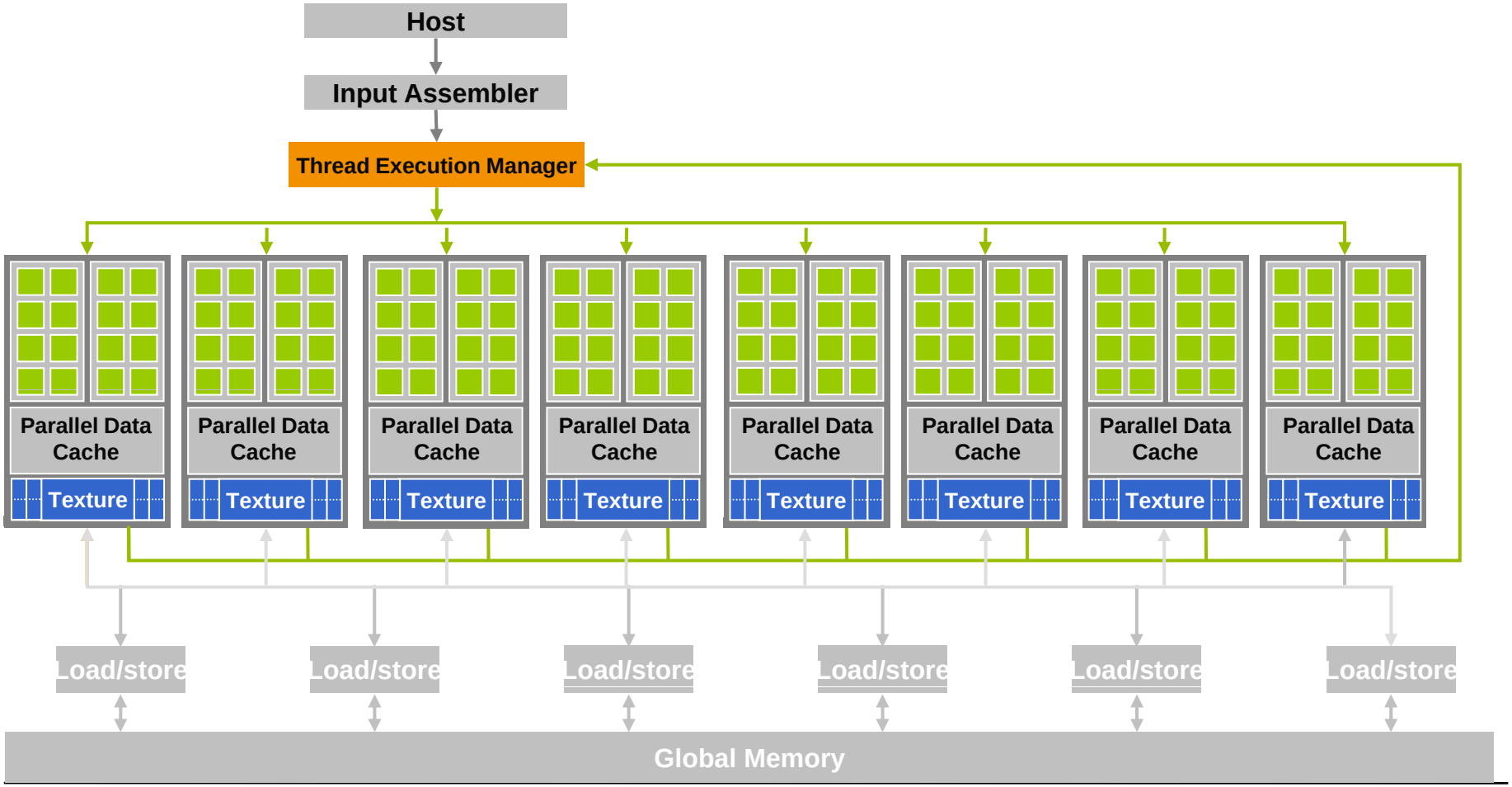
Ph.D. School in Scientific GPU Computing

