Performance enhancement of H.264 video decoder using NVIDIA CUDA

By

Jitiksha A. Patel

11MICT28

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

AHMEDABAD-382481

May 2013

Performance enhancement of H.264 video decoder using NVIDIA CUDA

Major Project

Submitted in partial fulfillment of the requirements

For the degree of

Master of Technology in Information and Communication Technology

Prepared By

Jitiksha Patel

(11MICT28)

Guided By

Prof. Priyanka Sharma

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

INSTITUTE OF TECHNOLOGY

NIRMA UNIVERSITY

AHMEDABAD-382481

May 2013

Certificate

This is to certify that the Major Project entitled "Performance enhancement of H.264 video decoder using NVIDIA CUDA" submitted by Jitiksha Patel (11MICT28), towards the partial fulfillment of the requirements for the degree of Master of Technology in Information and Communication Technology of Nirma University, Ahmedabad is the record of work carried out by her under my supervision and guidance. In my opinion, the submitted work has reached a level required for being accepted for examination. The results embodied in this major project part-II, to the best of my knowledge, haven't been submitted to any other university or institution for award of any degree or diploma.

Prof. Priyanka Sharma Guide, Associate Professor, Department of C.S.E., Institute of Technology, Nirma University, Ahmedabad.	Prof. Gaurang Raval Associate Professor, PG-Coordinator(ICT), Department of C.S.E, Institute of Technology, Nirma University, Ahmedabad.
Dr. Sanjay Garg Professor and Head, Department of C.S.E, Institute of Technology, Nirma University, Ahmedabad.	Dr. K. Kotecha Director, Institute of Technology, Nirma University, Ahmedabad.

Abstract

H.264 / AVC is an industry standard for video compression, the process of converting digital video into a format that takes up less capacity when it is stored or transmitted. It provides a good compression ratio with good quality as compared to the previous video compression standards. But it all comes at a higher computational cost. So some part of this standard can be computed on GPU to free the CPU. In this report, we have discussed the H.264 video compression standard in detail and have explored the NVIDIA CUDA for using the GPU for reducing the computational requirements. NVIDIA provides Video decoding API using which we can enhance the efficiency of the decoding process. Here, we have implemented a decoder using this API's functions and compared its execution efficiency in terms of time and its Frame rate with the Joint Model(JM) Reference Software. JM Reference software is used for academic reference of H.264 and it was developed by JVT (Joint Video Team) of ISO/IEC MPEG and ITU-T VCEG (Video coding experts group).

Acknowledgements

I would like to thank **Prof. Priyanka Sharma**, Associate Professor, Department of Computer Science and Engineering, Institute of Technology, Nirma University, Ahmedabad the Guide of the project that I undertook for giving her valuable inputs and correcting various documents of mine with attention and care. She has taken the pain to go through the project and make necessary amendments as and when needed.

My deep sense of gratitude to **Prof. Gaurang Raval**, Associate Professor, PG Coordinator(ICT), Department of Computer Science and Engineering, Institute of Technology, Nirma University, Ahmedabad for an exceptional support and continual encouragement throughout part-1 of the major project.

I would like to thanks **Dr.Ketan Kotecha,** Hon'ble Director, Institute of Technology, Nirma University, Ahmedabad for his unmentionable support, providing basic infrastructure and healthy research environment.

I would like to thanks **Dr.Sanjay Garg,** Head of Department, Institute of Technology, Nirma University, Ahmedabad for his continual kind words of encouragement and providing basic infrastructure and healthy research environment.

I would like to thank all my Friends and especially Ripal Patel, Vishal Viradia, Amit Zala, Om Mehta, Bhavesh Purohit and Wasim Ghada, for being with me in any condition. I would like to thank them for their care and support,for the encouragement they have provided whenever I fell down.They have been the great motivators.

Last, but not the least, no words are enough to acknowledge constant support and sacrifices of my family members because of whom I am able to complete my dissertation work successfully.

-Jitiksha Patel
11MICT28

Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 General Overview

H.264 is a video codec standard which can achieve high quality video in relatively low bitrates. One can think it as the successor of the existing formats (MPEG2, MPEG4, DivX, XviD, etc.) as it aims in offering similar video quality in half the size of the formats mentioned before. Also known as AVC (Advanced Video Coding, MPEG-4 Part 10), H.264 is actually defined in an identical pair of standards maintained by different organizations, together known as the Joint Video Team (JVT). While MPEG-4 Part 10 is an ISO/IEC standard, it was developed in cooperation with the ITU, an organization heavily involved in broadcast television standards. Since the ITU designation for the standard is H.264, you may see MPEG-4 Part 10 video referred to as either AVC or H.264. Both are valid, and refer to the same standard.

The standard is complex and therefore challenging to the engineer or designer who has to develop, program or interface with an H.264 codec. H.264 has more options and parameters - more 'control knobs' than any previous standard codec. Getting the controls and parameters right for a particular application is not an easy task. Get it right and H.264 will deliver high compression performance; get it wrong and the result is poor-quality pictures and/or poor bandwidth efficiency. Computationally expensive, an H.264 codec can lead to slow coding and decoding times or rapid battery drain on handheld devices.[1]

1.2 Motivation

The computational requirements of encoding and decoding the H.264 force us to look for the better options to reduce the computational load of CPU. Looking for the better options, GPUs are the best pick for computationally very expensive jobs now-a-days. GPUs are small devices with hundreds of computing cores which are designed for high performance computing. The GPU accelerates applications running on the CPU by offloading some of the compute-intensive and time consuming portions of the code. The rest of the application still runs on the CPU. From a users perspective, the application runs faster because it is using the massively parallel processing power of the GPU to boost performance. This is known as heterogeneous or hybrid computing.[2]

For leveraging the parallel compute engine in GPU for solving computational problems in a more efficient way than in CPU, NVIDIA introduced a parallel computing architecture named CUDA (Compute Unified Device Architecture).It is a general purpose parallel computing architecture with a new parallel programming model and instruction set architecture which comes with a software environment that allows developers to use C as a high-level programming language. We are aiming to use CUDA for enhancing the performance of H.264 decoder using GPU.

1.3 Objective

The aim is to improve the decoding efficiency of H.264 video decoder using NVIDIA CUDA.

1.4 Scope of work

- Understanding the programmatic flow of H.264 decoder
- Understanding the NVIDIA CUDA CODEC libraries related to this work
- Using those libraries for the enhancement of the decoding efficiency in terms of execution time
- comparison of video decoding efficiency of implemented decoder and JM reference decoder

1.5 Thesis Organization

The rest of thesis is organized as follows:

- **Chapter 2** is a literature survey which provides the detailed explaination of H.264 codec and its architecture followed by the brief introduction of NVIDIA CUDA and its architecture.
- **Chapter 3** contains the proposed work.It provides the details related to the JM 18.4 video decoding software and the details of implementation of the proposed decoder, and its results and analysis.
- **Chapter 4** contains the conclusion and the future work.

Chapter 2

Literature Survey

2.1 H.264/MPEG-4 AVC (Advanced Video Coding)

As mentioned before, H.264/MPEG-4 or AVC is a standard for video compression and currently, it is one of the most commonly used formats for the recording, compression, and distribution of high definition videos.

H.264/MPEG-4 AVC is a block-oriented motion-compensation-based codec standard developed by the ITU-T Video Coding Experts Group (VCEG) together with the ISO/IEC JTC1 Moving Picture Experts Group (MPEG). The project partnership effort is known as the Joint Video Team (JVT). The ITU-T H.264 standard and the ISO/IEC MPEG-4 AVC standard (formally, ISO/IEC 14496-10 MPEG-4 Part 10, Advanced Video Coding) are jointly maintained so that they have identical technical content.[3]

The H.264 video format has a broad application range that covers all forms of digital compressed video from low bit-rate Internet streaming applications to HDTV broadcast and Digital Cinema applications with nearly lossless coding. With the use of H.264, bit rate savings of 50% or more are reported. For example, H.264 has been reported to give the same Digital Satellite TV quality as current MPEG-2 implementations with less than half the bitrate, with current MPEG-2 implementations working at around 3.5 Mbit/s and H.264 at only 1.5 Mbit/s.[4]

2.1.1 Video Format

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.

