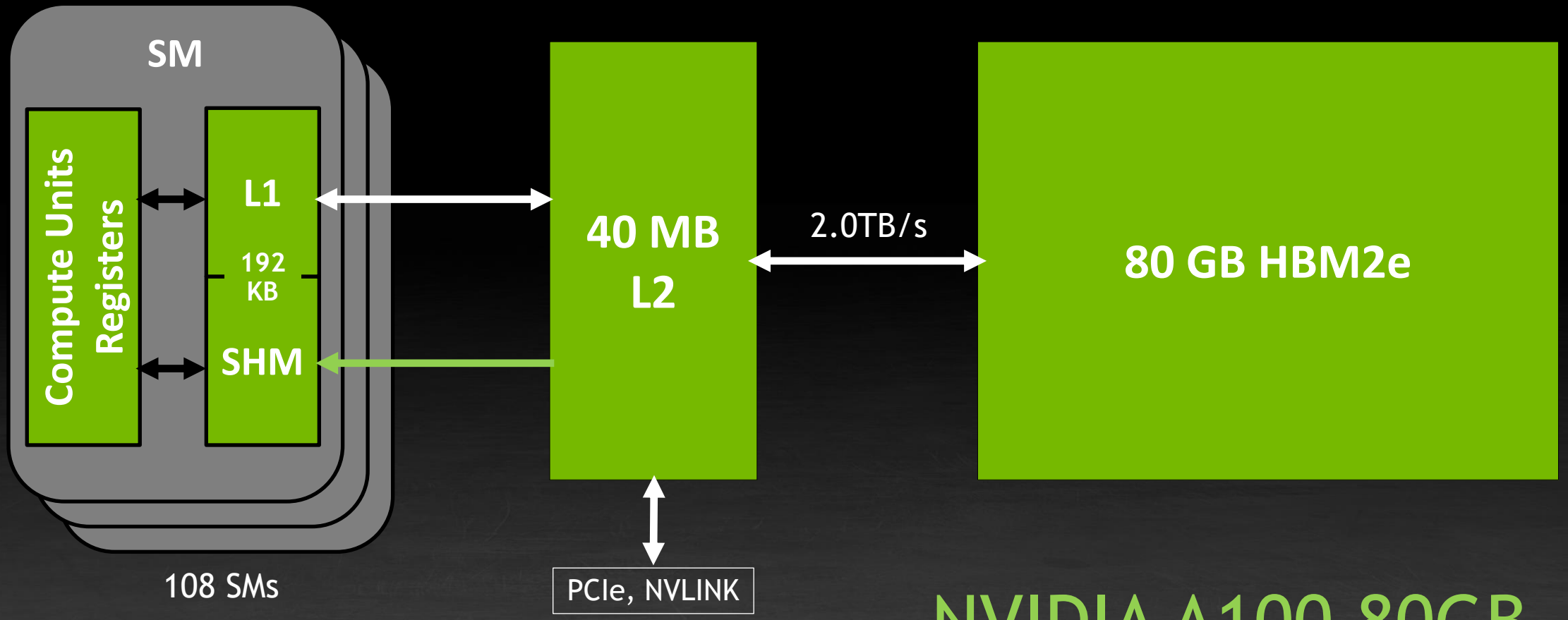
PERFORMANCE OPTIMIZATION WITH MODERN CUDA PROGRAMMING TECHNIQUES

GUILLAUME THOMAS-COLLIGNON, VISHAL MEHTA
DEVTECH COMPUTE


• Memory Hierarchy
▪ Memory Access Patterns
▪ Memory Model
▪ L2 Cache
▪ Shared Memory
• Thread Divergence
• GPU Occupancy
• CUDA streams & Graphs

MEMORY HIERARCHY

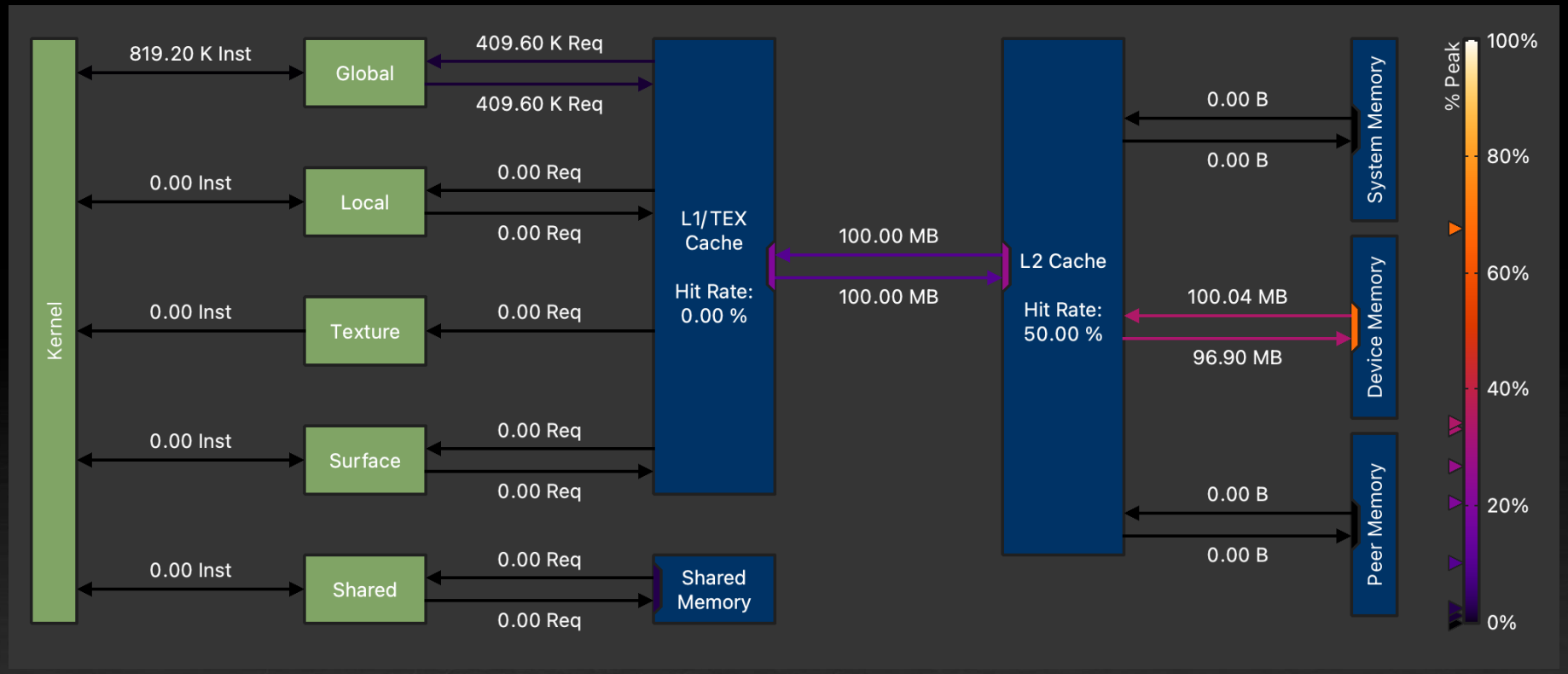
Understanding Memory and Caches



NVIDIA A100 80GB

MEMORY HIERARCHY

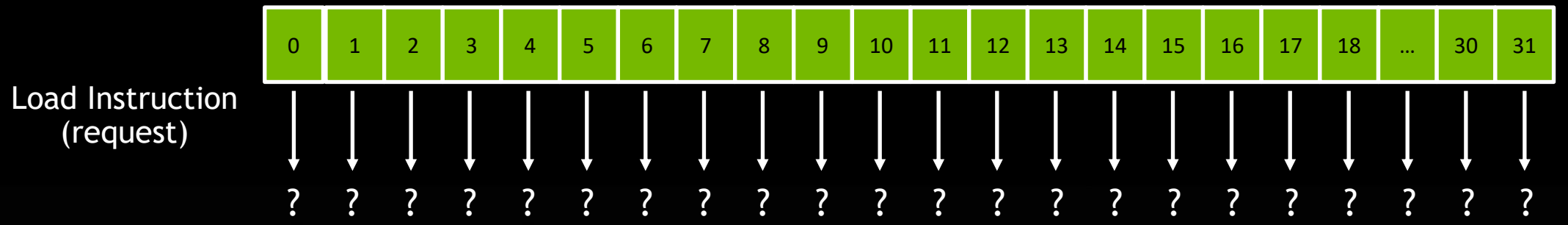
Nsight Compute View



MEMORY HIERARCHY

Instructions, Requests, Sectors

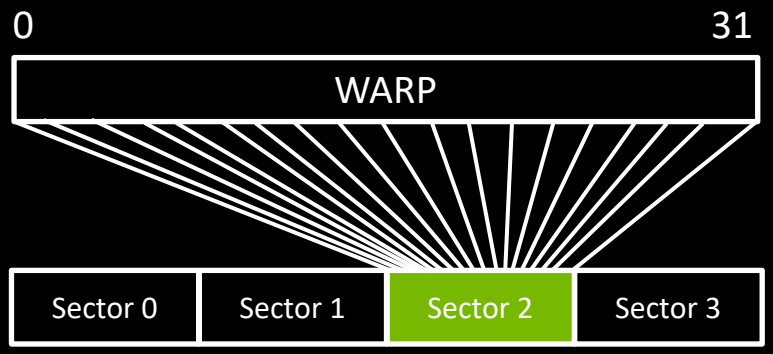
One warp = 32 Threads



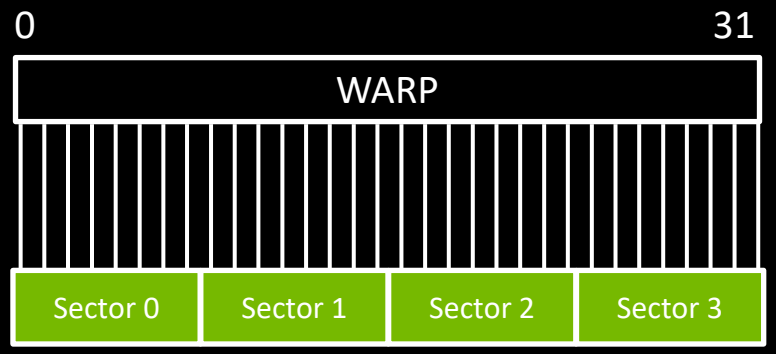
Granularity = 32 Byte sector

How many different 32-Byte sectors are being touched by the WARP?

MEMORY ACCESS PATTERNS



1-byte per thread coalesced memory access

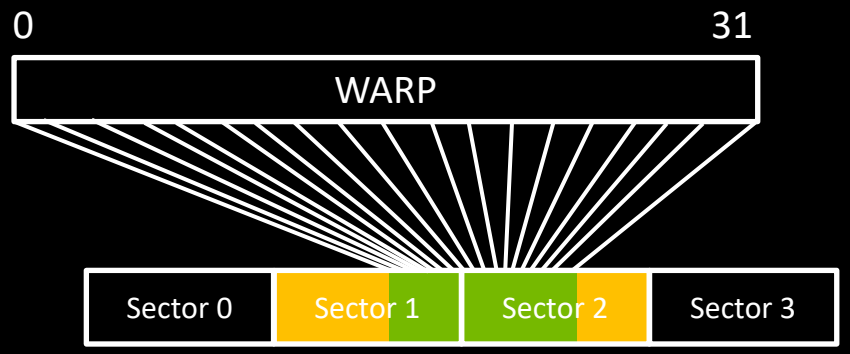


4-bytes per thread coalesced memory access

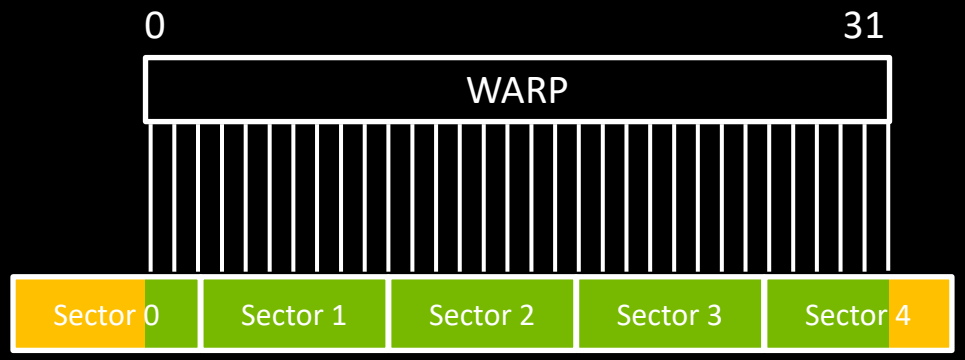
Coalesced memory accesses, touching only the ideal number of sectors

8-bytes per thread coalesced memory access

MEMORY ACCESS PATTERNS



1-byte per thread unaligned memory access

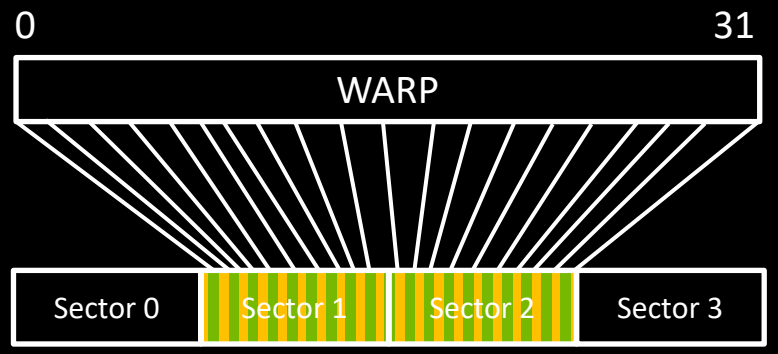


4-bytes per thread unaligned memory access

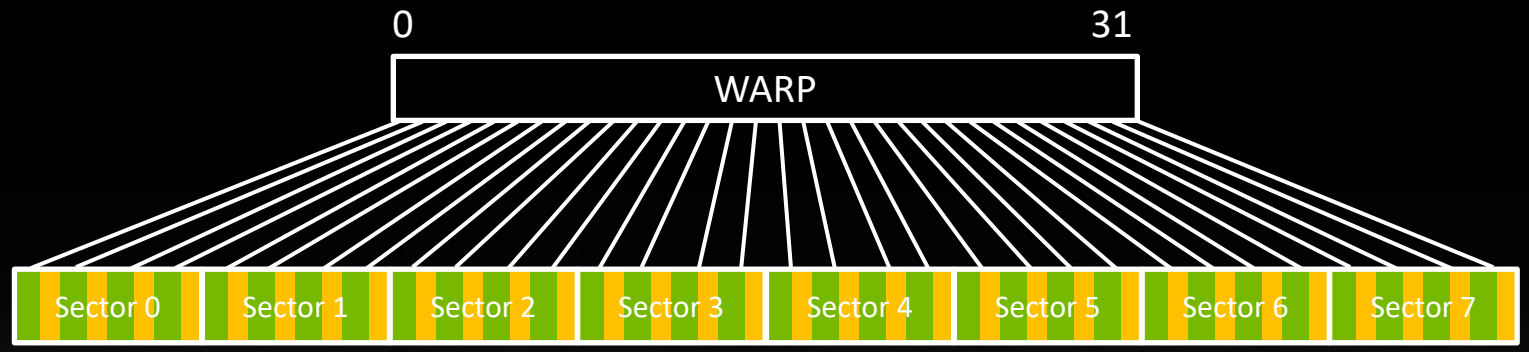
Coalesced but unaligned memory accesses will increase the number of sectors per request

