COMPLEXITY REDUCTION OF H.264 USING PARALLEL PROGRAMMING

by

SUDEEP PRAKASH GANGAVATI

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

December 2012

ACKNOWLEDGEMENTS

The successful completion of my thesis is due to the invaluable suggestions and support of number of people.

This research work would not have been possible without the support and encouragement of my supervisor Dr.K.R.Rao. I am honored to present my special thanks and sincere gratitude to Dr.K.R.Rao for his guidance and insightful feedback during the duration of this thesis.

I would like to thank Dr. Alan Davis and Dr. Saibun Tjuatja for taking time off from their busy schedule and being a part of my thesis committee.

I take this opportunity to thank my colleagues and friends at LSI Corporation for all the guidance and suggestions during my short tenure at LSI Corporation. My sincere thanks to Bahman Barazesh and George Kustka for their guidance and advice.

I thank my friends, Abhishek, Avinash, Bharath, Harsha, Manan,Subrahmanya,Sumanth and Sujay for their suggestions.

I am forever grateful to my grandmothers Pramilabai Gangavati and Sucheta Chaklabbi for their blessings and wishes.

Finally, I thank my family for their continued motivation, unconditional support and wishes. I thank my father, Prakash Gangavati and my mother, Sadhana Gangavati for their constant support and advice.

November 19, 2012

ABSTRACT

COMPLEXITY REDUCTION OF H.264 USING PARALLEL PROGRAMMING

Sudeep Gangavati, M.S

The University of Texas at Arlington, 2012

Supervising Professor: K.R.Rao

The H.264 encoder provides for adaptive directional intra-prediction, motion-compensated inter-prediction followed by transform, quantization, deblocking filtering and either variable length encoding or arithmetic coding. All these blocks of the H.264 encoder make it highly complex as compared to previous video coding standards. There is a need for measures to reduce the complexity. This thesis aims at reducing the complexity of the .H.264 encoder by reducing the encoding time of the H.264 standard while not sacrificing the video quality, compression efficiency and bitrate by parallel programming. There are several parallel programming models that can be used. The massively parallel Graphics Processing Units (GPUs) provided by NVIDIA Corp. are used in this thesis for parallel processing. The main focus of the thesis is to reduce the time it takes for the motion estimation during the inter prediction. Motion estimation is the most compute-intensive process of H.264 and involves basic mathematical operations like subtraction and addition between the pixels of the reference frame and the frame under prediction. The subtraction between the pixels, also known as the sum of absolute differences (SAD) is done in parallel. The frame is partitioned into smaller 8 x 8 blocks and for these blocks, threads are invoked on the GPU and all the calculations are done in parallel. The largest macroblock has a size 16 x 16, if this is divided into 4 equal parts each of size 8 x 8, a set of threads called blocks are invoked that carry out the operation for each particular 8 x 8 block. Up to 50% reduction in total time is observed for various input sequences of different characteristics.

For implementation of the encoder, JM 16.0 reference software is used in this thesis. The manual for this reference is available and provides reference of different encoder parameters, syntax, and additional information regarding the best practices and configurations of the software.

TABLE OF CONTENTS

LIST OF ILLUSTRATIONS

LIST OF TABLES

LIST OF ACRONYMS

API - Application Programming Interface

ASO - Arbitrary Slice Order

AVC - Advanced Video Coding

B MB - B Macro Block

B slice – Bi-directional predictive slice

CABAC - Context Adaptive Binary Arithmetic Coding

CAVLC - Context Adaptive Variable Length Coding

CAVLC - Context- Adaptive Variable Length Coding

CIF - Common Intermediate Format

CPU - Central Processing Unit

CUDA – Compute Unified Device Architecture

DCT - Discrete Cosine Transform

FMO - Flexible Macro-block Order

FPS - Frames per Second

GOP - Groups of Pictures

GPGPU – General Purpose Graphics Processing Units

GPU – Graphics Processing Unit

H420P - High 4:2:0 profile

H422P - High 4:2:2 profile

H444P - High 4:4:4 profile

HP - High Profile

HPC – High Performance Computing

I MB - I Macro Block

I slice – Intra predictive slice

IDCT - Inverse Discrete Cosine Transform

JPEG - Joint Picture Experts Group

JVT - Joint Video Team

MB - Macro Blocks

ME - Motion estimation

MIMD - Multiple Instructions Multiple Data

MISD - Multiple Instructions Single Data

MPEG - Moving Picture Experts Group

MV - Motion Vector

OpenMP - Open Multiprocessing

P MB - P Macro Block

P slice – Inter predictive slice

PSNR - Peak Signal to Noise Ratio

QCIF - Quarter Common Intermediate Format

QP - Quantization Parameter

RDO - Rate Distortion Optimization

SIMD - Single Instruction Multiple Data

SISD - Single Instruction Single Data

SSIM - Structural Similarity Index Metric

VCEG - Video Coding Experts Group

CHAPTER 1

INTRODUCTION

1.1 Introduction to digital video processing

The growth in digital signal processing technology, which started off with the theory of basic digital filter design, has now experienced an explosion in DSP applications spurred by significant advances in digital computer technology and integrated-circuit fabrication. Digital image processing being a subarea of DSP has entered into many aspects of modern technology. Since the early days, digital image sequence processing has been an attractive research area because an image sequence contains more information than a single image frame. The direct result of this growth, advancement, and application is the increased amount of digital data being generated and calls for the development of effective techniques to handle this huge amount of data. Modern data compression techniques has made it possible for the storage or transmission of vast amount of data to efficiently represent images and videos.

The digital image and video technology, today, finds applications in a variety of places from health industry to entertainment industry to telecommunications. Applications like videoconferencing over mobile devices, HDTV broadcast, video content generation and distribution has become an integral part of the modern life. With the advent of smart phones and other mobile devices, video bandwidth requirement becomes important. To make this real time video communications efficient, many different algorithms, compression techniques are being developed. To compress a video, it has to be encoded at the transmitter and decoded at the receiver. Video compression mainly involves the exploitation of redundancy present in the video. Lossless compression reduces only the statistical redundancy and there is no loss of information. The compression achieved by this method is less than lossy compression and the modern algorithms employ lossy compression methods to achieve higher efficiency. Video compression algorithms strive for efficient compression and low distortion. There is trade-off between video quality, computational complexity and the cost of hardware. Video data can be represented as a sequence of still images also known as frames. The sequence of frames contains spatial and temporal redundancy that the algorithms attempt to reduce to a smaller size. The information present in a frame is mostly the same across the next or previous frames, and there is low information change between the frames. One way of compression is to obtain the new information which does not exist in the previous frames and store and transmit only that new information and the amount of frame data can be reduce to a greater extent. The encoder is responsible for the compression by applying different algorithms to the input video sequence thereby creating a bit stream that can be either stored or transmitted. At the decoder, inverse algorithms are applied to get back the original video sequence. Note that the quality of the decoded video may not be as good as the transmitted video due to loss of information during the compression process. The combined encoder and decoder are called as a video codec. Video compression is a highly complex process and standards like MPEG-2 [1], H.264/AVC [1] [2] [3] make it possible. Video compression generally involves the encoding of the first frame. Then this frame is compared to subsequent frames and the difference between the first frame and the next frame in the video sequence is encoded. This reduces the amount of information stored and sent. This process of representing the transformation of a reference frame to the current frame is termed motion compensation. The reference picture could be a previous frame or future one within the video sequence. This is illustrated in Figure 1.1

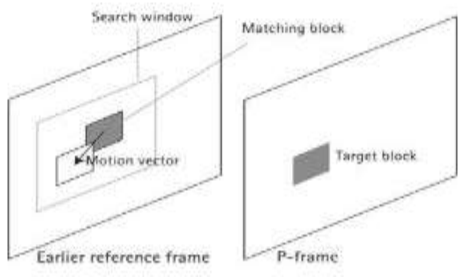


Figure 1.1 Illustration of block based motion-compensation

The H.264 video coding standard is the most widely used coding standard and was developed by ITU-T (International Telecom Union-Telecom Standardization) Video Coding Experts Group (VCEG) along with ISO/IEC Moving Picture Experts Group (MPEG) [4] [5]. H.264 provides better compression than any other standard and comes with a host of different profiles and settings suitable for different applications. The standard has many built-in features that make it complex requiring greater processing power. To have this standard take its place in practical applications, complexity reduction has to be achieved without sacrificing the coding efficiency [6] [7] [8].

1.2 Thesis outline

Chapter 2 provides the introduction to video quality and formats along with a high level description of H.264 / AVC Standard. The different tools, their configuration and profiles along with their applications are discussed briefly.

Chapter 3 introduces the concepts of parallel programming. It starts off with classification of parallel computers and then goes onto to explain different parallel programming models and the criteria before developing any parallel program.

Chapter 4 discusses the basic NVIDIA GPU (Graphics Processing Unit) Architecture and CUDA (Compute Unified Device Model) Programming Model. It also discusses the different built-in functions that CUDA provides and the ways of implementing CUDA programs.

Chapter 5 discusses all the implementation details and algorithms that have been used in the research work. It also discusses the experiments and the results. Here the analysis of the results and a comparative analysis between different standard reference models like JM 16.0 [32] is provided.

Chapter 6 outlines the conclusions and the further possible research. The configuration files used by the JM 16.0 H.264 encoder for the generation of the bit streams are also given.

1.3 Concluding remarks

This chapter provides an introduction into the area of digital image and video processing. It starts out by explaining how DSP evolved and applications of DSP. Later it explains why and how digital video processing and coding is done. This chapter focuses on the most widely used video coding standard, H.264 / AVC. All the internal modules of H.264 video codec are explained in detail.

CHAPTER 2

OVERVIEW OF THE H.264 / AVC VIDEO CODING STANDARD

2.1 Video quality and formats

A ‘real world’ or a natural video scene most of the time consists of objects each of which has their own characteristic dimensions, texture and illumination. The basic characteristics of a typical video scene relevant to video processing are the spatial characteristics such as color, shape of the objects, texture variation and temporal characteristics like changes in the illumination, motion of objects. The natural video scene is spatially and temporally continuous. To represent the video scene, the frames have to be sampled spatially and temporally as still frames or part of the frames at regular intervals. Each element or sample termed as picture element or pixel is represented as one or more values that indicate the brightness or luminance and color or the chrominance. The sampling in the spatial and temporal domain is shown in Figure 2.1.

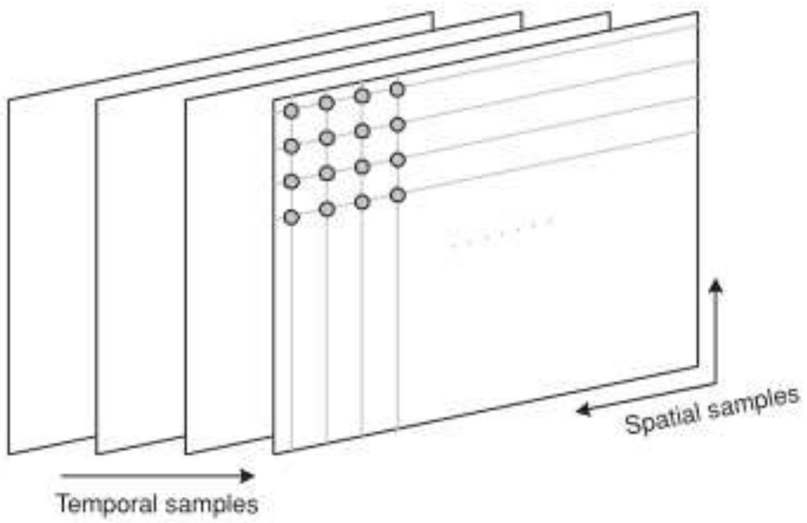


Figure 2.1 Spatial and temporal sampling

2.1.1 Color spaces

Most digital video applications rely on the display of color video and so need a mechanism to capture and represent color information. Color images need three numbers per pixel to represent the color accurately. This method of presenting the color and brightness is described as color space. In the RGB (Red, Green and Blue) color space, the color image pixel is represented with three numbers that describe the relative proportions of the Red, Green and Blue, the primary colors. Any color can be created by varying the proportions of these three colors. The human visual system is less sensitive to color or chrominance than it is to luminance. But in the RGB color space, all the three colors are equally significant and stored at same resolution. To represent the color image more efficiently according to the human visual system, the luminance component has to be separated from the chrominance component and the luminance is at a higher resolution than chrominance. The Y: Cr: Cb color space is a way of efficiently representing the color images. Y is the luminance (luma) component and can be calculated as the weighted average of R,G and B :

$$Y = k_r R + k_g + k_b B \qquad (2.1)$$

where *k* are the weighting factors. The color information can be represented as the difference components, where each chroma component is the difference between R, G or B and the luma Y:

$$\begin{aligned} C_r &= R - Y \\ C_b &= B - Y \qquad (2.2) \\ C_g &= G - Y \end{aligned}$$

To completely describe a color image, Y, the luma component, and three color differences Cr, Cb and Cg that represent the difference between the color intensity and the mean luminance of each image sample.

An RGB image may be converted to Y: Cr: Cb after capture in order to reduce the storage requirement and before displaying the image it is necessary to convert back to RGB. Equations in 2.3 and 2.4 can be used to convert Y: Cr: Cb to RGB and RGB to Y: Cr: Cb.

$$\begin{aligned} Y &= 0.299R + 0.587G + 0.114B \\ Cb &= 0.564(B - Y) \\ Cr &= 0.713(R - Y) \end{aligned} \tag{2.3}$$

$$\begin{aligned} R &= Y + 1.402Cr \\ G &= Y - 0.344Cb - 0.714Cr \\ B &= Y + 1.772Cb \end{aligned} \tag{2.4}$$

2.1.2 YCrCb sampling formats

There are three sampling formats supported by coding standards like H.264/AVC. Starting with 4:4:4, in this sampling format all three components (Y: Cr: Cb) have the same resolution and a sample of each component exists at each pixel position. In the 4:2:2 sampling format, also referred to as YUY2, the chrominance components have the same vertical resolution as the luma, but it has half the horizontal resolution. This means that for every 4 luminance samples in the horizontal direction there are 2 Cr and 2 Cb samples. In the 4:2:0 sampling format, also referred to as YV12, the chroma components have half the vertical and horizontal resolution compared to luminance components.

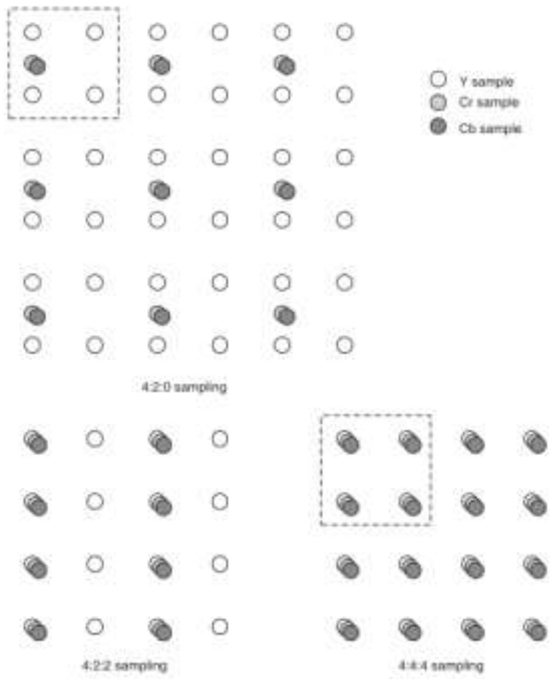


Figure 2.2 YCrCb sampling formats

2.1.3 Video quality

Video quality is subjective, influenced by many factors making it difficult to obtain a common metric for the quality. Video quality, if measured objectively, typically gives an accurate measure of quality, but these metrics do not reproduce completely the subjective experience a human observer gets on watching the image. Generally PSNR (Peak Signal to Noise Ratio) and SSIM (Structural Similarity Index) are the used as the objective metrics.

2.1.3.1 PSNR

Peak signal to noise ratio (PSNR) is the widely used objective quality metric. This is measured on a logarithmic scale and also depends on mean squared error (MSE) between the original and the impaired image or a video frame. Equation (2.5) formulates the calculation of PSNR.

$$\mathrm{PSNR}_{\mathrm{dB}} = 10 \log_{10} ((2^n - 1)^2 / \mathrm{MSE}) \qquad (2.5)$$

2.1.3.2 SSIM

The Structural Similarity Index (SSIM) [12] is one of the most popular methods used to assess the quality of an image. This metric actually compares the structural similarity between the original and the decoded image or a video frame and generally ranges between 0 and 1.

2.2 Introduction to H.264 video coding standard

The H.264 Advanced Video Coding standard is block based motion compensation based video codec standard developed by ITU-T VCEG and ISO/IEC MPEG. The main objective behind the development of H.264 was to have a video codec that is capable of having higher compression rates, high quality video, provide error resilience, and can be supported by the modern communication networks. H.264 provides nearly 50% bitrate reduction when compared to the previous standards like MPEG-2 and MPEG-4. It provides a high quality video at various levels of bitrates. H.264 comes with robust tools that provide error resilience in case of packet loss in networks. H.264 can be used to send the encoded bit stream across different networks thereby making it network friendly. Due to these main reasons it finds applications in areas like: Cable TV broadcasting, Video on demand services over networks, Video communications over mobile devices, Video content generation and distribution. The basic block diagram of H.264 video codec is shown in Figure 2.3.

