General Purpose Computing on Graphical Processing Units (**GPGPU** / GPGP /GP^2)

By Simon J.K. Pedersen
Aalborg University, Oct 2008
VGIS, Readings Course Presentation no. 7

Presentation Outline

- Part 1: Introduction
- Part 2: GPGPU Environments
- Part 3: GPGPU Programming
- Part 4: Using CUDA

Part 1: Introduction

- **Part 1: Introduction**
- Part 2: GPGPU Environments
- Part 3: GPGPU Programming
- Part 4: Using CUDA

Why General Purpose Computing on Graphical Processing Units

- The cheapest available computing power
- Increase in CPU frequency has come to an halt [4]
  - GPU computing power is still on the rise, due to parallelism
- CPUs are becoming increasingly parallel
- GPU programming (stream processing) is the programming paradigm of the multi-core future

Limitations to GPGPU

- None (the sky is the limit) ;)
- Memory access on current hardware pose a bottleneck
- Thus, best suited for algorithms with high “arithmetic intensity” = many instructions per memory access.
- Lacking branching capabilities of the CPU
- Development environments are still relative immature, few debugging/profiling tools

Computing Power

- What is computing power?
  - Memory access time
  - Clock frequency
  - Number of processors
  - Number of transistors
  - Bit-wise logic
  - Integer arithmetic
  - Floating Point Operations per Second (FLOPS)
  - ....

Computing Power cont

- One common measure is FLOPS
  - Many scientific problems deal with floating points
  - Alternatively use MIPS (Million of Instructions Per Second)
- Floating point precision (Standard IEEE 754)
  - Consumer GPUs at least 24-bit floating point since DirectX 9.0 [2]
  - Industry GPUs recently moved to 64-bit (e.g. AMD FireStream [1])

Measuring FLOPS

- Marketing FLOPS vs. Real-life FLOPS
  - Typically these do not match
  - Marketing FLOPS:

    No of Cores * Core Clock Frequency * No of Floating Point operations Per Clock Frequency

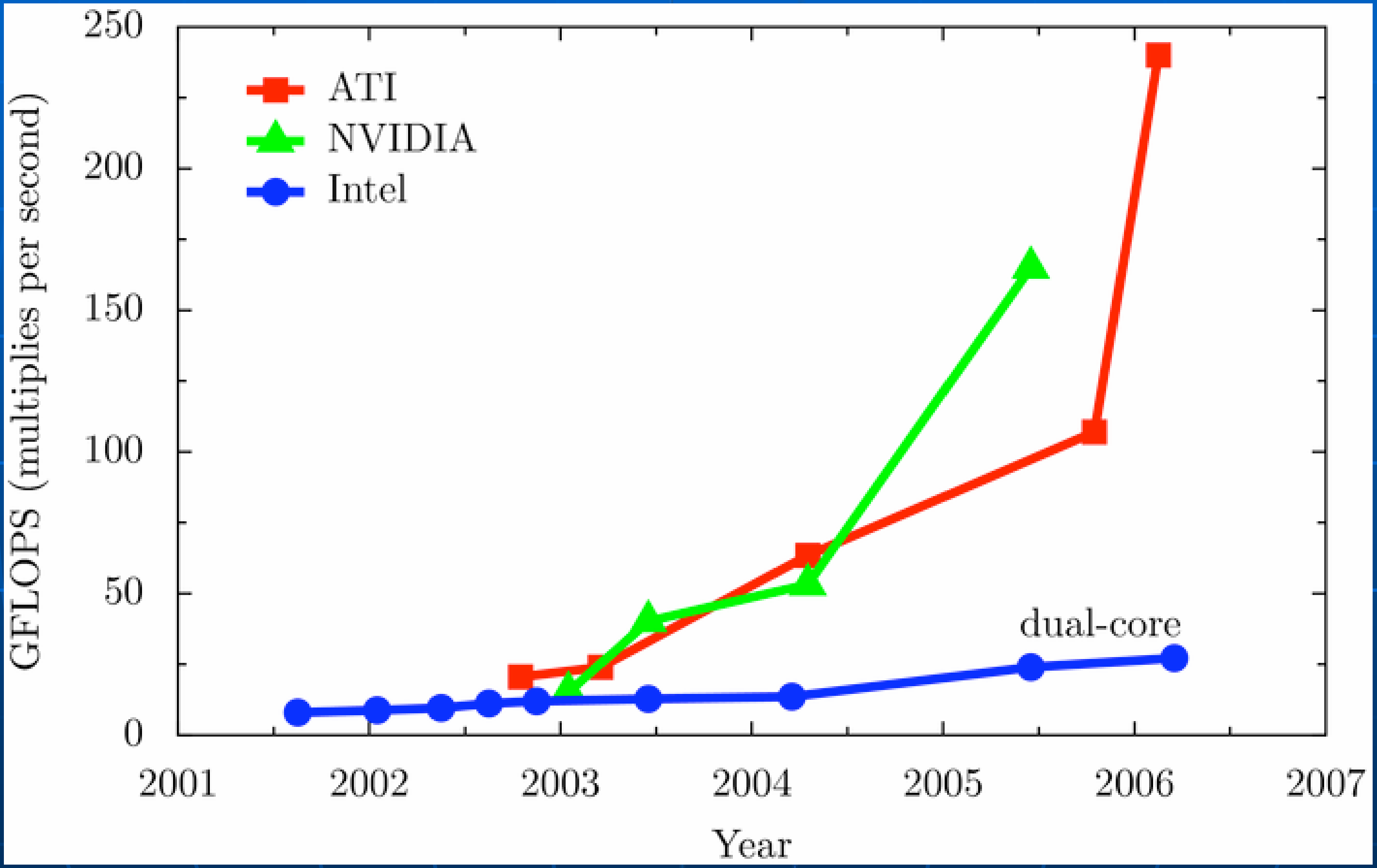
    nVidia 280GTX: 240 * 1.296GHz * 3 = 933 GFLOPS
  - Assumption: 3 FLOPS, MAD (Multiple Add) + MUL (Multiplication) per clock.

