Graphics Hardware for Gradient Based Motion Estimation

Francis Kelly and Anil Kokaram

Department of Electronic and Electrical Engineering, University of Dublin,
Trinity College, Dublin 2, Ireland

ABSTRACT

Motion estimation and compensation is a key component in video processing. Motion estimation is necessary for high quality video compression. It is also a key component in archive video restoration and motion picture post-production. Very accurate motion vectors are usually required in the latter two applications. More accurate motion vectors can also lead to greater coding efficiency. Real-time, accurate motion estimation is currently not attainable on standard desktop PCs. It usually requires some kind of dedicated hardware such as on video coding chips. Gradient based motion estimation is one which gives good accuracy for reasonable computational cost. This paper uses the Wiener based motion estimator as a vehicle to explore the acceleration of gradient based motion estimation on the PC.

Keywords: Gradient Based Motion Estimation, Programmable Graphics Hardware, Image Interpolation

1. INTRODUCTION

Modern graphics hardware contain very powerful processing units. The performance growth of these GPUs is in fact greater than Moore's Law for CPUs. These GPUs are also becoming much more programmable as the computer games industry searches for more realistic and impressive visual effects. With improving programmability and floating point operation these GPUs are becoming an attractive option for more general purpose computing than they were designed for.

It is not however a straight forward thing to use the GPU as a general purpose processor. The algorithm in question has to be adapted to fit in with the GPUs data format, data structure, program execution model, etc. As an example of using the GPU as a co-processor to the CPU, a novel implementation of a well known motion estimation algorithm is presented which uses the GPU to accelerate sections of the algorithm.

Motion estimation is a key component in video processing. Motion estimation and compensation is necessary for high quality video compression.[1] It is also a key component in archive video restoration[2] and in motion picture post-production and special effects.[3] Motion estimation and compensation is also useful for scan rate conversion in broadcast television.[4]

Motion estimation is however very computational intensive. Typically dedicated hardware such as on a MPEG encoding chip is needed for accurate real-time motion estimation. On the PC real-time motion estimation is some way off. Because the motion in real sequences is typically non-integer, sub-pel accurate motion vectors are required to accurately motion compensate the scene. This means interpolation of the image signal is required, which adds further computational demands.

Further author information: (Send correspondence to F.K.)
F.K.: E-mail: frkelly@tcd.ie, Telephone: +353 (0)1 608 2202
Address: Department of Electronic and Electrical Engineering, University of Dublin, Trinity College, Dublin 2, Ireland
A.K.: E-mail: anil.kokaram@tcd.ie, Telephone: +353 (0)1 608 3412
Address: Department of Electronic and Electrical Engineering, University of Dublin, Trinity College, Dublin 2, Ireland
Web: `http://www.mee.tcd.ie/~sigmedia`
This project was funded in part by Enterprise Ireland and Trinity Foundation.

2. MOTION ESTIMATION

There are three main types of motion estimation: block matching methods, phase correlation methods, and gradient based methods. In most algorithms the image is divided up into blocks with one motion vector estimated for each block. The model used here for image sequence processing is a translational one, and is defined as

$$I_n(\mathbf{x}) = I_{n-1}(\mathbf{x} + \mathbf{d}_{n,n-1}), \tag{1}$$

where $I_n(\mathbf{x})$ is the grey level of the pixel at the location given by position vector $\mathbf{x}$ in the frame n, and $\mathbf{d}_{n,n-1}$ is a displacement mapping the region in the current frame n into the previous frame $n-1$. Using this model, the grey level at each pixel in the current frame can be predicted from a shifted location in the previous frame. The only parameter that is required for this model is the motion vector $\mathbf{d}$.

Motion estimation algorithms can be split into their three main categories by the means in which they search for this motion parameter. Block matching algorithms usually search a candidate set of motion vectors, choosing the motion vector for each block which minimises some function of the Displaced Frame Difference (DFD)

$$\mathbf{DFD}(\mathbf{x}, \mathbf{d}) = I_n(\mathbf{x}) - I_{n-1}(\mathbf{x} + \mathbf{d}). \tag{2}$$

The most common matching criteria are Mean Square Error (MSE), Mean Absolute Difference (MAD), and Sum Absolute Difference (SAD). This method is quite simple to implement and most video codecs use block matching for motion estimation, including MPEG-2, MPEG-4, and H.263.

Gradient based methods and phase correlation methods try and estimate the motion parameter directly. Phase correlation methods[5] estimate the displacement between two image blocks by means of a normalized cross-correlation function in the 2-D spatial Fourier Domain. It is based on the principle that a relative shift in the spatial domain results in a linear phase term in the Fourier domain.[6] A 2-D discrete Fourier transform of the relevant blocks in the two frames is firstly computed. From these a phase correlation function of the two blocks is established. The location of peak(s) in this phase-correlation function indicates the relative displacement between the two blocks. This technique has been implemented on a chip for real-time motion compensated archive video restoration by Snell & Wilcox.[7]

