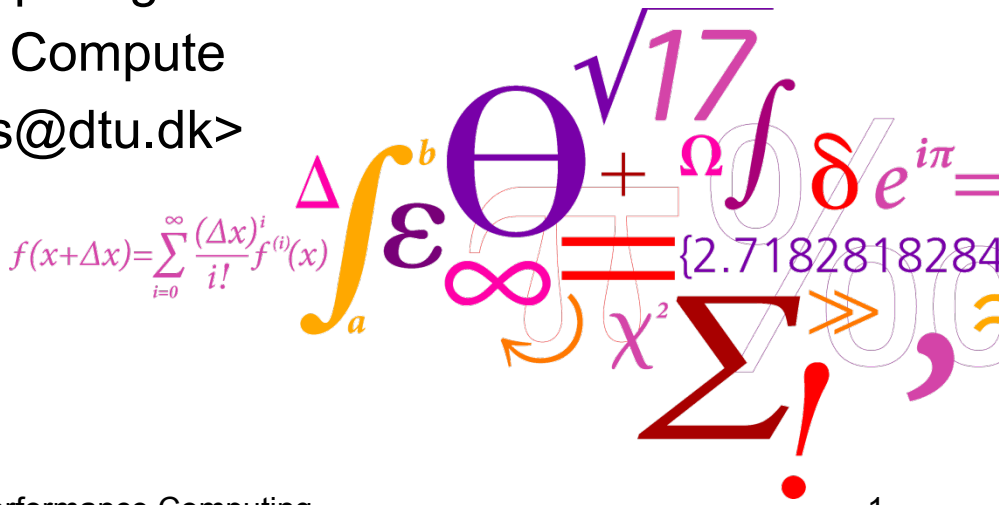
CUDA Programming Model


Hans Henrik Brandenborg Sørensen
DTU Computing Center
DTU Compute
<hhbs@dtu.dk>

DTU DCC Computing Center

GPUlab DTU Informatics



Overview

- What is CUDA? What is OpenCL?
- CUDA programming model
- CUDA C extensions
  - Function qualifiers
  - Vector types
  - Math intrinsics
- Launching CUDA kernels
  - Thread hierarchies
  - Launch configuration
  - ThreadID calculation

Acknowledgements

- Some slides are from Paul Richmond, The University of Sheffield
  - http://paulrichmond.shef.ac.uk/teaching/COM4521/
- Some slides are from David Kirk and Wen-mei Hwu's UIUC course:
  - http://courses.engr.illinois.edu/ece498/al/
- Some slides are from Patrick Cozzi, University of Pennsylvania, CIS 565 course:
  - http://cis565-fall-2012.github.com/schedule.html
- Some slides are from NVIDIA Developer
  - https://developer.nvidia.com/

CUDA / OpenCL

What is CUDA?

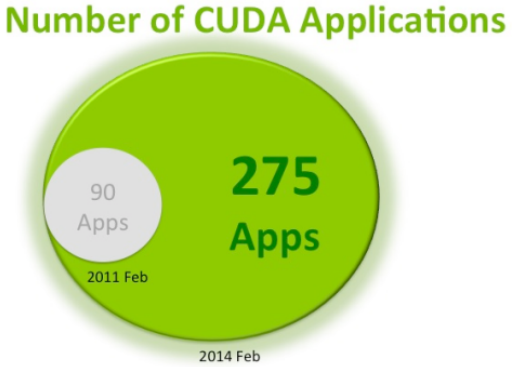
DTU

Compute **U**nified **D**evice **A**rchitecture

- A standard proposed by Nvidia for general-purpose computations on CUDA-enabled GPUs
- Priority #1: Make things easy (Sell GPUs)
- Priority #2: Get performance
- Result: Simple to get started, but..
  - Requires expert knowledge to get best performance
- Scalable
- Well documented

CUDA by the numbers

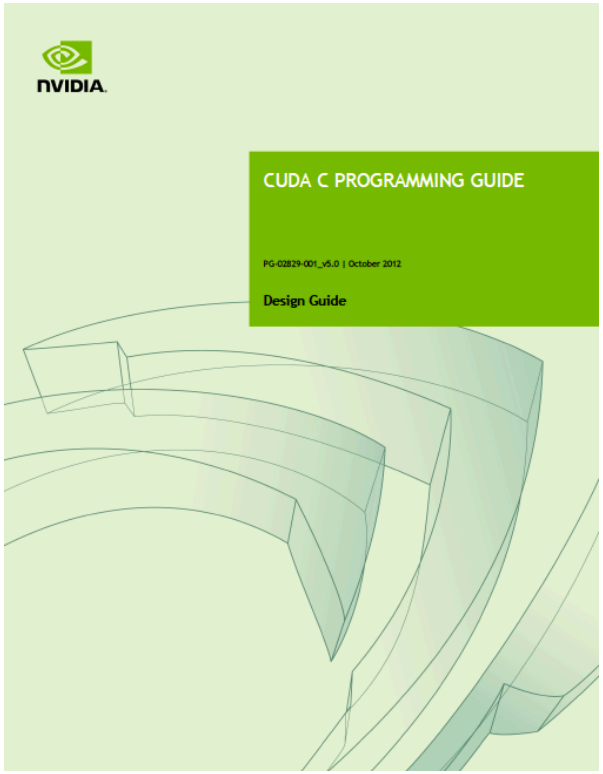
- CUDA-Capable GPUs
  - More than 506 mill. (2012: 375 mill.)
- Toolkit downloads
  - More than 2.200.000 (1.000.000)
- Active Developers
  - More than 190.000 (120.000)
- Universities Teaching CUDA
  - More than 738 (500) [in Denmark DTU, AU and KU].

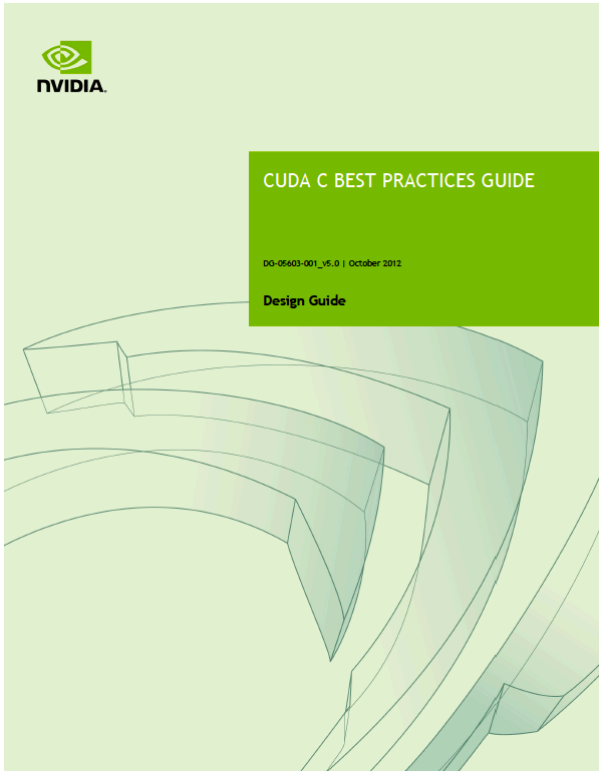


- 1 download every 60 seconds!

