Parallel Programming

# Parallel Execution in CUDA

Phạm Trọng Nghĩa
ptnghia@fit.hcmus.edu.vn

# Review of previous lecture

## CUDA allows us to organize grid and block as 1D or 2D or 3D

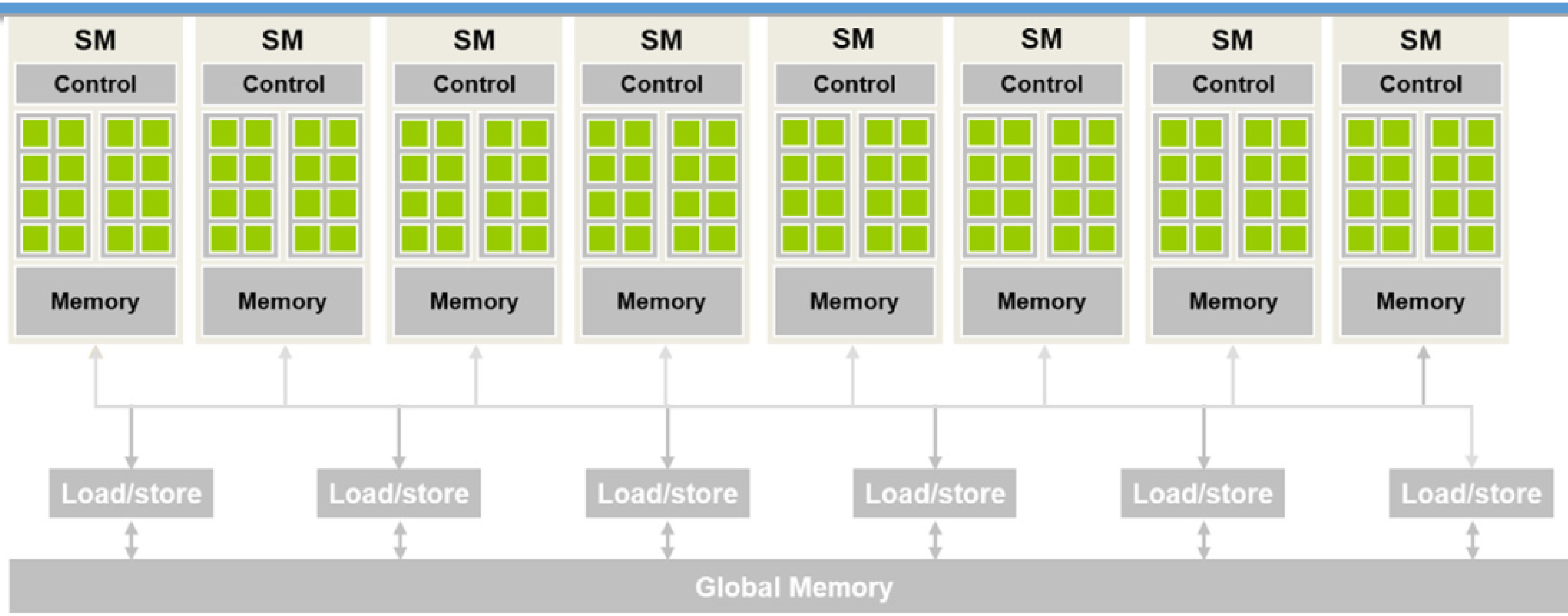
| | Compute Capability | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **Technical Specifications** | 5 | 5.2 | 5 | 6 | 6.1 | 6.2 | 7 | 7.2 | 7.5 | 8 | 8.6 | 8.7 | 8.9 | 9 |
| Maximum number of resident grids per device (Concurrent Kernel Execution) | 32 | | 16 | 128 | 32 | 16 | 128 | 16 | 128 | | | | | |
| Maximum dimensionality of grid of thread blocks | 3 | | | | | | | | | | | | | |
| Maximum x -dimension of a grid of thread blocks | $2^{31}$-1 | | | | | | | | | | | | | |
| Maximum y- or z-dimension of a grid of thread blocks | 65535 | | | | | | | | | | | | | |
| Maximum dimensionality of thread block | 3 | | | | | | | | | | | | | |
| Maximum x- or y-dimensionality of a block | 1024 | | | | | | | | | | | | | |
| Maximum z-dimension of a block | 64 | | | | | | | | | | | | | |
| Maximum number of threads per block | 1024 | | | | | | | | | | | | | |
| Warp size | 32 | | | | | | | | | | | | | |
| Maximum number of resident blocks per SM | 32 | | | | | | | | 16 | 32 | 16 | | 24 | 32 |
| Maximum number of resident warps per SM | 64 | | | | | | | | 32 | 64 | 48 | | | 64 |
| Maximum number of resident threads per SM | 2048 | | | | | | | | 1024 | 2048 | 1536 | | | 2048 |
| Number of 32-bit registers per SM | 64 K | | | | | | | | | | | | | |
| Maximum number of 32-bit registers per thread block | 64 K | | | | | | | | | | | | | |
| Maximum number of 32-bit registers per thread | 255 | | | | | | | | | | | | | |
| Maximum amount of shared memory per SM | 64 KB | 96 KB | 64 KB | | 96 KB | 64 KB | 96 KB | | 64 KB | 164 KB | 100 KB | 164 KB | 100 KB | 228 KB |
| Maximum amount of shared memory per thread block | 48 KB | | | | | | 96 KB | 96 KB | 64 KB | 163 KB | 99 KB | 163 KB | 99 KB | 227 KB |
| Number of shared memory banks | 32 | | | | | | | | | | | | | |
| Maximum amount of local memory per thread | 512 KB | | | | | | | | | | | | | |
| Constant memory size | 64 KB | | | | | | | | | | | | | |
| Cache working set per SM for constant memory | 8 KB | | | 4 KB | 8 KB | | | | | | | | | |

# Today: level 2 CUDA

Aspects of the **GPU compute architecture** that are essential for CUDA C programmers.

- A high-level, simplified view of the **compute architecture** and explore the concepts of flexible *resource assignment*, *scheduling of blocks*, and *occupancy*.
- *Thread scheduling*, *latency tolerance*, *control divergence*, and *synchronization*.
- **API functions** that can be used to query the resources that are available in the GPU and the tools to help estimate the occupancy of the GPU when executing a kernel.

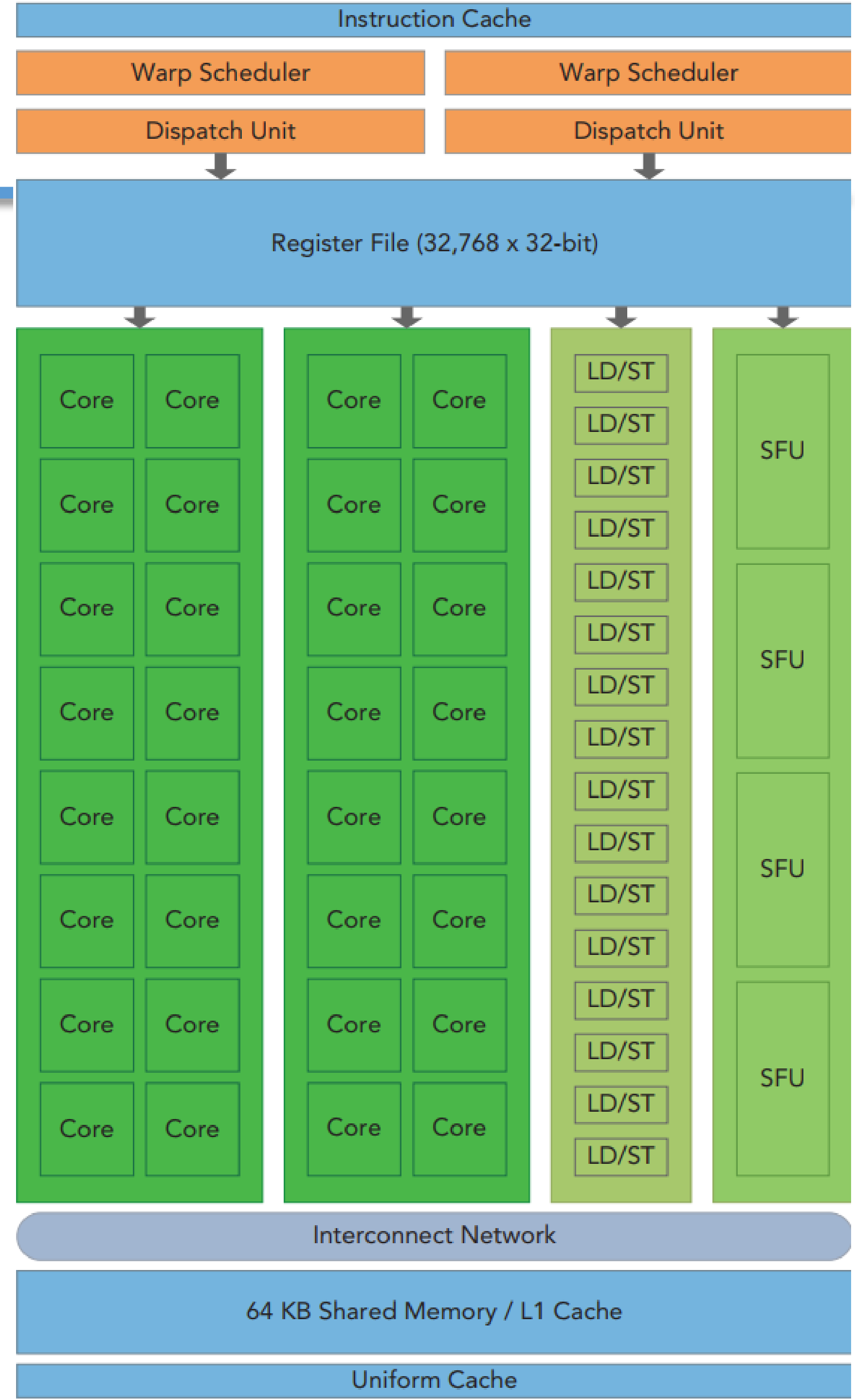
# Architecture of a modern GPU




- GPU consists of SMs – Streaming Multiprocessors
  - Each SM consists of SPs – Streaming Processors (or CUDA cores)
  - The SMs have special on-chip memory
- E.g.: the Ampere A100 GPU has 108 SMs with 64 cores each, totaling 6912 cores in the entire GPU

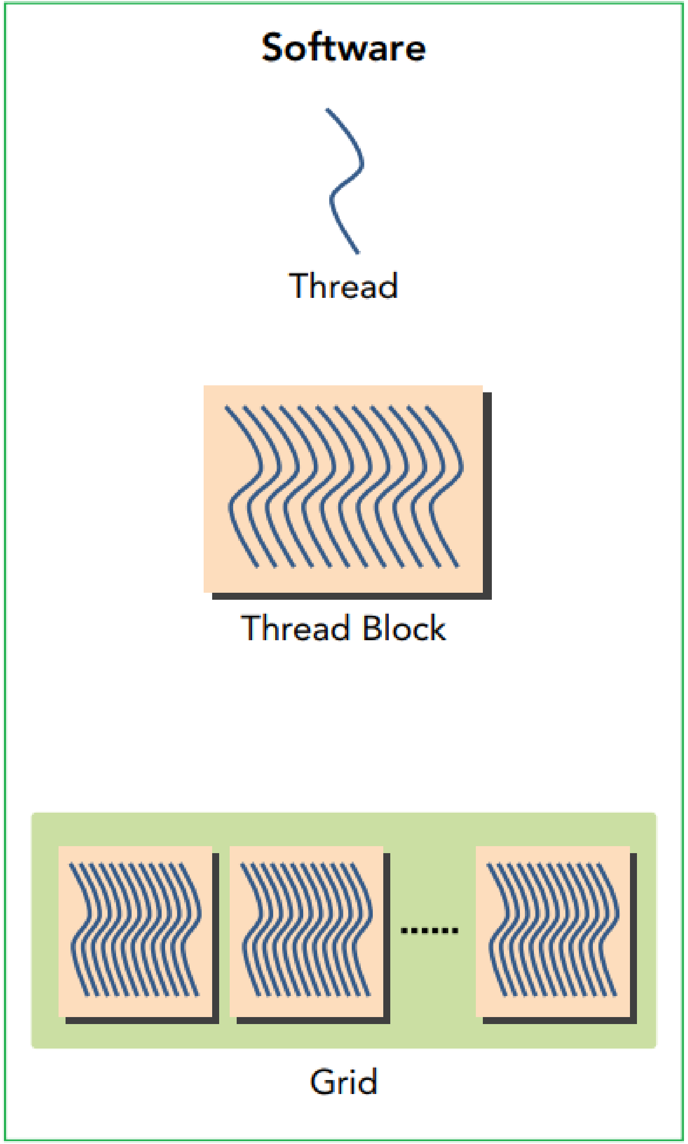
# More about SM

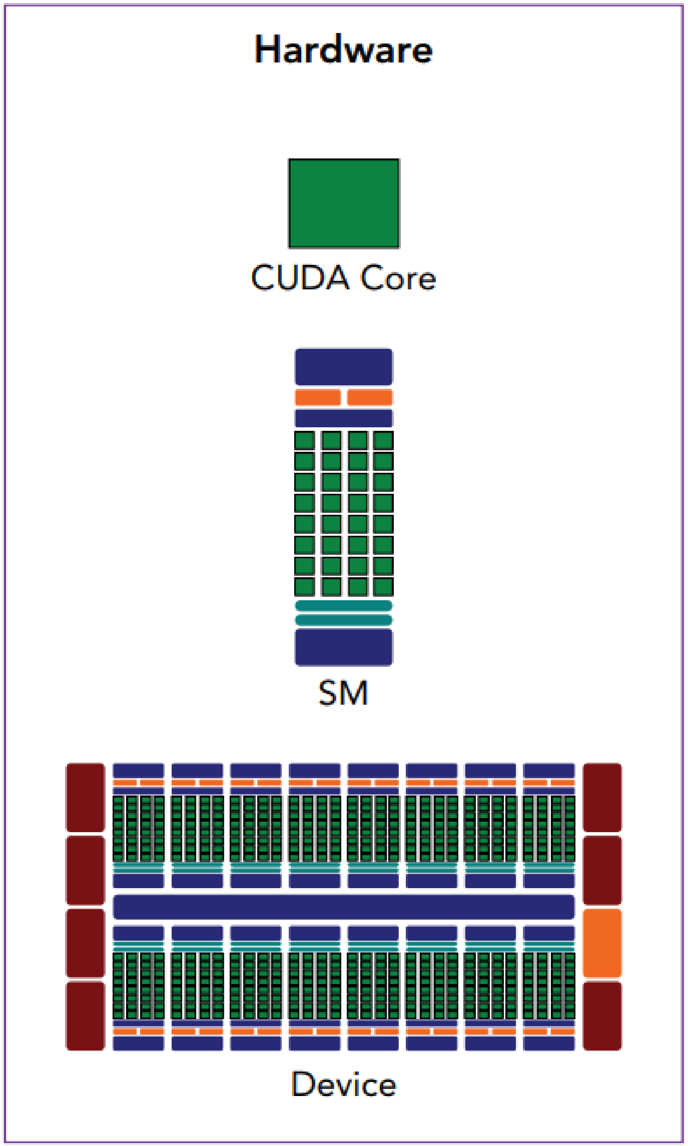
- CUDA Cores
- Shared Memory/L1 Cache
- Register File
- Load/Store Units
- Special Function Units
- Warp Scheduler




