NTNU | Norwegian University of Science and Technology

Cooperation Organisation

Bart Iver van Blokland

Today

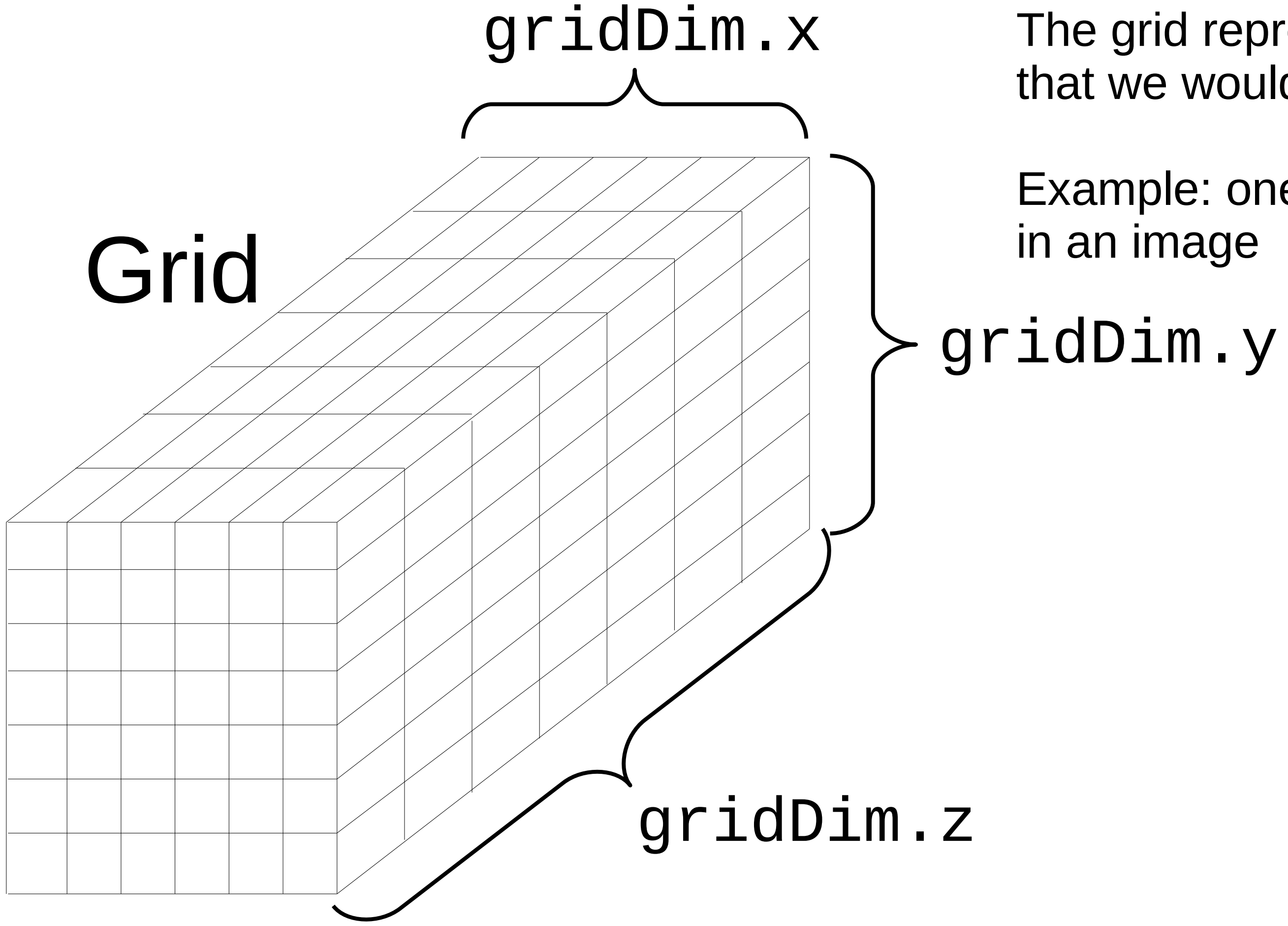
- **More about SM limits**
- Cooperative groups