2.1. Gradient Based Motion Estimation

Gradient based motion estimation techniques arise by rearranging equation 1 to have direct access to the motion parameter $\mathbf{d}$. By taking a Taylor series expansion of equation 1, it may be linearized about $\mathbf{d}$ to give

$$I_n(\mathbf{x}) = I_{n-1}(\mathbf{x}) + \mathbf{d}^T \nabla I_{n-1}(\mathbf{x}) + e_{n-1}(\mathbf{x}) \tag{3}$$

where ∇ is the multidimensional gradient operator,$\nabla I_{n-1}(\mathbf{x}) = [\frac{\partial I_{n-1}(\mathbf{x})}{\partial x} \ \frac{\partial I_{n-1}(\mathbf{x})}{\partial y}]$, and $e_{n-1}(\mathbf{x})$ represents the higher order terms of the expansion. Using equation 2, this can be rearranged to give an expression involving the DFD with zero displacement

$$\begin{aligned} \mathbf{DFD}(\mathbf{x}, 0) &= I_n(\mathbf{x}) - I_{n-1}(\mathbf{x}) \\ &= \mathbf{d}^T \nabla I_{n-1}(\mathbf{x}) + e_{n-1}(\mathbf{x}). \end{aligned} \tag{4}$$

Neglecting the higher order terms allows an equation to be set up at each pixel in a block. This can be expressed as a vector equation, where the vector $\mathbf{z}_0$ and matrix $\mathbf{G}$ are made up of the DFD and gradient values for the block respectively.

$$\mathbf{z}_0 = \mathbf{G}\mathbf{d} \tag{5}$$

A solution for $\mathbf{d}$ can then be found in one step using a pseudo inverse approach[8]

$$\mathbf{d} = [\mathbf{G}^T\mathbf{G}]^{-1}\mathbf{G}^T\mathbf{z}_0. \tag{6}$$

Because the Taylor series expansion is only valid over very small distances, this method can only be used to estimate very small displacements. To overcome this problem a recursive solution called the pel-recursive method

is used.[9] In pel-recursive methods equation 1 is linearised about a current guess for $\mathbf{d}$, say $\mathbf{d}_i$. An update $\mathbf{u}_i$ is defined as the difference between this estimate $\mathbf{d}_i$ and the true displacement $\mathbf{d}$

$$\mathbf{u}_i = \mathbf{d} - \mathbf{d}_i. \tag{7}$$

Equation 3 then becomes

$$I_n(\mathbf{x}) = I_{n-1}(\mathbf{x} + \mathbf{d}_i) + \mathbf{u}_i^T \nabla I_{n-1}(\mathbf{x} + \mathbf{d}_i) + e_{n-1}(\mathbf{x} + \mathbf{d}_i) \tag{8}$$

There are many different methods of solving this equation to yield the displacement $\mathbf{d}$. These include a least squares approach for finding a direct solution,[10] and optical flow based techniques.[11] There are also methods which refine the motion estimation solution by employing constraints on the motion itself.[12] Rather than explore all these we concentrate on the Wiener based motion estimator algorithm instead.

2.1.1. Wiener Based Motion Estimation

Biemond et al[13] presented a Wiener solution for the displacement which is robust to noise. It considers the effect of the higher order terms on the solution as being the same as Gaussian white noise.

$$\mathbf{z}_i = \mathbf{G}\mathbf{u}_i + \mathbf{e}, \tag{9}$$

This is a pel-recursive solution which generates updates based on the current estimate of the motion vector, $\mathbf{d}_i$. The update is intended to be small and should, over many iterations, force the estimation process to converge on the correct motion vector. The solution for the update $\mathbf{u}_i$ of the current estimate $\mathbf{d}_i$ is then

$$\begin{aligned} \mathbf{u}_i &= [\mathbf{G}^T\mathbf{G} + \mu\mathbf{I}]^{-1}\mathbf{G}^T\mathbf{z} \\ \mu &= \frac{\sigma_{ee}^2}{\sigma_{uu}^2} \end{aligned} \tag{10}$$

Here, σ_{uu}^2 is the variance of the estimate for $\mathbf{u}_i$ and σ_{ee}^2 is the variance of the Gaussian white noise representing the higher order terms. After the update is estimated, $\mathbf{d}_i$ is refined to yield the displacement that is used in the next iteration, $\mathbf{d}_{i+1} = \mathbf{d}_i + \mathbf{u}_i$. This process is repeated until some convergence criteria has been met. We will refer to this estimator as the Wiener based motion estimator (WBME).

The slowest part of this algorithm is generating the two matrix multiplies, $G^T z$ and $G^T G$. The G matrix is a $N^2 \times 2$ element matrix, where N is the size of the block. Calculating $G^T G$ requires a $(2 \times N^2) \times (N^2 \times 2)$ matrix multiplication. Similarly $G^T z$ is a $(2 \times N^2) \times (N^2 \times 1)$ matrix-vector multiplication. Also the update u_i is typically a fractional value and hence the current estimate for the displacement is a fractional number. A fractional displacement means that the image signal has to be interpolated to generate $\mathbf{G}$ and $\mathbf{z}$. This interpolation adds further to the computational complexity of the algorithm.

