Introduction to GPGPUs

Sandra Wienke, M.Sc.
wienke@rz.rwth-aachen.de
Center for Computing and Communication
RWTH Aachen University

PPCES, March 2013

Contents

Overview

- **GPGPUs = General Purpose Graphics Processing Units**

- **History – a very brief overview**

→ ‘80s - ‘90s: Development is mainly driven by games

Fixed-function 3D graphics pipeline

Graphics APIs like OpenGL, DirectX popular

→ Since 2001: Programmable pixel and vertex shader in graphics pipeline

(adjustments in OpenGL, DirectX)

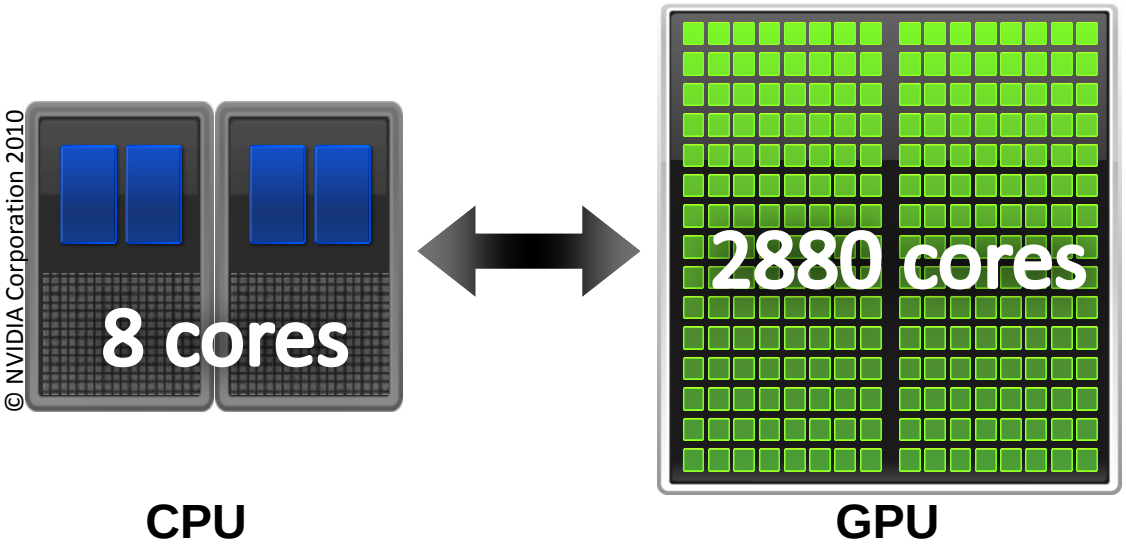
Researchers take notice of performance growth of GPUs: Tasks must be cast into native graphics operations

→ Since 2006: Vertex/pixel shader are replaced by a single processor unit

Support of programming language C, synchronization,…

→ “General purpose”

Comparison CPU ⇔ GPU

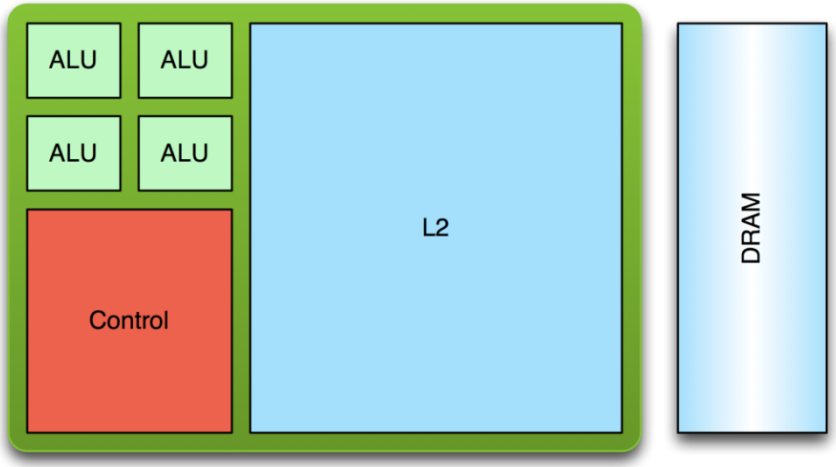


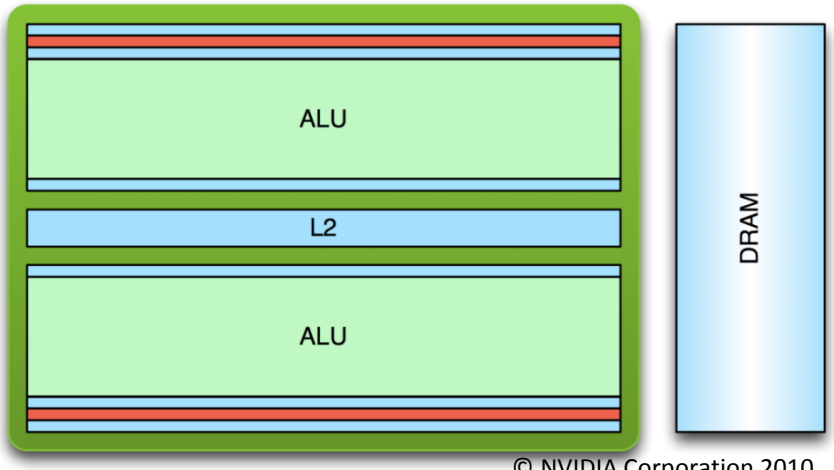
- **Massively Parallel Processors**
- **Manycore Architecture**

- **GPU-Threads**
  - → Thousands ("few" on CPU)
  - → Light-weight, little creation overhead
  - → Fast switching

Comparison CPU ⇔ GPU

- **Different design**





CPU

- → Optimized for low latencies
- → Huge caches
- → Control logic for out-of-order and speculative execution

GPU

- → Optimized for data-parallel throughput
- → Architecture tolerant of memory latency
- → More transistors dedicated to computation

Comparison CPU ⇔ GPU

- **Considerations for GPU parallelization**
  - → Hardware-related programming
    - → Knowledge of hardware essential
    - → Code modifications needed (low or high-level GPU languages)
  - → Very small shared memory
  - → Global synchronization not possible within one kernel
  - → Number of suitable problems limited

- **Why GPGPUs?**

Motivation for GPUs

- **Performance: High rate of Flops achievable!**
  - → Little overhead (threads), 1000s of threads
- **(Massive) data parallelism in application**
  - → Independent data
  - → Uniform operations
- **Heterogeneous computer architecture (CPU + GPU)**
  - → Asynchronous computations, overlapping
  - → OpenMP/MPI + GPU parallelization
- **Relative low cost + power consumption (→ "GreenIT")**
  - → Compared to computers/clusters having a similar performance
- **GPU available in almost every computer**

