Project 2

A brief description of the project. A paragraph or two recap will suffice?
In this project, we use Kernels to blur/sharpen/detect edges in the images. We use average values of r, g and b and using it as the value for each r, g and b to create black and white images.
For **sequential** implementation, each effect is applied to each image one at a time and the value is stored temporarily for that image if multiple effects are to be applied on that image
For **pipelined** implementation, each effect was applied one at a time, but to multiple images at a time and within each image, to multiple strips of the image at a time
For **BSP**, each effect was applied to one image at a time, but to multiple parts of the same image.

Instructions on how to run your testing script. We should be able to just say `sbatch benchmark-proj2.sh`; however, if we need to do another step then please let us know in the report.
To run the script, following command is required:
sbatch benchmark.sh
This needs to run from the proj2 folder.

What are the hotspots and bottlenecks in your sequential program?
In the sequential program, there are many bottlenecks/hotspots. We work on one image at a time and apply only one effect at a time. This can be avoided by running a parallel code.
Also, for kernels, we are repeating computations of kernel multiplication with the image pixels. This can be avoided by storing the values in some other array and reusing them.

Which parallel implementation is performing better? Why do you think it is?
The pipelined version is performing better. This is because, in pipelined implementation, each thread is spawning its own threads and those threads are working on the images. However, in the bsp implementation, one image is getting worked upon at a time by several threads. Since we are not working on multiple images, the performance is not as good as pipelined version.

Does the problem size (i.e., the data size) affect performance?
Yes, the problem size does affect the performance. In an ideal world scenario, the bigger the problem size, the better than speed up. This is because, the overhead costs remain the same, and more part of the program gets parallelized.

The Go runtime scheduler uses an `N:M` scheduler. However, how would the performance measurements be different if it used a `1:1` or `N:1` scheduler?

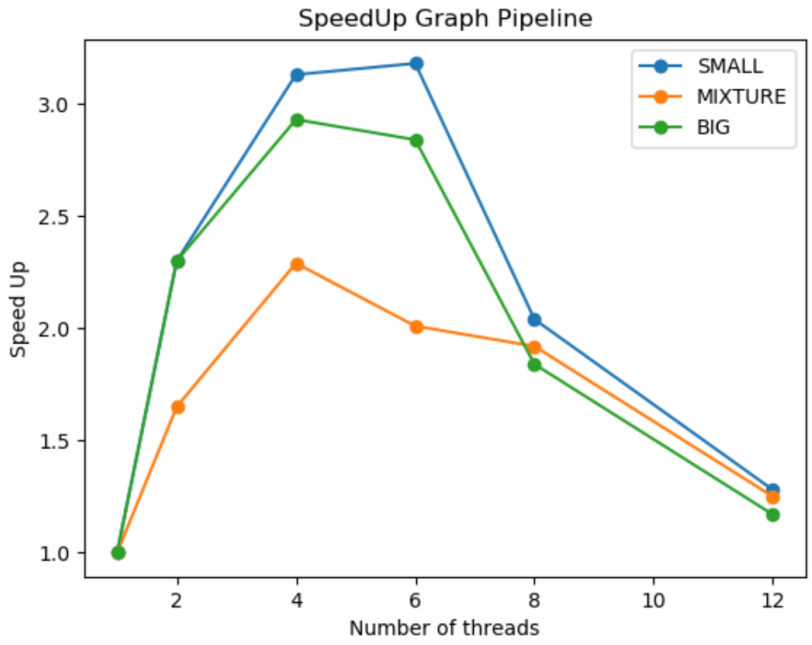
Yes, the performance measurements would be different on 1:1 and N:1 scheduler.

N:1 – The N:1 scheduler is basically each thread waiting to be run sequentially hence, its performance wouldn't be any better than sequential. In fact, it might even be worse than sequential because of the overhead cost associated with spawning threads.

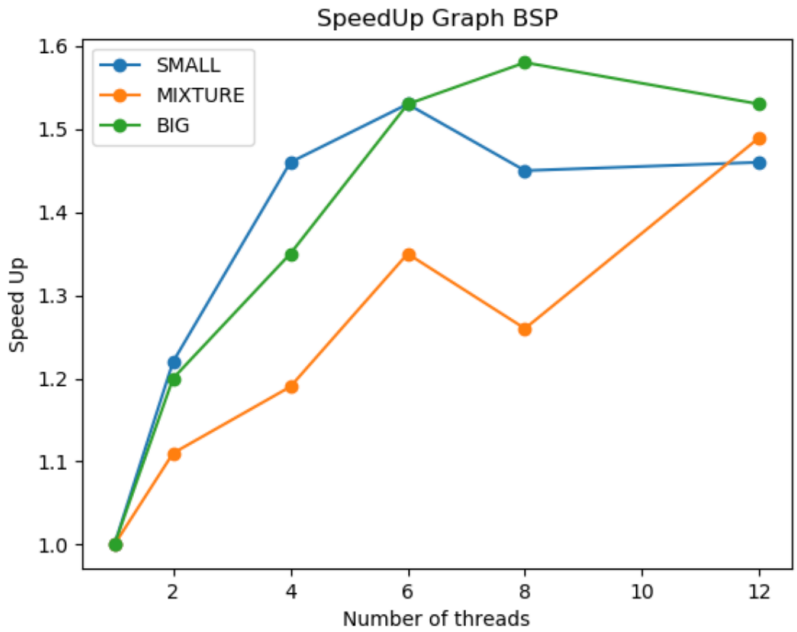
1:1 – The 1:1 scheduler might also perform worse, because all threads spawning involve kernel and they might end up interfering with the OS. Hence, it is always wise to use N:M Scheduler.

Based on the topics we discussed in class, identify the areas in your implementation that could hypothetically see increases in performance (if any). Explain why you would see those increases.

We can improve the performance by work balancing/work stealing algorithms. That is, whenever a thread gets free before its peers, it will try and steal their work, so that the whole program finishes earlier than it was supposed to. We can try to do this by calling say a 100 pixels a task and each thread given a task to complete and whenever a thread is idle, it can steal work from its peers and finish it to reduce the completion time.



In this graph, we get speed up till we increase the threads to a certain point post which, we start seeing a dip because the overhead cost associated with spawning threads starts outweighing the time reduced in parallelizing the code by the threads.



In this graph, we see that the speed up curve increases gradually with the number of threads. The speed up is not too much because all the threads work on the same image and the cost of spawning the threads is too much. Also, since there are locks associated with this implementation, the length of the serial part increases, resulting in slowing down of the process.