Graphics Hardware (2006)
M. Olano, P. Slusallek (Editors)

Efficient Video Decoding on GPUs by Point Based Rendering

Bo Han and Bingfeng Zhou

National Engineering Research Center for New Technologies in Electronic Publishing
Institute of Computer Science and Technology
Peking University, Beijing, P.R.China

Abstract

To accelerate computation intensive video decoding tasks, we present a novel framework to offload most decoding operations to current GPUs. Our method is based on rendering graphics points and suitable for block-based video standards. By representing video blocks as graphics points, we achieve great flexibility and high parallelism to utilize the GPU's pipelined stream processing architecture. The computational resources within texture units and blending units are also exploited to facilitate computations. We propose a high performance implementation of IDCT on GPUs, which efficiently excludes most zero-value coefficients to save the bandwidth and the computations. Compared with the existing quad-based representation, our point based implementation of MC greatly reduces data transfer and redundancy. We have demonstrated the efficiency of our proposed framework by a MPEG-2 decoder. Our results indicate a significant improvement over prior CPU and GPU solutions.

Categories and Subject Descriptors (according to ACM CCS): I.3.3 [Computer Graphics]: Graphics hardware

1. Introduction

Digital video applications have become an essential component of our daily lives, ranging form high-definition televisions to multimedia mobile devices. For most users, efficient video decoding is their most concern. But advanced video compression techniques adopted in current video standards make the decoding process one of the most computational demanding tasks. The popular high definition (HD) videos and the new video standards propose challenges to current CPUs for both intensive computations and huge volumes of data. Therefore, a means should be found to offload some decoding tasks from CPUs to other sub-systems.

Graphics cards have been used to assist video decoding over the last decade. First, video overlay was introduced to handle expensive color space conversion. Then, dedicated decoding hardware emerged on PCs and has been widely adopted on today's commodity graphics chips due to the well defined DirectX Video Acceleration(DXVA) specification. Unfortunately, most of them are built on hard-wired circuits only for certain specific video standards (most for MPEG-2 playback). Recently, graphics vendors have began to integrate programmable video-processing engines in their products, such as nVidia's PureVideo and ATI's Avivo. The engines are built on underlying SIMD vector processors(VPs) designed specifically for video algorithms. However, they cost additional transistors and are independent from the traditional 3D graphics hardware. Until now these VPs still lack a high level programming interface to release the computational power.

Driven by interactive 3D graphics applications, commodity graphics hardware has evolved into a powerful and flexible graphics processor , known as GPUs. Their performance and functionalities have made GPUs attractive as coprocessors for various general-purpose computation problems [OLG*05]. Architecturally, GPUs are highly parallel streaming processors optimized for vector operations, which is similar to some media processors [ORK*02] [JL05]. Due to this facts, it should be possible to map the video coding algorithms on GPUs, and we expect such a solution can have several advantages. First, the well established graphics APIs and high level shader languages can provide a convenient programming interface. Therefore, the implementation can be independent of underlying hardware and platforms. Secondly, the higher performance growth rate over Moore's law and increasingly enhanced functionalities make GPUs more promising. Thirdly, a unified multimedia processing subsystem, capable of handling different types multimedia contents, could be implemented on GPUs, which can achieve

higher utilization of hardware resources and is specially attractive for mobile devices.

Since GPUs are specially designed for graphics operations, it is hard to directly map the complex and branchy video decoding algorithms onto the graphics pipeline. Today's video standards share a similar block/macroblock structure. Each block has its own parameters and characteristics, which makes the computation not efficient in a single quad-texture used by traditional GPGPU applications. Recently some researchers have began to exploit GPUs to implement partial video/image decoding process [SGL*05] [FSLC05] [HL05]. They adopt the texture-based GPGPU model and demonstrate the feasibilities, but the insignificant performance speedup makes them not attractive enough in a practical real-time decoder.

The main contributions of this paper are concluded as follows. First, we propose a novel GPU-accelerated video decoding framework based on rendering graphics points. By mapping video blocks to points, we transform decoding operations into highly parallel graphics tasks. Our point-based block representation greatly reduces the data transfer and redundancy. Secondly, we present a high performance IDCT implementation of the straightforward basis-image combination approach on GPUs, which excludes most zero coefficients and fully utilizes the graphics pipeline for higher efficiency. It is specially suitable for decoding highly compressed videos and images. Thirdly, our study demonstrate the GPU's great potential capabilities to perform video coding algorithms. Some discussions about modifications on GPUs are given and expected to improve the performance for generic media processing.

