

CUDA Programming Model



Hans Henrik Brandenborg Sørensen

DTU Computing Center

DTU Compute

[<hhbs@dtu.dk>](mailto:hhbs@dtu.dk)

A vibrant collage of mathematical symbols and numbers. It features a large purple integral symbol with a yellow boundary, a red summation symbol with a black boundary, a blue infinity symbol, a green theta symbol, a pink delta symbol, and several orange arrows pointing right. Interspersed among these are the numbers 17, 2.7182818284, and 14. The background is white with faint, semi-transparent mathematical drawings.



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?

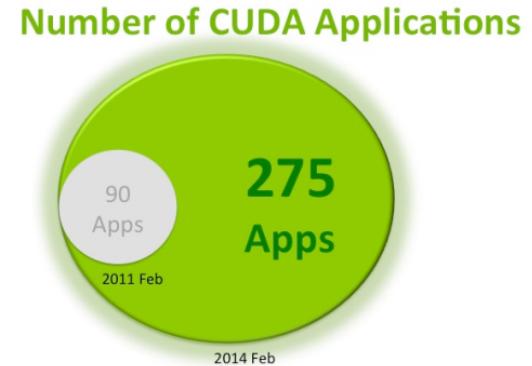


Compute Unified Device Architecture

- 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

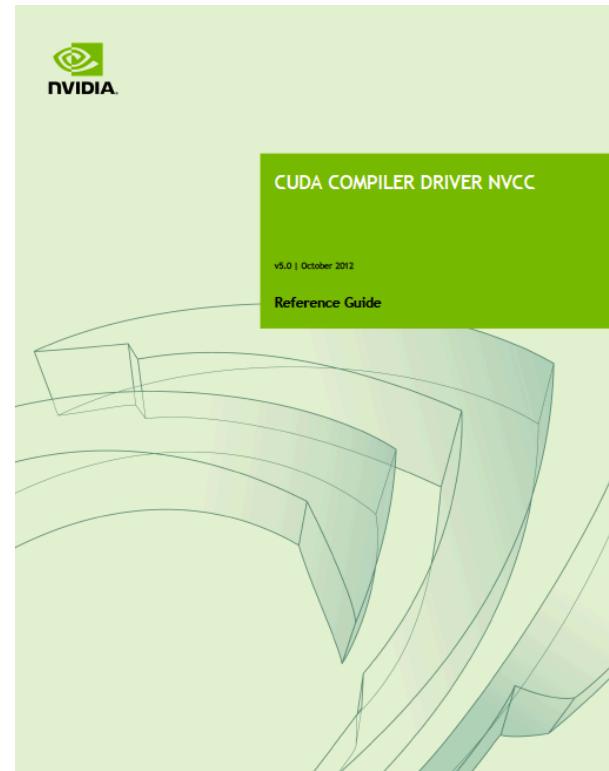
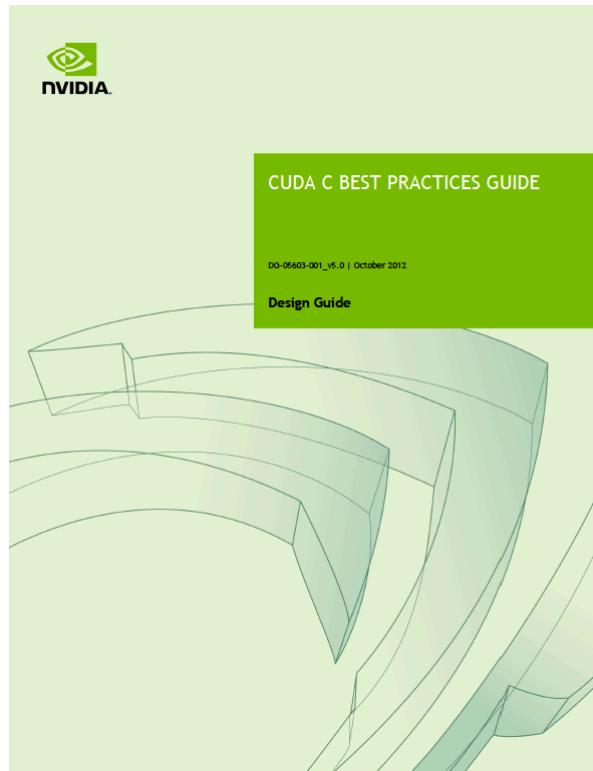
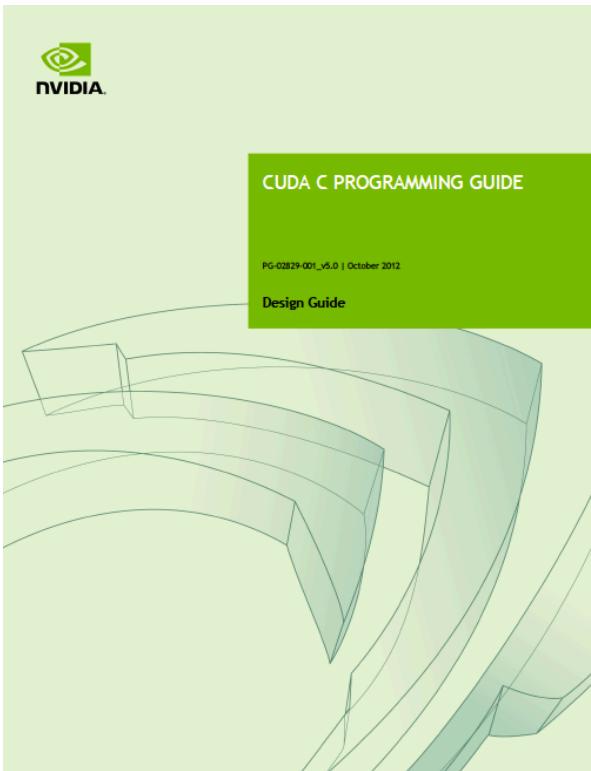
CUDA by the numbers

- 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!**



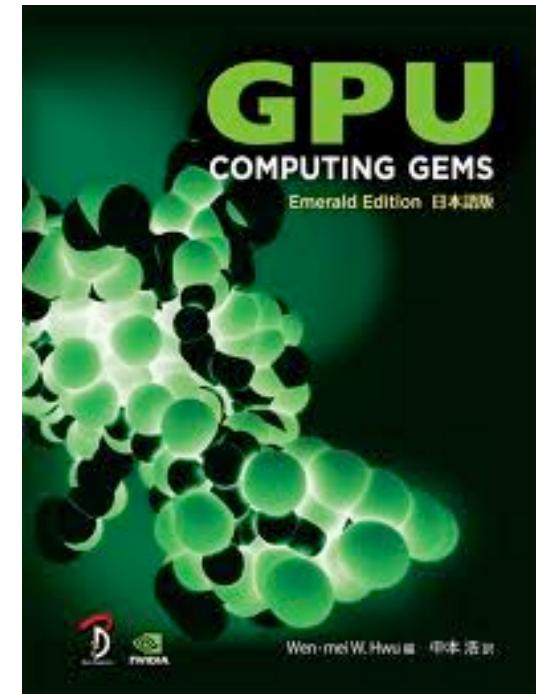
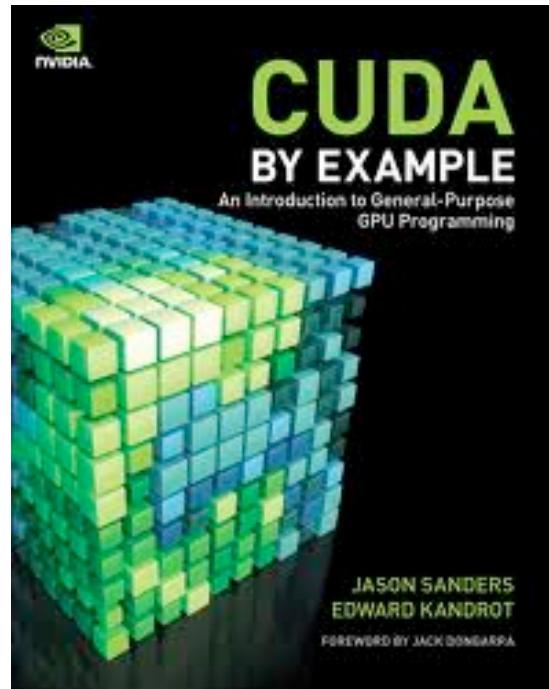
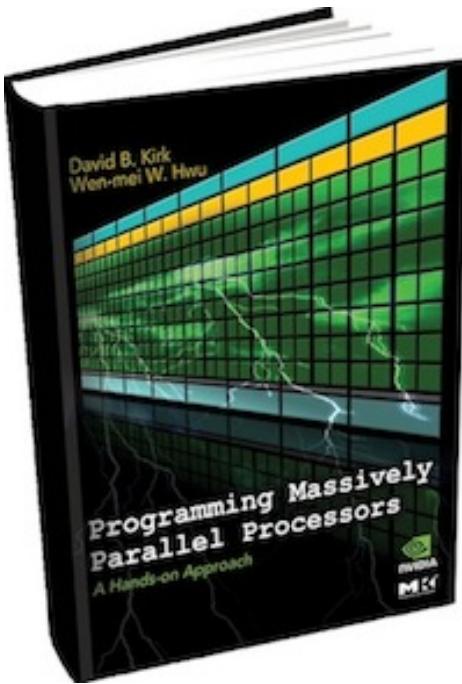
Source: <http://blogs.nvidia.com/blog/2014/02/25/cuda-by-numbers>, Nvidia 2014

Free CUDA material



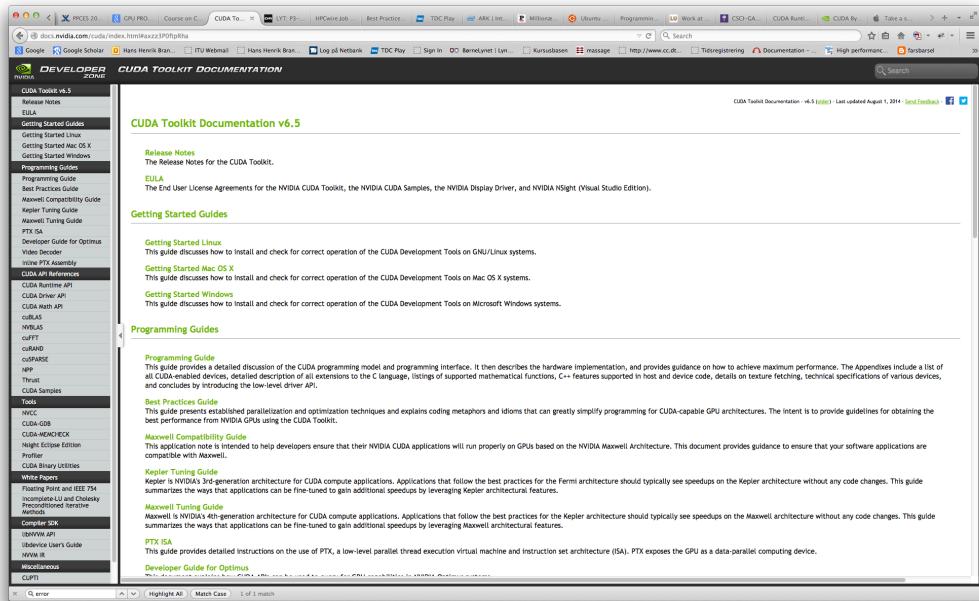
- Free online from Nvidia developer webpage
 - <http://docs.nvidia.com/cuda/index.html>
 - Pdf versions, see installation; /appl/cuda/8.0/doc/pdf/

Additional CUDA material

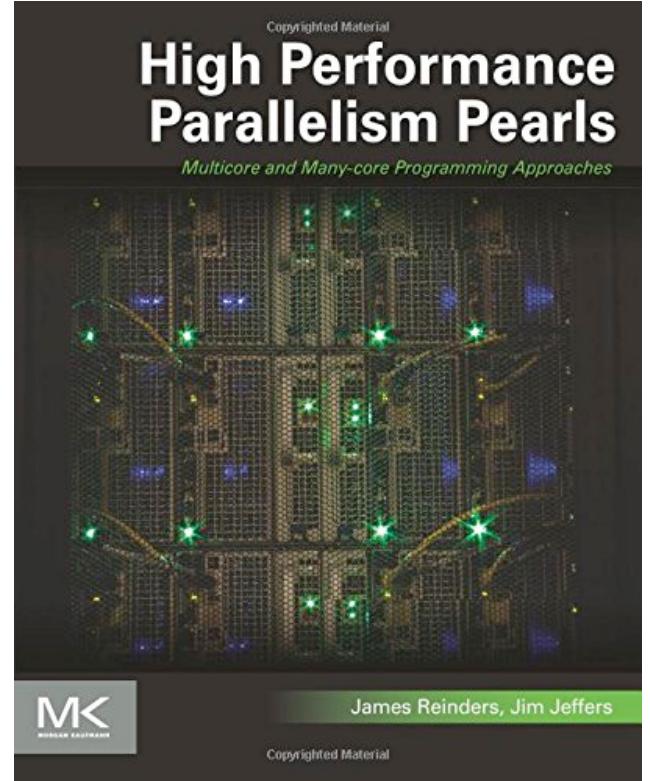


- These are currently the most widely used books
- Feel free to come by my office building 324, room 280 to take a peek in these references

Our suggestion (if you get hooked)



+



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

What is OpenCL?

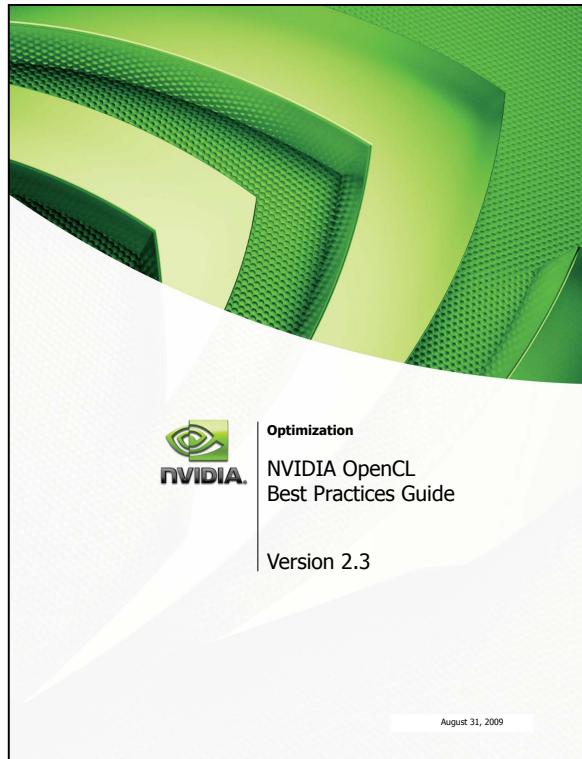
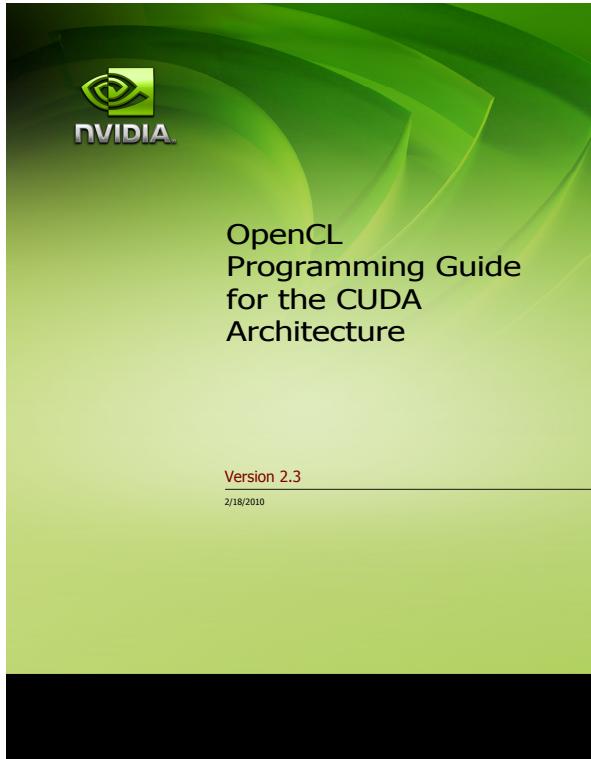


Open Computing Language (current v2.2)

- 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!