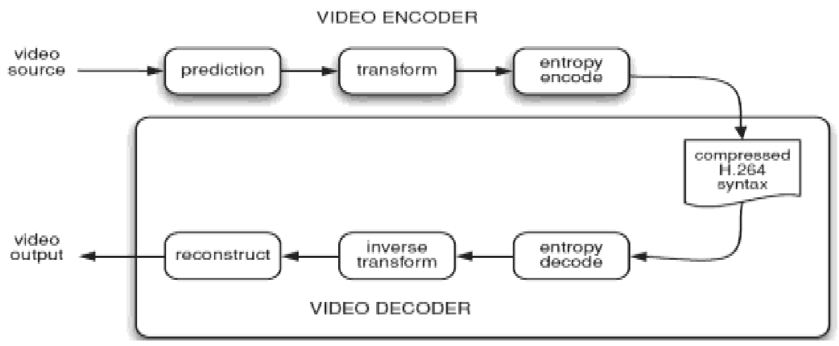


Figure 2.3 Basic block diagram of H.264 video codec [9]

From Figure 2.3, H.264 codec mainly consists of a prediction block where spatial and temporal predictions are performed to exploit the redundancy between the frames. The process of temporal prediction consists of motion estimation and compensation. After the predictions, transform operation is done on the residual data. This transformed data is then quantized to compress and confine the data. After the quantization, the entropy encoding is done to further compress the data and reduce the statistical correlation. Finally a bit stream is produced which the decoder can use up to decode and produce the video sequence. H.264 provides a format or syntax for representing the compressed video and information related to it. The H.264 syntax is made up of series of packets or also called as Network Adaptation Layer Units (NAL Units). These packets contain the parameters that are used by the decoder to correctly decode the video data and slices, which are the coded video frames or can be parts of video frames. The NAL Units are be classified into two as VCL (video coding layer) and non-VCL units. The VCL units contain the data representing the values of the samples in the video pictures and the non-VCL NAL units contain data that represents the information that enables decoder with the timing information, header data and data that would make decoder process more efficiently. As mentioned earlier, H.264 is a block based video coding standard. This means that the entire picture frame can be divided into fixed size macroblocks ranging from 16x16 to 4x4. The macroblocks are organized in slices representing a subset of a given picture that can be decoded independently. H.264 defines five different types of slices : I, P, B, SI and SP [9].

- Intra (I) slice- Describes a full still image containing reference only to itself. If an I slice is to be coded then reference is made to this I slice itself and no other frames in the video sequence are considered.
- Predictive (P) slice- Describes the slices that use one or more recently decoded slices as reference. P frames usually require fewer bits than I frames but they are very sensitive to transmission errors.
- Bi-predictive (B) slice – Describes the slices that are similar to P slices and differ in a way that a previous and/or future I or P slices can be used for prediction. The use of B frames increases latency.

- SI and SP slices – Describes the switching slices used for transitions between two different H.264 video streams.

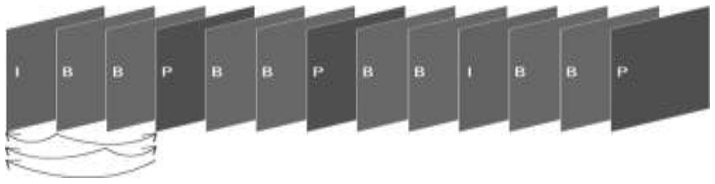


Figure 2.4 I, P and B frames.

The H.264 encoder block diagram is given in Figure 2.4. There are two paths, the forward path and the reconstruction path. The reconstruction path is responsible for the reconstruction of the coded frames which will be used for prediction by the blocks in the forward path.

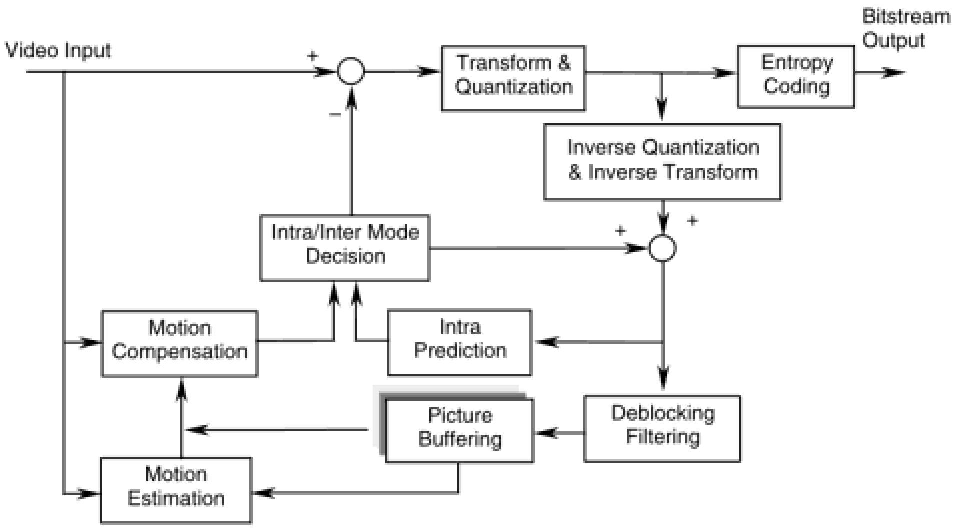


Figure 2.5 The H.264 encoder block diagram [2]

2.2.1 Intra prediction

Intra prediction is a prediction method where only the spatial redundancies are exploited. Intra macroblocks are coded without referring to any data outside the current slice. Typically there is a high relative correlation between the samples in the block and the samples that are immediate adjacent to the considered block. Intra prediction uses samples from adjacent macroblock which are already coded to predict the values in the current block. For an intra macroblock, for the luma component the sizes can be 16 x16, 8 x 8 or 4 x 4. A single prediction block is generated for each chroma component. Based on the intra macroblock sizes, there are various possible prediction modes. For a 16 x 16 luma macroblock, there are 4 possible prediction modes. For 8 x 8 and 4 x 4, there are nine possible prediction modes. For chroma macroblock there are four possible prediction modes.

When a block size is chosen for the luma component, the intra prediction is created from the samples that are above or to the left of the current macroblock or a combination of these. These samples that are above and to the left of the current macroblock are already encoded and reconstructed and are available to the encoder and decoder for prediction. The difference between the prediction and the original macroblock is coded resulting in information that is much less than the original values.

To predict a 4 x 4 luma blocks, H.264 offers 9 modes that includes a DC mode and eight directional modes. This is shown in Figure 2.5.

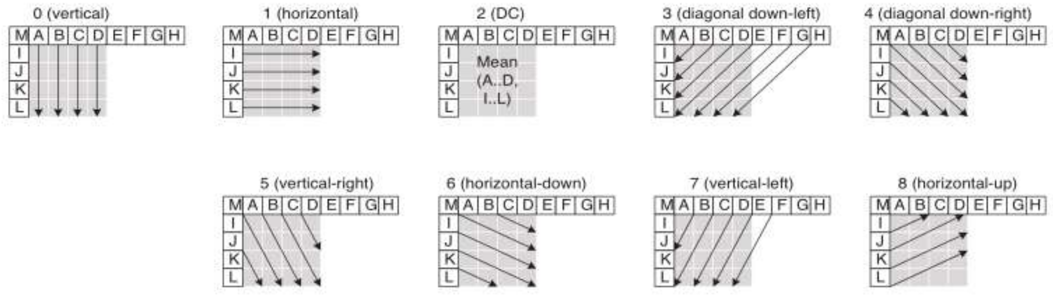


Figure 2.6 Intra prediction modes of 4 x 4 luma macroblocks [1]

According the Figure 2.5, the samples from A to M are the neighboring samples which have been encoded and reconstructed are used for the prediction of other samples within the macroblock.

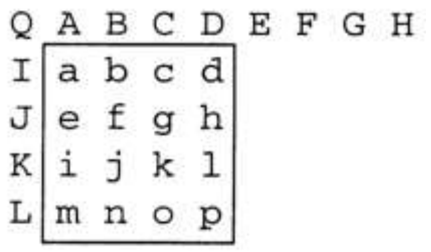


Figure 2.7 Block with samples from a till p are to be predicted [1]

Consider Figure 2.6 where in the macroblock with pixel values a till p are to be predicted using the pixels above i.e. Q to H and to the left i.e. Q to L are to be used. Based on the mode selected, the values of the pixels are predicted. If Mode 0 is selected then the pixel values a, e, i and m are equal to A, the pixels b, f, j and n are equal to B and the same order continues for the other pixels in the block.

For 16 x 16 luma blocks, for intra prediction there are 4 different modes as show in Figure 2.7.

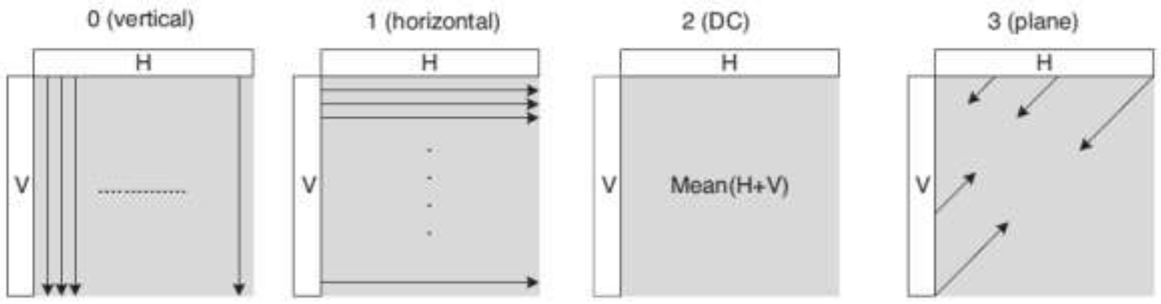


Figure 2.8 Prediction blocks for 16 x 16 block size[1]

The H.264 coding standard offers 4 prediction modes for a intra prediction of 16 x 16 luma blocks including DC mode, horizontal, vertical and planar mode. Also H.264 offers 4 prediction modes for 8 x 8 chroma blocks.

2.2.2 Inter prediction

Inter prediction is another technique that H.264 uses to efficiently code the video. There are two processes: motion estimation and motion compensation that takes the advantage of the temporal redundancies that are present between the successive frames. Temporal prediction typically involves the prediction of a frame by referring another frame in future or a past frame and known as the reference frames. The whole prediction process involves selection of a prediction region and generating a prediction block and subtracting this from the original block of samples to form a residual that is then coded and transmitted.

Motion compensation is a process that is performed to compensate for the motion in the rectangular blocks of the current frame. The process involves the following steps:

- Search for an area in the reference frame and find a matching block M x N region. To do this, a comparison of the M x N block in current frame with some or entire frame of the reference frame and finalizing on the M x N block that best matches the block in current frame. To do the comparisons, the absolute difference between the each pixel value in the original block and the corresponding pixel in the block in the reference frame used for comparison. The region that provides the lowest difference value is chosen as the best match. This process is known as the calculation of SAD or sum of absolute differences. The entire process of finding the matching block in the reference frame by using the SAD computation is known as the motion estimation.
- The candidate region that was chosen as the best matching region becomes the predictor for the current M x N block and is subtracted from the current M x N block. A residual block is obtained.
- The residual block so obtained is further transformed, quantized and then entropy encoded and transmitted. Also transmitted is the motion vector (MV)

which is has the offset value between the current block and position of the candidate region.

Each 16 x 16 P or B frame may be predicted after having them partitioned into sub blocks and there can be four types of different sizes of sub blocks, 16x16, 16x8, 8x16, 8x8. Again each 8x8 partition can be divided into 4 partitions, 8x8, 8x4, 4x8 and 4x4. This is shown in Figure 2.8. Generally, a large block size is appropriate for the areas that are homogeneous in a frame and smaller block sizes are appropriate for detailed areas.

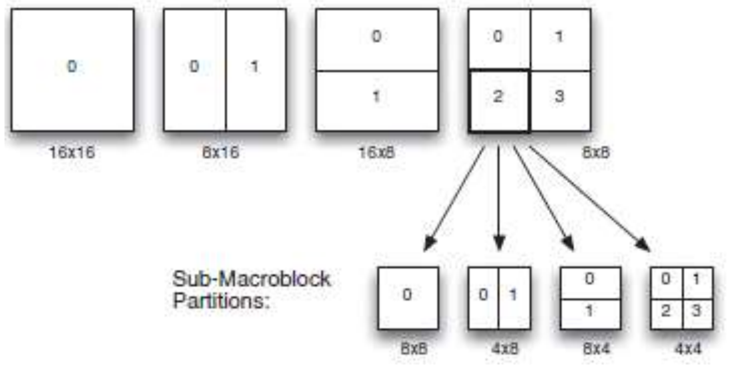


Figure 2.9 Macroblock partition sizes for inter prediction [1]

Motion compensation for a 16 x 16 block in H.264 is done for different block sizes as shown in Figure 2.8. Motion vectors are calculated for every sub block that is used. As the block sizes can be as small as 4 x 4 in a 16 x 16 macroblock, there could be 16 motion vectors transmitted for a single macroblock. As the number of motion vectors are more when a 4 x 4 block is used, better prediction can be achieved. The small blocks improve the ability to handle fine motion details.

2.2.2.1 Interpolating reference pictures

Each partition in an inter-coded macroblock is predicted from an area same in size in a reference picture. The offset, motion vector, has quarter-pixel resolution for the luma component and 1/8-pixel resolution for the chroma components. The sub-pixel prediction improves the prediction performance and produces a smaller residual. In the luma component, the sub-pixel samples at half-pixel positions are generated first and then interpolated from neighboring integer samples using a 6-tap filter with weights (1, -5, 20, 20, -5, 1)/32. After all the half-pixel samples are obtained, each quarter-pixel sample is obtained by bilinear interpolation between neighboring half or integer samples. This is shown in Figure 2.9.

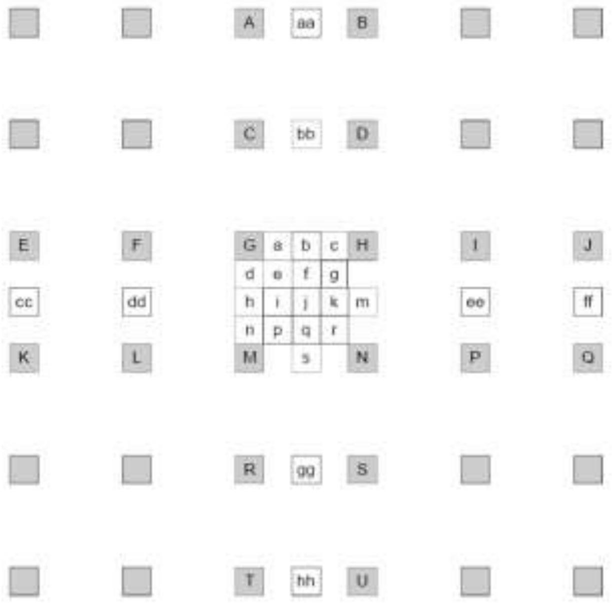


Figure 2.10 Half and quarter pel interpolations [9]

2.2.2.2 Skipped mode

P slice macroblock can also be coded in a mode known as the Skipped Mode. If a macroblock has characteristics that allow its motion to be predicted from neighboring macroblocks, then it can be skipped [11] [9]. If this mode is selected then no information is transmitted for this macroblock. The reconstructed signal is computed similar to the prediction of a macroblock with partition size 16x16.

2.2.2.3 Weighted prediction

Weighted prediction involves scaling of the samples of the motion-compensated prediction data in a P or B slice macroblock. There are three types of weighted prediction in H.264 : P slice macroblock with explicit weighted prediction, B slice macroblock with explicit weighted prediction and B slice macroblock with implicit weighted prediction. Each prediction sample is scaled by a weighting factor before the motion compensated prediction. In the explicit weighted prediction, weighting factor(s) are determined by the encoder and transmitted in the header associated for the slice. In the implicit weighted prediction, the weights are determined based on the temporal positions of the reference frames. If the reference frame is temporally close to the current frame then a larger weighting factor is applied. A smaller factor is applied if the reference frame is temporally further away from the current frame. Weighted prediction provides a means to control the relative contributions of reference frames to the motion compensated prediction process.

2.2.2.4 Motion vector prediction

Encoding motion vectors for each partition can sometimes be an expensive task, especially when small partition sizes are chosen. Motion vectors for neighboring partitions are highly correlated most of the times and so each motion vector can be predicted from the neighboring motion vectors which have been previously coded. A predicted vector MVp is formed based on previously obtained motion vectors and MVD, the difference between the current vector and the predicted vector is then encoded. To illustrate this consider E as a current macroblock, macroblock partition or sub macroblock partition. A is a partition or a sub macroblock partition immediately left of E and B is a partition or a sub macroblock partition immediately above E. C is a partition or sub macroblock partition above and to the right of E. If there is more than one partition immediately to the left of E, then the topmost of these partitions is chosen as A. If there is more than one partition immediately above E, then the left most of these partitions is chosen as B. This is shown in Figure 2.11.

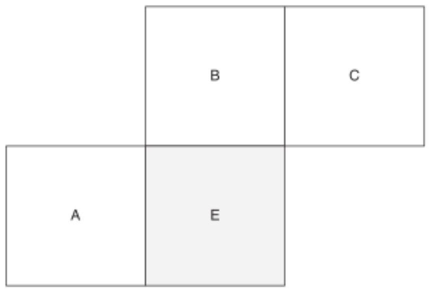


Figure 2.11 Current and neighboring partitions of same size

For partition sizes that are different from the current partition size of E, the choice of prediction partitions is done based on the following and is illustrated in Figure 2.11.

- For partitions excluding 16 x 8 and 8 x 16 partition sizes, MVp is the media of the motion vectors for A, B and C partitions.
- For 16 x 8 partitions, MVp for the upper 16 x 8 partition is predicted from B and MVp for the lower 16 x 8 partition is predicted from A.
- For 8 x 16 partitions, MVp for the left 8 x 16 paritions is predicted from A and MVp for the right 8 x 16 partition is predicted from C.

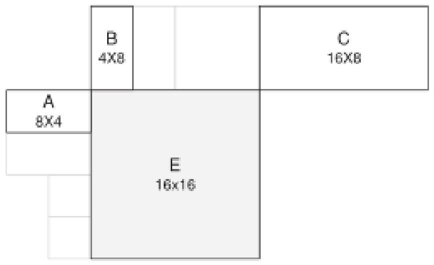


Figure 2.12 Current and neighboring partitions of different sizes

- For skipped blocks, a 16 x 16 vector MVp is generated as if the block were encoded in 16 x 16 Inter mode.

A bipredicted macroblock in B slice has two motion vectors, one for the past frame and one for the future frame. The two motion vectors are predicted from neighboring motion vectors that have same temporal direction. At the decoder, the MVp is formed in the same way and then added to the MVD, the decoded vector difference. For skipped macroblock, there is no decoded vector and motion compensated macroblock is obtained using the MVp.

