Characterizing Scientific Application Performance on GPUs

Base Paul
(base@cs.wisc.edu)

Abstract

The new-generation GPUs (Graphics Processing Units) have enormous computational power these days that it beats the common dual core CPUs by a factor of 10x when it comes to raw computational power (measured in FLOPS). This power, combined with advancements in I/O and large amount of on-board memory on GPUs has made them an attractive option for doing scientific computations that are, in general, highly data parallel.

Controlling this power, though, have not been easy task. The options for programming GPU have not been very simple, and the GPU design provides a non-graphics programmer with many syntactic, semantic and implementation challenges that are hard to deal with. This paper quickly surveys some of the options available to the programmers today to implement Scientific Applications on GPU, and then proceeds to characterizing the application performance on GPUs. The paper describes experiments done to analyze the performance of scientific applications on GPU. The parameters measured include the *raw computational power*, *end-to-end (wall clock) performance, ability of the GPU to off-load work from CPU* and *accuracy of computation on GPUs*.

The experiments included known implementing data-parallel, scientific application workloads such as Ocean and Barnes (from Splash-2 benchmarks). The measurements were done with an nVidia 8800 GTX on a Dual Xeon workstation.

1. Introduction

The GPUs have come a long way since early 2000, when they were used mostly to achieve higher resolution and some video acceleration. The recent GPUs, however, provide huge memory bandwidth and computational horsepower, with fully programmable vertex and pixel processing units that support vector operations with IEEE floating point precision. Since GPUs can be (and are) highly optimized for arithmetic computations, their raw performance is often an order of magnitude better than CPUs. This computational power can be harnessed to do other computationally intensive, "general purpose" work, with the help of special programming languages such as GLSL, Cg and Cuda.

If GPUs offer so much power, they offer a lot more challenges too. Programming GPUs to do general-purpose computation is a lot harder than any other parallel programming model. They are designed for the video game marker and the programming model is quite unusual. Regardless of the application that needs to be programmed on the GPU, the developer must be an expert in computer graphics and its computational style to make effective use of the hardware. Mapping the problem from the more familiar CPU programming model to the GPU programming model is half the work in implementing the solution on GPU.

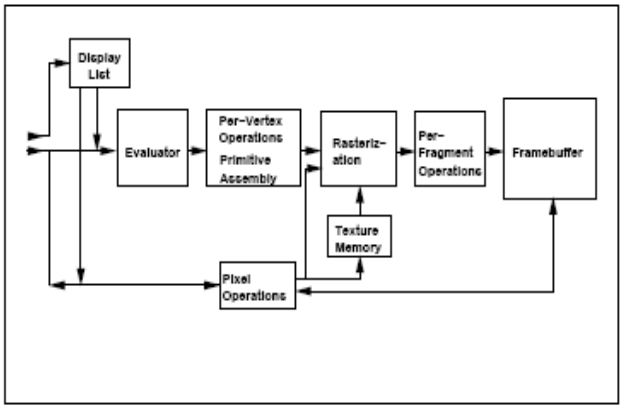

Figure 1: Typical graphics pipeline on a graphics processor (taken from OpenGL Specification)

In addition to the programming model, there are other considerations too while programming the GPU. The GPU can operate only on its own high-speed memory. Or in other words, the data has to be transferred to the GPU memory from CPU memory. This transfer often has an impact on overall

computation time. Complicating the scientific computation is the fact that the GPU designs are more optimized for write access (i.e. to write to the display), and reading processed data back to CPU memory is often costly. So minimizing the data transfer between CPU and GPU is an important part of the overall implementation.

Remainder of this paper is organized as follows. Section 2 surveys the programming methodologies available for GPUs. Section 3 describes the design of experiments to analyze the application performance. Section 4 describes the results from the experiments. Section 5 contains key take-aways for anyone planning to use GPU for parallel computation.

2. GPU Programming

The main programmability of GPUs are on the "Per-Vertex Operations" and "Per-fragment Operations" blocks in Figure 1. These blocks are commonly referred to as "vertex shader" and "fragment shader" (or sometimes Pixel Shader" respectively. Programming GPUs typically means programming these two blocks using a type of languages called "shading language". Following sections describe some of the options available to the GPU programmers.

