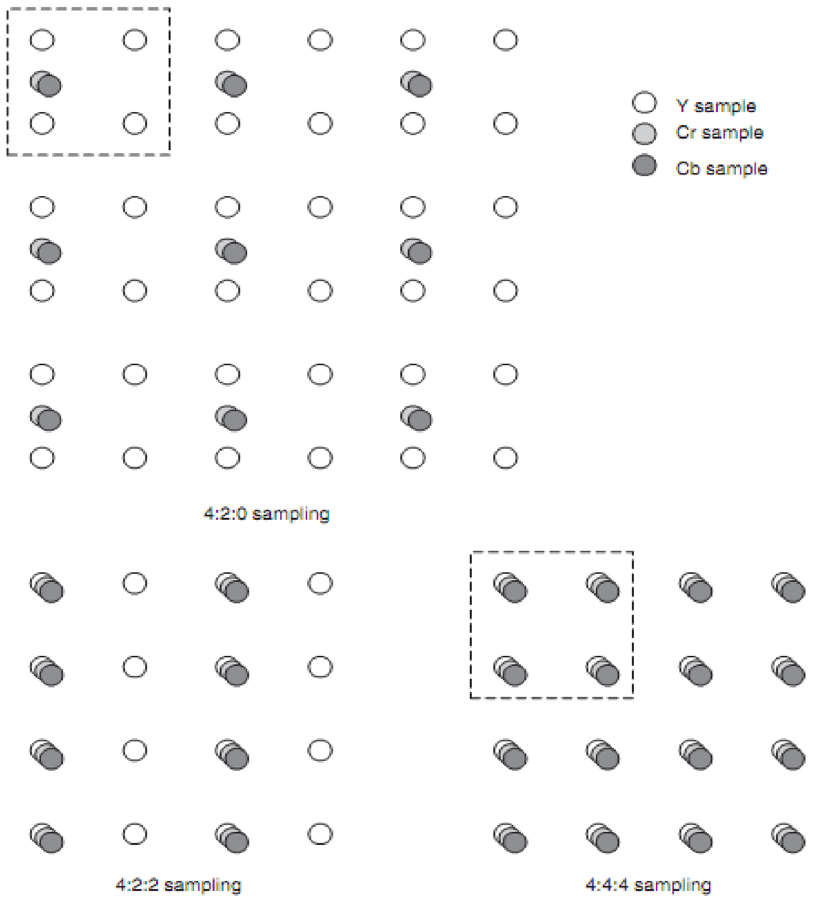


Figure 2.10 4:2:0, 4:2:2 and 4:4:4 sampling patterns (progressive) [11]

One of the most important factors behind the increased coding efficiency at high bitrates and high video resolutions is sub-pel accuracy. In the luma component, the sub-pel samples at half-pel positions are generated first and are interpolated from neighboring integer-pel samples using a 6-tap FIR filter with weights (1, -5, 20, 20, -5, 1) / 32. Once all the half-pel samples are available, each quarter-pel sample is produced using bilinear interpolation between neighboring half- or integer-pel samples, as shown in Figure 2.11. For 4:2:0 video source sampling, 1/8 pel samples are required in the chroma components (corresponding to 1/ 4 pel samples in the luma). These samples are interpolated (linear interpolation) between integer-pel chroma samples. Sub-pel motion vectors are encoded differentially with respect to predicted values formed from nearby encoded motion vectors.

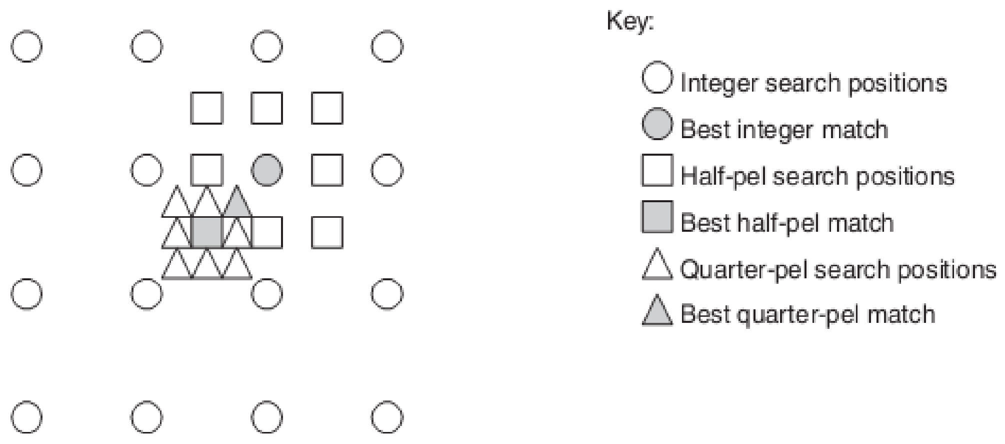


Figure 2.11 Integer, half-pixel and quarter-pixel motion estimation [11]

Reference pictures that are used for inter-prediction of a sample block are stored in the picture buffer. With respect to the current picture, the pictures before and after the current picture, in the display order are stored into the picture buffer.

2.1.7 Rate Distortion Optimization [13]

The H.264/AVC intra-prediction is conducted for all types of blocks such as 4x4 luma blocks, 16x16 luma blocks, and 8x8 chroma blocks. The residual between the current block and its prediction is then transformed, quantized, and entropy coded.

To obtain the best mode among these modes, the H.264/AVC encoder performs the rate-distortion optimization (RDO) technique for each macro block.

1. Set macro block parameters : QP (quantization parameter) and Lagrangian multiplier λ
2. Calculate : *λMODE = 0.85·2(QP-12)/3*
3. Then calculate cost, which determines the best mode

Cost = D + λMODE. R   D – Distortion   R - Bit rate with given QP

Distortion (D) is obtained by SSD (sum of squared differences) between the original macro block and its reconstructed block.

Bit rate(R) includes the number of bits for the mode information and transforms coefficients for macro block. Considering the RDO procedure for intra mode selection in H.264/AVC, the number of mode combinations in one macro block is N8x (16xN4 + N16)

N8 – number of modes of an 8x8 chroma block, N4 – number of modes of an 4x4 luma block

N16 – number of modes of a 16x16 luma block

2.1.8 Transform, Scaling, and Quantization [3]

H.264/MPEG4-AVC also uses transform coding of the prediction residual. H.264/MPEG4-AVC specifies a set of integer transforms of different block sizes. An additional M × N transform stage is further applied to all resulting DC coefficients in the case of the luma component of a macroblock that is coded using the 16 × 16 intra-coding type (with N = M =4) as well as in the case of both chroma components. For these additional transform stages, separable combinations of the four-tap Hadamard transform and two-tap Haar/Hadamard transform are applied.

Besides the important property of low computational complexity, the use of these small block-size transforms in H.264/MPEG4-AVC has the advantage of significantly reducing ringing artifacts. For high-fidelity video, however, the preservation of smoothness and texture generally benefits from a representation with longer basis functions. A better trade-off between these conflicting objectives can be achieved by making use of the 8×8 integer.

For the quantization of transform coefficients, H.264/MPEG4-AVC uses uniform reconstruction quantizers (URQs). One of 52 quantizer step size scaling factors is selected for each macroblock by a quantization parameter (QP). The scaling operations are arranged so that there is a doubling in quantization step size for each increment of six in the value of QP.

The quantized transform coefficients of a block are usually scanned in a zig-zag fashion as shown in Figure 2.12 and further processed using the entropy coding.

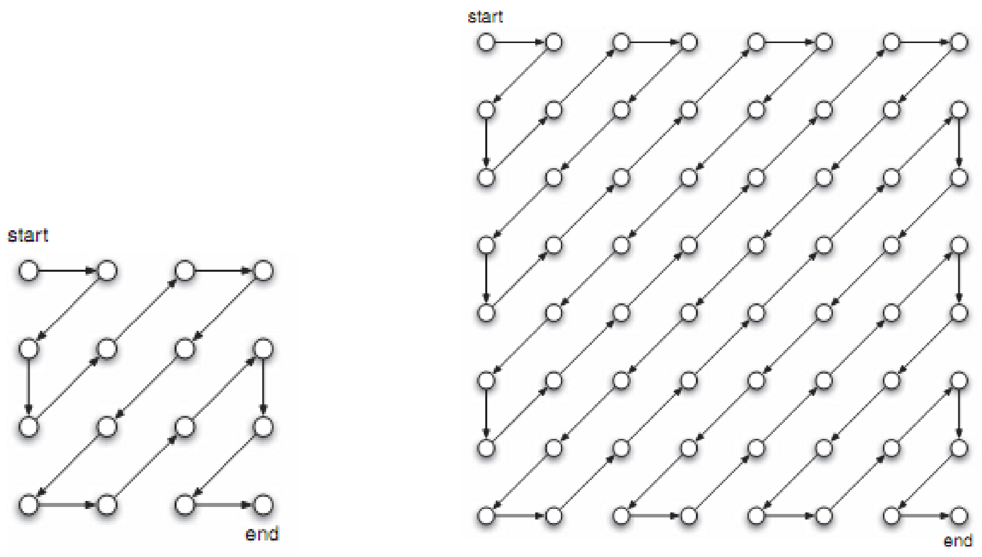


Figure 2.12 Zigzag scan orders for 4 × 4 and8 × 8 blocks [11]

2.1.9 In loop deblocking filter [1], [3], [11]

Since AVC uses coarse quantization, at low bit rates, the block-based coding typically shows clearly visually noticeable artifacts or discontinuities along the block boundaries. If no further provision is made to deal with this, these artificial discontinuities may also diffuse into the interior of blocks by means of the motion-compensated prediction process. The removal of such blocking artifacts can provide a substantial improvement in perceptual quality.

For that purpose, H.264/MPEG4-AVC defines a deblocking filter that operates within the predictive coding loop, and thus constitutes a required component of the decoding process. The filtering process exhibits a high degree of content adaptivity on different levels, from the slice level along the edge level down to the level of individual samples. As a result, the blockiness is reduced without much affecting the sharpness of the content. Consequently, the subjective quality is significantly improved. At the same time, the filter reduces bit rate by typically 5–10 percent while producing the same objective quality as the non-filtered video.

2.1.10 Entropy Coding [3]

In H.264/MPEG4-AVC, many syntax elements are coded using the same highly-structured infinite-extent variable-length code (VLC), called a zero-order exponential-Golomb code. A few syntax elements are also coded using simple fixed-length code representations. For the remaining syntax elements, two types of entropy coding are supported.

When using the first entropy-coding configuration, which is intended for lower-complexity (esp. software-based) implementations, the exponential-Golomb code is used for nearly all syntax elements except those of quantized transform coefficients, for which a more sophisticated method called context-adaptive variable length coding (CAVLC) is employed. When using CAVLC, the encoder switches between different VLC tables for various syntax elements, depending on the values of the previously transmitted syntax elements in the same slice. Since the VLC tables are designed to match the conditional probabilities of the context, the entropy coding performance is improved from that of schemes that do not use context-based adaptivity.

The entropy coding performance is further improved if the second configuration is used, which is referred to as context-based adaptive binary arithmetic coding (CABAC).

CABAC design is based on three components: binarization, context modeling, and binary arithmetic coding as shown in Figure 2.13. Binarization enables efficient binary arithmetic coding by mapping nonbinary syntax elements to sequences of bits referred to as bin strings. The bins of a bin string can each be processed in either an arithmetic coding mode or a bypass mode. The latter is a simplified coding mode that is chosen for selected bins such as sign information or lesser significance bins in order to speed up the overall decoding (and encoding) processes. Compared to CAVLC, CABAC can typically provide reductions in bit rate of 10–20 percent for the same objective video quality when coding DTV/HDTV signals.

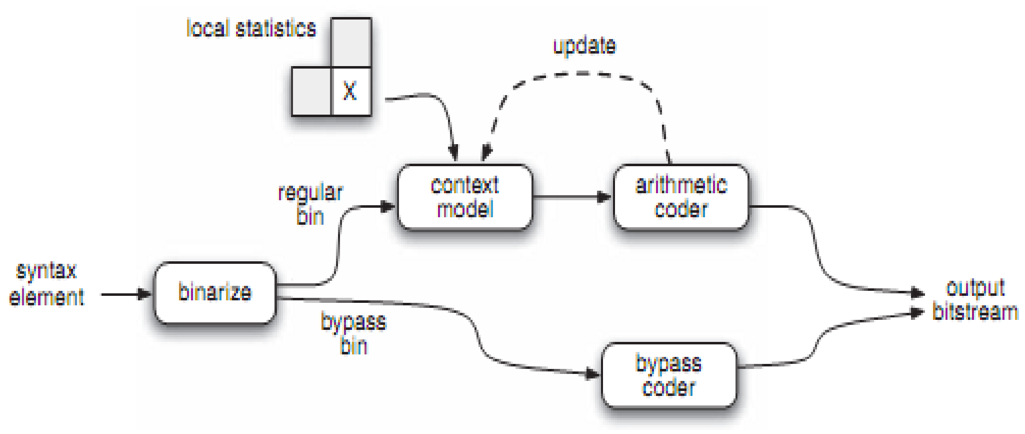


Figure 2.13 CABAC coding process [11]

2.2 Need for time complexity reduction [2], [8], [11]

The new standard is aimed at high-quality coding of video contents at very low bit-rates. H.264 uses the same hybrid block-based motion compensation and transform coding model. Furthermore, a number of new features and capabilities has been introduced in H.264 to efficiently improve the coding performance.

Finally, with all the enhancements available in H.264/AVC compared to previous standards, the codec outperforms in terms of compression efficiency and video quality. But, again, the price to pay for all of these advantages is the complexity involved in the standard, thereby making it challenging to the engineer or designer who has to develop, program or interface with an H.264 codec. As the standard becomes more complex, the encoding and decoding processes require much more computation power than most existing standards.

H.264 has various choices and parameters than any previous standard codec. Of course, getting the perfect controls and parameters for a particular application is not an easy task. If done correctly, H.264 delivers high compression performance; on the contrary, the result is poor-quality pictures and/or poor bandwidth efficiency.

Computationally expensive, an H.264 coder can lead to slow coding and decoding times or rapid battery drain on handheld devices. Thus, in order to make the codec usable on a handheld device and to be able to encode a video real time, there has to be some ways to reduce the time complexity involved in the video encoding.

