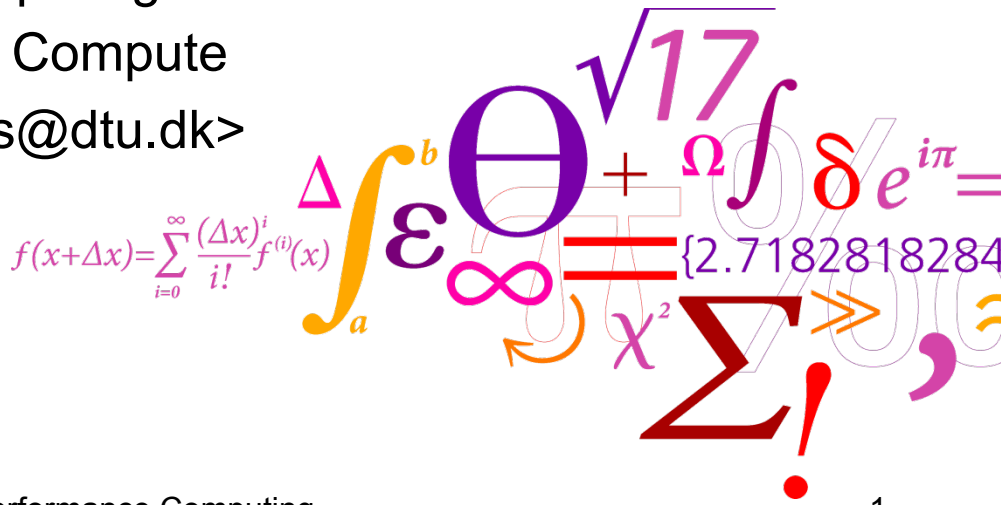
CUDA Programming Model

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



Acknowledgements

- Some slides are from Allan P. Engsig-Karup, DTU Compute.
- 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 from NVIDIA Developer
  - https://developer.nvidia.com/

Overview

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

CUDA / OpenCL

What is CUDA?

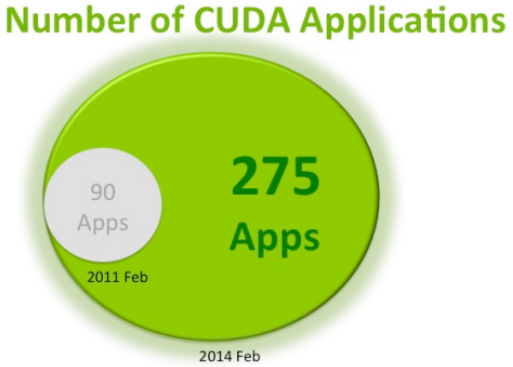
DTU

Compute Unified Device Architecture

- 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

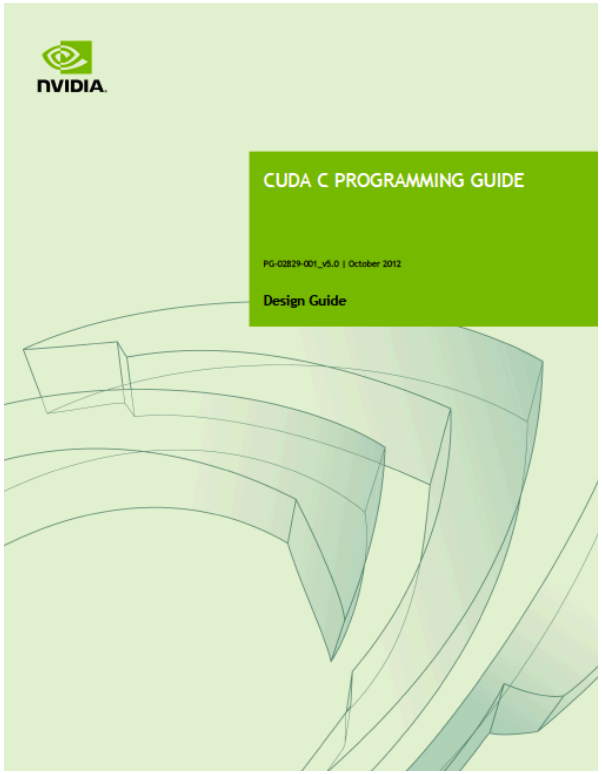
- CUDA-Capable GPUs
  - More than 506 mill. (2012: 375 mill.)
- Toolkit downloads
  - More than 2.200.000 (1.000.000)
- Active Developers
  - More than 190.000 (120.000)
- Universities Teaching CUDA
  - More than 738 (500) [in Denmark DTU, AU and KU].

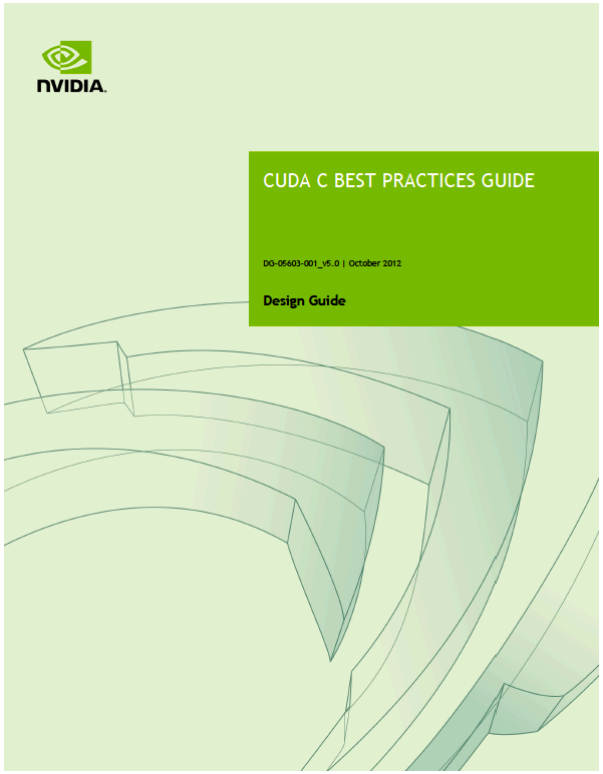


- 1 download every 60 seconds!

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

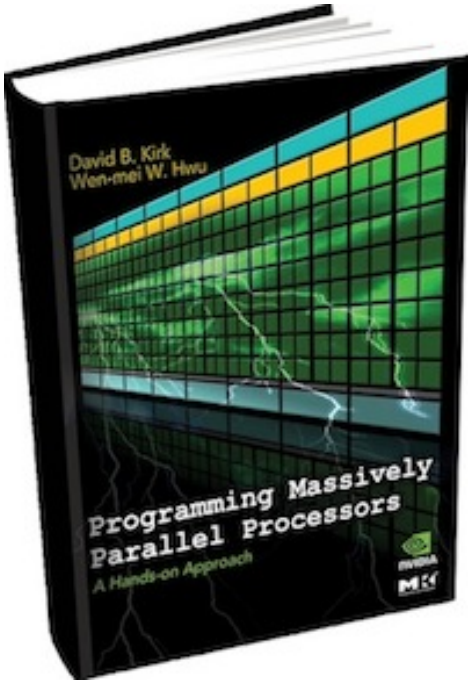
Free CUDA material

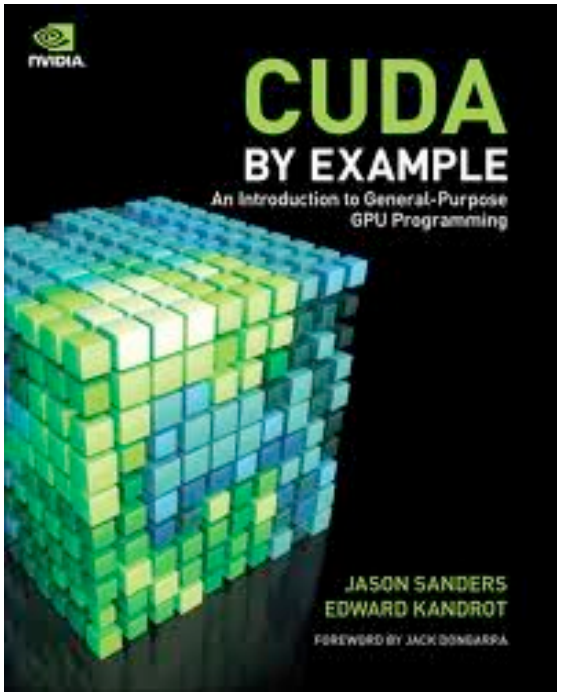


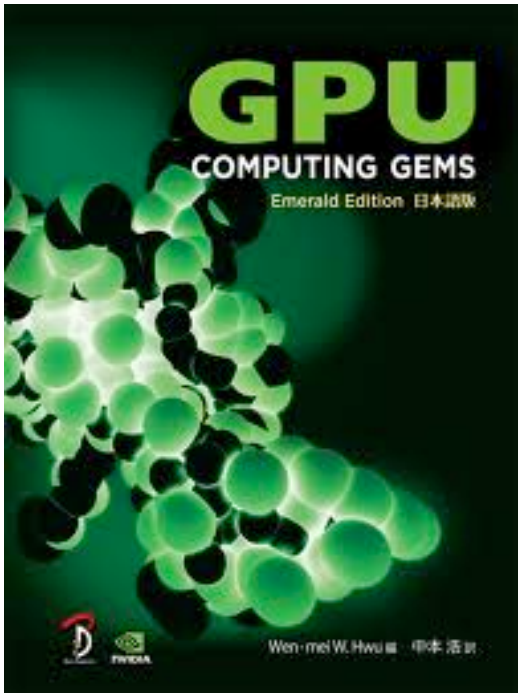


- Free online from Nvidia developer webpage
  - http://docs.nvidia.com/cuda/index.html
  - Pdf versions, see installation; /appl/cuda/8.0/doc/pdf/

Additional CUDA material

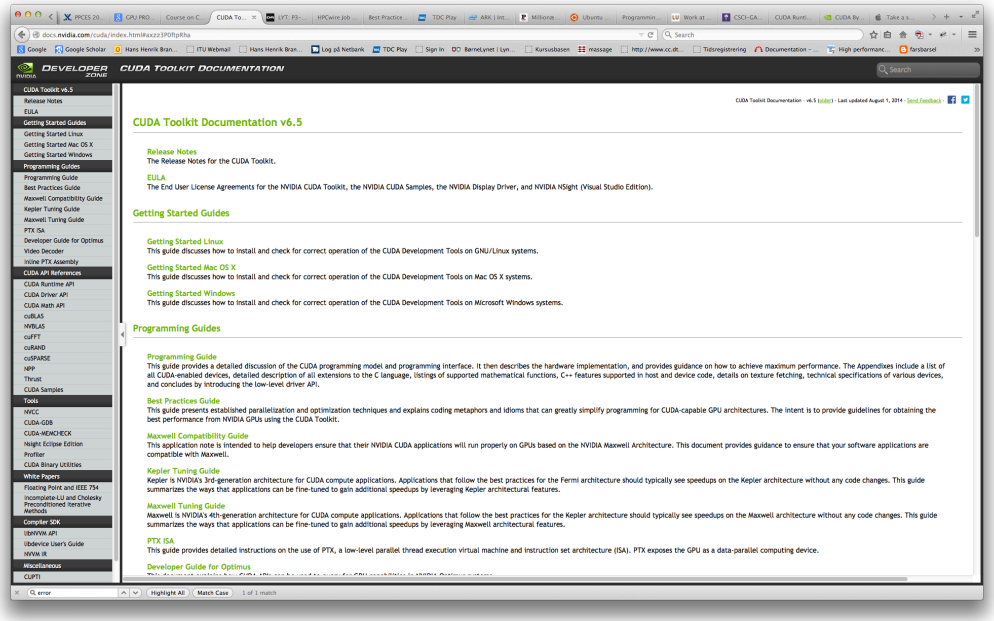




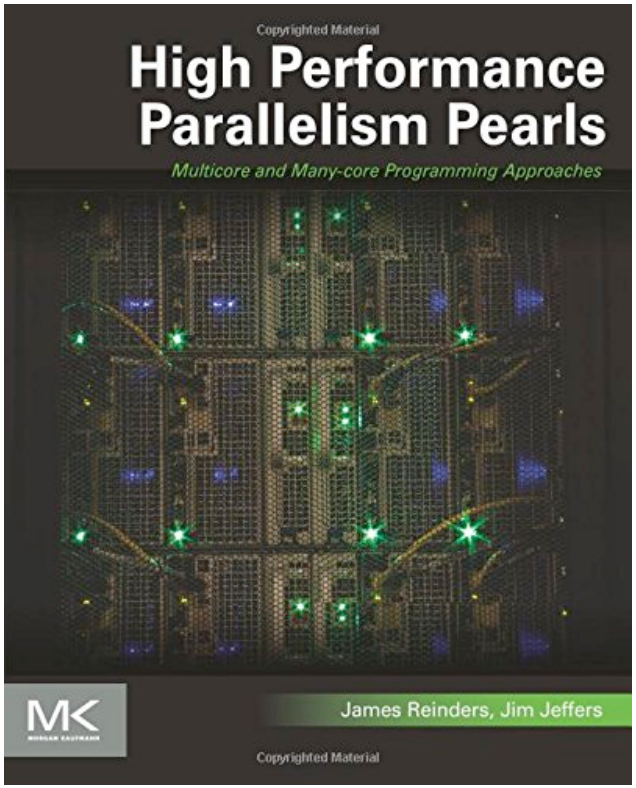


- These are currently the most widely used books
- Feel free to come by my office building 324, room 280 to take a peek in these references

Our suggestion (if you get hooked)



+



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

What is OpenCL?

