Accelerator-Based Programming
-
CUDA – part 1

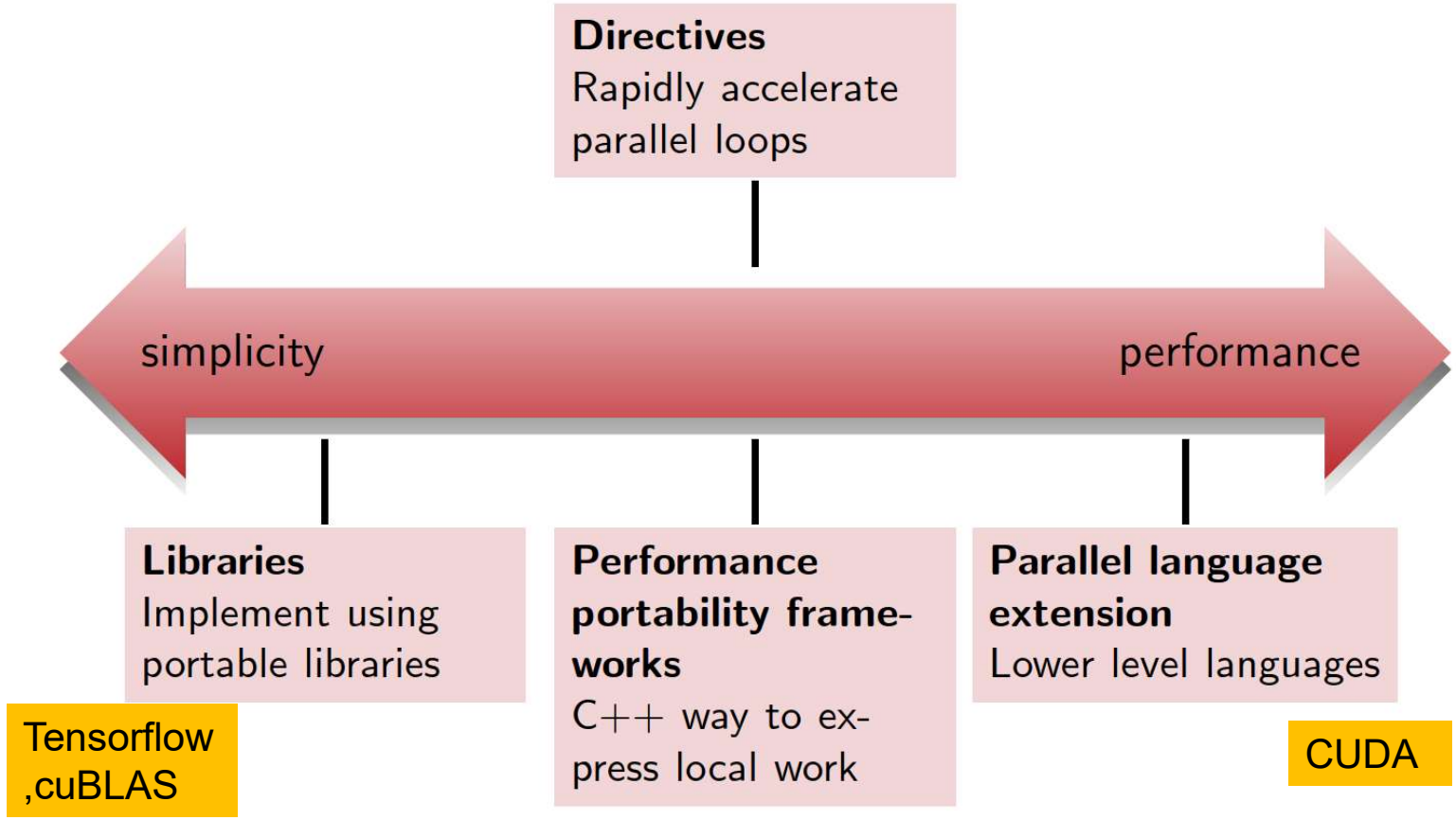
Jörn Zimmerling

September 6, 2022

UPPSALA
UNIVERSITET

Using a GPU

- Specific hardware architecture of GPUs necessitates problem-adapted codes
- Tradeoff between programming effort and achievable performance



Writing applications for a GPU

- Libraries
  - Small code changes – call an accelerated function for big chunk of work
  - High performance
  - Limited availability, no fine-grained control
  - Example: cuBLAS
- Directives
  - Annotate loops in existing languages
  - Simple to adapt, but need to touch all relevant “loops”
  - Less performance control
  - Example: OpenACC
- Languages
  - Maximal performance
  - Low-level interface close to the hardware
  - Time-consuming to develop and maintain
  - Example: CUDA

