MOTION ESTIMATION FOR H.264/AVC USING PROGRAMMABLE GRAPHICS HARDWARE

Chi-Wang Ho[†] *Oscar C. Au*[‡] *S.-H. Gary Chan*[†] *Shu-Kei Yip*[‡] *Hoi-Ming Wong*[‡]

[†]Dept. of Computer Science [‡]Dept. of Electrical and Electronic Engineering
The Hong Kong University of Science and Technology,
Clear Water Bay, Kowloon, Hong Kong, China.
Email: {jodyho, eeau, gchan, sukiyip, hoimingw}@ust.hk

ABSTRACT

We present an efficient implementation of motion estimation (ME) for H.264/AVC using programmable graphics hardware. The cost function for ME in H.264/AVC depends on the motion vector (MV) predictor which is the median MV of three neighboring coded blocks. Previous implementations assume no dependency among adjacent blocks, which is not true for H.264/AVC, they also perform unsatisfactorily because of their low arithmetic intensity, which is defined as operation per word transferred. To overcome the dependency problem, we introduce a new implementation which performs ME on block-by-block basis. Moreover, we can adjust the arithmetic intensity easily to optimize the performance on different graphics cards. Experimental results show that our implementation is substantially faster (by 10 times) than our SIMD optimized CPU implementation.

1. INTRODUCTION

H.264/AVC is the most current international video coding standard [1] which significantly improves the compression efficiency compared with existing standards, such as H.263+ and MPEG-4. To achieve such a high coding efficiency, it comes with a set of new tools to enhance the ability to predict the picture content in the cost of additional complexity. These tools include variable block-size motion compensation (MC), quarter-pixel accuracy MC, etc. There has been much of work on the complexity reduction. Besides this, Single Instruction Multiple Data (SIMD) extensions have been developed for central processing unit (CPU) to dramatically enhance the performance in multimedia applications. However, for high-definition video encoding, it is still difficult to process in real-time on CPU even with highly optimized code.

Recently, consumer graphics hardware has become increasingly more powerful. It is equipped with a powerful graphics processing unit (GPU), a stream processor and specialized for processing graphics operations in parallel. In term of raw computation power, it far surpasses that of CPU and the speed is growing at a faster rate than that of CPU. Furthermore, in modern GPUs, some stages of the traditional fixed-function pipeline have been replaced by fully programmable modules. With the increasing programmability, GPU is becoming a strong candidate for performing computationally intensive operations in many general-purpose applications. Some recent work has used GPU for non-graphics applications, such as linear algebra, physical simulation, etc.

This work has been supported in part by the Innovation and Technology Commission (project no. ITS/122/03 and project no. GHP/033/05) and Research Grants Council (DAG04/05.EG34) of the Hong Kong Special Administrative Region, China.

Some work has developed a GPU-based module for image and video decoding [2–4]. However, there is little research on encoder acceleration using GPU [5,6]. Kelly *et. al.* proposed to perform motion estimation (ME) on GPU [5,6]. They calculate the absolute difference at frame-level using GPU. In H.264/AVC, the motion vector (MV) predictor, which is the median MV of three neighboring coded blocks, is involved in the cost function for ME, it also affects the search center of ME. The existing implementations are not suitable for H.264/AVC because of the dependency introduced by the MV predictor. Furthermore, they suffer from unsatisfactory performance in integer-pixel ME because of their low arithmetic intensity which is defined as operation per word transferred [7]. Motivated by these, this paper presents a new implementation to perform ME on GPU for H.264/AVC on block-by-block basis. We first divide the macroblock (MB) into sixteen 4x4 blocks, then we calculate the cost of each 4x4 block and add them up by using multiple rendering passes. With the proposed method, the arithmetic intensity can be easily adjusted by changing the number of 4x4 blocks being processed per rendering pass. This feature is particularly useful when the proposed method is applied on different GPUs.