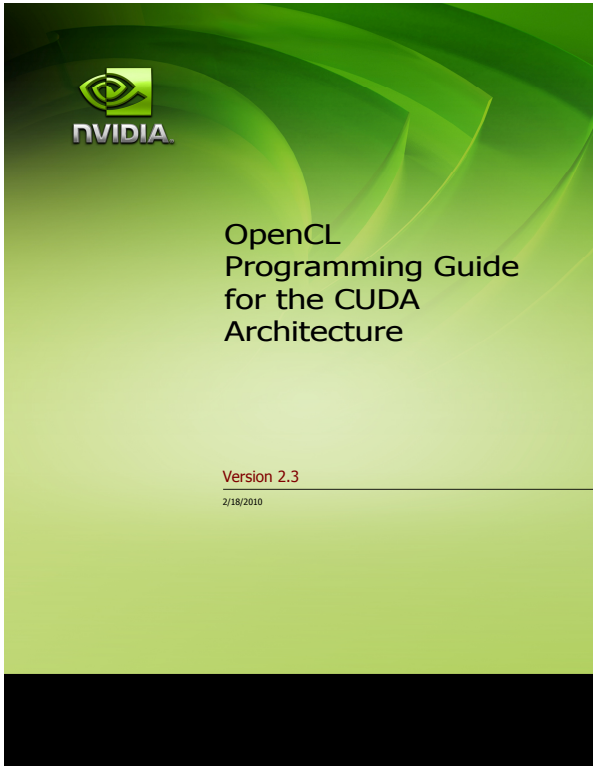
DTU

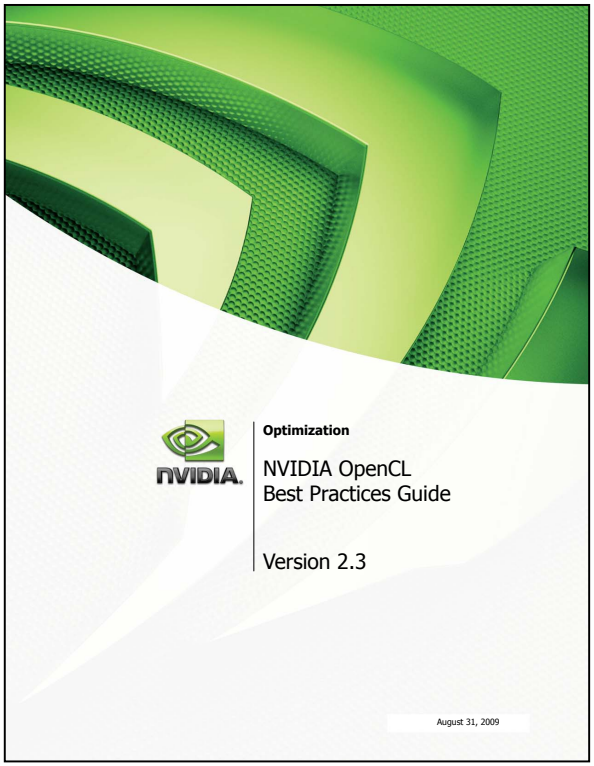
Open Computing Language (current v2.2)

- 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

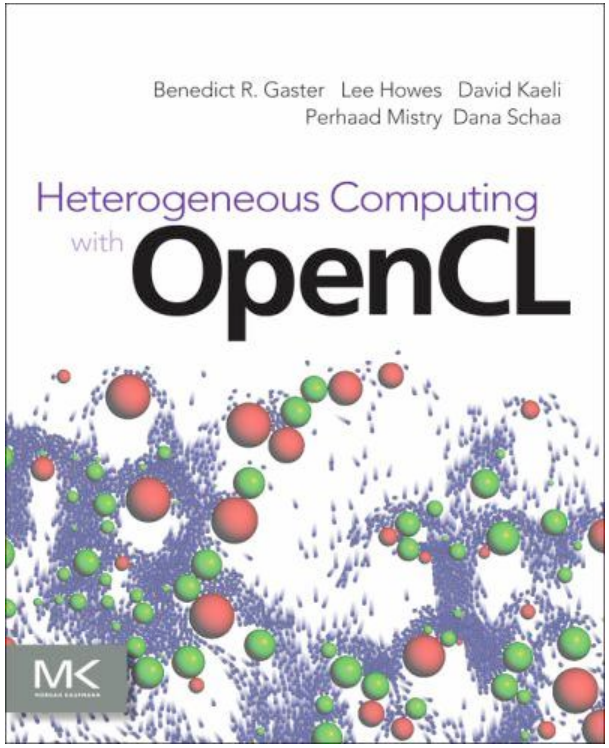
Free OpenCl material

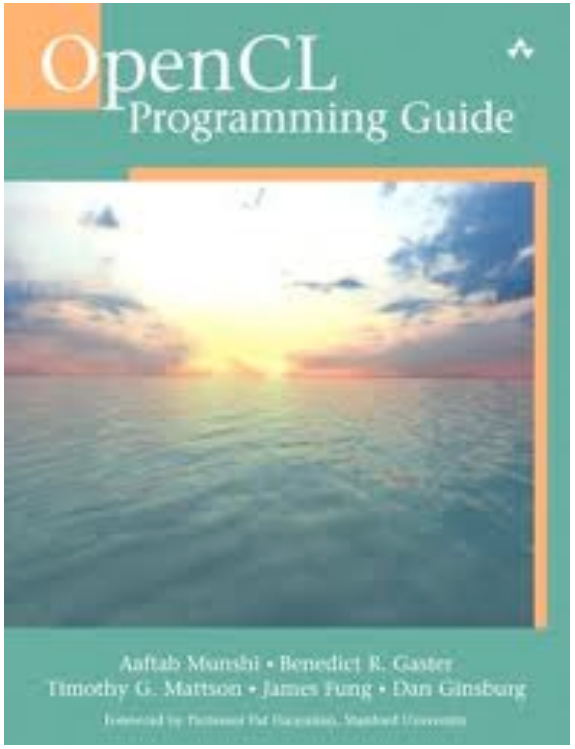


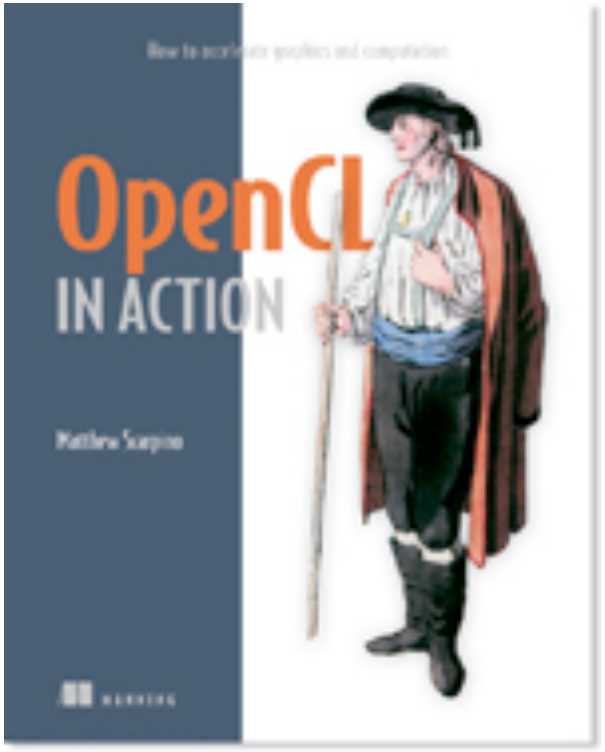


- Free from Nvidia and AMD developer webpages
  - https://developer.nvidia.com/opencl
  - Pdf versions, see installation; /usr/local/cuda/doc/pdf/

Additional OpenCL material







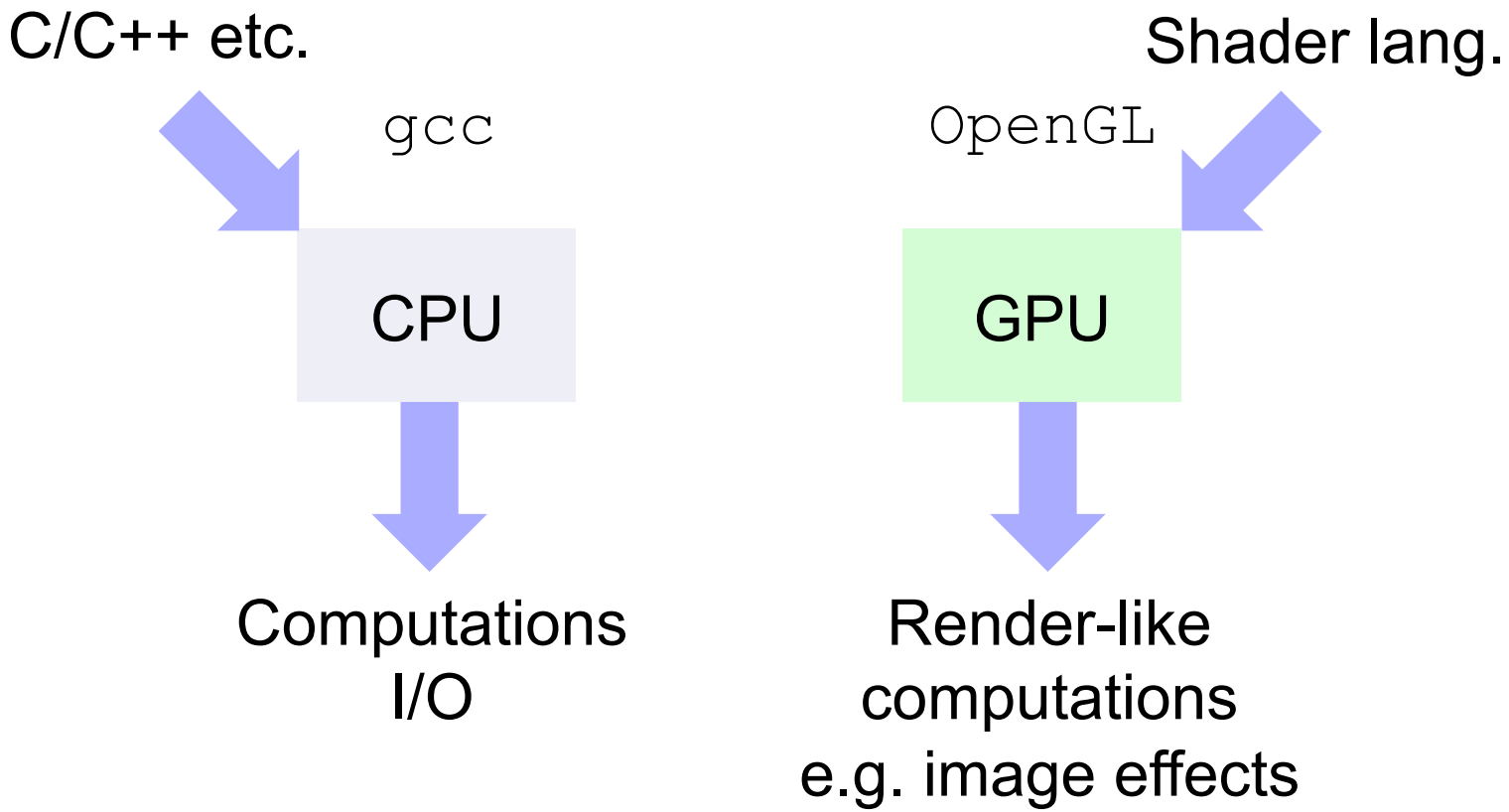
- All published within the last four years
  - Expect more to be published in the future
  - Feel free to come by 324-280 to take a peek

Why CUDA in this course?

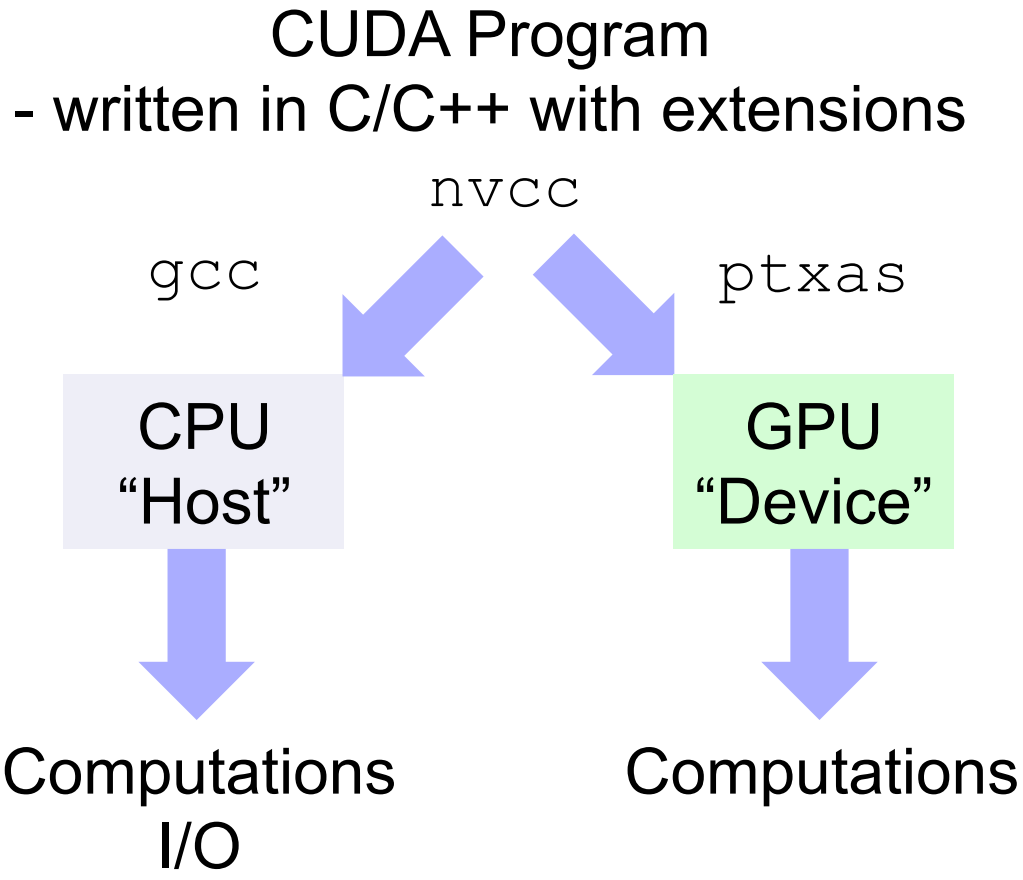
- CUDA and OpenCL are very similar
  - Most CUDA features map one-to-one to OpenCL features (only the syntax is different)
  - OpenCL provides more explicit functionality and is supported by a less mature software framework
- CUDA comes with tuned high-performance libs
  - OpenCL have less effort in this direction (currently)
- CUDA is well documented (by Nvidia)
- Nvidia products are widely used in HPC (>95%)
  - OpenCL still lags in performance for Nvidia products

