Introduction to CUDA Performance Optimization

Athena Elafrou, Guillaume Thomas Collignon, NVIDIA DevTech Compute

GPU Technology Conference, March 18th 2024

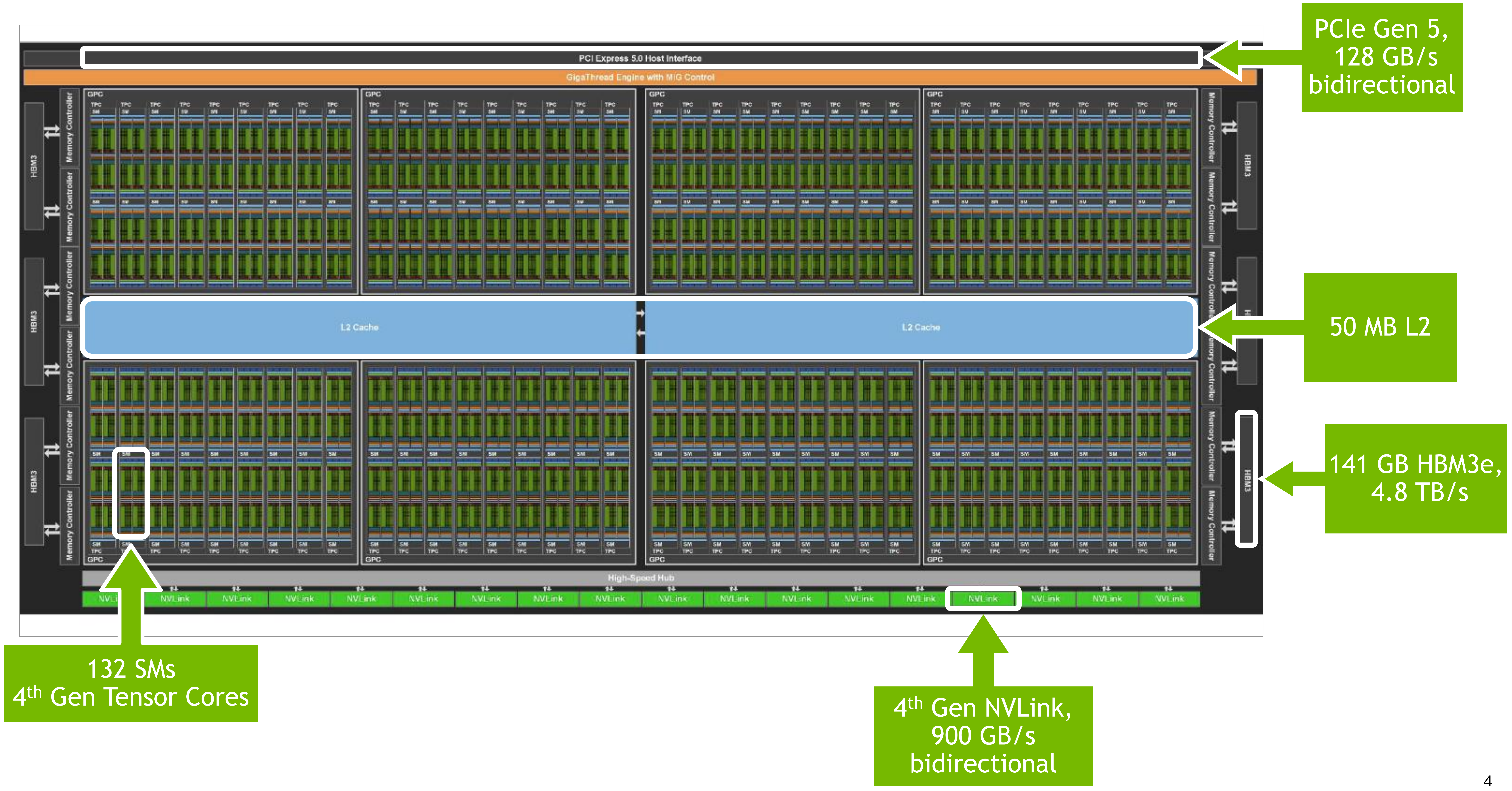
Agenda

- GPU Architecture and CUDA Programming Basics
- Fundamental Performance Optimizations
- Summary

GPU Architecture and CUDA Programming Basics

GPU Overview

NVIDIA H200 SXM



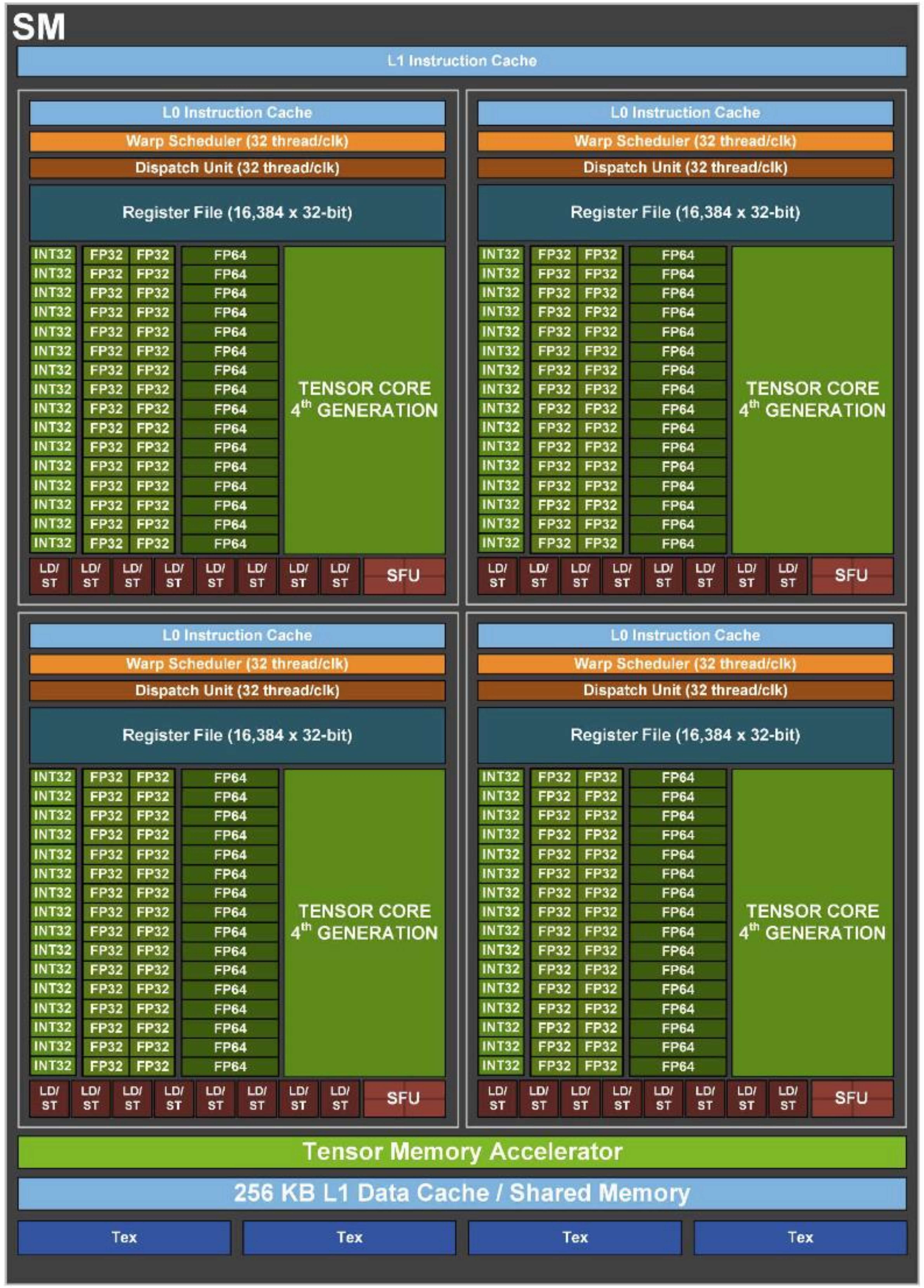
PCI Express 5.0 Host Interface
GigaThread Engine with MIG Control
GPC
TPC
SM
Memory Controller
HBM3
L2 Cache
High-Speed Hub
NVLink
PCIe Gen 5, 128 GB/s bidirectional
50 MB L2
141 GB HBM3e, 4.8 TB/s
132 SMs 4th Gen Tensor Cores
4th Gen NVLink, 900 GB/s bidirectional

Streaming Multiprocessor (SM)

Hopper architecture

- 128 FP32 cores
- 64 FP64 cores
- 64 INT32 cores
- 4 mixed-precision Tensor Cores
- 16 special function units (transcendentals)
- 4 warp schedulers

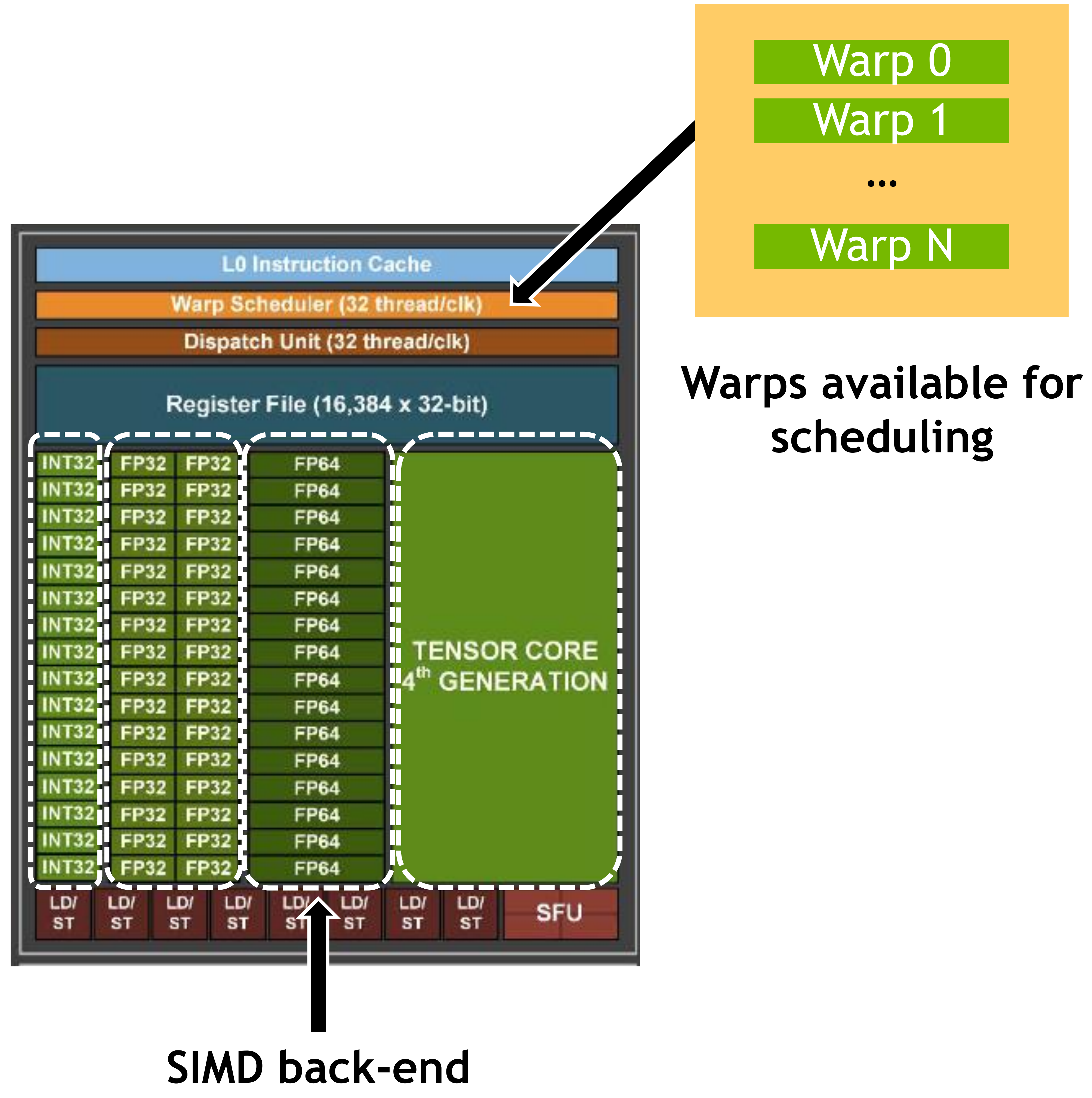
- 32 LD/ST units
- 64K 32-bit registers
- 256 KiB unified L1 data cache and shared memory
- Tensor Memory Accelerator (TMA)


SM
L1 Instruction Cache
L0 Instruction Cache
Warp Scheduler (32 thread/clk)
Dispatch Unit (32 thread/clk)
Register File (16,384 x 32-bit)
INT32
FP32
FP64
TENSOR CORE 4th GENERATION
LD/ ST
SFU
Tensor Memory Accelerator
256 KB L1 Data Cache / Shared Memory
Tex

SIMT Architecture

Single-Instruction, Multiple-Thread

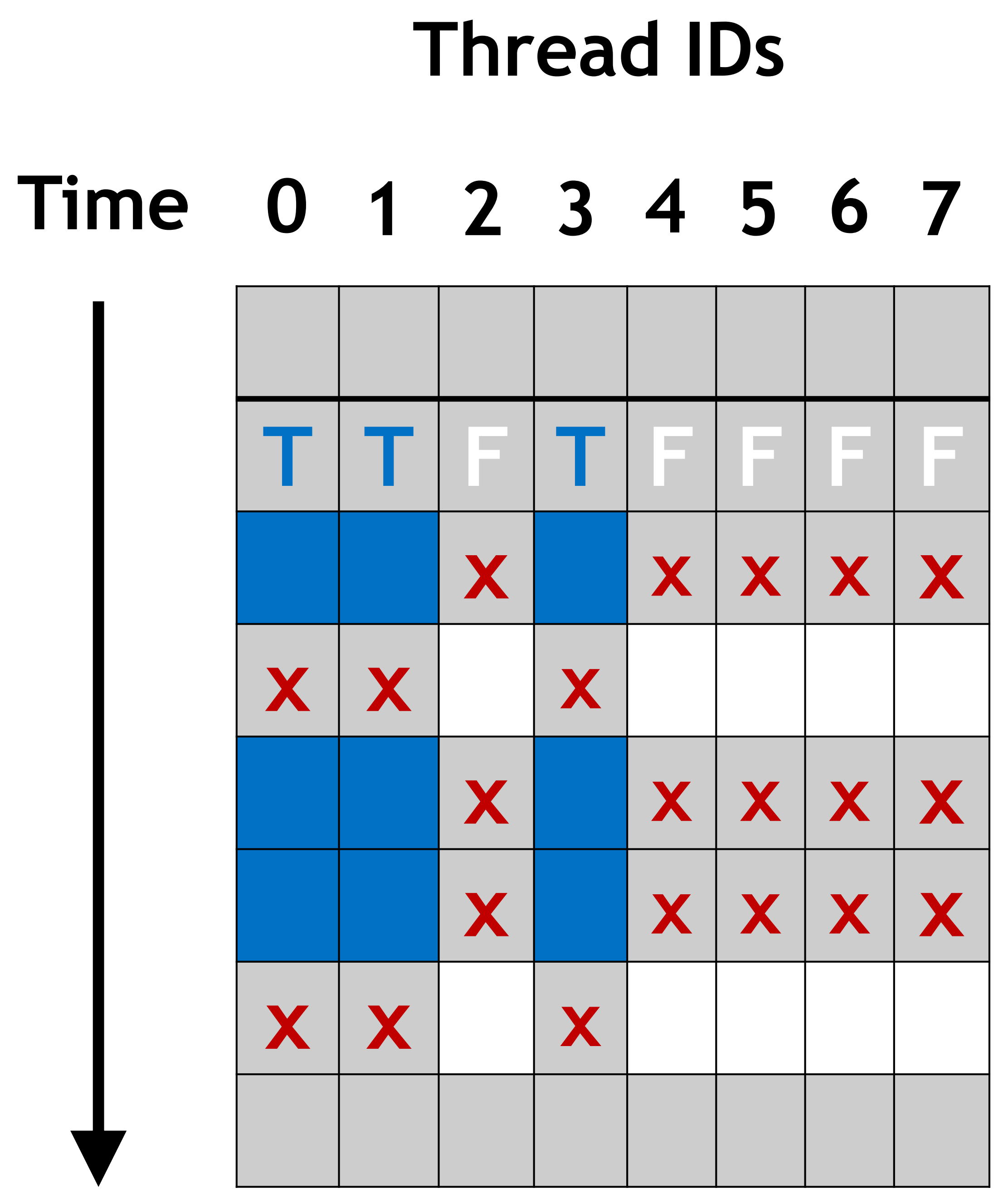
- Akin to a **single-instruction multiple-data (SIMD) array processor** per Flynn's taxonomy combined with **fine-grained multithreading**.
- SIMT architectures expose a large set of hardware threads, which is partitioned into groups called **warps.**
  - Interleave warp execution to hide latencies.
  - Execution context for each warp is kept on-chip for fast interleaving.
- When scheduled, each thread of a warp executes on a given lane of a SIMD functional unit.
- Each SM sub-partition can be thought of as a SIMT engine that creates, manages, schedules, and executes warps of 32 parallel threads.


Warp 0
Warp 1
...
Warp N
Warps available for scheduling
L0 Instruction Cache
Warp Scheduler (32 thread/clk)
Dispatch Unit (32 thread/clk)
Register File (16,384 x 32-bit)
INT32
FP32
FP64
TENSOR CORE 4th GENERATION
LD/ ST
SFU
SIMD back-end

SIMT Architecture

Warp divergence

- If threads in a warp **diverge** via a conditional branch, the warp **executes every branch** path.
- **Full efficiency** is realized when all **32 threads** of a warp agree on their execution path.
    - Aka they are **converged**.



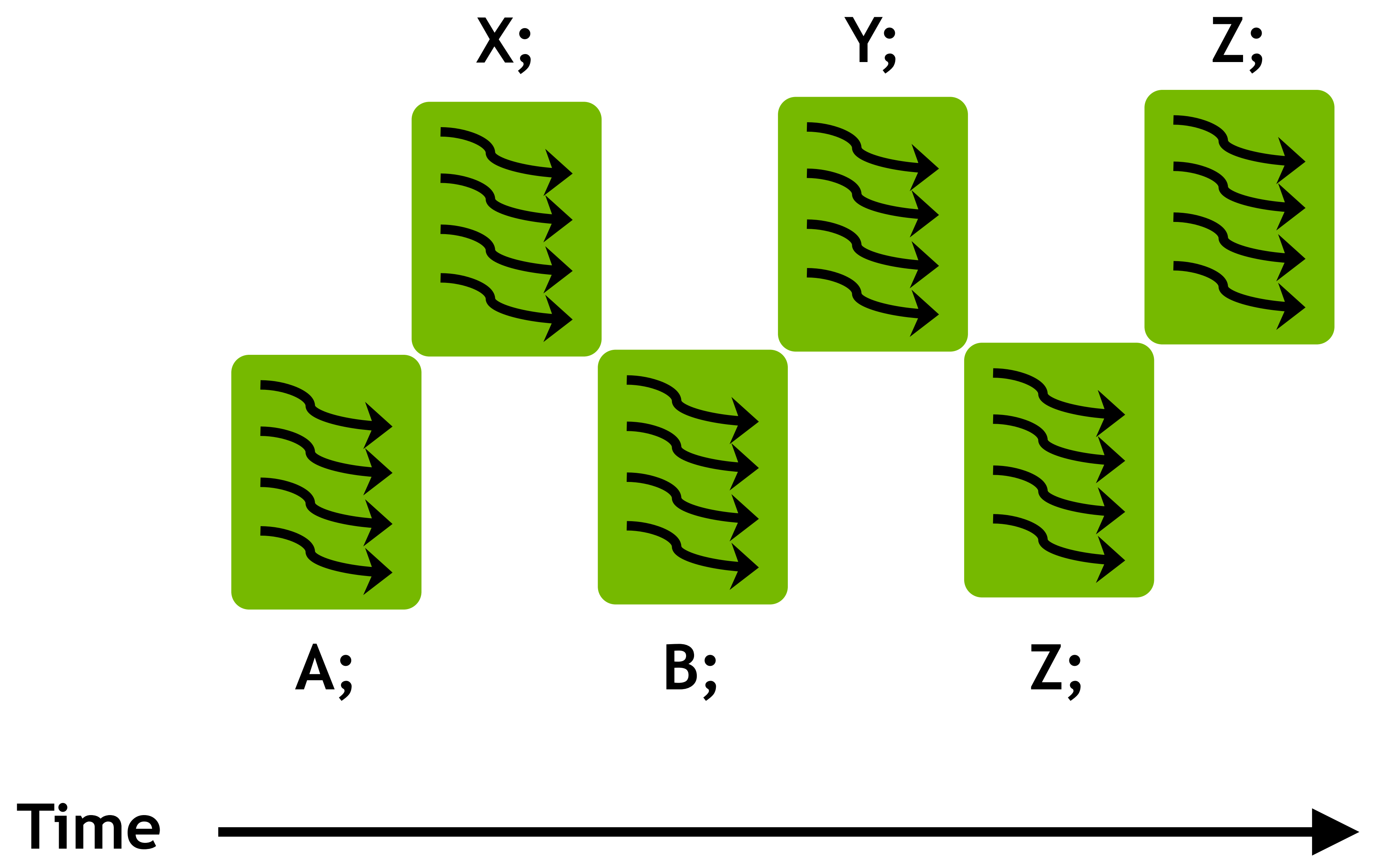
```
if (true) {
   instruction 1
   instruction 2
   instruction 3
} else {
   instruction 4
   instruction 5
}
```

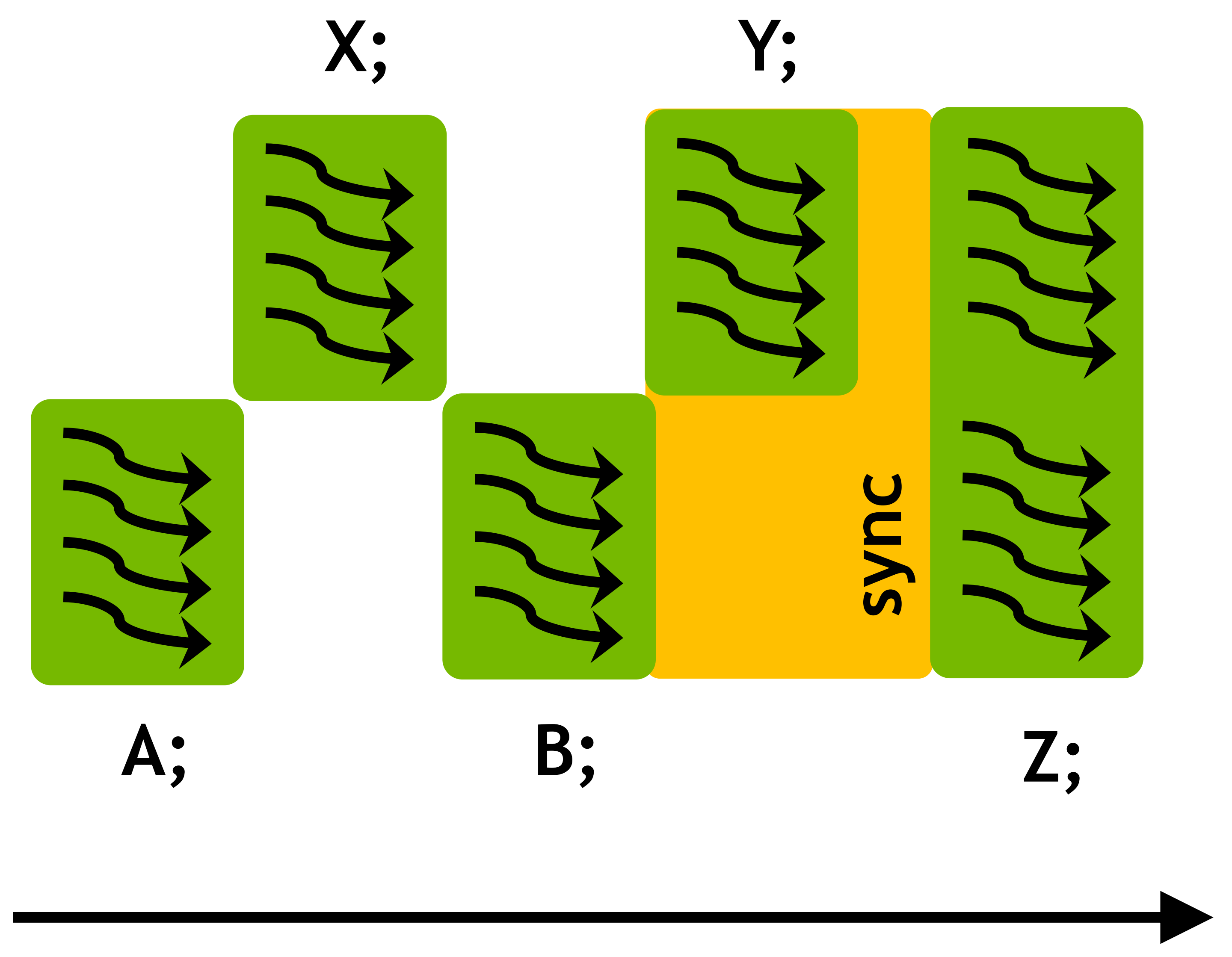
SIMT Architecture

Independent Thread Scheduling

- Individual **threads** in a warp have their own program counter and call stack and are therefore **free to execute independently**.

```
if (thread_id < 4) {
   A;
   B;
} else {
   X;
   Y;
}
Z;
```





Do **not** assume threads in a warp are automatically re-converged after a conditional or at any point!

The compiler **might** sync to enforce re-convergence for better performance.

CUDA Programming Model

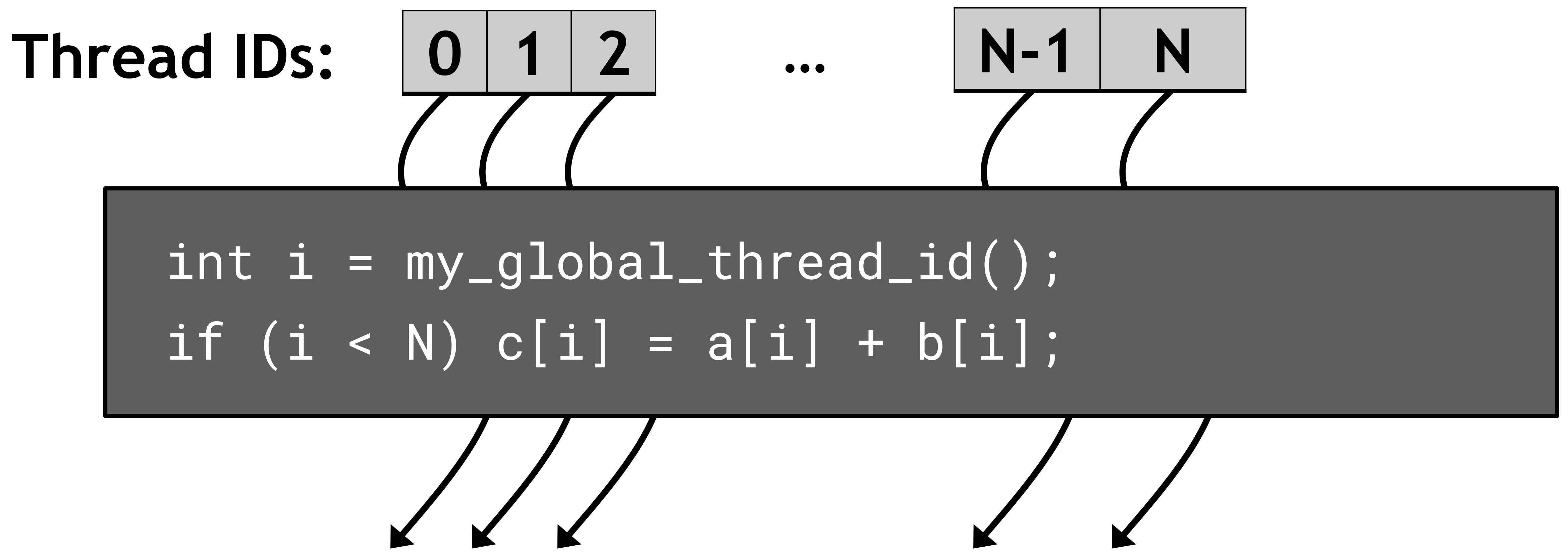
Single-Program Multiple-Data

- SIMT instructions specify the execution of a **single** thread.
- A SIMT kernel is launched on many threads that execute in parallel.
- Threads use their thread index to work on disjoint data or to enable different execution paths.
- Three key software abstractions enable efficient programming through the CUDA programming model:
  - a hierarchy of **thread groups**,
  - **memory spaces**, and
  - **synchronization**.