2.2. Hierarchical Motion Estimation

A hierarchical representation of images may be used to improve motion estimation. A multiresolution representation of an image, built up in the form of a Laplacian pyramid, is depicted in figure 1. The base of the pyramid, level 0, is the original image. Each subsequent level in the pyramid is obtained by low-pass filtering and subsampling the previous level. Typically each level is subsampled by a factor of two. Motion estimation is then performed from coarse to fine levels with motion vectors refined at each level. Hierarchical methods have been used for block matching[14] and phase correlation.[15]

Hierarchical methods are particularly useful for gradient based algorithms as they allow for the estimation of large displacements. By definition, the updates that are generated should be small to keep the Taylor Series expansion valid. Using a pel-recursive method is not enough if the motion is very large, for example greater than ± 10 pixels. When the image is subsampled the displacement in the image is also reduced. Because at the lowest resolution levels the displacement is small, the WBME can be used to determine a rough estimate of the displacement in the image at this level. The estimated vectors are passed to the higher resolution levels as an initial estimate of the displacement. The higher resolution levels can then fine-tune the displacement vector for accurate motion estimation.

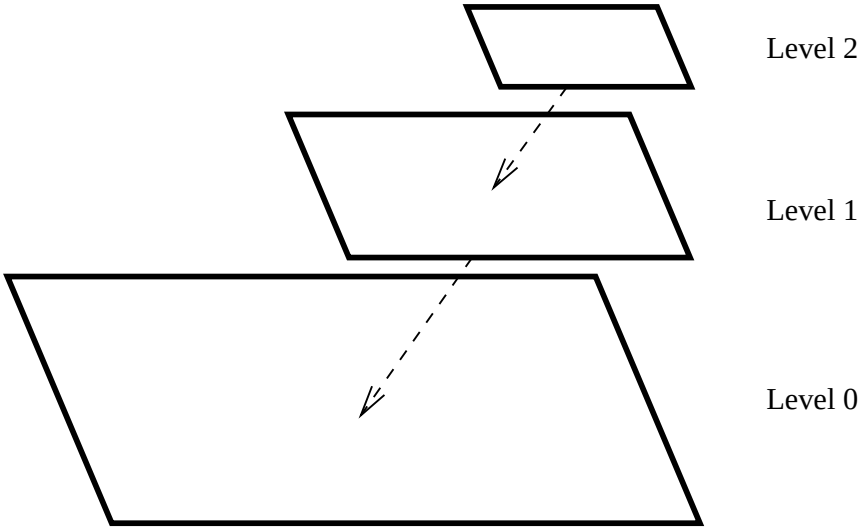

Figure 1. Three level pyramid for multiresolution scheme.

2.2.1. Low-pass Filter

The low-pass filtering is usually performed with a gaussian filter kernel. The Intel Integrated Performance Primitives library has a function for pyramid generation which uses the following gaussian kernel

$$K = \frac{1}{256}\begin{bmatrix} 1 & 4 & 6 & 4 & 1 \\ 4 & 16 & 24 & 16 & 4 \\ 6 & 24 & 36 & 24 & 6 \\ 4 & 16 & 24 & 16 & 4 \\ 1 & 4 & 6 & 4 & 1 \end{bmatrix}. \tag{11}$$

This corresponds to a 2-D gaussian with variance 1.0. A simpler low-pass filter to use is the box filter. This simply replaces each pixel by a local mean

$$K = \frac{1}{4}\begin{bmatrix} 1 & 1 \\ 1 & 1 \end{bmatrix}. \tag{12}$$

Figure 6 compares the effect of the gaussian filtering kernel and the box filter in pyramid generation, for a multiresolution WBME algorithm. It shows the peak signal to noise ratio (PSNR) of the motion compensated DFD for four test sequences. As can be seen, there is very little difference in results between the two. This suggests that box filtering is sufficient for pyramid generation in multiresolution WBME, for video coding particularly where PSNR is a good indicator of performance. It should be noted that the accuracy of the motion vectors generated by the two methods may be different. This is not always evident in a PSNR comparison where different motion vectors can yield similar low prediction errors. However only one of these vectors may correspond to the true motion of the block. In applications such as video restoration, vector accuracy can be more important than PSNR performance.

3. PROGRAMMABLE GRAPHICS HARDWARE

Modern computer graphics hardware contain extremely powerful graphics processing units (GPU). The performance of these GPUs is also growing at an extraordinary rate. Over the last ten to fifteen years this processing power has been growing at a rate faster than Moore's Law, which governs the the performance growth rate of CPUs.[16] A recent presentation at Graphics Hardware 2003 titled "Data Parallel Computing on Graphics Hardware", estimated that the current Nvidia GeForce FX 5900 GPU performance peaks at 20 GigaFlops. This is equivalent to a 10-GHz Pentium 4 processor. GPUs get their impressive speeds by doing stream processing.[17]

Stream processors are designed to work on large amounts of data which arrive continuously. These GPUs are designed to perform a limited number of operations on very large amounts of data. They typically have more than one processing pipeline working in parallel with each other.

The latest generation of graphics hardware also now contain programmable GPUs. Previously the number of operations that could be performed on the GPU was limited to fixed functionality such as Texture and Lighting. However the GPU has evolved to a situation now, where there are user programmable vertex and texture units. These are commonly referred to as vertex shaders and fragment shaders respectively. These programmable shaders allow for much more realistic visual effects in todays computer games especially, which is the main driving force behind the computer graphics hardware industry.