Free OpenCl material

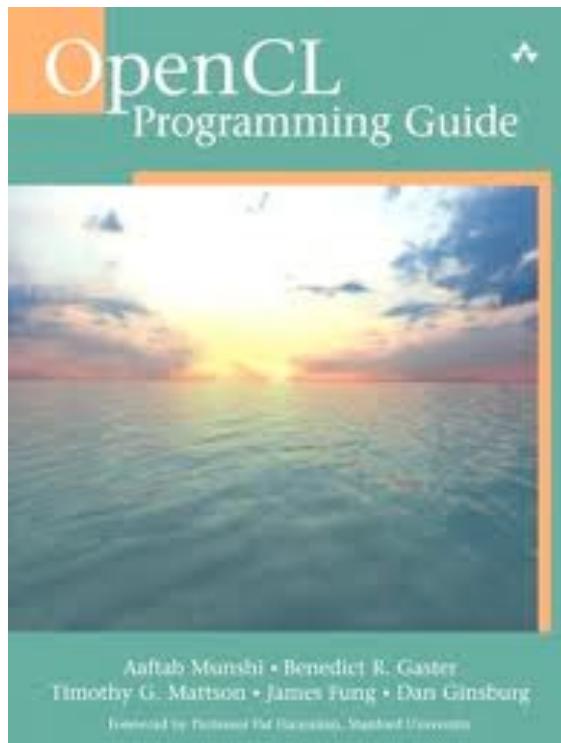
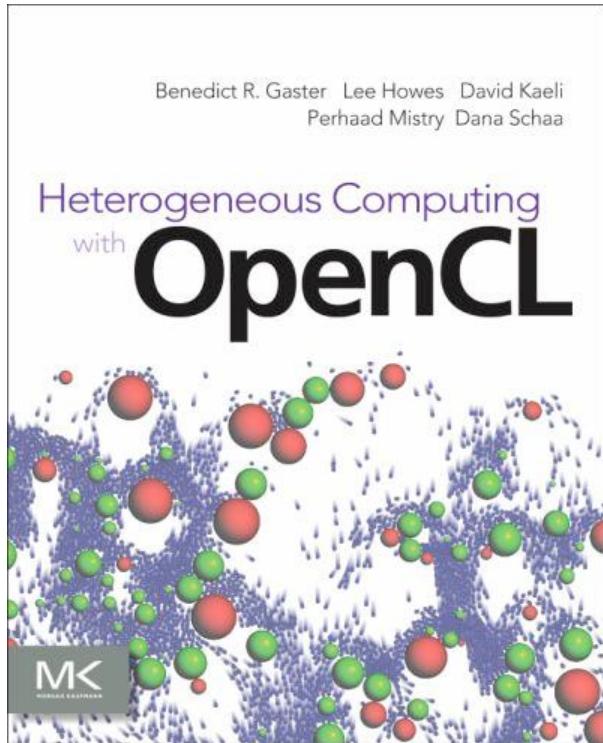


OpenCL API 1.1 Quick Reference Card - Page 1	
<p>OpenCL [1.0] (Open Computing Language)</p> <p>A multi-vendor open standard for parallel programming of heterogeneous systems consisting of CPUs, GPUs and other processors.</p> <p>OpenCL provides a uniform programming environment for writing efficient, portable code for high-performance compute servers, desktop computer systems and mobile devices.</p> <p>[n.n.n] refers to the section in the API Specification available at www.khronos.org/opencl.</p>	
<p>The OpenCL Runtime</p> <p>Command Queues [1.1]</p> <ul style="list-style-type: none"> d_command queue clCreateCommandQueue [d_context context, d_device devic, d_command_queue_properties queue_props, const void *user_data, size_t user_size, int *errcode_ret]; d_int clReleaseCommandQueue [queue]; d_int clEnqueueCommandQueue [d_command cmd, queue, command, queue, const void *buf, size_t buf_size, const void *user_data, size_t user_size, int *errcode_ret]; d_int clEnqueueCommand [queue, command, queue, const void *buf, size_t buf_size, const void *user_data, size_t user_size, int *errcode_ret]; d_int clEnqueueCommandWithWaitList [queue, command, queue, const void *buf, size_t buf_size, const void *user_data, size_t user_size, const void *wait_list, size_t wait_list_size, int *errcode_ret]; d_int clEnqueueCommandWithWaitAll [queue, command, queue, const void *buf, size_t buf_size, const void *user_data, size_t user_size, const void *wait_all, size_t wait_all_size, int *errcode_ret]; 	<p>The OpenCL Platform Layer</p> <p>The OpenCL Platform layer provides specific features that allow applications to query OpenCL devices, device configuration information and device contexts using one or more devices.</p> <p>Contexts [4.1.2]</p> <ul style="list-style-type: none"> d_int clCreateContext [const properties *properties, cl_uint num_devices, const device *devices, void *ctx, CL_CALLBACK *notify, const void *user_data, size_t user_size, int *errcode_ret]; d_int clCreateContextWithProperties [const properties *properties, cl_uint num_devices, const device *devices, void *ctx, CL_CALLBACK *notify, const void *user_data, size_t user_size, int *errcode_ret]; d_int clCreateContextFromType [const properties *properties, cl_device_type type, void *ctx, CL_CALLBACK *notify, const void *user_data, size_t user_size, int *errcode_ret]; d_int clCreateContextFromName [const properties *properties, const char *name, void *ctx, CL_CALLBACK *notify, const void *user_data, size_t user_size, int *errcode_ret]; <p>Properties: See clCreateContext()</p> <ul style="list-style-type: none"> d_int clGetContextInfo [context, info, value, size, value_size, int *errcode_ret]; d_int clSetContextInfo [context, info, value, size, value_size, int *errcode_ret]; <p>clGetContextInfo [cl_int context]</p> <ul style="list-style-type: none"> d_int clGetContextInfo [context, info, value, size, value_size, int *errcode_ret]; <p>Platform Info and Devices [4.1.4]</p> <ul style="list-style-type: none"> d_int clGetPlatformInfo [cl_int num_entries, CL_PLATFORM_INFO_TYPE type, void *platform, size_t *platform_size, int *errcode_ret]; d_int clGetPlatformInfo [platform, platform, cl_platform_info_info, size_t *platform_size, void *platform, size_t *platform_size, int *errcode_ret]; d_int clGetDeviceInfo [cl_platform platform, cl_device device, cl_device_type type, device_info_info, size_t *size, void *platform, size_t *platform_size, int *errcode_ret]; d_int clGetDeviceInfo [cl_platform platform, cl_device device, cl_device_type type, device_info_info, size_t *size, void *platform, size_t *platform_size, int *errcode_ret];
<p>Buffer Objects</p> <p>(Elements of a buffer object can be a scalar or vector data type or a pointer to memory. Buffer objects are stored sequentially in memory and are accessed using a pointer by a kernel executing on a device.)</p> <p>Create Buffer Objects [1.1]</p> <ul style="list-style-type: none"> d_mem mem_create [const void *buf, size_t size, const void *host_addr, int *errcode_ret]; d_int clCreateBuffer [cl_mem buffer, cl_mem_flags flags, cl_ulong host_ptr, const void *create_type, const void *buf, size_t buf_size, int *errcode_ret]; d_int clCreateBufferWithHostAddress [cl_mem buffer, cl_mem_flags flags, cl_ulong host_ptr, const void *buf, size_t buf_size, int *errcode_ret]; d_int clCreateBufferWithProperties [cl_mem buffer, cl_mem_flags flags, cl_ulong host_ptr, const void *buf, size_t buf_size, const void *user_data, size_t user_size, int *errcode_ret]; d_int clCreateBufferWithWaitList [cl_mem buffer, cl_mem_flags flags, cl_ulong host_ptr, const void *buf, size_t buf_size, const void *user_data, size_t user_size, const void *wait_list, size_t wait_list_size, int *errcode_ret]; d_int clCreateBufferWithWaitAll [cl_mem buffer, cl_mem_flags flags, cl_ulong host_ptr, const void *buf, size_t buf_size, const void *user_data, size_t user_size, const void *wait_all, size_t wait_all_size, int *errcode_ret]; 	<p>clEnqueueReadBuffer [1]</p> <ul style="list-style-type: none"> d_int clEnqueueReadBuffer [queue, cl_mem buffer, cl_int num_events, const void *buf, size_t buf_size, const void *user_data, size_t user_size, int *errcode_ret]; d_int clEnqueueReadBuffer [queue, cl_mem buffer, cl_int num_events, const void *buf, size_t buf_size, const void *user_data, size_t user_size, const void *event, const void *event, int *errcode_ret]; d_int clEnqueueReadBuffer [queue, cl_mem buffer, cl_int num_events, const void *buf, size_t buf_size, const void *user_data, size_t user_size, const void *event, const void *event, const void *event, int *errcode_ret]; <p>clEnqueueWriteBuffer [1]</p> <ul style="list-style-type: none"> d_int clEnqueueWriteBuffer [queue, cl_mem buffer, cl_int num_events, const void *buf, size_t buf_size, const void *user_data, size_t user_size, int *errcode_ret]; d_int clEnqueueWriteBuffer [queue, cl_mem buffer, cl_int num_events, const void *buf, size_t buf_size, const void *user_data, size_t user_size, const void *event, const void *event, int *errcode_ret]; d_int clEnqueueWriteBuffer [queue, cl_mem buffer, cl_int num_events, const void *buf, size_t buf_size, const void *user_data, size_t user_size, const void *event, const void *event, const void *event, int *errcode_ret]; <p>clEnqueueCopyBuffer [1]</p> <ul style="list-style-type: none"> d_int clEnqueueCopyBuffer [queue, cl_mem buffer, cl_int num_events, const void *src, cl_mem dst, size_t bytes, const void *user_data, size_t user_size, int *errcode_ret]; d_int clEnqueueCopyBuffer [queue, cl_mem buffer, cl_int num_events, const void *src, cl_mem dst, size_t bytes, const void *user_data, size_t user_size, const void *event, const void *event, int *errcode_ret]; d_int clEnqueueCopyBuffer [queue, cl_mem buffer, cl_int num_events, const void *src, cl_mem dst, size_t bytes, const void *user_data, size_t user_size, const void *event, const void *event, const void *event, int *errcode_ret]; <p>clEnqueueCopyBufferRect [1]</p> <ul style="list-style-type: none"> d_int clEnqueueCopyBufferRect [queue, cl_mem buffer, cl_int num_events, const void *src, cl_mem dst, size_t bytes, const void *user_data, size_t user_size, int *errcode_ret]; d_int clEnqueueCopyBufferRect [queue, cl_mem buffer, cl_int num_events, const void *src, cl_mem dst, size_t bytes, const void *user_data, size_t user_size, const void *event, const void *event, int *errcode_ret]; d_int clEnqueueCopyBufferRect [queue, cl_mem buffer, cl_int num_events, const void *src, cl_mem dst, size_t bytes, const void *user_data, size_t user_size, const void *event, const void *event, const void *event, int *errcode_ret]; <p>clBuildProgram [4.4.1]</p> <ul style="list-style-type: none"> d_int clBuildProgram [cl_program program, cl_int num_devices, const device *devices, const char *options, void *log, CL_CALLBACK *notify, const void *user_data, size_t user_size, int *errcode_ret]; <p>Build Options [5.4.1]</p> <p>Program options must be listed in order listed in clBuildProgram()</p> <ul style="list-style-type: none"> -O0: disable optimizations -O1: -fno-strict-aliasing -O2: -fno-strict-aliasing -fno-common -O3: -fno-strict-aliasing -fno-common -finline-functions -Ofast: -fno-strict-aliasing -fno-common -finline-functions -funsafe-math-only -Ofast-math: -fno-strict-aliasing -fno-common -finline-functions -funsafe-math-only -funsafe-math-optimizations
<p>Program Objects</p> <p>Create Program Objects [5.4.1]</p> <ul style="list-style-type: none"> d_int clCreateProgramWithSource [cl_context context, cl_int num_devices, const device *devices, const char *options, const void *source, size_t source_size, const void *user_data, size_t user_size, int *errcode_ret]; d_int clCreateProgramWithBinary [cl_context context, cl_int num_devices, const device *devices, const void *binary, size_t binary_size, const void *user_data, size_t user_size, int *errcode_ret]; d_int clCreateProgram [cl_program program, const void *source, size_t source_size, const void *user_data, size_t user_size, int *errcode_ret]; d_int clCreateProgramWithSource [cl_program program, const void *source, size_t source_size, const void *user_data, size_t user_size, int *errcode_ret]; 	<p>Map Buffer Object [5.2.2]</p> <ul style="list-style-type: none"> d_int clMapBuffer [cl_mem buffer, cl_int num_events, const void *user_data, size_t user_size, const void *host_ptr, size_t host_size, const void *user_offset, size_t offset, void *map, CL_CALLBACK *notify, const void *user_data, size_t user_size, int *errcode_ret]; <p>Map Buffer Objects [5.4.1]</p> <ul style="list-style-type: none"> d_int clMapMultipleObjects [cl_mem memobj, cl_int num_objects, const void *user_data, size_t user_size, const void *host_ptr, size_t host_size, const void *user_offset, size_t offset, void *maps, CL_CALLBACK *notify, const void *user_data, size_t user_size, int *errcode_ret]; <p>Map Buffer Object With Callback [5.4.1]</p> <ul style="list-style-type: none"> d_int clMapBufferWithCallback [cl_mem memobj, void *host_ptr, size_t host_size, const void *user_offset, size_t offset, void *map, CL_CALLBACK *notify, const void *user_data, size_t user_size, int *errcode_ret]; <p>Unmap Buffer Object [5.2.2]</p> <ul style="list-style-type: none"> d_int clUnmapBuffer [cl_mem buffer, void *map, CL_CALLBACK *notify, const void *user_data, size_t user_size, int *errcode_ret]; <p>Query Buffer Object [5.4.4]</p> <ul style="list-style-type: none"> d_int clGetMemObjectInfo [cl_mem memobj, cl_mem_info info, size_t *size, void *value, CL_CALLBACK *notify, const void *user_data, size_t user_size, int *errcode_ret]; <p>Memory Information</p> <ul style="list-style-type: none"> CL_MEM_TYPE, CL_MEM_FLAGS, CL_MEM_SIZE, CL_MEM_CONTEXT, CL_MEM_ASSOCIATED, CL_MEM_METADATA
<p>Program Properties</p> <p>Create Program Properties [5.4.1]</p> <ul style="list-style-type: none"> d_int clCreateProgramProperties [cl_context context, cl_int num_devices, const device *devices, const char *options, const void *source, size_t source_size, const void *user_data, size_t user_size, int *errcode_ret]; 	<p>Math Functions</p> <ul style="list-style-type: none"> cl_khr_fp64: -fdouble-precision-constant -fdouble-precision-store-constant Warning request/suppress: <ul style="list-style-type: none"> CL_KHR_OPENCL_C: -Wno-sign-compare <p>Control OpenCL C language version</p> <ul style="list-style-type: none"> d_int clSetGLControl [cl_context context, cl_GL_version gl_version, int *errcode_ret]; <p>clGetGLControl [1.1.1]</p> <ul style="list-style-type: none"> d_int clGetGLControl [cl_context context, cl_GL_version *gl_version, int *errcode_ret]; <p>Query Program Objects [5.4.5]</p> <ul style="list-style-type: none"> d_int clGetProgramInfo [cl_program program, cl_int num_entries, CL_PROGRAM_INFO_TYPE type, void *info, size_t *info_size, int *errcode_ret]; <p>Program Cache [5.4.6]</p> <ul style="list-style-type: none"> d_int clSetProgramCache [cl_program program, cl_int num_entries, const void *source, size_t source_size, const void *user_data, size_t user_size, int *errcode_ret]; <p>Program Cache Control [5.4.6]</p> <ul style="list-style-type: none"> d_int clGetProgramCacheCount [cl_program program, int *count, int *errcode_ret]; <p>Program Cache Get [5.4.6]</p> <ul style="list-style-type: none"> d_int clGetProgramCache [cl_program program, cl_int num_entries, const void *source, size_t source_size, const void *user_data, size_t user_size, int *errcode_ret]; <p>Program Cache Set [5.4.6]</p> <ul style="list-style-type: none"> d_int clSetProgramCache [cl_program program, cl_int num_entries, const void *source, size_t source_size, const void *user_data, size_t user_size, int *errcode_ret]; <p>Program Cache Release [5.4.6]</p> <ul style="list-style-type: none"> d_int clReleaseProgramCache [cl_program program, int *errcode_ret];

