CS 33, Spring 2023

Lab 4: Parallel Lab

Due Date: June 9, 2023 11:59 PM

Disha Zambani (dmzambani@g.ucla.edu) is the creator and lead for this lab.

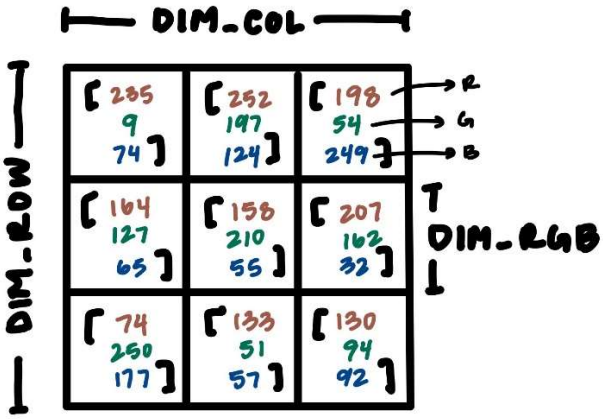
1. Introduction

Welcome to parallel lab! This is the last lab of CS 33 and is quite different from earlier labs. The purpose of this lab is to introduce you to optimization and parallelization on somewhat relevant tasks in the real world. It is split up into three phases in which you perform a different image manipulation task. The first phase deals with calculating an average pixel value across all pixels in an image. The second phase deals with converting an image into grayscale as well as recording the minimum and maximum grayscale values in the image. The third phase deals with blurring an image via an operation called convolution. You will not have to come up with algorithms on your own, sequential solutions for these manipulations are provided. Your task is to speed up and parallelize these algorithms. In addition, you are given buggy parallelized code for phases 1 and 2 which need to be fixed.

2. Images

This lab deals with parallelization of image manipulations. I really wanted to include a visual component to this lab, but unfortunately it introduced a lot of complexity. Instead, I have statically generated random images for you in the `long original_img[DIM_ROW][DIM_COL][DIM_RGB]` and `long padded_original_img[DIM_ROW+PAD][DIM_COL+PAD][DIM_RGB]` arrays. The dimensions of these arrays are provided in `utils.h`. Please read the next section, Developing Your Solutions, for more information on this file.

An image is structured as a 2D array with each element being an array of 3 elements. The individual 3 element arrays emulate pixels with RGB values in that order. RGB values range from 0 to 255.



3. Developing Your Solutions

Download the parallel_lab.zip file from BruinLearn, unzip it and get it onto SeasNet. You can work on this lab locally but your submissions will be tested submissions on SeasNet, so make sure your solutions work as intended there. The files are as follows:

```
main.c
Makefile
sequential_phase1.c
parallel_phase1.c
```

`sequential_phase2.c`
`parallel_phase2.c`
`sequential_phase3.c`
`parallel_phase3.c`
`utils.h`

Feel free to start developing your solutions on any of the phases, you do not need to work on this lab in sequential order. Each phase has 2 files, one containing the sequential solution and the other containing a possibly buggy parallelized solution. The sequential solution is guaranteed to give correct results. You are expected to fix any buggy parallel code and speed it up to the best of your ability.

A main function testing your solutions is given to you. It tests each phase in order and lets you know if your parallel solution is correct. Additionally, a speedup of your parallel solution from the sequential solution is also produced. This value may change across runs and you will be graded on the best speedup out of some number of runs. See the grading section of this spec for more. Be aware of the types and allocations of the arrays. These design choices are intentional and may become pitfalls if you choose to alter them in your code.

You are also provided with a `utils.h` file which contains some constants that you are free to play around with. There are constants that define the dimensions of the original image, kernel size, and padding of an image as well as a normalization factor for phase 3 based on the kernel size chosen. See the Phase 5 section of this spec for more information. A good idea would be to start out with testing smaller dimensions for your images to ensure that your parallel solution generates correct results. Then to test for a good speedup, increase the dimensions of the images. For more testing, sample functions to print your images are given in `main.c`. Feel free to introduce your own auxiliary functions for your personal testing. Your code will be tested against something similar to the main function provided.

To compile the lab run the following command in the parallel lab directory:
`make`
Note: if you change utils.h, run make clean before make.