CUDA programming model

Earlier than 2007

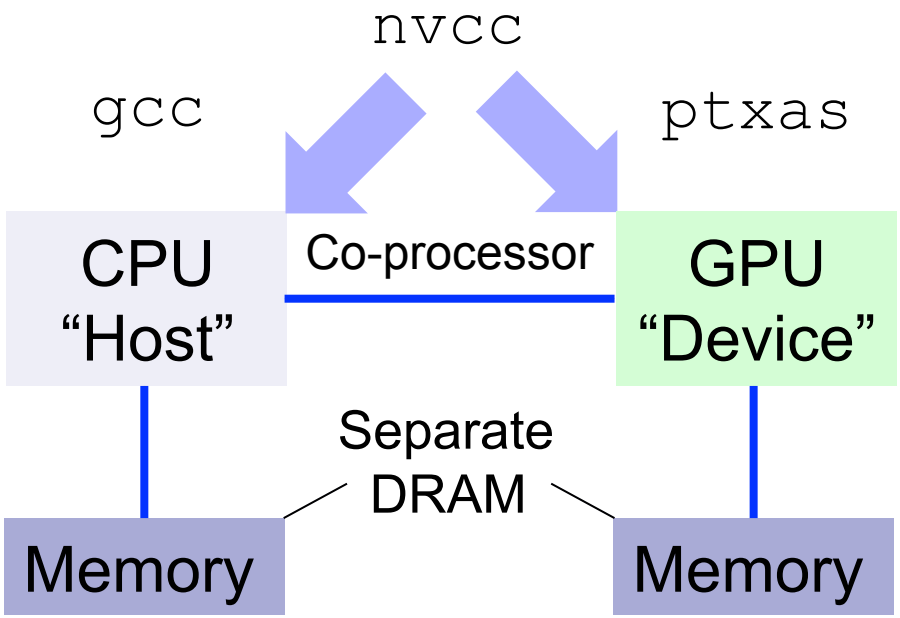


CUDA released



CUDA programming model

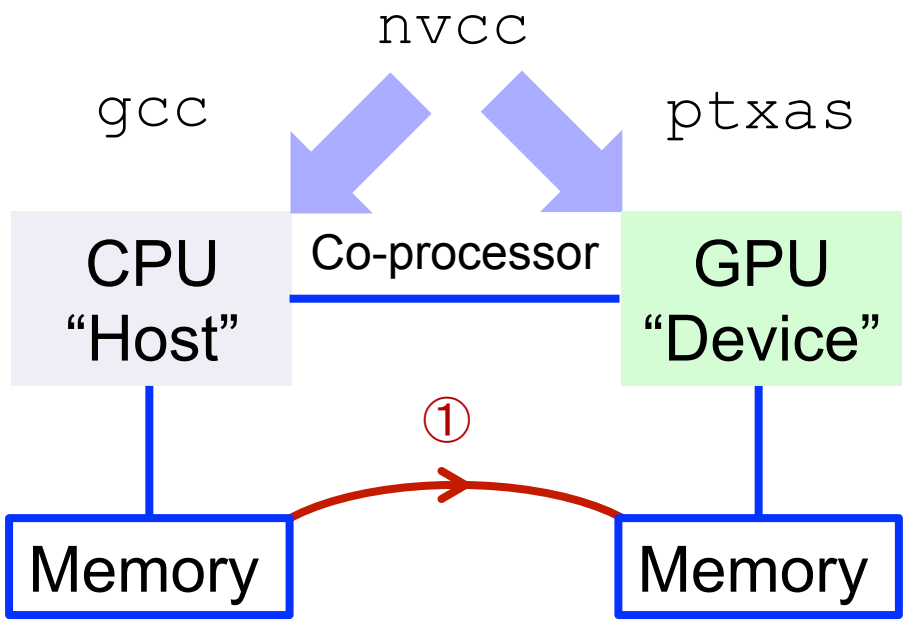
CUDA Program
- written in C/C++ with extensions



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

CUDA programming model

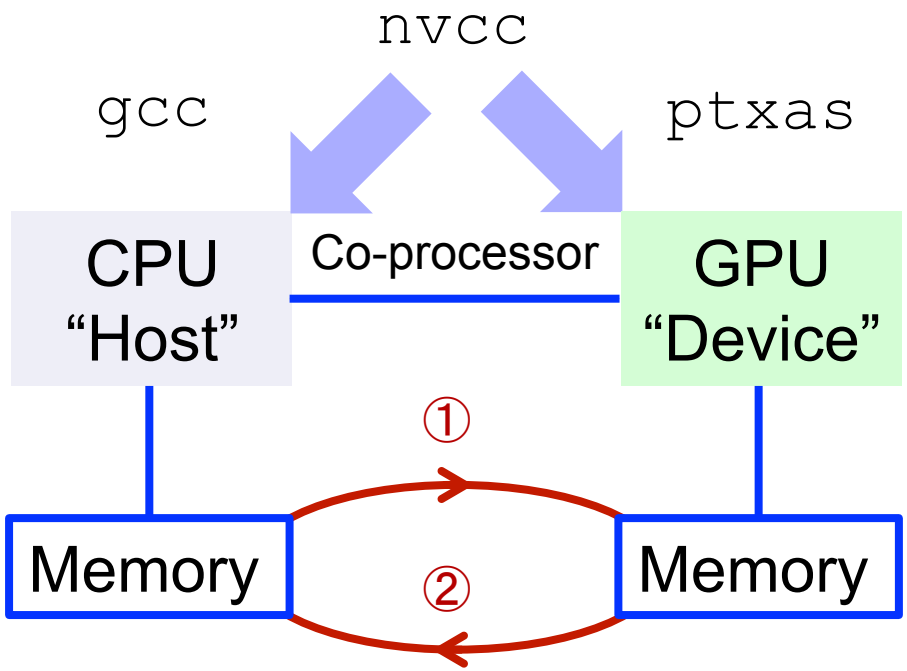
CUDA Program
- written in C/C++ with extensions



① Data CPU ➔ GPU

CUDA programming model

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

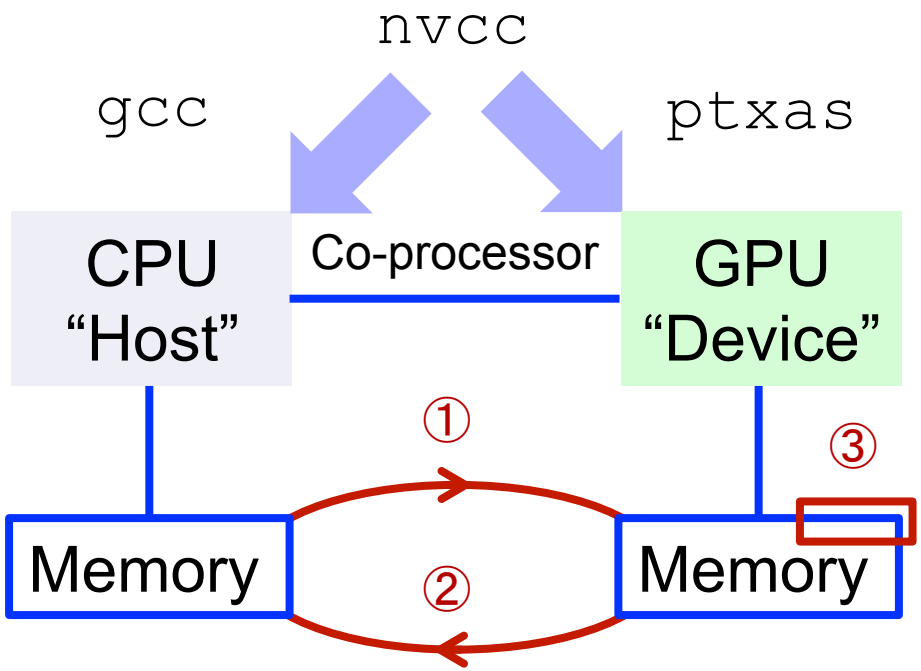


① Data CPU ➔ GPU

② Data GPU ➔ CPU

CUDA programming model

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



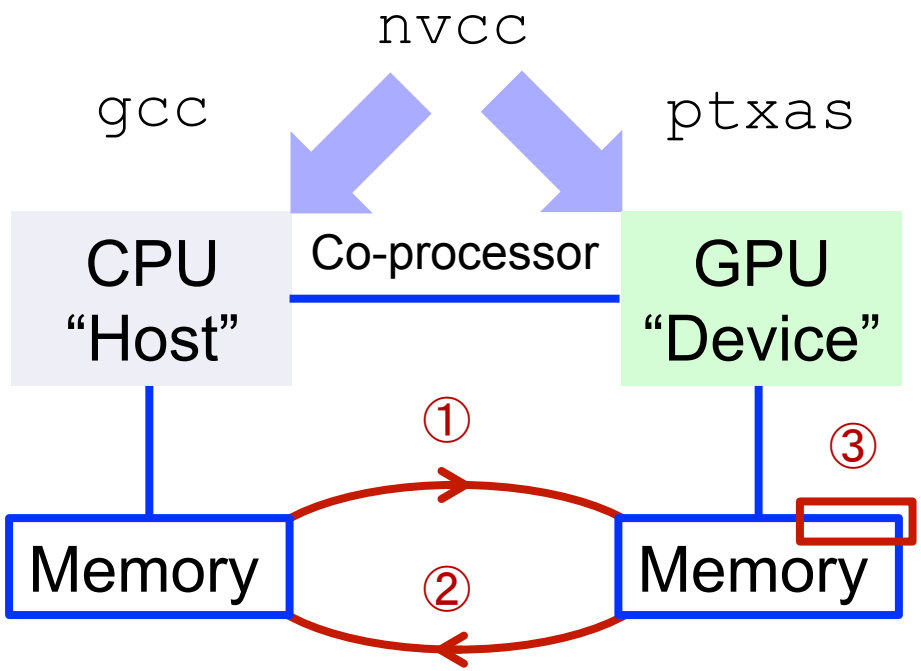
① Data CPU ➔ GPU

② Data GPU ➔ CPU

③ Allocate GPU memory

CUDA programming model

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



① Data CPU ➔ GPU

② Data GPU ➔ CPU

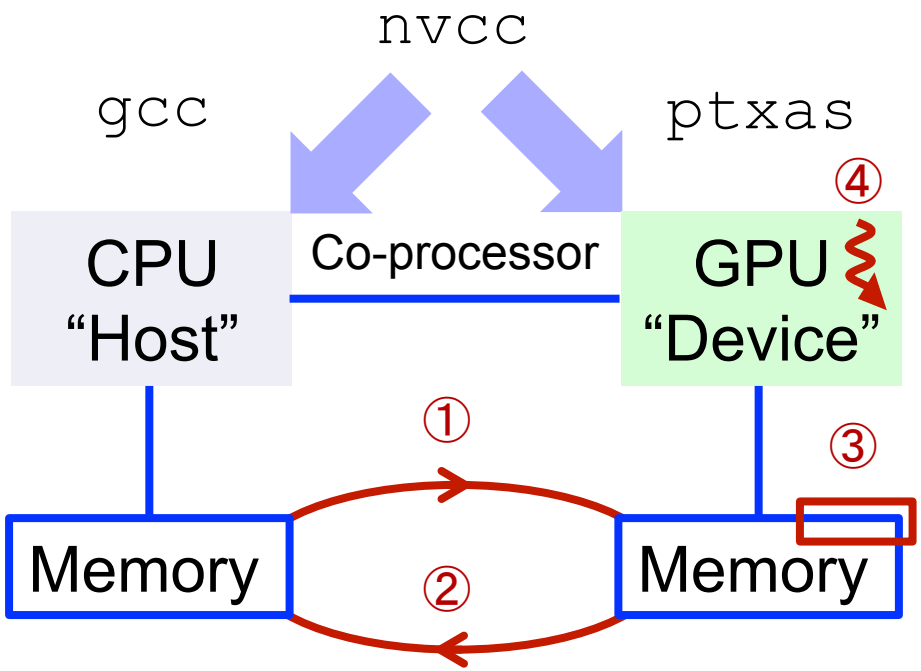
- `cudaMemcpy()`

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 kernels on device

SIMT execution model

- Kernels look like serial programs
- Programs are written as if they run on ONE thread
- CUDA will run that program on MANY threads mapped to the SIMD hardware ALUs (one-to-one)

What is a GPU good at?

