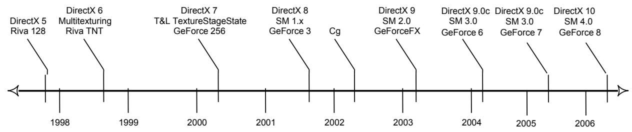
INF5062 – GPU & CUDA

Håkon Kvale Stensland
Simula Research Laboratory

PC Graphics Timeline

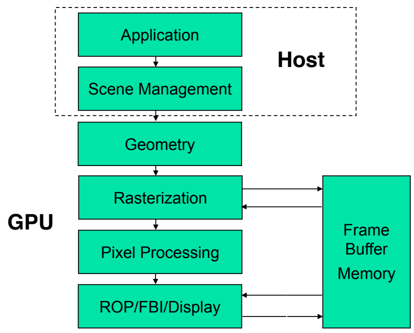
- Challenges:
  - Render infinitely complex scenes
  - And extremely high resolution
  - In 1/60th of one second (60 frames per second)
- Graphics hardware has evolved from a simple hardwired pipeline to a highly programmable multiword processor



(Some) 3D Buzzwords!!

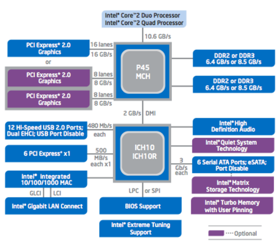
- GPU: Graphics Processing Unit: **A graphics chip with integrated programmable geometry and pixel processing**
- Fill Rate: **How fast the GPU can generate pixels, often a strong predictor for application frame rate**
- Shader: **A set of software instructions, which is used by the graphic resources primarily to perform rendering effects.**
- API: Application Programming Interface – **The standardized layer of software that allows applications (like games) to talk to other software or hardware to get services or functionality from them – such as allowing a game to talk to a graphics processor**
- DirectX: **Microsoft's API for media functionality**
- Direct3D: **Portion of the DirectX API suite that handles the interface to graphics processors**
- OpenGL: **An open standard API for graphics functionality. Available across platforms. Popular with workstation applications**

Basic 3D Graphics Pipeline



Graphics in the PC Architecture

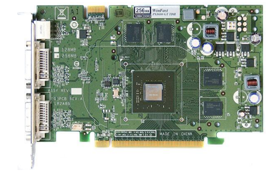
- FSB connection between processor and Northbridge (P45)
  - Memory Control Hub
- Northbridge handles PCI Express 2.0 to GPU and DRAM.
  - PCIe 2 x16 bandwidth at 16 GB/s (8 GB in each direction)
- Southbridge (ICH10) handles all other peripherals

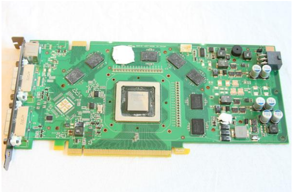


High-end Hardware

- nVidia GeForce GTX 280
- Based on the latest generation GPU, codenamed GT200
- 1400 million transistors
- 240 Processing cores (SP) at 1296MHz
- 1024 MB Memory with 141.7GB/sec of bandwidth.
- 933 GFLOPS of computing power

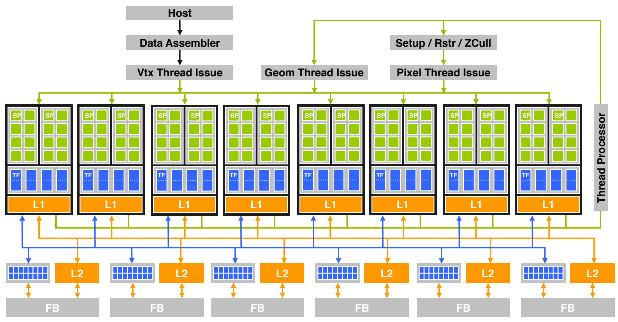
Lab Hardware



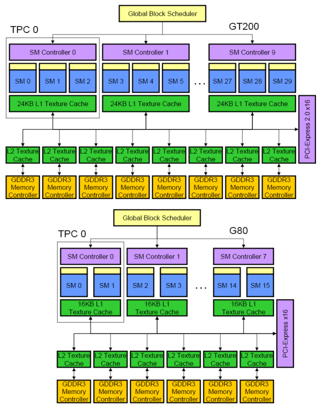


- nVidia GeForce 8600GT
- Based on the G84 chip
  - 289 million transistors
  - 32 Processing cores (SP) at 1190MHz
  - 512/256 MB Memory with 22.4GB/sec bandwidth
- nVidia GeForce 8800GT
- Based on the G92 chip
  - 754 million transistors
  - 112 Processing cores (SP) at 1500MHz
  - 256 MB Memory with 57.6GB/sec bandwidth