UPPSALA
UNIVERSITET

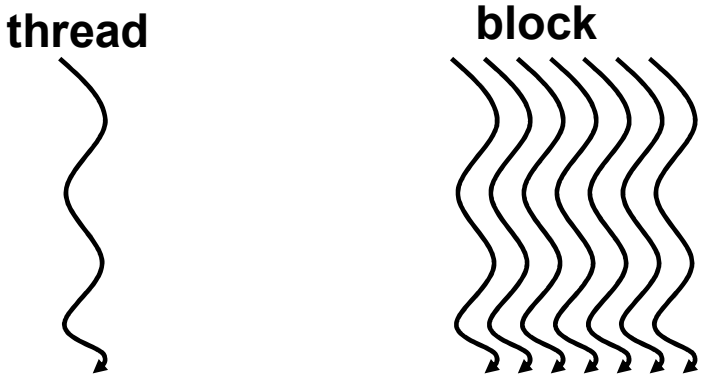
GPU parallellism

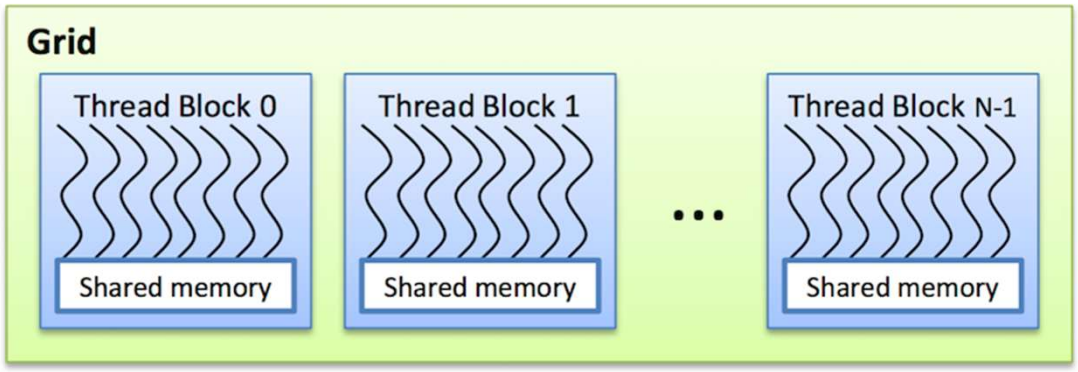
- Fine-grained data parallelism
- **SIMT** (Single Instruction **Multiple Thread**) execution
  - Many threads execute concurrently
  - Different data elements correspond to different threads
  - Hardware automatically handles thread divergence
- Not the same as SIMD because of multiple register sets, addresses, and flow paths see
  - SIMD < SIMT < SMT: parallelism in NVIDIA GPUs (yosefk.com)
- Hardware multithreading
  - High number of threads to hide latency
  - Context switching is essentially free

UPPSALA
UNIVERSITET

Parallelization for GPUs

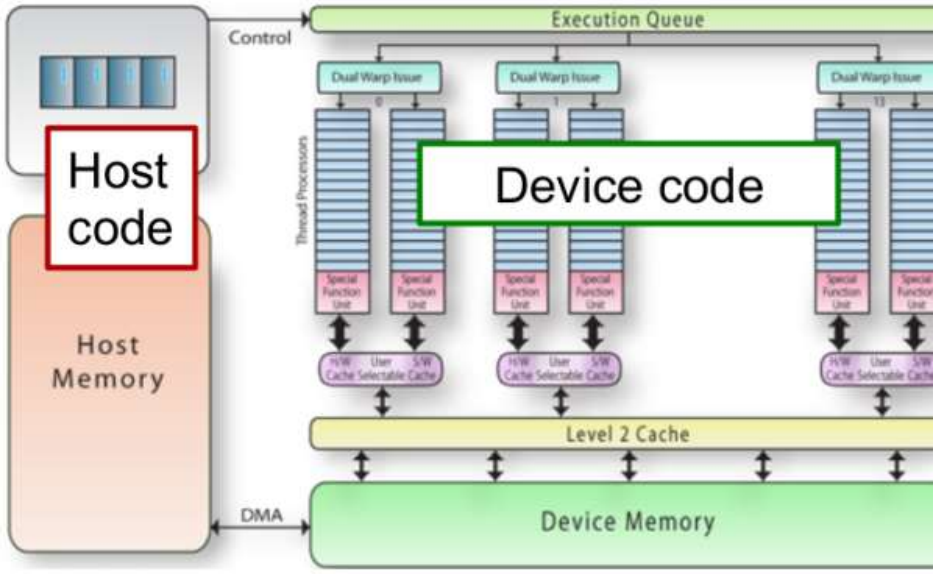
- Parallelization = find a mapping of the problem to the machine model to maximize concurrent execution
- For GPUs with their fine-grained data parallelism
  - Fine-grained = split program in large number of small tasks
  - Map data and associated work to **threads**
  - Write the computation for 1 thread!
  - Organize threads in **blocks** and blocks in **grids**
  - Let the hardware scheduler do the rest
- Assumption: Work is expressed in terms of loops with little or no data dependencies between items





GPU program organization

- Two types of code:
  - **Device code** = GPU code = kernel(s)
    - A kernel is a sequential program
    - Write for 1 thread, execute for all
  - **Host code** = CPU code
    - Instantiate the “grid” and run the kernel
    - Memory allocation, management, deallocation
    - C, C++, Java, Python, . . .
- Host-device communication
  - Explicit or implicit
  - Hardware side:
    - Via PCI/e
    - Via NVLink (if available)



