Assignment 4 - GPUs and CUDA

Due: Dec 5, at 10 p.m.

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.

You may work with a partner on this assignment. Please log into MarkUs as soon as possible to find your repository and invite your partner (should you choose to have one), 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 cslinux under /student/cslec/367/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.

Keep in mind that most courses have assignments due in the last week of classes, so make sure to start early on this assignment!

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 (suggestion: try different filter sizes, what do you observe?). You choice of filter will likely affect how to implement your kernel 5 (see below).

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 fix your A2 implementation if needed.

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:

1. - 1 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.
   ```

2. - 1 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.
```

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]
```

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.

Your fifth implementation should be consistently faster than all of the 4 above by at least 10%. That is, for a big image (say at least 10 MB), if the best implementation among {1,2,3,4} takes 1s to run, your fifth implementation should take no more than .9s on most runs. To do that, you can either find a better partitioning scheme, add more code optimizations, memory access optimizations, memory transfer optimizations or exploit filter properties (recall you only have to implement one of the filters!). Feel free to try any combination of the abovem, or something entirely different, but explain your rationale. We will accept 10% improvement on the gpu run time, that is, your 10% does not need to include the transfer times. If you go for memory transfer optimizations, we will also accept 10% on the total time.

Note: 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 gradual approach, is that a technique called **reduction** will be necessary for the normalization step and we will only cover how to perform a reduction *effectively* on a GPU in future lectures. Once we cover that in class, you're good to go to do the normalization.

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 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. If you apply this strategy, include the CPU reduction time in the GPU_time variable, as it should be negligible. 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.

Measuring the performance of your code

You must measure 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.

Program input

Your program should receive two strings as arguments: "-i input_image_filename -o output_image_filename". This is already satisfied by the starter code.

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 "./main -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 header and .cu files in the starter code to help you with code organization.

Compiling your code

We have provided a CMakeLists.txt file that has everything required to compile your code. The following steps will produce the required binaries.

1. Make sure you are in the source directory.
2. mkdir build
3. cd build
4. cmake ../ -DCMAKE_BUILD_TYPE=Release
5. make

The Release flag is important! It sets the optimization level to O3, an important step before you measure the performance of your code. If you want to debug using gdb or any tools that require debug information, replace "Release" with "Debug". **Do not measure performance using Debug mode!**

To recompile your code, you only need to run the "make" command inside the build directory, you don't need to repeat the other steps.

The procedure above will produce the following binaries inside the build directory: main, test_solution, pgm_creator.

1. main, which works as described previously.
2. pgm_creator, which works the same as in A2.
3. test_solution, described below.

You don't need to modify the CMakeLists.txt file. If you do, it is your job to ensure that the procedure above works and produces the required binaries. When in doubt, ask us on Piazza!

The ideal submission will change only these files:

1. main.cu, look for the TODO string.
2. best_cpu.cu, put the code for your best CPU implementation here.
3. kernel1.cu (2,3,4,5), implement the 3 functions in those files.
4. kernels.h, you only need to change the signature of the kernel5 functions.
5. test_solution.cu, described below.

Testing your code

Producing fast programs is important, but performance is not useful without correctness. Therefore, we have provided you with a set of minimal automated tests that can be used to check if your code is correct.

This step is optional and does not guarantee that you will get a perfect correctness score, but it will prevent you from making the most common mistakes. If you submit code that does not pass these tests, you are guaranteed to not get a perfect score.

The tests are run by executing `./test_solution`. Inspect the file `tests.cu`, we will now describe the tests it contains.

The function `compare_kernel_against_handwritten_example` defines a simple 2x2 image, whose pixels are 0, 1, 2, 3. It runs one of the kernels, are compares its result to that of the expected answer (also hardcoded in the test). The line `ASSERT_EQ` asserts that the kernel produced the correct pixel value; when an error is found, the assertion will stop the test and print the invalid pixel.

The `TEST(csc367a4, some_name)` lines register a new test to be run when `./test_solution` is executed. Note that we register 6 tests that call the function above, one per kernel and one for the CPU implementation.

Now look at the test

compare_kernel_against_each_other

. What does it do?

Feel free to add any extra testing in this file. If you do, mention it briefly on the report and we will consider your testing efforts when grading the correctness of your code.

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.
Do not submit the "Starter Code" directory.

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 exaplanation 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 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 graded.

Finally, whether you work individually or in pairs with a partner, 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 and use of features from the GPU architecture in kernel 5 (don't try to use artificial tricks to make your kernel 5 look better). As with A2, **we will be checking your code for many corner cases,** and your code is expected to work with images as big as 150MB (pgm file size).

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:

- 10% for the correctness of each kernel (50% total)
- 10% for achieving the desired speedup with kernel 5
- 10% for ideas used in implementing kernel 5
- 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%**