C-DAC Four Days Technology Workshop

ON

Hybrid Computing – Coprocessors/Accelerators
Power-Aware Computing – Performance of Applications Kernels

hyPACK-2013
(Mode-2 : GPUs)

Classroom lecture :

Basics of GPU-Based Programming
GPU Architecture

Venue : CMSD, UoHYD ; Date : October 15-18, 2013

Overview

- What is GPU ? Graphics Pipeline
- GPU Architecture
- GPU Programming – OpenGL, DirectX, NVIDIA (CUDA), AMD (Brook+)
- Rendering pipeline on current GPUs
- Low-level languages
  - Vertex programming
  - Fragment programming
- High-level shading languages
- GPU Architecture - Graphics Programming

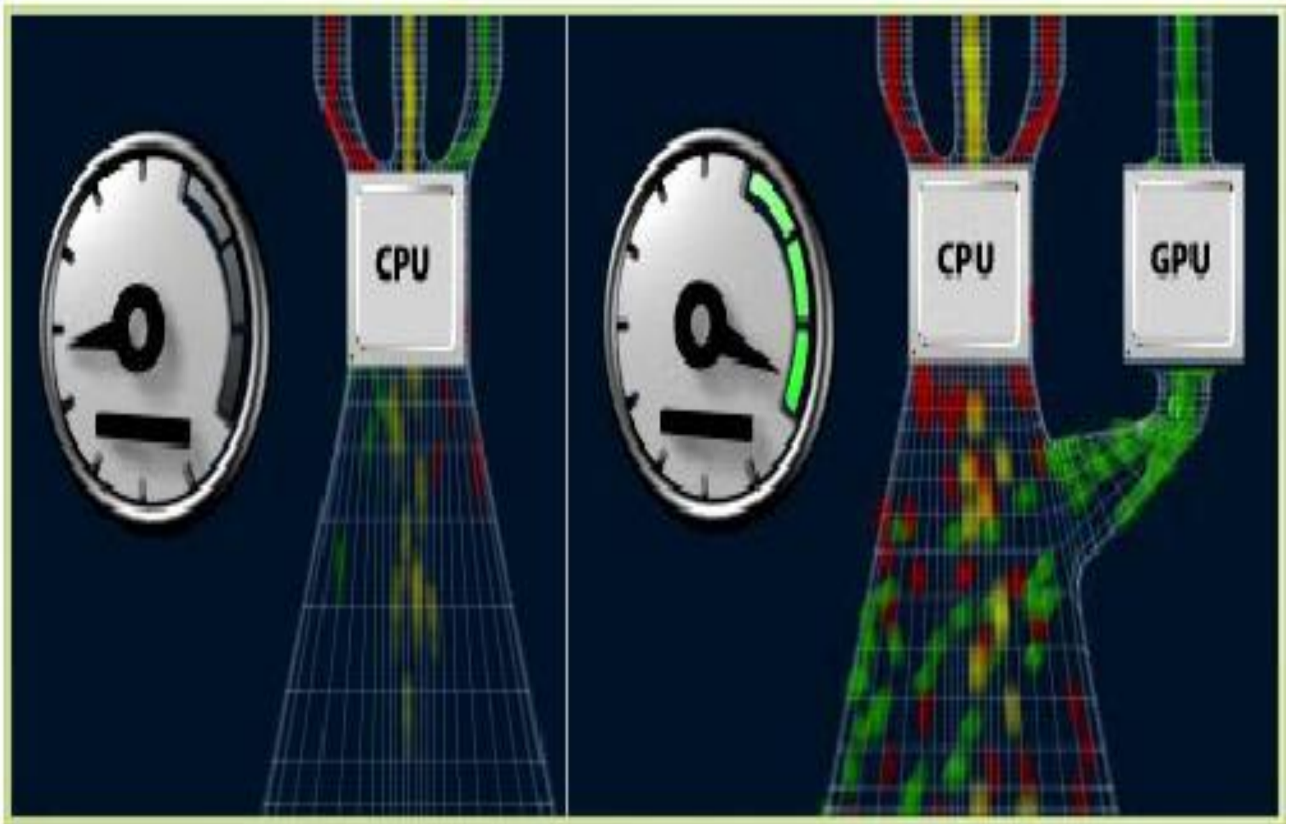
Source : References

What is GPU ?

- From Wikipedia : A specialized processor efficient at manipulating and displaying computer graphics
- 2D primitive support – bit block transfers
- Some might have video support
- And of course 3D support (a topic at the heart of this presentation)
- GPUs are optimized for raster graphics

Source : References

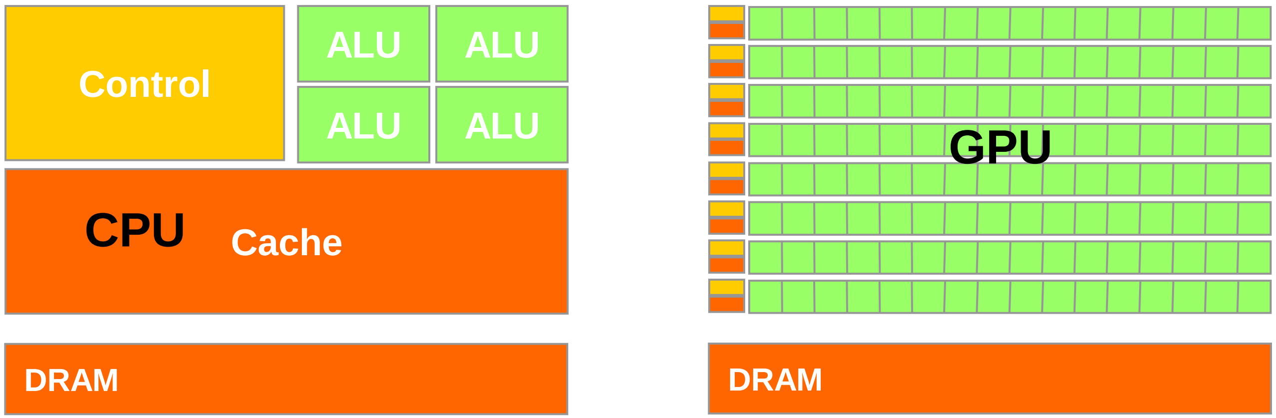
What is GPU ?



Without GPU

With GPU

What is GPU ?



- The GPU is specialized for compute-intensive, highly data parallel computation (exactly what graphics rendering is about)
  - ✓ So, more transistors can be devoted to data processing rather than data caching and flow control
- Data-parallel portions of an application are executed on the device as kernels which run in parallel on many threads
- GPU threads are extremely lightweight
- GPU needs 1000s of threads for full efficiency

What is GPU ?

❖ Graphics Processing Unit

❖ GPU also occasionally called visual processing unit or VPU

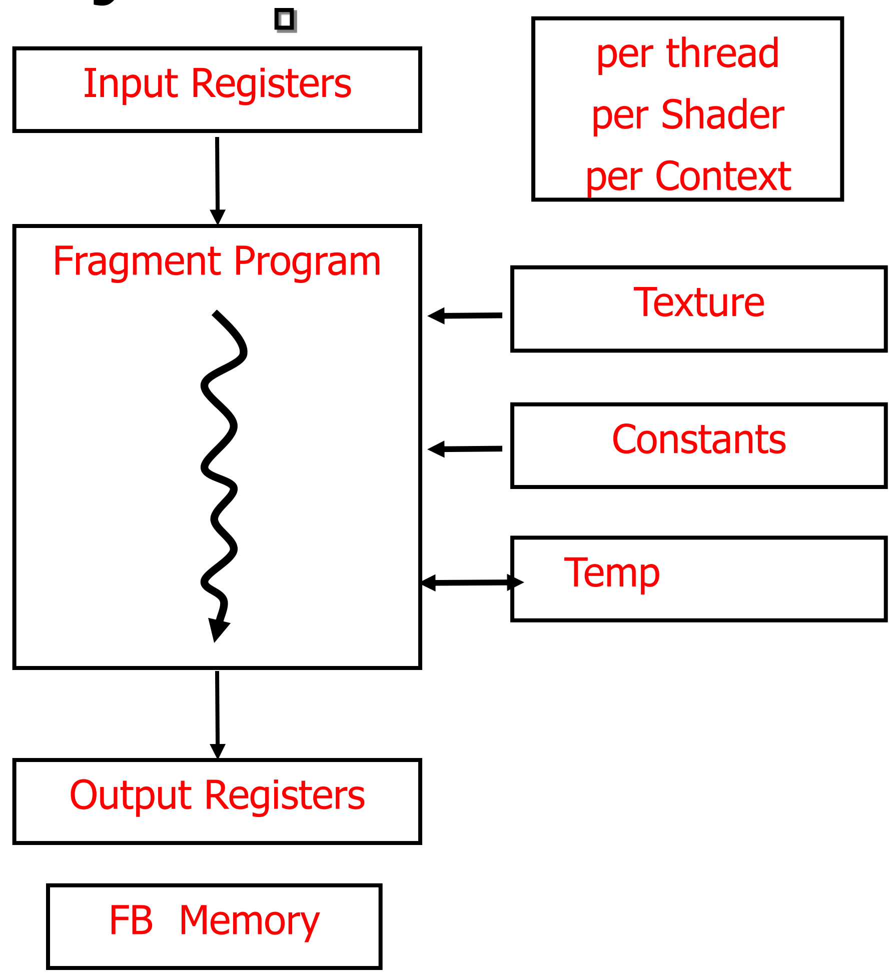
❖ It's a dedicated graphics rendering device for a personal computer, workstation, or game console.

❖ GPU is viewed as compute device that :
- Is a coprocessor to CPU or host machine
- Has its own DRAM (on the device)
- Runs many threads in parallel

❖ Thus GPU is dedicated super-threaded, massively data parallel co-processor

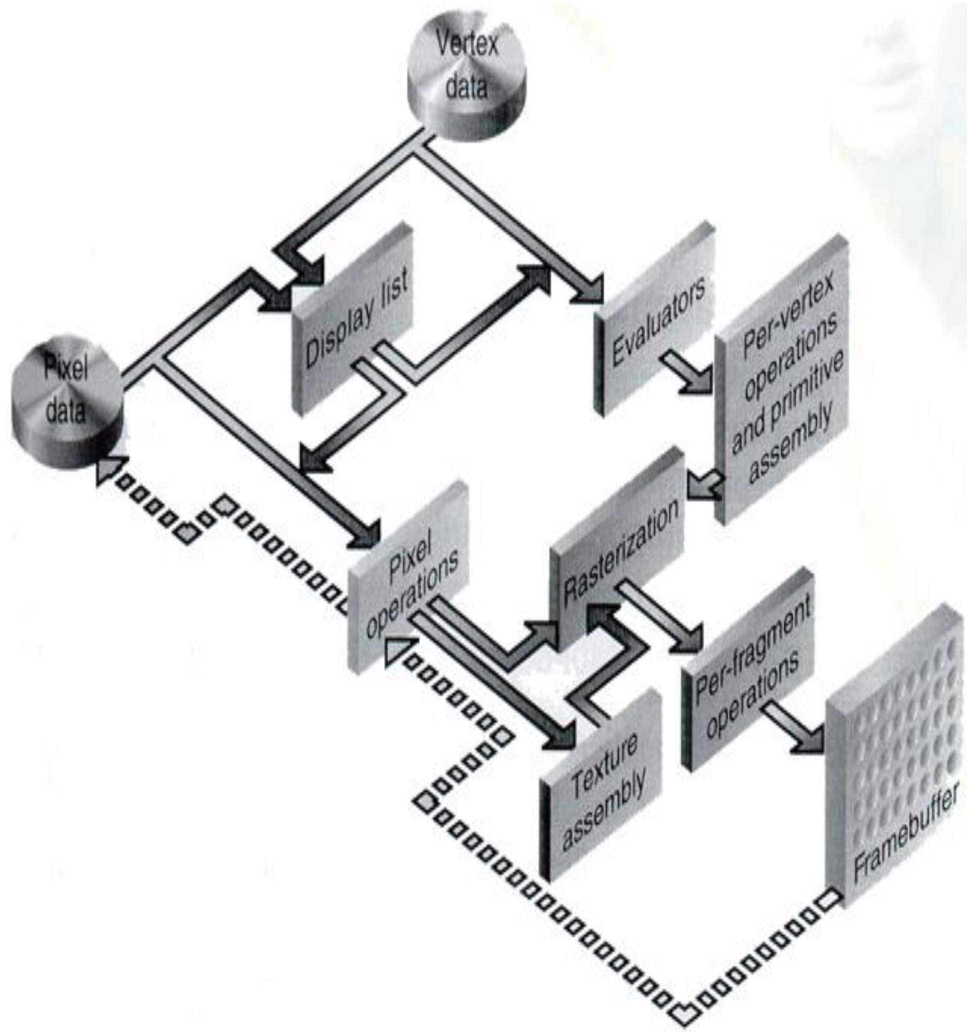
History

- Dealing complex with Graphics API
- Sequential Flow of Execution
- Limited Communication



