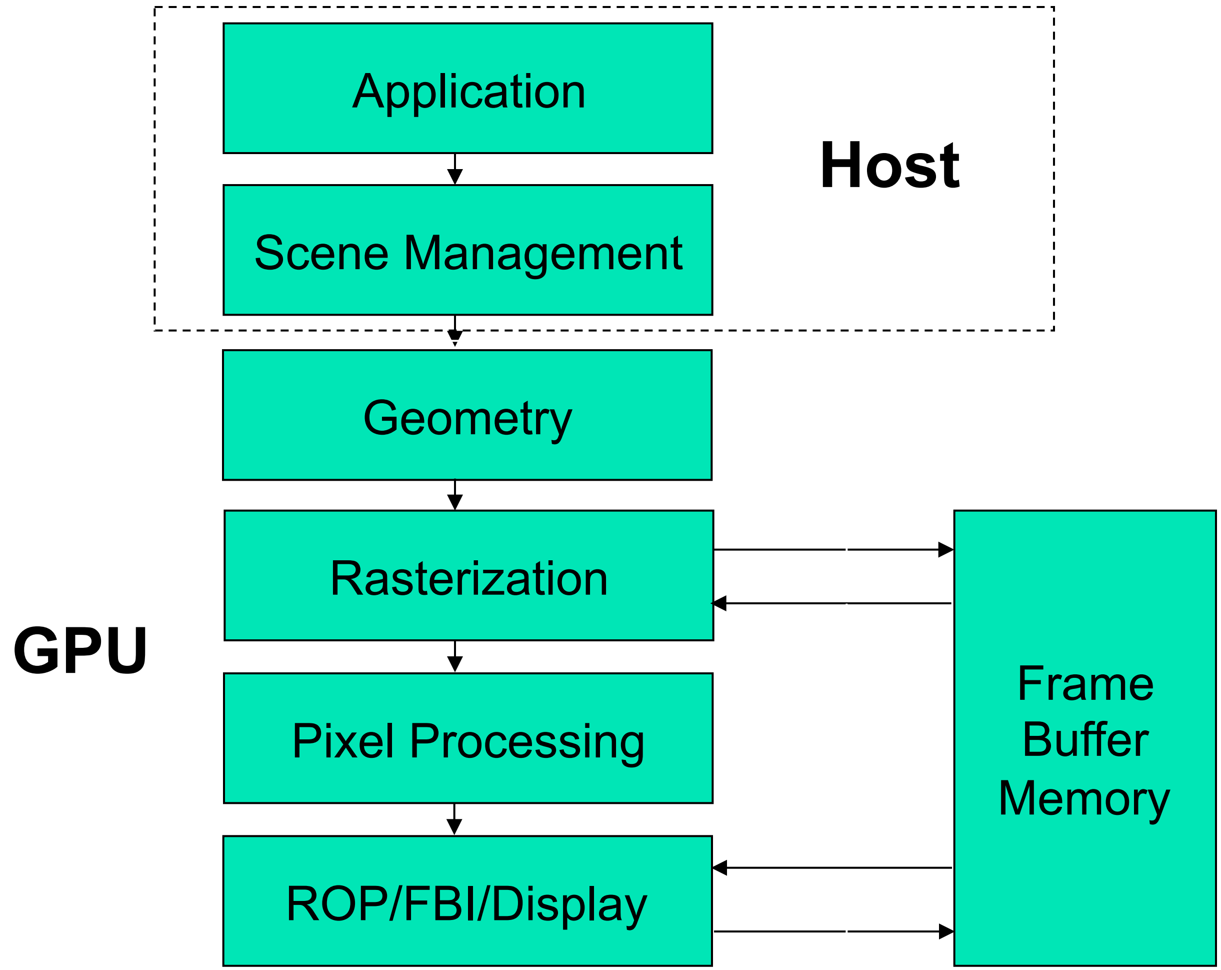
INF5063 – GPU & CUDA

Håkon Kvale Stensland
iAD-lab, Department for Informatics

INFORMATION ACCESS DISRUPTIONS

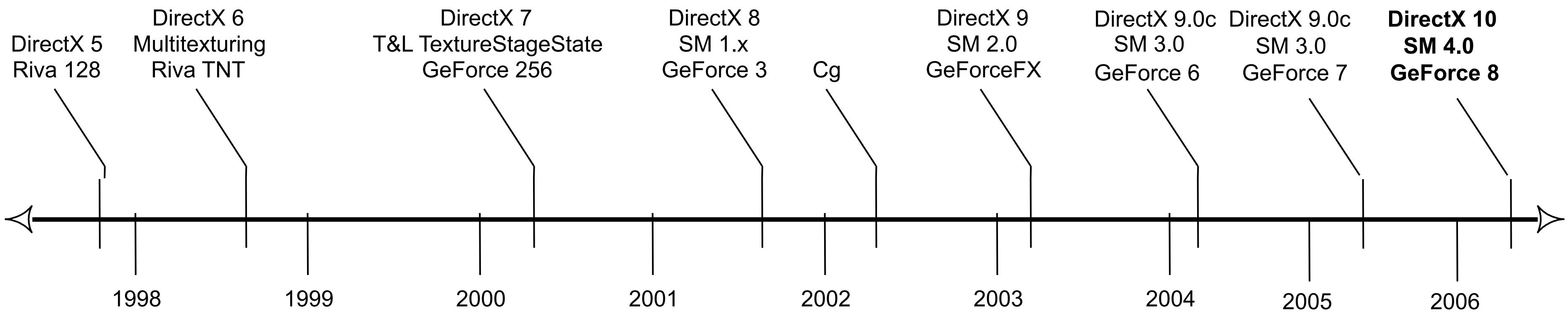
UiO : University of Oslo

Basic 3D Graphics Pipeline



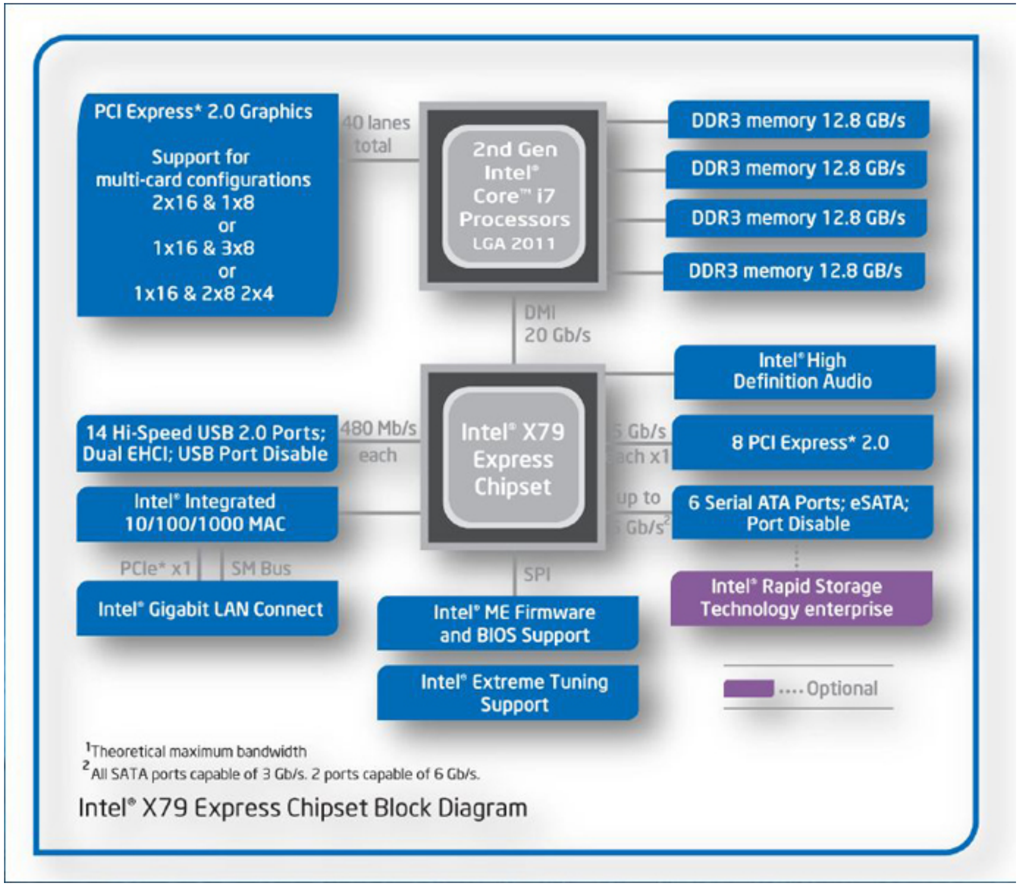
PC Graphics Timeline

- 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



Graphics in the PC Architecture

- DMI (Direct Media Interface) between processor and chipset
  - Memory Control now integrated in CPU
- The old "Northbridge" integrated onto CPU
  - PCI Express 3.0 x16 bandwidth at 32 GB/s (16 GB in each direction)
- Southbridge (X79) handles all other peripherals

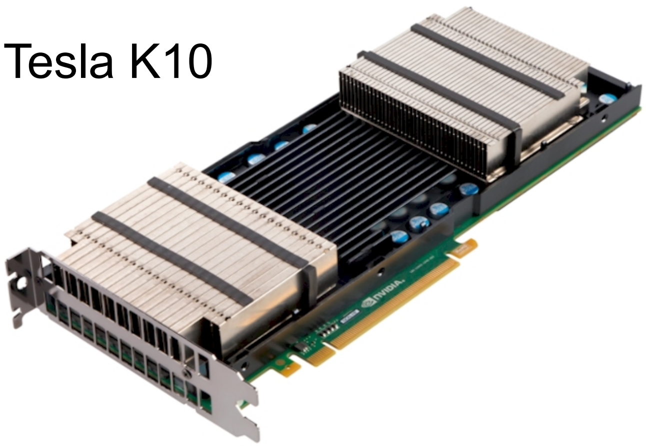


GPUs not always for Graphics

GeForce GTX 690

- GPUs are now common in HPC
- Largest supercomputer in November 2012 will be the **Titan** at Oak Ridge National Laboratory