- Free from Nvidia and AMD developer webpages

- ❑ <https://developer.nvidia.com/opengl>
 - ❑ Pdf versions, see installation; /usr/local/nvidia/doc/opengl

Additional OpenCL material



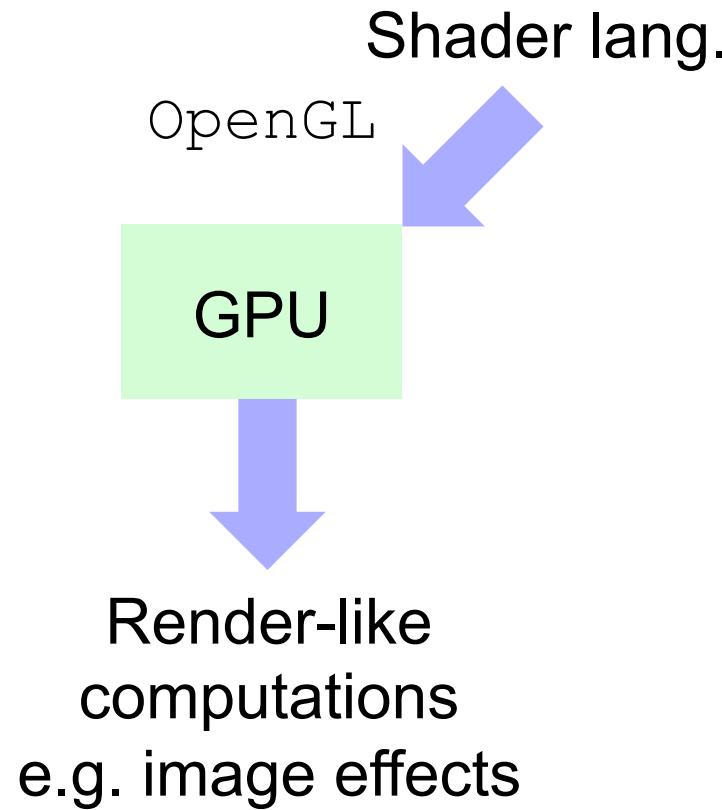
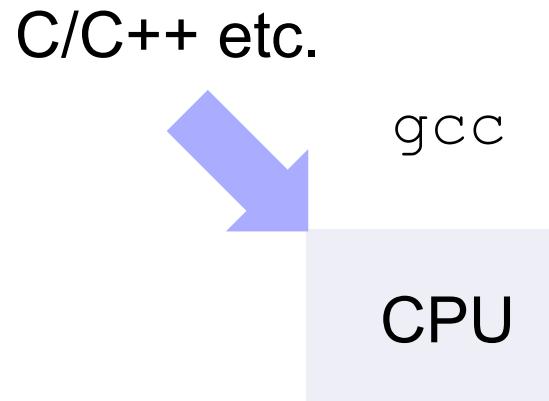
- All published within the last four years
 - Expect more to be published in the future
 - Feel free to come by 324-280 to take a peek

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

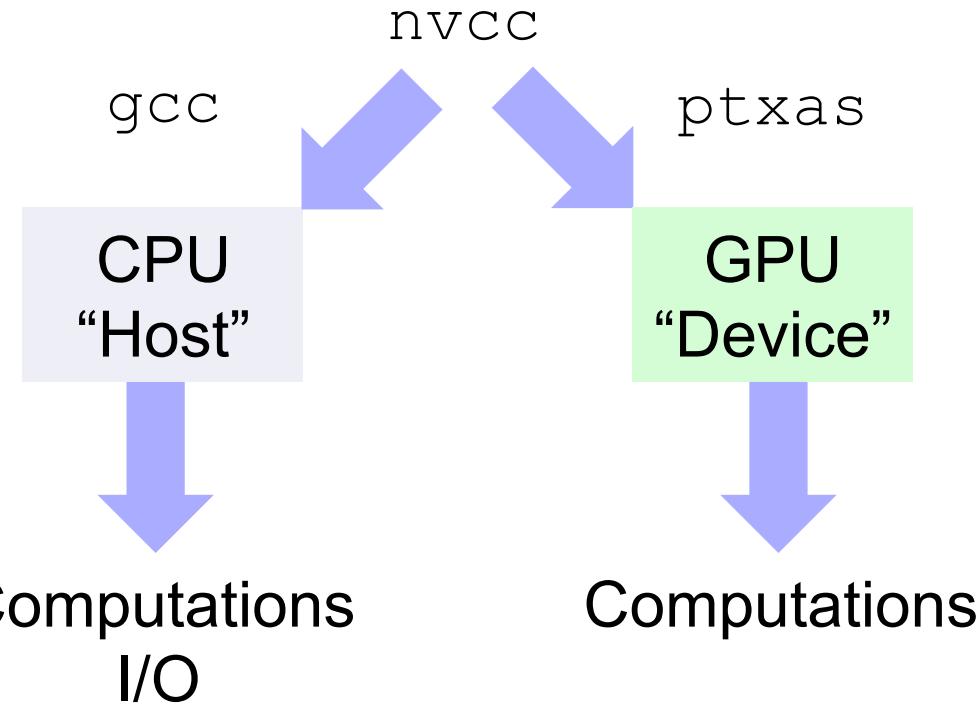
CUDA programming model

Earlier than 2007



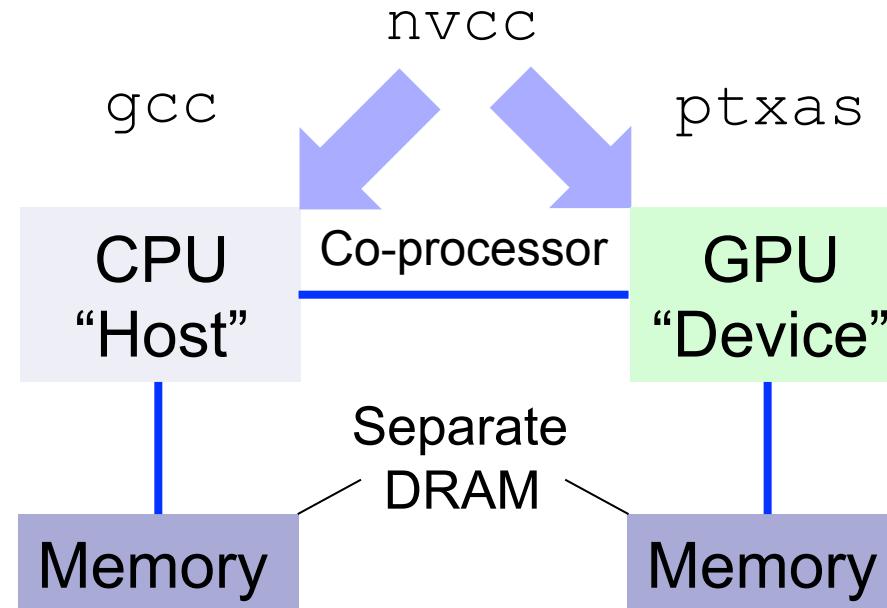
CUDA released

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



CUDA programming model

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



■ Host – the CPU

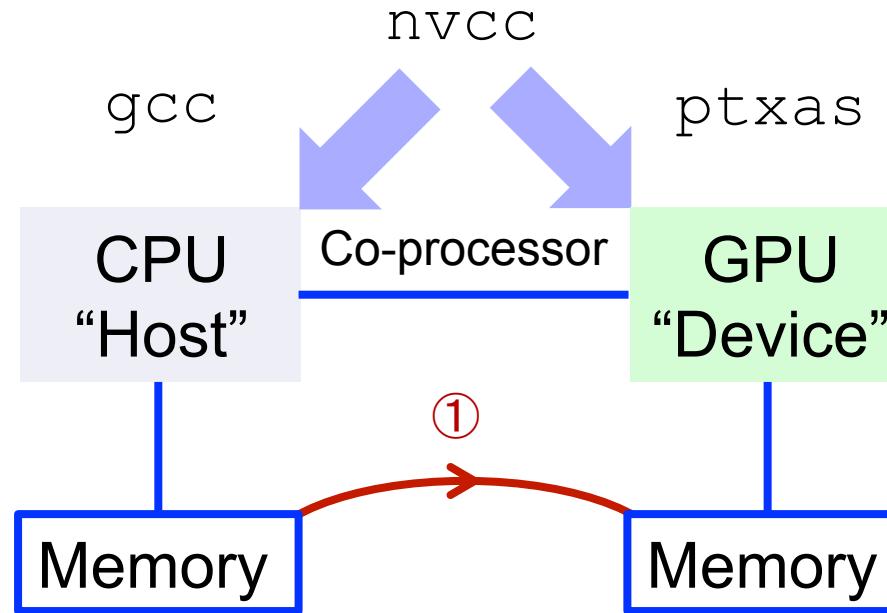
- In charge, manages resources
- Runs main(), etc.

■ Device – the GPU

- Co-processor / accelerator
- Runs specific tasks

CUDA programming model

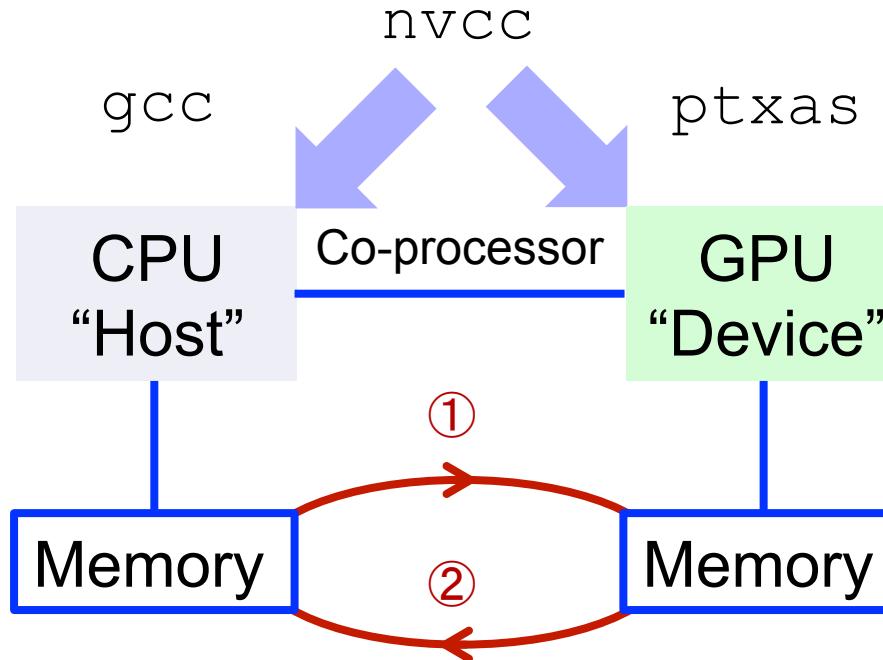
CUDA Program
- written in C/C++ with extensions



- ① Data CPU → GPU

CUDA programming model

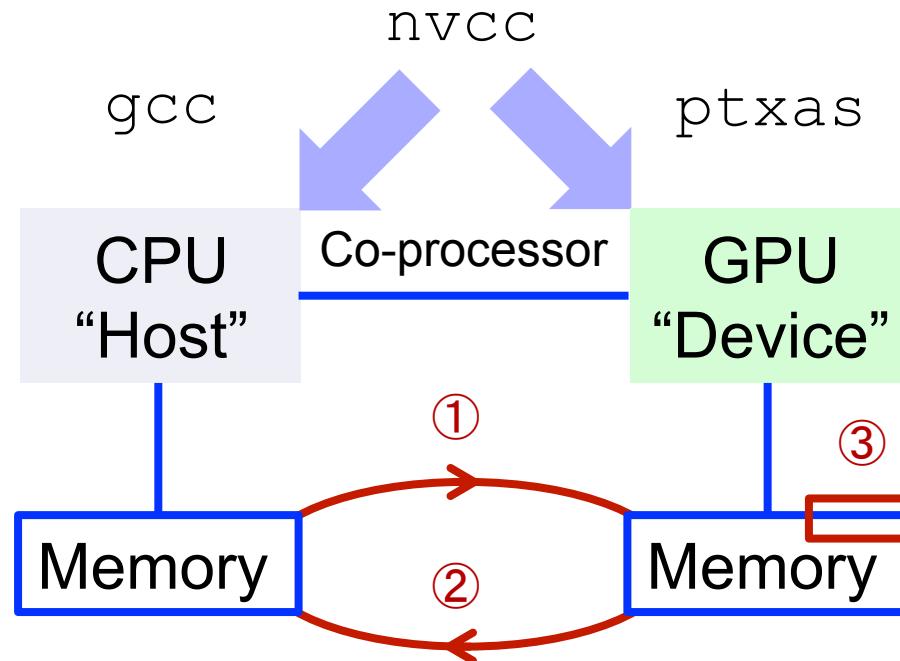
CUDA Program
- written in C/C++ with extensions



