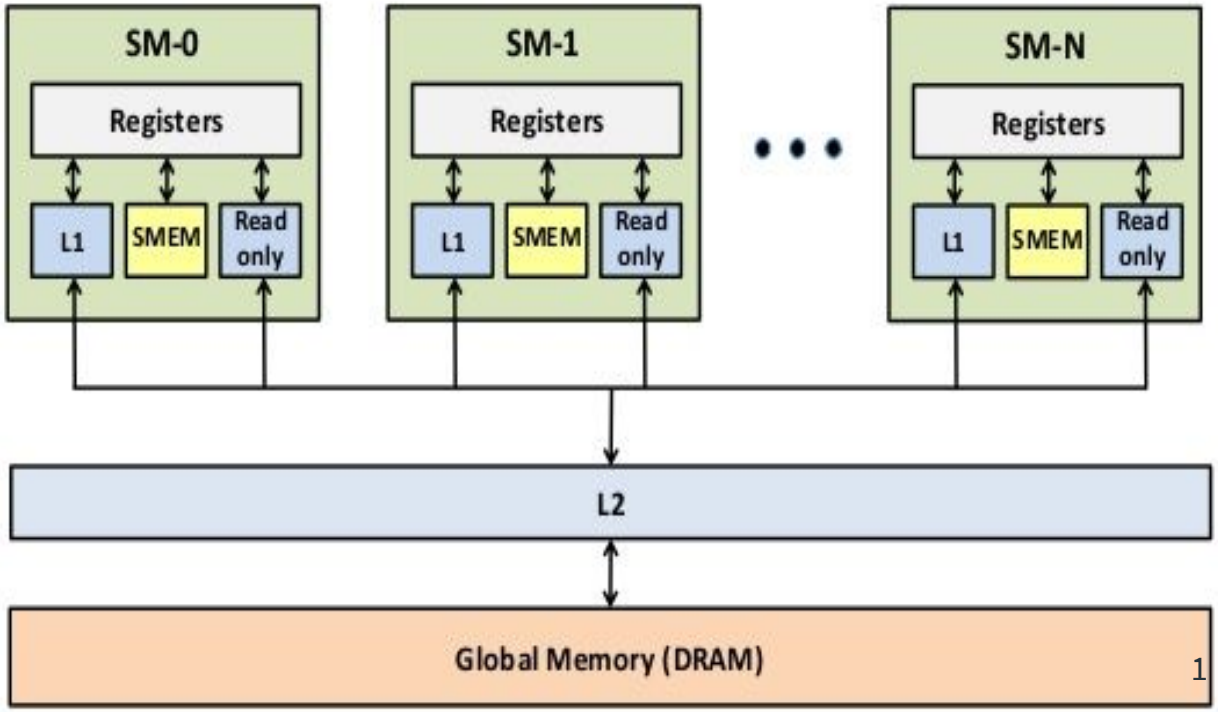
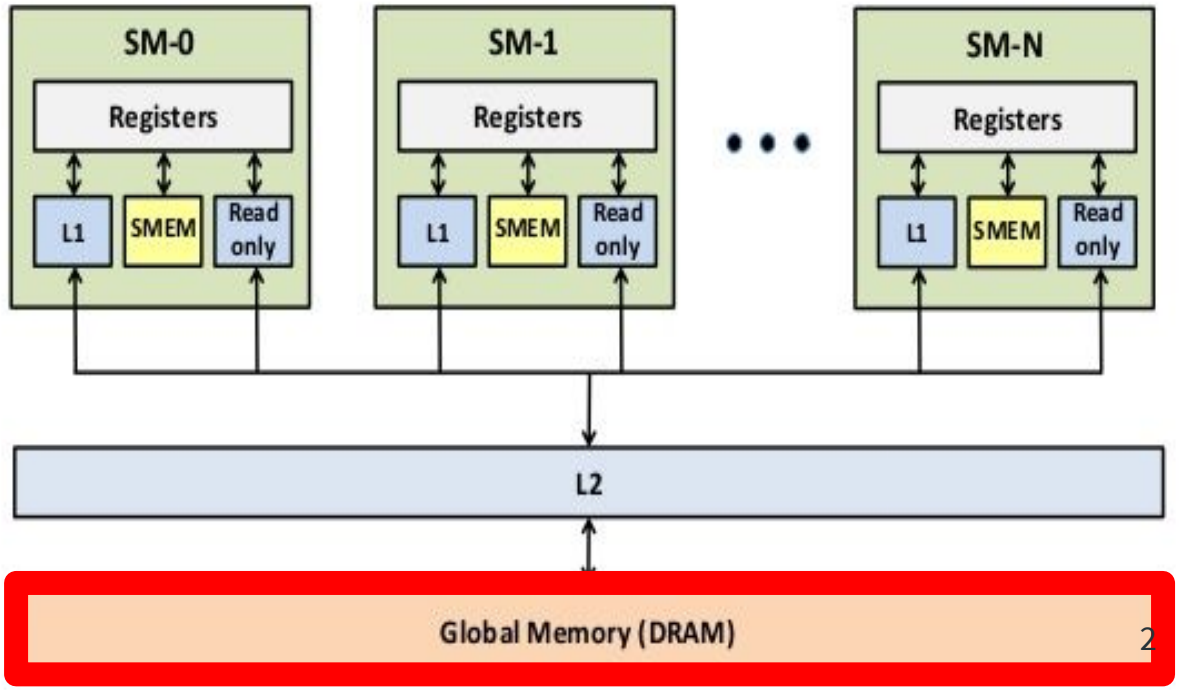
GPU Memory Systems

- GPUs have several types of memory:
  1. Global
  2. Shared
  3. Constant
  4. Register



