



GPU Programming Concepts

Data Parallel Computing

Prof. Dr. Matthias S. Müller

Dr. Christian Terboven

Dr. Sandra Wienke

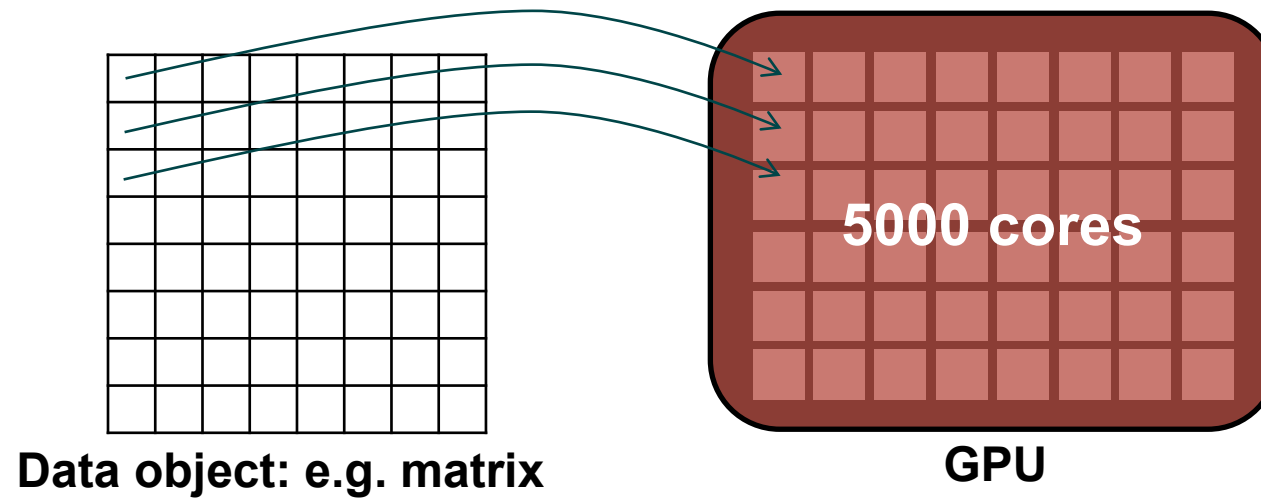
Julian Miller

What is This Chapter About?

- Important concepts of programming GPUs
 - Data-Parallel Execution Models
 - Single Instruction Multiple Threads (SIMT)
 - Warps and their scheduling
 - How to create parallelism on the GPU?
 - Offloading concept
 - Memory & data management
 - Memory consistency model
 - Mapping parallelism to the GPU hardware

Data-Parallel Computing

- "If you were plowing a field, which would you rather use: Two strong oxen or 1024 chickens?"
Seymour Cray
 - Latency vs. throughput-oriented hardware
 - Latency indicates how long it takes for packets to reach their destination. Throughput is the term given to the number of packets that are processed within a specific period of time.
- GPU design goal: maximize throughput
 - A single thread is executed on each processing element simultaneously
 - Threads are logically organized like data



Data-Parallel Execution Models

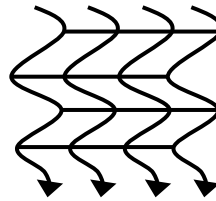
- Most common approaches to data parallelism

SIMD/ Vector



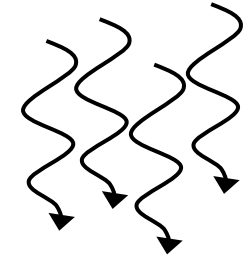
- One thread with wide execution data-path
- Example: x86 SSE/ AVX instructions