The rest of the paper is organized as follows. Section 2 gives a brief survey on related works. Section 3 highlights some characteristics of a typical block-based video decoder to facilitate understanding of our implementation. We present details of our point-based decoding methods in section 4 and evaluate our methods on aspects of performance and visual quality in section 5. Section 6 and 7 gives the discussion and concludes the paper.

2. Related Works

In this section we give a brief overview of previous works on using GPUs for video coding/processing and highlight several related point based applications.

In the area of video processing, GPUs have been widely used to render or post process video pictures such as filtering, composition, visual effects, and color space conversion(CSC) [App05] [DNVRH*05] . Moreover, Ben et al. [CCLW05] gave the comparison and analysis of FPGAs and GPUs in their use for video processing, which indicates GPUs can offer a viable solution but fall down on applications with high memory accesses, such as 2D convolution in a big size. For the low level video coding algorithms, Shen et al. [SGL*05] first put forward to use generic GPUs to accelerate video decoding. By using small quads to represent video blocks, they move the whole motion compensation feedback loop and the CSC to the GPU, thus leading to a considerable performance speedup. Kelly and Kokaram [KK04] proposed to use texture bilinear filtering units on the GPU for fast image interpolation and combined it with motion estimation. Then, two DCT/IDCT methods on GPUs [FSLC05] [Nvi05] were proposed; they are based on direct matrix multiplication and the JPEG ANN fast algorithms respectively. The performance of the former one is higher due to its regular memory accesses. Both two methods can achieve comparable performance to the optimized CPU implementation. However, in practice the expensive floating point texture updating and unpacking can greatly hamper the overall performance, which has been addressed in the recent study of [HL05]. In that paper, Hirvonen and Leppänen implemented a GPU-based H.263 decoder based on similar techniques with the previous works. They demonstrated that current GPUs are capable of handling all the processing stages of H.263 video decoder except serial variable length decoding. However, their work mainly focuses on the feasibility and concerns a little about the performance.

In terms of graphics points, due to the simplicity and superior flexibility, point sets have became increasingly attractive as an alternative surface representation as well as for processing of complex 3D models [KB04]. Our work in this paper is mainly inspired by its advantages shown in large particle systems [KKKW05] [KSW04]. In addition, Krüger et al. [KW03] also introduce a special case of using points to perform linear algebra operations on sparse random matrices. They use sets of vertices to render the matrix values at the correct position, which exploits the sparseness and saves a significant amount of GPU memory and operations.

3. Characteristics of Decoding Modules

In this section we give an overview of a typical block-based video decoder and analyze its characteristics to facilitate understanding of our implementation on the GPU.

Today's major video standards incorporate block-based transform coding and motion compensation techniques to exploit both the spatial and the temporal redundancy of a video sequence. The macroblock, corresponding to a 16×16-pixel region of a frame, is typically the basic unit for motion compensation. It is usually organized with four 8×8 luminance sample blocks and two chrominance blocks used for transform coding which is typically the discrete cosine transform (DCT). A block diagram of a standard video decoder is shown in Figure 1.

Each decoding module has its own characteristics. In general, variable length decoding (VLD) is a sequential bit-wise process and is the only functional module where GPUs can not offer any advantage. The following inverse zigzag scan

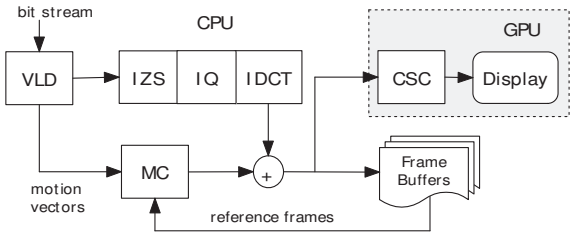

Figure 1: *The architecture of a typical decoder for block based video standards. The CSC module is moved to GPUs because nearly all graphics hardware have built-in support for YCrCb-RGB conversion.*

(IZS), inverse DCT (IDCT) and inverse quantization (IQ) target on each transformed block; the parallelism among each block and its elements makes these operations feasible to implement on GPUs. Moreover, due to the excellent decorrelation and energy compaction properties of DCT and the lossy quantization, the block often has only a few non-zero DCT coefficients in the low frequency zone. Motion compensation (MC) is a macroblock level operation that retrieves the prediction blocks from the reference pictures according to motion vectors and adds the residual signal on the prediction to form the final result. It is essentially series of memory read-write operations along with some arithmetic computations for interpolation, combination and saturation, which fits well with the GPU's texture fetch scheme and the fragment processing model. Finally, color space conversion (CSC) is just a per-pixel vector-matrix multiplication. In the following, we will give more details on IDCT and MC.