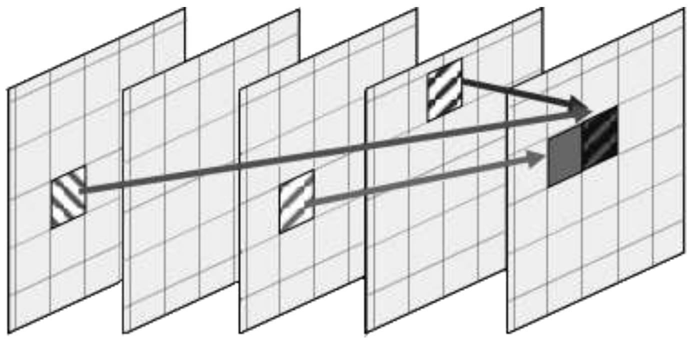


Figure 2.13 Motion compensated prediction with multiple reference frames

2.2.3 Integer transform, Scaling and Quantization

After the intra and inter prediction, the prediction residual obtained is split into 4 x 4 block or 8 x 8 blocks. These blocks are later converted to transform domain and then quantized. H.264 uses adaptive transform block sizes of 4 x 4 and 8 x 8. The smaller block size helps in reducing the blocking artifacts. The basic transform or ‘core transform’ is a 4 x 4 or 8 x 8 integer transform, a scaled approximation to the DCT [13]. An additional M x 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 16 x 16 intra prediction as well as in case of chroma components. For these additional transform stages, separable combinations of the four-tap Hadamard transform and two-tap Haar/Hadamard transform are applied.

The H.264 uses uniform quantizers to quantize the transform coefficients. Quantization is another process where a significant amount of data compression is achieved. One of the 52 quantizer step scaling factors is selected for each macroblock by a quantization parameter. The fidelity of the chroma coefficients is improved by using finer quantization step sizes compared to those used for luma coefficients, particularly when the luminance coefficients are coarsely quantized.

The quantized transform coefficients correspond to different frequencies with the top left coefficient being the DC value. These coefficients are to be arranged in an array starting with the

DC component. A single coefficient scanning pattern is available in H.264 for frame coding and is done in a zigzag order as illustrated in Figure 2.14. The scan order is intended to group together significant coefficients, i.e. non-zero quantized coefficients.

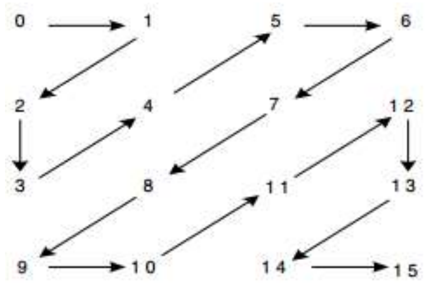


Figure 2.14 Zigzag scan order in a 4 x 4 block [2]

2.2.4 Deblocking filter

The deblocking filter is used to remove blocking artifacts due to the block based encoding pattern. This filter is applied after the inverse transform in the encoder before reconstructing the macroblock and storing it in the decoded picture buffer for future predictions and in the decoder before reconstructing and displaying the macroblock. The filter smoothens the block edges thereby improving the appearance of the decoded frames. This filtered image is later used for motion compensation of future frames and improves the compression performance. The filtering is applied to vertical or horizontal edges of 4 x 4 blocks in a macroblock excluding edges on slice boundaries. The luma deblocking filter process is done on four 16-sample edges and the chroma deblocking filter process is performed on two 8-sample edges. The boundaries that are to be filtered is shown in Figure 2.15

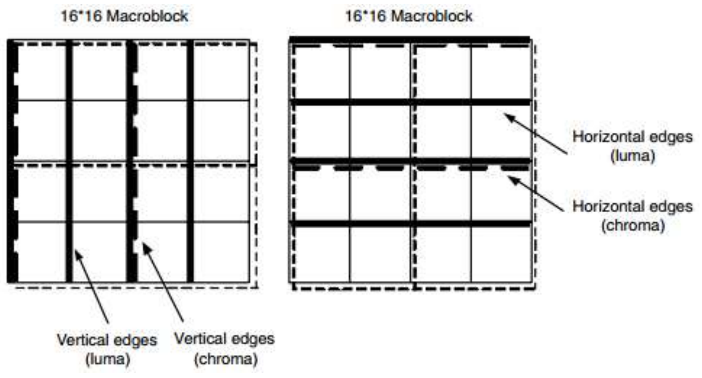


Figure 2.15 Boundaries in a macroblock to be filtered [2]

The deblocking process is done at three levels [2]:

- At slice level : global filtering strength is adjusted to the individual characteristics of the video sequence
- At block-edge level : filtering strength is made dependent on the inter and/or intra prediction decision, motion differences and the presence of coded residuals in the two participating blocks
- At sample level: sample values and the quantizer-dependent thresholds can turn off filtering for each individual sample.

The filtering process not only reduces the blocking artifacts but also reduces the bitrate while producing the same objective quality as the non-filtered video.

2.2.5 Entropy coding [1] [2]

Entropy coding is the last step in the video coding process. Entropy coding is based on assigning codewords shorter in length to the symbols that occur more frequently and longer codewords for symbols occurring less frequently. H.264 standard specifies two types of entropy coding. The first method is the context adaptive variable length coding (CAVLC) and the second method is the context based adaptive binary arithmetic coding (CABAC).

In H.264, many syntax elements are coded using the highly structured infinite-extent variable length code called zero-order exponential Golomb code. A few syntax elements are also coded using simple fixed length code representations. 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 conditional probabilities of the context, the entropy coding performance is improved from that of schemes that do not use context based adaptivity.

By using CABAC, the entropy coding performance is further improved. It is basically based on three components: binarization, context modeling and the binary arithmetic coding. This is shown Figure 2.13.

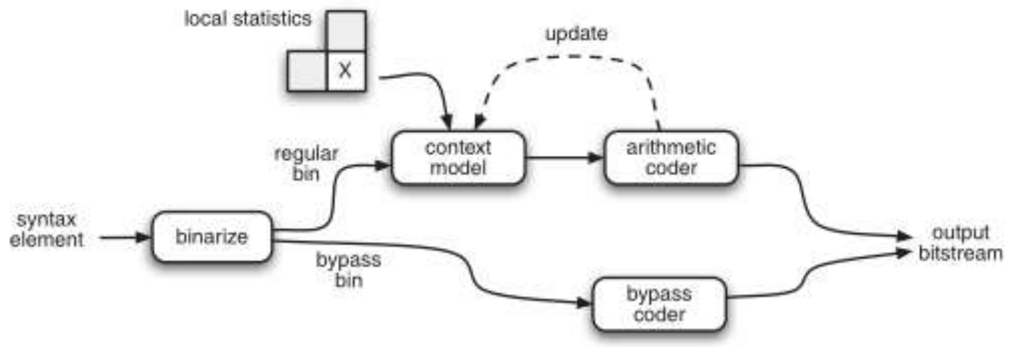


Figure 2.16 CABAC coding process overview.

The binarization enables efficient binary arithmetic coding by mapping non-binary 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. Compared to CAVLC, CABAC provide bit reductions upto 10-20 percent for the same objective video quality.

The characteristics of each coding method is explained below [14] :

1. Context adaptive variable length coding
   i. No end block, but number of coefficients is decoded
   ii. Coefficients are scanned backwards and contexts are built depending on transform coefficient
   iii. Transform coefficients are coded with the following elements: number of non-zero coefficients, levels and signs for all non-zero coefficients, total number of

zeros before the last non-zero coefficient, and number of successive zeros preceding the last non-zero coefficient.

2. Context adaptive binary arithmetic coding
   i. Exploits symbol correlations by using contexts
   ii. Probability estimation is realized by the look up table
   iii. Use of adaptive probability models for most symbols

2.2.6 H.264 profiles [1] [3]

The H.264 standard provides different profiles and levels that specify conformance points and provide interoperability between encoder and decoder implementations within applications of the standard and between various applications that have similar functional requirements. A profile define a set of syntax features generating conforming bitstreams, whereas level places constraints on certain key parameters of the bitstream such as maximum bit rate and maximum picture size. There are three profiles in the first version ofH.264: Baseline, Main and Extended. Also the fidelity range extensions include four additional high profiles for applications such as content distribution and studio editing : High, High 10, High 4:2:2 and High 4:4:4.

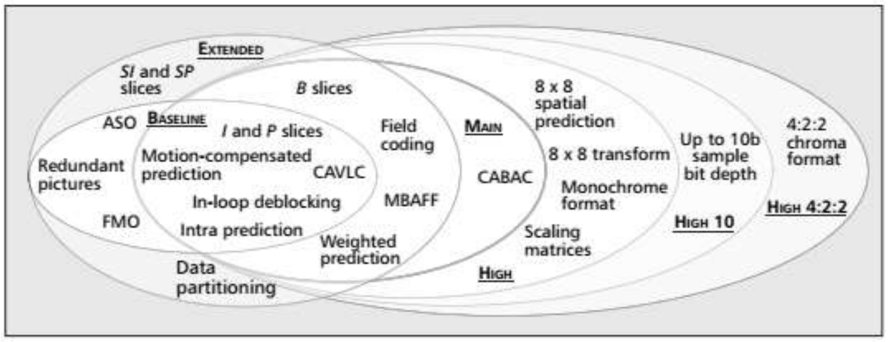


Figure 2.17 Profiles and levels in H.264 [1][2]

The High profile supports 8 bit video with 4: 2:0 sampling for applications using high resolution. The High 10 profile supports 4:2:0 sampling with video that can be represented with 10 bit depth. The High 4:2:2 profile supports 4:2:2 chroma sampling and up to 10 bits per sample. The High 4:4:4 profile to support up to 4:4:4 chroma sampling, up to 12 bits per sample. Figure

2.14 discusses the different features present in different profiles. The Figure 2.14 also shows the profiles that have common coding parts and specific coding parts. Some of the common parts are:

- I Slice: Intra-coded slice, predicted from the values present within the same frame
- P slice: Predictive coded slice, inter-prediction from previously decoded reference pictures, using at most one motion vector and reference index to predict the sample values of each block
- CAVLC for entropy coding

2.2.6.1 Baseline profile [1] [3]

The main features of this profile are :

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

2.2.6.2 Main profile [1] [3]

The main features of this profile are:

- B slice: the coded slice by using inter-prediction from previously decoded reference pictures, using at most two motion vector and reference indices to predict the sample values of each block.

- Weighted prediction: scaling operation by applying a weighting factor to the samples of motion-compensated prediction data in P or B slice.
- CABAC for entropy coding

2.2.6.3 Extended profile [1] [3]

The main features of this profile are :

- Includes all parts of Baseline profile: flexible macroblock order arbitrary slice order, redundant slice
- SI slice: coded slice for efficient switching between video streams, similar to coding of a I slice
- SP slice: the switched slice, similar to coding of an I slice
- Data partition: the coded data is placed in separate data partitions, each partition can be placed in a different layer unit.
- B slice
- Weighted prediction

2.2.6.4 High profile [1] [3]

The main features of this profile are:

- Includes all parts of Main profile: B slice, weighted prediction, CABAC.
- Adaptive transform block size: 4 x 4 or 8 x 8 integer transform for luma samples
- Quantization scaling matrices: different scaling according to specific frequency associated with the transform coefficients in the quantization process to optimize quality.

Table 2.1 lists the different profiles and their applications along with the requirements.

Table 2.1 Profiles and applications

APPLICATION	*H.264 Profile*	*Requirements*
Broadcast television	Main	Coding efficiency, reliability, low complexity, interlace
Streaming video	Extended	Coding efficiency, reliability over a network
Mobile Video	Baseline	Coding efficiency, low latency, low complexity and low power consumption
Video conferencing	Baseline	Coding efficiency, low latency, low complexity encoder and decoder

CHAPTER 3

BASICS OF PARALLEL PROGRAMMING

3.1 Introduction to parallel computing [15]

Parallel computing [15] is a method of computation in which many calculations are carried out simultaneously, based on the notion that large problems can be divided into smaller chunks and then operated on concurrently in parallel. Parallel computing can be carried out in different forms: bit level, instruction level, data level and task level parallel computing. Parallel computing has been used in high performance computing but there has been an increase in the interest and now parallel computing is being used wherever possible.

The basic idea behind parallel computing is to write a parallel program that can perform operations in parallel. The parallel programs are generally complex to develop as one has to take care of different conditions like race conditions, data dependency condition etc. Challenge also lies in synchronization and communication among different subtasks.

Parallel computers can be classified according to the level at which the hardware supports parallelism, with multi-core and multi-processor computers having multiple processing elements within a single machine, massively parallel processors use many processors to work on the same task. Specialized processors are sometimes used alongside traditional processors for accelerating certain class of tasks for specific applications.

3.1.1 Serial programming [15][16]

Typically software written for serial computation has characteristics such as

- To be able to run on a single computer with single central processing unit.
- The main task is broken down into discrete series of instructions.
- Instructions are executed sequentially, one after the other.

- Only one instruction may execute at any instant of time.

A general scenario of serial computation can be illustrated with Figure 3.1 where the task or the problem is solved by dividing into instructions that run serially.

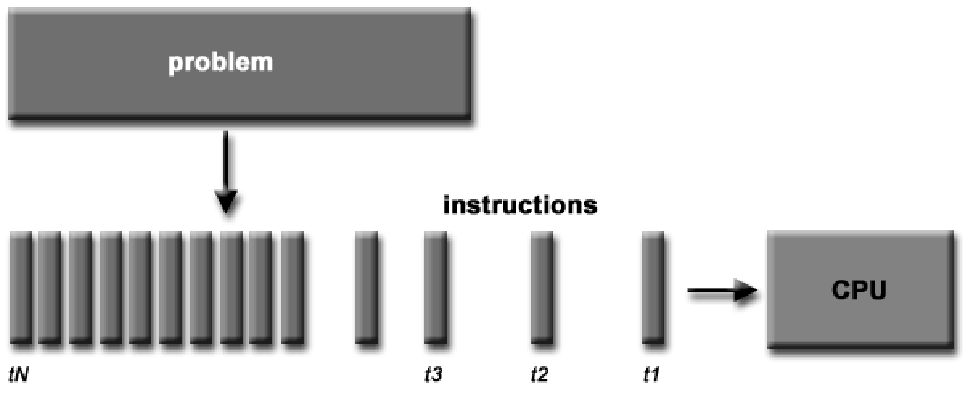


Figure 3.1 Serial execution of a program [15]

3.1.2 Parallel programming

In the simplest way, parallel computing refers to the use of multiple compute resources to solve a computational problem. The problem is broken into discrete parts and these parts are provided to the computing units which solve them concurrently. A general scenario for parallel computation is shown in Figure 3.2 where the problem to be solved is divided into instructions and multiple instructions are executed simultaneously depending upon the number of available cores.

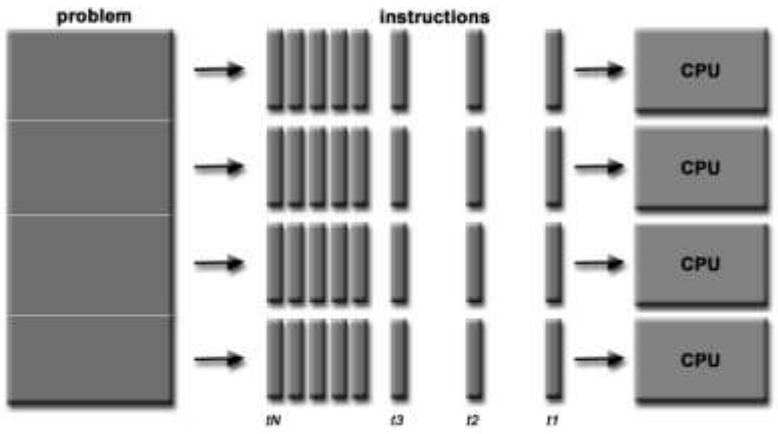


Figure 3.2 Parallel execution of a program [15]

3.1.3 Limitations of serial computing

Physical and practical reasons pose significant limitations for building faster serial computers:

- Transfer speeds – the speed of a serial computer directly depends on how fast data can be moved through hardware. Transmission limit of copper wire is about 9 cm/nanosecond. Increasing speeds necessitate increasing proximity of processing elements.
- Limits to miniaturization - processor technology is allowing an increasing number of transistors to be placed on a chip.
- Economic limitations – it is increasingly expensive to make a single processor faster. Using a larger number of moderately fast commodity processors to achieve the same (or better) performance is less expensive.
- Modern computer architectures rely on hardware level parallelism to improve the performance using multiple execution units, pipelined instructions and multi-core processors.

3.2 Classification of parallel computers [15][16]

There are different ways to classify parallel computers. One of the more widely used classification is called Flynn`s Taxonomy [38]. Flynn`s taxonomy distinguishes multi-processor computer architectures according to how they can be classified along the two independent dimensions of instruction and data. These dimensions can only have two states: Single or Multiple. Flynn`s taxonomy defines 4 types of classifications: 1.Single Instruction, Single data (SISD) 2. Single Instruction Multiple Data (SIMD) 3. Multiple Instructions, Single Data (MISD) 4.Multiple Instruction, Multiple Data (MIMD)

3.2.1 Single Instruction Single Data (SISD)

In SISD computers, only one instruction and a data stream is being operated on a CPU during one clock cycle. It is the oldest and most common type of computers. It differs from other types of parallel computers as this is a serial computer.

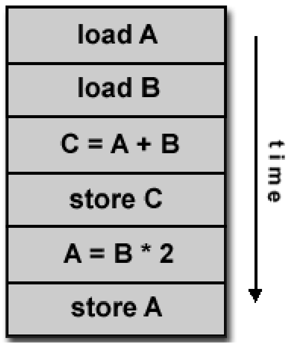


Figure 3.3 Single instruction single data flow [15][16]

3.2.2 Single Instruction Multiple Data (SIMD)

In this type, all processing units execute the same instruction at any given clock cycle. Each unit can operate on a different data element. This type of computing is best suited for problems with high degree of regularity like image processing.

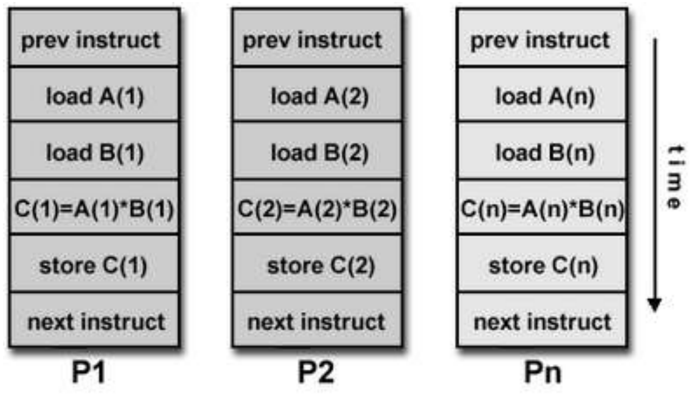


Figure 3.4 Single instruction multiple data flow [15]

3.2.3 Multiple Instruction Single Data (MISD)

In this type of parallel computer, each processing unit operates on a data independently through separate instruction streams. A single data stream is fed into multi-processing units.