- ① Data CPU → GPU
- ② Data GPU → CPU

CUDA programming model

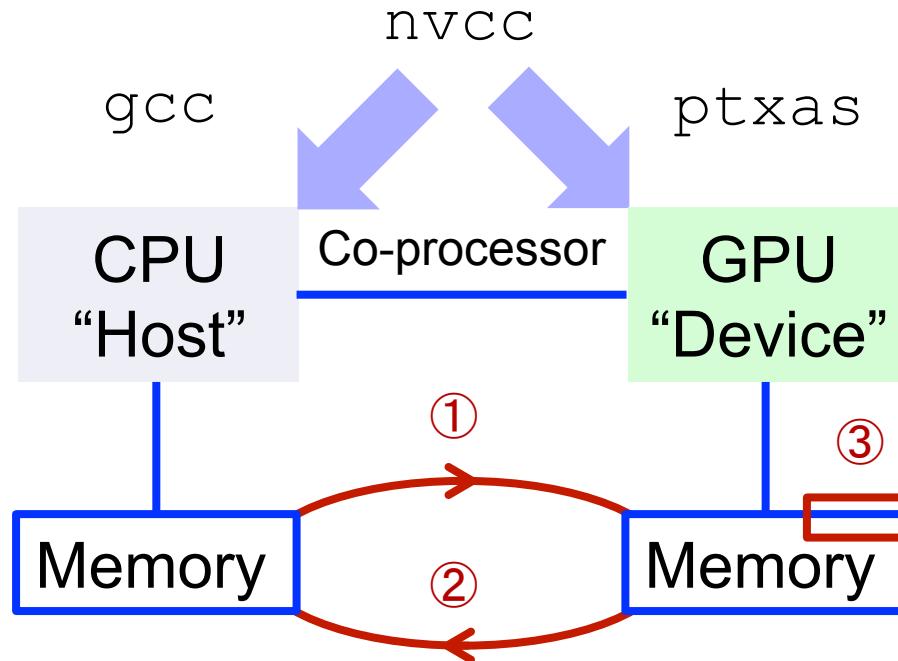
CUDA Program
- written in C/C++ with extensions



- ① Data CPU → GPU
- ② Data GPU → CPU
- ③ Allocate GPU memory

CUDA programming model

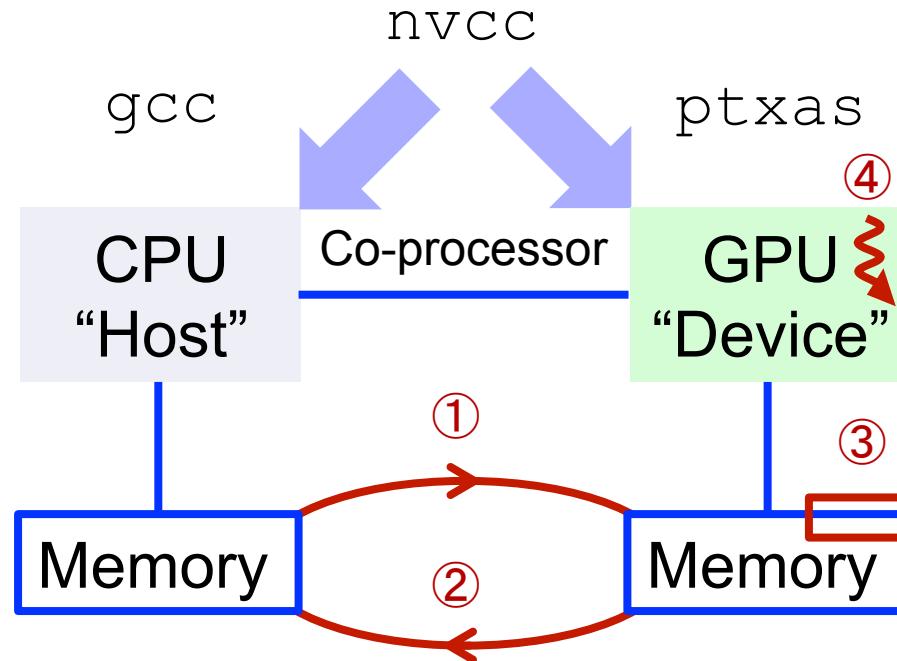
CUDA Program
- written in C/C++ with extensions



- 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:
<code>__global__ void KernelFunc()</code>	device	host
<code>__device__ double DeviceFunc()</code>	device	device
<code>__host__ double HostFunc()</code>	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`

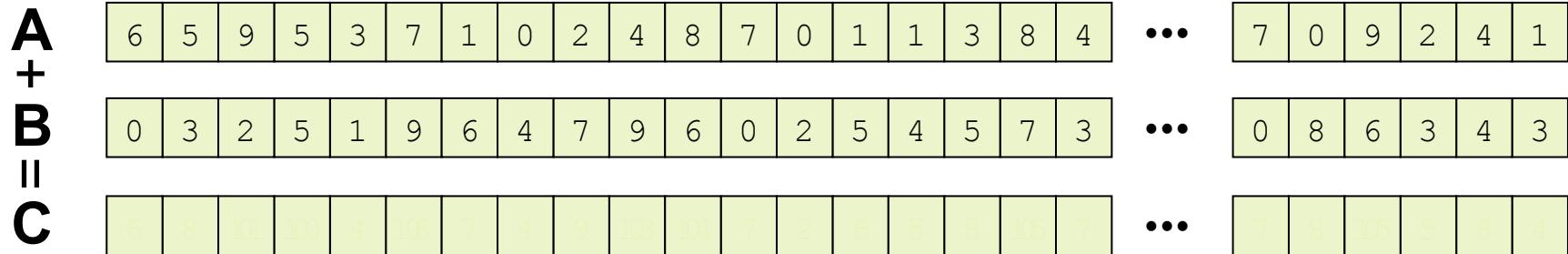
■ On the host, functions use the C runtime implementation if available `<math.h>`

Math functions

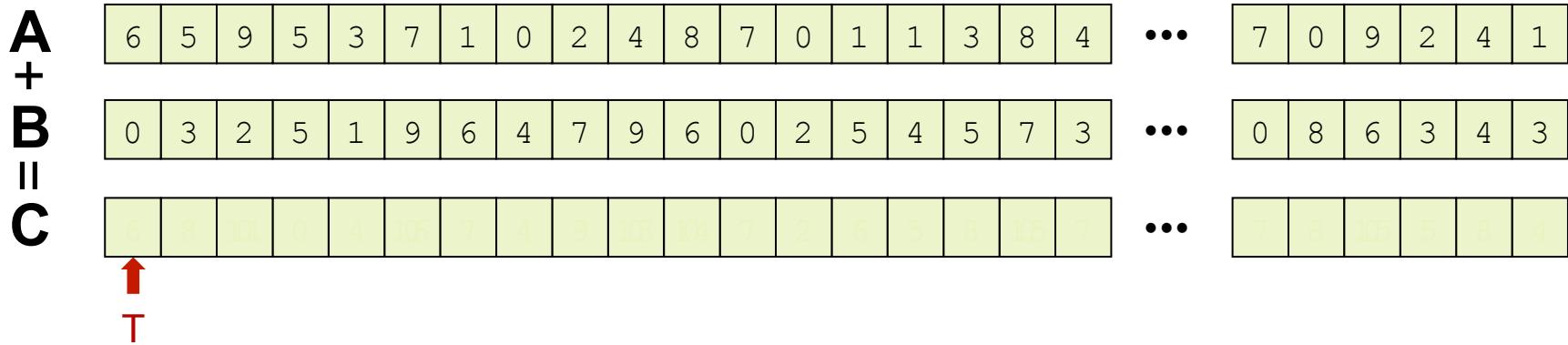
- On the device, also **intrinsic** math functions are available (reminiscent from fast graphics):
 - Device only
 - Faster, but less accurate
 - Prefixed with `_`
 - `_exp`, `_log`, `_sin`, `_pow`, ...
- Use explicitly or force all math to be intrinsic using `-use_fast_math` compiler option

Warps, Blocks and Grids

Simple example



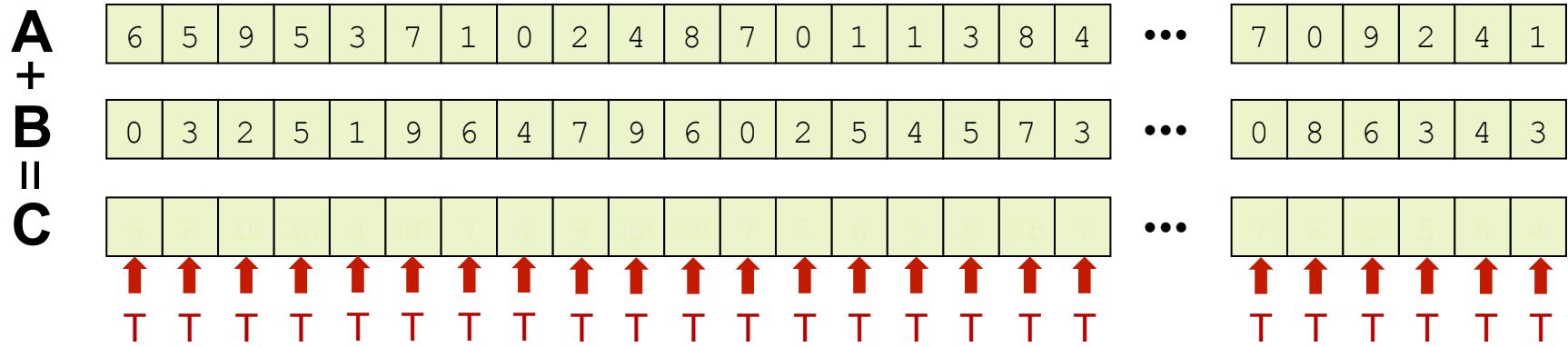
Simple example



```
__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

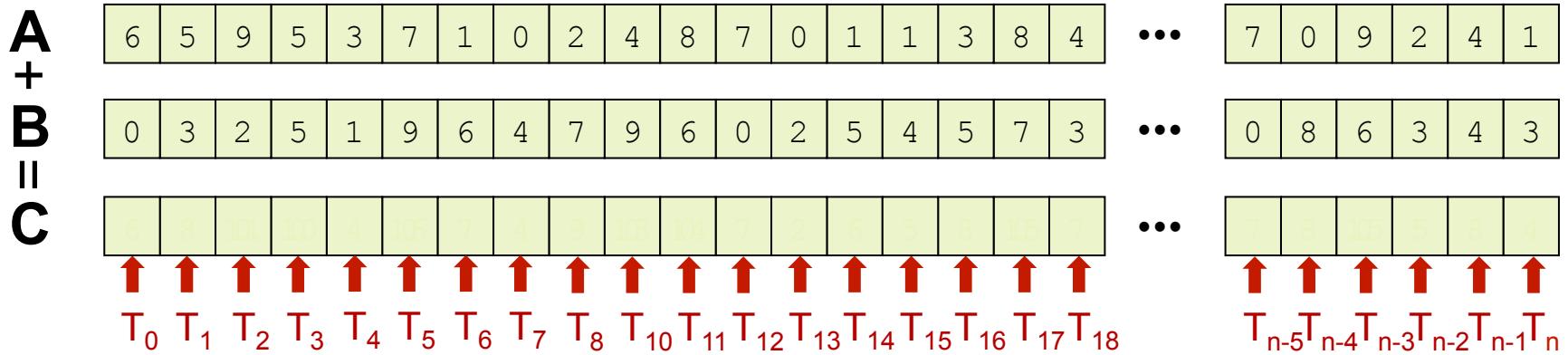
Simple example



```
__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

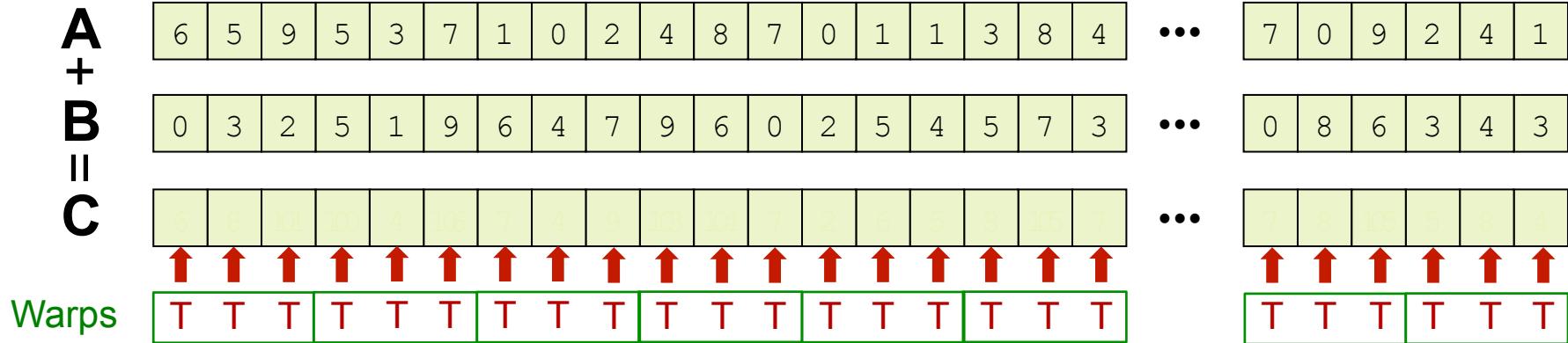
Simple example



```
__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



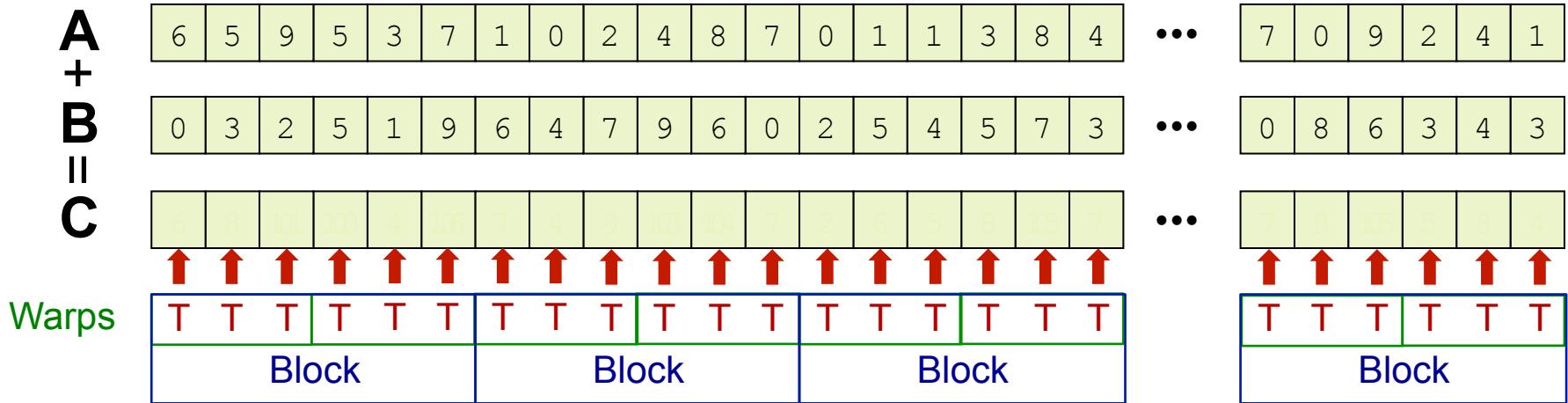
```

