Master Informatics Eng.

2020/21

A.J.Proença

Programming GPUs with CUDA

(most slides are borrowed)

The CUDA programming model

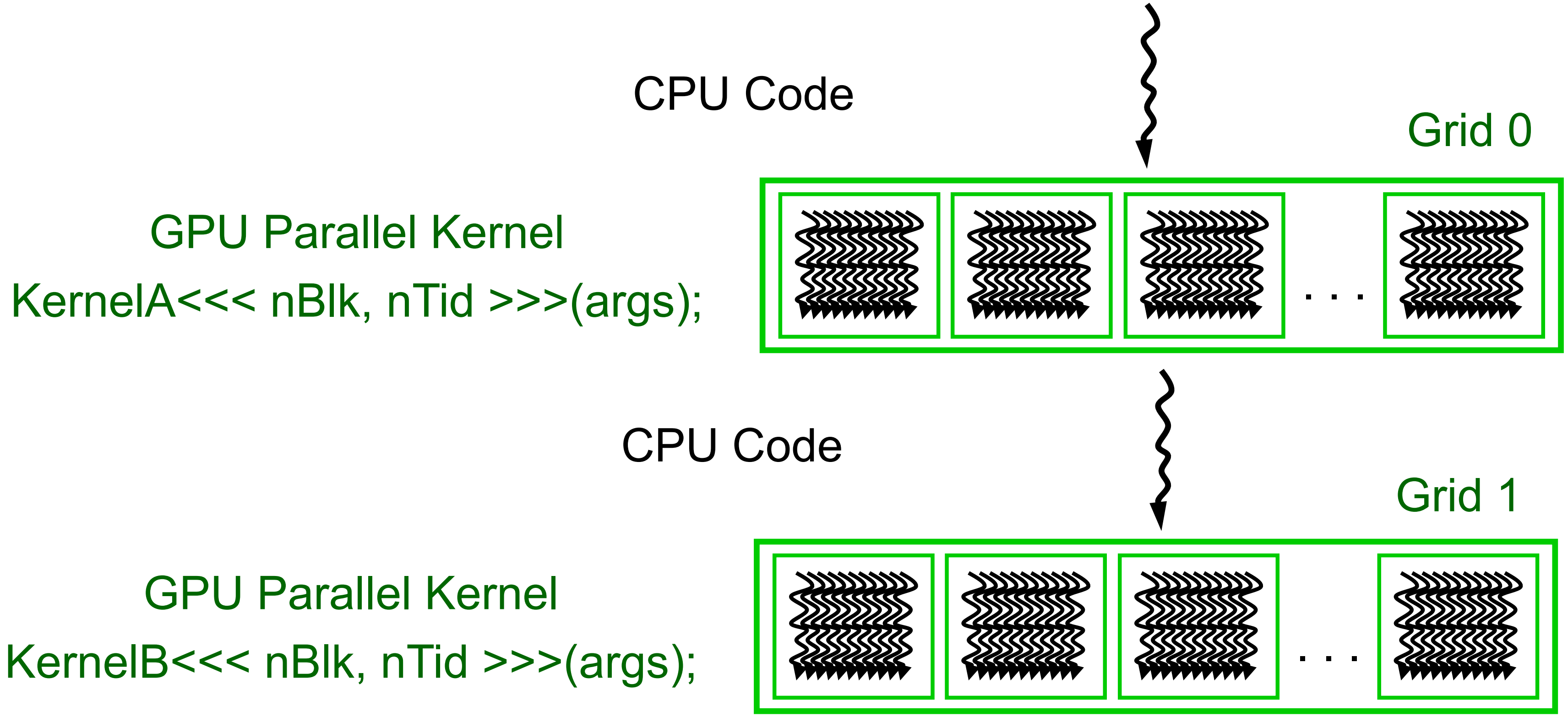
- ***C**ompute **U**nified **D**evice **A**rchitecture*
- **CUDA** is a programming model, designed for
  - a multicore CPU ***host*** coupled to a many-core ***device***, where
  - *devices* have wide SIMD/SIMT parallelism, and
  - the *host* and the *device* do not share memory
- **CUDA** provides:
  - a thread abstraction to deal with SIMD
  - synchronization & 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 or **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

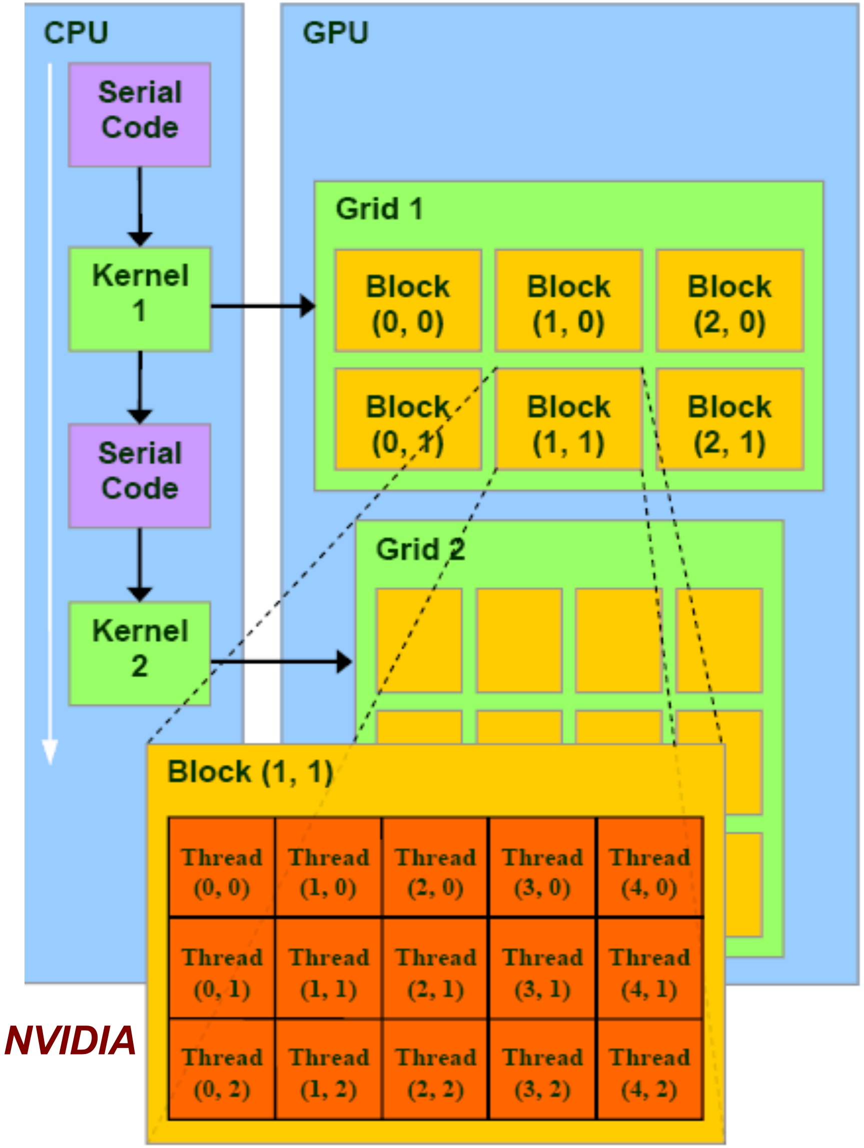
CUDA basic model: Single-Program Multiple-Data (SPMD)