The rest of the paper is organized as follows. In Section 2, a brief overview of the programmable graphics pipeline and the dependency problem of rate-constrained ME in H.264/AVC is discussed. Then, the proposed method is presented in Section 3, followed by the implementation details in Section 4. The performance is evaluated by comparing with the Intel's SIMD optimized CPU version in Section 5. We conclude in Section 6.

2. PROGRAMMABLE GRAPHICS PIPELINE AND MOTION ESTIMATION IN H.264/AVC

2.1. Programmable Graphics Pipeline

Figure 1 shows a high-level diagram of modern graphics pipeline. The typical pipeline consists of several stages including transform and lighting (T&L) unit, primitive assembly unit, rasterizer, texture mapping unit and frame buffer. In a modern graphics architecture, programmable vertex processor and fragment processor are introduced. A set of customized operations can be applied on per-vertex and per-fragment basis by executing a program, called shader, on these programmable processors as an alternative for the T&L unit and texture mapping unit respectively. They are a SIMD machine which is capable of performing operations on a vector with 4 components. In graphics applications, sophisticated visual effects are generated with a series of shaders and multiple rendering passes.

For general purpose computation, the GPU is considered a stream

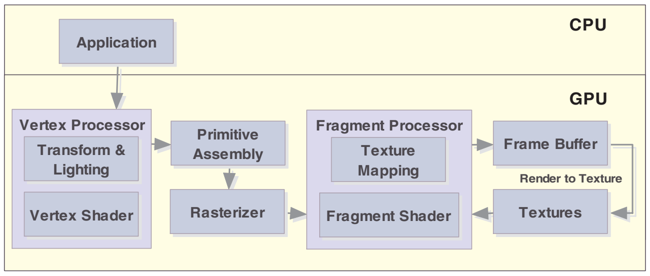


Fig. 1. The modern programmable graphics pipeline.

processor which executes a number of kernels on data streams. Application kernels are written as a series of vertex shaders or fragment shaders. Application-dependent data streams are stored as the geometries and textures. In the proposed ME module, kernels are used for cost calculation, merging and reduction. Textures are used to store the reference frames and the intermediate results. The details in GPU-based ME will be presented in Section 3.

2.2. Rate-constrained Motion Estimation in H.264/AVC

In H.264/AVC, it provides highly flexible MC and ME scheme which supports a combination of different block sizes ranging from 4x4 to 16x16 with separate MVs. Therefore, besides the sum of absolute difference (SAD), the cost function for ME also needs the MV predictor to estimate the bits required to code MVs, and hence introduces the dependency among adjacent blocks. The problem of choosing the best MV can formulate as a rate-constrained optimization problem and the best MV is the one which minimizes the following Lagrangian cost function,

$$J_{motion} = D_{DFD} + \lambda_{motion} R_{motion}. \quad (1)$$

Therefore, the ME in H.264/AVC can be written as

$$mv^* = \arg \min_{mv^i \in s} J_{motion}. \quad (2)$$

In the above equation, λ_{motion} is a Lagrangian multiplier imposing rate constraint of motion information which is QP dependent, R_{motion} is the bits required to code MVs. D_{DFD} can be either the SAD or the sum of absolute difference of Hadamard-transformed coefficients (SATD) in H.264/AVC. In this paper, SAD is used as distortion measure due to its simplicity. The candidate MV i, $mv^i \in S$, which minimizes the cost J_{motion} is the best MV, mv^*.

As shown in the Eq. (1), J_{motion} is a function of D_{DFD}, λ_{motion} and R_{motion}. R_{motion} captures the dependency among the adjacent blocks. It depends on the difference between the current candidate MV and the MV predictor. Since the implementations proposed in [5,6] are not designed for the cost function in Eq. (1), they are not applicable for H.264/AVC. In this paper, we propose a GPU-based implementation to perform ME efficiently in H.264/AVC using the above cost function. Furthermore, the proposed implementation allows us to adjust the arithmetic intensity to utilize the computing power of different graphics cards.