SM Limits

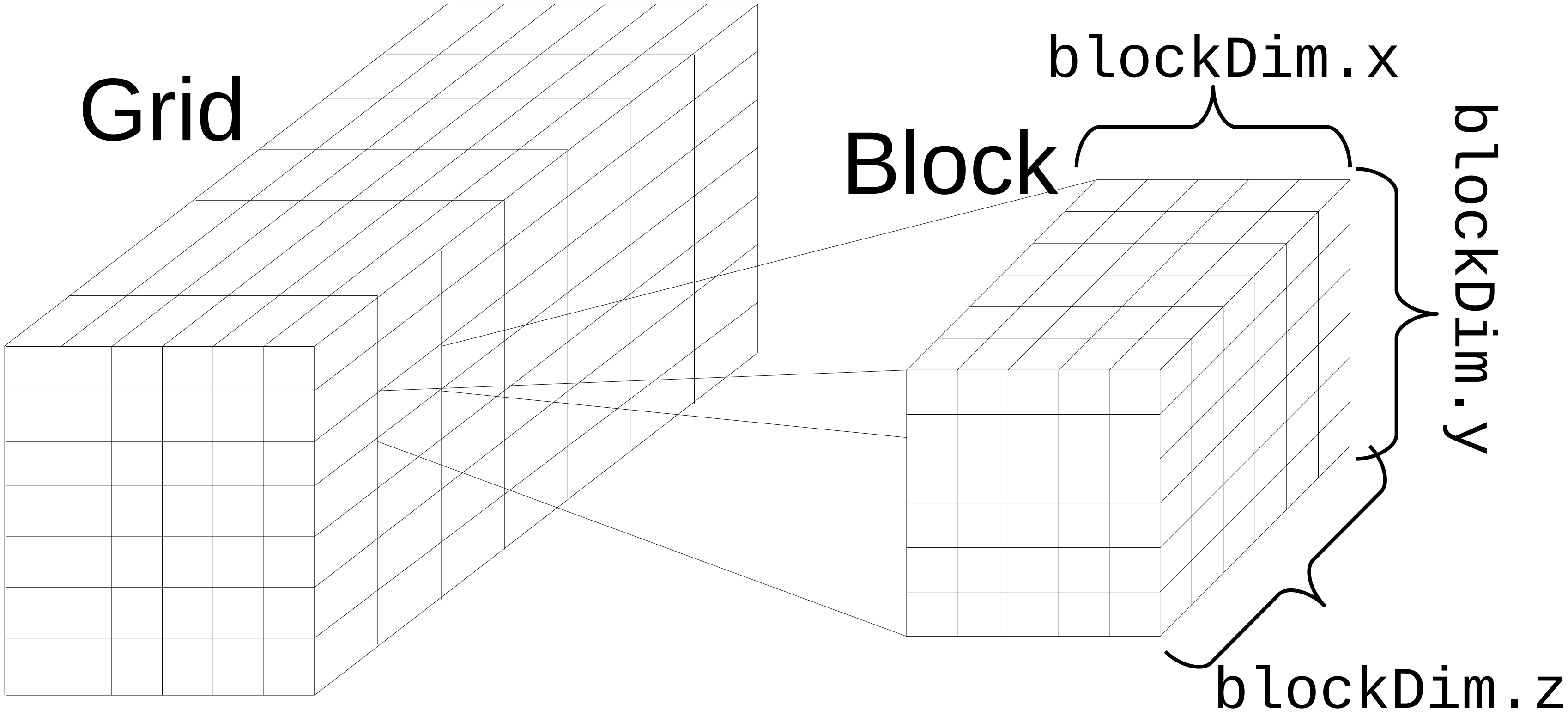
- SM functionality is implemented in hardware
  - Imposes limits on the number of simultaneously executing threads
  - If misconfigured can leave a lot of performance on the table
  - Upside: easy changes can improve performance quite a bit

The grid represents all threads that we would like to run

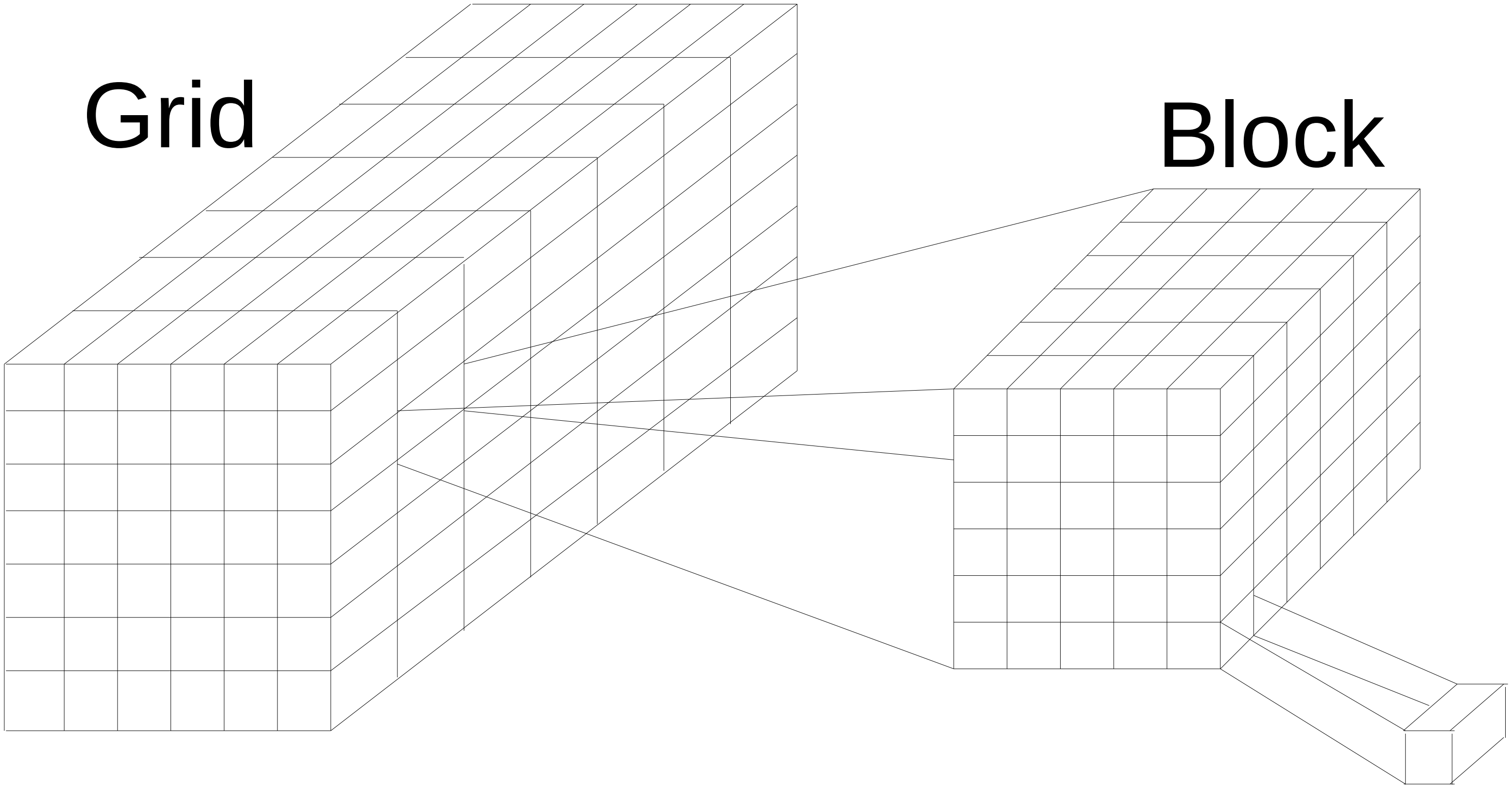
Example: one thread per pixel in an image



