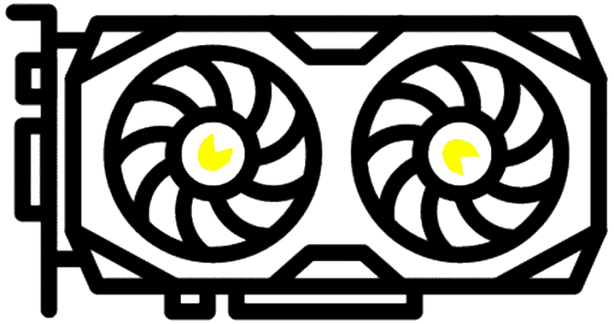
University of Applied Sciences and Arts Northwestern Switzerland
School of Engineering

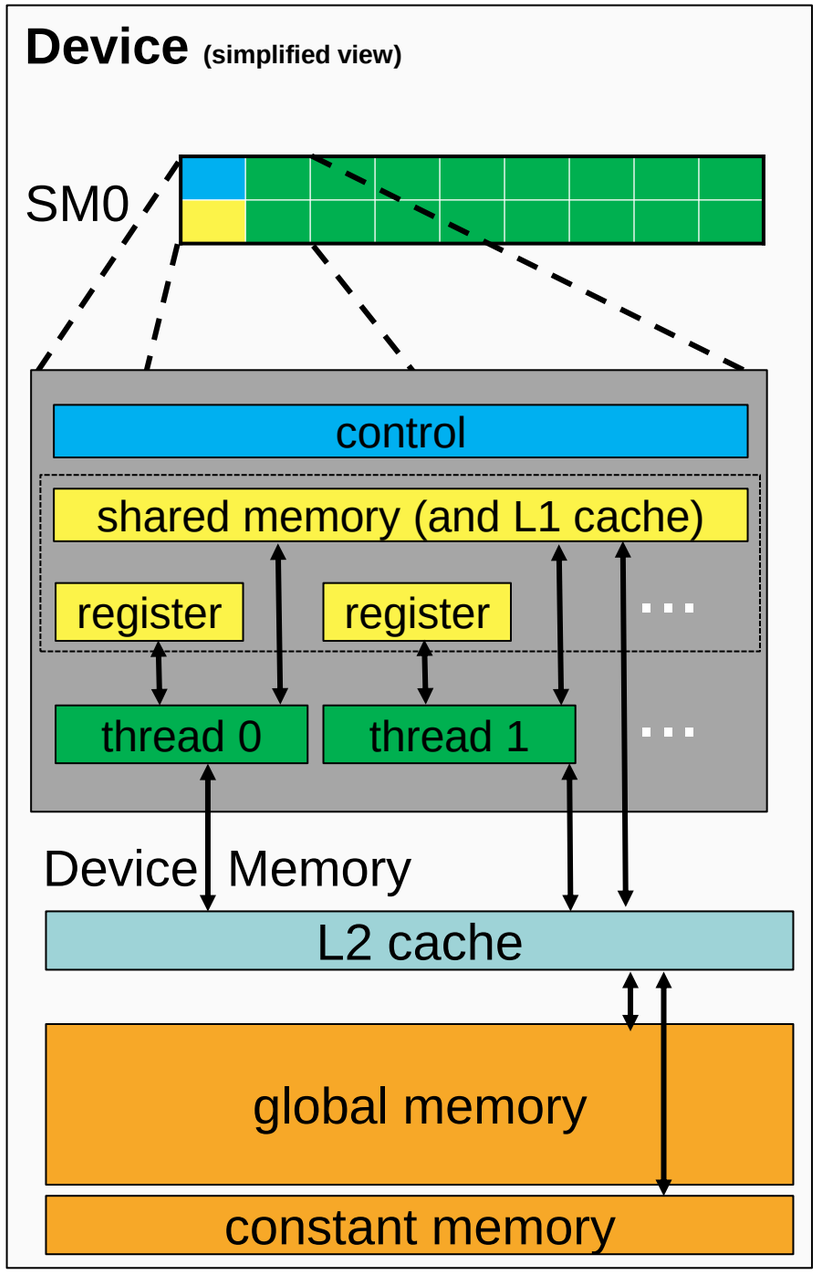
member of
swissuniversities

Parallel Computing
CUDA memory model



CUDA memory model – hardware view

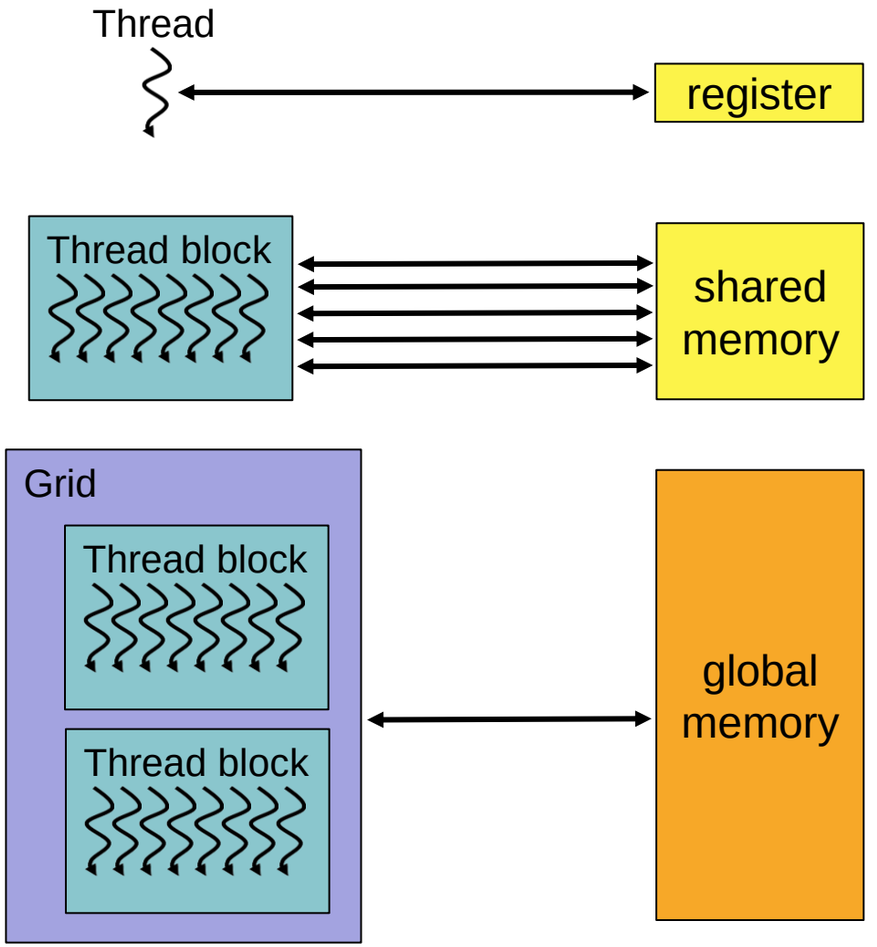
- There are several levels of memory on a GPU
- The closer to the cores the faster the access and the smaller the memory space
- 64K 32bit registers per SM
- 64KB – 164KB shared memory per SM (hardware dependent)
- 1MB – 40MB L2 cache (global – shared for all SM)
- Global memory from 6GB up to 48GB
- Constant memory 64KB but heavily cached (Different caches not drawn on image)



See https://en.wikipedia.org/wiki/CUDA#Version_features_and_specifications for the exact specs of the different architectures

CUDA memory model – software view

- Multiple thread blocks can run on a single SM (The exact allocation cannot be controlled)
- The access to the shared memory is on a per-thread-block basis (secured by CUDA)
- Multiple kernels (different grids) can access the same global memory
- Writing a kernel with **optimized memory access patterns** can lead to enormous performance gain



CUDA memory model – optimized memory patterns

- Since a GPU can compute much faster than load data, we must maximize the arithmetic intensity

$$\text{arithmetic intensity} = \frac{\text{math}}{\text{memory}} = \frac{\text{compute ops per thread}}{\text{time spent on memory per thread}}$$

- Not the total amount of loaded bytes is important
- The total time spent waiting on data to be loaded/stored from/to any kind of memory is important
- We can hide memory access by executing other threads with data ready in the registers
  --> In fact, the whole execution model of CUDA is about hiding memory access by running concurrent threads, remember last week?