- CUDA integrates **CPU+GPU** code in a **C** program
  - Serial C code executes on CPU
  - Parallel **Kernel** C code executes on GPU **thread blocks**



Programming Model: SPMD + SIMT/SIMD

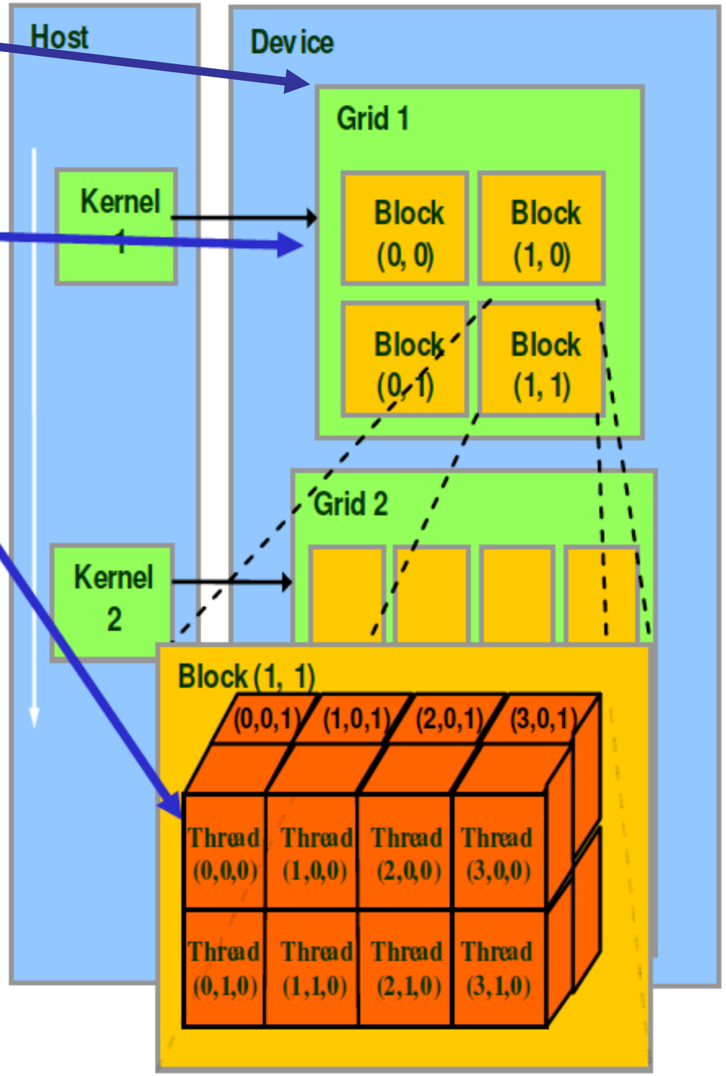
- Hierarchy
  - Device => Grids
  - Grid => Blocks
  - Block => Warps
  - Warp => Threads
- Single kernel runs on multiple blocks (SPMD)
- Threads within a warp are executed in a lock-step way called single-instruction multiple-thread (SIMT)
- Single instruction are executed on multiple threads (SIMD)
  - 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*

    SM in NVidia terminology…

    … or simply a CPU-type core
  - Current-generation NVidia GPU (GA100 Ampere) has 128 *multithreaded SIMD processors*

Strip-mining and Cleanup

Strip-mining, also known as loop sectioning, is a loop transformation technique for enabling SIMD-encodings of loops, as well as a means of improving memory performance. By fragmenting a large loop into smaller segments or strips, this technique transforms the loop structure in two ways:

- It increases the temporal and spatial locality in the data cache if the data are reusable in different passes of an algorithm.
- It reduces the number of iterations of the loop by a factor of the length of each vector, or number of operations being performed per SIMD operation. In the case of Streaming SIMD Extensions, this vector or strip-length is reduced by 4 times: four floating-point data items per single Streaming SIMD Extensions single-precision floating-point SIMD operation are processed.

First introduced for vectorizers, this technique consists of the generation of code when each vector operation is done for a size less than or equal to the maximum vector length on a given vector machine.

The compiler automatically strip-mines your loop and generates a cleanup loop. The following examples demonstrate strip mining and cleaning up loops.