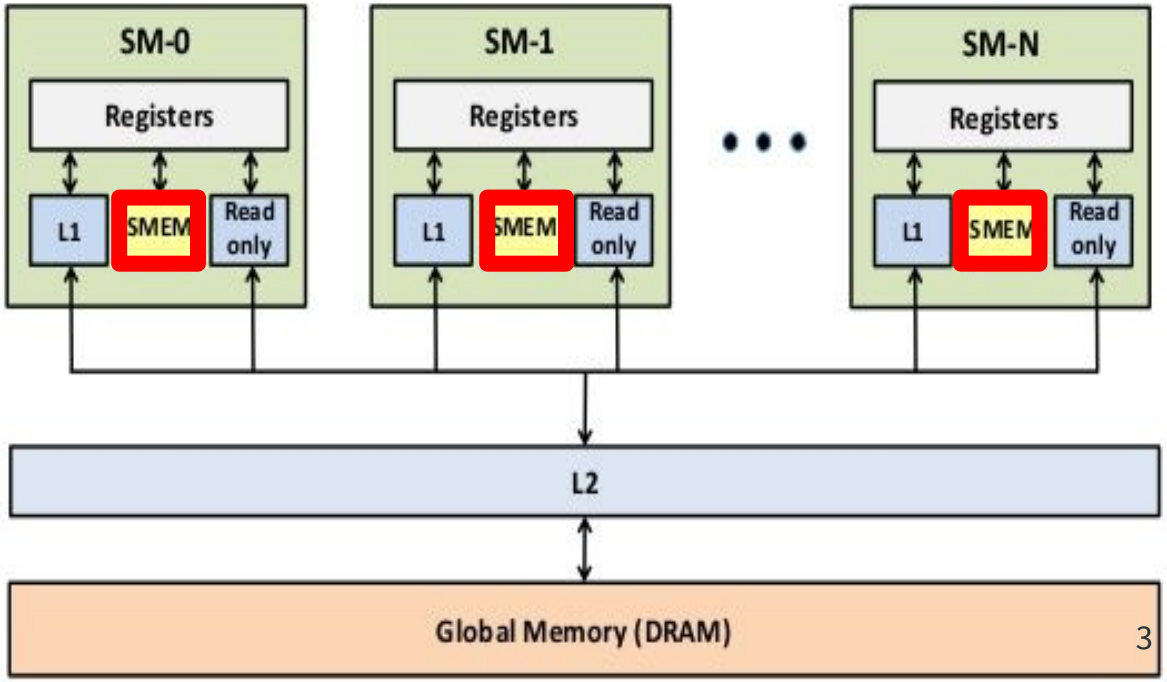
1. Global Memory

Capacity:	8 GB
Cache	L1, L2
Access:	GPU-wide
Latency:	200-400 cycles • Most instructions take ~20-30



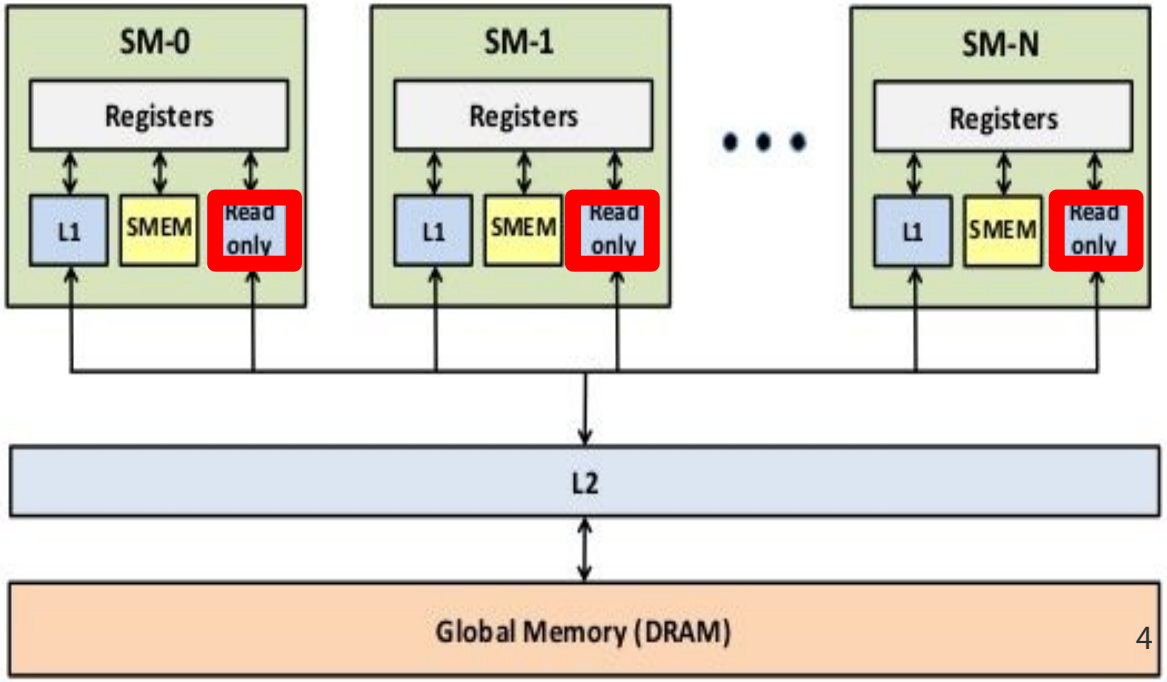
2. Shared Memory

Capacity:	48 KB / SM
Cache	None
Access:	SM-wide
Latency:	1 cycle (!)



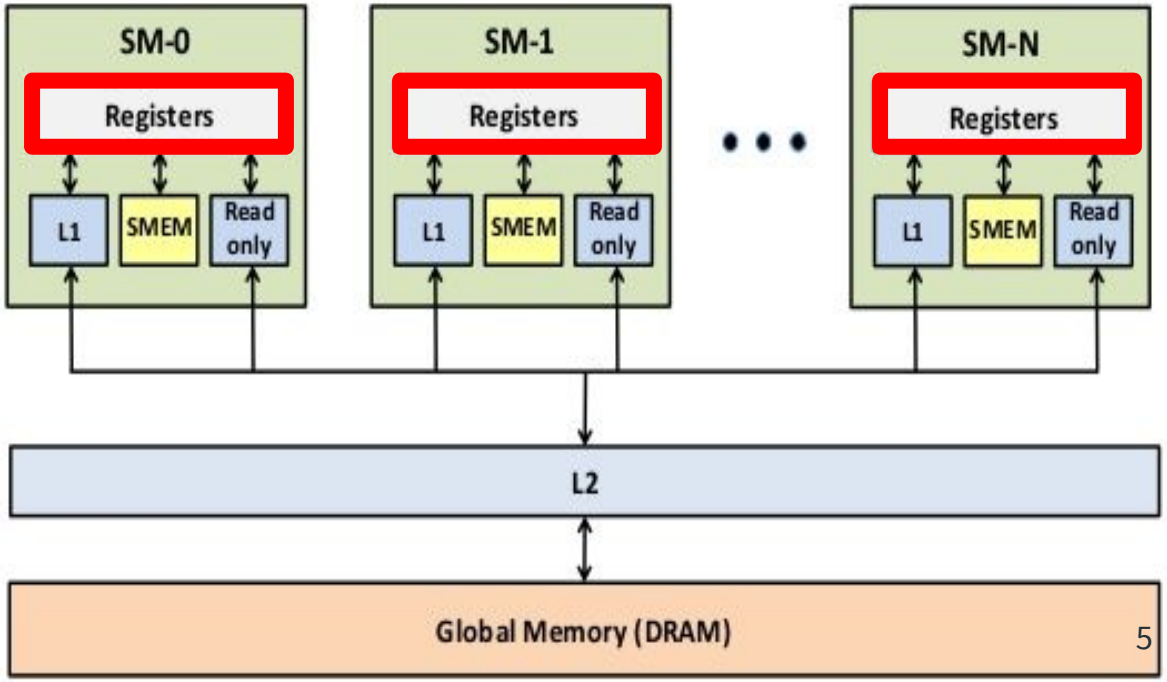
3. Constant Memory

Capacity:	64 KB / SM
Cache	Special cache
Access:	SM-wide
Latency:	1 cycle (hit) 200 - 400 cycles (miss)