IDCT is typically computation intensive. The action of the 2D IDCT can be described in terms of transform matrices: $x = T^T XT$. where X is a matrix of DCT coefficients, T is the DCT transform matrix and T^T is its transpose. Many fast algorithms have been extensively studied. Most of them adopt the row-column decomposition and utilize the symmetry property of transform matrix T. In this way 2D IDCT shares the same architecture with DCT and is performed in two 1D IDCT passes, which is also adopted by two prior GPU implementations [FSLC05] [Nvi05]. In this study we highlight the basis image linear combination method. Due to the fact that the DCT coefficient specifies the contribution of the particular basis image, the 2D image block can be reconstructed by combining all basis images with their corresponding DCT coefficients. The process is described as:

$$x = T^T XT = \sum_{u=0}^{N} \sum_{v=0}^{N} X(u,v)[T(u)^T T(v)] \qquad (1)$$

where $X(u,v)$ is the coefficient matrix entry in u_{th} row and the v_{th} column. Its corresponding basis image is the outer product of the column vector $T(u)^T$ and the row vector $T(v)$. The process is inherently parallel on each DCT coefficient and can directly omit zero values to utilize the sparseness property of DCT matrix X, which is quite suitable for the GPU's stream processing architecture. In addition, all the basis images can be pre-calculated and easily be represented as a combined texture, as shown in Figure 3.

Motion compensation aims to exploit the temporal correlation between neighboring frames. The best match blocks in the reference frames is simply indicated by a motion vector. For higher coding efficiency, some techniques have been developed for better prediction at the cost of higher computational complexity, such as bidirectional prediction and sub-pixel precision prediction. The sub-pixel precision is achieved by interpolation, thus the filtering hardware of texture units on GPUs can be utilized for acceleration.

4. Point Based Video Decoding on GPU

In this section we first analyze the efficiency of our point based representation for video blocks. Then, we present our decoding framework and give the implementation of each decoding module. For the sake of simplicity the implementations are illustrated by a MPEG-2 decoder with OpenGL, but the general ideas also apply to other video standards.

4.1. Mapping Video Blocks to Graphics Points

Graphics points offer an efficient representation for a video block. A point primitive is only a single vertex along with associated attributes such as position, size and texture coordinates. On the other hand, each video macroblock possesses its own position, motion vectors, prediction types, etc; each of its transformed blocks has only a few nonzero values due to the sparseness property of quantized DCT coefficients. Since the point primitive can have more than ten 4D-vector attributes on current GPUs, it is practical to store the block-wise parameters and DCT coefficients into the attributes of a point. In addition, a point primitive with size n is rasterized to a $n \times n$ fragment block. The size can be globally specified by graphics APIs or dynamically manipulated within vertex programs, which fits well with various video block size. Furthermore, as points are naturally a set of vertices, the vertex processors can be easily utilized to preprocess the block information within the attributes of points. Moreover, after the rasterization we can exactly access each fragment covered by a point primitive with a fragment program by means of the texture coordinates generated by the point sprite extension and the window position semantics(*WPOS*). In this way, the powerful fragment processors can be utilized for per-pixel texture fetching and arithmetic computations. Therefore, we can fully utilize the graphics pipeline to achieve the block-level parallel decoding.

The GPU's nature of high parallelism requires that it's feeding elements are highly regular and well batched. In contrast, various video blocks with different types, parameters and coding modes always lead to quite different operations.

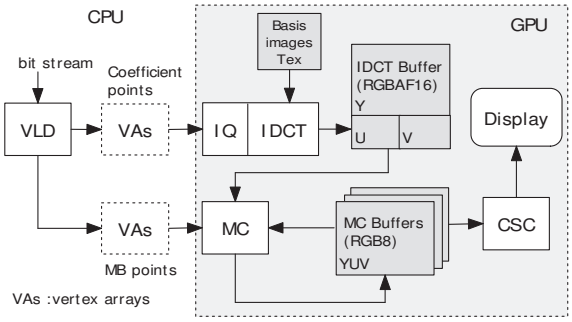

Figure 2: *Our point based decoding architecture. The graph also illustrates different texture layouts and formats for IDCT and MC buffers.*

Considering relatively expensive branch penalty on current GPUs, we move the flow-control decision up to the CPU and classify points into different vertex arrays in advance.