A block is a chunk of our grid that is small enough to fit within an SM



A block consists of a number of threads



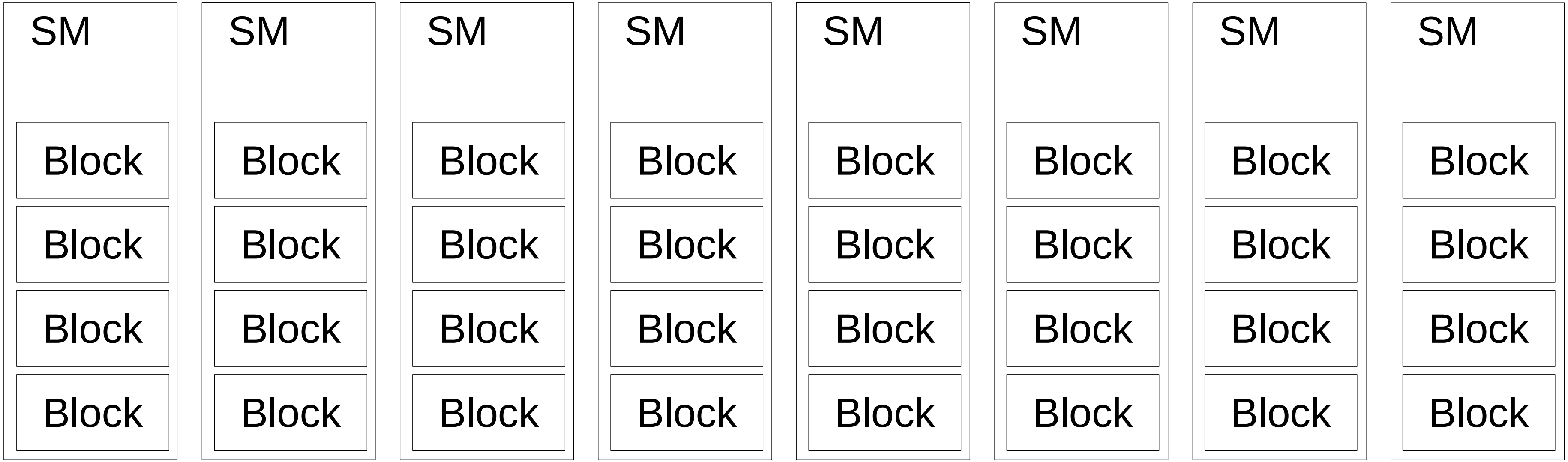
Launching a 1D kernel:

```
someKernel<<<50, 32>>>(parameter1, parameter2);
```

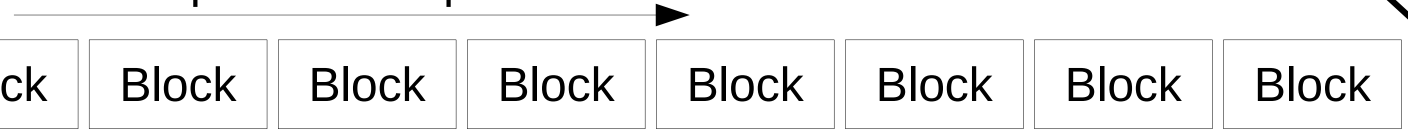
Launching multiple dimensions:

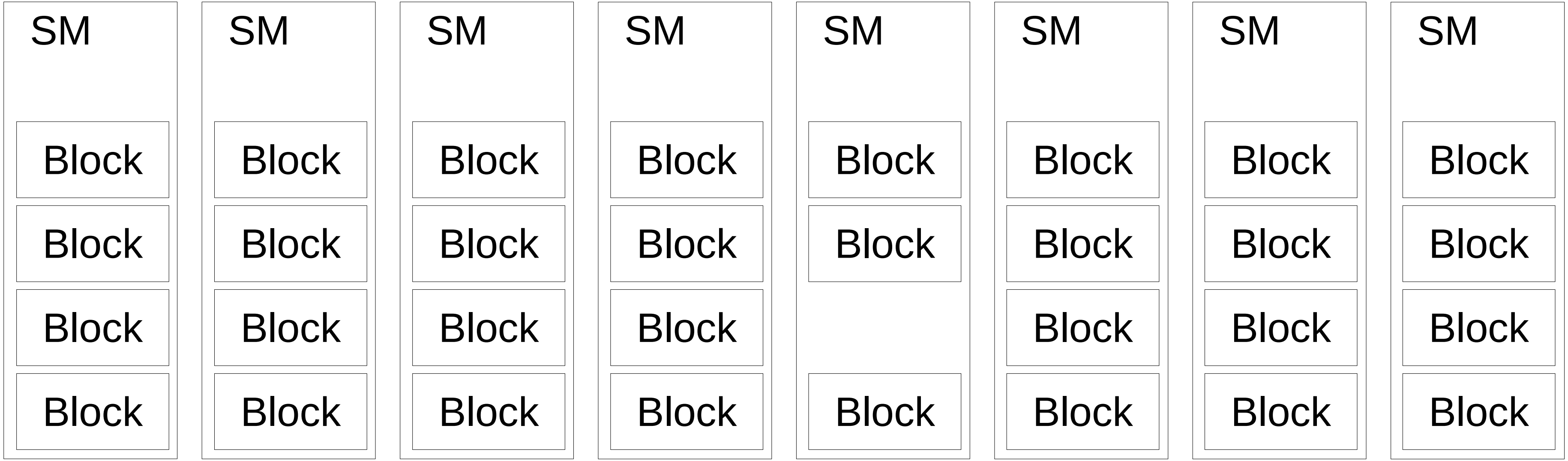
```
dim3 gridDimensions = {30, 20, 1};
dim3 blockDimensions = {32, 2, 1};
someKernel<<<gridDimensions, blockDimensions>>>(...);
```

