Heterogeneous computing with performance modelling

More GPU programming basics

Mirko Myllykoski
mirkom@cs.umu.se

Department of Computing Science and HPC2N
Umeå University

4-5. November 2020

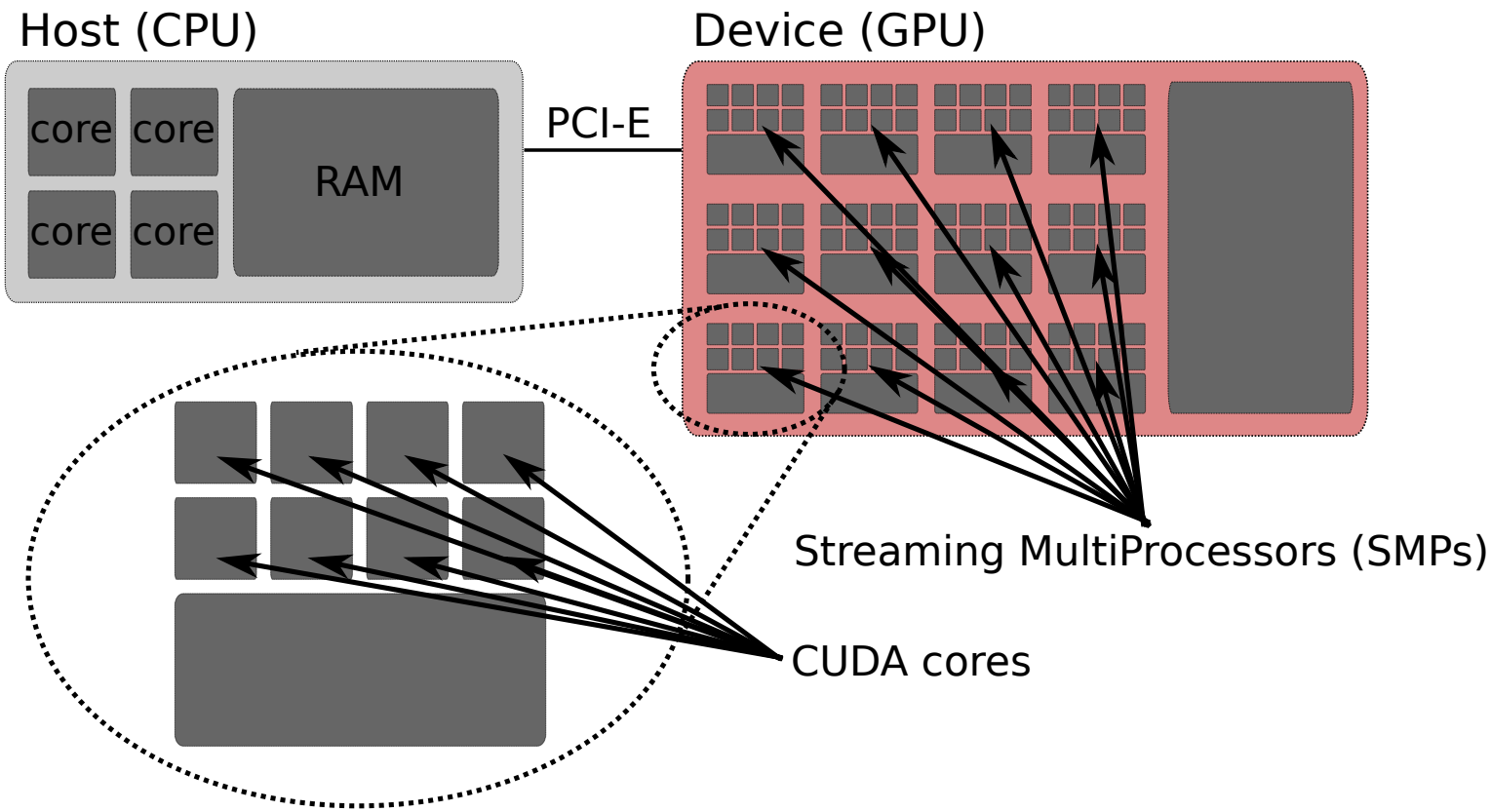
UMEÅ UNIVERSITY

Compute capability

- Different GPUs have different architecture and capabilities.
- Nvidia uses **Compute Capabilities** to enumerate the differences:
  - 3.x → Kepler, GTX 700 series, Tesla K80
  - 5.x → Maxwell, GTX 900 series
  - 6.x → Pascal, GXT 1000 series
  - 7.x → Volta, **Tesla V100**
  - 7.5 → Turing, RTX 2000 series
  - 8.x → Ampere, RTX 300 series, Tesla A100
- During this course, we are consentrating on 7.0 (Volta).
  - However, we are not going to discuss any fancy optimizations that are only 7.0 specific.
- Every new CUDA version introduces new functionality.
  - Some older GPUs do not support everything.
  - Some GPUs require customized code.

Recap

- Let's return back to the earlier figure...

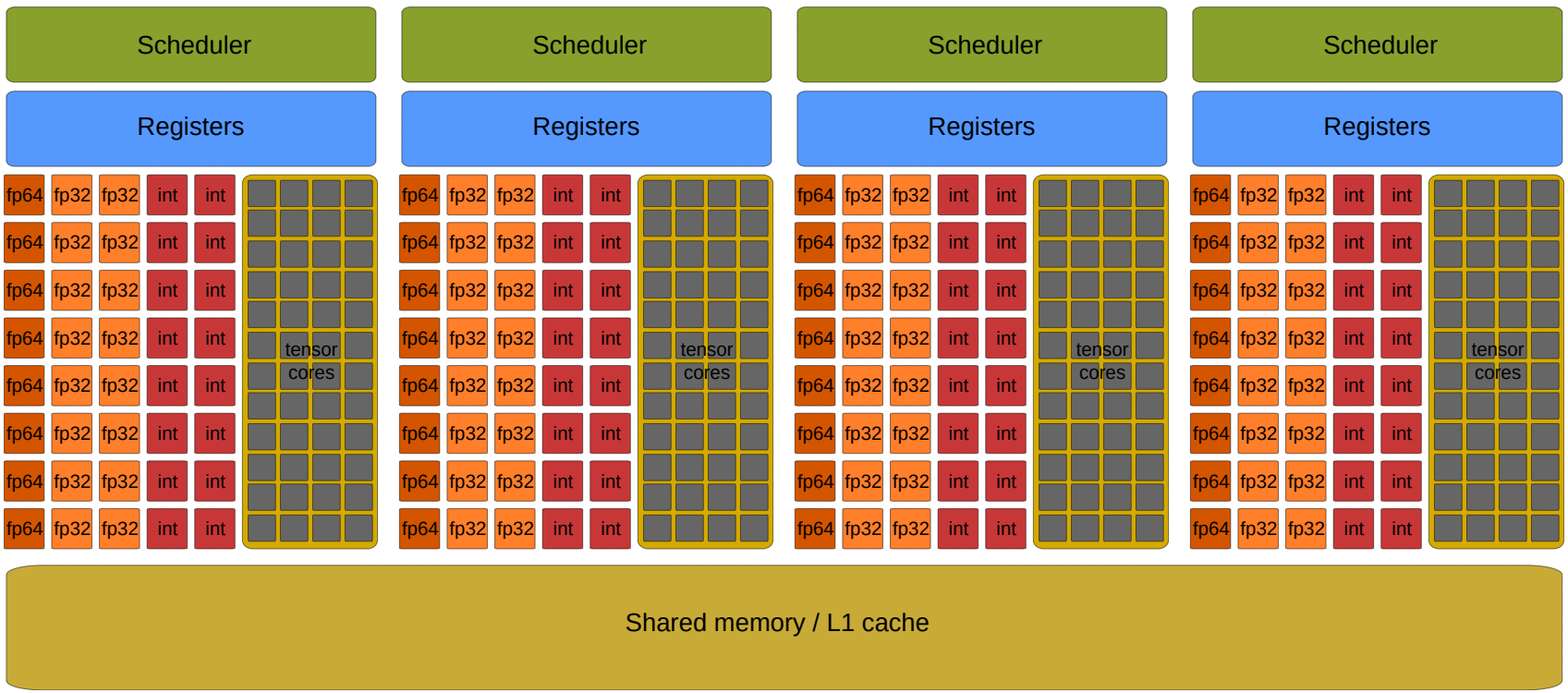


Recap

- A GPU contains several Streaming MultiProcessors (SMPs).
- Each SMP contains several CUDA cores.
- The threads are divided into thread blocks.
  - Each thread block is mapped to a single SMP.
  - A SMP can have multiple thread blocks mapped to it.
- Simultaneous hardware multithreading.
  - Each CUDA core can execute several threads simultaneously.
  - A scheduler select the next instruction among a pool of active threads.