Single-threaded CPU vector addition

```
for (int i = 0; i < N; i++) {
    c[i] = a[i] + b[i];
}
```

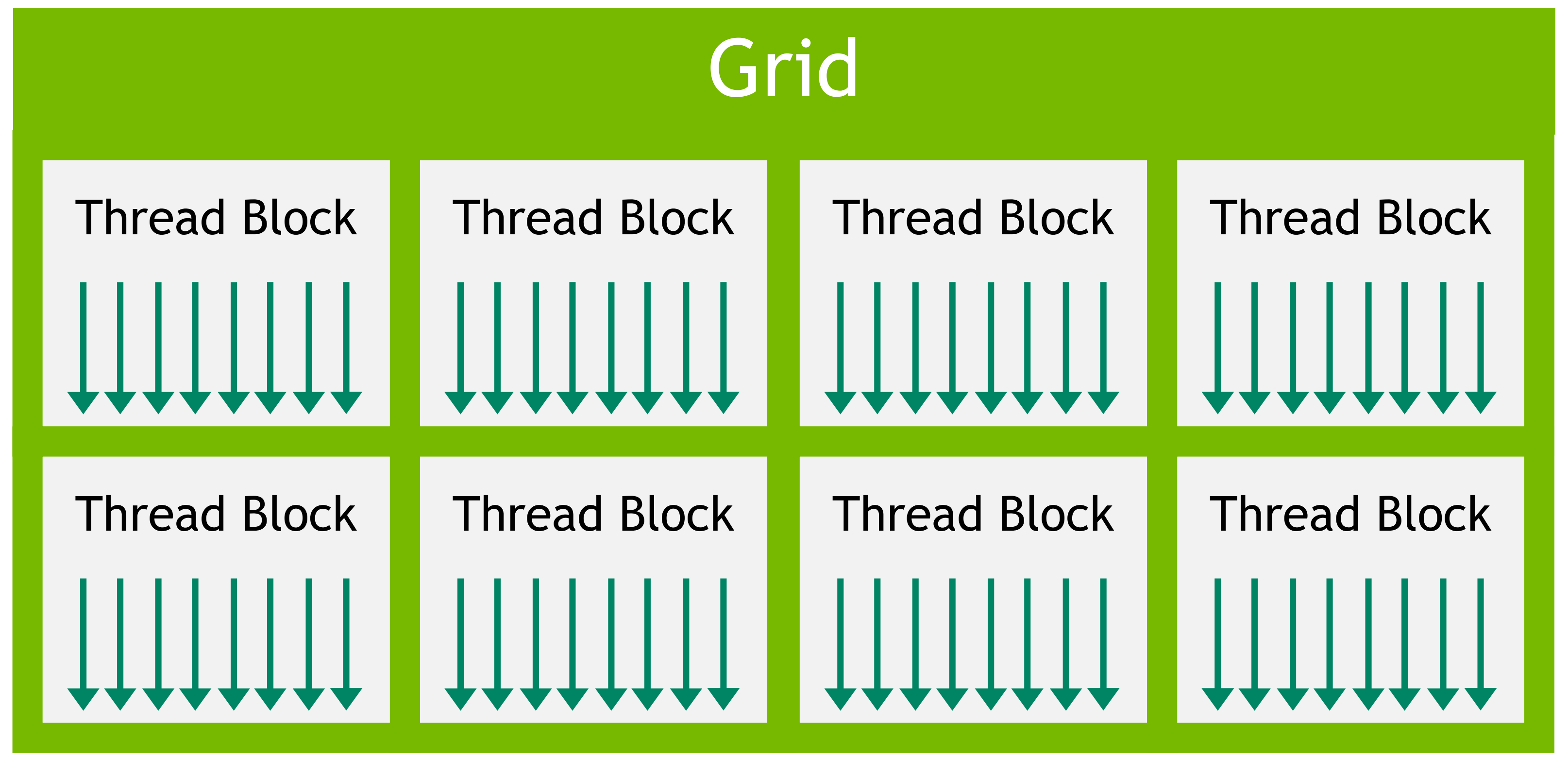
GPU vector addition



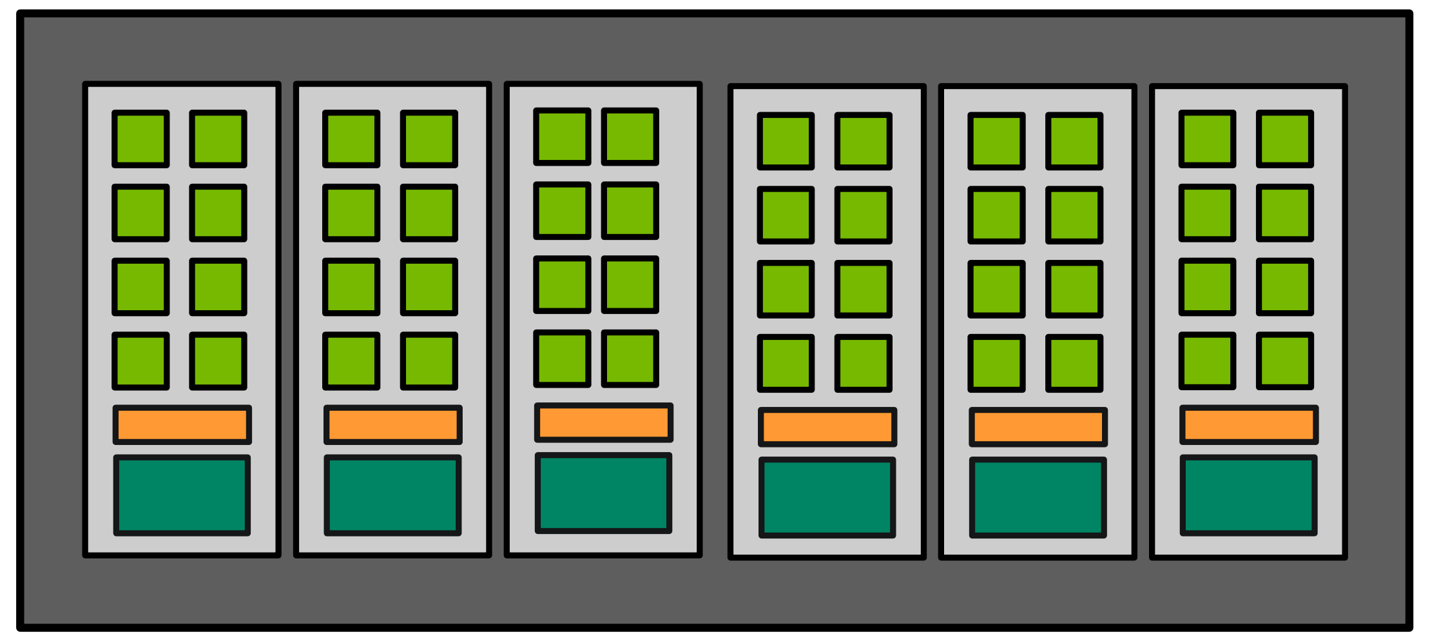
```
int i = my_global_thread_id();
if (i < N) c[i] = a[i] + b[i];
```

Thread Hierarchy

CUDA/Software

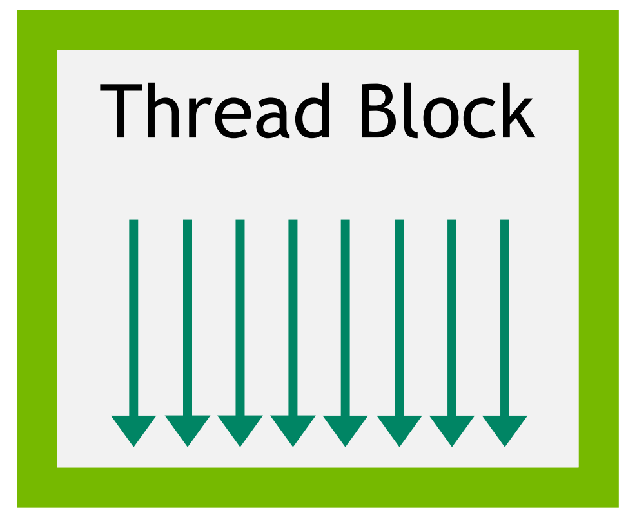


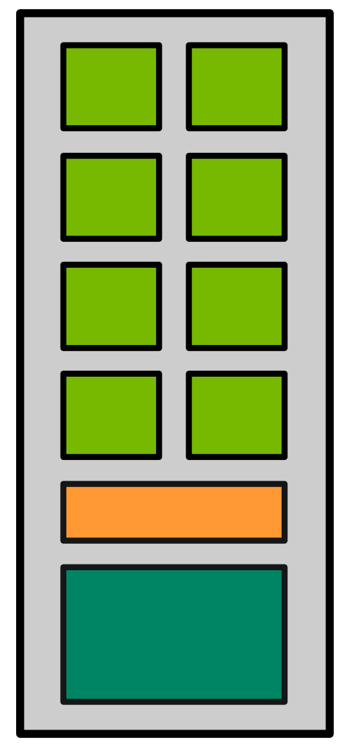
Hardware



Device

- A CUDA kernel is launched on a grid of thread blocks, which are completely independent.

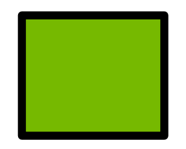




SM

- Thread blocks are executed on SMs.
  - Several concurrent thread blocks can reside on an SM.
  - Thread blocks do not migrate.
  - Each block can be scheduled on any of the available SMs, in any order, concurrently or in series.

Thread



Scalar CUDA core

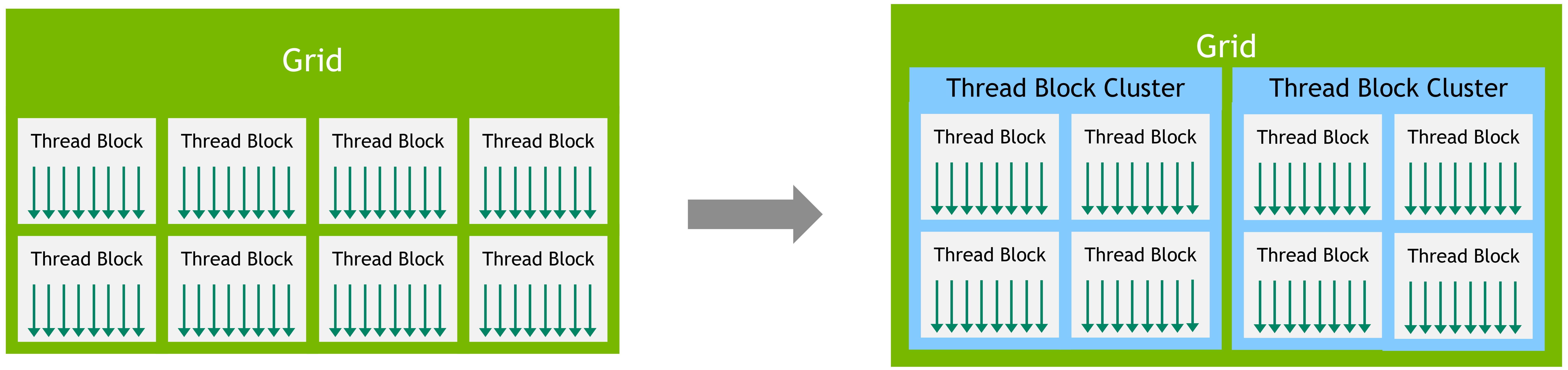
- Individual threads execute on scalar CUDA cores.

Thread Block Clusters

- For Hopper GPUs, CUDA introduced an optional level in the thread hierarchy called **Thread Block Clusters**.

- Thread blocks in a cluster are guaranteed to be concurrently scheduled and enable efficient cooperation and data sharing for threads across multiple SMs.

- For more information on this topic visit GTC session **[S62192]: "Advanced Performance Optimization in CUDA"**.

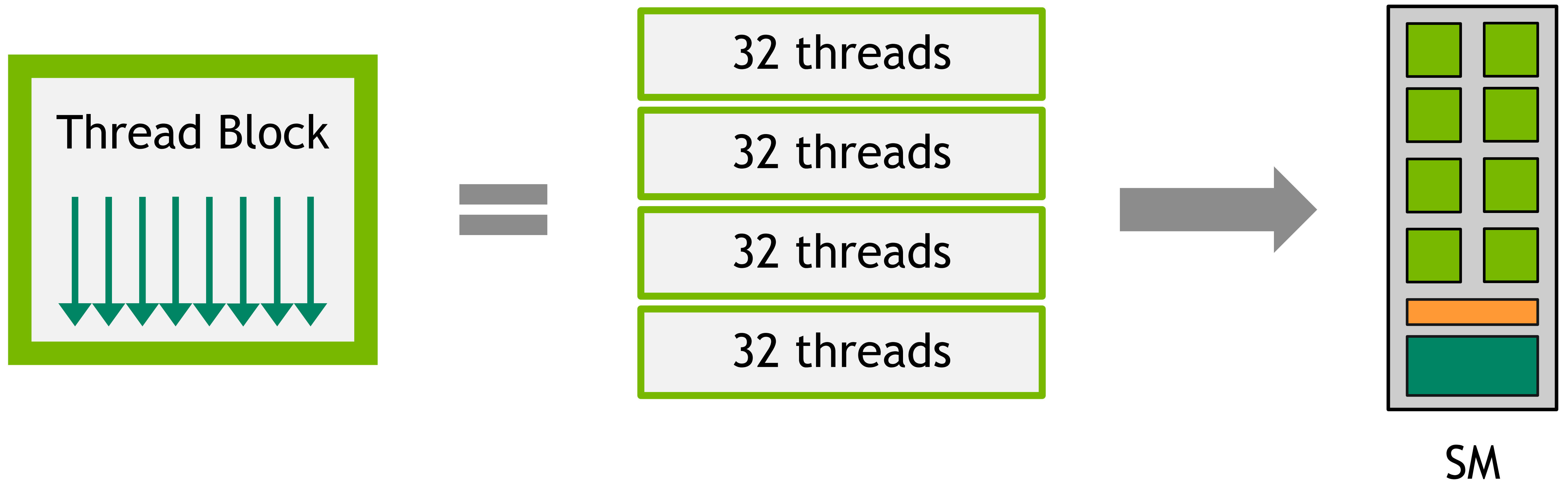


Thread Hierarchy

What about warps?

- At runtime, a block of threads is divided into warps for SIMT execution.
  - The way a block is partitioned into warps is always the same.
    - Each warp contains threads of consecutive, increasing thread IDs with the first warp containing thread 0.

- The total number of warps in a block is defined as:
  - $ceil\left(\frac{threads\ per\ block}{warp\ size}, 1\right)$



Thread Hierarchy

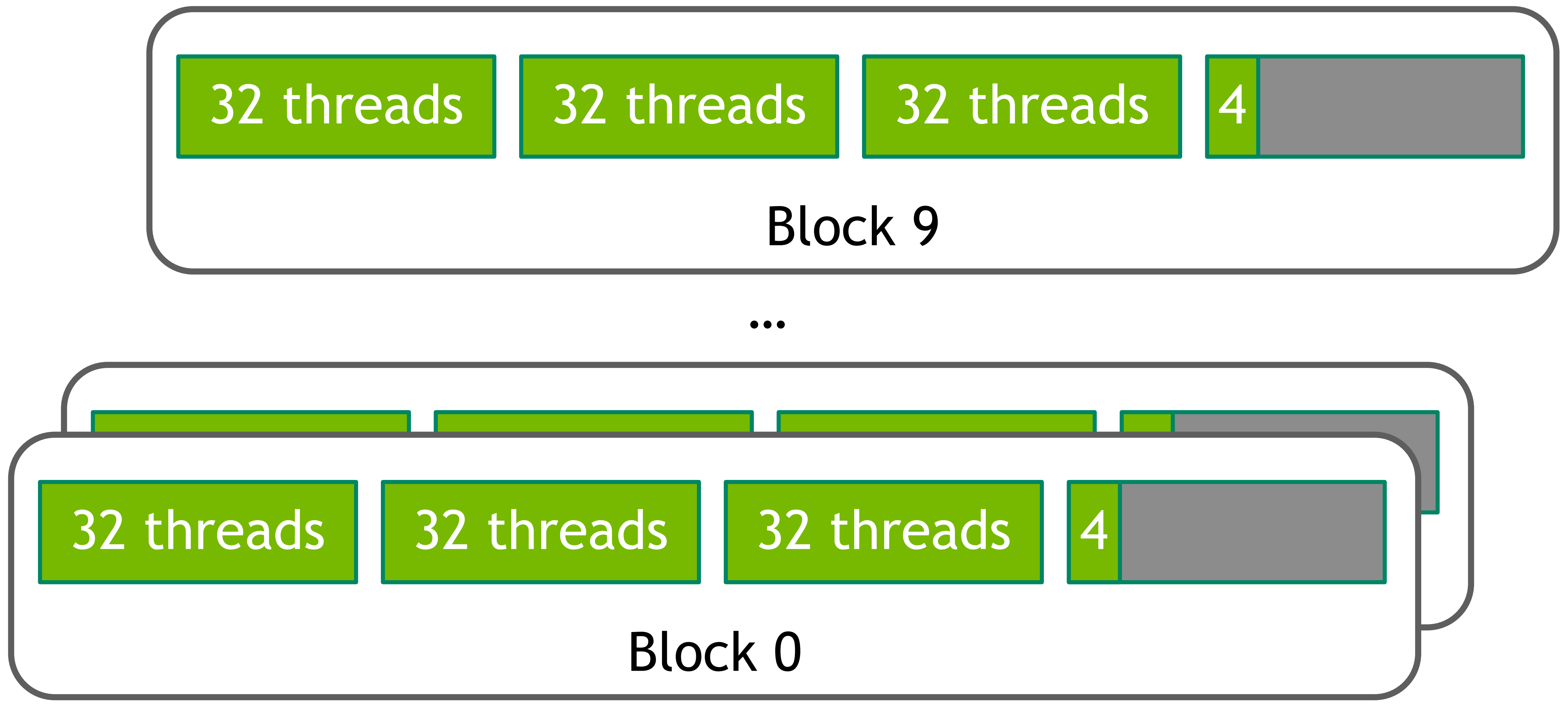
Thread block sizing

- Let's say we want to add two vectors of size N = 1000.
  - **Scenario #1:** 1-D grid of 10 1-D blocks of size 100.
  - **Scenario #2:** 1-D grid of 8 1-D blocks of size 128.
- Which option is better in terms of thread resource utilization?

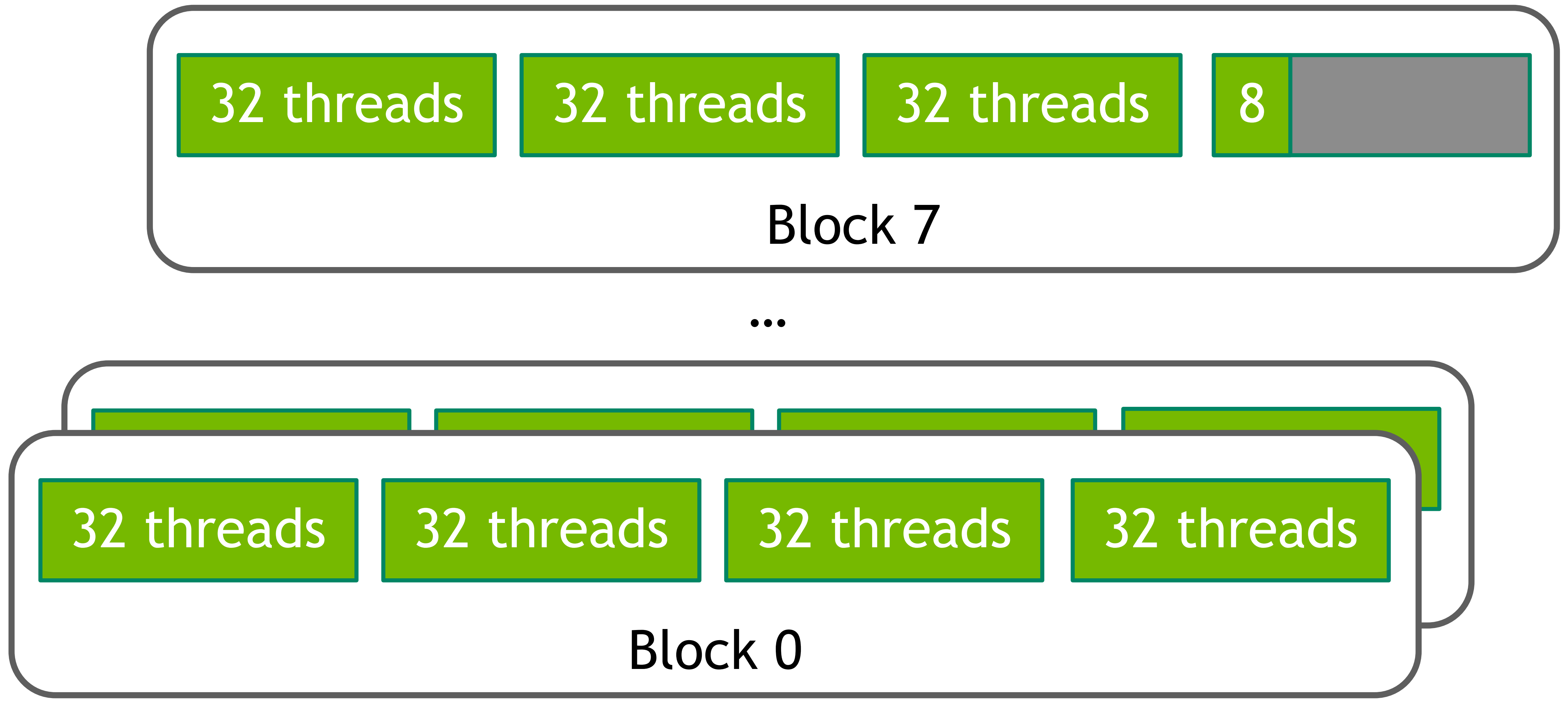
Thread Hierarchy

Thread block sizing

- Let's say we want to add two vectors of size N = 1000.
  - **Scenario #1:** 1-D grid of 10 1-D blocks of size 100.
  - **Scenario #2:** 1-D grid of 8 1-D blocks of size 128.
- Which option is better in terms of thread resource utilization?

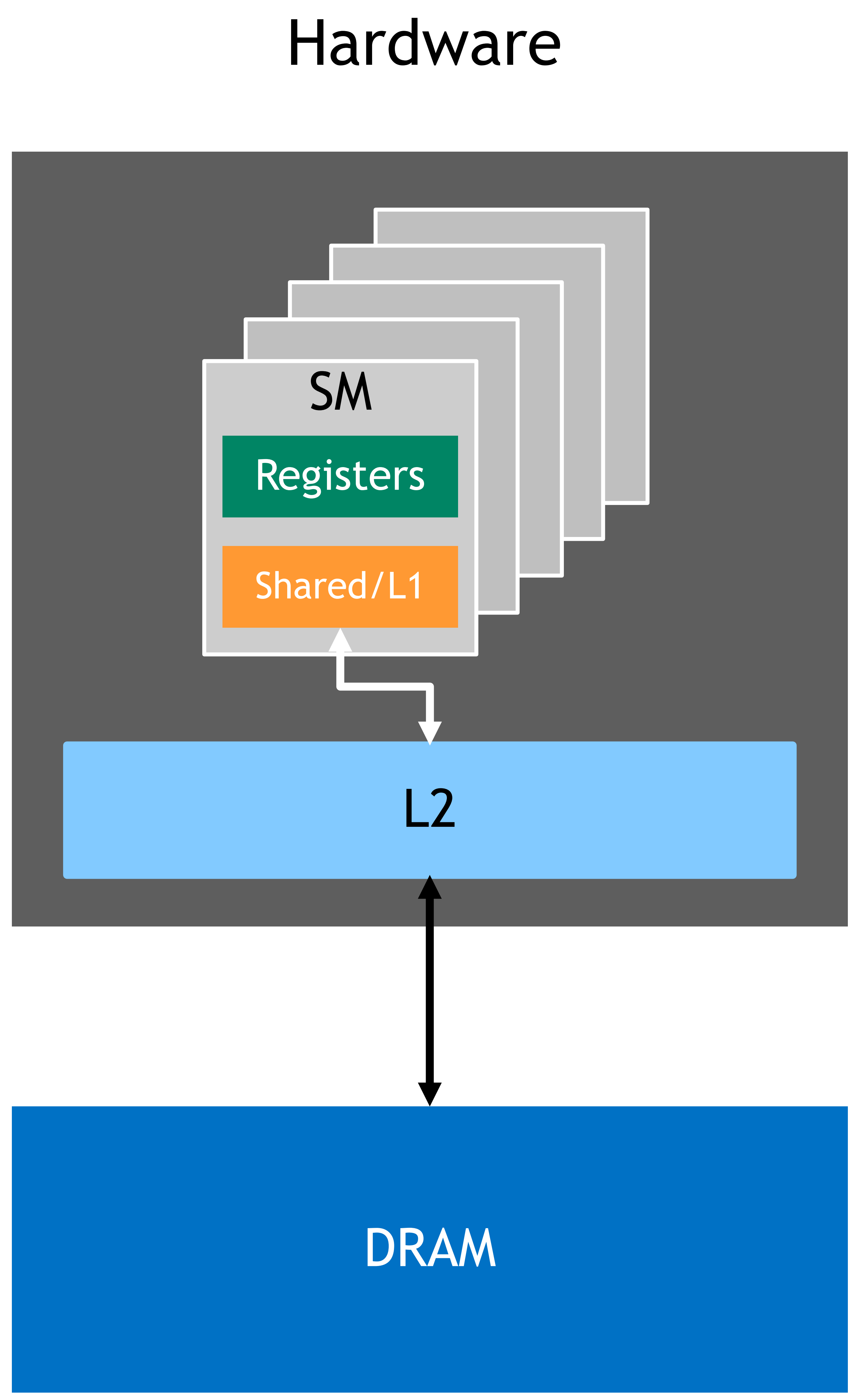


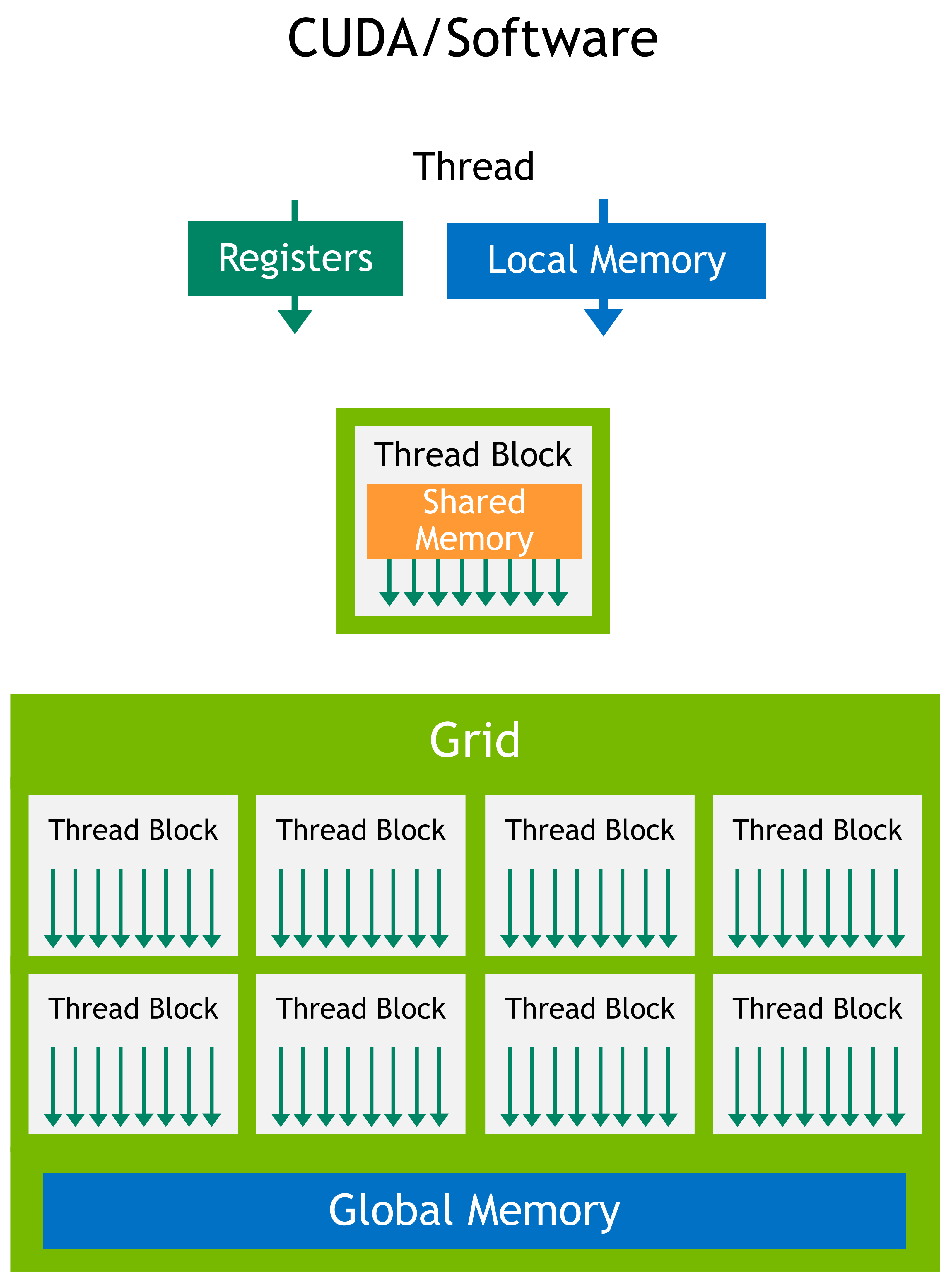
Scenario #1:
3 full warps and 1 warp with 4 active threads per block
Average thread utilization = 78.125%



Scenario #2:
4 full warps per block, except last block
Average thread utilization = 97.656%

Memory Hierarchy

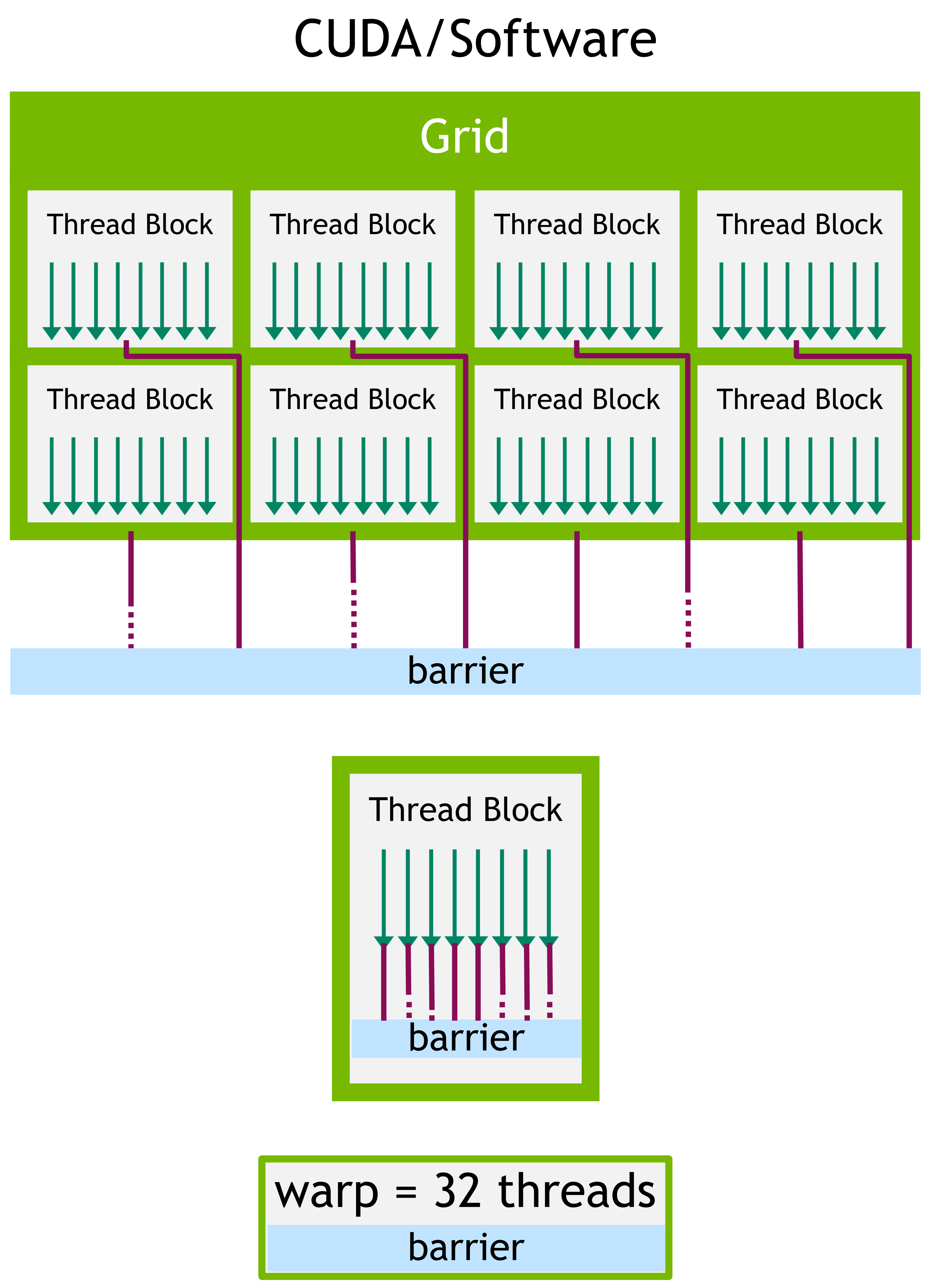




- Per-thread **registers**.
  - Lowest possible latency.
- Per-thread **local** memory.
  - Private storage.
  - Slowest access.
- Per-block **shared** memory.
  - Visible by all threads in a block.
  - Can be used to exchange data between threads in a thread block.
  - Very fast access.
- **Global** memory.
  - Visible by all threads in a grid.
  - Slowest access.

