Frame-Based Parallelization of MPEG-4 on Compute Unified Device Architecture (CUDA)

Dishant Ailawadi, Milan Kumar Mohapatra, Ankush Mittal
Department of Electronics & Computer Engineering,
Indian Institute of Technology Roorkee, India.
{disntumt,milanupt,ankumfec}@iitr.ernet.in

Abstract

Due to its object based nature, flexible features and provision for user interaction, MPEG-4 encoder is highly suitable for parallelization. The most critical and time-consuming operation of encoder is motion estimation. Nvidia's general-purpose graphical processing unit (GPGPU) architecture allows for a massively parallel stream processor model at a very cheap price (in a few thousands Rupees). However synchronization of parallel calculations and repeated device to host data transfer is a major challenge in parallelizing motion estimation on CUDA. Our solution employs optimized and balanced parallelization of motion estimation on CUDA. This paper discusses about frame-based parallelization wherein parallelization is done at two levels – at macroblock level and at search range level. We propose a further division of macroblock to optimize parallelization. Our algorithm supports real-time processing and streaming for key applications such as e-learning, telemedicine and video-surveillance systems, as demonstrated by experimental results.

Keywords – MPEG-4, CUDA, motion estimation, Frame-based parallelization, real-time processing.

1. Introduction

As a result of continued demand for programmability, modern graphics processing units (GPUs) such as the NVIDIA GeForce8 Series are designed as programmable processors employing a large number of processor cores [11]. This makes an inexpensive, highly parallel system available to a broader community of application developers.

Video encoding has emerged out to be a common and highly resource intensive process for today's users who are dealing with huge amounts of data. Video encoding is a highly time consuming prospect on modern standard CPU hardware. All compression-decompression schemes (codecs), including the MPEG standards, are concerned with using fewer data or less bandwidth to store or transmit pictures and sound. An MPEG-2 encoder expects as an input a complete picture repeating at the frame rate whereas an MPEG-4 encoder can handle the graphic instructions directly. The motion of a virtual object could be described by one vector. As a result, the amount of data transmitted reduces significantly. With the advent of real time video transmission in fields like telemedicine and video surveillance, the transmission times as well as the quality of video have become a factor of prime importance. Uses of parallel architectures like CUDA have drastically reduced the overall cost in the video transmission. Encoding becomes highly efficient due to the high computation capability of GPU, having multiple cores and large memory bandwidth.

Attempts have been made for parallelization of other video compression standards on CUDA. In [8] the authors were able to achieve 1.47 times speed up for H.264/AVC motion estimation, however they acknowledge "A serial dependence between motion estimation of macroblocks in a video frame is removed to enable parallel execution of the motion estimation code. Although this modification changes the output of the program, it is allowed within the H.264 standard." In [1] significant results were obtained for H.264 standard.

Several Works [1,2,3,7,15] have been done in past to exploit the parallel architectures for parallelizing video compression standards. [7] discusses about real-time parallelization of MPEG-1 on Paragon supercomputer. As MPEG-4 is more computationally intensive than MPEG-1, if such supercomputing co-processor is employed with our GPU based processing, very useful results could be obtained. In [14] heterogeneous MP-SoC architecture is used for MPEG-4 parallelization.

In this paper, emphasis is laid on cost-effective parallelization which could be useful in real life applications. This is ensured by use of cheaper GPUs, readily available for few thousands and use of application based video sequences in our experiments. We propose a frame based parallel implementation of motion estimation part of MPEG-4.

2. Introduction to MPEG-4

MPEG-4 video is an object-based hybrid natural and synthetic coding standard which specifies the technologies enabling the functionalities such as content-based interactivity, efficient compression, error resilience, and object scalability [4]. MPEG-4 is an ISO/IEC standard developed by MPEG (Moving Picture Experts Group) [5]. MPEG-4 compresses images by dealing with objects in the video rather than interpreting individual frame. Frames are divided into sub-parts called blocks. A block usually has a similar pattern with their neighbor which is called spatial redundancy. This redundancy can be reduced by predicting the pixel values of one block from its neighboring blocks in the same frame and compensate errors with residuals. Reducing spatial redundancy does not require much computation, but the compression efficiency is relatively low.