Souce: http://blogs.nvidia.com/blog/2014/02/25/cuda-by-numbers ,Nvidia 2014

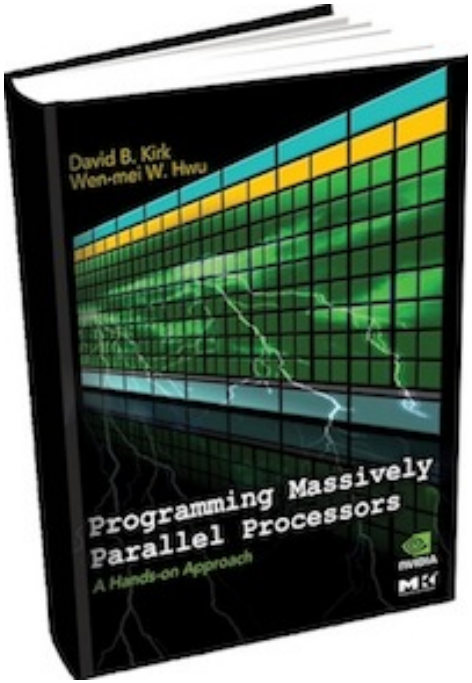
Free CUDA material

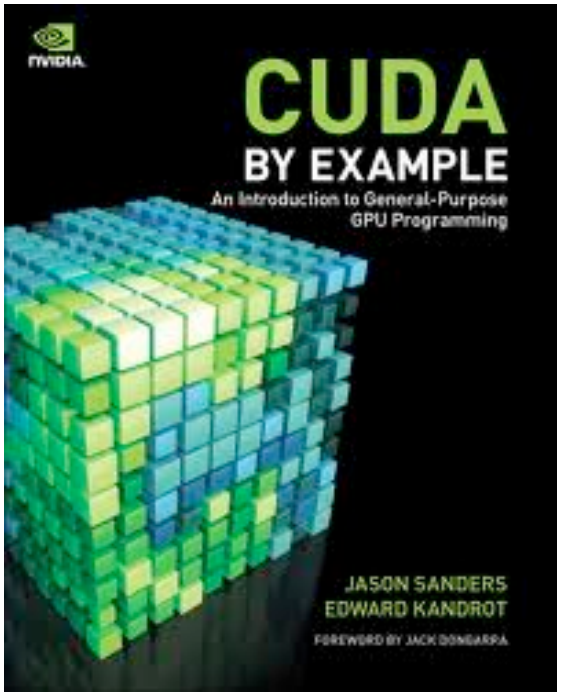


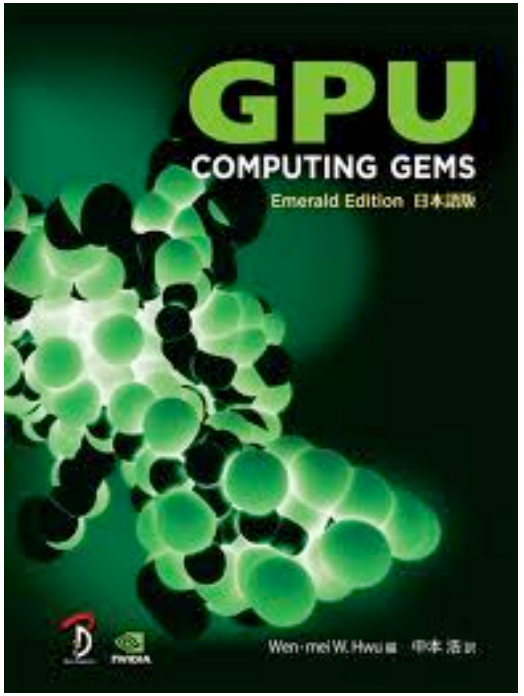


- Free online from Nvidia developer webpage
  - http://docs.nvidia.com/cuda/index.html
  - Pdf versions, see installation; /appl/cuda/8.0/doc/pdf/

Additional CUDA material

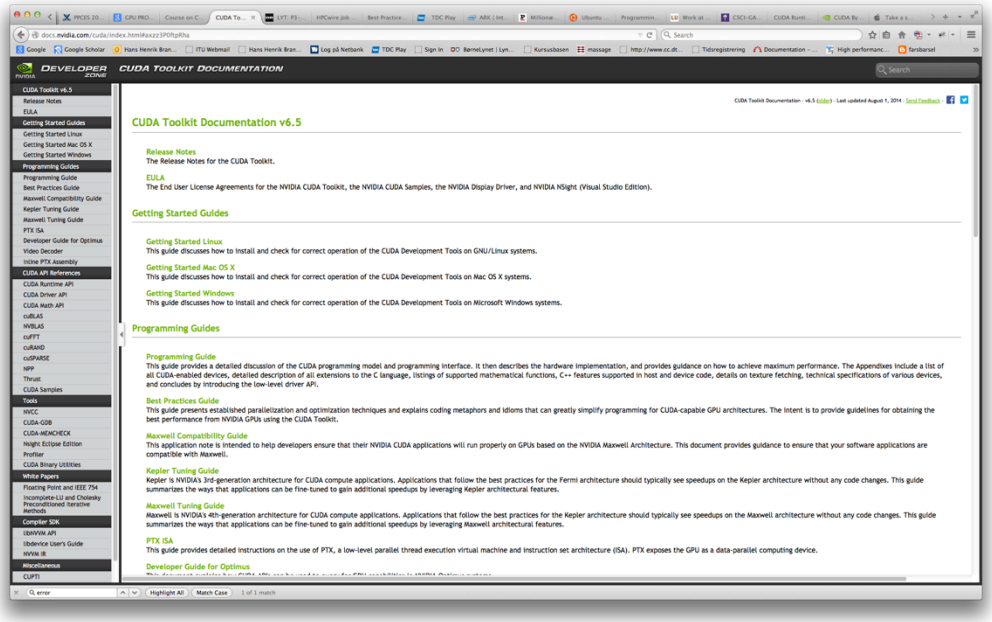




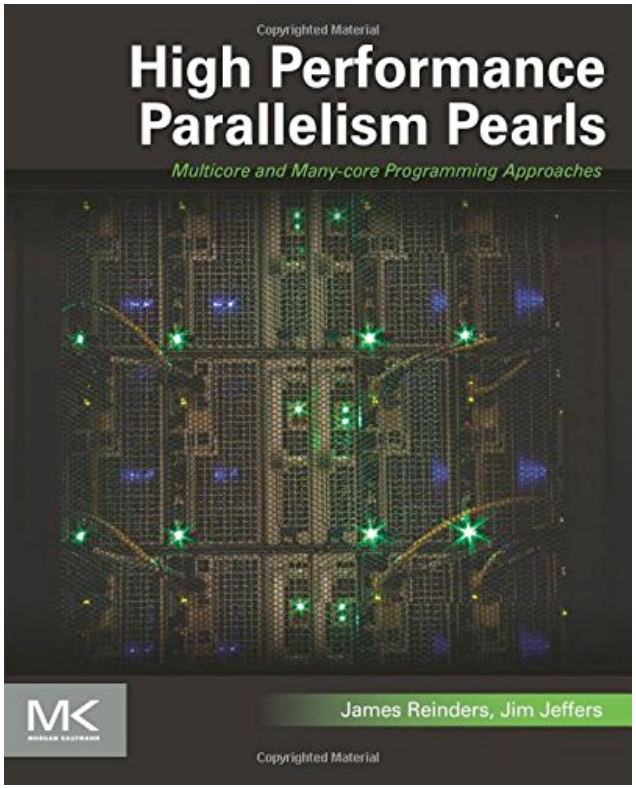


- These are currently the most widely used books
- Feel free to come by my office building 324, room 280 to take a peek in these references

Our suggestion (if you get hooked)



+



http://docs.nvidia.com/cuda/index.html

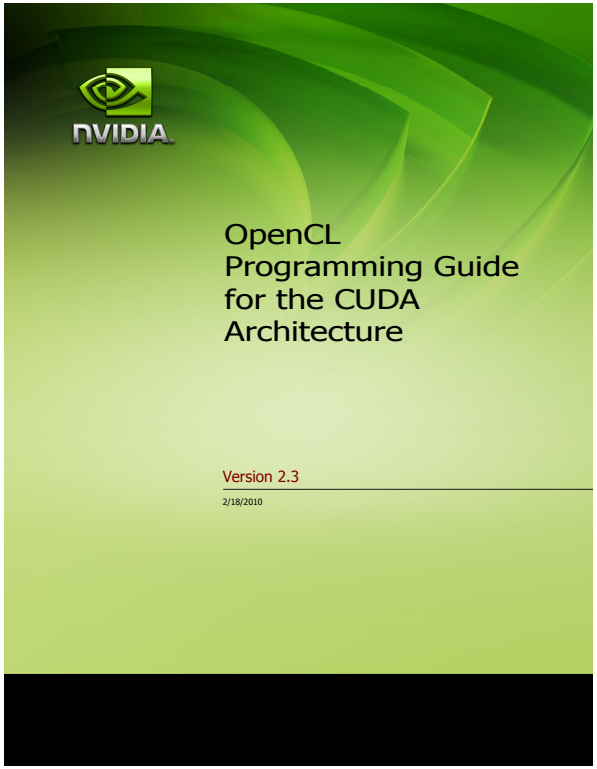
What is OpenCL?

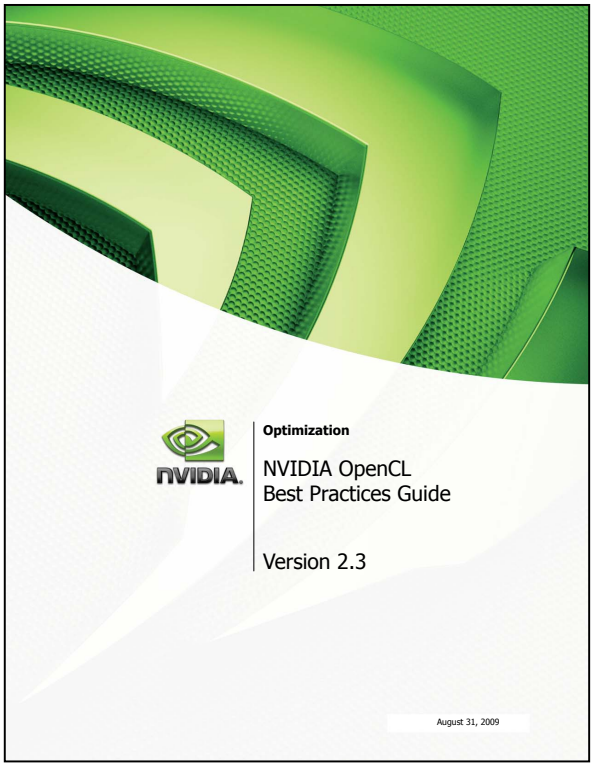
Open Computing Language (current v2.2)

- Khronos group (non-profit organization):
  - "*OpenCL is an open, royalty-free standard for cross-platform, parallel programming of modern processors found in parallel computers, servers and handheld/embedded devices*."
- Open standard for heterogeneous computing
- Priority #1: Become the *industry-wide future standard* for heterogeneous computing
- Priority #2: Use all computational resources in the system efficiently
- Vendor specific!

