International Journal of Emerging Engineering Research and Technology
Volume 3, Issue 3, March 2015, PP 25-30
ISSN 2349-4395 (Print) & ISSN 2349-4409 (Online)

Framework for Video Image Compression Using CUDA and NVIDIA's GPU

Ruchi Dhore[1], Amruta Gogawale[2], Shivani Borkar[3], Priyanka Jadhav[4], D D Sapkal[5]

[1,2,3,4] *Students, Computer Engineering Dept, PVG's COET, SPPU Pune, India*
[5] *Assistant Professor, Computer Engineering Dept, PVG's COET, SPPU Pune, India*

ABSTRACT

Digital videos consists major applications in today's world, like television broadcasting, multimedia and gaming. An uncompressed video of normal as well as high definition requires large amount of bandwidth in Gigabytes. Video compression techniques have got immense interest which has lead to efficient solutions for storage and transmission of video. With the age of parallel computing technology, the sequential computation algorithms can be converted in parallel computing algorithms. In the current decade, the GPU (Graphics Processing Units) have proved as a very powerful and highly parallel processing environment. NVIDIA's GPU provide a novel parallel programming language and architecture CUDA (Common Unified Device Architecture).This paper proposes an algorithm for video images compression using GPU and CUDA.

Keywords: CPU, CUDA, GPU, Parallel Processing, Parallel Computing, RLE

INTRODUCTION

Over the last four decades, lot of the work has been done on video compression techniques by many researchers. Many methods have been suggested to achieve more compression rate. MPEG standard is considered as a pioneer work in this area in which they have used conversion from RGB to YUV followed by the division of the image in blocks.

For the uncompressed video image having a resolution of 640 x 480 pixels with 30fps and 24-bit colour depth requires a bandwidth around 221.184Mbps. Bandwidth requirement increases gradually as the resolution increases. Accordingly, encoding mechanisms have been devised. A video with normal resolution 640x480 pixels as well as high resolutions such as 720x480, 1280x960, 1600x1200, 2048x1536, 2560x1920 pixels requires a huge amount of storage space. Processing the frames of a video in sequential fashion puts a heavy burden on a PC. Consider a single frame with resolution 640 x 480 pixels has in total 307,200 pixels. Applying algorithm on these many no of pixels takes huge amount of CPU time as well as places burden on CPU, resulting in serious CPU load [1].

GPU is gaining popularity because of its parallel features. GPU technology introduced CUDA environment which is a general purpose platform for parallel computing and programming model which leverages the parallel computing engine in NVIDIA's GPUs to resolve many highly complex computational problems more efficiently than on a CPU and the language comes with a software environment that allows developers to use C as a high-level programming language [2].

Compression deals with two aspects. Firstly, with redundant data, where the data is repeated can be stored in less space by replacing repeating data by a sequence. Secondly, the RGB format requires minimum 24-bit colour depth with 8-bit for each component and each having a value from range 0-255. These RGB images can be replaced by YUV colour format where colour information is stored in only 2 components U and V.

This paper proposes an algorithm for video images compression using GPU and CUDA. Our algorithm dynamically calculates next key frame by using difference between key frame and sub-frame which is less than threshold value.

Address for correspondence

ruchidhore93@gmail.com

GPU AND CUDA

In 2007, NVIDIA released CUDA for general-purpose and parallel computing. CUDA has C-like environment which uses C programming tools and C compiler which provides general programming environment. There are two main terms in CUDA, host and device. The host is CPU and its memory and the device is GPU and its memory. A programmer can write C code which executes using normal C compiler and kernel function which executes using nvcc compiler. A function that executes on the device is called kernel. The parallel nature is implemented using number of threads which execute in parallel. Basic outline of a CUDA program with interaction between CPU and GPU:

- CPU allocates storage on GPU - cudaMalloc
- CPU copies input data from CPU to GPU - cudaMemcpy
- CPU launches kernel(s) on GPU to process the data - Kernel launch
- CPU copies results back to CPU from GPU - cudaMemcpy

A kernel function consist the part of program which executes no of threads in parallel way. For instance, we want to calculate cubes of 64 numbers, we write cube<<<1, 64>>>(...) which is a kernel function with 64 threads in 1 block called block of threads. When a function call occurs, the cubes of 64 numbers will be calculated at a time in parallel. One can use more than 1 block according to requirement. The older GPUs supported 512 threads per block, the newer GPUs support 1024 threads per block. A more hierarchical structure involves grids of blocks. Its structure can be given as Kernel<<<Grid of blocks, Block of threads>>> (...) where Grid of blocks and Block of threads can be 1, 2 or 3D. A single thread uses local memory, a thread block (per-block) use shared memory and grid of blocks (all threads) use global memory. The GPU is made up of Streaming Multiprocessors (SMs) and is responsible for allocating blocks to SMs. A SM has its own simple processors and memory. The major application of CUDA is image and video processing [3].