8-bytes per thread unaligned memory access

MEMORY ACCESS PATTERNS



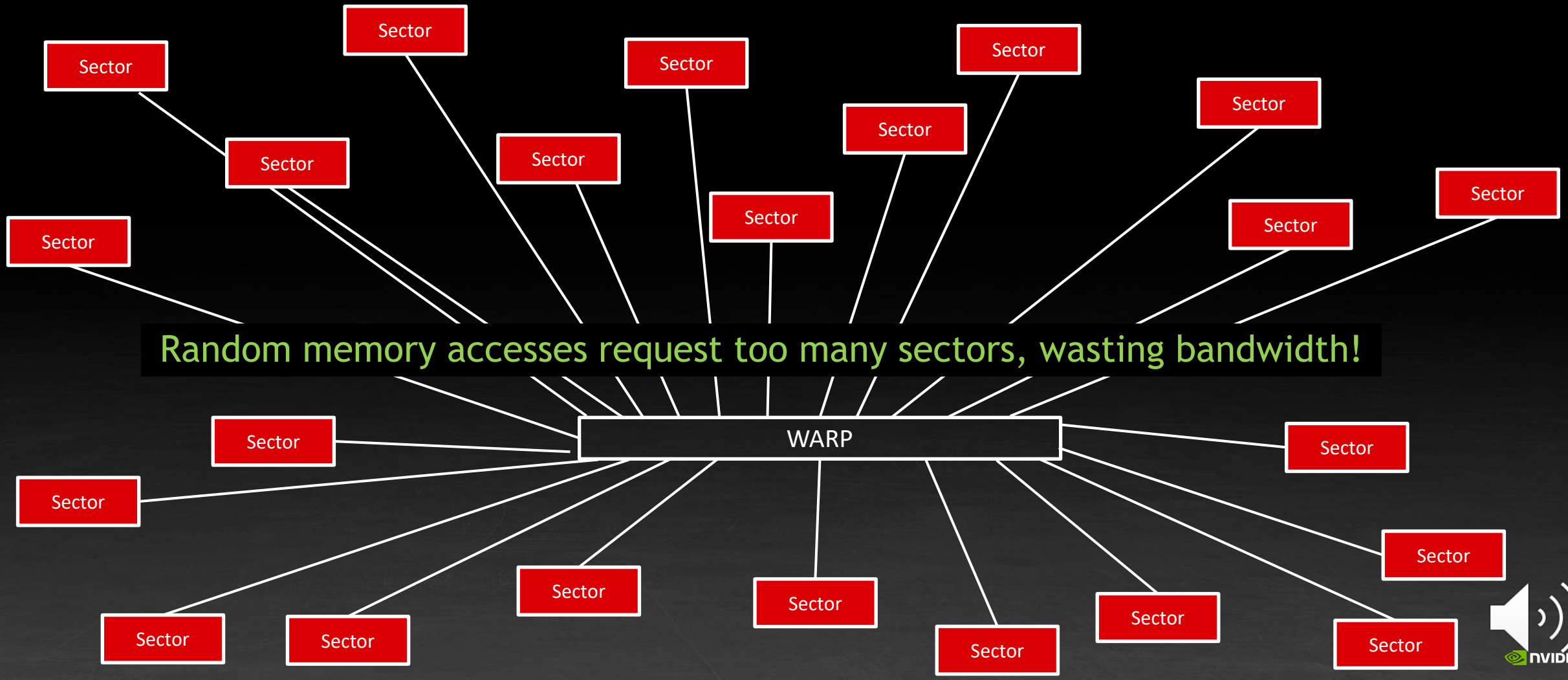
1-byte per thread, stride = 2



4-bytes per thread, stride = 2

Strided memory accesses also increase the number of sectors per request

MEMORY ACCESS PATTERNS



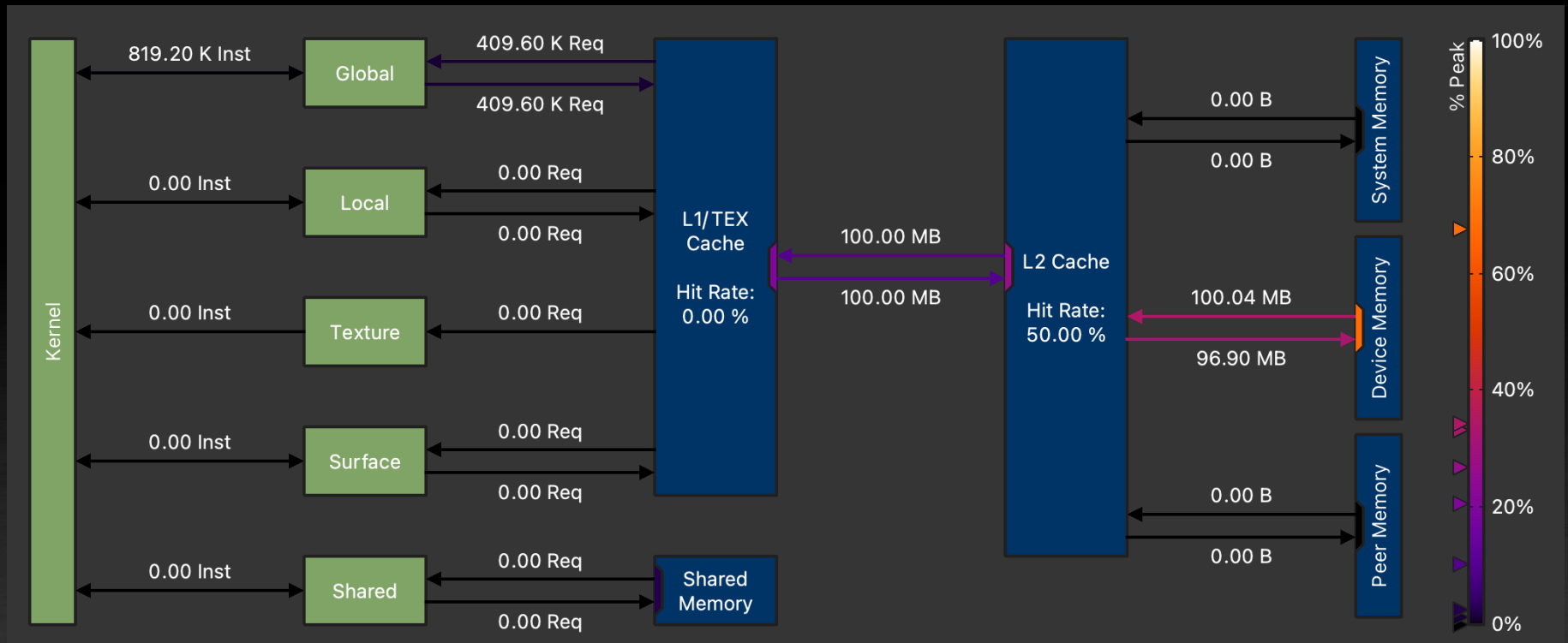
MEMORY HIERARCHY

Nsight Compute

Instructions generate requests to L1 cache

32B sectors are transferred between L1 and L2 caches

and eventually to/from memory

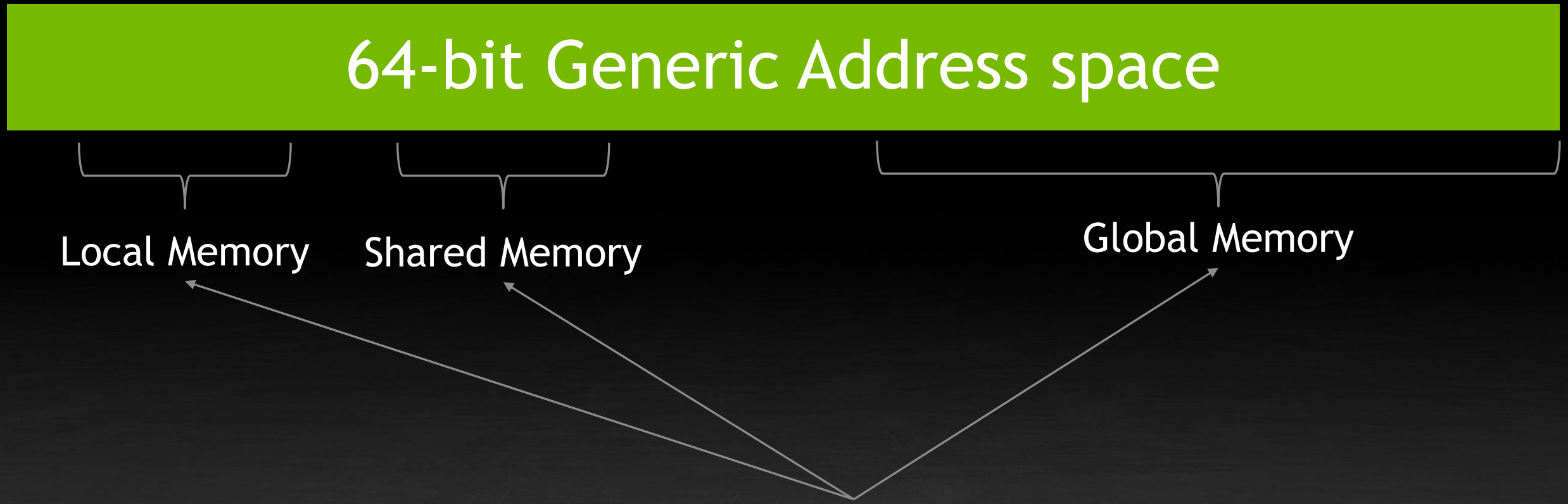


Make sure these numbers match your expectations

MEMORY MODEL

MEMORY MODEL

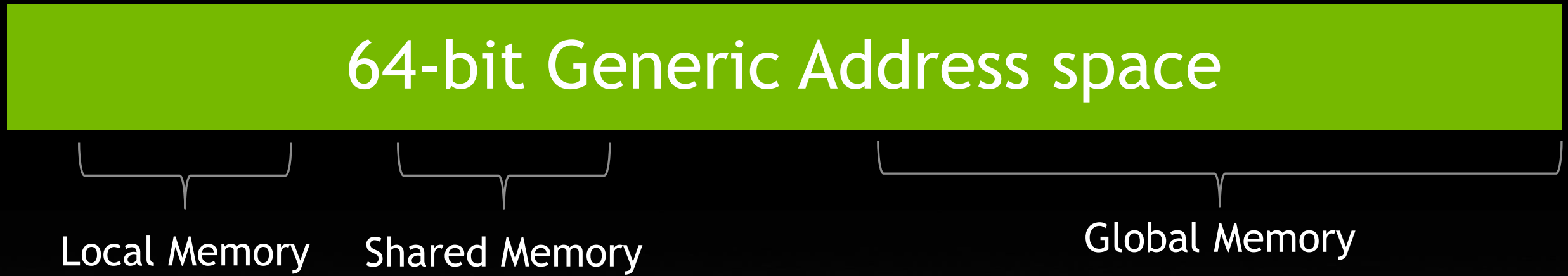
CUDA Generic Address Space



- Single Generic address space Visible to Thread
- Fully C++ object model conforming

MEMORY MODEL

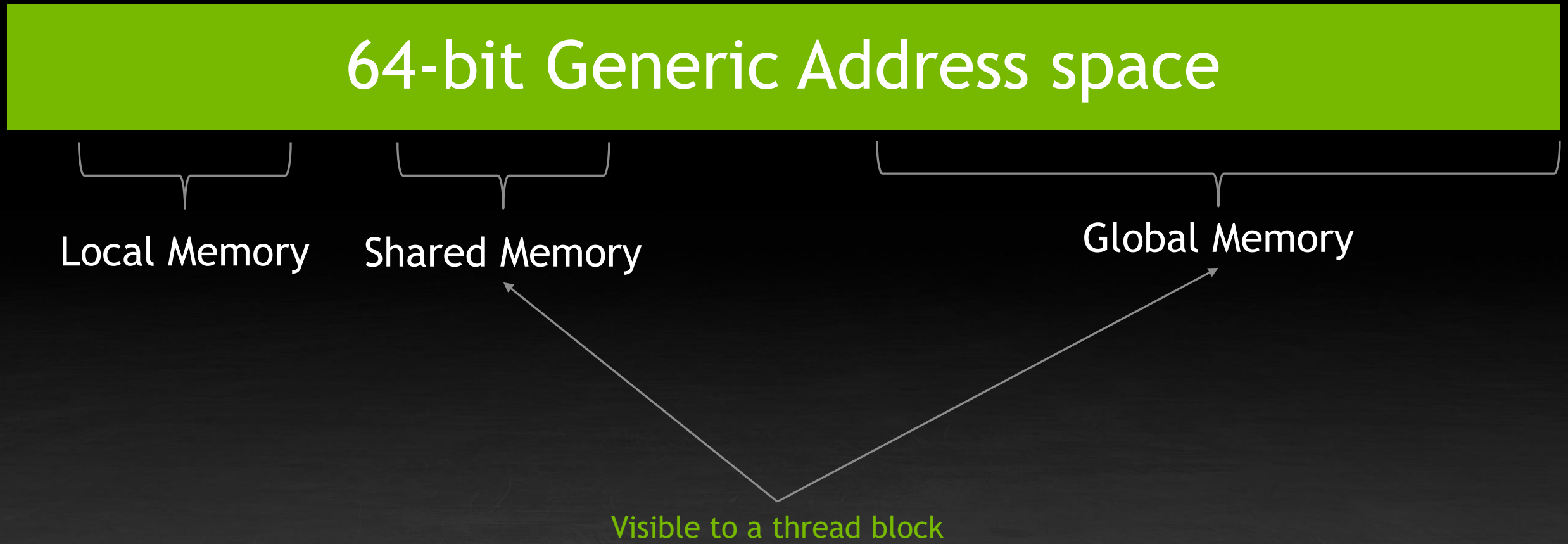
CUDA Generic Address Space




```
__device__ float square(float* data) {
    float x = data[0] // Memory operations can be performed with generic address
                      // agnostic of shared, local or global memory
                      // Compiler can optimize if it can determine the address space
    return (x*x);
}
```