Some of the different ways adopted to tackle this problem are: Changing the motion estimation algorithm and making intra/inter mode selection more time efficient, using Single Instruction Multiple Data (SIMD) execution model, optimizing the codec to make it run on DSP or dedicated hardware, parallel programming etc.

The complexity of emerging multimedia applications imposes new demands on processor performance. Most modern microprocessors, now a days have multimedia instructions and multithreading capabilities to facilitate multimedia applications. For example, the single-instruction-multiple-data (SIMD) execution model was introduced in Intel architectures. Intel also introduced hyper-threading technology [8], which enables a processor to execute multiple threads simultaneously. These advances in personal computers in addition to higher clock frequency have provided the necessary computational power for many multimedia applications.

To implement multimedia applications on personal computers requires some hardware-specific algorithm modifications. This thesis uses the parallel programming approach to reduce the overall encoding time by up to 60% by making use of four threads and making them work in parallel.

2.3 Summary

This chapter describes the working of H.264 codec. It then illustrates various blocks of the encoder such as transform and quantization, inter/intra prediction, motion estimation/motion compensation etc. The explanation gives overview of complexity involved in the codec. Finally it depicts the need for complexity reduction in the codec.

This thesis is aimed at complexity reduction in the H.264 codec using parallel programming technique. Next chapter mainly describes various basic concepts involved in parallel programming.

CHAPTER 3

PARALLEL PROGRAMMING BASICS

3.1 Introduction to parallel programming

Parallel programming and design of efficient parallel programs have been well established in high performance, scientific computing for many years. In parallel computing, calculations are carried out simultaneously, operating on the principle that large problems can often be divided into smaller ones, which are solved in parallel. Parallel computer programs are difficult to write as compared to sequential ones because, various potential software bugs are to be avoided e.g. data dependencies, race conditions etc. Communication and synchronization between the different subtasks are typically most challenging parts, in order to achieve good parallel program performance. There are several different forms of parallel computing: bit-level, instruction level, data, thread level, task parallelism.

3.2 Multicore [6], [8]

A multicore is an architecture design that places multiple processors on a single die (computer chip). Each processor is called a core. As chip capacity increased, placing multiple processors on a single chip became practical. These designs are known as Chip Multiprocessors (CMPs) because they allow for single chip multiprocessing. Multicore is simply a popular name for CMP or single chip multiprocessors. The concept of single chip multiprocessing is not new, and chip manufacturers have been exploring the idea of multiple cores on a uniprocessor since the early 1990s.

These days CMP [6] has become the preferred method of improving overall system performance. This is a departure from the approach of increasing the clock frequency or processor speed to achieve gains in overall system performance. Increasing the clock frequency has started to hit its limits in terms of cost - effectiveness. Higher frequency requires more power, making it harder and more expensive to cool the system. This also affects sizing and packaging considerations. So, instead of trying to make the processor faster to gain performance, the response is now just to add more processors. The simple realization that this approach is better has prompted the multicore revolution. Multicore architectures are now center stage in terms of improving overall system performance. The approaches to designing and implementing application software that will take advantage of the multicore processors are radically different from techniques used in single core development. Thus, the focus of software design and development should change from sequential programming techniques to parallel and multithreaded programming techniques.

In single core configurations there is one general purpose processor, although it is important to note that many of today's single core configurations contain special graphic processing units, multimedia processing units, and sometimes special math coprocessors. But even with single core or single processor computers multithreading, parallel programming, pipelining, and multiprogramming are all possible.

3.3 Parallel Computing in Microprocessors [7], [8]

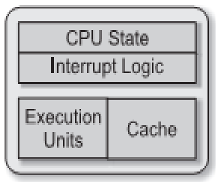
Parallel programming is the art and science of implementing an algorithm, a computer program, or a computer application, using sets of instructions or tasks designed to be executed concurrently.

With the advancements in software, applications have become increasingly capable of running multiple tasks simultaneously. There are several approaches, both in software and hardware, in order to support thread-level parallelism. One approach to address the increasingly concurrent nature of modern software involves using a preemptive, or time-sliced, multitasking operating system. Time-slice multi-threading allows developers to hide latencies associated with I/O by interleaving the execution of multiple threads. This model does not allow for parallel execution. Only one instruction stream can run on a processor at a single point in time. Another approach is to increase the number of physical processors in the computer. Multiprocessor systems allow true parallel execution; multiple threads or processes run simultaneously on multiple processors, but this comes with drawback of increased overall system cost.

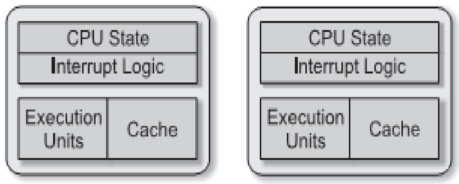
A thread, which is a basic unit of CPU utilization, contains a program counter that points to the current instruction in the stream. It also contains CPU state information for the current thread. It also contains other resources such as a stack.

A logical processor can thus be created by duplicating this architecture space in a physical processor. The execution resources can be shared among the different logical processors. This technique is known as simultaneous multi-threading or hyper threading technology [15], in terms of Intel’s implementation. It makes a single processor appear, from software’s perspective, as multiple logical processors. Like CMP, a hyperthreaded processor allows two or more threads to execute on a single chip. However, in a hyperthreaded package the multiple processors are logical instead of physical. There is some duplication of hardware but not enough to qualify a separate physical processor. So hyperthreading allows the processor to present itself to the operating system as complete multiple processors when in fact there is a single processor running multiple threads.

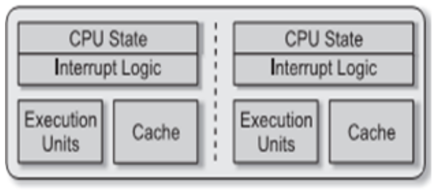
The next logical step is the multi-core processor. Multi-core processors use chip multiprocessing (CMP). Processor manufacturers take advantage of improvements in manufacturing technology to implement two or more “execution cores” within a single processor. These cores are nothing but two individual processors on a single die. Execution cores have their own set of execution and architectural resources. These processors may or may not share a large on-chip cache, depending on design. These individual cores, in turn, may be combined with SMT; effectively increasing the number of logical processors by twice the number of execution cores. The different processor architectures are highlighted in figure 3.1



a)



b)



c)

Figure 3.1 Simple comparison of single core, multiprocessor and multi core a) Single core, b) Multiprocessor c) Multicore [7]

3.4 Multi-threading on Single-Core versus Multi-Core Platforms [7]

These days, most modern applications use threads in one fashion or another. Hence, many developers are already familiar with the concept of threading, and have probably worked on applications that have multiple threads. However, there are certain important considerations developers should be aware of when writing applications targeting multi-core processors:

In order to achieve optimal application performance on multi-core architectures one must effectively use threads to partition software workloads.

Thread is used for improvement of user responsiveness on single-core platforms by many applications. Rather than blocking the user interface (UI) on a time consuming database query or disk access, an application will spawn a thread to process the user's request. This allows the scheduler to individually schedule the main control loop task that receives UI events as well as the data processing task that is running the database query. In this model, developers rely on straight-line instruction throughput improvements to improve application performance. This is the significant limitation of multi-threading on single-core processors. Since single-core processors are really only able to interleave instruction streams, but not execute them simultaneously, the overall performance gains of a multi-threaded application on single-core architectures are limited. On these platforms, threads are generally seen as a useful programming abstraction for hiding latency. This performance restriction is removed on multi-core architectures. On multi-core platforms, threads do not have to wait for any one resource. Instead, threads run independently on separate cores. For example, consider two threads that both wanted to execute a shift operation. If a core only had one "shifter unit" they could not run in parallel. On two cores, there would be two "shifter units," and each thread could run without contending for the same resource. Multi-core platforms, thus, allow developers to optimize applications by intelligently partitioning different workloads on different processor cores. Application code can be optimized to use multiple processor resources, resulting in faster application performance.

Multi-threaded applications running on multi-core platforms have different design considerations than do multi-threaded applications running on single-core platforms.

Considering the case of memory caching, each processor core may have its own cache. At any point in time, the cache on one processor core may be out of sync with the cache on the other processor core.

Suppose there are two threads running on a dual-core processor. Thread 1 runs on core 1 and thread 2 runs on core 2. The threads are reading and writing to neighboring memory locations. Since cache memory works on the principle of locality, the data values, while independent, may be stored in the same cache line. As a result, the memory system may mark the cache line as invalid, even though the data that the thread is interested in hasn’t changed. This problem is known as false sharing. On the other hand, considering a single-core platform, there is only one cache shared between threads; therefore, cache synchronization is not an issue. Thread priorities can also result in different behavior on single-core versus multi-core platforms. For example, consider an application that has two threads of differing priorities. In an attempt to improve performance, the developer assumes that the higher priority thread will always run without interference from the lower priority thread. On a single-core platform, this may be valid, as the operating system’s scheduler will not yield the CPU to the lower priority thread. However, on multi-core platforms, the scheduler may schedule both threads on separate cores. Therefore, both threads may run simultaneously. If the developer had optimized the code to assume that the higher priority thread would always run without interference from the lower priority thread, the code would be unstable on multicore and multi-processor systems.

3.5 Concurrency [15]

Two events are said to be concurrent if they occur within the same time interval. Two or more tasks executing over the same time interval are said to execute concurrently. Concurrent does not necessarily mean at the same exact instant. For example two tasks may execute concurrently within the same second but with each task executing within different fractions of the second. The first task may execute for the first tenth of the second and pause. The second task may execute for the next tenth of the second and pause. The first task may start again executing in the third tenth of a second and so on. Each task may alternate executing. However, the length of a second is so short that it appears that both tasks are executing simultaneously.

Concurrent tasks can execute in a single or multiprocessing environment. In a single processing environment, concurrent tasks exist at the same time and execute within the same time period by context switching. In a multiprocessor environment, if enough processors are free, concurrent tasks may execute at the same instant over the same time period. The determining factor for what makes an acceptable time period for concurrency is relative to the application.

3.6 Thread level Parallelism [6]

When implemented properly, threading can enhance performance by making better use of hardware resources.

A thread can be defined from both, hardware and software point of view. A thread is a discrete sequence of related instructions that is executed independently of other instruction sequences. In a program there is at least one thread called main thread, which, furthermore, can create other threads. On the other hand, at hardware level, thread is an execution path that remains independent of other hardware execution paths.

To take advantage of multicore processors, knowledge of details of software threading model as well as capabilities of the platform hardware is necessary.

3.7 Basic concepts in parallel programming

The design of parallel algorithm or program for a given application problem starts with decomposition of computations of an applications into several parts, called as tasks. There are various possibilities of decomposition for the same application algorithm. Identifying tasks that can run independently of each other in parallel is a challenging work.

The tasks of an application are coded in parallel programming language or environment and are assigned to processes or threads which are then assigned to physical computational units for execution. The assignment of these tasks or processes to threads, known as scheduling, can be done by hand in the source code or by the programming environment, at compile time or dynamically at runtime.

The assignment of processes or threads to physical computational units such as cores or processors is known as mapping. The main constraint for the scheduling is data dependencies between tasks, in which one task needs data produced by another task. In order to execute the overall code correctly, parallel program needs synchronization and coordination of threads.

It is important to know why we need parallel processing. In a typical case, a sequential code will execute in a thread which is executed on a single processing unit. Thus, if a computer has 2 processors or more (or 2 cores, or 1 processor with HyperThreading), only a single processor will be used for execution, thus wasting the other processing power. Rather than letting the other processor sit idle (or process other threads from other programs), we can use it to speed up our algorithm.

Parallel processing can be divided into two groups, task based and data based.

1) Task based: Divide different tasks to different CPUs to be executed in parallel. For example, a printing thread and a spell checking thread are running simultaneously in a word processor. Each thread is a separate task.
2) Data based: Execute the same task, but divide the work load on the data over several CPUs. For example, to convert a color image to grayscale. We can convert the top half of the image on the first CPU, while the lower half is converted on the second CPU (or as many CPUs as possible), thus processing in half the time.

3.8 Methods to do parallel processing

1) MPI: Message Passing Interface - MPI is most suited for a system with multiple processors and multiple memories. For example, a cluster of computers with their own local memory. You can use MPI to divide workload across this cluster, and merge the result when it is finished. Available with Microsoft Compute Cluster Pack.
2) OpenMP: OpenMP is suited for shared memory systems like we have on our desktop computers. Shared memory systems are systems with multiple processors but each are

sharing a single memory subsystem. Using OpenMP is just like writing your own smaller threads but letting the compiler do it. Available in Visual Studio 2005 Professional and Team Suite.

3) SIMD intrinsics: Single Instruction Multiple Data (SIMD) has been available on mainstream processors such as Intel's MMX, SSE, SSE2, SSE3, Motorola's (or IBM's) Altivec and AMD's 3DNow! SIMD intrinsics are primitive functions to parallelize data processing on the CPU register level. For example, the addition of two unsigned char will take the whole register size, although the size of this data type is just 8-bit, leaving 24-bit in the register to be filled with 0 and wasted. Using SIMD (such as MMX), we can load 8 unsigned chars (or 4 shorts or 2 integers) to be executed in parallel on the register level. Available in Visual Studio 2005 using SIMD intrinsics or with Visual C++ Processor Pack with Visual C++ 6.0.

3.9 Requirements of a parallel programming language

A parallel programming language must provide support for the three basic aspects of parallel programming: specifying parallel execution, communicating between multiple threads, and expressing synchronization between threads. Most parallel languages provide this support through extensions to an existing sequential language; this has the advantage of providing parallel extensions within a familiar programming environment. Different programming languages have taken different approaches to providing these extensions. Some languages provide additional constructs within the base language to express parallel execution, communication, and so on.

Rather than designing additional language constructs, other approaches provide directives that can be embedded within existing sequential programs in the base language.

Finally, application programming interfaces such as MPI and various threads packages such as Pthreads don’t design new language constructs, rather, they provide support for expressing parallelism through calls to runtime library routines.

3.10 Summary

This chapter starts with illustration of various concepts like multicore, multithreading, concurrency etc. It then briefly describes types of parallelism and methods for parallel processing. Next chapter is all about API used in this thesis- OpenMP.