Synchronization

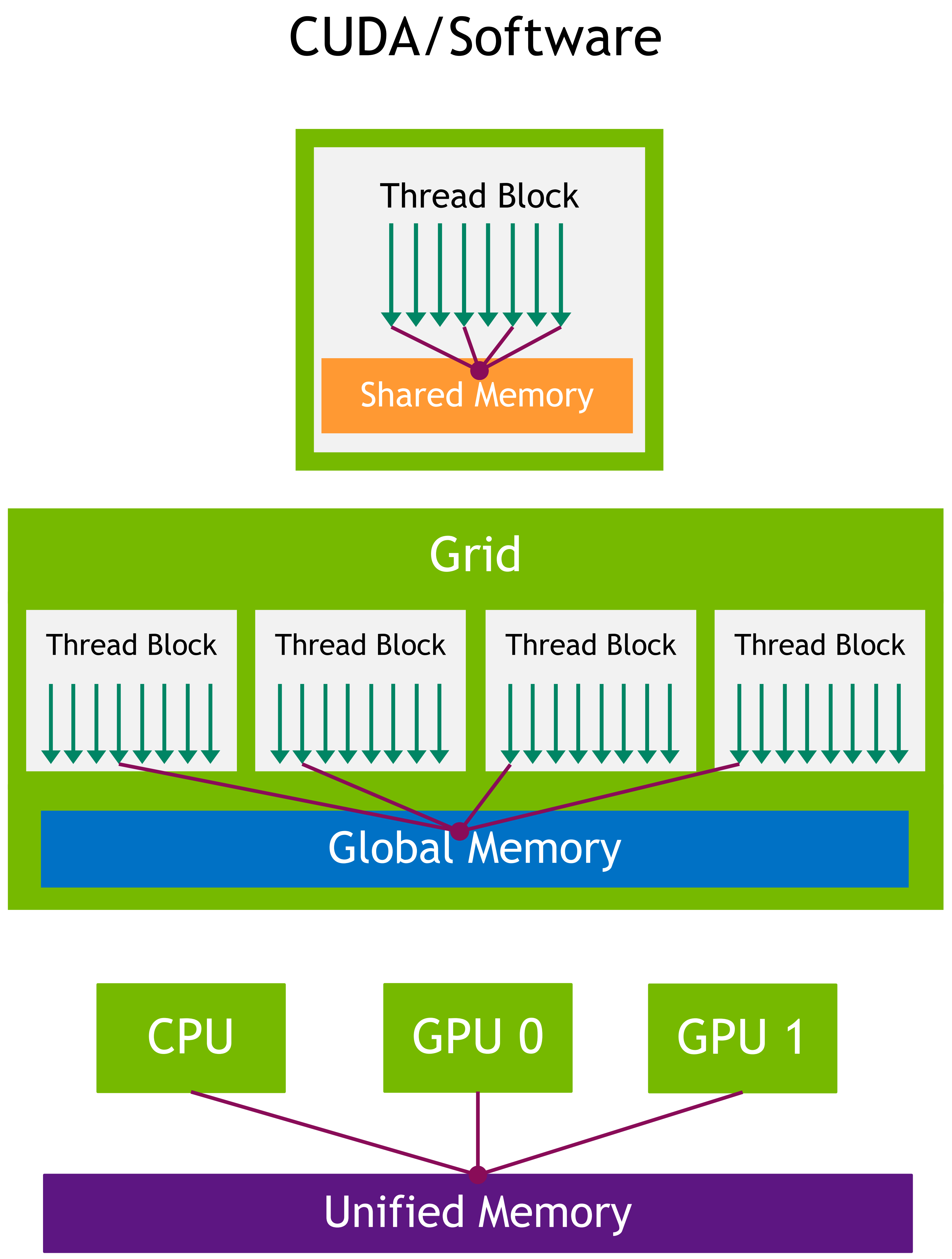
Barriers



- **Grid** boundary.
  - Kernel completion.
  - `grid_group::sync()` via Cooperative Groups API
    - Requires the kernel to be launched via the `cudaLaunchCooperativeKernel()` API
    - **Slow!** Avoid unless necessary.
- **Thread-block** boundary.
  - `__syncthreads()`
  - `thread_block::sync()` via Cooperative Groups API
  - **Fast!** The most common synchronization level.
- **Warp** or **sub-warp** boundary.
  - `__syncwarp()`
  - `coalesced_group::sync()` via Cooperative Groups API
  - **Very fast!**

Atomics

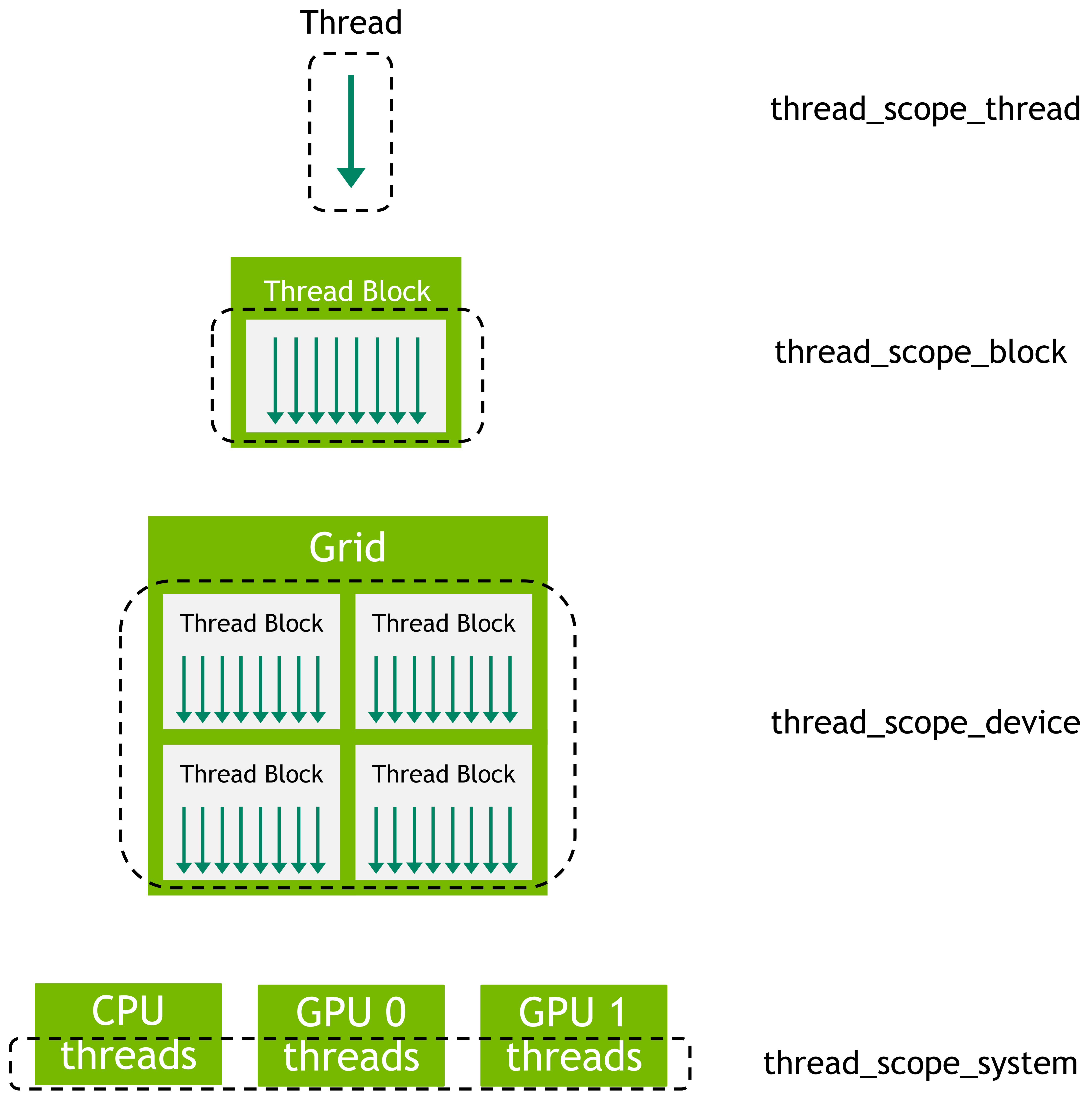
Memory spaces



- Read-modify-write operations on 16–, 32-, 64- or 128-bit words.
  - Available as CUDA primitives or C++ atomics through `libcu++` extended API.
- **Shared memory** atomics.
- **Global memory** atomics.
  - Facilitated by special hardware in the L2 cache.
- **Unified memory** atomics.

Thread Scopes

- To account for **non-uniform** thread synchronization costs, CUDA has introduced the notion of **thread scopes**.
- A thread scope specifies **which threads can communicate** with each other using a primitive such as an atomic or a barrier.
- Thread scopes are exposed to the programmer in 3 ways:
  - PTX
  - CUDA Math API
  - CUDA C++
- Always use the narrowest scope that ensures correctness of your application.
- More on thread scopes in the GTC session **[S62192]: “Advanced Performance Optimization in CUDA”**.



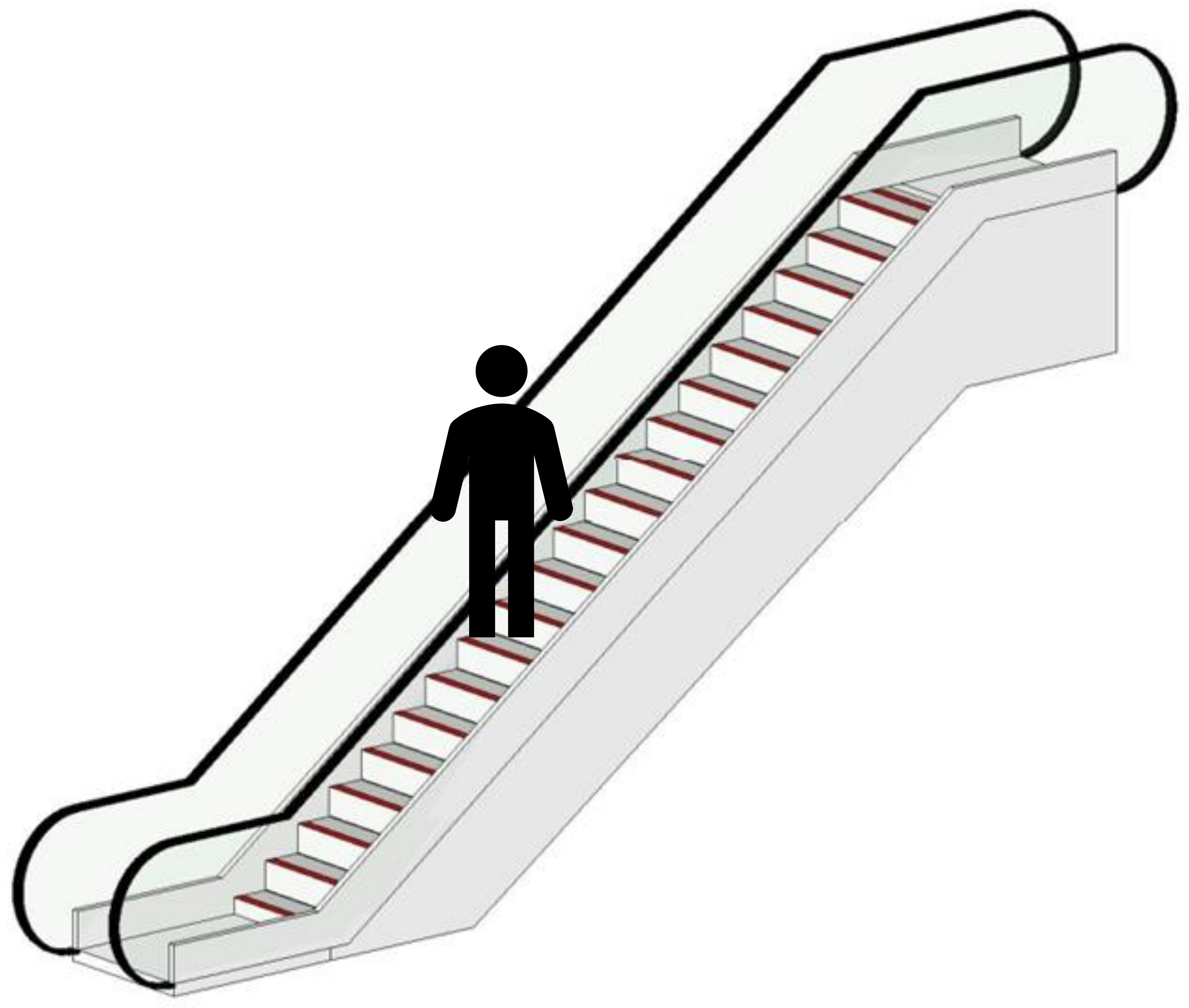
Fundamental Performance Optimizations

Little's Law

For escalators

Our escalator parameters:

- 1 person per step
- A step arrives every 2 seconds
  - **Bandwidth**: 0.5 person/s
- 20 steps tall
  - **Latency** = 40 seconds



One person in flight?

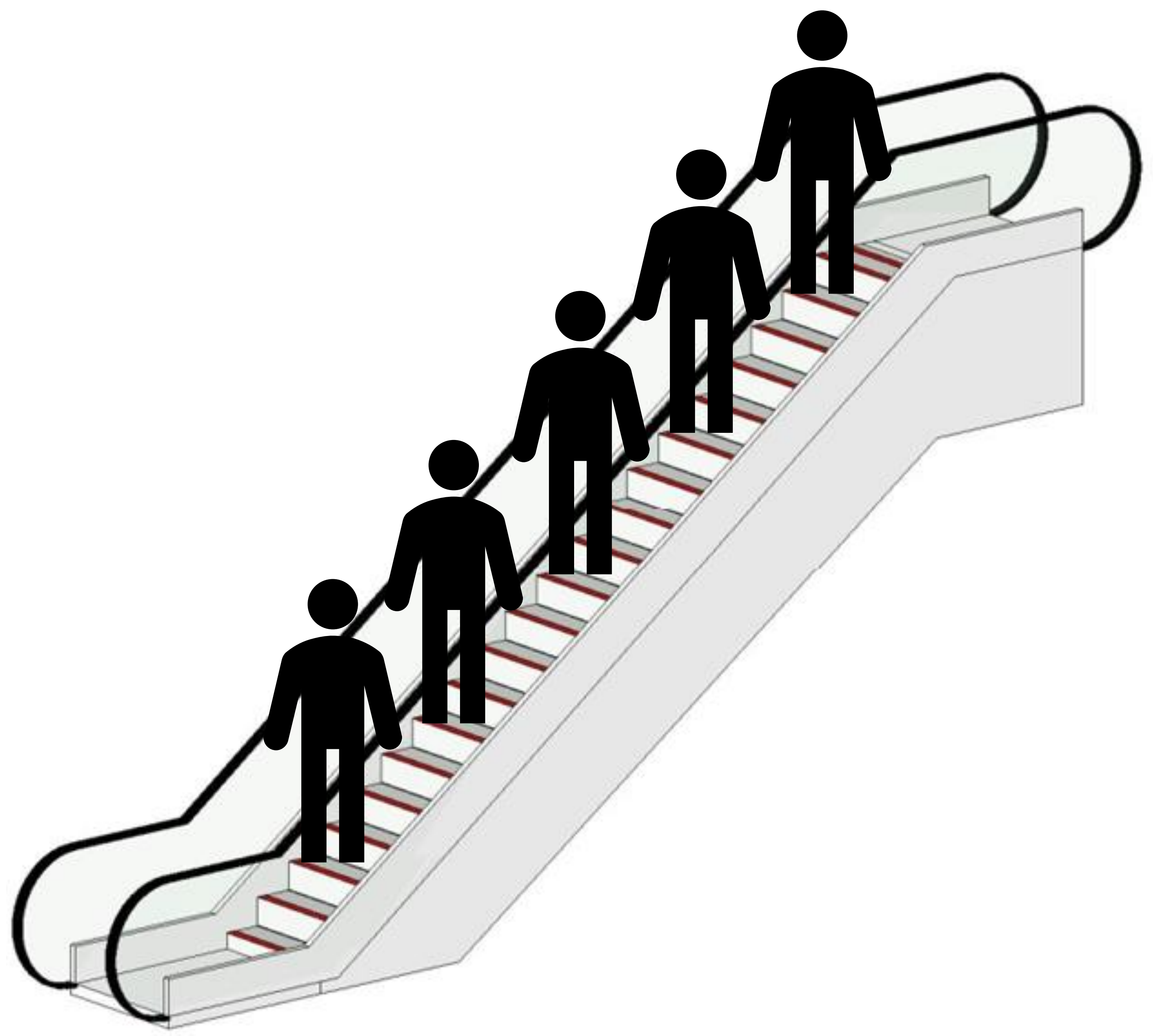
Throughput = 0.025 person/s

Little's Law

For escalators

Our escalator parameters:

- 1 person per step
- A step arrives every 2 seconds
  - **Bandwidth**: 0.5 person/s
- 20 steps tall
  - **Latency** = 40 seconds



How many persons do we need in-flight to saturate bandwidth?

Concurrency = Bandwidth x Latency
= 0.5 persons/s x 40 s
= 20 persons

Little’s Law

For GPUs

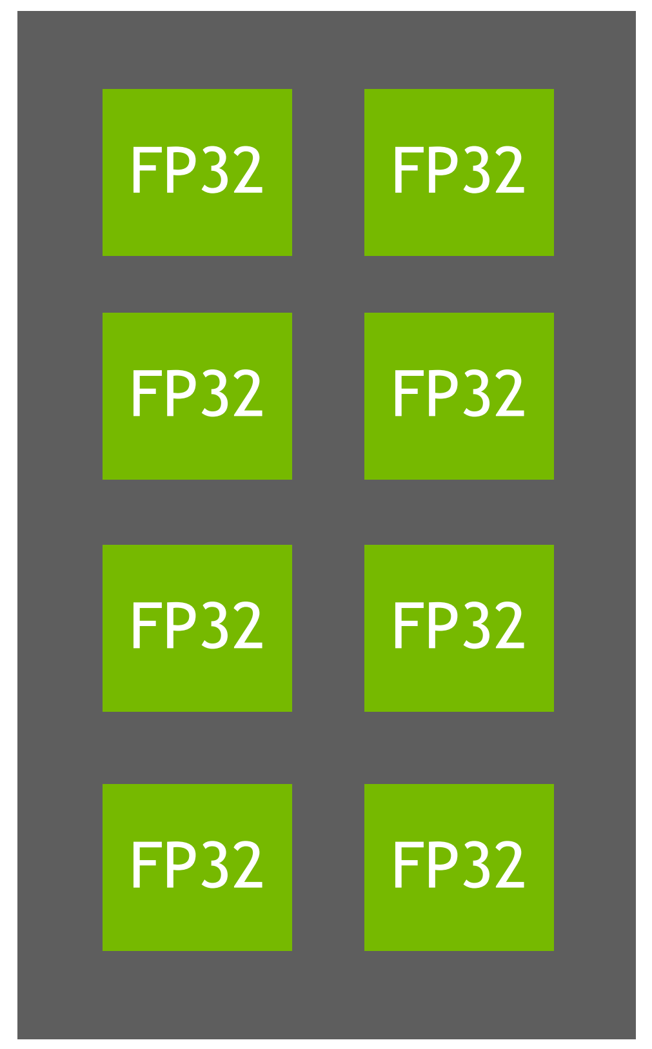
- How to maximize performance?

    1. **Saturate compute units.**
    2. **Saturate memory bandwidth.**

- Need to hide the corresponding latencies to achieve this.

    - Compute latencies.
    - Memory access latencies.

- Latencies can be hidden by having more instructions in flight.



FP32 Latency = 24 cycles
8 FP32 ops per cycle

Concurrency = Bandwidth x Latency =
8 x 24 operations in-flight

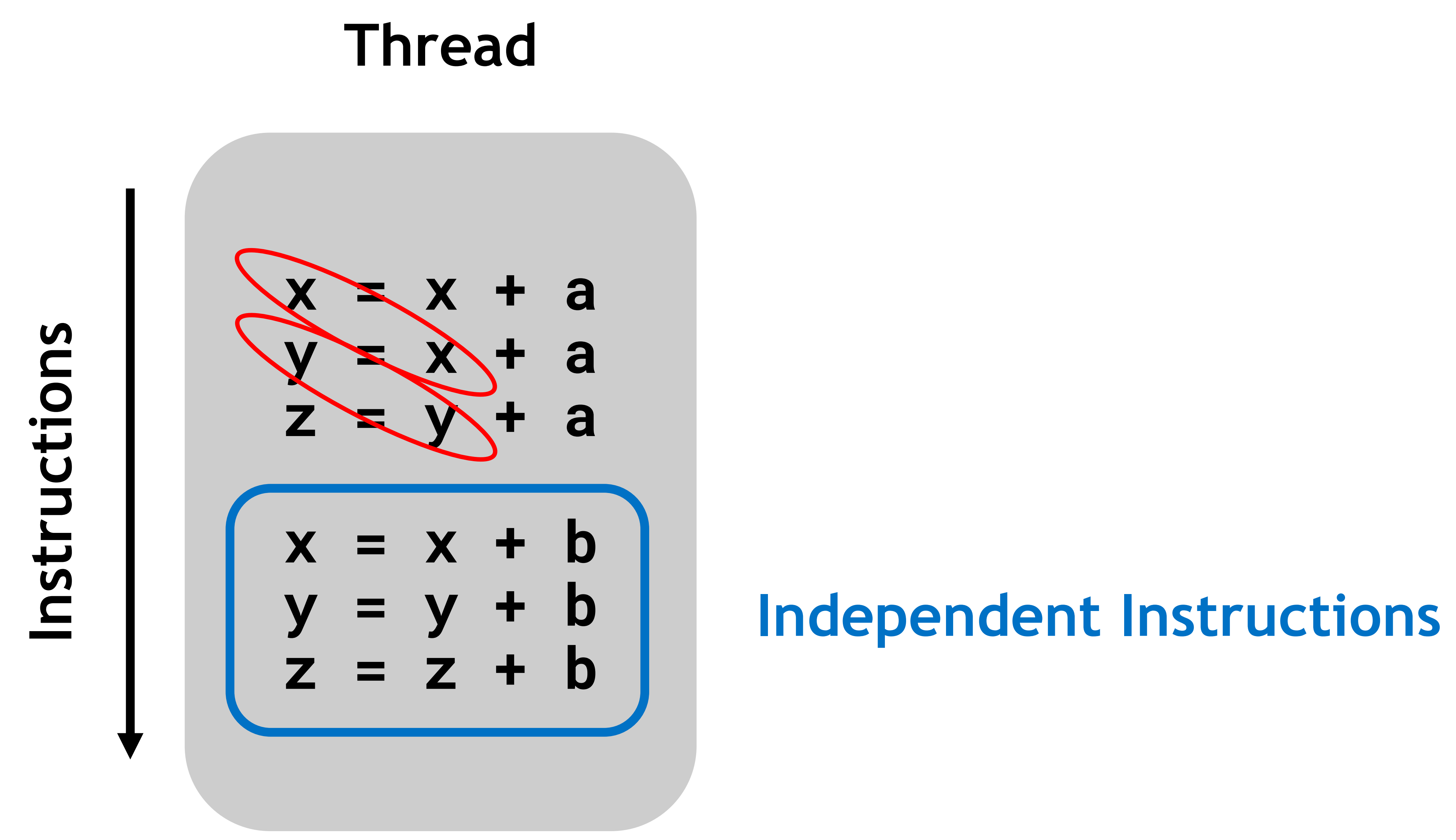
Hiding Latencies

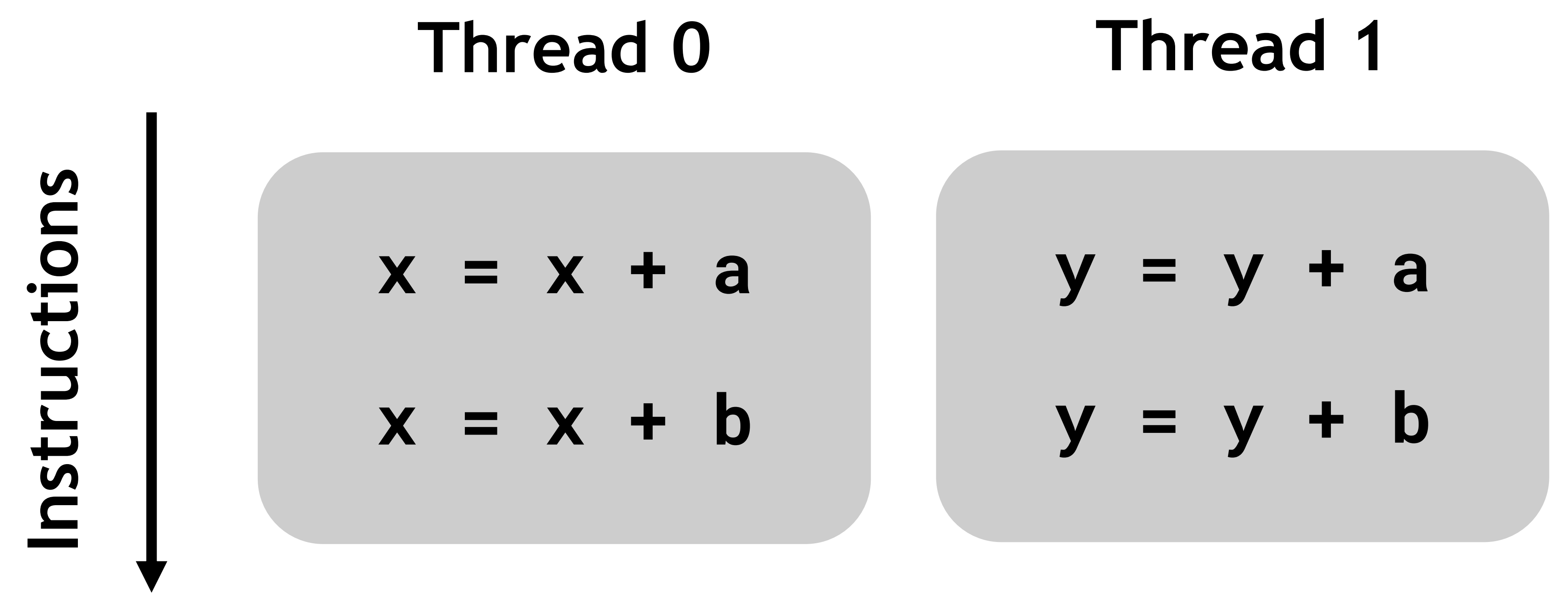
Increasing in-flight instructions

- **Two ways to increase in-flight instructions:**

1. Improve **Instruction-Level Parallelism** (ILP).
   - Higher ILP -> more independent instructions per thread.

2. Improve **Thread-Level Parallelism** (TLP).
   - Higher TLP -> more threads -> more independent instructions per kernel.





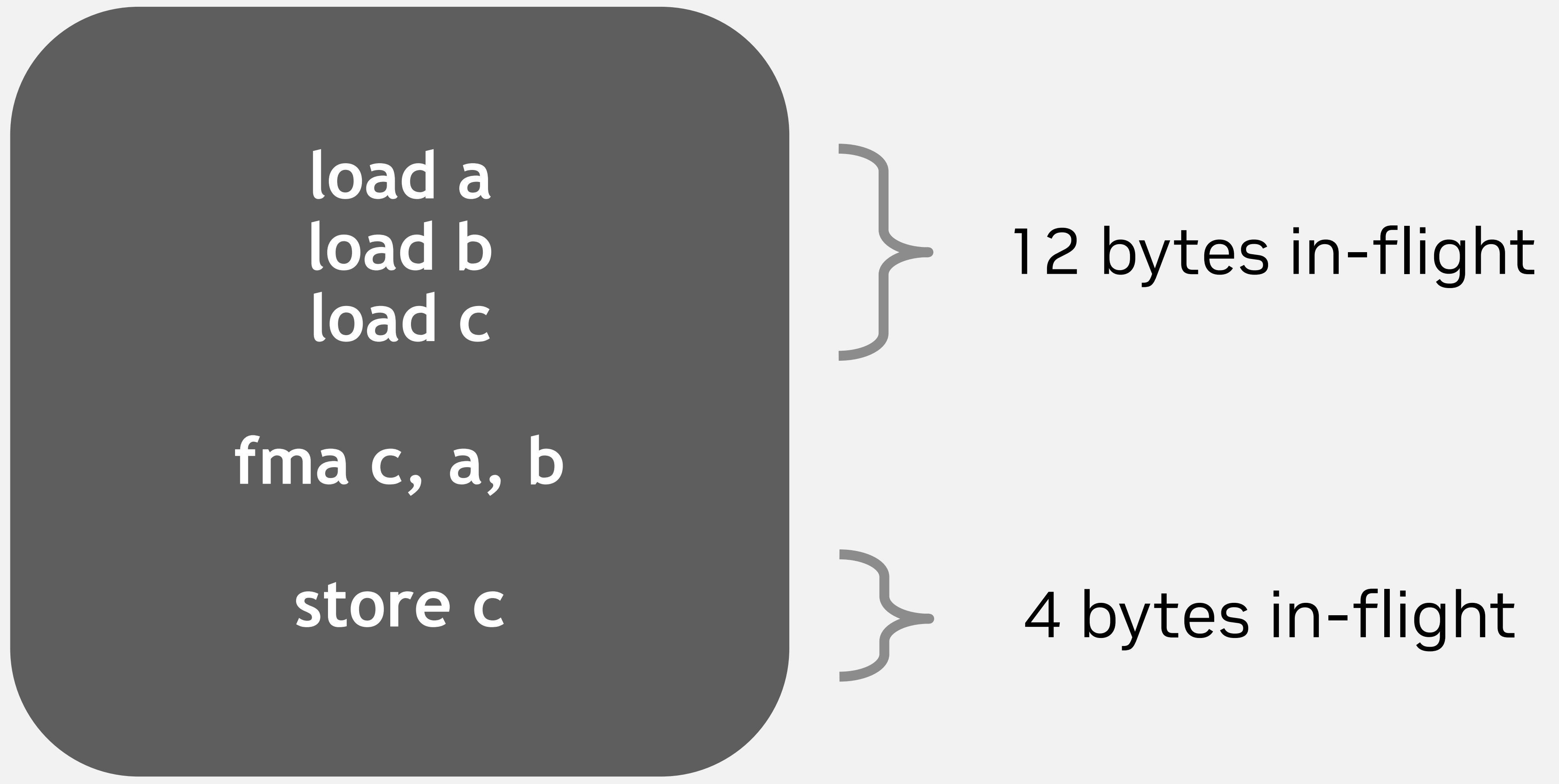
Instruction Issue

- Assumptions
  - LDG/STG
    - Dependent Issue Rate: 1000 cycles
    - Issue Rate: 1 cycle
  - FP32 pipeline
    - Dependent Issue Rate: 4 cycles
    - Issue Rate: 2 cycles
  - 1 available warp per scheduler

Cycle	N	N+1	N+2		N+1002		N+1006
	LDG	LDG	LDG	(stall)	FFMA	(stall)	STG

Total cycles = 1006

```
__global__
void kernel(const float * __restrict__ a,
            const float * __restrict__ b,
            float * __restrict__ c)
{
  int idx = blockIdx.x * blockDim.x + threadIdx.x;

  c[idx] += a[idx] * b[idx];
}
```



```
load a
load b
load c

fma c, a, b

store c
```