The major challenge lies with the temporal redundancy. In video, pixel values often do not change much across scenes (frames), and thus the pixel values of a block can be predicted from the values of a similar block in a previous frame (reference frame). The motion estimation is an operation of finding a motion vector, which indicates the similar block in the reference frame, and residuals to compensate prediction errors. To find the motion vector, the motion estimation algorithm finds the best match in a certain window of the reference frame. Inter coded blocks yield a high compression ratio, but require heavy computations to find the best match. Parallelization of this motion estimation part would significantly affect the encoding time, and hence result in faster transmission of videos.
Further details on MPEG-4 video encoder can be found in [5].

3. Introduction to Compute Unified Device Architecture (CUDA)

NVIDIA introduced CUDA in November 2006 which is a general purpose parallel computing architecture-with a new programming model and instruction set architecture –that leverages the parallel compute engine in NVIDIA GPUs to solve many complex computational problems in a more efficient manner than on a CPU [12].

A *kernel* is simply C code for one thread of the hierarchy. Each multiprocessor has on-chip memory of the four following types:

- One set of local 32-bit *registers* per processor,
- A parallel data cache or *shared memory* that is shared by all scalar processor cores and is where the shared memory space resides,
- A read-only *constant cache* that is shared by all scalar processor cores and speeds up reads from the constant memory space, which is a read-only region of device memory,
- A read-only *texture cache* that is shared by all scalar processor cores and speeds up reads from the texture memory space, which is a read-only region of device.

When a CUDA program on the host CPU invokes a kernel grid, the blocks of the grid are enumerated and distributed to multiprocessors with available execution capacity. The threads of a thread block execute concurrently on one multiprocessor. As thread blocks terminate, new blocks are launched on the vacated multiprocessors. For more details, readers can refer to [13].

4. Components of MPEG-4 video encoder

The MPEG-4 encoder mainly consists of following modules:
1. Motion Estimation- Motion vectors and SAD values for each macroblock(MB) are calculated.
2. Frame Statistics- It calculates a pseudo-variance quantity for I MBs in P frames detection and calculate overall scene brightness for scene change detection.
3. Motion Compensation- The prediction (residual) error is calculated by both inter and intra compensation.
4. Transform –The prediction from the frame is subtracted and discrete cosine transform (DCT) is performed.
5. Quantize –the DCT coefficients are Quantized and the variable length code (VLC) bit stream data is generated.
6. I-Quantize–The DCT Coefficients are then inverse quantized.
7. I-Transform –The inverse DCT is performed and is added to the prediction.

On time profiling it is found that motion estimation takes around 91% of the time, which is the bottleneck of our code. The prediction of frames using motion compensation takes around 2% of the time. Subsequent transforms and quantization accounts for around 5% of total time. Rest of the portion of the code like calculating frame statistics correspond to about 2%.

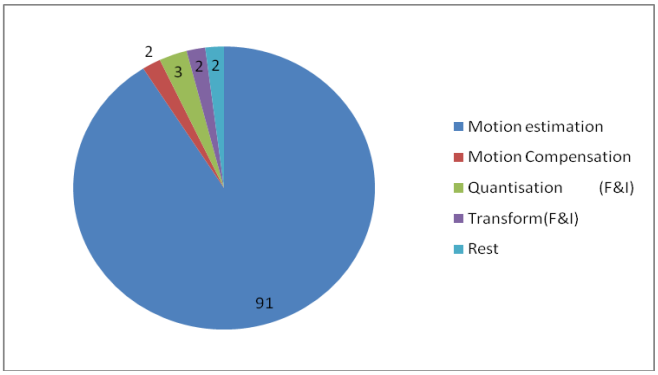


Fig 1. Time profiling of the MPEG-4 encoder.

