Fundamental CUDA Optimization

Guillaume Barnier, Solutions Architect, Energy
gbarnier@nvidia.com

NVIDIA

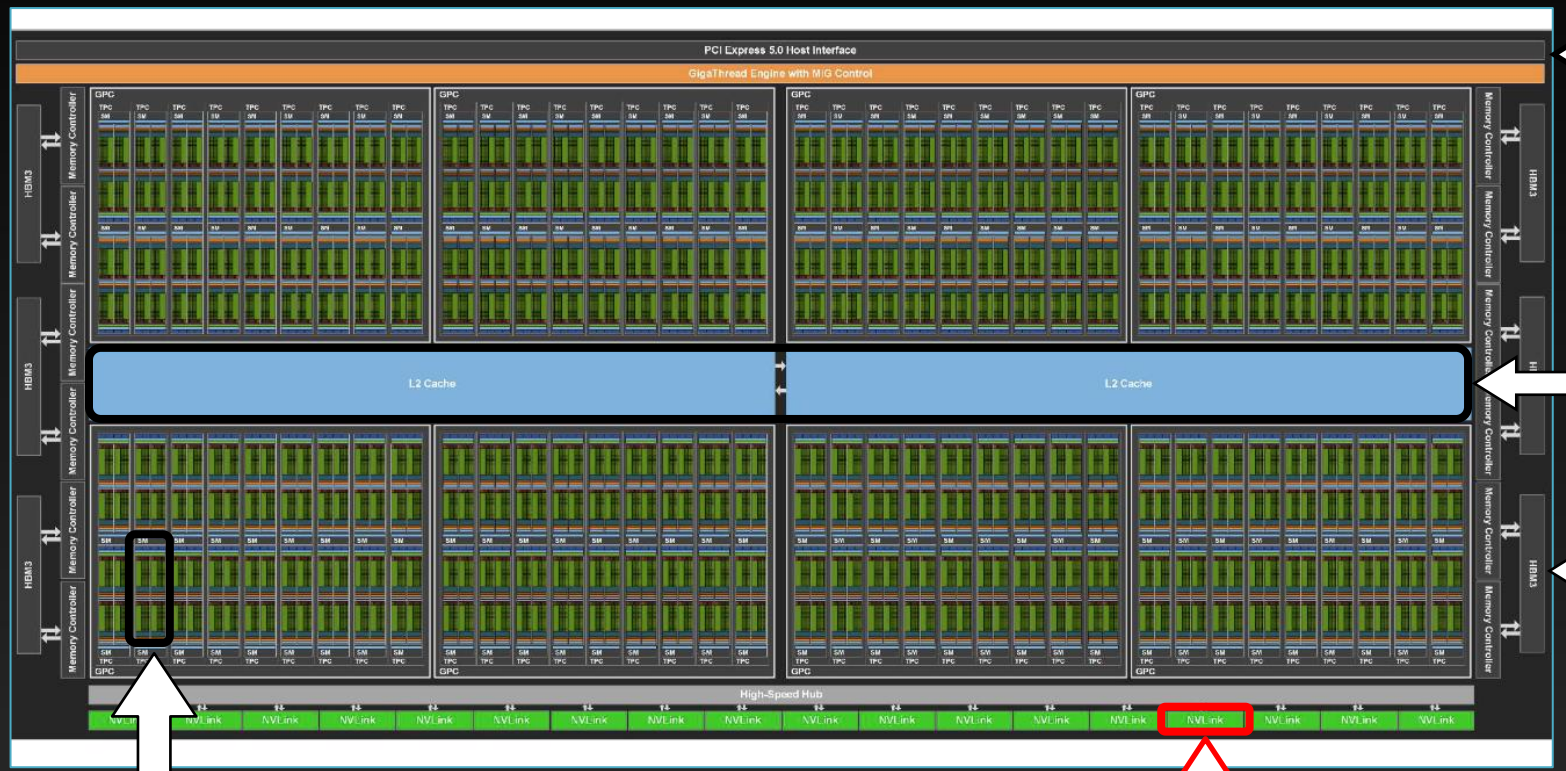
Outline

- Architecture:
  - Hopper
- Kernel optimizations
  - Launch configuration
  - Global memory throughput
  - Shared memory access

Most concepts in this presentation apply to *any* language or API on NVIDIA GPUs

GPU Overview

NVIDIA H100 SXM



PCIe Gen 5,
128 GB/s
bidirectional

50 MB L2

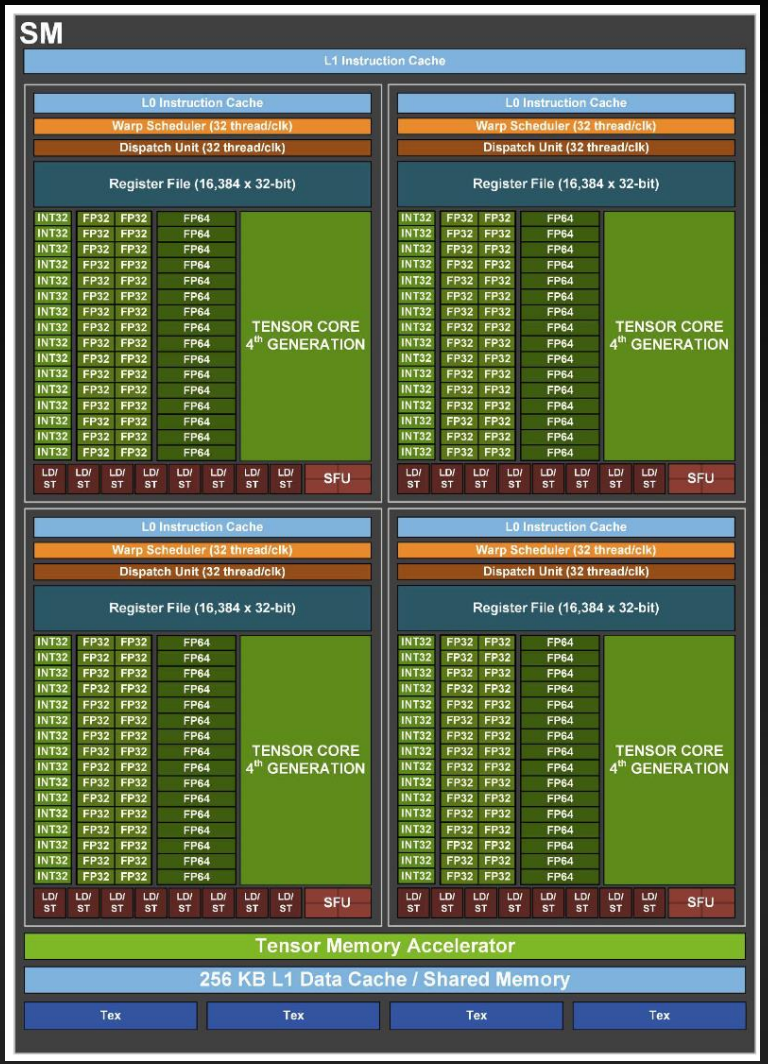
80 GB HBM3,
3.3 TB/s
bidirectional

132 SMs
4th Gen Tensor Cores

4th Gen NVLink,
900 GB/s
bidirectional

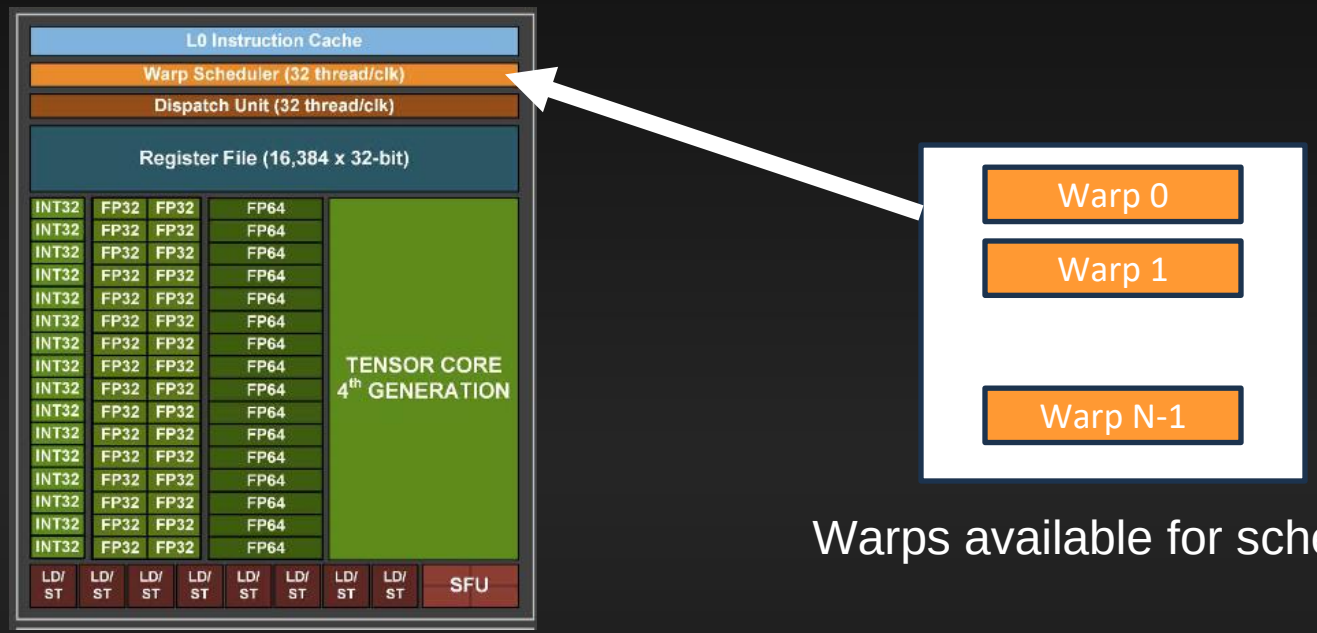
Streaming Multiprocessor (SM)

- SM has **4 sub-partitions**
- 128 FP32 units
- 64 FP64 units
- 64 INT32 units
- 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)



Thread-block execution on a SM

- Threads are co-scheduled within a thread block
- Threads are grouped into warps (32 consecutive threads)
- A warp is the smallest “unit” of work performedby the GPU
- Warp runs in a SIMT fashion in a sub-partition
- Warp context switching is fast. Interleaving warp execution hides latencies
- When a thread-block is launched, warps are preassigned to sub-partitions (and do not migrate)



Execution Model

Software

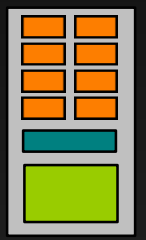
Thread

Thread Block

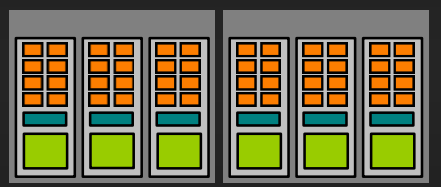
Grid

Hardware

Scalar Processor



Multiprocessor

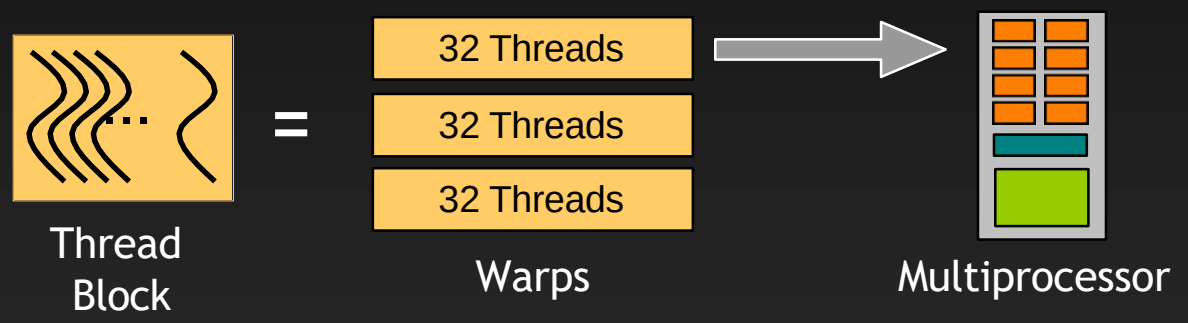


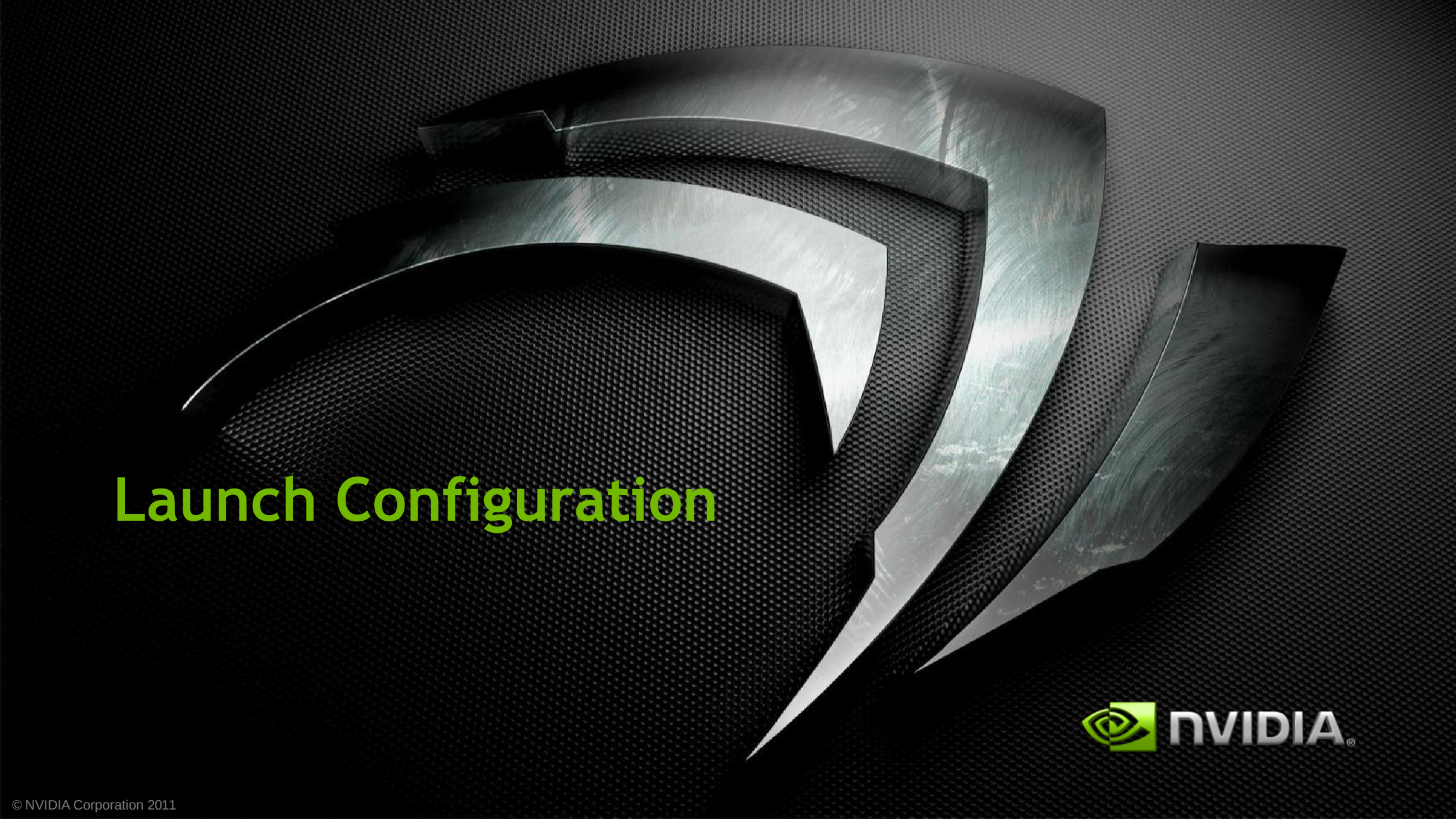
Device

- Threads are executed by scalar processors
- Thread blocks are executed on multiprocessors
- Thread blocks do not migrate
- Several concurrent thread blocks can reside on one multiprocessor, limited by:
  - Shared memory
  - Register file
- Kernel is launched as a grid of thread blocks

