In this paper, multi-pass and frame parallel algorithms are proposed to accelerate various motion estimation (ME) tools in H.264 with the graphics processing unit (GPU). By the multi-pass method to unroll and rearrange the multiple nested loops, the integer-pel ME can be implemented with two-pass process on GPU. Moreover, fractional ME needs six passes for frame interpolation with six-tap filter and motion vector refinement. Motion estimation with multiple reference frames can be implemented with two-pass process with frame-level parallel scheme by use of SIMD vector operations of GPU. Experimental results show that, compared to implementations with only CPU, about 6 times to 56 times speed-up can be achieved for different ME algorithms.

1. INTRODUCTION

Video compression plays a key role in many multimedia consumer products, such as digital still cameras (DSCs) and digital video recorders (DVs). Among various video coding standards, H.264/AVC, the newly established video coding standard, can achieve much better coding performance than others with the cost of high computational complexity. The instruction profiling results of H.264 is shown in Table 1 [1]. It is evident that motion estimation is the most significant part. That is, accelerating motion estimation would actually speed up the whole video encoding. Thus the motion estimation is chosen to be our main concern.

On the other hand, more and more commodity PC and game consoles are commonly equipped with graphics processing units (GPUs), and in the recent years, graphics processors have changed from fixed pipelines to programmable pipelines. In the programmable graphics pipeline, the vertex and fragment shader could be programmed to achieve different special graphics effects [2]. Because of the programmability, some nongraphics applications are also considered to be executed on GPU instead of CPU [3], such as FFT [4] and motion compensation [5].

In fact, there are many advantages to use GPU instead of CPU for DSP related applications, including video coding. First, the architecture of GPU is highly parallel and supports SIMD operations, which can be more appropriately described as a streaming processor. Second, the memory bandwidth between the GPU and the video memory is about five times as fast as that between the CPU and the system memory. Both these two features are beneficial for DSP algorithms. In the past, motion estimation has been implemented by use of dedicated hardware of multiple parallel processing elements (PEs) [6] [7]. Since GPU is parallel architecture, it is able to efficiently process motion estimation as well.

In this paper, we first propose an algorithm to map motion estimation (ME) on generic GPU to accelerate video encoding. Second, advanced motion estimation algorithms in H.264 [8], such as quarter-pel ME and multiple reference frame ME, are implemented as well as showing the pros and cons caused by GPU’s characteristics.

2. MOTION ESTIMATION WITH GPU

A typical programmable GPU could be viewed as two programmable blocks that run serially: the vertex shader for MIMD processing and the fragment shader for SIMD processing [2]. Motion estimation is inherently an image-based algorithm and is suitable to be executed successively by several fragment programs in a SIMD-like fashion.

2.1. A Novel Full Search Integer-Pel Motion Estimation Algorithm

The full search motion estimation algorithm is complex and time-consuming because multiple nested loops are required. Beside the sum of absolute difference (SAD) computation and SAD comparison, there are four loops in the classic ME algorithm, as shown in Fig. 1 [7]. In our proposed ME algorithm shown in Fig. 2, the four classic nested loops are unrolled and rearranged to a one-tiered loop and a two-tiered loop. When fragment shader draws pixels, it would perform the program concurrently for many pixels. Thus we can think of each pixel as one PE and the texture as one video frame here.

It is highly recommended to execute rendering at only one pass in [3]. However, the total instruction amount of full-search block matching motion estimation is much more than the instruction limit.

<table>
<thead>
<tr>
<th>Tools</th>
<th>MIPS</th>
<th>Percentage(%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Integer-pel motion estimation</td>
<td>234,238</td>
<td>74.36</td>
</tr>
<tr>
<td>Fractional-pel motion estimation</td>
<td>65,575</td>
<td>21.40</td>
</tr>
<tr>
<td>Transform &amp; Quant</td>
<td>5,403</td>
<td>1.70</td>
</tr>
<tr>
<td>Others</td>
<td>3,685</td>
<td>1.17</td>
</tr>
</tbody>
</table>
of the target GPU, GeForce 7800 GT. Therefore, a multi-pass ME algorithm on GPU is needed. We perform the proposed ME algorithm by executing two fragment programs in two passes. Figure 3 shows the block diagram of the two passes we proposed to implement the ME on GPU. Figure 4 is the illustration of the proposed algorithm. The hollow circles represent candidates in the search range, while the other opaque circles represent PEs.