Streaming MultiProcessors

- In reality, a SMP looks something like this:



Streaming MultiProcessors (processing blocks)

- Each SMP is divided into multiple **processing blocks**.
- Each processing block contains the following:

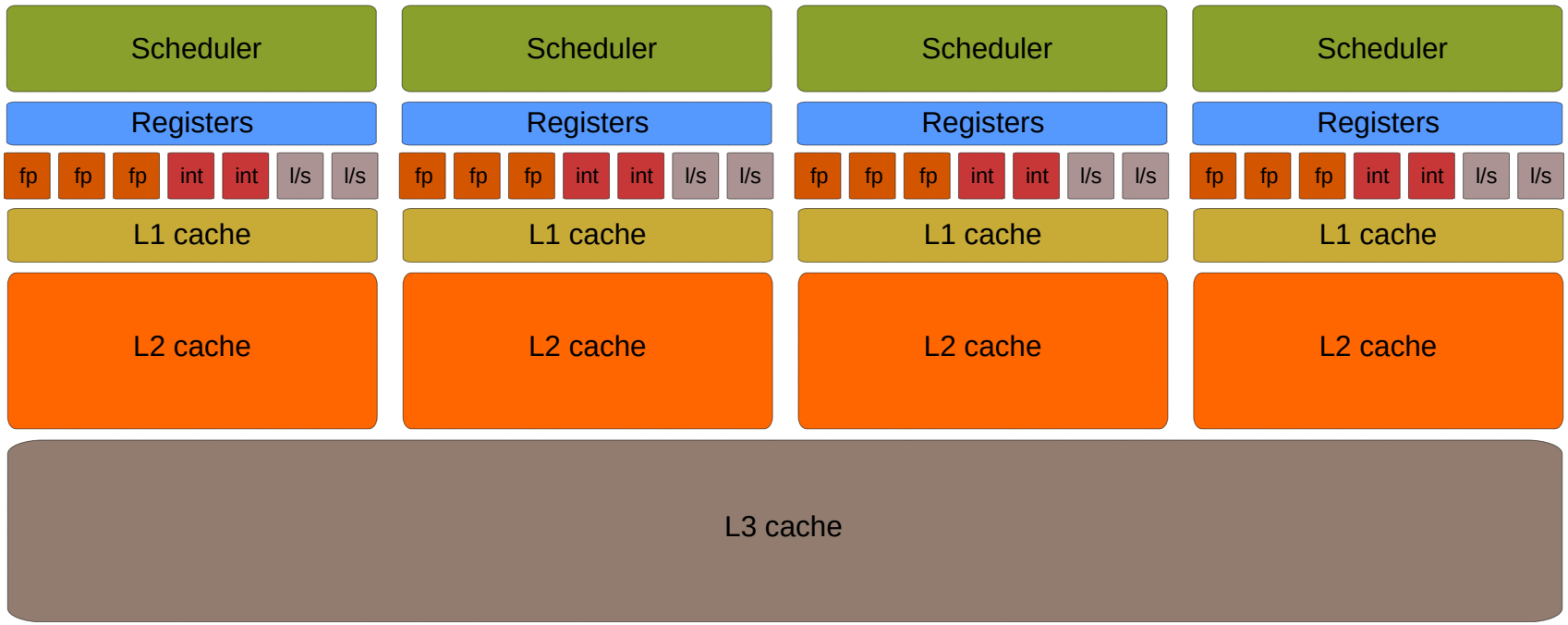
  CUDA cores Large number of cores for various operations (integer, 32-bit floating point, 64-bit floating point, reduced precision, special functions, etc).

  Scheduler During each cycle, a scheduler selects a set of threads and issues an instruction to the cores.

  Register file All threads share a large pool of registers.
- All processing blocks share a combined shared memory and L1 data cache.
- Left out from the figure: dispatch units, L0/L1 instruction caches, load/store units, texture units, ...

Streaming MultiProcessors (compared to CPU)

- This is a drastically different approach compared to CPUs.
  - Each CPU core contains its **own** scheduler, registers, caches, ...
  - Each CPU core contains several execution ports but these execution ports are not generally referred to as cores.

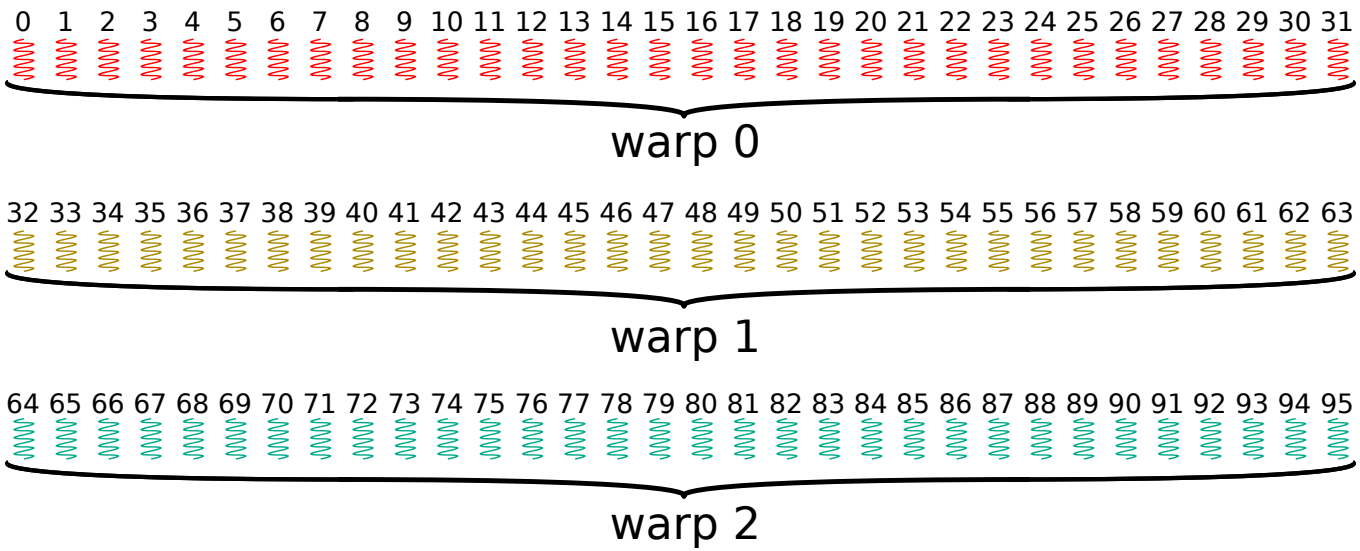


Streaming MultiProcessors (summary)

- In general, GPUs allocate
  - more silicon to units that compute, and
  - less silicon to units that direct computation.
- More computing power in the same form factor.
- Modular approach: SMPs, processing blocks, CUDA cores.
  - Easier to scale to thousands of cores.
- However, the **CUDA cores share a lot of resources**.
  - Should we call CUDA cores cores?
  - Are CUDA cores just execution ports?
  - What are the **limitations**?

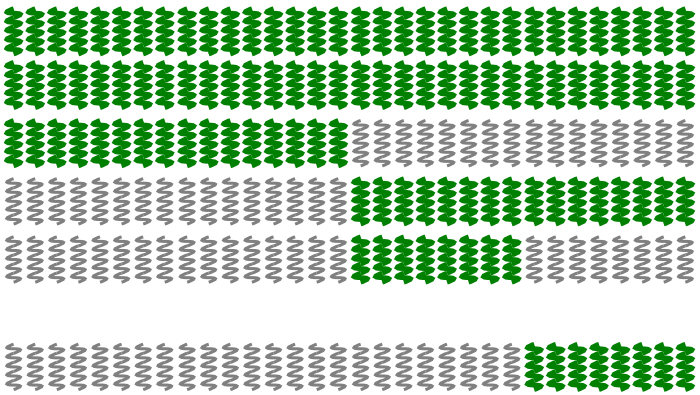
Warps

- Underneath, each thread blocks is subdivided into **warps**.
- Each warps consist of 32 sequentially numbered threads.
  - Given a thread block of the size (Dx, Dy, Dz), the hardware indexes a thread of index (x, y, z) as (x + y*Dx + z*Dx*Dy).
  - As a rule of thumb, threads that are adjacent in the `x` dimension belong to the same warp.



Warps (warps scheduling)

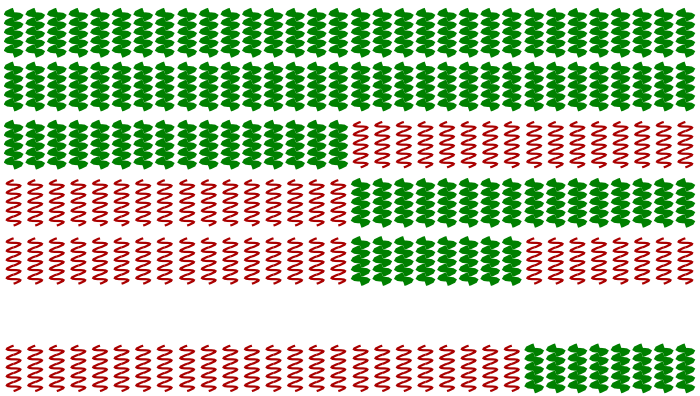
- The warps are distributed among the processing blocks.
- During each instruction issue time, the scheduler
  - picks a warp that is ready to execute an instruction and
  - issues the instruction to a set of CUDA cores.
  - The number of instructions issued depends on the architecture.
- All threads in a warp **execute the same instruction!**
- "Non-contributing" threads are disabled:



```
printf("Everyone\n");
if (threadIdx.x < 16)
    printf("Less than 16.\n");
else if (threadIdx.x < 24)
    printf("Between 16 and 23\n");
else
    printf("Larger than 23\n");
```

Warps (disabled threads)