SIMT



- Multiple threads with a shared program counter (lockstep)
- Example: GPUs

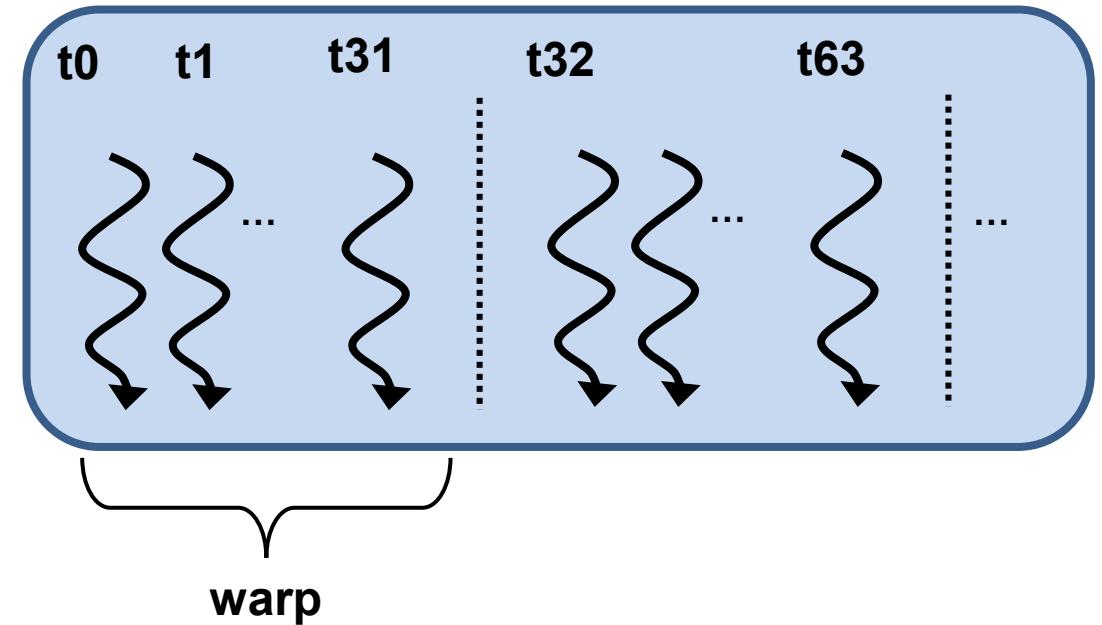
MIMD/ SPMD



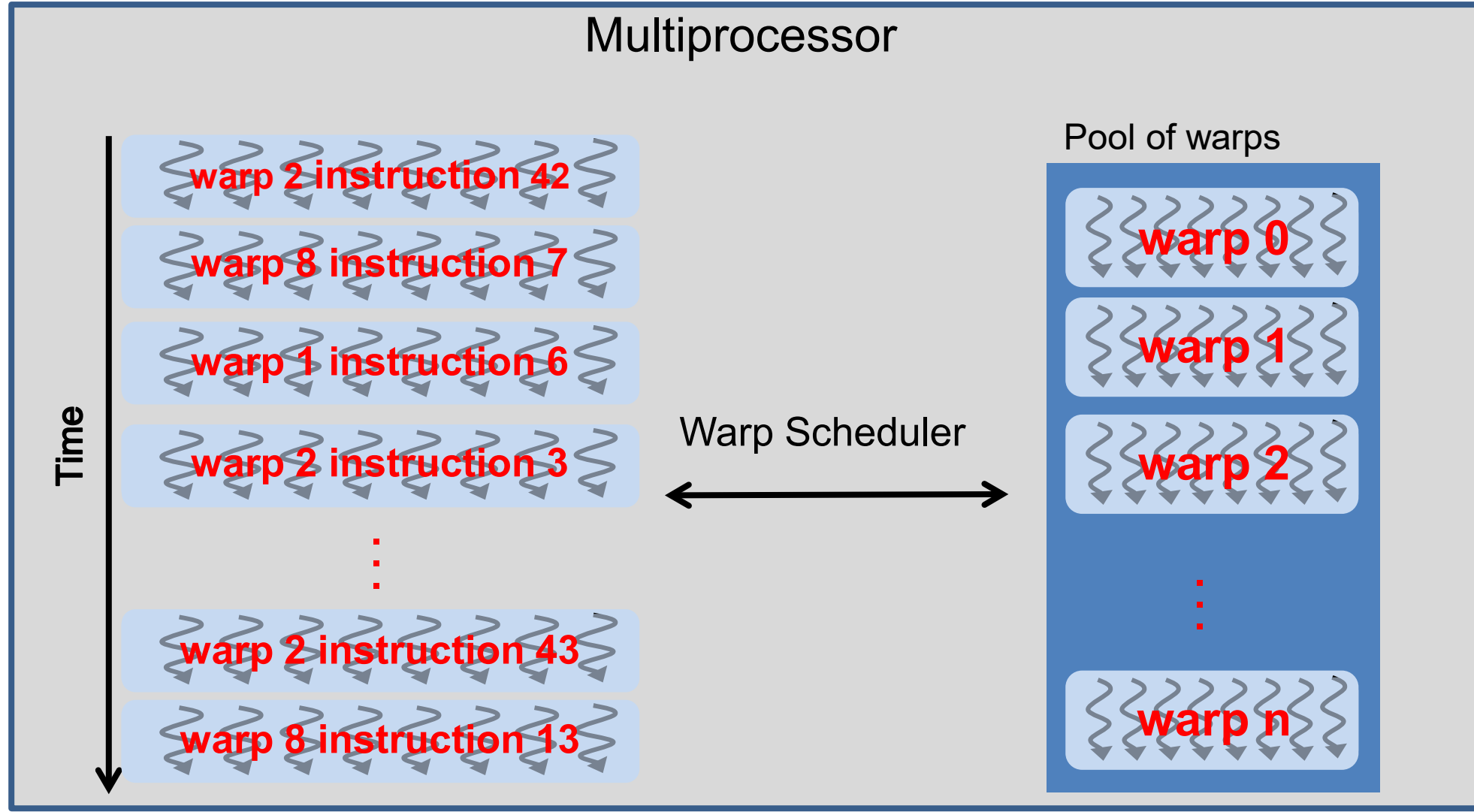
- Multiple independent instruction streams
- Example: Multicore CPUs
- (Outlook: will be introduced in the following lectures)

