NVIDIA

CSCI 5451

Jeremy Iverson | April 20, 2023

Your experience with CUDA

Agenda

- About me
- HW/SW co-design at NVIDIA
- GPU occupancy
- Thread block clusters

About me

2021 –

2016 – 2021

2010 – 2016

About me

2021 –

2016 – 2021

2010 – 2016

About me

2021 –

2016 – 2021

2010 – 2016

About me

2021 –

NVIDIA

2016 – 2021

College of Saint Benedict Saint John's University

2010 – 2016

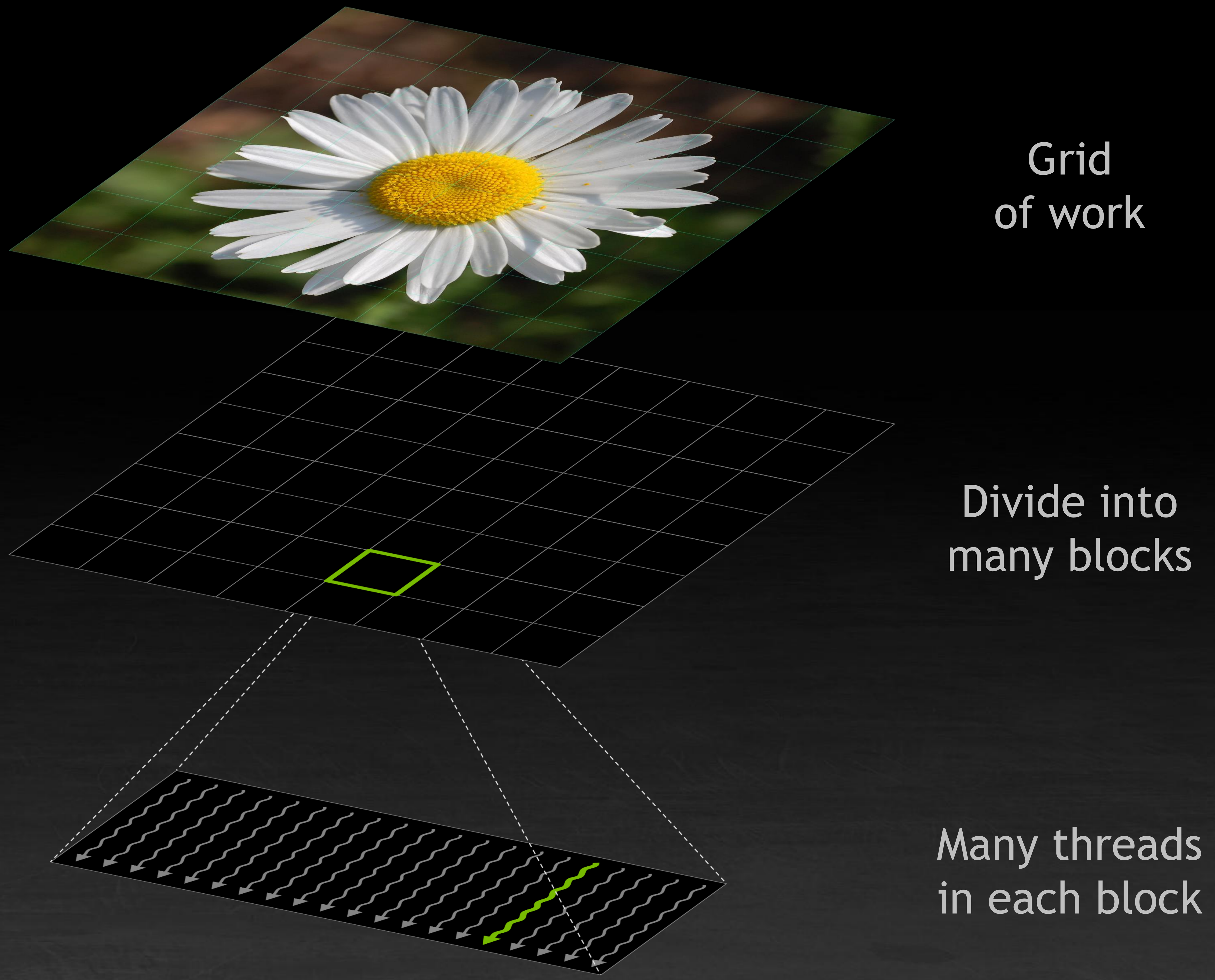
HW/SW Co-design

- Different uses of GPUs: HPC, AI, Graphics, ProViz
- Different needs drive different requirements from HW
- HW innovates based on perceived needs of usages
- SW gathers feedback and characterizes customer needs
- SW considers how the innovation might address open problems
- SW considers how the innovate features will be exposed to users

WHY ~~HOW~~ CUDA PROGRAMMING ~~WORKS~~ IS THE WAY IT IS

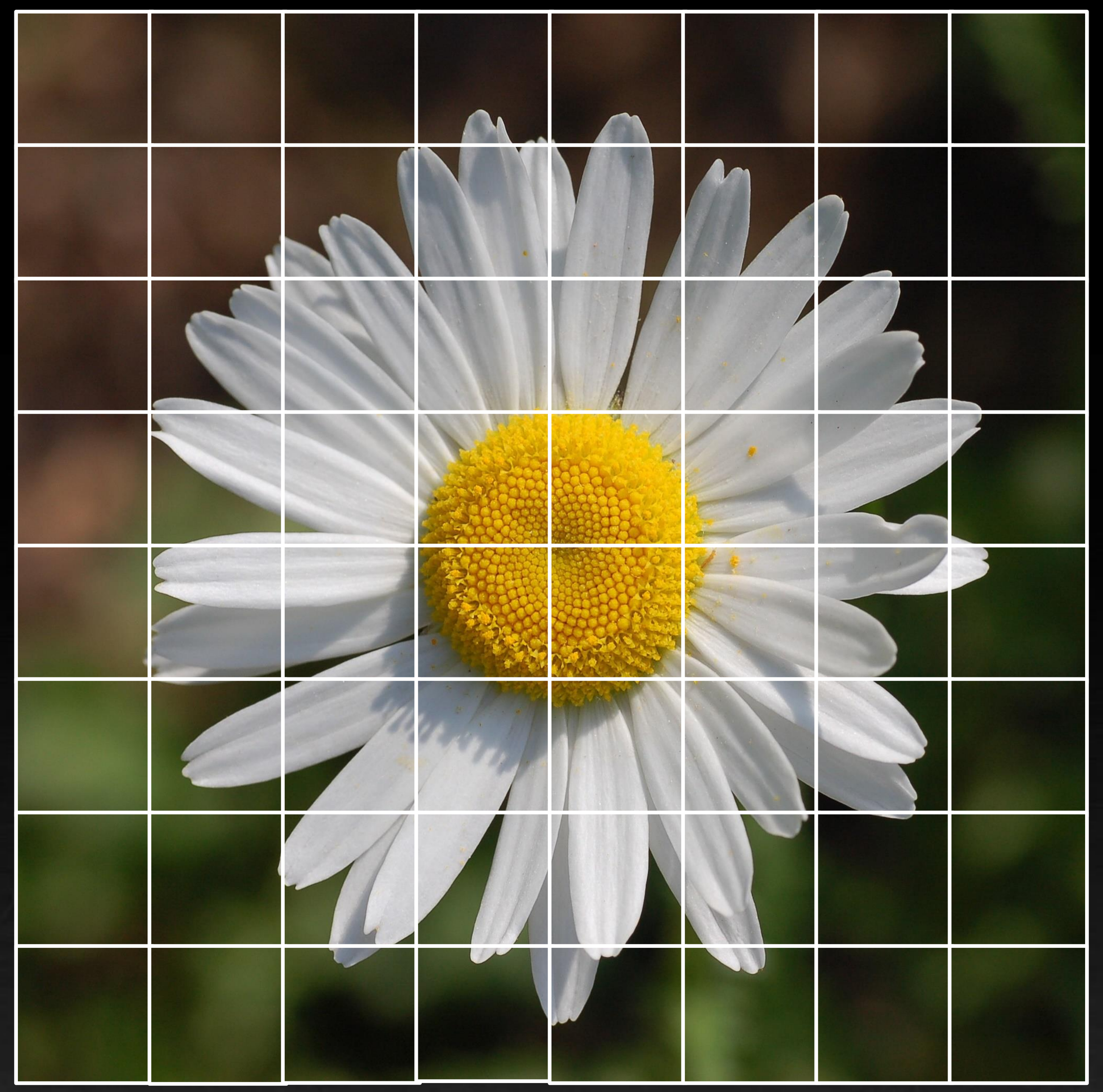
STEPHEN JONES, GTC FALL 2022

CUDA'S GPU EXECUTION HIERARCHY



START WITH SOME WORK TO PROCESS

DIVIDE INTO A SET OF EQUAL-SIZED BLOCKS: THIS IS THE “GRID” OF WORK

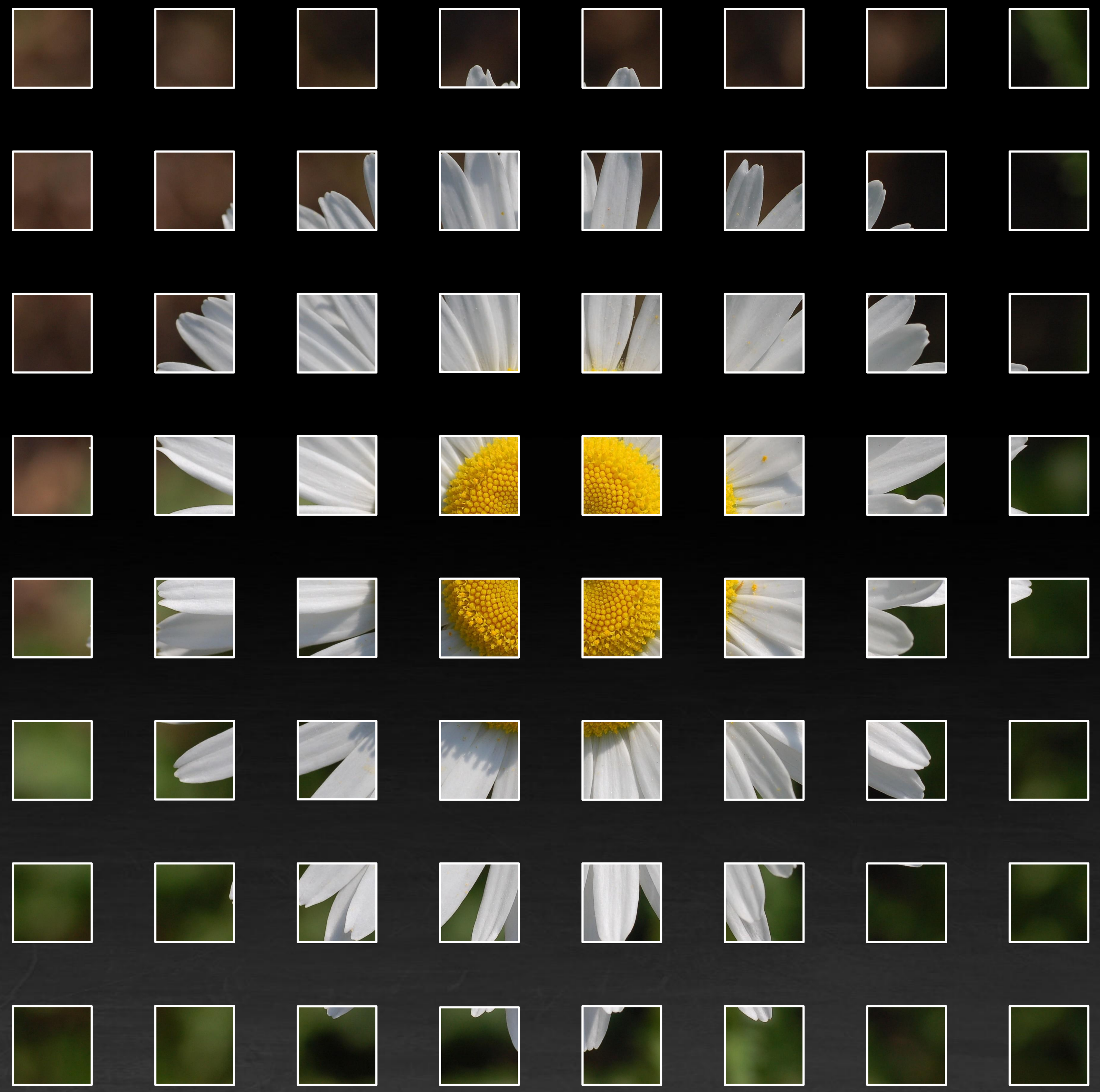


EACH BLOCK WILL NOW BE PROCESSED INDEPENDENTLY

CUDA does not guarantee the order of execution and you cannot exchange data between blocks

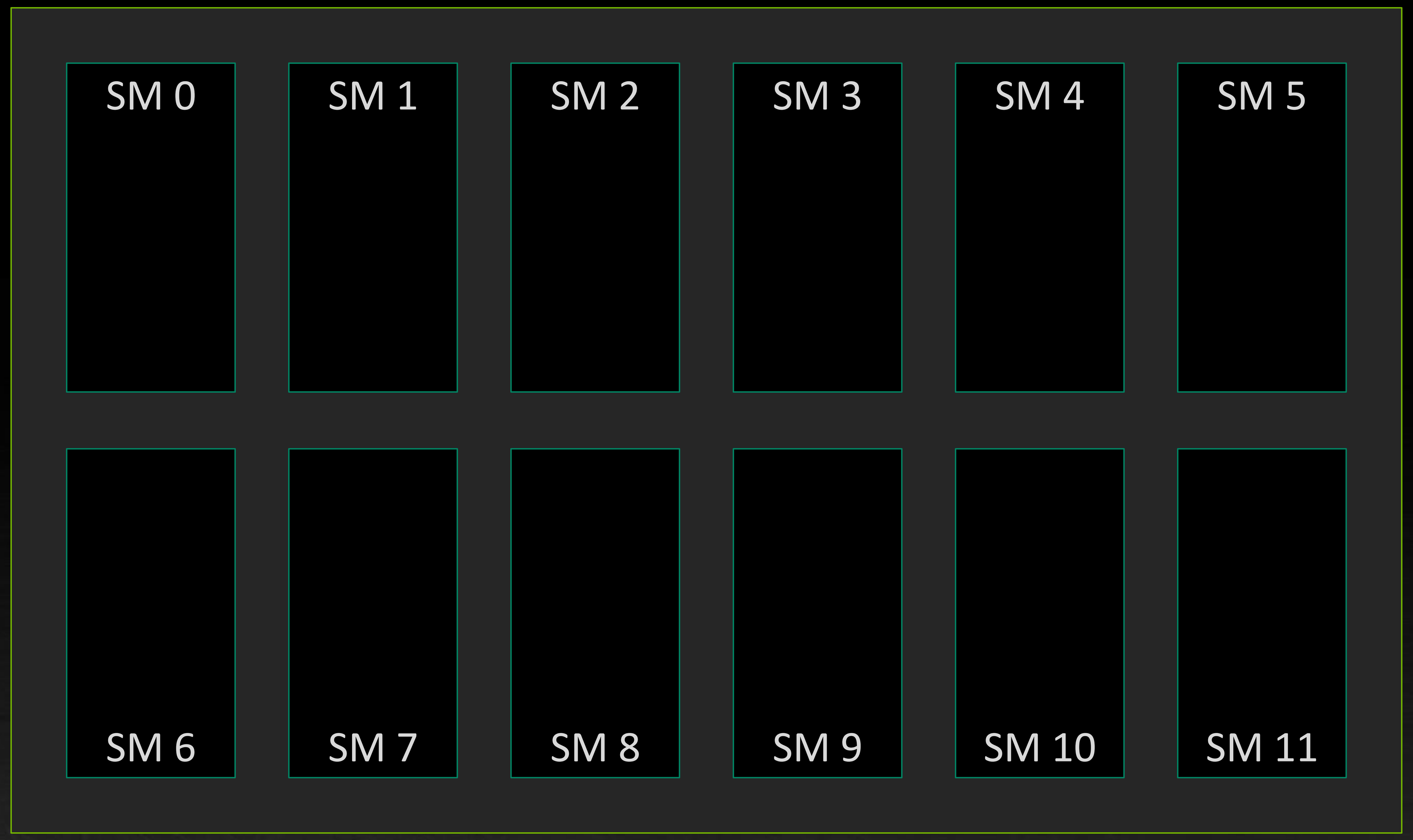
EACH BLOCK WILL NOW BE PROCESSED INDEPENDENTLY