OpenGL

OpenGL (Open Graphics Library) has been one of the most popular way of programming graphics hardware for a long time – even before GPUs became programmable. The OpenGL defines an interface that consists of a set of several hundred procedures and functions that allow a programmer to specify the objects and operations involved in producing high-quality graphical images, specifically color images of three-dimensional objects. These interfaces could be implemented in hardware or software. The programmer does not have to know how or where a certain function is implemented.

When the GPUs became programmable, OpenGL was extended with a shading language. This extension is known as GLSL (GL Shading Language). Like, OpenGL, this extension is also supported by most, if not all, GPUs in market today.

OpenGL programming model is based on the assumption that graphics hardware is meant to do graphics operations. Many OpenGL calls deal with drawing objects such as points, lines, polygons, and bitmaps. Solving a computational problem in GPU using OpenGL involves mapping the problem into graphics domain – for e.g. An array is stored as a texture, and running a shader program on some data (stored as textures) involve "rendering" to the screen (or an off-screen texture).

The steps involved in programming a generic computational problem on GPU using OpenGL are the following:

1. Map the problem into graphics domain. This involves defining textures, deciding on which shader to use (vertex Vs fragment Vs a combination), figuring out data representation on GPU, etc
2. Writing shader programs that perform the computation on GPU using GLSL
3. Creating a graphics context (and windows – mostly invisible) invoke rendering operations on.
4. Transferring the data to one or more textures on GPU, performing any translation required for differences in data representations between CPU and GPU.
5. Doing a rendering – in which vertex/fragment shader programs run through the input data and write to an output buffer (texture)
6. Transferring the data back to the CPU, similar to Step 3, performing an conversions required.

As can be seen, most of the steps described above have no connection with the problem being solved. They are completely independent of the problem, yet take up majority of the effort in implementing a solution!!

Another problem with GLSL (and other shading languages) is the way in which a program accesses GPU memory. They all allow programs to read from anywhere on the device memory, but they can, at a time, write to only one target area (typically, the frame buffer). This hampers general purpose programming severely.

Cg

Cg stands of 'C for Graphics'. It was developed by nVidia as an optimized language for their own GPUs. Unlike OpenGL, Cg deals with only the shaders.

Cg is not a complete framework to do general purpose computation on GPU. Referring to the steps for OpenGL programming in earlier section, Cg comes in Step 2. All the other steps there are still required. The programmer must use other toolkits such as xlib to create and manage graphics contexts.

Though Cg is optimized for nVidia's GPUs, it has all the drawbacks of GLSL, when it comes to general purpose computation. The added draw back is the incompatibility with non-nVidia graphics cards.

DirectX

DirectX is suite of application programming interfaces from Microsoft, and it is answer to multimedia programming in general. Though the graphics variants of DirectX (namely, DirectDraw and Direct3D) allows programming shaders through a shading language called HLSL (High Level Shading Language), it has not been used much for general purpose computing on GPU.

The steps involved in writing general purpose programs using HLSL is similar to that of OpenGL, the only changes being APIs.

BrookGPU

BrookGPU is one of the simpler and more portable options for general purpose computation on GPU. It is the GPU version of the more popular Brook stream programming language from Stanford University. It basically provides an abstraction over other programming languages and runtime systems such as OpenGL.

BrookGPU has an ANSI C-like syntax, includes a compiler and a runtime system. The program is written in “Brook” language, and is then compiled to C++ using the Brook compiler. The generated code refers to APIs supported by the Brook runtime. The runtime abstracts the underlying hardware or primitive windowing system (such as OpenGL). The most recent version of BrookGPU can run over OpenGL, DirectX and AMD's “Close To Metal”.

While Brook is easier to program than any shading language, and provides excellent portability, it still is fairly complicated and unfamiliar to traditional C programmers. In order to support portability across all graphics cards (or otherwise), it does not have interfaces to do random access on GPU memory (provides just gather – like OpenGL and others).

Cuda

Cuda is another programming model proposed by nVidia. It is completely different from all other models we discussed so far in the following ways:

- It completely eliminates the 'graphics' concepts from computation on GPU.
- It is very close to C, in syntax and most semantics.
- It allows truly random read/write access on device memory from functions running on device.
- Allows good control over data layout and thread management

Cuda differentiates functions running on the GPU and those running on CPU. It adds some keywords to that of C to qualify functions and variables as running/accessible from CPU, GPU or both.

