PCAP (CSE -3263) MINI PROJECT

REPORT ON

iFilter

A CUDA-based Parallel

Computing Architecture for Image Filtering

SUBMITTED TO

Department of Computer Science & Engineering

By

Name	**Registration Number**	**Roll Number**	**Sem/Sec**
Akshay Gupta	200905040	04	VI/C
Kaustubh Pandey	200905142	24	VI/C
Rajpreet Lal Das	200905052	08	VI/C
Shubham Srivastava	200905152	26	VI/C

Dr.Radhika Kamath
(Associate Professor)

Name & Signature of
Evaluator 1

Dr.Manjunath K N
(Associate Professor)

Name & Signature of
Evaluator 2

(09-05-2023)

INTRODUCTION

Image filtering is a common technique used in image processing to enhance, manipulate, or analyze digital images. The filtering process involves applying a predefined mathematical function or algorithm to each pixel in an image to produce a new image. This can be computationally intensive, particularly for large images, which is where parallel computing can offer significant advantages. CUDA is a popular parallel computing architecture used for image processing due to its ability to leverage the parallel processing power of NVIDIA GPUs. In this report, we present iFilter, a CUDA-based parallel computing architecture for image filtering.

OBJECTIVE

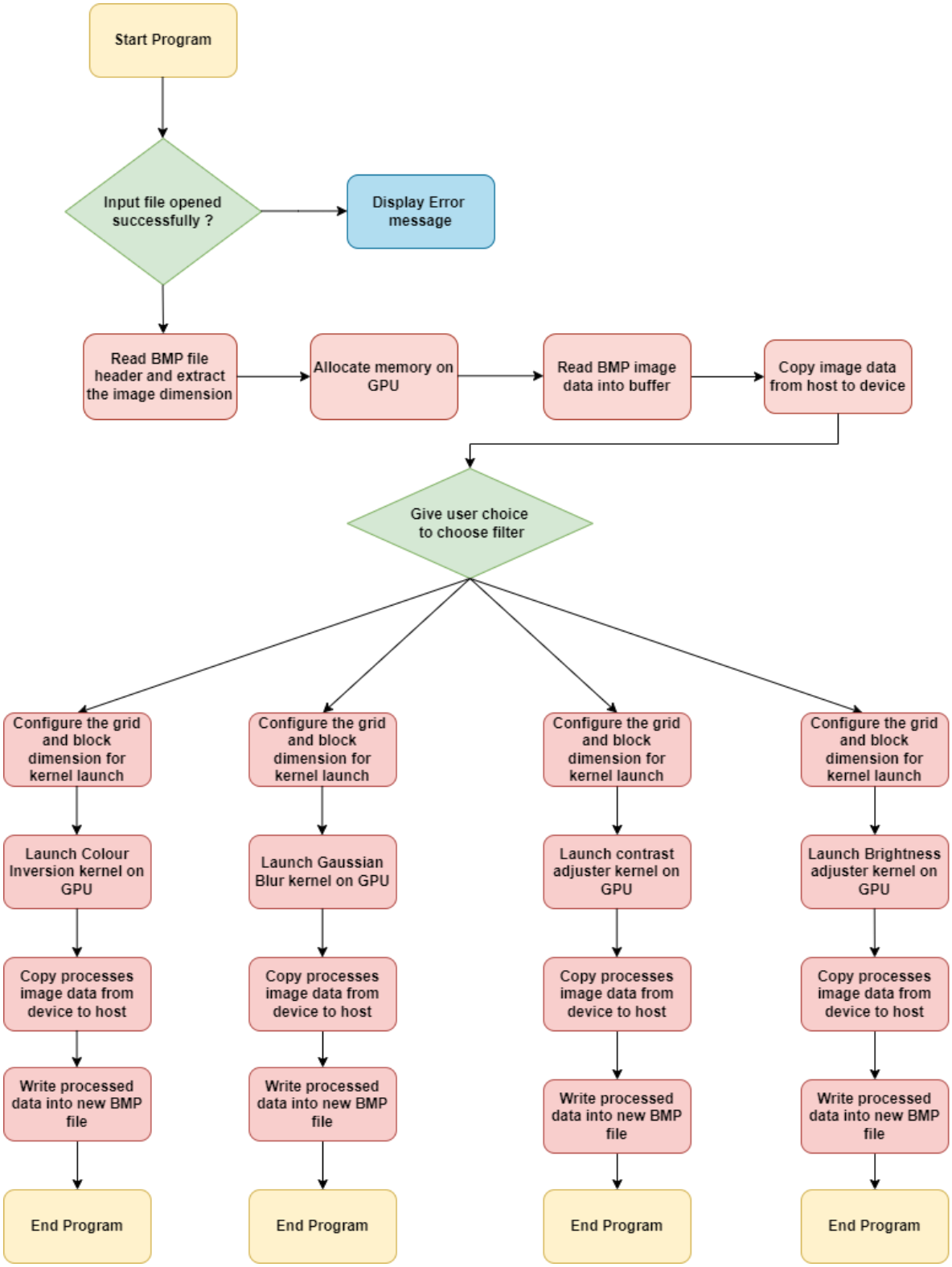
The main objective of iFilter is to demonstrate the use of CUDA for image filtering. Specifically, iFilter allows users to apply four common image filters (gaussian blur, color invert, adjust brightness, and adjust contrast) to a bitmap image file. The implementation is parallelized using CUDA to take advantage of the GPU's parallel processing power, resulting in significant speedup over a traditional CPU-based implementation.