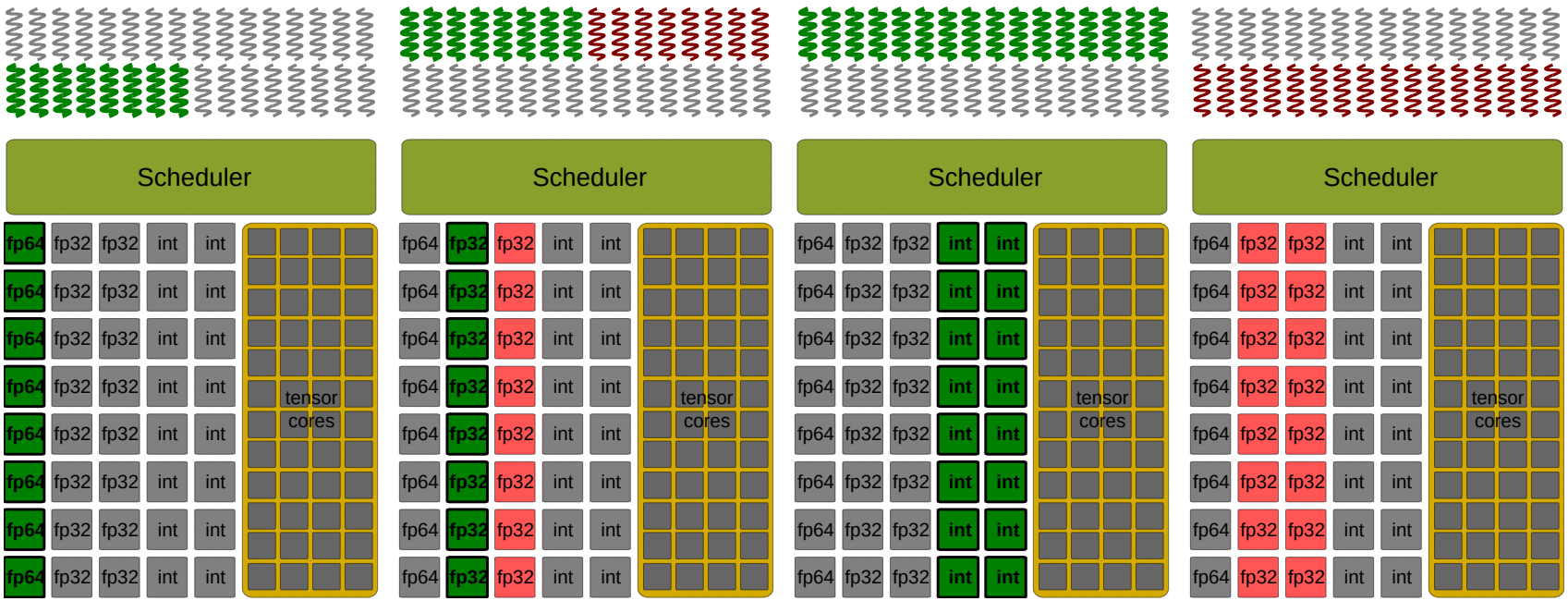
- Even though some threads are disabled, the overall cost is still almost as high as if all threads in the warp had executed **all diverging paths**:



```
printf("Everyone\n");
if (threadIdx.x < 16)
    printf("Less than 16.\n");
else if (threadIdx.x < 24)
    printf("Between 16 and 23\n");
else
    printf("Larger than 23\n");
```

Warps (disabled threads)

- The disabled threads do not trigger memory transfers etc, but the associated resources (cores) are not being utilized:

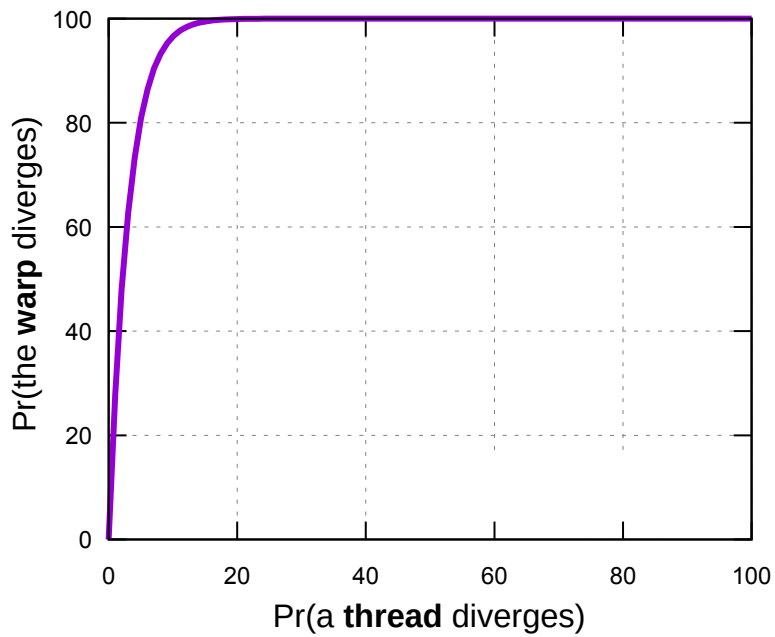


- Above, the warps are issued over two or four cycles in half-warps (int, fp32) or quarter-warps (fp64), receptively.

Warps (diverging execution paths)

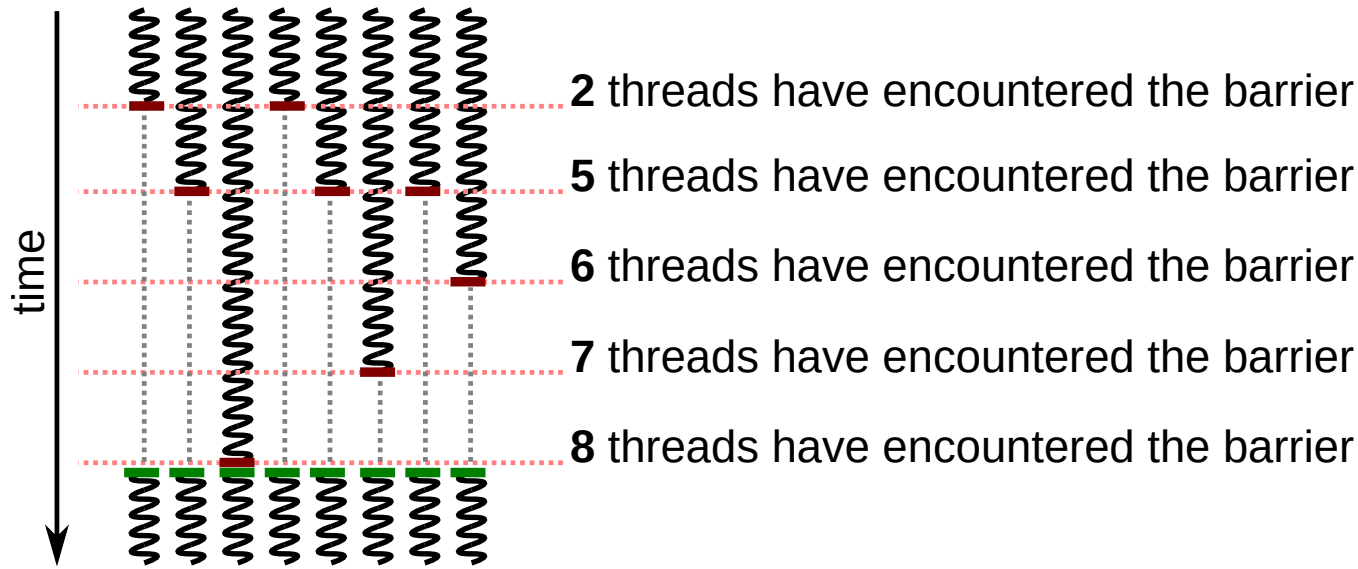
- The diverging execution paths can be a bigger problem that expected.
- If a thread diverges with the probability $p \in [0, 1]$, then probability that at least one thread within a warp diverges is

$$1 - (1 - p)^{32}.$$



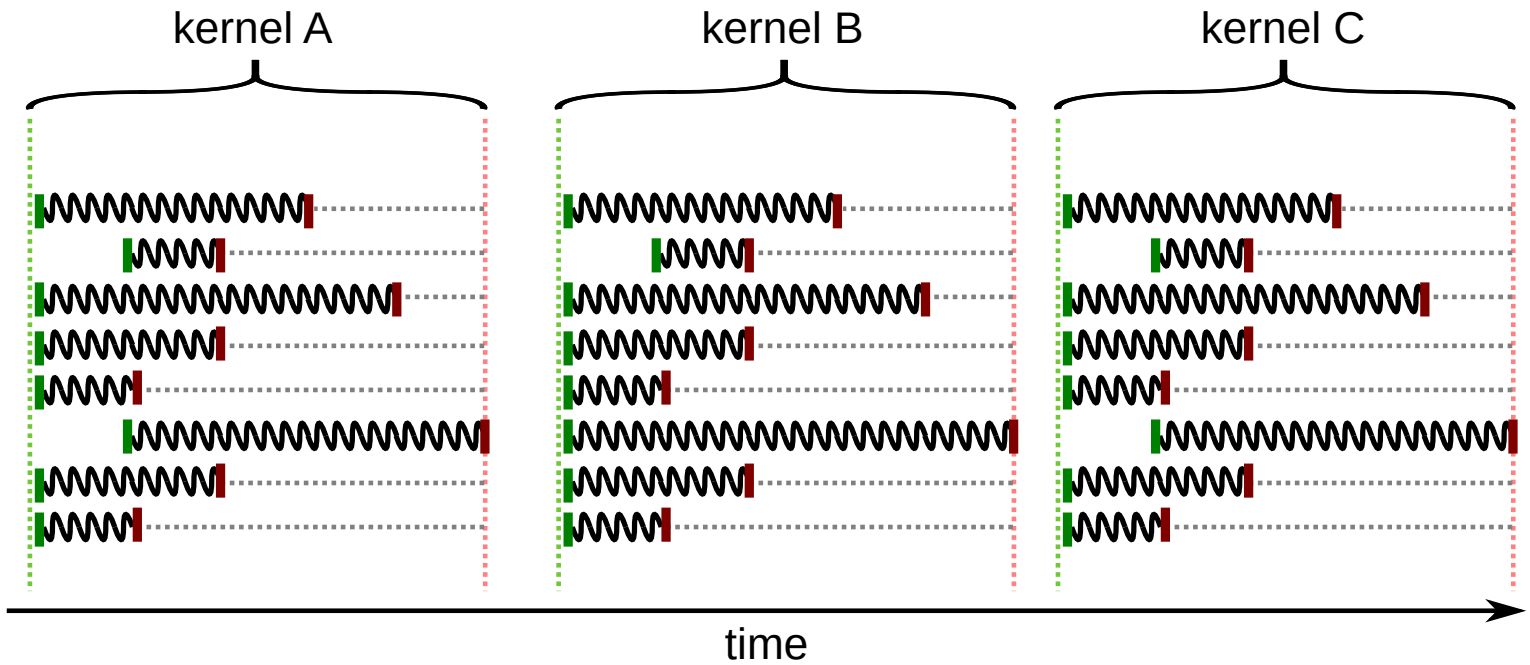
Synchronization

- It is often necessary to **synchronize** the threads.
  - We want to be sure that specific operations have been completed.
- One of the simplest approaches is to create a **barrier**.
  - All threads must encounter the barrier.
  - Threads that have encountered the barrier **wait until all threads have encountered the barrier**.