4.2. Overview of Proposed Architecture

In our implementations, there are two general categories of points: coefficient points for IDCT and macroblock(MB) points for MC. For MB points, motion vectors are the main attributes. This point set can be further subdivided according to frame/field prediction and intra/inter mode for better performance. For coefficient points, the main challenge comes from the irregular distribution of DCT coefficients within a video block. To create vertex arrays, we need apply a regular pattern to generate points. Notice that, after VLD, the DCT coefficients of a block are arranged into a 1D array following the zigzag scan order that provides a more compact representation. We simply group every four coefficients into a 4D vector through the array and assign each vector to a point. For the sparseness property and the compact representation, in general only 1–3 points are generated for each block. The solution is not optimal but practical, later in IDCT we further benefit from the SIMD vector operations.

Our point-based method has several advantages over the existing quad-based and the texture-based representation.

- Transfer cost for coefficient uploading is greatly reduced by culling most zero values.
- The data redundancy in four vertices of the quad-based representation is avoided and the cost is reduced to 1/4.
- The block-level representation is much more flexible. Non-coded blocks can be directly excluded.
- The vertex array can directly support integer data types without clipping, such as *short* used in video codecs. No needs for explicit data type or data range conversion on the CPU or GPU compared with the texture.
- Both the vertex and fragment processors can be utilized to balance the graphics pipeline for better performance.

Our proposed decoding architecture is depicted as Figure 2. We move most workloads to the GPU and left only the variable length decoding and the point arranging to the CPU. The decoding procedure has four steps. First, we generate points for video blocks and classified them into the corresponding vertex arrays. Then, we draw the coefficient points to complete the inverse quantization and inverse DCT. After that, the marcorblock points are rendered to finish motion compensation. Finally, an additional pass is used to display the MC off-screen buffer with color space conversion.

We use short integer data type for vertex arrays and 16-bit floating-point (*half*) for IDCT computations. The half representation has the range of contiguous counting numbers [-2048,2048], which fits well with the precision of the DCT coefficient. However, because the decimal parts are introduced for sub-pixel prediction, the precision of *half* becomes insufficient for sampling high definition video frames. So we use 32-bit floating-point for texture coordinates.

We use frame buffer objects (FBOs) and rectangle textures to implement our off-screen buffers. Although single channel fp16 texture is enough for our IDCT buffer, in practice we choose RGBA16F format to enable 16-bit blending of our GPU. To facilitate motion compensation and reduce the internal bandwidth requirement, the reference frames are stored in 8-bit RGB texture format, as shown in Figure 2.

4.3. Inverse Quantization

The inverse quantization (IQ) is essentially a multiplication of the quantizer step size. The basic IQ operation is:

$$X_{iq}(u,v) = qp \times QM(u,v) \times X_q(u,v) \qquad (2)$$

where $QM(u,v)$ is the quantization matrix entry. The quantization parameter qp is a block-wise variable scale factor.

In our framework the IQ is actually the vertex processing stage of the IDCT implementation. The related parameters and information are prepared by the CPU and stored in the attributes of points; the quantization matrix is loaded into the const memory by uniform parameters. The Equation 2 is efficiently performed by vector multiplications. To reduce transfer costs, we can pack the associated parameters into compact bits and later use a vertex program to decompress. All these additional operations on the vertex processors will not cause any performance penalty because of the graphics pipeline. However, the mismatch control in MPEG2/4, which is aimed to minimize the error accumulation due to IDCT mismatch between the decoder and encoder, is not supported because it is not a point-independent operation that requires the sum of all the coefficients.

4.4. Point Based IDCT

Our IDCT on GPUs are based on the fact that the 2-D image data is given by the linear combination of basis images with

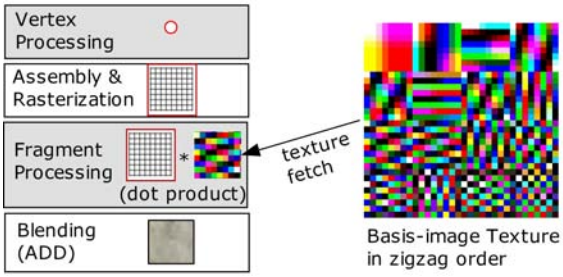

Figure 3: *Schematic illustration of our point-based IDCT. By rasterization IDCT is transformed to per-fragment vector operations. Basis images are packed in a fp16 RGBA texture.*

their respective DCT coefficients, as described in Equation 1. The process includes two steps: the scalar-matrix multiplication and the matrix linear combination. In our implementation, we transform multiplications into per-fragment operations through rasterization and combine the results from separate points by blending. Figure 3 gives an overview of the procedure.