Make sure no dimension is 0!

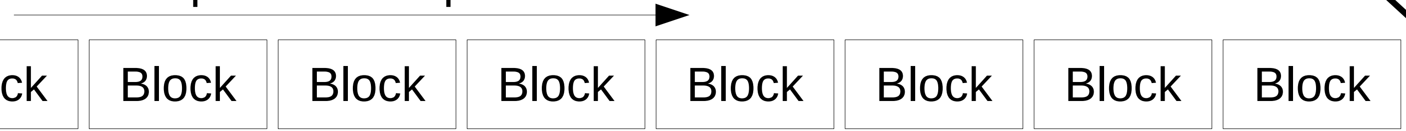


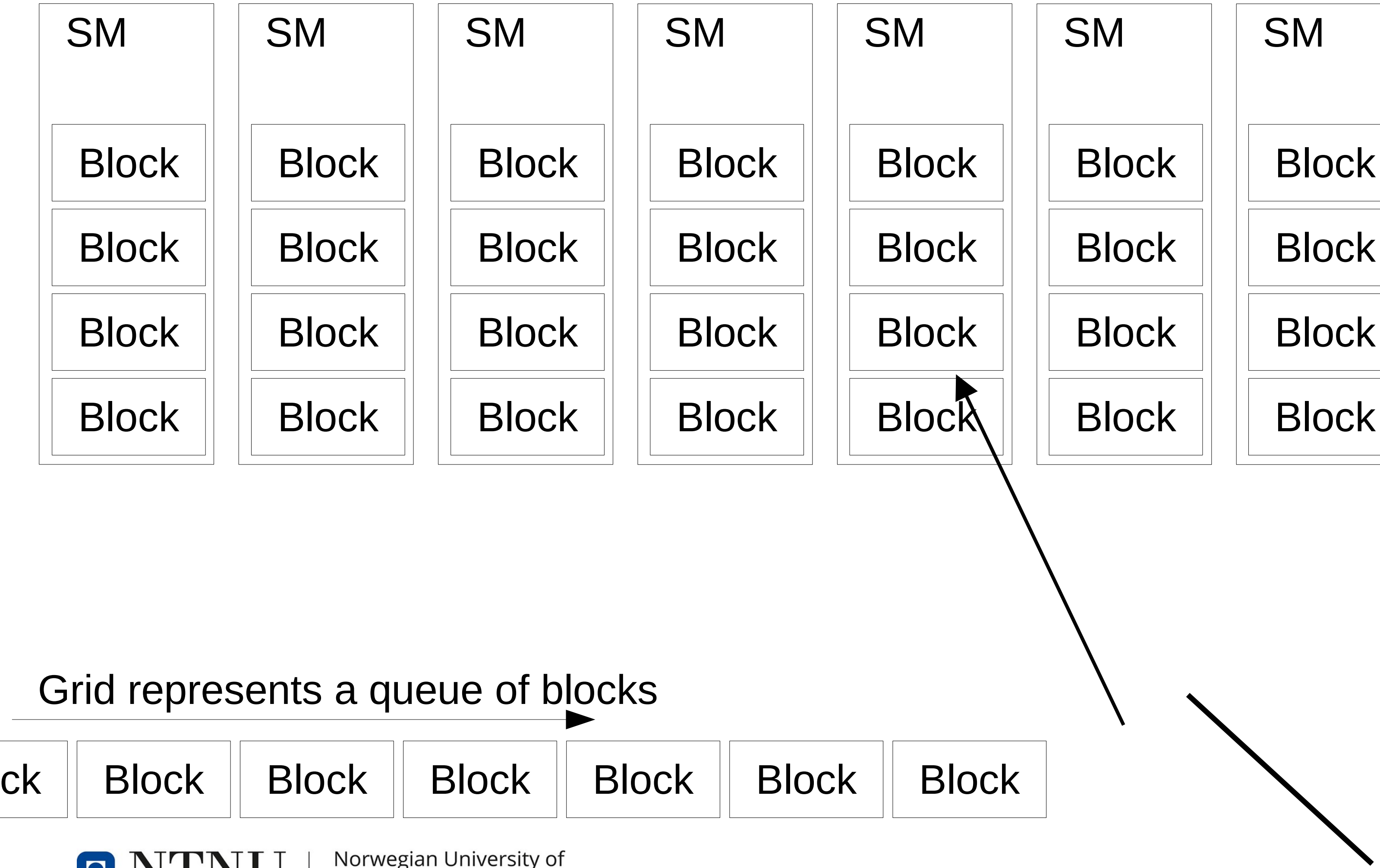
Grid represents a queue of blocks

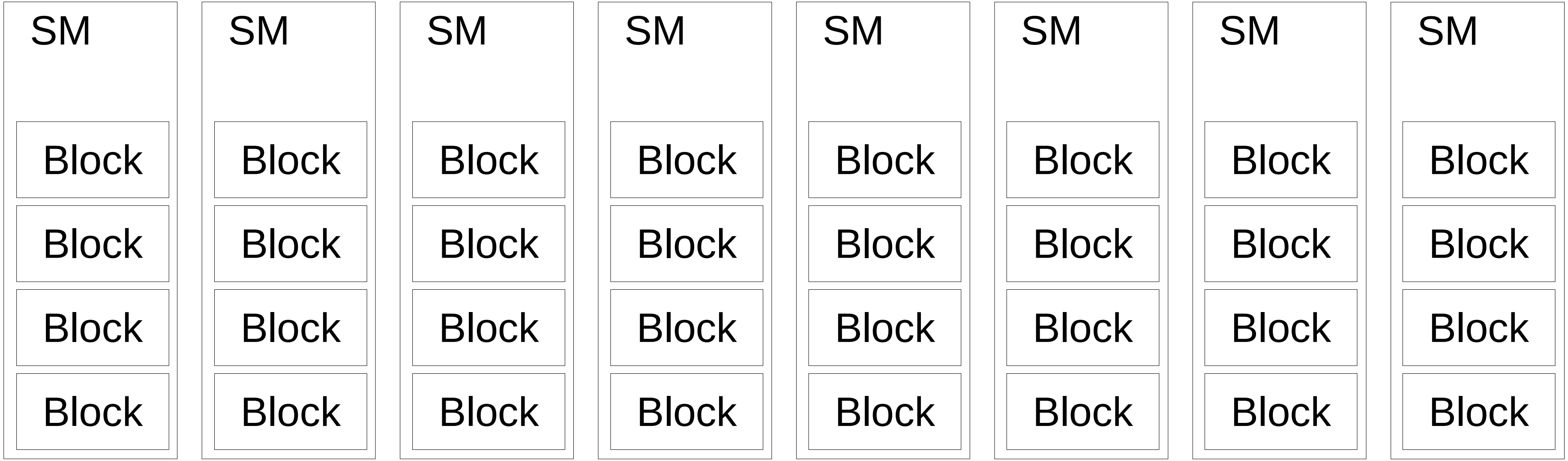




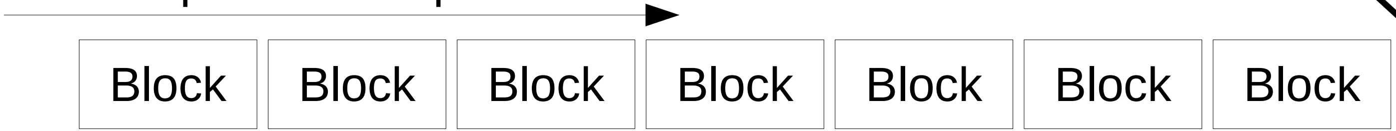
Grid represents a queue of blocks

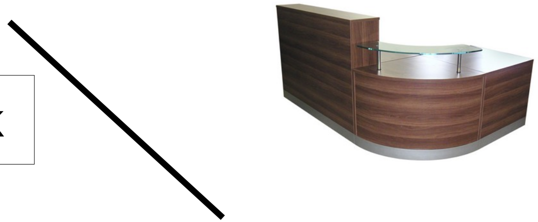



SM
Block
Grid represents a queue of blocks
ck
Block



Grid represents a queue of blocks





