GPUs

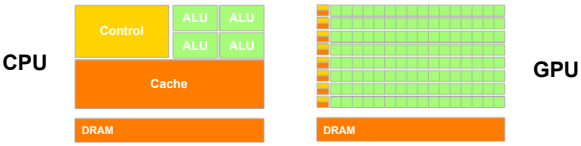
E. Sanchez, M. Sonza Reorda
Politecnico di Torino
Dipartimento di Automatica e Informatica (DAUIN)
Torino - Italy

Graphics Processing Unit (GPU)

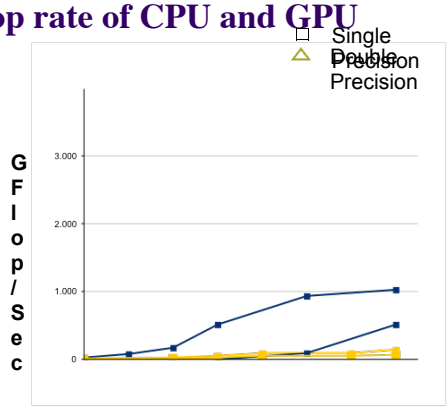
- GPU is the chip in computer video cards, PS3, Xbox, etc
  - Designed to realize the 3D graphics pipeline
    - Application → Geometry → Rasterizer→ image
- GPU development:
  - Fixed graphics hardware
  - Programmable vertex/pixel shaders
  - GPGPU
    - general purpose computation (beyond graphics) using GPUs in applications other than 3D graphics, such as
      - High Performance Computing (HPC)
      - ADAS (Advanced Driver Assistance Systems)

CPU and GPU

- GPU is specialized for compute intensive, highly data parallel computation
  - More area is dedicated to processing
  - Good for high arithmetic intensity programs with a high ratio between arithmetic operations and memory operations.

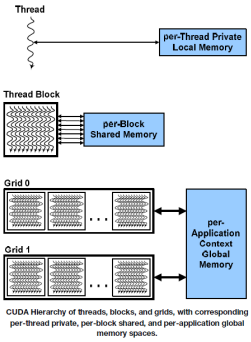


Flop rate of CPU and GPU



Compute Unified Device Architecture (CUDA)

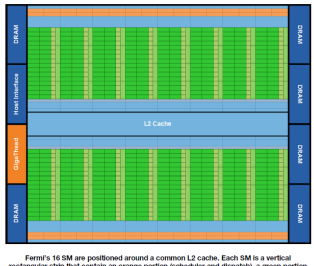
- Hardware/software architecture for NVIDIA GPU to execute programs with different languages
  - Main concept: hardware support for hierarchy of threads



CUDA Hierarchy of threads, blocks, and grids, with corresponding per-thread private, per-block shared, and per-application global memory spaces.

Fermi architecture

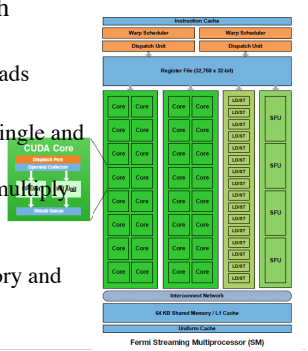
- First generation (GTX 465, GTX 480, TesLa C2050, etc) has 512 CUDA cores
  - 16 stre… multiprocessors (SMs) of 32 pr…cessing units (
  - Each … teger instruc…



Fermi's 16 SM are positioned around a common L2 cache. Each SM is a vertical rectangular strip that contain an orange portion (scheduler and dispatch), a green portion (execution units), and light blue portions (register file and L1 cache).

Fermi Streaming Multiprocessor (SM)

- 32 CUDA processors with pipelined ALU and FPU
  - Execute a group of 32 threads called warp.
  - Support IEEE 754-2008 (single and double precision floating point) with fused multiply-add (FMA) instruction).
  - Configurable shared memory and L1 cache

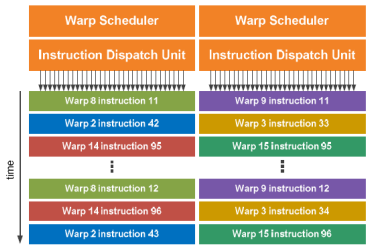


SIMT and warp scheduler

- SIMT: Single instruction, multi-thread
  - Threads in groups (or 16, 32) that are scheduled together are called *warps*.
  - All threads in a warp start at the same Program Counter (PC), but are free to branch and execute independently.
  - A warp executes one common instruction at a time
    - To execute different instructions at different threads, the instructions are executed serially
      - To get efficiency, we want all instructions in a warp to be the same.
  - SIMT is basically SIMD (Single Instruction Multiple Data) without programmers knowing it.

