

# State of the Art Report on GPU Visualization

<b>Prepared by</b>		<b>Institution</b>	
Rita Borgo		The University of Leeds	
Ken Brodlie		The University of Leeds	
<b>Verified by</b>			
John O'Brien		Loughborough University	
<b>Ref:</b>	VIZNET-WP4-24-LEEDS-GPU	<b>Date:</b>	01/05/2009

# State of the Art Report on GPU

Rita Borgo, Ken Brodlie  
University of Leeds

## Summary

This report aims to provide a beginner's introduction to GPUs, from both a hardware and a software angle. We look at the evolution of specialist graphics hardware from the early days of PC graphics cards to the present day. We describe the currently available hardware from NVIDIA and AMD/ATI, and the current software from both the OpenGL family (GLSL) and the Direct3D family (HLSL). We also include non-graphics APIs such as CUDA. The report ends with a look ahead to the future including GPU clusters and the Larrabee architecture.

# State of The Art Report on GPU

\*Visualization and Virtual Reality Research Group  
School of Computing - University of Leeds  
- VizNET REPORT -

Rita Borgo\*      Ken Brodlie\*

February, 2009

# Contents

<b>1</b>	<b>Introduction</b>	<b>3</b>
<b>2</b>	<b>History of GPUs</b>	<b>4</b>
<b>3</b>	<b>Current available hardware architectures</b>	<b>8</b>
3.1	Nvidia GT200 series . . . . .	8
3.2	ATI Radeon R700 series . . . . .	9
<b>4</b>	<b>Programming models &amp; APIs</b>	<b>10</b>
4.1	Graphics APIs . . . . .	10
4.1.1	OpenGL . . . . .	11
4.1.1.1	GLSL . . . . .	11
4.1.1.2	OpenGL   ES . . . . .	11
4.1.2	Direct3D . . . . .	12
4.1.2.1	HLSL . . . . .	15
4.2	Non graphics API . . . . .	16
4.2.1	Vendor specific APIs . . . . .	17
4.2.1.1	CUDA . . . . .	17
4.2.1.2	StreamSDK . . . . .	18
4.2.2	Vendor-independent APIs . . . . .	20
4.2.2.1	RapidMind . . . . .	20
4.2.2.2	OpenCL . . . . .	21
4.2.3	Direct3D11 Compute shader . . . . .	23
<b>5</b>	<b>Future Outlook</b>	<b>24</b>
5.1	Larrabee . . . . .	24
5.2	GPU Clusters . . . . .	25
<b>A</b>	<b>Getting Started</b>	<b>27</b>
A.0.1	The Rendering Pipeline . . . . .	28
A.0.2	Vertex Shader . . . . .	29
A.0.3	Fragment Shader . . . . .	29
A.0.4	Programming the GPU with Python . . . . .	29
A.0.4.1	PyGPU Shader Example . . . . .	29

A.1 Acknowledgments . . . . . 31

# Chapter 1

## Introduction

The present document aims at providing a beginner's introduction to the exciting world of Graphics Processing Units (GPU) and the current state of the art in terms of both hardware and software solutions. Not expecting to be exhaustive we still hope it will be of use to those novice users who wish to know a bit more but found themselves lost in the ocean of material available on the web. We are aware that due to the rapid development of graphics accelerators technologies most of the material overviewed will be soon considered obsolete; however it is also true that given a sound basis updates are easier to develop. We apologize for any missing reference and encourage the reader to contact us to report any suggestions that might improve this manuscript.

The document unfolds into five main sections. Chapter 2 provides a brief history of graphics hardware accelerators from the CGA Hercules to the Radeon 47x0. Chapter 3 surveys some of the most recent hardware architectures available on the market: the NVIDIA GeForce GTX 200 series and the ATI Radeon R700 series. Chapter 4 introduces some of the most commonly used programming models and API platforms available for graphics and general purpose program development on the GPU. Both vendor-dependent and cross platform solutions are addressed. Chapter 5 casts some light on future trends in graphics hardware and GPGPU HPC architectures. A *Getting Started* appendix introduces the concepts at the base of shaders programming and PyGPU, an embedded language for GPU programming with the high level language Python.

## Chapter 2

# History of GPUs

Graphics processors or graphics cards are about as old as the PC itself. The first graphics processor unit for PC was introduced in 1981 from IBM and it was called CGA (Color Graphics Adapter). It was the first colour graphics card and also the first colour computer display standard for the IBM PC. IBM CGA graphics cards were equipped with 16 KBytes of video memory and supported several graphics and text modes. The highest resolution of any mode is 640x200 and the highest colour depth supported is 4bit (16 colours). The graphics card was very primitive and mostly consisted of a RAMDAC and memory; there were no specific raster or blitting operations, that means all the processing / rasterization had to be done on the CPU and the result copied from main memory into the display memory. A year after CGA was introduced by IBM another company released the Hercules Graphics Card (HGC) which offered monochrome (1bit) graphics but with a resolution up to 720x348.

In 1984 with the advance of the IBM AT, the successor to the IBM PC, a new graphics adapter was introduced: EGA (Enhanced Graphics Adapter). EGA allowed displaying of 16 colours simultaneously up to a resolution of 640x350. It included 16KByte of ROM to extend the BIOS for additional graphics functions, included the Motorola MC6845 video address generator and had 64Kbyte of video memory which could be extended to 256Kbyte with an additional daughter board.

EGA was superseded by VGA that was introduced by IBM in 1987 and was the last graphics card standard that had been introduced by IBM as of today. VGA differed quite significantly from the previous standard since it was the first card that came with hardware smooth scrolling, hardware split screen, and raster operations. It had 256Kbyte of Video memory and supported 16 and 256 colour modes out of a palette of 262144 colour values (6bit RGB). Also VGA was the first graphics card that incorporated everything into a single chip and therefore could be classified as predecessor to all modern GPUs.

With the downfall of IBM as being the dominating and standard setting company in the PC market and the introduction of windowing systems like Microsoft Windows 2.1, 3.0 and 3.11, IBM OS/2 and Microsoft Windows95, the

graphics card and graphics chips sector started to bloom and many different vendors and graphics chips came onto the market that all differed in terms of features, display resolutions, and functionality. However, this led to a nightmare for application programmers supporting them which meant that most programmers stuck to the VGA standard which all cards were still compatible to (until today). To attack this problem the VESA (Video Electronics Standard Association) introduced VBE (VESA BIOS Extensions) which provides a common interface to access compliant video boards at high resolutions and bit depths and provided access to the card's linear frame buffer. Many of the graphics cards during that time offered a level of 2D operation mostly bit blitting, double and triple buffering to accelerate the drawing of widgets and windows.

The first graphics chip to introduce 3D acceleration on PC desktop was the Matrox Impression from Matrox Electronic Systems Ltd., an add-on card for the Matrox Millennium aimed for the PC CAD market. While during that time 3D graphics was still dominated by expensive graphics workstations such as those from SGI introducing OpenGL as the first programmable graphics API for 3D and 2D operations. The first real breakthrough of 3D graphics chip on the PC was the Voodoo Graphics Chip from 3dfx Interactive, a company founded by former SGI employees in 1994. Similar to the Matrox Millennium the 3dfx was a PCI add-on card which didn't come with a RAMDAC by itself but instead daisy-chained to a second standard 2D graphics card. The Voodoo heralded a new era in 3D graphics on the PC and marked the start of the end of expensive graphics workstation. 3dfx introduced their own API called Glide which had a different strategy to all other APIs of that time (Direct3D, OpenGL, QuickDraw3D, and Intel 3DR) it did not hide low-level hardware details behind an "abstraction layer" as all the others did but instead implemented nothing more than what the chip was capable of. At that time it gave Glide a significant performance advantage since the overhead of an abstraction layer was quite costly in terms of memory and CPU processing.

In 1998 NVidia introduced the RIVA (Real-time Interactive Video and Animation accelerator) TNT (TwinTexel), NVidia's 4th graphics chip at that time. The TwinTexel as its name states introduced a two single texturing pipeline and also offered tri-linear texture filtering. Unlike the Voodoo graphics chip and its successor it was fully OpenGL 1.1 compatible. It had a 24bit ZBuffer and full 32bit colour framebuffer.

