ISSN No. 0976-5697

Volume 3, No. 3, May-June 2012

International Journal of Advanced Research in Computer Science

CASE STUDY AND REPORT

Available Online at www.ijarcs.info

High Performance System in GPU and CUDA Media Processing System

M.Chithik Raja
Salalah College of Technology
Department of Information Technology Salalah , Oman
chithik25@gmail.com

Abstract **-** This paper focuses on An Overview of High Performance with GPU and CUDA Media Processing System. The GPU ubiquitous graphics processing unit in every PC, laptop, desktop computer, and workstation. In its most basic form, the GPU generates 2D and 3D graphics, images, and video that enable window based operating systems, graphical user interfaces, video games, visual imaging applications, and video. The modern GPU that we describe here is a highly parallel, highly multithreaded multiprocessor optimized for visual computing. To provide real-time visual interaction with computed objects via graphics images, and video, the GPU has a unified graphics and computing architecture that serves as both a programmable graphics processor and a scalable parallel computing platform. PCs and game consoles combine a GPU with a CPU to form heterogeneous systems.

Keywords**:** *GPU , CUDA, VGA,C++,SPMD*

I. INTRODUCTION

Graphics on a PC were performed by a video graphics array (VGA) controller. A VGA controller was simply a memory controller and display generator connected to some DRAM. In the 1990s, semiconductor technology advanced sufficiently that more functions could be added to the VGA controller. By 1997, VGA controllers were beginning to incorporate some three-dimensional (3D) acceleration functions, including hardware for triangle setup and pasteurization (dicing triangles into individual pixels) and texture mapping and shading (applying “decals” or patterns to pixel sand blending colors).In 2000, the single chip graphics processor incorporated almost every detail of the traditional high-end workstation graphics pipeline and therefore, deserved a new name beyond VGA controller.

The term GPU was coined to denote that the graphics device had become a processor. Over time, GPUs became more programmable, as programmable processors replaced fixed function dedicated logic while maintaining the basic 3D graphicspipeline organization. In addition, computations became more precise over time, progressing from indexed arithmetic, to integer and fixed point, to single precision floating-point, and recently to double precisionfloating-point. GPUs have becomemassively parallel programmable processors with hundreds of cores and thousandsof threads. Recently, processor instructions and memory hardware were added to supportgeneral purpose programming languages, and a programming environment wascreated to allow GPUs to be programmed using familiar languages, including Cand C++. This innovation makes a GPU a fully general-purpose, programmable,manycore processor, albeit still with some special benefits and limitations.GPUs and their associated drivers implement the OpenGL and DirectX models ofgraphics processing. OpenGL is an open standard for 3D graphics programmingavailable for most computers. DirectX is a series of Microsoft multimedia programminginterfaces, including Direct3D for 3D graphics. Since these applicationprogramming interfaces (APIs) have well-defined behaviour, it is possible to buildeffective hardware acceleration of the graphics processing functions defined by theAPIs.

This is one of the reasons (in addition to increasing device density) that newGPUs are being developed every 12 to 18 months that double the performance ofthe previous generation on existing applications[1].Frequent doubling of GPU performance enables new applications that werenot previously possible. The intersection of graphics processing and parallel computing invites a new paradigm for graphics, known as visual computing. Itreplaces large sections of the traditional sequential hardware graphics pipelinemodel with programmable elements for geometry, vertex, and pixel programs.Visual computing in a modern GPU combines graphics processing and parallelcomputing in novel ways that permit new graphics algorithms to be implemented,and open the door to entirely new parallel processing applications on pervasivehigh-performance GPUs.

II. GPU EVOLVES INTO SCALABLE PARALLEL PROCESSOR SYSTEM

GPUs have evolved functionally from hardwired, limited capability VGA controllersto programmable parallel processors. This evolution has proceeded by changingthe logical (API-based) graphics pipeline to incorporate programmable elementsand also by making the underlying hardware pipeline stages less specialized andmore programmable. Eventually, it made sense to merge disparate programmablepipeline elements into one unified array of many programmable processors.In the GeForce 8-series generation of GPUs, the geometry, vertex, and pixelprocessing all run on the same type of processor[2].

