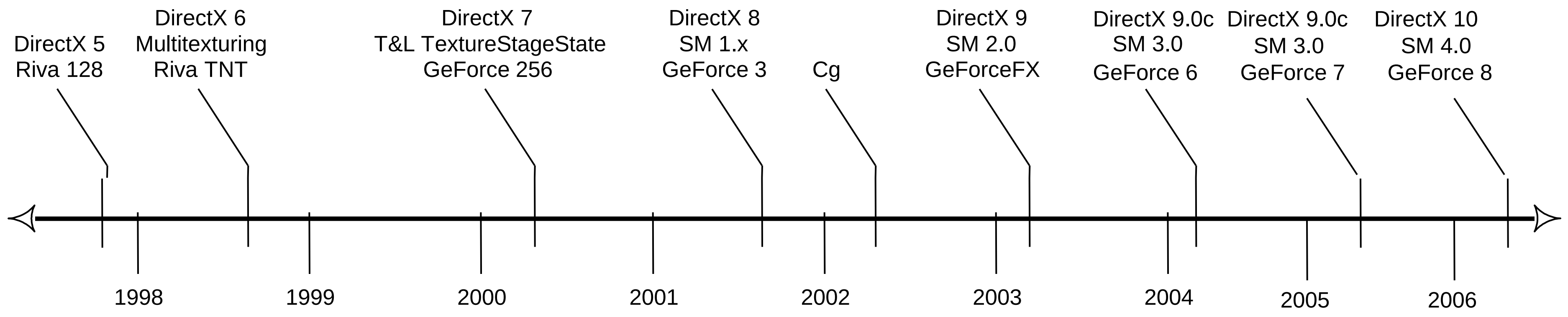
# INF5063 – GPU & CUDA

Håkon Kvale Stensland

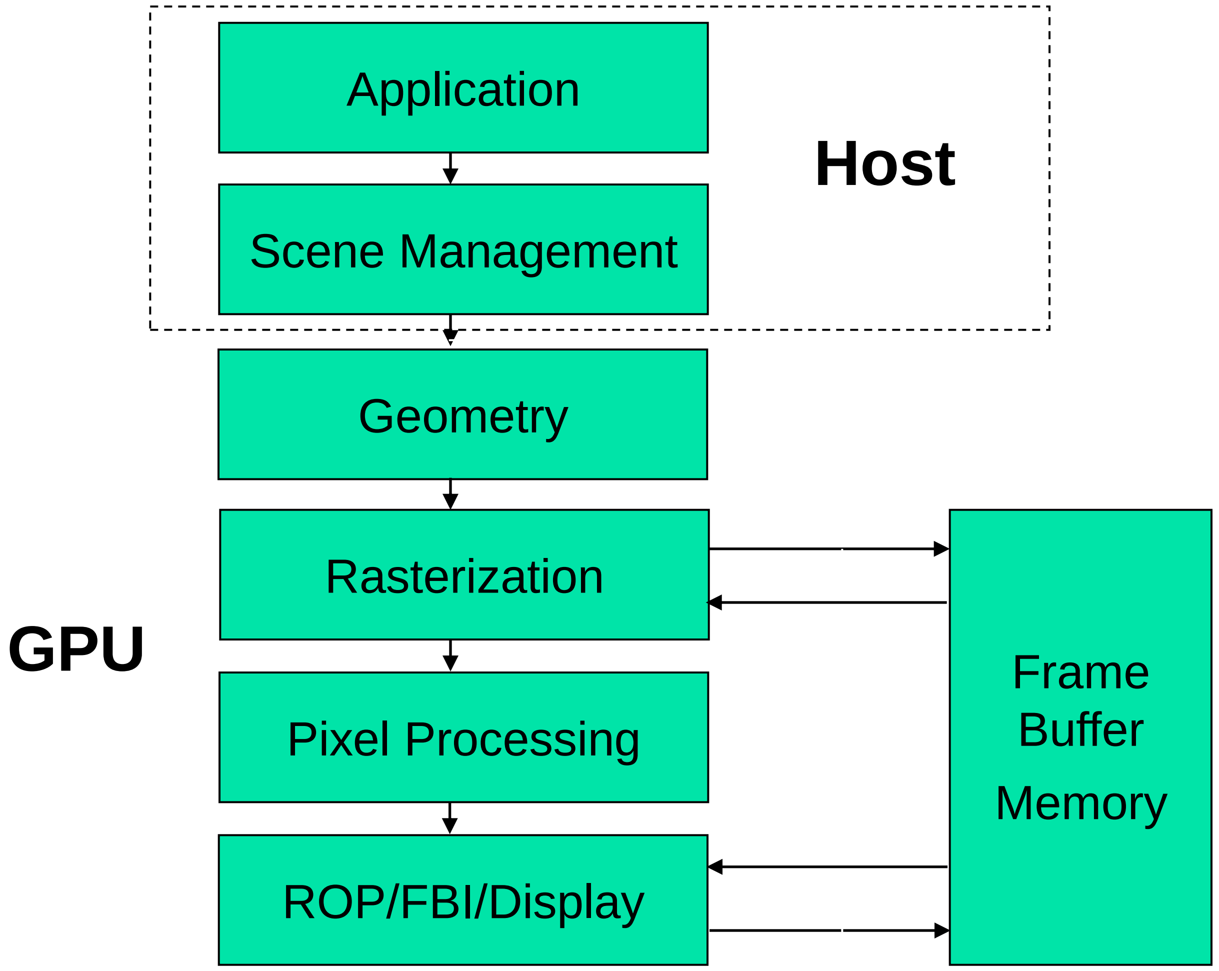
Simula Research Laboratory

# 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

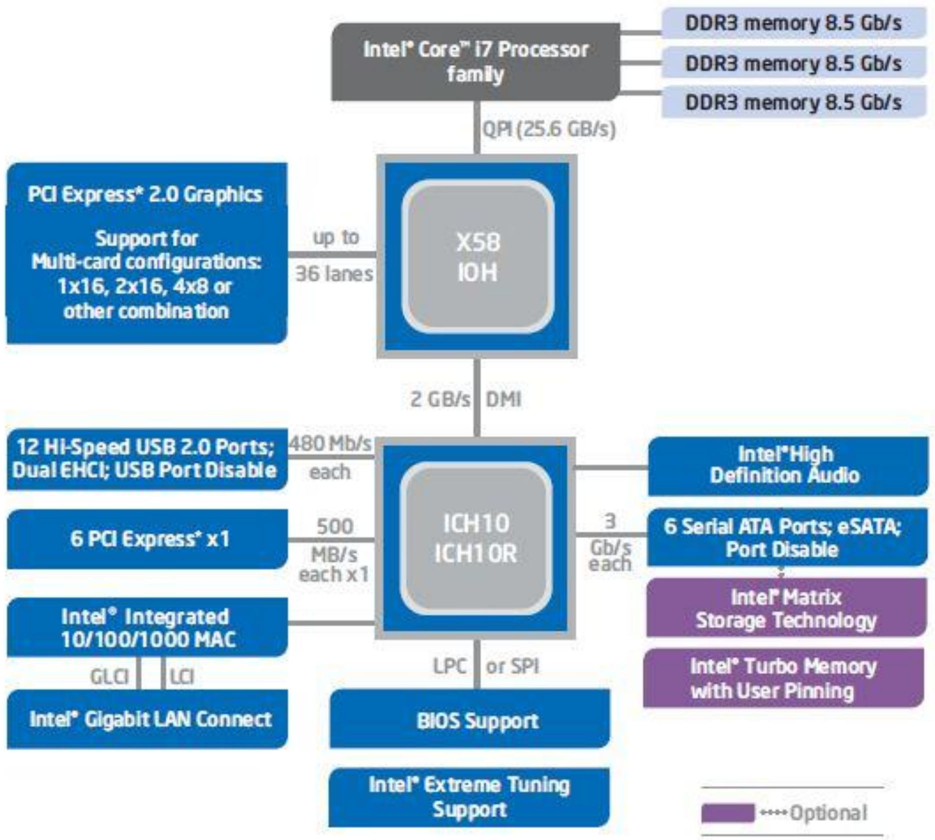


# Basic 3D Graphics Pipeline



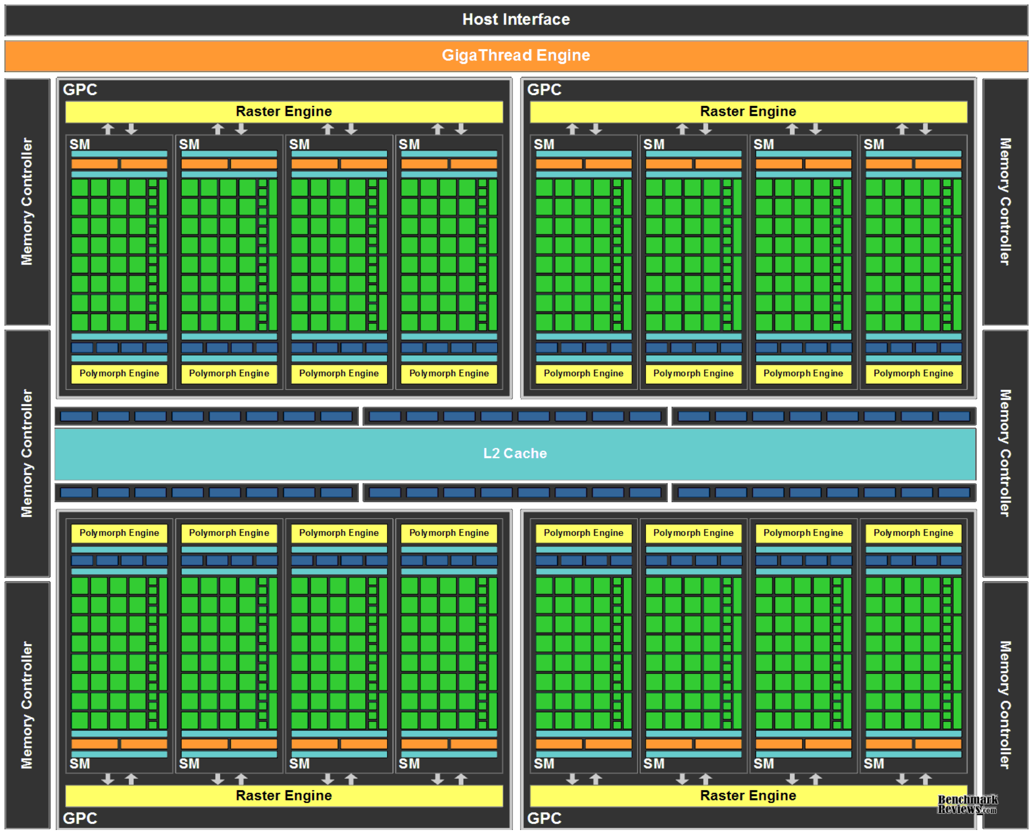
# Graphics in the PC Architecture

- QuickPath (QPI) between processor and Northbridge (X58)
  - Memory Control in CPU
- Northbridge (IOH) handles PCI Express
  - PCIe 2.0 x16 bandwidth at 16 GB/s (8 GB in each direction)
- Southbridge (ICH10) handles all other peripherals



Intel® Core™ i7 Processor family
DDR3 memory 8.5 Gb/s
DDR3 memory 8.5 Gb/s
DDR3 memory 8.5 Gb/s
QPI (25.6 GB/s)
PCI Express* 2.0 Graphics