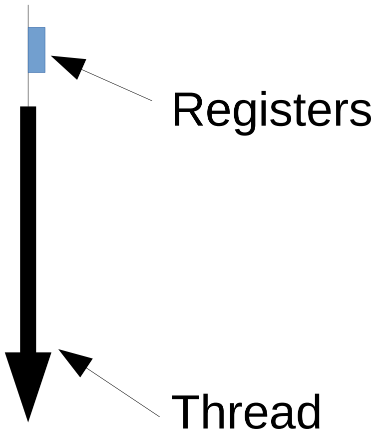
SM Limits (Ada Lovelace)

- Limit 1: Number of threads per block: 1024
- Limit 2: Number of blocks per SM: 24
- Limit 3: Number of warps per SM: 48 (1536 threads)
- Limit 4: Number of registers per thread: 255
- Limit 5: Available shared memory: up to 100Kb

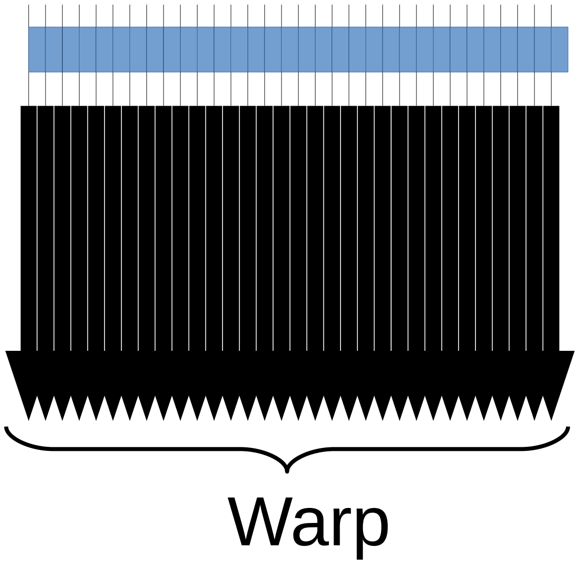
Note: vary quite a bit for each architecture generation

And there are more..

