Final Project

In this project, you will implement a CUDA program aimed at finding the connected components in an image (Connected Component Labeling (CCL) problem). The CCL problem (which is a particular case of the more general problem of finding Strongly Connected Components (SCC) in a graph) is of particular importance, especially in the fields of image processing and computer vision. Therefore, being able to find time-efficient solutions is important and that will be your goal for the project.

Honor Code Rules
The CCL problem has been studied quite extensively. For this project, you are **not allowed** to copy any piece of code from someone else (other students, code found on the web or in publications, etc). This will be considered a violation of the Honor Code. All code that you use in your paper must be your own (with the exception of the code provided by the instructor). As a precaution, we recommend that you do not look at anyone else's code.

However, one of the goals of the project is for you to learn how to read the literature on CUDA and be able to use that information to implement your algorithms. Therefore, you **are allowed** to search the web, publications, presentations, written reports, etc, for information. You must cite every piece of work that you use in the preparation of your paper.

Schedule and grading

Date	Percent	Description
June 7 11:59PM	20%	One page report
June 12 8:30AM	80%	4-page report or oral presentation

The one page report will be used to determine whether you will give an oral presentation on June 12 or whether you need to turn in a 4-page report (these are exclusive). For full mark, this one page report should:

- Include code that performs the calculation correctly (17%)
- Describe how you are planning to optimize your code, in particular using shared memory and thread collaboration (3%)

For the final report or oral presentation, your grade will be determined as follows:

- 55%: Algorithm 1: unoptimized CUDA code that runs correctly. This code should not use shared memory.
- 17%: Algorithm 2: CUDA code that runs correctly and uses shared memory (and other thread-block optimization). You should demonstrate a performance improvement over Algorithm 1.
- Write-up or oral presentation. You will be graded on:
  - 2% Overall quality of report: technical correctness, clarity of the explanations, and conciseness
  - 2% Discussion of performance: bottlenecks, use of hardware resources by the code
  - 2% Data analysis: quality of plots and benchmark data
  - 2% Literature discussion: discussion/presentation of the literature surveyed and material used to prepare the solution

Connected component labeling problem and references

Refer to Fig. 3 below for a quick view of the problem.

You will need to solve the Connected Component Labeling Problem (CCL). This problem is for general graphs but for this project we are going to consider only graphs built from images. A graph is defined as a set of vertices, here the pixels of the image, and a set of edges.

For this project, an edge can only exist between neighboring pixels. The definition of neighbor varies in the literature but we will use the 4-connectivity definition. See Fig. 1.

We consider that two pixels are connected by an edge when their color is the same. We simplified the problem further by considering images with only two colors: black and white.

In summary, two pixels are connected by an edge if:

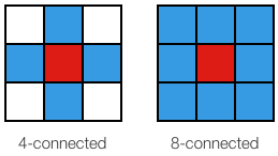


Figure 1: 4-connectivity and 8-connectivity. In red, the pixel of interest. In blue, its neighbors for the chosen connectivity.

- They are neighbors.
- Their color (can only be black or white) is the same.

What is a connected component? This is standard concept in graph theory. A connected component is a set of pixels such that (more formal definitions are possible but this will do for this project):

- If there is an edge from pixel a to b, then a and b must belong to the same connected component.
- If two pixels a and b are in the same connected component, then there must be a path $(p_1, \ldots, p_n)$ such that $p_1 = a$, $p_n = b$ and for any i, the pixel p_i is connected to p_{i+1} by an edge.

See Fig. 2.

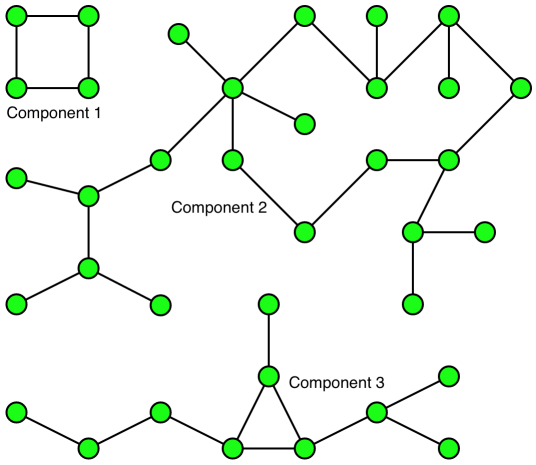


Figure 2: General graph showing 3 connected components.

