PROGRAMMING THE NEXT GPU GENERATION

MARKUS HRYWNIAK, DEVTECH COMPUTE

TO THE NEXT 10 YEARS

Parallelism is not decreasing – quite the opposite

Core counts of CPUs and GPUs

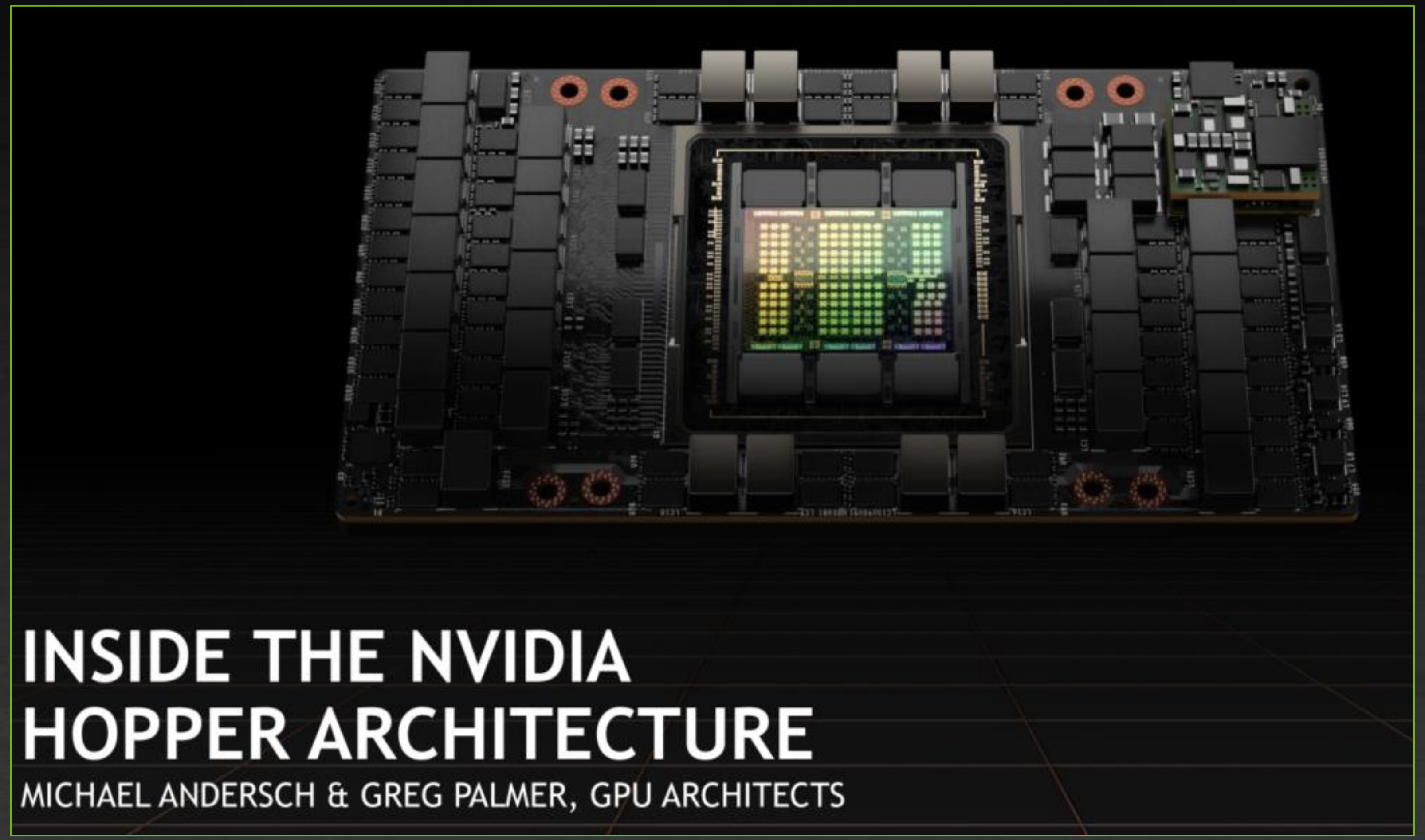
Programming models need to keep up with new Hardware

Data Locality and Asynchronicity

CUDA: NEW FEATURES AND BEYOND

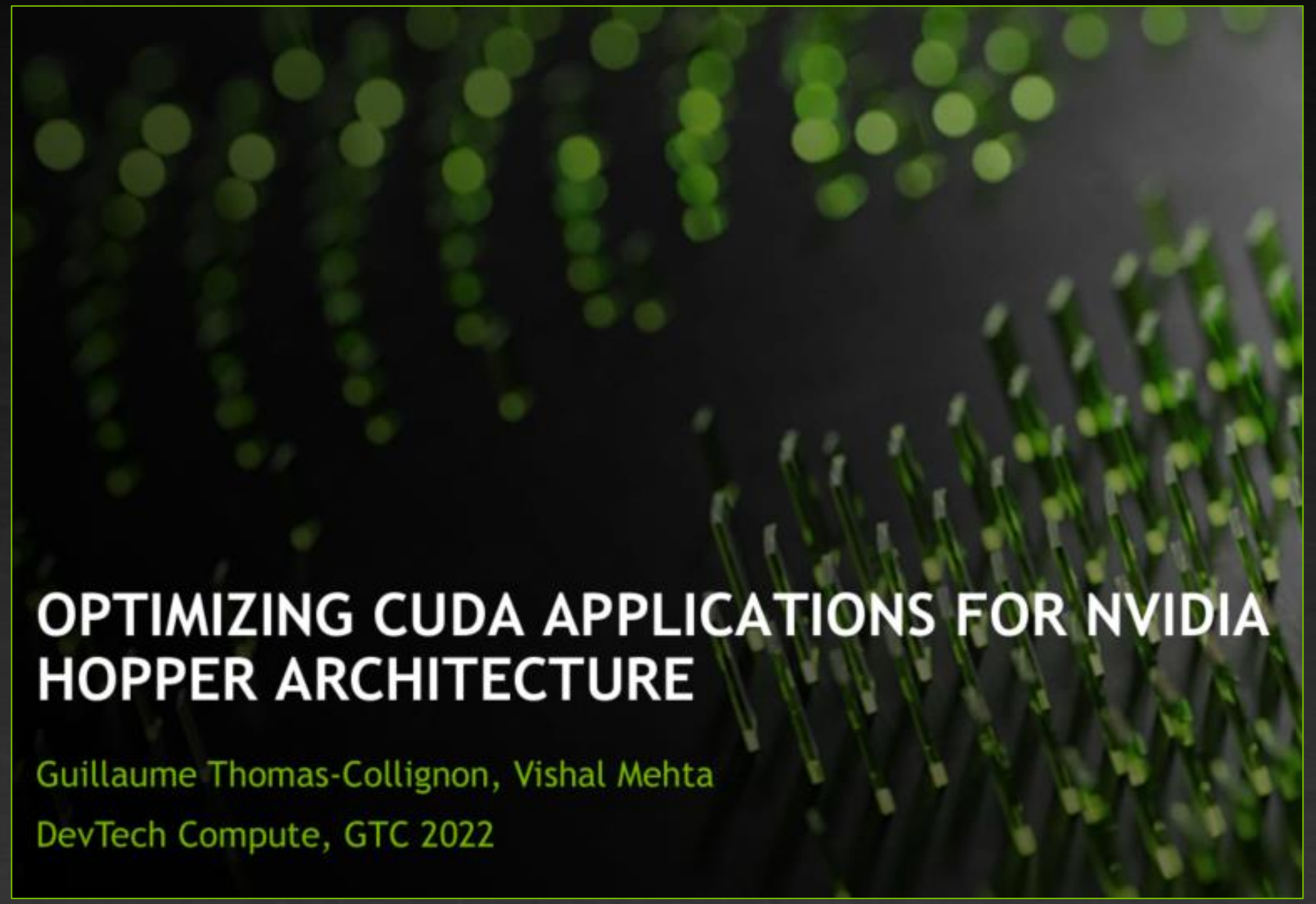
Stephen Jones, GTC 2022

CUDA: New Features and Beyond [S41486]


INSIDE THE NVIDIA HOPPER ARCHITECTURE

MICHAEL ANDERSCH & GREG PALMER, GPU ARCHITECTS

Inside the NVIDIA Hopper Architecture [S42663]

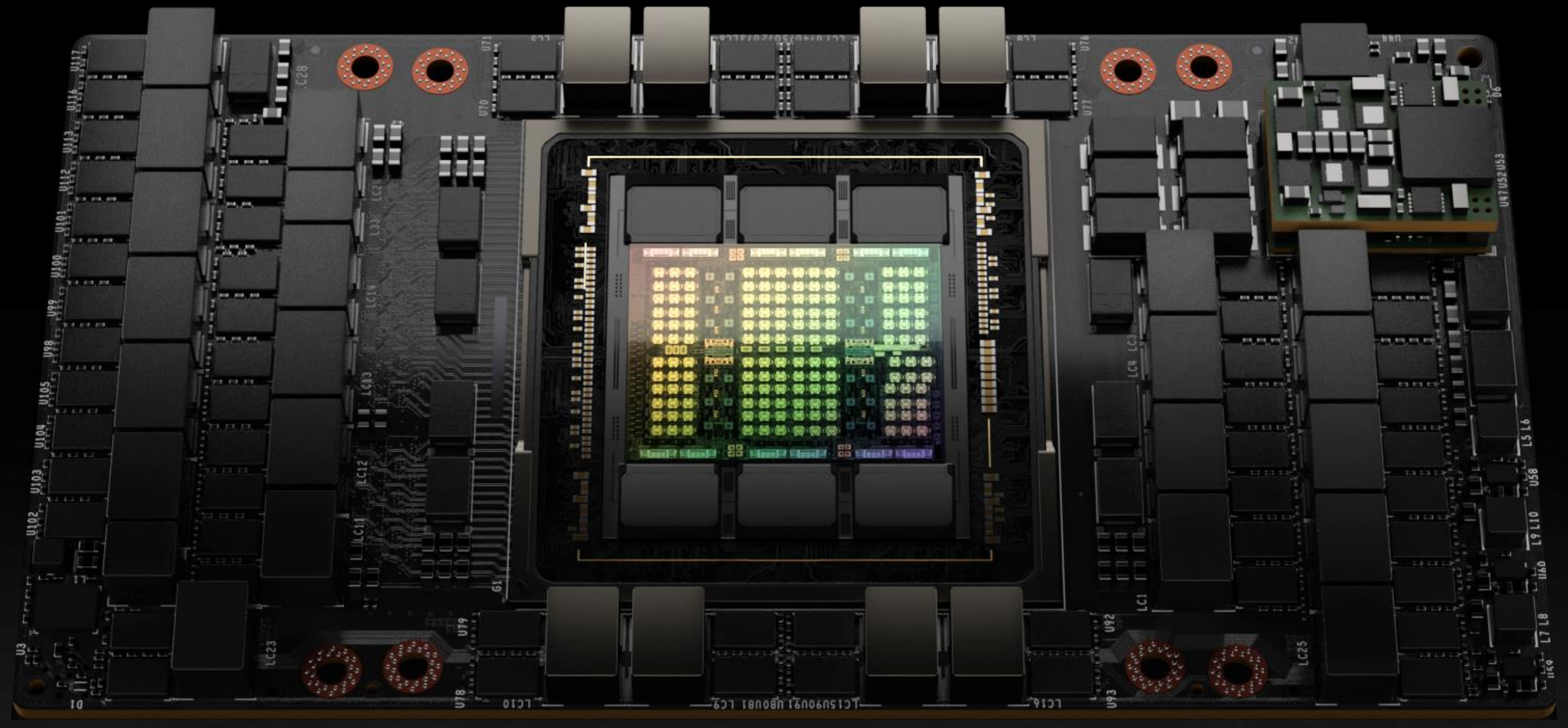

OPTIMIZING CUDA APPLICATIONS FOR NVIDIA HOPPER ARCHITECTURE

Guillaume Thomas-Collignon, Vishal Mehta

DevTech Compute, GTC 2022

Optimizing CUDA Applications for NVIDIA Hopper Architecture [S41489]

INTRODUCING HOPPER



H100 Physical Architectural Features

132 Streaming Multiprocessors (SMs)

PCIe Gen5 with PCIe Atomics

HBM3 Memory with 3TB/sec Bandwidth

50MB L2 Cache

4th Generation NVLink @ 900GB/sec total bandwidth

New NVLink Switch system: Up to 256 GPUs, SHARP in-network compute

H100 Next-Generation Capabilities

Thread Block Clusters

Distributed Shared Memory

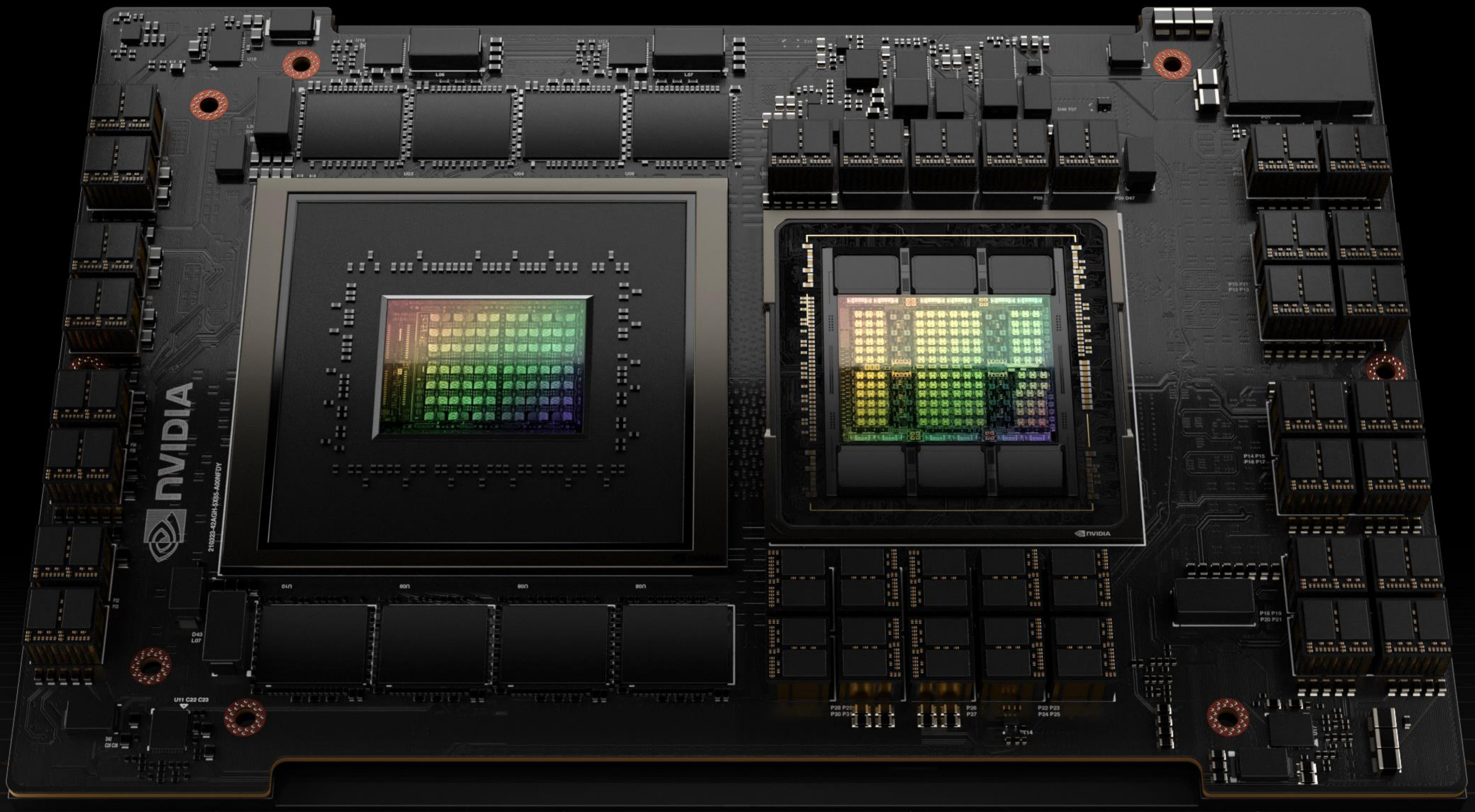
Tensor Memory Accelerator (TMA)

Tensor Core Transformer Engine

Confidential Computing Support

Asynchronous Architecture

Inside the NVIDIA Hopper Architecture [S42663]