The basis images are pre-arranged into a single texture. Considering the precision and performance, we choose RGBAF16 format for our basis-image texture, as shown in the right of Figure 2. The basis images are arranged according to our point-generation pattern. Four basis images corresponding to a certain 4D coefficient vector are packed into a 8×8 RGBA texture block.

The procedure is described in details as follows.

1. A vertex program locates the coordinates of a corresponding basis-image texture block according to the coefficient vector's information.
2. The fragments that cover a block inherit the point's attributes by rasterization. In this way, we transform scalar-matrix multiplications into per-fragment operations.
3. The local coordinates within the block are generated by the point sprite extension. Combining with the location of the texture block, we can exactly access the corresponding basis-image texel for each fragment.
4. A fragment program fetches the texels and performs a vector dot product to efficiently accomplish the multiplications and the combination for four individual coefficients.
5. By enabling blending and setting to *add* function, the results from the separate points of one video block are accumulated in the IDCT off-screen buffer.

Compared with the texture-based methods, our point-based IDCT has several advantages. First, only a one-pass rendering is needed to perform both IQ and IDCT. Meanwhile, the inverse scan is completely removed. Secondly, most zero coefficients have been culled to save computations and much less texture fetching and arithmetic operations in fragment programs are required. Thirdly, the visual quality and computational complexity can be controlled by rendering only a part of coefficient points. Finally, the output IDCT off-screen buffer can be directly used by the following motion compensation. No needs for extra unpacking operations.

4.5. Point Based Motion Compensation

Motion compensation is implemented by rendering marcorblock points. Motion vectors (MVs) are pre-processed in a vertex program according to their precisions. Then, each fragment within the macroblock inherits the MVs by rasterization. In a fragment program, we offset the window position coordinates (*WPOS*) with MVs to sample the reference pictures in the MC buffers and fetch the residual data in the IDCT buffer. The combined results are saturated to an off-screen buffer for further processing. Compared with the quad-based method, our point-based MC greatly reduces the number of vertices and easily utilizes the vertex processors.

If the bilinear interpolation is adopted for the sub-pixel MC, such as MPEG2, we can further exploit the texture bilinear filtering hardware to facilitate the processing. Since the decimal parts of the texture coordinates directly decide how to interpolate four neighboring pixels, we use a vertex program to generate the proper decimal parts for MVs according to their precisions. When the fragment program fetches the texels, the texture unit automatically performs the interpolation, which greatly simplifies the fragment program by removing the branches and the complex flow-control. However, when handling the interlaced frame structure, we have to explicitly fetch the texels in the vertical direction and perform the interpolation in the fragment program. The parities of *WPOS* are used to choose the corresponding field-prediction MVs.

5. Evaluation and Analysis

To verify the effectiveness of our proposed method, we implement a MPEG-2 video decoder as a practical example. We compare the decoder's performance against other CPU-based and GPU-based solutions. All the experiments were run on a 2.8G Pentium 4 with an Nvidia Geforce 6800 GT. Our programs are implemented with OpenGL and Cg 1.4. The competitive CPU decoder is based on Sklmpeg4 [Skl05], which have been highly optimized with the CPU SIMD assembly codes.

5.1. Decoding Performance

Five different implementations are presented to evaluate the performance.

- CPU-Only. All the work is done on the CPU including the color space conversion (CSC).
- CPU-noCSC. The CSC is moved to the GPU, which is common in the practical decoders.

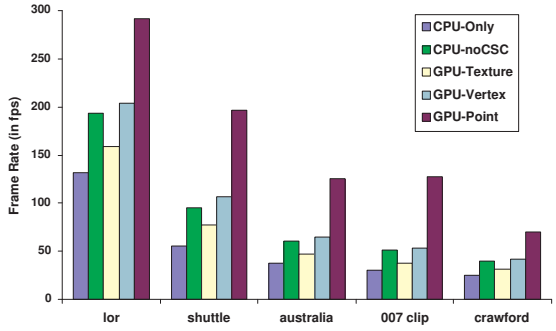

Figure 4: *Comparison of decoding performance. Our proposed solution significantly outperform the others.*

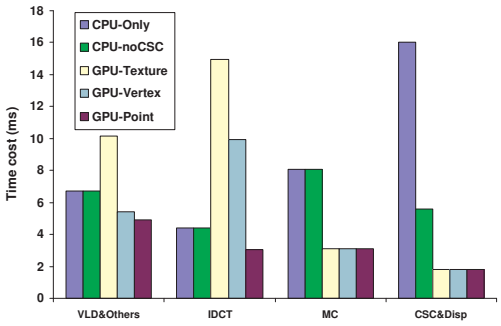

Figure 5: *Time costs of main decoding modules . Measured on the clip* australia*. The IDCT and the CSC&Disp take into account the data transfer costs for the GPU-based and CPU-based solutions respectively.*