Thread model on a GPU

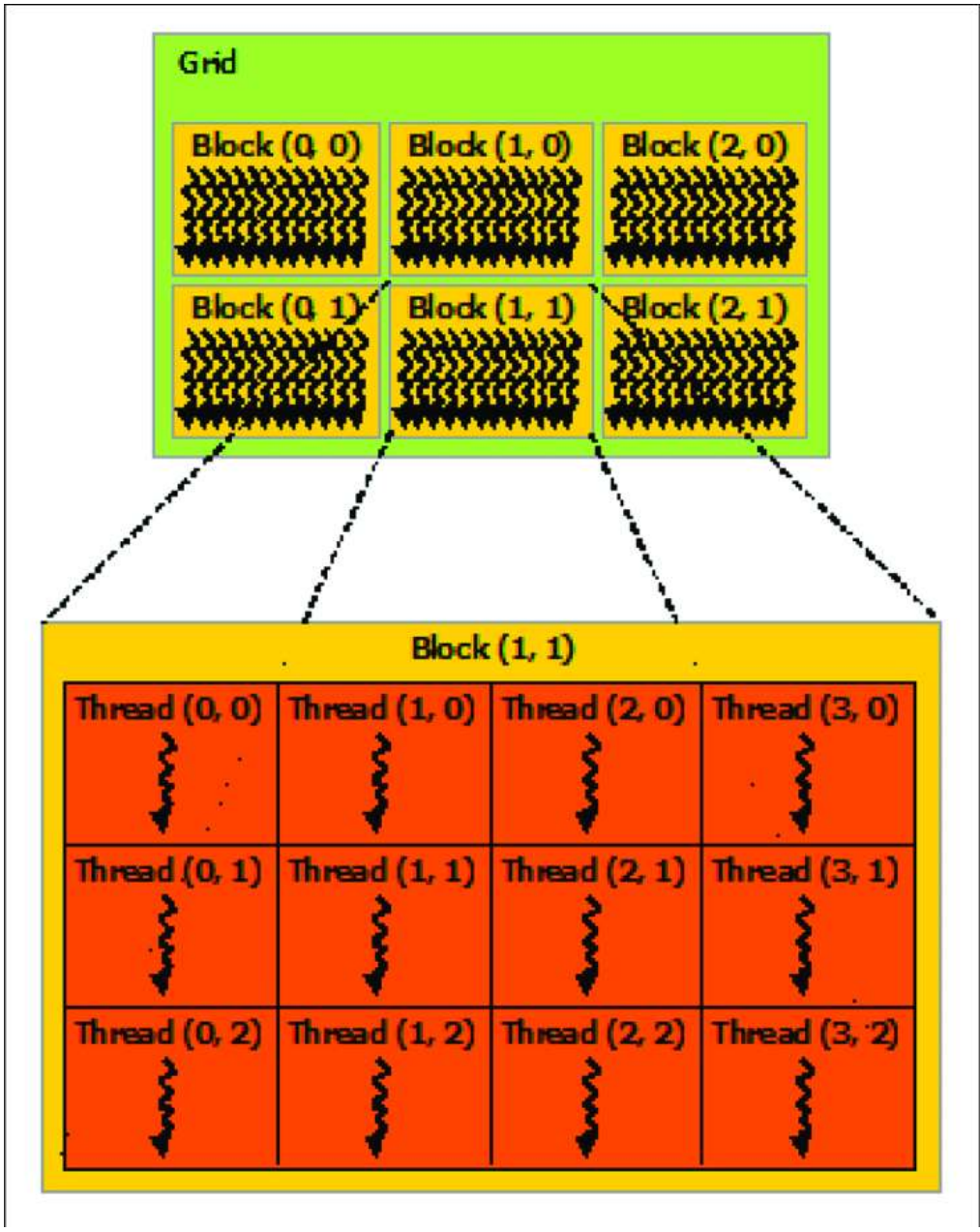
Thread hierarchy:

- Threads execute a kernel
- Threads grouped into **blocks**
- Threads in a block run together
- Blocks organized in a **grid**
- Origin of these names:

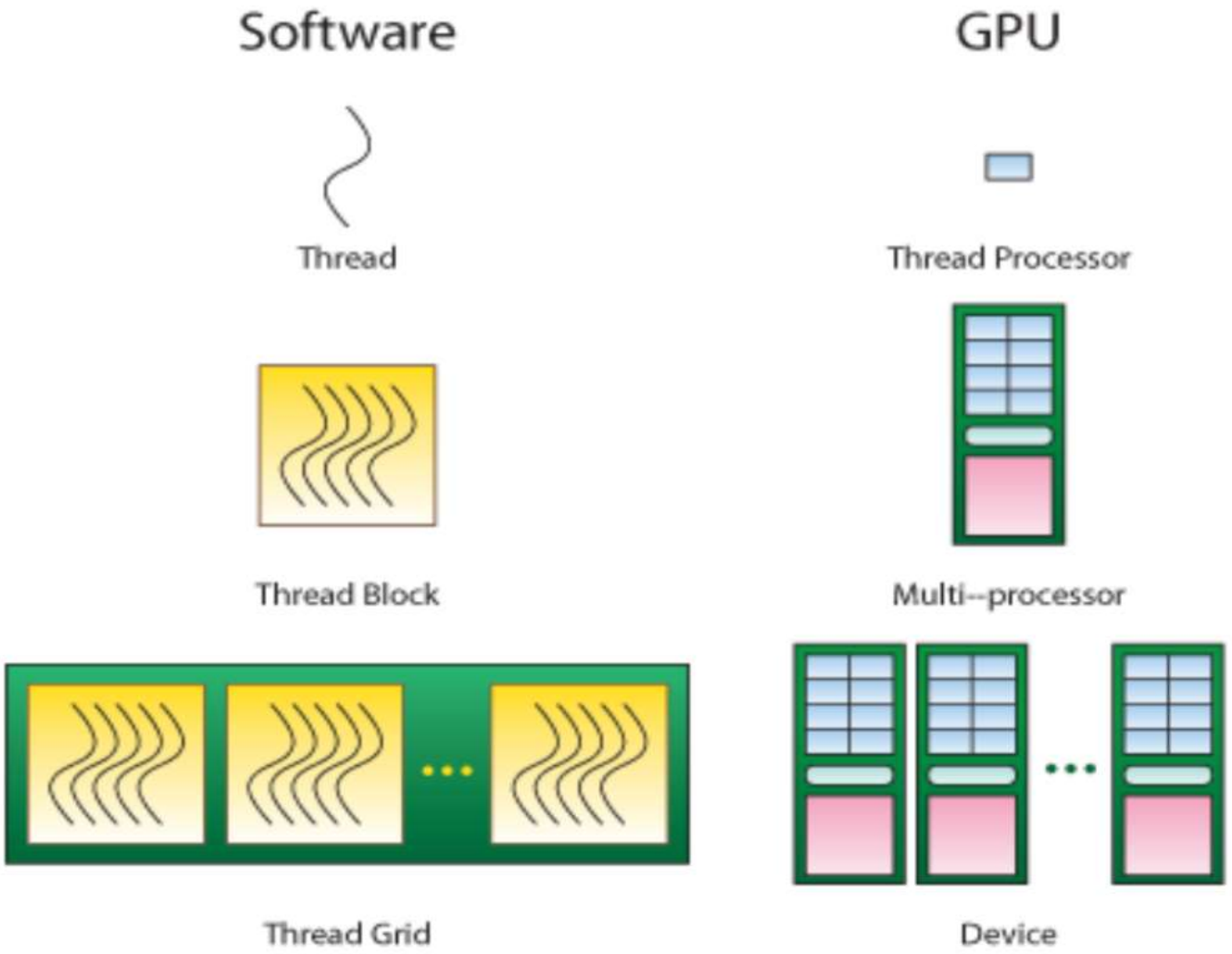
computer graphics of 2D image

Block size:

- Configurable, optimum depends application and hardware
- Often, large blocks are better
- Typically, ~ 256–1024 threads/block