Some (programmable) GPU types

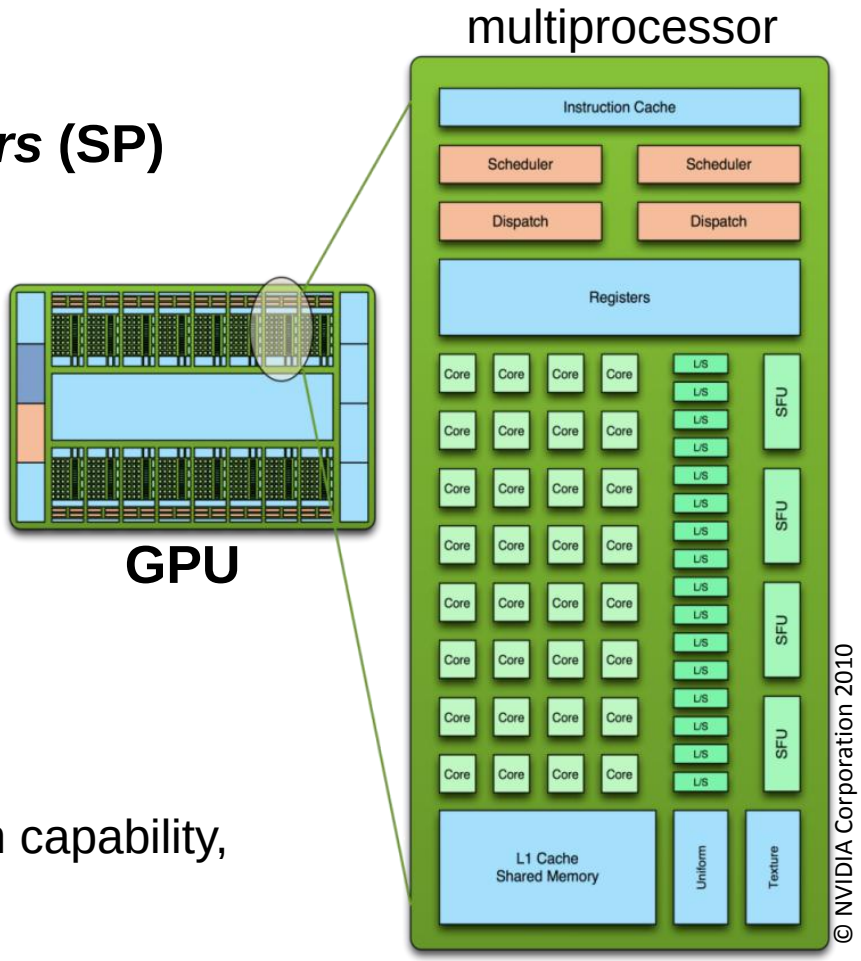
- **NVIDIA**
  - → GeForce: 8800GTX, GT220, GTX 470, …
  - → Quadro: 6000, FX 4800, NVS 450, …
  - → Tesla: C870, C1060, C2050, K20, …
- **AMD**
  - → Radeon: HD 3870, HD 5850, …
  - → FirePro: 3D V3800, 3D V9800, S10000, …
  - → FireStream: 9350, …

Here we will go into NVIDIA-GPUs.

Contents

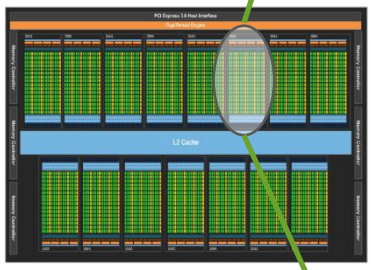
GPU architecture: Fermi

- **3 billion transistors**
- **14-16 *streaming multiprocessors* (SM)**
  - → Each comprises 32 cores
- **448-512 *cores*/ *streaming processors* (SP)**
  - → i.a. Floating point & integer unit
- **Memory hierarchy**
- **Peak performance**
  - → SP: 1.03 TFlops
  - → DP: 515 GFlops
- **ECC support**
- **Compute capability: 2.0**
  - → Defines features, e.g. double precision capability, memory access pattern



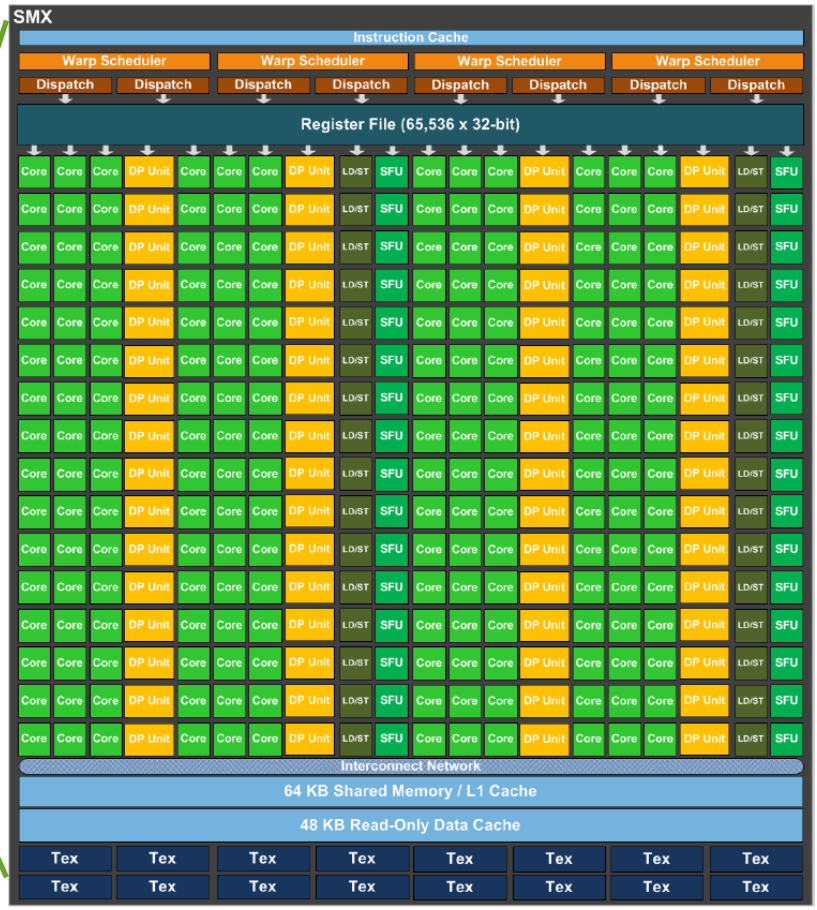
GPU architecture: Kepler K20 (GK110)

- **7.1 billion transistors**
- **13-15 streaming multiprocessors extreme (SMX)**
  - → Each comprises 192 cores
- **2496-2880 cores**
- **Memory hierarchy**
- **Peak performance**
  - → SP: 3.52 TFlops
  - → DP: 1.17 TFlops
- **ECC support**
- **Compute capability: 3.5**
  - → E.g. dynamic parallelism = possibility to launch dynamically new work from GPU