Measuring FLOPS cont

- Difficult to compare FLOPS measurements across different architectures (CELL, CPU, GPU)

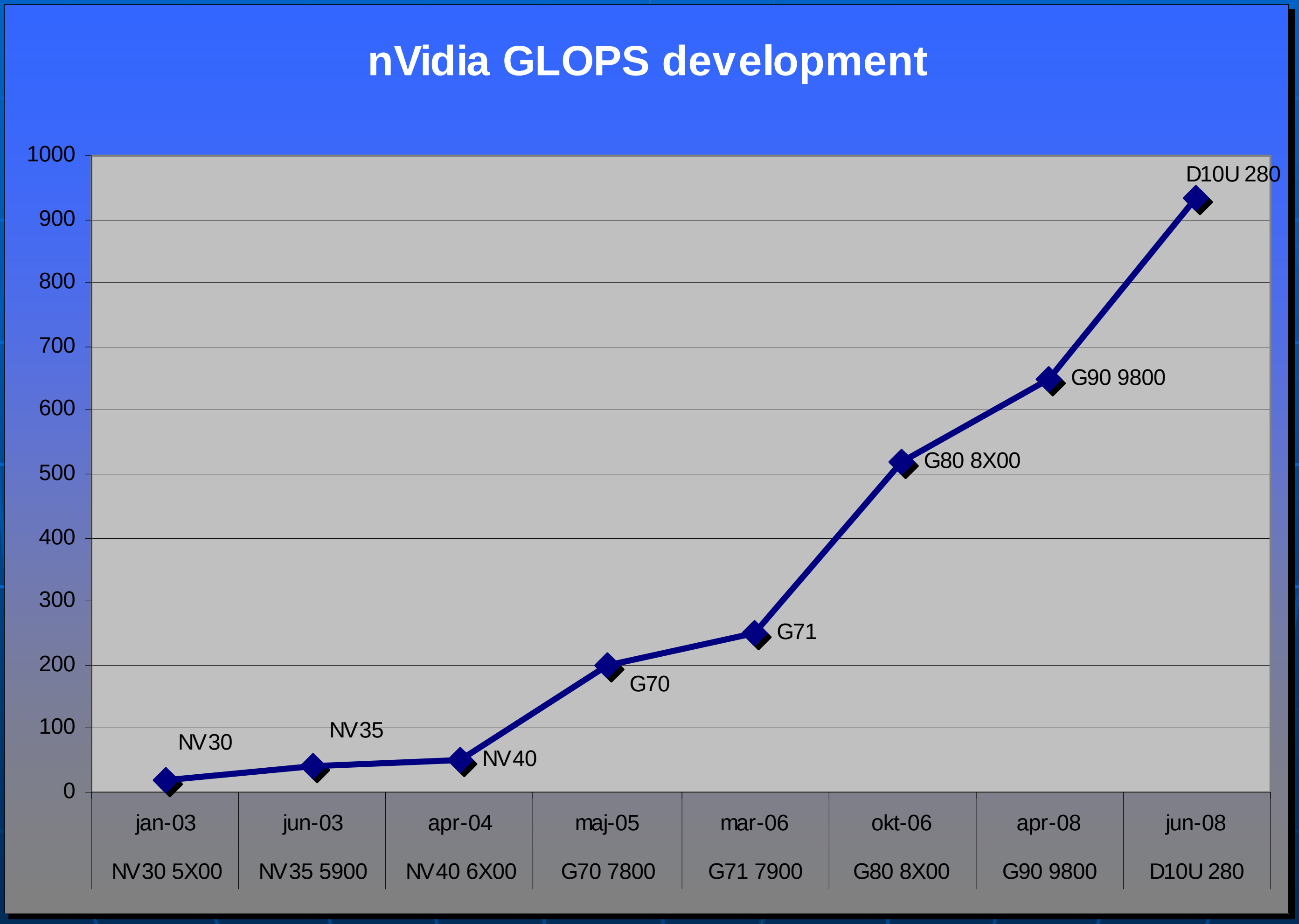
- Fair comparisons require benchmarking
  - LINPACK Benchmark
  - Solve a N x N system of linear equations
  - Architecture differences still a problem

Development of FLOPS



[8]

Development of FLOPS (nVidia)



Adapted from [3]

Fun Facts

- Gears of War: Modern Cross-Platform Game

	Game Simulation	Numeric Computation	Shading
Languages	C++, Scripting	C++	CG, HLSL
CPU Budget	10%	90%	n/a
Lines of Code	250,000	250,000	10,000
FPU Usage	0.5 GFLOPS	5 GFLOPS	500 GFLOPS

[5]

GPGPU Research Publications

- Many researchers are beginning to take advantage of GPGPU
- Areas of particular interest
  - Flow simulations
  - Physics
  - Image Processing
  - Ray-tracing

GPGPU Havok FX

- Commercial physics middleware
- Utilize hybrid GPU and CPU for complex physics calculations
- Speed up 10x:
  - Collision detection on 15,000 objects
    - CPU (2.9GHz Core Duo 2): 6.2 fps
    - GPU (Geforce 8800GTX): 64.5 fps

GPGPU Research Publications 2

- Fast Virus Signature Matching on the GPU
  - Speed up 11x-27x compared to open soure Clam AV
  - Drawbacks:
    - Rely on CPU for verification
    - At most 64,000 signatures in database
    - Only does part of the scan process (no MD5 hashing)

GPGPU Research Publications 3

- The AES Implementation on the GPU
  - OpenGL based implementation
  - Relies heavily on integer processing
  - Speed up 1x-1.7x, for vertex and fragment shaders
  - Openssl CPU based implementation achieved 55MB/sec compared to 95MB/sec

Part 2: GPGPU Environments

- Part 1: Introduction
- **Part 2: GPGPU Environments**
- Part 3: GPGPU Programming
- Part 4: Using CUDA