Hendrik Lensch

CUDA Programming Model

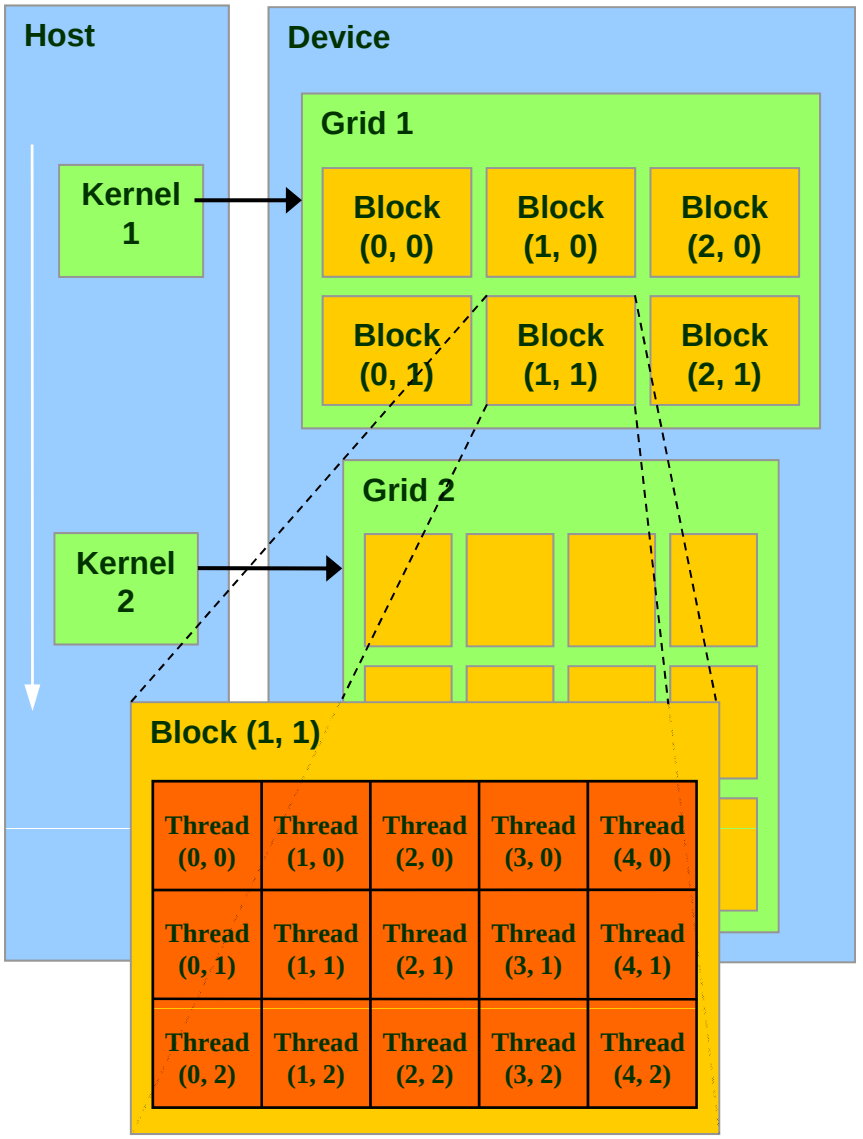
GeForce 8800

16 highly threaded SM's, >128 FPU's, 367 GFLOPS, 768 MB DRAM, 86.4 GB/S Mem BW, 4GB/S BW to CPU



Thread Batching: Grids and Blocks

- **A kernel is executed as a grid of thread blocks**
  - All threads share data memory space
- **A thread block is a batch of threads that can cooperate with each other by:**
  - Synchronizing their execution
    - For hazard-free shared memory accesses
  - Efficiently sharing data through a low latency shared memory
- **Two threads from two different blocks cannot cooperate**