CUDA does not guarantee the order of execution and you cannot exchange data between blocks



EVERY BLOCK GET PLACED ONTO AN SM

CUDA does not guarantee the order of execution and you cannot exchange data between blocks



EVERY BLOCK GET PLACED ONTO AN SM

CUDA does not guarantee the order of execution and you cannot exchange data between blocks

BLOCKS CONTINUE TO GET PLACED UNTIL EACH SM IS "FULL"

When a block completes its work and exits, a new block is placed in its spot until the whole grid is done

NVIDIA

BLOCKS CONTINUE TO GET PLACED UNTIL EACH SM IS "FULL"

When a block completes its work and exits, a new block is placed in its spot until the whole grid is done

BLOCKS CONTINUE TO GET PLACED UNTIL EACH SM IS “FULL”

When a block completes its work and exits, a new block is placed in its spot until the whole grid is done

BLOCKS CONTINUE TO GET PLACED UNTIL EACH SM IS "FULL"

When a block completes its work and exits, a new block is placed in its spot until the whole grid is done

BLOCKS CONTINUE TO GET PLACED UNTIL EACH SM IS "FULL"

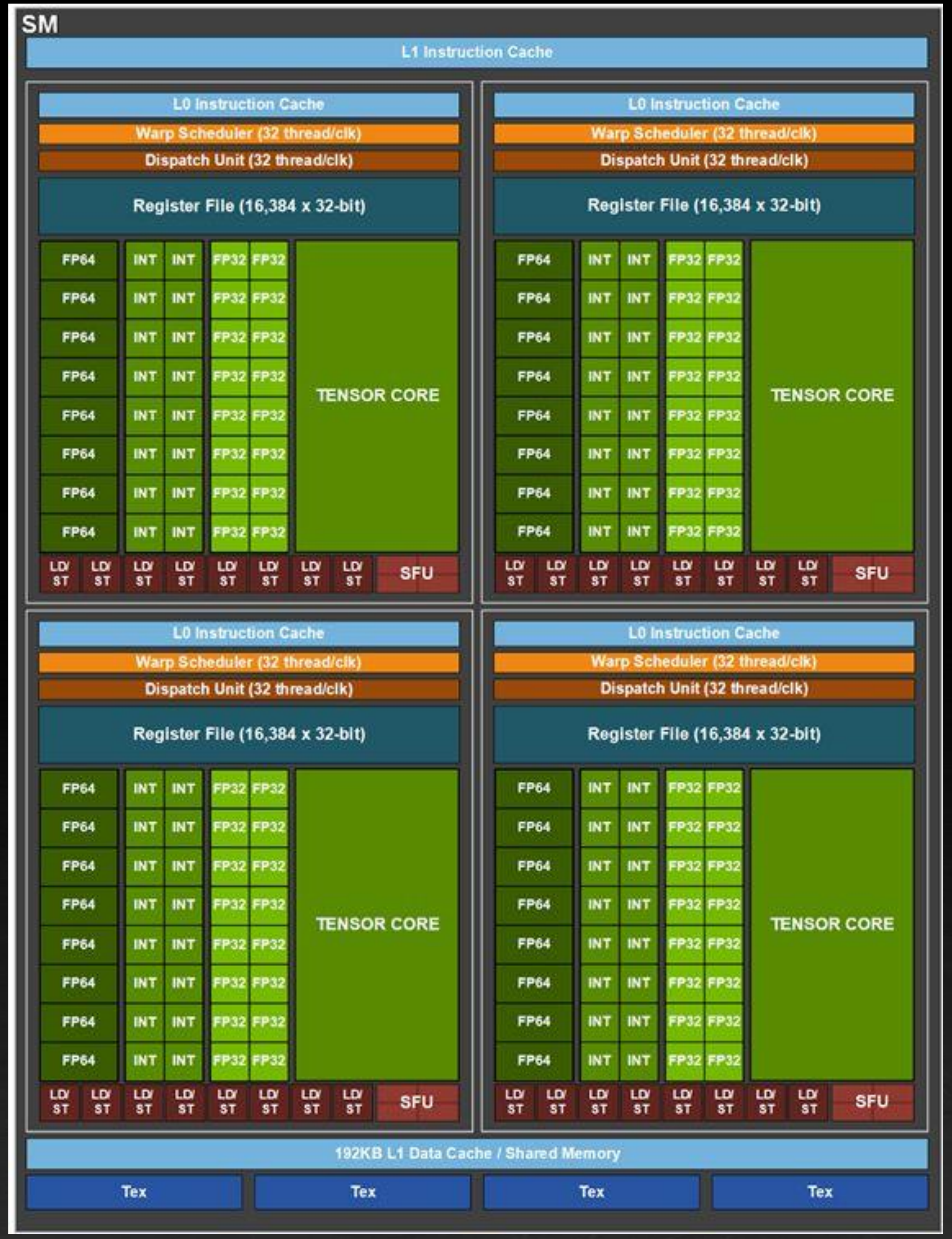
When a block completes its work and exits, a new block is placed in its spot until the whole grid is done

NVIDIA

WHAT DOES IT MEAN FOR AN SM TO BE “FULL”?

NVIDIA

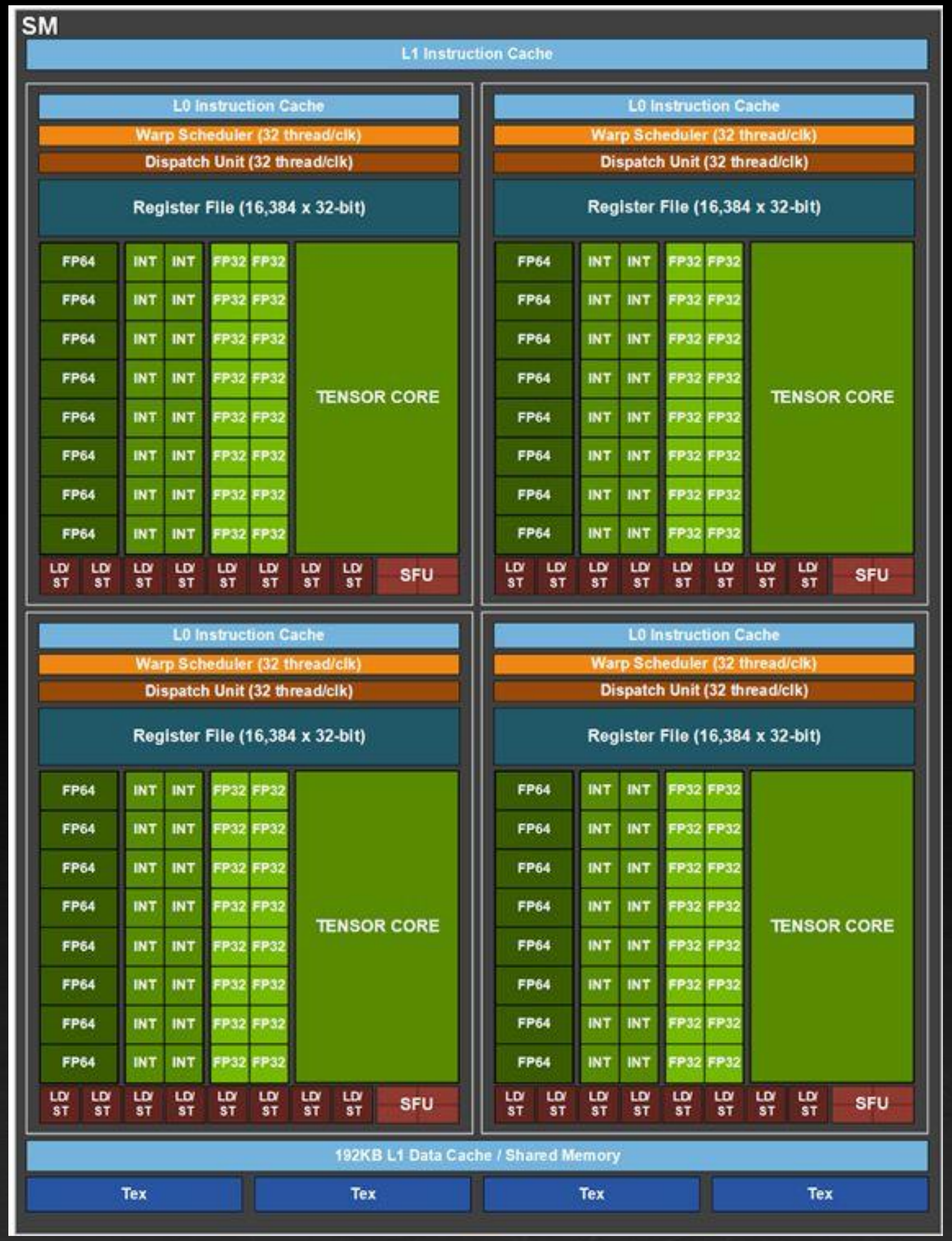
LOOKING INSIDE A STREAMING MULTIPROCESSOR



A100 SM Resources

2048	Max threads per SM
32	Max blocks per SM
65,536	Total registers per SM
160 kB	Total shared memory in SM
32	Threads per warp
4	Concurrent warps active
64	FP32 cores per SM
32	FP64 cores per SM
192 kB	Max L1 cache size
90 GB/sec	Load bandwidth per SM
1410 MHz	GPU Boost Clock

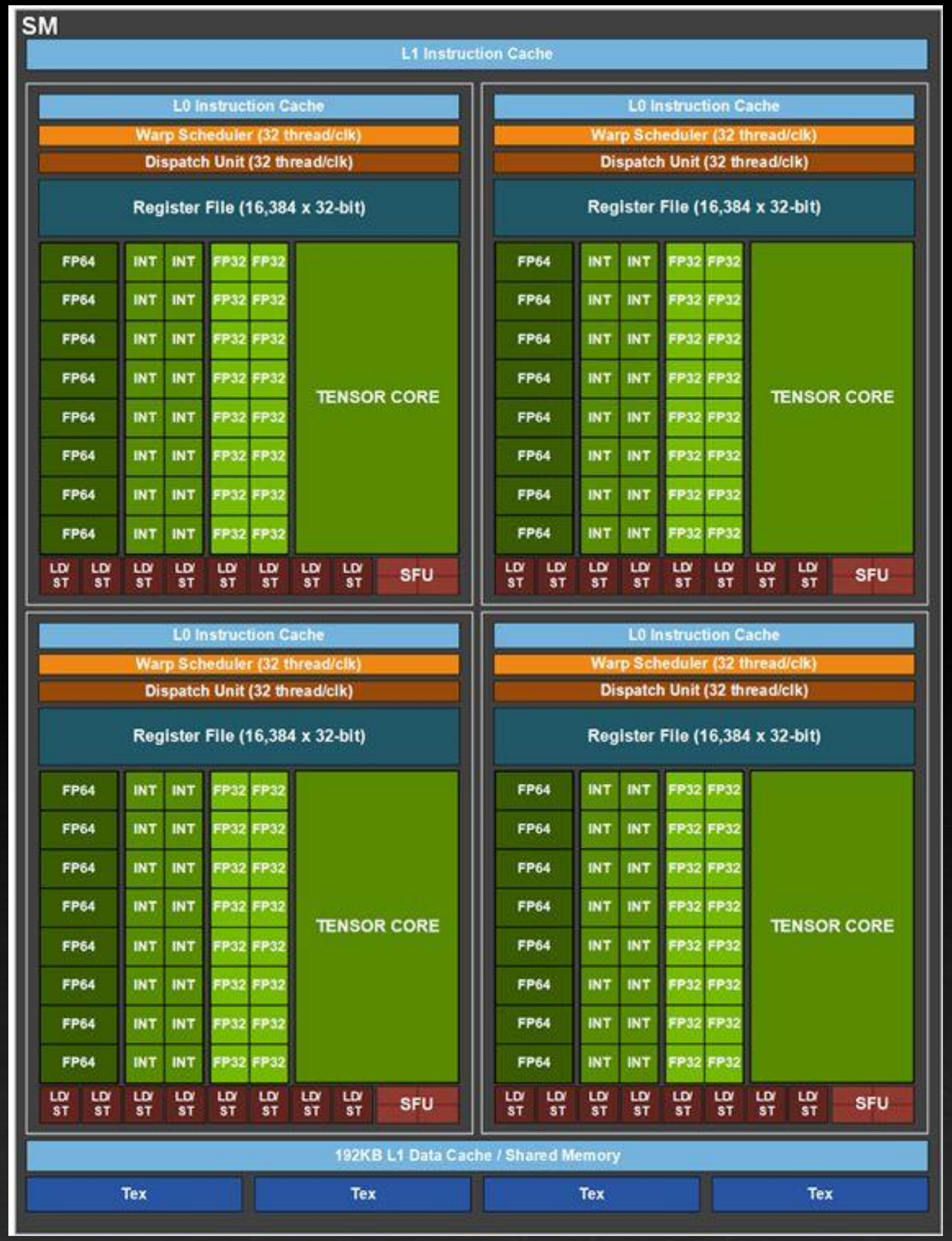
LOOKING INSIDE A STREAMING MULTIPROCESSOR



A100 SM Resources

2048	Max threads per SM
32	Max blocks per SM
65,536	Total registers per SM
160 kB	Total shared memory in SM
32	Threads per warp
4	Concurrent warps active
64	FP32 cores per SM
32	FP64 cores per SM
192 kB	Max L1 cache size
90 GB/sec	Load bandwidth per SM
1410 MHz	GPU Boost Clock

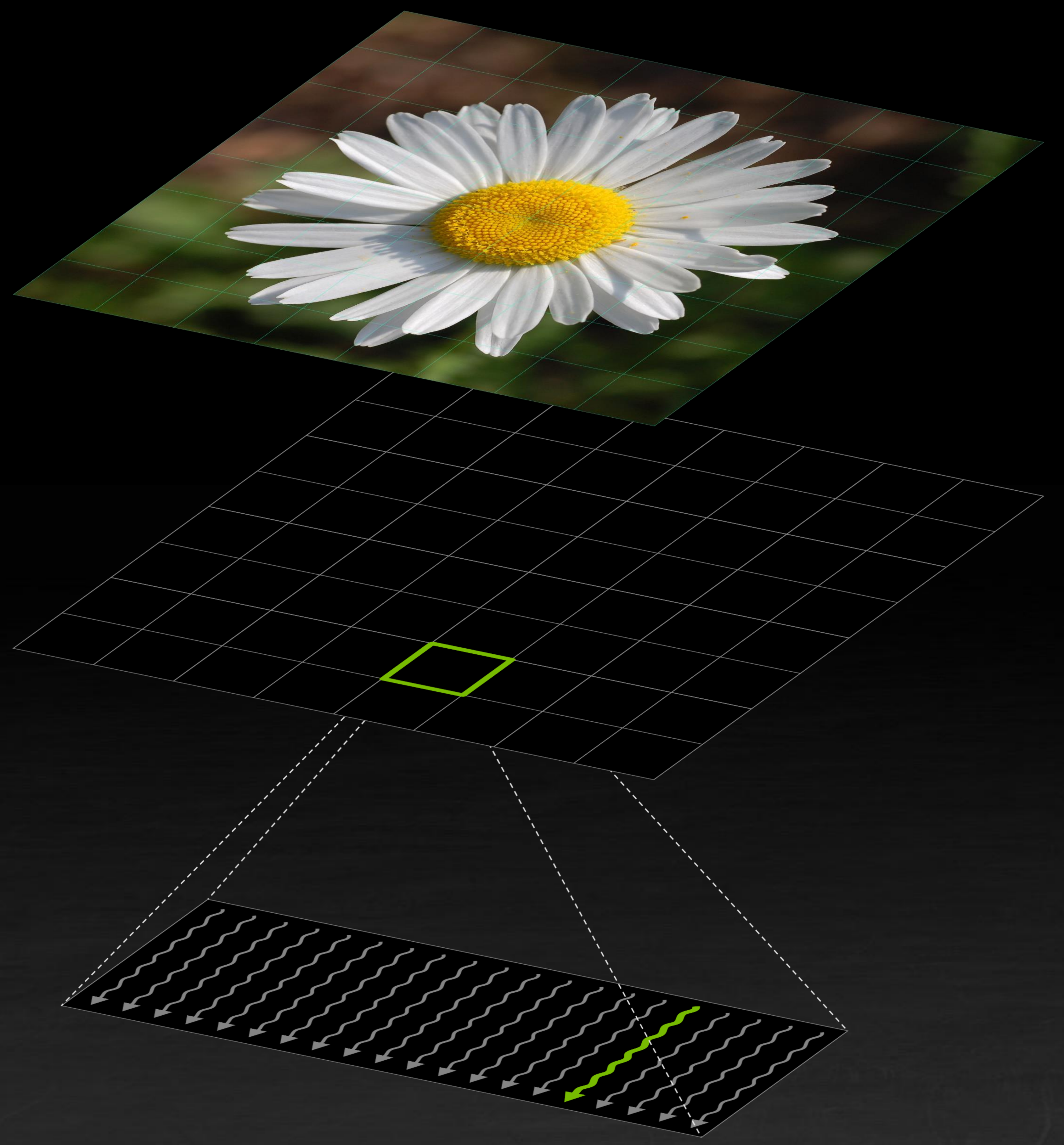
LOOKING INSIDE A STREAMING MULTIPROCESSOR