CUDA memory model – optimized memory patterns

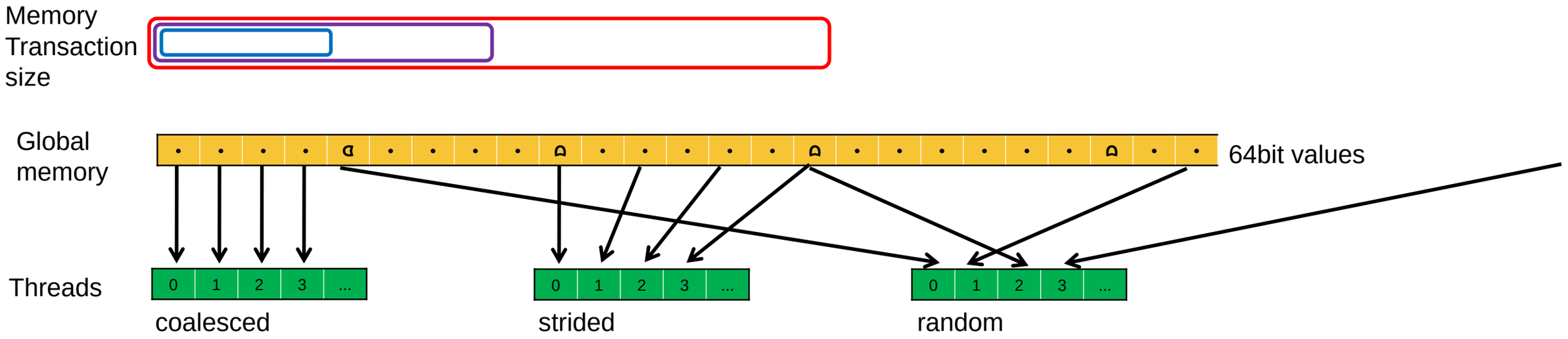
- Example of a naïve matrix multiply kernel

```
1   for (int i=0; i < linewidth: ++i)
2      tmpSum += matrixA[row * linewidth + i] * matrixB[i * linewidth + col]
```

- Compute to global memory access (CGMA) ratio of 1:1 or 1.0
- Assume a global memory bandwidth of 200GB/s → max 50 GFLOPS

CUDA memory model – optimized memory patterns

- Use coalesced global memory access!
- CUDA uses 32-, 64- or 128-byte memory transactions on global memory



07_GlobalMemory

Task:

- Implement the GPU kernel `strided_kernel`
- Implement a loop using a `stride size` from 1 to 32
- Measure the bandwidth for each executed kernel

Link:

- https://classroom.github.com/a/I26sQa6P

Goal:

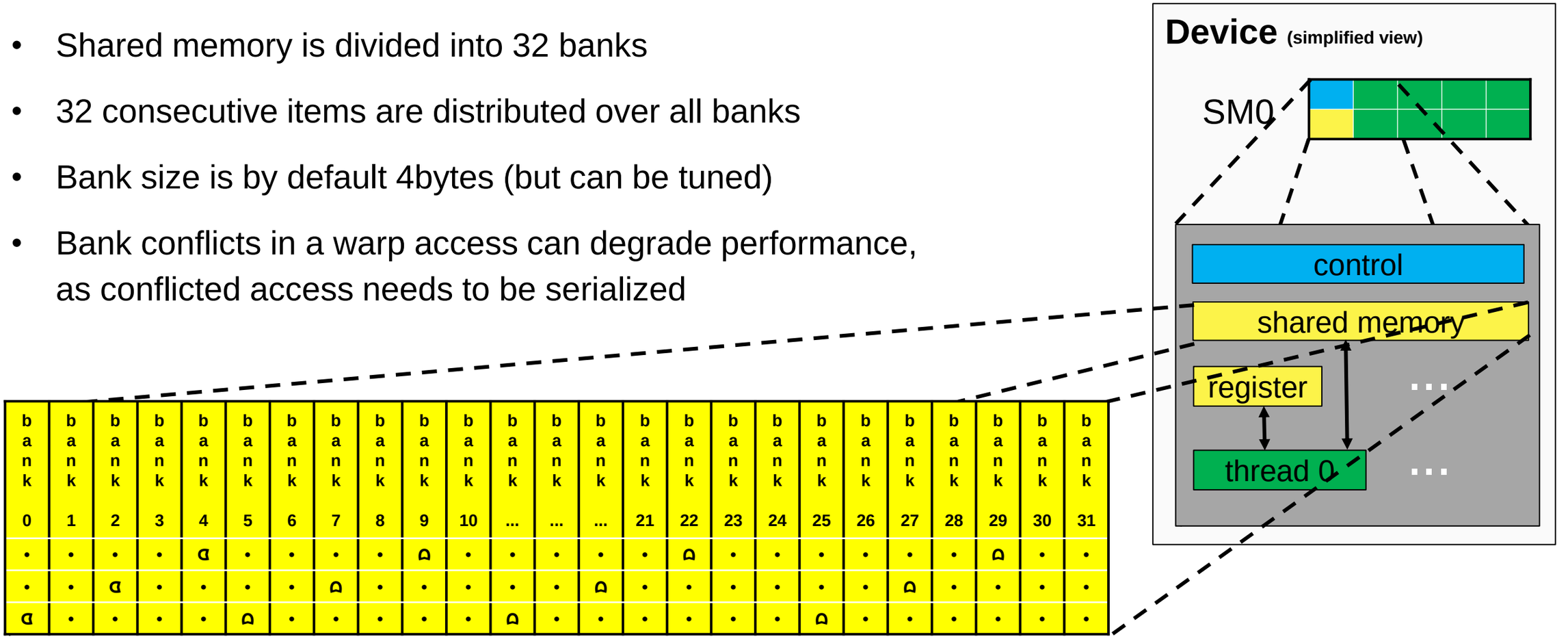
- Learn how to measure GPU kernel execution times
- Basic knowhow about the Nvidia tools

Observations

- The CUDA memory model consists of different sized memory areas
- The closer the memory to the cores, the faster the access
- Coalesced memory access on global memory is 100x faster than random access
- Registers, shared mem and global mem have different functionalities, latencies and bandwidth