GPU

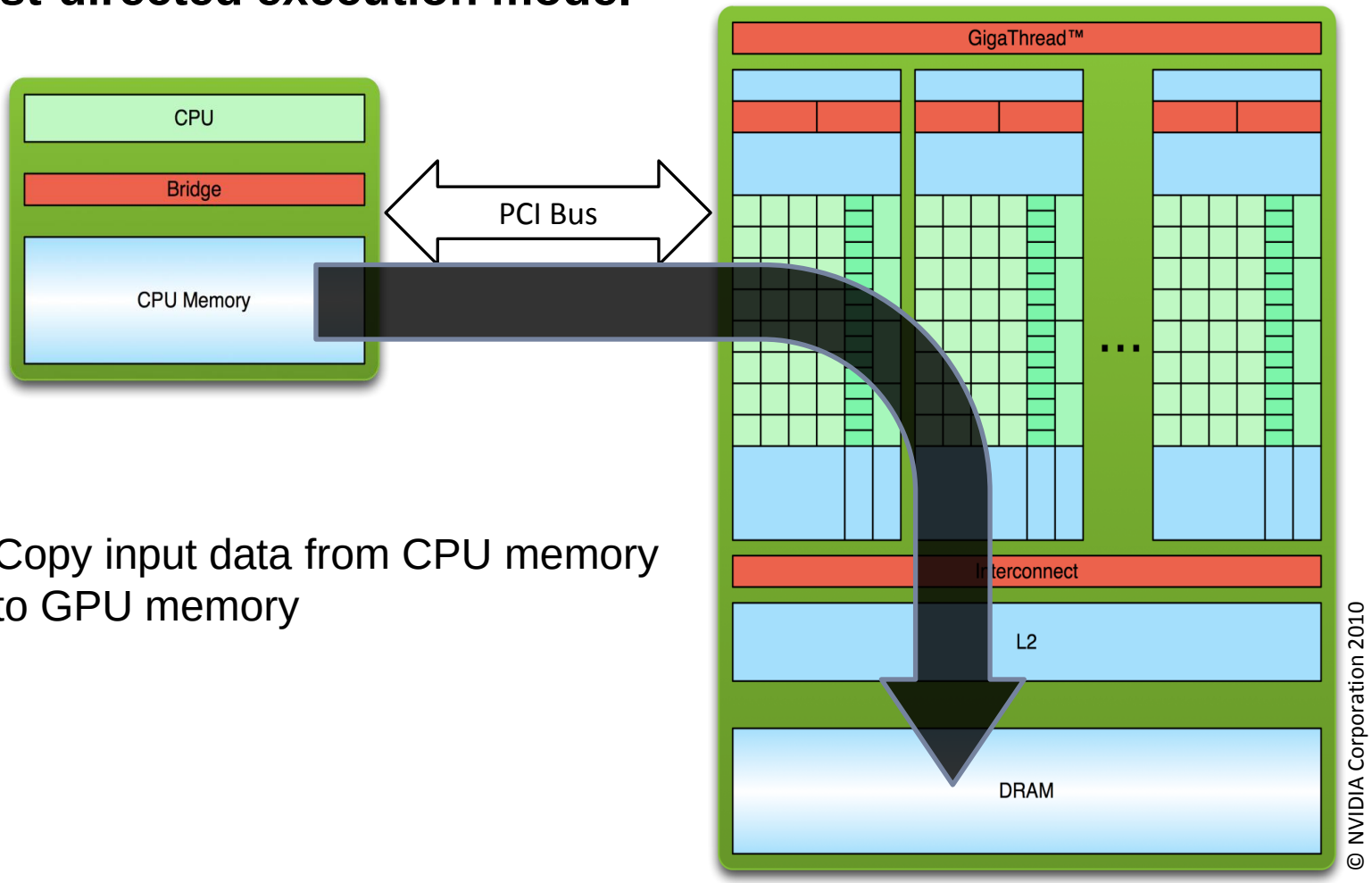
SMX



http://www.nvidia.com/content/PDF/kepler/NVIDIA-Kepler-GK110-Architecture-Whitepaper.pdf