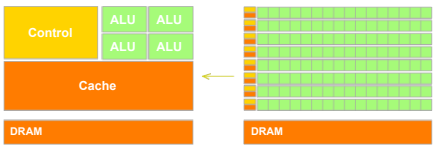
Warp scheduler

- 2 per SM: representing a compromise between cost and complexity



NVIDIA GPUs (toward general purpose computing)

GPU	G80	GT200	Fermi
Transistors	681 million	1.4 billion	3.0 billion
CUDA Cores	128	240	512
Double Precision Floating Point Capability	None	30 FMA ops / clock	256 FMA ops /clock
Single Precision Floating Point Capability	128 MAD ops/clock	240 MAD ops / clock	512 FMA ops /clock
Special Function Units (SFUs) / SM	2	2	4
Warp schedulers (per SM)	1	1	2
Shared Memory (per SM)	16 KB	16 KB	Configurable 48 KB or 16 KB
L1 Cache (per SM)	None	None	Configurable 16 KB or 48 KB
L2 Cache	None	None	768 KB
ECC Memory Support	No	No	Yes
Concurrent Kernels	No	No	Up to 16
Load/Store Address Width	32-bit	32-bit	64-bit



GPU as a co-processor

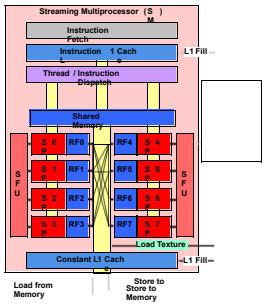
- CPU gives compute intensive jobs to GPU
- CPU stays busy with the control of execution
- Main bottleneck:
  - The connection between main memory and GPU memory
    - Data must be copied for the GPU to work on and the results must come back from GPU
    - PCIe is reasonably fast, but is often still the bottleneck.

GPGPU constraints

- Dealing with programming models for GPU such as CUDA C or OpenCL
- Dealing with limited capability and resources
  - Code is often platform dependent.
- Problem of mapping computation on to a hardware that is designed for graphics.

Streaming Multiprocessor (SM)

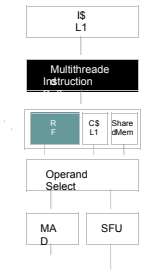
- Streaming Multiprocessor (SM)
  - 8 Streaming Processors (SPs)
  - 2 Super Function Units (SFUs)
- Multi-threaded instruction dispatch
  - 1 to 768 threads active
  - Try to Cover latency of texture/memory loads
- Local register file (RF)
- 16 KB shared memory
- DRAM texture and memory access



Foils adapted from nVIDIA

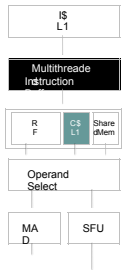
SM Register File

- 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



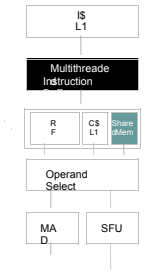
Constants

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



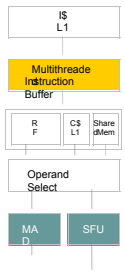
Shared Memory

- 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 SMs in a TPC)
- Load/Store pipe
  - CUDA has both global and local memory access through Load/Store



GPGPU

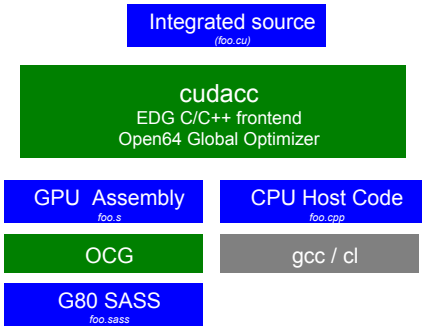
What is really a GPGPU?

- 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)
  - Image processing
  - Video Encoding/Transcoding
  - Distributed processing
  - RAID6, AES, MatLab, etc.

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

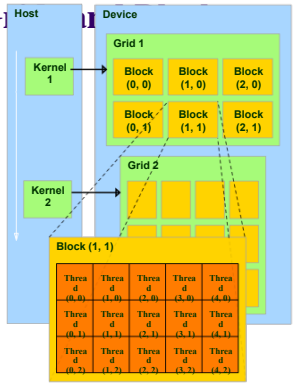


The CUDA Programming Model

- 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

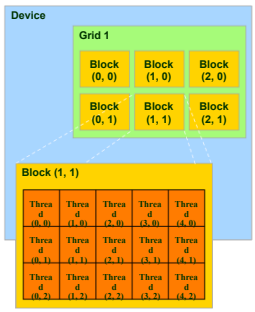
Thread Batching: G

- A kernel is executed as a grid of thread blocks
  - All threads share the 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



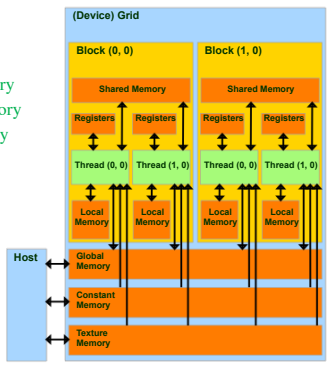
Block and Thread IDs

- 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…)