  - 18688 16-core Opteron processors
  - 16688 Nvidia Kepler GPU's
  - Target: 20+ petaflops

Tesla K10



- Before: Dedicated compute card released after grapics model
- Now: Nvidia's high-end Kepler GPU is currently only produced as compute product

High-end Hardware

- nVIDIA Kepler Architecture
- The latest generation GPU, codenamed GK110
- 7,1 **billion** transistors
- 2688 Processing cores (SP)
  - IEEE 754-2008 Capable
  - Shared coherent L2 cache
  - Full C++ Support
  - Up to 32 concurrent kernels
  - 6 GB memory with ECC
  - Supports GPU virtualization

Lab Hardware #1

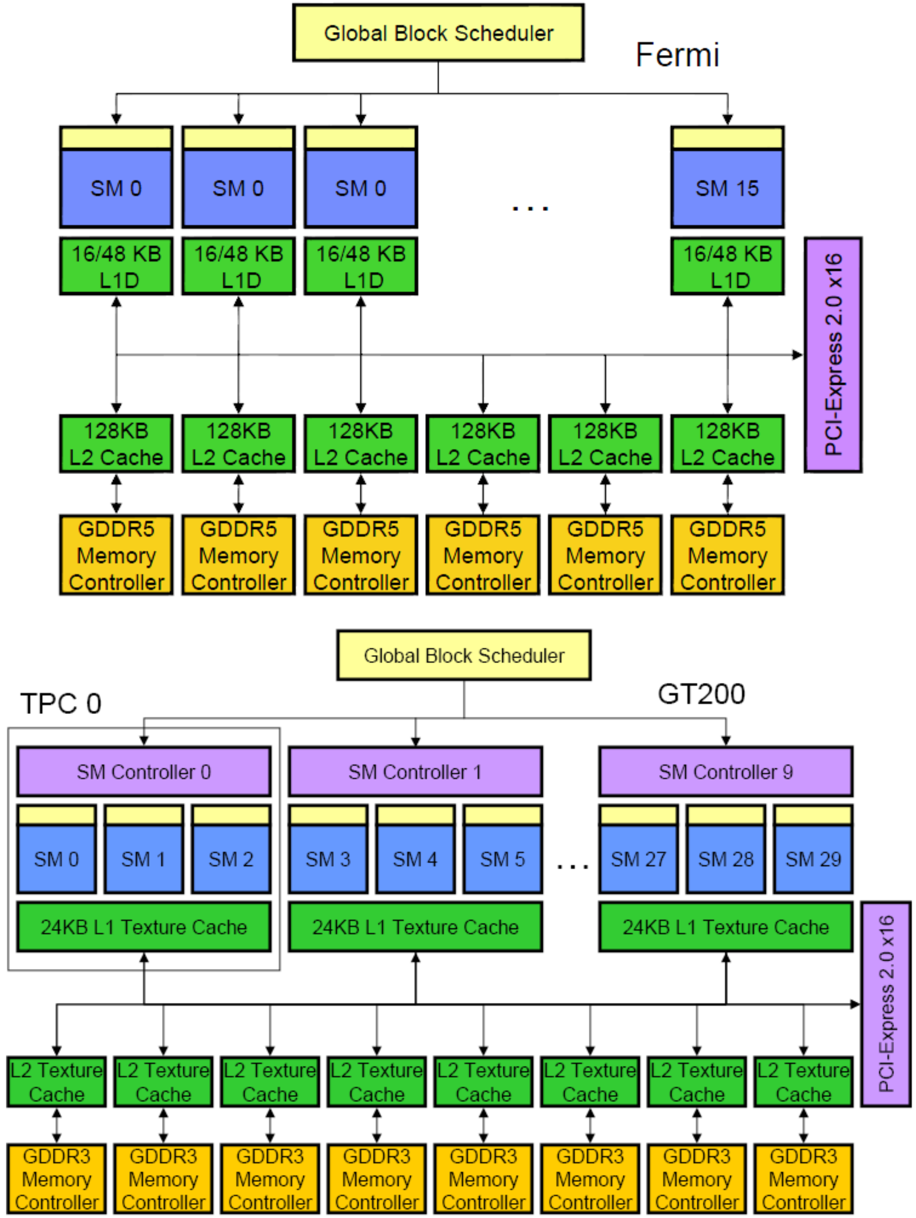
- **nVidia Quadro 600**
  - **GPU-5, GPU-6, GPU7, GPU-8**
  - **Fermi Architecture**
- Based on the GF108(GL) chip
  - 585 million transistors
  - 96 Processing cores (CC) at 1280MHz
  - 1024 MB Memory with 25,6 GB/sec bandwidth
  - Compute version 2.1

