Advanced Topics in Computer Graphics I

Universität Bonn
Institut für Informatik II
April 8, 2025

Summer term 2025
Prof. Dr. Reinhard Klein
Domenic Zingsheim

Sheet R00 - Introduction to CUDA

Hand in your solutions via eCampus by Tue, 15.04.2025, **12:00 p.m.**. Compile your solution to the theoretical part into a single printable PDF file. For the practical part, hand in a single ZIP file containing only the exercise* folder within the src/ directory. Please refrain from sending the entire framework.

In this lecture you will have the chance to learn many interesting theoretical as well as practical topics. In the case you have any problem understanding, please always feel free to ask in the discussion forum on eCampus[1]. This should be a place where you students can talk freely about the lecture, so do not hesitate to ask *and* reply! Of course the tutors are there as well and will also reply.

Assignment 1) Installation *(0Pts)*

For all the exercises we recommend to use a Linux OS, e.g. Ubuntu 22.04 LTS. The code should also work on Windows, but the probability of encountering technical difficulties is higher there. For Windows, we will provide limited support only. In case of problems we recommend switching to a Linux based OS.

In the exercises we will do a lot of ray tracing using NVIDIA's OptiX API [2] that is built on CUDA, which itself is an API created to use your GPU for general purpose processing. The caveat is that OptiX only works with NVIDIA GPUs of the Maxwell architecture (GTX 9xx) and later, but at least you don't need an RTX-class GPU. If you don't have a sufficient GPU at home, you can use the pool computers[3] in the basement of our institute. We won't need the OptiX API right now, but we will install it here anyways and introduce it in a later exercise.

Make sure that the following list of software is installed on your computer:

- NVIDIA CUDA Toolkit, preferably the latest version or at least 11.6, which you can find at `https://developer.nvidia.com/cuda-downloads`.
- NVIDIA OptiX SDK $\geq$ 7.4, which you can find at `https://developer.nvidia.com/designworks/optix/download`. Note that you might need to update your drivers if you choose the newest version.
- CMake to create the project files of the exercise framework for your platform. On Windows: Make sure that the `bin` directory of the CMake installation is added to the `PATH` environment variable. Otherwise you might have to type in the full path to the `cmake.exe` executable.
- On Windows: Visual Studio Community 2022 with "Desktop development with C++", other versions might work as well if you are lucky.
- On Linux: CMake, recent G++ or Clang++ compiler capable of C++17 and various libraries that CMake will complain about during the initial configuration of the exercise framework.
- Your favourite code editor.

[1] `https://ecampus.uni-bonn.de/goto_ecampus_frm_3325448.html`
[2] `https://developer.nvidia.com/optix`
[3] `https://gsg.informatik.uni-bonn.de/doku.php?id=en:pool`

When everything is installed continue as follows:

- On Linux, create a `build` directory in the exercise framework, and inside the `build` directory call `cmake ..`.
- On Windows, simply open the exercise frameworks root directory in Visual Studio, you should be prompted to configure the project using CMake.

In any case, make sure to create a project that builds 64-bit executables (should be the default by now). Now you can build the exercise framework by running `make` or compiling the Visual Studio solution. Executables will be placed into the `bin` directory that will be created in the exercise frameworks root directory. If everything is installed correctly and the compilation of the exercise framework was successful, you should be able to run the `bin/exercise00_CUDAIntro[.exe]` executable.

Hint: If CMake is unable to find the OptiX SDK, you can copy the include directory of the SDK and place it into the directory `external/OptiX` inside the exercise framework.

Assignment 2) Introduction to CUDA *(0 Pts)*

While we will use C++ to work with the CUDA and OptiX frameworks, giving a C++ introduction is out of scope for this module. In case you are not yet familiar with the C++ programming language, please work through one of the many great online resources as soon as possible. For our purposes we only need very basic knowledge of CUDA. Here are a few links to check out:

- `https://developer.nvidia.com/blog/even-easier-introduction-cuda/`
  An Even Easier Introduction to CUDA – Very basic tutorial on how to get started with CUDA.
- `https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html`
  CUDA C++ Programming Guide – A more elaborate guide on how to use CUDA efficiently and how it maps to hardware.
- `https://docs.nvidia.com/cuda/index.html`
  CUDA Toolkit Documentation – Useful as a reference if you are searching for a particular function.

In computer graphics, especially in rendering, we naturally work with two to four dimensional vectors and matrices a lot of the time. The built-in datatypes (e.g. `float3`) that CUDA provides for this purpose are unfortunately a little bit clumsy to use because common arithmetic operations are not defined on them. The exercise framework uses the GLM library, which also works well inside of CUDA kernels. This library provides the types and functionality that you would expect when writing GLSL (or HLSL) shaders. Most math operations like dot product, cross product, min, max, floor, lerp, etc. are provided in GLM for scalar (e.g. `float`), vector (e.g. `glm::vec3`) and matrix (e.g. `glm::mat4`) types. Since we have to interact with the CUDA built-in types later on, we added the handy conversion functions `glm2cuda()` and `cuda2glm()` which convert input vector types to the corresponding output vector types.