2.1.1. The First Pass: Generating Local Minimum SADs

The whole video frames are first transferred from system memory to video memory as texture data to make use of the large memory bandwidth. The size of the texture must be power of two. Thus we need to pad the frame, and the padded frame size is denoted as \( W_T \times H_T \). The macroblock (MB) size is \( MB_{row} \times MB_{col} \), and the search range is \( SR_{row} \times SR_{col} \). Finally, a quadrilateral whose size equals to \( W_{p1} \times H_{p1} \) is drawn. Since each pixel can be viewed as one PE, there are \( W_{p1} \times H_{p1} \) PEs. In order to perform the full search ME, the number of candidates in charge per PE is

\[
n = \frac{W_T}{MB_{col}} \times \frac{H_T}{MB_{row}} \times SR_{row} \times SR_{col} \times \frac{1}{W_{p1}} \times \frac{1}{H_{p1}}. \tag{1}
\]

If we take \( W_T = H_T = 512 \), \( MB_{row} = MB_{col} = 16 \), \( SR_{row} = SR_{col} = 32 \) ([−16, +15]), \( W_{p1} = H_{p1} = 512 \) as illustrated in Fig. 4, the result is \( n = 4 \), which means that every four candidates are grouped and mapped to one PE in the corresponding MB. This is the only loop in the first pass. Each PE compares the sum of absolute differences (SADs) of the four candidates and writes the motion vector with the smallest SAD value to texture with the render-to-texture extensions. Note that, in Fig. 4, only four PEs are shown for a MB in the first pass, instead of 256 PEs in this example case, for simplification.

2.1.2. The Second Pass: Generating Global Minimum SADs

The remained computation for the integer-pel ME (IME) is much smaller in pass 2. Thus each MB only needs to be processed by a PE. The number of PEs is

\[
Quad\ size\ of\ pass\ 2 = \frac{W_T}{MB_{col}} \times \frac{H_T}{MB_{row}}. \tag{2}
\]

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

\[
W_{p1} \times H_{p1} \times \left( \frac{W_T}{MB_{col}} \times \frac{H_T}{MB_{row}} \right)^{-1}. \tag{3}
\]

In the above example shown in Fig. 4, the quadrilateral size of pass 2 is 32×32, and 256 local minimum SAD values would be compared to find an integer-pel motion vector.

2.2. Implement Fractional Motion Estimation on GPU

Fractional motion estimation (FME) can be used to improve the coding performance by offering better temporal prediction, and in H.264, quarter-pel precision is supported. As shown in Table 1, since it also occupies a large portion of computation, the acceleration of FME is also required. There are usually two steps to perform FME. The first step is to interpolate the reference frame, and the second step is to refine the motion vectors. The details will be explained in the next two subsections.

---

**Fig. 1.** The pseudo code of the classic integer-pel ME algorithm.

**Fig. 2.** The pseudo code of the proposed integer-pel ME algorithm.

**Fig. 3.** The block diagram of the proposed ME algorithm.

**Fig. 4.** The illustration of the proposed ME algorithm.
2.2.1. Frame Interpolation with Three-Pass Algorithm

In H.264 coding standard, it defines a six-tap filters for half-pel interpolation [8]. If the build-in bilinear texture filter in GPU is applied instead, it will lead to video quality degradation. Therefore, the frame interpolation should be done with shader programs developed by ourself.

Unfortunately, the interpolation operations also need to be done with many passes. The reason is not limited in instruction amounts but the nature of the GPU. That is, the results of one PE cannot be the input of another PE in the same pass. As illustrated in Fig. 5, where A circles stand for integer pixels, to compute the value of C, the values of Bv and Bh are needed. Consequently, C and Bv cannot be calculated in the same pass because of the data dependency. Therefore, at the first pass, Bv and Bh are computed and stored in the texture memory by render-to-texture extensions. Then compute C at the second pass. Finally, the quarter pixels represented as opaque circles are computed at the third pass. Note that, during these three passes, especially the second pass, many pixels just pass through, such as Bv and Bh, at the second pass. It means the GPU’s computing power is not fully utilized, which would decline the performance of speed-up. The detailed comparison will be shown in Section 3.

