COE-506: GPU Programming and Architecture
(Course Project Report) - Implementation of a Real-World Application Using OpenACC, CUDA Python, and CUDA C/C++

1. Title Page

- Gaussian Blur
- Khalid Alhazmi 201526710
- COE 506
- Ayaz
- 17 dec 2023

2. Abstract

The project aimed to develop a solution for implementing Gaussian blur using CUDA in both Python and C programming languages. Gaussian blur, a common image processing technique, smooths images by reducing noise and detail. The project's approach involved leveraging the parallel processing capabilities of CUDA, a parallel computing platform and application programming interface model created by Nvidia. By utilizing CUDA, the project sought to enhance the efficiency and speed of the Gaussian blur process. Key findings included significant improvements in processing time compared to traditional CPU-based methods, demonstrating the effectiveness of using CUDA for high-performance image processing tasks.

3. Introduction

In this article we will go thoroughly implementing gaussian blur using CUDA C and CUDA Python and show you how improvements we got when we compare it to sequential versions using python

4. Background and Related Work

Details: Present an overview of the key literature, previous studies, or existing solutions related to your project. Discuss how your work builds upon or differs from these previous efforts.
Gaussian blur is used across many if not most photo editing application like photoshop and also there are many famous library implement gaussian blur such OpenCV for C and Pillow for python

5. Methodology

For this problem the stack of programming models and algorithm will separated into two parts

1. Sequential code and it will contains all functionalities related to work with loading the image and creating the kernel – will talk about it later – and exploring the image
2. Parallel(Cuda code) and it will handle applying the filter in each pixel in the image

for this project will be implementing the problem into three GPU Models approach
CUDA Python , CUDA C/C++ and using Python/Numba

6. Implementation Details

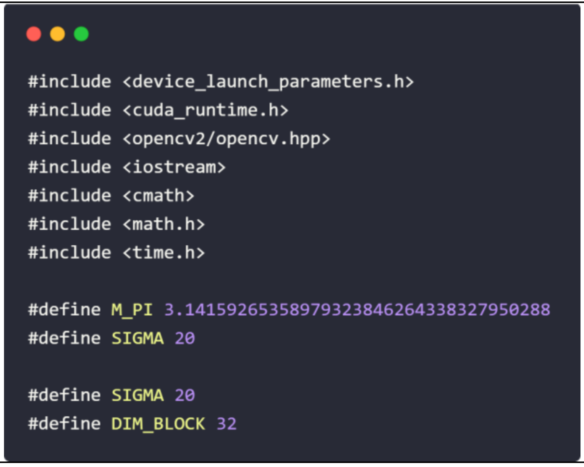
CUDA C/C++

Main function

<table>
<tr>
<td>

Includes standard libraries and CUDA-related headers.
Uses OpenCV for image handling.
Defines constants like M_PI, SIGMA, and DIM_BLOCK.

</td>
<td>



```
#include <device_launch_parameters.h>
#include <cuda_runtime.h>
#include <opencv2/opencv.hpp>
#include <iostream>
#include <cmath>
#include <math.h>
#include <time.h>

#define M_PI 3.14159265358979323846264338327950288
#define SIGMA 20

#define SIGMA 20
#define DIM_BLOCK 32
```

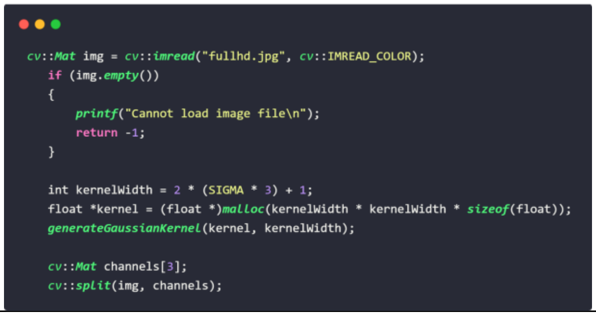
</td>
</tr>
<tr>
<td>

Image Loading: Uses OpenCV to load an image

Image Channel Splitting: Splits the image into three color channels.

Image Channel Splitting: Splits the image into three color channels

</td>
<td>



```
cv::Mat img = cv::imread("fullhd.jpg", cv::IMREAD_COLOR);
    if (img.empty())
    {
        printf("Cannot load image file\n");
        return -1;
    }

    int kernelWidth = 2 * (SIGMA * 3) + 1;
    float *kernel = (float *)malloc(kernelWidth * kernelWidth * sizeof(float));
    generateGaussianKernel(kernel, kernelWidth);

    cv::Mat channels[3];
    cv::split(img, channels);
```

</td>
</tr>
<tr>
<td>

Converts each channel to an appropriate format.

Allocates memory on the GPU for each channel and the kernel.