The steps involved in programming with Cuda are the following:

1. Identify how the data is going to be split to parallel tasks (arranged as a 3-D matrix of threads, which are then arranged as 3-D matrix of blocks – they can be treated as 1-D as well, depending on Application)
2. Write a device program (*kernel*) that operates on one chunk of data.
3. Transfer the data into device memory using Cuda APIs
4. Run the kernel with the matrices mentioned in Step 1 as arguments. GPU creates and manages the threads and thread-blocks.
5. Transfer the processed data back to CPU Memory.

As can be seen, most of the steps involved are about parallelizing the application, instead of creating and managing graphic contexts and mapping the problem into an unrelated domain.

All the programs in this project were implemented using Cuda.

3. Design Of Experiments

As mentioned earlier in this paper, programmability is just one of the quirks of GPU programming. There are other parameters too, those decide a GPUs suitability for a (type of) application(s). They are r*aw computational power*, *end-to-end (wall clock) performance, ability of the GPU to off-load work from CPU* and *accuracy of computation on GPUs*.

There are various ways of characterizing these parameters. This project attempts to do that by using two popular workloads from the Splash-2 benchmarks – Ocean and Barnes.

The Ocean application simulates ocean movements based on eddy and boundary currents. The simplified version implemented for this project simulates temperature distribution in an ocean over a period of time. The ocean is considered as a grid. At each instant of time, the temperature at a given grid location is a function of the temperature of its neighboring grid locations at the previous instant of time. Our implementation averages the temperature of the north, east, south and west locations of a given grid location to find its new temperature value. In order to avoid skews because of using a just computed value for the same instant in a later grid location, the new values are written to an alternate buffer. The main loop alternates between these buffers every iteration.

The Barnes application simulates the interaction of a system of bodies in three dimensions over a number of time-steps, using the Barnes-Hut hierarchical N-body method. An N-body simulation calculates the gravitational effects of the masses of N bodies. The calculation is done by finding pair-wise force exerted by bodies on each other, and accumulating such forces on a given body by all the other bodies in the 'universe'.Since calculating force on a given object requires accessing all other bodies once, effect of caching is minimal in this case. Also, the number of computations are O(N^2), but force on a given object can be computed independent of such computations for other bodies. Like Ocean, the version implemented in this project is a scaled-down version, and works on a 2-D space, instead of 3-D.

This project performs measurements on 3 configurations.

1. Chianti is a Sun Ultra-SPARC T1 (Niagara) machine with 8 cores capable of running 32 threads at a time. This configuration is programmed using pthreads or OpenMP
2. Clover is an Intel Clovertown based system that has 8 cores, capable of running 3 threads at a time. This configuration is programmed using pthreads or Intel's TBB.
3. 'GPU' – is a Dual Intel Xeon with an nVidia 8800 GTX graphics card. This configuration is programmed using Cuda.

Raw Computational Power

Raw Computational Power, in this project, refers to a processor's capability to process data when it is accessible at the fastest possible location (in this project, L2 cache or the GPU's shared memory). The experiment to measure raw computational power uses a small data set that can be fit into the processor's cache and processes that data for a certain number of iterations.

The cache sizes are different between Chianti, Clover and GPU. For the sake of comparison, the smallest of these was taken as the data size (16KB on GPU). This allows a 32x32 sized ocean (and its copy) to fit in.

End-to-End (Wall Clock) Performance

End-to-End performance, in this project, refers to the time taken to finish a computation, regardless of its size of location. This includes time to allocate temporary buffers, transfer data to a faster cache from slower memory, etc. In this project, the input data set is always expected in main memory in the beginning.

Characterizing the end-to-end performance has two parts to it.

1. For a given data set size, find how different configurations perform.

2. For a given configuration, find how the end-to-end performance scale as the data set size varies. This is especially important for GPU configuration, where the main memory is too far away (i.e. too slow, compared to devices own memory)

For the 1st part, the experiment involves using a data set that does not fit in the cache. The computation is then performed a certain number of times.

For the 2nd part, the experiment involves varying data set size from something that fits in memory to something that doesn't. When data doesn't fit in device memory, it has to be split into slices that can be individually brought into device memory to process. This may involve additional data transfers and hence impact performance. This experiment is performed only for GPU, since that is the main focus of this project.

Ability of the GPU as a co-processor

For GPU to be considered a “co-processor”, it should be able to off-load work that will otherwise be done on CPU. Or in other words, when a computation is performed on GPU, it should use as little CPU resources as possible.