NVIDIA GRACE HOPPER

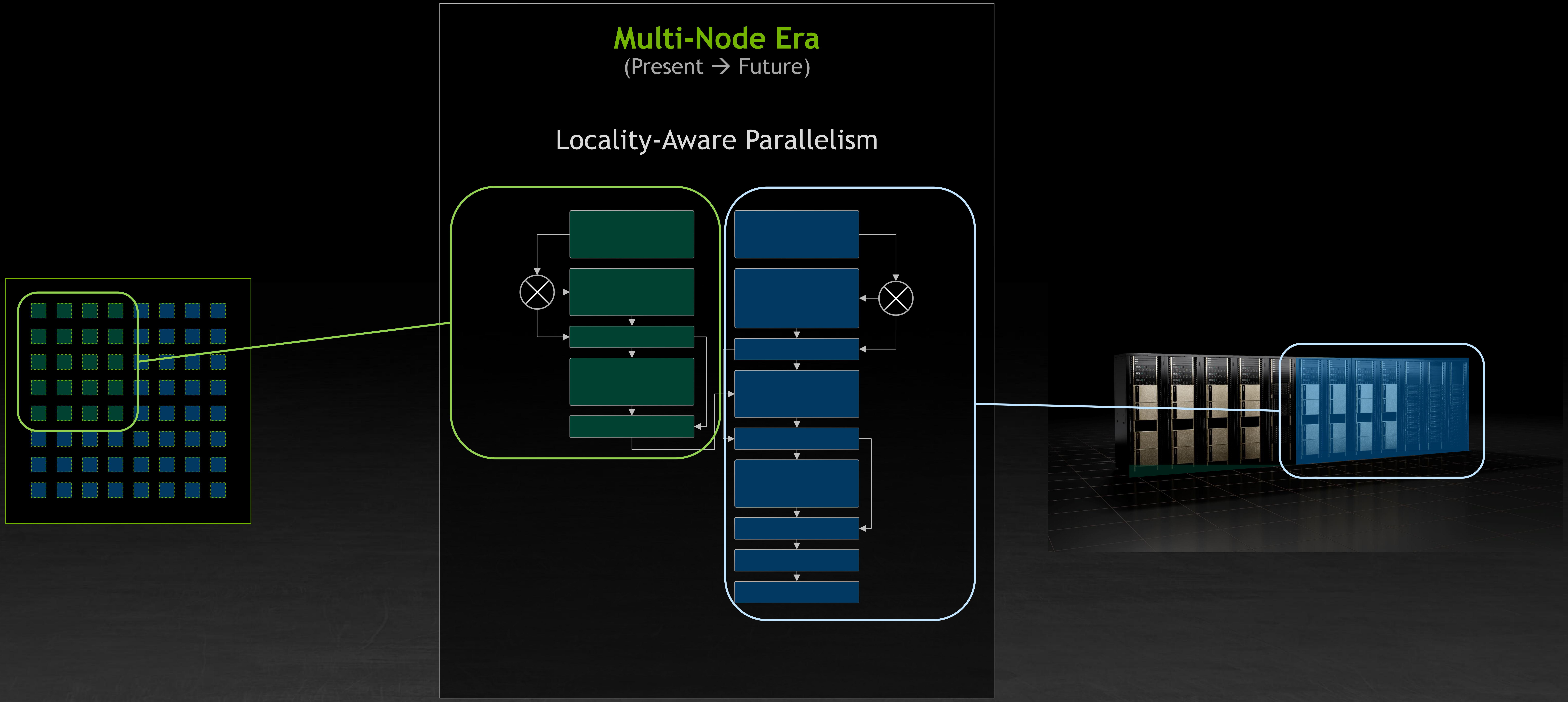
Grace Hopper Superchip

Densest NVIDIA Accelerated Computing System

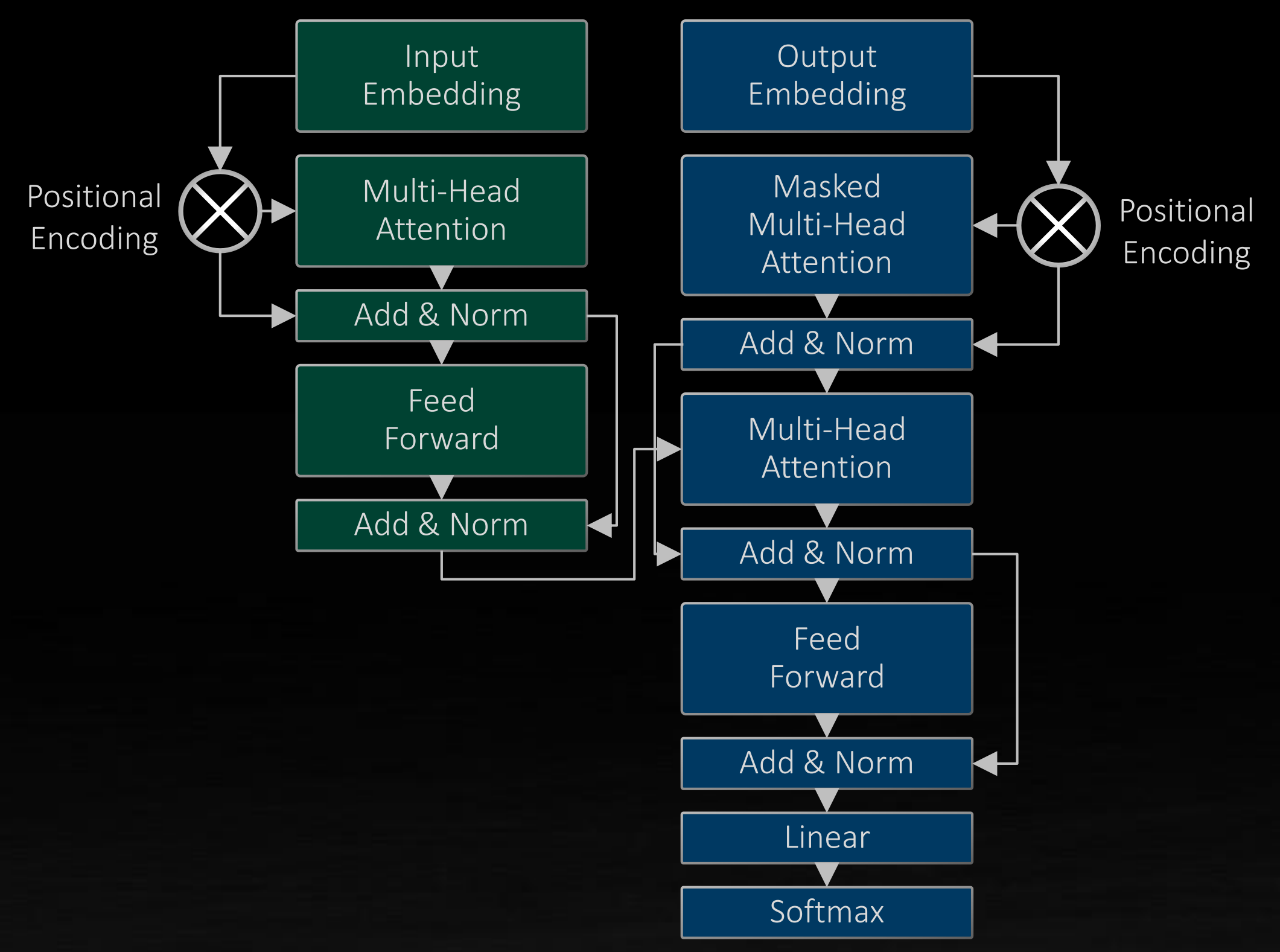
New NVLink Chip-to-Chip Coherent Interface