Courtesy: NDVIA

New Stuff

- **Function Quantifiers**
  - **`__device__` callable on the GPU from the GPU**
  - **`__global__` callable on the GPU from the CPU**
  - **`__host__` callable on the CPU from the CPU**
- **Variable Quantifiers**
  - **`__device__` global memory on the GPU**
  - **`__constant__` constant memory on the GPU**
  - **`__shared__` shared per-block memory on the GPU**
- ***Built-in Variables***
  - **`gridDim, blockDim` gives dimensions of grids and blocks in kernel**
  - **`blockIdx, threadIdx` gives index of block and thread in kernel**
- **Built-in Vector Types**
  - **`float2, float3, float4`, etc.**

CUDA Function Declarations

	Executed on the:	Only callable from the:
`__device__ float DeviceFunc()`	device	device
`__global__ void KernelFunc()`	device	host
`__host__ float HostFunc()`	host	host

- **`__global__`** defines a kernel function, must return **`void`**
- **`__device__`** and **`__host__`** can be used together

Calling a Kernel Function – Thread Creation

- **A kernel function must be called with an execution configuration:**

```
__global__ void KernelFunc(...);
dim3   DimGrid(100, 50);    // 5000 thread blocks
dim3   DimBlock(4, 8, 8);   // 256 threads per block
size_t SharedMemBytes = 64; // 64 bytes of shared memory
KernelFunc<<< DimGrid, DimBlock, SharedMemBytes >>>(...);
cudaThreadSynchronize();
```

- **Any call to a kernel function is asynchronous from CUDA 1.0 on, explicit sync needed for blocking**

- **Need to synchronize/ wait for the execution to end:**
  `cudaThreadSynchronize()`

Cuda Utility Functions

- **Use them to hunt bugs more easily!**
- **It is so easy to start a kernel that does not do anything ;-(**
- **You better want to know if the device memroy is really allocated, if the kernel is called or not.**

```
CUDA_SAFE_CALL( cudaMalloc( &devData, size ) );
…
kernel<<< 400234, 1024 >>>();
CUT_CHECK_ERROR(“kernel failed\n");
CUDA_SAFE_CALL( cudaThreadSynchronize() );
```

Principle Mode of Cuda Programming

1. **Upload data to GPU**
   - various different ways
   - try to minimize this upload
2. **Execute the kernel**
   - each thread typically produces one output element (the most easy way of thinking about parallelism)
3. **Download and analyze the results**
   - one can distill results down to a single number on the GPU
   - programming effort might be reduced if downloading the results of all threads, computing a final aggregation on the CPU

Allocation&Up/Download of Data

- **allocation (standard memory)**

```
float* devData;
cudaMalloc( (void**)&devData, nElements * sizeof(float) );
…
cudaFree( devData );
```

- **upload/download**

```
float* hostData = new float[imgsize];
// upload
cudaMemcpy(devData,hostData,size,cudaMemcpyHostToDevice);

// download
cudaMemcpy(hostData,devData,size,cudaMemcpyDeviceToHost);
```

CUDA Example

Simple Example: Gamma Correction

- **2000x3500 pixels**
- **for each color channel**
  - c = pow(c,0.8)

First steps

- **locate nvcc**
  - **`nvcc -V`**
  - release 2.0
- **locate cuda_sdk**
- **make your own project folder**
- **compile with: `nvcc -o gamma.o -c gamma.cu`**
- **link with (if necessary): `g++ -o gamma.o`**