The way to measure this characteristic is to schedule a long running computation on the GPU, and then to monitor the CPU usage and memory usage. Tools like top or kpm can be used to monitor CPU and memory usage.

Accuracy of computation

When it comes to accuracy, each application has its own requirements. Some applications (like weather prediction) may require more accuracy than the others (like a 3D image processing for display).

This experiment is aimed at estimating the accuracy of different kinds of operations (such as arithmetic, trigonometric, logarithmic, etc) done on GPU. The experiment performs the said operations on thousands of values, and compare the results with the results from doing the same set of operations on same set of values on CPU (or in other words, results from CPU is considered “gold standard”). In addition, the experiment compares the accuracy of computation of Ocean and Barnes.

4.Results

The project implemented two scientific application workloads – ocean and Barnes – using GPU. The configurations used to conduct the experiments were the following.

Hardware Environments

Name	Configuration
Chianti	SUNW,UltraSPARC-T1, 32 Cores @ 1Ghz, 3M L2 cache, 16G RAM
Clover	Intel Xeon CPU E5345 @ 2.33GHz x 2 (i.e. 8 cores), 4M L2 cache, 32GB RAM
'GPU'	Intel Xeon @ 2.8G x 2, 4GB RAM, nVidia 8800 GTX GPU with 768MB on-board memory

Software Implementation

Both Ocean and Barnes have been implemented on all three configurations.

Ocean was implemented on Chianti using OpenMP, and was made to run on with 32 threads. Due to Chianti's weakness with floating point operations, the implementation used integers. On Clover, ocean was implemented using Intel TBB and was run on 8 threads. GPU implementation of ocean used Cuda. Both Clover and GPU implementation was based on floating point arithmetic.

Barnes implementation on Chianti used pthreads and it made use of all 32 available cores. It distributed the threads equally among all cores, and, like Ocean, used integer arithmetic instead of floating point. On Clover, the implementation used pthreads, but used floating point arithmetic. GPU version was, again, implemented using Cuda.

The accuracy tests were done only on the GPU, and was done based on Cuda. It was done only for single precision floating point operations.

Raw Performance

In this experiment, ocean implementation was run with a small data set (32x32) on Chianti (pthread implementation), Clover (pthread implementation) and GPU (Cuda implementation). Their performances were then compared against each other.

It was found that GPU implementation is about 2x faster than that of Clover, and about 5x faster than that of Chianti). This is on the expected lines, but slightly off. With its 16 'processors' each capable of running 8 threads, GPU was expected to be much better than the 8 core Clover. The less-than-ideal performance is attributed to the fact that Clover's clock speed is more twice as that of GPU, and also to the fact that the data set was so small that it fit in Clover's L1 cache (instead of L2), and hence was faster to access than GPU's shared memory.

End-to-End Performance

There were two experiments here, one that compares the performances of 3 hardware configurations mentioned above, and the other that characterizes the performance of GPU for different data set sizes.

For the first experiment, the following graphs show the results.

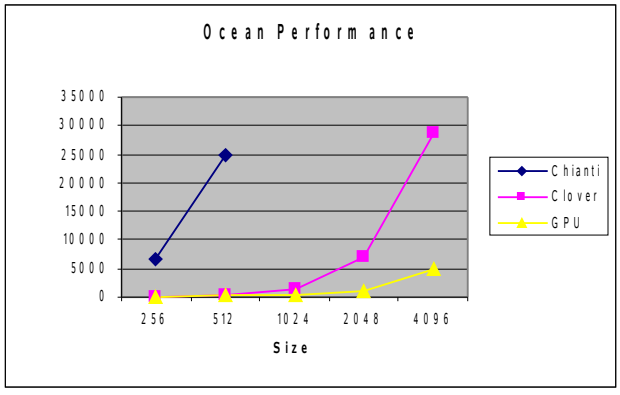

Figure 2: Ocean's performance on 3 configurations (X-axis: ocean size, Y-axis: time (msec) for computation)

