OPTIMISING DATA MOVEMENT RATES FOR PARALLEL PROCESSING APPLICATIONS ON GRAPHICS PROCESSORS

Owen Harrison
Computer Architecture Group
Trinity College Dublin
Dublin 2, Ireland
email: harrisoo@cs.tcd.ie

Dr. John Waldron
Computer Architecture Group
Trinity College Dublin
Dublin 2, Ireland
email: john.waldron@cs.tcd.ie

ABSTRACT
Graphics processing units(GPUs) are starting to play an increasingly important role in non-graphical applications which are highly parallelisable. With the latest graphics cards boasting a theoretical 165GFlops and 54GB/s memory bandwidth spread across 48 ALUs it is easy to see why. The GPU architecture is particularly suited to the parallel stream processing paradigm of low levels of data dependency, high data to instruction ratio and predictable memory access patterns. One largely ignored, yet key, bottleneck for this type of processing on GPUs is both download and readback transfer performance to and from the graphics card. Existing tools provide great developer assistance in many areas of GPU application development, though provide very limited assistance in gaining the best bi-directional data transfer performance. In this paper, we discuss these limitations and present new investigative tools which allow general purpose processing GPU developers to explore the complex array of configuration states which affect both the download and readback performance.

KEY WORDS
Parallel Processing, High Performance Computing, Graphics Processors, Transfer Rates.

1 Introduction

In 2003 Buck[1] and Venkatasubramanian[2] presented some of the first observations that GPUs were becoming faster at processing data parallel tasks than standard CPU architectures. They predicted that this trend would continue and that the performance gap would increase over time in favour of the GPU. These predictions have proven correct as GPUs continue to outperform Moore's law. There is a strong trend to make GPUs more programmable and more feature rich, thus further expanding their application reach. Correspondingly there is an increase in activity in using GPUs for general purpose data parallelisable applications. Currently GPUs are being used for data parallel problems in fields as diverse as databases[3], computer vision[4], audio and signal processing and data mining[5]. One of the most popular projects running on GPUs is the Folding @ Home Stanford project[6]. This project has reported up to a 40 times speed increase over a single core 2.8GHz P4 processor. For more information on general purpose processing using GPUs and a survey of current application areas see Owens et al.[7].

Traditional use of GPUs typically do not put great demands on data transfer rates across the system bus. Also any demands that are put on the system bus are largely in the download(CPU to GPU) direction only. In comparison, general purpose processing applications running on the GPU can require large amounts of data transfer across the system bus in both directions. Achieving optimum bi-directional data transfer performance is a complex task involving a large array of configuration states and transfer methods. Slight mis-configurations can lead to unexpected and difficult to detect significant drops in transfer performance. We present two new tools in this paper focused on download and readback transfer performance respectively. The tools allow the user to create typical application transfer scenarios by providing full control over relevant configuration states such as texture dimensions, system memory alignment, transfer method, external and internal texture formats, data type, texture type, dummy work loads, the use of fragment buffer objects, pixel buffer objects and fragment programs/fixed function. By providing the ability to have full control over all the relevant configuration states, the user can quickly simulate the majority of application transfer scenarios thus allowing the exploration of relevant states and their performance implications.

The tools are written using the OpenGL api, the reason for using this api over DirectX is due to it being platform independent and open, thus being the primary choice for academic research. Section 3 covers optimisation of download data and section 4 covers readback optimisation. Both sections cover the corresponding tool details, typical usage pitfalls and general performance observations. The optimisation observations presented in this paper are based on the following cards: Nvidia GeForce 5200Go(AGP4x), 6600GT(AGP8x) and a 6600Go(PCIe). The paper is not designed to be a complete survey of hardware and its transfer capabilities, but rather an introduction to how these tools can be used to help optimise bus transfer rates for general purpose data parallel applications. For more technical details and access to these tools refer to the project's website[8].

2 Background and Related Work

A GPU's primary source of computational power lies in its fragment processors, each typically consisting of one or more 32bit floating point processing units. Current cards host a significant number of these fragment processors each of which largely operate in isolation. This architectural design descision of isolated processing units leads to a greater allocation of the chip transistor budget to data processing rather than data movement and complex memory hierachies with large caches. This allows GPUs to achieve their high FLOP count. The high performance comes at the price of a resricted programming model with traits such as a high degree of data parallelism, limited program lengths, patterned memory access, limited data gather and no data scatter support.