Support for Multi-card configurations: 1x16, 2x16, 4x8 or other combination
up to 36 lanes
X58 IOH
2 GB/s DMI
12 Hi-Speed USB 2.0 Ports; Dual EHCI; USB Port Disable
480 Mb/s each
6 PCI Express* x1
500 MB/s each x1
Intel® Integrated 10/100/1000 MAC
GLCI
LCI
Intel® Gigabit LAN Connect
ICH10 ICH10R
Intel® High Definition Audio
3 Gb/s each
6 Serial ATA Ports; eSATA; Port Disable
Intel® Matrix Storage Technology
Intel® Turbo Memory with User Pinning
LPC or SPI
BIOS Support
Intel® Extreme Tuning Support
Optional

# High-end Hardware

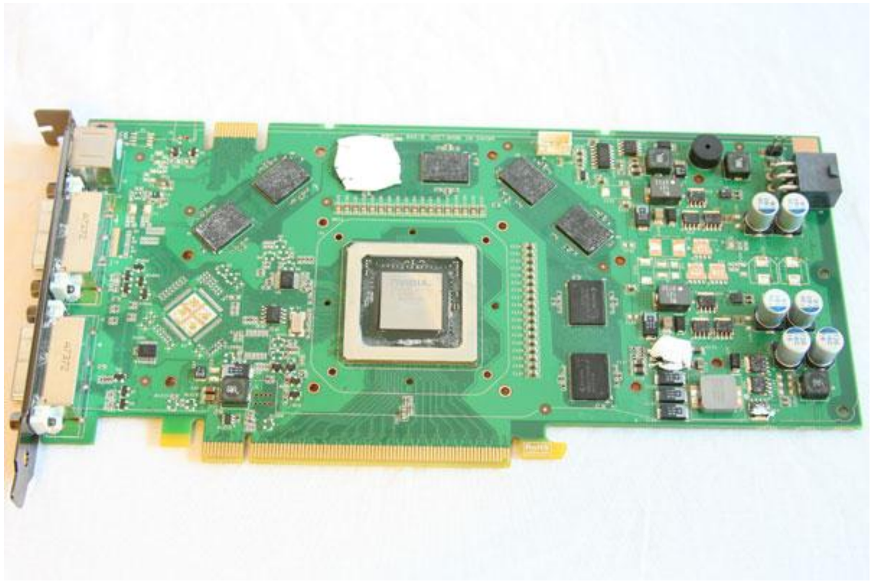


- nVIDIA Fermi Architecture
- The latest generation GPU, codenamed GF100

- 3,1 **billion** transistors
- 512 Processing cores (SP)
  - IEEE 754-2008 Capable
  - Shared coherent L2 cache
  - Full C++ Support
  - Up to 16 concurrent kernels