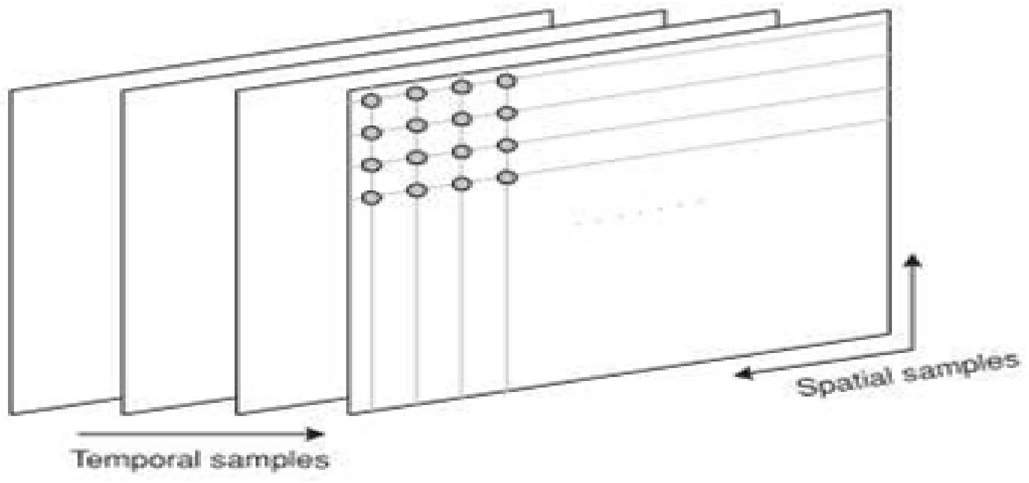


Figure 2.1: Spatial and temporal sampling [1]

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.[1] 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:

$$Y = krR + kg + kbB \tag{2.1}$$

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:

$$\begin{aligned} Cr &= R - Y \\ Cb &= B - Y \\ Cg &= G - Y \end{aligned} \tag{2.2}$$

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.

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

$$\begin{aligned} R &= Y + 1.402Cr \\ G &= Y0.344Cb0.714Cr \\ B &= Y + 1.772Cb \end{aligned} \tag{2.4}$$

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.[1]

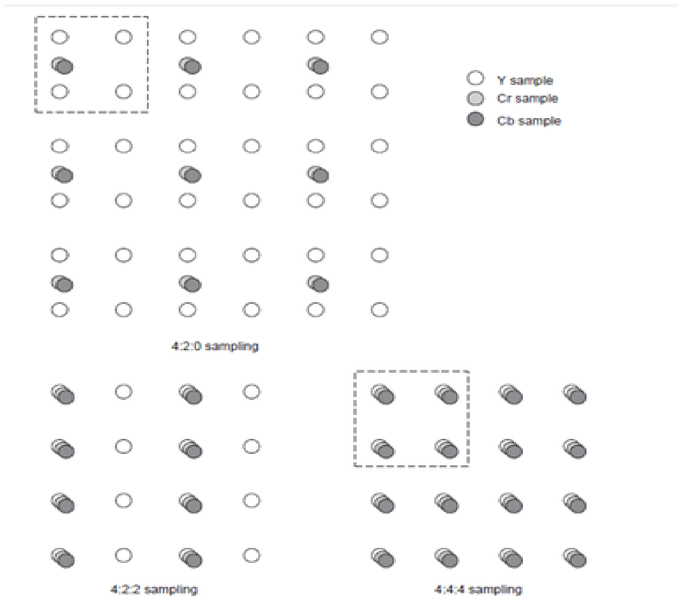


Figure 2.2: YCrCb sampling formats [1]

2.2 Basic Architecture

H.264/AVC like any other motion-based codecs, uses the following basic principles of video compression:[5]

- Transformation for reduction of spatial correlation.
- Quantization for controlling the bitrate.
- Motion compensated prediction for reduction of temporal correlation.
- Entropy coding for reduction in statistical correlation

Figure 2.3 shows the basic building blocks of H.264 video codec.

From Figure2.3, H.264 codec mainly consists of a prediction block where spatial and temporal predictions are performed to exploit the redundancy between the frames. The

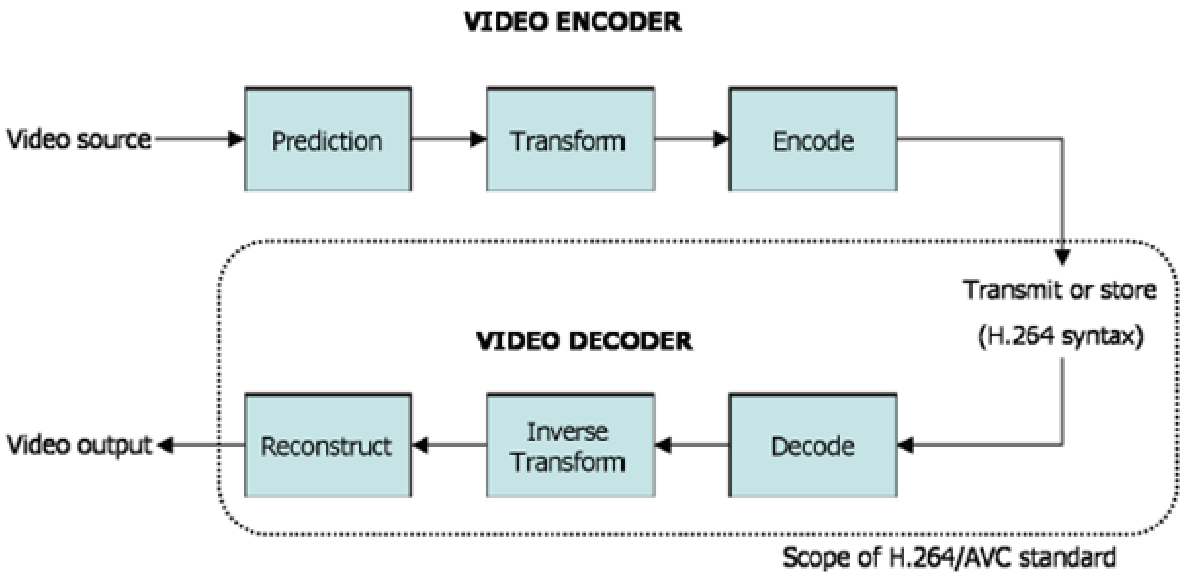


Figure 2.3: H.264 video coding and decoding process [1]

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 as shown in figure 2.4.

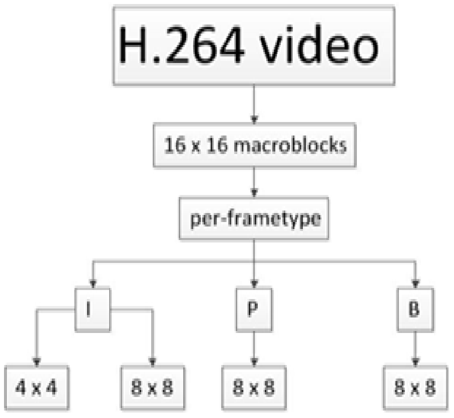


Figure 2.4: Frame division

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.[7]

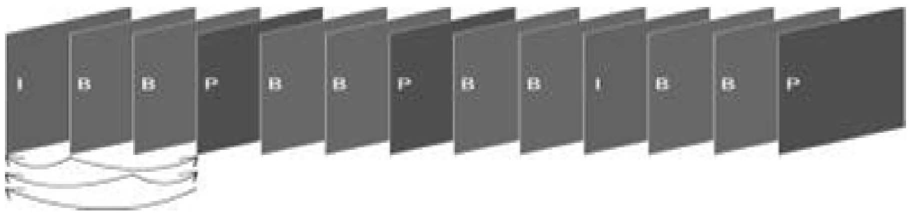


Figure 2.5: I, P and B frames [1]

- 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.