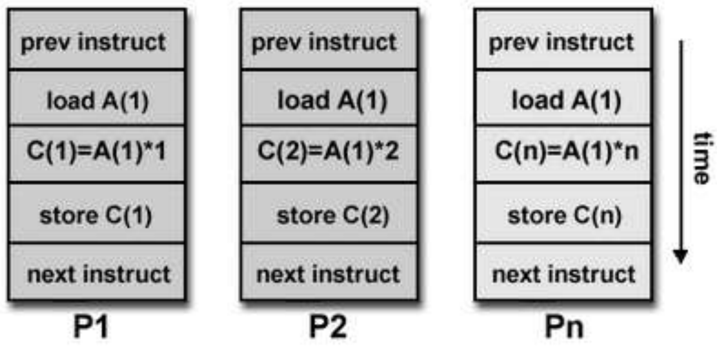


Figure 3.5 Multiple instruction single data flow [15]

3.2.4 Multiple Instruction Multiple Data (MIMD)

In this type of parallel computer, every processing unit may be executing a different set of instructions. Each processing unit may be working with a different data streams. These executions can be synchronous or asynchronous, deterministic or non-deterministic.

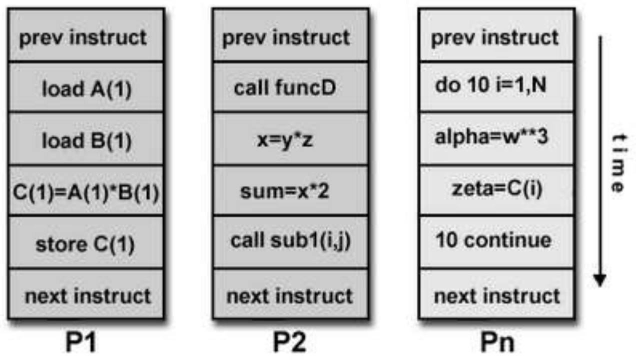


Figure 3.6 Multiple instruction multiple data flow [15]

<u>3.3 Programming models for parallel computing [15][16]</u>

Programming models for parallel computing exist as an abstraction above hardware and memory architectures. There are several parallel programming models in use. These models are not specific to any hardware architecture. Theoretically, any of these models can be implemented on any underlying hardware. Some of the models are explained in the next few sections.

3.3.1 Shared memory

Tasks share a common address space in this programming model. Mechanisms like locks / semaphores are used to control access to the shared memory. An advantage of this model from the programmer`s perspective is that the notion of data “ownership” is lacking, so there is no need to specify explicitly the communication of data between tasks. An important disadvantage in terms of performance is that it becomes difficult to manage the data locality.

3.3.2 Threads

Thread programming model is a type of shared memory programming. In this model, a single process can have multiple execution paths. Threads can be explained using the concept of a single program that includes a number of subroutines. The main program would have several subroutines. A number of threads can be created and assigned the task of processing each subroutine parallelly.

3.3.3 Distributed memory / Message passing

In this model, multiple tasks reside on the same physical machine and/or across arbitrary number of machines. Tasks exchange data through communications by sending and receiving messages. Data transfer usually requires cooperative operations to be performed by each process.

3.3.4 Data parallel model

In this programming model, operations are done on a particular data set. This data set is organized into common structure, such as an array. A set of tasks, work collectively on different partitions of the same data structure. A typical scenario for a data parallel programming model is shown in Figure 3.7.

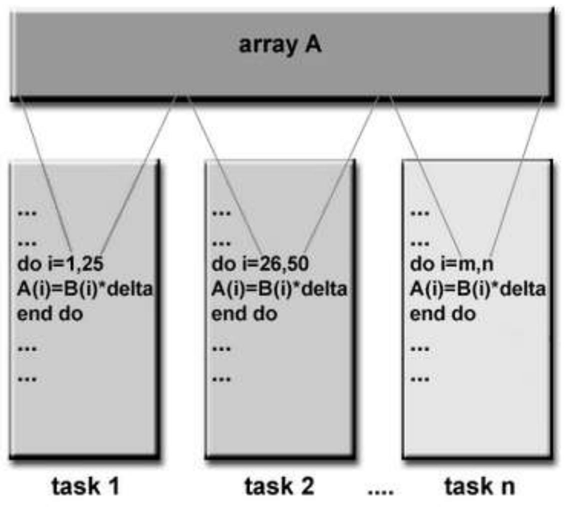


Figure 3.7 Data parallel programming model [15]

3.4 Considerations before developing a parallel program

The very first step in developing parallel software is to understand the problem at hand. Starting with a serial program analysis, one needs to determine which part of the problem can be actually parallelized. Few points that one needs to consider are the program`s hotspots, bottlenecks in the program and the inhibitors to the parallelism.

3.4.1 Program hotspots

The majority of scientific and technical programs usually have most of their work done in few places. Profilers and performance analysis tools can help to identify these critical hot spots. Hence the focus should be on parallelizing these hotspots and ignore the sections of the program that account for less CPU usage.

3.4.2 Bottlenecks

Bottlenecks are the parts and areas that slow down or halt the parallelizable work. For example, waiting on an I/O is something that is going to slow down a program. In such cases, a different algorithm should be used to reduce or eliminate unnecessary slow areas.

3.5 Concluding remarks

This chapter has provided an overview of different parallel programming paradigms, parallel programming models, limitations, inhibitors and hotspots to be considered while parallelizing. Next chapter focuses on describing the NVIDIA GPU Programming Model. It explains, in detail, different modules of CUDA programming model, underlying architecture, different system calls that will be used and environment considerations one needs to take care of before using this model for parallel programming.

CHAPTER 4

NVIDIA GPUS AND CUDA PROGRAMMING MODEL

4.1 GPU computing

Since the beginnings of the last decade, the scientific community has realized the large amounts of computational power imbedded in the Graphics Processing Units (GPU) that has to be used. Driven by its power integrated with a massively-parallel and multithreaded many-core architecture, many areas of science and technology have benefitted by the use of the GPU in addressing their advanced computational problems, which were previously thought of as challenging and not feasible [21] [22].

The GPUs are not well suited to solve all types of problems [23], however there are many kinds of applications that have achieved quite significant speed-up depending on the hardware platform [21][22]. As seen in Figures 4.1 and 4.2, todays` GPUs greatly outperform CPUs in their arithmetic throughput and memory bandwidth and targeting a specific type of applications, which most often has large arithmetic density i.e. with large amounts of mathematical operations for memory access. These applications range from audio processing to image and video processing to bioinformatics and visual computing [23][22].

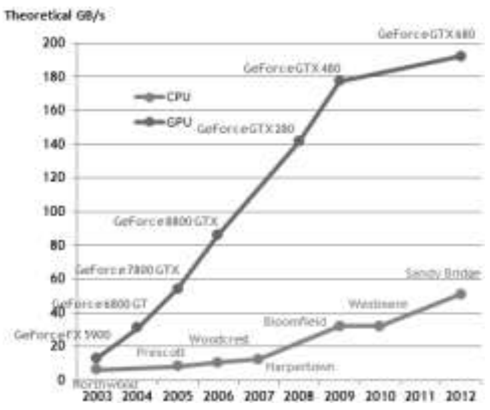


Figure 4.1 Memory bandwidth for the CPU and the GPU [22]

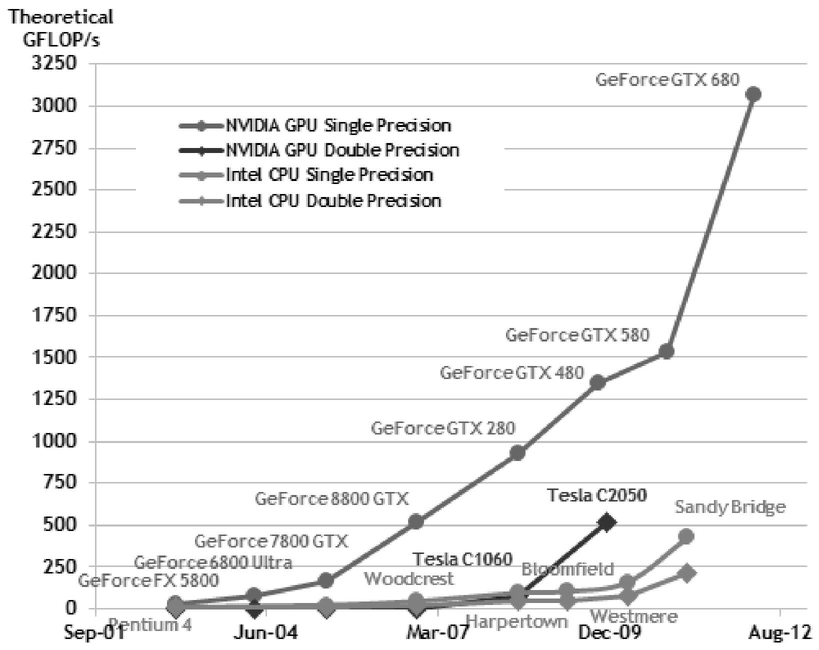


Figure 4.2 Floating point operations per second for the CPU and GPU [22]

4.2 Heterogeneous computing model

To use GPU in solving computational problems, a heterogeneous computing model must be adopted. It basically combines the multi-core CPU along with a many-core GPU to form a complete computing environment [24]. In this model as illustrated in Figure 4.3, the CPU is used for executing the serial portions of the code and to control the flow of the algorithm. The GPU is used to operate on the data parallel portions of the code that would take up significant amounts of time when allowed to be processed by the CPU. The model requires the first step to be the analysis of the problem at hand a priori to identify and separate out the parallel portions from the serial ones and then modifying the flow of the operations in a proper manner to be able to process stage by stage accurately and successfully. The serial portions of the algorithms are written in normal high level programming language functions like C/C++ and the parallel portions are written in GPU kernels. The C functions are executed on the CPU while the parallel kernels are offloaded to the GPU for processing. This flow is illustrated in Figure 4.4.

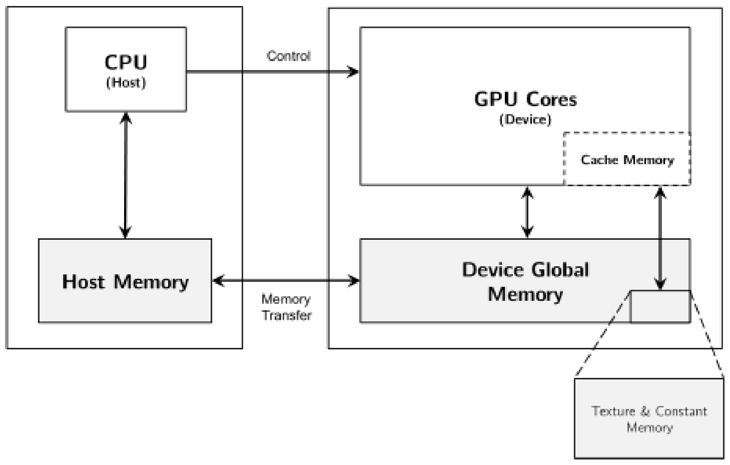


Figure 4.3 Heterogeneous computing model [25]

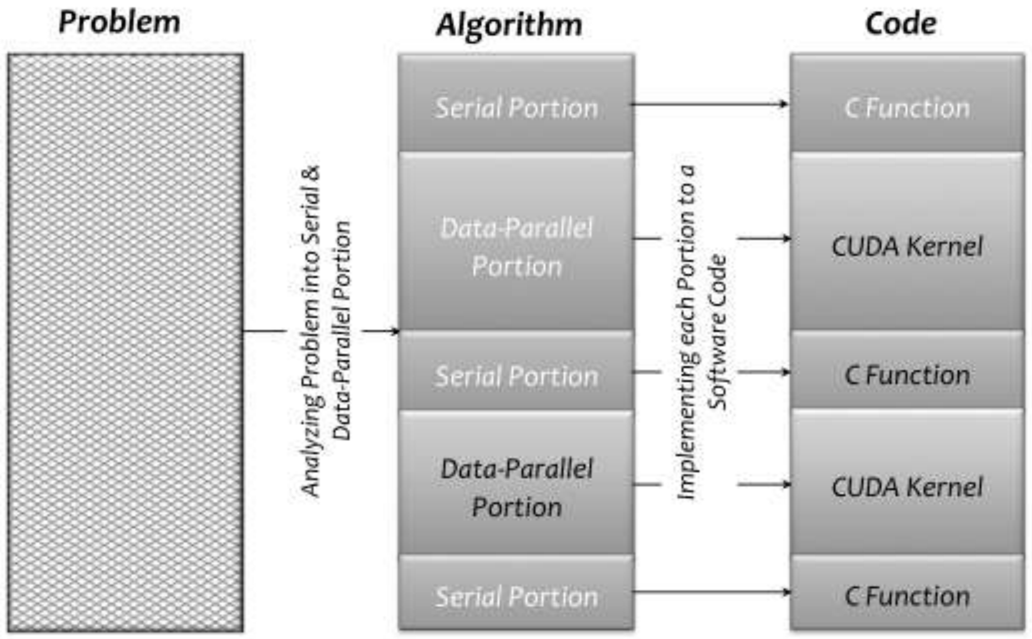


Figure 4.4 Problem decomposition for serial parts to be executed on CPU and parallel parts to be executed on the GPU [25]

4.3 Compute Unified Device Architecture

CUDA was introduced by NVIDIA in 2006 and is the parallel programming framework for general purpose computations. To make applications parallel, CUDA allows developers to program in C/C++ with some extensions and poses a low learning curve for programmers already familiar with these programming languages. The extensions include allocating memory on the GPU, copying data back and forth from GPU to the CPU, making an abstraction for processing data in parallel on the GPU and synchronization and also for communicating between the threads executing in parallel [26].

4.3.1 Programming model

The parallel code, which gets executed on the GPU, is written as a function which is referred to as a kernel. When the kernel is launched, many threads share the same code and they start executing in parallel. All the threads which are launched are referred to as grid and the grid is divided into blocks of threads. A block of threads is a group of threads which is executed about the same time on the GPU. They have a shared memory for communication and fast access to the data between the threads, and it is also possible to synchronize execution between the threads. The number of threads per block and the number of blocks per grid are important decisions when programming using the CUDA framework since it affects the performance.

CUDA provides built-in variables for accessing the thread index and the block index for the thread. These variables can be used in the CUDA kernel code to make each thread operate on different parts of the data. The thread blocks are scheduled on the GPU by the hardware and it is not possible to know when the thread blocks are executed in relation to one another. Hence it is important to organize the threads such that each thread block can execute independent of each other and thus the results of the computations do not depend on the execution order, synchronization or communication between thread blocks as shown in Figure 4.5 [26][27].

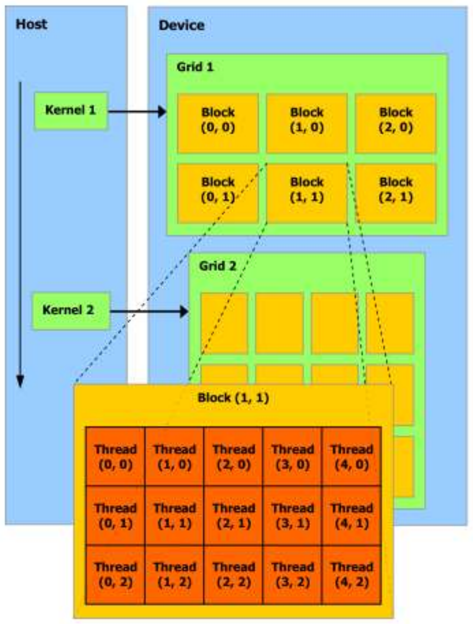


Figure 4.5 Thread grouped into blocks [27]

4.3.2 Scalability

CUDA addresses the situation of increase in the number of cores on the processors by scaling the parallelism of an application. CUDA provides a hierarchy of thread groups as an abstraction to the programmer, and the programmer needs to decompose the problem into sub-problems which can be solved by a block of threads independently from other thread-blocks. This enables automatic scalability when the CUDA program is executed on different GPUs. When the program is executed on more cores, the run time environment schedules more CUDA blocks to be executed concurrently. Each CUDA block can be executed on any core and in any order. Thus, a CUDA program can scale across a wide range of different GPUs.

4.3.3 Memory model

The memory hierarchy of the GPU is similar to the most regular desktop CPUs. There is large chunk of off-chip RAM (Random access memory) called the global memory and small amount of fast on-chip memory known as shared memory. None of this memory is cached [24][25]. This memory model is illustrated is illustrated in Figure 4.6 [27].

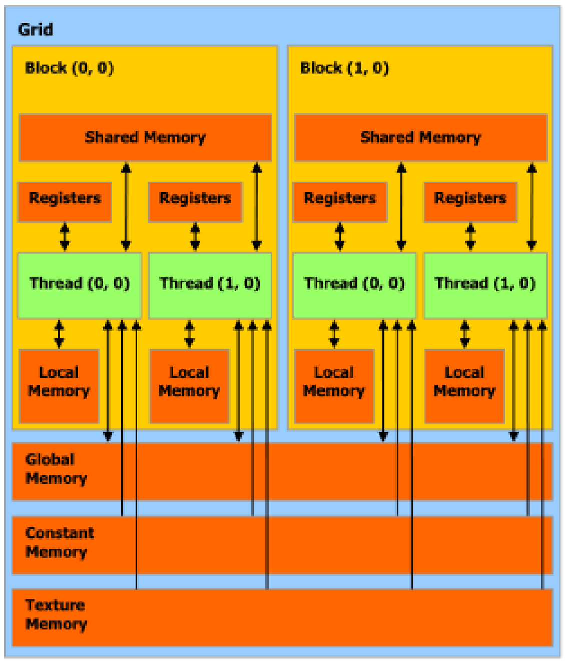


Figure 4.6 CUDA memory hierarchy [27]

4.3.3.1 Global memory

Data which are transferred from the host CPU to the GPU device are first copied to the global memory. This is the memory space on the GPU with the biggest size, and the highest latency. It is accessible by all the threads and thread blocks, and hence useful for storing large amounts of data and data that is needed to be accessed by all the threads in the different thread blocks.