This unification allows fordramatic scalability. More programmable processor cores increase the total systemthroughput. Unifying the processors also delivers very effective load balancing,since any processing function can use the whole processor array. At the other endof the spectrum, a processor array can now be built with very few processors, sinceall of the functions can be run on the same processors. This uniform and scalable array of processors invites a new model of programming for the GPU. The large amount of floating-point processing power in the GPU

processor array is very attractive for solving non graphics problems. Given the large degree of parallelism and the range of scalability of the processor array for graphics applications, the programming model for more general computing must express the massive parallelism directly, but allow for scalable execution. **GPU computing** is the term coined for using the GPU for computing via a parallel programming language and API, without using the traditional graphics API and graphics pipeline model[3].

This is in contrast to the earlier **General Purpose computation on GPU (GPGPU)** approach, which involves programming the GPU using a graphics API and graphics pipeline to perform non graphics tasks. **Compute Unified Device Architecture (CUDA)** is a scalable parallel programming model and software platform for the GPU and other parallel processors that allows the programmer to bypass the graphics API and graphics interfaces of the GPU and simply program in C or C++. The CUDA programming model has an SPMD (single-program multiple data) software style, in which a programmer writes a program for one thread that is instanced and executed by many threads in parallel on the multiple processors of the GPU. traditional types of graphics applications plus many new applications.

The original purview of a GPU was "anything with pixels," but it now includes many problems without pixels but with regular computation and/or data structure. GPUs are effective at 2D and 3D graphics, since that is the purpose for which they are designed. Failure to deliver this application performance would be fatal. 2D and 3D graphics use the GPU in its "graphics mode," accessing the processing power of the GPU through the graphics APIs, OpenGLTM, and DirectXTM. Games are built on the 3D graphics processing capability[4]. Beyond 2D and 3D graphics, image processing and video are important applications for GPUs. These can be implemented using the graphics APIs or as computational programs, using CUDA to program the GPU in computing mode. Using CUDA, image processing is simply another data-parallel array program. To the extent that the data access is regular and there is good locality, the program will be efficient[4]. In practice, image processing is a very good application for GPUs. Video processing, especially encode and decode (compression and decompression according to some standard algorithms) is quite efficient.

The greatest opportunity for visual computing applications on GPUs is to "breakthe graphics pipeline." Early GPUs implemented only specific graphics APIs, albeitat very high performance. This was wonderful if the API supported the operationsthat you wanted to do. If not, the GPU could not accelerate your task, because earlyGPU functionality was immutable. Now, with the advent of GPU computing andCUDA, these GPUs can be programmed to implement a different virtual pipelineby simply writing a CUDA program to describe the computation and data flowthat is desired. So, all applications are now possible, which will stimulate new visualcomputing approaches.

III. GPU SYSTEM ARCHITECTURES

In this section, we survey GPU system architectures in common use today. We discuss system configurations, GPU functions and services, standard programming interfaces, and a basic GPU internal architecture. Heterogeneous CPU–GPU System Architecture A heterogeneous computer system architecture using a GPU and a CPU can be described at a high level by two primary characteristics: first, how many functional subsystems and/or chips are used and what are their interconnection technologies and topology; and second, what memory subsystems are available to these functional subsystems [5]. The Historical PC Figure 1 is a high-level block diagram of a legacy PC, circa 1990. The north bridge contains high-bandwidth interfaces, connecting the CPU, memory, and PCI bus. The south bridge contains legacy interfaces and devices: ISA bus (audio, LAN), interrupt controller; DMA controller; time/counter. In this system, the display was driven by a simple frame buffer subsystem known as a VGA (video graphics array) which was attached to the PCI bus.

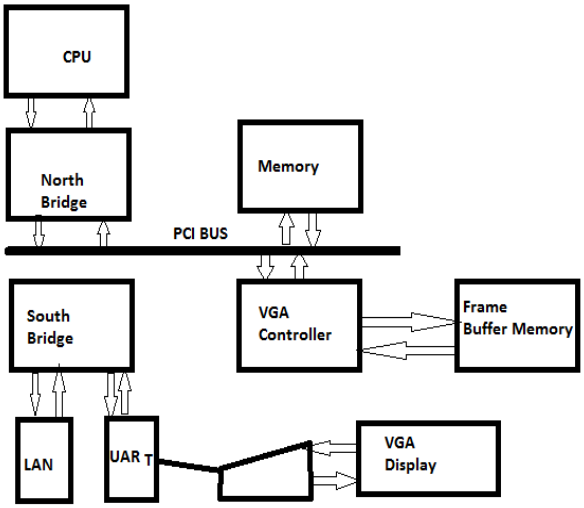

Figure 1 Historical PC. VGA controller drives graphics display from frame buffer memory.