GPGPU Environments

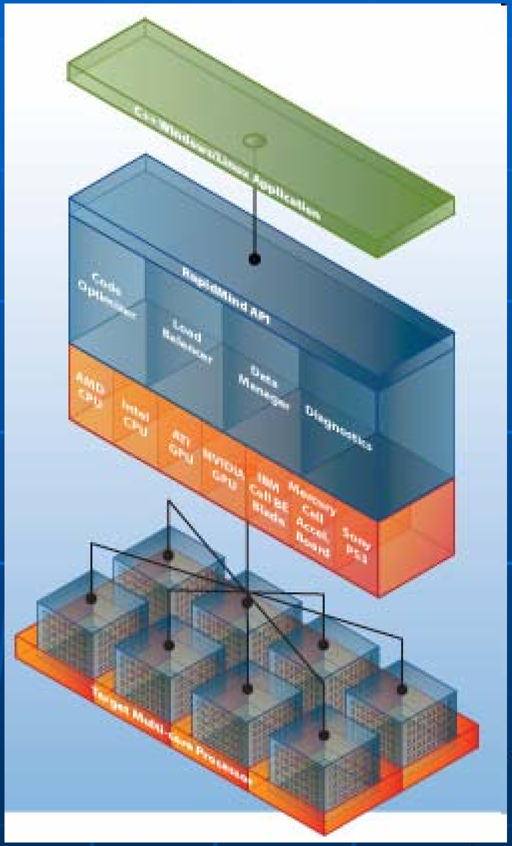
- No standard, each vendor has its own API
- Rapid development within the last few years (expected to continue)
- GPGPU APIs:
  - Shaders (Dx8, 2000)
  - RapidMind (early 2006)
  - AMD-ATI (CTM (Nov 06), Stream SDK)
  - nVidia (CUDA) (Nov 06)
  - Apple/Khronos (OpenCL) (Yet to be finalized)

Shader Languages

- Languages: GLSL, Cg/HLSL
- Programmable Shaders
  - Vertex (Position, Color, Texture Coords, Normals)
  - Fragment (Per Pixel)
- DirectX 8 (Shader Model 1.1)
- DirectX 8.1 (SM 1.2, 1.3, 1.4)
- DirectX 9 (SM 2.x)
- DirectX 10 (SM 4.0, Geometry Shaders)
- DirectX 11 (SM 5.0, GPGPU)

RapidMind Development Platform

- Started as a commercialization of research (Sh) from University of Waterloo (Canada)
- Middleware between high level C++ and the hardware
- Very broad platform support
  - Hardware: CELL, GPU (nVidia, AMD FireSteam Radeon Series), CPU (Intel, AMD)
  - Software: Mac OS X, Windows, Unix (Ubuntu, Red Hat, Fedora etc.)
- Easy to use, special data types and loop syntax
- Commercial product ☹



nVidia CUDA (Common Unified Device Architecture)

- Widespread, 50 million graphic cards sold capable of running CUDA [9]
- Support for Linux and Windows
- Widely used in research
- High level C syntax-like language
  - Exposes the underlying hardware structure
  - Skilled programmers able to take full advantage of the hardware
- Shipped with BLAS and FFT libraries

AMD-ATI

- CTM (Close to metal)
  - First attempt on GPGPU, now discontinued
- Current solution: Stream Computing SDK 1.0
  - Includes Brook+, APL, ACML, CAL
- Brook is a stream programming language similar to ANSI C for GPGPU
  - Access to GPU resources via OpenGL, DirectX, or CTM
- AMD will be supporting OpenCL and DirectX 11

OpenCL (Open Computing Language) [7]

- Support CPUs and GPUs and combinations
- Profiles for desktop and handheld devices
- Open language like OpenGL and OpenAL
- Specifications currently being review by Khronos Group
- Proposed by Apple
- Already implemented as performance enhancing technology in Mac OS X (Snow Leopard)

OpenCL cont.

- Official support from AMD
- Based on a subset of ISO C99
- IEEE 754 floating point spec. compliant
- Integration with OpenGL (sharing of data)
- Built in C data types (vectors, image types, data type conversions)
- Few C restrictions (Recursion, function points)

Part 3: GPGPU Programming

- Part 1: Introduction
- Part 2: GPGPU Environments
- **Part 3: GPGPU Programming**
- Part 4: Using CUDA

Stream processing/computing [6]

- Computational problems that can be split into parallel identical operations and run simultaneously
- Stream processing uses the SIMD (single instruction, multiple data) methodology
- The data is defined as a stream
- The collection of operations applied to the stream is typically called a kernel function
- Uniform streaming is when the same kernel is applied to all elements of the stream

Stream Processing on the GPU

- The host (CPU) sees the GPU as co-processor
- Some definitions:
  - Host memory
  - Device (GPU) memory
- The co-processor cannot access the host memory
- The host can transfer data to the device memory

The CUDA approach

- The remaining of the presentation is based on NVIDIA CUDA
- Maps well to other GPGPU APIs
- Bottom up walk-through

