Computer System Architecture

By Prof Morris Lancaster

Term Paper

Graphics Processing Units

By,

Anirudh Pandey

G34824024

Table of Contents

**I. INTRODUCTION1**

i. Graphics Processing2

ii. Graphics Pipeline3

**II. EVOLUTION9**

A. 1960s Origins9

B. 1970s Uncovering Basics9

C. 1980s Hardware Acceleration10

D. 1990s Standardization, Consolidation10

i. Generation 010

ii. Generation I11

iii. Generation II11

E. 2000s Programmability, Ubiquity12

i. Generation III12

ii. Generation IV12

iii. Generation V12

iv. Generation VI13

**III. THE MODERN GPU13**

Generation VII; 2010 and Beyond 13

**IV.** **REAL TIME APPLICATIONS14**

A. Data Parallel Processing14

B. Parallel Languages15

C. Image and Signal Processing16

D. Engineering Analysis and Physical Simulation16

E. Database Management17

F. Financial Services17

G. Molecular Biology18

**V. THE FUTURE18**

**VI. CONCLUSION19**

REFERENCES19

**Graphics Processing Units (GPU)**

**Term Paper Abstract**

The Intentionof this term paper is to provide on an overview of Graphics Processing Units (GPU). GPUs are used everywhere from 3-D applications, High End Graphic games to Real Time Simulation or Animation in the modern day computers. GPUs became more popular as the demand for graphic applications increased. Eventually, they became not just an enhancement but a necessity for optimum performance of a PC. Specialized logic chips now allow fast Graphic and Video implementations. This term paper highlights the importance of GPU’s in the emerging world of computers with practical applications of GPU’s.

1. **INTRODUCTION**

In the early 1990s, ubiquitous interactive 3D graphics was still the stuff of science fiction. By the end of the decade, nearly every new computer contained a graphics processing unit (GPU) dedicated to providing a high-performance, visually rich, interactive 3D experience.

This dramatic shift was the in-evitable consequence of consumer demand for videogames, advances in manufacturing technology, and the exploitation of the inherent parallelism in the feed-forward graphics pipeline. Today, the raw computational power of a GPU dwarfs that of the most powerful CPU, and the gap is steadily widening.

Over the past 40 years, dedicated graphics processors have made their way from research labs and flight simulators to commercial workstations and medical devices and later to personal computers and entertainment consoles. The most recent wave has been to cell phones and auto mobiles. As the number of transistors in the devices has begun to exceed those found in CPUs, attention has focused on applying the processing power to computationally intensive problems beyond traditional graphics rendering.

Furthermore, GPUs have moved away from the traditional fixed-function 3D graphics pipeline toward a flexible general-purpose computational engine. Today, GPUs can implement many parallel algorithms directly using graphics hardware. Well-suited algorithms that leverage all the underlying computational horsepower often achieve tremendous speedups. Truly, the GPU is the first widely deployed commodity desktop parallel computer.

The very first computer graphics film ever on computer. The processing was done on an IBM 7090 or 7094 series computer (Image 1).



(Image 1)

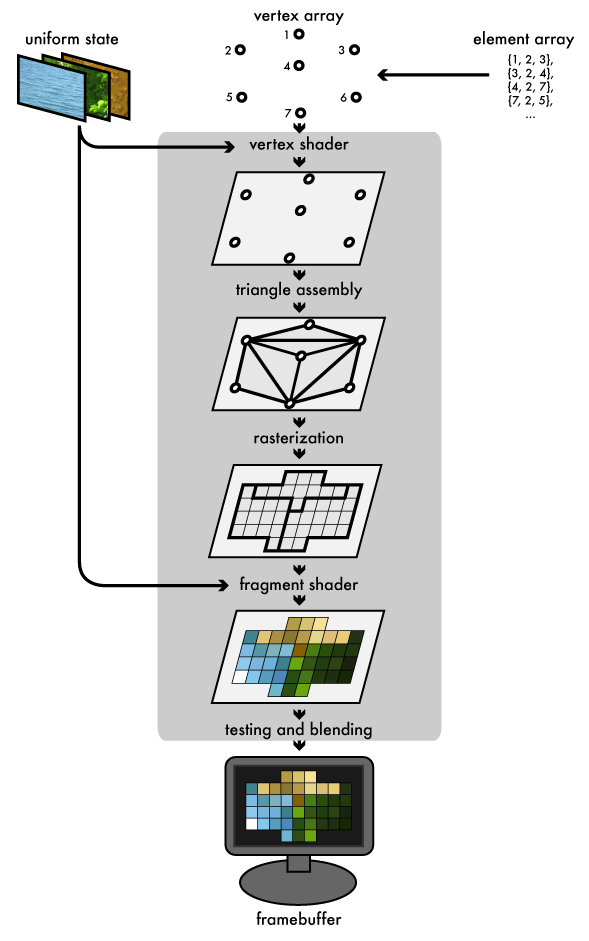
1. **Graphics Processing**

Graphics processors are employed to accelerate a variety of tasks ranging from drawing the text and graphics in an internet web browser to more sophisticated synthesis of three-dimensional (3-D) imagery in computer games. The paper would briefly describe the nature of processing necessary for the 3-D image synthesis fundamental to many of the application areas. Other applications of graphics processing use a subset of this 3-D processing capability.