A further improvement in the latest generation GPUs is the increase in pixel accuracy from 32 bits per pixel to 128 bits per pixel. This means that each pixels red, green, blue, and alpha component can now have 32-bit floating point accuracy throughout the graphics pipeline. This increase in data accuracy combined with the increased programmability of the GPU makes it an attractive option for doing more general purpose computing than the graphics operations for which it was designed. In particular it may be thought of as a useful co-processor to the CPU for image processing tasks.

This paper looks at using the GPU to accelerate motion estimation. By offloading common image processing tasks such as generating DFDs and image gradients from the CPU, we hope to speed up the WBME algorithm. In particular we wish to generate the two large matrix multiplications on the GPU. Doing this also means taking advantage of the GPUs fast interpolation hardware.

3.1. Programming the GPU

The GPU is designed as a stream processor for processing large amounts of data in parallel. Vertices are points in 3-D space which define graphics primitives like triangles, polygons, rectangles etc. These primitives are used to build up the geometry of the scene and define any 2-D or 3-D models to be displayed. Vertex programs are used to transform and process these points. Textures are images which can be mapped onto the geometry and models to add detail to a scene. A image of skin for example, can be mapped onto a model of a face to make the model look more realistic. In order to use texture mapping, texture coordinates detailing how the texture is to be applied to the model are required. These coordinates can be generated and manipulated in a vertex program along with the vertices representing the models etc. in the scene. A fragment program then uses these coordinates to index the texture image data and generate a final output pixel color.

Vertex and fragment programs are written in a special type of assembly language for these programmable GPUs. There are higher level languages such as Nvidia's Cg[18] or Mircosoft's High Level Shading Language (HLSL) available. These are similar to the C,C++ type programming languages and use compilers to generate the appropriate assembly language for the GPU. In this case however the programs are simple enough to be written directly in the assembly language. This also ensures that the programs can be fully optimised for the underlying hardware and do not use any unnecessary registers or instructions. The code in this paper is based on the OpenGL ARB fragment and vertex program standards. The following is the instruction set for fragment programs; the instruction set for vertex programs is very similar. There are operations such as add, subtract, multiply, sine, maximum, minimum, etc. There is however no support for program flow control such as branching or looping. These are features which will hopefully be implemented in the next generation of graphics hardware.

```
ABS ADD CMP COS DP3 DP4 DPH DST EX2 FLR FRC KIL LG2 LIT
LRP MAD MAX MIN MOV MUL POW RCP RSQ SCS SGE SIN SLT SUB
SWZ TEX TXB TXP XPD
```

In order to use the GPU for image processing, the images have to be downloaded to the graphics hardware as textures. The GPU can then be used to sample values from these textures and perform any necessary operations in a fragment program. A grid is made up of vertices representing the blocks in the image. A corresponding grid of texture coordinates is also established. Using these coordinates the image is texture mapped onto the grid. Offsetting the coordinates of blocks on the grid effectively means displacing the blocks in the image. The GPU will sample the blocks from the offset position before applying the texture. Each blocks texture coordinates can

be offset to correspond to the motion vector for that block. The GPU can perform bilinear interpolation while sampling from the texture if necessary. In this way it is possible to compensate all the blocks in the image in one go. Using fragment programs, per pixel operations be done quickly on the compensated image. The result of these operations are stored in pixel buffers (Pbuffers). These Pbuffers can also be used as textures if further processing of the results on the GPU is required, or else the results are read back to the CPU.

3.1.1. Image Differencing

Subtracting two images on the GPU requires the following fragment program. Firstly the two images are loaded into two texture units, texture0 and texture1. The `TEX` instruction is then used to sample the required pixels from each texture unit (lines 3 & 4). This uses the texture coordinates passed to it to sample a value from a texture, with the result stored in a temporary register. The texture unit can do bilinear interpolation if necessary when sampling the texture value. Finally the `SUB` instruction is used to subtract the two values and output the result (line 5).

```
!!ARBfp1.0
1 OUTPUT output = result.color;
2 TEMP frame1,frame2;

3 TEX frame1, fragment.texcoord[0], texture[0], RECT;
4 TEX frame2, fragment.texcoord[1], texture[1], RECT;
5 SUB output, frame1, frame2;
END
```

3.1.2. Image Gradients

Generating image gradients is also quite straightforward. In this paper the gradients are calculated using the central difference formula

$$\begin{aligned} \nabla I_{n-1}(\mathbf{x}) &= [\frac{\partial I_{n-1}(\mathbf{x})}{\partial x} \ \frac{\partial I_{n-1}(\mathbf{x})}{\partial y}] \\ \frac{\partial I_{n-1}(\mathbf{x})}{\partial x} &= \frac{I_{n-1}(x+1,y) - I_{n-1}(x-1,y)}{2} \\ \frac{\partial I_{n-1}(\mathbf{x})}{\partial y} &= \frac{I_{n-1}(x,y+1) - I_{n-1}(x,y-1)}{2}. \end{aligned} \tag{13}$$

