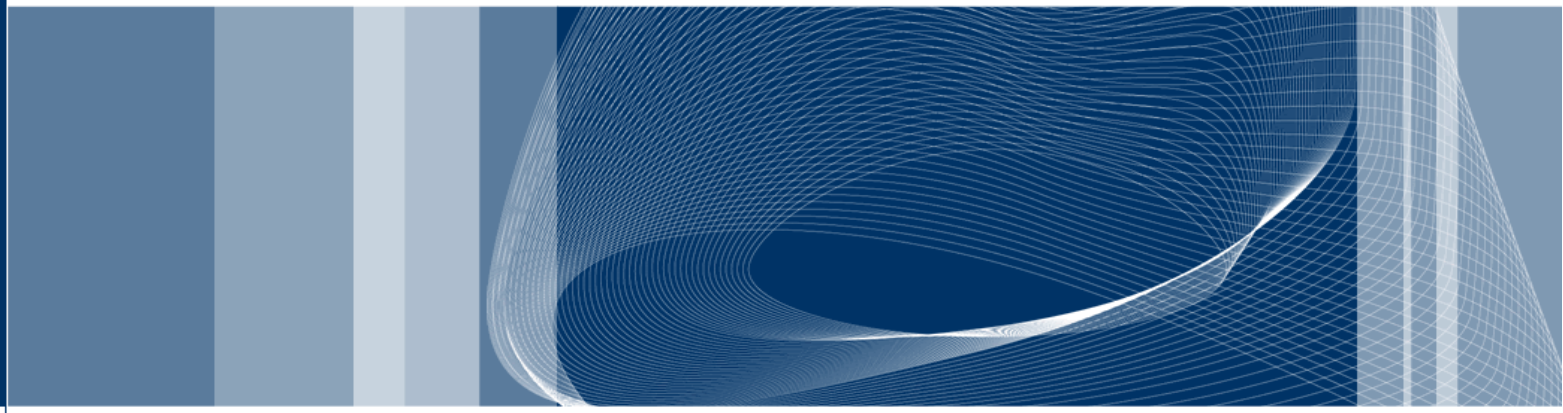
POLITECNICO DI MILANO



Introduction to CUDA

Algoritmi e Calcolo Parallelo

Daniele Loiacono

References

- This set of slides is mainly based on:
  - CUDA Technical Training, Dr. Antonino Tumeo, Pacific Northwest National Laboratory
  - Slide of *Applied Parallel Programming* (ECE498@UIUC) http://courses.engr.illinois.edu/ece498/al/
- Useful references
  - *Programming Massively Parallel Processors: A Hands-on Approach*, David B. Kirk and Wen-mei W. Hwu
  - http://www.gpgpu.it/ (CUDA Tutorial)

GPGPU

What is (Historical) GPGPU ?

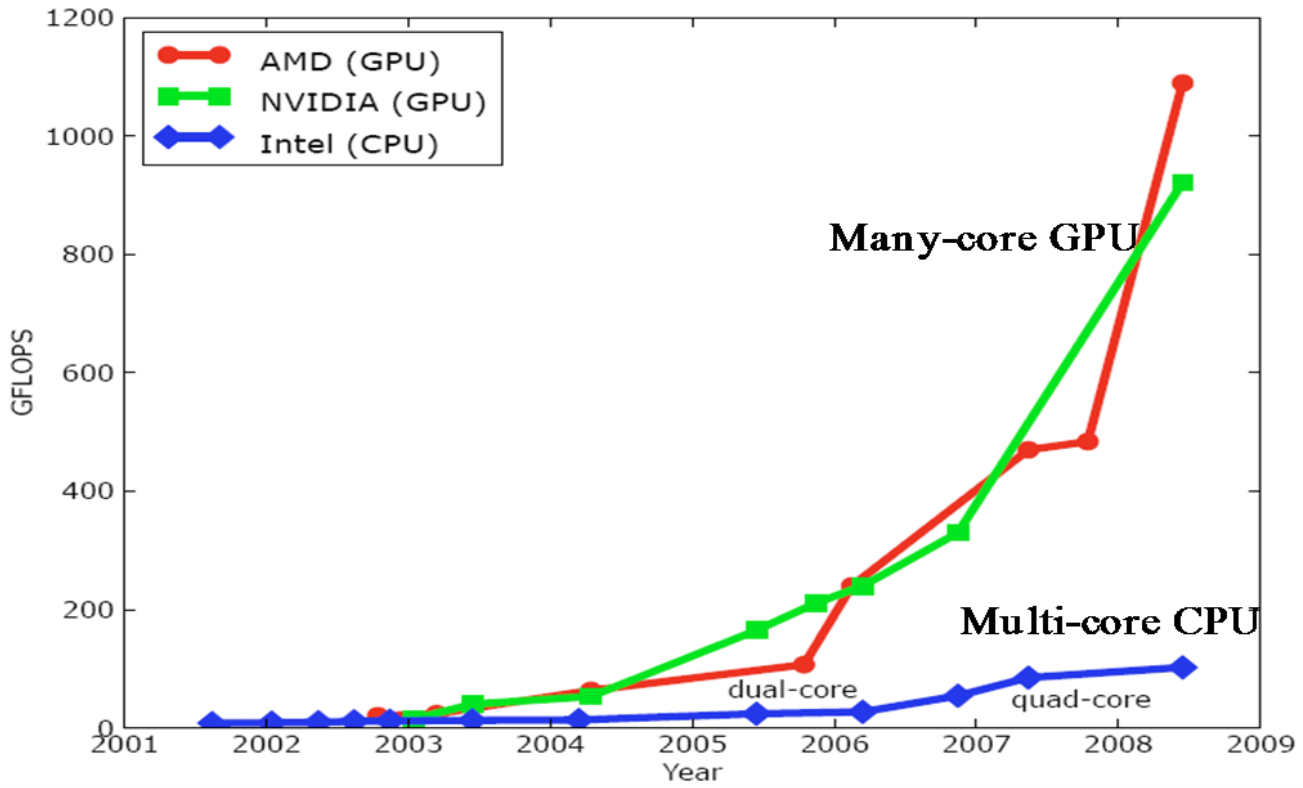
- General Purpose computation using GPU and graphics API in applications other than 3D graphics
  - GPU accelerates critical path of application
- Data parallel algorithms leverage GPU attributes
  - Large data arrays, streaming throughput
  - Fine-grain SIMD parallelism
  - Low-latency floating point (FP) computation
- Applications – see //GPGPU.org
  - Game effects (FX) physics, image processing
  - Physical modeling, computational engineering, matrix algebra, convolution, correlation, sorting