Warps

- A thread block consists of 1 to 32 warps (maximum 1024 threads per block)
- Each warp is composed of 32 threads
- A warp is executed physically in parallel (SIMD) on a multiprocessor





Launch Configuration

- Key to understanding:
  - Instructions are issued in order
  - A thread stalls when one of the operands isn't ready:
    - Memory read by itself doesn't stall execution
  - Latency is hidden by switching threads
    - GMEM latency: ~1000 cycles
    - Arithmetic latency: ~4 cycles
- How many threads/threadblocks to launch?
- Conclusion:
  - Need enough threads to hide latency

GPU Latency Hiding

CUDA source code

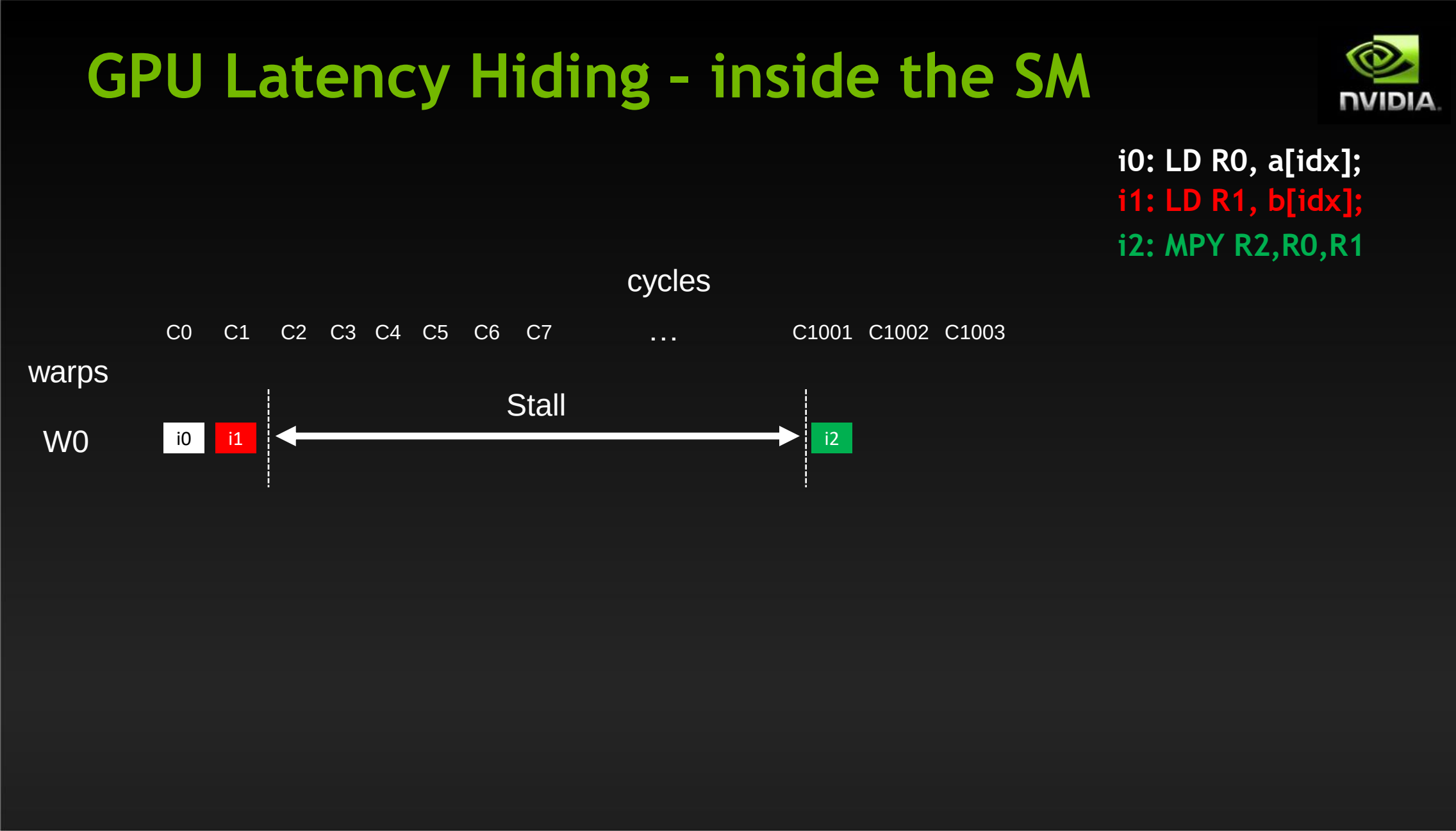
```
// Compute index
int idx = threadIdx.x + blockDim.x * blockIdx.x;

// Perform multiplication
c[idx] = a[idx] * b[idx];
```

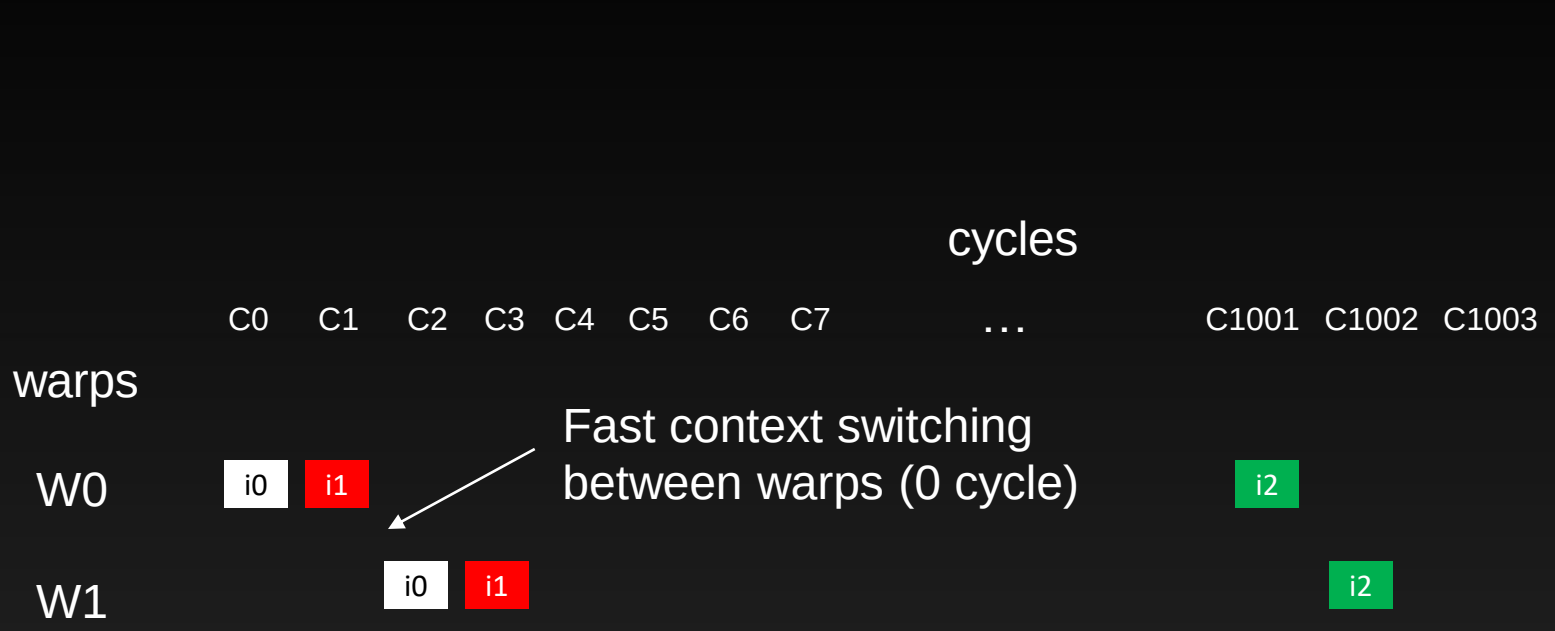
Machine code (SASS)

```
i0: Load R0, a[idx]
i1: Load R1, b[idx]
i2: MPY R2, R1, R0
I3: Store c[idx], R2
```

