GPU Programming Basics

Northeastern University | *for all* | Julian Gutierrez
NUCAR Laboratory | Nicolas Agostini, David Kaeli

Hands-on Lab #5

LAB #5 - Objective

- Calculate the best operating point for a simple image processing algorithm.
- Implementing a noise removing algorithm.
- Study different approaches for a convolutional filter on an image and its performance impact.

First of all, you need to open 2 terminals and connect them to discovery cluster. Please enable X forwarding (**`ssh -X`** `<username>@login.discovery.neu.edu`) so that you can execute commands that display graphics).

- Terminal 1: this terminal will be used to connected to an interactive compilation node.
- Terminal 2: this terminal will be used to schedule the gpu jobs, it will always be connected to the login node.

In Terminal 1, to request an interactive node for code development and compilation, we can use the following command. Once a resource is available, you will be connected automatically into the compute node. If you are having trouble with this, go back to Lab 2 and review part 1.

```
# TYPE this command, do not copy and paste

srun --pty --export=ALL --partition=short --tasks-per-node 1 --nodes 1
--mem=4Gb --time=02:00:00 /bin/bash # Reserve the compilation node
```

Copy the following folder to your scratch directory (or folder of your choosing):

```
cd /scratch/$USER/GPUClassS19
cp -r /scratch/gutierrez.jul/GPUClassS19/HOL5/ .
cd HOL5/
```

Part 1: Brightness

In this lab, we will be modifying images. To visualize the images, we need a program that can view them. I assume all of you either use MobaXterm to connect to the server or have a terminal directly on your computer with **ssh -X** capabilities (MobaXterm supports X forwarding by default).

We can visualize the results in two different ways. The most complicated one is by running the code, and once the code has generated the outputs, we can use FileZilla or MobaXterm to copy those files back to our personal computer so that we can open them there.

The second way to do this is with the MobaXterm terminal connection X forwarding support, so we can open the images directly in the terminal using different programs. Sadly, I haven't been able to get the X forwarding to work from the compute nodes which means you will need to open 2 tabs, one where you are using the compute node to compile and launch the experiments, and another one that is connected to the login nodes where you can use it to open the images.

You can test if you can do this by running the following command from the main directory of the lab and using the login node directly (it might take a while to open the image depending on your connection):

```
display input/SaPnoise.png
```

All the files needed for this part can be found in the following folder on the server:

```
cd ./brightness/
```

Read the following files and understand how they work and answer the questions.

- Makefile.
  - Helps automate the process of compiling the code for the project.
  - What could you add in this file that might improve the performance of your code? Look at how the files are compiled.
- src/config.h
  - This file oversees the definition of the different variables used in the execution of the code.
  - What changes in this file could significantly affect the performance of the algorithm?
- src/main.cpp
  - Main function that calls the function to execute the algorithm on the gpu.
  - Why do we run a warmup function?
  - Extra: How do we read the input image and write the output image?
- src/brightness.cu
  - This is the function that controls the GPU and executes the kernel.
  - Look at how the kernel is executed. Grid size, block size, kernel code.
  - Understand how the brightness function and the kernel work.

To be able to execute this code, as you might have noticed, we are using the OpenCV library. This means we need to load the module to support OpenCV by running the following command or adding it to your .bashrc file (please do this to make sure your code executes correctly on the compute node):

```
module load opencv/3.4.3-contrib
```

The first time you are compiling use the following command before running the application. It will compile with the DEBUG flag enabled, which will enable the checkCuda function to look for errors.

```
make debug
```

Once you've made sure the output is correct, recompile the code by running the following commands. These commands will compile with the optimization flags:

```
make clean; make all
```

To run the code, submit the sbatch script. The script we will use is the one in the folder named exec.bash. Look at what it does and update the fields **reservation** and **gres** to use the resources available to our class (you will have to do it for every .bash file in this class):

```
vim exec.bash

#SBATCH --reservation=GPU-CLASS-SP20

#SBATCH --gres=gpu:1
```

Now you can use the command:

```
sbatch exec.bash
```

It will produce 3 output images called:

```
input.jpg
```

```
output_cpu.jpg
```

```
output_gpu.jpg
```

You can visualize them using your terminal connected to the login node (if using MobaXterm) by using the display command or you can copy these files with FileZilla to your computer. For example:

```
display input.jpg &
```

To compare the input and output, open both. Do you see any differences?

You can change the brightness level by specifying a different <value> in the second argument in the line of the exec.bash script which executes the program. Try writing a negative value. Remember pixel values go from 0 to 255. This is an example of the input arguments to the program (Don't run it directly on the terminal).

```
./brightness ../input/fractal.pgm -100
```

Resubmit the script.