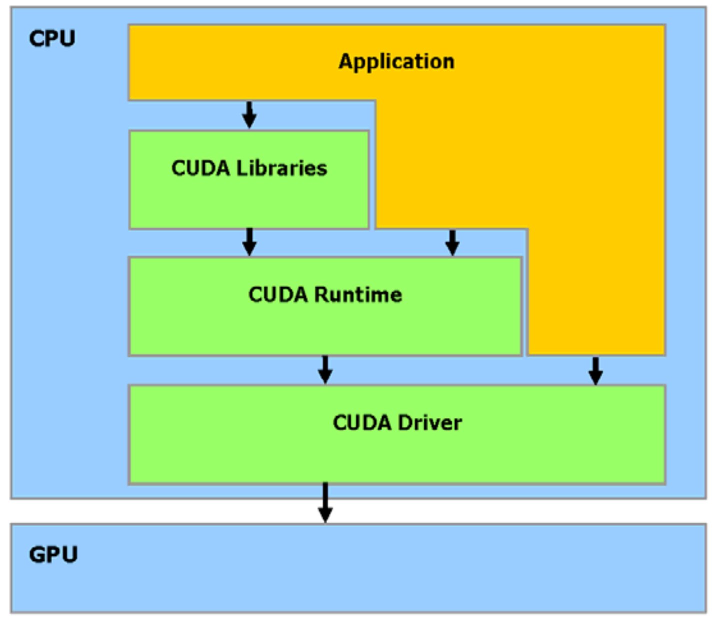
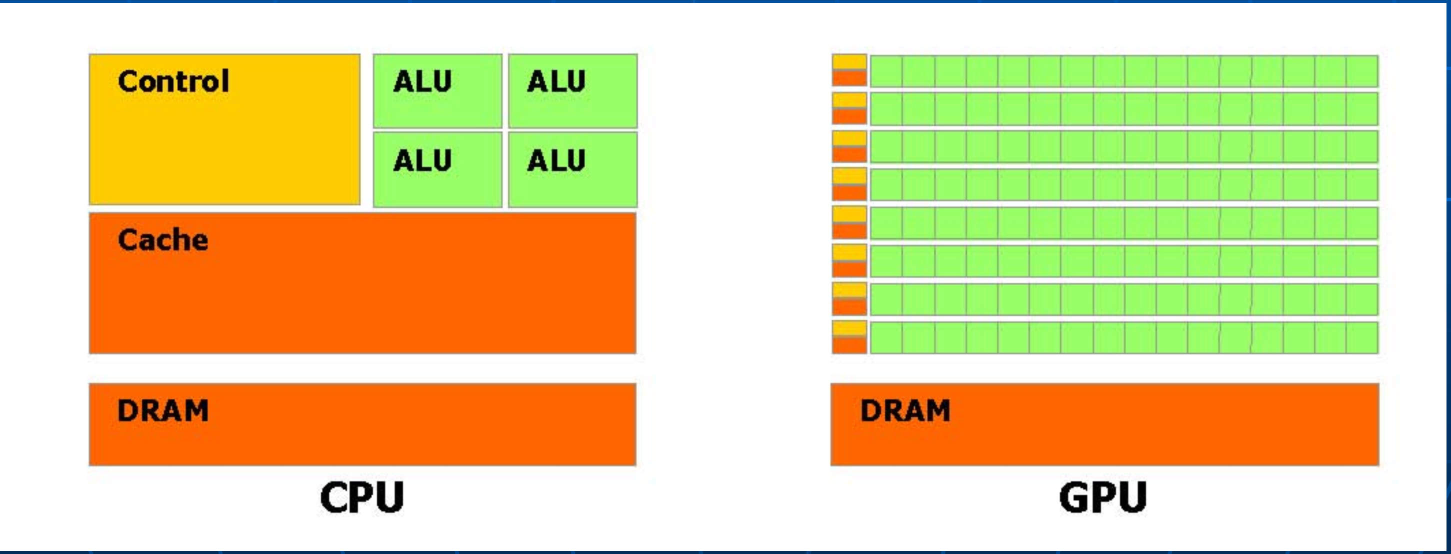


Figure 1-3. Compute Unified Device Architecture Software Stack

[10]

GPU Hardware vs. CPU

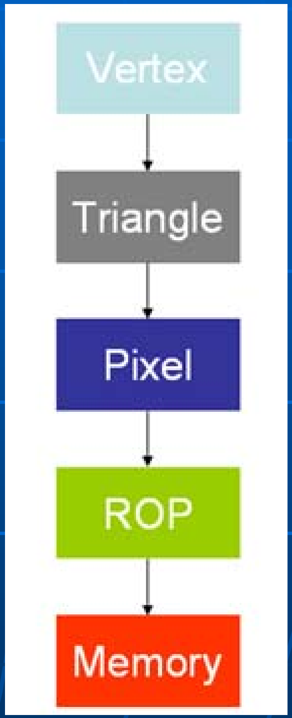
- What makes GPUs different
  - Number of transistors and their purpose
  - Memory bandwidth CPU 10GB/s, GPU 100GB/s
  - Production methods and cycles 6 vs 24months



From NVIDIA CUDA Programming Guide

GPU Hardware Model

- In the old days, 1-2 year ago
  - Vertices, fragments or textures can constitute the stream
  - Vertex and Fragment shaders can constitute the kernels
  - Each shader unit should produce the output solely from the input (no additional memory lookups or shared data between shader units)
    - Feed-forward



The elements of a traditional GPU pipeline. [10]