① Efficiently launching lots of threads

② Running lots of threads in parallel

Compiling CUDA

- The CUDA compiler is called `nvcc`
- The extension of a CUDA program file is `.cu`
- Since CUDA 5.0 nvcc works as you would expect
  - Compile phase: .cu to .o
  - Link phase: .o to .exe (use `-lcudart` if you use `gcc`)
- Standard CPU code is automatically piped to `gcc`
- We provide a `Makefile` template for exercises!
- Most important compiler flag: `-arch=sm_30`
  - Compile code for compute capability 3.0 (Kepler)
  - Default is cc. 1.0 (Tesla), latest is cc. 6.0 (Pascal)

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

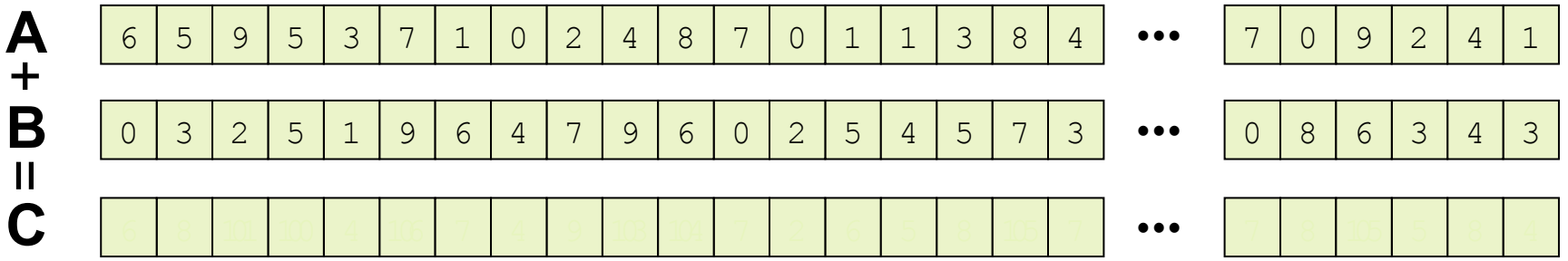
- Partial list:
  - `sqrt`, `rsqrt`
  - `exp`, `log`
  - `sin`, `cos`, `tan`, `sincos`
  - `asin`, `acos`, `atan2`
  - `trunc`, `ceil`, `floor`
- On the host, functions use the C runtime implementation if available `<math.h>`

Math functions

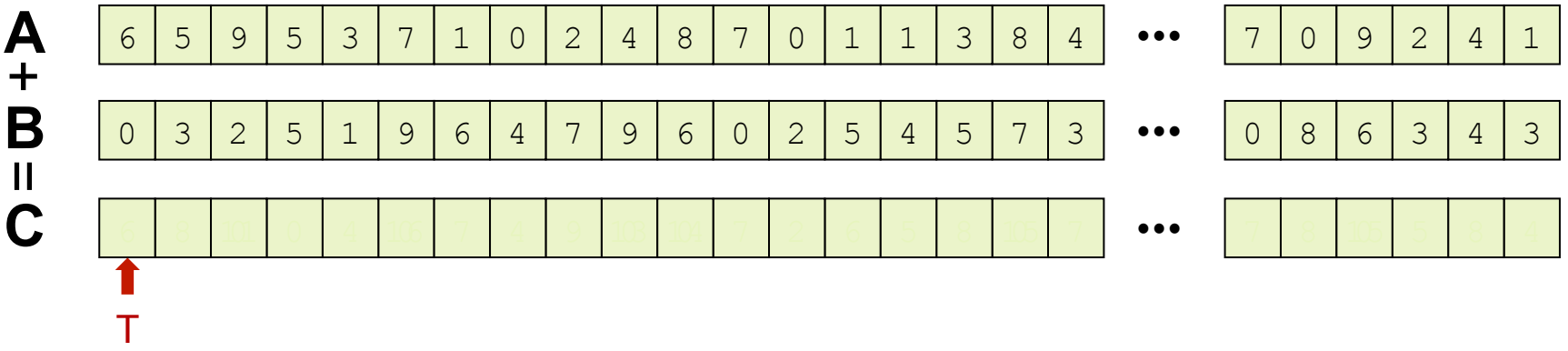
- On the device, also intrinsic math functions are available (reminiscent from fast graphics):
  - Device only
  - Faster, but less accurate
  - Prefixed with `__`
  - `__exp`, `__log`, `__sin`, `__pow`, …
- Use explicitly or force all math to be intrinsic using `-use_fast_math` compiler option

Warps, Blocks and Grids

Simple example



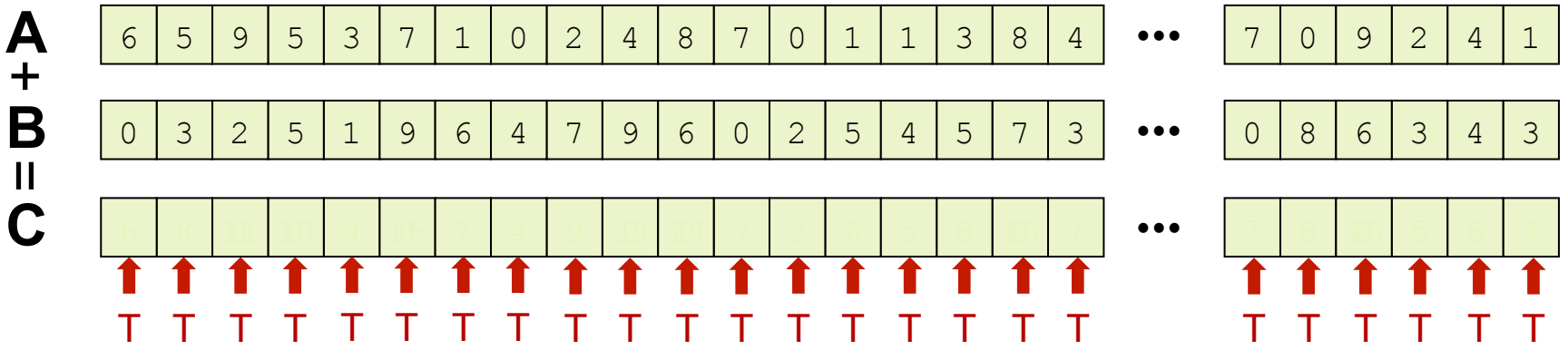
Simple example




```
__global__
void vecadd(double *a, double *b, double *c, int n)
{
   for(int i=0; i<n; i++)
      c[i] = a[i]+b[i];
}
```

In CUDA – as in C – it is still possible to write a sequential loop so one thread does all elements

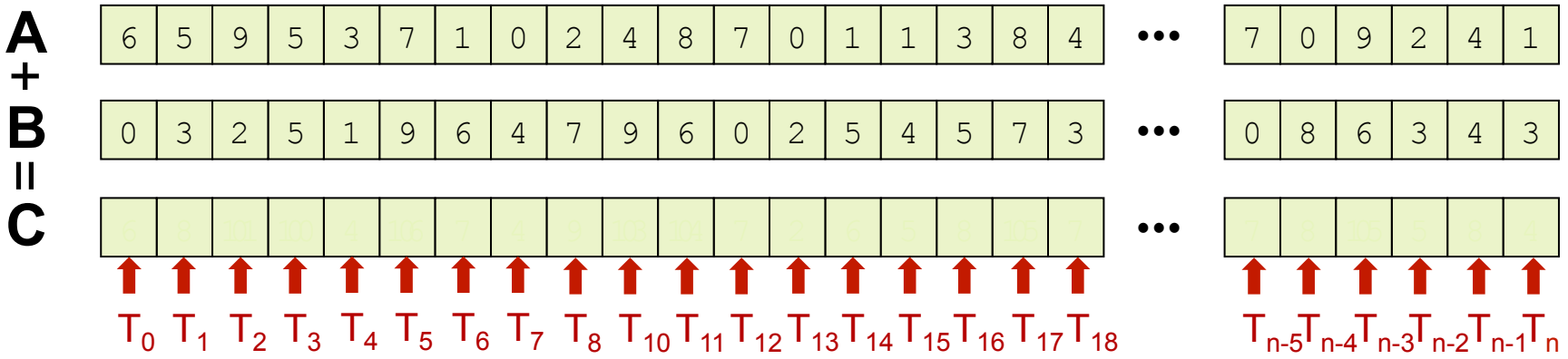
Simple example




```
__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];
}
```

However, since GPUs are efficient at launching threads we might also launch a thread per element

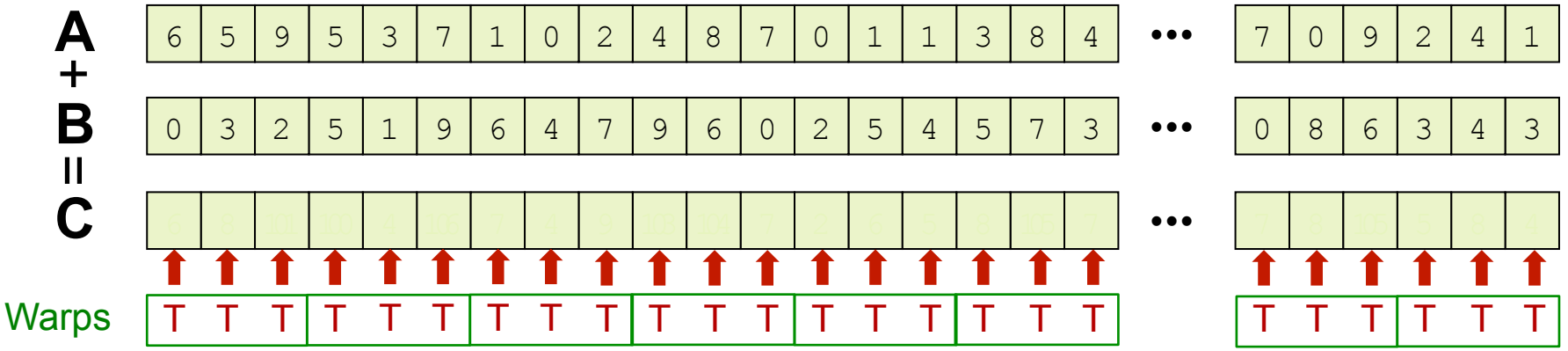
Simple example




```
__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];
}
```

A million threads cannot run simultaneously – but SIMT is fastest if neighbor threads are coalesced

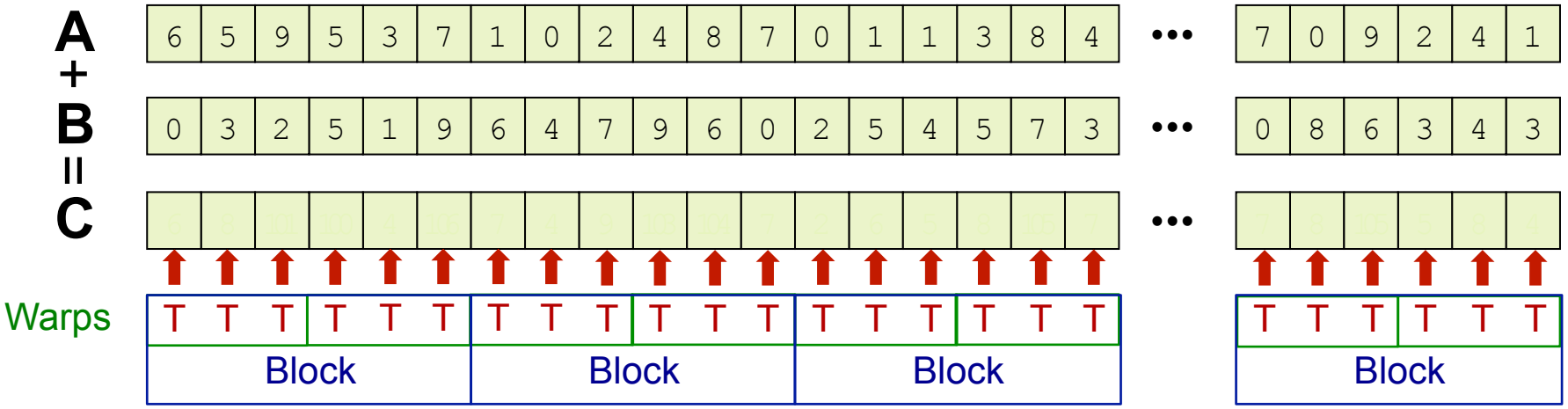
Simple example




```
__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];
}
```

The CUDA execution model will group 32 neighbor threads – “warps” – to run on SIMD units (32 lanes)

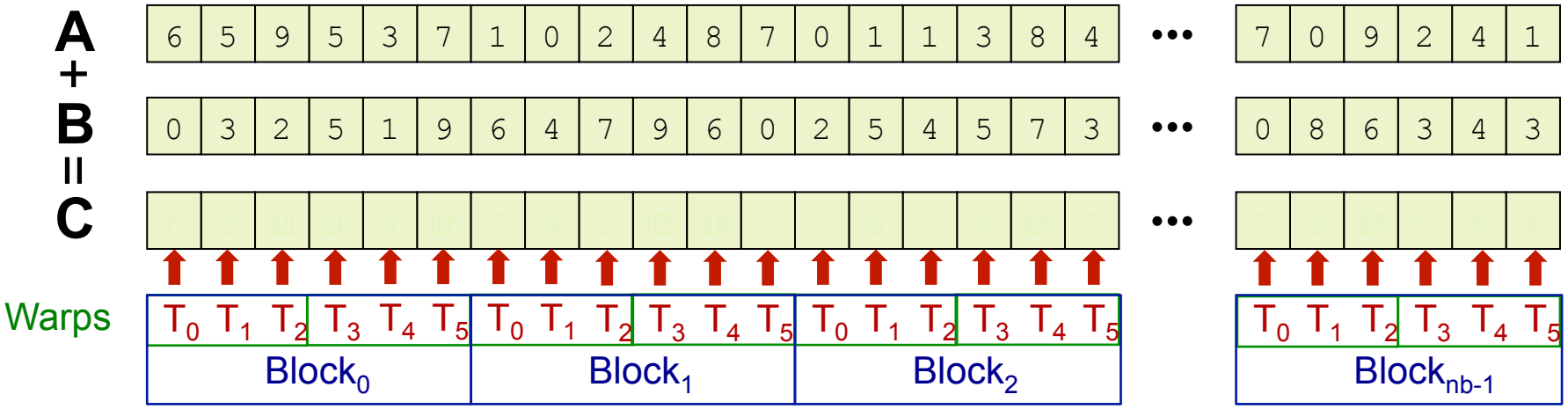
Simple example




```
__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];
}
```

