Master Informatics Eng.

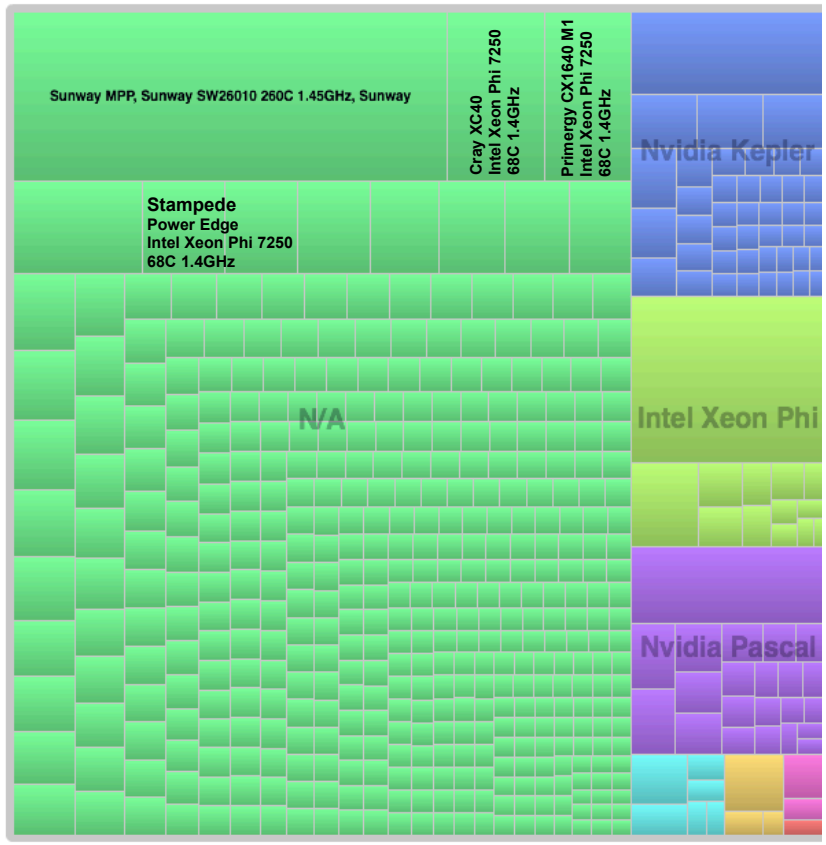
2017/18

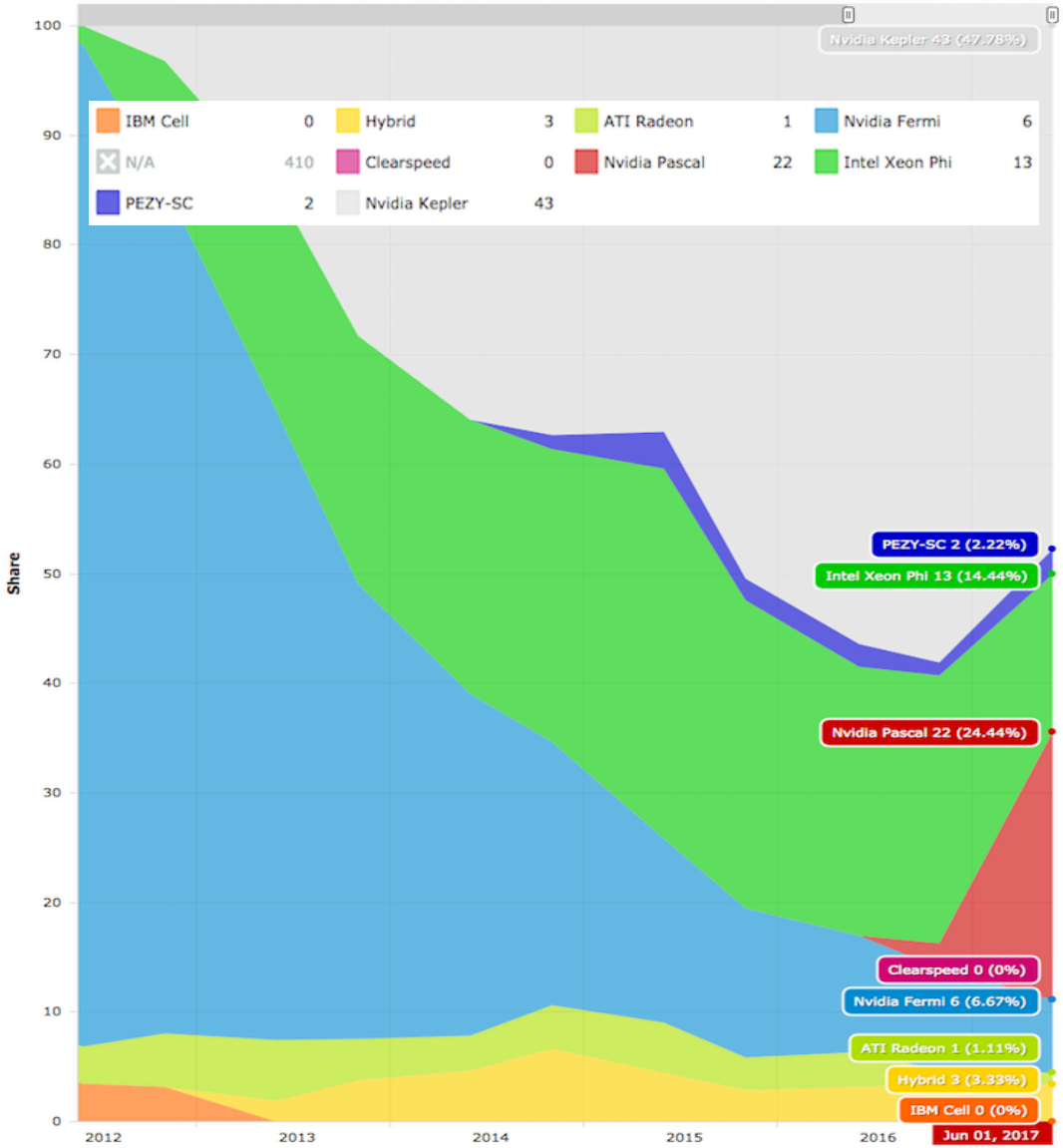
A.J.Proença

Introduction to CUDA programming

(most slides are borrowed)

