International Journal of Advanced Information Science and Technology (IJAIST) ISSN: 2319:2682
Vol.3, No.12, December 2014 DOI:10.15693/ijaist/2014.v3i12.69-73

Performance enhancement of H.264 video decoder using NVIDIA CUDA

Om Mehta Assistant Professor, Indus University, Ahmadabad, Gujarat India Jitiksha Patel, Assistant Professor,A.D. Patel Institute of Technology, Aanand, Gujarat India Jignesh Patel, Assistant Professor,Indus University,Ahmedabad,Gujarat,India

***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 paper, we have discussed the H.264 video compression standard 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).**

Index Terms—H.264, NVIDIA CUDA codec API, JM 18.4 decoder, Performance comparison.

I. INTRODUCTION

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.The standard is complex and therefore challenging to the engineer or designer who has to develop, program or interface with an H.264 codec.Computationally expensive, an H.264 codec can lead to slow coding and decoding times or rapid battery drain on handheld devices.[1]

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.

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).We are aiming to use CUDA for enhancing the performance of H.264 decoder using GPU.

In this paper we have discussed the H.264 decoder in brief in the section II. Introduction to Joint Model(JM) Reference software is given in section III. CUDA based implementation of proposed decoder is given in section IV and its results are presented and compared with JM 18.4 decoder in section V. Finally section VI draws the conclusion.

II. H.264 DECODER

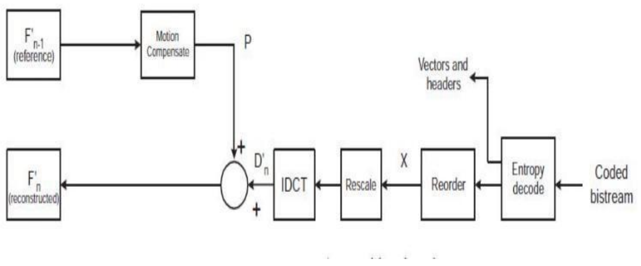


Fig. 1. Decoder [2]

1. • Entropy Decoder: The H.264 CODEC uses variable length entropy coding to encode integers. H.264 uses two techniques for this: CAVLC(Context Adaptive Variable Length Coding) and CABAC(Context Adaptive Binary Arithmetic Coding). Both techniques feature context- aware bit-mappings that vary during decoding. CABAC produces better compression but its complicated proba- bility models makes it more computationally intensive.[5]
2. • Inverse Transformation and Quantization: H.264, like many video CODECs, represents data via a fixed predict- tion, based on previously decoded image data, coupled with a residual error value representing the difference between the fixed prediction and the original image. This greatly enhances compression, since the prediction modes can be concisely expressed. In H.264 error-correction residual can be either 4x4 or 8x8 pixels (previous standards used only 8x8 blocks). Since residual data exhibits high spatial entropy, H.264 employs a lossy, low-pass discrete cosine transform to develop a compact representation of the residual values. H.264 also allows variable quantization of DCT coefficients to enhance coding density.[5]

International Journal of Advanced Information Science and Technology (IJAIST) ISSN: 2319:2682
Vol.3, No.12, December 2014 DOI:10.15693/ijaist/2014.v3i12.69-73

3. Intraprediction: Video frames have a high amount of spatial similarity. Intraprediction use previously decoded, spatially local macroblocks to predict the next macroblock. Intraprediction works well for low-detail images.[5]
4. • Interpretation: In video, frames nearby in time have only small differences. Interpretation attempts to capitalize on this similarity by encoding macroblocks in the current frame using a reference to a macroblock in a previous frame and a vector representing the movement that macroblock took to a 14 pixel granularity. The decode uses an interpolation process known as motion compensation to generate the prediction value. Fractional motion vectors are interpolated from multiple previous macroblocks[5].
5. • Deblocking Filter: Since loss compression used to en- code pixel blocks in H.264, decoding errors appear most visibly at the block boundaries. To remove these visual artifacts, the H.264 CODEC incorporates a smoothing filter into its encoding loop. However, not all inter- block discontinuities are undesirable; edges in the original image may naturally occur on block boundaries. H.264 incorporates fine-grained filter control to preserve these edges.[5]
6. • Buffer Control: H.264 does not require interpredicted images to depend on temporally-local, temporally- ordered images. Rather, frames can be predicted from previously decoded frames corresponding to frames far in the past or future of the video. Buffer control maintains a set of previously decoded frames and is responsible for handling the in stream requests to access (e.g.,delete, prediction logic reads, writes from deblocking) these frames in its store.[5]

We note in passing that H.264 decoding entails a large amount of computation. Most of these computations take place in four blocks Inverse Discrete cosine transform and Quantization, Inter and Intra prediction and the Deblocking filter.