CUDA will schedule which warps are executing simultaneously by grouping them into blocks

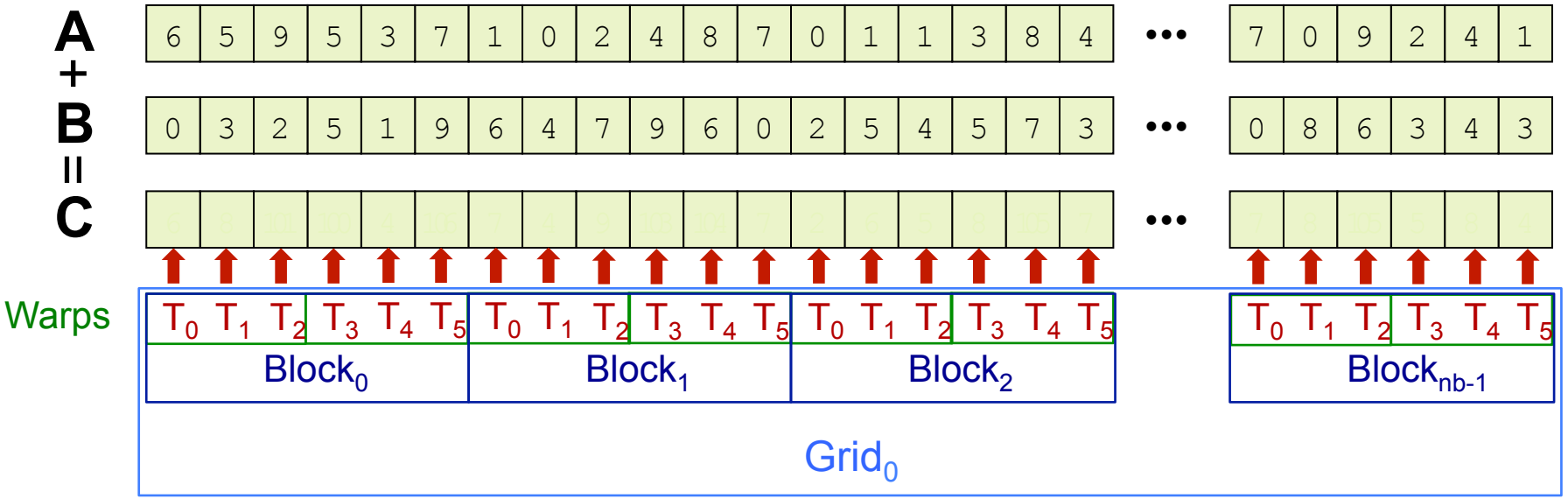
Simple example




```
__global__
void vecadd(double *a, double *b, double *c, int n)
{
  int i = <block_i * num_threads_per_block + thread_i>;
  c[i] = a[i]+b[i];
}
```

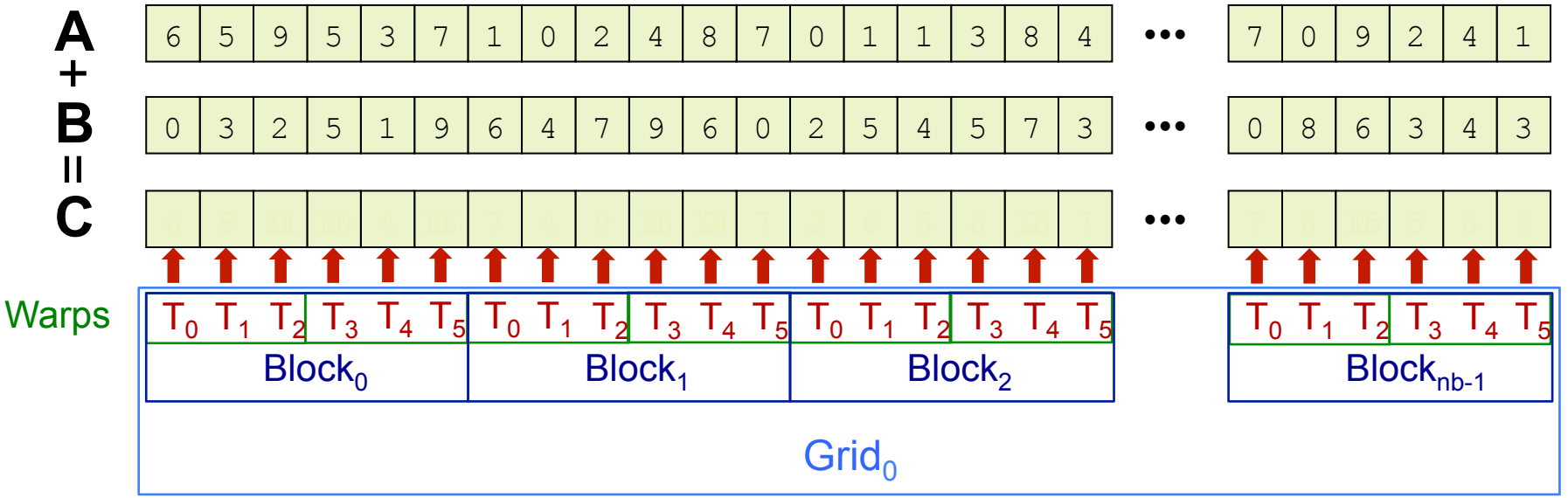
By enumerating all blocks and all threads within a block we can calculate the unique thread number

Simple example

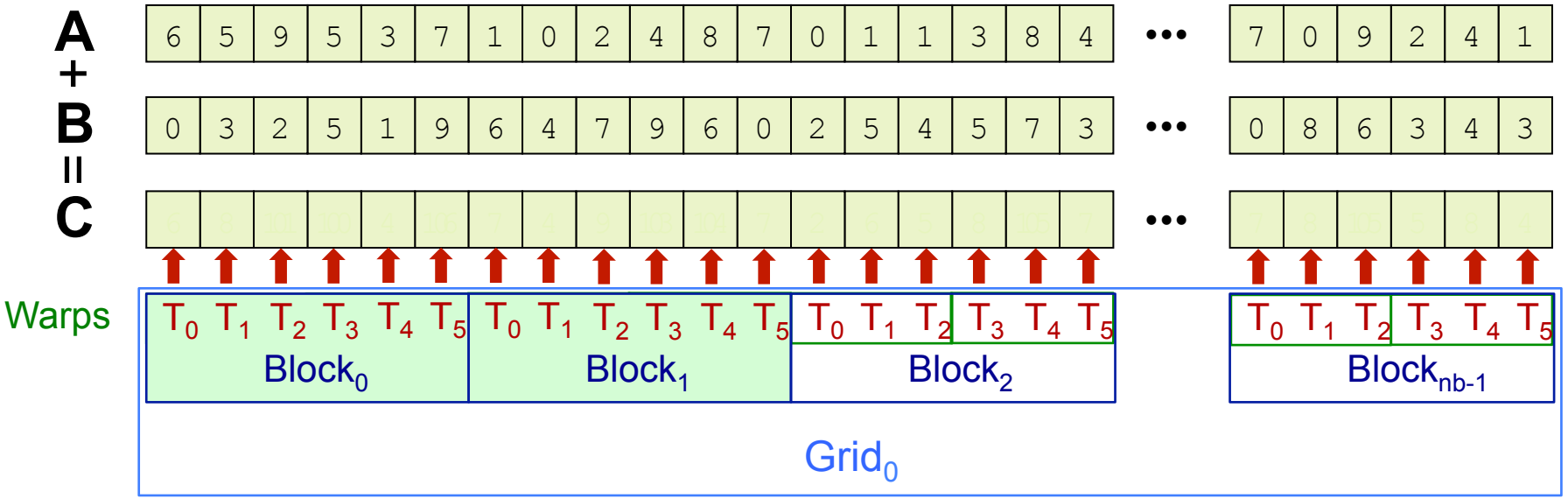


CUDA introduces grids, which are a grouping of the blocks, for launching more kernels at a time that may have different thread hierarchies