In order to calculate the gradient at the current pixel, the pixels directly above and below, and to the left and right of the current pixel are needed. This can be done using the following vertex program to change the texture coordinates of four texture units. This program outputs four sets of texture coordinates, two each for the vertical and horizontal gradient calculation (lines 13-16).

```
!!ARBvp1.0
1 OUTPUT oPos = result.position;
2 OUTPUT oTex0 = result.texcoord[0];
3 OUTPUT oTex1 = result.texcoord[1];
4 OUTPUT oTex2 = result.texcoord[2];
5 OUTPUT oTex3 = result.texcoord[3];
6 ATTRIB iPos = vertex.position;
7 ATTRIB iTex0 = vertex.texcoord[0];
⋮
13 ADD oTex0, iTex0, {1.0, 0.0, 0.0, 0.0};
14 ADD oTex1, iTex0, {-1.0, 0.0, 0.0, 0.0};
15 ADD oTex2, iTex0, {0.0, 1.0, 0.0, 0.0};
16 ADD oTex3, iTex0, {0.0, -1.0, 0.0, 0.0};
END
```

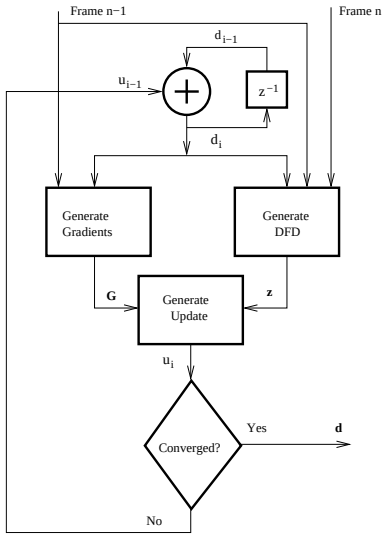

Figure 2. Flow diagram for gradient based motion estimation.

A fragment program similar to the one for subtracting two images is then used to generate the horizontal and vertical gradients for the image. It uses four sets of texture coordinates, generated in the vertex program, to access the pixels required for the gradient operation. The correct pixels needed for the gradient of the current pixel, according to equation 13, are then subtracted and the gradients are output to two channels of the output Pbuffer.

4. WBME AND THE GPU

Figure 2 shows a flow diagram depicting the operation of the WBME algorithm. Using the current estimate for the displacement of the block, $\mathbf{d}_i$, the DFD and gradients of the block are generated. These results are then grouped into the $\mathbf{z}$ vector and $\mathbf{G}$ matrix. An update for the displacement, $\mathbf{u}_i$, is then generated. A convergence test is finally performed to see if it is time to stop estimation for the block. There are usually various different checks to see when convergence is reached, including

- Stop if the magnitude of the update is below some threshold.
- Stop if the number of iterations passes some predefined limit.
- Stop if the mean square error of the DFD for the block is below some threshold.
- Stop if current mean square error is less than previous one.

If any one of these tests fail the current estimate for the displacement is output as the motion vector for that block. If all tests are passed then a new estimate for the displacement is generated using the current update and the algorithm is repeated to generate a new update.

To exploit the processing power of the GPU for WBME it is necessary to modify this approach a little. The GPU is designed for working on large volumes of data. Fragment programs operate independently for each pixel in a texture. Typically there will be more than one pixel pipeline executing fragment programs in parallel. Utilising this processing power efficiently involves operating on all the pixels in the image at once, rather than working on a block basis like on the CPU. There are two main bottlenecks in the WBME as implemented on the CPU. These are generating the DFD and gradients for each block, which requires image interpolation, and generating the matrix multiplications for the update equation. The image interpolation can be done quickly on the GPU if the images are treated as textures.

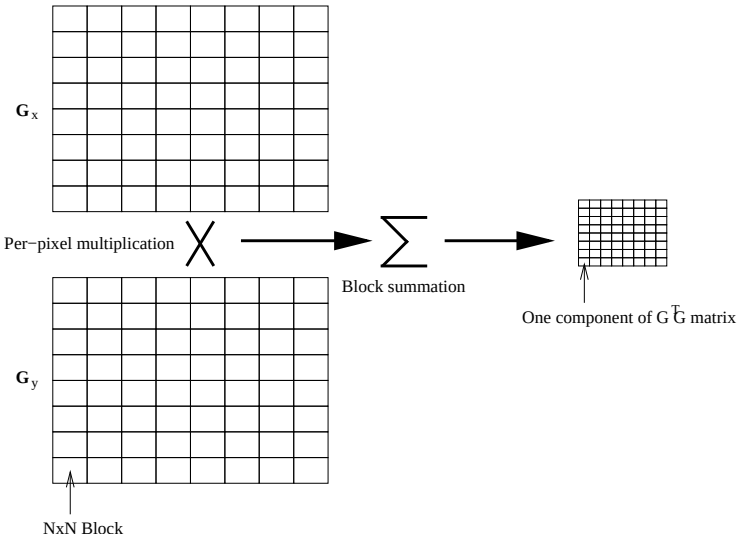

Figure 3. Generating G^TG matrix for each block.