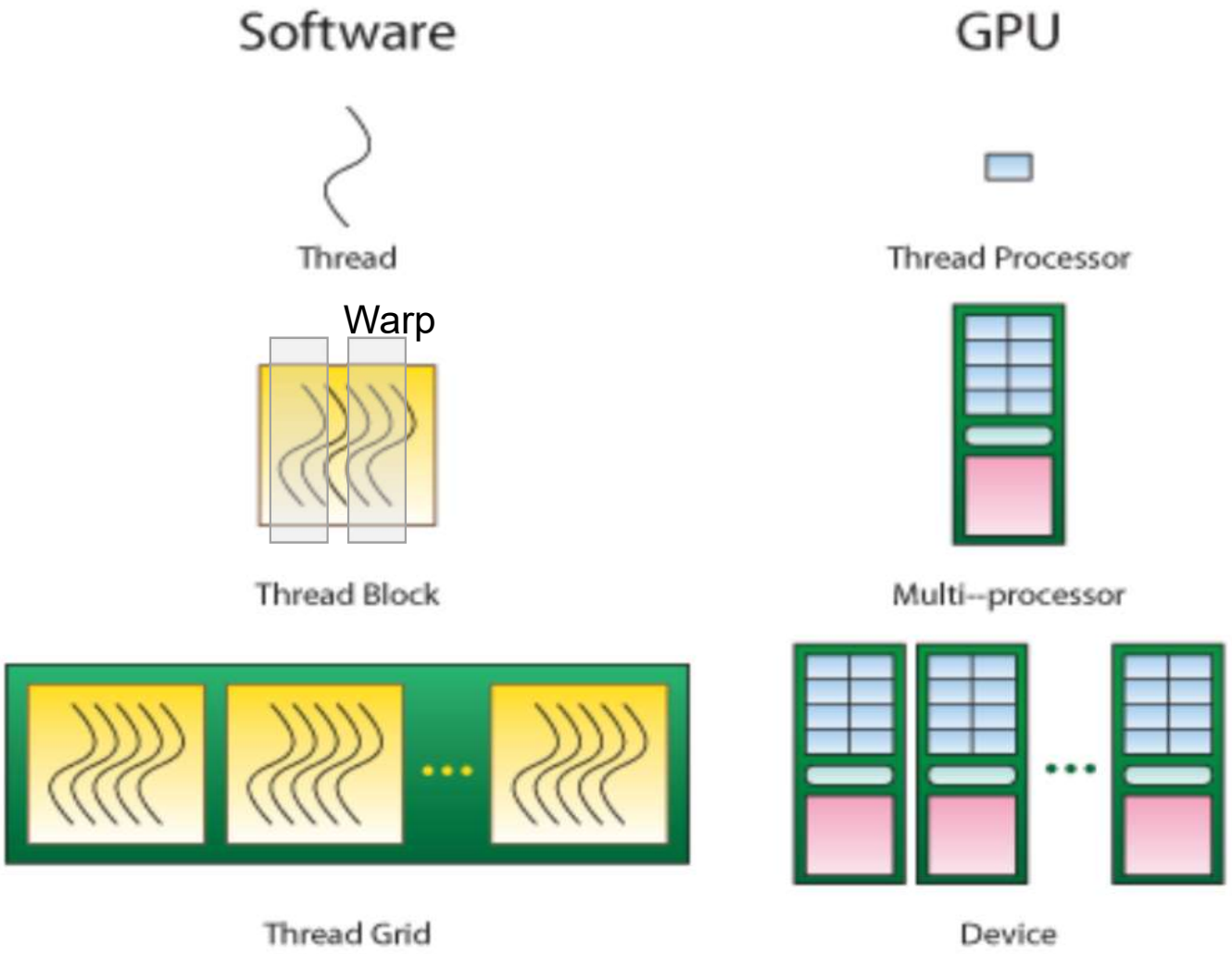


Model of parallelism vs GPU hardware



- A **Thread** is the code executed on one processor
- A **Block** gets scheduled to be executed on **one** Streaming Multiprocessor:
  - (share memory),
  - communication
  - synchronization easy
- A **Grid** is executed on a device

Model of parallelism vs GPU hardware



- A **Thread** is the code executed on one processor
- A **Warp** is a subdivision of a block
  - Warp = 32 threads
  - Threads in a warp are executed in parallel on one SM in a **Single Instruction (SIMD)** fashion

- A **Block** gets scheduled to be executed on **one** Streaming Multiprocessor:
  - (share memory),
  - communication
  - synchronization easy
- A **Grid** is executed on a device

Scheduling threads to the GPU hardware

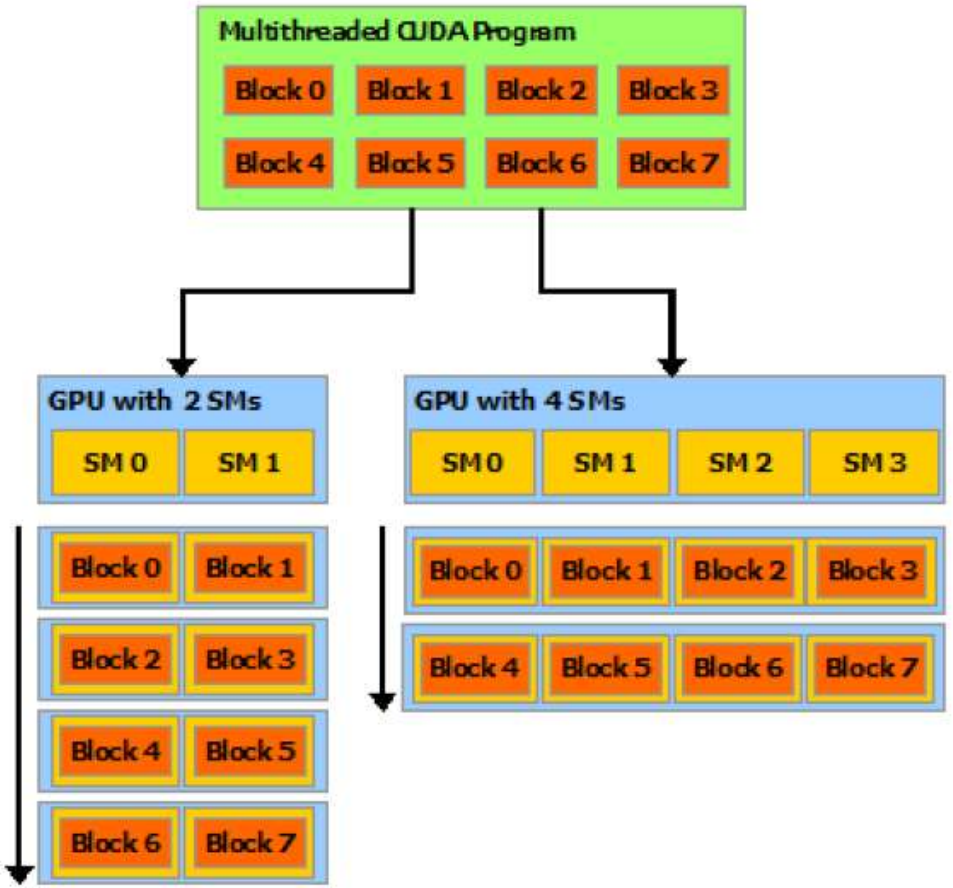
At runtime:

- Blocks assigned to SMs
- Many blocks ensure good utilization
- of hardware and hence scalability

Warps:

- Smaller subdivision of a block
- One warp = 32 threads (HW:16)
- Threads in warp executed in

SIMD (Single Instruction Multiple Data) fashion



The hardware schedules warps:

- Want high number of active warps, or occupancy
- Usage of resources (registers, shared memory, etc) limits occupancy
- Not handled explicitly when programming, but can matter for performance (next lecture)