Lab Hardware #2

- **nVidia GeForce GTX 650**
  - **Clinton, Bush, Kennedy**
  - **Kepler Architecture**
- *Based on the GK107 chip*
  - 1300 million transistors
  - 384 Processing cores (SP) at 1058 MHz
  - 1024 MB Memory with 80 GB/sec bandwidth
  - Compute version 3.0

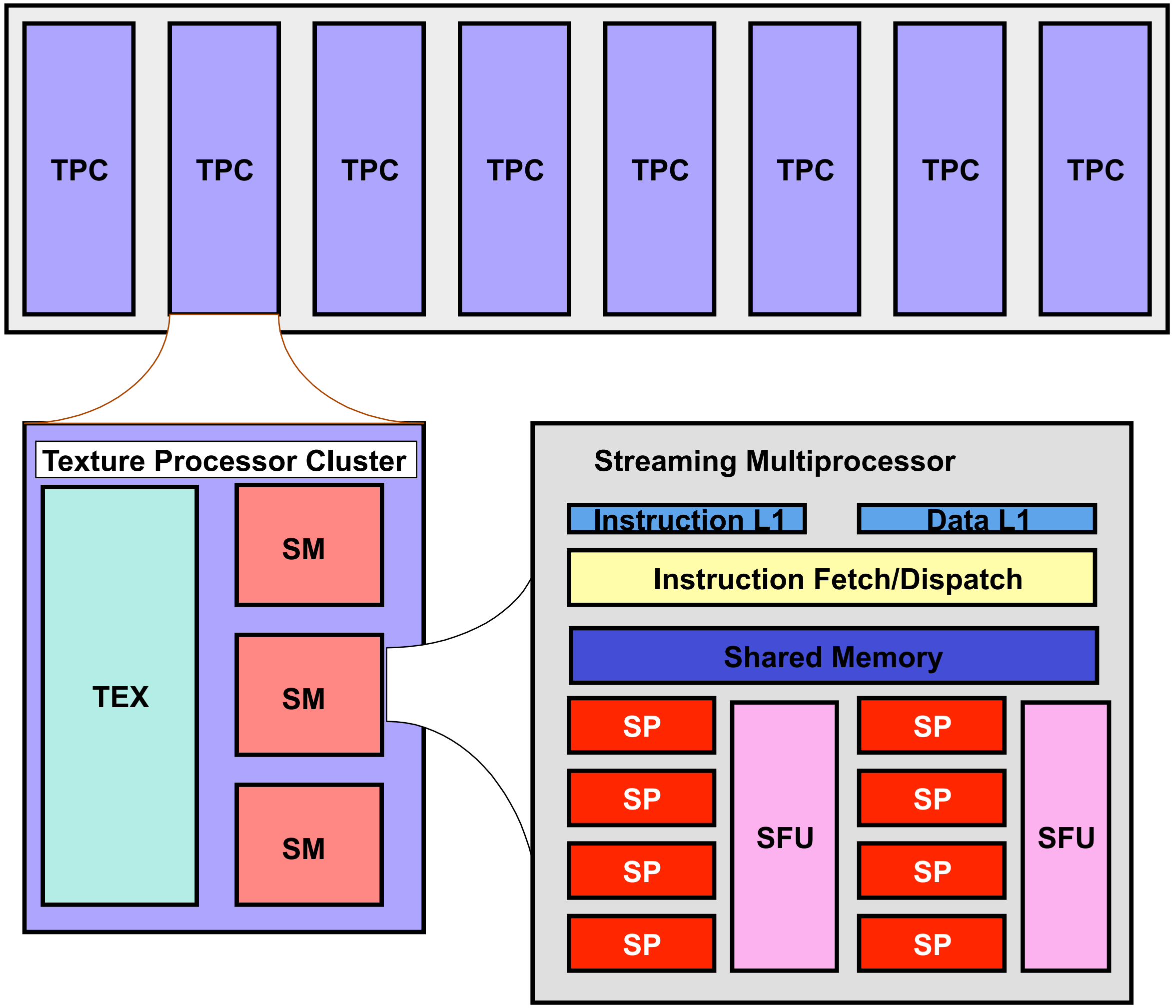
GeForce GK110 Architecture

nVIDIA GF100 vs. GT200 Architecture



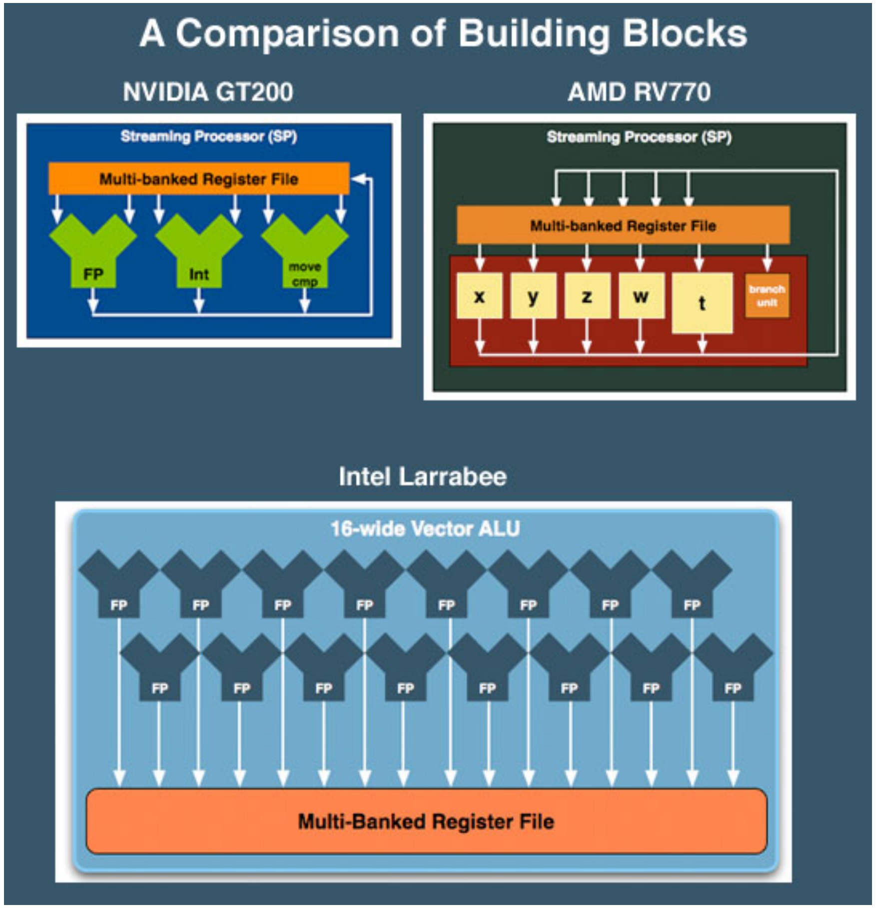
TPC… SM… SP… Some more details…

- 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



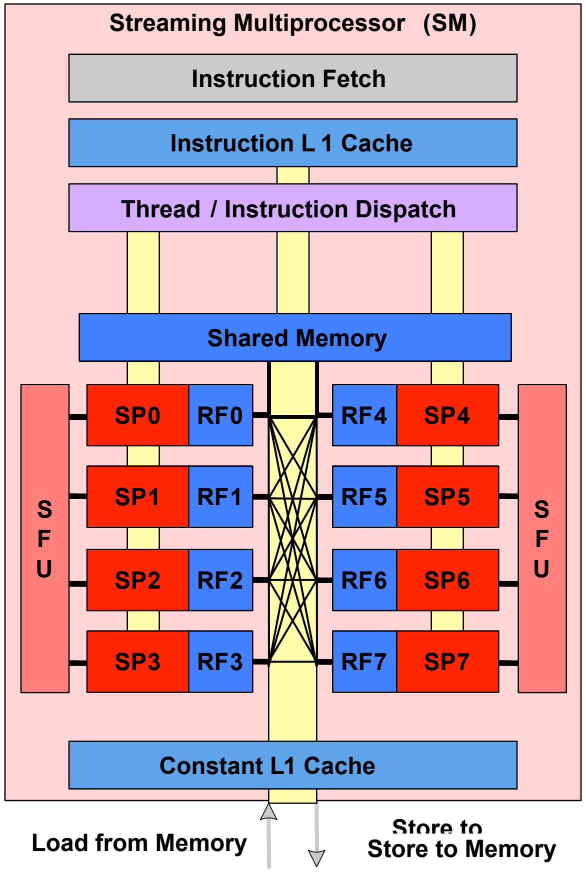
SP: The basic processing block

- The nVIDIA Approach:
  - A Stream Processor works on a single operation
- AMD GPU's work on up to five or four operations, new architecture in works.
- Now, let's take a step back for a closer look!



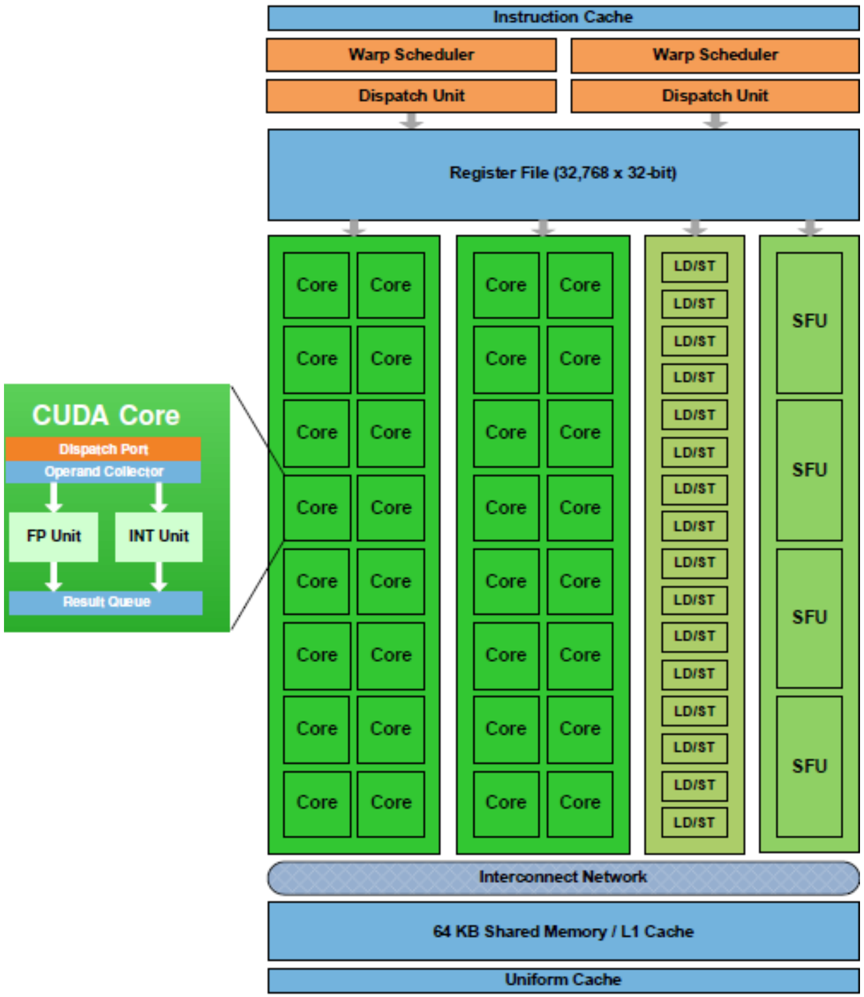
