Chair for System Simulation

Harald Köstler, Dr.-Ing.
Sebastian Kuckuk, M. Sc.
Sebastian Eibl, M. Sc.

High End Simulation in Practice

Summer Term 2016

Assignment 1: CUDA / OpenCL Basics

Submission Guidelines

To pass the course it is obligatory to submit (to Studon) and present your solution of all four assignment sheets to the tutors. The deadline for every assignment is available in StudOn. The deadline applies to both the submission as well as the presentation. Till the deadline the presentation can be given in any computer exercise you like. Submission in groups of three is mandatory. Your solution must always include a appropriate Makefile. Additional requirements may be stated in the tasks.

In this exercise you will write infrastructure code needed to develop CUDA / OpenCL applications. It is recommended to write yourself some utility functions and classes for e.g. device handling, memory management/transfer and kernel launches which you can reuse in later exercises.

This assignment has to be done with CUDA **AND** OpenCL. For the upcoming assignments you can choose whichever suits you best.

In the Casa Huber CIP-Pool, already OpenCL 1.2 headers are installed, while the available platforms identify themselves as OpenCL 1.1. Therefore (and since C++-bindings for OpenCL 1.2 are not yet installed), you should use the (deprecated) OpenCL 1.1 C++-bindings. To do so, you have to define the C-Macro `CL_USE_DEPRECATED_OPENCL_1_1_APIS` before including `CL/cl.hpp`.

1 Querying Devices and their Properties

The first step is to familiarize yourself with the hardware you are working with. Use the CUDA[1] and OpenCL[2] API to get a list of installed devices and their properties. Print the information to the console. An example output can be seen in Listing 1 and Listing 2. Make yourself comfortable with the most important parameters.

```
CL_PLATFORM_VERSION    = OpenCL 1.1 CUDA 4.2.1
CL_PLATFORM_NAME       = NVIDIA CUDA
CL_PLATFORM_VENDOR     = NVIDIA Corporation
CL_PLATFORM_EXTENSIONS = cl_khr_byte_addressable_store cl_khr_icd
    cl_khr_gl_sharing cl_nv_compiler_options cl_nv_device_attribute_query
    cl_nv_pragma_unroll
```

Listing 1: Example output of platform information.

[1] `http://docs.nvidia.com/CUDA/index.html`
[2] `https://www.khronos.org/opencl/resources`

```
CL_DEVICE_EXTENSIONS                    = cl_khr_byte_addressable_store
    cl_khr_icd cl_khr_gl_sharing cl_nv_compiler_options
    cl_nv_device_attribute_query cl_nv_pragma_unroll
    cl_khr_global_int32_base_atomics cl_khr_global_int32_extended_atomics
    cl_khr_local_int32_base_atomics cl_khr_local_int32_extended_atomics
    cl_khr_fp64
CL_DEVICE_GLOBAL_MEM_SIZE               = 1609760768
CL_DEVICE_LOCAL_MEM_SIZE                = 49152
CL_DEVICE_MAX_CLOCK_FREQUENCY           = 1544
CL_DEVICE_MAX_COMPUTE_UNITS             = 16
CL_DEVICE_MAX_CONSTANT_ARGS             = 9
CL_DEVICE_MAX_CONSTANT_BUFFER_SIZE      = 65536
CL_DEVICE_MAX_MEM_ALLOC_SIZE            = 402440192
CL_DEVICE_MAX_PARAMETER_SIZE            = 4352
CL_DEVICE_MAX_WORK_GROUP_SIZE           = 1024
CL_DEVICE_MAX_WORK_ITEM_DIMENSIONS      = 3
CL_DEVICE_MAX_WORK_ITEM_SIZES           = [1024, 1024, 64]
CL_DEVICE_NAME                          = GeForce GTX 580
CL_DEVICE_VENDOR                        = NVIDIA Corporation
CL_DEVICE_VERSION                       = OpenCL 1.1 CUDA
CL_DRIVER_VERSION                       = 295.40
```

Listing 2: Example output of device parameters.

2 Julia Set

The second task of the first assignment is to compute pictures based on the Julia sets[3] for the polynomial $p(z) = z^2 + c$ with $z \in \mathbb{C}$ and a fixed $c \in \mathbb{C}$.

To accomplish this we use the iteration rule $z_{n+1} = z_n^2 + c$. For every pixel we set the starting value z_0 according to its position. The real part of z_0 is given by the x coordinate and the imaginary part is given by the y coordinate. As the most interesting starting points are located in the range between -2 and 2 for both the real as well as the imaginary part you should scale your pixel coordinates accordingly. For every pixel apply the iteration rule a given amount of times (for example 200). To decide the color of the pixel check if the absolute value of the final z exceeds a given threshold (for example 10). Apply one color if this criterion is met and another if z is below the threshold.