900 GB/s

EXPLOITING LOCALITY, EXPOSING PARALLELISM



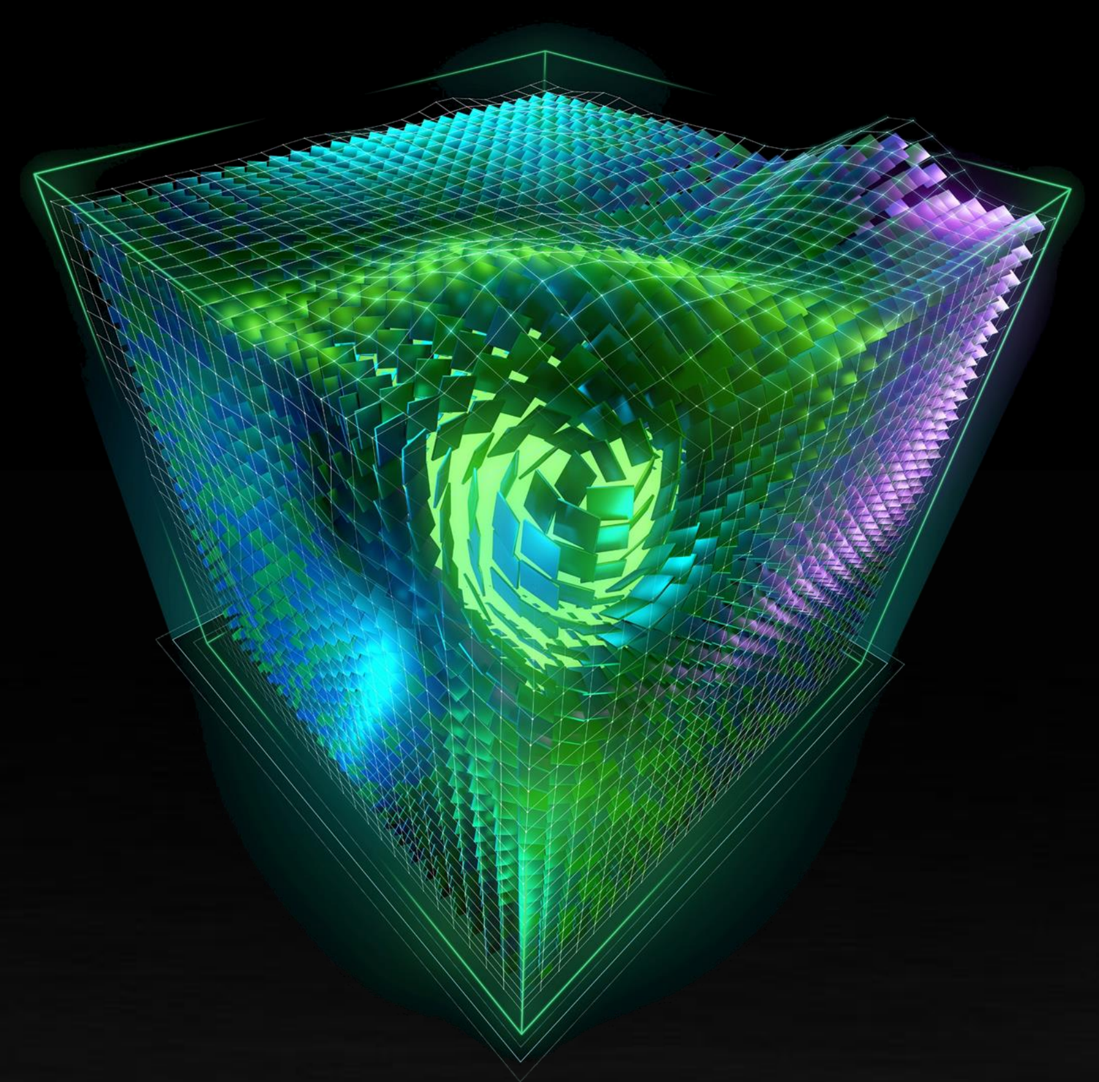
PROGRAMMING TO THE HIERARCHY



Application level

Framework level

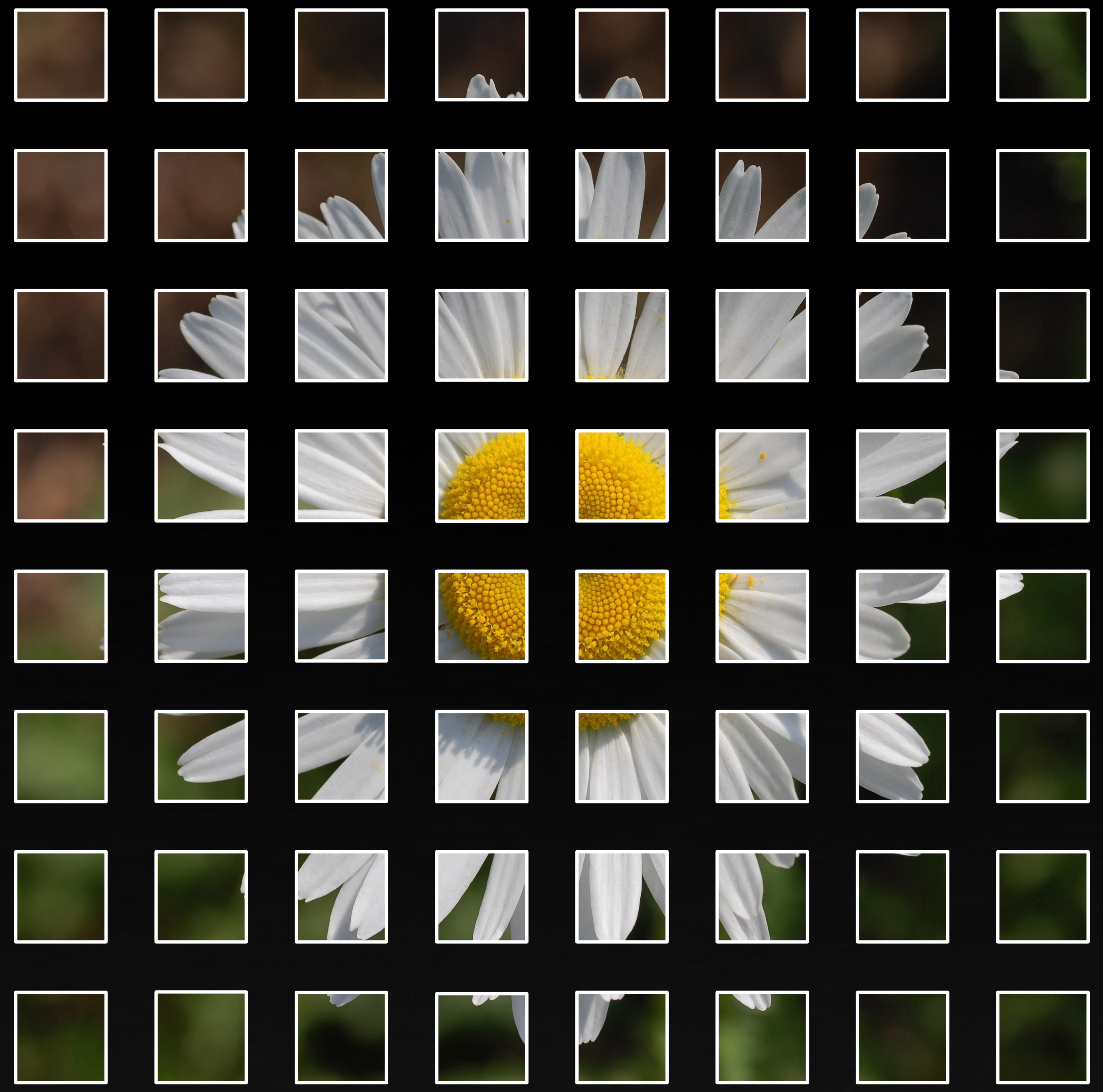
Library level



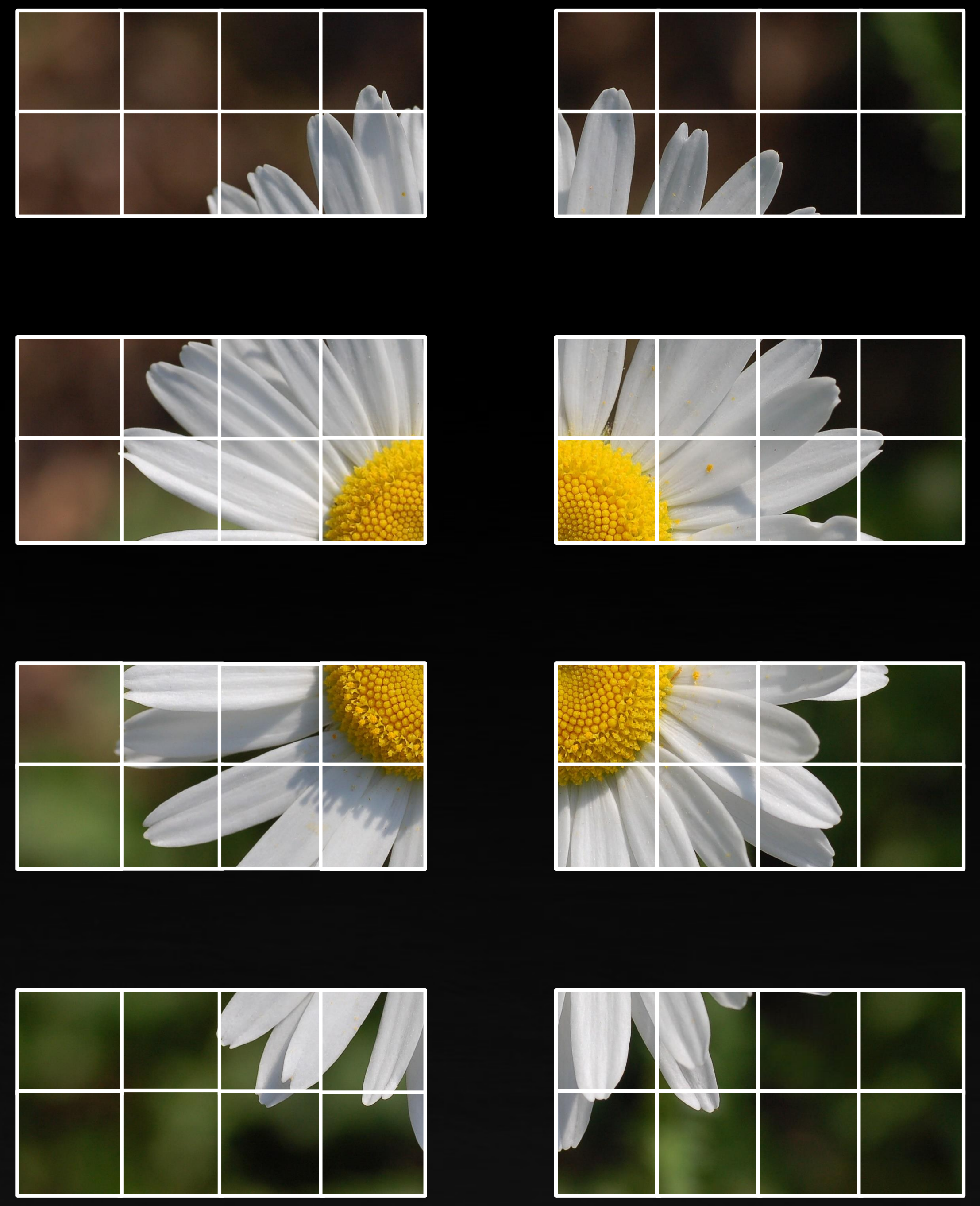
Runtime level

SCALING: DATA PARALLELISM + LOCALITY OF DATA

Scheduling based on throughput

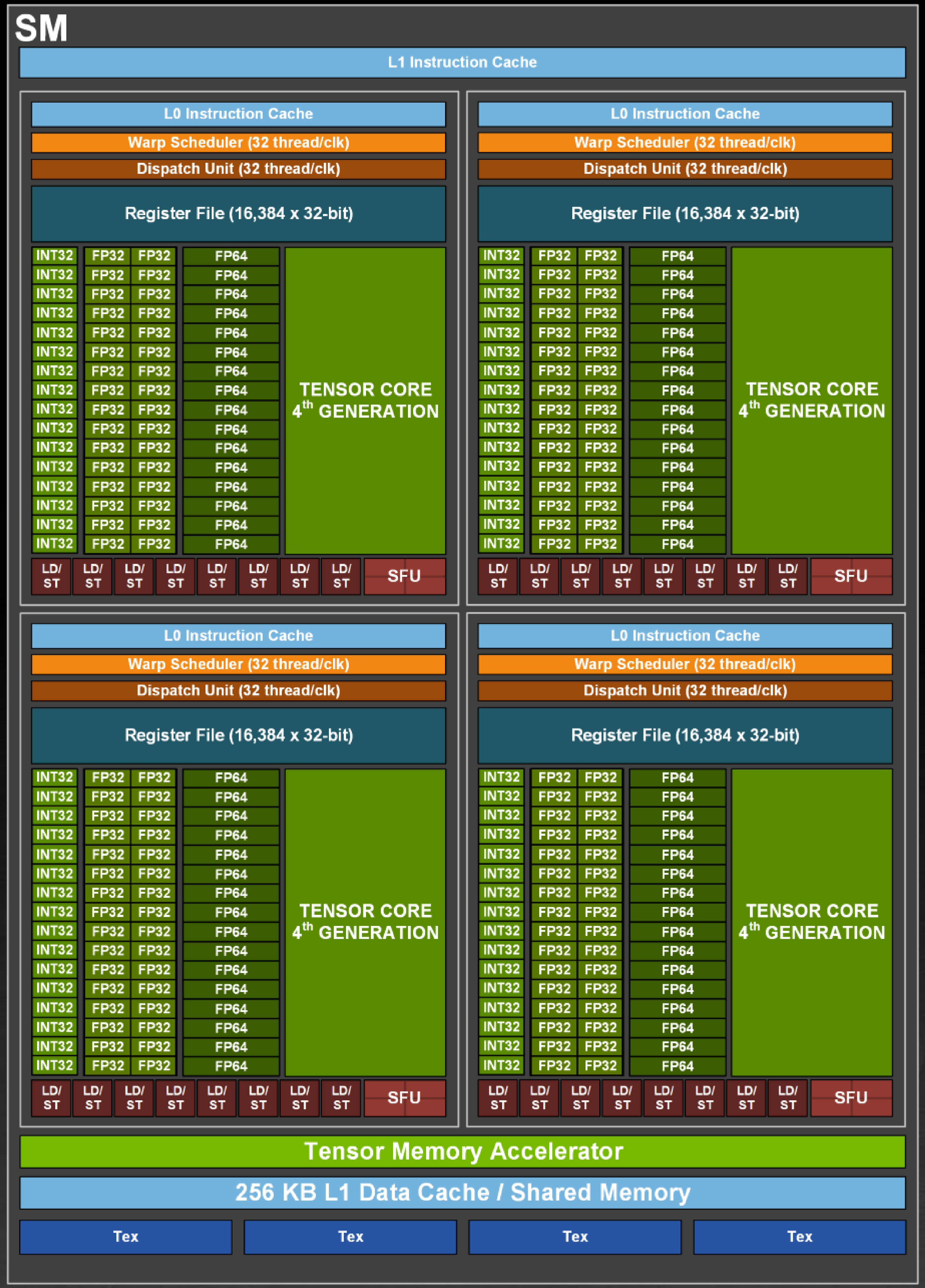


Placement based on data reuse



HOPPER ARCHITECTURE

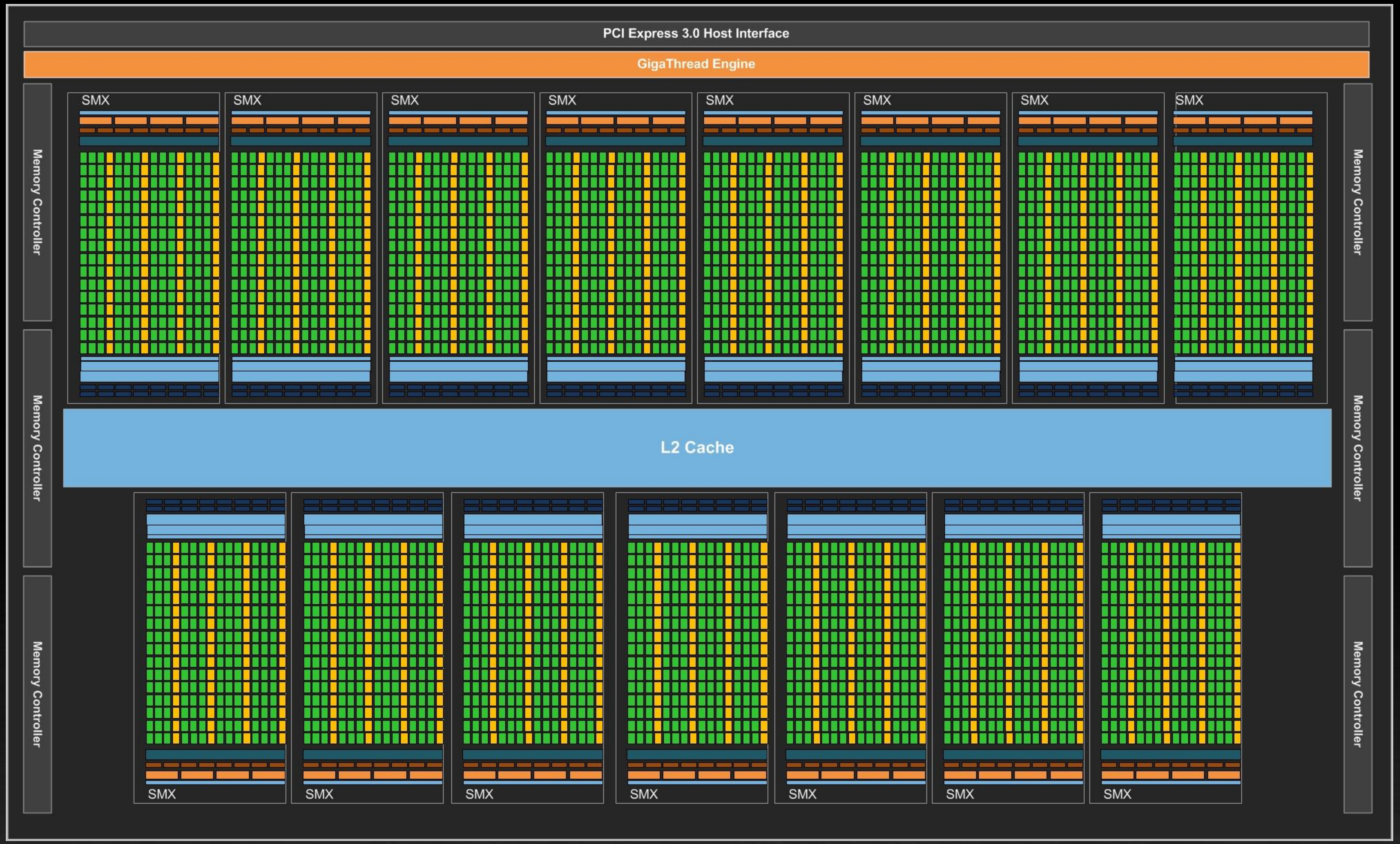
H100 Streaming Multiprocessor Key Features


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

- 256 KB combined L1 cache/Shared memory per SM. 33% over A100
- New Thread Block Clusters and Distributed Shared Memory
- New Tensor Memory Accelerator and Asynchronous Transaction Barriers
- 4th Generation Tensor Core, 2x perf per clock

SOME HISTORY: THE KEPLER GK110 GPU, 2012

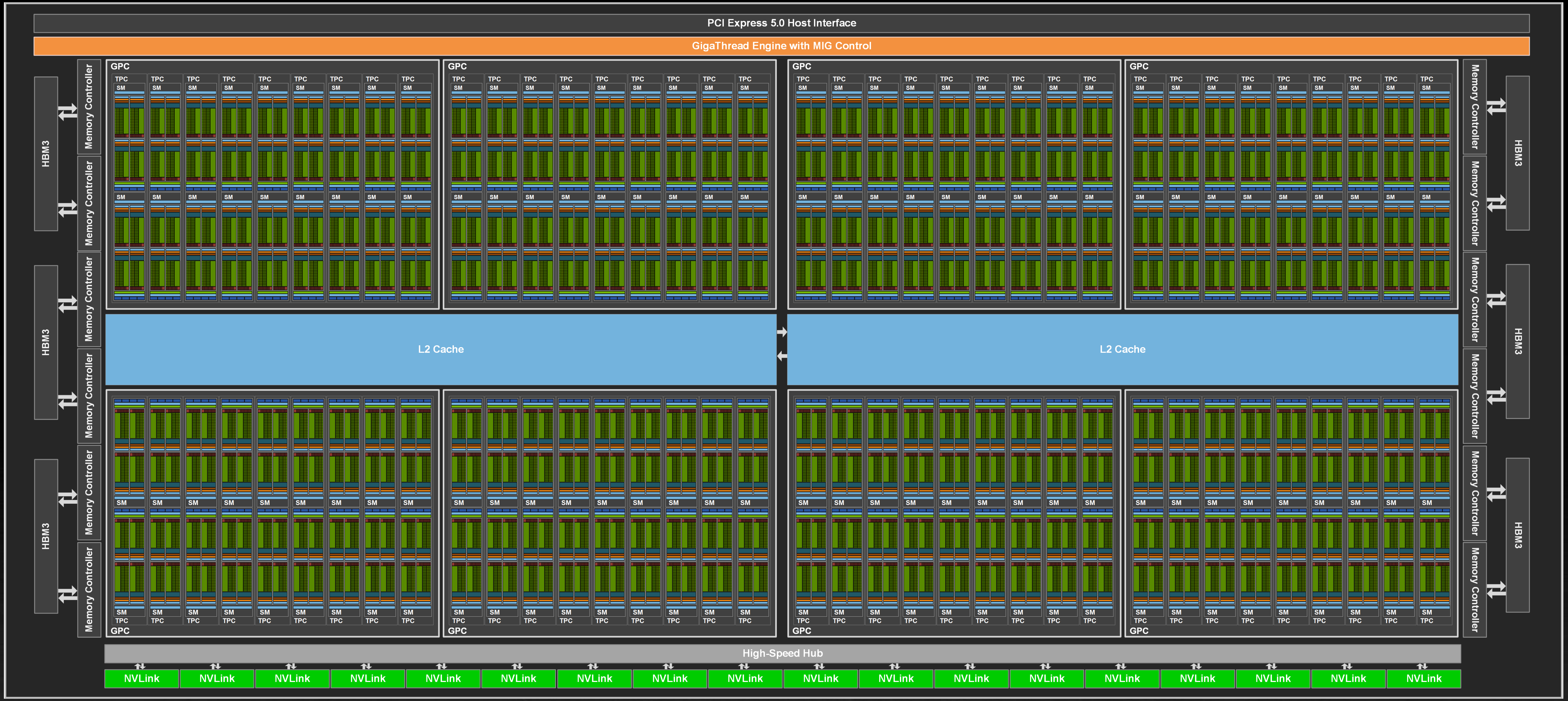
Tesla K20x



Kepler GK110 Full Chip

15 SMs

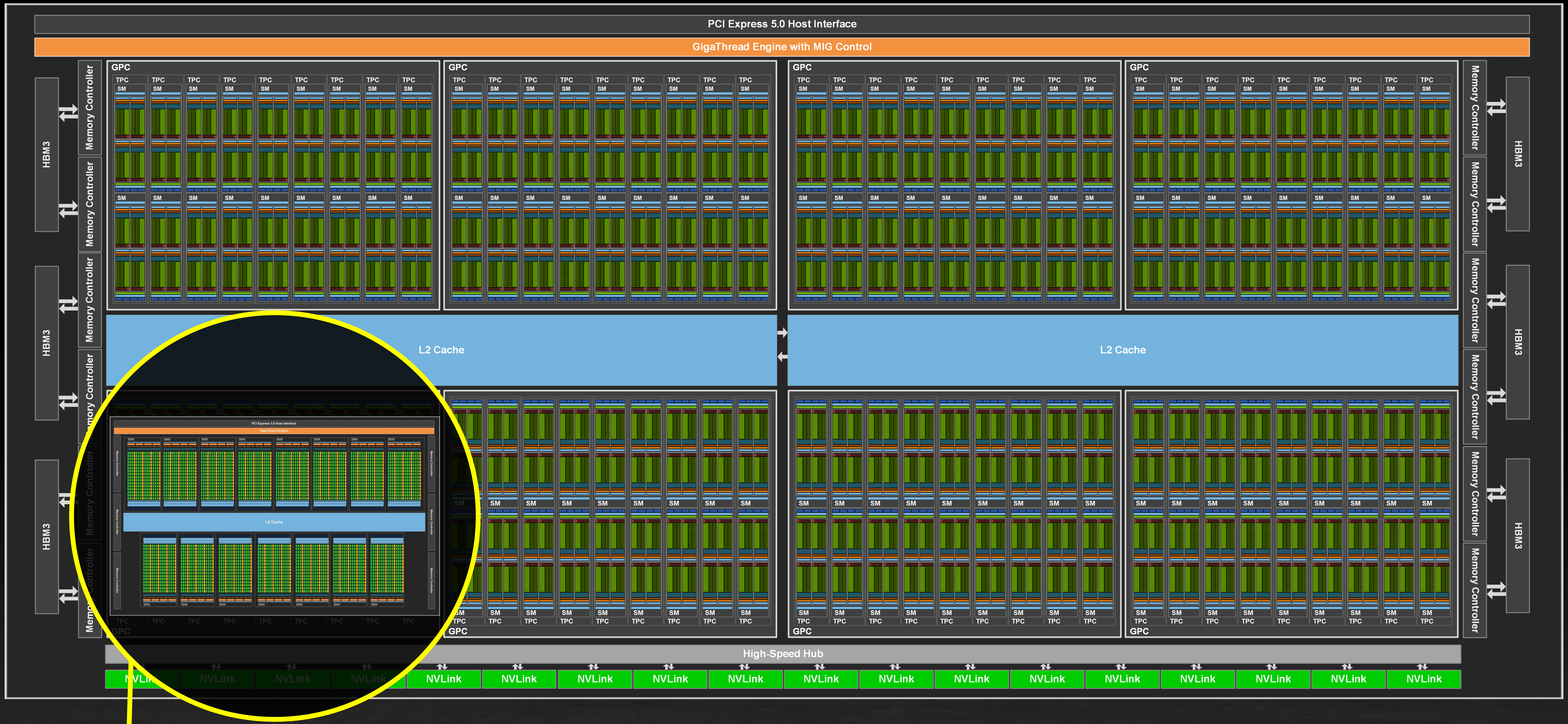
THE HOPPER H100 GPU, 2022



Hopper H100 Full Chip

132 SMs

9x SMs OVER 10 YEARS



PCI Express 5.0 Host Interface
GigaThread Engine with MIG Control
GPC
TPC
SM
Memory Controller
HBM3
L2 Cache
High-Speed Hub
NVLink