These restrictions amongst others have lead to a difficult task of extracting the most performance from a GPU. As a result most development assistant tools focus around the shader processors and their optimum ALU and memory usage. This combined with the traditional low bus requirements of applications using GPUs explain why the topic of optimising data transfer to and from the graphics card is largely overlooked. Data transfer between the system and GPU must cross either an AGP or PCIe system bus, both of which provide relatively slow and high latency data movement. Thus this type of data movement can present the first bottlenecks to classes of applications which require high levels of CPU-GPU communication.

Currently there exists only two reputable tools which cover the area data transfer rates. These tools are designed to be used solely as benchmarking tools for a number of predefined configuration states. They are not designed to give the user the flexibility required to simulate the virtually endless application data transfer scenarios. Stanford's GpuBench[9] suffers from aging and does not include critical recent GPU transfer features such as pixel buffer objects and asynchronous readback support. Nvidia's Performance Tool[10] does provide access to these features, however they are exposed in a limited fashion, for example the user cannot vary the amount of dummy CPU work executed to measure the amount of performance gain from asynchronous readback support. These tools are not designed to help the developer achieve optimal data transfer rates. The authors attempt to resolve this by the use of the presented tools in this paper.

3 Download Optimisation

3.1 Tool Introduction

Data can be transferred from the system to the GPU(downloaded) by means of texture transfer or direct rendering of pixels to the active framebuffer. Highly parallelisable applications which are data heavy and have a low processing complexity per data unit are the most likely to bottleneck at the data download stage. This is due to the relatively slow CPU to GPU bus speeds when compared to on-card data movement rates. With these applications it is crucial to fully explore all download related options. The use of this tool can show that slight variations in configuration settings can result in large differences in transfer rates. Each execution of the tool transfers 6.25GBytes of data frame by frame to the active framebuffer, thus averaging out any individual frame's impact on the final result. Transferring the same amount of data for each execution allows for easy comparison of various scenarios. The tool will not transfer the full data allocation to the card if a scenario is estimated to take longer than 3 minutes to complete, in such a case an estimated rate for the full data transfer is output. The tool generates these estimates by first running the frame transfer loop an initial 10 times before running the full scenario. These estimates are generally pessimistic as the first frame transfer time occupies a larger percentage of the run time.

At the start of each scenario execution, an initial test frame is downloaded and subsequently read back from the GPU using the user requested state. The readback data is then compared against the downloaded data, taking into account any padding bytes, see packAlignment below. If there is any mismatch between the readback data and the transmitted data a warning message is generated indicating that the transfer rates are unreliable. This is especially pertinent regarding readback optimisation, see Usage Pitfalls section 4.4.

3.2 Tool Details

The full list and description of configuration states which the tool exposes is presented below. These provide sufficient control to simulate the majority of data download scenarios. The controllable states are presented as the following input parameters to the download tool.

pixelDimension Un/packAlignment transMethod glExtFormat glExtType glIntFormat glTexTarget FBOState PBOState FPState

The parameter *pixelDimension* allows the user to specify the dimension of the buffer sizes and texture sizes involved in the transfer. Currently to simplify the number of parameters this is restricted to square 2D textures, though this could easily be extended. *Un/packAlignment* allows the user to specify the alignment in system memory that the data will be unpacked from, this is described in full in The OpenGL Programming Guide[11], see glPixelStore(). In general, architectures experience greater system memory read performance when the reads are aligned to some byte boundary, this setting should correspond to the user's system's recommendations. *transMethod* indicates the choice of method for downloading data to the graphics card, the tool supports DrawPixels and TexSubImage(recommended over TexImage for repeated transfers). The *glExtFormat* and *glExtType* specify both the format and type of data that is held in system memory. *glIntFormat* specifies the requested internal storage format and resolution on the graphics card. The *glTexTarget* parameter specifies the type of texture target to use to bind the texture to, the supported and tested targets are GL_TEXTURE_RECTANGLE_ARB and GL_TEXTURE_2D. *FBOState* can be set to FBO_ON or FBO_OFF, indicating whether or not the data is to be transferred into a texture attached framebuffer object(FBO). The internal format and texture target can be set to NONE if the transfer method is set to DrawPixels, unless a framebuffer object is being used. *FPState* specifies whether a fragment program is bound and used or whether the fixed function pipeline is used. Both FBOState and FPState make most sense in the context of DrawPixels, during TexSubImage mode only a point is used as the rendering operation to flush the texture transfer. The *PBOState* parameter allows the user to select whether a pixel buffer object is used to unpack data from during the transfers.