1. **Graphics Pipeline**

The original GPUs were modeled after the concept of a graphics pipeline. The graphics pipeline is a conceptual model of stages that graphics data is sent through, and is usually implemented via a combination of hardware (GPU cores) and CPU software (OpenGL, DirectX). The graphics pipeline design approach is fairly consistent among the major GPU manufacturers like NVIDIA, ATI, etc., and helped accelerate GPU technology adoption.

The task of any 3D graphics system is to synthesize an image from a description of a scene—60 times per second for real-time graphics such as videogames. Ever since the early days of real-time 3D, the triangle has been the paintbrush with which scenes have been drawn. Although modern GPUs can perform all sorts of flashy effects to cover up this dirty secret, underneath all the shading, triangles are still the medium in which they work.



The Graphics Pipeline

**The vertex and element arrays**

A rendering job starts its journey through the pipeline in a set of one or more vertex buffers, which are filled with arrays of vertex attributes. These attributes are used as inputs to the vertex shader. Common vertex attributes include the location of the vertex in 3d space, and one or more sets of texture coordinates that map the vertex to a sample point on one or more textures. The set of vertex buffers supplying data to a rendering job are collectively called the vertex array. When a render job is submitted, we supply an additional element array, an array of indexes into the vertex array that select which vertices get fed into the pipeline. The order of the indexes also controls how the vertices get assembled into triangles later on.

**Uniform state and textures**

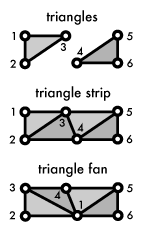
A rendering job also has uniform state, which provides a set of shared, read-only values to the shaders at each programmable stage of the pipeline. This allows the shader program to take parameters that don't change between vertices or fragments. The uniform state includes textures, which are one-D, two-D, or three-dimensional arrays that can be sampled by shaders. As their name implies, textures are commonly used to map texture images onto surfaces. They can also be used as lookup tables for precalculated functions or as datasets for various kinds of effects.

**The vertex shader**

The GPU begins by reading each selected vertex out of the vertex array and running it through the vertex shader, a program that takes a set of vertex attributes as inputs and outputs a new set of attributes, referred to as varying values that get fed to the rasterizer. At a minimum, the vertex shader calculates the projected position of the vertex in screen space. The vertex shader can also generate other varying outputs, such as a color or texture coordinates, for the rasterizer to blend across the surface of the triangles connecting the vertex.

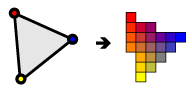
**Triangle assembly**

The GPU then connects the projected vertices to form triangles. It does this by taking the vertices in the order specified by the element array and grouping them into sets of three. The vertices can be grouped in a few different ways:



* Take every three elements as an independent triangle
* Make a **triangle strip**, reusing the last two vertices of each triangle as the first two vertices of the next
* Make a **triangle fan**, connecting the first element to every subsequent pair of elements

**Rasterization**



The rasterizer takes each triangle, clips it and discards parts that are outside of the screen, and breaks the remaining visible parts into pixel-sized fragments. As mentioned above, the vertex shader's varying outputs are also interpolated across the rasterized surface of each triangle, assigning a smooth gradient of values to each fragment. For example, if the vertex shader assigns a color value to each vertex, the rasterizer will blend those colors across the pixelated surface as shown in the diagram.

**The fragment shader**

The generated fragments then pass through another program called the fragment shader. The fragment shader receives the varying values output by the vertex shader and interpolated by the rasterizer as inputs. It outputs color and depth values that then get drawn into the framebuffer. Common fragment shader operations include texture mapping and lighting. Since the fragment shader runs independently for every pixel drawn, it can perform the most sophisticated special effects; however, it is also the most performance-sensitive part of the graphics pipeline.