Increasing ILP

Computing 2 elements per thread – version #1

- Every thread computes 2 elements using a **grid stride.**

Cycle

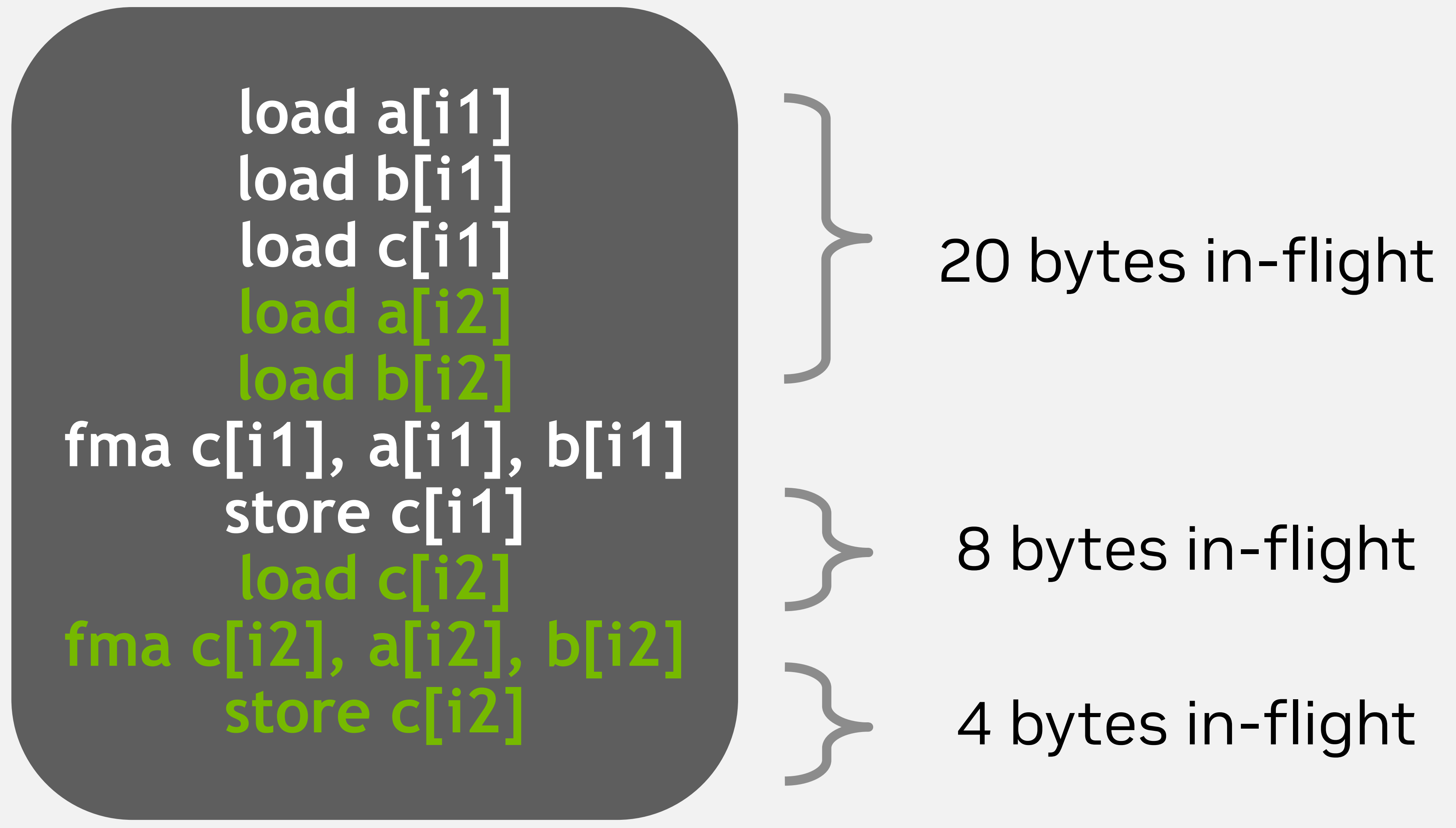
N	N+1	N+2	N+3	N+4		N+1002		N+1006
LDG	LDG	LDG	LDG	LDG	(stall)	FFMA	(stall)	STG

N+1007		N+2007		N+2011
LDG	(stall)	FFMA	(stall)	STG

Total cycles = 2011

2x the amount of work in 2x more cycles!

```
__global__
void kernel(const float * __restrict__ a,
            const float * __restrict__ b,
            float * __restrict__ c)
{
  int tid = blockIdx.x * blockDim.x + threadIdx.x;
  int stride = blockDim.x * gridDim.x;

  #pragma unroll 2
  for (int i = 0; i < 2; i++) {
    const int idx = tid + i * stride;
    c[idx] += a[idx] * b[idx];
  }
}
```



```
load a[i1]
load b[i1]
load c[i1]
load a[i2]
load b[i2]
fma c[i1], a[i1], b[i1]
store c[i1]
load c[i2]
fma c[i2], a[i2], b[i2]
store c[i2]
```

Increasing ILP

Computing 2 elements per thread – version #2

- Every thread computes 2 elements using a **constant block stride.**

Cycle

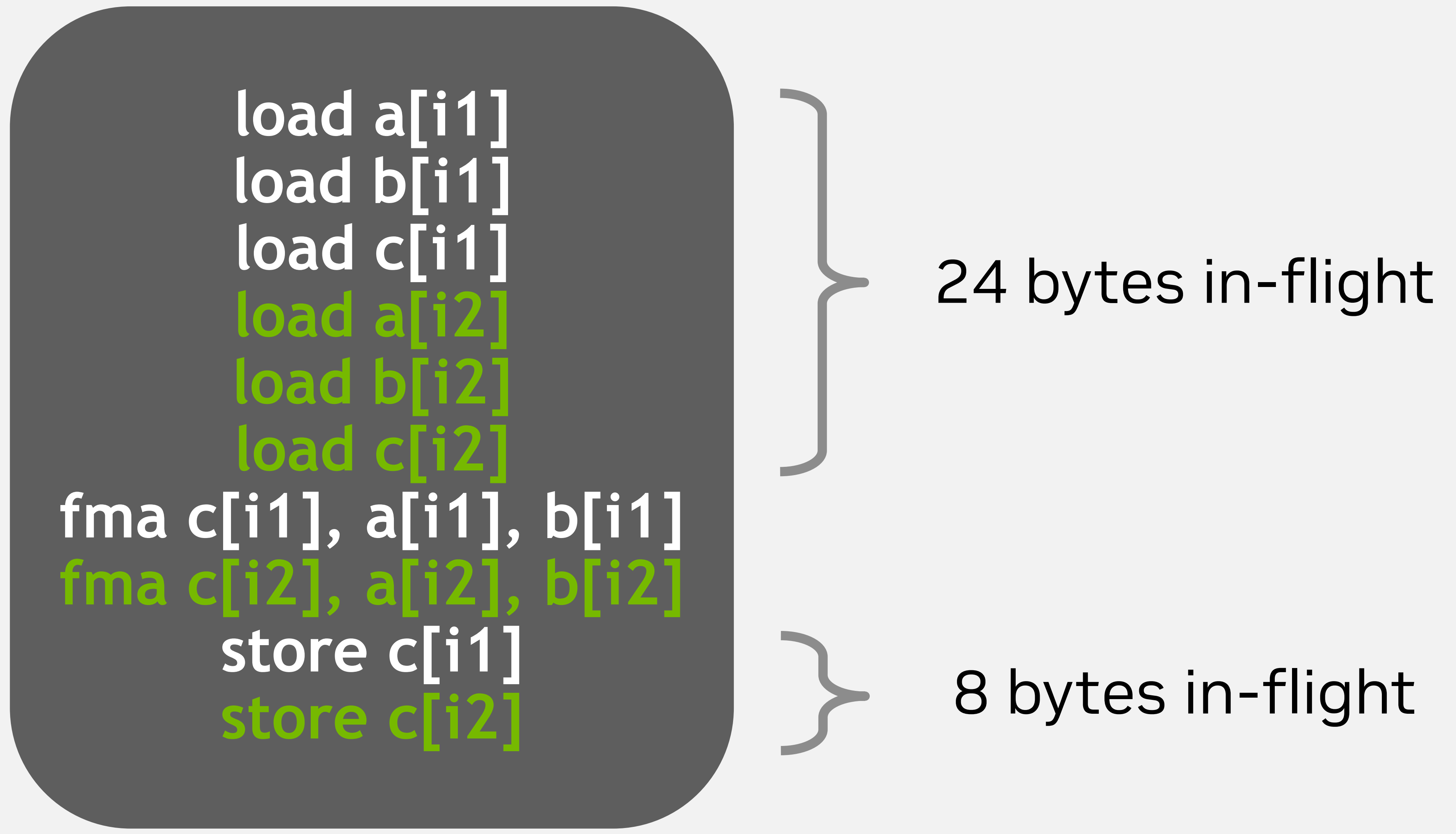
N	N+1	N+2	N+3	N+4	N+5		N+1002	
LDG	LDG	LDG	LDG	LDG	LDG	(stall)	FFMA	(stall)

N+1004		N+1006		N+1008
FFMA	(stall)	STG	(stall)	STG

Total cycles = 1008

2x the amount of work in the ~same number of cycles!

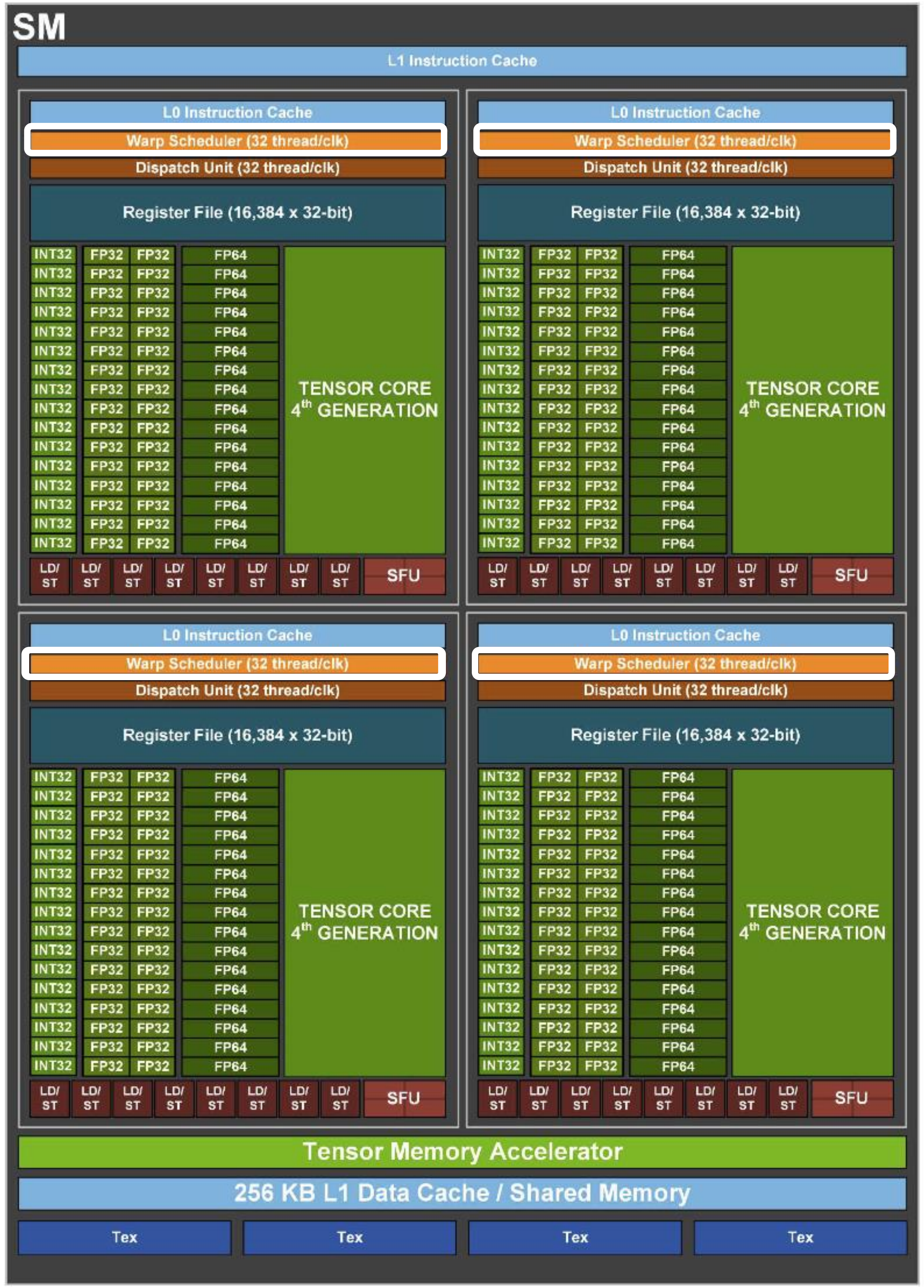
```
#define THREAD_BLOCK_DIM 128

__global__
void kernel(const float * __restrict__ a,
            const float * __restrict__ b,
            float * __restrict__ c)
{
  int tid = blockIdx.x * blockDim.x + threadIdx.x;
  int off = 2 * THREAD_BLOCK_DIM * blockIdx.x + threadIdx.x;

  #pragma unroll 2
  for (int i = 0; i < 2; i++) {
    const int idx = off + i * THREAD_BLOCK_DIM;
    c[idx] += a[idx] * b[idx];
  }
}
```


```
load a[i1]
load b[i1]
load c[i1]
load a[i2]
load b[i2]
load c[i2]
fma c[i1], a[i1], b[i1]
fma c[i2], a[i2], b[i2]
store c[i1]
store c[i2]
```

Warp Scheduling

Hopper SM

- 4 warp schedulers per SM.
- Each scheduler manages a pool of warps.
  - Hopper: 16 warp slots per scheduler.
- Each scheduler can issue 1 warp per cycle.


SM
L1 Instruction Cache
L0 Instruction Cache
Warp Scheduler (32 thread/clk)
Dispatch Unit (32 thread/clk)
Register File (16,384 x 32-bit)
INT32
FP32
FP64
TENSOR CORE 4th GENERATION
LD/ ST
SFU
Tensor Memory Accelerator
256 KB L1 Data Cache / Shared Memory
Tex

Warp Scheduling

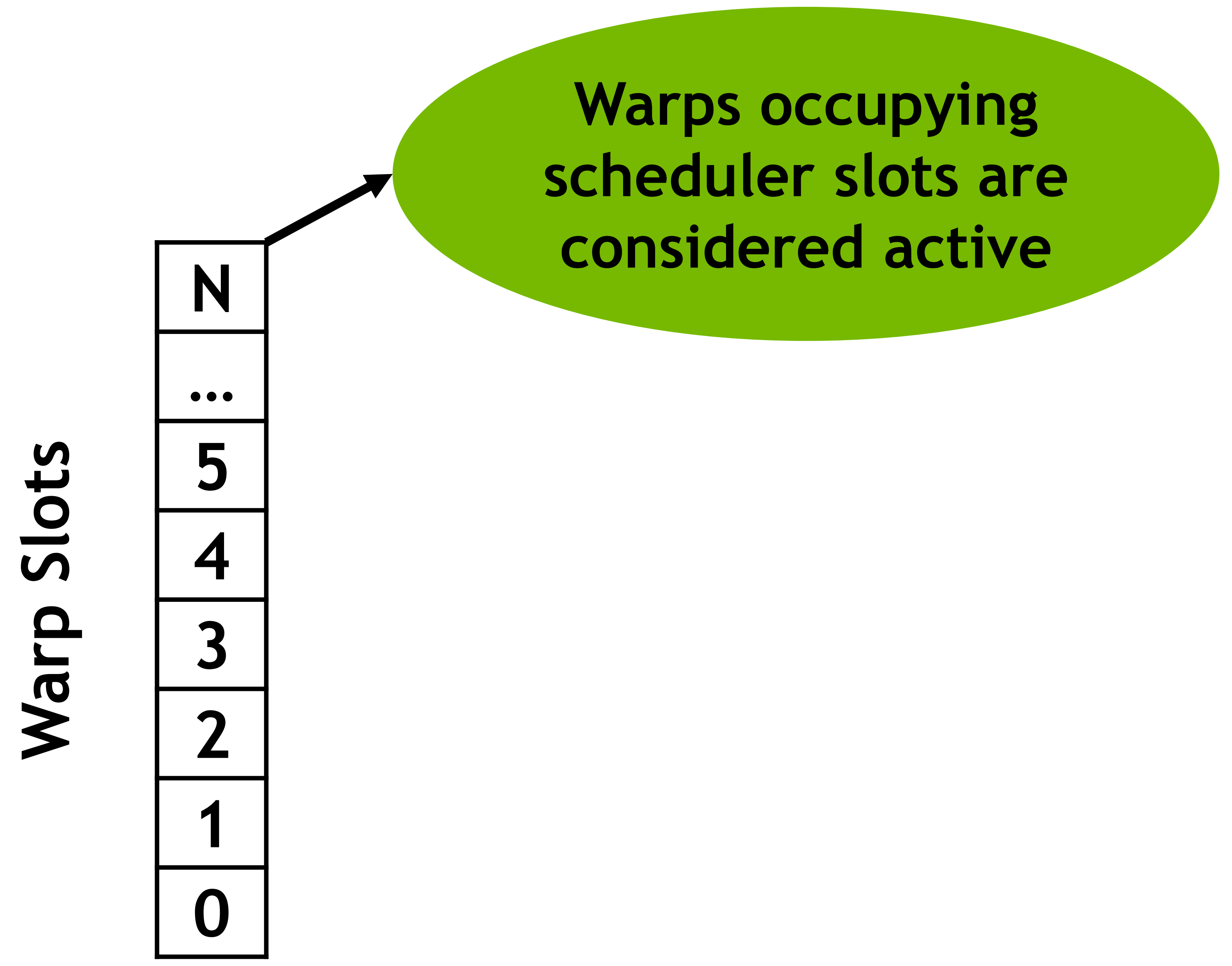
Mental model

Active Warp States:

🟩 **Stalled**
Waiting on:
an instruction fetch,
a memory dependency,
an execution dependency, or
a synchronization barrier.

⬜ **Eligible**
Ready to issue an instruction.

🟩 **Selected**
Eligible that is selected to issue an instruction.



Warp Scheduling

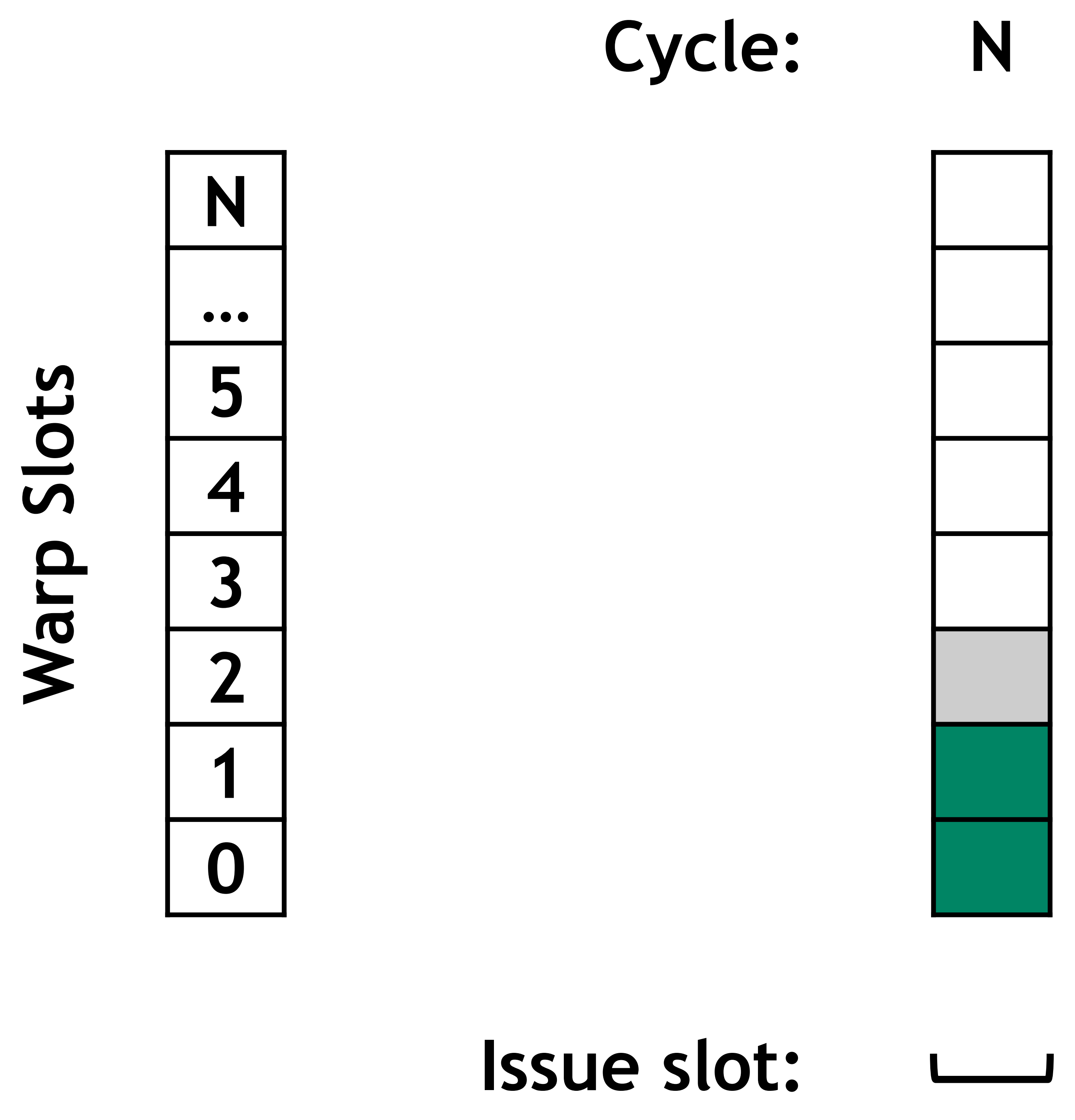
Mental model

Active Warp States:

🟩 **Stalled**
Waiting on:
- an instruction fetch,
- a memory dependency,
- an execution dependency, or
- a synchronization barrier.

⬜ **Eligible**
Ready to issue an instruction.

🟩 **Selected**
Eligible that is selected to issue an instruction.



Warp Scheduling

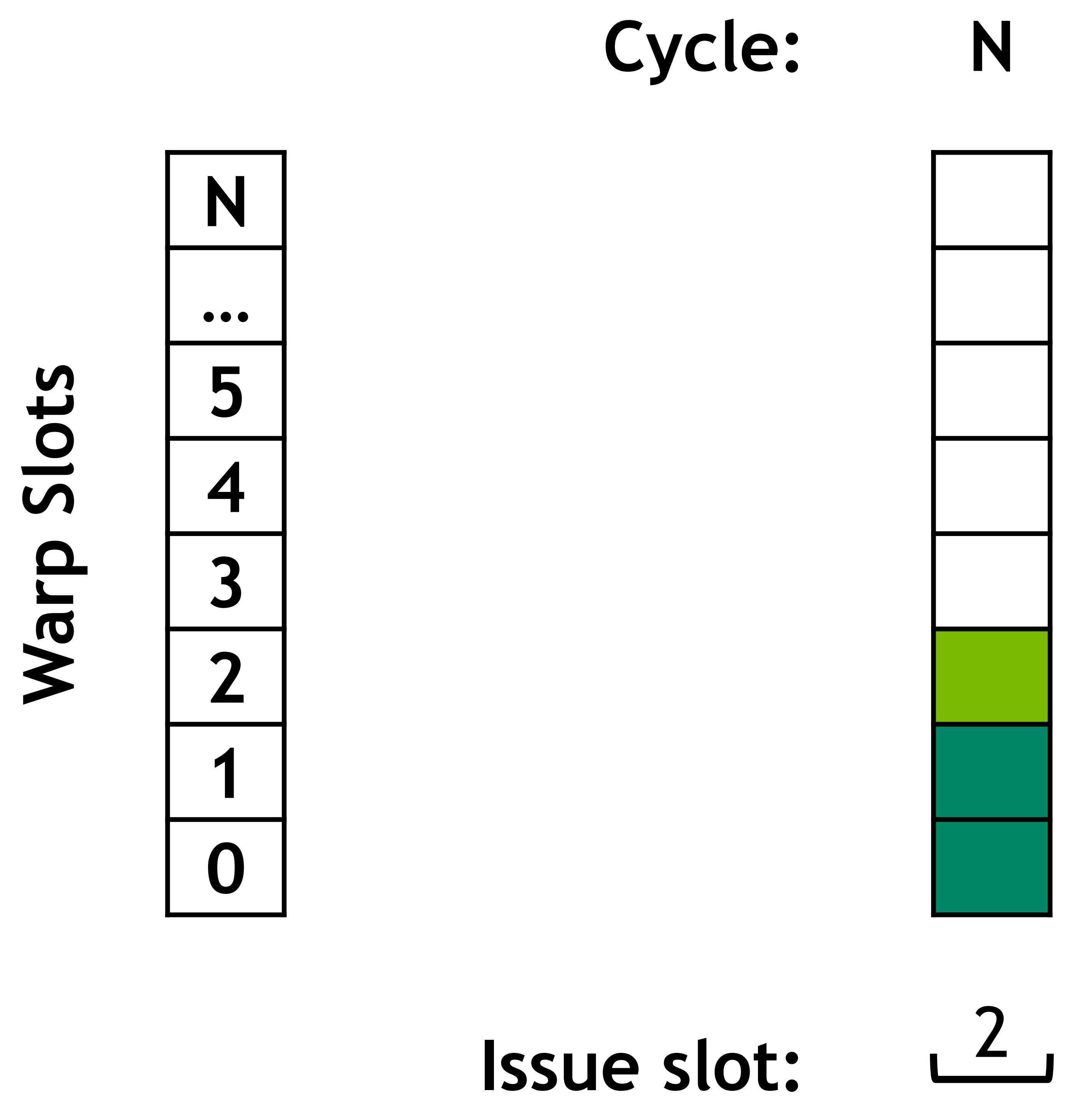
Mental model

Active Warp States:

🟩 **Stalled**
Waiting on:
  an instruction fetch,
  a memory dependency,
  an execution dependency, or
  a synchronization barrier.

⬜ **Eligible**
Ready to issue an instruction.

🟩 **Selected**
Eligible that is selected to issue an instruction



Each cycle: out of all eligible warps, select one to issue on that cycle

Warp Scheduling

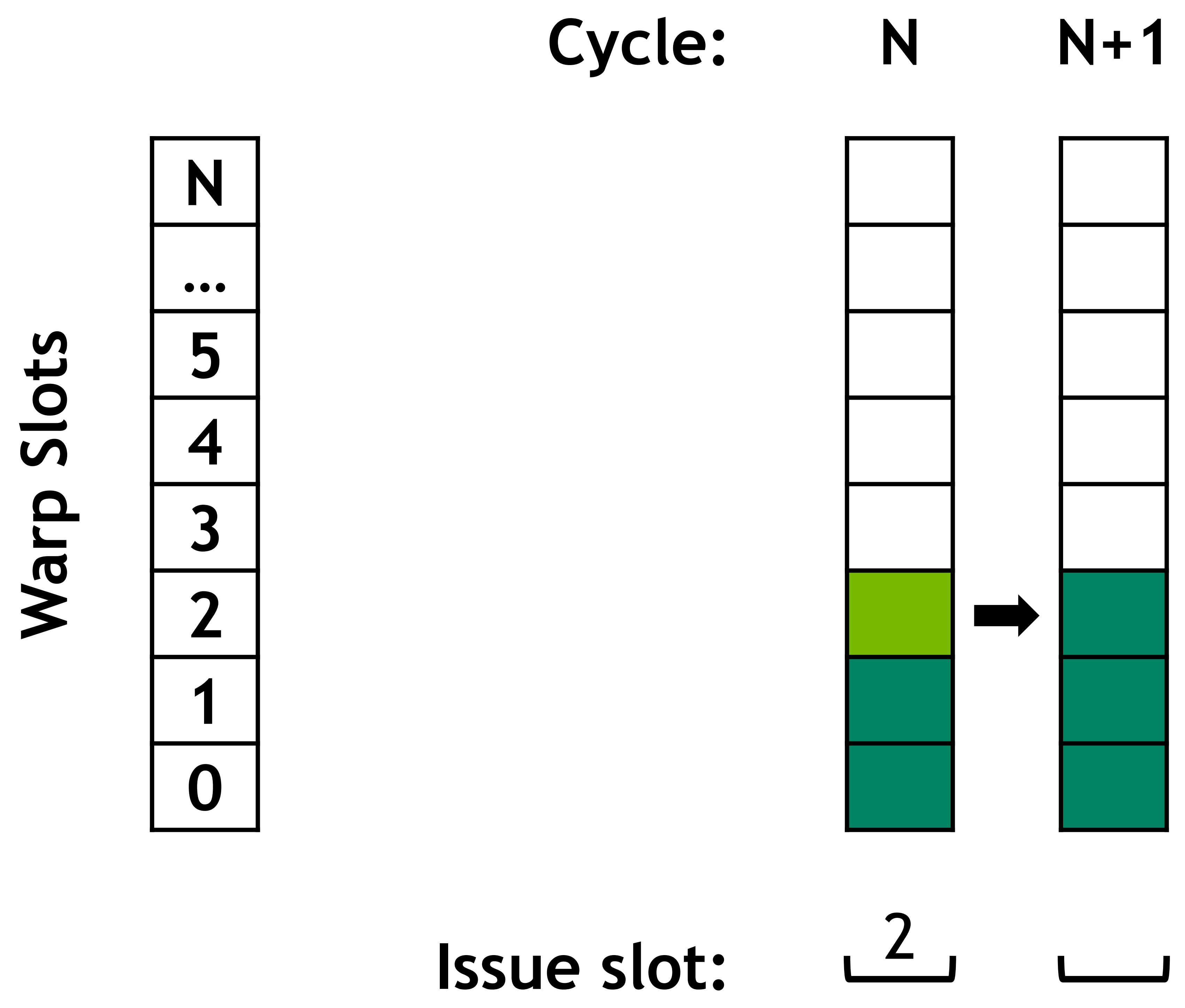
Mental model

Active Warp States:

🟩 **Stalled**
Waiting on:
- an instruction fetch,
- a memory dependency,
- an execution dependency, or
- a synchronization barrier.

⬜ **Eligible**
Ready to issue an instruction.

🟩 **Selected**
Eligible that is selected to issue an instruction



Warp selected at cycle N is not eligible in cycle N+1.
E.g., instructions with longer latencies.

Warp Scheduling

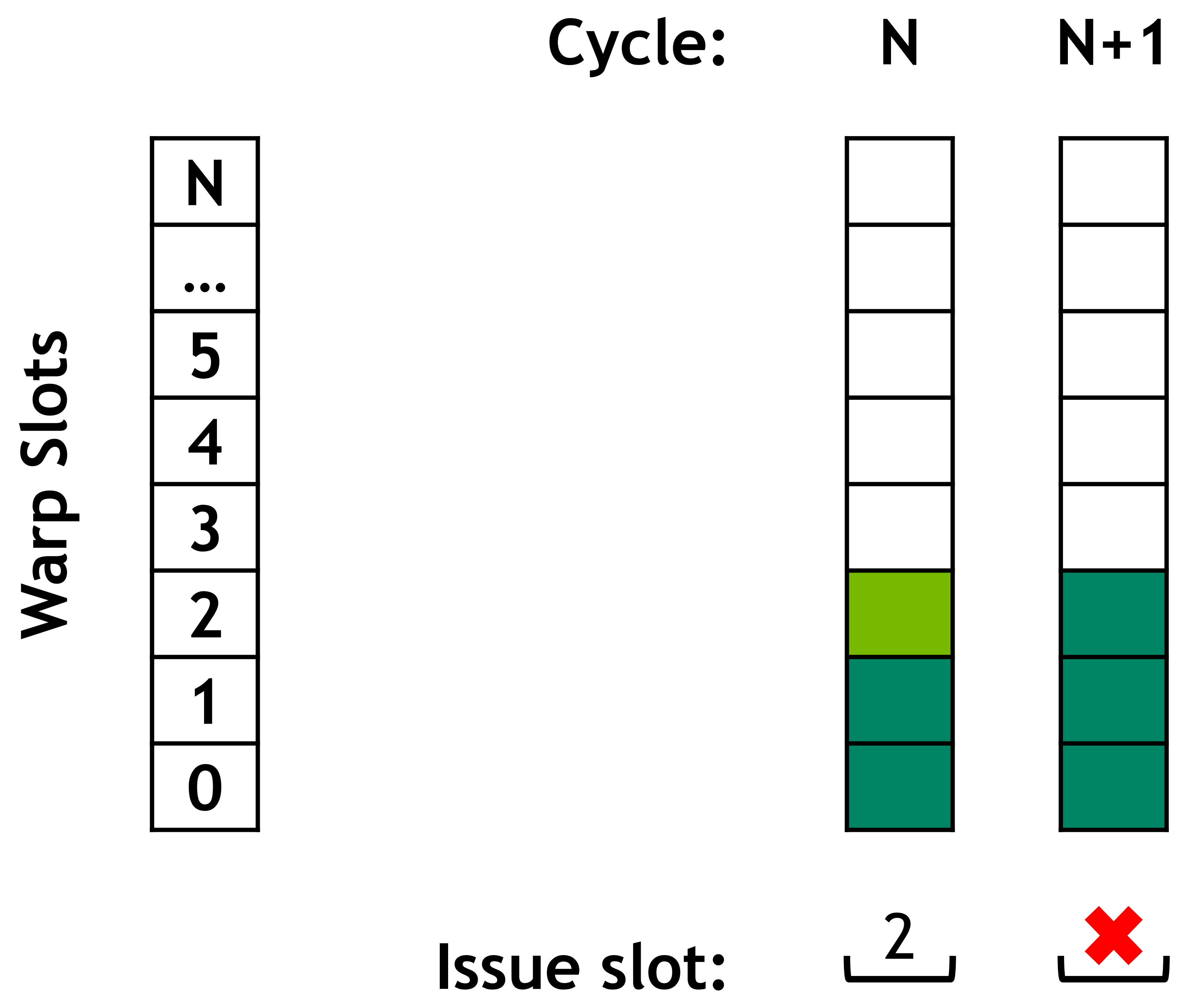
Mental model

Active Warp States:

🟩 **Stalled**
Waiting on:
- an instruction fetch,
- a memory dependency,
- an execution dependency, or
- a synchronization barrier.