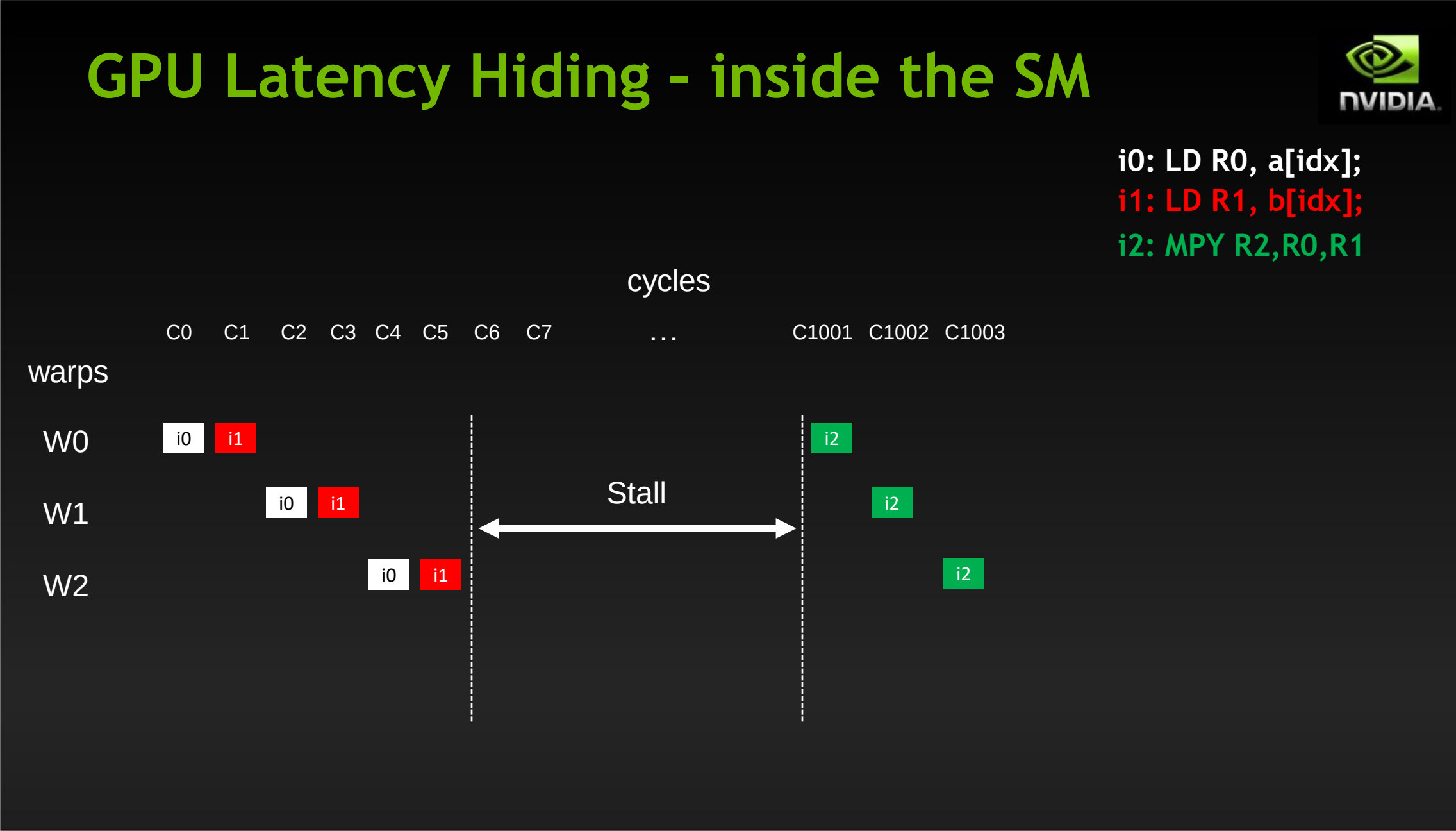
GPU Latency Hiding – inside the SM



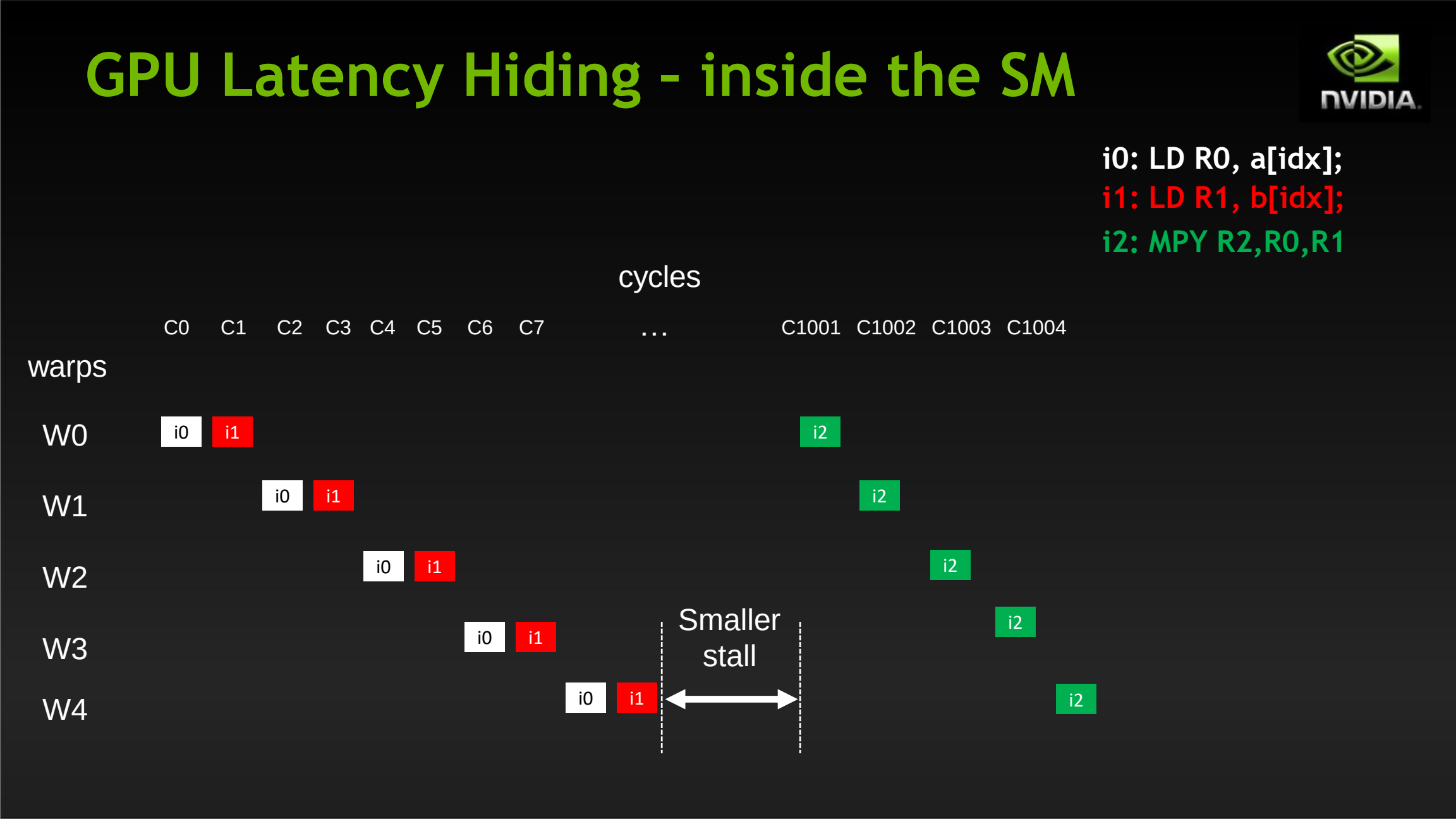
GPU Latency Hiding – inside the SM



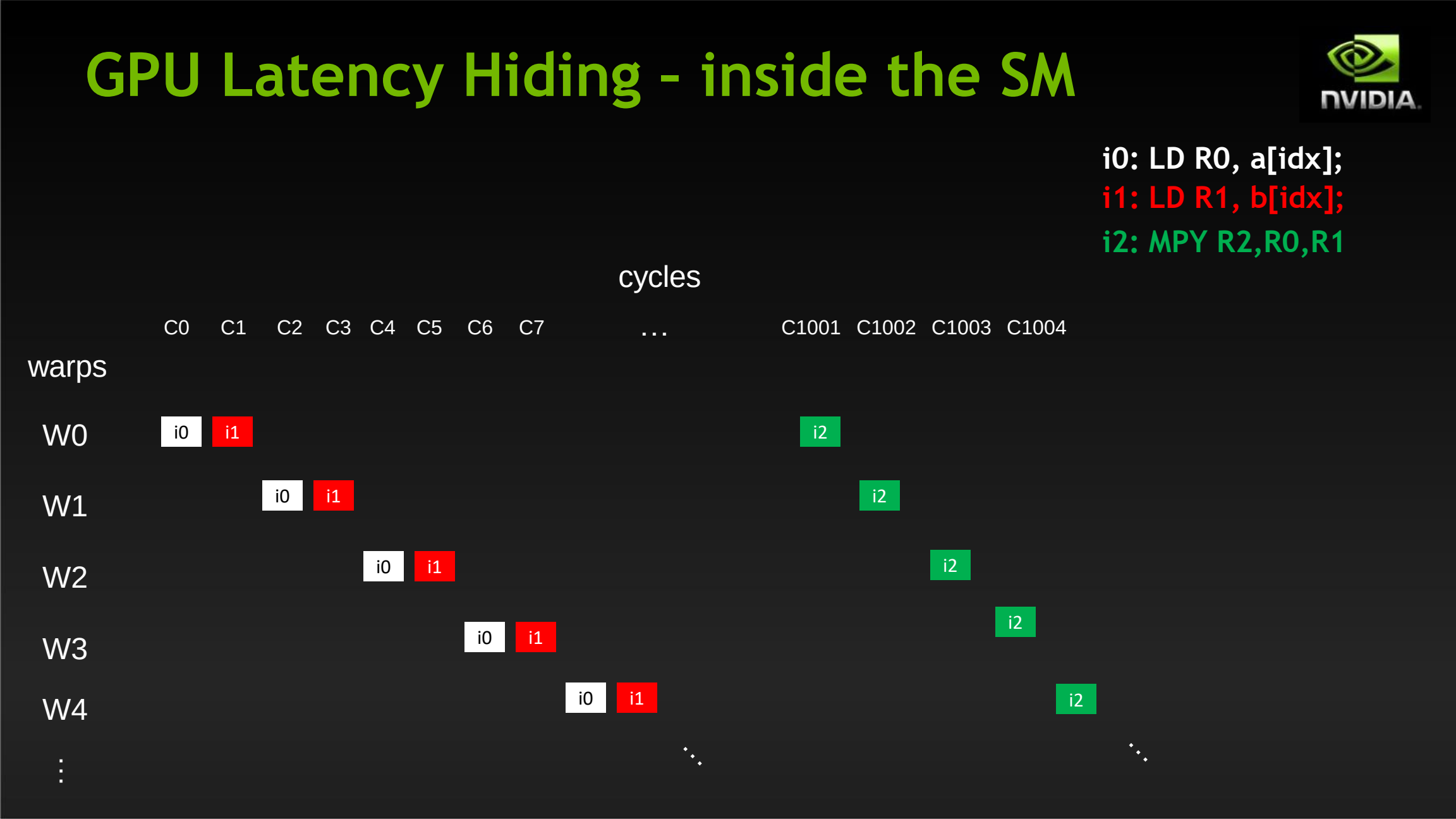
GPU Latency Hiding – inside the SM



GPU Latency Hiding – inside the SM



GPU Latency Hiding – inside the SM



GPU Latency Hiding – inside the SM

- Assumption: addition takes 4 cycles to execute

Case 1

A = B + C + …

- We need 4 warps on scheduler to hide compute latency and fill other slots
- We expose more parallelism with more threads

Case 2

A1 = B1 + C1 + …
A2 = B2 + C2 + …
A3 = B3 + C3 + …
A4 = B4 + C4 + …

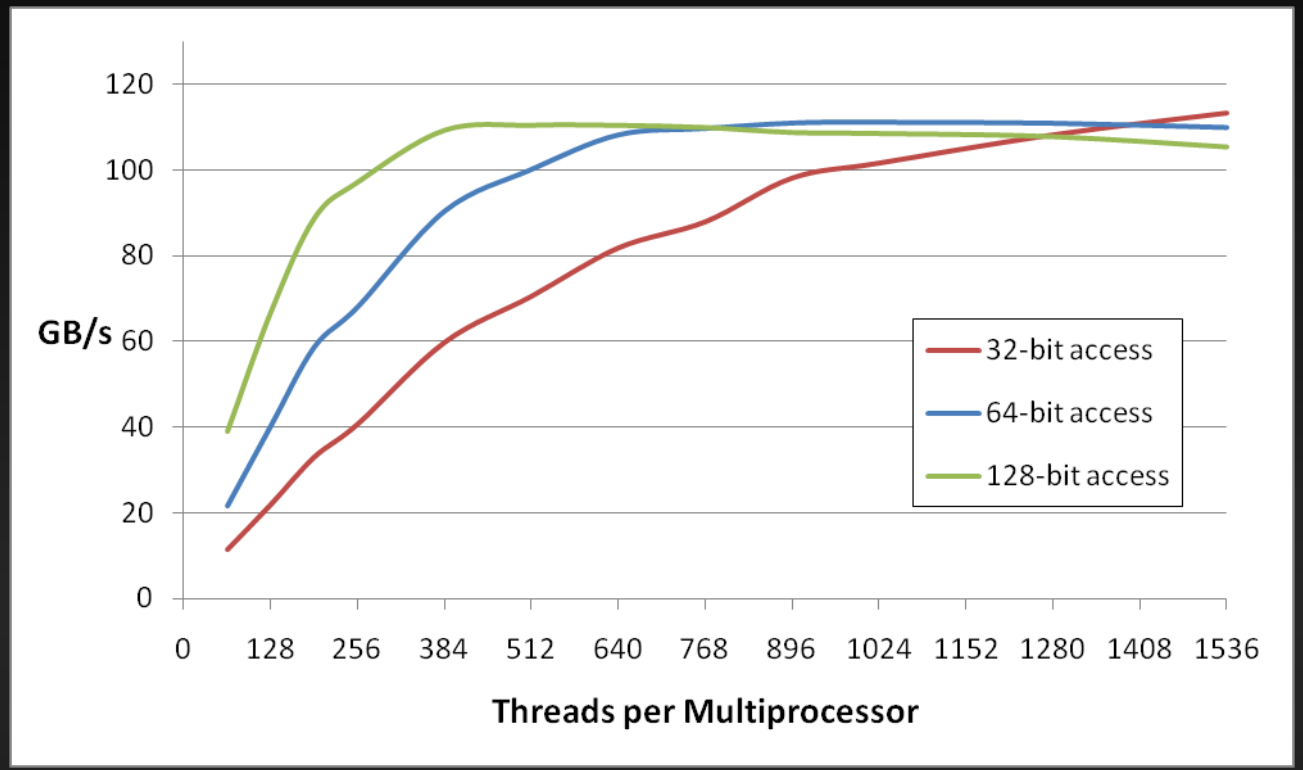
- Need 1 warp on scheduler to hide compute latency
- Expose more parallelism with more independent instructions per thread
- Increase Instruction-Level Parallelism (ILP)

Maximizing Global Memory Throughput

- Depends on the access pattern and word size
- Need enough memory transaction in flight to saturate the bus
  - Independent loads and stores from the same thread (~more work per thread)
  - Load and stores from different threads (~more threads)
  - Larger word sizes can help (e.g., load/store in float2, float4)

Maximizing Memory Throughput

- Increment of an array of 64M elements
  - Two accesses per thread (load then store)
  - The two accesses are dependent, so really 1 access per thread at a time
- Tesla C2050, ECC on, theoretical bandwidth: ~120 GB/s



Several independent smaller accesses have the same effect as one larger one.

For example:

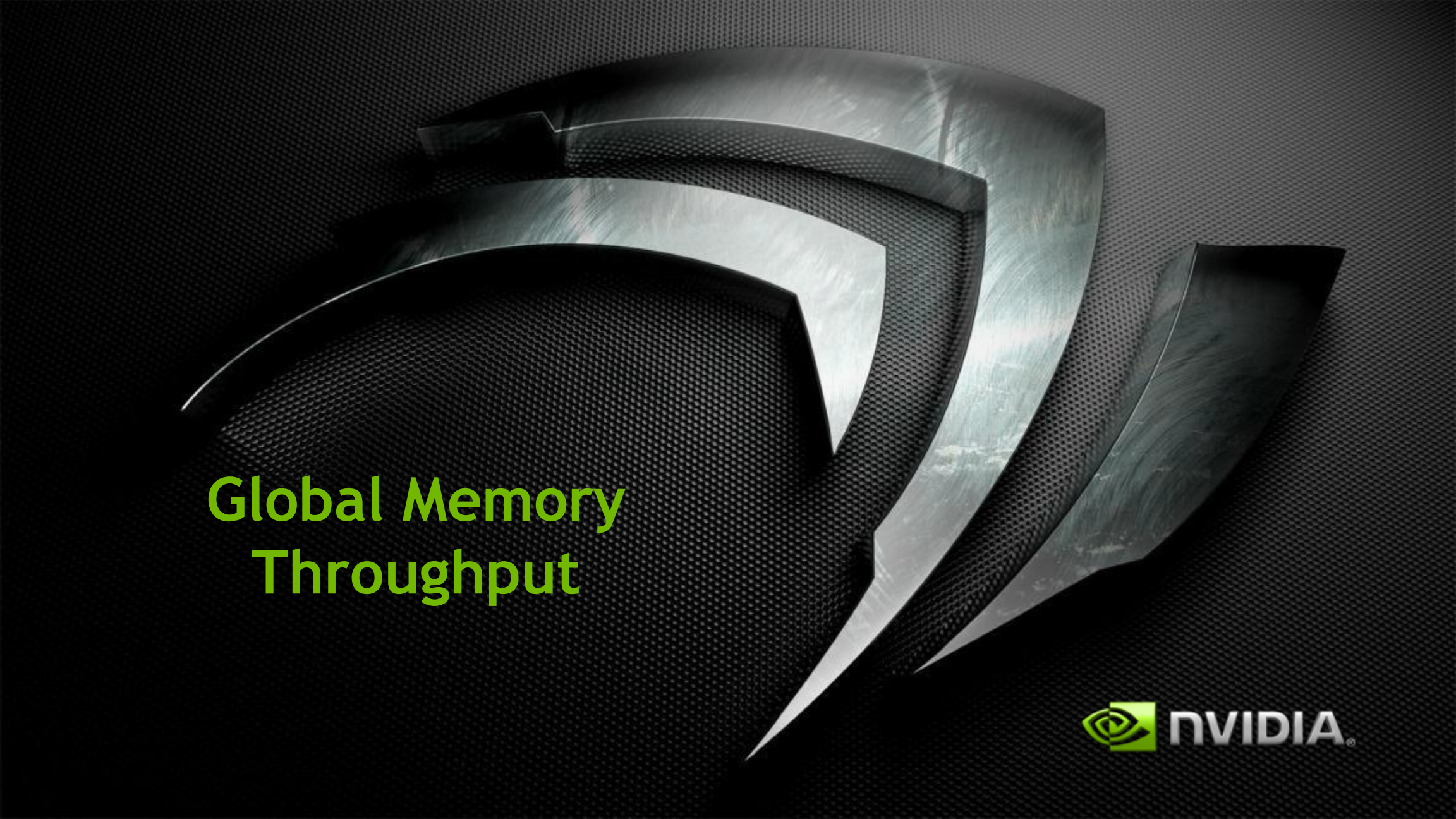
Four 32-bit ~ one 128-bit

Launch Configuration

- Need enough total threads to keep GPU busy
  - Typically, you'd like 256+ threads per SM (aim for 2048 - maximum "occupancy")
    - More if processing one fp32 element per thread

- Threadblock configuration
  - Threads per block should be a multiple of warp size (32)
  - SM can concurrently execute up to 32 thread blocks
    - Really small thread blocks prevent achieving good occupancy
    - Really large thread blocks are less flexible
    - Rule of thumb is ~128-256 threads/block, but use whatever is best for the application

Global Memory Throughput



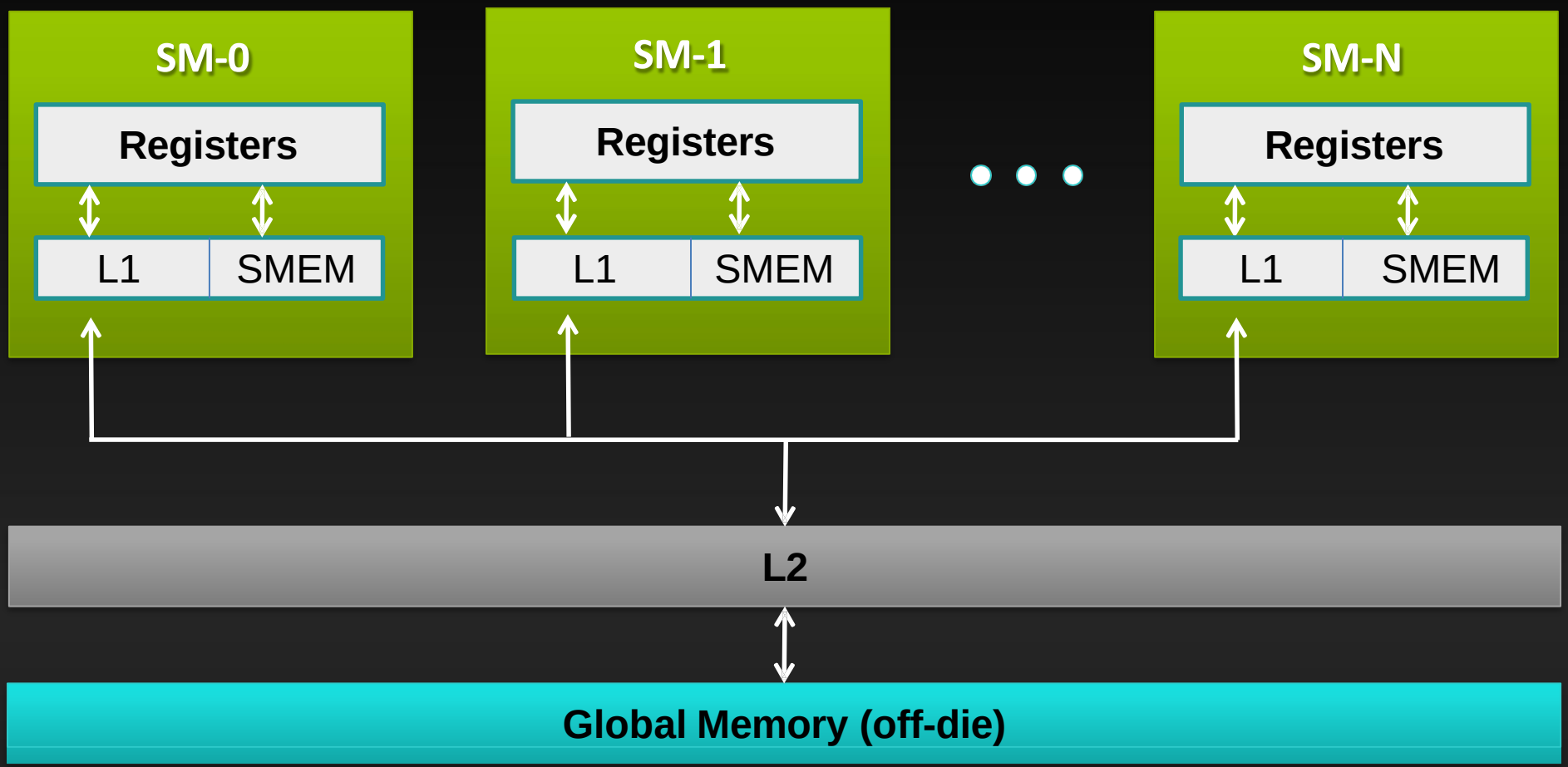
Memory Hierarchy Review

- Local memory
  - Each thread has own local memory (backed by global memory)
  - Mostly registers (managed by the compiler)

- Shared memory / L1
  - Program configurable: 256 kB in total (varies between GPUs)
  - Can allocate up to 227 kB in shared memory (needs to be dynamically allocated for > 48 kB)
  - Shared memory is accessible by the threads in the same threadblock
  - Very low latency
  - Very high throughput
- L2
  - All accesses to global memory go through L2, including copies to/from CPU host