- GPU-Texture. We implement the IDCT using direct matrix multiplications which is similar to the SMMCM method proposed in [FSLC05]. The DCT coefficients are transformed from *short* to *half* on the CPU and stored as a RGBA fp16 texture. The IQ is done on the CPU and the MC is our point-based method.
- GPU-Vertex. Very similar to the GPU-Texture, but we use vertices to represent sparse coefficients and render them at the correct position in the source buffer of the texture-based IDCT. The idea comes from handling the sparse random matrices as described in [KW03]. We apply it to sparse DCT coefficients to investigate how the transfer cost impact the overall performance.
- GPU-Point. It is our proposed method. Only the VLD is done by the CPU.

To get additional efficiency, we use pixel buffer objects (PBOs) to update textures and utilize early Z-culling [MS04] to skip non-coded video blocks for the texture-based IDCT. The early Z-culling is implemented through a block-coding-indicator texture. Before IDCT, we use a fragment program to sample the texture and modify the Z-buffer through *DEPTH* semantics.

Clip	Resolution	Bit-Rate (Mbps)	Frame Structure
lor	720×480@29.97	4.6 vbr	progressive
shuttle	1280×720@59.94	15.5 vbr	progressive
australia	1440×1088@25.00	12.3 cbr	interlaced
007	1920×1080@29.97	10.9 vbr	progressive
crawford	1920×1080@29.97	30.0 cbr	interlaced

Table 1: *Characteristics of five different test clips.*

We choose five different MPEG-2 video clips to evaluate the performance, as shown in Table 1.The clip *lor* (Lord of the Ring) and *007* (Die another day) are commercial DVD and HD movie clips respectively. The others are captured from HDTV or taken with HD cameras, which are aviable through links of the clip list in [VHD05].

Our proposed point-based method significantly outperforms the prior optimized CPU and GPU based implementations, as shown in Figure 4. Most frame rates exceed 120fps, including the 1080p clip. Our results indicate a factor of 1.7-2.4 performance speedup over the second-fastest GPU-Vertex. The performance of the clips strongly depends on the bit-rate; the interlaced frame structure also leads to performance penalties due to more texel fetching and arithmetic operations. Therefore, *crawford* has the lowest frame rate. The speedup for *lor* is not obvious, which can be explained by its high frame-rate and low resolutions. The former increases the costs for state switching in OpenGL and Cg; the latter results in smaller working data set which is more suitable for the CPU's architecture.

To understand the various performances of our five implementations, we analyze the time costs of decoding modules. The execution time on the GPU is measured by putting *glFinish* at the end of the rendering. We take the clip *australia* as an example and give the statistics in Figure 5. Because the CPU and GPU are pipelined to work in parallel, the final frame rate is actually determined by the most time consuming stage. For three GPU-based implementations Figure 5 indicates that the GPU related operations cause the bottleneck. Therefore, the performance improvement of the GPU-Vertex over the GPU-Texture is determined only by their IDCT costs, which explains why the speedup is not significant. In terms of MC, our results demonstrate the efficiency of our point-based method and indicate a factor of 1.8–3.1 performance improvement over the CPU. In the terms of IDCT , we observe our method offers great advantages over the other GPU and CPU implementations. It is nearly 1.5 times faster than the CPU SSE2 IDCT and 3–5 times faster than the texture-based GPU-IDCT.

Solution	Transfer time (ms)			IDCT time (ms)		
	I	P	B	I	P	B
G-Texture	7.06	7.13	7.18	9.98	8.87	7.06
G-Vertex	4.76	3.12	1.64	10.06	9.12	7.42
G-Point	0	0	0	6.51	4.20	2.26
CPU-SSE2	0	0	0	7.12	5.97	3.52

Table 2: *Comparison of time costs (australia) according to frame types. Our point-based IDCT incorporates the data transfer and IQ, so here we set the transfer cost to zero.*

Sequence	Average PSNR (db)	PSNR Degradation (db)		
		I	P	B
stefan	31.722	0.006	0.008	0.021
mobilecal	31.134	0.003	0.010	0.030
foreman	37.245	-0.011	0.027	0.055

Table 3: *Quality degradation of our GPU-Point decoder compared with the CPU decoder. All the degradation results are less than 0.06 db and indicate our method is nearly degradation free.*

We highlight more details of IDCT in our implementations, see Table 2. Compared with the texture-based coefficient uploading, our point-based representation is flexible enough to exclude zero values. In addition, the less IDCT time costs of P and B frames from the texture-based IDCT indicate our early Z-culling is effective to skip the non-coded blocks. But their performance is still much lower than the optimized rival on the CPU. For our point-based IDCT, although it incorporates the IQ and data transfer, it is still much faster than the CPU . We also observe the time cost is directly proportional to the point number, which agrees well with the prior theoretical analysis.

