Scalable Parallel Motion Estimation on Muti-GPU system

Dong Chen, Huayou Su, Wen Mei, Lixuan Wang, Chunyuan Zhang
Computer School
National University of Defense Technology
ChangSha, China
chendong@nudt.edu.cn

***Abstract*—With NVIDIA's parallel computing architecture CUDA, using GPU to speed up compute-intensive applications has become a research focus in recent years. In this paper, we proposed a scalable method for multi-GPU system to accelerate motion estimation algorithm, which is the most time consuming process in video encoding. Based on the analysis of data dependency and multi-GPU architecture, a parallel computing model and a communication model are designed. We tested our parallel algorithm and analyzed the performance with 10 standard video sequences in different resolutions using 4 NVIDIA GTX460 GPUs, and calculated the overall speedup. Our results show that a speedup of 36.1 times using 1 GPU and more than 120 times for 4 GPUs on 1920x1080 sequences. Further, our parallel algorithm demonstrated the potential of nearly linear speedup according to the number of GPUs in the system.**

Keywords-scalable;motion estimation;full search;multi-GPU;

I. INTRODUCTION

From SIGGRAPH 2003 Conference which first proposed general purpose GPU computing to NVIDIA's CUDA and Fermi architecture for now, GPU is playing a more and more important role in video processing, physics simulation, computational finance, computational biology and many other fields which require massive parallel computing. Applications based on camera array such as 3D video and free-view point TV are becoming a part of our lives. The high-definition and multi-view video will inevitably increases the encoding complexity and the amount of computation. Video encoding in real time is even more challenging.

Motion estimation is the most compute-intensive component in video encoding, whose computation is 50% to 90% of the entire encoding system. So an efficient implementation of motion estimation is essential. Current research mainly focuses on two aspects: one is to improve the performance and PSNR by modify the searching strategy of the serial algorithm, but the effect is limited; the other is to take advantage of parallel computing, the effect is obvious. Our study belongs to the latter approach. Before CUDA proposed, Yu-Cheng Lin used graphics API to accelerate motion estimation module on GPU in 2006 [1]. After CUDA proposed, Wei-Nien Chen implement parallel motion estimation algorithm using CUDA and achieved 12 times faster than the serial program [2]. Bart Pieters created a novel job scheduling system for motion estimation which supports multi-GPU system in 2009 [3]. Xinbiao Gan proposed a parallel full search motion estimation algorithm using CUDA based on one GPU system in 2010 [4]. Compared to the previous work, we modified and parallelized the full search motion estimation algorithm targeted on multi-GPU system. Our algorithm can automatically balance the workload and take full advantage of multi-GPU's computing power without lowering the quality of the video obviously. Our result shows, on a multi-GPU system, the speedups can be linear to the number of GPUs.

The rest of this paper is organized as follows. Section II briefly provides an introduction to motion estimation and GPU computing. Section III describes the parallel motion estimation algorithm in detail using two models: computing model on each GPU and collaborative model between GPUs. The parallel algorithm is tested and analyzed using standard test sequences in Section IV. The paper is summarized in Section V.

II. MOTION ESTIMATION AND GPU COMPUTING

Motion estimation is first to divide each video frame into a number of non-overlapping macroblocks and assume all the pixels in the same macroblock have the same displacement. Second, for each macroblock using a certain matching rule to find the most similar macroblock in the given reference frame within a specific search range. Finally the relative displacement of the current macroblock and its matching macroblock is the motion vector. During the video compression, only motion vectors and the residual data are saved. According to the saved data we can recover the original frame macroblock by macroblock. In the current video coding standard, many motion estimation algorithms are being used, such as: full search, spiral search, three-step search, etc. Our work is based on one of the modified full search algorithms – Lower Resolution Full Search [8].

Lower Resolution Full Search (LRFS) should get the frame's pyramid representation first. Pyramid representation is an array of images in different resolutions. The images are created by sampling or averaging the pixel values of each block in the original frame. Different block size leads to different resolutions. Then we array the images from low resolution to high resolution, forming a pyramid-like image sequence. In this paper, we average 4 pixel values in each 2x2 block to gain an N/2xN/2 image from an NxN frame and then gain an N/4xN/4 image from an N/2xN/2 image.