Assignment 3) Working with CUDA and GLM *(6 Pts)*

You can find the framework used for the exercises on the lecture homepage and on eCampus. We have split the framework for this exercise in two parts. The `framework_base.zip` is the core structure of the framework and is reused in future exercises. The `framework_r00.zip` contains the additional code relevant for this exercise. You can simply extract both archives into the same folder.

The framework might seem a little bit large at first glance, but it is actually quite simple. The `CMake`, `external` folders contain CMake scripts and external libraries that are used during compilation. You don't have to worry about those. More interesting is the `data` folder, which currently only contains a single image, but we will add scene description files here in future exercises. The `opgutil` folder is the "core" utility library, and it contains some useful classes and methods that are reused in future exercises. The library will also be extended in future exercises. Most important is the `src` folder, in which a new subfolder will be added for each exercise sheet. The code for the current exercise is located in `src/exercise00_CUDAIntro`.

a) The set of points $\mathcal{P}$ that lie on a plane in 3d space is defined as

$$\mathcal{P} := \left\{\mathbf{x} \in \mathbb{R}^3 \mid \langle \mathbf{n}, \mathbf{x} \rangle + d = 0\right\}, \tag{1}$$

where $\mathbf{n} \in S^2$ is a unit-length vector and $d \in \mathbb{R}$ is a scalar.
A ray in 3d space can be written as a function $\mathcal{R} : \mathbb{R} \to \mathbb{R}^3$.

$$\mathcal{R}(t) := \mathbf{o} + t \cdot \mathbf{d}, \tag{2}$$

with origin $\mathbf{o} \in \mathbb{R}^3$ and direction $\mathbf{d} \in \mathbb{R}^3$.
Using the GLM math library in the C++ framework:

- Determine the parameters $\mathbf{n}$, d that describes the plane spanned by the points

$$\begin{pmatrix} 3 \\ 0 \\ 2 \end{pmatrix}, \quad \begin{pmatrix} 4 \\ 3 \\ 2 \end{pmatrix}, \quad \begin{pmatrix} 1 \\ 2 \\ 4 \end{pmatrix}, \tag{3}$$

  given in the code as `glm::vec3` vectors.
- Swap the y and z components of the resulting plane normal.
- Compute the ray-plane intersection between the previously computed plane and the ray with

$$\mathbf{o} := \begin{pmatrix} 1 \\ 4 \\ 2 \end{pmatrix}, \quad \mathbf{d} := \begin{pmatrix} 2 \\ 4 \\ 3 \end{pmatrix}, \tag{4}$$

  that are given in the code as CUDA-builtin `float3` vectors. Convert the ray parameters to `glm::vec3` using the `cuda2glm()` function. Implement the ray-plane intersection. Convert the intersection point back to a CUDA-builtin `float3` vector using the `glm2cuda()` function.

b) Generate an array of integer numbers $[1, \ldots, 10^7]$ on the GPU. Use a CUDA kernel to multiply each number by a constant in parallel.

c) Apply the separable Sobel filter (`https://en.wikipedia.org/wiki/Sobel_operator`) to an image. Use two kernel executions for each dimension. Output $\mathbf{G} = \sqrt{\mathbf{G}_x^2 + \mathbf{G}_y^2}$.
Hint: You have to synchronize the execution on the GPU between both kernel launches! You can use the macro `CUDA_SYNC_CHECK()` defined in `opgutil/opg/exception.h` for this purpose.

d) Generate an array of uniformly distributed random floating point numbers between 0 and 1 on the GPU. Count all entries that are greater than 0.5 in parallel.
Hint: The header `opgutil/opg/hostdevice/random.h` defines utilities for random number generation. You can use the function `uint32_t sampleTEA32(uint32_t a, uint32_t b)` to initialize a seed for a random number generator based on a 2D index. The struct `PCG32` provides an implementation of a "permuted congruential generator" to produce a stream of pseudo-random numbers. The given implementation provides methods for sampling uniformly distributed floating point numbers in $[0, 1)$ using the member functions `float PCG32::nextFloat()`, and `glm::vecX PCG32::nextXD()` for creating higher dimensional samples, where `X` is in $\{1, \ldots, 4\}$.
Hint: You have to synchronize the counter accesses e.g. by using `atomicAdd(...)`!

e) Use the concepts you have learned above to implement a general matrix multiplication (with large dimensions) efficiently.

Good luck!