The Graphics pipeline

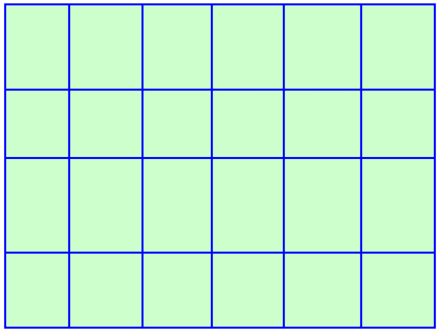
Application → Command → Geometry → Rasterization → Texture → Fragment → Display



3D Graphics Software Interfaces OpenGL (v2.0 as of now)

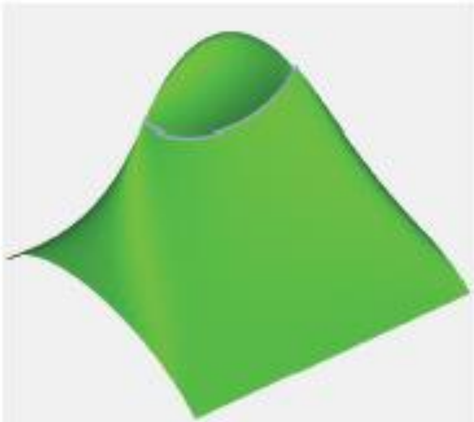
- Low level
- Specification not an API
- Crossplatform implementations
- Popular with some games
- A simple seq of opengl instr (in C)

```
glClearColor(0.0,0.0,0.0,0.0);
glClear(GL_COLOR_BUFFER_BIT);
glColor3f(1.0,1.0,1.0);
glOrtho(0.0,1.0,0.0,1.0,-1.0,1.0);
        glBegin(GL_POLYGON);
        glVertex(0.25,0.25,0.0);
        glVertex(0.75,0.25,0.0);
        glVertex(0.75,0.75,0.0);
        glVertex(0.25,0.75,0.0);
glEnd();
```



Source : References

Geometry Processing



Self intersections

Dynamic silhouette refinement

Algebraic Geometry

Preparation of FEM grids

Source : References

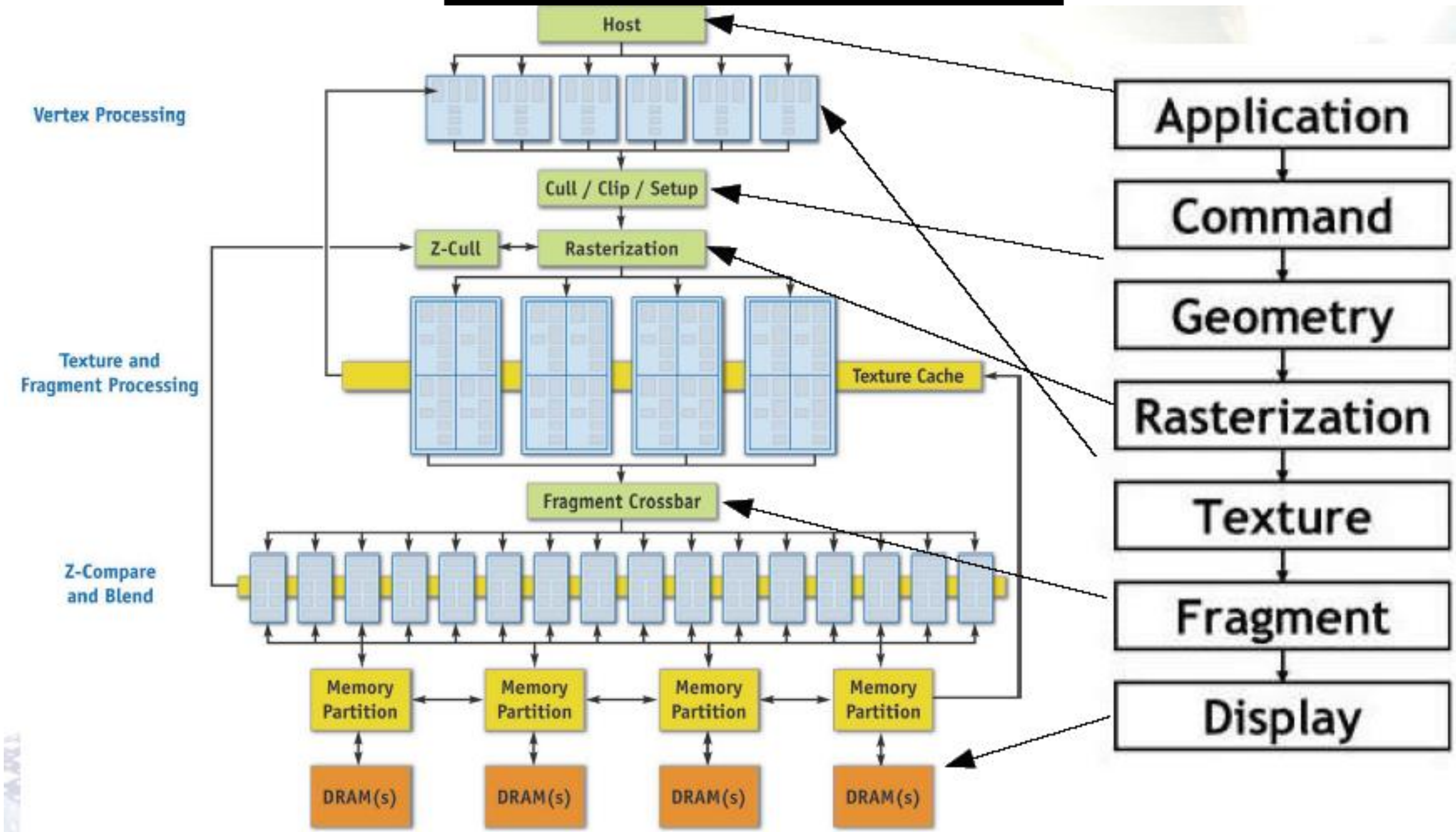
NVIDIA GeForce 6800 General Info

- Impressive performance stats
  - 600 Million vertices/s
  - 6.4 billion texels/s
  - 12.8 billion pixels/s rendering z/stencil only
  - 64 pixels per clock cycle early z-cull (reject rate)
- Riva series (1st DirectX compatible)
  - Riva 128, Riva TNT, Riva TNT2
- GeForce Series
  - GeForce 256, GeForce 3 (DirectX 8), GeForce FX, GeForce 6 series

Source : References

GeForce 8800 GT Card Specification's

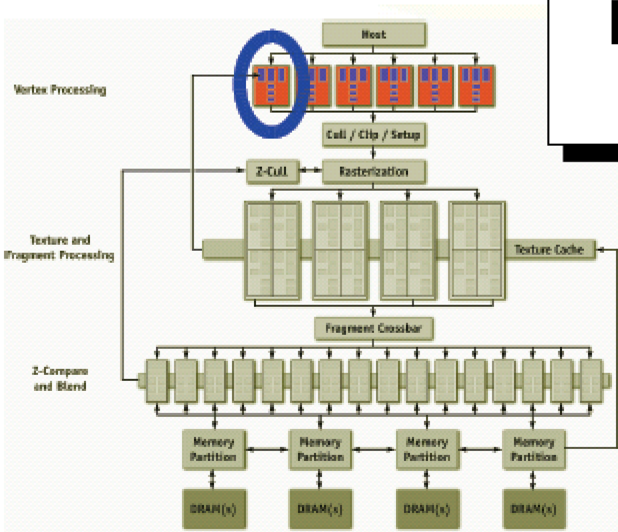
NVIDIA GeForce 6800 Block Diagram

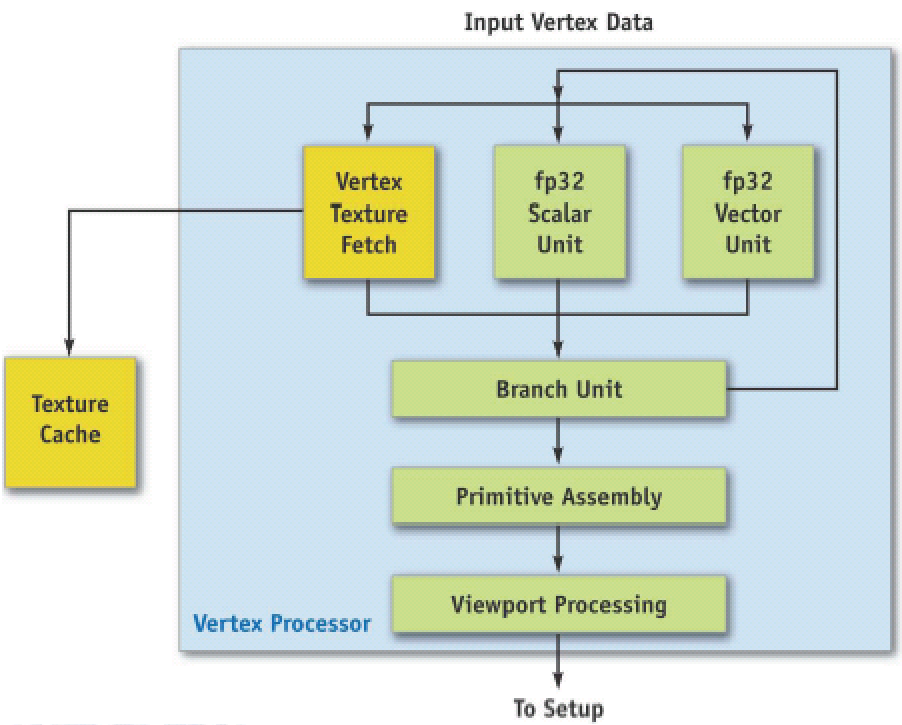


Source : References

NVIDIA GeForce 6800

Vertex Processor (or vertex shader)





- ❖ Allow shader to be applied to each vertex
- ❖ Transformation and other per vertex ops
- ❖ Allow vertex shader to fetch texture data (6 series only)

Source : References

GPU from comp arch perspective Processing units

- Focus on Floating point math
- fp32 and fp16 precision support for intermediate calculations