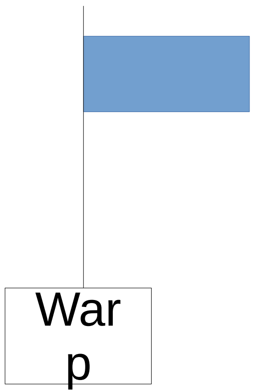
Each thread uses a fixed number of registers



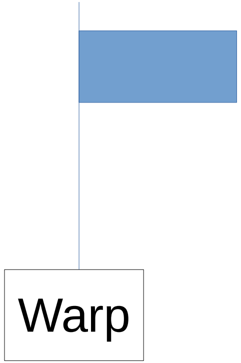
A warp uses 32x that number of registers



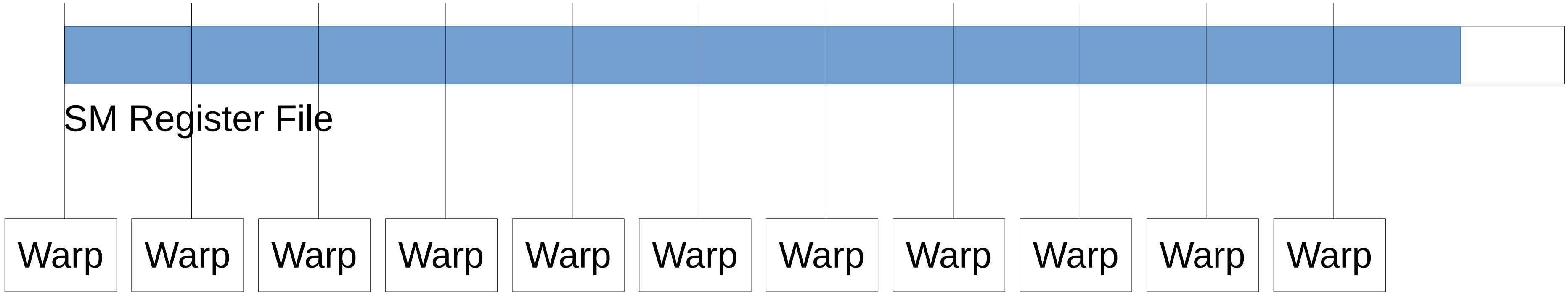
Visual simplification



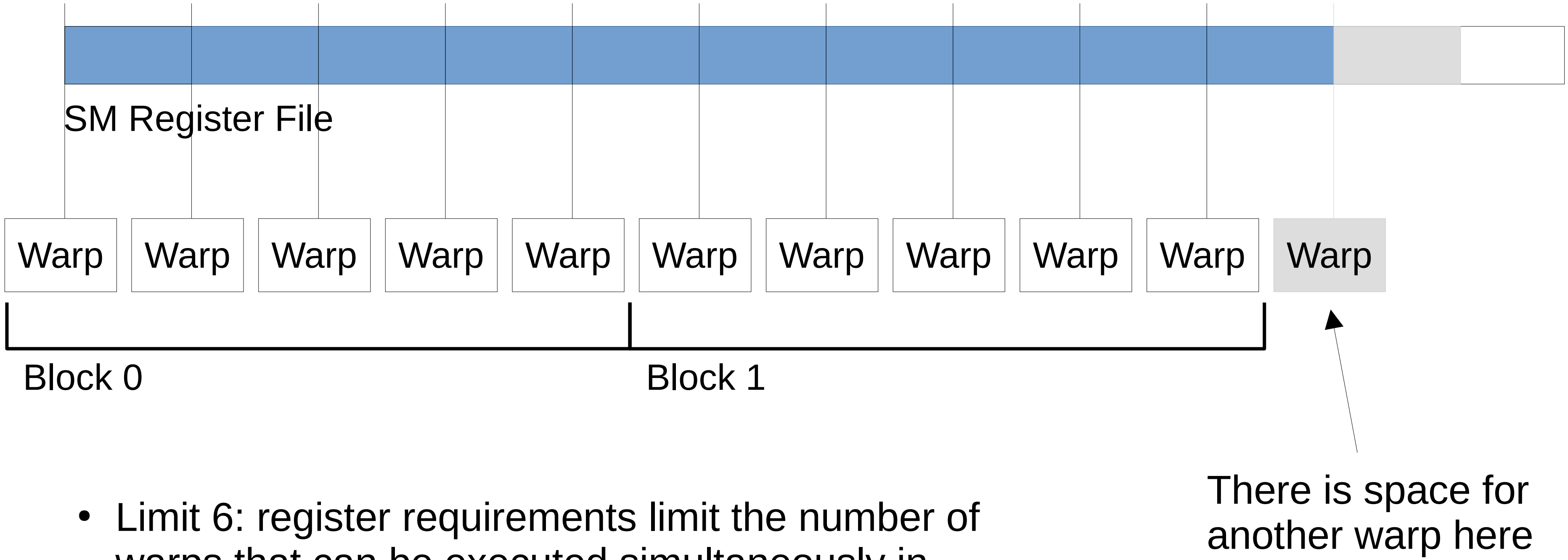
SM Register File



- Register values are kept in the register files within the SM

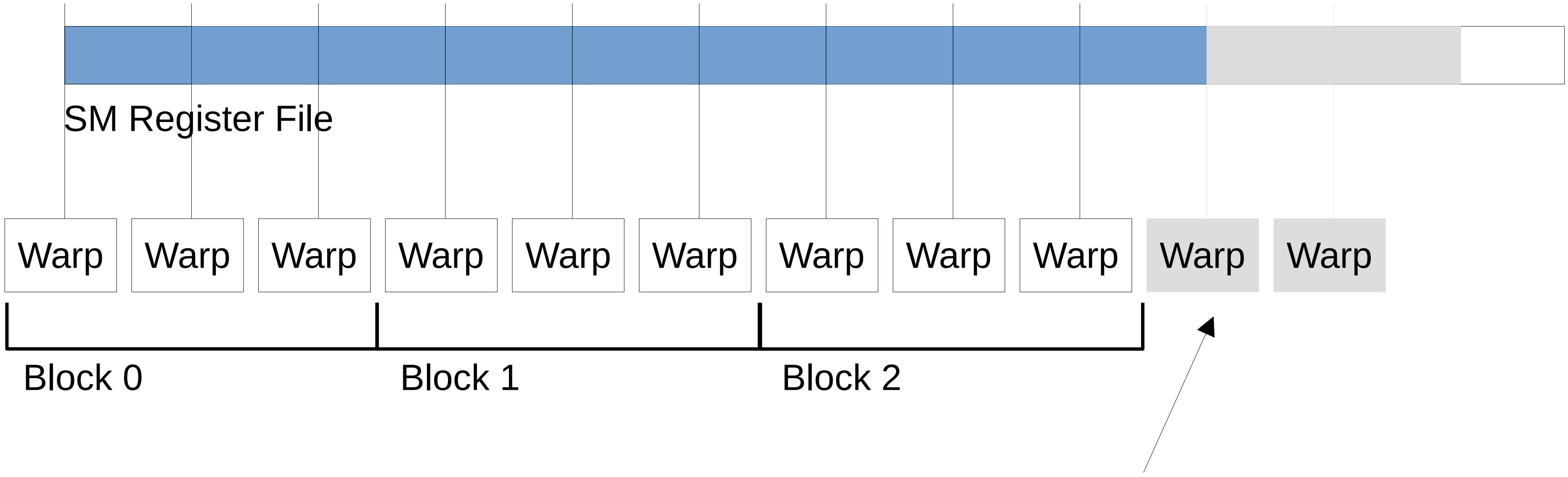


- Limit 6: register requirements limit the number of warps that can be executed simultaneously in an SM



- Limit 6: register requirements limit the number of warps that can be executed simultaneously in an SM

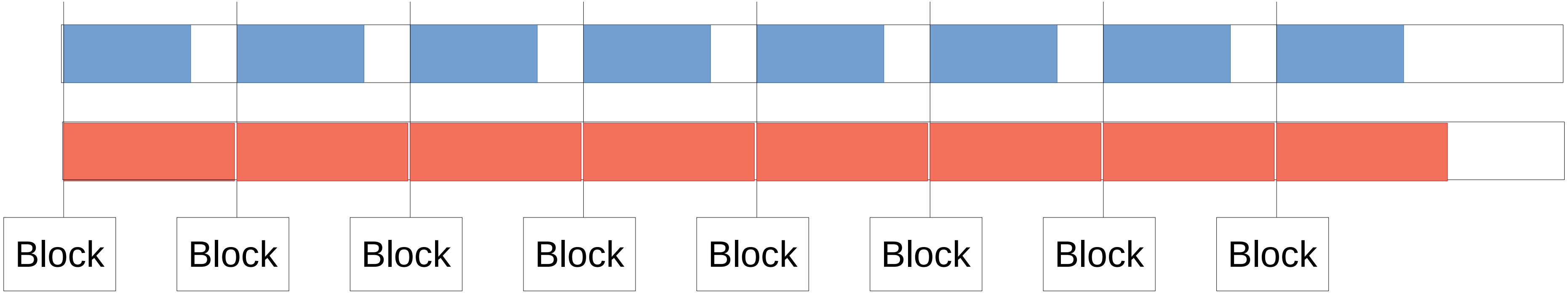
- Limit 7: blocks have a constant number of warps and cannot be partially allocated to an SM



Note: blocks are units of threads allocated to SM's.

While running the threads in a block, the SM executes instructions as warps

Making blocks smaller can help, but not necessarily.



- Limit 8: shared memory required by each block limits the number of warps

```
__shared__ float partialSums[128];
```

You may be able to alter the amount depending on your algorithm

Block size tradeoffs

- Why larger blocks:
  - More ability to cooperate between threads
  - Better reuse of shared memory

- Why smaller blocks:
  - Less waiting for all warps in the block to finish
  - May (but not always) improve occupancy by allowing more warps to execute simulateneously

SM Limits

- All these limits affect the active warp count
- Even a small increase in block size or shared memory requirements can halve your active warp count and thereby (often) performance
- nvidia has made tools available to help you find the optimal block dimensions based on your kernel

Today

- More about SM limits
- **Cooperative groups**

Cooperative Groups

- Take the threads in blocks and warps, and partitions them into groups that work together.
  - Most of this collaboration stems from communication between threads within the same warp being cheap on the GPU
  - We will talk about this collaboration in the next lecture
  - For today: can get specific threads to wait for each other

Cooperative Groups

- How to use cooperative groups:

```
#include <cooperative_groups.h>

// Not required, but recommended
namespace cg = cooperative_groups;
```

Cooperative Groups

- Group types (sorted smallest to largest)
  - Coalesced group: the threads in the current warp, but only the ones that are executing at that point in time
  - Block group: a group with all threads in the current block
  - Grid group: a group of all threads in the entire grid
  - Cluster group: a cluster is a union of multiple thread blocks. Currently unavailable to us mortals (only supported on the H100 GPU)

