Multi-Pass Algorithm of Motion Estimation in Video Encoding for Generic GPU

Yu-Cheng Lin, Pei-Lun Li, Chin-Hsiang Chang, Chi-Ling Wu, You-Ming Tsao, and Shao-Yi Chien
Media IC and System Lab
Graduate Institute of Electronics Engineering and Department of Electrical Engineering
National Taiwan University
yucheng@media.ee.ntu.edu.tw

***Abstract*— The importance of video encoding has boomed rapidly since video data communication was widely needed. In this paper, we propose a multi-pass algorithm to accelerate the motion estimation (ME), the dominant part in video encoding, with the graphics processing unit (GPU). By the multi-pass method to unroll and rearrange the multiple nested loops, the complex ME can be implemented on GPU. Besides, ME can be executed efficiently with the built-in parallel processing and texture filter of GPU. Experimental results show that, by utilizing the computing power of GPU, about two times and 14 times speed-up can be achieved for integer-pel ME and MPEG-1/2 half-pel ME, respectively.**

I. Introduction

Digital entertainment with multimedia content, is definitely one of the most important applications in the twenty-first century. Fancy multimedia content, especially video content, usually accompanies huge volume of data. Consequently, video encoding and decoding play key roles to the most multimedia applications. Thanks to the development of modern technologies, digital still cameras (DSC) and digital video recorders (DV) have become common and popular consumer electronics products. Everyone could be a multimedia content provider as long as he or she has a DSC or DV. High resolution video equipments are occupying more and more market share in addition. In the 3G mobile era, there are potential multimedia content providers everywhere. As a result of the above reasons, video encoding has become as important as video decoding has been.

Comparing the computation profile of MPEG-4 in Table. I [1] and H.264 in Table. II [2], it is evident that motion estimation is the most significant part in video encoding. That is, improving the performance of motion estimation would actually speed up the whole video encoding. Thus the motion estimation is chosen to be accelerated as the starting point of real-time video encoding.

On the other hand, more and more commodity PC and game consoles are commonly equipped with graphics processing units (GPUs). Graphics processors are designed to perform a mass of operations on a crowd of vertices or pixels, and they do this very efficiently because of their inherent multiple parallel pipelines. In the recent years, graphics processors have changed from fixed pipelines to programmable pipelines. Figure 1 shows that the vertex and fragment shader could be programmed to make changes in the routine pipelines [3] [4]. A GPU is no longer with fixed pipeline but is now more appropriately described as a SIMD parallel processor or a streaming processor [5].

TABLE I
Instruction Profiling Results of MPEG-4 Video Encoder.

Tools	Datapath Operation (MIPS)	Percentage(%)
Motion Estimation	24,768.2	97.94
Transform & Quant	432.527	1.710
Others	88.0320	0.348

TABLE II
Instruction Profiling Results of H.264 Encoder.

Tools	Datapath Operation (MIPS)	Percentage(%)
Motion Estimation	152,897	94.66
Transform & Quant	5,402.50	3.345
Others	3,225.30	1.997

Moreover, the performance of GPU evolves much faster than the famous Moore's law for the performance of CPU, that is, 2.4 times/year versus 2 times per 18 months [6] [7]. With all these advantages over CPU, especially SIMD operation and parallel processing, there is an active research area of using GPU for nongraphics oriented operations, such as FFT [6] and motion compensation [7]. However, due to the instruction limit of GPU and large input data amount, the complex motion

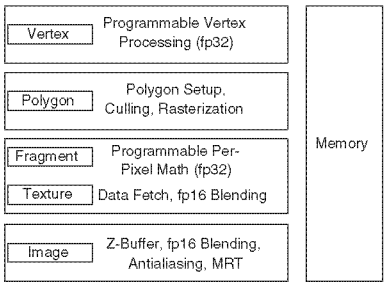

Fig. 1. A programmable graphics pipeline [4].

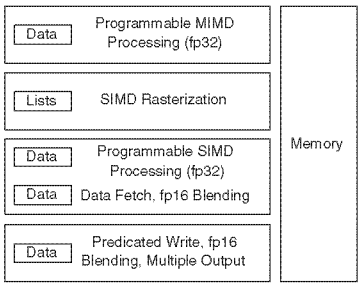

Fig. 2. The GeForce 6 Series architecture for non-graphics applications [4].