Streaming Multiprocessor (SM) – 1.0

- Streaming Multiprocessor (SM)
  - 8 Streaming Processors (SP)
  - 2 Super Function Units (SFU)
- Multi-threaded instruction dispatch
  - 1 to 1024 threads active
  - Try to Cover latency of texture/ memory loads
- Local register file (RF)
- 16 KB shared memory
- DRAM texture and memory access
- 2 operations per cycle
- **GeForce 8800 GTX**



Streaming Multiprocessor (SM) – 2.0

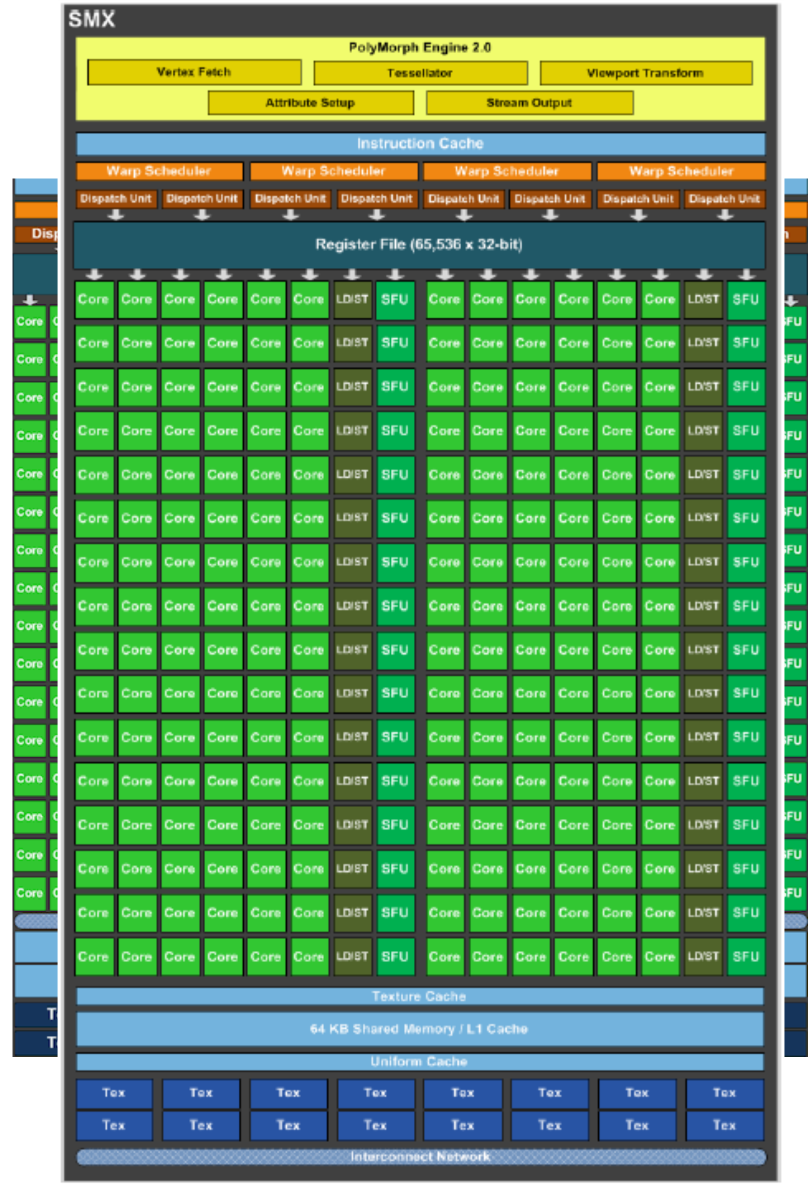
- Streaming Multiprocessor (SM) on the Fermi Architecture
  - 32 CUDA Cores (CC)
  - 4 Super Function Units (SFU)
- Dual schedulers and dispatch units
  - 1 to 1536 threads active
  - Try to optimize register usage vs. number of active threads
- Local register (32k)
- 64 KB shared memory
- DRAM texture and memory access
- 2 operations per cycle
- **GeForce GTX 480**