2.2.1 H.264 Encoder

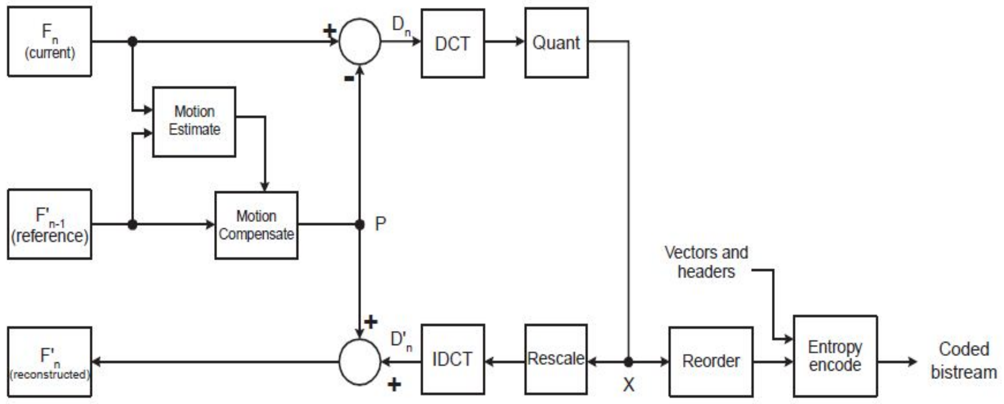


Figure 2.6: Encoder [1]

The H.264 encoder block diagram is given in Figure 2.6. 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.

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.[1]

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 2.7

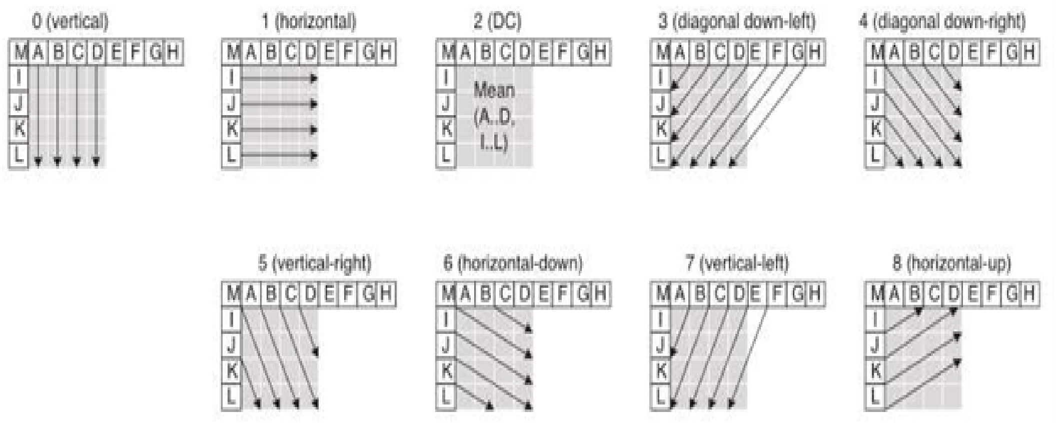


Figure 2.7: Intra prediction modes of 4 x 4 luma macroblocks [1]

According the Figure 2.7, 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.

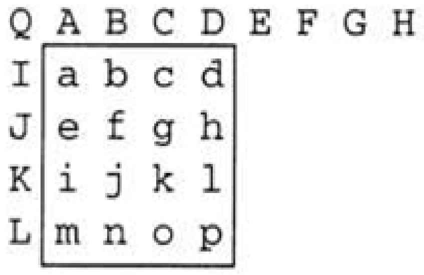


Figure 2.8: Block with samples from a till p are to be predicted [1]

Consider Figure 2.8 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 2.9.

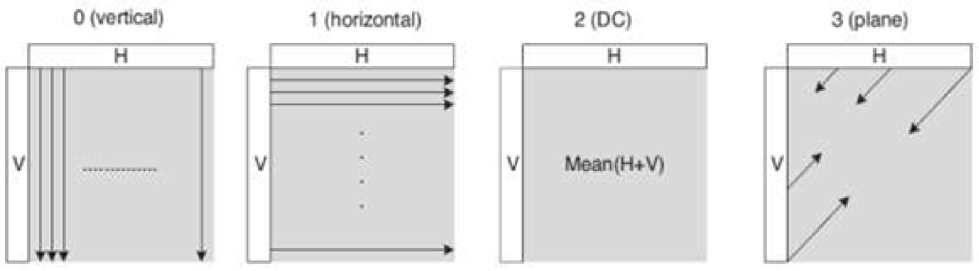


Figure 2.9: Prediction blocks for 16 x 16 block size [1]

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.

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.[1]

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 2.10. 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.

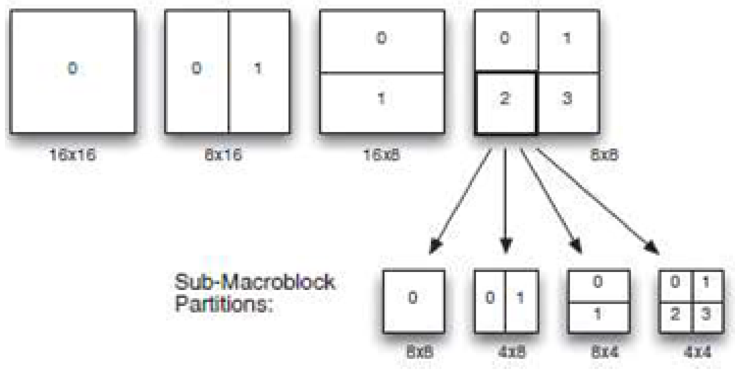


Figure 2.10: Macroblock partition sizes for inter prediction [1]

Motion compensation for a 16 x 16 block in H.264 is done for different block sizes as shown in Figure 2.10. 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.

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.[8] 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 twotap 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.[8]

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 2.11. The scan order is intended to group together significant coefficients, i.e. non-zero quantized coefficients.

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 pre-

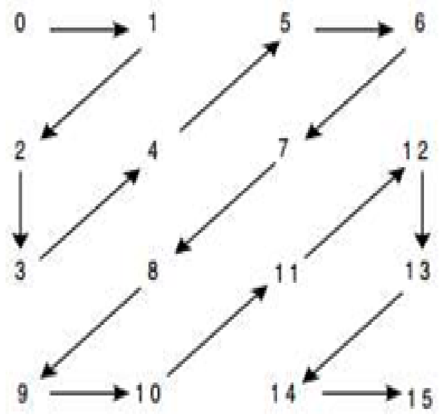


Figure 2.11: Zigzag scan order in a 4 x 4 block [9]

dictions 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.[9] The boundaries that are to be filtered is shown in Figure 2.12.

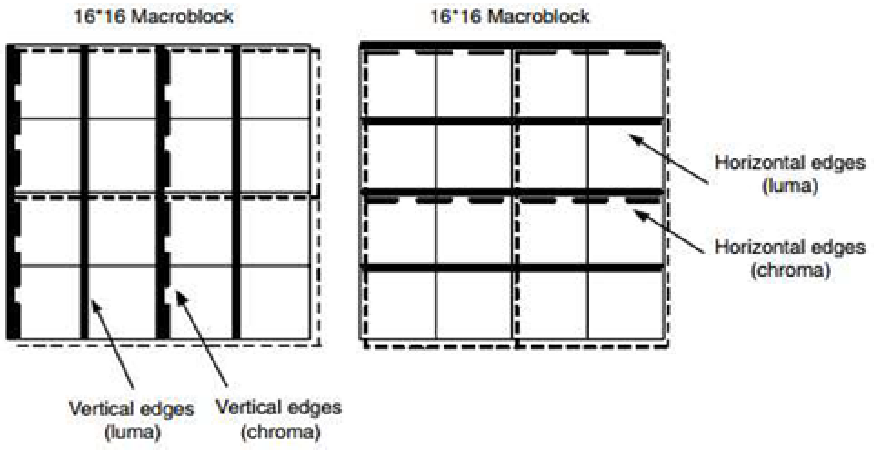


Figure 2.12: Boundaries in a macroblock to be filtered [9]

The deblocking process is done at three levels:[9]

- 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.