MEMORY MODEL

CUDA Generic Address Space



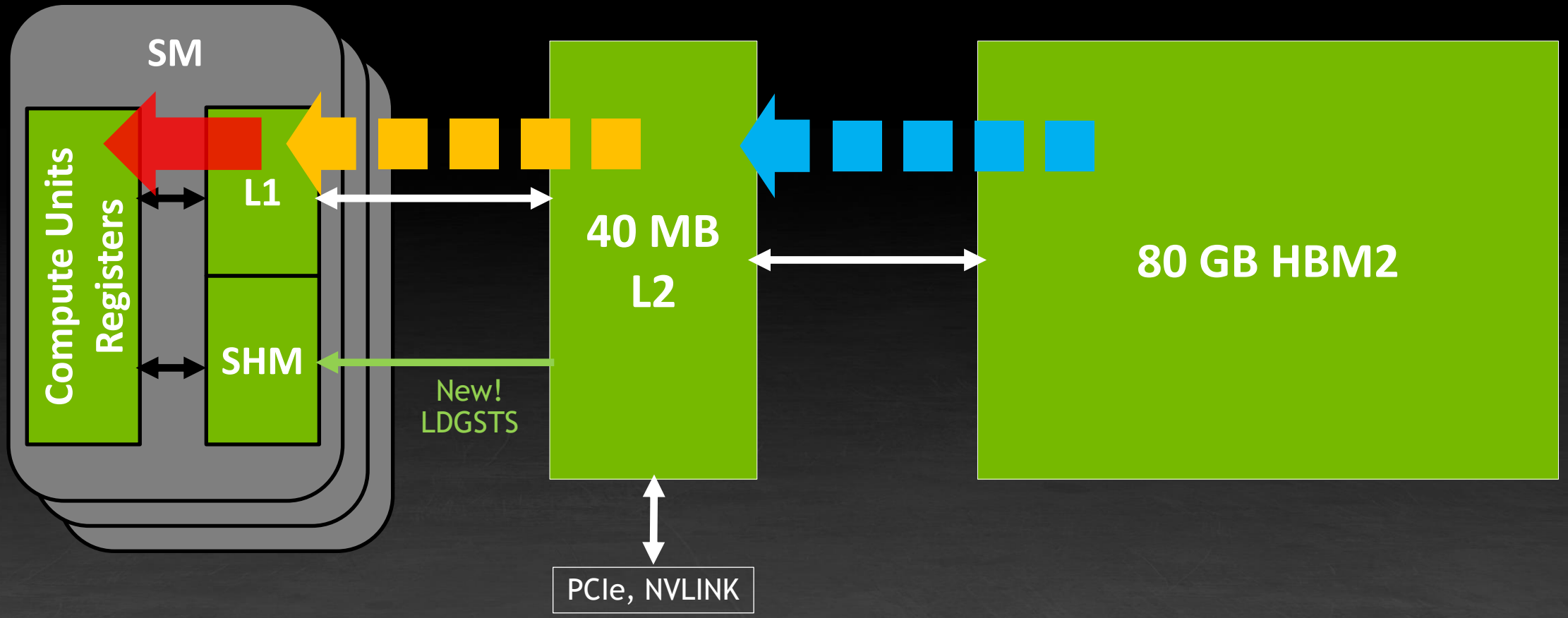
MEMORY MODEL

CUDA Generic Address Space

MEMORY MODEL

Reads

Reading from local / Global Mem can hit in L1 or L2

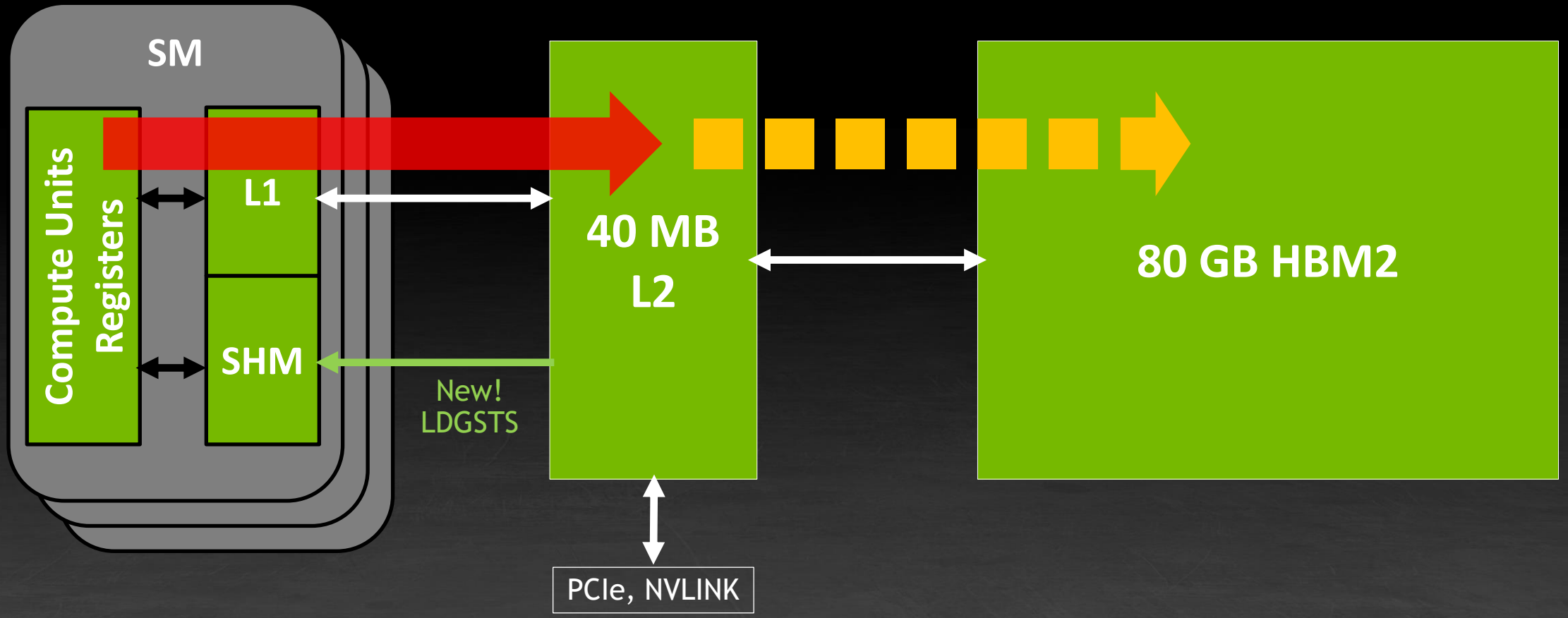


MEMORY MODEL

Writes

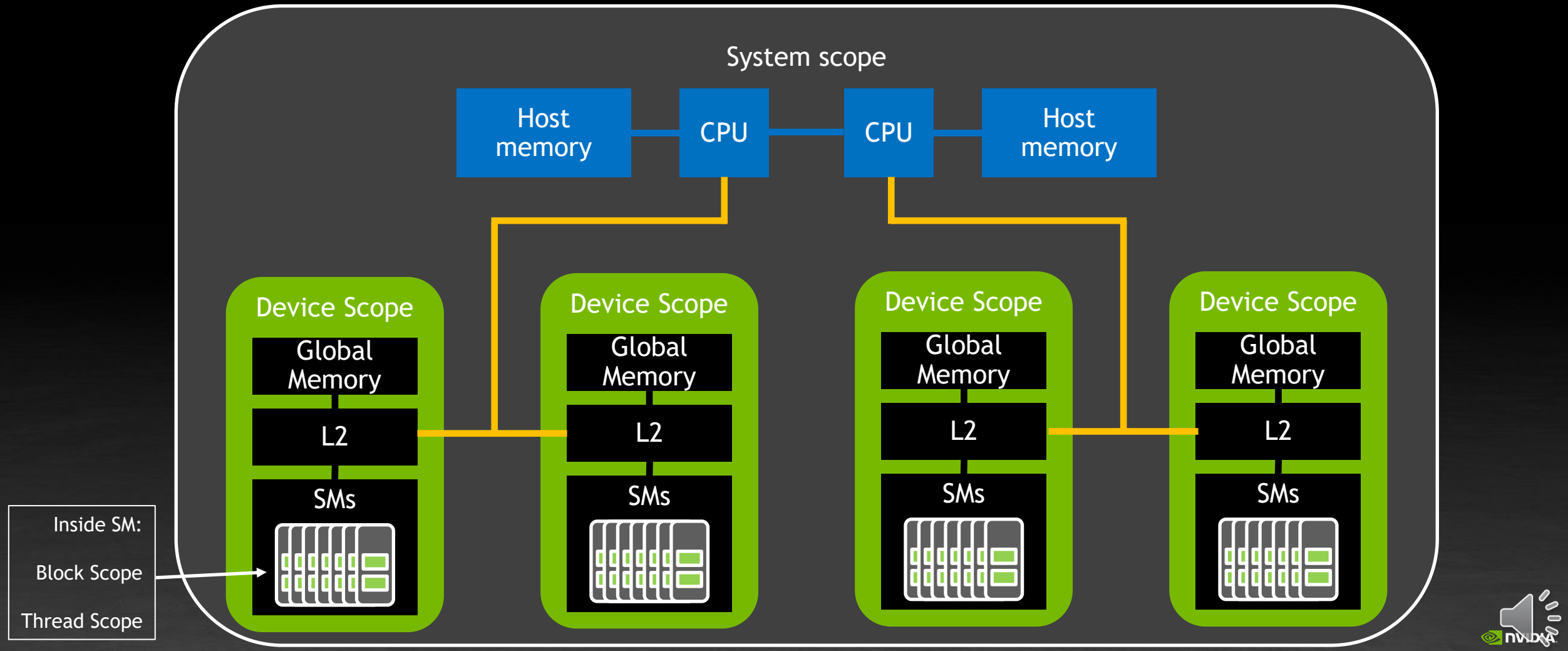
L1 is write-through, L2 is write-back

Writes will always reach at least L2, Read After Write can hit in L1 (e.g. register spills)



MEMORY MODEL

Scope levels



MEMORY MODEL

Synchronizing memory between multiple Thread Blocks

Cooperative Kernel

L1 Cache

```
__host__ __device__
int poll_then_read(int& flag, int& data)
{
    while (flag != 1) ;
    return data;
}
```

Reader: Thread 0, Block 0, SM 0

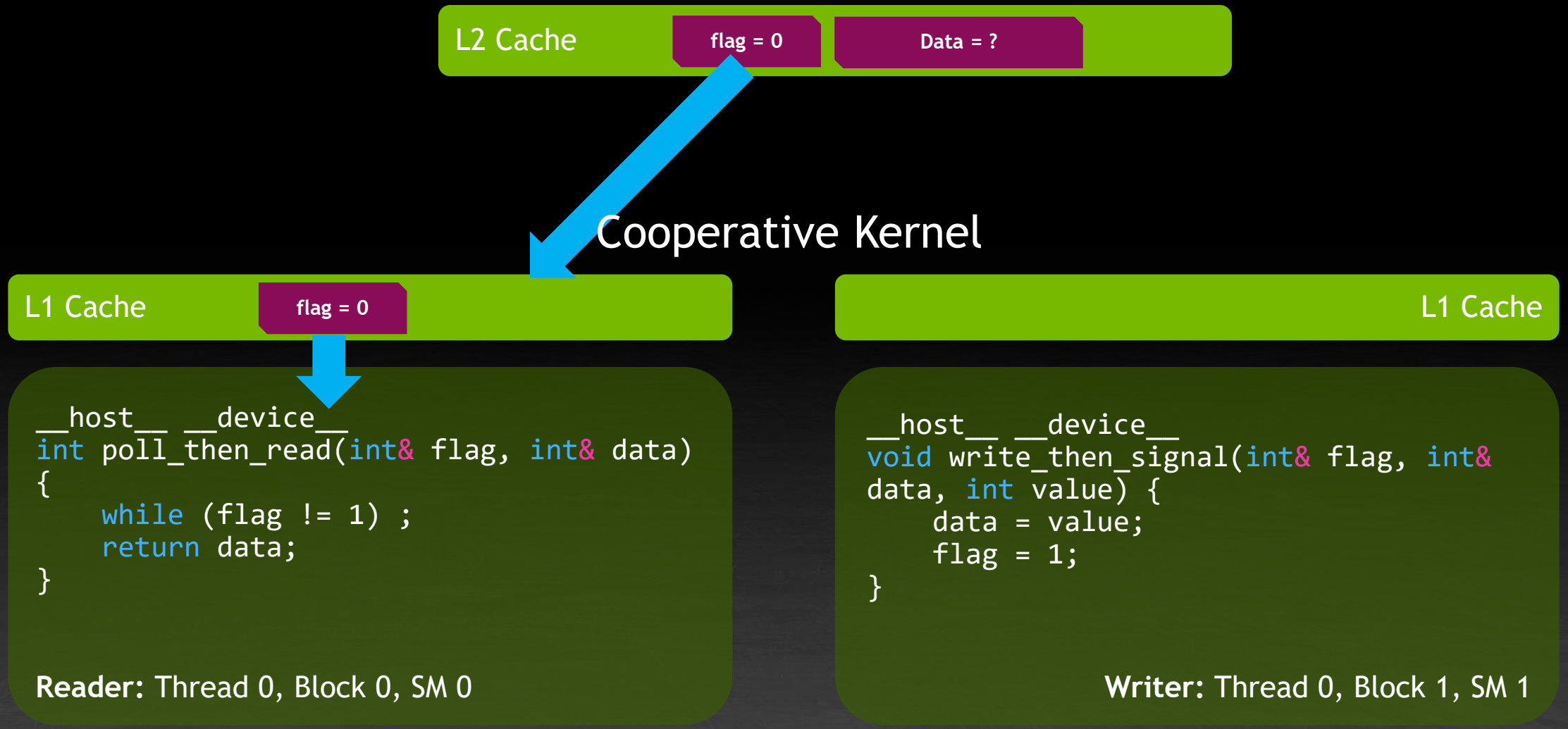
L1 Cache

```
__host__ __device__
void write_then_signal(int& flag, int&
data, int value) {
    data = value;
    flag = 1;
}
```

Writer: Thread 0, Block 1, SM 1

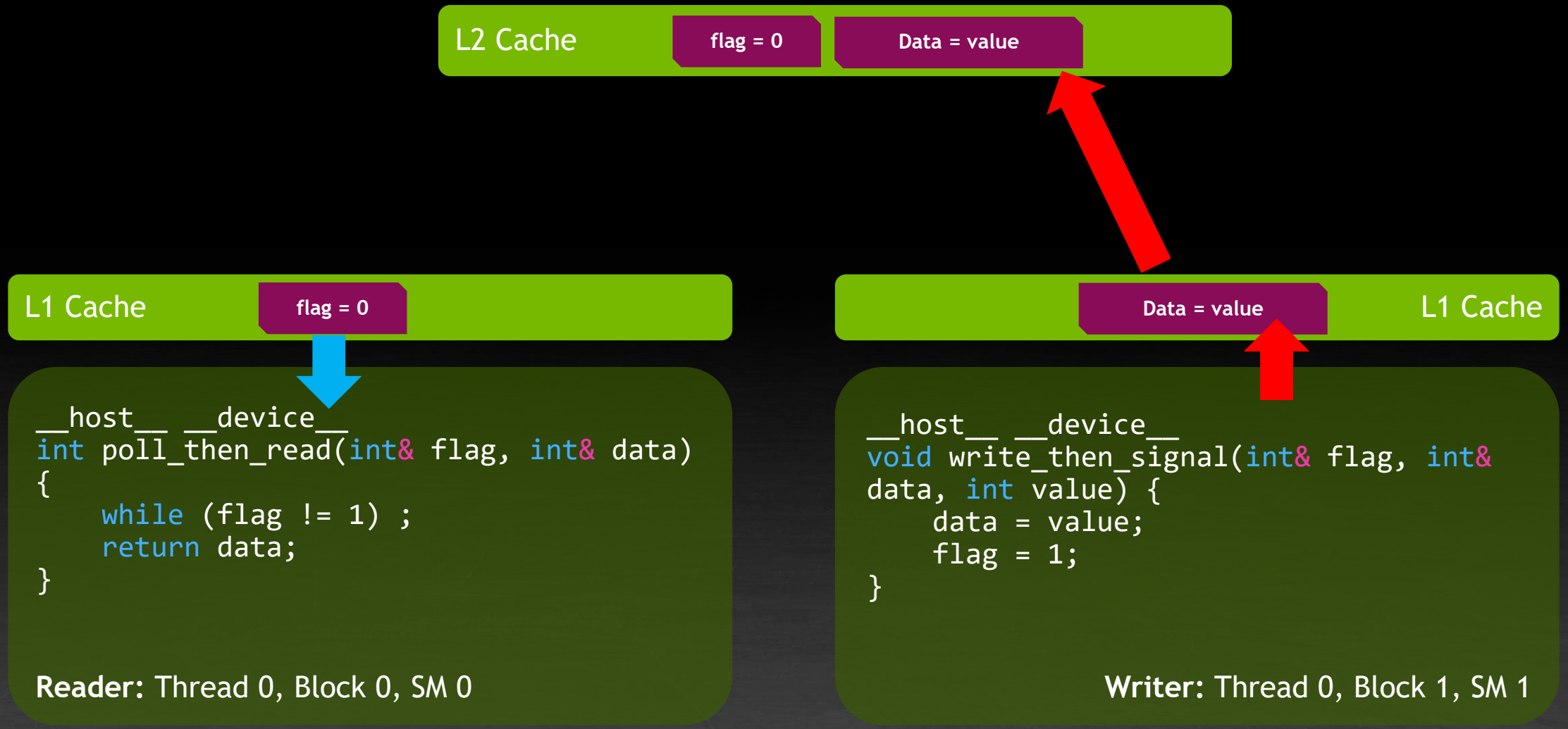
MEMORY MODEL

Synchronizing memory between multiple Thread Blocks



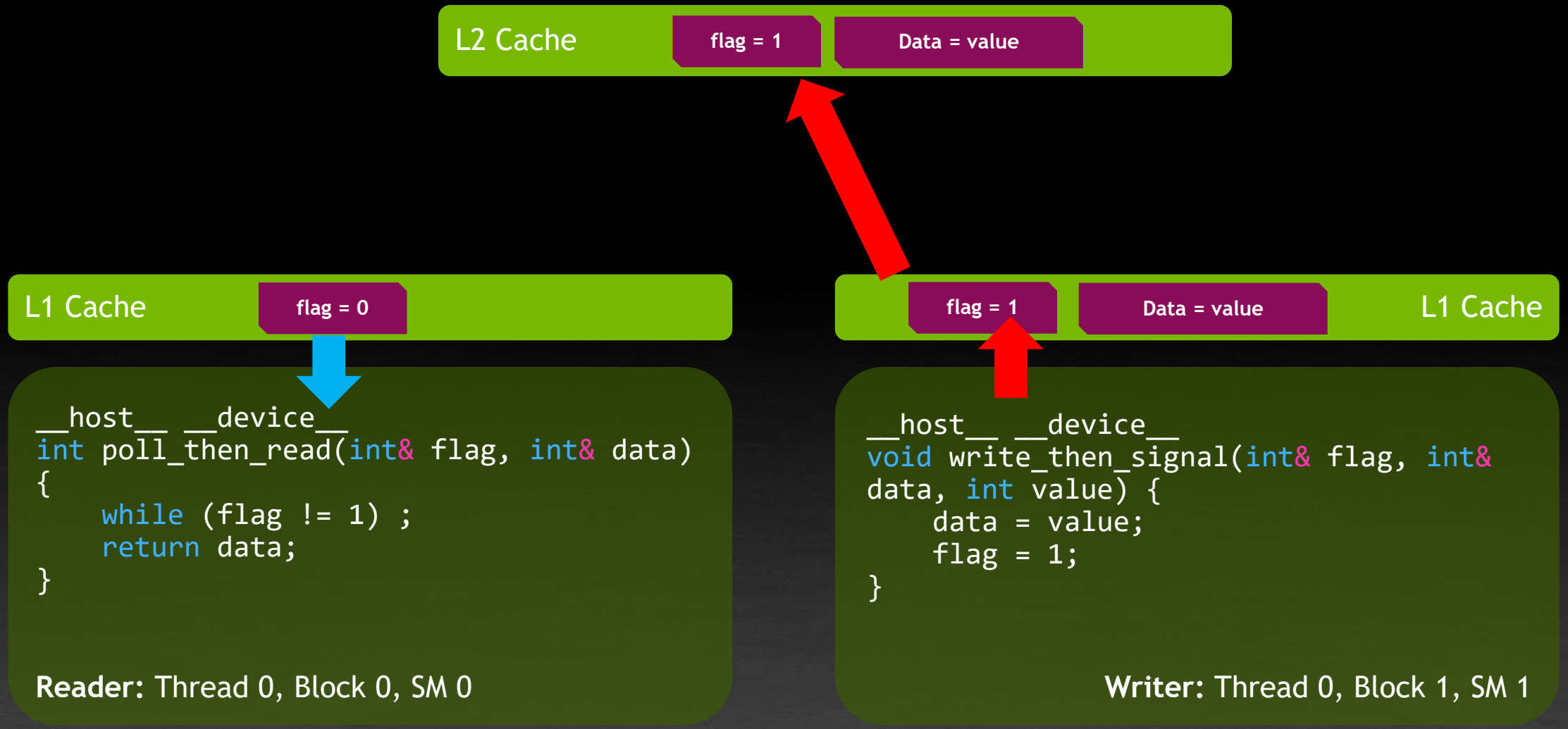
MEMORY MODEL

Synchronizing memory between multiple Thread Blocks



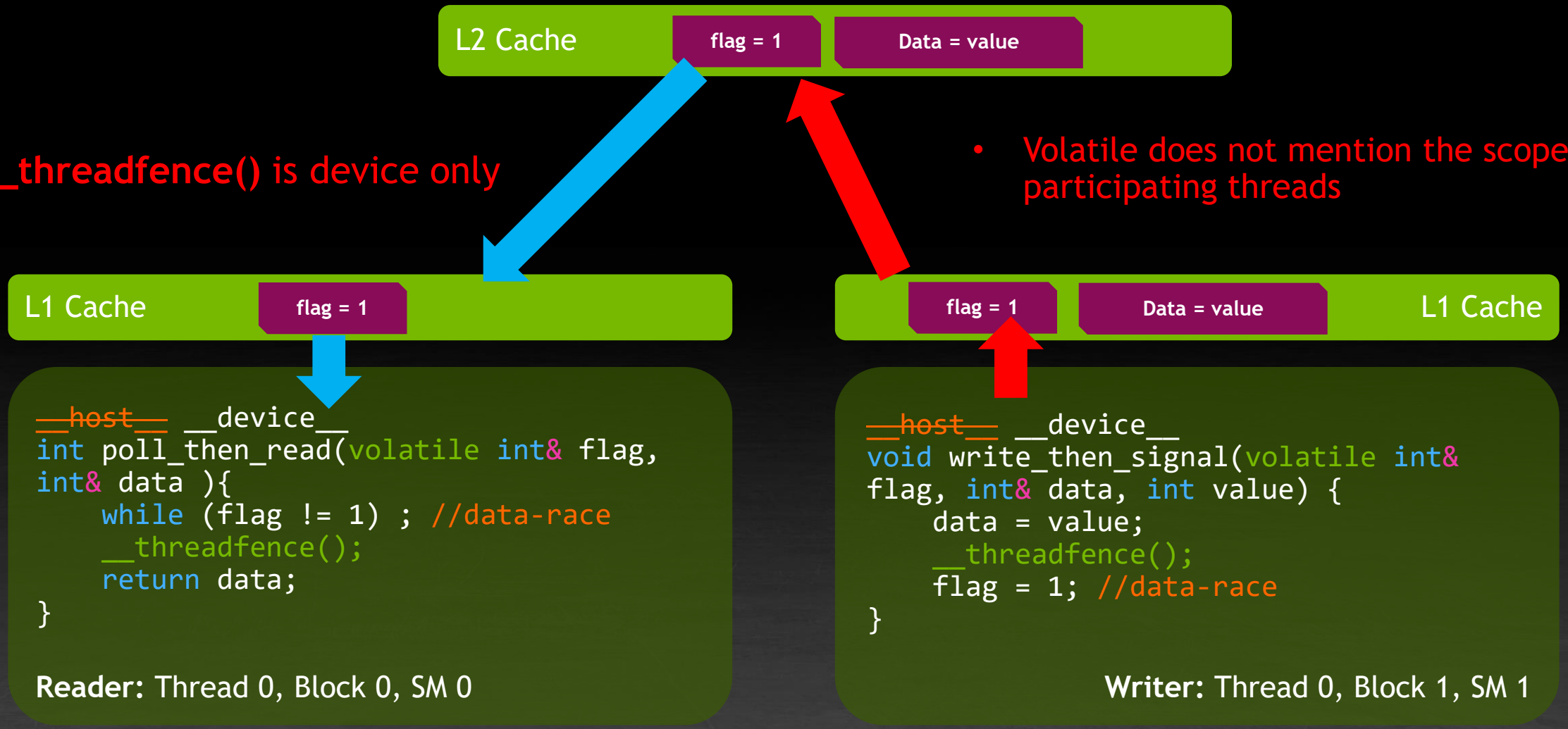
MEMORY MODEL

Synchronizing memory between multiple Thread Blocks



MEMORY MODEL

Synchronizing memory between multiple Thread Blocks



- **__threadfence()** is device only

- Volatile does not mention the scope of participating threads

```
__host__ __device__
int poll_then_read(volatile int& flag,
int& data ){
    while (flag != 1) ; //data-race
    __threadfence();
    return data;
}
```