Fermi Streaming Multiprocessor (SM)

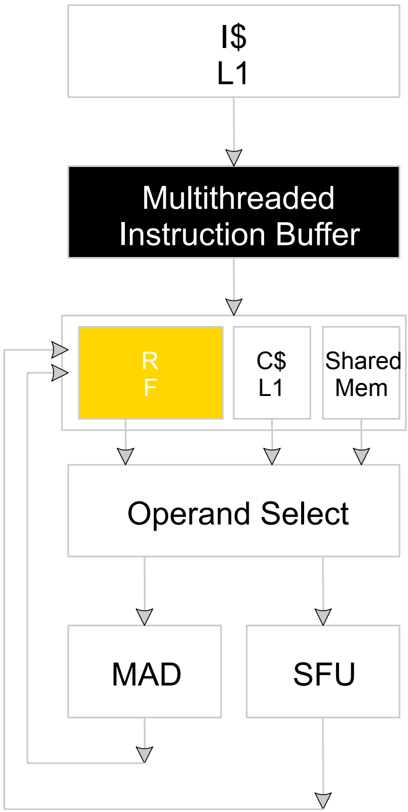
Streaming Multiprocessor (SMX) – 3.0

- Streaming Multiprocessor (SMX) on Kepler
  - 192 CUDA Cores (Core)
  - 64 DP CUDA Cores (DP Core)
  - 32 Super Function Units (SFU)
- Four schedule and dispatch units
  - 1 to 2048 active threads
  - Software controlled scheduling
- Local register (64k)
- 64 KB shared memory
- 1 operation per cycle
- **GeForce GTX 680**



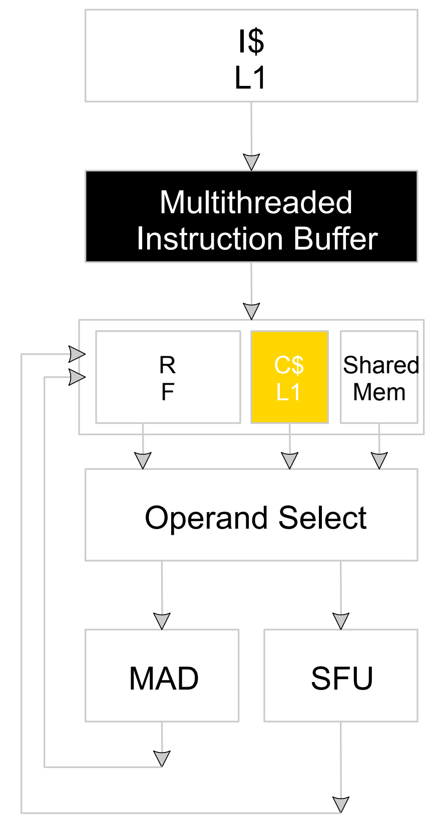
SM Register File

- 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



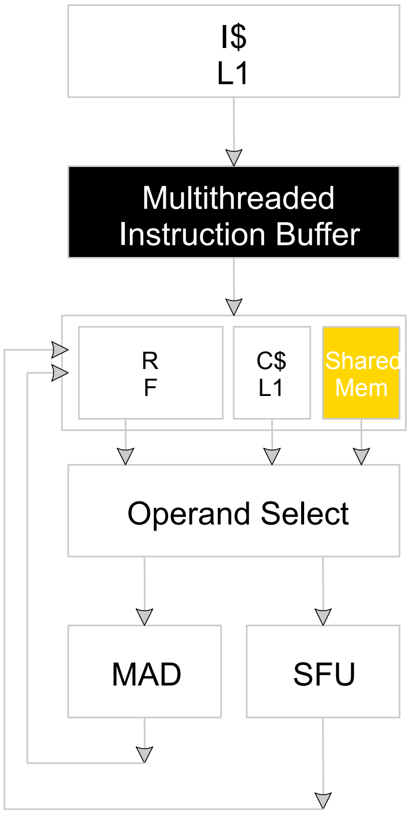
Constants

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



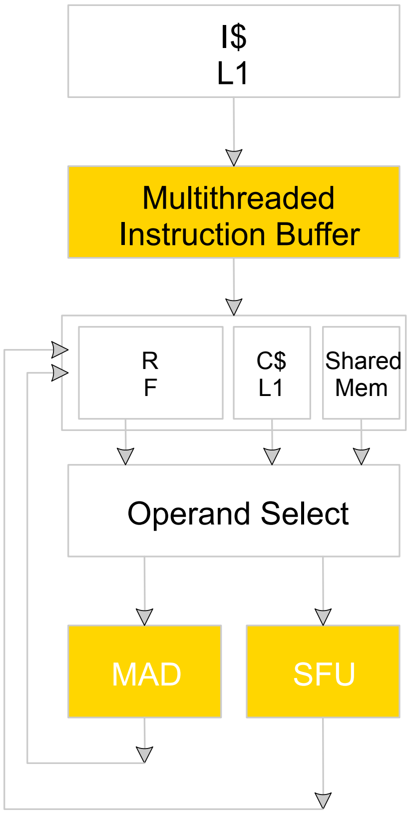
Shared Memory

- 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

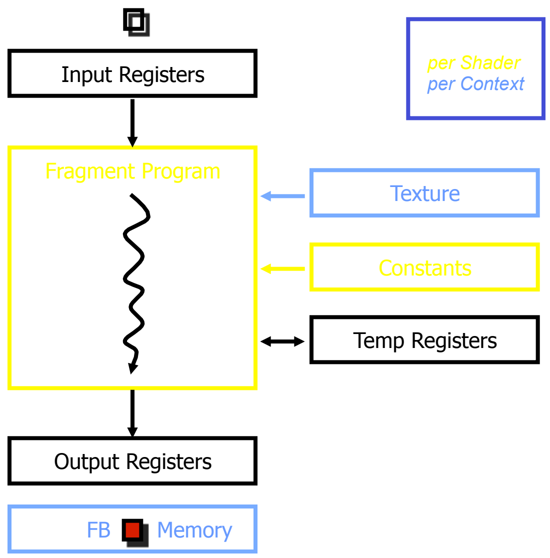
Foils adapted from nVIDIA

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, Bullet Physics, etc.
  - Image processing: Photoshop CS4, CS5, etc.
  - Video Encoding/Transcoding: Elemental RapidHD, etc.
  - Distributed processing: Stanford Folding@Home, etc.
  - RAID6, AES, MatLab, BitCoin-mining, etc.

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

Khronos Group OpenCL

- Open Computing Language
- Framework for programing heterogeneous processors
  - Version 1.0 released with Apple OSX 10.6 Snow Leopard
  - Current version is version OpenCL 1.1
- Two programing models. One suited for GPUs and one suited for Cell-like processors.
  - GPU programing model is very similar to CUDA
- Software Stack:
  - Graphics driver, language compilers (Toolkit), and tools (SDK).
  - Lab machines with nVIDIA hardware support both CUDA & OpenCL.
  - OpenCL also supported on all new AMD cards (must run on lab machine).