Synchronization (globally)

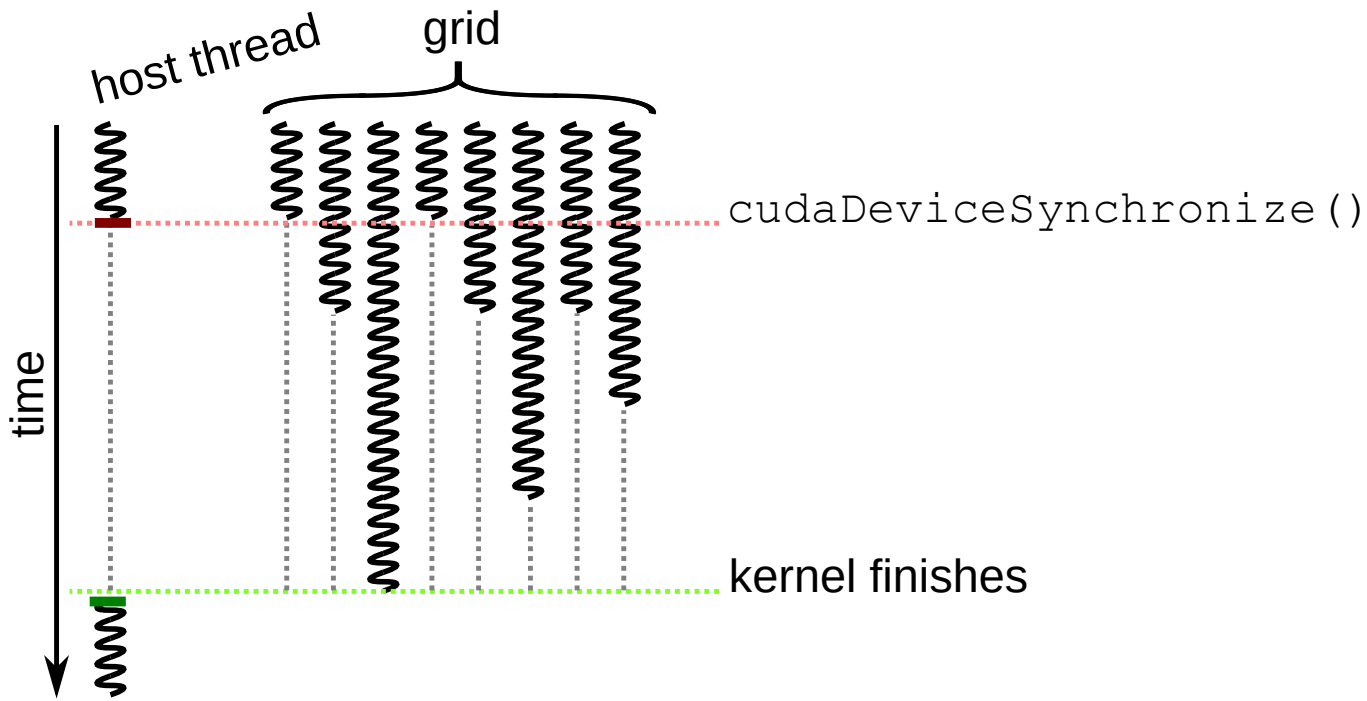
- The maximum lifetime of a thread is the same as the lifetime of the corresponding kernel.
  - All threads in a grid are synchronized at **the beginning and the end of a kernel**.



Synchronization (globally)

- The **host thread** and the grid are synchronized with:

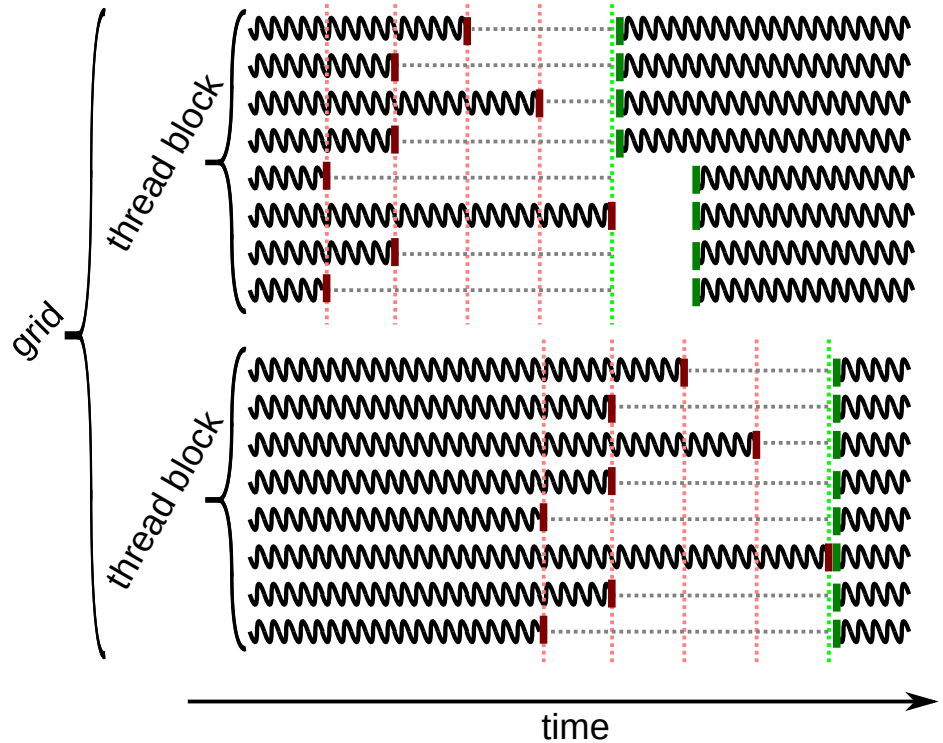
```
__host__ __device__ cudaError_t cudaDeviceSynchronize ( void )
```



Synchronization (thread block)

- Threads that belong to the **same thread block** are synchronized with:

```
__device__ void __syncthreads ( void )
```



Synchronization (warp)

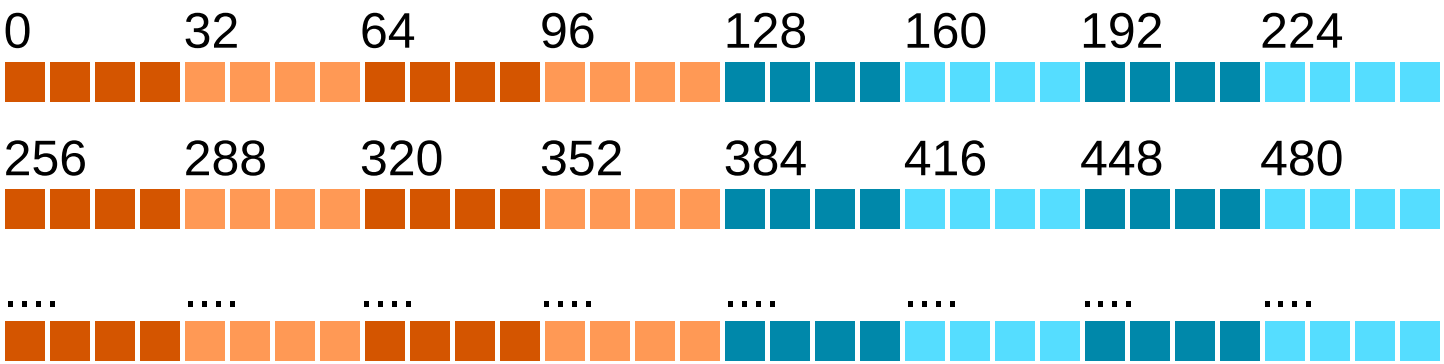
- Threads that belong to the **same warp** are synchronized with:

```
__device__ void __syncwarp(unsigned mask=0xffffffff)
```

- Guarantees memory ordering among threads participating in the barrier.
- Limited usefulness with pre-Volta GPUs.
- Sometimes necessary with Volta and post-Volta GPUs.
  - See *Independent Thread Scheduling*.