Reader: Thread 0, Block 0, SM 0

```
__host__ __device__
void write_then_signal(volatile int&
flag, int& data, int value) {
    data = value;
    __threadfence();
    flag = 1; //data-race
}
```

Writer: Thread 0, Block 1, SM 1

Modern CUDA

MEMORY MODEL

Synchronizing memory between multiple Thread Blocks


```
__host__ __device__
int poll_then_read(atomic<bool>& flag,
int& data ) {
    while (!flag.load()) ;
    return data;
}
```

Reader: Thread 0, Block 0, SM 0

```
__host__ __device__
void write_then_signal(atomic<bool>&
flag, int& data, int value) {
    data = value;
    flag = true;
}
```

Writer: Thread 0, Block 1, SM 1

Modern CUDA

MEMORY MODEL

Synchronizing memory between multiple Thread Blocks


```
__host__ __device__
int poll_then_read(atomic<bool>& flag,
int& data ) {
    while (!flag.load()) ;
    return data;
}
```

Reader: Thread 0, Block 0, SM 0

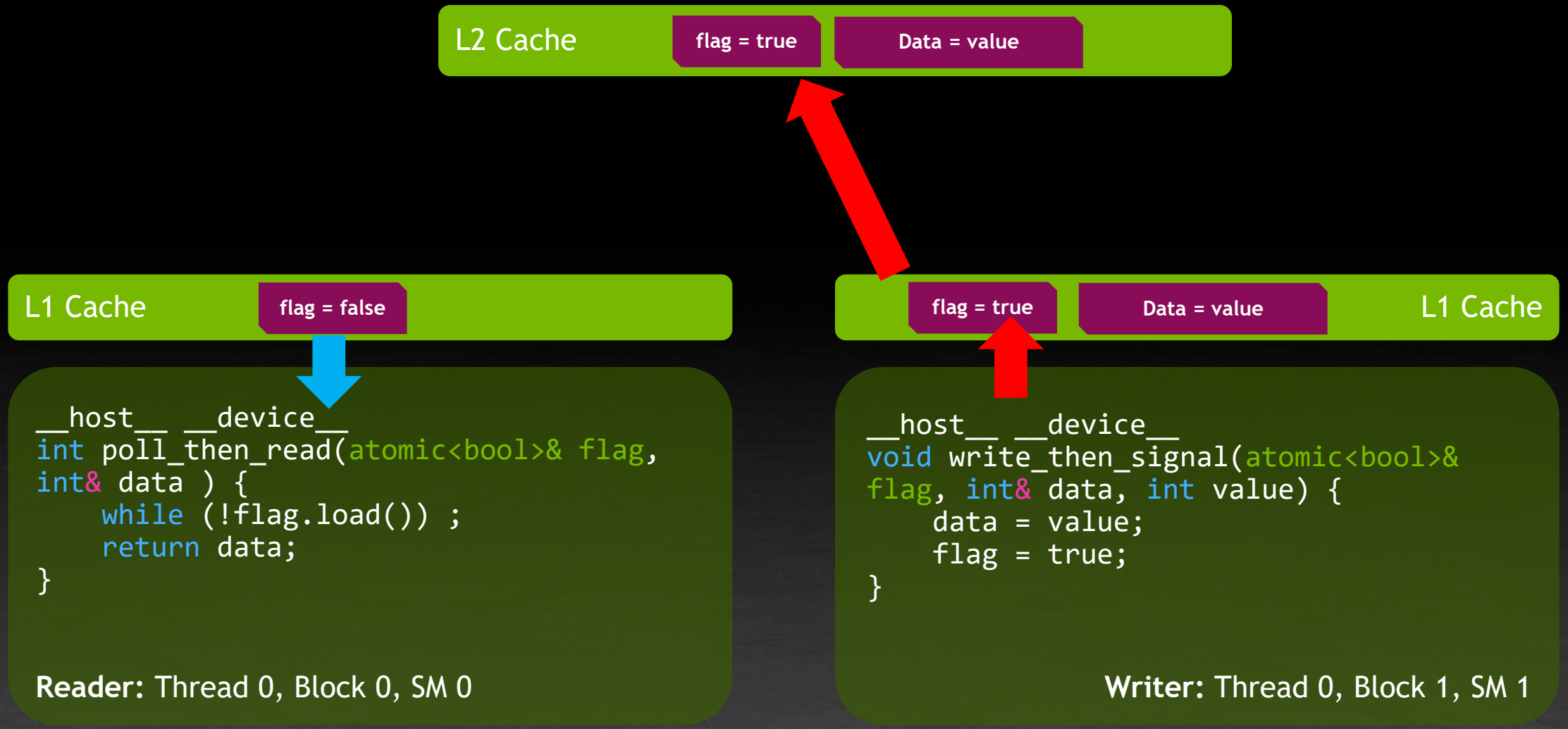
```
__host__ __device__
void write_then_signal(atomic<bool>&
flag, int& data, int value) {
    data = value;
    flag = true;
}
```

Writer: Thread 0, Block 1, SM 1

Modern CUDA

MEMORY MODEL

Synchronizing memory between multiple Thread Blocks



Modern CUDA

MEMORY MODEL

Synchronizing memory between multiple Thread Blocks


```
__host__ __device__
int poll_then_read(atomic<bool>& flag,
int& data ) {
    while (!flag.load()) ;
    return data;
}
```

Reader: Thread 0, Block 0, SM 0

```
__host__ __device__
void write_then_signal(atomic<bool>&
flag, int& data, int value) {
    data = value;
    flag = true;
}
```

Writer: Thread 0, Block 1, SM 1

MEMORY MODEL

Synchronizing memory between different Thread Blocks

Three Performance issues:

1. Uses system scope atomic. atomic<Type, cuda::thread_scope_system>
2. Full sequential consistency not needed
3. Spinning while loop, no backoff

```
__host__ __device__
int poll_then_read(atomic<bool>& flag,
int& data ) {
    while (!flag.load()) ;
    return data;
}
```

Reader: Thread 0, Block 0, SM 0

```
__host__ __device__
void write_then_signal(atomic<bool>&
flag, int& data, int value) {
    data = value;
    flag = true;
}
```

Writer: Thread 0, Block 1, SM 1

MEMORY MODEL

Synchronizing memory between different Thread Blocks

Three Performance issues:

1. ~~Uses system scope atomic~~ Use GPU Device scope
2. Full sequential consistency not needed
3. Spinning while loop, no backoff

```
__host__ __device__
int poll_then_read(
cuda::atomic<bool, cuda::thread_scope_device>&
flag, int& data ) {
    while (!flag.load());
    return data;
}
```

Reader: Thread 0, Block 0, SM 0

```
__host__ __device__
void write_then_signal (
cuda::atomic<bool, cuda::thread_scope_device>&
flag, int& data, int value) {
    data = value;
    flag = true;
}
```

Writer: Thread 0, Block 1, SM 1

MEMORY MODEL

Synchronizing memory between different Thread Blocks

Three Performance issues:

1. ~~Uses system scope atomic~~ Use GPU Device scope
2. ~~Full sequential consistency not needed~~ C++ acquire - release semantics are sufficient
3. Spinning while loop, no backoff

```
__host__ __device__
int poll_then_read(
cuda::atomic<bool, cuda::thread_scope_device>&
flag, int& data ) {
    while (!flag.load(memory_order_acquire));
    return data;
}
```

Reader: Thread 0, Block 0, SM 0

```
__host__ __device__
void write_then_signal (
cuda::atomic<bool, cuda::thread_scope_device>&
flag, int& data, int value) {
    data = value;
    flag.store(true, memory_order_release);
}
```

Writer: Thread 0, Block 1, SM 1

MEMORY MODEL

Synchronizing memory between different Thread Blocks

Three Performance issues:

1. ~~Uses system scope atomic~~ Use GPU Device scope
2. ~~Full sequential consistency not needed~~ C++ acquire - release semantics are sufficient
3. ~~Spinning while loop, no backoff~~ Use notify_all() wait() API with built in exponential backoff

```
__host__ __device__
int poll_then_read(
cuda::atomic<bool, cuda::thread_scope_device>&
flag, int& data ) {
    flag.wait(false, memory_order_acquire);
    return data;
}
```

Reader: Thread 0, Block 0, SM 0

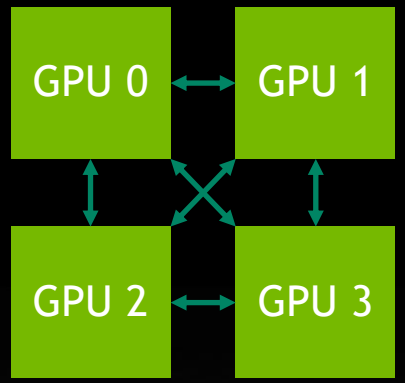
```
__host__ __device__
void write_then_signal (
cuda::atomic<bool, cuda::thread_scope_device>&
flag, int& data, int value) {
    data = value;
    flag.store(true, memory_order_release);
    flag.notify_all();
}
```

Writer: Thread 0, Block 1, SM 1

Modern CUDA

MEMORY MODEL

Synchronizing memory at system scope




```
__host__ __device__
int poll_then_read(
cuda::atomic<bool, cuda::thread_scope_system>&
flag, int& data ) {
    flag.wait(false, memory_order_acquire);
    return data;
}
```

Reader: Thread 0, Block 0, SM 0, GPU 0

```
__host__ __device__
void write_then_signal (
cuda::atomic<bool, cuda::thread_scope_system>&
flag, int& data, int value) {
    data = value;
    flag.store(true, memory_order_release);
    flag.notify_all();
}
```

Writer: Thread 0, Block 1, SM 1, GPU 1

L2 CACHE MANAGEMENT

ANNOTATED POINTER

A pointer annotated with an access property

```
template <typename T, typename AccessProperty>
class cuda::annotated_ptr;
```

```
int* g;
cuda::annotated_ptr<int, access_property::global> p{g};
p[threadIdx.x] = 42;
```

A pointer to **global** memory.
Like raw pointer with hint: property might be applied/ignored!

```
__device__ void independently_compiled(
    cuda::annotated_ptr<int, access_property>
);
```

Propagates properties through ABI boundaries for independently compiled device code.

ANNOTATED POINTER

Access Properties

cuda::annotated_ptr<T, cuda::access_property>

Static
Shared
Global
Dynamic

cuda::access_property::shared	Memory access to the **shared-memory**
cuda::access_property::global	Memory access to the **global-memory** (does not indicate frequency of access)
cuda::access_property::normal	Memory access to the **global-memory** as frequent as others
cuda::access_property::persisting	Memory access to the **global-memory** more frequent than others
cuda::access_property::streaming	Memory access to the **global-memory** less frequent than others
cuda::access_property • Interleaved • Range	Memory access to the **global-memory** with dynamic hint • Request properties for **probabilities** of memory addresses • Request properties for elements of an address **range**

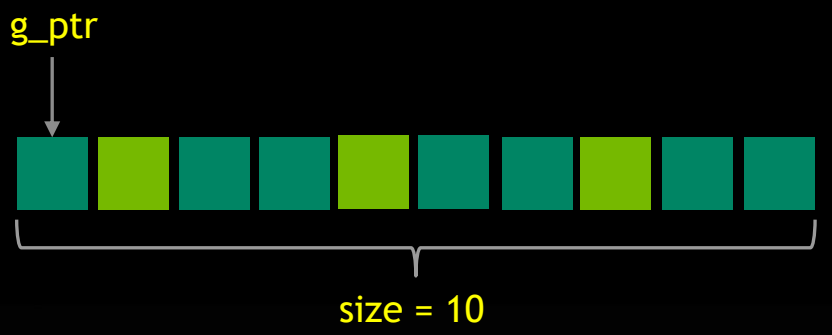
- sizeof(cuda::annotated_ptr<T, StaticAccessProperty>) == sizeof(T*)
- sizeof(cuda::annotated_ptr<T, cuda::access_property>) == 2*sizeof(T*)

Modern CUDA

ANNOTATED POINTER

Interleaved dynamic property

```
int* g_ptr; size_t sz;
cuda::access_property interleaved{
    cuda::access_property::persisting{},
    0.3,
    cuda::access_property::streaming{}
};
```



30% of memory addresses accessed with **persisting** property.

70% of memory addresses accessed with **streaming** property.

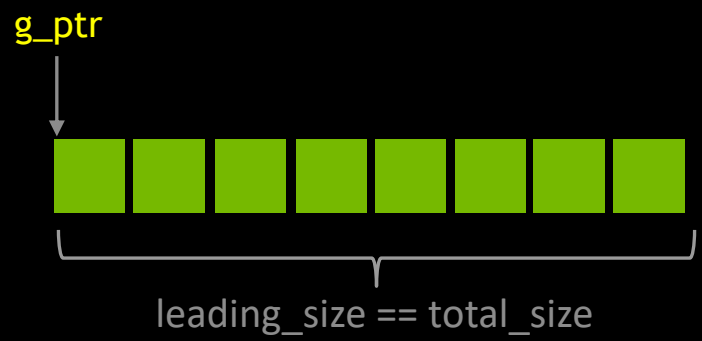
```
cuda::annotated_ptr<int, access_property> p{
  g_ptr, interleaved
};
p[threadIdx.x] = 42;
int v = p[threadIdx.x];
```

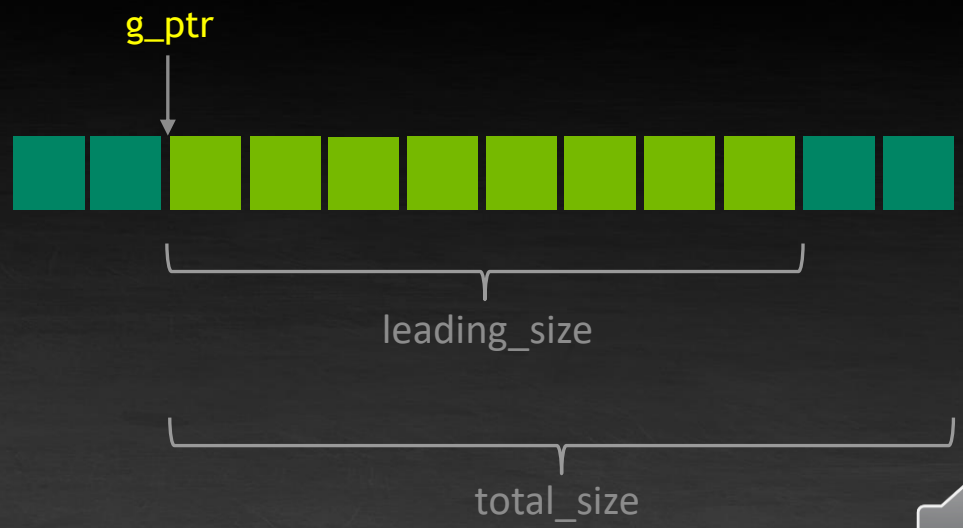
ANNOTATED POINTER

Range dynamic property

```
int* g_ptr; size_t leading_size, total_size;

cuda::access_property range{
    g_ptr, leading_size, total_size,
    cuda::access_property::persistent{},
    cuda::access_property::streaming{}
};
cuda::annotated_ptr<int, access_property::global> p{
  g_ptr, range
};
```





Modern CUDA

ANNOTATED POINTER

Prefetching memory

- Prefetches memory at L2 cache line granularity and sets access frequency
- Can use it as a larger shared memory (backed by global memory) that can be shared across thread-blocks

```
__global__ void pin(int* a) {

    if(threadIdx.x  == 0 ) // Thread 0 prefetches one L2 cache line or 32 integer elements
        cuda::apply_access_property(a, 32*sizeof(int), cuda::access_property::persisting{});
}
```

Modern CUDA

ANNOTATED POINTER

Discarding or Resetting memory

- Once done using it, either set its access frequency back to normal
- Or discard if the application does not need the lines to be written back to main memory
- Otherwise, **the memory might be kept in the L2 for a very long time.**


```
cuda::apply_access_property(ptr, size,
  cuda::access_property::normal{} );
```

No write back. Saves Bandwidth !