- You decide what to use for the home exam!

Outline

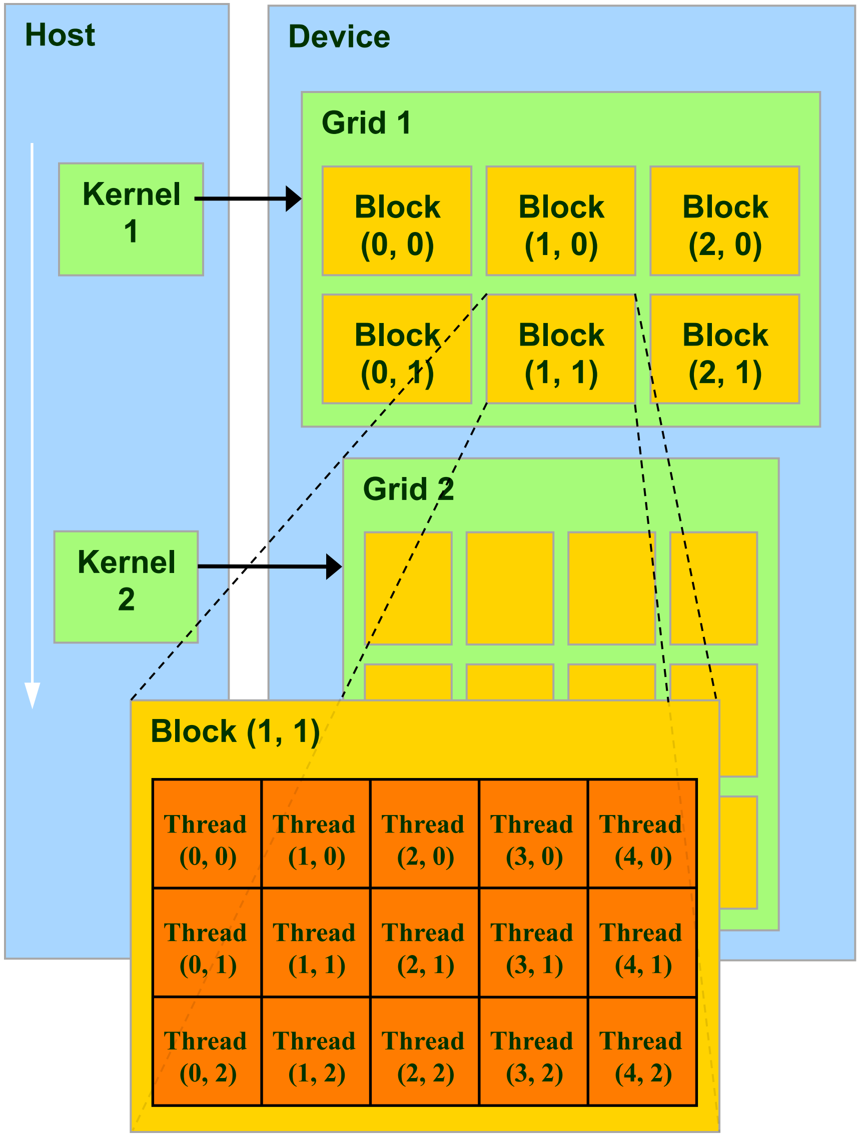
- The CUDA Programming Model
  - Basic concepts and data types
- An example application:
  - The good old Motion JPEG implementation!
- **Thursday:**
  - More details on the CUDA programming API
  - Make an example program!