- Global memory (off-die)
  - Accessible by all threads as well - high latency (~1,000 cycles)
  - Throughput: up to ~4.9 TB/s (H200)

Memory Hierarchy Review



Memory transactions

Cache management:

- Cache is managed by chunks of 128 bytes (one cache line)
- Each cache line is composed of 4 sectors of 32 bytes each

Load:

- Attempts to hit in L1, then L2, then GMEM
- GMEM -> L2 load granularity is 64-byte (2 sectors, adjustable)
- L2 -> L1 load granularity is 32-bytes (1 sector)

Stores:

- L1: write-through cache (wait until data reaches L2)
- L2: write-back cache

Load Operation

- Memory operations are issued per warp (32 threads)
  - Just like all other instructions
- Operation:
  - Threads in a warp provide memory addresses
  - Determine which lines/sectors are needed
  - Request the needed sectors

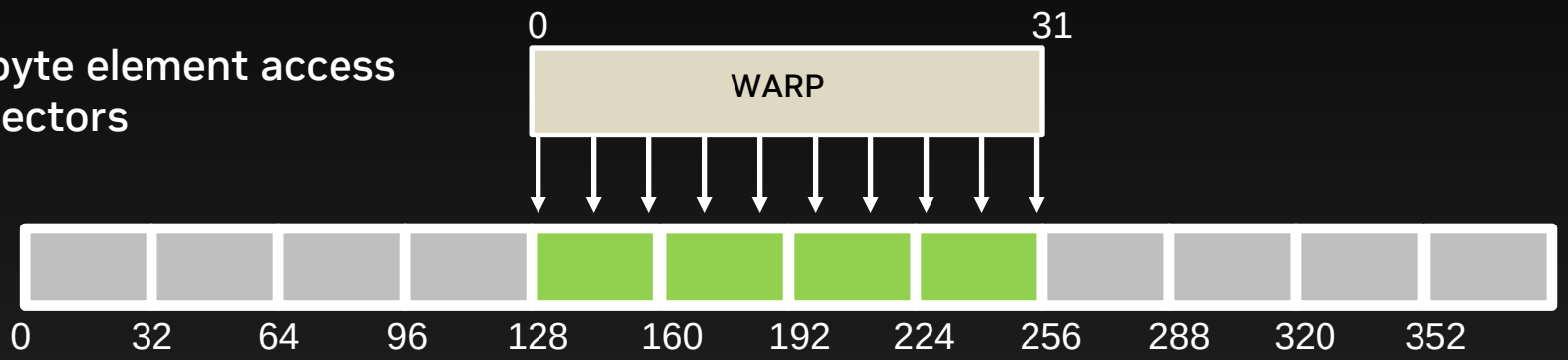
Global Memory Access Patterns

Global Memory Access Patterns

Global Memory Access Patterns

Aligned and sequential

4-byte element access
4 sectors

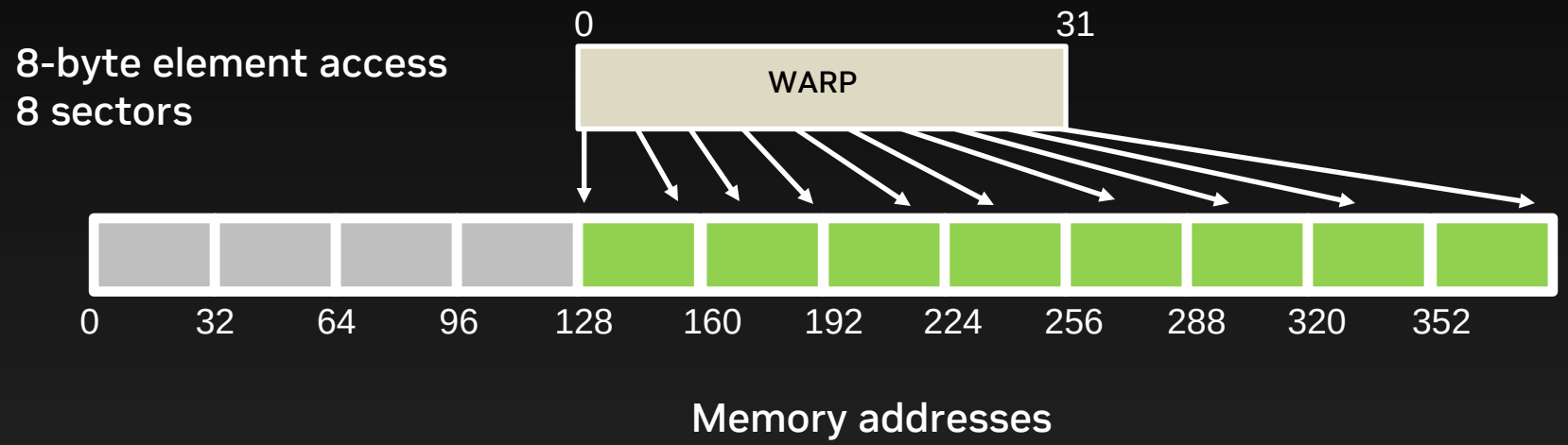


COALESCED ACCESS

The addresses and requests are grouped so that the data transfer is executed with a minimum number of transactions

Global Memory Access Patterns

Aligned and sequential

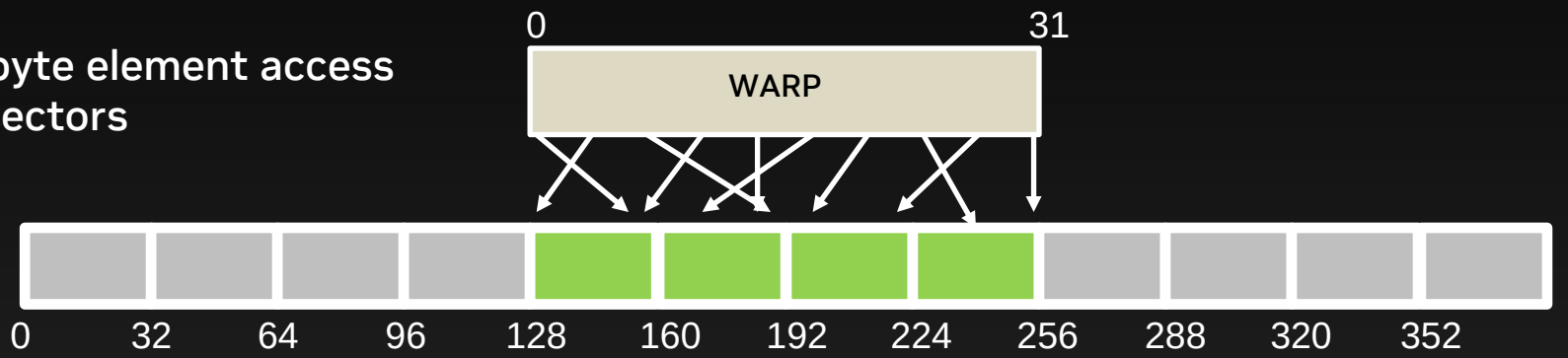


COALESCED ACCESS

Global Memory Access Patterns

Aligned and non-sequential

4-byte element access
4 sectors

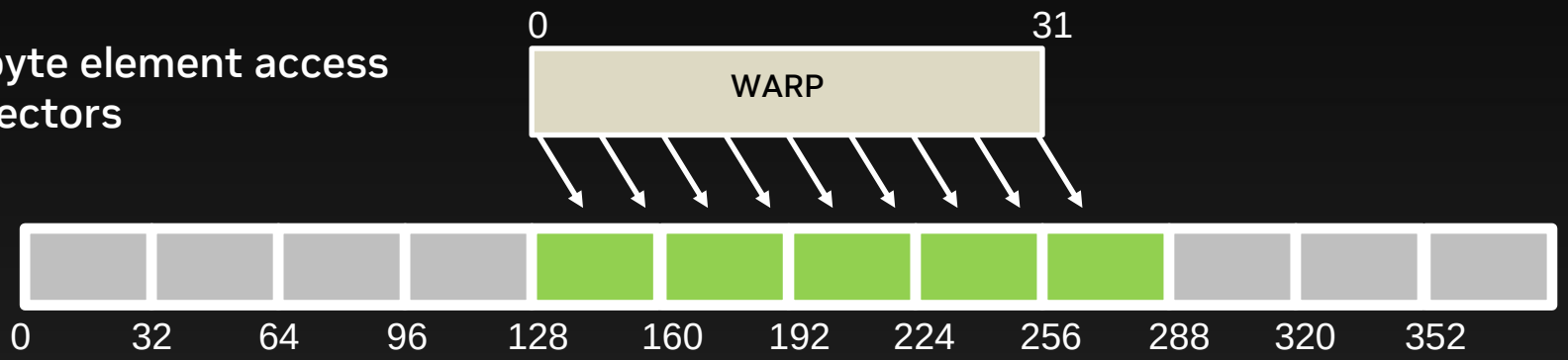


COALESCED ACCESS

Global Memory Access Patterns

Mis-aligned and sequential

4-byte element access
5 sectors

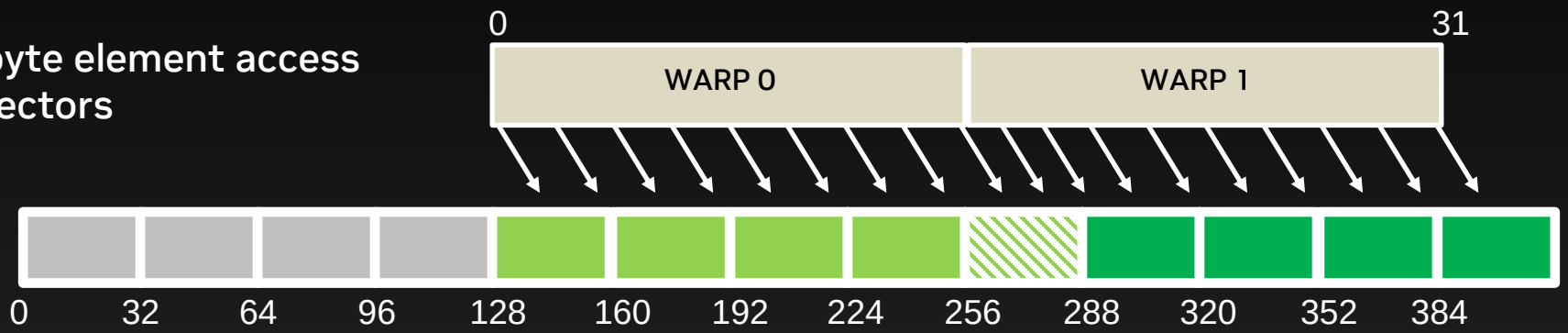


MIS-ALIGNED

Global Memory Access Patterns

Mis-aligned and sequential

4-byte element access
9 sectors

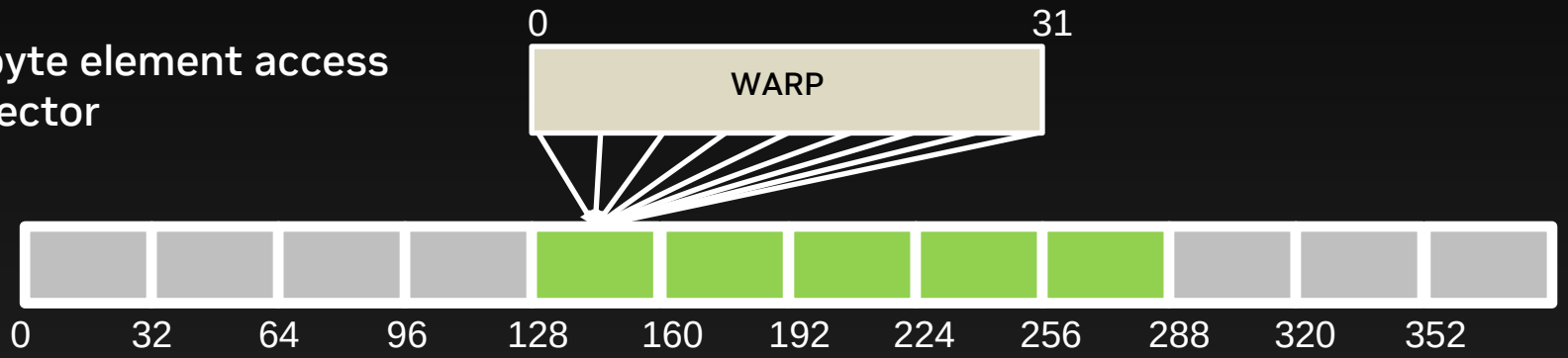


MIS-ALIGNED

Global Memory Access Patterns

Same address

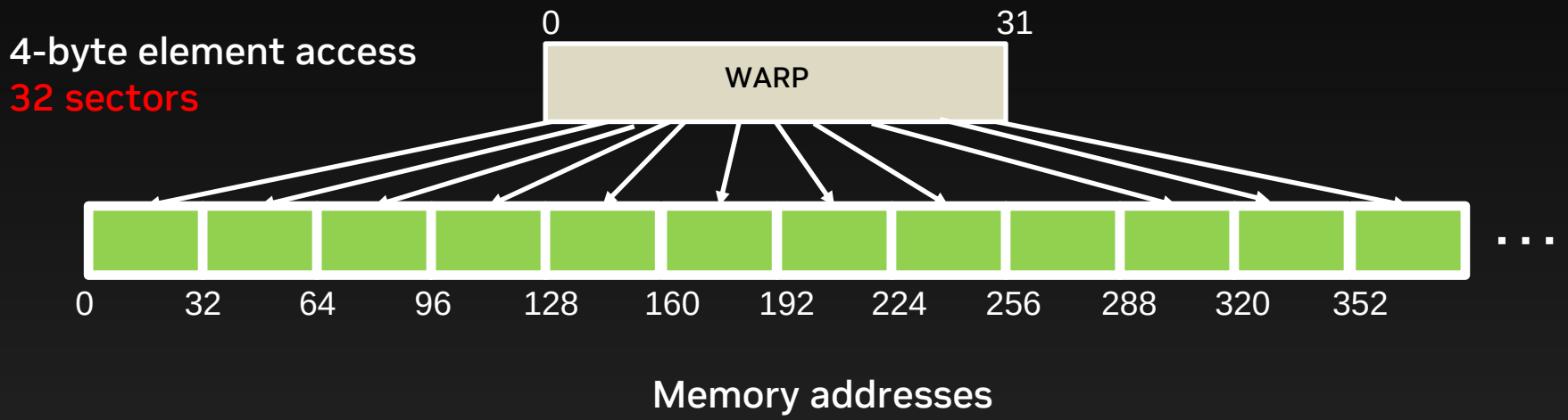
4-byte element access
1 sector



BROADCAST

Global Memory Access Patterns

Aligned and strided



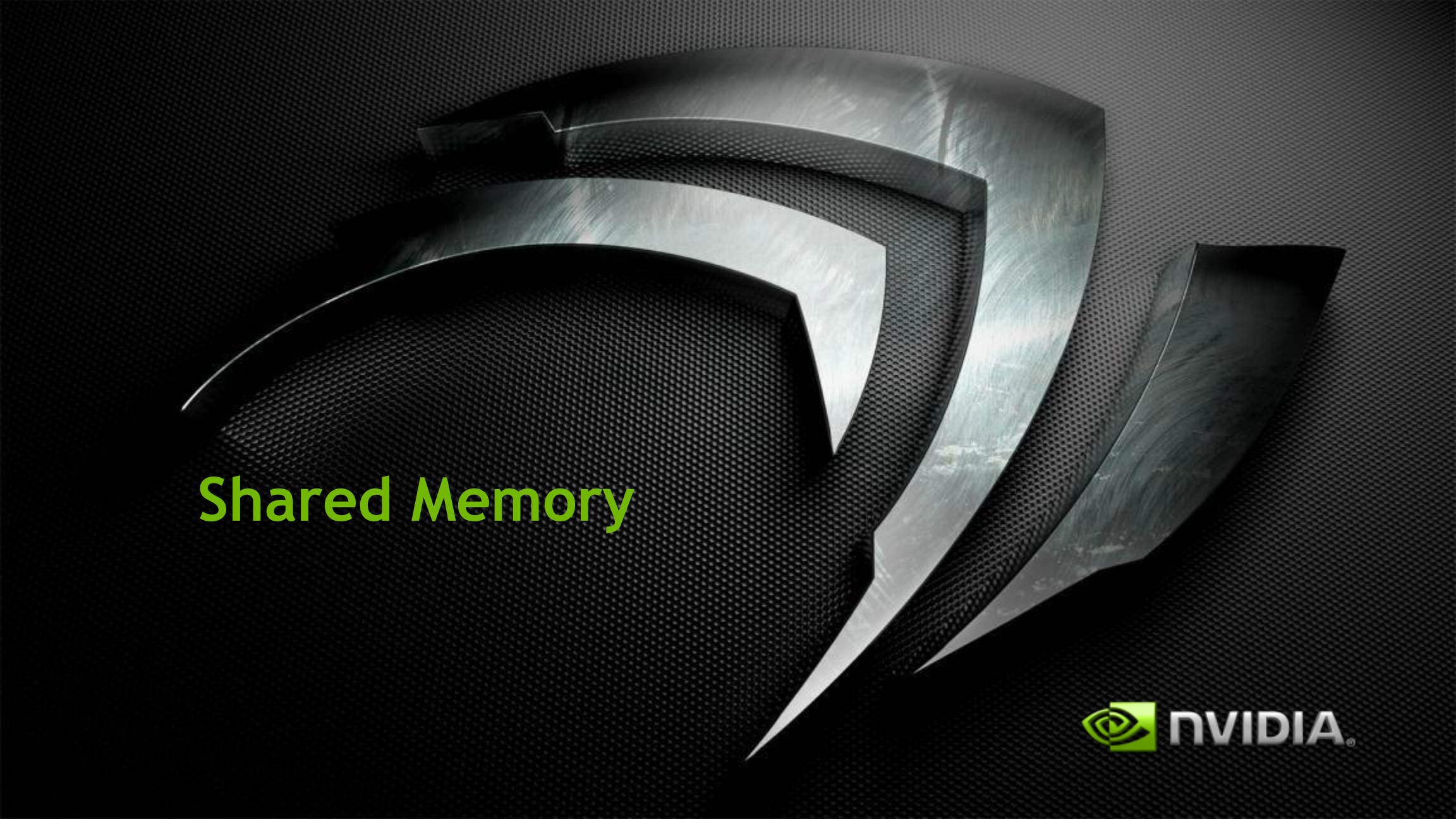
STRIDED – One sector per thread

GMEM Optimization Guidelines

- Strive for perfect coalescing
  - Align starting address (may require padding)
  - A warp should access within a contiguous region