4. Register Memory

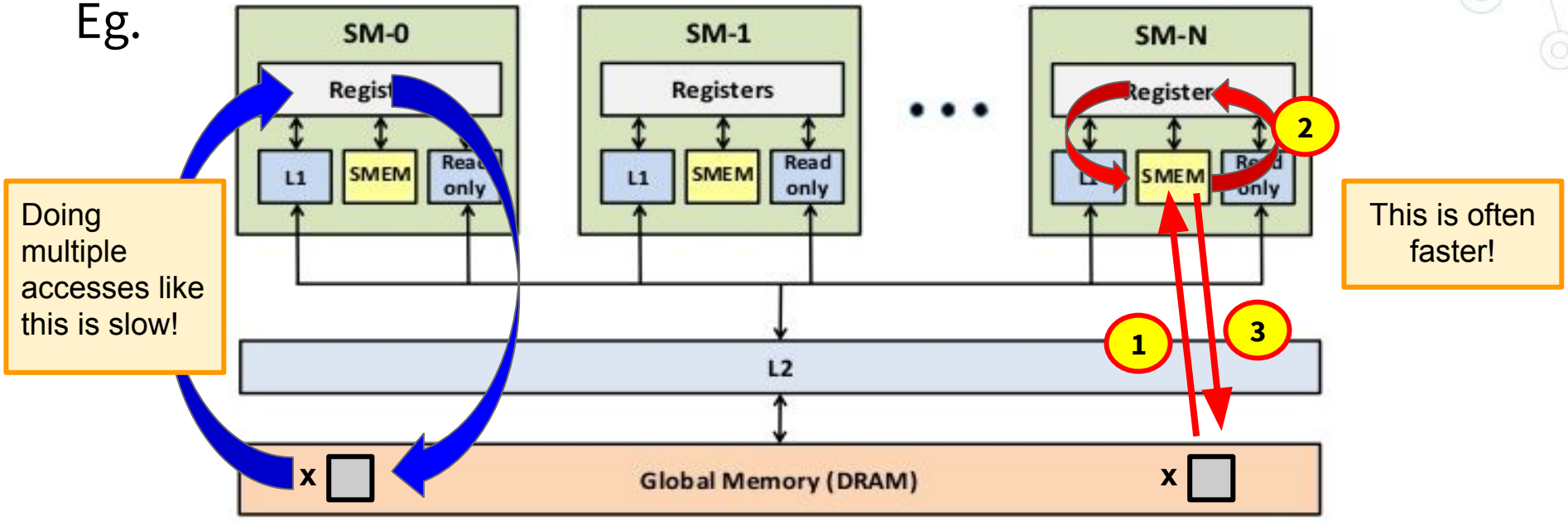
Capacity:	256 KB / SM
Cache	None
Access:	Private to each thread
Latency:	1 cycle



GPU Memory Systems - Summary

- CUDA allows us to choose where we want to store the variables we declare
  - these choices affect the performance of our code

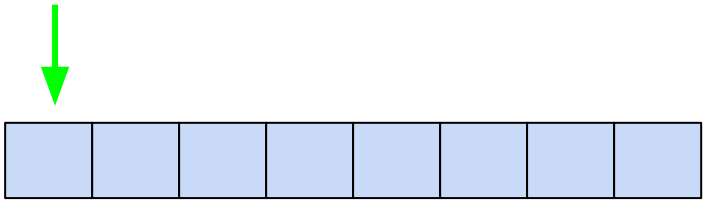
Eg.



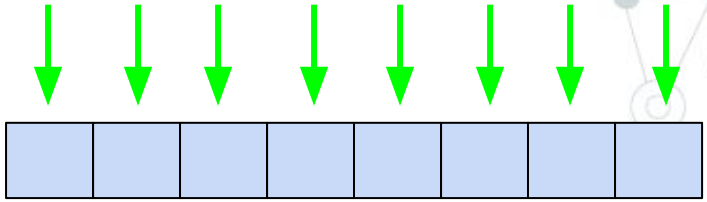
- Performance is also impacted by *memory access patterns*...

Memory Access Patterns

- Parallel hardware increases the demands placed on the memory system

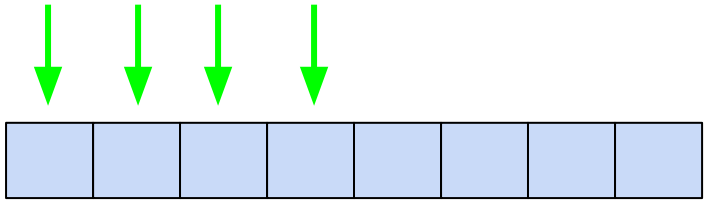


1 Sequential read instruction
(fetches 1 element)

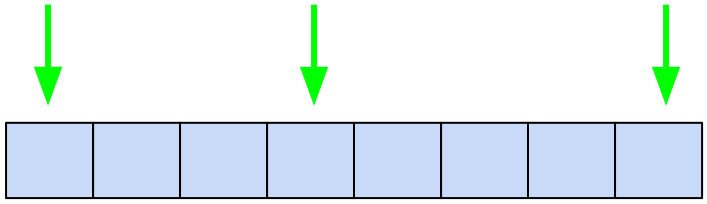


1 Parallel read instruction
(fetches 8x the data!)

- As a result, locality (especially spatial) becomes very important on these systems