KHRONOS GROUP

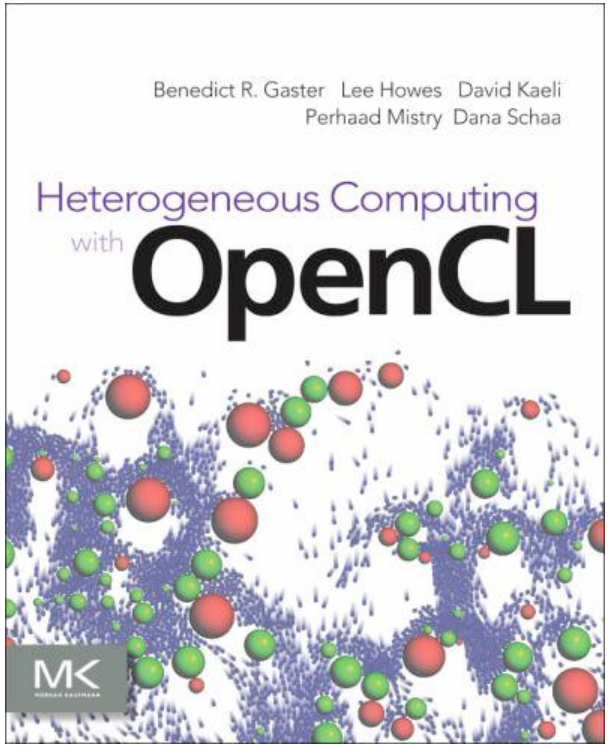
Free OpenCl material

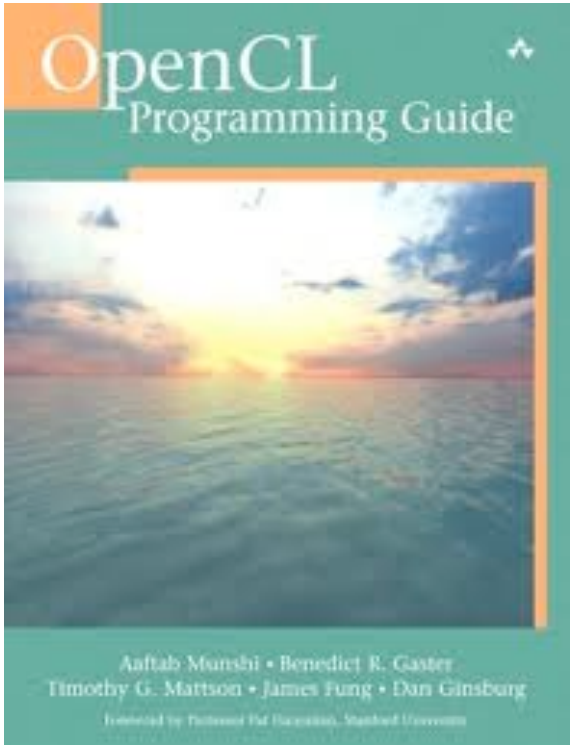


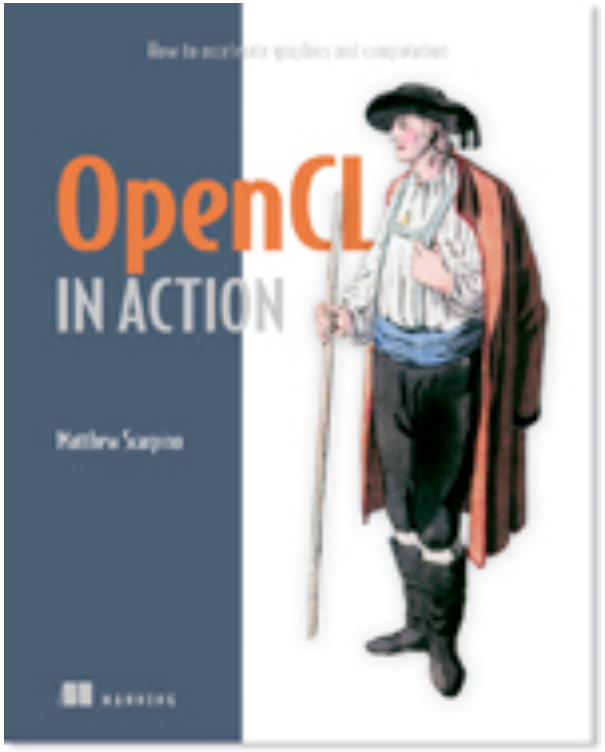


- Free from Nvidia and AMD developer webpages
  - https://developer.nvidia.com/opencl
  - Pdf versions, see installation; /usr/local/cuda/doc/pdf/

Additional OpenCL material







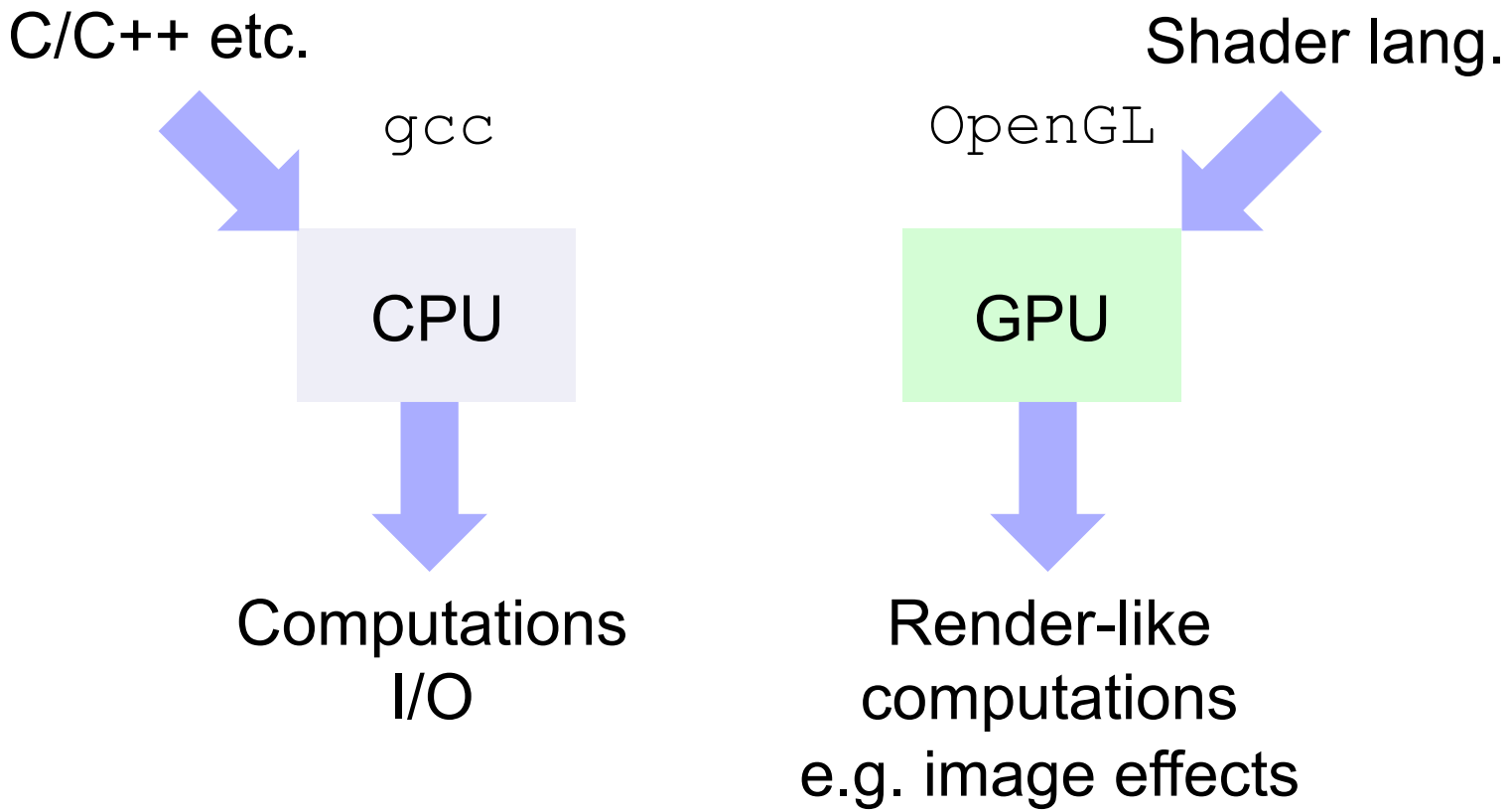
- All published within the last four years
  - Expect more to be published in the future
  - Feel free to come by 324-280 to take a peek

Why CUDA in this course?