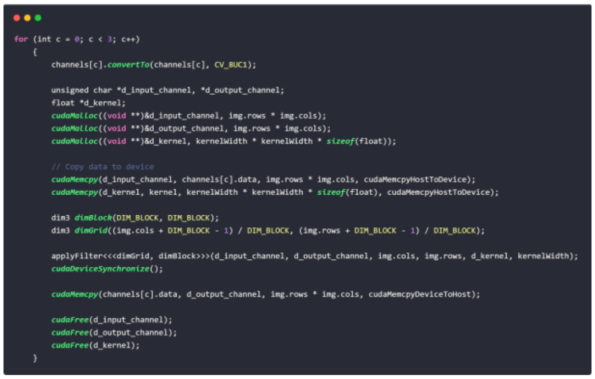
Copies data to the GPU.

Sets up CUDA dimensions and applies the filter using the applyFilter kernel.

Copies the processed channel back to host memory.

Frees GPU memory.

</td>
<td>



```
for (int c = 0; c < 3; c++)
    {
        channels[c].convertTo(channels[c], CV_8UC1);

        unsigned char *d_input_channel, *d_output_channel;
        float *d_kernel;
        cudaMalloc((void **)&d_input_channel, img.rows * img.cols);
        cudaMalloc((void **)&d_output_channel, img.rows * img.cols);
        cudaMalloc((void **)&d_kernel, kernelWidth * kernelWidth * sizeof(float));

        // Copy data to device
        cudaMemcpy(d_input_channel, channels[c].data, img.rows * img.cols, cudaMemcpyHostToDevice);
        cudaMemcpy(d_kernel, kernel, kernelWidth * kernelWidth * sizeof(float), cudaMemcpyHostToDevice);

        dim3 dimBlock(DIM_BLOCK, DIM_BLOCK);
        dim3 dimGrid((img.cols + DIM_BLOCK - 1) / DIM_BLOCK, (img.rows + DIM_BLOCK - 1) / DIM_BLOCK);

        applyFilter<<<dimGrid, dimBlock>>>(d_input_channel, d_output_channel, img.cols, img.rows, d_kernel, kernelWidth);
        cudaDeviceSynchronize();

        cudaMemcpy(channels[c].data, d_output_channel, img.rows * img.cols, cudaMemcpyDeviceToHost);

        cudaFree(d_input_channel);
        cudaFree(d_output_channel);
        cudaFree(d_kernel);
    }
```

</td>
</tr>
</table>

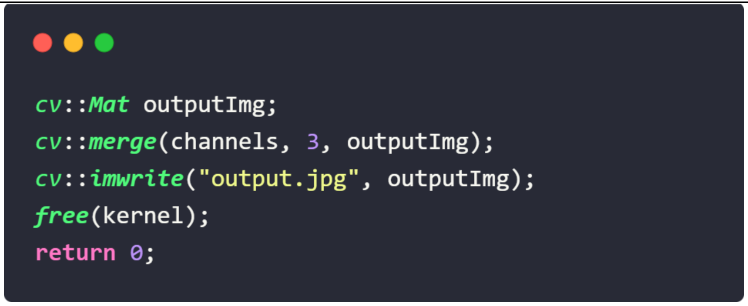
<table>
<tr>
<td>

Image Reconstruction: Merges the processed channels back into one image.

Output Saving: Saves the processed image using OpenCV.

Memory Cleanup: Frees the allocated kernel memory on the host.

</td>
<td>



```
cv::Mat outputImg;
cv::merge(channels, 3, outputImg);
cv::imwrite("output.jpg", outputImg);
free(kernel);
return 0;
```

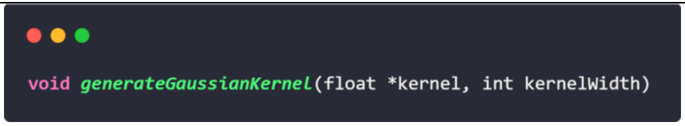
</td>
</tr>
</table>

generate Gaussian Kernel function.

<table>
<tr>
<td>

Generates a Gaussian kernel based on the given sigma value.
Normalizes the kernel values so their sum equals 1.