Single Instruction Multiple Threads (SIMT)

- Warps as scheduling units
 - Threads execute as groups of 32
 - Threads in warp share same program counter (execute a single instruction)
 - “Warp scheduler” schedules units of warps (not threads)
 - Less control logic needed
- Single instruction per warp → **SIMT** architecture
- Comparison to SIMD
 - SIMD is a multi-threaded approach
 - All threads process data in their own registers (e.g. no explicit packing)

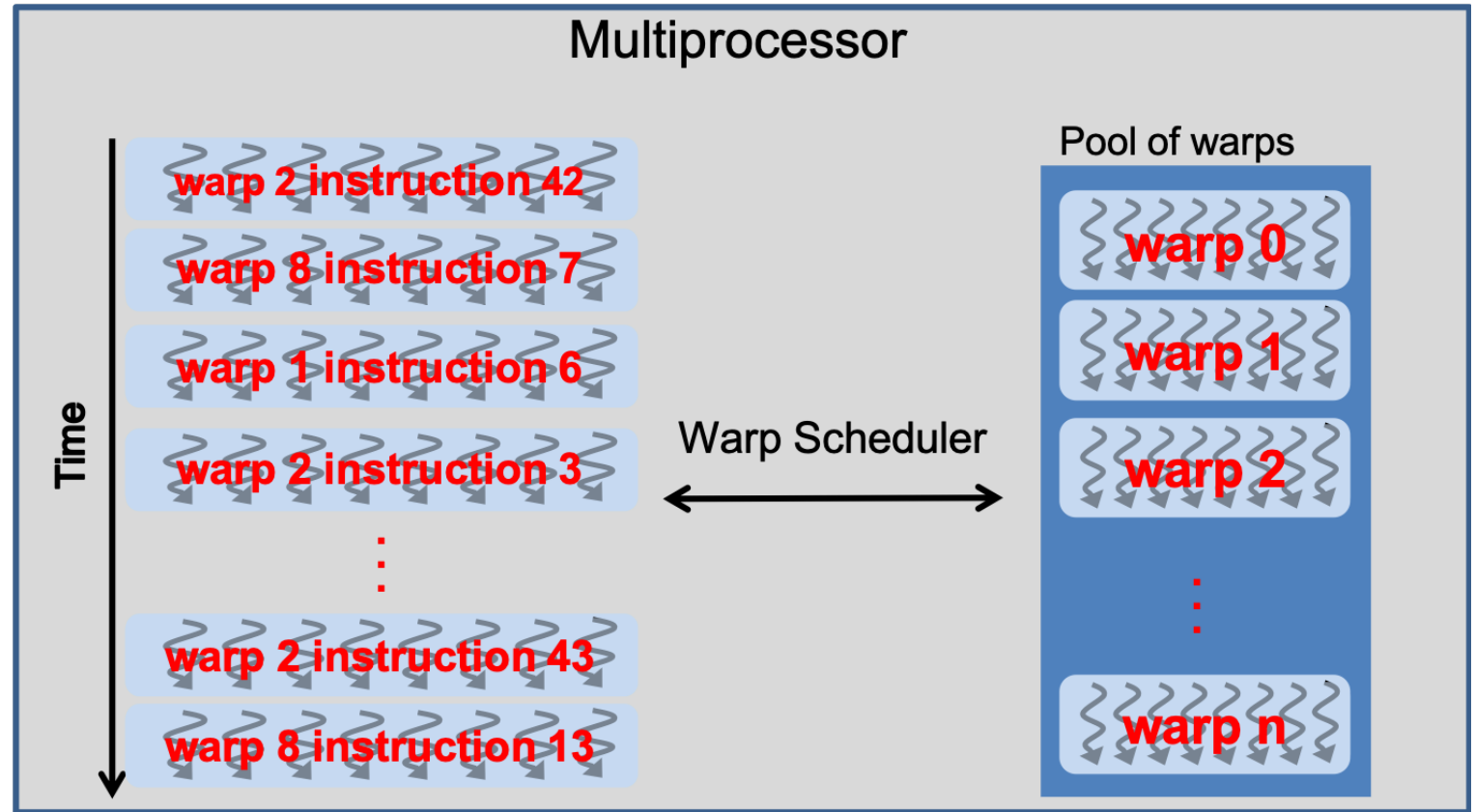


Warp Scheduling




Warp Scheduling (cont.)