GPGPU Constraints

- Dealing with graphics API
  - Working with the corner cases of the graphics API
- Addressing modes
  - Limited texture size/dimension
- Shader capabilities
  - Limited outputs
- Instruction sets
  - Lack of Integer & bit ops
- Communication limited
  - Between pixels

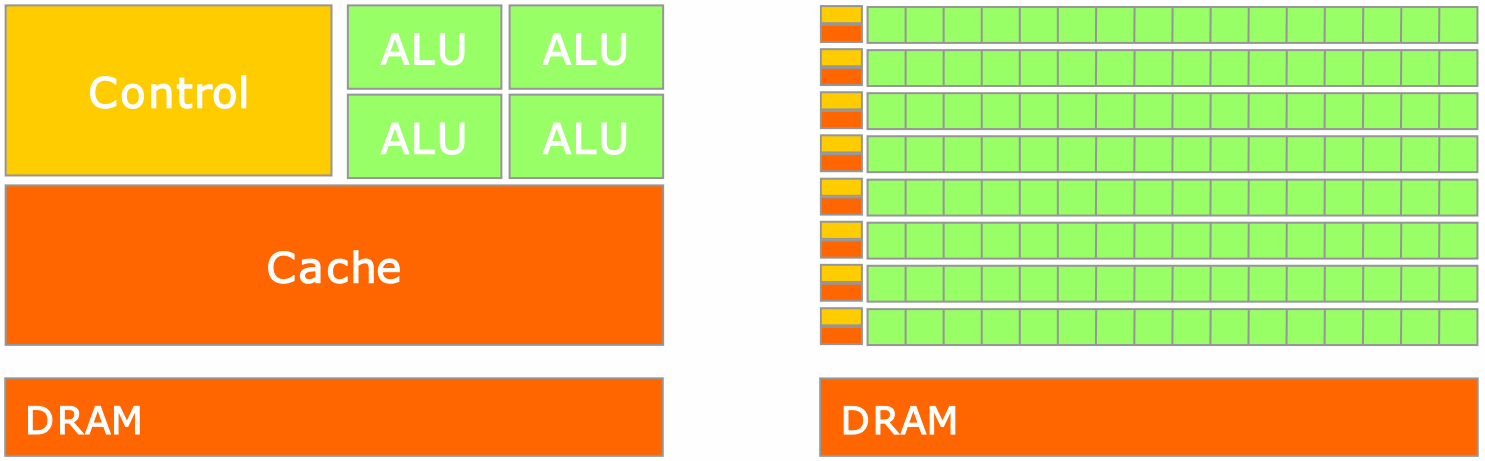
Why GPUs?

- The GPU has evolved into a very flexible and powerful processor:
  - It's programmable using high-level languages
  - Now supports 32-bit and 64-bit floating point IEEE-754 precision
  - It offers lots of GFLOPS
- GPU in every PC and workstation



What is behind such an evolution?

- The GPU is specialized for compute-intensive, highly parallel computation (exactly what graphics rendering is about)
  - So, more transistors can be devoted to data processing rather than data caching and flow control



CPU GPU

- The fast-growing video game industry exerts strong economic pressure that forces constant innovation

Application Domains

Massive Data Parallelism

Instruction Level Parallelism

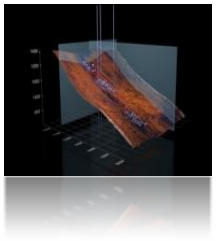
Graphics

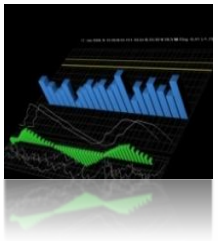
GPU
(Parallel Computing)

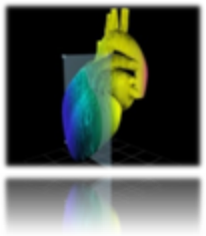
CPU
(Sequential Computing)