A100 SM Resources

2048	Max threads per SM
~~32~~	~~Max blocks per SM~~
65,536	Total registers per SM
160 kB	Total shared memory in SM
32	Threads per warp
4	Concurrent warps active
64	FP32 cores per SM
32	FP64 cores per SM
192 kB	Max L1 cache size
90 GB/sec	Load bandwidth per SM
1410 MHz	GPU Boost Clock

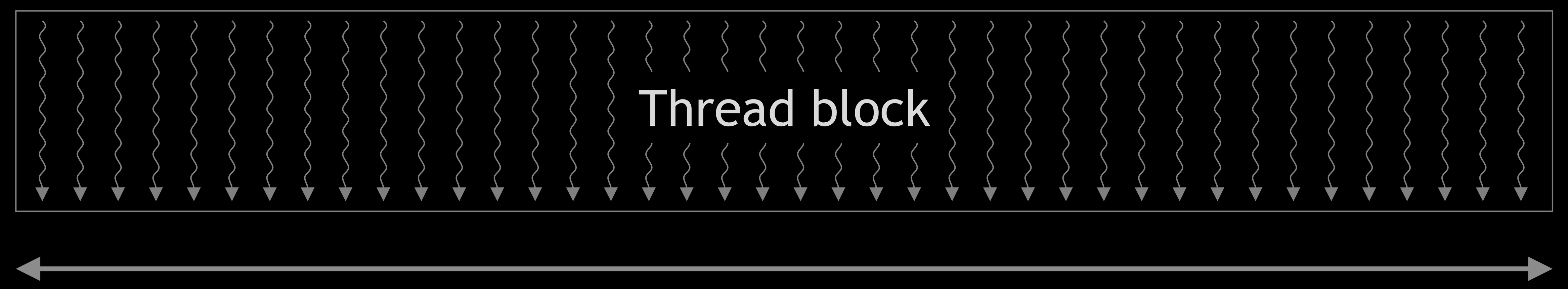
THE CUDA PROGRAMMING MODEL



Grid
of work

Divide into
many blocks

Many threads
in each block



A block has a fixed number of threads

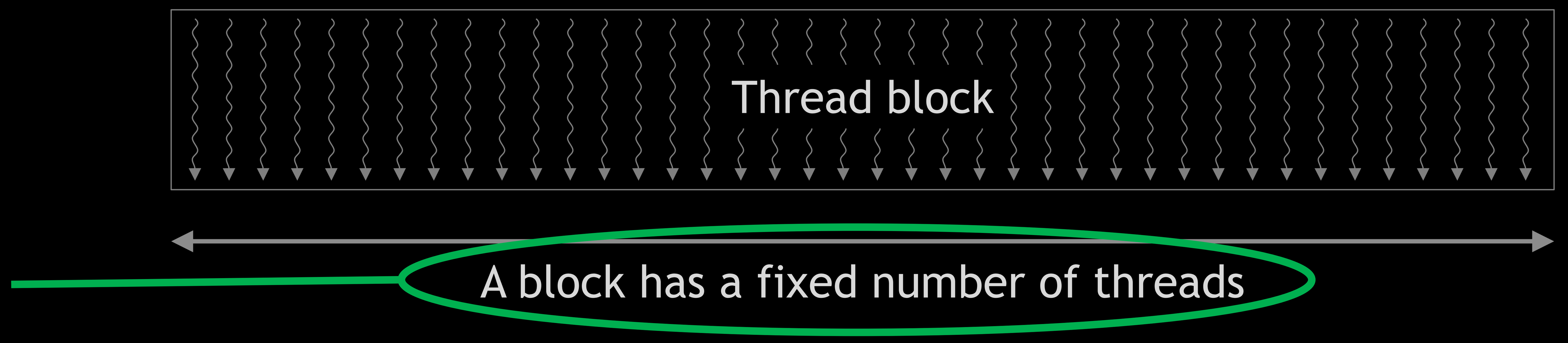
```
__shared__ float mean = 0.0f;
__device__ float mean_euclidian_distance(float2 *p1, float2 *p2) {
    // Compute the Euclidian distance between two points
    float2 dp = p2[threadIdx.x] - p1[threadIdx.x];
    float dist = sqrtf(dp.x * dp.x + dp.y * dp.y);

    // Accumulate the mean distance atomically and return distance
    atomicAdd(&mean, dist / blockDim.x);
    return dist;
}
```

Every thread runs exactly the same program
(this is the “SIMT” model)

ANATOMY OF A THREAD BLOCK

All blocks in a grid run the same program using the same number of threads, leading to 3 resource requirements

1. Block size – the number of threads which must be concurrent




```
__shared__ float mean = 0.0f;
__device__ float mean_euclidian_distance(float2 *p1, float2 *p2) {
    // Compute the Euclidian distance between two points
    float2 dp = p2[threadIdx.x] - p1[threadIdx.x];
    float dist = sqrtf(dp.x * dp.x + dp.y * dp.y);

    // Accumulate the mean distance atomically and return distance
    atomicAdd(&mean, dist / blockDim.x);
    return dist;
}
```

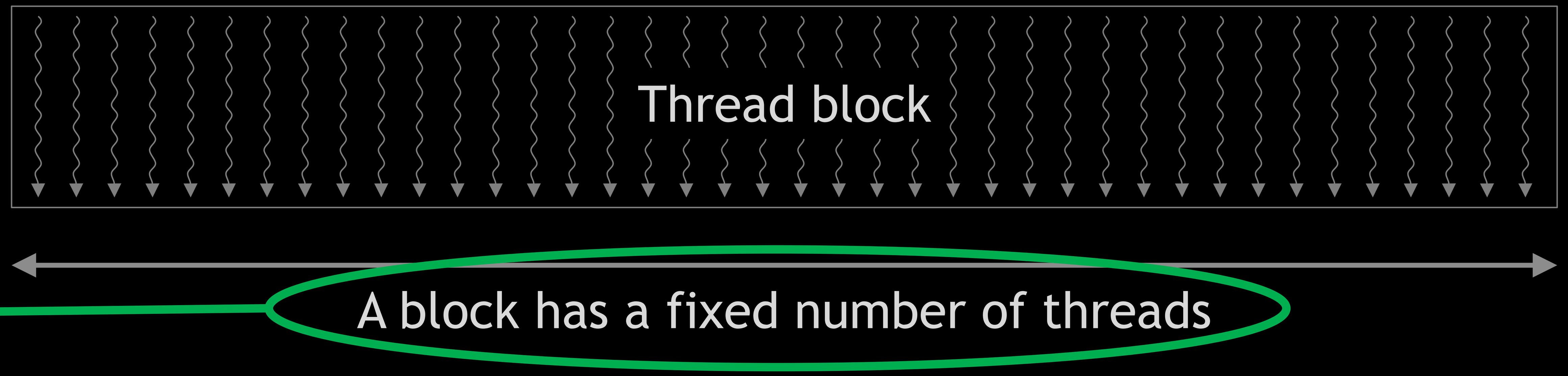
Every thread runs exactly the same program
(this is the "SIMT" model)

ANATOMY OF A THREAD BLOCK

All blocks in a grid run the same program using the same number of threads, leading to 3 resource requirements

1. Block size - the number of threads which must be concurrent

2. Shared memory - common to all threads in a block

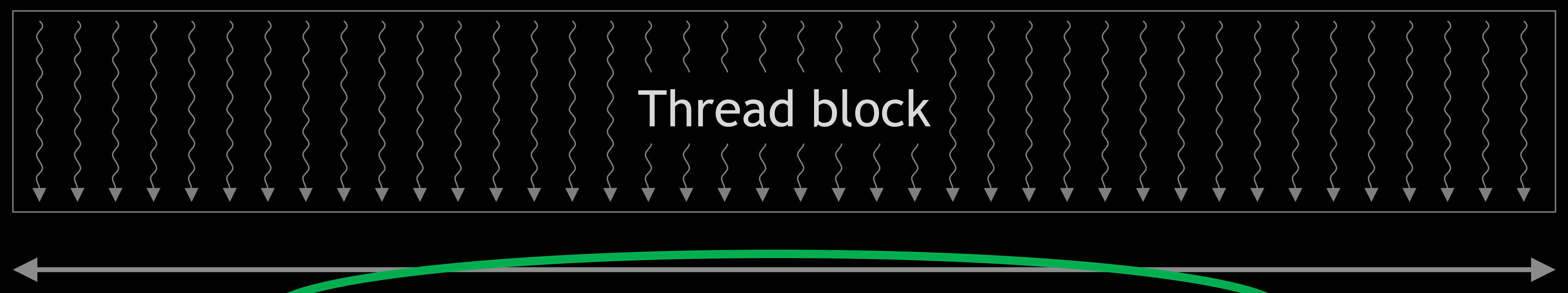



```
__shared__ float mean = 0.0f;
__device__ float mean_euclidian_distance(float2 *p1, float2 *p2) {
    // Compute the Euclidian distance between two points
    float2 dp = p2[threadIdx.x] - p1[threadIdx.x];
    float dist = sqrtf(dp.x * dp.x + dp.y * dp.y);

    // Accumulate the mean distance atomically and return distance
    atomicAdd(&mean, dist / blockDim.x);
    return dist;
}
```

Every thread runs exactly the same program
(this is the “SIMT” model)

ANATOMY OF A THREAD BLOCK

All blocks in a grid run the same program using the same number of threads, leading to 3 resource requirements



A block has a fixed number of threads

1. Block size - the number of threads which must be concurrent

2. Shared memory - common to all threads in a block

3. Registers - depends on program complexity

   Registers are a per-thread resource, so total budget is:
   (threads-per-block x registers-per-thread)

```
__shared__ float mean = 0.0f;
__device__ float mean_euclidian_distance(float2 *p1, float2 *p2) {
    // Compute the Euclidian distance between two points
    float2 dp = p2[threadIdx.x] - p1[threadIdx.x];
    float dist = sqrtf(dp.x * dp.x + dp.y * dp.y);

    // Accumulate the mean distance atomically and return distance
    atomicAdd(&mean, dist / blockDim.x);
    return dist;
}
```

Every thread runs exactly the same program
(this is the "SIMT" model)

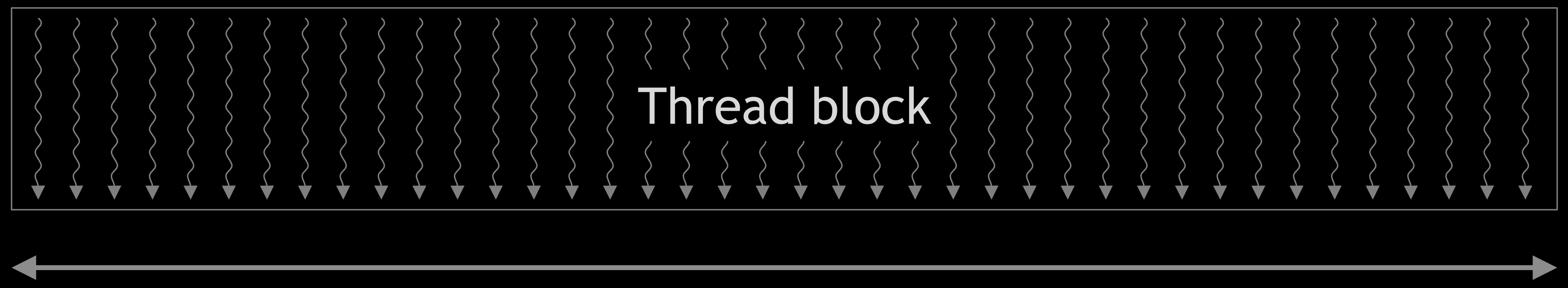
HOW THE GPU PLACES BLOCKS ON AN SM

A100 SM Key Resources

2048 Threads

65536 Registers

160kB Shared Memory

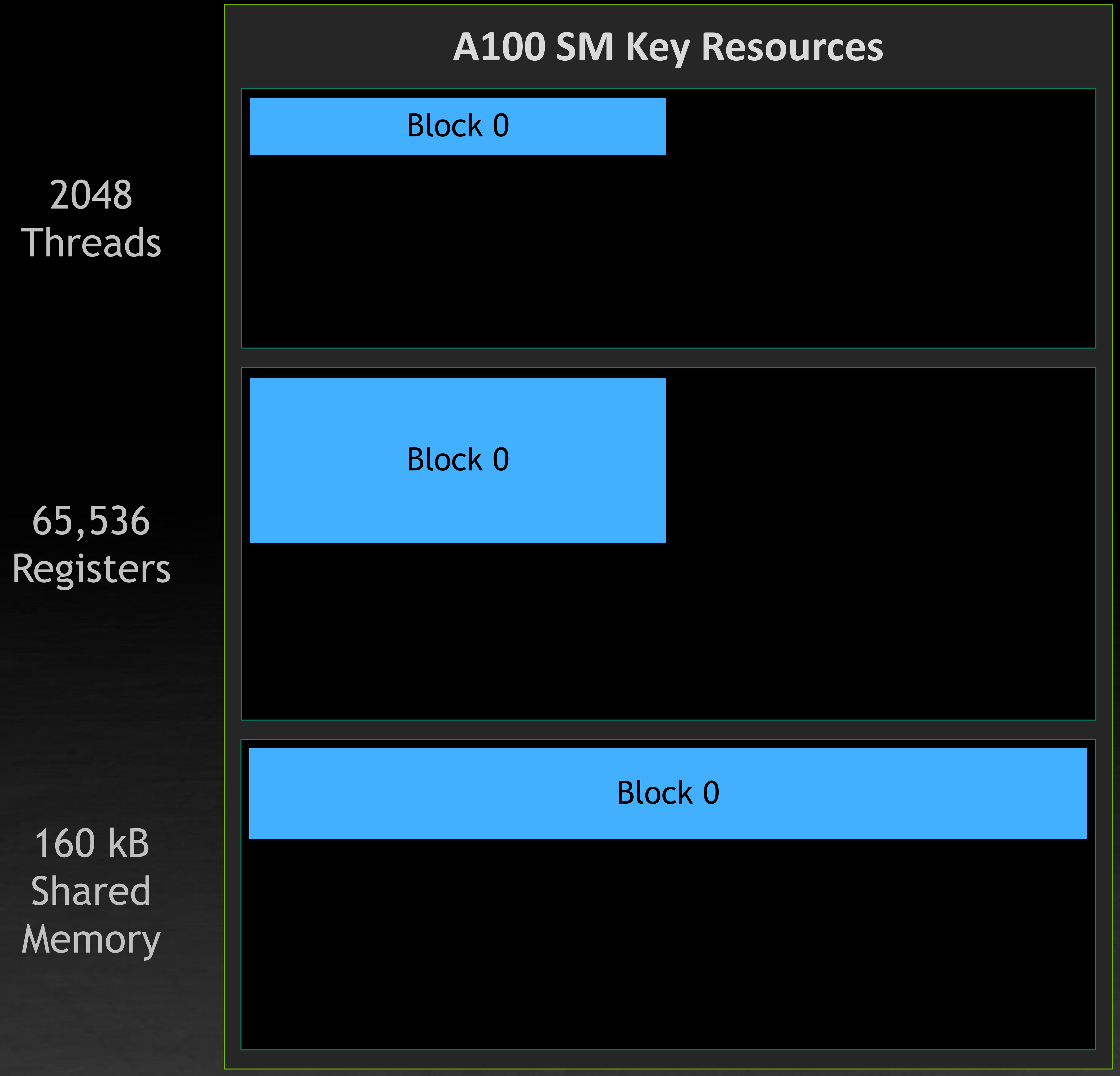


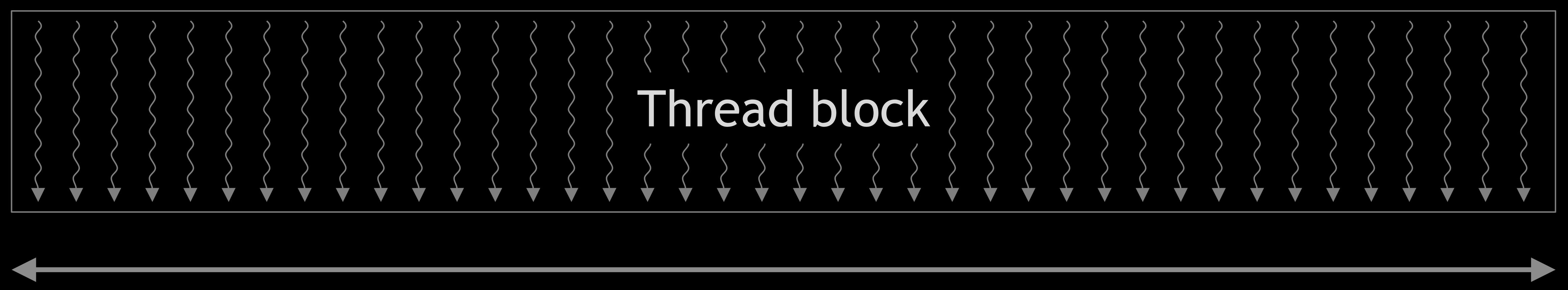
A block has a fixed number of threads
always running on a single SM