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

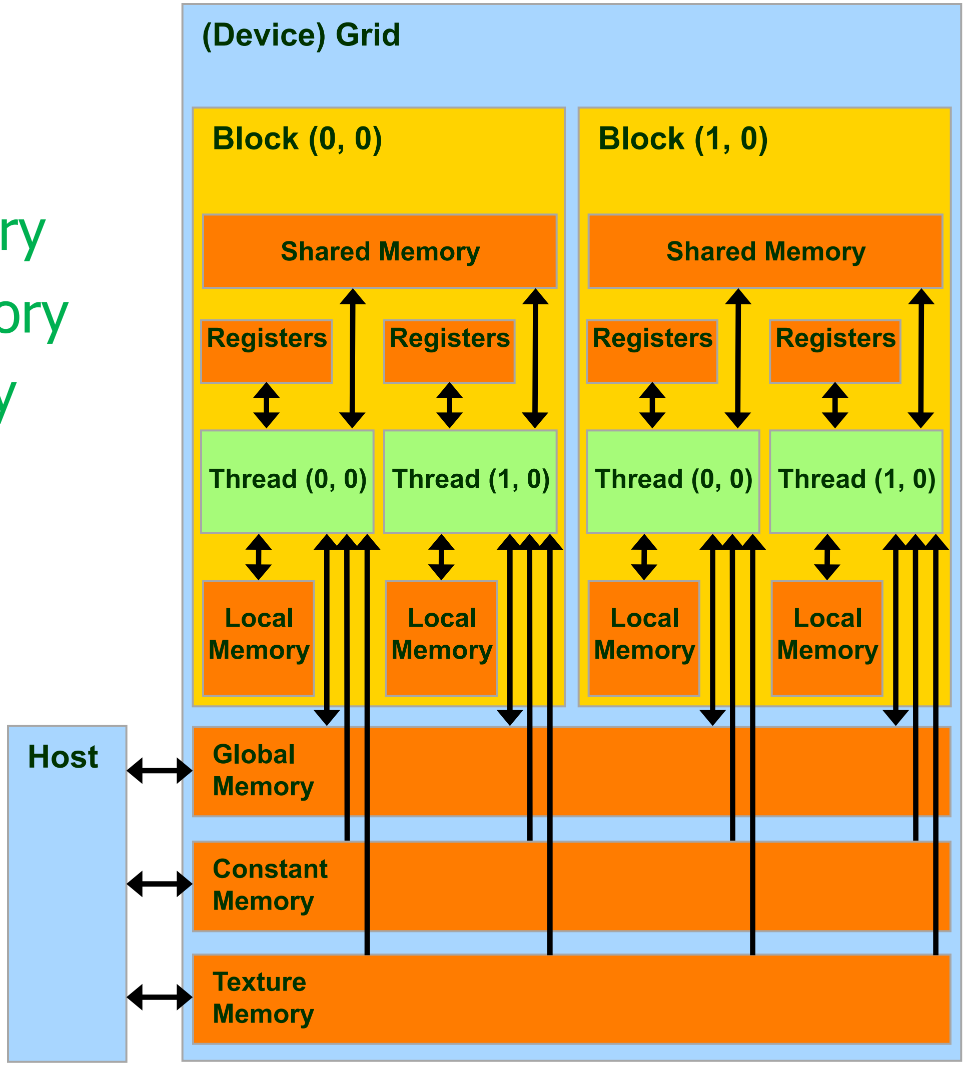
Thread Batching: Grids and Blocks

- 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



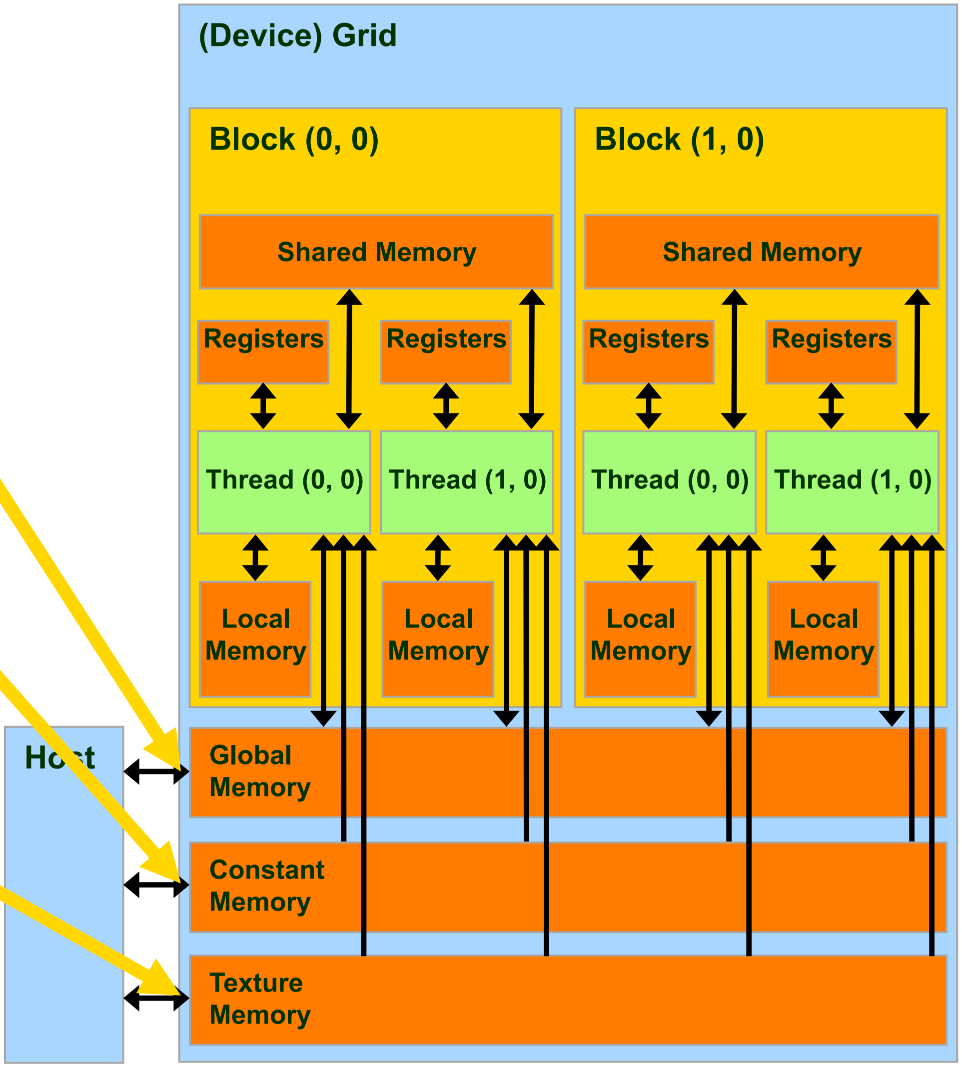
CUDA Device Memory Space Overview

- 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

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

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

Some Information on the 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 & Profiling

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

- Several tools are available to optimize your application
  - nVIDIA CUDA Visual Profiler
  - nVIDIA Occupancy Calculator

- NVIDIA Parallel Nsight for Visual Studio and Eclipse

Debugging Using Device Emulation

- An executable compiled in device emulation mode (`nvcc -deviceemu`):
  - No need of any device and CUDA driver

- When running in device emulation mode, one can:
  - Use host native debug support (breakpoints, inspection, etc.)
  - Call any host function from device code
  - Detect deadlock situations caused by improper usage of `__synchthreads`

- nVIDIA CUDA GDB (available on clinton, bush and kennedy)

- printf is now available on the device! (cuPrintf)

Before you start…

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

```
PATH=$PATH:/usr/local/cuda-5.0/bin
LD_LIBRARY_PATH=$LD_LIBRARY_PATH:/usr/local/cuda-5.0/lib64:/lib

export PATH
export LD_LIBRARY_PATH
```

- Code samples is installed with CUDA
- Copy and build in your users home directory

Some usefull resources

nVIDIA CUDA Programming Guide 5.0

http://docs.nvidia.com/cuda/pdf/CUDA_C_Programming_Guide.pdf

nVIDIA OpenCL Programming Guide

http://developer.download.nvidia.com/compute/DevZone/docs/html/OpenCL/doc/OpenCL_Programming_Guide.pdf

nVIDIA CUDA C Best Practices Guide

http://docs.nvidia.com/cuda/pdf/CUDA_C_Best_Practices_Guide.pdf

Tuning CUDA Applications for Kepler

http://docs.nvidia.com/cuda/kepler-tuning-guide/index.html

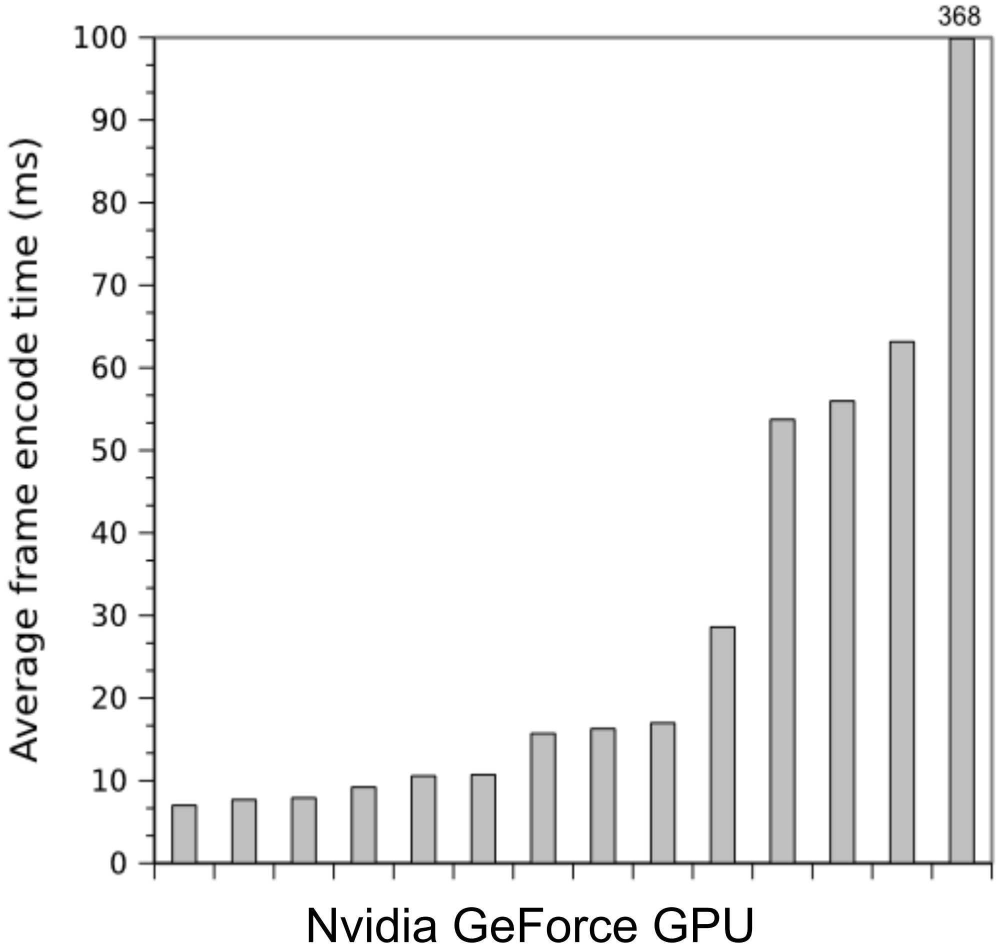
Tuning CUDA Applications for Fermi

http://developer.download.nvidia.com/compute/DevZone/docs/html/C/doc/Fermi_Tuning_Guide.pdf