- 6 four-wide fp32 vector MADs/clock in shaders and 1 scalar multifunction op

- 16 four-wide fp32 vector MADs/clock in frag-proc plus 16 four-wide fp32 MULs

- Dedicated fp16 normalization hardware

Source : References

GPU from comp arch perspective Memory

- Use dedicated but standard memory architectures (eg DRAM)
- Multiple small independent memory partitions for improved latency
- Memory used to store buffers and optionally textures
- In low-end system (Intel 855GM) system memory is shared as the Graphics memory

GPU from comp arch perspective Memory

- GPU interfaces with the CPU using fast buses like AGP and PCI Express

- Port speeds
  - PCI express upto 8GB/sec (4 + 4)
  - Practically upto (3.2 + 3.2)
  - AGP upto 2 GB/sec (for 8x AGP)

- Such bus speeds are important because textures and vertex data needs to come from CPU to GPU (after that it's the internal GPU bandwidth that matters)

GPU from comp arch perspective Memory

- Texture caches (2 level)
  - Shared between vertex procs and fragment procs
  - Cache processed/filtered textures
- Vertex caches
  - cache processed and unprocessed vertexes
  - improve computation and fetch performance
- Z and buffer cache and write queues

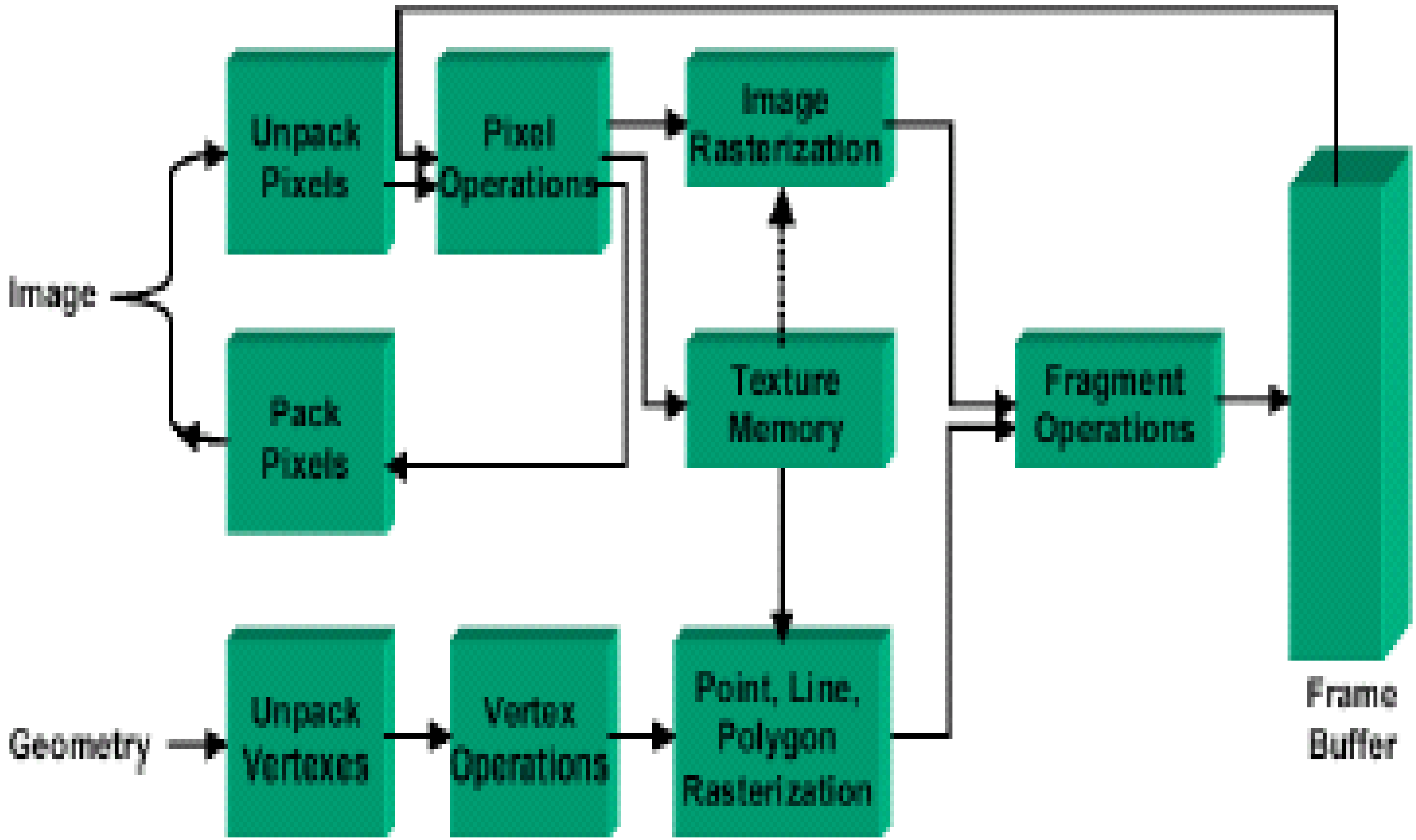
GPGPU

- ❖ Look at GPU as a fast SIMD processor
- ❖ It is a specialized processor, so not all programs can be run
- ❖ Example computational programs – FFT,
- ❖ Cryptography, Ray Tracing, Segmentation and even sound processing!

3D Graphics Software Interfaces Direct 3D (v9.0 as of now)

- High level
- 3D API – part of DirectX
- Very popular in the gaming industry
- Microsoft platforms only

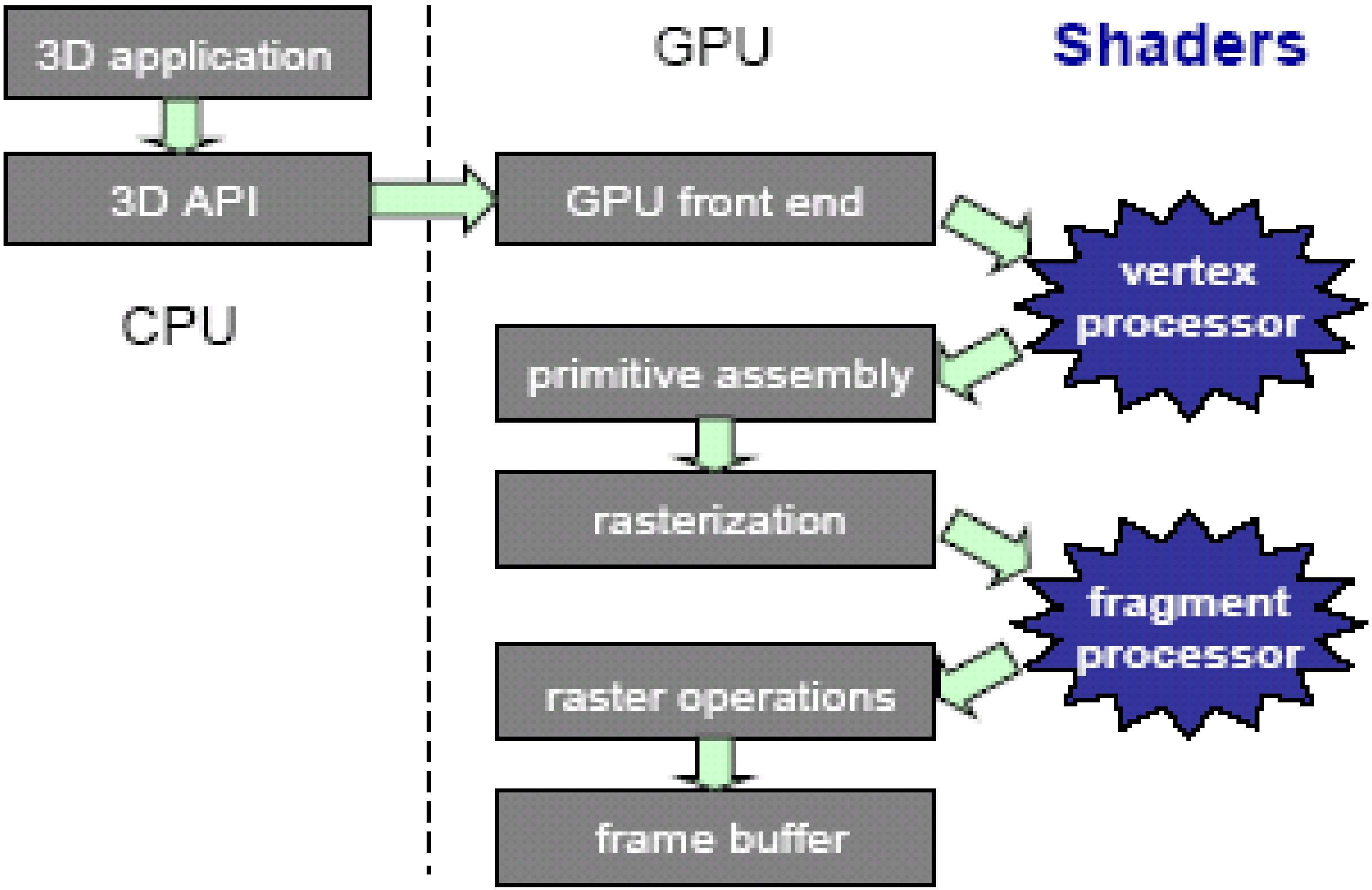
Traditional OpenGL Pipeline



Programmable Pipeline

- Most parts of the rendering pipeline can be programmed
- Shading programs to change hardware behavior
  - Transform and lighting:
    vertex shaders / vertex programs
  - Fragment processing:
    pixel shaders / fragment programs
- History: from fixed-function pipeline to configurable pipeline
  - Steps towards programmability

Programmable Pipeline



GPU - Issues

- How are vertex and pixel shaders specified?
  - Low-level, assembler-like
  - High-level language

- Data flow between components
  - Per-vertex data (for vertex shader)
  - Per-fragment data (for pixel shader)
  - Uniform (constant) data: e.g. modelview matrix, material parameters

GPU Overview

- Rendering pipeline on current GPUs
- Low-level languages
  - Vertex programming
  - Fragment programming
- High-level shading languages

What Are Low-Level APIs?

- Similarity to assembler
  - Close to hardware functionality
  - Input: vertex/fragment attributes
  - Output: new vertex/fragment attributes
  - Sequence of instructions on registers
  - Very limited control flow (if any)
  - Platform-dependent
    BUT: there is convergence

What Are Low-Level APIs?

- Current low-level APIs:
  - OpenGL extensions: GL_ARB_vertex_program,
  - GL_ARB_fragment_program

- DirectX 9: Vertex Shader 2.0, Pixel Shader 2.0
  - Older low-level APIs:
  - DirectX 8.x: Vertex Shader 1.x, Pixel Shader 1.x
  - OpenGL extensions: GL_ATI_fragment_shader, GL_NV_vertex_program, …

Why Use Low-Level APIs?