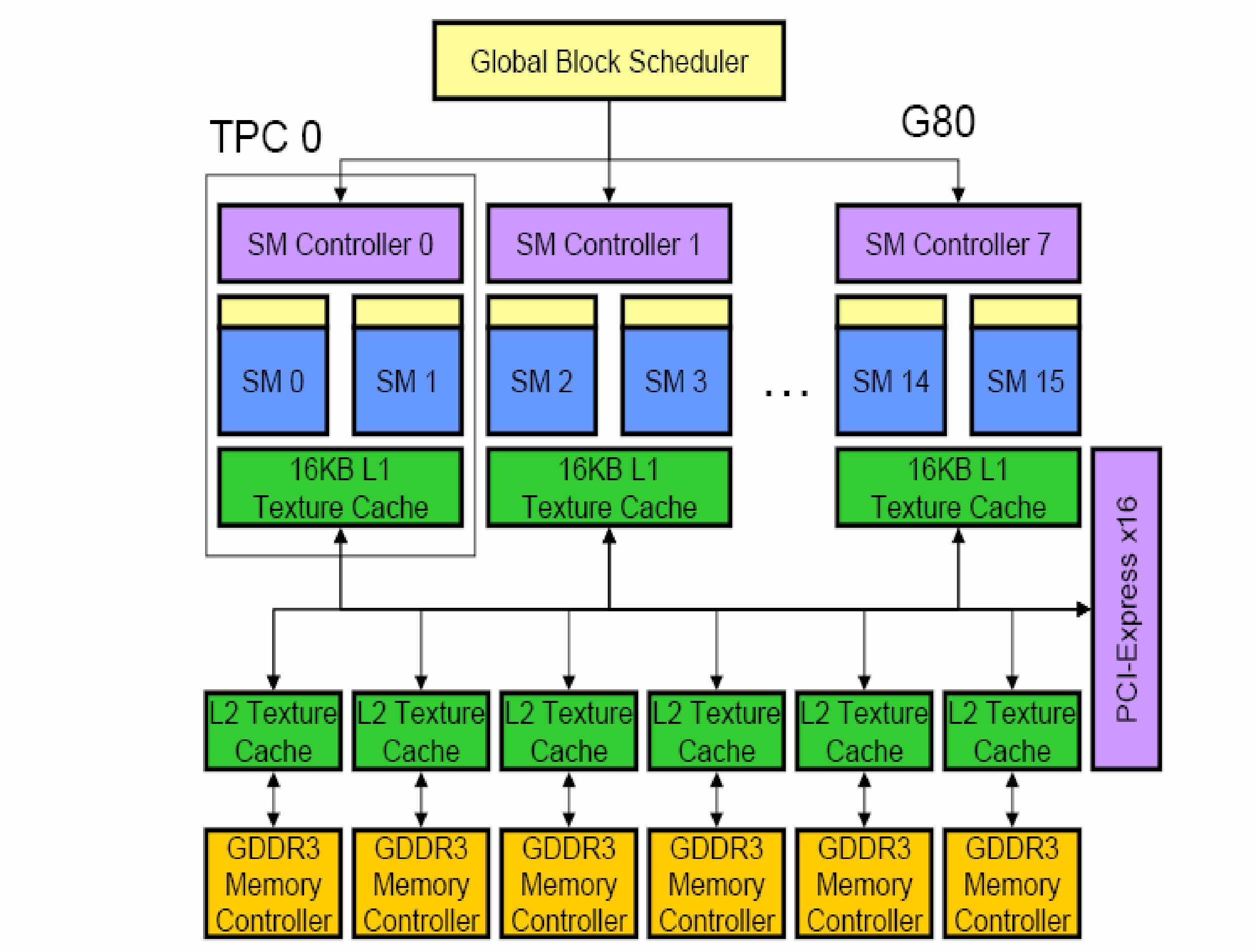
GPU Hardware Model cont.

- Today
  - New abstraction level, unified shaders or simply steam processors (SP)
  - Local and global memory
  - Possible to read and write from global memory (gather/scatter)
- An example the Geforce 8800 GTX

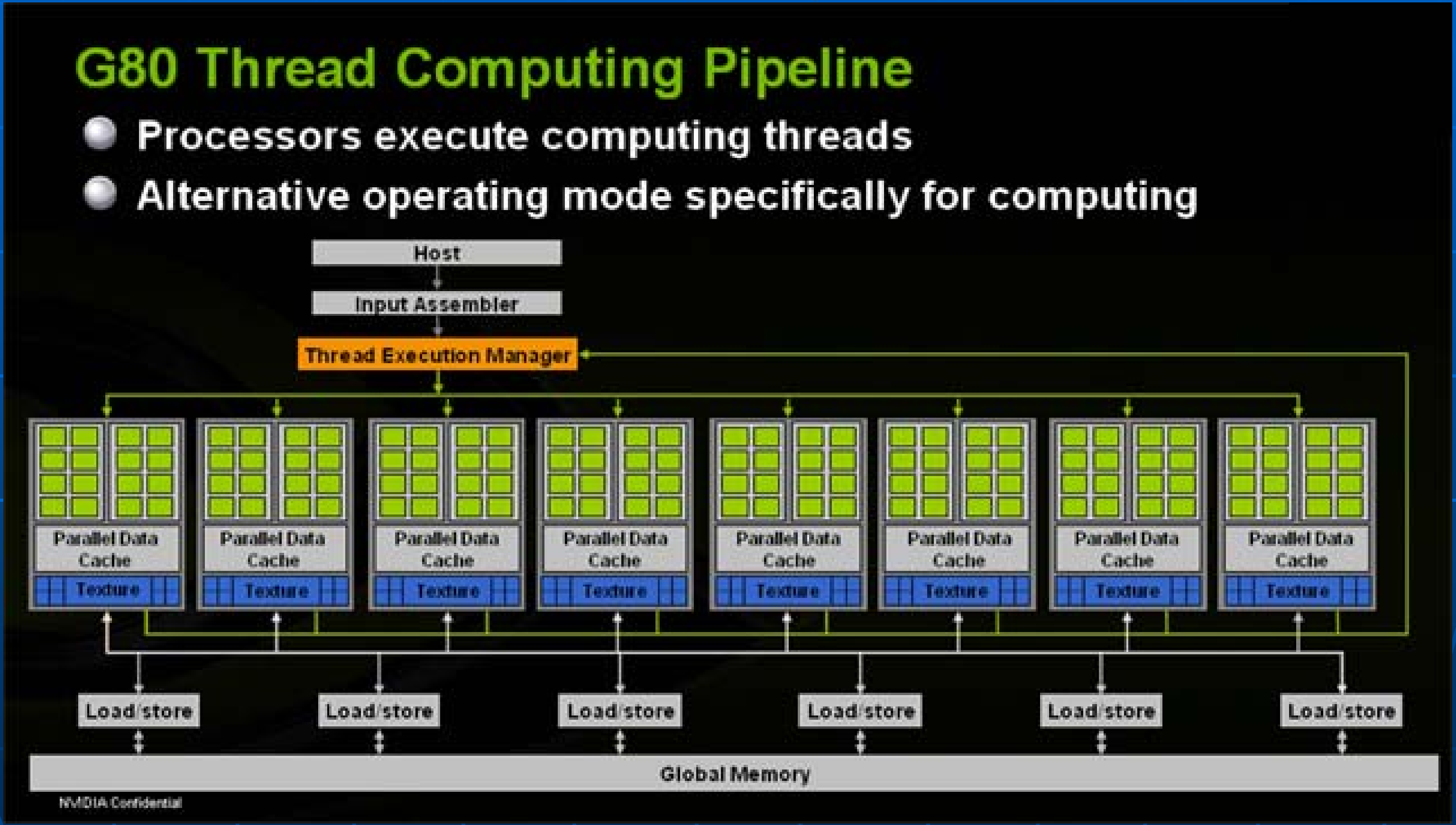
The Geforce 8800 GTX Hardware Architecture in details