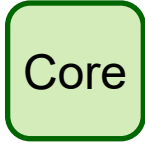
- Fast context switching for warps
 - Hide latencies introduced by, e.g., memory accesses, data dependencies, etc.
- Instruction is eligible for execution if all its operands are ready for consumption
- Scheduling policy required if multiple warps are eligible for execution
 - E.g. round robin, least recently fetched, fair policies, throughput-oriented policies



Mapping to Hardware

Thread


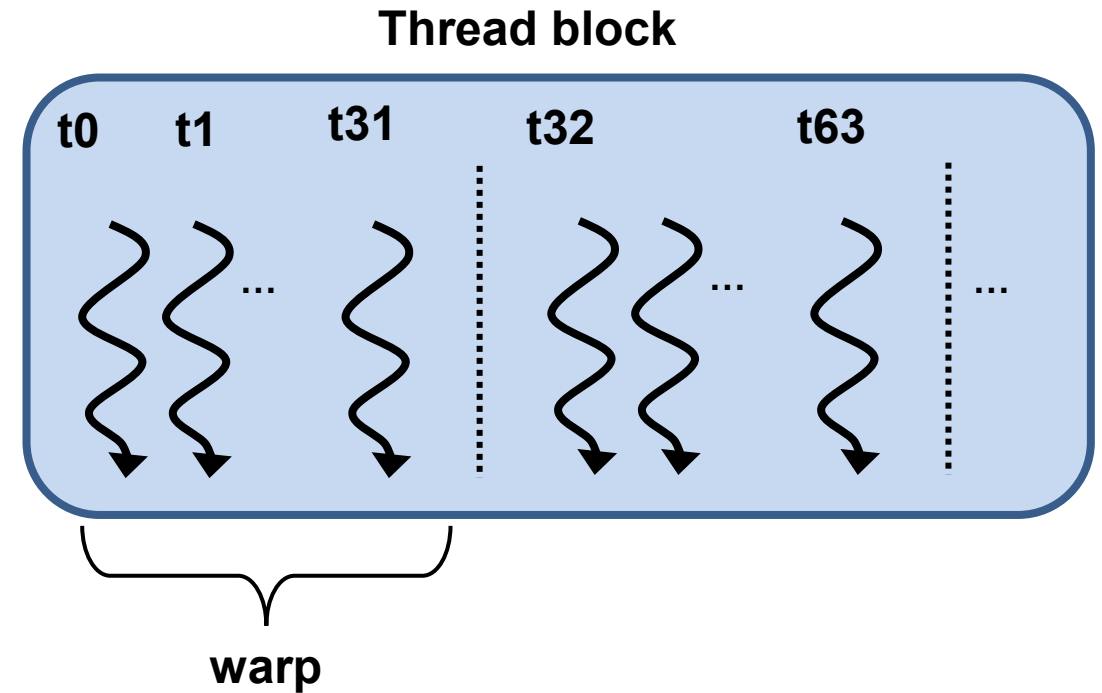


Core


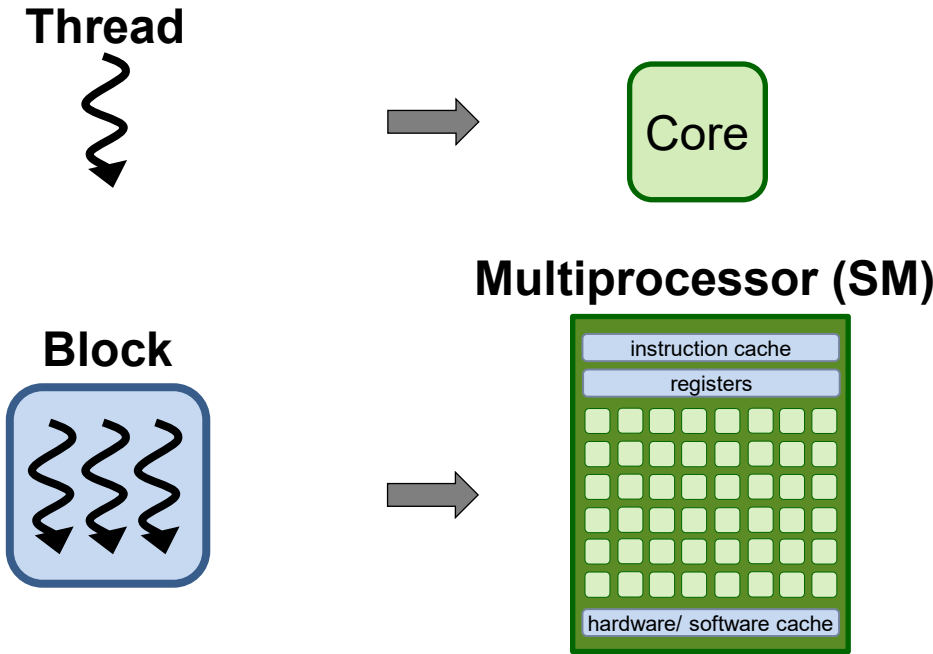
- Each thread is executed by a core

Thread Blocks

- Multiple warps are grouped into thread blocks
 - Enables cooperation of threads within a block
 - Block synchronization
 - Shared memory
- Threads inside a block are scheduled on the same multiprocessor
- How large should a block be?
 - Communication becomes slow with large number of threads
 - Typical number is up to 512
 - Better to schedule large problems to more blocks which can be scheduled onto multiple multiprocessors