- Have enough concurrent accesses to saturate the bus
  - Process several elements per thread
    - Multiple loads get pipelined
  - Launch enough threads to maximize throughput
    - Latency is hidden by switching threads (warps)

Shared Memory

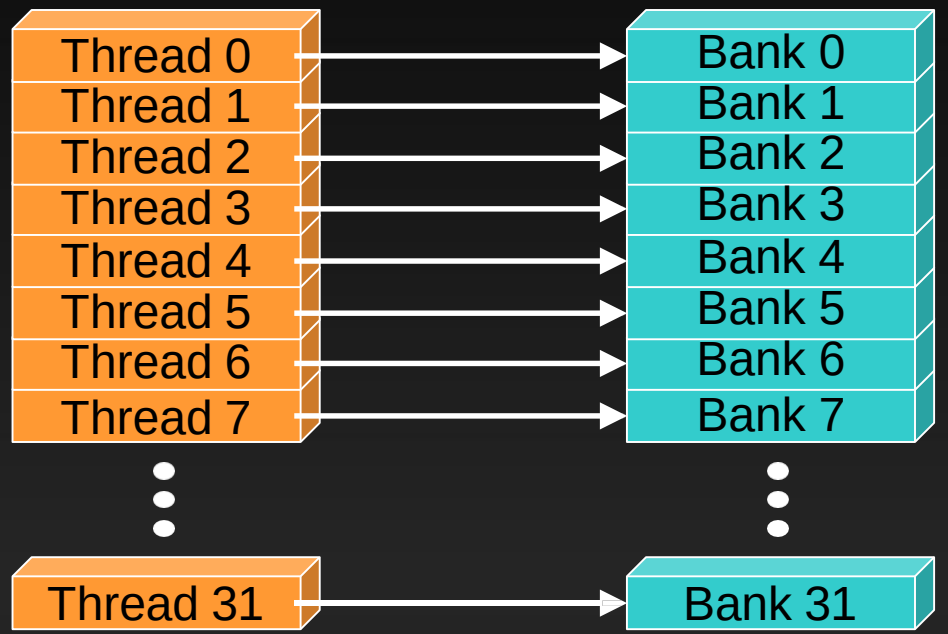


Shared Memory

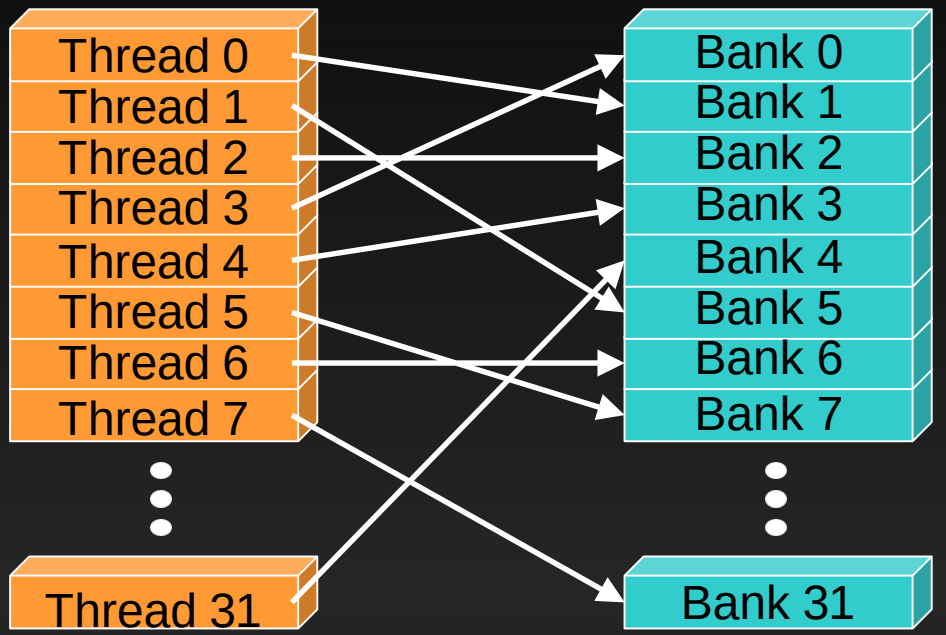
- Uses:
  - Inter-thread communication within a block
  - Cache data to reduce redundant global memory accesses
- Organization:
  - 32 banks, 4-byte wide banks
  - Successive 4-byte words belong to different banks
- Performance:
  - 4 bytes per bank per 2 clocks per multiprocessor
  - Accesses are issued per 32 threads (warp)
  - Serialization: if *N* threads of 32 access different 4-byte words in the same bank, *N* accesses are executed serially ("bank conflict")
  - Multicast: *N* threads access the same word in one fetch
    - Could be different bytes within the same word

Bank Addressing Examples

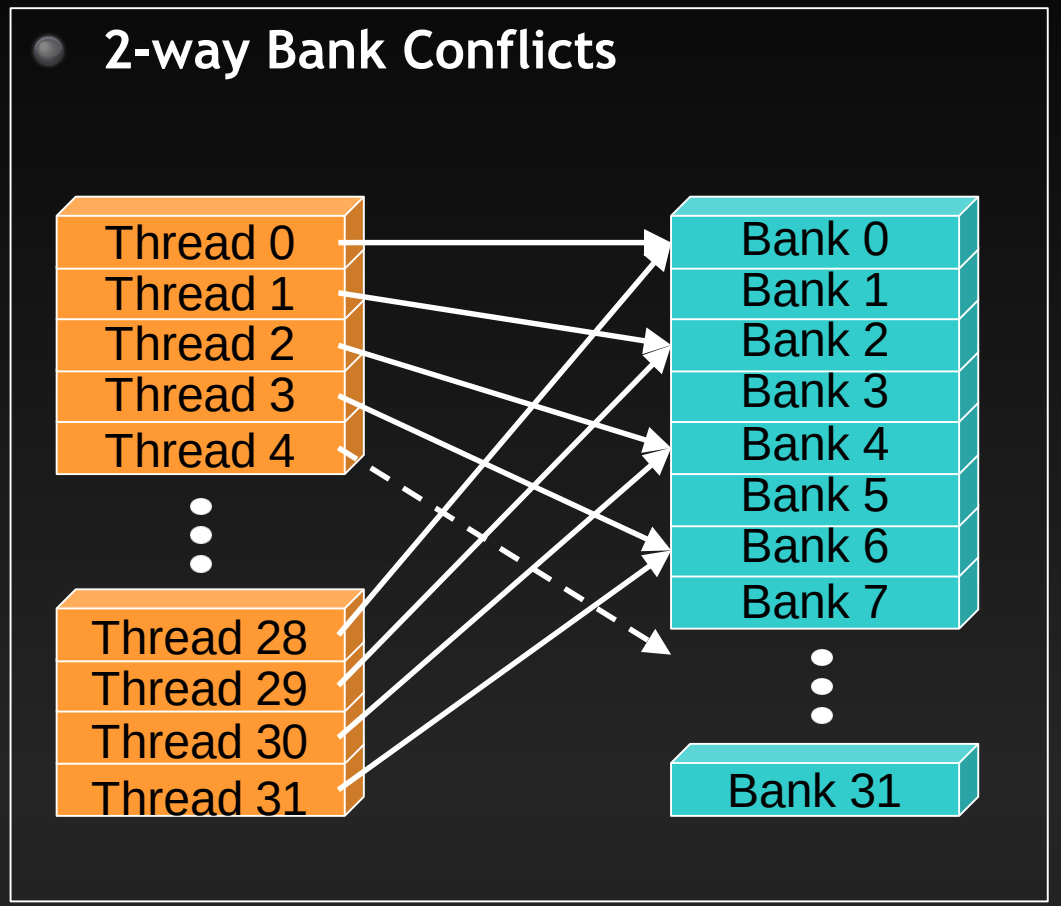
- No Bank Conflicts

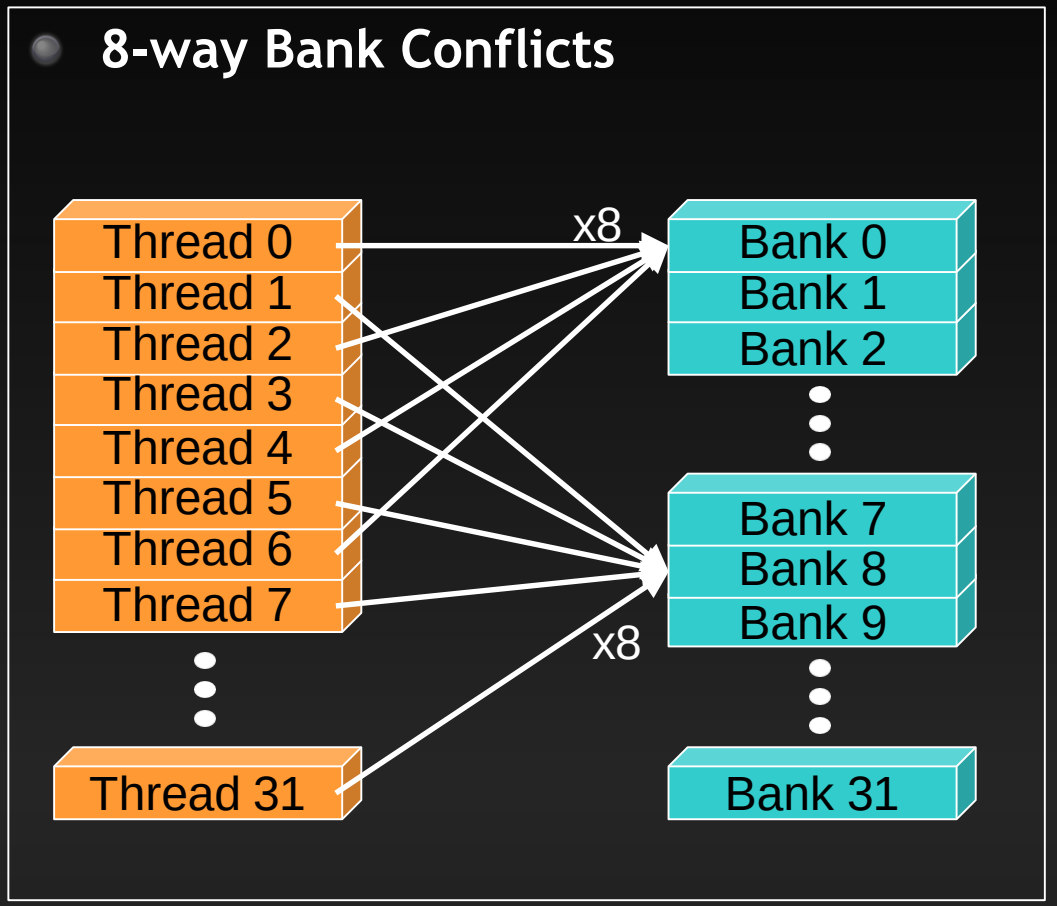


- No Bank Conflicts



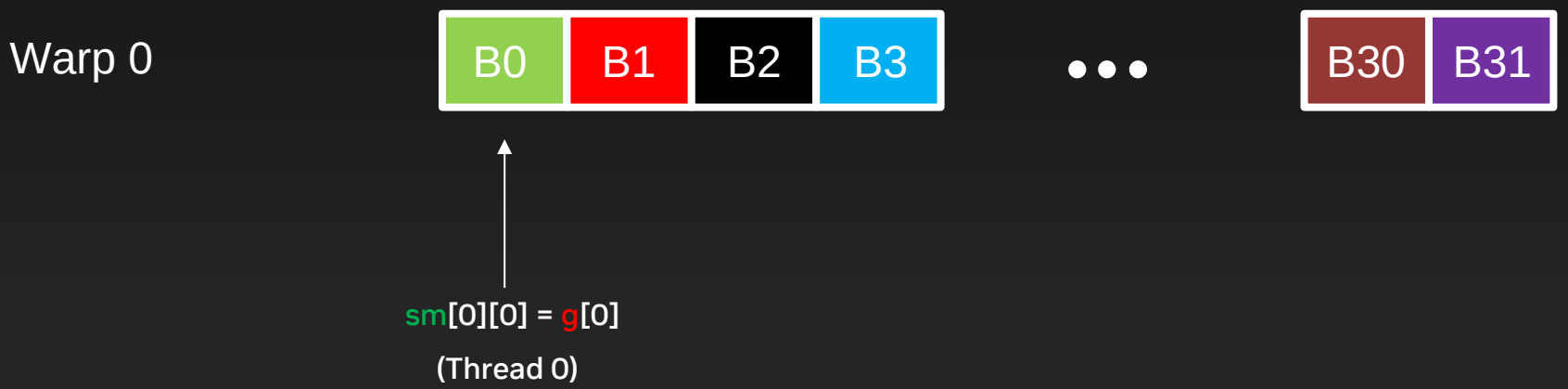
Bank Addressing Examples





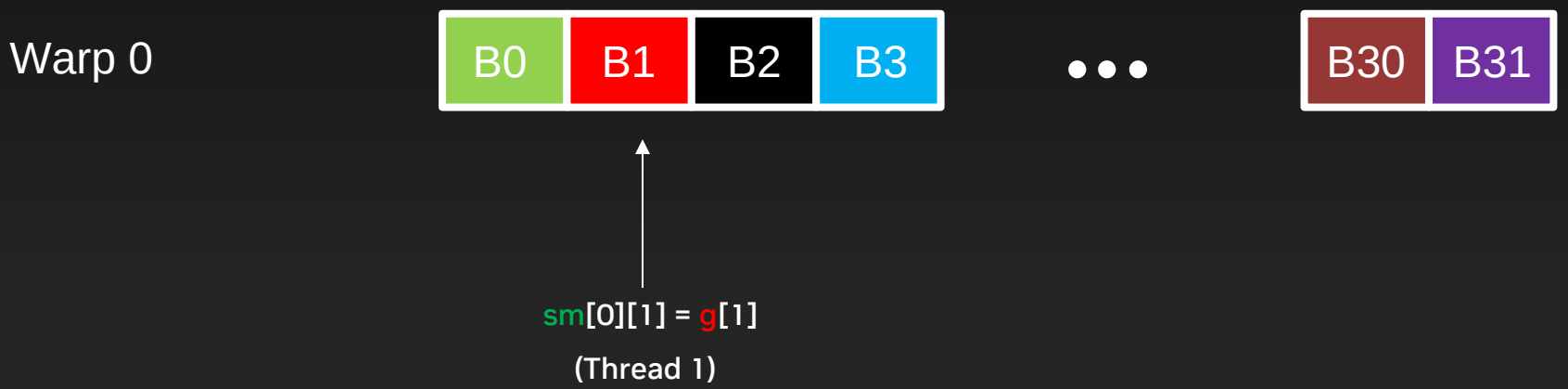
Shared Memory - Example

- Array g allocated on global memory
- Allocate sm[32][32] shared memory
- Load from global memory into shared memory
- Each warp loads 32 contiguous elements (float)