4.3.3.2 Shared memory

Shared memory is on chip fast memory with access latency which can be 100 times lower than the access latency of global memory. The shared memory is used to speed up the computations with fast memory access and for inter-thread communication between threads within a block.

4.3.3.3 Local memory

Local memory is allocated in the global memory and is private for each thread. The local memory is used for arrays and data structures which are too big for registers or when the registers are not available. Local memory has the same latency and memory bandwidth as those of global memory.

4.3.3.4 Constant memory

Constant memory is a small chunk of cache memory residing on each multiprocessor. The main advantage of having this memory is the fast access due to its low latency. Constant memory can be read and written from the host CPU, but it is readable only by the threads running on the device.

4.3.3.5 Texture memory

Texture memory can be read and written from the host, but it is only readable from the device. It is capable of linear interpolation for one, two and three-dimensional arrays stored in the texture cache memory.

4.3.4 Execution model

Threads are scheduled and executed in groups of 32 threads called as a warp. All threads in a warp start at the same address in the CUDA program and are executed concurrently in the hardware. Each thread in a warp has a separate instruction counter and maintains its own register state. Thus it can branch out and execute independently of the other threads in the warp.

4.3.5 Performance considerations

The multiprocessor maintains the state of execution contexts such as program counters and registers for the whole time of a warp. Context switches between different warps of threads can be executed very fast by the hardware with no overhead.

4.3.5.1 Grid size and block size

The size of blocks per grid and size of the threads per block are important factors when programming CUDA. The size of a block can be one, two or three dimensional and the size of the grid can be one or two dimensional. The multidimensional aspects of the blocks and grids are used for allowing more easy mapping of multidimensional data structures to CUDA and do not affect performance.

4.3.5.2 Data transfer between host and the device

The bandwidth between host and device is much lower than the bandwidth between the GPU and the device memory. Hence the data transfer between the host and the device should be minimized. Sometimes it may be preferable to run kernels on the GPU which do not show better performance than running the same computations on the CPU, if it reduces the amount of data transfer between the host and the device [27].

4.3.5.3 Divergent Warps

Threads are executed in groups of 32 threads. This group of threads is referred to as warp and is executed at the same time on the hardware. Performance can be maximized if each warp of threads should follow the same execution path. This is because threads can only run in parallel on the hardware as long as their execution paths do not diverge. Control statements like “if”, “while” or “for” cause the threads within a warp to follow different execution paths. In such situations, the hardware cannot execute the different execution paths in parallel and they need to be serialized. Hence, branching in the code can result in divergent warps which should be avoided at all times [27].

4.4 Concluding remarks

This chapter has introduced the NVIDIA GPU architecture. It explains all the details about the same and its memory hierarchy, execution model and CUDA programming model. Different performance considerations are also explained in this chapter. The next chapter focuses on the experimentation, description of the motion estimation algorithm and the results. Different plots provided in the next chapter show a 50% reduction in the encoding time.

CHAPTER 5

ALGORITHM, IMPLEMENTATION AND RESULTS

5.1 Motion estimation algorithm

The temporal prediction technique used in H.264 is based on motion estimation. The basic idea behind motion estimation is that in most of the cases, neighboring and consecutive video frames are similar except for the changes produced by the objects moving within the frames. The process involves selecting a prediction region, generating a prediction block and subtracting this from the original block of samples to form a residual. The macroblock to be predicted can range in size from a complete macroblock 16 x 16 to 4 x 4 sub-blocks. The main idea is to find a block in the reference frame that closely matches the block in the current frame which is being predicted as shown in Figure 1.1. This is also known as the block matching process. To achieve this, H.264 reference software JM 16.0 [32] uses several algorithms and one such algorithm is the full-search algorithm.

On performing the time profiling, it is found that motion estimation alone takes around 90% of the time and it is the most dominating module within H.264. The intra prediction operation takes up to 2 %, transforms and quantization take up around 5% and rest of the portion like variable length encoding and frame statistics take up close to 3% of the total encoding time. Motion estimation is the most compute intensive part of H.264. This is shown in Figure 5.1

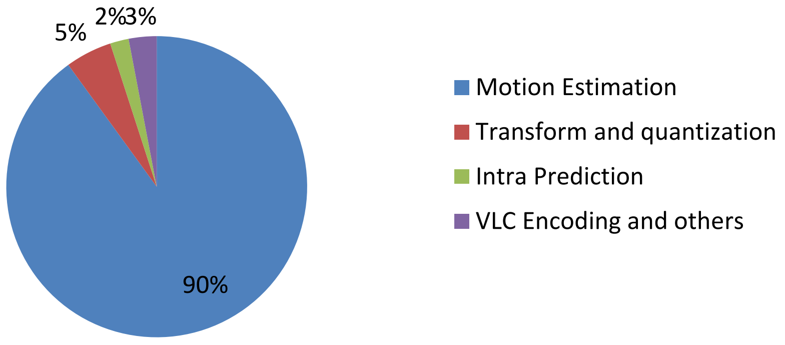


Figure 5.1 Time profiling of H.264 encoding operation

The block matching algorithm used in H.264 divides current frame into equal size blocks also known as the source block. The objective of this algorithm is to find a candidate block in the search region best matched to the source block. The relative displacement between a source block and its candidate blocks are called motion vectors. This process is illustrated in Figure 5.2.

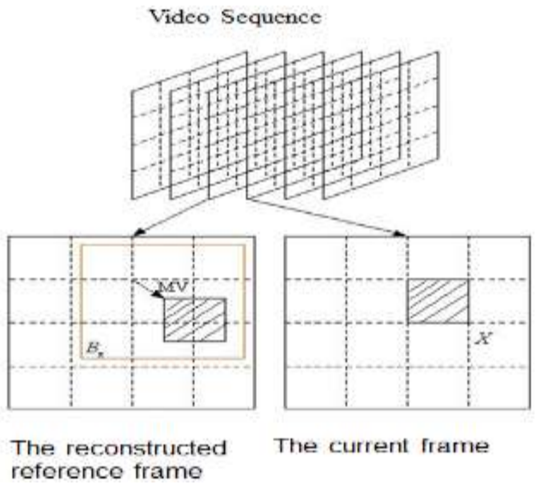


Figure 5.2 Block matching between reference frame and current frame
X: source block for block matching, B_x: search area associated with x, *MV*: motion vector

Because the search is performed for all blocks of the current frame in the reference frames confined within a search range, it is highly compute intensive. In determining which block in the search area of the reference frame best matches for the block in the current frame, the difference between the two blocks is taken and the block that provides lowest difference is chosen. This method is known as the sum of absolute difference (SAD) computation. The equation 5.1 [30] gives all the parameters to calculate SAD. In this equation, I_{k-1} *(m, n)* is the pixel value at *(m, n)* in the reference frame and I_k *(m + dx, n + dy)* is the pixel value at *(m, n)* in the current frame and *dx, dy* are the motion vectors in horizontal and vertical axes, *N* represents the search range (default value is 32),

$$SAD\,(dx,dy) = \sum_{m=x}^{x+N-1} \sum_{n=y}^{y+N-1} | I_k(m,n) - I_{k-1}(m+dx, n+dy) | \qquad (5.1)$$

$$(MV_x, MV_y) = \min SAD(dx, dy) \qquad (5.2)$$

The motion-estimation algorithm in H.264 reference software allows selecting the search range to either 8 or 16 or to a maximum of 32. Larger the search size, more the computations and better the selection on the best matching block. Figure 5.3 shows the search area and the candidate block.

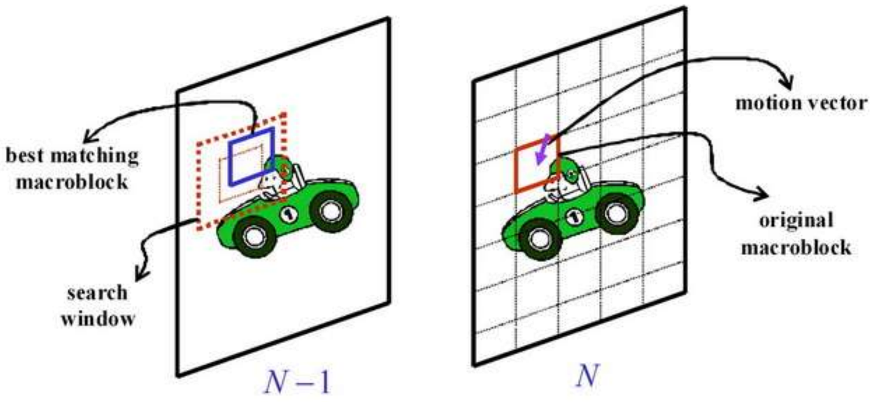


Figure 5.3 Search area and the best matching candidate block

The standard flow for the motion estimation is shown in Figure 5.4. The process starts with dividing the frame into 16 x 16 and then depending on the mode that is selected through the configuration, the frame is divided further into 8 x 8 or other sizes supported by the JM software. The searching loop takes a set of 8 x 8 blocks and searches for the best matching block by computing the SAD between the current frame and the reference frame. Motion vectors are then obtained once the most matching block is found out. As this whole process is computationally complex process, several attempts have been made to reduce the complexity. Authors in [41] considers pyramid algorithm for the motion estimation. The authors consider motion vector predicted to calculate the SAD cost. It has been reported that there is slight degradation in the video quality. Rate-distortion is performance is not shown and there is no mention of how the encoder performs in terms of the video quality and bitrate. The authors in [40], proposed a multi-pass motion estimation algorithm. In this process, SAD is generated in two passes. The main advantage of this method is that approximately 6 times speed up was achieved. Again the focus is only on speed but rate and distortion performance are not considered. Similar work in [42],[44], have attempted to reduce the complexity by utilizing different algorithms and techniques. The main issues with the earlier work are that, they only consider speed up. Also, the rate-distortion performance is not shown.

5.1.1 Theoretical estimation

In order to estimate the performance gain that can be obtained after parallel processing motion estimation can be theoretically predicted using Amdahl`s Law. Amdahl`s law states that, if *P* is portion of the code that can be made parallel and *(1-P)* is the portion of the code that cannot be parallelized, then the amount of speed up that can be achieved is given by

$$S = \frac{1}{(1-P) + \frac{P}{N}} \tag{5.3}$$

In the H.264 encoder code, motion estimation accounts to approximately 2/3rd code. According to the Amdahl`s law, if code this is parallelized on a machine with 96 cores, then the approximate speed-up that can be achieved is close to 2.2 times or 55% time reduction.

In this thesis, three techniques are used to address the issues of motion estimation dominating other processes, degradation in the video quality and increase in the bitrate. CUDA programming is utilized in order to map SAD calculation to the GPU and threads are created per block and not per pixel. This aids in the process and keeps the encoder scalable. Utilize a better search algorithm to maintain video quality. Combine SAD costs to save bitrate. Along with these techniques, shared and texture memory of the GPU is utilized for faster memory transfer since shared and texture memory are the on-chip memories.

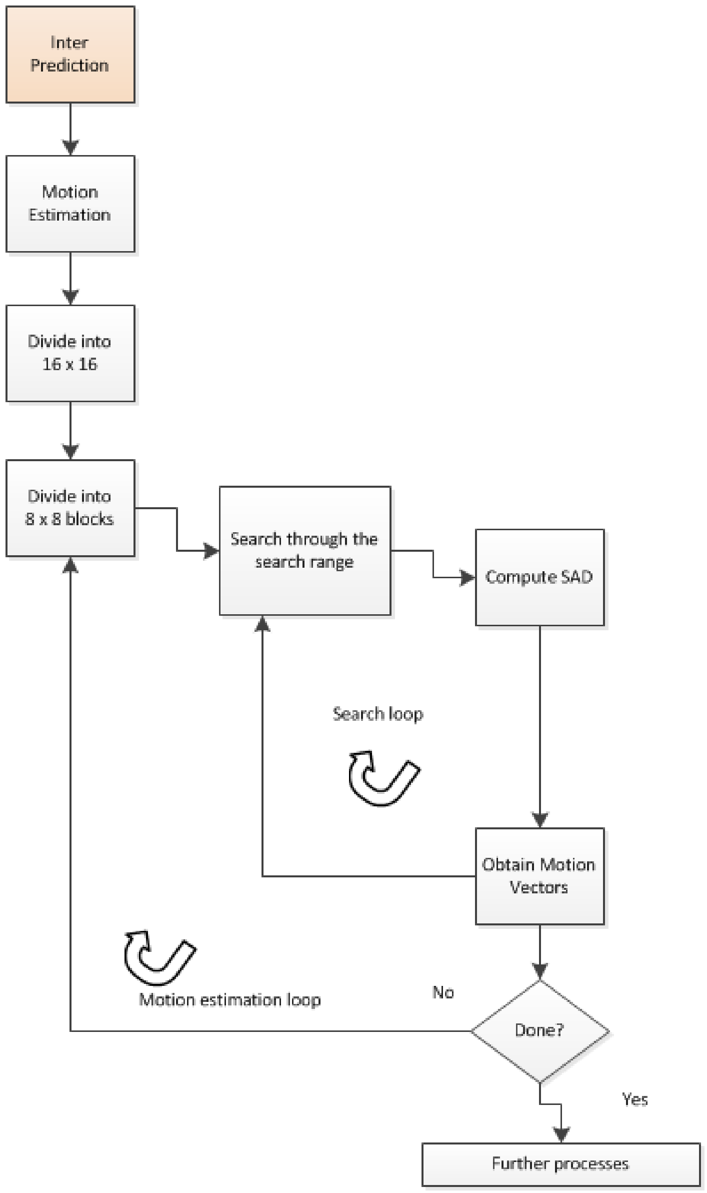


Figure 5.4 Motion estimation flow

From Figure 5.4, it can be seen that the searching loop takes a set of 8 x 8 sub blocks, compares this particular block with the reference frame within the search range and computes the SAD and also the motion vector. This process can be broken down into discrete tasks and made parallel. The outer loop of motion estimation involves lot of memory transfer and memory references since the block undergoes division and storage. This loop is not chosen for parallelization. Therefore, in this thesis, focus is on parallelizing the search loop by modifying the motion estimation flow. The modified flow diagram is shown in Figure 5.5.

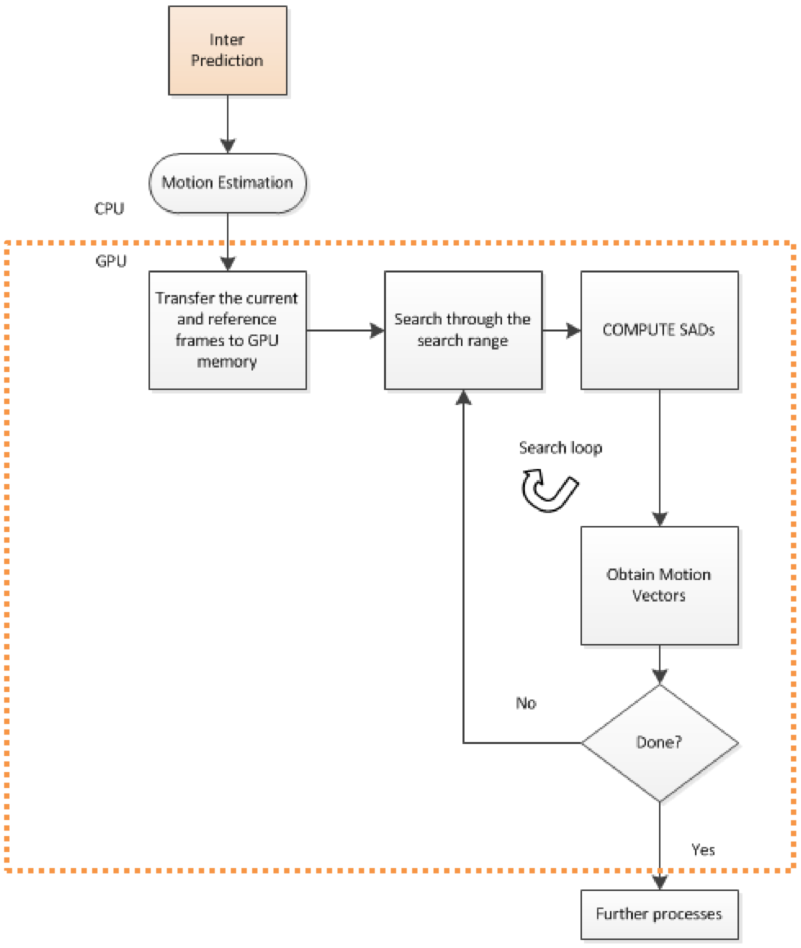


Figure 5.5 Modified flow diagram for motion estimation

The main idea in this thesis work has been to minimize memory references and make the thread block creation dependent on the image frame. Memory transfer rates between the host and GPU device poses a bottleneck for the speed-up that can be achieved. Advantages of shared and texture memory of the GPU are considered while implementing the motion estimation process parallely. Shared and texture memory reside on the chip and the access to these is faster as compared to the access to the global memory of the GPU. So instead of transferring the data to the global memory of the GPU, data is transferred to the texture memory of the GPU. This texture memory is cached into shared memory. The threads have direct access to the shared memory. This aids in significant increase in the speed up. To minimize the bitrate, SADs for 4 8 x 8 blocks are combined into a single 16 x 16 block. And to enhance the video quality, according to the studies done in [30], exhaustive full search algorithm is known to provide the best possible video quality and this is chosen as the motion estimation block matching algorithm. This exhaustive full search algorithm is highly computational, but chosen as it provides the best performance.

Figure 5.4 depicts the search region and the candidate block that provides the best match. From the figure, *Rx* and *Ry* are the search ranges in *X* and *Y* directions. For a macroblock of size N x N, the total number of operations to find out the best matching block by computing SAD is $N^2 \times (2R + 1)^2$ and the number of operations increase enormously when a larger search area is chosen.