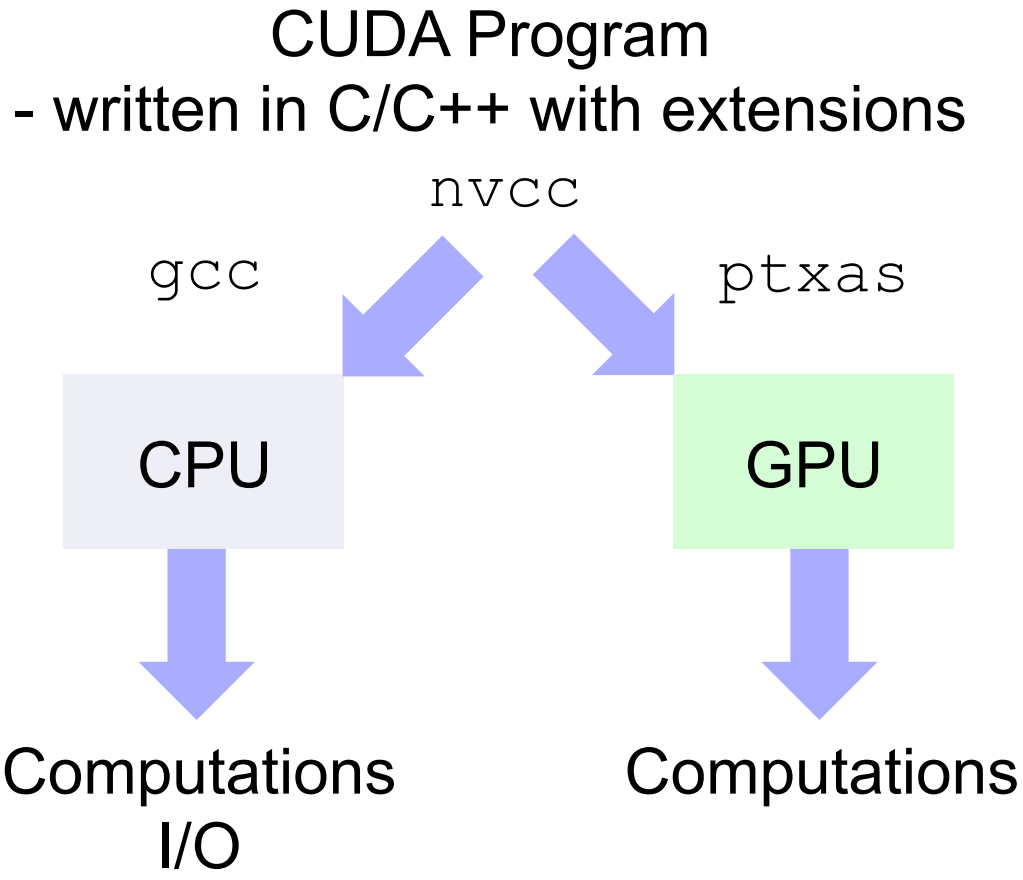
- CUDA and OpenCL are very similar
  - Most CUDA features map one-to-one to OpenCL features (only the syntax is different)
  - OpenCL provides more explicit functionality and is supported by a less mature software framework
- CUDA comes with tuned high-performance libs
  - OpenCL have less effort in this direction (currently)
- CUDA is well documented (by Nvidia)
- Nvidia products are widely used in HPC (>85%)
  - OpenCL still lags in performance for Nvidia products

CUDA programming model

Earlier than 2007

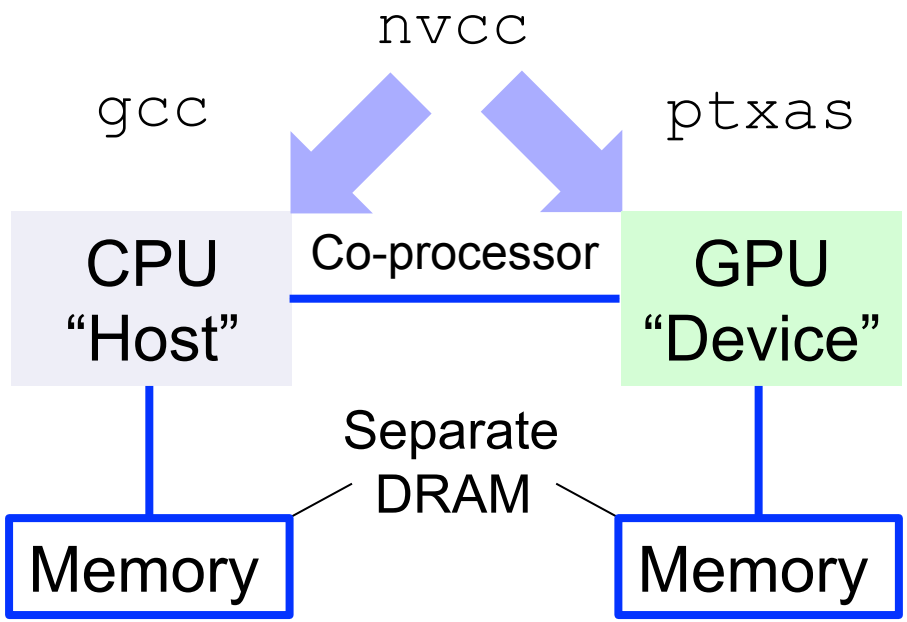


CUDA released 2007



CUDA programming model

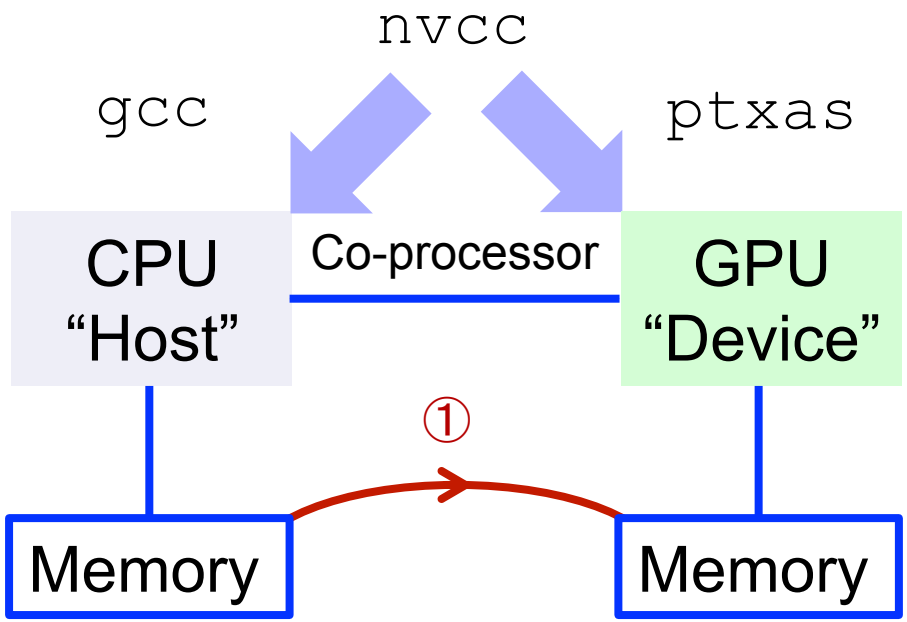
CUDA Program
- written in C/C++ with extensions



- Host – the CPU
  - In charge, manages resources
  - Runs main(), etc.
- Device – the GPU
  - Co-processor / accelerator
  - Runs specific tasks

CUDA programming model

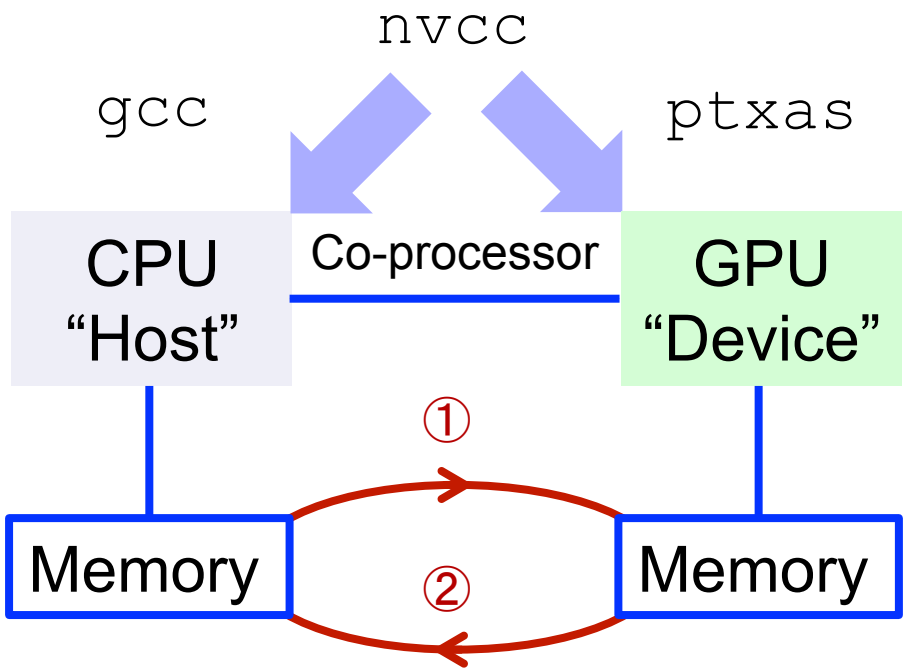
CUDA Program
- written in C/C++ with extensions