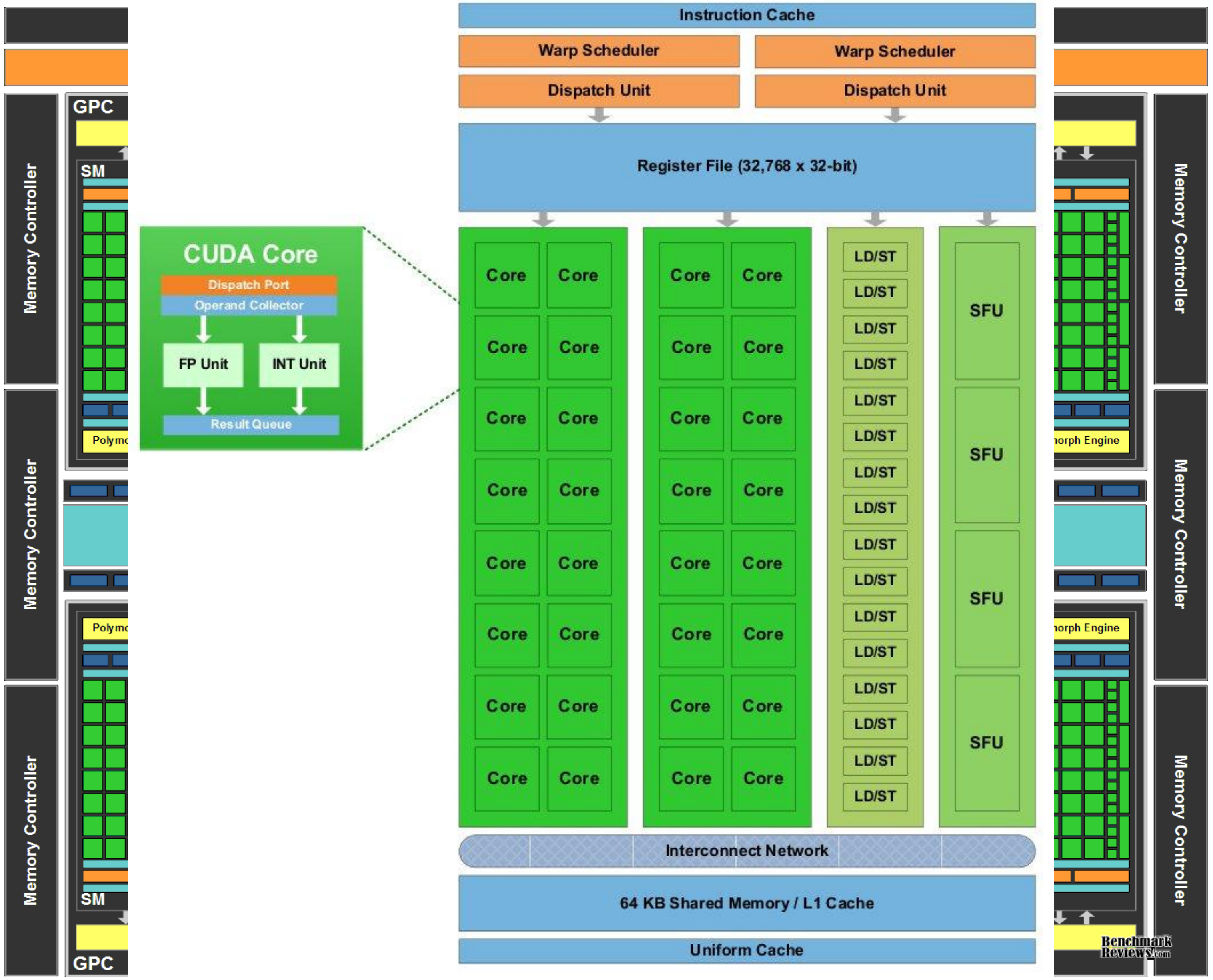
# Lab Hardware

- nVidia GeForce GTX 280
- Based on the GT200 chip
  - 1400 million transistors
  - 240 Processing cores (SP) at 1476MHz
  - 1024 MB Memory with 159 GB/sec bandwidth

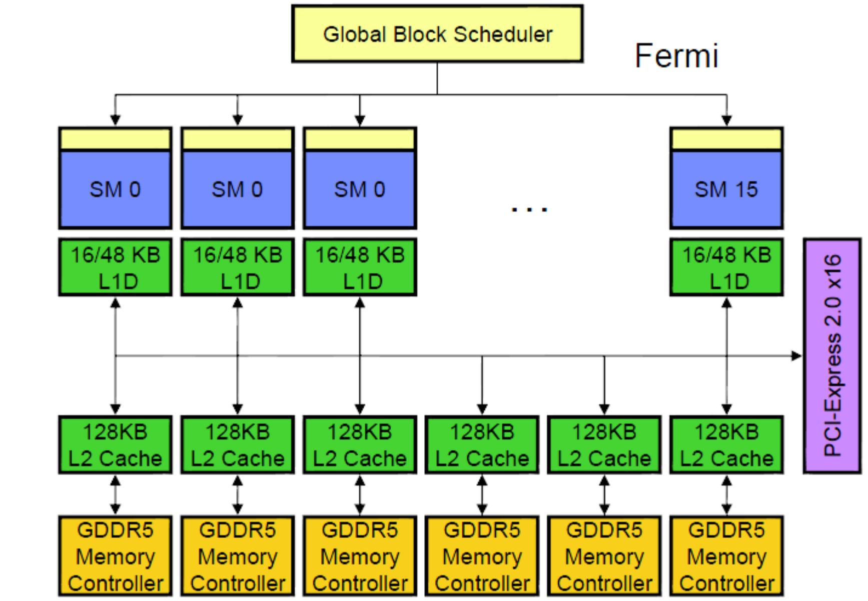


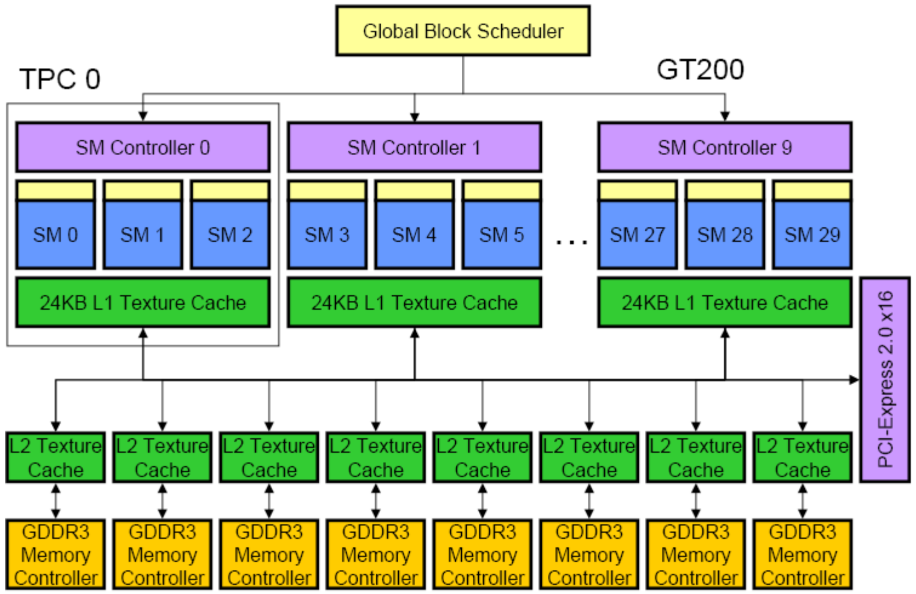
- nVidia GeForce 8800GT
- Based on the G92 chip
  - 754 million transistors
  - 112 Processing cores (SP) at 1500MHz
  - 256 MB Memory with 57.6GB/sec bandwidth

