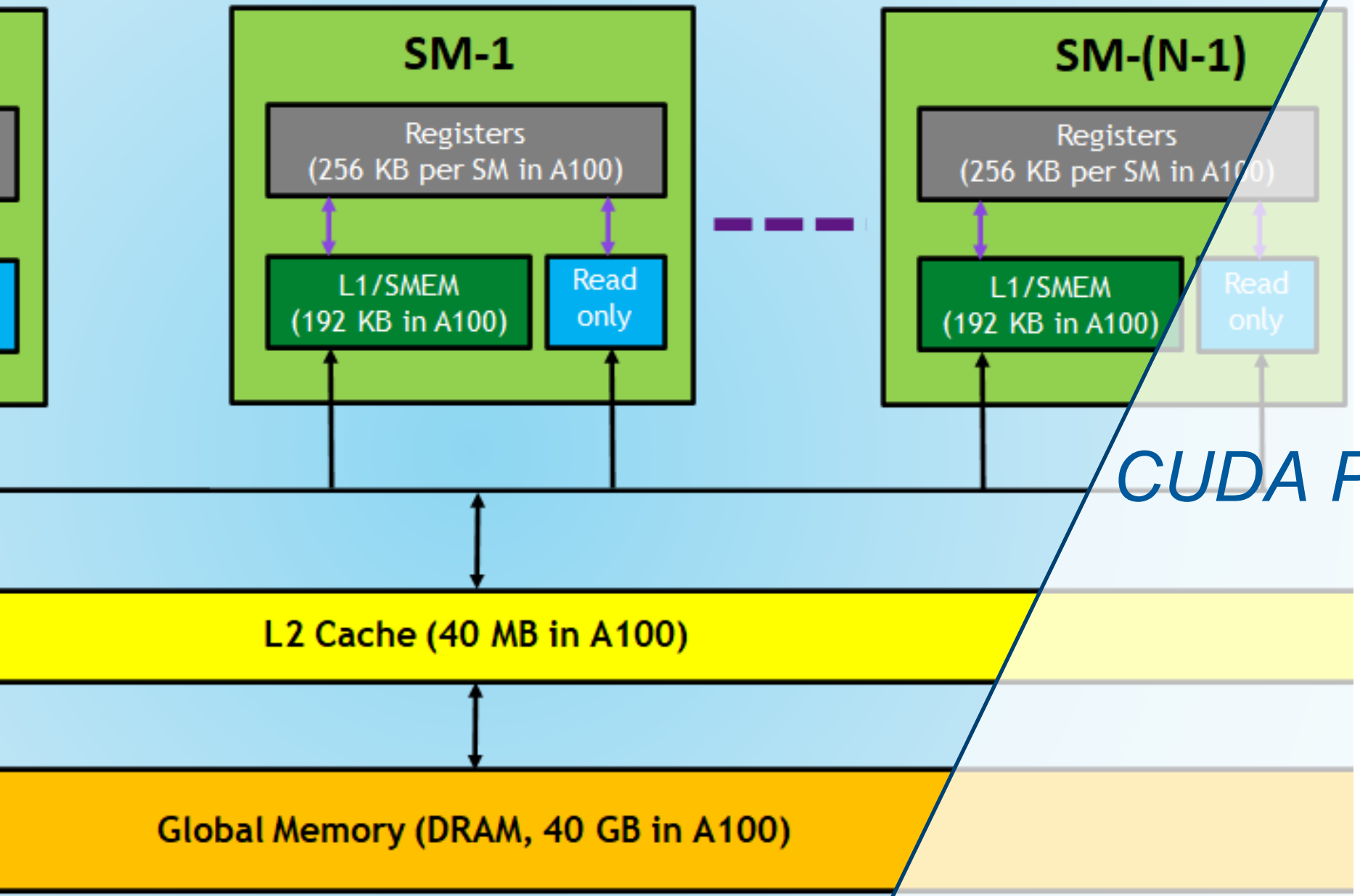


CUDA Programming Model

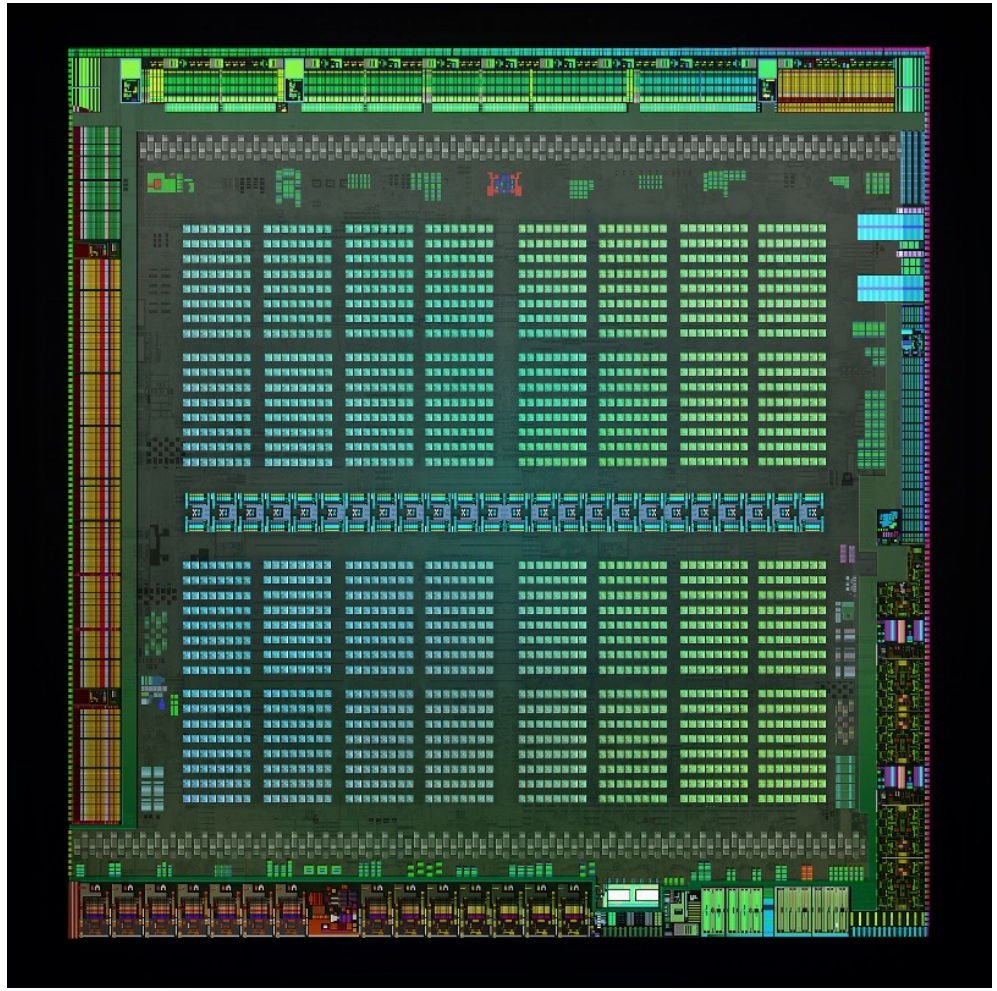
22.03.23

CUDA Programming Model

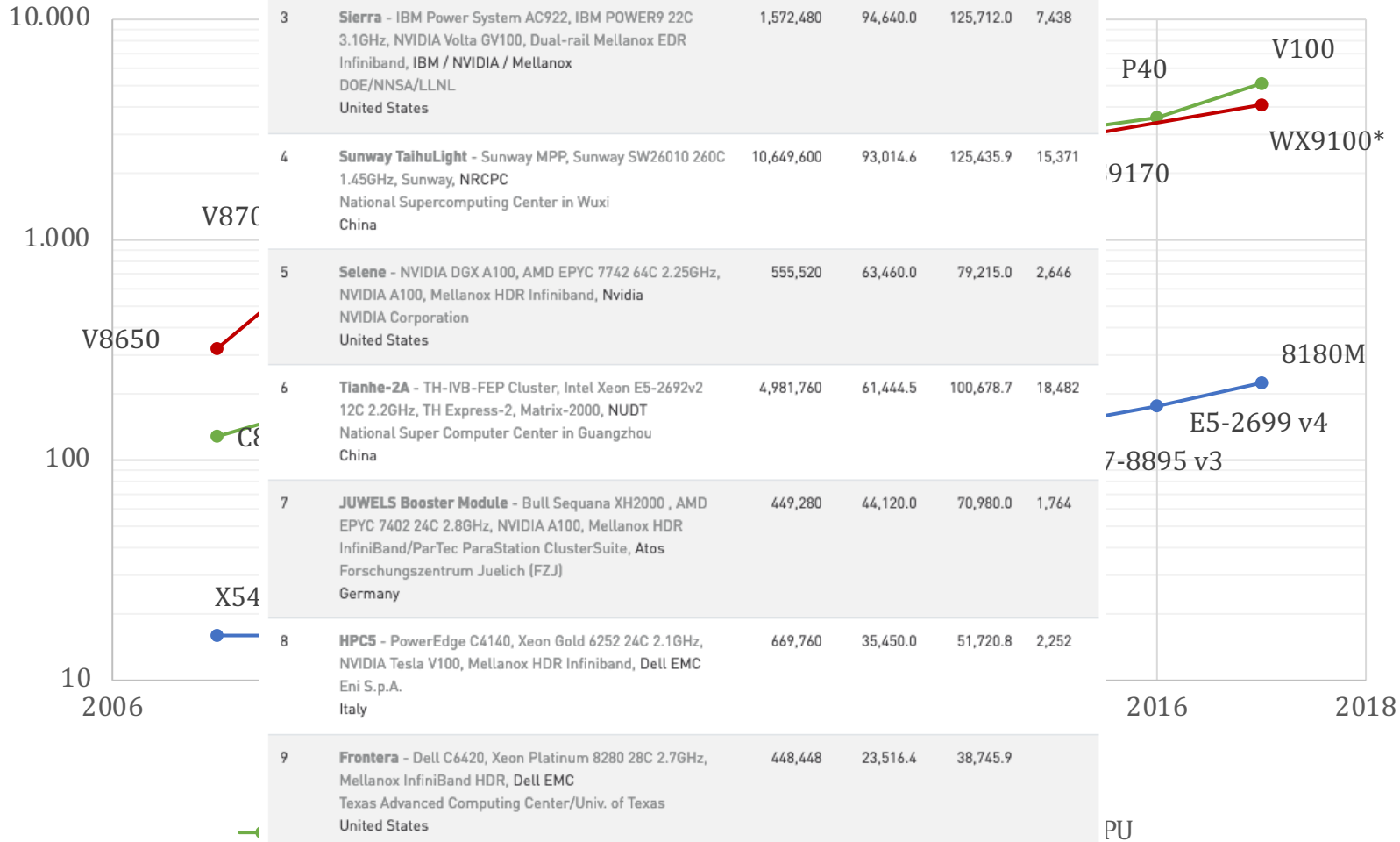
- Basic understanding of GPU hardware architecture and its difference to the CPU
- Ability to write simple parallel programs that run on the GPU
- Understanding of the different GPU memory types and the ability to use them in a simple program

NVIDIA CUDA

- Compute Unified Device Architecture



Rank	System	Cores	Rmax (TFlop/s)	Rpeak (TFlop/s)	Power (kW)
1	**Supercomputer Fugaku** - Supercomputer Fugaku, A64FX 48C 2.2GHz, Tofu interconnect D, Fujitsu RIKEN Center for Computational Science Japan	7,630,848	442,010.0	537,212.0	29,899
2	**Summit** - IBM Power System AC922, IBM POWER9 22C 3.07GHz, NVIDIA Volta GV100, Dual-rail Mellanox EDR Infiniband, IBM DOE/SC/Oak Ridge National Laboratory United States	2,414,592	148,600.0	200,794.9	10,096
3	**Sierra** - IBM Power System AC922, IBM POWER9 22C 3.1GHz, NVIDIA Volta GV100, Dual-rail Mellanox EDR Infiniband, IBM / NVIDIA / Mellanox DOE/NNSA/LLNL United States	1,572,480	94,640.0	125,712.0	7,438
4	**Sunway TaihuLight** - Sunway MPP, Sunway SW26010 260C 1.45GHz, Sunway, NRCPC National Supercomputing Center in Wuxi China	10,649,600	93,014.6	125,435.9	15,371
5	**Selene** - NVIDIA DGX A100, AMD EPYC 7742 64C 2.25GHz, NVIDIA A100, Mellanox HDR Infiniband, Nvidia NVIDIA Corporation United States	555,520	63,460.0	79,215.0	2,646
6	**Tianhe-2A** - TH-IVB-FEP Cluster, Intel Xeon E5-2692v2 12C 2.2GHz, TH Express-2, Matrix-2000, NUDT National Super Computer Center in Guangzhou China	4,981,760	61,444.5	100,678.7	18,482
7	**JUWELS Booster Module** - Bull Sequana XH2000 , AMD EPYC 7402 24C 2.8GHz, NVIDIA A100, Mellanox HDR InfiniBand/ParTec ParaStation ClusterSuite, Atos Forschungszentrum Juelich (FZJ) Germany	449,280	44,120.0	70,980.0	1,764
8	**HPC5** - PowerEdge C4140, Xeon Gold 6252 24C 2.1GHz, NVIDIA Tesla V100, Mellanox HDR Infiniband, Dell EMC Eni S.p.A. Italy	669,760	35,450.0	51,720.8	2,252
9	**Frontera** - Dell C6420, Xeon Platinum 8280 28C 2.7GHz, Mellanox InfiniBand HDR, Dell EMC Texas Advanced Computing Center/Univ. of Texas United States	448,448	23,516.4	38,745.9	
10	**Dammam-7** - Cray CS-Storm, Xeon Gold 6248 20C 2.5GHz, NVIDIA Tesla V100 SXM2, InfiniBand HDR 100, HPE Saudi Aramco Saudi Arabia	672,520	22,400.0	55,423.6	