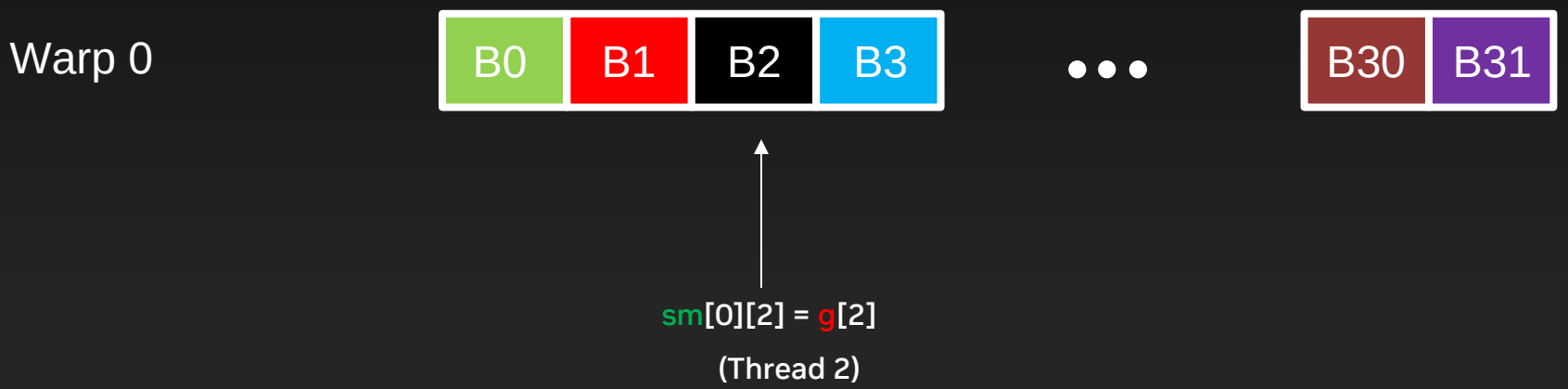
Shared Memory - Example

- Array g allocated on global memory
- Allocate sm[32][32] shared memory
- Load from global memory into shared memory
- Each warp loads 32 contiguous elements (float)



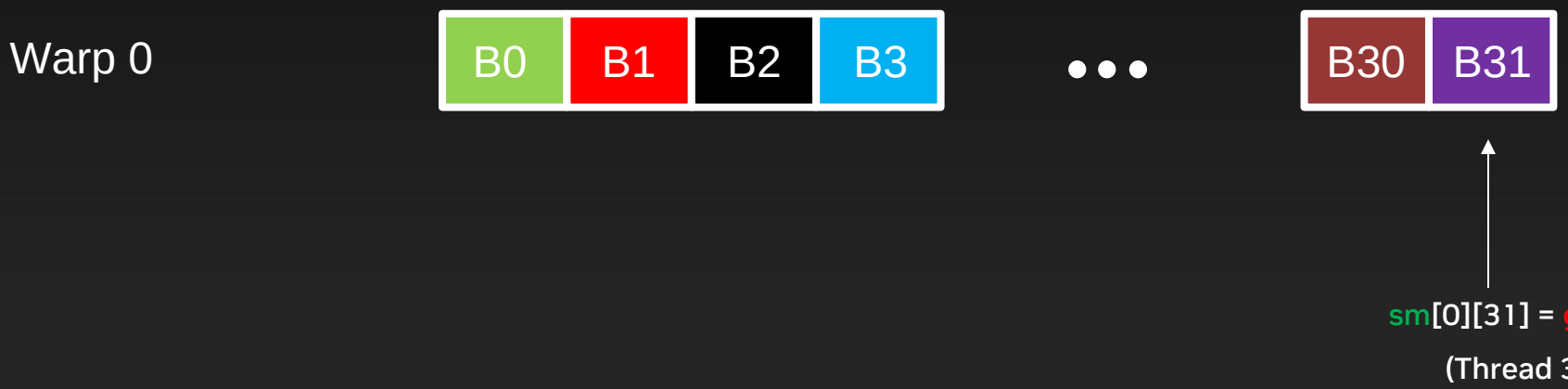
Shared Memory - Example

- Array g allocated on global memory
- Allocate sm[32][32] shared memory
- Load from global memory into shared memory
- Each warp loads 32 contiguous elements (float)



Shared Memory - Example

- Array g allocated on global memory
- Allocate sm[32][32] shared memory
- Load from global memory into shared memory
- Each warp loads 32 contiguous elements (float)

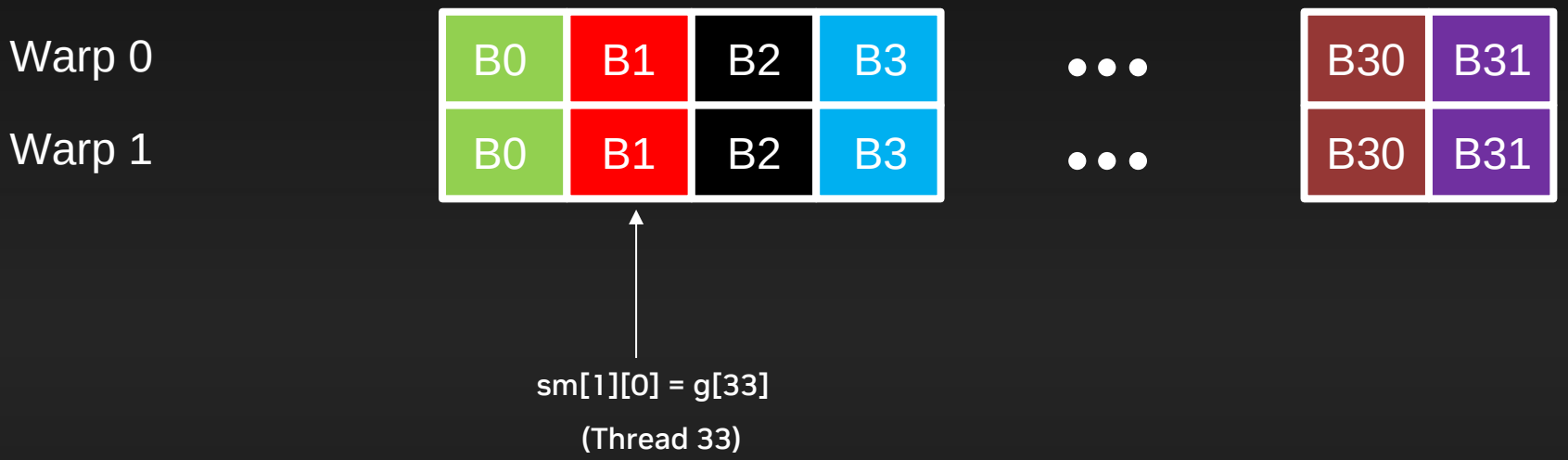


Shared Memory - Example

- Array g allocated on global memory
- Allocate sm[32][32] shared memory
- Load from global memory into shared memory
- Each warp loads 32 contiguous elements (float)

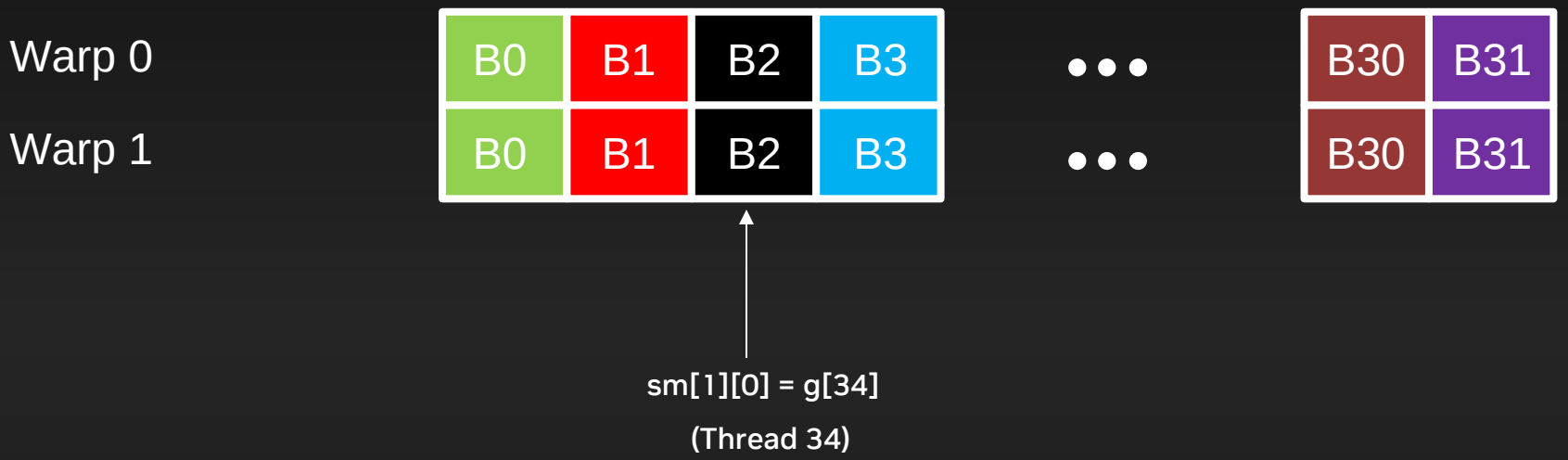
Shared Memory - Example

- Array g allocated on global memory
- Allocate sm[32][32] shared memory
- Load from global memory into shared memory
- Each warp loads 32 contiguous elements (float)



Shared Memory - Example

- Array g allocated on global memory
- Allocate sm[32][32] shared memory
- Load from global memory into shared memory
- Each warp loads 32 contiguous elements (float)



Shared Memory - Example

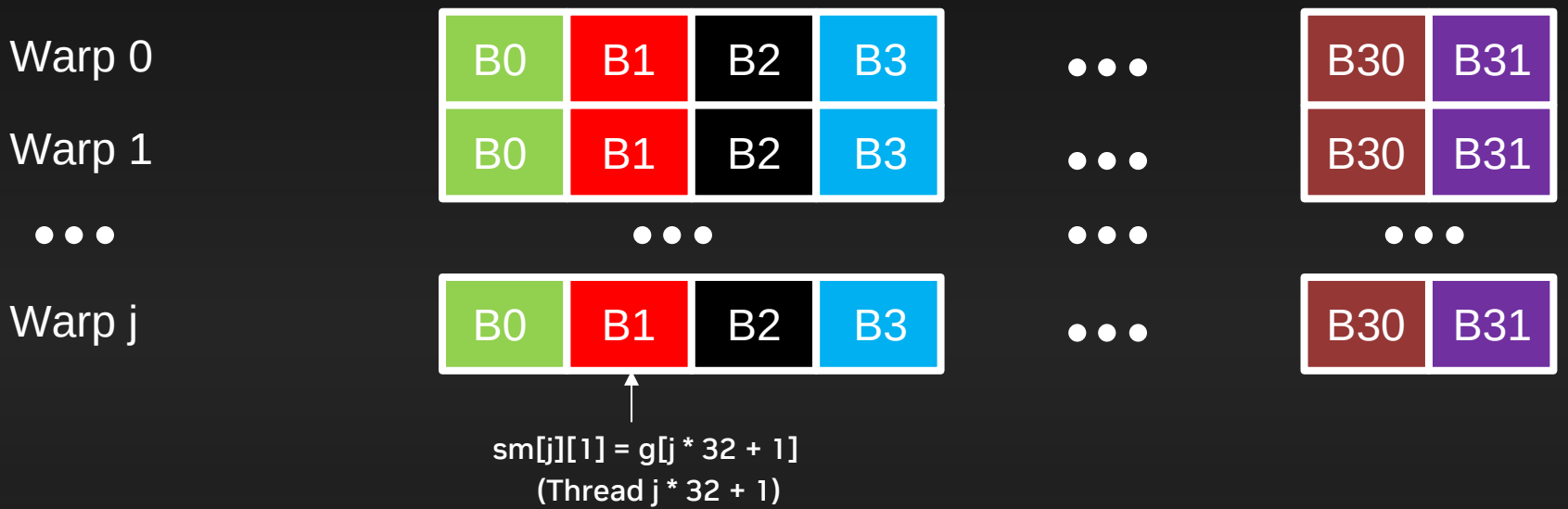
- Array g allocated on global memory
- Allocate sm[32][32] shared memory
- Load from global memory into shared memory
- Each warp loads 32 contiguous elements (float)

Shared Memory - Example

- Array g allocated on global memory
- Allocate sm[32][32] shared memory
- Load from global memory into shared memory
- Each warp loads 32 contiguous elements (float)

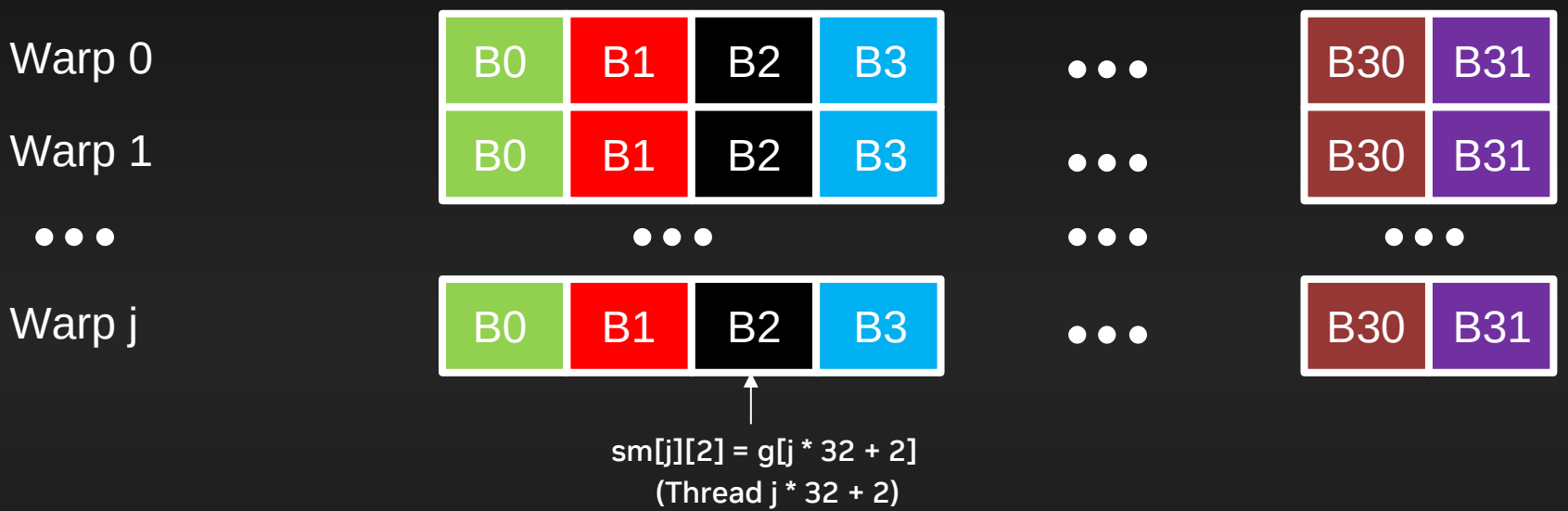
Shared Memory - Example

- Array g allocated on global memory
- Allocate sm[32][32] shared memory
- Load from global memory into shared memory
- Each warp loads 32 contiguous elements (float)