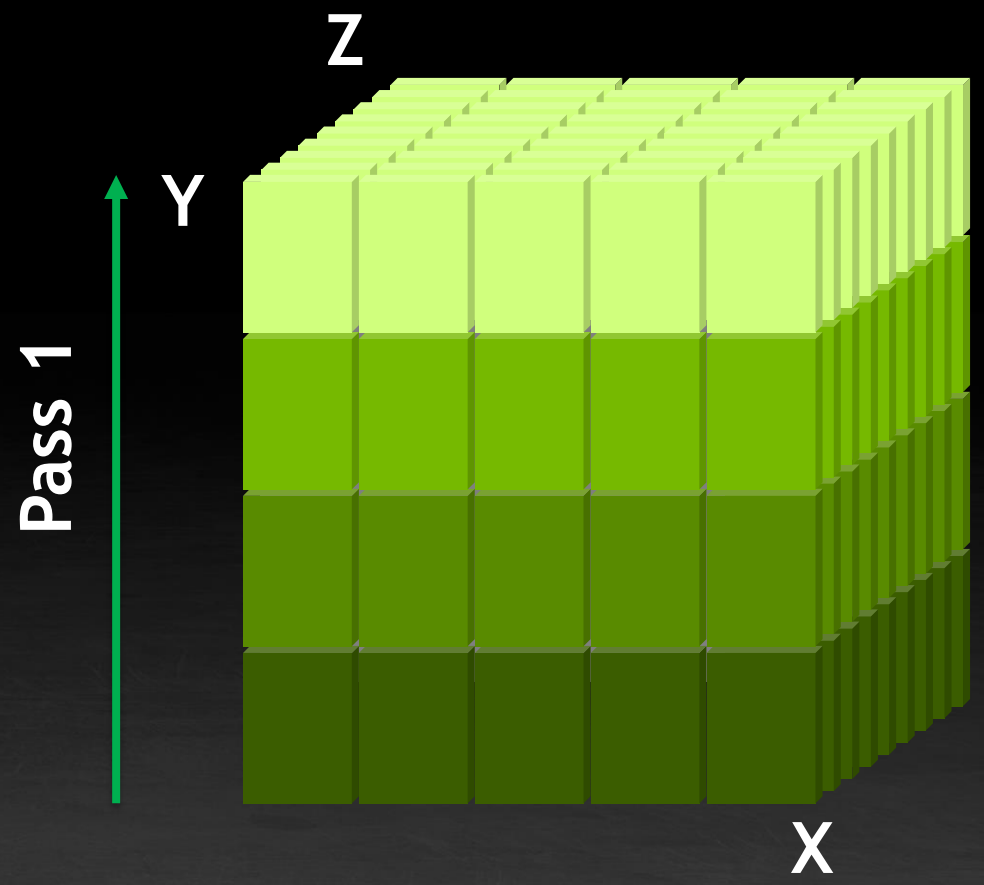

```
cuda::discard_memory(ptr, size);
```

Modern CUDA

ANNOTATED POINTER

Pass 1: Multi sweep on weather and climate stencils

```
for (int k =  0; k < nz; ++k) {
        pos += y_stride;
        float tmp_reg = dstm * src_p[pos];
        dst[pos] = tmp_reg + 1;
        tmp[pos] = tmp_reg - 1;
        dstm = tmp_reg;
  }
for (int k =  nz - 1; k >= 0; --k) {
        pos -= z_stride;
        dstm += (tmp[pos] - dst[pos] + 2.f);
        dst[pos] = dstm;
  }
```

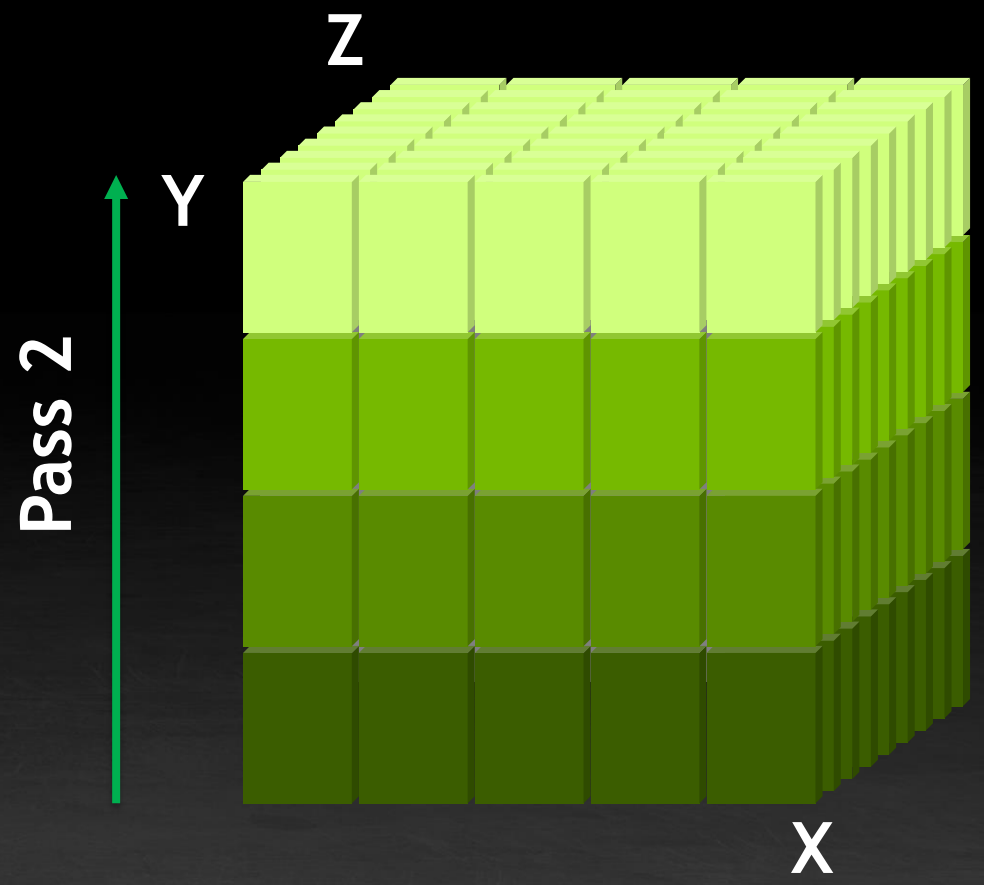


Modern CUDA

ANNOTATED POINTER

Pass 2: Multi sweep on weather and climate stencils

```
for (int k =  0; k < nz; ++k) {
        pos += y_stride;
        float tmp_reg = dstm * src_p[pos];
        dst[pos] = tmp_reg + 1;
        tmp[pos] = tmp_reg - 1;
        dstm = tmp_reg;
  }
for (int k =  nz - 1; k >= 0; --k) {
        pos -= z_stride;
        dstm += (tmp[pos] - dst[pos] + 2.f);
        dst[pos] = dstm;
  }
```



Modern CUDA

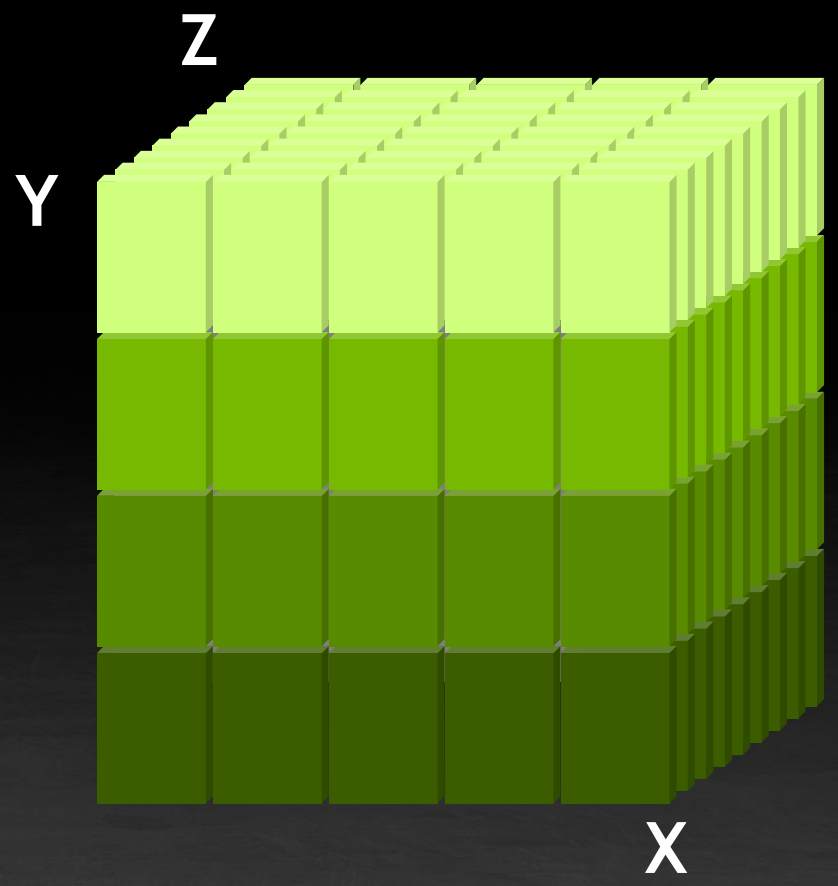
ANNOTATED POINTER

Multi sweep on weather and climate stencils

```
cuda::annotated_ptr<float,cuda::access_property
::persisting> dst_p{dst};

cuda::annotated_ptr<float,cuda::access_property
::persisting> tmp_p{tmp};

cuda::annotated_ptr<float,cuda::access_property
::streaming> src_p{src};
```



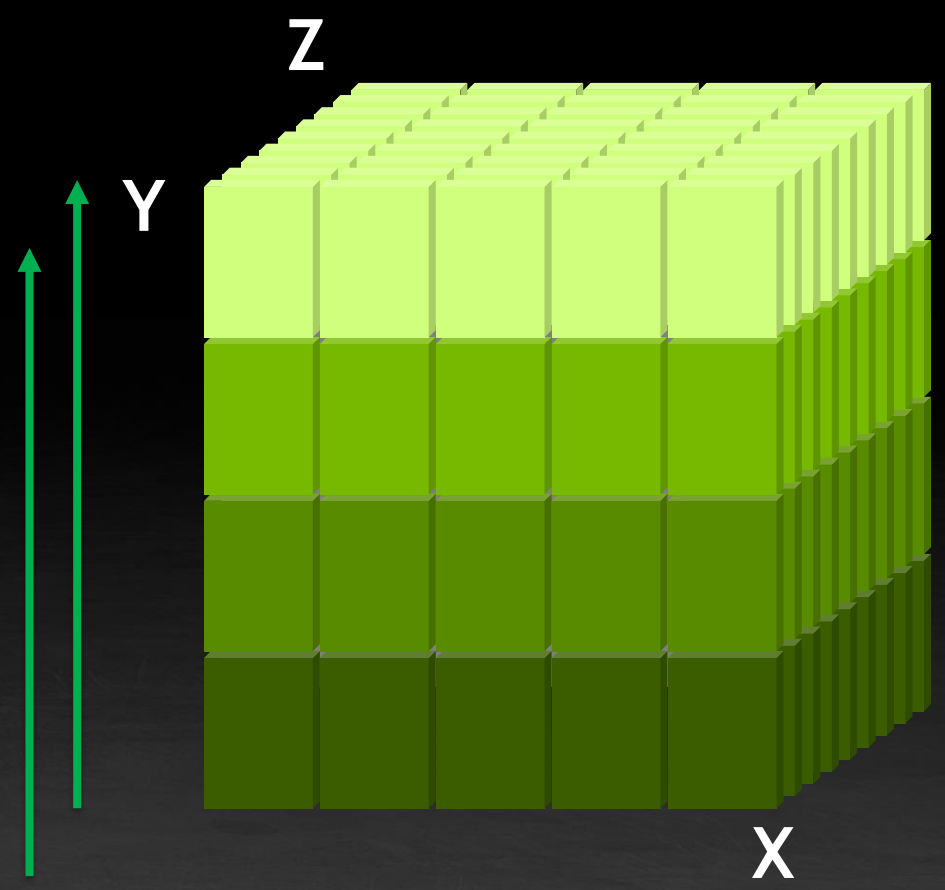
Modern CUDA

ANNOTATED POINTER

Pass 2: Multi sweep on weather and climate stencils

```
cuda::annotated_ptr<float,cuda::access_property::persisting> dst_p{dst};
cuda::annotated_ptr<float,cuda::access_property::persisting> tmp_p{tmp};
cuda::annotated_ptr<float,cuda::access_property::streaming> src_p{src};

for (int k =  0; k < nz; ++k) {
        pos += y_stride;
        float tmp_reg = dstm * src_p[pos];
        dst_p[pos] = tmp_reg + 1;
        tmp_p[pos] = tmp_reg - 1;
        dstm = tmp_reg;
  }
for (int k =  nz - 1; k >= 0; --k) {
        pos -= z_stride;
        dstm += (tmp_p[pos] - dst_p[pos] + 2.f);
        dst_p[pos] = dstm;
  }
```

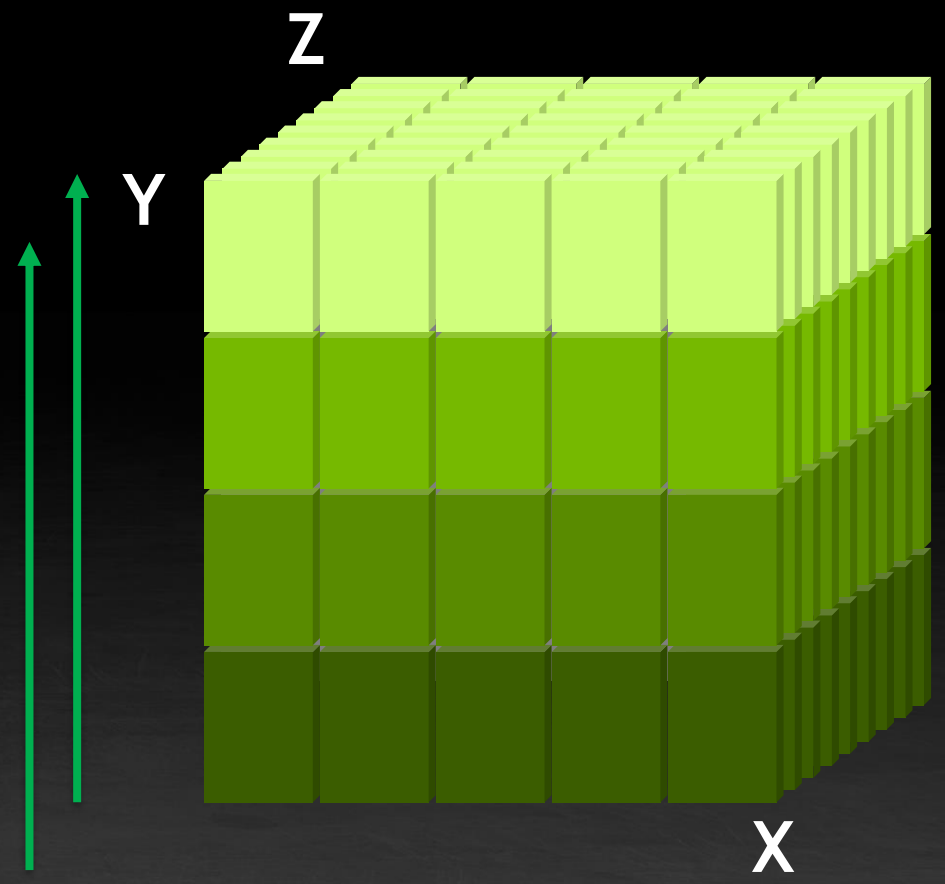


Modern CUDA

ANNOTATED POINTER

Pass 2: Multi sweep on weather and climate stencils

```
for (int k =  0; k < nz; ++k) {
        pos += y_stride;
        float tmp_reg = dstm * src_p[pos];
        dst_p[pos] = tmp_reg + 1;
        tmp_p[pos] = tmp_reg - 1;
        dstm = tmp_reg;
  }
for (int k =  nz - 1; k >= 0; --k) {
        pos -= z_stride;
        dstm += (tmp_p[pos] - dst_p[pos] + 2.f);
        dst_p[pos] = dstm;
        if( threadIdx.x == 0 )
        cuda::discard_memory(&tmp_p[pos],32*sizeof(float));
  }
```

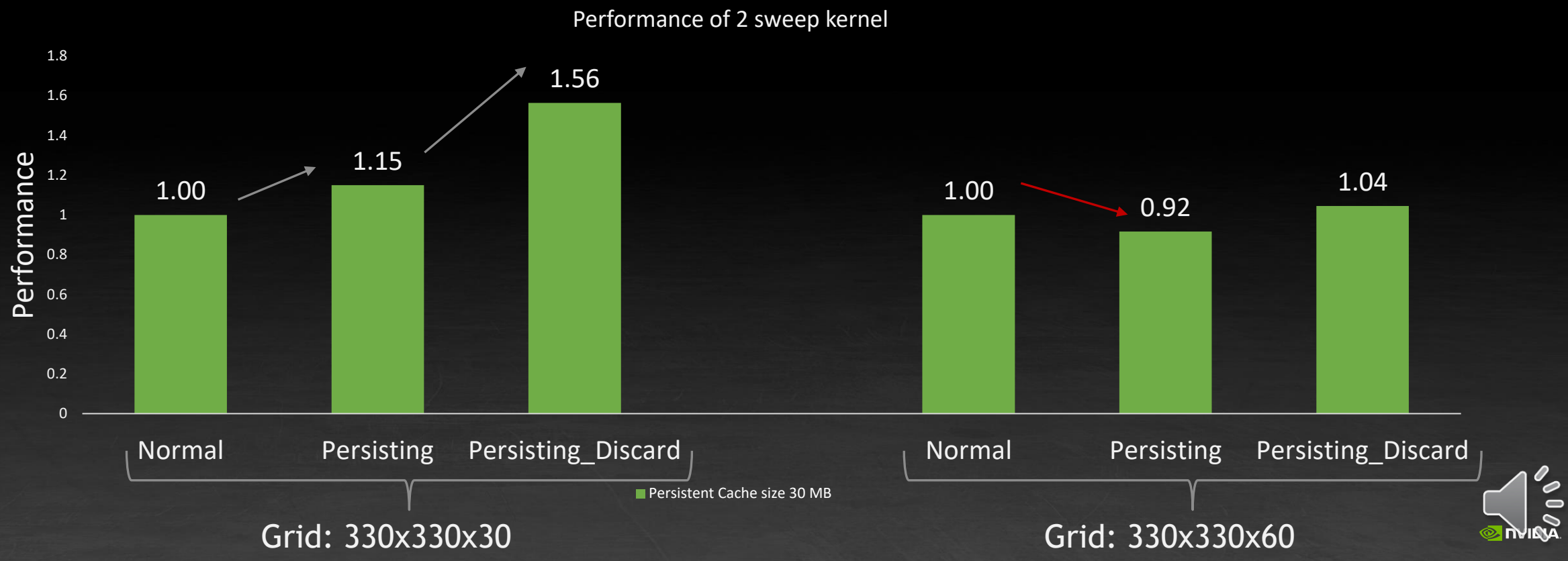


Modern CUDA

ANNOTATED POINTER

Performance Multi sweep on weather and climate stencils

```
auto l2_persistent_size = prop.persistingL2CacheMaxSize; // 30MB on A100. 75% of total
cudaDeviceSetLimit(cudaLimitPersistingL2CacheSize, l2_persistent_size);
```