For this part of the lab, your job is to find which block size performs best by modifying it on the `src/config.h` file. Test it with the `fractal.pgm` image and the `world.pgm` separately. NOTE: Sometimes execution time might vary a lot, so doing a couple of test runs and averaging the result is encouraged (you can execute the same command multiple times inside the `exec.bash` script and then average the results).

Note: Every time you change the block size you must recompile the code using make:

```
make clean; make all
```

- Which block size is the best one for each input? **Please note**, we are looking at kernel execution time, not GPU execution time.

Block Size	**Kernel Execution Time (fractal)**	**Kernel Execution Time (world)**
2		
4		
8		
16		
32		

- Could these values be different if the size of the image varies significantly?
- How do you think it affects the performance?
- Why do we use only block sizes with 2^n form?
- What optimization could be done to improve this code?

Run nvprof the same way we did for the second lab. Use the best configuration for the block size based on the smallest kernel execution time for the fractal image. MAKE SURE YOU COMPILE THE CODE AGAIN. NOTE: Also remember that we are running a warmup code which executes the same instructions. You can remove this by setting the WARMUP flag to 0 in the config file (Do it for this test). Use the nvprof.bash script.

```
sbatch nvprof.bash
```

Look at the percentages and decide, is it worth optimizing this code? If yes, where would you optimize?

Now let’s look at some of the metrics by looking at the output from the nvprof script where we measured the metrics. You can identify this output with the line METRICS.

Open the output file and look at the results, is there anywhere you think we could further improve the kernel?

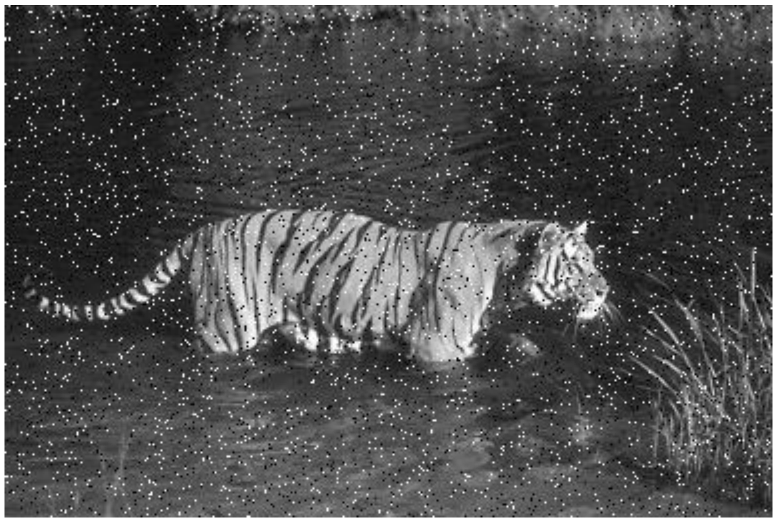
Extra: ONLY IF YOU FINISHED THE REMAINING PARTS OF THE LAB. Test one of your ideas to improve the performance. Did it work?

Part 2: Noise Removing Algorithm

All the files needed for this part can be found in the following folder on the server:

```
cd ./noise/
```

This part will be a bit more complicated. As it is well known, it is common to observe noise in images (E.g. noise coming from the camera, environment, etc). A common type of noise that images suffer from (sometimes due to data corruption) is known as salt and pepper. These appear in the image as bright (salt) and dark (pepper) pixels randomly around the image. We are going to focus on trying to reduce this type of noise in an image by applying a denoising algorithm. The following image is an example of an image with salt and pepper noise.



A popular approach to remove this type of noise is by looking at neighboring pixels (imagine a mask of 3x3) and reordering the values of these pixels in ascending order. Finally, the output value of our algorithm will be the middle value in that array. Here is a quick example:

200	198	198
0	195	196
199	255	199

We rearrange the values in ascending order:

0	195	196	198	198	199	199	200	255

The output for the pixels we are analyzing will be the center value of this array which is 198 in this case.

0	195	196	198	198	199	199	200	255

As you can see from these results, the salt and pepper noise (255 and 0 respectively) will be added to the beginning and the end of this array, so they are usually filtered out.

To be able to apply this algorithm we will have to sort an array on the GPU. We will do this by using a simple sorting algorithm known as bubble sort. This algorithm works by repeatedly swapping adjacent elements that are in the wrong order. If you don't know how it works, they explain it in detail in the following webpage:

https://www.geeksforgeeks.org/bubble-sort/

Follow these steps in order to implement this algorithm on the GPU. You can use the CPU implementation as a guideline as well.