- Low-level APIs offer best performance & functionality
- Help to understand the graphics hardware (ATI's r300, NVIDIA's nv30, ...)
- Help to understand high-level APIs (Cg, HLSL, ...)
- Much easier than directly specifying configurable graphics pipeline (e.g. register combiners)

GPU - Overview

- Rendering pipeline on current GPUs
- Low-level languages
  - Vertex programming
  - Fragment programming
- High-level shading languages

Applications Vertex Programming

- Customized computation of vertex attributes

- Computation of anything that can be interpolated linearly between vertices

- Limitations:
  - Vertices can neither be generated nor destroyed
  - No information about topology or ordering of vertices is available

OPEN_GL GL_ARB_vertex_program

- Circumvents the traditional vertex pipeline

- What is replaced by a vertex program?
  - Vertex transformations
  - Vertex weighting/blending
  - Normal transformations
  - Color material
  - Per-vertex lighting
  - Texture coordinate generation
  - Texture matrix transformations
  - Per-vertex point size computations
  - Per-vertex fog coordinate computations
  - Client-defined clip planes

OPEN_GL GL_ARB_vertex_program

- What is not replaced?
  - Clipping to the view frustum
  - Perspective divide (division by w)
  - Viewport transformation
  - Depth range transformation
  - Front and back color selection
  - Clamping colors
  - Primitive assembly and per-fragment operations
  - Evaluators

DirectX 9: Vertex Shader 2.0

- Vertex Shader 2.0 introduced in DirectX 9.0
- Similar functionality and limitations as GL_ARB_vertex_program
- Similar registers and syntax
- Additional functionality: static flow control
  - Control of flow determined by constants (not by per-vertex attributes)
  - Conditional blocks, repetition, subroutines

Applications for Fragment Programming

- Customized computation of fragment attributes

- Computation of anything that should be computed per pixel

- Limitations:
  - Fragments cannot be generated
  - Position of fragments cannot be changed
  - No information about geometric primitive is available

OPEN_GL_ARB_fragment_program

- Circumvents the traditional fragment pipeline

- What is replaced by a pixel program?
  - Texturing
  - Color sum
  - Fog

  for the rasterization of points, lines, polygons, pixel rectangles, and bitmaps

- What is not replaced?
  - Fragment tests (alpha, stencil, and depth tests)
  - Blending

GPU Overview

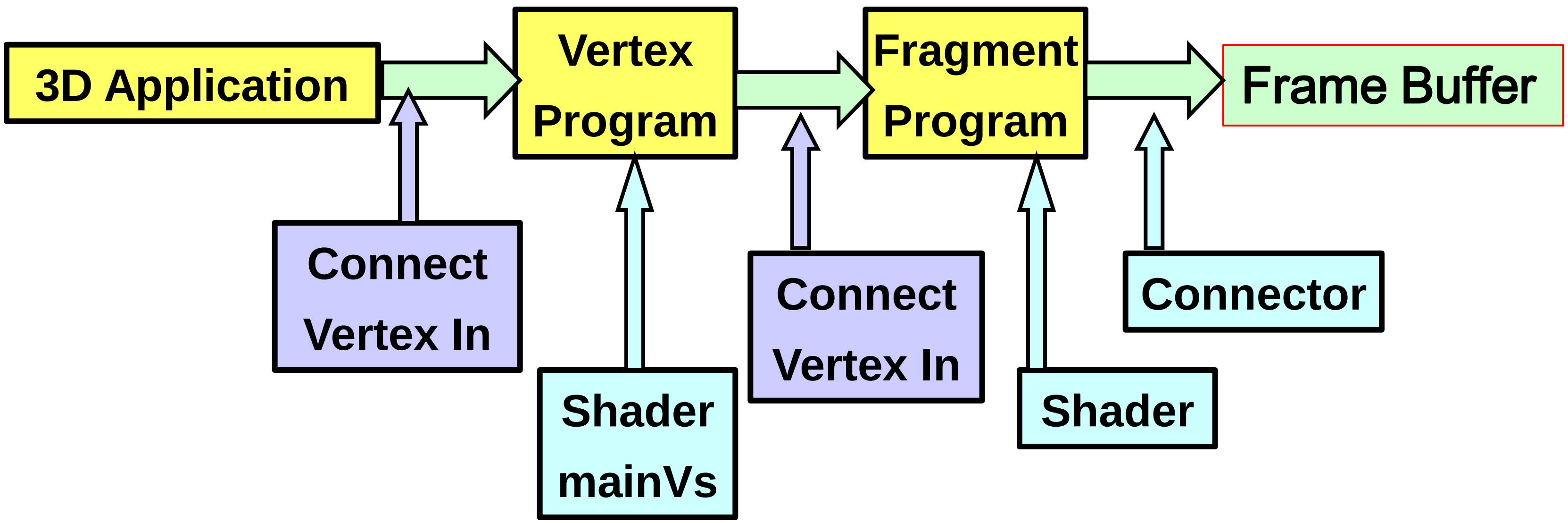
- Rendering pipeline on current GPUs
- Low-level languages
  - Vertex programming
  - Fragment programming
- High-level shading languages

High-Level Shading Languages

- Why?
  - Avoids programming, debugging, and maintenance of long assembly shaders
  - Easy to read
  - Easier to modify existing shaders
  - Automatic code optimization
  - Wide range of platforms
  - Shaders often inspired RenderMan shading language

Data Flow through Pipeline

- ❖ Vertex shader program
- ❖ Fragment shader program
- ❖ Connectors



High-Level Shading Languages

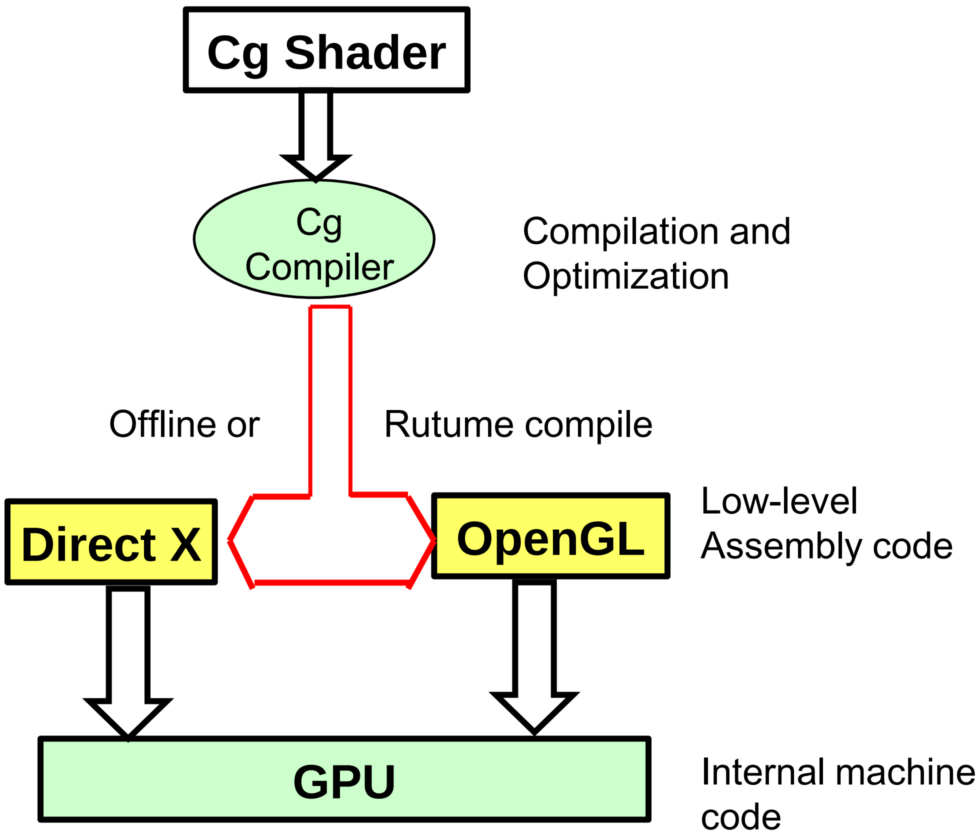
- Cg
  - “C for Graphics”
  - By NVIDIA
- HLSL
  - High-level shading language”
  - Part of DirectX 9 (Microsoft)
- OpenGL 2.0 Shading Language
  - Proposal by 3D Labs

GPU - Cg

- ❖ Typical concepts for a high-level shading language
- ❖ Language is (almost) identical to DirectX HLSL
- ❖ Syntax, operators, functions from C/C++
- ❖ Conditionals and flow control
- ❖ Backends according to hardware profiles

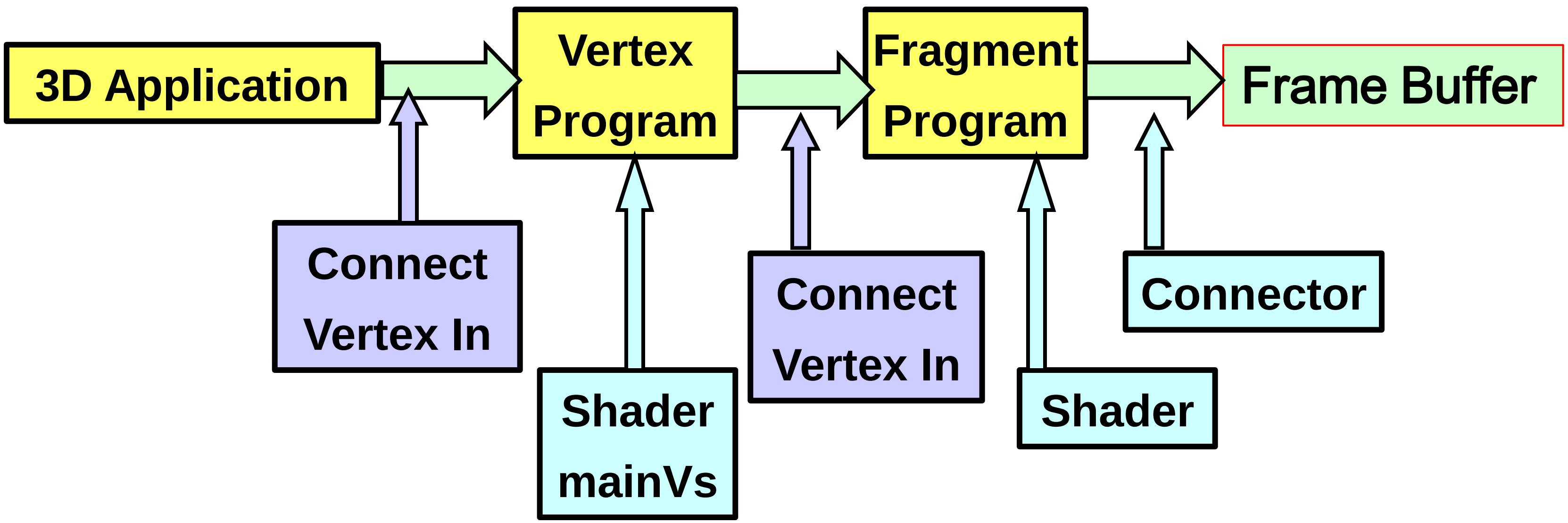
- ❖ Support for GPU-specific features (compare to low-level)
  - ➢ Vector and matrix operations
  - ➢ Hardware data types for maximum performance
  - ➢ Access to GPU functions: mul, sqrt, dot, …
  - ➢ Mathematical functions for graphics, e.g. reflect
  - ➢ Profiles for particular hardware feature sets

