General Purpose Computing with a Graphics Processing Unit

A dissertation submitted in partial fulfilment of the requirements for the DIT's Master of Science Degree in Applied Computing for Technologists.

Jason Ruane BSc.

Dublin Institute of Technology

Supervisor: Gerard Heapes

Computing, Faculty of Engineering, Bolton Street

October 2006

Volume 1

Declaration:

I certify that this dissertation which I now submit for examination for the award of Master of Science Degree in Applied Computing for Technologists, is entirely my own work and has not been taken from the work of others save and to the extent that such work has been cited and acknowledged within the text of my work.

This dissertation has not been submitted in whole or in part for an award in any other Institute or University.

The Institute has permission to keep, to lend or to copy this dissertation in whole or in part, on condition that any such use of the material of the dissertation be duly acknowledged

Signature _____ Date _____

Candidate

Table of Contents:

List of Figures

Abstract

Graphics hardware in mainstream PCs has experienced rapid growth in performance and capabilities recently. The emergence of high level programming languages, such as Cg, for this commodity hardware has brought a tool of immense computational power to the mainstream programmer. Examples of how researchers have harnessed this resource to perform general purpose computing are examined. The hardware of modern Graphics Processing Units (GPUs) and their parallel nature is explored. The major facets of how to enact General Purpose computing on a GPU (GPGPU) are described, and two implementations to demonstrate GPU programming for general purposes are presented.

The first implementation is a program to sort a list of random items and the second implementation is to search for motion vectors in successive video frames, which is a central component for video encoding and conversion. The performance of the programs running on the GPU is compared against that of similar programs running on CPU and it is shown that the GPU performs favourably, outperforming the CPU by a factor of 6 for Motion Estimation. The performance of the sorting implementation presented here is also compared with similar published results.

In closing, a commentary of the results found is presented, the possibilities for the future of GPGPU are considered and current trends in this area are discussed.

1. Introduction to GPGPU.

The GPGPU acronym has become associated with the field of General Purpose computing with a Graphics Processing Unit. This field of study relates to the programming of Graphics Processing Units (GPU) for uses beyond the original and traditional sphere of image generation for screen projection. GPUs are the central processing units of graphics cards (or video cards) found mainly in modern computers. Originally GPUs were designed to supplement the CPU of a computer for the task of creating a frame buffer which would be projected onto the monitor or VDU. The arrival of a new wave of programmable GPUs, however, has facilitated the usage of these units for purposes other than just on-screen projection, but rather to perform generic computations hitherto confined to the realm of CPUs.

The programming of a GPU for a general, non-rendering task is fundamentally different to creating a program for a familiar CPU. While a CPU of a modern PC is based on the Von Neumann architecture, the GPU is a stream processor, taking a set of inputs, known as a stream, operating a function, known as a kernel, upon that set of inputs and then producing a set of outputs to a buffer and possibly to screen. Parallelism is employed within the GPU so the algorithm, or kernel, must be formed such that it can operate independently without reference to neighbouring streams. This parallelism frames the structure of GPU computing and underpins the activity of porting an algorithm to a GPU.

1.1 GPU versus CPU:

Heuston (2006) comments on the dichotomy between GPUs and CPUs, noting how CPUs are more suited to "task parallel" situations, where there are independent processes running with little communication and adding another process to satisfy a new task is easily implemented (www[HEU]). For a GPU however, there is much data (elements of a graphic) for which the same operation is being computed with no dependencies between the data elements at that time thus being a more "data parallel" situation, suited to the pipelined nature of GPU hardware.

Many applications have been successfully demonstrated on the GPU out-performing the CPU. The PennySort competition is designed to test a computing platform's performance at the task of sorting database records, per averaged cent of

hardware cost. Gray (2006) demonstrates the importance of the arrival of the GPU as a general purpose coprocessor when commenting on the PennySort results:

"This year saw a breakthrough with GpuTeraSort which uses the GPU interface to drive the memory more efficiently (and uses the 10x more memory bandwidth inside the GPU). GpuTeraSort gave a 3x records/second/cpu improvement". (www[GRA])

Figure 1 below, depicts the number of multiplies which were achieved by a series of products from ATI, nVidia and Intel demonstrating the increasing gap between the capabilities of GPUs and CPUs.

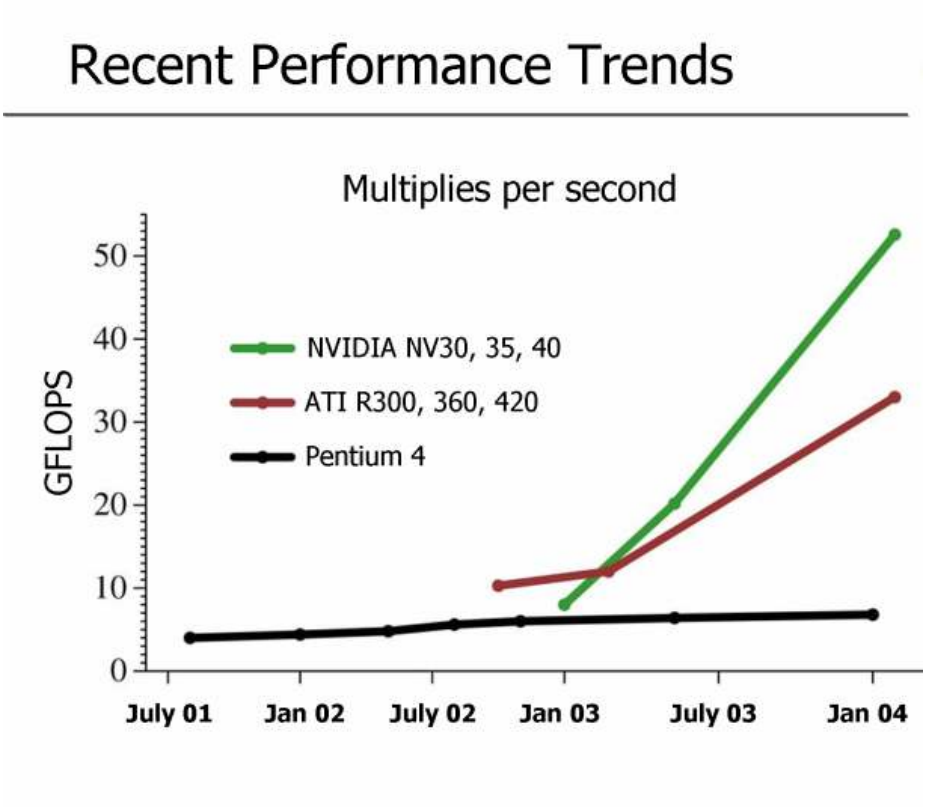


Figure 1. Source: www[HAN]

A more recent comparison is presented by Green (2006) who compares the observed performance of the latest GPUs against even the theoretical maximums of CPU performance and demonstrates a continuing trend of divergence (www[GRE]), as seen in figure 2 below:

GFLOP performance

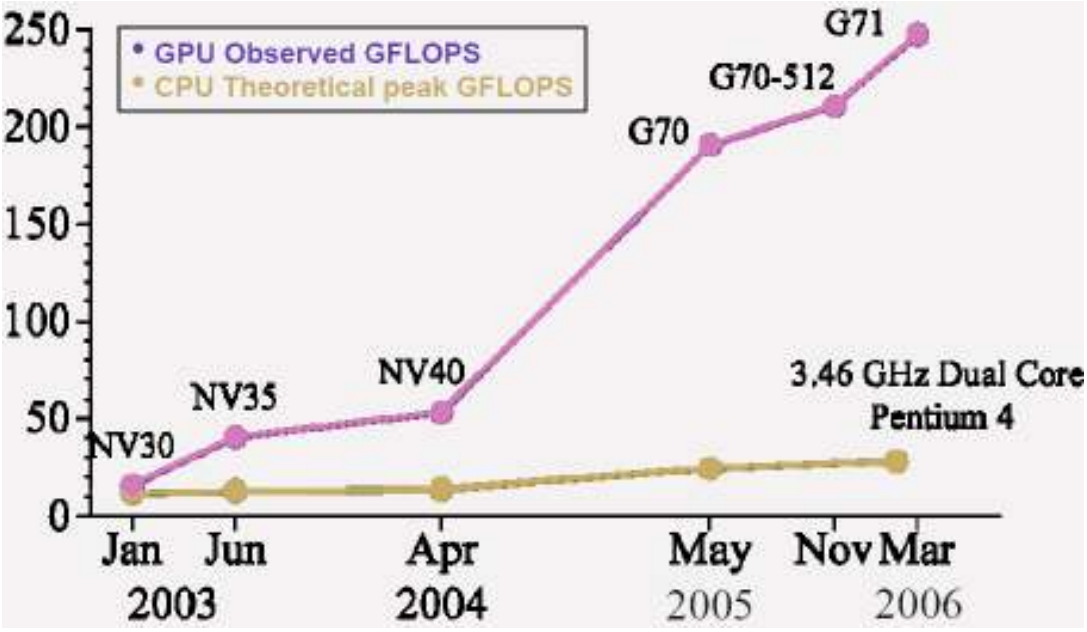


Figure 2. Source: [GRE]www.

1.2 Non-von Neumann

Backus (1978) posits that the Von Neumann style processor (which encompasses modern PC CPUs), while suiting general purpose functional demands, suffers from needing to dedicate much of its silicon to the overheads associated with local memory management, an idea reiterated by Venkatasubramanian (2003). While the GPU is more of a stream processor it does still suffer from standard CPU limiters such as memory bandwidth, as described by Hanrahan (2004) where dense matrix-matrix multiplication has been shown to be bandwidth limited on the GPU of the day. Such limiters are being addressed by the advances in hardware with ever-increasing speeds, but since all of this programmable GPU technology is relatively new, Hanrahan (2004) suggests this may offer an "incredible opportunity for reinventing parallel computing software" and "architectural innovation". This notion inspires the idea that the GPGPU applications of the future may diverge from the von Neumann style programming constructs (such as the porting of usual data parallel CPU algorithms) and foster their own unique brand of stream processing (such as the inception of new GPU specific algorithms). This idea was expanded and reinforced by Owens et al. (2005) who commented:

"...new work must go beyond simply `porting` an existing algorithm to the GPU, to demonstrating general principles and techniques or making significantly new and non-obvious use of the hardware."

1.3 Physics Processing on the Desktop

While the CPU has dominated the PC computational strength to date, its monopoly on the computation domain is being eroded in modern high-end PCs (which one can presume will become the commodity PCs of the future). The GPU is well established as a powerful coprocessor, primarily used for graphics to date, but the emergence of non-graphics activities on the GPU expands this role.

In parallel, the PPU (Physics Processing Unit) is making a tentative introduction to the desktop PC market. Ageia is one hardware vendor which is making physics cards (similar to graphics cards), such as the PhysX product, where these cards operate alongside the GPU in another PCI slot and are dedicated to calculating the intensive physics of games and scientific applications. The gaming market could drive the proliferation of PPU cards in mainstream PCs, as it has already done with GPU cards, thus creating a new market segment. The following is a quote from an interview with Manju Hegde, CEO of Ageia in 2005, published at gdhardware.com (www[HED]):

"There is a possibility that the GPU can supply limited snippets of "physics" effects. The GPU community has been trying to encourage the use of this functionality for years with limited success. As we've seen with 3D, there is nothing like dedicated hardware to allow interactive physics to be fully exploited in tomorrow's games."

Installing a PPU card is not the only way to leverage large scale physics computations for 3D real time games though; the dual-core and multi-core CPUs emerging from the mainstream manufacturers are promoted as being the platform for such computations, encouraging the market to return to the CPU, as evidenced by its appearance in marketing media, such as an AMD press release of 2006 (www[AMD]) and Tom's Hardware Guide (www[THG]). Physics processing is computationally demanding so require multiple, well considered threads on at least one core for a CPU, as discussed by Dawson and Walbourn (www[DAW]).

The third alternative location for physics processing is on the GPU. Havok is a company which originated in Dublin, Ireland who create the Havok Game Dynamics SDK (Havok) and suite of middleware physics engines which are designed to run primarily on the GPU, not just of PCs but also gaming consoles such as Microsoft Xbox and Sony Playstation 2. Havok (2006) describe the reasoning for running their software on the GPU (www[HAV]):

GPUs also have a clear advantage as a pre-existing technology familiar and readily available to consumers and game developers, providing other benefits such as wide-spread availability, commodity pricing, and mature standards for hardware and software interfaces.

The Havok suite of developer tools and runtime software have gained much market share and acceptance, having been proliferated into over 150 products on all the main gaming platforms (www[HAV2]). As of 2006 it appears prominently in most of the current best selling titles. Green (2006) describes using a 3 GPU setup with two of the GPUS dedicated to graphics generation and one dedicated to physics calculations. Apart from physics engines, they also provide middleware for other aspects, such as event-driven character behaviour (as in the "Havok Behaviour" product). This could be viewed as one scenario where non-graphics computations, performed on the GPU has already proven itself successful, useful and commercially viable.

1.4 GPGPU Samples

In 1999, Kedem and Ishihara showed how a graphics system with its inherently SIMD nature could be used to ensure a decryption of any Unix password (with a 56-bit cipher) in two days via brute force methods. Trendall and Stewart (www[TRE]) were early pioneers of the modern GPGPU arena, calculating refractive caustics in a general purpose calculation on graphics system in 2000. Also within the sphere of graphics, was the work of Proudfoot (2001) who demonstrated procedural shading on a GPU. In 2001 also, Rumpf and Strzodka used GPGPU methods to solve the linear heat equation and anisotropic diffusion. Kim and Lin's simulation of ice crystal formation in 2003 was performed both on CPU and GPU, for which they observed between 2.5 and 9.4 times speedup with a GPU implementation versus that of a CPU.

In the 2004 book "GPU Gems", Harris describes a GPU implementation of the Navier-Stokes equation for stable fluid simulation with a six times speedup compared to the CPU equivalent. This publication and its successor, "GPU Gems 2" of 2005 contain a compendium of actual GPU implementations from a number of leading figures in the GPGPU field. Other examples from these publications include protein structure prediction by Micikevicius (2005) and financial options pricing by Kolb and Pharr (2005).

1.5 GPGPU Commercial Success

The Havoc suite of products described above is one example of commercial success but other, even more archetypal examples of non-graphics computation on the GPU are also available. For example the GPUTeraSort project by Govindaraju et al. (2006) has experienced much success, having won the PennySort title for 2006 (www[GRA]) (an open competition where the task is to sort as much input as possible for the averaged hardware cost of one penny) by sorting at the rate of 60GB per penny's worth of hardware (Govindaraju et al., 2006).

Commercial software is also appearing which makes increasing use of the GPU in a non-real time graphics sense, such as the Avivo products from ATI which can transcode video using the GPU as a coprocessor (www[AVI]).

1.6 Wheel of Reincarnation

The phrase "Wheel of Reincarnation" coined by Myer and Sutherland (1968) was used to describe a phenomenon where display functions which the CPU could not satisfy due to performance, were offloaded to dedicated processing units elsewhere in the system, and as the tasks became more complex, ever more sub-processors were added in to offload further sub-functions. However, due to CPU evolution being so great, these functions eventually became subsumed again within the CPU. Some commentators, for example Crow (2004), have surmised that this reversion to the CPU will not happen again in the case of graphics processing because the mass-production of GPUs ensures its cost effectiveness and the rate of performance increase is greater in the GPU arena than that of the CPU. The

following is from the conclusions of Myer and Sutherland, in 1968, before GPUs became mass-produced entities:

General computing power, whatever its purpose, should come from the central resources of the system. If these resources should prove inadequate, then it is the system, not the display that needs more computing power.

The emergence of GPGPU activities demonstrates quite how far the GPU has progressed, into the stage of actually outperforming the central resources (read: CPU) which it serves; not just in the task of display handling, but also in certain GPGPU experiments.

2. Graphics Cards

2.1 Graphics Card Hardware

The Graphics Processing Unit, as commonly found in modern PCs and games consoles, refers to the central calculating engine of the graphics card. The graphics card, in turn is the self contained array of electronics hardware which encloses and supports the functions of the GPU. This card is often a physical electronics board which is attached to the PC motherboard via a PCI Express (Peripheral Component Interconnect) or AGP (Accelerated Graphics Port) slot.

The graphics subsystems of today's PCs are either the dedicated device mentioned, or an integrated graphics setup whereby less powerful graphics processors are embedded into the motherboard and share the resources of the host PC, such as RAM. The dedicated graphics cards are designed primarily to handle the substantial computational complexity of 3D computer graphics. The integrated graphics solution lags behind the dedicated card with respect to performance and capabilities, but is most often capable of satisfying the 2D demands placed on it by traditional home and business users of PCs. The dedicated systems, while employing GPUs of higher specifications also benefit from higher access speeds to their dedicated on-board RAM, such as the current 64Gb/s of the ATI Radeon X1950 XTX card (www[TOR]), whereas integrated graphics systems rely on sharing the system RAM with the CPU at speeds of 8Gb/s currently. The graphics choice for this dissertation focuses only on dedicated systems with commercially available graphics cards, for only these currently provide the required capabilities such as Shader 3.0 support with Frame Buffer Objects.

Graphics cards (and in particular the fragment processors within them, described below) are more closely associated with SIMD (Single Instruction, Multiple Data) architecture than their CPU counterparts which are more aligned with MIMD (Multiple Instruction, Multiple Data). The SIMD paradigm is characterised by its ability to load a number of inputs simultaneously and to perform a fixed sequence of operations (kernels) upon this input. MIMD processors, such as CPUs in modern PCs are designed to operate in a more varied, general purpose way, thus they tend to load fewer inputs at a time and perform more varying operations on this set of inputs. For a SIMD device, the advantages include less instruction codes and a

greater opportunity for parallelism. Hanrahan (2004) compares the SIMD architecture of the GPU to stream processing, in which kernels are units of operations which process a set of inputs arriving in a stream and generate a similar stream of outputs, onwards to the next kernel for processing.

2.2 Rate of growth

The rate of growth in GPU computing power has surpassed even the CPU market, with GPU transistor counts increasing at a rate of Moore's Law cubed (www[HAN2]). GPUs are doubling their transistor count approximately every 6 months compared to the CPU equivalent of 18 months (Crow, 2004). The number of transistors is a crude comparison though, as it does not consider the nature of their designs, GPUs being more typical of SIMD devices and CPUs more aligned with MIMD devices.

"Their `Moore's Law` is faster than that for CPUs, owing primarily to their stream architecture which enables all additional transistors to be devoted to increasing computational power directly."

(Venkatasubramanian, 2003).

The following graph in figure 3 demonstrates the rate of increase for processing power and bandwidth for a selection of nVidia GPUs through 2003. The relative faster rate of processing power growth can be observed alongside the slower growth of off-chip bandwidth. This trend encourages higher compute intensity algorithms on the GPU hardware.

Processing Power and Bandwidth of GPUs.

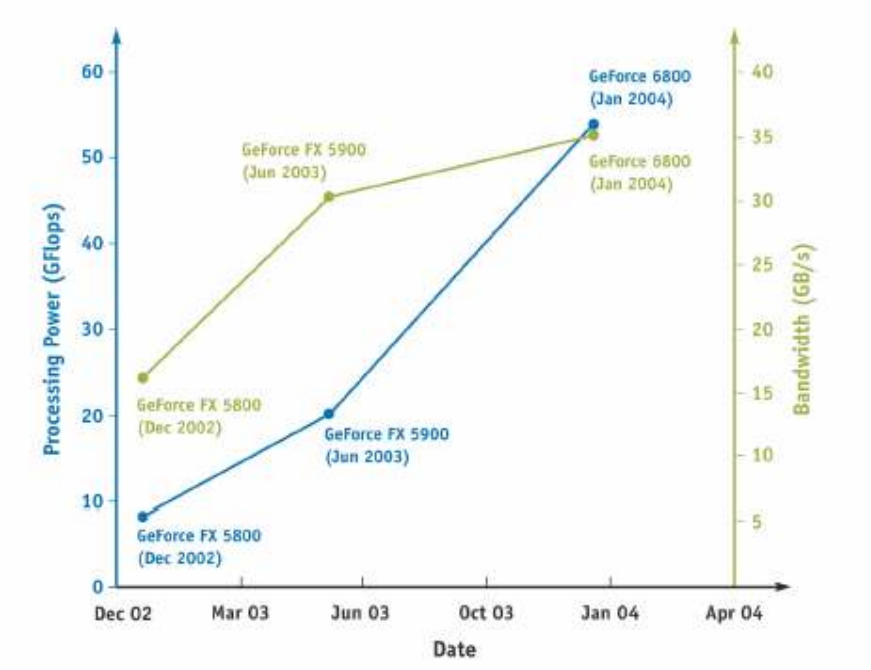


Figure 3. Source: Owens, 2005.

As GPUs are becoming more powerful, they have relieved the CPU of much of the processing work associated with 3D visualisation, such as that of 3D games. The processing powers of both are difficult to compare, as they do not align directly, but with respect to FLOP (Floating Logic OPerations) throughput per second, sample results of today's PC market from Heuston (2006) show a 3.0GHz Intel Pentium 4 can generate approximately 12GFLOPs peak, while an ATI Raden X1800XT can process 120GFLOPS peak (www[HEU]).

The GPU which has been a veritable co-processor for the CPU previously, now has the ability to compute more floating point arithmetic than the actual CPU. Taking advantage of this processing bounty contained within the GPU is the aim of the GPGPU community. One reason why this counter-intuitive performance difference has not been more publicised or impactful upon the PC market is that the GPU does not suit all types of programs/problems given its stream nature and hardware restrictions.

2.2.1 Pipelining

The graphics cards of today are deeply pipelined, meaning that they order their computations in a series of stages, each of which can operate simultaneously, akin to a production assembly line. Geometric primitives delivered as input from the CPU and are progressively converted and operated upon in each stage. Since each of the stages operates independently they are candidates for parallelism. Some cards are highly parallel, such as the nVidia GeForce 7950 GT which has 24 fragment processors (www[TOR2]).

The general pipelined assemblage of components in most graphics cards today is described in the below diagram which illustrates the primary functions or areas. The initial stage is Vertex Transformation, wherein the inputs of vertices arrives from the CPU and are translated into the various coordinate systems such as world-space and screen-space which are required downstream. The output of the first stage is fed into the next stage, freeing the Vertex Transformation stage to begin processing the next set of inputs. The next stage is the Primitive Assembly and Rasterisation in which the processed vertices are assembled into geometric primitives and rasterisation occurs, which is how the GPU generates a fragment for each potential pixel update which may occur later. The fragments are then passed to the next stage in which they are textured by applying predefined patterns and colouring which calculates the various components of colour for each fragment. The fragments then pass onto the next stage where they may update a particular pixel location in the framebuffer.

Pipeline Structure of a GPU

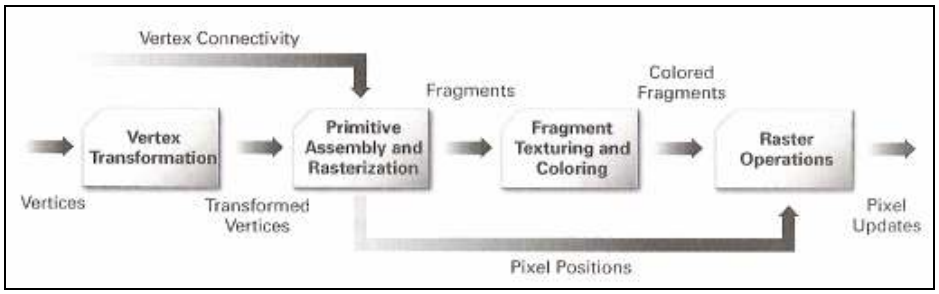


Figure 4. Source: Fernando and Kilgard, 2003.

2.2.2 Parallelism

The task of graphics generation is highly suited to being performed in parallel for many stages. The pipelined structure of the hardware compounds this, and a review of the specifications of current GPU products supports this point. For example the ATI Radeon X1900 XTX has 48 pixel pipelines (www[TOR]) and the nVidia GeForce 7950 GT has 24 (www[TOR2]). GPU parallelism appears at another level also, as described by Govindaraju et al (2006):

In addition to the SIMD and vector processing capabilities, each fragment processor can also exploit instruction-level parallelism, evaluating multiple instructions simultaneously using different ALUs.

Parallelism is being observed at a macro scale currently; as entire GPU cards are being installed alongside each other within the same PC in order to share the graphics workload. The two main technologies supporting this in consumer PCs are SLI (Scalable Link Interface) from nVidia (www[NVI]) and CrossFire from ATI (www[ATI]).

The leverage of parallel processing units for increased throughput is an idea currently being embraced by the CPU community. The current releases of high-end CPU models are already dual-core (www[INT]) and are on the cusp of quad-core (www[THG2]) products.

2.3 Programmable Graphics

With the hardware available, the task of directing the graphics systems in a systematic and controlled fashion is the domain of software. For graphics systems, this can be summarised by the following diagram, figure 5, which illustrates the hierarchical structure of most GPU software infrastructures found in modern PCs.

GPU Software Infrastructure

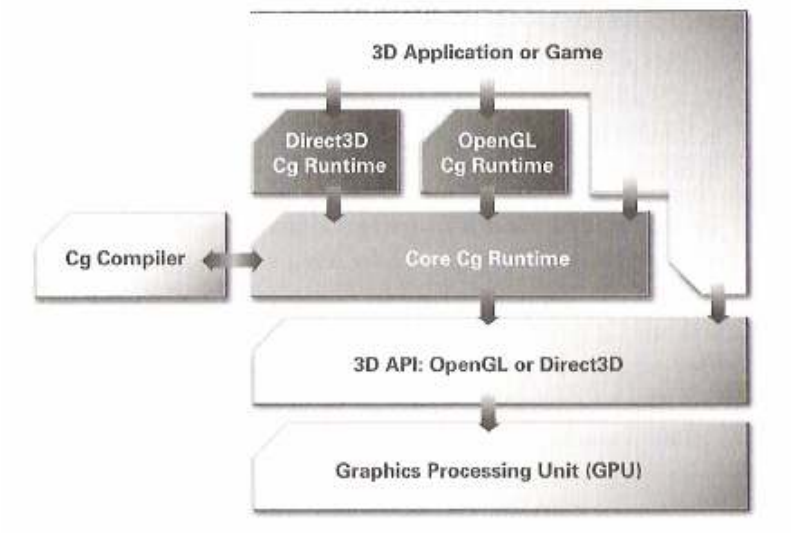


Figure 5. Source: Fernando and Kilgard, 2003.

In the above diagram, the Cg elements may be interchanged with other shader languages, with some caveats, and usually only one of either OpenGl or Direct3D are implemented in a given application. The programming of the graphics card though is not simply another revision of "von Neumann languages" (a phrase from Backus, 1978 at his Turing award speech describing CPU languages such as FORTRAN and C) but hints instead at the state machine underneath. While the GPU resembles the von Neumann-style CPU in many respects, such as its reliance on memory addressing for textures etc, its state machine descendancy is evidenced in its programming such as the OpenGL calls which set up operational modes, rather than execute a function and return a value.

2.4 Shader Languages

The above description and diagram of the hardware pipeline in figure 5 conveys a high level view of the operations within the graphics system, but the essence of this study focuses on a particular evolution within this assembly. The evolution of note was the introduction of programmable vertex and fragment processors within the GPU. This marked an epoch for computer graphics as it gave designers and code developers the ability to create custom, configurable effects, known as shaders, using assembly language. The first commercial card supporting

fragment (or pixel) shading was the nVidia GeForce 3 released in 2001 followed by the ATI Radeon 9700 of 2002, which permitted looping constructs within the shader.

CPU-GPU Overview

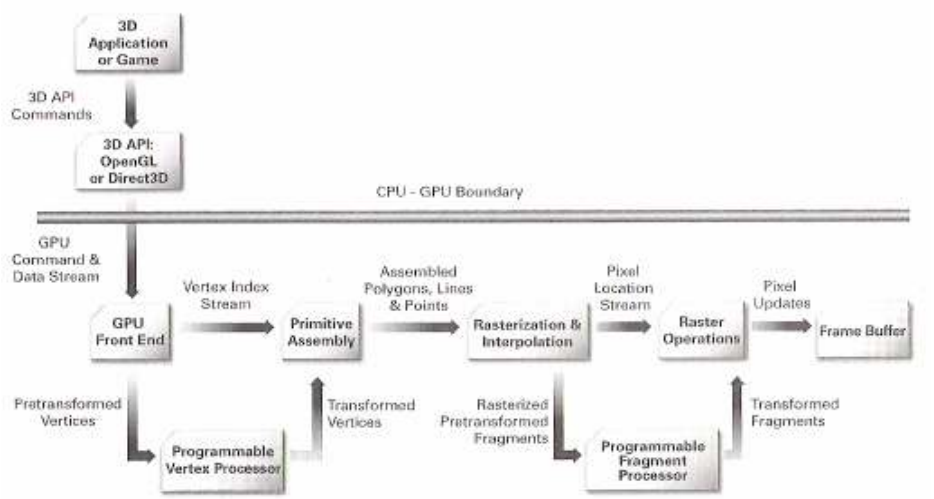


Figure 6. Source: Fernando and Kilgard, 2003.

The programmable vertex and fragment processors are the salient points in the above diagram, with respect to this dissertation, for it is directly through the programming of these which allows the programmer to take advantage of the computing power within, and from which the field of GPGPU has arisen. Originally designed to allow graphic designers more freedom for their creation of effects in real time gaming, they have become the focal point for GPGPU activities. The programmable vertex and pixel processors are generally considered as state machines, where one initiates a program running and from then on it processes all input through the program.

Recently, the programming of these elements has been abstracted to a higher level. Rather than labour with arcane and esoteric assembly language, there now exists a range of high level languages which can be used, in various configurations, to program the vertex and fragment processors. This migration to a higher level language is central to the enabling of GPGPU applications. It has allowed programmers to harness the programmable vertex and fragment/pixel processors without needing to learn the particular assembly language, or be overly burdened with the specifics of the hardware layer. The learning curve for the prospective GPU programmer is lowered by the similarities between these languages and other more

familiar languages such as C. A cataloguing of the major versions is given in the following list, from Du Toit and McCool (2004):

- OpenGL Shading Language (GLSL)
- Stanford Real-Time Shading Language (RTSL) (Not currently active)
- Microsoft High Level Shading Language (HLSL) (DirectX specific)
- nVidia Cg (API neutral)

This dissertation is primarily focussed on the Cg language as a means of programming the GPU as it was found to be freely available, well documented and API neutral. Cg is a language somewhat similar to C, but a much smaller subset. It compiles either at runtime, to accommodate the permutations of hardware in the consumer's PC, or statically at development stage. The language was developed primarily at nVidia and was presented in a paper from Mark et al. (2003) at the SIGGRAPH 2003 convention. The following quote from the introduction foretells of its later adoption by the GPGPU community as an ideal platform for general purpose computations:

The Cg language is based on both the syntax and the philosophy of C [Kernighan and Ritchie 1988]. In particular, Cg is intended to be general-purpose (as much as is possible on graphics hardware), rather than application specific, and is a hardware-oriented language.

However, the shader language is merely one component of the steps required to conduct the orchestra of computation occurring on the programmable GPU. The other major component is the graphics API, usually one of the popular duopoly: OpenGL and Direct3D.

2.5 Graphics Programming Interfaces

To write applications for computer graphics, today's programmers rely mainly on OpenGL (Open Graphics Library) or Direct3D each of which is a specifications standard defining a general computer graphics API (Application Programming Interface). From this standard, graphics hardware manufacturers can create libraries to effect the functions declared in the standard, and programmers can write their programs to call the functions relying on a standard implementation

occurring. The API is thus an interface to the hardware of the graphics infrastructure. These standards have progressed with the demands of programmers and advances in hardware offered by the manufacturers. This allows compatibility between software and hardware to be captured succinctly, such as a 3D game which declares OpenGL 2.0 as part of its minimum requirements, and a graphics card manufacturer which produces an item declaring OpenGL 2.0 support.

2.5.1 OpenGL

Widely used, OpenGL supports multiple platforms, languages and windowing systems. The specifications standard was developed initially by SGI (Silicon Graphics Incorporated) who also founded the OpenGL ARB (Architecture Review Board) in 1992 (Ruge, 2001). The OpenGL ARB comprises of a group of companies which have directed the development of this standard to date, announcing at SIGGRAPH 2006 that it would transfer ownership to the Khronos Group, an open membership consortium, by the end of 2006 (www[RIE]).

While OpenGL is a specification which declares the functions which must be made available to the programmer, it does not dictate precisely how that implementation should occur on the graphics hardware.

Individual calls can be executed on dedicated hardware, run as software routines on the standard system CPU, or implemented as a combination of both dedicated hardware and software routines. (www[OPE])

This flexibility permits the hardware manufacturers to differentiate themselves, by providing hardware acceleration (via the GPU etc.) or reverting to a software implementation; meanwhile the programmers can rely on a consistent result. The standard is also extensible, in that manufacturers may provide their own extended functionality for the programmer, and that extension may be considered by the governing body (e.g. OpenGL ARB) after which it may become part of the ratified API. This flexibility allows advances made in hardware to quickly propagate to developers and consumers.

2.5.2 Direct3D

Microsoft operating systems since Windows '95 support OpenGL, (Microsoft was a founding member of the ARB (Fernando and Kilgard, 2003) but Microsoft also developed in parallel, a 3D graphics API of their own, as part of the DirectX project available for Windows '95 and later (www[ISL]). An initial impetus for the project was the aquiring of the company RenderMorphics in 1995 and their 3D API of that time "Reality Lab" (www[LIN]). The DirectX initiative encompasses other media components such as sound (DirectSound), user interaction (DirectInput) and communications (DirectPlay) in addition to graphics management. This API is not cross-platform, as it is compatible only with Microsoft OS on PC and the Xbox gaming console. Direct3D exposes the hardware features of the graphics system to the programmer, similar to OpenGL, and currently also supports shader languages such as Cg and HLSL (Microsoft's own shader language, similar to Cg).

2.6 Dynamic Compilation

Cg, as a shader language is designed to instruct the programmable vertex and fragment processors how to operate but the Cg source code must be compiled into assembly language first. This can be accomplished in either of two ways, the first being via static compilation whereby the programmer compiles the code before publishing it. The other method is dynamic compilation whereby the Cg code is delivered to the graphics API (OpenGL or Direct3D) via the Cg Runtime, an intermediary compiler which compiles the Cg as necessary for the GPU when it is required by the application.