Data Fits in Cache

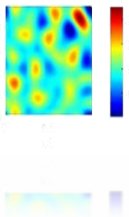
Larger Data Sets


Oil & Gas


Finance


Medical


Biophysics


Numerics

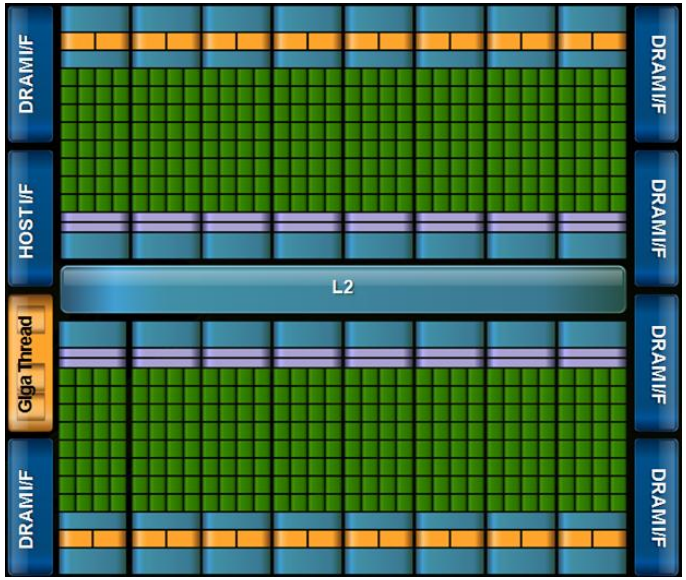

Audio


Video


Imaging

GPUs

- Each NVIDIA GPU has up to 448 parallel cores
- Within each core
  - Floating point unit
  - Logic unit (add, sub, mul, madd)
  - Move, compare unit
  - Branch unit
- Cores managed by thread manager
  - Thread manager can spawn and manage
    12,000+ threads per core
  - Zero overhead thread switching

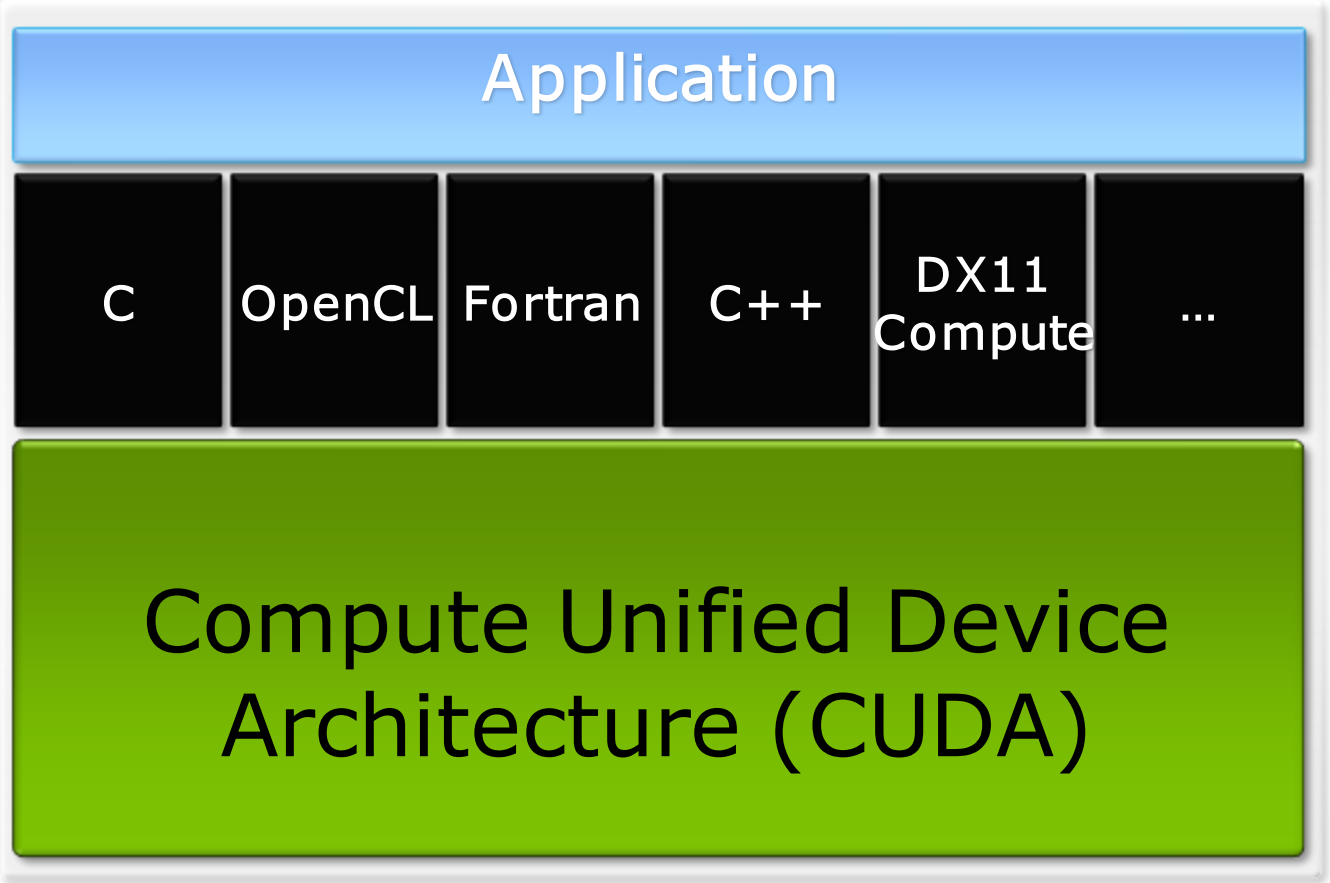


NVIDIA Fermi Architecture

CUDA

CUDA Parallel Computing Architecture

- Parallel computing architecture and programming model
- Includes a C compiler plus support for OpenCL and DX11 Compute
- Architected to natively support all computational interfaces (standard languages and APIs)
- NVIDIA GPU architecture accelerates CUDA
  - Hardware and software designed together for computing
  - Expose the computational horsepower of NVIDIA GPUs
  - Enable general-purpose GPU computing