Entropy coding[1][9]

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:

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:[10]

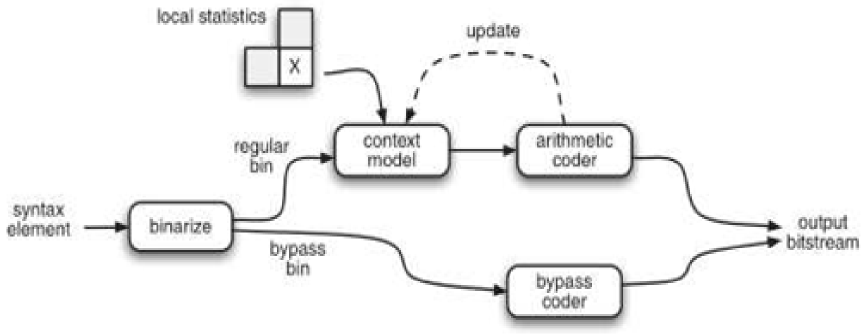


Figure 2.13: CABAC coding process [1]

1. **Context adaptive variable length coding**

    - No end block, but number of coefficients is decoded
    - Coefficients are scanned backwards and contexts are built depending on transform coefficient
    - Transform coefficients are coded with the following elements: number of nonzero 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**

    - Exploits symbol correlations by using contexts
    - Probability estimation is realized by the look up table
    - Use of adaptive probability models for most symbols

H.264 profiles [1][11]

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.

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

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

Table 2.1: Profiles and applications

2.2.2 H.264 Decoder

Decoding process is the exact opposite of the encoding process. A video decoder receives the compressed H.264 bitstream, decodes the syntax elements and extracts information such as quantized transform coefficients, prediction information etc. This data is used to recreate the video sequence. The quantized transform coefficients are multiplied by the quantization parameter. The quantization parameter is an integer value. After the transform coefficients are rescaled, the inverse transform combines the standard basis pattern, weighed by the rescaled coefficients, to re-create each block of residual data.

These blocks are combined together to form the residual data macroblock. For each macroblock, the decoder performs prediction identical to the one created by the encoder. This is then added to the decoded residual data to reconstruct a decoded macroblock which can then be displayed as part of a video frame.

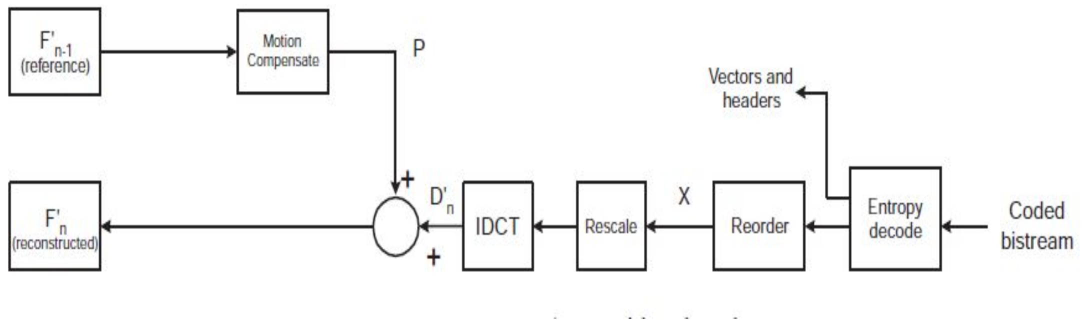


Figure 2.14: Decoder [2]

2.3 Introduction to NVIDIA CUDA

Compute Unified Device Architecture (CUDA) is a parallel computing architecture developed by NVIDIA for graphics processing. CUDA is the computing engine in NVIDIA graphics processing units (GPUs) that is accessible to software developers through variants of industry standard programming languages. Programmers use 'C for CUDA' (C with NVIDIA extensions and certain restrictions) to code algorithms for execution on the GPU.[6]

2.3.1 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 2.15.[12][13]

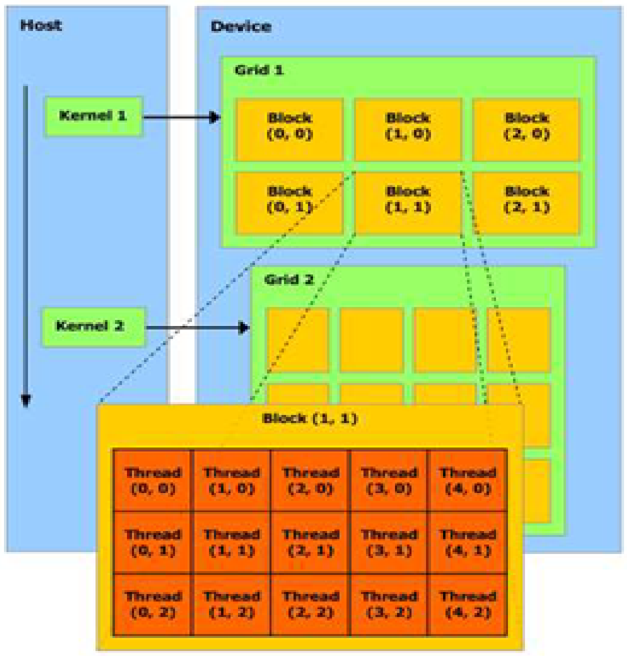


Figure 2.15: Threads grouped in blocks [13]

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.

2.3.2 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.[14][15] This memory model is illustrated is illustrated in Figure 2.16.[13]

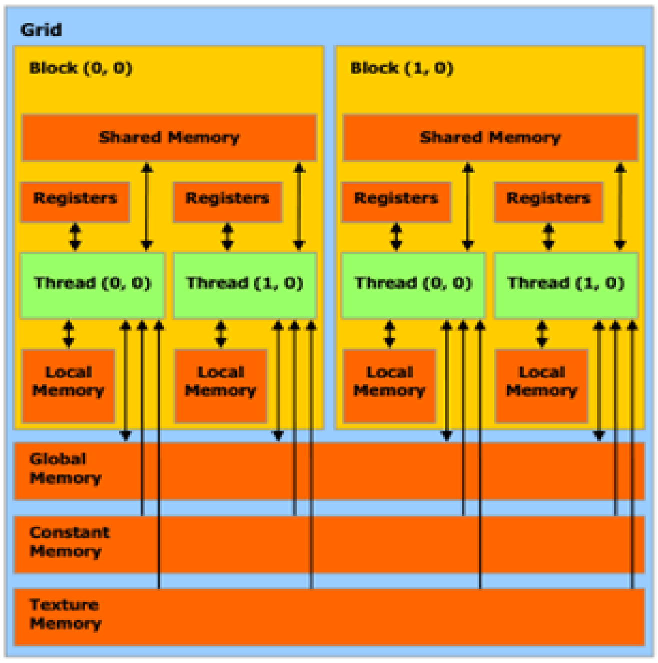


Figure 2.16: CUDA memory hierarchy [13]

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.

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.

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.

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.

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.

2.3.3 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.

2.3.4 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.

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.

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.[13]

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.[13]

Chapter 3

Proposed work

3.1 Background

NVIDIA provides high quality GPU-accelerated libraries that developers can use for video encode and decode (codec) operations on NVIDIA GPUs. The NVIDIA CUDA Video Encoder (NVCUVENC) is a hybrid CPU/GPU accelerated library that creates streams complaint with AVC/H.264 (MPEG-4 Part 10 AVC, ISO/IEC 14496-10). NVCUVENC takes advantage of hundreds of CUDA cores to accelerate the encoding of H.264. The inputs are YUV frames, and outputs are generated NAL packets. The NVIDIA CUDA Video Decoder (NVCUVID) is a library that allows developers to use video hardware acceleration for decoding MPEG-2, VC-1, and H.264 video on the GPU.

The GPU Computing SDK includes a code sample demonstrating how to use the said libraries. Using one of these samples, we will analyze the code of that sample and generate a trace file for the further use for defining the optimization area.