UPPSALA UNIVERSITET

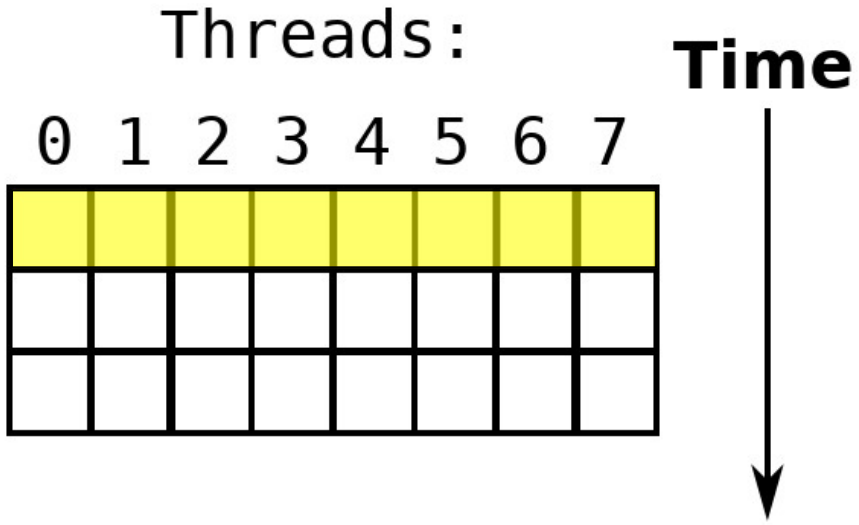
Thread model

Branch divergence:

- Threads in warp execute simultaneously on the SM
- Must execute the same instruction
- Branch divergence can hurt performance

Example:

```
if(cond(thrid)):
  do A
else: do B
```



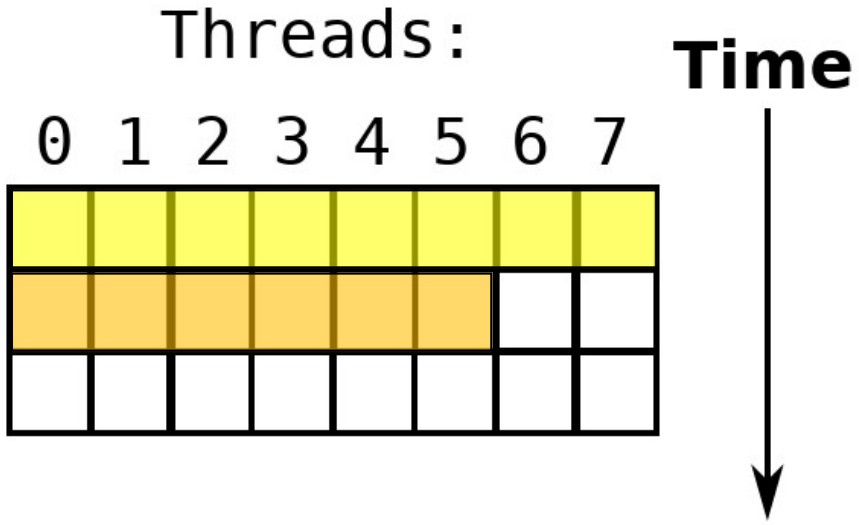
Thread model

Branch divergence:

- Threads in warp execute simultaneously on the SM
- Must execute the same instruction
- Branch divergence can hurt performance

Example:

```
if(cond(thrid)):
  do A
else: do B
```



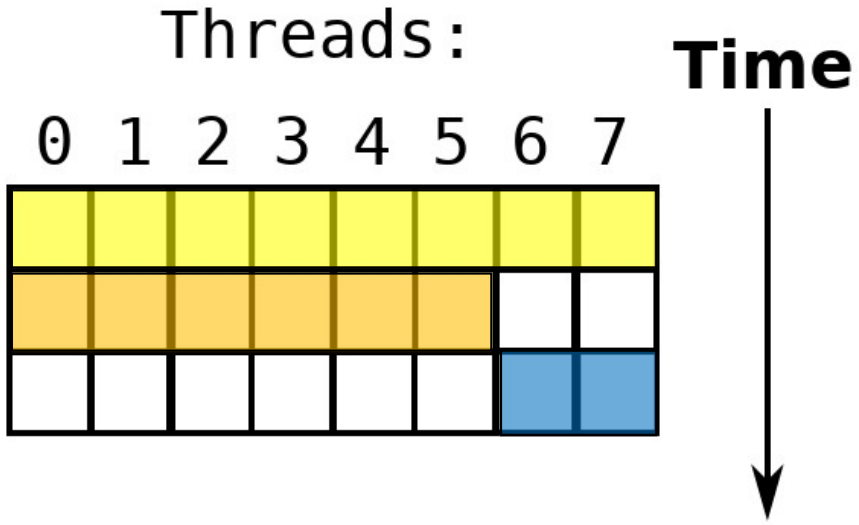
Thread model

Branch divergence:

- Threads in warp execute simultaneously on the SM
- Must execute the same instruction
- Branch divergence can hurt performance

Example:

```
if(cond(thrid)):
  do A
else: do B
```



Efficiency loss!

Note: Threads in the same block but different Warps do not have this problem

CUDA introduction

Compute Unified Device Architecture

- Dedicated framework for GPU programming
  - Programming language (C/C++ extension)
  - Hardware and thread model
  - Development toolkit

Pros/Cons

++ Low-level (high performance)

\- Low-level (hard to program)

\+ Mature (debugger, profiler, ...