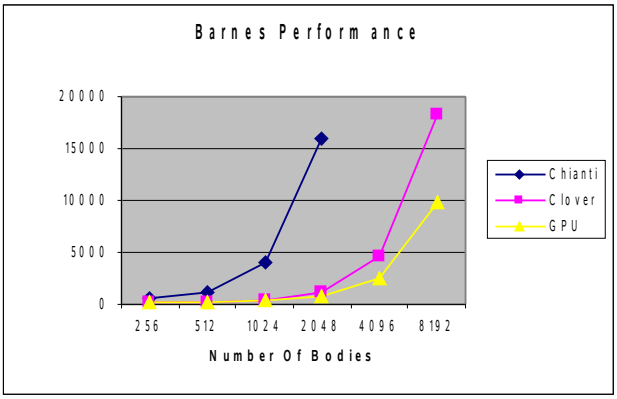

Figure 3: Barne's performance on 3 configurations (X-axis: Universe size, Y-axis: time (msec) for computation)

Figure 2 shows the Ocean's performance on all 3 configurations. Data from Chianti is shown only up to a data set size of 512x512, because bigger data set was expanding the graph's range, making it impossible to read. The performance of Clover and GPU are very close to each other up to an ocean size 1024x1024. Until then the data fits in Clover's L2 cache (which has access times comparable to GPU's device memory). Beyond that, Clover performance drops faster, such that for a 4096x4096 ocean, the performance on GPUs is 7x that of Clover.

Figure 3 shows the Barnes' performance on all 3 configurations. The trends are similar to that of Ocean, but the gain is not as much as you see on Ocean. For e.g, for a data size 4096, the performance gap in Barnes is just 2x, instead of ~7x in Ocean. This attributed to the change in application's memory access pattern. Ocean has a more localized memory access pattern (to process a grid location, it just accesses 4 other locations, which are spread across 3 rows – previous, current and next). Barnes, on the other hand has a more distributed access pattern (to process one body, it has to access all other bodies in the universe). This difference allows the Ocean implementation to take advantage of the 'shared memory' which are completely user controlled (unlike CPU's cache) and much faster than device memory. The GPU implementation of Ocean pins 3 rows (previous, current and next) in to the shared memory before processing a row.

Another interesting thing to note is the fact that CPU implementation starts following the order of the algorithm much earlier than GPU implementation. For e.g. Ocean algorithm is O(N), and the CPU implementation starts showing this for an ocean size of 1024 (till then the effect of caching keeps the execution time in check). The GPU implementation on the other hand starts this trend only at around 4096. This is attributed to the fact that access time for CPU to main memory is slower than that of GPU to device memory.

When it comes to Barnes, the order is O(N^2). The CPU starts showing it at around 1024 (~4x increase from 1024 to 2048), while GPU shows it from 2048 (3.8x from 2048 to 4096).

Experiment 2 (data sets of varying sizes on GPU) created Figure 4 below.

The X-axis shows the number of bodies for Barnes and size of ocean in one of the dimensions in Ocean (that is, for x=N, Ocean size is NxN). Y-axis is the time to perform the simulation for 100 iterations.

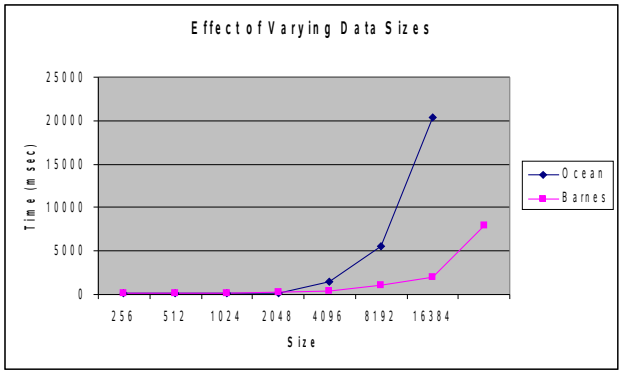

Figure 4: Varying data sizes on GPU ((X-axis: ocean/universe size, Y-axis: time (msec) for computation)

As can be seen, the performance degrades quite significantly as the size increases beyond a threshold. This threshold corresponds the amount of device memory available. Ocean requires memory O(N^2), when Barnes requires O(N). Once the application crosses the threshold the performance degrades rapidly.

An interesting observation here, is the slope of the curve. Though the Ocean's curve looks steeper, it only appears so. A 2x increase in X-axis for Ocean means 4x increase in memory requirements. For such an increase, the time taken is still approx 4x. The Barnes, on the other hand, takes 4x time to go from 16K to 32K (though the memory requirements have just doubled). This happens, again, because of the memory access pattern. Since Barnes requires accessing the entire 'universe' once to process one body, it requires additional data transfers to/from main memory.