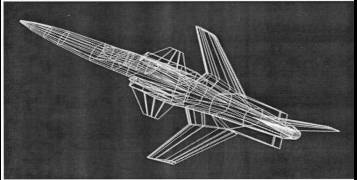
**Framebuffers, testing, and blending**

A framebuffer is the final destination for the rendering job's output. In addition to the default framebuffer OpenGL gives you to draw to the screen, most modern OpenGL implementations let you make framebuffer objects that draw into offscreen renderbuffers or into textures. Those textures can then be used as inputs to other rendering jobs. A framebuffer is more than a single 2d image; in addition to one or more color buffers, a framebuffer can have a depth buffer and/or stencil buffer, both of which optionally filter fragments before they are drawn to the framebuffer: Depth testing discards fragments from objects that are behind the ones already drawn, and stencil testing uses shapes drawn into the stencil buffer to constrain the drawable part of the framebuffer, "stencilling" the rendering job. Fragments that survive these two gauntlets have their color value alpha blended with the color value they're overwriting, and the final color, depth, and stencil values are drawn into the corresponding buffers.

1. **EVOLUTION**
2. **1960s The Origins**

The earliest applications driving the development of computer graphics were computer-aided design and flight simulation. Following close behind were entertainment applications such as computer games and content creation for film. Early graphics processing focused on controlling an analog vector display to stroke a line or wire frame representation of an object by tracing the object’s shape with the electron beam. Displays supporting animated drawing required periodic refresh by redrawing the object, whereas a static figure could be drawn once on a long persistence storage display.

Concurrently, research led to development of algorithms to project a 3-D object representation on to a 2-D plane and simulate linear perspective, as well as methods for doing hidden-line and hidden -surface elimination. By the mid-1960s, the first graphics terminals with autonomous display processing and hardware-computed (accelerated) line-drawing commands were introduced commercially. These are likely the earliest ancestors of the modern graphics processor



First Generation Wireframe

1. **1970s Uncovering Basics**

With the introduction of semiconductor memory, vector display techniques gave way to raster techniques using a discretely sampled (pixel) representation of an image. This was combined with a television-like display device that scanned an electron beam from left to right and top to bottom to display the image-- the modern-day computer monitor.

The use of raster representations also inspired research into a variety of areas. There was also considerable interest in algorithms for rapid generation of images, including those amenable to dedicated acceleration hardware, in the pursuit of interactive generation of complex images.

1. **1980s Hardware Acceleration**

Leading up to the early 1980's, the "GPUs" of the time were really just integrated frame buffers. They were boards of TTL logic chips that relied on the CPU, and could only draw wire-frame shapes to raster displays. The term “GPU” would not be introduced until 1999 by NVIDIA, but for consistency, it will be used throughout this paper. The IBM Professional Graphics Controller (PGA) was one of the very first 2D/3D video cards for the PC. The PGA used an on-board Intel 8088 microprocessor to take over processing all video related tasks. Though it was released in 1984, 10 years before 2D/3D hardware acceleration was standardized, lack of compatibility with many programs and non-IBM systems at the time, made it unable to achieve mass-market success.

The PGA's separate on-board processor marked an important step in GPU evolution to further the paradigm of using a separate processor for graphics computations. By 1987, more features were being added to early GPUs, such as Shaded Solids, Vertex lighting, Rasterization of filled polygons, and Pixel depth buffer, and color blending. There was still much reliance on sharing computation with the CPU.

In the late 1980's, Silicon Graphics Inc. (SGI) emerged as a high performance computer graphics hardware and software company. With the introduction of OpenGL in 1989, SGI created and released the graphics industry's most widely used and supported, platform independent, 2D/3D application programming interface (API). OpenGL support has also become an intricate part of the design of modern graphics hardware. SGI also pioneered the concept of the graphics pipeline early on.

Second Generation Shaded Solids

1. **1990s Standardization, Consolidation**
2. **Generation 0**

GPU hardware and the graphics pipeline started to really take shape in 1993, when SGI released its RealityEngine board for graphics processing. There were distinct boards and logic chips for various later stages of the graphics pipeline, but still relied on the CPU for the first half. Data had a fixed flow through each stage.

By the mid 1990's SGI cards were mainly found on workstations, while 3D graphics hardware makers 3DFX (Voodoo), NVIDIA (TNT), ATI (Rage) and Matrox started providing consumer 3D graphics boards. A combination of "cheap" hardware with games such as Quake and Doom really drove the gaming industry and GPU adoption.

Even with deep pipelines though, early GPUs still only output one pixel per clock cycle, meaning CPUs could still send more triangles to the GPU than it could handle. This lead to adding more pipelines in parallel to the GPU (and ultimately more cores), so multiple pixels could be processed in parallel each clock cycle.



Evolution of Games with GPU’s

1. **Generation I**

The 3dfx Voodoo (1996) was considered one of the first true 3D game cards. It only offered 3D acceleration, so you still needed a 2D accelerator. It operated over the PCI bus, had 1 million transistors, 4 MB of 64-bit DRAM, and the core clock operated at 50 MHz. The CPU still did vertex transformations, while the Voodoo provided texture mapping, z-buffering, and rasterization.

1. **Generation II**

In 1999, the first cards to implement the entire pipeline (now with transform and lighting calculations) in GPU hardware were released. With the introduction of NVIDIA's GeForce256 and ATI's Radeon 7500, the first true GPUs were available at the consumer level.