Amdahl's Law

- Given
  - a computation that takes time T_{seq} to complete sequentially
  - N processors
- How much faster can we go?

$$\text{Speedup} \quad S = \frac{T_{seq}}{T_{par}}$$

Amdahl's Law

- Not every operation can be parallelized
- Speedup limited by
  - How much of our program can be parallelized
  - How many processors we can use to run code in parallel

- Assume:
  - p fraction of our program that is parallelizable
  - $s = 1 - p$ sequential fraction

Amdahl's Law

- Purely sequential:

$$T_1 = (1 - p)T_{seq} + pT_{seq}$$

- Using N processors:

$$T_N = (1 - p)T_{seq} + \frac{p}{N}T_{seq}$$

$$S = \frac{T_1}{T_N} = \frac{1}{(1 - p) + \frac{p}{N}} = \frac{1}{s + \frac{p}{N}}$$

Amdahl's Law

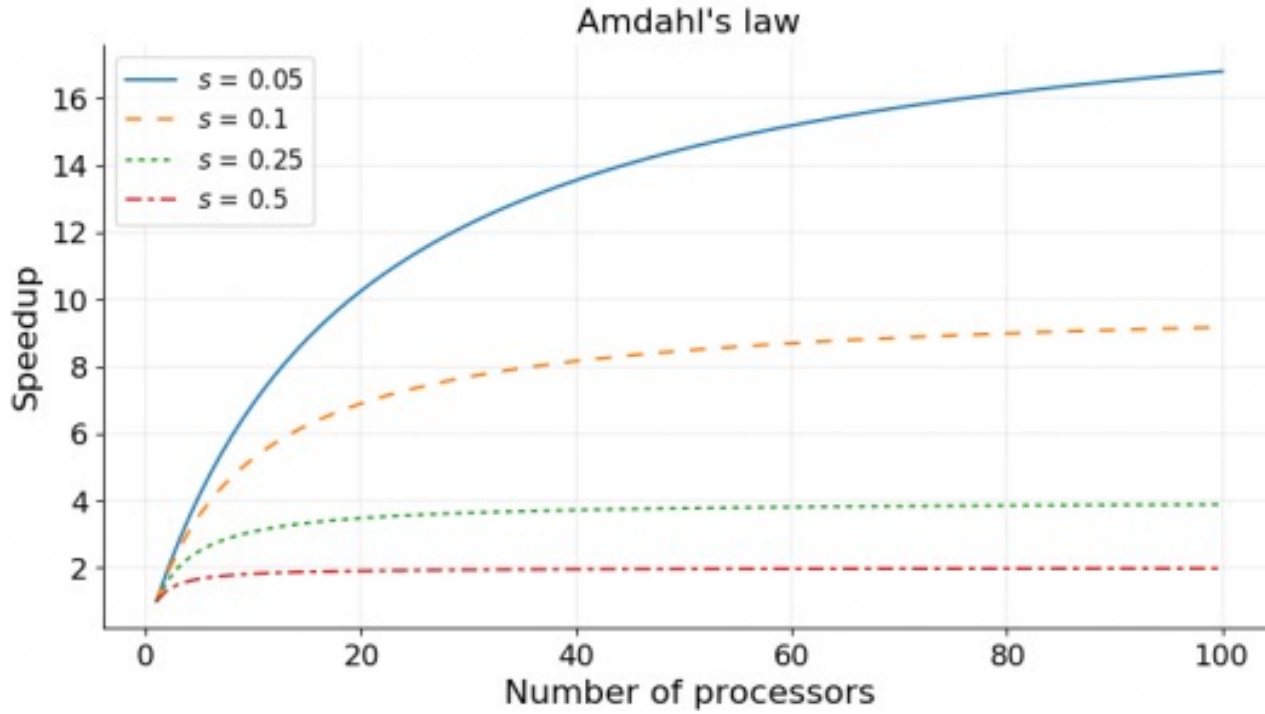
- Amdahl's law from 1967
- Speedup S over purely sequential implementation

$$S = \frac{1}{s + \frac{p}{N}}$$

s: fraction of sequential code

p: fraction of parallel code

N: # of processors

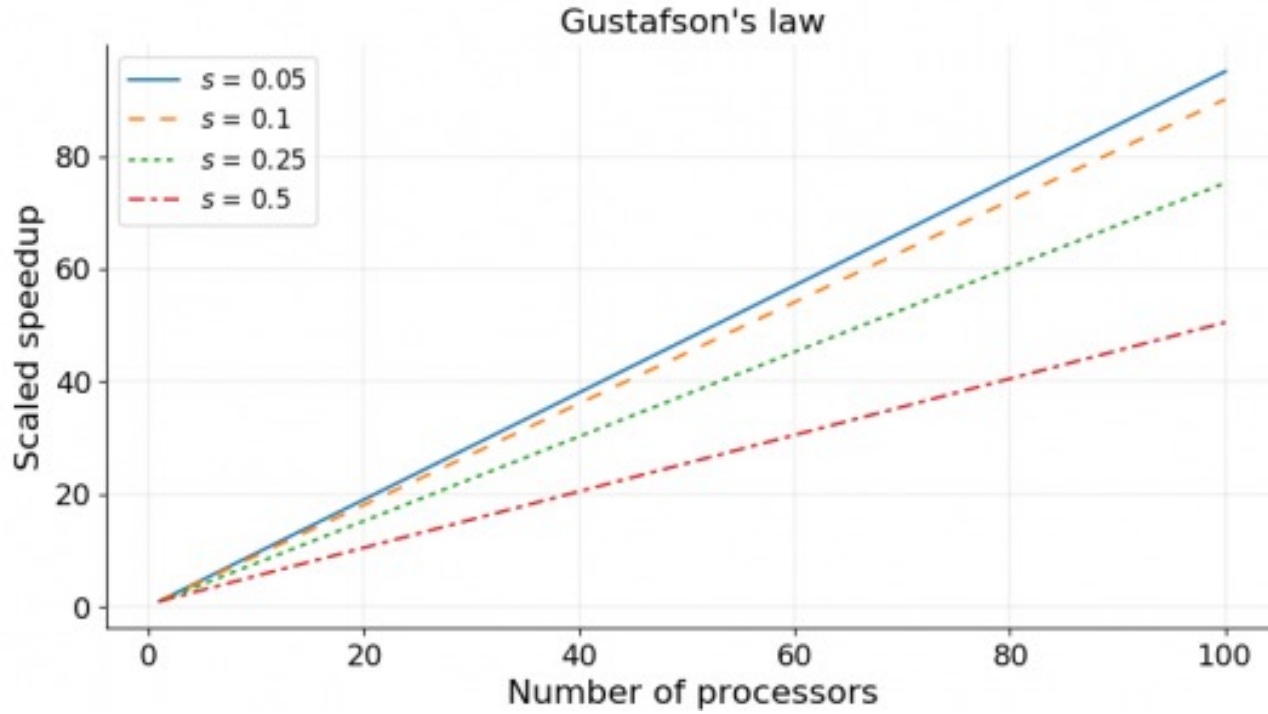


- Strong scaling
  - Limited by sequential fraction
  - Problem size fixed → scale processors

Gustafson's Law

- Gustafson's law from 1988
- Speedup S over purely sequential implementation

$$S = s + p \cdot N$$



s: fraction of sequential code

p: fraction of parallel code

N: # of processors

- Weak scaling
  - Sequential fraction less important
  - Problem size scales with processors

Types of Parallelism

Task Parallelism

- Parallelize different, independent computation
- Distribute tasks to processors
- Examples: multitasking, pipeline parallelism

Data Parallelism

- Parallelize same computation on different, independent data
- Distribute data to processors
- Examples: image processing, loop-level parallelism, divide and conquer

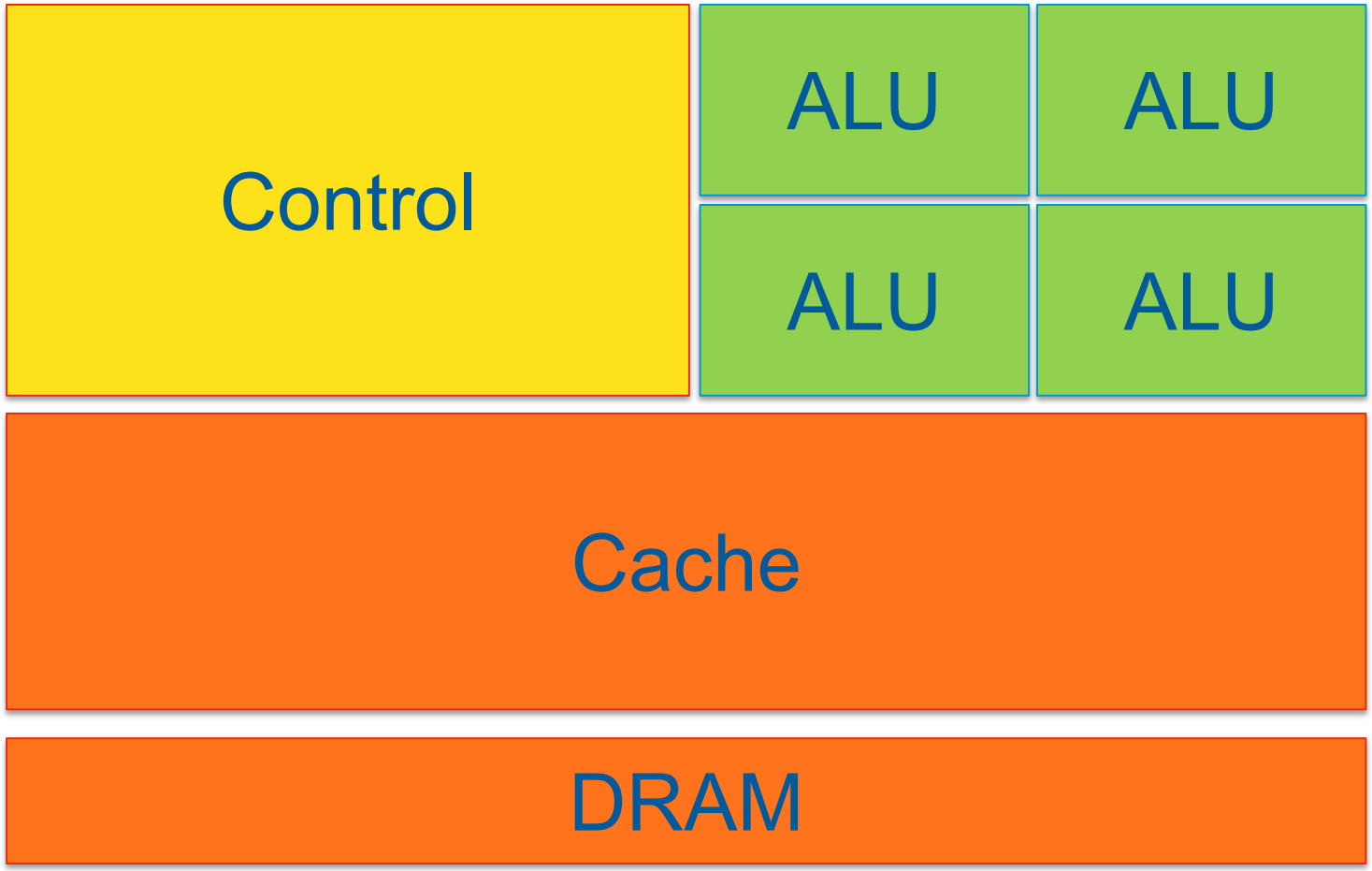
GPU Architecture

CPU Architecture

- Goal
  - Get the best performance for a single heavy-weight thread
- Big data caches
- Low-latency arithmetic units
- Complex control logic
  - Branch prediction
  - Out-of-order execution

