Exercise 7

Task 7.1 (OpenCL Basics)**.** The following tasks will introduce you into the OpenCL device initialization and the special OpenCL datatypes. For this purpose you will need to work with the OpenCL specification. It can be downloaded at `http://www.khronos.org/registry/cl/specs/opencl-1.2.pdf`. You need NOT to read the whole document. It is only demanded that you look up several library routines.

(1) (a) The first function to work with is *clGetPlatformIDs*. Write a small code snippet that reads the number of platforms in the actual system and prints the result.

(b) Now modify your snippet. Use the result value from the previous task and allocate an array to save all platformIDs. Use *clGetPlatformIDs* to fill your array with data.

(c) Now consider the function *clGetDeviceIDs*. How is it possible to read the device count within a given platform (regarding all types of devices)? Write another code snipped that reads the number of devices for platform number i (for tests you can use platform number 0). Dynamically allocate an array that is capable of holding all device IDs for a given platform.

(d) Modify your code from (c). Use *clGetDeviceIDs* to fill the array from task (c) with device IDs.

(e) The next function to look at is *clGetDeviceInfo*. Write a piece of code that is able to read and print the following information about a given device: device name, maximum compute units, size of local memory.

(f) Now combine all results from the previous tasks: Write an executable code that runs over all platforms and devices of the system and prints the result queried in task (e) for all devices in the system.

(2) As a next step you will learn how to cope with OpenCL's special datatypes as mentioned during the lecture. Here you will work with the *float4* datatype. The task also covers the point of serializing data before it can be transferred to the GPU (i.e. to break down classes into appropriate arrays and values).

Imagine you have a class *Point3D* with four public data members:

- X, Y, Z
- W

The first three variables are the 3D coordinates of the point, while W is always set to 1 (i.e. the datatype is prepared to work for *homogeneous coordinates* as well, but this feature is not used during this exercise). The program you develop must modify the points by exchanging coordinates within them. The coordinates are changed in the following way: $x \to y$, $y \to z$ and $z \to x$. w keeps its value. This is applied to every point of the input data. The amount of points is given by the variable *width*.

Hint: For this task you need not to implement an executable program. But of course if you want you can create one or do it later after you have more experience with OpenCL.

(a) Allocate an array A of floats that contains the complete input data for the kernel.

(b) Convert the input data from an array of *Point3D** pointers called *points* to serialized data in array A (i.e. copy the data in the correct way).

(c) The header of the kernel method is: *__kernel void coordswitch (__global float4* A, __global float4* R)*. Add the body of the method that writes the transformed points into R.

Task 7.2 (Matrix-Matrix-Multiplication (MMM) in OpenCL)**.** The goal of this task is to implement several types of MMM as OpenCL kernels. For this task you should use the code framework 04task_mm.zip.

Hint: The *Executer.cpp* reads on execution a parameter from the command-line to choose the multiplication algorithm. Pass 0 for a the native implementation of task (b), 3 for the line-based algorithm from (c) and 2 for the block-based version from task (d).

(a) For this exercise and all the subtasks you have to configure the *NDRange* on your own. Depending on the kernel you want to start with, the NDRange is either two or one dimensional. The comments within *runCL* indicate where to set the range (starting at line 190 of *04task_mm.cpp*). Set the appropriate parameters for the *clEnqueueNDRangeKernel* function. Reconsider this subtask every time when you implement a new kernel version.

(b) First do the naive implementation. Each Work-item is responsible for calculating one entry of the result matrix C. Use file *MMult.cl*.

(c) Now change the code in the following way: Each Work-item has to calculate one row of the result matrix. Use file *MMult_line.cl*. How does the runtime change?

(d) Now implement the block based algorithm presented on the lecture slides into file *MMult_blocked.cl*. The block-size is 8.

Task 7.3 (Reduction in OpenCL)**.** This task deals with the reduction algorithm introduced in the lecture. Sum is the used reduction operation. Download the OpenCL framework *03task_reduction.zip* from moodle.
For the sake of simplicity we limit the program in several points.

1. Only one compute unit (CU) will do the work in order to avoid the need for global synchronization.

2. The vector length for the input data must always be 256, so that the local size does not exceed the hardware limits.

3. The kernels should place the result in *R[0]*, where R is the result vector that is passed to the kernel.

Hint: The *Executer.cpp* reads on execution a parameter from the command-line to choose the reduction algorithm. Pass 1 for a the serial implementation of task (a), 0 for the tournament reduction from (b) and 3 for the SIMD friendly version from task (d).

(a) Do a very naive implementation of a reduction. Only one thread should do all the work. Use the code skeleton in file *reduction_ser.cl*.

(b) Now realize a tournament reduction as shown on the slides of the lecture. Add the code to the file *reduction.cl*.

(c) Realize the optimization concerning the SIMD friendly reduction algorithm from the slides. Use file *reduction_simd.cl*.

Discussion of this exercise on 03^{th} February 2015.

Bibliography

Khronos OpenCL Working Group; Editor: Aaftab Munshi. The opencl specification. Technical report, Khronos Group, 2013. http://www.khronos.org/registry/cl/specs/opencl-1.2.pdf.