# GeForce GF100 Architecture



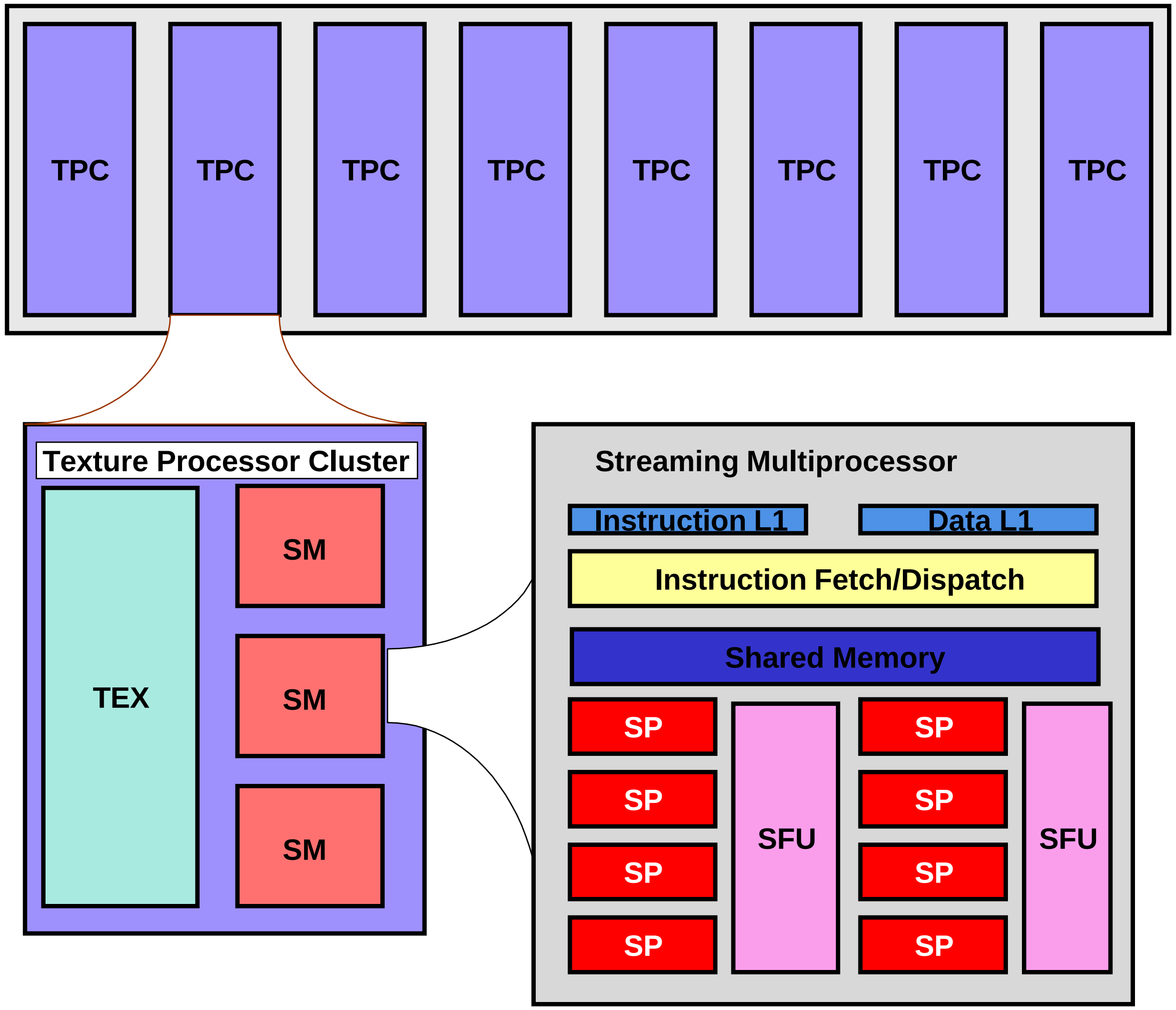
# nVIDIA GF100 vs. GT200 Architecture





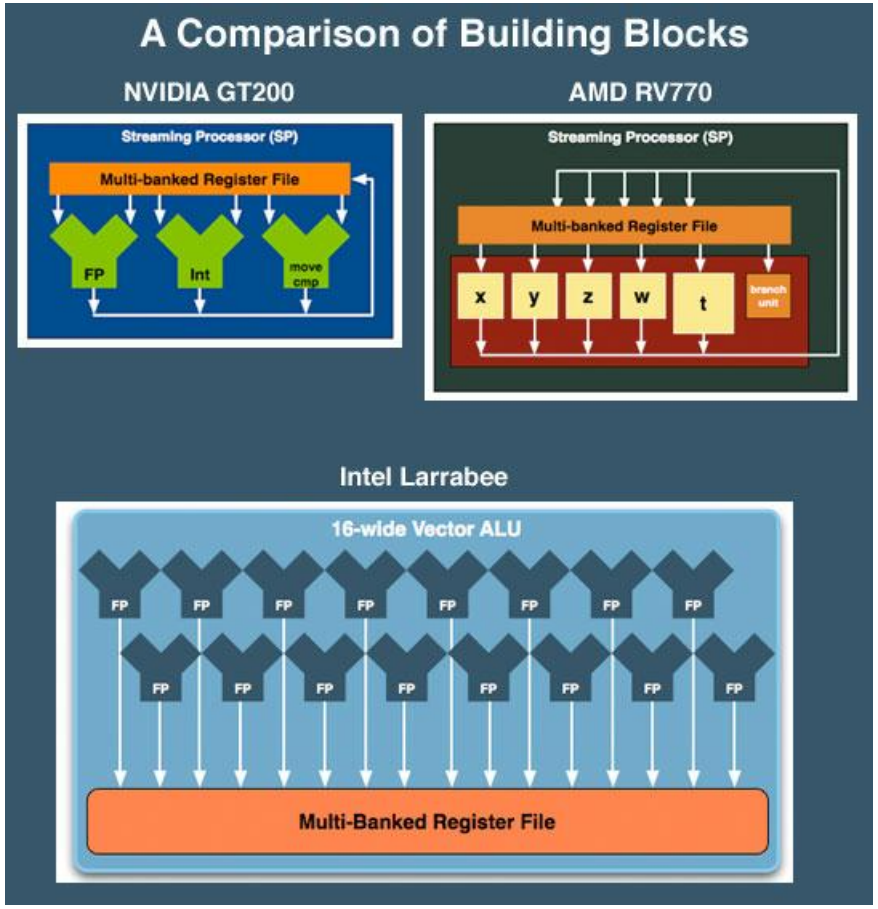
# TPC… SM… SP… Some more details…

- 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



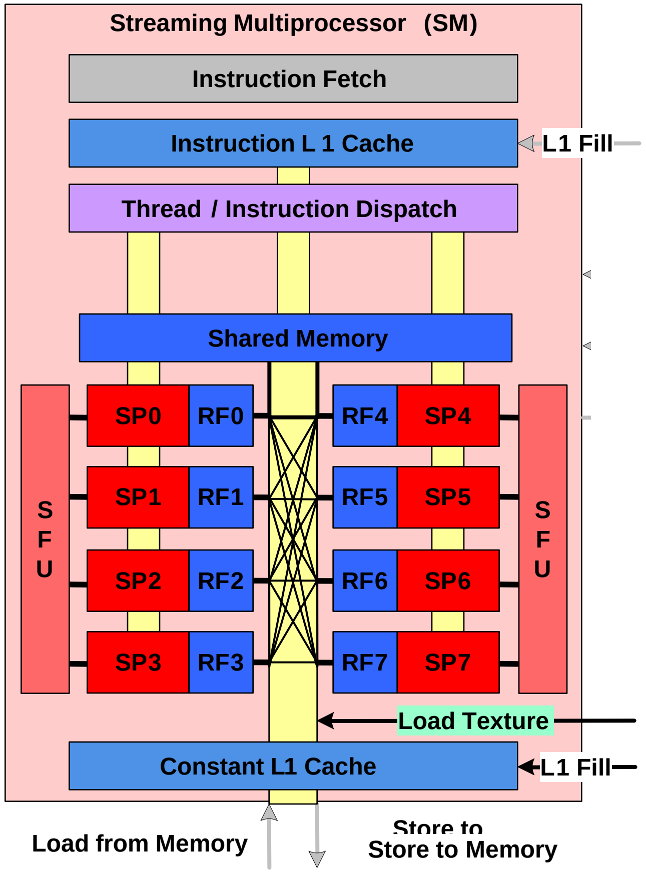
# SP: The basic processing block

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



# Streaming Multiprocessor (SM)

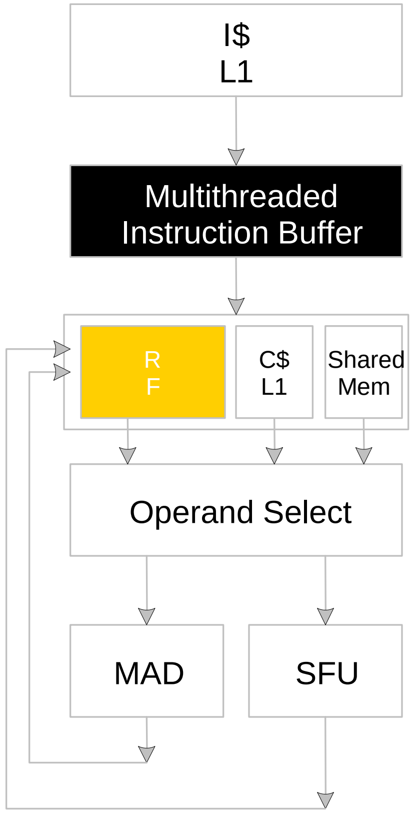
- Streaming Multiprocessor (SM)
  - 8 Streaming Processors (SP)
  - 2 Super Function Units (SFU)
- Multi-threaded instruction dispatch
  - 1 to 1024 threads active
  - Try to Cover latency of texture/memory loads
- Local register file (RF)
- 16 KB shared memory
- DRAM texture and memory access



Foils adapted from nVIDIA