CHAPTER 4

OPENMP: API SPECIFICATION FOR PARALLEL PROGRAMMING

4.1 Introduction [19], [20]

OpenMP (Open Multiprocessing) is an Application Program Interface (API) that can be used to explicitly direct multi-threaded, shared memory parallelism.
Pioneered by SGI and developed in collaboration with other parallel computer vendors, OpenMP is fast becoming the de facto standard for parallelizing applications. OpenMP provides a collection of compiler directives, library routines and environmental variables. It has been designed to introduce parallelism in existing sequential programs. It is, basically, a portable API specified for C/C++ and FORTRAN and a standard for the programming of shared memory systems.

There is an independent OpenMP organization today with most of the major computer manufacturers on its board, including Compaq, Hewlett-Packard, Intel, IBM, Kuck & Associates (KAI), SGI, Sun, and the U.S. Department of Energy ASCI Program. The OpenMP effort has also been endorsed by over 15 software vendors and application developers, reflecting the broad industry support for the OpenMP standard.

OpenMP is not a new computer language; it works in conjunction with either standard FORTRAN or C/C+ +. It is not meant for distributed memory parallel systems. In addition, it is not guaranteed to make the most efficient use of shared memory. It is consists of a set of compiler directives that describe the parallelism in the source code, along with a supporting library of subroutines available to applications.

These directives and library routines altogether are formally described by the API - OpenMP. The directives are instructional notes to any compiler supporting OpenMP. They take the form of source code comments (in FORTRAN) or #pragmas (in C/C+ +) in order to enhance application portability when porting to non-OpenMP environments. The simple code segment in Example 1.1 demonstrates the concept.

OpenMP is an implementation of multithreading, a method of parallelization whereby the master thread forks a specified number of slave threads and a task is divided among them. The threads then run concurrently, with the runtime environment allocating threads to different processors. The programming model of OpenMP is based on cooperating threads running simultaneously on multiple processors or cores. Thus, the OpenMP program begins with a main thread or master thread. Slave threads in the program are created and destroyed in a *fork-join* [18] pattern, as shown in figure 4.1. When the parallel construct is encountered, the initial thread, as a master thread creates a team of threads consisting of a certain number of new threads and the initial thread itself. This fork operation is performed implicitly. The program code inside the parallel construct is called as a parallel region and is executed in parallel by all threads of the team. At the end of a parallel region, there is implicit barrier synchronization, and only the master thread continues to execute after this region (implicit join operation). There can be nested parallel regions as well inside the code.

All OpenMP threads of a program have access to the same shared memory. Synchronization primitives have to be employed in the code in order to avoid conflicts, deadlocks, race conditions etc. The OpenMP provides various library routines for avoiding deadlocks, conflicts and race conditions. At compile time, multi-threaded program code is generated based on the compiler directives. Various compilers have a support for OpenMP standard.

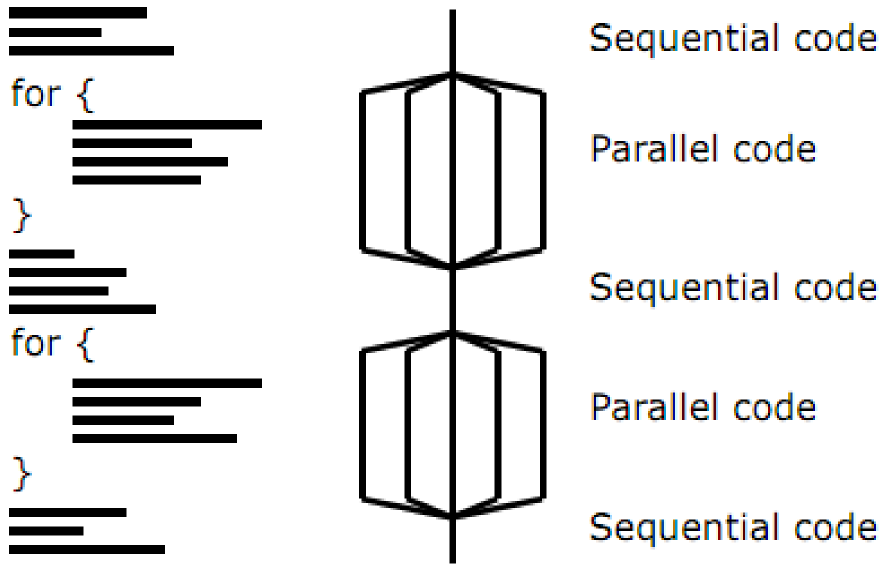


Figure 4.1 Fork/join model in OpenMP [18]

4.2 Goals of OpenMP [19]

1. To provide a standard among a variety of shared memory architectures/platforms
2. To establish a simple and limited set of directives for programming shared memory machines. Significant parallelism can be implemented by using just 3 or 4 directives
3. To provide capability to incrementally parallelize a serial program, unlike message-passing libraries which typically require an all or nothing approach
4. To provide the capability to implement both coarse-grain and fine-grain parallelism
5. To provide support for portability and C, and C++

4.3 OpenMP Programming Model [19], [20]

1) OpenMP is based upon the existence of multiple threads in the shared memory programming paradigm. There exist multiple threads in the shared memory programming paradigm. Figure 4.2 depicts the programming model or logical view presented to a programmer. In this case, all of the processors are able to directly access all of the memory in the machine, through a logically direct connection.

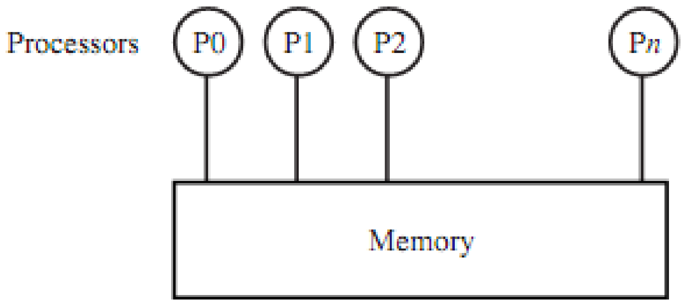


Figure 4.2 A canonical shared memory architecture [20]

2) OpenMP is an explicit (or non-automatic) programming model, which offers the programmer full control over parallelization.

3) As mentioned earlier, OpenMP uses the fork-join model of parallel execution as shown in figure 4.3

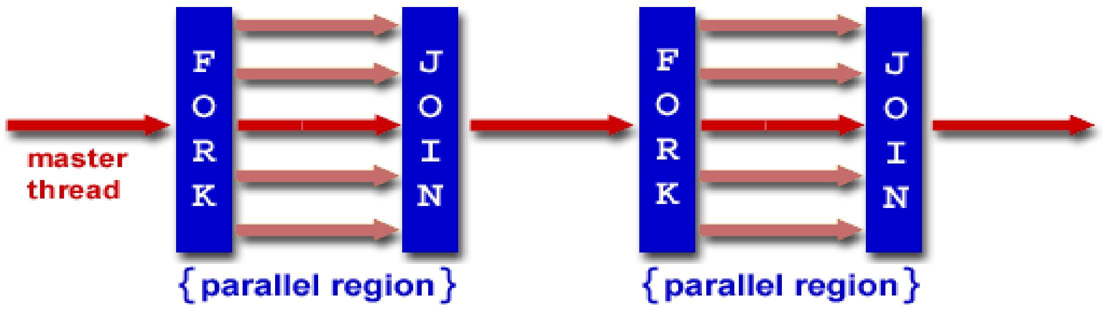


Figure 4.3 Parallel regions [19]

I. All OpenMP programs begin as a single process: the master thread. The master thread executes sequentially until the first parallel region construct is encountered.

II. FORK: the master thread then creates a team of parallel threads

III. The statements in the program that are enclosed by the parallel region construct are then executed in parallel among the various team threads

IV. JOIN: When the team threads complete the statements in the parallel region construct, they synchronize and terminate, leaving only the master thread.

4) Most OpenMP parallelism is specified through the use of compiler directives which are imbedded in C/C++ or FORTRAN source code.

5) The API provides support for the placement of parallel regions inside other parallel regions, called as nested parallelism.

6) The API allows runtime environment to dynamically alter the number of threads used to execute parallel regions. OpenMP is intended to promote more efficient use of resources, if possible.

7) Regarding parallel I/O, OpenMP specifies nothing about it. If every thread conducts I/O to a different file, the issues are not that significant. So, it is entirely up to the programmer to insure that I/O is conducted correctly within the context of a multithreaded program.

8) Components of OpenMP are as shown in figure 4.4.

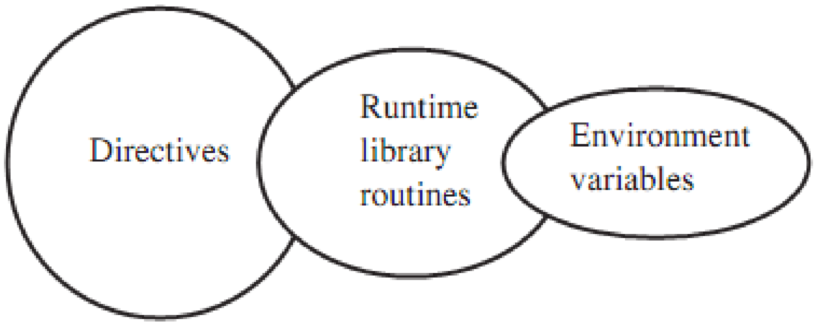


Figure 4.4 Components of OpenMP [20]

4.4 Reasons behind the popular usage of OpenMP [20]

There has seen a tremendous increase in the widespread availability and affordability of shared memory parallel systems since last decade. Such multiprocessor systems become more prevalent; they also contain increasing numbers of processors. Meanwhile, most of the high-level, portable standard parallel programming models are designed for distributed memory systems. The OpenMP aims at providing a standard and portable API for writing shared memory parallel programs.

There has been a surge in both the quantity and scalability of shared memory computer platforms over the last several years. Initially the systems contained only two processors, but this has quickly evolved to four- and eight-processor systems with scalability showing no signs of slowing. The growing demand for business/enterprise and technical/scientific servers has driven the quantity of shared memory systems in the medium- to high-end class machines as well. As the cost of these machines continues to fall, they are deployed more widely than traditional mainframes and supercomputers. Typical of these are bus-based machines in the range of 2 to 32 RISC processors like the SGI Power Challenge, the Compaq Alpha Server, and the Sun Enterprise servers. On the software front, the various manufacturers of shared memory parallel systems have supported different levels of shared memory programming functionality in proprietary compiler and library products. Application portability between different systems is extremely important to software developers. Basic goal of OpenMP, ultimately, is to provide a portable standard parallel API specifically for programming shared memory multiprocessors.

Programming with a shared memory model has been typically associated with ease of use at the expense of limited parallel scalability. On the other hand, distributed memory programming is usually regarded as more difficult but the only way to achieve higher levels of parallel scalability. Some of this common wisdom is now being challenged by the current generation of scalable shared memory servers coupled with the functionality offered by OpenMP.

The choice of an implementation model is largely determined by the type of computer architecture targeted for the application, the nature of the application, and a healthy dose of personal preference. The message passing programming model has now been very effectively standardized by MPI. MPI is a portable, widely available, and accepted standard for writing message passing programs. Unfortunately, message passing requires that the program’s data structures be explicitly partitioned, and typically the entire application must be parallelized in order to work with the partitioned data structures. Usually, there is no incremental path to parallelizing an application in this manner. Furthermore, modern multiprocessor architectures are increasingly providing hardware support for cache-coherent shared memory; therefore, message passing is becoming unnecessary and overly restrictive for these systems. The option of developing new computer languages may be the cleanest and most efficient way to provide support for parallel processing.

Initially, a pure library approach was considered as an alternative for what eventually became OpenMP. Reasons behind the rejection of a library only methodology are:

I. It is far easier to write portable code using directives because they are automatically ignored by a compiler that does not support OpenMP.

II. Since directives are recognized and processed by a compiler, they offer opportunities for compiler-based optimizations.

A pure directive approach is difficult as well, since some necessary functionality is quite difficult to express through directives. It finally ends up looking like executable code in directive syntax. Therefore, a small API defined by a mixture of directives and some simple library calls was chosen. The OpenMP API does address the portability issue of OpenMP library calls in non-OpenMP environments.

4.5 OpenMP Directives [22]

This section gives a brief overview of directives typically used in OpenMP. An OpenMP executable directive applies to the succeeding structured block or an OpenMP Construct.
A structured-block is a single statement or a compound statement with a single entry at the top and a single exit at the bottom.

4.5.1 parallel Construct

It is one of the basic constructs that initiates a parallel execution. The parallel construct forms a team of threads and starts parallel execution. It is used in the code as explained below:

```
#pragma omp parallel [clause[ [, ]clause] ...] new-line
        structured-block
```

where clause is one of the following:

```
if(scalar-expression)
num_threads(integer-expression)
default(shared | none)
private(list)
firstprivate(list)
shared(list)
copyin(list)
reduction(operator: list)
```

4.5.2 loop Construct

The loop construct specifies that the iterations of loops will be distributed among and executed by the encountering team of threads.

```
#pragma omp for  [clause[ [, ]clause] ...]
          for-loops
```

Clause:

```
private(list)
firstprivate(list)
lastprivate(list)
reduction(operator: list)
schedule(kind[, chunk_size])
collapse(n)
ordered
nowait
```

There can be various kinds in the loop as explained below:

1. static: Iterations are divided into chunks of size chunk_size. Chunks are assigned to threads in the team in round-robin fashion in order of thread number.

2. dynamic: Each thread executes a chunk of iterations then requests another chunk until no chunks remain to be distributed.
3. guided: Each thread executes a chunk of iterations then requests another chunk until no chunks remain to be assigned. The chunk sizes start large and shrink to the indicated chunk_size as chunks are scheduled.
4. auto: The decision regarding scheduling is delegated to the compiler and/or runtime system.
5. runtime: The schedule and chunk size are taken fromthe run-sched-var ICV.