⬜ **Eligible**
Ready to issue an instruction.

🟩 **Selected**
Eligible that is selected to issue an instruction



No eligible warps! Issue slot unused.

Warp Scheduling

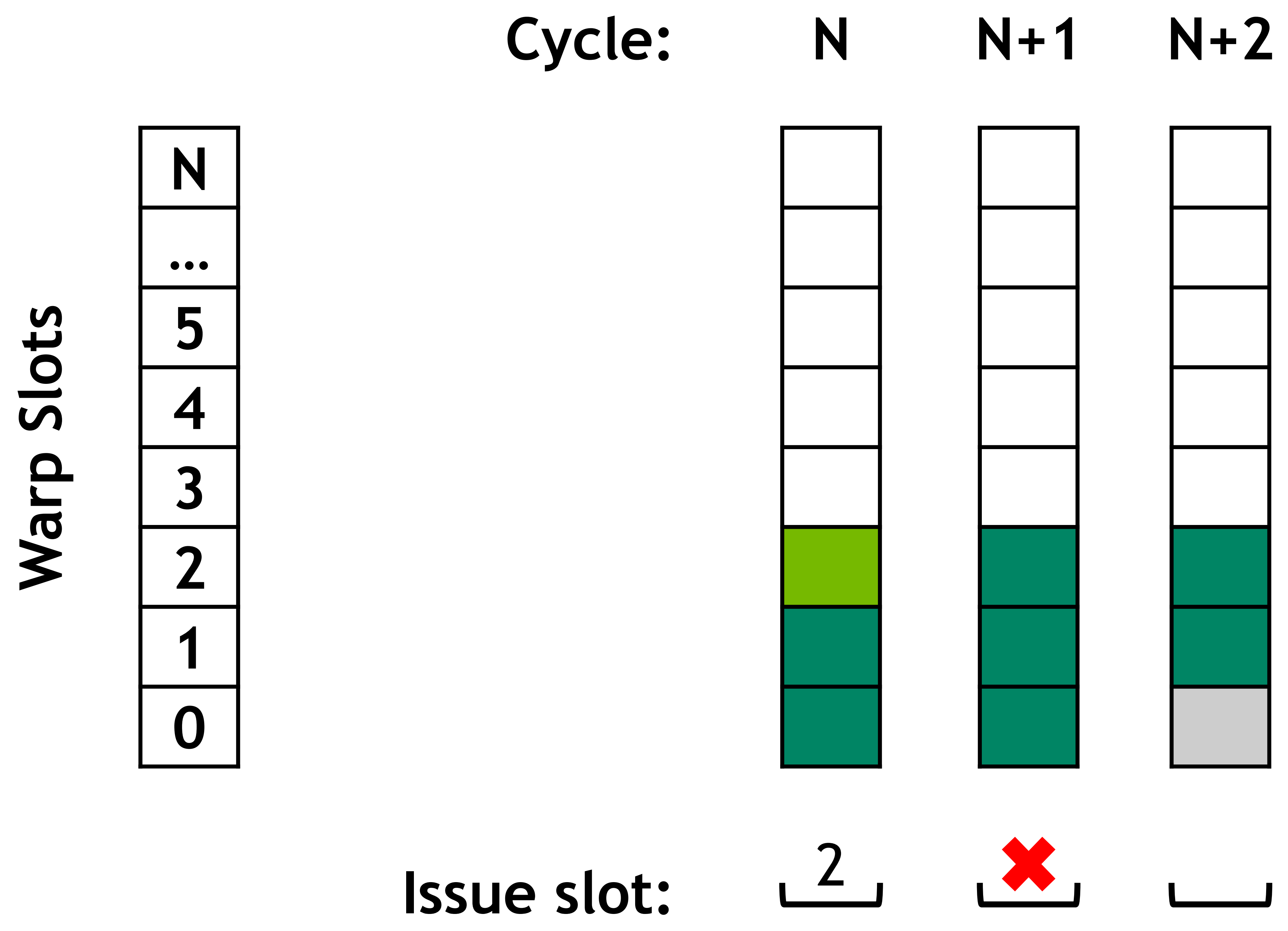
Mental model

Active Warp States:

🟩 **Stalled**
Waiting on:
- an instruction fetch,
- a memory dependency,
- an execution dependency, or
- a synchronization barrier.

⬜ **Eligible**
Ready to issue an instruction.

🟩 **Selected**
Eligible that is selected to issue an instruction



Warp at slot 0 becomes eligible.

Warp Scheduling

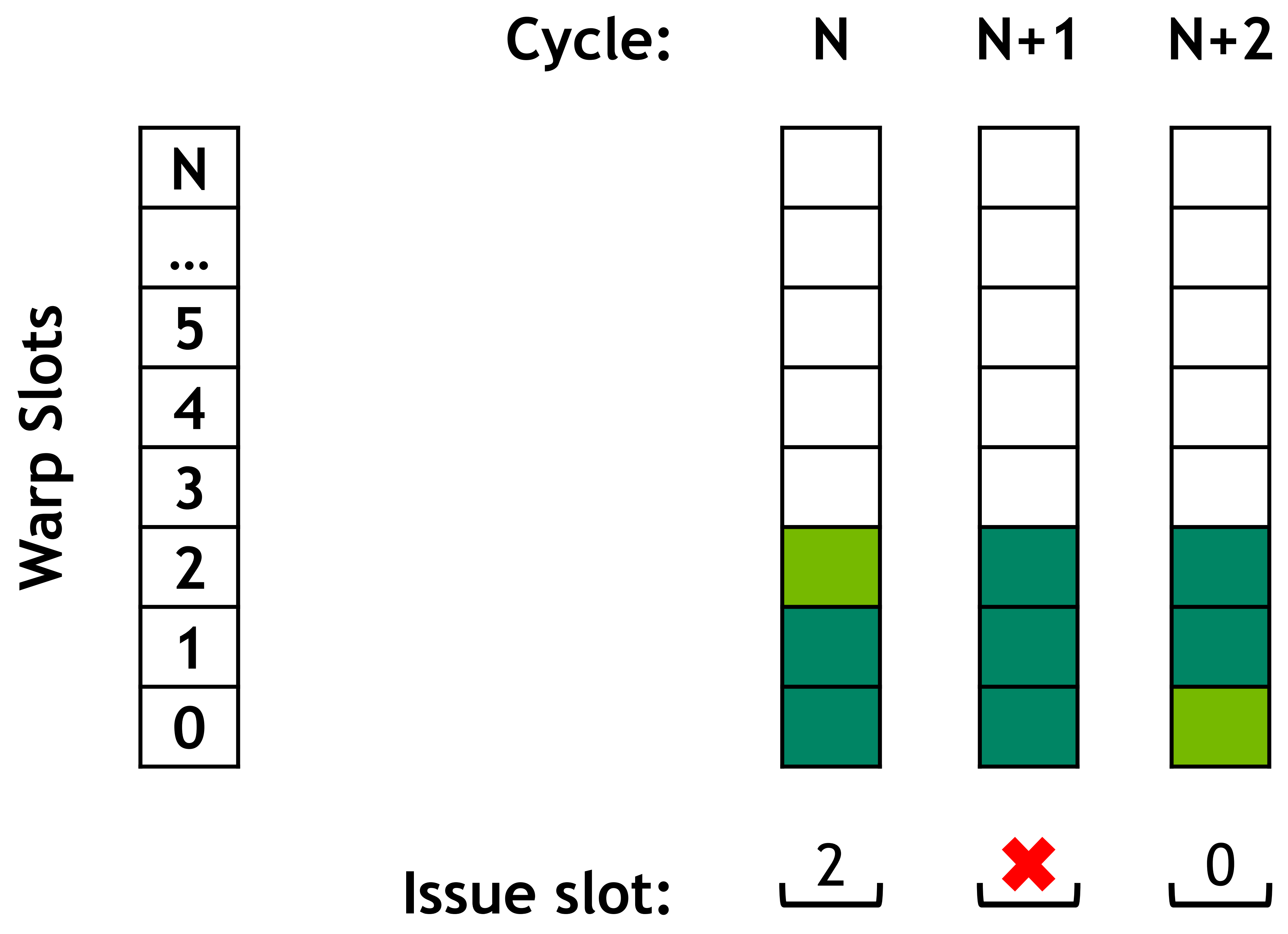
Mental model

Active Warp States:

🟩 **Stalled**
Waiting on:
- an instruction fetch,
- a memory dependency,
- an execution dependency, or
- a synchronization barrier.

⬜ **Eligible**
Ready to issue an instruction.

🟩 **Selected**
Eligible that is selected to issue an instruction



Warp at slot 0 is selected.

Warp Scheduling

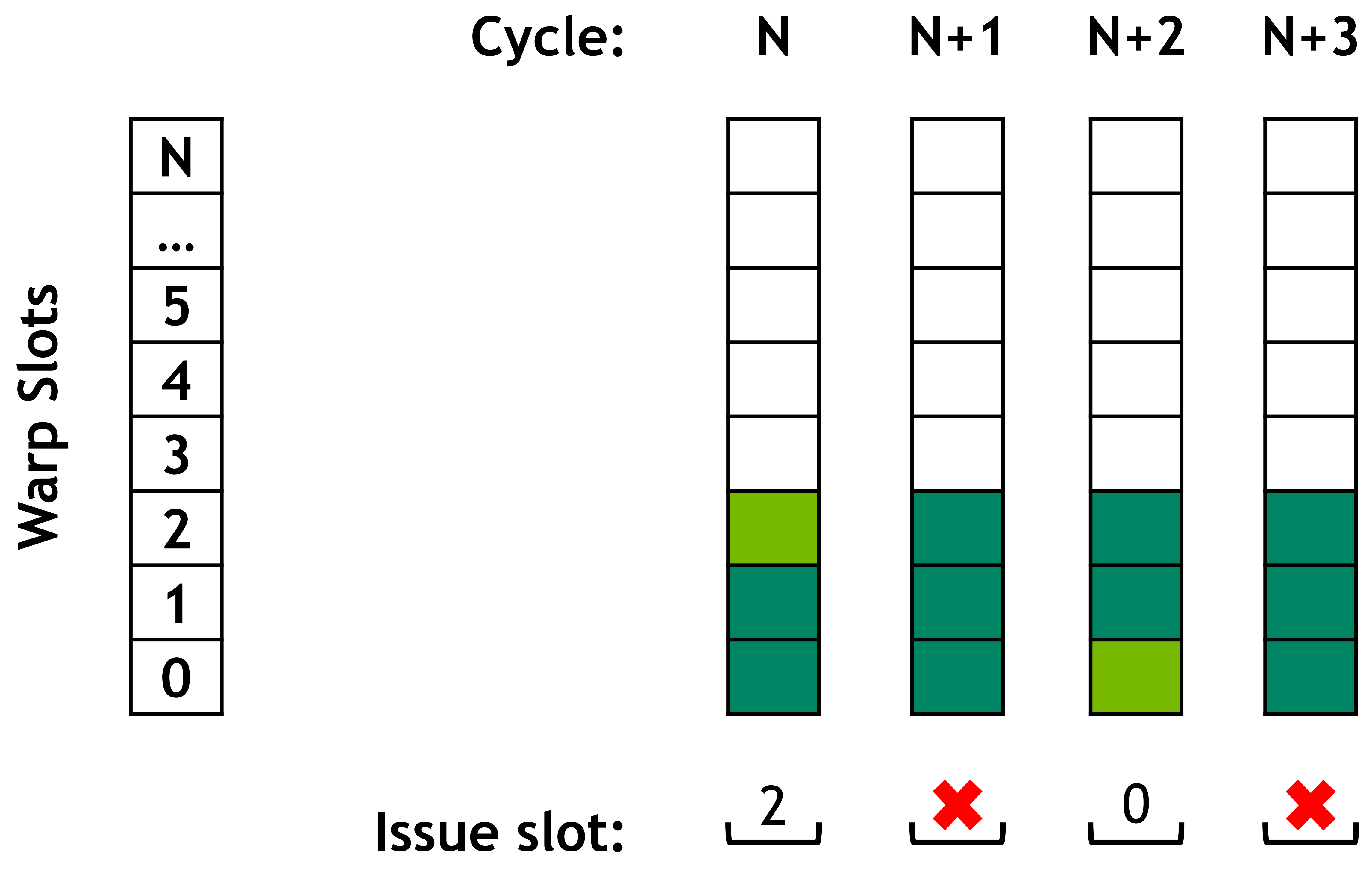
Mental model

Active Warp States:

🟩 **Stalled**
Waiting on:
- an instruction fetch,
- a memory dependency,
- an execution dependency, or
- a synchronization barrier.

⬜ **Eligible**
Active warp that is not stalled.

🟩 **Selected**
Eligible warp that is selected to issue an instruction.



No eligible warps! Issue slot unused.

Warp Scheduling

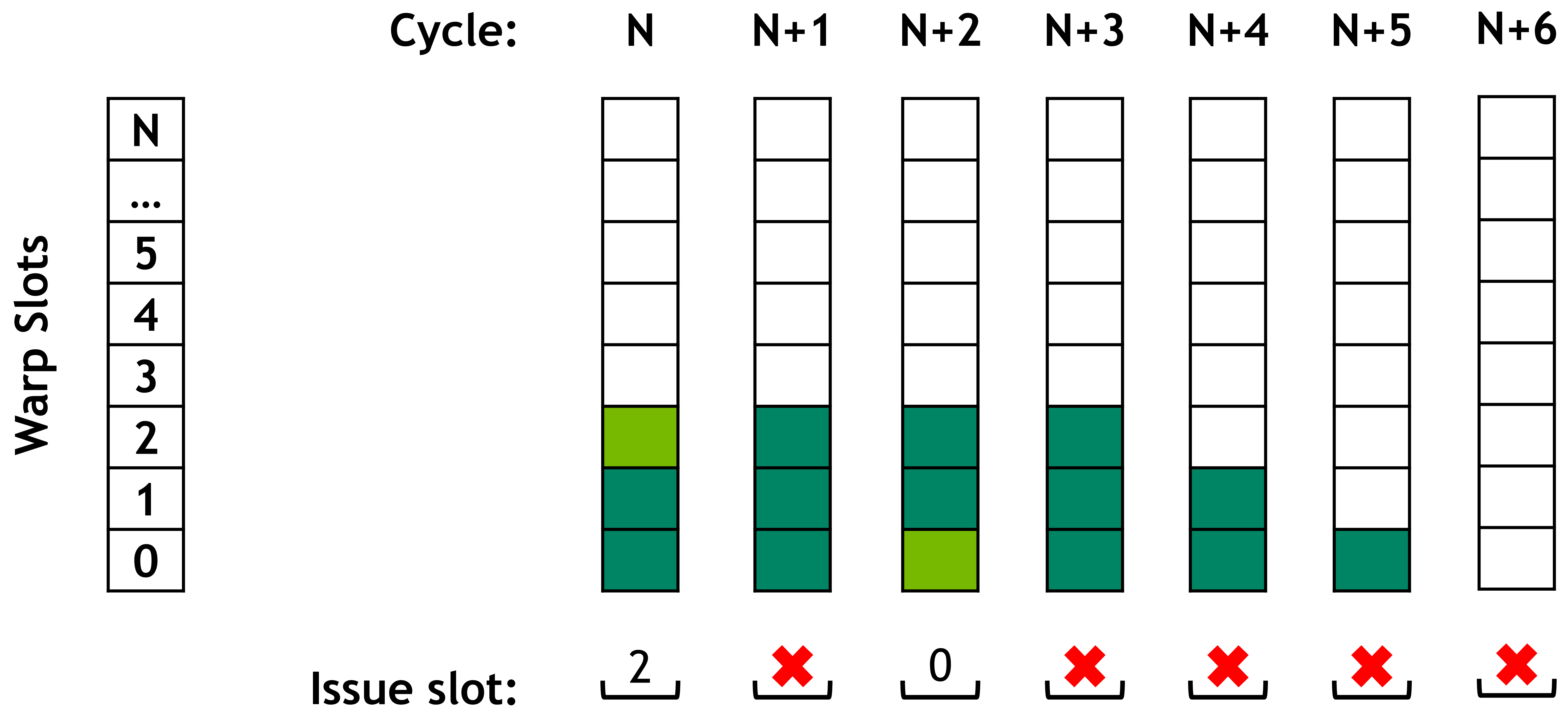
Mental model

Active Warp States:

🟩 **Stalled**
Waiting on:
an instruction fetch,
a memory dependency,
an execution dependency, or
a synchronization barrier.

⬜ **Eligible**
Active warp that is not stalled.

🟩 **Selected**
Eligible warp that is selected to issue an instruction.



Having more **active warps** would help reduce idle issue slots and hide latencies of stalled warps.

How to Increase Active Warps?

Occupancy

- There is a maximum number of warps which can be concurrently active on an SM.
  - **Device** (depends on compute capability of the GPU)
  - **Achievable** (depends on kernel implementation + compiler)
  - **Achieved** (depends mostly on the grid size)

$$Occupancy = \frac{Achievable\ \#\ active\ warps\ per\ SM}{Device\ \#\ active\ warps\ per\ SM}$$

- Occupancy of a CUDA kernel may be limited by:
  - **Register usage**
    - SM registers are partitioned among threads.
  - **Shared memory usage**
    - SM shared memory is partitioned among thread blocks.
  - **Thread block size**
    - Threads are allocated at thread block granularity.

Analyze the occupancy of CUDA kernels with **NVIDIA Nsight Compute**!

Occupancy Limiters

Registers

- Register usage: compile with `--ptxas-options=-v`
  - Reports registers per thread
- The maximum number of registers per thread can be set manually:
  - At compile time on a per-file basis using the `--maxrregcount` flag of nvcc
  - Per-kernel using the `__launch_bounds__` qualifier
- Hopper has 64K (65536) registers per SM
  - Allocated in fixed-size chunks of 256 registers

- **Example:**
  - Kernel uses 63 registers per thread
  - Registers per warp = 63 * 32 = 2016
  - Registers allocated per warp = 2048
  - Achievable active warps per SM = 65536 / 2048 = 32
  - Occupancy = 32 / 64 * 100 = 50%
    - Hopper supports up to 64 warps per SM

Occupancy Limiters

Shared memory

- Shared memory usage: compile with `--ptxas-options=-v`.
  - Reports static shared memory usage per thread block.
- Hopper has 228 KiB of shared memory.
  - 1KiB per thread block is reserved for system use.
  - With opt-in using dynamic shared memory.

- **Example:**
  - Kernel uses 17408 bytes of shared memory per 128-thread block.
  - Blocks per SM = 233472 / (17408 +1024) = 12.66
  - Achievable active warps per SM = 12 * 128 / 32 = 48
  - Occupancy = 48 / 64 * 100 = 75%
    - Hopper supports up to 64 warps per SM.

Occupancy Limiters

Thread block size

- Thread block size is a multiple of warp size (32).
  - Even if you request fewer threads, HW rounds up.
- Each thread block can have a maximum size of 1024.
- Each SM can have up to 64 warps, 32 blocks and 2048 threads (Hopper).

Block Size	Active threads per SM	Active Warps per SM	Active Warps per Block	Active Blocks per SM	Occupancy (%)
32	1024	32	1	32	**50**
64	2048	64	2	32	100
256	2048	64	8	8	100
512	2048	64	16	4	100
768	1536	48	24	2	**75**
1024	2048	64	32	2	100

ILP vs TLP for Hiding Latencies

Computing **c = c + a * b**

- **Experimental setup:**
  - NVIDIA H100 SXM, 1980 MHz
  - Problem size = 2^28
  - Datatype = float
  - Baseline thread block size = 32 (50% occupancy)
  - **Experiment #1**: increase occupancy
    - Thread block size = 64 (100% occupancy)
  - **Experiment #2**: increase ILP by computing more elements per thread
    - Elements per thread = 2, 4

Implementation	Elements per Thread	Thread Block Size	Main Memory Bandwidth Utilization (%)	SM Occupancy (%)	GPU Time (ms)
Baseline	1	32	25	50	5.0
Experiment #1	1	64	51	100	2.5
Experiment #2	2	32	51	50	2.5
Experiment #2	4	32	82	50	**1.6**

What Occupancy Do I Need?

General guidelines

Rule of thumb: Try to maximize occupancy.

But some algorithms will run better at low occupancy.

More registers and shared memory can allow higher data reuse, higher ILP, higher performance.

Low Occupancy

Fewer threads per SM.

More resources per thread.

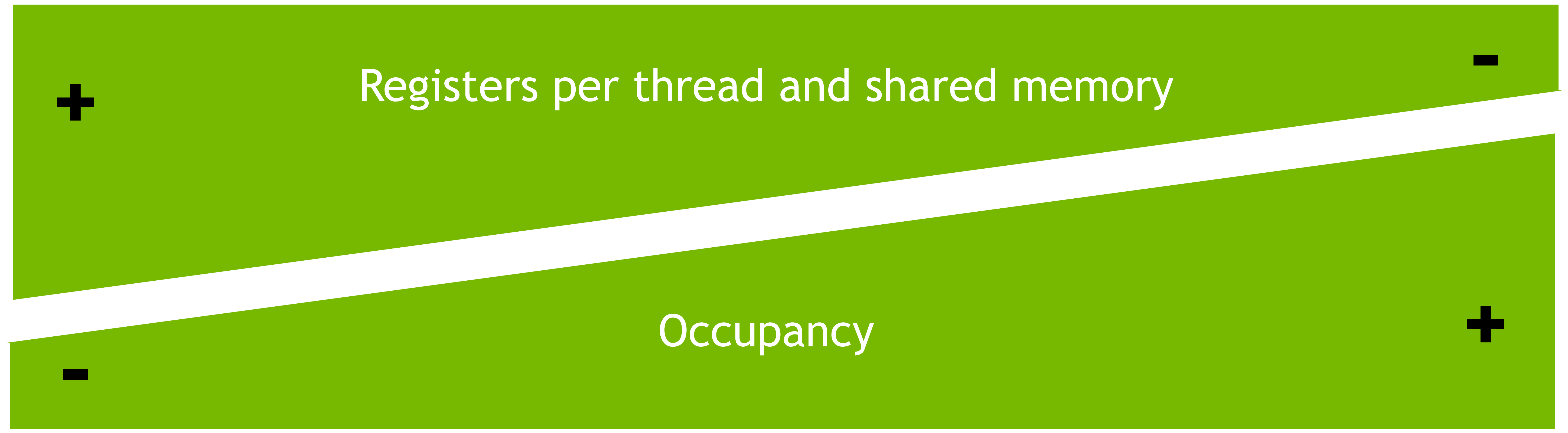
Enough instruction-level parallelism or GPU will starve!

High Occupancy

More threads per SM.

Fewer registers per thread.

Rely on thread parallelism to hide latencies!

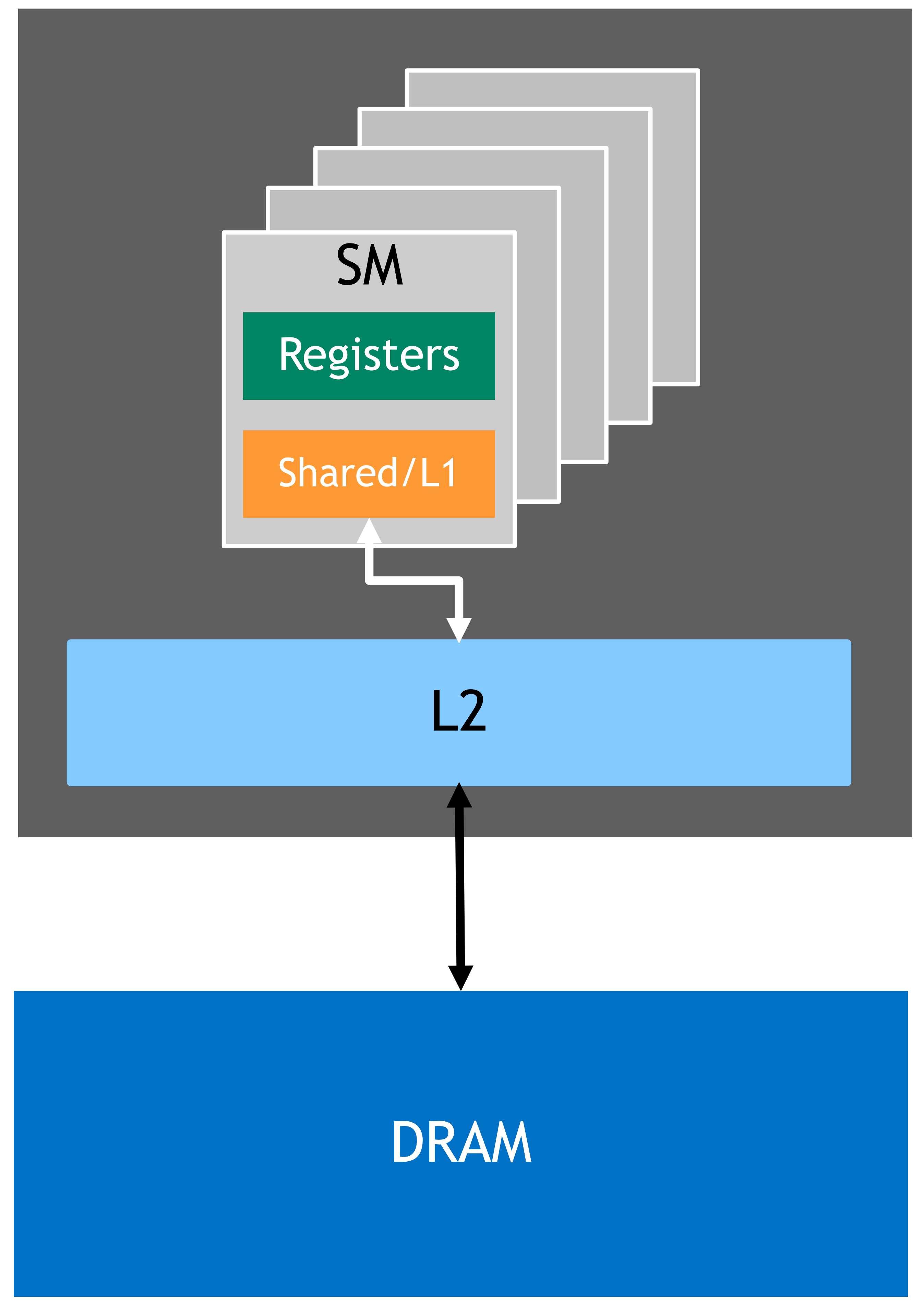


Complex algorithms ⟷ Simple algorithms

Maximizing Memory Throughput

Memory Hierarchy

NVIDIA H200 SXM



4.8 TB/s

Register File (64K 32-bit registers per SM)

Unified Shared Memory / L1 Cache (228 KiB per SM, variable split)

L2 Cache (50 MiB)

HBM3e (141 GB)

Why Do GPUs Have Caches?

- 100s ~ 1000s of threads sharing the L1 and ~100000s of threads sharing the L2.
  - L1, L2 capacity per thread is relatively small.

Caches on GPUs are mostly useful for:

- “Smoothing” irregular, misaligned access patterns.
- Caching common data accessed by multiple threads.
- Faster register spills, local memory.
- Faster atomics.

What about cache blocking?

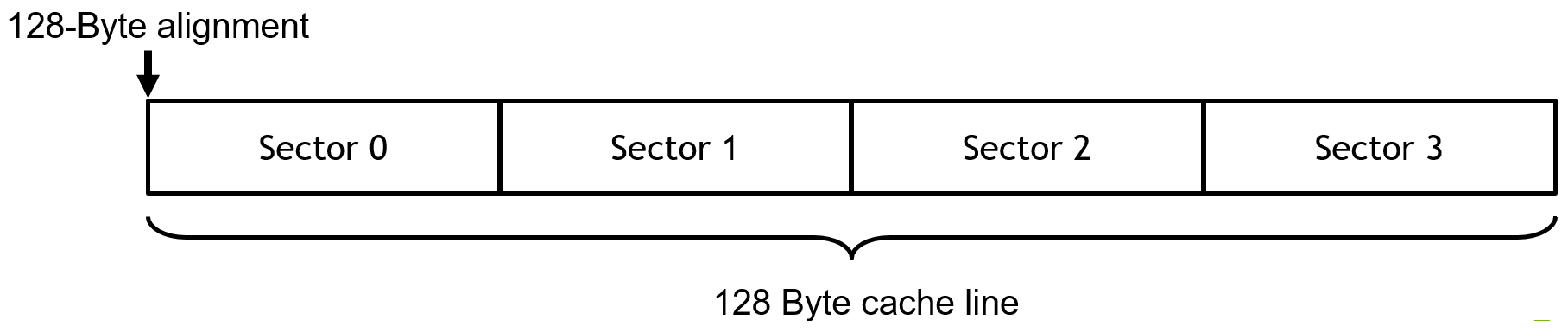
- L2 cache blocking may be feasible.
  - For an example of efficient use of L2 cache blocking, see **[S62192]: “Advanced Performance Optimization in CUDA”**.

Memory Transactions

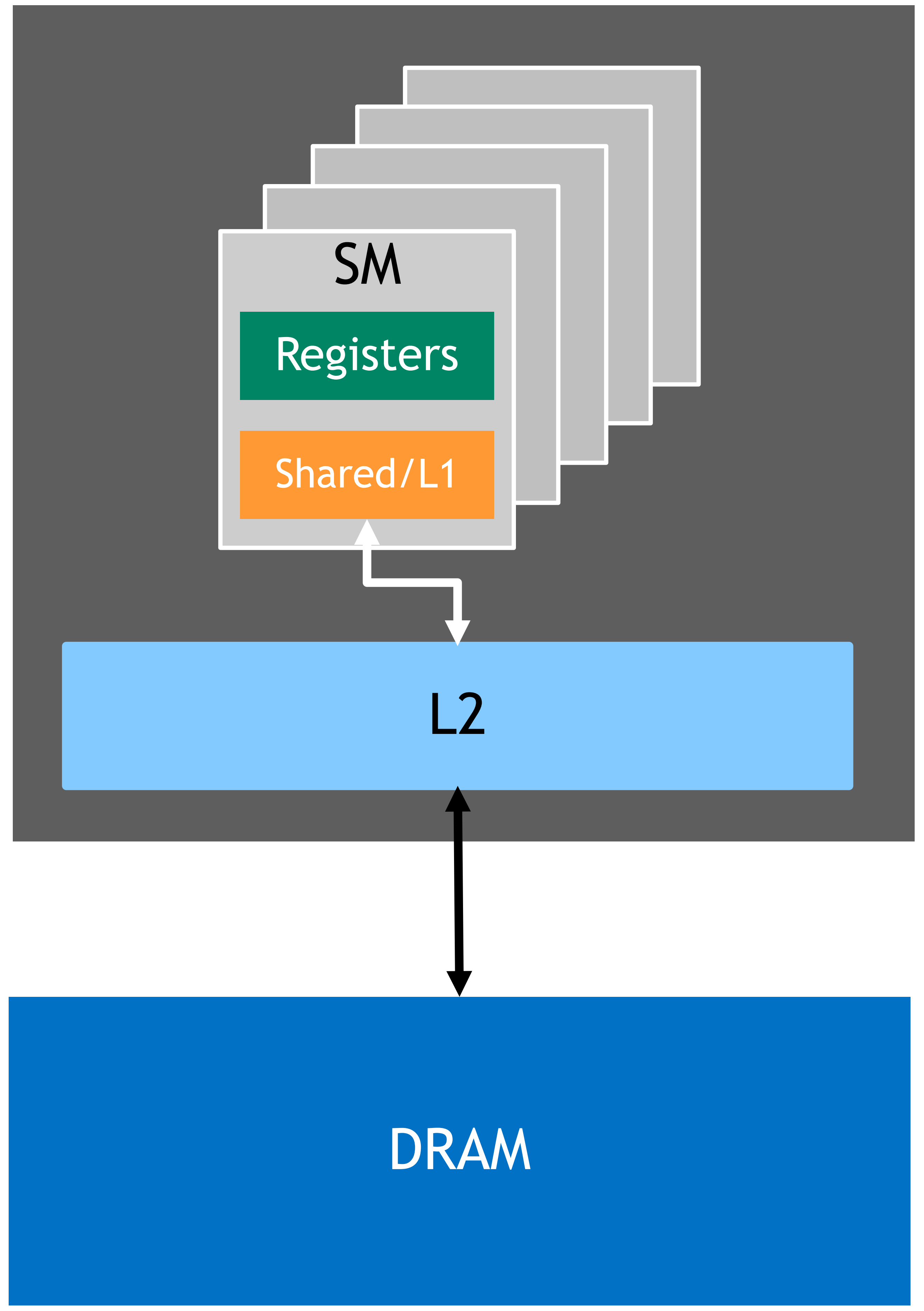
Cache lines and sectors

- Minimum memory access granularity: **32 bytes** = **1 sector**
  - **L1 to L2:** 1 sector
  - **L2 to Global:** 2 sectors (default)
    - User can set a preferred granularity with `cudaDeviceSetLimit()` and `cudaLimitMaxL2FetchGranularity`.
    - Only a hint though!