Our method is fragment processing bound, which was identified by our overclock experiments on the GPU. A higher core frequency from 350M to 400M leads to a frame rate speedup form 124 to 133 for the clip *australia*, while a higher memory frequency takes neglectable effect. This offers our method an additional advantage because compared with the memory bandwidth, the fragment processing power is much easilier improved by more fragment processors and higher frequency.

5.2. Visual Quality

Visual quality is another important issue to evaluate a decoder. To objectively measure the visual quality, we encoded three representative MPEG test sequence and adopted PSNR(Y component) to compare the decoded frames from our GPU-Point and the CPU decoders. Each sequence, composed of 300 CIF(352×288) frames, is encoded by the MPEG-2 reference software (TM5) with a GOP size 15 at the bit rate of 2.0 Mbps. Their PSNR results are nearly identical , here we give the average differences in Table 3 according to different frame types.

The results show very slight quality degradation from our point-based solution. This may be explained by the following three aspects: the lack of mismatch control in the IQ, the low precision of half for IDCT computations and the rounding issues of the bilinear filtering of the texture unit. As we notice that the inter frames (P and B) have a little higher degradations than the intra frames (I), we infer that the rounding control for sub-pixel interpolations in motion compensation could be the main factor to impact the visual quality. In addition, no drift-error accumulation was observed for the GOP structure with 15 frames.

6. Discussion

In this section, we discuss limitations of our proposed method and address some issues of GPUs for generic media/video processing.

Our method for video decoding has several limitations. First, our IDCT implementation requires fp16 blending to ensure the precision and high pixel fill-rate to ensure the performance, which challenges most of today's low-end GPUs. But with the popularity of HDR and the fast evolution of hardware we believe it can be solved in the near future. Secondly, the point primitive constrains its shape to be a square, which would cause troubles when handling some non-square motion compensation(such as 16x8, 8x4). One possible solution is to split the rectangular region into two adjacent squares. Thirdly, advanced non-bilinear sub-pixel interpolations can not directly benefit from the current texture units, In that case explicit texture fetching and preprocessing are required. Finally, the deblocking filter and the intra prediction in some advanced video standards are not concerned in this study. High correlations of blocks in those operations make it a big challenge to implement them efficiently on current graphics hardware.

Based on the performance of our GPU decoder and lots of previous GPGPU applications, such as image and signal processing [OLG*05], we can expect that the GPU can evolve into a promising and powerful platform for a broad class of media processing applications. Most of them fit well with the stream processing model, but some applications feature frequent memory accesses and data reuse, such as motion estimation and high order filtering. However, according to previous works [FSH04] [CCLW05], applications on GPUs with high memory accesses always suffer in efficiency due to the relatively low bandwidth connection between arithmetic units and local memories. Including wider and closer caches or register level blocking has been proposed to overcome this issue [FSH04]. In addition, considering that texture units on current GPUs have considerable computational resources to perform bilinear, trilinear or anisotropic filtering and advanced filter kernels can often dissected into groups of bilin-

ear footprints, it would be possible to enhance texture units to more flexible configurable or programmable units, which could work in parallel with arithmetic units and directly improve the performance of filtering related operations. Some GPGPU operations, such as *sum* and *reduction*, would also benefit from it.

7. Conclusions and Future Work

In this paper, we have presented a novel point-based framework for efficient video decoding on GPUs. We analyzed the characteristics of decoding process and proposed the point-based representation for video blocks, which fits well with the GPU's stream processing model. Furthermore, our new IDCT implementation efficiently removes most zero-value coefficients to save the bandwidth and computation. It fully takes advantage of the graphics pipeline and leads to much less memory accesses and arithmetic operations. Moreover, Our motion compensation reduces data redundancy and utilize texture units to facilitate sub-pixel interpolations. We have demonstrated the efficiency of our proposed framework by a MPEG-2 GPU decoder. Our results indicate a significant improvement over prior CPU and GPU solutions.