The clauses *firstprivate* and *lastprivate* are explained in section 4.8

4.5.3 sections Construct

The sections construct contains a set of structured blocks that are to be distributed among and executed by the encountering team of threads.

```
        #pragma omp sections [clause[[,] clause] ...]
{
[#pragma omp section]
        structured-block
[#pragma omp section
        structured-block]
...
}
```

Clause:

private(list)

firstprivate(list)

lastprivate(list)

reduction(operator: list)

nowait

4.5.4 single Construct

The single construct specifies that the associated structured block is executed by only one of the threads in the team (not necessarily the master thread), in the context of its implicit task.

```
#pragma omp single [clause[ [, ]clause] ...]
        structured-block
```

Clause:

private(list)

firstprivate(list)

copyprivate(list)

nowait

4.5.5 parallel loop Construct

The parallel loop construct is a shortcut for specifying a parallel construct containing one or more associated loops and no other statements.

```
#pragma omp parallel for [clause[ [, ]clause] ...]
        for-loop
```

Clause:

Any accepted by the parallel or for directives, except the nowait clause, with identical meanings and restrictions.

4.5.6 parallel Sections Construct

The parallel sections construct is a shortcut for specifying a parallel construct containing one sections construct and no other statements.

```
#pragma omp parallel sections [clause[ [, ]clause] ...]
{
[#pragma omp section]
```

```
        structured-block
[#pragma omp section
        structured-block]
...
}
```

Clause:

Any of the clauses accepted by the parallel or sections directives, except the nowait clause, with identical meanings and restrictions.

4.5.7 task Construct

The task construct defines an explicit task. The data environment of the task is created according to the data-sharing attribute clauses on the task construct and any defaults that apply.

```
#pragma omp task [clause[ [, ]clause] ...]
        structured-block
```

Clause:

```
if(scalar-expression)
final(scalar-expression)
untied
default(shared | none)
mergeable
private(list)
firstprivate(list)
shared(list)
```

4.5.8 critical Construct

The critical construct restricts execution of the associated structured block to a single thread at a time.

```
#pragma omp critical [(name)]
        structured-block
```

4.5.9 master Construct

The master construct specifies a structured block that is executed by the master thread of the team. There is no implied barrier either on entry to, or exit from, the master construct.

```
#pragma omp master
        structured-block
```

4.5.10 barrier Construct

The barrier construct specifies an explicit barrier at the point at which the construct appears.

```
#pragma omp barrier
```

4.5.11 taskwait Construct

The taskwait construct specifies a wait on the completion of child tasks of the current task.

```
#pragma omp taskwait
```

4.5.12 atomic Construct

The atomic construct ensures that a specific storage location is updated atomically, rather than exposing it to the possibility of multiple, simultaneous writing threads.

```
#pragma omp atomic [read | write | update | capture]
        expression-stmt
#pragma omp atomic capture
        structured-block
```

<u>4.6 Runtime Library Routines</u>

Execution Environment Routines: Execution environment routines affect and monitor threads, processors, and the parallel environment.

4.6.1 void omp_set_num_threads(int num_threads)

Affects the number of threads used for subsequent parallel regions that do not specify a num_threads clause.

4.6.2 int omp_get_num_threads(void)

Returns the number of threads in the current team.

4.6.3 int omp_get_max_threads(void)

Returns maximum number of threads that could be used to form a new team using a parallel construct without a num_threads clause.

4.6.4 int omp_get_thread_num(void)

Returns the ID of the encountering thread where ID ranges from zero to the size of the team minus 1.

4.6.5 int omp_get_num_procs(void)

Returns the number of processors available to the program.

4.6.6 int omp_in_parallel(void)

Returns true if the call to the routine is enclosed by an active parallel region; otherwise, it returns false.

4.6.7 int omp_get_team_size(int level)

Returns, for a given nested level of the current thread, the size of the thread team to which the ancestor or the current thread belongs.

Lock Routines: Lock routines support synchronization with OpenMP locks.

*4.6.8 void omp_init_lock(omp_lock_t *lock)*

void omp_init_nest_lock(omp_nest_lock_t *lock);

These routines initialize an OpenMP lock.

*4.6.9 void omp_destroy_lock(omp_lock_t *lock)*

void omp_destroy_nest_lock(omp_nest_lock_t *lock);

These routines ensure that the OpenMP lock is uninitialized.

*4.6.10 void omp_set_lock(omp_lock_t *lock)* void omp_set_nest_lock(omp_nest_lock_t *lock);

These routines provide a means of setting an OpenMP lock.

*4.6.11 void omp_unset_lock(omp_lock_t *lock)*

void omp_unset_nest_lock(omp_nest_lock_t *lock);

These routines provide a means of unsetting an OpenMP lock.

*4.6.12 int omp_test_lock(omp_lock_t *lock);*

int omp_test_nest_lock(omp_nest_lock_t *lock);

These routines attempt to set an OpenMP lock but do not suspend execution of the task executing the routine.

4.7 Clauses

The set of clauses that is valid on a particular directive is described with the directive. Most clauses accept a comma-separated list of list items. All list items appearing in a clause must be visible.

Data Sharing Attribute Clauses:

Data-sharing attribute clauses apply only to variables whose names are visible in the construct on which the clause appears.

4.7.1 default(shared | *none)*

Controls the default data-sharing attributes of variables that are referenced in a parallel or task construct.

4.7.2 shared(list)

Declares one or more list items to be shared by tasks generated by a parallel or task construct.

4.7.3 private(list)

Declares one or more list items to be private to a task.

4.7.4 firstprivate(list)

Declares one or more list items to be private to a task, and initializes each of them with the value that the corresponding original item has when the construct is encountered.

4.7.5 lastprivate(list)

Declares one or more list items to be private to an implicit task, and causes the corresponding original item to be updated after the end of the region.

4.8 Environment Variables

Environment variables are described in section [4] of the API specification. Environment variable names are upper case, and the values assigned to them are case insensitive and may have leading and trailing white space.

4.8.1 OMP_SCHEDULE type [, chunk]

Sets the run-sched-var ICV for the runtime schedule type and chunk size. Valid OpenMP schedule types are static, dynamic, guided, or auto. Chunk is a positive integer that specifies chunk size.

4.8.2 OMP_NUM_THREADS list

Sets the nthreads-var ICV for the number of threads to use for parallel regions.

4.8.3 OMP_DYNAMIC dynamic

Sets the dyn-var ICV for the dynamic adjustment of threads to use for parallel regions. Valid values for dynamic are true or false.

4.8.4 OMP_NESTED nested

Sets the nest-var ICV to enable or to disable nested parallelism. Valid values for nested are true or false.

4.8.5 OMP_THREAD_LIMIT limit

Sets the thread-limit-var ICV that controls the maximum number of threads participating in the OpenMP program.

4.9 Race Conditions

A race condition exists when two unsynchronized threads access the same shared variable with at least one thread modifying the variable. The outcome may be unpredictable and depends on the timing of the threads in the team. Race conditions are an insidious problem because they can remain undetected for many thousands of executions, and it is not always obvious that the program has generated incorrect results. Because communications and synchronizations are often implicit in shared memory programming, race conditions can arise unexpectedly. It is the programmer's responsibility to ensure that the code is free from situations that could give rise to race conditions that corrupt the computational results. This article discusses some common scenarios that cause race conditions and provides easy coding alternatives to avoid them.

Following simple example demonstrates the race condition

```
int i=0;
#pragma omp parallel
{
   :
   i++;
   :
}
```

Consider a possible time-line for a two-thread example.

Timeline:

Clock	**Thread 0**	**Thread 1**
1	load i (i = 0)	
2	incr i (i = 1)	
3	swapped out	load i (i = 0)
4		incr i (i = 1)
5		store i (i = 1)
6	store i (i = 1)	swapped out

In this case, the result in i is 1 and not 2, as one would expect. Because the increment (++) operation is not atomic, it can be interrupted before completion and cause incorrect results. A simple increment on a shared variable like this is a prime candidate for the use of the OpenMP atomic directive, as shown below, which eliminates the possibility of a race condition.

```
#pragma omp atomic
            i++;
```

Finally, the following two-step process goes a long way towards eliminating race conditions from the code:

1. Identify all shared variables within an OpenMP region
2. Guard all modifications of those variables with critical regions or atomic directives, even when they look innocuous

Even though it is easy to write shared memory programs, it is not easy to write correct shared memory programs.

4.10 Summary

This chapter gives detailed description of key concepts in the OpenMP such as programming model, directives, constructs, environmental variables etc. In the end, it explains race conditions that can occur while processing parallely.

Next chapter describes how OpenMP is incorporated in this thesis to achieve task based parallelism. Finally, results, especially time complexity reduction are clearly mentioned with various graphs.

CHAPTER 5

RESULTS OF COMPLEXITY REDUCTION USING TASK BASED PARALLELISM

5.1 Prediction structures [11]

H.264 has various options for choosing reference pictures for inter prediction. Typically, the encoder uses reference pictures in a structured way.

Figure 5.1 shows low delay, minimal storage - type prediction structure in which there are only I and P slices. It is compatible with the Baseline Profile or Constrained Baseline Profile of H.264, which do not allow B slices, and would be suitable for an application requiring low delay and/or minimal storage memory at the decoder.

The first frame is coded as an I slice and subsequent frames are coded as P slices. P slices are, essentially, predicted from the previous frame. In this case, the efficiency of prediction is relatively low, because only one prediction direction and one reference is allowed for each frame. Such scheme is used for video conferencing where latency has to be kept as minimum as possible. I slices may be inserted in the stream at regular intervals to limit the propagation of transmission errors and to enable random access to the coded sequence

The original JM 18.0 software encodes a video sequence in the following manner, if baseline profile is selected. The original software runs on only one thread called main thread, executes serially and the software is not optimized, as far time complexity is concerned.

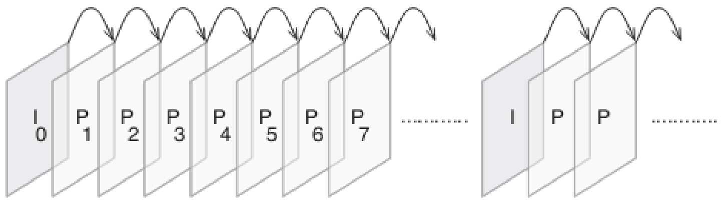


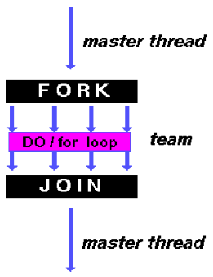
Figure 5.1 Low delay prediction structure [11]

.

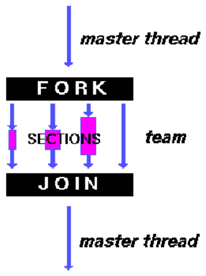
5.2 Task Based Parallelism using OpenMP [6], [22]

Until now the software developer could rely on the next new processor to speed up the software without having to make any actual improvements to the software. But now, the situation has changed. In order to increase overall system performance, computer manufacturers have decided to add more processors rather than increase clock frequency. This means if the software developer wants the application to benefit from the next new processor, the application will have to be modified to exploit multiprocessor computers.

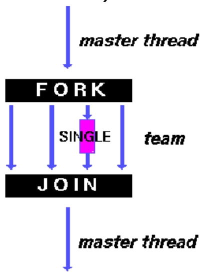
As described in Chapter 3, using OpenMP API, there are two basic types of parallelisms that can be incorporated in a code.



a)



b)



c)

Figure 5.2 Different types of parallelisms using OpenMP a) Data parallelism b) Functional Parallelism using SECTION c) Serialization of a section of code [19]

This thesis is based on dividing the tasks in the JM 18.0 reference software [12] in such a way that individual tasks can be assigned a thread and all threads can work in parallel.

As shown in Figure 5.2 b), in this thesis, SECTIONS are used, which work in parallel. As mentioned before, each thread executes its own section and thus, functional parallelism is implemented. Figure 5.3 depicts the actual scenario in this thesis. Each thread, which is assigned a task, may consist of many procedures, variables, and functions calls etc. There are total 100 frames used from the original video sequence to encode a video, in which 4 I frames are used and all remaining ones are P frames.

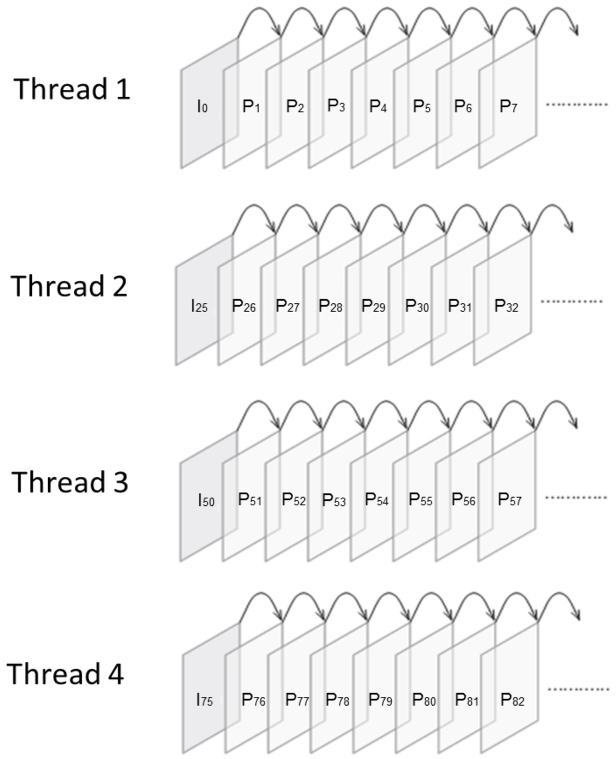

Thread 1
I_0 P_1 P_2 P_3 P_4 P_5 P_6 P_7
Thread 2
I_{25} P_{26} P_{27} P_{28} P_{29} P_{30} P_{31} P_{32}
Thread 3
I_{50} P_{51} P_{52} P_{53} P_{54} P_{55} P_{56} P_{57}
Thread 4
I_{75} P_{76} P_{77} P_{78} P_{79} P_{80} P_{81} P_{82}

Figure 5.3 Parallel encoding of 100 frames using 4 threads, with each thread encoding 25 frames [11]

5.3 Experimental Results

5.3.1 QCIF and CIF sequences

For testing, 8 CIF (352 × 288) and 8 QCIF (176 × 144) [27] sequences have been used with frame rate selected as 25 Hz. Compared to original JM 18.0 reference software [12], results obtained by optimizing the software are shown based on PSNR, bit rate, SSIM (Structural Similarity Index Metric) [26] and total encoding time.

SSIM, a recently proposed approach to image fidelity measurement has proven to be highly effective for measuring the fidelity of coded images. Human visual system is highly adapted to extract structural information from visual scenes; this is the basis of SSIM. For image fidelity measurement, the retention of signal structure should be an important ingredient.