Workflow in Cg

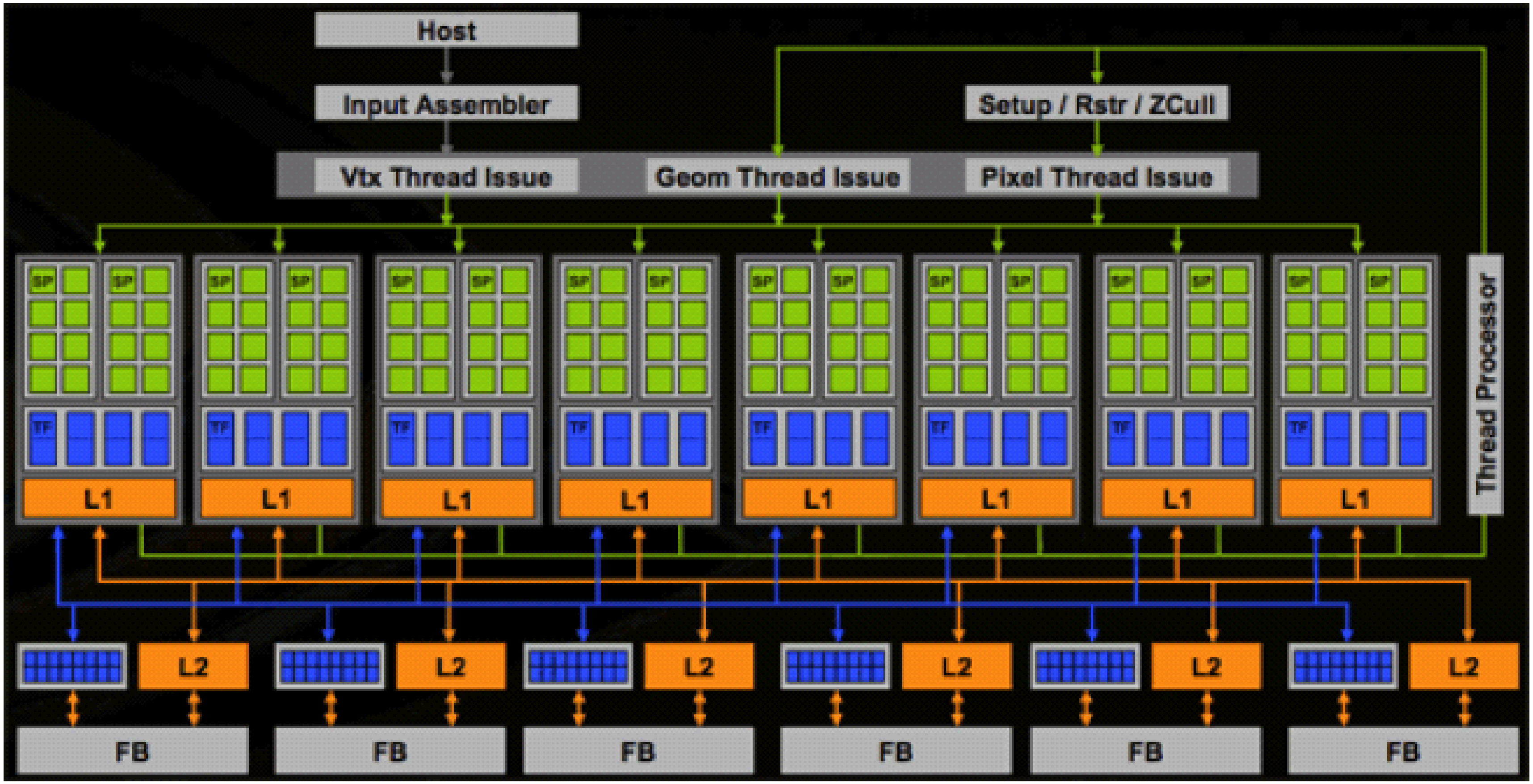


Phong Shading in Cg: Vertex Shader

- ❖ First part of pipeline
- ❖ Connectors: what kind of data is transferred to/from vertex program?
- ❖ Actual vertex shader



NVIDIA G80 Block Diagram



- ❖ Very little of this is graphic specic
- ❖ ...but, assumes threads are independent

Hyper "Core" Computers

Speculation about the computer of the next decade:

- 10s of CPU cores
  - Use for scheduling
  - Use for \irregular" part of problem
  - Maybe higher precision (correction steps)
- 100s of GPU cores
  - Use for \regular" part of problem
- NUMA (Non-Uniform Memory Access) for both
  - Programming languages must expose this
  - Runtime systems?
  - Always out-of-(some)-core
- Clusters of these?
  - OpenMP/MPI not sufficient

Limitations of GPUs

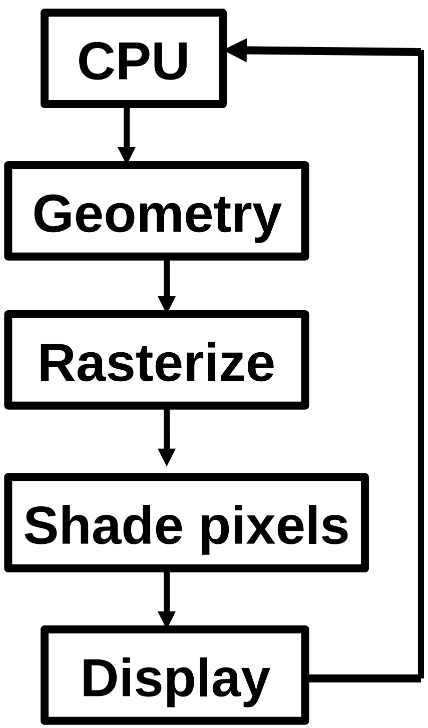
If the GPU is so great, why are we still using the CPU?
You can not simply “port" existing code and algorithms!

- ❖ Data-stream mindset required
  - ➢ Parallel algorithms
  - ➢ New data structures (dynamic data structures are troublesome)
- ❖ Not suitable to all problems
  - ➢ Pointer chasing impossible or inecient
  - ➢ Recursion
- ❖ Debugging is hard
  - ➢ Hardware is designed without debug bus
  - ➢ Driver is closed
- ❖ Huge performance clis
- ❖ No standard API
  - ➢ More about this later...

GPU Programming

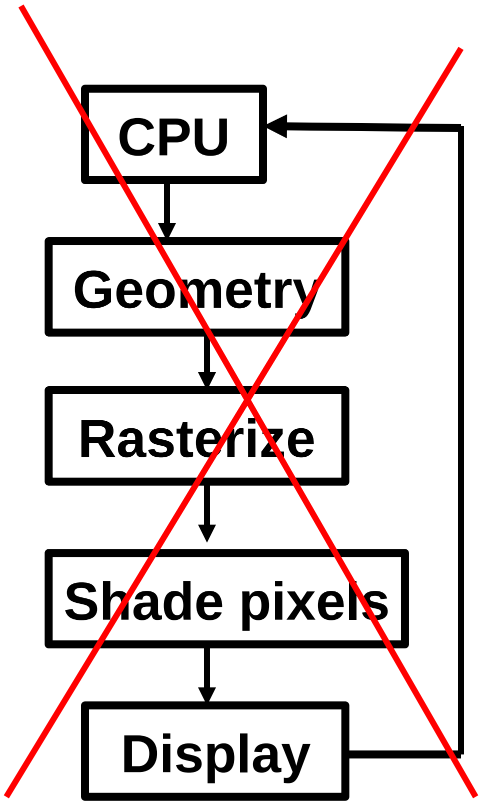
- GPUs have traditionally been closed architectures.
  - Must program them through closed-source graphics driver
  - Driver is like an OS (threads, scheduling, protected memory)
- OpenGL/DirectX are standard, but
  - Designed for graphics, not general purpose computations
  - Many revisions of each standard
    - New revisions for each HW-generation
  - Allows for \capabilities"
  - Large variations between vendors
- Both vendors now have dedicated GPGPU APIs
  - Nvidia CUDA (Compute Unified Device Architecture)
  - AMD CTM (Close To Metal) – AMD ATI - FireStream
- GPGPU version" of hardware as well

Computer Graphics



- ❖ Hardware mimicked graphics APIs
- ❖ It is possible to formulate many problems in this framework
  - ➢ Uses graphics APIs
  - ➢ Classical GPGPU"

Computer Graphics

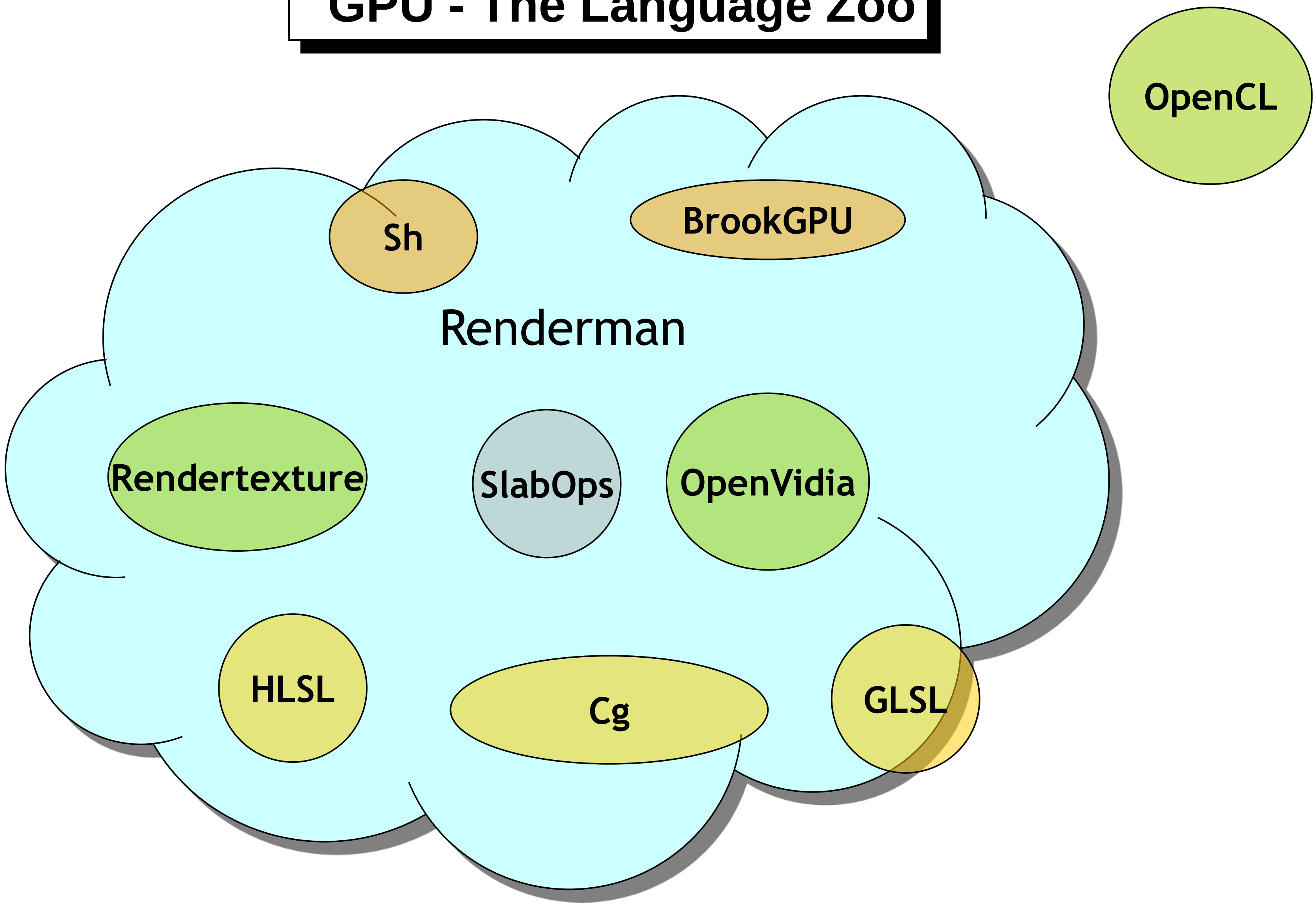


- Hardware mimicked graphics APIs
- It is possible to formulate many problems in this framework
  - Uses graphics APIs
  - Classical GPGPU"

DO NOT DO THIS ANYMORE!
(Unless for graphics)

Overview –
GPU Programming “Languages”

GPU - The Language Zoo



GPU - Some History

- Cook and Perlin first to develop languages for performing **shading calculations**
- Perlin computed noise functions procedurally; introduced control constructs
- Cook developed idea of *shade trees* @Lucasfilm
- These ideas led to development of Renderman at Pixar (Hanrahan *et al*) in 1988.
- Renderman is STILL **shader language** of choice for high quality rendering !
- Languages intended for offline rendering; no interactivity, but high quality.