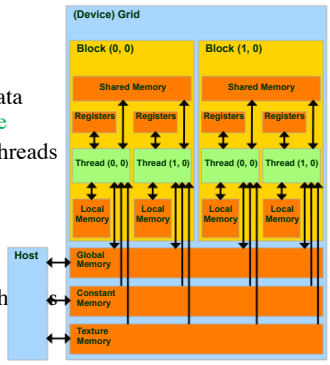
CUDA Device Memory Space Overview

- 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 th



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 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 code example (C version)

```
// Invoke DAXPY
daxpy(n, 2.0, x, y);
// DAXPY in C
void daxpy(int n, double a, double *x, double *y)
{
          for (int i = 0; i < n; ++i)
                y[i] = a*x[i] + y[i];
}
```

CUDA code example

```
// Invoke DAXPY with 256 threads per Thread Block
__host__
int nblocks = (n+ 255) / 256;
   daxpy<<<nblocks, 256>>>(n, 2.0, x, y);
// DAXPY in CUDA
__device__
void daxpy(int n, double a, double *x, double *y)
{
   int i = blockIdx.x*blockDim.x + threadIdx.x;
   if (i < n) y[i] = a*x[i] + y[i];
}
```

GRID

CUDA code exam

These 2 parameters represent
- The number of blocks
- The number of threads per block

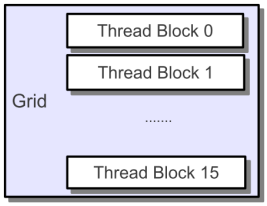
```
// Invoke DAXPY with 256 threads per Thread Block
__host__
int nblocks = (n+ 255) / 256;
   daxpy<<<nblocks, 256>>>(n, 2.0, x, y);
// DAXPY in CUDA
__device__
void daxpy(int n, double a, double *x, double *y)
{
   int i = blockIdx.x*blockDim.x + threadIdx.x;
   if (i < n) y[i] = a*x[i] + y[i];
}
```

GRID

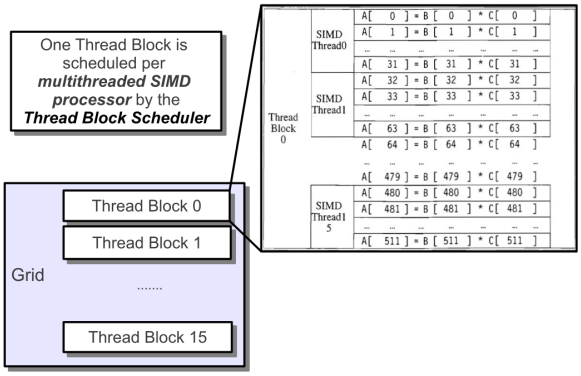
CUDA code example

Elementwise multiplication of 2 vectors of 8192 elements each

512 threads per Thread Block | 8192/512 = 16 Thread Blocks

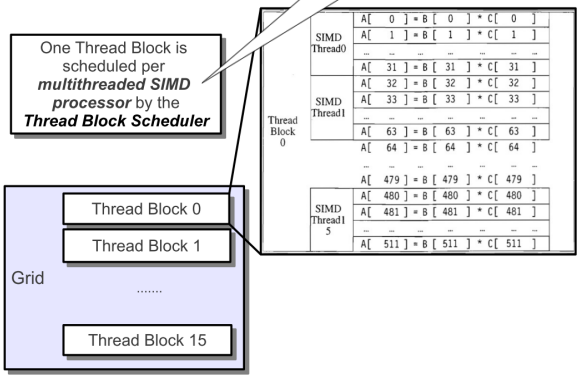


NVIDIA GPU Computational Structures

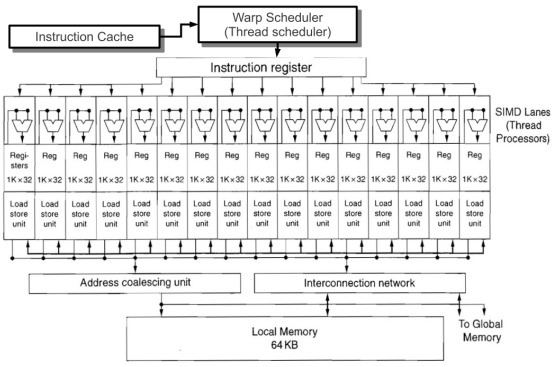


NVIDIA GPU Co Structures

Corresponding to a SM



Multithreaded SIMD Processor



Performance comparison between CPU and GPU

Dimensions	CUDA	CPU
64x64	0.417465 ms	18.0876 ms
128x128	0.41691 ms	18.3007 ms
256x256	2.146367 ms	145.6302 ms
512x512	8.093004 ms	1494.7275 ms
768x768	25.97624 ms	4866.3246 ms
1024x1024	52.42811 ms	66097.1688 ms
2048x2048	407.648 ms	Didn't finish
4096x4096	3.1 seconds	Didn't finish