Coalesced Group

```
__global__ void kernel() {
    if(threadIdx.x < 12) {
        return;
    }
    cg::coalesced_group warp = cg::coalesced_threads();
    printf("Thread %i/%i\n", warp.thread_rank(),
                             warp.num_threads());
}

int main() {
    kernel<<<1, 32>>>();
    cudaDeviceSynchronize();
    return 0;
}
```

32 threads active here

20 threads active here

Block Group

```
__global__ void kernel() {
    cg::thread_block block = cg::this_thread_block();
    printf("Thread %i/%i\n", block.thread_rank(),
                              block.num_threads());
}

int main() {
    kernel<<<1, 256>>>();
    cudaDeviceSynchronize();
    return 0;
}
```

Here:
`thread_rank()` is between 0 and 255
`num_threads()` is 256

Grid Group

```
__global__ void kernel() {
    cg::grid_group grid = cg::this_grid();
    printf("Thread %i/%i\n", grid.thread_rank(),
                             grid.num_threads());
}

int main() {
    cudaLaunchCooperativeKernel(kernel, {20, 1, 1},
                                {256, 1, 1}, nullptr);
    cudaDeviceSynchronize();
    return 0;
}
```

Here: 0 to 5119

Here: 5120

Restrictions:

- GPU you run the grid on **MUST** be able to run **ALL** blocks **at the same time**
- Cannot use the <<<>>> syntax, must use function to launch kernel

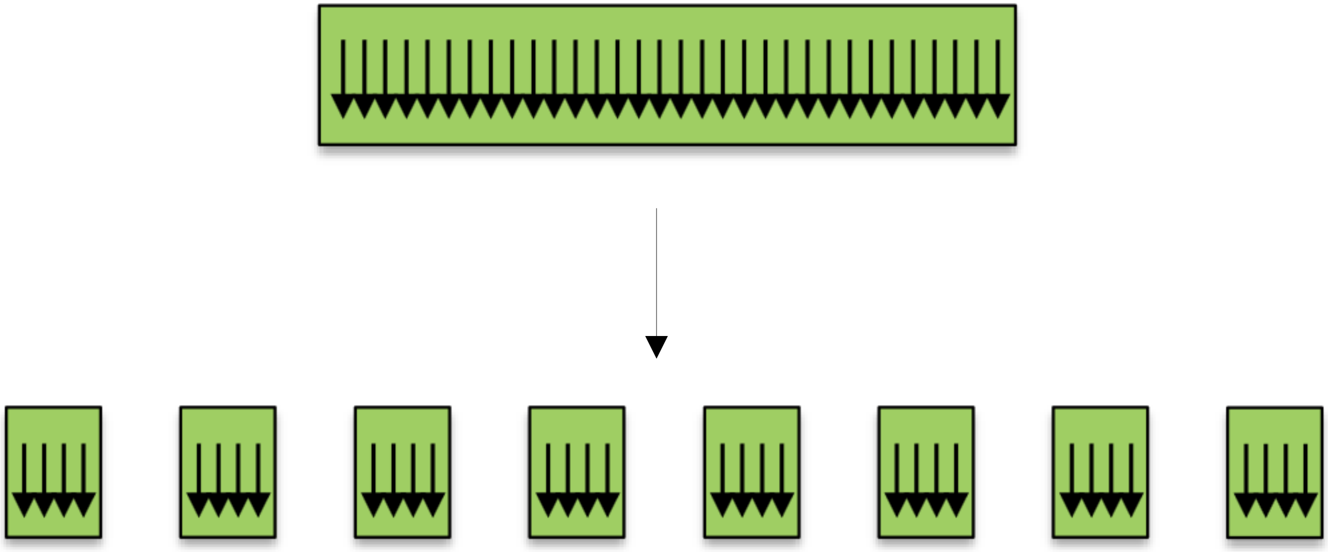
Cooperative Groups

- `num_threads()` and `thread_rank()` are thin abstractions over `threadIdx`, `blockDim`, `blockIdx`, and `gridDim`.

- The advantages of cooperative groups lie elsewhere:
  - Group partitioning
  - Selective synchronisation
  - Somewhat simplified warp communication

Group Partitioning

- Create smaller subgroups from larger ones
- Similar capabilities and interface to the larger groups



Group Partitioning

- Create smaller subgroups from larger ones

- From cg::thread_block:

```
cg::thread_block block = cg::this_thread_block();

// Use if size known at compile time:
cg::thread_block_tile<8> tile8 = cg::tiled_partition<8>(block);

// Use if size unknown at compile time:
cg::thread_group tile8_dynamic = cg::tiled_partition(block, 8);
```

Warp partitioning

- From cg::thread_block:

```
cg::coalesced_group warp = cg::coalesced_threads();

// Use if partition count known at compile time:
int label = threadIdx.x % 8;
cg::coalesced_group<8, int> part
    = cg::labeled_partition<int>(warp, label);

// Use if partition count unknown at compile time:
cg::coalesced_group<int> part_dynamic
    = cg::labeled_partition(warp, label);

// binary_partition: use if you only need to split in 2 using a
boolean predicate
```

Selective synchronisation

- For ANY group or partition, call the sync() function on the group object to place a barrier and wait for all threads to reach that point in the kernel

```
cg::thread_block block = cg::this_thread_block();

block.sync();
```

Cooperative Groups

- Gotchas:
  - All groups must contain a thread count that is a power of 2
  - Once created, the members of a group do not change (important for `coalesced_group`)
  - Need cooperative groups for syncing in branch

Today

- More about SM limits
- Cooperative groups

Next week

- GPU Shenanigans!