RELATED WORK

In 2008, Zhiyi Yang et al., presented a paper on parallel image processing based on CUDA. They implemented several classical image processing algorithms by CUDA including histogram equalization, removing clouds, edge detection and DCT encode and decode. Through their experimentation, they have shown that 40x, 79x and 200x speedup for histogram, cloud and edge detection accordingly [4]. In 2008, Lei Pan et al., presented a method for medical image segmentation using CUDA. They implemented different segmentation algorithms on real brain images. They have shown the advantages of using CUDA and GPU [5]. In 2009, Yasuyuki Miura and Shogo Yamato presented a method in their IEEE paper based on parallel encoding system for the cluster grid environment having multi-core PCs to build a video encoder. Grid was constructed of multiple PCs in LAN, and video images are encoded. The logic behind their method is to apply simple compression method in order to transmit an encoded data over the fast Ethernet LAN. In the initial step they transform the video images from RGB to YUV followed by a simple difference between key frame and predicted frames of mpeg video. After that, RLE (Run Length Encoding) with its variations RLE2, RLE3, RLE4 and Huffman encoding have been used to compress the differential images. They have shown efficient compression by using simple difference, RLE and Huffman encoding [1]. In 2013, Yasuyuki Miura, Sho Nakane and Shigeyoshi Watanabe presented encoding method by using motion vector from video encoder. In this method, difference processing using the motion vector from encoder PC is performed instead of simple difference. It was shown that to progress the compression ratio by using motion vector. It seems that real time processing becomes possible to some extent with the help of GPU [6]. In 2013, Stamos Katsigiannis et al., developed a GPU based real-time video compression method for video conferencing. They presented a scalable video coding algorithm for lossy and lossless methods and variable bitrate encoding schemes in order to achieve compression [7]. In 2014, Huayou Su et al., presented an Efficient Parallel Video Processing Techniques on GPU. They used serial optimization method for motion estimation and enhanced the parallelization. They offloaded 96% encoding load of H.264 encoder to GPU [8].

PROPOSED FRAMEWORK

Figure 1 depicts the overall proposed system architecture based on Yasuyuki Miura et el [1], framework to be implemented by using CUDA C on GPU.

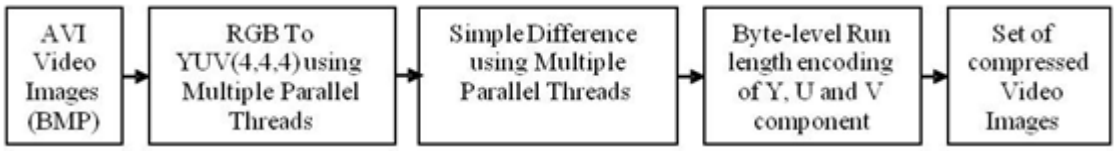

Figure1. *Overall System Architecture*

RGB to YUV Mapping on CUDA

Initially, frames of an AVI (Audio Video Interleaved) video are extracted. All these frames are BMP images in which all pixels are represented using their RGB values. BMP is an uncompressed image format where each pixel is stored using 24-bits color format. Each pixel is associated with three intensity values as Red, Green and blue accordingly. As BMP is an uncompressed file format and requires more amount of space in memory and good amount of bandwidth for data transfer, it is obvious to convert BMP images in RGB color format to YUV format.

The first step of the implementation is to convert BMP images in RGB plane to YUV plane where Y component represents the luminance while U and V components used to represent color. As the human eyes cannot easily differentiate change in color, it is possible to reduce the size of color data by using YUV color model. YUV color model supports three different modes for the conversion as YUV(4,4,4), YUV(4,2,2),YUV(4,2,0) and YUV(4,1,1). The first step in this process of encoding is transformation of video image of RGB (Red, Green, Blue) format to YUV(Y-Luminance, U&V-Chrominance) format. The equations for RGB to YUV conversion are:

Y = R * 0.2990 + G * 0.5870 + B *0.1140

U = R * -0.1470 + G * -0.2890 + B * 0.4360 + 128

V = R * 0.6150 + G * -0.515 + B * -0.1000 + 128

A generalized formula can be used to calculate number of grids and blocks. This formula will work on different resolutions where height and width of an image is divisible by 16. For image of 640x480 resolution image, total of 307200 threads are created with each thread computing Y, U and V. Total of 1200 blocks can be created with 256 threads per block. The size of grids and blocks can be calculated as:

```
dim3 block(16,16);
dim3 grid(Height_of_input_image/16, Width_ of_input_image /16);
```