3.3 Usage Pitfalls

False increases in reported transfer rates for simulated application scenarios are the most common problem when using the download and readback tools. These can occur due to transparent driver optimisations and restrictions, which are difficult to detect, and must be taken into account.

When rendering to the default visible framebuffer there are many opportunities for false rates to be reported. If the screen display size does not encompass the entire visible rendering window then the data which corresponds to the cropped region of the window do not need to be transferred. If drawing to a position outside of the area specified when creating the render window, the data transfer does not need to take place. Also when another window obscures the visible window, the obscured region's corresponding data do not need transferring. These three issues affect the DrawPixels mode by artificially increasing the download rates, TexSubImage does not result in a speed up as only a point is drawn to screen in this mode, the texture is still transferred but the omission of a single fragment rendering makes little performance difference. The verification stage of both the DrawPixels and TexSubImage modes fails due to the data not being transferred and consequently not being correctly read back. The tool user must take care not to obscure the visible window with any gui objects for any portion of time and to ensure the visible area of the screen can encompass the render window. The use of a framebuffer object removes these obscuring issues as rendering doesn't target a visible window.

3.4 Download Observations

An array of scenarios were run using the author's hardware resulting in the following observations across significant groupings of scenarios. These are designed to act as a guide to the type of insight that can be gained from using the download tool. Note that no existing tool allows the user to gather such insights into the transfer behaviour of a system.

Pixel Buffer Object: In general using pixel buffer objects while in TexSubImage mode and transferring floats with an internal format from the OpenGL float buffer extensions(ARB or Nvidia) or unsigned bytes results in the best performance. When using pixel buffer objects, TexSubImage mode for data transfer is vastly superior, 3 times faster, compared with DrawPixels mode. The use of integer or short data type shows poor performance and is not improved with the use of PBOs. Under no scenario did DrawPixels experience a speed up when switching between PBO_OFF and PBO_ON. Part of the difference in speed between these modes can be explained by DrawPixels resulting in fragment generation and subsequent processing whereas TexSubImage does not produce fragments but transfers directly to texture memory. However when testing TexSubImage using texture mapped quad rendering, there still existed a 2 times speed up: thus the main difference in speed must be explained by the lack of affect PBOs have with DrawPixels.

Texture Format and Type: There are hundreds of combinations of internal and external texture formats and types all of which have performance implications. The following covers some points of note regarding the mix of formats and types. With regard to floats usage with PBOs, swizzled external formats leads to no performance increase. Also, the use of RGBA or RED delivers improvements though the external formats must be accompanied by a resolution specific internal format from the OpenGL float buffer extensions. Use of non specific resolutions cause down conversion, loss of precision, and a failure to verify, see section 4.4. It is worth noting that there is a consistent 6-7% performance advantage using internal formats from NV_float_buffer extension over ARB_texture_buffer. Unsigned byte transfers can be accelerated using PBOs, however concerning 4 component transfers BGRA external format must be used. If padding is avoided for a single byte component by using LUMINANCE8 as the internal format, speed up can also occur.

Fragment Program: The use of a fragment program(FP_ON) has no significant impact on TexSubImage mode as expected, however when using DrawPixels mode there is a dramatic slow down compared to the fixed function pipeline. The fragment program used is a simple colour in colour out pass through program. No explanation could be found for this behaviour. It is assumed some configuration error is causing a part of the rendering process to resort to software mode.

Data Padding: Scenarios which result in data padding, i.e. increased number of components in the internal format compared to the external format, lead to no speed up when using PBOs. To compound this, component padding also causes a slow down compared to scenarios that don't pad when PBOs are turned off.

Texture Size: Varying texture sizes that are transferred can have a significant effect on overall trans-

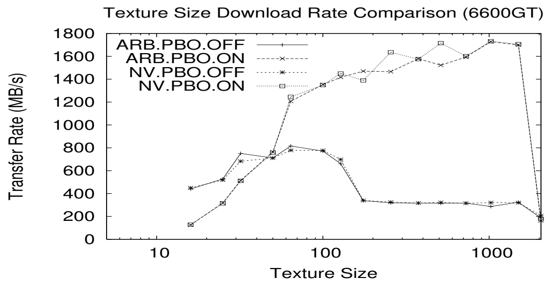

Figure 1. Download Rates For Varying Texture Sizes

fer rates. To demonstrate this, Figure 1 shows the results of running four scenarios, with and without the use of PBOs and the use of two different internal formats, FLOAT_RGBA32_NV and RGBA32F_ARB on the 6600GT AGP card. It was noted that for small textures non PBO use results in higher rates of transfer. Also when using FLOAT_RGBA32_NV there is a significant increase in performance at powers of two texture sizes, where as RGBA32F_ARB shows no correlation. In general the larger the buffer size the faster the download rate, though over a certain size, depending on the memory availability on the card, the performance drops severely.