3.1.1 GPU analysis

The configuration of the GPU (which is used for the experiment) is as follows:

Name: GeForce GT 525M

Compute capability: 2.1

Clock rate: 1200000

— Memory Information for device 0 —

Total global mem: 1073741824

Total constant Mem: 65536

— MP Information for device 0 —

Multiprocessor count: 2

Shared mem per mp: 49152

Registers per mp: 32768

Threads in warp: 32

Max threads per block: 1024

Max thread dimensions: (1024, 1024, 64)

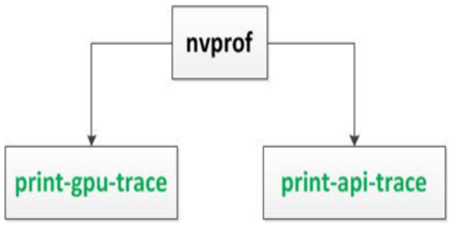
Max grid dimensions: (65535, 65535, 65535)

3.1.2 Method for profiling any CUDA executable file

CPU and GPU kernel core analysis:

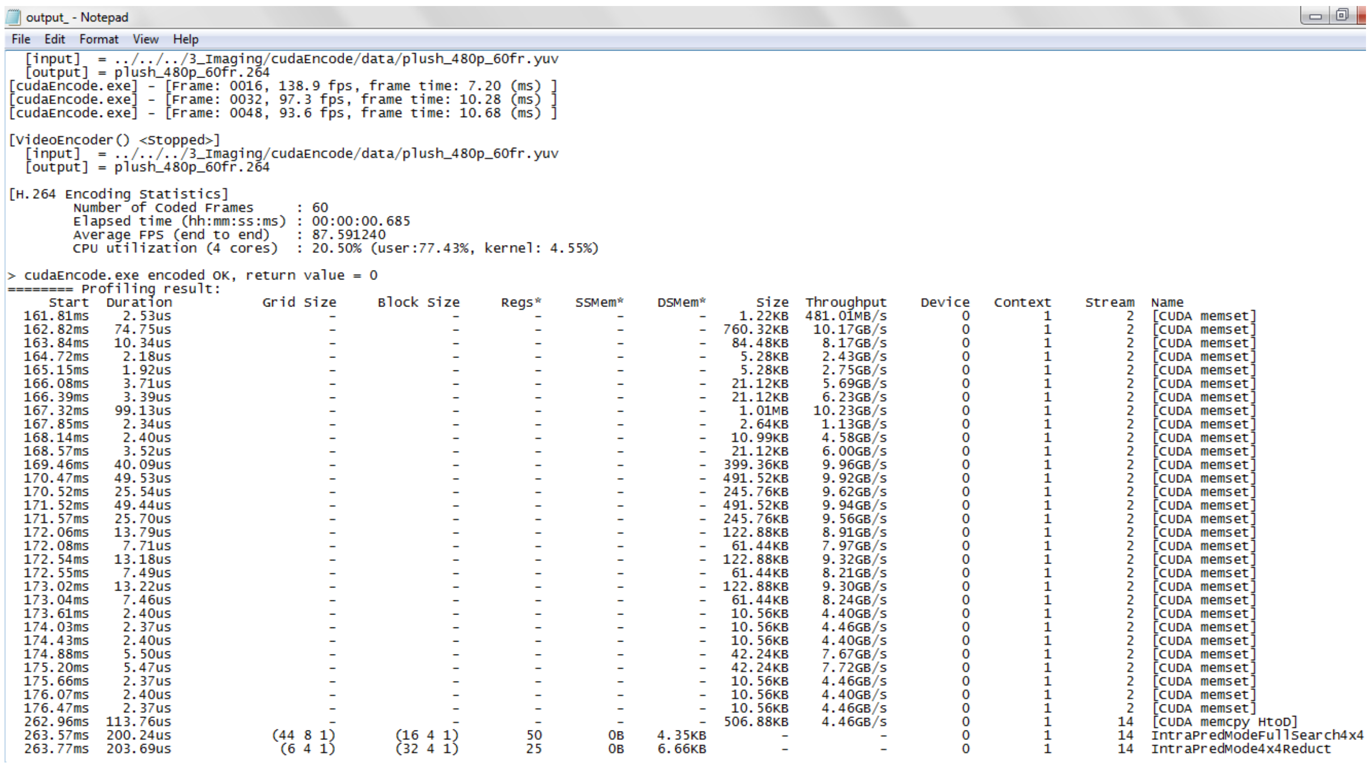
Usage: nvprof [options] [CUDA-application] [application-arguments] **Options:**

- **–print-gpu-trace** Print individual kernel invocations (including CUDA memcpy/memset kernels) and sort them in chronological order.
- **–print-api-trace** Print CUDA runtime/driver API trace.



GPU-Trace and API-Trace Modes

- GPU-Trace and API-Trace modes can be enabled individually or at the same time.
- GPU-trace mode provides a timeline of all activities taking place on the GPU in chronological order.
- Each kernel execution and memory copy instance is shown in the output.
- For each kernel or memory copy detailed information such as kernel parameters, shared memory usage and memory transfer throughput are shown.
- Here's an example: (o/p file for encoder)



```
  [input]  = ../../../3_Imaging/cudaEncode/data/plush_480p_60fr.yuv
  [output] = plush_480p_60fr.264
[cudaEncode.exe] - [Frame: 0016, 138.9 fps, frame time: 7.20 (ms) ]
[cudaEncode.exe] - [Frame: 0032, 97.3 fps, frame time: 10.28 (ms) ]
[cudaEncode.exe] - [Frame: 0048, 93.6 fps, frame time: 10.68 (ms) ]

[VideoEncoder() <Stopped>]
  [input]  = ../../../3_Imaging/cudaEncode/data/plush_480p_60fr.yuv
  [output] = plush_480p_60fr.264

[H.264 Encoding Statistics]
        Number of Coded Frames     : 60
        Elapsed time (hh:mm:ss:ms) : 00:00:00.685
        Average FPS (end to end)   : 87.591240
        CPU utilization (4 cores)  : 20.50% (user:77.43%, kernel: 4.55%)

> cudaEncode.exe encoded OK, return value = 0
======== Profiling result:
    Start  Duration           Grid Size      Block Size     Regs*    SSMem*    DSMem*      Size  Throughput    Device   Context    Stream  Name
 161.81ms    2.53us                   -               -         -         -         -    1.22KB  481.01MB/s         0         1         2  [CUDA memset]
 162.82ms   74.75us                   -               -         -         -         -  760.32KB   10.17GB/s         0         1         2  [CUDA memset]
 163.84ms   10.34us                   -               -         -         -         -   84.48KB    8.17GB/s         0         1         2  [CUDA memset]
 164.72ms    2.18us                   -               -         -         -         -    5.28KB    2.43GB/s         0         1         2  [CUDA memset]
 165.15ms    1.92us                   -               -         -         -         -    5.28KB    2.75GB/s         0         1         2  [CUDA memset]
 166.08ms    3.71us                   -               -         -         -         -   21.12KB    5.69GB/s         0         1         2  [CUDA memset]
 166.39ms    3.39us                   -               -         -         -         -   21.12KB    6.23GB/s         0         1         2  [CUDA memset]
 167.32ms   99.13us                   -               -         -         -         -    1.01MB   10.23GB/s         0         1         2  [CUDA memset]
 167.85ms    2.34us                   -               -         -         -         -    2.64KB    1.13GB/s         0         1         2  [CUDA memset]
 168.14ms    2.40us                   -               -         -         -         -   10.99KB    4.58GB/s         0         1         2  [CUDA memset]
 168.57ms    3.52us                   -               -         -         -         -   21.12KB    6.00GB/s         0         1         2  [CUDA memset]
 169.46ms   40.09us                   -               -         -         -         -  399.36KB    9.96GB/s         0         1         2  [CUDA memset]
 170.47ms   49.53us                   -               -         -         -         -  491.52KB    9.92GB/s         0         1         2  [CUDA memset]
 170.52ms   25.54us                   -               -         -         -         -  245.76KB    9.62GB/s         0         1         2  [CUDA memset]
 171.52ms   49.44us                   -               -         -         -         -  491.52KB    9.94GB/s         0         1         2  [CUDA memset]
 171.57ms   25.70us                   -               -         -         -         -  245.76KB    9.56GB/s         0         1         2  [CUDA memset]
 172.06ms   13.79us                   -               -         -         -         -  122.88KB    8.91GB/s         0         1         2  [CUDA memset]
 172.08ms    7.71us                   -               -         -         -         -   61.44KB    7.97GB/s         0         1         2  [CUDA memset]
 172.54ms   13.18us                   -               -         -         -         -  122.88KB    9.32GB/s         0         1         2  [CUDA memset]
 172.55ms    7.49us                   -               -         -         -         -   61.44KB    8.21GB/s         0         1         2  [CUDA memset]
 173.02ms   13.22us                   -               -         -         -         -  122.88KB    9.30GB/s         0         1         2  [CUDA memset]
 173.04ms    7.46us                   -               -         -         -         -   61.44KB    8.24GB/s         0         1         2  [CUDA memset]
 173.61ms    2.40us                   -               -         -         -         -   10.56KB    4.40GB/s         0         1         2  [CUDA memset]
 174.03ms    2.37us                   -               -         -         -         -   10.56KB    4.46GB/s         0         1         2  [CUDA memset]
 174.43ms    2.40us                   -               -         -         -         -   10.56KB    4.40GB/s         0         1         2  [CUDA memset]
 174.88ms    5.50us                   -               -         -         -         -   42.24KB    7.67GB/s         0         1         2  [CUDA memset]
 175.20ms    5.47us                   -               -         -         -         -   42.24KB    7.72GB/s         0         1         2  [CUDA memset]
 175.66ms    2.37us                   -               -         -         -         -   10.56KB    4.46GB/s         0         1         2  [CUDA memset]
 176.07ms    2.40us                   -               -         -         -         -   10.56KB    4.40GB/s         0         1         2  [CUDA memset]
 176.47ms    2.37us                   -               -         -         -         -   10.56KB    4.46GB/s         0         1         2  [CUDA memset]
 262.96ms  113.76us                   -               -         -         -         -  506.88KB    4.46GB/s         0         1        14  [CUDA memcpy HtoD]
 263.57ms  200.24us            (44 8 1)        (16 4 1)        50        0B    4.35KB         -           -         0         1        14  IntraPredModeFullSearch4x4
 263.77ms  203.69us             (6 4 1)        (32 4 1)        25        0B    6.66KB         -           -         0         1        14  IntraPredMode4x4Reduct
```