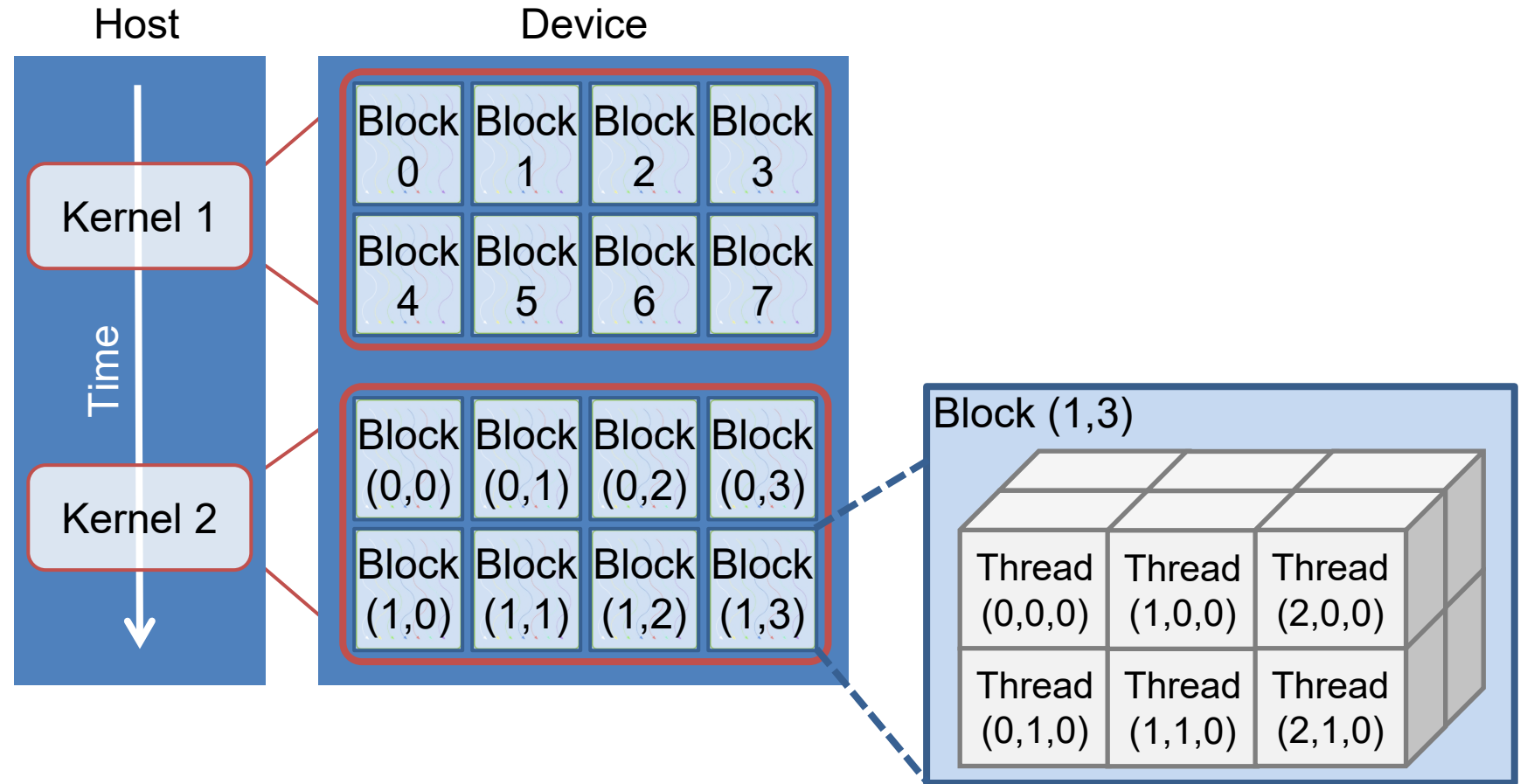
Mapping to Hardware



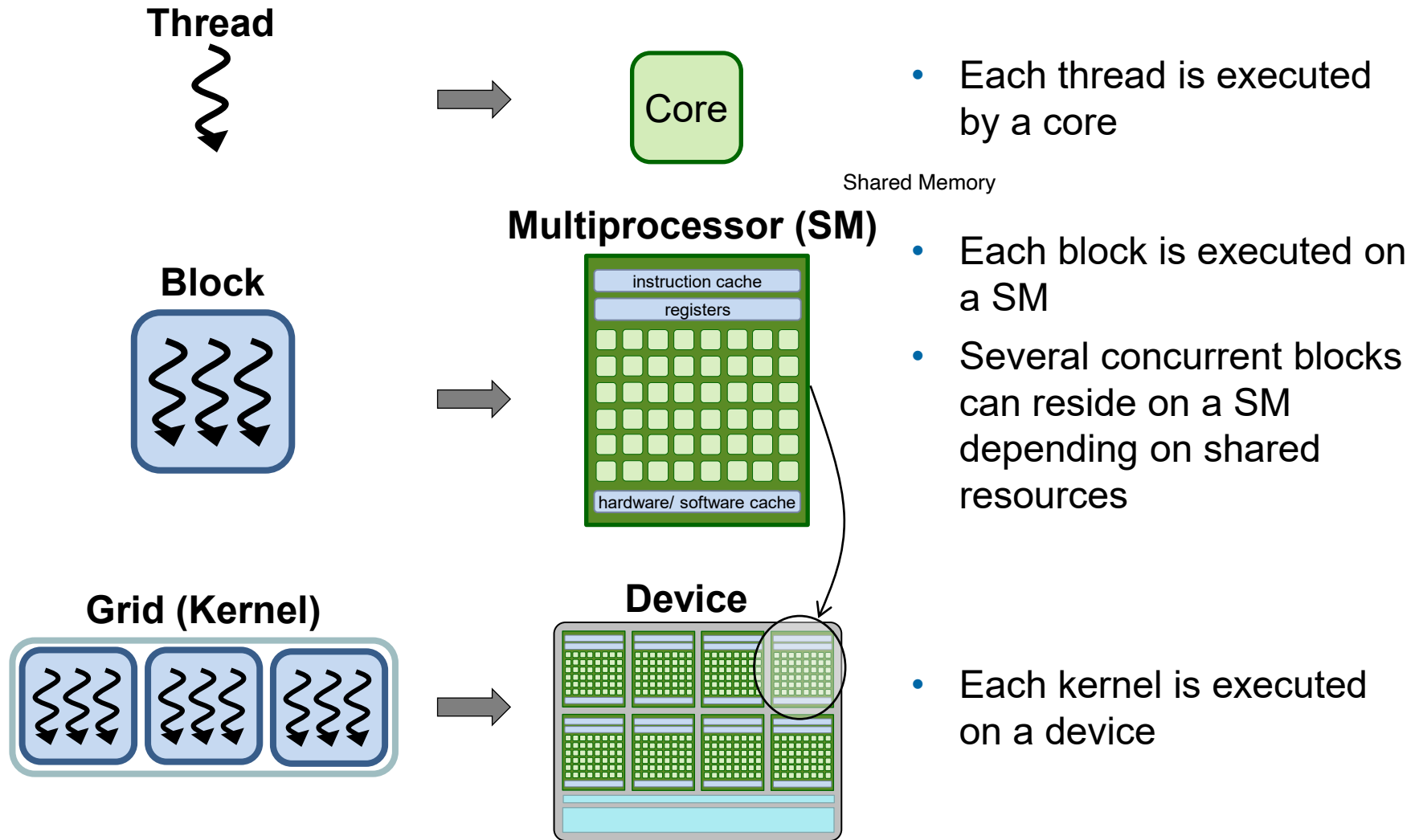
- Each thread is executed by a core
- Each block is executed on a SM
- Several concurrent blocks can reside on a SM depending on shared resources