III. JOINT MODEL REFERENCE SOFTWARE

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 2.

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

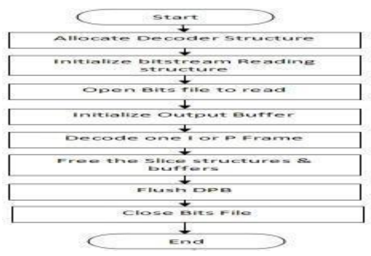


Fig. 2. JM 18.4 decoder Flow graph (1)

the slice structures and global buffers and it also closes bits file, else it keeps on reading the next frame data.

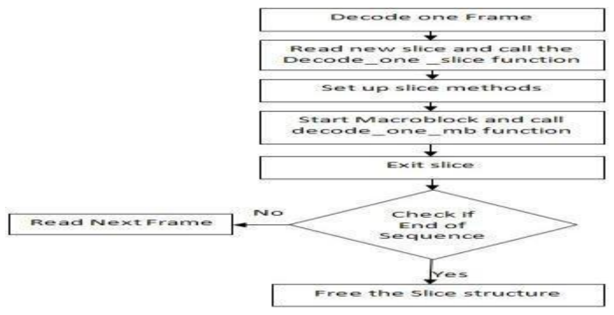


Fig. 3. JM 18.4 decoder Flow graph (2)

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 to GPU using CUDA API to be executed in parallel. Command to Execute the JM 18.4 Decoder:

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

IV.CUDA BASED VIDEO DECODER IMPLEMENTATION *A. Background of 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.[2]

1) Programming model: CUDA programming environment allows the GPU to be programmed through traditional CPU. It means you can use C++ language and compiler to realize operations on GPU. A fundamental building block of CUDA programs is the CUDA kernel function, which is a special C++ function. The kernel is downloaded to the GPU device that acts as a coprocessor to the CPU(host). Figure 4 is CUDA programming model.

The kernel function is executed by threads which are organized in a block. There are maximally 1024 threads in a block

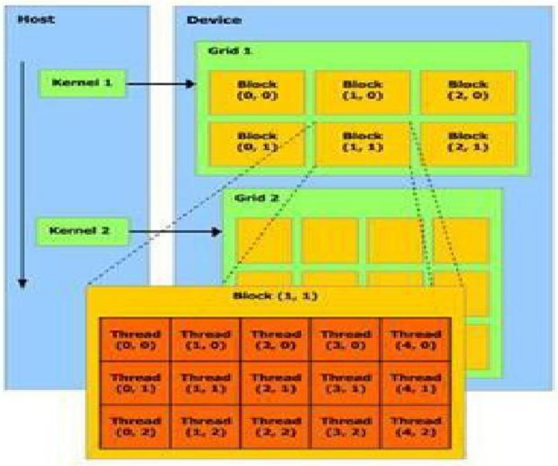


Fig. 4. CUDA programming model [3]

and the threads within a block can co-work with each other through the shared memory. Though the number of threads in a block are limited, we can co-operate multiple blocks or grid whose basic unit is a block to get enough threads for data paralle processing. Nevertheless, blocks within a grid cannot communicate with each other. The CUDA architecture provides access to three kinds of memory: Global Memory, Local Memory and Shared Memory. And memory instructions include any instruction that reads from or writes to shared, local or global memory. Global memory and local memory spaces are not cached. By contrast, share memory is on-chip,so it is much faster than any other two kinds of memories.

2) CUDA video decoder API: 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. The tasks that take major time, i.e. motion compensation, inverse discrete cosine transform, inverse quantization, VLD (variable-length decoding) and deblocking, can be offloaded to GPU using this API and can be executed in parallel[4]. The CUDA Video Decode API consists of:

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

B. CUDA Implementation

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 5.