This dynamic element permits the graphics system to compile the Cg code in a method which matches the capabilities of the GPU. Thus the programmer can write various Cg programs to match various consumer hardware permutations and the resultant program can maximise portability. The various configurations of graphics hardware, API and shader languages are managed by means of profiles. A profile depicts the compile and run capabilities of a given computer, and it is against the active profile, which the Cg will be compiled. For example a sample Cg fragment profile is "ps_2_0" which equates to a DirectX 9 compatible setup (Fernando and Kilgard, 2003).

2.7 GPU Computing Framework

To perform computing on the GPU, a number of techniques and idiosyncrasies of the hardware and APIs must be considered. Firstly, in what part of the GPU hardware these computations will be performed. The fragment processor of today's current hardware is the suitable choice for most scenarios because the output of the fragment shader goes to the framebuffer, and can be captured back to the CPU thus facilitating results feedback. Also, graphics cards usually utilise a number of fragment shader in parallel, such as the 24 in the nVidia 7950GT (www[TOR2]). One drawback of the fragment processor is that since it is predestined to write to a particular pixel location, scatter writing to a random memory address is not permitted currently, so a gather operation may be required afterwards to collate the output.

The fragment (and recently vertex) processors have the ability to reference the texture unit in order to retrieve texture information. This facility can be used to bring input data to the kernel, by loading the data as one or more textures. For example in the implementation section described below, the input sets of random data which is to be sorted is stored inside a 2D texture with one input per pixel, similar to a 2D memory array within a CPU context.

The computation itself is implemented as a kernel on the fragment processor. For the implementation below, this is completed by a Cg program which is loaded onto the fragment processor via the Cg runtime compiler (which generates assembly bytecode for the processor) and the OpenGL API which conducts the proceedings. When the computation has been performed, the output is sent to the framebuffer, or in the GPGPU case, it may be redirected to a texture, in a technique known as "render-to-texture". This texture may later become the input to another phase, or pass, of the program.

To actually initiate any computation though, data must be sent through the graphics pipeline, which translates into effectively attempting to draw a geometric figure. In the implementations of this dissertation, and most GPGPU applications, this geometry is a simple quadrilateral which as it passes through the graphics pipeline will become a 2D array of fragment operations occurring effectively in parallel.

Given the nature of computing on the GPU, certain classes of problems emerge as being more suited to the GPU than others. For example, compute intensive

problems are suited since texture fetches incur a delay and GPUs do not have the same magnitude of on-die memory cache as CPUs. Buck, (2005) describes this effect and why it leads to the scenario where higher compute intensity is preferred:

Therefore, if we want to be limited by the computing performance of the GPU and not by the memory, our programs need to contain enough arithmetic instructions to cover the latency of any texture fetches we perform.

2.8 GPU Computing Tools

For programming a modern CPU, source code debugging can be aided by a number of tools. Such tools include Microsoft Visual Studio Debugger for Visual Studio .NET or GNU Debugger for Unix and Windows based GNU (GNU's Not Unix) development (www[GNU]). Programming for a GPU however requires a quite different suite of tools, as they must analyse the activities of the GPU and its programs or performance. The state of development in this area is not as advanced as CPU debuggers however there is a selection of products available to serve this niche. The existence of these debuggers owe mainly to the efforts involved in graphics optimisation and the requirements of graphics designers/programmers rather than those of the GPGPU community.

To debug OpenGL code on both ATI and nVidia GPUs, a company named Graphic Remedy produce a product called "gDEBugger" which allows the user to perform many of the traditional debugging operations such as permitting breakpoints but also allowing an analysis of the pipeline's performance (www[GDE]). The "perfHud" product, as described by Kiel and Dietrich (2006) from nVidia is another such tool, but is restricted to nVidia GPUs currently. For basic debugging of the Cg shading language, the Cg compiler can return error codes via the OpenGL or DirectX APIs to indicate any syntactical errors or a situation where the hardware limits have been breached by the compilation process. For example, if after unrolling a loop the compiler finds it has more sequential instructions to perform than the shader allows. One utility concerned with GPGPU programming is "GPUBench". This product, presented by Buck, Fatahalian and Hanrahan (2004) is designed with profiling shader capabilities pertinent to GPGPU applications and is open source managed (www[BEN]).

3. Implementation

The implementations chosen for this dissertation were sorting and the calculation of motion vectors for motion estimation as part of video compression. The former, a classic problem in the sphere of computing, has been implemented innumerable times on all software platforms and demonstrates a primary capability of a computing system. The later was chosen for its high degree of computation required, and its applicability to the emerging importance of video compression in the commodity PC market. Sorting on a GPU has been shown by Buck and Purcell (2004) to be a viable proposition and a worthwhile endeavour, so revisiting this task on today's commodity graphics cards was inspiration for this undertaking. The latter choice for calculating motion vectors between successive video frames was chosen as it permitted the opportunity for innovative exploration, aligning with the previous work of Green (2005) and Fang et. al. (2004) which showed successful DCT (Discrete Cosine Transform, another major component of video transcoding) and IDCT (Inverse DCT, its reverse action) calculation on the GPU.

3.1 The Programming Environment

The programming framework used for both implementations herein comprises of a .cpp file containing all of the C/C++ code for each task (sorting and video), which utilises OpenGL with the GLUT system, and Cg programs (each a text file) loaded to the vertex and fragment processors. The C and C++ code has been written linearly in a C fashion, rather than in a strictly object oriented fashion, save for the usage of C++ only elements (such as std::sort). The code was compiled with Microsoft Visual Studio 6, producing a single binary for each task (such as GPUbitonicSort.exe), which relies on OpenGL libraries and Cg runtime to instruct the GPU how to load the textures of input data, run the Cg programs and report back the sorted output data.

3.2 GLUT

The OpenGL Utility Toolkit (GLUT) was originally written by Mark Kilgard in 1994 (www[GLU]), but the Win32 version used here was a port by Nate Robins

(www[ROB]). It is a popular toolkit which allows ease of introduction into the OpenGL programming arena; allowing for simpler creation of basic windowing and a higher level of abstraction for many OpenGL and operating System tasks. The OpenGL Extension Wrangler Library (GLEW) is a similar set of libraries which is often implemented alongside GLUT, but for portability reasons GLEW was not employed in this implementation.

3.3 Source Code

The commenting of the code was written in accordance with the JavaDoc specification of Sun Microsystems (www[SDN]) and along the guidelines specified by the official Sun Java document (www[SUN]).

The packaging of the resultant binary, Cg files and required libraries resulted in both demonstrations being able to run on a PC with base installation of Windows XP, a sufficient graphics card and its relevant drivers. This enveloped the programs into portable units of minimal size requiring no local machine installation. The actual listings of the source code are given in the appendices 7.3 and 7.4 sections.

3.4 Sorting

CPU Sorting has been demonstrated previously in numerous well-documented algorithmic forms, thus providing a basis for comparative performance analysis. To this extent, the standard sorting routines provided in the C and C++ languages were taken as exemplary baselines, against which the GPU implementation, undertaken here, could be compared and contrasted with. The nature of the GPU, which is a SIMD device, means that it operates independently on each element. This restricts the number of algorithms which are easily employed on a GPU. Classical approaches to sorting such as “quick sort” and “heap sort” are not suitable for GPU implementation as random write addressing is not yet available, but other approaches are entirely suitable for the hardware nature of the GPU. Such examples of suitable algorithms are binary sort, odd-even transposition merge sort, and bitonic sort as introduced by Batcher (1968). The bitonic sort algorithm was chosen for this dissertation as it exhibits $O(\log^2 n)$ performance (www[BLA]), surpassing that of both binary and transposition sort (Venkatasubramanian, 2003). It

was also the algorithm of choice for the 2004 implementation by Buck and Purcell (2004). Greß and Zachmann (2006) later improved upon the bitonic sort performance with an adapted version which they named "GPU-ABiSort", however this more elaborate version was not undertaken in this implementation in order to focus more on the GPU computations than the intricacies of the algorithms themselves.

3.4.1 Bitonic Sort Algorithm

The algorithm of Bitonic sorting is suited to the parallel nature of GPU hardware as it permits up to n processors to simultaneously sort n elements over $O(\log^2 n)$ steps (Buck and Purcell, 2004). For this algorithm, the merging function can be performed in parallel on the fragment processors. The following diagram from Buck and Purcell (2004) shows the algorithm as applied to an input set of 8 numbers:

Bitonic Sorting

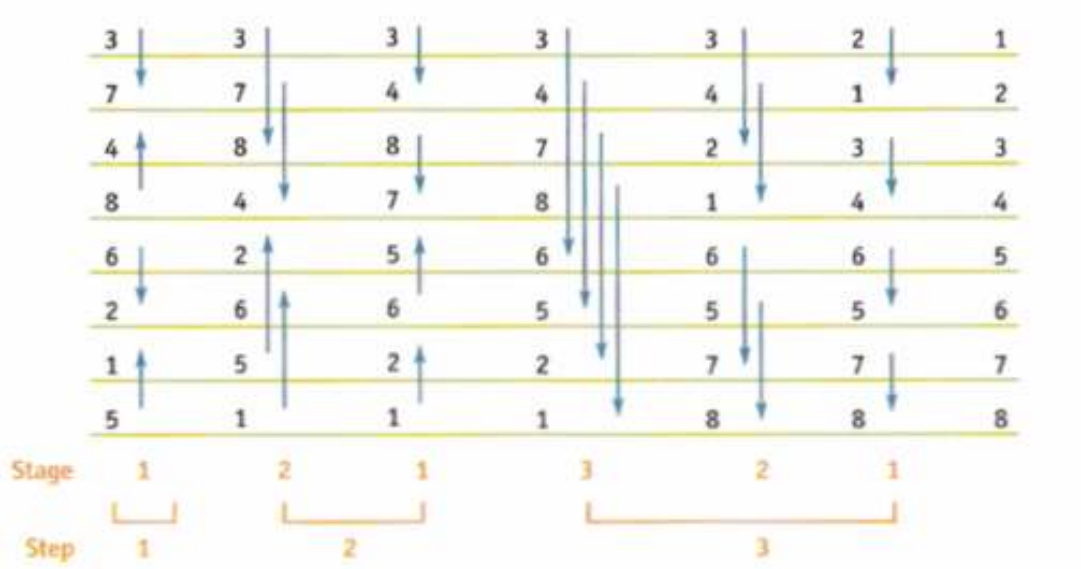


Figure 7. Source: Buck and Purcell, 2004.

The bitonic sort implementation of Govindaraju et. al. (2005) reported performance of almost six times that of Quicksort on a CPU. The following diagram from that paper compares their observed sorting throughputs of GPU and CPU examples:

Sorting Performance Comparison

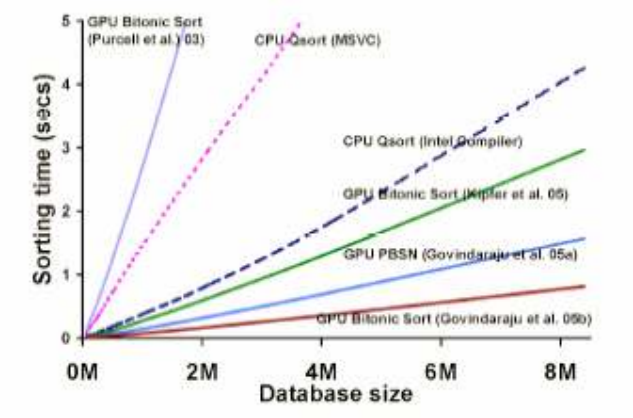


Figure 8. Source: Govindaraju et al., 2005

3.4.2 Application on a GPU

To sort a list of items on a GPU, that which needs to be sorted must be converted into a form of data which the GPU can handle, namely: numbers. The GPUSort project (www[SOR]), for example, sorts strings by first converting them into numbers, and sorting those numbers. Kumar describes this as: "*The GPU cannot do string compares. Hence, the keys are first converted into floating point numbers so that they can be operated inside the GPU*" (www[KUM]). For the sorting implementation here this string conversion is not included as the essence of the exercise is to demonstrate the GPU's strengths, so the input for this program is defined as numbers (specifically 8 bit [0 to 255] integers, supplied in ASCII text format).

The fragment processor is the most suitable location on GPU hardware to perform these operations as it requires many texture addresses and a large number of calculations. The vertex processor, while now becoming a texture addressing component, does not yet compare to the throughput capabilities of the fragment processor. The fragment processor does not currently support scatter operations, therefore it cannot write its results to a random memory address. The algorithm is

structured in such a way as to avoid this requirement, operating instead in gather form. The fragment processor (or rather, one of the available parallel operating fragment processors) operates solely on one fragment, or potential pixel in the frame buffer, at a time. During this operation, the fragment processor may access textures from the GPU memory to retrieve input, after which a single output is created. This output is generally the colour intended as the pixel update to the screen, for graphical purposes, but in GPGPU applications the output is instead viewed as a single, four component tuple which can represent data in the more conventional sense.

The texture addressing originally intended in graphical situations as the ability to retrieve colour information for paint onto surfaces, serves in a GPGPU sense to allow the inputting of large amounts of numerical data embedded into the textures. The data embedded into the textures in this implementation is a series of 8-bit numbers ranging from 0 to 255. The fragment processor inputs the data by accessing the texture and presents one output colour per fragment evaluated; this output being an element of the sorting data. Each fragment processor may access numerous textures; at random coordinate points within, but may only emit one resultant value for the output (Buck, 2005).

One aspect of performing the sorting actions on the GPU is that it is trivial to display the textures in memory after each pass, thus providing visual feedback as to the progress of the algorithm; since the colour of each pixel represents the number by which it is being sorted, one can literally see the elements being sorted in real time. Such feedback is an option which was availed of in this implementation.

For the implementation stages of this dissertation, Cg was chosen as the language with which to program because:

- It is well documented and freely available.
- It is platform independent.
- It is API neutral, operating with both OpenGL and DirectX.
- It is a familiar language construct, similar to C.

3.4.2.1 Frame Buffer Objects

For both applications, Frame Buffer Objects (FBO), a relatively recent feature of GPU programming, meant that the programs did not necessarily need to

reflect all calculations with an ultimate drawing of pixels to screen; rather the destination frame buffer was set to an area of memory (in these cases textures), which could then be rapidly redeployed as inputs to the next iteration. The use of up to four colour attachments to each FBO proved efficient for buffer reallocation. This resulted in the ability for feedback looping and a higher rate of processing, since vertical synchronising with a physical screen was not necessary. The FBO facility was provided by the OpenGL framework (www[OGL]). However, utilising such recently developed techniques preclude the use of these programs on older hardware, although this is not an issue for the future as graphics cards currently being produced are sufficiently powerful.

3.4.3 Application on a CPU

For the demonstration of sorting with a familiar CPU implementation, the following three methods were used:

- Qsort, the C standard library.
- Std::sort, the C++ standard template library sorting routine.
- Bitonic sort, as written in a similar fashion to the GPU equivalent.

The former two methods served as quantitative benchmarks for speed and ease of use, whereas the latter was an exercise to demonstrate the computational workload which the GPU was performing, and how that would compare on a CPU. All three CPU sorting methods were combined into one program, labelled CPUbitonicSort.

3.5 Motion Estimation

For this implementation, the task of calculating motion vectors between successive frames of video sequence was chosen. The calculation of motion vectors involves analysing the difference between two successive frames, and attempting to minimise that difference by finding motion among items in the frame. Block based motion estimation is the technique of splitting the reference and destination frames into equal areas (blocks) and searching for each reference block occurring at an offset location in the destination frame. By finding as close a match as possible, it

allows for the efficient construction of the target frame using the reference frame (already known), the motion vectors to apply to it and then the residual difference between the predicted image and the intended target image. The objective is to reduce the residual to as little as possible, resulting in a compacted datastream. Block based motion estimation was used in this instance, with exhaustive full search with single pixel accuracy within the defined search window. The Exhaustive Block Matching Algorithm (EBMA) is not an efficient method, since it searches every possible offset with no weighting or guidance; however, it does provide accurate results, not falling prey to local minima as other algorithms may. Only forward motion estimation is considered in this dissertation; other permutations exist but are not necessary to demonstrate the block matching facility.

3.5.1 Motion Estimation within Video Compression

Calculating the motion vectors comprises one major part of the general video compression task for most video compression and transcoding methods. After the motion vectors have been found, the predicted image is constructed, compared against the destination, or target frame and a DCT operation of the residual data is usually required thereafter. Most implementations of full video compression involve run length encoding to further compress the datastream. With respect to the following diagram, the implementation presented here constitutes the Motion Estimation and Motion Compensation sections:

Motion Estimation within Video Encoding

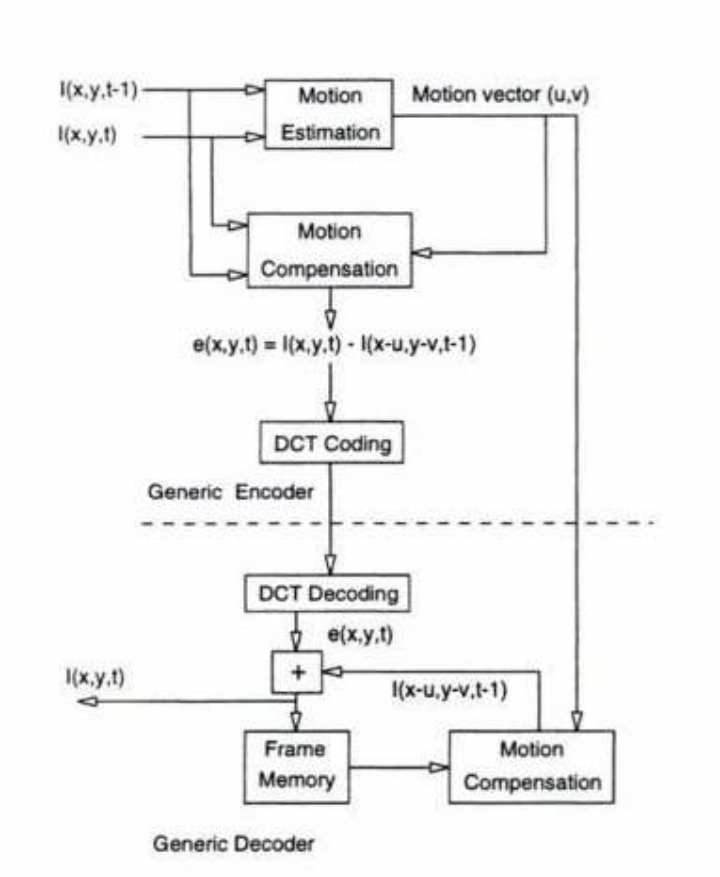


Figure 9. Source: Bhaskaran and Konstantinides, 1997.

Note: in the above diagram, *I* is the image stream, *x* and *y* are the horizontal and vertical components, *t* is the time slice and *e* is the image difference.

3.5.2 Implementation on a GPU

For this task, the choice of block size as 8 pixels was selected. This is a relatively small block size as the more often quoted block size is 16 pixels. The number for block size refers to the number of pixels on one side of the block. A smaller block size is associated with more computationally expensive routines, but suited the GPU implementation as 64 texture calls were required for each of the reference and destination textures, approaching the limit of operations which a fragment processor can handle on today's commodity GPU. A block size of 16 pixels was found to require more temporary registers on the GPU than were available; resulting in compile errors.

In this program, a search window of 7 pixels was chosen. This was selected as it represented sufficient distance to capture most of the motion occurring in the test sequences. The search window is the maximum pixel distance with which the destination frame is offset in either the horizontal or vertical axes, while attempting to match against a reference frame. As the search window n is increased, the number of computations scales as $(2n+1)^2$.

To compare the reference block with the destination's offset block, the two block-sized images are subtracted and their absolute difference is evaluated. A number of methods may be chosen such as Sum Absolute Difference (SAD) or Mean Square Error (MSE). The SAD method was chosen for this implementation as it accurately reflects the magnitude of difference at the currently tested offset, while minimising the computation required. In the GPU implementation, 8-bit formats were used, so in order to minimise the occurrences of truncation at 255, the values of the SAD were divided by 4 (a configurable scale within the program).

The reference and destination frames were first identified (loaded from bitmap file format in this instance) and from there the motion vector search was split into four major stages, as follows:

- Calculate an image subtraction of the two input textures (one is the reference and the other is the currently offset target) and write the output to a third texture.
- For each 8*8 pixel block, sum the difference (by reading the third texture)
- Compare the block summations to those found so far and record if lower
- Record the offset used to find the current lowest SAD for each block

After doing the above steps for each offset in the window of search area, the resultant array of offsets recorded denotes the motion vectors associated with that frame pair. The following diagram illustrates the block matching activity, where R is the search window and d is the motion vector found:

Exhaustive Block Matching Algorithm

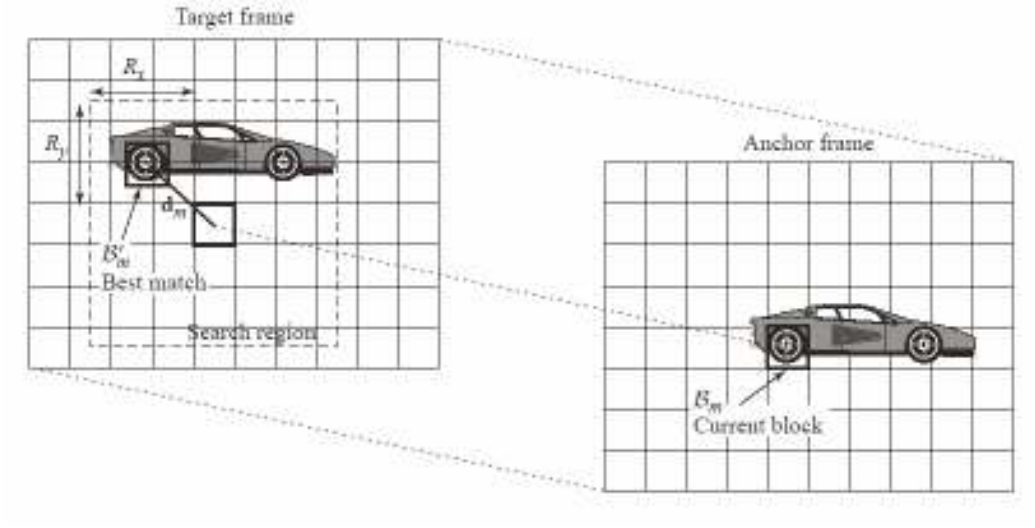


Figure 10. Source: www[WAN]

3.5.3 Implementation on a CPU

The calculation of motion vectors, performed on the CPU, was completed with the addition of a function to the main GPU program. The intention of the CPU function is to replicate the basic methods employed in the GPU sections. While this is likely not optimal it was chosen for this dissertation as it forces both the CPU and GPU to do a more equal number of calculations in the task.

4. Results

4.1 Test Conditions

A series of DOS batch scripts (called runTest.bat) were written to facilitate the execution of the tests. This permitted the passing of command line parameters and increased the portability of the test suite. Both platforms described below reflect commodity mainstream graphics cards, RAM and high performance CPUs. Microsoft Windows XP was the operating system in use for both scenarios.

- Testing Platform 1:

The CPU was a Pentium 4 Extreme Edition running at 3.88GHz with HyperThreading enabled, in conjunction with 1GB of DDR2/533 RAM. The GPU used was an ATI Radeon 9600 with a core speed of 400MHz and 4 pixel processors.

- Testing Platform 2:

The CPU was a Pentium D 840 Dual Core, running at 3.2GHz without HyperThreading, in conjunction with 0.5GB of DDR2/533 RAM. The GPU used was an ATI Radeon X1300 with a core speed of 450MHz and 4 pixel processors.

4.2 Sorting

For the GPU times recorded, the test invoked 10 iterations of the full sorting process. After the first sort was complete, that output is used as the input to the next sort. This sorting algorithm has equal best and worst case timings, however for the CPU sorting routines, their initial sanity checks quickly reveal that an already sorted array does not require further sorting work, so the process of simply looping the sort routine for those scenarios was not suitable, hence only one iteration was performed.

The input for each of the sorting operations was the same “input-255.txt” file. This file contained over 1 million integers, between the values of 0 and 255. This file was created by a separate ancillary program utilising the C random function. The numbers of actual items which were loaded and subsequently sorted were 256^2 and 512^2 because these suit the lowest common denominator, power of two texture sizes for the GPU. While this is a concession to the GPU implementation it may be noted that non-power of two textures are becoming part of commodity graphics card

capabilities so this will not be an issue in the future. The resultant timings recorded exclude file operations and were measured with the intention of reflecting only the computational time spent by each method. For the following results, lower times are better, indicating faster completion of the sort. The below graph summarises the performance of each hardware option for sorting an input of 512^2 items. It can be seen that the GPU implementation performs favourably with the model CPU sorting routines of Qsort and std::sort.

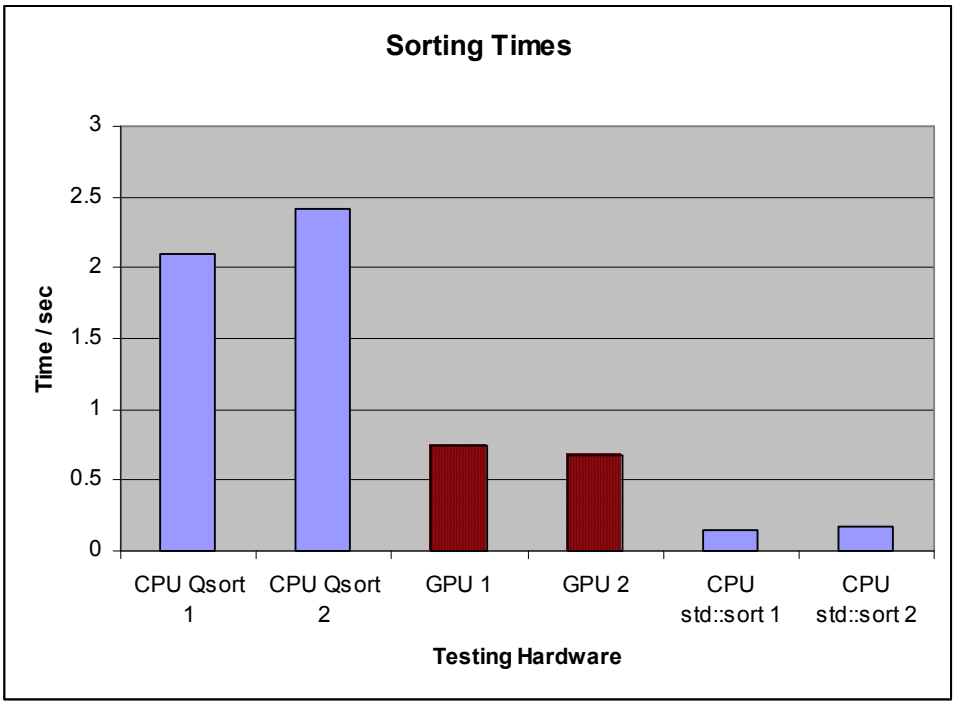


Figure 11. Source: Appendix 1.2.

Not included in the above graph, are the bitonic sorting results of the CPU based section of the implementation presented here, which were an order of magnitude larger (24 and 30 seconds) than the above values. It was found that the GPU based implementation processed the input 45 times faster than the CPU based bitonic implementation. The CPU based implementation was not optimal, as demonstrated by the performance of the other CPU bound Qsort and std::sort; however these standard C libraries are highly optimised and employ faster algorithms such as modified QuickSort.

4.3 Motion Estimation

The motion estimation implementation enacted an exhaustive block matching with a block size of 8 pixels and a search window of ±7 pixels. For the tests conducted, three video sequences were used, with 25 frames being used for each sequence. For each of these frame pairs, the motion estimation procedure was evaluated. The input images were 256 by 256 pixels each and only in one colour plane (grayscale). A single program was used which exercised the algorithm on each of the GPU and CPU in turn, recording the time required for each. The algorithm employed is implemented in four stages (described above) which map to each of four fragment shader programs. The number of texture calls and variables employed in some of the shader programs exceeded the limits for the ATI Radeon 9600 GPU so it could not be used for this particular implementation.

The primary intention of acquiring motion vectors is so that they may be used at the video decoding stage to create a predicted image which is as close as possible to the target frame. The following three images in figure 12, from the Coast Guard sequence demonstrate the effect of motion estimation achieved in this implementation:

Frame Prediction

Origin Frame:

Target Frame:

Predicted Frame:

Figure 12. Source: Coast Guard video sequence

The above images appear similar; such is the effect of temporal redundancy in video images, whereby consecutive frames in a video sequence do not change very much. The value of the motion vectors can be observed qualitatively though at some high change areas. For example, at the left edge of the frame, the predicted image has successfully compensated for the tail end of the other boat disappearing out of view, leaving a wave after it. Less noticeable however, is the scanning effect of only 5 pixels from right to left (as time progresses from origin to target frame). This scanning has been compensated for and as demonstrated below has a quantitative benefit.

To better demonstrate the effect of the motion compensation occurring, two difference images were generated by the GPU-CPU-video program. In the difference image, a black area indicates both frames closely resemble each other quantitatively, and a white area indicates a difference. The aim of motion estimation is to reduce the quantitative difference (and thus the white areas in the difference images). The first image below is an absolute difference between the origin image and the target image. This gives an indication of the amount of data which a video file would need to store in order to generate the target frame. The second image is an absolute difference between the predicted frame and the target frame. This gives an indication of the residual information which a video file must contain in order to generate the target frame while using the motion vectors.

From the images in figure 13 below, it can be appreciated that the motion estimation was successful in reducing the amount of residual information (appearing as white areas) which requires encoding. The following two images have been gamma corrected by a factor of +1.5 to accentuate the differences being highlighted.

Image Differences

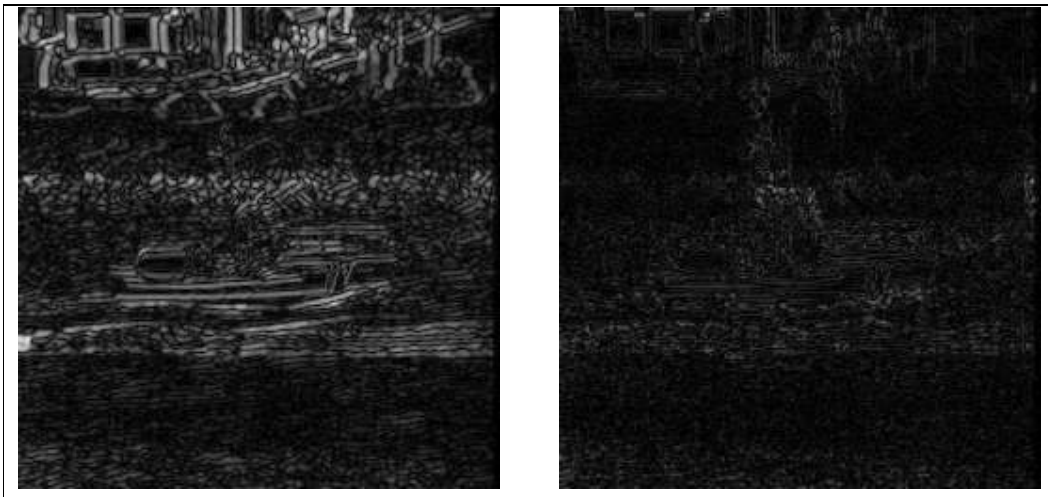


Figure 13. Source: GpuCpuVideo.exe output

The array of motion vectors found for a sample frame pair was exported to Matlab wherein a 2D image of the velocity vectors they embody was produced. The image demonstrates the central area of the coast guard vessel which is not moving much inside the frame of the video, the upper background which is moving right to left and the lower foreground depicting chaotic surf atop a generally right to left scanning waterline. The reason the arrows point right as the video scans from left to right indicates that for the predicted image, a given block is composed of a motion

vector which tells the decoder where to find the best matching offset block in the origin frame.

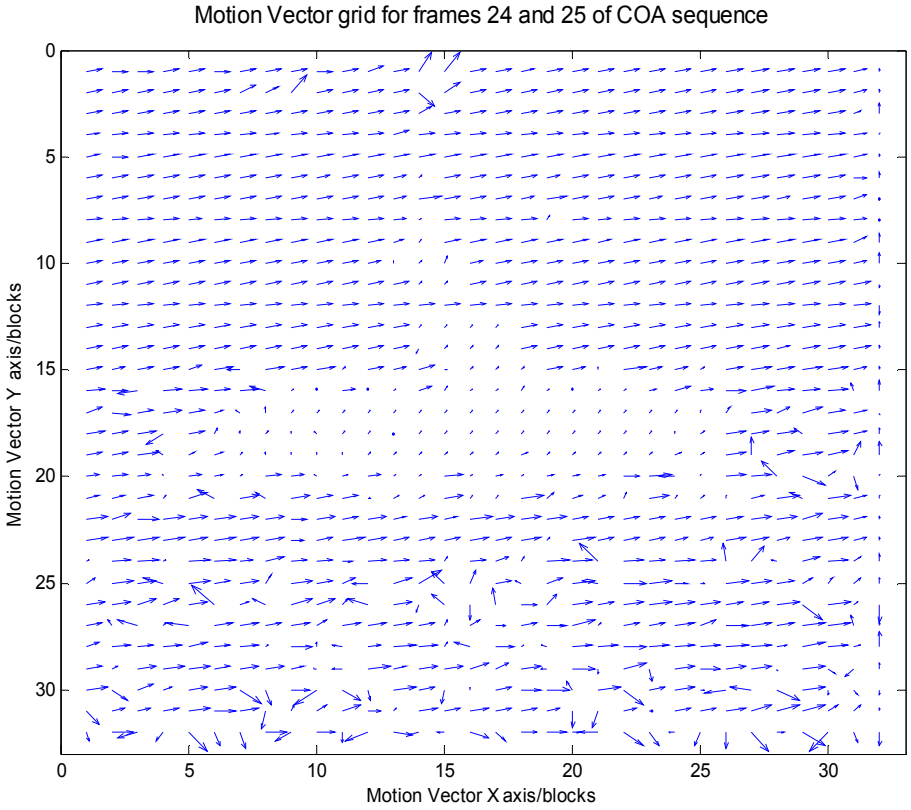


Figure 14. Source: GpuCpuVideo.exe output

The following image, in figure 15, is composed of the motion vectors superimposed upon the origin frame. The small motion vectors in the central boat region indicate that as the video sequence progresses, the boat stays relatively still, in the centre of the image, whereas the background is almost uniformly scanning from right to left.