__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)

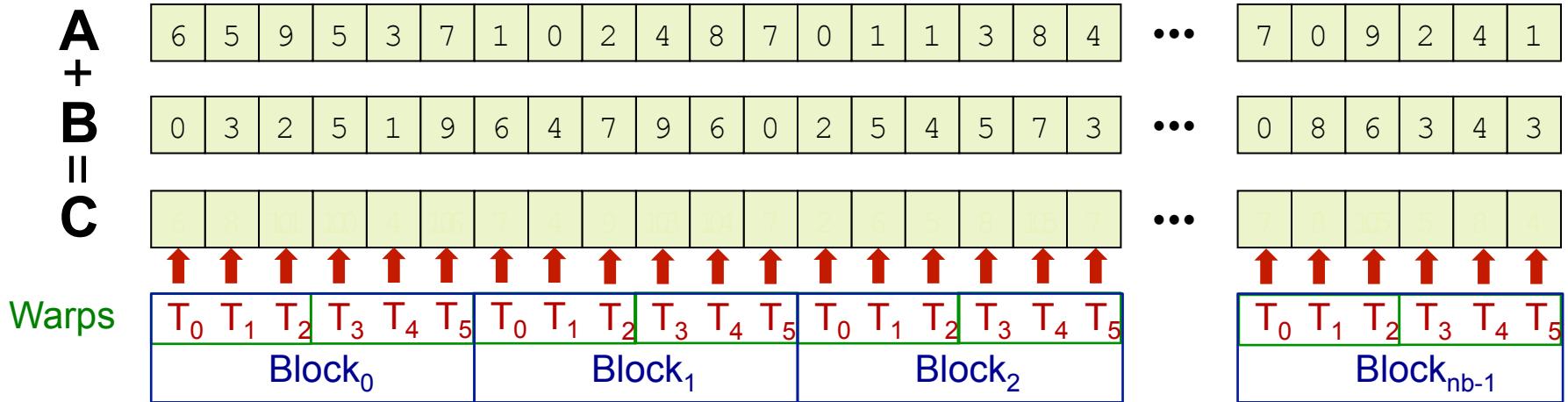
Simple example



```
__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

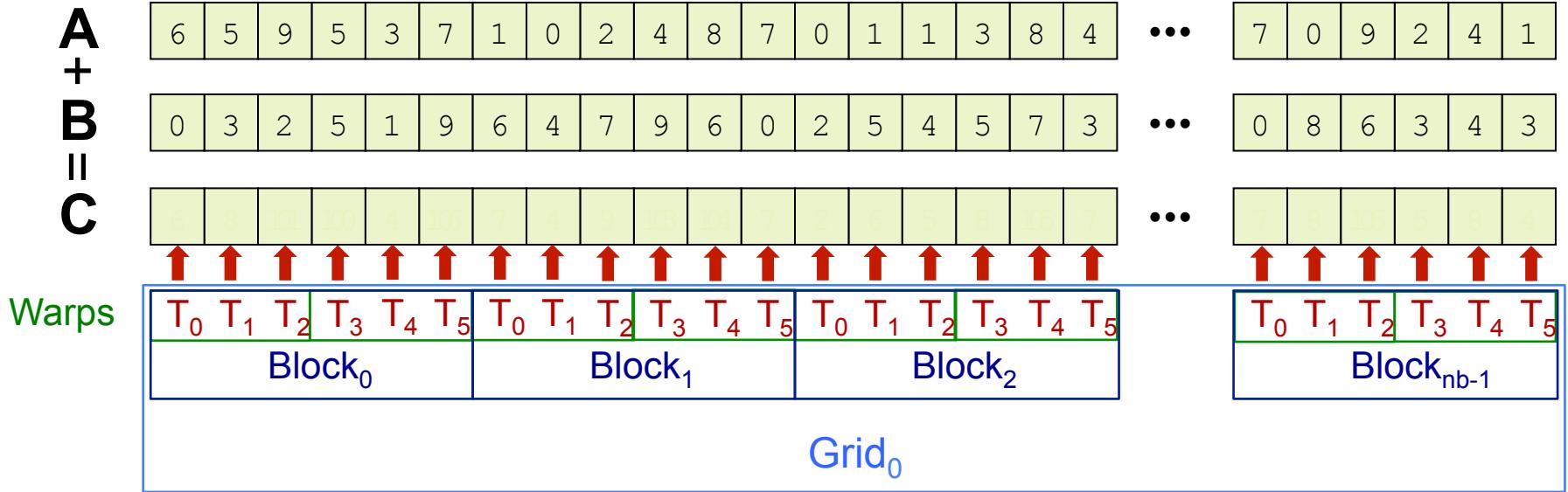
Simple example