There are many avenues for future work. We will evaluate point generation patterns to improve our IDCT performance and investigate new capabilities of GPUs for video coding operations. We also expect to apply our method to more video standards and even the future HDR video. Furthermore, we plan to utilize GPUs to accelerate complex and expensive motion estimation in the video encoding process.

References

[App05] Apple core image&video libary, 2005. http://www.apple.com/macosx/features/coreimage.

[CCLW05] COPE B., CHEUNG P. Y. K., LUK W., WITT S.: Have gpus made fpgas redundant in the field of video processing? In *Proceedings of the 2005 IEEE International Conference on Field-Programmable Technology* (2005), pp. 111–118.

[DNVRH*05] DE NEVE W., VAN RIJSSELBERGEN D., HOLLEMEERSCH C., DE COCK J., NOTEBAERT S., VAN DE WALLE R.: Gpu-assisted decoding of video samples represented in the ycocg-r color space. In *Proceedings of the 13th ACM International Conference on Multimedia* (Singapore, 11 2005), pp. 447–450.

[FSH04] FATAHALIAN K., SUGERMAN J., HANRAHAN P.: Understanding the efficiency of gpu algorithms for matrix-matrix multiplication. In *Proceedings of the ACM SIGGRAPH/Eurographics conference on Graphics hardware* (2004), pp. 133–137.

[FSLC05] FANG B., SHEN G., LI S., CHEN H.: Techniques for efficeitne dct/idct implementation on generic gpu. In *Proceedings of IEEE International Symposium on Circuits and Systems* (2005), pp. 1126–1129.

[HL05] HIRVONEN A., LEPPÄNEN T.: H.263 video decoding on programmable graphics hardware. In *Processing of IEEE International Symoposium on Signal Processing and Information Technology* (2005), pp. 902–907.

[JL05] J. LEE N. VIJAYKRISHNAN M. J. I.: High-performance array processor for video decoding. In *Proceedings of the VLSI Design Conference* (January 2005).

[KB04] KOBBELT L., BOTSCH M.: A survey of point-based techniques in computer graphics. *Computers & Graphics 28*, 6 (December 2004), 801–814.

[KK04] KELLY F., KOKARAM A.: Fast image interpolation for motion estimation using graphics hardware. In *Proceedings of the IS&T/SPIE Electronic Imaging.* (May 2004), pp. 184–194.

[KKKW05] KRÜGER J., KIPFER P., KONDRATIEVA P., WESTERMANN R.: A particle system for interactive visualization of 3d flows. *IEEE Transactions on Visualization and Computer Graphics 11*, 6 (Nov/Dec 2005), 744–756.

[KSW04] KIPFER P., SEGAL M., WESTERMANN R.: Uberflow: a gpu-based particle engine. In *HWWS '04: Proceedings of the ACM SIGGRAPH/Eurographics conference on Graphics hardware* (New York, NY, USA, 2004), ACM Press, pp. 115–122.

[KW03] KRÜGER J., WESTERMANN R.: Linear algebra operators for gpu implementation of numerical algorithms. *ACM Transactions on Graphics 22*, 3 (2003), 908–916.

[MS04] MITCHELL J. L., SANDER P. V.: Applications of explicit early-z culling. *Real-Time Shading Course, SIGGRAPH* (2004).

[Nvi05] Nvidia SDK code samples: discrete cosine transform, 2005. http://developer.nvidia.com/object/sdk_home.html.

[OLG*05] OWENS J. D., LUEBKE D., GOVINDARAJU N., HARRIS M., KRĹŹGER J., LEFOHN A. E., PURCELL T. J.: A survey of general-purpose computation on graphics hardware. In *Eurographics 2005, State of the Art Reports* (Aug. 2005), pp. 21–51.

[ORK*02] OWENS J. D., RIXNER S., KAPASI U. J., MATTSON P., TOWLES B., SEREBRIN B., DALLY W. J.: Media processing applications on the imagine stream processor. In *Proceedings of the IEEE International Conference on Computer Design* (Sept. 2002), pp. 295–302.

[SGL*05] SHEN G., GAO G., LI S., SHUM H.-Y., ZHANG Y.-Q.: Accelerate video decoding with generic gpu. *IEEE Transactions on Circuits and Systems for Video Technology 15* (May 2005), 685–693.

[Skl05] Skal's MPEG4 codec, 2005. http://skal.planet-d.net/coding/sklmp4.html.

[VHD05] High def forum: The offical hd video clip list, 2005. http:http://www.highdefforum.com/showthread.php?t=6537.