  → latency-oriented design

CPU Architecture

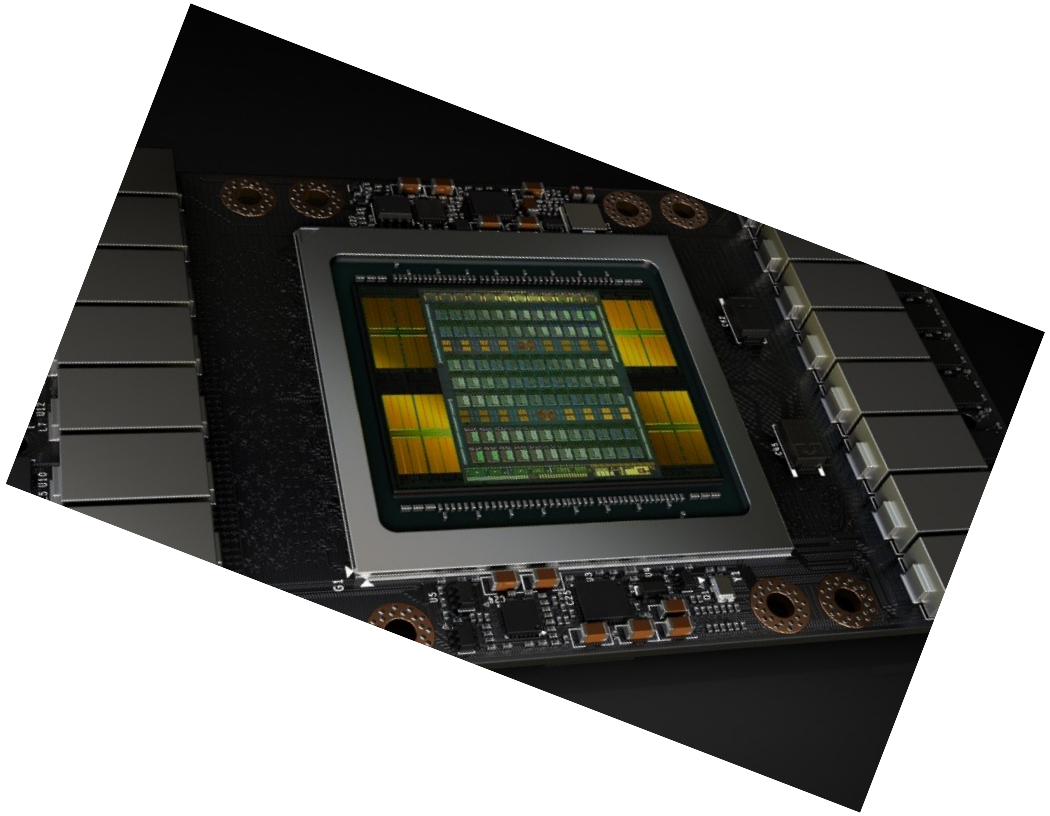


GPU Architecture

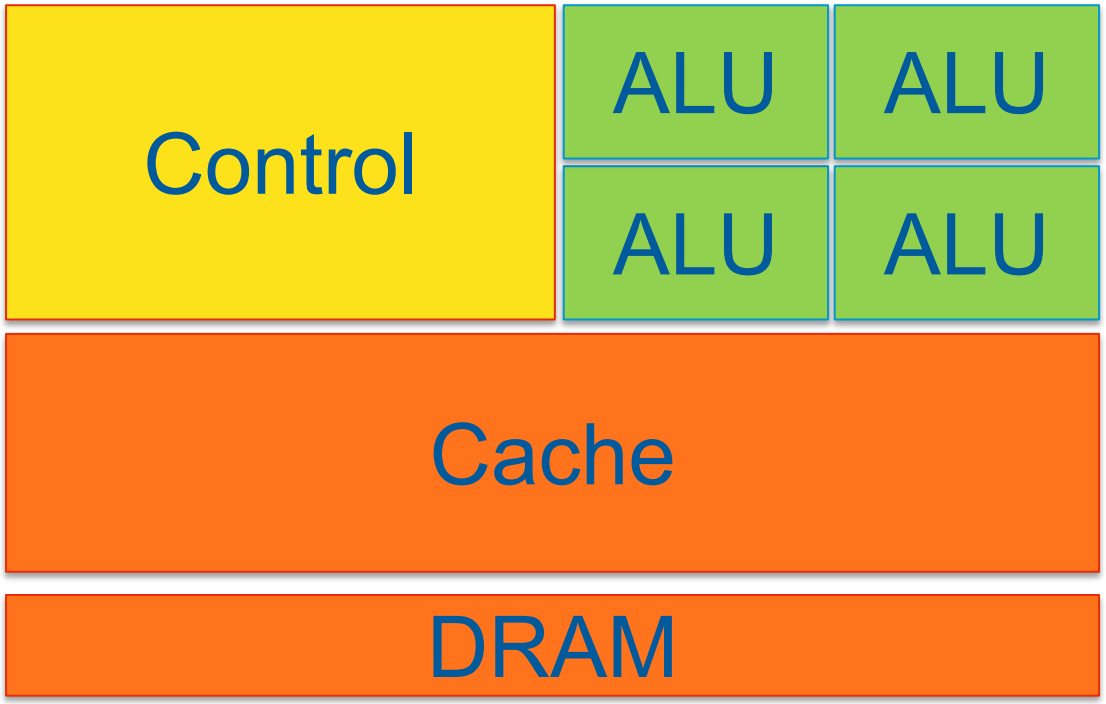
- Goal
  - Get the best performance

    for thousands of simple threads
- Small caches
- Hide latency with computation
- In-order execution with no branch prediction
- Issue the same command to multiple cores

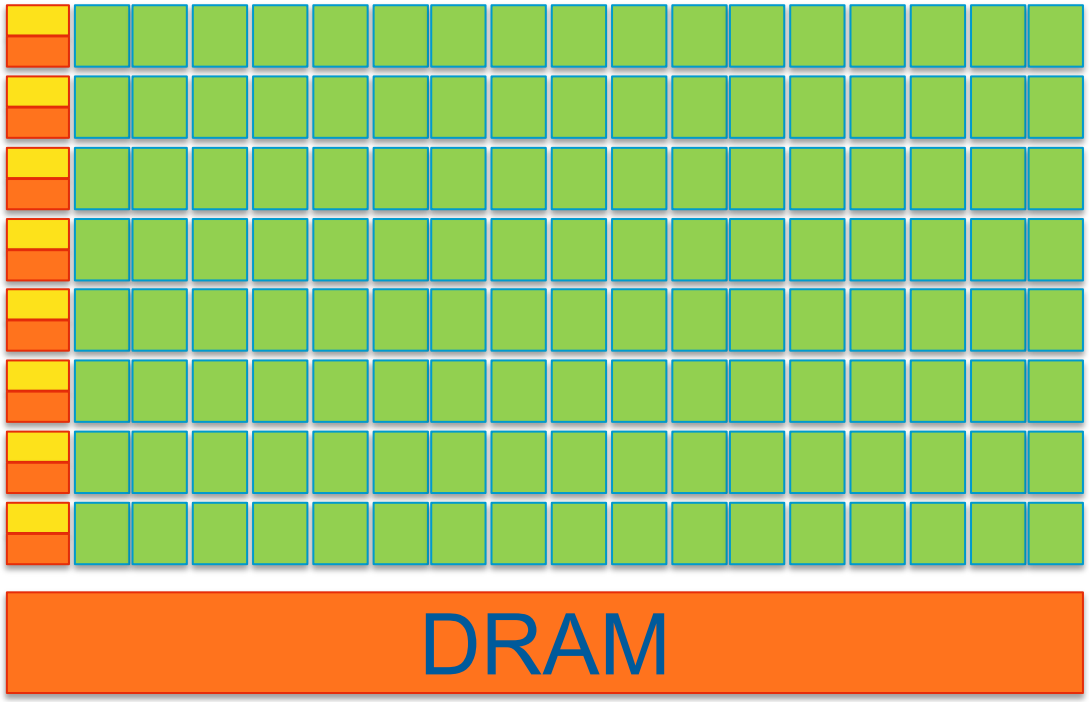
  → throughput-oriented design



CPU vs. GPU Architecture



CPU



GPU

Flynn's Taxonomy

- SISD — single-instruction, single-data
  (e.g., single core CPU)
- MIMD — multiple-instruction, multiple-data
  (e.g., multi core CPU)
- ~~SIMD~~ SIMT — single-instruction, multiple-data
  (e.g., data-based parallelism)
- MISD — multiple-instruction, single-data
  (e.g., fault-tolerant computers)

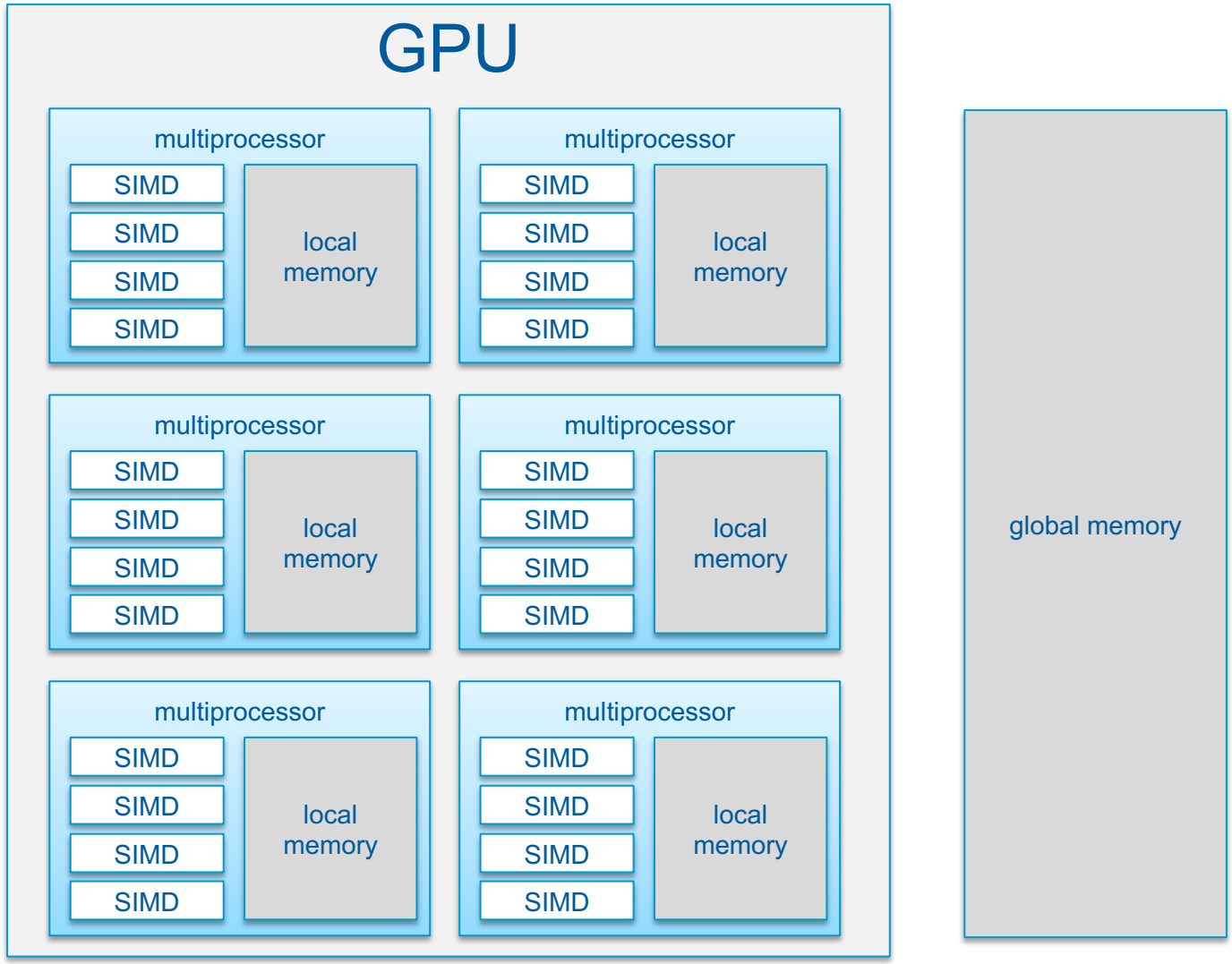
SIMD vs. SIMT

Both execute a single instruction on multiple processing elements

- SIMD exposes SIMD width to software
- SIMT exposes thread-level
  - Write code for independent, scalar threads
  - Threads can diverge within SIMD width

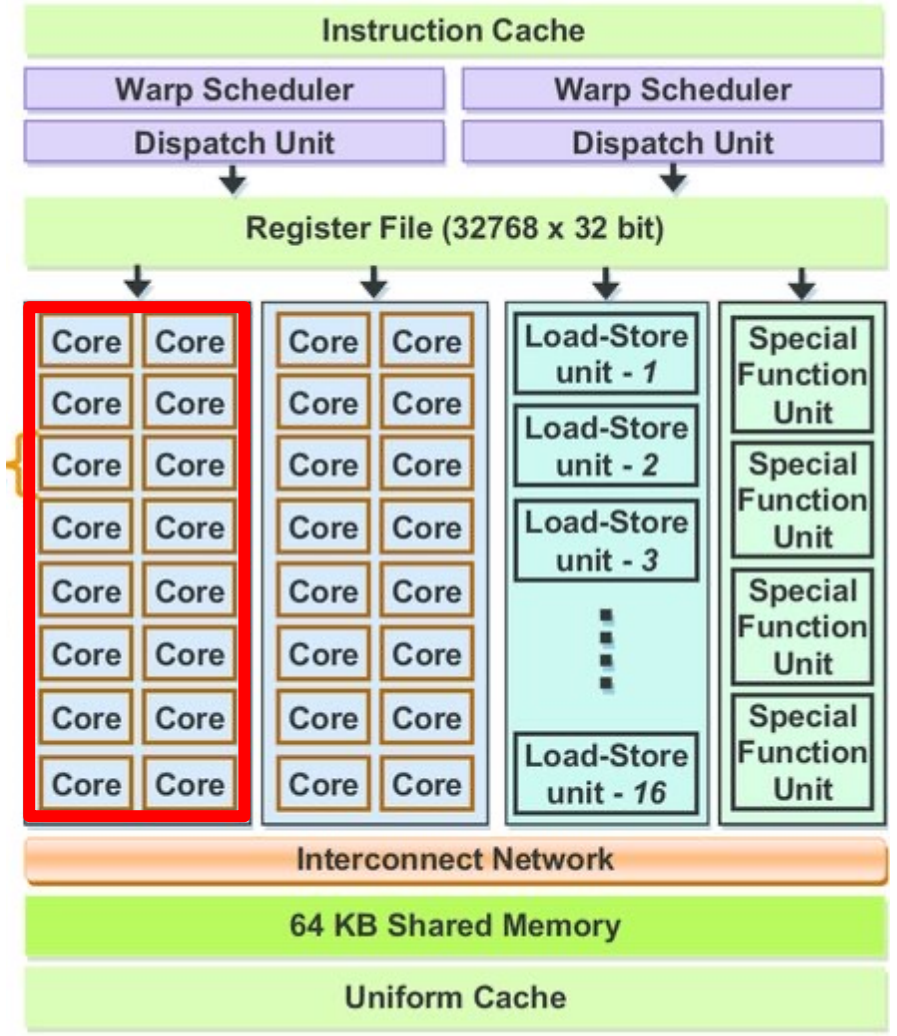
CUDA GPU (Volta)

