TDT4200 PS 5 - CUDA - Image Filtering

Jacob Odgård Tørring, Zawadi Berg Svela

Due Friday, Nov 6, 22:00, 2020

NOTE Re. grading of this assignment & course

This Problem Set is mandatory and will be graded, counting for 15% of your final grade. A passing grade is required for taking the final exam. Your overall grade in this course may be converted to Pass/Fail depending on the faculty's recommendation and how the situation develops with respect to COVID-19.

Introduction

In this assignment, you will repeat what you did in assignment 2, but this time using CUDA. You will also explore optimizations for the GPU in order to speed up the processing even further.

Access to hardware: Snotra cluster

If you do not have a laptop or PC with an NVIDIA card, you can log onto the Snotra cluster and get an NVIDIA GPU allocated:
ssh username@snotra.idi.ntnu.no.

See the previous posts for Cybele and Snotra under "Course Zoom URLs & information" on Blackboard for more tips on how to use this interface.

If you end up on a host called "minip" you need to run **connect**. This shell is upon for 2 hours before it automatically closes. You can reconnect when the shell closes if you need more time. You can also request a specific node by specifying the name of the node:
ssh -t username@snotra.idi.ntnu.no connect nodename

Current list of nodes (subject to change):

selbu: 20 T4s

1 GTX1080 each:

clab02, clab04, clab06, clab09, clab12,
clab15, clab18, clab21, clab23, clab25

The Code

The code provided contains a serial CPU-based version of the image filters. 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 1 - 20%

Each step is 5%.

1. Start by allocating device-side arrays of pixels and copying the host-side arrays over. You must allocate enough space for both the final image and the temporary working image on the GPU. As with assignment 2, you can use the array-of-arrays variable ”data” or the contiguous array ”rawdata” to access the pixels. We recommend using the contiguous array ”rawdata” (see the serial example for indexing examples).
2. Copy and rename the `applyFilter()` function to be a CUDA kernel.
3. Initialize and start the timer just before the kernel call and end the timer right after.
4. Launch it with 1 threadblock and 1 thread as a serial implementation on the GPU.

Check that it produces the correct results.

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.

You can also debug using cuda-gdb, cuda-memcheck or nvprof.

Task 2 – 20%

Parallelize the kernel. Keep in mind that because threadblocks are generally multiples of 32, and some threads might work on pixels outside the image. You should therefore insert boundary checks to make sure you are not reading or writing outside your allocated memory.

Task 3 – 10%

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.

Task 4 – 10%

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

Task 5 – 20%

The GPU's SMs has what is called shared memory, effectively programmable cache. In your current version of your program, you are reading every cell in the filter a number of times equal to the amount of threads, and probably reading every pixel at least nine times from global memory (because the smallest filters are 3x3 cells large). A variable prefixed by `__shared__` will be shared among all threads in a block. If you construct a shared array (e.g. `__shared__ int myarr[20]`, you could start your kernel by reading the filter-values and pixels you need into such arrays and then use those values in your computation. Because the warps in a block can generally execute in any order, use `__syncthreads()` to make sure all threads in a block have loaded their pixel before you continue. Also, keep in mind what you do with the threads at the "edges" of your block. Shared memory example form Nvidia can be found here: link.

Notes and hints: *You may experience a slowdown, and wonder why. Some hints: Shared memory is about 10 faster than global memory, but how many reads are we saving and how many are we adding from different memory locations using the current block dimensions? For a block to be active/"alive", we need to have room for all the registers that its threads are using. as well as enough shared memory. Having lots of blocks active at the same time generally increases performance (this concept is called "occupancy"), however, how does this new solution affect this? What could we do to minimize these effects? Try experimenting with your block dimensions and see how they affect performance.*

Note: We do not expect you to specifically answer the above questions in the document you turn in, but rather added them as hints to help you understand what is happening.

Task 6 – 20%

Use the CUDA Occupancy API to calculate the optimal block sizes. How high is your occupancy? What can you do to improve this? Compare performance before and after. See Nvidia's tutorial on how to use the API here.

Deliverables

The following tasks require documentation (if solved):

- 4, 5, 6

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.

*Note!: 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**.