Table 1. Comparison of various components of MPEG-4 encoder with varying search ranges and decimation factor. (All Results in secs)

Test Sequence	*Search range/Decimation*	*Total Time*	*Motion Esti.*	*Motion comp.*	*F-Quant.*	*F-Transform*	*I-Transform*	*I-Quant.*	*Residual*
AKIYO	8/1	14.76	12.854	0.522	0.094	0.085	0.203	0.211	0.03
176×144 pixels	16/1	43.421	40.434	0.589	0.277	0.228	0.383	0.279	0.097
	16/2	14.281	12.811	0.519	0.092	0.077	0.211	0.208	0.031
	32/2	38.937	36.123	0.532	0.251	0.221	0.345	0.234	0.078
FOREMAN	8/1	62.078	56.322	1.968	0.398	0.358	0.411	0.421	0.122
352 × 288 pixels	16/1	190.781	172.66	1.749	0.978	0.928	0.526	0.487	0.298
	16/2	63.022	57.887	1.889	0.397	0.346	0.419	0.419	0.125
	32/2	177.453	162.029	1.741	0.973	0.922	0.498	0.477	0.268

In motion estimation performed on a set of pixels (block based) every frame is divided into blocks of equal size and for each block in the current frame a search is performed in the reference frame.

Because a search is performed over whole reference frame for each block in the current frame, it is computational intensive and movements in video sequences are usually small, the search is limited to a search area [6]. After finding the best match for current block, the motion vector (the displacement relative to current block) is stored together with differences between the two blocks.

In determining which block in the searching area of reference frame is best match for the block in current block we use Sum of Absolute Difference (SAD).

$$d(J,K) = \sum_{i=1}^{n} \left| X(J,i) - X(K,i) \right|$$

Here J is some reference frame used to calculate SAD from current frame K.

5. Implementation

5.1. MPEG-4 Motion Estimation

The temporal prediction technique used in MPEG video is based on motion estimation. The basic premise of motion estimation is that in most cases, consecutive video frames will be similar except for changes induced by objects moving within the frames. In the trivial case of zero motion between frames (and no other differences caused by noise, etc.), it is easy for the encoder to efficiently predict the current frame as a duplicate of the prediction frame. When this is done, the only information necessary to transmit to the decoder becomes the syntactic overhead necessary to reconstruct the picture from the original reference frame. When there is motion in the images, the situation is not as simple. In a way it is a process of finding lowest cost representation (in terms of size) of a given macroblock's motion. Calculation of the motion estimation involves defining motion estimation search range and the decimation factor. This can greatly vary the time required to encode a given frame [9]. Experiments performed on different test sequences are tabularised in Table 1. It is observed that increasing the search range and decreasing the decimation factor increases time taken by motion estimation component which is also reflected in total time taken. Based on the above calculations, motion estimation was estimated to be the most feasible module for parallelization.

5.2. Parallelization of Motion Estimation on CUDA

As it has been discussed before that the basic aim is achieving real-time processing of video stream, implementation of parallelization technique is to be kept independent of number of frames to be encoded. This ensures real-time processing wherein frames can be continuously processed as they are made available. Each frame is itself divided into 16X16 pixel macroblocks. These macroblocks are fundamental units used to predict the motion vector of a particular object in the frame. However in our implementation we further subdivide these macroblocks into 8X8 blocks to exploit the large number of registers and texture referencing provided by NVIDIA CUDA architecture [1].

The sequential code was modified for CUDA implementation as it involved memory reference and computational instructions one after another. This was broken to two sections one of which was computationally intensive SAD calculations (still involved few memory fetches) and other majorly involved memory instructions for storing values calculated. This paved the way for efficient exploitation of CUDA programming.

Concepts of decimation and Motion estimation range have been discussed in last section, on which time required by motion estimation is dependent. For our implementation on CUDA we take search range from -16 to +16 with a decimation factor of 1. This makes motion estimation a significant contributor to the encoding process. The SAD calculation process further involves a loop for calculation of SAD row by row.

