Gaussian Blurring Process on Graphics Processing Unit with CUDA

1st Priya Tomar
Electrical Engineering
Roll No. 12041140
email: priyatomar@iitbhilai.ac.in

2nd Nancy Gupta
Computer Science and Engineering
Roll No. 12040950
email: nancygupta@iitbhilai.ac.in

3rd Konduri Naga Lakshmi Rekha
Computer Science and Engineering
RollNo. 12140930
email: kondurinaga@iitbhilai.ac.in

4th Riya
Computer Science and Engineering
Roll No. 12041230
email: riyad@iitbhilai.ac.in

***Abstract*—As we know that GPUs are good for image processing, we are going to implement a parallel version of Gaussain Blur using cuda programming. Gaussian blur is a technique to reduce the noise in images and hide details. If we blur the image in a sequential way, we will have to process each pixel one by one. We can't move to the next pixel until the previous pixel is processed. But in gaussian blur, processing for each pixel is independent of processing for another pixel, so we can assign each pixel to a different thread, and this computation can be parallelized. We will try to use optimisation techniques, such as launching multiple thread blocks and using shared memory, for better performance.**

I. Introduction

Gaussian blur is a technique to reduce the noise in images and hide details. It is also used as an image pre-processing step in many image processing algorithms. It smoothens the image, and the image looks as if it were seen through a translucent medium. It uses a gaussian function, which is used to generate the kernel. This kernel is a matrix that represents the weights we give to the nearby pixels. Pixels closer to the centre have higher weights as compared to the far away pixels. In a blurred image, the new pixel value is calculated by taking the weighted average over the neighbouring pixels.

The objective of the Gaussian blur is to reduce the image noise and detail by convolving the image with a Gaussian kernel, which results in a smoothed version of the original image. The challenge is to achieve this blur effect of the image while maintaining the image features as well as maintaining better computational performance. This technique is of great interest in the fields of computer vision and image processing where we need to maintain the good image quality.

The sample image with its corresponding gaussian blurred image is shown in the given figure:

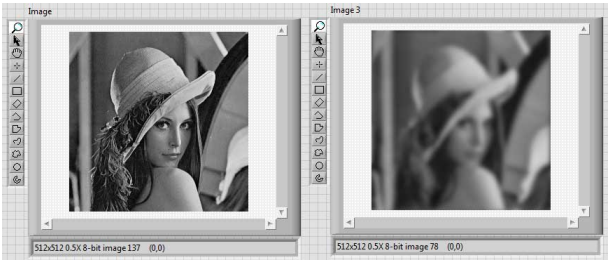

Fig. 1. Gaussian blur demonstration

Using a mean filter is an easy method to smooth things out. The goal is to replace each pixel with the average value of its neighbouring pixels, including itself. This technique has advantages such as simplicity and quickness. Outliers, particularly those far from the pixel of interest, can distort the true mean of the neighbourhood, which is a significant disadvantage.

Gaussian blur, often preferred over other techniques, addresses several limitations and is crucial in various image processing and computer vision applications. Some of the crucial features of Gaussian Blur include:

1) Noise Reduction
2) Preprocessing for Computer Vision
3) Edge Preservation
4) Data Compression
5) Emphasizing important patterns

II. Approach

We will implement Gaussian Blur in three ways, and then compare all these versions.

1) Sequential
2) Parallel without shared memory
3) Parallel with shared memory

I all the three versions, we will follow these steps-

1) Generate a grayscale input image
2) Generate a Gaussian Kernel
3) Convolution operation

For all these 3 versions, all steps are same except the convolution operation. Grayscale image will be generated by using generating random values between 0 and 255. For generating the Gaussian Kernel gaussian equation is being used.
The gaussian equation is defined as:

$$G(x,y) = \frac{1}{2\pi\sigma^2} e^{-\frac{x^2+y^2}{2\sigma^2}} \quad (1)$$

Where:

$G(x,y)$ is the Gaussian function
x is horizontal distance to centre pixel
y is vertical distance to centre pixel
σ is the standard deviation of the distribution.

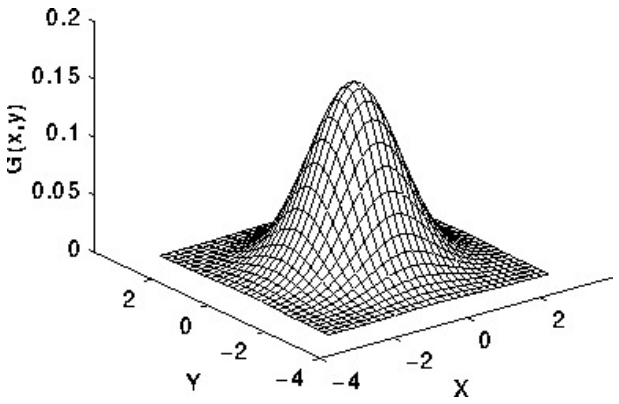

Fig. 2. Discrete kernel at (0,0) and $\sigma = 1$

In convoultion operation in sequential version, we will iterate over all the pixels in the image and then compute the new pixel by taking the weighted average over all the neighbouring pixels using the kernel.

