Assignment 4

Due Apr 19 by 10pm **Points** 10

Assignment 4 - GPUs and CUDA

Due: Monday, April 19, at 10 p.m.

Important note

You will be using the computing cluster provided by teaching labs for all GPU labs and assignments **instead of SciNet**. The detailed information about the platform can be found in this **link (https://www.teach.cs.toronto.edu/using_cdf/cluster.html)** . Similar to SciNet, you still **need to use the provided job scripts** to run code on the compute nodes with GPU cards.

There are two types of partitions for running code:
csc367-debug: the debug partition used for debugging code contains 16 corals; limited to 10 minutes of real-time per job.
csc367-compute: the compute partition used for measuring the performance of code contains 14 prawns; limited to 30 minutes real-time per job.
Notes: the result should be reported based on the setting in which the partition is csc367-compute.
The default setting for the type of partition is csc367-compute. You must modify it to csc367-debug if you need to debug the code.

Deadline to find your group partner and create your group on MarkUs: March 29th, 10pm. If possible, try to work with your A2 partner.

Overview

For this assignment, you will work with CUDA to design efficient code that leverages the parallel processing power of GPUs to do some image processing. In order to efficiently solve a problem and fully leverage the potential of GPUs, you have to consider everything we discussed in lectures and labs, to design a solution in the GPU model. See the **CUDA (https://docs.nvidia.com/cuda/pdf/CUDA_C_Programming_Guide.pdf)** documentation guide for more information.

We recommend that you work with your A2 partner. Please log into MarkUs as soon as possible to find your repository and invite your partner, and make sure that you can commit and push to your repo. For all assignments and labs you will be given your repo URL on MarkUs. Make sure to use this repository (which should already be created for you), otherwise, MarkUs won't know about it and we won't be able to see your work.

The starter code is available on wolf (the department's teach/CDF machines) under **/u/csc367h/winter/pub/assignments/a4/starter_code.tgz** so copy this into your repository and make sure you can do your first commit. Please make sure to read carefully over the code, including the licenses and the instructions from the comments.

Your task

In this assignment, you will implement a discrete Laplacian filter kernel on the GPU, using CUDA. You may choose any of the filters described in A2, but the filter dimension should be at least 3x3. **Your implementations must support all different filter sizes used in A2. You must also investigate the performance behaviors of the different filters in your report.**

Your code must efficiently process the image in parallel, and take advantage of the GPU's massive parallelism and memory bandwidth. Additionally, if you have not already in a previous assignment, you must implement a CPU version which also processes the image in parallel, using POSIX threads. You should use the fastest CPU implementation for a fair comparison. Make sure you tweak your A2 implementation if needed. Remember that CDF machines have a different number of cores and you **must** stick to **4 cores**.

Your code will be assessed based on efficiency/performance, so you must make sure to take advantage of all the things we discussed in class. You are welcome to perform additional optimizations, as long as they are supported by the capabilities of the cards from the GPU cluster. For the following examples, assume we have a 4x4 pixel matrix and threads T0, T1, ... Your code should contain at least 5 GPU implementations:

Kernel 1 - One pixel per thread, column-major. In this case, the work is distributed as follows:

```
T0: pixel[0][0]
T1: pixel[1][0]
T2: pixel[2][0]
T3: pixel[3][0]
T4: pixel[0][1]
T5: pixel[1][1], etc.
```

Kernel 2 - One pixel per thread, row-major. In this case, the work is distributed as follows:

```
T0: pixel[0][0]
T1: pixel[0][1]
T2: pixel[0][2]
T3: pixel[0][3]
T4: pixel[1][0]
T5: pixel[1][1], etc.
```

Kernel 3 - Multiple pixels per thread, consecutive rows, row-major. Assuming you have, for example, 2 threads:

```
T0: pixel[0][0] pixel[0][1] pixel[0][2] pixel[0][3] pixel[1][0] pixel[1][1] ... pixel[1][3].
T1: pixel[2][0] ... pixel[3][3]
```

Kernel 4 - Multiple pixels per thread, sequential access with a stride equal to the number of threads. Assuming you have 3 threads, then:

```
T0: pixel[0][0], pixel[0][3], pixel[1][2], pixel[2][1], pixel[3][0], pixel[3][3]
T1: pixel[0][1], pixel[1][0], pixel[1][3], pixel[2][2], pixel[3][1]
T2: pixel[0][2], pixel[1][1], pixel[2][0], pixel[2][3], pixel[3][2]
```

The way you choose how many threads and blocks to run might be different from kernel to kernel.