Pseudo-code Motion-estimation

```
MOTION-ESTIMATION (CurrentFrame, ReferenceFrame)

B ← No. of Macroblocks in Frame
While  B ≠ NIL
      Divide Macroblock into 4 8X8 blocks
      Assign block Ids to sub-blocks B0,B1,B2,B3
      Motion-Vector ← (0,0)
      SAD ← ∞ (A very high value)
      SAD0 ← ∞
      SAD1 ← ∞
      SAD2 ← ∞
      SAD3 ← ∞
      While  encMERangeYlo<encMERangeYhi
         While  encMERangeXlo<encMERangeXhi
           SAD0←SAD-Calculation(B0,CurrFrame,RefFrame)
           SAD0←SAD-Calculation(B1,CurrFrame,RefFrame)
           SAD0←SAD-Calculation(B2,CurrFrame,RefFrame)
           SAD0←SAD-Calculation(B3,CurrFrame,RefFrame)
           SAD ← SAD0 +SAD1+SAD2+SAD3
             If SAD < previous SAD
                If previous vector = zero vector
                     If Beats with ZeroMVBias considered
                          MBlock SAD ← SAD
                Else Previous Vector != zero vector
                          MBlock SAD ← SAD
             encMERangeXhi←encMERangeXhi-DecimationX
         encMERangeYhi← encMERangeYhi -DecimationY
      In Global array bSAD and SAD
      bSAD[B0] ← SAD0
      bSAD[B1] ← SAD1
      bSAD[B2] ← SAD2
      bSAD[B3] ← SAD3
      SAD[B] ← MBlockSAD
      bMV[B]= Motion-vector
      B ← B – 1
```

Parallelization involves targeting the loops involved in the process. In our implementation we have two possible solutions:-

1. **Parallelization on macroblock level:** While loop1 in the pseudo-code iterates for all macroblocks. For a typical CIF 352×288 frame size this will invoke 1584 threads if calculation is based on 8×8 sub-blocks. But as it can be seen that this loop basically involves data storage and memory transfer , it is not viable to parallelize at this level due to high latency in data transfer from host to device and device to host[13]. Moreover Quadro Fx 370 GPU uses two multiprocessors with maximum of 768 active threads per processor. Parallelization of this loop requires two cycles of multiprocessor operation, which will delay the processing time.
2. **Parallelization on search range**: While loop 2 & 3 in pseudo-code iterates for the search range specified. This is highly computational and requires only host to device data transfer which is well accomplished by read-only texture memory, therefore in our implementation it is parallelized.

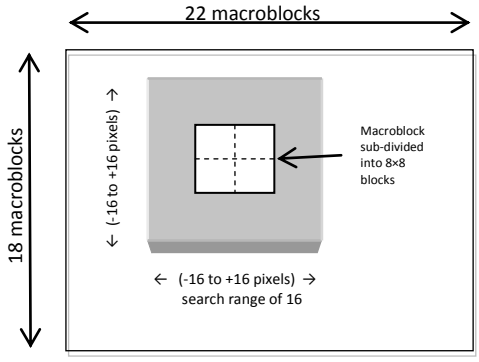


Fig 2: Motion estimation search range depiction for 352×288 Cif image with 16/1 search range.

The threads invoked thus are dependent on motion estimation range and decimation factor. Threads are invoked in parallel for whole frame. Results of calculation are stored in global arrays through a small loop in CPU.

Moreover the function **SAD-Calculation** is called as a *__device__* function. This further helps in improving the performance. SAD-Calculation process needs reference and source frames which require memory transfer. While our current implementation does not take advantage of shared memory to preload the needed pixel values, we do store both the current frame and the reference (previous) frame in texture memory. This helps to avoid global memory access. It also allows for cheap sub-pixel interpolation (searching for non-integer motion vectors) because the texture provides interpolated access in hardware. The texture memory space is cached so a texture fetch costs one memory read from device memory only on a cache miss, otherwise it just costs one read from the texture cache. The source and reference frames are first stored in texture memory in each iteration. This speeds up the memory transfer for **SAD-Calculation**.

5.3. Theoretical evaluation of parallelization model

Based on the approach and algorithm we may theoretically predict the optimization order expected for various sequences. The concept utilized in prediction is simply the parallelization of kernel task into n-parts and utilizing time taken by one part.

We will restrict ourselves to search range of -16 to +16 and a decimation factor of 1 unit on CIF-352×288 video sequences. For calculations we require approximate number of computations being done in SAD- calculation for one frame. After the **SAD-calculation()** function call is being made, SAD of 8 rows consisting of 8 pixels per row is being calculated. This involves total 8 subtraction operations per row and 16 memory references per row, 8 each for reference and current frames. Besides that addition of differences to calculate final SAD and other comparison costs are also there.