① Data CPU ➔ GPU

CUDA programming model

CUDA Program
- written in C/C++ with extensions

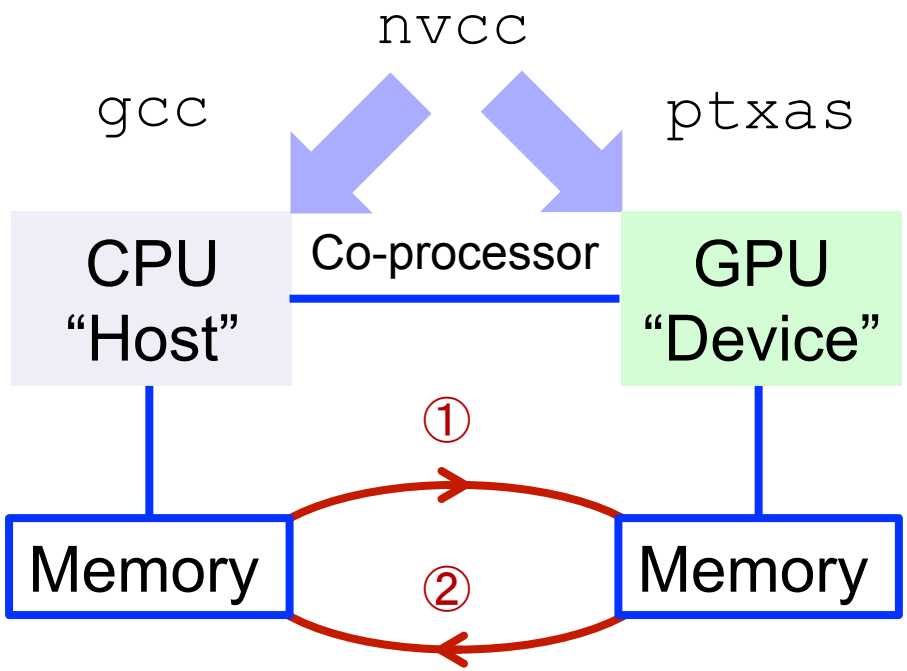


① Data CPU ➔ GPU

② Data GPU ➔ CPU

CUDA programming model

CUDA Program
- written in C/C++ with extensions



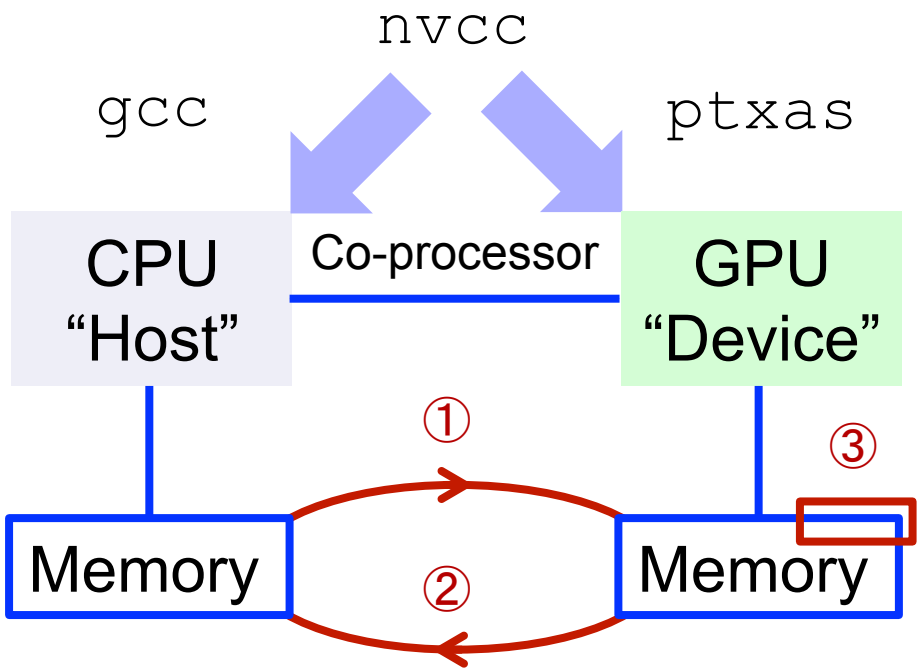
① Data CPU ➔ GPU

② Data GPU ➔ CPU

- `cudaMemcpy()`

CUDA programming model

CUDA Program
- written in C/C++ with extensions



① Data CPU ➔ GPU

② Data GPU ➔ CPU

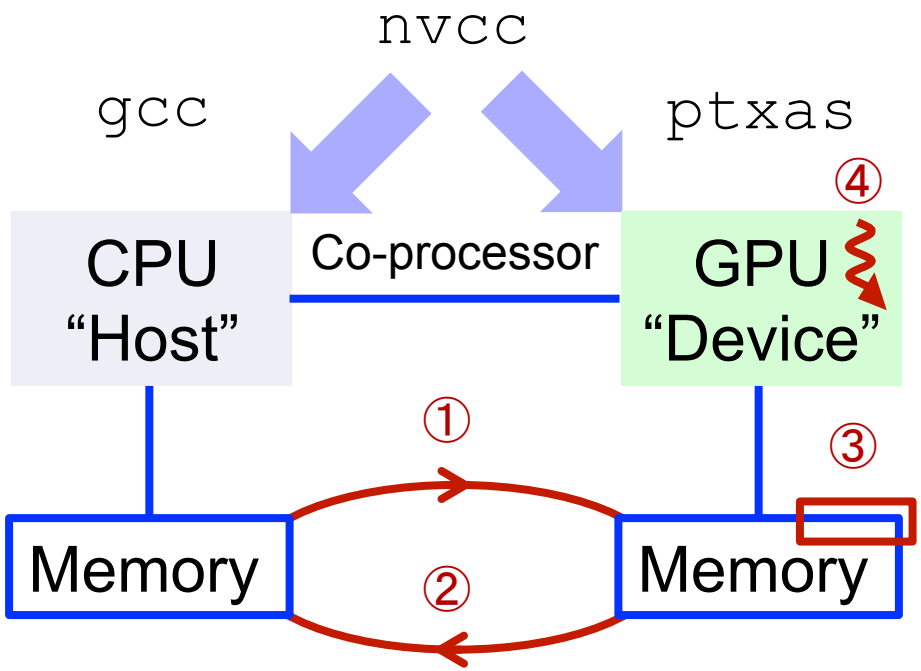
- `cudaMemcpy()`

③ Allocate GPU memory

- `cudaMalloc()`

CUDA programming model

CUDA Program
- written in C/C++ with extensions



① Data CPU ➔ GPU

② Data GPU ➔ CPU

- `cudaMemcpy()`

③ Allocate GPU memory

- `cudaMalloc()`

④ Launch "kernel" on device

Compiling CUDA

- The CUDA compiler is called `nvcc`
- The extension of a CUDA program file is `.cu`
- Standard CPU code is automatically piped to `g++`
  - Use `-ccbin=` if you use another compiler (e.g `icpc`)
- Two phase compilation
  - Compile phase: .cu to .o
  - Link phase: .o to .exe (use `-lcudart` if you use any other compiler than `nvcc` as linker)
- If you run into linking problems (undefined refs.)
  - Avoid mixing `.c` and `.cu` files (`.cpp` and `.cu` is fine)
  - Or use `extern "C" { … }` apropriately

Compiling CUDA

- We provide a `Makefile` template for exercises!
- Most important compiler flag: `-arch=sm_70`
  - Compile code for compute capability 7.0 (Volta)
  - Default is cc. 1.0 (Tesla), latest is cc. 7.0 (Volta)
- Other flags:`-Xptxas=-v`
  - Set output from ptxas compiler to verbose
- Other flags: `-g -G`
  - Generate debug information for host and device code
- Other flags: `-lineinfo`
  - Generate line-number information for device code (e.g., used in visual profiler)

Executing CUDA

- Execution model maps code to instructions
  - SIMD – Single Instruction, Multiple Data
  - SIMT – Single Instruction, Multiple Threads

Executing CUDA

- Execution model maps code to instructions
  - SIMD – Single Instruction, Multiple Data
  - SIMT – Single Instruction, Multiple Threads
- CPU (SIMD execution model)
  - The compiler takes care of mapping code to the most efficient instructions at compile time

Executing CUDA

- Execution model maps code to instructions
  - SIMD – Single Instruction, Multiple Data
  - SIMT – Single Instruction, Multiple Threads
- CPU (SIMD execution model)
  - The compiler takes care of mapping code to the most efficient instructions at compile time
- GPU (SIMT execution model)
  - Kernels look like serial functions for a SINGLE thread
  - CUDA will automatically launch on MANY threads
  - Code is mapped to instructions at runtime!

CUDA C extensions

Function qualifiers

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

Or leave it out…

- `__device__`
  - Inlined when deemed appropriate by the compiler
- `__noinline__`
  - Used to avoid inlining
- `__forceinline__`
  - Used to force the compiler to inline the function

See Appendix B.1 in the NVIDIA CUDA C Programming Guide for more details

Function qualifiers

- Combining the qualifiers
  - `__device__ __host__ void func()`

Function qualifiers

- Combining the qualifiers
  - `__device__ __host__ void func()`
- Use with Macro `__CUDA_ARCH__`

```
__host__ __device__ void func()
{
#if __CUDA_ARCH__ == 100
   // Device code path for compute capability 1.0
#elif __CUDA_ARCH__ == 200
   // Device code path for compute capability 2.0
#elif !defined(__CUDA_ARCH__)
   // Host code path
#endif
}
```

Vector types

- `char[1-4], uchar[1-4]`
- `short[1-4], ushort[1-4]`
- `int[1-4], uint[1-4]`
- `long[1-4], ulong[1-4]`
- `longlong[1-4], ulonglong[1-4]`
- `float[1-4]`
- `double1, double2`
- `dim3`

Vector types

- Available in host and device code
- Construct with `make_<type name>`
- When defining a variable of type `dim3`, any component left unspecified is initialized to 1
- Access with `.x`, `.y`, and `.z`

```
float4 f4 = make_float4(
  1.0f, 2.0f, 3.0f, 4.0f);
dim3 blocks = dim3(16,16);
int bx = blocks.x;
```

Math functions

- Partial list:
  - `sqrt`, `rsqrt`
  - `exp`, `log`
  - `sin`, `cos`, `tan`, `sincos`
  - `asin`, `acos`, `atan2`
  - `trunc`, `ceil`, `floor`
- On the host, functions use the C runtime implementation if available `<math.h>`