To save the picture to disk you can use the freely available lodepng library[4]. It consists of a single header and cpp file which has to be compiled along your other sources. The needed API to write pngs to disk can be seen in Listing 3

An example output can be seen in Figure 1.

Task: Implement this algorithm on the CPU as well as the GPU using CUDA AND OpenCL and provide 3 executables named "juliaCPU", "juliaCUDA" and "juliaCL". All executables should output a picture called "julia.png" with dimensions

[3]`https://en.wikipedia.org/wiki/Julia_set`

[4]`http://lodev.org/lodepng/`

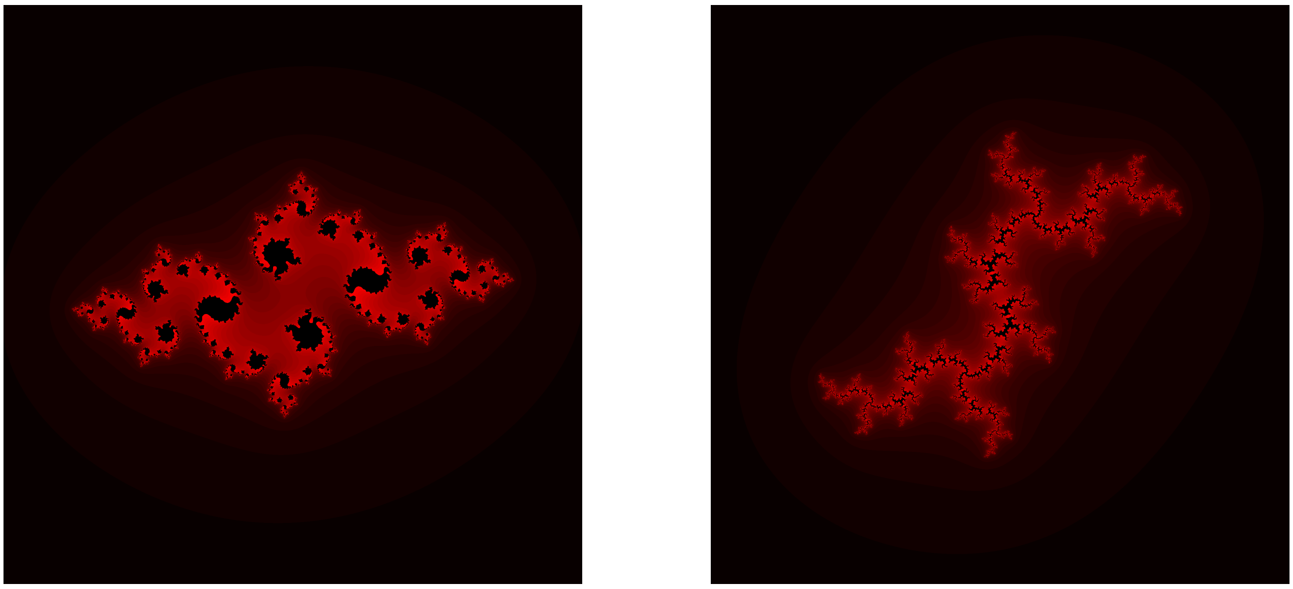

Figure 1: Example output using a red-black colorbar and $c = -0.8 + 0.2i$ (left), $c = 0 + 0.8i$ (right)

of 2048x2048 and be called with 2 additional command line parameters specifying the block/work group size (e.g. "juliaCUDA threadsX threadsY"). For the CPU version you can ignore the two parameters. For the GPU Versions the iteration and the coloring should be done on the GPU. Time the execution of all three versions. Use different block/work group sizes and switch between double and single precision. Prepare a PDF documenting you timings and explaining your observations. Upload the PDF to StudOn.

Optional1: Separate your kernel into three kernels. The first kernel should calculate the number of iterations needed to reach the threshold. The second kernel finds the minimum and maximum number of iterations needed over the whole picture. And the third kernel calculates your final picture. Use the minimal and maximal value obtained with the second kernel to map the number of iterations needed to a colorbar. You do not have to use a linear mapping. Experiment!

Optional2: Change the iteration rule and come up with more interesting pictures.

```
namespace lodepng {
unsigned encode(const std::string& filename,          \\ name of the image
                const unsigned char* in,      \\ buffer storing pixel data
                unsigned w, unsigned h,   \\ width and height of the image
                LodePNGColorType colortype = LCT_RGB, \\pixel data format
                unsigned bitdepth = 8);            \\bits per pixel color
}
```

Listing 3: API of lodepng to save png images to disk.