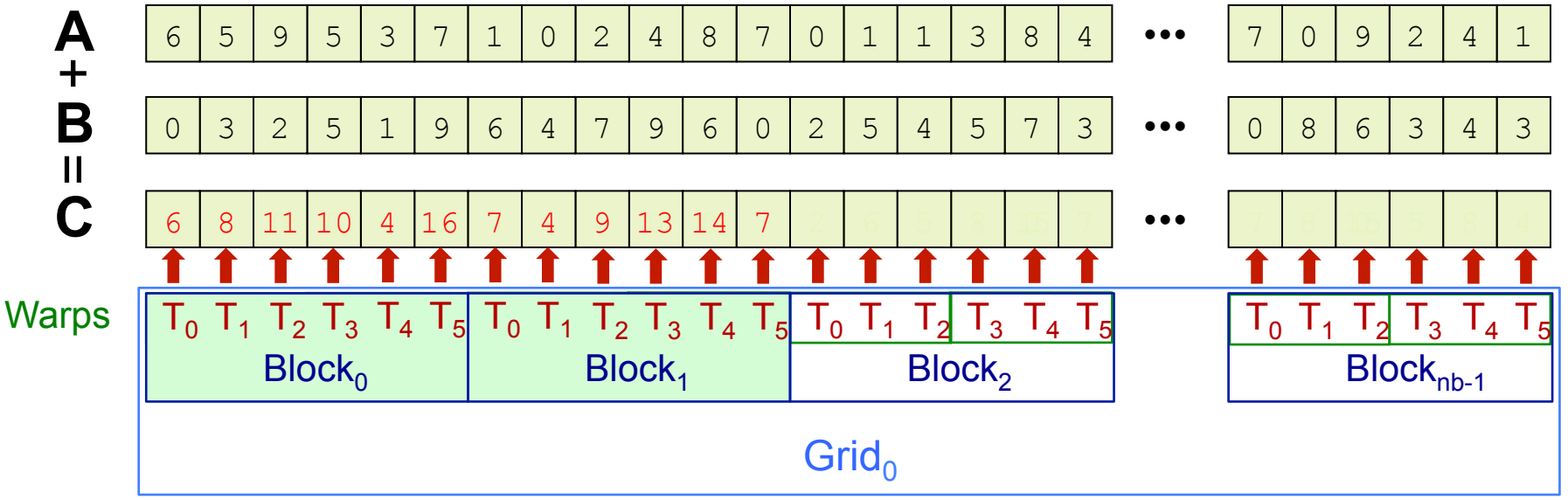
Execution at runtime



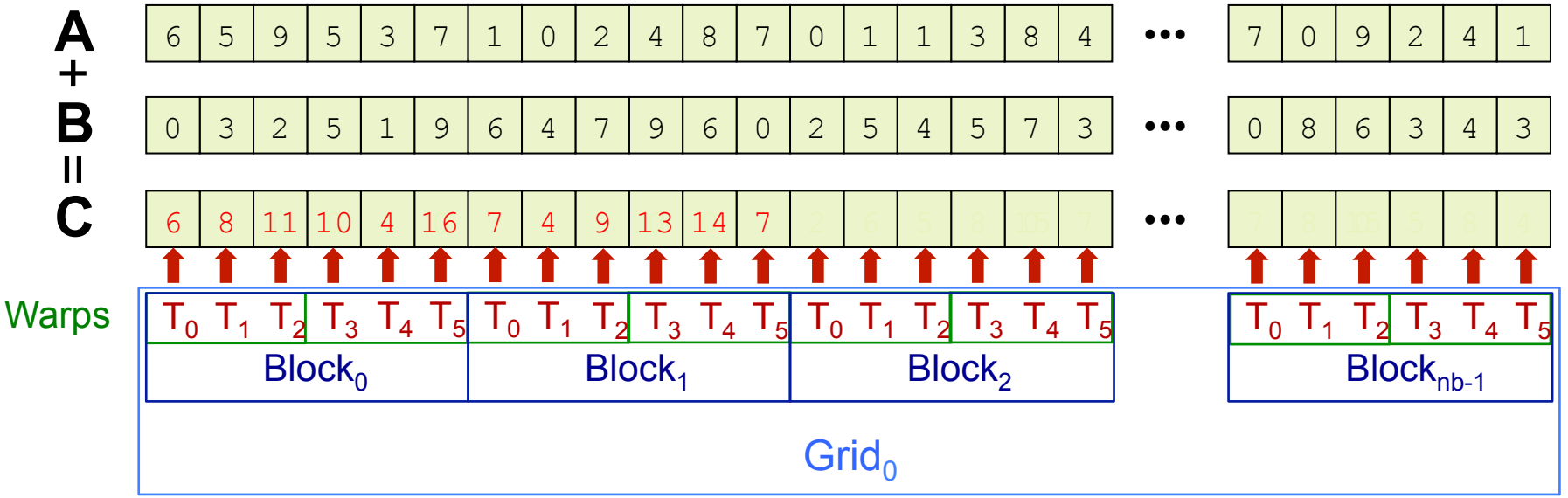
Execution at runtime



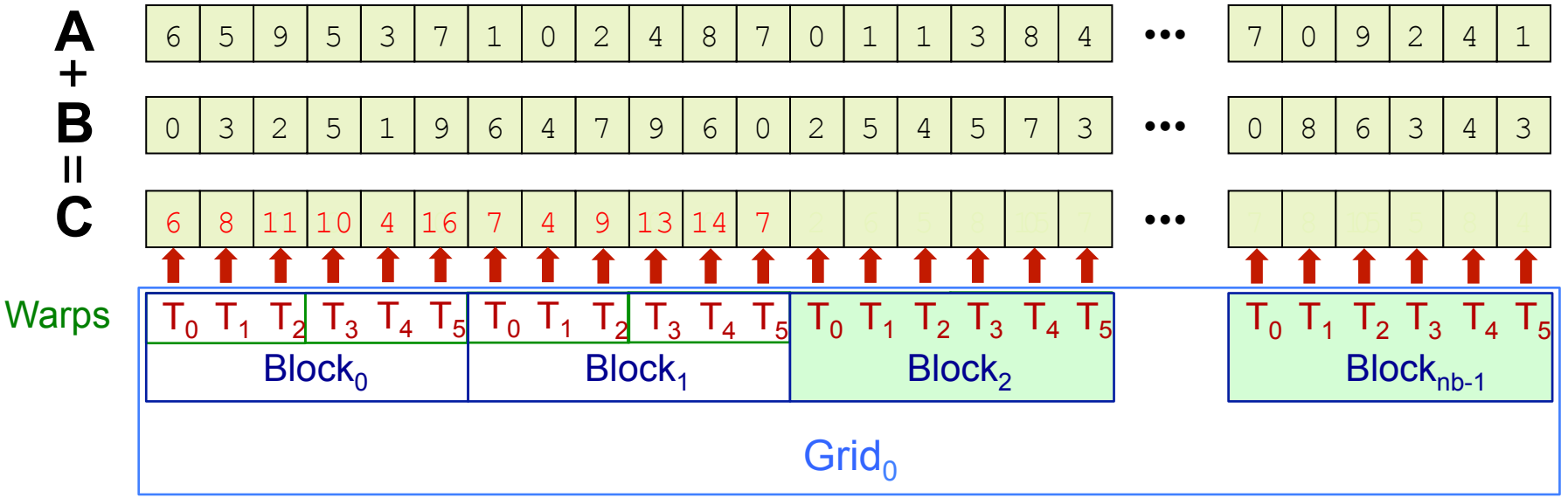
Execution at runtime



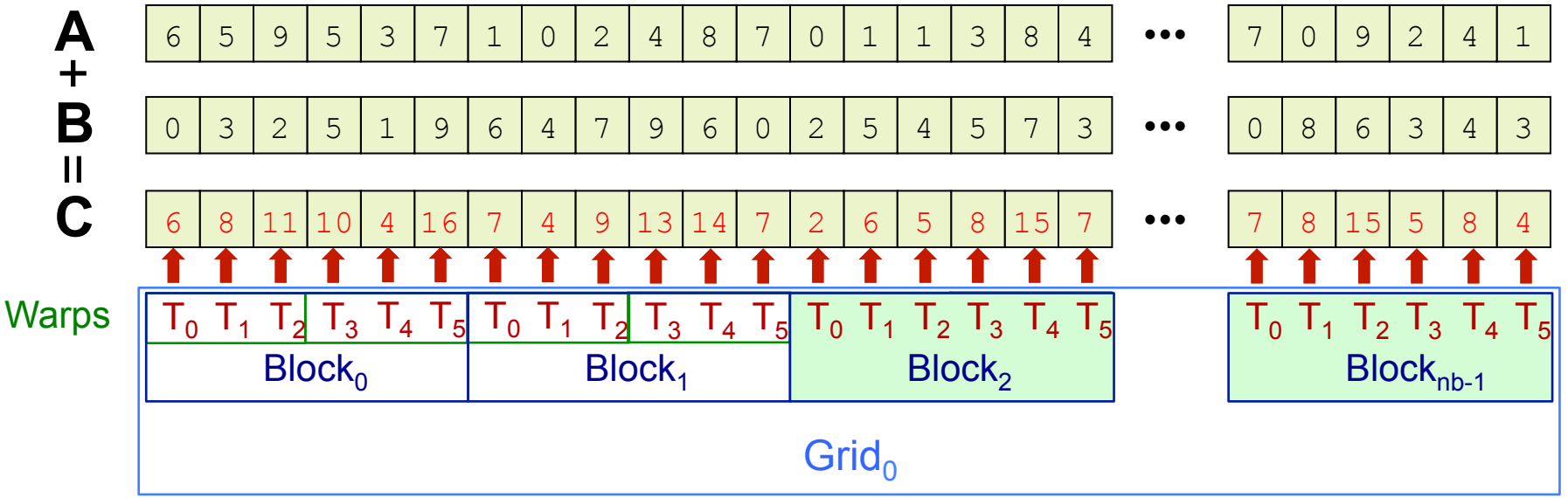
Execution at runtime



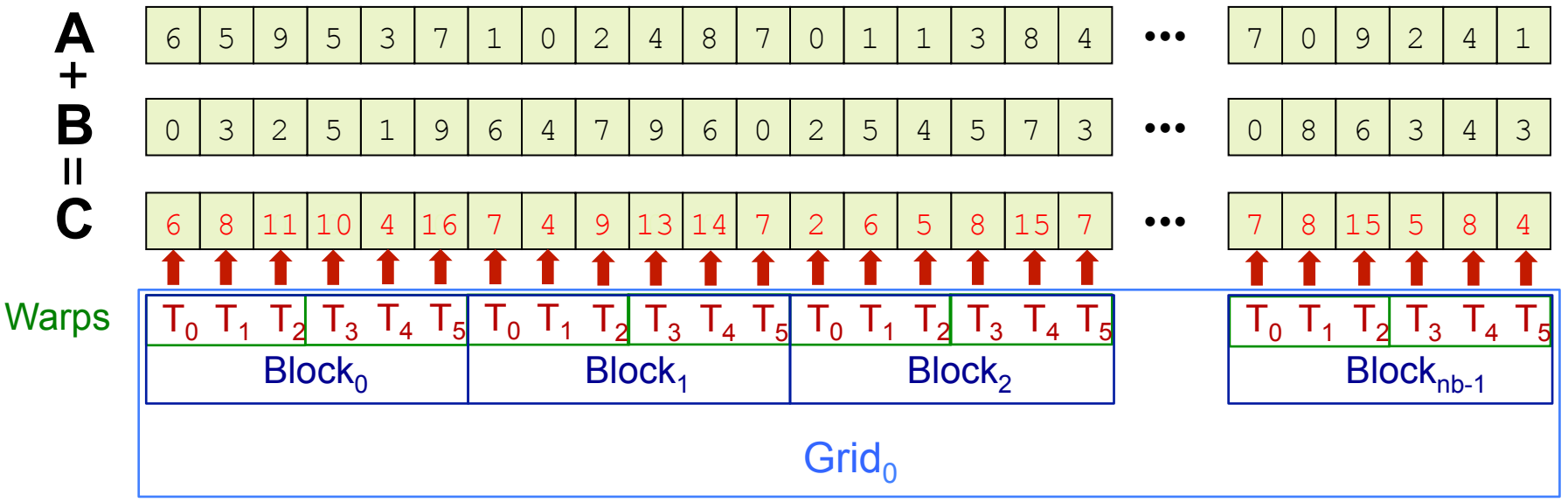
Execution at runtime

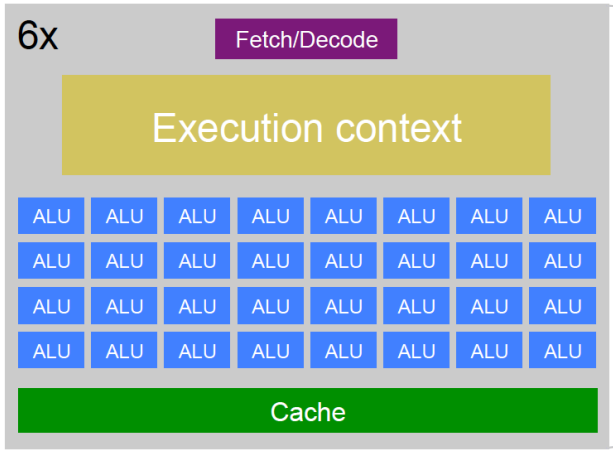


Execution at runtime

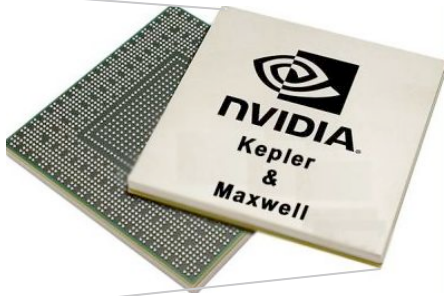


Execution at runtime

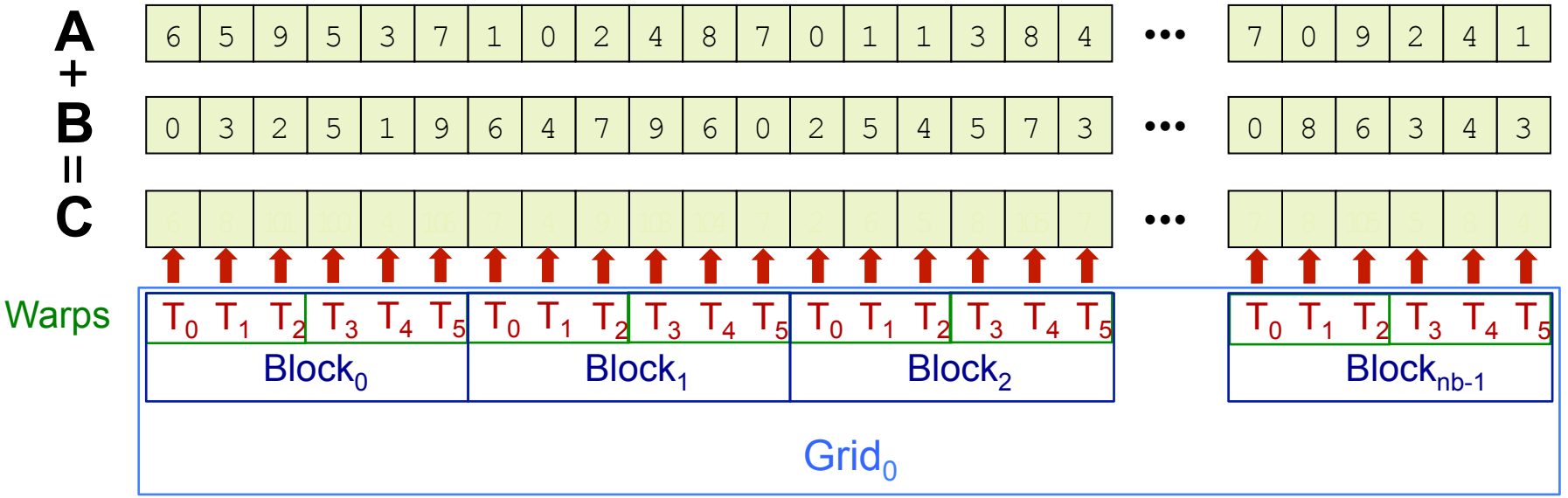


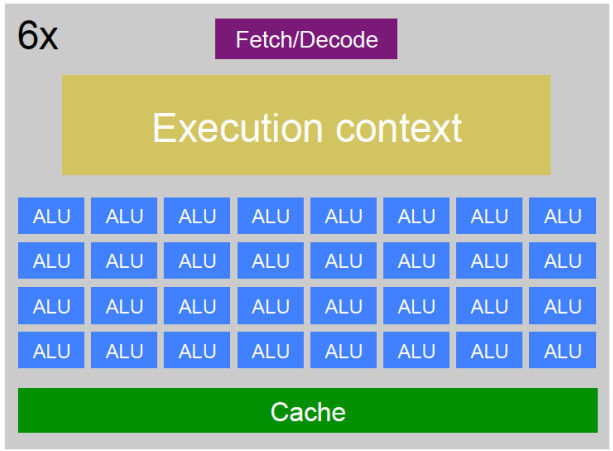


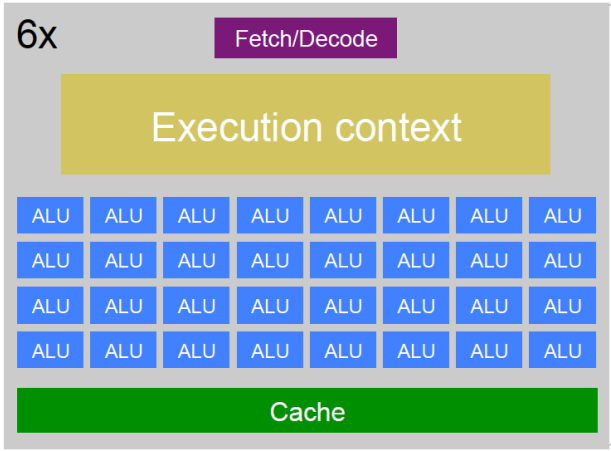
Blocks do not execute in any particular order – do not assume so in your code!