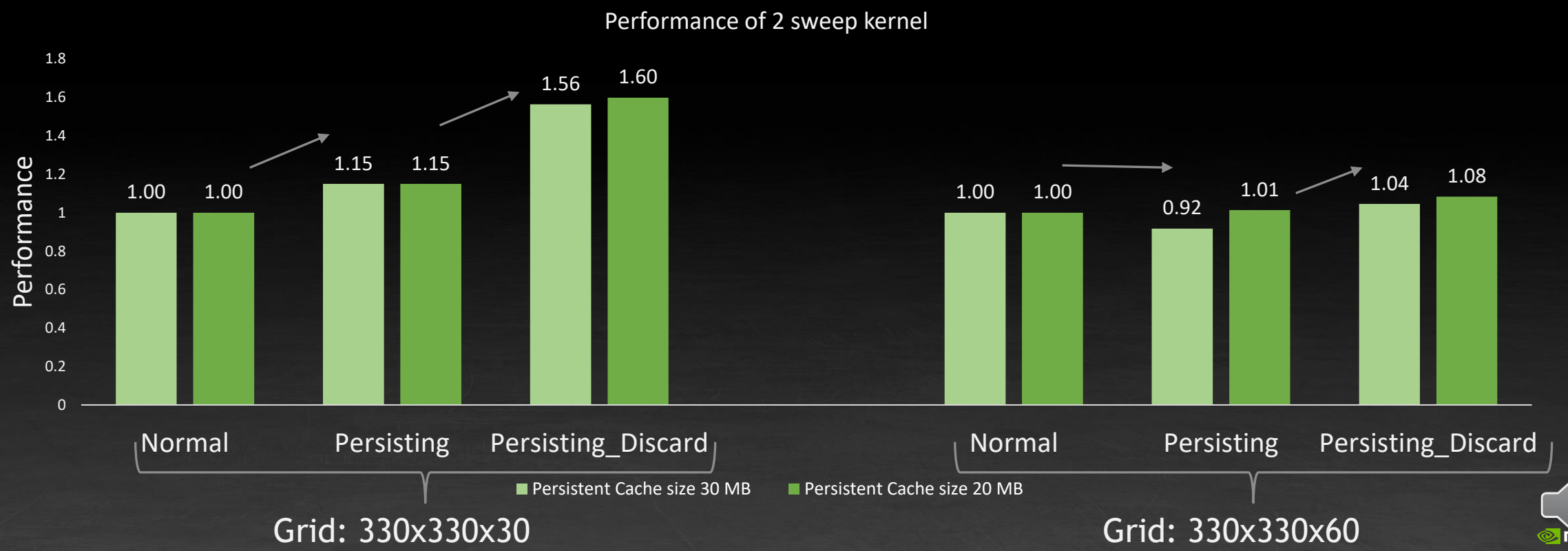


Modern CUDA

ANNOTATED POINTER

Performance Multi sweep on weather and climate stencils

```
auto l2_persistent_size = 20*1000*1000; // 20MB. 50% of total on A100
cudaDeviceSetLimit(cudaLimitPersistingL2CacheSize, l2_persistent_size);
```

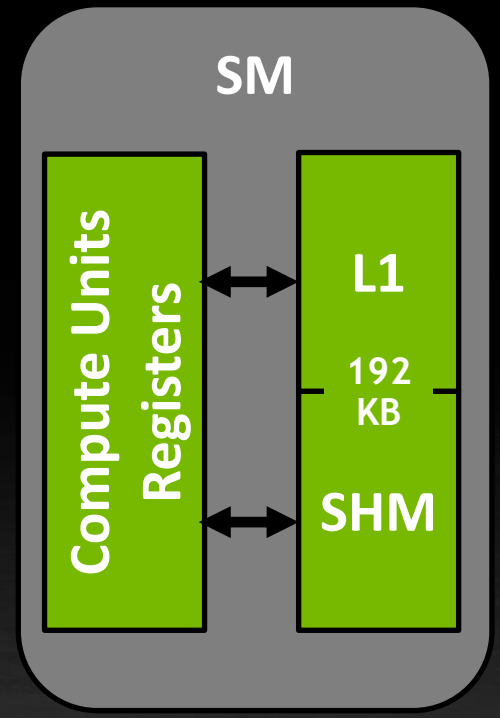


SHARED MEMORY

SHARED MEMORY

Fast, Threadblock local scratchpad

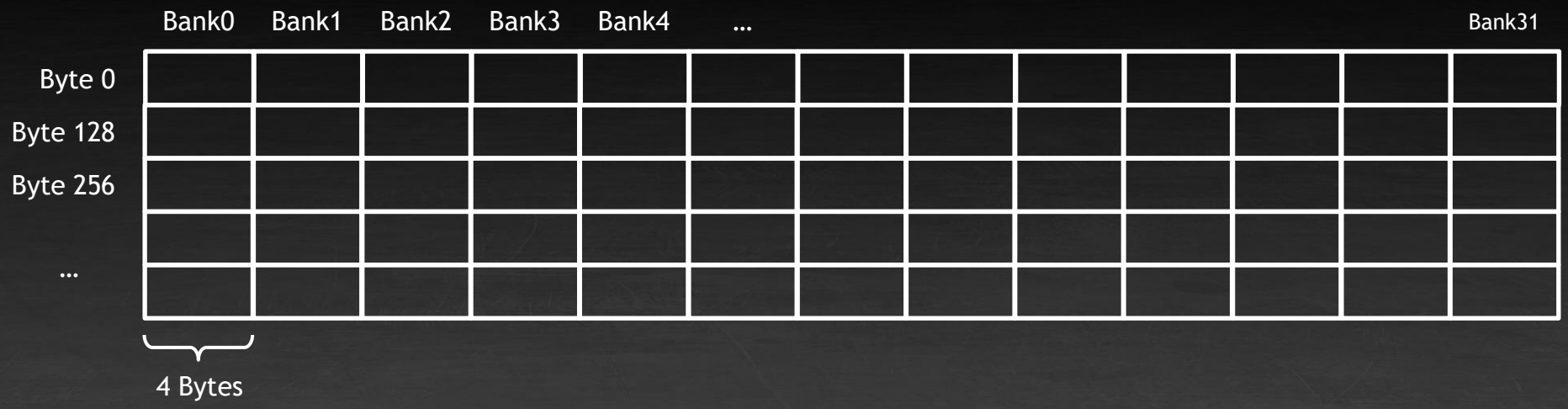
- Inside SM, private to thread block, user-controlled cache
- Default max = **48KB per thread block**, up to **164KB** (explicit opt-in) in A100
- **Max bandwidth A100** = 108 SMs x 1.41GHz x 128 Bytes/clock = **19.5 TB/s**
- **~10x more bandwidth** than Global Memory, very low latency
- Only **4-Byte** access **granularity**



SHARED MEMORY

Fast, SM-local scratchpad

Shared memory organization: 32 banks x 4 Bytes = 128 Bytes per row



SHARED MEMORY

Fast, SM-local scratchpad

Shared memory organization: 32 banks x 4 Bytes = 128 Bytes per row

Bank conflicts can happen, inside a warp:

- 2+ threads access different 4-Byte words from the same bank
- Worst case = 32-way conflicts, 31 replays
- Replays increase latency, decrease bandwidth

SHARED MEMORY

Example: 32x32 shared memory transpose

```
__shared__ float sm[32][32];
```

0	1	2	…	31
0	1	2	…	31
0	1	2	…	31
…	…	…	…	…
0	1	2	…	31

No conflicts when accessing rows

32-way conflicts accessing columns

Bank numbers

```
__shared__ float sm[32][33];
```

padding

0	1	2	…	31	0
1	2	3	…	0	1
2	3	4	…	1	2
…	…	…	…	…	…
31	0	1	…	30	31

No conflicts when accessing rows

No conflicts when accessing columns

SHARED MEMORY

Bank conflicts guidelines

Accessing elements	Banks per element	Bank Conflict Resolution level	
1,2,4 bytes	1	Warp	Up to 32-way conflicts!
8 bytes	2	½ warp	Up to 16-way conflicts
16 bytes	4	¼ warps	Up to 8-way conflicts

Accessing same word, or bytes inside same word = no conflict (multicast)

Easy to avoid conflicts by adding padding or changing access patterns

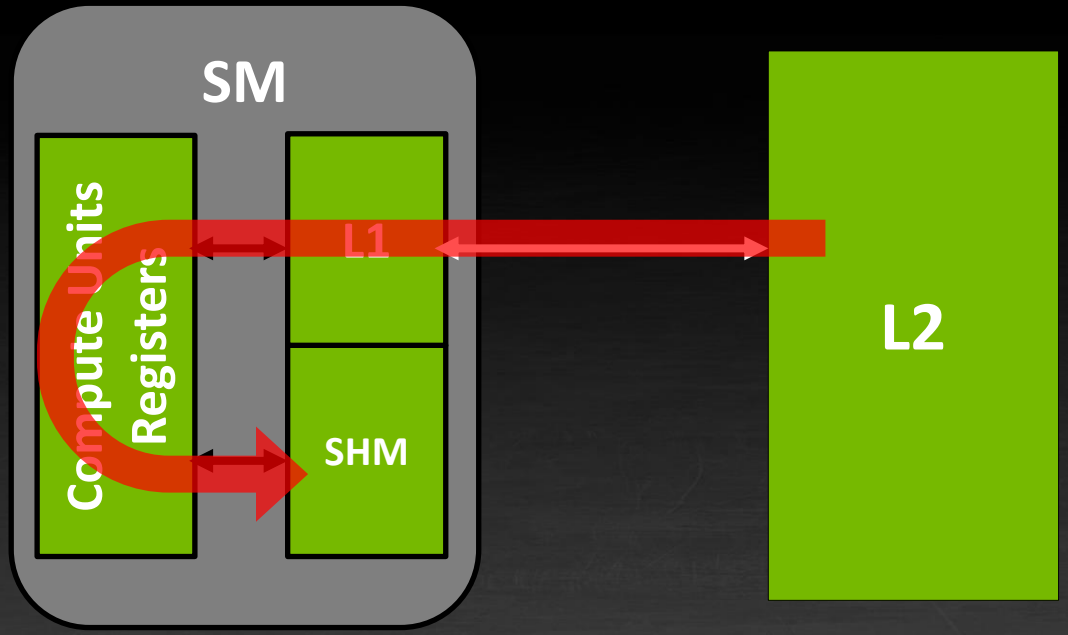
SHARED MEMORY

Async load to shared memory

Typical way of loading data to shared memory:

```
__shared__ int smem[1024];
smem[threadIdx.x] = input[index];
```

```
LDG.E.SYS R0, [R2] ;
* STALL *
STS [R5], R0 ;
```



- Wasting registers
- Stalling while the data is loaded
- Wasting L1/SHM bandwidth

Modern CUDA

SHARED MEMORY

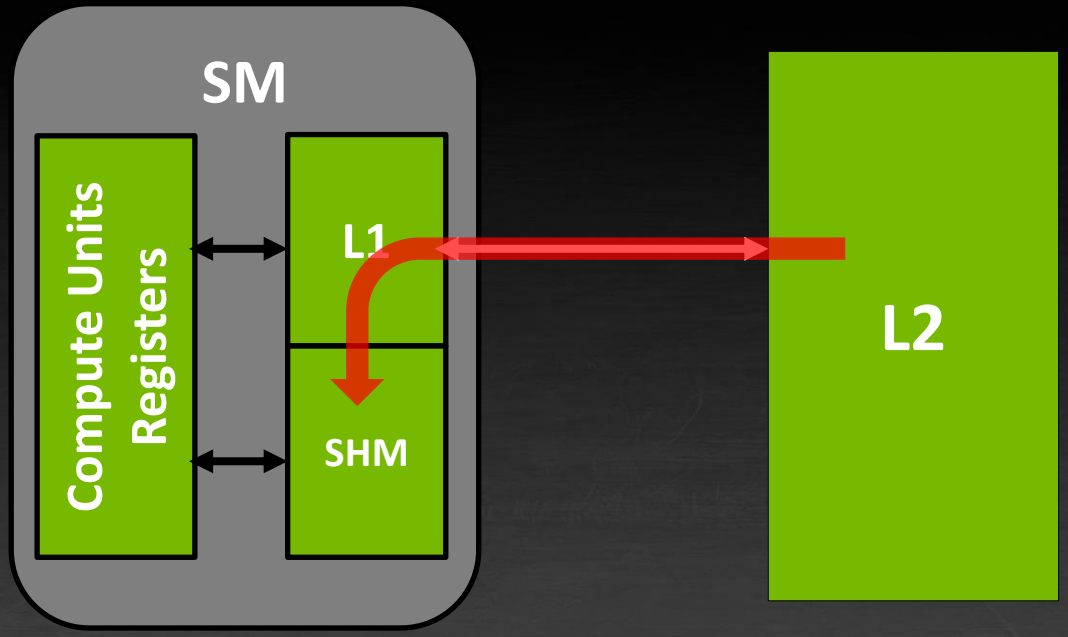
Async load to shared memory

The cooperative_groups API:

```
cooperative_groups::memcpy_async(Group, dst*, src*, Shape);
```

The CUDA APIs with synchronization primitives:

```
cuda::memcpy_async(Group, dst*, src*, Shape, cuda::barrier);
cuda::memcpy_async(Group, dst*, src*, Shape, cuda::pipeline);
```



Copy the data to shared memory asynchronously

Modern CUDA

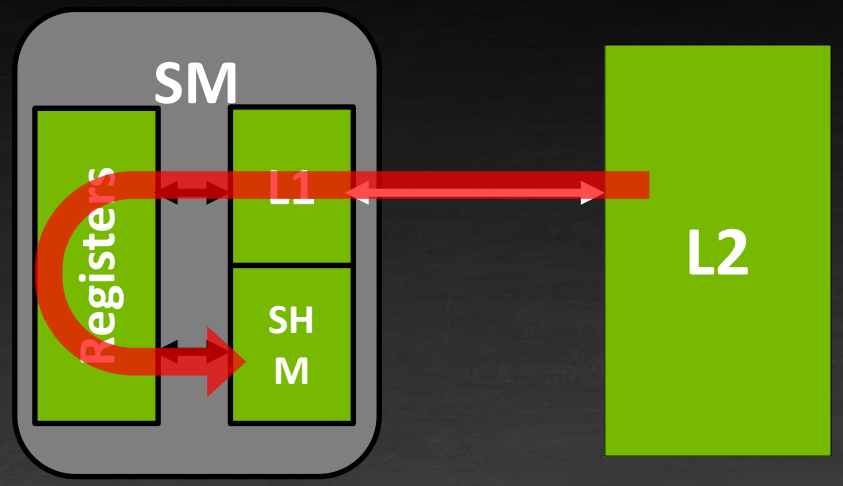
SHARED MEMORY

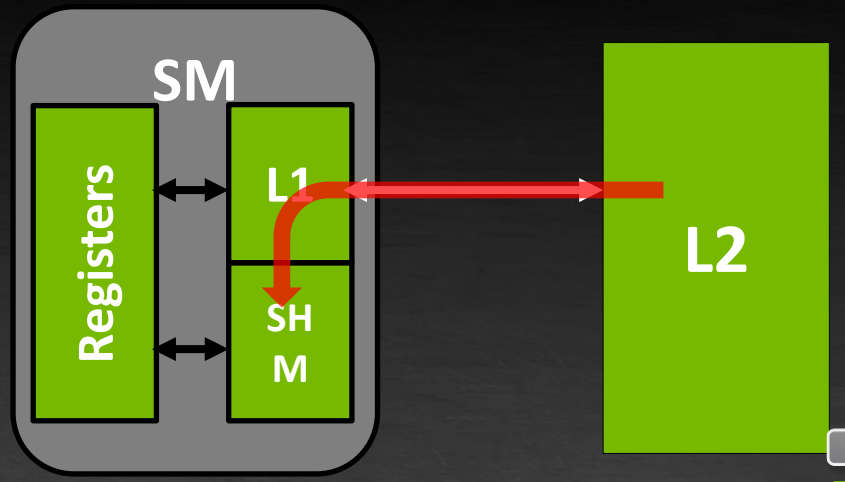
Simple replacement for filling in shared memory

```
extern __shared__ float shmem[];
for(int i=threadIdx.x; i<size; i+=blockDim.x)
{
    shmem[i] = gmem[i];
}
__syncthreads();
```

```
extern __shared__ float shmem[];
namespace cg = cooperative_groups;
auto block = cg::this_thread_block();

cg::memcpy_async(block, shmem, gmem, size*sizeof(float));
cg::wait(block);
```





MEMCPY_ASYNC()

Comparing variants

Code flow

Cooperative Groups	cuda::barrier	cuda::pipeline	What happens
		Pipe.producer_acquire()	Wait for consumer to release oldest pipeline stage. circular queue
cg::memcpy_async(g,…)	cuda::memcpy_async(g,…,bar)	cuda::memcpy_async(g,…, pipe)	Issue Async copies on the *barrier* or *pipe->barrier[i]*
		Pipe.producer_commit()	Commit the issued memcpy_async to the barrier.
cg::wait() cg::wait_prior<N>()	bar.arrive()	Pipe.consumer_wait()	Thread arrival on the barrier
	bar.wait()		Wait on the barrier
		Pipe.consumer_release()	Release current pipeline stage (only pipeline)


THREAD DIVERGENCE

DIVERGENCE

Intra-warp vs extra-warp divergence

CUDA deals with divergence at the warp level

```
if (condition)
      A();
else
      B();
```

Divergence between warps:
All threads inside each warp take the same branch

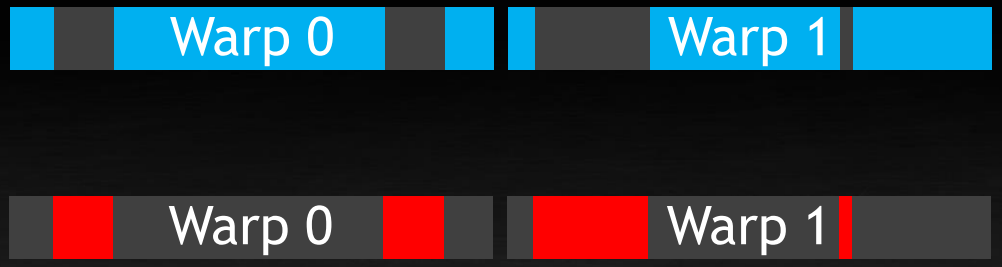
Divergence between warps is OK.