Creating Parallelism - Logical Hierarchy

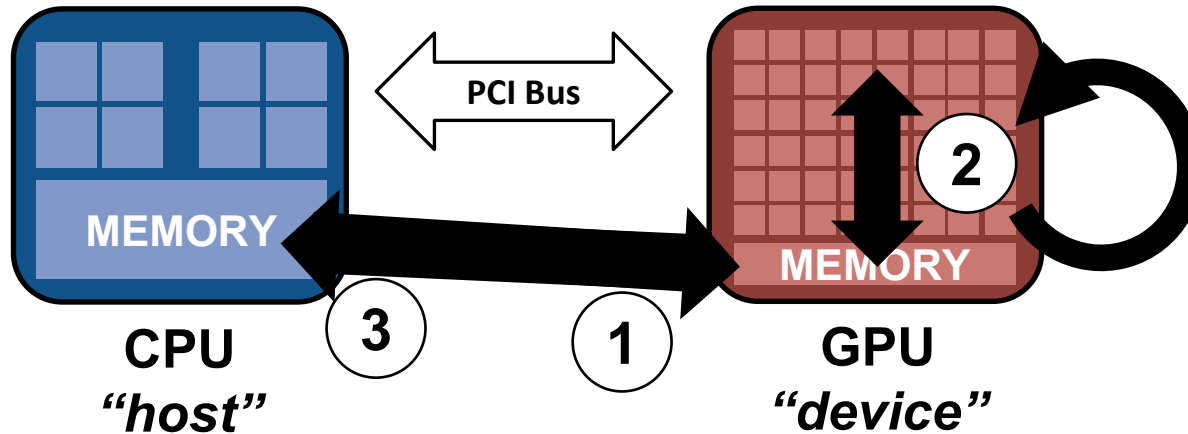
- Kernel is executed as a grid of blocks of threads
 - Similar structure to multidimensional arrays
 - Dimensions of blocks and grids: ≤ 3



