MIC-GPU: High-Performance Computing for Medical Imaging on Programmable Graphics Hardware (GPUs)

GPU Hardware

Klaus Mueller, Ziyi Zheng, Eric Papenhausen

Stony Brook University

Computer Science

Stony Brook, NY

NVIDIA Fermi Architecture

GeForce 500 series → consumer graphics board

- 1.5 GB DRAM

Tesla 2000 series → general computing board

- 6 GB DRAM
- 2 x double precision performance
- ECC (Error Correcting Code) memory

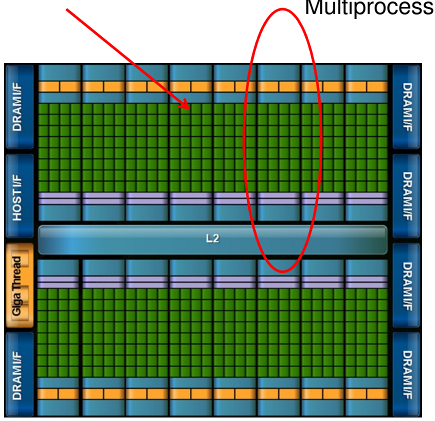
Quadro 6000 series → professional graphics board

- Similar as Tesla but with video output

NVIDIA Fermi

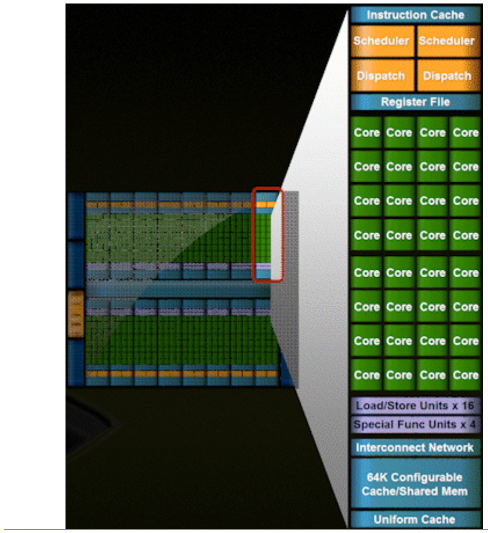
CUDA Core

SM (Streaming Multiprocessor) → has 32 Streaming Processors (SP) = CUDA core



On chip:
SMs: up 16
CUDA cores: 32/SM → up to 512/chip

NVIDIA Fermi



full cross-bar interface

32 CUDA Cores

4 special function units (sin, cosine, reciprocal, and square root)

Host and Device

Host → CPU

- controls program flow
- manages threads
- loads GPU programs (kernels)
- has host memory

Device → GPU

- loads data
- performs computations
- has device memory

Heterogeneous programming model

Thread Hierarchy : Coarse Grain

Parallelism is exposed as *threads*

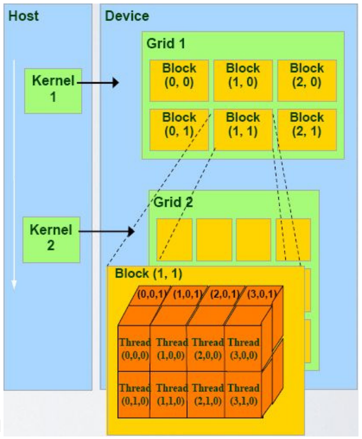
- all threads run the same code
- a thread runs on one core

The threads divide into *blocks*

- each block has a unique ID → *block ID*
- each thread has a unique ID within a block → *thread ID*
- block ID and thread ID can be used to compute a *global ID*

The blocks form a *grid*

Block/grid size can be set in program



Thread Hierarchy: Fine Grain

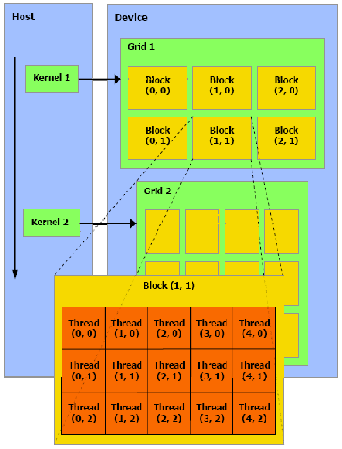
Threads within a block are organized into *warps*

- execute the same instruction simultaneously with different data

A warp is 32 threads (fixed)

One SM can maintain 48 warps simultaneously

- keep one warp active while 47 wait for memory → latency hiding
- 32 threads × 48 warps ×16 SMs → 24,576 threads !