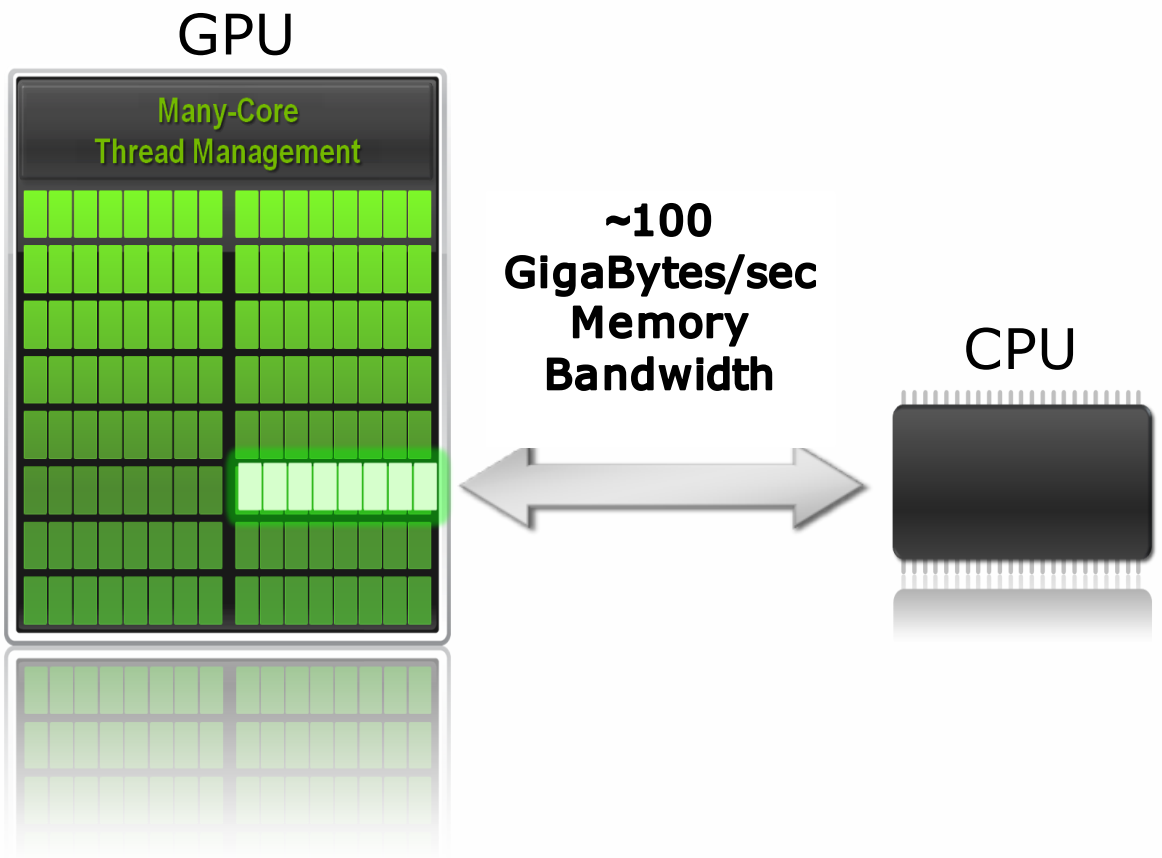


CUDA is C for Parallel Processors

- CUDA is industry-standard C with minimal extensions
  - Write a program for one thread
  - Instantiate it on many parallel threads
  - Familiar programming model and language
- CUDA is a scalable parallel programming model
  - Program runs on any number of processors without recompiling
- CUDA parallelism applies to both CPUs and GPUs
  - Compile the same program source to run on different platforms with widely different parallelism
  - Map to CUDA threads to GPU threads or to CPU vectors

A Highly Multithreaded Coprocessor

- The GPU is a highly parallel compute coprocessor
  - serves as a coprocessor for the host CPU
  - has its own device memory with high bandwidth interconnect



CUDA Uses Extensive Multithreading

- CUDA threads express fine-grained data parallelism
  - Map threads to GPU threads
  - Virtualize the processors
  - You must rethink your algorithms to be aggressively parallel
- CUDA thread blocks express coarse-grained parallelism
  - Blocks hold arrays of GPU threads, define shared memory boundaries
  - Allow scaling between smaller and larger GPUs
- GPUs execute thousands of lightweight threads
  - (In graphics, each thread computes one pixel)
  - One CUDA thread computes one result (or several results)
  - Hardware multithreading & zero-overhead scheduling

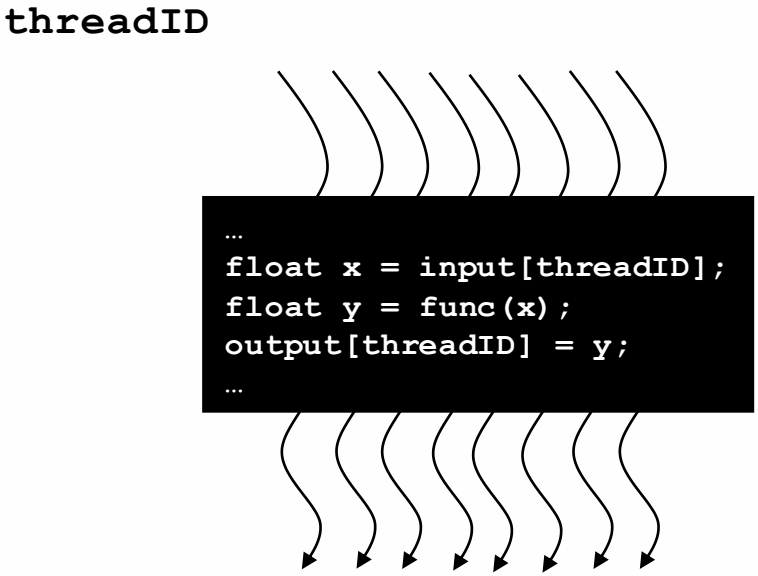
CUDA Kernels and Threads

- Parallel portions of an application are executed on the device as kernels
  - One kernel is executed at a time
  - Many threads execute each kernel
- Differences between CUDA and CPU threads
  - CUDA threads are extremely lightweight
    - Very little creation overhead
    - Instant switching
  - CUDA uses 1000s of threads to achieve efficiency
    - Multi-core CPUs can use only a few