Equivalently, an algorithm may seek to measure structural distortion to achieve image fidelity measurement. If the human visual system is considered as an ideal information extractor that seeks to identify and recognize objects in the visual scene, then it must be highly sensitive to the structural distortions and automatically compensates for the nonstructural distortions. Thus, an effective objective signal fidelity measure simulates this functionality.

CIF (Common Intermediate Format) is a format used to standardize the horizontal and vertical resolutions in pixels of Y, C_b, C_r sequences in video signals, commonly used in video teleconferencing systems.

QCIF means "Quarter CIF". To have one fourth of the area as "quarter" implies the height and width of the frame are halved.

The differences in Y, C_b, C_r of CIF and QCIF are shown in Figure 5.4.

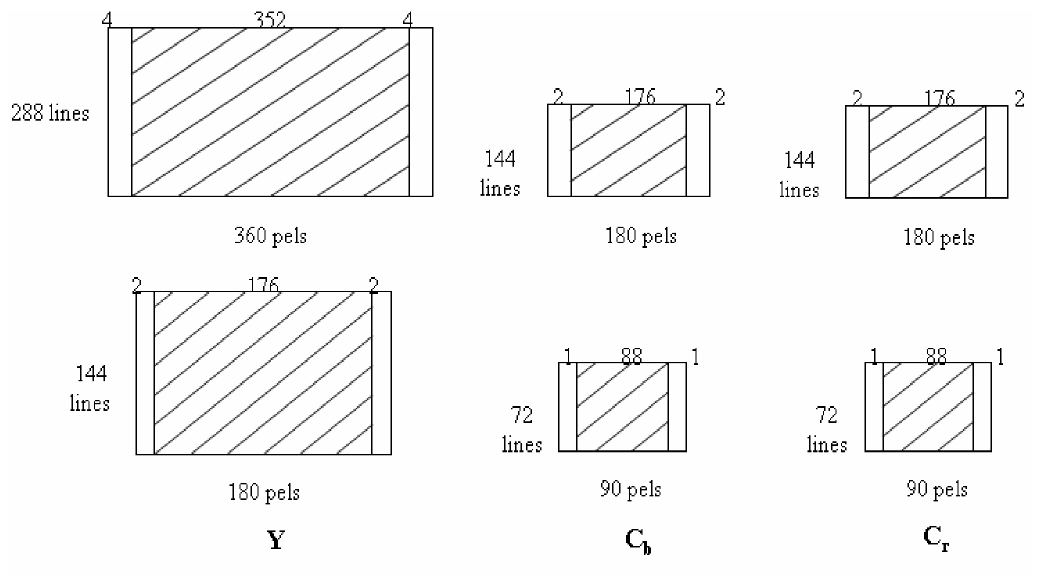


Figure 5.4 CIF and QCIF formats [27]

5.3.2 Preview of test sequences [26] used

Following are various test sequences that have been used in this thesis:

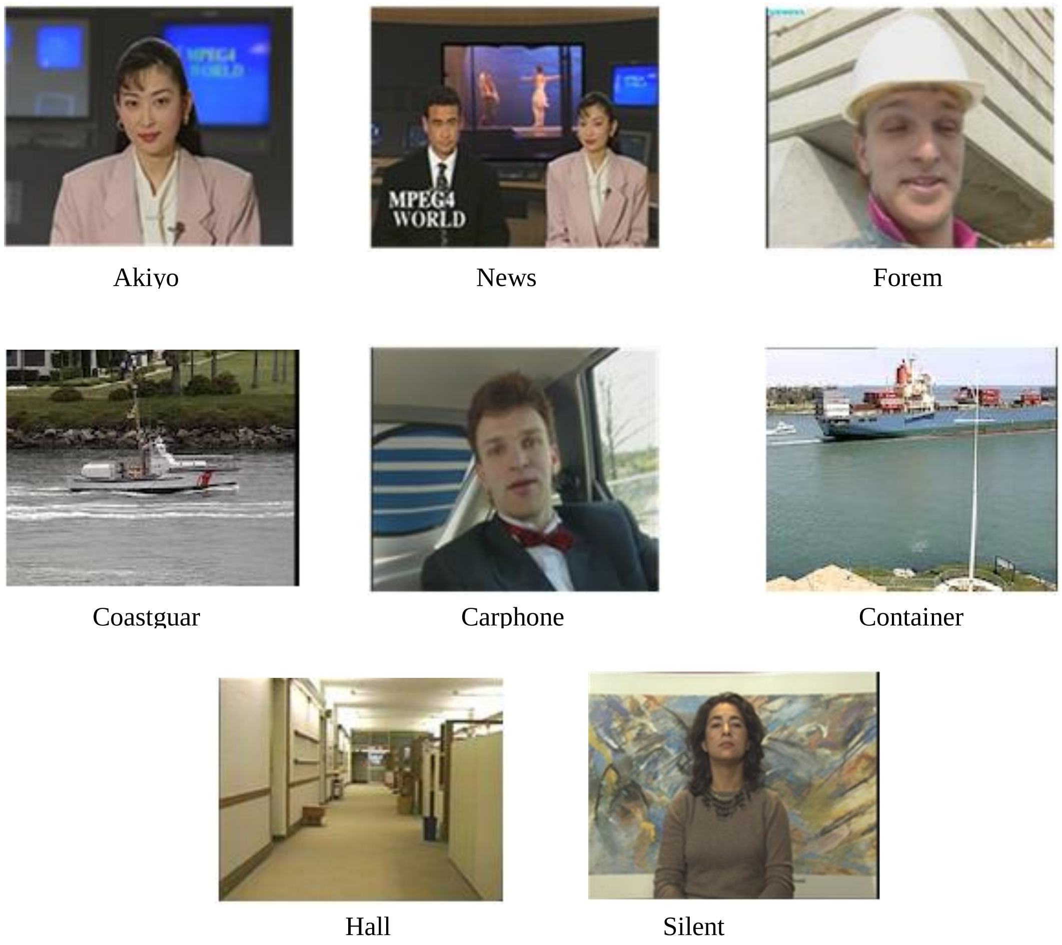


Figure 5.5 Preview of various test sequences used for testing [26]

5.3.3 Performance metrics

The results are compared in terms of change of PSNR (ΔPSNR), bit-rate (Δ bit rate), SSIM (ΔSSIM) and encoding time (Δ Time).

5.3.3.1 %T reduction

Computational efficiency is measured by the amount of time reduction, which is computed as follows:

$$\Delta\,\mathrm{T}\,(\%) = \frac{\text{Time taken by original software} - \text{Time taken by optimized software}}{\text{Time taken by original software}} * 100$$

5.3.3.2 Delta bit rate

$$\Delta\,\text{bit rate}\;\;\% \;= \frac{\text{Bit rate}\;\frac{\text{kbits}}{\text{s}}\;\text{by original software} - \text{Bit rate}\;\frac{\text{kbits}}{\text{s}}\;\text{by optimized software}}{\text{Bit rate}\;\frac{\text{kbits}}{\text{s}}\;\text{by original software}} * 100$$

5.3.3.3 Δ PSNR (Peak Signal to Noise Ratio) is computed as follows:

$$\Delta\,\text{PSNR}\;\;\% \;= \frac{\text{PSNR}\;\;\text{dB}\;\;\text{by original software} - \text{PSNR}\;\;\text{dB}\;\;\text{by optimized software}}{\text{PSNR}\;\;\text{dB}\;\;\text{by original software}} * 100$$

5.3.3.4 Δ SSIM (%) can be measured on similar lines, as follows:

$$\Delta\,\text{SSIM}\,(\%) = \frac{\text{SSIM by original software} - \text{SSIM by optimized software}}{\text{SSIM by original software}} * 100$$

5.3.4 Encoding specifications

5.3.4.1 Software

GOP structure is IPPP (No B frames), motion estimation search range was set to 32 pixels for both QCIF and CIF, QP values were varied as 22, 27, 32, 37, Hadamard transform was used, number of reference frames set to 5, CABAC was enabled, 100 frames encoded.

5.3.4.2 Hardware

Processor: Intel(R) core(TM) i5 CPU, 2.53GHz

RAM: 4.0 GB

Operating system: Windows 7 Home Premium (64 bit)

5.3.5 Results obtained with QCIF sequences

Tables 5.1 and 5.2 show the final simulation results of QCIF videos for various QP values. Results are tabulated based on templates used in [29]. From the results, it can be observed that, more than 60% average encoding time reduction was achieved with the optimized software using OpenMP.

Table 5.1 Simulation results for QCIF video sequences at QP = 22, 27

Test Sequence (QCIF)	**QP = 22**				**QP = 27**			
	ΔT (%)	ΔPSNR (%)	Δ Bit rate (%)	Δ SSIM (%)	ΔT (%)	ΔPSNR (%)	Δ Bit rate (%)	Δ SSIM (%)
akiyo	62.159	0.284	-2.427	-0.01	61.769	-0.506	-6.458	-0.061
carphone	62.051	0.556	-1.309	0	59.682	0.655	-3.939	-0.01
coastguard	62.488	0.594	-0.878	0.05	61.812	0.498	-3.328	0.105
container	61.88	-0.138	-5.716	-0.124	60.42	-0.572	-15.188	-0.148
foreman	60.221	0.3	-6.189	0.05	60.294	0.026	-9.315	0
hall	60.752	0.582	-1.77	0.081	61.818	-0.29	-6.669	-0.03
news	61.02	0.379	-2.315	0.02	60.786	0.169	-6.464	0.01
silent	61.253	0.276	4.455	0.121	61.207	0.054	0.72	0.165

Table 5.2 Simulation results for QCIF video sequences at QP = 32, 37

Test Sequence (QCIF)	**QP = 32**				**QP = 37**			
	ΔT (%)	ΔPSNR (%)	Δ Bit rate (%)	Δ SSIM (%)	ΔT (%)	ΔPSNR (%)	Δ Bit rate (%)	Δ SSIM (%)
akiyo	61.815	0.237	-5.175	-0.063	62.593	0.034	-13.583	-0.189
carphone	63.148	0.338	-5.452	0.033	60.276	0.166	-9.755	0.076
coastguard	62.777	-0.127	-2.736	-0.092	64.046	-1.705	-9.983	0.04
container	60.994	-1.051	-26.344	-0.334	60.217	-1.286	-26.829	0.066
foreman	60.127	-0.261	-13.916	0	60.366	-0.387	-15.118	-0.311
hall	62.328	-0.694	-13.787	-0.104	62.132	-1.227	-12.847	-0.3
news	61.443	0.782	-7.727	0.105	60.881	0.897	-10.176	-0.365
silent	61.269	0.481	1.449	-0.021	60.949	0.851	2.317	-0.401

5.3.6 Results obtained with CIF sequences

Tables 5.3 and 5.4 show the final simulation results of CIF videos for various QP values. Results are tabulated based on templates used in [29]. From the results, it can be observed that, more than 60% average encoding time reduction was achieved with the optimized software using OpenMP.

Table 5.3 Simulation results for CIF videos at QP = 22, 27

Test Sequence (CIF)	**QP = 22**				**QP = 27**			
	ΔT (%)	Δ PSNR (%)	Δ Bit rate (%)	Δ SSIM (%)	ΔT (%)	Δ PSNR (%)	Δ Bit rate (%)	Δ SSIM (%)
akiyo	62.781	0.422	-0.894	0.03	64.546	0.227	-0.699	0.02
carphone	60.129	0.168	-0.544	0.134	62.193	0.726	-0.663	0.125
coastguard	62.291	-0.275	-0.515	-0.092	62.905	-0.489	-2.914	-0.107
container	60.394	0.289	-0.903	0.0312	61.126	0.148	-2.195	0.0541
foreman	61.096	-0.009	-0.693	0.061	59.396	0.04	-2.07	-0.053
hall	60.038	0.267	-0.6	0.0103	58.021	0.36	-3.106	0.02
news	60.119	0.175	0.228	0.04	59.904	0.189	-1.77	0.041
silent	60.86	0.195	-0.971	0.072	63.122	0.134	-1.396	0.042

Table 5.4 Simulation results for CIF videos at QP = 32, 37

Test Sequence (CIF)	**QP = 32**				**QP = 37**			
	ΔT (%)	Δ PSNR (%)	Δ Bit rate (%)	Δ SSIM (%)	ΔT (%)	Δ PSNR (%)	Δ Bit rate (%)	Δ SSIM (%)
akiyo	62.338	0.691	-0.568	0.041	59.925	0.448	-2.397	0.032
carphone	61.067	1.418	-0.657	0.054	60.438	0.597	-2.594	0.044
coastguard	62.723	-0.185	-1.067	0.0718	64.149	0.942	-2.328	0.0144
container	59.522	0.425	-2.419	0.09	60.136	0.639	-5.416	0.105
foreman	60.575	0.437	-2.263	0.044	61.236	0.965	-2.654	0.0811
hall	61.104	0.232	-2.405	0.031	61.664	0.415	-1.933	-0.086
news	59.502	0.326	-2.212	0.052	64.135	0.413	-4.026	0.021
silent	58.385	0.168	-1.777	0.056	61.736	0.366	-4.656	0.036