A year after that NVidia released the Geforce256 which NVidia marketed "the world's first GPU" and the term Graphics Processing Unit was born. It was the first single-chip processor with integrated transform, lighting, triangle setup/clipping and texturing and rendering engines. A professional version named Quadro signalled NVidia's entry into the professional 3D workstation market. And with the Geforce256 it had reached a performance level that rivalled that of high-end professional 3D workstations from SGI that were using the infinite reality engine. The Geforce256 was fully OpenGL1.2 compatible as well as Direct3D7 which introduced hardware texturing and lighting to the API.

The next step in the evolution of GPUs was the Geforce3 released in 2001. It was the first chip to introduce custom programmable features into the otherwise

common fixed graphics pipeline. The concept of vertex and pixel shaders was born. Direct3D8 was the first graphics API that had mandatory customisable parts in an otherwise fixed pipeline. Even though the feature set of pixel and vertex shaders at that time was very limited it still introduced the basis of a common paradigm until today. Both vertex and pixel inputs can be seen as a single stream where an equal number of vertices or pixels are coming into the processing unit as they are coming out. Vertices and pixels can not be duplicated, replicated or added during processing but they can be destroyed under certain circumstances and costs. Each of these vertices and pixels can in theory be processed independently from each other but they all go through the same executing programming code, similar to the behaviour of a traditional stream processor. Pixel and vertex programmes in Direct3D8 were written in sort of pseudo-assembly language that had specific vector (4D), scalar and texture load operations. But there were neither jumps, labels, loops or branches in the instruction set which made it a really easy, non interruptible execution model allowing maximum throughput.

The ATI Radeon 8500 added further extensions to these two new programmable units that were introduced in Direct3D8.1. This added Pixel shader Version 1.1,1.2,1.3 and 1.4 and Vertex shader Version 1.1 to the API which basically changed the assembly language slightly and added new instructions, allowed longer programmes and more texture operations.

Another landmark in terms of programmability of GPUs marked the release of the Direct3D9 and later Direct3D9C API. This extended the pixel and vertex programs even further to Pixel shader version 2.0, 2+, 3.0 and Vertex shader version 2.0, 2+ and 3.0. Additionally, to the assembly nature of previous vertex and pixel programmes it added a high level C-Style/Renderman like language called HLSL which allowed to write vertex and pixel programs in a more abstract, readable and reusable fashion. To achieve this it added the concept of code compilation to the world of GPUs similar to the CPU decades ago but unfortunately without the maturity seen so far on CPUs. During that time OpenGL with the decline and exit of SGI as driving force more or less played catch-up with the significant and radical changes in the industry and it struggled for quite a long time with vendor-specific extensions and solutions until finally with OpenGL1.5 and then later OpenGL2.0 a common high level shading language for vertex and pixel programs had been introduced called GLSL.

In 2006 Microsoft again led the way for more radical changes in the world of graphics API with the release of Direct3D10 API only available through their new operating systems Windows Vista. It tried to force to combine the zoo of different shader version and capabilities of GPUs into one single entity by forcing the vendor to go for a unified shading core. This means that both vertex and pixel processing can be handled by one single programmable unit instead of having separate programmable units for both the vertex and pixel pipeline as it used to be the case in Direct3D9. This has the advantage that depending on wherever the highest workload is required in your graphical application more of the total processing power can be automatically dedicated too, therefore theoretically removing any processing bottleneck. Of course this unification is

only on paper it leaves it to the vendors to implement it adequately. Additionally to the unified shading model Direct3D10 added another programmable stage called the geometry shader (see Diagram) to the pipeline and also a new virtual memory model.

Meanwhile the two major hardware vendors on the market NVidia and AMD/ATI introduced their own APIs (CUDA from NVidia and CAL/CTM and now StreamSDK from AMD/ATI) to breakout of the black box and abstraction layer model of GPU programmability that graphics API provide to give the programmer a finer grained access to their underlying internal hardware functionality. By hoping to tap into the growing GPGPU (general purpose GPU) / HPC market, these allow the programmer to take more advantage of the underlying compute power that modern GPUs provide while paying with the price of reduced portability and code complexity.

## Chapter 3

# Current available hardware architectures

As of today two major commercial solutions control the market of 3D graphics specialized hardware: NVIDIA and AMD/ATI. ATI is behind the Radeon series while Nvidia is responsible for the GeForce, Quadro and Tesla line. This chapter tries to cast some light on the state of the art of current available graphics hardware architecture perfectly aware, due to the rapid development of new solutions, how rapidly the proposed material can become obsolete.

### 3.1 Nvidia GT200 series



Figure 3.1: Nvidia GT295 series (image courtesy of NVIDIA group).

The GeForce GTX 200 series represents the tenth generation of NVIDIA's graphics cards. Precursor to this series are the GeForce 8 and 9 series which share the same shader model architecture also known as Unified Shader Model

(or Shader Model 4.0 under DirectX 10). Latest unveiled model of the GTX200 family are GTX285 and GTX295 released in January 2009. The 295 (see Figure 3.1) model introduces a dual GPU architecture in terms of a pair of updated GT200 (referred to as GTX200b) with 240 processors each boosting the total number of shader cores to 480. Each GTX200b chip features also 80 texture units, 30 render back ends (RBEs or ROPs) and 448-bit memory interface. It supports multi-display desktop mode, therefore enabling stereoscopic 3D visualization, increased multi-thread architecture and less power consumption. The 285 model is still a single GPU model represents a step up from the GTX 280. As the GTX295 model it relies on a GTX200b architecture which increases the performance by at least 15% with respect to its 280 ancestor. Designed to support CUDA technology, the GTX200 architecture is claimed to be OpenCL compliant as well, with respect to future OpenCL releases.

### 3.2 ATI Radeon R700 series



Figure 3.2: ATI Radeon R700 series (image courtesy of AMD/ATI group).

The ATI Radeon R700 series (see Figure 3.2) represents the graphics processing unit series released by AMD/ATI Graphics Product Group. Based on the RV770 chip its architecture extends the R600's unified shader architecture by increasing the stream processing unit from 320 to 800 units grouped into 10 SIMD cores. The RV770 chip features also 40 texture units and 16 render back ends (RBEs or ROPs) and 256-bit memory interface. Designed for dual GPUs architectures and being the first chip designed to fully support GDDR5 memory the RV770 chip has been employed in the manufacturing of the Radeon HD 4870 released in August 2008. With a stream power of 640 shader processors the Radeon HD 4870 X2 features 2 GB GDDR5 memory, while the Radeon HD 4850 X2 still being a dual GPU architecture relies on a cut down version of the RV770 GPU referred to as RV770 LE adapted for a 256-bit GDDR3 memory interface.

## Chapter 4

# Programming models & APIs

With the advent of programmable graphics hardware, several solutions have been proposed to ease the process of accessing programmable units like fragment and vertex shaders unit bypassing the assembly language constraint. Vendor-dependent and independent APIs are available each featuring a plethora of capabilities ranging from the cross-platform OpenGL Shading Language to the sophistication of DirectX. The new era is also seeing GPUs becoming more and more sophisticated to the extent of assuming a unique identity distinct from the purpose they were originally designed for. General-purpose computing on graphics processing units (GPGPU) is pushing what once was a dedicated graphics rendering device to perform tasks traditionally handled by CPUs. CUDA, StreamSDK, RapidMind and OpenCL represent today the latest programming interface for parallel computing hardware like GPGPUs. Still far from determining the “standard”, each solution brings its own perspective into the market, opening up general-purpose computing from servers to workstations to handheld devices, to the world of GPU-accelerated software.

### 4.1 Graphics APIs

OpenGL and DirectX are representatives of established API for both graphics and graphics hardware programming. Diverse by design, OpenGL is an open standard API compatible with most modern operating systems, DirectX is a proprietary API designed by Microsoft Corporation and officially implemented only for Microsoft family operating systems. Both API provide a set of functions for traditional rendering of 2D and 3D graphics and graphics hardware acceleration. It is interesting to highlight the design choices behind the development of these two tools and how they respectively relate and respond to the rapid advance of graphics technology and software solutions.

### 4.1.1 OpenGL

OpenGL stands for OpenGraphics library and was first developed by Silicon Graphics Inc. (SGI) in 1992 as an alternative to Iris GL the proprietary graphics API of Silicon Graphics workstations. Mark Seagal and Kurt Akeley authored the first library release named OpenGL 1.0. With respect to its Iris GL ancestor OpenGL featured a formal specification which represented the first attempt at establishing a formal definition of a graphics API. Conceived to be cross-platform the OpenGL project provided third party implementation and support from its start plus a complete suite of conformance tests to allow third party library extension development. In 1992 the OpenGL Architecture Review Board (ARB) was established as an independent consortium to oversee the future of OpenGL. Main duties of the OpenGL ARB are the certification of extensions, additions and major modification to the library core. As of today OpenGL has moved to become part of the Khronos Group [3].