Graphics subsystems with built-in processing elements (GPUs) did not exist in the PC landscape of 1990. Figure.2a illustrates two configurations in common use of Contemporary PCs with Intel and AMD CPU. These are characterized by a separate GPU (discrete GPU) and CPU with respective memory subsystems[6]. In Figure.2 with an Intel CPU, we see the GPU attached via a 16-lane **PCI-Express** 2.0 link to provide a peak 16 GB/s transfer rate, (peak of 8 GB/s in each direction). Similarly, in Figure2b, with an AMD CPU, the GPU is attached to the chipset, also via PCI-Express with the same available bandwidth. In both cases, the GPUs and CPUs may access each other's memory, albeit with less available bandwidth than their access to the more directly attached memories.

In the case of the AMD system, the north bridge or memory controller is integrated into the same die as the CPU. A low-cost variation on these systems, a **unified memory architecture (UMA)** system, uses only CPU system memory, omitting GPU memory from the system. These systems have relatively low performance GPUs, since their achieved performance is limited by the available system memory bandwidth and increased latency of memory access, whereas dedicated GPU memory provides high bandwidth and low latency[7]. A high performance system variation uses multiple attached GPUs, typically two to four working in parallel, with their displays daisy-chained. An example is the NVIDIA SLI (scalable link interconnect) multi-GPU system, designed for high performance gaming and workstations. The next system category integrates the GPU with the north bridge (Intel) or chipset (AMD) with

and without dedicated graphics memory. Console systems such as the Sony PlayStation 3 and the Microsoft Xbox 360 resemble the PC system architectures previously described.

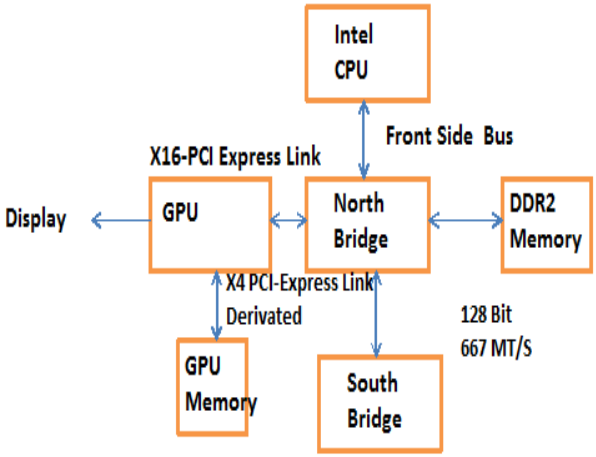

Figure.2a. Contemporary PCs with Intel and AMD CPUs.

Console systems are designed to be shipped with identical performance and functionality over a lifespan that can last five years or more. During this time, a system may be implemented many times to exploit more advanced silicon manufacturing processes and thereby to provide constant capability at ever lower costs. Console systems do not need to have their subsystems expanded and upgraded the way PC systems do, so the major internal system buses tend to be customized rather than standardized.

IV. GPU INTERFACES AND DRIVERS

In a PC today, GPUs are attached to a CPU via PCI-Express. Earlier generations used **AGP**. Graphics applications call OpenGL [Segal and Akeley, 2006] or Direct3D [Microsoft DirectX Specification] API functions that use the GPU as a coprocessor. The APIs send commands, programs, and data to the GPU via a graphics device driver optimized for the particular GPU[10]. The graphics logical pipeline is described in. Figure .3 illustrates the major processing stages, and highlights the important programmable stages (vertex, geometry, and pixel shader stages).

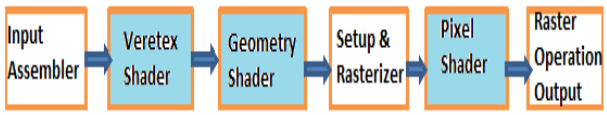

Figure 3 Graphics logical pipeline. Programmable graphics shaded stages are blue, and fixed-function blocks are white.