CUDA Hardware Implementation

Upon invoking a CUDA program from the host:

Block-level

- blocks are serially distributed to SMs
- threads of a block execute on one SM
- as thread blocks terminate, new blocks are launched on vacated SMs

Thread-level

- each SM launches warps of threads
- SM schedules and executes warps that are ready to run
- as warps and thread blocks complete, resources are freed

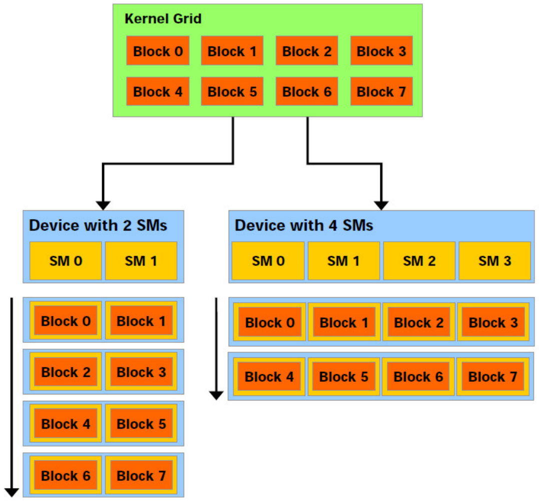
Mapping the Architecture to Parallel Programs

Mapping of blocks to SMs

- depends on device hardware
- *transparent scalability*

Thread management

- very lightweight thread creation, scheduling
- in contrast, on the CPU thread management is very heavy



Block Scheduling: Example

Threads are assigned to SMs in block granularity

- up to 8 blocks to each SM as resource allows

An SM can take up to 1,536 threads

- could be 512 (threads/block) * 3 blocks
- or 256 (threads/block) * 6 blocks, etc.

The optimal block size depends on:

- how much latency needs to be hidden (larger blocks)
- how much memory is needed per thread (smaller blocks)

Memory Hierarchy

CUDA threads may access data from multiple memory spaces:

Thread-level

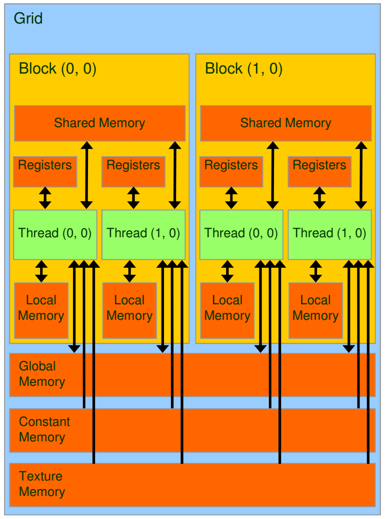
- registers (fast)
- local memory to handle register spills (slow)

Block-level

- shared memory

Grid-level

- global memory
- constant memory (read-only)
- texture memory (read-only)

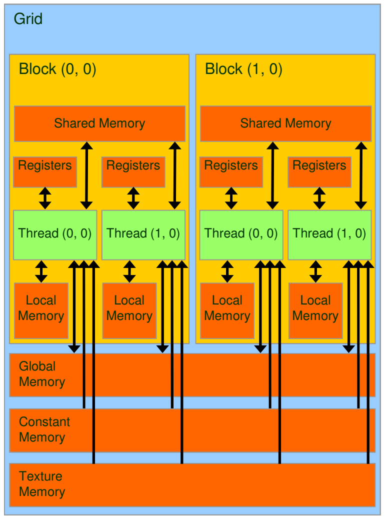


Memory Hierarchy

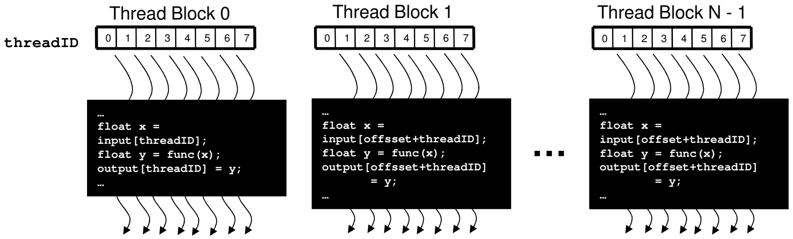
Memory	On-chip	Cached	Access
Local	N	Y	RW
Shared	Y	Y	RW
Global	N	1D	RW
Constant	N	Y	R
Texture	N	1-3D	R

Code development strategy

- start by using just global memory
- then optimize
- more about this later



No Thread Communication

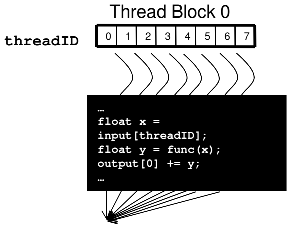


```
…
float x =
input[threadID];
float y = func(x);
output[threadID] = y;
…
```

```
…
float x =
input[offsset+threadID];
float y = func(x);
output[offsset+threadID]
       = y;
…
```

```
…
float x =
input[offset+threadID];
float y = func(x);
output[offsset+threadID]
       = y;
…
```

Thread Communication



```
…
float x =
input[threadID];
float y = func(x);
output[0] += y;
…
```

Thread communication

- threads within a block cooperate via
  - atomic operations on global memory or shared memory,
  - shared memory + barrier synchronization

Course Schedule

1:30 – 1:45:	Introduction
1:45 – 2:00:	Parallel programming primer
2:00 – 2:15:	GPU hardware
2:15 – 3:00:	CUDA API, threads level optimization (Ziyi)
	Coffee Break
3:30 – 4:00:	CUDA memory optimization (Eric)
4:00 – 4:15:	CUDA programming environment (Ziyi)
4:15 – 4:45:	Parallelism in medical image (Klaus)
4:45 – 5:25:	CT reconstruction examples (Eric + Ziyi)
5:25 – 5:30:	Closing remarks (Klaus)