The evolution of the OpenGL library has seen the incremental addition of new features to the core API. Extensions to the library have been gradually incorporated and as of this date OpenGL 2.1 appears to be the latest release with significant additions like the OpenGL Shading Language (GLSL) and the OpenGL for Embedded Systems (OpenGL ES).

#### 4.1.1.1 GLSL

The OpenGL Shading Language (GLSL) [18, 16] is a high level shading language largely based on the ANSI C programming language. Being part of the OpenGL library GLSL-developed shaders are not standalone applications but require an application that utilizes the OpenGL API. A shader is a computer program fragment that calculates rendering effects on graphics hardware. GLSL enables the programmer to implement shaders for the programmable hardware contained in the OpenGL processing pipeline. At runtime GLSL programs have automatic access to part of the OpenGL state which can be affected by a shader. This allows the application to use existing OpenGL state commands for state management and have the current values of such state (like colour or light intensity) automatically available for use in a shader. GLSL supports the implementation of both vertex and fragment shaders.

#### 4.1.1.2 OpenGL | ES

OpenGL for Embedded Systems (OpenGL ES) is a low level API for 2D and 3D graphics acceleration on embedded and mobile devices. Currently two releases are available: OpenGL ES 1.X defined relative to OpenGL 1.5 and for fixed function hardware only, OpenGL ES 2.X defined relative to OpenGL 2.0 and for programmable hardware. OpenGL ES can be seen as a subset of the corresponding OpenGL since it does not support the entire OpenGL functionalities. The complexity of the processing pipeline is highly simplified in both releases. OpenGL ES 2.0 however presents the most significant differences: fixed function transformation, lighting and fragment rendering pipelines are completely

removed and replaced by programmable ones (i.e. shaders), commands can no longer be accumulated in display lists for later processing and the first stage of the processing pipeline which evaluates geometric approximation for curves and surfaces is eliminated. As a result OpenGL ES 2.X is not backward compatible with OpenGL ES 1.X. For a more detailed review of the main differences between OpenGL ES and OpenGL 2.X refer to [17].

### 4.1.2 Direct3D

Direct3D is Microsoft’s platform specific API for programming 3D graphics on their operating systems. It is part of a larger collection of different APIs called the DirectX SDK. The DirectX SDK includes – besides Direct3D – APIs for sound programming, input devices and several other items of functionality. Historically, DirectX is aimed at the video games entertainment industry and therefore its focus is still heavily geared towards that sector even today. Unlike OpenGL, Direct3D underwent many radical changes since its first introduction in 1996 and very often subsequent versions are not backwards compatible to older versions of the API. Using this strategy it allowed Microsoft to evolve its API much quicker than for example OpenGL to changes in the graphics card market. That is why in recent years Direct3D seems to have led the way when it comes down to introducing new features in GPU programmability and their exposure to software development without the need to struggle with hardware vendor specific solutions but providing more or less a hardware platform independent minimal feature set application programmers can rely on.

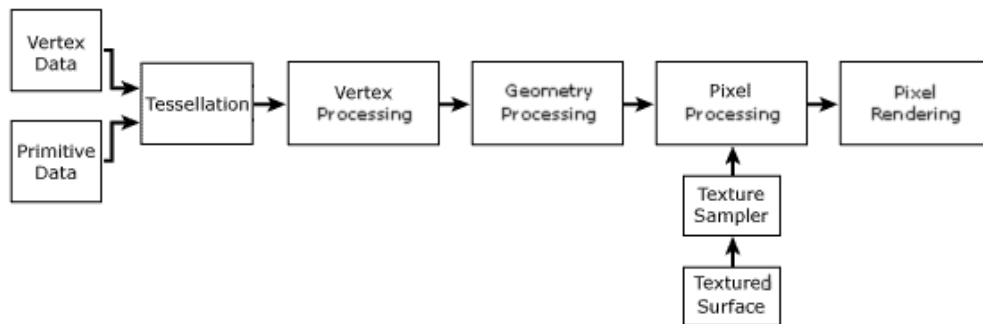


Figure 4.1: Direct3D 9 Graphics Pipeline (image courtesy of msd.microsoft.com).

The current latest version of Direct3D is Direct3D10 [13] with a preview of Direct3D11 currently available and a likely full release of Direct3D11 some time this year. The numbering would assume this is the tenth major version of Direct3D which is not actually true since there never had been a version 4.

The classic graphics processing pipeline of Direct3D is nearly identical to OpenGL (see figure A.1) and its integration into the operating system can be

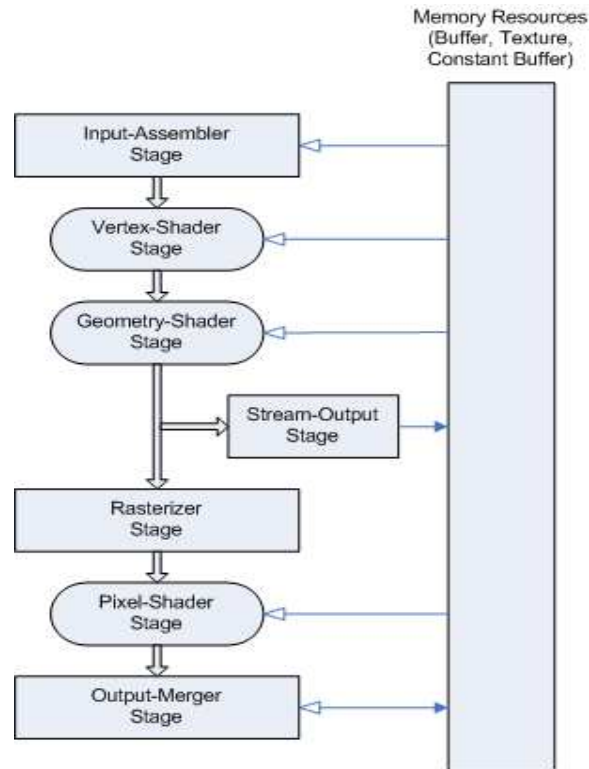


Figure 4.2: Direct3D 10 Graphics Pipeline (image courtesy of msd.microsoft.com).

seen in Figure 4.4. However, with Direct3D10 the pipeline has been changed and the fixed pipeline as seen in Figure 4.1 no longer exists, instead it looks now as shown in Figure 4.2.

Also compared to OpenGL the programming model of Direct3D differs quite extensively. Remember OpenGL is a pure C procedural command style API that controls the underlying graphics processor in a state machine model. Often OpenGL has many incarnations and variations of commands for doing the same thing for example creating, sending and drawing geometry on the graphics processing unit.

Direct3D doesn't follow this model. The common state machine model of graphics processing units is abstracted in Direct3D into ATL (abstract type library) pointer objects – a very specific Microsoft Windows API construct that can be found in many other Microsoft APIs. The central object in the API is the *Device*. It is responsible for everything connected to the hardware. All operations that talk to the graphics processing unit are either linked to the *Device* or are directly executed by the *Device*.