Figure 3.1: Generated Trace File

Profiling cudaEncode.exe Using NVPROF

======== Command: cudaEncode.exe Starting cudaEncode... [CUDA H.264 Encoder]

argv[0] = cudaEncode.exe

File: plush_480p_60fr.yuv

Source input file: plush_480p_60fr.yuv

Encoded output file: plush_480p_60fr.264

Measurement: (FPS) Frames per Second

GPU Device 0 (SM 2.1): GeForce GT 525M

Total Memory : 1024 MBytes

GPU Clock : 1200.00 MHz

Cores : 96 Cores

YUV Format Video Input to CPU System memory

NVSetParamValue: NVVE_GPU_OFFLOAD_LEVEL = 16, [GPU: Full Encode]

NVSetParamValue: NVVE_OUT_SIZE = 704,480

NVSetParamValue: NVVE_IN_SIZE = 704, 480

NVSetParamValue: NVVE_ASPECT_RATIO = 4, 3, 0

NVSetParamValue: NVVE_FIELD_ENC_MODE = 0, [Frame Mode]

NVSetParamValue: NVVE_RC_TYPE = 1, [Rate Control VBR]

NVSetParamValue: NVVE_AVG_BITRATE = 4000000

NVSetParamValue: NVVE_PEAK_BITRATE = 10000000

NVSetParamValue: NVVE_FRAME_RATE = 100

Output:

- Number of Coded Frames: 60
- Elapsed time (hh:mm:ss:ms): 00:00:00.685
- Average FPS : 87
- CPU utilization (4 cores): 20.50% (user: 77.43%, kernel: 4.55%)

Parameters for analysis

The parameters which are important for analyzing the generate trace file are as follows:

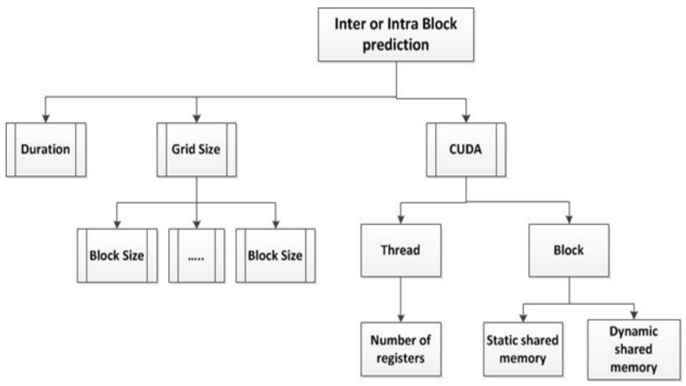


Figure 3.2: Parameters for analysis

- **Duration :** execution time for block process
- **Grid Size :** complete grid size of 1 frame
- **Block Size:** current block size in process inside GPU
- **Regs:** Number of registers used per CUDA thread.

- **SSMem :** Static shared memory allocated per CUDA block.
- **DSMem:** Dynamic shared memory allocated per CUDA block.

Need for profiling

The needs for profiling any CUDA executable, i.e. the needs to use nvprof for profiling any CUDA executable, are as follows:

- To get the execution time of any CUDA function
- To know the no. of threads generated
- To know the Static shared and Dynamic shared memory usage
- To identify the optimization area
- To get the % of GPU usage After getting these parameters values we can see that which function is taking how much time for execution, and try to reduce that time if possible.

3.2 H.264 decoder analysis and Implementation

In this chapter we have analyzed the JM 18.4 Video decoder and its execution flow. After that we have proposed our CUDA based decoder and explained the functionalities of its various functions.

3.2.1 JM 18.4 Decoder

Joint Model (JM) reference software is used for academic reference of H.264 and it was developed by JVT (Joint Video Team) of ISO/IEC MPEG and ITU-T VCEG (Video coding experts group). JM 18.4 is the latest version of the same. The JM 18.4 software provides all the features of H.264 codec which are specified in ITU-T H.264 standard but it is not optimized. It provides all features at very high computational cost. The execution flow graph of the JM 18.4 video decoder is shown in the figure 3.3:

As shown in the diagrams, the Decoder structure is allocated and the source video file is initialized first. After the initialization of the source file the file is opened for the reading of frame data by using the openBitFile function. Then the Frame decoding is started by the function named Decode_one_frame which is explained using another diagram as shown in figure 3.4

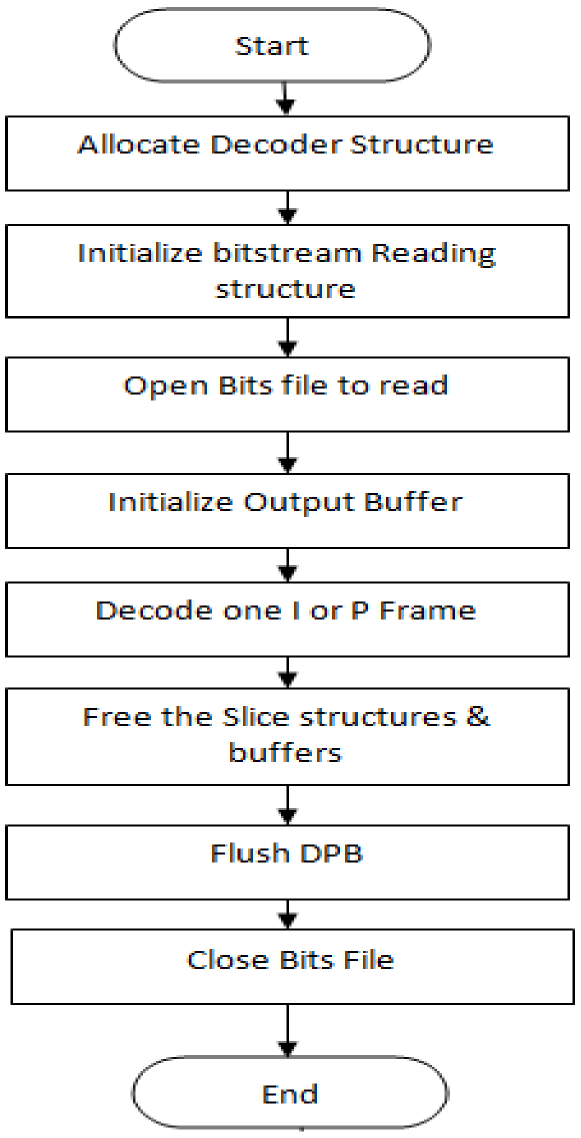


Figure 3.3: JM 18.4 decoder Flow graph (1)

In this function the frame is divided into slice and then to macroblocks of various sizes, i.e. 4x4, 8x8 or 16x16, for inverse quantization and inverse discrete cosine transform (iDCT). These two tasks are performed by the decode_one_mb function. After the whole frame is decoded, the decoder checks for the end of file if the file has reached to its end by checking the value of EOF flag. If the flag is set then the decoder frees the slice structures and global buffers and it also closes bits file, else it keeps on reading the next frame data.