Up until 1999, the term "GPU" didn't actually exist and NVIDIA coined the term during its launch of the GeForce 256. The GeForce 256 had 23 million transistors; 32 MB of 128-bit DRAM, the core clock operated at 120MHz, and had four 64-bit pipelines for rendering. This generation of cards were the first to use the new Accelerated Graphics Port (AGP) instead of the PCI bus and offered new graphics features in hardware, such as multi-texturing, bump maps, light maps and hardware geometry transform and lighting.

The first pipelines now in hardware were known as a "fixed function" pipeline, because once the programmer sent graphics data into the GPU's pipeline, the data could not be modified. With these cards, the GPU hardware and computer gaming market really started to take off. While much faster, the main problem with the fixed function pipeline model was the inflexibility of graphical effects.

1. **2000s Programmability, Ubiquity**
2. **Generation III**

The GPU hardware evolved with the introduction of the programmable pipeline on the GPU. In 2001, NVIDIA released the GeForce 3 which gave programmers the ability to program parts of the previously non-programmable pipeline. Instead of sending all the graphics description data to the GPU and have it simply flow through the fixed pipeline, the programmer can now send this data along with vertex "programs"(called shaders) that operate on the data while in the pipeline.

These shader programs were small "kernels" written in assembly-like shader languages. For the first time, there was limited amount of programmability in the vertex processing stage of the pipeline. Other popular cards at this time were ATI Radeon 8500.

1. **Generation IV**

One year later, in 2002, the first fully programmable graphics cards hit the market: NVIDIA GeForce FX, ATI Radeon 9700. These cards allowed for per-pixel operations with programmable vertex and pixel (fragment) shaders, and allowed for limited user-specified mapping of input-output data operations. Separate, dedicated hardware were allocated for pixel shader and vertex shader processing.

The first GeForce FX had 80 million transistors, 128 MB of 128-bit DDR DRAM, and the core clock operated at 400MHz.

In 2003, the first wave of GPU computing started to come about with the introduction of DirectX 9, by taking advantage of the programmability now in the GPU hardware, but for non-graphics. Full floating point support and advanced texture processing started to appear in cards.

1. **Generation V**

By this time, the rate of GPU hardware technology was accelerating at a rate much faster than Moore's Law. In 2004, the GeForce 6 and Radeon X800 were released, and were some of the first cards to use the PCI-express bus. For software, early high level GPU languages such as Brook and Sh started to appear, that offered true conditionals and loops and dynamic flow control in shader programs. On the hardware side, higher precision (64-bit double support), multiple rendering buffers, increased GPU memory and texture accesses were being introduced.

1. **Generation VI**

The introduction of NVIDIA's GeForce 8 series GPU in 2006 marked the next step in GPU evolution exposing the GPU as massively parallel processors. The G80 (GeForce 8800) architecture was the first to have "unified", programmable shaders - in other words a fully programmable unified processor called a Streaming Multiprocessor or SM, that handled vertex, pixel, and geometry computation. Also introduced was a new geometry shader, adding more programmability when combined with the vertex shader and pixel shaders. With a now unified shader hardware design, the traditional graphics pipeline model is now purely a software abstraction.



To harness all this "general purpose" GPU power was the new programming language CUDA by NVIDIA for NVIDIA. Not much later, ATI Stream for ATI cards and DirectX 10 for either card (Microsoft Windows) were introduced.

1. **THE MODERN GPU**

**Generation VII; 2010 and Beyond**

The trend towards more CPU-like, programmable GPU cores continues with the introduction of NVIDIA's Fermi architecture. Announced in late 2009, but released in early 2010, the Fermi GPU was the first GPU designed for GPGPU computing, bringing features such as: true HW cache hierarchy, ECC, unified memory address space, concurrent kernel execution, better double precision performance, and dual warp schedulers. The GTX480 Fermi had a total of 480 CUDA cores at launch.

The GPU hardware evolution thus far has gone from an extremely specific, single core, fixed function hardware pipeline implementation just for graphics rendering, to a set of highly parallel and programmable cores for more general purpose computation. Now, the architecture of many-core GPUs are starting to look more and more like multi-core, general purpose CPUs. In that respect, Fermi can essentially be thought of as a 16-core CPU with 32-way hyper-threading per core, with a wide vector width.

AMD recently announced their Fusion line of CPU+GPUs on the same die (dubbed APU, accelerated processing unit), to be released early 2011. AMD's APUs are designed so that a standard x86 processor for scalar workloads and a DX11 GPU for vector workloads are brought together on the same die. Intel's Larrabee processor brings many smaller x86 cores on a single die, augmented by a wide vector unit. Intel's SandyBridge CPU also has an integrated GPU which both cores share the L3 cache.