DIVERGENCE

Intra-warp vs extra-warp divergence

CUDA deals with divergence at the warp level

```
if (condition)
      A();
else
      B();
```



Divergence inside a warp :
Execute both branches,
masking out inactive threads

More instructions!
Avoid intra-warp divergence if you can

DIVERGENCE

CUDA Threads are Threads

Since Volta, NVIDIA GPUs guarantee forward progress

GPU must know which threads participate in warp-synchronous instructions

synchronization guaranteed only inside these instructions

The old warp-synchronous instructions without mask are deprecated!

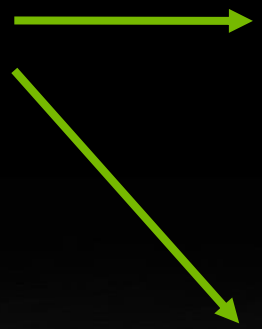
Update your code!

EXAMPLE

Update from shfl to shfl_sync

Simple case where all the threads are known to be active:

```
y = __shfl (x, 0);
```



```
y = __shfl_sync (0xffffffff, x, 0);
```

Or, use cooperative groups

```
namespace cg=cooperative_groups;
auto block = cg::this_thread_block();
auto tile32 =  cg::tiled_partition<32>(block);
y = tile32.shfl(x, 0);
```

EXAMPLE

Update from shfl to shfl_sync

More complex case where all threads might not participate:

```
for (int i=tid; i<N; i+=256)
{
    <…>
    y = __shfl(x, 0);
    <…>
}
```

```
for (int i=0; i<N; i+= 256)
{
    mask = __ballot_sync (0xffffffff, tid+i<N);
    if (tid+i<N)
    {
        <…>
        y = __shfl_sync(mask, x, 0);
        <…>
    }
}
```

EXAMPLE

Update to Cooperative Groups

You can also use Cooperative groups

```
for (int i=0; i<N; i+= 256)
{
     mask = __ballot_sync (0xffffffff, tid+i<N);
     if (tid+i<N)
     {
          <…>
          y = __shfl_sync(mask, x, 0);
          <…>
     }
}
```

```
namespace cg=cooperative_groups;
auto block = cg::this_thread_block();
auto tile32 =  cg::tiled_partition<32>(block);

for (int i=0; i<N; i+= 256)
{
   auto subtile = cg::binary_partition(tile32, tid+i<N);
   if (tid+i<N)
   {
      <…>
      y = subtile.shfl(x, 0);
      <…>
    }
}
```

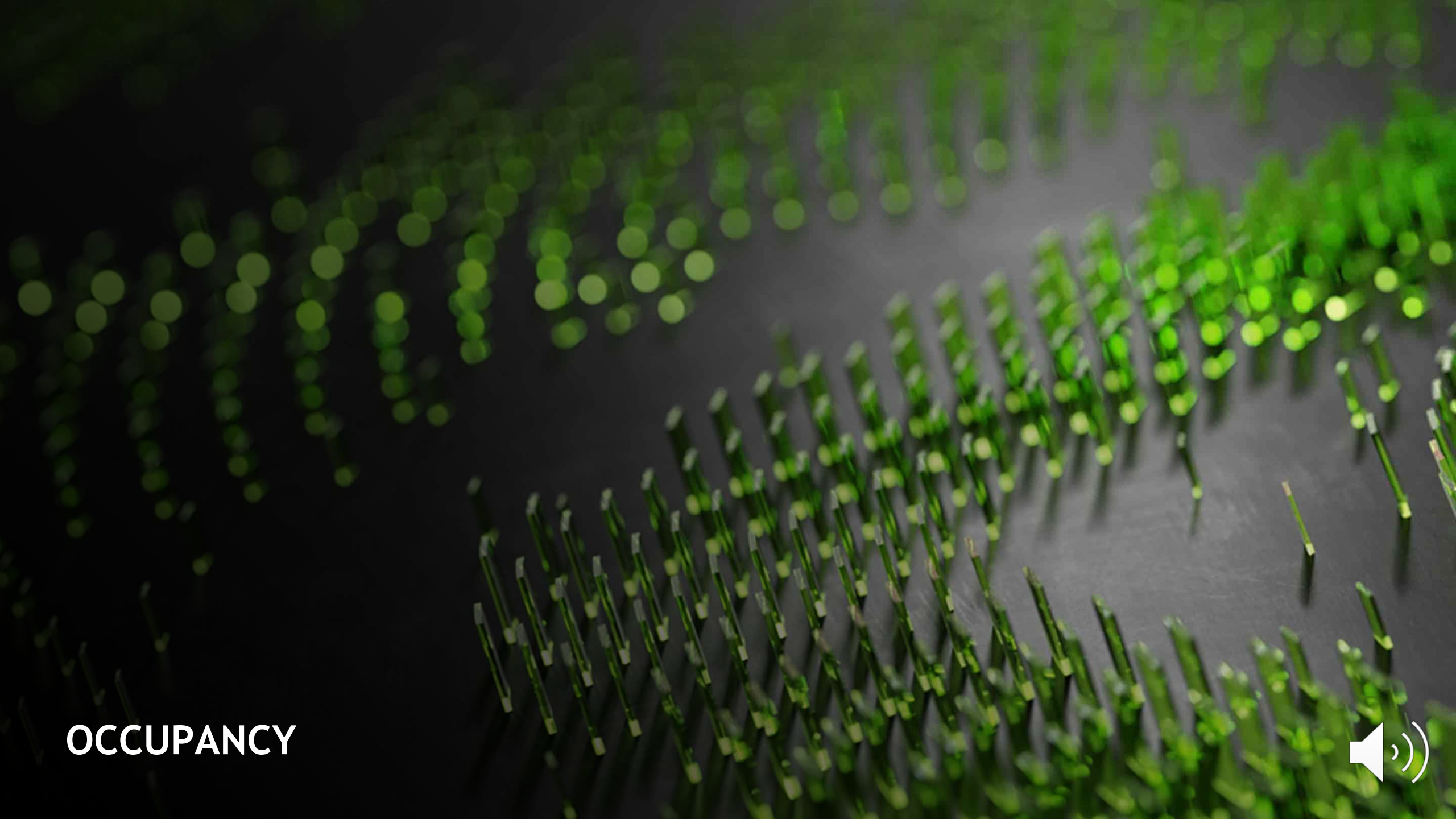
DIVERGENCE

No convergence assumptions

Expect ≠ Assume

- Expecting convergence is reasonable (performance)
- Assuming convergence is illegal!

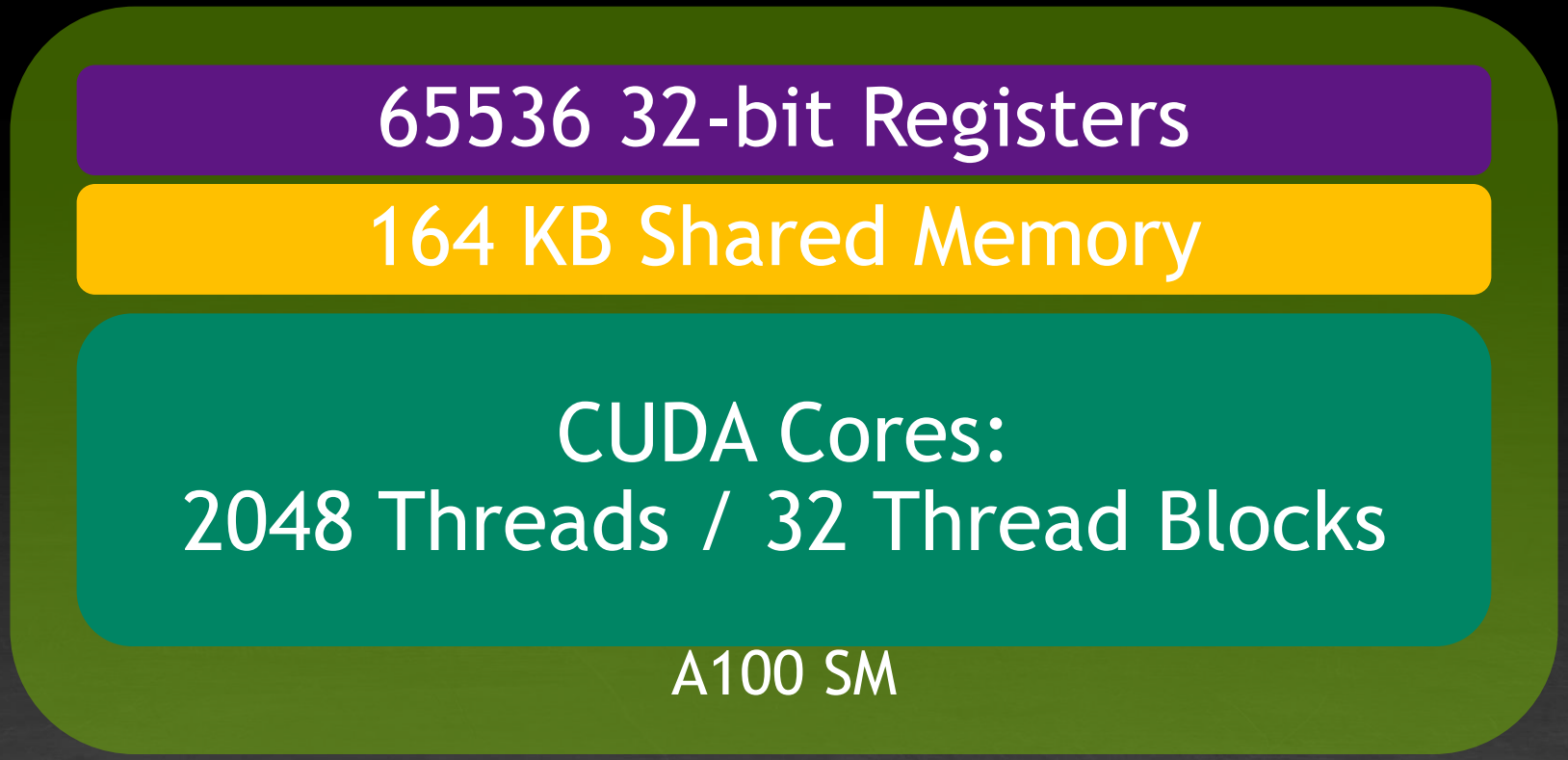

Keeping up with GPUs getting wider and bigger


OCCUPANCY

OCCUPANCY

SM Resources on A100

$$\text{Occupancy} = \frac{\text{Achieved number of threads per SM}}{\text{Maximum number of threads per SM}}$$

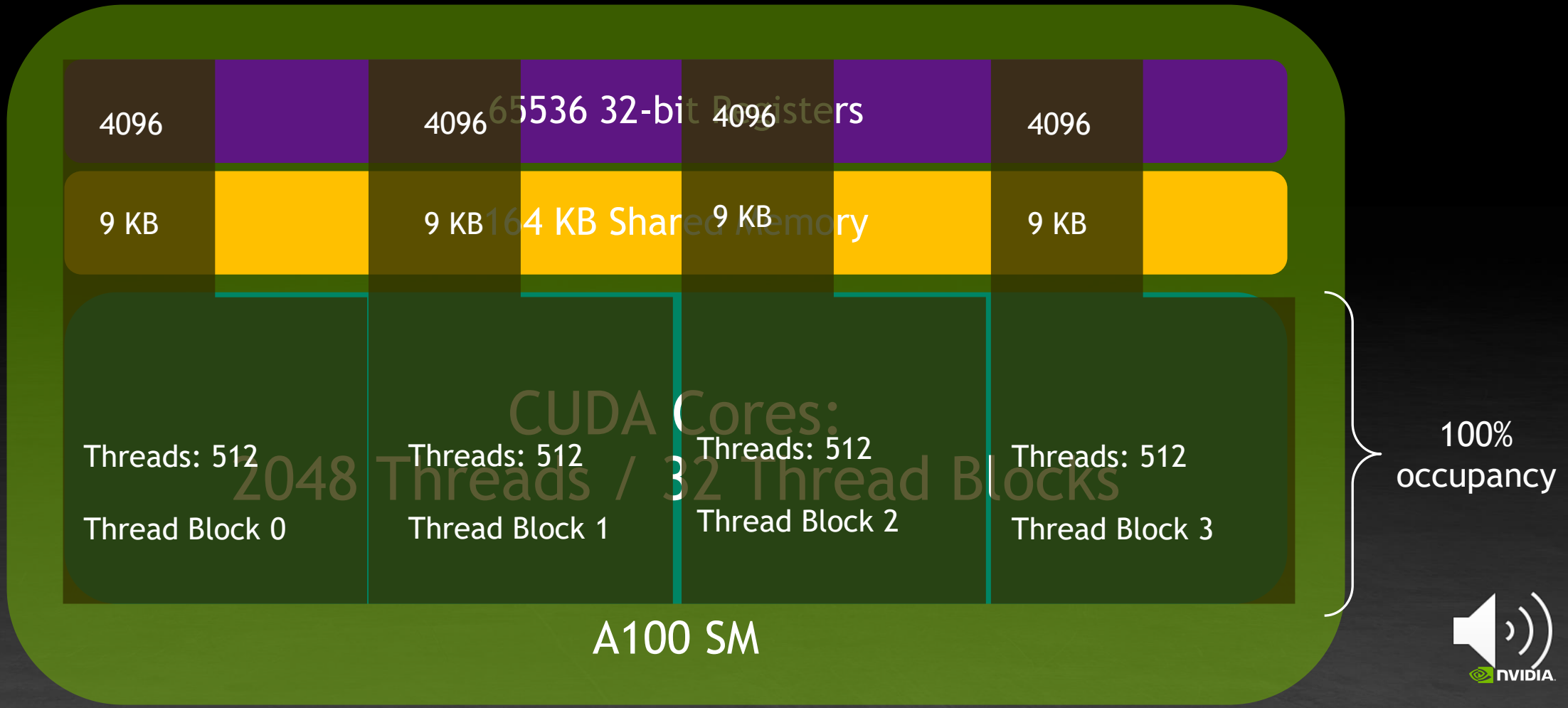


Occupancy = Achieved number of threads per SM / Maximum number of threads per SM

OCCUPANCY

SM Resources on A100

Example kernel: 512 threads per block, 8 registers / thread (4096 / block) , 8KB shared memory per block



OCCUPANCY

Higher occupancy

In general, higher occupancy is better:

- More warps per SM
- More parallelism to hide the latencies
- Can also do well with lower occupancy, but more work per thread.

Example kernel:

- 512 threads per block
- 8 registers per thread
- 8KB shared memory

} occupancy = 100%

OCCUPANCY

Higher occupancy

In general, higher occupancy is better:

- More warps per SM
- More parallelism to hide the latencies
- Can also do well with lower occupancy, but more work per thread.

Example kernel:

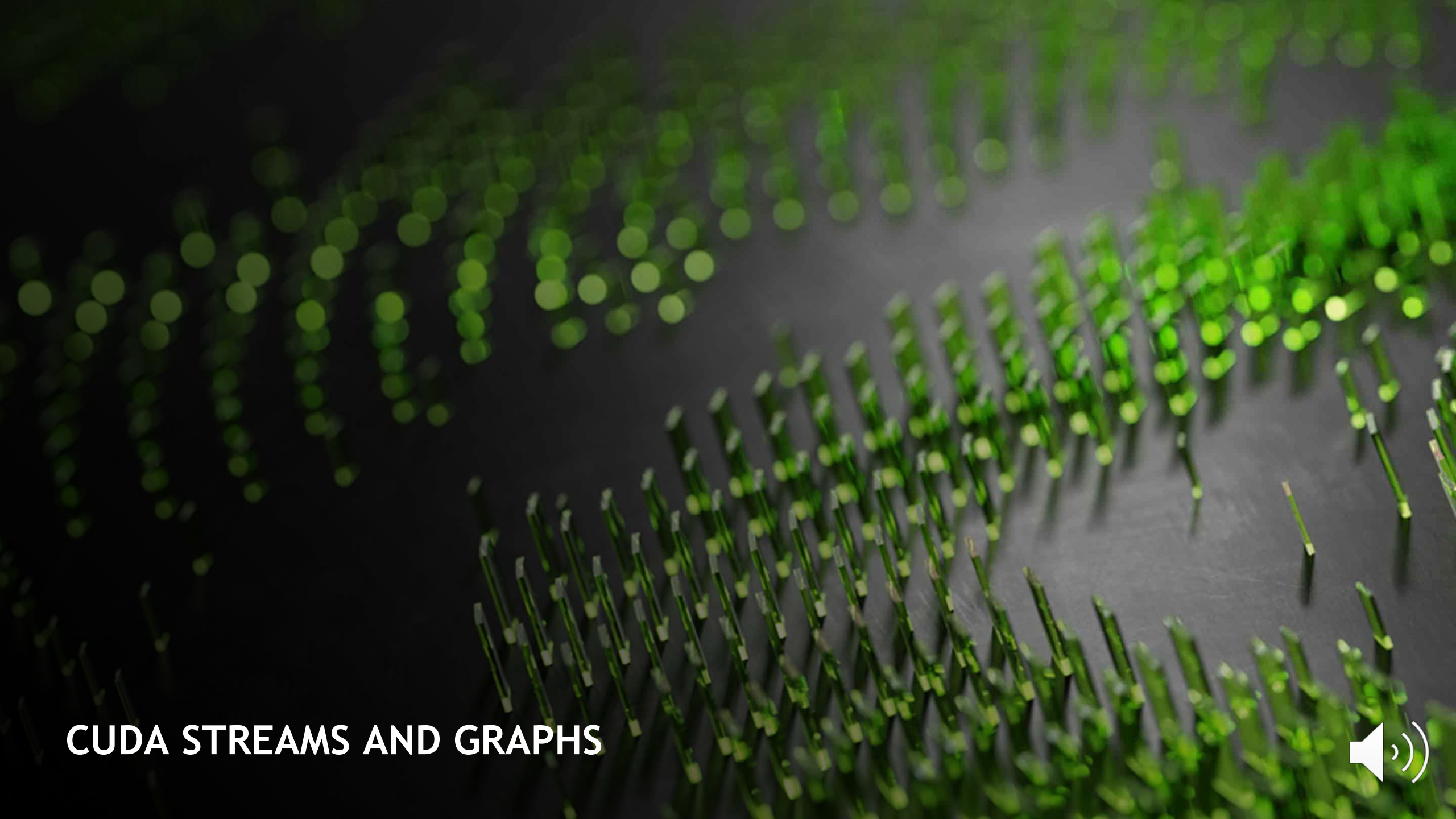
- 512 threads per block
- 8 registers per thread
- 8KB shared memory