Example block resource requirements	
256	Threads per block
64	(Registers per thread)
(256 * 64) = 16384	Registers per block
48 kB	Shared memory per block

NVIDIA

HOW THE GPU PLACES BLOCKS ON AN SM

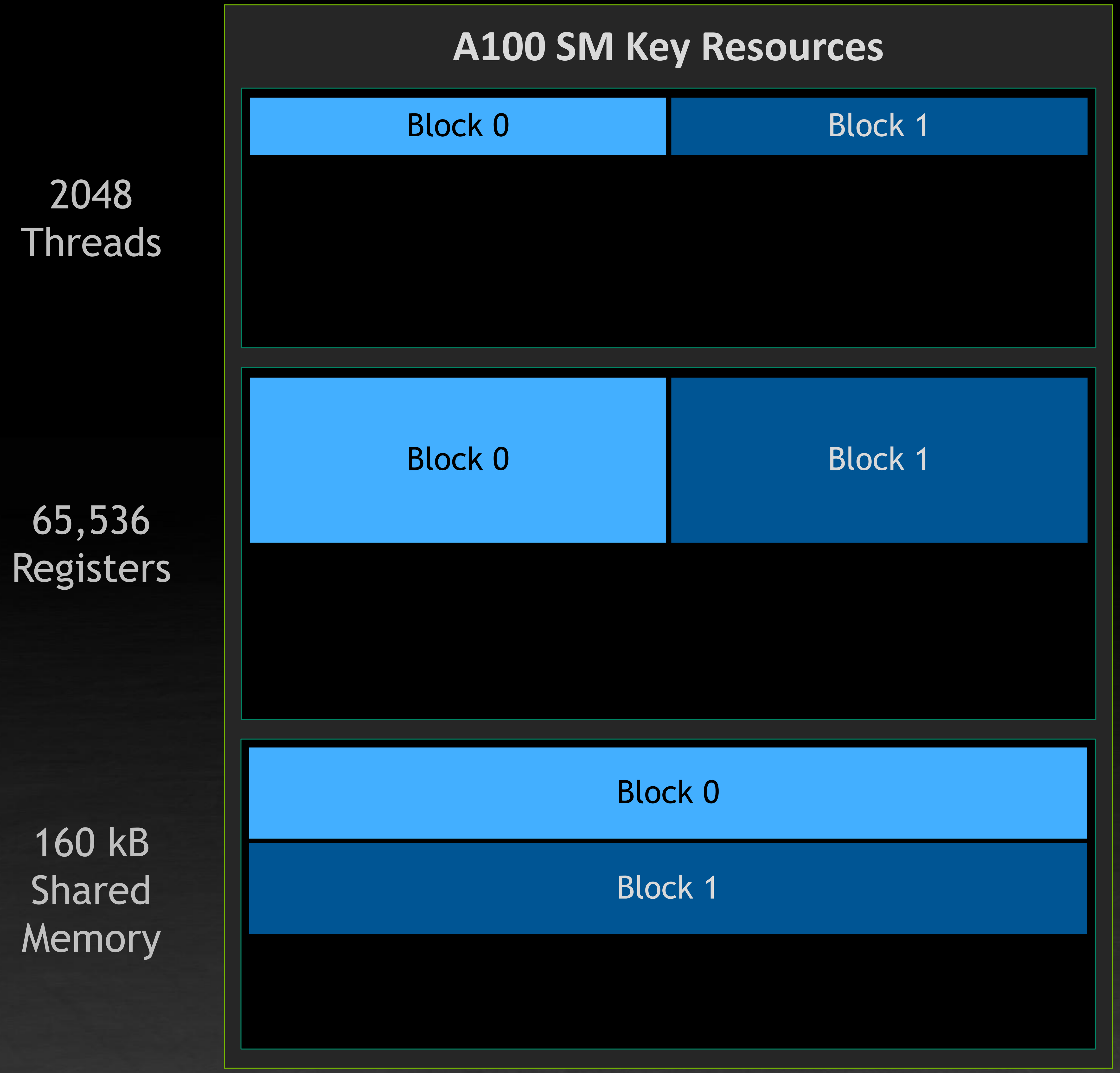


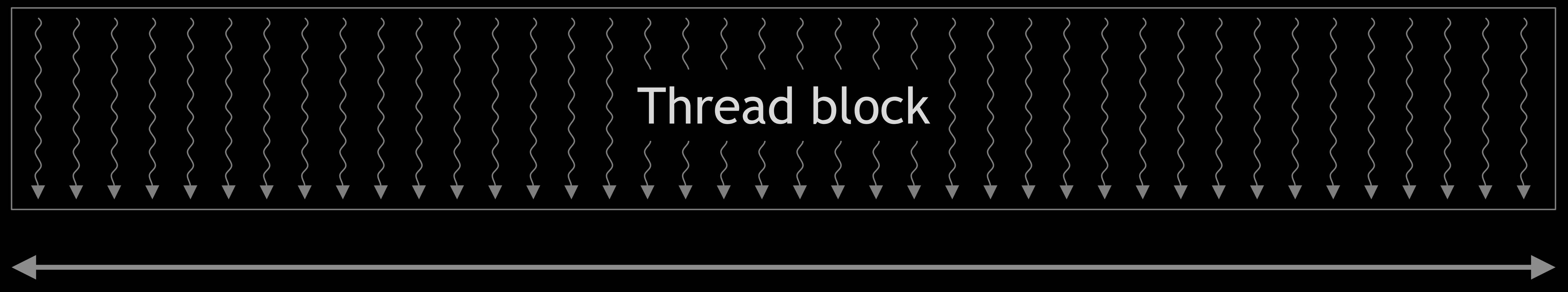


A block has a fixed number of threads
always running on a single SM

Example block resource requirements	
256	Threads per block
64	(Registers per thread)
(256 * 64) = 16384	Registers per block
48 kB	Shared memory per block

HOW THE GPU PLACES BLOCKS ON AN SM

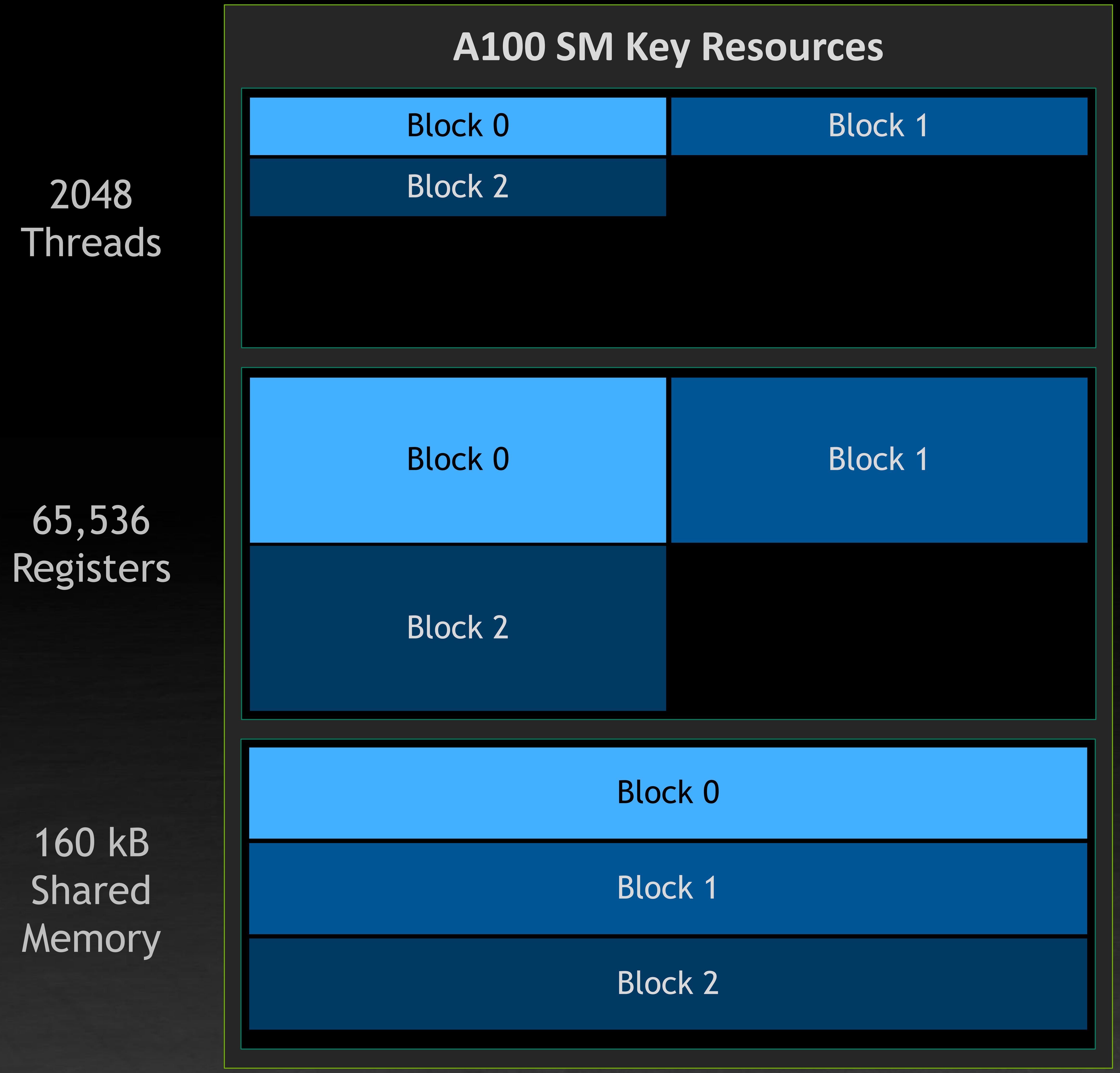


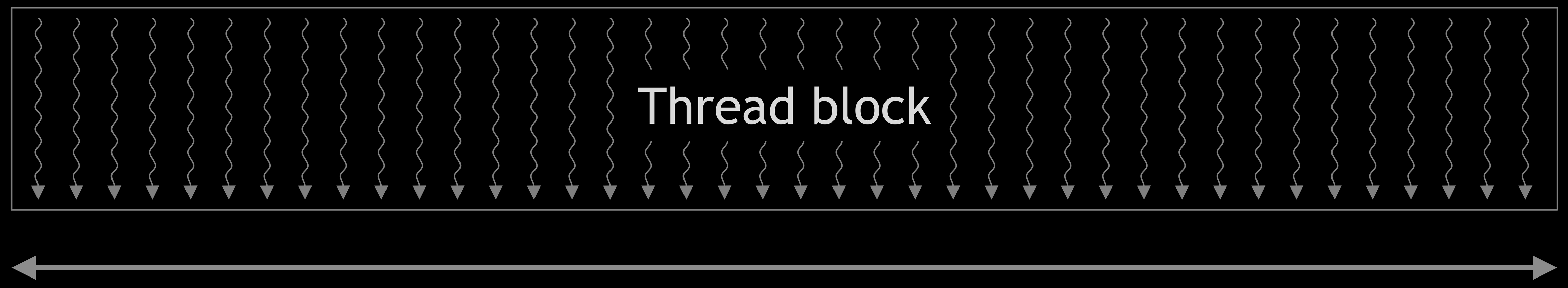


A block has a fixed number of threads always running on a single SM

Example block resource requirements	
256	Threads per block
64	(Registers per thread)
(256 * 64) = 16384	Registers per block
48 kB	Shared memory per block