Fermi SM
Source: Professional CUDA C programming

# Architecture of a modern GPU



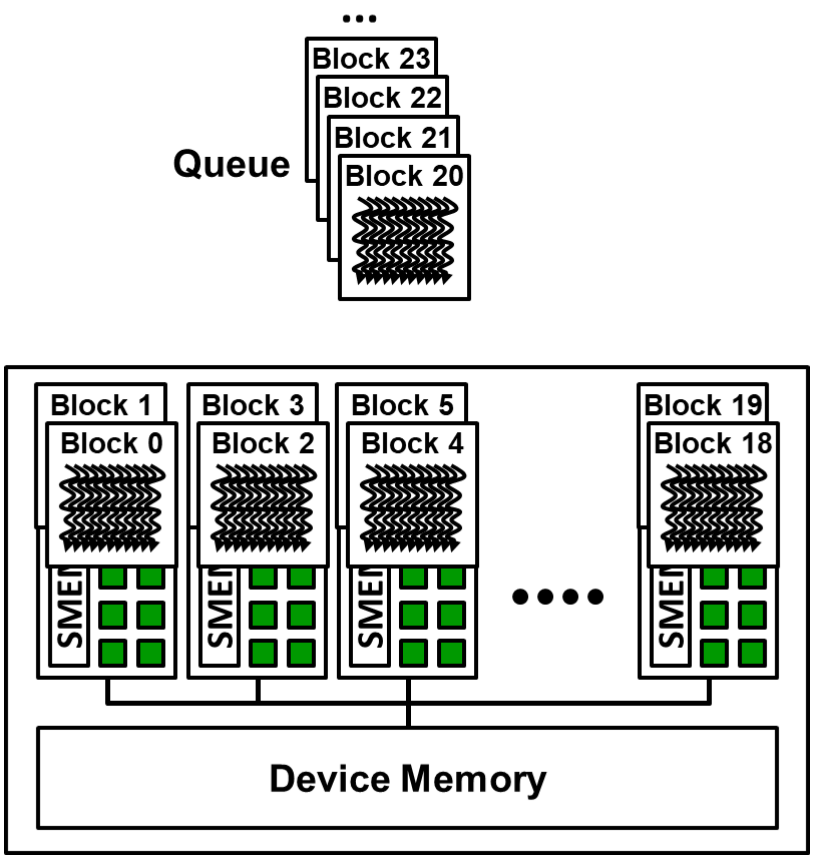
# Parallel execution – SM level

- CUDA virtualizes GPU hardware architecture
  - Block = virtual SM
  - Thread = virtual SP
- When host calls a kernel function, a grid of blocks will be created and each block (virtual SM) will be assigned to a real SM for execution
  - Each SM can contain > 1 block to execute
    - It depends on SM resource limitations and resources each block needs
    - E.g., SM needs resources (registers) to keep track of indexes of blocks and threads as well as their execution state, SM 2.x resources can afford at most 8 blocks and 1536 threads

      If block size is 512 → SM 2.x can contain 3 blocks

# Parallel execution – SM level

- CUDA virtualizes GPU hardware architecture
  - Block = virtual SM
  - Thread = virtual SP
- When host calls a kernel function, a grid of blocks will be created and each block (virtual SM) will be assigned to a real SM for execution
  - Each SM can contain > 1 block to execute
  - Blocks which have not been assigned to SMs will wait in a queue
  - When a block finishes its execution, a block from the queue will be assigned to the available slot in SM




☐ *Note: blocks can be assigned to SMs in an arbitrary order*

# Parallel execution – SM level

Such parallel execution in CUDA:

- Helps achieve scalability ☺

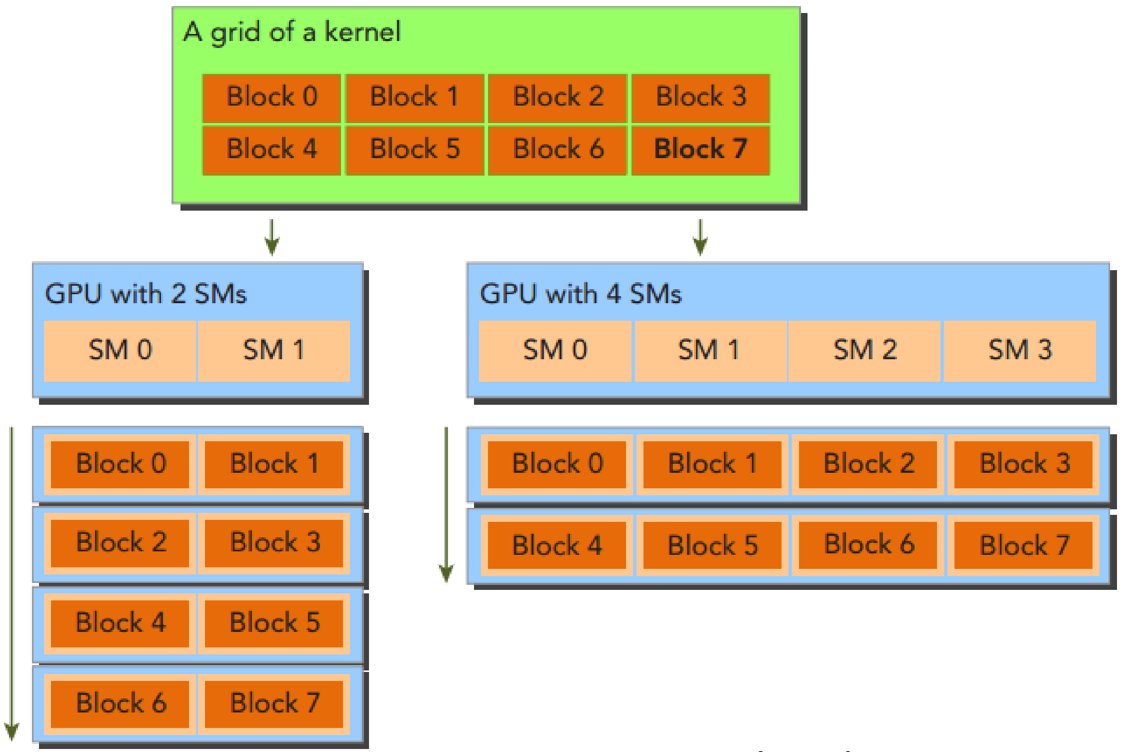



Image source: John Cheng et al. Professional CUDA C Programming. 2014

# Parallel execution – SM level

Such parallel execution in CUDA:

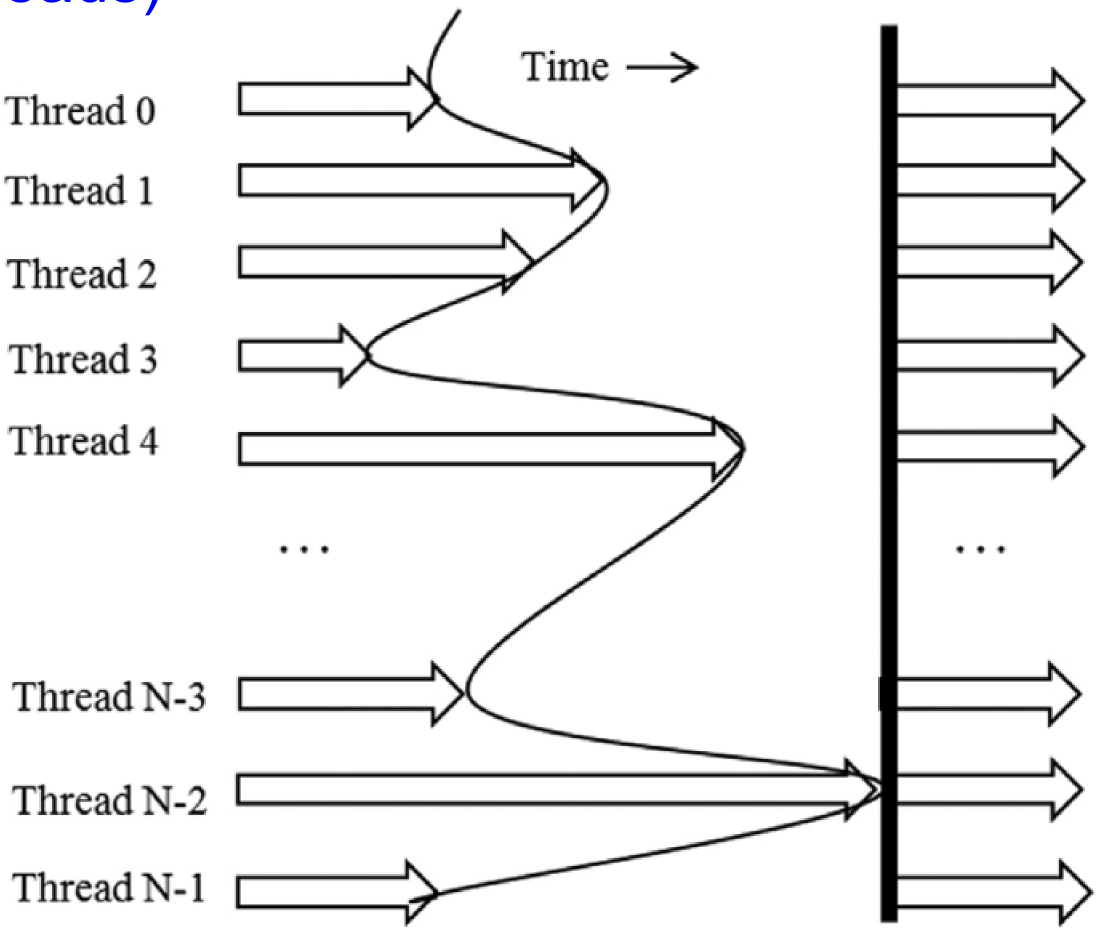
- Helps achieve scalability ☺
- But requires blocks to be independent of each other → threads in <u>different</u> blocks cannot cooperate (synchronize) with each other ☹
  - Assume thread a in block A wants to use a result from thread b in block B,

    and GPU resources can only afford executing one block at a time and currently block A is being executed

    But block B only can be executed when block A is done ☹
  - Threads in the same block can cooperate with each other by using `__syncthreads()` CUDA command

# Parallel execution – SM level

An example execution of barrier synchronization (__syncthreads)



# Parallel execution – SM level

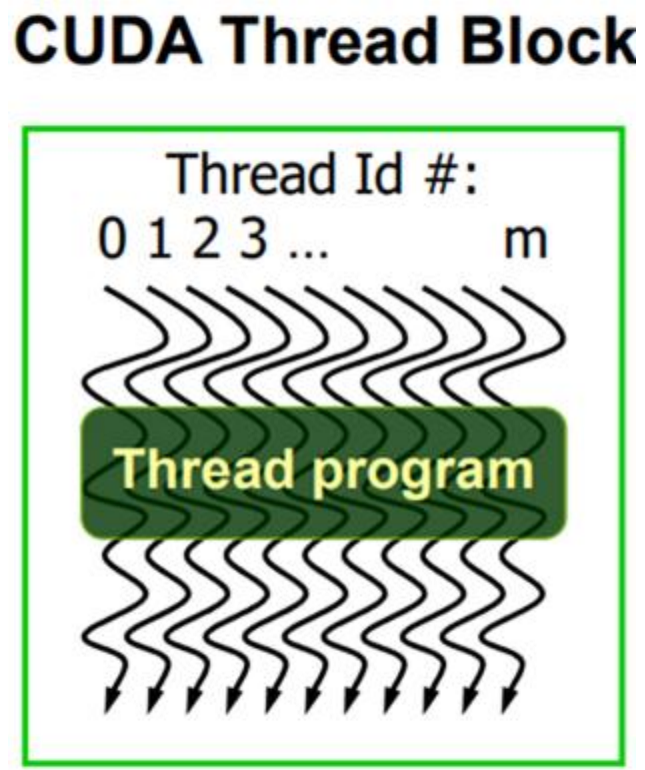
__syncthreads() it must be executed by all threads in a block

```
__global__ void incorrect_barrier_example(int n){
    //...
    if (threadIdx.x % 2 == 0){
        //...
        __syncthreads();
    }
    else{
        //...
        __syncthreads();
    }
}
```

- Not all threads in a block are guaranteed to execute either of the barriers ➔ undefined execution behavior

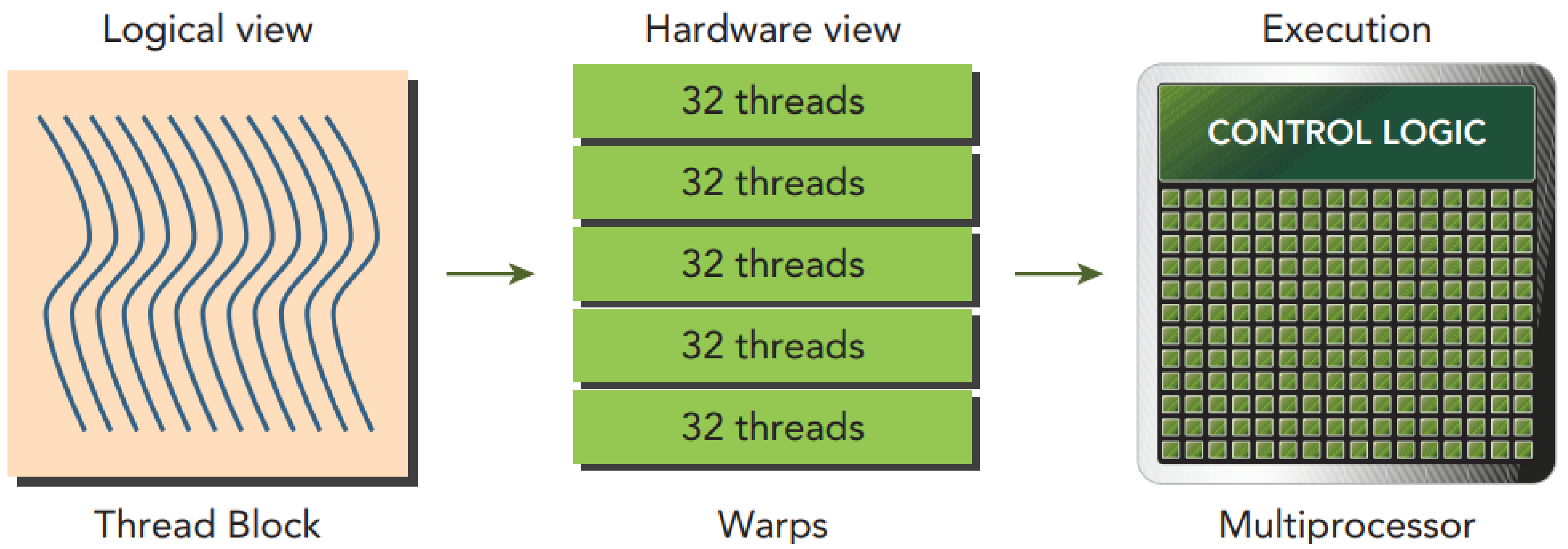
# CUDA Thread Blocks

- All threads in a block execute the same kernel program (**SPMD**)
- Programmer declares block:
  - Block size 1 to 1024 concurrent threads
  - Block shape 1D, 2D, or 3D
- Threads within block have **thread index** numbers
- Kernel code uses **thread index** and **block index** to select work and address shared data
- Threads in the same block **share data** and **synchronize** while doing their share of the work
- Threads in different blocks ***cannot cooperate***
- Blocks **execute in arbitrary order**!