- Cache line size: **128 bytes** = **4 sectors**
  - Cache ”management” granularity = 1 cache line
    - Coalescing of requests.
    - Evictions.



Memory Reads & Writes



Reads

Check if data is in L1 (if not, check L2)

Check if data is in L2 (if not, get from DRAM)

Unit of data moved: **full sector**

Writes

L1 is write-through: update both L1 and L2

L2 is write back: flush data to DRAM only when needed

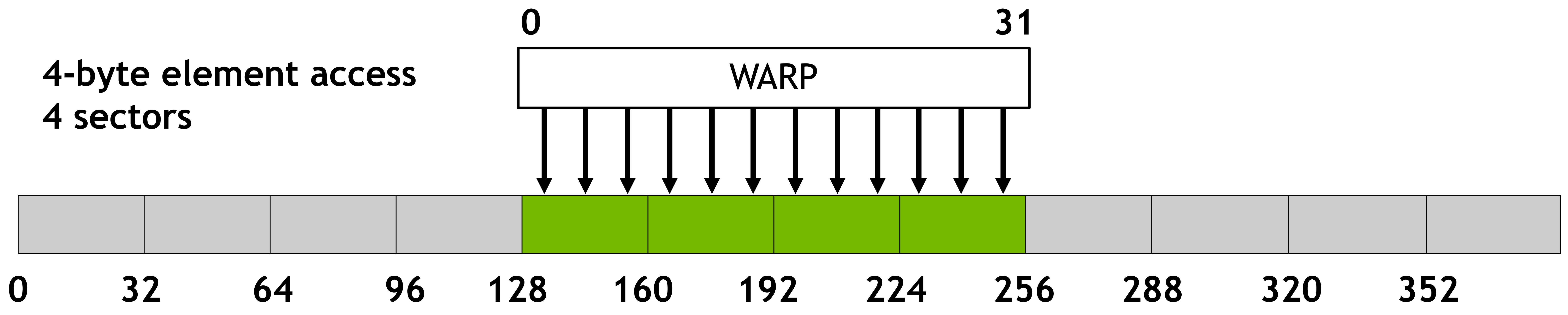
Unit of data moved: **partial sector***

* Depends on whether ECC is enable/disabled.

Global Memory Access Patterns

Aligned and sequential

4-byte element access
4 sectors



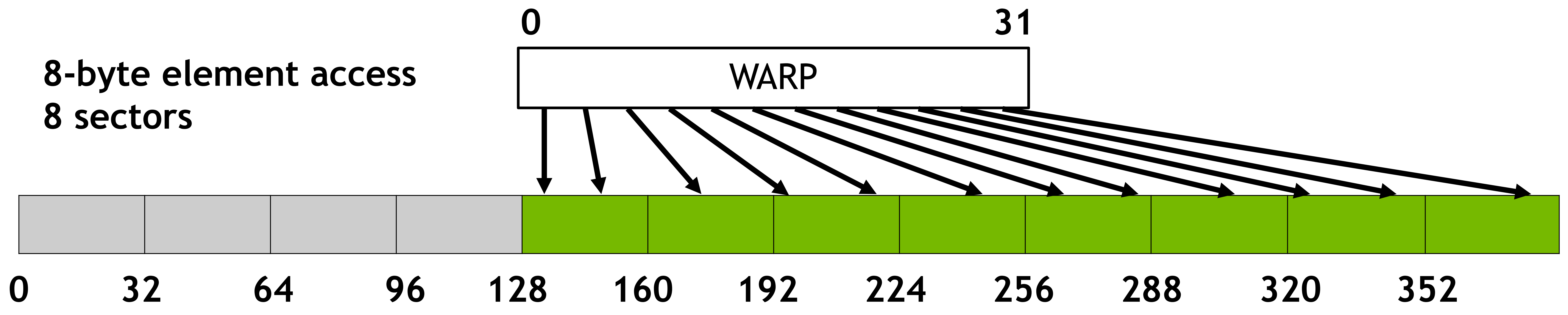
Memory Addresses

COALESCED!

Global Memory Access Patterns

Aligned and sequential

8-byte element access
8 sectors



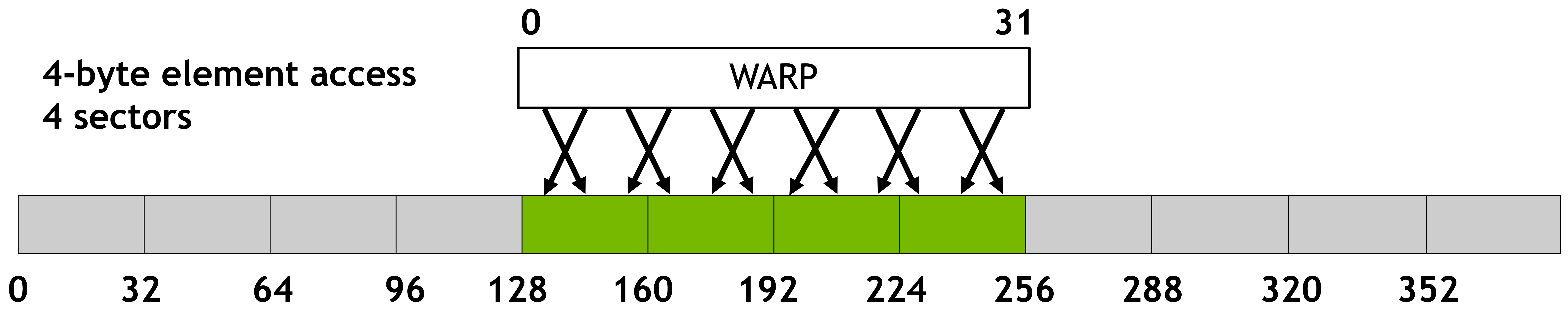
Memory Addresses

COALESCED!

Global Memory Access Patterns

Aligned and non-sequential

4-byte element access
4 sectors

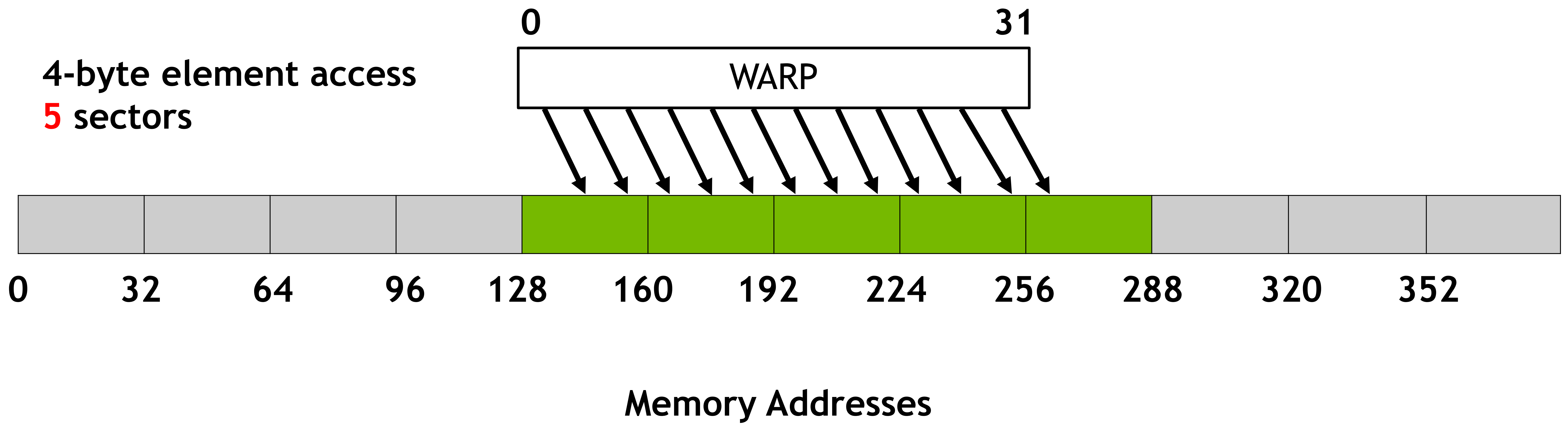


Memory Addresses

COALESCED!

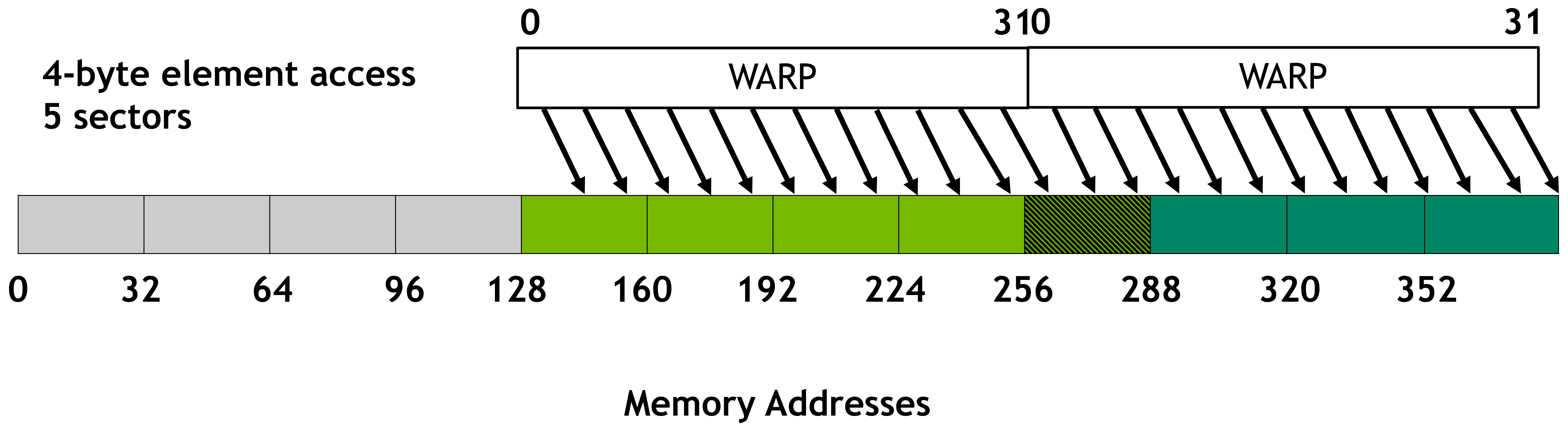
Global Memory Access Patterns

Mis-aligned and sequential



Global Memory Access Patterns

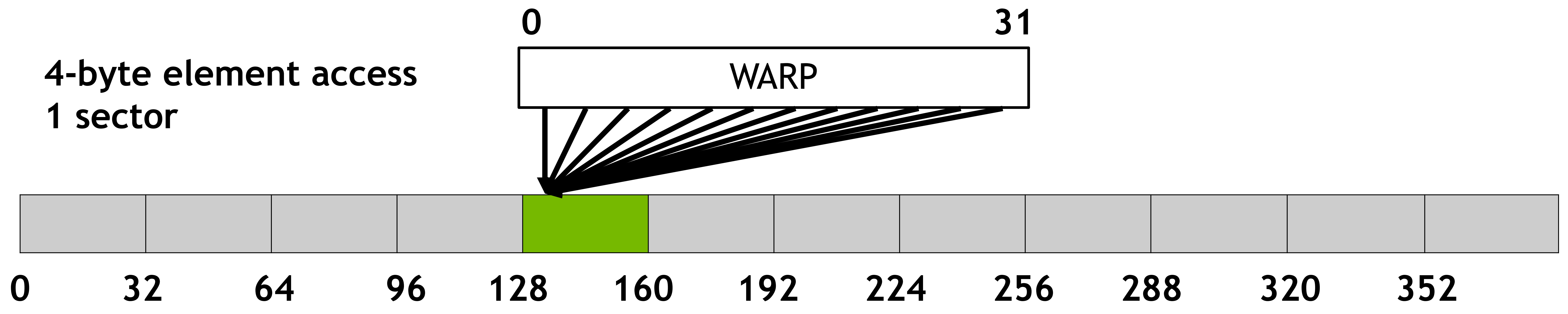
Mis-aligned and sequential



Global Memory Access Patterns

Same address

4-byte element access
1 sector

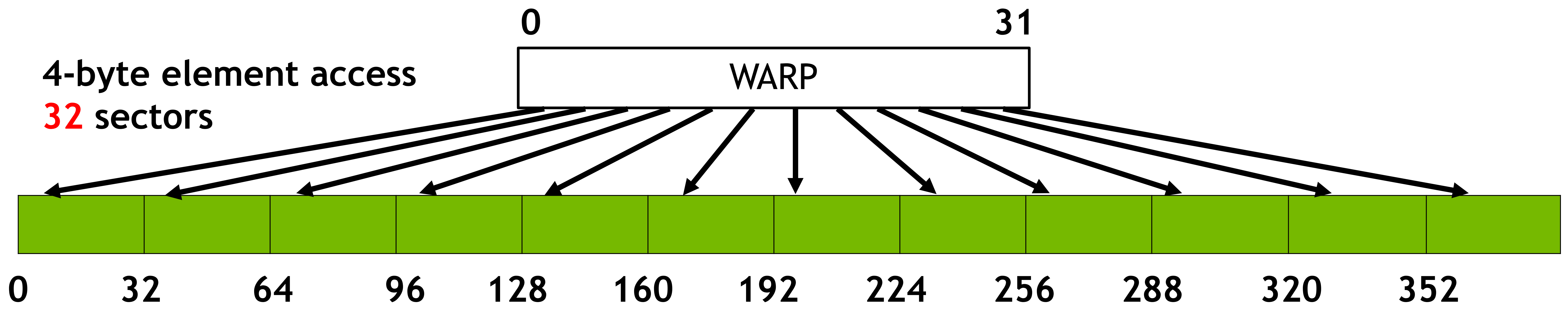


Memory Addresses

Global Memory Access Patterns

Aligned and strided

4-byte element access
32 sectors

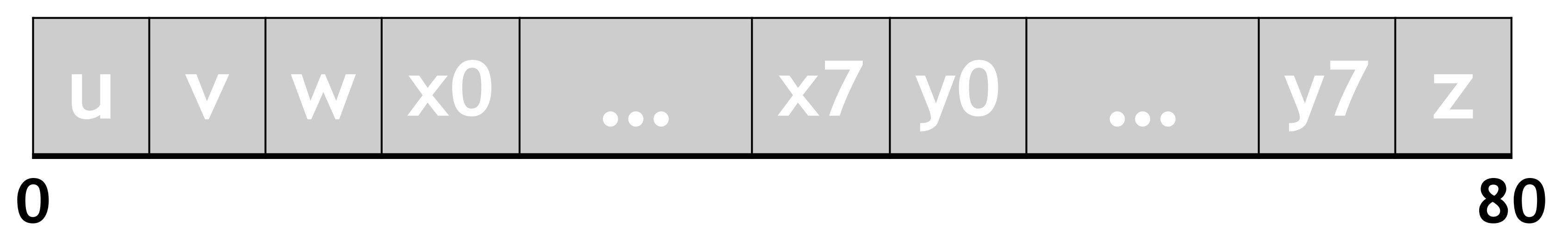


Memory Addresses

Impact of Data Layout

Array-of-Structures (AoS) vs Structure-of-Arrays (SoA)

AoS Memory Layout

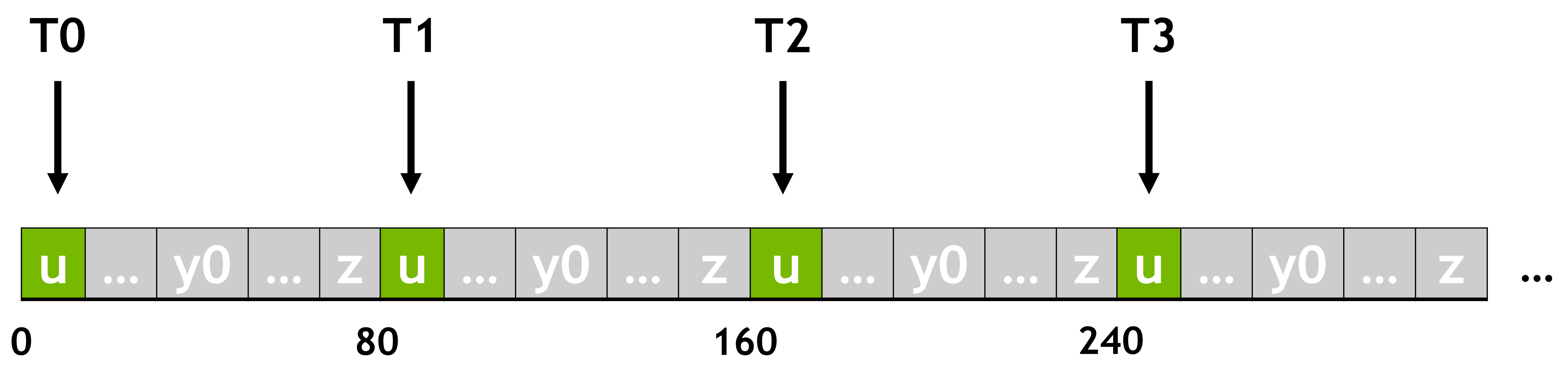


```
struct Coefficients
{
  float u, v, w;
  float x[8], y[8], z;
};

__global__ void kernel(Coefficients *data)
{
  int i = cg::this_grid.thread_rank();

  data[i].u = data[i].u + 10.f;
  data[i].y[0] = data[i].y[0] + 10.f;
}
```

Impact of Data Layout

Array-of-Structures (AoS) vs Structure-of-Arrays (SoA)

- When loading coefficients u and y[0]:
  - Successive threads in a warp read 4 bytes at 80-byte stride.

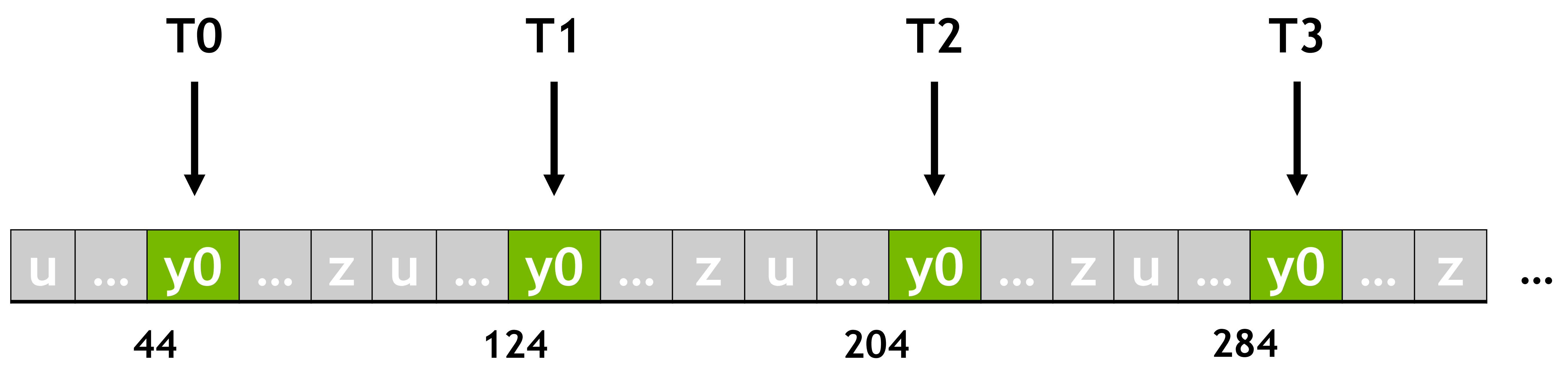


```
struct Coefficients
{
  float u, v, w;
  float x[8], y[8], z;
};

__global__ void kernel(Coefficients *data)
{
  int i = cg::this_grid.thread_rank();

  data[i].u = data[i].u + 10.f;
  data[i].y[0] = data[i].y[0] + 10.f;
}
```

Impact of Data Layout

Array-of-Structures (AoS) vs Structure-of-Arrays (SoA)

- When loading coefficients u and y[0]:
  - Successive threads in a warp read 4 bytes at 80-byte stride.
- We are reading **7x more bytes** than necessary!
  - Remember data is read in sectors of 32 bytes.
  - No potential reuse of the sectors loaded by the previous access.



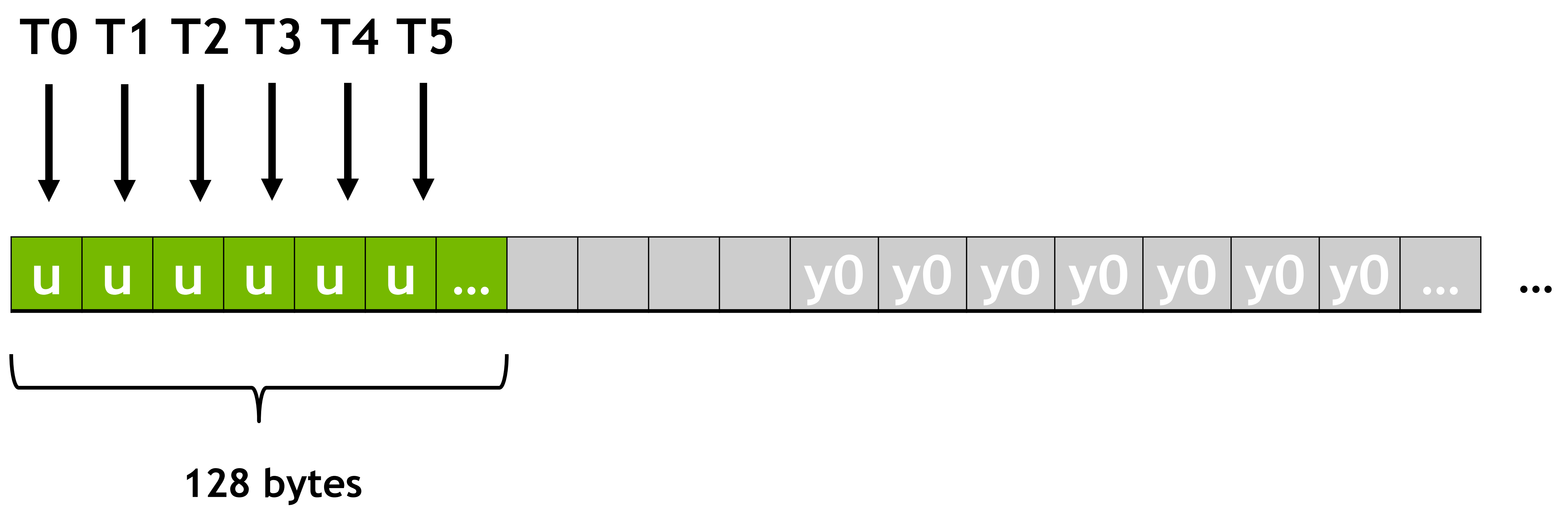
```
struct Coefficients
{
  float u, v, w;
  float x[8], y[8], z;
};

__global__ void kernel(Coefficients *data)
{
  int i = cg::this_grid.thread_rank();

  data[i].u = data[i].u + 10.f;
  data[i].y[0] = data[i].y[0] + 10.f;
}
```

Impact of Data Layout

Array-of-Structures vs Structure-of-Arrays

- Refactoring from **AoS** to **SoA** leads to coalesced memory accesses for u and `y[0]`.

SoA Memory Layout



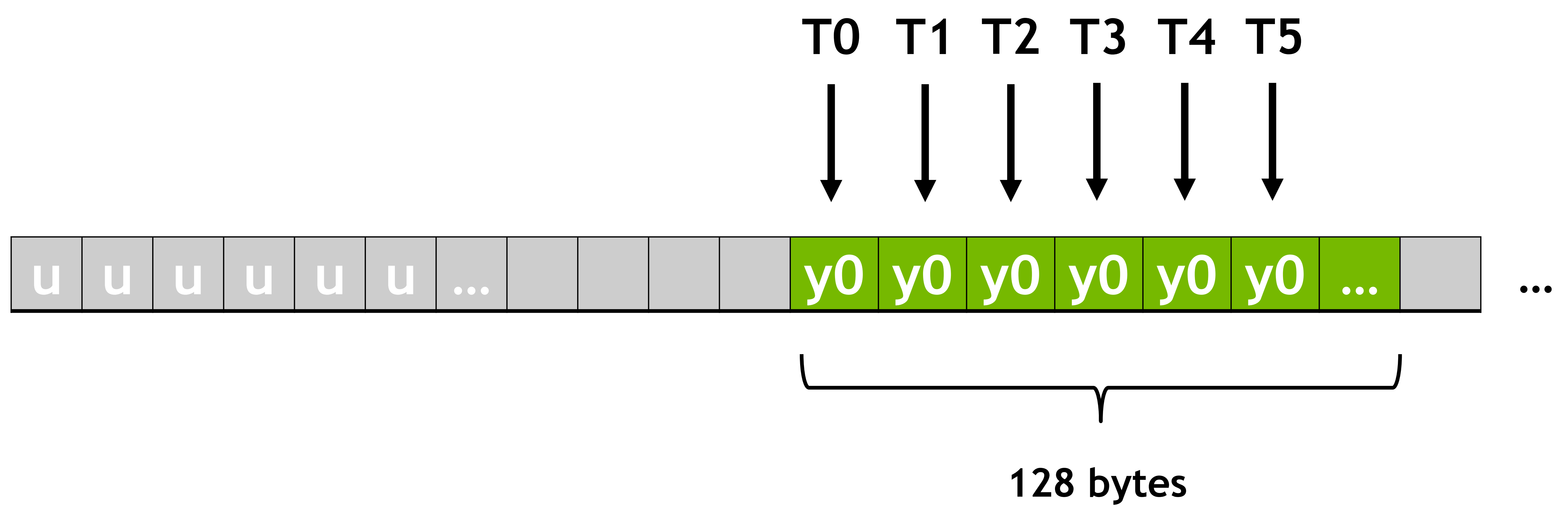
```
struct Coefficients
{
  float *u, *v, *w;
  float *x0, …, *x7, *y0, … *y7, *z;
};

__global__ void kernel(Coefficients data)
{
  int i = cg::this_grid.thread_rank();

  data.u[i] = data.u[i] + 10.f;
  data.y0[i] = data.y0[i] + 10.f;
}
```

Impact of Data Layout

Array-of-Structures vs Structure-of-Arrays

- Refactoring from **AoS** to **SoA** leads to coalesced memory accesses for u and y[0].

SoA Memory Layout



```
struct Coefficients
{
  float *u, *v, *w;
  float *x0, …, *x7, *y0, … *y7, *z;
};

__global__ void kernel(Coefficients data)
{
  int i = cg::this_grid.thread_rank();

  data.u[i] = data.u[i] + 10.f;
  data.y0[i] = data.y0[i] + 10.f;
}
```

Impact of Data Layout

Performance Analysis

- **Experimental setup:**
  - NVIDIA H100 SXM, 1980 MHz
  - Problem size = 2^28
  - Thread block size = 256

Implementation	Load Efficiency (%)	Store Efficiency (%)	Main Memory Bandwidth Utilization (%)	GPU Time (ms)
AoS	12.5	12.5	13.50	28.497
SoA	100	100	79.47	**4.836**

Unified L1 and Shared Memory

- Can be used as a typical hardware managed cache (L1) and/or a user-managed memory (Shared Memory)
  - An application can configure its preferred split at runtime using `cudaFuncSetAttribute()` with the attribute `cudaFuncAttributePreferredSharedMemoryCarveout`.

- **Shared memory** can be useful for:
  - Storing frequently used data
  - Improving global memory access patterns
  - Data layout conversion
  - Communication among threads of a thread block

Shared Memory

Capacity:

- Default 48 KiB per thread block, opt-in to get more using `cudaFuncSetAttribute()` with the attribute `cudaFuncAttributeMaxDynamicSharedMemorySize`.
    - Up to 227KiB per thread block on Hopper.

Organization:

- Divided into 32 banks, each 4-byte wide.
- Successive **4-byte** words map to successive banks.
- Bank index calculation examples:
    - (4-byte word index) % 32
    - (1-byte word index / 4) % 32

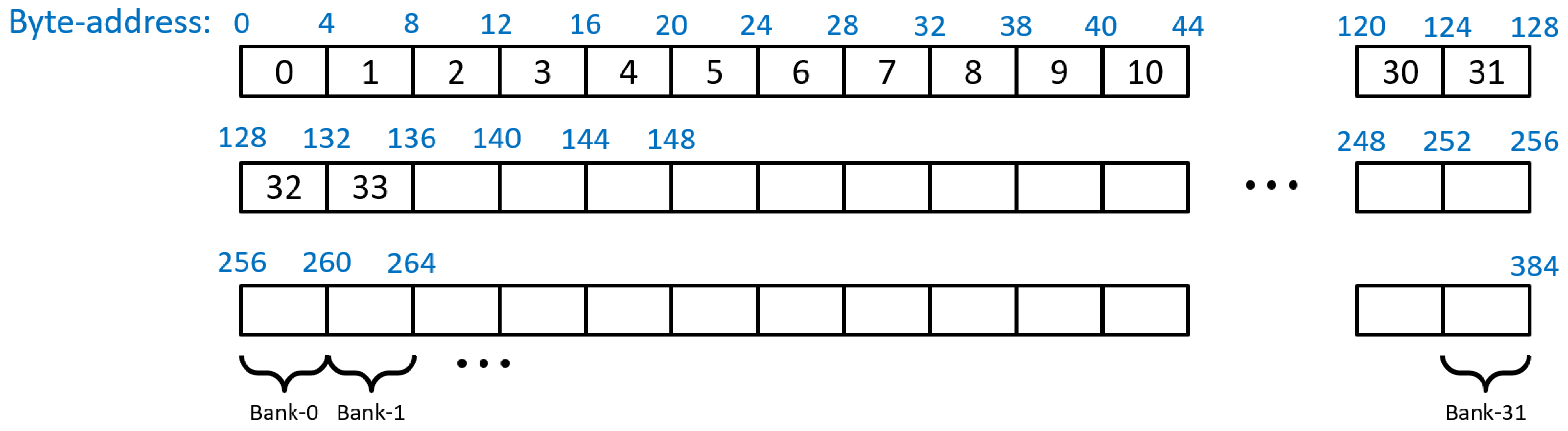
Performance:

- Slower than registers, but much faster than global memory.