</td>
<td>



```
void generateGaussianKernel(float *kernel, int kernelWidth)
```

</td>
</tr>
</table>

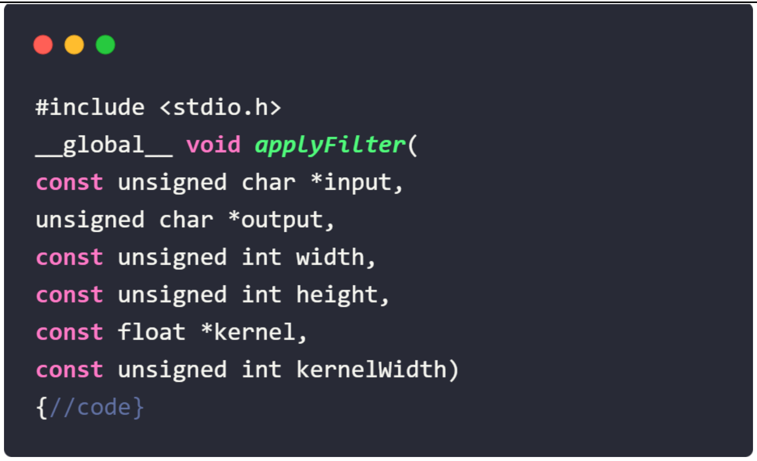
apply Filter functions.

<table>
<tr>
<td>

applyFilter is a __global__ function, meaning it's intended to be called from the host and executed on the device (GPU).

- input: Pointer to the input image data (unsigned char array).
- output: Pointer to the output image data (unsigned char array).
- width and height: Dimensions of the image.
- kernel: Pointer to the filter kernel (a float array).
- kernelWidth: The width of the kernel.

</td>
<td>



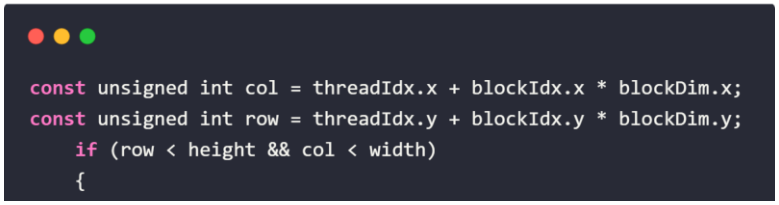
```
#include <stdio.h>
__global__ void applyFilter(
const unsigned char *input,
unsigned char *output,
const unsigned int width,
const unsigned int height,
const float *kernel,
const unsigned int kernelWidth)
{//code}
```

</td>
</tr>
<tr>
<td>

Calculates the column (col) and row (row) in the image that the current thread is processing, using thread and block indices.

checks if the calculated row and column are within the image bounds.

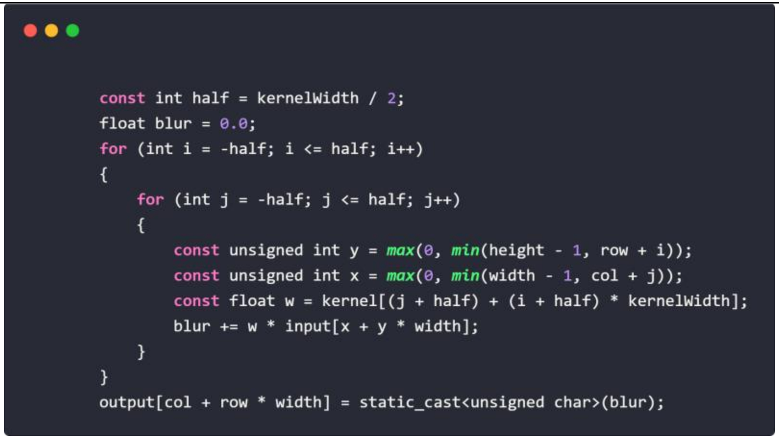
</td>
<td>



```
const unsigned int col = threadIdx.x + blockIdx.x * blockDim.x;
const unsigned int row = threadIdx.y + blockIdx.y * blockDim.y;
    if (row < height && col < width)
    {
```

</td>
</tr>
</table>

- Initializes a variable blur to
- accumulate the filtered value.
- Iterates over the kernel using two nested loops.
- For each position in the kernel:
- Calculates corresponding image coordinates (x, y), clamping them to stay within image boundaries.
- Retrieves the kernel weight (w) for the current position.
- Accumulates the weighted pixel value from the input image to blur.
- Sets the corresponding pixel in the output image to the accumulated blur value, casting it to unsigned char.

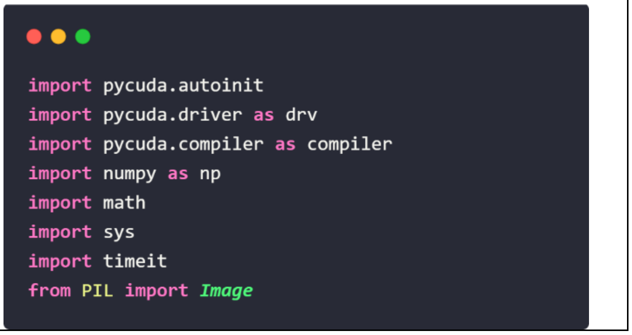


```
const int half = kernelWidth / 2;
float blur = 0.0;
for (int i = -half; i <= half; i++)
{
    for (int j = -half; j <= half; j++)
    {
        const unsigned int y = max(0, min(height - 1, row + i));
        const unsigned int x = max(0, min(width - 1, col + j));
        const float w = kernel[(j + half) + (i + half) * kernelWidth];
        blur += w * input[x + y * width];
    }
}
output[col + row * width] = static_cast<unsigned char>(blur);
```