Once we obtain the pyramid representation, we can start full search in the lowest resolution image which is on the top

Published by Atlantis Press, Paris, France.

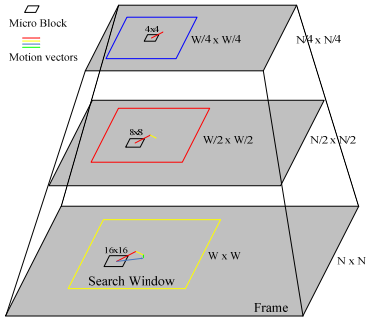


Figure 1. The process of Lower Resolution Full Search.

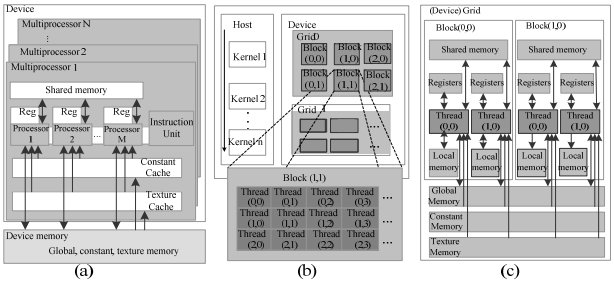


Figure 2. Hardware struture, programming model and memory heriechy of CUDA.

of the pyramid. The center of each macroblock's search window is the same position as current macroblock. With the motion vectors of each macroblock, we can continue to do the next full search in the higher resolution image in the next layer of the pyramid. But the center of each macroblock's search window is where the motion vectors gained from upper layer of the pyramid pointing to. Iterating like this until we get the motion vectors of the original frame at the bottom of the pyramid. As Fig. 1 shows, we first do the full search on the N/4xN/4 image. The blue search window's center is the macroblock and the motion vector we gained is marked red. Then we do the next full search on the next layer. The red motion vector pointed to the center of the search window which we marked red. We gain another motion vector marked yellow. Finally we do the full search on the original frame the same way and gain the final motion vector marked blue.

GPU is first to be used as Graphic Processing Unit. Compared to CPU, GPU has more computational logic, more floating-point computing power and more parallel computing power. Not until the concept of General Purpose GPU computing is proposed and the CUDA architecture is widely used, GPU extends the limited application field of image processing to various fields which need large-scale data parallel computing.

CUDA architecture is built on a scalable array of multi-threaded Stream Multiprocessors. As Fig. 2(a) shows, the hardware structure of GPU consists of N Stream Multiprocessors and the corresponding memory hierarchy. Each Stream Multiprocessor contains M stream Processors works in SIMD to allow data-level parallelism and multiple Stream Multiprocessors work in MIMT to make instruction-level parallelism possible.

CUDA architecture enables parallel computing process running on GPU by Kernel functions in the program. Due to the hardware structure, CUDA program provides two-level-parallelism, (1) thread-level parallelism: multiple threads running on multiple Stream Processors in parallel and (2) block-level parallelism: multiple thread blocks running on multiple Stream Multiprocessors. This allows CUDA program to handle the fine-grained and coarse-grained data parallelism flexibly. The programming model is shown by Fig. 2(b).

CUDA architecture has multiple memory spaces which can be accessed by the executing threads. Fig. 2(c) shows the memory hierarchy of CUDA. Each thread has private local memory. Each thread block has an on-chip shared memory which is visible to all the threads in the block and can also be used as a cache for the global memory to reduce the global memory access latency. In addition, two other read-only memories are provided, texture memory and constant memory, which enhanced the programming flexibility.

III. Parallel Algorithm on Multi-GPU System

In the parallelization process of the LRFS algorithm, we designed a parallel computing model for each GPU and a collaborative model for multi-GPU environment. Based on the hardware structure of GPU, we first weaken the data correlation of the LRFS algorithm. Then we use three sub-models to build the parallel computing model: SAD parallel computing sub-model, motion vector parallel selecting sub-model and memory accessing sub-model. Collaborative model is designed to balance the workload of each GPU. It allows the host to finish data partitioning and reconstruction automatically. No redundant data are introduced.