3. GPU-BASED MOTION ESTIMATION

In this section, we first present the data representation and the dataflow of GPU-based ME module, followed by a discussion on the techniques of implementation in Section 4.

3.1. Proposed GPU-based Motion Estimation

We have implemented the exhaustive motion search on GPU, instead of other fast ME algorithms, because of its regular memory access pattern. Although other fast ME algorithm can certainly be implemented, we need an additional layer of texture to specify the target searching position and this results in random memory access pattern and dependent texture read. The repercussion of these are quite significant in modern graphics architecture. In the following subsections, we present the representation of different elements of ME in graphics hardware and the high-level dataflow in the system.

3.2. Data Representation in Graphics Hardware

The current MB and reference frames can be represented as texture objects and stored in the texture memory. However, bandwidth is expensive in the graphics hardware, so too much texture access should be avoided. As the current MB participates in each cost calculation at different positions, it is beneficial to pass it as uniform parameter instead of store as texture object. This eliminates a large number of texture operations. We used sixteen 4x4 matrices (`float4x4`) to represent the values of current MB and pass them into the fragment program. This representation takes advantage of data-level parallelism and also matches the smallest partition size supported by H.264/AVC. For the reference frames, as their size is quite large and they will be repeatly accessed by the fragment program, it is desirable to load them once to the texture memory before the encoding of the current frame.

Motion search area specifies a set of possible candidate MVs. As the search area is clipped at the frame boundary, this is usually handle, in CPU implementation, by having boundary checking with an `if`-statement. However, limited branching support in the latest GPUs implies that it would introduce extra cost due to the execution of a branch in the fragment program. To avoid boundary checking, the search area is determined by the CPU. Then a quadrilateral is drawn to represent this in graphics hardware. Only the fragments inside this quadrilateral after projection is processed by the fragment program. The actual position in the reference frame is specified as texture coordinates. As a result, we have a branch-free fragment shader which is preferable in existing GPUs with limited branching support.

3.3. GPU Working Flow

We present in Figure 2 a high-level view of our shaders and the dataflow between them. The system is divided into three conceptual components. The first part is the cost calculation for 4x4 blocks. As shown in Eq. (1), the motion cost is calculated by CPU and intermediate results are loaded into the texture. The blending function adds the motion cost and result from the fragment shader. Then, the merging procedure which is necessary for our method to adjust the arithmetic intensity. This is used to combine the intermediate results in different rendering targets and determine the final cost. We present in Section 4.2 how to adjust arithmetic intensity. Finally, the cost for the search area goes through a labelling and reduction process and return the minimum cost and its corresponding position. Prior to the reduction, labelling process associates the cost with its corresponding position. Therefore, the result of reduction is a triple (SAD, ref_x, ref_y), where ref_x and ref_y are the coordinates of the reference position corresponding to the minimum cost. However, this is optional since, in some cases, extra rendering passes may be more expensive than data readback from GPU.

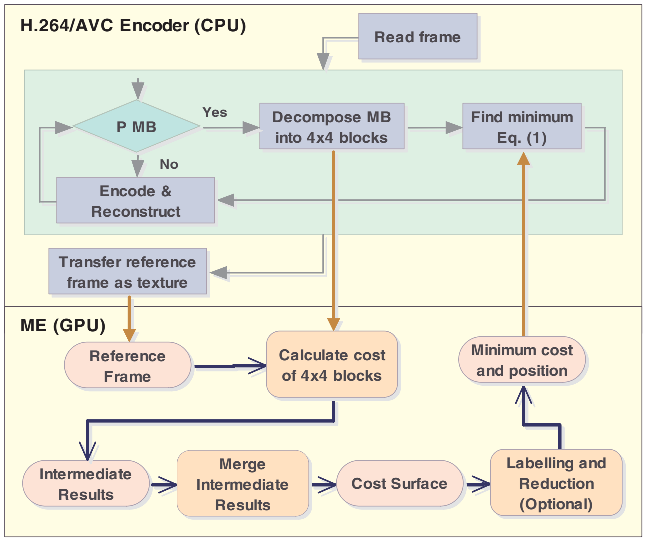
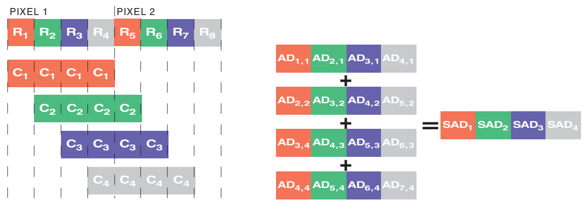