Origin Frame with Superimposed Motion Vectors

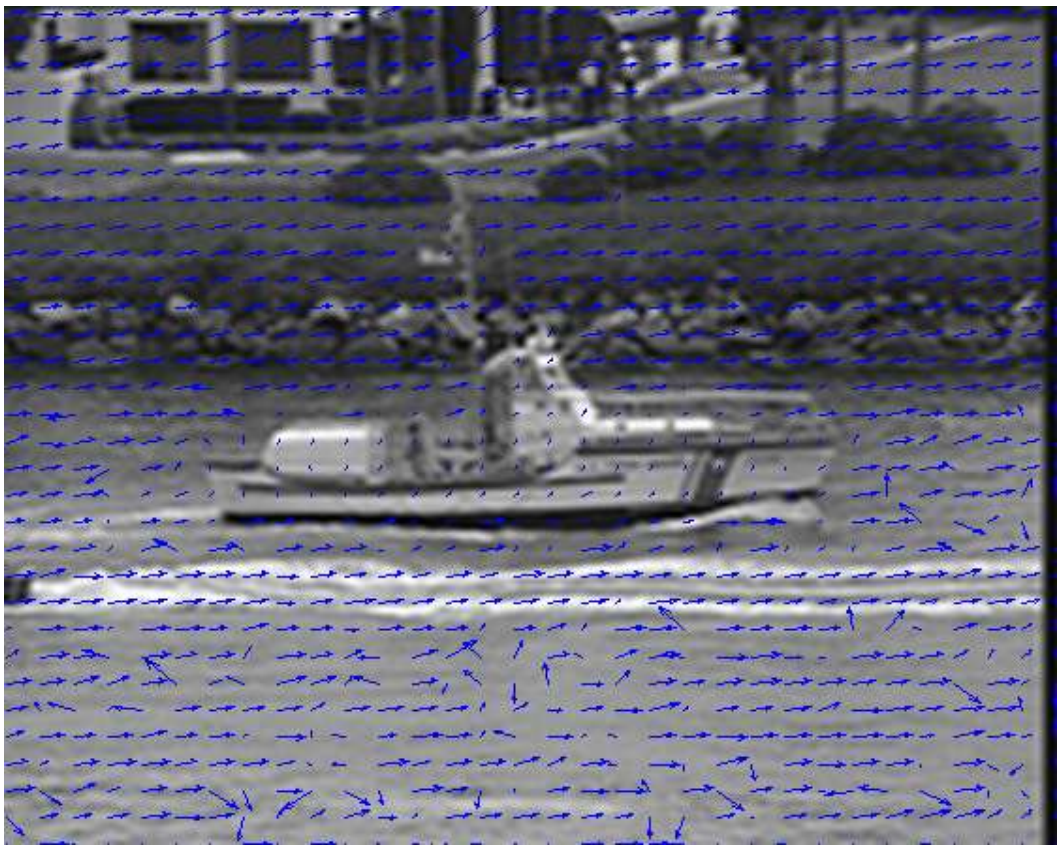


Figure 15. Source: GpuCpuVideo.exe output

For each pair of images in a video sequence which were processed by the program, a portion of text was output to file. One such sample is given in figure 16:

Sample Video Result

```
Start of results ----------
Input files:Data\FOR\for00.bmp and Data\FOR\for01.bmp
GPU performance:
0.188000 seconds for 1 iterations.
0.188000 seconds per iteration.(5.3 FPS)
Using a search window of 225 positions.
1196.809 full image permutations per second evaluated.
Block size of 8 in an image of size:256 by 256.
78434040 block permutations per second evaluated.
CPU performance:
0.812000 seconds for 1 iterations
0.812000 seconds per iteration.(1.2 FPS)
--------------------------------------------------
Motion Vectors matched in 1024 of 1024 instances.
Image difference 1 (image 1 and 2):PSNR: 25.454 (dB), SAD: 483475
Image difference 2 (image 2 and constructed):PSNR: 36.939 (dB), SAD: 129894
```

Figure 16. Source: GpuCpuVideo.exe output

The output details two image frames which are being evaluated at this stage. Firstly the GPU motion vector method produces its timing results and a listing of the image/block dimensions. Secondly the CPU motion vector timings are given. After this, a comparison of the two sets of motion vectors yields a count of the number of matching vectors. Finally, two lines are output detailing the qualitative evaluations of the differences between the original frame versus the target frame, and the predicted/constructed frame versus the target frame.

4.3.1 Process Quality

Within the program, the constructed frame is generated by using the origin frame and superimposing the motion vector adjusted block upon it. As the Sum Absolute Difference decreases for the predicted image versus the target image, this proves the motion vectors successfully decreased the amount of residual information

which is required to be encoded for video compression. The above output format was generated for each image pair, in each of the three video sequences. The following graph demonstrates the reduction in Sum Absolute Difference after the motion estimation has been performed for one of the video sequences:

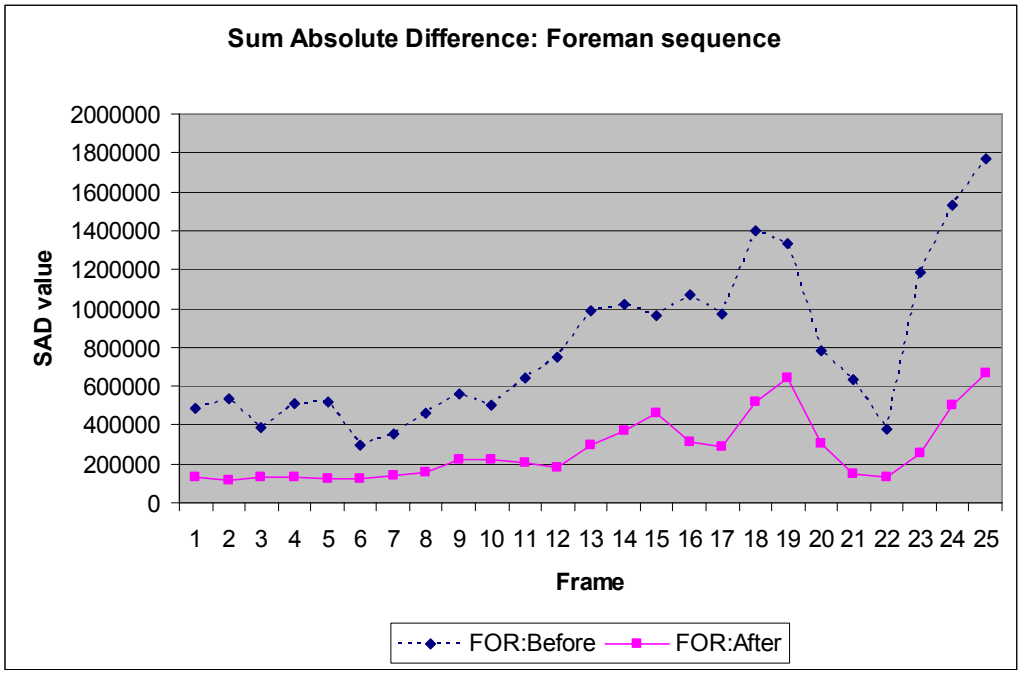


Figure 17. Source: Appendix 2.3

As the sum absolute difference decreases after the motion compensation, the PSNR (Peak Signal to Noise Ratio, equation provided in glossary) of the difference image of the pair correspondingly increases. The PSNR figure is a common measurement for quantising the information contained in images. In this situation the PSNR value is being calculated from the image difference of the origin frame versus the target frame, and the origin frame versus the predicted frame. The following figure demonstrates the increase in PSNR after the motion vectors have been found and applied:

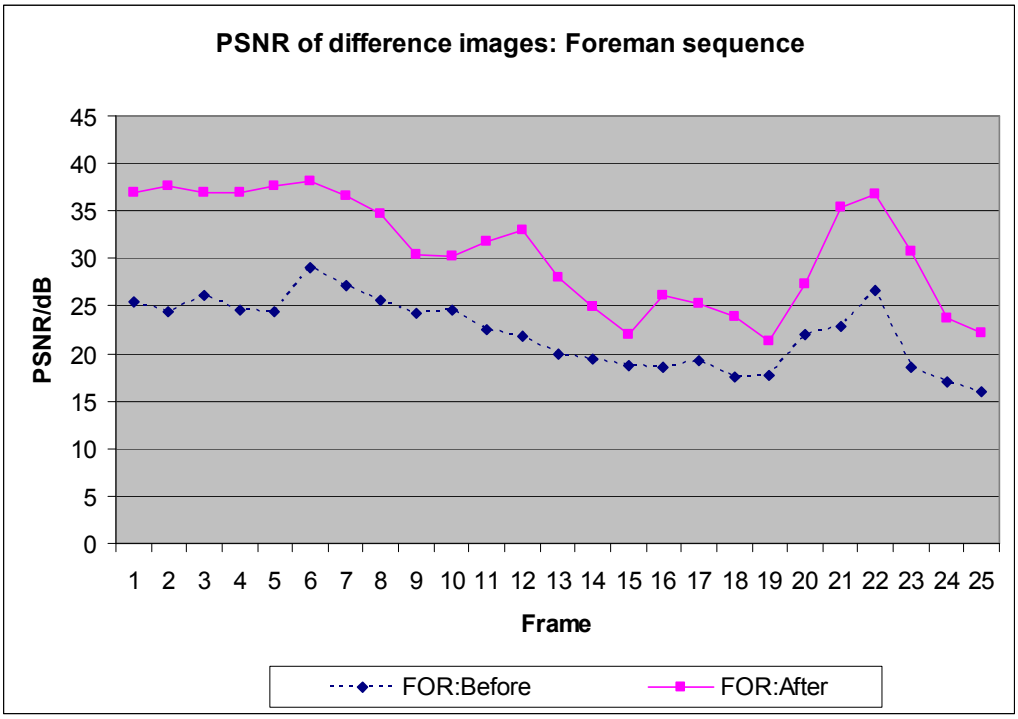


Figure 18. Source:Appendix 2.5.

The number of motion vectors which matched between the two methods of calculation (GPU and CPU) was tracked. With 75 images evaluated and 1024 blocks per image, a total of 76800 motion vectors were calculated by each method. The motion vectors matched for 76773 of those, accounting for a 99.965% correlation between the two methods. The slight differences may be due to rounding errors and boundary scenarios such as blocks at the image edge. Further investigation into this minor divergence was not pursued.

4.3.2 Process Speed

One of the primary aims of this undertaking was to examine if the GPU could complete the motion estimation algorithm in a time comparable to the CPU. The timings results for this implementation found that the GPU actually completed the task in much less time than required for the CPU. The condensed timing results for the calculation of motion estimation on both the GPU and CPU is summarised in the following diagram which dramatically conveys the margin by which the GPU completed its calculations six times faster than the CPU:

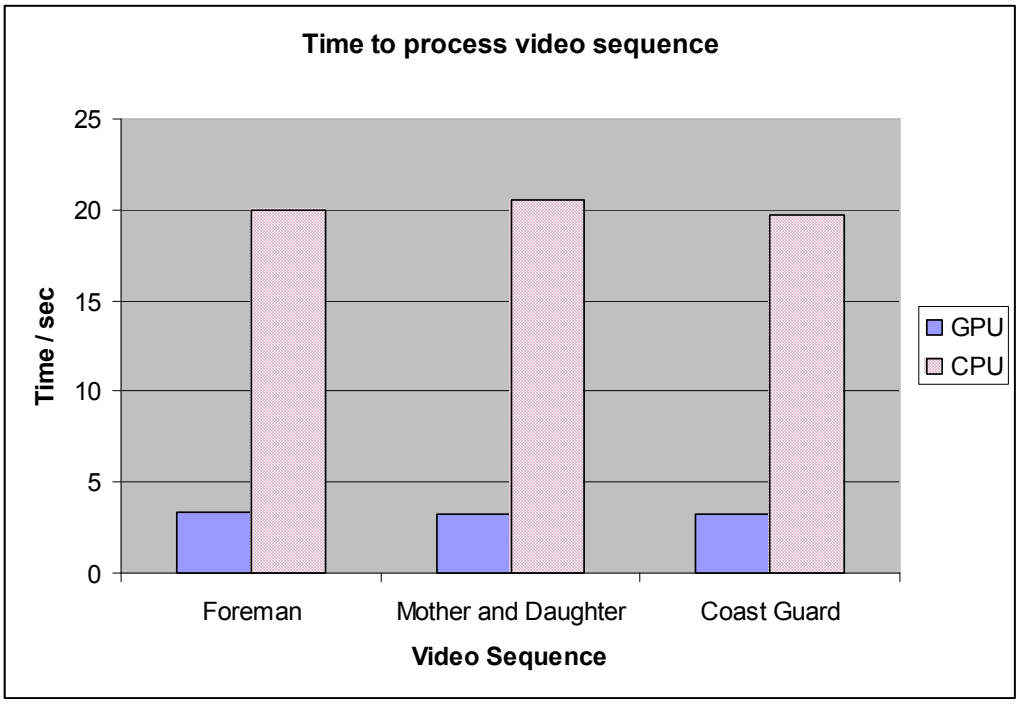


Figure 19. Source: Appendix 2.1.

Anecdotally, for an additional perspective, the video sequence test was run on a high performance graphics card, the ATI Radeon X1900XGT, and the results were captured. The results showed a 2.1 times speedup over the X1300 timings (i.e. the test completed in under half the time required for the X1300) and is demonstrated in the following graph of the findings:

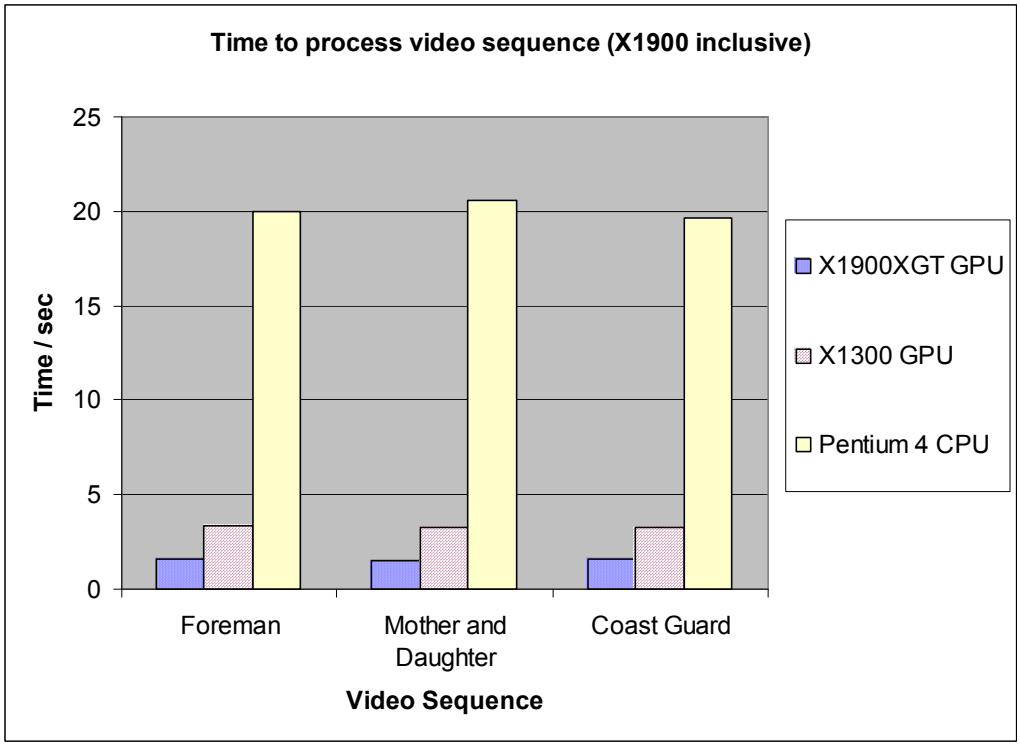


Figure 20. Source: Appendix 2.7.

4.4 Stress Testing

For this dissertation, the primary focus was to examine the applicability of computationally intensive algorithms running on a GPU. For programs which are being designed to run on a GPU, the intention is that the CPU based components of the programs take care of data delivery to the GPU, file handling, output management and miscellaneous tasks such as input value parsing. Thus the initial checks to confirm all input data for the GPU based routines is suitable are to be completed by the CPU. For this reason the GPU programs implemented here do not perform input value checking such as confirming the input video frames are of the correct dimensions etc.

Since this dissertation is concerned with the performance of the hardware to complete the tasks, a selection of under-clocking and over-clocking experiments were conducted. The ATI Tray Tool program was used to adjust the core clock speed of the GPU and the memory access speed on the graphics card. The normal speed of the GPU memory is 300MHz for the ATI Radeon 9600Pro. By adjusting the memory speeds over a range from -20% normal to +10% normal, no difference was observed

in the time taken for the GPU based sorting program. This indicates that the program is not memory bound, so latencies of the texture fetches appear to not be the limiting factor.

Adjusting the core frequency of the GPU was observed to have dramatic effects. For the ATI Radeon 9600Pro, a 50% over-clock of the 400MHz GPU core frequency resulted in a 35% faster runtime (i.e. Completed in 65% of the original time taken). Faster over-clocks ad-infinitum for even more performance gains were not feasible as the processor crashes beyond these points. The following two graphs demonstrate these findings:

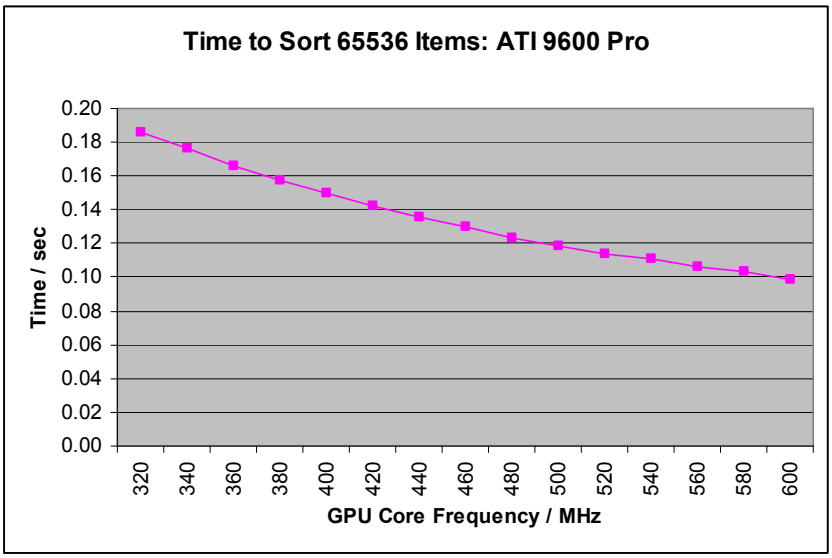


Figure 21. Source: Appendix 2.8.

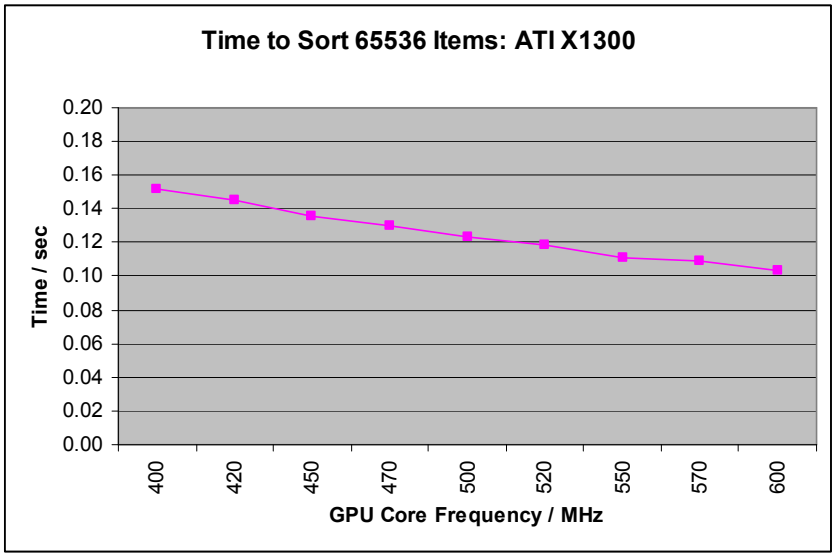


Figure 22. Source: Appendix 2.8.

5. Conclusion:

5.1 Commentary of Results

From the results obtained in the implementation presented here, the commodity graphics cards proved they are serious contenders for sheer processing speed in certain circumstances and the programming of them is both feasible and practical.

The GPU based sorting program demonstrated processing power 45 times that of an equivalent CPU implementation, however the nature of the bitonic sorting algorithm appears more suited to the GPU framework than the CPU. The GPU based bitonic sorting program achieved sorting times faster than the C Qsort library and approached that of the C++ std::sort library. As an opportunity for future work, the GPU sorting program could be made more modular, such as that of a dynamically linked library and thus usable by any mainstream programmer who wishes to benefit from the coprocessing power of the GPU, without troubling with the specifics of the actual GPU programming techniques. The sorting program implemented here produced quantitative results which can be compared against other published results of groups who have produced GPU sorting solutions. For comparison, the following published results for sorting times of 262144 (512^2) items is given in the following graph (figure 23):

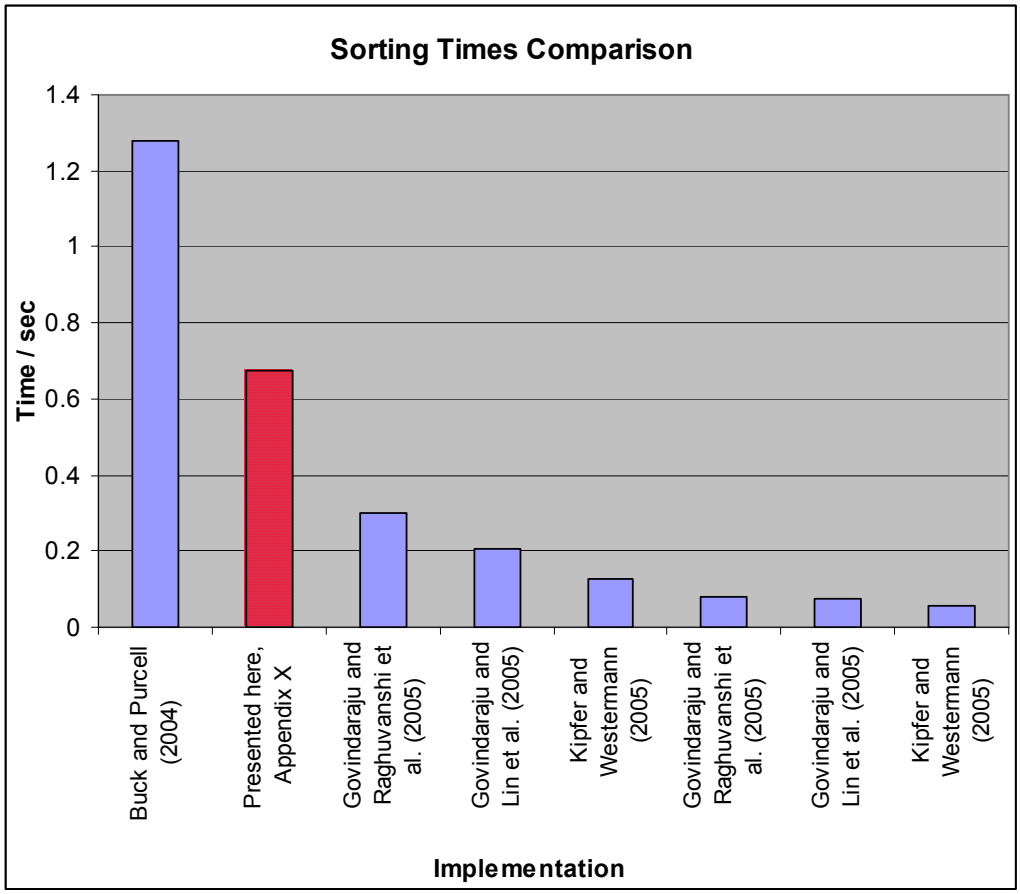


Figure 23. Source: Appendix 1.3

From the above graph it can be seen that the implementation presented here performed within the region of recently published results of GPU sorting. There are many optimisations which could have been implemented or factors altered to increase the performance, but were not included in the GPU sorting program presented here. Examples of such differences include:

- Over-clocking the core frequency showed large gains in performance, but is not included in the figure 23 data as it is not guaranteed to be reproducible on other hardware.
- The fragment shader code in this implementation was not hand-optimised, but rather the natural output of the Cg compiler was used.
- Performance testing of the available Cg standard library functions was not pursued, so optimisations may be available in the Cg code.
- The algorithm employed was the simpler version of Bitonic sort, rather than more complex, higher performing versions.

- Data-Packing, whereby multiple items of data can be stored in each of the 4 components per pixel, was not implemented.
- The implementation presented here is based on results from a commodity graphics card of only 4 pixel processors, whereas the other results are based on high-end graphics cards with 8, 16 and 24 such pixel processors.

The motion estimation results fared well also, displaying performance 6 times that of the CPU. While the Exhaustive Block Matching Algorithm is not optimal, it was implemented equally on both CPU and GPU. As a future opportunity, both implementations could be engineered to employ another of the more efficient block matching algorithms. Most of the aforementioned optimisations for GPU sorting also apply to the Motion Estimation GPU implementation, such as data packing, so further gains may be achievable.

The process of block matching for Motion Estimation has thus been shown to be viable for GPU implementation. As an opportunity for future work, the GPU Motion Estimation could be combined with GPU based DCT and IDCT as described by Green (2005) and Fang et. al. (2004) to create a GPU accelerated video transcoding program which could convert video files of one format (such as low compression MPEG2) to other formats (such as highly compressed MPEG4).

5.2 Short Term Technical Improvements:

To demonstrate the pace of advancement in the graphics industry, the following was a list of short term goals identified by Hanrahan in 2002 (www[HAN2]) as those which could aid the development of GPGPU applications:

- Full Set of arithmetic operators.
- Multiple Outputs (live variables, stream outputs).
- Conditional Branching.
- Floating point precision.

To a general extent, these have already been met, with current graphics cards able to claim support for each of the above items. This indicates the blistering pace at which the graphics industry is evolving, where the notional wishes of a commentator

in the field are converted into usable features of commodity GPUs in a few short years.

In addition to the primary hopes for the GPU future of increased operating frequencies and larger memories, Goodnight et al. (2003) also describe functionality in a GPU which would be of benefit as "superbuffers" (or at least cheaper context switching), more development tools such as debuggers, documentation and a global accumulator. Again, advances have been made for each of these points.

During the implementation phase of this dissertation a number of limits were encountered which could be alleviated by GPU developments which would allow finer granularity branching, so that the inclusion of an "if…else" condition is not so expensive, and an increased number of high speed temporary registers. There are currently only 32 such registers on an ATI Radeon 9600 thus limiting the breadth of computation which can be performed in a single pixel shader pass. These limits were circumnavigated by altering the algorithms but hardware support for the more general purposes of GPGPU would be beneficial.

The next revolution in GPU hardware for mainstream PCs is set to arrive with DirectX 10 and Shader Model 4.0 GPUs, bringing with it the "unified shader pipeline" which effectively redesigns the infrastructure of the programmable shaders. Shader processors will become generic and configurable, so that they can dynamically change from operating as a vertex shader one moment to being a pixel shader the next; thus permitting more flexibility in the hardware to meet the application's demands. Technology commentators such as Freeman have described some of the other changes upcoming in the hardware pipeline (www[FRE]):

The new pipeline adds some features to the mix including a Geometry Shader and Input Assembler. The former intercepts the data flow between the Vertex and Pixel Shaders and can add geometry effects on the fly; the latter is a way of taking graphics processing unit (GPU) data and directing it to various stages without using CPU resources.

Microsoft's upcoming release of Windows Vista will bring with it certain requirements which will increase demand for DirectX 10 capability on the PC.

5.3 Rasterisation and Ray-Tracing

Current GPUs are mainly concerned with the rasterisation method of graphics generation. It is the technique whereby primitives such as triangles or quadrilaterals are input to the GPU, which it orients and converts into pixel sized fragments which determine how the pixels will appear onscreen. There is a competing method of graphics generation called ray-tracing which is the method of simulating many individual rays of light through the modelled environment and then displaying the calculated resultant view. This method has been mainly the preserve of offline-rendering, used for creating animation films etc., not viable for the 60 frames per second arena of gaming. The rasterisation method is what current real-time GPUs have become optimised for, thus ensuring their widespread commercial value. When predicting long term future trends however, there are reports that this trend could be reversed in a transition to real time ray-tracing, or at least a symbiosis of both methods. Some figureheads of the industry speak of it as a possibility, such as Hurley (2005), a researcher at Intel:

Ray tracing has long been considered too expensive for mainstream rendering purposes. Movie production studios have only recently begun the transition to using it; however, the true cost of ray tracing has been very poorly understood until recently. It is now poised to replace raster graphics for mainstream rendering purposes.

If ray tracing becomes part of real time graphics, this could encourage radical changes in the nature of GPU hardware, likely to the benefit of GPGPU efforts, but if it occurs on the CPU then it could signal another revolution in the Wheel of Reincarnation, where the task of graphics is subsumed again within the CPU envelope. GPGPU methods however, offer the chance of GPU ray tracing; Purcell et al. (2002) have already shown that it is possible to use GPGPU techniques to perform ray-tracing on programmable GPUs and propose a possible path for smooth transitioning to include ray-tracing:

While many believe a fundamentally different architecture would be required for real-time ray tracing in hardware, this work demonstrates that a gradual convergence between ray tracing and the feed-forward hardware pipeline is possible.

5.4 Future Outlook

If GPGPU applications gain more of a foothold, their prominence may begin to influence the hardware vendors, who may in turn begin to provide dedicated hardware solutions to some of the current obstacles (such as scatter writing to random memory). For example, the GPU vendors could leverage their mass production capabilities to include custom physics and AI engine hardware into their GPU infrastructure, in order to compete with the PPU manufacturers. The currently prevalent option is that Havok, or a similar company/product can satisfy the market demand for such non-graphics computations via processing on the GPU, utilising the current graphics oriented pipeline.

GPUs are beginning to appear in embedded systems such as mobile phones. OpenGL ES (Embedded Systems), one governing standard in this field, is a simplified version of the primary specifications, which has already been declared for such devices. Similar to the PC situation, the GPU of these devices is a powerful co-processor. Its usefulness in a GPGPU scenario has not been explored though.

In July 2006 AMD announced that they would begin the process of merging with ATI for an acquisition cost of US$5.4 Billion (www[ATI2]). This merger between the CPU and GPU mainstays is scheduled to finalise in 2006. How this will affect the group dynamic between the other major parties (Intel and nVidia) is the source of speculation in the media, such as Takahashi who notes "*Because of that deal, the PC landscape has changed forever. Now there is an imbalance as Intel, Nvidia, and AMD-ATI try to find the centre of the future of computing*" (www[TAK]). One GPGPU perspective is that a hardware partnership between CPU and GPU could encourage more previously CPU based tasks being ported to the GPU.

GPGPU applications are emerging as an exciting trend in leveraging extreme processing power from widely available, cost effective hardware. The availability of higher level languages with increased potency, such as Cg, combined with the increasingly powerful GPU hardware and its proliferation to the PC community bodes well for the future of this (currently) niche field.

6. References

Backus, J., 1978, *Can programming be liberated from the von Neumann style,* ACM Turing Award Lecture, Communications of the ACM, Vol. 21 Issue 8. p.613.

Batcher, K.E., 1968, *Sorting Networks and Their Applications,* AFIPS Spring Joint Computing Conference 1968, pp.307-314.

Bhaskaran, V., Konstantinides, K., 1997, *Image and Video Compression Standards*, Kluwer Academic Publishers; Boston, p.104.

Buck, I., 2005, *Taking the plunge into GPU computing,* in GPU Gems 2, Pharr, M. (Ed.). Addison Wesley; NJ. p.509.

Buck, I, Fatahalian, K., Hanrahan, P., 2004, *GPUBench: Evaluating GPU Performance for Numerical and Scientifc Applications*, ACM Workshop on General Purpose Computing on Graphics Processors, ACM SIGGRAPH.

Buck, I., Purcell, T., 2004, *A toolkit for computation on GPUs*, in GPU Gems. Addison Wesley; NJ. p.621,629.

Crow, T. S., 2004, *Evolution of the Graphical Processing Unit*, Master's Thesis, University of Nevada, Reno.

Du Toit, S., McCool, M. D., 2004, *Metaprogramming Gpus with Sh*, Peters, A. K, Ltd., p.12.

Fang, B., Shen, G., Li, S., Chen, H., 2005, *Techniques for Efficient DCT/IDCT Implementation on Generic GPU*, Proc. of 2005 IEEE International Symposium on Circuits and Systems (ISCAS-2005); Kobe, Japan.

Fernando, R., Kilgard, M. J., 2003, *The CG Tutorial: The Definitive Guide to Programmable Real-Time Graphics*, Addison Wesley; NJ. p.13,17,30,55.

Goodnight, N., Wooley, C., Lewin, G., Luebke, D., Humphreys, G., 2003, *A Multigrid Solver for Boundary Value Problems Using Programmable Graphics Hardware*, Graphics Hardware 2003, San Diego.

Govindaraju, N. K., Gray, J., Kumar, R., Manocha, D., 2006, *GPUTeraSort: High Performance Graphics Coprocessor Sorting for Large Database Management*, Proceedings of ACM SIGMOD Conference, Chicago.

Govindaraju, N. K., Lin, M., Manocha, D., 2005, *GPGP: General Purpose Computations using Graphics Processors*, HPEC 2005, University of North Carolina, Chapel Hill. p.4.

Govindaraju, N. K., Raghuvanshi, N., Henson, M., Manocha, D., 2005, *A cache-efficient sorting algorithm for database and data mining computations using graphics processors*. Technical report, University of North Carolina, Chapel Hill.