Up until Version 11 Direct3D has supported two main types of *Devices*:

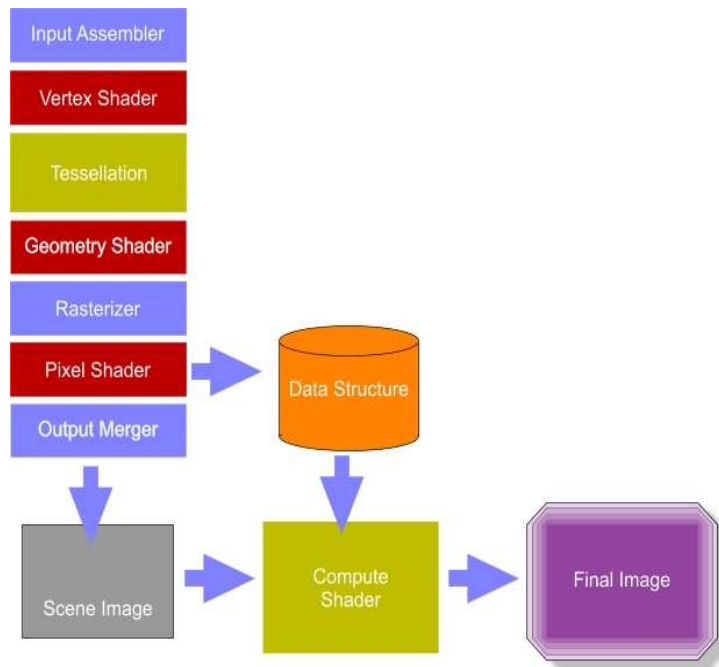


Figure 4.3: Direct3D System Integration plus Pipeline

- HAL Device. The primary device type is the Hal device, which supports hardware accelerated rasterization and both hardware and software vertex processing. If the computer on which your application is running is equipped with a display adapter that supports Direct3D, your application should use it for Direct3D operations. Direct3D Hal devices implement all or part of the transformation, lighting, and rasterizing modules in hardware.
- Reference Device. Direct3D supports an additional device type called a reference device or reference rasterizer. Unlike a software device, the reference rasterizer supports every Direct3D feature. This device is intended to be used for debugging purposes and is therefore only available on machines where the DirectX SDK has been installed. Because these features are implemented for accuracy rather than speed and are implemented in software, the results are not very fast. The reference rasterizer does make use of special CPU instructions whenever it can, but it is not intended for release type of applications.

The Reference Device in Direct3D can be used in two potential scenarios. The more significant one is probably when strange results are seen in the HAL Device leading to potential bugs in the hardware device driver or the hardware itself. These can easily be verified and identified by switching to the Reference Device.

The second scenario is when the underlying hardware doesn't support (yet) a feature that is required. An application programmer still is able to program the code paths that uses the feature and actually see the behaviour even though it will run very slow.

With the current version of the DirectX SDK (November 2008) a new beta *Device* has been added to the list. It is called the *WARP Device*. Similar to the Reference Device it is a pure software rasterizer. But unlike the Reference Device it is highly optimised and takes advantage of latest CPU technology increasing the speed compared to the Reference Device by a couple of magnitudes. Additionally it reaches similar rendering quality levels as modern GPUs in Direct3D10.

Creating a *Device* in Direct3D is at the beginning probably the hardest thing to understand and it needs an experienced programmer to fully grasp what is happening behind the scenes. The reason for this is that Direct3D allows to attach the *Device* to different graphics adapters and different SDK versions. A graphic adapter represents a single graphics processing unit installed in the system. For starters the easiest thing to do is to just use one of the many example codes that come with the DirectX SDK; they mostly contain most of the more complicated API calls for creating and housekeeping a *Device*.

Geometry and textures (1D,2D,3D) are created through the *Device* and are accessed and stored in different *Resource* ATL object types in Direct3D. All these objects have different methods to manipulate their content before they are sent to the graphics processing unit. During the creation of *Resources* different creation flags that determine the lifetime, CPU & GPU read/write access, and binding to a specific pipeline stage can be set. This provides the API and device driver hints to optimise the memory location of a *Resources*. Direct3D10 abstracts and generalizes the access to *Resources* by allowing different *Views* on a *Resource*. This means that *Resource* itself is just a block of memory somewhere in the system either directly on the graphics card memory or in main memory and the *View* describes the layout and structure of the *Resource*. This concept allows to have different interpretations of a *Resource* at the same time.

#### 4.1.2.1 HLSL

Since Direct3D9C Vertex, Pixel, – and since Direct3D10 Geometry – shaders can be written in a high-level C-style language called HLSL (High Level Shading Language). The shaders can either be compiled and linked at runtime using specific API calls or offline using a standalone compiler *fxc.exe* and then loaded onto the card. The output of the compiler is a hardware independent intermediate assembly language that is then linked into a hardware independent intermediate binary code which is then reinterpreted inside the hardware device driver to native code. Direct3D9 still allows to write shaders directly in the intermediate assembly language for maximizing performance or overcoming specific shader version restriction. Direct3D10 specific shader versions no longer allow this.

On top of HLSL Microsoft introduced the optional FX (effect system) often called HLSL FX. The effect system allows to group and combine different Geom-

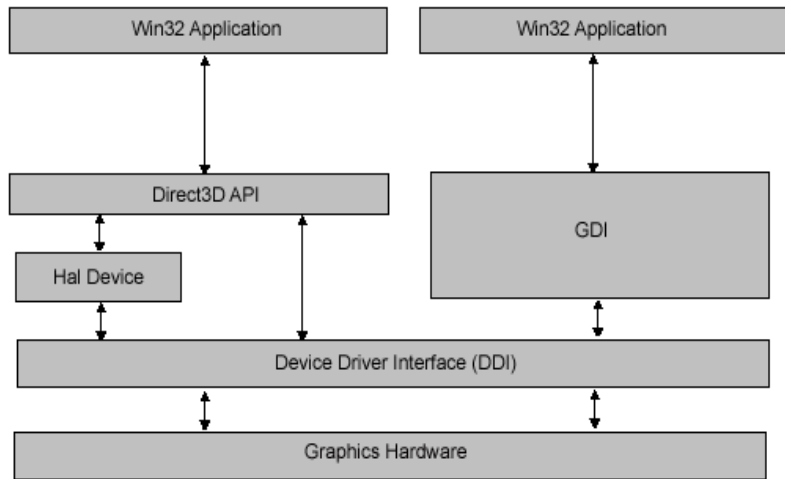


Figure 4.4: Direct3D System Integration (image courtesy of msdn.microsoft.com).

entry, Vertex and Pixel shaders and other non-programmable pipeline stages into so called *Techniques*. This allows easy access to constants, global variables, and texture samplers used in different shaders. Additionally all non-programmable pipeline stages, like alpha blending, culling mode, stencil buffer, alpha, stencil, depth test, etc. can be manipulated outside the application code and without changing the application code itself. This allows people without any knowledge of core Direct3D programming to change and manipulate key functionality as long as the application is taking advantage of the effect system.

Alongside the Direct3D API, the DirectX SDK provides additional useful features encapsulated in the D3DX (Direct3D extension) library. This extension library includes a complete set of vector and matrix math operations including quaternions, plus routines for loading, creating and manipulating meshes and different image file format loaders for creating texture resources.

## 4.2 Non graphics API

The evolution of graphics hardware as general purpose computing devices has speeded up the development of non-graphics API devoted at programming graphics processor to execute non-graphics computing tasks. Both vendor-specific and cross-platform solutions have been recently made available to the public each fighting the battle of creating the ultimate standard for GPGPU development from both an architectural and programming level.

## 4.2.1 Vendor specific APIs

The two major commercial competing standards for GPGPU development are represented by the NVIDIA “Compute Unified Device Architecture” (CUDA) [2] and ATI’s StreamSDK [1]. The CUDA developing platform has so far been adopted by the HPC market more extensively than the StreamSDK one, however both still equally compete on the vendors’ side.

### 4.2.1.1 CUDA

CUDA [2] represents the NVIDIA answer to GPGPU computing. Born as Computer Unified Device Architecture the CUDA acronym stands now on its own, replacing entirely the original extended name; CUDA is available on all latest NVIDIA graphics cards and starting from the GeForce 8 series, Quadro FX 5600/4600 and Tesla solutions. Its software architecture comprises several layers in terms of API, hardware driver, and utility libraries (see Figure 4.5). Through CUDA the GPU is seen as a massively parallel set of multiprocessors (see Figure 4.6) capable of executing a high number of threads in parallel. From a programming architecture point of view the GPU is treated as a co-processor (or *computing device*) to the main CPU (or *computing host*). The CPU downloads to the devices those functions which exhibit parallel behaviour. Before being downloaded each of these functions is translated into the device instruction set, the resulting device-program is referred to as *kernel*. A kernel can be composed of several threads organized as a grid of thread blocks. Threads are assigned to blocks according to their data access pattern to allow for efficient sharing of data and memory access.