GPU - Some History

- After RenderMan, independent efforts to develop high level shading languages at SGI (ISL), Stanford (RTSL).
- ISL targeted fixed-function pipeline and SGI cards (remember compiler from previous lecture): goal was to map a RenderMan-like language to OpenGL
- RTSL took similar approach with programmable pipeline and PC cards (recall compiler from previous lecture)
- RTSL morphed into **Cg**.

GPU - Some History

- **Cg** was pushed by N**VIDIA** as a platform-neutral, card-neutral programming environment.
- In practice**, Cg** tends to work better on NVIDIA cards (better demos, special features etc).
- ATI made brief attempt at competition with Ashli/RenderMonkey.
- HLSL was pushed by Microsoft as a DirectX-specific alternative.
- In general, HLSL has better integration with the DirectX framework, unlike **Cg** with OpenGL/DirectX.

GPU – Level 1: Better Than Assembly ?

Overview –

C-like vertex, Cg, HLSL, GLSL, Data Types, Shaders,Compilation

GPU Lang. - Prog.: C-like vertex and fragment code

- ❖ Languages are specified in a C-like syntax.
- ❖ The user writes explicit vertex and fragment programs.
- ❖ Code compiled down into pseudo-assembly
  - this is a source-to-source compilation: no machine code is generated.
- ❖ Knowledge of the pipeline is essential
  - Passing array = binding texture
  - Start program = render a quad
  - Need to set transformation parameters
  - Buffer management a pain

GPU Lang. - Prog.: Cg

- Platform neutral, architecture "neutral" shading language developed by NVIDIA.

- One of the first GPGPU languages used widely.

- Because Cg is platform-neutral, many of the other GPGPU issues are not addressed
  - managing pbuffers
  - rendering to textures
  - handling vertex buffers

> "As we started out with Cg it was a great boost to getting programmers used to working with programmable GPUs. Now Microsoft has made a major commitment and in the long term we don't really want to be in the programming language busies"
>
> David Kirk, NVIDIA

GPU Lang. - Prog.: HLSL

- Developed by Microsoft; tight coupling with DirectX
- Because of this tight coupling, many things are easier (no RenderTexture needed !)
- Xbox programming with DirectX/HLSL (XNA)
- But…
  - Cell processor will use OpenGL/Cg

GPU Lang. - Prog.: GLSL

- GLSL is the latest shader language, developed by 3DLabs in conjunction with the OpenGL ARB, specific to OpenGL.
- Requires OpenGL 2.0
- NVIDIA doesn't yet have drivers for OpenGL 2.0 !! Demos (appear to be) emulated in software
- ATI appears to have native GL 2.0 support and thus support for GLSL.

Multiplicity of languages likely to continue

GPU Lang. - Prog.: Datatypes

- Scalars: float/integer/boolean
- Scalars can have 32 or 16 bit precision (ATI supports 24 bit, GLSL has 16 bit integers)
- vector: 3 or 4 scalar components.
- Arrays (but only fixed size)
- Limited floating point support; no underflow/overflow for integer arithmetic
- No bit operations
- Matrix data types
- Texture data type
  - power-of-two issues appear to be resolved in GLSL
  - different types for 1D, 2D, 3D, cubemaps.

GPU Lang. - Prog.: DatatBinding

Data Binding modes:

- uniform: the parameter is fixed over a glBegin()-glEnd() call.
- varying: interpolated data sent to the fragment program (like pixel color, texture coordinates, etc)
- attribute: per-vertex data sent to the GPU from the CPU (vertex coordinates, texture coordinates, normals, etc).
- Data direction:
- in: data sent into the program (vertex coordinates)
- out: data sent out of the program (depth)
- inout: both of the above (color)

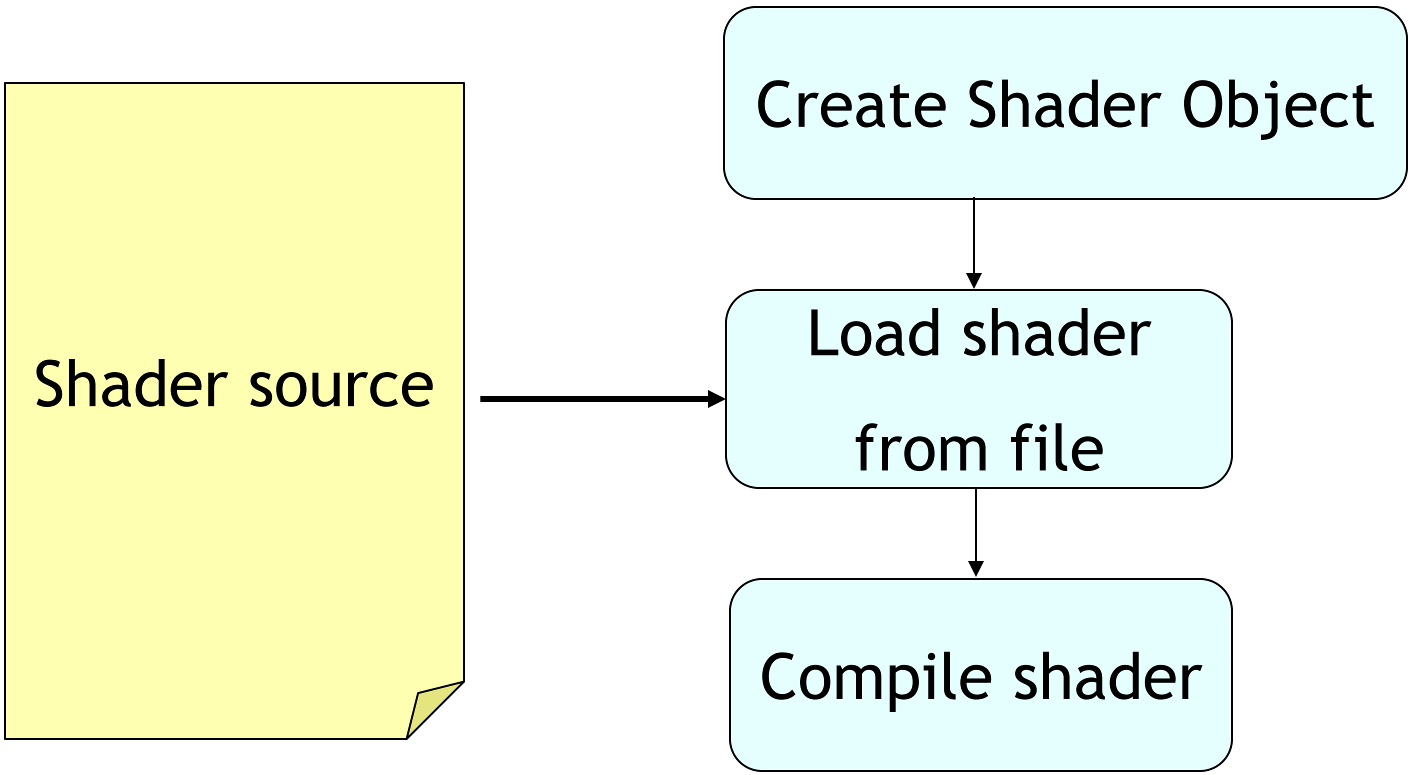
GPU Lang. - Prog.: Operations And Control Flow

- Usual arithmetic and special purpose algebraic ops (trigonometry, interpolation, discrete derivatives, etc)
- No integer mod…
- for-loops, while-do loops, if-then-else statements.
- **`discard`** allows you to kill a fragment and end processing.
- Recursive function calls are unsupported, but simple function calls are allowed
- Always one "main" function that starts the program, like C.

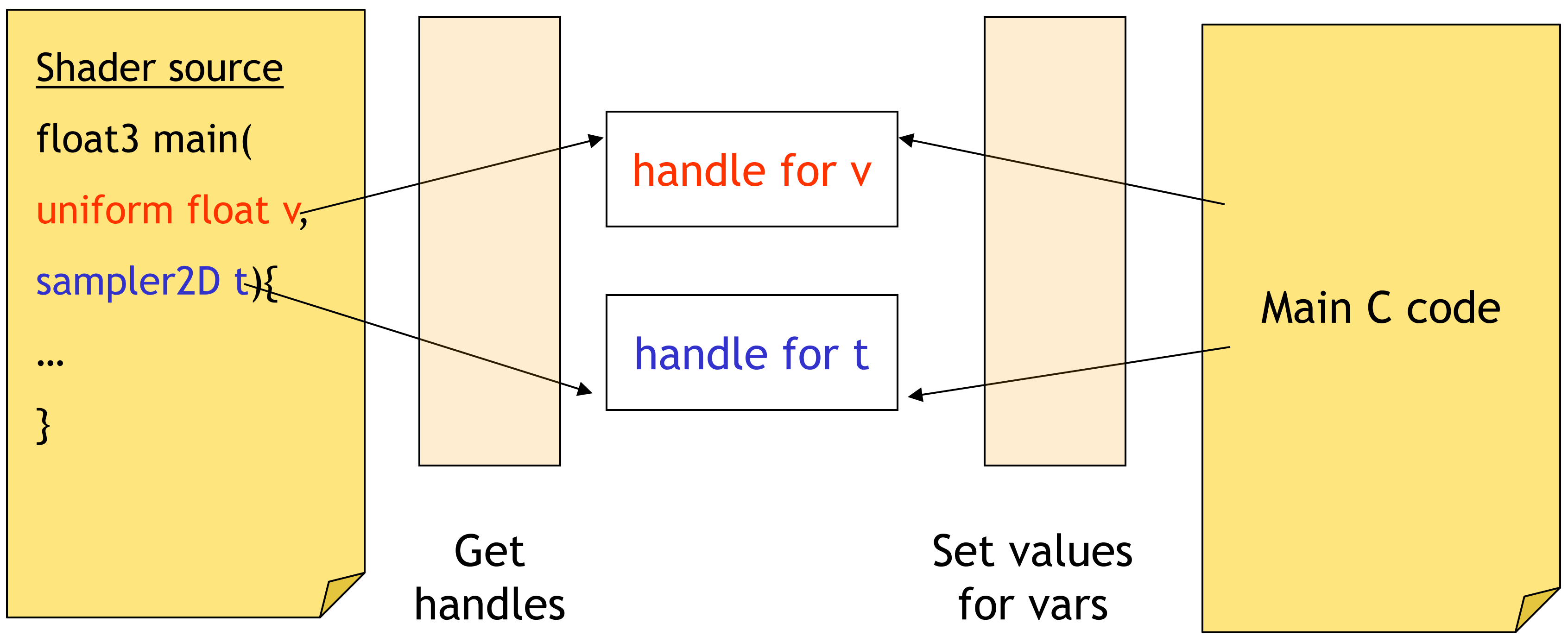
GPU Lang.-Prog.: working with Shaders : The Mechanics

- This is the most painful part of working with shaders.
- All three languages provide a “runtime” to load shaders, link data with shader variables, enable and disable programs.
- Cg and HLSL compile shader code down to assembly (“source-to-source”).
- GLSL relies on the graphics vendor to provide a compiler directly to GPU machine code, so no intermediate step takes place.