4 Readback Optimisation

4.1 Tool Introduction

Traditional graphical use of GPUs have little demand for high transfer rates and what demand there is exists largely for download data transfer. Thus readback rates are considerably lower due to hardware optimisations. This can present a challenge to general purpose applications which require their output to be consumed in a non visual way, ie. the results must be readback to the system. New advancements such as the introduction of PCIe and the addition of asynchronous readback transfer has somewhat helped this situation. Asynchronous readback allows CPU and GPU pipelining of data processing spread across both processors. The previously mentioned tools are either missing asynchronous support or provide it in a switch on/off manner. The presented readback tool provides two modes of operation, one for readback transfer rates and the second for analysing the asynchronous readback behaviour of the current configuration. Both modes provide a similar full control mechanism to the download tool for simulating various transfer scenarios. In both modes the same amount of data, as in the download tool, is readback from the GPU frame by frame. Also, the same verification stage and initial time limiter trial phase is executed. The specific modes of operation, transfer rate and asynchronous behaviour are discussed in the following sections.

4.2 Transfer Rate Mode

The transfer rates mode only outputs the rate of data readback from the GPU. It attempts to minimize the amount of data downloaded to the GPU, using a non texture mapped quad draw operation to ensure no driver optimisations are occurring to eliminate the repeated readback of the target buffer. The tool uses input parameters to calculate the number of readback iterations required to transfer a fixed amount of data as in the download tool. The input parameters are as follows:

pixelDimension un/packAlignment glExtFormat glExtType glIntFormat glTexTarget FBOState numberPBOs PBOUsagePattern [workLoad] [drawMode] [blockingDetail]

The last three parameters are optional and are covered in the asynchronous behaviour section below. When *FBOState* is set to FBO_ON or FBO_OFF the glReadPixels command is used to retrieve data from a texture attached FBO or the default framebuffer respectively. The FBOState can also be set to FBO_ON_GTI which allows the user to specify that the glGetTexImage command should be used to read from the texture attached FBO. The first five parameters are used similarly to the download tool and configure the draw, read and store OpenGL commands. *numberPBOs* allows the creation of multiple PBOs which are used as readback packing buffers used on a rotational basis. The sequence of actions within the transfer loop when PBOs are being used is: bind PBO[n], draw frame, read frame, bind and map PBO[n - x], unmap PBO[n - x], where x is the number of PBOs requested. This pattern of use is recommended in Nvidia's Fast Texture Transfers article[12]. The *PBOUsagePattern* parameter is used to specify the expected usage pattern of the pixel buffer object, see the BufferData command in the OpenGL Programming Guide.

4.3 Asynchronous Behaviour Mode

To expose a scenario's pipeline potential the tool can insert variable amounts of dummy work to simulate CPU side processing during data readback. The user can vary the amount of work load processing by entering an integer as the *workLoad* parameter. Table 1 shows an example output of the tool when running in asynchronous mode. It includes the total amount of map and read command blocking times, and also the time spent in the dummy work loop, all times are reported in micro seconds. To maximise the pipeline potential, a scenario ideally would reduce the total amount of blocking time by the same amount of dummy work time increase, thus maintaining the same transfer rate while gaining free CPU cycles. This can only happen until the amount of dummy work time is greater than the total transfer time.

The default draw mode used during the execution of the readback tool is a non texture mapped quad render operation as mentioned previously. This is suitable for measuring pure readback rates, however it is not always suit-

Table 1. Example output of readback tool in asynchronous mode

```
Map Blocking Time = 164279
Read Blocking Time = 52159259
Map+Read Blocking Time = 52323538
Dummy Work Time = 44455106
Bytes Transferred = 6710886400
Transfer Time = 97553896
Transfer Rate = 65.60MB/s
```

able for measuring the pipeline potential of a scenario. The asynchronous efficiency depends largely on how the particular hardware system handles DMA memory transfers during bus contention and how the graphics driver behaves. A drawing mode called PIXEL_DRAW can be used via the *drawMode* parameter, which instructs the tool to use DrawPixels to fill the entire window each frame. This generates download traffic to contend with DMA readbacks thus simulating an application which requires bidirectional data transfer. The default QUAD_DRAW mode can be used to more closely simulate the bus traffic during a render to texture ping-pong[13] application which largely uses the results generated on the GPU for the next frame's input. The *blockingDetail* parameter when used generates logging messages for each frame displaying the blocking times in detail.