CUDA/Python

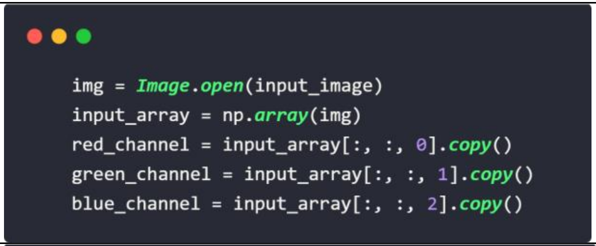
- Python Code

Imports: The script imports necessary modules including PyCUDA for GPU programming, NumPy for numerical operations, PIL for image processing, and others for system and timing functionalities.



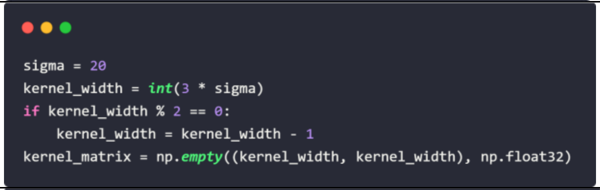
```
import pycuda.autoinit
import pycuda.driver as drv
import pycuda.compiler as compiler
import numpy as np
import math
import sys
import timeit
from PIL import Image
```

Image Loading and Channel Separation: Tries to open the input image using PIL. Converts it into a NumPy array and separates it into red, green, and blue color channels



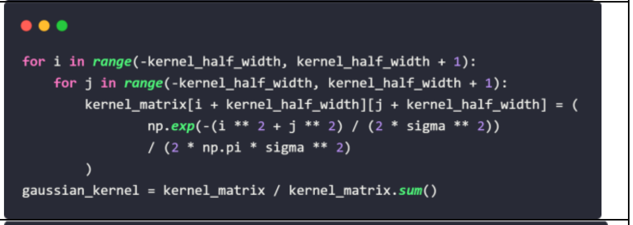
```
img = Image.open(input_image)
input_array = np.array(img)
red_channel = input_array[:, :, 0].copy()
green_channel = input_array[:, :, 1].copy()
blue_channel = input_array[:, :, 2].copy()
```

The kernel size: Creating the kernel size which is the part that will go through each pixel and get the average value from its around’s values



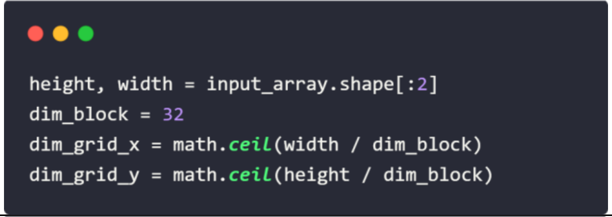
```
sigma = 20
kernel_width = int(3 * sigma)
if kernel_width % 2 == 0:
    kernel_width = kernel_width - 1
kernel_matrix = np.empty((kernel_width, kernel_width), np.float32)
```

Creates a Gaussian kernel matrix using the Gaussian formula.



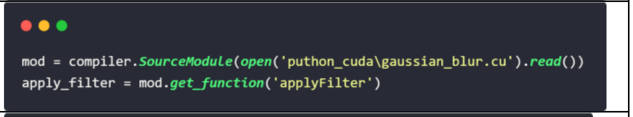
```
for i in range(-kernel_half_width, kernel_half_width + 1):
    for j in range(-kernel_half_width, kernel_half_width + 1):
        kernel_matrix[i + kernel_half_width][j + kernel_half_width] = (
                np.exp(-(i ** 2 + j ** 2) / (2 * sigma ** 2))
                / (2 * np.pi * sigma ** 2)
        )
gaussian_kernel = kernel_matrix / kernel_matrix.sum()
```

Calculates the dimensions for CUDA blocks and grids based on the image's width and height.