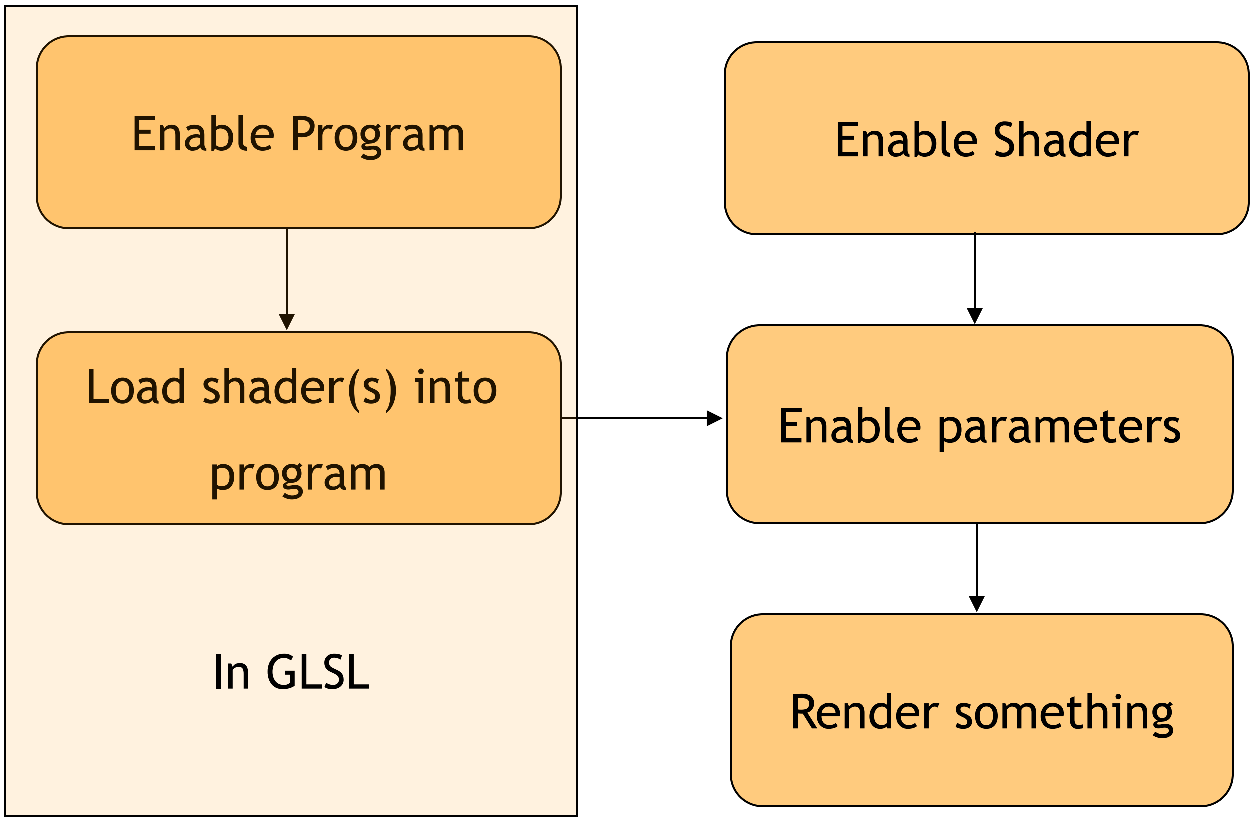
Step 1: Load the shader



Step 2: Bind Variables



Step 2: Run the Shaders



GPU Lang.-Prog.: Direct Compilation

- **Cg** code can be compiled to fragment code for different platforms (directx, nvidia, arbfp)
- HLSL compiles directly to directx
- GLSL compiles natively.
- It is often the case that inspecting the **Cg** compiler output reveals bugs, shows inefficiencies etc that can be fixed by writing assembly code (like writing asm routines in C)
- In GLSL you can’t do this because the code is compiled natively: you have to trust the vendor compiler !

GPU Lang.-Prog.: Overview

- ❖ Shading languages like Cg, HLSL, GLSL are ways of approaching Renderman but using the GPU.

- ❖ These will never be the most convenient approach for general purpose GPU programming

- ❖ But they will probably yield the most efficient code
  - you either need an HLL and great compilers
  - or you suffer and program in these.

GPU – Lang. Prog. ; Wrapper libraries

- Writing code that works cross-platform, with all extensions, is hard.
- Wrappers take care of the low-level issues, use the right commands for the right platform, etc.
- **Render Texture**:
  - Handles offscreen buffers and render-to-texture cleanly
  - works in both windows and linux (only for OpenGL though)
  - de facto class of choice for all Cg programming (use Cg for the code, and **RenderTexture** for texture management).

GPU – Lang. Prog. ; OpenVidia

- ❖ Video and image processing library developed at University of Toronto.
- ❖ Contains a collection of fragment programs for basic vision tasks (edge detection, corner tracking, object tracking, video compositing, etc)
- ❖ Provides a high level API for invoking these functions.
- ❖ Works with Cg and OpenGL, only on linux (for now)
- ❖ Level of transparency is low: you still need to set up GLUT, and allocate buffers, but the details are somewhat masked)