Accelerator family distribution over all systems

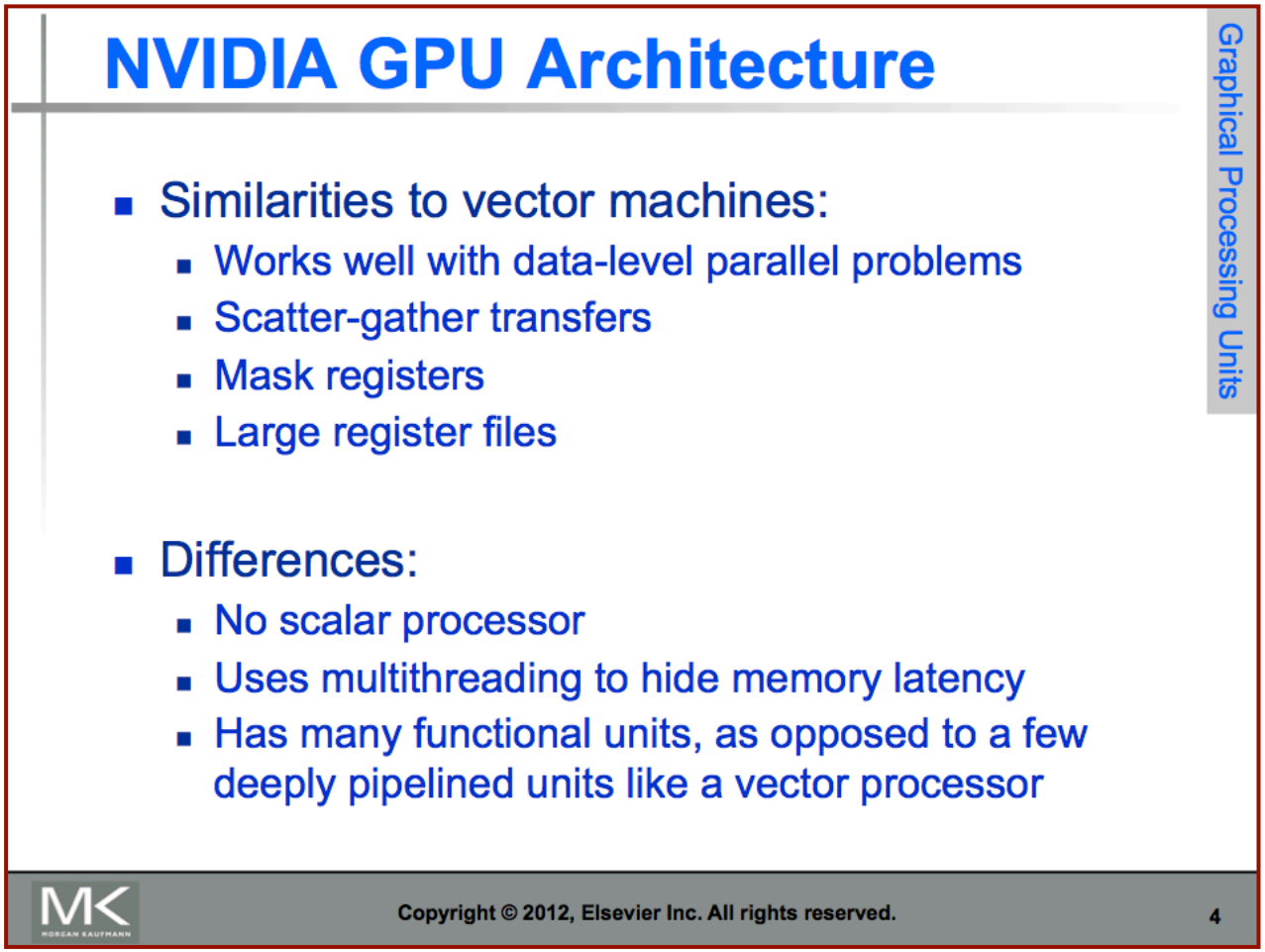




Why this interest for GPUs?

As seen before → and stressed in previous slide, but...

To be successful, GPUs require a user-friendly (for sw engineers) development environment: **CUDA**



The CUDA programming model

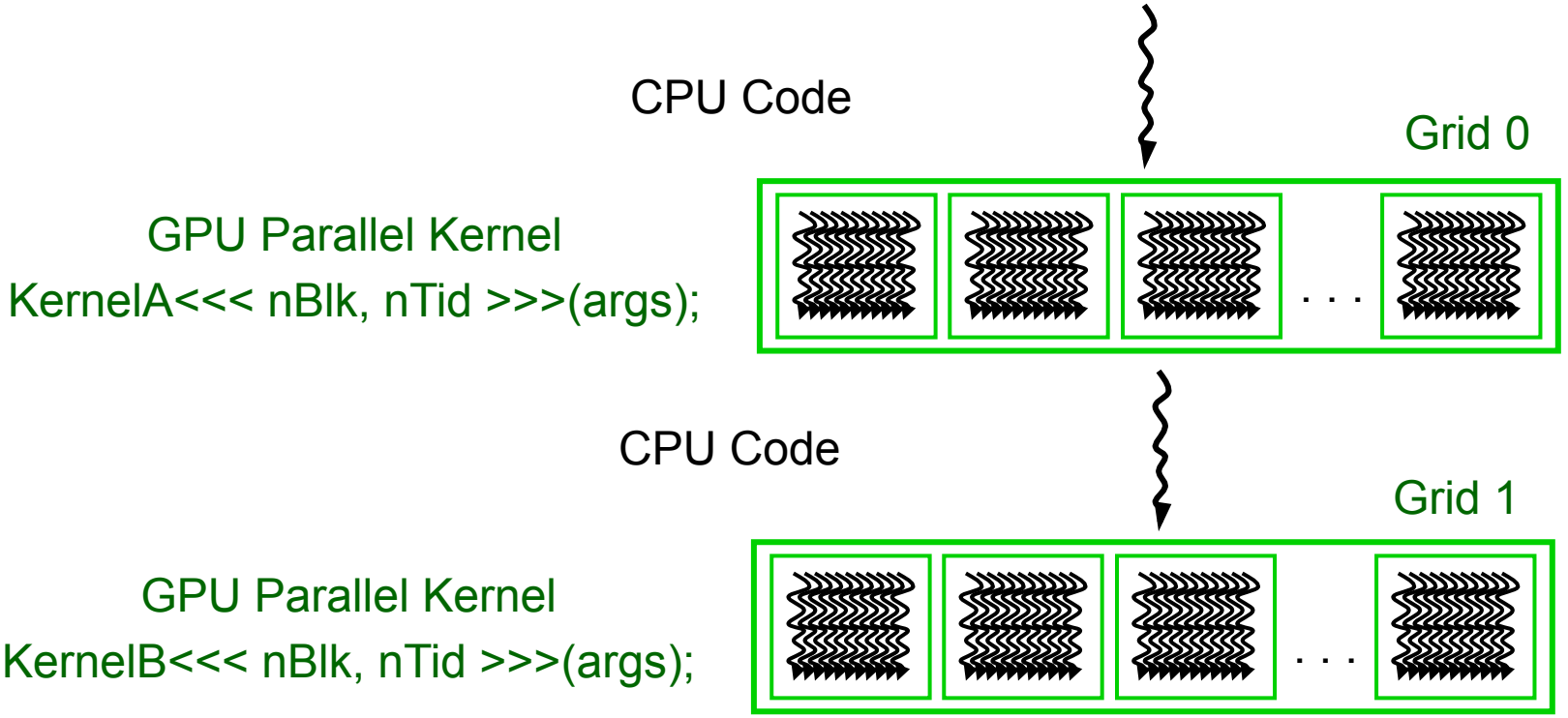
- ***C**ompute **U**nified **D**evice **A**rchitecture*
- CUDA is a programming model designed for
  - a multicore CPU ***host*** coupled to a many-core ***device***, where
  - *devices* have wide SIMD/SIMT parallelism, and
  - the *host* and the *device* do not share memory
- CUDA provides:
  - a thread abstraction to deal with SIMD
  - synchroniz. & data sharing between small groups of threads
- CUDA programs are **written in C** with extensions
- OpenCL inspired by CUDA, but hw & sw vendor neutral
  - programming model essentially identical