CUDA GPU


GPU
multiprocessor
SIMD
SIMD
SIMD
SIMD
local memory
multiprocessor
SIMD
SIMD
SIMD
SIMD
local memory
multiprocessor
SIMD
SIMD
SIMD
SIMD
local memory
multiprocessor
SIMD
SIMD
SIMD
SIMD
local memory
multiprocessor
SIMD
SIMD
SIMD
SIMD
local memory
multiprocessor
SIMD
SIMD
SIMD
SIMD
local memory
global memory

Streaming Multiprocessor (SM/SMX)

- SIMD hardware
- 32 threads form a *warp*
- Each thread within a warp must execute the same instruction (or be deactivated)
- **1 instruction → 32 values computed**
- Instruction / memory latency

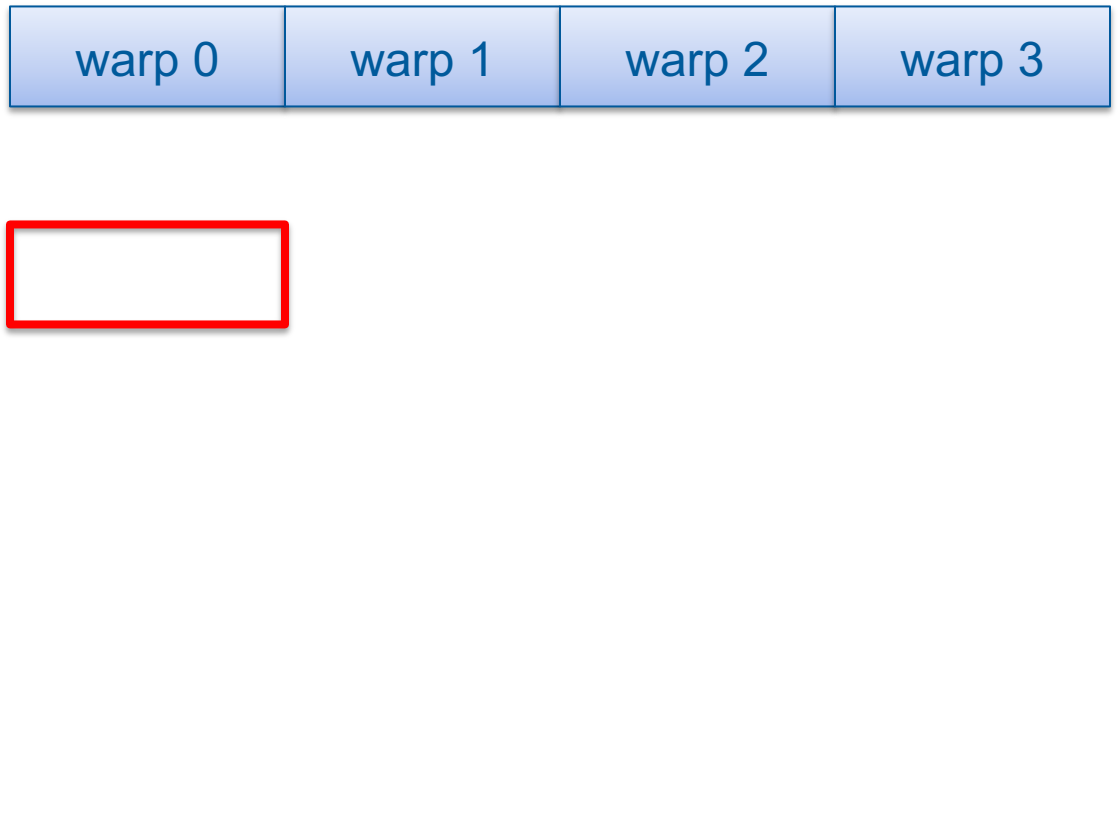


Memory Latency

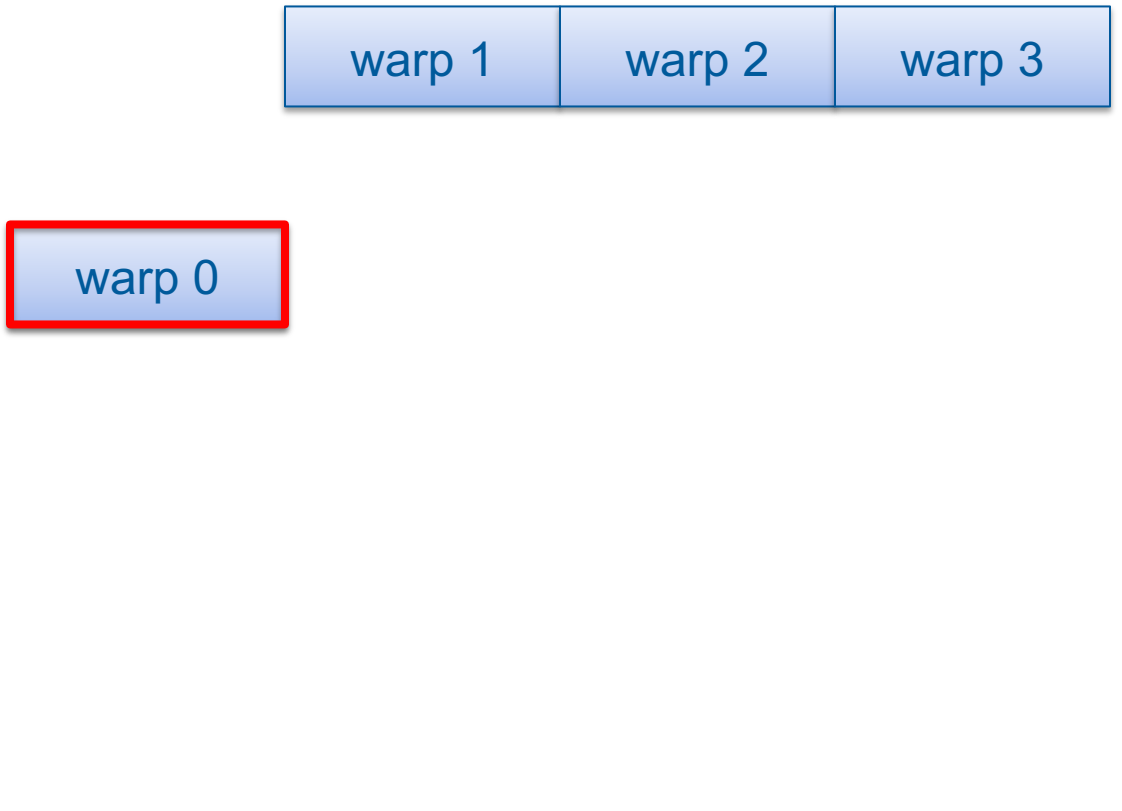
	Volta (in cycles)
L1 hit	28
L1 miss – L2 hit	193
L2 miss – TLB hit	375
TLB miss	1029