From [12] we know that a comparison and addition operation in CUDA takes 4 cycles, whereas multiplication takes 16 cycles. Latency and pipelining concept of Pentium architecture can be found in [10]. Theoretically time taken by motion estimation should be defined by following equation:

$$T = \frac{F \times MB \times Sx \times Sy \times SB \times R \times (Sub \times LoS + Load \times LoL)}{Freq.}$$

F= Total number of frames.
MB= Number of macroblocks in a frame.
S*x*= search range in x-direction.
S*y*=search range in y-direction.
SB= No. of 8×8 sub-blocks.
R=No. of pixels in each row of sub-block.
Sub= No. of subtraction operation in each SAD-Calc.
LoS=Latency of subtraction operation in clock cycles.
Load = No. of memory references in each SAD-Calc.
LoL= Latency of load operation in clock cycles.
Freq.= Clock frequency of the processor used.

Assuming issue time to be same as latency, an approximate analysis of motion estimation process for our model can be calculated as:-

$$T = \frac{300 \times (22 \times 18) \times 32 \times 32 \times 4 \times 8 \times (8 \times 1 + 16 \times 3)}{3 \times 10^9}$$

So the time in taken in computation of motion estimation by this model should be 72.66 secs. This is close to actual results obtained (Table 1) and will be useful in computing the achievable speed up by model.

As only second and third While loops (refer pseudo-code) are computational, on parallelization their computation time will be evaluated by following equation:

$$T = \frac{F \times MB \times SB \times R \times (Sub \times LoS + Load \times LoL)}{Freq.}$$

So, for our parallelized model

$$T = \frac{300 \times (22 \times 18) \times 4 \times 8 \times (8 \times 4 + 16 \times 4)}{603 \times 10^6}$$

This evaluates to 0.6 secs. So the theoretical bound on motion estimation parallelization for this particular approach is 72.66/0.6=121x. In next section we observe that actual experimental result produce less optimization. This can be accounted for the memory latency and other computations which we have not considered while theoretical calculations. This theoretical approach can prove to be a good tool to approximate maximum possible results.

6. Results and Discussion

Experiments were performed on both the processors described in Table2. The Video sequences used in experimentation are application specific. We have used five standard video sequences emphasizing different applications as summarized below. Each of them in CIF (352 × 288)[1] , 4:2:0 chroma format and YUV QCIF (176 × 144)[1], 4:2:0 chroma format. All files included 300 frames.

Test sequence	*Properties*
Akiyo	Rich in still scenes, useful in e-learning
Foreman	Rich in facial expressions and lip-movements, useful for tele-conferencing.
Hall	Includes indoor video surveillance.
Coast-guard	Sea-based video
Silent	Deaf sign language usage.

Table 2: Test specifications

Processor	**Intel Core2**	**NVIDIA Quadro FX 370**
Cores	dual	1 + 2
Clock	3 Ghz	603 MHz
Memory	3.68GB DDR2	256MB RAM
OS	Windows XP	Windows XP
Compiler	Microsoft Visual C++ 2005	Nvcc

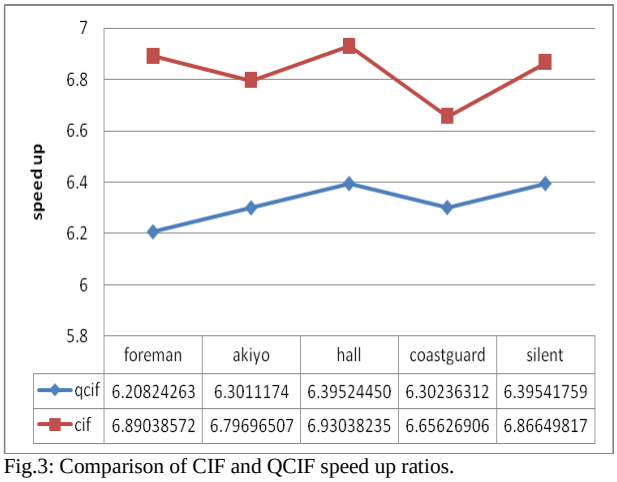
Table 3: Result compilation for Cif format(time in secs)