- Unified shader design
- 8 Thread Processing Clusters (TPC)
  - Each consist of 2 streaming multiprocessors (SM)
    - Which again consist of 8 streaming processors (SP) clocked at 1.35GHz
  - Texture pipeline providing memory access

The Geforce 8800 GTX Hardware Architecture in details



The Geforce 8800 GTX Hardware Architecture in details



[10]

The Geforce 8800 GTX Hardware Architecture in details

- The memory/cache available in a streaming multiprocessor
  - 16KB shared memory
  - 64KB of constant memory
- Global memory access is slooooow

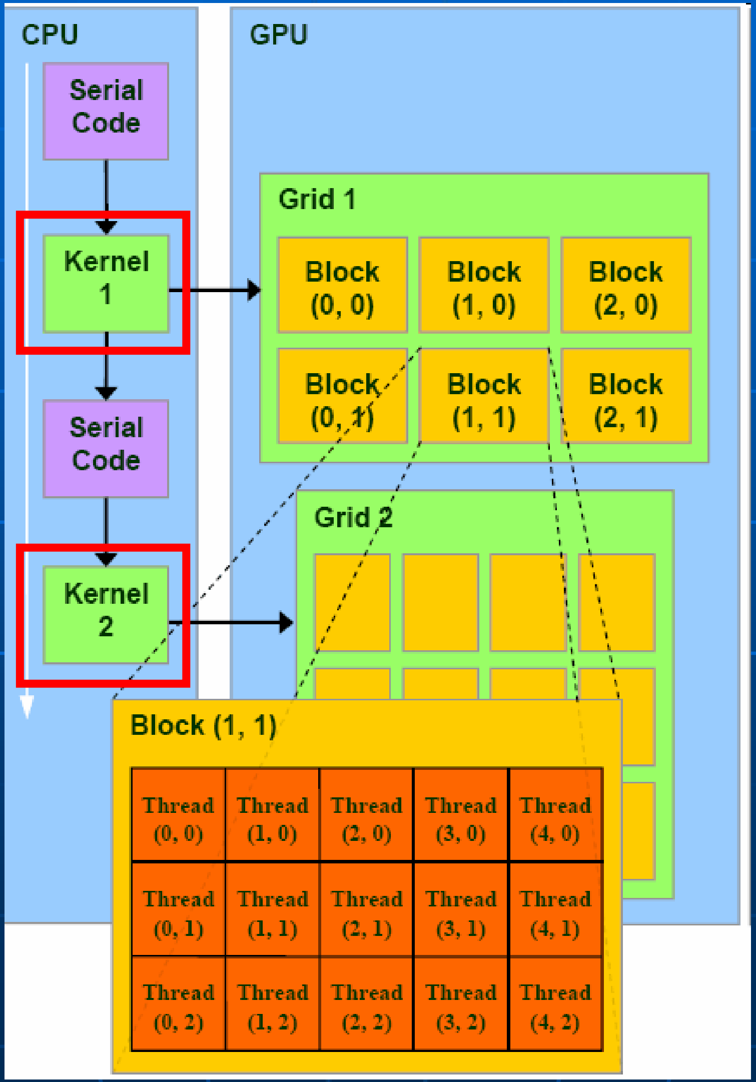
CUDA in Details

- Based on revision 1.0 of CUDA
- Geforce 8800 series and newer are CUDA 1.0 compliant
- First some terminology:
  - Kernel
  - Grid
  - Blocks
  - Warps
  - Threads

Kernels

- The general building block of GPGPU programming
- Used whenever a code section can be highly parallelized
- Executed on the GPU across multiple TPC (Thread Processing Clusters)
- A unified kernel is executed N times in parallel by N different CUDA threads on different input data

Kernels


CPU
Serial Code
Kernel 1
Serial Code
Kernel 2
GPU
Grid 1
Block (0, 0)
Block (1, 0)
Block (2, 0)
Block (0, 1)
Block (1, 1)
Block (2, 1)
Grid 2
Block (1, 1)
Thread (0, 0)
Thread (1, 0)
Thread (2, 0)
Thread (3, 0)
Thread (4, 0)
Thread (0, 1)
Thread (1, 1)
Thread (2, 1)
Thread (3, 1)
Thread (4, 1)
Thread (0, 2)
Thread (1, 2)
Thread (2, 2)
Thread (3, 2)
Thread (4, 2)