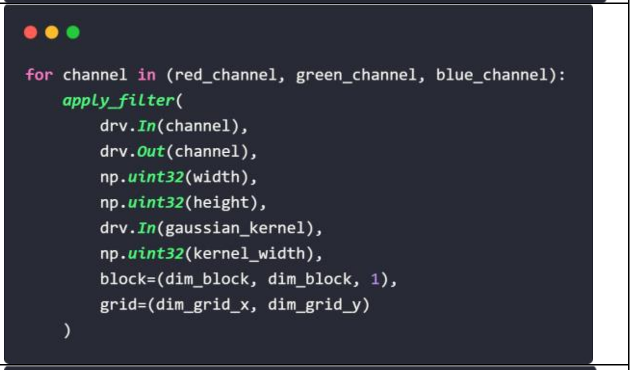
```
height, width = input_array.shape[:2]
dim_block = 32
dim_grid_x = math.ceil(width / dim_block)
dim_grid_y = math.ceil(height / dim_block)
```

Load the Cuda code using python



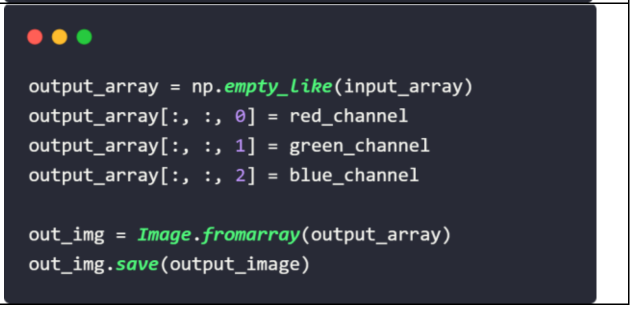
```
mod = compiler.SourceModule(open('puthon_cuda\gaussian_blur.cu').read())
apply_filter = mod.get_function('applyFilter')
```

Apply the filter for each color channel



```
for channel in (red_channel, green_channel, blue_channel):
    apply_filter(
        drv.In(channel),
        drv.Out(channel),
        np.uint32(width),
        np.uint32(height),
        drv.In(gaussian_kernel),
        np.uint32(kernel_width),
        block=(dim_block, dim_block, 1),
        grid=(dim_grid_x, dim_grid_y)
    )
```

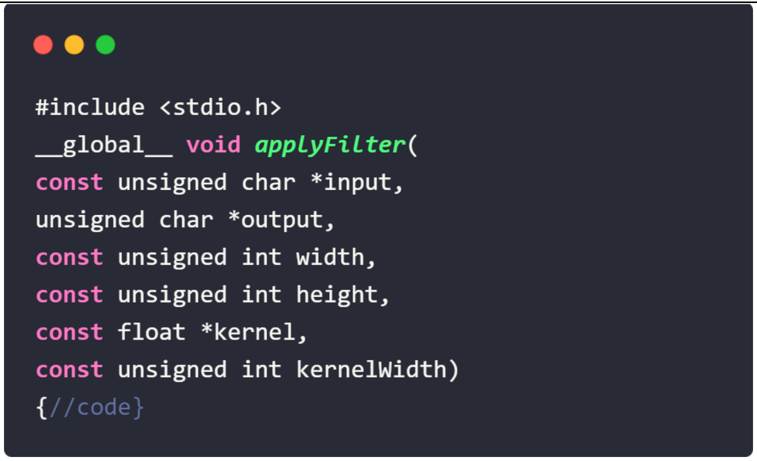
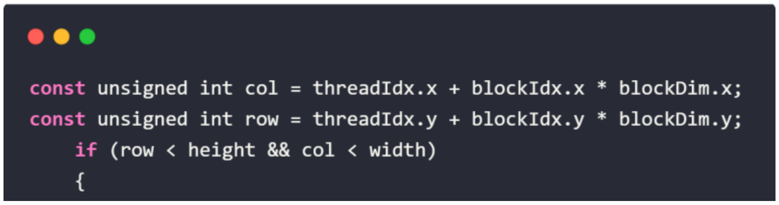
Create a new image that have the same size as the original image and then update it with new colors after applying the filter

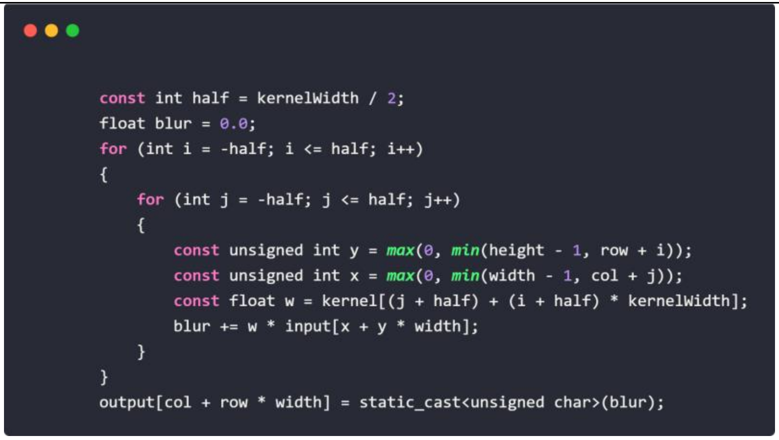


```
output_array = np.empty_like(input_array)
output_array[:, :, 0] = red_channel
output_array[:, :, 1] = green_channel
output_array[:, :, 2] = blue_channel

out_img = Image.fromarray(output_array)
out_img.save(output_image)
```