Green, S., 2006, *GPU Physics,* nVidia CEDEC at Game Developers Conference; San Francisco.

Greß, A., Zachmann, G., 2006, *GPU-ABiSort: optimal parallel sorting on stream architectures*, Parallel and Distributed Processing Symposium, 2006. 20th International; Greece.

Hanrahan, P., 2004, *Stream Programming Environments,* GP2 Workshop, ACM SIGGRAPH 2004; Los Angeles.

Harris, M., 2005, *Mapping Computational Concepts to GPUs*, in GPU Gems 2, Addison Wesley; NJ. p.493.

Hurley, J., 2005, *Ray Tracing Goes Mainstream,* Intel Technology Journal 2005, vol. 8 Issue.2. Intel Corporation; Santa Clara.

Kedem, G., Ishihara, Y., 1999, *Brute Force Attack on UNIX Passwords with SIMD Computer,* Proceedings of the 8th USENIX Security Symposium 1999, Washington D.C., pp.93-98.

Kiel, J., Dietrich, S., 2006, *GPU Performance Tuning with NVIDIA Performance Tools*, Game Developers Conference 2006; San Jose.

Kim, T., Lin, M. C., 2003, *Visual Simulation of Ice Crystal Growth*, Eurographics/SIGGRAPH Symposium on Computer Animation 2003; san Diego.

Kipfer, P., Westermann, R., 2005, *Improved GPU Sorting*, in GPU Gems 2, Addison Wesley; NJ., p.745.

Kolb, C., Pharr, M., 2005, *Options Pricing on the GPU,* in GPU Gems 2, Addison Wesley; NJ., p.719.

Mark, W. R., Glanville, R. S., Akeley, K., Kilgard, M. J., 2003, *Cg: a system for programming graphics hardware in a C-like language*, Proceedings of ACM SIGGRAPH 2003, ACM Press; NY., p.896.

Micikevicius, P., 2005, *GPU Computing for Protein Structure Prediction*, in GPU Gems 2, Addison Wesley; NJ., p.695.

Myer, T. H., Sutherland, I.E., 1968, *On the Design of Display Processors*, Communications of the ACM, Vol. 11, no. 6., ACM Press; NY.

Owens, J. D., 2005, *Streaming Architectures and Technology Trends,* in GPU Gems 2, Pharr, M. (Ed.). Addison Wesley; NJ. p.460.

Owens, J. D., Luebke, D., Govindaraju N., Harris, M., Kruger J., Lefohn, A., Prucell, T. J., 2005, *A Survey of general-Purpose Computation on Graphics Hardware*, Eurographics 2005, State of the Art Reports, August 2005, pp.21-25.

Proudfoot, K., Mark, W. R., Hanrahan, P., Tzvetkov, S., 2001, *A Real-Time Procedural Shading System for programmable Graphics Hardware,* Proceedings of ACM SIGGRAPH 2001, ACM Press; NY.

Purcell, T. J., Buck, I., William M. R., Hanrahan, P., 2002, ACM Transactions on Graphics. Proceedings of ACM SIGGRAPH 2002, ACM Press; NY., pp. 703-712.

Ruge, T. G., 2001, Hello 3D World, Linux Magazine, Linux New Media AG; Munich. Issue 8, p.68.

Rumpf, M., Strzodka, R., 2001, *Using Graphics Cards for Quantized FEM Computations*, Proceedings of VIIP 2001, pp.193–202, 2001.

Venkatasubramanian, S., 2003, *The Graphics Card as a stream computer*, SIGMOD-DIMACS Workshop on Management and Processing of Data Streams 2003; San Diego.

6.1 Internet References

www[AMD]

AMD (2006).

Software Industry Embraces AMD's Upcoming PC Enthusiast Platform. Available from: http://www.amd.com/us-en/Corporate/VirtualPressRoom/0,,51_104_543_544~110132,00.html [Accessed 1st Oct. 2006]

www[ATI]

ATI Technologies Inc, (2006).

CrossFire: Multiply & Conquer. Available from: http://www.ati.com/technology/crossfire/index.html [Accessed 1st Oct. 2006]

www[ATI2]

ATI Technologies Inc., (2006).

AMD & ATI: A Processing Powerhouse. Available from: http://www.amd.com/us-en/0,,3715_14197_14198,00.html?redir=goBG01 [Accessed 1st Oct. 2006]

www[AVI]

ATI Technologies Inc., (2006).

ATI Avivo video converter. Available from: http://www.ati.com/technology/avivo/technology.html [Accessed 1st Oct. 2006]

www[BEN]

Buck, I., Fatahalian, K., Houston, M., Foley, T., (2006).

GPUBench. Available from: http://sourceforge.net/projects/gpubench [Accessed 1st Oct. 2006]

www[BLA]

Black, P. E., (2005).

Bitonic Sort, Dictionary of Algorithms and Data Structure, U.S. National Institute of Standards and Technology. Available from: http://www.nist.gov/dads/HTML/bitonicSort.html [Accessed 1st Oct. 2006]

www[DAW]

Dawson, B., Walbourn, C., (2006).
Coding for Multiple Cores. Available from:
download.microsoft.com/download/5/b/e/5bec52bd-8f96-4137-a2ab-df6c7a2580b9/Coding_for_Multiple_Cores.ppt [Accessed 1st Oct. 2006]

www[DUB]

Dubois, E., 2006.
Image Compression. Available from:
http://www.site.uottawa.ca/~edubois/courses/ELG5378/elg5378_notes_w06_ch5_pt1.pdf [Accessed 1st Oct. 2006]

www[FRE]

Freeman, V., (2006).
Platform Trends: DirectX 10 and Next-Generation Graphics, Jupitermedia Corporation. Available from:
http://www.hardwarecentral.com/hardwarecentral/reports/article.php/3616546 [Accessed 1st Oct. 2006]

www[GDE]

Graphic Remedy, (2006).
Products: gDEBugger. Available from: http://www.gremedy.com/products.php [Accessed 1st Oct. 2006]

www[GLU]

Kilgard, M. J., (1996).
The OpenGL Utility Toolkit (GLUT) Programming Interface API Version 3. Silicon Graphics, Inc. Available from:
http://www.opengl.org/documentation/specs/glut/spec3/spec3.html [Accessed 1st Oct. 2006]

www[GNU]

Free Software Foundation, (2006).
GDB: The GNU Project Debugger. Available from:
http://www.gnu.org/software/gdb [Accessed 1st Oct. 2006]

www[GRA]

Gray, J., (2006).
Summary of 2006 Sort Entrants. Available from:
http://research.microsoft.com/barc/SortBenchmark/2006%20Summary%20Comments.mht

www[GRE]

Green, S., (2006).
GPU Physics. Available from:
http://developer.download.nvidia.com/presentations/2006/cedec/2006-cedec-gpu-physics.pdf [Accessed 1st Oct. 2006]

www[HAN]

Hanrahan, P. (2004).
Stream Programming Environments. Available from:
http://www.graphics.stanford.edu/~hanrahan/talks/gp2/index.html [Accessed 1st Oct. 2006]

www[HAN2]

Hanrahan, P., (2002).
Why is Graphics Hardware so Fast. Computer Science Department, Stanford University. Available from:
http://www.graphics.stanford.edu/~hanrahan/talks/why/walk004.html [Accessed 1st Oct. 2006]

www[HAV]

Havok Inc. (2006).
Havok FX. Available from: http://www.havok.com/content/view/187/77 [Accessed 1st Oct. 2006]

www[HAV2]

Havok Inc. (2006).

Titles that use Havok Dynamics. Available from:

http://www.havok.com/content/blogcategory/29/73 [Accessed 1st Oct. 2006]

www[HED]

Hegde, M., Pemberton, D., (2005).

Ageia sets the record straight. Available from:

http://www.gdhardware.com/interviews/agiea/havoc/001.htm [Accessed 1st Oct. 2006]

www[HEU]

Heuston, M, (2006).

General Purpose Computation on Graphics Processors (GPGPU), Stanford University. Available from:

http://graphics.stanford.edu/~mhouston/public_talks/R520-mhouston.pdf [Accessed 1st Oct. 2006]

www[INT]

Intel Corp. (2006).

Intel Dual-Core Processors. Available from:

http://www.intel.com/technology/computing/dual-core [Accessed 1st Oct. 2006]

www[ISL]

Isler, C., (2006).

DirectX Then and Now. Available from:

http://craig.theeislers.com/2006/02/directx_then_and_now_part_1.php [Accessed 1st Oct. 2006]

www[KUM]

Kumar, R., (2006).

Learning from GPUSort. Available from:

http://defectivecompass.wordpress.com/2006/06/25/learning-from-gpusort [Accessed 1st Oct. 2006]

www[LIN]

Lind, R., (1997).

3D Visualization on Personal Computers. Available from:

http://www.student.nada.kth.se/~d91-rli/rapport/thesis.html [Accessed 1st Oct. 2006]

www[NVI]

nVidia Corp. (2006).

SLI Zone home. Available from: http://www.slizone.com/page/home.html [Accessed 1st Oct. 2006]

www[OGL]

OpenGL.org. (2006).

EXT_framebuffer_object. SGI Developer Central. Available from:

http://oss.sgi.com/projects/ogl-sample/registry/EXT/framebuffer_object.txt [Accessed 1st Oct. 2006]

www[OPE]

OpenGL.org. (2006).

OpenGL Overview. OpenGL.org. Available from:

http://www.opengl.org/about/overview [Accessed 1st Oct. 2006]

www[RIE]

Riegel, E. (2006).

OpenGL ARB to Pass Control of OpenGL Specification to Khronos Group, Khronos Group*,* Available from: http://www.intel.com/technology/computing/dual-core [Accessed 1st Oct. 2006]

www[ROB]

Robins, N., (2001).
GLUT for Win32. Available from: http://www.xmission.com/~nate/glut.html [Accessed 1st Oct. 2006]

www[SDN]

Sun Microsystems Inc. (2006).
Core Java, JavaDoc Tool, Sun Developer Network, Available from: http://java.sun.com/j2se/javadoc/index.jsp [Accessed 1st Oct. 2006]

www[SOR]

Department of Computer Science, UNC Chapel Hill. (2003).
GPUSort: High Performance Sorting using Graphics Processors, UNC Chapel Hill, Available from: http://gamma.cs.unc.edu/GPUSORT [Accessed 1st Oct. 2006]

www[SUN]

Sun Microsystems Inc. (2006).
How to Write Doc Commentsfor the Javadoc Tool, Sun Developer Network, Available from: http://java.sun.com/j2se/javadoc/writingdoccomments/index.html [Accessed 1st Oct. 2006]

www[TAK]

Takahashi, D., (2006).
The Coming Combo Of The CPU And GPU, Ray Tracing Versus Rasterization. Available from: http://blogs.mercurynews.com/aei/2006/08/the_coming_comb.html#more [Accessed 1st Oct. 2006]

www[THG]

Chung. H., (2006).
Multi-core processors may replace physics cards. Tom's Hardware Guide. Available from: http://tomshardware.co.uk/2006/09/26/idf_fall2006_quadcore_gaming [Accessed 1st Oct. 2006]

www[THG2]

Voelkel, F., Toepelt, B., (2006).

Intel's Core 2 Quadro Kentsfield. Tom's Hardware Guide.

Available from:

http://tomshardware.co.uk/2006/09/26/idf_fall2006_quadcore_gaming [Accessed 1st Oct. 2006]

www[TOR]

Torres, G., (2006).

ATI Chips Comparison Table, Hardware Secrets. Available From:

http://www.hardwaresecrets.com/article/131 [Accessed 1st Oct. 2006]

www[TOR2]

Torres, G., (2006).

nVidia Chips Comparison Table, Hardware Secrets. Available From:

http://www.hardwaresecrets.com/article/132 [Accessed 1st Oct. 2006]

www[TRE]

Trendall, C., Stewart, J., (2000).

General calculations using graphics hardware, with application to interactive caustics. Available from:

http://www.dgp.utoronto.ca/~trendall/research/egwr00/egwr00.pdf [Accessed 1st Oct. 2006]

www[WAN]

Wang, Y., (2003).

Motion Estimation for Video Coding. Available from: http://www-inst.eecs.berkeley.edu/~ee290t/sp04/lectures/motion_estimation.pdf [Accessed 1st Oct. 2006]

7. Appendices

7.1 Appendix 1. Sorting Results

7.1.1 Appendix 1.1 Sample Sorting Output

The following results relating to the GPU and CPU sorting programs were recorded from testing platform 1 (ATI Radeon 9600):

Sample GPU Sort Output:

```
GPU Sort with 256*256 items over 10 iterations:
Starting GPU sorting.
Starting at 0.125000
Finished loop 1 of 10.
Finished loop 2 of 10.
Finished loop 3 of 10.
Finished loop 4 of 10.
Finished loop 5 of 10.
Finished loop 6 of 10.
Finished loop 7 of 10.
Finished loop 8 of 10.
Finished loop 9 of 10.
Finished loop 10 of 10.
Sorting time: 0.1500 seconds per iteration
            ...
-----------------------------------------------
```

Sample CPU Sort Output:

```
CPU Sort with 256*256 items:
STDsorted in 0.031000 seconds
Qsorted in 0.156000 seconds
finishing: Counter: 137 by 0 in Time 4.578 , fps:29.9

CPU Sort with 512*512 items:
STDsorted in 0.140000 seconds
Qsorted in 2.093000 seconds
finishing: Counter: 172 by 0 in Time 26.235 , fps:6.6
```

7.1.2 Appendix 1.2 Sorting Times Collated

Test Platform 1:

		Timings/seconds			
		GPU	CPU std::sort	CPU Qsort	CPU Bitonic
Number of Items					
65536	$=256^2$	0.150	0.031	0.156	4.578
262144	$=512^2$	0.741	0.140	2.093	26.235

Test Platform 2:

		Timings/seconds			
		GPU Time	CPU std::sort	CPU Qsort	CPU Bitonic
Number of Items					
65536	$=256^2$	0.137	0.031	0.188	5.984
262144	$=512^2$	0.673	0.172	2.422	30.453

7.1.3 Appendix 1.3 Comparison with Published GPU Solutions

Time/ Seconds to Sort 262144 Items	**Referenced From**	**Algorithm and Hardware Employed**
1.28	Buck and Purcell (2004)	Bitonic Merge Sort, GeForce FX 5900
0.673	Presented here, Appendix X	Bitonic Merge Sort, Radeon X1300
0.302	Govindaraju and Raghuvanshi et al. (2005)	GPUSort, GeForce 6800 Ultra
0.208	Govindaraju and Lin et al. (2005)	GPU-ABiSort, GeForce 6800 Ultra
0.128	Kipfer and Westermann (2005)	Odd-Even Merge Sort, GeForce 6800 Ultra
0.08	Govindaraju and Raghuvanshi et al. (2005)	GPUSort, GeForce 7800 GTX
0.076	Govindaraju and Lin et al. (2005)	GPU-ABiSort, GeForce 7800 GTX
0.054	Kipfer and Westermann (2005)	Bitonic Merge Sort, GeForce 6800 Ultra

7.2 Appendix 2 Motion Estimation Results

7.2.1 Appendix 2.1 Timings for Motion Vector Calculation

Image pair	GPU:FOR	CPU:FOR	GPU:MAD	CPU:MAD	GPU:COA	CPU:COA
1	0.172	0.812	0.125	0.828	0.125	0.781
2	0.141	0.813	0.125	0.828	0.125	0.797
3	0.125	0.812	0.125	0.828	0.125	0.781
4	0.141	0.813	0.125	0.828	0.125	0.797
5	0.125	0.813	0.125	0.829	0.14	0.782
6	0.141	0.797	0.141	0.828	0.14	0.797
7	0.141	0.813	0.125	0.813	0.125	0.797
8	0.141	0.812	0.125	0.828	0.141	0.766
9	0.14	0.797	0.141	0.829	0.14	0.766
10	0.125	0.812	0.125	0.828	0.125	0.781
11	0.125	0.797	0.141	0.829	0.125	0.797
12	0.125	0.813	0.125	0.828	0.125	0.797
13	0.125	0.813	0.125	0.813	0.141	0.781
14	0.14	0.797	0.125	0.813	0.125	0.781
15	0.125	0.797	0.125	0.812	0.141	0.781
16	0.141	0.781	0.141	0.829	0.125	0.797
17	0.125	0.797	0.125	0.828	0.125	0.797
18	0.125	0.797	0.125	0.813	0.141	0.797
19	0.125	0.781	0.125	0.813	0.125	0.796
20	0.125	0.797	0.125	0.813	0.141	0.781
21	0.141	0.813	0.125	0.813	0.125	0.797
22	0.125	0.813	0.125	0.829	0.125	0.797
23	0.14	0.781	0.125	0.829	0.125	0.781
24	0.125	0.781	0.141	0.813	0.125	0.781
25	0.125	0.765	0.141	0.829	0.14	0.782
Totals	3.329	20.017	3.221	20.571	3.265	19.688
Speedup		6.012917		6.386526		6.030015

Total times taken to perform motion estimation on each video sequence

	GPU	CPU
Foreman	3.329	20.017
Mother and Daughter	3.221	20.571
Coast Guard	3.265	19.688

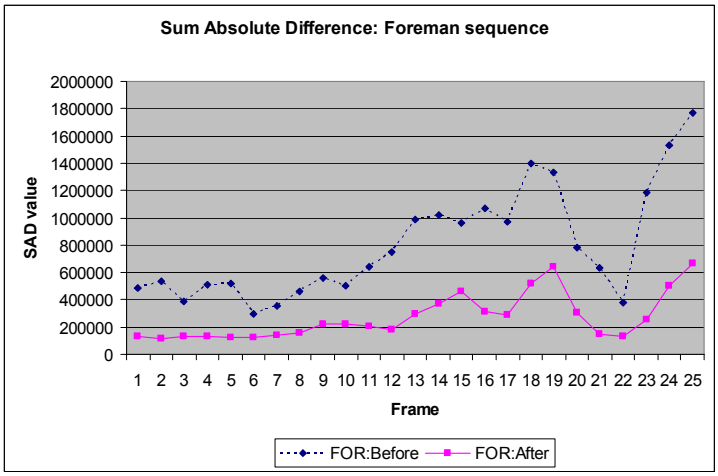
7.2.2 Appendix 2.2 Sum Absolute Differences Detail

Sum Absolute Difference for the difference images, before and after motion estimation applied

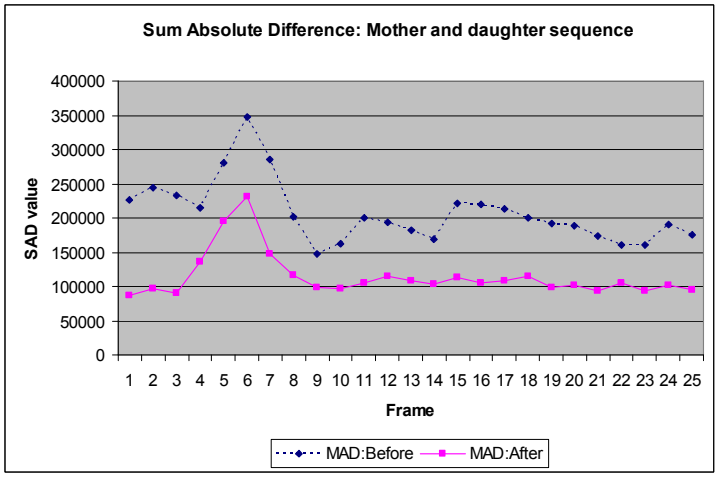
FOR:Before	FOR:After	MAD:Before	MAD:After	COA:Before	COA:After
483475	129894	225965	87451	796556	388997
538176	113474	243938	96893	786651	367615
383294	128824	232525	90961	803634	393224
510753	127676	215371	135506	813687	405585
517387	127439	281126	195343	805170	394358
296242	119683	348300	230405	903218	349182
356979	142886	284790	147975	1234294	394823
459318	159209	202390	116702	1534597	845757
558161	221922	146815	98856	1828011	1263065
498609	226192	162568	96270	1659661	510005
645428	204569	200059	105053	835693	367150
746612	183538	193928	114011	624869	349187
989337	292751	181542	107683	876409	355402
1023674	367425	169656	102836	1279745	491947
965433	457204	221707	113201	725207	285279
1073307	310311	219502	104489	764484	333962
975204	290231	212846	108497	834963	365048
1402278	521632	199602	114293	735931	270893
1329907	639211	192049	98656	819171	356073
778425	308545	188209	102209	934220	331066
630566	151927	173278	92647	1157102	362779
375212	133857	160774	104671	1009360	254899
1184891	255845	160227	94244	1011468	347888
1527495	504132	189434	102161	1005006	355414
1767229	669659	175133	95743	1078734	313403

7.2.3 Appendix 2.3 Sum Absolute Differences Graphs

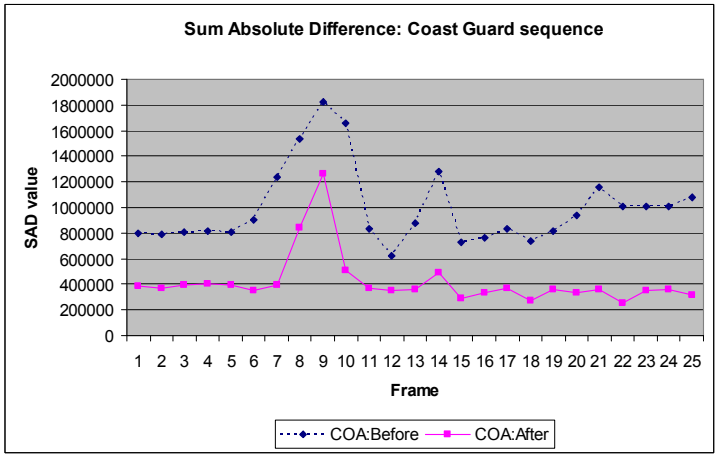
Sum Absolute Differences Graph: Video Sequence 1 (FOR)



Sum Absolute Differences Graph: Video Sequence 2 (MAD)



Sum Absolute Differences Graph: Video Sequence 3 (COA)



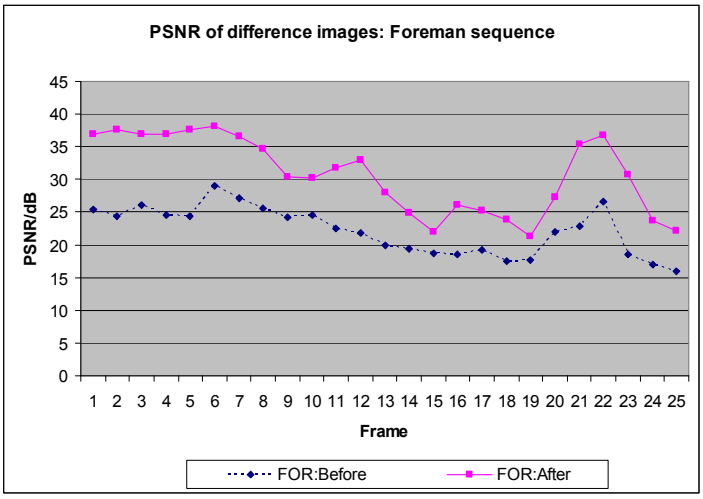
7.2.4 Appendix 2.4 PSNR Values Detail

PSNR for the difference images, before and after motion estimation applied

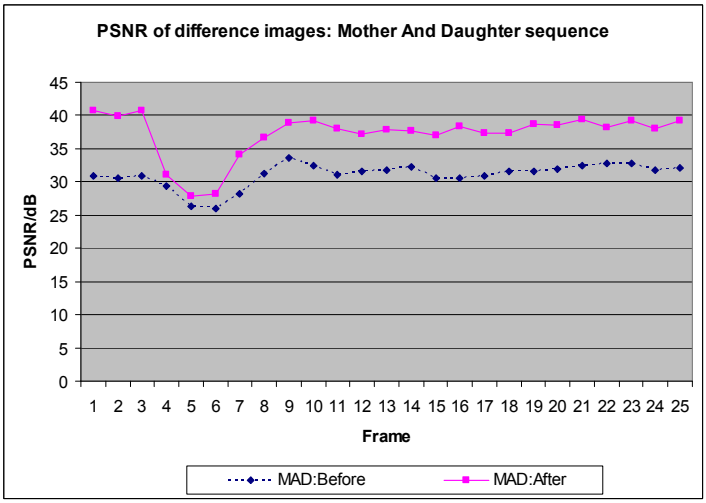
FOR:Before	FOR:After	MAD:Before	MAD:After	COA:Before	COA:After
25.454	36.939	30.973	40.73	21.947	27.333
24.36	37.621	30.57	39.841	21.946	27.654
26.056	36.938	30.934	40.691	21.762	28.58
24.554	36.912	29.368	31.029	21.73	27.546
24.432	37.602	26.363	27.865	21.839	28.894
28.988	38.115	26.024	28.173	20.999	29.826
27.172	36.499	28.182	34.089	18.532	28.28
25.569	34.634	31.19	36.721	17.255	20.905
24.179	30.39	33.667	38.895	15.955	17.867
24.487	30.228	32.455	39.151	16.6	25.612
22.47	31.723	31.136	38.095	21.767	29.616
21.747	32.964	31.664	37.115	23.548	29.879
19.908	27.996	31.719	37.785	20.858	29.744
19.389	24.982	32.186	37.638	18.271	25.695
18.679	22.01	30.577	36.97	22.486	31.282
18.582	26.178	30.631	38.458	22.206	30.18
19.193	25.251	30.887	37.423	21.512	29.495
17.44	23.832	31.503	37.405	22.309	31.322
17.643	21.252	31.53	38.676	21.72	29.716
21.907	27.33	31.841	38.505	20.364	29.63
22.856	35.395	32.387	39.369	19.009	28.728
26.564	36.701	32.749	38.132	19.797	29.925
18.632	30.738	32.853	39.193	19.988	28.765
17.011	23.787	31.791	38.057	20.055	29.492
16.021	22.229	32.123	39.255	19.602	30.624

7.2.5 Appendix 2.5 PSNR Values Graphs

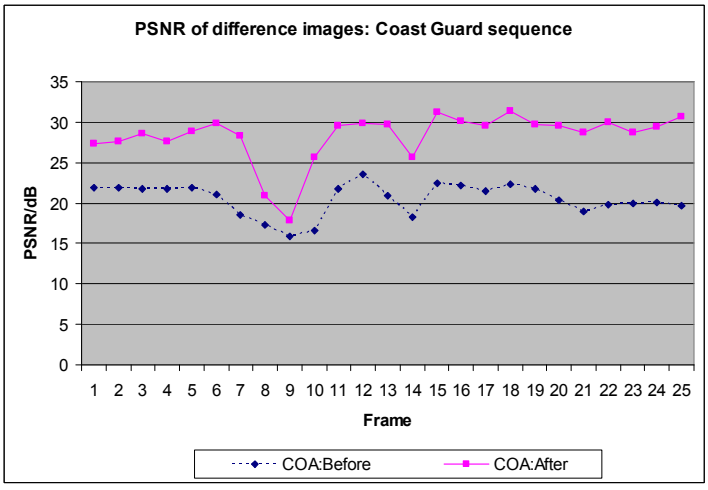
PSNR Value Graph: Video Sequence 1 (FOR)



PSNR Value Graph: Video Sequence 2 (MAD)



PSNR Value Graph: Video Sequence 1 (COA)



7.2.6 Appendix 2.6 X1900XGT Timings for Video Motion Estimation

Image Pair	GPU:FOR	CPU:FOR	GPU:MAD	CPU:MAD	GPU:COA	CPU:COA
1	0.0624	0.7344	0.0562	0.7312	0.0594	0.8094
2	0.0626	0.7406	0.0594	0.7282	0.0718	0.872
3	0.0594	0.7314	0.0594	0.7532	0.0624	0.7968
4	0.0594	0.7314	0.0594	0.725	0.0656	0.8626
5	0.0594	0.722	0.0656	0.7436	0.0592	0.8406
6	0.0594	0.7718	0.0594	0.778	0.0624	0.853
7	0.0594	0.8	0.0656	0.8374	0.0686	0.875
8	0.0624	0.7562	0.0626	0.8312	0.0626	0.7938
9	0.0626	0.7842	0.0656	0.85	0.0624	0.7656
10	0.0718	0.8156	0.0624	0.8188	0.0626	0.825
11	0.0688	0.875	0.0562	0.7592	0.0656	0.8
12	0.0656	0.828	0.0562	0.75	0.0688	0.8562
13	0.0688	0.8156	0.0562	0.7562	0.0626	0.8124
14	0.0656	0.7688	0.0562	0.7562	0.0626	0.7624
15	0.0686	0.928	0.0594	0.7406	0.0594	0.7626
16	0.0656	0.7874	0.0562	0.7968	0.0594	0.7408
17	0.0782	0.878	0.0564	0.822	0.0594	0.7594
18	0.0626	0.7438	0.0656	0.8314	0.0594	0.7438
19	0.0658	0.7688	0.0626	0.8282	0.0626	0.7532
20	0.0624	0.875	0.0594	0.7688	0.0562	0.7562
21	0.0626	0.828	0.0594	0.753	0.0594	0.7342
22	0.0656	0.9092	0.0624	0.7562	0.0594	0.7406
23	0.0656	0.8062	0.0594	0.7594	0.0594	0.7468
24	0.0688	0.8062	0.0564	0.7532	0.0592	0.7656
25	0.0624	0.7186	0.0594	0.7936	0.0592	0.7406
Totals	1.6158	19.9242	1.497	19.4214	1.5496	19.7686
Speedup		12.330858		12.973547		12.757228

7.2.7 Appendix 2.7 Total Times for Motion Estimation

	X1900XGT GPU	X1300 GPU	P4 840D CPU
Foreman	1.6158	3.329	20.017
Mother and Daughter	1.497	3.221	20.571
Coast Guard	1.5496	3.265	19.688
Speedup of X1900XGT over X1300	2.105139		

7.2.8 Appendix 2.8 Stress Testing:

Core Clock Frequency Adjustments
GPU: ATI Radeon 9600 Pro
Normal core frequency: 400 MHz

Core Frequency / MHz	Time to Sort 65536 items / sec
320	0.1860
340	0.1766
360	0.1656
380	0.1578
400	0.1500
420	0.1422
440	0.1359
460	0.1297
480	0.1235
500	0.1188
520	0.1141
540	0.1109
560	0.1063
580	0.1032
600	0.0984

Core Clock Frequency Adjustments
GPU: ATI Radeon X1300
Normal core frequency: 450 MHz

Core Frequency / MHz	Time to Sort 65536 items / sec
400	0.1515
420	0.1453
450	0.1359
470	0.1296
500	0.1234
520	0.1187
550	0.1109
570	0.1093
600	0.1031

7.3 Appendix 3. Source Code: Sorting

7.3.1 Appendix 3.1 GPU Sorting: GPUbitonicSort.cpp

The following code is from GPUbitonicSort.cpp. This program sorts an input file of integers using the GPU.

```
/**
 * Bitonic Sorting on the GPU.
 * GPUbitonicSort.cpp
 * This program enacts the Bitonic sorting alorithm using OpenGl
 * and the GPU processor to sort an inputted list of integers.
 *
 * @param       inputfile  The file containing integers to be sorted
 * @param       imageSize  The magnitude of 1 side of the array used
 * @param       numberOfTestsToRun      The iterations to rerun
 * @param       debugFlag  1:On,Enable visualisations onscreen. 0:Off
 * @return      A successfule exit(0) signal.
 *
 * @author      Jason Ruane, DIT Bolton St. Dublin, Ireland. B773.2006.
 * @version     1.0
 */

// Preprocessor directives
// Invoke the Cg and OpenGL libraries required
#ifdef _MSC_VER
#pragma comment( lib, "cg/lib/cg.lib" )
#pragma comment( lib, "cg/lib/cgGL.lib" )
#endif
#pragma comment (lib, "opengl32.lib")

// Implementation specific items
#define maxArraySize 256                // Size of the image array
#define SleepTime 0                     // Milliseconds to allow CPU to
     pause
#define maxString 512                   // The maximum of characters in
     command line parameters

// Include files required
#include <stdlib.h>
#include <stdio.h>
#include <time.h>
#include <windows.h>
#include <winbase.h>
#include <assert.h>
#include <math.h>

// Include Cg requirements
#include "cg/include/Cg/cg.h"                                   // Cg runtime
     API
#include "cg/include/Cg/cgGL.h"                                 // OpenGL-
     specific Cg runtime API
#include "cg/include/GL/glext.h"                                // Local header
     file for GL extensions
#include "cg/include/GL/glut.h"                                 // Glut and
     OpenGL api

// EXT_framebuffer_object related definitions
// as per - http://oss.sgi.com/projects/ogl-
     sample/registry/EXT/framebuffer_object.txt
extern PFNGLISRENDERBUFFEREXTPROC glIsRenderbufferEXT = NULL;
extern PFNGLBINDRENDERBUFFEREXTPROC glBindRenderbufferEXT = NULL;
extern PFNGLDELETERENDERBUFFERSEXTPROC glDeleteRenderbuffersEXT = NULL;
```