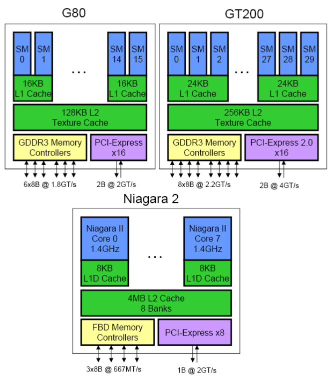
GeForce G80 Architecture



nVIDIA G80 vs. GT92/GT200 Architecture

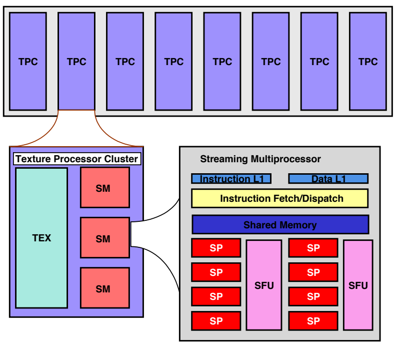


Compared with a multicore RISC-CPU



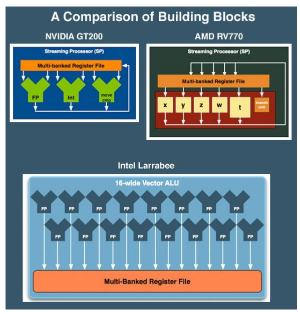
TPC… SM… SP… Some more details…

- TPC
  - Texture Processing Cluster
- SM
  - Streaming Multiprocessor
  - In CUDA: Multiprocessor, and fundamental unit for a thread block
- TEX
  - Texture Unit
- SP
  - Stream Processor
  - Scalar ALU for single CUDA thread
- SFU
  - Super Function Unit



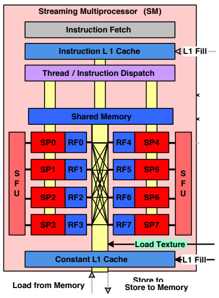
SP: The basic processing block

- The nVIDIA Approach:
  - A Stream Processor works on a single operation
- AMD GPU's work on up to five operations, and Intel's Larrabee will work on up to 16
- Now, let's take a step back for a closer look!



Streaming Multiprocessor (SM)

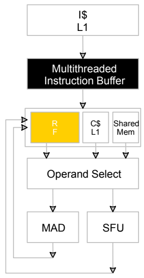
- Streaming Multiprocessor (SM)
  - 8 Streaming Processors (SP)
  - 2 Super Function Units (SFU)
- Multi-threaded instruction dispatch
  - 1 to 768 threads active
  - Try to Cover latency of texture/memory loads
- Local register file (RF)
- 16 KB shared memory
- DRAM texture and memory access



Foils adapted from nVIDIA

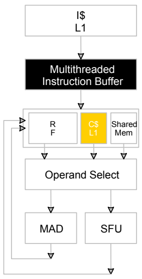
SM Register File

- Register File (RF)
  - 32 KB
  - Provides 4 operands/clock
- TEX pipe can also read/write Register File
  - 3 SMs share 1 TEX
- Load/Store pipe can also read/write Register File



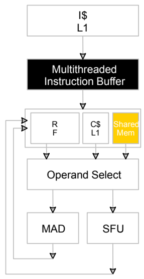
Constants

- Immediate address constants
- Indexed address constants
- Constants stored in memory, and cached on chip
  - L1 cache is per Streaming Multiprocessor



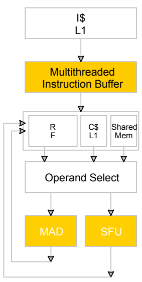
Shared Memory

- Each Stream Multiprocessor has 16KB of Shared Memory
  - 16 banks of 32bit words
- CUDA uses Shared Memory as shared storage visible to all threads in a thread block
  - Read and Write access



Execution Pipes

- Scalar MAD pipe
  - Float Multiply, Add, etc.
  - Integer ops,
  - Conversions
  - Only one instruction per clock
- Scalar SFU pipe
  - Special functions like Sin, Cos, Log, etc.
    - Only one operation per four clocks
- TEX pipe (external to SM, shared by all SM's in a TPC)
- Load/Store pipe
  - CUDA has both global and local memory access through Load/Store



GPGPU

Foils adapted from nVIDIA

What is really GPGPU?