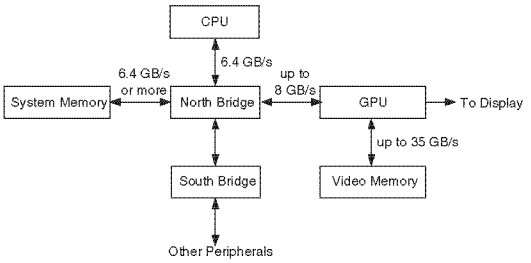

Fig. 3. The overall system architecture of a PC [4].

```
Loop(rows of macro blocks (MBs)){
    Loop(columns of MBs){
        Loop(rows of search range (SR)){
            Loop(columns of SR){
                SAD computation;
                SAD comparison;
            }
        }
    }
}
```

Fig. 4. The pseudo code of the classic integer-pel ME algorithm.

estimation algorithm, which is the most important in video encoding, has never been implemented on GPU. Besides, the time-consuming interpolation computing of fractional motion estimation is another challenge for speed-up.

In the past, motion estimation has been implemented by use of dedicated hardware of multiple parallel processing elements (PEs) [8] [9] [10]. The graphics processors nowadays also have similar parallel architecture because of the graphics processing requirements. Furthermore, motion estimation is block-wise and thus suitable for SIMD-like GPU processing. In conventional approach, the powerful GPU is idle when no graphics-related process is going on. It is natural to leverage GPU to off-load some of the CPU's tasks to achieve the maximum co-throughput of CPU and GPU. In this paper, we propose an algorithm to map motion estimation (ME) on generic commodity GPU to accelerate the video encoding. We expect to accelerate motion estimation with GPU and would accomplish a real-time video encoding system hereafter. With unrolling and rearranging loops of the motion estimation, the ME could be implemented on GPU in the limited instruction number with multi-pass technique. Second, the large input data would be bound as textures in graphics pipeline to reduce the memory access time. Finally, GPU has inherent built-in interpolation engine, which readily accelerates the fractional interpolation process. The implementation detail would be shown in the following section. Note that, nVidia GeForce 6800, a famous GPU, is used as an example in this paper.

The organization of this paper is shown below. The proposed motion estimation algorithm is described in Sec. II. Next, in Sec. III, the experimental results will be shown. Finally, a short conclusion is given in Sec. IV.

II. Motion Estimation Implementation

A. General-Purpose Computation on GPUs

When used for non-graphics applications, the nVidia GeForce 6 Series could be viewed as two programmable blocks that run serially: the vertex shader for MIMD processing and the fragment shader for SIMD processing, both with support for fp32 operands and intermediate values [4]. The architecture is shown in Fig. 2. As shown in Fig. 3, the memory bandwidth between the GPU and the video memory is about five times as fast as that between the GPU and the system memory. The texture unit could be used as a random-access data fetch unit to make use of the astonishing 35 GB/sec memory bandwidth.

B. Implementation Environment

We implemented the motion estimation (ME) algorithm on a nVidia GeForce 6800 GT graphics card, which features fully programmable vertex and fragment units, 32-bit floating point frame buffers and textures. The fragment processor can sustain 12 floating-point operations per pixel per clock cycle [11]. Both the vertex and fragment shaders were programmed with OpenGL and the Cg computer language and runtime libraries of nVidia Corporation [12] [13]. The ME is inherently an image-based algorithm. As such, we perform the ME by executing several fragment programs successively in a SIMD-like fashion.