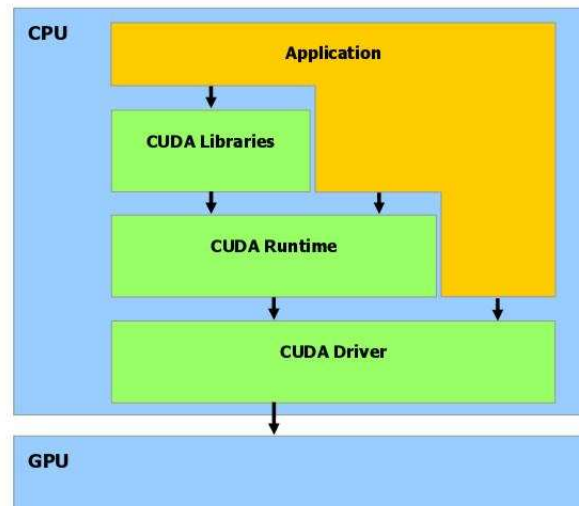


Figure 4.5: Cuda Software Stack (image courtesy of NVIDIA group).

The API programming model is based on an extension of the C programming language according to four major development lines: function type qualifiers to characterize a function as belonging to the host or to the device, variable type qualifiers to specify the memory location of a variable, directives to describe the execution of a kernel on a device, built-in variables related to grid and block dimensions and blocks and threads indexing system.

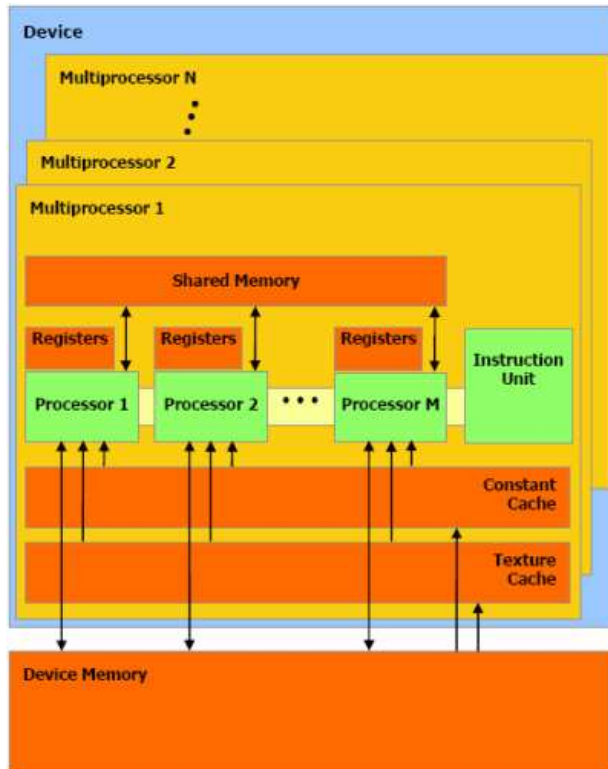


Figure 4.6: GPU as a set of SIMD multiprocessors (image courtesy of NVIDIA group).

#### 4.2.1.2 StreamSDK

The AMD Stream Computing Model 4.7 is AMD's response to GPU programming for high-performance data parallel tasks. The model includes a software stack (StreamSDK) and the AMD stream processor born from the acquisition of ATI. The Stream SDK consists of a suite of tools for GPU programming which leverages the offload of arithmetic operations onto the GPU. This capability is achieved through a hierarchy of three main levels which integrates with C and C++ development products. The hierarchy comprises: performance libraries such as the AMD Core Math Library (ACML) and COBRA for opti-

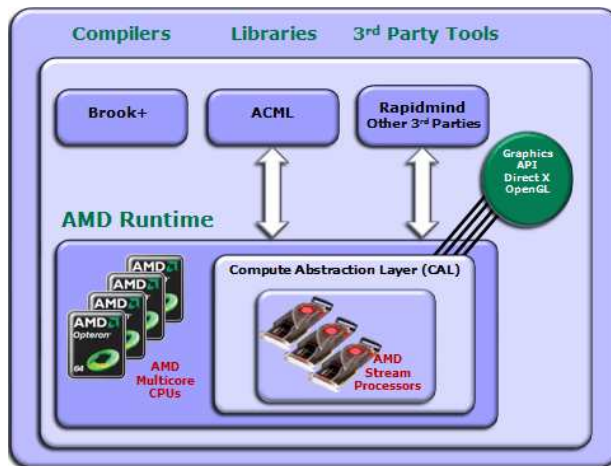


Figure 4.7: AMD Stream Computing Model (image courtesy of AMD/ATI group).

mized domain-specific algorithms, compilers such as Brook+ and RapidMind, lower level drivers and programming languages such as the AMD Compute Abstraction Layers (CAL) and the Intermediate Language (IL), performance profiling tools such as GPU ShaderAnalyzer and AMD CodeAnalyst. Brook+ is a high-level language, extension of Brook for GPU programming, an abstract processor model used for simplifying computations on graphics processors. AMD SDK includes a Brook+ compiler which converts Brooks+ files into C for execution on the CPU and GPU. CAL is a device-driver library that provides a programmer-friendly interface to AMD’s stream processors devices. IL or intermediate language is a high-level assembly language for the GPU designed to allow developers to access directly the graphics hardware’s lower levels.

Most of AMD Stream SDK [1] is a result of AMD’s acquisition of ATI. The initiative created a unique stream of knowledge which saw expertise in semiconductor technologies combined with expertise in high-performance development tools. This latest generation of processors support the unified shader programming model. Programmable stream cores execute user-developed programs called *stream kernels*. Streams and Kernels are the basic building blocks for code development in Brook+. Streams can be defined as collection of data which are read from or to (CPU or GPU). Kernels are clusters of operations which define functions on elements in the streams. Stream cores can execute both graphics and non-graphics functions through a virtualized SIMD programming model operating on the stream of data.

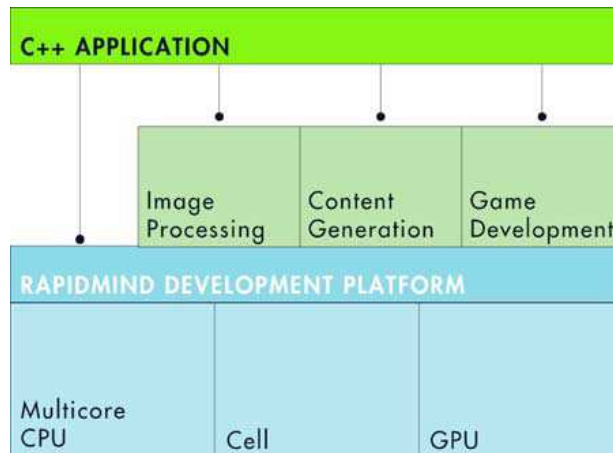


Figure 4.8: RapidMind Application Structure (image courtesy of RapidMind ltd.).

## 4.2.2 Vendor-independent APIs

Cross platform APIs to face the GPGPU programming issues have been proposed by both RapidMind [12] and the Khronos Group [15]. Compared with their commercial antagonists these vendor-independent APIs focus is much more oriented towards a unified standard for parallel programming adaptable not only to GPU programming but to a wider range of multiprocessor architectures like multi-core CPUs and Cell Broadband Engines.

### 4.2.2.1 RapidMind

RapidMind [12] is a private enterprise devoted at the development of platform independent solutions for high-performance processors including multi-core CPUs and accelerators such as GPUs. RapidMind flagship is the RapidMind Multicore Development Platform . In the case of GPU the RapidMind platform can be used for both shaders and general purpose processing. It provides a software development platform supporting as programming language ISO-standard C++ with no requirement for any GPU-specific extension. Developers use their own existing C++ compilers and build systems, but are required to use specific platform defined types for numbers, vectors, matrices and arrays. This allows the RapidMind embedded system to identify and record parts of an application which can be accelerated at runtime. Parallel processing is automatically managed by the RapidMind platform at runtime, the optimized code is mapped onto all the available computational resources in a given system or on the specifically targeted hardware (see Figure 4.8). RapidMind recently embraced Open Source and Standards Projects policies taking part in the LLVM (Low Level Virtual Machine) Compiler Infrastructure project and OpenCL standard as one of the development partners and user (see Section 4.2.2.2).