Basic Unified GPU Architecture Unified GPU architectures are based on a parallel array of many programmable processors[8]. They unify vertex, geometry, and pixel shader processing and parallel computing on the same processors, unlike earlier GPUs which had separate processors dedicated to each processing type. The programmable processor array is tightly integrated with fixed function processors for texture filtering, rasterization, raster operations, anti-aliasing, compression, decompression, display, video decoding, and high-definition video processing. Although the fixed-function processors significantly outperform more general programmable processors in terms of absolute performance constrained by an area, cost, or power budget, we will focus on the programmable processors here. Compared with multicore CPUs, many core GPUs have a different architectural design point, one focused on executing many parallel threads efficiently on many Figure.4 shows how the logical pipeline comprising separate independentprogrammable stages is mapped onto a physical distributed array of processors.processor cores. By using many simpler cores and optimizing for data-parallel behaviour among groups of threads, more of the per-chip transistor budget is devoted to computation, and less to on-chip caches and overhead.

Programming multiprocessor GPUs is qualitatively different than programming other multiprocessors like multicore CPUs. GPUs provide two to three orders of magnitude more thread and data parallelism than CPUs, scaling to hundreds of processor cores and tens of thousands of concurrent threads in 2008. GPUs continue to increase their parallelism, doubling it about every 12 to 18 months, enabled by Moore's law [1965] of increasing integrated circuit density and by improving architectural efficiency. To span the wide price and performance range of different market segments, different GPU products implement widely varying numbers of processors and threads. Yet users expect games, graphics, imaging, and computing applications to work on any GPU, regardless of how many parallelthreads it executes or how many parallel processor cores it has, and they expect more expensive GPUs (with more threads and cores) to run applications faster.

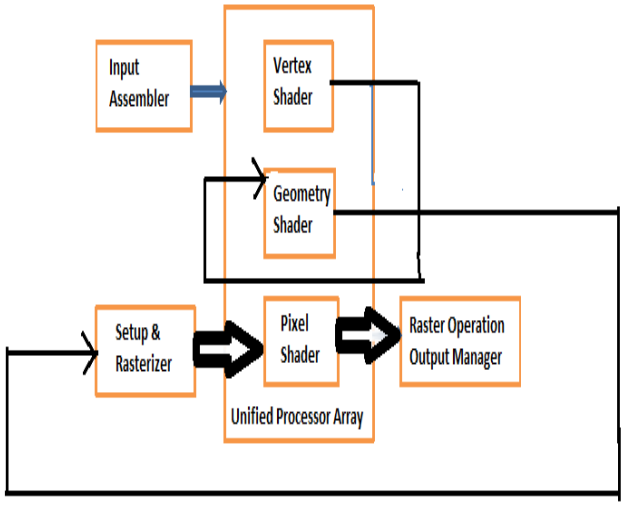

Figure 4 Logical pipeline mapped to physical processors.

The programmable shaderstages execute on the array of unified processors, and the logical graphics pipeline dataflow recirculatesthrough the processors.As a result, GPU programming models and application programs are designed to scale transparently to a wide range of parallelism. The driving force behind the large number of parallel threads and cores in a GPU is real-time graphics performance—the need to render complex 3D scenes with high resolution at interactive frame rates, at least 60 frames per second. Correspondingly, the scalable programming models of graphics shading languages such as Cg (C for graphics) and HLSL (high-level shading language) are designed to exploit large degrees of parallelism via many independent parallel threads and to scale to any number of processor cores[9].

The CUDA scalable parallel programming model similarly enables general parallel computing applications to

leverage large numbers of parallel threads and scale to any number of parallel processor cores, transparently to the application. In these scalable programming models, the programmer writes code for a single thread, and the GPU runs myriad thread instances in parallel. Programs thus scale transparently over a wide range of hardware parallelism. This simple paradigm arose from graphics APIs and shading languages that describe how to shade one vertex or one pixel. It has remained an effective paradigm as GPUs have rapidly increased their parallelism and performance since the late 1990s. This section briefly describes programming GPUs for real-time graphics applications using graphics APIs and programming languages.