CUDA Devices and Threads

- A compute device
  - is a coprocessor to the CPU (the host)
  - has its own DRAM (device memory)
  - runs many threads in parallel
  - is typically a GPU but can also be another type of parallel processing device
- Data-parallel portions of an application are expressed as device kernels which run on many threads - **SIMT**
- Differences between GPU and CPU threads
  - GPU threads are extremely lightweight
    - very little creation overhead, requires LARGE register bank
  - GPU needs 1000s of threads for full efficiency
    - multi-core CPU needs only a few

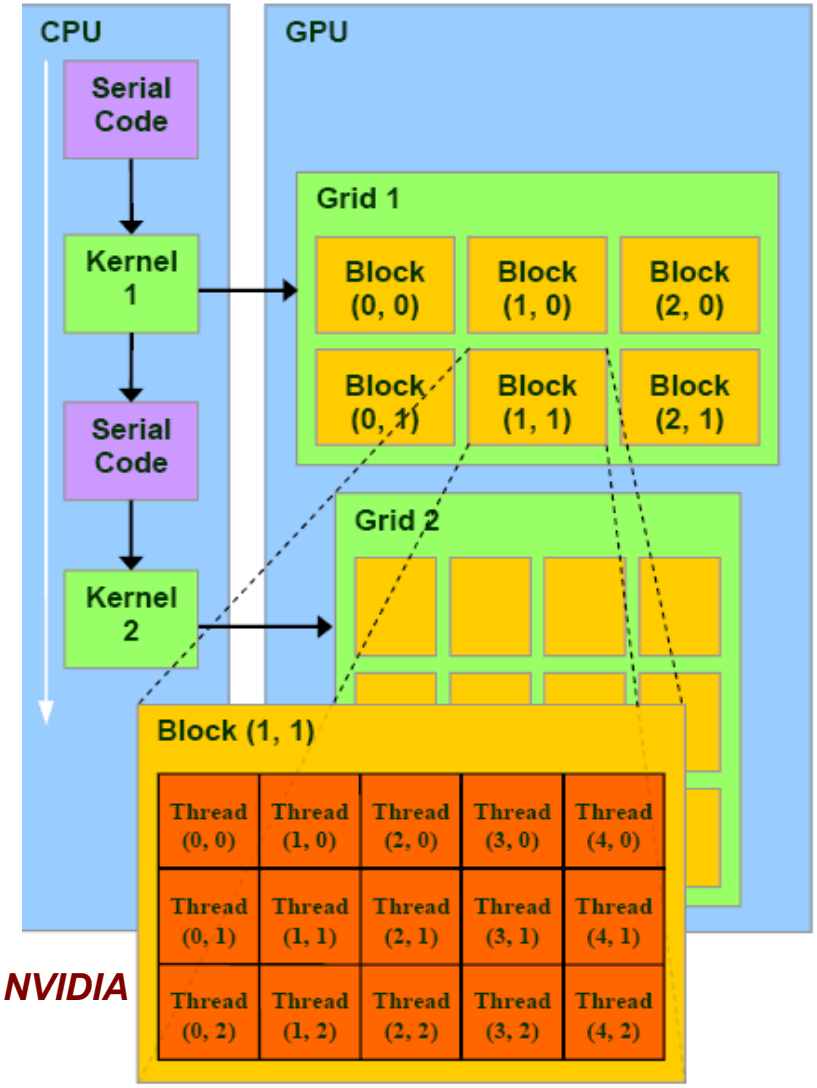
CUDA basic model: Single-Program Multiple-Data (SPMD)

- CUDA integrated CPU + GPU application C program
  - Serial C code executes on CPU
  - Parallel Kernel C code executes on GPU thread blocks



Programming Model: SPMD + SIMT/SIMD

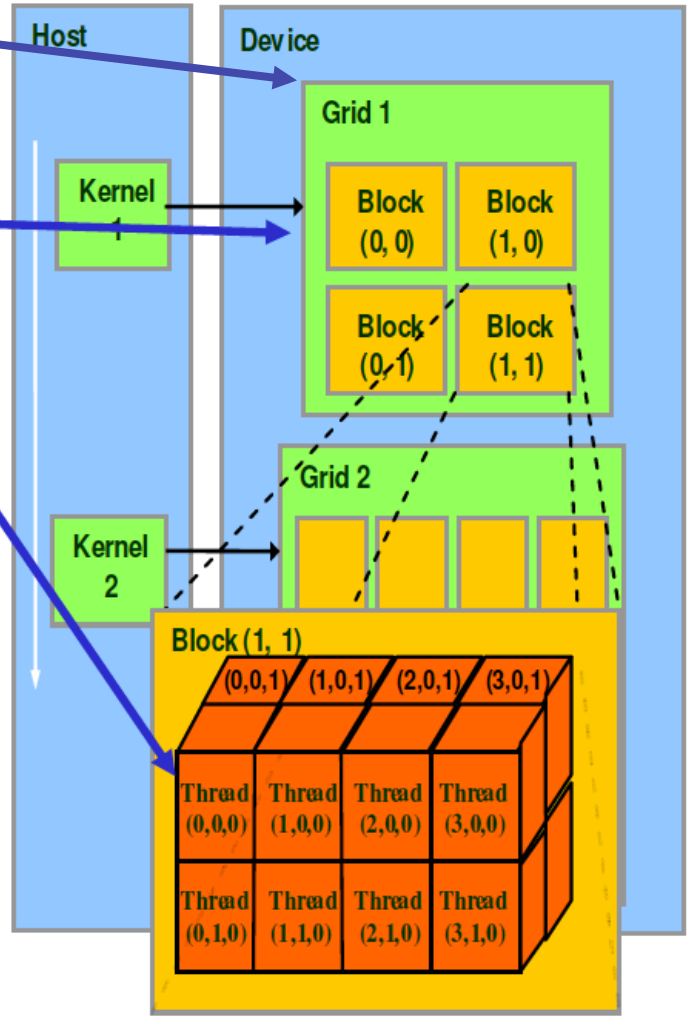
- Hierarchy
  - Device => Grids
  - Grid => Blocks
  - Block => Warps
  - Warp => Threads
- Single kernel runs on multiple blocks (SPMD)
- Threads within a warp are executed in a lock-step way called single-instruction multiple-thread (SIMT)
- Single instructions are executed on multiple threads (SIMD)
  - Warp size defines SIMD granularity (32 threads)
- Synchronization within a block uses shared memory



Courtesy NVIDIA

The Computational Grid: Block IDs and Thread IDs

- A kernel runs on a computational grid of thread blocks
  - Threads share global memory
- Each thread uses IDs to decide what data to work on
  - Block ID: 1D or 2D
  - Thread ID: 1D, 2D, or 3D
- A thread block is a batch of threads that can cooperate by:
  - Sync their execution w/ barrier
  - Efficiently sharing data through a low latency shared memory
  - Two threads from two different blocks cannot cooperate