5.3.7 Graphs of average encoding time for all QCIF sequences

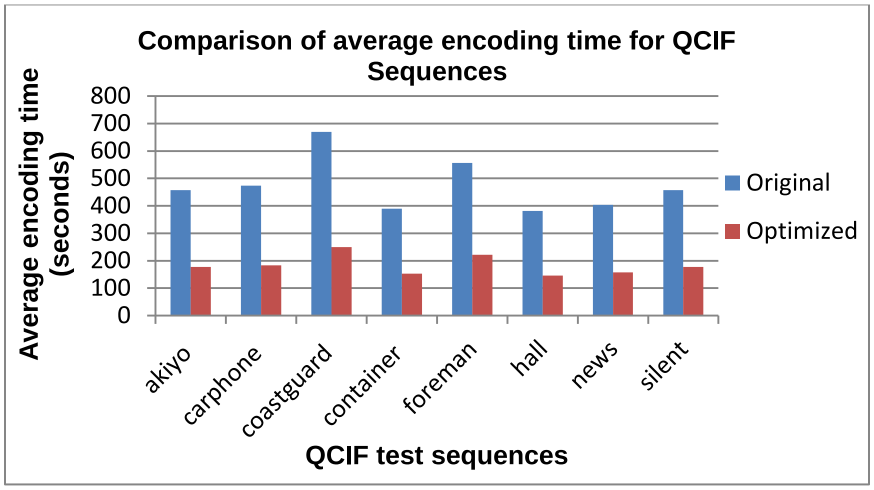


Figure 5.6 Comparison of average encoding time for all QCIF sequences

5.3.8 Graphs of average encoding time for all CIF sequences

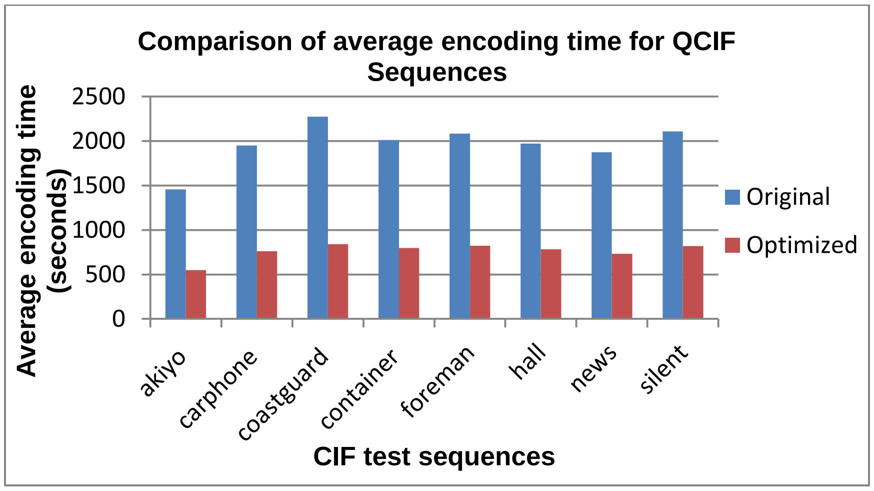


Figure 5.7 Comparison of average encoding time for all CIF sequences

5.3.9 Rate-Distortion graphs for QCIF sequences

5.3.9.1 Akiyo_qcif.yuv

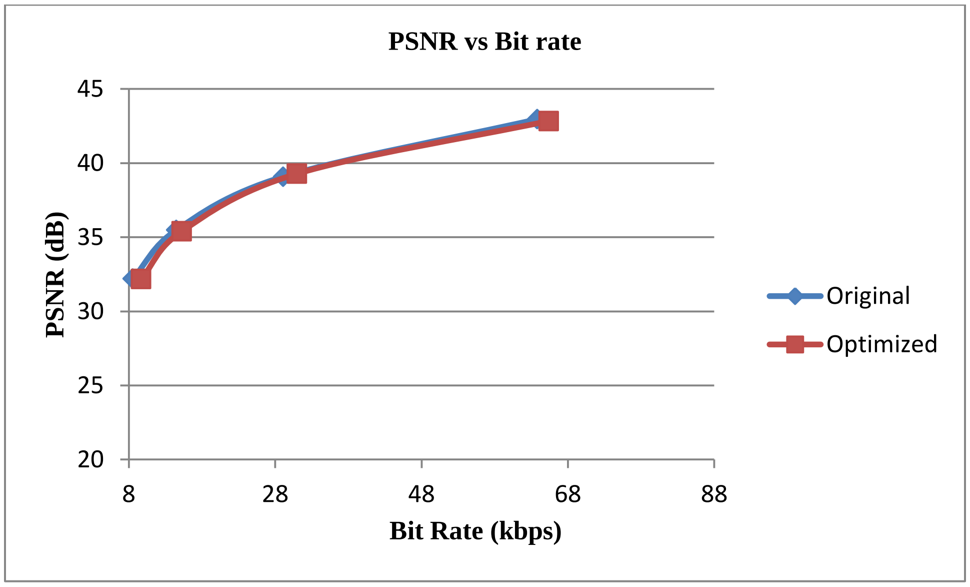


Figure 5.8 Rate-distortion graph for Akiyo_qcif.yuv

5.3.9.2 Carphone_qcif

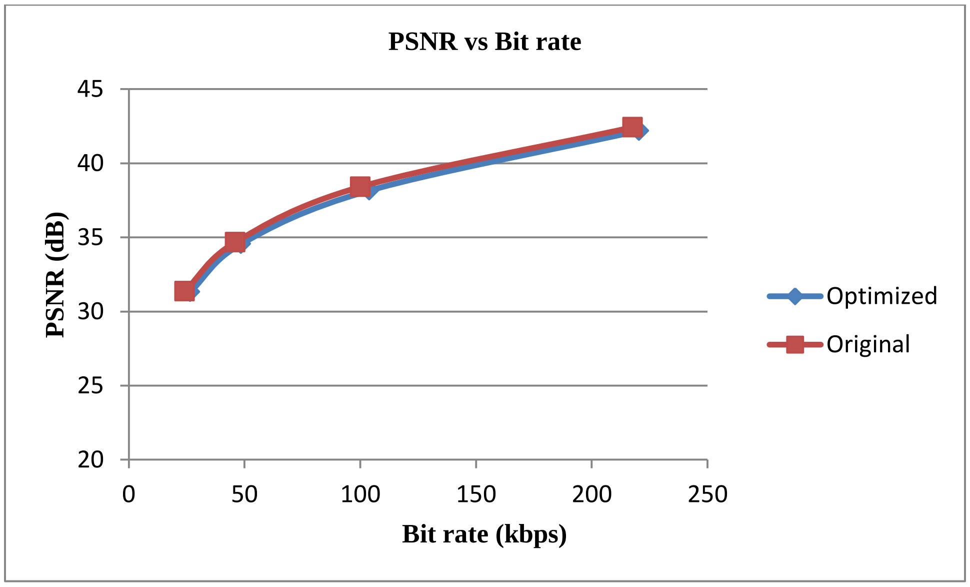


Figure 5.9 Rate-distortion graph for Carphone_qcif.yuv

5.3.9.3 Coastguard_qcif

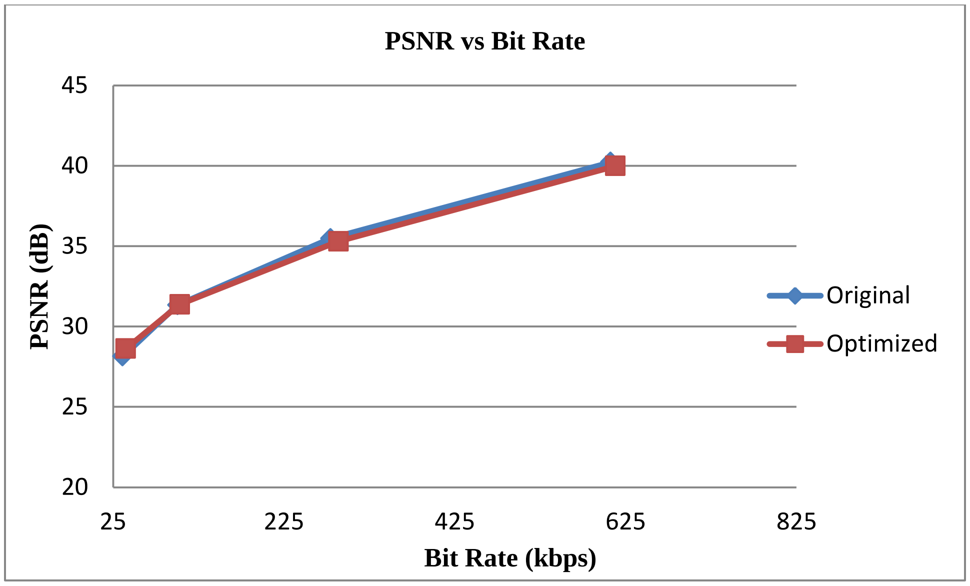


Figure 5.10 Rate-distortion graph for Coastguard_qcif.yuv

5.3.9.4 Container_qcif.yuv

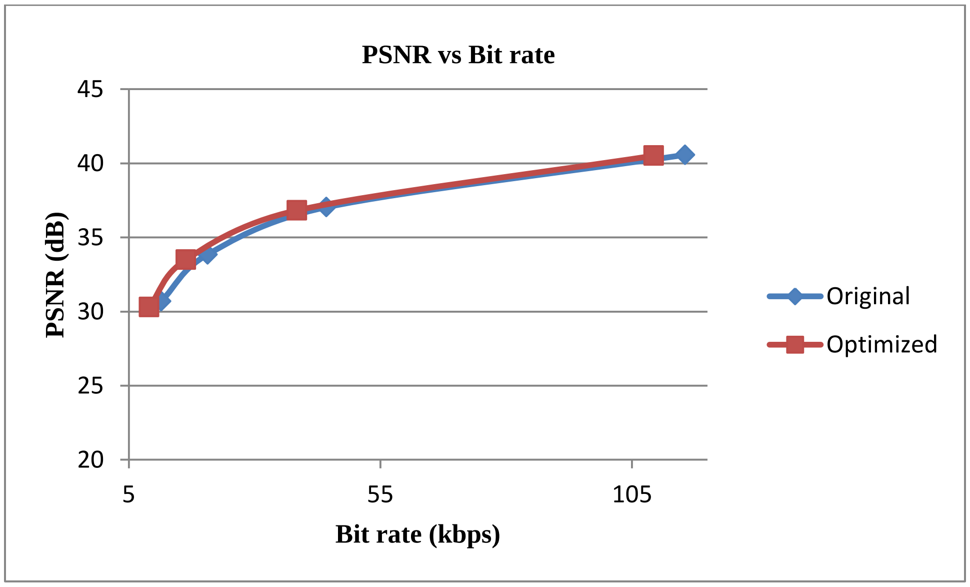


Figure 5.11 Rate-distortion graph for Container_qcif.yuv

5.3.9.5 Foreman_qcif.yuv

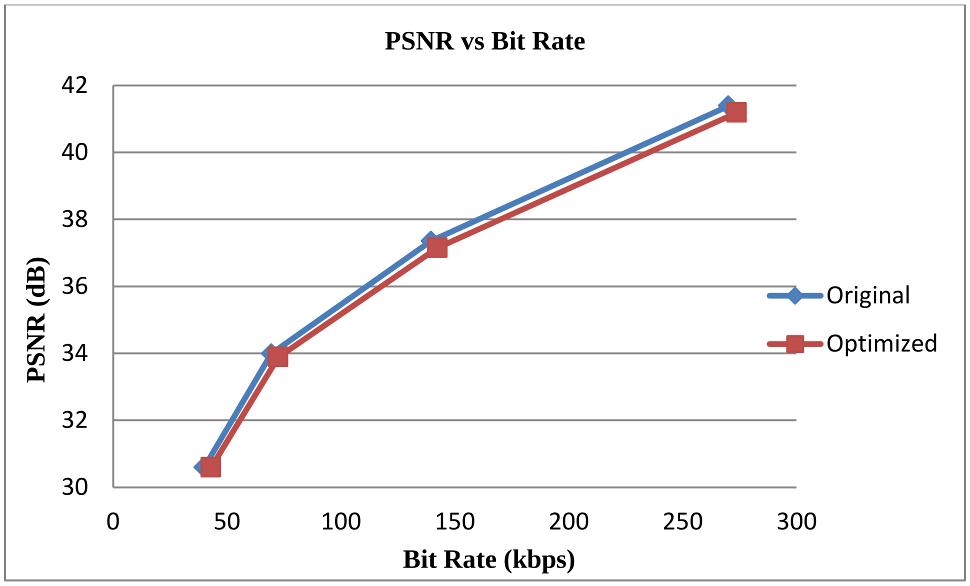


Figure 5.12 Rate-distortion graph for Foreman_qcif.yuv

5.3.9.6 Hall_qcif.yuv

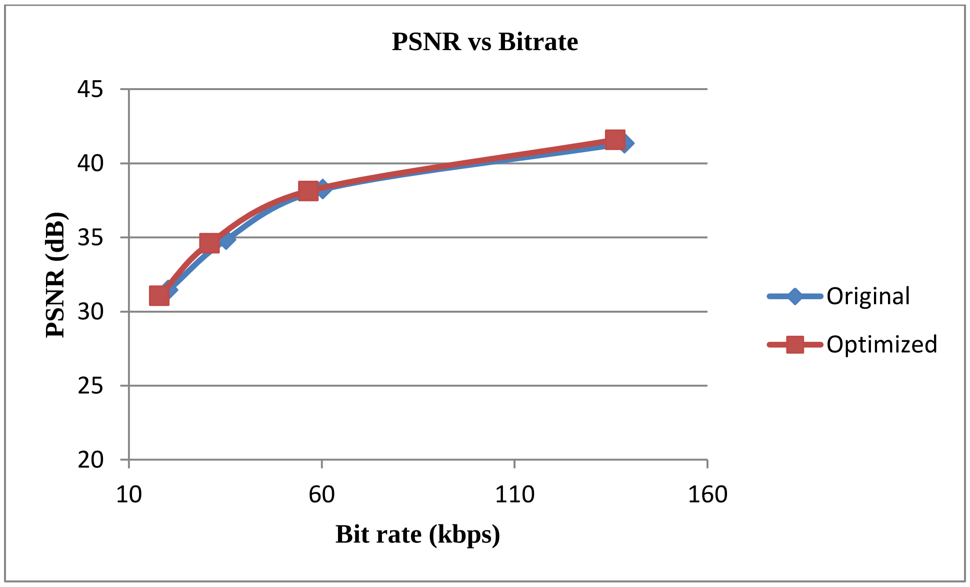


Figure 5.13 Rate-distortion graph for Hall_qcif.yuv

5.3.9.7 News_qcif.yuv

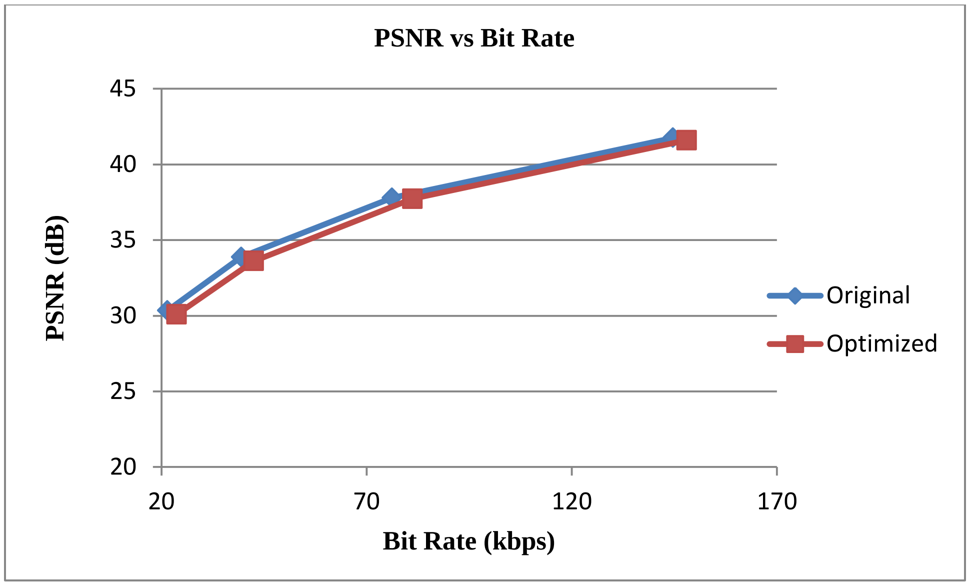


Figure 5.14 Rate-distortion graph for News_qcif.yuv

5.3.9.8 Silent_qcif.yuv

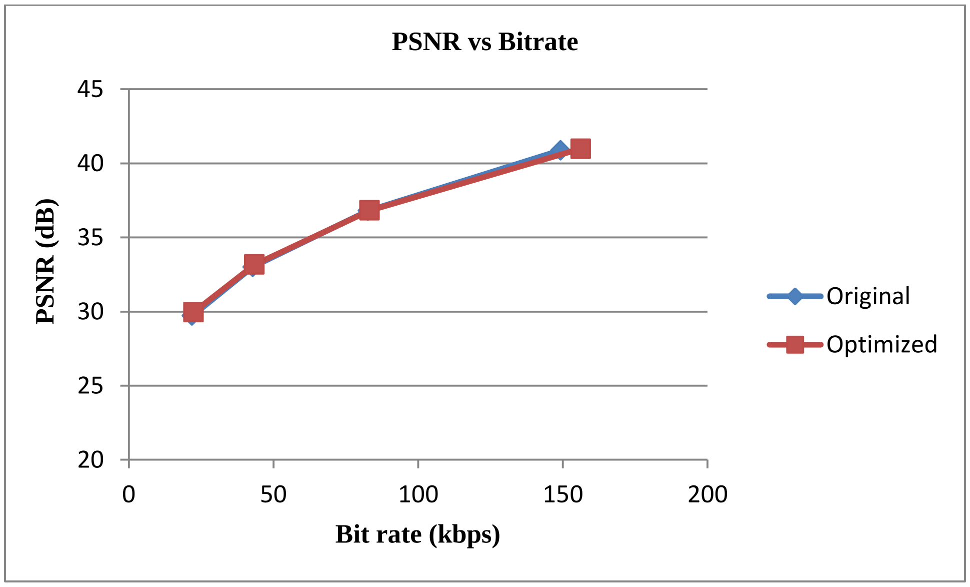


Figure 5.15 Rate-distortion graph for Silent_qcif.yuv

5.3.10 Rate-Distortion graphs for CIF sequences

5.3.10.1 Akiyo_cif,yuv

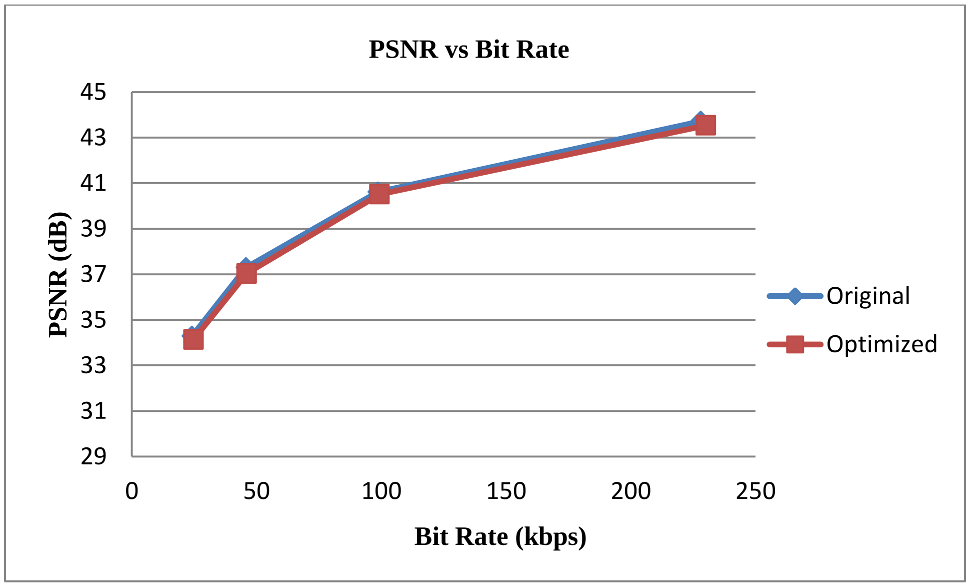


Figure 5.16 Rate-distortion graph for Akiyo_cif.yuv

5.3.10.2 Carphone_cif.yuv

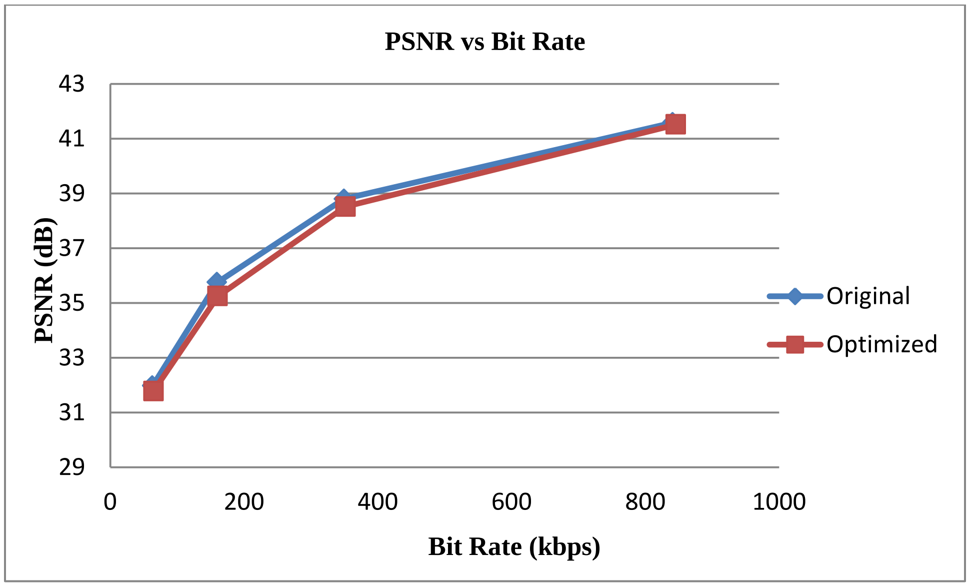


Figure 5.17 Rate-distortion graph for Carphone_cif.yuv

5.3.10.3 Coastguard_cif.yuv

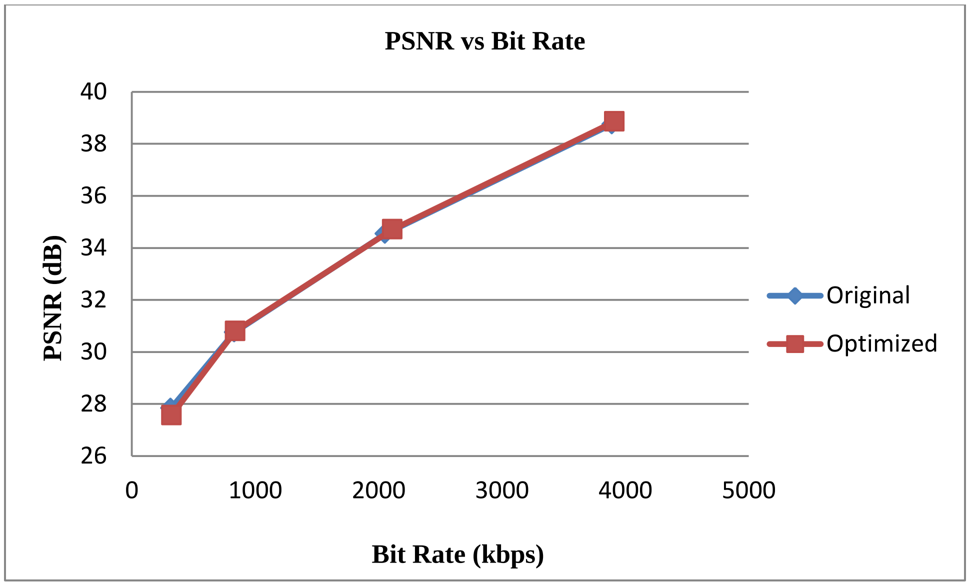


Figure 5.18 Rate-distortion graph for Coastguard_cif.yuv

5.3.10.4 Container_cif.yuv

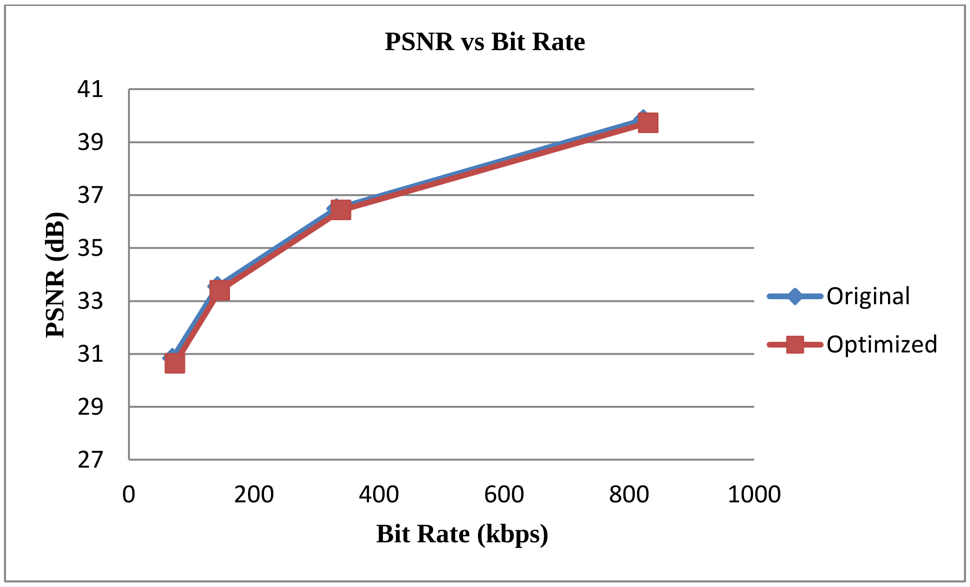


Figure 5.19 Rate-distortion graph for Container_cif.yuv

5.3.10.5 Foreman_cif.yuv

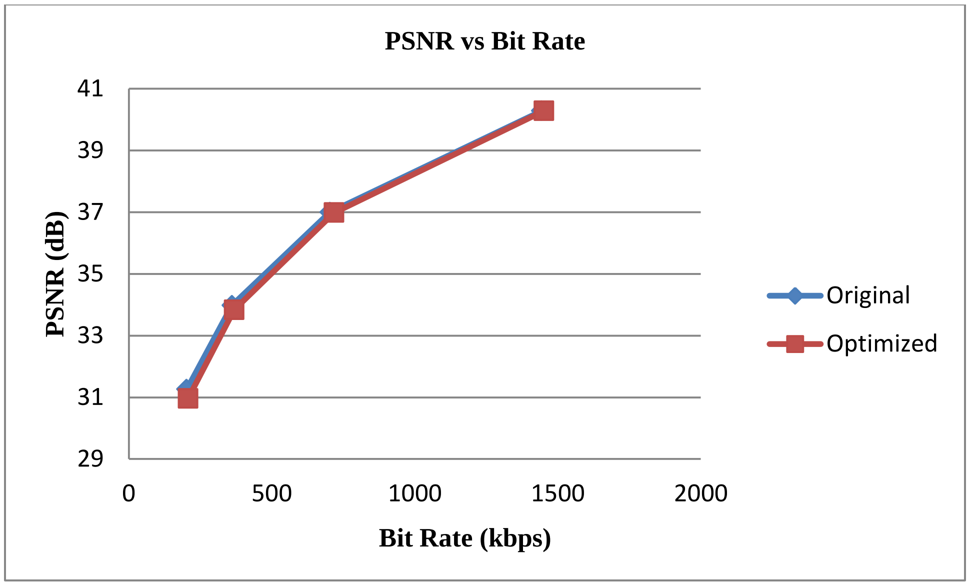


Figure 5.20 Rate-distortion graph for Foreman_cif.yuv

5.3.10.6 Hall_cif.yuv

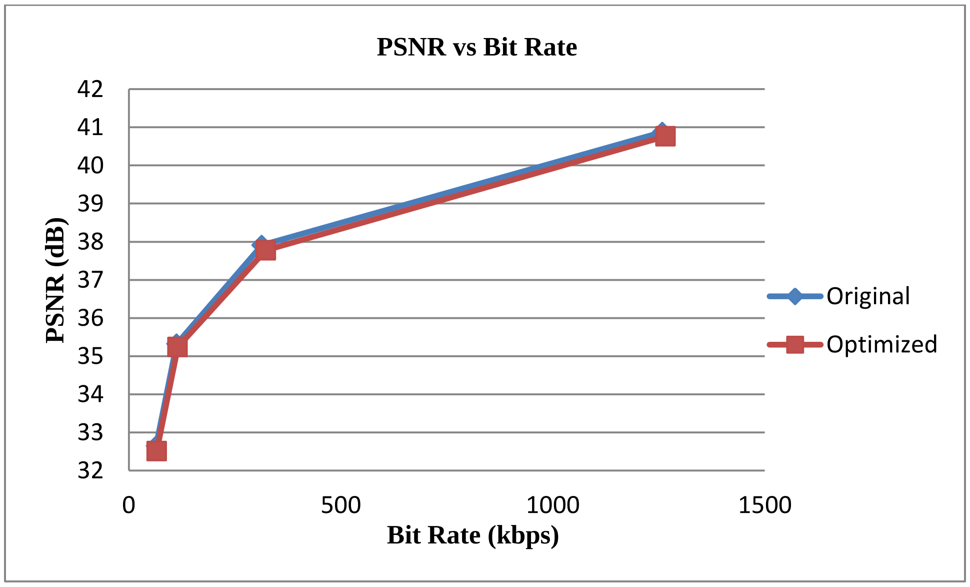


Figure 5.21 Rate-distortion graph for Hall_cif.yuv

5.3.10.7 News_cif.yuv

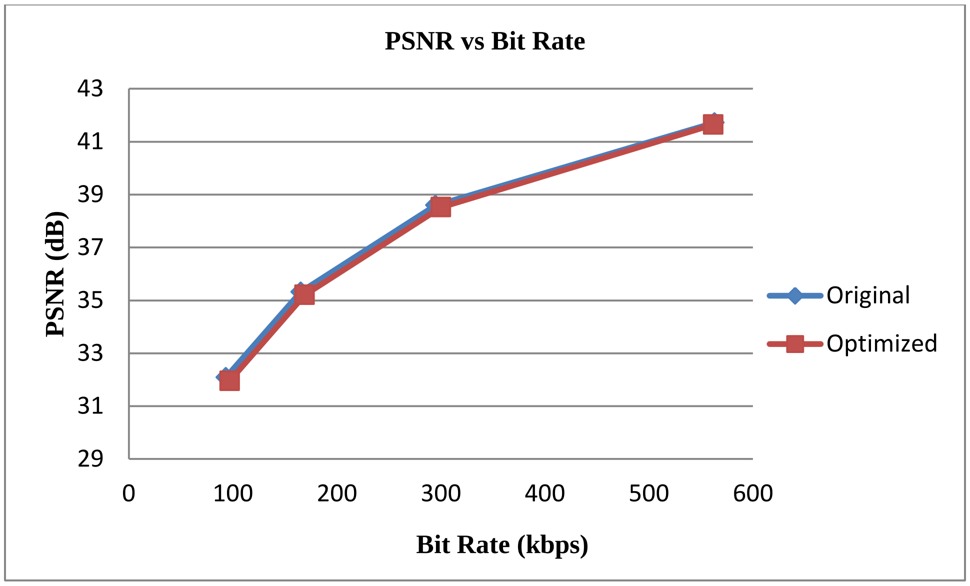


Figure 5.22 Rate-distortion graph for News_cif.yuv

5.3.10.8 Silent_cif.yuv

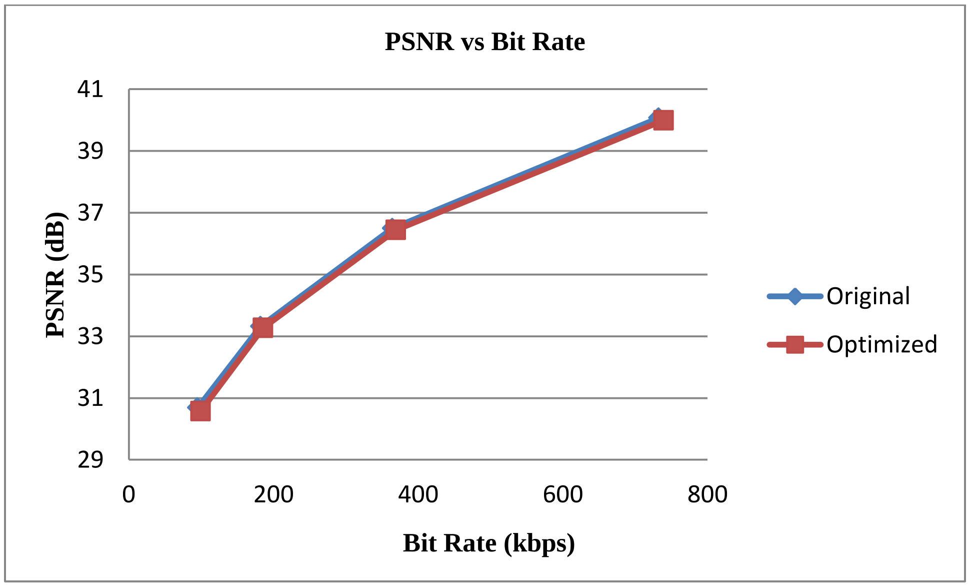


Figure 5.23 Rate-distortion graph for Silent_cif.yuv

5.4 Analysis of task based parallelism used in this thesis

Results can be analyzed based on comparison of task based parallelism incorporated in this thesis with data level parallelism method.

5.4.1 Comparison of task based with data level parallelism method

A new efficient data parallel algorithm for H.264 encoder based on the MB region partition is explained in [35]. It clearly explains data dependencies involved in intra prediction inter prediction, loop filtering and CAVLC. The algorithm used, firstly, partitions a frame into several MB regions, in which each little square stands for a MB, and each MB region comprises several adjoining columns of MBs. These MB regions are mapped onto different processors, and the data is exchanged appropriately according to the data dependencies. Parallel algorithm uses the wave-front technique.

The algorithm used in [35] gives maximum of 78% time reduction with average 72.67% time reduction. It is very challenging to change the way original software code is written to a parallel algorithm with less data dependencies. This thesis avoids complicated algorithm changes in order to get rid of data dependencies and focuses on task based parallelism. The results mentioned in [35] indicate that the time complexity reduction achieved with data parallelism is 20% more as compared to task based parallelism used in this thesis.

5.4.2 Limitations of task based parallelism method

The optimized software in this thesis runs on a dual core processor by Intel with 4 GB of RAM along with Windows 7 operating system running. If the same code is ported and run on a single core processor, although with HTT enabled, does not result in more than 60% encoding time reduction.