A. Parallel Computing Model for each GPU

- Weaken Data Correlation

In the serial LRFS algorithm, motion vector of current macroblock is based on the left, up-left and up macroblocks' motion vectors because of the predict motion vector, so motion estimation must be done in row-major order by CPU as Fig. 3(a) shows. In order to weaken data dependency and at the same time maintain the compression efficiency, we divided the whole frame into several sub sections. The marcoblocks within a sub-section have no data dependency with the marcoblocks in other sub-sections. The estimation order is shown by Fig. 3(b). Due to this sub-section division, we can make full use of the multiprocessors in different GPUs to process different sub-sections in parallel.

Based on sub section division, we use the following three sub-models to describe the parallel computing model.

- SAD Parallel Computing Sub-model

SAD is the sum of absolute differences between the pixel values of the macroblock in the original frame and its matching macroblock within the search window in the reference frame. It is often used to evaluate similarity of two

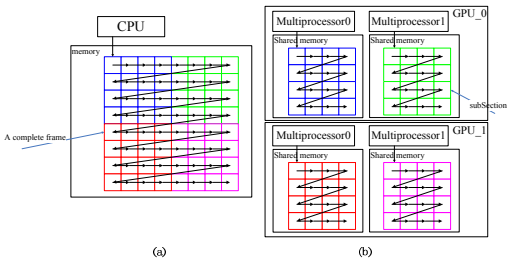


Figure 3. Processing order before and after the sub-section division.

macroblocks. Two marcoblocks' SAD is independent of other marcoblocks, so the SADs of different macroblock in the original frame can be calculated in parallel and each SAD between the same original macroblock and the different matching macroblock in its search window can also be calculated in parallel. Fig. 4 shows the two-level SAD parallel calculation model: macroblock level introduces the parallelism of different original macroblocks; pixel level introduces the parallelism of different matching marcoblocks in the same search window.

In this sub-model, one SAD between two macroblocks can be calculated using one GPU thread in spite of the block size. Hundreds of threads running at the same time can make the process parallel without communication. But for the macroblocks which contain a lot of pixels such as 16x16 or 8x8, only using one thread to process all the calculations to gain the SAD will make the workload of the thread too heavy. In another way, if we split the parallelism by each pixel, there will be too many threads and the startup cost will greatly increase. Therefore, considering the balance between the startup cost and the workload of the threads, we finally choose 4x4 block size for one thread. That is to say, each thread will calculate 16 pixel values. And for the macroblock larger than 4x4 such as 16x16 or 8x8, we will use several threads to work together. Fig. 5 shows four-thread computation model for 8x8 macroblock.

- Motion Vector Parallel Selecting Sub-model

According to the SAD parallel computing sub-model, all the SADs of different matching macroblocks in the same search window can be calculated at the same time. And then we should compare the SADs to get the best motion vector of the original macroblock. So we use parallel reduction which can cut half of the data each time. During each parallel

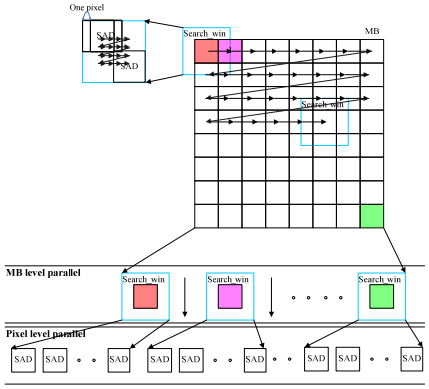


Figure 4. Two-level SAD parallel calculation model.

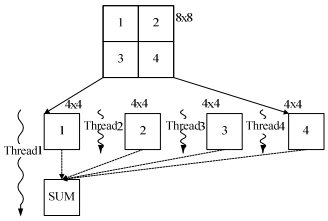


Figure 5. Four-thread calculation model for 8x8 marcoblock.

reduction, the SAD values in the first half of the threads will compare with those in the second half of threads in pairs and save the small SADs and their motion vectors in the first half, then we divide the first half of threads into another two halves and repeat the previous procedure until only one thread left. Fig. 6 shows the procedure of 8 SADs' parallel reduction.