Kernel 5: Your fifth implementation should be consistently faster than all of the 4 above by at least 15%. That is, for example, if the best implementation among {1,2,3,4} takes 1s to run, your fifth implementation should take no more than 0.85s on most runs.

You should first implement a version of this problem without normalization. Once you have that thoroughly tested, you should implement the normalization step. The reasoning for this is that a technique called reduction might be necessary for that step, which might only be covered on lectures that proceed the assignment release date.

Recall that CUDA can only synchronize threads in the same block; to achieve synchronization across blocks, your kernel must be split into two. Thus, computing the max/min value of the image probably can't be done with a single kernel, and it is your job to find a way to do it. For instance, you could have all blocks write two values (one for max and one for min) into global arrays and then perform a reduction on the global array to compute the global max/min. Notice that this will also involve some reduction within each block, as discussed in the lectures. Alternatively, you could perform the reduction in CPU, which involves copying the global array from device memory to host memory. In this case, you should include both the CPU time to compute min/max and the memory transfer time in the appropriate variables. How the max/min values are computed is up to you, but the normalization should follow the same memory access patterns outlined above (1-4).

The PGM format

You may read more about the PGM specification **here** (http://netpbm.sourceforge.net/doc/pgm.html). You may use an image processing program of your choice to create such PGM images for testing. We're also providing you with code that is capable of reading pgm images from files, saving pgm images to files and generating pgm images.

We are providing two images in the I*mages* folder in the starter code. Your report should only include results for those two images. But your code should work for various images since the autograder will generate all kinds of images and use them to test your implementation for both correctness and performance.

Measuring the performance of your code

You must measure the performance of your kernel and transfer times. Your measurements should include separate timings for the transfer times to/from the GPU, and calculate the speedup of pure GPU computation against the CPU version: `CPU_time/GPU_time`, and the speedup of total time taken to compute on the GPU (including transfers in and out!) against the CPU implementation: `CPU_time / (GPU_time + TransferIn + TransferOut)`.

Your output should be in the following format:

```
        CPU_time(ms) Kernel  GPU_time(ms) TransferIn(ms) TransferOut(ms) Speedup_noTrf Speedup
               2.00       1        0.50           0.25            0.25         4.0      2.0
        ...
```

The timings and speedups above are solely for illustration purposes of the output formatting.

The provided starter code already satisfies this requirement.

Your program must not print anything other than what the starter code prints.

Running `make` on your directory should produce your solution as an executable called "solution.out".

IMPORTANT: Your tables and report should be only for the two provided images in the I*mages* folder in the stater code. You should tabulate your data to be more readable and clear.

Program input

Your program should receive two strings as arguments: "-i input_image_filename -o output_image_filename". Other than printing the timings above, your program should also output 1 file per kernel, using the following naming convention: "kernel_number"+"output_image_filename". So if your program is invoked with "./solution.out -i input.pgm -o output.pgm", the following files should be output: "1output.pgm", "2output.pgm", "3output.pgm", "4output.pgm" and "5output.pgm". Your program should

also output the file generated by the cpu implementation, with the name passed as the "output_image_filename" argument. Each of your kernels should be in a separate .cu file. We have provided a suggested header and .cu files in the starter code. You do not have to use them, but each kernel should be in its own appropriately named file. When compiling kernels separately, you might need to use the "--device-c" flag. We have provided an illustrative Makefile.