First Steps

- **cuda toolkit:**
  `/opt/cuda`

- **cuda sdk:**
  `/opt/cuda/sdk`

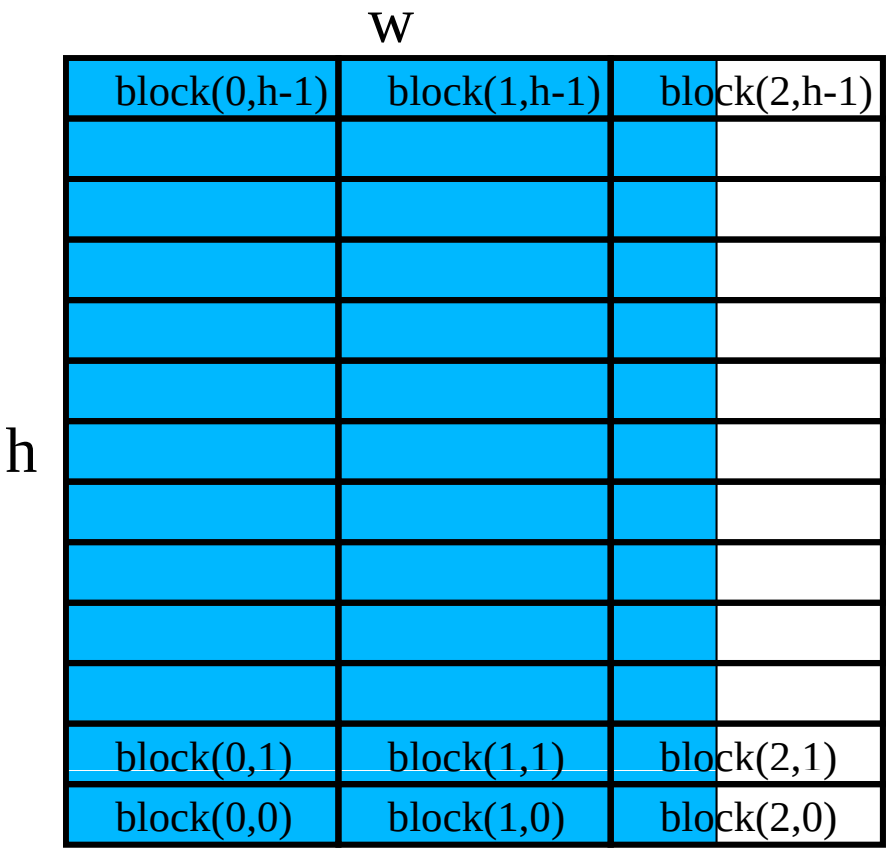
- **in your shell:**

```
export CUDA_INSTALL_PATH=“/opt/cuda”
export CUDA_SDK_DIR=“/opt/cuda/sdk”
export PATH=“/opt/cuda/open64/bin/:$PATH”

tar –xzf excercise01.tar.gz
cd gamma
make
./testGamma vase.ppm 0.5 1 out.ppm
```

