# Introduction to CUDA & CUDA+OpenGL

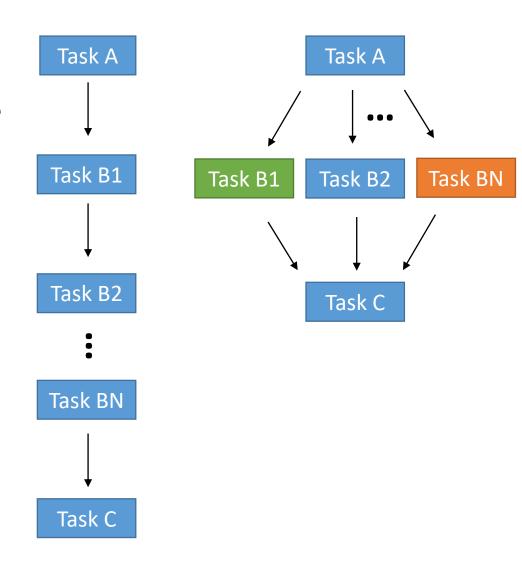
Philipp Haas 29.04.2021

https://github.com/phhaas/CUDASeminar\_Intro

# Parallelization in Computing

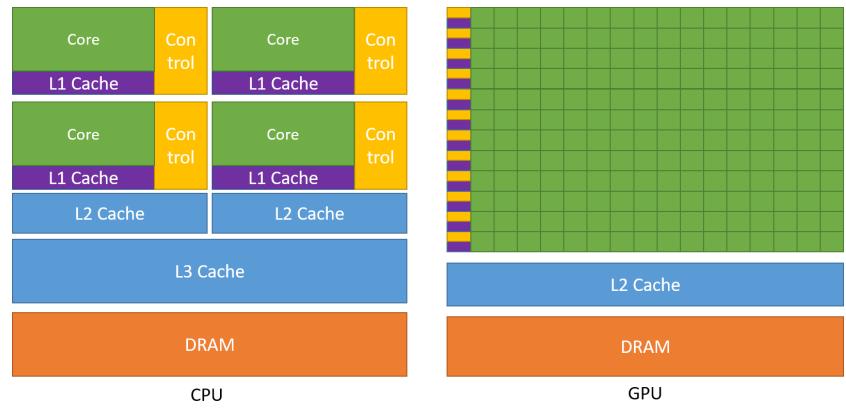
- Computing of tasks in parallel on different cores
- Works especially well for problems, where "Divide and Conquer" can be applied
  - Divide one big problem in many smaller subproblems, which are solved individually
  - Merge sort, hierarchical methods, FFT, machine learning, ...
  - In physics: MC, multi-body systems, metropolis algorithm, lattice gauge theory, ...

Sequentiell vs. Parallel Computing



#### GPU vs. CPU Architecture

- Why do we want to use the GPU for parallelization?
- ⇒ CPU is designed for few sequential tasks at a time, GPU for many parallel tasks (threads)
- ⇒ Less complex data flow, more processing



https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html

#### CUDA is ..

- a model for general-purpose computing on GPUs (GPGPU)
  - The CPU (host) gets access to additional ressources on the GPU (device)
- a platform to implement GPGPU on NVIDIA GPUs in several programming languages:
  - C, C++, Fortran, ...

#### 3 major concepts of CUDA:

- Kernels
- Thread Hierarchy
- Memory Management



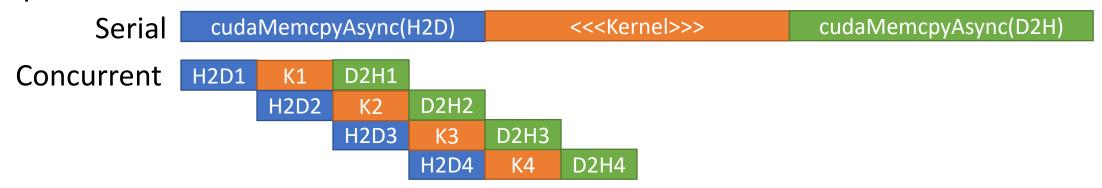


#### Kernels

- A kernel is similar to a C function, but is intended to be processed on the GPU
- For this only the specifier \_\_global\_\_ is needed:

```
__global__
void helloWorldDevice() {
    printf("Hello world from device!\n");
}
```

- The rest of the code runs on the host, while the kernel launches threads on the device
- Threads are non-blocking, they are processed as fast as possible
- Kernels can also be executed in concurrent streams for further parallelization



#### Kernels

• To run kernels in N different parallel threads, each thread has its own ID, the threadldx:

```
global
void addVectors(float* in1, float* in2, float* out) {
   //global index
    int i = threadIdx.x;
    out[i] = in1[i] + in2[i];
int main() {
   // code executed in CPU
   // vector addition (size N) executed element-wise in N threads on GPU
    addVectors << <1, N >> > (Din1, Din2, Dout);
   // code executed in CPU
```