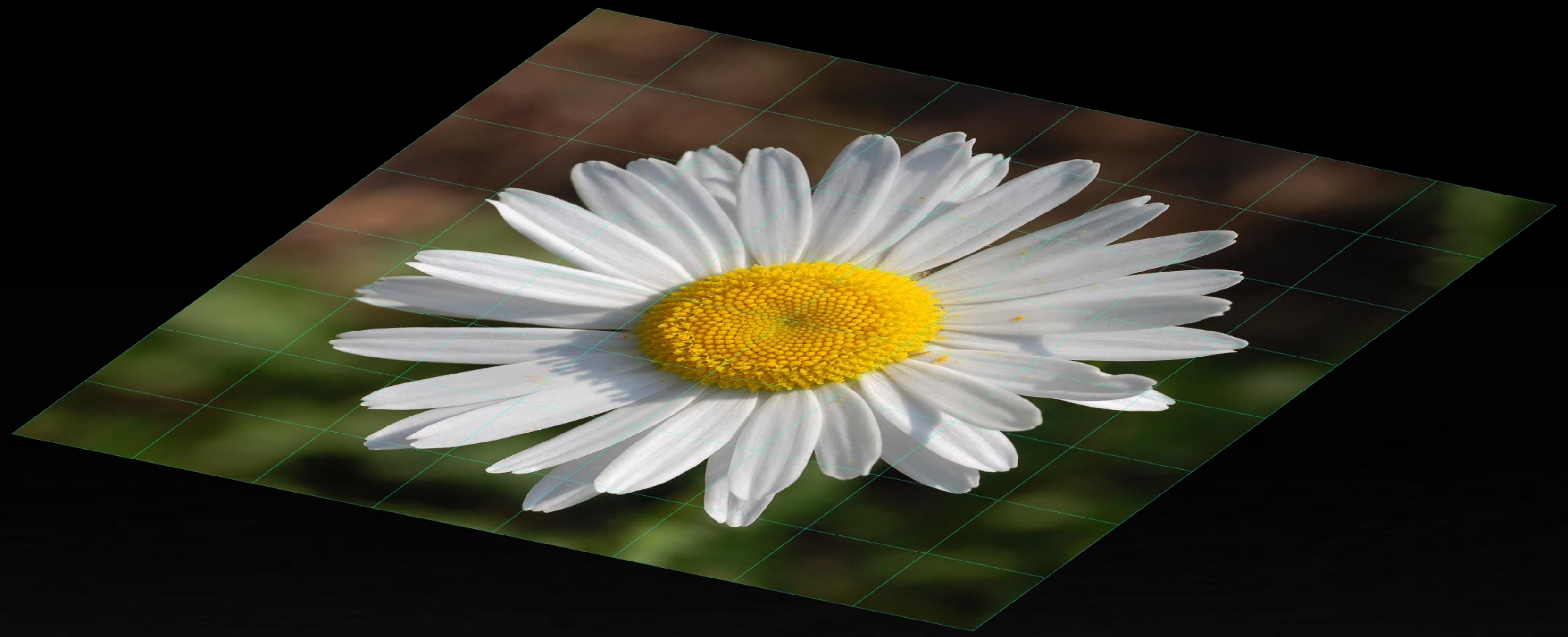
Hopper H100 Full Chip

132 SMs

Kepler GK110 Full Chip

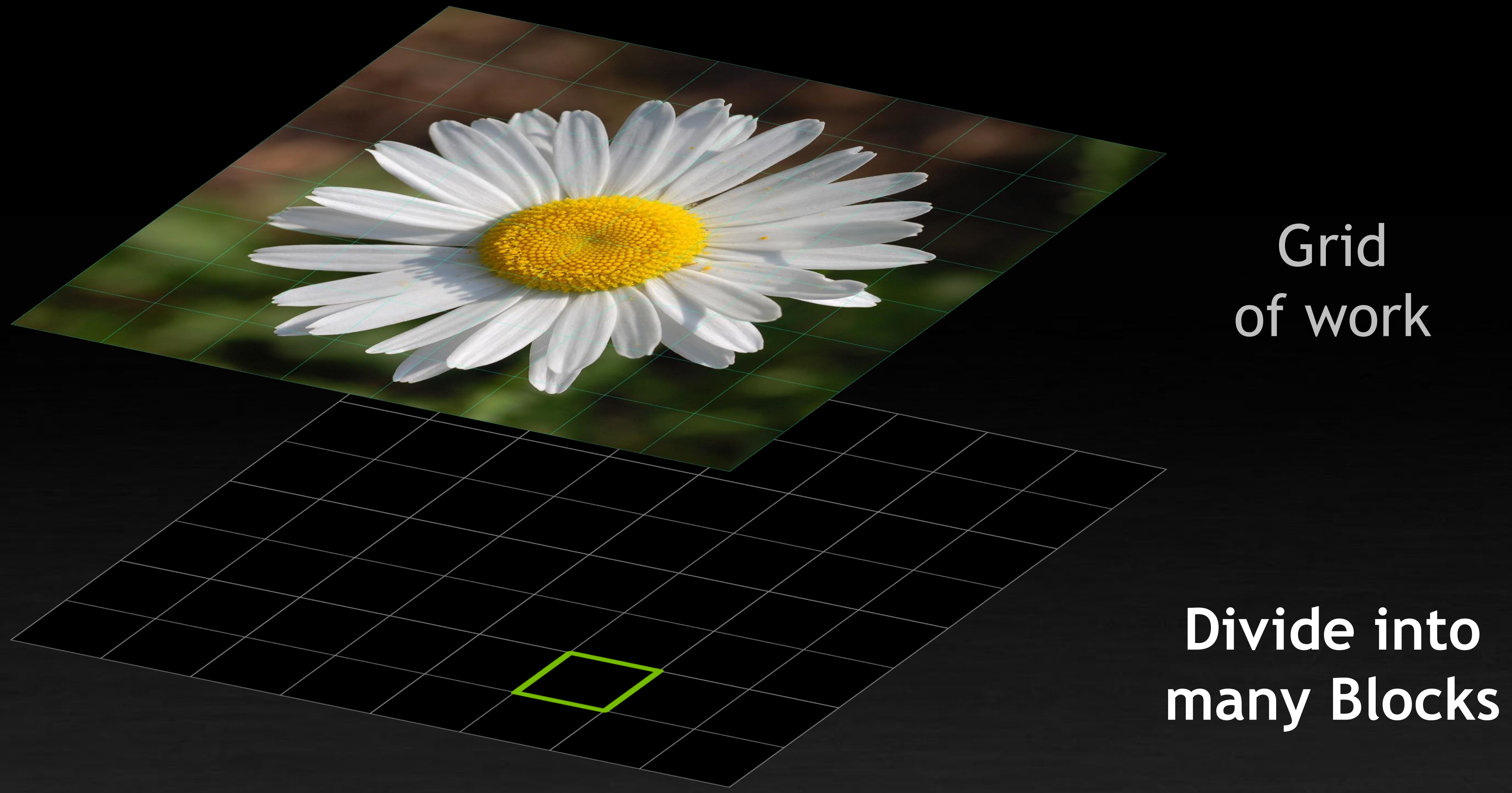
15 SMs

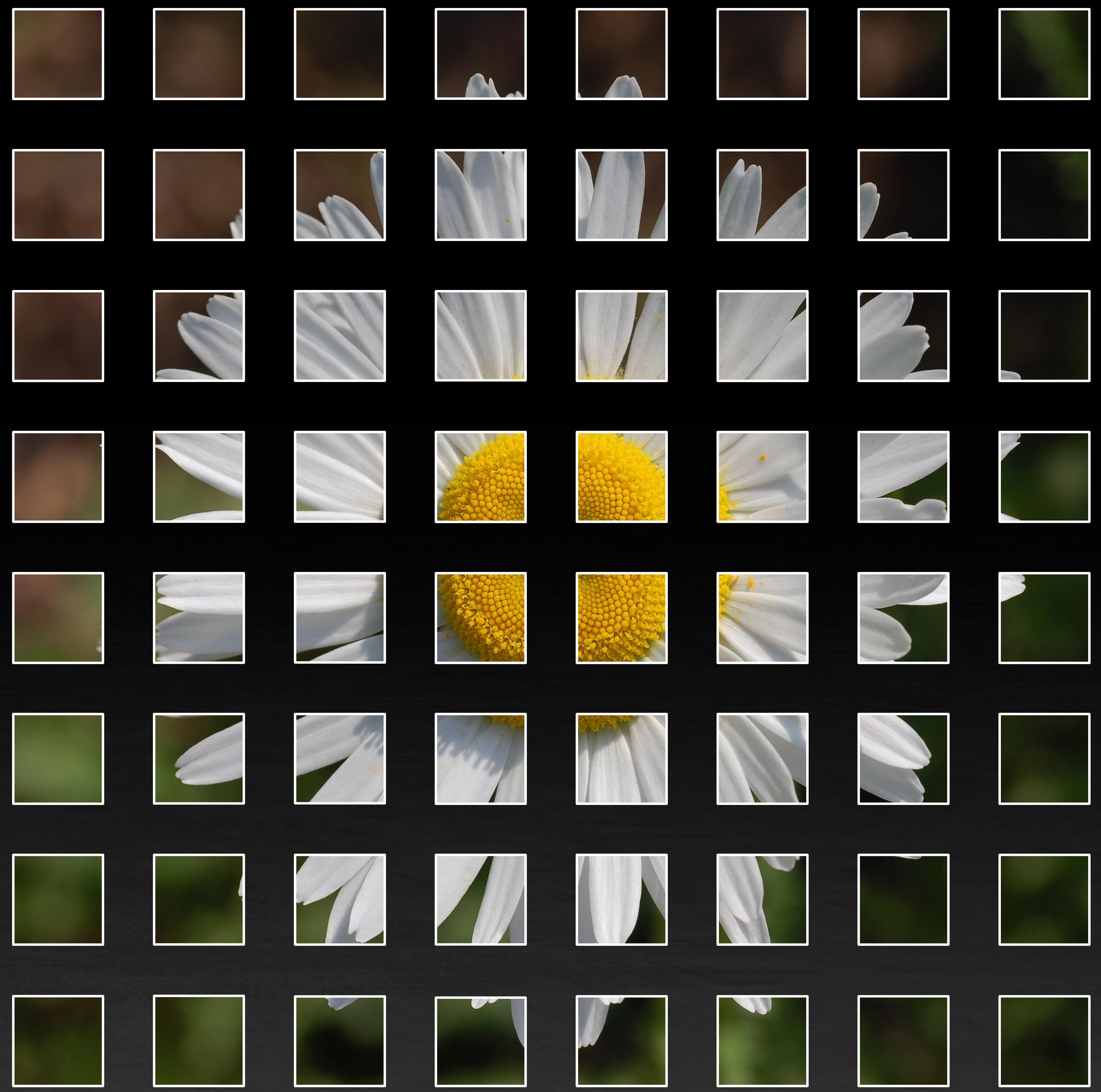
THE CUDA PROGRAMMING MODEL: GRID → BLOCKS → THREADS