4.4 Usage Pitfalls

The following pitfall is applicable to both download and readback transfers, though is discussed here as it is more likely to cause false speed ups for readback. The internal format used to store data within the graphics card is implementation dependent. According to the OpenGL specification an implementation can use the input parameters such as external format, external type and internal format merely as a guide. The graphics driver makes this process transparent and thus when specifying these parameters the user cannot be sure exactly what form the data will take when transferred from and to the card. The result being that data can be down converted into a lower precision representation before being transferred over the bus, leading to false speed ups.

An example of this pitfall on is the mapping of the internal format GL_RGBA to GL_RGBA8, meaning that floats, ints, shorts all get down converted to an 8 bit representation when this internal format is used. There is no way of programmatically knowing a smaller resolution is being used apart from using a strict verification stage which requires an exact match between data downloaded and readback from the GPU. When a verification failure is detected the tools continue, however a warning message is outputted to instruct the user of this danger. The reason for not aborting the run on verification failure is that 32 bit integers, even when using a 32 bit internal representation on currently available GPUs, will still loose precision on the round trip. To aid this judgement the internal formats natively supported on the GPU should be known, for Nvidia this information can be found in the Nvidia Programmers Guide[14].

Window obscuring affects readback in the same manner as it affects download transfers, care must be taken to ensure when using the default framebuffer that the entire window is visible, otherwise the verification stage will fail and artificial increases in transfer rates can occur. Also, it was noted that for asynchronous behaviour to function the most recent graphics drivers must be used. Namely, with regard to Nvidia running on Linux, driver versions ¿= 1.0-8762.

4.5 Readback Observations

As the array of runnable scenarios and hence their results are virtually limitless, we present the notable observations across groups of scenarios rather than individual results.

4.5.1 Transfer Rate Observations

Short and integer data type scenario transfers all under perform by a minimum of 50% compared to byte and float scenarios. Single external components suffer a four times reduction if the internal format is not explicitly requested as a single component due to all four components being transferred across the bus and filtering occurring within the driver. When using GetTexImage to readback there is a large drop in performance compared to using ReadPixels. When transferring bytes, the use of TEXTURE_RECTANGLE_ARB consistently out performs TEXTURE_2D by 6-7%. Note that this performance difference does not exist when using the extended internal formats for float transfer such as FLOAT_RGBA32_NV or RGBA32F_ARB. When varying the size of buffer dimensions, there is a notable performance increase using higher numbers of PBOs for small buffer sizes. Also there are large rates increases when using sizes which are powers of two, though in general the larger the buffer size the faster the readback rate.

4.5.2 Asynchronous Behaviour Observations

Asynchronous Support: On the tested hardware the vast majority of scenarios do not support asynchronous readback. The notable groups which don't are: all scenarios with an external type of unsigned byte that don't use BGRA external format; all scenarios with an external type of signed byte; all scenarios with external types of signed and unsigned shorts and integers; all scenarios with an external type of Luminance; all floats without a specific internal resolution formats, e.g. those from the float_buffer or arb_texture_float extensions; all scenarios which use GetTexImage to readback data.