C. A Novel Motion Estimation Algorithm

The motion estimation algorithm is intrinsically complex and time-consuming as a result of its multi-loop nature. Besides the sum of absolute difference (SAD) computation and SAD comparison, there are four loops in the classic ME algorithm, as shown in Fig. 4 [10]. In our proposed ME algorithm shown in Fig. 5, the four classic nested loops are unrolled and rearranged. Abstractly, the operations on all the pixels could be thought as executed in parallel. Therefore, the inner loop of processing elements is unrolled and executed almost simultaneously. The original four-tiered loop are transformed into a one-tiered loop and a two-tiered loop. The computation complexity is reduced and the parallelism is increased in this way. In fact, there are only 16 pipelines in GeForce 6800. All the operations would be scheduled, folded, and executed as concurrently as possible in the 16 pipelines. Thus we could still think of each pixel of fragment shader as a PE performed in parallel. We refer to the PE-like pixel as PE and the pixel of the texture as pixel here. It is highly recommended to include as many instructions as possible in [3]. However, the total instruction amount of full-search

```
Pass 1:
Loop(candidates per processing element){
    Loop(processing elements){
        SAD computation;
        SAD comparison;
    }
}

Pass 2:
Loop(rows of a MB){
    Loop(columns of a MB){
        Loop(processing elements){
            SAD comparison;
        }
    }
}
```

Fig. 5. The pseudo code of the proposed integer-pel ME algorithm.

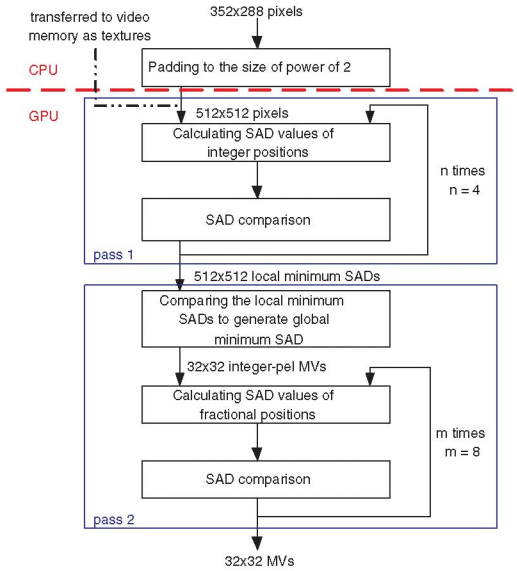

Fig. 6. The block diagram of the proposed ME algorithm.

block matching motion estimation is much more than the instruction limit of GeForce 6800 GT. Therefore, a multi-pass ME algorithm on GPU is proposed. We perform the proposed ME algorithm by executing two fragment programs in two passes. We draw quadrilaterals to invoke fragment programs for every PE. Figure 6 shows the block diagram of the three passes we proposed to implement the ME on GPU. We take the sequence in CIF format (352x288) for illustration convenience in Fig. 7. The hollow circle points represent candidates in the search range while the other opaque circle points represent PEs.

1) The First Pass: Generating Local Minimum SADs: The whole video frames are first transferred from system memory to video memory as textures to make use of the large memory bandwidth in the following operations. The size of the texture must be power of two. Thus we prepare a padded reference frame as well as a padded current frame. Both of the frames equal to $W_T \times H_T$. Next, a quadrilateral which equals to $W_{p1} \times H_{p1}$ is drawn, which can be viewed as $W_{p1} \times H_{p1}$ PEs. We take $MB_{row} \times MB_{col}$ pixels as a macroblock (MB) and let the search range (SR) equal to $SR_{row} \times SR_{col}$. Taking all the

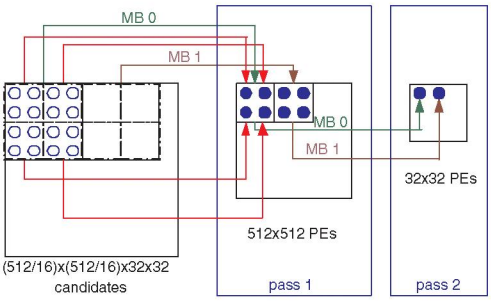

Fig. 7. The illustration of the proposed ME algorithm.

candidates in the search range of every MB in a frame into consideration, the formula for the number of candidates per PE is shown below:

$$n = \frac{W_T}{MB_{col}} \times \frac{H_T}{MB_{row}} \times SR_{row} \times SR_{col} \times \frac{1}{W_{p1}} \times \frac{1}{H_{p1}} \quad (1)$$