For parallel version, we are launching a grid of thread blocks. Each thread block contains BLOCK_SIZE $\times$ BLOCK_SIZE number of threads. Each thread corresponds to particular pixel in image then we iterate over the gaussian kernel and calculate the output pixel value.

While computing the weighted average value, many pixels in the image will be reaccessed, so we will integrate shared memory so that we can decrease the memory access and improve the performance. We will also try to store the gaussian kernel in shared memory.

III. Evaluation Methodology

We will find execution times of all the versions for different-sized images. We will find the speedups and plot the speedups as a function of image size. We will compare our results with the reference paper's results. We will also compare all the three versions among each other.

A. The hardware platform we plan to use for conducting the experiments - Google Colab with the specification as follows:

Technical Features	Version and Calculating Capacity
Device Name	Tesla T4
CUDA version	12.2
Number of streaming multiprocessors	40
Shared memory available in each SM (bytes)	65536
Maximum size of the thread block	1024
Warp Size	32

B. Input Data-Sets

Our input dataset is a set of grayscale images of different sizes. In a grayscale image, each pixel value is between 0 and 255.

C. The third-party framework/implementation, which you will compare the effectiveness of your implementation

We will compare our effectiveness (i.e., execution times by experimenting with different kernel sizes and input images) with the results that are already stated in the reference papers we mentioned.

IV. Experimental Results and Related Work

Processing times for all the three versions which we have implemented is given in the following tables.

TABLE I
SEQUENTIAL VERSION

Image resolution(pixels)	Kernel size	Processing time (s)
256 x 256	7 x 7	0.021036
	13 x 13	0.068195
	15 x 15	0.092056
	17 x 17	0.116104
1920 x 1200	7 x 7	0.804126
	13 x 13	2.995063
	15 x 15	3.369317
	17 x 17	4.705112
3840 × 2160	7 x 7	2.944086
	13 x 13	9.876870
	15 x 15	12.663888
	17 x 17	16.805064

TABLE II
PARALLEL VERSION WITHOUT SHARED MEMORY

Image resolution(pixels)	Kernel size	Processing time(s)
256 x 256	7 x 7	0.000452
	13 x 13	0.000816
	15 x 15	0.000963
	17 x 17	0.001178
1920 x 1200	7 x 7	0.005749
	13 x 13	0.019116
	15 x 15	0.027132
	17 x 17	0.032446
3840 x 2160	7 x 7	0.020108
	13 x 13	0.068127
	15 x 15	0.090519
	17 x 17	0.116237

TABLE III
PARALLEL VERSION WITH SHARED MEMORY

Image resolution(pixels)	Kernel size	Processing time(s)
256 x 256	7 x 7	0.000608
	13 x 13	0.001072
	15 x 15	0.001334
	17 x 17	0.001612
1920 x 1200	7 x 7	0.008197
	13 x 13	0.023408
	15 x 15	0.030297
	17 x 17	0.038006
3840 x 2160	7 x 7	0.027867
	13 x 13	0.082299
	15 x 15	0.106925
	17 x 17	0.134335

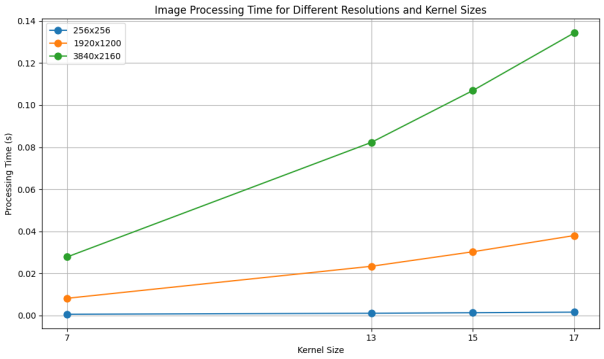

Fig. 5. Processing time for parallel version (with shared memory)

A. Visualization of Results

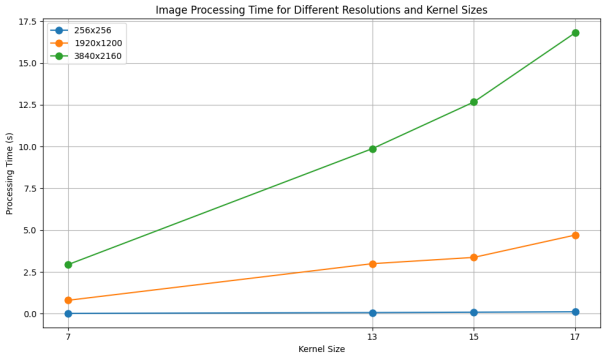

Fig. 3. Processing time for sequential version

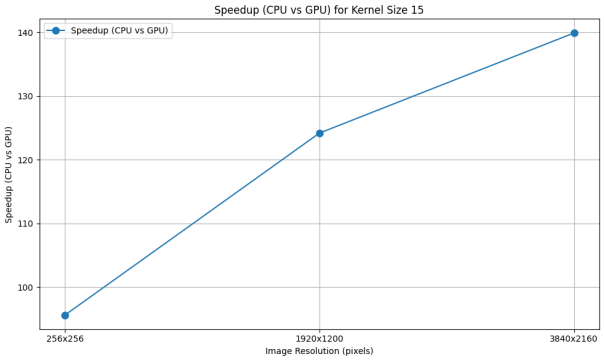

Fig. 6. Speedup for parallel version (without shared memory)