Definitions
Device = GPU
Host = CPU
Kernel = function called from the host that runs on the device

Arrays of Parallel Threads

- A CUDA kernel is executed by an array of threads
  - All threads run the same program, SIMT (Singe Instruction multiple threads)
  - Each thread uses its ID to compute addresses and make control decisions

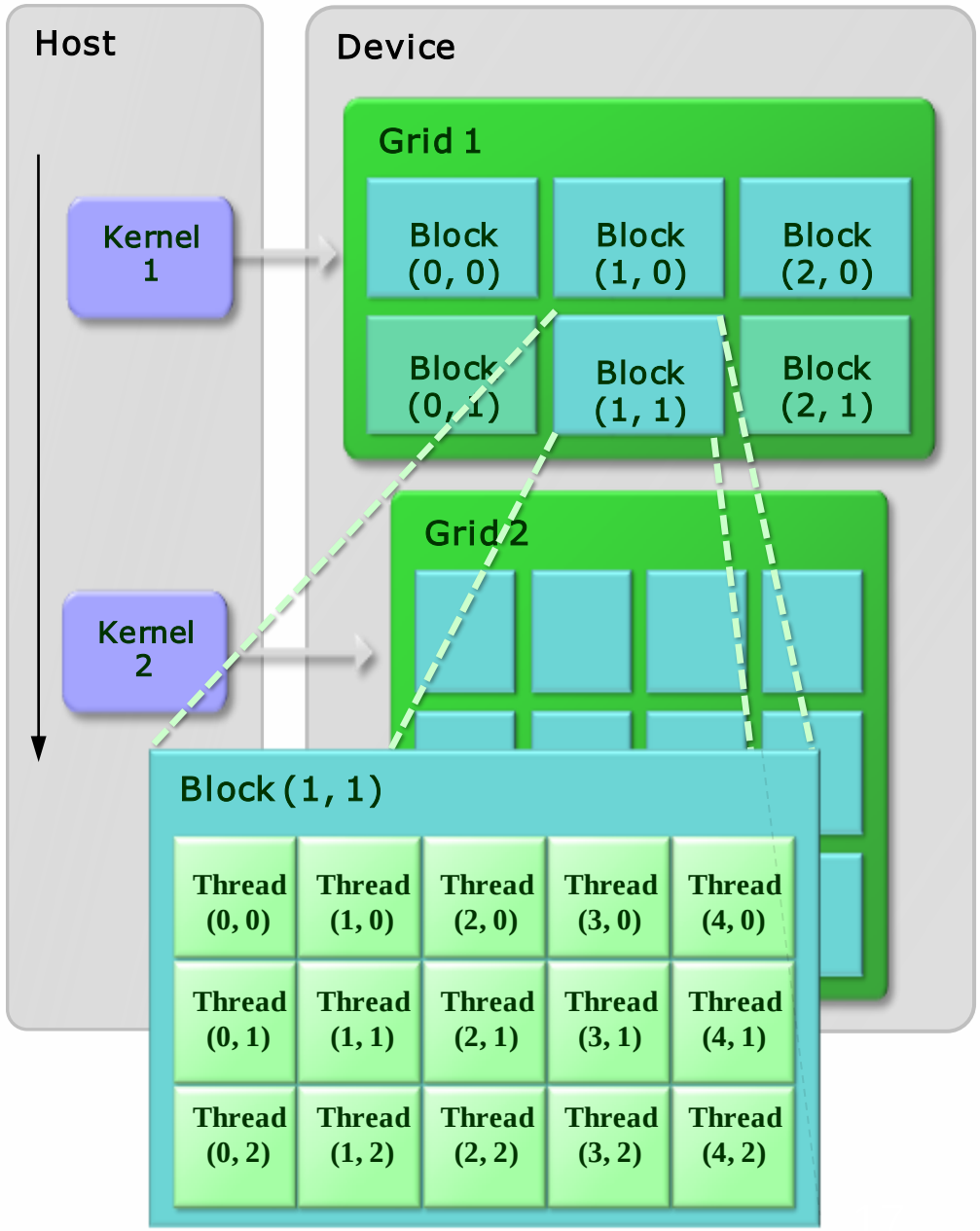


CUDA Programming Model

A kernel is executed by a grid, which contain blocks.

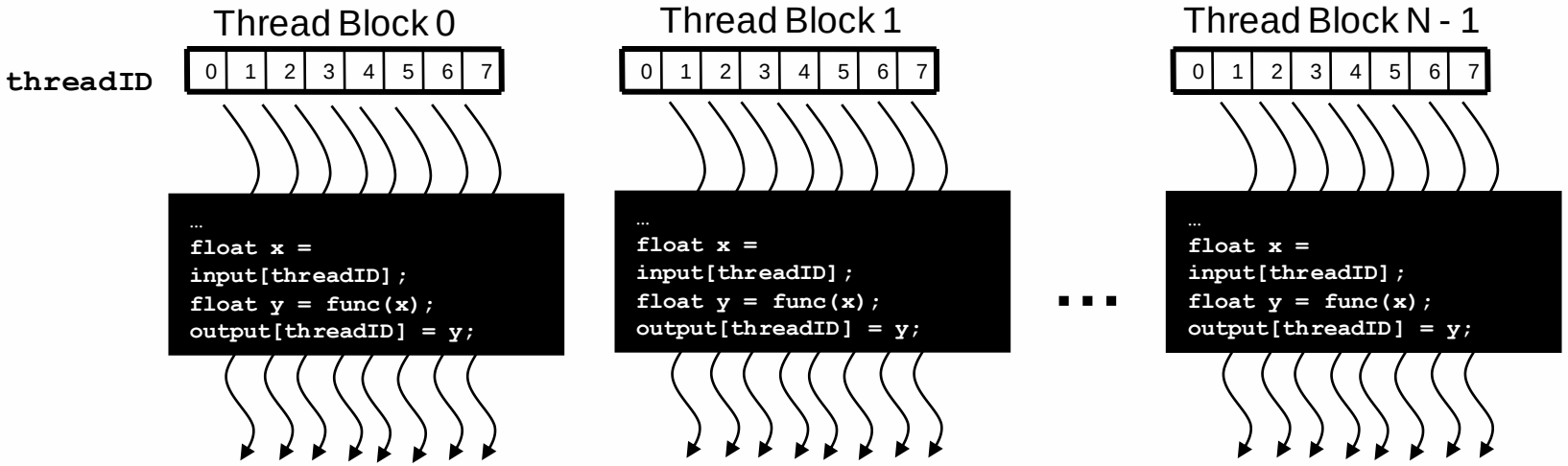
These blocks contain our threads.