```
__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];
}
```

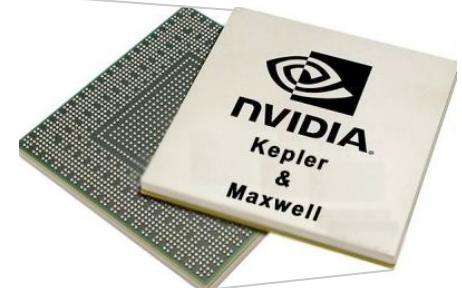
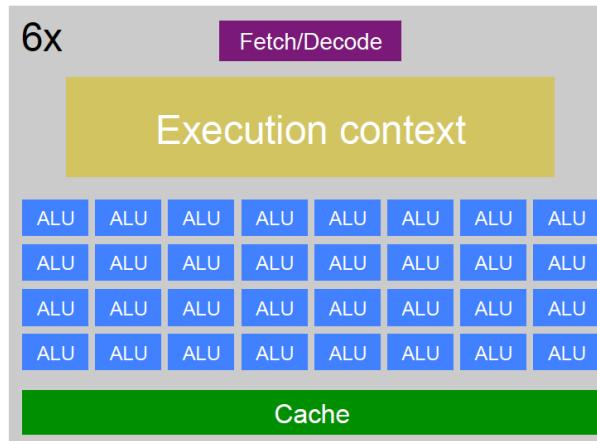
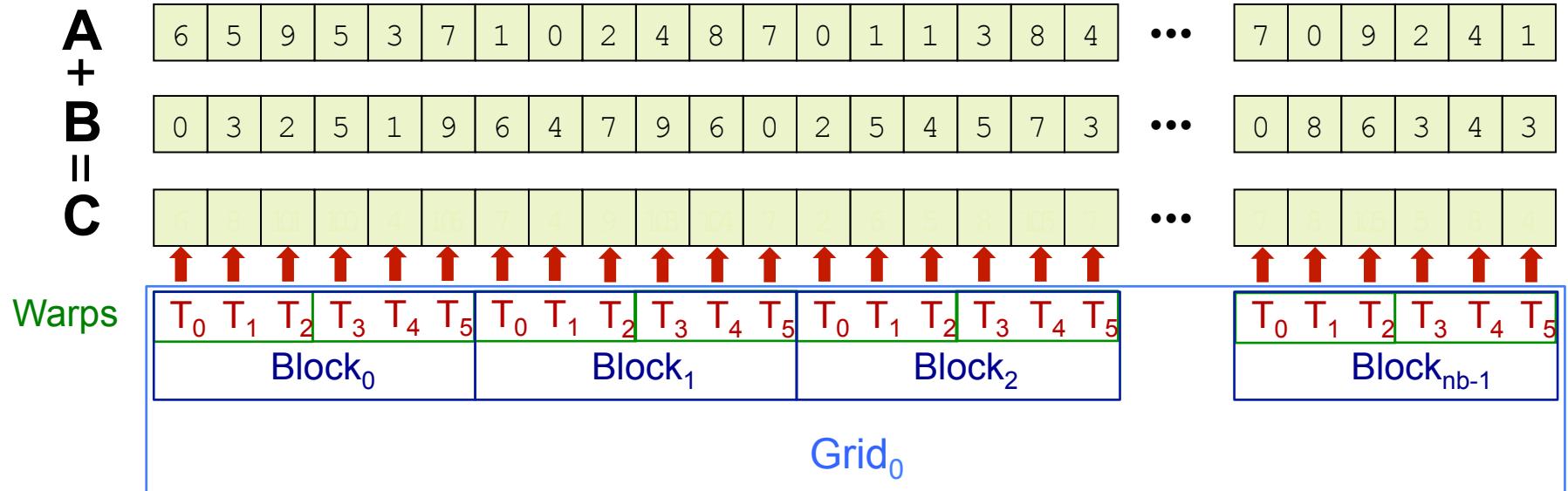
By enumerating all blocks and all threads within a block we can calculate the unique thread number

Simple example

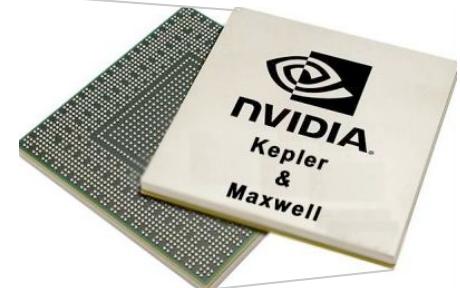
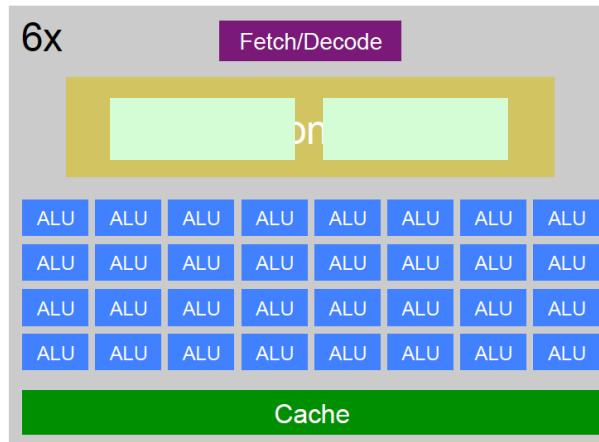
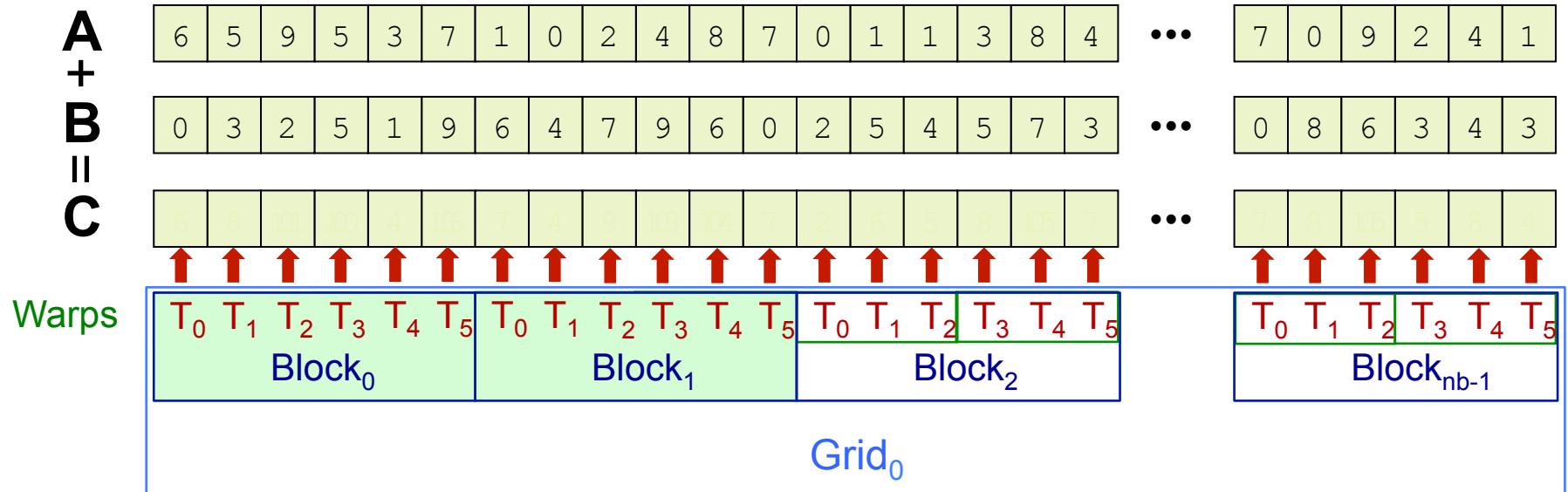


CUDA introduces grids, which are a grouping of the blocks, for launching more kernels at a time that may have different thread hierarchies

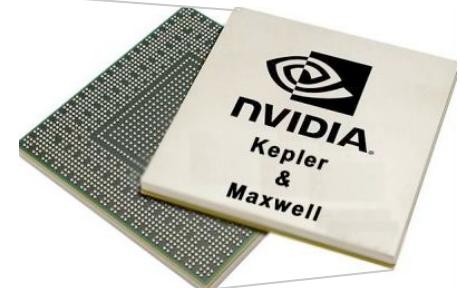
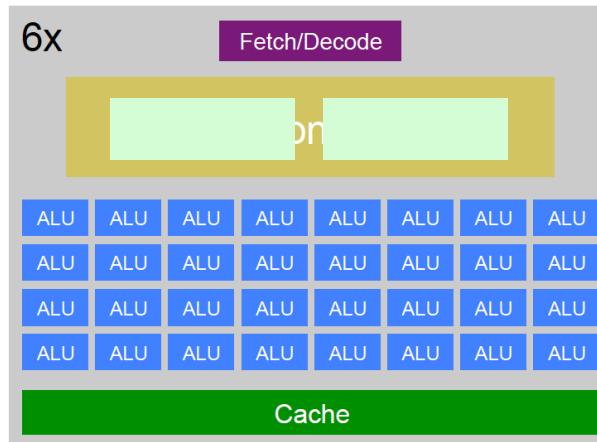
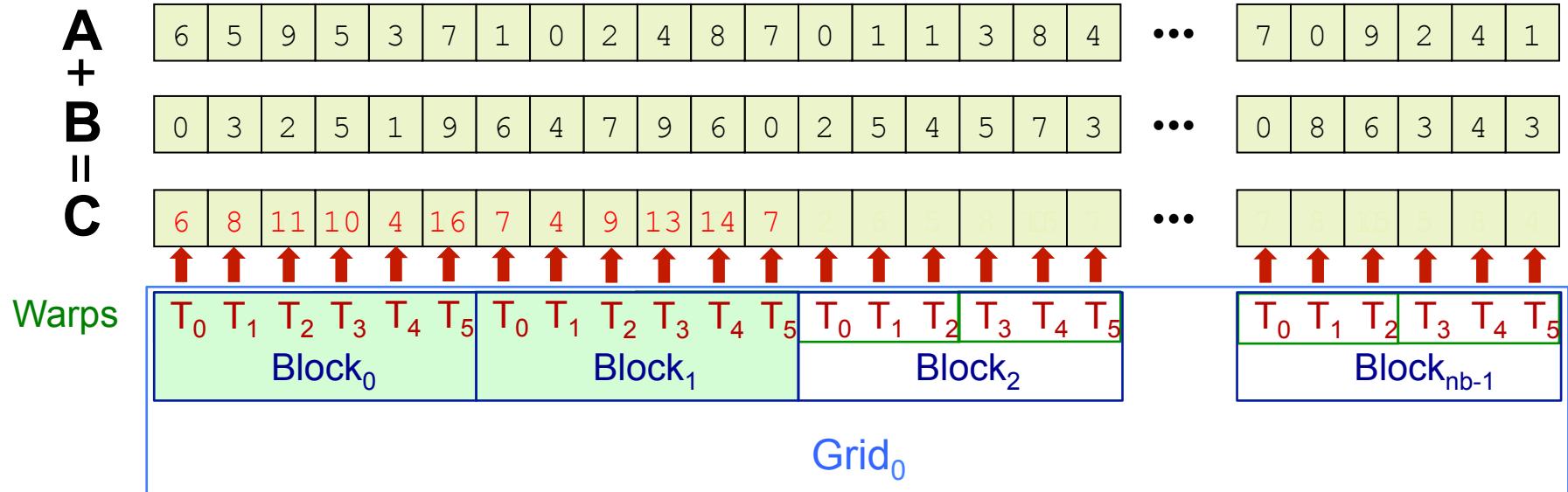
Execution at runtime



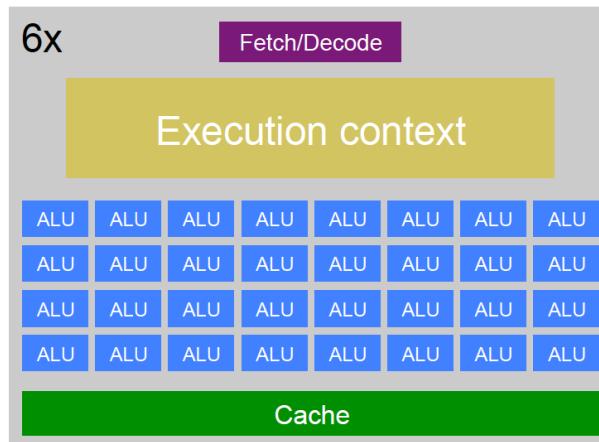
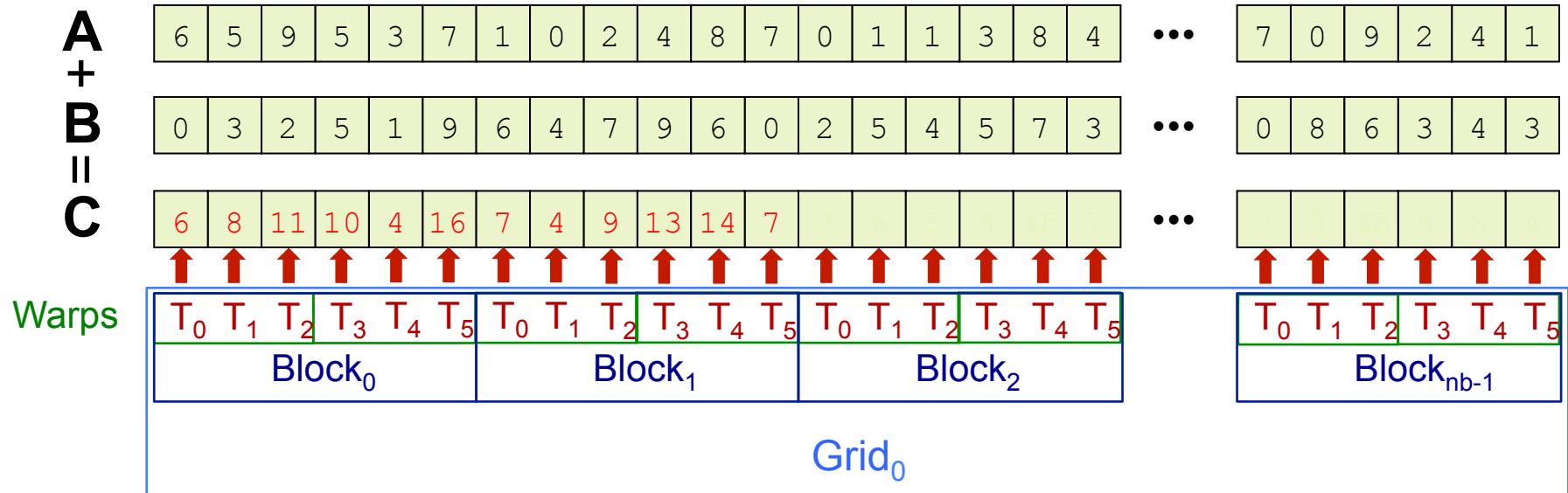
Execution at runtime



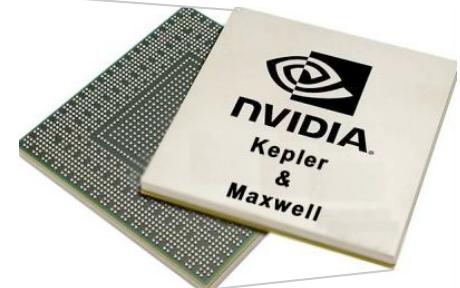
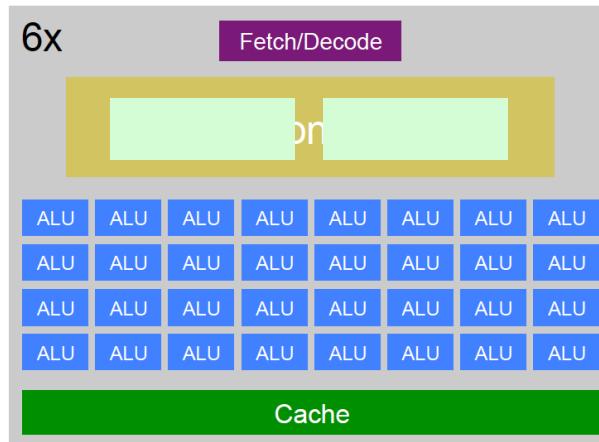
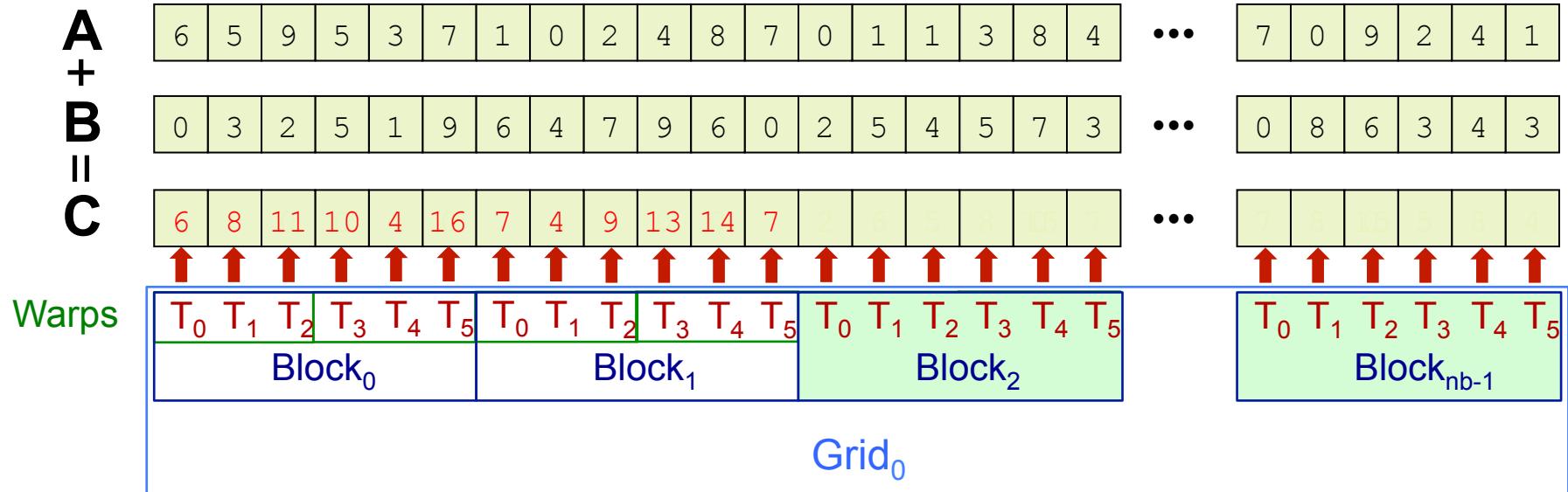
Execution at runtime



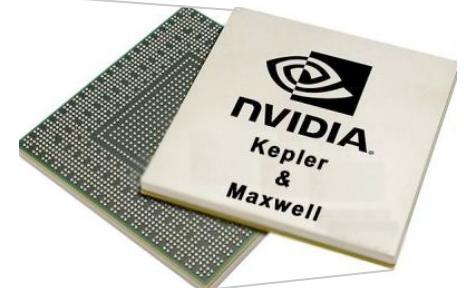
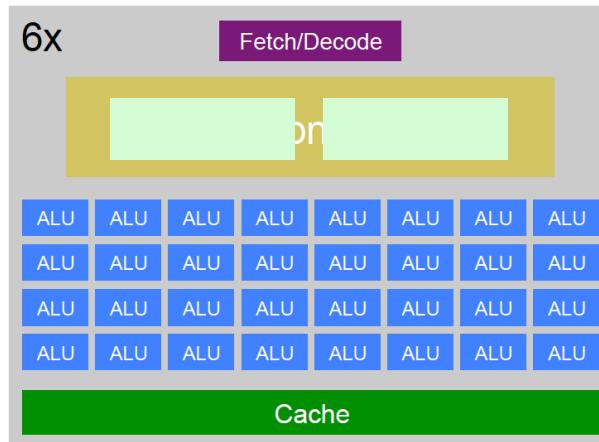
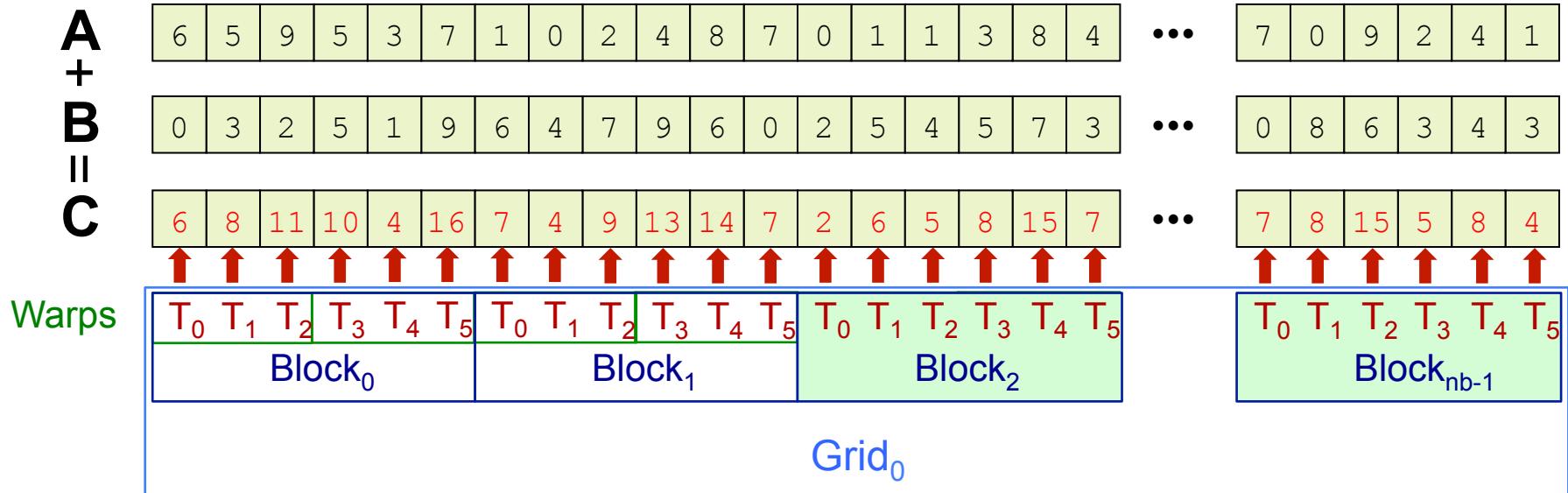
Execution at runtime



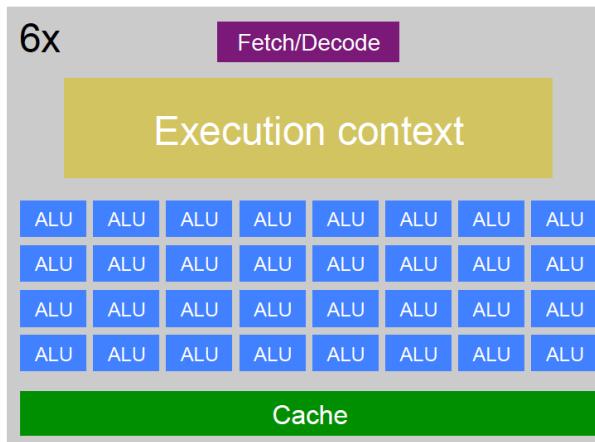
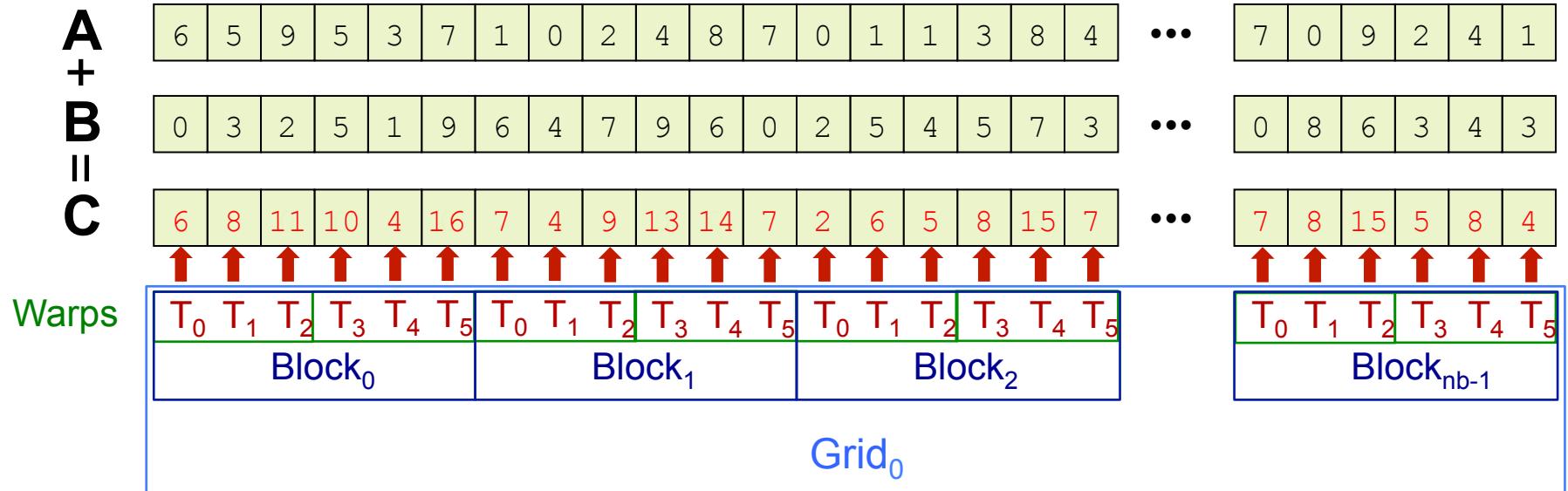
Execution at runtime



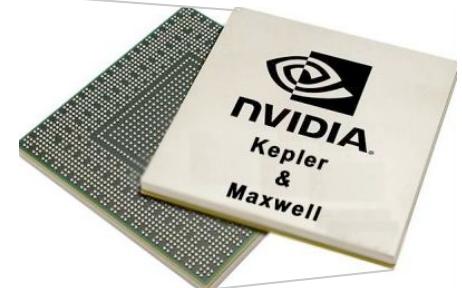
Execution at runtime



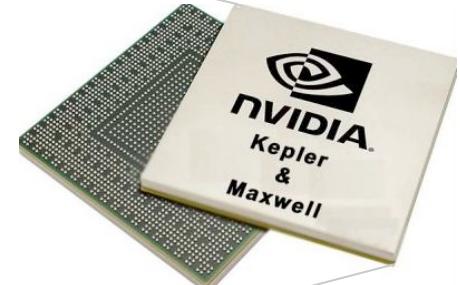
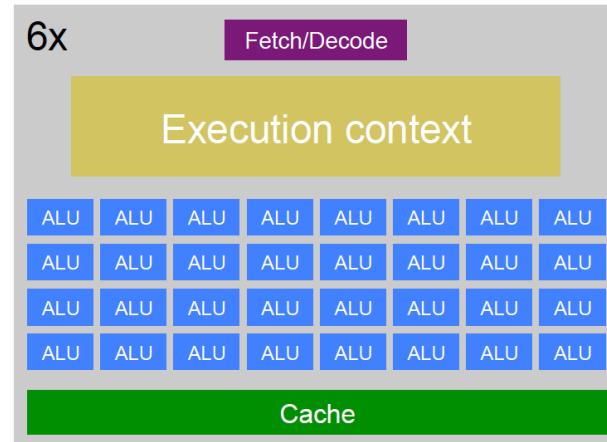
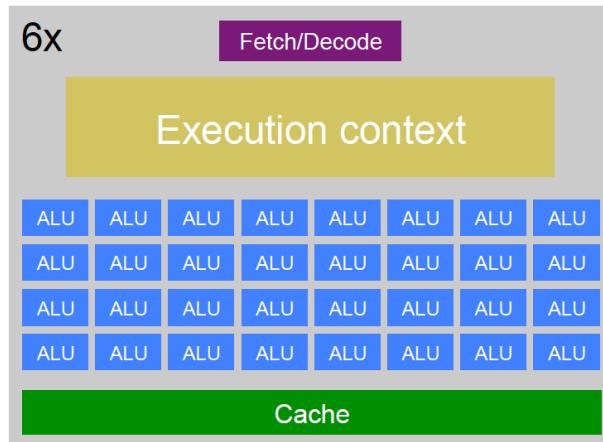
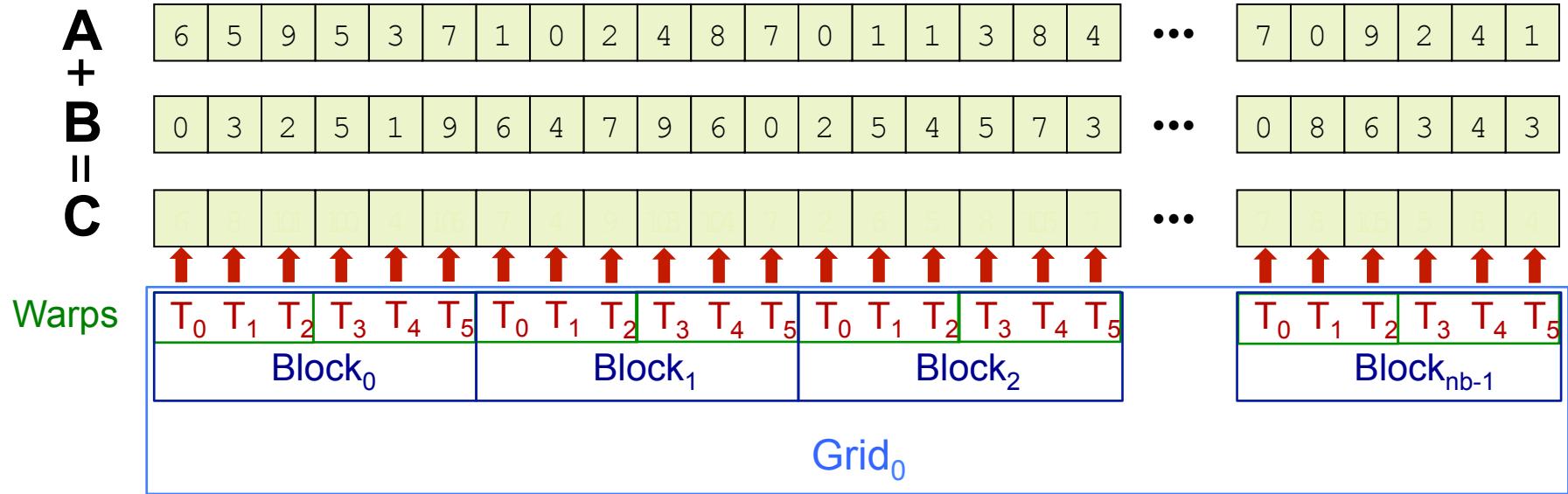
Execution at runtime