CUDA code

applyFilter is a __global__ function, meaning it's intended to be called from the host and executed on the device (GPU). - input: Pointer to the input image data (unsigned char array). - output: Pointer to the output image data (unsigned char array). - width and height: Dimensions of the image. - kernel: Pointer to the filter kernel (a float array). - kernelWidth: The width of the kernel.	 #include <stdio.h> __global__ void applyFilter(const unsigned char *input, unsigned char *output, const unsigned int width, const unsigned int height, const float *kernel, const unsigned int kernelWidth) {//code}
Calculates the column (col) and row (row) in the image that the current thread is processing, using thread and block indices. checks if the calculated row and column are within the image bounds.	 const unsigned int col = threadIdx.x + blockIdx.x * blockDim.x; const unsigned int row = threadIdx.y + blockIdx.y * blockDim.y; if (row < height && col < width) {

- Initializes a variable blur to
- accumulate the filtered value.
- Iterates over the kernel using two nested loops.
- For each position in the kernel:
- Calculates corresponding image coordinates (x, y), clamping them to stay within image boundaries.
- Retrieves the kernel weight (w) for the current position.
- Accumulates the weighted pixel value from the input image to blur.
- Sets the corresponding pixel
	
	const int half = kernelWidth / 2;
	float blur = 0.0;
	for (int i = -half; i <= half; i++)
	{
	    for (int j = -half; j <= half; j++)
	    {
	        const unsigned int y = max(0, min(height - 1, row + i));
	        const unsigned int x = max(0, min(width - 1, col + j));
	        const float w = kernel[(j + half) + (i + half) * kernelWidth];
	        blur += w * input[x + y * width];
	    }
	}
	output[col + row * width] = static_cast<unsigned char>(blur);

in the output image to the accumulated blur value, casting it to unsigned char.

NUMBA/Python

- Python Code

numpy: A fundamental package for scientific computing in Python.

math: Provides access to mathematical functions.

sys: System-specific parameters and functions.

timeit: Measures execution time of small code snippets.

Image from PIL: Image processing capabilities.

cuda from numba: Support for CUDA GPU programming.

```
import numpy as np
import math
import sys
import timeit
from PIL import Image
from numba import cuda
```

This function, decorated with @cuda.jit, is a CUDA kernel designed to run on the GPU.

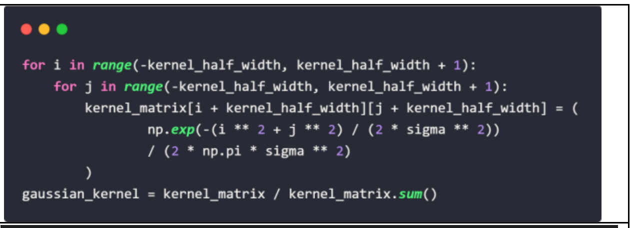
It calculates the Gaussian blur for each pixel of the image in parallel. The col and row variables determine the pixel's position.

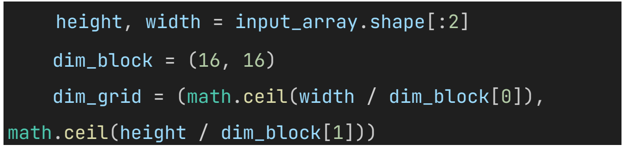
The nested loops apply the Gaussian kernel (blur effect) to each pixel by computing a weighted average of the surounding pixels.

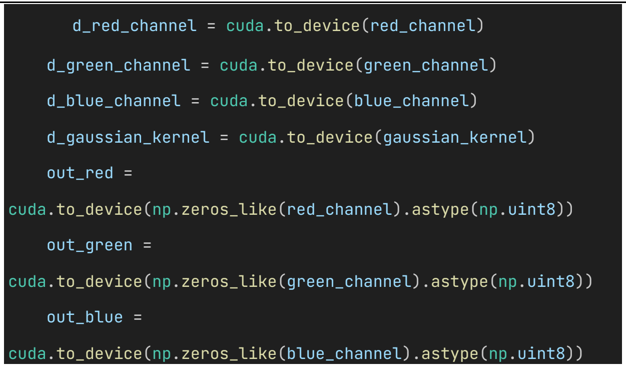
```
@cuda.jit
def apply_filter_numba(input, output, width, height, kernel,
kernelWidth):
    col = cuda.threadIdx.x + cuda.blockIdx.x * cuda.blockDim.x
    row = cuda.threadIdx.y + cuda.blockIdx.y * cuda.blockDim.y
    if row < height and col < width:
        half = kernelWidth // 2
        blur = 0.0
        for i in range(-half, half + 1):
            for j in range(-half, half + 1):
                y = max(0, min(height - 1, row + i))
                x = max(0, min(width - 1, col + j))
                w = kernel[i + half][j + half]
                blur += w * input[x][y]
        output[x][y] = np.uint8(blur)
```