To clean the project (get rid of object files) run the following command in the parallel lab directory:
`make clean`

To run the lab run the following command in the parallel lab directory:
`./Test`

Take this lab at face value, all optimizations you are expected to implement are things we have covered in class, discussions, the homeworks, and the LA worksheets and workshops. Think about program optimizations, exploiting instruction level parallelism, writing cache friendly code, and using OpenMP pragmas.

4. Phase 1

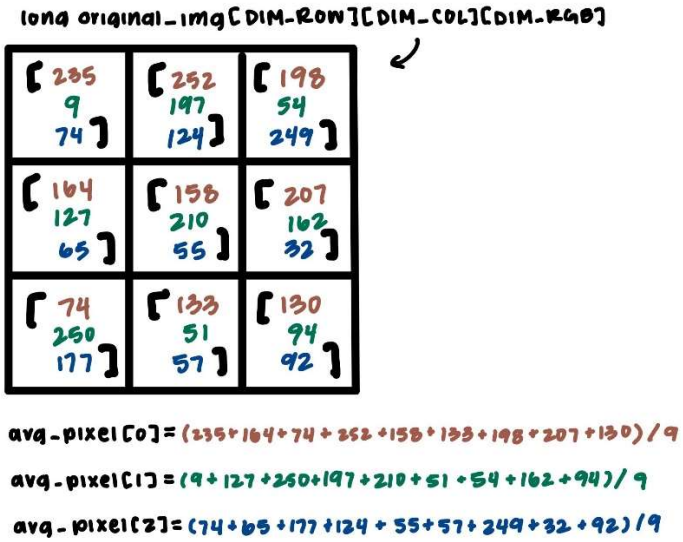
This phase implements a solution for calculating the average pixel value across all pixels in an image.

`sequential_phase1.c`

This file has a sequential solution for the pixel averaging algorithm. It takes in a `long original_img[DIM_ROW][DIM_COL][DIM_RGB]` and a `long *avgs` of length `DIM_RGB`, which is intended to be modified. The algorithm does the following:

- Adds up all R values in `original_img` and divide by the number of pixels in the image and stores the result in `avgs[0]`.

- Adds up all G values in `original_img` and divide by the number of pixels in the image and stores the result in `avgs[1]`.
- Adds up all B values in `original_img` and divide by the number of pixels in the image and stores the result in `avgs[2]`.



`parallel_phase1.c`

This file contains buggy parallelized code for the pixel averaging algorithm. You are expected to identify the bug and speed up the given code.

`main.c`

Main will dynamically allocate `long *sequential_avgs` and `long *parallel_avgs` of length `DIM_RGB` to be passed into the sequential and parallel solutions respectively. They are to be modified in the solutions and checked after execution for correctness. The solutions are timed and a speedup is calculated.

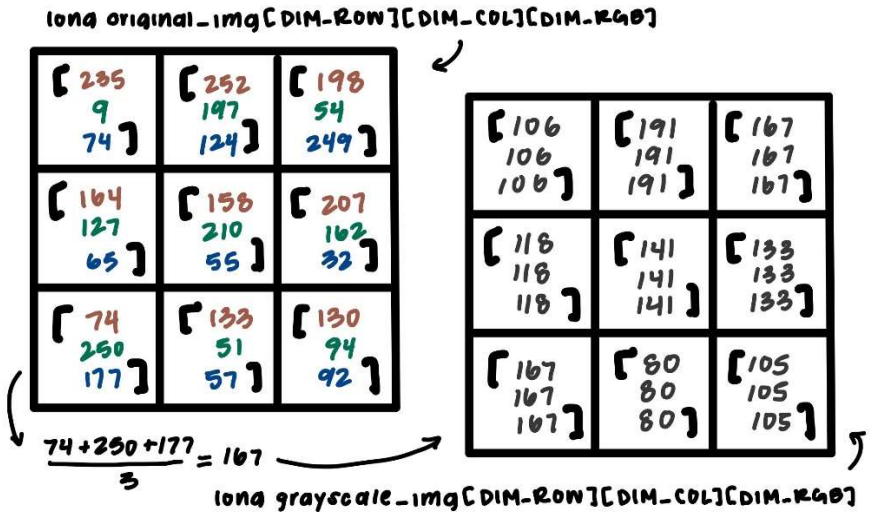
5. Phase 2

This phase implements a solution for calculating and modifying an image with its corresponding grayscale values. It also keeps track of the minimum and maximum grayscale values in the new image. The grayscale value of a pixel is calculated by taking the sum of its R, G, and B values divided by 3. Then each R, G, and B value is set to the calculated grayscale value. This is done for each pixel.