Example

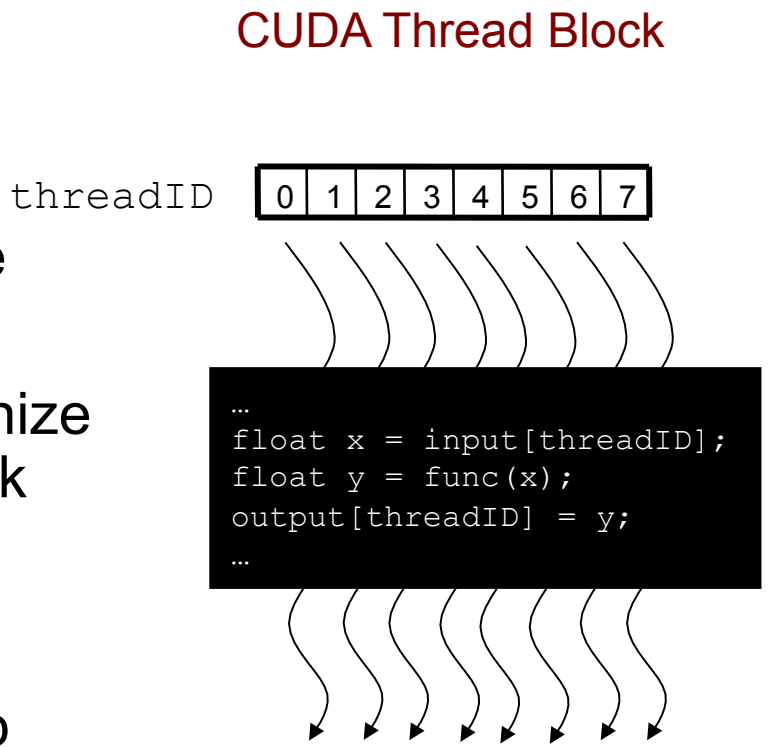
- Multiply two vectors of length 8192
  - Code that works over all elements is the grid
  - Thread blocks break this down into manageable sizes
    - 512 threads per block
  - SIMD instruction executes 32 elements at a time
  - Thus grid size = 16 blocks
  - Block is analogous to a strip-mined vector loop with vector length of 32
  - Block is assigned to a *multithreaded SIMD processor* by the *thread block scheduler*
  - Current-generation GPUs (Fermi) have 7-16 multithreaded SIMD processors

Terminology *(and in NVidia)*

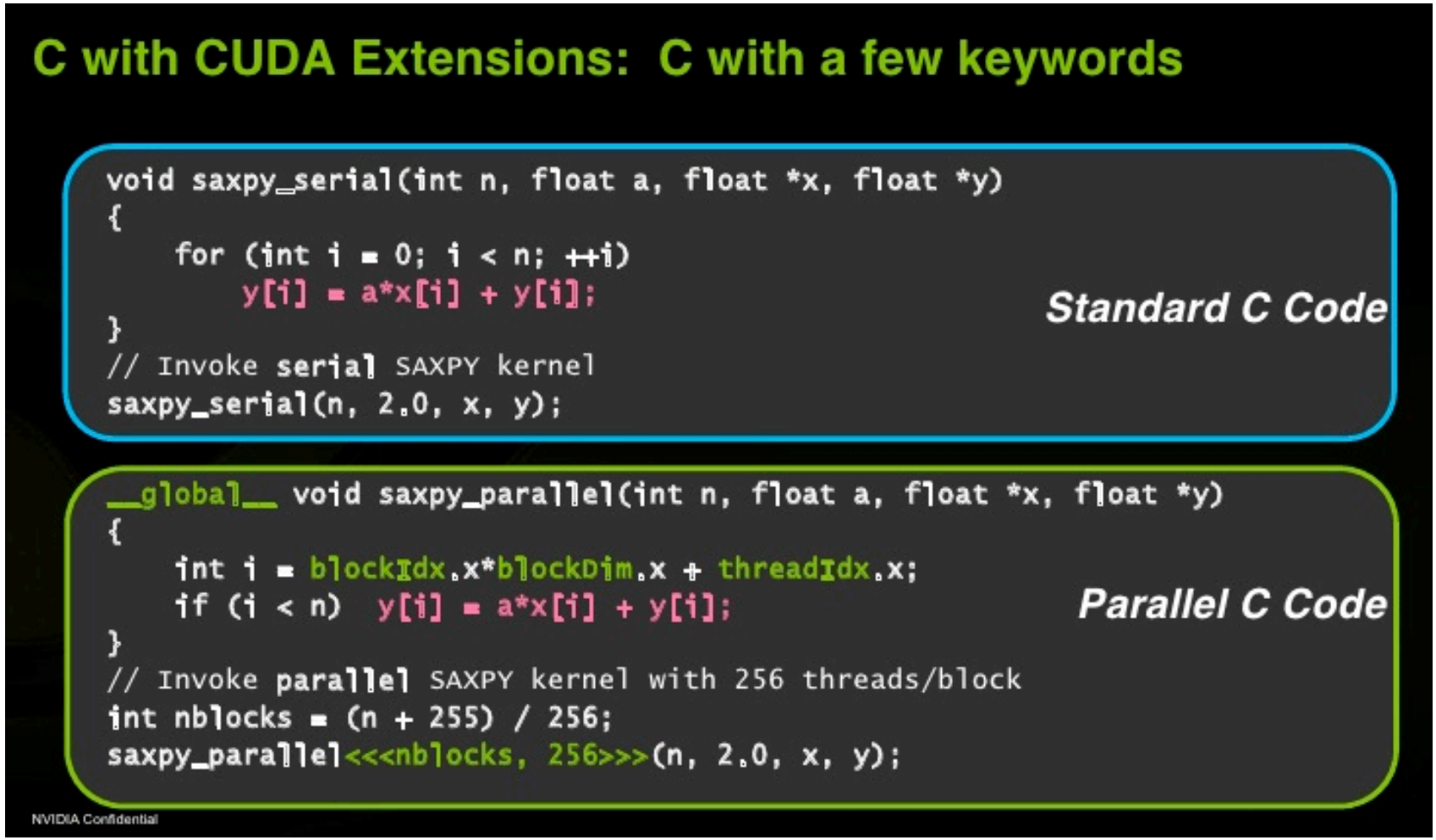
- *Threads of SIMD instructions (**warps**)*
  - Each has its own IP (up to 48/64 per SIMD processor, Fermi/Kepler)
  - Thread scheduler uses scoreboard to dispatch
  - No data dependencies between threads!
  - Threads are organized into blocks & executed in groups of 32 threads *(**thread block**)*
    - Blocks are organized into a grid
- The thread block scheduler schedules blocks to SIMD processors *(**Streaming Multiprocessors**)*
- Within each SIMD processor:
  - 32 SIMD lanes *(**thread processors**)*
  - Wide and shallow compared to vector processors