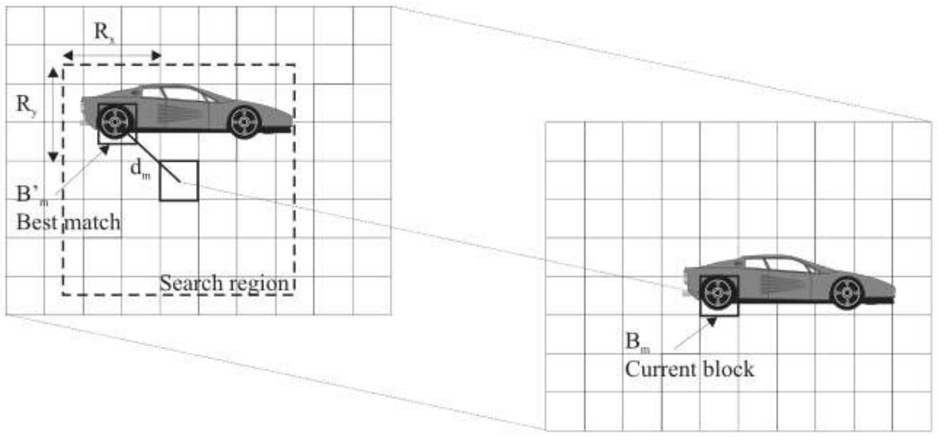


Figure 5.6 Search area and the candidate block [30][31] [32]

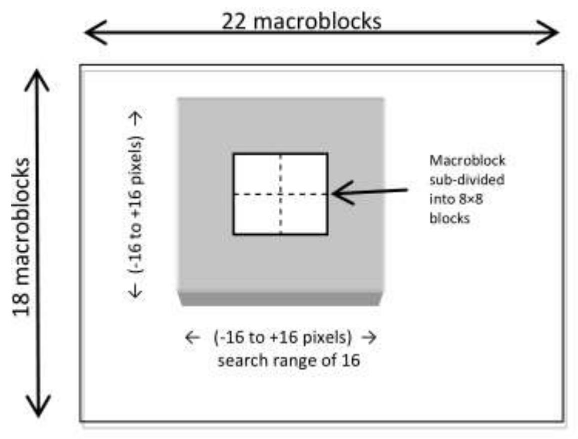


Figure 5.7 Search depiction for 352 x 288 CIF video [31]

This thesis aims at implementing the calculation of SADs algorithm on NVIDIA GPU hardware to harness its processing power. The modified pseudo-code for the motion estimation is given in Figure 5.6.

```
Motion-Estimation (CurrFrame, RefFrame)
B ← No. of macroblocks in frame
While B ≠ NIL
Divide 16 x 16 macroblock into 4  8 x 8 blocks
Block Ids to sub-blocks A1, A2, A3, A4
Motion_Vector ← (0, 0)
While column range
 While row range

   SAD1 ← COMPUTESAD (A1, CurrFrame, RefFrame)     Offload to GPU
   SAD2 ← COMPUTESAD (A2, CurrFrame, RefFrame)      to process in
   SAD3 ← COMPUTESAD (A3, CurrFrame, RefFrame)      parallel
    SAD4 ← COMPUTESAD (A4, CurrFrame, RefFrame)

SAD←SAD1 + SAD2 + SAD3 + SAD4
If SAD < previous SAD
   MBLOCKSAD ← SAD
Motion_Vector← [MV]
```

Figure 5.8 Pseudo code of a section of modified motion-estimation algorithm

The strategy that has been adopted in this thesis is to offload all the computations that calculate SAD to the graphics processor that executes all the functions given to it in a parallel fashion and returns the result back to the host Intel CPU which continues with the encoding operation. By doing this operation, there is a significant reduction in the total encoding time. For each SAD calculation, threads are invoked that do the calculation. The process of SAD implementation can be explained from the pseudo code of the motion estimation. The motion-estimation module first obtains the number of macroblocks of size 16 x 16 which is then divided further into 8 x 8 subblocks. Based on this, for a 352 x 288 frame, there will be 1584 threads created. Block Ids are then assigned to the sub-blocks. The next part is the calculation of difference of the pixel values for the block that resembles the reference block the most. There are four SAD compute functions for each 8 x 8 sub-block. Then add all the SADs obtained to find the SAD of the 16 x 16 macroblock. This value is then compared to SAD previously computed for the macroblock and the value is replaced if it is less than the previously computed value. After the computation of SAD, motion vectors are calculated in a standard fashion according to the underlying algorithm.

5.2 Prediction structure [1][2]

The H.264 standard provides several options for the selection of reference pictures for inter prediction. Figure 5.7 shows a prediction structure in which there are only I and P frames. This is the low delay, minimal storage profile known as the Baseline profile. There are no B slices in the Baseline profile as shown in Figure 2.14.

The very first frame is coded as an I slice and all subsequent frames are coded as P slices. Since P slices are predicted from previous frames, the efficiency of the prediction is quite low as only one prediction direction and one reference are allowed for each frame. The main objective of using this type of prediction structure is to keep the latency as less as possible. I slices can be inserted in between several P slices at regular intervals so as to limit the propagation of the transmission errors and to enable random access to the coded sequence.

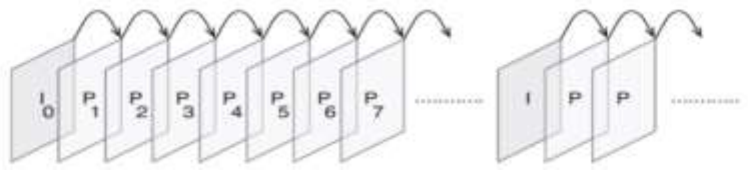


Figure 5.9 Prediction structure for low delay, random access applications

5.3 Performance Metrics [33]

The results are analyzed and compared in terms of change in the encoding time (Δ Time), change in PSNR (Δ PSNR), change in bitrate (Δ bitrate) and change in SSIM (Δ SSIM) [10]. A conclusion is drawn finally based on the results obtained from the reduction of the encoding time using task based and data based parallelism.

5.3.1 % Time reduction

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

$$\%\,Time\ reduction = \frac{Time\ taken\ by\ reference\ software - Time\ take\ by\ optimized\ software}{Time\ taken\ by\ reference\ software} * 100$$

5.3.2 % Bitrate

$$\Delta \text{bitrate}(\%) = \frac{Bitrate\ \left(\frac{kbits}{s}\right) by\ reference\ software - Bitrate\left(\frac{kbits}{s}\right) by\ optimized\ software}{Bitrate\ \left(\frac{kbits}{s}\right) by\ reference\ software} * 100$$

5.3.3 Δ PSNR [33]

$$\Delta\text{PSNR}\ (\%) = \frac{PSNR(dB) by\ reference\ software - PSNR(dB) by\ optimized\ software}{PSNR(dB) by\ reference\ software} * 100$$

5.3.4 Δ SSIM [33]

$$\Delta SSIM\ (\%) = \frac{SSIM\ by\ reference\ software - SSIM\ by\ optimized\ software}{SSIM\ by\ reference\ software}$$

5.4 Encoding specifications

The inputs to the encoder are the raw videos. The video sizes vary and generally the formats like CIF (Common Intermediate Format) are used in video conferencing systems. QCIF means " Quarter CIF", which has one fourth of the area as that of CIF. The resolution of CIF is 352 x 288 and that of QCIF is 176 x 144. Figure 5.8 shows these formats.

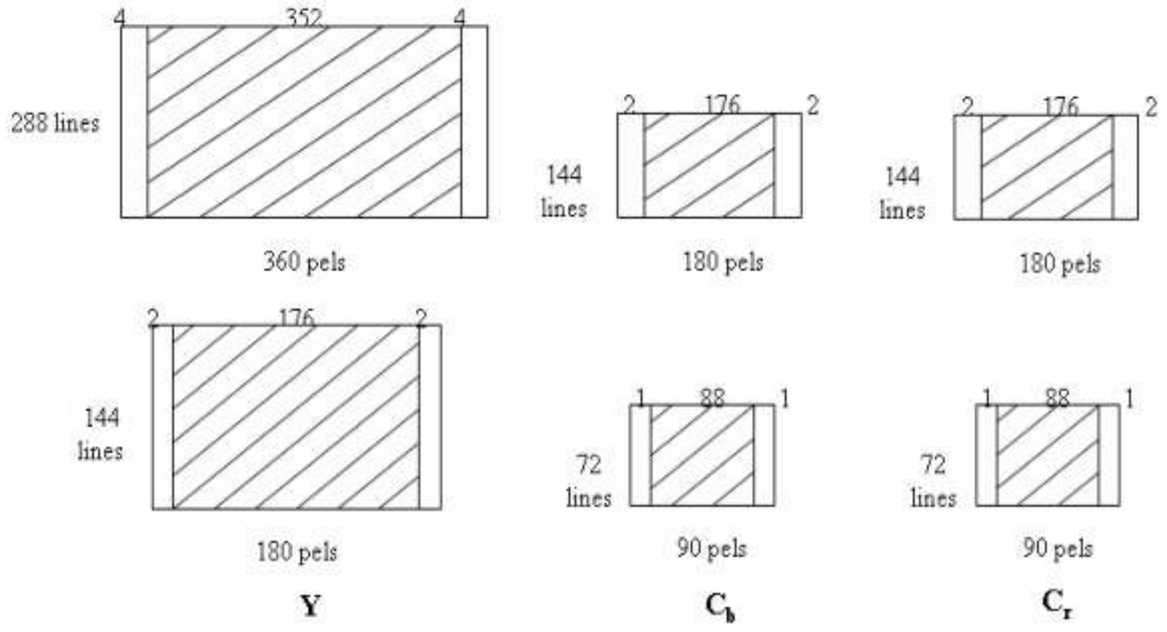


Figure 5.10 CIF and QCIF formats

The various CIF and QCIF sequences [36] are used with a frame rate of 25 frames per second. Comparison is done with the results of original JM 16.0 reference software [32] and the results obtained on optimizing the software are shown based on the different metrics like total encoding time, PSNR, bitrate and SSIM.

5.4.1 Configuring parameters and hardware specifications

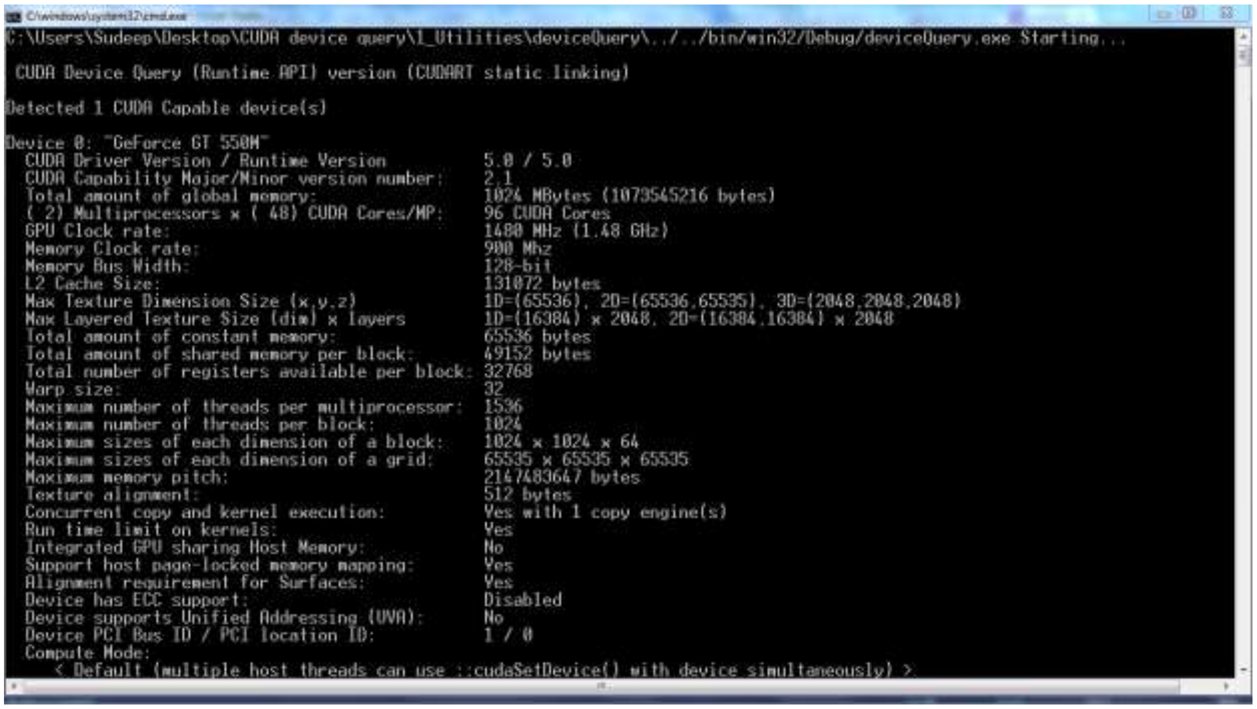
5.4.1.1 Software

GOP structure is IPPP (No B frames), search range used for motion estimation is set at 32 for both CIF and QCIF,QP (quantization parameter) values are varied as 25 and 30. Number of reference frames set to 5 and a total of 100 frames are encoded.

5.4.1.2 Hardware

The host processor is the Intel ® Core i5 CPU running at clock frequency of 2.3 GHz. The total RAM is 4.0 GB. The graphics processor used is NVIDIA GeForce GT 550M with CUDA 5.0

programming model. The details of the graphics processor are given in Figure 5.9.



```
C:\Users\Sudeep\Desktop\CUDA device query\1_Utilities\deviceQuery\../../bin/win32/Debug/deviceQuery.exe Starting...

 CUDA Device Query (Runtime API) version (CUDART static linking)

Detected 1 CUDA Capable device(s)

Device 0: "GeForce GT 550M"
  CUDA Driver Version / Runtime Version          5.0 / 5.0
  CUDA Capability Major/Minor version number:    2.1
  Total amount of global memory:                 1024 MBytes (1073545216 bytes)
  ( 2) Multiprocessors x ( 48) CUDA Cores/MP:    96 CUDA Cores
  GPU Clock rate:                                1480 MHz (1.48 GHz)
  Memory Clock rate:                             900 Mhz
  Memory Bus Width:                              128-bit
  L2 Cache Size:                                 131072 bytes
  Max Texture Dimension Size (x,y,z)             1D=(65536), 2D=(65536,65535), 3D=(2048,2048,2048)
  Max Layered Texture Size (dim) x layers        1D=(16384) x 2048, 2D=(16384,16384) x 2048
  Total amount of constant memory:               65536 bytes
  Total amount of shared memory per block:       49152 bytes
  Total number of registers available per block: 32768
  Warp size:                                     32
  Maximum number of threads per multiprocessor:  1536
  Maximum number of threads per block:           1024
  Maximum sizes of each dimension of a block:    1024 x 1024 x 64
  Maximum sizes of each dimension of a grid:     65535 x 65535 x 65535
  Maximum memory pitch:                          2147483647 bytes
  Texture alignment:                             512 bytes
  Concurrent copy and kernel execution:          Yes with 1 copy engine(s)
  Run time limit on kernels:                     Yes
  Integrated GPU sharing Host Memory:            No
  Support host page-locked memory mapping:       Yes
  Alignment requirement for Surfaces:            Yes
  Device has ECC support:                        Disabled
  Device supports Unified Addressing (UVA):      No
  Device PCI Bus ID / PCI location ID:           1 / 0
  Compute Mode:
     < Default (multiple host threads can use ::cudaSetDevice() with device simultaneously) >
```

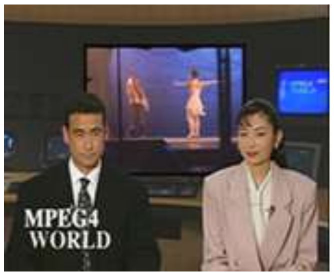
Figure 5.11 GPU Specifications of GeForce GT 550M

5.4.2 Preview of the test sequences [35]

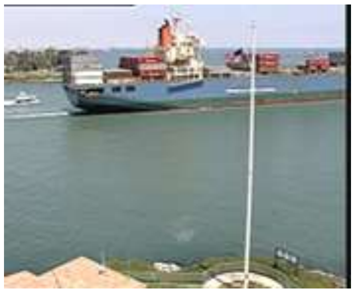
The test sequences used in this thesis are:

Akiyo

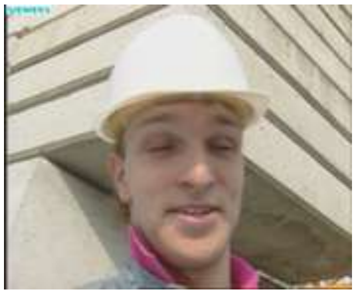
Carphone



News



Container



Foreman

Figure 5.12 Preview of the test sequences [35]

5.5 Results

5.5.1 Results for QCIF and CIF video sequences

Table 5.1 Simulation results for QCIF video sequences for QP of 25

Test Sequence	% Time reduction	ΔPSNR	Δbitrate	ΔSSIM
Akiyo	51.17	0.0112	0.012	0.0214
Carphone	48.11	0.0356	0.007	0.0215
News	51.07	0.0244	0.087	0.010
Container	50.87	0.067	0.0144	0.032
Foreman	52.12	0.0981	-0.0156	0.0117

Table 5.2 Simulation results for CIF video sequences for QP of 30

Test Sequence	% Time reduction	ΔPSNR	Δbitrate	ΔSSIM
Akiyo	49.33	0.011	0.054	0.015
Carphone	46.94	0.019	0.008	0.0053
News	48.15	0.015	0.012	0.0052
Container	48.98	0.026	0.0144	0.0105
Foreman	50.078	0.013	0.009	0.0106