HOW THE GPU PLACES BLOCKS ON AN SM

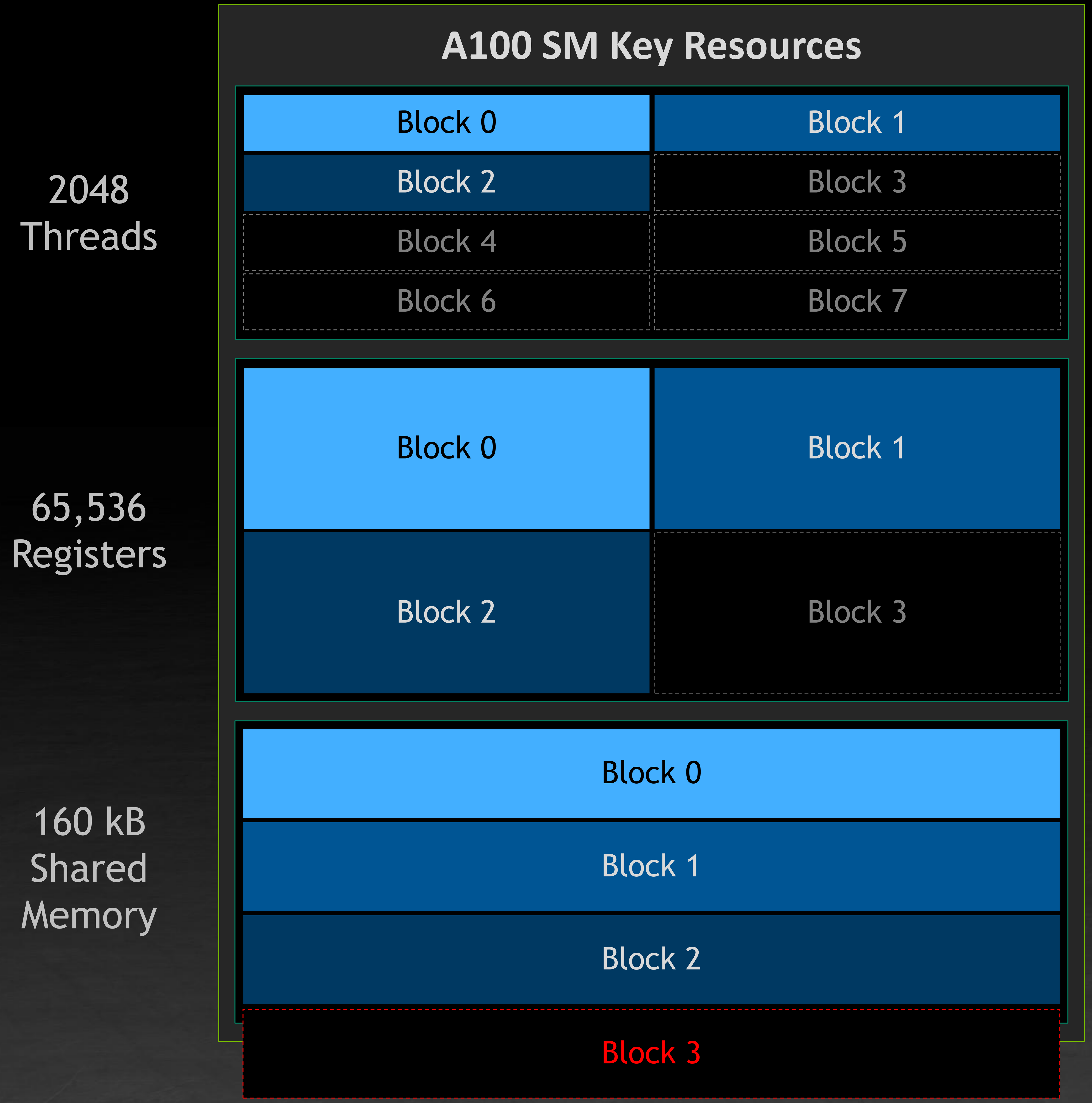


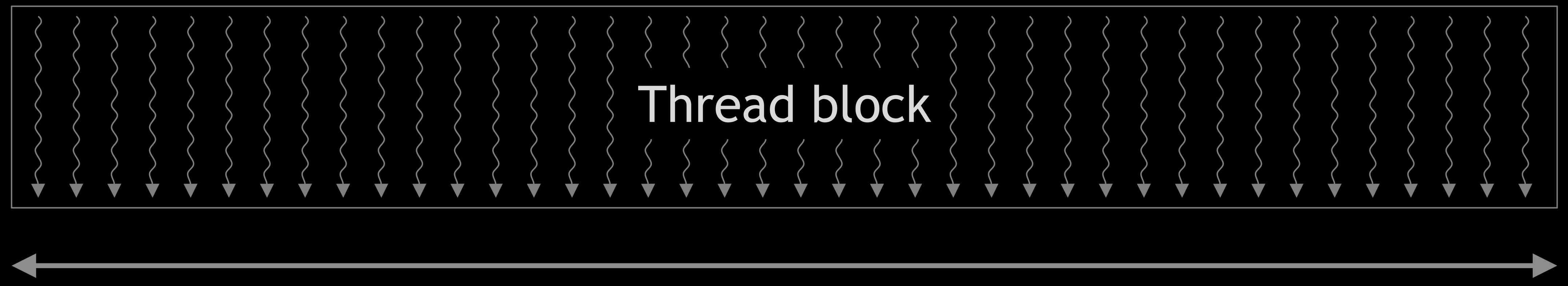


A block has a fixed number of threads always running on a single SM

Example block resource requirements	
256	Threads per block
64	(Registers per thread)
(256 * 64) = 16384	Registers per block
48 kB	Shared memory per block

HOW THE GPU PLACES BLOCKS ON AN SM

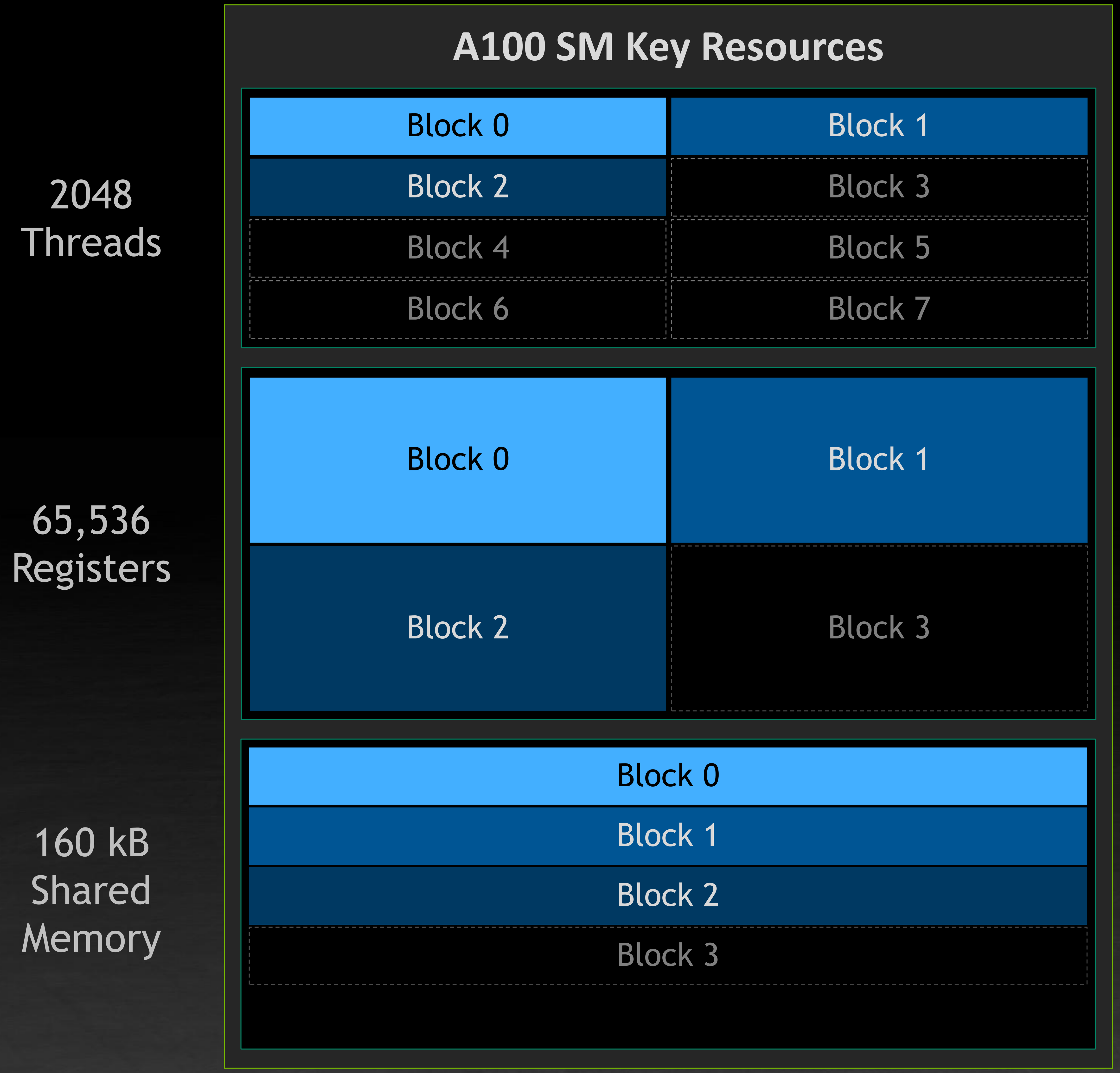


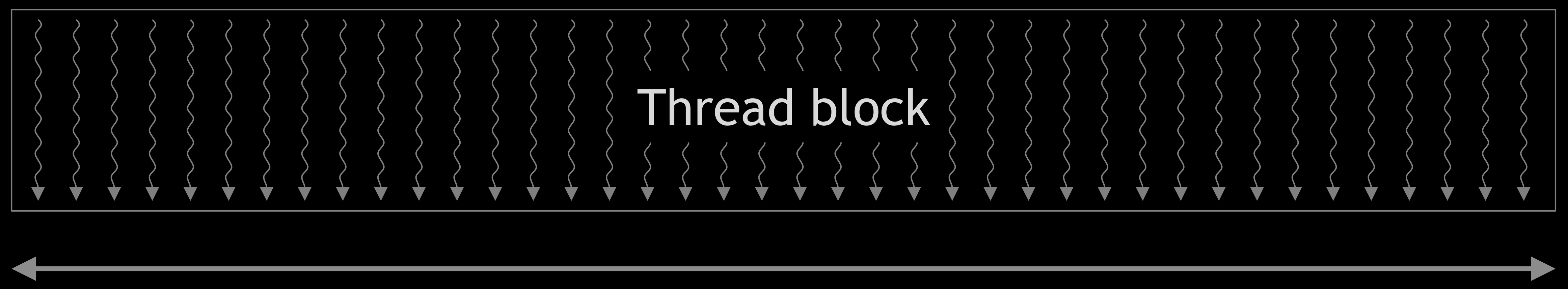


A block has a fixed number of threads always running on a single SM

Example block resource requirements	
256	Threads per block
64	(Registers per thread)
(256 * 64) = 16384	Registers per block
48 kB	Shared memory per block

HOW THE GPU PLACES BLOCKS ON AN SM



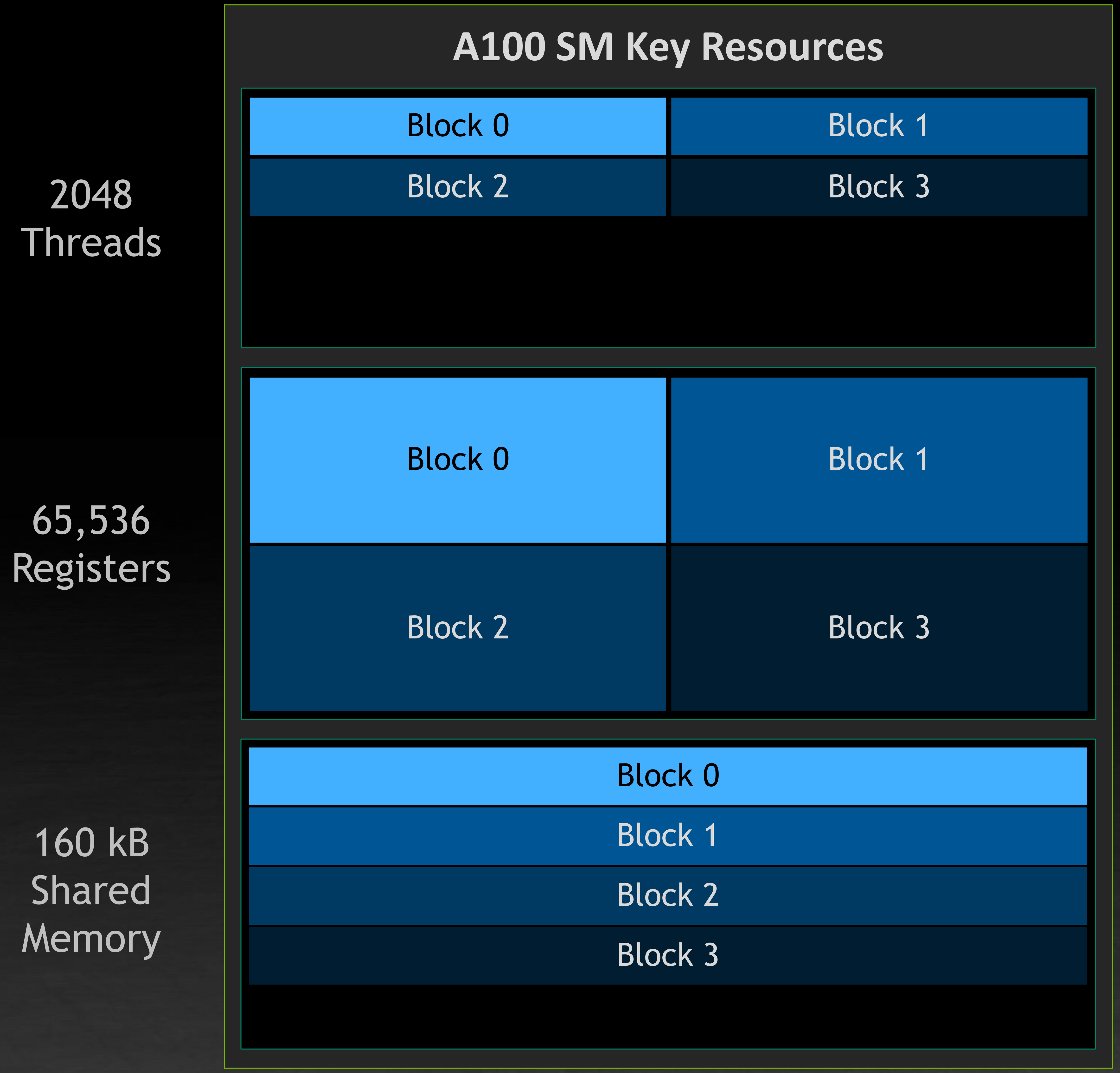


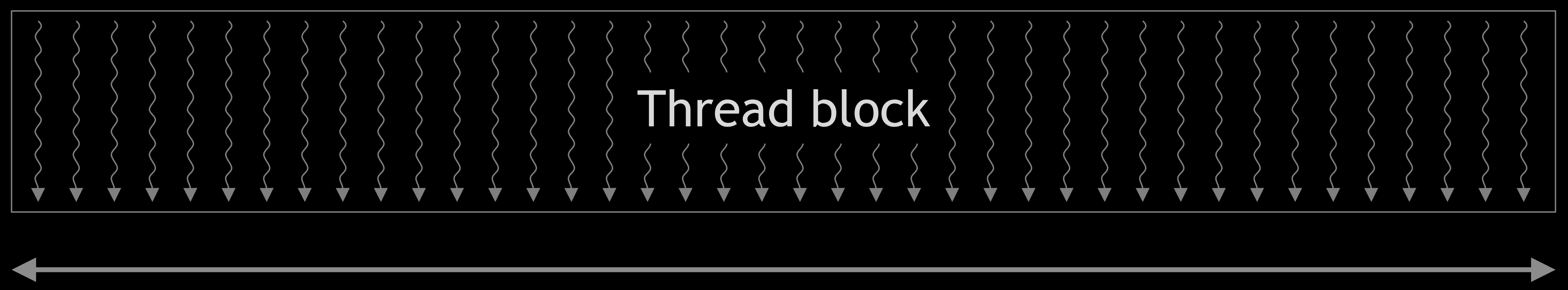
A block has a fixed number of threads always running on a single SM