Optimized software assumes that there is already raw video data available and then encodes the video with H.264 standard using four threads running in parallel. The software cannot be directly used for encoding a real time video directly, due to unavailability of entire raw video data prior to start encoding.

5.5 Summary

This chapter clearly shows capabilities of OpenMP. Results shown are based on implementation of modified encoder software on a CPU having two cores. More than 60% encoding time reduction can be achieved with the modified codec.

Next chapter concludes the thesis with an idea of what can be implemented in the future using parallel programming.

CHAPTER 6

CONCLUSIONS AND FUTURE WORK

6.1 Conclusions

Data dependencies in a program pose a major challenge in parallel processing, especially for multimedia related applications. Data level parallelism was the first option considered in this thesis but, it was observed that the encoding time, in fact, was increasing. This was due to recreation of the threads for each new frame, again and again, which incurred a high thread creation overhead. Thus, finally, this thesis incorporates task based parallelism, rather than data level parallelism, in which, encoding of 4 sub video streams is done in parallel.

Chapter 4 gives a good idea of capabilities of parallel programming by showing significant time complexity reduction. Encoding time, PSNR, bit-rate and SSIM comparisons were performed on CIF and QCIF sequences. The rigorous testing was done using QP in steps of 22, 27, 32 and 37.

Based on the results tabulated in the previous chapter, it can be concluded that the modified software using OpenMP APIs is much faster as compared to original serially executing JM 18.0 reference software [12].

Up to 64.89% encoding time reduction was achieved using the new modified algorithm, without significant degradation in the output video quality, increment in bit rate.

6.2 Future work

This thesis is based on the idea of changing the original software using parallel programming based on task based parallelism using OpenMP. But, thread creation overhead is more in this case. The same idea of using parallel programming can be incorporated using Compute Unified Device Architecture (CUDA) [34] programming model by NVIDIA and implemented on GPU using the concept of GPGPU.

CUDA is a parallel computing platform and programming model invented by NVIDIA. With the power of the graphics processing unit (GPU), CUDA dramatically increases the computing performance. It is a scalable parallel programming model and a software environment for parallel computing. CUDA threads are extremely lightweight, having very low creation overheads and switching time. Execution model using CUDA, if managed properly can give dramatic results, as far as time complexity reduction in H.264 encoder is concerned

REFERENCES

[1] Soon-kak Kwon, A. Tamhankar and K.R. Rao, “Overview of H.264/MPEG-4 part 10”, JVCIR vol. 17, pp. 186-216, April 2006.

[2] T. Wiegand, et al “Overview of the H.264/AVC video coding standard”, IEEE Trans. on circuits and systems for video technology, vol. 13, pp. 560-576, July 2003.

[3] D. Marpe, T. Wiegand and G. J. Sullivan, “The H.264/MPEG-4 AVC standard and its applications”, IEEE Communications Magazine, vol. 44, pp. 134-143, Aug. 2006.

[4] J. Kim, et al “Complexity reduction algorithm for intra mode selection in H.264/AVC video coding” J. Blanc-Talon et al. (Eds.): ACIVS 2006, LNCS 4179, pp. 454 – 465, 2006.Springer-Verlag Berlin Heidelberg, 2006.

[5] Ju-Ho Hyun, “Fast mode decision algorithm based on thread-level parallelization and thread slipstreaming in H.264 video coding” Multimedia and Expo (ICME), 2010 IEEE International Conference

[6] C. Hughes and T. Hughes, “Professional Multicore Programming Design and Implementation for C++ Developers”, Wiley 2010

[7] S. Akhter and J. Roberts, “Multi-Core Programming Increasing Performance through Software Multi-threading”, Intel Press 2006

[8] Eric Q. Li and Yen-Kuang Chen, “Implementation of H.264 Encoder on General-Purpose Processors with Hyper-Threading Technology”, Visual Communications and Image Processing 2004, edited by S. Panchanathan and B. Vasudev, Proc. of SPIE-IS&T Electronic Imaging, SPIE Vol. 5308