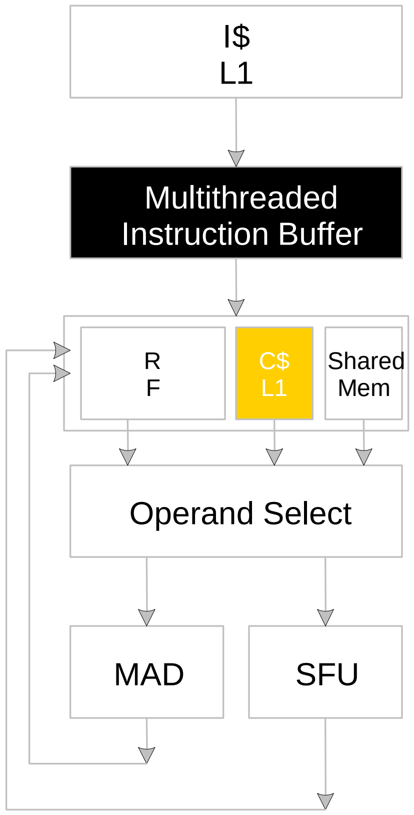
# SM Register File

- 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



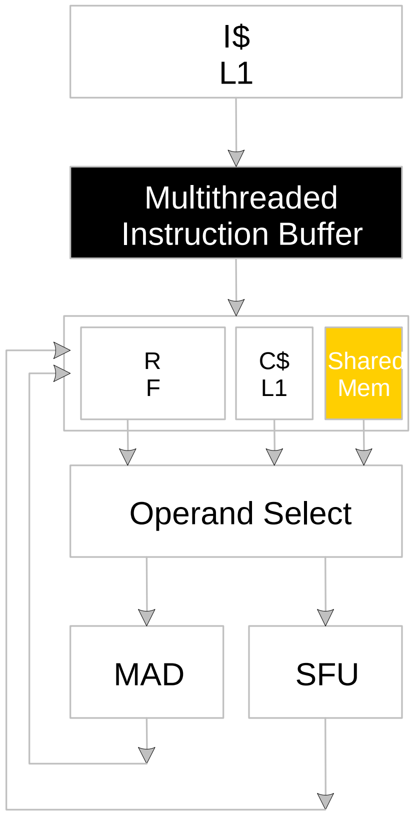
# Constants

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



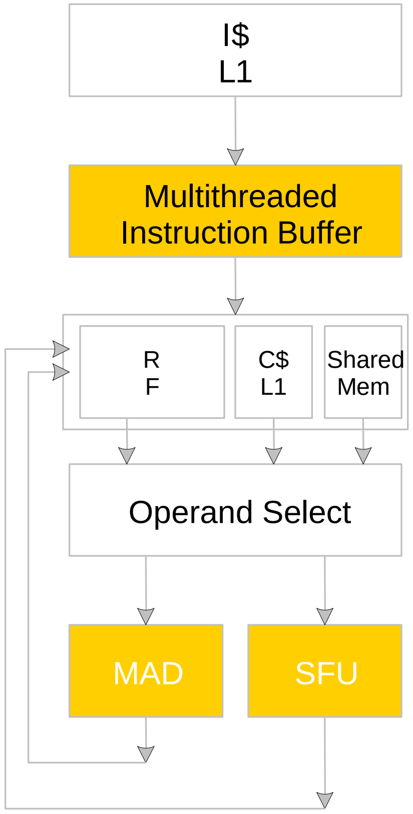
# Shared Memory

- 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

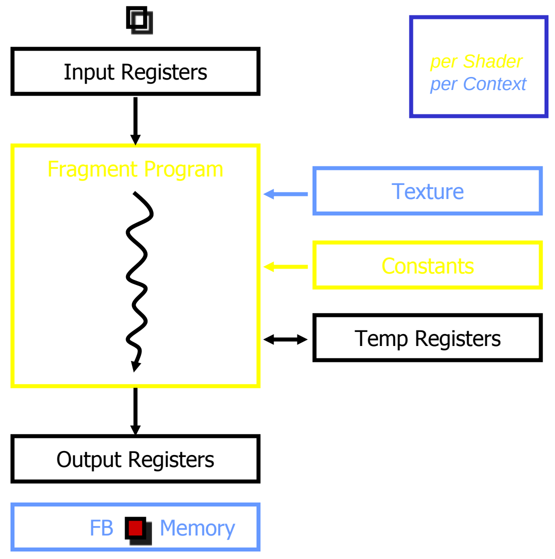
# What is really GPGPU?

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.

# 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

## Outline

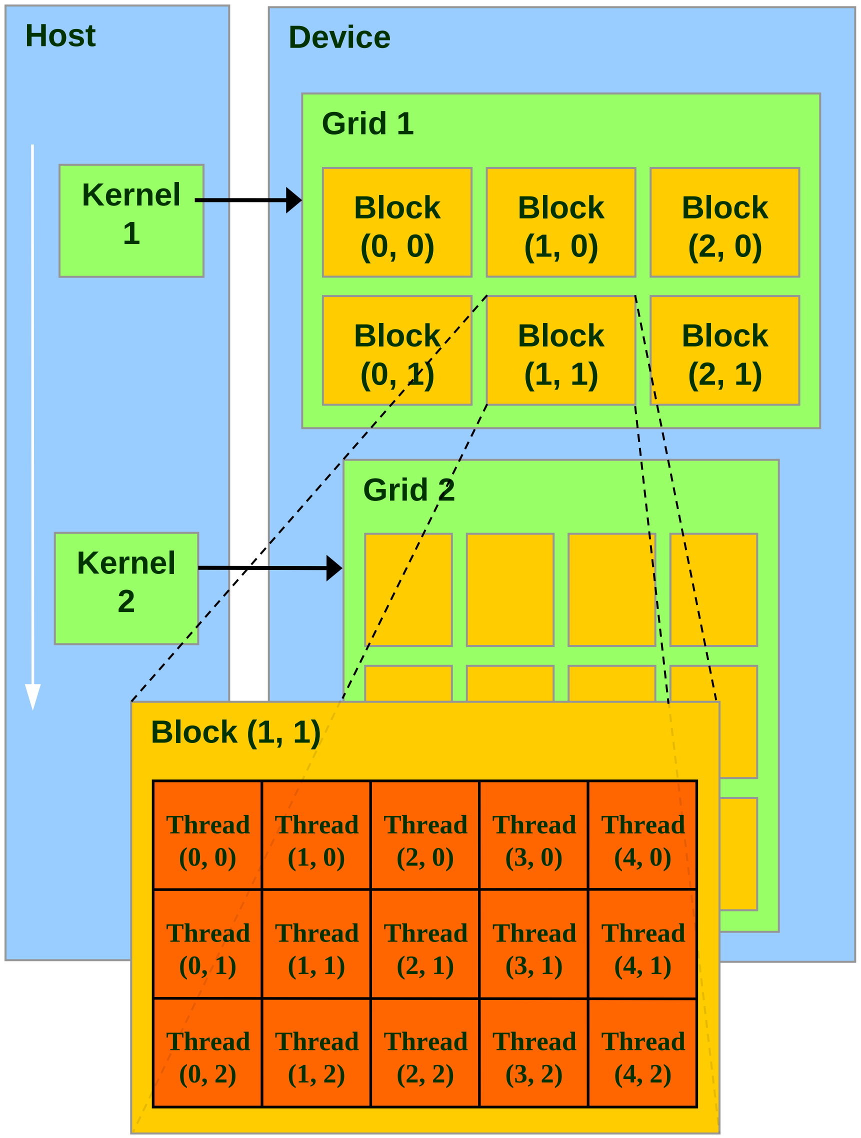
- The CUDA Programming Model
  - Basic concepts and data types
- The CUDA Application Programming Interface
  - Basic functionality
- An example application:
  - The good old Motion JPEG implementation!