Example block resource requirements

256	Threads per block
64	(Registers per thread)
(256 * 64) = 16384	Registers per block
32 kB	Shared memory per block

HOW THE GPU PLACES BLOCKS ON AN SM

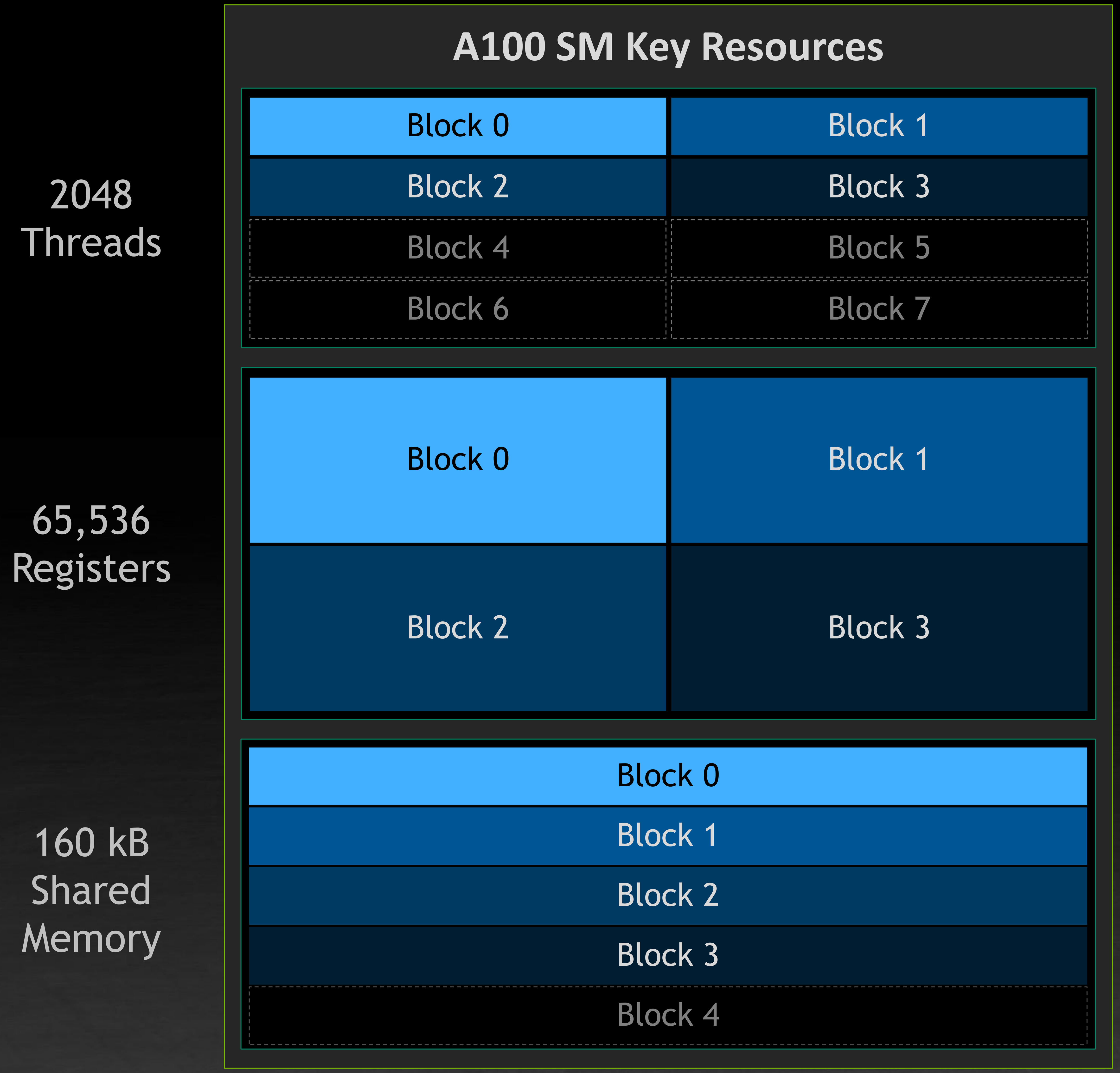


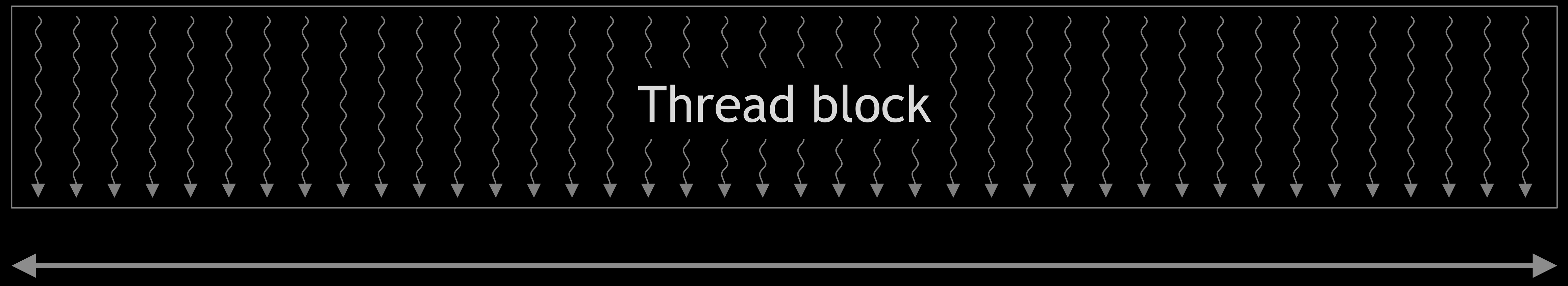


A block has a fixed number of threads
always running on a single SM

Example block resource requirements	
256	Threads per block
64	(Registers per thread)
(256 * 64) = 16384	Registers per block
32 kB	Shared memory per block

HOW THE GPU PLACES BLOCKS ON AN SM

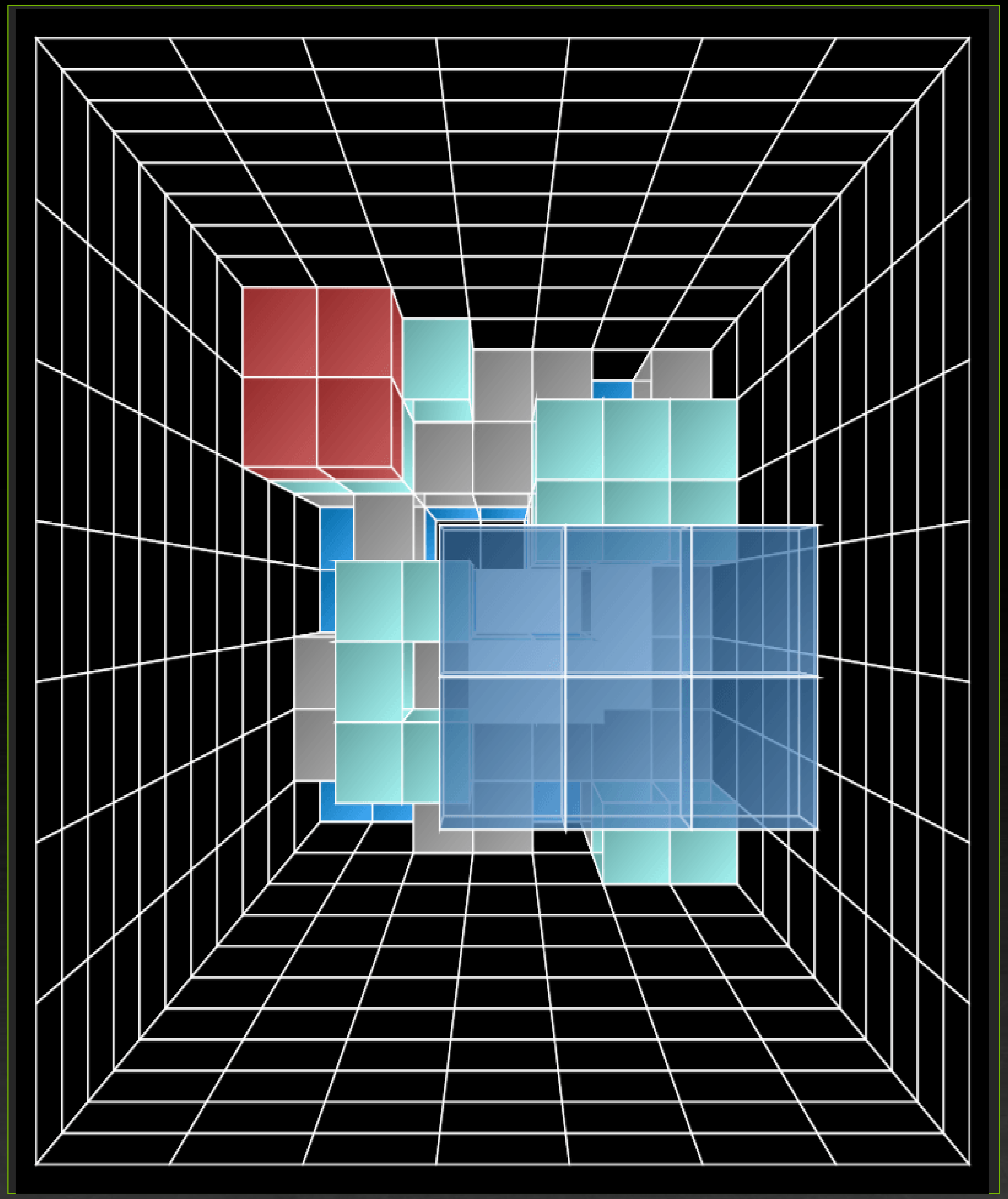


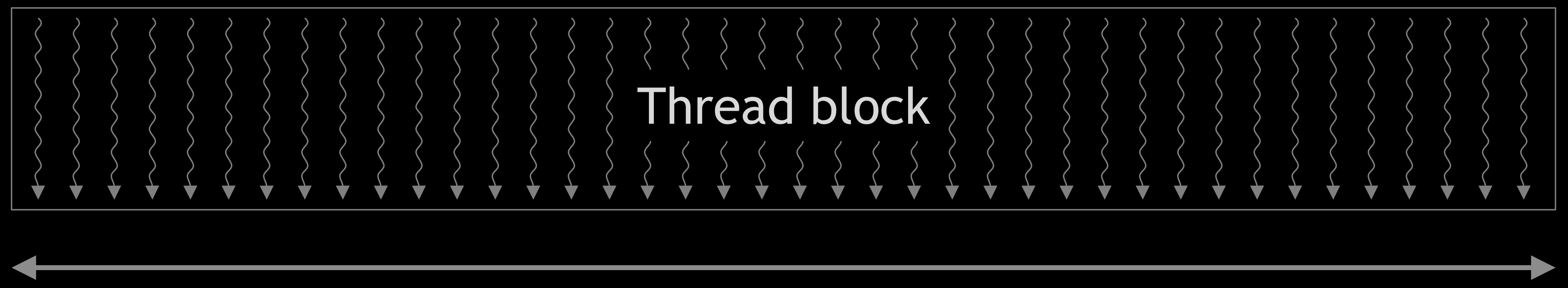


A block has a fixed number of threads always running on a single SM

Example block resource requirements	
256	Threads per block
64	(Registers per thread)
(256 * 64) = 16384	Registers per block
32 kB	Shared memory per block

HOW THE GPU PLACES BLOCKS ON AN SM

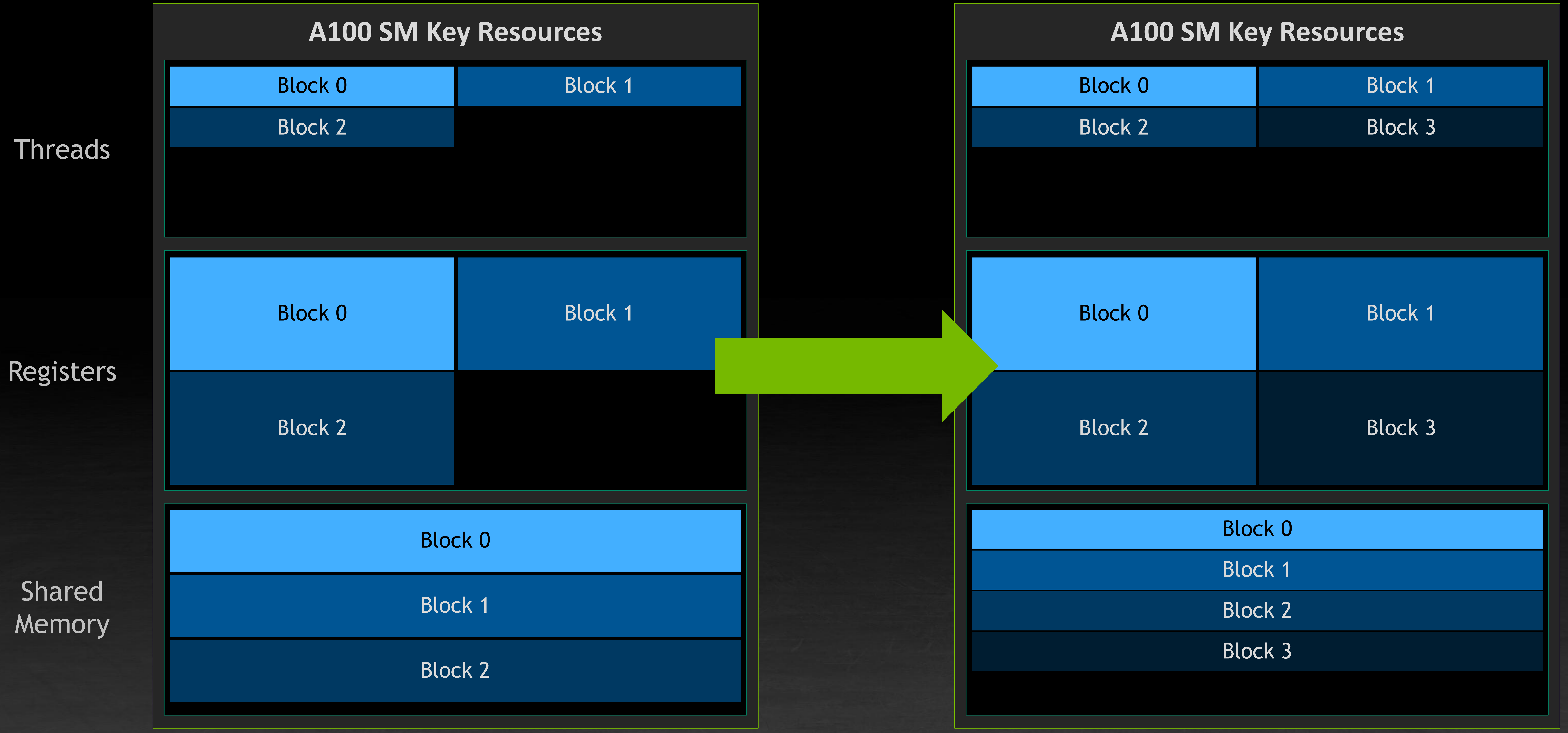




A block has a fixed number of threads
always running on a single SM

Example block resource requirements	
256	Threads per block
64	(Registers per thread)
(256 * 64) = 16384	Registers per block
32 kB	Shared memory per block