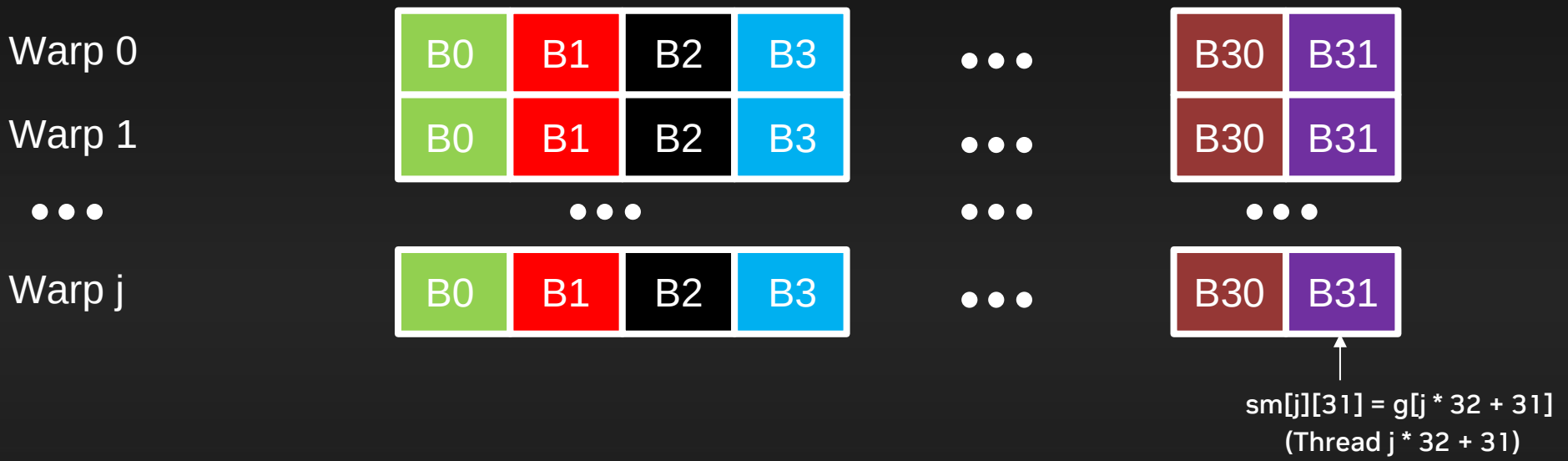
Shared Memory - Example

- Array g allocated on global memory
- Allocate sm[32][32] shared memory
- Load from global memory into shared memory
- Each warp loads 32 contiguous elements (float)



Shared Memory - Example

- Array g allocated on global memory
- Allocate sm[32][32] shared memory
- Load from global memory into shared memory
- Each warp loads 32 contiguous elements (float)

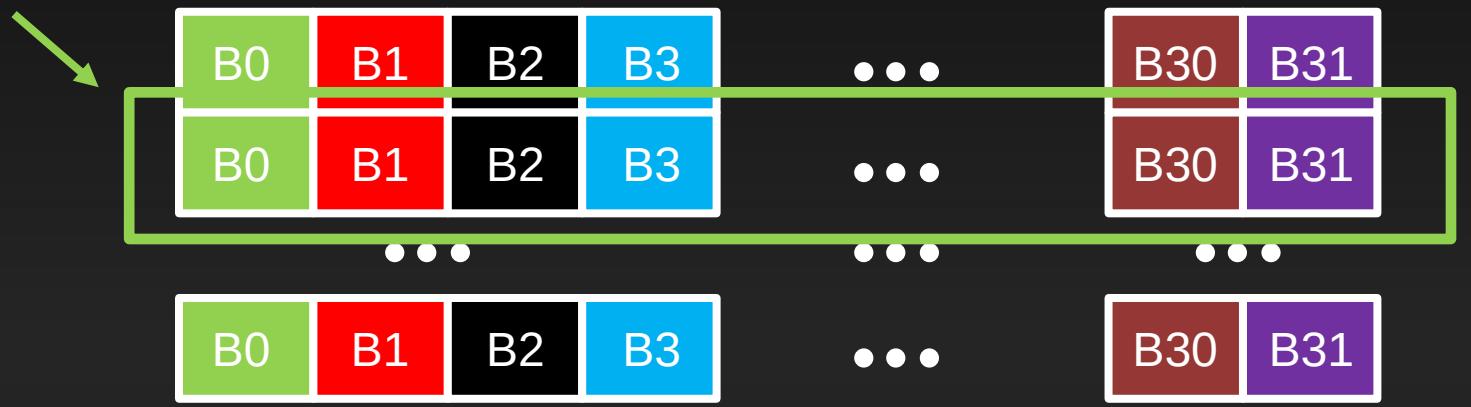


No Bank Conflict

- What if each thread in warp 1 wants to access elements in the second row?

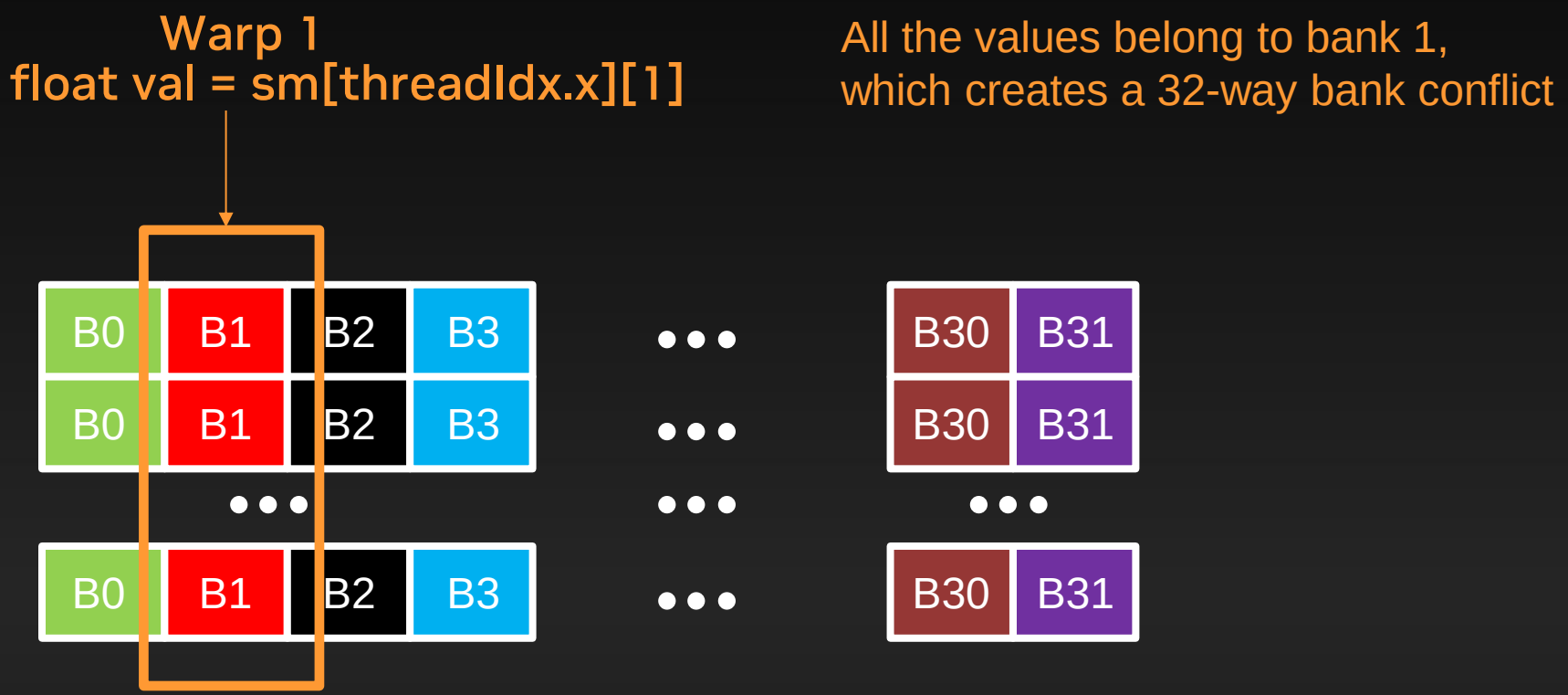
All the values belong to different banks, no bank conflict

Warp 1
float val = sm[1][threadIdx.x]



Bank Conflict

- What if each thread in warp 1 wants to access elements in the second column?

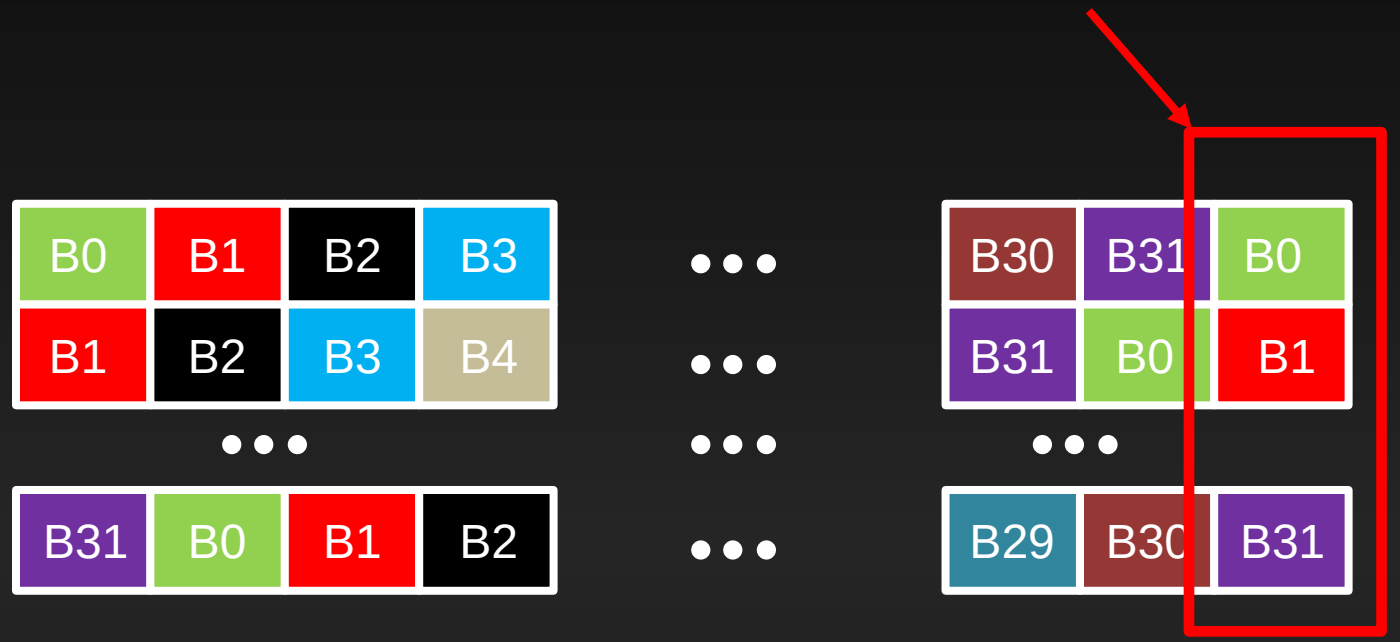


tid = threadIdx.x

Avoiding Bank Conflicts - Padding

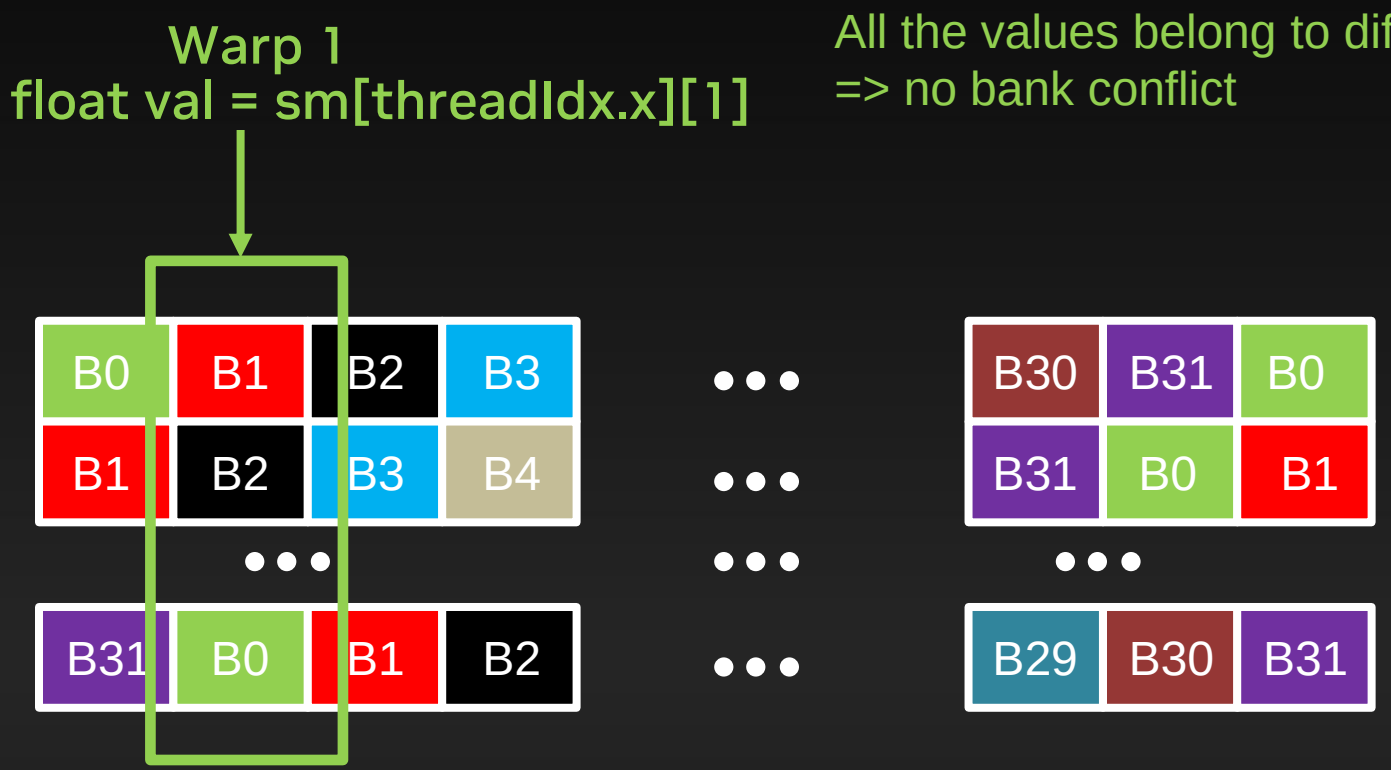
- What if each thread in warp 1 wants to access elements in the second column?

Pad with one element in leading dimension
Allocate sm[32][33]



Avoiding Bank Conflicts - Padding

- What if each thread in warp 1 wants to access elements in the second column?