GPUbitonicSort.cpp, continued.

```
extern PFNGLGENRENDERBUFFERSEXTPROC glGenRenderbuffersEXT = NULL;
extern PFNGLRENDERBUFFERSTORAGEEXTPROC glRenderbufferStorageEXT = NULL;
extern PFNGLGETRENDERBUFFERPARAMETERIVEXTPROC
     glGetRenderbufferParameterivEXT = NULL;
extern PFNGLISFRAMEBUFFEREXTPROC glIsFramebufferEXT = NULL;
extern PFNGLBINDFRAMEBUFFEREXTPROC glBindFramebufferEXT = NULL;
extern PFNGLDELETEFRAMEBUFFERSEXTPROC glDeleteFramebuffersEXT = NULL;
extern PFNGLGENFRAMEBUFFERSEXTPROC glGenFramebuffersEXT = NULL;
extern PFNGLCHECKFRAMEBUFFERSTATUSEXTPROC glCheckFramebufferStatusEXT =
     NULL;
extern PFNGLFRAMEBUFFERTEXTURE1DEXTPROC glFramebufferTexture1DEXT = NULL;
extern PFNGLFRAMEBUFFERTEXTURE2DEXTPROC glFramebufferTexture2DEXT = NULL;
extern PFNGLFRAMEBUFFERTEXTURE3DEXTPROC glFramebufferTexture3DEXT = NULL;
extern PFNGLFRAMEBUFFERRENDERBUFFEREXTPROC glFramebufferRenderbufferEXT =
     NULL;
extern PFNGLGETFRAMEBUFFERATTACHMENTPARAMETERIVEXTPROC
     glGetFramebufferAttachmentParameterivEXT = NULL;
extern PFNGLGENERATEMIPMAPEXTPROC glGenerateMipmapEXT = NULL;

// Function declarations
void   reshape(int w, int h);
void   keyb(unsigned char k, int x, int y);
void        loadImage(void);
void   loadFile(void);
void   initialise(void);
void   display(void);
void   saveFile(void);
void   cgErrorCallback(void);
bool   checkFramebufferStatus(void);

// Global variables required throughout
int writeTex = 0, toggleTex=0, readTex=1, actualAttachment, myPass,
     myLoop=1;
int debug=0, testsToRun, testsRan=0;
float myTemp2[4];   // a parameter being passed to the CG shader
float myTemp[4];    // a parameter being passed to the CG shader
float counter=0;    // count the frames rendered
// Timing for frames
double startTime,endTime, timeForUpdate=0, gpuStart;
// Dimensions of the image array used
float checkImageWidth,checkImageHeight;
// OpenGL related global variables
GLenum attachmentpoints[] = { GL_COLOR_ATTACHMENT0_EXT,
     GL_COLOR_ATTACHMENT1_EXT };
GLuint fbo, color, depth, depth_rb, latestTextureUpdated,
     destinationAttachment;
CGcontext   g_cgContext;
CGprofile   g_cgProfile;

// Filenames for usage
char inputFile1[maxString];
             // Input file which will be used
char outputFilename[maxString]="Data\\GPUoutputFile.txt";   // Ouput file to
     be used
char resultsFilename[maxString]="Data\\GPUresultsFile.txt"; // Performance
     file to be used

// Data array for saving the texture to
static GLubyte data[maxArraySize][maxArraySize][4];
static GLubyte outData[maxArraySize][maxArraySize][4];
static GLubyte checkImage[maxArraySize][maxArraySize][4];

// Placeholders for textures used as a data arrays
unsigned int  _iTexture, _jTexture, _jNewTexture, actualTexture,
     outDataTexture;
```

GPUbitonicSort.cpp, continued.

```
// Cg related variables
CGprogram     _fragmentProgram;  // the fragment program used to update
CGparameter   _textureParam;     // a parameter to the fragment program
CGparameter   outDataTextureParam;     // a parameter to the fragment
     program


/**
 * initialise
 *
 * Sets up the OpenGL and Cg environment in preparation for GLUT.
 *
 * @param       void
 * @return      void
 *
 * @author      Jason Ruane
 * @version     1.0
 */
void initialise(void)
{
    // Setup Cg
    cgSetErrorCallback(cgErrorCallback);       // Enable reporting of Cg
     errors
    g_cgContext = cgCreateContext();           // Initialise a Cg context

    // Get the best profile for this hardware
    g_cgProfile = cgGLGetLatestProfile(CG_GL_FRAGMENT);
    assert(g_cgProfile != CG_PROFILE_UNKNOWN);
    cgGLSetOptimalOptions(g_cgProfile);

     // FYI check to see this machine's largest texture size possible
     int maxtexsize;
     glGetIntegerv(GL_MAX_TEXTURE_SIZE,&maxtexsize);
     if(debug)
     {
       printf("GL_MAX_TEXTURE_SIZE, %d\n",maxtexsize);
       printf("CG_profile, %d\n",g_cgProfile);
     }

     // Setup inital variable values related to the Bitonic sorting sequence
     myTemp[0]=0;
     myLoop=1;
     myPass=2; // 1 greater than expected because of pre decrementing later

     // Setup the parameters to be passed to the Cg program
     myTemp2[0]=pow(2,(myPass-1));
     myTemp2[1]=checkImageWidth;
     myTemp2[2]=floor(pow(2,myLoop));
     myTemp2[3]=floor(pow(2,myPass));

     // FYI the bitonic sequence information
     if(debug)
     {
     printf("       Stage: %d   Step: %d\n",myLoop,myPass);
     printf("       Setting Offset: %f\n",myTemp2[0]);
     printf("       Setting pbufwidth: %f\n",myTemp2[1]);
     printf("       Setting stageno: %f\n",myTemp2[2]);
     printf("       Setting stepno: %f\n",myTemp2[3]);
     printf("After initialise: counter: %.0f  Pass: %d 2Stage
     %d\n",counter,myPass,myLoop);
     }

     // A float4 variable holding data to send to the Cg shader
     myTemp[0] = 1;
     myTemp[1] = checkImageWidth;
     myTemp[2] = checkImageHeight;
```

GPUbitonicSort.cpp, continued.

```
    myTemp[3] = checkImageHeight*checkImageWidth;

    // FYI check to find the largest possible renderBuffer for this machine
    GLint max_size = 0;
    glGetIntegerv( GL_MAX_RENDERBUFFER_SIZE_EXT, &max_size );
    if (debug)     printf("glRenderBuffer:: max size %d\n",max_size);


    // Generate a blank texture to be used as the destination of results
   glGenTextures(1, &_iTexture);
   glBindTexture(GL_TEXTURE_2D, _iTexture);
   glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_NEAREST);
   glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_NEAREST);
   glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_CLAMP);
   glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_CLAMP);
    // Set it up with null contents
    glTexImage2D(GL_TEXTURE_2D, 0, GL_RGBA8, checkImageWidth,
    checkImageHeight,
               0, GL_RGBA, GL_FLOAT, NULL);

    // Generate a texture and bind it to the array checkImage
    glGenTextures(1, &_jTexture);
    glGenTextures(1, &_jNewTexture);
   glBindTexture(GL_TEXTURE_2D, _jTexture);
   glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_NEAREST);
   glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_NEAREST);
   glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_CLAMP);
   glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_CLAMP);
   glTexImage2D(GL_TEXTURE_2D, 0, GL_RGBA8, checkImageWidth,
    checkImageHeight,
                0, GL_RGBA, GL_UNSIGNED_BYTE, checkImage);

    // Setup the viewport for OpenGL to be same size as input
    int viewport[4];
   glGetIntegerv(GL_VIEWPORT, viewport);
    glViewport(0, 0, checkImageWidth, checkImageHeight);


    // EXT_framebuffer_object required definitions
    // as per myxo.css.msu.edu:443/d2x-xl/trunk/arch/ogl/fbuffer.c
    char *ext = (char*)glGetString( GL_EXTENSIONS );
    if( strstr( ext, "EXT_framebuffer_object" ) == NULL )
    {
      printf("EXT_framebuffer_object extension was not found");
    }
    else
    {
      glIsRenderbufferEXT =
    (PFNGLISRENDERBUFFEREXTPROC)wglGetProcAddress("glIsRenderbufferEXT");
      glBindRenderbufferEXT =
    (PFNGLBINDRENDERBUFFEREXTPROC)wglGetProcAddress("glBindRenderbufferEXT"
    );
      glDeleteRenderbuffersEXT =
    (PFNGLDELETERENDERBUFFERSEXTPROC)wglGetProcAddress("glDeleteRenderbuffe
    rsEXT");
      glGenRenderbuffersEXT =
    (PFNGLGENRENDERBUFFERSEXTPROC)wglGetProcAddress("glGenRenderbuffersEXT"
    );
      glRenderbufferStorageEXT =
    (PFNGLRENDERBUFFERSTORAGEEXTPROC)wglGetProcAddress("glRenderbufferStora
    geEXT");
      glGetRenderbufferParameterivEXT =
    (PFNGLGETRENDERBUFFERPARAMETERIVEXTPROC)wglGetProcAddress("glGetRenderb
    ufferParameterivEXT");
      glIsFramebufferEXT =
    (PFNGLISFRAMEBUFFEREXTPROC)wglGetProcAddress("glIsFramebufferEXT");
```

GPUbitonicSort.cpp, continued.

```
  glBindFramebufferEXT =
(PFNGLBINDFRAMEBUFFEREXTPROC)wglGetProcAddress("glBindFramebufferEXT");
  glDeleteFramebuffersEXT =
(PFNGLDELETEFRAMEBUFFERSEXTPROC)wglGetProcAddress("glDeleteFramebuffers
EXT");
  glGenFramebuffersEXT =
(PFNGLGENFRAMEBUFFERSEXTPROC)wglGetProcAddress("glGenFramebuffersEXT");
  glCheckFramebufferStatusEXT =
(PFNGLCHECKFRAMEBUFFERSTATUSEXTPROC)wglGetProcAddress("glCheckFramebuff
erStatusEXT");
  glFramebufferTexture1DEXT =
(PFNGLFRAMEBUFFERTEXTURE1DEXTPROC)wglGetProcAddress("glFramebufferTextu
re1DEXT");
  glFramebufferTexture2DEXT =
(PFNGLFRAMEBUFFERTEXTURE2DEXTPROC)wglGetProcAddress("glFramebufferTextu
re2DEXT");
  glFramebufferTexture3DEXT =
(PFNGLFRAMEBUFFERTEXTURE3DEXTPROC)wglGetProcAddress("glFramebufferTextu
re3DEXT");
  glFramebufferRenderbufferEXT =
(PFNGLFRAMEBUFFERRENDERBUFFEREXTPROC)wglGetProcAddress("glFramebufferRe
nderbufferEXT");
  glGetFramebufferAttachmentParameterivEXT =
(PFNGLGETFRAMEBUFFERATTACHMENTPARAMETERIVEXTPROC)wglGetProcAddress("glG
etFramebufferAttachmentParameterivEXT");
  glGenerateMipmapEXT =
(PFNGLGENERATEMIPMAPEXTPROC)wglGetProcAddress("glGenerateMipmapEXT");

  if( !glIsRenderbufferEXT || !glBindRenderbufferEXT ||
!glDeleteRenderbuffersEXT ||
        !glGenRenderbuffersEXT || !glRenderbufferStorageEXT ||
!glGetRenderbufferParameterivEXT ||
        !glIsFramebufferEXT || !glBindFramebufferEXT ||
!glDeleteFramebuffersEXT ||
        !glGenFramebuffersEXT || !glCheckFramebufferStatusEXT ||
!glFramebufferTexture1DEXT ||
        !glFramebufferTexture2DEXT || !glFramebufferTexture3DEXT ||
!glFramebufferRenderbufferEXT||
        !glGetFramebufferAttachmentParameterivEXT ||
!glGenerateMipmapEXT )
  {
        printf("One or more EXT_framebuffer_object functions were not
found");
  }
}
// Create an FBO, Frame Buffer object
glGenFramebuffersEXT(1, &fbo);
// Create a color texture, null contents yet
glGenTextures(1, &color);
glBindTexture(GL_TEXTURE_2D, color);
glTexImage2D(GL_TEXTURE_2D, 0, GL_RGBA8, checkImageWidth,
                      checkImageHeight, 0, GL_RGBA, GL_UNSIGNED_BYTE,
NULL);
// FYI check of the framebuffer so far
if(debug)      printf("Check 1 : \n");checkFramebufferStatus();
// Create depth renderbuffer
glGenRenderbuffersEXT(1, &depth);
// Bind the FBO and attach the color textures to it
glBindFramebufferEXT(GL_FRAMEBUFFER_EXT, fbo);
glFramebufferTexture2DEXT(GL_FRAMEBUFFER_EXT, GL_COLOR_ATTACHMENT0_EXT,
GL_TEXTURE_2D,
                                          _jTexture, 0);
glFramebufferTexture2DEXT(GL_FRAMEBUFFER_EXT, GL_COLOR_ATTACHMENT1_EXT,
GL_TEXTURE_2D,
                                          _iTexture, 0);
// RenderBuffer generation
```

GPUbitonicSort.cpp, continued.

```
     glBindRenderbufferEXT(GL_RENDERBUFFER_EXT, depth_rb);
     glRenderbufferStorageEXT(GL_RENDERBUFFER_EXT, GL_DEPTH_COMPONENT24,
     checkImageWidth, checkImageHeight);
     // Attach renderbufferto framebuffer depth buffer
     glFramebufferRenderbufferEXT(GL_FRAMEBUFFER_EXT,
     GL_DEPTH_ATTACHMENT_EXT, GL_RENDERBUFFER_EXT, depth_rb);
     // Set the destination draw buffer to attachment 1 initially
     glDrawBuffer(GL_COLOR_ATTACHMENT1_EXT);
     // FYI check of the framebuffer so far
     if(debug)      printf("Check 1b : \n");checkFramebufferStatus();

     // Create the fragment shader
    _fragmentProgram = cgCreateProgramFromFile(g_cgContext, CG_SOURCE,
                                       "fragmentShader.cg", g_cgProfile,
                                       "edges", NULL);

    // Create the texture parameter for the fragment program
     // Since this is the inital setup, just load the Cg program - do not
     enable it until ready
    if(_fragmentProgram != NULL)
    {
        cgGLLoadProgram(_fragmentProgram);
      cgGLBindProgram(_fragmentProgram);
     //      cgGLEnableProfile(g_cgProfile); // don't enable until ready to
     process
        _textureParam = cgGetNamedParameter(_fragmentProgram, "texture");
      outDataTextureParam = cgGetNamedParameter(_fragmentProgram,
     "outDataTexture");
     }


     // Initially load the input data into the texture
     // by drawing a single quad with the texture as the destination
     glBindTexture(GL_TEXTURE_2D, _jTexture);
     cgGLSetTextureParameter(_textureParam, _jTexture);
     cgGLEnableTextureParameter(_textureParam);
     glEnable(GL_TEXTURE_2D);
     //      use OpenGL to draw the quad
     glBegin(GL_QUADS);
     {
       glTexCoord2f(0, 0); glVertex3f(-1, -1, -0.5f);
       glTexCoord2f(1, 0); glVertex3f( 1, -1, -0.5f);
       glTexCoord2f(1, 1); glVertex3f( 1,  1, -0.5f);
       glTexCoord2f(0, 1); glVertex3f(-1,  1, -0.5f);
     }
     glEnd();
     cgGLDisableTextureParameter(_textureParam);

     // setting this so the toggle at the beginning of updatePart2 will
     maintain the current value
     toggleTex=0;

}    // End of initialise(void)
//--------------------------------------------------------------------------
```

GPUbitonicSort.cpp, continued.

```
/**
 * updatePart2
 *
 * Sets the input to the Cg shader and enacts a drawing to the screen
 * so that a single sorting pass is completed by the Cg shader.
 * This function is called repeatedly by GLUT.
 *
 * @param       void
 * @return      void
 *
 * @author      Jason Ruane
 * @version     1.0
 */
void updatePart2()
{
     counter++;
     myTemp[0]=counter;
     // Running variables for the Cg program
     myPass=myPass-1;
     if (myPass == 0)
     {
       myLoop = myLoop+1;
       myPass = myLoop;
     }

     // Check to see if the sorting operation has completed
     if (pow(2,myLoop) > checkImageWidth*checkImageHeight)
     {
       testsToRun--;
       testsRan++;
       // Inform user of completion status
       printf("Finished loop %d of %d.\n",testsRan,testsRan+testsToRun);
       // Finished sorting all the input data, so flush operations on the
     GPU
       glFinish();

       // This is where the program will exit, it has completed sorting
       if(testsToRun==0)
       {
              // Track the timing required
              endTime = clock();
              timeForUpdate = clock()-gpuStart;
              if(debug)     printf("Finished at %f\n",(clock()-
     startTime)/CLOCKS_PER_SEC);
              // Save the results to file
              saveFile();
              // Exit the program, return to the OS
              exit(0);
       }
       else
       {
              // Sorting has not completed, update the bitonic pass
     variables
              counter=0;
              myLoop=1;
              myPass=1;
              // FYI tracking
              if(debug)     printf("done at %f\n", (clock()-
     startTime)/CLOCKS_PER_SEC);
       }
     }


     // By toggling the textures, only 2 are required in total,
     // 1 to read from and 1 to write to
     // actualTexture is the one to read from,
```

GPUbitonicSort.cpp, continued.

```
    // framebuffertexture/glDrawBuffer is the one to write to
    if (toggleTex==0)
    {
      toggleTex=1;readTex=0;
      actualTexture=_iTexture;
      latestTextureUpdated=_jTexture;
      glDrawBuffer(GL_COLOR_ATTACHMENT0_EXT);
      destinationAttachment=GL_COLOR_ATTACHMENT0_EXT;
    }
    else
    {
      toggleTex=0;
      readTex=1;
      actualTexture= _jTexture;
      latestTextureUpdated=_iTexture;
      glDrawBuffer(GL_COLOR_ATTACHMENT1_EXT);
      destinationAttachment=GL_COLOR_ATTACHMENT1_EXT;
    }
    // set the destination for drawing to
    glDrawBuffer(destinationAttachment);

    // set the uniform parameter for passing to the Cg Shader
    myTemp2[0]=pow(2,(myPass-1));
    myTemp2[1]=checkImageWidth;
    myTemp2[2]=floor(pow(2,myLoop));
    myTemp2[3]=floor(pow(2,myPass));

    // FYI of progress
    if(debug)
    {
    printf("       Bitonic Step: %d   Stage: %d\n",myLoop,myPass);
    printf("       Setting Offset: %f\n",myTemp2[0]);
    printf("       Setting pbufwidth: %f\n",myTemp2[1]);
    printf("       Setting stepno: %f\n",myTemp2[2]);
    printf("       Setting stageno: %f\n",myTemp2[3]);
    }

    // edit the uniform parameter for the fragment program
    // set the first item to toggle
    CGparameter myCParameter = cgGetNamedParameter( _fragmentProgram,
    "myCgParameter");
    cgGLSetParameter4fv(myCParameter, myTemp );
    CGparameter myCDistance = cgGetNamedParameter( _fragmentProgram,
    "myDistance");
    cgGLSetParameter4fv(myCDistance, myTemp2 );

    // bind the Cg program in preparation
    cgGLBindProgram(_fragmentProgram);

    // bind the alternating texture as input (the other will be output
    glBindTexture(GL_TEXTURE_2D, actualTexture);
   cgGLSetTextureParameter(_textureParam, actualTexture);
    cgGLEnableTextureParameter(_textureParam);

    // Set the OpoenGL drawing mode
    glEnable(GL_TEXTURE_2D);
   glPolygonMode(GL_FRONT,GL_FILL);

    // Actually enable the Cg Shader
    cgGLEnableProfile(g_cgProfile);
    // Draw the window sized quad
   glBegin(GL_QUADS);
   {
       glTexCoord2f(0, 0); glVertex3f(-1, -1, -0.5f);
       glTexCoord2f(1, 0); glVertex3f( 1, -1, -0.5f);
       glTexCoord2f(1, 1); glVertex3f( 1,  1, -0.5f);
```

GPUbitonicSort.cpp, continued.

```
       glTexCoord2f(0, 1); glVertex3f(-1,  1, -0.5f);
    }
    glEnd();

    // Disable the Cg shader
    cgGLDisableProfile(g_cgProfile);

     // Disable the texture
     cgGLDisableTextureParameter(_textureParam);
     glDisable(GL_TEXTURE_2D);

     // Present the visualisation of the data
     // if the debug setting is set to 1
     if (debug) display();

}  // end of updatePart2(void)
//--------------------------------------------------------------------------



/**
 * display
 *
 * Draws onscreen the data which is being currently rearranged by
 * the Bitonic sorting activities.
 * Only called if debug setting is set to 1 as it requires relatively
 * substantial time overhead
 *
 * @param       void
 * @return      void
 *
 * @author      Jason Ruane
 * @version     1.0
 */
void display()
{
     // Set the destination to the screen window
     glBindFramebufferEXT(GL_FRAMEBUFFER_EXT, 0);
    // Bind the latest texture which was updated
    glBindTexture(GL_TEXTURE_2D, latestTextureUpdated);
    glEnable(GL_TEXTURE_2D);
     // FYI information
     if(debug)      printf("in display, frame %.0f\n", counter);

     // Render a window sized quad
     glBegin(GL_QUADS);
    {
       glTexCoord2f(0, 0); glVertex3f(-1, -1, -0.5f);
        glTexCoord2f(1, 0); glVertex3f( 1, -1, -0.5f);
        glTexCoord2f(1, 1); glVertex3f( 1,  1, -0.5f);
        glTexCoord2f(0, 1); glVertex3f(-1,  1, -0.5f);
    }
     glEnd();
     glutSwapBuffers();  // Is a time-expensive operation
                                 // but it is required to view the effects

     // Restore the previous off-screen operations
    glDisable(GL_TEXTURE_2D);
     glBindFramebufferEXT(GL_FRAMEBUFFER_EXT, fbo);

} // end of display(void)
//--------------------------------------------------------------------------
```

GPUbitonicSort.cpp, continued.

```
/**
 * saveFile
 *
 * Write the sorted array of data to file.
 * Also write to the results file, the performance metrics
 * such as the time required to sort the data.
 *
 *
 * @param       void
 * @return      void
 *
 * @author      Jason Ruane
 * @version     1.0
 */
void saveFile(void)
{
     // Variables and file handle required
     int p,q,i,j=0;
     FILE *j_file;

     // Ensure all activities pending on the GPU are completed
     glFinish();

     // Record the timing for saving this file
     double fileStart = clock();

     // Copy the texture to a data array
     // Note: It will arrive in a 90 degree rotation
     glReadBuffer(destinationAttachment);
     glReadPixels(0, 0, checkImageWidth,
     checkImageHeight,GL_RGBA,GL_UNSIGNED_BYTE,data);
     glFinish();

     // Note the timing for transferring the data from GPU to CPU
     double fileTransfer = clock();

     // File saving: open the file of sorted data
     if( (j_file = fopen(outputFilename, "w"))==NULL) return ;
     // Save the data
     for (i=0; i < checkImageWidth*checkImageHeight; i++)
     {
       q=floor(i/checkImageHeight);
       p=i%(int)checkImageWidth;
       // Note: writing only only the Blue component as this is the colour
     plane in use
       fprintf(j_file,"%d\n",data[p][q][2]);
     }
    fclose(j_file); // Close the file stream

     // note the timings
     float fileEnd = clock();
     float total = (endTime-startTime);
     // Ouput to user
     printf("Sorting time: %.4f seconds per
     iteration\n",timeForUpdate/CLOCKS_PER_SEC/testsRan);

     // Capture the performance metrics
     //
     // Write to the results file, in appending fashion
     if( (j_file = fopen(resultsFilename, "a+"))==NULL) return ;
     fprintf(j_file,"\n\nGPU Sort ---------------------------------------
     \n");
     fprintf(j_file,"Pixel data transfer time %.4f \nFile access time
     %.4f\n",
       (fileTransfer-fileStart)/CLOCKS_PER_SEC, (fileEnd-
     fileTransfer)/CLOCKS_PER_SEC );
```

GPUbitonicSort.cpp, continued.

```
     fprintf(j_file,"Counter: %.0f frames with %d loops in total time %.3f
     seconds (fps:%.1f)\n",
       counter,testsRan, total/CLOCKS_PER_SEC,
       (float)counter*(float)testsRan/((float)total/(float)CLOCKS_PER_SEC)
     );
     float sortingTime = timeForUpdate/CLOCKS_PER_SEC/testsRan;
     fprintf(j_file,"Sorting time: %.4f seconds per
     iteration\n",sortingTime);
     fprintf(j_file,"%.0f image size with %.0f
     items\n",checkImageWidth,checkImageWidth*checkImageHeight);
     fprintf(j_file,"Items per
     second:%.0f\n",checkImageWidth*checkImageHeight/sortingTime);
     fprintf(j_file,"Debug value:%d\n",debug);
     fclose(j_file); // Close the file stream

}    // end of saveFile(void)
//--------------------------------------------------------------------------



/**
 * idle
 *
 * Glut will call this function each time it believes the GPU is ready
 * to receive more instructions.
 * This function will allow the CPU to rest, and/or run another
 * iteration of the Bitonic sort by calling updatePart2
 *
 *
 * @param       void
 * @return      void
 *
 * @author      Jason Ruane
 * @version     1.0
 */
void idle(void)
{

     // FYI information
     if(debug)      printf("starting idle. counter=%d\n",counter);
     // Allow the CPU to rest, if the timer is set to a positive value
     Sleep(SleepTime);
     // Run the next iteration of Bitonic sort routine
     updatePart2();

}    // end of idle(void)
//--------------------------------------------------------------------------



/**
 * reshape
 *
 * Glut will call this function each time it notices a resize of the
 * drawing window.
 *
 *
 * @param       w   Width of the window, in pixels
 * @param       h   Height of the window, in pixels
 * @return      void
 *
 * @author      Jason Ruane
 * @version     1.0
 */
void reshape(int w, int h)
{
```

GPUbitonicSort.cpp, continued.

```
     // Avoid division by zero
    if (h == 0) h = 1;

     // Set the viewport to the new size
    glViewport(0, 0, w, h);
     // Restore the unit sized window and mode for drawing
    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();
    gluOrtho2D(-1, 1, -1, 1);
    glMatrixMode(GL_MODELVIEW);
    glLoadIdentity();

}    // end of reshape(int w, int h)
//--------------------------------------------------------------------------



/**
 * cgErrorCallback
 *
 * Standard Cg function to facilitate the reporting of errors
 * experienced in the Cg environment to the user.
 * Only called when Cg experiences an error.
 *
 *
 * @param       void
 * @return      void
 *
 * @author      Jason Ruane
 * @version     1.0
 */
void cgErrorCallback(void)
{
     // Capture the error
     CGerror lastError = cgGetError();
     // If there is an actual error
     if(lastError)
     {
       // Report the string to the user
       printf("%s\n\n", cgGetErrorString(lastError));
       printf("%s\n", cgGetLastListing(g_cgContext));
       printf("Cg error!\n");
     }

}    // end of cgErrorCallback(void)
//--------------------------------------------------------------------------



/**
 * main
 *
 * Standard C main function - Entry point for program
 *
 *
 * @param       argc       Number of arguments passed
 * @param       argv       Items of arguments passed
 * @return      int        0 if successful, non-zero if not.
 *
 * @author      Jason Ruane
 * @version     1.0
 */
int main(int argc, char **argv)
{

     // Inform user program is starting
```

GPUbitonicSort.cpp, continued.

```
    printf("Starting GPU sorting.\n");

    // If sufficient arguments have been passed
    if(argc>3)
    {
      // Assign the image array size
      checkImageWidth=atoi(argv[2]);
      checkImageHeight=checkImageWidth;

      // Assign the number of tests to run
      testsToRun=atoi(argv[3]);

      // Set the debug flag on if required
      if(argc>4)
      {
             if(atoi(argv[4])==1)
             {
                    printf("Debug On\n");
                    debug=1;
             }
      }

      // Assign the filename for input data
      strcpy(inputFile1,argv[1]);
    }
    else
    {
      // Inform user incorrect values supplied
      printf("Please specify an input file and a value for debug\n");
      // Return to the OS
      exit(0);
    }

    // FYI information
    if(debug)      printf("loading input file\n");
    loadFile();
    glFinish();
    // Set timing variable
    startTime = clock();

    // Set up GLUT system
    // RGBA display mode required
    glutInitDisplayMode(GLUT_DOUBLE | GLUT_RGBA);
    // Window size in pixels
    glutInitWindowSize(checkImageWidth, checkImageWidth);
    // Initial position of window
    glutInitWindowPosition(100, 100);
    // Create the display window
    glutCreateWindow("GPUcompute");
    // Set the funtion to call when the GPU is ready
    glutIdleFunc(idle);
    // How to update the GLUT window
    glutDisplayFunc(display);
    // If the GLUT window is resized, what to run
    glutReshapeFunc(reshape);

    // Set up system and load texture
    initialise();
    // Ensure GPU actions are finished
    glFinish();
    // Inform user
    printf("Starting at %f\n",(clock()-startTime)/CLOCKS_PER_SEC);
    // Timing variable
    gpuStart = clock();
```

GPUbitonicSort.cpp, continued.

```
     // Enter the GLUT system.
     // GLUT does not return, so the exiting of the entire program
     // will be enacted from within the updatePart2 function, wherein
     // exit(0) can be found
     glutMainLoop();

     return(0); // Never actually reached, but left for completeness

}    // end of main(int argc, char **argv)
//--------------------------------------------------------------------------



/**
 * loadFile
 *
 * Load the input data (unsorted integers)
 * and store it into the checkImage array
 *
 *
 * @param       void
 * @return      void
 *
 * @author      Jason Ruane
 * @version     1.0
 */
void loadFile(void)
{
     // Local variables and file handle required
     int p, q, field1, i=0;
     FILE *j_file;

     // Open the file for reading
     if( (j_file = fopen(inputFile1, "r"))==NULL) return ;
     // Read the data
    // format is 1 integer [0-255] per line
     while( fscanf( j_file , "%i\n" , &field1 ) != EOF )
     {
       // the 1 Dimension variable i, converted the 2-D
         q=floor(i/checkImageHeight);
       p=i%(int)checkImageWidth;
       checkImage[p][q][2] = 0;
       checkImage[p][q][1] = 0;
       checkImage[p][q][0] = field1;
       checkImage[p][q][3] = 255;
       i++;
       // if the data file is too large, ignore extra items
       if(i>=checkImageWidth*checkImageHeight) { break; }
     }
    fclose(j_file); // Close the file stream
}    // end of loadFile(void)
//--------------------------------------------------------------------------



/**
 * checkFramebufferStatus
 *
 * Standard OpenGL function to check the status of the framebuffer
 *
 *
 * @param       void
 * @return      bool       0=Success, 1=Fail
 *
 * @author      Jason Ruane, as per official OpenGL definitions
```

GPUbitonicSort.cpp, continued.

```
 * @see                    http://oss.sgi.com/projects/ogl-
     sample/registry/EXT/framebuffer_object.txt
 * @version     1.0
 */
bool checkFramebufferStatus(void)
{
     GLenum status;
    status=(GLenum)glCheckFramebufferStatusEXT(GL_FRAMEBUFFER_EXT);
    switch(status) {
        case GL_FRAMEBUFFER_COMPLETE_EXT:
            return true;
        case GL_FRAMEBUFFER_INCOMPLETE_ATTACHMENT_EXT:
            printf("Framebuffer incomplete,incomplete attachment\n");
            return false;
        case GL_FRAMEBUFFER_UNSUPPORTED_EXT:
            printf("Unsupported framebuffer format\n");
            return false;
        case GL_FRAMEBUFFER_INCOMPLETE_MISSING_ATTACHMENT_EXT:
            printf("Framebuffer incomplete,missing attachment\n");
            return false;
        case GL_FRAMEBUFFER_INCOMPLETE_DIMENSIONS_EXT:
            printf("Framebuffer incomplete,attached images must have same
     dimensions\n");
            return false;
        case GL_FRAMEBUFFER_INCOMPLETE_FORMATS_EXT:
             printf("Framebuffer incomplete,attached images must have same
     format\n");
            return false;
        case GL_FRAMEBUFFER_INCOMPLETE_DRAW_BUFFER_EXT:
            printf("Framebuffer incomplete,missing draw buffer\n");
            return false;
        case GL_FRAMEBUFFER_INCOMPLETE_READ_BUFFER_EXT:
            printf("Framebuffer incomplete,missing read buffer\n");
            return false;
    }
     return false;
}// end of checkFramebufferStatus(void)
//------------------------------------------------------------------------
```

7.3.2 Appendix 3.2 GPU Sorting: fragmentShader.cg

The following code is from fragmentShader.cg. It is the Cg code to accompany the bitonic sorting program for the GPU.

```
// Bitonic sorting fragment shader
//
half4 edges(float2 coords : TEX0,  //:  TEX0 for clamped version of
     coordinates, WPOS for integer
            uniform sampler2D texture,
            uniform half4 myDistance,
            uniform float4 myCgParameter) : COLOR
{

// Retrieve the variables from the uniform parameters
float offset  = myDistance[0];
float imageWidth= myDistance[1];
float stepno  = myDistance[2];
float stageno       = myDistance[3];

// Calculate the 2D and 1D equivalent locations
// for the current location
// The floor is required to start at zero and reach 255
int current2dx = floor(coords.y*imageWidth );
int current2dy = floor(coords.x*imageWidth );
int current1d  = floor((current2dy * imageWidth) + current2dx);


// would like to do:
// half csign = ((current1d % int(stageno)) < offset) ?  1 : -1;
// But performance is 2X for the following method:
float temp  = frac(current1d /stageno);
float temp2 = temp*stageno;
half  csign = (temp2 < offset) ?  1 : -1;

// half cdir = (int(current1d/stepno) % 2 == 0) ?  1 : -1;
// But performance is 2X for the following method:
temp     = floor(current1d/stepno);
temp2    = frac(temp/2);
half cdir = ( temp2 == 0) ?  1 : -1;


// Calculate the 2D and 1D equivalent locations
// for the other, matching location
float other1d = current1d + (csign * offset);
float other2dreal = other1d/imageWidth;
float other2dx = (frac(other2dreal) * imageWidth);
float other2dy = float(floor(other2dreal));

// Find the value in the textures for the 2D locations found above
half4
     val0=tex2D(texture,half2((current2dy/imageWidth),(current2dx/imageWidth
     )) );
half4 val1=tex2D(texture,half2((other2dy/imageWidth),(other2dx/imageWidth))
     );

// Find the current minimum and maximum
float jmin = min( val0.z , val1.z );
float jmax = max( val0.z , val1.z );

// Return either the minumum or maximum depending on the sign and direction
return( csign == cdir) ? half4(0,0,jmin,1) : half4(0,0,jmax,1);

}
```

7.3.3 Appendix 3.3 GPU Sorting: vertexShader.cg

The following code is from vertexShader.cg. It is used by GPUbitonicSort.exe and is designed to pass the vertex values onto the rest of the GPU pipeline without interference.

```
// Standard Vertex shader
// required to just pass values through
//
struct vert_Output {
  float4 position : POSITION;
  float4 color    : COLOR;
};

vert_Output vertexMain(float4 position : POSITION ,
              float4 color : COLOR,
              uniform float4 Kd)
{
  vert_Output OUT;

  OUT.position = position;
  OUT.color = Kd;

  return OUT;
}
```