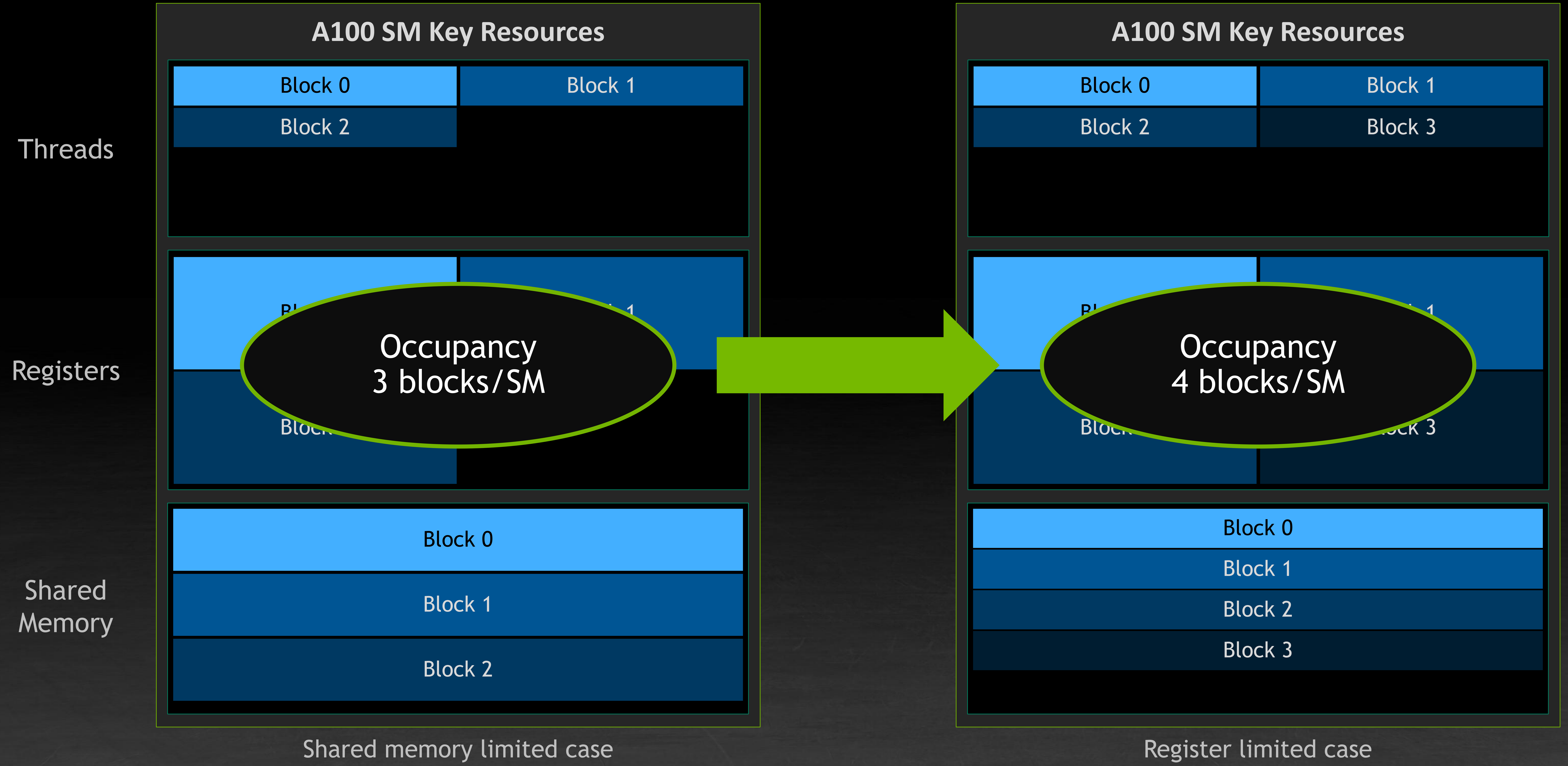
OCCUPANCY



Shared memory limited case

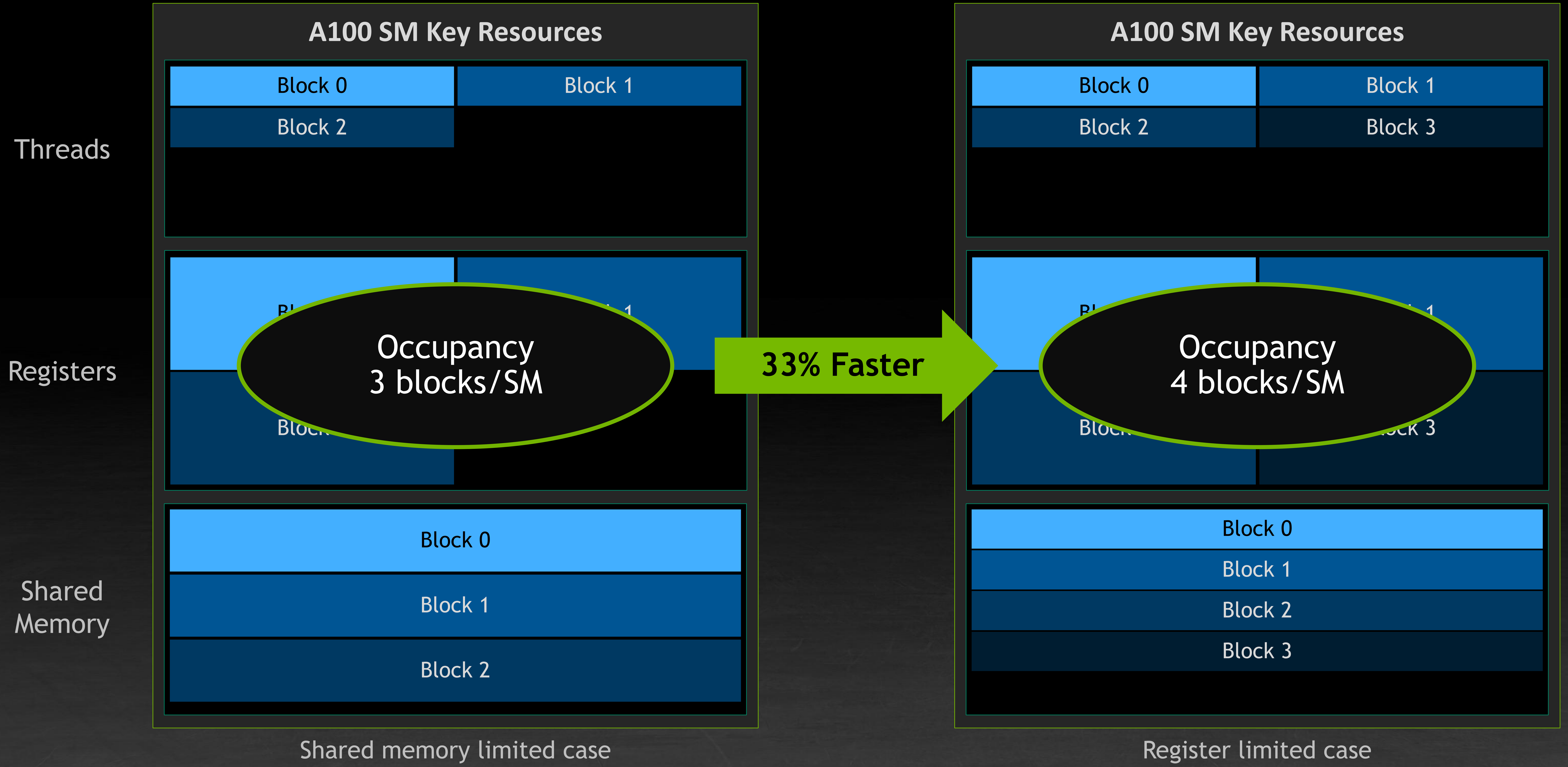
Register limited case

OCCUPANCY

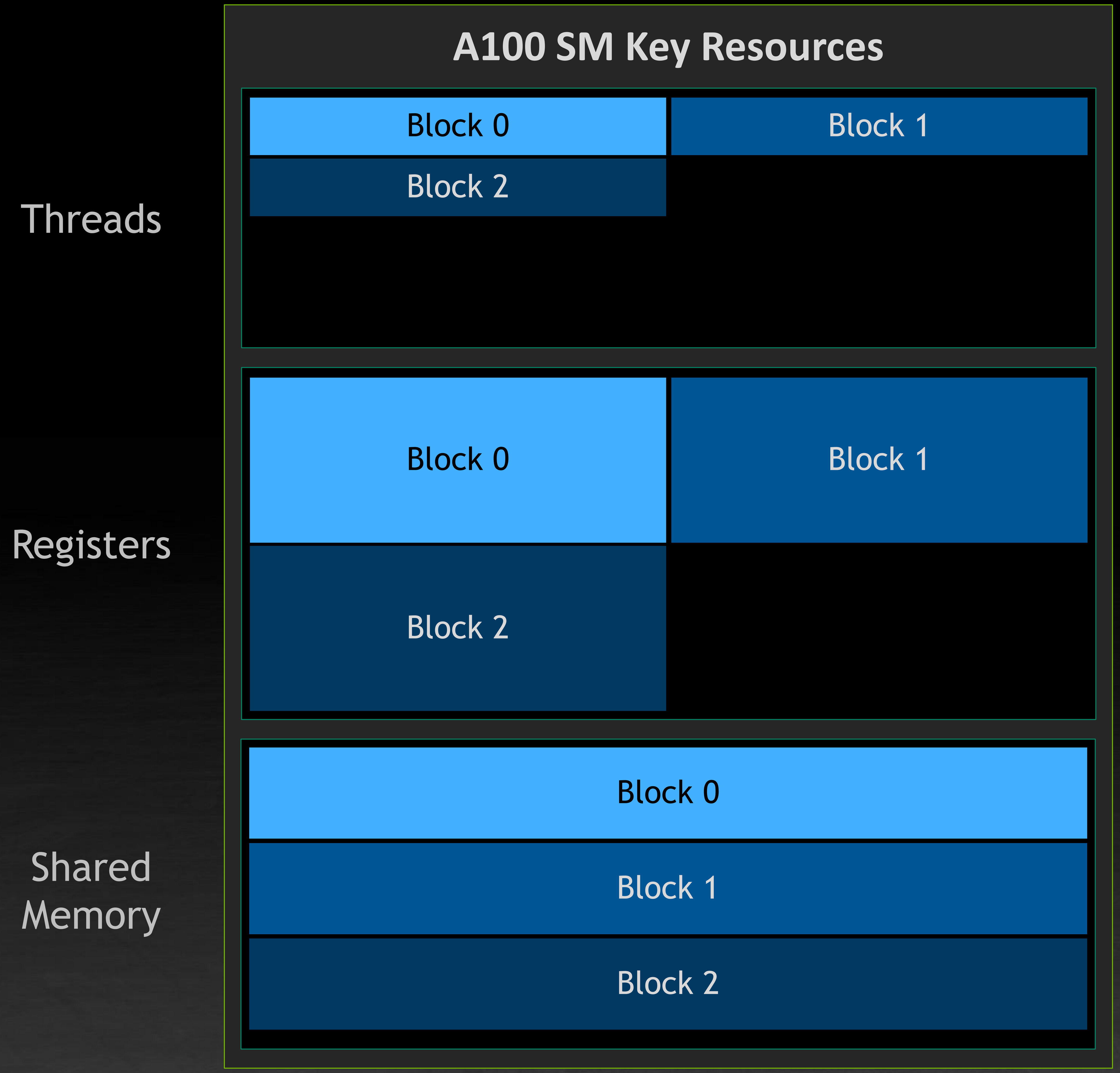


A100 SM Key Resources
Threads
Registers
Shared Memory
Block 0
Block 1
Block 2
Occupancy
3 blocks/SM
Shared memory limited case
A100 SM Key Resources
Block 0
Block 1
Block 2
Block 3
Occupancy
4 blocks/SM
Register limited case

OCCUPANCY IS THE MOST POWERFUL TOOL FOR TUNING A PROGRAM



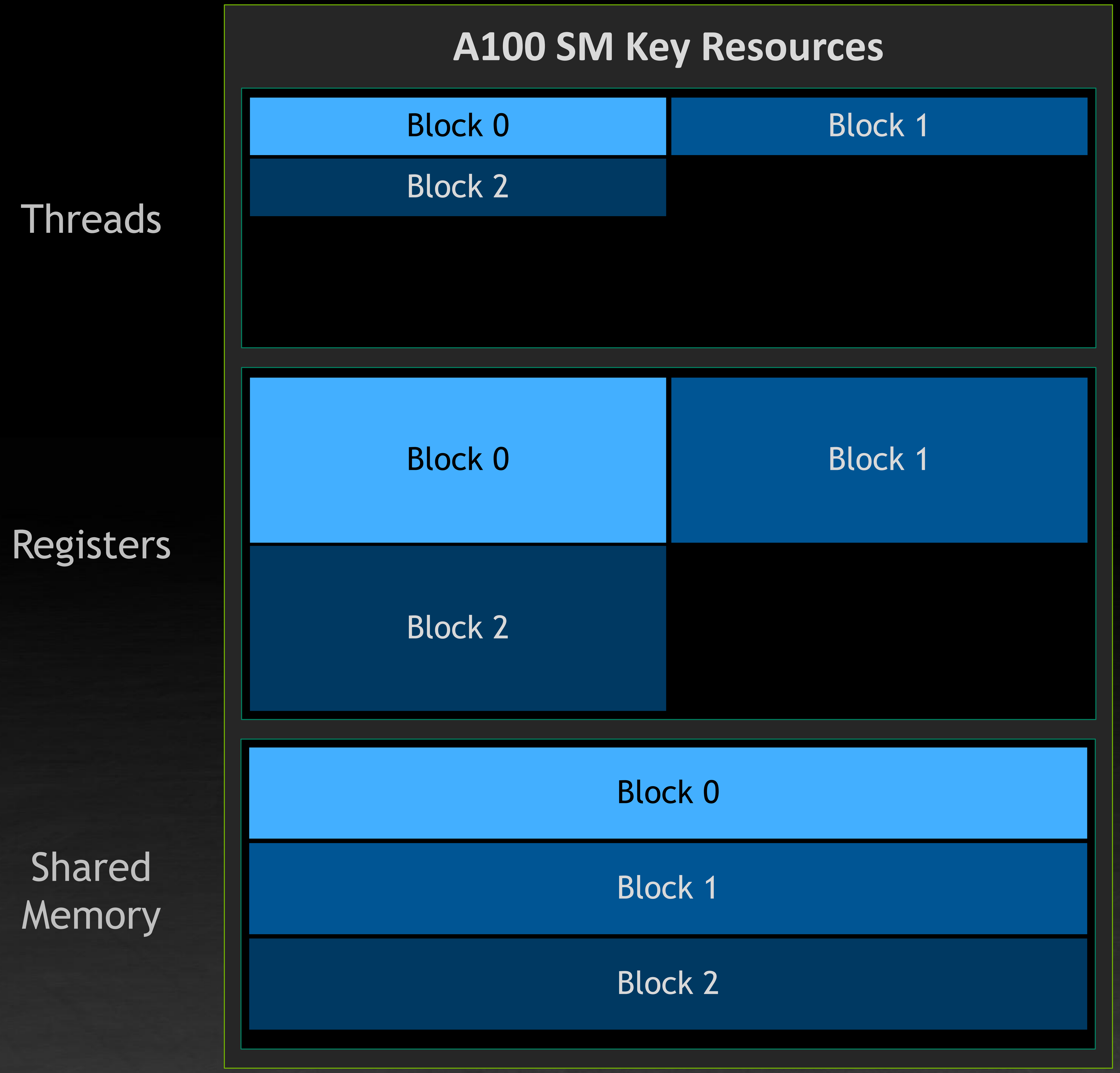
FILLING IN THE GAPS



Shared memory limited case

Resource requirements (blue grid)	
256	Threads per block
64	(Registers per thread)
(256 * 64) = 16384	Registers per block
48 kB	Shared memory per block

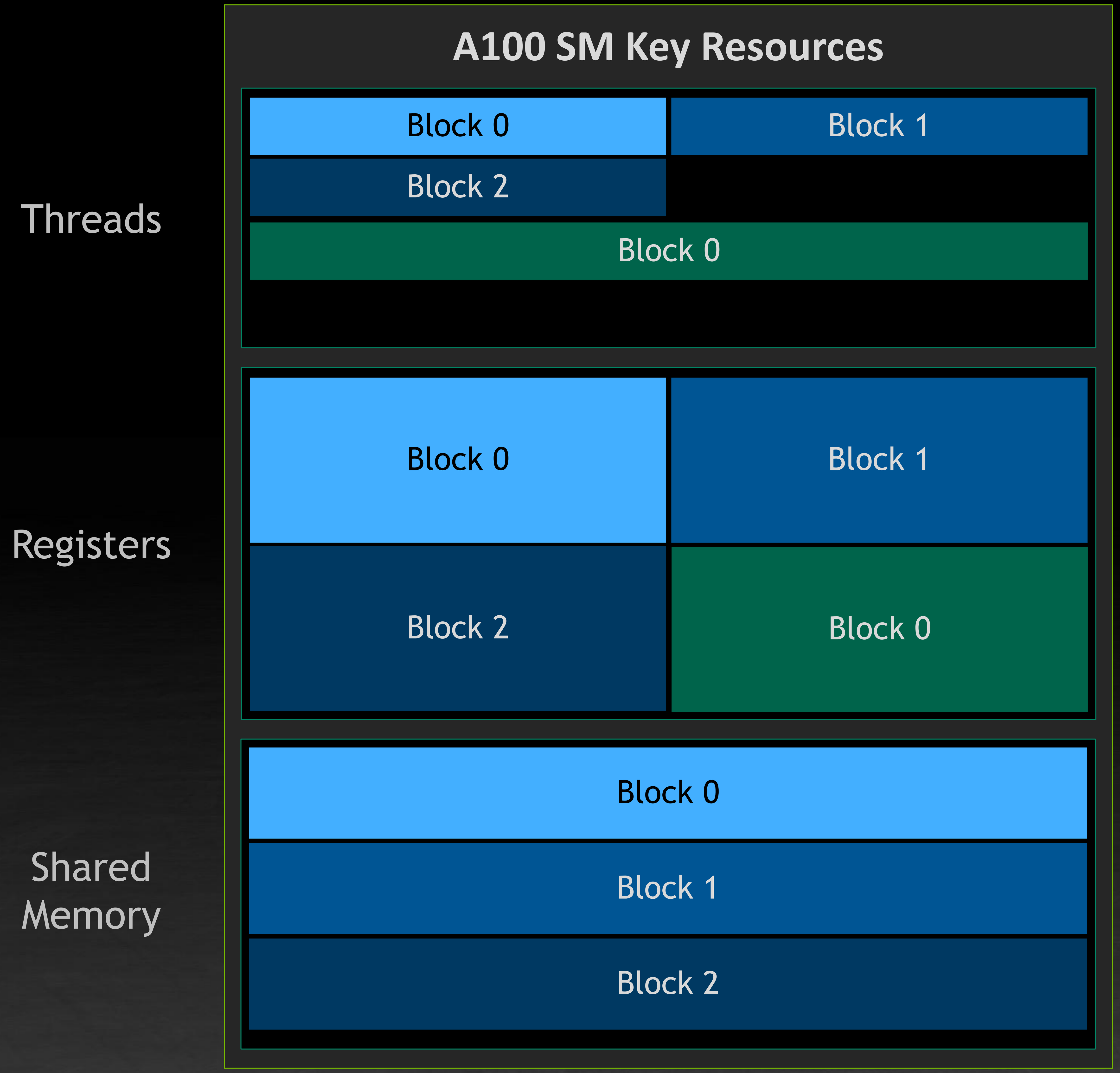
FILLING IN THE GAPS



Resource requirements (blue grid)	
256	Threads per block
64	(Registers per thread)
(256 * 64) = 16384	Registers per block
48 kB	Shared memory per block

Resource requirements (green grid)	
512	Threads per block
32	(Registers per thread)
(512 * 32) = 16384	Registers per block
0 kB	Shared memory per block