Processing flow

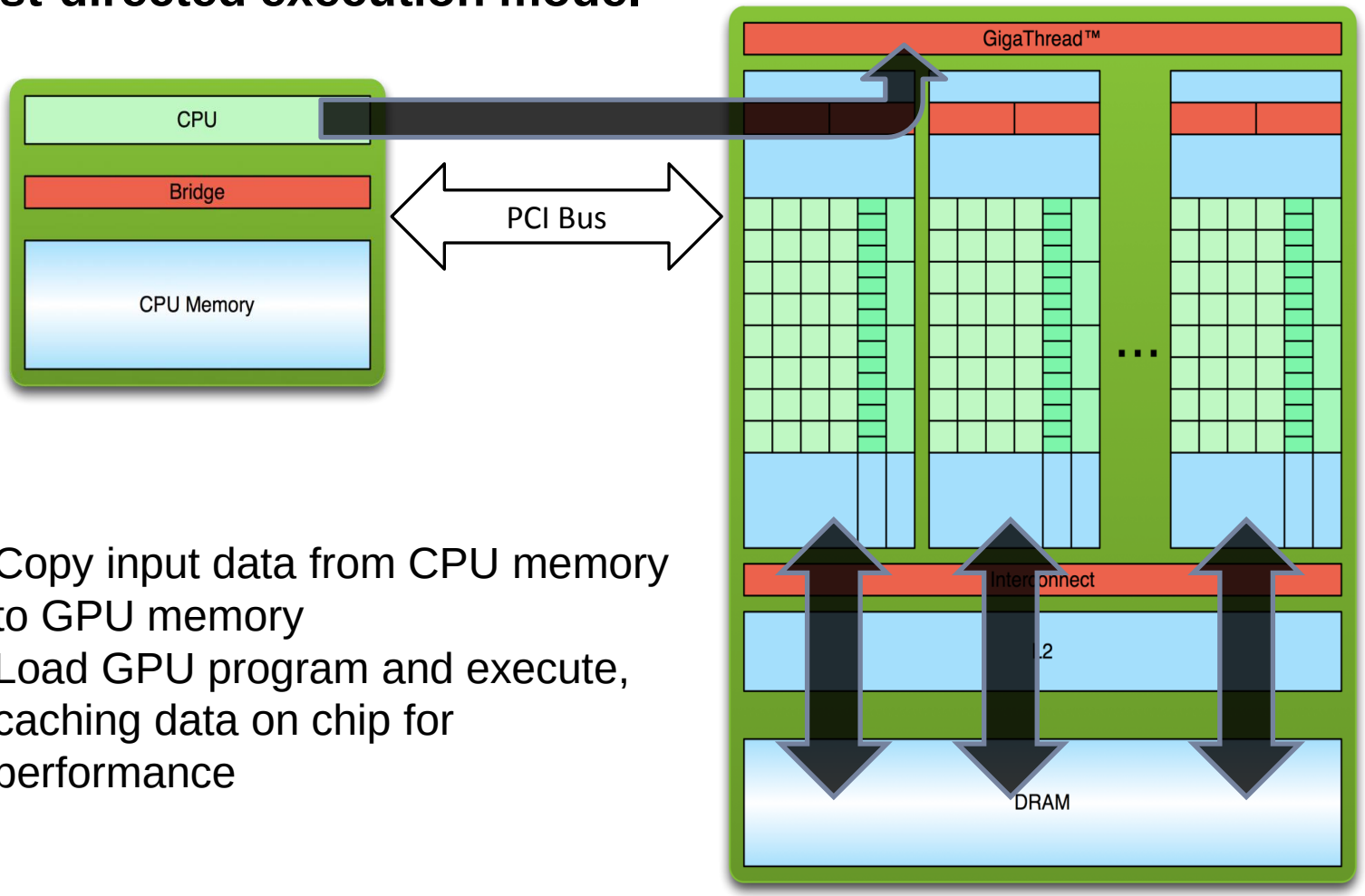
- **Host-directed execution model**



1. Copy input data from CPU memory to GPU memory

Processing flow

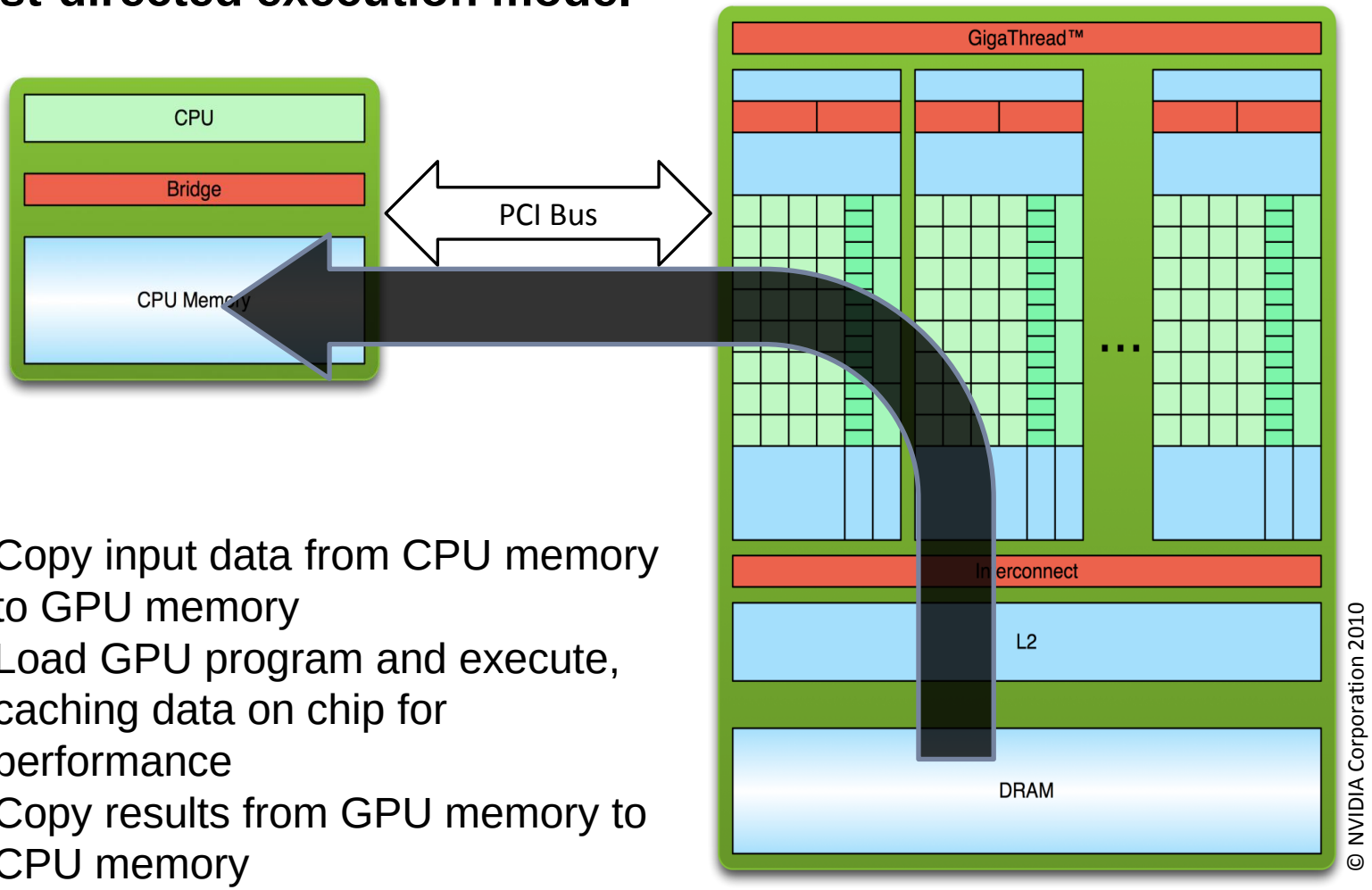
- **Host-directed execution model**



1. Copy input data from CPU memory to GPU memory
2. Load GPU program and execute, caching data on chip for performance

Processing flow

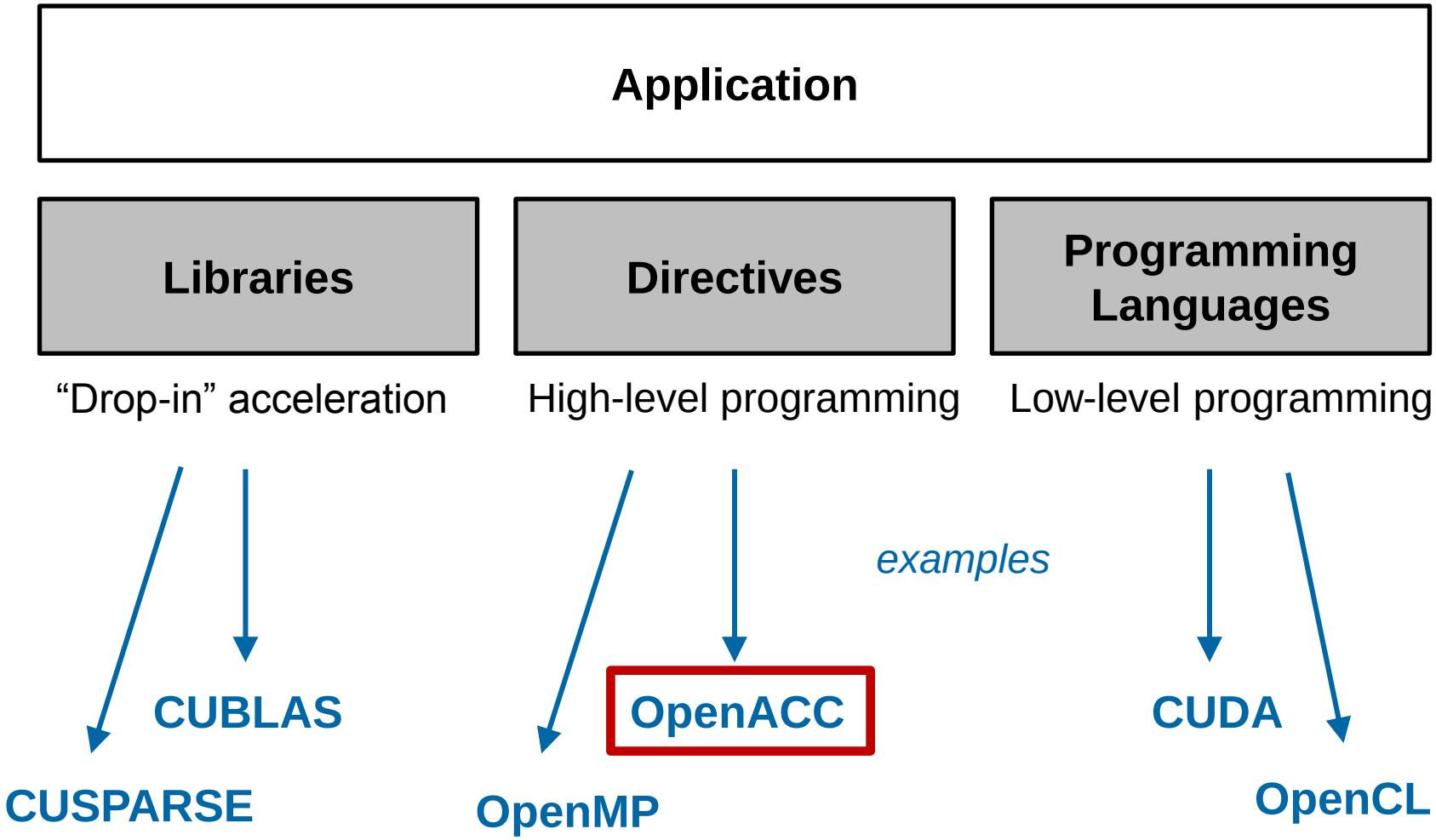
- **Host-directed execution model**



1. Copy input data from CPU memory to GPU memory
2. Load GPU program and execute, caching data on chip for performance
3. Copy results from GPU memory to CPU memory

Contents

GPGPU programming



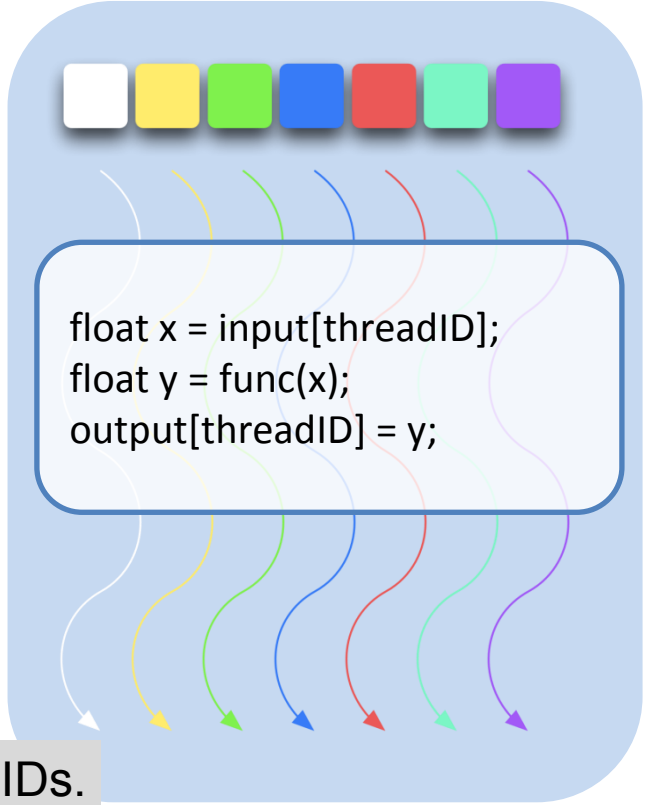
GPGPU programming

- **CUDA (Compute Unified Device Architecture)**
  - → C/C++ (NVIDIA): architecture + programming language, NVIDIA GPUs
  - → Fortran (PGI): NVIDIA‘s CUDA for Fortran, NVIDIA GPUs
- **OpenCL**
  - → C (Khronos Group): open standard, portable, CPU/GPU/…
- **OpenACC**
  - → C/Fortran (PGI, Cray, CAPS, NVIDIA): Directive-based accelerator programming, industry standard published in Nov. 2011 (NVIDIA GPUs)
- **OpenMP**
  - → C/C++, Fortran: Directive-based programming for hosts and accelerators, standard, portable, announced in Q1 2013, implementations soon
- **…**

