

CSCD 445/545 GPU Computing Homework 2 (Total 100 points)

Simple Image Processing On CUDA Device

Description

In this project, we manipulate PGM images (Portable Gray Map). PGM files could be pure ASCII text files, with header information and intensity (brightness) value for each pixel in an image file. If you have an image named `ballon.pgm`, you can use the Linux command `less ballon.pgm` to explore its format. Of course, you can install an image viewer to visualize the image with your naked eye. On windows, you can use **Irfanview**, download is available at <http://www.irfanview.com>

On a Mac machine, you can download **ToyViwer** in your **apple store for free**.

The detailed format description for PGM file can be found here. (Also you can download more sample PGM files, besides one included in this project package.)

<http://people.sc.fsu.edu/~jburkardt/data/pgma/pgma.html>

The following is an example PGM file named smallFile.pgm

```
P2
# feep.ascii.pgm
24 7
15
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 3 3 3 3 0 0 7 7 7 7 0 0 11 11 11 11 0 0 15 15 15 15 0
0 3 0 0 0 0 0 7 0 0 0 0 0 11 0 0 0 0 0 15 0 0 15 0
0 3 3 3 0 0 0 7 7 7 0 0 0 11 11 11 0 0 0 15 15 15 15 0
0 3 0 0 0 0 0 7 0 0 0 0 0 11 0 0 0 0 0 15 0 0 0 0
0 3 0 0 0 0 0 7 7 7 7 0 0 11 11 11 11 0 0 0 15 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
```

The image header consists of the first 4 lines.

1. The first line **P2**, is a magic number to tell the image viewer that the file is a ASCII pgm file.
2. The second line starts with **#**, is a line of comment.
3. The numbers in the third line means this image has 24 **COLUMNS**, and 7 **ROWS** of pixels in it.
Note: column goes first!
4. The fourth line means the maximum intensity value in the image is 15. Each pixel has intensity value. **The bigger intensity value is, the brighter or whiter at that location is in the image.**
The intensity value 0 means a total black color in the image.

Intensity values of all pixels are listed following the file header, starting with 5th line. These intensity values could be considered as a 2D array, with the **row index** and **column index** specifying each point (or pixel) at a particular location.

What you should do?

Please carefully read the comments on top of each function declared in pgmUtility.h file.

- 1) In pgmUtility.c (or you can renamed it as pgmUtility.cu if needed.) file, implement the function that reads in the image (**actually a text file**) using file I/O. The function has been declared already in pgmUtility.h file. Note this function will **not** be called on GPU device.
`int ** pgmRead(char **header, int *numRows, int *numCols, FILE *in);`

You **probably have to change** to the following signature if you **linearize** the 2D pixel matrix into 1D storage.

```
int * pgmRead( char **header, int *numRows, int *numCols, FILE *in );
```

- 2) In pgmUtility.c file (or you can renamed it as pgmUtility.cu if needed.), implement the function that paints a black dot (circle) in the image, you have to parameterize the center pointer and the radius of the circle you will draw. The function has been declared already in pgmUtility.h file. Inside this function, you have to invoke a CUDA kernel function that performs all image processing on GPU device. Please **also** design and implement the kernel function that can draw a circle on the image using CUDA. Note that this function pgmDrawCircle() is invoked by the CPU, but it launches a kernel that runs on GPU device.

```
int pgmDrawCircle( int **pixels, int numRows, int numCols, int centerRow,
                   int centerCol, int radius, char **header );
```

You **probably have to change** to the following signature if you **linearize** the 2D pixel matrix into 1D storage.

```
int pgmDrawCircle( int *pixels, int numRows, int numCols, int centerRow,
                   int centerCol, int radius, char **header );
```

- 3) In pgmUtility.c file (or you can renamed it as pgmUtility.cu if needed.), implement the function that paints a black edge frame in the image, you have to parameterize the width of the edge you will paint. The function has been declared already in pgmUtility.h file. Inside this function, you have to invoke a CUDA kernel function that performs all image processing on GPU device. Please **also** design and implement that kernel function that can draw an edge frame on the image using CUDA. Note that this function pgmDrawEdge() is invoked by the CPU, but it launches a kernel that runs on a GPU device.

```
int pgmDrawEdge( int **pixels, int numRows, int numCols, int edgeWidth, char **header );
```

You **probably have to change** to the following signature if you **linearize** the 2D pixel matrix into 1D storage.

```
int pgmDrawEdge( int *pixels, int numRows, int numCols, int edgeWidth, char **header );
```