Example:

Motion JPEG Encoding

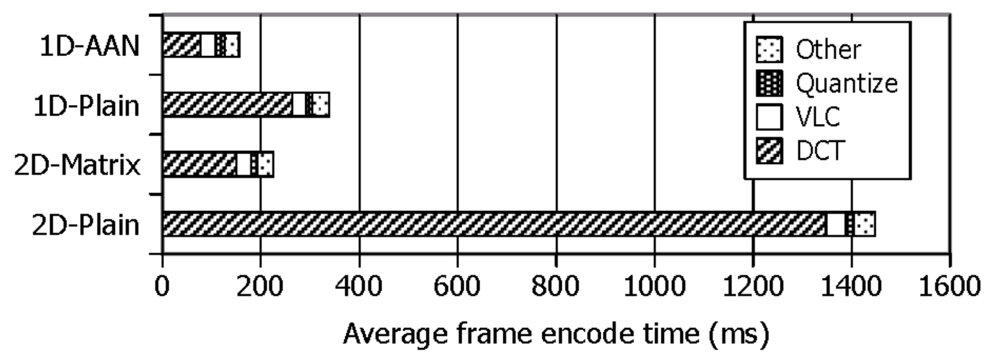
14 different MJPEG encoders on GPU



Problems:
- Only used global memory
- To much synchronization between threads
- Host part of the code not optimized

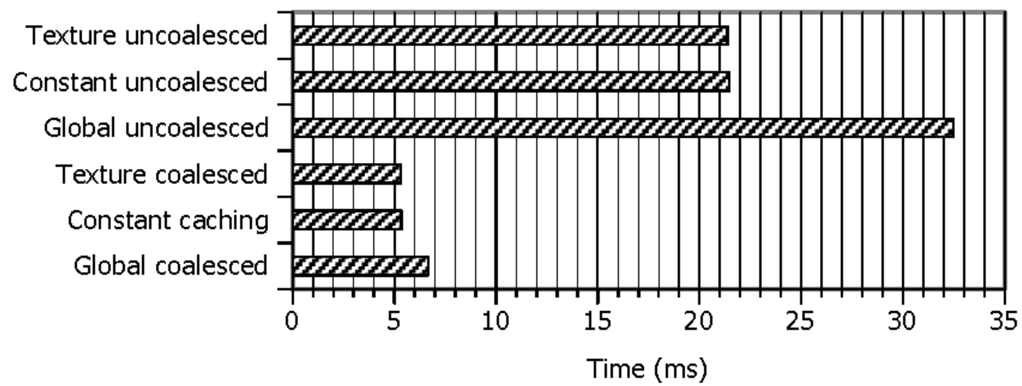
Profiling a Motion JPEG encoder on x86

- A small selection of DCT algorithms:
  - *2D-Plain:* Standard forward 2D DCT
  - *1D-Plain:* Two consecutive 1D transformations with transpose in between and after
  - *1D-AAN:* Optimized version of 1D-Plain
  - *2D-Matrix:* 2D-Plain implemented with matrix multiplication
- Single threaded application profiled on a Intel Core i5 750



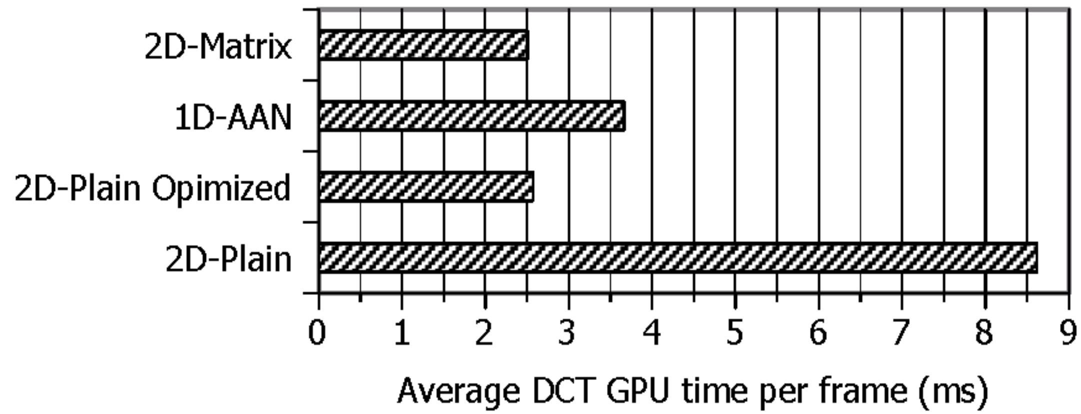
Optimizing for GPU, use the memory correctly!!

- Several different types of memory on GPU:
  - Global memory
  - Constant memory
  - Texture memory
  - Shared memory
- First Commandment when using the GPUs.
  - Select the correct memory space, AND use it correctly!



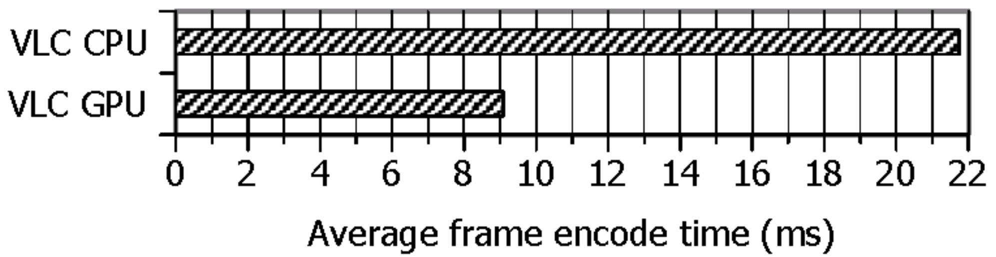
How about using a better algorithm??

- Used CUDA Visual Profiler to isolate DCT performance
- 2D-Plain Optimized is optimized for GPU:
  - Shared memory
  - Coalesced memory access
  - Loop unrolling
  - Branch prevention
  - Asynchronous transfers
- Second Commandment when using the GPUs:
  - Choose an algorithm suited for the architecture!



Effect of offloading VLC to the GPU

- VLC (Variable Length Coding) can also be offloaded:
  - One thread per macro block
  - CPU does bitstream merge
- Even though algorithm is not perfectly suited for the architecture, offloading effect is still important!



Example: Hello World

Example: Hello World

```
// Hello World CUDA - INF5063

// #include the entire body of the cuPrintf code (availible in the SDK)
#include "util/cuPrintf.cu"
#include <stdio.h>


__global__ void device_hello(void)
{
  cuPrintf("Hello, world from the GPU!\n");
}


int main(void)
{
  // greet from the CPU
  printf("Hello, world from the CPU!\n");

  // init cuPrintf
  cudaPrintfInit();

  // launch a kernel with a single thread to say hi from the device
  device_hello<<<1,1>>>();

  // display the device's greeting
  cudaPrintfDisplay();

  // clean up after cuPrintf
  cudaPrintfEnd();

  return 0;
}
```