- Memory Accessing Sub-model

The data which GPU needs for computation is the pixel values of the original frame and the reference frame. Before the computation process, the data should be copied to the global memory. During the computation process, Each Stream Multiprocessor will access the global memory to get one sub-section of the original frame and the corresponding search windows. The original frame data are loaded macroblock by macroblock, but the reference frame data are loaded search window by search window. As Fig. 7 shows, there are no overlapping data in loading the original macroblock, but the adjacent search windows are largely overlapped. Most of the areas in the reference frame will be reloaded two or four times from global memory. The data reloading cost is high. Considering this cost, we merge the search windows corresponding to one sub-section into a common search window and use multiple threads to load it in parallel into shared memory. In this way, we only need to access the global memory once and the other accesses will be done by share memory which is 10 times faster than global memory.

B. Collaborative Model for multi-GPU system

- CPU Threads Organization

In CUDA architecture, One CPU thread can only manage one GPU device. We create multiple host threads to manage

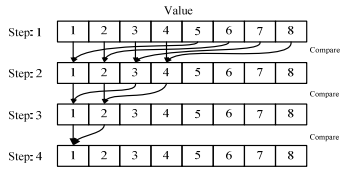


Figure 6. The procedure of 8 SADs' parallel reduction

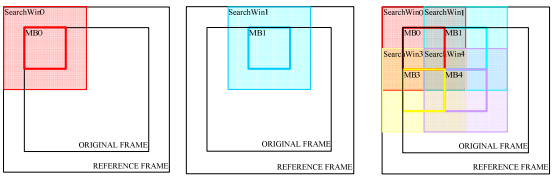


Figure 7. The overlap of the search windows

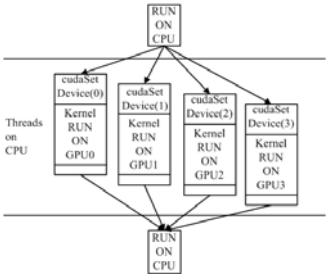


Figure 8. Four CPU threads manage four GPUs

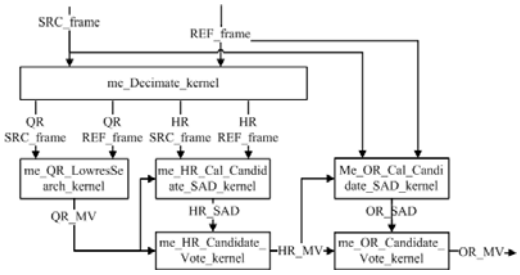


Figure 10. CUDA implementation of the parallel algorithm

the contexts of multiple GPUs. The number of threads is the same as the number of GPUs in the system. All host threads can start almost at the same time. As Fig. 8 shows, the host starts four CPU threads to make four GPUs work in parallel.

- Dynamic Data Partition and Reconstruction

If we want to use multiple GPUs to process a frame, we should use the same number of CPU threads to control the multiple GPUs. So each CPU thread should contain all the data which will be processed by the GPU it managed. Although we can achieve this simply by transfer the entire frame into each CPU threads, this will certainly cause a lot of unnecessary data transmission. So when we start multiple CPU threads, we should divide the data according to the number of GPU devices and merge the results after the parallel processing. In order to ensure there is no communication among GPUs, we divide the frame in the unit of sub-section.

Fig. 9 shows the partition procedure for original frame and the reference frame. The divided original frame slices for each GPU have no overlapping data and the adjacent divided reference frame slices have overlapping data because of the search window. And the amount of overlapping data can be calculated by the following equation.

$$OD = (GN - 1) * SWH * RFW$$

OD is the Overlapping data. GN is the GPU number. SWH is the height of the search window. RFW is the width of the reference frame.

IV. Experiments and Results

In our CUDA implementation, we use the following six Kernel functions to complete the whole parallel algorithm. As Fig. 10 shows, me_Decimate_kernel is used to generate half and quarter resolution images. The following five QR, HR and OR Kernel functions are used to finish the three full searches. We separate the full search process into the SAD computing Kernel and the parallel reduction Kernel in the calculation of the half and the original resolution images. In order to synchronous the data of different GPUs, all the Kernel functions communicate through main memory.