- 4) As we learned in math, two points in a 2D plane determine a unique line segment. In pgmUtility.c (or pgmUtility.cu if needed) file, implement the function that draw a straight line segment in the image, after users pass in two points as parameters, a start point and an end point. You have to parameterize the start point and the end point of the line segment you will paint. The function has been declared already in pgmUtility.h file. Inside this function, you have to invoke a CUDA kernel function that performs all image processing on GPU device. Please **also** design and implement that kernel function that can draw an edge frame on the image using CUDA. Note that this function pgmDrawLine() is invoked by the CPU, but it launches a kernel that runs on a GPU device.

```
int pgmDrawLine( int **pixels, int numRows, int numCols, char **header, int p1row, int
                  p1col, int p2row, int p2col );
```

You **probably have to change** to the following signature if you **linearize** the 2D pixel matrix into 1D storage.

```
int pgmDrawLine( int *pixels, int numRows, int numCols, char **header, int p1row, int
                  p1col, int p2row, int p2col );
```

- 5) Note that after you paint the edge or the black dot in an image, you might have to update the header to reflect the new maximum intensity value in the header (specifically the last line of

image header). But in this homework, you are **NOT** required to update the maximum intensity value in the image header in your GPU code, nor in the CPU code.

- 6) In pgmUtility.c (or pgmUtility.cu if needed) file, implement the function that writes back to a new image file that contains your painting using file I/O. The function has been declared already in pgmUtility.h file. Note that this function only executes on CPU.

```
int pgmWrite( const char **header, const int **pixels, int numRows, int numCols, FILE *out );
```

You **probably have to change** to the following signature if you **linearize** the 2D pixel matrix into 1D storage.

```
int pgmWrite( const char **header, const int *pixels, int numRows, int numCols, FILE *out );
```

- 7) You are required to define the functions that are provided in the *.h file, you **CANNOT** change the signature (parameter list and return type) of these **existing functions except for** what have been allowed above when you linearize 2D array into 1D storage, though you are allowed to add other functions or definitions. You have to implement them in its corresponding *.c file and use them in main or other functions if necessary. **Please carefully read the comments on top of each function declared in pgmUtility.h file.**
- 8) In addition, please design and implement your three CUDA kernels to do the image processing, and call these kernels in the relevant functions defined above. You are required to place all CUDA functions into a separate pgmProcess.cu file, and declare them in a corresponding pgmProcess.h file.
- 9) You are required to include a Makefile in your folder that compiles all your code and link them into a target named **myPaint**. An example makefile has been provided named as **makefile_Sample**, which shows you how to jointly compile a cuda source file main.cu and a C file timing.c. You could follow this pattern to add more files for compiling.
- 10) You are required to deallocate all memories you dynamically allocated in program (either on host or device). Please make sure you do it in a proper time and at proper location in your program, where you know these memories are no longer useful.
- 11) You have to write your own main() function, in which if necessary, you call some of these function(s) specified above in order to **output a new image file on disk**. Depending on the command-line arguments passed in, the new image should look like what is shown in the **test cases section** below.
- 12) In addition to the input images that have been included, you have to **synthesize** your large pgm images for tests, with size 10000 by 10000, and size 1000 by 10000. When you synthesize your pgm images, basically you can assign a brightness value of 255(pure white) to each pixel. Please follow the pgm file format when you write to image file. You are ALSO required to write a **sequential** solution for step 2, step 3 and step 4 above, where drawEdge and drawCircle and drawLine are performed exclusively on CPU. **Meanwhile** you have to write a parallel GPU version for drawCircle , drawEdge and drawLine, as described above in step 2, 3 and 4. Then you have to measure the time cost for the sequential CPU solution and GPU solution of draw circle operation in step 2. You will do the same for step 3 and step 4 – draw edge and line

operations. Please calculate speedups of the GPU solution in comparison with sequential solution and save these results into a text file for submission.

- 13) In your main function, you have to write code to parse the command-line arguments. (FYI: sample command-line arguments parsing code is provided in package). When you run your program, you pass in the Circle Center, Radius or Edge Width as command line arguments. For example,

If the number of command line argument is not expected, your program are required to show a message:

Usage:

-e edgeWidth oldImageFile newImageFile

-c circleCenterRow circleCenterCol radius oldImageFile newImageFile

-l p1row p1col p2row p2col oldImageFile newImageFile

You have to run your program using command with this synopsis:

./programName -e edgeWidth originalImage newImageFile

to paint an edge of width of **edgeWidth** in the image of **originalImage**

./programName -c circleCenterRow circleCenterCol radius originalImage newImageFile
to paint a big round dot on the image with center at (**circleCenterRow**, **circleCenterCol**) and radius of **radius**.