1. **REAL TIME APPLICATIONS**

The combination of raw performance and programmability has made the GPU an attractive vehicle for applications beyond the general realms. Development of nongraphical applications on graphics accelerators also has a rich history with research projects dating back to more than a decade.

1. **Data Parallel Processing**

In computers, parallel processing is the processing of program instructions by dividing them among multiple processors with the objective of running a program in less time. In the earliest computers, only one program ran at a time. A computation-intensive program that took one hour to run and a tape copying program that took one hour to run would take a total of two hours to run. An early form of parallel processing allowed the interleaved execution of both programs together. The computer would start an I/O operation, and while it was waiting for the operation to complete, it would execute the processor-intensive program. The total execution time for the two jobs would be a little over one hour.

Vector processing was another attempt to increase performance by doing more than one thing at a time. In this case, capabilities were added to machines to allow a single instruction to add (or subtract, or multiply, or otherwise manipulate) two arrays of numbers. This was valuable in certain engineering applications where data naturally occurred in the form of vectors or matrices. In applications with less well-formed data, vector processing was not so valuable. The next step in parallel processing was the introduction of multiprocessing. In these systems, two or more processors shared the work to be done. The earliest versions had a master/slave configuration. One processor (the master) was programmed to be responsible for all of the work in the system; the other (the slave) performed only those tasks it was assigned by the master. This arrangement was necessary because it was not then understood how to program the machines so they could cooperate in managing the resources of the system.

Research in parallel programming has produced a set of basic operators for data parallel processing. Parallel algorithms are constructed from these operations. The point-wise processing algorithm corresponds to the map operator that applies a function to each element in a domain. Other operators include reduce, gather, scatter, scan, select, sort, and search. GPU architecture is specifically tailored to rendering tasks with no underlying intent to support more general parallel programming. This makes mapping some of the data parallel operators onto the GPU a challenge. To alleviate these difficulties, efforts have been made to create reusable GPU data structure libraries.

1. **Parallel Languages**

As GPUs are designed for running graphics applications, most of the development for

programming languages and APIs has been targeted at writing graphics applications.

This makes non graphics programming more challenging as the programmer must master idioms from the graphics APIs and languages, such as drawing triangles to create a set of domain points and trigger fragment processing across that domain. Shading programs must be written to process the domain points, using texture mapping operations to read data associated with each domain point and writing the computed result as a color value.

To simplify this programming task and hide the underlying graphics idioms, several programming languages and runtimes have been created. These ranges from systems to graphics vendors that expose low-level details of the underlying graphics hardware implementation to research and commercial higher-level languages and systems intended to simplify development of data parallel programs. These systems cover a spectrum of approaches including more traditional array processing style languages to newer stream processing languages (derived from parallel processing research in specialized stream processing hardware. In many cases, the language combines the parts of the code that execute on the CPU and the parts that execute on the GPU in a single program. This differs from many of the graphics APIs (OpenGL, Direct3D) that deliberately make the boundary between the CPU and GPU explicit.

One advantage of higher level languages is that they preserve high-level information that can potentially be used by the underlying runtime to manage execution and memory coherence. In contrast, lower level systems leave that largely up to the programmer, requiring the programmer to learn various architectural details to approach peak efficiencies. Low-level systems may allow programmers to achieve better performance at the cost of portability, but they also may allow access to newer, more efficient processing constructs that are not currently available in the graphics APIs. One example of this is additional support for explicit scatter operations and sharing on-chip memory available in NVIDIA’s CUDA system. These may prove beneficial, though also at the cost of portability at least initially.

1. **Image and Signal Processing**

One of the more obvious application areas is general image and signal processing on 1D, 2D, or 3D data. Graphics applications already make use of image-processing operations such as sampling and filtering in texture mapping operations. Extensions for fixed-function imaging operations, such as histogram and convolution, were added as extensions to the OpenGL architecture in the late 1990s, but they have effectively been subsumed by the programmable pipeline. A number of projects have implemented convolution with various size kernels, fast Fourier transforms (FFTs), segmentation, and histograms. Such signal-processing methods are often critical in analysis applications. For example, GPU accelerated backprojection accelerates reconstruction from sensor data sufficiently fast to allow interactive scanning and visualization of tomograms. The reconstruction process involves taking the multiple  
 2-D projections of 3-D data captured by the sensors and applying an approximate inverse Radon transform. In geophysical analysis, wavelet compression can significantly reduce (100x) the size of raw seismic sensor data.

1. **Engineering Analysis and Physical Simulation**

Computationally intensive physical simulations (fluids, weather, crash analysis, etc.) have traditionally relied on high-performance computing (HPC) systems (supercomputers). The processing power of GPUs is attractive in this area for two reasons. One is the opportunity to provide lower cost solutions for solving these research and industrial problems, perhaps using the cost differential to solve even larger problems. Secondly, physical simulations are becoming an increasingly important way of adding new types of behaviors to entertainment applications (e.g., better character animations; more complex, but not necessarily realistic, interactions with the physical world).