4.1. G^TG and G^Tz

Instead of finding a motion vector for each block before moving onto the next block, an update is generated for each block in the image at each iteration. In this way the GPU can be used to accelerate the generation of the updates for each block. To do this we take advantage of the structure of the **G** and **z** matrices. For a block size of $N \times N$ pixels the composition of these matrices is as follows

$$\mathbf{z} = \begin{bmatrix} DFD(\mathbf{x}_1, \mathbf{d}_i) \\ DFD(\mathbf{x}_2, \mathbf{d}_i) \\ \vdots \\ DFD(\mathbf{x}_{N^2}, \mathbf{d}_i) \end{bmatrix}$$

$$and \tag{14}$$

$$\mathbf{G} = \begin{bmatrix} \frac{\partial}{\partial x} I_{n-1}(\mathbf{x}_1 + \mathbf{d}_i) & \frac{\partial}{\partial x} I_{n-1}(\mathbf{x}_1 + \mathbf{d}_i) \\ \frac{\partial}{\partial x} I_{n-1}(\mathbf{x}_2 + \mathbf{d}_i) & \frac{\partial}{\partial x} I_{n-1}(\mathbf{x}_2 + \mathbf{d}_i) \\ \vdots & \vdots \\ \frac{\partial}{\partial x} I_{n-1}(\mathbf{x}_{N^2} + \mathbf{d}_i) & \frac{\partial}{\partial x} I_{n-1}(\mathbf{x}_{N^2} + \mathbf{d}_i) \end{bmatrix}$$

Looking at the resulting G^TG matrix shows the following structure

$$G^TG = \sum_{p=1}^{N^2} \begin{bmatrix} G_x^2(p) & G_x(p)G_y(p) \\ G_x(p)G_y(p) & G_y^2(p) \end{bmatrix}, \tag{15}$$

where $G_x(p)$ and $G_y(p)$ are the horizontal and vertical gradients of the block with displacement, $\mathbf{d}_i$. Generating G^Tz is a similar problem. In this case **z** is a $N^2 \times 1$ element matrix. This results in a 2×1 matrix as follows

$$G^Tz = \sum_{p=1}^{N^2} \begin{bmatrix} G_x(p)z(p) \\ G_y(p)z(p) \end{bmatrix}, \tag{16}$$

where $z(p)$ is the p^{th} element of the DFD for the block with displacement, $\mathbf{d}_i$, and G_x, G_y are the gradients of the displaced block as before.

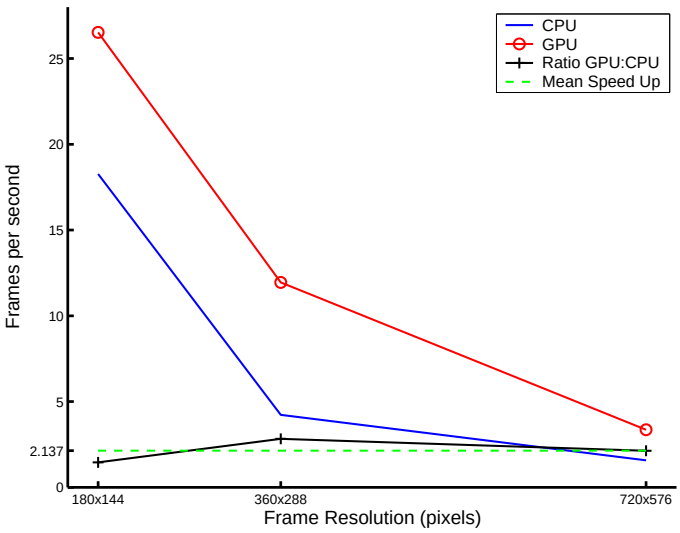

Figure 4. Average speed of WBME for CPU and GPU at three frame resolutions.

As equations 15 and 16 show, these matrix multiplications may be generated by a point-wise multiplication of the matrices followed by a summation. This multiplication can be done very efficiently on the GPU in a fragment program. Figure 3 shows how the second and third elements of the G^TG matrix can be generated for all the blocks in the image simultaneously. Firstly the image gradients for compensated frame I_{n-1} are generated on the GPU, using vertex and fragment programs described earlier. These are saved as two textures, G_x and G_y. These textures are then passed through the graphics pipeline again and a different fragment program is used to multiply the two images together. Finally the image is summed up on a block basis, giving a single value for each block in the image. This summation can be done either on the graphics card or on the CPU. Calculating the other elements of G^TG and G^Tz is performed in a similar manner.

To do the summation on the GPU the results of the matrix multiplication must be saved as textures. A sum-reduce operation is then performed where the image is summed in 2×2 blocks over multiple passes. The image is subsampled by a factor of two with each pass. For a block size of $N \times N$ pixels, $log_2 N$ passes are required. Alternatively the gradient and DFD images can be read back to the CPU for summation. However due to a PCI bandwidth problem,[19] reading these images back is very time consuming. There is a special high speed interface called the AGP bus for transferring data quickly to the graphics hardware. However reading data from the graphics hardware is performed over the relatively slow PCI bus. It is faster to do the summation on the GPU than reading back these full images across the slow PCI bus for summation on the CPU.

4.2. Multiresolution

Modern graphics hardware have built in pyramid generation schemes which use a box filter to low-pass filter each level. This is known as automatic mipmap generation. Mipmaps is the name in the graphics industry given to a hierarchical representation of textures. While downloading textures to the graphics hardware the GPU can be set to perform automatic mipmap generation. The GPU can then choose the correct mipmap level to be used depending on the size of the object that the texture is being mapped to.