Mapping to Hardware



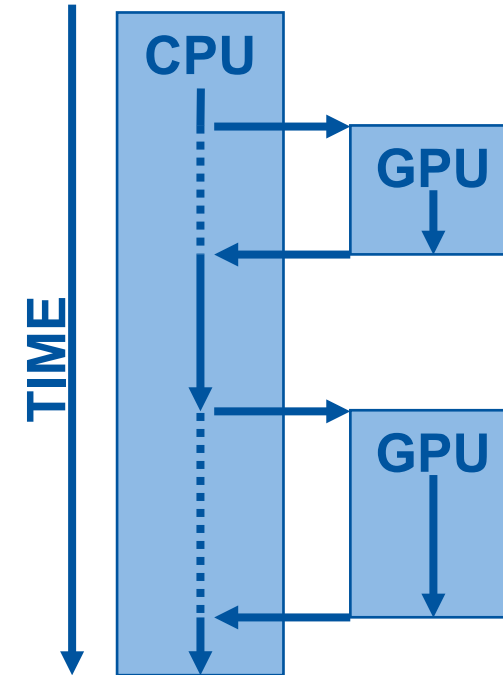
Offloading



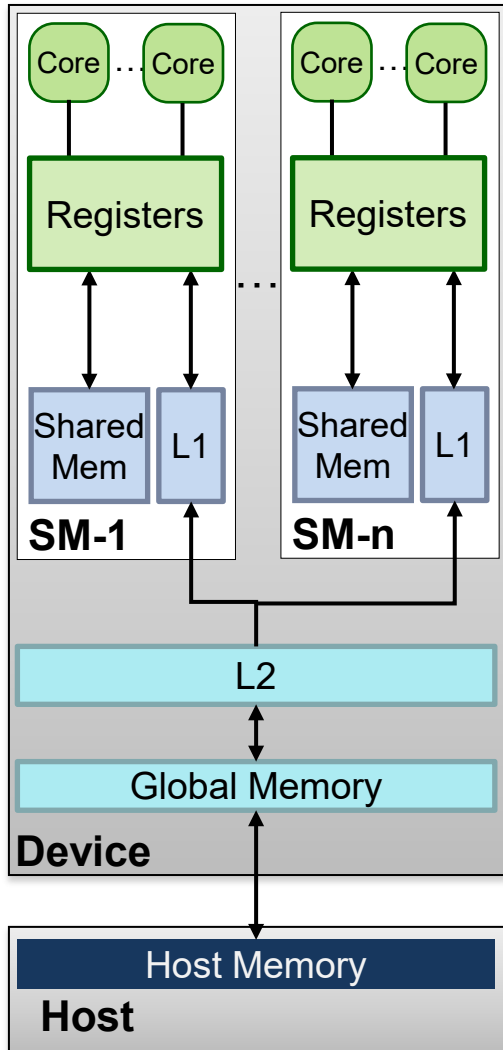
We refer to “discrete GPUs” here.


- Separate host and device memory
 - No coherence between host + device
 - **Data transfers** needed
- Host-directed execution model
 - Copy input data from CPU mem. to device mem.
 - Execute the device program
 - Copy results from device mem. to CPU mem.

processing flow (simplified)