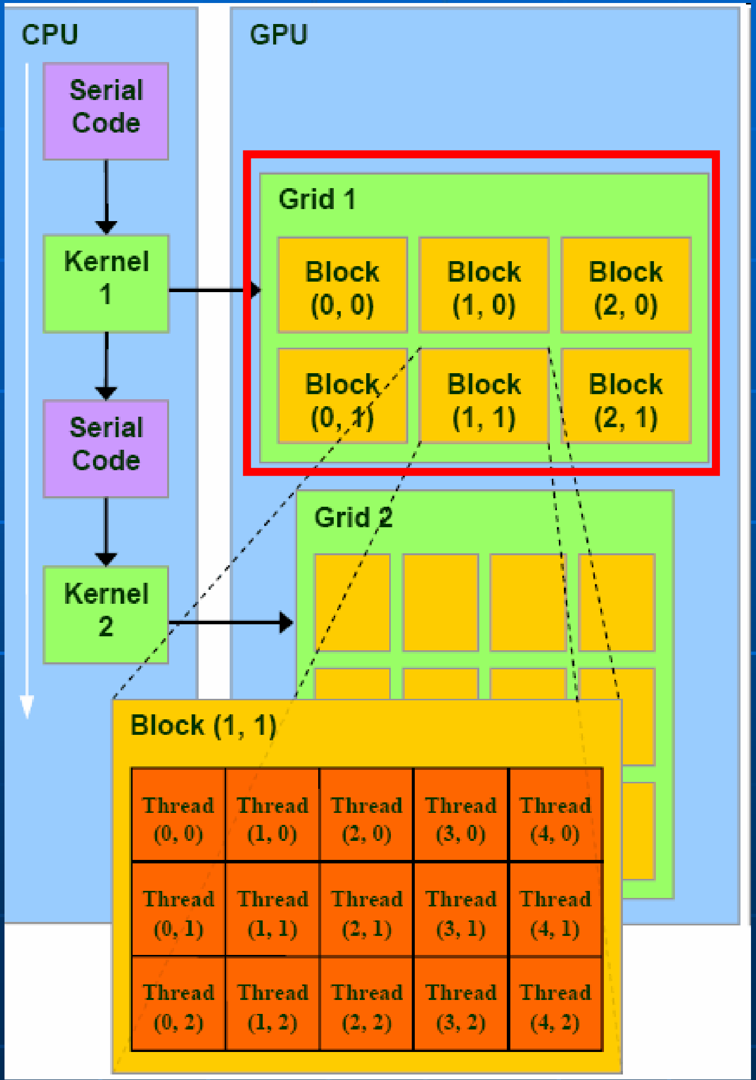
[11]

Grid

- If the number of threads needed by the kernel exceeds the limit of one thread block several thread blocks are collected in a grid
- Grids are up to 3-dimensional collections of thread blocks
- Maximum number of thread blocks per grid is $(2^8-1)^3$ = 281462092005375!!!

- The number of thread blocks in a grid is determined from the amount of data not the architecture of the GPU
- Performance should scale with new hardware

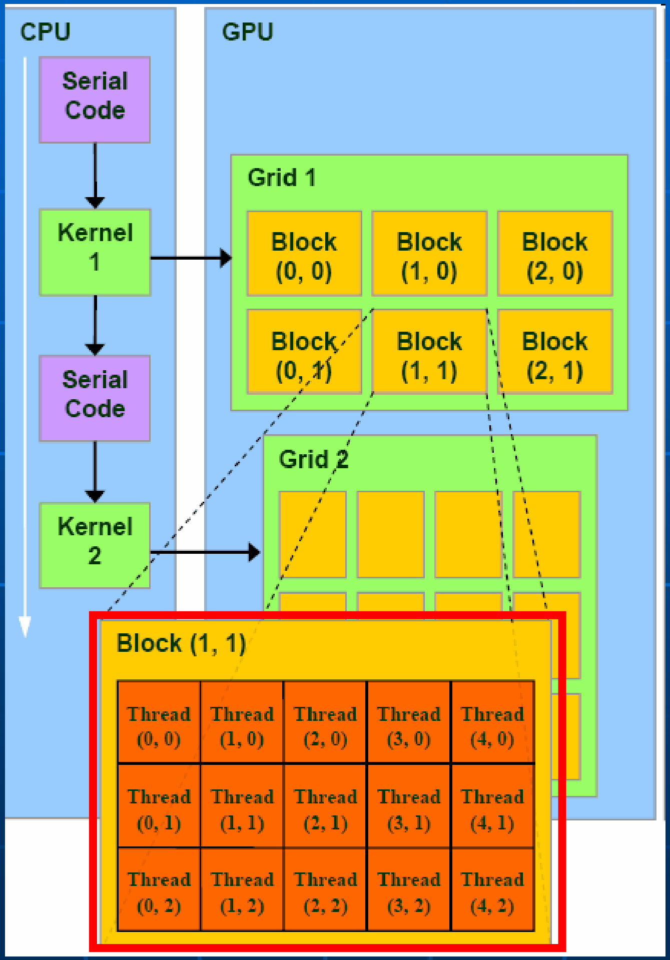
Grid


CPU
GPU
Serial Code
Kernel 1
Grid 1
Block (0, 0)
Block (1, 0)
Block (2, 0)
Block (0, 1)
Block (1, 1)
Block (2, 1)
Serial Code
Grid 2
Kernel 2
Block (1, 1)
Thread (0, 0)
Thread (1, 0)
Thread (2, 0)
Thread (3, 0)
Thread (4, 0)
Thread (0, 1)
Thread (1, 1)
Thread (2, 1)
Thread (3, 1)
Thread (4, 1)
Thread (0, 2)
Thread (1, 2)
Thread (2, 2)
Thread (3, 2)
Thread (4, 2)

Thread Blocks

- Is a collection of threads
- The maximum number of threads per block is 512
- Can be 3-dimensional but restricted to (x = 512, y = 512, z = 64)
- A thread block is executed by one streaming multiprocessor
- Threads within a block can share data
  - By synchronization
  - Shared local memory