./programName -l p1row p1col p2row p2col oldImageFile newImageFile

to draw a line at a start point with row number = **p1row** and column number = **p1col**, the line segment ends at a point with row number = **p2row** and column number **p2col**.

When user inputs wrong number of argument in command line or wrong input format, your program **should not crash**, instead showing the Usage message above.

If your program could handle drawing a circle and an edge at the same time in one command in your terminal window you get another **10** bonus points on top of 100.

In this case, your command line should look like the following; otherwise you cannot get the bonus.

./programName -ce circleCenterRow circleCenterCol radius edgeWidth originalImage newImageFile

AND

./programName -c -e circleCenterRow circleCenterCol radius edgeWidth originalImage newImageFile

Test cases with the provided image

./myPaint -c 470 355 100 ./balloons.ascii.pgm balloons_c100_4.pgm

Your program yields an image looks like:



./myPaint -c 228 285 75 ./balloons.ascii.pgm balloons_c75_5.pgm
The command above yields an image,



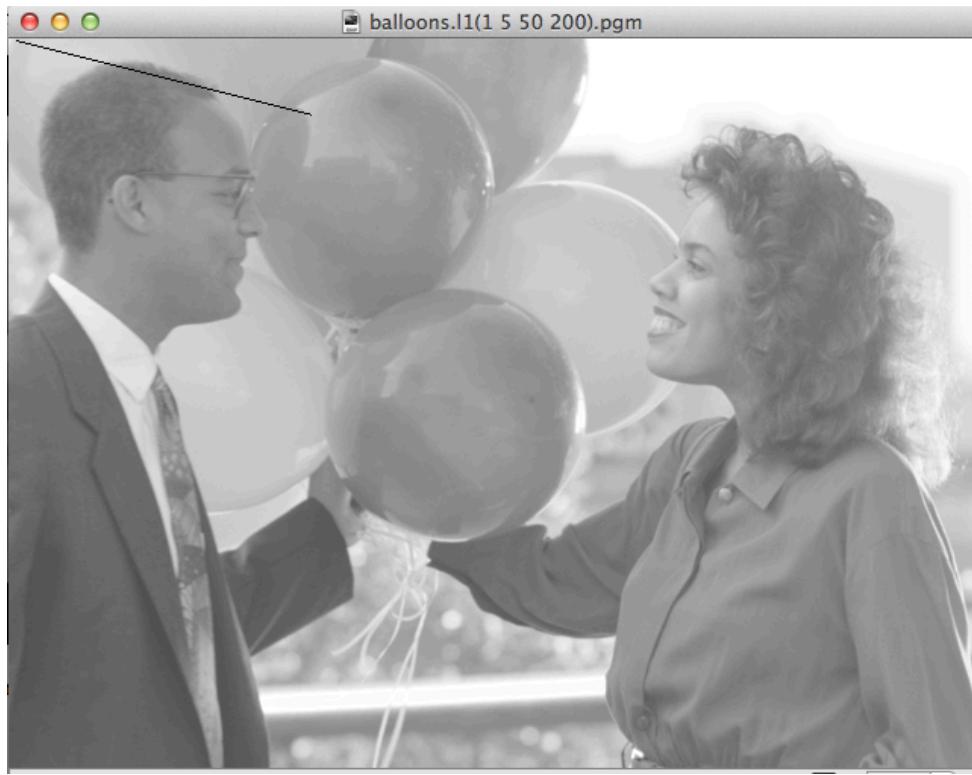
```
./myPaint -e 50 ./balloons.ascii.pgm balloons_e50_2.pgm
```

The command above produces an image that looks like,



```
./myPaint -l 1 5 50 200 ./balloons.ascii.pgm balloons_l1.pgm
```

The command above produces an image that looks like,



./myPaint -l 1 50 479 639 ./balloons.ascii.pgm balloons_l2.pgm
The command above produces an image that looks like,