In JM 18.4 decoder the processes which consumes most time are inverse quantization, iDCT, motion compensation and variable length coding. These processes can be offloaded

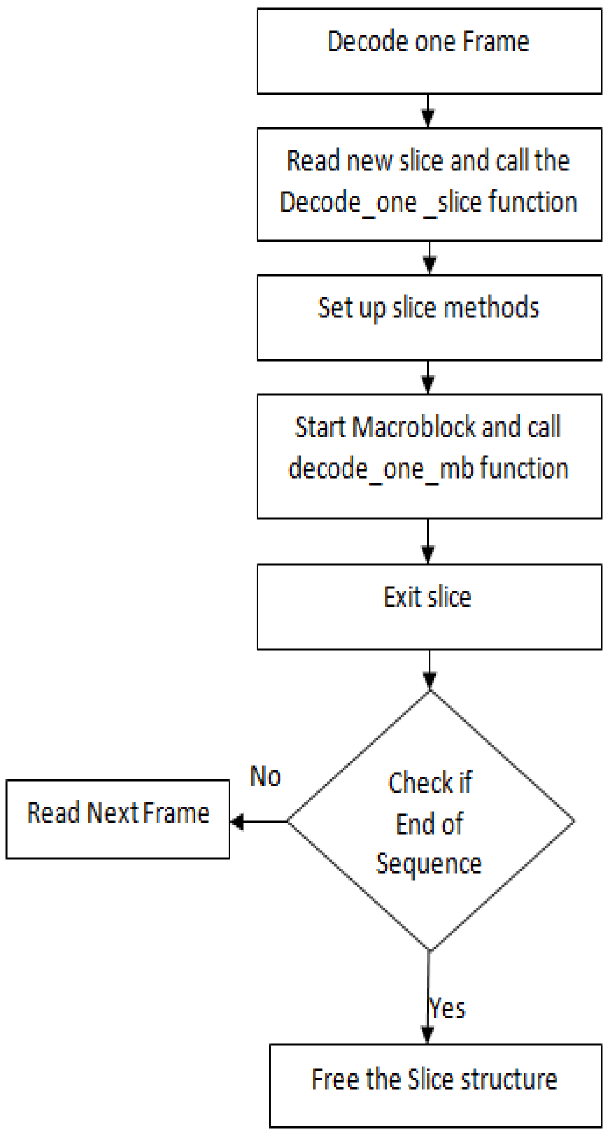


Figure 3.4: JM 18.4 decoder Flow graph (2)

to GPU using CUDA API to be executed in parallel.

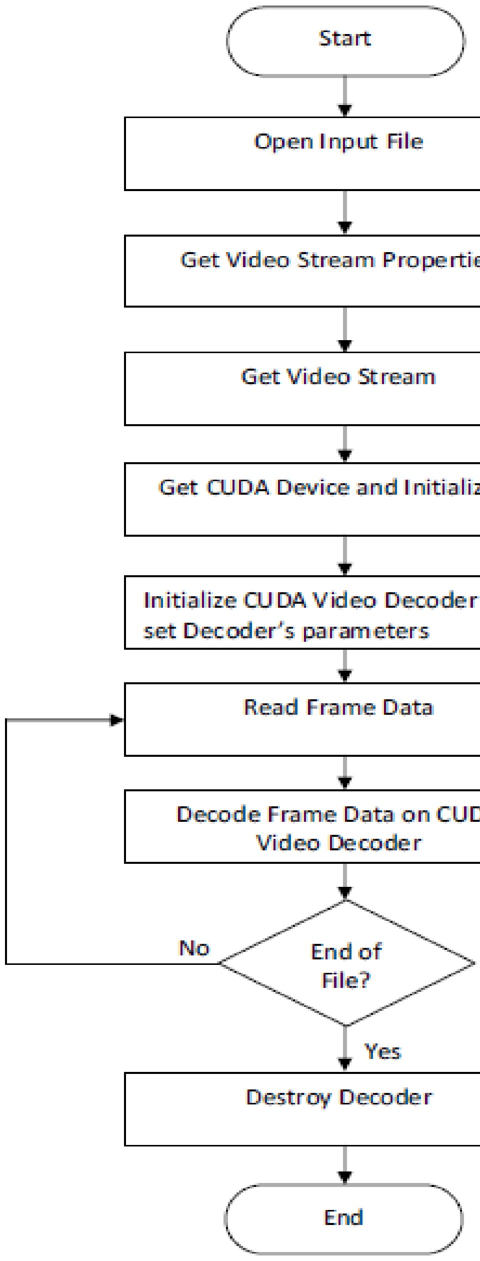
The JM 18.4 decoder takes an h.264 file as an input and generates a YUV file as an output.

Command to Execute the JM 18.4 Decoder:

```
ldecod -i testfile.264 -o testoutput.yuv
```

3.2.2 Proposed Decoder

NVIDIA provides a video decoding API for enhancing the efficiency of the H.264 video decoder. This API gives developers access to hardware video decoding capabilities on NVIDIA GPU.This CUDA Video Decoder API allows developers access the video decoding features of NVIDIA graphics hardware.This API allows the video bitstream decode to be fully offloaded to the GPUs video processor[16]. So here we have proposed a decoder which uses this API for reducing the computational cost of compute intensive tasks. The flowgraph of the proposed Decoder which uses this API is shown in Figure 3.5.



Start
Open Input File
Get Video Stream Properties
Get Video Stream
Get CUDA Device and Initialize it
Initialize CUDA Video Decoder & set Decoder's parameters
Read Frame Data
Decode Frame Data on CUDA Video Decoder
End of File?
No
Yes
Destroy Decoder
End

Figure 3.5: Proposed Decoder

For the implementing the proposed decoder, we have developed a video parser which can parse the video data to the slice level. This slice data are then fed to the API decoder function named cuvidDecodePicture( ).

The parser contains the read_frame_data( ) functions, which calls the cuvidparseVideo-Data( ) function. This function gets the various parameters like flag values of headers, payload size, pointer to the payload, payload type etc.
After the parameters are fetched and passed to the cuvidDecodePicture( ) function the frame data is copied to the device memory and the decoding process is started on GPU.The decoder provides a the output in YUV format which rests in device memory which can be brought back to the host memory.
The flag named End of file is checked after every single frame decoding to check whether the file has come to its end or not.If yes then the decoder is destroyed and if no then the next frame data is read by the read_frame_data( ) function.
The CUDA Video Decode API consists of:

- cuviddec.h
- nvcuvid.h
- nvcuvid.lib
- nvcuvid.dll

Functions provided by CUDA video Decoder API for Decoder creation and Use are listed below:
We have not used all functions in our implementation.