As stated earlier in section 2.2.1, the results in figure 6 suggest that box filtering is good enough for pyramid generation in WBME. This means that the techniques described in the previous section can be used to accelerate a multiresolution implementation also. To start with the images are downloaded to the GPU as textures with mipmap generation enabled. By drawing a grid of blocks with resolution corresponding to the current pyramid level, the GPU will choose the correct mipmap level to sample pixels from.

4.3. Results

The results shown in this paper were generated on a 1.6 GHz Pentium 4 machine, with a Nvidia GeForce FX 5600 graphics card installed. Figure 5 shows a sample frame from the four test sequences used. These are the well known calendar/mobile sequence, foreman sequence and salesman sequence. Also included is a sequence from a 1911 Irish silent movie, called Rory O' More. There were three different image resolutions, 180x144, 360x288, and 720x576. A three level hierarchical WBME scheme was used, with the maximum number of iterations at each level set to ten.

The criteria used for comparing the various implementations is the Peak Signal to Noise Ratio (PSNR). It is defined as

$$PSNR = 20log_{10}\left(\frac{255}{MAE}\right). \tag{17}$$

The Mean Absolute Error (MAE) is calculated over all $M \times N$ pixels of the motion compensated DFD image,

$$MAE = \frac{1}{MN}\sum_{i=0}^{M}\sum_{j=0}^{N}|DFD((i,j),\mathbf{d}_{(i,j)})|, \tag{18}$$

where $\mathbf{d}_{(i,j)}$ is the displacement vector for pixel (i,j). All the pixels in a block are assigned the displacement vector for that block.

Figure 4 shows the average frames per second for WBME on both the CPU and GPU. The GPU implementation is approximately twice as fast as the CPU implementation, for the three different resolutions tested. Real-time performance of 25 frames per second is achieved at QCIF resolution.

Figure 6 details the PSNR performance for the four test sequences. There are four sets of results for each sequence: a CPU implementation and GPU implementation using both gaussian and box filtering in the pyramid generation scheme. As the results show, there is very little difference in PSNR performance between each method. There also seems to be little to choose between which type of low-pass filter to use. The Intel library gaussian filter is highly optimised for the Pentium 4 processor and does not have any effect on the timings. However using the built-in pyramid generator with the GPU makes implementation slightly more straightforward. The images for the current frames only have to be downloaded to the graphics hardware once at the start of each frame, rather than downloading each level of the pyramid separately.

5. CONCLUSION

In this paper we show how to implement a novel hierarchical Wiener based pel-recursive motion estimator, which uses the GPU as a co-processor to the CPU. We use the GPU for some fundamental image-processing functions such as image addition, subtraction, multiplication as well as doing image interpolation and finding image gradients. We used these as building blocks to implement our WBME. We found that we could achieve on average a two fold speed increase while using the GPU in this way, with little change in PSNR performance. We also investigated the effects of using a box filter in the pyramid generation scheme on our hierarchical motion estimator. The GPU has a built in pyramid generator which uses a box filter to low-pass filter each level. Surprisingly this was found to have little effect on the result.

REFERENCES

1. T. Koga, K. Iinmua, A. Hirano, Y. Iijima, and T. Ishiguro, "Motion-compensated interframe coding for video conferencing," in *Proceedings NTC'81 (IEEE)*, pp. G.5.3.1–G.5.3.4, 1981.
2. A. Kokaram, *Motion Picture Restoration*, Springer-Verlag, May 1998.
3. A. Kokaram, B. Collis, and S. Robinson, "A bayesian framework for recursive object removal in movie post-production," in *IEEE International Conference on Image Processing, Barcelona*, September 2003.
4. G. de Haan, P. Biezen, H. Huijgen, and O. A. Ojo, "True-motion estimation with 3-d recursive search block matching," *IEEE Transactions on Circuits and Systems for Video Technology* , pp. 368–379, October 1993.

5. G. A. Thomas, "Television motion measurement for DATV and other applications," tech. rep., BBC RD 1987/11, 1987.
6. A. M. Tekalp, *Digital Video Processing*, Prentice Hall signal processing series, 1995.
7. *Snell & Wilcox: Archangel Ph.C Image Restoration System.*
8. D. M. Martinez, *Model-based motion estimation and its application to restoration and interpolation of motion pictures.* PhD thesis, Massachusetts Institute of Technology, 1986.
9. A. Netravali and J. Robbins, "Motion-compensated television coding: Part 1.," *The Bell System Technical Journal* , pp. 59:1735–1745, November 1980.
10. C. Cafforio and F. Rocca, "Methods for measuring small displacements of television images," *IEEE transactions on Information Theory* , pp. 22:573–579, 1976.
11. B. Horn and B. Schunck, "Determining optical flow," *Artificial Intelligence* , pp. 17:185–203, 1981.
12. H. Nagel and W. Enkelmann, "An investigation of smoothness constraints for the estimation of displacement vector field from image sequences," *IEEE Transactions on Pattern Analysis and Machine Intelligence* , pp. 8:565–592, September 1986.
13. J. Biemond, D. E. Boekee, L. Looijenga, and R. Plompen, "A pel-recursive wiener based displacement estimation algorithm," *Signal Processing* **13**, pp. 399–412, 1987.
14. M. Bierling, "Displacement estimation by hierarchical block-matching," in *Proceedings of Visual Communications and Image Processing*, **SPIE vol. 1001**, pp. 942–951, 1988.
15. Y. M. Erkam, M. I. Sezan, and A. T. Erdem, "A hierarchical phase-correlation method for motion estimation," in *Proceedings Conference on Information Science and Systems*, pp. 419–424, 1991.
16. M. C. Lin and D. Manocha, "Interactive geometric computations using graphics hardware," in *SIGGRAPH'02 Tutorial Course 31*, 2002.
17. M. Macedonia, "The gpu enters computing's mainstream," *IEEE Computer* , October 2003.
18. W. Mark, R. Glanville, K. Akeley, and M. Kilgard, "Cg: A system for programming graphics hardware in a c-like language," in *Proceedings of SIGGRAPH 2003*, July 2003.
19. F. Kelly and A. Kokaram, "Fast image interpolation for motion estimation using graphics hardware," in *Real Time Imaging VIII*, **SPIE vol. 5297**, January 2004.