- A thread block is a batch of threads that can cooperate:
  - Sharing data through shared memory
  - Synchronizing their execution
- Threads from different blocks operate independently



Thread Blocks: Scalable Cooperation

- Divide monolithic thread array into multiple blocks
  - Threads within a block cooperate via shared memory
  - Threads in different blocks cannot cooperate
- Enables programs to transparently scale to any number of processors!

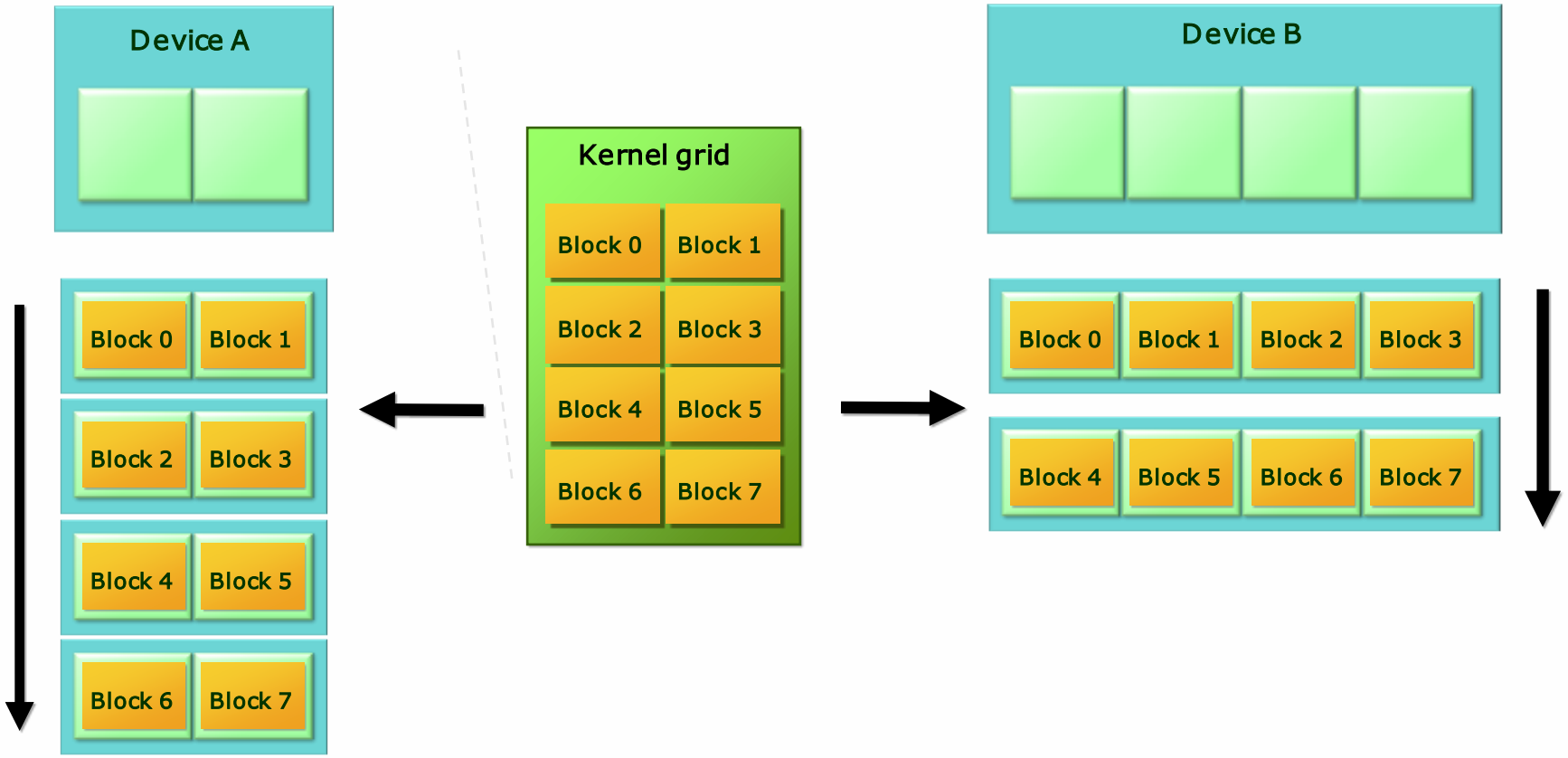


Thread Cooperation

- Thread cooperation is a powerful feature of CUDA
  - Threads can cooperate via on-chip shared memory and synchronization
- The on-chip shared memory within one block allows:
  - Share memory accesses, drastic *memory bandwidth reduction*
  - Share intermediate results, thus: *save computation*
- Makes algorithm porting to GPUs a *lot* easier (vs. GPGPU and its strict stream processor model)