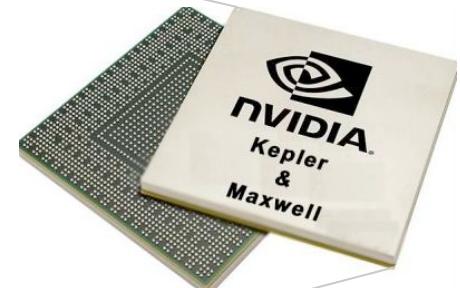
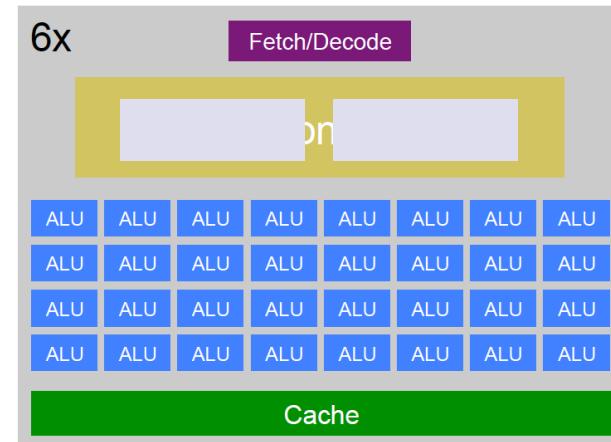
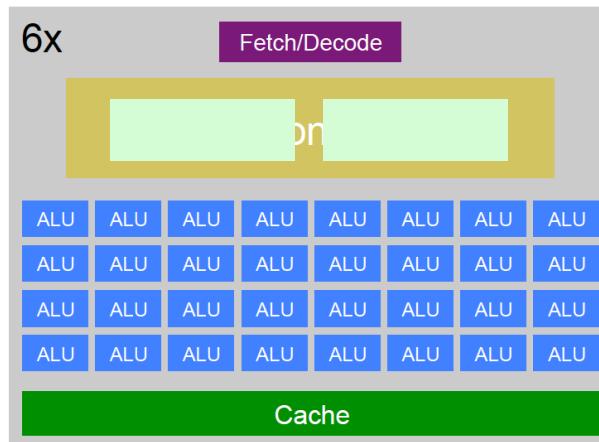
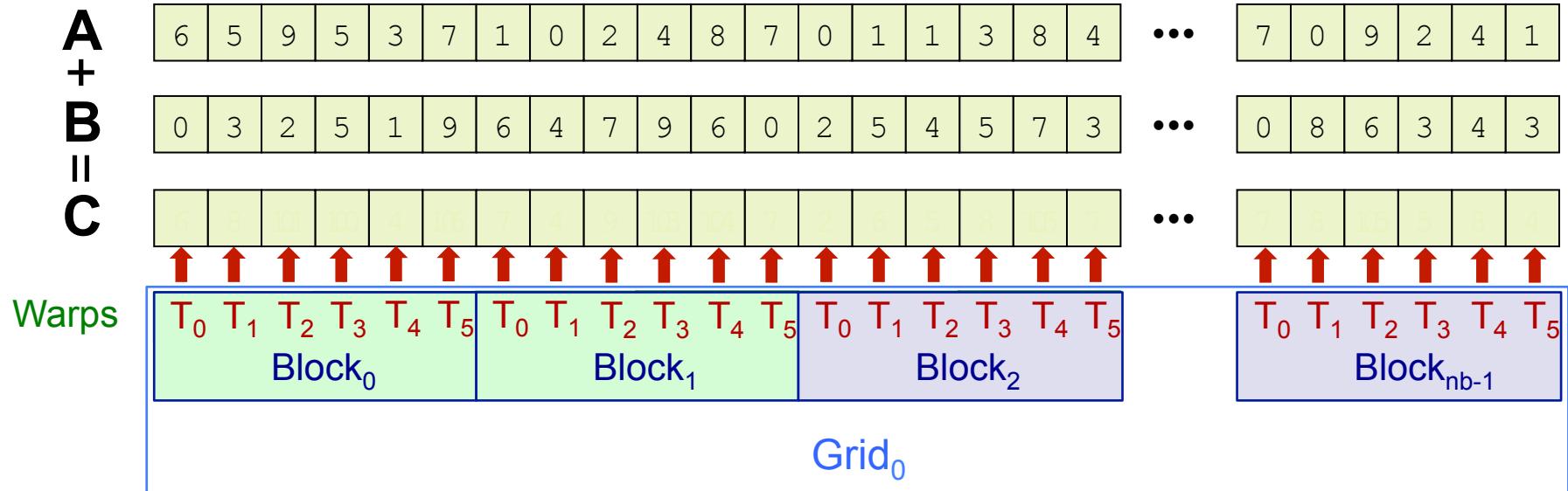
Blocks do not execute in any particular order – do not assume so in your code!



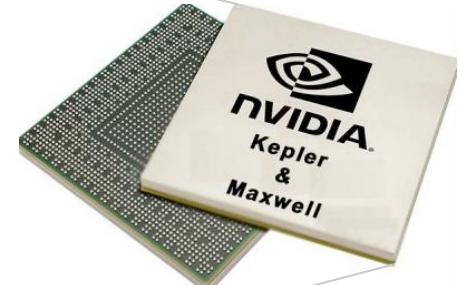
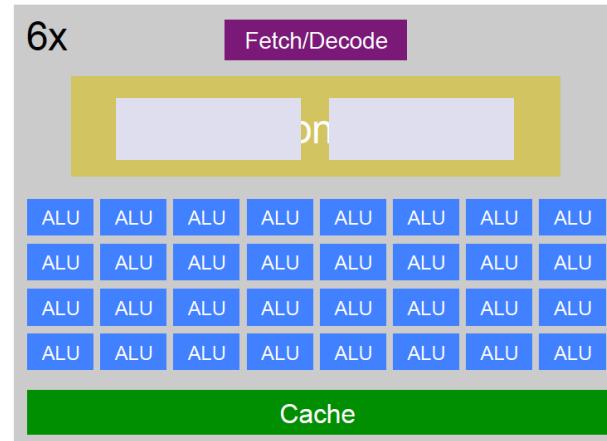
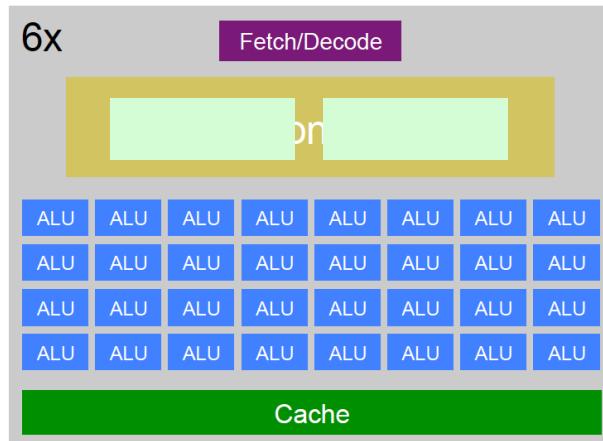
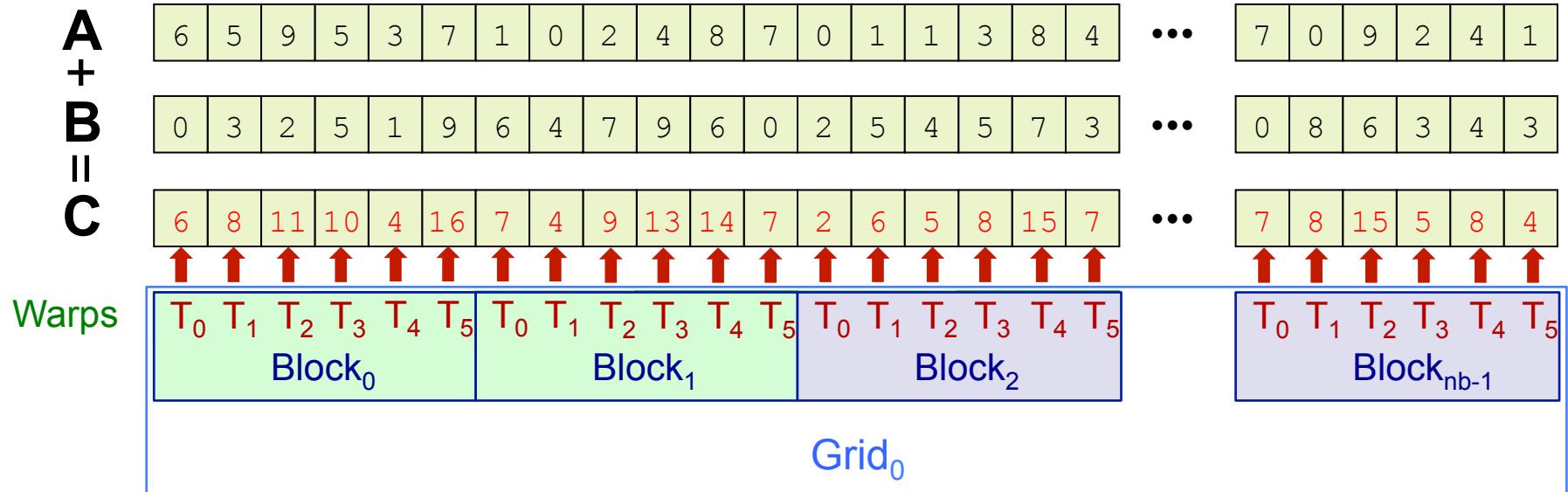
Execution at runtime



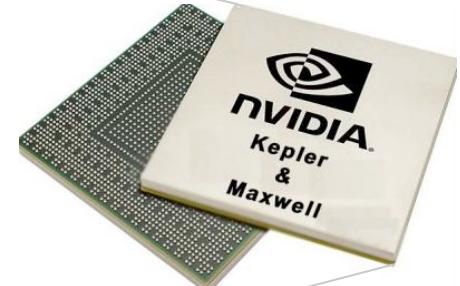
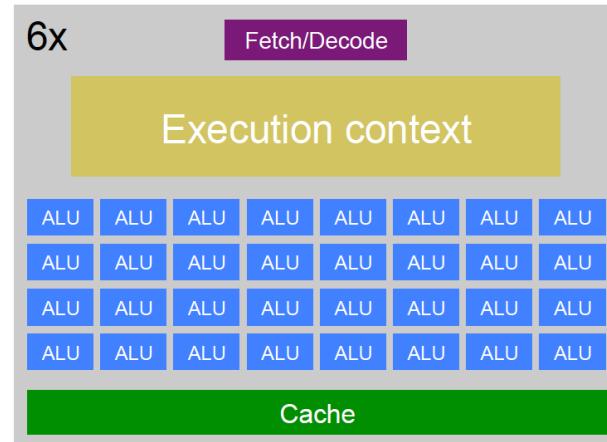
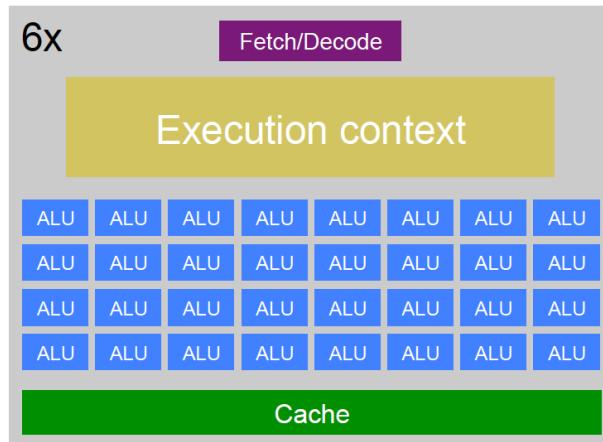
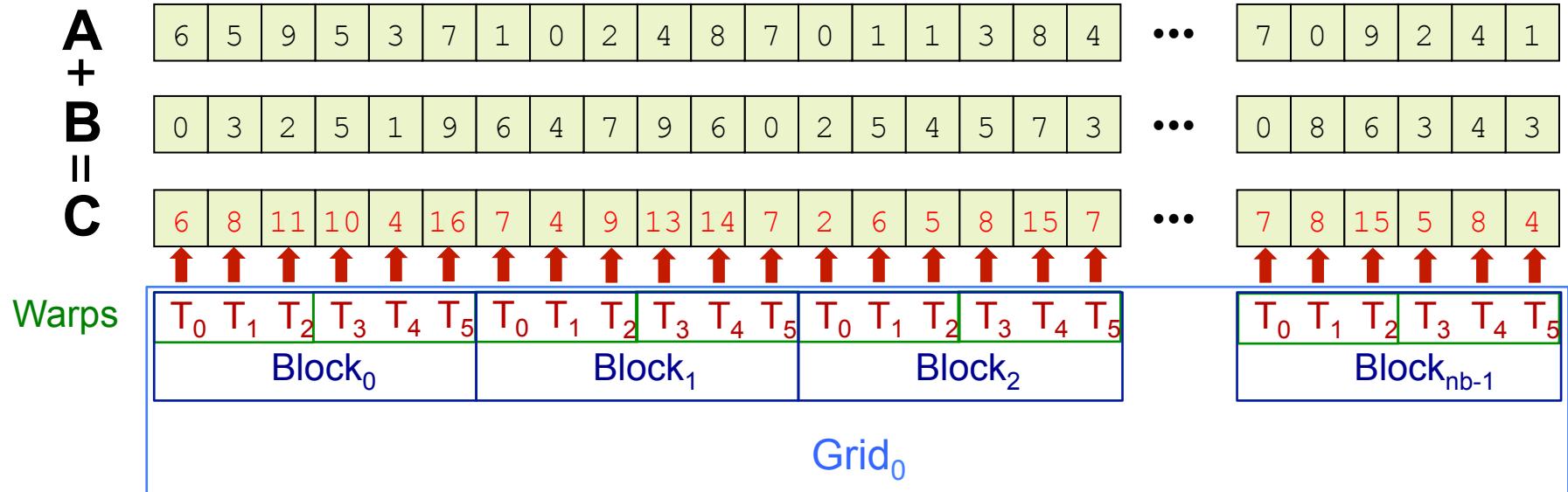
Execution at runtime



Execution at runtime



Execution at runtime



CUDA scheduling of blocks

■ Advantages

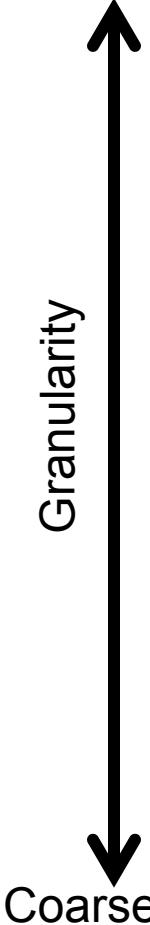
- Hardware can run things efficiently
- No queue from slow blocks
- Scalability
 - Applies from mobile phones to supercomputers
 - Now and in the future!

■ Consequences

- No assumptions on what blocks will run where
- No explicit communication between blocks
- All threads and blocks must complete

Summary of CUDA terminology

Fine



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

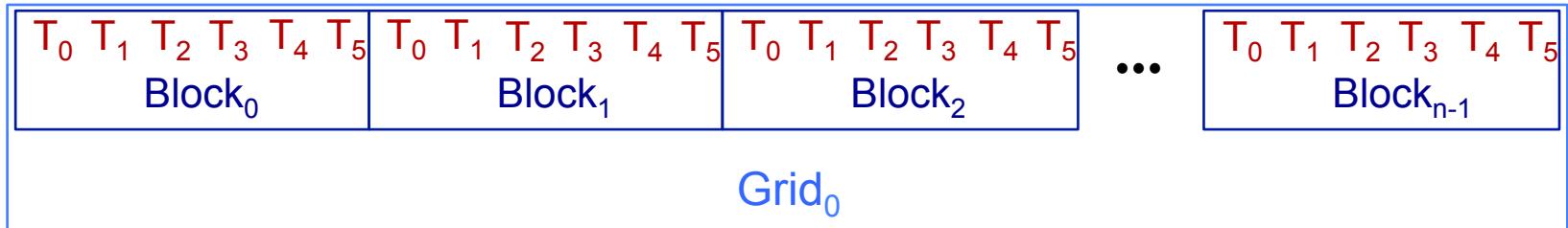
- Kernel invocation from host:

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

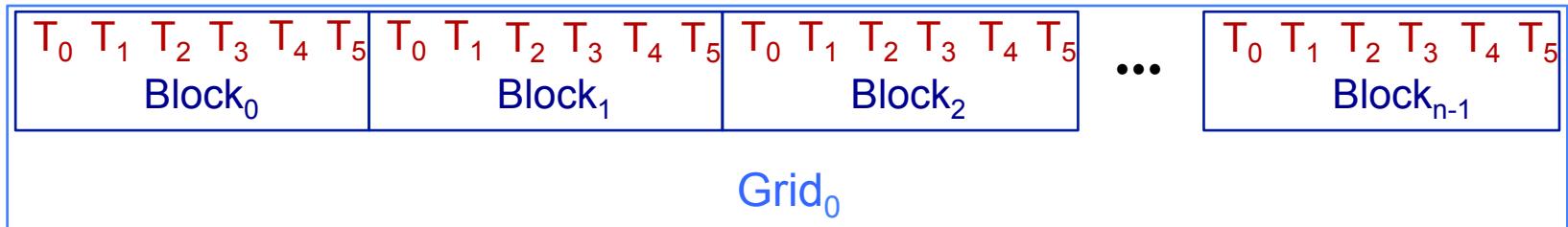
- Kernels are launched asynchronously from host and therefore explicit synchronization is needed to make sure we are finished.
- E.g., for the vector addition example