} occupancy = 100%

But launching only 80 thread blocks?

$$\frac{80 \text{ blocks x } 512 \text{ threads}}{56 \text{ SMs x } 2048 \text{ threads}} = 36\% \text{ of P100}$$

$$\frac{80 \text{ blocks x } 512 \text{ threads}}{108 \text{ SMs x } 2048 \text{ threads}} = 18\% \text{ of A100}$$

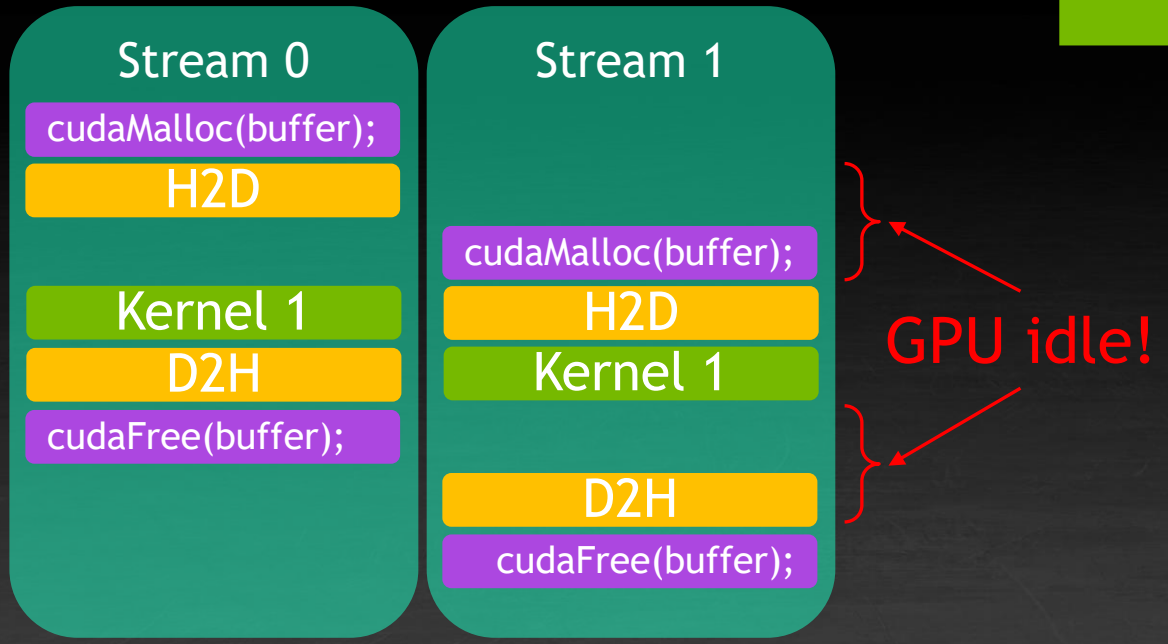

CUDA STREAMS AND GRAPHS

Modern CUDA

CUDA STREAMS

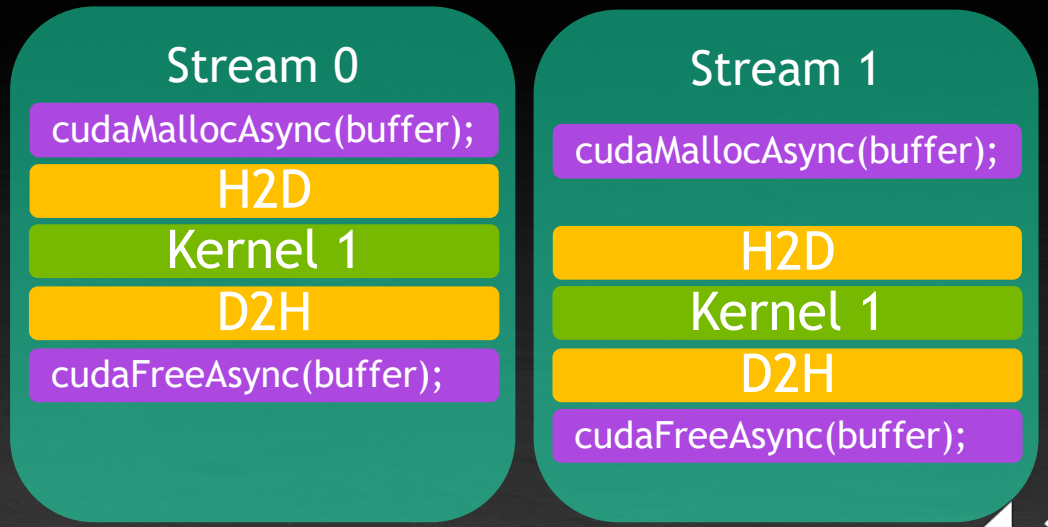
Overlapping compute, copies and allocations

cudaMalloc() or cudaFree() halts GPU



New stream-ordered asynchronous memory allocations

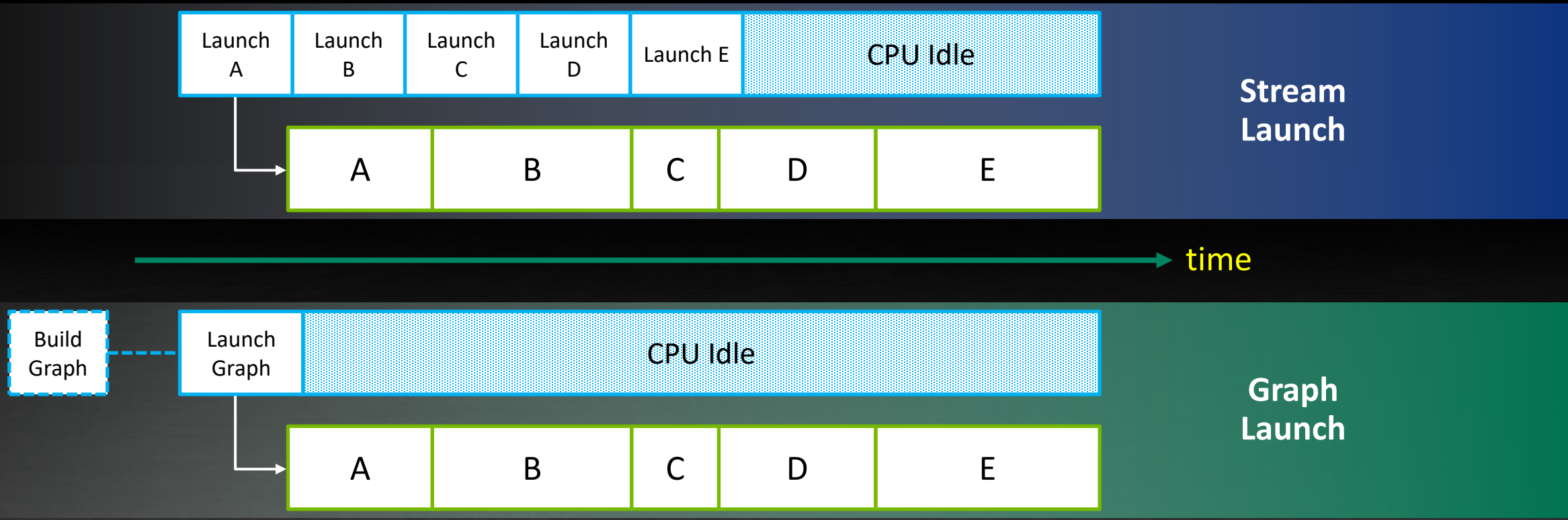
Very low latency (memory pool)



NVIDIA

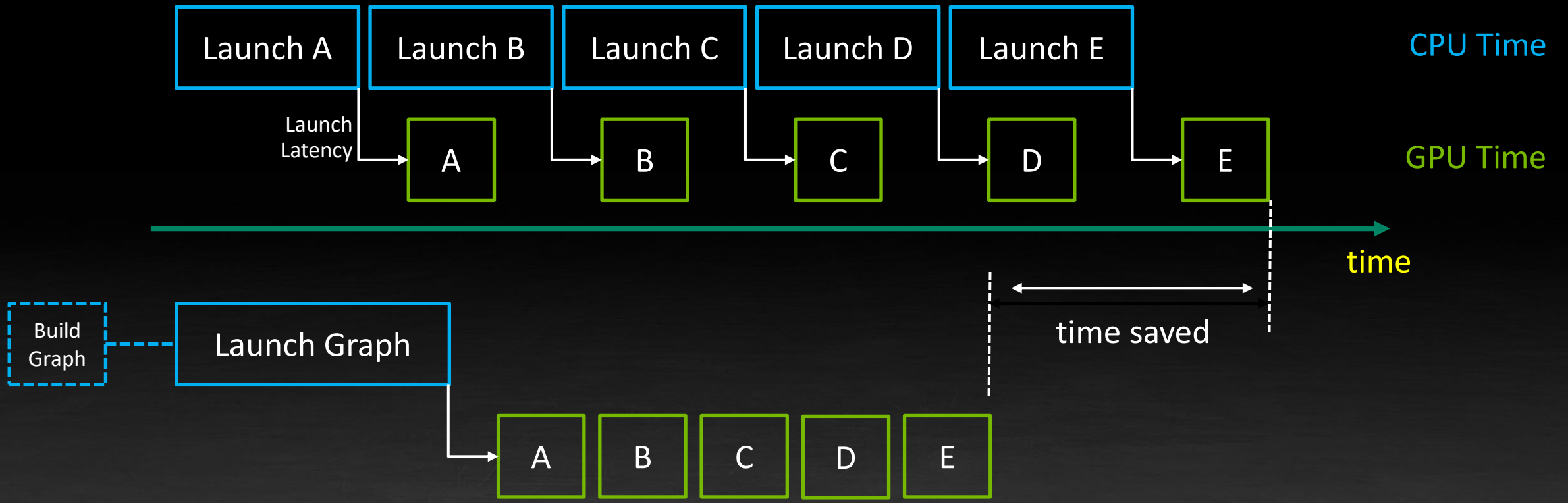
CUDA GRAPHS

Free up CPU resources



CUDA GRAPHS

Reduction in launch overhead



When kernel runtime is short, execution time is dominated by CPU launch cost

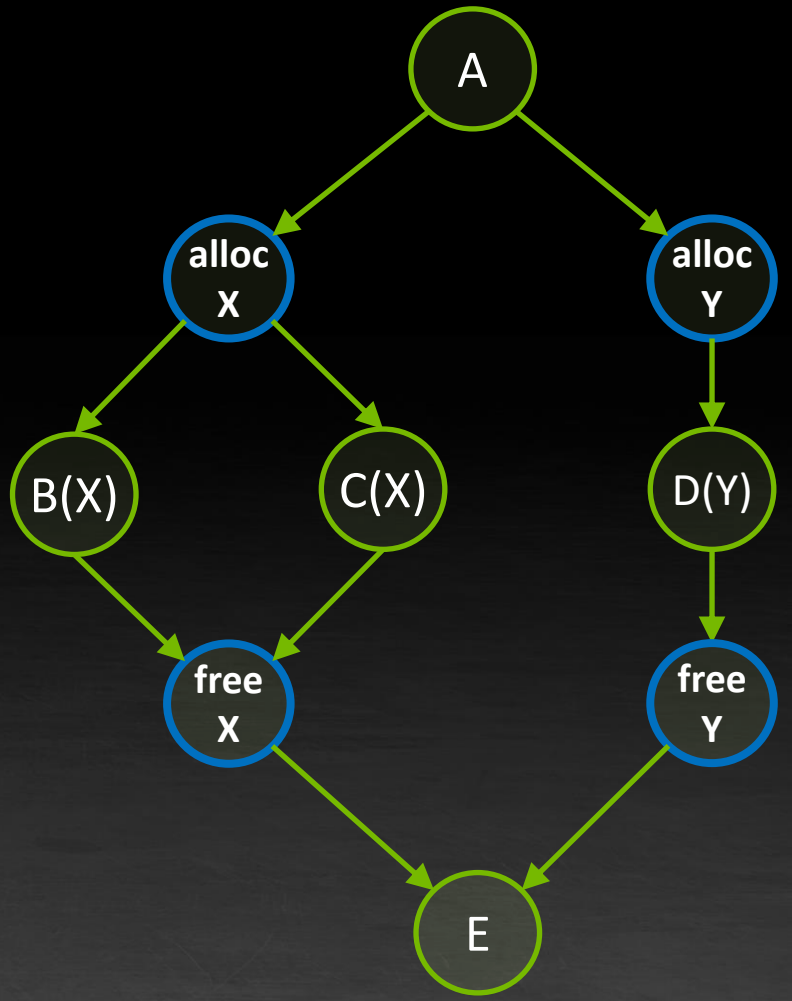
CUDA GRAPHS

Embedding memory allocation in graphs

CUDA Graphs offers two new node types:
allocation & **free**

Identical semantics to stream *cudaMallocAsync()*

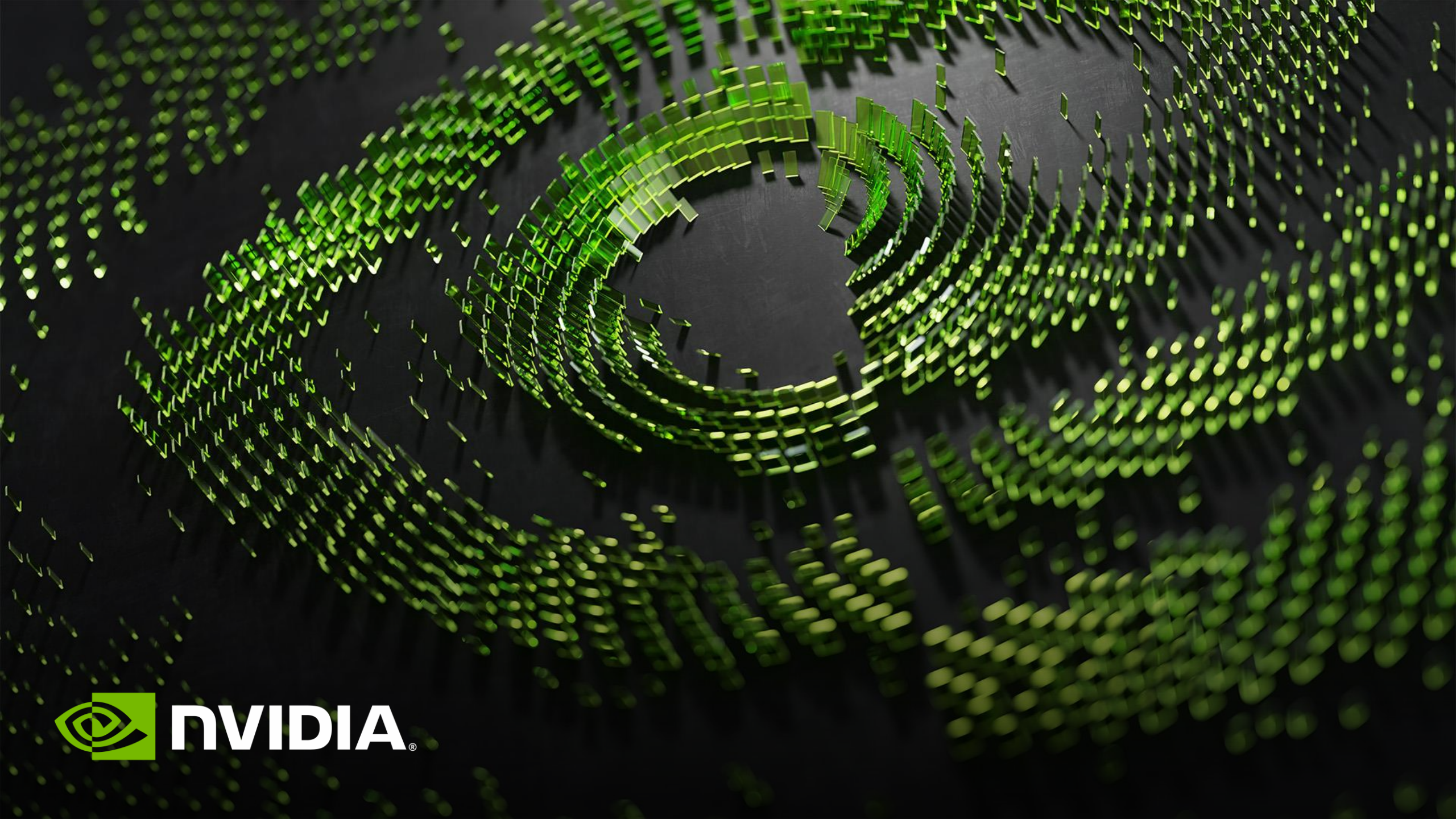
- Pointer is returned **at node creation time**
- Pointer may be passed as argument to later nodes
- Dereferencing pointer is only permitted downstream of allocation node & upstream of free node



TAKEAWAYS

How to keep up with larger GPUs

- Be aware of your grid sizes and achieved occupancy
- Re-visit code, express more parallelism
- Launch more independent kernels in parallel: CUDA streams, graphs
- Share the GPU with multiple processes: MPS
- Split the GPU into smaller instances: MIG


NVIDIA