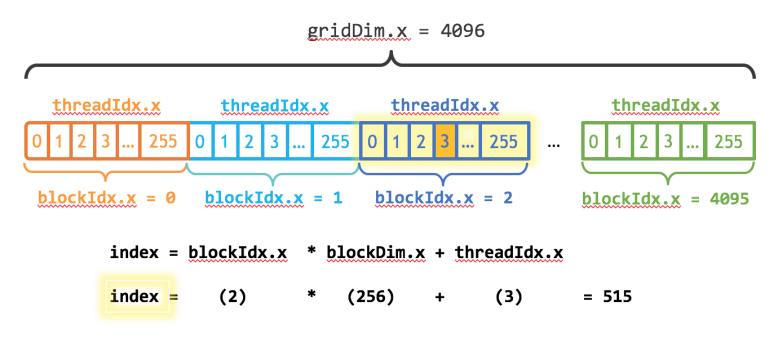
#### Thread Hierarchy

- A group of N threads is called block with ID blockldx
- A block runs on one streaming multiprocessor (SM) core
   => Limited threads (1024 for GTX 1080 Ti)
- If the vector size is larger than the limit, one can use M blocks organized in a grid

```
__global__
void addVectors(float* x, float* y) {
    // global index
    int i = blockIdx.x * blockDim.x + threadIdx.x;
    y[i] = x[i] + y[i];
}
int main() {
    // ...
    addVectors<<<M, N>>>(x, y)
    // ...
}
```

## Thread Hierarchy

- Every block must be independently executable from all others in a grid
  - ⇒ No constraints on scheduling scheme of the blocks => Scalable
- Index of blocks and threads can also be organised in arrays (2D & 3D)



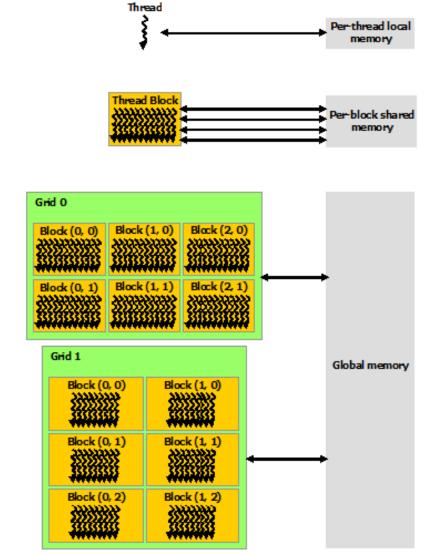
https://developer.nvidia.com/blog/even-easier-introduction-cuda/

## Thread Hierarchy – Warps

- Threads within a block are organised in warps
  - ⇒32 threads for current GPU's
- The SM executes all threads in one warp with the same instruction
  - ⇒By using the same instructions, memory access is significantly reduced
  - ⇒If the executed code between threads in a warp take different control paths, they diverge and benefits from warps are lost
    - ⇒ Control divergence

## Memory Management

- Memory in CUDA is managed hierarchically:
  - Every thread has local memory
  - All threads of a block have a common shared memory
  - All threads share a global memory
- This device memory is assumed to be independent of the one of the host
  - ⇒ Device memory has to be managed via calls to CUDA, not visible for CPU
- Exception: Managed/Unified Memory visible for both CPU/GPU



https://docs.nvidia.com/cuda/cudac-programming-guide/index.html

#### Memory Management – Data Transfer

- Since host and device memory are independent, both have to be managed independently:
  - Allocate host memory:

```
float* XHost = (float*)malloc(N*floatSize);
```

Allocate device memory:

```
float* XDevice;
cudaMalloc(&XDevice, N*floatSize);
```

Copy from host to device memory:

```
cudaMemcpy(XDevice, XHost, N*floatSize, cudaMemcpyHostToDevice);
```

Copy from device to host memory:

```
cudaMemcpy(XHost, XDevice, N*floatSize, cudaMemcpyDeviceToHost);
```

Free device memory:

```
cudaFree(XDevice);
```

## Memory Management – Synchronisation

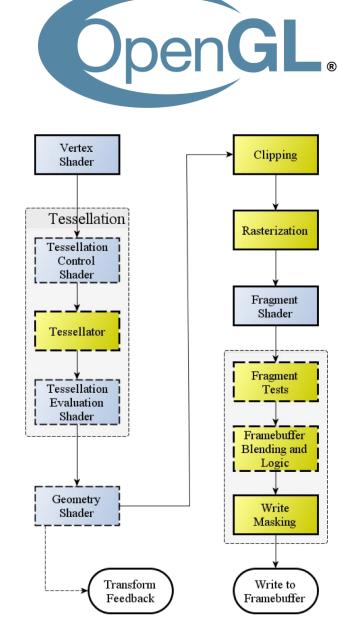
- Synchronisation of shared memory (memory in a block):
  - Shared memory is allocated in kernels using the <u>shared</u> specifier:

```
__global__
void func(...) {
    __shared__ float shared[N];
    // ..
}
```