Programming model

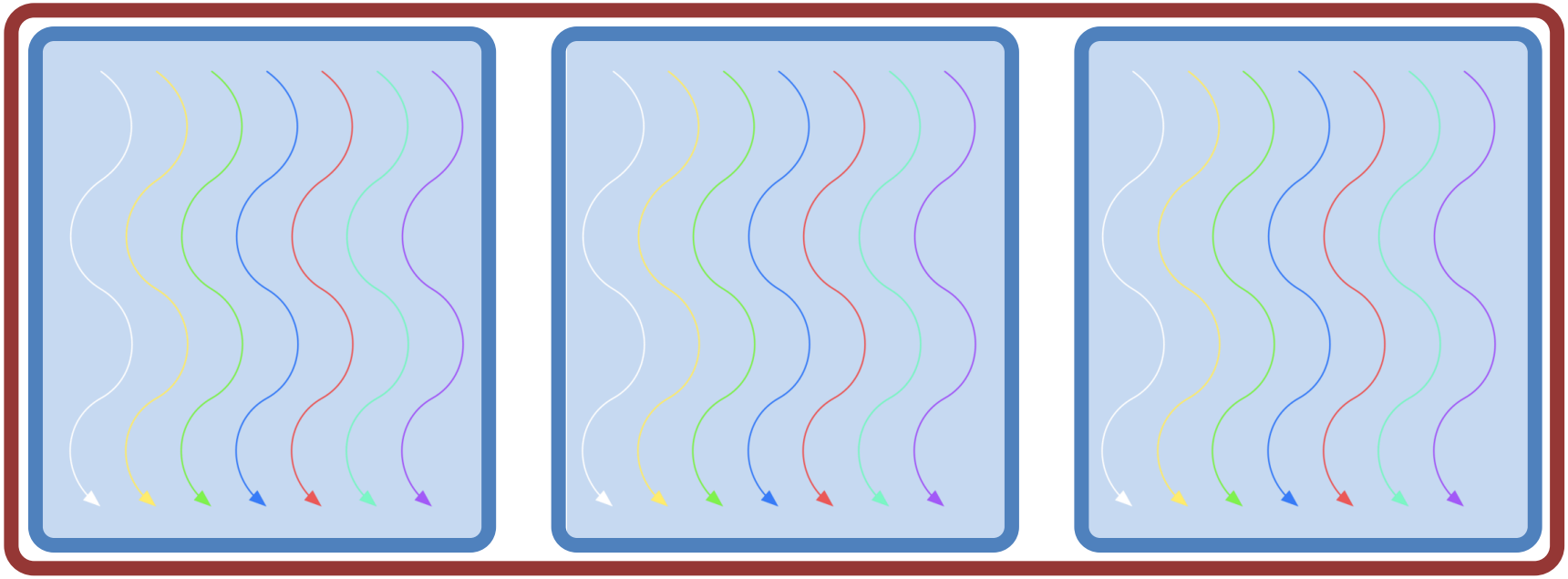
- **Definitions**
  - → Host: CPU, executes functions
  - → Device: usually GPU, executes kernels

- **Parallel portion of application executed on device as kernel**
  - → Kernel is executed as array of threads
  - → All threads execute the same code
  - → Threads are identified by IDs
    - → Select input/output data
    - → Control decisions



With OpenACC, you don't have to bother with thread IDs.