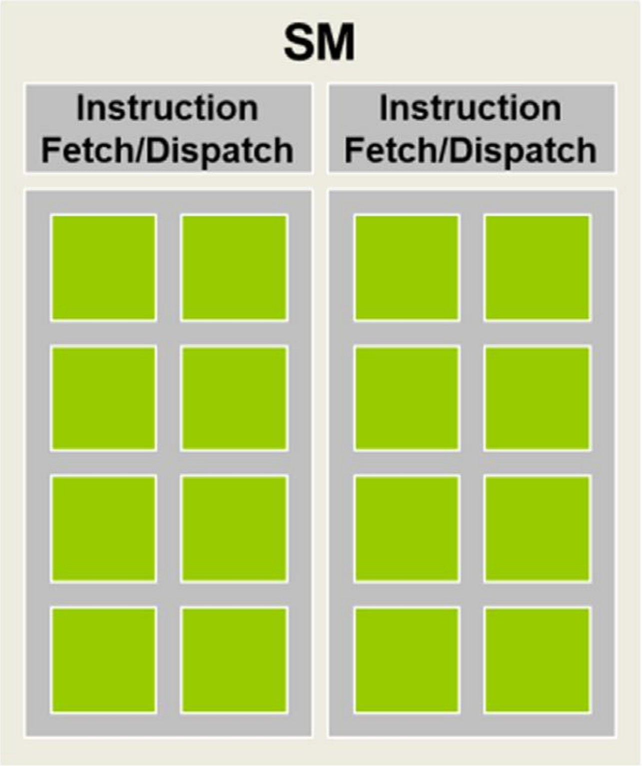
# Parallel execution – Warp

- In SM, with each block, system don't manage and execute each individual thread but a group of 32 threads - called a warp



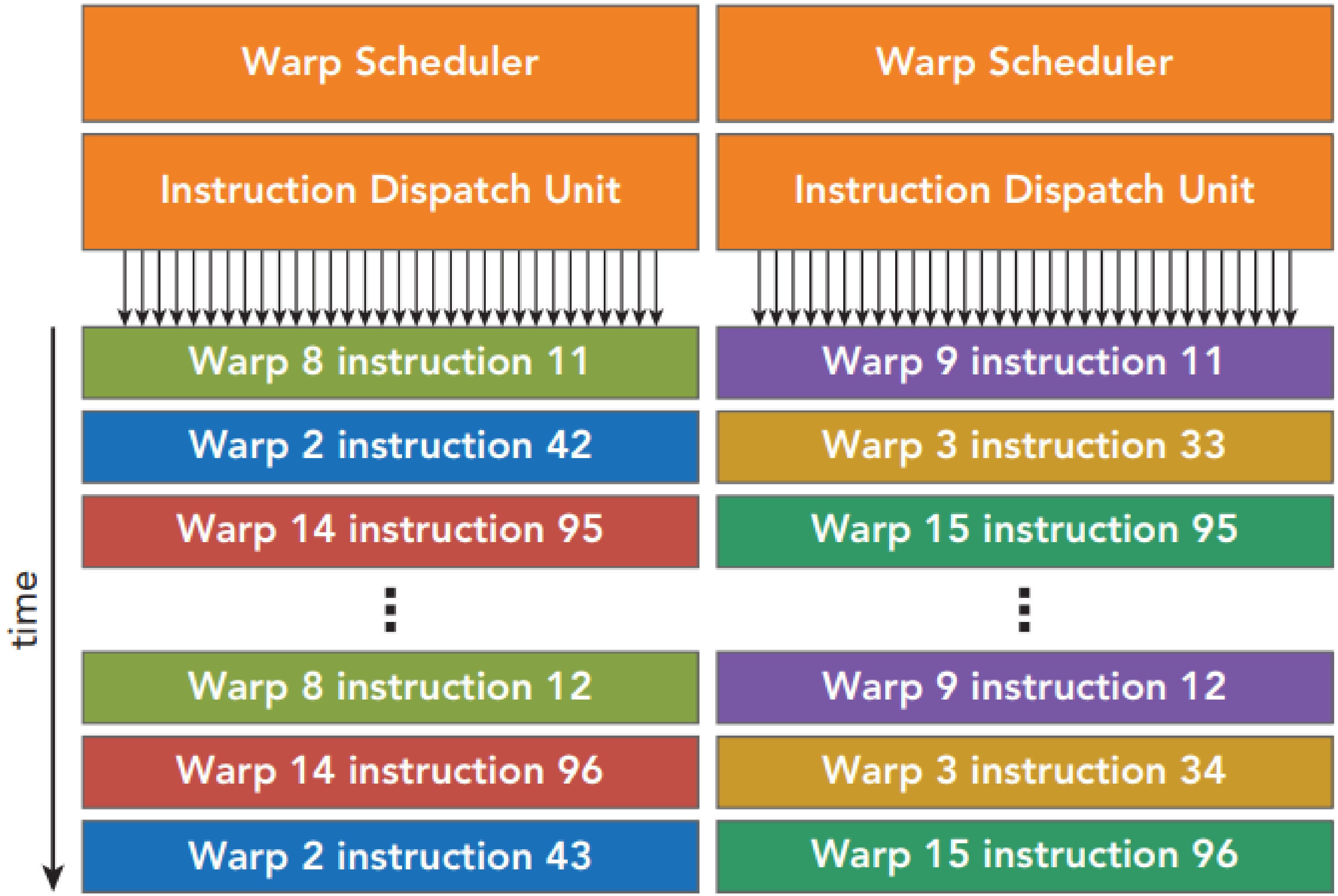
# Parallel execution – Warp

- One instruction is executed for all threads in a warp (each thread has its own data
  - This execution model is called SIMT (Single Instruction Multiple Threads)
- The benefit of this execution model?
  - Help simplify hardware: less resources for control and more resources for arithmetic throughput




***All threads in a warp execute the same instruction when selected***

# Parallel execution – Warp



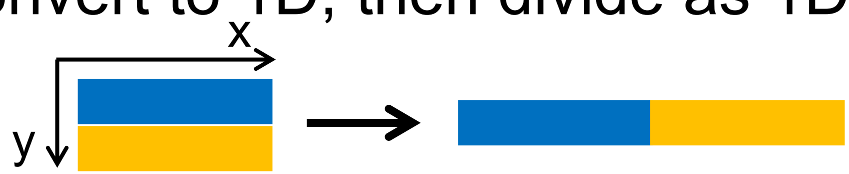
# Parallel execution – Warp

**How is a block in SM divided into warps?**

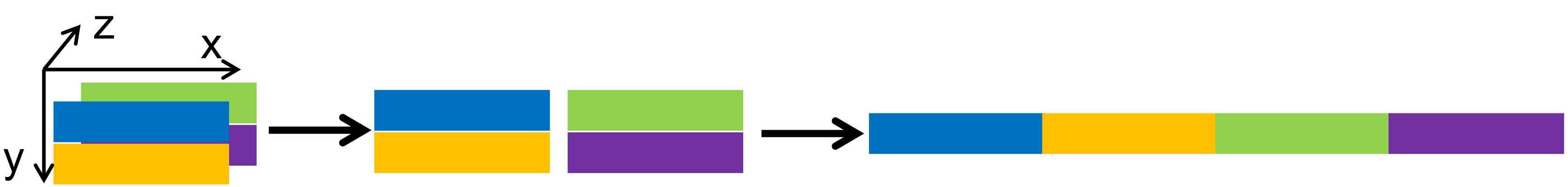
- **Block 1D:** 32 consecutive threads form 1 warp (warp 1: threads 0-31, warp 2: threads 32-63, …)

  If block size is not a multiple of 32 then the last warp will be padded extra threads so that its size is 32, these threads are useless but still consume resources

- **Block 2D:** convert to 1D, then divide as 1D




- **Block 3D:** convert to 1D, then divide as 1D



# Parallel execution – SM-inside level

- What if threads in a warp cannot execute the same instruction?

➔ Warp divergence

  - Correctness? OK
  - Speed? Hmm…

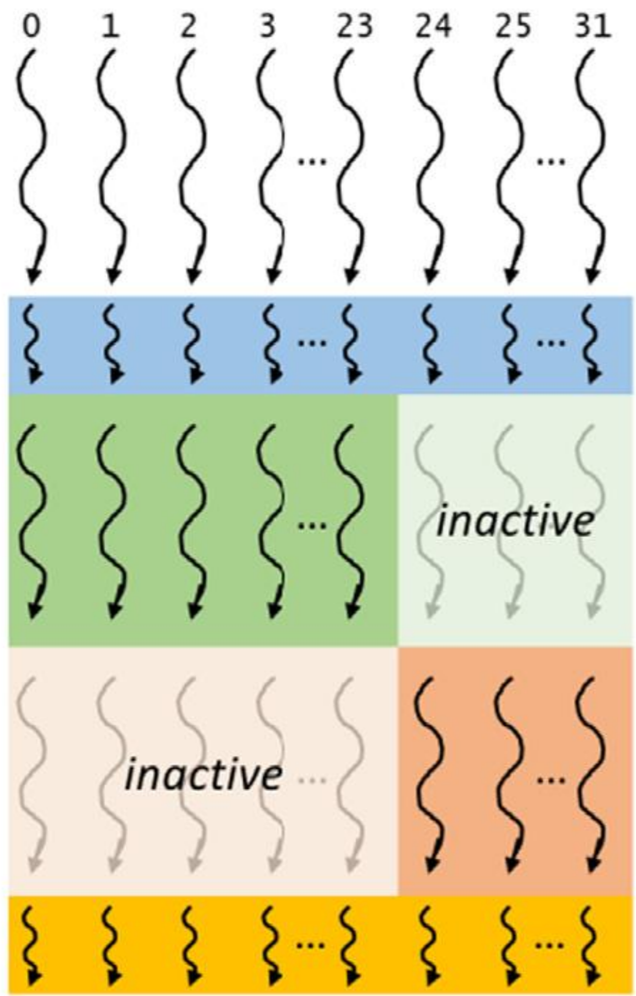
- GPUs use **predicated execution**
  - Each thread computes a yes/no answer for each path
  - Multiple paths taken by threads in a warp are executed serially!
- If all threads in a warp need a barrier synchronization mechanism: __syncwarp()

# Parallel execution – SM-inside level

Warp divergence example: branching

**ALL THREADS EXECUTE BOTH PATHS**
(results kept only when predicate is true for thread)

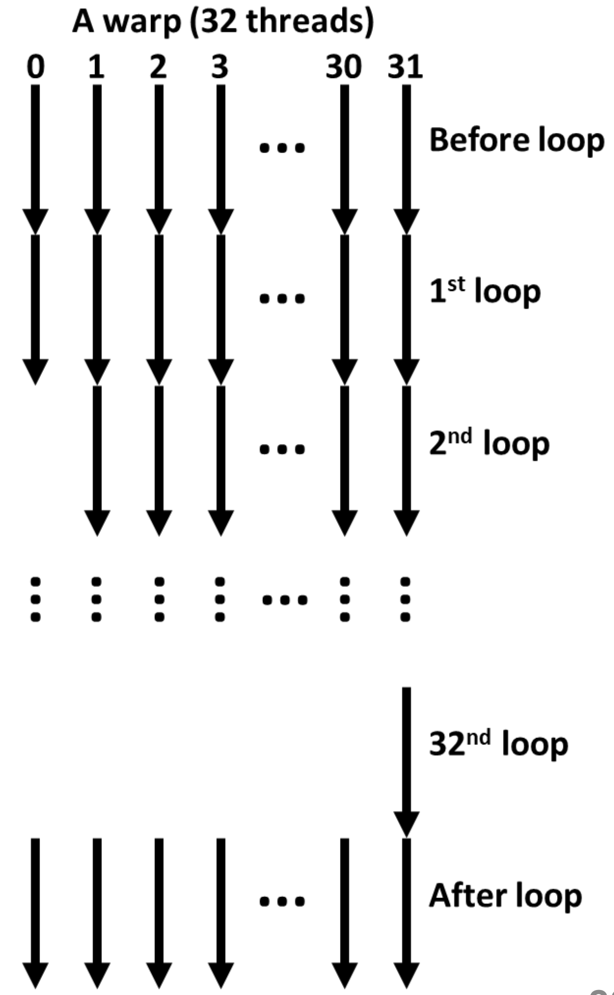
```
if(threadIdx.x < 24) {

      A

} else {

      B

}
C
```



# Parallel execution – SM-inside level

Warp divergence example: looping

```
 ...
for (int i = 0; i ≤ threadIdx.x; i++)
{
     ...
}
 ...
```



# Avoiding Branch Divergence

- Make branch granularity a multiple of warp size.

```
__global__ void divergence1(float* A, float* B, float* C, int n){
    int i = blockDim.x * blockIdx.x + threadIdx.x;
    if (i < n){
        if (i % 2)
            C[i] = A[i] * B[i];
        else
            C[i] = A[i] / B[i];
    }
}
```

```
__global__ void divergence2(float* A, float* B, float* C, int n)
{
    int i = blockDim.x * blockIdx.x + threadIdx.x;
    if (i < n) {
        if ( (i / 32) % 2 == 0)
            C[i] = A[i] * B[i];
        else
            C[i] = A[i] / B[i];
    }
}
```

# Example: warp divergence

- Task: adding 2 matrixes
  - Matrix size 1000×1000
  - Each thread computes an element in the result matrix
  - Block size 32×32
- How many diverged warps?