It then describes programming GPUs for visual computing and general parallel computing applications using the C language and the CUDA programming model. Programming Real-Time Graphics APIs have played an important role in the rapid, successful development of GPUs and processors. There are two primary standard graphics APIs: **OpenGL** and **Direct3D**, one of the Microsoft DirectX multimedia programming interfaces. OpenGL, an open standard, was originally proposed and defined by Silicon Graphics Incorporated[11]. The ongoing development and extension of the OpenGL standard is managed by Khronos, an industry consortium. Direct3D, a de facto standard, is definedand evolved forward by Microsoft and partners. OpenGL and Direct3D aresimilarly structured, and continue to evolve rapidly with GPU hardware advances.They define a logical graphics processing pipeline that is mapped onto the GPUhardware and processors, along with programming models and languages for theprogrammable pipeline stages.Logical Graphics Pipeline. OpenGL has asimilar graphics pipeline structure.

The API and logical pipeline provide a streaming dataflow infrastructure and plumbing for the programmable shader stages, shown inblue. The 3D application sends the GPU a sequence of vertices grouped into geometricprimitives—points, lines, triangles, and polygons. The input assembler collectsvertices and primitives. The vertex shader program executes per-vertex processing including transforming the vertex 3D position into a screen position and lighting thevertex to determine its color. The geometry shader program executes per-primitiveprocessing and can add or drop primitives. The setup and rasterizer unit generatespixel fragments (fragments are potential contributions to pixels) that are covered bya geometric primitive.

The pixel shader program performs per-fragment processing, including interpolating per-fragment parameters, texturing, and coloring. Pixelshaders make extensive use of sampled and filtered lookups into large 1D, 2D, or3D arrays called **textures**, using interpolated floating-point coordinates. Shaders usetexture accesses for maps, functions, decals, images, and data[12]. The raster operationsprocessing (or output merger) stage performs Z-buffer depth testing and stenciltesting, which may discard a hidden pixel fragment or replace the pixel's depth withthe fragment's depth, and performs a color blending operation that combines thefragment color with the pixel color and writes the pixel with the blended color.The graphics API and graphics pipeline provide input, output, memory objects,and infrastructure for the shader programs that process each vertex, primitive, andpixel fragment.

V. PROGRAMMING PARALLEL COMPUTING APPLICATIONS

CUDA, Brook, and CAL are programming interfaces for GPUs that are focused on data parallel computation rather than on graphics. CAL (Compute Abstraction Layer) is a low-level assembler language interface for AMD GPUs. Brook is a streaming language adapted for GPUs by Buck, et. al. [2004]. CUDA, developed by NVIDIA [2007], is an extension to the C and C++ languages for scalable parallel programming of manycore GPUs and multicore CPUs. With the new model the GPU excels in data parallel and throughput computing, executing high performance computing applications as well as graphics applications. Ata Parallel Problem Decomposition to map large computing problems effectively to a highly parallel processing architecture, the programmer or compiler decomposes the problem into many small problems that can be solved in parallel. For example, the programmer partitions a large result data array into blocks and further partitions each block into elements, such that the result blocks can be computed independently in parallel, and the elements within each block are computed in parallel. Figure 5 shows a decomposition of a result data array into a 3 × 2 grid of blocks, where each block is further decomposed into a 5 × 3 array of elements [13].

The two-level parallel decomposition maps naturally to the GPU architecture: parallel multiprocessors compute result blocks, and parallel threads compute result elements. The programmer writes a program that computes a sequence of result data grids, partitioning each result grid into coarse-grained result blocks that can becomputed independently in parallel. The program computes each result block with an array of fi ne-grained parallel threads, partitioning the work among threads so that each computes one or more result elements. Scalable Parallel Programming with CUDA The CUDA scalable parallel programming model extends the C and C++ languages to exploit large degrees of parallelism for general applications on highly parallel multiprocessors, particularly GPUs. Early experience with CUDA showsThe GeForce 8800 Ultra clocks the SP thread processor cores and SFUs at 1.5 GHz,for a theoretical operation peak of 576 GFLOPS. The GeForce 8800 GTX has a1.35 GHz processor clock and a corresponding peak of 518 GFLOPS.The following three sections compare the performance of a GeForce 8800 GPUwith a multicore CPU on three different applications—dense linear algebra, fastFourier transforms, and sorting. The GPU programs and libraries are compiledCUDA C code. The CPU code uses the single precision multithreaded Intel MKL10.0 library to leverage SSE instructions and multiple cores.