There are many types of problems that fit in this simulation category, but two important classes of problems are solving ordinary and partial differential equations. Ordinary differential equations are functions of a single variable containing one or more derivatives with respect to that variable. A simple example is Newton’s law of acceleration (force = mass x acceleration). In computer graphics, it is common to model objects as collections of particles (smoke, fluids) and to solve this equation for each of the particles using an explicit integration method. Such a method computes the new value (e.g., position at a new time step) using the computed values from previous time steps. If the particles are independent and subject to global forces, then each particle can be computed independently and in parallel. If the particles also exert local forces on one another then information about neighboring particles must also be collected (from the previous time step) and used in the computation. Several GPU-based simulators that model the physics and render the results have been built, showing speed improvements of 100 or more over CPU-based algorithms.

GPU implementations have been applied to simulation problems such as fluid flow and reaction-diffusion.

1. **Database Management**

Another interesting area is data management problems including database operations such as joins and sorting on large amounts of data. The large memory bandwidth makes the GPU very attractive, but maintaining coherence in sorting algorithms is challenging. This has led to a resurgence of interest in sorting networks such as the bitonic sort, which can be mapped relatively efficiently onto a GPU. GPUs are particularly effective in performing out of core sorting, that is, a sort where the number of records is too large to fit in system memory and must be read from disk. The GPUTeraSort implements a CPU–GPU hybrid system in which the CPU manages disk transfers while the GPU performs the bandwidth-intensive sorting task on large blocks of keys. The result is a system that is more cost-effective at sorting a large collection of records than a similarly priced system using only CPUs and no GPU, as measured by the Penny Sort benchmark.

In addition to bulk data processing, the graphics processing ability of GPUs can also be leveraged for specialized database queries such as spatial queries used in mapping or geospatial processing.

1. **Financial Services**

The financial services industry has produced increasingly complex financial instruments requiring greater computational power to evaluate and price. Two examples are the use of Monte Carlo simulations to evaluate credit derivatives and evaluation of Black–Scholes models for option pricing. More sophisticated credit derivatives that do not have a closed-form solution can use a Monte Carlo (nondeterministic) simulation to perform numerical integration over a large number of sample points. PeakStream has demonstrated speed increases of a factor of 16 over a CPU when using a high-end GPU to evaluate the differential equation and integrate the result.

The Black–Scholes model prices a put or call option for a stock assuming that the stock prices follow a geometric Brownian motion with constant volatility. The resulting partial differential equation is evaluated over a range of input parameters (stock price, strike price, interest rate, expiration time, volatility) where the pricing equation can be evaluated in parallel for each set of input parameters. Executing these equations on a GPU for a large number of inputs (15 million), the GPU approaches a factor of 30 faster than a CPU using PeakStream’s software and 197 times faster using CUDA(measured on different GPU types).

1. **Molecular Biology**

There have been several projects with promising results in the area of molecular biology, in particular, with the analysis of proteins. Molecular dynamics simulations are used to simulate protein folding in order to better understand diseases such as cancer and cystic fibrosis. Stanford University’s folding@home project has created a distributed computational grid using volunteer personal computers. The client program executes during idle periods on the PC performing simulations. Normally, PC clients execute the simulation entirely on the CPU, but a version of the molecular dynamics program gromacs has been written for the GPU using the Brook programming language. This implementation performs as much as a factor of 20–30 faster than the CPU client alone.

Another example is protein sequence analysis using mathematical models based on hidden Markov models (HMMs). A model for a known program sequence (or set of sequences) with a particular function on homology is created. A parallel algorithm is used to search a large database of proteins by computing a probability of each search candidate being in the same family as the model sequences. The probability calculation is complex, and many HMMs may be used to model a single sequence, so a large amount of processing is required. The GPU version of the algorithm uses data-parallel processing to simultaneously evaluate a group of candidates from the database. This makes effective use of the raw processing capabilities and provides an order of magnitude or better speedup compared to a highly tuned sequential version on a fast CPU.

1. **THE FUTURE**

Despite years of research, deployment of highly parallel processing systems has, thus far, been a commercial failure and is currently limited to the high-performance computing segment. The graphics accelerator appears to be an exception, utilizing data-parallel techniques to construct systems that scale from a 1/2 Tflop of floating-point processing in consoles and PCs to fractions of that in handheld devices, all using the same programming model. Some of this success comes from focusing solely on the graphics domain and careful exposure of a programming model that preserves the parallelism. Another part comes from broad commercial demand for graphics processing. The question on many people’s minds is whether this technology can be successfully when applied commercially to other application spaces.