Simple Difference Using CUDA

After conversion into YUV format the next step is to measure the similarity between the key frame and sub-frame. It can be done by calculating the simple difference. In AVI video format, there are no separate key frames. Key frames can be calculated on basis of similarity of frames. Identifying similarity results in finding key frames in dynamic way and remaining frames act as sub-frames which only store difference according to respective key frame. Therefore, video with fast changing motion after compression will contain more number of key frames and less number of sub-frames. For deciding which frame is to be considered key frame, number of similar pixel in key frame and sub-frames can be calculated. Next key frame can be selected by selecting the appropriate threshold value. In our case we will look for 70% similarity between the key frame and last sub-frame.

In key frame, intensities of all pixels remain same whereas in sub-frames subtracted values get stored. For calculating simple difference, grids and blocks can be calculated in similar fashion as in first module where each thread takes difference between corresponding components (i.e. Y, U, V) of key frame and sub-frame. The process of calculating the simple difference can be accomplished by following mathematical formulas. The pixel value after simple difference is as follows:

$df1f2y = pixel_{iY} - pixel_{i+1Y}$

$df1f2u = pixel_{iU} - pixel_{i+1U}$

$df1f2v = pixel_{iV} - pixel_{i+1V}$

The size of grids and blocks can be calculated as:

```
dim3 block(16,16);
```

dim3 grid(Height_of_input_image/16, Width_ of_input_image /16);

Run Length Encoding

Run length encoding (RLE) is a kind of lossless technique which makes use of consecutive data. A consecutive repeated data can be represented using a scheme (count, data). General RLE, Pack Bits, and RLE-n are some of the methods.

In this method, the pixel values after Simple Difference df1f2y, df1f2u, df1f2tv can be scanned horizontally, and RLE byte streams can be generated for Y,U and V. Run length encoding is advantageous when there is lots of consecutive data but is consumes more space if non-consecutive data is more.

Example: 0 0 0 0 0 144 126 234 234 16 16 16 1 1 1 1 1 155 155 155 155 255 0 255 0

RLE : (0,5) (144,1) (126,1) (234,2) (16,3) (1,5) (155,4) (255,1) (0,1) (255,1) (0,1)

PROPOSED ALGORITHM

Algorithm: Video Image Compression Using Dynamic Method

Input: Set of Video frames/BMP Images of AVI video format

begin

foreach *video frame* ***do***

Convert video frame in RGB format to YUV(4,4,4) format using multiple parallel threads

end foreach

foreach *YUV frame* ***do***

Consider first frame as a key frame

Calculate difference between key frame and each sub-frame using multiple parallel threads

if *difference between key frame and sub-frame is less than threshold value* ***then***

Consider sub frame as a key frame

end if

end foreach

foreach *Key Frame and Difference Frame* ***do***

Byte-level Run length encoding of Y, U and V component and store in file

end foreach

end

Output: Set of compressed Video frames

MATHEMATICAL MODEL

The overall system architecture can be represented as set of four modular components as below:

SYSTEM = {INPUT, FUNCTIONS, OUTPUT, FAILURE_CASES}

INPUT = {Set of Video frames/images of AVI video format}

Mathematically, an input video can be represented as a set of frames as follows:

$invideo = \{inframe_1, inframe_2, inframe_3, ...inframe_n\}$

$inframe_i \subset$ *invideo is a subset of RGB frames*

where

invideo is an input video

$inframe_1$ *to* $inframe_n$ are video frames of RGB format

FUNCTIONS = {RGBToYUV(), SimpleDifference(), RLE())}

OUTPUT = {Compressed video frames of AVI}

FAILURE_CASES = {video format used other than AVI}

FUNCTION: *RGBtoYUV()*

The first step in this process of encoding is transformation of video image of RGB (Red, Green, Blue) format to YUV(Y-Luminance, U&V-Chrominance) format.

An input frame in RGB format can be represented as a set of pixels as follows:

$inframe_i = \{pixel_1, pixel_2, pixel_3, ...pixel_n\}$

$pixel_i \subset inframe_i$ *is a subset of pixels from* $inframe_i$

where

$inframe_i$ *is an* i^{th} video frame

$pixel_1$ *to* $pixel_n$ are pixels in the frame

Each pixel has the three color component R, G and B and can be represented as a set of three color components as follows:

$pixel_i = \{\{pixel_{1R}, pixel_{1G}, pixel_{1B}\},\{pixel_{2R}, pixel_{2G}, pixel_{2B}\},\{pixel_{3R}, pixel_{3G}, pixel_{3B}\},.....\{pixel_{nR}, pixel_{nG}, pixel_{nB}\}\}$