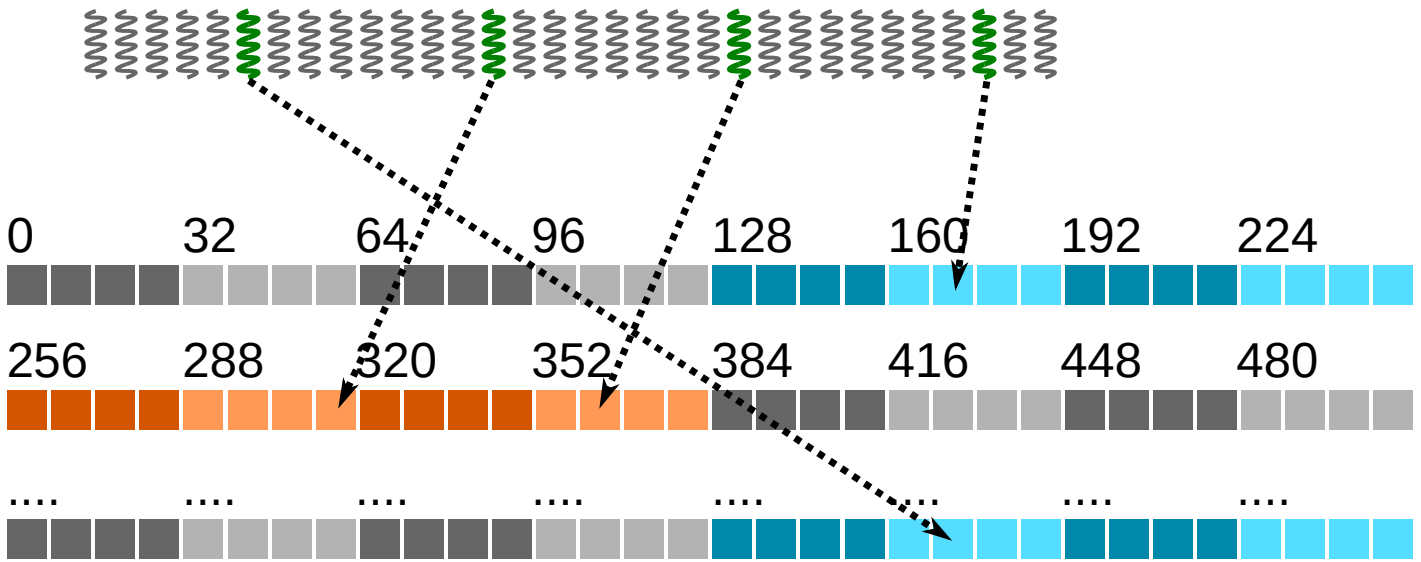
Global memory

- Memory requests pass through L1 and L2 caches.
  - A **cache line is 128 bytes** (32 floats, 16 doubles) and maps to a 128 byte aligned segment of global memory.
  - Unless L1 cache is explicitly disabled in which case the cache line is 32 bytes and maps to a 32 byte aligned segment of global memory.



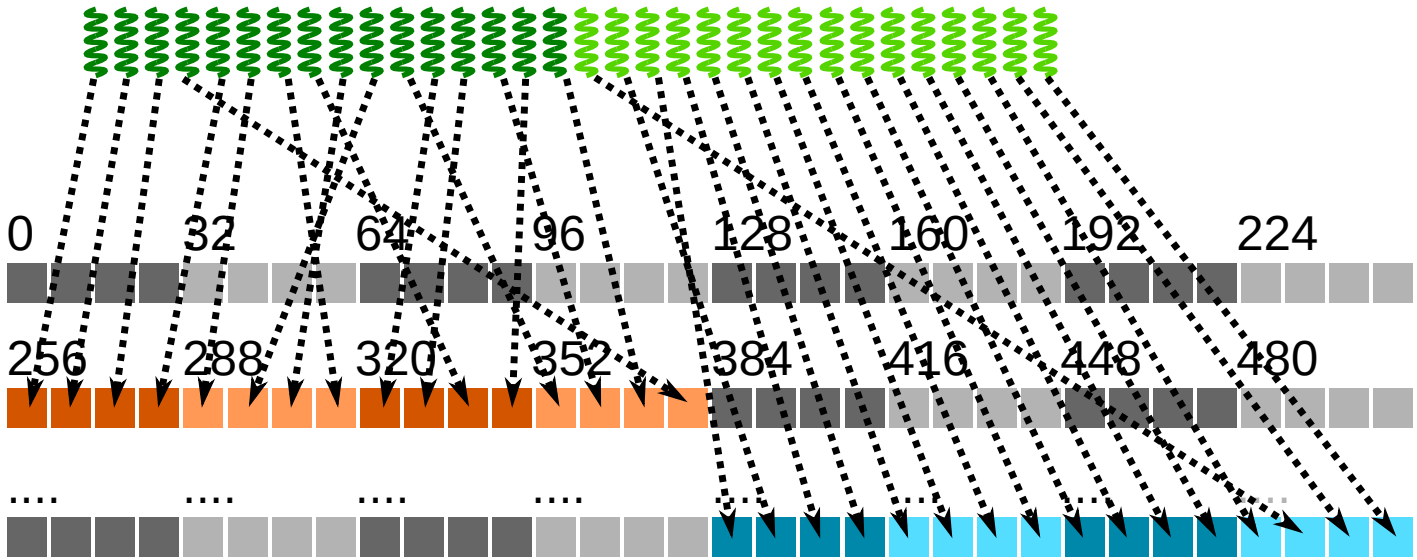
Global memory (cache lines)

- A warp accesses the memory together.
- If a single thread in a warp accesses a memory address, then the **entire cache line is loaded**:



Global memory (optimal access)

- For optimal performance, a warp should access **adjacent memory locations** that span across as **few cache lines** as possible:

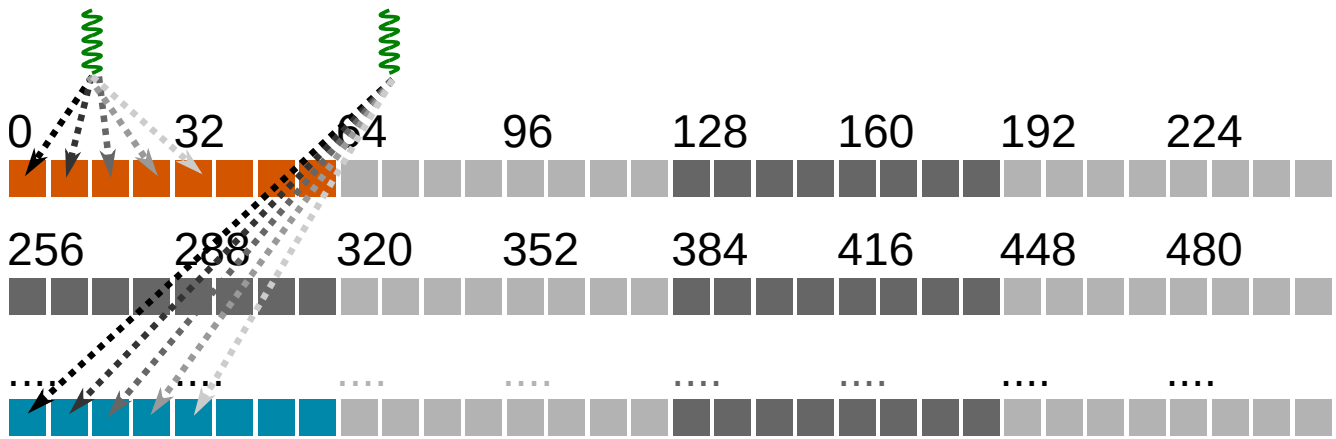


Global memory (CPU versus GPU)

- The following access patter would work well on a CPU:

```
#pragma omp parallel for schedule(static, 64)
for (int i = 0; i < N; i++)
    x[i] = alpha * x[i];
```

- Each core accesses a different cache line, no false sharing.

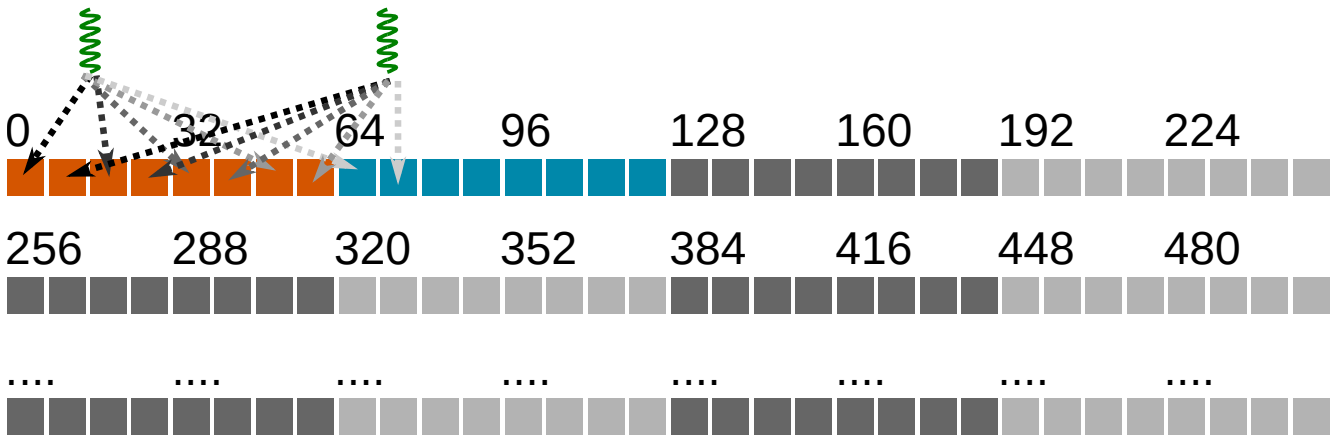


Global memory (CPU versus GPU)

- The following access patter would work horribly on a CPU:

```
#pragma omp parallel for schedule(static, 1)
for (int i = 0; i < N; i++)
    x[i] = alpha * x[i];
```

- Cores access the same cache line, **false sharing**.