Avoiding Bank Conflicts - Swizzling

- Swizzling: another method to avoid bank conflicts
  - Modifying (by a permutation) the way we store/access data in shared memory
  - We apply the following re-mapping

g[j * 32 + i] ⟹ sm[j][i]

Avoiding Bank Conflicts - Swizzling

- Swizzling: another method to avoid bank conflicts
  - Modifying (by a permutation) the way we store/access data in shared memory
  - We apply the following re-mapping

g[j * 32 + i]

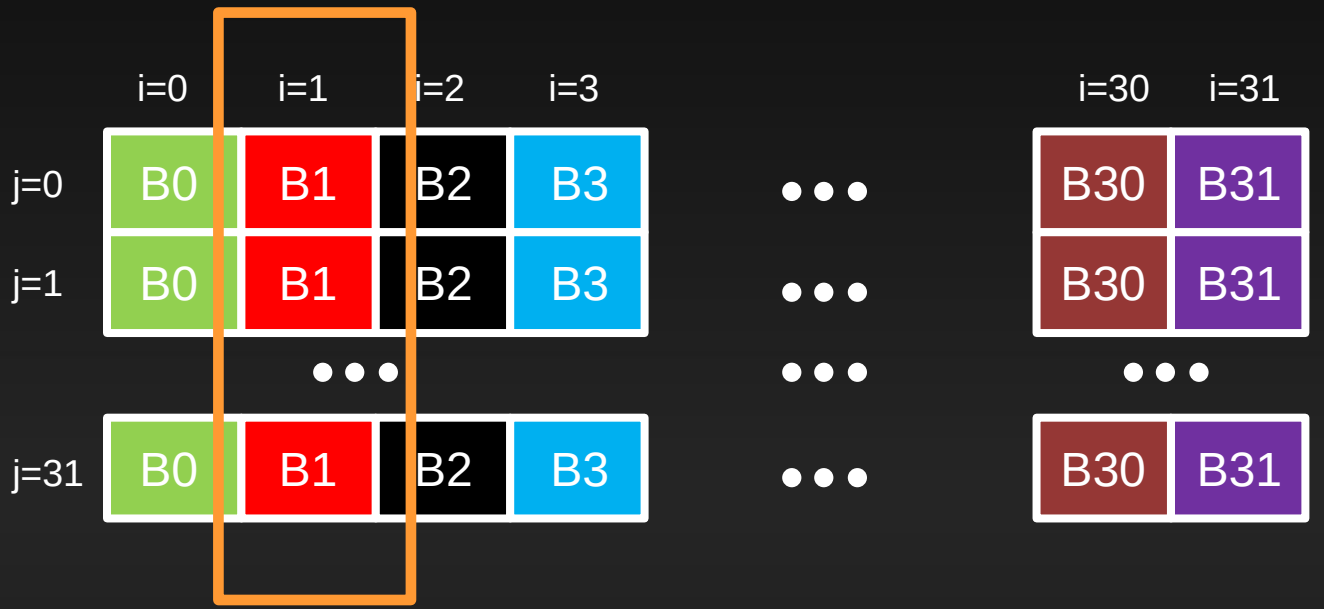
sm[j][i^j]

^ means “xor”

Avoiding Bank Conflicts - Swizzling

- Example: warp 1 wants to store and access g[j * 32 + 1] for j = 0,…,31
- Each threads accesses a different element from column 1:

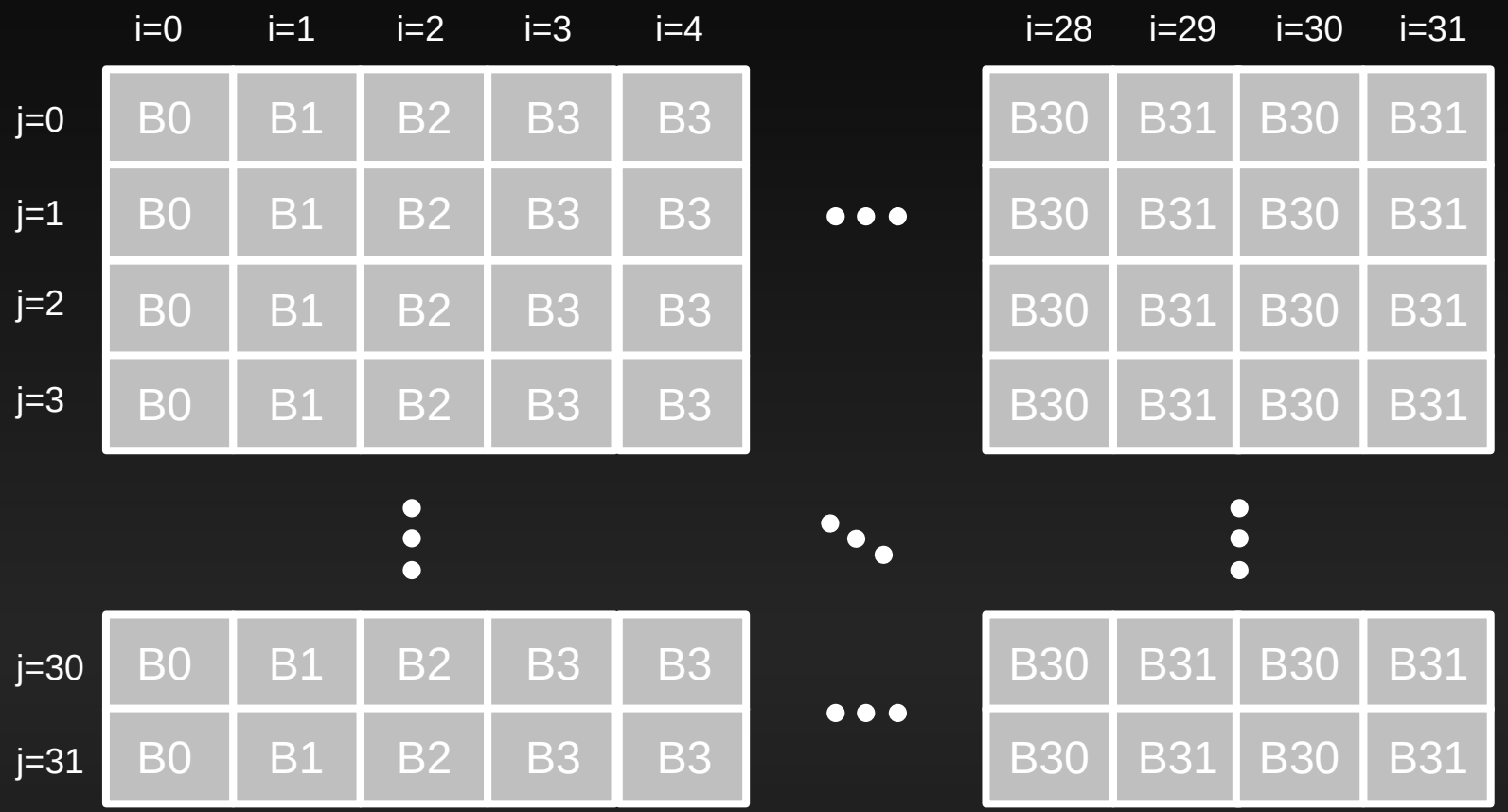
  sm[0][1], sm[1][1], …, sm[31][1]
- 32-way bank conflict



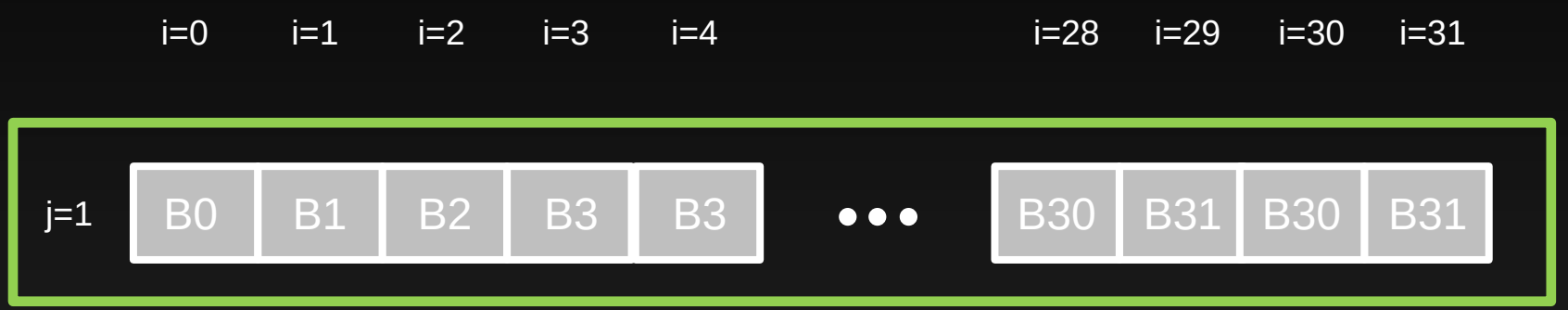
Avoiding Bank Conflicts - Swizzling

- With the new mapping, where would the elements g[j * 32 + 1] be?

Avoiding Bank Conflicts - Swizzling



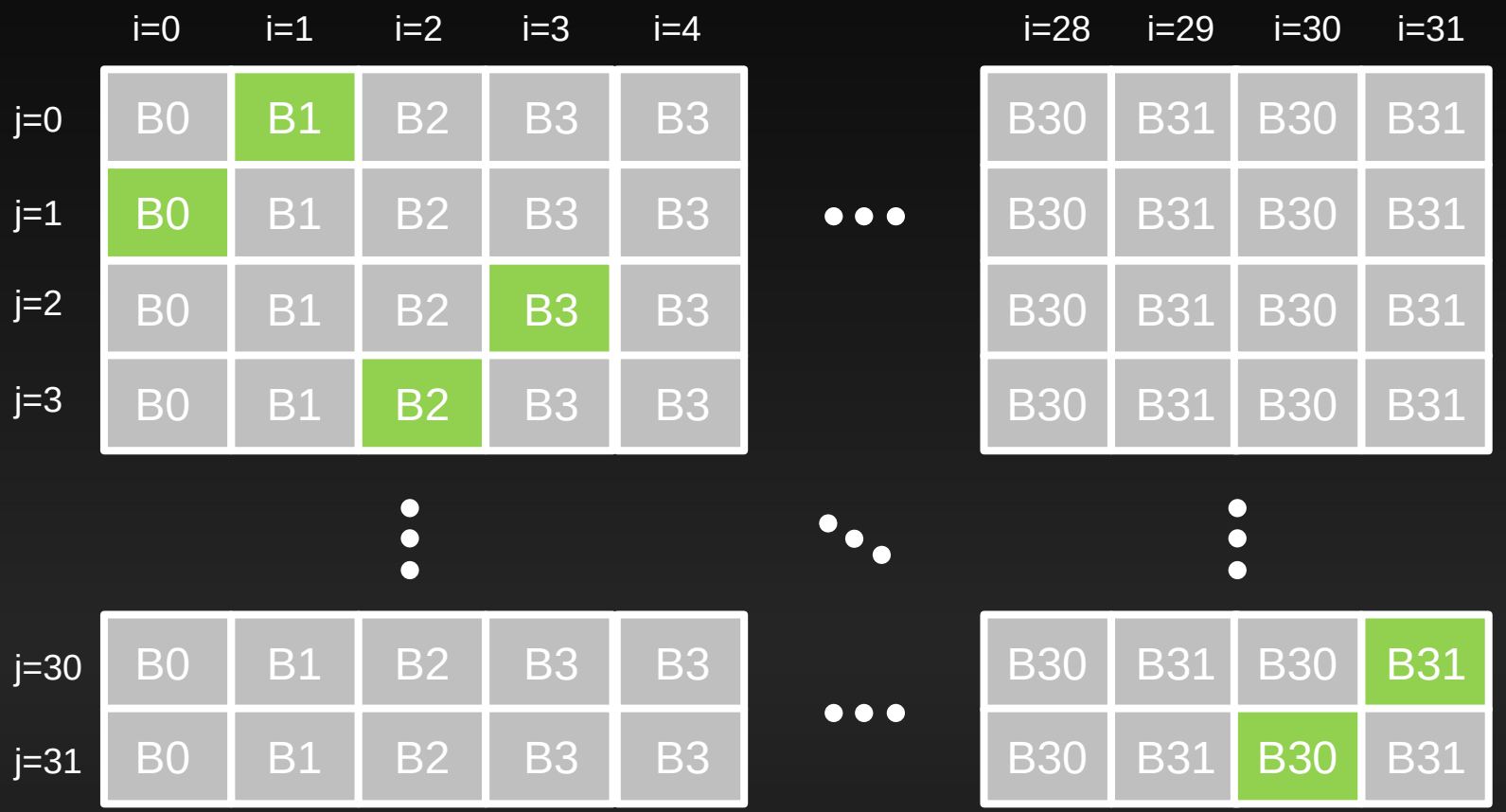
Avoiding Bank Conflicts - Swizzling



Accessing g[1 * 32 + i] for i = 0,…,31 No bank conflict

Avoiding Bank Conflicts - Swizzling

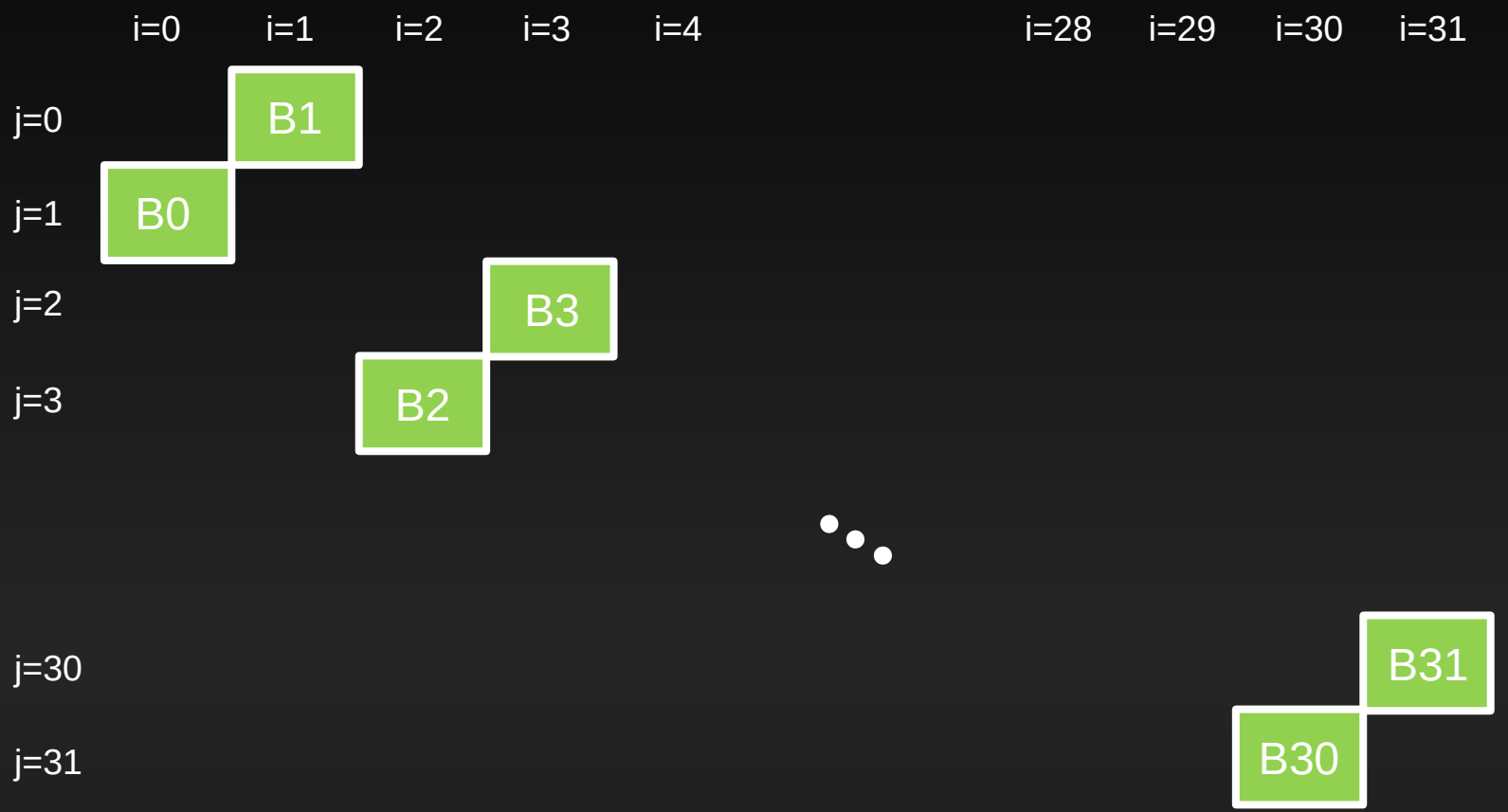
- With the new mapping, where would the elements g[j * 32 + 1] be?



Avoiding Bank Conflicts - Swizzling

Accessing g[j * 32 + 1] for j = 0,…,31

No bank conflict



Avoiding Bank Conflicts - Swizzling

- No need to pad (less memory)
- Need to perform addition “xor” operation to compute index

Summary

- Kernel Launch Configuration
  - Launch enough threads per SM to hide latency
  - Launch enough threadblocks to load the GPU
- Global memory
  - Maximize throughput (GPU has lots of bandwidth, use it effectively)
- Use shared memory when applicable
- Use analysis/profiling when optimizing:
  - “Analysis-driven Optimization” (future session)

Future sessions

- Atomics, Reductions, Warp Shuffle
- Using Managed Memory
- Concurrency (streams, copy/compute overlap, multi-GPU)

Homework

- Get HW2 tarball
- Follow the instructions in the readme.txt file
- Assumptions: basic linux skills, (ls, cd, etc), text editor like vi, plus completion of HW1


Questions?
NVIDIA