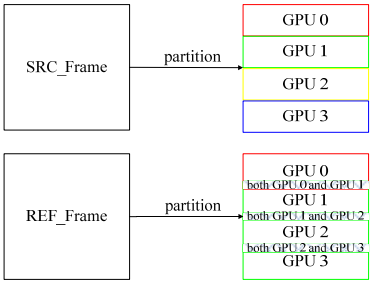


Figure 9. Data partition procedure

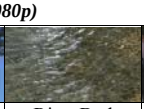
The configuration of the experiment environment is shown as Table I and ten standard video sequences are used, listed in Table II.

PSNR is the most widely used objective video quality metric. In order to make sure that the parallel algorithm is not at the cost of declining the quality of the compacted videos, we firstly evaluate the PSNR of each video processed by serial LRFS algorithm and the PSNR of each video processed by our parallel algorithm as Fig. 11 listed. We can see that there is no obvious decline of the video quality and even increase in some sequences owing to larger search range. The decrease of the PSNR is less than 0.1 DB. For QP less 32, the quality of the parallel algorithm is stable.

TABLE I. Experiment Environment

CPU	16 x Intel® Xeon® CPU E5620 @ 2.4GHz
GPU	4 x GTX460: 7 multiprocessors, 1 GB Global memory
Memory	4 x 4GB
Software	CUDA 4.2 driver, toolkit, SDK

TABLE II. Standard Video Test Sequences

Standard video test sequences				
1920x1080 (1080p)				
				
ParkJoy (PJ)	RushHour (RH)	BlueSky (BS)	RiverBed (RB)	PesestrainArea (PA)
1280x720 (720p)				
				
MobcalTer (MT)	ParkRun (PR)	DuckTakeOff (DTO)	OldTownCross (OTC)	IntoTree (IT)

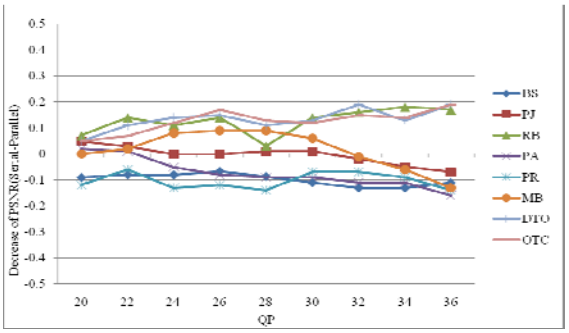


Figure 11. Quality Evaluation of Two Algorithms

Then we run the serial algorithm on CPU and the parallel algorithm on multiple GPUs. The processing time and speedups are listed in Fig. 12 Parallel algorithm on one GPU can achieve more than 30 times faster than serial algorithm on CPU for all the ten sequences. From the figure, we can see the parallel implementation on one GPU can achieve real-time processing for HD H.264. The performance gains from the massively parallel algorithm. For the most computational component, SAD calculation, the computation on all the search windows is independent and can be parallelized. More important, the scalability of our parallel realization is very well. The Fig. 13 shows the scalability of our parallel estimation. From the figure, the near linear speedup can be obtain with the number of the GPUs. Two GPUs may expect to double the performance, but the overlapping data, the CPU threads startup cost and GPU startup cost may somehow reduce the performance. The more GPUs we use, the more overlapping data and startup cost are introduced. In addition, the communication between CPU and GPU is growing with the number of the GPUs. For 720p, the proportion of communication time is higher than that of 1080p, so its speedup is lower than result of 1080p sequences.