→ handle more warps than cores to hide latency

Warp Scheduling

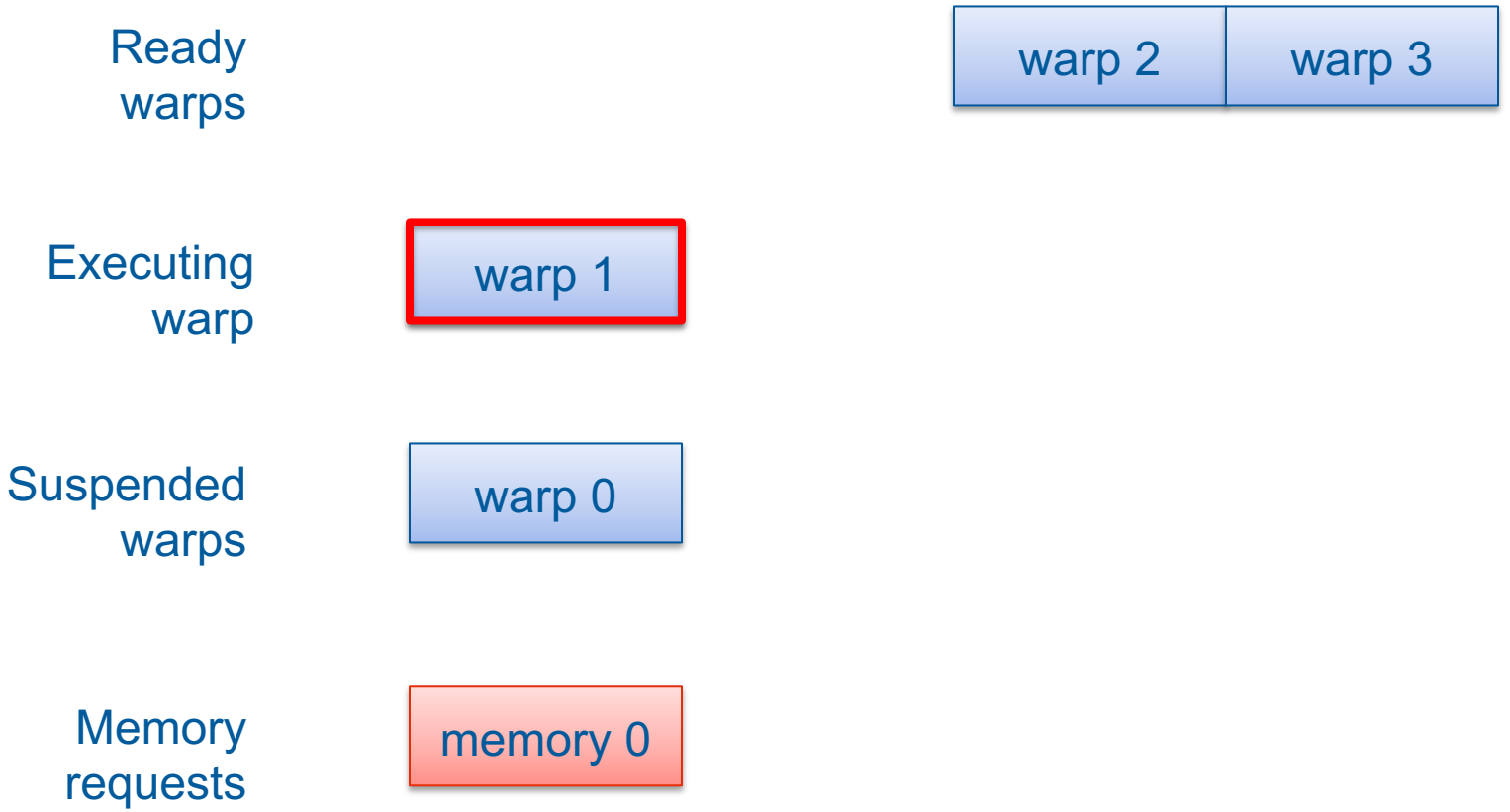


Warp Scheduling

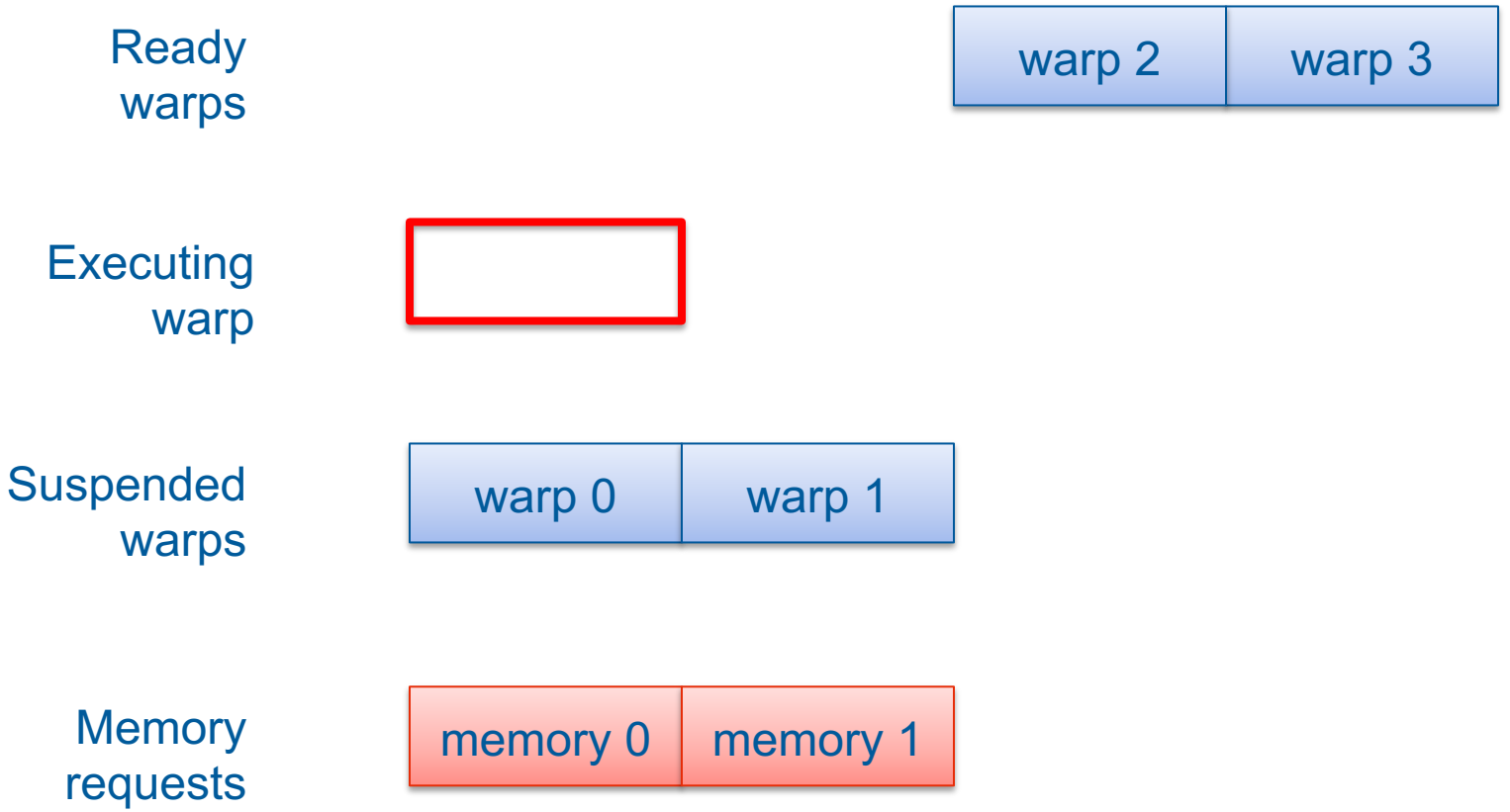


Warp Scheduling

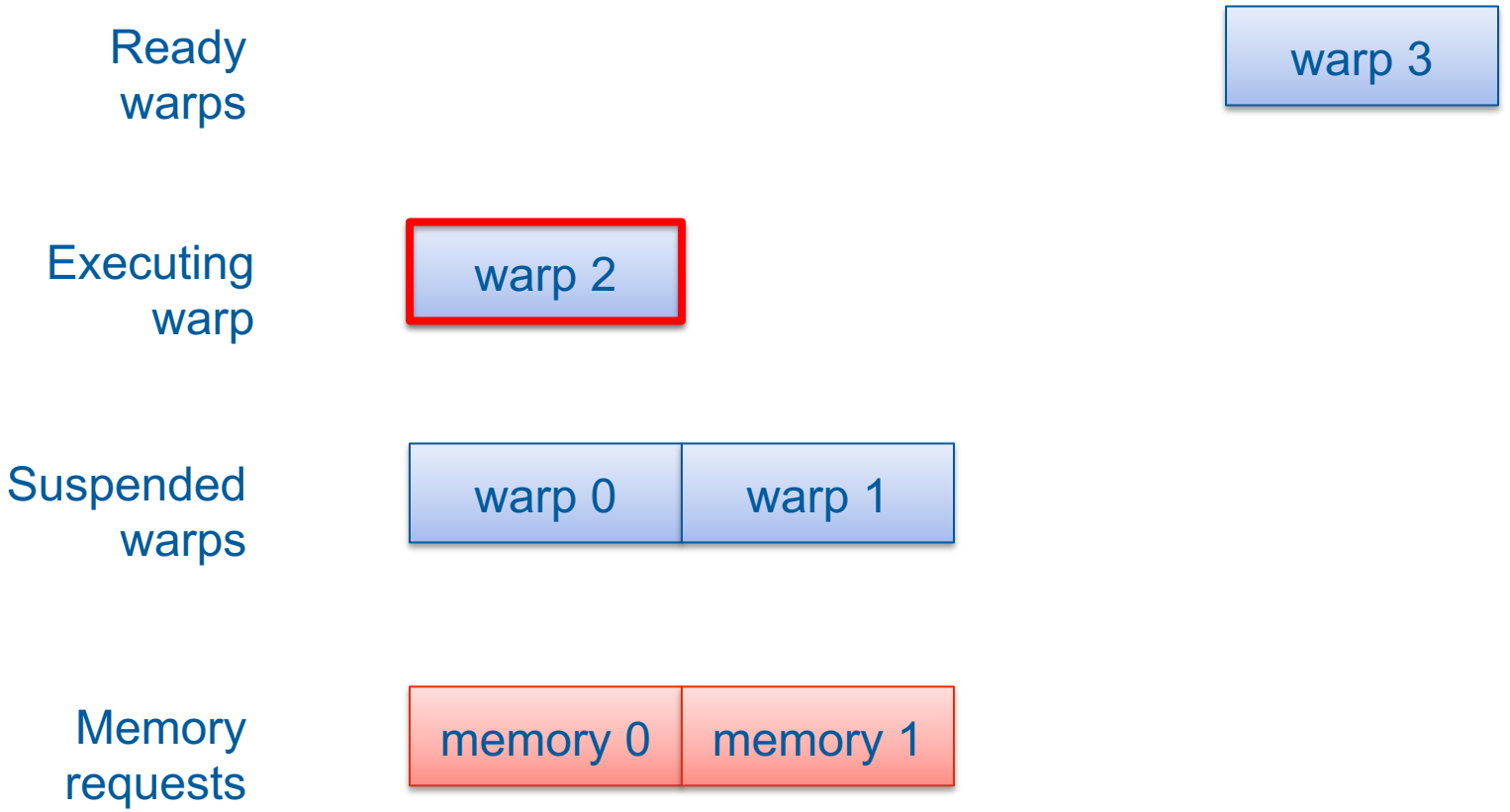
Warp Scheduling



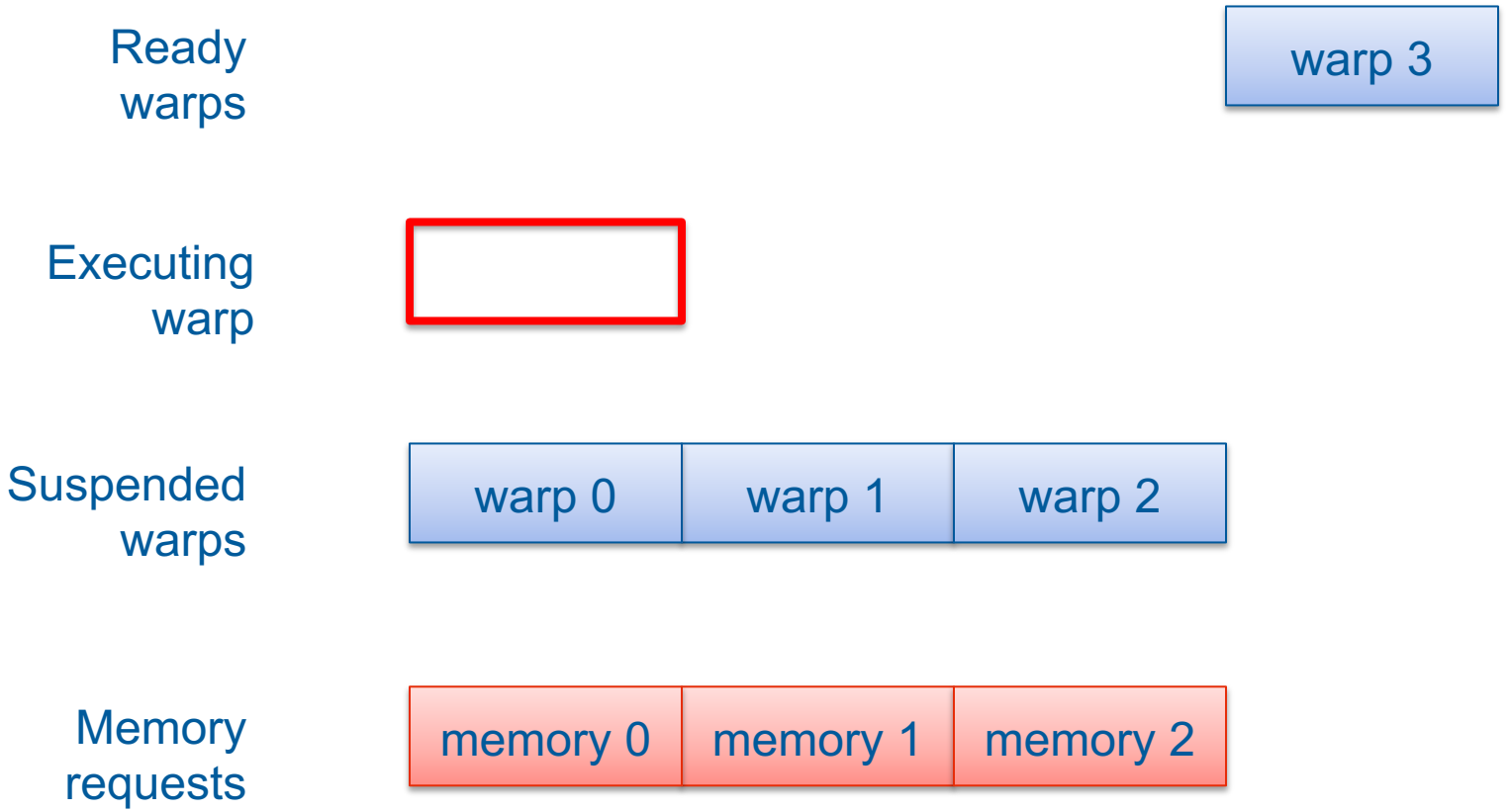
Warp Scheduling



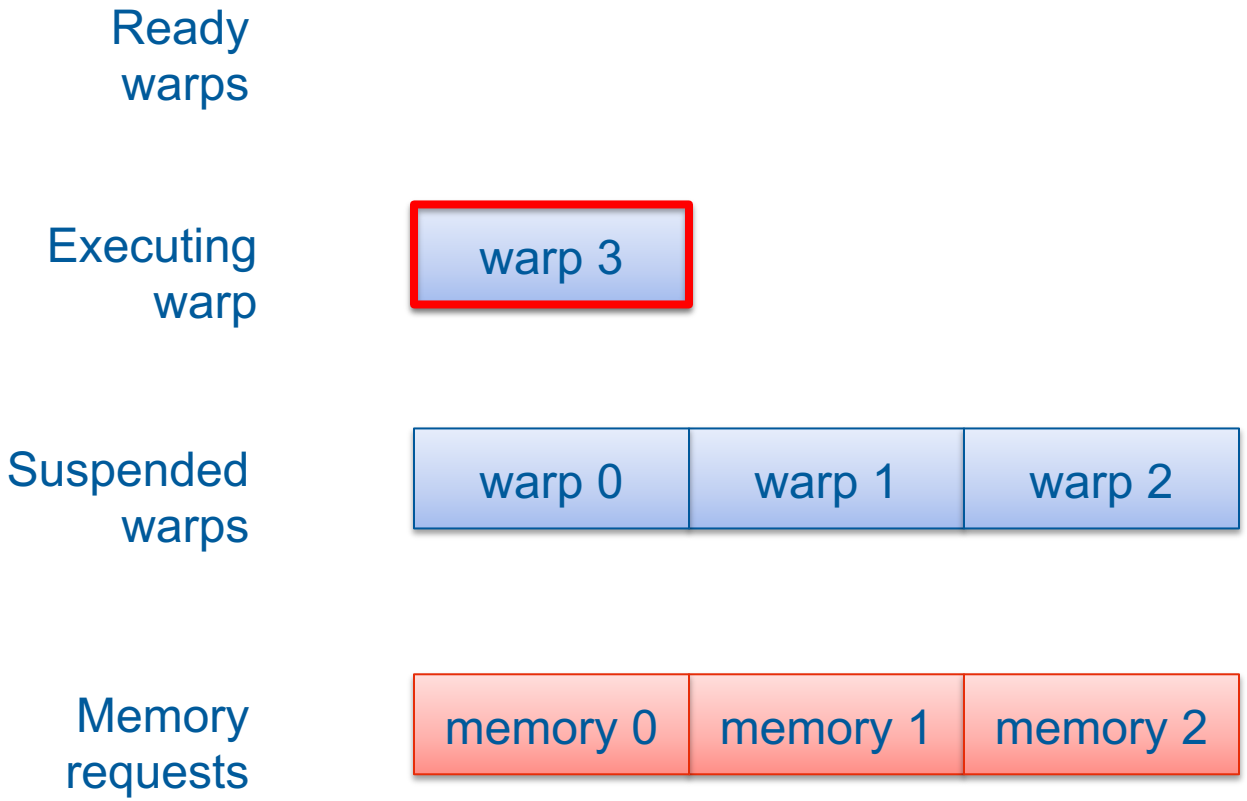
Warp Scheduling



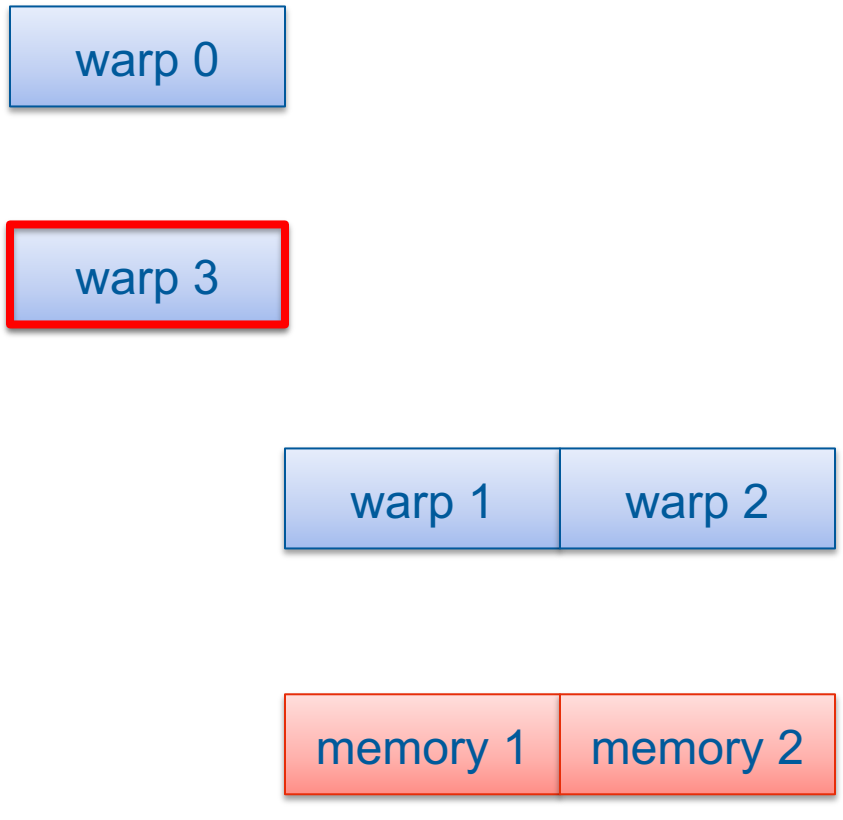
Warp Scheduling



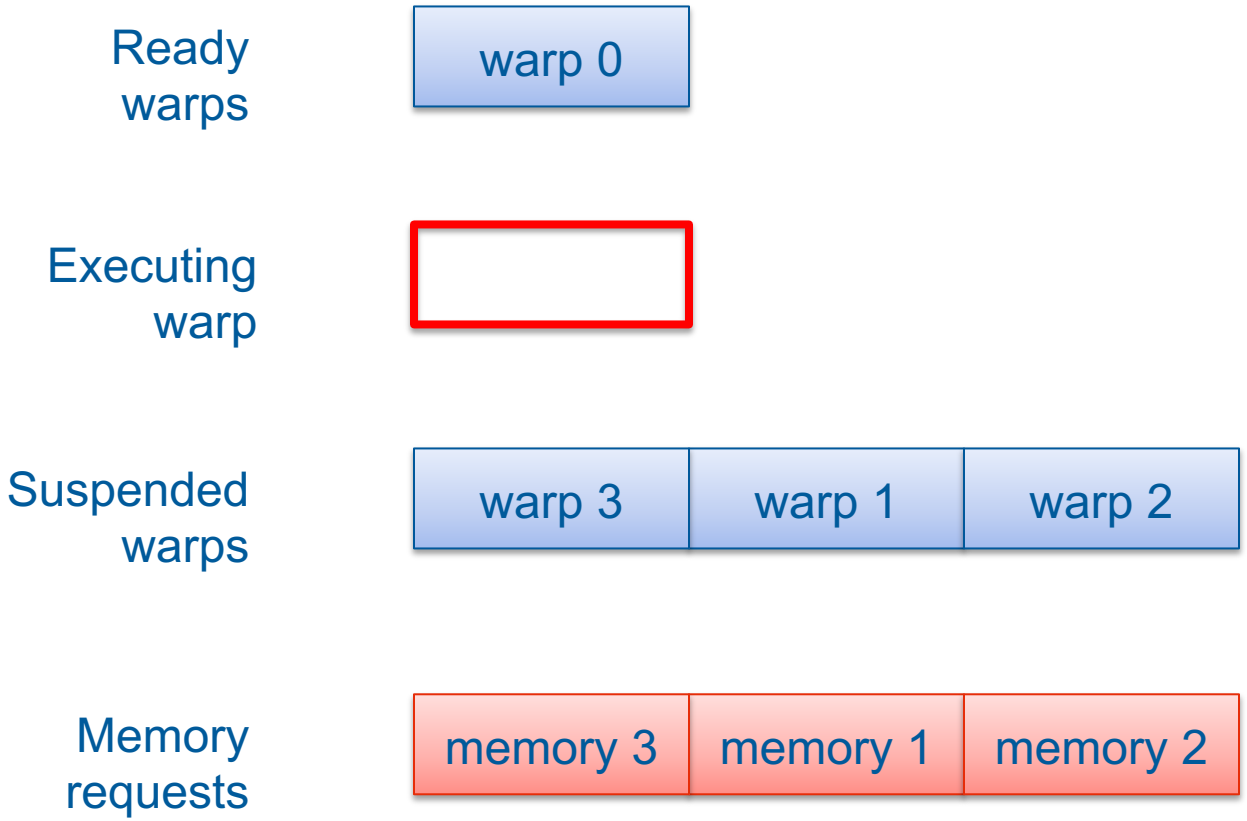
Warp Scheduling



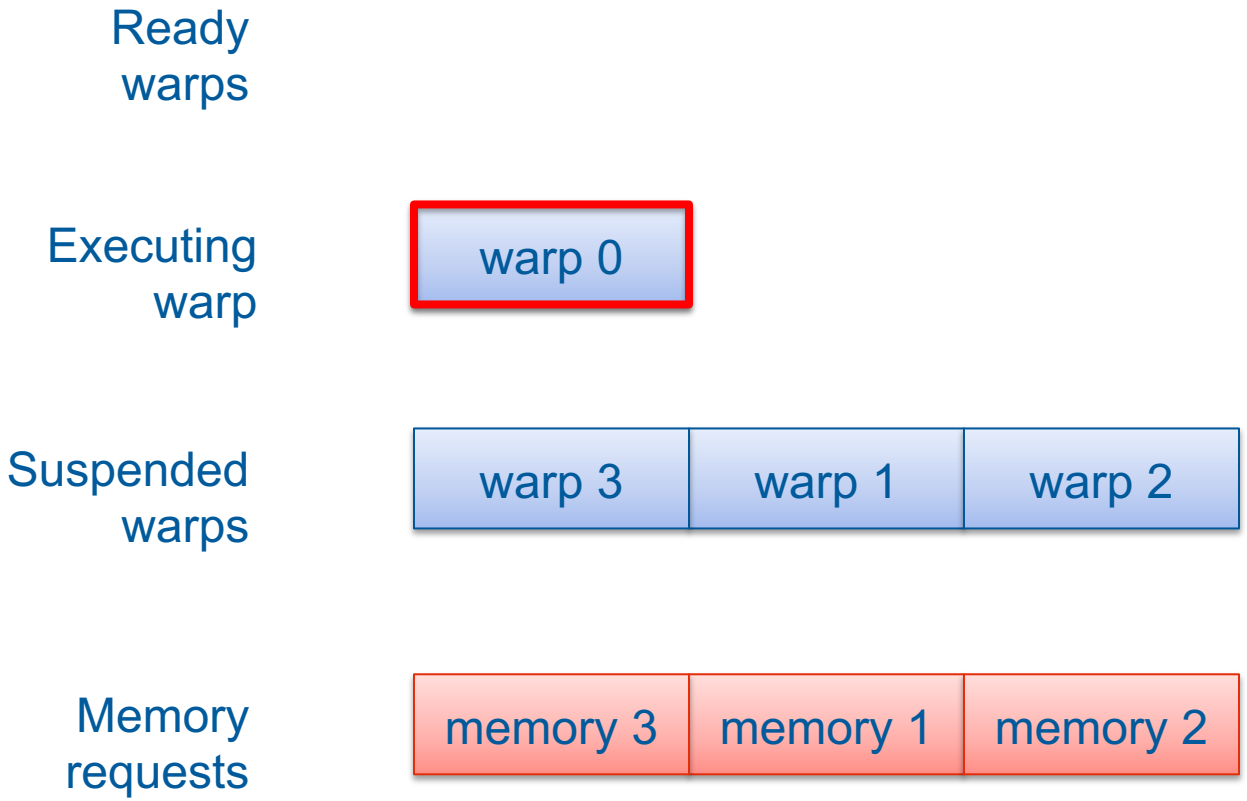
Warp Scheduling



Warp Scheduling



Warp Scheduling



The CUDA Programming Model

Writing Parallel Code

- Current GPUs have 10496 cores (RTX 3090 TITAN V)