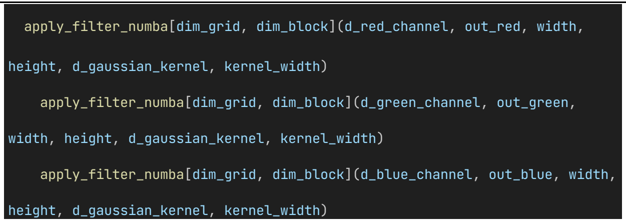
The kernel size: Creating the kernel size which is the part that will go through each pixel and get the average value from its around's values

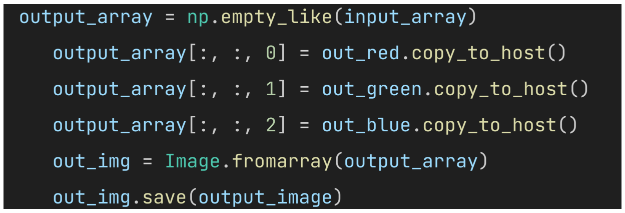
```
  img = Image.open(input_image)
    input_array = np.array(img).astype(np.uint8)
    red_channel = input_array[:, :, 0].copy().astype(np.uint8)
    green_channel = input_array[:, :, 1].copy().astype(np.uint8)
    blue_channel = input_array[:, :, 2].copy().astype(np.uint8)
```

Creates a Gaussian kernel matrix using the Gaussian formula.	
	`for i in range(-kernel_half_width, kernel_half_width + 1):`
	`    for j in range(-kernel_half_width, kernel_half_width + 1):`
	`        kernel_matrix[i + kernel_half_width][j + kernel_half_width] = (`
	`                np.exp(-(i ** 2 + j ** 2) / (2 * sigma ** 2))`
	`                / (2 * np.pi * sigma ** 2)`
	`        )`
	`gaussian_kernel = kernel_matrix / kernel_matrix.sum()`

Calculates the dimensions for CUDA blocks and grids based on the image's width and height.	
	`    height, width = input_array.shape[:2]`
	`    dim_block = (16, 16)`
	`    dim_grid = (math.ceil(width / dim_block[0]),`
	`math.ceil(height / dim_block[1]))`

Allocate memory on device to move data from host(CPU) to GP	
	`        d_red_channel = cuda.to_device(red_channel)`
	`    d_green_channel = cuda.to_device(green_channel)`
	`    d_blue_channel = cuda.to_device(blue_channel)`
	`    d_gaussian_kernel = cuda.to_device(gaussian_kernel)`
	`    out_red =`
	`cuda.to_device(np.zeros_like(red_channel).astype(np.uint8))`
	`    out_green =`
	`cuda.to_device(np.zeros_like(green_channel).astype(np.uint8))`
	`    out_blue =`
	`cuda.to_device(np.zeros_like(blue_channel).astype(np.uint8))`

Apply the filter for each color channel use the varable we create in the previous step	
	`  apply_filter_numba[dim_grid, dim_block](d_red_channel, out_red, width,`
	`height, d_gaussian_kernel, kernel_width)`
	`    apply_filter_numba[dim_grid, dim_block](d_green_channel, out_green,`
	`width, height, d_gaussian_kernel, kernel_width)`
	`    apply_filter_numba[dim_grid, dim_block](d_blue_channel, out_blue, width,`
	`height, d_gaussian_kernel, kernel_width)`