$\{pixel_{iR}, pixel_{iG}, pixel_{iB}\} \subset pixel_i$ *is a subset of pixels with RGB components*

where

$\{pixel_{iR}, pixel_{iG}, pixel_{iB}\}$ *are the R, G and B components of an* i^{th} *pixel in the* $inframe_i$

Converted frame in YUV format can be represented as a set of pixels as follows:

$cframe_i = \{pixel_1, pixel_2, pixel_3, ...pixel_n\}$

$pixel_i \subset cframe_i$ *is a subset of pixels from converted YUV frame*$_i$

where

$cframe_i$ *is an* i^{th} *converted video frame in YUV format*

$pixel_1$ *to* $pixel_n$ *are pixels in the frame*

Each pixel has the three components Y, U and V and can be represented as a set of three components. The process of converting RGB frames into YUV frames is accomplished by following mathematical formulas:

$$pixel_{iY} = pixel_{iR} * 0.2990 + pixel_{iG} * 0.5870 + pixel_{iB} * 0.1140$$

$$pixel_{iU} = pixel_{iR} * -0.1470 + pixel_{iG} * -0.2890 + pixel_{iB}\ 0.4360 + 128$$

$$pixel_{iV} = pixel_{iR} * 0.6150 + pixel_{iG} * -0.515 + pixel_{iB} * -0.1000 + 128$$

FUNCTION: *SimpleDifference()*

The process of calculating the simple difference is accomplished by following mathematical formulas:

$df1f2y = pixel_{iY} - pixel_{i+1Y}$

$df1f2u = pixel_{iU} - pixel_{i+1U}$

$df1f2v = pixel_{iV} - pixel_{i+1V}$

FUNCTION: *RLE()*

Run length encoding (RLE) is a kind of lossless encoding. RLE In this method, the pixel values after Simple Difference *df1f2y, df1f2u, df1f2v* are scanned horizontally, and RLE byte streams RLE_y, RLE_u, RLE_v are generated.

CONCLUSION

Video compression using the proposed method described in this paper will definitely use less storage space and parallel implementation of the methods by using CUDA and GPU will help to process lakhs of pixels in parallel in less time as compared to time required by sequential implementation.

ACKNOWLEDGEMENT

We express our sincere thanks to Prof. M. V. Kulkarni, Vishwakarma Institute of Technology, Pune for motivating to do the project work on GPU and CUDA and providing the necessary material and guidance time to time. We would also like to thank Prof. Dr. M L Dhore, Vishwakarma Institute of Technology, Pune for their valuable guidance in the overall execution of this project work.

REFERENCES

[1] Yasuyuki Miura and Shogo Yamato " Simple Compression Method for Parallel Encoding Environment of Video Image", PACRIM, IEEE, 2009, pp. 284-289

[2] John D. Owens, Mike Houston, David Luebke, Simon Green, John E. Stone, and James C. Phillips, "GPU Computing, Graphics Processing Units - powerful, programmable, and highly parallel are increasingly targeting general-purpose computing applications" , Proceedings of the IEEE, Vol. 96, No. 5, May 2008, pp. 979-899

[3] Introduction to Parallel Programming Using CUDA to Harness the Power of GPUs [Online Course]. Available: https://www.udacity.com/course/cs344

[4] Zhiyi Yang, Yating Zhu and Yong Pu, "Parallel Image Processing Based on CUDA", International Conference on Computer Science and Software Engineering, CSSE 2008, Volume 3: Grid Computing / Distributed and Parallel Computing / Information Security, December 12-14, 2008, Wuhan, China pp. 198-201

[5] Lei Pan, Lixu Gu, and Jianrong Xu, "Implementation of Medical Image Segmentation in CUDA", Proceedings of the 5th International Conference on Information Technology and Application in Biomedicine, in conjunction with The 2nd International Symposium & Summer School on Biomedical and Health Engineering Shenzhen, China, May 30-31, 2008, pp 82-85

[6] Yasuyuki Miura, Sho Nakane and Shigeyoshi Watanabe, "Simple Compression Method Using Motion Vector of the Distributed Video Encoder System" , Journal of Communication and Computer, 2013, pp. 49-58

[7] Stamos Katsigiannis, Dimitris Maroulis, and Georgios Papaioannou, "A GPU based real-time video compression method for video conferencing", 18th International Conference on Digital Signal Processing, 2013, pp. 1-6

[8] Huayou Su,Mei Wen, NanWu, Ju Ren, and Chunyuan Zhang, "Efficient Parallel Video Processing Techniques on GPU: From Framework to Implementation", Hindawi Publishing Corporation, The Scientific World Journal Volume 2014, Article ID 716020, pp. 1-19