H.264/AVC Encoder (CPU)
Read frame
P MB
Yes
No
Decompose MB into 4x4 blocks
Find minimum Eq. (1)
Encode & Reconstruct
Transfer reference frame as texture
ME (GPU)
Reference Frame
Calculate cost of 4x4 blocks
Minimum cost and position
Intermediate Results
Merge Intermediate Results
Cost Surface
Labelling and Reduction (Optional)

Fig. 2. Block diagram of the high-level view of the GPU-based motion estimation module (Vertex shader is not shown in the figure).

The address calculation, such as the position of reference block and intermediate result, is performed by the vertex processor and the rasterization interpolates them for each fragment. The fragment processor is responsible for cost calculation. The workload is distributed over vertex processor and fragment processor.



(a) Absolute difference calculation of four reference positions.

(b) Summation of four vectors to get the SAD.

Fig. 3. Demonstration of SAD calculation and its data packing.

4. IMPLEMENTATION

In this section, we describe in details how our implementation leads to data-level parallelism and hence efficient SAD computation using GPU and how arithmetic intensity can be adjusted by our implementation. Arithmetic intensity is formally defined as operation per word transferred. With higher arithmetic intensity, more computation can be performed while a word is fetched from memory. Our implementation is based on OpenGL as this is a cross-platform API and all the shaders are written in Cg language.

4.1. Data-level Parallelism

In section 3.2, we presented how different parts of the ME are represented in graphics sense. The current MB is decomposed into sixteen 4x4 matrices as `float4x4` so that we can use the matrix manipulation routines in GPU. On the other hand, the reference frames are loaded into the texture memory once so that the fragment program fetches the pixel values for calculation. GL_RGBA8 is chosen as the internal data type for storing the reference frames. Four luminance values are packed in a pixel with 8-bit per channel. This provides the minimum sufficient precision needed. Also, data packing is preferred to take advantage of data-level parallelism and better bandwidth utilization, because we can process and fetch more data at a time. However, ME need to access a chunk of values at any position. With this packing strategy, it restricts the access to the multiple of four position only.

To solve this problem, the fragment program fetches two adjacent pixels (eight luminance values) from texture and performs calculation of four reference positions in a single pass. A 1D demonstration is illustrated in Figure 3(a). Pixel 1 and 2 contain eight values in reference frame, R_i, where $i = 1, 2, ..., 8$ and given the current block as C_j where $j = 1, 2, 3, 4$. By making use of swizzle operator, we compute absolute difference on each component of current block against the reference frame as shown in the figure and obtain four vectors containing the absolute difference, ADi, j where i and j indicate the position corresponding to reference frame and current block respectively, $i = 1, 2, ..., 8$ and $j = 1, 2, 3, 4$ in Figure 3(b). Then, we can compute the four SADs corresponding to reference position R_1, R_2, R_3 and R_4 by summing these four vectors as shown in Figure 3(b). The result will render to frame buffer. This method solves the problem of restricted access and provides a certain degree of parallelism.