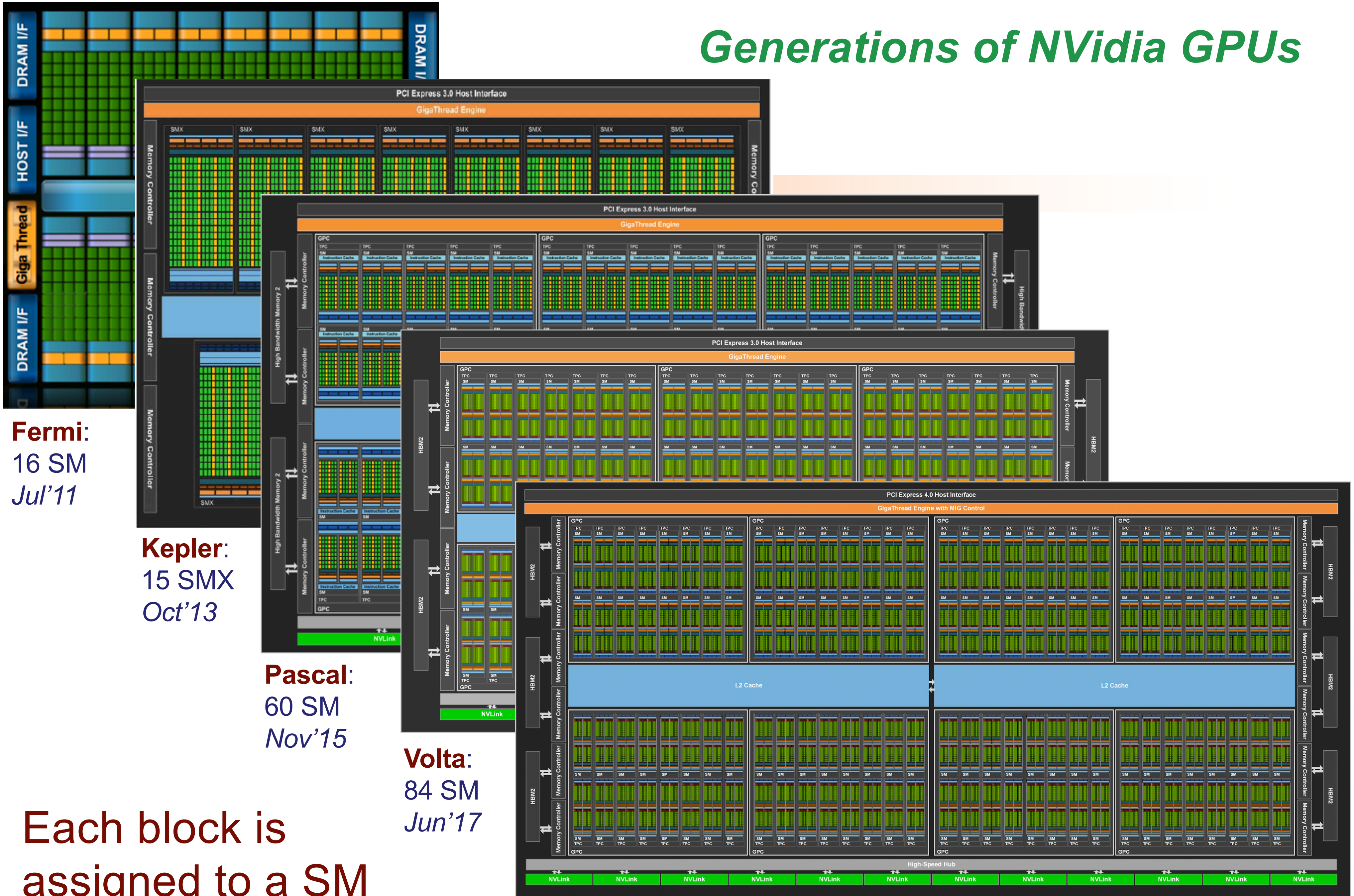
Example1: Before Vectorization

```
i = 1
do while (i<=n)
a(i) = b(i) + c(i) ! Original loop code
i = i + 1
end do
```

Example 2: After Vectorization

```
!The vectorizer generates the following two loops
i = 1
do while (i < (n - mod(n,4)))
! Vector strip-mined loop.
a(i:i+3) = b(i:i+3) + c(i:i+3)
i = i + 4
end do
do while (i <= n)
a(i) = b(i) + c(i)     !Scalar clean-up loop
i = i + 1
end do
```

Generations of NVidia GPUs



Fermi:
16 SM
Jul'11

Kepler:
15 SMX
Oct'13

Pascal:
60 SM
Nov'15

Volta:
84 SM
Jun'17

Ampere:
128 SM
May'20

Each block is assigned to a SM

C with CUDA extensions

C with CUDA Extensions: C with a few keywords

Standard C Code

```
void saxpy_serial(int n, float a, float *x, float *y)
{
    for (int i = 0; i < n; ++i)
        y[i] = a*x[i] + y[i];
}
// Invoke serial SAXPY kernel
saxpy_serial(n, 2.0, x, y);
```

Parallel C Code

```
__global__ void saxpy_parallel(int n, float a, float *x, float *y)
{
    int i = blockIdx.x*blockDim.x + threadIdx.x;
    if (i < n)  y[i] = a*x[i] + y[i];
}
// Invoke parallel SAXPY kernel with 256 threads/block
int nblocks = (n + 255) / 256;
saxpy_parallel<<<nblocks, 256>>>(n, 2.0, x, y);
```

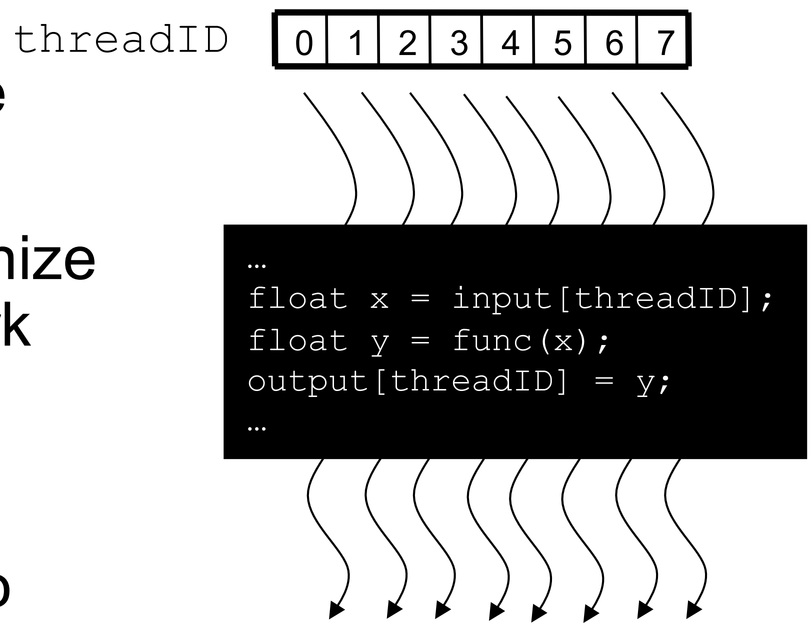
Terminology (and in NVidia)