VI. DENSE LINEAR ALGEBRA PERFORMANCE

Another issue is the bit depth of textures that can be used with these techniques. Because the FFT requires many passes, 16 bits per channel are necessary even for non-HDR images. Convolution in spatial domain can be done with only 8 bits per channel in case the application doesnot require higher range. The bit depth has a major impact on the speed – the speed ratios for 8 bit integer and 16 and32 bit floats per channel are approximately 4:2:1.The overview of major pros and cons of the convolution and the FFT is

recapitulated in table 1. All implementations and measurements were done on a machine with 2.6GHz Intel Celeron D processor, 1GBRAM and NVIDIA GeForce 6600 GT 128MB at PCIE. Programs were implemented using HLSL and DirextX9.0c. Both the CPU and GPU are typical mainstream hardware of the beginning of 2005.The GPU implementations of both the FFT and spatial convolutions clearly outperform CPU versions. Figure6 charts performance results for spatial convolution (for textures 256x256 and 512x512) – 32bit GPU version is 3times faster than the CPU implementation and 16bit GPU version gives an average speed up of 7.4 over 32bit CPU implementation (16bit precision brings no advantage on a32bit architecture).Complex FFT of a 256x256 texture took 5.9ms on the GPU while CPU implementation took about 20ms – i.e .the speedup of 3.4. For the CPU FFT algorithm we used FFTW libraries by M. Frigo and S. G. Johnson [2]. It is a heavily-optimized algorithm that uses advanced features of today's CPUs. Multiplying the transformed data array with a filter took additional 20ms on the CPU while it added only about 0.1ms on the GPU.

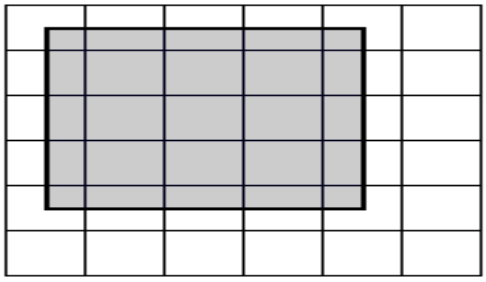

Figure 5: Mapping pixels to Texel's

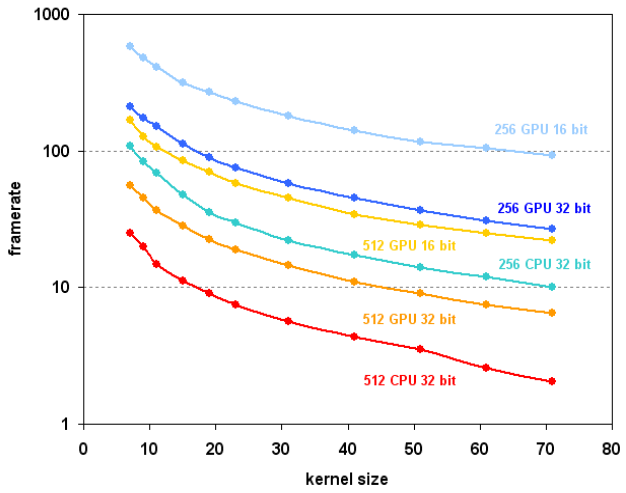

Figure.6. Comparison of GPU and CPU implementationsof convolution (textures 256x256 and 512x512)

VII. CONCLUSION

The results show that GPU implementations of both the convolution and the FFT outperform their CPU counterparts. Convolution on GPU performs better than the FFT on GPU when we want to apply simple and small filters. Separable convolution allows more optimizations, because of or kernels up to approximately 100 pixels in size all the values and texturing coordinates can be pre computed and stored in a uniform data array. This offers a significant speedup in comparison with storing the values in texturiser computing them directly. GPUs are massively parallel processors and have become widely used, not only for 3D graphics, but also for many other applications. This wide application was made possible by the evolution of graphics devices into programmable processors.