`sequential_phase2.c`

This file has a sequential solution for the grayscale algorithm. It takes in a `long original_img[DIM_ROW][DIM_COL][DIM_RGB]`, `long ***grayscale_img` of dimensions `[DIM_ROW][DIM_COL][DIM_RGB]`, and `long *min_max_gray` of length 2 as input where `grayscale_img` and `min_max_gray` are intended to be modified. The algorithm does the following:

- For each pixel in `original_img`, calculate the grayscale value and set the corresponding pixel in `grayscale_img` to the calculated value.
- If the grayscale value calculated is smaller than a previously recorded minimum grayscale value, update `min_gray` and update `min_max_gray[0]` with the min grayscale value.
- If the grayscale value calculated is smaller than a previously recorded maximum grayscale value, update `max_gray` and update `min_max_gray[1]` with the max grayscale value.



`parallel_phase2.c`

This file contains buggy parallelized code for the grayscale algorithm. You are expected to identify the bug and speed up the given code.

`main.c`

Main will dynamically allocate `long ***sequential_modified_img` and `long ***parallel_modified_img` of dimensions `[DIM_ROW][DIM_COL][DIM_RGB]`, and `long *sequential_min_max_gray` and `long *parallel_min_max_gray` of length 2 to be passed into the sequential and parallel solutions respectively. They are to be modified in the solutions and checked after execution for correctness. The solutions are timed and a speedup is calculated.

6. Phase 3

This phase implements a solution for applying a convolution kernel over a padded version of the original image to generate a convoluted image. We don't expect most of you to know what a convolution looks like or what the previous sentence meant. Convolution is hard to describe in words. Here are a couple resources to understand the convolution operation on images:

- https://medium.com/@bdhuma/6-basic-things-to-know-about-convolution-daef5e1bc411
- https://www.youtube.com/watch?v=YgtModJ-4cw

For our purposes, I have given two kernels for gaussian blur for you to test your convolution solutions on. They are 3x3 and 5x5 kernel sizes, each with a different normalization factor. Make sure to adjust utils.h to make sure you are testing with the correct kernel constants. We are not looking into changing the stride for the convolution, it remains at 1. For the resources given, the medium article shows a padded image operated upon during convolution and the output image will be the size of the original unpadded image. The video resource does not account for this padding we are using but is a better resource to see how a convolution operation is calculated. The padding is added to ensure a simpler convolution calculation. The image on the next page has a sample calculation per RGB value of each pixel. Notice how the kernel is applied to each RGB value individually and is stored that way in the convoluted image.

`sequential_phase3.c`

This file has a sequential solution for the convolution algorithm. It takes in a `long padded_original_img[DIM_ROW+PAD][DIM_COL+PADS][DIM_RGB]`, `long kernel[DIM_KERNEL][DIM_KERNEL]`, and `long ***convolved_img` of dimensions `[DIM_ROW][DIM_COL][DIM_RGB]` as input where `convolved_img` is intended to be modified. The algorithm does the following:

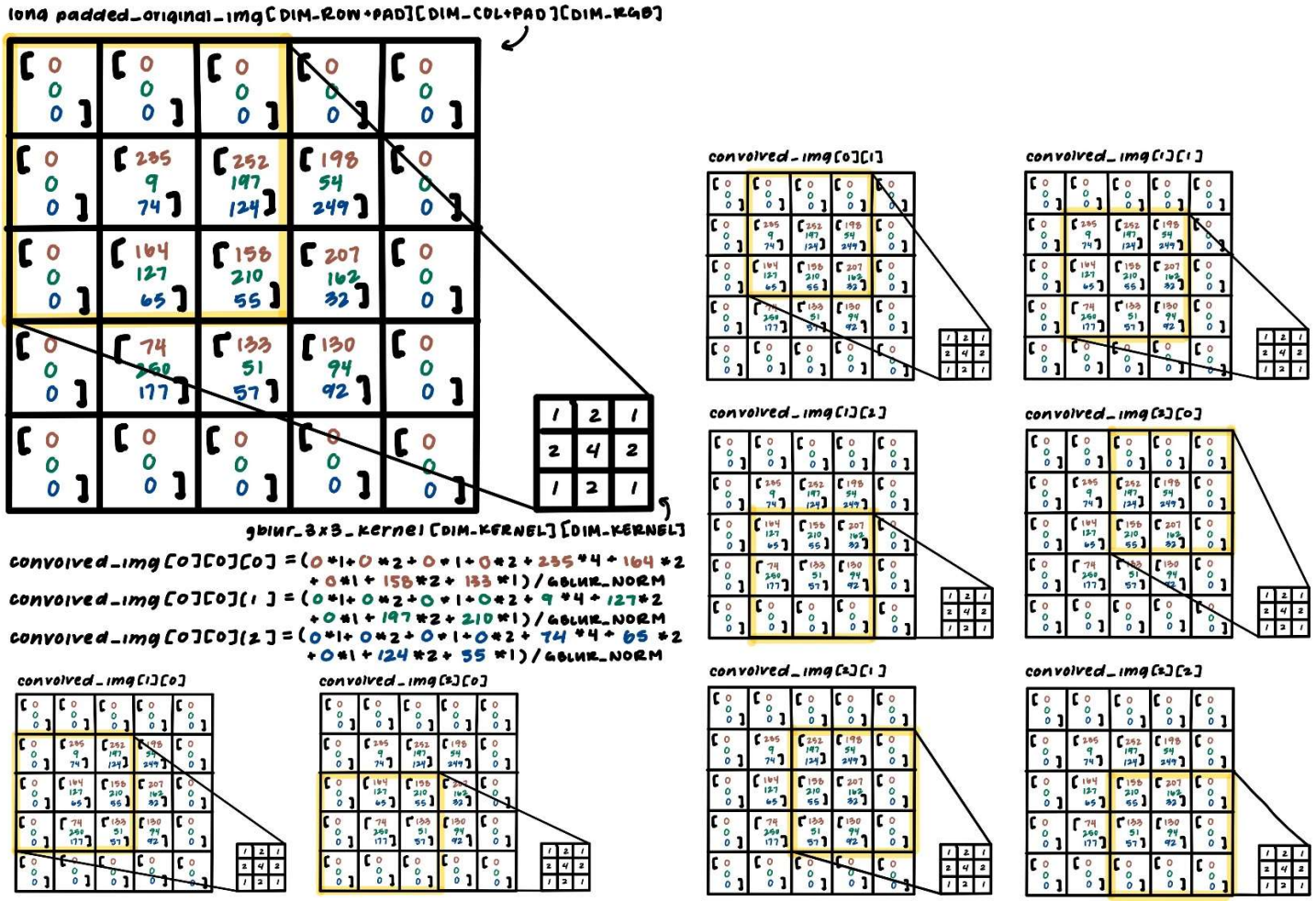
- For each RGB value in each pixel in `padded_original_img`, take convolve the kernel with the submatrix of `padded_original_img` starting with pixel as `[0][0]` and has width and height of `DIM_KERNEL`.
- Normalize the convolved RGB values by the appropriate gaussian blur normalization factor and update `convolved_img`.

`parallel_phase3.c`

This file contains the sequential code for the convolution algorithm. You are expected to speed up the given code.

`main.c`

Main will zero out `long ***sequential_modified_img` and `long ***parallel_modified_img` of dimensions `[DIM_ROW][DIM_COL][DIM_RGB` to be passed into the sequential and parallel solutions respectively. They are to be modified in the solutions and checked after execution for correctness. The solutions are timed and a speedup is calculated.



7. Turning In Your Solutions

Make sure that your solutions produce the intended results with the main function given to you. On BruinLearn, submit the following files in a zip.

```
parallel_phase1.c
parallel_phase2.c
parallel_phase3.c
```

8. Grading

The whole lab will be out of 100 points. Each phase is worth a total of 30 points. For phase 1, 20 points are given for correctness and 10 for speedup. For phase 2, 15 points will be given for correctness and 15 for speedup. For phase 3, 10 points will be given for correctness and 20 for speedup. Speedup will be graded based on a threshold (around 4-5x) for each phase. Extra credit up to 10 points will be considered for significant speedup on any of the phases. The final 10 points will be given if you fill out a survey detailing some of your thoughts on this lab. I made this lab as my capstone project for my Master's degree. It would help me out a lot if you could fill out the survey so I can get some tangible results from this project. It will be posted on BruinLearn closer to the lab due date.