7.3.4 Appendix 3.4 CPU Sorting: CPUbitonicSort.cpp

The following code is from CPUbitonicSort.cpp. It compiles to become CPUbitonicSort.exe, the CPU implementation for sorting task.

```
/**
 * Bitonic Sorting on the CPU.
 *
 * This program enacts the Bitonic sorting alorithm on a CPU using a
 * similar setup to the GPU based implementation.
 * This program also uses the standard C Qsort and C++ std::sort
 * for comparison of performance.
 *
 * @param       inputFile  The location of the input file of data
 * @return      A successful exit(0) signal.
 *
 * @author      Jason Ruane, DIT Bolton St. Dublin, Ireland. B773.2006.
 * @version     1.0
 */

// Preprocessor directives
#define checkImageWidth 256
#define checkImageHeight 256
#define debug 0
#define SleepTime 0

// Include files required
#include <stdlib.h>
#include <stdio.h>
#include <time.h>
#include <windows.h>
#include <winbase.h>
#include <assert.h>
#include <math.h>
#include <algorithm>

// For the C++ standard sorting routine
using std::sort;

// Function declarations
void   loadFile(void);
void   saveFile(void);
void   initialise(void);
void   idle(void);
void   updatePart2(void);
void   display(void);
int          compare (const void * a, const void * b);

// global variables employed throughout
int testsToRun=1;

float myTemp2[4];
float myTemp[4];
int myPass, myLoop=1;
float counter=0;    // count the frames rendered
double start,end;   // used for timing the frames
double timeForUpdate=0;

int testsRan=0, maxCounterRuns = 1000;
int toggleTex=0,otherTex=1;
int oneDarray[checkImageWidth*checkImageHeight];
double timeQsort,timeQsortStart, tempStart;

// data array for saving the texture to
int checkImage[2][checkImageHeight][checkImageWidth][4];
```

CPUbitonicSort.cpp, continued.

```
int data[checkImageWidth][checkImageHeight][4];
int outData[checkImageWidth][checkImageHeight][4];

// Filename for usage
char inputFile1[255];                          // Input file which will be
     used


/**
 * compare
 *
 * Used by the standard C sorting routine to compare items.
 *
 * @param       a   First item to compare
 * @param       a   Second item to compare
 * @return      int Number to indicate which of the inputs is larger
 *                      Non-negative if a>=b or negative otherwise
 *
 * @author      Jason Ruane
 * @version     1.0
 */
int compare (const void * a, const void * b)
{
     return ( *(int*)a - *(int*)b );
}    // End of compare
//---------------------------------------------------------------------


/**
 * updatePart2
 *
 * Enact a single pass of the bitonic sort
 *
 * @param       void
 * @return      void
 *
 * @author      Jason Ruane
 * @version     1.0
 */
void updatePart2(void)
{
     // Increment the counter of passes
     counter++;
     myTemp[0]=counter;

     // Running variables RE: pass and loop/stage
     myPass=myPass-1;
     if (myPass == 0)
     {
       myLoop = myLoop+1;
       myPass = myLoop;
     }

     // Check to see if the sorting operation has completed
     if (pow(2,myLoop) > checkImageWidth*checkImageHeight)
     {
       testsToRun--;
       if(testsToRun==0)
       {
              // Note the time taken
              end = clock();
              double total = (end-start);
              // Inform user
              printf("finishing: Counter: %.0f by %d in Time %.3f ,
     fps:%.1f\n",
```

CPUbitonicSort.cpp, continued.

```
  counter,testsToRun,total/CLOCKS_PER_SEC,counter/(total/CLOCKS_PER_SEC
));
        timeForUpdate += clock()-tempStart;
        // Save the output (sorted data) to file
        saveFile();
        // Terminate the program
        exit(0);
  }
  else
  {
        // Progress to the next phase, setting variables
        counter=0;
        myLoop=1;
        myPass=1;
  }
} // End of check if sorting operation has completed

// Note the pass/stage/loop information
myTemp2[0]=pow(2,(myPass-1));
myTemp2[1]=float(checkImageWidth);//pow(2,myLoop);
myTemp2[2]=floor(pow(2,myLoop));
myTemp2[3]=floor(pow(2,myPass));

// FYI the phase information
if(debug)
{
printf("       Bitonic Step: %d   Stage: %d\n",myLoop,myPass);
printf("       Setting Offset: %f\n",myTemp2[0]);
printf("       Setting pbufwidth: %f\n",myTemp2[1]);
printf("       Setting stepno: %f\n",myTemp2[2]);
printf("       Setting stageno: %f\n",myTemp2[3]);
}

/* CPU Sorting in a GPU simulation fashion */
for(int pixelY=0;pixelY<checkImageHeight;pixelY++)
{
  for(int pixelX=0;pixelX<checkImageWidth;pixelX++)
  {
        // This section is similar to the fragment shader code
        float imageWidth = myTemp2[1];
        float offset = myTemp2[0];
        float pbufwidth = myTemp2[1]-1.0;
        float stepno = myTemp2[2];
        float stageno = myTemp2[3];

        // Find the 2D and 1 D location for the current element
        // The floor is required to make it start at zero and reach
255
        int current2dx = floor(pixelX);
        int current2dy = floor(pixelY);
        int current1d = current2dy * imageWidth + current2dx;

        // Find the current sign and direction
        float csign = ((current1d % int(stageno)) < offset) ?  1 : -1;
        float cdir = (int(current1d/stepno) % 2 == 0) ?  1 : -1;

        // Find the 2D and 1D location for the corresponding compare
element
        int other1d = current1d + (csign * offset);
        float other2dreal = float(other1d)/float(imageWidth);
        int other2dx = ((other2dreal-int(other2dreal)) * imageWidth);
        int other2dy = float(floor(other2dreal));
```

CPUbitonicSort.cpp, continued.

```
            // Find the values of the elements
            int val1=checkImage[toggleTex][other2dx][other2dy][2];
            int val0=checkImage[toggleTex][current2dx][current2dy][2];

            // Find the min and max of each element
            int cmin = min(val0,val1);
            int cmax = max(val0,val1);
            //float cmax = ( val0 > val1 ) ? val0 : val1;

            // Return either min or max depending on sign and direction
            int returnValue = ( csign == cdir) ? cmin : cmax;
            checkImage[otherTex][pixelX][pixelY][2] = returnValue;

      } // End of pixelX loop
    } // End of PixelY loop

    // Set the toggling values (similar to textures), for next run
    if(toggleTex==0)
    { toggleTex=1;otherTex=0;}
    else
    { toggleTex=0;otherTex=1;}

}  // end of updatePart2
//--------------------------------------------------------------------



/**
 * display
 *
 * Inform the user of the current "frame" in use
 * Essentially reports the bitonic sort phase
 *
 * @param       void
 * @return      void
 *
 * @author      Jason Ruane
 * @version     1.0
 */
void display()
{
    if(debug)      printf("in display, frame %.0f\n", counter);
}    // end of display
//--------------------------------------------------------------------



/**
 * saveFile
 *
 * Write the cpu bitonic sorted array of data to file.
 * Also save the Qsorted array to file
 *
 * @param       void
 * @return      void
 *
 * @author      Jason Ruane
 * @version     1.0
 */
void saveFile(void)
{
    int p,q,i, j=0;
    FILE *j_file;
    double fileStart = clock();
```

CPUbitonicSort.cpp, continued.

```
     // Copy the data array to another array (CPU to CPU data copies)
     for (i=0; i < checkImageWidth*checkImageHeight; i++)
     {
       p=i%checkImageWidth;  // col
          q=i/checkImageWidth;  // row
       data[p][q][0]=0;
       data[p][q][1]=0;
       data[p][q][2]=checkImage[toggleTex][p][q][2];
       data[p][q][3]=0;
     }

     // Note the timing
     double fileTransfer = clock();

     // Open the file for writing
     if( (j_file = fopen("Data\\output-bitonic.txt", "w"))==NULL) return ;

     // print the data to file
     for (i=0; i < checkImageWidth*checkImageHeight; i++)
     {
       q=floor(i/checkImageHeight);
       p=i%checkImageWidth;
       fprintf(j_file,"%d\n",data[p][q][2]);
     }
    fclose(j_file); // Close the file stream

     // Note the timing
     double fileEnd = clock();

     // the QSORT() version
     if( (j_file = fopen("Data\\output-qsort.txt", "w"))==NULL) return ;
     for (i=0; i < checkImageWidth*checkImageHeight; i++)
     {
       // oneDarray contains the Qsorted version
       fprintf(j_file,"%d\n",oneDarray[i]);
     }
    fclose(j_file); // Close the file stream

}    // end of saveFile
//-------------------------------------------------------------------



/**
 * idle
 *
 * Run the next phase of the bitonic sort
 *
 * @param       void
 * @return      void
 *
 * @author      Jason Ruane
 * @version     1.0
 */
void idle(void)
{
     // Allow the CPU to rest, if set in SleepTime
     Sleep(SleepTime);

     // Run the next phase
     updatePart2();

}    // end of idle
//-------------------------------------------------------------------
```

CPUbitonicSort.cpp, continued.

```
/**
 * initialise
 *
 * Set up variables required for the phase definition of bitonic sort
 *
 * @param       void
 * @return      void
 *
 * @author      Jason Ruane
 * @version     1.0
 */
void initialise()
{

     // Initial phase information
     myTemp[0]=0;
     myLoop=1;
     myPass=2; // 1 greater than expected 1 because of pre decrementing
     toggleTex=0;

     // Store in vector
     myTemp2[0]=pow(2,(myPass-1));
     myTemp2[1]=checkImageWidth;//pow(2,myLoop);
     myTemp2[2]=floor(pow(2,myLoop));
     myTemp2[3]=floor(pow(2,myPass));

     // FYI information
     if(debug)
     {
     printf("       Stage: %d   Step: %d\n",myLoop,myPass);
     printf("       Setting Offset: %f\n",myTemp2[0]);
     printf("       Setting pbufwidth: %f\n",myTemp2[1]);
     printf("       Setting stageno: %f\n",myTemp2[2]);
     printf("       Setting stepno: %f\n",myTemp2[3]);
     printf("After initialise: counter: %.0f  Pass: %d 2Stage
     %d\n",counter,myPass,myLoop);
     }

}    // end of initialise
//--------------------------------------------------------------------



/**
 * main
 *
 * Standard C main function - Entry point for program
 *
 *
 * @param       argc       Number of Command line arguments
 * @param       argv       Command line arguments
 * @return      int        0 if successful, non-zero if not.
 *
 * @author      Jason Ruane
 * @version     1.0
 */
int main(int argc, char **argv)
{

     // If sufficient arguments have been passed
     if(argc>1)
     {
       // Assign the filename for input data
       strcpy(inputFile1,argv[1]);
     }
```

CPUbitonicSort.cpp, continued.

```
// std::sort method
     // Load the input file
     loadFile();
     // Note the timing
     double timeSTDsortStart=clock();
     // Enact the sorting
     sort(oneDarray,oneDarray+( checkImageWidth*checkImageHeight ));
     // Note the finish time
     double timeSTDsort=(clock()-timeSTDsortStart)/CLOCKS_PER_SEC;
     // Output timing result to user
     printf("STDsorted in %f seconds\n",timeSTDsort);


// Qsort method
     // Load the input file
     loadFile();
     // Note the timing
     timeQsortStart=clock();
     // Enact the sorting
     qsort (oneDarray, float(checkImageWidth)*float(checkImageHeight),
     sizeof(int), compare);
     // Note the finish time
    timeQsort=(clock()-timeQsortStart)/CLOCKS_PER_SEC;
     // Output timing result to user
     printf("Qsorted in %f seconds\n",timeQsort);


// CPU bitonic sort in GPU fashion
     // Set up initial bitonic phase variables
    initialise();
     // Note the timing
     start = clock();
     tempStart = clock();
     // Enter the bitonic sorting loop
     // This will not exit, the program termination occurs inside
     updatepart2()
     // when it deems the sorting is complete
     do{
       idle();       // repeatedly run another phase
     }while(1);

return(0); // Never actually reached, but left for completeness

}    // end of main
//--------------------------------------------------------------------



/**
 * loadFile
 *
 * Load the input file. A list of single random integers [0->255]
 *
 *
 * @param       void
 * @return      void
 *
 * @author      Jason Ruane
 * @version     1.0
 */
void loadFile(void)
{
     int p,q,i=0, j=0, count255=0;
     FILE *j_file;
     int tempInput;
```

CPUbitonicSort.cpp, continued.

```
     // Open the input file
     if( (j_file = fopen(inputFile1, "r"))==NULL)
     {
       // Inform user
       printf("Input file not found\n");
       // Terminate program
       exit(0);
     }

    // format is 1 integer [0-255] per line
     while( fscanf( j_file , "%d\n" , &tempInput) != EOF )
     {
       q=floor(i/checkImageHeight);
       p=i%checkImageWidth;
       // checkImage array used by CPU bitonic sort
       checkImage[toggleTex][p][q][0] = tempInput;
       checkImage[toggleTex][p][q][1] = tempInput;
       checkImage[toggleTex][p][q][2] = tempInput;
       checkImage[toggleTex][p][q][3] = tempInput;
       // oneDarray used by std::sort and qsort
       oneDarray[i]=tempInput;
       i++;
       // if the data is too large, ignore extra items
       if(i>=checkImageWidth*checkImageWidth) { break; }
   }
    fclose(j_file); // Close the file stream

}    // end of loadFile
//--------------------------------------------------------------------------
```

7.4 Appendix 4. Source Code: Motion Estimation

7.4.1 Appendix 4.1 runTest.bat

The following code is from runTest.bat, the batch file used to initiate the GpuCpuVideo.exe program which executes the motion vector program using all of the input images from each of the three video sequences.

```
echo OFF
REM
REM This batch file is designed to run the GPU-CPU program mutiple times to
     record
REM performance metrics

set loops=5
set program=GpuCpuVideo.exe
set resultsFile=Data\Results-File.txt

@echo Will run the test using all inputs, %loops% times.

:STAGE1
set counter=0
set counterTwo=1
:STAGE1A
%program% Data\FOR\for0%counter%.bmp Data\FOR\for0%counterTwo%.bmp
     %resultsFile% %loops%
set /a counter=%counter%+1
set /a counterTwo=%counterTwo%+1
if %counterTwo% == 26 GOTO STAGE2
GOTO STAGE1A

:STAGE2
set counter=0
set counterTwo=1
:STAGE2A
%program% Data\MAD\mad0%counter%.bmp Data\MAD\mad0%counterTwo%.bmp
     %resultsFile% %loops%
set /a counter=%counter%+1
set /a counterTwo=%counterTwo%+1
if %counterTwo% == 26 GOTO STAGE3
GOTO STAGE2A

:STAGE3
set counter=0
set counterTwo=1
:STAGE3A
%program% Data\COA\coa0%counter%.bmp Data\COA\coa0%counterTwo%.bmp
     %resultsFile% %loops%
set /a counter=%counter%+1
set /a counterTwo=%counterTwo%+1
if %counterTwo% == 26 GOTO STAGE4
GOTO STAGE3A

:STAGE4
pause
```

7.4.2 Appendix 4.2 GpuCpuVideo.cpp

The following code is from GpuCpuVideo.cpp When compiled, it creates GpuCpuVideo.exe which is the program used to find motion vectors between successive images in video sequences, using both the GPU and CPU.

```
/**
 * GPU-CPU Motion Vectors
 *
 * This program is designed to use the Graphics Processor Unit and
 * Central Processing Unit to demonstrate the ability to perform
 * Exhaustive Block Matching between two video frames to aid the
 * process of motion estimation.
 * The output files conatin the motion vectors as calcualted by
 * the GPU method and CPU method.
 *
 * @param  imageFile1      Location of the first input image (256*256 bmp)
 * @param  imageFile2      Location of second input image (256*256 bmp)
 * @param  resultsFile     Location of output file to write results to
 * @param  testsToRun      number of times to run each method
 * @return      void
 *
 * @author      Jason Ruane, DIT Bolton St. Dublin, Ireland. B773.2006.
 * @version     1.0
 */

// Preprocessor directives
// Cg and OpenGL libraries
#ifdef _MSC_VER
#pragma comment( lib, "cg/lib/cg.lib" )
#pragma comment( lib, "cg/lib/cgGL.lib" )
#endif
#pragma comment (lib, "opengl32.lib")

// Standard includes required
#include <stdlib.h>
#include <stdio.h>
#include <time.h>
#include <windows.h>
#include <winbase.h>
#include <assert.h>
#include <math.h>

// Cg requirements
#include "cg/include/Cg/cg.h"                       // Cg runtime API
#include "cg/include/Cg/cgGL.h"                     // OpenGL-specific Cg
     runtime API
#include "cg/include/GL/glext.h"                    // local header file
#include "cg/include/GL/glut.h"                     // Glut and OpenGL api

// Definitions to be refererenced throughout
#define checkImageWidth 256      // The width dimension of the input images
#define checkImageHeight 256     // The height dimension of the input
     images
#define debug 0                  // Set to 0 for Debug off, 1 for Debug On
#define SleepTime 1              // How many milliseconds to wait between
     iterations
#define maxString 255            // The maximum characters in command line
     parameters
#define reducedInfoSize 32       // Equal to checkImageWidth/scale
```

GpuCpuVideo.cpp, continued.

```
// EXT_framebuffer_object related definitions
// as per - http://oss.sgi.com/projects/ogl-
     sample/registry/EXT/framebuffer_object.txt
// EXT_framebuffer_object - http://oss.sgi.com/projects/ogl-
     sample/registry/EXT/framebuffer_object.txt
extern PFNGLISRENDERBUFFEREXTPROC glIsRenderbufferEXT = NULL;
extern PFNGLBINDRENDERBUFFEREXTPROC glBindRenderbufferEXT = NULL;
extern PFNGLDELETERENDERBUFFERSEXTPROC glDeleteRenderbuffersEXT = NULL;
extern PFNGLGENRENDERBUFFERSEXTPROC glGenRenderbuffersEXT = NULL;
extern PFNGLRENDERBUFFERSTORAGEEXTPROC glRenderbufferStorageEXT = NULL;
extern PFNGLGETRENDERBUFFERPARAMETERIVEXTPROC
     glGetRenderbufferParameterivEXT = NULL;
extern PFNGLISFRAMEBUFFEREXTPROC glIsFramebufferEXT = NULL;
extern PFNGLBINDFRAMEBUFFEREXTPROC glBindFramebufferEXT = NULL;
extern PFNGLDELETEFRAMEBUFFERSEXTPROC glDeleteFramebuffersEXT = NULL;
extern PFNGLGENFRAMEBUFFERSEXTPROC glGenFramebuffersEXT = NULL;
extern PFNGLCHECKFRAMEBUFFERSTATUSEXTPROC glCheckFramebufferStatusEXT =
     NULL;
extern PFNGLFRAMEBUFFERTEXTURE1DEXTPROC glFramebufferTexture1DEXT = NULL;
extern PFNGLFRAMEBUFFERTEXTURE2DEXTPROC glFramebufferTexture2DEXT = NULL;
extern PFNGLFRAMEBUFFERTEXTURE3DEXTPROC glFramebufferTexture3DEXT = NULL;
extern PFNGLFRAMEBUFFERRENDERBUFFEREXTPROC glFramebufferRenderbufferEXT =
     NULL;
extern PFNGLGETFRAMEBUFFERATTACHMENTPARAMETERIVEXTPROC
     glGetFramebufferAttachmentParameterivEXT = NULL;
extern PFNGLGENERATEMIPMAPEXTPROC glGenerateMipmapEXT = NULL;

// Function definitions
void   reshape(int w, int h);
void   keyb(unsigned char k, int x, int y);
void        loadBMP(void);
void        loadBMP2(void);
void   saveInputFile1(void);
void   saveInputFile2(void);
void   saveConstructedImage(void);
void   saveImageDiff1(void);
void   saveImageDiff2(void);
void   evaluateImageDiffs(void);
void   CPUmotionEstimation(void);
void   checkCorrelation(void);
bool   checkFramebufferStatus(void);
void   cgErrorCallback(void);
void   updatePart2(void);
void   display(void);
void   saveMVdata(void);
void   saveStats(void);

// Cg related globals
CGcontext   g_cgContext;
CGprofile   g_cgProfile;
float myTemp2[4];   // a parameter being passed to the CG shader
float myTemp[4];    // a parameter being passed to the CG shader

// Input filename containers
char inputFile1[maxString],inputFile2[maxString];
// Output filename container
char resultsFilename[maxString];
// Predefined itermediate results files
char outputFile1[maxString]="Data\\output1.pgm";
char outputFile2[maxString]="Data\\output2.pgm";
char outputFile3[maxString]="Data\\output3.pgm";
char outputFile4[maxString]="Data\\imageDiff1.pgm";
char outputFile5[maxString]="Data\\imageDiff2.pgm";
char cpuMVfile[maxString]="Data\\cpuMVfile.pgm";
char gpuMVfile[maxString]="Data\\gpuMVfile.pgm";
```

GpuCpuVideo.cpp, continued.

```
// number of pixels in one side of the block used
float scale=8;
// A suitable divisor to alleviate the 255 truncation at block summation
// Also appears in the Cg program in a similar role
int truncationScale = 8;
// The number of times to repeat the exercise to increase results accuracy
// will be set inside main()
int testsToRun;
double end,timeForUpdate=0;      // Timing the frames

// Arrays which will contain the grayscale image data in integer form
static GLubyte checkImage[checkImageHeight][checkImageWidth][4];         //
     Input image 1
static GLubyte checkImageTwo[checkImageHeight][checkImageWidth][4];
       // Input image 2
static GLubyte constructedImage[checkImageHeight][checkImageWidth][4];   //
     Motion Estimated image
// Data array for saving the texture to, when required to move from GPU to
     CPU
static GLubyte data[checkImageWidth][checkImageHeight][4];
static GLubyte outData[checkImageWidth][checkImageHeight][4];
// Store the motion vector index associated with the best SAD found so far
     in the bottom left reducedInfo area
static GLubyte MVdata[checkImageWidth][checkImageWidth][4];
// Store the lowest SAD sums found so far for each block after offset image
     subtraction
static GLubyte blockSums[checkImageWidth][checkImageWidth][4];

// Global integers related to the OpenGL management
GLuint fbo, color, depth, depth_rb, latestTextureUpdated,
     destinationAttachment;

// The predefined offsets to use for block matching.
// Set in such a fashion that currentOffsetIndex is an index into the
     arrays, to signify
// what offsets in X and Y directions to search.
// Note: The 0,0 elements are deliberately last to force a failed search for
     optimum offsets
//      to return a 0,0 result.
// Note: For Looping could have been used, but this method permits custom
     search strategies
int currentOffsetIndex;
int offsetArrayX[]={-7,-6,-5,-4,-3,-2,-1,0,1,2,3,4,5,6,7,
                          -7,-6,-5,-4,-3,-2,-1,0,1,2,3,4,5,6,7,
                          -7,-6,-5,-4,-3,-2,-1,0,1,2,3,4,5,6,7,
                          -7,-6,-5,-4,-3,-2,-1,0,1,2,3,4,5,6,7,
                          -7,-6,-5,-4,-3,-2,-1,0,1,2,3,4,5,6,7,
                          -7,-6,-5,-4,-3,-2,-1,0,1,2,3,4,5,6,7,
                          -7,-6,-5,-4,-3,-2,-1,0,1,2,3,4,5,6,7,
                          -7,-6,-5,-4,-3,-2,-1,0,1,2,3,4,5,6,7,
                          -7,-6,-5,-4,-3,-2,-1,0,1,2,3,4,5,6,7,
                          -7,-6,-5,-4,-3,-2,-1,0,1,2,3,4,5,6,7,
                          -7,-6,-5,-4,-3,-2,-1,0,1,2,3,4,5,6,7,
                          -7,-6,-5,-4,-3,-2,-1,0,1,2,3,4,5,6,7,
                          -7,-6,-5,-4,-3,-2,-1,0,1,2,3,4,5,6,7,
                          -7,-6,-5,-4,-3,-2,-1,0,1,2,3,4,5,6,7,
                          -7,-6,-5,-4,-3,-2,-1,7,6,5,4,3,2,1,0
                          };
int offsetArrayY[]={-7,-7,-7,-7,-7,-7,-7,-7,-7,-7,-7,-7,-7,-7,-7,
                          -6,-6,-6,-6,-6,-6,-6,-6,-6,-6,-6,-6,-6,-6,-6,
                          -5,-5,-5,-5,-5,-5,-5,-5,-5,-5,-5,-5,-5,-5,-5,
                          -4,-4,-4,-4,-4,-4,-4,-4,-4,-4,-4,-4,-4,-4,-4,
                          -3,-3,-3,-3,-3,-3,-3,-3,-3,-3,-3,-3,-3,-3,-3,
                          -2,-2,-2,-2,-2,-2,-2,-2,-2,-2,-2,-2,-2,-2,-2,
                          -1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,
```

GpuCpuVideo.cpp, continued.

```
                            7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,
                            6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,
                            5,5,5,5,5,5,5,5,5,5,5,5,5,5,5,
                            4,4,4,4,4,4,4,4,4,4,4,4,4,4,4,
                            3,3,3,3,3,3,3,3,3,3,3,3,3,3,3,
                            2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,
                            1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,
                            0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
                            };

// The dimensions of the image array
int           intImageWidth=checkImageWidth,
     intImageHeight=checkImageHeight;
// Handles to texture data arrays
unsigned int  iTexture;
unsigned int  jTexture,_jNewTexture, jTexture2, jTextureDiff,
     jTextureBlockSums, jTextureMVdata;
// The toggling texture which is read from
unsigned int  actualTexture;
unsigned int  outDataTexture;
 // Handles to fragment programs
CGprogram
     _fragmentProgram,_fragmentProgramDiff,_fragmentProgramBlockSums,
     _fragmentProgramMVdata;
// Handles to the uniform parameters of fragment programs
CGparameter   _textureParam, _textureParam2, _textureParamDiff,
     _textureParamBlockSums, _textureParamMVdata;
CGparameter   outDataTextureParam, myCParameter, myCDistance;



/**
 * gpuInit
 *
 * Sets up the OpenGL and Cg environment in preparation for GLUT.
 *
 * @param       void
 * @return      void
 *
 * @author      Jason Ruane
 * @version     1.0
 */
void gpuInit(void)
{
     // initialise the unifiorm parameter
     // A float4 variable holding data to send to the Cg shader
     myTemp[0] = 1;
     myTemp[1] = intImageWidth;
     myTemp[2] = intImageHeight;
     myTemp[3] = intImageHeight*intImageWidth;
     // FYI information
     if (debug)     printf("Using Width %d and Height
     %d\n",intImageWidth,intImageHeight);

    // Setup Cg, how to handle any errors in Cg
    cgSetErrorCallback(cgErrorCallback);
     // Declare a handle to the Cg context
    g_cgContext = cgCreateContext();

    // Get the best profile for this hardware
    g_cgProfile = cgGLGetLatestProfile(CG_GL_FRAGMENT);
    assert(g_cgProfile != CG_PROFILE_UNKNOWN);
    cgGLSetOptimalOptions(g_cgProfile);
```

GpuCpuVideo.cpp, continued.

```
    // FYI check to see what this machine's largest texture size is
    int maxtexsize;
    glGetIntegerv(GL_MAX_TEXTURE_SIZE,&maxtexsize);
    if(debug)
    {
      printf("GL_MAX_TEXTURE_SIZE, %d\n",maxtexsize);
      printf("CG_profile, %d\n",g_cgProfile);
    }

    // FYI check to see what this machine's largest render buffer is
    GLint max_size = 0;
    glGetIntegerv( GL_MAX_RENDERBUFFER_SIZE_EXT, &max_size );
    if (debug) printf("glRenderBuffer:: max size %d\n",max_size);
    // FYI check to see how many colour attachments are possible
    glGetIntegerv( GL_MAX_COLOR_ATTACHMENTS_EXT, &max_size );
    if (debug) printf("glRenderBuffer:: max number of colour attachments
    %d\n",max_size);

    // Generate a texture for writing to
    glGenTextures(1, &iTexture);
    glBindTexture(GL_TEXTURE_2D, iTexture);
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_NEAREST);
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_NEAREST);
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_CLAMP);
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_CLAMP);
    glTexImage2D(GL_TEXTURE_2D, 0, GL_RGBA8, intImageWidth, intImageHeight,
                  0, GL_RGBA, GL_FLOAT, NULL);

    // generate a texture and bind to the checkImage array
    glGenTextures(1, &jTexture);
    glBindTexture(GL_TEXTURE_2D, jTexture);
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_NEAREST);
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_NEAREST);
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_CLAMP);
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_CLAMP);
    glTexImage2D(GL_TEXTURE_2D, 0, GL_RGBA8, intImageWidth, intImageHeight,
                   0, GL_RGBA, GL_UNSIGNED_BYTE, checkImage);


    // generate a texture and bind to the checkImageTwo array
    glGenTextures(1, &jTexture2);
    glBindTexture(GL_TEXTURE_2D, jTexture2);
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_NEAREST);
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_NEAREST);
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_CLAMP);
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_CLAMP);
    glTexImage2D(GL_TEXTURE_2D, 0, GL_RGBA8, intImageWidth, intImageHeight,
                   0, GL_RGBA, GL_UNSIGNED_BYTE, checkImageTwo);


    // create a third texture, to be used as the image difference target
    glGenTextures(1, &jTextureDiff);
    glBindTexture(GL_TEXTURE_2D, jTextureDiff);
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_NEAREST);
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_NEAREST);
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_CLAMP);
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_CLAMP);
    glTexImage2D(GL_TEXTURE_2D, 0, GL_RGBA8, intImageWidth, intImageHeight,
                   0, GL_RGBA, GL_UNSIGNED_BYTE, NULL);

    // create a texture, to be used as the blocksums array (best SAD value
    found so far)
    glGenTextures(1, &jTextureBlockSums);
    glBindTexture(GL_TEXTURE_2D, jTextureBlockSums);
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_NEAREST);
```

GpuCpuVideo.cpp, continued.

```
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_NEAREST);
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_CLAMP);
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_CLAMP);
    glTexImage2D(GL_TEXTURE_2D, 0, GL_RGBA8, intImageWidth, intImageHeight,
                  0, GL_RGBA, GL_UNSIGNED_BYTE, NULL);

    // create a texture, to be used as the MVdata array (index of the
    offset for the best SAD value found so far)
    glGenTextures(1, &jTextureMVdata);
    glBindTexture(GL_TEXTURE_2D, jTextureMVdata);
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_NEAREST);
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_NEAREST);
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_CLAMP);
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_CLAMP);
    glTexImage2D(GL_TEXTURE_2D, 0, GL_RGBA8, intImageWidth, intImageHeight,
                  0, GL_RGBA, GL_UNSIGNED_BYTE, NULL);


    // Create the fragment programs and assign to handles
    // 8*8 summation
    _fragmentProgram = cgCreateProgramFromFile(g_cgContext, CG_SOURCE,

    "fragmentShaderSum8by8.cg", g_cgProfile,
                                                        "edges", NULL);
    // Image subtraction / difference
    _fragmentProgramDiff = cgCreateProgramFromFile(g_cgContext, CG_SOURCE,

    "fragmentShaderDiff.cg", g_cgProfile,
                                                        "edges", NULL);
    // Block summation comparison
    _fragmentProgramBlockSums = cgCreateProgramFromFile(g_cgContext,
    CG_SOURCE,

    "fragmentShaderBlockSums.cg", g_cgProfile,
                                                        "edges", NULL);
    // Block summation comparison
    _fragmentProgramMVdata = cgCreateProgramFromFile(g_cgContext,
    CG_SOURCE,

    "fragmentShaderMVdata.cg", g_cgProfile,
                                                        "edges", NULL);


    // Load the fragment programs and check ability to get handle of
    parameters
    if(_fragmentProgramDiff != NULL)
    {
      if(debug)     printf("fragmentProgramDiff loaded ok\n");
      cgGLLoadProgram(_fragmentProgramDiff);
      cgGLBindProgram(_fragmentProgramDiff);
      cgGLEnableProfile(g_cgProfile);
      _textureParam = cgGetNamedParameter(_fragmentProgramDiff, "texture");
      _textureParam2 = cgGetNamedParameter(_fragmentProgramDiff,
    "texture2");
      _textureParamDiff = cgGetNamedParameter(_fragmentProgramDiff,
    "textureDiff");
    }
    if(_fragmentProgram != NULL)
    {
      if(debug)     printf("fragmentProgram loaded ok\n");
      cgGLLoadProgram(_fragmentProgram);
      cgGLBindProgram(_fragmentProgram);
      cgGLEnableProfile(g_cgProfile);
      _textureParam = cgGetNamedParameter(_fragmentProgram, "texture");
      _textureParam2 = cgGetNamedParameter(_fragmentProgram, "texture2");
```