The graphics application programming model for GPUs is usually an API such as DirectXTM or OpenGLTM. For more general-purpose computing, the CUDA programming model uses an SPMD (single-program multiple data) style, executing a program with many parallel threads. GPU parallelism will continue to scale with Moore's law, mainly by increasing the number of processors. Only the parallel programming models that can readily scale to hundreds of processor cores and thousands of threads will be successful in supporting manycore GPUs and CPUs. Also, only those applications that have many largely independent parallel tasks will be accelerated by massively parallel many core architectures. GPU architecture will continue to adapt to the usage patterns of both graphics and other application programmers. GPUs will continue to expand to include more processing power through additional processor cores, as well as increasing the thread and memory bandwidth available for programs.

VIII. REFERENCES

[1]. S. Ryoo, C. I. Rodrigues, S. S. Stone, S. S. Baghsorkhi, S. Ueng, J. A. Stratton, and W. Hwu. Program optimization space pruning for a multithreaded GPU. In CGO, April 2008.

[2]. J. D. Owens, D. Luebke, N. Govindaraju, M. Harris, J. Kr¨uger, A Lefohn, and T. J. Purcell. Asurvey of general-Purpose computation on graphics hardware. Computer Graphics Forum, 26(1):80–113, 2007.

[3]. J. Guo, G. Bikshandi, B. B. Fraguela, and D. Padua. Programming with tiles. In PPoPP'08: Proceedings of the 13th ACM SIGPLAN Symposium on Principles of Parallel Programming, 2008.

[4]. M. M. Baskaran, U. Bondhugula, S. Krishnamoorthy, J. Ramanujam, A. Rountev, and Sadayappan. Automatic data movement and computation mapping for multi-levelparallel architectures with explicitlmanagedmemories. In PPoPP '08: Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, 2008.

[5]. Amada, T., Imura, M., Yasumuro, Y., Manabe, (2003). Particle-based fluid simulation the GPU. Proc. ACM Workshop on General-purpose Computing on Graphics

[6]. Dalrymple, R.A., Gómez-Gesteira, M., Rogers, B.D., Panizzo,A., Crespo, A.J.C., Cuomo, Narayanaswamy, M.(2009). Smoothed particle hydrodynamics forwater waves., World Scientific Publishing.

[7]. http://wiki.manchester.ac.uk Green, S. (2008). CUDA Particles. Technical Report contained in the CUDA SDK, www.nvidia.com.

[8]. Levada A. M. L., Mari J. F., Saito J. H. (2007). Voice Command Recognition with Dynamic Time Warping (DTW) using Graphics Processing Units (GPU) with Compute Unified Device Architecture(CUDA), SBAC-PAD International Symposium on Computer Architecture and High Performance computing, 2007, pages 19-27

[9]. Fukushima K. and Miyake S. (2010). Neocognitron: A New Algorithm for Pattern Recognition Tolerant of Deformations and Shift in Position, Pattern Recognition, Vol.15, pages 455-469

[10]. S. H. Yoo, J. H. Park, C. S. Jeong, "Accelerating Multiscale Image Fusion Algorithms Using CUDA," International Conference of Soft Computing and Pattern Recognition, SOCPAR '09, pp. 278-282, 2009.

[11]. S. P. Mohanty, N. Pati, and E. Kougianos, "A watermarking co-processor for new generation graphics processingunits," In Consumer Electronics, 2007. Digest of Technical Papers. International Conference on, pp. 1– 2, 2007.

[12]. C. T. Li, "Digital fragile watermarking scheme for authentication of jpeg images," EE Proc.-Vis. Image Signal Process., Vol. 151, pp. 460 – 466, 2010.

[13]. H. Kourkchi and S. Ghaemmaghami, "Improvement to semi-fragile water marking scheme against a proposed counterfeiting attack," Advanced Communication Technology, 2009. 11th International Conference on, Vol. 03, pp. 1928-1932, 2009

Short Bio Data for the Author

M.Chithik Raja MSc.,M.E.(PhD)., He has finished his Master Degree in M.S.S.Wakf Board College at Madurai. Master of Engineering is awarded by Anna University Chennai Affiliation, Tamilnadu. Now He is pursuing his research in Network Security and System Architecture. He has written more than 8 Reputed International Journal and Conference Proceedings. He has published 3 International Standard Academic Books. He has more than 11 years of Academic Experience in International level Technological College as well as University. He is font of conducting workshop and witting Books for Recent Communication Technologies System Architecture.