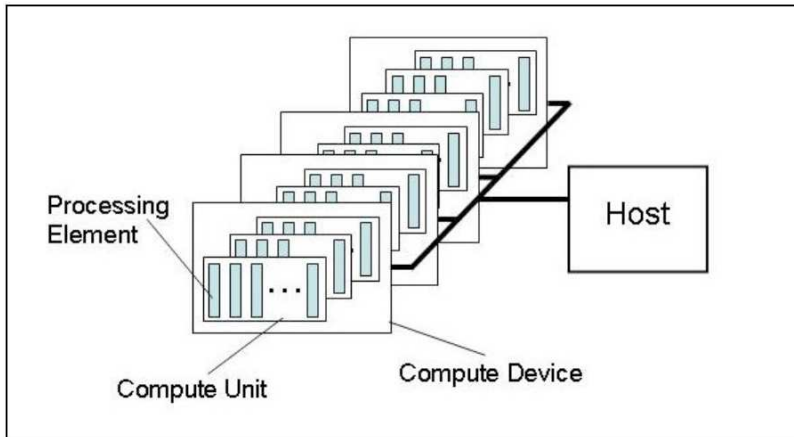


Figure 4.9: OpenCL Platform Model (image courtesy of KHRONOS group).

#### 4.2.2.2 OpenCL

OpenCL (Open Computing Language) [15] represents the first endeavour in creating a open standard for parallel programming of heterogeneous computational resources at processor level. More than just a programming language it includes an API, libraries and runtime system for software development. The framework aspires at enabling portable and efficient access to general purpose parallel programming across CPUs, GPUs, Cell and ManyCores architectures for both HPC and commodity applications. The main key is to allow applications to use a host and one or more OpenCL devices as a single heterogeneous parallel computer system. Experienced programmers are supported throughout the process of developing general purpose algorithm without the necessity of mapping the algorithm onto architecture/platform specific features like 3D graphics API such as OpenGL or DirectX. Originally started by Apple which served as specification editor, led in the development by Khronos group the OpenCL working panel lists partners as: 3DLans, AMD/ATI, IBM, Intel, Motorola, Nokia, Nvidia, RapidMind, Texas Instruments. Being an open source royalty-free standard it is free to join the developers consortium through the Khronos group.

The core ideas behind OpenCL are described by four main models: **platform model**, **memory model**, **execution model** , **programming model**.

**platform model** The Platform model consist of a **host** connected to one or more OpenCL **compute devices**. A compute device is divided into one or more compute units (CUs) which are further divided into one or more processing elements (PEs). Computations on a device occur within the processing elements. Each processing element can either behave as a SIMD (single instruction multiple data) or as a SPMD (single program multiple data) unit. The core

difference between SIMD and SPMD relies in whether a kernel is executed concurrently on multiple processing elements each with its own data and a shared program counter or each with its own data but its program counter. In the SIMD case all processing elements execute a strictly identical set of instructions which cannot be always true for the SPMD case due to possible branching in a kernel. Each OpenCL application runs on a host according to the hosting platform models, and submits commands from the host to be executed on the processing elements within a device.

**execution model** The Execution model of an OpenCL program occurs in two parts: a **kernel**, basic unit of executable code which is executed on one or more OpenCL devices, and a **host program**, collection of compute kernels and internal functions, which is executed on the host. A host program defines the context for the kernels and manages their execution. When a kernel is submitted for execution by the host, an index space is defined. An instance of the kernel executes for each point in this index space. This kernel instance is called a work-item and is identified by its point in the index space, which provides a global ID for the work-item. Work-items are organized into work-groups. The work-groups provide a more coarse-grained decomposition of the index space. Work-groups are assigned a unique work-group ID with the same dimensionality as the index space used for the work-items. Work-items are assigned a unique local ID within a work-group so that a single work-item can be uniquely identified by its global ID or by a combination of its local ID and work-group ID.

**memory model** The Memory model of OpenCL is a shared memory model with relaxed consistency. each *work-item* has access to four distinct memory regions: **global memory** accessible by all work-items in all work-groups; **constant memory** a read only global space; **local memory** local to a *work-group*; **private memory** private to a work-item.

**programming model** OpenCL supports two main Programming models: the **data parallel programming model** and the **task parallel programming model**. Hybrids of the two models are allowed though the driving one remains the data-parallel. In a data-parallel programming model sequences of instructions are applied to multiple elements of memory objects. OpenCL maps data to work-items and work-items to work-groups. The data-parallel model is implemented in two possible ways. The first or **explicit** model lets the programmer define both the number of work-items to execute in parallel and how work-items are divided among work-groups. The second or **implicit** model lets the programmer specify the number of work-items but OpenCL to manage the division into work-groups.

### 4.2.3 Direct3D11 Compute shader

With Direct3D11 Microsoft will introduce another shader along side the already existing shaders in the common Direct3D10 pipeline to generalize the data-parallel programming model that exists in Direct3D. It will be fully integrated into Direct3D allowing to exchange data for example between Pixel shaders and Compute shaders (see Figure 4.3). Unlike Pixel Shaders it will have cross-thread data sharing, unordered access I/O operations like gather and scatter. It will enable far more general data structures – like irregular arrays, trees, linked list – compared to the previous *Resource* types in Direct3D10 which are basically representing linear arrays. With the availability of these datastructures it will allow more general algorithms that go far beyond the common shading. Compute shader will be focused on client scenarios; this means not for HPC computation cluster farms but enabling the Tera Flop performance that a graphics card provide on a single machine that wants to display/render the result of the computation with very low latency in realtime.

Computer shaders can simply spawn a regular array of threads. That array can be one, two or three dimensional. There are shared registers between threads to reduce the register pressures on the underlying graphics processing unit and also will eliminate redundancy computation and I/O operations. The initial version will have 32KByte of registers that can be shared. These registers will be 32bit only. Not all threads in a call will be able or should share registers with each other, otherwise the latency of accessing the registers would be too high. That is why sharing threads are broken down into subsets (groups) of threads. All the different thread indices, like *threadID*, *threadGroupID*, and *threadIDinGroup* are accessible in the compute shader. Compute shaders will share the same language subset as the other shaders and following the same evolutionary path. Meaning that they will be backwards and forwards compatible between different generations.

# Chapter 5

## Future Outlook

GPGPU, GPU Clusters, Multi-cores or multi-processors architecture, not only the future of graphics hardware but high performance general purpose computing is also at a turning point. With so many new possible designs and unexplored routes what will be the real hardware turnover is still a guess. If at the level of its expectation the Intel's Larrabee project [20] could drastically change the profile of the current market, if not then solutions like GPU clusters could lead the way.

### 5.1 Larrabee

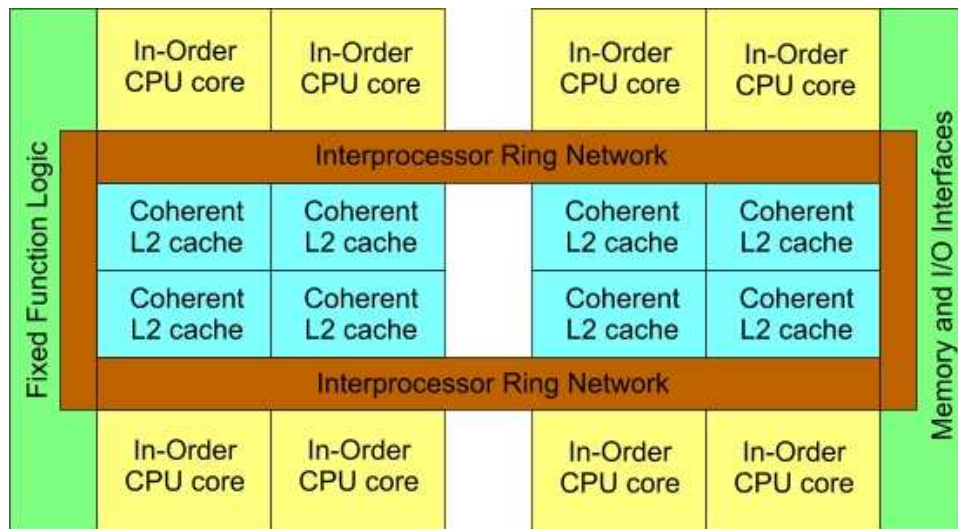


Figure 5.1: Schematic for the Larrabee many-core architecture

Larrabee is the codename for a new many-core visual computing architecture that Intel is developing in parallel with its current line of integrated graphics accelerators (Intel GMA). Larrabee's blueprint[20] was first released in August 2008 at the SIGGRAPH 2008 conference, the first chip release is expected to be in early 2010. Published specifics and claimed performances make Larrabee the potential direction to where GPUs are going in the future. From a design point of view Larrabee exhibits similarities with both multi-core CPUs and GPUs. Of a multi-core CPU architecture it inherits a coherent cache hierarchy and x86 architecture compatibility of the GPU architecture it inherits the wide SIMD vector units and texture sampling hardware. As described in [20] the main architecture (see Figure 5.1) is based on in-order CPU cores running an extended version of the x86 instruction set. The extensions include wide vector processing operations, specialized scalar instructions such as bit count and bit scan, new instruction and instruction modes for explicit cache control. Each CPU is augmented with a wide vector processing unit (VPU) (see Figure 5.2) which should allow for extremely efficient mathematical operations. Larrabee is equipped with coherent on-die 2<sup>nd</sup> level cache to allow efficient inter-processor communication and high-bandwidth local data to be accessed by CPU cores. Each core can support up to four execution threads with separate register sets per thread. Task scheduling is performed entirely in software rather than fixed function logic. This allows for pipelines to adjust their resource scheduling and load balancing algorithms. Larrabee programming model (or *Larrabee Native*) is C/C++ based, it supports both DirectX and OpenGL applications and thread programming through OpenMP. Unlike current GPUs architectures Larrabee will feature cache coherency across all its cores. From a rendering point of view the Larrabee graphics rendering pipeline includes very little specialized hardware. A *Larrabee Native* application being mostly software written it can be easily extended.