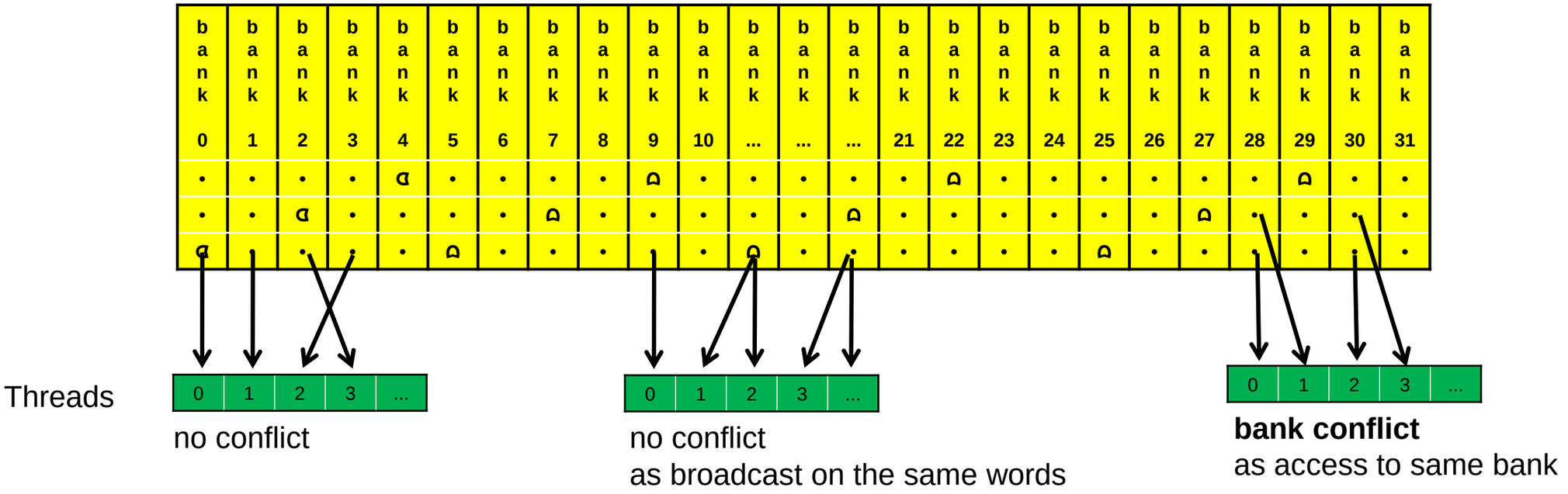
CUDA memory model – shared memory

- Shared memory is divided into 32 banks
- 32 consecutive items are distributed over all banks
- Bank size is by default 4bytes (but can be tuned)
- Bank conflicts in a warp access can degrade performance, as conflicted access needs to be serialized



CUDA memory model – shared memory

- Accessing the same (32-bit) word is done in a broadcast and thus no bank-conflict



CUDA memory model – shared memory and barriers

- Assume a shared memory access over one thread block like this:

```
1  __shared__ int shm_array[1024]                 // Init of constant sized shared memory array
2  shm_array[tIdx] = global_array[threatIdx]      // Load data from global to shared memory
3
4  global_array[threatIdx] = shm_array[1023 - tIdx]  // Revert order
```

- Since the execution order of threads is not defined, race conditions can appear!
- What if not all threads are done with loading the data yet, but some are already on line 4?
- Barriers to the rescue – all threads in that block wait until all of them reach the barrier

```
3  __syncthreads();
```

CUDA memory model – memory as limiting factor for parallelism

- Assume: 1536 Threads/SM,16K 32-bit registers per SM, max 8 blocks/SM, 16KB shared memory
- What is the difference using 10 or 11 registers per Thread

CUDA memory model – memory as limiting factor for parallelism

- Assume: 1536 Threads/SM,16K 32-bit registers per SM, max 8 blocks/SM, 16KB shared memory
- What is the difference of thread blocks of size 16x16, 32x16, 32x32 (NxM) with a kernel like this:

```
1  __shared__ int shmTileA[N*M]
2  __shared__ int shmTileB[N*M]
   __shared__ int shmTileC[N*M]
   for (int i=0; i < width: ++i)
     tmpSum += shmTileA[row * width + i] * shmTileB[i * width + col] + shmTileC[row * width + i]
```

CUDA memory model – memory as limiting factor for parallelism

- There are ways to help the compiler to choose the “right” number of registers to use
  - Less register = more data load from and to global memory
  - This data can and will be cached on L1 and L2
- Useful if register pressure occurs
- Try to get more blocks running by reducing the number of registers per thread

```
1  __global__ void
2  __launch_bounds__(MAX_THREADS_PER_BLOCK, MIN_BLOCKS_PER_MP)
3  pacKernel(float* matrixA, float* matrixB, float* matrixC, int size)
4  {...}
```

- Or use `--maxrregcount` as compile argument

07_MovingSum

Task:

- Implement the GPU kernel `movingSumSharedMemStatic`
- Implement the GPU kernel `movingSumSharedMemDynamic`
- Implement the GPU kernel `movingSumAtomics`

Link:

- https://classroom.github.com/a/eUW6n8_r

Goal:

- Learn how to use shared memory
- Learn how to use atomic operations

Observations

- Use `__shared__` to declare a variable/array in shared memory
- Shared memory acts as a user managed L1 cache
- Race conditions can happen on shared memory as well as on global memory
- Barriers like `__syncthreads()` are needed to control the race conditions
- Details of the hardware implementation need to be known to avoid bad memory patterns