A. 0

B. 1000 ✓

C. 1024

D. 2000

E. I don't know

# Example: warp divergence

Task: adding 2 matrixes

- Matrix size 1000×1000
- Each thread computes an element in the result matrix
- Block size 32×32

Execution time of a diverged warp vs non-diverged warp? (we don't consider non-diverged warps in which all 32 threads fail the if condition and do nothing)

A. Faster

B. Slower

✓ C. Equal

D. I don't know

# Check Divergent with ncu

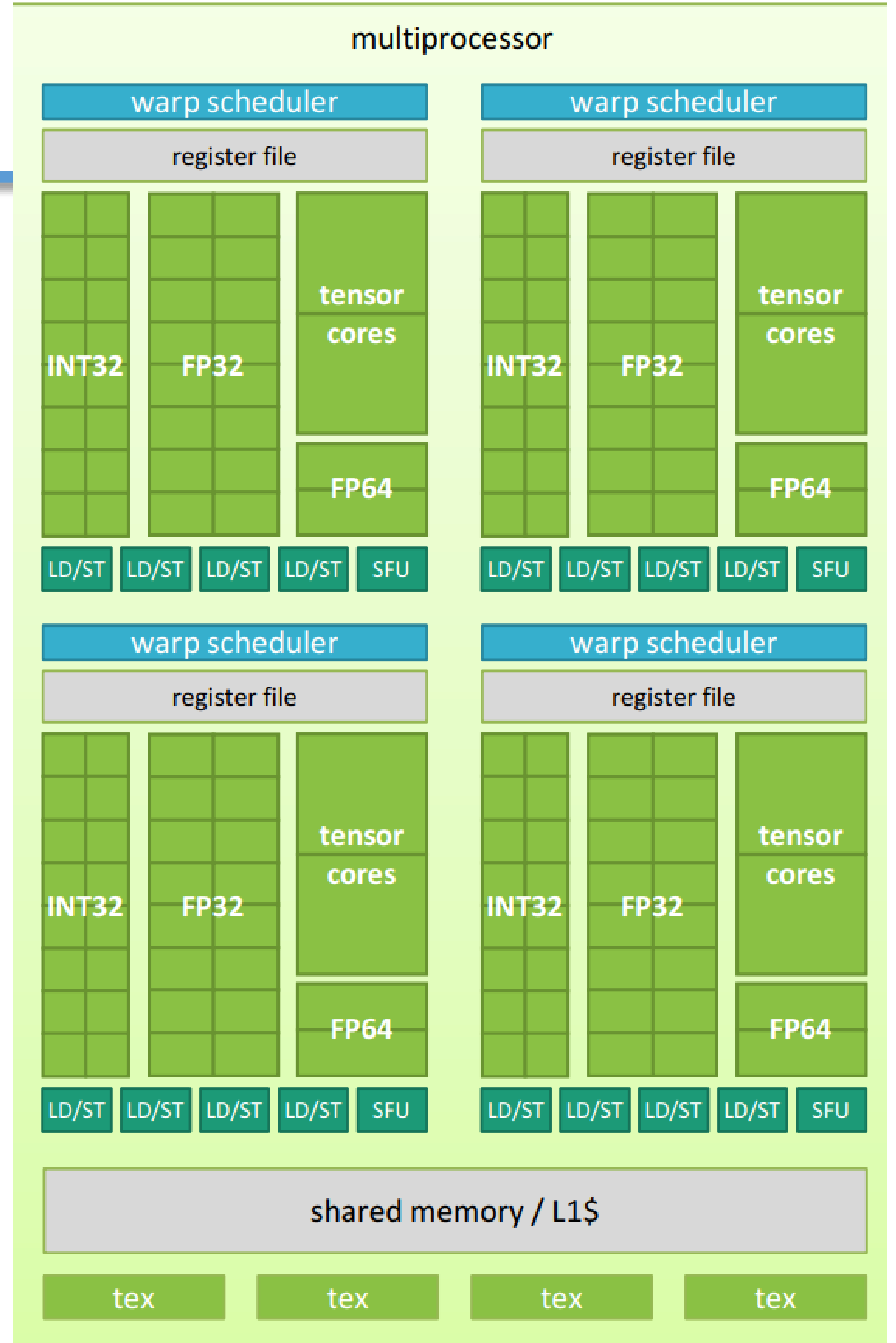
- Demo!

```
# Check if ncu is available
!which ncu

# Profile branch efficiency with ncu
!ncu --metrics
smsp__sass_branch_targets_threads_divergent.sum,smsp__sass_branch_targets.sum
./warp_demo

# May need to compile without optimize
!nvcc -G -arch=sm_75 YourCode.cu
```

- Check: *smsp__sass_branch_targets_threads_divergent.sum*

# SM and warp

- Each SM contains warp schedulers.
- Each warp scheduler has: a register file, 32-bit integer (INT32), 32-bit floating point (FP32), 64-bit floating point (FP64), and tensor core ALUs, load/store units (LD/ST), and a special function unit (SFU).
- Each SM also contains a local shared memory as well as a number of texture units that are shared among its warp schedulers.
- Part of the local shared memory is used as an L1 cache (L1$), the rest can be used to facilitate fast communication between threads that reside on the same SM

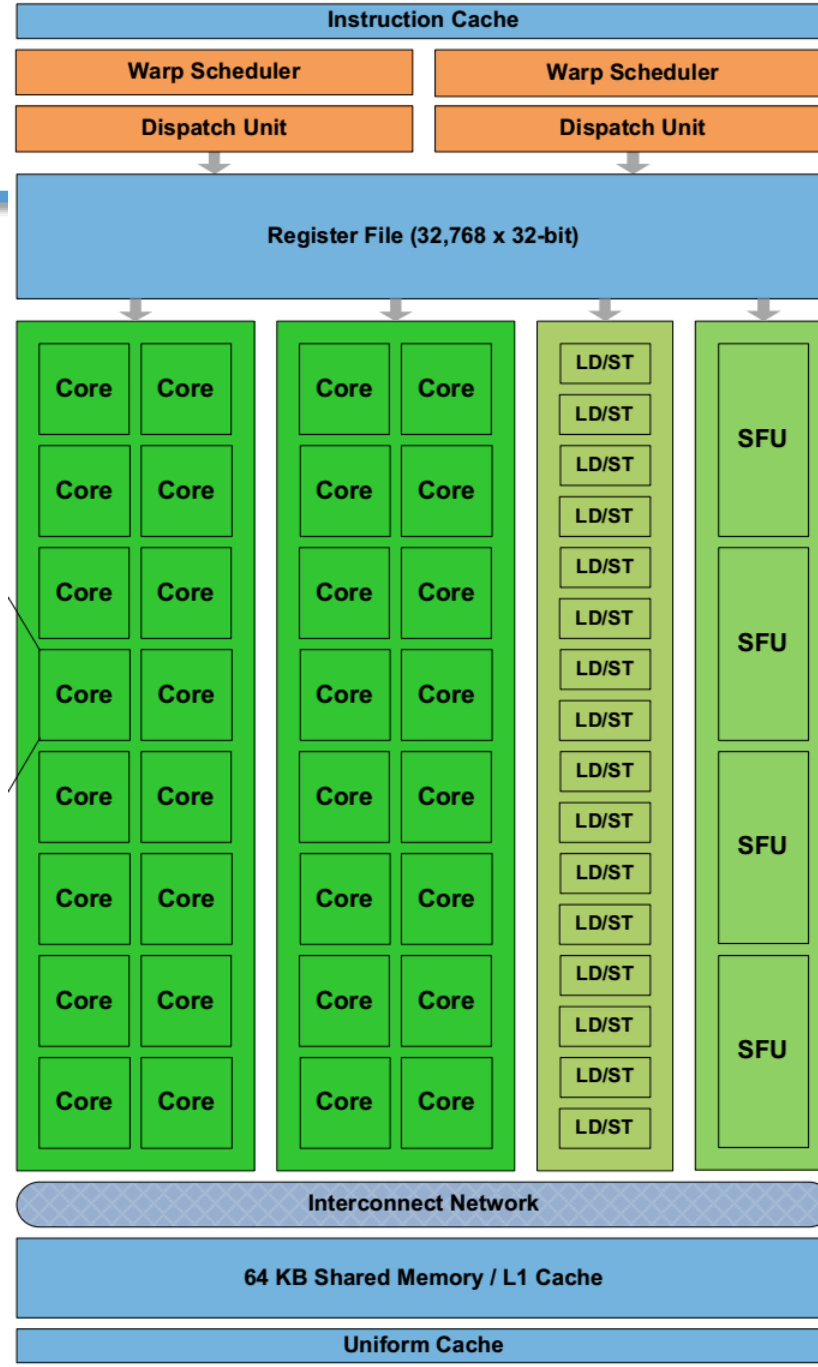


# Warp scheduling & latency tolerance

- In SM, are warps executed in parallel?
  - Not totally so

    E.g., Fermi SM (2.x) can contain at most 48 warps (1536 threads), but it has only 32 cores
- So, in SM how are warps executed exactly?
- Why does SM contain many warps / threads compared to its execution resources (cores)?

Image source: NVIDIA. Fermi white paper




Fermi Streaming Multiprocessor (SM)

# Warp scheduling & latency tolerance

Before continuing, let's review instruction pipeline

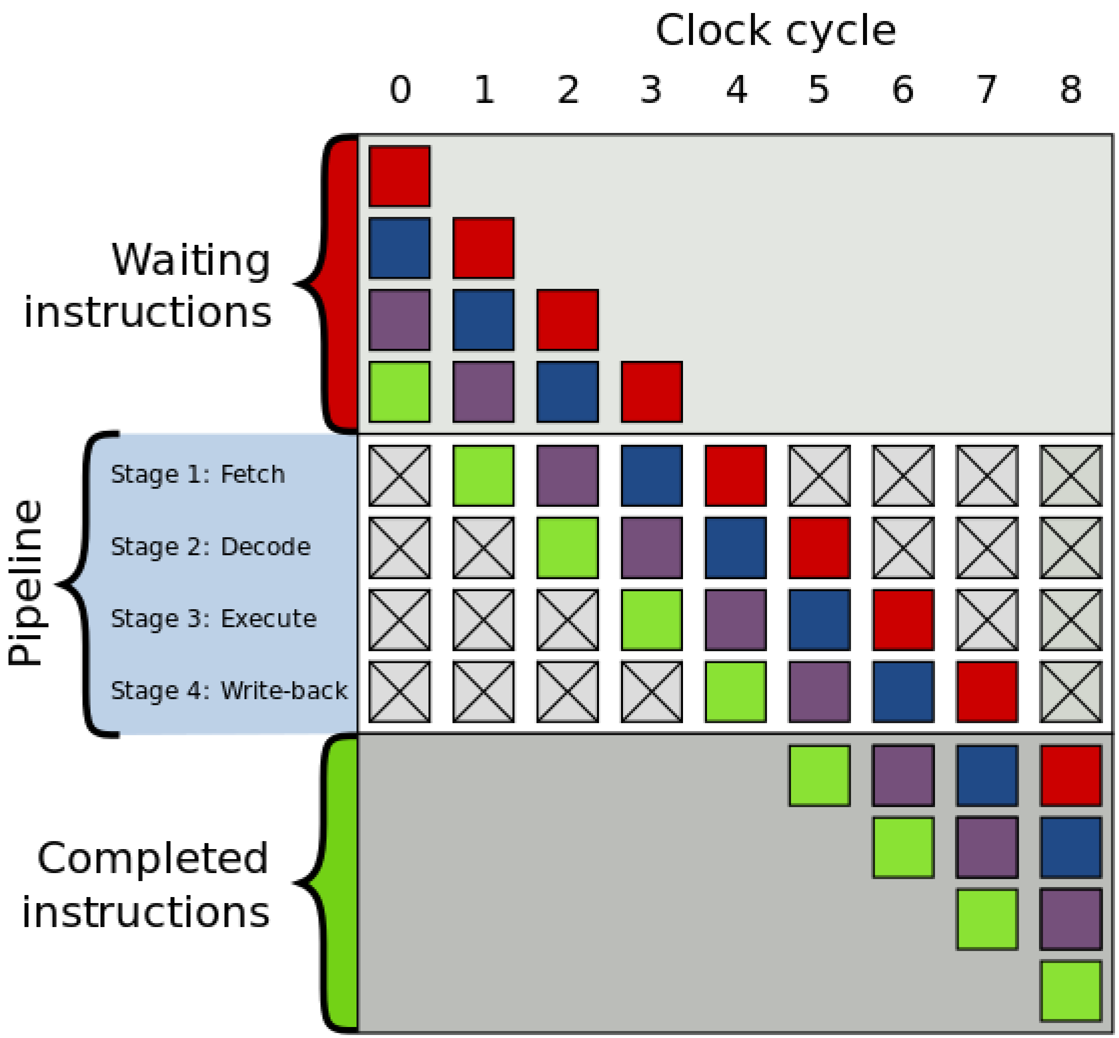
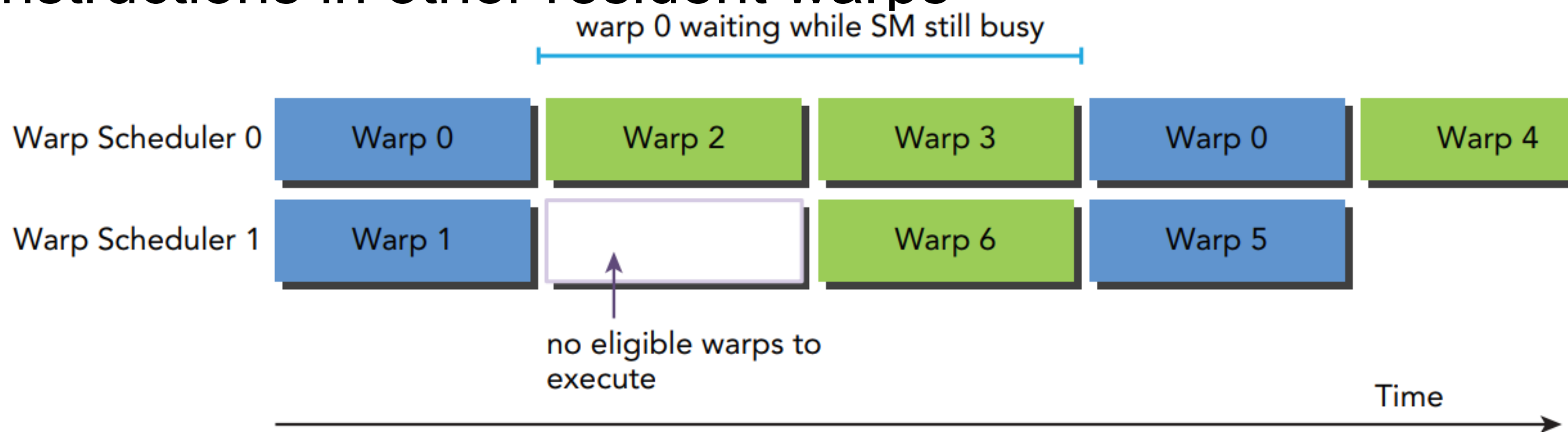