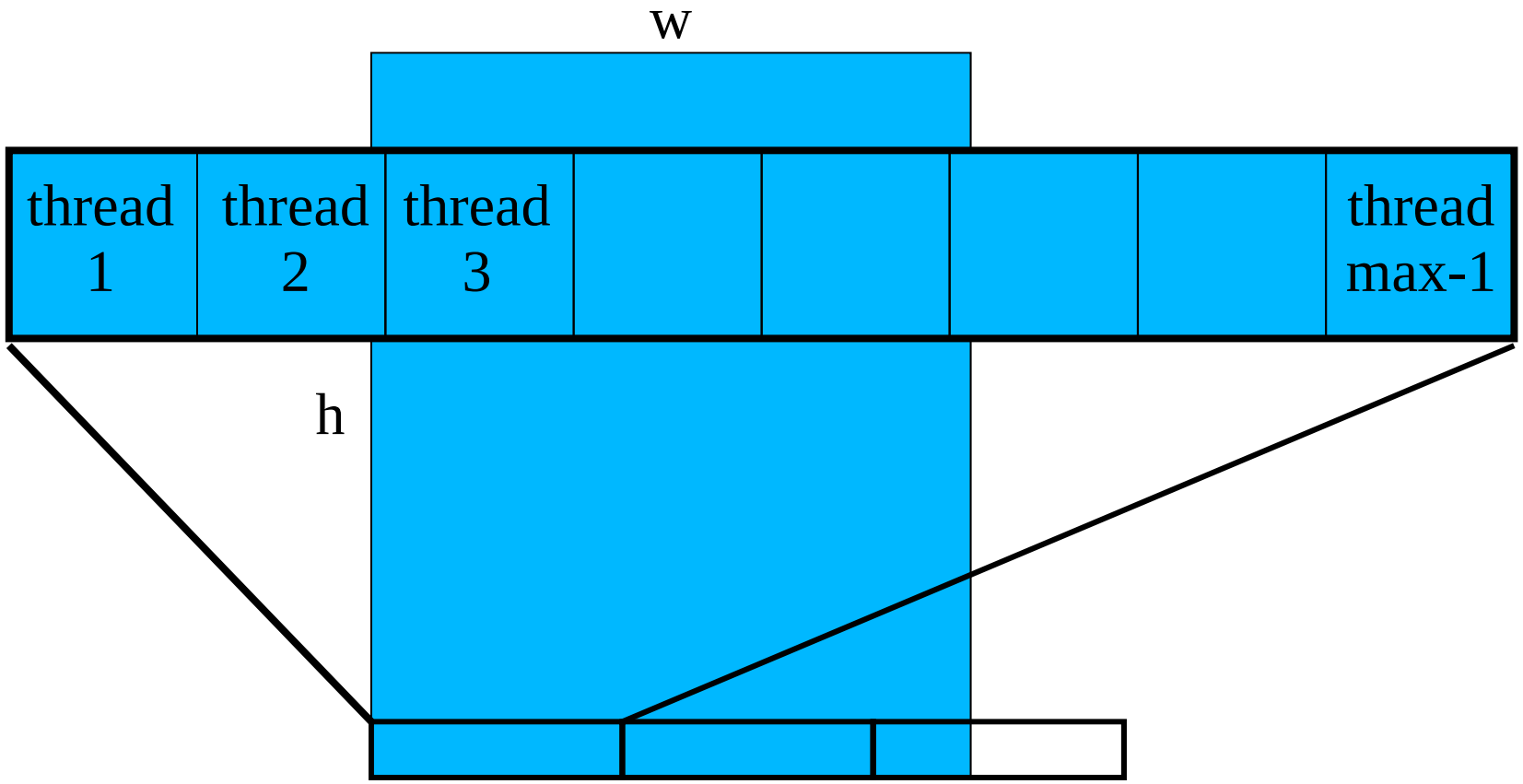
Subdividing into Blocks&Threads

- **in this example**
  - each thread computes the output of a single pixel
  - each block is one pixel high (1xMAX_THREADS)
  - multiple blocks span each line of the image