CUDA Thread Block

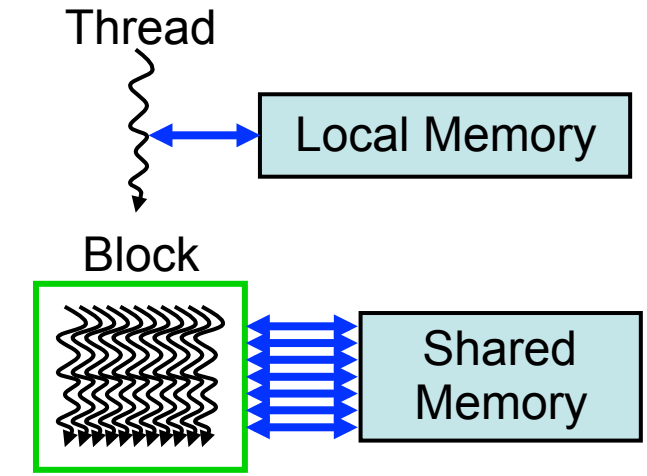
- Programmer declares (Thread) Block:
  - Block size 1 to **512** concurrent threads
  - Block shape 1D, 2D, or 3D
  - Block dimensions in threads
- All threads in a Block execute the same thread program
- Threads share data and synchronize while doing their share of the work
- Threads have thread id numbers within Block
- Thread program uses thread id to select work and address shared data



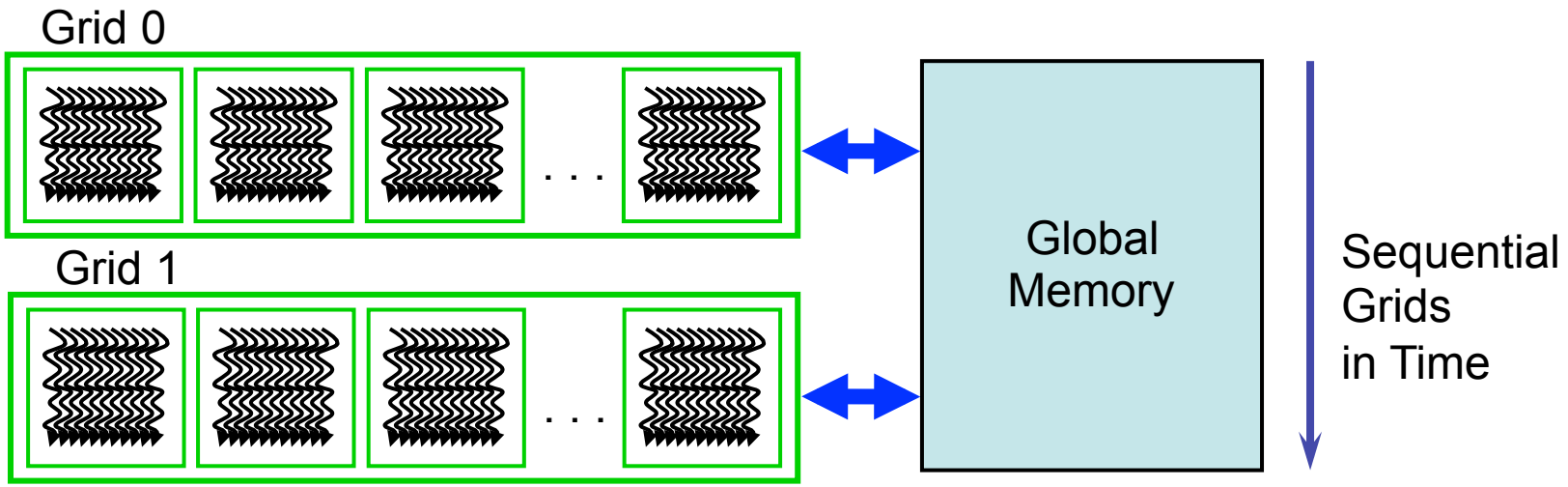
Example of Host & Device code



Parallel Memory sharing

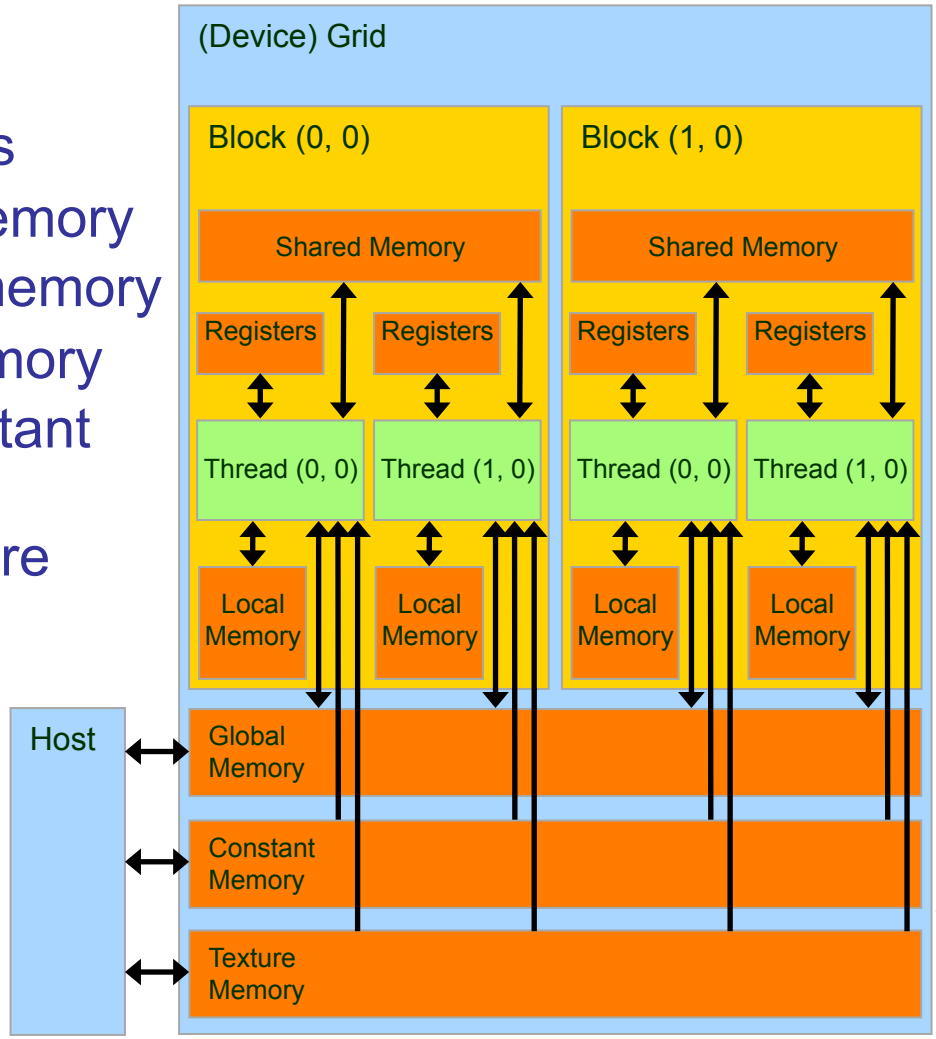


- Local Memory: **per-thread**
  - Private per thread
  - Auto variables, register spill
- Shared Memory: **per-block**
  - Shared by threads of the same block
  - Inter-thread communication
- Global Memory: **per-application**
  - Shared by all threads
  - Inter-Grid communication