- General Purpose computation using GPU in other applications than 3D graphics
  - GPU can accelerate parts of an application
- Parallel data algorithms using the GPUs properties
  - Large data arrays, streaming throughput
  - Fine-grain SIMD parallelism
  - Fast floating point (FP) operations
- Applications for GPGPU
  - Game effects (physics) nVIDIA PhysX
  - Image processing (Photoshop CS4)
  - Video Encoding/Transcoding (Elemental RapidHD)
  - Distributed processing (Stanford Folding@Home)
  - RAID6, AES, MatLab, etc.

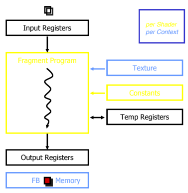
Performance?

- Let's look at Standfords Folding@Home....
- Distributed Computing
- Folding@Home client is available for CUDA
  - Windows
  - All CUDA-enabled GPUs
- Performance GFLOPS:
  - Cell: 28
  - nVIDIA GPU: 110
  - ATI GPU: 109

OS Type	Current TFLOPS*	Active CPUs	Total CPUs
Windows	210	220936	2205693
Mac OS X/PowerPC	6	7316	120911
Mac OS X/Intel	24	7860	65496
Linux	55	32245	335484
ATI GPU	507	4612	12284
NVIDIA GPU	1614	14675	35626
PLAYSTATION®3	1679	59525	631489
Total	4095	347169	3406983

Previous GPGPU use, and limitations

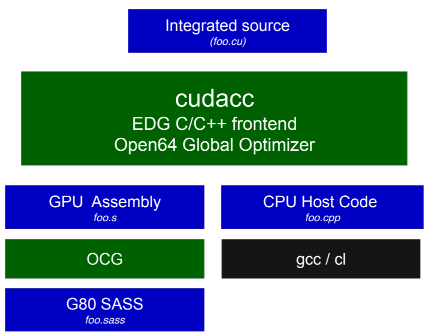
- Working with a Graphics API
  - Special cases with an API like Microsoft Direct3D or OpenGL
- Addressing modes
  - Limited by texture size
- Shader capabilities
  - Limited outputs of the available shader programs
- Instruction sets
  - No integer or bit operations
- Communication is limited
  - Between pixels



nVIDIA CUDA

- "Compute Unified Device Architecture"
- General purpose programming model
  - User starts several batches of threads on a GPU
  - GPU is in this case a dedicated super-threaded, massively data parallel co-processor
- Software Stack
  - Graphics driver, language compilers (Toolkit), and tools (SDK)
- Graphics driver loads programs into GPU
  - All drivers from nVIDIA now support CUDA.
  - Interface is designed for computing (no graphics ☺)
  - "Guaranteed" maximum download & readback speeds
  - Explicit GPU memory management

"Extended" C



Outline

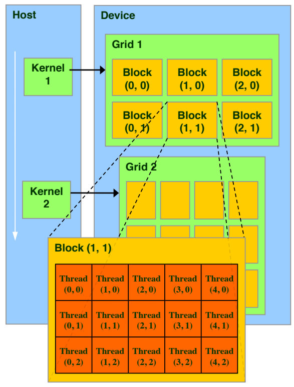
- The CUDA Programming Model
  - Basic concepts and data types
- The CUDA Application Programming Interface
  - Basic functionality
- More advanced CUDA Programming
  - 24th of October

The CUDA Programming Model

- The GPU is viewed as a compute device that:
  - Is a coprocessor to the CPU, referred to as the host
  - Has its own DRAM called device memory
  - Runs **many** threads in parallel
- Data-parallel parts of an application are executed on the device as kernels, which run in parallel on many threads
- Differences between GPU and CPU threads
  - GPU threads are extremely lightweight
    - Very little creation overhead
  - GPU needs 1000s of threads for full efficiency
    - Multi-core CPU needs only a few

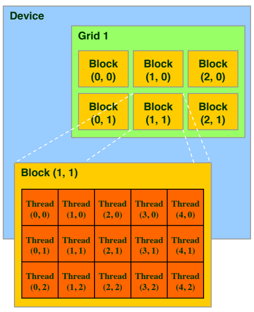
Thread Batching: Grids and Blocks

- A kernel is executed as a grid of thread blocks
  - All threads share data memory space
- A thread block is a batch of threads that can cooperate with each other by:
  - Synchronizing their execution
    - Non synchronous execution is very bad for performance!
  - Efficiently sharing data through a low latency shared memory
- Two threads from two different blocks cannot cooperate