## 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

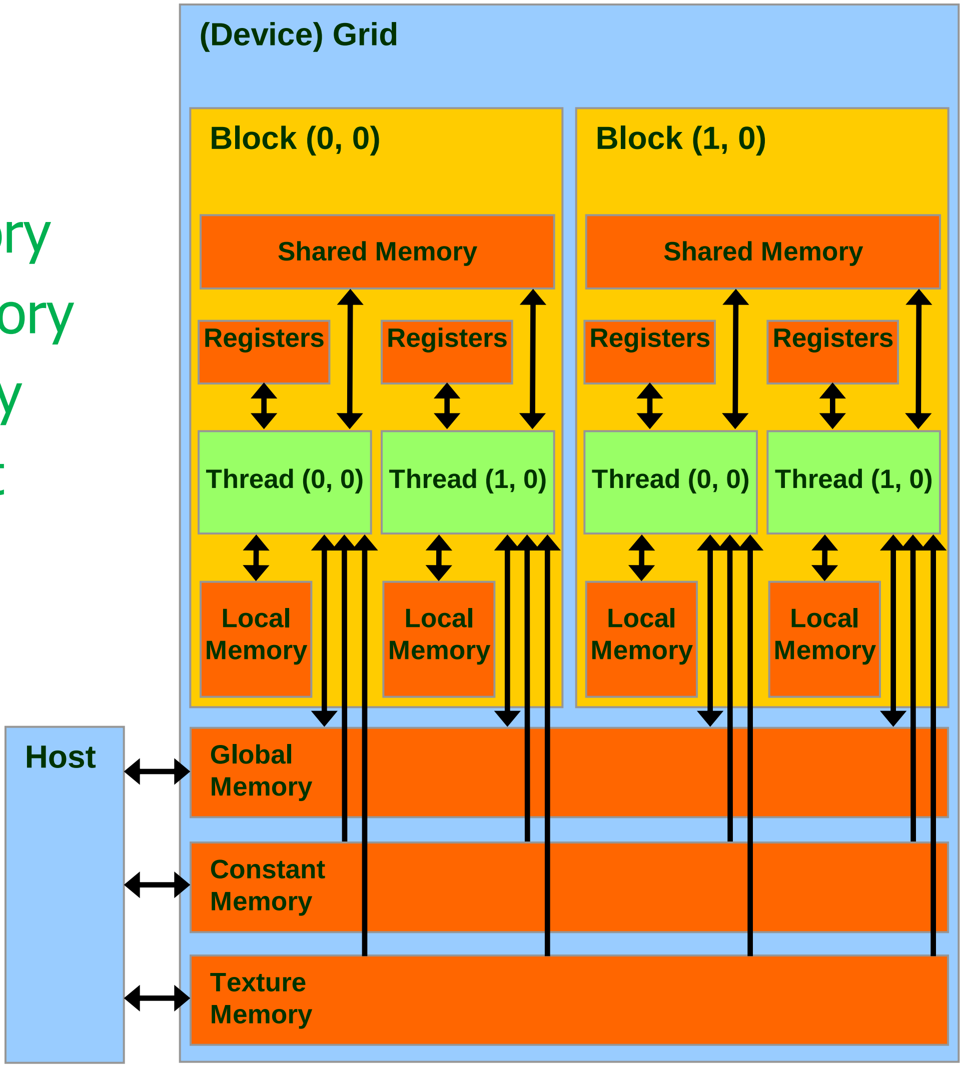
# Thread Batching: Grids and Blocks

- 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



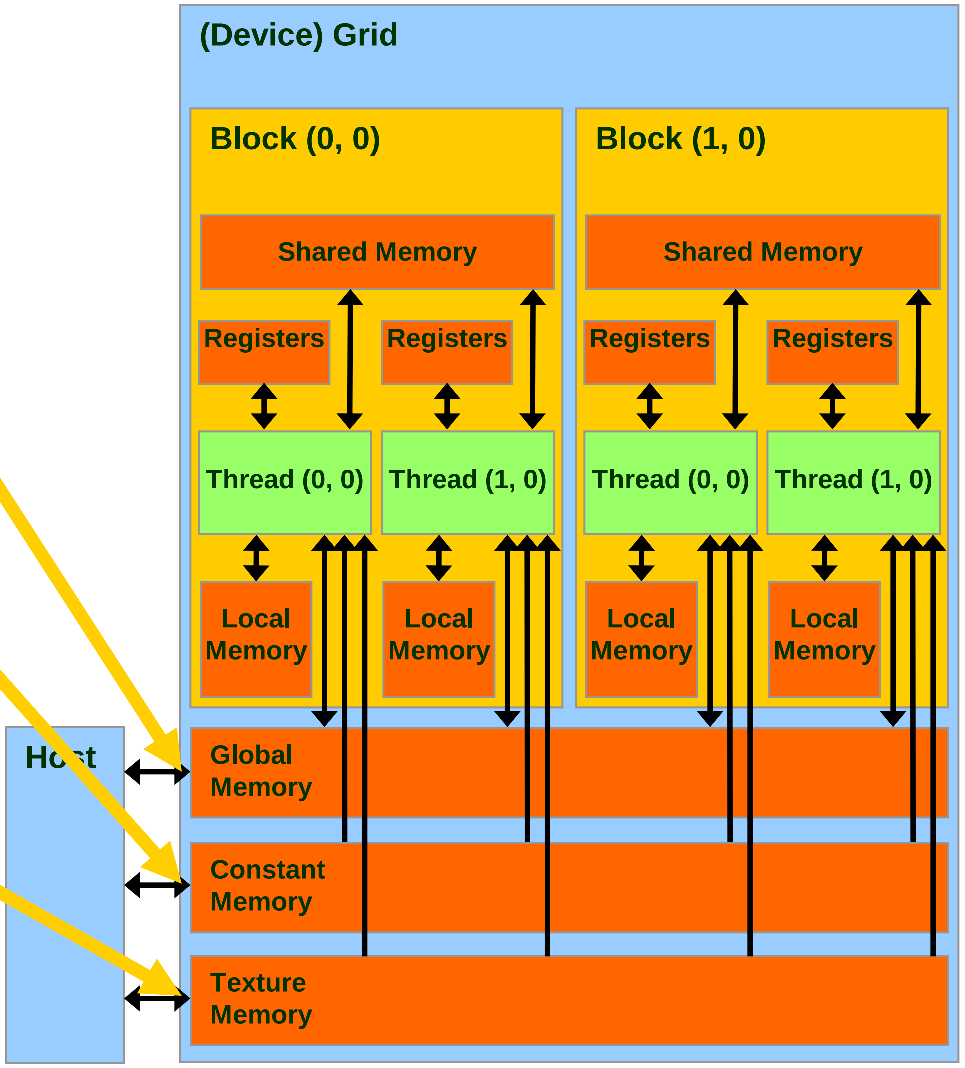
# CUDA Device Memory Space Overview

- 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

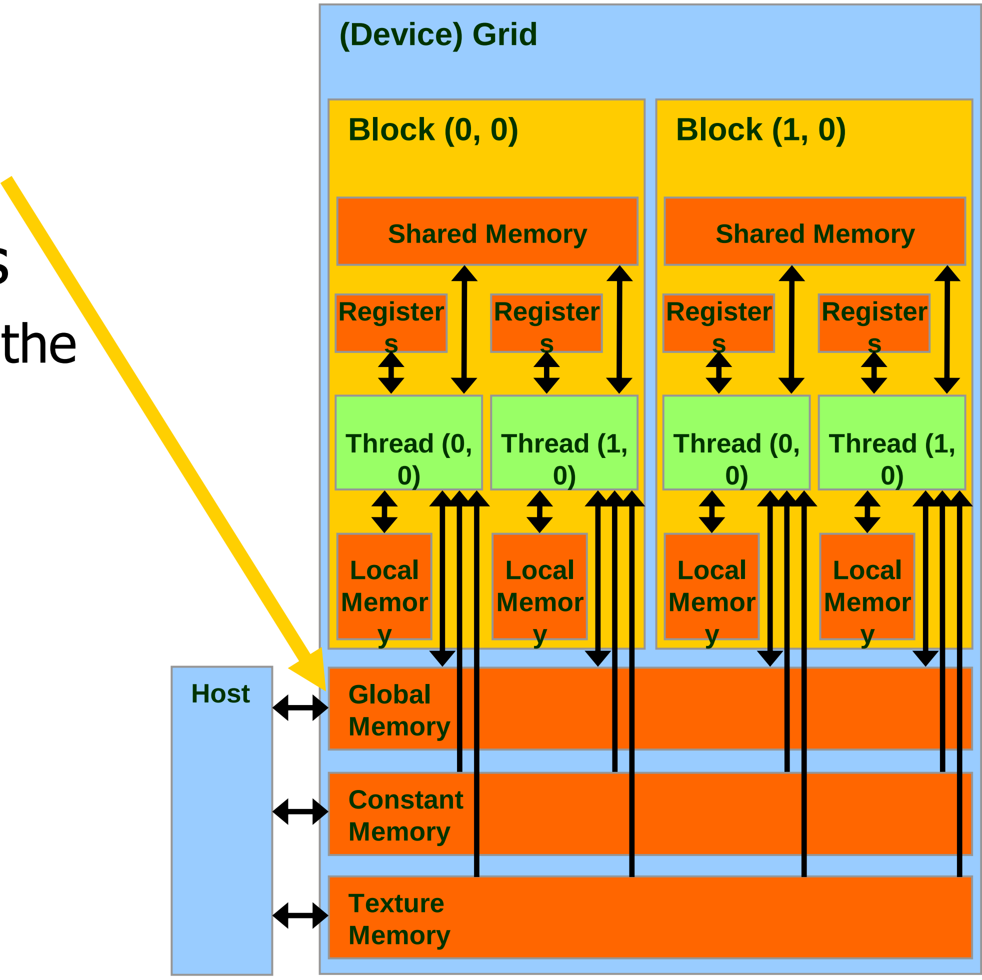
## CUDA – API

# 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

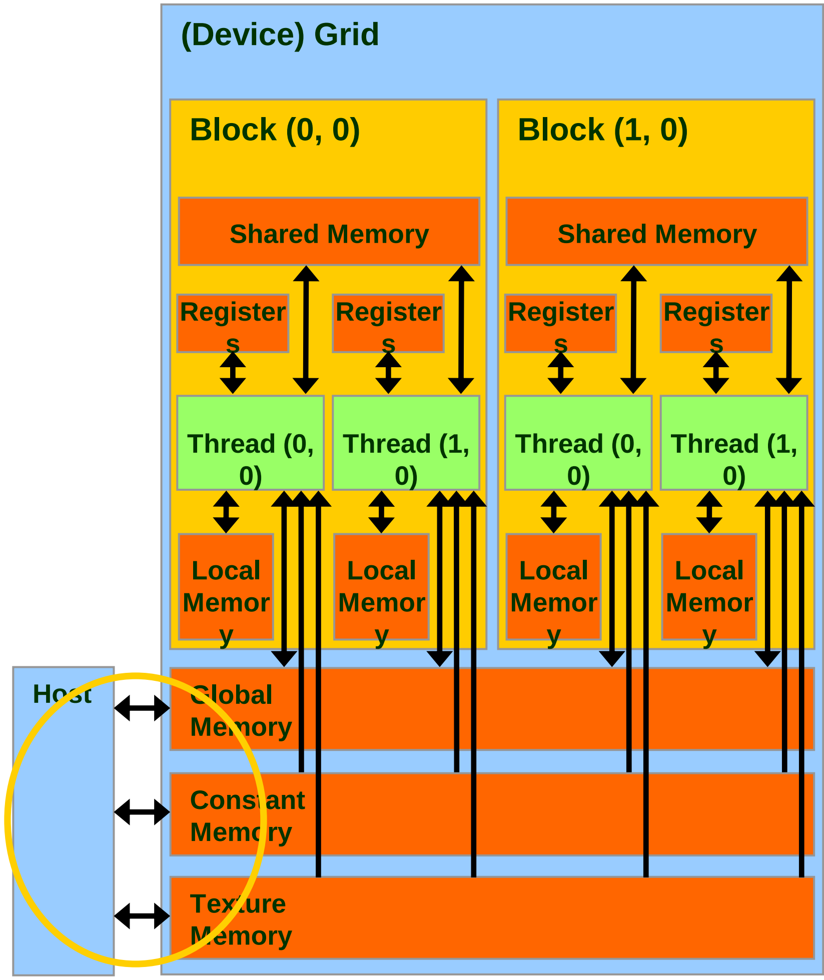
- Code example:
  - Allocate a  64 * 64 single precision float array
  - Attach the allocated storage to Md.elements
  - “d” is often used to indicate a device data structure

```
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 operations available (Streams)



## 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

## 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

- Windows users: NVIDIA Parallel Nsight for Visual Studio

# 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

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

export PATH
export LD_LIBRARY_PATH
```

- SDK is downloaded in the **/opt/** folder
- Copy and build in your users home directory

## Some usefull resources

**nVIDIA CUDA Programming Guide 3.2**

http://developer.download.nvidia.com/compute/cuda/3_2/toolkit/docs/CUDA_C_Programming_Guide.pdf

**nVIDIA CUDA C Programming Best Practices Guide**

http://developer.download.nvidia.com/compute/cuda/3_2/toolkit/docs/CUDA_C_Best_Practices_Guide.pdf