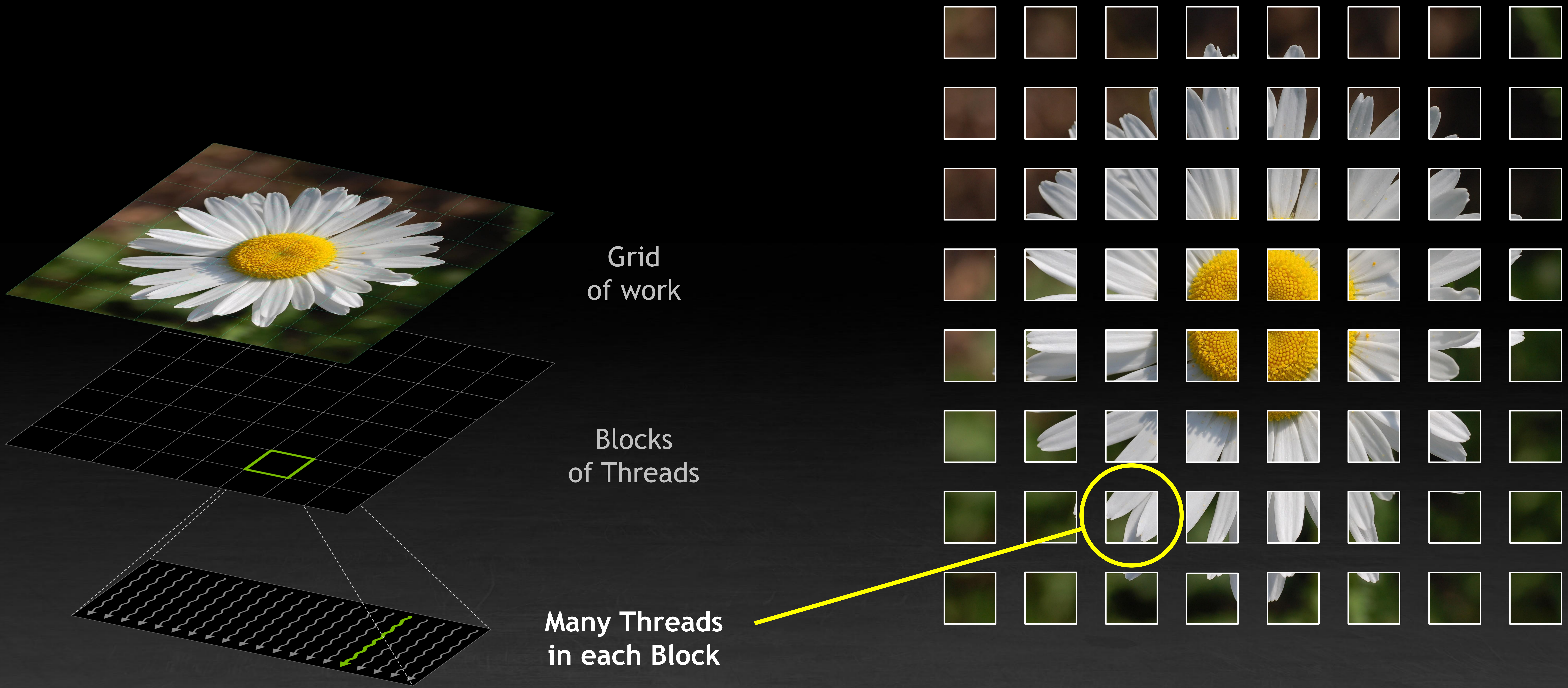
Grid
of work

DIVIDE THE WORK INTO A GRID OF EQUAL BLOCKS



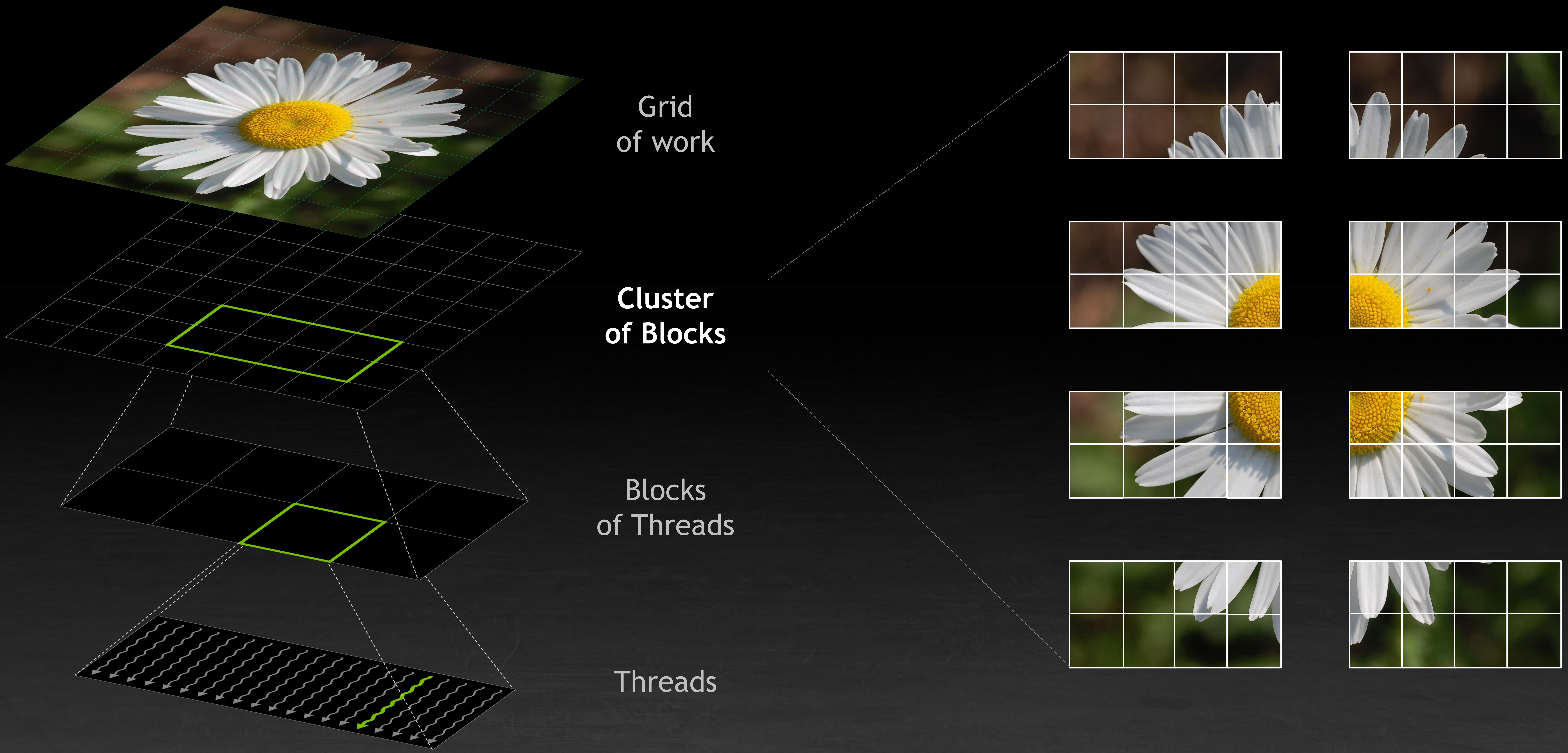


EACH BLOCK RUNS AS IF IT'S AN INDEPENDENT PROGRAM

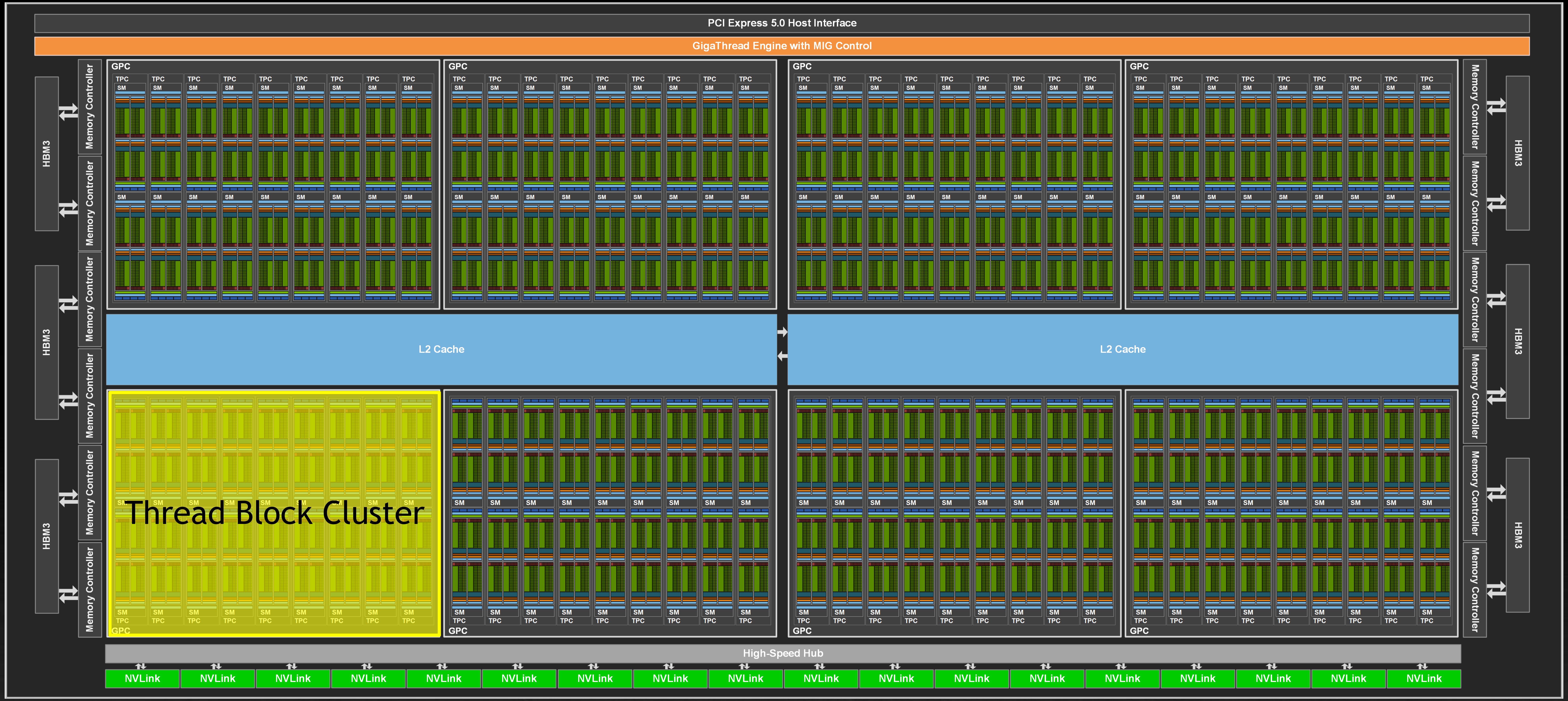


THREAD BLOCK CLUSTER

A collective of blocks, co-scheduled on adjacent multiprocessors



TAKING ADVANTAGE OF LOCALITY AT A GPU SCALE



Guaranteed co-located blocks
New tier of guaranteed concurrency
Fast data exchange & sync

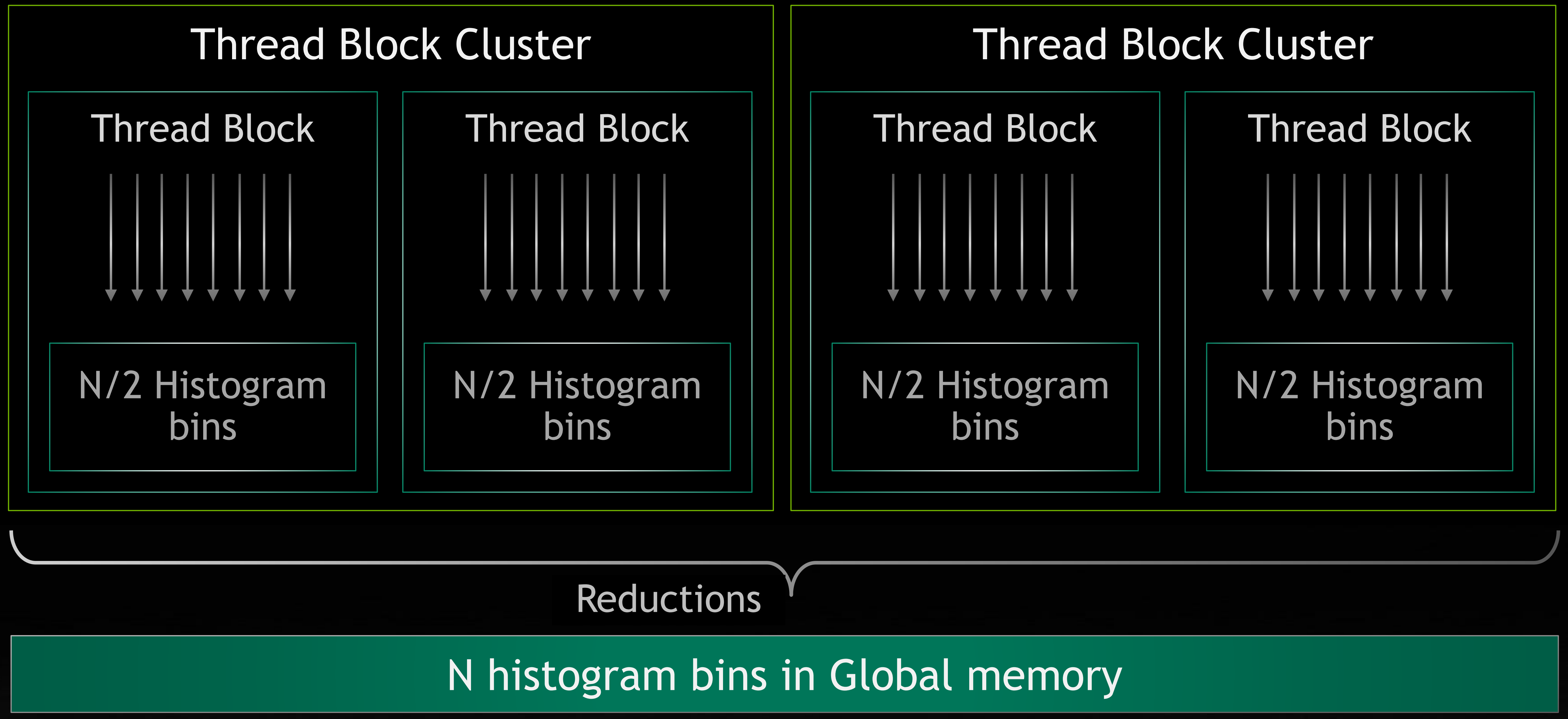
CLUSTER DISTRIBUTED SHARED MEMORY (DSMEM)

Blocks within a cluster are able to access each others’ shared memory directly

4x2 Cluster
SM 0
Block 0,0
Shared Memory
SM 1
Block 1,0
Shared Memory
SM 2
Block 2,0
Shared Memory
SM 3
Block 3,0
Shared Memory
All-to-All Shared Memory Access
Shared Memory
Block 0,1
SM 4
Shared Memory
Block 1,1
SM 5
Shared Memory
Block 2,1
SM 6
Shared Memory
Block 3,1
SM 7

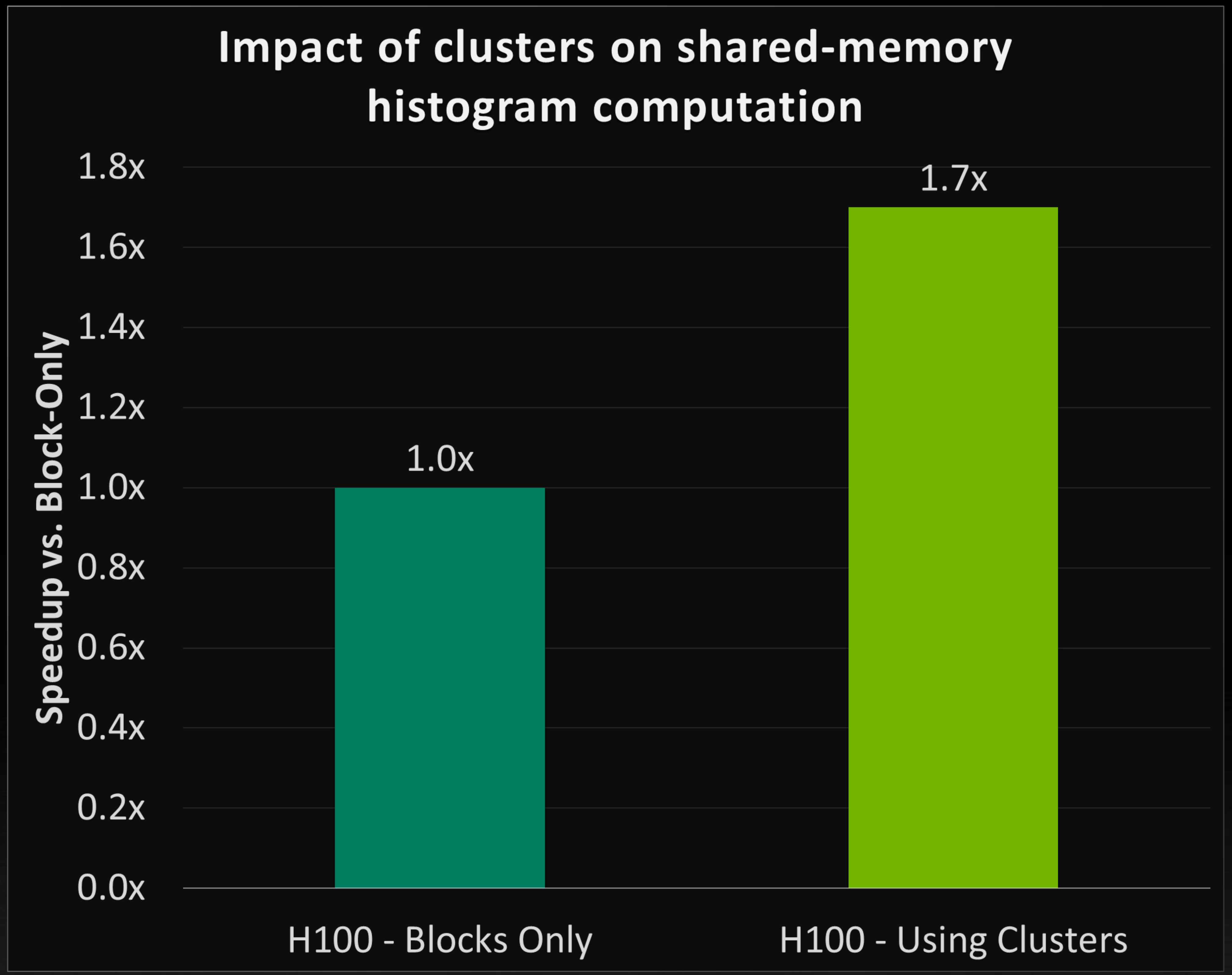
Full load/store/atomic access to all shared memory between blocks within a cluster

EXAMPLE: HIERARCHICAL HISTOGRAM USING CLUSTER DSMEM



Histograms in CUDA are typically computed in shared memory, followed by reductions in global memory.