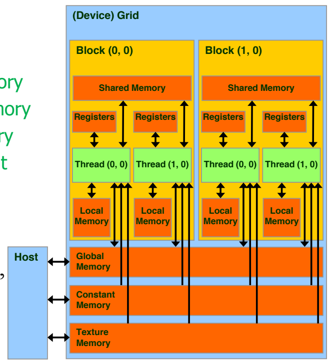
Block and Thread IDs

- Threads and blocks have IDs
  - Each thread can decide what data to work on
  - Block ID: 1D or 2D
  - Thread ID: 1D, 2D, or 3D
- Simplifies memory addressing when processing multidimensional data
  - Image and video processing (e.g. MJPEG…)



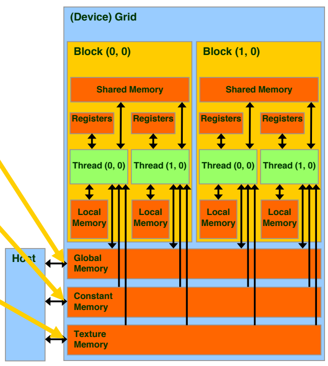
CUDA Device Memory Space Overview

- Each thread can:
  - R/W per-thread registers
  - R/W per-thread local memory
  - R/W per-block shared memory
  - R/W per-grid global memory
  - Read only per-grid constant memory
  - Read only per-grid texture memory
- The host can R/W global, constant, and texture memories



Global, Constant, and Texture Memories

- Global memory:
  - Main means of communicating R/W Data between host and device
  - Contents visible to all threads
- Texture and Constant Memories:
  - Constants initialized by host
  - Contents visible to all threads



Terminology Recap

- device = GPU = Set of multiprocessors
- Multiprocessor = Set of processors & shared memory
- Kernel = Program running on the GPU
- Grid = Array of thread blocks that execute a kernel
- Thread block = Group of SIMD threads that execute a kernel and can communicate via shared memory

Memory	Location	Cached	Access	Who
Local	Off-chip	No	Read/write	One thread
Shared	On-chip	N/A - resident	Read/write	All threads in a block
Global	Off-chip	No	Read/write	All threads + host
Constant	Off-chip	Yes	Read	All threads + host
Texture	Off-chip	Yes	Read	All threads + host

Access Times

- Register – Dedicated HW – Single cycle
- Shared Memory – Dedicated HW – Single cycle
- Local Memory – DRAM, no cache – "Slow"
- Global Memory – DRAM, no cache – "Slow"
- Constant Memory – DRAM, cached, 1…10s…100s of cycles, depending on cache locality
- Texture Memory – DRAM, cached, 1…10s…100s of cycles, depending on cache locality

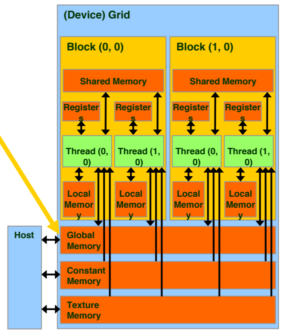
CUDA – API

CUDA Highlights

- The API is an extension to the ANSI C programming language
  - ➡ Low learning curve than OpenGL/Direct3D
- The hardware is designed to enable lightweight runtime and driver
  - ➡ High performance

CUDA Device Memory Allocation

- cudaMalloc()
  - Allocates object in the device **Global Memory**
  - Requires two parameters
    - **Address of a pointe**r to the allocated object
    - **Size of** allocated object
- cudaFree()
  - Frees object from device Global Memory
    - Pointer to the object



CUDA Device Memory Allocation

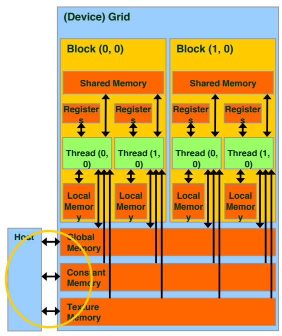
- Code example:
  - Allocate a 64 * 64 single precision float array
  - Attach the allocated storage to Md.elements
  - "d" is often used to indicate a device data structure

```
BLOCK_SIZE = 64;
Matrix Md
int size = BLOCK_SIZE * BLOCK_SIZE * sizeof(float);

cudaMalloc((void**)&Md.elements, size);
cudaFree(Md.elements);
```


CUDA Host-Device Data Transfer

- cudaMemcpy()
  - memory data transfer
  - Requires four parameters
    - Pointer to source
    - Pointer to destination
    - Number of bytes copied
    - Type of transfer
      - Host to Host
      - Host to Device
      - Device to Host
      - Device to Device
- Asynchronous in CUDA 1.3



Memory Management

- Device memory allocation
  - `cudaMalloc(), cudaFree()`
- Memory copy from host to device, device to host, device to device
  - `cudaMemcpy(), cudaMemcpy2D(), cudaMemcpyToSymbol(), cudaMemcpyFromSymbol()`
- Memory addressing
  - `cudaGetSymbolAddress()`