- Threads of SIMD instructions *(**warps**)*
  - Each has its own IP (up to 48/64 per SIMD processor, Fermi/after-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, SM**)*
- Within each SIMD processor:
  - 32 SIMD lanes *(**thread processors**)*
  - Wide and shallow compared to vector processors

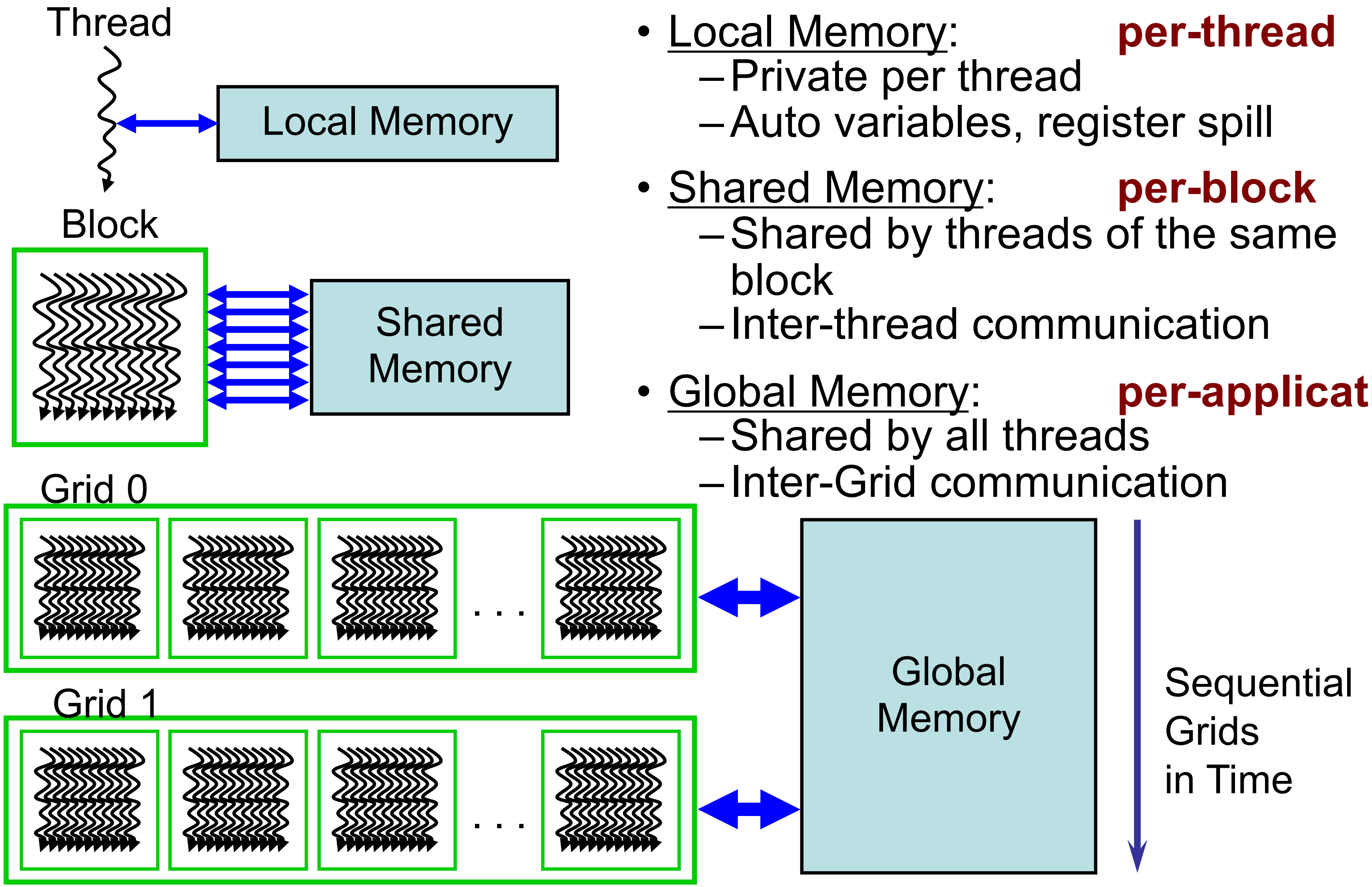
CUDA Thread Block

- 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