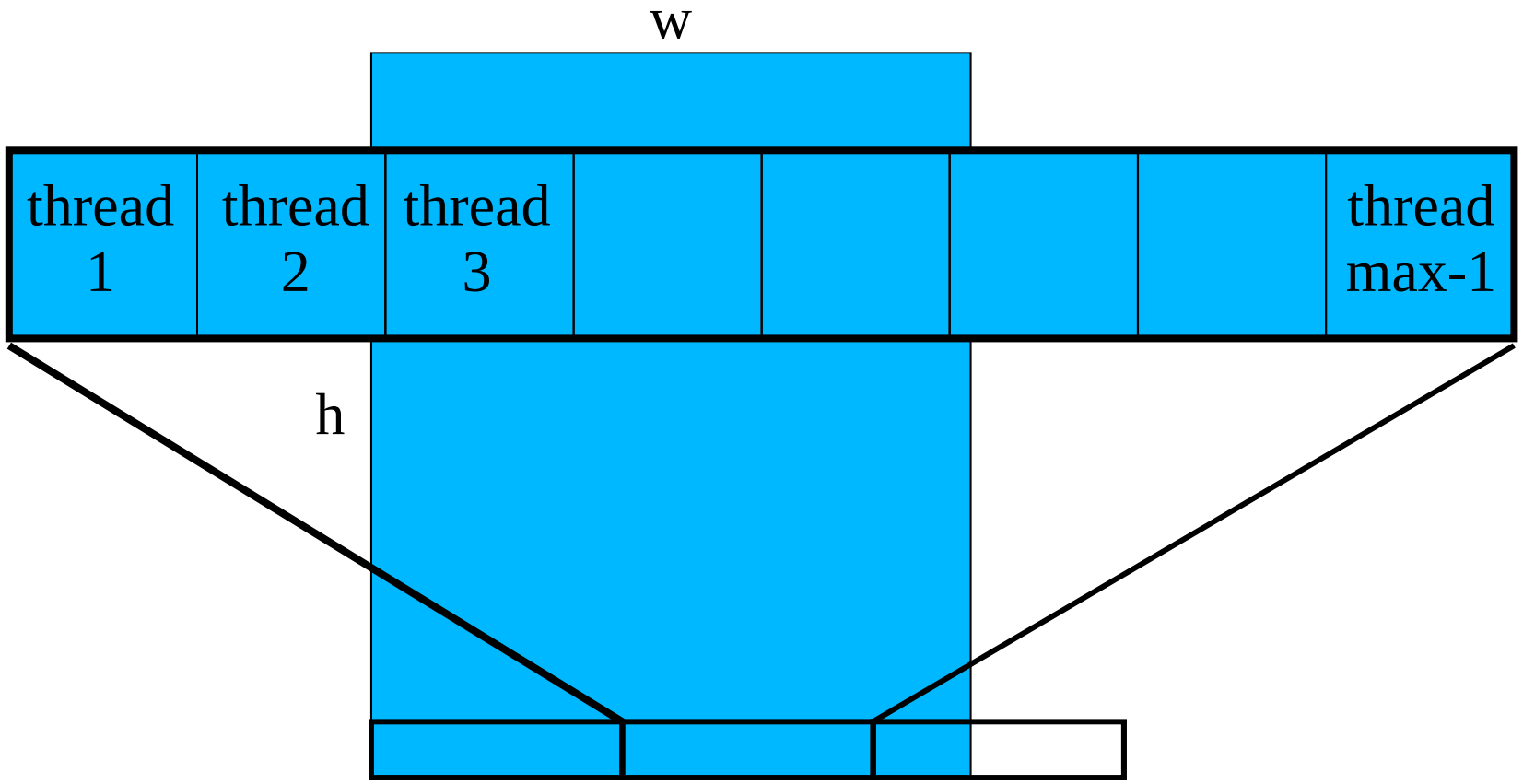
Subdividing into Blocks&Threads

- **in this example**
  - 1D block of threads: **`dim threads(MAX_THREADS)`**
  - 2D grid of blocks: **`dim blocks( w / MAX_THREADS +1, h)`**