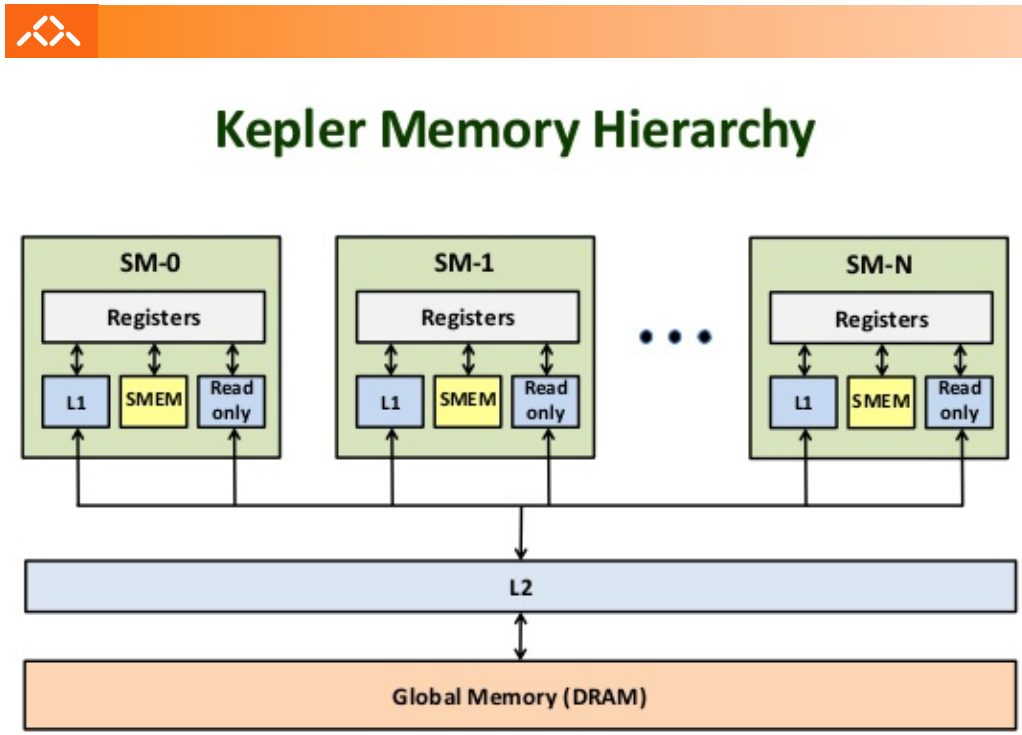


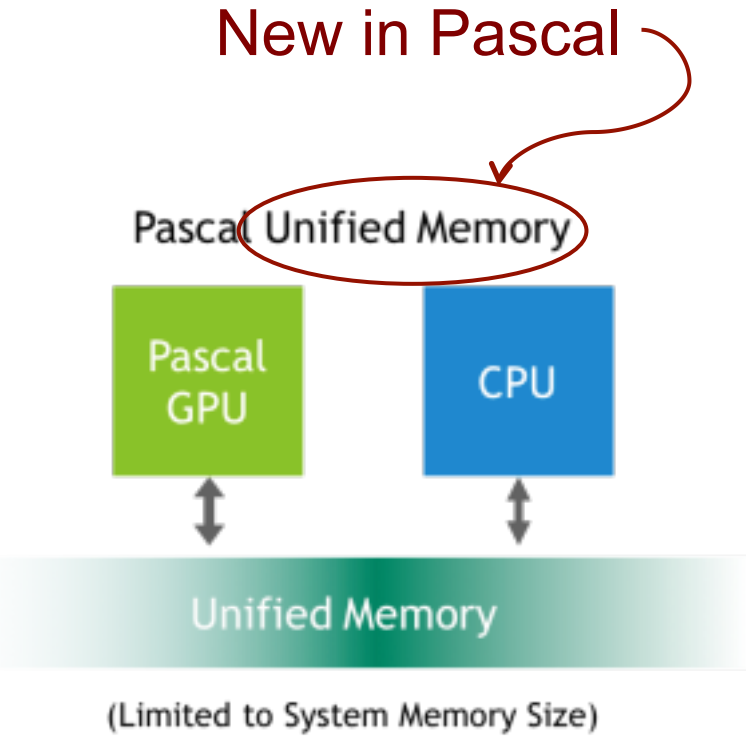
CUDA Memory Model overview

- 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



Memory Hierarchies in NVidia GPUs

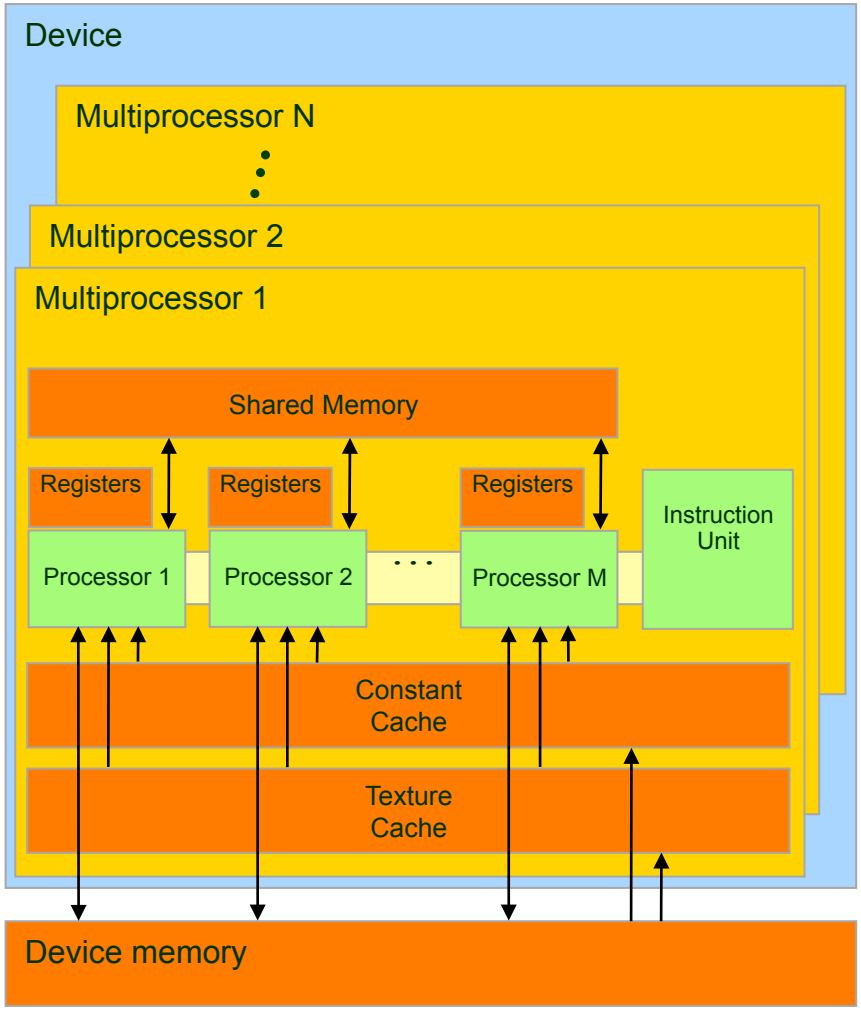




CUDA 8 launched
for Pascal architecture

Hardware Implementation: Memory Architecture

- Device memory (DRAM)
  - Slow (2~300 cycles)
  - Local, global, constant, and texture memory
- On-chip memory
  - Fast (1 cycle)
  - Registers, shared memory, constant/texture cache

