TDT4200 PS 7 - CUDA

Zawadi Berg Svela

Due Tuesday, Nov 12, 10pm, 2019

Introduction

This Problem Set is mandatory and will be graded, counting for 10% of your final grade. A passing grade is required for taking the final exam.

In this assignment, you will repeat what you did in assignment 3, parallelizing the simple edge detection algorithm, but this time using CUDA. You will also explore the "shared memory" location of the GPU in order to speed up the processing even further.

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 edge detection algorithm. The provided Makefile will compile the program. For a description of the algorithm, see the text for assignment 3. The handed-out code is mostly the same as in assignment 3, but all the filters are now called "filters" rather than "kernels" to reduce confusion.
NB!: Carefully read the section "Deliverables" on how to submit your results!

Task 1a - 20%

The serial code reads the BMP image and saves as an array of pixels. You must allocate enough space for both the final image and the temporary working image on the GPU, and copy the data from the CPU. As with assignment 3, you can use the array-of-arrays variable "data" or the contiguous array "rawData" to access the pixels.

Important: Check for errors! Most CPU-run functions of the CUDA API returns a `cudaError_t` variable to indicate whether the operation was successful or if it produced an error. The program will continue running even if an error occurs, so you should check these return values and verify that all is fine, and probably print a message to your console if it isn't. See this page for how to extract the contents of the error. You can wrap any API call you make in the `checkError()` macro, which will exit the program and print error details if it detects any non-successful result.

Task 1b – 20%

Create your kernel. Copy and rename the `applyFilter()` function and adjust it to work on the GPU. Keep in mind that because threadblocks are generally multiples of 32, and some threads might work on pixels outside the image. You should insert boundary checks to make sure you are not reading or writing outside your allocated memory.

Task 1c – 20%

Launch your kernel. You can choose whether to divide the image row-wise or in blocks, but as mentioned, the block size should be a multiple of 32. Your kernel needs to be launched one time per iteration. You also need to copy the results back from the GPU.

Tip: The function `cudaGetLastError()` can be used to catch any errors from launching the kernel. Also, you might want to keep the CPU code in the for-loop. This will enable you to run both the GPU and CPU versions when testing, and compare the results to make sure everything is correct.

Task 1d – 10%

Time your new program and note the speedup versus the serial version.

Task 1e – 20%

Implement another kernel using shared memory and measure speedup compared to both the CPU version and the "basic" GPU version.

Task 2 (only do this if you have time) – 15%

Even with shared memory, every pixel is read at least once every iteration. CUDA 9 introduced cooperative groups that can be used to synchronize threads across thread blocks. More information on their use here. For an example of use, which includes probing the GPU for information about the number of blocks that can be run on each SM, see this sample from Nvidia.

Note that the whole cooperative group has to be able to be run simultaneous across the SMS. The main limitations for our application is register usage and shared memory since **in order not to synchronize with global memory, the picture manipulated needs to fit in the SM's shared memory.** Note that there is a large difference in no. of SMs in, say NVIDIA TX2 vs GTX 1080 (20SMs) vs Tesla V100 (80SMs).

a) (5%) Outline how cooperative groups can be used to improve your code to avoid synchronization via the CPU between steps, and how to handle the arising boarder issue, including noting what data needs to be shared and updated for each iteration.

b) (10% – Yes, possible total is 105% :-))
Implement your solution and describe how architecture-specific it is, including what is need to run it on a given architecture. Measure the speedup.

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):

- 1d, 1e, 2a, 2b