1. **CONCLUSION**

Graphics processors have undergone a tremendous evolution over the last 4 decades, in terms of expansion of capabilities and increases in raw performance. The popularity of graphical user interfaces, presentation graphics, and entertainment applications has made graphics processing ubiquitous through an enormous installed base of personal computers and cell phones.

GPUs tremendous computational capacity and rapid growth, far transcended traditional CPUs, highlight the advantages of domain-specialized data-parallel computing. We can expect increased programmability and generality from future GPU architectures. Today, GPU developers need new high-level programming models for massively multithreaded parallel computation. So do GPU vendors, graphics developers, and the GPGPU research community build on their success with commodity parallel computing to transcend their computer graphics roots and develop the computational idioms, techniques, and frameworks for the desktop parallel computing environment of the future?

**REFERENCES**

* **Crow, Thomas. Evolution of the Graphical Processing Unit. 2004 Paper**
* **Wikipedia: Graphics Processing Unit.**
* **Luebke, David. GPU Architecture: Implications & Trends. SIGGRAPH 2008**
* **Franken, Yannick. Introduction to Graphics Hardware and GPUs. 2008 Lecture.**
* **Göddek, Dominik. GPU Computing with CUDA - Part 1: GPU Computing Evolution. 2009 Lecture.**
* **Seiler, Larry. Larrabee: a many-core x86 architecture for visual computing.**
* **Chris McClanahan : History and Evolution of GPU Architecture A Paper Survey**
* **David Blythe: Rise of the Graphics Processor**
* **F. Yuan, Windows Graphic s Programming: Win 32 GDI and Direct Draw. Englewood Cliff s, NJ: Prentice-Hall , Dec. 2000**
* **R. Bunker and R. Economy, “ Evolution of GE CIG systems,” Simul. Contr . Syst. Dept., General Electric , Daytona Beach, FL, 1989, Tech. Rep.**
* **NVIDIA. (1999). GeForce 256 . [Online].Available: http://www.nvidia.com/page/geforce256.html**
* **AMD . (2000). Radeon 7200 . [Online]Available: http: //www .ati.amd.com /products/radeon7200/**
* **NVIDIA. (1999) . Register combiners OpenGL extension specification. [Online]. Available:http://www.oss.sgi.com/projects/ogl-sample/registry/NV/register\_combiners.txt**
* **G. Bishop, B Gary’s Ikonas assembler, version 2:Difference s between Gia2 and C ,[ Univ. of North Carolina V Chapel Hill, Computer Science Tech. Rep. T R82-010, 1982**
* **Microsoft Corp . (2002). High-level shader language : In DirectX 9 .0 graphics R [Online ]. Available: http:// www .msdn.microsoft.com/directx/**
* **NVIDIA . (2007). GeForce 8800 GPU architecture overview R [Online]. Available:**

**http:// www .nvidia.com/object/IO\_37100.html.**

* **AMD .(2007). ATI Radeon HD 2900 series VGPU specifications R [Online]. Available: http:// www.ati.amd.com/products/Radeonhd2900/specs.html.**
* **PeakStream. (2007). Technology Overview. [Online]. Available: www.peakstreaminc. com/product/overview/**
* **NVIDIA. (2007). CUDA Documentation. [On-line]. Available: http://www.developer.nvi-dia.com/cuda/**
* **PeakStream. (2007). Technology Overview. [Online]. Available: www.peakstreaminc. com/product/overview/**
* **I. Buck, T. Foley, D. Horn, J. Sugerman,K. Fatahalian, M. Houston, and P. Hanrahan, BBrook for GPUs: Stream computing on graphics hardware,[ in Proc. ACM SIGGRAPH 2004, Aug. 2004, pp. 777–786.**
* **U. Kapasi, W. J. Dally, S. Rixner, J. D. Owens, and B. Khailany, The imagine stream processor,[ in Proc. Int. Conf. Comput.Design, Sep. 2002, pp. 282–288.**
* **Headwave Inc. (2007). Technology Overview.[Online]. Available: http://www.**

**headwave.com**

* **D. Horn. (2006). libgpufft. [Online].Available: http://www.sourceforge.net/projects/gpufft**
* **W. Reeves, Particle systems VA technique for modeling a class of fuzzy objects,[ in**

**Proc. ACM SIGGRAPH ’83, Jul. 1983, vol. 17,no. 3, pp. 359–375.**

* **K. Fatahalian, J. Sugerman, and P. Hanrahan, Understanding the efficiency of GPU**

**algorithms for matrix-matrix multiplication,[ in Proc. Graph. Hardware 2004, Aug. 2004, pp. 133–138**

* **J. Kruger, P. Kipfer, P. Kondratieva, and R. Westermann, BA particle system for**