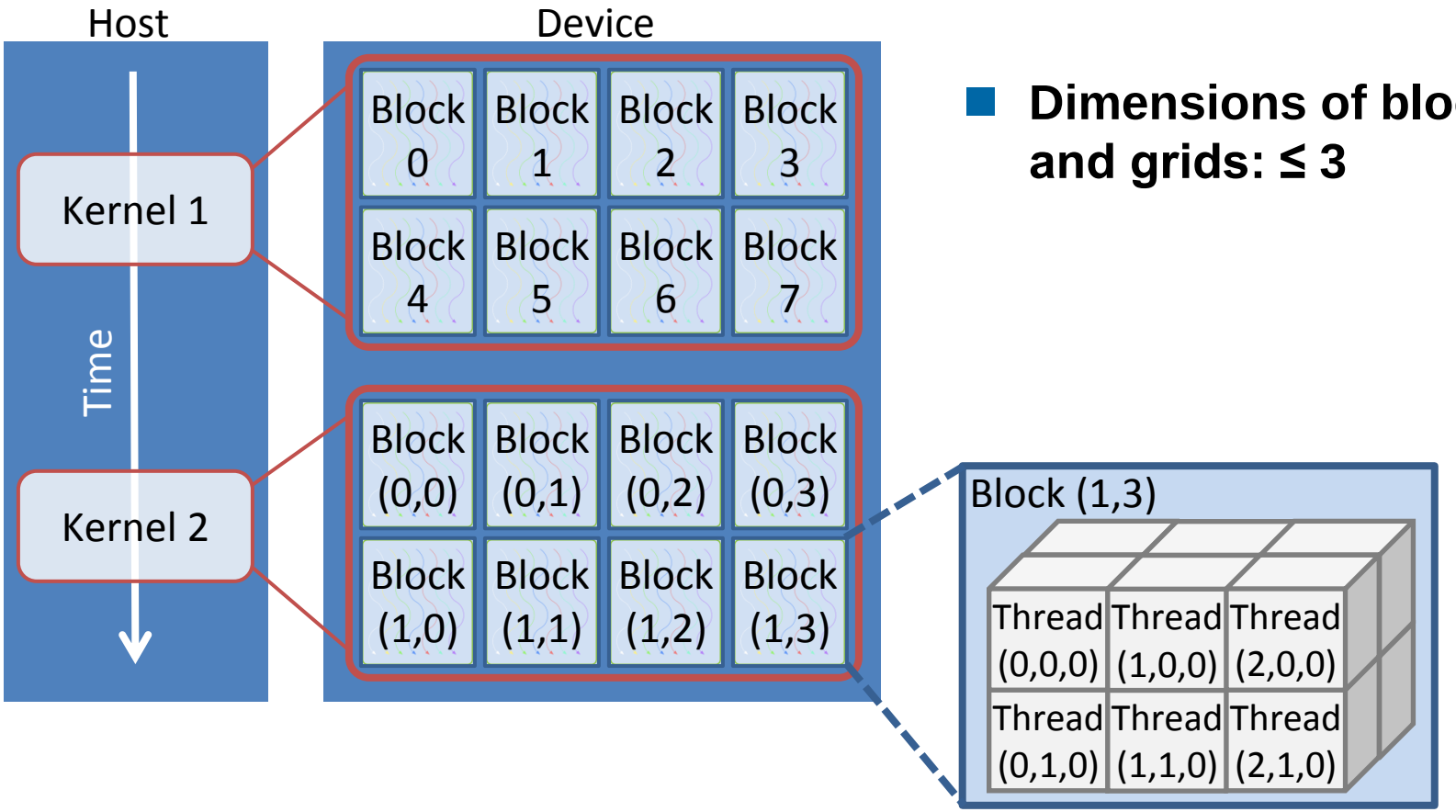
Programming model



- **Threads are grouped into *blocks***
- **Blocks are grouped into a *grid***

Programming model

- **Kernel is executed as a grid of blocks of threads**

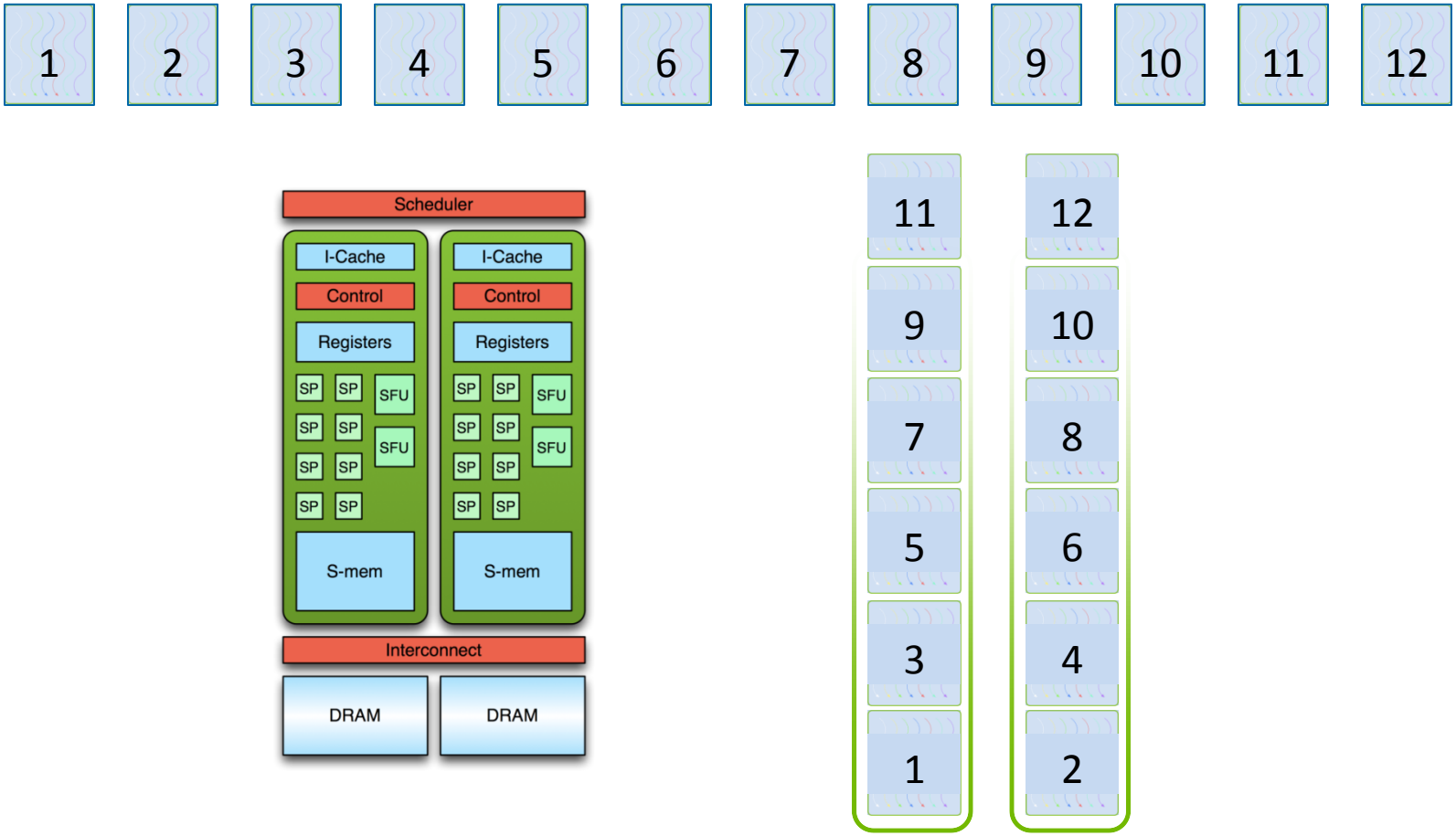


- **Dimensions of blocks and grids: ≤ 3**

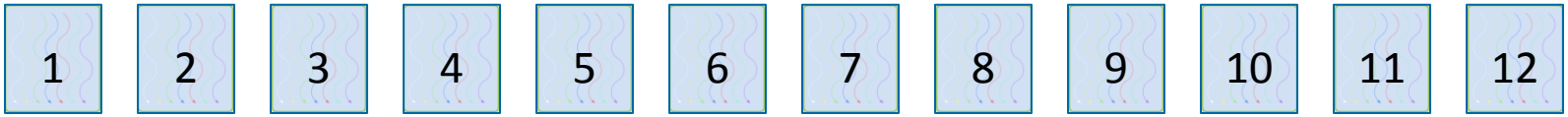
Programming model

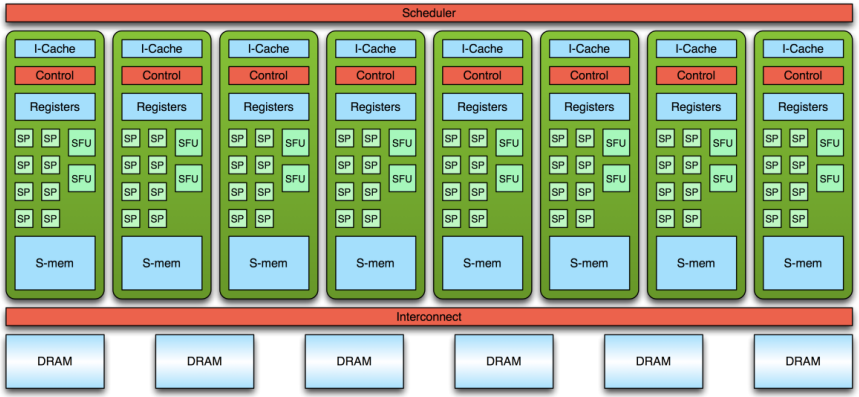
- **Why blocks?**
  - → Cooperation of threads within a block possible
    - → Synchronization (barrier)
    - → Share data/ results using shared memory
  - → Scalability
    - → Fast communication between n threads is not feasible when n large
    - → But: blocks are executed independently
    - ➔ Blocks can be distributed across arbitrary number of multiprocessors
      - → In any order
      - → Concurrently
      - → Sequentially

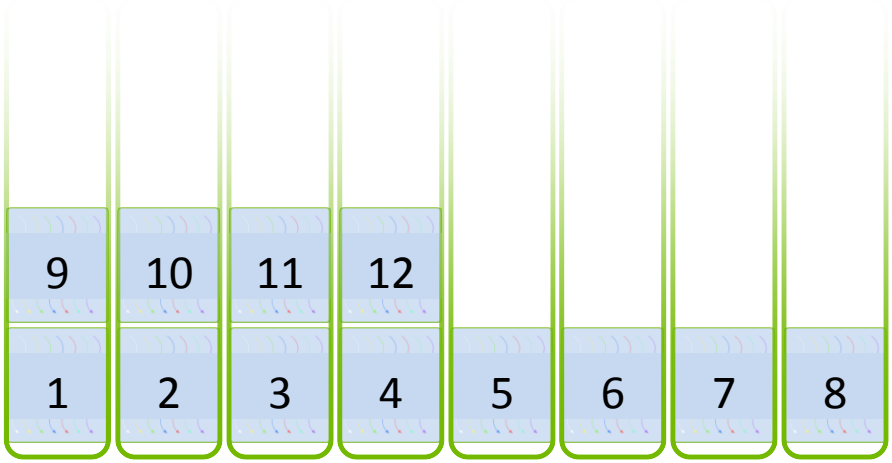
Programming model: scalability – G84 (very old architecture)