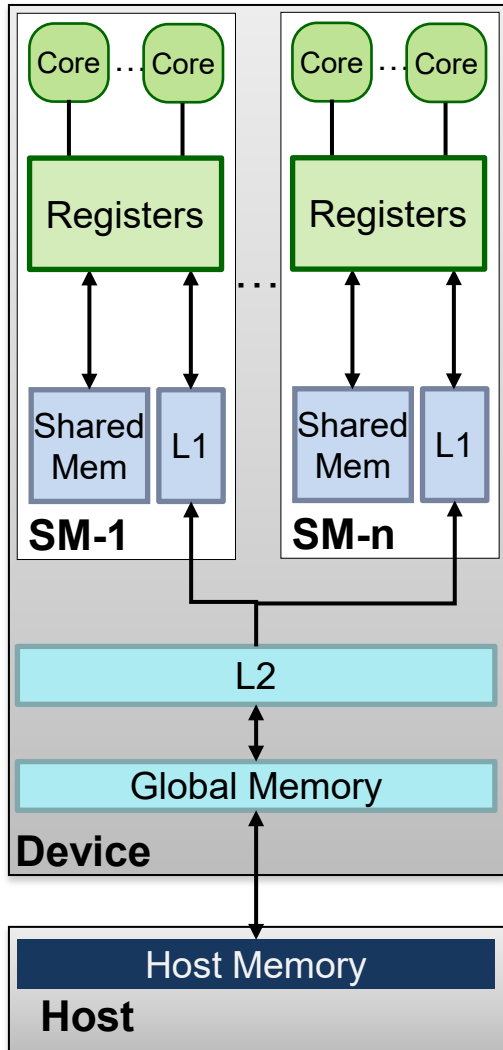




Memory & data management



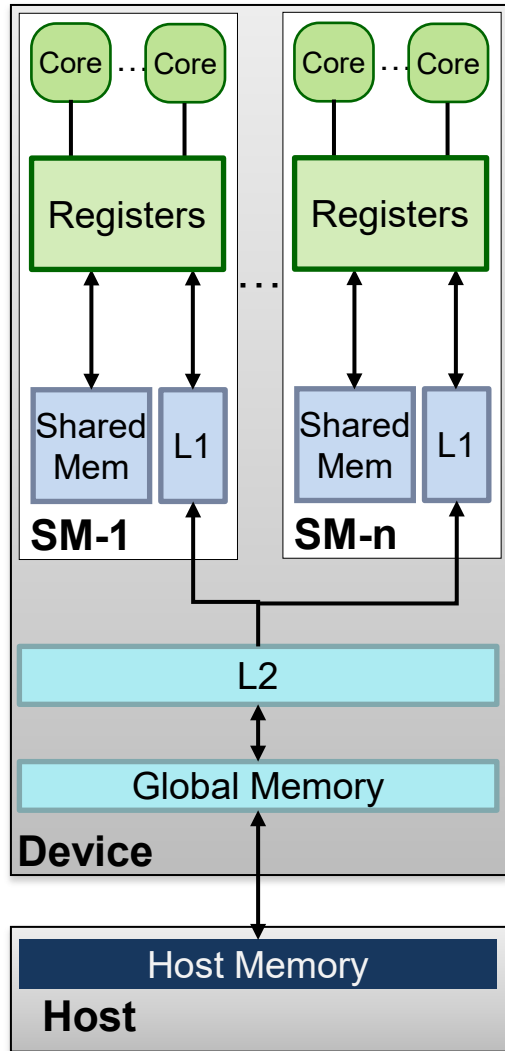
- Global memory (gmem)
 - Regular data transfer: between host & gmem
 - Recent GPU architectures: stacked memory (“HBM2”)
 - Size: V100: ~16 GB
 - Scope: all threads in a grid 
- L2 cache
 - Hardware cache, “cache lines”
 - Size: V100: 6 MB
- L1 cache
 - Hardware cache (split w/ smem), “cache lines”
 - Size: V100: up to 128 KB/SM

Memory & data management



- Shared memory (smem)
 - Software cache (split w/ L1), manually managed
 - Size: V100: up to 128 KB/SM
 - Scope: threads within block 
- Registers
 - Thread-local memory
 - Size: V100: 256 KB/SM
 - max 255 (32-bit) registers per thread
 - Scope: thread 
- Special memories/ memory regions
 - Local memory, texture cache, constant memory
 - Not discussed here

Memory & data management



- Memory hierarchy (fast to slow)

1. Registers
2. Shared memory
3. Global memory
e.g, V100 gmem: 900 GB/s (peak), several 100s cycles latency

Memory Consistency Model

- Definition in CUDA: “In multi-threaded executions, the side-effects of memory operations performed by each thread become visible to other threads in a partial and non-identical order. This means that any two operations may appear to happen in no order, or in different orders, to different threads.”

<https://docs.nvidia.com/cuda/parallel-thread-execution/index.html#memory-consistency-model>

Memory Consistency Model (cont.)

- Weakly-ordered memory model

```
// Global
__device__ volatile int X = 1, Y = 2;

// Executed by thread 1
__device__ void writeXreadY() {
    X = 10;
    int A = Y;
}

// Executed by thread 2
__device__ void writeYreadX() {
    Y = 20;
    int B = X;
}
```

Multiple results possible:

- A = 2 and B = 10
- A = 20 and B = 1
- A = 20 and B = 10
- A = 2 and B = 1

(not possible in a strongly-ordered memory model)

- Enforced ordering required to prohibit the last case

<https://docs.nvidia.com/cuda/parallel-thread-execution/index.html#memory-consistency-model>

Memory Consistency Model (cont.)

- Enforced ordering with memory fences required
- Memory fence ensures that:
 - All writes to all memory made by the calling thread before the fence are observed by all threads as occurring before all writes to all memory made by the calling thread after the fence.
 - All reads from all memory made by the calling thread before the fence are ordered before all reads from all memory made by the calling thread after the fence.
- Fences between both statements in each thread would prohibit case $A = 2$ and $B = 1$

<https://docs.nvidia.com/cuda/parallel-thread-execution/index.html#memory-consistency-model>

Summary