Logical View of Shared Memory Banks

4-byte data

With 4-Bytes data



Processing Data Types of Different Sizes

- **4-byte** or smaller data types:
  - Process addresses of all threads in a warp in a single phase

- **8-byte** data types:
  - Process addresses of all threads in a warp in **2** phases
    - Each phase processes addresses of **half** of a warp

- **16-byte** data types:
  - Process addresses of all threads in a warp in **4** phases
    - Each phase processes addresses of a **quarter** of a warp

Shared Memory Access Patterns

Bank conflicts

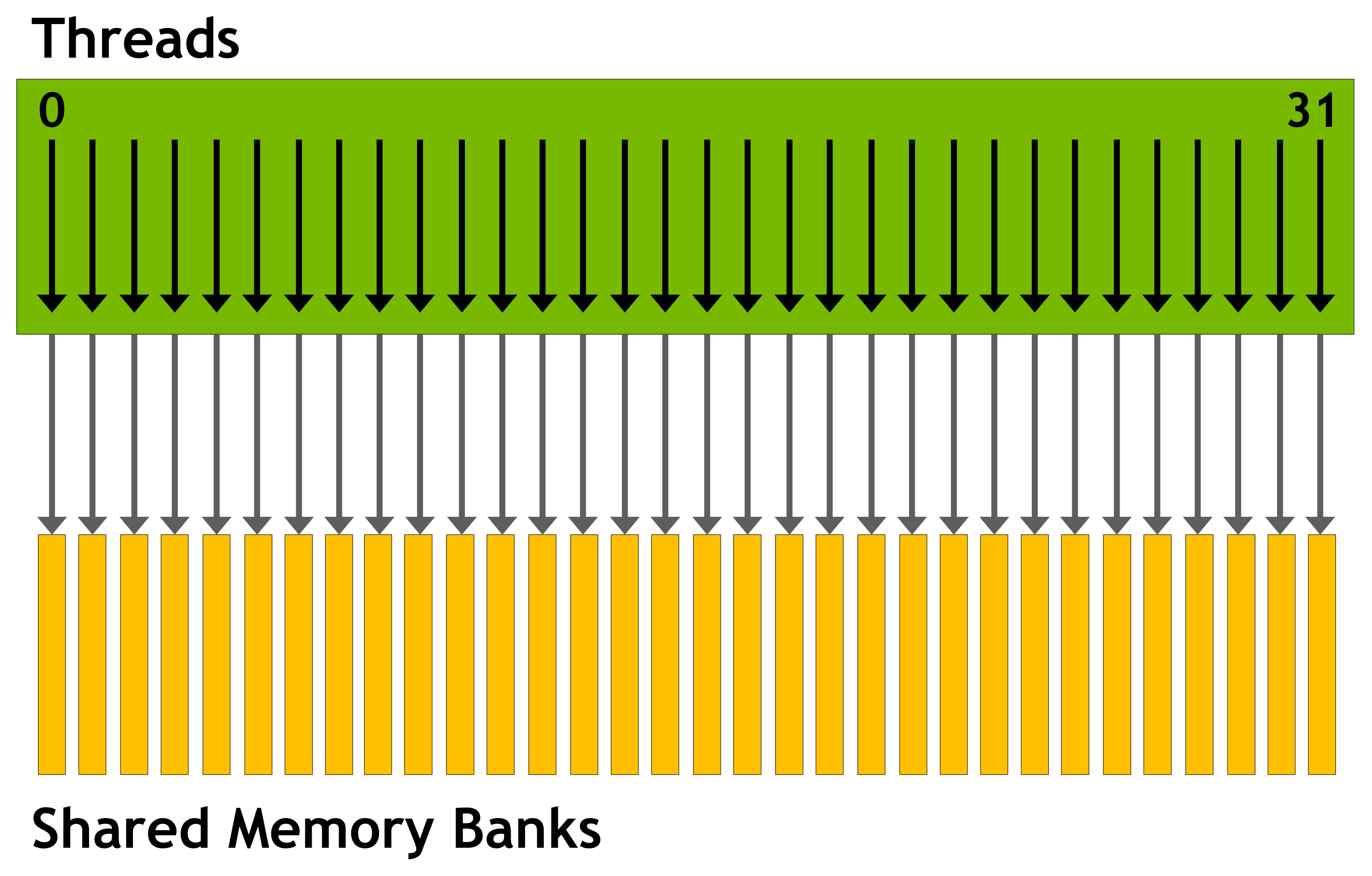
- Bank conflicts occur when threads in the **same phase** want to access the **same bank**.

Coalesced access

(No bank conflicts)

`shmem[threadIdx.x] = data[tid]`

4-byte data

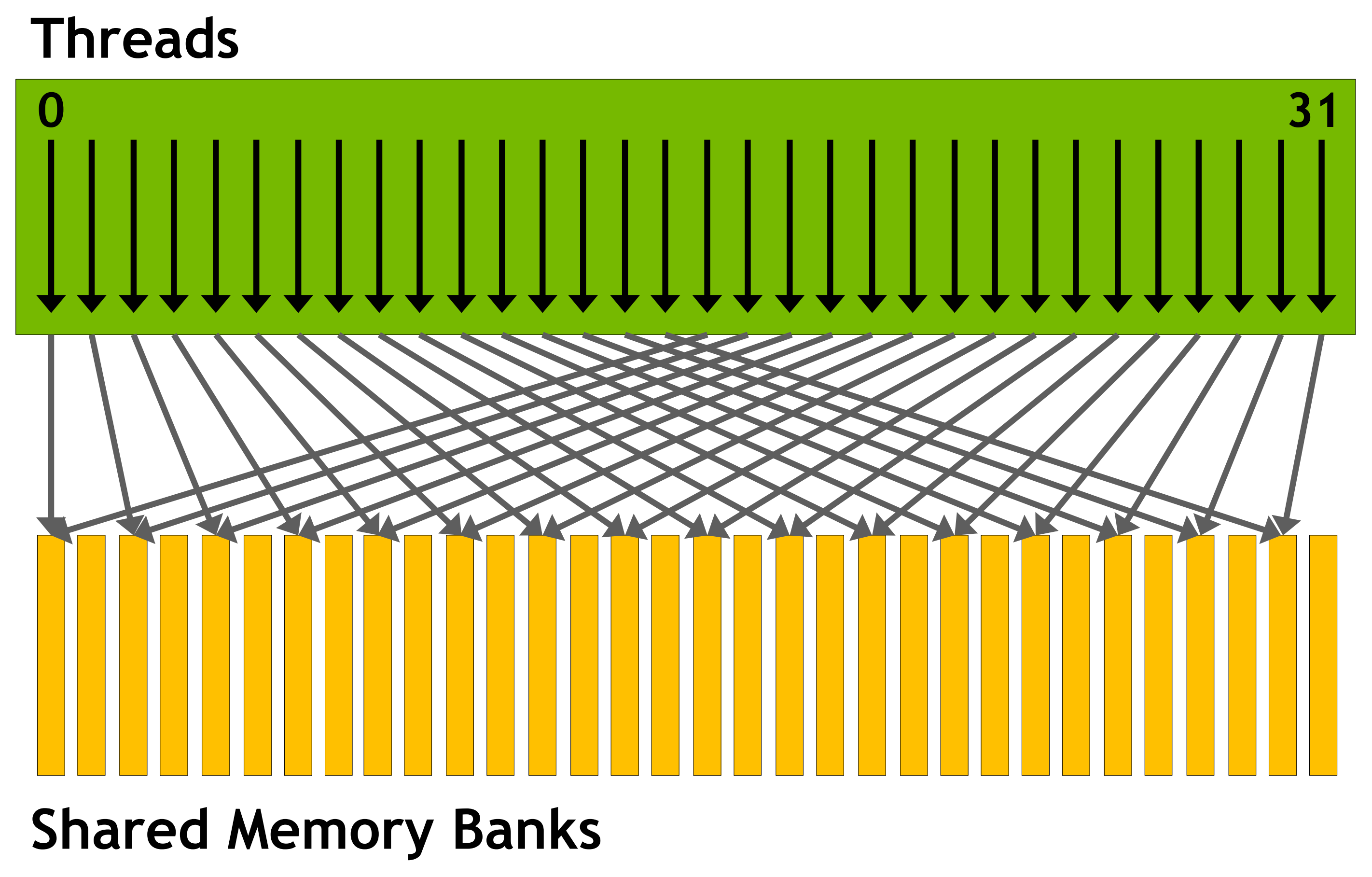


Conflict access

(2-way bank conflicts)

`shmem[threadIdx.x * 2] = data[tid]`

4-byte data

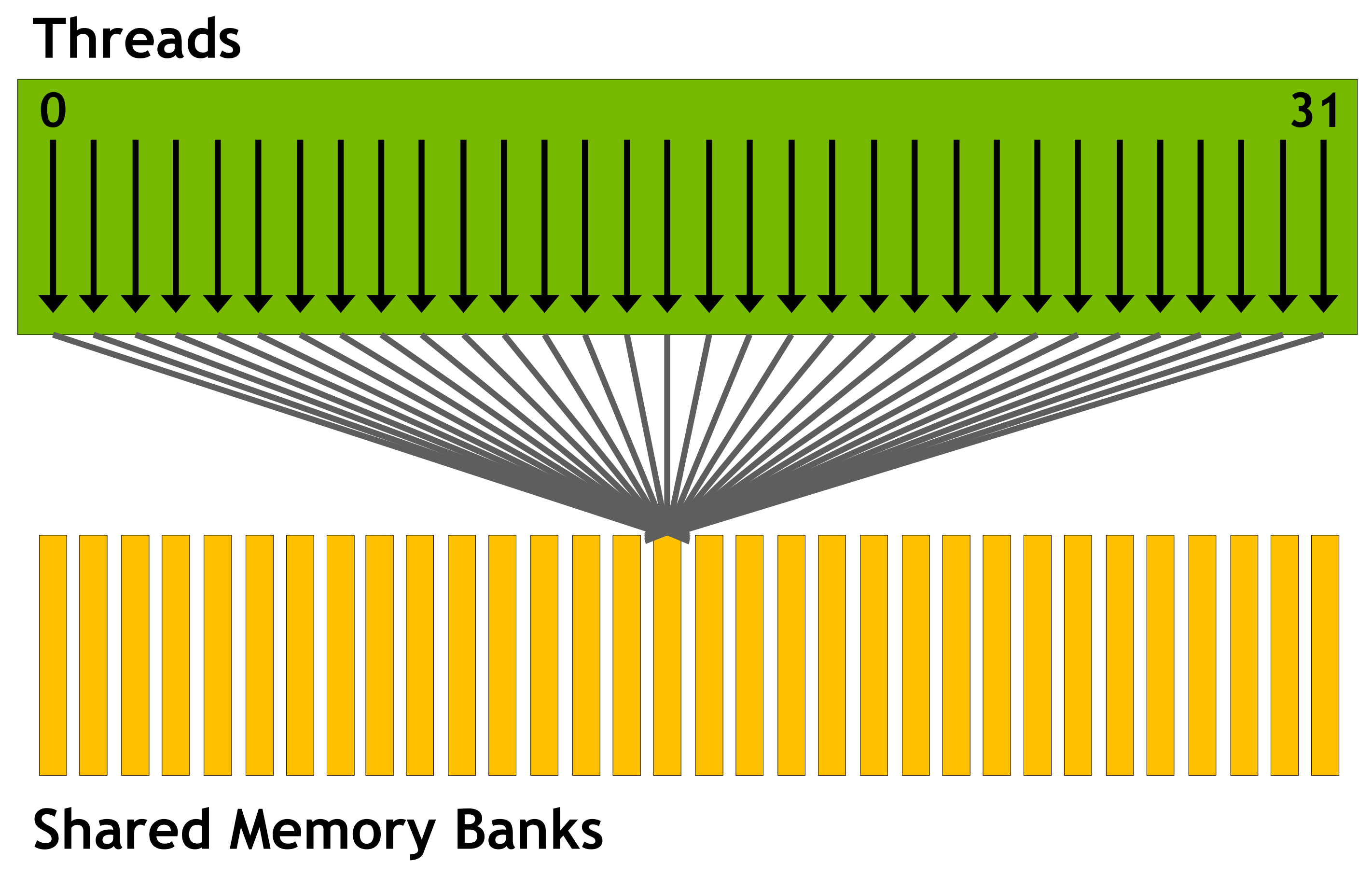


Broadcast access

(No bank conflicts)

`data = shmem[0]`

4-byte data

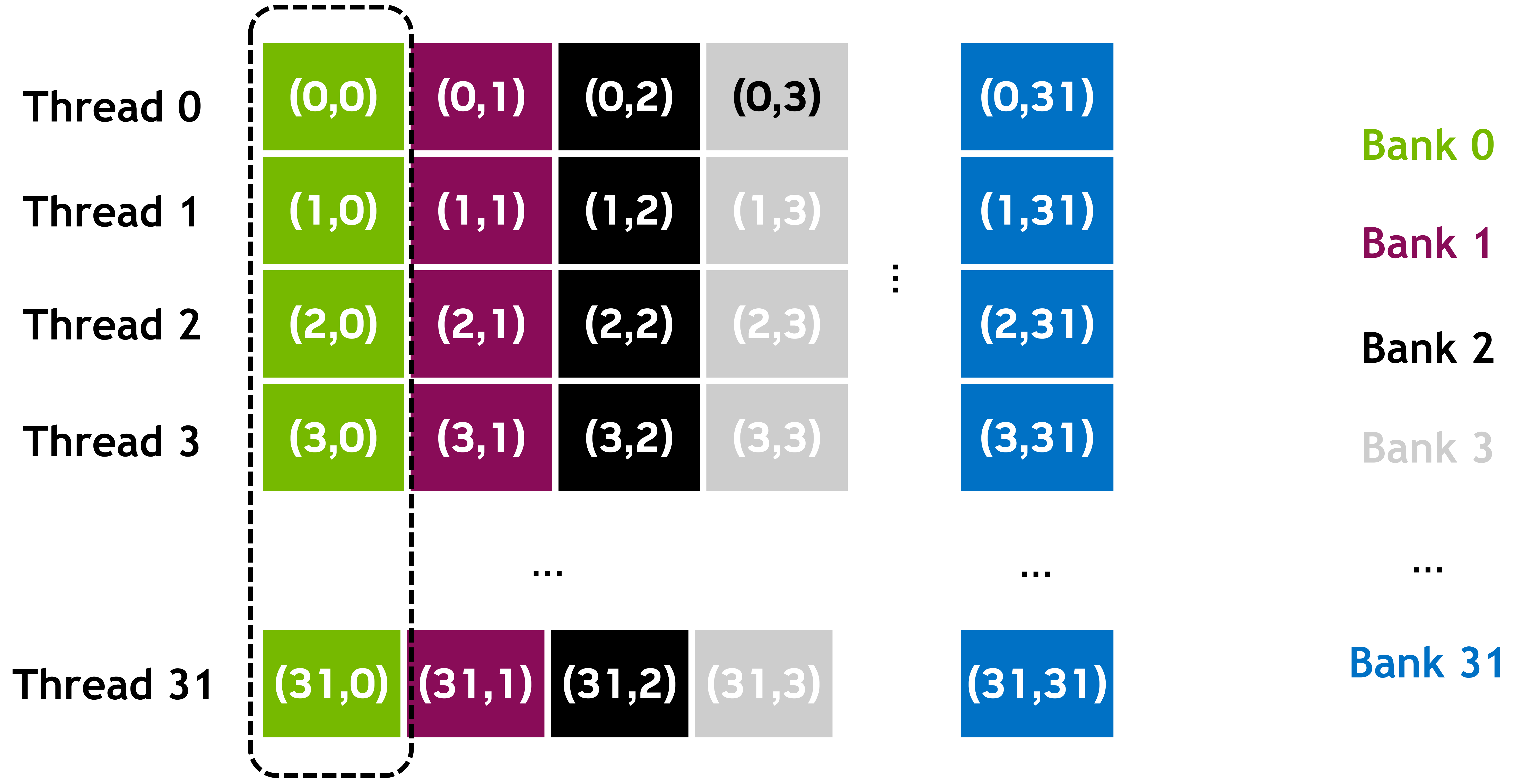


Bank Conflicts

Example

- 32 x 32 array of floats in shared memory
  - 4-byte data, 1 array element per bank
  - Row-major layout
  - 2D thread block

- **Access pattern:**
  - `idx := threadIdx.x*32 + threadIdx.y`
  - 32-way bank conflicts



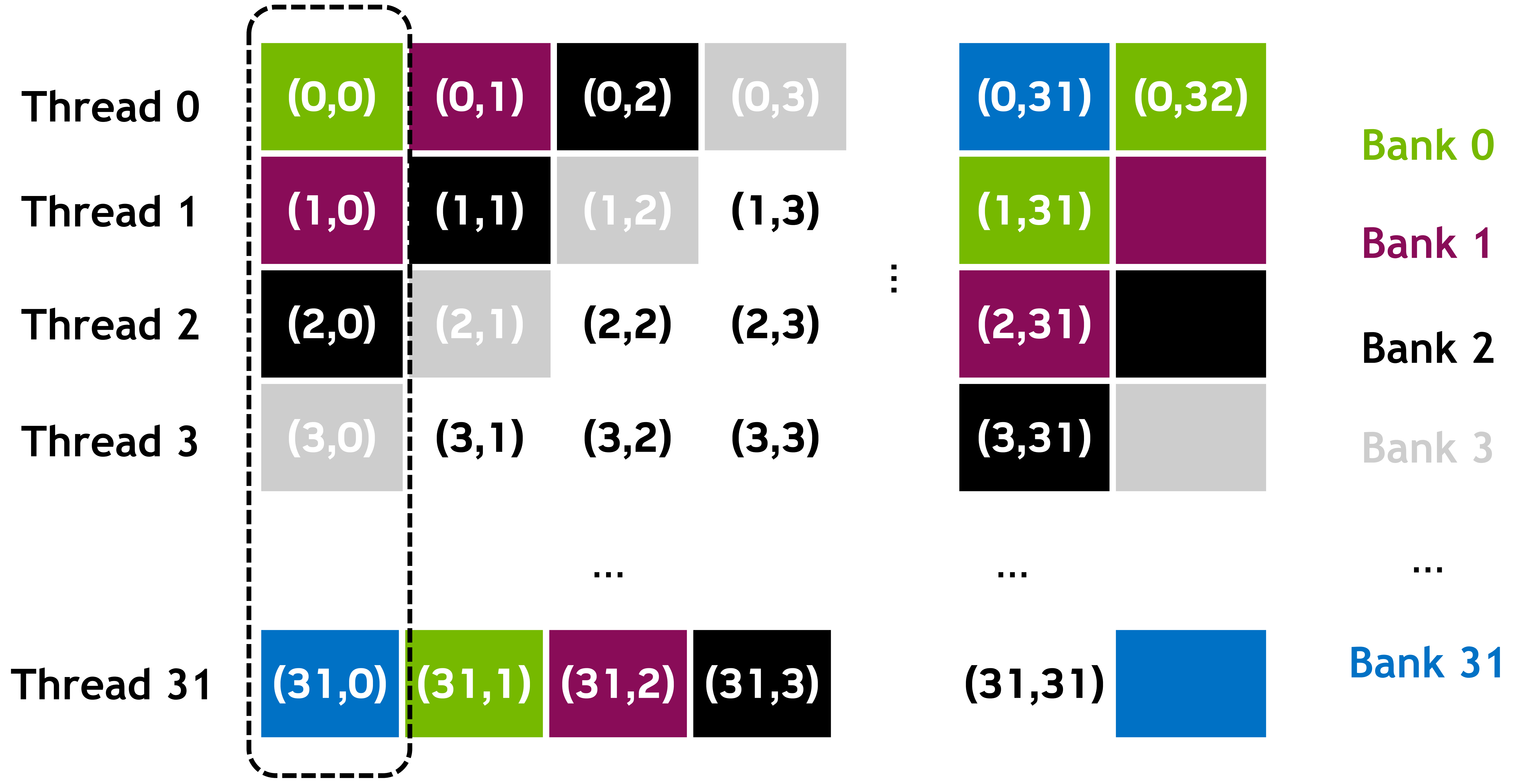
All threads in a warp access the same bank!

Resolving Bank Conflicts

Padding

- 32 x **33** array of floats in shared memory
  - 4-byte data, 1 array element per bank
  - Row-major layout
  - 2D thread block

- **Access pattern:**
  - `idx := threadIdx.x*33 + threadIdx.y`
  - No conflicts!



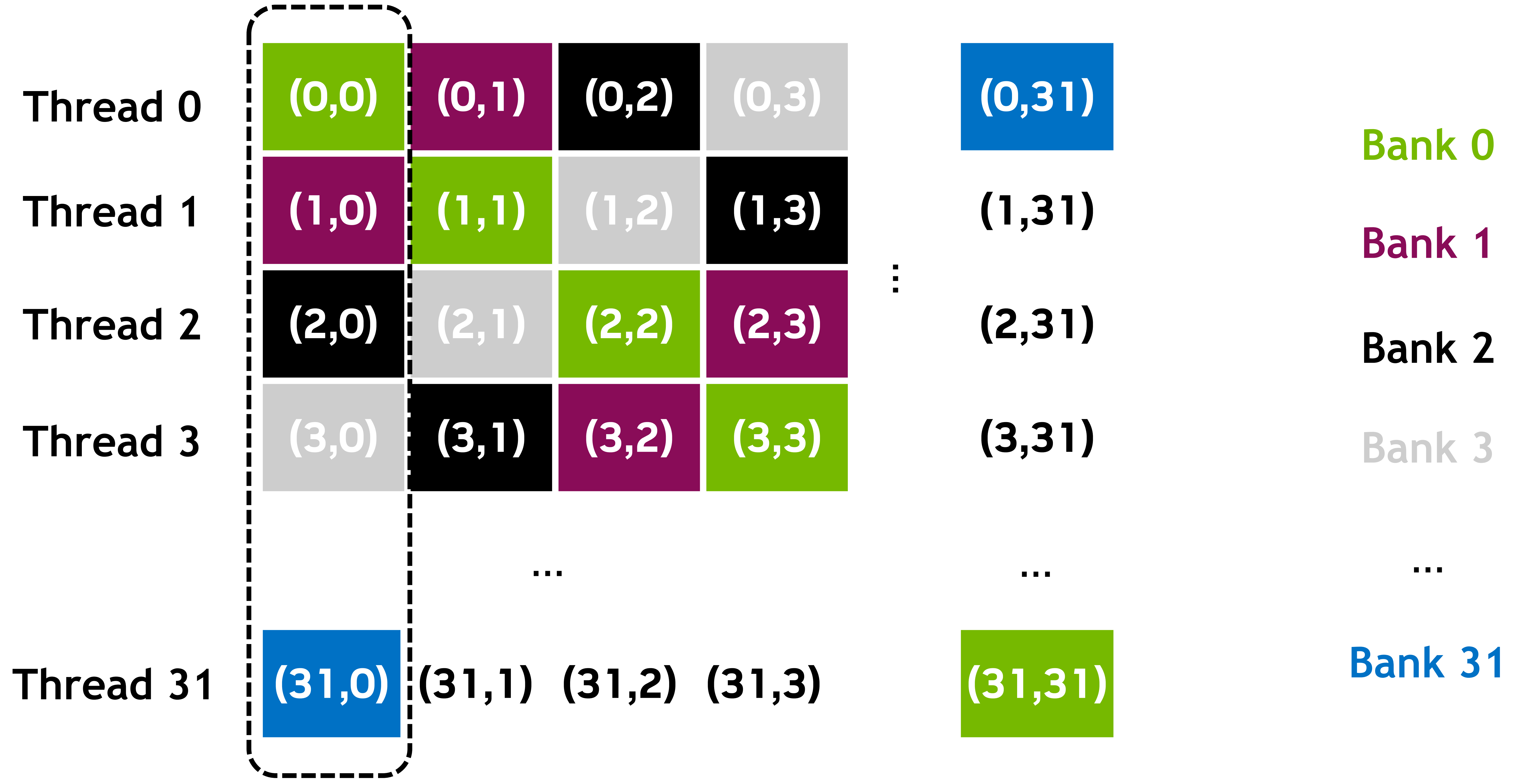
Each thread in a warp accesses a distinct bank!

Resolving Bank Conflicts

Swizzling

- 32 x 32 array of floats in shared memory
  - 4-byte data, 1 array element per bank
  - Row-major layout
  - 2D thread block

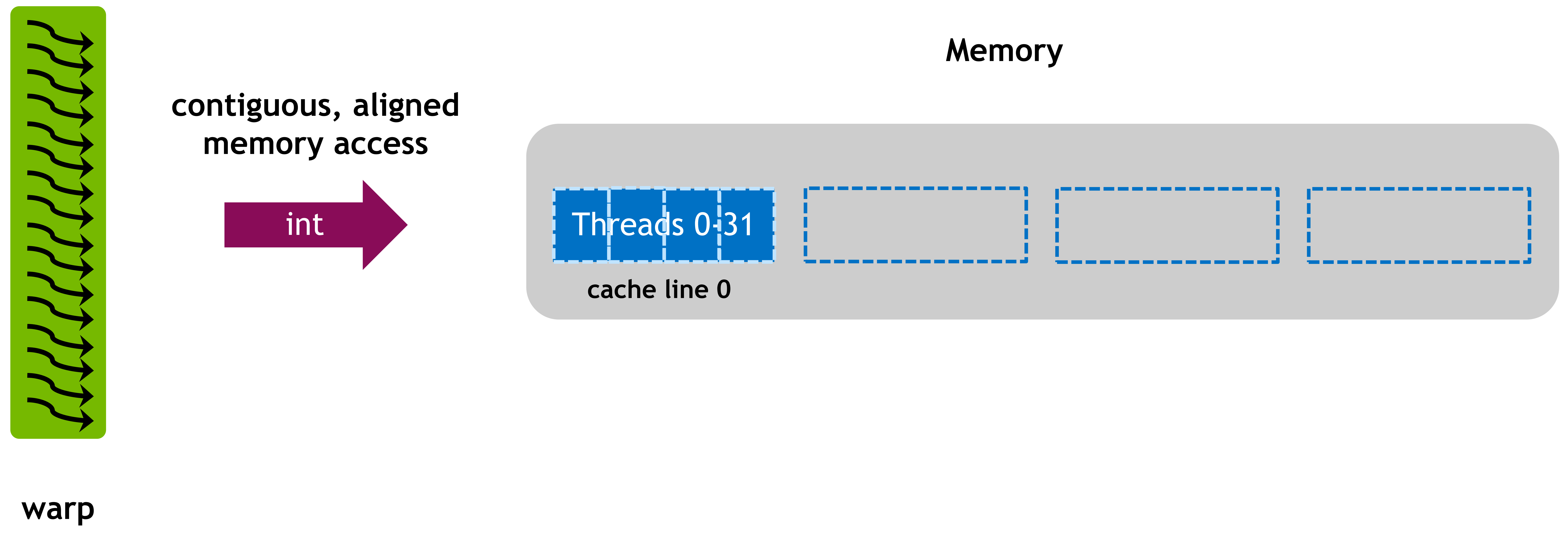
- **Access pattern:**
  - `idx = threadIdx.x*32 + threadIdx.y ^ threadIdx.x`
  - No conflicts!
  - No shared memory wasted!



Each thread in a warp accesses a distinct bank!

Vectorized Memory Accesses

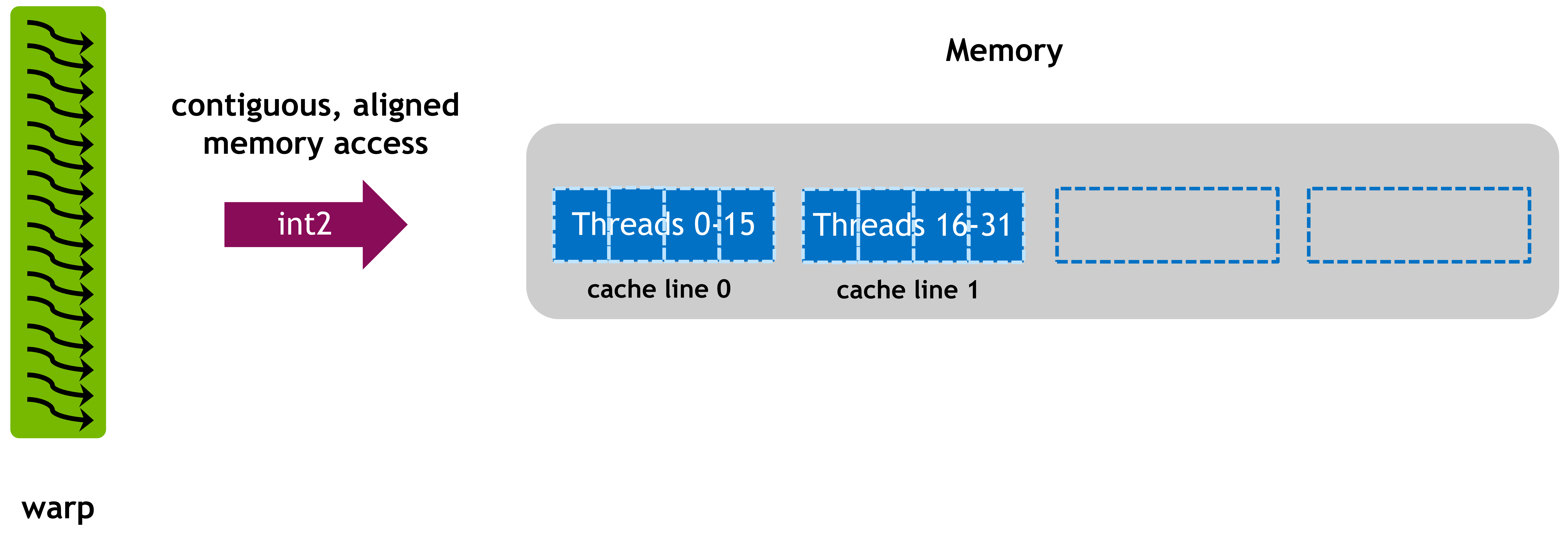
Multi-word as well as multi-thread



Fills 1 cache line in a single fetch.

Vectorized Memory Accesses

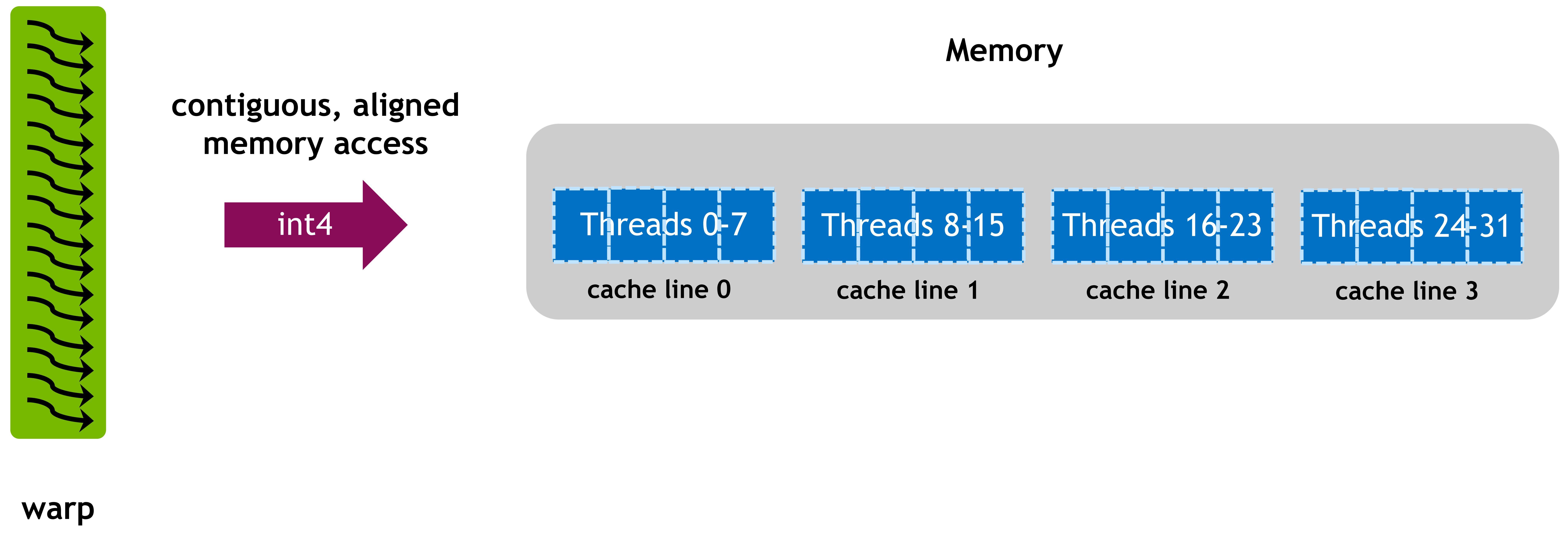
Multi-word as well as multi-thread



Fills 2 cache lines in a single fetch.