Larrabee innovative design and projected performances make it a potential breakthrough for both graphics and traditional HPC computing.

## 5.2 GPU Clusters

An alternative to multi-core/multi-processor architectures like the one proposed by Larrabee is clusters of GPUs. As for CPU clusters a GPU cluster consist of a computer cluster in which each node is equipped with a Graphics Processing Unit. Target of this kind of clusters is to harness the computational power of GPUs treating them as general purpose processors. From a hardware point of view GPU clusters fall into two categories: heterogeneous and homogeneous. Heterogeneous clusters feature hardware from different manufacturers (mostly NVIDIA and AMD/ATI) or same make but different model, while in homogeneous clusters all GPUs are identical to each other (same class, brand and model). An example of heterogeneous GPU cluster is the NCSA's Innovative Systems Laboratory 16-node cluster. The NCSA infrastructure combines both GPUs and FPGA (field-programmable gate array) technology to explore the application of

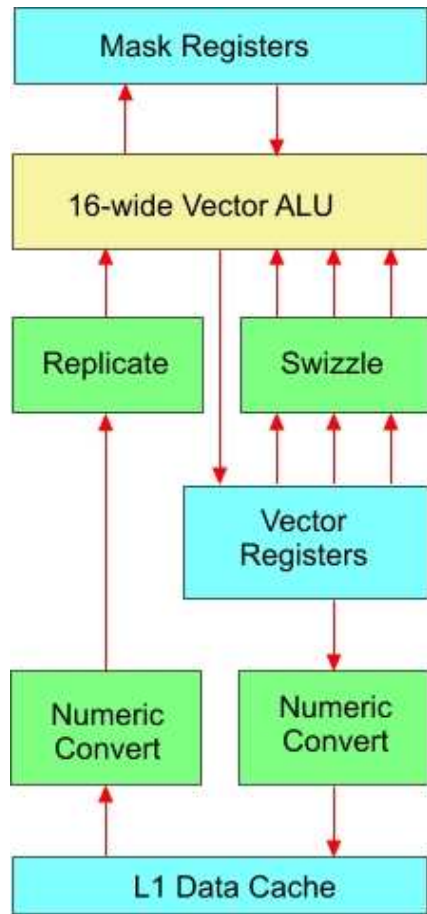


Figure 5.2: Vector Processing Unit block diagram.

these architectures to accelerate scientific computing tasks. According to [4] each of the 16 nodes in the cluster features at present: two dual-core 2.4 GHz AMD Opterons with 8 GB of memory, four NVIDIA Quadro 5600 GPUs with 1.5 GB of memory each, and a Nallatech H101-PCIX FPGA accelerator with 16 MB SRAM and 512 MB SDRAM. Another interesting attempt at building a GPGPU cluster was made in 2004 by the Center For Visual Computing and Department of Computer Science at Stony Brook University [14]. The Stony Brook cluster featured 32 nodes connected by a 1 Gigabit Ethernet switch, with each node being an HP PC equipped with two Pentium Xeon 2.4GHz processors and 2.5GB memory, and a GeForce FX 5800 Ultra with 128MB memory, used for the GPU cluster computation.

# Appendix A

## Getting Started

The purpose of this section is to give a headstart into GPU programming using the Python programming language. Not expecting of being comprehensive of all the features involved into GPU programming this document aims instead at introducing the novice programmer to the graphics acceleration world abstracting from low-level details peculiar to languages as GLSL or HLSL.

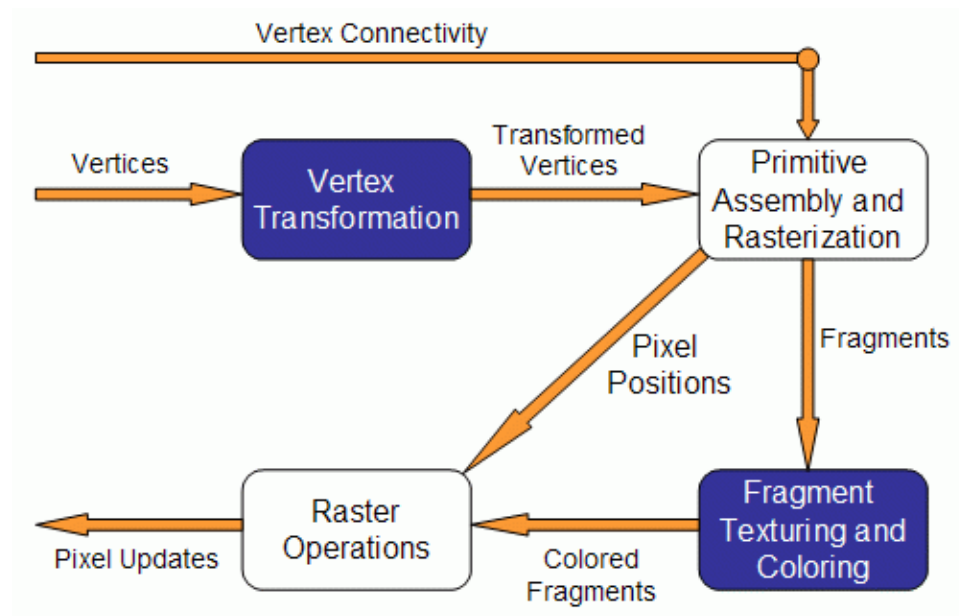


Figure A.1: Graphics Processing Pipeline <sup>1</sup>

<sup>1</sup>Courtesy of [www.lighthouse3d.com](http://www.lighthouse3d.com)

### A.0.1 The Rendering Pipeline

Rendering is the conversion of a scene into an image. The rendering pipeline or graphics pipeline in computer graphics is a conceptual representation of the stages through which a certain representation of a 2D or 3D scene goes through to produce as output the final 2D raster image. Each graphics packages has its own rendering pipeline in terms of 2D or 3D primitives representation and sequence of processes applied to them. Graphics rendering pipeline are commonly represented as state-diagram, where each state refers to the sequence of transformations applied to the input data. Fig. A.1 represents an abstraction of a rendering pipeline diagram. Input data to a graphics pipeline are generally vertices together with their respective set of attributes such as position in space, colour, normals and texture coordinates. Each transformation step within the pipeline vertices and their abstract attributes are mathematically processed to generate physical screen coordinates and display values. In old graphics card each stage of the pipeline was implemented as fixed function, i.e. the algorithms (shaders) used to perform the transformations were decided at the card manufacturing stage and could not be altered afterwards. The novelty introduced by graphics accelerators was to give the programmer access to two of the graphics pipeline stages: vertex transformation stage and fragment transformation stage. The rendering pipeline has then become programmable, piece of code used to apply transformation effects on the input data would be called *shaders*.