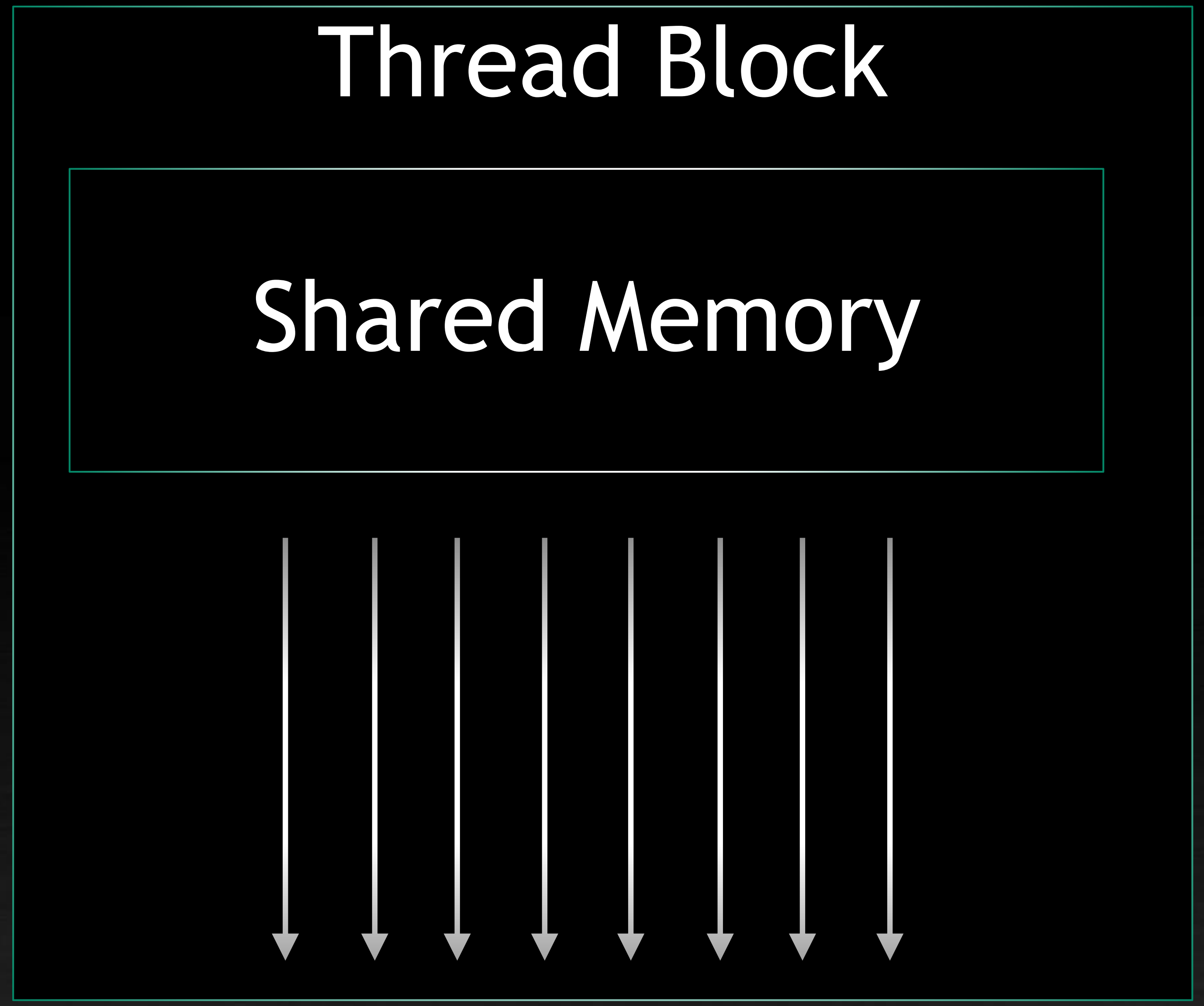
For large histograms, shared memory capacity of a single block is not sufficient.



75K Histogram bins (300KB) fit in distributed shared memory of 2-block clusters → 37.5K (150KB) per thread block

THREAD AND MEMORY HIERARCHY

CUDA Thread & Memory Hierarchy pre-Hopper

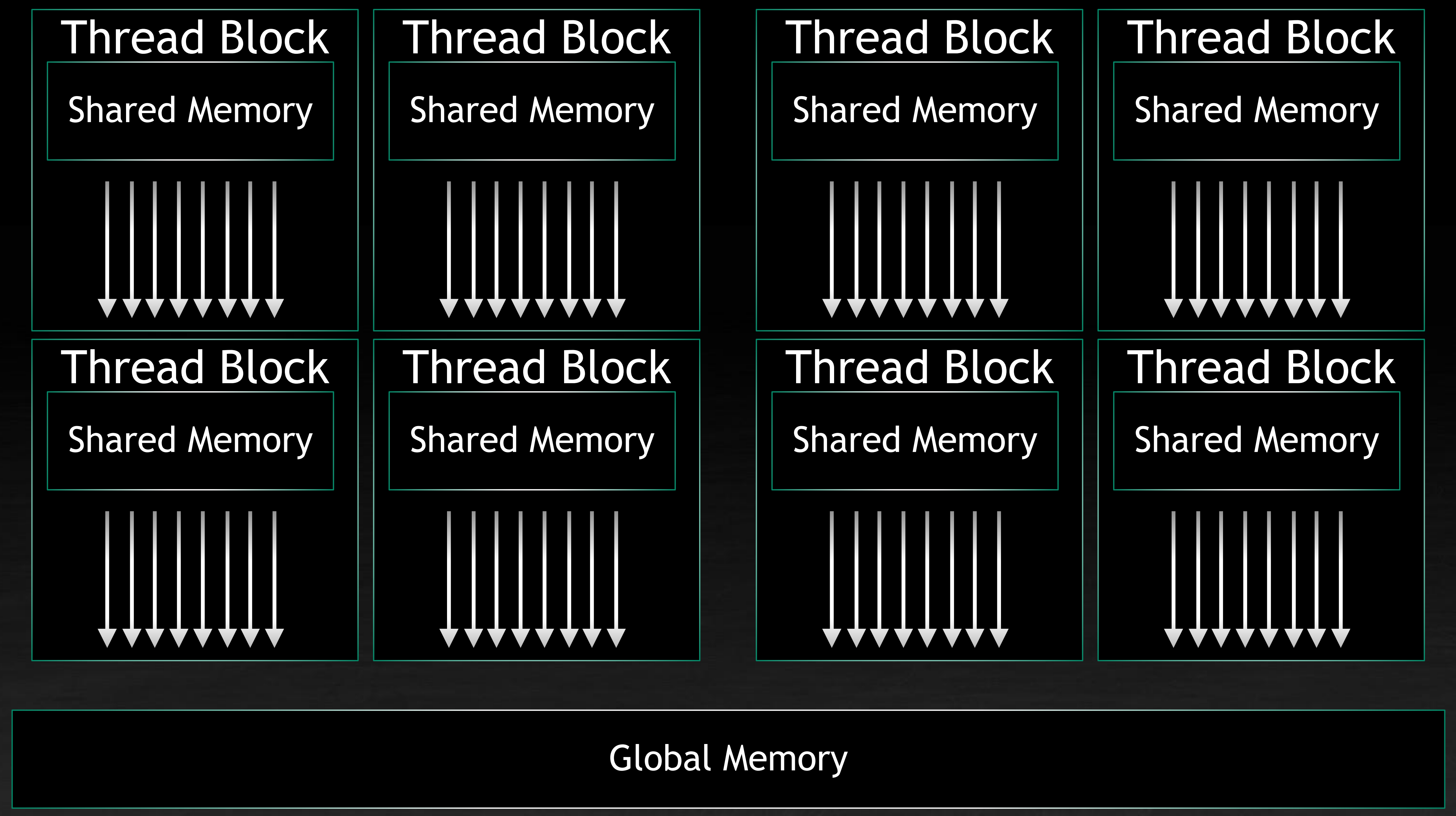


Threads in the same Thread Block

- collaborate via shared memory
- are guaranteed to be co-scheduled on same SM
- can synchronize / communicate data using
  - `__syncthreads();`
  - `cooperative_groups::this_thread_block.sync();`
  - `cuda::barrier<thread_scope_block>` `::arrive()` and `::wait()`
- can also perform collectives like `cooperative_groups::reduce()`

THREAD AND MEMORY HIERARCHY

CUDA Thread & Memory Hierarchy pre-Hopper



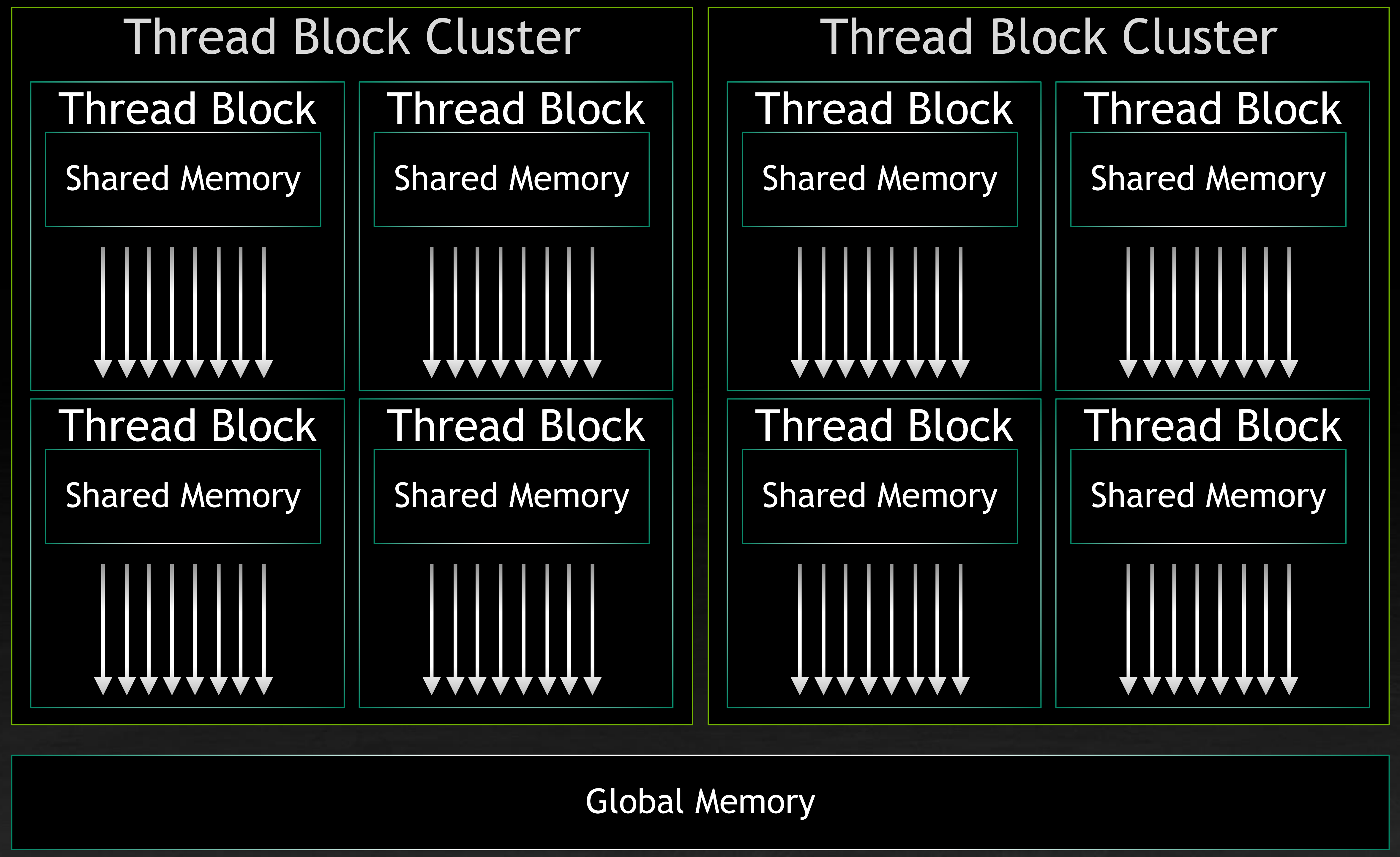
All thread blocks share global memory to collaborate

Independent thread blocks can be scheduled out of order to improve occupancy, and hence GPU utilization

CUDA cooperative launch is required for all thread blocks to synchronize on the GPU.

THREAD AND MEMORY HIERARCHY

Introducing Thread Block Clusters in Hopper



Thread Block Clusters introduce a new **optional** level of hierarchy in the CUDA programming model.

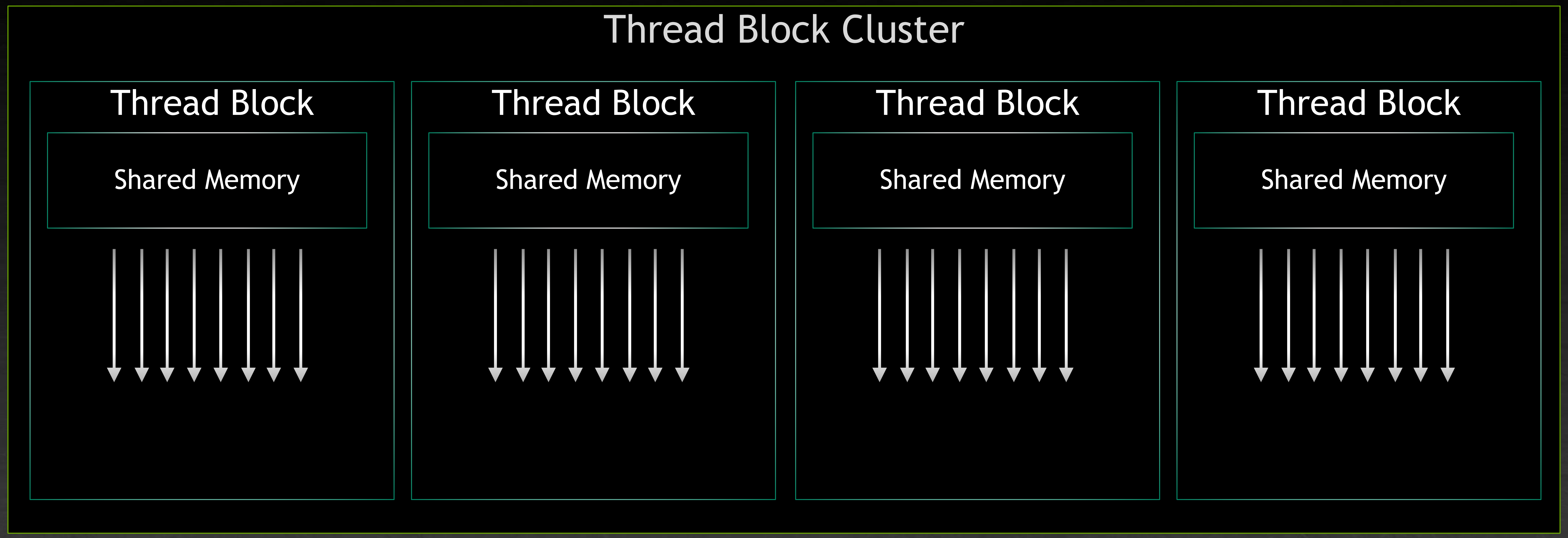
Thread Blocks in a Cluster are guaranteed to be co-scheduled on SMs in a GPU Processing Cluster (GPC)

All shared memory within a cluster forms Distributed Shared Memory

THREAD AND MEMORY HIERARCHY

Getting the current cluster

```
namespace cg = cooperative_groups;
auto block = cg::this_thread_block();
cg::cluster_group cluster = cg::this_cluster();

<..>
```



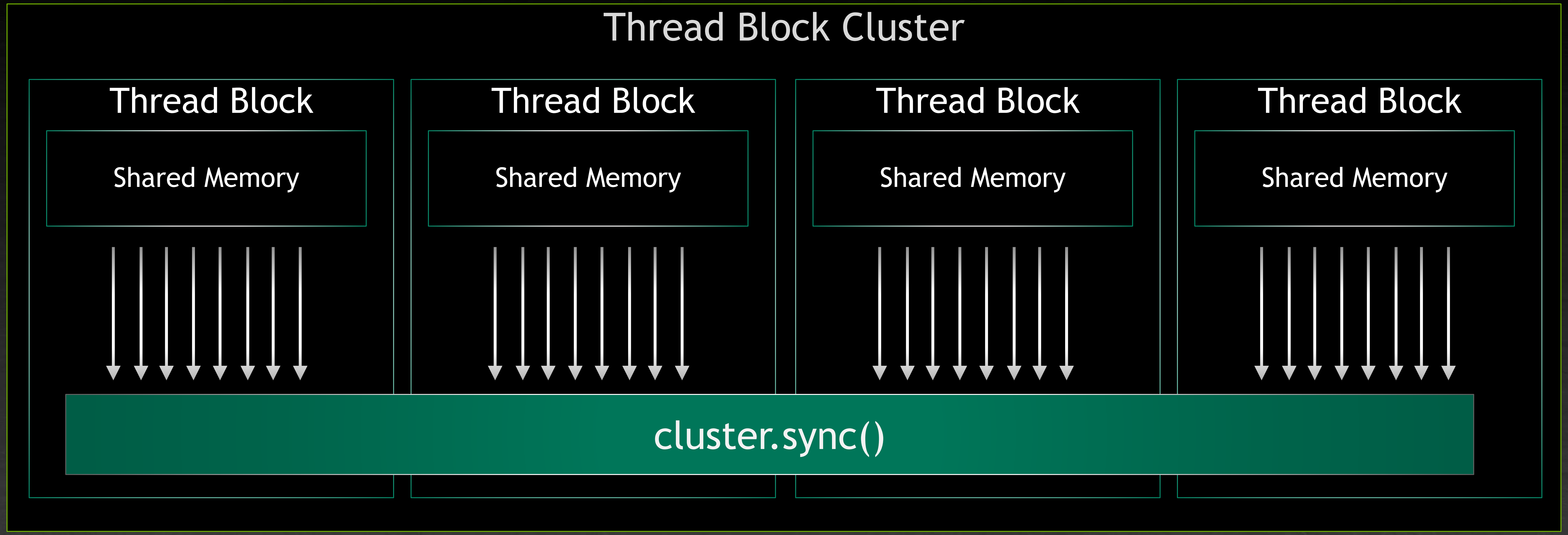
THREAD AND MEMORY HIERARCHY

Accelerated Synchronization for all threads in cluster

```
namespace cg = cooperative_groups;
auto block = cg::this_thread_block();
cg::cluster_group cluster = cg::this_cluster();

<..>

cluster.sync();
```