Suitability of GPU as a co-processor

This experiment involved monitoring processor and memory usage when the GPU was performing a computation. The measurements were done using the Linux commands top, free and kpm.

The following observations were made:

- GPU keeps one of the cores busy all through the computation while leaving the others idle (in this case 1 of the 4 hyper-threaded cores). Doing the same computation on Clover kept all 8 cores busy.
- The memory usage during GPU computation is same as that on CPU. However, the implementation was keeping a copy of the buffer while the computation was in progress. It is possible to free the memory allocated on main memory, provided the entire data can be fit within device memory (or in other words, Cuda does not require that a host-copy be present).

Accuracy or computation

The measurements were done for the accuracy of the Ocean and Barnes implementation. These implementations require only single precision arithmetic operations. A comparison with output of CPU-based implementation shows that the accuracy was within 6 decimal places.

When it came to trigonometric operations, though, the experiment found that the accuracy is not as good. The results show an accuracy to 3 decimal points. Logarithmic operations showed accuracy to 6 decimal places, but exponential functions and sqrt fell apart with just 3 decimal places accuracy.

5.Conclusion

The GPUs really have come a long way – in terms of programmability and in terms of performance. Cuda has simplified general purpose computing on GPUs to such an extend that implementing the solution to the problem is really only about implementing the solution, and not about identifying the correct geometric shapes to represent one's data or which of the hundreds of data representations should be used on GPU. The fact that GPU memory can be accessed just as randomly as CPU memory (though only from device programs) has increased the suitability of GPU programming to a wider variety of problems.

As the results show here, some of the scientific workloads show great performance improvement (upto 7x) over high-end CPUs. Remember that the GPU configuration used in this project costs less than a fourth of the Clover machine used in this project.

Cuda is currently supported only by nVidia. Chances are it will stay that way for a long time, if not forever. However, Cuda has proved that CPU-like programming flexibility is possible for the highly parallel (16 multiprocessors that run 8 threads simultaneously) and special-purpose GPU hardware. Other manufacturers are following the suite, with Intel working on a 'graphics version of CPU', apparently code-named Larrabee. It is rumored to be as capable as GPUs, with an instruction set closer to the CPUs.

GPUs are definitely not for all applications. But it

works well (extremely well) for certain classes of applications. With a newer, simpler programming models, they are an attractive option, as long as you know your application's characteristics.

6.References

1. SIGGRAPH 2005 GPGPU COURSE on GPGPU, organized by Mark Harris and David Luebke
2. “Mapping computational concepts to GPUs”, International Conference on Computer Graphics and Interactive Techniques, Mark Harris, ACM SIGGRAPH 2005
3. “Taking the Plunge into GPU computing”, Ian Buck, GPU Gems 2, Addison Wesley, Chapter 32.
4. “The Cuda Programming Guide” at www,nVidia.com
5. “Brook for GPUs: stream computing on graphics hardware”, Buck, et.al, International Conference on Computer Graphics and Interactive Techniques archive,ACM SIGGRAPH 2004
6. “GPU flow control idioms”, Harris et.al, GPU Gems 2, Addison Wesley, Chapter 34.
7. “A memory model for scientific algorithms on Graphics Processors”, Govindaraju, et al., SC06 International Conference for High Performance Computing, Networking, Storage and Analysis 2006.
8. “Extended-precision floating-point numbers for GPU computation”, Thall, Andrew, Annual Conference on Computer Graphics, ACM SIGGRAPH 2006
9. “Understanding the efficiency of GPU algorithms for matrix-matrix multiplication”, Fatahalian, et al., SIGGRAPH/EUROGRAPHICS Conference On Graphics Hardware
10. ”Glift: Generic, efficient, random-access GPU data structures”, Lefohn, et.al, ACM Transactions on Graphics, 2006
11. “Implementing efficient parallel data structures on GPUs”, Lefohn et.al, GPU Gems 2, Addison Wesley, Chapter 33.
12. “A cache-efficient sorting algorithm for database and data mining computations using graphics processors”, Govindaraju et.al,
13. “Introduction to GPGPU programming”, Steele et.al, ACM Southeast Regional Conference, 2007.
14. www.gpgpu.org
15. www.opengl.org
16. “The GPU Enters Computing’s Mainstream”, Macedonia, M, IEEE Computer Society Library