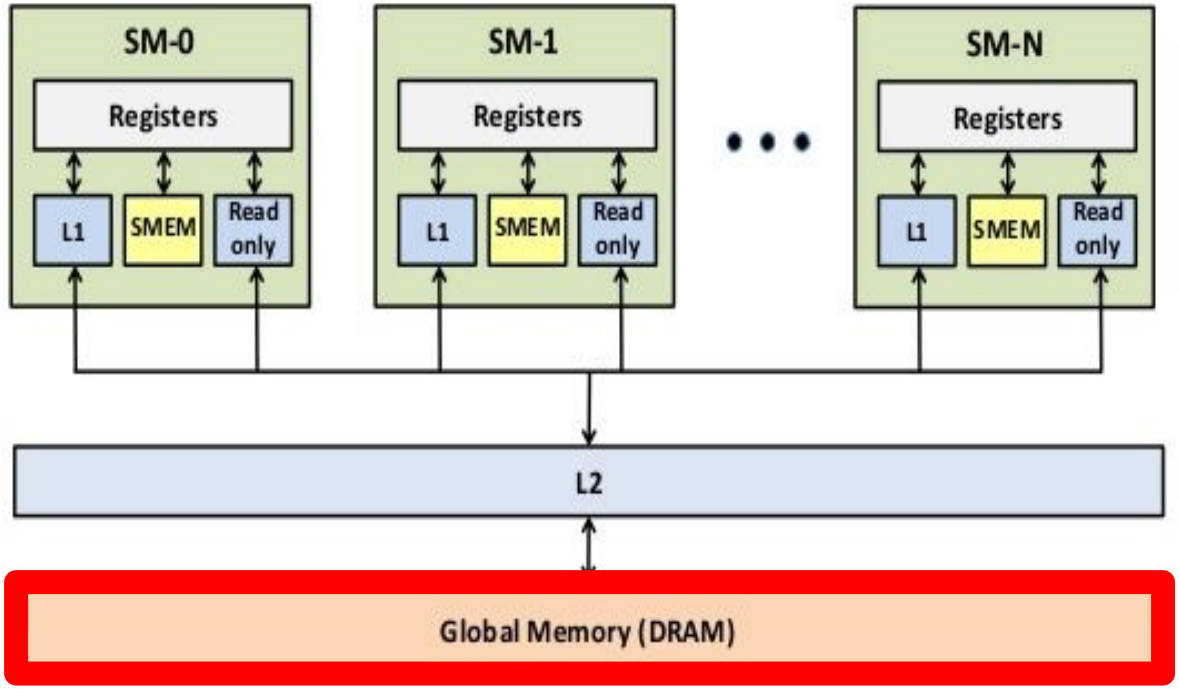


Good Spatial Locality



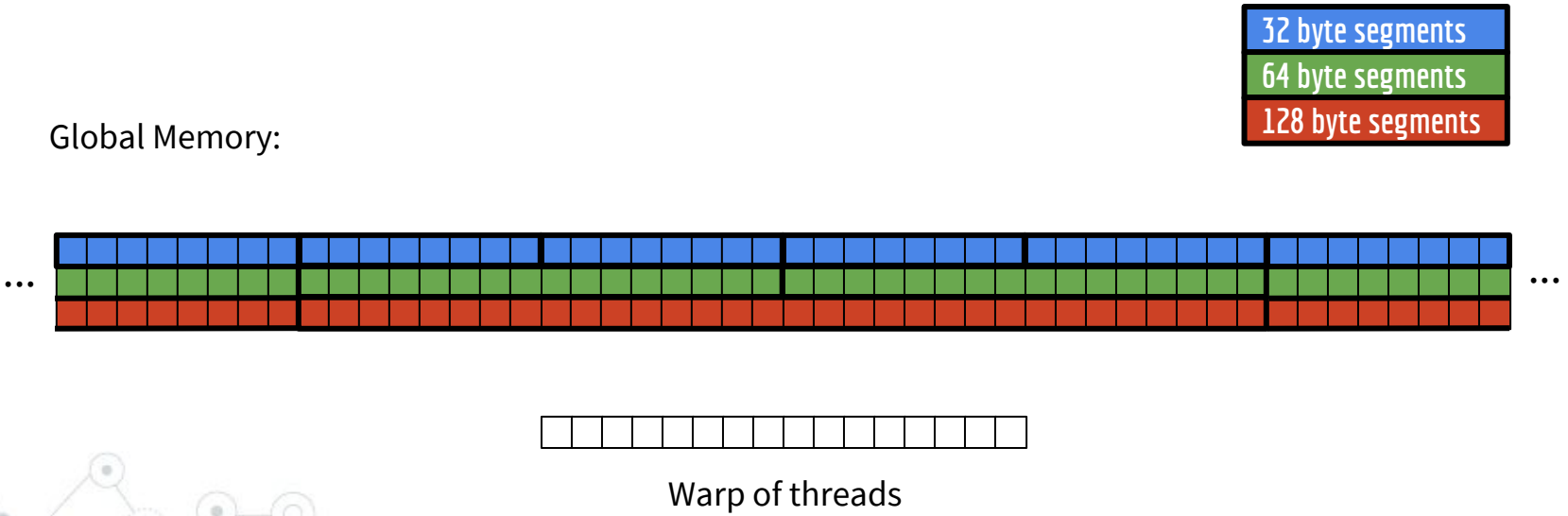
Poor Spatial Locality

Global Memory Access Patterns



Accessing Global Memory

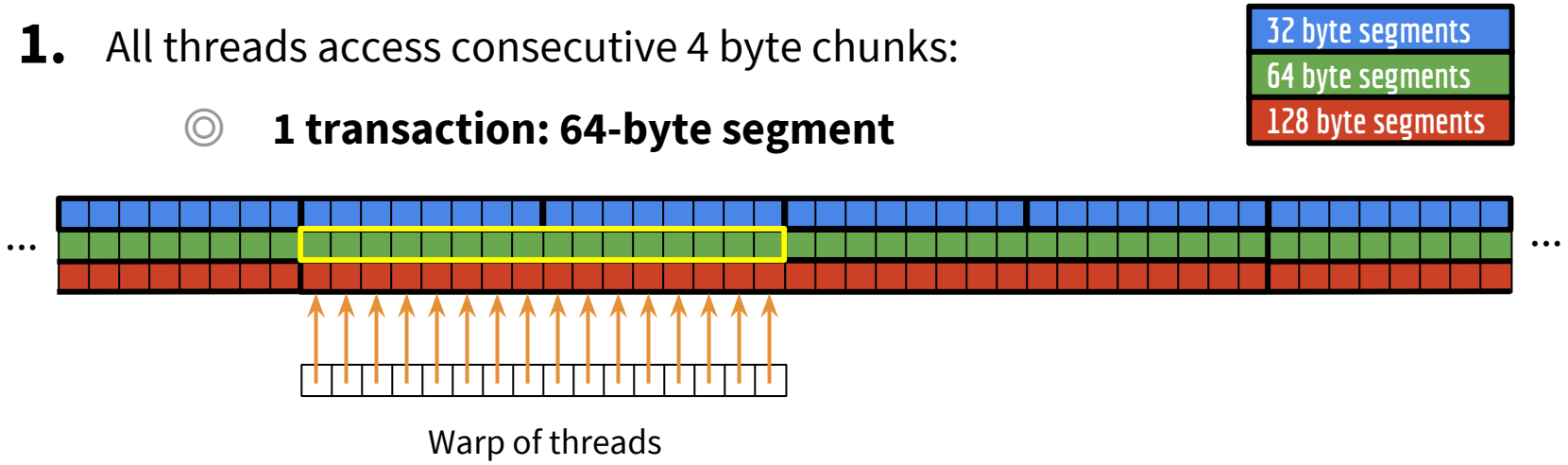
- Global memory is accessed in wide swaths:
  - 32, 64, or 128 byte segments