\- Nvidia only

\+ Portable across Nvidia GPUs

UPPSALA
UNIVERSITET

OpenCL & CUDA

- Also: OpenCL
  - Open standard
  - Cross platform (GPUs, multicores,FPGA, etc)
  - Low-level (high performance)
- Why CUDA?
  - OpenCL must be tuned to perform
  - CUDA performs better
  - Not as mature
  - Messy to program
- They are similar:
  - Very similar programming models
  - Can ”easily” convert ~ CUDA ⇔ OpenCL

First CUDA code

Vector addition: $x := x + y$

CPU function:

```
void vec_add(int N, float *x, const float *y) {
  for(int i=0; i<N; ++i)
    x[i] = x[i] + y[i];
}
```

CUDA kernel:

```
__global__
void vec_add(int N, float *x, const float *y) {
  int i = threadIdx.x;
  x[i] = x[i] + y[i];
}
```

Call:

```
vec_add<<<1,N>>>(N, x, y);
```

Additional reading: https://developer.nvidia.com/blog/even-easier-introduction-cuda/

Understanding CUDA kernels

Types of functions:

`__global__`	Device code called from host – kernel
`__device__`	Device code called from device
`__host__`	Host code called from host (usual functions)

Identifying threads:

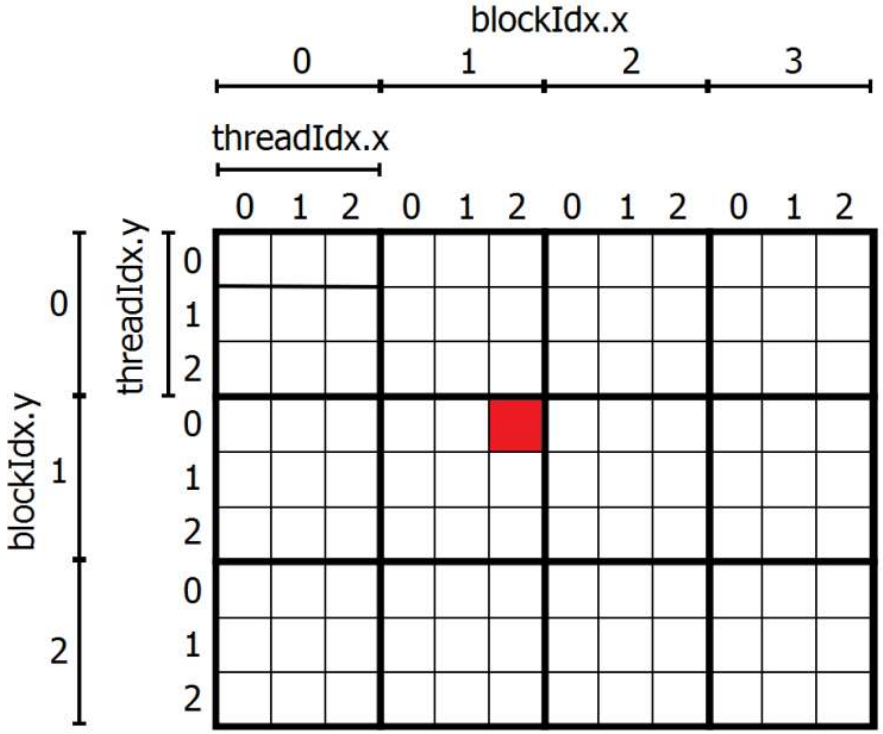
`threadIdx`	Thread index in a block
`blockIdx`	Block index in a grid
`blockDim`	Size of a block

Data Type:

`dim3`

- 1D, 2D or 3D data type
- Integer vector type:

`threadIdx.x/ threadIdx.y / threadIdx.z`



Launching CUDA kernels

- Kernel invocation via triple chevrons:

```
kernel<<<grid_dim, block_dim>>>(args);
```

- Configuration parameters:

`block_dim`	Size of thread block	`(dim3)`
`grid_dim`	Blocks per grid	`(dim3)`

UPPSALA
UNIVERSITET

Example: matrix addition. Given N, A, B

Code:

```
/* kernel */
__global__
void matrix_sum(int N, float *C, const float *A,
                const float *B) {
  int i = threadIdx.x + blockIdx.x*blockDim.x;
  int j = threadIdx.y + blockIdx.y*blockDim.y;
  if(i<N && j<N)
    C[i*N+j] = A[i*N+j] + B[i*N+j];
}
```

Masked not branched

Invocation:

```
...
  /* kernel configuration */
  dim3 block_dim(8, 8);
  int num_blocks = 1 + (N-1)/8;
  dim3 grid_dim(num_blocks, num_blocks);
  /* kernel launch */
  matrix_sum<<<grid_dim,block_dim>>>(N, C, A, B);
...
```