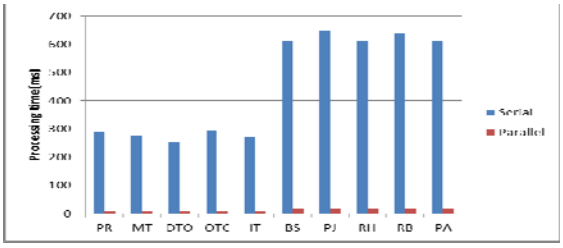


Figure 12. Performance on single GPU

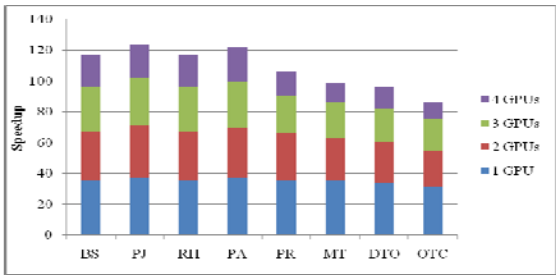


Figure 13. Scalability of Parallel Motion Estimation

V. CONCLUSION

In this paper, we designed a parallel motion estimation algorithm for multi-GPU system. This algorithm is based on cutting the data correlation of the serial LRFS algorithm. This cut is targeted on GPU's hardware structure and it makes the algorithm become scalable for multi-GPU system. Then we proposed the basic computing sub-models for parallel calculation and the memory accessing sub-model for efficiency in the parallel computing model. A common search window is designed to take the advantage of the shared memory of Stream Multiprocessor. A collaborative model is used to divide the input data and merge the output results of multiple GPUs automatically. With this model, Workload balance is ensured and no redundant data are introduced. We tested the CUDA implementation on a server with four GPUs. It can achieve almost linear speed up according to the number of GPUs in the system.

This method can be a reference for the parallelization of the other core algorithms in video encoding and also for applications which require large amount of calculations.

ACKNOWLEDGMENT

R.B.G thanks for the supports from National Nature Science Foundation of China under NSFC No. 61033008, 61272145, 60903041 and 61103080, Research Fund for the Doctoral Program of Higher Education of China under SRFDP No. 2010430711002, Hunan Provincial Innovation Foundation for Postgraduate under No.CX2010B028, Fund of Innovation in Graduate School of NUDT under No.B100603, No.B120605.

REFERENCES

[1] Yu-Cheng Lin, Pei-Lun Li, Chin-Hsiang Chang, Chi-Ling Wu, You-Ming Tsao, and Shao-Yi Chien, "Multi-Pass algorithm of motion estimation in video encoding for generic GPU," Proc. IEEE Symp. Circuits and Systems. Proceedings. 21-24 May 2006, pp. 4451-4454.

[2] Wei-Nien Chen and Hsueh-Ming Hang, "H.264/AVC motion estimation implementation on compute unified device architecture (CUDA)," IEEE International Conference on Multimedia and Expo(ICME08), June 23 2008-April 26 2008 , pp. 697-700.

[3] Bart Pieters, Charles F. Hollemeersch, Peter Lambert, and Rik Van de Walle, "Motion estimation for H.264/AVC on multiple GPUs using NVIDIA CUDA," Applications of Digital Image Processing XXXII, August 2-5 2009, Vol. 7743 77430X-2.

[4] Gan Xinbiao, Shen Li, and Wang Zhiying, "Parallel full search algorithm for motion estimation using CUDA", Journal of Computer-Aided Design and Computer Graphics, vol. 22, Mar. 2010, pp. 457-460.

[5] Youngsub Ko, Youngmin Yi,and Soonhoi Ha, "An efficient parallel motion estimation algorithm and X264 parallelization in CUDA," Design and Architectures for Signal and Image Processing(DASIP), 2-4 Nov. 2011, pp. 1-8.

[6] M.C. Kung, Oscar C. Au, P.H.W. Wong, Chun Hung Liu, "Block based parallel motion estimation using programmable graphics hardware," International Conference on Audio, Language and Image Processing (ICALIP 08), 7-9 July 2008. pp.599-603.

[7] Ngai-Man Cheung, Xiaopeng Fan, Oscar C. Au, Man-Cheung Kung, "Video coding on multicore graphics processors," Signal Processing Magazine , vol. 27, 2010, pp.79-89.

[8] Lee S, Kim JM, Chae S, "New motion estimation using low-resolution quantization for MPEG2 video encoding," IEEE Workshop on VLSI Signal Processing, 20 Oct 1996.