Accessing Global Memory

1. All threads access consecutive 4 byte chunks:

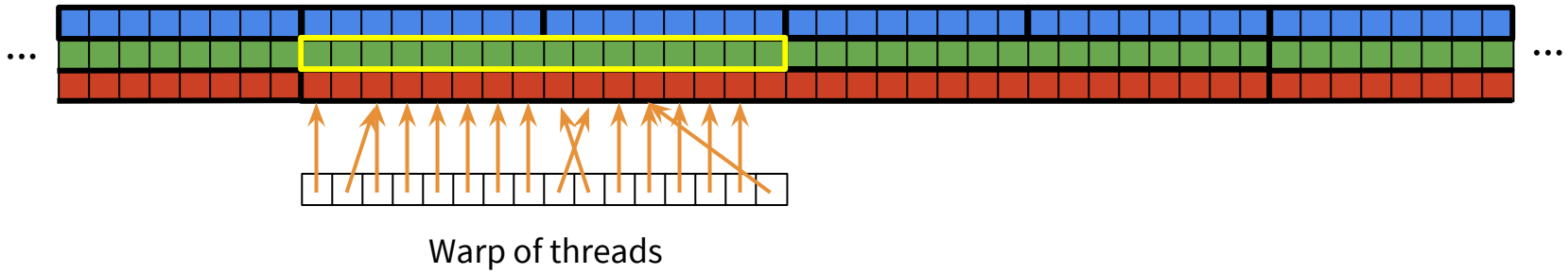
- **1 transaction: 64-byte segment**



Accessing Global Memory

2. All threads access 4 bytes out of order

- **1 transaction: 64-byte segment**



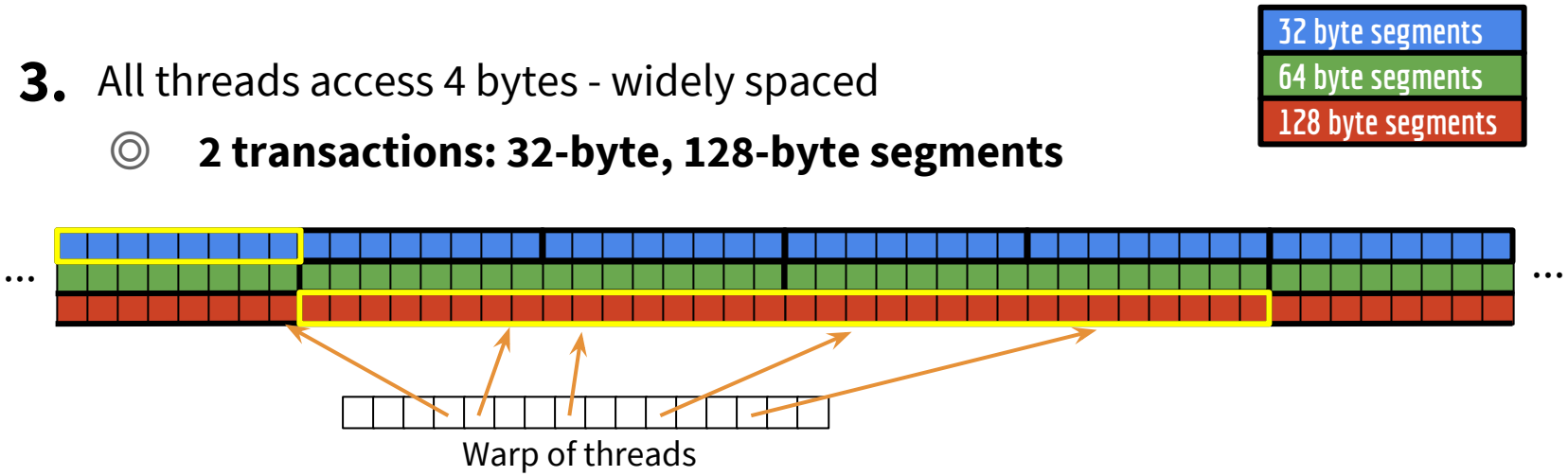
Global memory coalescing

- h/w recognizes all threads are within single 64-byte addressable segment; only 1 transaction is performed

Accessing Global Memory

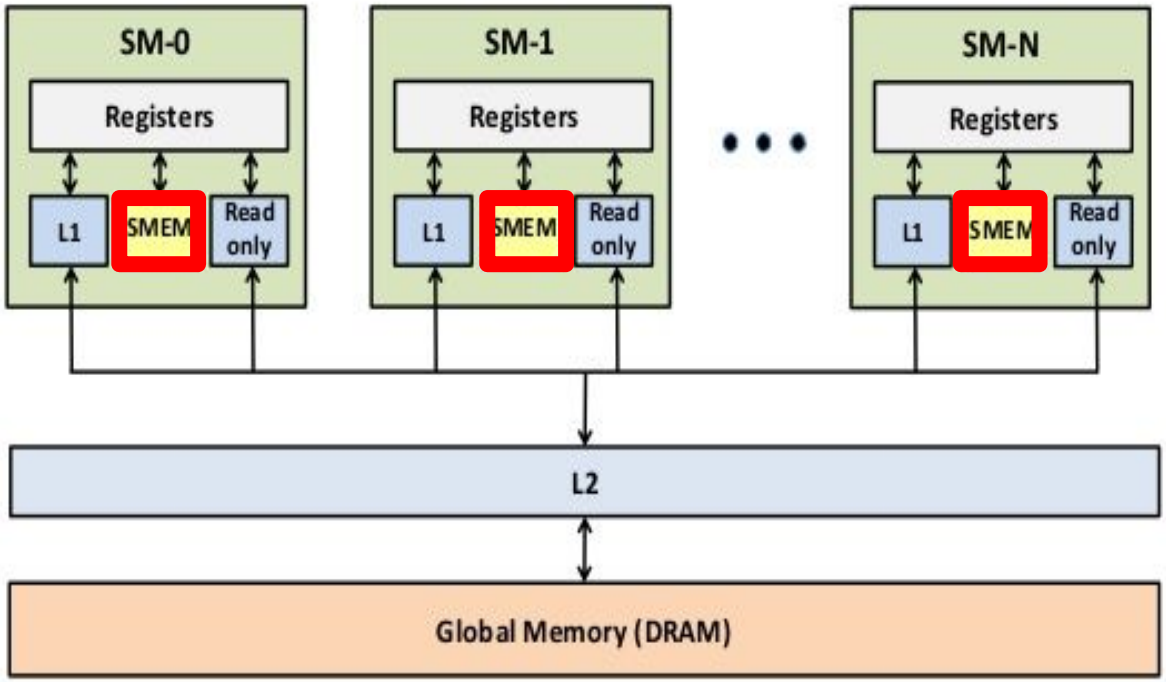
3. All threads access 4 bytes - widely spaced