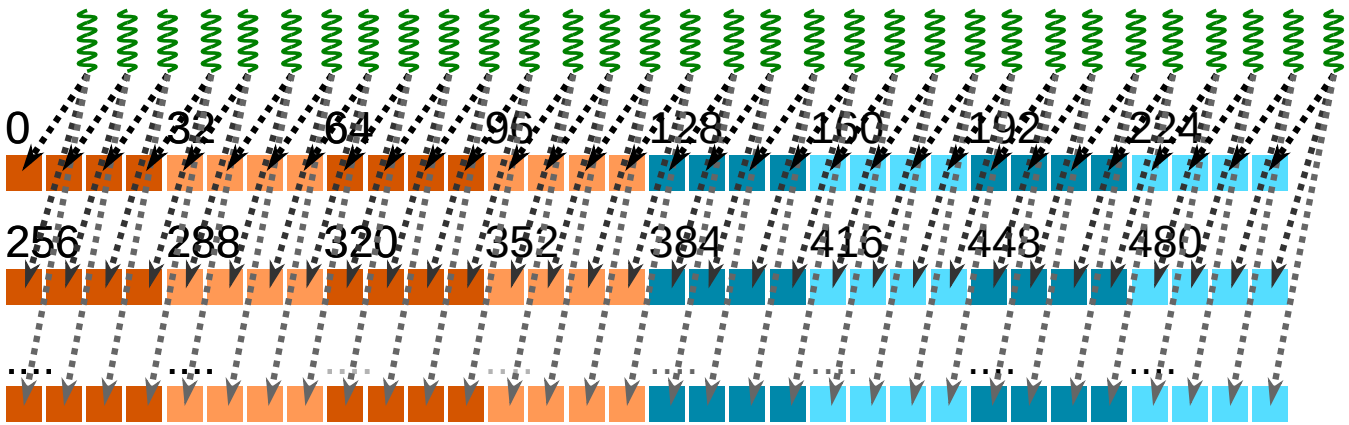


Global memory (CPU versus GPU)

- The following access patter would work well on a GPU:

```
for (int i = threadIdx.x; i < N; i += blockDim.x)
    x[i] = alpha * x[i];
```

- Warp loads a minimal number of cache lines, each cache line is accessed completely.

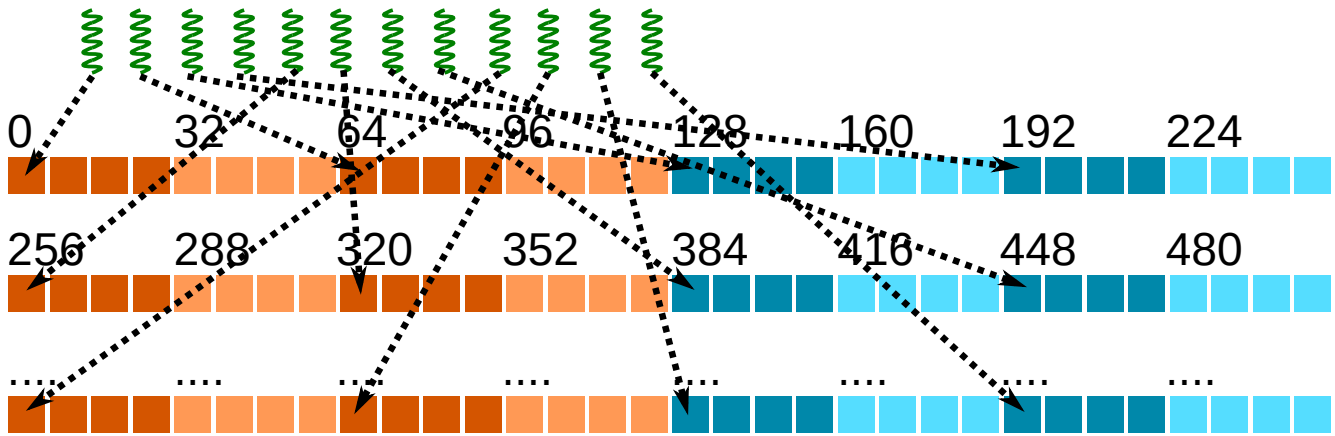


Global memory (CPU versus GPU)

- The following access patter would work horribly on a GPU:

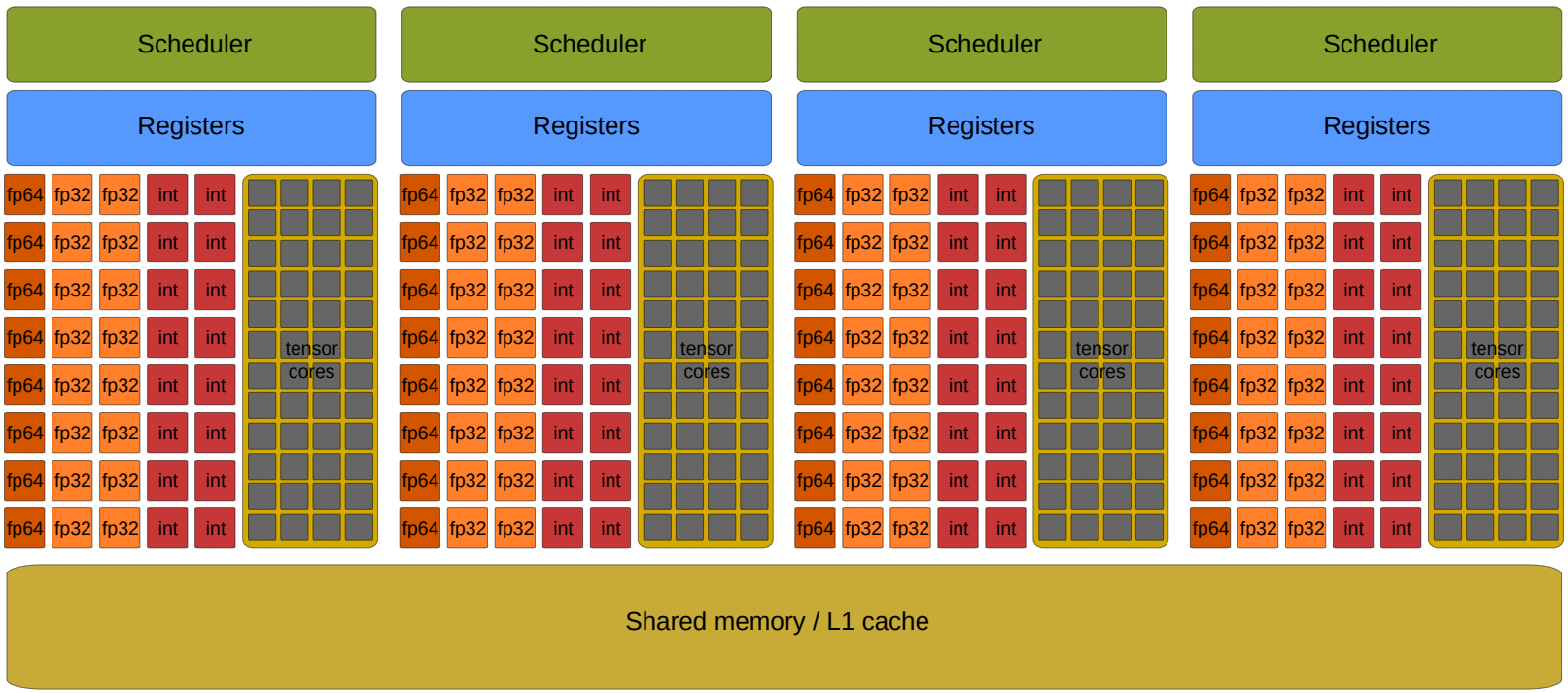
```
for (int i = 0; i < M; i++)
    x[threadIdx.x*M+i] = alpha * x[threadIdx.x*M+i];
```

- Warp loads a large number of cache lines, only **two words** are accessed from each cache line.



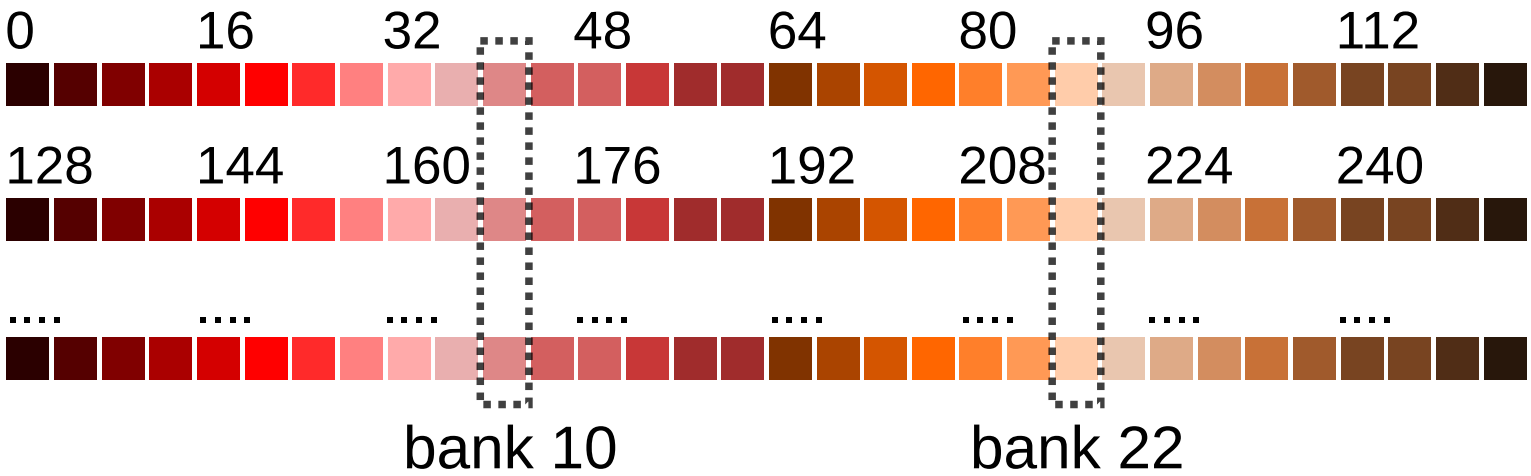
Shared memory

- Each SMP has a fast on-chip memory (128 KB Volta) that is divided between a L1 data cache and a **shared memory**.
- A portion of the shared memory can be allocated for a thread block.