CUDA Thread Block



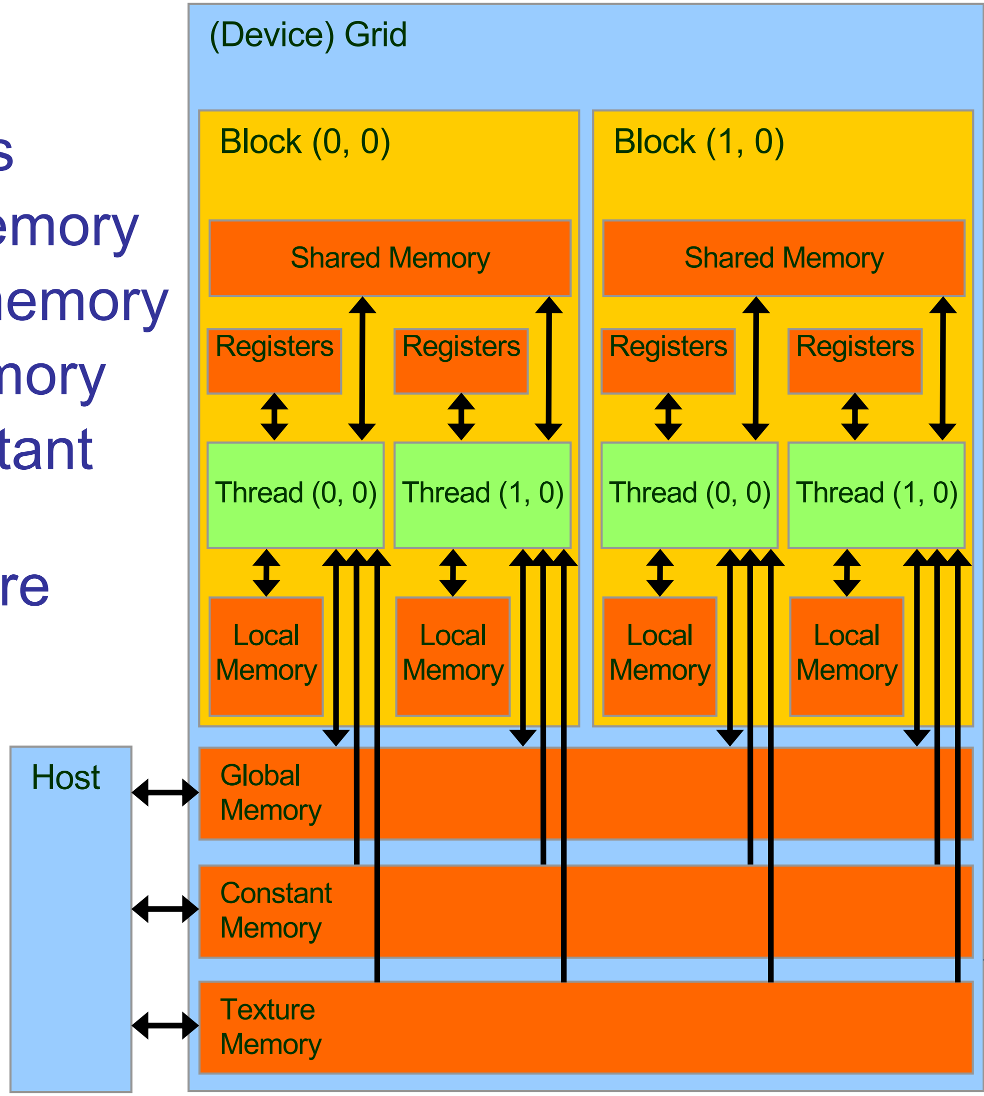
Parallel Memory Sharing



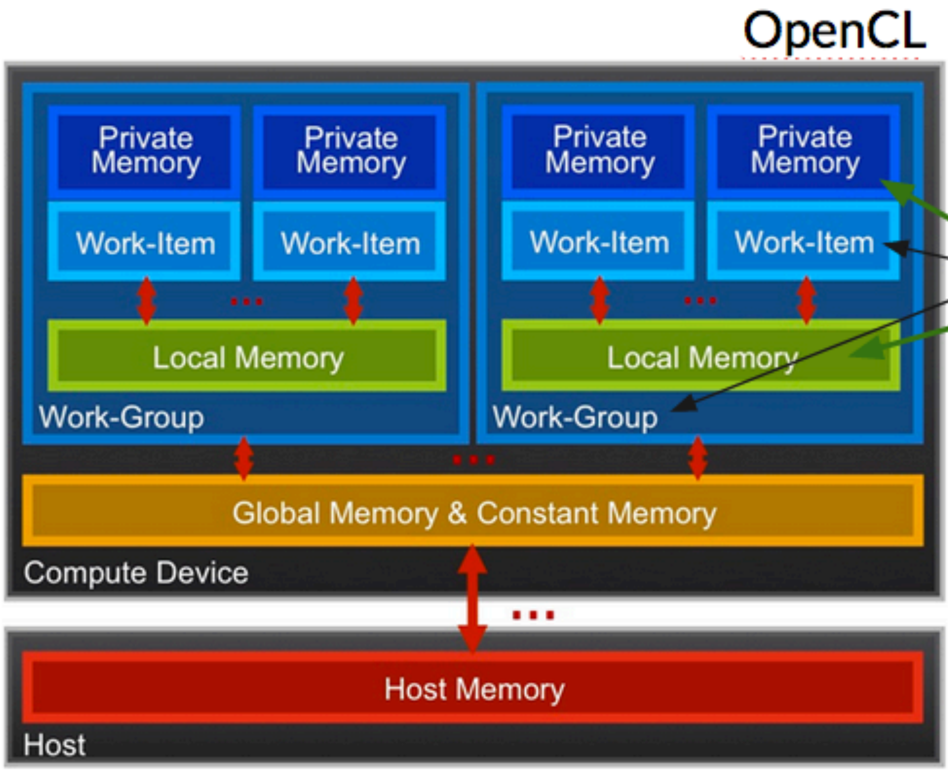
- Local Memory: **per-thread**
  - Private per thread
  - Auto variables, register spill
- Shared Memory: **per-block**
  - Shared by threads of the same block
  - Inter-thread communication
- Global Memory: **per-application**
  - Shared by all threads
  - Inter-Grid communication