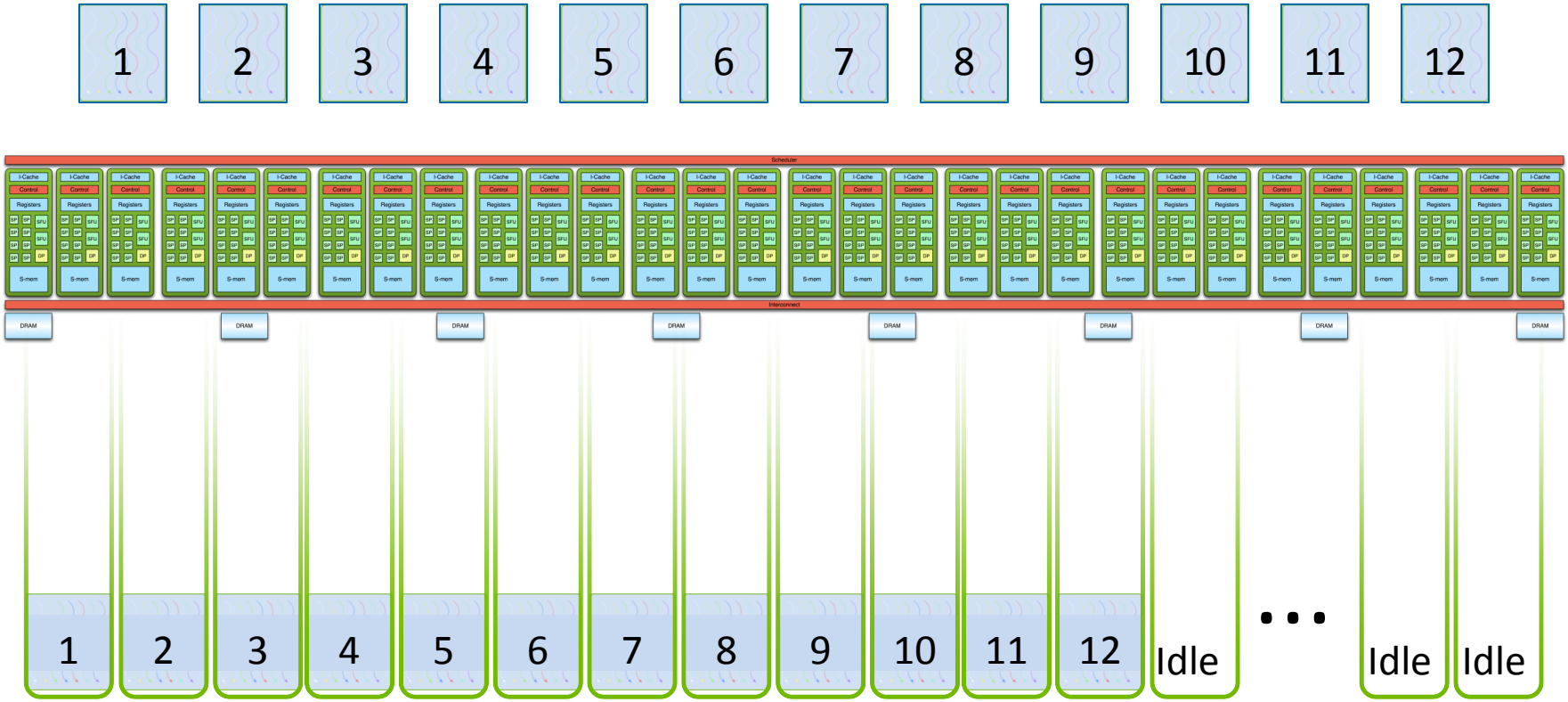
Programming model: scalability – G80 (medium old architecture)







Programming model: scalability – GT200 (old architecture)



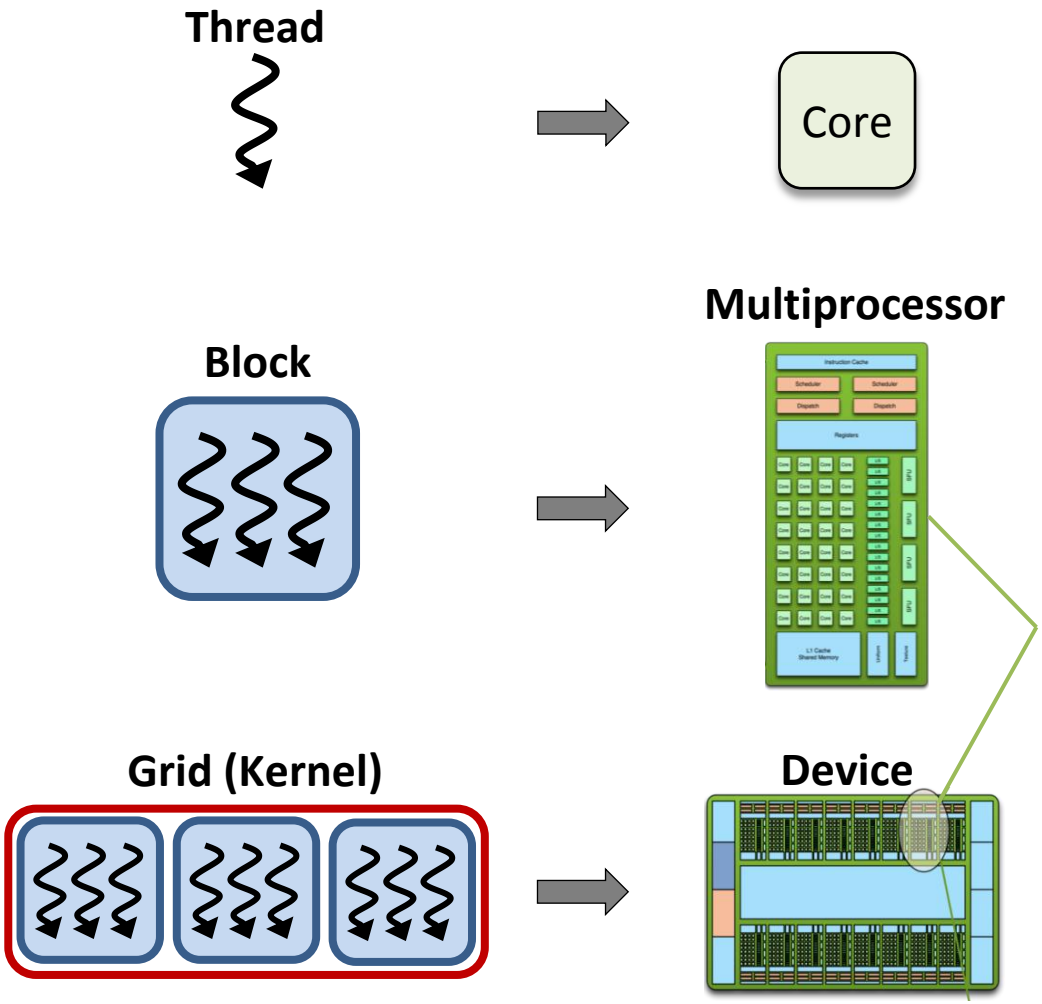
Contents

Execution model

- **Host-directed execution model**
  - → Main program runs on host
  - → Certain code regions run on device
- **Launch configuration: blocks per grid, threads per block**
- **Warps**
  - → Threads execute as groups of 32
  - → Threads in warp share same program counter