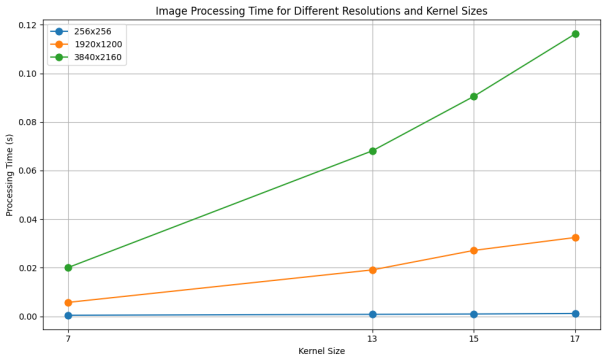

Fig. 4. Processing time for parallel version (without shared memory)

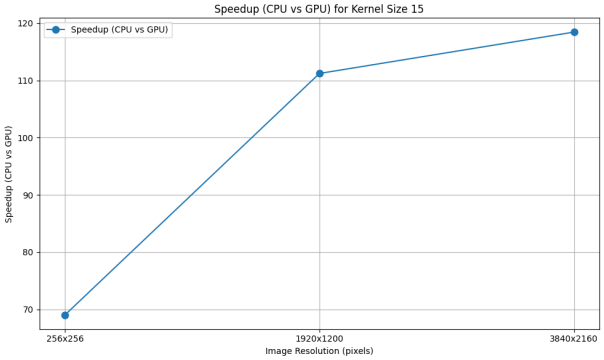

Fig. 7. Speedup for parallel version (with shared memory)

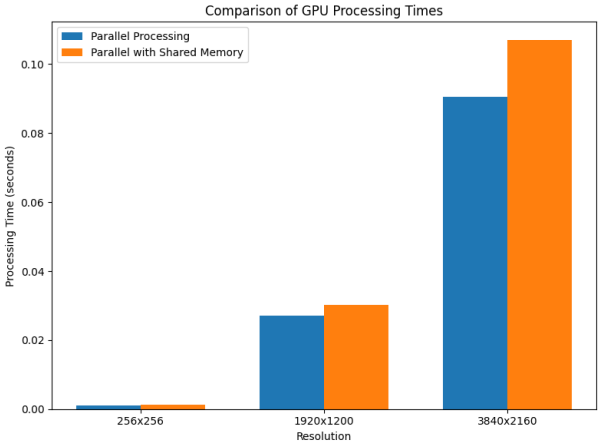

Fig. 8. Comparison between parallel versions with and without shared memory

Results from reference paper-

Table 1: CPU time.

Image resolution (pixels)	Kernel size	Processing time (s)
256 x 256	7 x 7	8.2
	13 x 13	34.9
	15 x 15	47.8
	17 x 17	62.3
1920 x 1200	7 x7	429.3
	13 x 13	1358.2
	15 x 15	1882.9
	17 x 17	2294.15
3840 × 2160	7 x 7	1296.4
	13 x 13	5322.2
	15 x 15	7347.02
	17 x 17	9225.1

Table 2: GPU time.

Image resolution (pixels)	Kernel size	Processing time (s)
256 x 256	7 x 7	0.00200
	13 x 13	0.00298
	15 x 15	0.00312
	17 x 17	0.00363
1920 x 1200	7 x 7	0.02726
	13 x 13	0.03582
	15 x 15	0.04410
	17 x 17	0.054377
3840 × 2160	7 x 7	0.06950
	13 x 13	0.11282
	15 x 15	0.14214
	17 x 17	0.17886

V. Conclusion

Our both parallel versions with and without shared memory are working better than reference paper's parallel version. This may be due to the reason that they are doing it for 3-channel images and they are using different GPU as well for measuring the perfomance of the code. Using shared memory is not giving any improvement in the performance. Processing times are almost comparable for both parallel version with and without shared memory. Shared memory is not providing any performance improvement. This can be due to the following reasons.

- **Bank Conflicts:** Shared memory in CUDA has banks, and if multiple threads access the same bank simultaneously, bank conflicts occur, leading to serialized access.
- **Memory Coalescing in Global Memory:** Global memory access patterns that are coalesced can be quite fast. If shared memory doesn't offer a clear advantage, the default global memory access pattern might be sufficient.

References

[1] Gaussian Blur through Parallel Computing : Nahla M. Ibrahim, Ahmed Abou ElFarag and Rania Kadry
[2] Effective Gaussian Blurring Process on Graphics Processing Unit with CUDA : Ferhat Bozkurt, Mete Yağanoğlu, and Faruk Baturalp Günay
[3] IMAGE PROCESSING WITH CUDA by Jia Tse,Bachelor of Science, University of Nevada, Las Vegas 2006