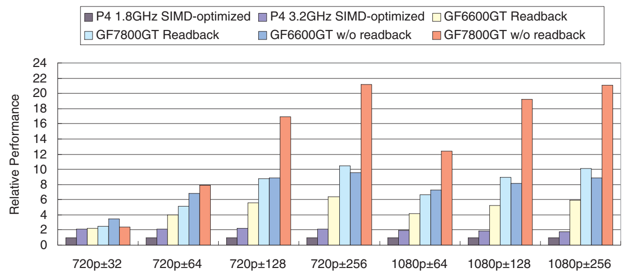
4.2. Adjustable Arithmetic Intensity

The performance of the GPU-based method is usually measured by the arithmetic intensity. It is good to provide a mechanism to adjust the arithmetic intensity depending on the processing ability of hardware. In the proposed method, the arithmetic intensity can be adjusted by processing more 4x4 blocks per rendering pass. Referring to Figure 3(a), besides C_1, C_2, C_3 and C_4, we can also compute the absolute difference for C_5, C_6, C_7 and C_8 with R_i and render the results to another rendering target. This approach results in more output values so it needs to use with multiple rendering target (MRT), which allows more than one RGBA pixels output in each pass. Then, in the merging step, the results are added together by offsetting the textures. We perform more more arithmetic operations with the same number of texture operations. However, the increase in arithmetic intensity is bounded by the limited number of MRTs available in current graphics hardware, which is up to four.

5. PERFORMANCE EVALUATION

We now examine the performance of the proposed GPU-based implementation on both consumer-level (Nvidia GeForce 6600GT AGP) and high-end (GeForce 7800GT PCIe) graphics cards running version 81.98 driver on Windows XP. Extensive tests have been performed on a PC with Intel Pentium 4 3.2GHz processor and 1 GB DDR2 memory. Another PC with Intel Pentium 4 1.8GHz processor and 1GB RDRAM memory is used as reference, and all results will be normalized by the results of this PC. Since the existing GPU-based implementations are not applicable for H.264/AVC, the comparison will not be given. Experiments are conducted to evaluate the performance of the proposed GPU implementation, the impact on limited download bandwidth and the performance change by adjusting the arithmetic intensity.

Figure 4 provides a breakdown of the performance of proposed GPU implementation. The execution time is the user time measured



	P4-1.8GHz	P4-3.2GHz	GF6600GT Readback	GF7800GT Readback	GF6600GT w/o readback	GF7800GT w/o readback
1080p ±256	256197	142196	42962	25428	29050	12156
1080p ±128	71331	39146	13706	7975	8768	3706
1080p ±64	19909	10359	4800	2996	2756	1603
720p ±256	33684	15787	5271	3215	3509	1593
720p ±128	11268	5184	2025	1290	1275	665
720p ±64	3134	1481	778	615	459	396
720p ±32	847	400	378	343	247	359

Fig. 4. Comparison between the relative performance of CPU and GPU implementation which is normalized by the Intel Pentium4 1.8GHz performance. The table gives the average execution time for processing one frame in millisecond (ms).

from the beginning to the end of ME. We can see the speed of P4-3.2GHz shows constantly about two times faster than P4-1.8GHz in different resolutions and search ranges. However, the speed of the proposed GPU implementation with data readback shows ten times and five times faster than the CPU implementation on P4-1.8GHz and P4-3.2GHz respectively. We observed that the speed increases with the search range because larger search range diminishes the effect of setup overhead of graphics API.

To study how the download bandwidth (GPU to CPU) may affect the overall performance, we simply perform same set of experiments without readback the data from GPU to CPU. The speed is doubled comparing with the previous experiments with the data readback in both graphics cards tested. Although the newest PCIe is expected to provide higher readback bandwidth than AGP, however, they show similar readback bandwidth of float texture in RGBA format (around 500 MB/sec) in GPUbench [8], which is a benchmark tool for GPU. This is the main bottleneck in most general purpose computation on GPU which needs to read the data for further processing.