GPU – Lang. Prog. : OpenVidia Example

- Create processing object:
  - ```
    d=new FragPipeDisplay(<parameters>);
    ```
- Create image filter
  - ```
    filter1 = new GenericFilter(…,<cg-program>);
    ```
- Make some buffers for temporary results:
  - ```
    d->init_texture(0, 320, 240, foo);
    ```
  - ```
    d->init_texture4f(1, 320, 240, foo);
    ```
- Apply filter to buffer, store in output buffer
  - ```
    d->applyFilter(filter1, 0,1);
    ```
    ```

GPU – Lang. Prog. : High Level C-like languages

- Main goal is to hide details of the runtime and distill the essence of the computation.
- These languages exploit the ***stream*** aspect of GPUs explicitly
- They differ from libraries by being general purpose.
- They can target different backends (including the CPU)
- Either embed as C++ code (Sh) or come with an associated compiler (Brook) to compile a C-like language.

GPU Lang. Prog. : High Level C-like languages :Sh

- Open-source code developed by group led by Michael McCool at Waterloo
- Technical term is 'metaprogramming'
- Code is embedded inside C++; **no** extra compile tools are necessary.
- **Sh** uses a *staged compiler*: parts of code are compiled when C++ code is compiled, and the rest (with certain optimizations) is compiled at runtime.
- Has a very similar flavor to functional programming
- Parameter passing into streams is seamless, and resource constraints are managed by ***virtualization.***

GPU Lang. Prog. : High Level C-like languages :Sh And more ….. DirectX

- All kinds of other functions to extract data from streams and textures.
- Lots of useful 'primitive' streams like passthru programs and generic vertex/fragment programs, as well as specialized lighting shaders.
- **Sh** is closely bound to OpenGL; you can specify all usual OpenGL calls, and **Sh** is invoked as usual via a display() routine.
- Plan is to have **DirectX** binding ready shortly (this may be already be in)
- Because of the multiple backends, you can debug a shader on the CPU backend first, and then test it on the GPU.

GPU Lang. Prog. : High Level C-like languages
Brook GPU

- Open-source code developed by Ian Buck and others at Stanford.
- Intended as a pure stream programming language with multiple backends.
- Is not embedded in C code; uses its own compiler (brcc) that generates C code from a .br file.
- Workflow:
  - Write Brook program (.br)
  - Compile Brook program to C (brcc)
  - Compile C code (gcc/VC)

GPU Lang. Prog. : High Level C-like languages
Brook GPU

- Designed for general-purpose computing (this is primary difference in focus from **Sh**)
- You will almost never use any graphics commands in Brook.
- Basic data type is the stream.
- Types of functions:

GPU Lang. Prog. : High Level C-like languages
Brook GPU

- Types of functions:
  - Kernel: takes one or more input streams and produces an output stream.
  - Reduce: takes input streams and reduces them to scalars (or smaller output streams)
  - Scatter: $a[o_i] = s_i$. Send stream data to array, putting values in different locations.
  - Gather: Inverse of scatter operation. $s_i = a[o_i]$.
- Support of all operations are required … check.

GPU Lang. Prog. : High Level C-like languages Sh Vs Brook GPU

- ☺ Brook is more general: you don't need to know graphics to run it.
- ☺ Very good for prototyping
- ☹ You need to rely on compiler being good.
- ☹ Many special GPU features cannot be expressed cleanly.

- ☺ Sh allows better control over mapping to hardware.
- ☺ Embeds in C++; no extra compilation phase necessary.
- ☹ Lots of behind-the-scenes work to get virtualization: is there a performance hit ?
- ☹ Still requires some understanding of graphics.

NVIDIA CUDA (Compute Unified Device Architecture)

C-like API for programming newer Nvidia GPUs

- Computation kernels are written in C
  - Compiles with dedicated compiler, nvcc
- Kernels are executed as threads, threads organized into blocks
  - Programmer decides #threads, #threads/block, and mem/block
- Exposes different kinds of memory
  - Thread-local (register)
  - Shared per block
  - Global (not cached, write everywhere)
  - Texture (cached read only memory)
  - Constant(cached read only memory)
- Some synchronization primitives
- cudaMalloc, cudaMemcpy for allocating and copying memory

GPU Lang. Prog. : High Level C-like languages
The Big Picture

- The advent of Cg, and then Brook/Sh signified a huge increase in the number of GPU apps. Having good programming tools is worth a lot !

- The tools are still somewhat immature; almost non-existent debuggers and optimizers, and only one GPU simulator (Sm).

- I shouldn’t have to worry about the correct parameters to pass when setting up a texture for use as a buffer: we need better wrappers.

GPU Lang. Prog. : High Level C-like languages
The Big Picture

- Compiler efforts are lagging application development: more work is needed to allow for high level language development without compromising performance.

- In order to do this, we need to study stream programming. Maybe draw ideas from the functional programming world ?

- Libraries are probably the way forward for now.

Hyper "Core" Computers

Speculation about the computer of the next decade:

- 10s of CPU cores
  - Use for scheduling
  - Use for \irregular" part of problem
  - Maybe higher precision (correction steps)
- 100s of GPU cores
  - Use for \regular" part of problem
- NUMA (Non-Uniform Memory Access) for both
  - Programming languages must expose this
  - Runtime systems?
  - Always out-of-(some)-core
- Clusters of these?
  - OpenMP/MPI not sufficient

Limitations of GPUs

If the GPU is so great, why are we still using the CPU?
You can not simply “port" existing code and algorithms!

- Data-stream mindset required
  - Parallel algorithms
  - New data structures (dynamic data structures are troublesome)
- Not suitable to all problems
  - Pointer chasing impossible or inecient
  - Recursion
- Debugging is hard
  - Hardware is designed without debug bus
  - Driver is closed
- Huge performance clis
- No standard API
  - More about this later...

GPU Programming

- GPUs have traditionally been closed architectures.
  - Must program them through closed-source graphics driver
  - Driver is like an OS (threads, scheduling, protected memory)
- OpenGL/DirectX are standard, but
  - Designed for graphics, not general purpose computations
  - Many revisions of each standard
    - New revisions for each HW-generation
  - Allows for \capabilities"
  - Large variations between vendors
- Both vendors now have dedicated GPGPU APIs
  - Nvidia CUDA (Compute Unified Device Architecture)
  - AMD CTM (Close To Metal) – AMD ATI - FireStream
- GPGPU version" of hardware as well

Conclusions

- ❖ GPU Programming Language

- ❖ GPU Programming – OpenGL, DirectX, NVIDIA (CUDA), AMD (Brook+)

- ❖ OPECG-2009 -Hands-on session : Examples

References

1. Randi J. Rost, OpenGL – shading Language, Second Edition, Addison Wesley 2006
2. GPGPU Reference http://www.gpgpu.org
3. NVIDIA http://www.nvidia.com
4. NVIDIA tesla http://www.nvidia.com/object/tesla_computing_solutions.html
5. NVIDIA CUDA Reference http://www.nvidia.com/object/cuda_home.html
6. CUDA sample source code: http://www.nvidia.com/object/cuda_get_samples.html
7. List of NVIDIA GPUs compatible with CUDA: The href://www.nvidia.com/object/cuda_learn_products.html
8. Download the CUDA SDK: www.nvidia.com/object/cuda_get.html
9. Specifications of nVIDIA GeForce 8800 GPUs:
10. RAPIDMIND http://www.rapidmind.net
11. Peak Stream - Parallel Processing (Acquired by Google in 2007) http:/www.google.com
12. guru3d.com http://www.guru3d.com/news/sandra-2009-gets-gpgpu-support/
    ATI & AMD http://ati.amd.com/products/radeon9600/radeon9600pro/index.html
13. AMD http:www.amd.com
14. AMD Stream Processors http://ati.amd.com/products/streamprocessor/specs.html
15. RAPIDMIND & AMD http://www.rapidmind.net/News-Aug4-08-SIGGRAPH.php
16. Merrimac - Stream Architecture Standford Brook for GPUs
    http://www-graphics.stanford.edu/projects/brookgpu/
17. Standford : Merrimac - Stream Architecture http://merrimac.stanford.edu/
18. ATI RADEON - AMD http://www.canadacomputers.com/amd/radeon/
19. ATI & AMD - Technology Products http://ati.amd.com/products/index.html
20. Sparse Matrix Solvers on the GPU ; conjugate Gradients and Multigrid by *Jeff Bolts, Ian Farmer, Eitan Grinspum, Peter Schroder* , Caltech Report (2003); Supported in part by NSF, nVIDIA, etc....
21. Scan Primitives for GPU Computing by *Shubhabrata Sengupta, Mark Harris*, Yao Zhang and John D Owens* University of California Davis & *nVIDIA Corporation *Graphic Hardware (2007).*
22. Horm D; *Stream reduction operations for GPGPU applciations in GPU Genes 2 Phar M.,* (Ed.) Addison Weseley, March 2005; Chapter 36, pp. 573-589 Graphic Hardware (2007).
23. *Bollz J., Farmer I., Grinspun F., Schroder F :* Sparse Matris Solvers on the GPU ; Conjugate Gradients and multigrid ACM Transactions on Graphics *(Proceedings of ACM SIGRAPH 2003) 22, 2 (Jul y2003) pp 917-924 Graphic Hardware (2007).*
24. NVIDIA CUDA Compute Unified Device Architecture - Programming Guide - Version 1.1 November 2007

References

25. Tom R. Halfhill, *Number crunching with GPUs PeakStream Math API Exploits Parallelism in Graphics Processors, Ocotober 2006; Microprocessor* *http://www.mdronline.com*
26. Tom R. Halfhill, *Parallel Processing with CUDA Nvidia's High-Performance Computing Platform Uses Massive Multithreading ;* Microprocessors, Volume 22, Archive 1, January 2008 *http://www.mdronline.com*
27. J. Tolke, M.Krafczyk *Towards Three-dimensional teraflop CFD Computing on a desktop PC using graphics hardware Institute for Computational Modeling in Civil Engineering,* TU Braunschweig (2008)
28. I. Buck, T. Foley, D. Horn, J. Sugerman, K. Fatahalian, M. Hoston, P.Hanrahan, Brook for GPUs ; *Stream Computing on GRaphics Hadrware, ACM Tran. GRaph (SIGGRAPH) 2008*
29. Z. Fan, F. Qin, A.E. Kaufamm, S. Yoakum-Stover*, GPU cluster for Hgh Performance Computing in : Proceedings of ACM/IEEE Superocmputing Conference 2004 pp. 47-59.*
30. J. Kriiger, R. Wetermann, *Linear Algeria operators for GPU implementation of Numerical Algorithms ACm Tran, Graph (SIGGRAPH) 22 (3) pp. 908-916. (2003)*
31. Tutorial SC 2007 SC05 : *High Performance Computing with CUDA*
32. *FASTRA http://www.fastra.ua.ac.bc/en/faq.html*
33. *AMD Stream Computing software Stack ; http://www.amd.com*
34. *BrookGPU : http://graphics standafrod.edu/projects/brookgpu/index.html*
35. *FFT – Fast Fourier Transform www.fftw.org*
36. BLAS – *Basic Linear Algebra Suborutines – www.netlibr.org/blas*
37. LAPACK : *Linear Algebra Package – www.netlib.org/lapack*
38. Dr. Larry Seller, Senipr Principal Engineer; Larrabee : A Many-core Intel Architecture for Visual computing, Intel Deverloper FORUM 2008
39. *Tom R Halfhill,* Intel's Larrabee Redefines GPUs – Fully Programmable Many core Processor Reaches Beyond Graphics, Microprocessor Report September 29, 2008
40. Tom R Halfhill AMD's Stream Becomes a River – Parallel Processing Platform for ATI GPUs Reaches More Systems, Microprocessor Report December 2008
41. AMD's ATI Stream Platform http://www.amd.com/stream
42. General-purpose computing on graphics processing units (GPGPU) http://en.wikipedia.org/wiki/GPGPU
43. Khronous Group, OpenGL 3, December 2008 URL : http://www.khronos.org/opencl

References

44. *Mary Fetcher and Vivek Sarkar,* Introduction to GPGPUS – Seminar on Heterogeneous Processors, Dept. of computer Science, Rice University, October 2007
45. *OpenCL - The open standard for parallel programming of heterogeneous systems* URL : http://www.khronos.org/opencl
46. Tom R. Halfhill, Parallel Processing with CUDA Nvidia's High-Performance Computing Platform Uses Massive Multithreading ; Microprocessors, Volume 22, Archive 1, January 2008 http://www.mdronline.com
47. *Matt Pharr (Author), Randima Fernando,* GPU Gems 2: Programming Techniques for High-Performance Graphics and General-Purpose Computation *,Addison Wesley , August 2007*
48. *NVIDIA GPU Programming Guide* http://www.nvidia.com
49. *Perry H. Wang1, Jamison D. Collins1, Gautham N. Chinya1, Hong Jiang2, Xinmin Tian3* , EXOCHI: Architecture and Programming Environment for A Heterogeneous Multi-core Multithreaded System, PLDI'07
50. Karl E. Hillesland, Anselmo Lastra GPU Floating-Point Paranoia, University of North Carolina at Chapel Hill
51. KARPINSKI, R. 1985. Paranoia: A floating-point benchmark. Byte Magazine 10, 2 (Feb.), 223–235.
52. GPGPU Web site : http://www.ggpu.org
53. Graphics Processing Unit Architecture (GPU Arch) With a focus on NVIDIA GeForce - 6800 GPU, Ajit Datar, Apurva Padhye Computer Architecture
54. Nvidia 6800 chapter from GPU Gems 2 http://download.nvidia.com/developer/GPU_Gems_2/GPU_Gems2_ch30.pdf
55. OpenGL design http://graphics.stanford.edu/courses/cs448a-01-fall/design_opengl.pdf
56. OpenGL programming guide (ISBN: 0201604582)
57. Real time graphics architectures lecture notes http://graphics.stanford.edu/courses/cs448a-01-fall/
58. GeForce 256 overview http://www.nvnews.net/reviews/geforce_256/gpu_overviews.html
59. GPU Programming "Languages http://www.cis.upenn.edu/~suvenkat/700/
60. Programming the GPU and a brief intro to the OPENGL shading language – Marcel Cohan & VVR Talk
61. *Johan Seland,* GPU Programming and Computing, Workshop on High-Performance and Parallel Computing Simula Research Laboratory October 24, 2007
62. *Daniel Weiskopf,* Basics of GPU-Based Programming, Institute of Visualization and Interactive Systems, Interactive Visualization of Volumetric Data on Consumer PC Hardware: Basics of Hardware-Based Programming University of Stuttgart, **VIS 2003**

References

1. AMD Accelerated Parallel Processing (APP) SDK (formerly ATI Stream) with OpenCL 1.1 Support http://developer.amd.com/sdks/AMDAPPSDK/Pages/default.aspx
2. AMD Accelerated Parallel Processing (APP) SDK (formerly ATI Stream) with AMD APP Math Libraries (APPML); AMD Core Math Library (ACML); AMD Core Math Library for Graphic Processors (ACML-GPU) http://developer.amd.com/sdks/AMDAPPSDK/Pages/default.aspx
3. AMD Accelerated Parallel Processing (AMD APP) Programming Guide OpenCL : August 2012 http://developer.amd.com/sdks/AMDAPPSDK/assets/AMD_Accelerated_Parallel_Processing_OpenCL_Programming_Guide.pdf
4. AMD Developer Central - OpenCL Zone, http://developer.amd.com/zones/OpenCLZone/Pages/default.aspx
5. AMD Developer Central - Programming in OpenCL http://developer.amd.com/zones/OpenCLZone/programming/Pages/default.aspx
6. AMD Developer Central - Programming in OpenCL - Benchmarks performance http://developer.amd.com/zones/OpenCLZone/programming/pages/benchmarkingopenclperformance.aspx
7. *The open standard for parallel programming of heterogeneous systems* URL : http://www.khronos.org/opencl
8. OpenGL design http://graphics.stanford.edu/courses/cs448a-01-fall/design_opengl.pdf
9. OpenGL programming guide (ISBN: 0201604582)
10. Real time graphics architectures lecture notes http://graphics.stanford.edu/courses/cs448a-01-fall/
11. GeForce 256 overview http://www.nvnews.net/reviews/geforce_256/gpu_overviews.html
12. GPU Programming “Languages http://www.cis.upenn.edu/~suvenkat/700/
13. Programming the GPU and a brief intro to the OPENGL shading language – Marcel Cohan & VVR Talk
14. *Johan Seland,* GPU Programming and Computing, Workshop on High-Performance and Parallel Computing Simula Research Laboratory October 24, 2007

References

1. NVIDA CUDA C Programming Guide Version V4.0, May 2012 (5/6/2012) http://developer.download.nvidia.com/compute/DevZone/docs/html/C/doc/CUDA_C_Programming_Guide.pdf
2. NVIDIA Developer Zone http://developer.nvidia.com/category/zone/cuda-zone
3. NVIDIA CUDA Toolkit 4.0 (May 2012) http://developer.nvidia.com/cuda-toolkit-4.0
4. NVIDIA CUDA Toolkit 4.0 Downloads http://developer.nvidia.com/cuda-toolkit
5. NVIDIA Developer ZONE – GPUDirect http://developer.nvidia.com/gpudirect
6. NVIDIA OpenCL Programming Guide for the CUDA Architecture version 4.0 Feb, 2012 (2/14,2012) http://developer.download.nvidia.com/compute/DevZone/docs/html/OpenCL/doc/OpenCL_Programming_Guide.pdf
7. Optimization : NVIDIA OpenCL Best Practices Guide Version 1.0 Feb 2012 http://developer.download.nvidia.com/compute/DevZone/docs/html/OpenCL/doc/OpenCL_Best_Practices_Guide.pdf
8. NVIDIA OpenCL JumpStart Guide - Technical Brief http://developer.download.nvidia.com/OpenCL/NVIDIA_OpenCL_JumpStart_Guide.pdf
9. NVIDA CUDA C BEST PRACTICES GUIDE (Design Guide) V4.0, May 2012
10. http://developer.download.nvidia.com/compute/DevZone/docs/html/C/doc/CUDA_C_Best_Practices_Guide.pdf
11. NVIDA CUDA C Programming Guide Version V4.0, May 2012 (5/6/2012)
12. http://developer.download.nvidia.com/compute/DevZone/docs/html/C/doc/CUDA_C_Programming_Guide.pdf

Thank You

Any questions ?