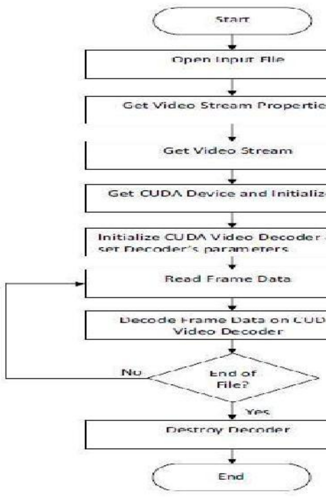

Start
Open Input File
Get Video Stream Properties
Get Video Stream
Get CUDA Device and Initialize it
Initialize CUDA Video Decoder & set Decoder's parameters
Read Frame Data
Decode Frame Data on CUDA Video Decoder
No
End of File?
Yes
Destroy Decoder
End

Fig. 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 cuvidparseVideoData( ) 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.

V. IMPLEMENTATION RESULTS

We have used various video files for testing our decoder. The list of the details of the video files is shown in Table I. 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 II.

- Input file type: H.264 file
- Output file type: yuv file

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 I
COMPARISON OF THE EXECUTION TIME

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

TABLE II
Video Files

The graphical presentation of the Execution time and the Frame rates of both decoders is shown in Figures 6 and 7. One can compare both decoders using these graphs.

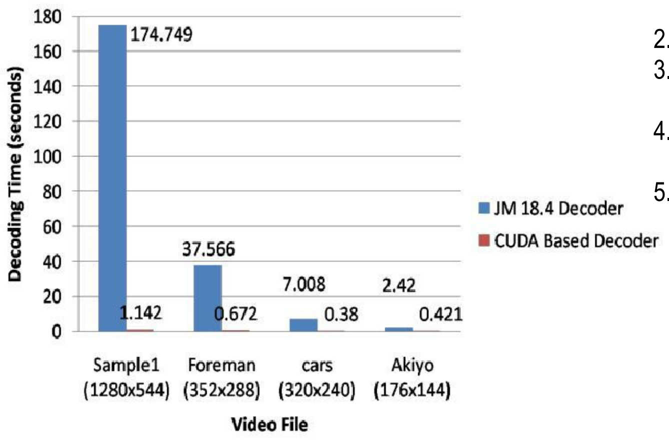
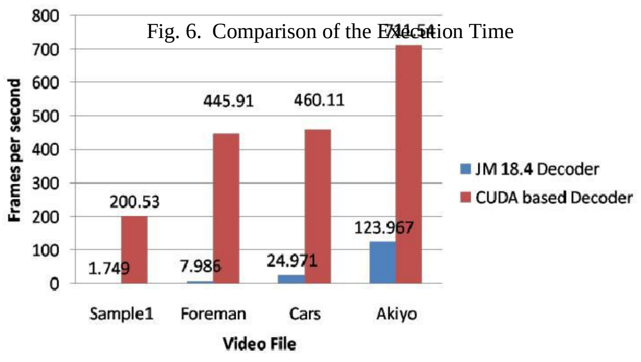


Fig. 6. Comparison of the Execution Time



From the graphs, it is clear that the implemented decoder provides better decoding frame rate than the JM 18.4.

VI. 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 Graphs. 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 quan-tization, 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.

REFERENCES

1. [1] Iain E. Richardson, The H.264 advanced video compression standard 2nd Edition ,2010.
2. [2] http://en.wikipedia.org/wiki/CUDA
3. [3] NVIDIAs Next Generation CUDA Compute Architecture:Fermi, White Paper, Version 1.1 ,2009.
4. [4] NVIDIA CUDA VIDEO DECODER API specification, Version 1.1 ,August 2010.
5. [5] Fleming, Chun-Chieh Lin, Dave, Arvind, Raghavan, Hicks, H.264 De- coder: A Case Study in Multiple Design Points, Version 1.1 , 2008.

Authors Profile

Om Mehta received the **B.E.** degree in information and technology from Vishwakarma Govt. Engg. College,Gujarat,India in 2010.Currently pursuing **PH.D.** from Indus University **in Computer Engineering.** His research interest includes wireless sensor netwoks,multimedia,operating system,fuzzy logic.

Jitiksha Patel received the **B.E.** degree in information and technology from A.D. Patel Institute of Technology in 2011. Currently working as an assistant professor in A.D. Patel Institute of Technology.His research interests are Maths , Multimedia ,Compilers ,Software Engineering.

Jignesh Patel received the **B.E.** degree in computer engineering H.N.G.U University in 2008. Currently working as an assistant professor in Indus University.His research interestsareParallelprocessing, Multimedia & Computer graphics Compilers ,Software Engineering.