Take $W_T = H_T = 512$, $MB_{row} = MB_{col} = 16$, $SR_{row} = SR_{col} = [-16, +15] = 32$, $W_{p1} = H_{p1} = 512$ as the illustration of Fig. 7. The result is n = 4, which means that every four candidates are grouped and mapped to a PE in the corresponding MB. We use texture coordinates to perfectly allot four differential candidates so that every candidate is taken into account. This is the only loop in the first pass besides the two loops of SAD computation. Each PE compares the sum of absolute difference (SAD) of the four candidates and writes the motion vector with the smallest SAD value to the texture by using the render-to-texture extensions. Note that, in Fig. 7, only four PEs are shown for a MB in the first pass, instead of 256 PEs in this example case, for simplification.

2) The Second Pass: Generating Global Minimum SADs As Well As Integer-Pel MVs and Generating Fractional-Pel MVs: The remained computation amount for the integer-pel ME is much smaller in pass 2. Thus each MB only needs to be processed by a PE. The formula follows:

$$Quad\ size\ of\ pass\ 2 = \frac{W_T}{MB_{col}} \times \frac{H_T}{MB_{row}} \quad (2)$$

In the second pass, each PE compares the remained local minimum SAD values computed from the first pass in each MB, transferred via the texture, to find the smallest global minimum SAD values and the corresponding motion vector. The number of local minimum SADs compared per PE is as follows:

$$W_{p1} \times H_{p1} \times \left(\frac{W_T}{MB_{col}} \times \frac{H_T}{MB_{row}}\right)^{-1} \quad (3)$$

In the above example as in Fig. 7, the quadrilateral size of pass 2 would be 32x32, and 256 local minimum SAD values would be compared to find a integer-pel motion vector. Then the fractional-pel ME is computed if needed. The fractional-pel motion estimation is similar to the integer-pel motion estimation. After integer-pel ME is completed, this pass takes the integer-pel MVs and both the reference and current frames as inputs to find the smallest SADs and the corresponding MVs in the neighboring fractional search candidate positions. The

TABLE III
THE COMPARISON OF INTEGER-PEL ME BETWEEN CPU AND GPU

Search range	Frame rate (s) (CPU only)	Frame rate (s) (CPU +GPU)	Speed-up
16x16	6.451	11.57	1.794
32x32	1.694	3.709	2.189
48x48	0.795	1.492	1.877
64x64	0.462	0.673	1.457
80x80	0.308	0.400	1.299
96x96	0.220	0.240	1.091

TABLE IV
THE COMPARISON OF HALF-PEL ME BETWEEN CPU AND GPU

Search range	Frame rate (s) (CPU only)	Frame rate (s) (CPU +GPU)	Speed-up
16x16	0.805	7.275	9.037
32x32	0.217	3.091	14.24
48x48	0.102	1.374	13.47
64x64	0.06	0.65	10.83
80x80	0.04	0.393	9.825
96x96	0.028	0.236	8.429

fragment processor uses the texture unit to fetch data from memory, optionally filtering the data before returning it to the fragment processor [4]. Thus the built-in texture filter is used to do the bilinear interpolation for the fractional candidates.

3) Future Optimization: From the resources' point of view, the integer-pel ME must be divided into two passes due to the instruction limit of GPU and the large data amount of video encoding. As the instruction capacity grows, the two passes of integer-pel ME may be combined to one pass. More passes might be needed if quarter-pel ME is included. For optimization consideration, all the loops of the two passes could be unrolled and distributed averagely to achieve the maximum performance.

III. EXPERIMENTAL RESULTS

We performed extensive tests on a PC with an Intel Pentium IV 3.00 GHz CPU, 1024 MB memory, and a nVidia GeForce 6800 GT GPU with 128 MB video memory. Video sequence Stefan in CIF (352x288) format with 30 fps with the architecture in Fig. 7 is used as an example to measure processing time.