The CCL problem is then to:

- Identify all the connected components.
- Assign a unique label to each connected component.
- Assign to each pixel its unique connected component label.

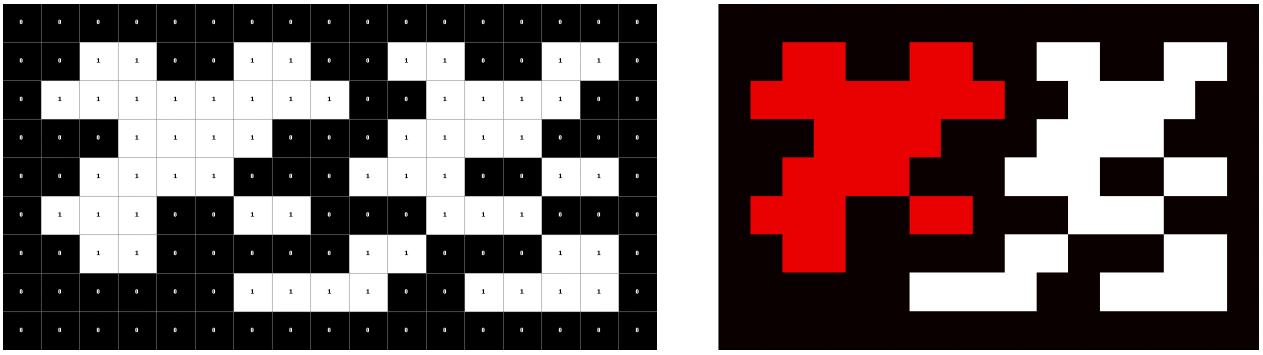
Here are the input and output data your code should produce:

- Input: a binary image (0=black, 1=white).
- Output: a data structure containing for each pixel the label of its connected component.

The precise labeling of the connected components does not matter as long as each connected component has a unique label.

Figure 3 shows an example of the problem with its solution. The labels have been replaced by colors for the output.

The wikipedia page explains various sequential algorithms with pseudo-code. The main reference we recommend for the project is [1]. You should refer to the bibliography at the end of this paper for more references. You may also find it useful to look at the website of the GTC Conference (Ziegler and Ceska's presentations) for years 2010 and 2013.



(a) Input (b) Output

Figure 3: Connected Component Labeling of a binary image. In the image on the left we have only two pixel colors, shown with (tiny) labels 0 (black) or 1 (white). 8-connectivity was assumed to get the figure on the right. Each connected component is identified by a unique label (integer) which is then mapped to a unique color for display (right panel).

Starter code

Your executable should be called `cudaccl`. The starter code is composed of the following files:

- `img/`: this folder contains the test images.
- `inputs/`: this folder contains the text files for the images in `img/`. These are files that can be used as input to your program.
- `results/`: this folder contains the images produced by the reference CPU algorithm.
- `main.cpp`: this file contains a serial implementation of the algorithm. You do not need to modify this file. This is the reference solution. Study this algorithm carefully to make sure you correctly understand the calculation to perform with CUDA.
- `directions.h`: this file contains the inline functions used to find the indexes of the neighboring pixels. For the CUDA code, you will need to copy this file and create similar functions that work with your CUDA code.
- `imghelpers.h` and `imghelpers.cpp`: header and source files that contain the helper functions for image processing. You should look at `imghelpers.h` to get an idea of the functions in the interface you can use in your program. You should use this file to read/write images (`ReadImgFromFile` and `WriteImgToFile`). To compare your output (the file produced by `cudaccl`) to the reference solution (the given CPU code), you should call `CompareSolution`; this function takes as arguments the names (include the correct path if the files are in a different directory) of the files to compare.
- `processCuda.sh`: this shell script: 1) takes as input a text files with the bit information for the input image (these files are given to you); 2) runs your code, which produces a text file with the labels; 3) converts this text file into a JPEG image.
  The script takes two arguments, the second one being optional:

  ```
  ./processCuda.sh inputFile [ouputImgName]
  ```

  If you do not specify the second argument, the image will be written to `result.jpg`. By default, the intermediate file containing your labels is not deleted (the file is called `out`). If you want to keep your directory clean and delete this file, uncomment the last line in `processCuda.sh`.
- `createinput.py`: this Python script converts an image into a text file with the bits associated with each pixel (0 for black and 1 for white). The text file can then be easily read by the C++ code (using `ReadImgFromFile`). Please refer to the comments in the file for the usage of this script. We will give you images already converted to this text format. However you may want to use this script for other images, for testing purposes for example.