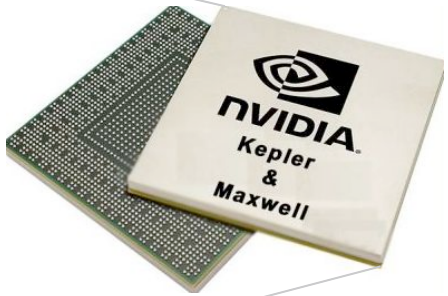


Execution at runtime

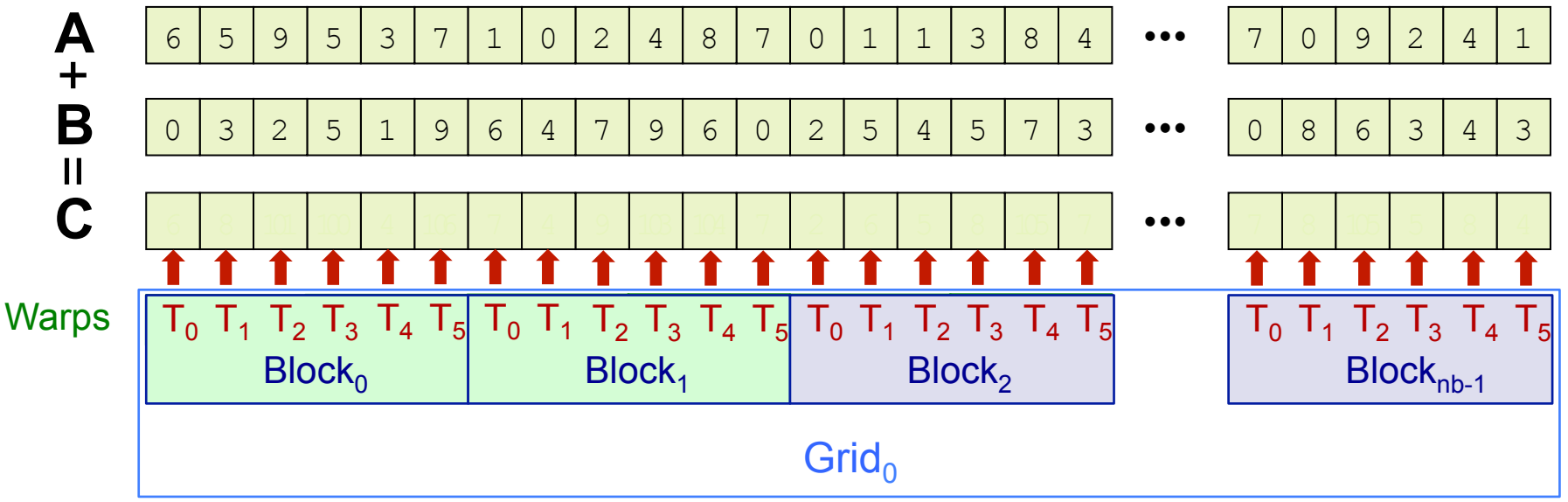


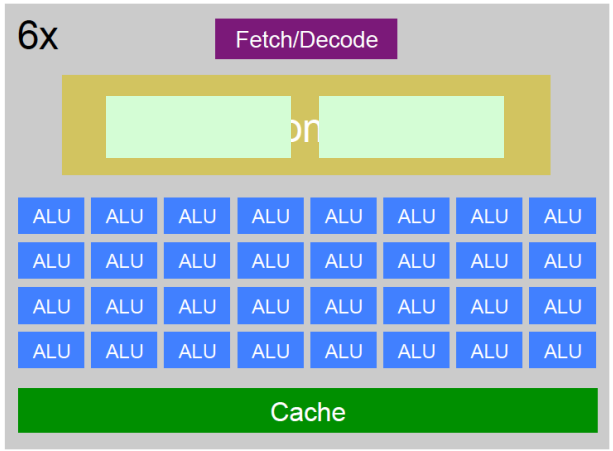


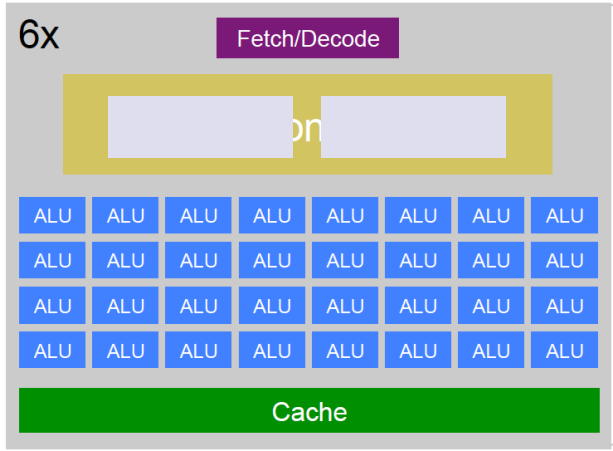


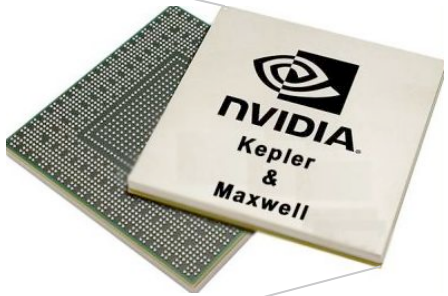


Execution at runtime

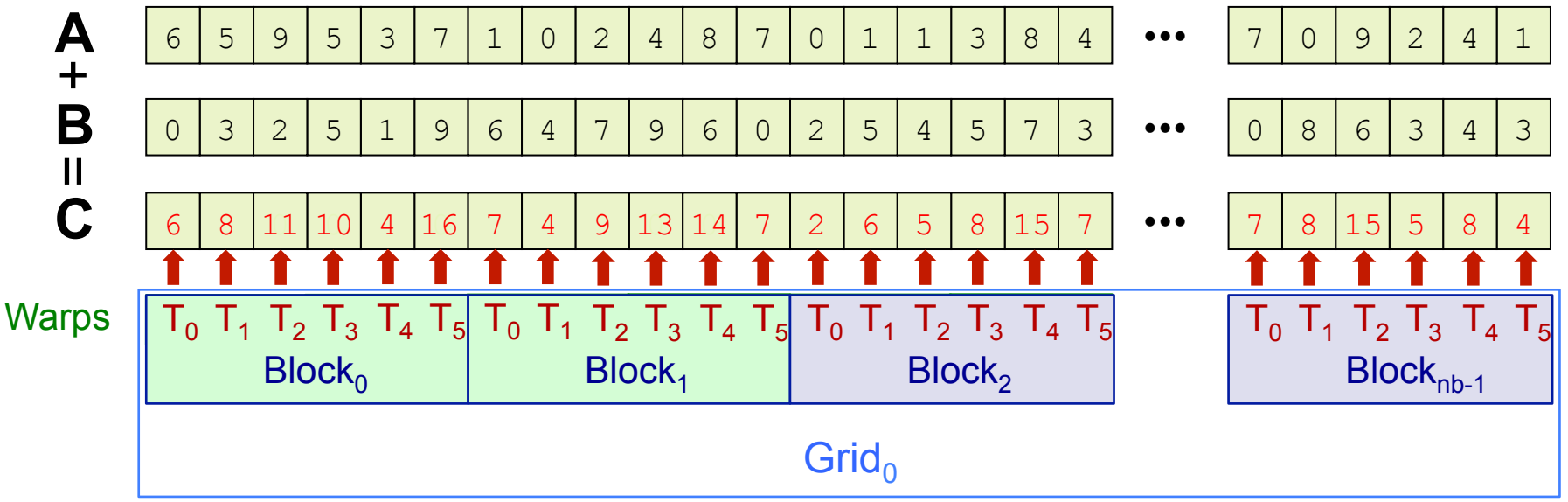


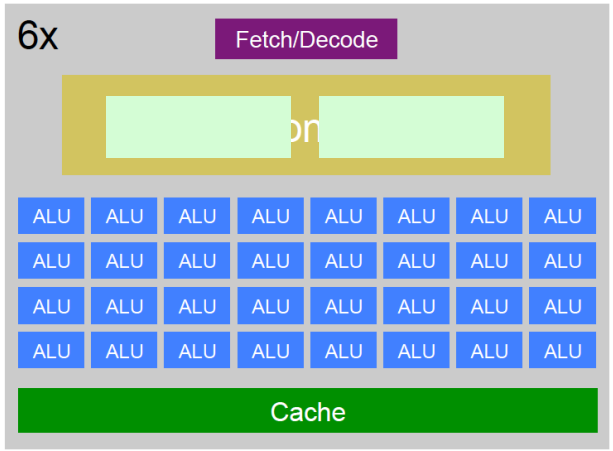


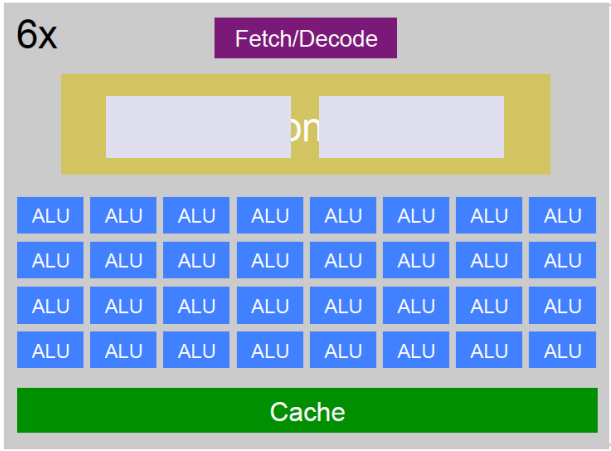


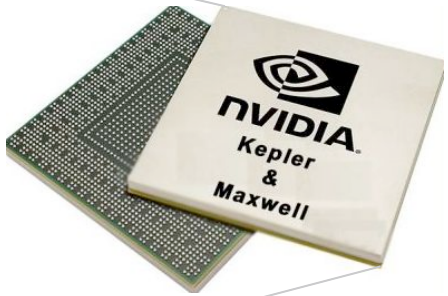


Execution at runtime

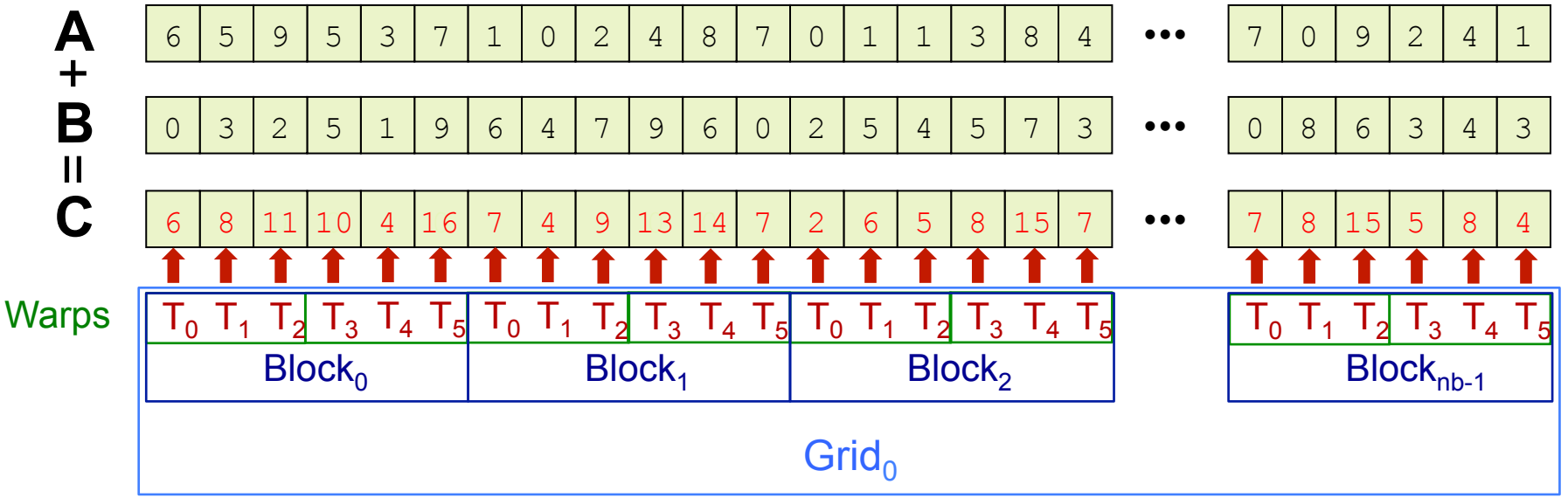


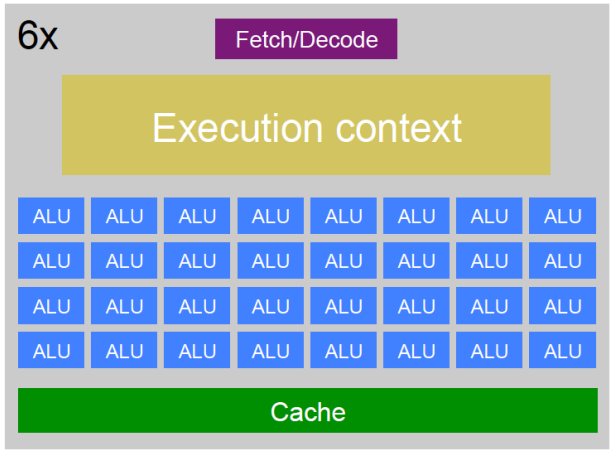


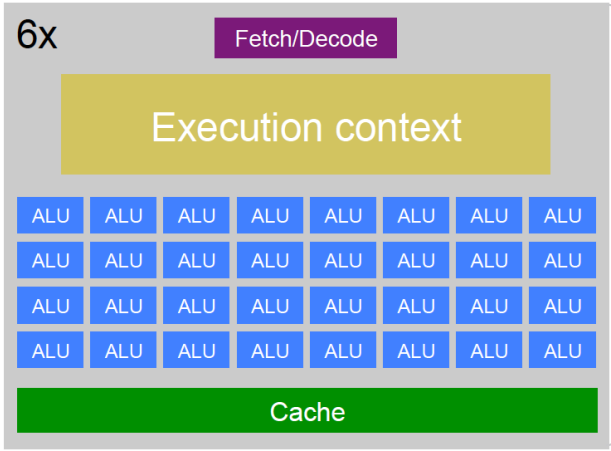


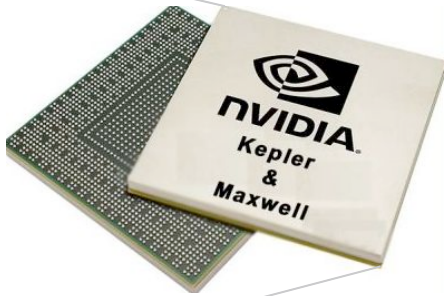


Execution at runtime





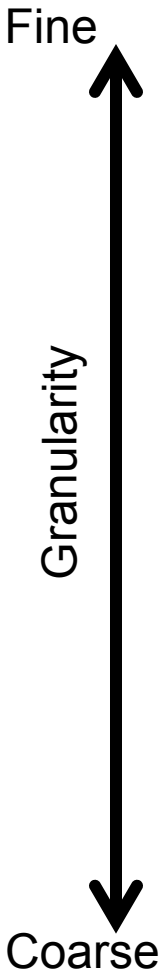




CUDA scheduling of blocks

- Advantages
  - Hardware can run things efficiently
  - No queue from slow blocks
  - Scalability
    - Applies from mobile phones to supercomputers
    - Now and in the future!
- Consequences
  - No assumptions on what blocks will run where
  - No explicit communication between blocks
  - All threads and blocks must complete

Summary of CUDA terminology



- Thread ("Unit of parallelism" in CUDA)
  - Concurrent code and associated state executed on the CUDA device in parallel with other threads. Independent control flow.
- Warp ("Unit of execution")
  - A group of threads in same block that are executed physically in parallel – currently 32 threads.
- Block ("Unit of resource assignment")
  - A virtual group of threads executed together that can cooperate and share data.
- Grid ("Task unit")
  - A virtual group of thread blocks that must all finish before the invoked kernel is completed.