Test sequence	*Host CPU*	*NVIDIA*	*Speed Up*
Foreman	190.781	27.688	6.89x
Akiyo	182.75	26.906	6.79x
Hall	185.062	26.703	6.93x
Coast-guard	185.437	27.859	6.65x
Silent	184.75	26.906	6.866x

Table 4: Result compilation for Qcif format(time in secs)

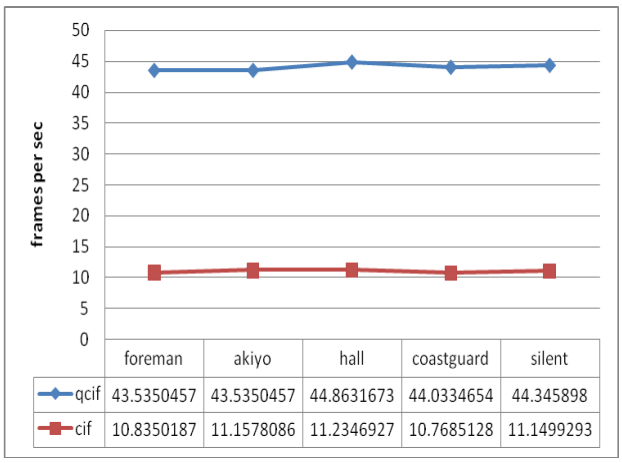
Test sequence	*Host CPU*	*NVIDIA*	*Speed Up*
Foreman	42.781	6.891	6.2x
Akiyo	43.421	6.891	6.3x
Hall	42.765	6.687	6.39x
Coast-guard	42.938	6.813	6.30x
Silent	43.265	6.765	6.39x

[1] Although picture resolutions specified by CCIR601 (720 × 480 or 720 × 576) are often used for MPEG-4 implementations, our experiments involved smaller sized pictures described above. However, since our implementation is scalable, it can be performed on pictures of any resolution.



	foreman	akiyo	hall	coastguard	silent
qcif	6.20824263	6.3011174	6.39524450	6.30236312	6.39541759
cif	6.89038572	6.79696507	6.93038235	6.65626906	6.86649817

Fig.3: Comparison of CIF and QCIF speed up ratios.



	foreman	akiyo	hall	coastguard	silent
qcif	43.5350457	43.5350457	44.8631673	44.0334654	44.345898
cif	10.8350187	11.1578086	11.2346927	10.7685128	11.1499293

Fig. 4: Graphical representation of fps achieved after parallelization

For our implementation on CUDA we take search range from -16 to +16 with a decimation factor of 1. This makes motion estimation a significant contributor to the encoding process. The results obtained after parallelization are summarized in Table 3. All qcif format sequences have been optimised to real time, whereas cif sequences are also close to real time applications. The processor frequency and other specifications are given in Table 2. Intel Core2 was also used as co-processor (host) with NVIDIA GPU. Note that there is no performance loss due to parallelization since, in our parallel programming model, various MPEG-4 modules in each processor use the same computational logic that is used in the sequential version. Although some issues which will arise on parallelization are

- If sum of SAD values till k^{th} row (k<8) exceeds the current SAD value, then there is no need to compute further. But due to concurrent processing no best SAD is available at to the threads. This leads to extra computations.
- Numbers of kernel calls depend on decimation factor and search range. So if different portions of sequence are requesting for varying decimation or search range, that is not possible.
- Results may vary upon varying decimation factor and search range.

It is observable from fig.3. that speed up obtained for CIF format is higher than qcif format. Fig.4. analyses the frame rate of different sequences.

7. Conclusions

We have tested several video sequences on an open source MPEG-4 sequential encoder with the final goal of getting real-time, good quality and cheaper video compression. Study of some important issues like how to access the input data, how to synchronize concurrent SAD processing, how to avoid the extra calculations if a row SAD exceeds current best SAD were important issues dealt in the paper. At the same time the correctness of the implementation was proven as no compromises were made for parallel implementation. We have concluded that a simple parallel algorithm based on decomposing the input video in blocks that are processed independently gives good results.