GpuCpuVideo.cpp, continued.

```
      _textureParamDiff = cgGetNamedParameter(_fragmentProgram,
    "textureDiff");
    }
    if(_fragmentProgramBlockSums != NULL)
    {
      if(debug)     printf("fragmentProgramBlockSums loaded ok\n");
      cgGLLoadProgram(_fragmentProgramBlockSums);
      cgGLBindProgram(_fragmentProgramBlockSums);
      cgGLEnableProfile(g_cgProfile);
      _textureParam = cgGetNamedParameter(_fragmentProgramBlockSums,
    "textureBlockSums");
    }
    if(_fragmentProgramMVdata != NULL)
    {
      if(debug)     printf("fragmentProgramMVdata loaded ok\n");
      cgGLLoadProgram(_fragmentProgramMVdata);
      cgGLBindProgram(_fragmentProgramMVdata);
      cgGLEnableProfile(g_cgProfile);
    }


    // Set a normal uniform parameter for the fragment program
    if(_fragmentProgram != NULL)
    {
      CGparameter myCParameter = cgGetNamedParameter( _fragmentProgram,
    "myCgParameter");
      cgGLSetParameter4fv(myCParameter, myTemp );
    }

    // Set the OpenGL viewport
    glViewport(0, 0, intImageWidth, intImageHeight);

    // EXT_framebuffer_object requirements to enable Frame Buffer Object
    usage
    // as per myxo.css.msu.edu:443/d2x-xl/trunk/arch/ogl/fbuffer.c
    char *ext = (char*)glGetString( GL_EXTENSIONS );
    if( strstr( ext, "EXT_framebuffer_object" ) == NULL )
    {
      printf("EXT_framebuffer_object extension was not found");
    }
    else
    {
      glIsRenderbufferEXT =
    (PFNGLISRENDERBUFFEREXTPROC)wglGetProcAddress("glIsRenderbufferEXT");
      glBindRenderbufferEXT =
    (PFNGLBINDRENDERBUFFEREXTPROC)wglGetProcAddress("glBindRenderbufferEXT"
    );
      glDeleteRenderbuffersEXT =
    (PFNGLDELETERENDERBUFFERSEXTPROC)wglGetProcAddress("glDeleteRenderbuffe
    rsEXT");
      glGenRenderbuffersEXT =
    (PFNGLGENRENDERBUFFERSEXTPROC)wglGetProcAddress("glGenRenderbuffersEXT"
    );
      glRenderbufferStorageEXT =
    (PFNGLRENDERBUFFERSTORAGEEXTPROC)wglGetProcAddress("glRenderbufferStora
    geEXT");
      glGetRenderbufferParameterivEXT =
    (PFNGLGETRENDERBUFFERPARAMETERIVEXTPROC)wglGetProcAddress("glGetRenderb
    ufferParameterivEXT");
      glIsFramebufferEXT =
    (PFNGLISFRAMEBUFFEREXTPROC)wglGetProcAddress("glIsFramebufferEXT");
      glBindFramebufferEXT =
    (PFNGLBINDFRAMEBUFFEREXTPROC)wglGetProcAddress("glBindFramebufferEXT");
```

GpuCpuVideo.cpp, continued.

```
  glDeleteFramebuffersEXT =
(PFNGLDELETEFRAMEBUFFERSEXTPROC)wglGetProcAddress("glDeleteFramebuffers
EXT");
  glGenFramebuffersEXT =
(PFNGLGENFRAMEBUFFERSEXTPROC)wglGetProcAddress("glGenFramebuffersEXT");
  glCheckFramebufferStatusEXT =
(PFNGLCHECKFRAMEBUFFERSTATUSEXTPROC)wglGetProcAddress("glCheckFramebuff
erStatusEXT");
  glFramebufferTexture1DEXT =
(PFNGLFRAMEBUFFERTEXTURE1DEXTPROC)wglGetProcAddress("glFramebufferTextu
re1DEXT");
  glFramebufferTexture2DEXT =
(PFNGLFRAMEBUFFERTEXTURE2DEXTPROC)wglGetProcAddress("glFramebufferTextu
re2DEXT");
  glFramebufferTexture3DEXT =
(PFNGLFRAMEBUFFERTEXTURE3DEXTPROC)wglGetProcAddress("glFramebufferTextu
re3DEXT");
  glFramebufferRenderbufferEXT =
(PFNGLFRAMEBUFFERRENDERBUFFEREXTPROC)wglGetProcAddress("glFramebufferRe
nderbufferEXT");
  glGetFramebufferAttachmentParameterivEXT =
(PFNGLGETFRAMEBUFFERATTACHMENTPARAMETERIVEXTPROC)wglGetProcAddress("glG
etFramebufferAttachmentParameterivEXT");
  glGenerateMipmapEXT =
(PFNGLGENERATEMIPMAPEXTPROC)wglGetProcAddress("glGenerateMipmapEXT");

  if( !glIsRenderbufferEXT || !glBindRenderbufferEXT ||
!glDeleteRenderbuffersEXT ||
        !glGenRenderbuffersEXT || !glRenderbufferStorageEXT ||
!glGetRenderbufferParameterivEXT ||
        !glIsFramebufferEXT || !glBindFramebufferEXT ||
!glDeleteFramebuffersEXT ||
        !glGenFramebuffersEXT || !glCheckFramebufferStatusEXT ||
!glFramebufferTexture1DEXT ||
        !glFramebufferTexture2DEXT || !glFramebufferTexture3DEXT ||
!glFramebufferRenderbufferEXT||
        !glGetFramebufferAttachmentParameterivEXT ||
!glGenerateMipmapEXT )
  {
        printf("One or more EXT_framebuffer_object functions were not
found");
  }
}


// Create an FBO (Frame Buffer Object) for offscreen rendering
glGenFramebuffersEXT(1, &fbo);

// Create color texture, null contents yet
glGenTextures(1, &color);
glBindTexture(GL_TEXTURE_2D, color);
glTexImage2D(GL_TEXTURE_2D, 0, GL_RGBA8, intImageWidth,
                     intImageHeight, 0, GL_RGBA, GL_UNSIGNED_BYTE,
NULL);
// Confirm the framebuffer is OK so far
checkFramebufferStatus();
// Create depth renderbuffer
glGenRenderbuffersEXT(1, &depth);

// Bind the FBO and attach color textures to it
glBindFramebufferEXT(GL_FRAMEBUFFER_EXT, fbo);
glFramebufferTexture2DEXT(GL_FRAMEBUFFER_EXT, GL_COLOR_ATTACHMENT0_EXT,
GL_TEXTURE_2D,
                                        jTexture, 0);
```

GpuCpuVideo.cpp, continued.

```
     glFramebufferTexture2DEXT(GL_FRAMEBUFFER_EXT, GL_COLOR_ATTACHMENT1_EXT,
     GL_TEXTURE_2D,
                                                jTexture2, 0);
     glFramebufferTexture2DEXT(GL_FRAMEBUFFER_EXT, GL_COLOR_ATTACHMENT2_EXT,
     GL_TEXTURE_2D,
                                                iTexture, 0);
     glFramebufferTexture2DEXT(GL_FRAMEBUFFER_EXT, GL_COLOR_ATTACHMENT3_EXT,
     GL_TEXTURE_2D,
                                                jTextureDiff, 0);
     glBindFramebufferEXT(GL_FRAMEBUFFER_EXT, fbo);
     glGenRenderbuffersEXT(1, &depth_rb);// render buffer
     // initialise depth renderbuffer
     glBindRenderbufferEXT(GL_RENDERBUFFER_EXT, depth_rb);
     glRenderbufferStorageEXT(GL_RENDERBUFFER_EXT, GL_DEPTH_COMPONENT24,
     intImageWidth, intImageHeight);
     // attach renderbuffer to framebuffer
     glFramebufferRenderbufferEXT(GL_FRAMEBUFFER_EXT,
     GL_DEPTH_ATTACHMENT_EXT, GL_RENDERBUFFER_EXT, depth_rb);
     // Define which is the current drawing location
     glDrawBuffer(GL_COLOR_ATTACHMENT2_EXT);
     // Confirm the framebuffer is OK so far
     checkFramebufferStatus();

}    // End of gpuInit
//-------------------------------------------------------------------------



/**
 * main
 *
 * Standard C main function - Entry point for program
 *
 *
 * @param       argc       Number of arguments passed
 * @param       argv       Items of arguments passed
 * @return      int        0 if successful, non-zero if not.
 *
 * @author      Jason Ruane
 * @version     1.0
 */
int main(int argc, char **argv)
{
     // Inform user
     printf("Starting GPU-CPU motion vector search with %d
     parameters,\n",argc-1);
     // If the correct number of arguments were supplied
     if(argc==5)
     {
       // Retrieve the first input image filename
       printf("Input file 1:      %s\n",argv[1]);
       strcpy(inputFile1,argv[1]);
       // Retrieve the second input image filename
       printf("Input file 2:      %s\n",argv[2]);
       strcpy(inputFile2,argv[2]);
       // Retrieve the first output image filename
       printf("Output file:       %s\n",argv[3]);
       strcpy(resultsFilename,argv[3]);
       // Retrieve the number of times to run the search
       printf("Repeat times:      %s\n",argv[4]);
       testsToRun = atoi(argv[4]);
     }
     else
     {
```

GpuCpuVideo.cpp, continued.

```
      // Inform user of required parameters
      printf("Please use 2 input files,1 result file\n");
      printf("and number of loops to repeat as command line
    parameters,\n");
      // Terminate program and return to OS
      exit(0);
    }

    // Load the two input files into the data arrays
    loadBMP();
    loadBMP2();

    // Setup the GLUT system
    // RGBA display mode required
   glutInitDisplayMode(GLUT_DOUBLE | GLUT_RGBA);
    // Window size in pixels
   glutInitWindowSize(checkImageWidth, checkImageWidth);
    // Initial position of window
    glutInitWindowPosition(100, 100);
    // Create the display window
   glutCreateWindow("GPUcompute");
    // Set the funtion to call when the GPU is ready
   glutIdleFunc(updatePart2);
    // If the GLUT window is resized, what to run
   glutReshapeFunc(reshape);
    // Set up system and load texture
    gpuInit();

    // Initiate the GLUT windowing system
    // It will not return so the program exit() is contained in the
    // updatePart2 function specified in the above glutIdleFunc
    glutMainLoop();

    return(0); // Never actually reached, but left for completeness

}    // end of main
//--------------------------------------------------------------------



/**
 * updatePart2
 *
 * Sets the input to the Cg shader and enacts a drawing to the screen
 * so that a single image offset is evaluated, its image difference is
 * found, the sum absolute difference found, compared with the best SAD
 * so far, and if lower, is used as the lowest SAD for next run.
 * This function is called repeatedly by GLUT.
 *
 * @param       void
 * @return      void
 *
 * @author      Jason Ruane
 * @version     1.0
 */
void updatePart2()
{
    // The number of offsets to be checked (related to the window size)
    int numberOfOffsets=sizeof(offsetArrayX)/sizeof(offsetArrayX[0]);

    // Fill the winning BlockSums texture with max values, which the SADs
    will later beat (ie. be lower than)
    glFramebufferTexture2DEXT(GL_FRAMEBUFFER_EXT, GL_COLOR_ATTACHMENT3_EXT,
    GL_TEXTURE_2D, jTextureBlockSums, 0);
    destinationAttachment=GL_COLOR_ATTACHMENT3_EXT;
```

GpuCpuVideo.cpp, continued.

```
    glDrawBuffer(destinationAttachment);
    glClearColor( 1.0f, 1.0f, 1.0f, 1.0f );                                  //
    Assigns max value to each element
    glClear( GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT );

    // Clear the jTextureMVdata array before starting (set to all zero)
    glFramebufferTexture2DEXT(GL_FRAMEBUFFER_EXT, GL_COLOR_ATTACHMENT3_EXT,
    GL_TEXTURE_2D, jTextureMVdata, 0);
    destinationAttachment=GL_COLOR_ATTACHMENT2_EXT;
    glDrawBuffer(destinationAttachment);
    glClearColor( 0.0f, 0.0f, 0.0f, 1.0f );                                  //
    Assigns min value to each element
    glClear( GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT );

    // Clear the iTexture array before starting (set to all zero)
    glFramebufferTexture2DEXT(GL_FRAMEBUFFER_EXT, GL_COLOR_ATTACHMENT2_EXT,
    GL_TEXTURE_2D, iTexture, 0);
    destinationAttachment=GL_COLOR_ATTACHMENT2_EXT;
    glDrawBuffer(destinationAttachment);
    glClearColor( 0.0f, 0.0f, 0.0f, 1.0f );                                  //
    Assigns min value to each element
    glClear( GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT );

    // Restore the starting attachments
    glFramebufferTexture2DEXT(GL_FRAMEBUFFER_EXT, GL_COLOR_ATTACHMENT0_EXT,
    GL_TEXTURE_2D, jTexture, 0);
    glFramebufferTexture2DEXT(GL_FRAMEBUFFER_EXT, GL_COLOR_ATTACHMENT1_EXT,
    GL_TEXTURE_2D, jTexture2, 0);
    glFramebufferTexture2DEXT(GL_FRAMEBUFFER_EXT, GL_COLOR_ATTACHMENT2_EXT,
    GL_TEXTURE_2D, iTexture, 0);
    glFramebufferTexture2DEXT(GL_FRAMEBUFFER_EXT, GL_COLOR_ATTACHMENT3_EXT,
    GL_TEXTURE_2D, jTextureDiff, 0);

    double tempStart = clock();        // Start the GPU timer


    // Loop through the number of times the iteration is to be performed
    // Done to increase accuracy of the timings
    for ( int testRun=0;testRun<testsToRun;testRun++)
    {
      // Inform user of progress
      printf("GPU iteration %d of %d.\n",testRun+1,testsToRun);

      // Scan through the entire array of offsets to be examined
      for (currentOffsetIndex=0;currentOffsetIndex <
    numberOfOffsets;currentOffsetIndex++)
      {
            // The actual offsets are contained in the predefined arrays,
            // into which the index currentOffsetIndex is used
            int currentOffsetX=offsetArrayX[currentOffsetIndex];
            int currentOffsetY=offsetArrayY[currentOffsetIndex];

            //-------------------------------------------------------------
            // Step 1: calculate an image subtraction of the two input
    textures
            //                  and write the output to jTextureDiff
            if (debug)    printf("Performing the image subtraction\n");
            latestTextureUpdated=jTextureDiff;  // destination to write to
            glFramebufferTexture2DEXT(GL_FRAMEBUFFER_EXT,
    GL_COLOR_ATTACHMENT3_EXT,

      GL_TEXTURE_2D, latestTextureUpdated, 0);
            destinationAttachment=GL_COLOR_ATTACHMENT3_EXT;
            glDrawBuffer(destinationAttachment);
            // FYI check of the framebuffer completeness
```

GpuCpuVideo.cpp, continued.

```
        if(debug)    printf("Check frameBufferStatus :
\n");checkFramebufferStatus();

        // Bind the Cg program
        cgGLEnableProfile(g_cgProfile);
        cgGLBindProgram(_fragmentProgramDiff);

        // Connect to the variables and pass in the values of the
offsets to check
        myTemp[0]=currentOffsetX; // the Xoffset to test
        myTemp[1]=currentOffsetY; // the Yoffset to test
        myTemp[2]=checkImageWidth;
        myTemp[3]=checkImageHeight;
        myCParameter = cgGetNamedParameter( _fragmentProgramDiff,
"myCgParameter");
        cgGLSetParameter4fv(myCParameter, myTemp );

        // bind the alternating texture as input (the other will be
output
        glBindTexture(GL_TEXTURE_2D, jTexture);
        glBindTexture(GL_TEXTURE_2D, jTexture2);

        // bind the textures as input to the filter
        _textureParam = cgGetNamedParameter(_fragmentProgramDiff,
"texture");
        _textureParam2 = cgGetNamedParameter(_fragmentProgramDiff,
"texture2");
        _textureParamDiff = cgGetNamedParameter(_fragmentProgramDiff,
"textureDiff");
        cgGLSetTextureParameter(_textureParam, jTexture);
        cgGLEnableTextureParameter(_textureParam);
        cgGLSetTextureParameter(_textureParam2, jTexture2);
        cgGLEnableTextureParameter(_textureParam2);

        // The output texture will be jTextureDiff, clear it before
writing to
        glClearColor( 0.0f, 0.0f, 0.0f, 1.0f );
        glClear( GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT );

        // Actually draw the data via a quad, thus enacting the GPU
computations (Cg program)
        glEnable(GL_TEXTURE_2D);
        glPolygonMode(GL_FRONT,GL_FILL);
        glBegin(GL_QUADS);
        {
               glTexCoord2f(0, 0); glVertex3f(-1, -1, -0.5f);
               glTexCoord2f(1, 0); glVertex3f( 1, -1, -0.5f);
               glTexCoord2f(1, 1); glVertex3f( 1,  1, -0.5f);
               glTexCoord2f(0, 1); glVertex3f(-1,  1, -0.5f);
        }
        glEnd();
        // disable the shader
        cgGLDisableProfile(g_cgProfile);
        cgGLDisableTextureParameter(_textureParamDiff);
        cgGLDisableTextureParameter(_textureParam2);
        cgGLDisableTextureParameter(_textureParam);
        glDisable(GL_TEXTURE_2D);

        // End of image subtraction step

        //-------------------------------------------------------------
        // Step 2: sum the 8*8 pixel block in jTextureDiff (after
offset image subtraction)
        //                  and write it to iTexture
```

GpuCpuVideo.cpp, continued.

```
            // FYI information
            if(debug)    printf("Starting the 8*8 summation step\n");

            // FBO related switches
            // actualTexture is the one to read from,
    framebuffertexture/glDrawBuffer is the one to write to
            // Set the source and destination buffers for this stage
            actualTexture=jTextureDiff;             // source of the sums
    information
            latestTextureUpdated=iTexture;          // destination to
    write to
            glFramebufferTexture2DEXT(GL_FRAMEBUFFER_EXT,
    GL_COLOR_ATTACHMENT2_EXT,

      GL_TEXTURE_2D, latestTextureUpdated, 0);
            destinationAttachment=GL_COLOR_ATTACHMENT2_EXT;
            glDrawBuffer(destinationAttachment);
            checkFramebufferStatus();

            // Bind the 8*8 summation fragment shader:
            cgGLEnableProfile(g_cgProfile);
            cgGLBindProgram(_fragmentProgram);

            // connect to the variables and pass in the values of the
    offsets to check
            myTemp[0]=currentOffsetX; // the Xoffset to test
            myTemp[1]=currentOffsetY; // the Yoffset to test
            myTemp[2]=checkImageWidth;
            myTemp[3]=checkImageHeight;
            myCParameter = cgGetNamedParameter( _fragmentProgram,
    "myCgParameter");
            cgGLSetParameter4fv(myCParameter, myTemp );

            // bind the texture as input
            glBindTexture(GL_TEXTURE_2D, jTexture);
            glBindTexture(GL_TEXTURE_2D, jTexture2);
            glBindTexture(GL_TEXTURE_2D, jTextureDiff);
            _textureParam = cgGetNamedParameter(_fragmentProgram,
    "texture");
            _textureParam2 = cgGetNamedParameter(_fragmentProgram,
    "texture2");
            _textureParamDiff = cgGetNamedParameter(_fragmentProgram,
    "textureDiff");
            cgGLSetTextureParameter(_textureParam, jTexture);
            cgGLEnableTextureParameter(_textureParam);
            cgGLSetTextureParameter(_textureParam2, jTexture2);
            cgGLEnableTextureParameter(_textureParam2);
            cgGLSetTextureParameter(_textureParamDiff, jTextureDiff);
            cgGLEnableTextureParameter(_textureParamDiff);

            glEnable(GL_TEXTURE_2D);
            glPolygonMode(GL_FRONT,GL_FILL);
            glBegin(GL_QUADS);
            {
                  glTexCoord2f(0, 0); glVertex3f(-1, -1, -0.5f);
                  glTexCoord2f(1, 0); glVertex3f( -1+2/scale, -1, -0.5f);
                  glTexCoord2f(1, 1); glVertex3f( -1+2/scale,  -1+2/scale,
    -0.5f);
                  glTexCoord2f(0, 1); glVertex3f(-1,  -1+2/scale, -0.5f);
            }
            glEnd();

            // disable the shader
            cgGLDisableProfile(g_cgProfile);
            cgGLDisableTextureParameter(_textureParamDiff);
```

GpuCpuVideo.cpp, continued.

```
            glDisable(GL_TEXTURE_2D);

            // ------------------- End of the 8*8 summation

            //-------------------------------------------------------------
            // Step 3: Compare the block summations to those found so far
    and record if lower
            //                  writing the ouput to jTextureBlockSums

            // FYI information
            if (debug)   printf("Performing the block sums comparison\n");

            // Set the source and destination buffers for this stage
            actualTexture=iTexture;                                 // source
    of the sums information
            latestTextureUpdated=jTextureBlockSums; // destination to
    write to
            glFramebufferTexture2DEXT(GL_FRAMEBUFFER_EXT,
    GL_COLOR_ATTACHMENT3_EXT,
                                                                    GL_TEXTURE_2D,
    jTextureBlockSums, 0);
            destinationAttachment=GL_COLOR_ATTACHMENT3_EXT;
            glDrawBuffer(destinationAttachment);
            checkFramebufferStatus();

            // Bind the Cg program
            cgGLEnableProfile(g_cgProfile);
            cgGLBindProgram(_fragmentProgramBlockSums);

            // connect to the variables and pass in the values of the
    offsets to check
            myTemp[0]=currentOffsetX; // the Xoffset to test
            myTemp[1]=currentOffsetY; // the Yoffset to test
            myTemp[2]=checkImageWidth;
            myTemp[3]=checkImageHeight;
            myCParameter = cgGetNamedParameter( _fragmentProgramBlockSums,
    "myCgParameter");
            cgGLSetParameter4fv(myCParameter, myTemp );

            // Bind the texture as input
            glBindTexture(GL_TEXTURE_2D, iTexture);

            // Bind the textures as input to the shader
            _textureParam = cgGetNamedParameter(_fragmentProgramBlockSums,
    "textureSums");
            cgGLSetTextureParameter(_textureParam, iTexture);
            cgGLEnableTextureParameter(_textureParam);

            // this texture is also the destination
            // - note a toggling destination could be used to eliminate
    read/write to same texture
            _textureParam2 =
    cgGetNamedParameter(_fragmentProgramBlockSums,
    "textureBlockSumsWinning");
            cgGLSetTextureParameter(_textureParam2, jTextureBlockSums);
            cgGLEnableTextureParameter(_textureParam2);
            // FYI information
            if(debug)    printf("Drawing the image block sums comparisons
    quad\n");

            // Actually draw the window sized quad
            glEnable(GL_TEXTURE_2D);
            glPolygonMode(GL_FRONT,GL_FILL);
            glBegin(GL_QUADS);
            {
```

GpuCpuVideo.cpp, continued.

```
                // Only use the lower left reducedInfo area (containing
the block sums)
                // ie reducedInfoSize*reducedInfoSize pixels on origin
and destination
                glTexCoord2f(0, 0); glVertex3f(-1, -1, -0.5f);
                glTexCoord2f(1/scale, 0); glVertex3f( -1+2/scale, -1, -
0.5f);
                glTexCoord2f(1/scale, 1/scale); glVertex3f( -1+2/scale,
-1+2/scale, -0.5f);
                glTexCoord2f(0, 1/scale); glVertex3f(-1,  -1+2/scale, -
0.5f);
        }
        glEnd();

        // Disable the shader
        cgGLDisableProfile(g_cgProfile);
        cgGLDisableTextureParameter(_textureParam);
        cgGLDisableTextureParameter(_textureParam2);
        glDisable(GL_TEXTURE_2D);

        // End of blocks summation comparison step 3
        //-------------------------------------------------------------
------


        // Step 4: Record the offset used to find the current lowest
SAD for a block
        //-------------------------------------------------------------
------
        //                  writing the ouput to jTextureMVdata

        // FYI information
        if (debug)    printf("Performing the MVdata recording
stage\n");

        // Set the source and destination buffers for this stage
        latestTextureUpdated=jTextureMVdata;    // destination to
write to
        glFramebufferTexture2DEXT(GL_FRAMEBUFFER_EXT,
GL_COLOR_ATTACHMENT1_EXT, GL_TEXTURE_2D, jTextureBlockSums, 0);
        glFramebufferTexture2DEXT(GL_FRAMEBUFFER_EXT,
GL_COLOR_ATTACHMENT2_EXT, GL_TEXTURE_2D, iTexture, 0);
        glFramebufferTexture2DEXT(GL_FRAMEBUFFER_EXT,
GL_COLOR_ATTACHMENT3_EXT, GL_TEXTURE_2D, jTextureMVdata, 0);
        destinationAttachment=GL_COLOR_ATTACHMENT3_EXT;
        glDrawBuffer(destinationAttachment);
        if(debug)    printf("Check frameBufferStatus :
\n");checkFramebufferStatus();

        // Bind the Cg program
        cgGLEnableProfile(g_cgProfile);
        cgGLBindProgram(_fragmentProgramMVdata);
        // connect to the variables and pass in the values of the
offsets to check
        myTemp[0]=currentOffsetIndex; // the index of the current
offset
        myTemp[1]=0;
        myTemp[2]=checkImageWidth;
        myTemp[3]=checkImageHeight;
        myCParameter = cgGetNamedParameter( _fragmentProgramMVdata,
"myCgParameter");
        cgGLSetParameter4fv(myCParameter, myTemp );
        // Bind the texture as input
        glBindTexture(GL_TEXTURE_2D, iTexture);
        // Bind the textures as input to the filter
```

GpuCpuVideo.cpp, continued.

```
        _textureParam = cgGetNamedParameter(_fragmentProgramMVdata,
"textureBlockSumsWinning");
        cgGLSetTextureParameter(_textureParam, jTextureBlockSums);
        cgGLEnableTextureParameter(_textureParam);
        _textureParam2 = cgGetNamedParameter(_fragmentProgramMVdata,
"textureBlockSumsCurrent");
        cgGLSetTextureParameter(_textureParam2, iTexture);
        cgGLEnableTextureParameter(_textureParam2);
        _textureParamDiff =
cgGetNamedParameter(_fragmentProgramMVdata, "textureMVdata");
        cgGLSetTextureParameter(_textureParamDiff, jTextureMVdata);
        cgGLEnableTextureParameter(_textureParamDiff);
        // FYI information
        if(debug)    printf("Drawing the quad for recording the
winning motion vector\n");
        // Actually draw the window sized quad
        glEnable(GL_TEXTURE_2D);
        glPolygonMode(GL_FRONT,GL_FILL);
        glBegin(GL_QUADS);
        {
              // Only use the lower left reducedInfo area (containing
the block sums)
              // ie reducedInfoSize*reducedInfoSize pixels on origin
and destination
              glTexCoord2f(0, 0); glVertex3f(-1, -1, -0.5f);
              glTexCoord2f(1/scale, 0); glVertex3f( -1+2/scale, -1, -
0.5f);
              glTexCoord2f(1/scale, 1/scale); glVertex3f( -1+2/scale,
-1+2/scale, -0.5f);
              glTexCoord2f(0, 1/scale); glVertex3f(-1,  -1+2/scale, -
0.5f);
        }
        glEnd();

        // disable the shader
        cgGLDisableProfile(g_cgProfile);
        cgGLDisableTextureParameter(_textureParam);
        cgGLDisableTextureParameter(_textureParam2);
        glDisable(GL_TEXTURE_2D);

        // End of step 4
        //-------------------------------------------------------------
-------

  } // end the loop of currentOffsetIndex

  // Allow the CPU to rest if SleepTime is set
  Sleep(SleepTime);

}// end testRun loop

// Stop the timer of GPU computations
timeForUpdate = clock()-tempStart;

display();     // Show the input image onscreen

saveMVdata();  // Copy the MVdata from GPU to CPU
saveStats();   // Record the performance metrics to output file

saveConstructedImage();      // Reconstruct the image using the input
image and the motion vectors
saveImageDiff1();            // Save the difference image of image1 -
image2
saveImageDiff2();            // Save the difference image of
reconstructed image - image2
```

GpuCpuVideo.cpp, continued.

```
     evaluateImageDiffs(); // Record the properties of the image differences
     (the PSNR)

     CPUmotionEstimation();       // Use the CPU to compute the motion
     vectors and record performance

     // Finished this program instance, exit normally.
     exit(0);

}  // end of updatePart2
//----------------------------------------------------------------------



/**
 * display
 *
 * Draws onscreen the input image within the GLUT window
 *
 * @param       void
 * @return      void
 *
 * @author      Jason Ruane
 * @version     1.0
 */
void display()
{
     // Bind to the GLUT window, after FBO usage
     glBindFramebufferEXT(GL_FRAMEBUFFER_EXT, 0);
     // Bind the input image texture
     glBindTexture(GL_TEXTURE_2D, jTexture2);
     glEnable(GL_TEXTURE_2D);
     // Draw the texture onscreen using a single quad
     glBegin(GL_QUADS);
     {
       glTexCoord2f(0, 0); glVertex3f(-1, -1, -0.5f);
       glTexCoord2f(1, 0); glVertex3f( 1, -1, -0.5f);
       glTexCoord2f(1, 1); glVertex3f( 1,  1, -0.5f);
       glTexCoord2f(0, 1); glVertex3f(-1,  1, -0.5f);
     }
     glEnd();
     // bring the framebuffer to the forefront, so image may be seen
     glutSwapBuffers();

     // Restore the FBO previous scenario
     glDisable(GL_TEXTURE_2D);
     glBindFramebufferEXT(GL_FRAMEBUFFER_EXT, fbo);

} // end of display
//----------------------------------------------------------------------



/**
 * saveMVdata
 *
 * Write the array of motion vectors to file.
 *
 *
 * @param       void
 * @return      void
 *
 * @author      Jason Ruane
 * @version     1.0
 */
```

GpuCpuVideo.cpp, continued.

```
void saveMVdata(void)
{
     // Just saving the one plane currently in use
     // Local variables in use
     int p,q,j=0;
     FILE *j_file;
     // hard coding the texture to write to file (the winning motion vectors
     so far)
     latestTextureUpdated=jTextureMVdata;  // destination to write to
     glFramebufferTexture2DEXT(GL_FRAMEBUFFER_EXT, GL_COLOR_ATTACHMENT3_EXT,
     GL_TEXTURE_2D, jTextureMVdata, 0);
     destinationAttachment=GL_COLOR_ATTACHMENT3_EXT;
     glDrawBuffer(destinationAttachment);
     // FYI information
     if(debug)      printf("Saving current MVdata :
     \n");checkFramebufferStatus();
     // bind the motion vector texture as input
     glBindTexture(GL_TEXTURE_2D, jTextureMVdata);
     glEnable(GL_TEXTURE_2D);
    glPolygonMode(GL_FRONT,GL_FILL);
     // Draw the window sized quad
    glBegin(GL_QUADS);
    {
       glTexCoord2f(0, 0); glVertex3f(-1, -1, -0.5f);
        glTexCoord2f(1, 0); glVertex3f( 1, -1, -0.5f);
        glTexCoord2f(1, 1); glVertex3f( 1,  1, -0.5f);
        glTexCoord2f(0, 1); glVertex3f(-1,  1, -0.5f);
    }
    glEnd();

     // Copy the buffer to a data array
     // It will arrive in a 90 degree rotation
     glReadBuffer(destinationAttachment);
     // Copy the pixel data into the array MVdata
     glReadPixels(0, 0, intImageWidth,
     intImageHeight,GL_RGBA,GL_UNSIGNED_BYTE,MVdata);
     glFinish();

     // Open the ouput file for writing
     if( (j_file = fopen(gpuMVfile, "w"))==NULL) return ;

     // Looping by row (max to 0) and column (0 to max) so that the 90
     degree rotation can be comprehended
    for (p=reducedInfoSize-1; p>=0; p--)
    {
       for (q=0; q<reducedInfoSize; q++)
       {
              //Perform sanity checks to ensure MVs destined for outside the
     images are not used
              if ( (p*scale)-offsetArrayY[MVdata[p][q][2]]<0 || (p*scale)-
     offsetArrayY[MVdata[p][q][2]]>checkImageHeight-scale
                     || (q*scale)+offsetArrayX[MVdata[p][q][2]]<0 ||
     (q*scale)+offsetArrayX[MVdata[p][q][2]]>checkImageWidth)
              {
                     if(debug)     printf("Taking out GPU value %d,%d which
     is
     %d,%d\n",p,q,offsetArrayX[MVdata[p][q][2]],offsetArrayY[MVdata[p][q][2]
     ]);
                     MVdata[p][q][2]=int(sizeof(offsetArrayX) /
     sizeof(offsetArrayX[0] )-1); // last MV item (Zero)
              }
              fprintf(j_file,"%d,%d
     \t",offsetArrayX[MVdata[p][q][2]],offsetArrayY[MVdata[p][q][2]]);
       }
    fprintf(j_file,"\n");
```

GpuCpuVideo.cpp, continued.

```
    }
    fclose(j_file); // Closes the file stream
     // FYI information
     if(debug)      printf("Finished writing the Motion Vectors to file\n");

}    // end of saveMVdata
//---------------------------------------------------------------------



/**
 * saveStats
 *
 * Write the performance data to file so as to
 * track the performance of the program (GPU section)
 *
 *
 * @param       void
 * @return      void
 *
 * @author      Jason Ruane
 * @version     1.0
 */
void saveStats(void)
{
     // Local file handle required
     FILE *j_file;
     if( (j_file = fopen(resultsFilename, "a+"))==NULL)
     {
       printf("Could not create the results file");
       exit(0);
     }

     if (debug)     printf("Saving performance results to file.\n");
     fprintf(j_file,"\n\nStart of results -------------\n");
     fprintf(j_file,"Input files:%s and %s\n",inputFile1,inputFile2);
     fprintf(j_file,"GPU performance:\n");
     fprintf(j_file,"%.6f seconds for %d
     iterations.\n",timeForUpdate/CLOCKS_PER_SEC,testsToRun);
     float secondsPerIteration = timeForUpdate/CLOCKS_PER_SEC/testsToRun;
     fprintf(j_file,"%.6f seconds per iteration.(%.1f
     FPS)\n",secondsPerIteration, 1.0/secondsPerIteration);
     fprintf(j_file,"Using a search window of %d
     positions.\n",sizeof(offsetArrayX) / sizeof(offsetArrayX[0]));
     float imagePermutationsPerSecond = (sizeof(offsetArrayX) /

       sizeof(offsetArrayX[0]))/(timeForUpdate/CLOCKS_PER_SEC/testsToRun);
     fprintf(j_file,"%.3f full image permutations per second
     evaluated.\n",imagePermutationsPerSecond);
     fprintf(j_file,"Block size of %.0f in an image of size:%d by
     %d.\n",scale,int(checkImageWidth),
                                        int(checkImageHeight));
     float numberOfBlocks =
     (checkImageWidth/scale)*(checkImageHeight*scale);
     fprintf(j_file,"%.0f block permutations per second
     evaluated.\n",imagePermutationsPerSecond*numberOfBlocks);

     fclose(j_file); // Close the file stream

}    // end of saveStats
//---------------------------------------------------------------------
```