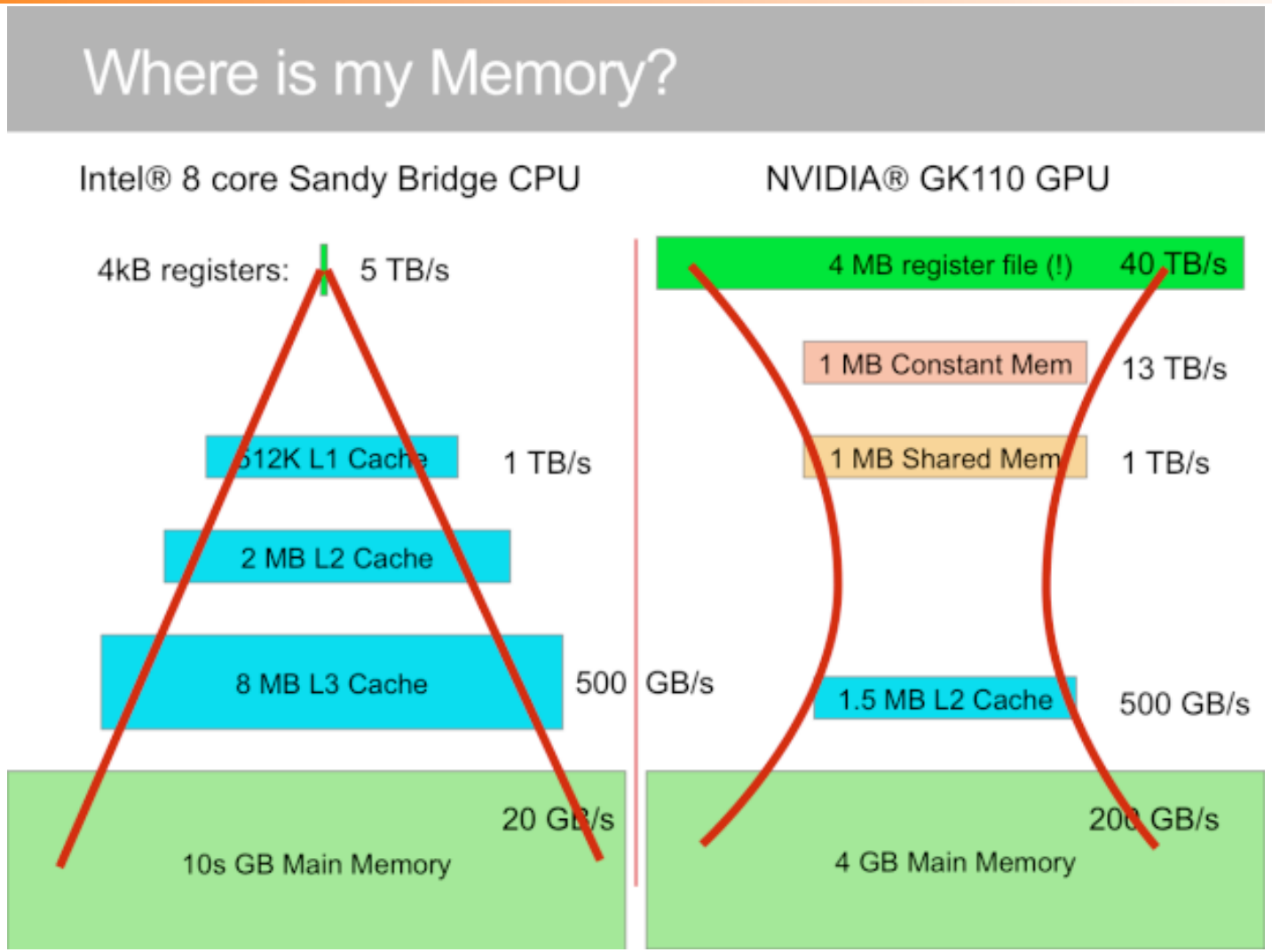


Courtesy NVIDIA

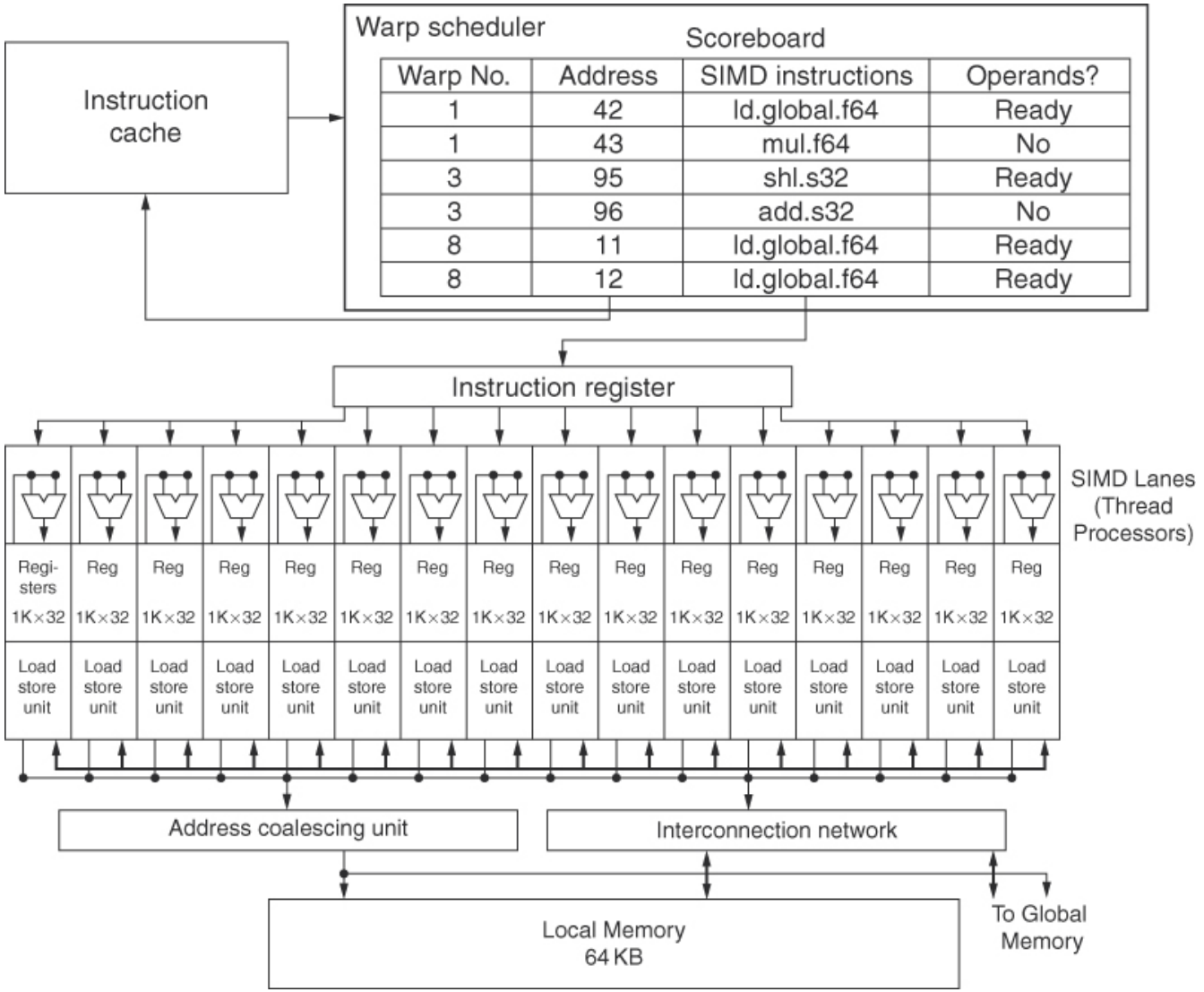
NVIDIA GPU Memory Structures

- Each SIMD Lane has private section of **off-chip DRAM**
  - "Private memory" *(Local Memory)*
  - Contains stack frame, spilling registers, and private variables
- Each multithreaded SIMD processor also has local memory *(Shared Memory)*
  - Shared by SIMD lanes / threads within a block
- Memory shared by SIMD processors is GPU memory *(Global Memory)*
  - Host can read and write GPU memory