Image source: https://en.wikipedia.org/wiki/Instruction_pipelining

# Warp scheduling & latency tolerance

- Every clock cycle:
  - Warp scheduler elects eligible warp
  - Schedules instruction from that warp
- **Latency**: #clock cycles between an instruction being issued and being completed
- **Full compute resource utilization** is achieved when *all warp schedulers have an eligible warp at every clock cycle*.
- The latency of each instruction can be **hidden** by issuing other instructions in other resident warps

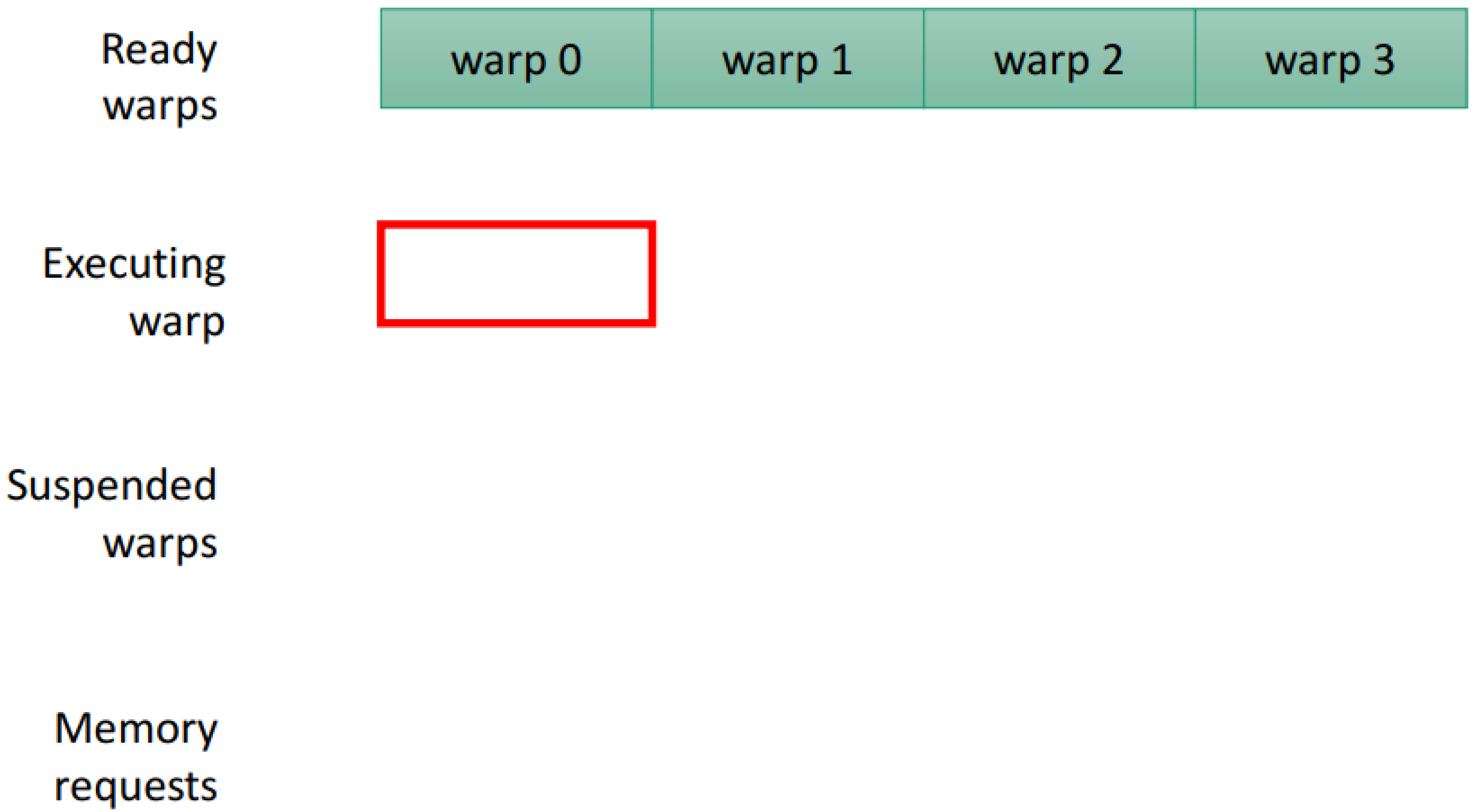


# Warp scheduling & latency tolerance

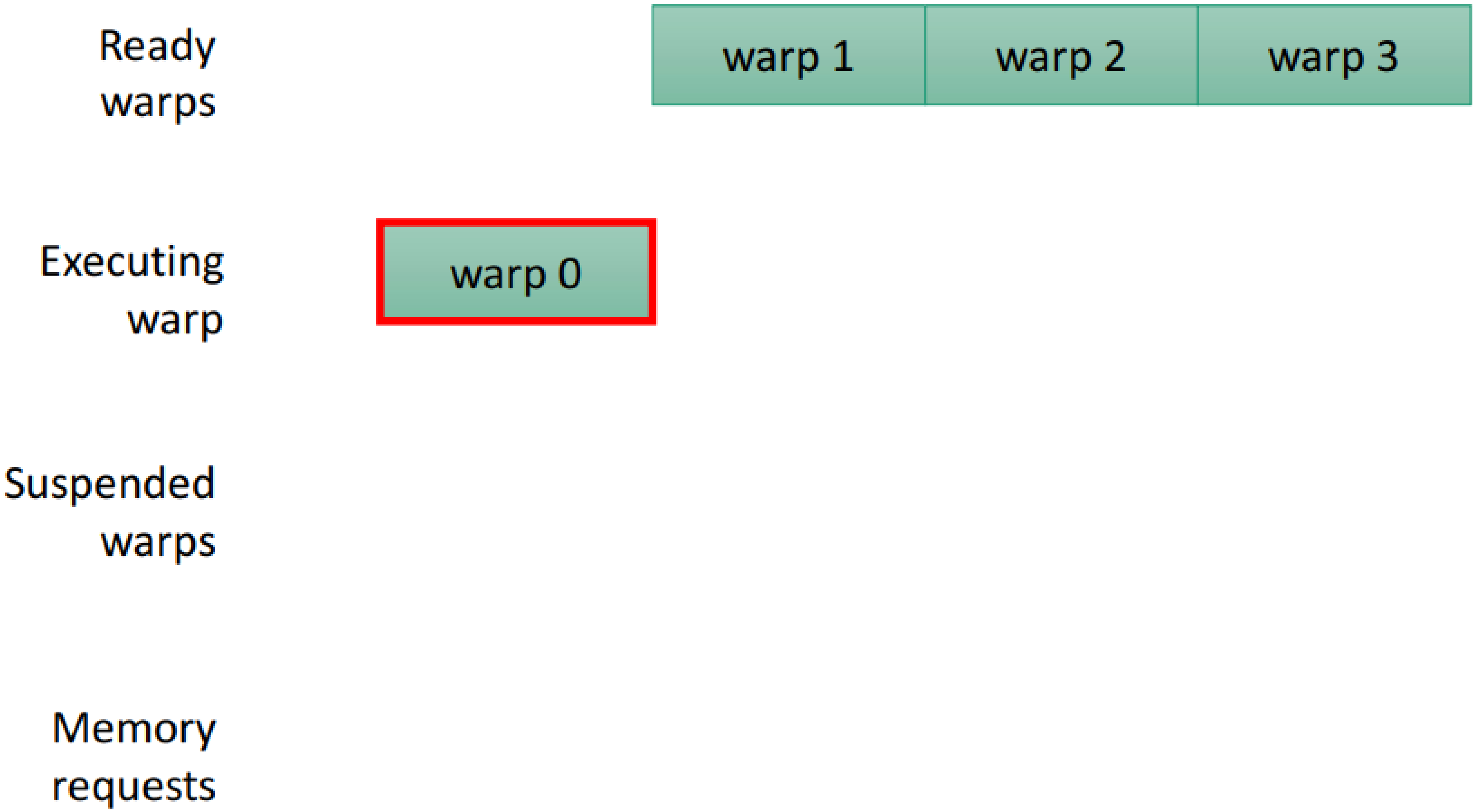
- Two instruction types:
  - Arithmetic instructions: 10-20 cycles
  - Memory instructions: 400-800 cycles for global memory accesses
- If an instruction has latency of n clock cycles, then scheduler will need ~n ready instructions (coming from the same warp or other warps) to “hide” latency, keep pipelines full. (latency tolerance or latency hiding)

➔ desirable for an SM to have many more threads assigned to it than can be simultaneously supported with its execution resources to maximize the chance of finding a warp that is ready to execute at any point in time