- Need more threads than cores (warp scheduler)
- Writing code for 40000 threads / 1200 warps?

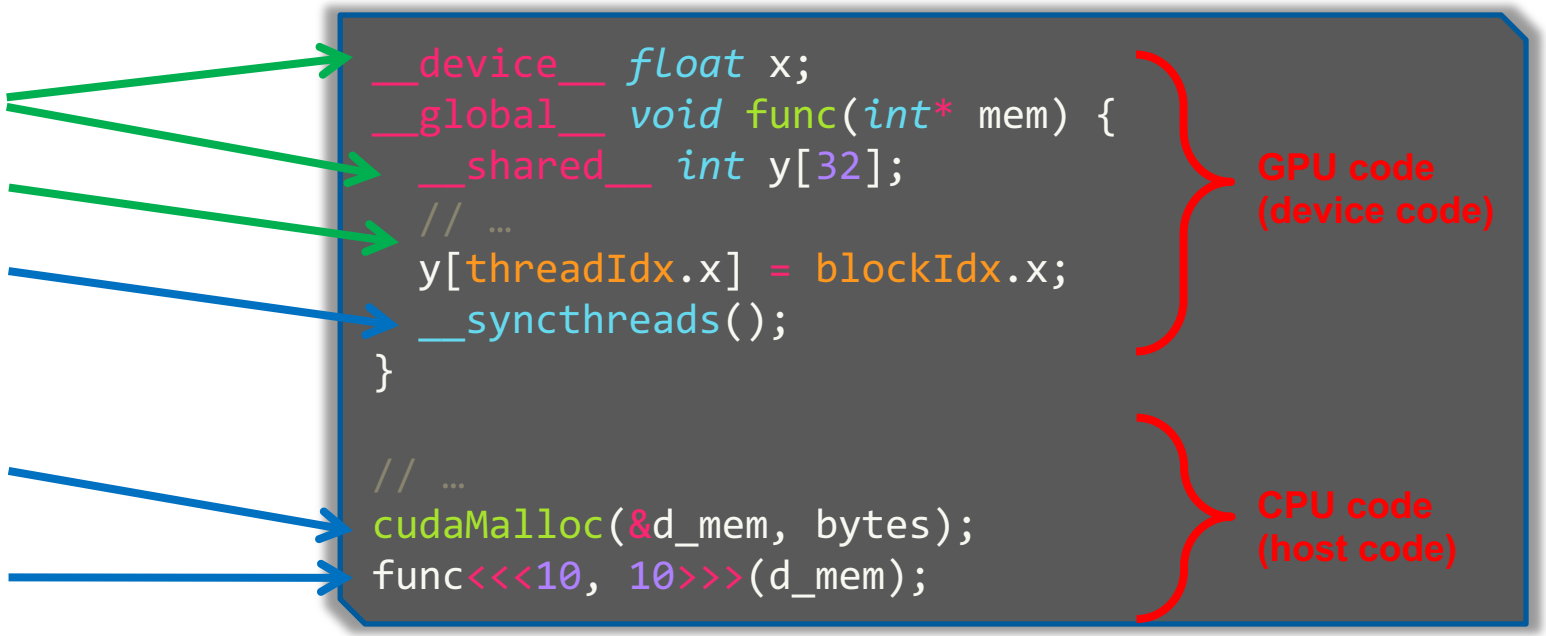
  → Single-program, multiple-data (SPMD) model

  - Write one program that is executed by all threads

- CUDA C++ is C++ with additional keywords to control parallel execution
- Single-source language for host and device code

- Type qualifiers
- Builtins
- Intrinsics

- Runtime API
- GPU kernel launches



- Write **host** (CPU) and **device** (GPU) code into the same file
- `nvcc` compiler driver takes care of
  - Separating host and device code
  - Generating additional host code
    - For example, for kernel launches
  - Sending each part through the respective toolchain