Cluster synchronization is accelerated in hardware

H100 can support up to 16 thread blocks or 16384 threads per cluster



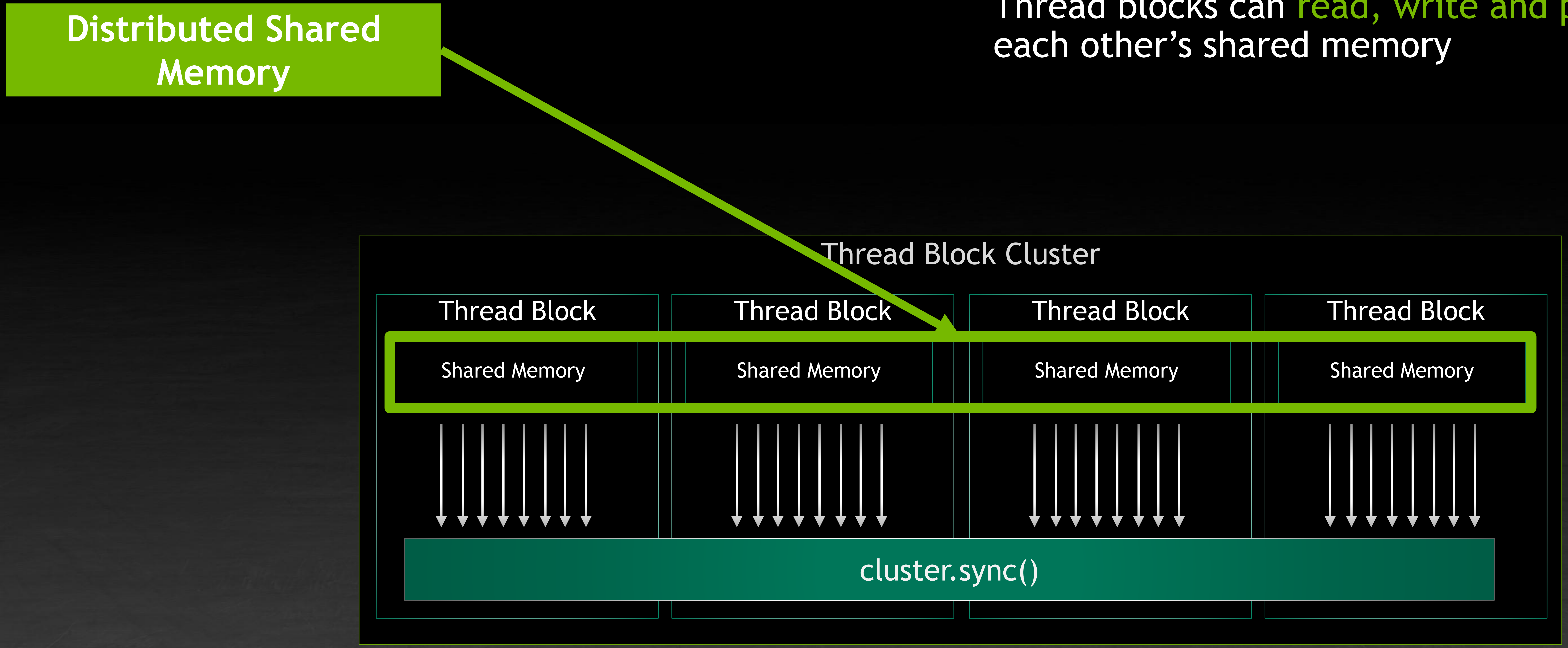
Disclaimer: Preliminary CUDA API, subject to change

THREAD AND MEMORY HIERARCHY

Distributed Shared Memory Operations

All blocks within a thread block cluster can collaborate using Distributed Shared Memory

Thread blocks can read, write and perform atomics on each other's shared memory



Disclaimer: Preliminary CUDA API, subject to change

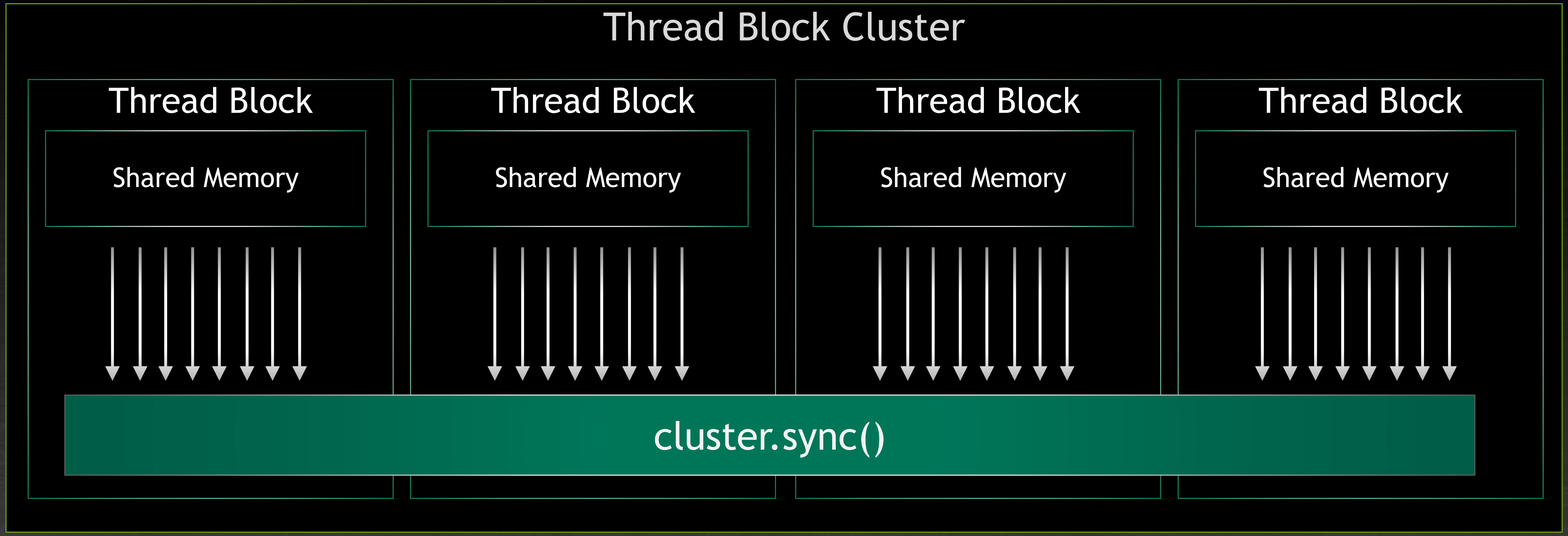
THREAD AND MEMORY HIERARCHY

Distributed Shared Memory Operations

```
// All blocks in the cluster have the variable smem
__shared__ int smem;
namespace cg = cooperative_groups;
cg::cluster_group cluster = cg::this_cluster();
unsigned int BlockRank = cluster.block_rank();
int cluster_size = cluster.dim_blocks().x;
```

All blocks within a thread block cluster can collaborate using Distributed shared memory

Thread blocks can read, write and perform atomics on each other's shared memory



Disclaimer: Preliminary CUDA API, subject to change

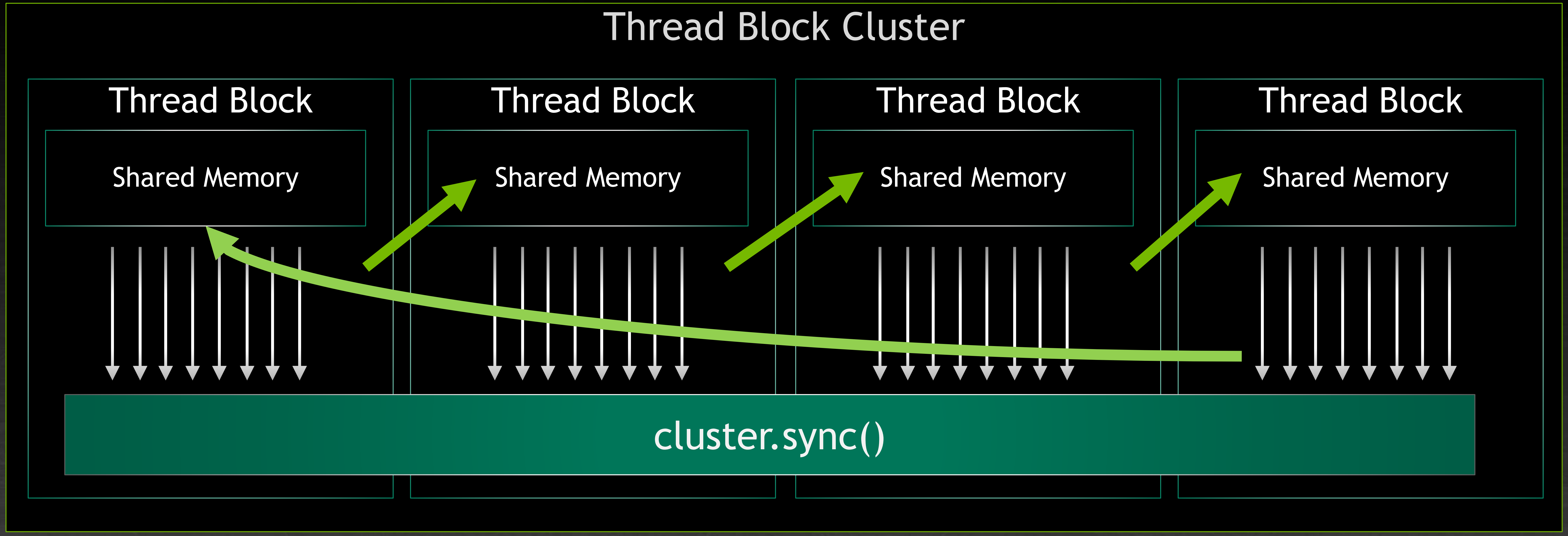
THREAD AND MEMORY HIERARCHY

Distributed Shared Memory Operations

```
// All blocks in the cluster have the variable smem
__shared__ int smem;
namespace cg = cooperative_groups;
cg::cluster_group cluster = cg::this_cluster();
unsigned int BlockRank = cluster.block_rank();
int cluster_size = cluster.dim_blocks().x;

// Get a pointer to peer smem variable based on
// pointer from current block
int *remote_smem = cluster.map_shared_rank(&smem,
                (BlockRank + 1) % cluster_size);

if (threadIdx.x == 0)
  *remote_smem = 10; // Store to remote memory

cluster.sync(); // Sync to ensure
                // store is done
```

All blocks within a thread block cluster can collaborate using Distributed shared memory

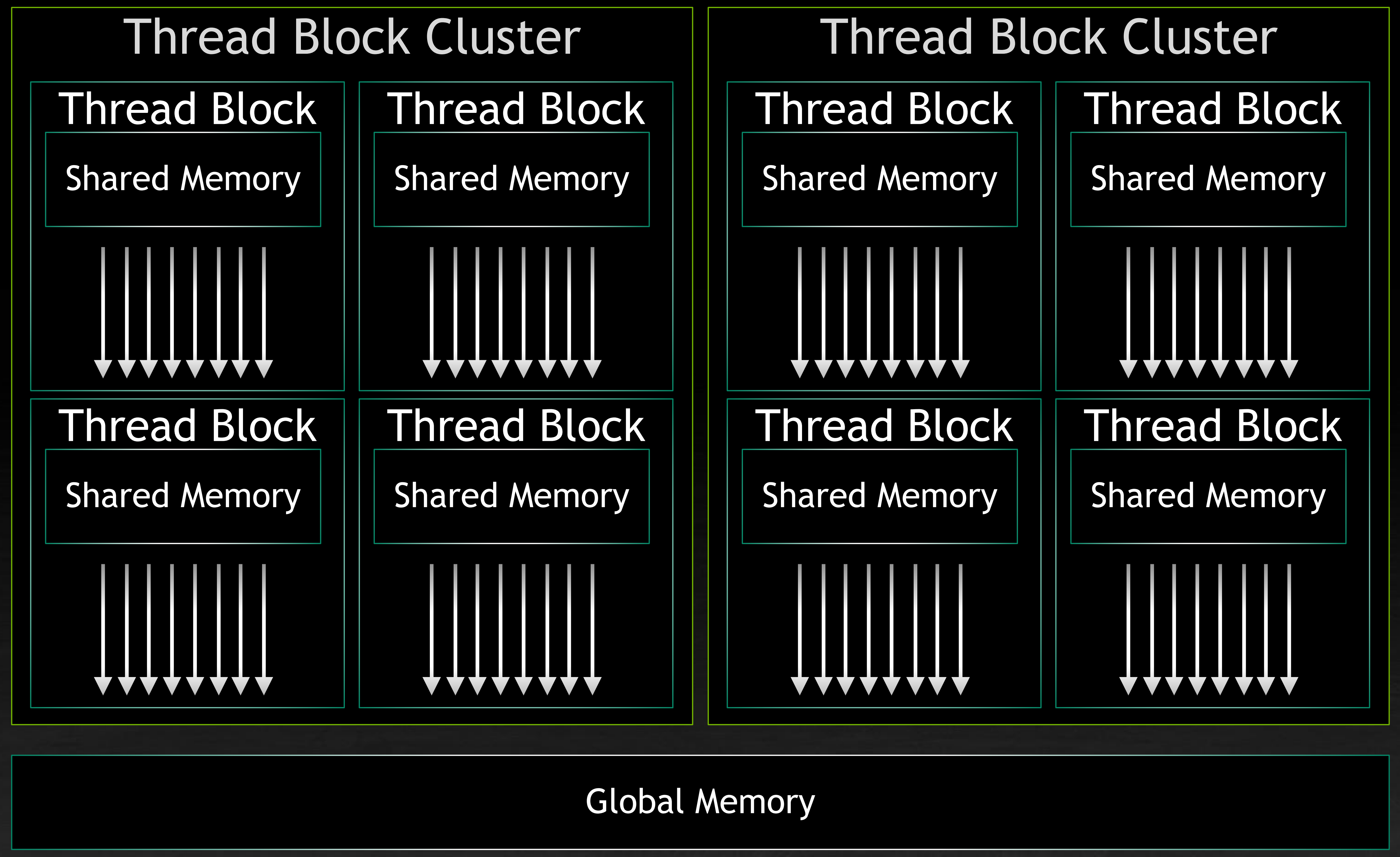
Thread blocks can read, write and perform atomics on each other's shared memory