Finally, we adjust the arithmetic intensity by using our method and study the performance impacts. By increasing the number of 4x4 blocks processed in a single pass, the average execution time needed on GeForce 7800GT PCIe is shown in Figure 5. We found that the best performance of GeForce 7800GT PCIe can be achieved when two 4x4 blocks are processed in a single pass with two rendering targets. We have the arithmetic intensity about 10:1. To further increase the number of 4x4 blocks processed in a single pass, the speed decreases since the GPU is overloaded. The proposed GPU implementation allows the performance optimization by adjusting the arithmetic intensity.

6. CONCLUSIONS

We have presented a GPU-based ME implementation to offload the computation burden from CPU to GPU which is applicable in the latest H.264/AVC video coding standard. It provides a mechanism to adjust the arithmetic intensity to maximize the performance on different GPUs. We have implemented the proposed ME module and demonstrated its effectiveness with both the consumer-level and

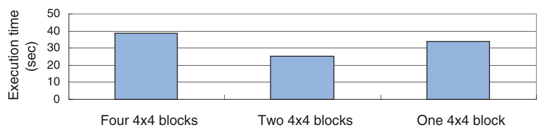


Fig. 5. Comparison between the execution time of performing motion estimation on 1080p format and ±256 search range with different number of 4x4 blocks processed in a single pass.

high-end graphics cards. Both of them outperform our optimized CPU implementation, and achieve about ten times speed-up. The performance boost is currently limited by the download bandwidth as shown in our experiments. This bottleneck is expected to be alleviated with the increasing bandwidth of PCIe bus in the future. With our GPU-based ME module, an alternative way to speed-up the computationally intensive encoding processing by utilizing the processing power of GPU is provided.

Acknowledgment

This work has been supported in part by the Innovation and Technology Commission (project no. ITS/122/03 and project no. GHP/033/05) and Research Grants Council (DAG04/05.EG34) of the Hong Kong Special Administrative Region, China.

7. REFERENCES

[1] Joint Video Team (JVT) of ISO/IEC MPEG and ITU-T VCEG, *Draft ITU-T recommendation and final draft international standard of joint video specifiction (ITU-T Rec. H.264/ISO/IEC 14 496-10 AVC)*, May 2003, JVT-G050.

[2] Guobin Shen, Guang-Ping Gao, Shipeng Li, Heung-Yeung Shum, and Ya-Qin Zhang, "Accelerate video decoding with generic GPU," *IEEE Transactions on Circuits and Systems for Video Technology*, vol. 15, no. 5, pp. 685–693, May 2005.

[3] Jianqing Wang and Tien-Tsin Wong and Pheng-Rnn Heng and Chi-Sing Leung, "Discrete Wavelet Transform on GPU," May 2004, http://www.cse.cuhk.edu.hk/ ttwong/demo/dwtgpu/dwtgpu.html.

[4] Bo Fang, Guobin Shen, Shipeng Li, and Huifang Chen, "Techniques for efficient DCT/IDCT implementation on generic GPU," in *Proceedings of the IEEE International Symposium on Circuits and Systems*, May 2005, vol. 2, pp. 1126–1129.

[5] Francis Kelly and Anil Kokaram, "General purpose graphics hardware for accelerating motion estimation," in *Irish Machine Vision and Image Processing Conference (IMVIP)*, Sept. 2003.

[6] Francis Kelly and Anil Kokaram, "Fast image interpolation for motion estimation using graphics hardware," in *IS&T/SPIE Electronic Imaging - Real-Time Imaging VIII*, May 2004, vol. 5297, pp. 184–194.

[7] Matt Pharr and Randima Fernando, *GPU Gems 2: Programming Techniques for High-Performance Graphics and General-Purpose Computation*, Addison Wesley Professional, 2005.

[8] Ian Buck, Kayvon Fatahalian, and Pat Hanrahan, "GPUBench: Evaluating gpu performance for numerical and scientific applications," in *ACM Workshop on General Purpose Computing on Graphics Processors*, Aug. 2004.