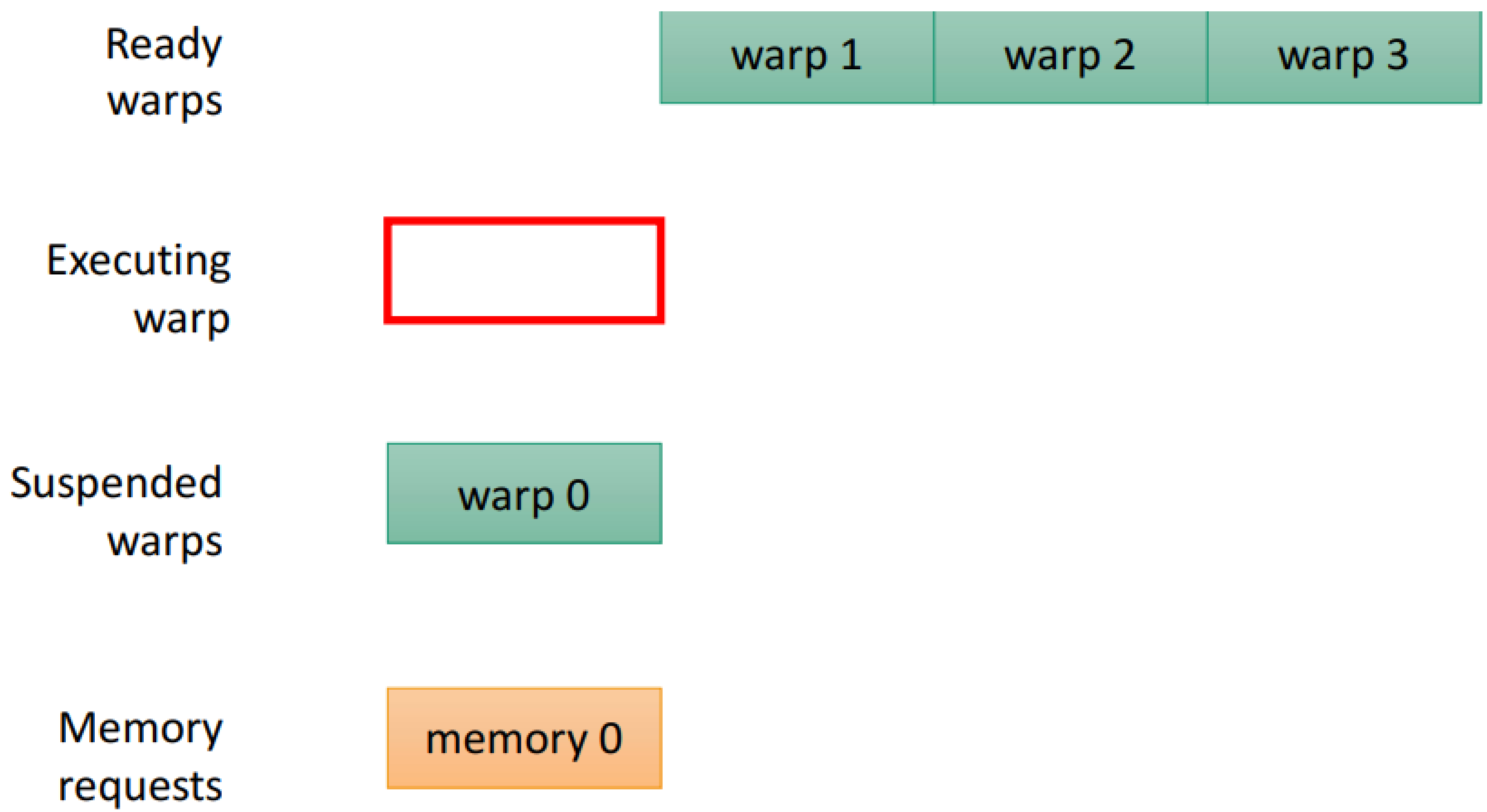
# Warp scheduling & latency tolerance



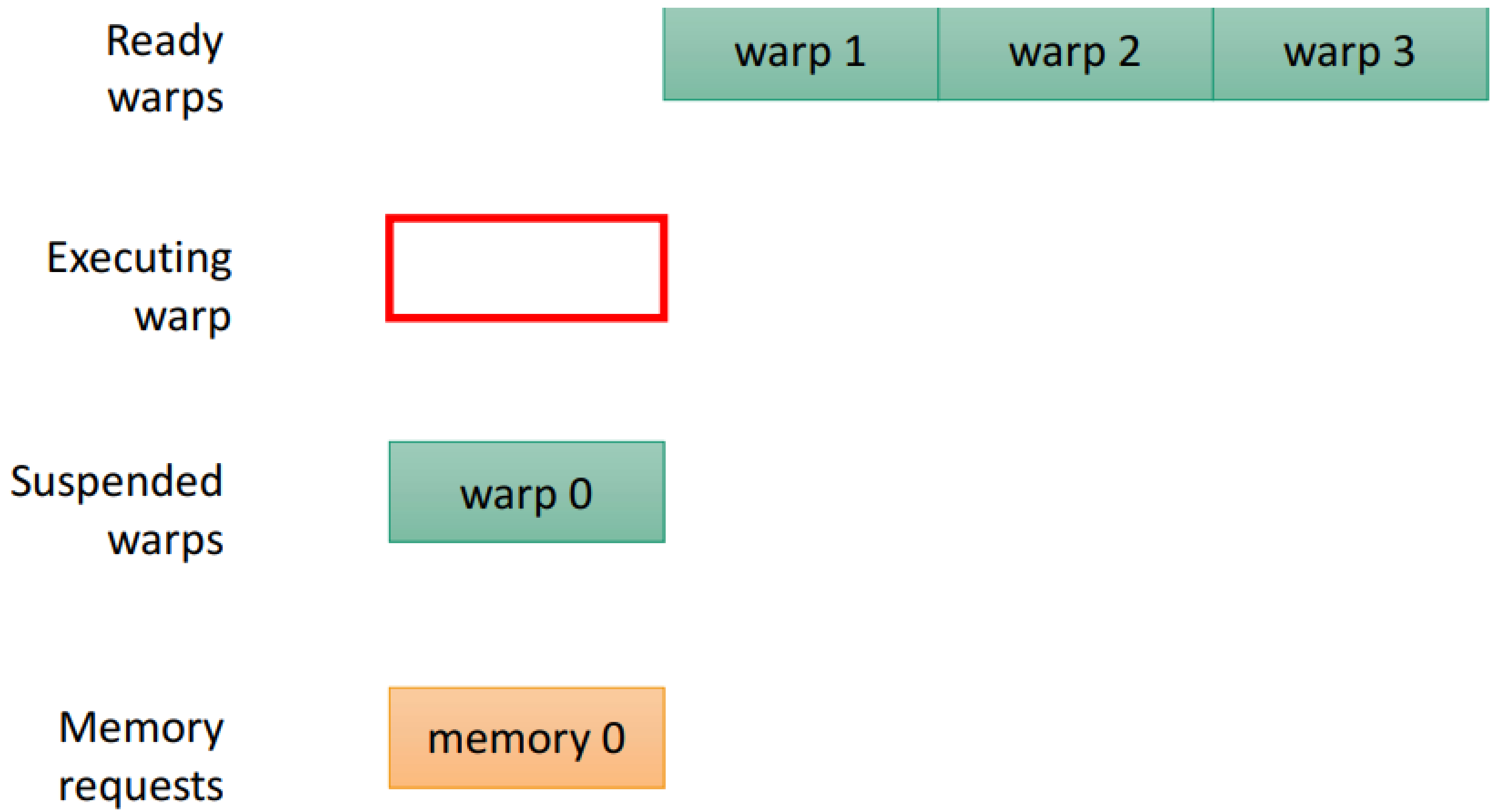
# Warp scheduling & latency tolerance



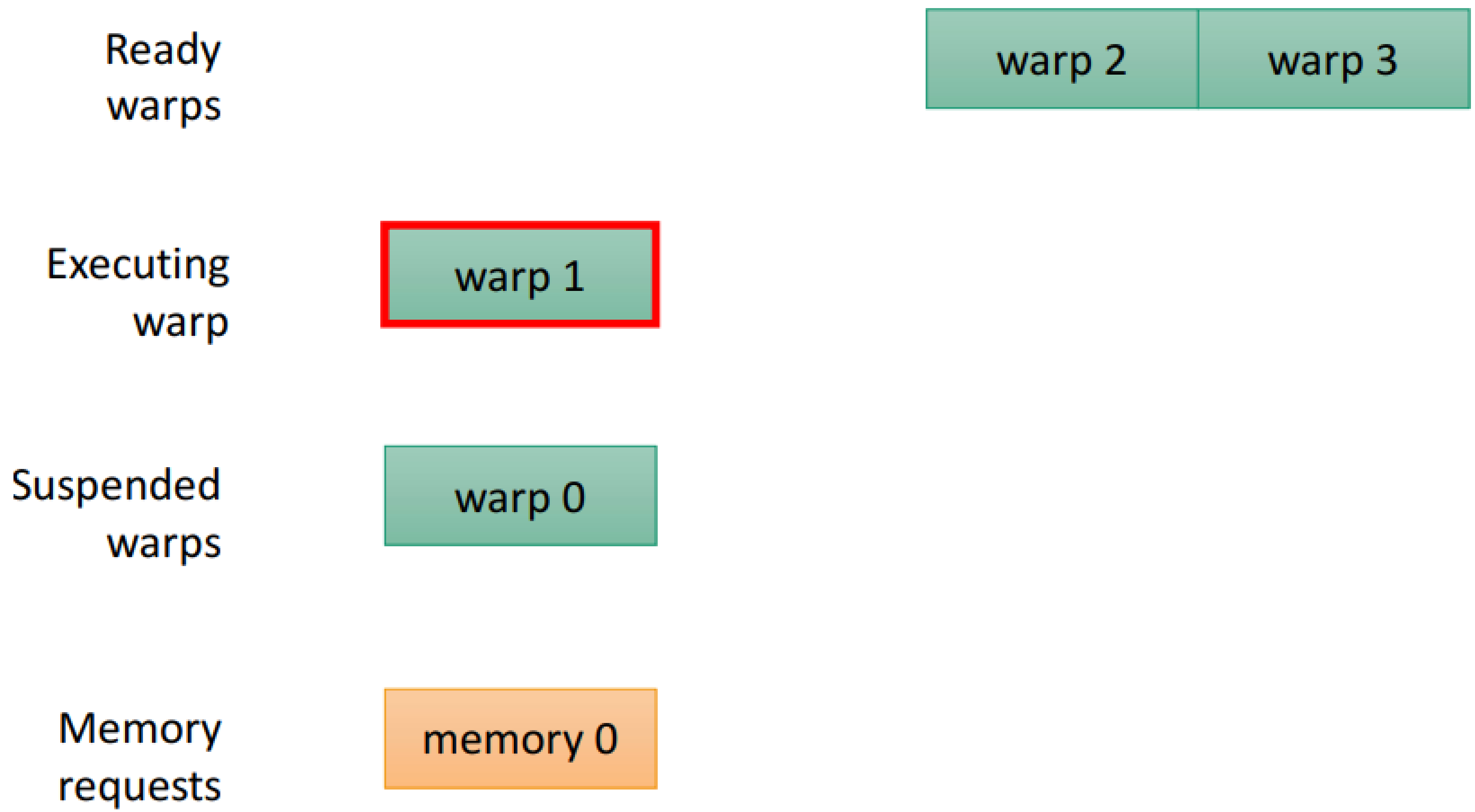
# Warp scheduling & latency tolerance



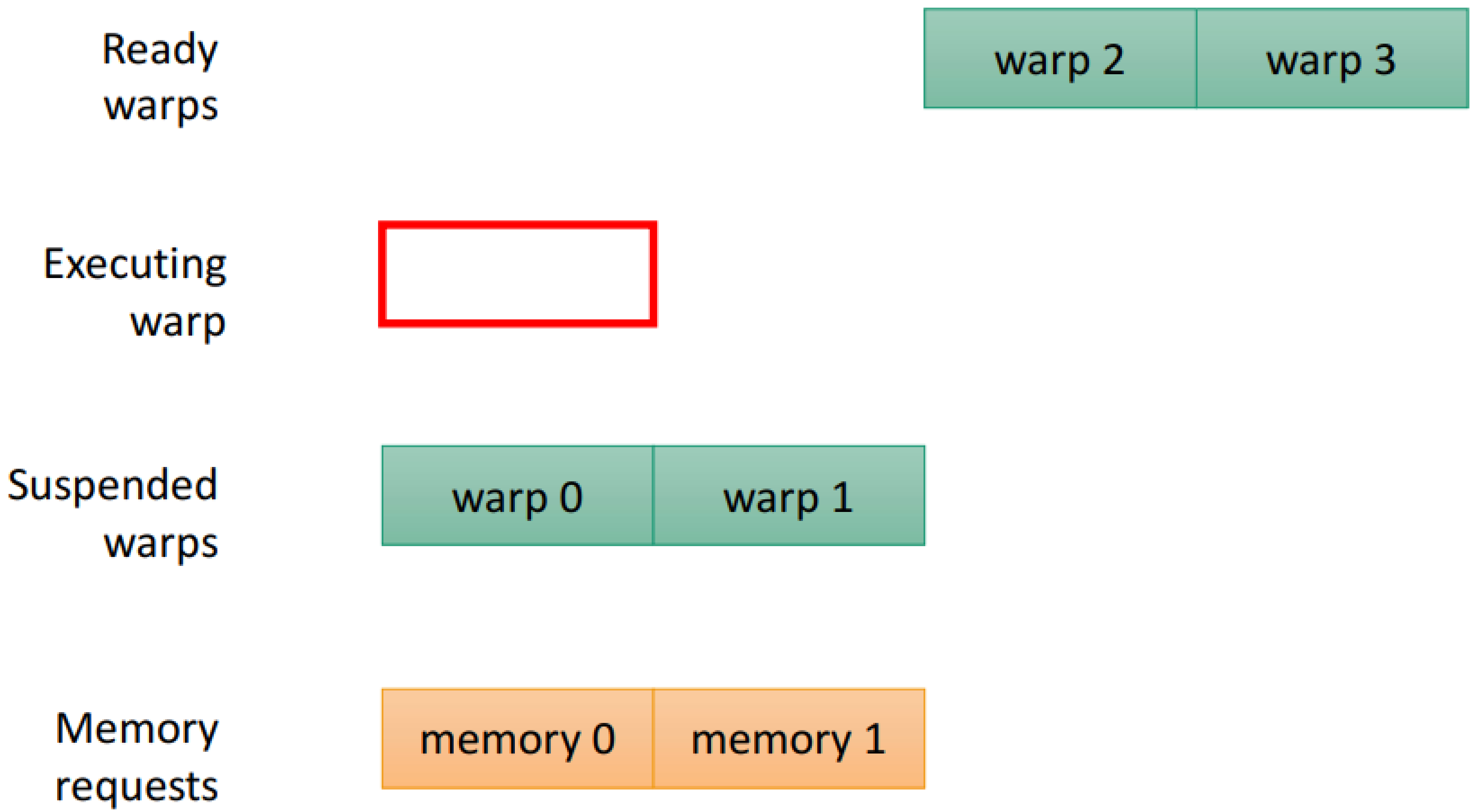
# Warp scheduling & latency tolerance



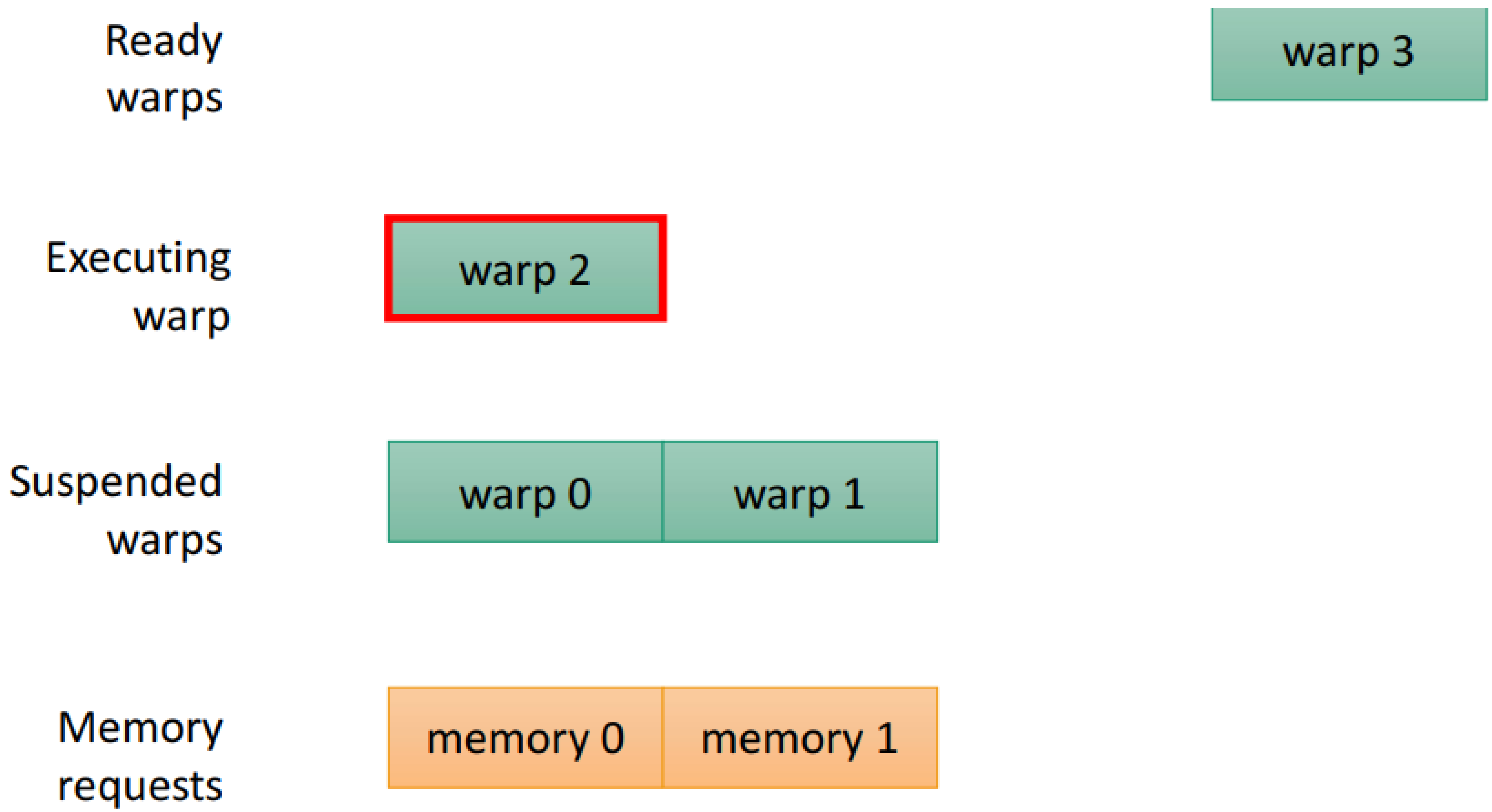
# Warp scheduling & latency tolerance



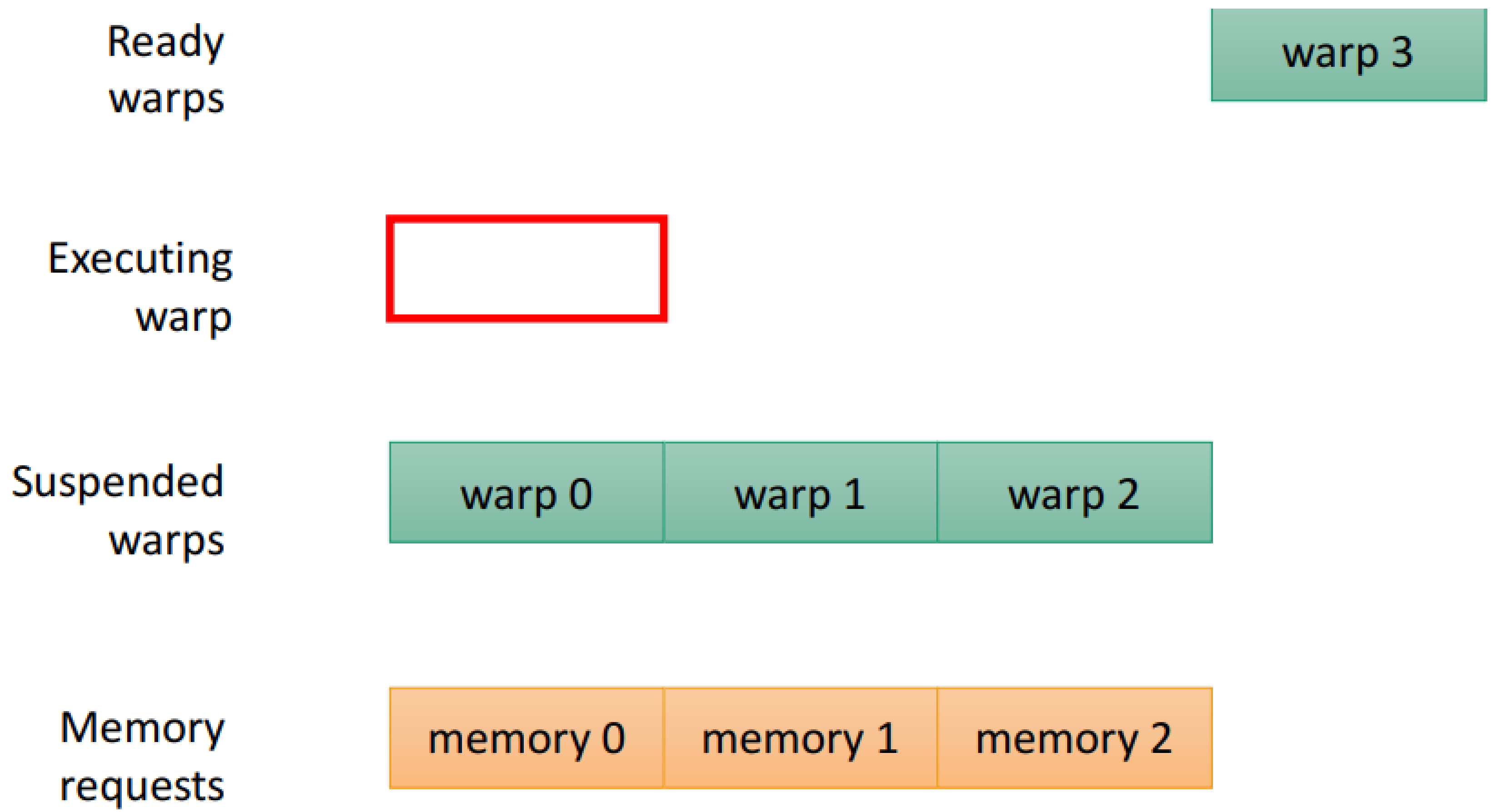
# Warp scheduling & latency tolerance



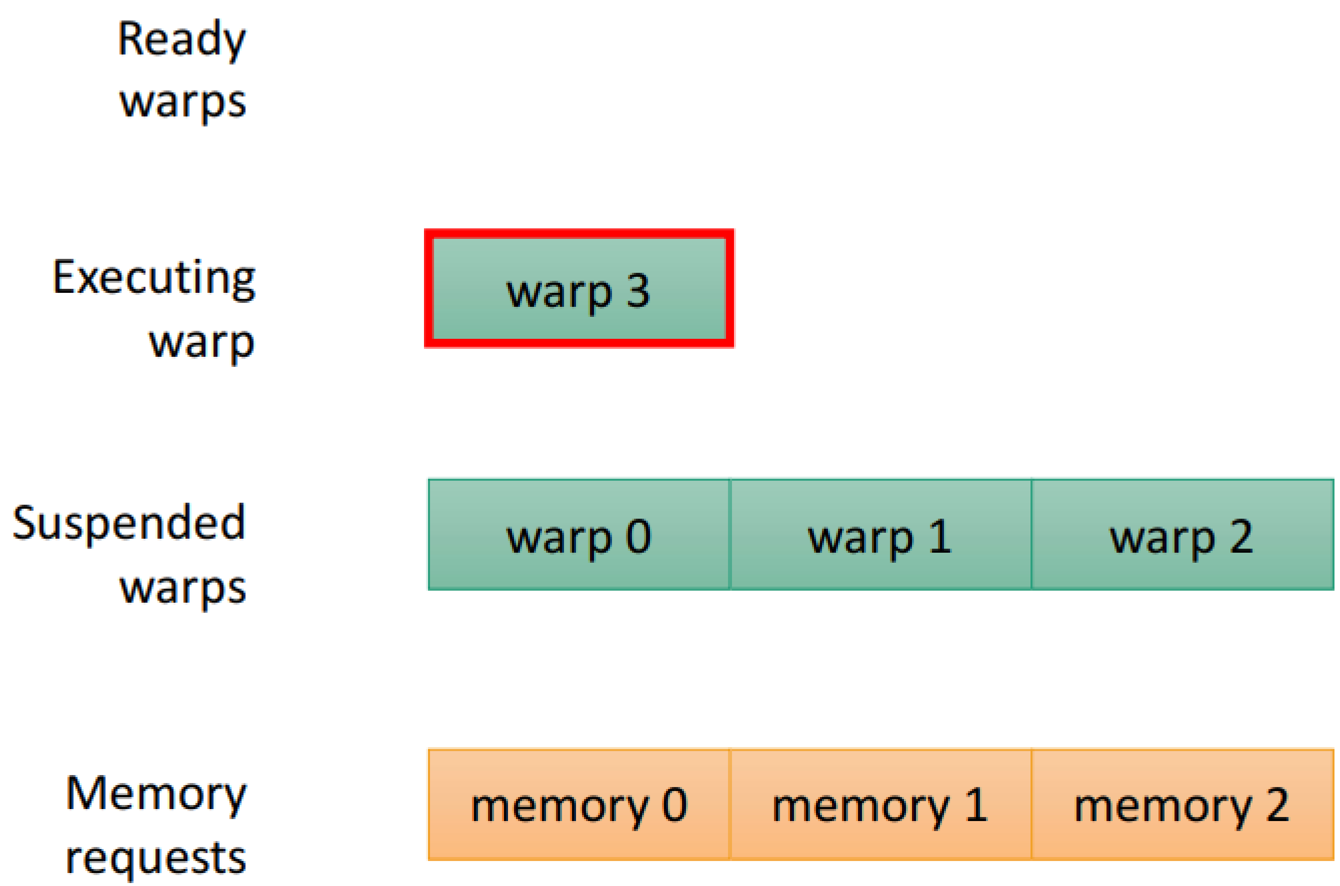
# Warp scheduling & latency tolerance



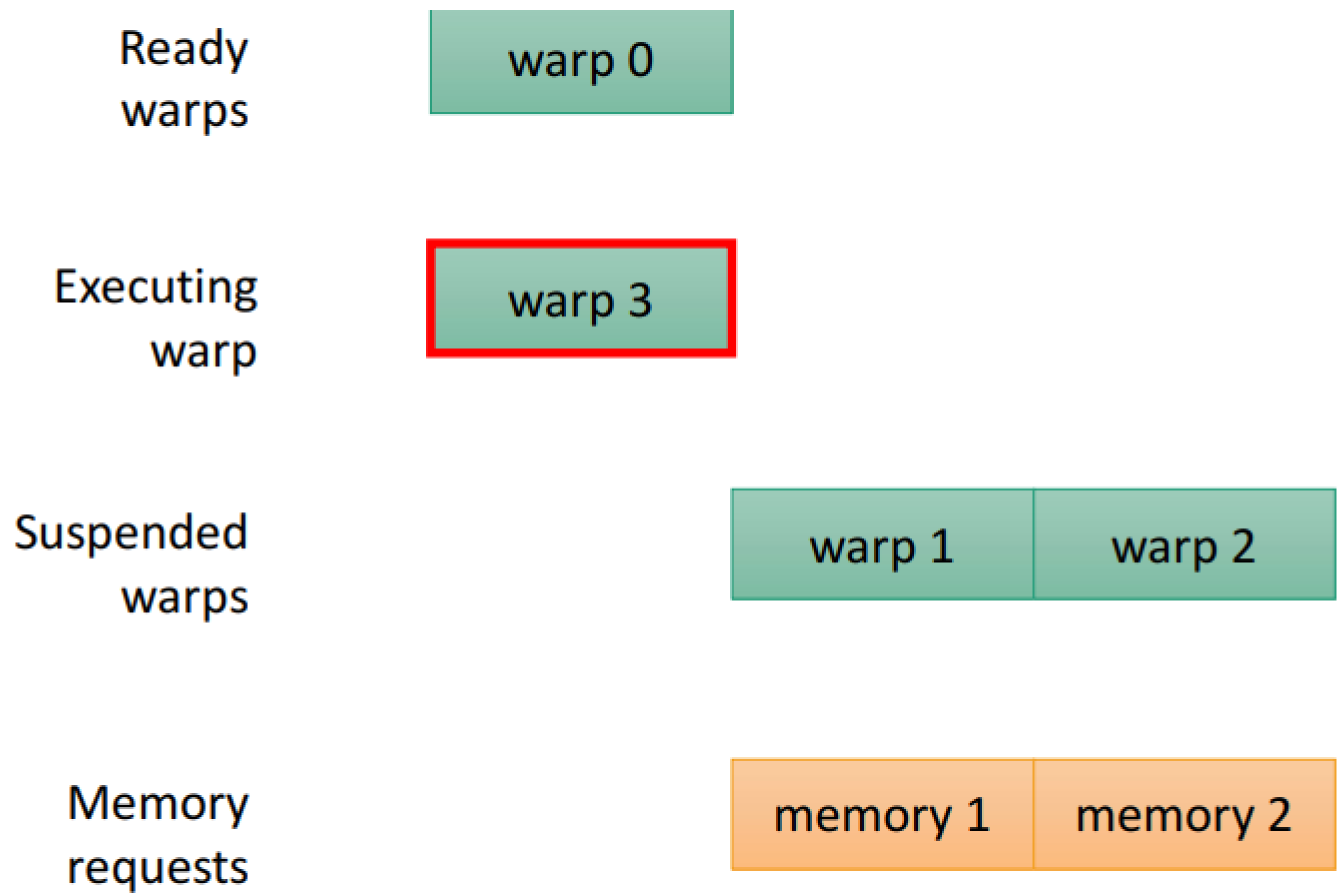
# Warp scheduling & latency tolerance



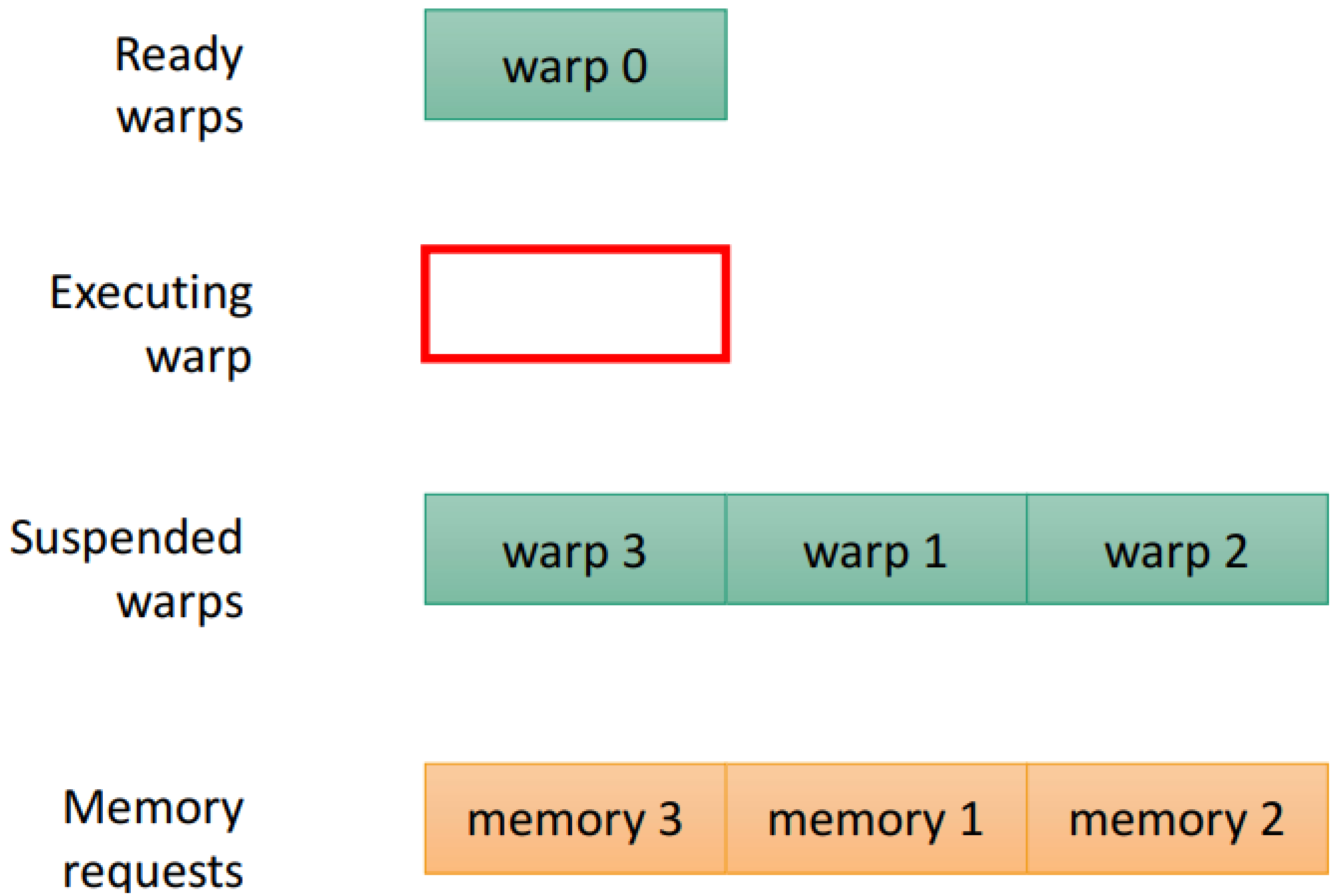
# Warp scheduling & latency tolerance



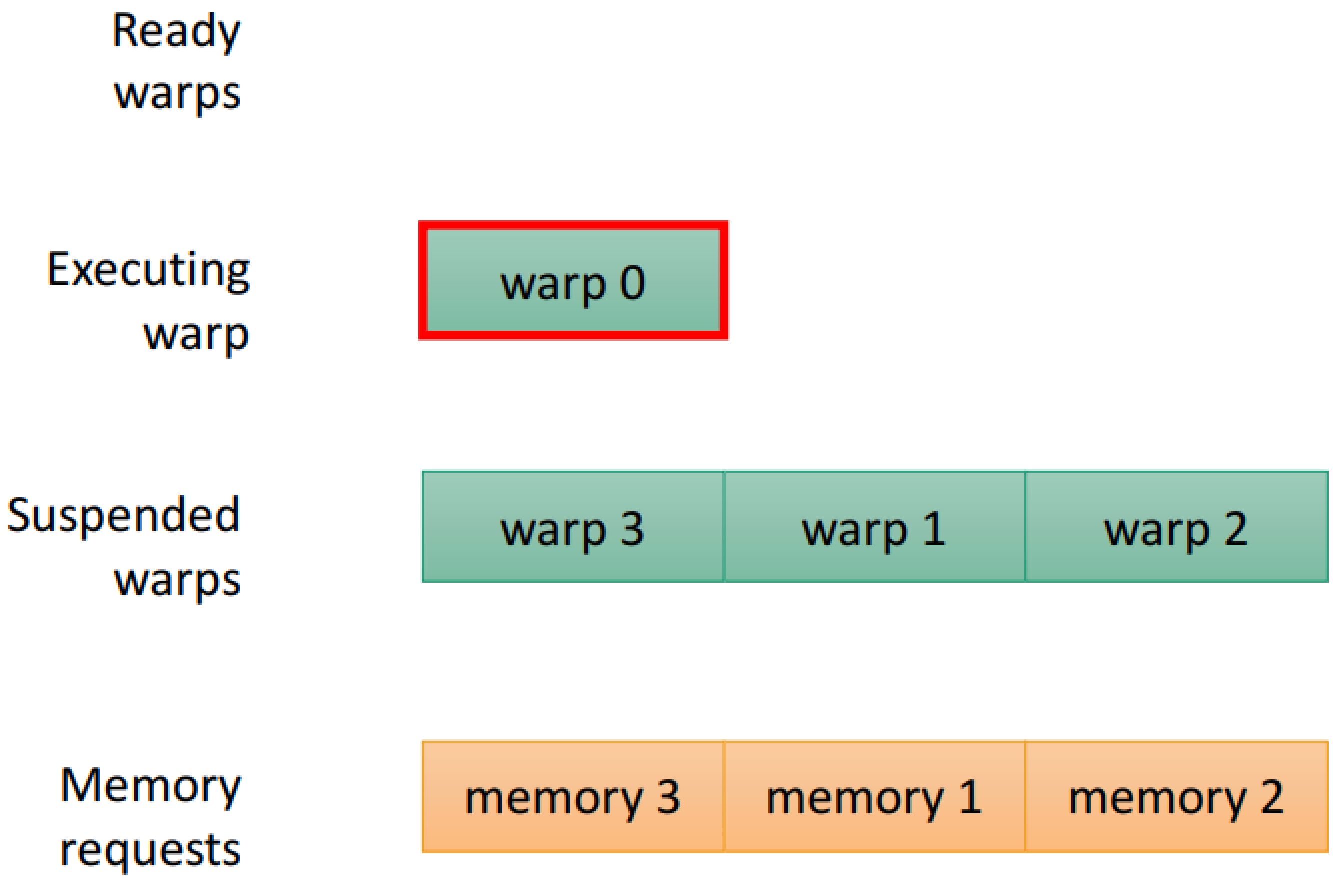
# Warp scheduling & latency tolerance



# Warp scheduling & latency tolerance



# Warp scheduling & latency tolerance



# Resource partitioning

- For each SM:
  - The more warp (threads), the better (to hide latency)
  - Maximum: 2048 Threads/SM = 64 warps/SM (*)
  - Not always can reach this maximum due to resource availability

$$\text{Occupancy} = \frac{\text{\#warps SM contains}}{\text{\# max warps SM can contain}}$$

(*) Depend on CC

# Resource partitioning

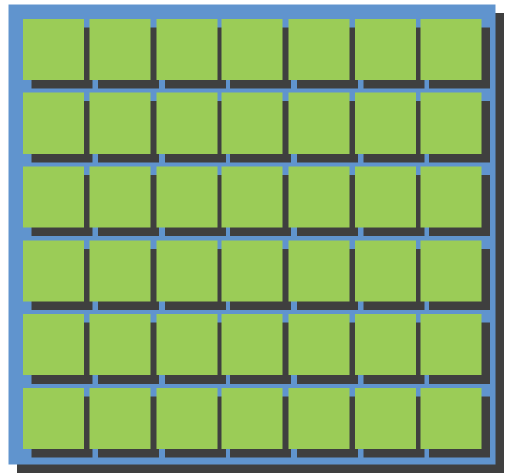
- The execution of a warp mainly consists of the following resources:
  - Program counters
  - Registers
  - Shared memory
- The number of blocks and warps that can simultaneously reside on an SM depends on the number of registers and amount of shared memory available on the SM and required by the kernel.

# Resource partitioning

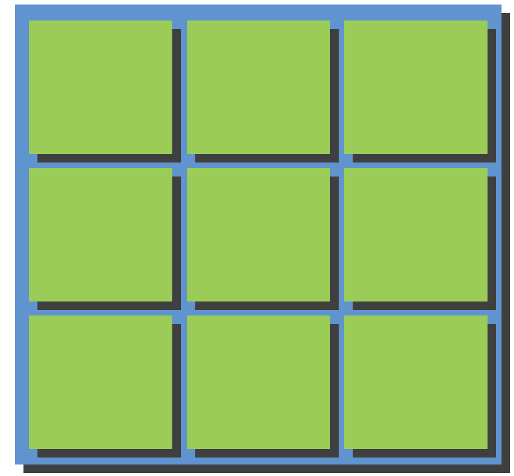
Registers per SM

Kepler: 64K

Fermi: 32K



More threads with fewer registers per thread

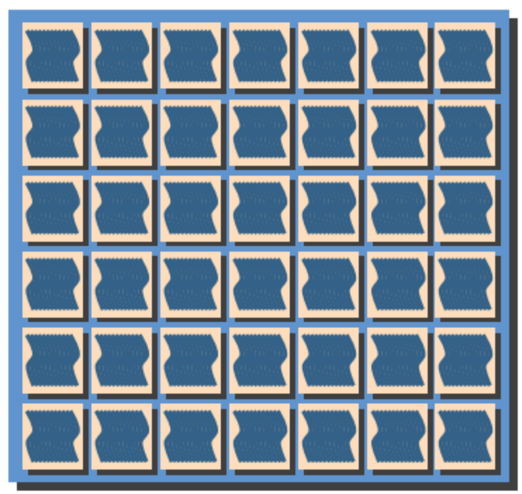


Fewer threads with more registers per thread

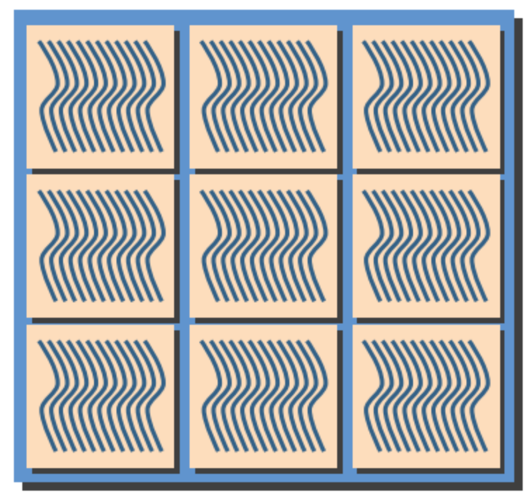
Shared Memory per SM

Kepler: up to 48K

Fermi: up to 48K



More blocks with less shared memory per block



Fewer blocks with more shared memory per block

# Resource partitioning - Example

- A100 GPU: 32 blocks/SM, 64 warps (2048 threads)/ SM, 1024 threads/block, and 65,536 registers/SM
- BlockSize = 32 ➔ ? Theads/SM ➔ Occupacy = ?
- BlockSize = 768 ➔ ? Theads/SM ➔ Occupacy = ?
- Kernel uses 64 registers per thread ➔ ? Theads/SM ➔ Occupacy = ?
- 31 registers/thread, 512 threads/block ➔ ? Theads/SM ➔ Occupacy = ?