**interactive visualization of 3D flows,[ IEEETrans. Vis. Comput. Graphics, vol. 11, no. 6, pp. 744–756, 2005.**

* **Y. Liu, X. Liu, and E. Wu, Real-time 3D fluid simulation on GPU with complex**

**obstacles,[ in Proc. Pacific Graph. 2004,Oct. 2004, pp. 247–256.**

* **A. Sanderson, M. Meyer, R. Kirby, and C. Johnson, BA framework for exploring numerical solutions of advection-reaction-diffusion equations using a GPU-based**

**approach,[ Comput. Vis. Sci., 2007.**

* **K. E. Batcher, BSorting networks and their applications,[ in AFIPS Spring Joint Comput. Conf., 1968, vol. 32, pp. 307–314.**
* **J. Gray. (2006). Indy penny sort benchmark resultsR [Online]. Available: http://www. research.microsoft.com/barc/ SortBenchmark/**
* **PeakStream. (2007). High performance modeling of derivative prices using the PeakStream platformR [Online]. Available: http://www.peakstreaminc. com/reference/peakstream\_finance\_technote.pdf**
* **Stanford University. (2007). Folding@Home Project. [Online]. Available: http://www. folding.stanford.edu/**
* **I. Buck. (2006, Nov. 13). BGeForce 8800 & NVIDIA CUDA: A new architecture for computing on the GPU,[ in Proc. Supercomput. 2006 Workshop: General Purpose GPU Comput. Practice Exper., Tampa, FL. [Online]. Available: www.gpgpu.org/ sc2006/workshop/presentations/ Buck\_NVIDIA\_Cuda.pdf**
* **Stanford Univ. (2007). Folding@Home on ATI GPU’s: A major step forwardR [Online]. Available: http://www.folding.stanford.edu/ FAQ-ATI.html**
* **D. R. Horn, D. M. Houston, and P. Hanrahan, ClawHMMER: A streaming HMMer-search implementation,[ in Proc. 2005 ACM/IEEE Conf. Supercomput.,**

**Nov. 2005, p. 11**

* **S. Pronovost, H. Moreton, and T. Kelley. (2006, May). Windows**

**display driver model (WDDM) v2 and beyond [Online]. Available: http://www. download.microsoft.com/download/5/b/9/ 5b97017b-e28a-4bae-ba48-174cf47d23cd/ PRI103\_WH06.ppt**

* **J. Stokes. (2007, Jun.). Clearing up the confusion over Intel’s Larrabee,**
* **Part II. [Online]. Available:http://www. arstechnica.com/news.ars/post/2007 0604-clearing-up-the-confusion-over-intels-larrabee-part-ii.html**
* **P. Otellini, BExtreme to mainstream,[ in Proc. IDF Fall 2006, San Francisco, CA. [Online]. Available: http://www.download. intel.com/pressroom/kits/events/ idffall\_2007/KeynoteOtellini.pdf**
* **P. Hester and B. Drebin. (2007, Jul.).2007 Technology Analyst DayR [Online]. Available:http://www.amd.com/us-en/ assets/content\_type/DownloadableAssets/ July\_2007\_AMD\_2Analyst\_Day\_Phil\_ Hester-Bob\_Drebin.pdf**
* **AMD. (2006). Enterprise stream processing[Online]. Available: http://www.ati.amd. com/products/streamprocessor/index.html**
* **NVIDIA. (2006, May). NVIDIA Tesla: CPU computing technical briefR [Online]. Available: http://www.nvidia.com/docs/IO/43395/ Compute\_Tech\_Brief\_v1-0-0\_final.pdf**
* **Intel, B45 nm product press briefing,[ in Proc. IDF Fall 2007, San Francisco, CA. [Online]. Available: http://www.intel.com/ pressroom/kits/events/idffall\_2007/ BriefingSmith45nm.pdf**
* **A. Adl-Tabatabai, C. Kozyrakis, and B. Saha. (2006, Dec.). BUnlocking concurrency:Multicore programming with transactional memory,[ ACM QueueR [Online]. 4(10), pp. 24–33. Available: http://www.acmqueue. org/modules.php?name=Content&pa= showpage&pid=444**
* **David Luebke and Greg Humphreys “How GPUs Work” ,HOW THINGS WORK, p.126 -130**
* **Graphics Pipeline :http://duriansoftware.com/joe/An-intro-to-modern-OpenGL.-Chapter-1:-The-Graphics-Pipeline.html**
* **Ian Buck : The Evolution of GPUs for General Purpose Computing; San Jose Convention Center, CA | September 20–23, 2010. Available:** **www.nvidia.com/content/GTC-2010/pdfs/2275\_GTC2010.pdf**
* **The Evolution of GPUs for General Purpose Computing, par Ian Buck. Available:**

**https://indico.cern.ch/event/305730/session/11/.../29/.../0.pdf**