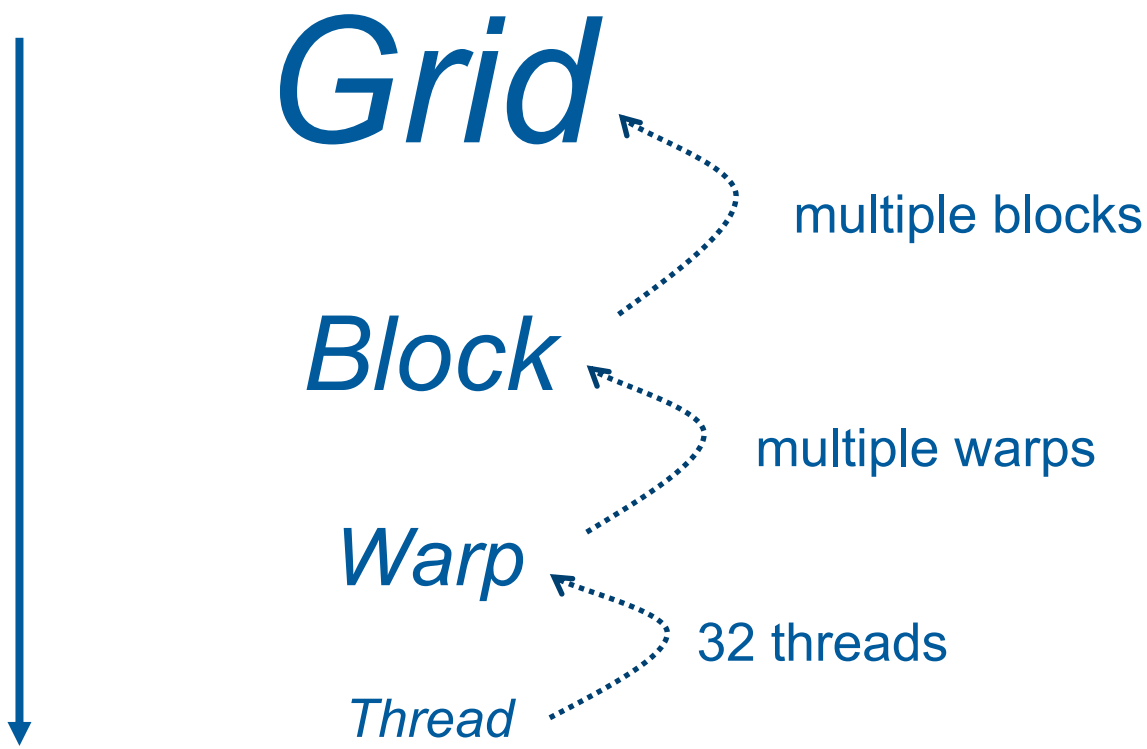
Kernel

- A function that is executed on the GPU
- Each thread is executing the same function

```
__global__ void myfunction(float* input, float* output) {
    *output = *input;
}
```

- Indicated by `__global__`
- Must have return value `void`

Hierarchy


Grid
multiple blocks
Block
multiple warps
Warp
32 threads
Thread

Blocks

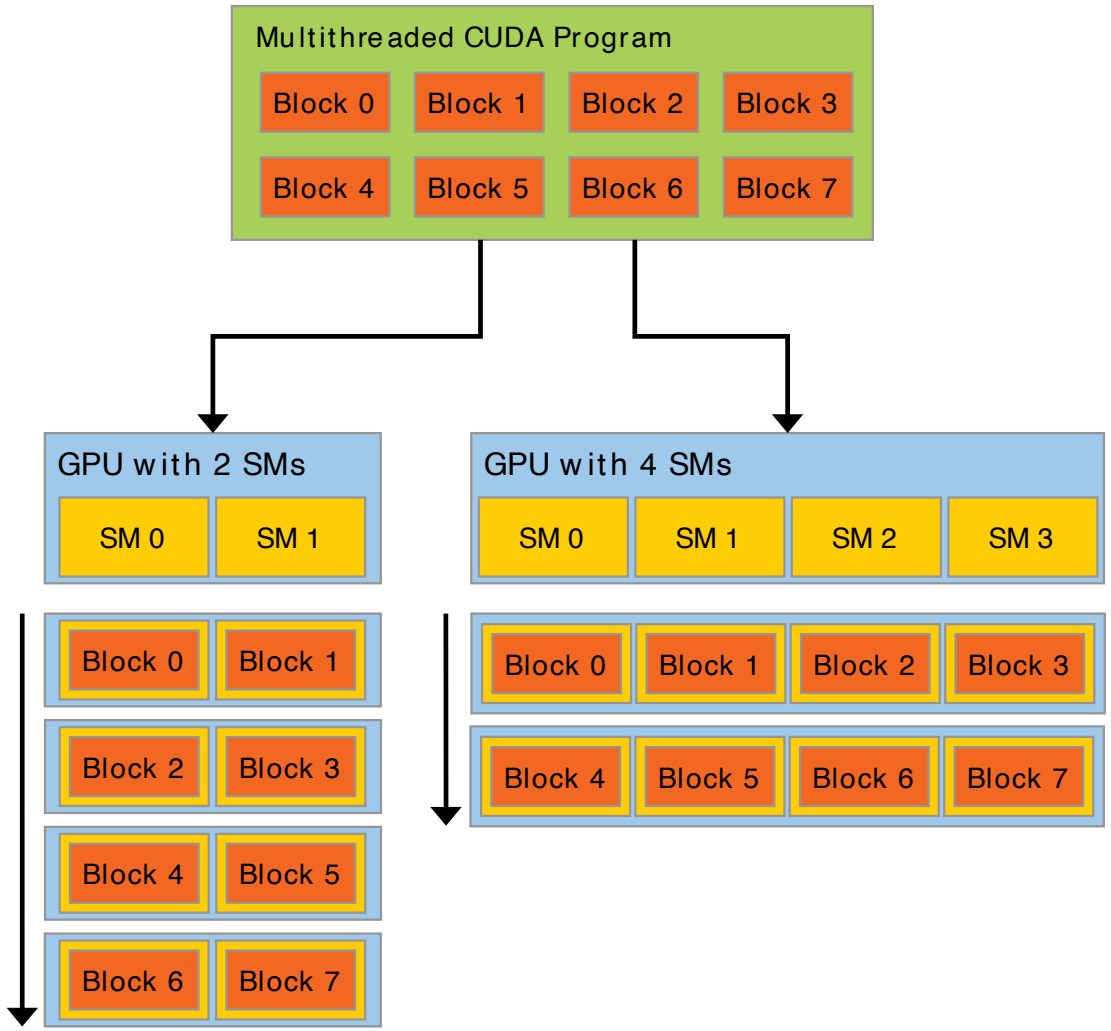
- Threads within one block …
  - Are executed together on the same SM
  - Can synchronize
  - Can communicate efficiently
  - Share the same local cache

  → Can be used to cooperatively compute some result
- Threads of different blocks…
  - May be executed one after another on different SMs
  - Cannot synchronize
  - Can only communicate inefficiently

  → Should work independently of other blocks

Block Scheduling

- Block queue feeds multiprocessors
- Number of available multiprocessors determines number of concurrently executed blocks



Parallel Kernel

- Kernel is split up in blocks of threads

Launching a Kernel

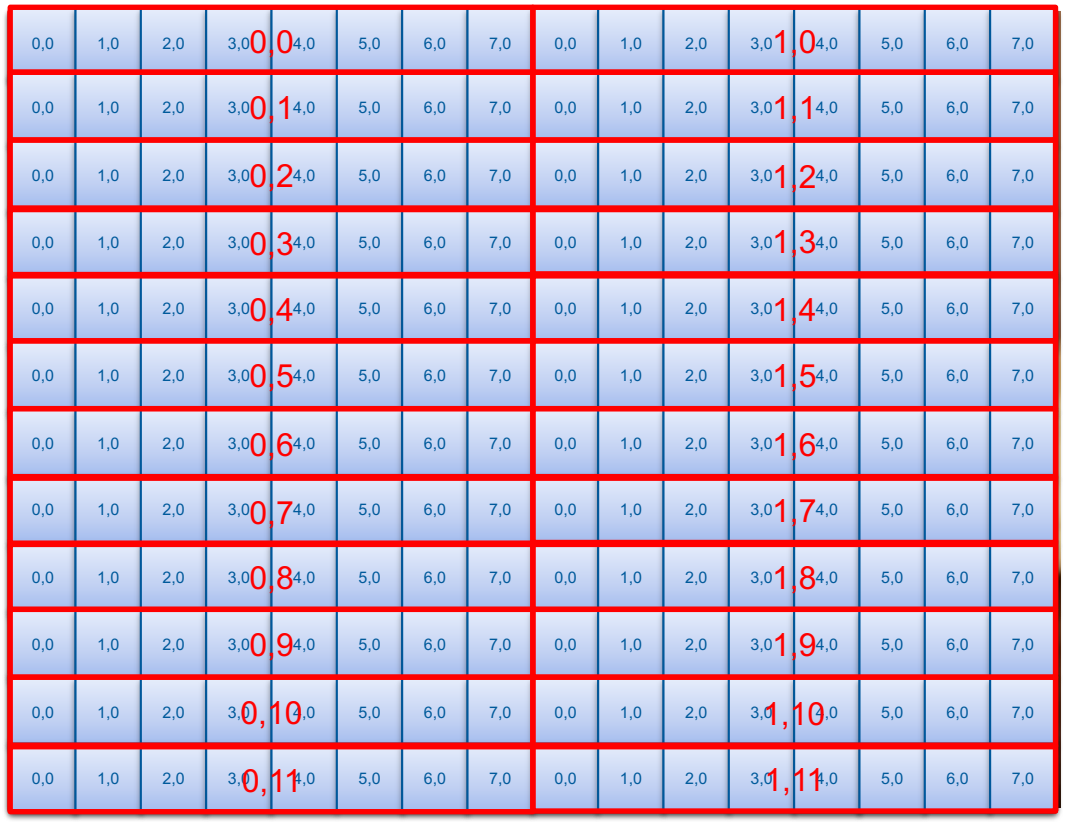
- Requires threads per block, number of blocks
- Launch is asynchronous

```
dim3 block_size(128, 1, 1);
dim3 grid_size(12, 1, 1);
myfunction<<<grid_size, block_size>>>(input, output);
```

- Block and thread setup specified within <<< , >>>
- Input parameters are copied to the GPU

Distinguishing between Threads

- blockIdx and threadIdx



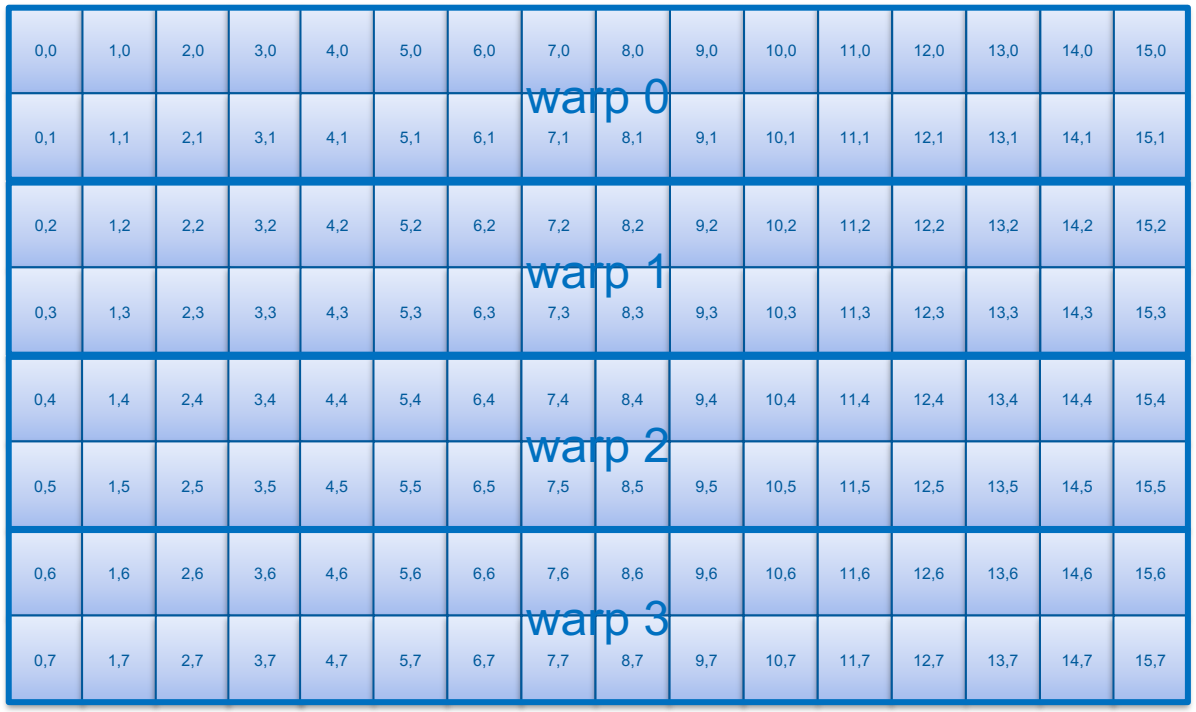
Distinguishing between Threads

- Using **threadIdx** and **blockIdx** execution paths are chosen
- With **blockDim** and **gridDim** number of threads can be determined

```
__global__ void myfunction(float* input, float* output) {
    uint id = threadIdx.x + blockIdx.x * blockDim.x;
    output[id] = input[id];
}
```