1. The file we need to modify is denoise.cu so open it with vim or some other text editor.
2. The first thing we will do is load the values around the pixel the thread is processing. We will be using a 3x3 mask. In line 65 of the file, write the size of the array we will need. Note: Use a one-dimensional array.
3. Now, we need to bring all those values from global memory into our local array (notice we are not using shared memory. That would improve the performance of this algorithm, but we are focusing on getting the algorithm to work). Implement the necessary code to populate the "values" array with the data we need. You can do this manually, with for loops or whatever you would like.
4. Once we have populated the local array, we are going to sort it using the bubble_sort function (notice its definition is done using __device__ meaning this function can only be called from the device). The second argument to this function is the size of the array, specify the same value you used in 2.
5. Before we write the bubble_sort algorithm, let's make sure we write the correct value to the output. In the output line, write the index for the element of the array we want to use as the output.
6. Now let's focus on the implementation of the bubble sort function.
   a. Based on the definition of this algorithm, we iterate through the array N*(N-1) times to make sure the array is ordered correctly. Implement this code in the device function defined in line 43 by using the following code snippet as a guideline (taken from the geeks for geeks page).

```
// A function to implement bubble sort
void bubbleSort(int arr[], int n)
{
   int i, j;
   for (i = 0; i < n-1; i++)

        // Last i elements are already in place
        for (j = 0; j < n-i-1; j++)
            if (arr[j] > arr[j+1])
               swap(&arr[j], &arr[j+1]);
}
```

Once you have implemented all the above changes to the code compile it using make (remember to use debug the first time, once it works, compile with the optimization flags) and execute it using the sbatch script. This will run the denoise code on the 5perSaPnoise input image (the tiger one). Use any means you prefer to open the files and look at the outputs.

```
display input.jpg &
```

```
display output_gpu.jpg &
```

Visually inspecting the output will give a better idea if it worked or not. You can also run the measure_noise code providing the input image and running it again providing the output image to compare the estimated noise for each (this is already done in the exec.bash script):

```
g++ -O3 src/measure_noise.cpp `pkg-config opencv –cflags –libs` -o
measure_noise
```

```
./measure_noise input.jpg
```

```
./measure_noise output_gpu.jpg
```

If you got good results, rerun the sbatch script changing the input image to SaPnoise.png. Answer the following questions:

- How does this algorithm reduce the noise for this type of image?
- Is the object in the image clearer after denoising the image?
- Was it able to remove all the noise? If not, what could avoid this from fixing all the problems?
- Try running the algorithm multiple times in a row (by passing the output image of the first run into a second run) And so on. Do this 2 times, or 4 or 8. How does the output look?
- What would happen if we apply this image to an image without any noise at all?
- How could we improve the performance of this algorithm? If there is still time, try and implement it.
- Would increasing the size of the mask help? How can you implement this based on the code you have written?

Note: Please remember to ask as many questions as possible. I am here to help as much as we can.

Part 3: Extra: Sobel Algorithm (helpful for final project)

As shown in class, Sobel algorithm is an edge detection algorithm. It applies a convolution filter on the image and outputs another image highlighting the pixels that are located on borders based on a certain threshold.

In this part, we are going to analyze different implementations of the same algorithm and compare their performance. The structure of the folders is identical to the brightness algorithm we saw on the previous part. Compilation and execution are the same as well.

To compile the codes run the following command inside their respective folders:

```
make all
```

To run the code, use the exec.bash scripts which execute the following command:

```
./sobel ../../input/fractal.pgm 100
```

It will produce the same outputs as the previous codes.

We have 5 different versions of the code:

1. Basic
2. Opt1
3. Opt2
4. Opt3
5. Opt4

"Basic" is a naïve implementation of the Sobel algorithm (we saw in class). The other versions have different improvements done to the code to try and achieve better performance. The idea is to fill out the following table with the execution times by running each code on both inputs (no need to change the block size, though this might give you an idea).

Version	**Kernel Execution Time (fractal)**	**Total Execution Time (fractal)**	**Kernel Execution Time (world)**	**Total Execution Time (world)**
Basic				
Opt1				
Opt2				
Opt3				
Opt4				

- Note: To understand what they're doing, look at the sobel.cu files inside the src folder.

After recollecting the data, answer these questions:

- Explain what each optimization version does.
- Which one works best and why?
- Which one is the easiest to implement?
- Extra: Choose one of the optimized versions and run nvprof. What metrics change compared to the basic implementation?
- Extra: Change the algorithm by changing the filter to something else you would like and don't use a threshold for the value. How does the output look?