TDT4200 PS 6 - CUDA intro

Exercise based on Ch1 of Anne C. Elster's CUDA Book

Due Monday, 10pm, Oct 28, 2019

Introduction

This Problem Set is mandatory and evaluated as Pass/Fail. I.e. a passing grade is required for taking the final exam.

In this assignment, you will parallelize an algorithm for computing the Mandelbrot set, this time employing the power of GPU's by using CUDA.

CUDA recap

For our purposes, CUDA is an extension of C that allow us to run (part of) our programs on the GPU. Cuda C-files have the extension .cu, and are compiled using "nvcc", Nvidia's own compiler, but otherwise, you can treat it mostly as C-code with a sprinkling of extra syntax.

When writing a CUDA-program, it is important to note that the program will start running on the CPU, which will then start up the GPU by calling a function (called a kernel). This has two important consequences:

1. Functions have labels. "__global__" if it's called by CPU and run on GPU (that's our kernel), and "__host__" or "__device__" if it's only used on the CPU or GPU, respectively.

2. We have to transfer values explicitly between CPU and GPU, because they do not have access to the same memory.

Recipes for allocating memory in GPU and transferring values to/from it, as well as how to construct and call kernels can be found in chapter 1 of the CUDA book on blackboard.

NVIDIA Jetson TX1 (and some TX2) kits

If you do not have a laptop or PC with an NVIDIA card, and find Cybele too crowded, you may borrow one of the NVIDIA Jetson TX1 or TX2 kits we have. You can borrow the Jetson kits from Rolf Harald Dahl in IT-Vest Room 306 from now until the end of the semester.

The Code

The code provided contains a serial CPU-based version of the Mandelbrot algorithm, as well as designated "blanks" for each subtask you will fill in to execute a GPU-based version of the same algorithm. The code can be compiled using the provided Makefile using the target "mandel", and then run by executing the command `./mandel X`, X being 1 or 0 depending on whether you want to save the rendered image.
NB!: Carefully read the section "Deliverables" on how to submit your results!

Subtask 1

In this task you will make your actual kernel. There is a function called `host_calculate()` you can use as a baseline. **Remember:** Any global variables the function uses will need to be provided as arguments, as the GPU does not have these in memory (unless they are explicitly labelled "__device__". Don't do this, however, as it would break the CPU-based algorithm).

When run, the kernel will behave similarly to a MPI-program in that we have to use the ID of our thread to determine where our work is located. These structures will be useful:

- `threadIdx`. The fields `.x` and `.y` contain the coordinates of the thread within the block.
- `blockIdx`. The fields `.x` and `.y` contain the coordinates of the block within the grid.
- `blockDim`. The fields `.x` and `.y` contain the dimensions of the block.

Note: These structures are global variables defined for each thread, so there is no need to instantiate them.

Because the Nvidia GPUs are optimized for very large amounts of threads that can be switched in and out whenever they block (usually because they need to access memory), you should probably write your kernel in such a way that it calculates the value for a single pixel.

Because of the GPU architecture, thread blocks should be launched with sizes that are multiples of 32. This means that you might launch a thread who's coordinates fall outside your image. To keep nasty segmentation faults away, this should be checked before you start calculating the dwell value.

Subtask 2

Use `cudaMalloc` to allocate enough memory for the pixels on the GPU. (**Don't** use the variable `device_pixel`, this is an array on the CPU that will later store the results from your kernel).

Subtask 3

Execute your kernel. You need two `dim3` structs in order to launch it, one for the grid dimensions (in terms of blocks) and one for the block dimensions (in terms of threads). You can use the `BLOCKX` and `BLOCKY` as x and y dimensions of your blocks, they make sure the resulting thread count is a multiple of 32.

Remember to pass all the needed arguments, the pointer you allocated in subtask 2 and any other variables you deemed necessary in Subtask 1.

Subtask 4

Now it is time to get the result back from the GPU. Use `cudaMemcpy` to transfer the values from the array pointed to by your GPU-pointer (Subtask 2) to the array named `device_pixel`.

Subtask 5

Use `cudaFree` to free the memory allocated in Subtask 2.

Subtask 6 (Optional)

Time the serial version and the CUDA version.

How much faster is the CUDA version – or is it slower?

You can also compare the times on those systems, and look into downloading the CUDA SDK (See Chapter 1).

Deliverables

The documentation must be delivered as a PDF file together with the source code packed in a ZIP file and submitted in time on blackboard. The code must be documented and in a runnable state. Do not use any libraries apart from standard libraries or those provided. Do not change the calling conventions (parameters) of the application.

NB!: Deliver your code in the folder structure that is provided to you. Include at least every file that you have changed. Your application must stay compatible with the provided Makefiles or CMake files. **Do not include any object, input or output files**.

The following tasks require documentation (if solved):

- 1a, 1b, 2b, 3, 4, 5