[9] B. Jung, et al “Adaptive Slice-Level Parallelism for Real-Time H.264/AVC Encoder with Fast Inter Mode Selection”, Multimedia Systems and Applications X, edited by S. Rahardja, J.W. Kim and J. Luo, Proc. of SPIE Vol. 6777, 67770J, (2007)

[10] S. Ge, X. Tian and Yen-Kuang Chen, "Efficient Multithreading Implementation of H.264 Encoder on Intel Hyper-Threading Architectures", ICICS-PCM 2003.

[11] I. Richardson, "The H.264 advance video compression standard", 2nd Edition, Wiley 2010.

[12] JM software – http://iphome.hhi.de/suehring/tml/

[13] J. Ren, et al, "Computationally efficient mode selection in H.264/AVC video coding", IEEE Trans. Consumer Electronics, vol. 54, pp. 877 – 886, May 2008.

[14] H.264/ MPEG-4 Part 10 White Paper: www.vcodex.com.

[15] T. Rauber and G. Runger, "Parallel Programming for Multicore and Cluster Systems", 2nd edition, Wiley Publishing, 2008.

[16] OpenMP - http://openmp.org/wp/

[17] Intel Software Network –http://software.intel.com/en-us/articles/getting-started-with-openmp/

[18] System Overview of Threading: http://ranger.uta.edu/~walker/CSE%205343_4342_SPR11/Web/Lectures/Lecture-4-Threading%20Overview-Ch2.pdf

[19] OpenMP Introduction: https://computing.llnl.gov/tutorials/openMP/#Introduction

[20] R. Chandra, et al "Parallel Programmingin OpenMP", Academic Press, 2001.

[21] www.openmp.org/mp-documents/OpenMP3.0-SummarySpec.pdf

[22] OpenMP Manual: http://www.openmp.org

[23] Detecting and Avoiding OpenMP Race Conditions in C++: http://developers.sun.com/solaris/articles/cpp_race.html

[24] Digital Video Coding Standards and Their Role in Video Communications: http://www.cs.ucsb.edu/~almeroth/classes/F03.201B/papers/video-coding.pdf

[25] Test sequences: http://media.xiph.org/video/derf/

[26] Z. Wang, et al, “Image quality assessment: From error visibility to structural similarity,” IEEE Trans. Image Processing, vol. 13, pp. 600–612, Apr. 2004.

[27] T.Wiegand et al, “Rate-constrained coder control and comparison of video coding standards,” IEEE Trans. Circuits Systems Video Technology, vol. 13, no.7, pp.688-703, July 2003.

[28] T.Purushotham, “Low complexity H.264 encoder using machine learning”, M.S. Thesis, E.E Dept, UTA, 2010.

[29] A. Kulkarni, “Implementation of fast inter-prediction mode decision in H.264/AVC video encoder”, M.S. Thesis, E.E Dept, UTA, 2012.

[30] S.Muniyappa, “Implementation of complexity reduction algorithm for intra mode selection in H.264/AVC”, M.S. Thesis, E.E Dept, UTA, 2011.

[31] K.R. Rao and J.J. Hwang, “Techniques and Standards for Image/Video/Audio Coding”, Prentice Hall, 1996.

[32] D. Han, A. Kulkarni and K.R. Rao, “Fast Inter-prediction Mode Decision Algorithm for H.264 Video Encoder”, International Research Journal of Engineering Science, Technology and Innovation www.interesjournals.org (under review)

[33] D. Han, A. Kulkarni and K.R. Rao, “Fast inter-prediction mode decision algorithm for H.264 video encoder”, ECTICON 2012, Cha Am, Thailand, May 2012.

[34] Getting Started with CUDA: http://www.nvidia.com/content/cudazone/download/Getting_Started_w_CUDA_Training_NVISION08.pdf

[35] Shuwei Sun, et al, “A Highly Efficient Parallel Algorithm for H.264 Encoder Based on Macro-Block Region Partition”, Springer-Verlag Berlin Heidelberg, pp. 577–585, 2007

BIOGRAPHICAL INFORMATION

Tejas Pravin Sathe was born in Pune, India in 1987.He received the Bachelor's degree in Electronics and Communication Engineering from Pune University, India in 2009. He worked as Research Assistant in College of Engineering, Pune, India from Dec. 2009 to July 2010. In addition, from Jan. 2010 to Apr. 2010, he worked as Adjunct Faculty in Pune Vidyarthi Griha's College of Engineering and Technology, Pune, India.

He decided to pursue the Master's degree from The University of Texas at Arlington in Fall 2010. He worked as a Graduate Research Assistant under Dr. Rao in the Multimedia Processing Lab from Spring 2011 to Summer 2011. He got an opportunity to work as OS Embedded Software Developer, Intern at Research in Motion in Sunrise, Florida from Fall 2011 till Spring 2012. His interests lie in the field of video coding and embedded systems.