Launching kernels

Launching kernels

- Kernel code defined once – same for all threads:

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

Launching kernels

- Kernel code defined once – same for all threads:

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

- Kernel invocation from host:

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

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

Launching kernels

- Kernel code defined once – same for all threads:

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

- Kernel invocation from host:

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

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

- E.g., for the vector addition example

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

Launching kernels

- Kernel code defined once – same for all threads:

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

- Kernel invocation from host:

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

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

- E.g., for the vector addition example

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

Launching kernels

- Kernel code defined once – same for all threads:

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

- Kernel invocation from host:

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

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

- E.g., for the vector addition example

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

Launching kernels

- Kernel code defined once – same for all threads:

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

- Kernel invocation from host:

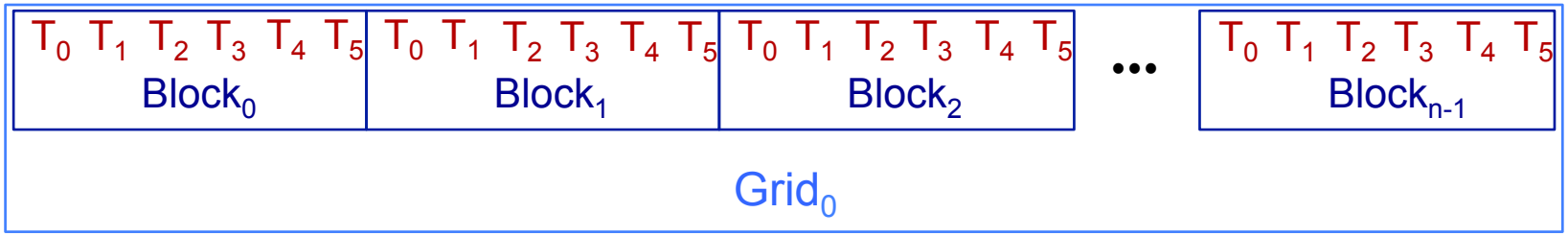
```
kernelFunc<<<NUM_BLOCKS, THREADS_PER_BLOCK>>>(...);
cudaDeviceSynchronize();
```

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

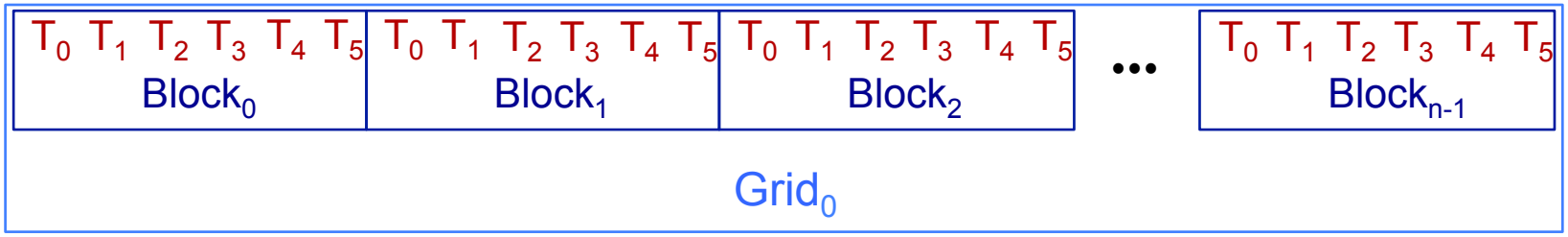
- E.g., for the vector addition example

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

Which thread am I?

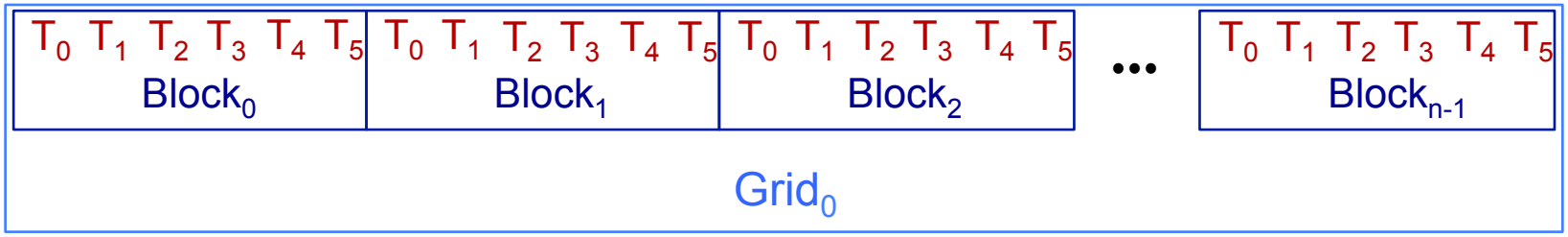


Which thread am I?



- 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 kernelFunc(...) {...};
```

Launching kernels for 2D/3D

- Kernel code defined once – same for all threads:

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

- Kernel execution 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 kernelFunc(...) {...};
```

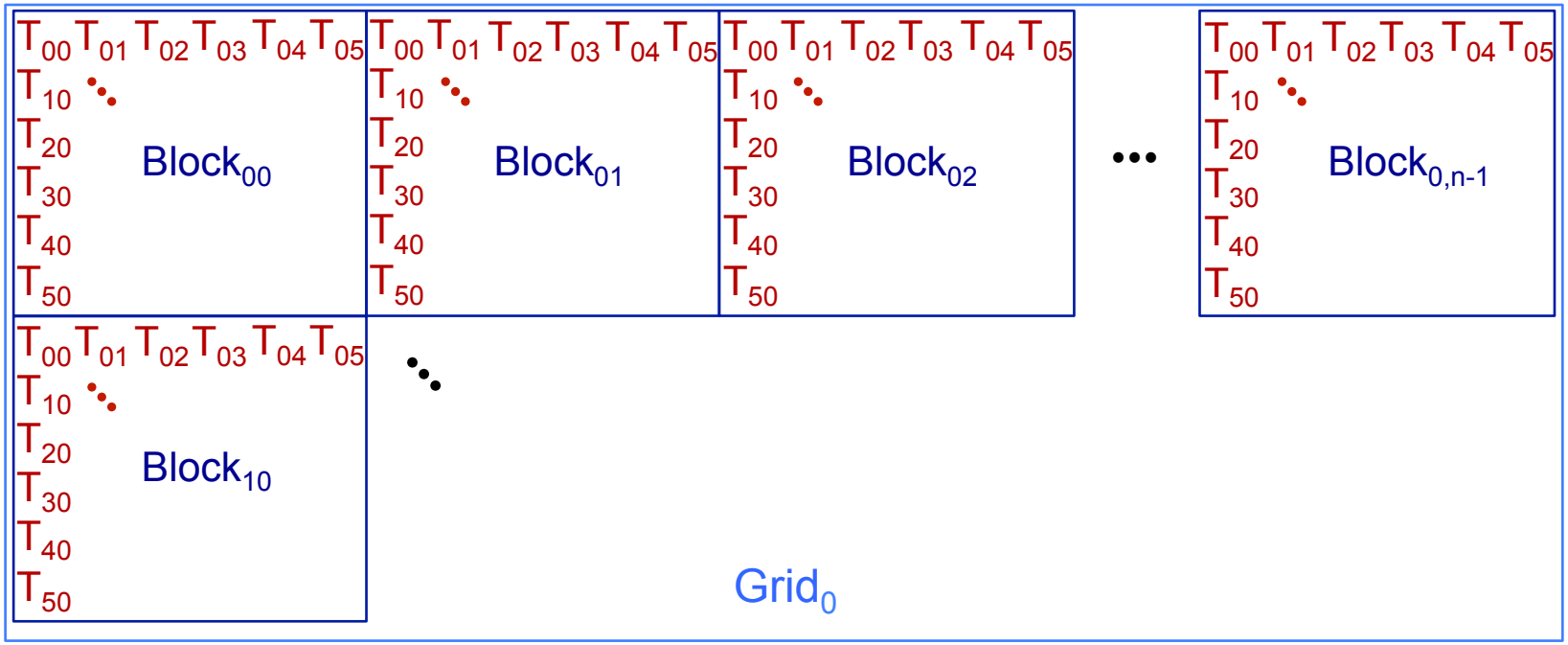
- Kernel execution configuration specified at host:

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

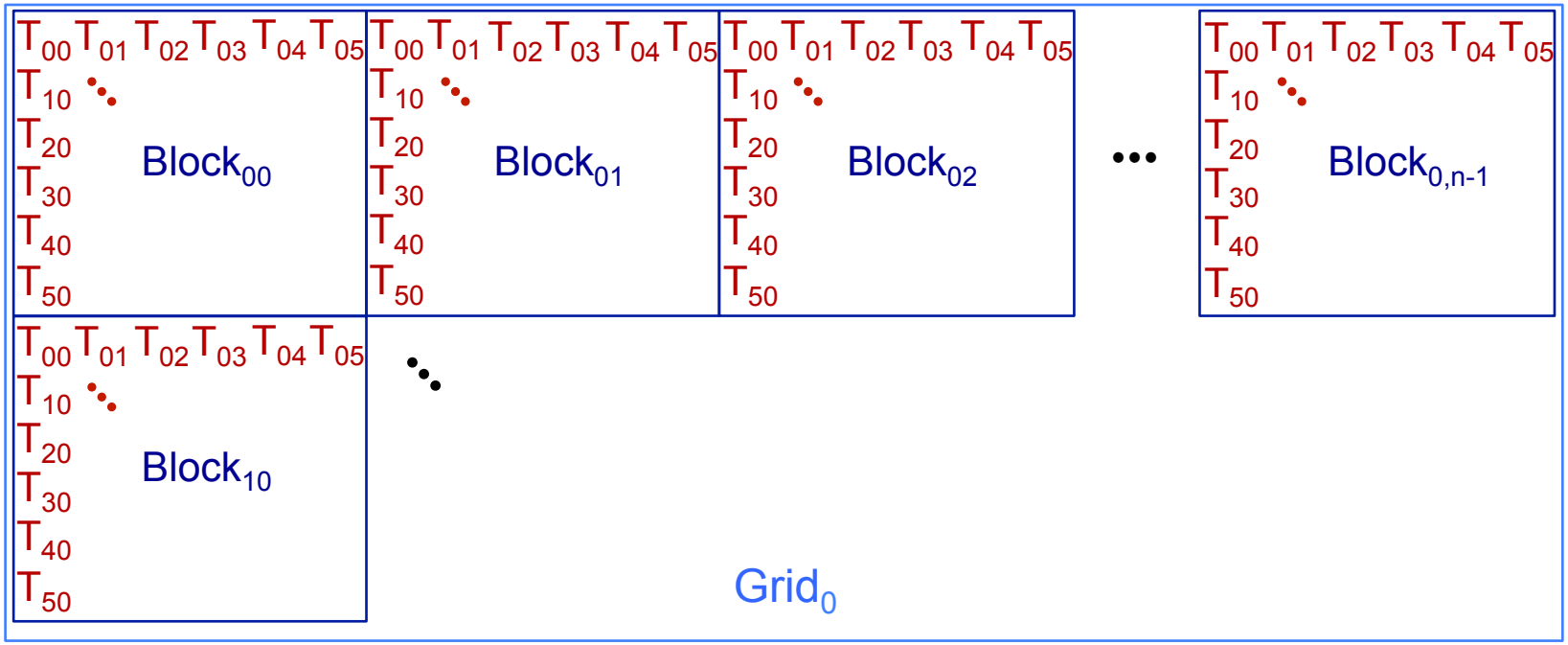
- Kernel invocation from host:

```
KernelFunc<<<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) 5.x (Maxwell)
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(...);
}
```

- Do the exercises 1+2 (deviceQuery and Hello World) now!
  - Please note that `nvcc` with cuda 8.0 requires `gcc` version 5.3 or older; e.g. `module load gcc/5.3`
  - A template Makefile is available on CampusNet
- Get a CUDA Reference Card from the desk!
- Next presentation “CUDA Memory Model” at 13.00 (Monday)!

End of lecture