- `createimg.py`: this Python script transforms the text file with the labels into an image; this allows visualizing your results. It takes as input a file formatted in the same way as the output of `WriteImgToFile`. Please refer to the comments in the file for the usage of this script. The shell script that we are giving you, `./processCuda.sh` will call this script for you.
- `helpers.py`: this file contains helper functions needed by the two other scripts.
- `colors.py`: this file chooses appropriate colors in the RGB spectrum to be associated with each label in your output. This script is needed for the other scripts.

Contrary to the programming assignments, we do not provide you with any CUDA files. You will need to turn in *all* the CUDA files that the graders need to compile and run your project.

Format of the files
If you want to build some custom examples (small examples for testing for instance), you will need to produce infiles that have the correct format. The format `ReadImgFromFile` uses is the following:

```
xLen
yLen
0 1 ... 1
.
.
.
```

Namely, the first (resp. second) line corresponds to the width (resp. height) of the image. Then, each line corresponds to a row of the image and the pixels (0 or 1 only) are whitespace separated. Here is an example of valid input file:

```
6
4
0 0 1 1 1 0
0 0 1 1 0 0
1 1 0 0 0 1
1 1 0 0 0 1
```

The output files (produced by `WriteImgToFile`) use the same format.

Your tasks

Your executable will take two arguments: the input image file (a text file formatted as in `inputs/`) and the output file name. Your program should output a file named after that second argument. For instance, when your program is called as follows:

```
./cudaccl inputs/stanford.in stanford.out
```

your program should output a file named `stanford.out`.

Your program should print out the time your kernel(s) took to run (see for example the output format of PA3) and the bandwidth associated with it.

If you implement multiple kernels (with different levels of optimization), you should do the following: run all kernels in your code and provide timing information for each. Then, use the result of the fastest kernel to produce a single output file. We are not expecting you to produce a separate output file for each kernel.

Part 1

Your first goal is to come up with an implementation of a kernel that outputs the correct labeling. Your kernel should *not* use shared memory at this point. Kernel A in [1] can give you a good idea of what is expected here but you do not need to specifically implement this kernel.

Part 2

For the second part of the project, you will need to implement a second kernel that uses shared memory and other thread-block optimizations. You should observe a speed-up between the kernel you implemented in part 1 and this kernel.

A good reference for this task is Kernel B in [1], but again, this may not be the best implementation you can come up with and you are totally free to implement a kernel of your choice.

Do not forget to write tests for all your codes. This time, they are not provided and making sure that your code runs correctly will be an important part of the project. One useful strategy is to write small hand-crafted tests; then increase the size and the complexity of the tests.

Deliverables

You must turn in a zip file named `final.zip`. The file should contain a single directory named `LastName_FirstName_Final` with:

- One or several source files and a Makefile. To run your code, the grader will run the following commands:
  - `make`
  - `./cudaccl inputImgFile outputLabelFile`

  Note that this is important that your executable be named `cudaccl` because the scripts we will run will call this particular executable.
- (Optional) A plain text file named `readme` that explains the eventual specifics of your code.
- For the write-up, you should submit a pdf file named `writeup.pdf`.

References

[1] K. A. Hawick, A. Leist, and D. P. Playne. Parallel graph component labelling with gpus and cuda. *Parallel Comput.*, 36(12):655–678, December 2010.

[2] Oleksandr Kalentev, Abha Rai, Stefan Kemnitz, and Ralf Schneider. Research note: Connected component labeling on a 2d grid using cuda. *J. Parallel Distrib. Comput.*, 71(4):615–620, April 2011.

[3] Victor M. A. Oliveira and Roberto A. Lotufo. A study on connected components labeling algorithms using gpus.

[4] Jiri Barnat, Petr Bauch, Lubos Brim, and Milan Ceska. Computing strongly connected components in parallel on cuda. In *IPDPS*, pages 544–555, 2011.