Math functions

- On the device, also intrinsic math functions are available (reminiscent from fast graphics):
  - Device only
  - Faster, but less accurate
  - Prefixed with `__`
  - `__exp`, `__log`, `__sin`, `__pow`, …
- Use explicitly or force all math to be intrinsic using `-use_fast_math` compiler option

Launching CUDA kernels

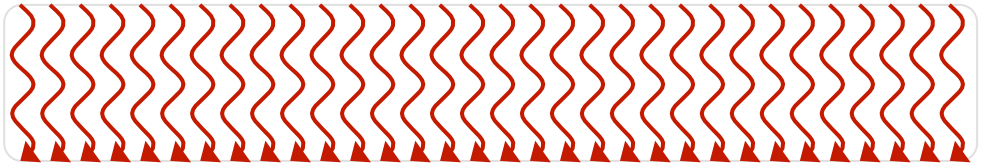
CUDA thread hierarchy

- Thread
  - Smallest “unit of parallelism”
  - Lightweight – OS not involved
  - Has its own independent control flow
    - Program counter (since CUDA 9.0)
    - Call stack (since CUDA 9.0)
    - Register file allocation
  - No fork / join / #omp parallel (launched automatically)

Thread

CUDA thread hierarchy

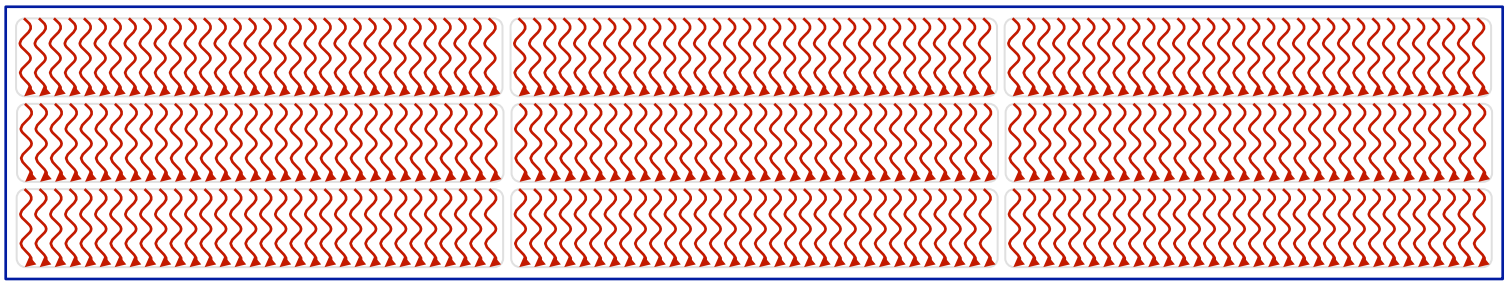
- Warp
  - “Unit of execution”
  - 32 threads – neighbors grouped together
  - Must always execute the same instruction (synchrony)
    - Scheduling individual threads is possible but the rest will be idle during the execution (since CUDA 9.0)
  - Dispatch and re-dispatch has no overhead
  - Warp is a terrible name (OpenCL: work-subgroup)



Warp

CUDA thread hierarchy

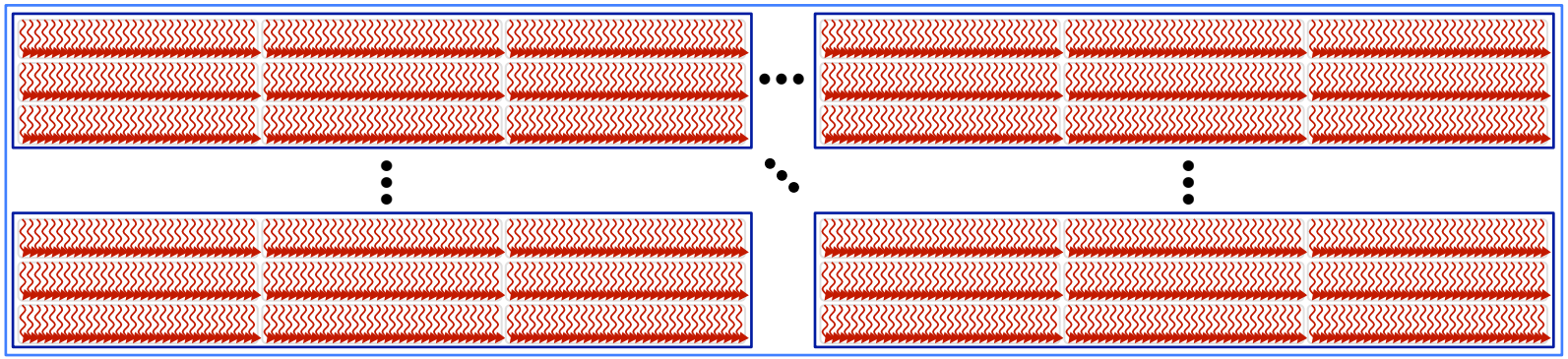
- Block
  - "Unit of SM scheduling"
  - Up to 1024 threads – 1D, 2D, or 3D neighbors
  - Think of warps grouped together
  - Scheduled to an SM at runtime when vacancy appears
  - All threads will be resident at same SM (same L1 cache)



Block

CUDA thread hierarchy

- Grid
  - "Total number of threads launched for kernel"
  - Up to 2147483647 threads – 1D, 2D, or 3D neighbors
  - Think of blocks grouped together
  - Launch cost is approx. 100 nanoseconds



Grid

Launch configuration

- Kernel code defined once – same for all threads:

```
__global__ void kernel(...) {...};
```

Launch configuration

- Kernel code defined once – same for all threads:

```
__global__ void kernel(...) {...};
```

- Kernel launched from host:

```
kernel<<<NUM_BLOCKS, THREADS_PER_BLOCK>>>(...);
cudaDeviceSynchronize();
```

  - Kernels are launched asynchronously from host and therefore explicit synchronization is needed to make sure we are finished.

Launch configuration

- Kernel code defined once – same for all threads:

```
__global__ void kernel(...) {...};
```

- Kernel launched from host:

```
kernel<<<NUM_BLOCKS, THREADS_PER_BLOCK>>>(...);
cudaDeviceSynchronize();
```

  - Kernels are launched asynchronously from host and therefore explicit synchronization is needed to make sure we are finished.

- For example:

```
kernel<<<512,6>>>(...); // 3072 threads in total
```

Launch configuration

- Kernel code defined once – same for all threads:

```
__global__ void kernel(...) {...};
```

- Kernel launched from host:

```
kernel<<<NUM_BLOCKS, THREADS_PER_BLOCK>>>(...);
cudaDeviceSynchronize();
```

  - Kernels are launched asynchronously from host and therefore explicit synchronization is needed to make sure we are finished.

- For example:

```
kernel<<<512,6>>>(...); // 3072 threads in total
kernel<<<6,512>>>(...); // 3072 threads in total
```

Launch configuration

- Kernel code defined once – same for all threads:

```
__global__ void kernel(...) {...};
```

- Kernel launched from host:

```
kernel<<<NUM_BLOCKS, THREADS_PER_BLOCK>>>(...);
cudaDeviceSynchronize();
```

  - Kernels are launched asynchronously from host and therefore explicit synchronization is needed to make sure we are finished.

- For example:

```
kernel<<<512,6>>>(...); // 3072 threads in total
kernel<<<6,512>>>(...); // 3072 threads in total
kernel<<<48,64>>>(...); // 3072 threads in total
```

Launch configuration

- Kernel code defined once – same for all threads:

```
__global__ void kernel(...) {...};
```

- Kernel launched from host:

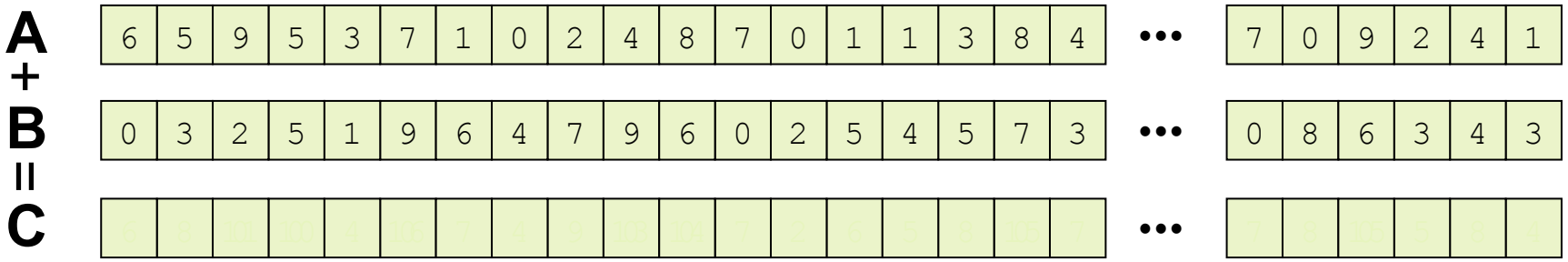
```
kernel<<<NUM_BLOCKS, THREADS_PER_BLOCK>>>(...);
cudaDeviceSynchronize();
```

  - Kernels are launched asynchronously from host and therefore explicit synchronization is needed to make sure we are finished.