Transparent Scalability

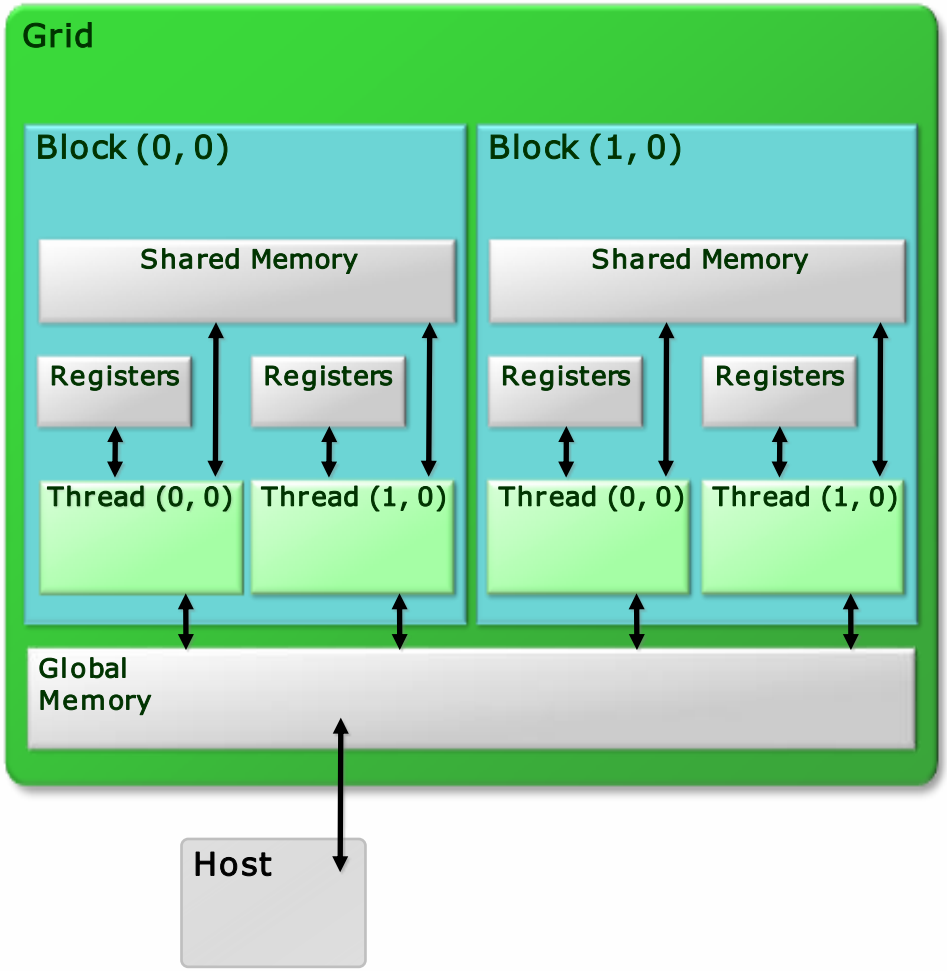
- Hardware is free to schedule thread blocks on any processor
  - Kernels scale to any number of parallel multiprocessors



Memory model seen from CUDA Kernel

- Registers (per thread)
- Shared Memory
  - Shared among threads in a single block
  - On-chip, small
  - As fast as registers
- Global Memory
  - Kernel inputs and outputs reside here
  - Off-chip, large
  - Uncached (use coalescing)

Note: The host can read & write global memory but not shared memory



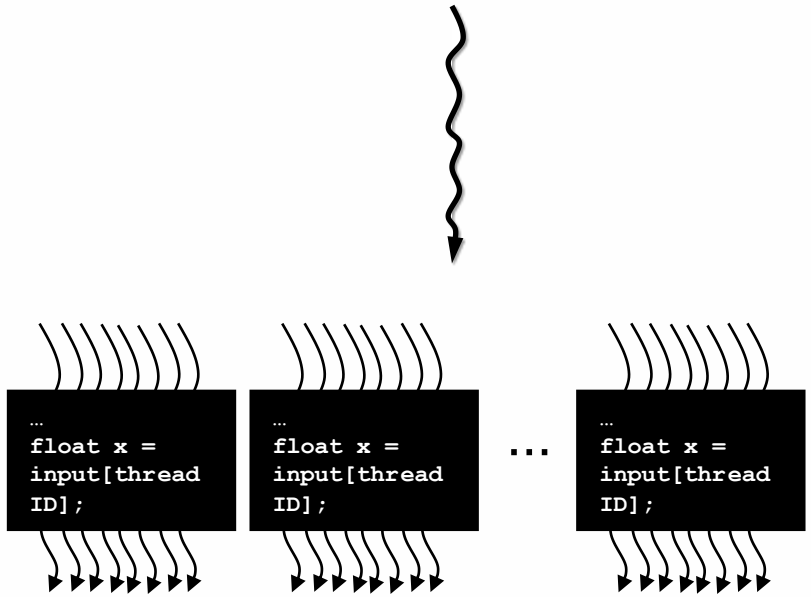
Execution Model

- Kernels are launched in grids
  - One kernel executes at a time
- A block executes on one multiprocessor
  - Does not migrate
- Several blocks can reside concurrently on one multiprocessor
  - Number is limited by multiprocessor resources
    - Register file is partitioned among all resident threads
    - Shared memory is partitioned among all resident thread blocks