```
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?



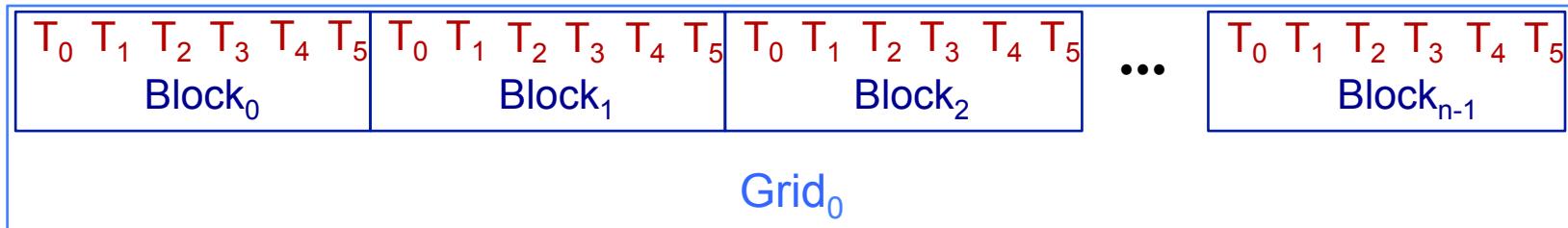
Which thread am I?



■ 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(...) { ... };
```

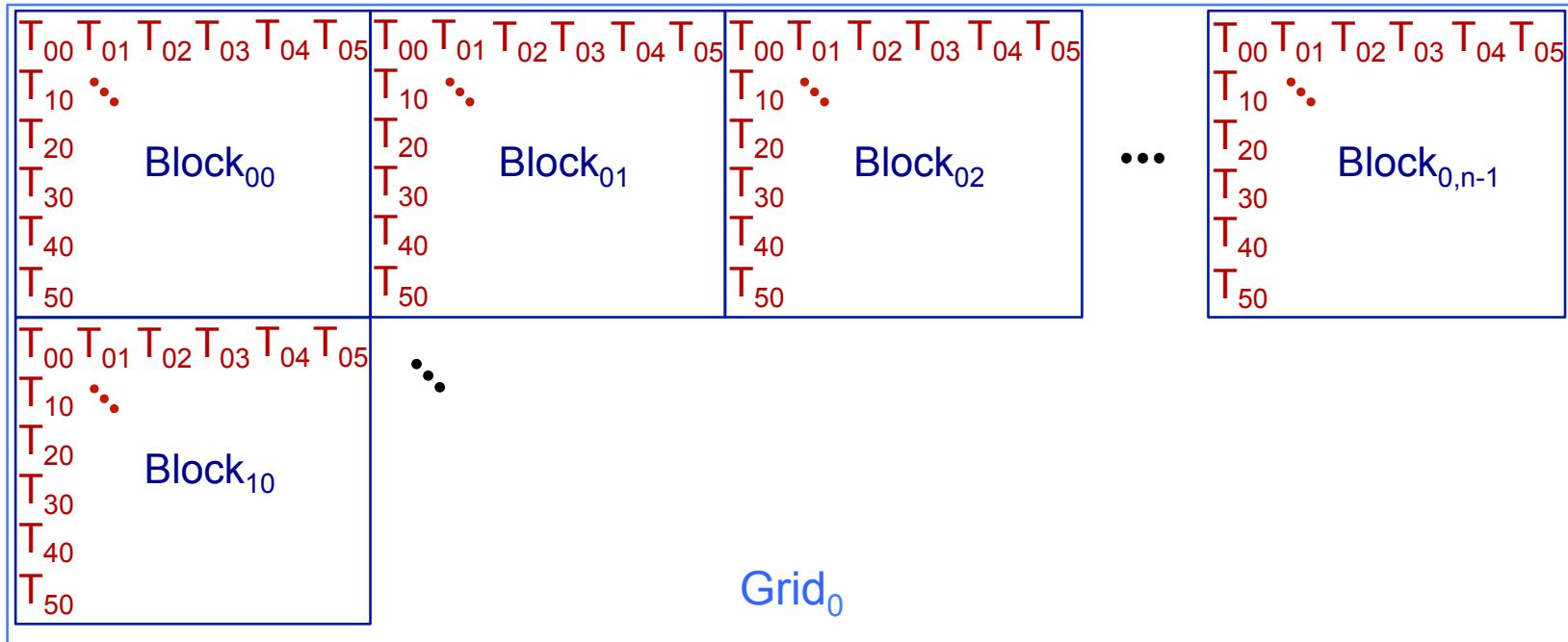
- Kernel execution configuration specified at host:

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

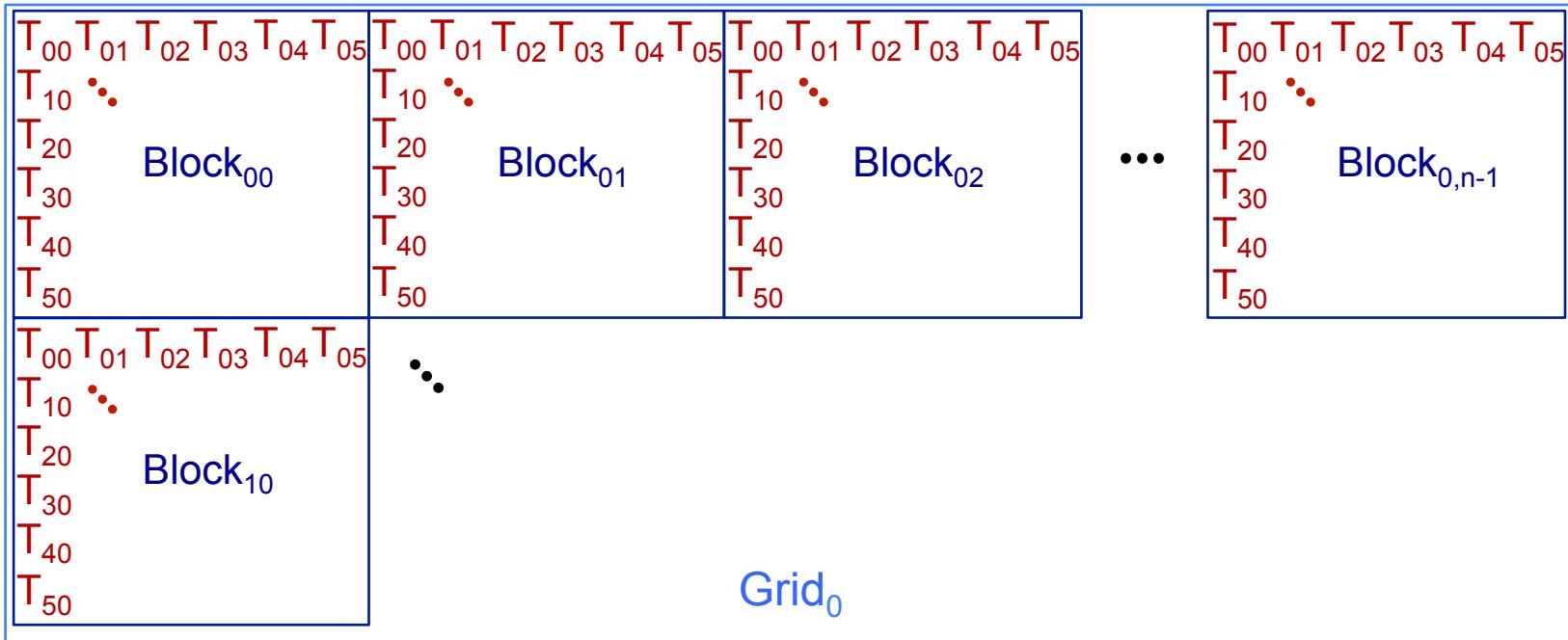
- Kernel invocation from host:

```
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 launch
    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