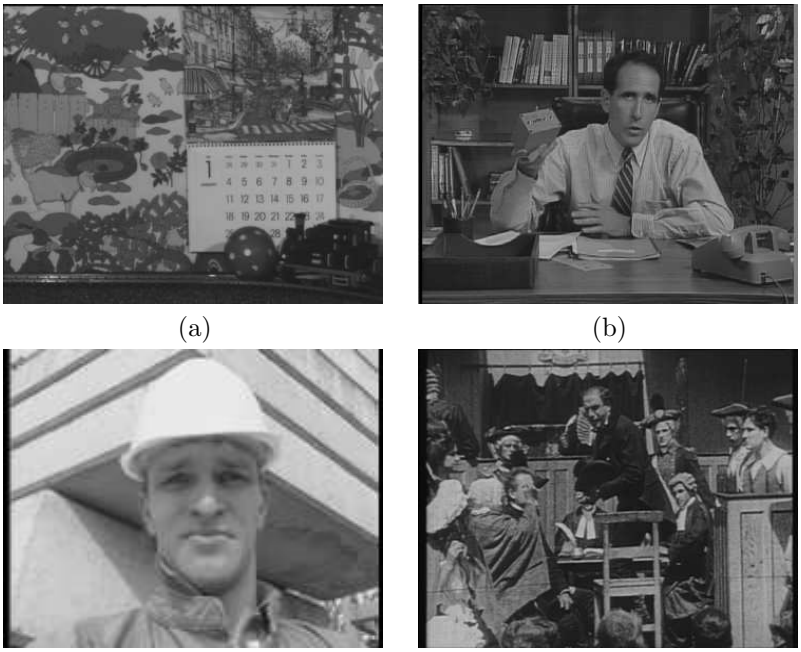

Figure 5. Frames from the four test sequences used. (a) calendar/mobile, (b) salesman, (c) foreman, (d) rory.

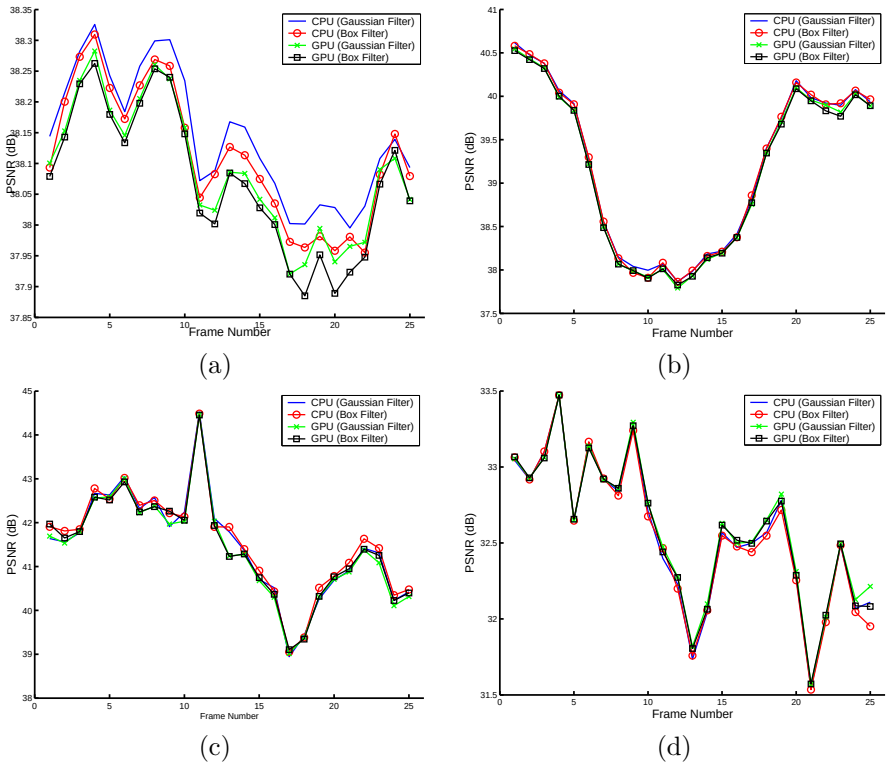

Figure 6. WBME PSNR results for the four test sequences. (a) calendar/mobile, (b) salesman, (c) foreman, (d) rory.