Heterogeneous programming in CUDA

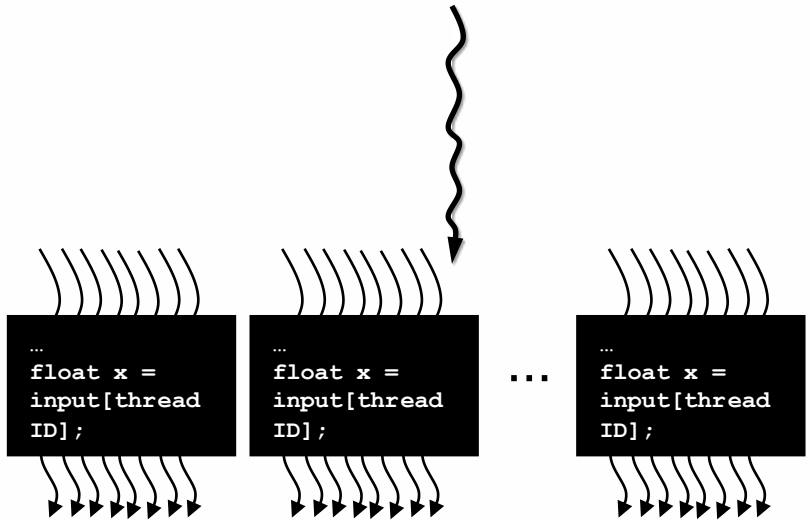
Serial Code

Parallel Kernel
KernelA<<< nBlk, nTid >>>(args);



Serial Code

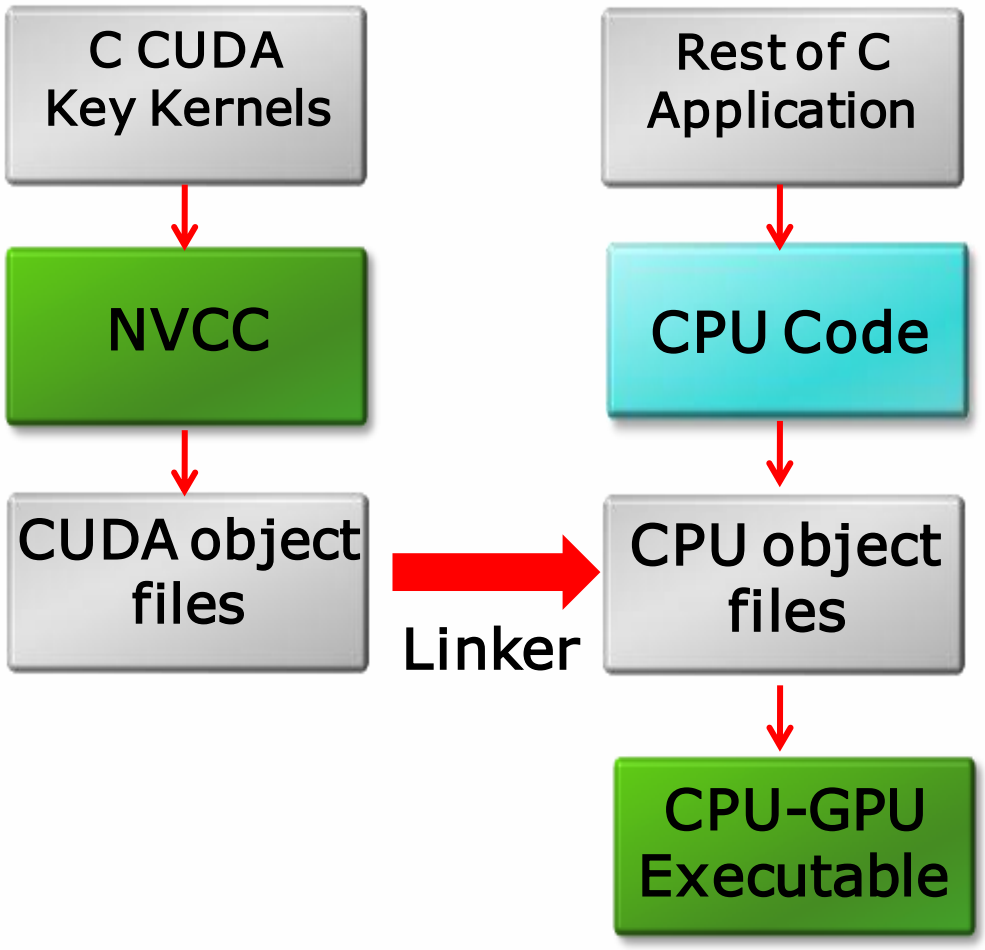
Parallel Kernel
KernelB<<< nBlk, nTid >>>(args);



CUDA Advantages over Legacy GPGPU

- Random access byte-addressable memory
  - Thread can access any memory location
- Unlimited access to memory
  - Thread can read/write as many locations as needed
- Shared memory (per block) and thread synchronization
  - Threads can cooperatively load data into shared memory
  - Any thread can then access any shared memory location
- Low learning curve
  - Just a few extensions to C
  - No knowledge of graphics is required

Compiling C for CUDA Applications



Compiling CUDA

- 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++, cl,...
- NVCC outputs
  - C code (host CPU Code)
    - Must then be compiled with the rest of the application using another tool
  - PTX
    - Object code directly
    - Or, PTX source, interpreted at runtime

Linking

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

Debugging Using the Device Emulation Mode

- An executable compiled in device emulation mode (**`nvcc -deviceemu`**) runs completely on the host using the CUDA runtime
  - No need of any device and CUDA driver
  - Each device thread is emulated with a host thread

- Running in device emulation mode, one can:
  - Use host native debug support (breakpoints, inspection, etc.)
  - Access any device-specific data from host code and vice-versa
  - Call any host function from device code (e.g. `printf`) and vice-versa
  - Detect deadlock situations caused by improper usage of `__syncthreads`

Device Emulation Mode Pitfalls

- Emulated device threads execute sequentially, so simultaneous accesses of the same memory location by multiple threads could produce different results.
- Dereferencing device pointers on the host or host pointers on the device can produce correct results in device emulation mode, but will generate an error in device execution mode