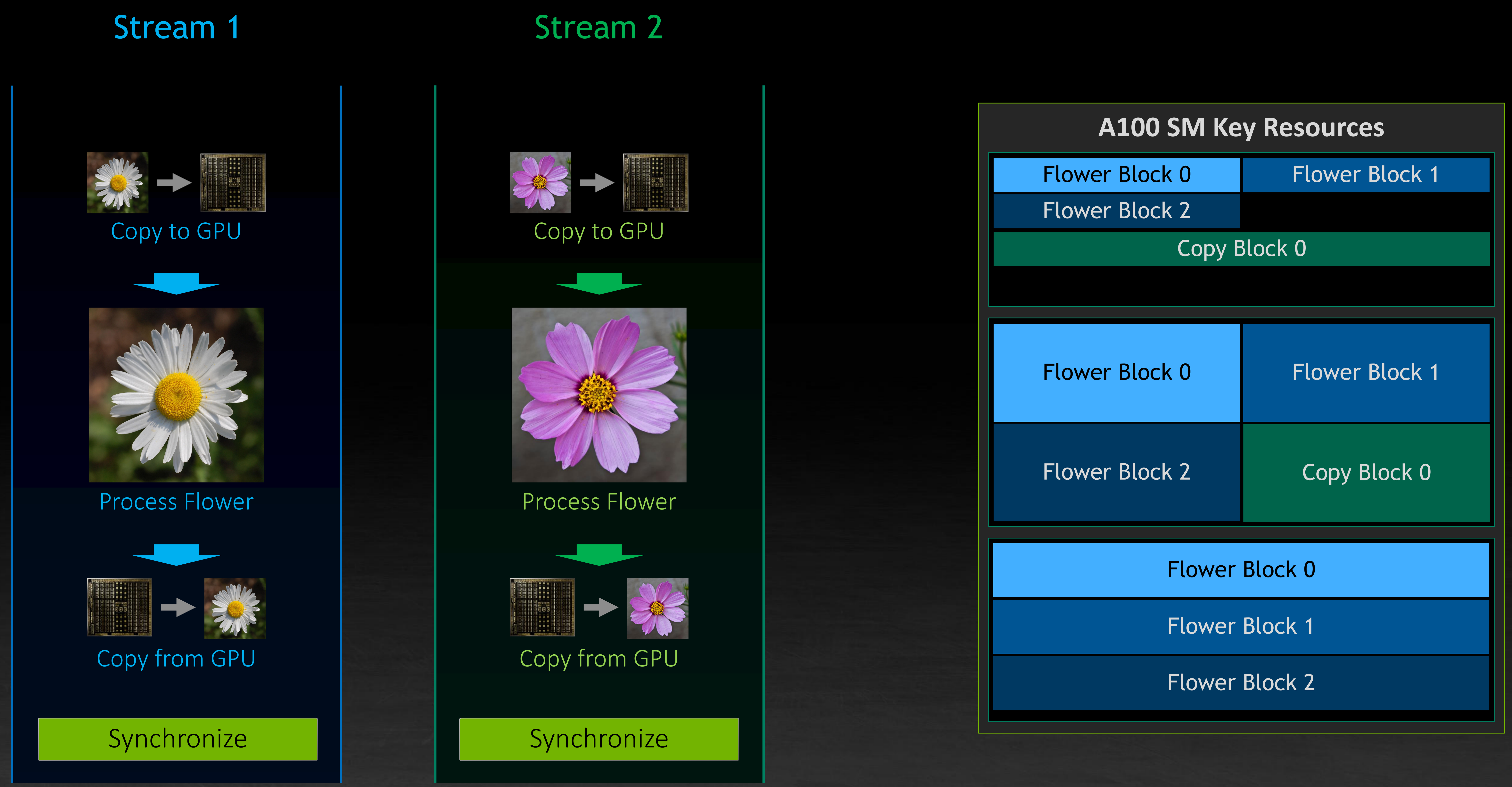
FILLING IN THE GAPS



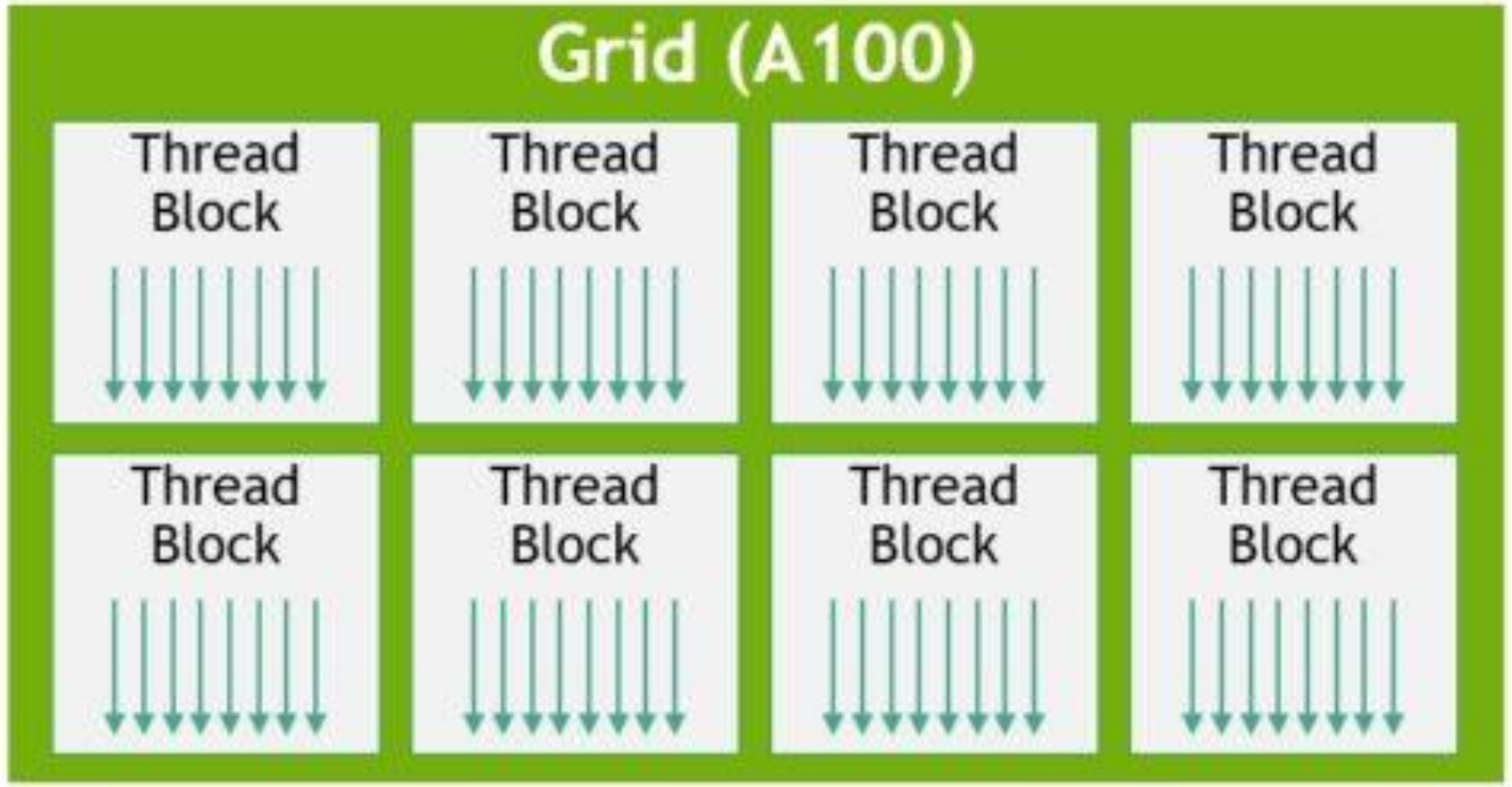
Resource requirements (blue grid)	
256	Threads per block
64	(Registers per thread)
(256 * 64) = 16384	Registers per block
48 kB	Shared memory per block

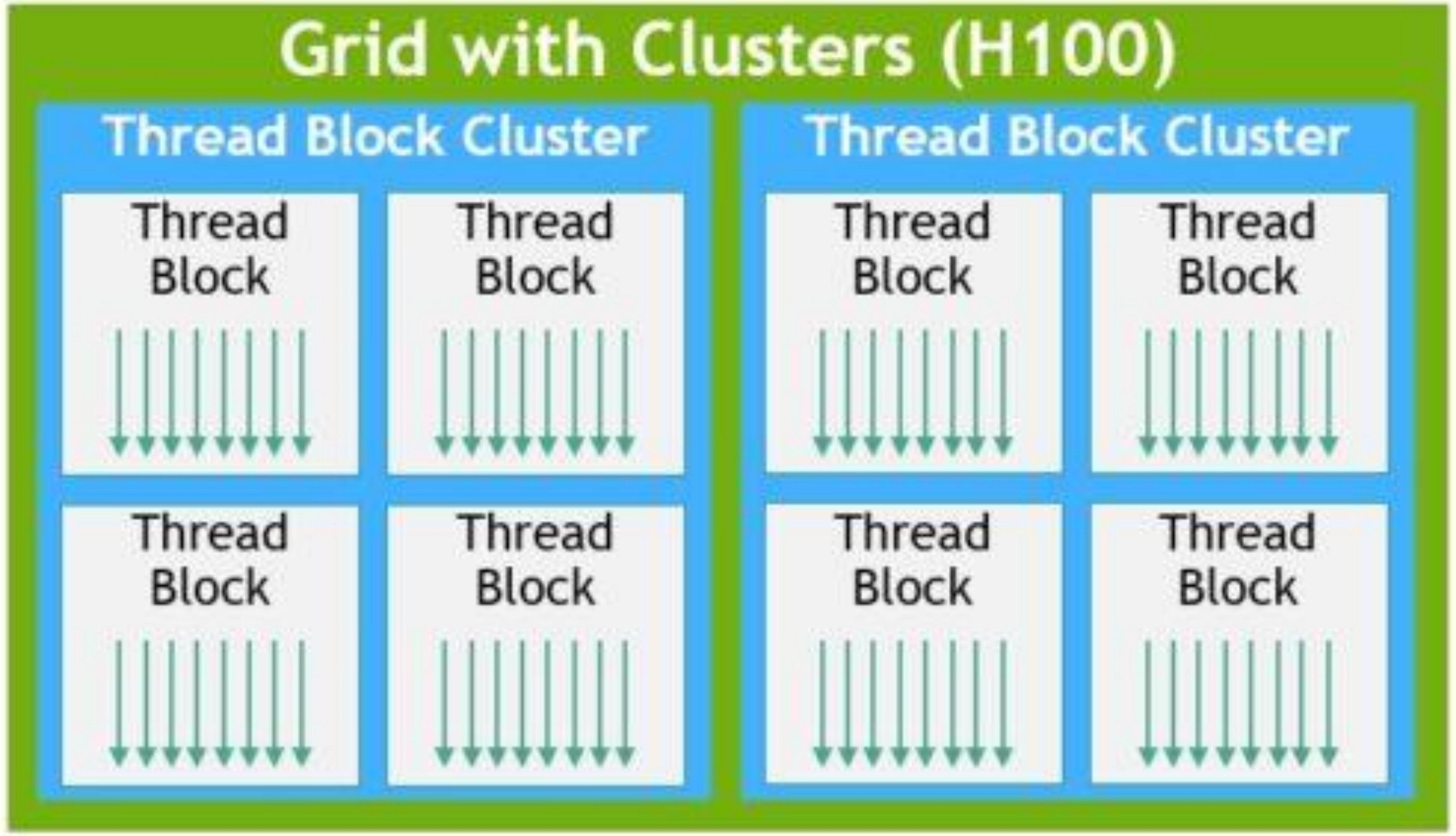
Resource requirements (green grid)	
512	Threads per block
32	(Registers per thread)
(512 * 32) = 16384	Registers per block
0 kB	Shared memory per block

KEEPING THE GPU FULL



Thread Block Clusters





<<< Dg, Db, Ns, S >>>

- Dg: specifies the dimension and size of the grid
- Db: specifies the dimension and size of each block
- Ns: specifies the number of bytes in shared memory that is dynamically allocated per block; Ns is an optional argument which defaults to 0
- S: specifies the associated stream; S is an optional argument which defaults to 0

```
dim3 gridDim = { gX, gY, gZ };
dim3 blockDim = { bX, bY, bZ };

// how to specify cluster dimensions??

kernel<<< gridDim, blockDim >>>(...);
auto err = cudaPeekAtLastError();


assert(cudaSuccess == err);
```

`cudaLaunchKernelEx(cfg, kern, args)`

~~`<<< Dg, Db, Ns, S >>>`~~

- cfg: specifies the launch configuration, including the dimension and size of the grid, the dimension and size of each block, the number of bytes in shared memory, and the associated stream
- kern: specifies the kernel function to launch
- args: specifies the arguments to the kernel function

```
dim3 gridDim = { gX, gY, gZ };
dim3 blockDim = { bX, bY, bZ };


kernel<<< gridDim, blockDim >>>(...);
auto err = cudaPeekAtLastError();


assert(cudaSuccess == err);
```

`cudaLaunchKernelEx(cfg, kern, args)`

~~`<<< Dg, Db, Ns, S >>>`~~

- cfg: specifies the launch configuration, including the dimension and size of the grid, the dimension and size of each block, the number of bytes in shared memory, and the associated stream
- kern: specifies the kernel function to launch
- args: specifies the arguments to the kernel function

```
cudaLaunchConfig_t cfg = { };
dim3 gridDim = { gX, gY, gZ };
dim3 blockDim = { bX, bY, bZ };


kernel<<< gridDim, blockDim >>>(...);
auto err = cudaPeekAtLastError();


assert(cudaSuccess == err);
```

`cudaLaunchKernelEx(cfg, kern, args)`

~~`<<< Dg, Db, Ns, S >>>`~~

- cfg: specifies the launch configuration, including the dimension and size of the grid, the dimension and size of each block, the number of bytes in shared memory, and the associated stream
- kern: specifies the kernel function to launch
- args: specifies the arguments to the kernel function

```
cudaLaunchConfig_t cfg = { };
dim3 cfg.gridDim = { gX, gY, gZ };
dim3 cfg.blockDim = { bX, bY, bZ };

kernel<<< gridDim, blockDim >>>(...);
auto err = cudaPeekAtLastError();

assert(cudaSuccess == err);
```

cudaLaunchKernelEx(cfg, kern, args)

~~<<< Dg, Db, Ns, S >>>~~

- cfg: specifies the launch configuration, including the dimension and size of the grid, the dimension and size of each block, the number of bytes in shared memory, and the associated stream
- kern: specifies the kernel function to launch
- args: specifies the arguments to the kernel function

```
cudaLaunchConfig_t cfg = { };
dim3 cfg.gridDim = { gX, gY, gZ };
dim3 cfg.blockDim = { bX, bY, bZ };

// what about cluster dimensions??

kernel<<< gridDim, blockDim >>>(...);
auto err = cudaPeekAtLastError();
auto err = cudaLaunchKernelEx(
    &cfg, kernel, ...
);
assert(cudaSuccess == err);
```

cudaLaunchAttribute

- id: specifies the type of launch attribute
- val: specifies the value of the launch attribute; interpreted differently based on the launch attribute type

```
cudaLaunchAttribute attr;
attr.id =
  cudaLaunchAttributeClusterDimension;
attr.val.clusterDim = { cX, cY, cZ };

cudaLaunchConfig_t cfg = { };
cfg.gridDim = { gX, gY, gZ };
cfg.blockDim = { bX, bY, bZ };
cfg.attrs = &attr;
cfg.numAttrs = 1;


auto err = cudaLaunchKernelEx(
    &cfg, kernel, ...
);
assert(cudaSuccess == err);
```

cudaLaunchAttribute

- accessPolicyWindow
- cooperative
- clusterDim
- clusterSchedulingPolicyPreference
- priority
- syncPolicy

```
cudaLaunchAttribute attr;
attr.id =
  cudaLaunchAttributeClusterDimension;
attr.val.clusterDim = { cX, cY, cZ };

cudaLaunchConfig_t cfg = { };
cfg.gridDim = { gX, gY, gZ };
cfg.blockDim = { bX, bY, bZ };
cfg.attrs = &attr;
cfg.numAttrs = 1;


auto err = cudaLaunchKernelEx(
    &cfg, kernel, ...
);
assert(cudaSuccess == err);
```

NVIDIA