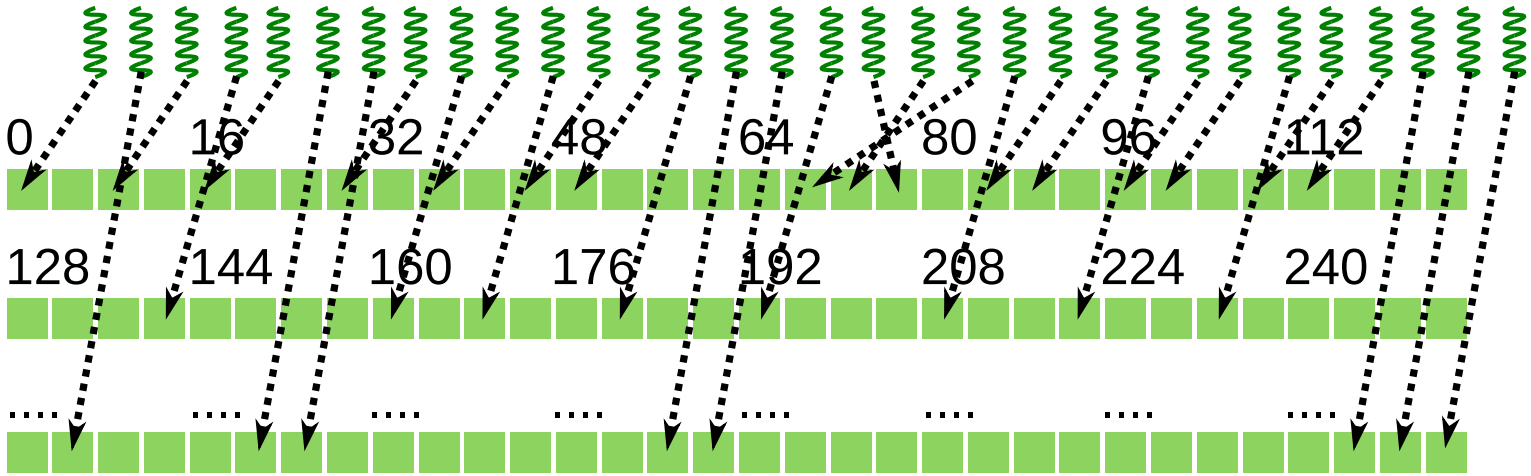
Memory (shared memory)

- Shared memory is significantly **faster** than the global memory.
- However, it is divided into 32 memory banks.
- Successive 4-byte words map to successive banks (Volta).
- Each bank has a bandwidth of 4-bytes per clock cycle.
  - Simultaneous access to the same bank causes a **bank conflict**.
  - Conflicting memory requests are served sequentially.



Memory (shared memory)

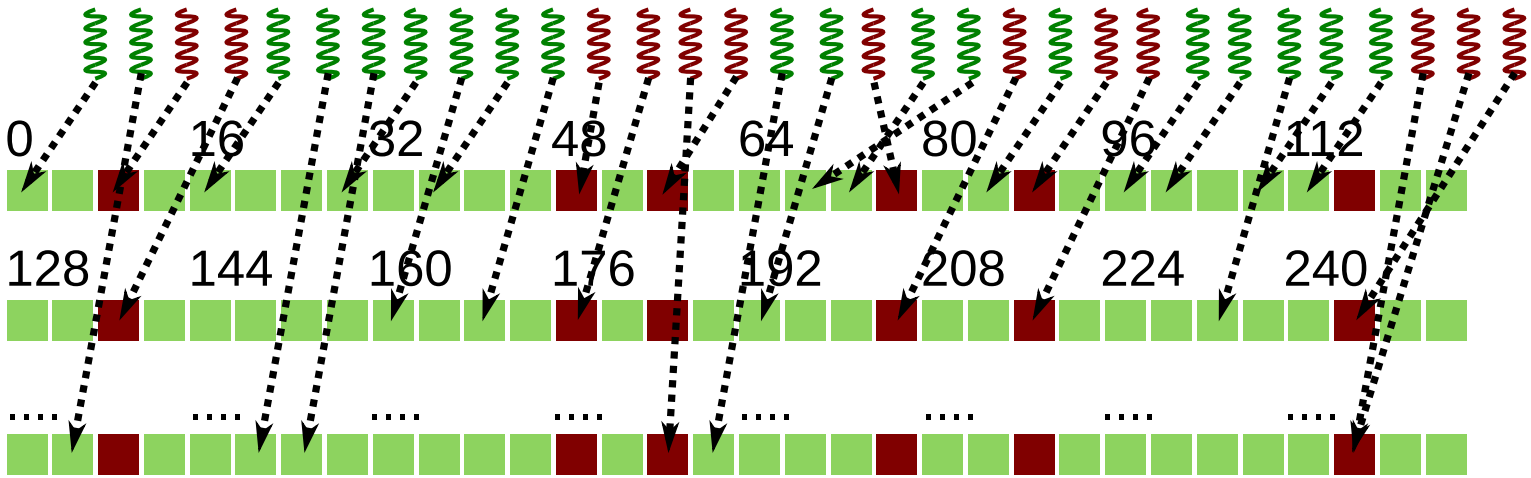
- An example where each memory bank receives one request:



- No bank conflicts, optimal bandwidth.

Shared memory (bank conflicts)

- An example where some memory banks receive multiple requests:



- Several **two-way bank conflicts**, two sequential transfers, effective bandwidth cut to half.

Shared memory (allocation)

- Shared memory can be allocated either statically or dynamically.
- **Static** allocation:

```
__global__ void kernel(....)
{
    __shared__ int x[256];

    ....
}
```

- **Dynamic** allocation:

```
__global__ void kernel(....)
{
    extern __shared__ int x[];

    ....
}

kernel<<<blocks, threads, 256*sizeof(int)>>>kernel(....);
```

Shared memory (example)

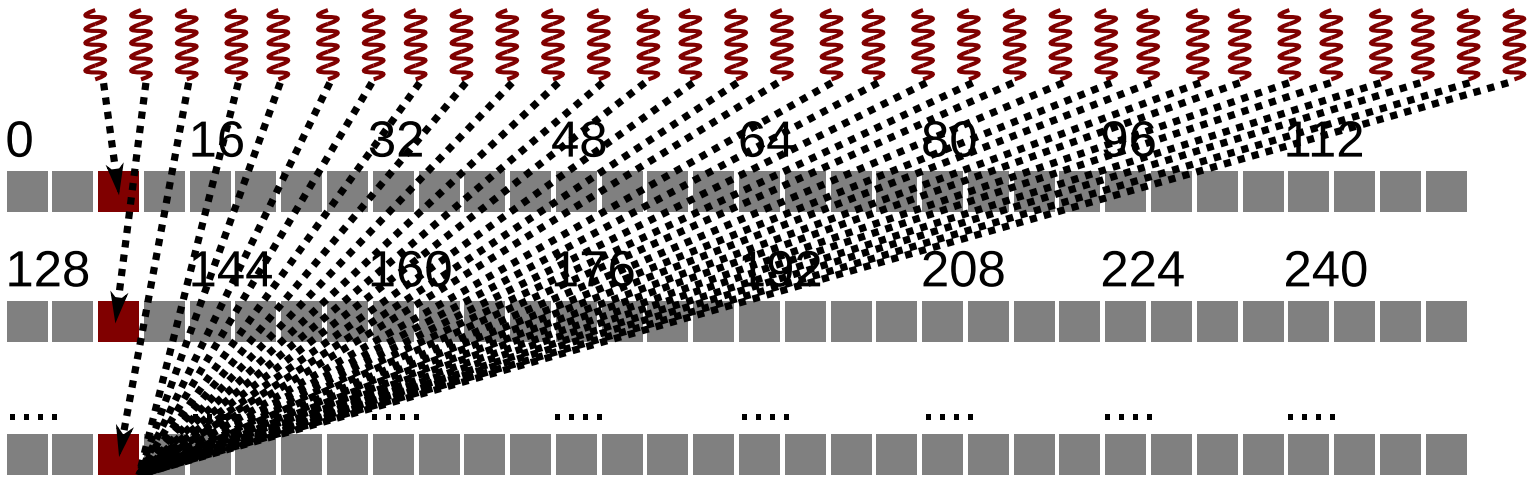
- Shared memory is typically used when the threads in the same thread block need to **communicate**.
- Imagine the following transpose operation:

```
__global__ void small_transpose(float A[32][32])
{
    // we are assuming that the thread block size is 32 x 32
    __shared__ float tmp[32][32];

    // each thread loads a matrix element
    tmp[threadIdx.y][threadIdx.x] = A[threadIdx.y][threadIdx.x];

    // each thread waits until all other threads are ready
    __syncthreads();

    // each thread stores a matrix element (note the swapped dimensions)
    A[threadIdx.y][threadIdx.x] = tmp[threadIdx.x][threadIdx.y];
}
```