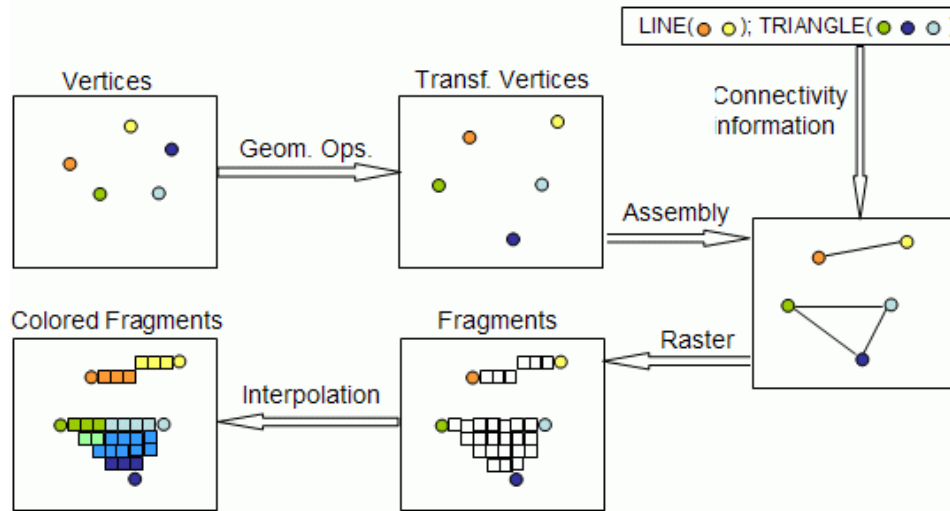


Figure A.2: Vertices and Fragments transformation stages <sup>1</sup>

## A.0.2 Vertex Shader

The vertex processor is a programmable unit that operates on incoming vertices and their associated data A.2. Vertex shaders are set of functions used to transform each abstract vertex position in world space to the 2D coordinate system at which it appears on the screen. Vertex shaders run on the vertex processor.

## A.0.3 Fragment Shader

The fragment processor is a programmable unit that operates on incoming fragments and their associated data A.2. Fragments shaders are set of functions used to transform each abstract vertex attribute such as colour, normal, texture value to the display colour system. Fragment shaders are typically used to mimic effects such as scene lighting and colour toning on a per pixel base, for this reason the are often referred to as *pixel shaders*. Fragment shaders run on the fragment processor.

## A.0.4 Programming the GPU with Python

The way to program the GPU is via vertex and fragment shaders creation. Several API exists to ease the process such as GLSL, HLSL, Cg and many more. In this section we will experiment with PyGPU [19] a Python based API for GPU programming. PyGPU runs as initially born as a domain specific language for image processing, it now features a compiler that generates code executable on the GPU. PyGPU runs as an embedded language inside Python and as such it inherits the functionality and syntax of the host language. It interfaces with the graphics card through a set of extension packages: NumPy [5] an extension to the Python language for scientific computing, Pygame [8] a Python API for game development, PyGlew [9] a Python binding for the OpenGL Extension Wrangler (GLEW) and PyCg [7] a Python binding for the NVIDIA Cg language [6] which makes PyGPU bounded to NVIDIA graphics card. To start experimenting download the aforementioned packages and install them on your PC, refer to [http://www.cs.lth.se/home/Calle\\_Lejdfors/pygpu/](http://www.cs.lth.se/home/Calle_Lejdfors/pygpu/) for instruction on how to download and install PyGPU and its complements.

### A.0.4.1 PyGPU Shader Example

The *compiler* plays a major role in PyGPU, it allows Python code to be compiled into native code executable on the graphics hardware. The example we show is taken from [19]. Edge detections is a problem of fundamental importance in image processing as edges characterize boundaries between objects within an image. Edges are parts of an image with high intensity contrast, i.e. the image brightness changes sharply. Edge detection methods can be grouped into two main categories: gradient based and Laplacian. The Sobel edge detection algorithm belongs to the first group. The Sobel operator performs a 2-D spatial gradient measurement of the image intensity at each point (or pixel). The



Figure A.3: Sobel PyGPU Edge Detector applied to the Lena image.

operator uses a pair of  $3 \times 3$  convolution masks (see Figure A.4), or *kernels*, to estimate gradients in the x-direction and y-direction respectively. Convolution masks are usually much smaller than the actual image therefore they slid over the image manipulating squares of pixel at a time.

-1	0	+1
-2	0	+2
-1	0	+1

**Gx**

+1	+2	+1
0	0	0
-1	-2	-1

**Gy**

Figure A.4: Sobel Convolution Kernels

The following function definition implements a gradient based edge detector algorithm in Python:

```
def edgeDetect(kernel, im=Image, p=Position):
    Gx = convolve(kernel, im, p)
    Gy = convolve(transpose(kernel), im, p)
    return sqrt(Gx**2 + Gy**2)
```

`edgeDetect` applies a pair of convolution kernels in the x and y direction and returns the magnitude of the image gradient. The edge detection algorithm is a multi-step process therefore can be easily implemented on the GPU as a sequence of filtering operations to take advantage of the graphics accelerator capabilities. We can then specialize the edge detection function and use the PyGPU compiler to translate it into native GPU code:

```
sobelEdgeDetGPU = pygpu.compile(edgeDetect, kernel=sobelKernel)
```

The function returned by the compiler is what would be called a *shader*, it runs entirely on the GPU. An example of the `sobelEdgeDetGPU` applied to the famous Lena image is shown in Figure A.3. For more insights into how the PyGPU compiler is implemented refer to [10] and [19]

Resources for GPU programming with Python are rather limited an interesting project worth to look at is PYGWA [11] an extended GPGPU library for Python an extended GPGPU library for Python.

## A.1 Acknowledgments

We thank Dr. Joachim Diepstraten for all the helpful suggestions and material provided. This report was produced with funding from VizNET, the UK visualization support network.

# Bibliography

- [1] Ati stream computing, jan 2009. <http://ati.amd.com/technology/streamcomputing/resources.html>.
- [2] Cuda zone, jan 2009. [http://www.nvidia.com/object/cuda\\_home.html](http://www.nvidia.com/object/cuda_home.html).
- [3] Khronos group - opengl, jan 2009. [www.khronos.org/opengl/](http://www.khronos.org/opengl/).
- [4] Ncsa's innovative systems laboratory - gpu cluster project, jan 2009. <http://www.ncsa.edu/Projects/GPUcluster/>.
- [5] Numpy, jan 2009. [numpy.scipy.org](http://numpy.scipy.org).
- [6] The nvidia cg toolkit, jan 2009. [http://developer.nvidia.com/page/cg\\_main.html](http://developer.nvidia.com/page/cg_main.html).
- [7] Pycg, jan 2009. [www.launchpad.net/pycg](http://www.launchpad.net/pycg).
- [8] Pygame, jan 2009. [www.pygame.org](http://www.pygame.org).
- [9] Pyglew, jan 2009. [www.launchpad.net/pyglew](http://www.launchpad.net/pyglew).
- [10] Pygpu, jan 2009. <http://code.google.com/p/pygpu/>.
- [11] Pygwa, jan 2009. <http://pygwa.sourceforge.net/>.
- [12] Rapidmind, jan 2009. [www.rapidmind.net](http://www.rapidmind.net).
- [13] David Blythe. The direct3d 10 system. *ACM Trans. Graph.*, 25(3):724–734, 2006.
- [14] Zhe Fan, Feng Qiu, Arie Kaufman, and Suzanne Yoakum-Stover. Gpu cluster for high performance computing. In *SC '04: Proceedings of the 2004 ACM/IEEE conference on Supercomputing*, page 47, Washington, DC, USA, 2004. IEEE Computer Society.
- [15] Khronos OpenCL Working Group. *The OpenCL Specification*, Dec. 2008. Version 1.0.
- [16] Khronos OpenGL Working Group. *The OpenGL Shading Language (GLSL) Quick Reference Guide*. Version 1.10.
- [17] Khronos OpenGL Working Group. *OpenGL ES 2.0.23 Difference Specification*, Sept. 2006. [www.khronos.org/opengles/2\\_X/](http://www.khronos.org/opengles/2_X/).

- [18] Khronos OpenGL Working Group. *The OpenGL Shading Language*, Sept. 2006. Version 1.20.
- [19] C. Lejdfors and L. Ohlsson. Implementing an embedded gpu language by combining translation and generation. In *SAC '06: Proceedings of the 2006 ACM symposium on Applied computing*, pages 1610–1614, New York, NY, USA, 2006. ACM.
- [20] Larry Seiler, Doug Carmean, Eric Sprangle, Tom Forsyth, Michael Abrash, Pradeep Dubey, Stephen Junkins, Adam Lake, Jeremy Sugerman, Robert Cavin, Roger Espasa, Ed Grochowski, Toni Juan, and Pat Hanrahan. Larrabee: a many-core x86 architecture for visual computing. *ACM Trans. Graph.*, 27(3):1–15, 2008.