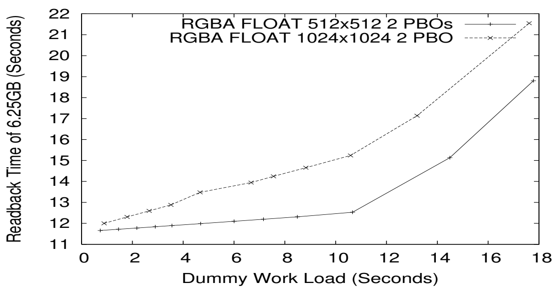

Figure 2. PCIe Async Comparison of 512x512 versus 1024x1024 Buffers

Pipeline Efficiency: The ideal behaviour of scenarios which support asynchronous readbacks would be that for every second added to the dummy work time, blocking time is reduced by a second. However in practice this is not true and varying settings such as number of PBOs and buffer sizes used affects how close a scenario comes to this ideal. On the AGP cards the number of PBOs used, buffer sizes and drawing mode have a significant effect on the pipelining efficiency. For example, using a 512x512 buffer size in quad draw mode, as the number of PBOs used increases the more efficient at hiding the extra work time the scenario becomes. However when the same scenarios use pixel draw mode, there is no difference between the pipelining efficiency as the number of PBOs used increases. Using the PCIe card eliminates the vast majority of the behavioural differences regarding the number of PBOs and draw modes used. Though in Figure 2, one can see that there is a significant difference between the asynchronous behaviours when buffer dimensions are varied with 512x512 outperforming 1024x1024 in terms of pipeline potential.

5 Conclusions

The use of graphics processors for data parallel processing applications can result in major increases in performance over conventional architectures. However, there is a high risk of failure to realise this potential gain due to the complex programming environment of GPUs. One of the primary bottlenecks to overcome is efficient movement of data onto and off the graphics card. The use of the tools presented in this paper allows the user to uncover a great deal of insight into data transfer related idiosyncrasies. This insight can be used to identify optimal state configurations for data movement and thus help avoid such bottlenecks.

Future work includes the use of these tools to present a comprehensive survey of existing hardware and their data transfer capabilities. Required for the survey is testing on ATI based hardware. The tools can also be extended to support colour index mode and to include more internal formats, external formats and external types. Also the ability to define non square textures and addition of more texture targets to encompass more than just two dimensional ones would be useful.

References

[1] I. Buck, Data parallel computing on graphics hardware, *Siggraph: Graphics Hardware Panel*, San Diego, USA, 2003.

[2] S. Venkatasubramanian, The graphics card as a stream computer, *DIMACS Workshop on Management and Processing of Data Streams*, San Diego, USA, 2003.

[3] N. Govindaraju, J. Gray, R. Kumar, and D. Manocha, Gputerasort: High performance graphics coprocessor sorting for large database management. *ACM SIGMOD/PODS*, Chicago, USA, 2006.

[4] J. Fung, S. Mann, and C. Aimone, Openvidia: Parallel gpu computer vision, *ACM Multimedia*, Singapore, 2005.

[5] N. K. Govindaraju, N. Raghuvanshi, and D. Manocha, Fast and approximate stream mining of quantiles and frequencies using graphics processors. *ACM SIGMOD/PODS* Baltimore, Maryland, USA, 2005.

[6] Stanford University, Folding At Home on ATI GPUs, http://folding.stanford.edu/faq-ati.html.

[7] J. D. Owens, A survey of general-purpose computation on graphics hardware. *Eurographics*, Dublin, Ireland, 2005.

[8] O. Harrison, J. Waldron. TransferBench Tool - available online at https://www.cs.tcd.ie/~harrisoo/research.html .

[9] I. Buck, K. Fatahalian, and P. Hanrahan, Gpubench: Evaluating gpu performance for numerical and scientifc applications. *ACM Workshop on General Purpose Computing on Graphics Processors*, LA, USA, 2004.

[10] Nvidia. Nvidia's PBO Texture Performance Tool, http://download.developer.nvidia.com/developer/sdk/individual_samples/featured_samples.html #textureperformancepbo.

[11] OpenGL ARB, D. Shreiner, M. Woo, J. Neider, and T. Davis,*OpenGL Programming Guide: The Official Guide to Learning OpenGL, Version 2*, Boston: Addison-Wesley Professional, 2005.

[12] Nvidia. Fast texture downloads and readbacks using pixel buffer objects in opengl, http://developer.nvidia.com/object/fast_texture_transfers.html, 2005.

[13] D. Goddeke, Render To Texture(inc. ping-pong) Tutorial, http://www.mathematik.uni-dortmund.de/goeddeke/gpgpu/tutorial.html.

[14] Nvidia. GPU programming guide, http://developer.nvidia.com/object/gpu_programming_guide.html, 2005.

Conference Proceedings:
This work was subsequently published as Optimising Data Movement Rates for Parallel Processing Applications on Graphics Processors, Harrison O and Waldron J.
Proceedings of the 25th International Conference on Parallel and Distributed Computing and Networks, PDCN 2007, February 13-15 2007, Innsbruck, Austria. pages 251-256. ISBN: 978-0-88986-637-9