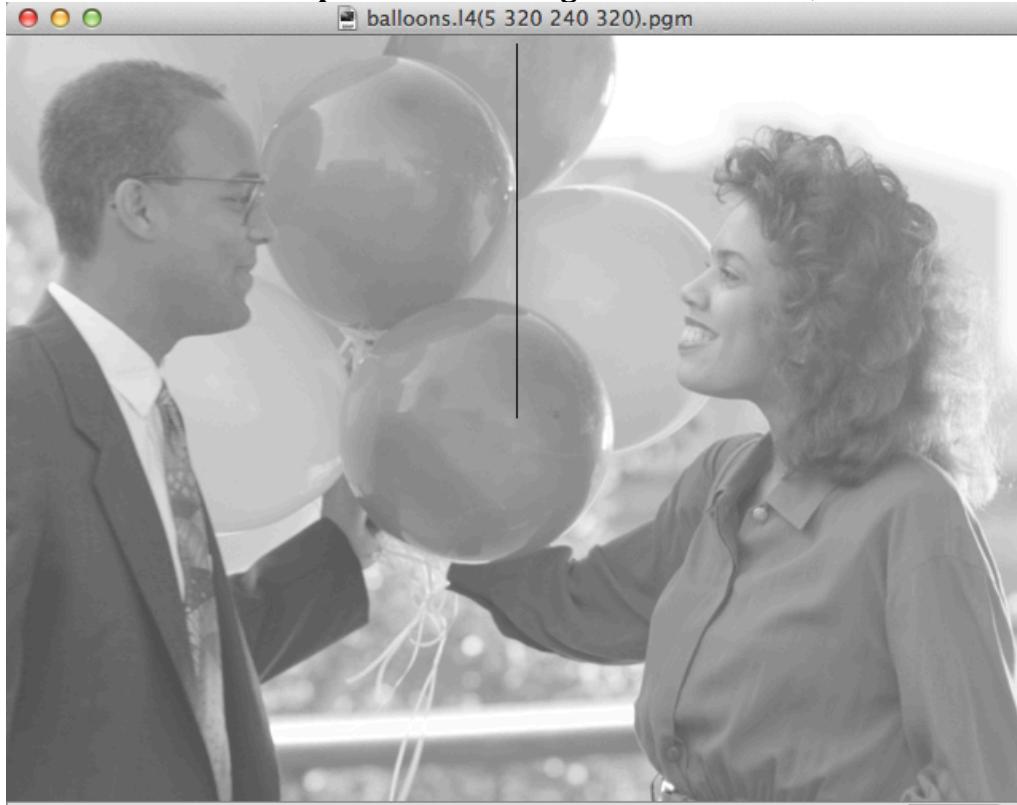
```
./myPaint -l 479 5 0 639 ./balloons.ascii.pgm balloons_l3.pgm
```

The command above produces an image that looks like,



```
./myPaint -l 5 320 240 320 ./balloons.ascii.pgm balloons_l4.pgm
```

The command above produces an image that looks like,



If input wrong command line parameters:

./myPaint -e 50 300 ./balloons.ascii.pgm

Usage:

-e edgeWidth oldImageFile newImageFile

-c circleCenterRow circleCenterCol radius oldImageFile newImageFile

./myPaint -e ab ./balloons.ascii.pgm

Usage:

-e edgeWidth oldImageFile newImageFile

-c circleCenterRow circleCenterCol radius oldImageFile newImageFile

./myPaint -e ./balloons.ascii.pgm

Usage:

-e edgeWidth oldImageFile newImageFile

-c circleCenterRow circleCenterCol radius oldImageFile newImageFile

./myPaint -c 470 355 ./balloons.ascii.pgm

Usage:

-e edgeWidth oldImageFile newImageFile

-c circleCenterRow circleCenterCol radius oldImageFile newImageFile

./myPaint -c 470 355 50 60 ./balloons.ascii.pgm

Usage:

-e edgeWidth oldImageFile newImageFile

-c circleCenterRow circleCenterCol radius oldImageFile newImageFile

./myPaint -c 470 90bc ./balloons.ascii.pgm

Usage:

-e edgeWidth oldImageFile newImageFile

-c circleCenterRow circleCenterCol radius oldImageFile newImageFile

The original image before your processing looks like:



To turn in:

Please wrap up all your source code, all test images, and a text file of time cost and speedup into a single zip file, name it as lastname + firstinitial + hw2.zip. If you are Will Smith, your zip file is named as smithwhw2.zip.

If you did the bonus points part, please clearly output a message on the stdout.

Turn in your single zip file on Canvas CSCD445 -> Assignments->Hw2->submit

If your code shows a compile error, you will get zero credit. If your file is corrupted, you get a zero credit.