Vectorized Memory Accesses

Multi-word as well as multi-thread



Fills 4 cache lines in a single fetch.

Vectorized Memory Accesses

Multi-thread, multi-word

- Vectorized **global** and **shared memory** accesses.
  - Require aligned data.
  - 64- or 128-bit width.

- Less executed instructions!
- More bytes in-flight!

- Approaches to enable vectorization:
  1) By using vector data types, e.g., `float2`, `float4`.
  2) Explicitly by casting to vector pointers.
     1) Proper alignment required.

```
// Using vector data types
__global__
void copy(const float2 * __restrict__ in,
          float2 * __restrict__ out,
          int N)
{
  auto grid = cg::this_grid();
  int tid = grid.thread_rank();
  int stride = grid.size();

  for (int i = tid; i < N / 2; i += stride) {
    out[i] = in[i];
    // Same as:
    // out[i].x = in[i].x;
    // out[i].y = in[i].y;
  }
}
```

Vectorized Memory Accesses

Performance Analysis

- **Experimental setup:**
  - NVIDIA H100 SXM, 1980 MHz
  - Problem size = 2^28
  - Thread block size = 256

Implementation	Main Memory Bandwidth Utilization (%)	GPU Time (ms)
float	60.62	1.033
float2	84.34	0.737
float4	88.82	**0.706**

Maximizing Memory Throughput

General guidelines

Global memory

- Strive for aligned and coalesced accesses within a warp.
- Maximize bytes in-flight to saturate memory bandwidth.
  - Process several elements per thread.
  - Use vectorized loads/stores.
  - Launch enough threads to maximize throughput.

L1 and L2 caches

- Cache blocking difficult, but not impossible.
- Rely on caches when you don’t have a choice.

Shared memory

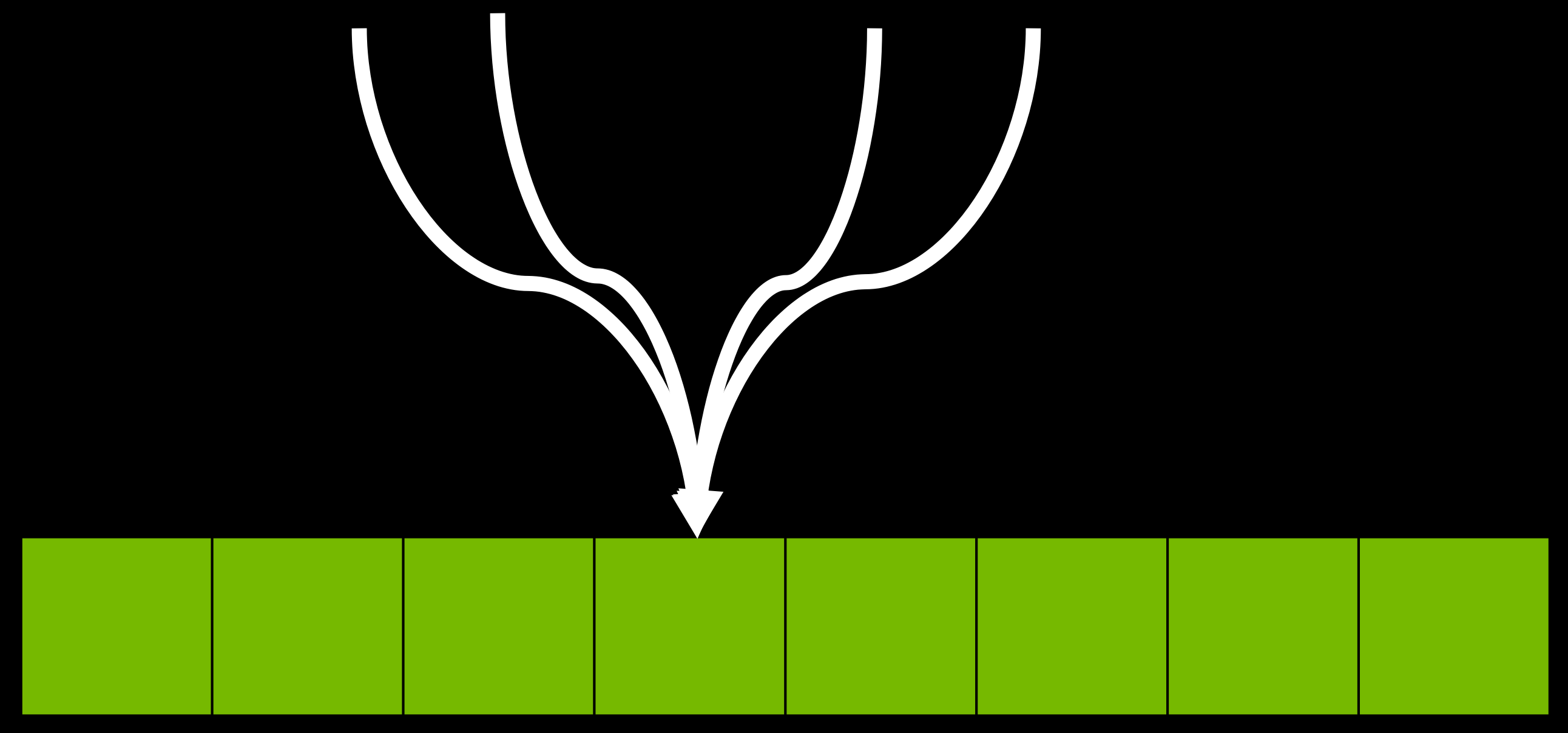
- Use it to reduce global memory traffic.
- Strive to avoid bank conflicts.
- Use vectorized loads/stores.

Atomics

Using Atomics Efficiently

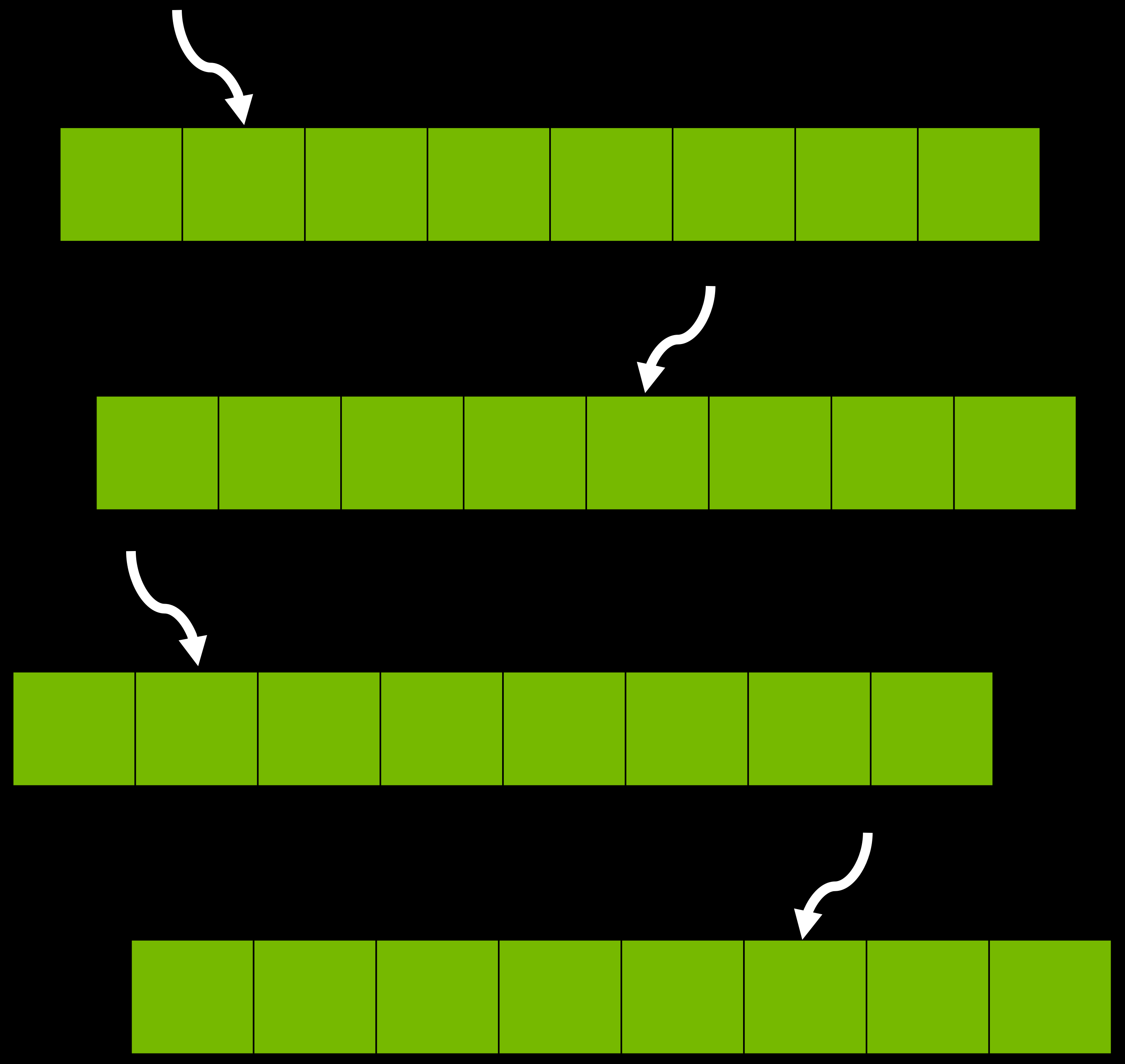
Access Patterns

Same address

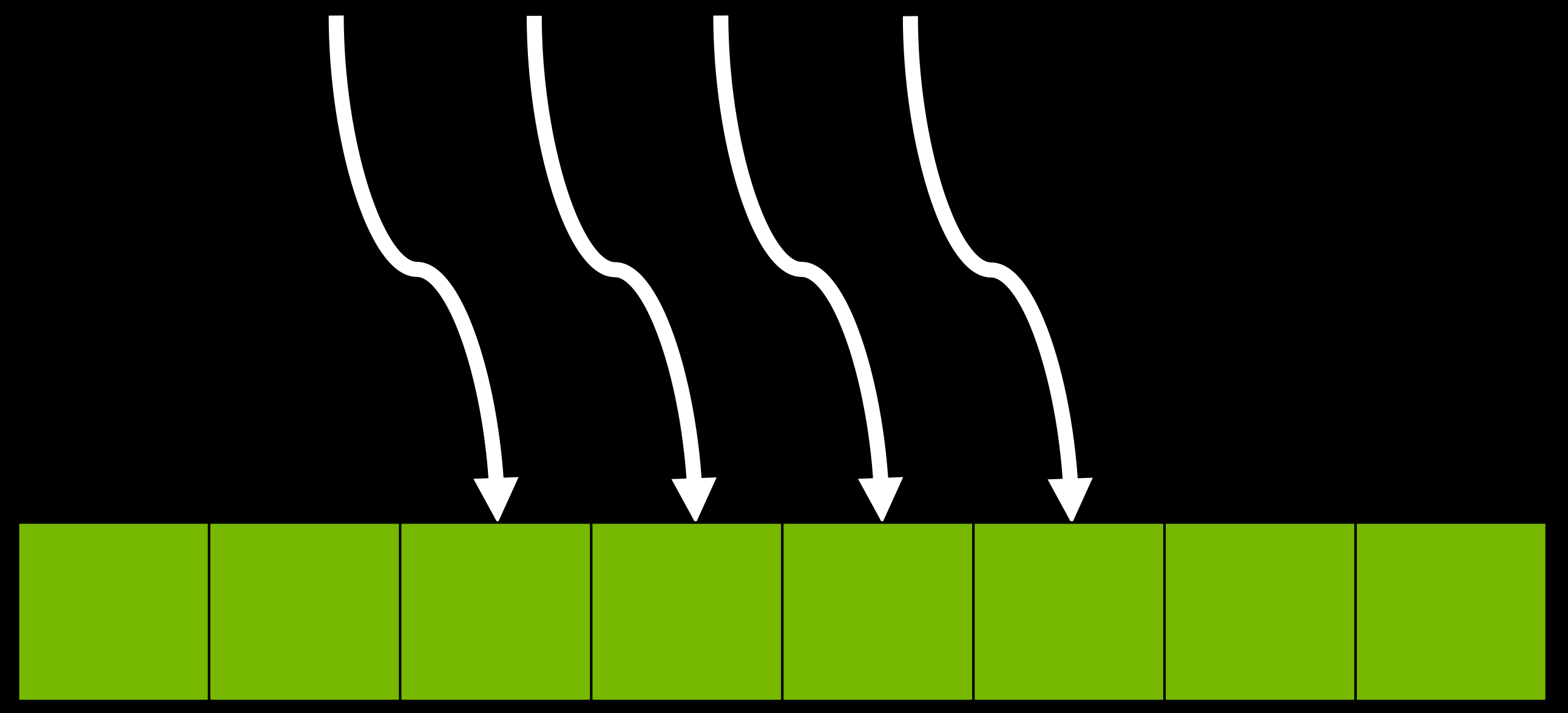


Serialized!
Least efficient access pattern.

Scattered



Coalesced



Most efficient access pattern.

Using Atomics Efficiently

Example #1: find the maximum value of an array

- **Problem description:** given an input array, find the maximum element in the array.
- **Naïve implementation:** every thread find its local maximum and then atomically updates the global maximum.
    - N / elements_per_thread same-address global atomics.

```
__global__
void find_max(const int * __restrict__ in, int *max, int N)
{
  int grid_tid = cg::this_grid().thread_rank();
  int grid_stride = cg::this_grid().num_threads();

  // Find my local maximum
  int local_max = INT_MIN;
  for (int i = grid_tid; i < N; i += grid_stride) {
    if (in[i] > local_max)
      local_max = in[i];
  }

  // Atomically update the global max
  atomicMax(max, local_max);
}
```

Using Atomics Efficiently

Example #1: find the maximum value of an array

- **Optimization #1**: maintain a block-level max in shared memory.
  - Reduces the number of same-address global atomics by a factor equal to the thread block size.

```
__global__
void find_max(const int * __restrict__ in, int *max, int N)
{
  int grid_tid = cg::this_grid().thread_rank();
  int grid_stride = cg::this_grid().num_threads();
  auto block = cg::this_thread_block();
  int block_tid = block.thread_rank();

  __shared__ int block_max;

  // Find my local maximum
  int local_max = INT_MIN;
  for (int i = grid_tid; i < N; i += grid_stride) {
    if (in[i] > local_max)
      local_max = in[i];
  }

  // Atomically update the block-level max
  atomicMax(&block_max, local_max);
  block.sync();

  // Atomically update the global max
  if (block_tid == 0)
    atomicMax(max, block_max);
}
```

Using Atomics Efficiently

Example #1: find the maximum value of an array

- **Optimization #2**: use a parallel reduction to calculate the block-level max in shared memory.

```
//Assumes a block dimension of 256
__global__
void find_max(const int * __restrict__ in, int *max, int N) {
  …
  auto tile = cg::tiled_partition<32>(block);
  extern __shared__ int sdata[];

  // Find my local maximum as before
  // Each thread puts its local max into shared memory
  sdata[block_tid] = thread_max;

  // Block-level reduction
  if (block_tid < 128) {
    if (sdata[block_tid + 128] > thread_max)
      thread_max = sdata[block_tid + 128];
    sdata[block_tid] = thread_max;
  }
  block.sync();
  if (block_tid < 64) {
    if (sdata[block_tid + 64] > thread_max)
      thread_max = sdata[block_tid + 64];
    sdata[block_tid] = thread_max;
  }
  block.sync();

  // Warp-level reduction
  if (tile.meta_group_rank() == 0) {
    thread_max = cg::reduce(tile, thread_max,
                            cg::greater<int>());
  }

  if (block_tid == 0)
    atomicMax(max, thread_max);
}
```

Performance Analysis

- **Experimental setup:**
  - NVIDIA H100 SXM, 1980 MHz
  - Problem size = 2^28
  - Uniform distribution (-50, 50).

Implementation	Thread Block Size	GPU Time (ms)
global atomics	256	6.839
shared memory atomics	256	1.334
shared memory reduction	256	**1.066**

Using Atomics Efficiently

Example #2: vector update

- **Problem description:** a = a + b * c

```
__global__
void vector(const float * __restrict__ b,
            const float * __restrict__ c,
            float * __restrict__ a,
            int N) {
  int grid_tid = cg::this_grid().thread_rank();
  int grid_stride = cg::this_grid().num_threads();

  for (int i = grid_tid; i < N; i += grid_stride) {
    a[i] += b[i] * c[i];
  }
}
```

Memory operations = 3 reads + 1 write

Using Atomics Efficiently

Example #2: vector update

- **Optimization**: use atomics to update each vector element even though atomicity is not required.
  - Offload some of the computation to the L2 cache.
  - Saves reading the value of `a[i]` in registers.
  - This reduces the latency to compute each element of the vector.
  - Can result in **more bytes in-flight**!

```
__global__
void vector(const float * __restrict__ b,
            const float * __restrict__ c,
            float * __restrict__ a,
            int N) {
  int grid_tid = cg::this_grid().thread_rank();
  int grid_stride = cg::this_grid().num_threads();

  for (int i = grid_tid; i < N; i += grid_stride) {
    atomicAdd(&a[i], b[i] * c[i]);
  }
}
```

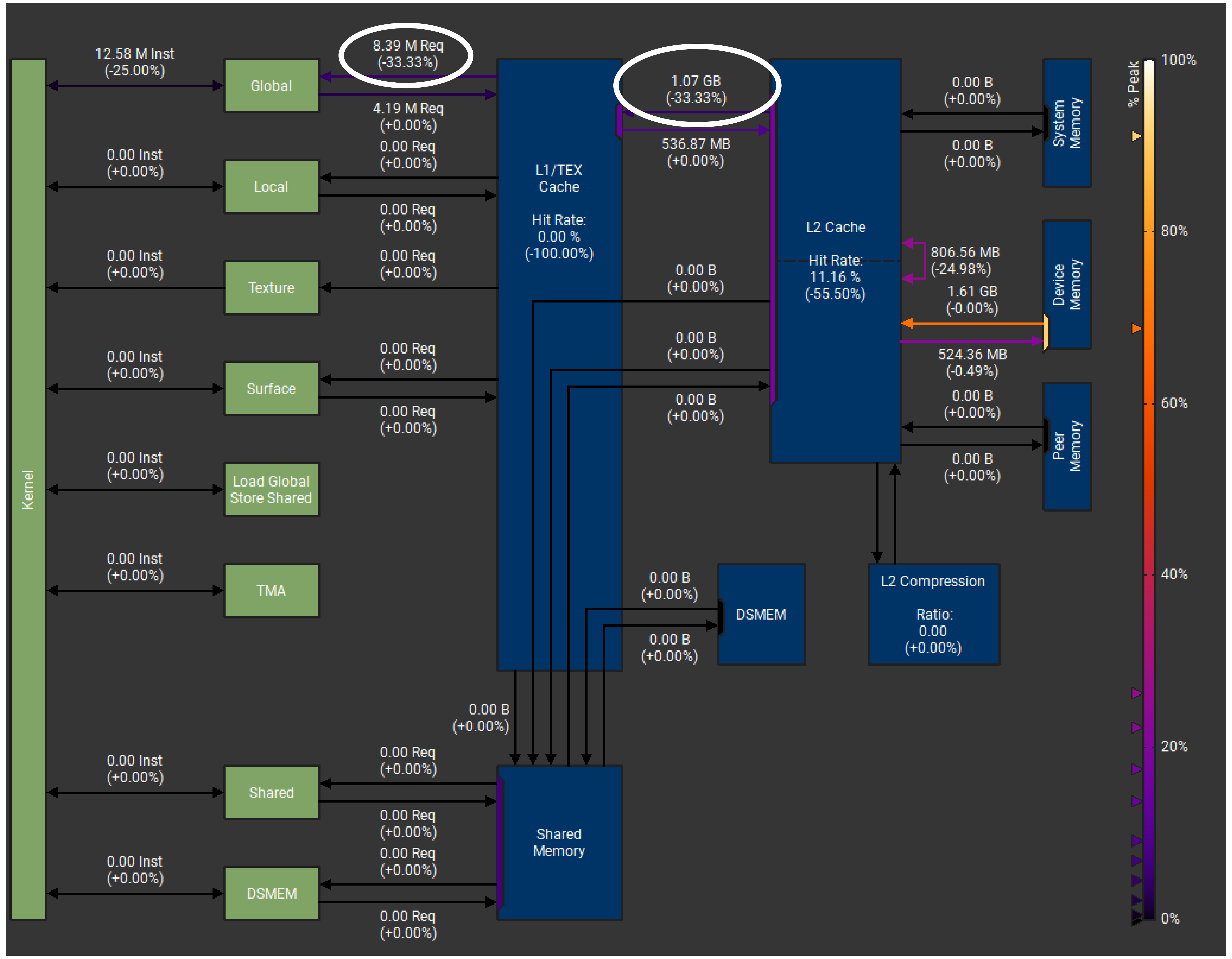
Memory operations = 2 reads + 1 write

Performance Analysis

NVIDIA H100 SXM

- **Experimental setup:**
  - NVIDIA H100 SXM, 1980 MHz
  - Problem size = 2^27

NCU Memory Chart (Transfer Size)

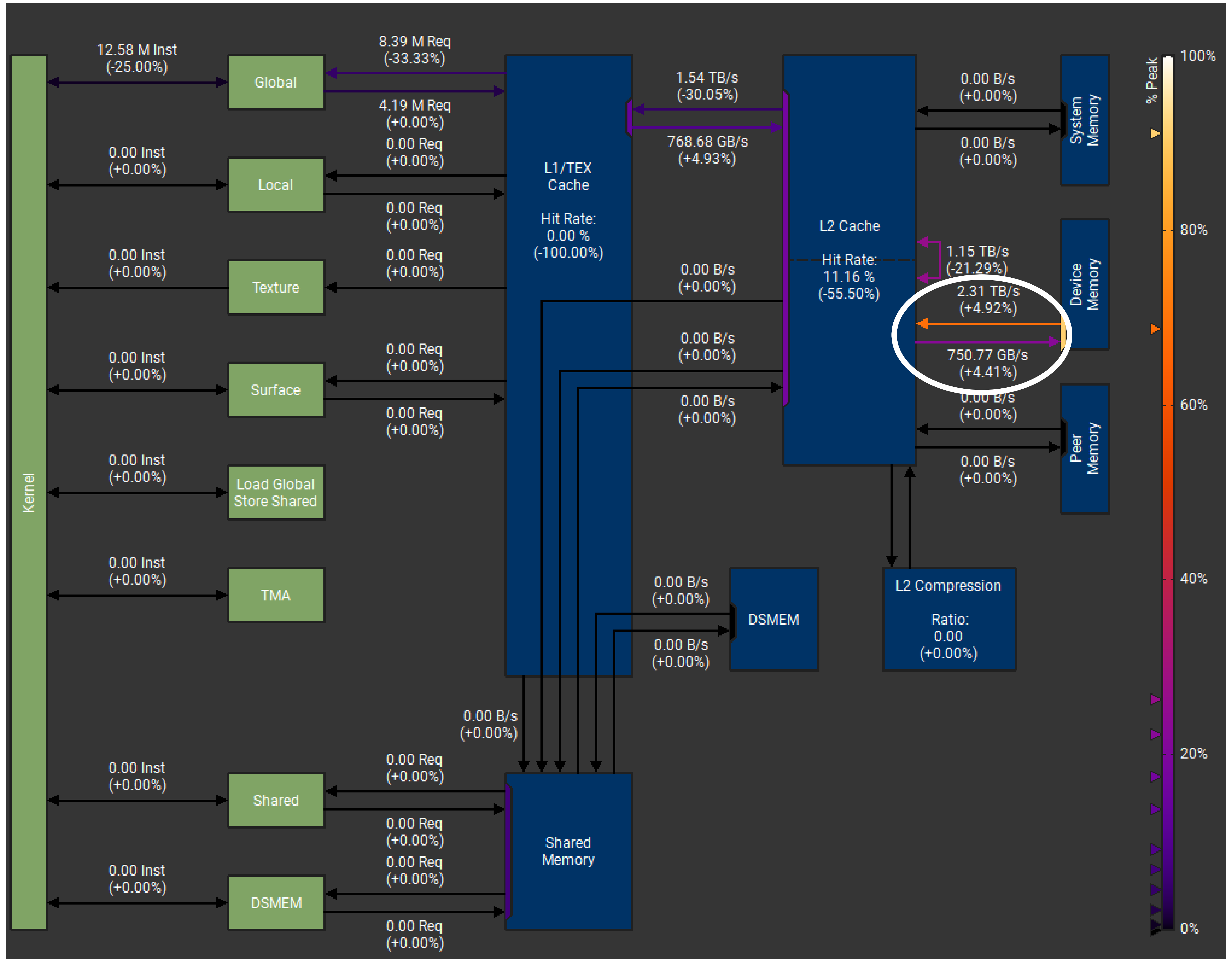


Performance Analysis

NVIDIA H100 SXM

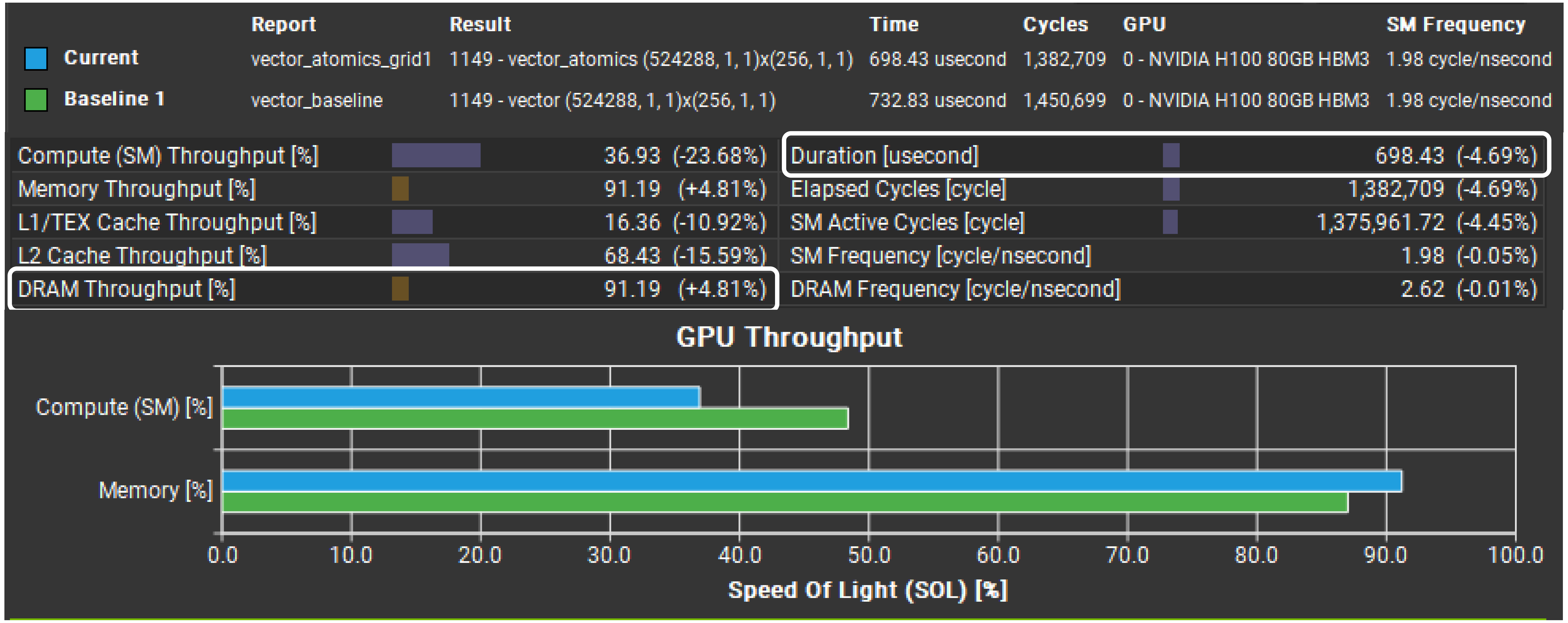
- **Experimental setup:**
  - NVIDIA H100 SXM, 1980 MHz
  - Problem size = 2^27

NCU Memory Chart (Throughput)



Performance Analysis

NVIDIA H100 SXM



		Report	Result	Time	Cycles	GPU	SM Frequency
🟦	**Current**	vector_atomics_grid1	1149 - vector_atomics (524288, 1, 1)x(256, 1, 1)	698.43 usecond	1,382,709	0 - NVIDIA H100 80GB HBM3	1.98 cycle/nsecond
🟩	**Baseline 1**	vector_baseline	1149 - vector (524288, 1, 1)x(256, 1, 1)	732.83 usecond	1,450,699	0 - NVIDIA H100 80GB HBM3	1.98 cycle/nsecond

Compute (SM) Throughput [%]	36.93 (-23.68%)	Duration [usecond]	698.43 (-4.69%)
Memory Throughput [%]	91.19 (+4.81%)	Elapsed Cycles [cycle]	1,382,709 (-4.69%)
L1/TEX Cache Throughput [%]	16.36 (-10.92%)	SM Active Cycles [cycle]	1,375,961.72 (-4.45%)
L2 Cache Throughput [%]	68.43 (-15.59%)	SM Frequency [cycle/nsecond]	1.98 (-0.05%)
DRAM Throughput [%]	91.19 (+4.81%)	DRAM Frequency [cycle/nsecond]	2.62 (-0.01%)

- ~**5%** increase in memory throughput translates into a corresponding reduction in execution time. **Why?**
  - This kernel is DRAM bandwidth bound.

Summary

Which optimizations to focus on?

Solving the bottlenecks

- **Compute bound**
  - Reduce instruction count.
    - E.g., use vector loads/stores.
  - Use tensor cores.
  - Use lower precision arithmetic, fast math intrinsics.
- **Bandwidth bound**
  - Reduce the amount of data transferred.
    - Optimize memory access patterns.
    - Lower precision datatypes.
    - Kernel fusion.
- **Latency bound**
  - Increase number of instructions and memory accesses in-flight.
  - Increase parallelism, occupancy.

Additional Resources

- **CUDA best practices guide:** https://docs.nvidia.com/cuda/cuda-c-best-practices-guide/
- **CUDA samples:** https://github.com/NVIDIA/cuda-samples

- **GTC’24 sessions:** https://www.nvidia.com/gtc/sessions/performance-optimization/
  - Advanced Performance Optimization in CUDA [S62192]
  - Performance Optimization for Grace CPU Superchip [S62275]
  - Grace Hopper Superchip Architecture and Performance Optimizations for Deep Learning Applications [S61159]
  - Multi GPU Programming Models for HPC and AI [S61339]
  - More Data, Faster: GPU Memory Management Best Practices in Python and C++ [S62550]
  - Harnessing Grace Hopper's Capabilities to Accelerate Vector Database Search [S62339]
  - From Scratch to Extreme: Boosting Service Throughput by Dozens of Times with Step-by-Step Optimization [S62410]

NVIDIA®