→ **Single instruction multiple threads (SIMT)**

Execution model



→ Each thread is executed by a core

→ Each block is executed on a multiprocessor

→ Several concurrent blocks can reside on a MP depending on shared resources

→ Each kernel is executed on a device

Contents

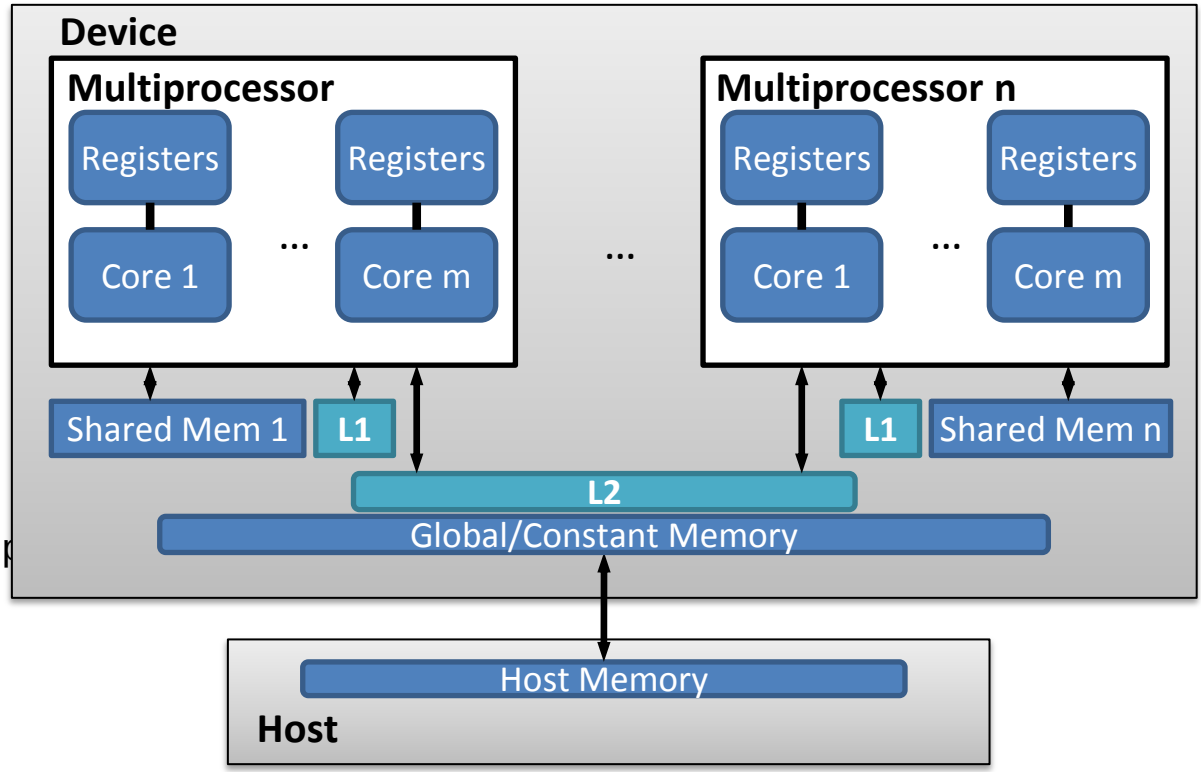
Memory model

- **Host + device memory = separate entities**
- **No coherence between host + device**
  - ➔ Data synchronization/transfer

- **Host**
  - → (De-)Allocates device memory (global, constant, texture)
  - → Triggers data transfer
- **Device**
  - → Works on device memory (hierarchy)

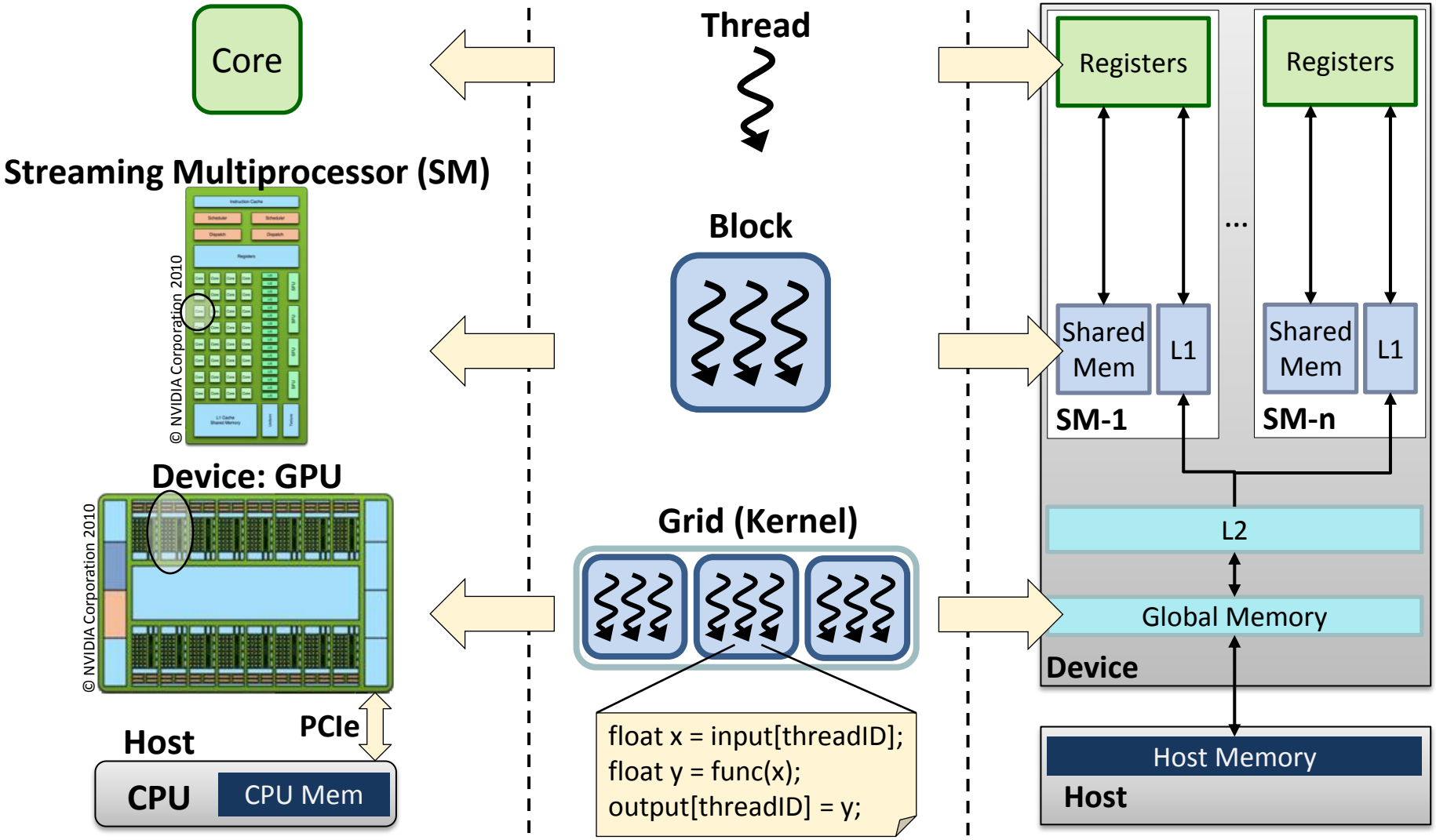
Memory model

- **Thread**
  - → *Registers*
    Fermi: 63 per thread
    Kepler2: 255 per thread
  - → *Local* memory
- **Block**
  - → *Shared* memory
    Fermi:64KB configurable,on-chip
    16KB shared +48KB L1 OR
    48KB shared +16KB L1
    Kepler2:64KB configurable,on-chip
    16KB shared +48KB L1 OR
    48KB shared +16KB L1 OR
    32KB shared +32KB L1
- **Grid/ application**
  - → *Constant* memory
    read-only; off-chip; cached
  - → *Global* memory
    several GB; off-chip
    Fermi/Kepler2: L2 cache



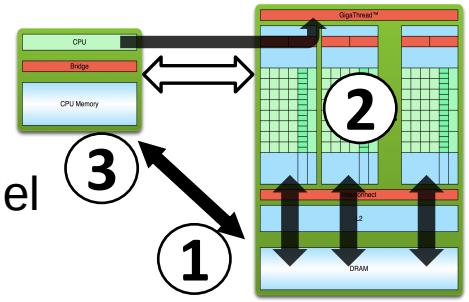
Contents

Summary


Core
Thread
Registers
Registers
Streaming Multiprocessor (SM)
© NVIDIA Corporation 2010
Block
...
Shared Mem
L1
Shared Mem
L1
SM-1
SM-n
Device: GPU
© NVIDIA Corporation 2010
Grid (Kernel)
L2
Global Memory
Device
PCIe
Host
CPU
CPU Mem
float x = input[threadID];
float y = func(x);
output[threadID] = y;
Host Memory
Host

Summary & General recommendation

- **Processing flow**
  - → Copy data from host to device
  - → Execute GPU code (kernel) in parallel
  - → Copy data from device to host



- **Launch many many threads**
  - → Think about threads per thread block
- **Use uniform operations on data**
- **See tuning slides**
- **Use tools**
  - → Debugger
  - → Profiler