CUDA Host-Device Data Transfer

- Code example:
  - Transfer a 64 * 64 single precision float array
  - M is in host memory and Md is in device memory
  - cudaMemcpyHostToDevice and cudaMemcpyDeviceToHost are symbolic constants

```
cudaMemcpy(Md.elements, M.elements, size,
   cudaMemcpyHostToDevice);

cudaMemcpy(M.elements, Md.elements, size,
   cudaMemcpyDeviceToHost);
```


CUDA Function Declarations

	Executed on the:	Only callable from the:
`__device__ float DeviceFunc()`	device	device
`__global__ void KernelFunc()`	device	host
`__host__ float HostFunc()`	host	host

- `__global__` defines a kernel function
  - Must return `void`
- `__device__` and `__host__` can be used together

CUDA Function Declarations

- `__device__` functions cannot have their address taken
- Limitations for functions executed on the device:
  - No recursion
  - No static variable declarations inside the function
  - No variable number of arguments

Calling a Kernel Function

- A kernel function must be called with an execution configuration:

```
__global__ void KernelFunc(...);
dim3   DimGrid(100, 50);     // 5000 thread blocks
dim3   DimBlock(4, 8, 8);    // 256 threads per block
size_t SharedMemBytes = 64; // 64 bytes of shared memory
KernelFunc <<< DimGrid, DimBlock, SharedMemBytes >>>(...);
```

- Any call to a kernel function is asynchronous from CUDA 1.0 on, explicit synch needed for blocking

Some Information on Toolkit

Compilation

- Any source file containing CUDA language extensions must be compiled with nvcc
- nvcc is a compiler driver
  - Works by invoking all the necessary tools and compilers like cudacc, g++, etc.
- nvcc can output:
  - Either C code
    - That must then be compiled with the rest of the application using another tool
  - Or object code directly

Linking

- Any executable with CUDA code requires two dynamic libraries:
  - The CUDA runtime library (`cudart`)
  - The CUDA core library (`cuda`)

Debugging Using Device Emulation

- An executable compiled in device emulation mode (`nvcc -deviceemu`) runs completely on the host using the CUDA runtime
  - No need of any device and CUDA driver
  - Each device thread is emulated with a host thread
- When running in device emulation mode, one can:
  - Use host native debug support (breakpoints, inspection, etc.)
  - Access any device-specific data from host code and vice-versa
  - Call any host function from device code (e.g. `printf`) and vice-versa
  - Detect deadlock situations caused by improper usage of `__syncthreads`

Lab Setup

- frogner.ndlab.net
  - GeForce 8600GT 256MB (G84)
  - 4 Multiprocessors, 32 Cores
- majorstuen.ndlab.net
  - GeForce 8600GT 512MB (G84)
  - 4 Multiprocessors, 32 Cores
- uranienborg.ndlab.net
  - GeForce 8600GT 512MB (G84)
  - 4 Multiprocessors, 32 Cores
- montebello.ndlab.net
  - GeForce 8800GT 256MB (G92)
  - 14 Multiprocessors, 112 Cores

Before you start…

- Four lines have to be added to your group users .bash_profile file

```
PATH=$PATH:/usr/local/cuda/bin
LD_LIBRARY_PATH=$LD_LIBRARY_PATH:/usr/local/cuda/lib

export PATH
export LD_LIBRARY_PATH
```

- When you use a machine, remember to update the message of the day! (etc/motd)

Compile and test SDK

- SDK is downloaded in the **/opt/** folder
- Copy and build in your users home directory
- Test machines uses Fedora Core 9, with gcc 4.3, SDK is for Fedora Core 8, and some fixing is needed to compile…

Add the following **#include** in these files:

common/src/paramgl.cpp: <cstring>
projects/cppIntegration/main.cpp: <cstdlib>
common/inc/exception.h: <cstdlib>
common/inc/cutil.h: <cstring>
common/inc/cmd_arg_reader.h: <typeinfo>

Some usefull resources

nVIDIA CUDA Programming Guide 2.0

http://developer.download.nvidia.com/compute/cuda/2_0/docs/NVIDIA_CUDA_Programming_Guide_2.0.pdf

nVIDIA CUDA Refference Manual 2.0

http://developer.download.nvidia.com/compute/cuda/2_0/docs/CudaReferenceManual_2.0.pdf

nVISION08: Getting Started with CUDA

http://www.nvidia.com/content/cudazone/download/Getting_Started_w_CUDA_Training_NVISION08.pdf