Thread Blocks



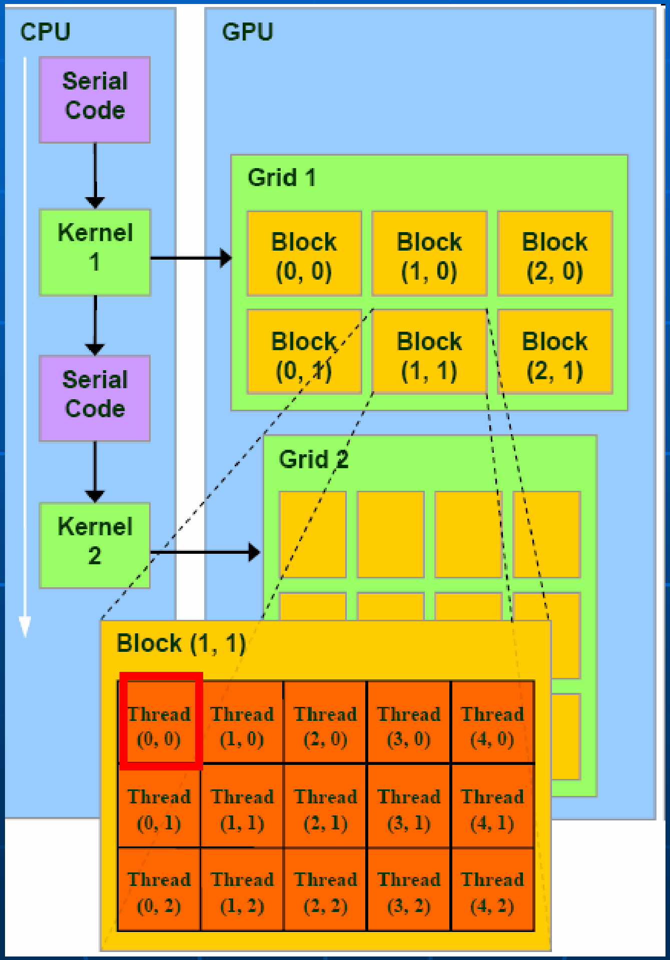
Warps

- A streaming multiprocessor consists of 8 streaming processors each capable of executing 1 thread at a time
- Warp is the process of scheduling threads for processing
- The warp size is 32, which imply that 32 threads are scheduled at once and executed within 4 clock cycles
- Warps are handled by the hardware scheduler so no worries ;)

Threads

- Different from CPU threads
- The smallest building blocks of GPGPU programming
- Executed on the streaming processors
- E.g. a multiplication of two matrix cells
- Each thread has a unique id
- The thread of a 2D (D_x,D_y) thread block at (x,y) has ID: $x + y*D_x$

Threads


CPU
GPU
Serial Code
Kernel 1
Serial Code
Kernel 2
Grid 1
Block (0, 0)
Block (1, 0)
Block (2, 0)
Block (0, 1)
Block (1, 1)
Block (2, 1)
Grid 2
Block (1, 1)
Thread (0, 0)
Thread (1, 0)
Thread (2, 0)
Thread (3, 0)
Thread (4, 0)
Thread (0, 1)
Thread (1, 1)
Thread (2, 1)
Thread (3, 1)
Thread (4, 1)
Thread (0, 2)
Thread (1, 2)
Thread (2, 2)
Thread (3, 2)
Thread (4, 2)

GPU Programming Flow Control

- Avoid when possible
- All threads of a block have to execute all brances but will only output from those they are supposed to

Instruction	If/endif	If/else/endif	Call	Return	Loop/end loop
Cost (Cycles)	4	6	2	2	4

From GPU Gems 2: Chapter 30

Summary of CUDA

- The smallest element of the kernel is the thread
- Threads are collected in thread blocks
- For optimal utilization of the multiprocessors divide the task in to a large number of thread blocks and organize them in a grid
- Branching is costly
- The local memory resources are limited
- Avoid global memory access

Part 4: Using Cuda

- Part 1: Introduction
- Part 2: GPGPU Environments
- Part 3: GPGPU Programming
- **Part 4: Using CUDA**

CUDA Development Environment

- Hardware Requirements:
  - NVIDIA Graphics Card 8800 series or newer
  - 8X00 series: CUDA 1.0
  - 9X00 series: CUDA 1.1
  - 2X0 series: CUDA 1.3
- Software Requirements
  - Windows
  - Linux
  - Mac OS X (Beta)

CUDA Development Environment

- Windows Requirements
- Three Versions 1.0, 1.1, 2.0
  - Visual Studio 7 or 8 (Yet no support for 9/2008)
  - CUDA Capable Graphic Card Drivers
    - All drivers from 169.21 (1.1) and 178.08 (2.0)
  - CUDA Toolkit
  - CUDA SDK

The Missing Link

- The SDK contains a simple CUDA application template to get you started
- The CUDA Programming Guide contains a simple Matrix multiplication example
- Performance measurements should be done with high precision timers, check: http://forums.nvidia.com/index.php?showtopic=73594

What to Remember

- When to consider GPGPU
  - High arithmetic intensity
  - Need for a lot of low cost computation power
- Use of GPGPU requires special program design
- Libraries for common functionalities
  - BLAS, FFT, MATLAB plug-in (Jacket)
- Most promising APIs:
  - CUDA
  - OpenCL

References

- [1] AMD FireStream 9170, http://ati.amd.com/technology/streamcomputing/product_firestream_9170.html
- [2] Multiple Pixel Shader Precision Modes, http://www.beyond3d.com/content/interviews/23/
- [3] Mark Harris, GPGPU Lessons Learned, GameDevelopers Conference Presentation
- [4] Magnus Ekman et al., An In-Depth Look at Computer Performance Growth
- [5] Tim Sweeney, The Next Mainstream Programming Language
- [6] AMD Stream Computing FAQ, http://forums.amd.com/devforum/messageview.cfm?catid=328&threadid=95060
- [7] Aaftab Munshi, Presentation at SIGGRAPH 2008, OpenCL Parallel Computing on the GPU and CPU
- [8] John Owens, University of California Davis, What's New With GPGPU?
- [9] David Luebke, nVidia Corp, CUDA: SCALABLE PARALLEL PROGRAMMING FOR HIGH-PERFORMANCE SCIENTIFIC COMPUTING
- [10] NVIDIA CUDA Programming Guide 1.1
- [11] David Kanter, NVIDIA's GT200: Inside a Parallel Processor, http://www.realworldtech.com/page.cfm?ArticleID=RWT090808195242&p=2