5.5.2 Comparison of average encoding time

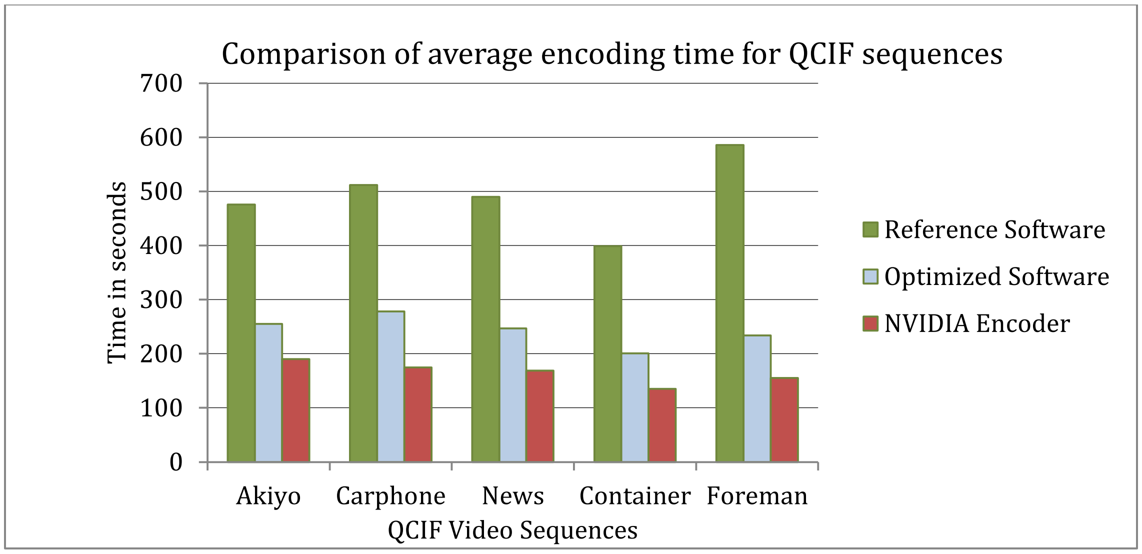


Figure 5.13 Comparison of encoding time for QCIF sequences

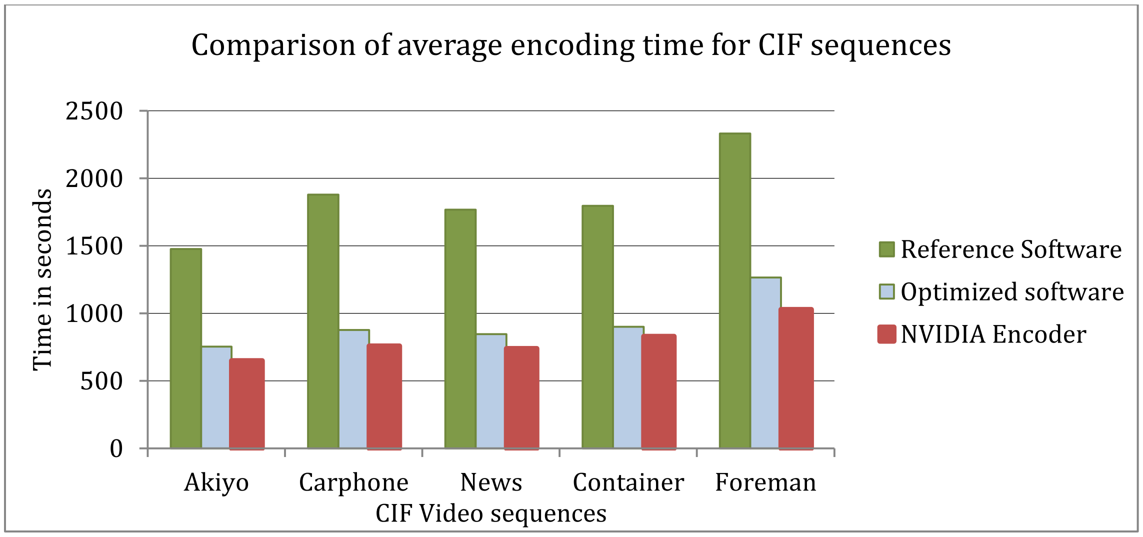


Figure 5.14 Comparison of encoding time for CIF sequences

5.5.3 Rate-distortion comparison for QCIF video sequences

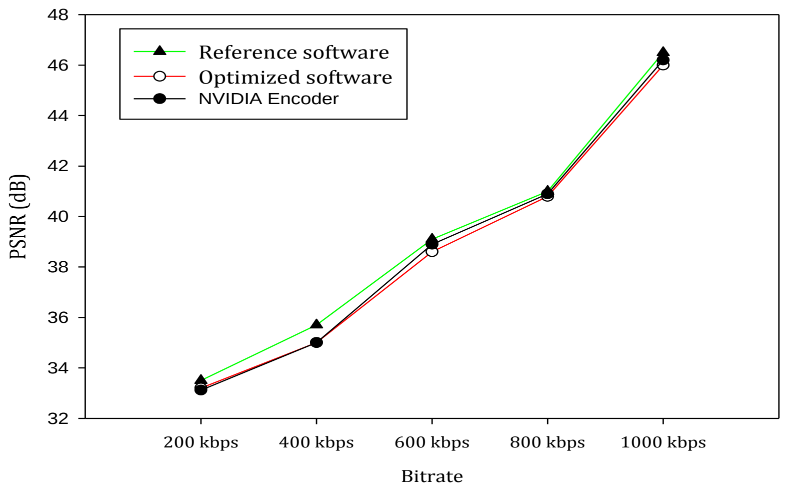


Figure 5.15 Rate-distortion plot of Akiyo sequence before and after optimization

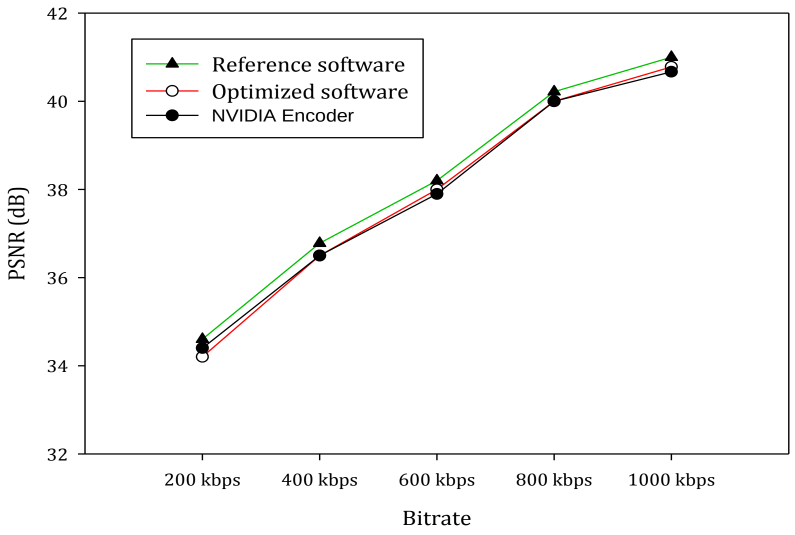


Figure 5.16 Rate-distortion plot of Carphone sequence before and after optimization

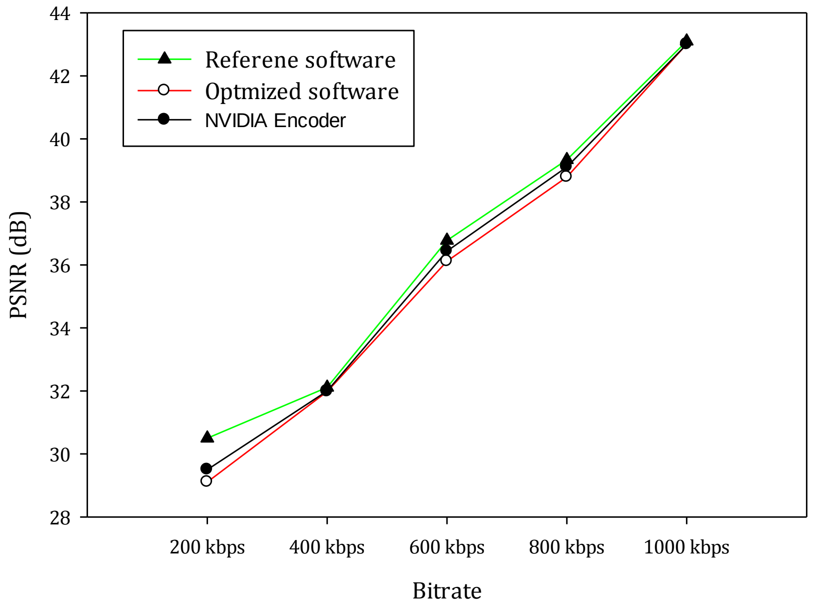


Figure 5.17 Rate-distortion plot of News sequence before and after optimization

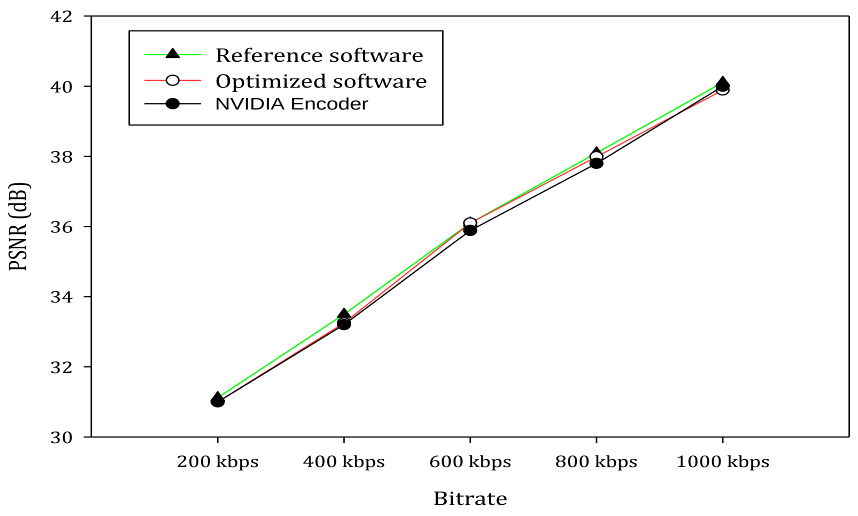


Figure 5.18 Rate-distortion plot of Container sequence before and after optimization

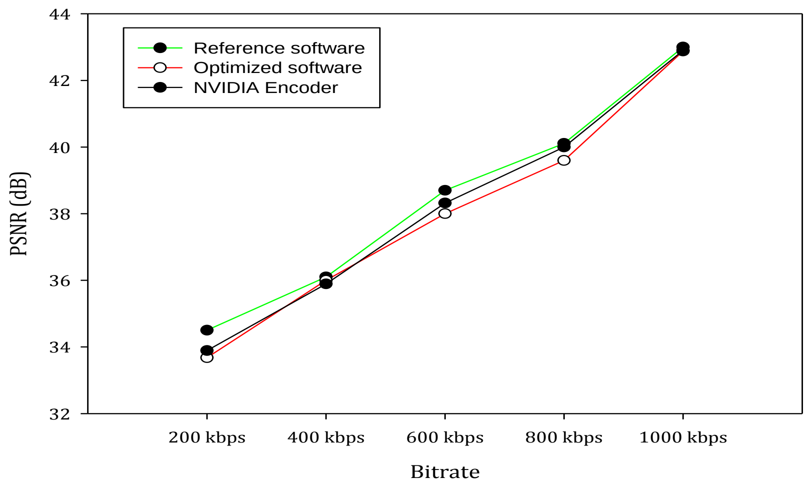


Figure 5.19 Rate-distortion plot of Foreman sequence before and after optimization

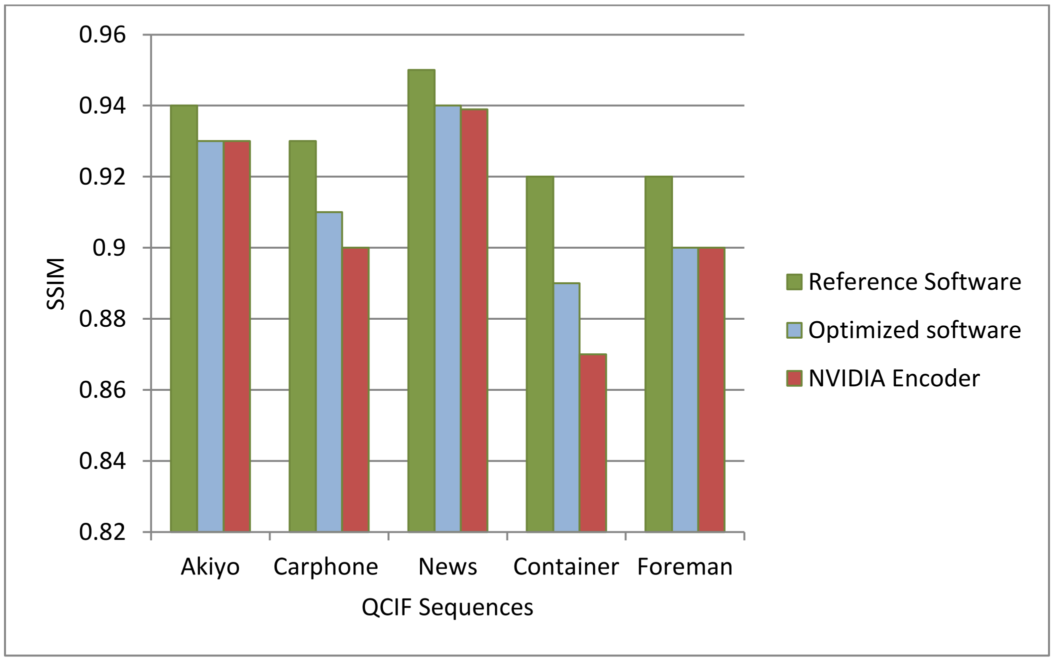


Figure 5.20 SSIM Comparison of the QCIF sequences before and after optimization at QP = 30

5.5.4 Rate-distortion comparison for CIF video sequences

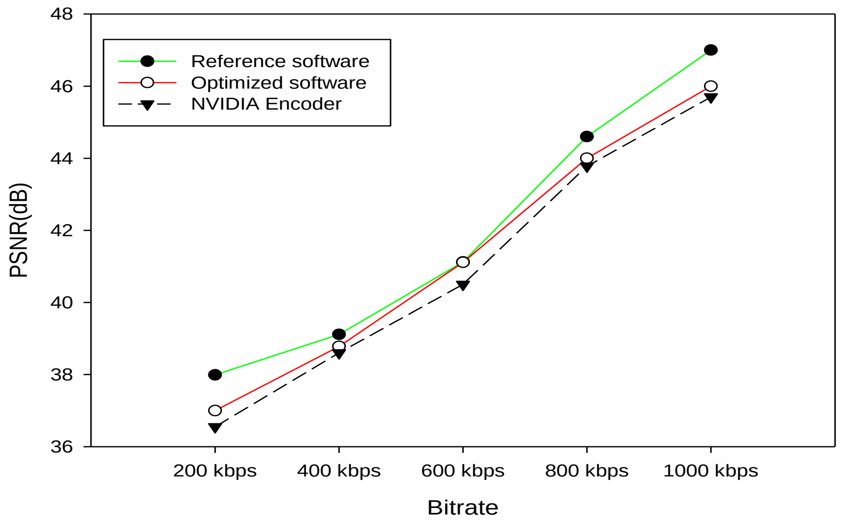


Figure 5.21 Rate-distortion plot of Akiyo sequence before and after optimization

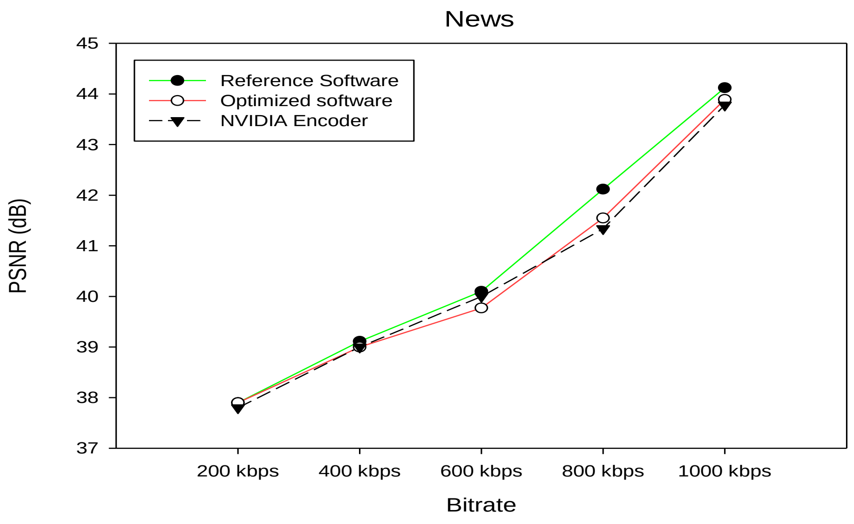


Figure 5.22 Rate-distortion plot of News sequence before and after optimization

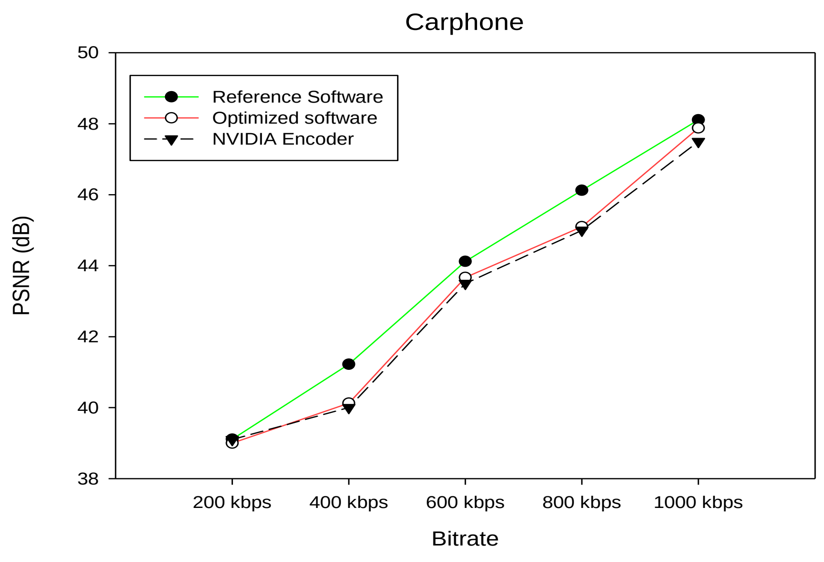


Figure 5.23 Rate-distortion plot of Carphone sequence before and after optimization