- For example:

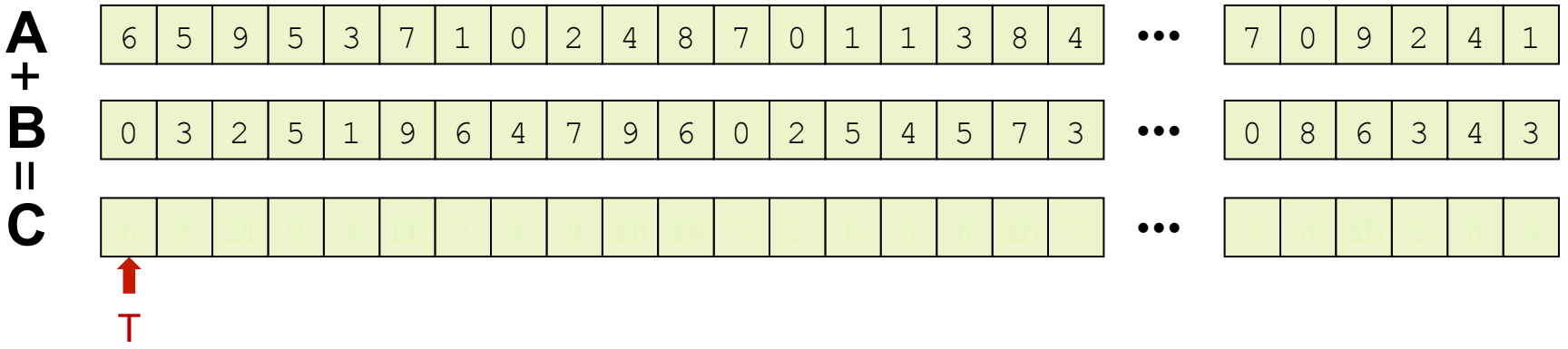
```
kernel<<<512,6>>>(...); // 3072 threads in total
kernel<<<6,512>>>(...); // 3072 threads in total
kernel<<<48,64>>>(...); // 3072 threads in total
kernel<<<1,3072>>>(...); // Not allowed (max 1024)
```

Simple example




```
// Sequential code run on host
void vecadd(double *a, double *b, double *c, int n)
{
  for(int i=0; i<n; i++)
     c[i] = a[i] + b[i];
}
```

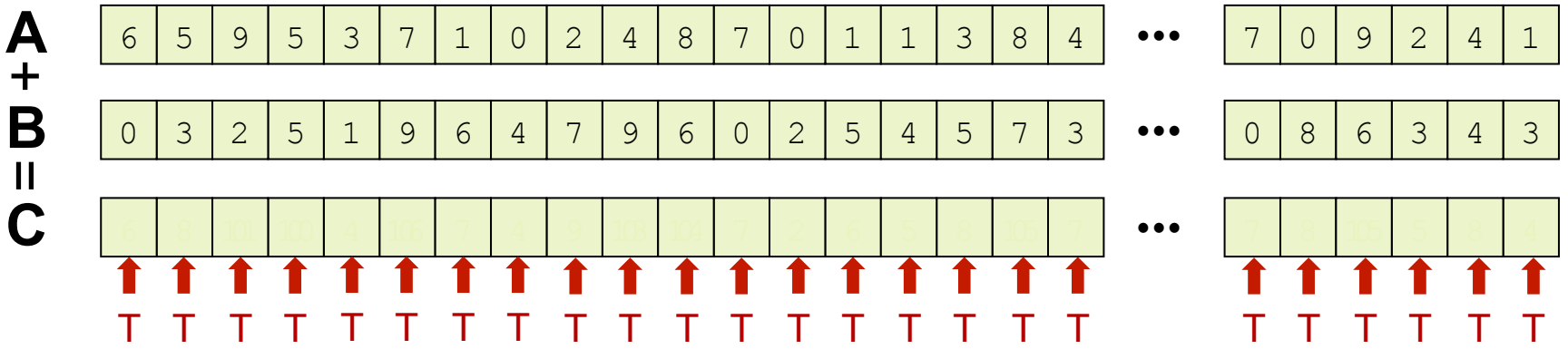
Simple example




```
// Sequential code launched on device using 1 thread
__global__
void vecadd(double *a, double *b, double *c, int n)
{
  for(int i=0; i<n; i++)
     c[i] = a[i] + b[i];
}
```

```
vecadd<<<1, 1>>>(a, b, c, n); // 1 thread in total
cudaDeviceSynchronize();
```

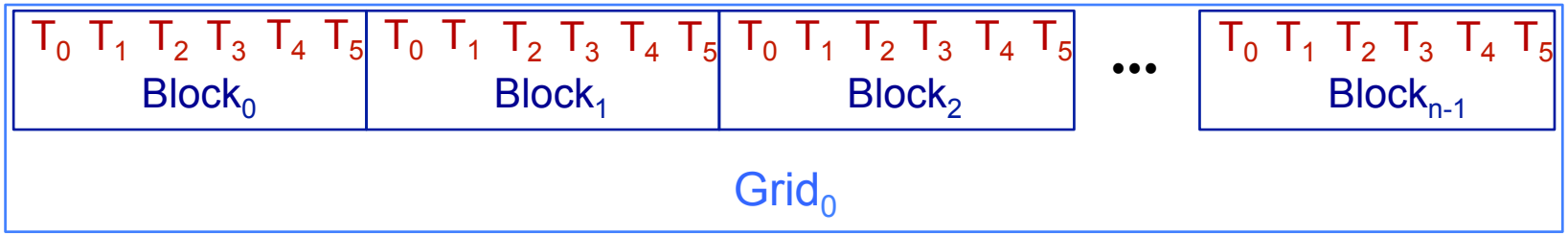
Simple example



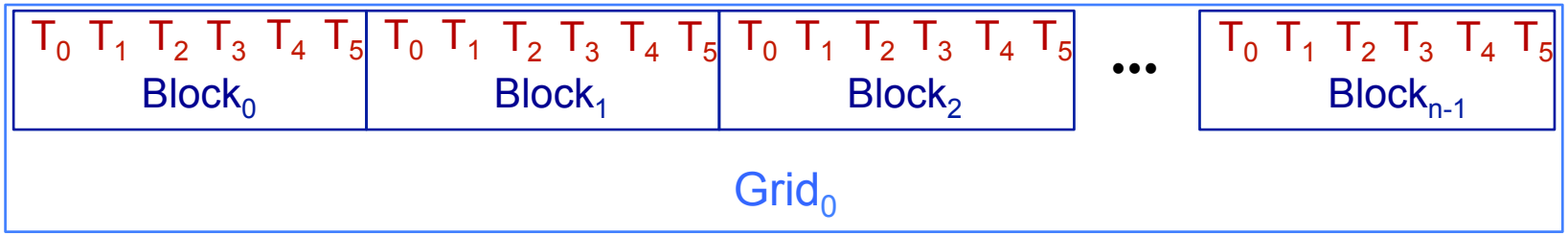

```
// Parallel code launched on device using n threads
__global__
void vecadd(double *a, double *b, double *c, int n)
{
  int i = <some_way_of_getting_the_thread_num>;
  c[i] = a[i] + b[i];
}
```

```
vecadd<<<n/256, 256>>>(...); // n threads in total
cudaDeviceSynchronize();
```

Which thread am I?

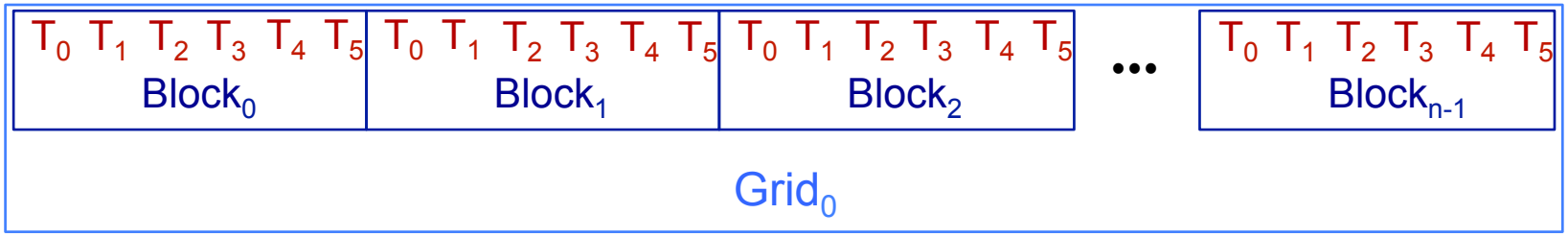


Which thread am I?



- Built-in variables
  - `threadIdx.x, threadIdx.y threadIdx.z`
  - `blockIdx.x, blockIdx.y, blockIdx.z`
  - `blockDim.x, blockDim.y, blockDim.z`
  - `gridDim.x, gridDim.y, gridDim.z`

Which thread am I?