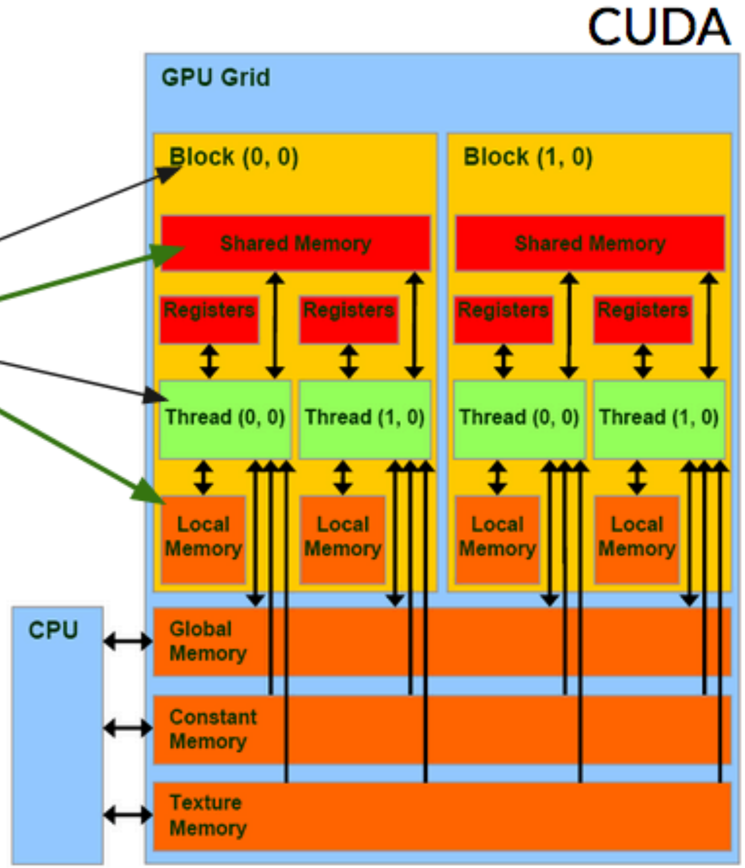
CUDA Memory Model Overview

- 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



CUDA and OpenCL

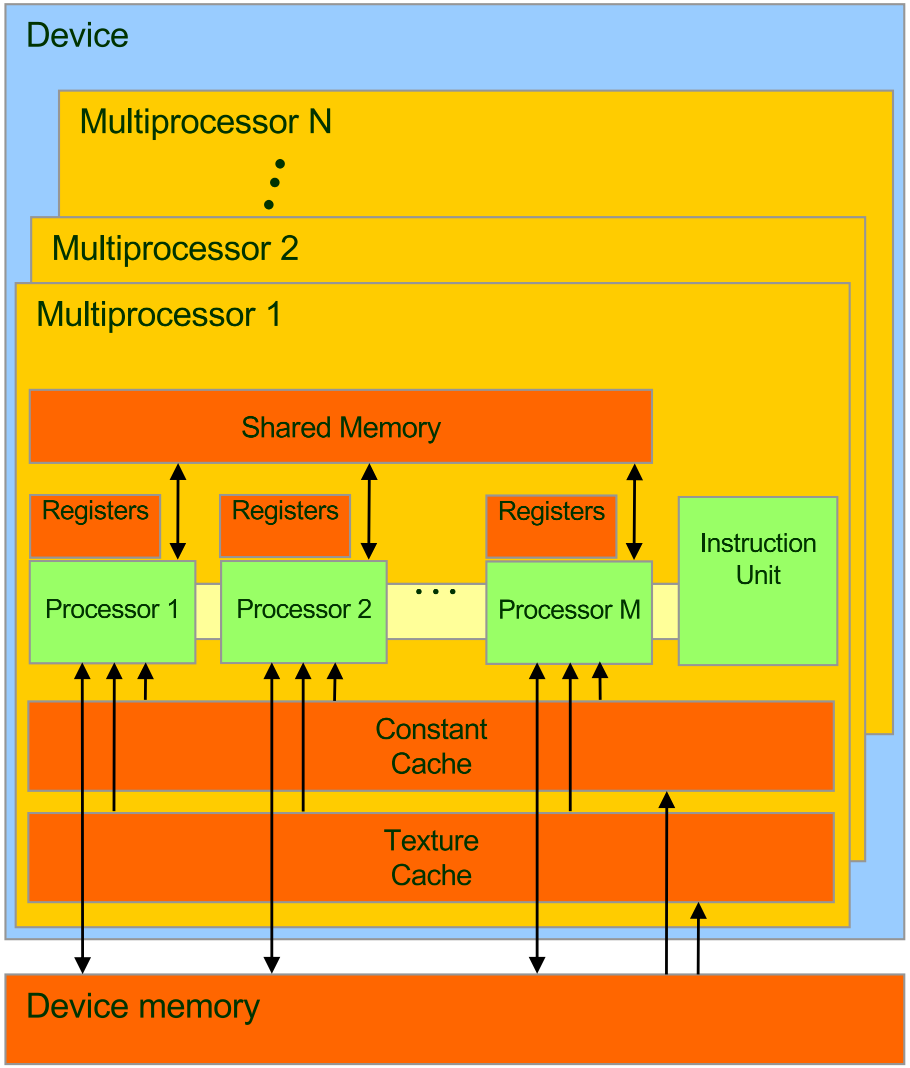




Hardware Implementation: Memory Architecture

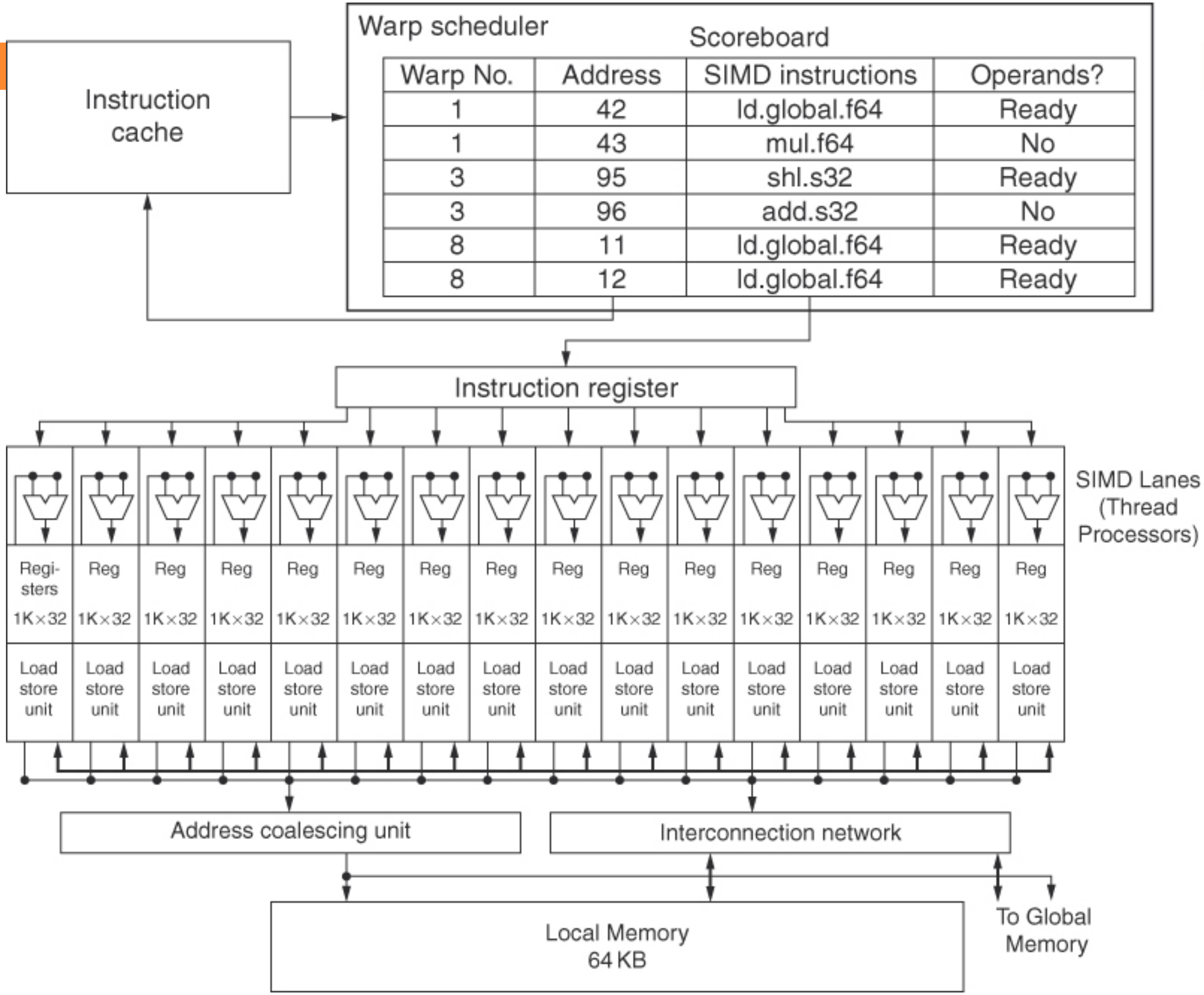
- Device memory (DRAM)
  - Slow (2~300 cycles)
  - Local, global, constant, and texture memory