2.2.2. Motion Vector Refinement

After the reference frame is interpolated, we can perform the motion vector refinement as shown in Fig. 6. A typical refinement has two steps. Find the best half-pel motion vector first and then the best quarter-pel motion vector. The first step can be integrated to the second pass of IME, where one more pass is required for the second step. Note that, before refinement, the motion vectors generated by IME should be multiplied by four to index the interpolated frame, and this refinement operation is blockwise inherently, so the required PEs are equal to MB number. It is 32x32 for the example described in Section 2.1. In summary, for IME and FME, six passes are required, including three passes for interpolation and three passes for candidate searching of IME and FME.

2.3. Multiple Reference Frame Motion Estimation with Frame Parallel Technique

Unlike FME, motion estimation with multiple reference frames (MRF) can be implemented very efficiently. It is because that GPU has SIMD vector operations and can do motion estimation in parallel among different reference frames, which is called as frame parallel technique here. To support MRF, we can just slightly modify the algorithm in Section 2.1, as shown in Fig. 7, where C denotes the pixel value of the current frame, R1, R2, and R3 denoted the pixel values of three different reference frames, respectively. And S1, S2, and S3 are the values of SAD for three candidates located in three different frames. \( S_{\text{min}} \) is the currently minimum SAD value, \((Mv_x, Mv_y)\) is the corresponding motion vector, and \(i\) is the corresponding frame index. Therefore, The data flow of this algorithm could be described as following. At the beginning, \(R1, R2, R3, C\) would be fetched from texture memory and stored in two Registers(vector with 4 values). \(C\) would be duplicated 3 times to the register for next vector operation. Then the SAD computation of 3 reference frames would be processed parallelly. It is because the + and − operations in the figure are vector operations, not just scalar operations. With 256 times iterations (macroblock size), three SAD values could be obtained and by comparison, the lower SAD could be found and stored. Finally, as the example in IME, 4 candidates would be processing and the local minimum motion vector would be got at the first pass.

From above, the main difference to one reference frame is the vector operation. In GPU, it has build-in hardware resources for vector computation, and the required cycles are equal to scalar computation. Therefore, ME with MRF could be executed with slightly lower speed to ME with one reference frame. The reason is the extra comparison and the additional bandwidth of reference frames. The bandwidth overhead could be minimized by packing the pixels to one texture \((R1, R2, R3, C\) to RGBA). It is also the reason we use only three reference frames. In spite that maximum four reference frames can be supported in this scheme.

3. EXPERIMENTAL ENVIRONMENTS AND RESULTS

We have performed extensive tests on a PC with an Intel Pentium IV 3.00 GHz CPU, 1 GB memory, and a NVIDIA GeForce 7800 GT GPU with 256 MB video memory. The fragment shader program to perform motion estimation is written with OpenGL and the Cg programming language and runtime libraries of nVidia Corporation [9] [10]. On the CPU side, we use Microsoft Visual C++ .net 2003 and open the SSE2 option to maximum the performance of CPU. One video sequence of MPEG, Stefan in CIF (352x288) format, is chosen as an example. And the parameters of ME (PE number etc.) are the same as those shown in Fig. 4. There are three modes in our ex-
4. CONCLUSION AND FUTURE WORKS

In this paper, we have demonstrated that GPU can accelerate motion estimation in H.264/AVC together with CPU. We implement the integer-pel ME, the fractional ME, and integer-pel ME with multiple reference frames with multi-pass and frame parallel algorithms.

Experimental results show that significant speed-up can be achieved. Currently we only focus on motion estimation and the other parts still are computed on CPU. It results in the communication between GPU and CPU and the performance are degraded. In the future, we would like to implement a complete video compression algorithms on GPU. Besides, we are also interested in more coding tools in H.264 like variable block size motion estimation and intra prediction. And finally, a complete framework of H.264 encoder with acceleration of GPU could be presented. We expect it would get higher performance and approach the realtime processing speeds.

5. REFERENCES