Due to the high level of computation involved in the motion estimation part, significant speed-up was achieved on CUDA than on CPU by data and task parallelization. In order to solve the problem of performing motion estimation in a massively parallel environment we needed to deal with the issue of needing the synchronization between various threads to find the optimal motion vector. Further speed up can be obtained by utilizing high frequency GPUs, but this will be costlier; or by varying search range with lesser decimation factor for some of the macroblocks. This process can be employed to Group of Pictures (GoPs) to obtain more optimization, though it's very challenging due to high dependency among various modules.

8. Acknowledgement

The authors acknowledge the contribution of Institute Computer Centre, IIT Roorkee for providing resources and facilities of NVIDIA GPU and Documentation.

9. References

[1] Wei-Nien Chen and Hsueh-Ming Hang, "H.264/AVC Motion Estimation Implementation on Compute Unified Device Architecture (CUDA).", *2008 IEEE International Conference on Multimedia and Expo*, April 26 2008, pp. 697–700.

[2] Yong He, Ishfaq Ahmad, and Ming L. Liou, "A Software-Based MPEG-4 Video Encoder Using Parallel Processing*", IEEE Transactions On Circuits And Systems For Video Technology, Vol. 8, No. 7,* November *1998* pp. 909-920.

[3] Yong He, Ishfaq Ahmad, and Ming L. Liou, "Real-Time Interactive MPEG-4 System Encoder Using a Cluster of

Workstations", *IEEE Transactions On Multimedia, Vol. 1, No. 2, June* 1999, pp.217-233.
[4] T. Sikora, "The MPEG-4 video standard verification model", *IEEE Trans. Circuits Syst.Video Technol.*, vol. 7, Feb. 1997,pp. 19–31.
[5] ISO/IEC, "MPEG-4 video verification model version 8.0," ISO/IEC JTC1/SC29/WG11 N1796, July 1997.
[6] Peter M. Kuhn , Kuhn Peter M., *Algorithms, Complexity Analysis and VLSI Architectures for MPEG-4 Motion Estimation*, Kluwer Academic Publishers, Norwell, MA, 1999.
[7] Ke Shen and Edward J. Delp, "A Spatial-Temporal Parallel Approach For Real-Time MPEG Video Compression", *Proceedings of the 25th International Conference on Parallel Processing*, 1996, pp. 100—107.
[8] Shane Ryoo, Christopher I. Rodrigues, Sara S. Baghsorkhi, Sam S.Stone, David B. Kirk, and Wen-mei W. Hwu, "Optimization principles and application performance evaluation of a multithreaded gpu using cuda", *In PPoPP '08: Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming, New York, NY, USA, 2008. ACM.*, pp. 73–82.
[9] Yankang Wang, Yanqun Wang, and Hideo Kuroda, "A Globally Adaptive Pixel-Decimation Algorithm for Block-Motion Estimation",*IEEE Transactions On Circuits And Systems For Video Technology, Vol. 10, NO. 6,* September 2000, pp.1006-1011.
[10] Randal E. Bryant, David R. O'Hallaron, *Computer Systems, A Programmers perspective,* Pearson Inc., India.
[11] J. Owens, *Streaming architectures and technology trends*, GPU Gems 2, March 2005.
[12]NVIDIA CUDA *Programming Guide ver 2.0* .
[13]NVIDIA Corp. CUDA Zone., *http://www.nvidia.com.*
[14] Bonaciu, M.; Bouchhima, S.; Youssef, W.; Xi Chen; Cesario, W.; Jerraya, A., "High-level architecture exploration for MPEG4 encoder with custom parameters," *Design Automation, 2006. Asia and South Pacific Conference on* , vol., no., pp. 6 pp.-, 24-27 Jan. 2006
[15] Jike Chong; Satish, N.; Catanzaro, B.; Ravindran, K.; Keutzer, K., "Efficient Parallelization of H.264 Decoding with Macro Block Level Scheduling," Multimedia and Expo, 2007 IEEE International Conference on , vol., no., pp.1874-1877, 2-5 July 2007