GpuCpuVideo.cpp, continued.

```
/**
 * reshape
 *
 * Standard GLUT function required for when GLUT
 * reacts to a resizing of the window
 *
 *
 * @param       w   Width of the window in pixels
 * @param       h   Height of the window in pixels
 * @return      void
 *
 * @author      Jason Ruane
 * @version     1.0
 */
void reshape(int w, int h)
{
     // Avoid division by zero
    if (h == 0) h = 1;
     // Set the viewport
    glViewport(0, 0, w, h);
     glMatrixMode(GL_PROJECTION);
    glLoadIdentity();
    gluOrtho2D(-1, 1, -1, 1);
    glMatrixMode(GL_MODELVIEW);
    glLoadIdentity();
}    // end of reshape(int w, int h)
//-----------------------------------------------------------------------



/**
 * cgErrorCallback
 *
 * Standard Cg function to facilitate the reporting of errors
 * experienced in the Cg environment to the user.
 * Only called when Cg experiences an error.
 *
 *
 * @param       void
 * @return      void
 *
 * @author      Jason Ruane
 * @version     1.0
 */
void cgErrorCallback(void)
{
  CGerror lastError = cgGetError();
  if(lastError)
  {
    printf("%s\n\n", cgGetErrorString(lastError));
    printf("%s\n", cgGetLastListing(g_cgContext));
    printf("Cg error!\n");
  }
}    // end of cgErrorCallback
//-----------------------------------------------------------------------



/**
 * loadBMP
 *
 * Load the first input file (256*256 bitmap) which
 * is the origin video frame and store in the checkImage array.
 *
 *
```

GpuCpuVideo.cpp, continued.

```
 * @param       void
 * @return      void
 *
 * @author      Jason Ruane
 * @version     1.0
 */
void loadBMP(void)
{
     //Local variables
     int p,q,i, j=0;
     FILE *l_file;
     BITMAPFILEHEADER fileheader;
     BITMAPINFOHEADER infoheader;
     RGBTRIPLE rgb;

     // Open the file for reading
     if( (l_file = fopen(inputFile1, "rb"))==NULL)
     {
       // Inform user
       printf("Input file1 not available\n");
       // Terminate program
       exit(0);
     }
     // Read the bitmap header
     fread(&fileheader, sizeof(fileheader), 1, l_file);
     fseek(l_file, sizeof(fileheader), SEEK_SET);
     fread(&infoheader, sizeof(infoheader), 1, l_file);
     // Only read a maximum of texture size worth of input data
     int maxWidth,maxHeight;
     if(checkImageWidth<infoheader.biWidth)
       maxWidth=checkImageWidth;
     else
       maxWidth=infoheader.biWidth;
     if(checkImageHeight<infoheader.biHeight)
       maxHeight=checkImageHeight;
     else
       maxHeight=infoheader.biHeight;

     // Read the data
     for (i=0; i < maxWidth*maxHeight; i++)
     {
       fread(&rgb, sizeof(rgb), 1, l_file);
       // Convert into a 2D array, the position of current pixel
       p=i/checkImageWidth;
       q=i%checkImageWidth;
       // Store data into the array
       checkImage[p][q][2] = (GLubyte) rgb.rgbtRed;
       checkImage[p][q][1] = (GLubyte) rgb.rgbtGreen;
       checkImage[p][q][0] = (GLubyte) rgb.rgbtBlue;
       // Setting the alpha plane to opaque
       checkImage[p][q][3] = (GLubyte) 255;
     }
     fclose(l_file); // Close the file stream

     // save the ASCII contents of the first input file in pgm format
     saveInputFile1();

}    // end of loadBMP
//--------------------------------------------------------------------
```

GpuCpuVideo.cpp, continued.

```
/**
 * loadBMP2
 *
 * Load the second input file (256*256 bitmap) which
 * is the origin video frame and store in the checkImageTwo array.
 *
 *
 * @param       void
 * @return      void
 *
 * @author      Jason Ruane
 * @version     1.0
 */
void loadBMP2(void)
{
     //Local variables
     int p,q,i, j=0;
     FILE *l_file;
     BITMAPFILEHEADER fileheader;
     BITMAPINFOHEADER infoheader;
     RGBTRIPLE rgb;

     // Open the file for reading
     if( (l_file = fopen(inputFile2, "rb"))==NULL)
     {
       // Inform user
       printf("Input file2 not available\n");
       // Terminate program
       exit(0);
     }

     // Read the bitmap header
     fread(&fileheader, sizeof(fileheader), 1, l_file);
     fseek(l_file, sizeof(fileheader), SEEK_SET);
     fread(&infoheader, sizeof(infoheader), 1, l_file);

     // Only want a maximum of texture size worth of input data
     int maxWidth,maxHeight;
     if(checkImageWidth<infoheader.biWidth)
       maxWidth=checkImageWidth;
     else
       maxWidth=infoheader.biWidth;
     if(checkImageHeight<infoheader.biHeight)
       maxHeight=checkImageHeight;
     else
       maxHeight=infoheader.biHeight;


     // Read the data
     for (i=0; i < maxWidth*maxHeight; i++)
     {
       fread(&rgb, sizeof(rgb), 1, l_file);
       // Convert into a 2D array, the position of current pixel
       p=i/checkImageWidth;
       q=i%checkImageWidth;
       // Store data into the array
       checkImageTwo[p][q][2] = (GLubyte) rgb.rgbtRed;
       checkImageTwo[p][q][1] = (GLubyte) rgb.rgbtGreen;
       checkImageTwo[p][q][0] = (GLubyte) rgb.rgbtBlue;
       // Setting the alpha plane to opaque
       checkImageTwo[p][q][3] = (GLubyte) 255;
     }
     fclose(l_file); // Closes the file stream
```

GpuCpuVideo.cpp, continued.

```
     // save the ASCII contents of the second input file to pgm format
     saveInputFile2();

}    // end of loadBMP2
//----------------------------------------------------------------------



/**
 * checkFramebufferStatus
 *
 * Standard OpenGL function to check the status of the framebuffer
 *
 *
 * @param       void
 * @return      bool       0=Success, 1=Fail
 *
 * @author      Jason Ruane, as per official OpenGL definitions
 * @see                    http://oss.sgi.com/projects/ogl-
     sample/registry/EXT/framebuffer_object.txt
 * @version     1.0
 */
bool checkFramebufferStatus() {
    GLenum status;
    status=(GLenum)glCheckFramebufferStatusEXT(GL_FRAMEBUFFER_EXT);
    switch(status) {
        case GL_FRAMEBUFFER_COMPLETE_EXT:
            return true;
        case GL_FRAMEBUFFER_INCOMPLETE_ATTACHMENT_EXT:
            printf("Framebuffer incomplete,incomplete attachment\n");
            return false;
        case GL_FRAMEBUFFER_UNSUPPORTED_EXT:
            printf("Unsupported framebuffer format\n");
            return false;
        case GL_FRAMEBUFFER_INCOMPLETE_MISSING_ATTACHMENT_EXT:
            printf("Framebuffer incomplete,missing attachment\n");
            return false;
        case GL_FRAMEBUFFER_INCOMPLETE_DIMENSIONS_EXT:
            printf("Framebuffer incomplete,attached images must have same
     dimensions\n");
            return false;
        case GL_FRAMEBUFFER_INCOMPLETE_FORMATS_EXT:
             printf("Framebuffer incomplete,attached images must have same
     format\n");
            return false;
        case GL_FRAMEBUFFER_INCOMPLETE_DRAW_BUFFER_EXT:
            printf("Framebuffer incomplete,missing draw buffer\n");
            return false;
        case GL_FRAMEBUFFER_INCOMPLETE_READ_BUFFER_EXT:
            printf("Framebuffer incomplete,missing read buffer\n");
            return false;
    }
     return false;
}    // end of checkFramebufferStatus
//----------------------------------------------------------------------
```

GpuCpuVideo.cpp, continued.

```
/**
 * saveInputFile1
 *
 * Save a pgm format copy of the input file1 being used
 *
 *
 * @param       void
 * @return      bool       0=Success, 1=Fail
 *
 * @author      Jason Ruane, as per official OpenGL definitions
 * @see                    http://oss.sgi.com/projects/ogl-
     sample/registry/EXT/framebuffer_object.txt
 * @version     1.0
 */
void saveInputFile1(void)
{
     // Local variables
     int p,q,j=0;
     FILE *j_file;
     // Open file for writing
     if( (j_file = fopen(outputFile1, "w"))==NULL) return ;
     // Print file header
     fprintf(j_file,"P2 %d %d 255\n",checkImageWidth,checkImageHeight);
     // Print the data to file
     // looping by row (max to 0) and column (0 to max) so that the 90
     degree rotation can be comprehended
    for (p=checkImageHeight-1; p>=0; p--)
    {
       for (q=0; q<checkImageWidth; q++)
              //Writing a single 8bit number per pixel to file
              fprintf(j_file,"%d ",checkImage[p][q][0]);
    fprintf(j_file,"\n");
    }
    fclose(j_file); // Close the file stream

}    // end of saveInputFile1
//---------------------------------------------------------------------



/**
 * saveInputFile2
 *
 * Save a pgm format copy of the input file2 being used
 *
 *
 * @param       void
 * @return      bool       0=Success, 1=Fail
 *
 * @author      Jason Ruane, as per official OpenGL definitions
 * @see                    http://oss.sgi.com/projects/ogl-
     sample/registry/EXT/framebuffer_object.txt
 * @version     1.0
 */
void saveInputFile2(void)
{
     // Local variables
     int p,q,j=0;
     FILE *j_file;
     // Open file for writing
     if( (j_file = fopen(outputFile2, "w"))==NULL) return ;
     // Print file header
     fprintf(j_file,"P2 %d %d 255\n",checkImageWidth,checkImageHeight);
     // Print the data to file
```

GpuCpuVideo.cpp, continued.

```
     // looping by row (max to 0) and column (0 to max) so that the 90
     degree rotation can be comprehended
    for (p=checkImageHeight-1; p>=0; p--)
    {
       for (q=0; q<checkImageWidth; q++)
              //Writing a single 8bit number per pixel to file
              fprintf(j_file,"%d ",checkImageTwo[p][q][0]);
    fprintf(j_file,"\n");
    }
    fclose(j_file); // Close the file stream

}    // end of saveInputFile2



/**
 * saveInputFile2
 *
 * Save a pgm format copy of the constructed image using
 * the input file1 and the motion vectors
 *
 *
 * @param       void
 * @return      bool       0=Success, 1=Fail
 *
 * @author      Jason Ruane, as per official OpenGL definitions
 * @see                    http://oss.sgi.com/projects/ogl-
     sample/registry/EXT/framebuffer_object.txt
 * @version     1.0
 */
void saveConstructedImage(void)
{
     // Local variables
     int p,q,mvx,mvy,mvOffset,mvxNew,mvyNew,j=0;
     FILE *j_file;
     // Open file for writing
     if( (j_file = fopen(outputFile3, "w"))==NULL) return ;
     // Print the header section
     fprintf(j_file,"P2 %d %d 255\n",checkImageWidth,checkImageHeight);
     // Print the data to file
     for (p=checkImageHeight-1;p>=0;p--)
    {
       for (q=0; q<checkImageWidth; q++)
       {
              // find the respective block
              mvy=int(floor(p/scale));
              mvx=int(floor(q/scale));
              // find the respective entry in the MV data
              mvOffset=MVdata[mvy][mvx][2];
              // find the actual offset reccommended by MV data
              mvyNew = p-offsetArrayY[mvOffset];
              mvxNew = q+offsetArrayX[mvOffset];
              // output the actual pixel values
              fprintf(j_file,"%d ",checkImage[mvyNew][mvxNew][0] );
              constructedImage[p][q][0]=checkImage[mvyNew][mvxNew][0];
       }
       fprintf(j_file,"\n");
    }
    fclose(j_file); // Close the file stream

}    // end of saveInputFile2
```

GpuCpuVideo.cpp, continued.

```
/**
 * saveImageDiff1
 *
 * Save a pgm format copy of the diference between
 * input image 1 and input image 2
 *
 *
 * @param       void
 * @return      bool       0=Success, 1=Fail
 *
 * @author      Jason Ruane, as per official OpenGL definitions
 * @see                    http://oss.sgi.com/projects/ogl-
     sample/registry/EXT/framebuffer_object.txt
 * @version     1.0
 */
void saveImageDiff1(void)
{
     // Local variables
     int p,q,j=0;
     FILE *j_file;
     // Open the file for writing
     if( (j_file = fopen(outputFile4, "w"))==NULL) return ;
     // Header information
     fprintf(j_file,"P2 %d %d 255\n",checkImageWidth,checkImageHeight);
     // Data
     // looping by row (max to 0) and column (0 to max) so that the 90
     degree rotation can be comprehended
    for (p=checkImageHeight-1; p>=0; p--)
    {
       for (q=0; q<checkImageWidth; q++)
              //Writing a single 8bit number to file per pixel
              fprintf(j_file,"%d ",abs(checkImage[p][q][0]-
     checkImageTwo[p][q][0]));
       fprintf(j_file,"\n");
     }
    fclose(j_file); // Close the file stream
}    // end of saveImageDiff1



/**
 * saveImageDiff1
 *
 * Save a pgm format copy of the diference between
 * input image 2 and the constructed image
 *
 *
 * @param       void
 * @return      bool       0=Success, 1=Fail
 *
 * @author      Jason Ruane, as per official OpenGL definitions
 * @see                    http://oss.sgi.com/projects/ogl-
     sample/registry/EXT/framebuffer_object.txt
 * @version     1.0
 */
void saveImageDiff2(void)
{
     // Local variables
     int p,q,j=0;
     FILE *j_file;
     // Open the file for writing
     if( (j_file = fopen(outputFile5, "w"))==NULL) return ;
     // The header section
     fprintf(j_file,"P2 %d %d 255\n",checkImageWidth,checkImageHeight);
     // The data
```

GpuCpuVideo.cpp, continued.

```
     // looping by row (max to 0) and column (0 to max) so that the 90
     degree rotation can be comprehended
    for (p=checkImageHeight-1; p>=0; p--)
    {
       for (q=0; q<checkImageWidth; q++)
              //Writing a single 8bit number to file
              fprintf(j_file,"%d ",abs(checkImageTwo[p][q][0]-
     constructedImage[p][q][0]));
       fprintf(j_file,"\n");
    }
    fclose(j_file); // Close the file stream
}    // end of saveImageDiff2



/**
 * evaluateImageDiffs
 *
 * Calculate the PSNR ratios and SAD values, recording them to file
 *
 *
 * @param       void
 * @return      bool       0=Success, 1=Fail
 *
 * @author      Jason Ruane, as per official OpenGL definitions
 * @see                    http://oss.sgi.com/projects/ogl-
     sample/registry/EXT/framebuffer_object.txt
 * @version     1.0
 */
void evaluateImageDiffs(void)
{
     // Local variables
     int p,q;
     float itemCounter=0,MSEsum=0,MSE,SADsum=0,tempResult;
     FILE *j_file;
     // FYI information
     if(debug)      printf("Evaluating the difference images\n");
     // Open the file for writing
     if( (j_file = fopen(resultsFilename, "a"))==NULL)
     {
       // Inform user
       printf("Could not open the results file");
       // Terminate program
       exit(0);
     }

     // The diference between input image 1 and input image 2
     // Calculate the MeanSquareError and SumAbsoluteDifference
    for (p=checkImageHeight-1; p>=0; p--)
    {
       for (q=0; q<checkImageWidth; q++)
       {
              tempResult = abs(checkImage[p][q][0]-checkImageTwo[p][q][0]);
              MSEsum += tempResult*tempResult;
              SADsum += tempResult;
              itemCounter++;
       }
    }
     // MSE only has the sum so far, divide by order to get mean
     MSE=MSEsum/itemCounter;
     // Print to file
     fprintf(j_file,"Image difference 1 (image 1 and 2):");
     // Avoiding division by zero in PSNR calculation
     if(MSE == 0)
     {
```

GpuCpuVideo.cpp, continued.

```
      fprintf(j_file,"PSNR: Infinity, SAD:0, both images are identical");
     }
     else
     {
       // PSNR formula used is:
       /* PSNR=10*log10( Max_Gray_Value*Max_Gray_Value / MSE ); */
       fprintf(j_file,"PSNR: %.3f (dB), ", 10.0*log10((255.0*255.0)/MSE));
       fprintf(j_file,"SAD: %.0f\n", SADsum);
     }

     //-----------------------------------------------------//
     // The diference between input image 2 and the constructed image
    // reset counters
     itemCounter=0,MSEsum=0;SADsum=0;
     // Calculate the MeanSquareError and SumAbsoluteDifference
     for (p=checkImageHeight-1; p>=0; p--)
    {
       for (q=0; q<checkImageWidth; q++)
       {
              tempResult = abs(checkImageTwo[p][q][0]-
     constructedImage[p][q][0]);
              MSEsum += tempResult*tempResult;
              SADsum += tempResult;
              itemCounter++;
       }
    }
     // MSE only has the sum so far, divide by order to get mean
     MSE=MSEsum/itemCounter;
     // Print to file
     fprintf(j_file,"Image difference 2 (image 2 and constructed):");
     // Avoiding division by zero in PSNR calculation
     if(MSE == 0)
     {
       fprintf(j_file,"PSNR: Infinity, SAD:0, both images are identical");
     }
     else
     {
       // PSNR formula used is:
       /* PSNR=10*log10( Max_Gray_Value*Max_Gray_Value / MSE ); */
       fprintf(j_file,"PSNR: %.3f (dB), ", 10.0*log10((255.0*255.0)/MSE));
       fprintf(j_file,"SAD: %.0f\n", SADsum);
     }

}    // end of evaluateImageDiffs



/**
 * CPUmotionEstimation
 *
 * Perform the motion estimation using the CPU
 * in a similar fashion to the prior GPU method
 *
 * @param       void
 * @return      bool       0=Success, 1=Fail
 *
 * @author      Jason Ruane, as per official OpenGL definitions
 * @see                    http://oss.sgi.com/projects/ogl-
     sample/registry/EXT/framebuffer_object.txt
 * @version     1.0
 */
void CPUmotionEstimation(void)
{
     // FYI information
     if(debug)      printf("Starting CPU based motion estimation\n");
```

GpuCpuVideo.cpp, continued.

```
    // Local variables required
    int p, q, matchCounter=0, totalCounter=0;
    int cpuMVarray[reducedInfoSize][reducedInfoSize];
    int cpuMVarrayTemp[reducedInfoSize][reducedInfoSize];
    int cpuMVarrayWinningSum[reducedInfoSize][reducedInfoSize];
    int currentBlockX, currentBlockY, currentDiff, currentMVindex=0;
    int numberOfOffsets=sizeof(offsetArrayX)/sizeof(offsetArrayX[0]);
    FILE *j_file;
    double timeCPUstart = clock();


    // Loop through the number of tests to perform for increasing the
    timing accuracy
    for(int cpuIteration=0;cpuIteration<testsToRun;cpuIteration++)
    {
      // FYI information
      printf("CPU iteration %d of %d.\n",cpuIteration+1,testsToRun);

      // Clear the arrays before starting
      for (currentBlockY=reducedInfoSize-1; currentBlockY>=0;
    currentBlockY--)
      {
            for (currentBlockX=0; currentBlockX<reducedInfoSize;
    currentBlockX++)
            {
            cpuMVarrayWinningSum[currentBlockX][currentBlockY] = 255;
            cpuMVarray[currentBlockX][currentBlockY]=numberOfOffsets;
            }
      }


      // Loop through all the offset positions
      for(currentMVindex=0;currentMVindex <
    numberOfOffsets;currentMVindex++)
      {
            // Clear the array before starting
            for (currentBlockY=reducedInfoSize-1; currentBlockY>=0;
    currentBlockY--)
            {
                  for (currentBlockX=0; currentBlockX<reducedInfoSize;
    currentBlockX++)
                  {
                  cpuMVarrayTemp[currentBlockX][currentBlockY] = 0;
                  }
            }

            // Calculate the image difference at the current offset
            for (p=checkImageHeight-1; p>=0; p--)
            {
                  for (q=0; q<checkImageWidth; q++)
                  {
                  currentBlockY=q/scale;
                  currentBlockX=p/scale;
                  cpuMVarrayTemp[currentBlockX][currentBlockY] +=
                        abs(checkImage[p-
    offsetArrayY[currentMVindex]][q+offsetArrayX[currentMVindex]][0]
                              - checkImageTwo[p][q][0]);
                  }
            }

            // Stepping through each row and cloumn within,
            for (currentBlockY=reducedInfoSize-1; currentBlockY>=0;
    currentBlockY--)
            {
```

GpuCpuVideo.cpp, continued.

```
                for (currentBlockX=0; currentBlockX<reducedInfoSize;
currentBlockX++)
                {
                        // Find the image difference for this block
position
                        // The next division and minimisation are to
reflect the 8 bit truncation of GPU setup

  currentDiff=min(int((float(cpuMVarrayTemp[currentBlockX][currentBlock
Y])

  /float(truncationScale))+0.5), 255);
                        // Examine whether this is a winning offset
candidate
                        if(currentDiff <=
cpuMVarrayWinningSum[currentBlockX][currentBlockY])
                        {
                                // Record the offset and sum if it is a
winning candidate

  cpuMVarrayWinningSum[currentBlockX][currentBlockY] = currentDiff;
                                cpuMVarray[currentBlockX][currentBlockY] =
currentMVindex;
                        }
                }
        } // end of loop stepping through each row and column

  } // end of for loop through offset positions

  // Allow the CPU to pause if variable is set
  Sleep(SleepTime);

} // end cpuIterations loop


// Record the finish time for CPU timing calculation
double timeCPUfinish = clock();

// Save the CPU MV array to file
// Open file for writing
if( (j_file = fopen(cpuMVfile, "w"))==NULL)
{
  // Inform the user
  printf("Could not open the results file");
  // Terminate the program
  exit(0);
}
// Loop through the data strucutre, in rows and columns
for (p=reducedInfoSize-1; p>=0; p--)
{
  for (q=0; q<reducedInfoSize; q++)
  {

        //Perform sanity checks to ensure MVs destined for outside the
images are not used
        if ( (p*scale)-offsetArrayY[MVdata[p][q][2]]<0 || (p*scale)-
offsetArrayY[MVdata[p][q][2]]>checkImageHeight-scale
                || (q*scale)+offsetArrayX[MVdata[p][q][2]]<0 ||
(q*scale)+offsetArrayX[MVdata[p][q][2]]>checkImageWidth)
        {
                if(debug)    printf("Taking out CPU value %d,%d which
is
%d,%d\n",p,q,offsetArrayX[MVdata[p][q][2]],offsetArrayY[MVdata[p][q][2]
]);
```

GpuCpuVideo.cpp, continued.

```
                MVdata[p][q][2]=int(sizeof(offsetArrayX) /
    sizeof(offsetArrayX[0] )-1); // last MV item (Zero)
            }
            fprintf(j_file,"%d,%d
    \t",offsetArrayX[cpuMVarray[p][q]],offsetArrayY[cpuMVarray[p][q]]);

      }

      fprintf(j_file,"\n");  // Finished writing one row

    }
     // Close the file
     fclose(j_file);


     // Save the CPU performance data to the results file
     // This time appending, as the GPU version has already written to the
     file
     // open the file
     if( (j_file = fopen(resultsFilename, "a+"))==NULL)
     {
       // Inform user
       printf("Could not create the results file");
       // Terminate program
       exit(0);
     }

     // FYI information
     if (debug)     printf("Saving CPU performance results to file.\n");
     // Write performance information to file
     fprintf(j_file,"CPU performance:\n%.6f seconds for %d iterations\n",
                          (timeCPUfinish-
     timeCPUstart)/CLOCKS_PER_SEC,testsToRun);
     float secondsPerIteration = (timeCPUfinish-
     timeCPUstart)/CLOCKS_PER_SEC/testsToRun;
     fprintf(j_file,"%.6f seconds per iteration.(%.1f
     FPS)\n",secondsPerIteration, 1.0/secondsPerIteration);
     fprintf(j_file,"----------------------------------------------------
     \n");

     // Record the correlation between MVs found by CPU and GPU methods
     // Stepping through each row and column
     for (p=reducedInfoSize-1; p>=0; p--)
    {
       for (q=0; q<reducedInfoSize; q++)
       {
              if(    cpuMVarray[p][q] == int(MVdata[p][q][2]) )
              {
                     // Increment the counter for matched items
                     matchCounter++;
              }
              else
              {
                     // FYI information
                     if(debug)     printf("not matched at %d,%d  cpu:%d
     versus gpu:%d with a \
                                  CPUsum of %d
     \n",p,q,cpuMVarray[p][q],int(MVdata[p][q][2]),cpuMVarrayWinningSum[p][q
     ] );
              }
              // Increment the counter of items examined
              totalCounter++;
       }
     }
```

GpuCpuVideo.cpp, continued.

```
     // Print to file the summary of matching items
     fprintf(j_file,"Motion Vectors matched in %d of %d
     instances.\n",matchCounter,totalCounter);

     fclose(j_file); // Close the file stream

} // End of CPUmotionEstimation
```

7.4.3 Appendix 4.3 fragmentShaderSum8*8.cg

The following code is from fragmentShaderSum8*8.cg. It is the Cg code to accompany the Motion Vector program. It is designed to calculate the sum of values in an 8*8 pixel region.

```
/**
 * fragmentShaderSum8*8.cg.
 *
 * This program calculates the sum of an 8*8 pixel region.
 *
 * @return      A half4 value relating to colour planes for 1 pixel
 *
 * @author      Jason Ruane, DIT Bolton St. Dublin, Ireland. B773.2006.
 * @version     1.0
 */

half4 edges(float2 coords : TEX0,  //:  TEX0 for clamped version of
     coordinates, WPOS for integer
            uniform sampler2D texture,
            uniform sampler2D texture2,
            uniform sampler2D textureDiff,
            uniform half4 myDistance,
            uniform float4 myCgParameter) : COLOR
{

float offsetX =     myCgParameter[0];   // The X offset, from 0 to 7
     // using the opposite from origin on the Y axis to maintain top left
     origin
float offsetY =     myCgParameter[1];   // The Y offset, from 0 to 7
float imageWidth =  myCgParameter[2];   // The image width
float imageHeight = myCgParameter[3];   // The image height

float  pbufwidth =imageWidth-1.0; // needed to stop coords.y showing up 156
     twice in a row

// we are using a scaled image here 1/block size in X and Y
// multiply the current position by the image size to get an integer
     position
float elem2dx = floor(coords.x*255.0)-3;
float elem2dy = floor(coords.y*255.0)+4;

//return(tex2D(textureDiff,half2((elem2dx)/255.0,(elem2dy)/255.0)) ); //
     origin top left pixel


// ----------  Calculate the summation of the 8*8 block this pixel
     represents
float4 val01=half4(0,0,0,0);
float i=0;
//   sums pixel values to the right and down from the origin pixel
for(i=0;i<8;i++) // scan across horizontally for 8 columns
{
     // add up each vertical row of 8 pixels
val01+=tex2D(textureDiff,half2((elem2dx+i)/255.0,(elem2dy-0)/255.0));
val01+=tex2D(textureDiff,half2((elem2dx+i)/255.0,(elem2dy-1)/255.0));
val01+=tex2D(textureDiff,half2((elem2dx+i)/255.0,(elem2dy-2)/255.0));
val01+=tex2D(textureDiff,half2((elem2dx+i)/255.0,(elem2dy-3)/255.0));
val01+=tex2D(textureDiff,half2((elem2dx+i)/255.0,(elem2dy-4)/255.0));
val01+=tex2D(textureDiff,half2((elem2dx+i)/255.0,(elem2dy-5)/255.0));
val01+=tex2D(textureDiff,half2((elem2dx+i)/255.0,(elem2dy-6)/255.0));
val01+=tex2D(textureDiff,half2((elem2dx+i)/255.0,(elem2dy-7)/255.0));
}
```

GpuCpuVideo.cpp, continued.

```
// Eliminate off image MV directions by discouraging with a large summation
// 255*64 < 17000
if(elem2dx<1 && offsetX<0)
     val01=17000;
if(elem2dy<8 && offsetY>0)
     val01=17000;
if(elem2dx>imageWidth-10 && offsetX>0)
     val01=17000;
if(elem2dy>imageHeight-2 && offsetY<0)
     val01=17000;
return( floor( floor(val01*255.0 +0.5  )/8.0 +0.5 )/255.0);
//return( (val01*255.0/8.0 )/255.0 ); // truncationScale divisor to avoid
     255 ceiling
// ----------------   Finished summation of 8*8 block

} // end of pixel shader function
```

7.4.4 Appendix 4.4 fragmentShaderDiff.cg

The following code is from fragmentShaderDiff.cg. It is the Cg code to accompany the Motion Vector program (phase 2). It is designed to calculate the image difference of two input textures/images.

```
/**
 * fragmentShaderDiff.cg.
 *
 * This program calculates an image difference of two images.
 *
 * @return      A half4 value relating to colour planes for 1 pixel
 *
 * @author      Jason Ruane, DIT Bolton St. Dublin, Ireland. B773.2006.
 * @version     1.0
 */

 half4 edges(float2 coords : TEX0,  //:  TEX0 for clamped version of
     coordinates, WPOS for integer
            uniform sampler2D texture,
            uniform sampler2D texture2,
            uniform sampler2D textureDiff,
            uniform float4 myCgParameter) : COLOR
{


float offsetX =     myCgParameter[0];   // The X offset, from 0 to 7
     // using the opposite from origin on the Y axis to maintain top left
     origin
float offsetY =     myCgParameter[1];   // The Y offset, from 0 to 7
float imageWidth =  myCgParameter[2];   // The image width
float imageHeight = myCgParameter[3];   // The image height

float  pbufwidth =imageWidth-1.0; // needed to stop coords.y showing up 156
     twice in a row

                                  // The floor is required to make it start
     at zero and reach 255
int elem2dx = floor(coords.x*imageWidth);      // NOTE: floor here messes
     the texture addressing up
int elem2dy = floor(coords.y*imageWidth);      // NOTE: floor here messes
     the texture addressing up


// ----------  Calculate the image subtraction, 1 pixel at a time
// finding texture2-texture  (texture will have offset applied to account
     for M.E. searching
//
float elemUnit= 1.0/imageWidth;  // this was pbufwidth, but reccommend it
     is imageWidth
float4 val01 = tex2D(texture,half2( (elem2dx+offsetX)/pbufwidth,(elem2dy-
     offsetY)/pbufwidth ));
float4 val02 = tex2D(texture2,half2( elem2dx/pbufwidth,elem2dy/pbufwidth ));
return(abs(val02 - val01));
// ----------------   Finished image difference
}
```

7.4.5 Appendix 4.5 fragmentShaderBlockSums.cg

The following code is from fragmentShaderBlockSums.cg. It is the Cg code to accompany the Motion Vector program (phase 3). It is designed to compare two block sums and record the winning sum.

```
/**
 * fragmentShaderBlockSums.cg
 *
 * This program compares two block sums and records the winning
 *     i.e. lowest running total.
 *
 * @return      A half4 value relating to colour planes for 1 pixel
 *
 * @author      Jason Ruane, DIT Bolton St. Dublin, Ireland. B773.2006.
 * @version     1.0
 */

 half4 edges(float2 coords : TEX0,  //:  TEX0 for clamped version of
     coordinates, WPOS for integer
            uniform sampler2D textureSums,
            uniform sampler2D textureBlockSumsWinning,
            uniform half4 myDistance,
            uniform float4 myCgParameter) : COLOR
{

float4 currentVal = tex2D(textureSums,half2(coords.x,coords.y));
float4 winningVal = tex2D(textureBlockSumsWinning,half2(coords.x,coords.y));

float4 myReturnVal= ( currentVal[2] <= winningVal[2] ) ?
     currentVal:winningVal;

return( myReturnVal );

} // end of pixel shader function
```

7.4.6 Appendix 4.6 fragmentShaderMVdata.cg

The following code is from fragmentShaderMVdata.cg. It is the Cg code to accompany the Motion Vector program (phase 4). It is designed to compare two block sums and record the index of the winning offset.

```
/**
 * fragmentShaderMVdata.cg
 *
 * This program compares two block sums and records the
 *     i.e. index of the winning offset
 *
 * @return      A half4 value relating to colour planes for 1 pixel
 *
 * @author      Jason Ruane, DIT Bolton St. Dublin, Ireland. B773.2006.
 * @version     1.0
 */

 half4 edges(float2 coords : TEX0,  //:  TEX0 for clamped version of
     coordinates, WPOS for integer
            uniform sampler2D textureBlockSumsCurrent,
            uniform sampler2D textureBlockSumsWinning,
            uniform sampler2D textureMVdata,
            uniform half4 myDistance,
            uniform float4 myCgParameter) : COLOR
{
//myCgParameter[0]=index of the motion vector offset

float4 currentVal = tex2D(textureBlockSumsCurrent,half2(coords.x,coords.y));
float4 winningVal = tex2D(textureBlockSumsWinning,half2(coords.x,coords.y));
float4 MVdata = tex2D(textureMVdata,half2(coords.x,coords.y));
float4 returnVal;

if( currentVal[2] <= winningVal[2] )
     {returnVal = half4(0,0,myCgParameter[0]/255.0,0);}
else
     {returnVal = MVdata;}


return( returnVal );

} // end of pixel shader function
```

8. Glossary

AGP	Accelerated Graphics Port
AI	Artificial Inteligence
API	Application Programming Interface
ARB	Architecture Review Board
Cg	C for graphics
CPU	Central Processing Unit
DCT	Discrete Cosine Transform
EBMA	Exhaustive Block Matching Algorithm
FBO	Frame Buffer Object
FPS	Frames Per Second
GFLOP	Giga Foating Logic Operations
GLEW	OpenGL Extension Wrangler Library
GLUT	OpenGL Utility Toolkit
GNU	GNU's Not Unix
GPGPU	General Purpose Computing on a Graphics Processing Unit
GPU	Graphics Processing Unit
IDCT	Inverse Discrete Cosine Transform
MAD	Mean Absolute Difference
MIMD	Multiple Instruction, Multiple Data
MSE	Mean Square Error
PC	Personal Computer
PCI	Peripheral Component Interconnect
PPU	Physics Processing Unit
PSNR	Peak Signal to Noise Ratio.

$$\mathrm{PSNR} = 10 \log_{10} \frac{(f_{\max} - f_{\min})^2}{\mathrm{MSE}},$$

(source: www[DUB])

RAM	Random Access Memory
SAD	Sum Absolute Difference
SIMD	Single Instruction, Multiple Data
SLI	Scalable Link Interface