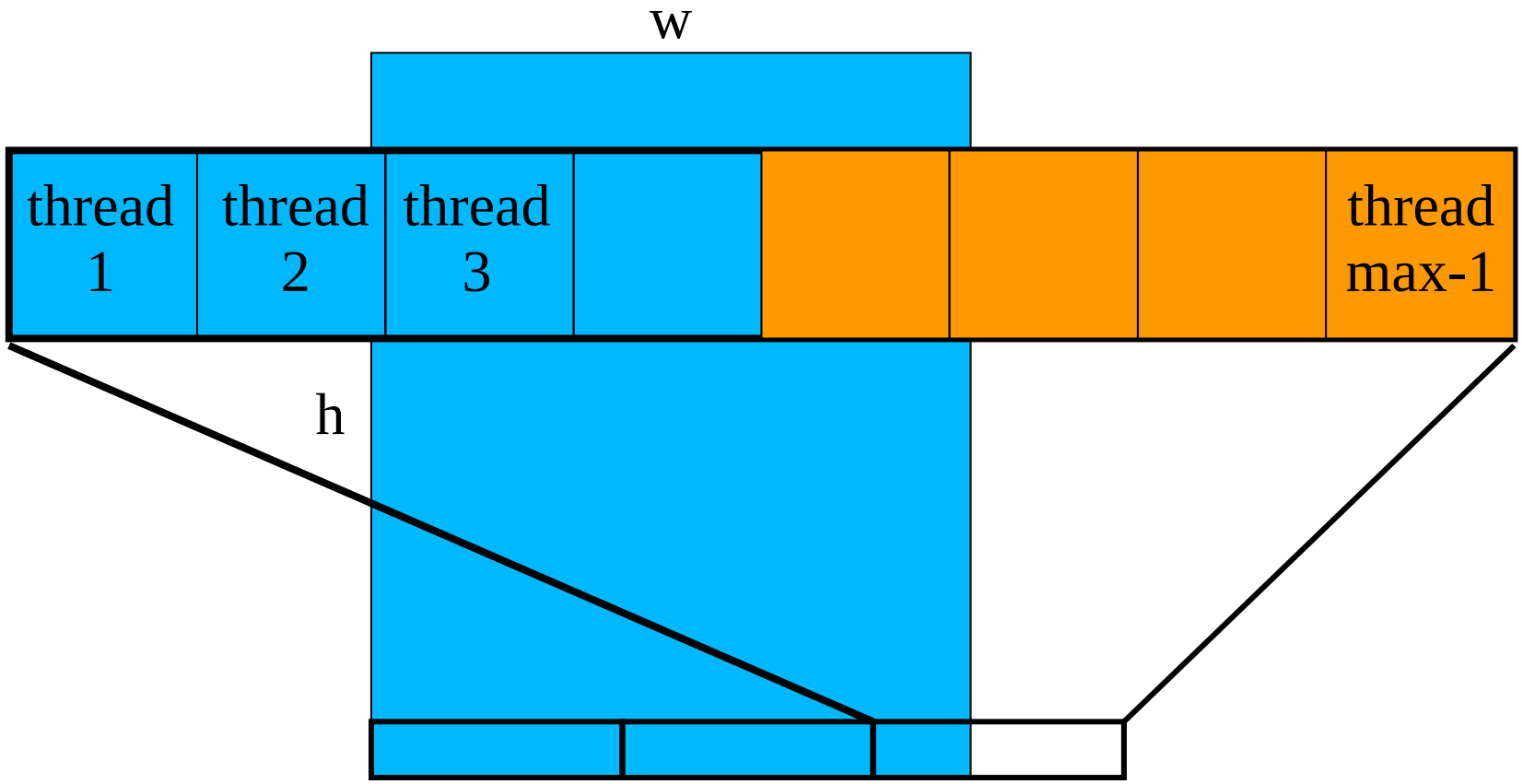
Subdividing into Blocks&Threads

- **in this example**
  - 1D block of threads: **`dim threads(MAX_THREADS)`**
  - 2D grid of blocks: **`dim blocks( w / MAX_THREADS +1, h)`**



Subdividing into Blocks&Threads

- **in this example**
  - 1D block of threads: **`dim threads(MAX_THREADS)`**
  - 2D grid of blocks: **`dim blocks( w / MAX_THREADS +1, h)`**



Assignment Sheet 01

Image Differences

Dot Product

Todo

- **Download excercise01.tar.gz from course web page.**

```
tar –xzf excercise01.tar.gz
```

- **Look into readme.txt howto setup environment variables**
- **Solve and submit the excercises**

Image Difference

- **Compute the absolute difference between two images independently for all three colors**
- **Output is again an RGB image**
- **Use the testGamma.cu example as a starting point**
- **Execute your program with the given image pair: vase.ppm & vase_blur.ppm**
- **For benchmarking: execute your kernel multiple times in a for-loop and use the \time command.**
  How does the MAX_THREADS and MAX_BLOCKS influence the performance? Plot a graph with different settings or write a table

Dot Product

- **Compute the dot product of two large scalar vectors**
- **Use the testDotProduct.cu skeleton example and fill in the missing gaps**
- **Execute for vector sizes 10000, 1000000, 100000000**
- **Similar to the already given CPU version, the dot product should be executed multiple times**
- **After how many iterations does the GPU start being faster than the CPU? (use the \time command)**
- **Examine again how the grid and block layout influence performance (again table or graph)**