Blocks to Warps

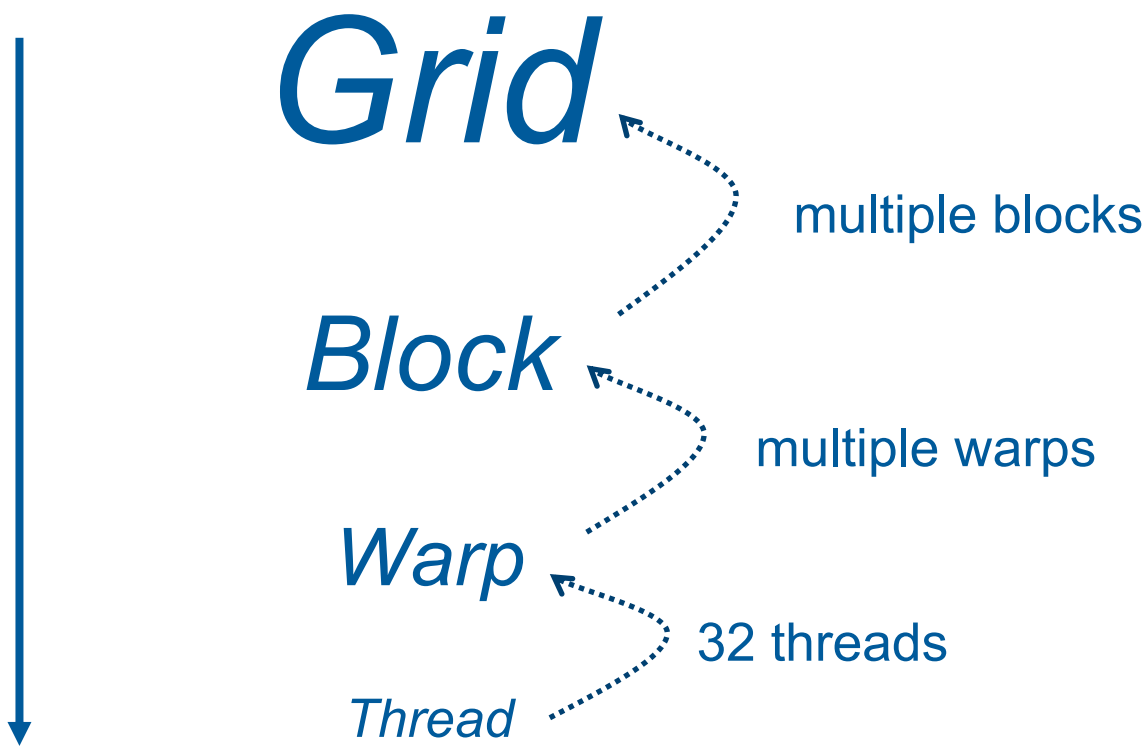
- On each multiprocessor each block is split up into warps
- Threads with the lowest id map to the first warp



Occupancy

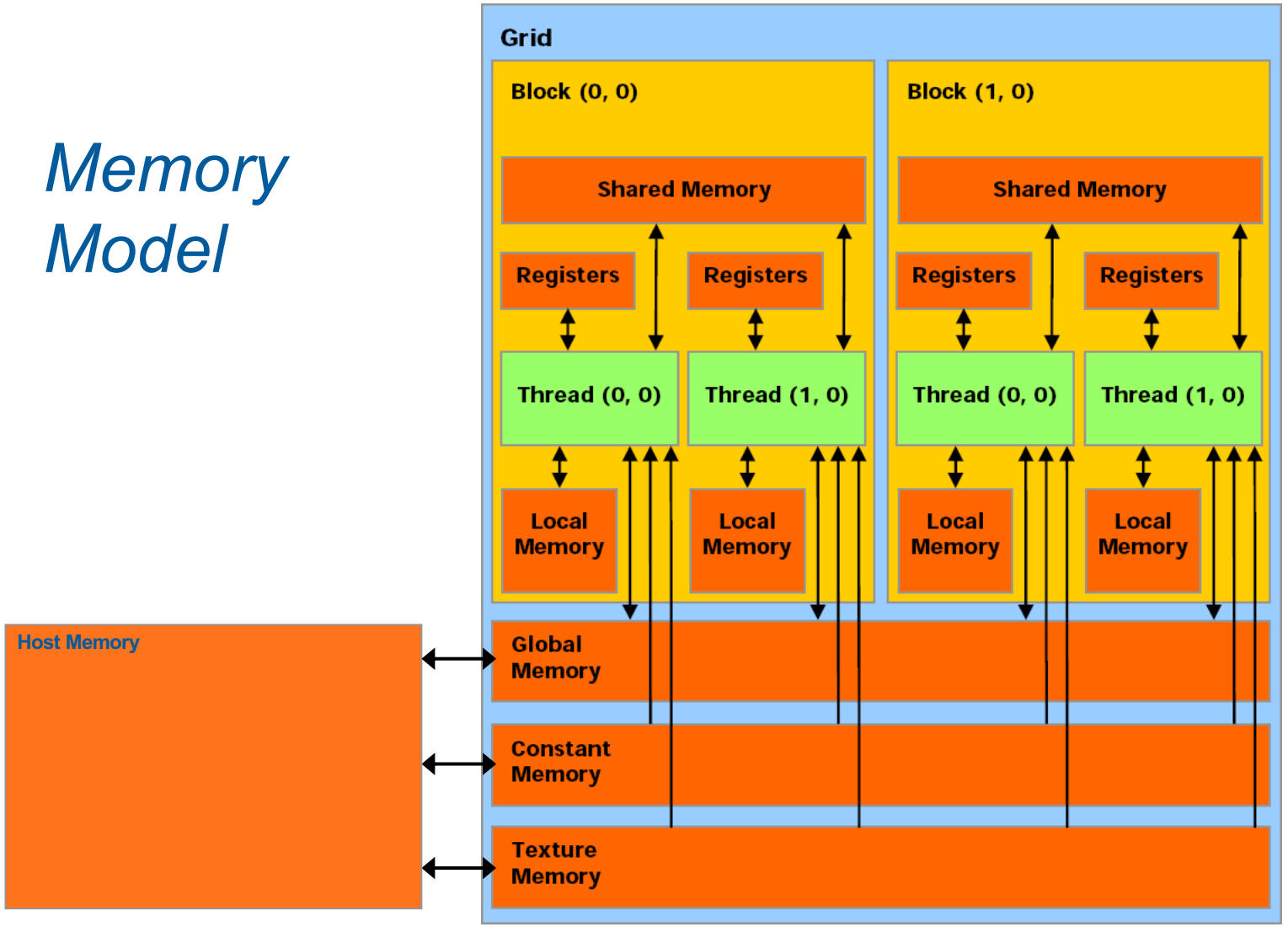
- What block size to choose?
- $occupancy = \frac{active\ warps}{maximum\ active\ warps}$
- Potential limits
  - Register usage
  - Shared memory usage
  - Block size
- Start with 128 / 256 threads
  - Maximum size 1024 threads

Hierarchy - Recap



Memory Spaces

Memory Model



Registers

- Private for each thread
- Fastest memory
- Automatically allocated from per SM register file (number of registers per block)
- No keyword

Global Memory

- Main GPU memory
- Accessible from all blocks/threads
- Access pattern matters
  - Older devices coalescing!
  - Newer devices: close together
- Slowest on device memory

  (up to 1000x times slower than registers)
- Cached on newer devices
- `__device__` / **`cudaMalloc`**

Texture Memory

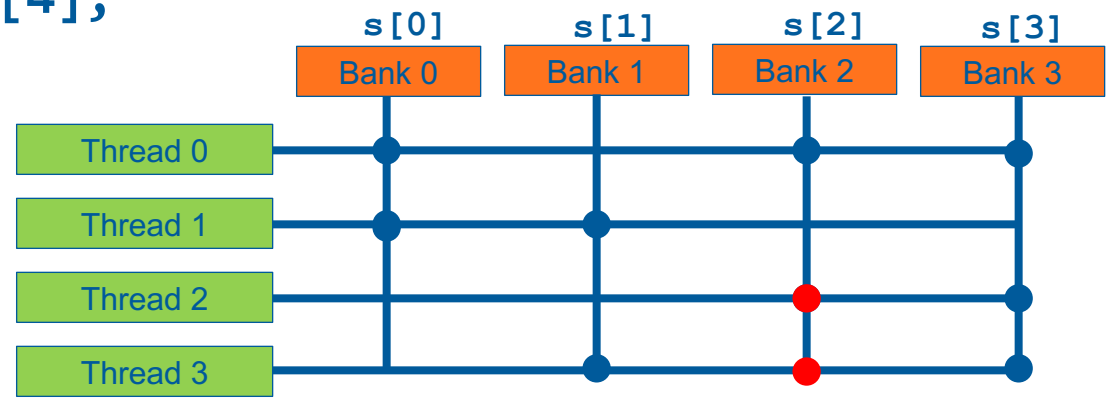
- Accessible from all blocks/threads
- Cached on all devices
  - Special layout for 2D and 3D textures
- Hardware interpolation (same as OpenGL)
- Automatic handling of boundary cases
- Normalized texture coordinates
- Conversion to float
- No concurrent read and write
- **`cudaMallocArray + cudaBindTextureToArray`**

Constant Memory

- Read-only
- Ideal for coefficients and other data that is read uniformly by warps
- Cached on all devices
- `__constant__` + `cudaMemcpyToSymbol`

Shared Memory

- Shared access within one block (lifetime: block)
- Located on multiprocessor → very fast
- Limited available size on multiprocessor
- Crossbar: simultaneous access to distinct banks
- `__shared__ int s[4];`



Local Memory

- If above register limited, registered are spilled to global
- Very slow in comparison to registers
- Besides that, same behavior as registers
- Always perfect access pattern

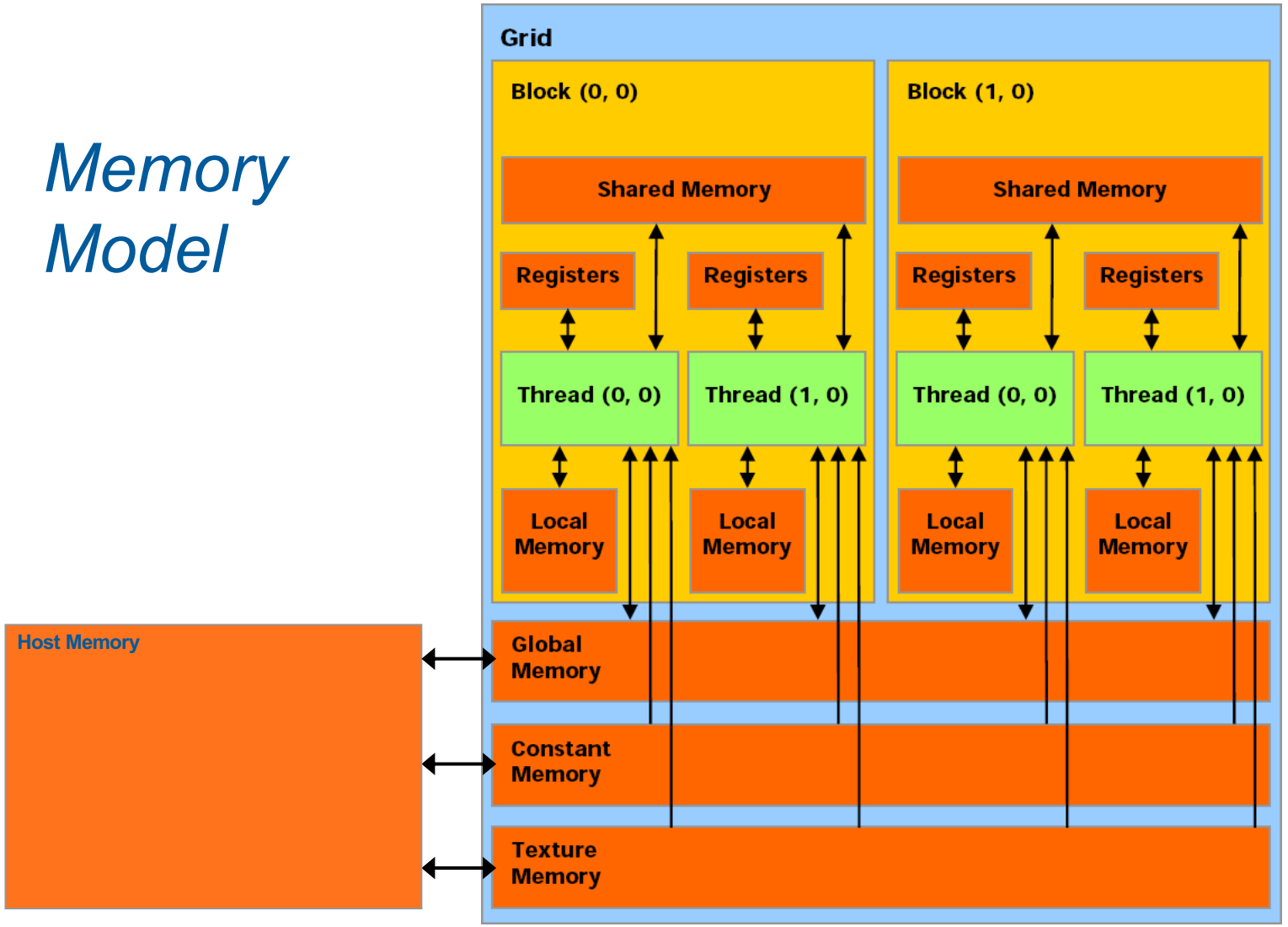
Host Memory

- Can be mapped to device memory space
- Same data can be accessed from host and device concurrently
- Even slower than global memory
- Mapped memory:

  `cudaHostAlloc` with **`cudaHostAllocMapped`**

Memory Model - Recap

Memory Model



Memory Allocation

	Global	Constant	Texture	Shared	Local	Registers
Host Code	Dynamic allocation R/W	No allocation R/W	Dynamic allocation R/W	Dynamic allocation No access	No allocation No access	No allocation No access
Device Code	Static allocation (Dynamic allocation) R/W	Static allocation Read-only	Static allocation Read-only	Static allocation R/W	Static allocation R/W	Static allocation R/W