- **2 transactions: 32-byte, 128-byte segments**



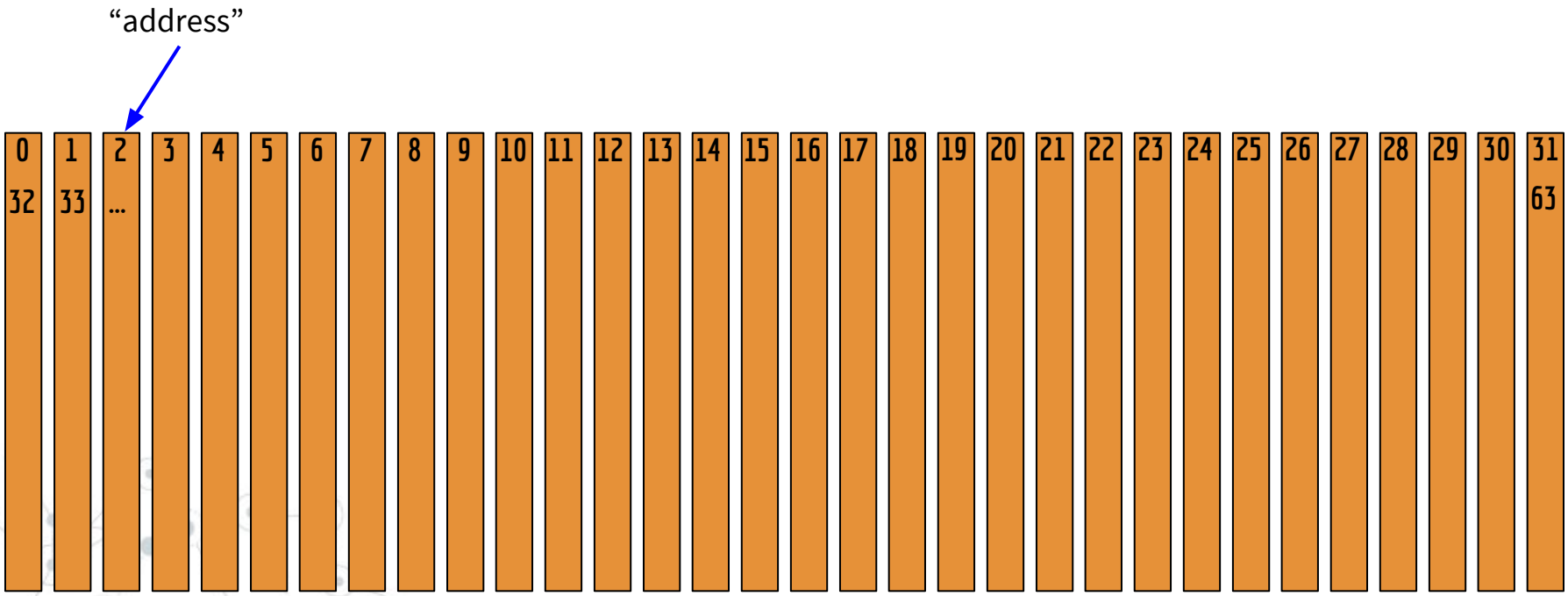
- Warp makes *fewer* requests than in the previous scenario
- But we still require *more* transactions!
  - Due to the *scattered* nature of the requests (poor locality)
- More transactions usually results in poorer performance
  - Memory controller can only do 1 transaction at a time

Shared Memory Access Patterns



Accessing Shared Memory

- Organized into 32 banks
- A bank can handle only 1 request at a time

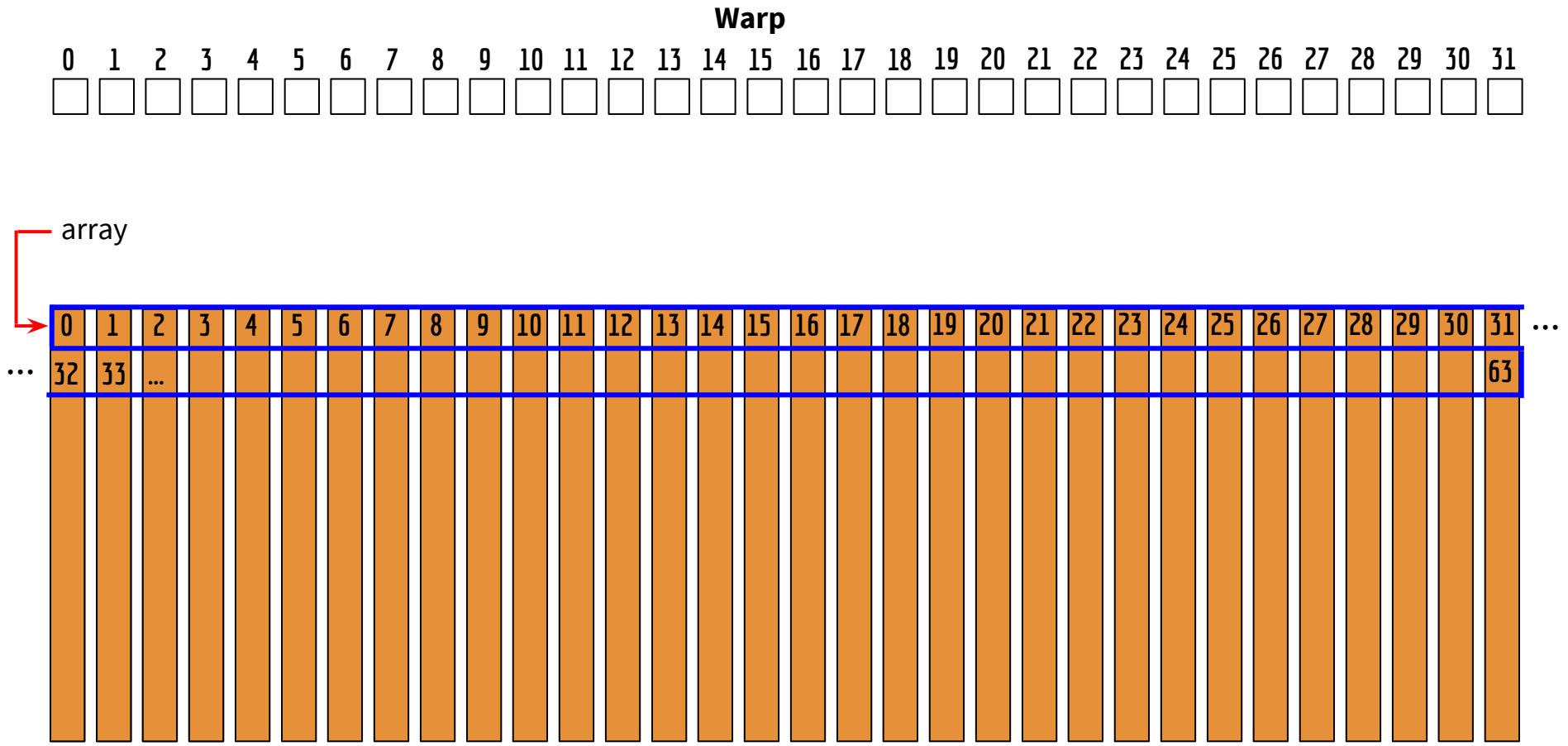


Accessing Shared Memory

- Suppose we have an array:

```
__shared__ float array[64];
```

- Being accessed by a warp of threads:



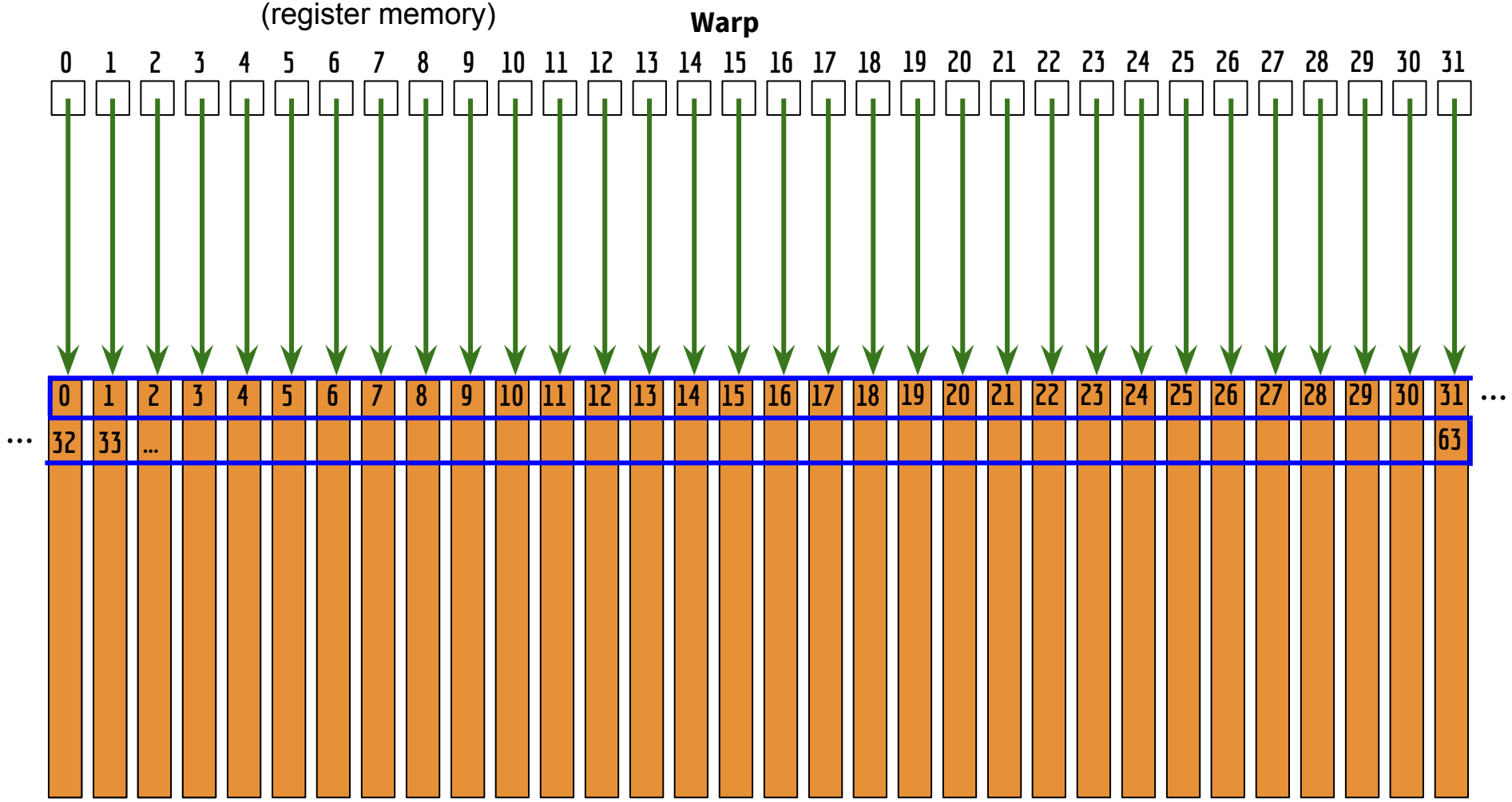
Accessing Shared Memory

1. `float my_val = array[id];`

private variable (register memory)

reads from shared mem array

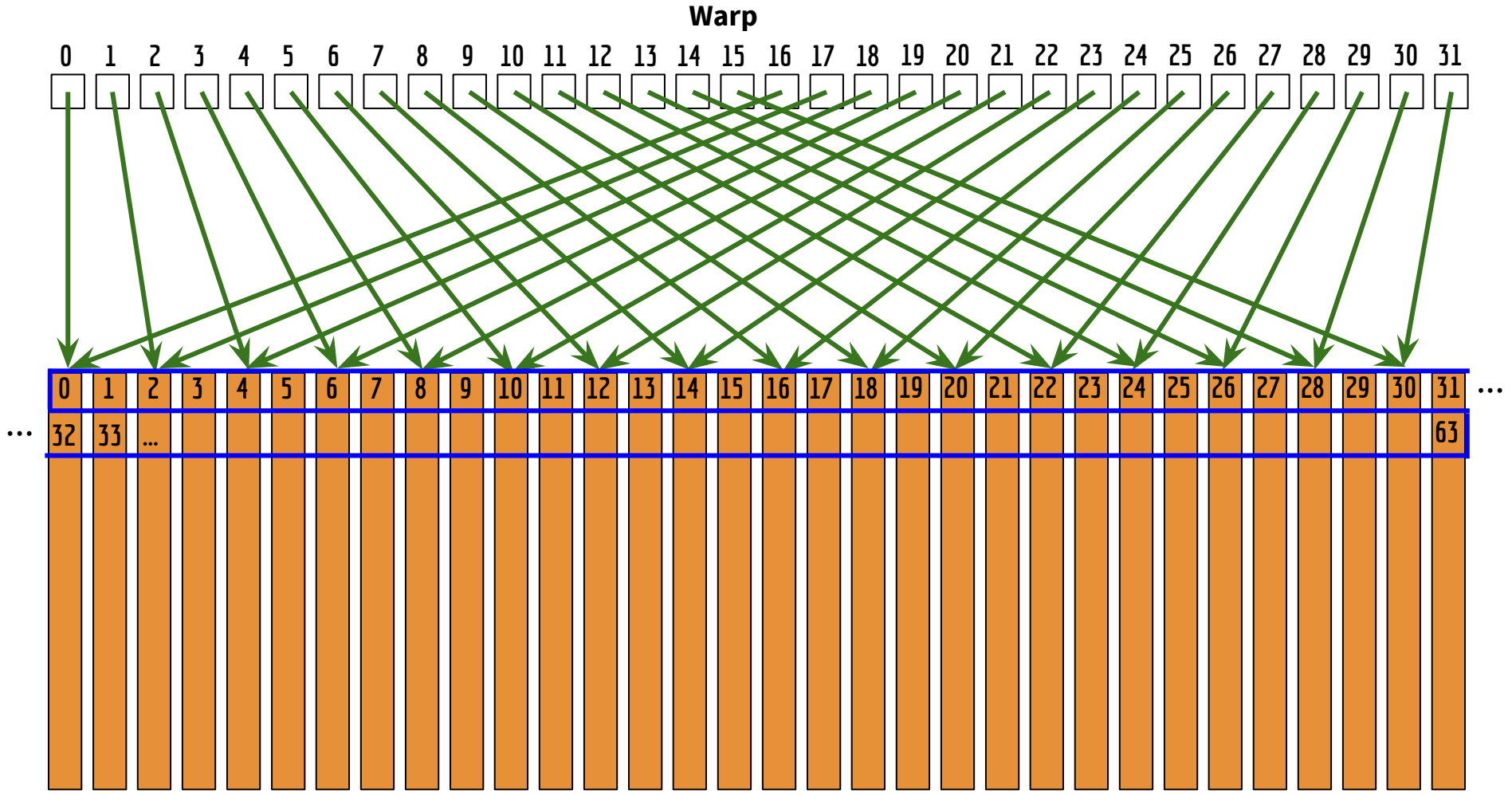
Best case:
32 values in 1 cycle!