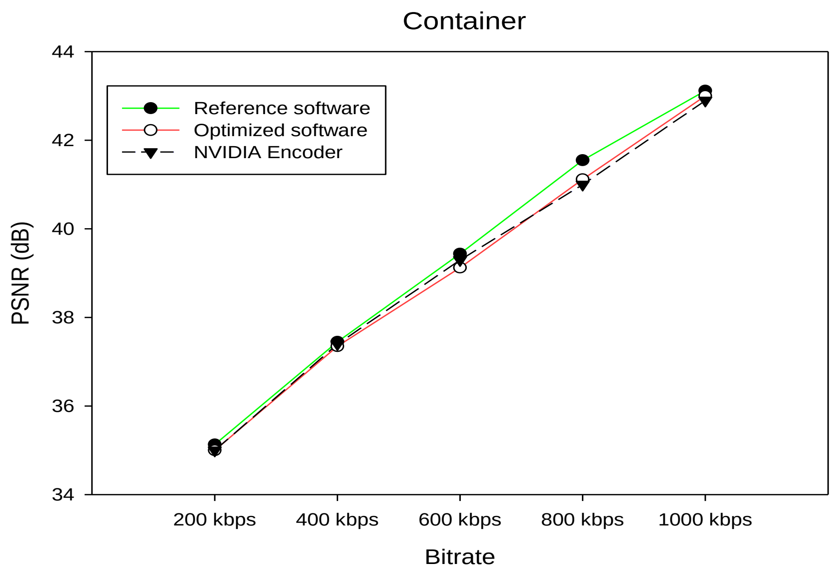


Figure 5.24 Rate-distortion plot of Container sequence before and after optimization

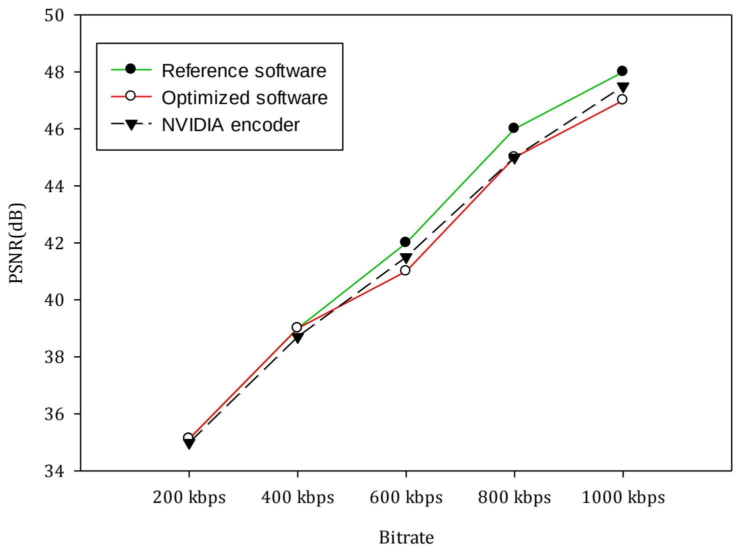


Figure 5.25 Rate-distortion plot of Foreman sequence before and after optimization

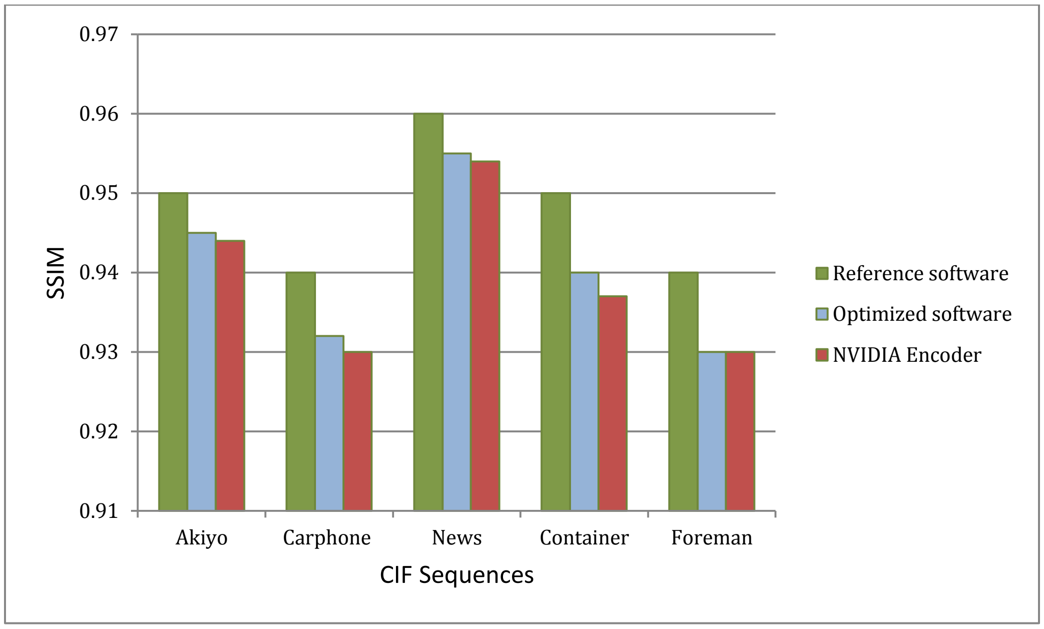


Figure 5.26 SSIM Comparison of the CIF sequences before and after optimization at QP = 30

5.6 Discussion

The objective of this thesis is on reducing the complexity of the encoding process by splitting the tasks between the host CPU and dedicated GPU. In order to be able to split the tasks between CPU and GPU, there exist hot spots or portions which have to be identified. Once the portions feasible for parallelization are identified, they are offloaded to GPU for parallel processing.

The video sequences [36] used in this thesis are application specific. The *Akiyo* sequence is rich in detailed still scenes useful for e-Learning type of applications which often time are highly detailed. The *Foreman* sequence is rich in facial expression and lip-movements useful for video conferencing type of applications. The *Container* is a sea-based video with redundancies and slow motion. The *Carphone* is again rich in motion in central part of the video again useful for video conferencing. The *News* is rich in motion and lot of lip-movement. It is challenging for any video encoder to encode videos which involve lot of motions and in turn increase the complexity. Computational complexity can be categorized based on the amount of resources required for the execution, number of interactions of various elements in the software or the amount of time or the speed with which a task is completed.

From the Tables 5.11 and 5.12, it can be observed that the time taken for the encoding operation after GPU implementation is almost 50% less than the standard reference CPU implementation of JM 16.0 [32] thereby reducing the time complexity. There is very less performance loss due to the parallelization as various modules of H.264 reference software are not modified and use the same computational logic that is used in the sequential version. From Figures 5.13 to 5.17, it can be observed that the there is a slight reduction in the PSNR value owing to the parallelization but the amount of reduction is of the order 0.1 dB which is insignificant considering the time gain that is obtained on parallelization. The amount of encoding time gain varies on the characteristics of the video. For a video with lots of motion, the encoder spends a significant amount of time on finding the best macroblock match between the reference and current frames. For a video with less motion, it is fairly easy for the encoder due to the redundant information and the time for finding out the macroblock match is less. The H.264 JM software [32] also employs motion vector reuse techniques which reduce the bitrate.

Along with the reference software, the optimized results are also compared with standard NVIDIA CUDA H.264 Encoder. The results show that the optimized encoder performs very close the NVIDIA encoder as the NVIDIA encoder is highly optimized at all levels of H.264 and not just motion estimation. The SSIM performance of the optimized software is quite close to the reference software and even better than NVIDIA encoder in some cases.

As mentioned earlier, in this thesis, a variety of videos with different motion characteristics are used for a fair comparison. The implementation strategy used in this thesis is scalable. The optimized encoder can be used across various video resolutions viz. QCIF, CIF, 4CIF, 720p, 1080p.

With parallelization there are certain limitations. Since all the threads work in parallel, in case when the SAD value till kt^{h} row (k<8) exceeds the current SAD, then there is no need to compute further. But due to the concurrent processing, no best SAD is available until the thread is done calculating. The search range is one of the important controlling factors for the performance. With a lower search range, faster calculation for motion estimation can be seen since less search range implies fewer numbers of blocks to be searched for. This search range cannot be modified while encoding is in progress. Search range cannot be made adaptive.

As this is a hardware implementation, performance also varies on the type of hardware used to run the software. With a device of higher computing capability, a better performance can be seen.

5.7 Concluding remarks

This chapter has provided the in depth details on the motion estimation and the amount of resources used by this operation in the encoding process. It then presents and discusses the results of the CPU and GPU implementation. From the different plots presented in this chapter, it is clear that the optimized software provides nearly 50% encoding time reduction. Next chapter concludes the thesis based on the analysis of the results and goes on to provide details on the extensions to the work done in this thesis.

CHAPTER 6

CONCLUSIONS AND FUTURE WORK

6.1 Conclusions

Due to the high level of computations involved in H.264 in general and motion estimation in particular, significant speed-up was achieved on GPU using CUDA and CPU combined than on CPU by data and task parallelization. As shown in Chapter 5, the implementation brings about 50% reduction in the encoding time with very less degradation in the video quality as compared to the standard reference software JM 16.0 [32].

From Tables 5.11 and 5.12, it can be observed that there is close to 50% encoding time reduction with negligible loss of PSNR and SSIM, with paltry increase in bitrate and compression ratio.

Figures 5.13 through 5.17 provide the plots for the variation of PSNR versus bitrate. The simulations were performed on both CIF and QCIF sequences. The quantization parameter (QP) was varied and 25 and 30 were chosen for the simulations. Figure 5.18 provides the SSIM values and it can be seen that there is a negligible reduction in the SSIM value and hence the quality is almost the same compared to the output of reference software.

Massively parallel processing, though, it provides huge time gains, it is often very complex to implement due to the data dependencies involved. This technique may not provide desirable results if the task under consideration cannot be divided into discrete individual problems but highly applicable when the task can be broken down into discrete problems that can be operated on parallely. Along with the strategy used in this thesis, a further speed up can be achieved if the group of pictures are processed parallely. But it will be difficult to break down the data due to the dependencies and operate on them as it may be required to transfer data back and forth from CPU to the GPU and this transfer is going to be a serious bottleneck to the performance due to the limited data transfer rate between the host CPU and GPU.

With a little modification in the code, the approach used in this thesis can be made scalable to better hardware and higher video resolutions.

6.2 Future work

This thesis is based on the idea of data-parallelism by dividing data and operating on them using the GPU. The thesis has focused on motion estimation part of the H.264 encoding. There are several other parts of H.264 that can be parallelized like arithmetic encoding or filtering. The speed up achieved in this thesis also depends on the type of GPU used. If a better GPU can be made use of, then that would provide more time gain. As H.264 has several modules like Intra prediction, transform, deblocking filtering, all these modules can be parallelized but it will be a challenging process.

There are several algorithms [30] that can be used for motion estimation like three step algorithm which can be used to decrease the encoding time. These algorithms can be incorporated into H.264 encoder and later the encoder can be broken down into discrete tasks for parallelization, which would give more time gain.

The next generation video coding standard High Efficiency Video Coding [45], currently under development, is known to be highly complex, much more than H.264. HEVC can be implemented using CUDA programming and attempts can be made to reduce the complexity.

This thesis focused only on the Baseline profile of H.264. Similar work can be carried further for High and Main H.264 profiles.

REFERENCES

[1] I.E. Richardson, "The H.264 advanced video compression standard", 2nd Edition, Wiley, 2010.

[2] S. Kwon, A. Tamhankar, and K.R. Rao, "Overview of H.264/MPEG-4 part 10", Journal of Visual Communication and Image Representation, vol. 17, no.2, pp. 186-216, April 2006.

[3] Draft ITU-T Recommendation and final draft international standard of joint video specification (ITU-T Rec. H.264/ISO/IEC 14 496-10 AVC), Mar. 2003.

[4] G. Sullivan, "Overview of international video coding standards (preceding H.264/AVC)", ITU-T VICA Workshop, July 2005.

[5] T.Wiegand, et al "Overview of the H.264/AVC video coding standard", IEEE Transactions on. Circuits and Sytems for Video Technology, vol.13, pp 560–576, July 2003.

[6] M. Jafari and S. Kasaei, "Fast intra- and inter-prediction mode decision in H.264 advanced video coding", International Journal of Computer Science and Network Security, vol.8, no.5, pp. 1-6, May 2008.

[7] W. Chen and H. Hang, "H.264/AVC motion estimation implementation on Compute Unified Device Architecture (CUDA)", 2008 IEEE International Conference on Multimedia and Expo, pp. 697-700, 26 April 2008.

[8] Y. He, I. Ahmad and M. Liou, " A software-based MPEG-4 video encoder using parallel processing", IEEE Transactions on Circuits and Systems for Video Technology, vol. 8, no.7, pp. 909-920, November 1998.

[9] 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.

[10] Z.Wang, et al, “ Image quality assessment : From error visibility to structural similarity”, IEEE Transactions on Image Processing, vol 13. Pp. 600-612, April 2004.

[11] G. Sullivan, P. Topiwala, and A. Luthra, “The H.264/AVC advanced video coding standard: overview and introduction to the fidelity range extensions” SPIE Conference on Applications of Digital Image Processing XXVII, vol. 5558, pp. 53-74, 2004.

[12] A. Puri, X. Chen and A. Luthra, “Video coding using the H.264/MPEG-4 AVC compression standard”, Signal Processing:Image Communication , vol.19 793–849, 2004.

[13] K.R. Rao and P. Yip, Discrete cosine transform, Academic Press, 1990.

[14] H. Yadav, “Optimization of the deblocking filter in H.264 codec for real time implementation” M.S. Thesis, E.E. Dept, UT Arlington, 2006.

[15] https://computing.llnl.gov/tutorials/parallel_comp/, Introduction to parallel computing.

[16] 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, Springer-Verlag, Berlin, Heidelberg, 2006.

[17] 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.

[18] S.Ge, X.Tian and Y. - K. Chen, “Efficient multithreading implementation of H.264 encoder on Intel Hyper-threading architectures”, ICICS-PCM 2003.

[19] T. Rauber and G.Runger, “Parallel programming for multicore and cluster systems”, 2nd Edition, Wiley, 2008

[20] D.Ailawadi, M.K.Mohapatra and A.Mittal, “Frame-based parallelization of MPEG-4 on Compute Unified Device Arcitecture(CUDA)”, IEEE Conference on Advanced Computing , pp 267-272 , 2010.

[21] M. A. F. Rodriguez, “CUDA: Speeding up parallel computing”, International Journal of Computer Science and Security, November 2010.

[22] NVIDIA, NVIDIA CUDA Programming Guide, Version 3.2, NVIDIA, September 2010.

[23] "http://drdobbs.com/high-performance-computing/206900471" Jonathan Erickson, GPU Computing Isn't Just About Graphics Anymore, Online Article, February 2008.

[24] J. Nickolls and W. J. Dally," The GPU Computing Era" , IEEE Computer Society Micro-IEEE, vol 30, Issue 2, pp . 56 - 69, April 2010.

[25] M.Abdellah, "High performance Fourier volume rendering on graphics processing units", M.S. Thesis, Systems and Bio-Medical Engineering Department, Cairo University, 2012.

[26] J. Sanders and E. Kandrot, "CUDA by example: an introduction to general-purpose GPU programming" Addison-Wesley Professional, 2010.

[27] NVIDIA, NVIDIA's Next Generation CUDA Compute Architecture:Fermi, White Paper, Version 1.1, NVIDIA 2009.

[28] NVIDIA, Best Programming Practices, 2009.

[30] P.Kuhn, "Algorithms, complexity analysis and VLSI architectures for MPEG-4 motion estimation", Kluwer Academic, 1999.

[31] K.Shen and E.J. Delp, " A spatial-temporal parallel approach for real time MPEG video compression", Proc. of 25th International conference on parallel processing, pp. 100-107, 1996.

[32] JM 16.0 software – http://iphome.hhi.de/suehring/tml/

[33] JM Reference Software Manual –http://iphome.hhi.de/suehring/tml/JM Reference Software Manual (JVT-AE010).pdf

[34] 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.

[35] S. 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.

[36] Test sequences - http://trace.eas.asu.edu/yuv/

[37] D.Kirk and W.-M. Hwu, "Programming massively parallel processors: A hands-on approach(Applications of GPU Computing series)", Morgan Kauffman, 2010

[38] Flynn`s Taxonomy, http://www.phy.ornl.gov/csep/ca/node11.html

[39] T. Saxena, “Reducing the encoding time of H.264 Baseline profile using parallel programming”, M.S. Thesis, E.E. Dept, UT Arlington, 2012.

[40] C-Y. Lee, Y-C. Lin, C-L. Wu, C-H. Chang, Y-M. Tsao and S-Y. Chien, “Multi-pass and frame parallel algorithms of motion estimation of H.264/AVC for generic GPU”, IEEE International Conference on Multimedia and Expo, pp. 1603-1606, 2010

[41] L. Chan, J.W. Lee, A. Rothberg and P. Weaver, “Parallelizing H.264 motion estimation algorithm using CUDA”, Proc. of Independent Activities Period (IAP), MIT, 2009

[42] R. Rodriguez, J.L. Martinez, G. Fernandez-Escribano, J.M. Claver and J.L. Sanchez, “ Accelerating H.264 inter prediction in GPU by using CUDA”, International Conference on Consumer Electronics, pp. 463-464, 2010

[43] “Amdahl`s Law”, “http://www.futurechips.org/thoughts-for-researchers/parallel-programming-gene-amdahl-said.html”

[44] N-M.Cheung, X.Fan, O.C.Au and M-C. Kung, “ Video coding on multicore graphics processors”, IEEE Signal Processing Magazine, vol. 27, pp. 79-89, 2010.

[45] B.Bross, W.-J. Han, J.-R. Ohm, G.J. Sullivan, T. Wiegand, “ High efficiency video coding (HEVC) text specification draft 8”, JCT-VC Document, JCTVC-J1003, Stockholm, Sweden, July 2012.

BIOGRAPHICAL INFORMATION

Sudeep Prakash Gangavati was born in Bagalkot, India in 1987. He received his Bachelor of Engineering degree in Electronics and Communication Engineering from Visvesvaraya Technological University in 2009. He decided to pursue Master`s degree from The University of Texas at Arlington from Fall of 2010. He has been a member of MPL, Multimedia Processing Lab, since 2011 and worked as a Research Assistant under Dr.K.R.Rao. During the summer of 2012, he got an opportunity to work at LSI Corporation as an Intern in Allentown, Pennsylvania with the DSP Software Group. His interests are in the areas of GPU Computing, Video processing and Wireless Communications. He will be joining PMC-Sierra, Sunnyvale, CA after his graduation.