- On-chip memory
  - Fast (1 cycle)
  - Registers, shared memory, constant/texture cache



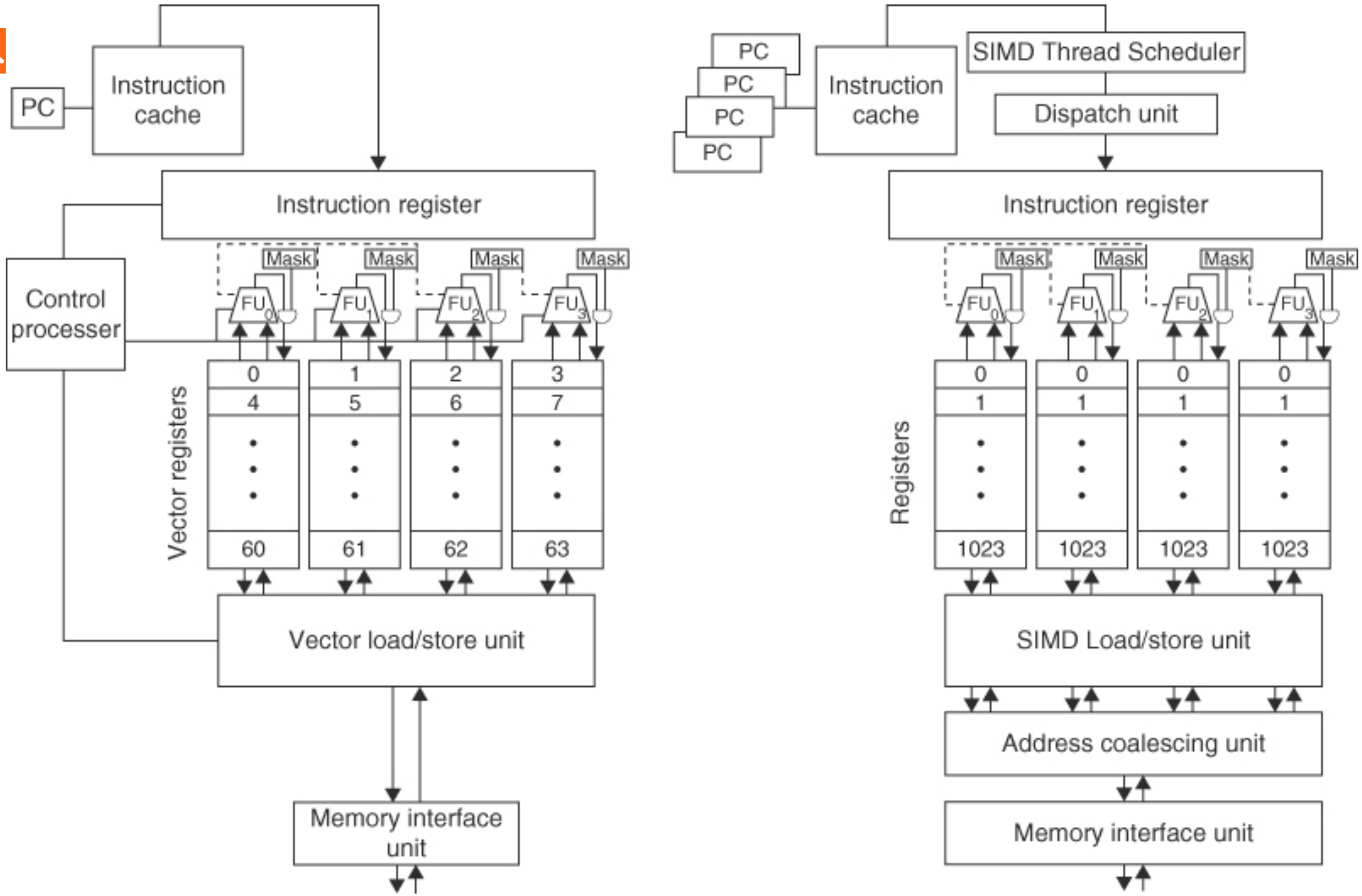
Courtesy NVIDIA

Example



Warp No.	Address	SIMD instructions	Operands?
1	42	ld.global.f64	Ready
1	43	mul.f64	No
3	95	shl.s32	Ready
3	96	add.s32	No
8	11	ld.global.f64	Ready
8	12	ld.global.f64	Ready