Disclaimer: Preliminary CUDA API, subject to change

THREAD AND MEMORY HIERARCHY

Launching CUDA Kernels with Clusters



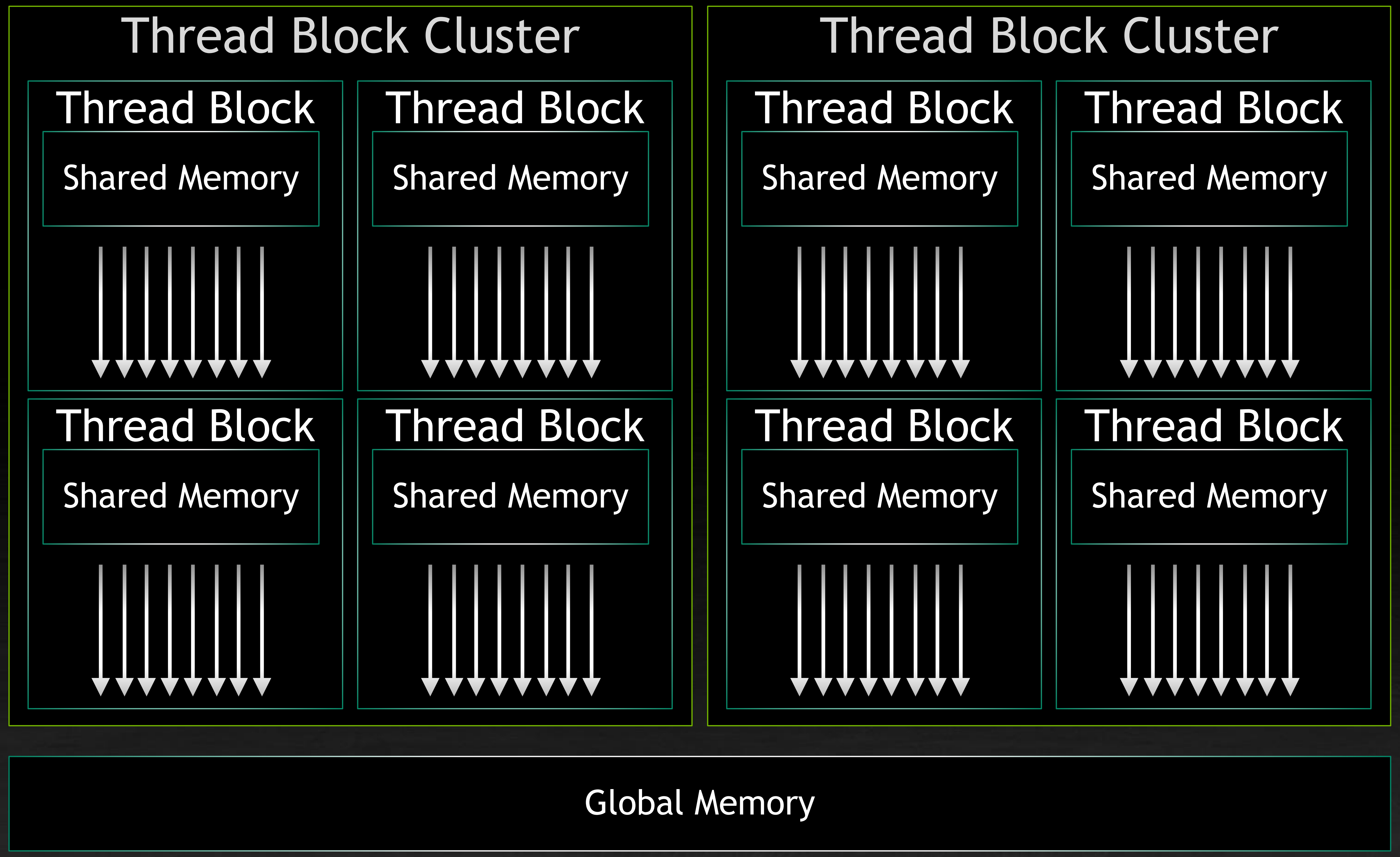
Annotate kernels with compile time cluster size

Kernel launch done in classical way <<< , >>>

```
// Compile time: Kernel where each kernel is
// 2 Thread Blocks in X-dimension and 2 in Y-dimension.

// Requires number of thread blocks to be multiple of 4
__global__ void __cluster_dims__(2, 2, 1) clusterKernel()
{ ... }
```

THREAD AND MEMORY HIERARCHY

Launching CUDA Kernels with Clusters



Using CUDA Extensible Kernel Launch API

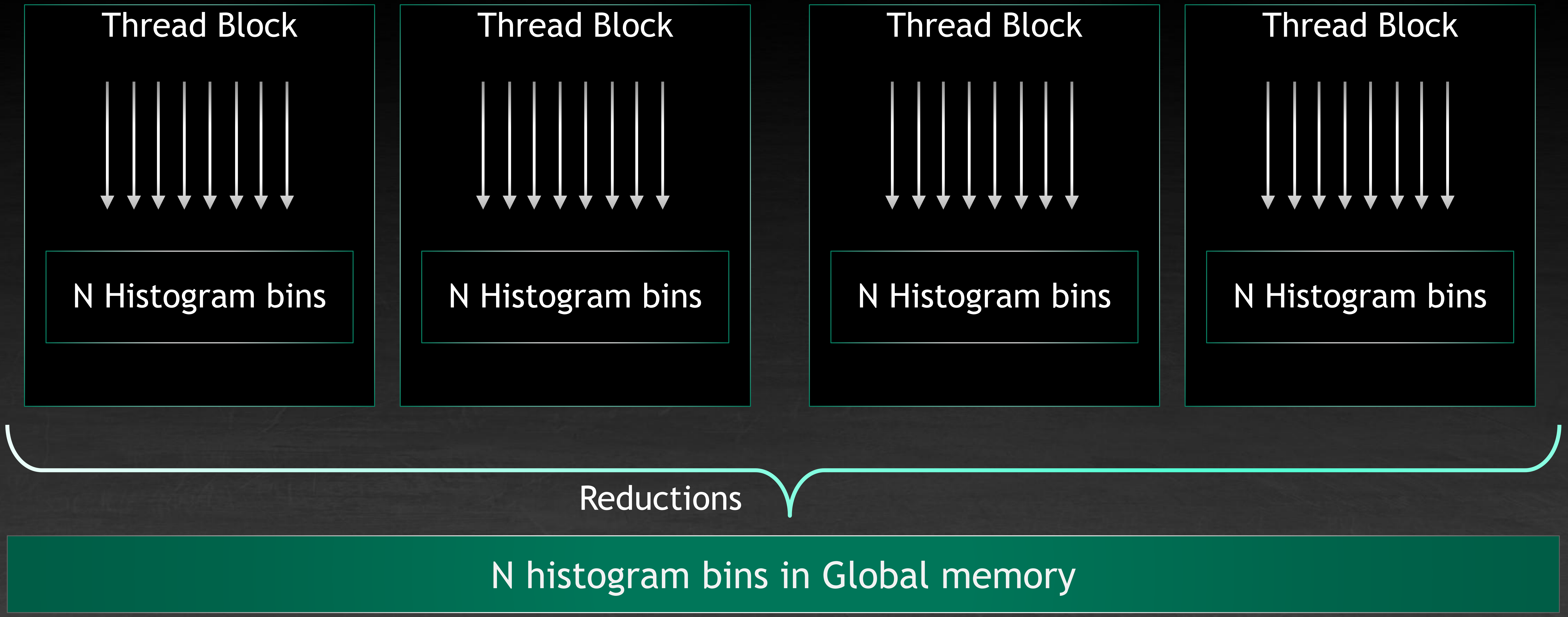
```
// Launch via extensible launch API
{
  cudaLaunchConfig_t config = {0};

  cudaLaunchAttribute attribute[1];
  attribute[0].id = cudaLaunchAttributeClusterDimension;
  attribute[0].val.clusterDim.x = 2; // 2 blocks in X
  attribute[0].val.clusterDim.y = 2; // 2 blocks in Y
  attribute[0].val.clusterDim.z = 1;
  config.attrs = attribute;
  config.numAttrs = 1;

  const int clusterSize  = 2 * 2;
  config.gridDim = numClusters * clusterSize;
  config.blockDim = numThreads;

  void *params[] = {…};
  cudaLaunchKernelEx(&config, (void*)clusterKernel, params);
}
```

Disclaimer: Preliminary CUDA API, subject to change

THREAD AND MEMORY HIERARCHY

Example: Shared Memory Histogram

Histograms in CUDA are usually computed in shared memory and followed by reductions in global memory.



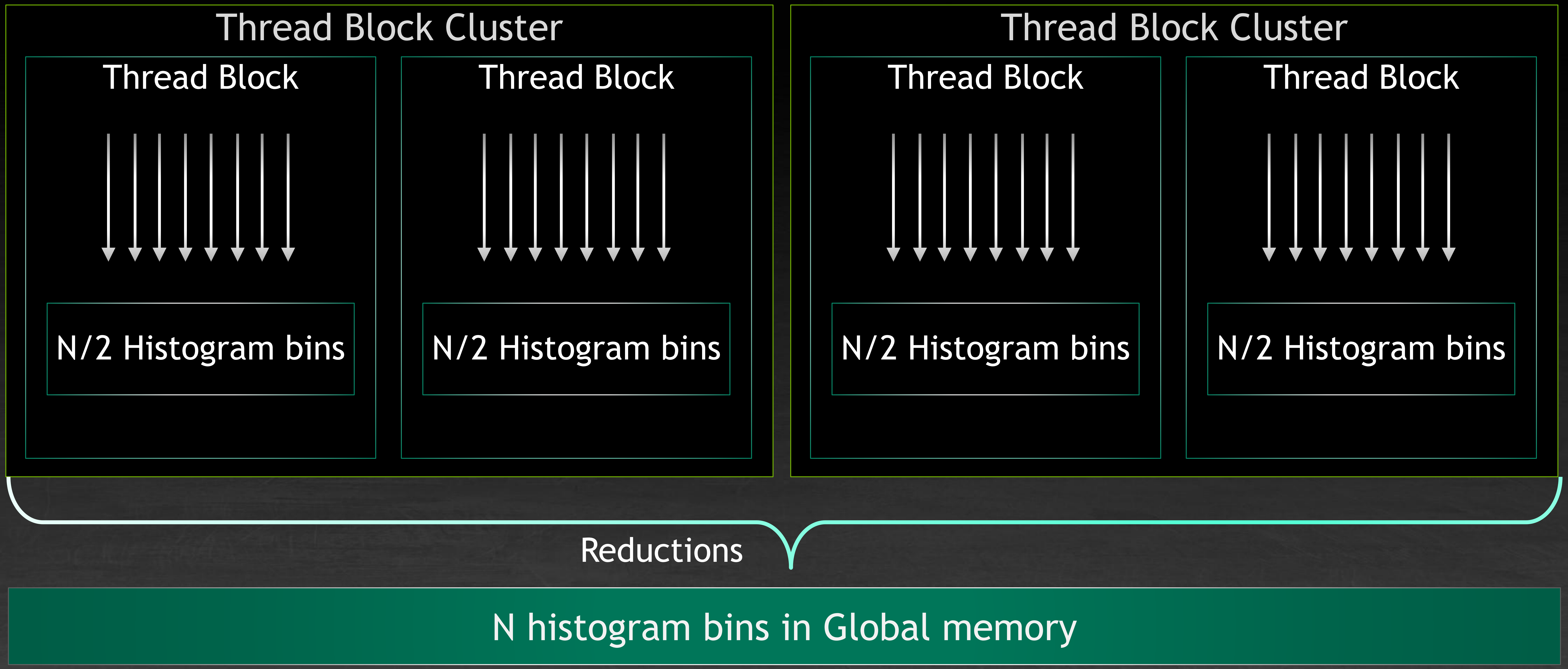
THREAD AND MEMORY HIERARCHY

Example: Distributed Shared Memory Histogram

Histograms in CUDA are usually computed in shared memory and followed by reductions in global memory.

For large histograms, shared memory capacity is not sufficient.
Example: 300KB or 75K integer histogram bins

Distributed shared memory to the rescue.



THREAD AND MEMORY HIERARCHY

Example: Distributed Shared Memory Histogram

```
namespace cg = cooperative_groups;
extern __shared__ int smem[];
cg::cluster_group cluster = cg::this_cluster();
unsigned int cluster_size = cluster.dim_blocks().x;

// Initialize all the pointer to DSMEM
int *sh_hist[cluster_size];
for (int i = 0; i < cluster_size; i++) {
    sh_hist[i] = cluster.map_shared_rank(smem, i);
}
// Initialize Shared memory histogram to zero
for (int i = threadIdx.x; i < bins_per_block; i += blockDim.x) {
    smem[i] = 0;
}
cluster.sync();
```

THREAD AND MEMORY HIERARCHY

Example: Distributed Shared Memory Histogram

```
namespace cg = cooperative_groups;
extern __shared__ int smem[];
cg::cluster_group cluster = cg::this_cluster();
unsigned int cluster_size = cluster.dim_blocks().x;

// Initialize all the pointer to DSMEM
int *sh_hist[cluster_size];
for (int i = 0; i < cluster_size; i++) {
    sh_hist[i] = cluster.map_shared_rank(smem, i);
}
// Initialize Shared memory histogram to zero
for (int i = threadIdx.x; i < bins_per_block; i += blockDim.x) {
    smem[i] = 0;
}
cluster.sync();

// Load input data and find histogram binid
<…>

int dst_block_rank = (int)(binid / bins_per_block);
int dst_offset = binid % bins_per_block;
atomicAdd(sh_hist[dst_block_rank] + dst_offset, 1);
cluster.sync();

// Perform Global memory reductions
<…>
```

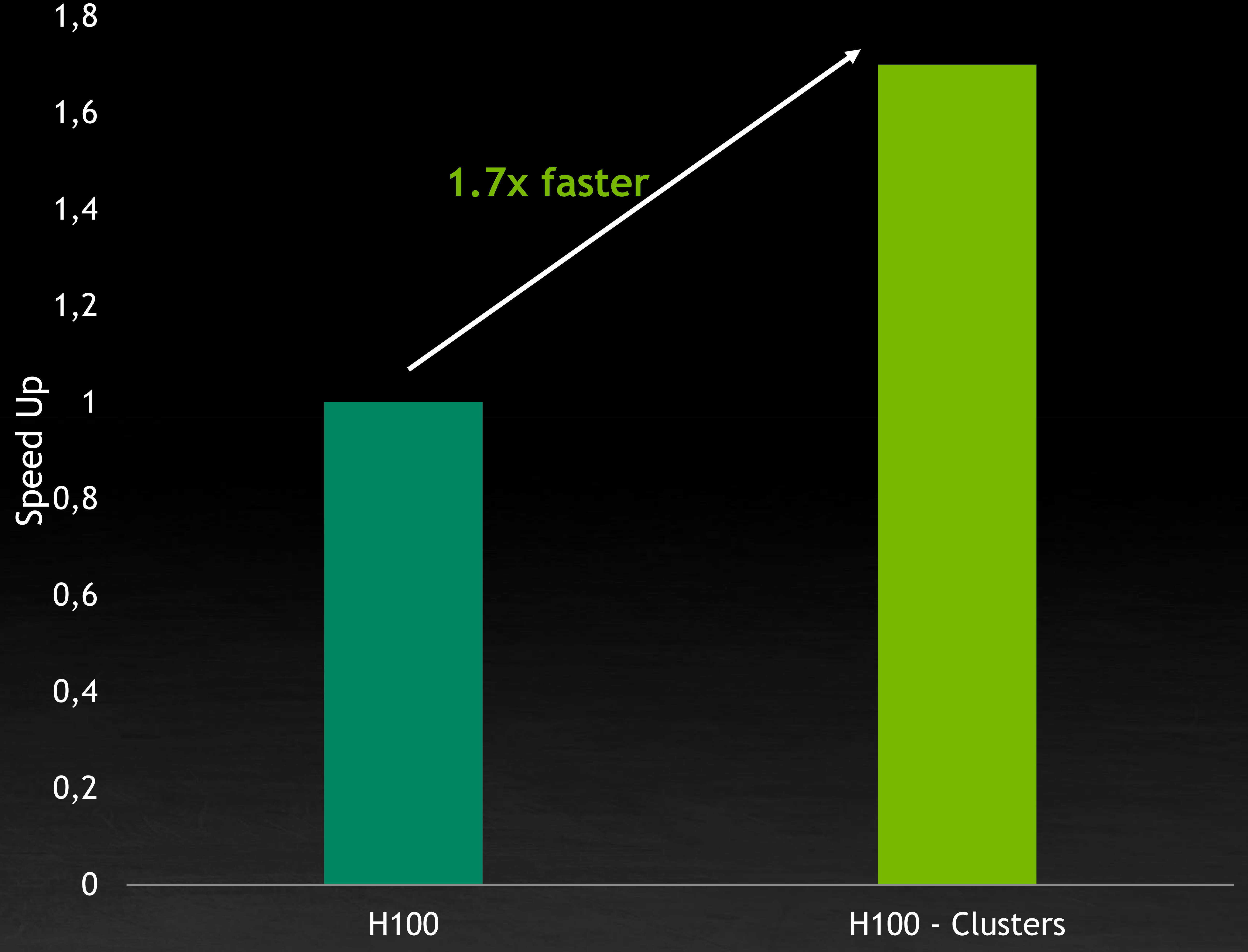
Disclaimer: Preliminary CUDA API, subject to change

THREAD AND MEMORY HIERARCHY

Example: Distributed Shared Memory Histogram

```
namespace cg = cooperative_groups;
extern __shared__ int smem[];
cg::cluster_group cluster = cg::this_cluster();
unsigned int cluster_size = cluster.dim_blocks().x;

// Initialize all the pointer to DSMEM
int *sh_hist[cluster_size];
for (int i = 0; i < cluster_size; i++) {
    sh_hist[i] = cluster.map_shared_rank(smem, i);
}
// Initialize Shared memory histogram to zero
for (int i = threadIdx.x; i < bins_per_block; i += blockDim.x) {
    smem[i] = 0;
}
cluster.sync();

// Load input data and find histogram binid
<…>

int dst_block_rank = (int)(binid / bins_per_block);
int dst_offset = binid % bins_per_block;
atomicAdd(sh_hist[dst_block_rank] + dst_offset, 1);
cluster.sync();

// Perform Global memory reductions
<…>
```

Histogram Performance



75K Histogram bins (300KB) fit in distributed shared memory of 2-block cluster → 37.5K (150KB) per thread block

Disclaimer: Preliminary CUDA API, subject to change

NVIDIA®