GPU Environment

You can check the node information by clicking the provided link, or you can add the command 'nvidia-smi' into the job script (job-a4.sh) to check the configuration of the Nvidia GPU card.

The path to the nvcc compiler can be found in the provided Makefile.

Testing

1. Modify your optimized CPU code from HW2 to work for **4 cores.**
2. A script called *run-job-a4.sh* is provided that builds and runs your code on a compute node with GPU card**.**
3. For each experiment, you must take the average of 5 runs and report this average in the report.

Submission

You will submit the code files which contain your implementation, along with the files required to build your program.
Do not submit executables, object files, or image files!
Do not submit any other subdirectories in your A4 repository, as these may prevent the autotester from running correctly and you will lose marks.

Aside from your code, you must submit the report documenting your implementation, presenting the results, and discussing your findings. Your report should include a discussion on how you chose the number of threads and blocks on each implementation, an explanation of how you computed the min/max values and a description of your most optimized kernel. When discussing your approach, feel free to also describe any problems encountered and workarounds, what isn't fully implemented (or doesn't work fully), any special design decisions you've taken or optimizations you made (as long as they conform to the assignment specs!), etc.

Additionally, you must submit an `INFO.txt` file, which contains as the first 2 lines the following:

- your name(s)
- your UtorID(s)

If you want us to grade an earlier revision of your assignment for whatever reason (for example, for saving some grace tokens if you had a stable submission before the deadline, tried to add new

functionality after the deadline but broke your submission irreparably), then you may specify the git hash for the earlier revision you want to be marked.
As a general rule, by default we will always take the last revision before the deadline (or last one after the deadline, up to your remaining unused grace tokens), so you should **not** be including a line with the git commit hash, except in the exceptional circumstances where it makes sense. So in general, please avoid using this option and just make sure that the last revision (either before the deadline if you submit on time, or up to a subset of your remaining grace tokens if you submit late) is the one you want to be graded.

Finally, you **must** submit a `plagiarism.txt` file, with the following statement:
"All members of this group reviewed all the code being submitted and have a good understanding of it. All members of this group declare that no code other than their own has been submitted. We both acknowledge that not understanding our own work will result in a zero on this assignment, and that if the code is detected to be plagiarised, severe academic penalties will be applied when the case is brought forward to the Dean of Arts and Science."

A missing INFO.txt file will result in a 10% deduction (on top of an inherent penalty if we do not end up grading the revision you expect). **Any missing code files or Makefile will result in a 0 on this assignment**! Please reserve enough time before the deadline to ensure correct submission of your files. No remark requests will be addressed due to an incomplete or incorrect submission!

Again, make sure your code compiles without any errors or warnings.
Code that does not compile will receive zero marks!

Marking scheme

We will be marking based on correctness, coding style, fulfilling the speedup requirements (when applicable). As with A2, **we will be checking your code for many corner cases,** and your code is expected to work with different images. Remember that your handout has to only report numbers for the two provided images, however, your code should work for others.

Make sure to write legible code, properly indented, and to include comments where appropriate (excessive comments are just as bad as not providing enough comments). Code structure and clarity will be marked strictly!
Once again: code that does not compile will receive 0 marks!

More details on the marking scheme:

- 14% for the correctness of each kernel (70% total)
- 20% for the report
- 5% for the correctness of the CPU implementation
- Code style and organization: 5% - code design/organization (modularity, code readability, reasonable variable names, avoid code duplication, appropriate comments where necessary, proper indentation and spacing, etc.)

- **Negative deductions (please be careful about these!):**
  - **Code does not compile: -100%** for *any* mistake, for example: missing source file necessary for building your code (including Makefile, header files, etc.), typos, any compilation error, etc
  - **No** `plagiarism.txt` **file: -100%** (we will assume that your code is plagiarised and that you wish to withdraw your submission, if this file is missing)
  - **Missing or incorrect** `INFO.txt`**: -10%**
  - **Warnings: -10%**
  - **Extra output: -20%** (for any output other than what is required in the handout)
  - **Code placed in other subdirectories than indicated: -20%**