Vector Processor versus CUDA core



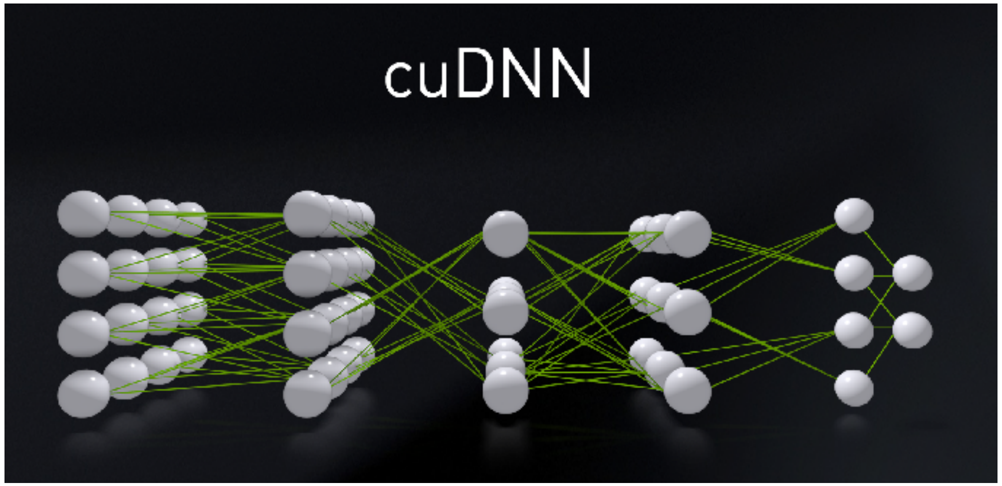
Conditional Branching

- 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

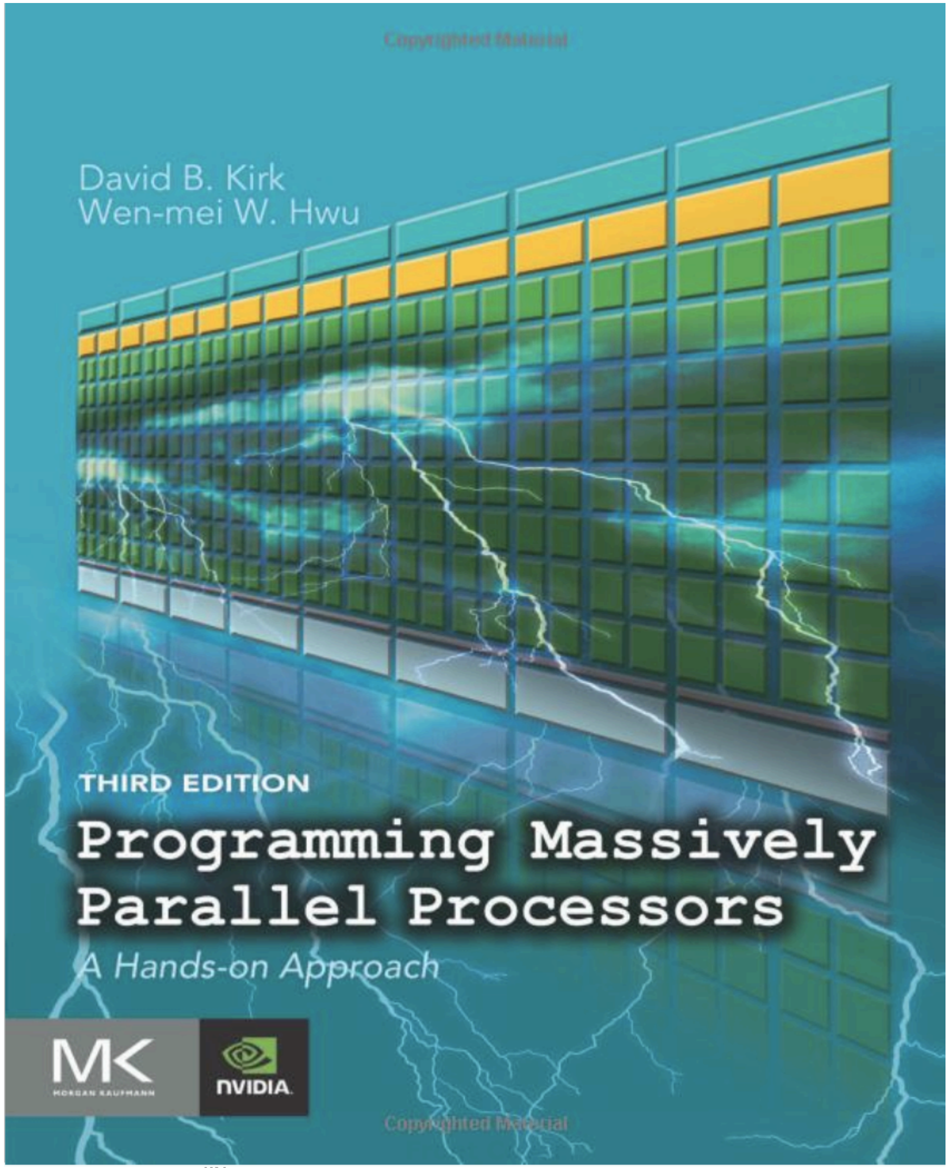
cuDNN: a GPU-accelerated library of primitives for deep neural networks

cuDNN contains:

- common training and inference routines,
- tensor utility routines,
- routines for CNN for training and inference time,
- ...



Recommended textbook (2)



Contents