Accessing Shared Memory

2. `float my_val = array[id * 2];`

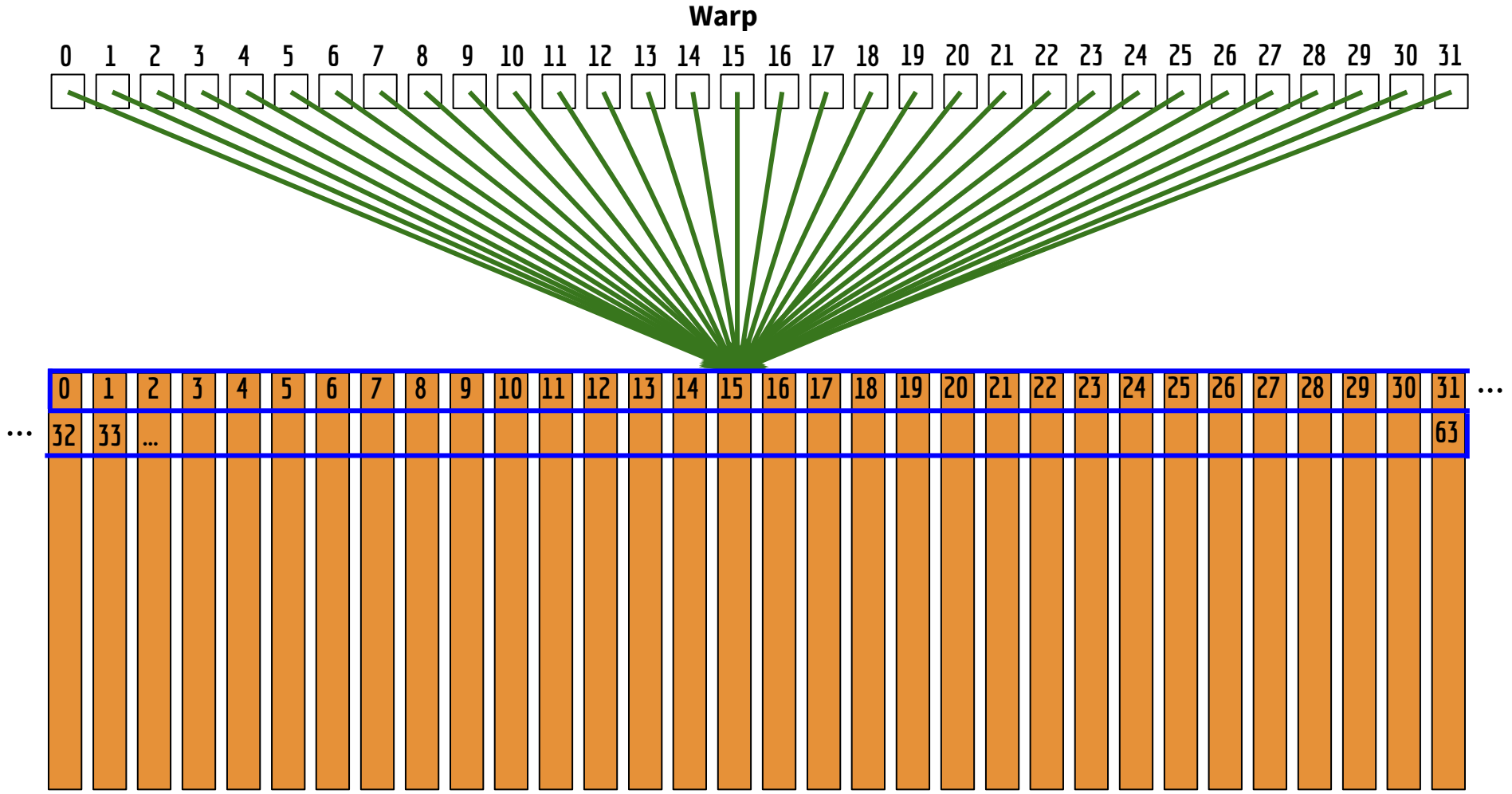
Bank conflict:
16 values on cycle 1
16 values on cycle 2



Accessing Shared Memory

3. `float my_val = array[15];`

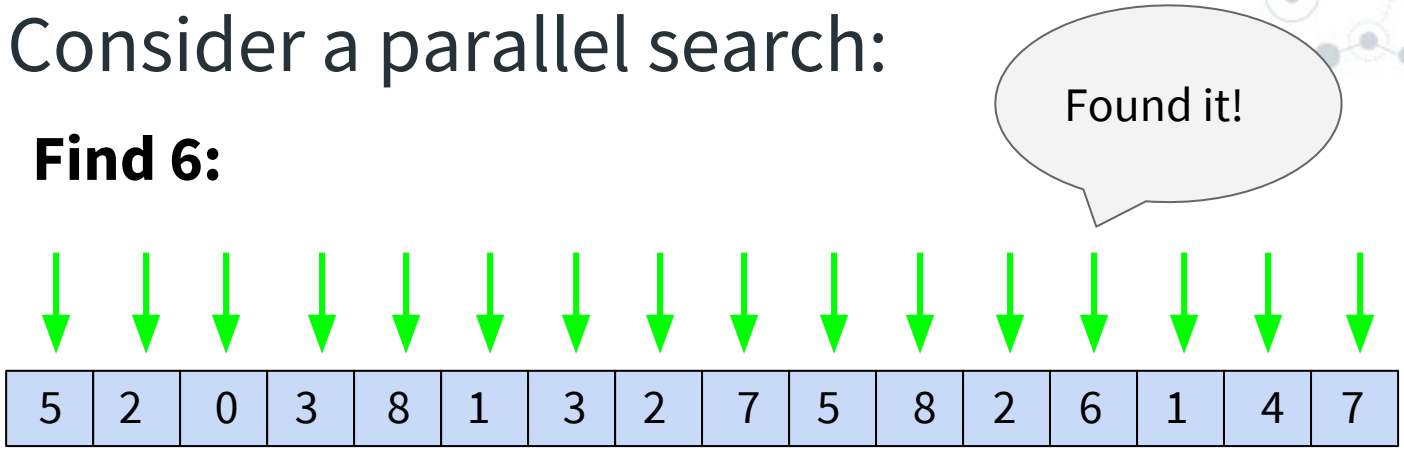
Broadcast feature:
32 bytes in 1 cycle.



Why?

- Consider a parallel search:

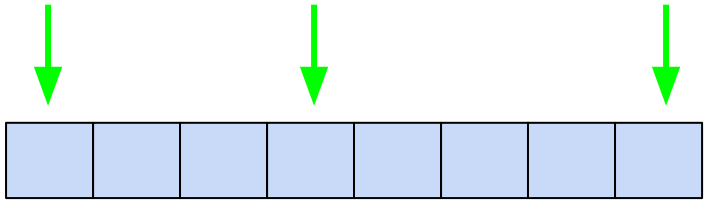
Find 6:



- How do we let everybody know it was found?
  - Broadcast! Finder writes "true" to a shared variable, then everyone reads the variable (at the same time).

GPU Algorithm Design - Rules of Thumb

1. Try to arrange your data so that accesses by different threads are close together



Poor Spatial Locality

2. Try to partition your data so that you can give **independent** work to each SM