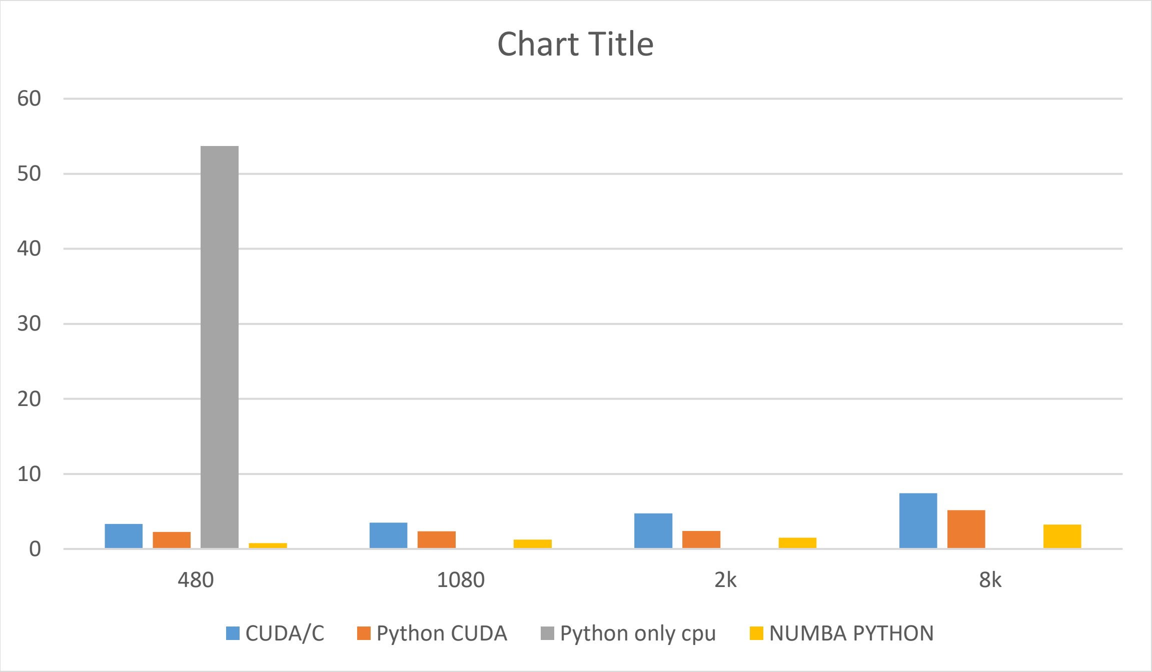
Moving data from device (GPU or CUDA) to the CPU and then save the new image	
	` output_array = np.empty_like(input_array)`
	`    output_array[:, :, 0] = out_red.copy_to_host()`
	`    output_array[:, :, 1] = out_green.copy_to_host()`
	`    output_array[:, :, 2] = out_blue.copy_to_host()`
	`    out_img = Image.fromarray(output_array)`
	`    out_img.save(output_image)`

7. Comparative Analysis

Image Size	Cuda/C time	Python Cuda Time	Python/numba	Python only CPI
480	3.34	2.28	0.775	53.66
1080	3.52	2.34	1.26	Too long
2k	4.76	2.39	1.50	-
8k	7.42	5.16	3.24	-

* The sequential implementations using almost the same algorithm as the CUDA one

8. Results and Discussion



As the result shows using the Cuda increase the performance dramatically but as we can see the Python (in both using pycuda or numba)version is faster than the C one and that come to bad memory management as python implementation out of the box implement the data syncing better than my C version but if we invest more time and effort into the C implementation we can make it faster

9. Conclusion

In summary, this analysis has highlighted that using CUDA implementation gives us a huge boost in computations power comparing to relying to CPU only . while that C is compiled which means its faster than python we knew that CUDA team have invest a lot time in making the memory

management and data passing between CPU and GPU is great as we got lower number in than python using numba and pycuda

10. References

Fernando, S. R. (n.d.). *Gaussian Blur*. OpenCV Tutorial C++. https://www.opencv-srf.com/2018/03/gaussian-blur.html

Computerphile. (2015, October 2). *How blurs & Filters work - Computerphile* [Video]. YouTube. https://www.youtube.com/watch?v=C_zFhWdM4ic

11. Appendix

Git repo link https://github.com/ipkalid/GaussianBlurUsingCUDA

COE-506: GPU Programming and Architecture
(Course Project Report) - Implementation of a Real-World Application Using OpenACC, CUDA Python, and CUDA C/C++

Grading Rubrics

1. **Originality and Relevance** - 15%

- Novelty of the project idea and approach.
- Relevance to GPU programming and architecture.
- Alignment with course objectives and current trends in GPU computing.

2. **Depth of Analysis and Implementation** - 25%

- Technical depth and correctness of the GPU implementation.
- Complexity of the problem tackled and effectiveness of the solution.
- Application of course concepts like parallel programming patterns, memory management, and algorithm optimization.

3. **Quality of Documentation** - 15%

- Clarity and coherence of writing.
- Logical organization and flow of the report.
- Quality of figures, graphs, and tables used.

4. **Final Report Quality (structure, clarity, coverage)** - 15%

- Adherence to the provided report template.
- Comprehensiveness of the content covering all required sections.
- Use of clear, concise, and technical language appropriate for the subject matter.

5. **References and Appendices** - 10%

- Adequacy and appropriateness of the references cited.
- Quality and relevance of the supplementary material in the appendices.

6. **Publishable Outcome (Conference Paper, Medium Article, or GitHub Repository)** - 20%

- Quality and professionalism of the draft/article/repository.
- Relevance and accuracy of the content in relation to the project.
- Clarity, coherence, and organization of the material.
- Engagement and potential impact on the intended audience.

Instructor: Dr. Ayaz ul Hassan Khan