- Built-in variables
  - `threadIdx.x, threadIdx.y threadIdx.z`
  - `blockIdx.x, blockIdx.y, blockIdx.z`
  - `blockDim.x, blockDim.y, blockDim.z`
  - `gridDim.x, gridDim.y, gridDim.z`
- 1D vector addition

```
int i = blockIdx.x * blockDim.x + threadIdx.x;
c[i] = a[i] + b[i];
```

Launching kernels for 2D/3D

- Kernel code defined once – same for all threads:

```
__global__ void kernel(...) {...};
```

Launching kernels for 2D/3D

- Kernel code defined once – same for all threads:

```
__global__ void kernel(...) {...};
```

- Kernel launch configuration specified at host:

```
dim3 dimGrid(512,8,1); // 4096 blocks in total
dim3 dimBlock(16,16,1);// 256 threads per block
```

Launching kernels for 2D/3D

- Kernel code defined once – same for all threads:

```
__global__ void kernel(...) {...};
```

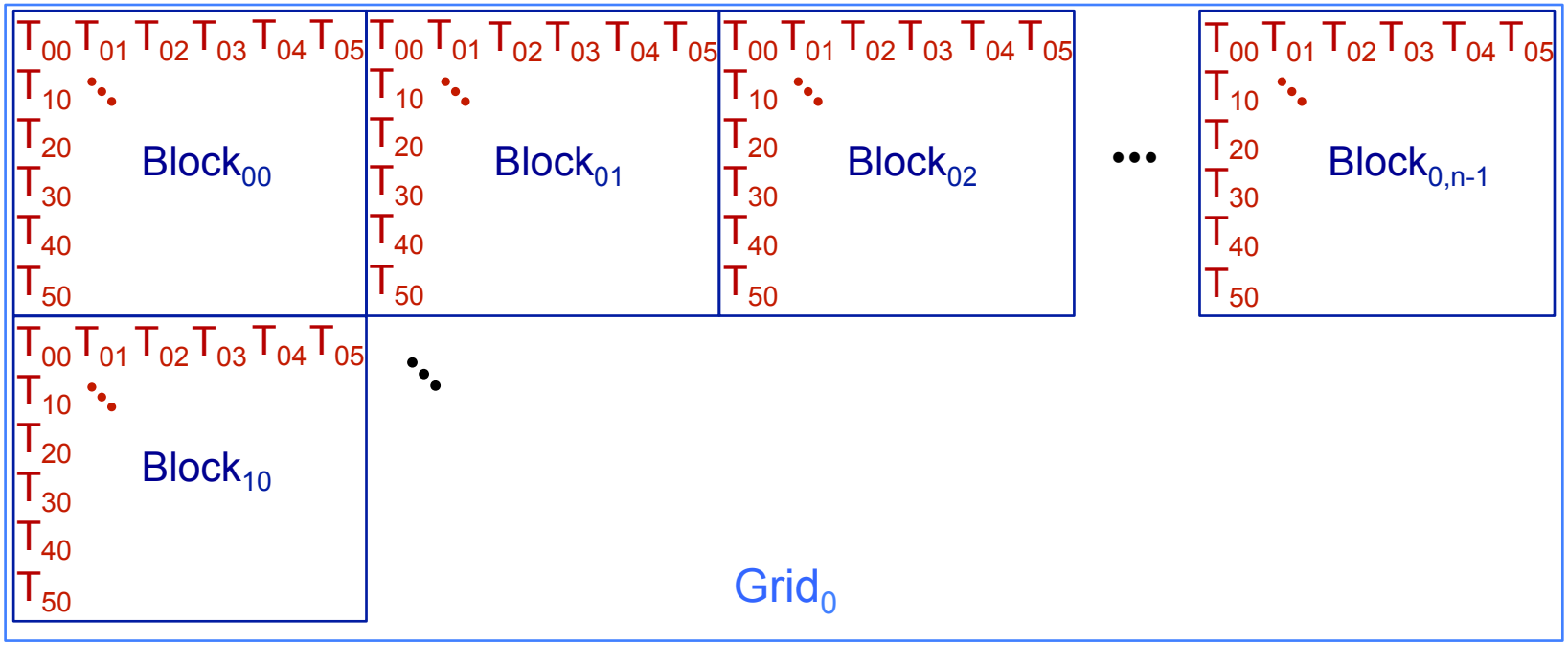
- Kernel launch configuration specified at host:

```
dim3 dimGrid(512,8,1); // 4096 blocks in total
dim3 dimBlock(16,16,1);// 256 threads per block
```

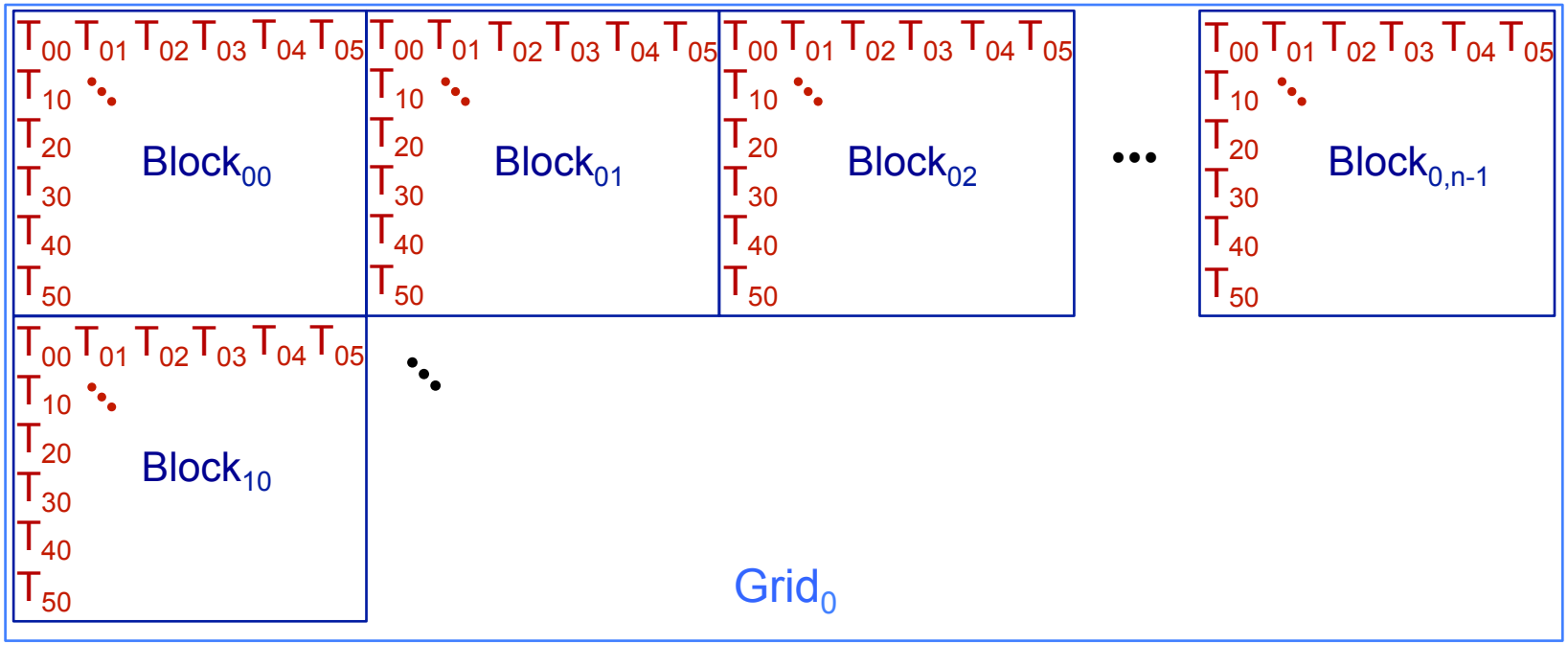
- Kernel launched from host:

```
kernel<<<dimGrid, dimBlock>>>(...);
cudaDeviceSynchronize();
```

Which thread am I?



Which thread am I?



- 2D matrix addition

```
// 2D thread indices defining row and col of element
int col = blockIdx.x * blockDim.x + threadIdx.x;
int row = blockIdx.y * blockDim.y + threadIdx.y;
c[row*N + col] = a[row*N + col] + b[row*N + col];
```

Launching kernels

- Hardware limits

Query		Compute Capability		
		1.x (Tesla)	2.x (Fermi)	3.x (Kepler) - 7.x (Volta)
Threads per block		512	1024	1024
Grid size	`gridDim.x`	65535	65535	2147483647
	`gridDim.y`	65535	65535	65535
	`gridDim.z`	1	65535	65535
Block size	`blockDim.x`	512	1024	1024
	`blockDim.y`	512	1024	1024
	`blockDim.z`	64	64	64

Typical structure of CUDA main()

```
int main(int argc, char **argv)
{
    // Allocate memory space on host and device
    h_data = malloc(...);
    cudaMalloc(...);

    // Transfer data from host to device
    cudaMemcpy(...);

    // Kernel lauch
    kernel<<<Grid, Block>>>(...);
    cudaDeviceSynchronize();

    // Transfer results from device to host
    cudaMemcpy(...);

    // Free memory
    free(h_data);
    cudaFree(...);
}
```

Warmup of GPUs

- It takes time to get a context on a CUDA device
  - Idle GPUs are in power saving mode
  - Just-in-time compilation
  - Transfer of kernel to GPU memory
  - Approx. 0.5 seconds (on our nodes)

Warmup of GPUs

- It takes time to get a context on a CUDA device
  - Idle GPUs are in power saving mode
  - Just-in-time compilation
  - Transfer of kernel to GPU memory
  - Approx. 0.5 seconds (on our nodes)
- Warmup run
  - Required for accurate performance benchmarking in case of short runtimes
  - First CUDA API call that modifies the GPU context will initiate ‘warm up’ of the device
  - E.g. `cudaMalloc()`

- Do the first exercise
  - ex1_deviceQuery
  - Please note that `nvcc` with CUDA 9.1 requires `gcc` version 6.x or older; e.g. `module load gcc/6.3.0`
  - Template Makefiles are available on CampusNet
- Then start the second exercise
  - ex2_helloworld
- Next presentation “CUDA Memory Model” at 13.00 (Monday)!

End of lecture