- Create/Destroy the decoder object
  CUresult cuvidCreateDecoder(CUvideodecoder *phDecoder, CUVIDDECODECREATEINFO *pdci);
  CUresult cuvidDestroyDecoder(CUvideodecoder hDecoder);

- Decode a single picture (field or frame)
  CUresult cuvidDecodePicture(CUvideodecoder hDecoder, CUVIDPICPARAMS *pPicParams);

- Post-process and map a video frame for use in cuda
  CUresult cuvidMapVideoFrame(CUvideodecoder hDecoder, int nPicIdx, CUdeviceptr * pDevPtr, unsigned int * pPitch, CUVIDPROCPARAMS *pVPP);

- Unmap a previously mapped video frame

  CUresult cuvidUnmapVideoFrame(CUvideodecoder hDecoder, CUdeviceptr DevPtr);

3.3 Results and Analysis

The CPU and GPU Configuration used for the Implementation are shown in tables 3.1 and 3.2 respectively:

Processor	Intel core i3-2310M
RAM	3 GB
Operating System	Windows 7
CPU Clock rate	2.10 GHz

Table 3.1: CPU Configurations

GPU Device	GeForce GT 525M
Compute capability	2.1
GPU Clock rate	1.2 GHz
CUDA cores	96
Global Memory	1024 MBytes

Table 3.2: GPU Configurations

We have used various video files for testing our decoder. The list of the details of the video files is shown in table 3.3. We have given this video files as an input to both JM 18.4 decoder and our CUDA based decoder. The decoding time taken by both decoders is given in table 3.4.

Video File Name	**Size of Frame**	**No. of Frames**
Sample1	1280x544	229
Foreman	352x288	300
cars	320x240	175
Akiyo	176x144	300

Table 3.3: Video Files

The graphical presentation of the Execution time and the Frame rates of both decoders is shown in figures 3.6 and 3.7. One can compare both decoders using these graphs.

Video File Name	Decoding Time	
	JM 18.4 Decoder	CUDA based Decoder
Sample1	174.749 sec (1.749 fps)	1.142 sec (200.53 fps)
Foreman	37.566 sec (7.986 fps)	0.672 sec (445.91 fps)
cars	7.008 sec (24.971 fps)	0.380 sec (460.11 fps)
Akiyo	2.420 sec (123.967 fps)	0.421 sec (711.54 fps)

Table 3.4: Comparison of the Execution Time

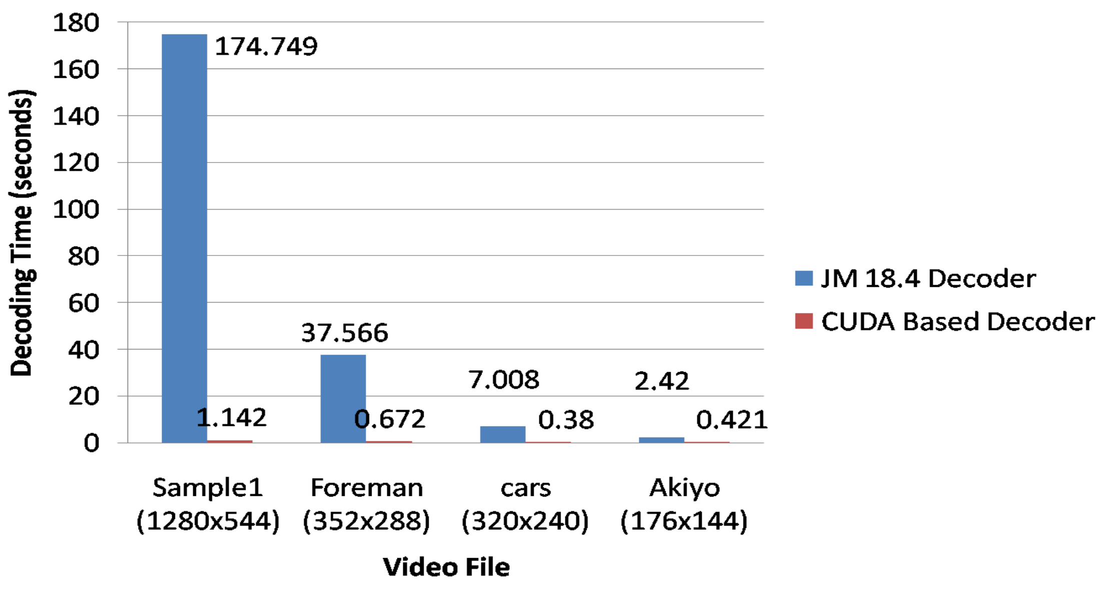


Figure 3.6: Comparison of the Execution Time

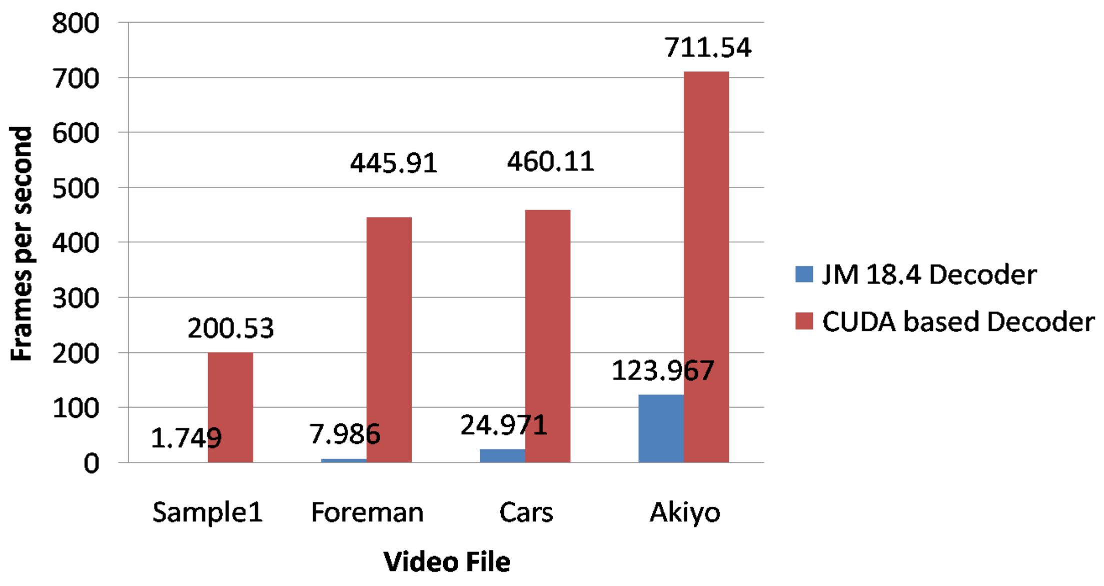


Figure 3.7: Comparison of Frame rate

Chapter 4

Conclusion and Future work

The decoding time is very less in implemented decoder compared to JM 18.4 and this is evident from the Table as well as from Graph. The reason why our decoder is so fast is because it is multi-threaded and uses GPU-optimized CUDA video Decoder API functions. It uses CUDA libraries to offloads the major compute intensive parts (i.e. motion compensation, inverse discrete cosine transform, inverse quantization, VLD (variable-length decoding) and deblocking) of decoder to the GPU for decoding. Thus, Parallel execution of those parts takes much less time than the JM 18.4.

It is possible to decode multiple video streams using CUDA API functions but it requires appropriate memory management. By managing the global and shared memory of GPU properly we can decode multiple video streams in parallel which will be our future work.

Bibliography

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

[2] Wu-chun Feng, D. Manocha., High-performance computing using accelerators, Parallel Computing,vol 33, n. 10-11, pp 645-647,November 2007.

[3] http://en.wikipedia.org/wiki/H.264/MPEG-4_AVC

[4] Wenger, et al. RFC 3984 : RTP Payload Format for H.264 Video

[5] S. Kwon, A. Tamhankar and K.R. Rao, Overview of H.264 / MPEG-4 Part 10, J. Visual Communication and Image Representation, vol. 17, pp.186-216, April 2006.

[6] http://en.wikipedia.org/wiki/CUDA

[7] 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.

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

[9] 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.

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

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

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

[13] NVIDIAs Next Generation CUDA Compute Architecture:Fermi, White Paper, Version 1.1, NVIDIA 2009.

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

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

[16] NVIDIA CUDA VIDEO DECODER API specification,August 2010.