We compare the motion estimation speed achieved using CPU only against that achieved with GPU acceleration. Six different search ranges are tested for integer-pel and fractional-pel motion estimation. The experimental results are shown in Table III for integer-pel ME and Table IV for half-pel ME of MPEG-1/2. Full search algorithm is employed, and there is no quality degradation in PSNR for ME on GPU. It is interesting to observe that the speed-up of 32x32 search range reaches the local maximum. That is, about two times and 14 times of speed-up can be achieved respectively for integer-pel and half-pel ME. The bilinear interpolation is readily accomplished with the help of the built-in texture filter. Thus the speed-up of the half-pel ME is much more significant than the speed-up of the integer-pel ME. However, as the search range exceeds 32x32, the speed-up declines. The most possible reason is that the speed-up might depend greatly on the memory bandwidth requirement. The larger computation amount and memory bandwidth requirement always accompany the bigger search range, which might be the cause of performance degradation.

IV. CONCLUSION AND FUTURE WORK

In this paper, we have demonstrated that GPU can accelerate motion estimation together with CPU. The ME could be run on GPU about two times as fast as run on CPU in 32x32 search range. Our proposed algorithm outperforms the classic software-based motion estimation computation because of fully utilizing GPU's parallel computation power. As we are composing this paper, a nVidia GeForce 7800 GT GPU with 256 MB video memory has been on sale on the market. The performance degradation problem may be solved using the more powerful GPU.

Our future work is to further investigate the power of GPU and think thoroughly how to partition the workloads in video encoding between CPU and GPU. The whole software-based video encoding architecture need to be accomplished as well. Combined with the video decoding system [7], this work would develop a complete real-time video codec system in the near future.

REFERENCES

[1] H.-C. Chang, L.-G. Chen, M.-Y. Hsu, and Y.-C. Chang, "Performance analysis and architecture evaluation of mpeg-4 video codec system," in *Proc. IEEE International Symposium on Circuits and Systems*, May 2000, pp. 449–452.

[2] S.-Y. Chien, Y.-W. Huang, C.-Y. Chen, H. H. Chen, and L.-G. Chen, "Hardware architecture design of video compression for multimedia communication systems," *IEEE Commun. Mag.*, vol. 43, no. 8, pp. 122–131, Aug. 2005.

[3] C. J. Thompson, S. Hahn, and M. Oskin, "Using modern graphics architectures for general-purpose computing: a framework and analysis," in *Proc. IEEE/ACM International Symposium on Microarchitecture*, Nov. 2002, pp. 306–317.

[4] M. Pharr, Ed., *GPU Gems 2*. Addison Wesley, 2005.

[5] B. Khailany, W. Dally, U. Kapasi, P. Mattson, J. Namkoong, J. Owens, B. Towles, A. Chang, and S. Rixner, "Imagine: media processing with streams," *IEEE Micro*, vol. 21, no. 2, pp. 35–46, Mar. 2001.

[6] K. Moreland and E. Angel, "Simulation and computation: The FFT on a GPU," in *Proc. ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware*, July 2003, pp. 112–119.

[7] G. Shen, G.-P. Gao, S. Li, H.-Y. Shum, and Y.-Q. Zhang, "Accelerate video decoding with generic GPU," *IEEE Trans. Circuits Syst. Video Technol.*, vol. 15, no. 5, pp. 685–693, May 2005.

[8] K.-M. Yang, M.-T. Sun, and L. Wu, "A family of VLSI designs for the motion compensation block-matching algorithm," *IEEE Trans. Circuits Syst.*, vol. 36, no. 10, pp. 1317–1325, Oct. 1989.

[9] Y.-S. Jehng, L.-G. Chen, and T.-D. Chiueh, "An efficient and simple VLSI tree architecture for motion estimation algorithms," *IEEE Trans. Signal Processing*, vol. 41, no. 2, pp. 889–900, Feb. 1993.

[10] H. Yeo and Y. H. Hu, "A novel modular systolic array architecture for full-search block matching motion estimation," *IEEE Trans. Circuits Syst. Video Technol.*, vol. 5, no. 5, pp. 407–416, Oct. 1995.

[11] J. Montrym and H. Moreton, "The GeForce 6800," *IEEE Micro*, vol. 25, no. 2, pp. 41–51, Mar. 2005.

[12] R. Fernando and M. J. Kilgard, *The Cg Tutorial*. Addison Wesley, 2003.

[13] (2004) The GPGPU website. [Online]. Available: http://www.gpgpu.org/