- 33 registers/thread, 512 threads/block ➔ ? Theads/SM ➔ Occupacy = ?

# Resource partitioning - Example

- A100 GPU: 32 blocks/SM, 64 warps (2048 threads)/ SM, 1024 threads/block, and 65,536 registers/SM
- BlockSize = 32 ➔ 1024 Theads/SM ➔ Occupacy = 50%
- BlockSize = 768 ➔ 1538 Theads/SM ➔ Occupacy = 75%
- Kernel uses 64 registers per thread ➔ 1024 Theads/SM ➔ Occupacy = 50%
- 31 registers/thread, 512 threads/block ➔ 2048 Theads/SM ➔ Occupacy = 100%
- 33 registers/thread, 512 threads/block ➔1538 Theads/SM ➔ Occupacy = 75%

# Resource partitioning - Example

- **Tesla T4 GPU**: 16 blocks/SM, 32 warps (1024 threads)/SM, 1024 threads/block, 65,536 registers/SM
- BlockSize = 32 ➔ ? Theads/SM ➔ Occupacy = ?
- BlockSize = 768 ➔ ? Theads/SM ➔ Occupacy = ?
- Kernel uses 64 registers per thread ➔ ? Theads/SM ➔ Occupacy = ?
- 31 registers/thread, 512 threads/block ➔ ? Theads/SM ➔ Occupacy = ?
- 65 registers/thread, 256 threads/block ➔? Theads/SM ➔ Occupacy = ?

# Resource partitioning - Example

- **Tesla T4 GPU**: 16 blocks/SM, 32 warps (1024 threads)/SM, 1024 threads/block, 65,536 registers/SM
- BlockSize = 32 ➔ 512 Theads/SM ➔ Occupacy = 50%
- BlockSize = 768 ➔ 768 Theads/SM ➔ Occupacy = 75%
- Kernel uses 64 registers per thread ➔ 1024 Theads/SM ➔ Occupacy = 100%
- 31 registers/thread, 512 threads/block ➔ 1024 Theads/SM ➔ Occupacy = 100%
- 65 registers/thread, 256 threads/block ➔768 Theads/SM ➔ Occupacy = 75%

# Resource partitioning

- Accurate determination of the number of threads running in each SM can be difficult
- Tool: [CUDA Occupancy Calculator]

## CUDA Occupancy Calculator

Compute Capability version

8.6

CUDA version

11.1

Threads per block

256

Registers per thread

32

Shared memory per block

2048 bytes

Calculate

- *Or code your own Calculator*

# Example: # needed warps for Kepler SM

- Each SM has 4 warps schedulers

  → need 4+ warps / SM

  In practice, need many more than 4 to hide latency (Kepler SM can contain up to 64 warps)

- With programs whose performance is limited by computation throughput (latency of a computation instruction: 10+ clock cycles)
  - Without ILP (Instruction Level Parallelism – adjacent independent instructions in a warp):

    need 4 schedulers × 10+ cycles

    = 40+ warps / SM

  - With ILP: may need fewer warps

# More about cores
# Example: Kepler SM


- **192 fp32 lanes (cores)**
  - fp32 math
  - Simple int32 math (add,min,etc.)
- **64 fp64 lanes**
- **32 SFU lanes**