Shared memory (bad access pattern)

- The second access to the `tmp` array causes **32-way bank conflict** but the global memory is accessed optimally:

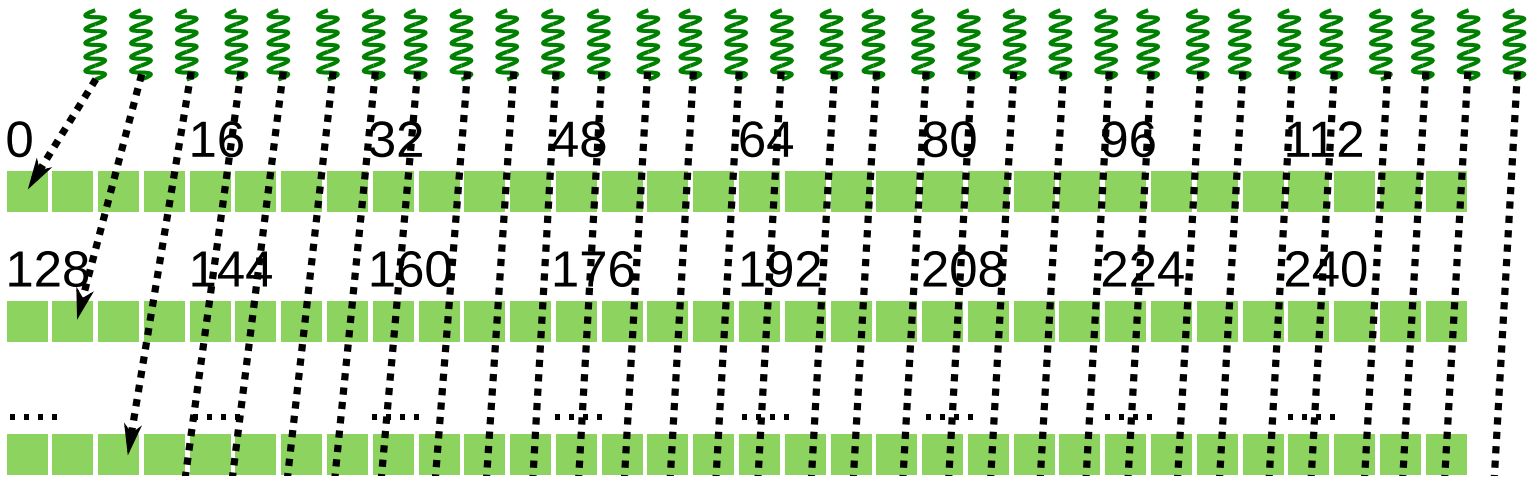
```
__global__ void small_transpose(float A[32][32])
{
    // we are assuming that the thread block size is 32 x 32
    __shared__ float tmp[32][32];

    ....

    // each thread stores a matrix element (note the swapped dimensions)
    A[threadIdx.y][threadIdx.x] = tmp[threadIdx.x][threadIdx.y];
}
```



Shared memory (optimal access pattern)

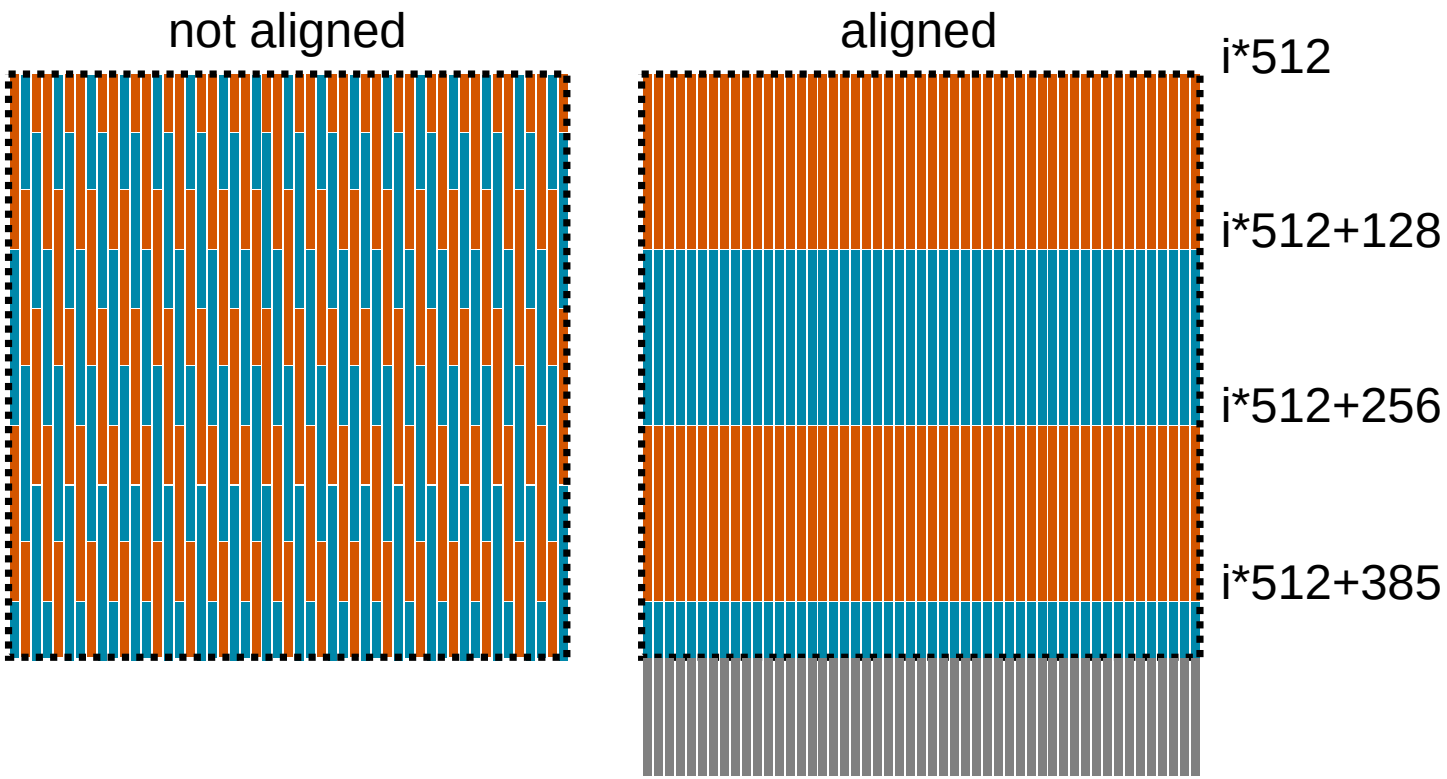
- This can be fixed quite easily:

```
__global__ void small_transpose(float A[32][32])
{
    // we are assuming that the thread block size is 32 x 32
    __shared__ float tmp[32][33]; // <= *** note 33 ***

    ....

    // each thread stores a matrix element (note the swapped dimensions)
    A[threadIdx.y][threadIdx.x] = tmp[threadIdx.x][threadIdx.y];
}
```



Matrices

- `cudaMalloc` and `cudaMallocManaged` **align** memory to 256 bytes (two 128-byte cache lines).
- Matrices (two-dimensional arrays) require special attention since **each column should also be aligned** (assuming column-major ordering):



Matrices (manual allocation)

- The alignment can be done manually:

```
// a function that returns the ceil of a/b. That is,
//     DIVCEIL(5, 2) = ceil(5/2) = ceil(2.5) = 3.
static int DIVCEIL(int a, int b)
{
    return (a+b-1)/b;
}

...

// allocate 256-byte aligned m x n matrix (m rows, n columns)
double *A;
int ldA = DIVCEIL(m, 256/sizeof(double))*(256/sizeof(double));
cudaMalloc(&A, n*ldA*sizeof(double));
```

- This is same as

```
// allocate 256-byte aligned m x n matrix (m rows, n columns)
double *A;
int ldA = DIVCEIL(m, 32)*32;
cudaMalloc(&A, n*ldA*sizeof(double));
```

Matrices (cudaMallocPitch)

- Or using the `cudaMallocPitch` function:

```
__host__ cudaError_t cudaMallocPitch (
    void ** devPtr,
    size_t * pitch,
    size_t width,
    size_t height
)
```

- `pitch` is the leading dimension. Both `pitch` and `width` are given in **bytes**.
- Note that the `cudaMallocPitch` function assumes the that matrix in stored in row-major format. Therefore, you should do the following when allocating in column-major format:

```
double *A; int ldA;
{
    size_t pitch;
    cudaMallocPitch(&A, &pitch, m*sizeof(double), n);
    ldA = pitch/sizeof(double);
}
```

Matrices (transfers)

- A matrix can be transferred with the `cudaMemcpy2D` function:

```
__host__ cudaError_t cudaMemcpy2D (
    void * dst,
    size_t dpitch,
    const void * src,
    size_t spitch,
    size_t width,
    size_t height,
    enum cudaMemcpyKind kind
)
```

- Assuming we have
  - a matrix `A` with the leading dimension `ldA` in the host memory and
  - a matrix `dA` with the leading dimension `ld_dA` in the global memory:

```
cudaMemcpy2D(d_A, ld_dA*sizeof(double), A, ldA*sizeof(double),
    m*sizeof(double), n, cudaMemcpyHostToDevice)
```

Hands-ons

- Two hands-ons under `hands-ons/2.intermediate`:
  - 1.sum Learn how to use shared memory and synchronize threads. Learn how to sum together the elements of a vector.
  - 2.gemv Learn how to use shared memory, synchronize threads and handle matrices. Learn how to perform a matrix-vector multiplication.
- Solutions can be found under `solutions/2.intermediate`.