- \_syncthreads() forces all threads in a block to wait until all others have cought up
- Synchronisation of host to devices: cudaDeviceSynchronize()

## OpenGL

- Open source 3D vector graphics rendering API
- Designed to exploit hardware acceleration (GPU)
- Language bindings for many programming languages
  - C, C++, Fortran, ..
- State-based
- Parts of the OpenGL workflow can be parallelized (blue boxes in the workflow chart)
- Shaders: Applying simple tasks to a large set of elemts
  - ⇒Well suited for parallel computing



https://opengl-notes.readthedocs.io/en/latest/topics/intro/opengl-pipeline.html

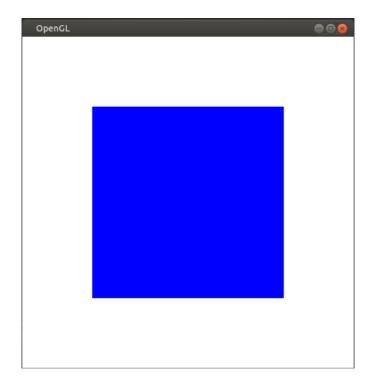
#### OpenGL

- Initialisation:
  - Set callback functions, display mode, ...
- Callback functions:
  - glutDisplayFunc()
  - glutMouseFunc()
  - glutMotionFunc()
  - ...
- Rendering loop:
  - glutMainLoop()
  - Infinite loop until program is terminated
  - Executes callback **glutDisplayFunc()**
  - Reacts on other callback functions

#### OpenGL – Simple Drawing

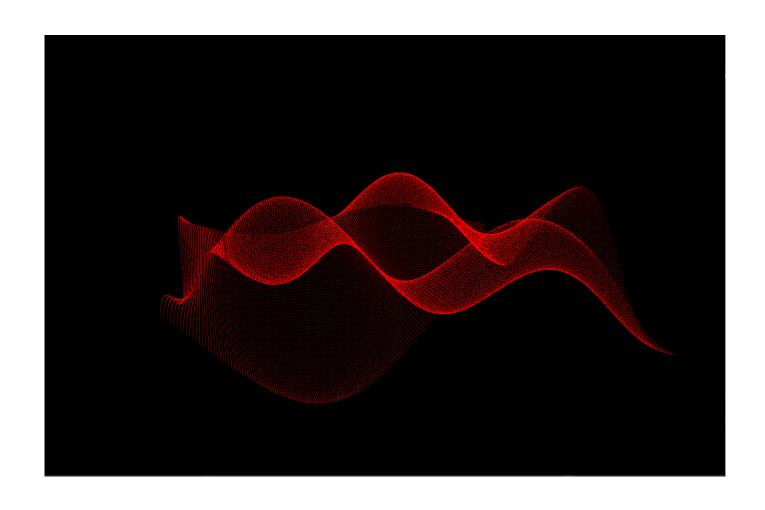
• Drawing a simple rectangle:

```
glBegin(GL_QUADS); // connects group of 4 vertices to polygon with 4 sides
    glVertex2f(-1.0, -1.0); // bottom-left vertex
    glVertex2f(-1.0, 1.0); // top-left vertex
    glVertex2f( 1.0, 1.0); // top-right vertex
    glVertex2f( 1.0, -1.0); // bottom-right vertex
glEnd();
```



## OpenGL+CUDA Interoperability

- Reduce trafic between CPU and GPU for less PCIe latency
- Example: <CUDA-samples>/2\_Graphics/simpleGL



#### How to get CUDA-samples running?

Copy sample to your home directory:

cp -r /mount/share/cuda-samples/2\_Graphics/simpleGL . ; cd simpleGL

• Edit "Makefile": e.g.

#### nano Makefile

• Edit following lines:

```
CUDA_PATH ?= /usr/local/cuda-11.0 INCLUDES := -I../../common/inc SMS ?= 35 37 50 52 60 61 70 75 80
```

- → CUDA\_PATH ?= /usr
- → INCLUDES := -I/mount/share/cuda-samples/common/inc
- → SMS ?= <CUDA Capability version number>
- The CUDA Capability version number is printed by calling

/mount/share/cuda-samples/deviceQuery

Comment out (#) following lines:

```
$(EXEC) mkdir -p bin/$(TARGET_ARCH)/$(TARGET_OS)/$(BUILD_TYPE) $(EXEC) cp $@ bin/$(TARGET_ARCH)/$(TARGET_OS)/$(BUILD_TYPE)
```

• Run make

#### Sources

- https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html
- https://developer.nvidia.com/blog/even-easier-introduction-cuda/
- https://www.khronos.org/opengl/wiki\_opengl/index.php
- https://www.opengl.org/
- https://www.3dgep.com/introduction-opengl/
- https://www.3dgep.com/opengl-interoperability-with-cuda/
- https://opengl-notes.readthedocs.io/en/latest/index.html