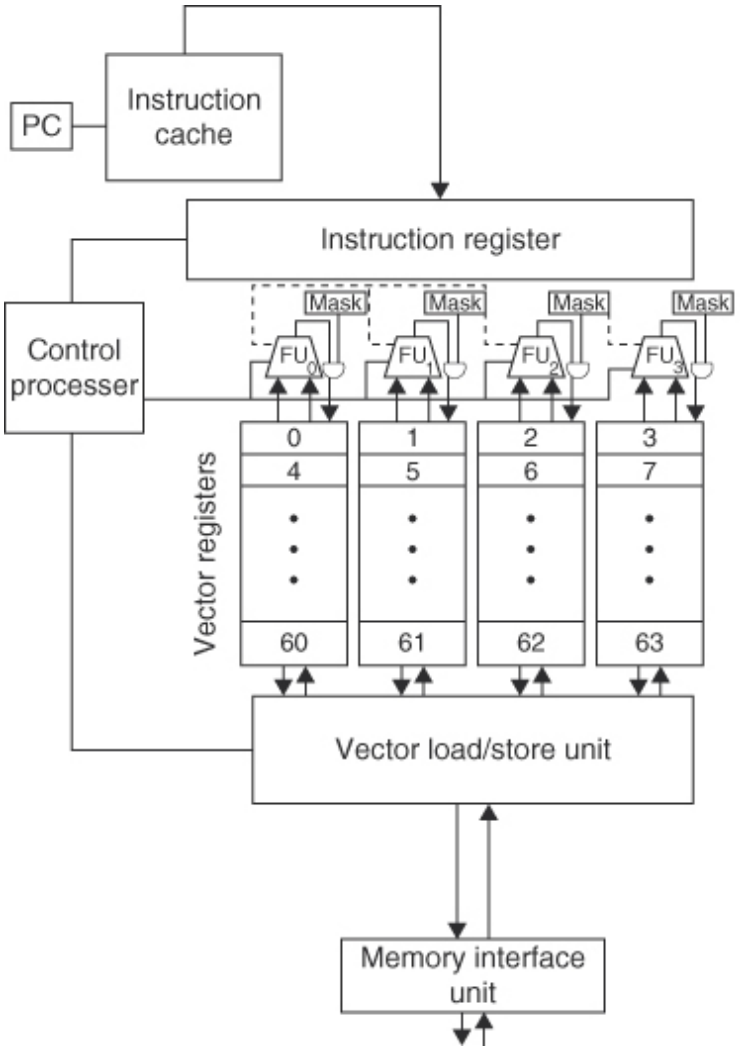
Memory Hierarchy: CPU vs. GPU

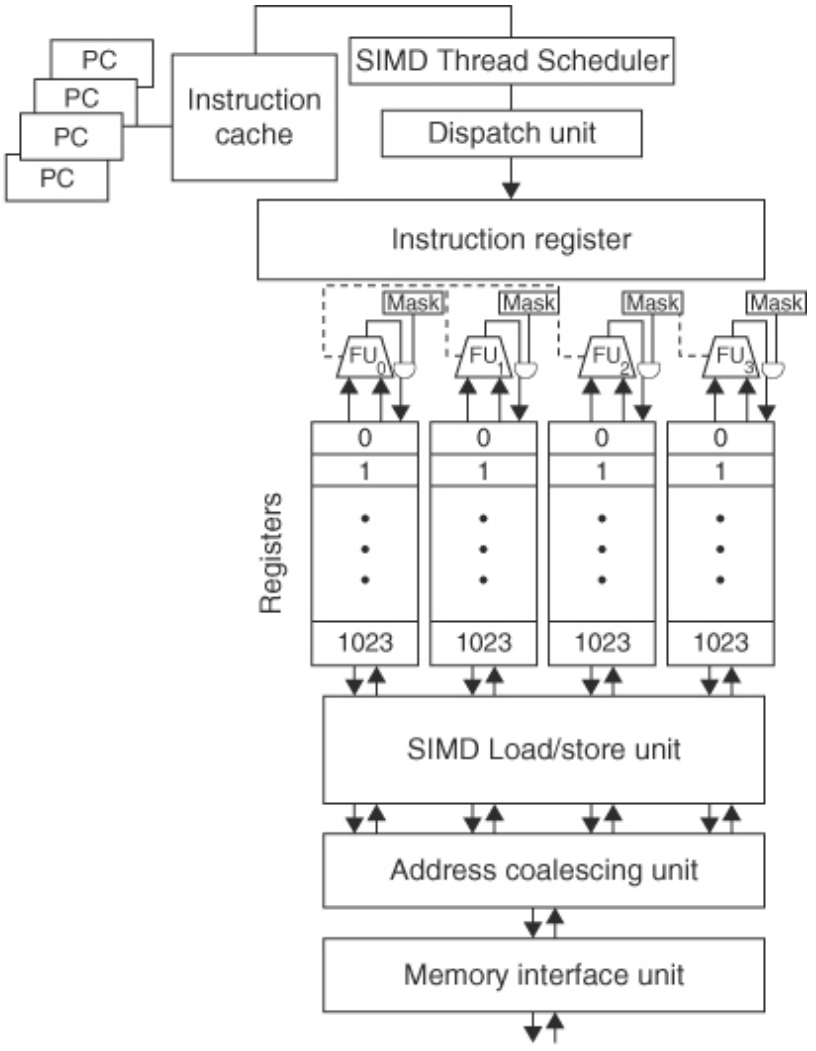


Example



Vector Processor *versus* CUDA core





Conditional Branching

- Like vector architectures, GPU branch hardware uses internal masks
- Also uses
  - Branch synchronization stack
    - Entries consist of masks for each SIMD lane
    - I.e. which threads commit their results (all threads execute)
  - Instruction markers to manage when a branch diverges into multiple execution paths
    - Push on divergent branch
  - …and when paths converge
    - Act as barriers
    - Pops stack
- Per-thread-lane 1-bit predicate register, specified by programmer

Example of conditional code execution

Up to Pascal
Volta supports finer thread scheduling

```
if (threadIdx.x < 4) {
    A;
    B;
} else {
    X;
    Y;
}
Z;
```