  - Int32 multiplies, etc.
  - Transcendentals
- **32 LD/ST lanes**
  - GMEM, SMEM, LMEM accesses
- **16 TEX lanes**
  - Texture access
  - Read-only GMEM access

Image source: Paulius Micikevicius. Performance Optimization, GTC2013

# More about cores
# Example: Kepler SM


- **192 fp32 lanes (cores)**
  - fp32 math
  - Simple int32 math (add,min,etc.)
- **64 fp64 lanes**
- **32 SFU lanes**
  - Int32 multiplies, etc.
  - Transcendentals
- **32 LD/ST lanes**

NVIDIA “core” refers to fp32 core

# fp32 cores > # fp64 cores
→ should use 32-bit float when possible

Image source: Paulius Micikevicius. Performance Optimization, GTC2013

# Guide for block size selection

- Block size should be a multiple of 32 (warp size)
- Block size should be selected so that SM has enough warps to hide latency and make use of available resources
  - Occupancy measure: the ratio of # warps SM contains to # max warps SM can contain

    Block size should be selected so that occupancy is high

      Example: assume SM can contain up to 8 blocks and 1536 threads (48 warps); what block size should we pick: 64, 256, 1024
  - Not necessary: 100% occupancy = max performance
    - Only need enough warp to hide latency and make use of available resources
    - If warps have ILP, we will need fewer warps
    - …

# Guide for block size selection

- High occupancy (75-100%):
  - ✓ Good for memory-bound kernels
  - ✓ Helps hide memory latency
  - ✗ May not help compute-bound kernels
- Lower occupancy (50-75%):
  - ✓ Can be fine if compute-bound
  - ✓ More registers per thread
  - ✓ More shared memory per block

# Block size summary

- Always use multiples of 32 (warp size)
- Default starting points:
  - 1D: 256 threads
  - 2D: 16×16 (256 threads)
  - 3D: 8×8×8 (512 threads)
- Aim for 50-100% occupancy (but profile to verify benefit)
- Memory-bound → higher occupancy (512-1024 threads)
- Compute-bound → moderate occupancy (128-256 threads)
- Check register and shared memory usage
- Profile with real data - theoretical occupancy ≠ performance
- Consider problem size - very small problems may need smaller blocks
- Test multiple sizes - optimal varies by kernel and GPU

# Reference


- [1] Wen-Mei, W. Hwu, David B. Kirk, and Izzat El Hajj. *Programming Massively Parallel Processors: A Hands-on Approach*. Morgan Kaufmann, 2022
- [2] Cheng John, Max Grossman, and Ty McKercher. *Professional Cuda C Programming*. John Wiley & Sons, 2014


# THE END