Round up

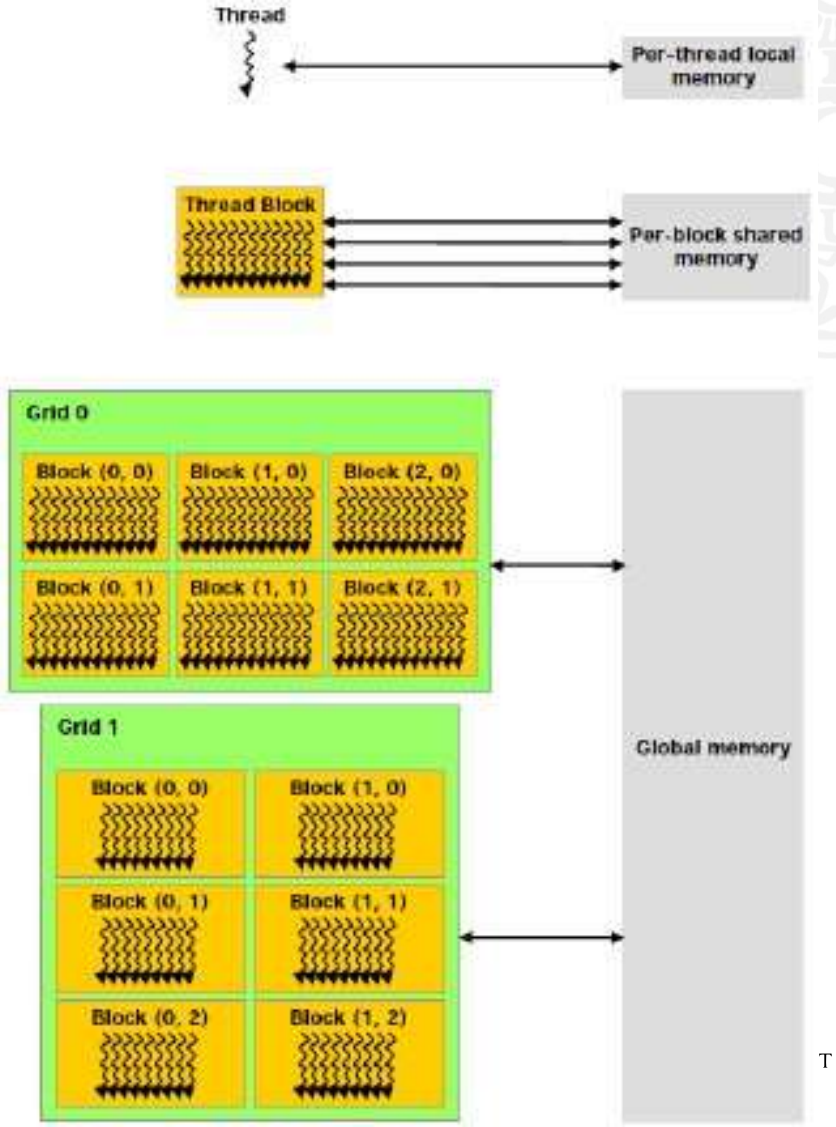
UPPSALA UNIVERSITET

Memory in CUDA – on GPU

Types of memory:

- Single thread has registers
  - ≤ 255 32-bit regs
  - 0 cycle access cost
- Threads in block share memory
  - shared memory up to 192 kB / SM
  - ~ 50 cycles
  - up to ~ 10 TB/s on recent GPUs
  - High BW, high latency compared to cpu
- Main device memory –

  global memory
  - ~ 8–40 GB
  - ~ 500 cycles
  - ~ 200–1600 GB/s



Memory in CUDA – on GPU

- Allocating device memory:
  - `cudaMalloc` – allocates global memory on device
  - `cudaFree` – frees it again
  - `cudaMemset` – used for initializing memory
- Used exactly like:
  - `malloc`
  - `free`
  - `memset`
- Device and host pointers:
  - Pointers either valid on host or device
- Exception: Unified memory
  - `cudaMallocManaged`

UPPSALA
UNIVERSITET

Memory in CUDA

- Host-device transfer – `cudaMemcpy`:
- Host to device:

```
cudaMemcpy( x_dev , x_host , num_bytes ,cudaMemcpyHostToDevice);
```

- Device to host:

```
cudaMemcpy( x_host , x_device , num_bytes ,cudaMemcpyDeviceToHost);
```

Also `cudaMemcpyAsync:`

- Non-blocking communication
- Overlap communication and computation
- Cf. `MPI_Isend`

UPPSALA
UNIVERSITET

Memory in CUDA

- Example:

```
void main() {
  int n = 256;
  int num_bytes = n*sizeof(int);
  int *x_dev, *x_host;
  /* allocate memory */
  x_host = (int*) malloc(num_bytes);
  cudaMalloc(&x_dev, num_bytes);
  /* set to 0 */
  cudaMemset(x_dev, 0, num_bytes);
  /* copy memory to host */
  cudaMemcpy(x_host, x_dev, num_bytes,
             cudaMemcpyDeviceToHost);
  /* free up memory */
  free(x_host);
  cudaFree(x_dev);
}
```

UPPSALA
UNIVERSITET

Further Sources

- Current capabilities see e.g.
  - https://developer.nvidia.com/blog/nvidia-ampere-architecture-in-depth/

Read the Whitepaper:
https://images.nvidia.com/aem-dam/en-zz/Solutions/data-center/nvidia-ampere-architecture-whitepaper.pdf

UPPSALA
UNIVERSITET

- Other resources are:

https://developer.nvidia.com/blog/even-easier-introduction-cuda/

They start with Unified Memory.