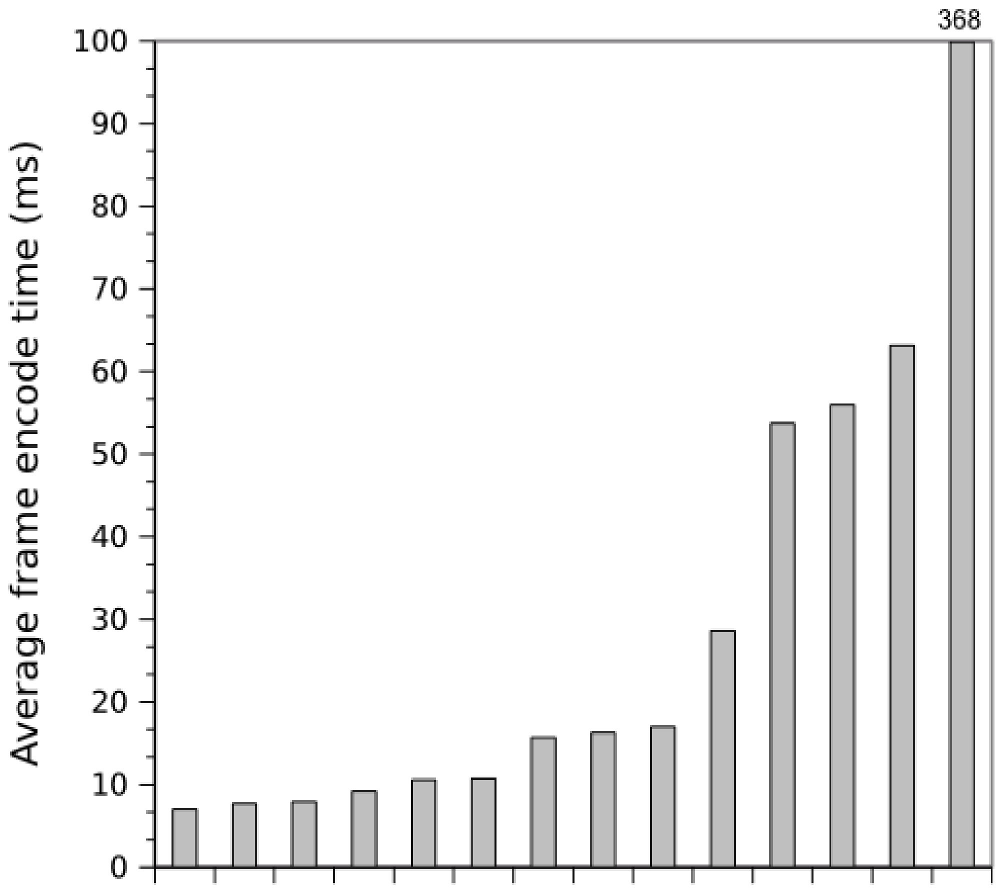
**nVIDIA CUDA Reference Manual 3.2**

http://developer.download.nvidia.com/compute/cuda/3_2/toolkit/docs/CUDA_Toolkit_Reference_Manual.pdf

## Example:

Motion JPEG Encoding

# 14 different MJPEG encoders on GPU



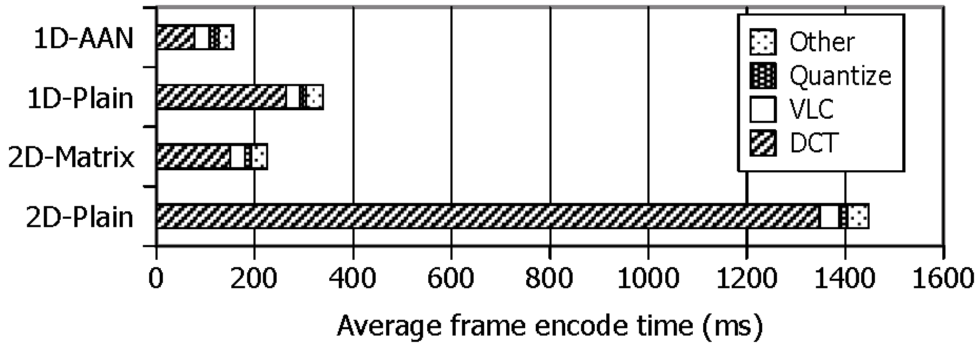
Nvidia GeForce GPU

*Problems:*

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

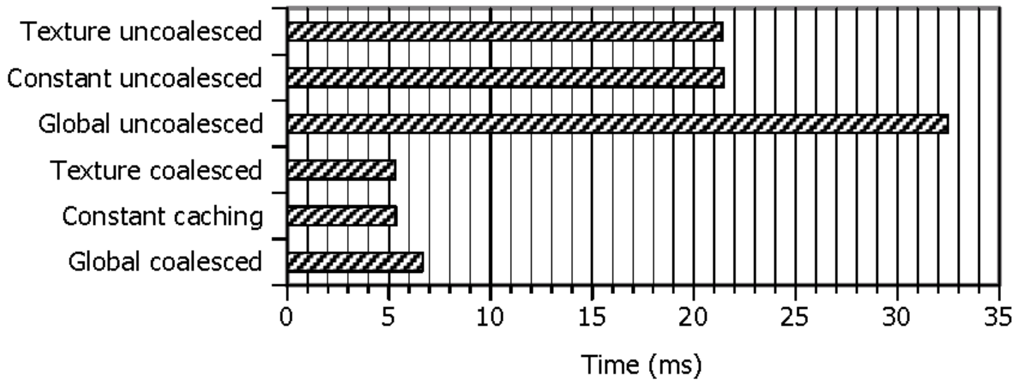
# Profiling a Motion JPEG encoder on x86

- 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



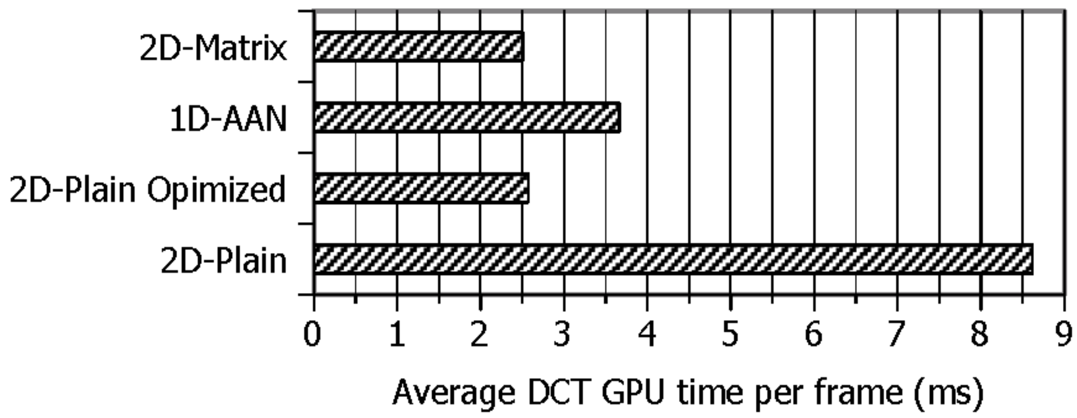
# Optimizing for GPU, use the memory correctly!!

- 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!



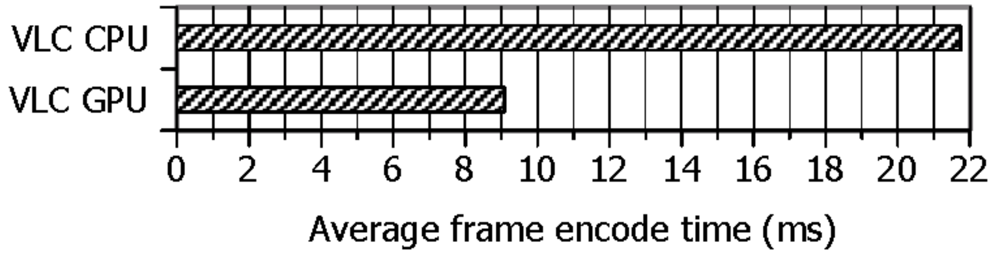
# How about using a better algorithm??

- 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!


VLC CPU
VLC GPU
Average frame encode time (ms)