DESIGN

The iFilter architecture is designed to take a bitmap image file as input and apply the selected image filter to produce an output bitmap image file. The architecture is composed of four main components: the input module, the filter module, the output module, and the CUDA kernel module.

The input module reads the bitmap image file and converts it into a two-dimensional array of pixel values. The filter module receives the input array and applies the selected image filter to produce a filtered array. The output module writes the filtered array to an output bitmap image file.The CUDA kernel module is responsible for parallelizing the filtering process using CUDA.

FLOW CHART


Start Program
Input file opened successfully ?
Display Error message
Read BMP file header and extract the image dimension
Allocate memory on GPU
Read BMP image data into buffer
Copy image data from host to device
Give user choice to choose filter
Configure the grid and block dimension for kernel launch
Launch Colour Inversion kernel on GPU
Copy processes image data from device to host
Write processed data into new BMP file
End Program
Configure the grid and block dimension for kernel launch
Launch Gaussian Blur kernel on GPU
Copy processes image data from device to host
Write processed data into new BMP file
End Program
Configure the grid and block dimension for kernel launch
Launch contrast adjuster kernel on GPU
Copy processes image data from device to host
Write processed data into new BMP file
End Program
Configure the grid and block dimension for kernel launch
Launch Brightness adjuster kernel on GPU
Copy processes image data from device to host
Write processed data into new BMP file
End Program

KERNEL CONFIGURATION SYNTAX

```
// Assuming size is the total number of pixels in the image
dim3 gridDim((size + 512 - 1) / 512, 1,1);
dim3 blockDim(512,1,1);

// Launch kernel function based on user-selected filter
if (userChoice == 1) {
        colourInversion<<<gridDim, blockDim>>>(inputImage, outputImage, width, height, [other parameters]);
} else if (userChoice == 2) {
    gaussianBlur<<<gridDim, blockDim>>>(inputImage, outputImage, width, height, [other parameters]);
} else if (userChoice == 3) {
        contrast<<<gridDim, blockDim>>>(inputImage, outputImage, width, height, [other parameters]);
} else if (userChoice == 4) {
      brightness<<<gridDim, blockDim>>>(inputImage, outputImage, width, height, [other parameters]);
}
```

In this syntax, we calculate the grid dimension as (size + blockDim - 1) / blockDim, which computes the minimum number of blocks required to cover all the pixels in the image. The block dimension is set to blockDim, blockDim, which is 512 for efficient memory access. The inputImage and outputImage parameters are pointers to device memory allocated using cudaMalloc, which point to the input and output buffers, respectively. These pointers are passed as arguments to the kernel functions. The [other parameters] in the function calls represent any additional parameters that the kernel function may require for its operation.

Based on the user's choice of filter, the appropriate kernel function is launched using the <<<gridDim, blockDim>>> syntax. For example, if the user chooses the Gaussian blur filter, we launch the gaussianBlur kernel function with the specified grid and block dimensions, and pass the input and output image buffers along with any other required parameters.

PSEUDO KERNEL METHOD CODE

- **Colour Inversion:**

```
global function colourInversion(buffer, outBuf):
        i = blockDim.x * blockIdx.x + threadIdx.x;

redIdx = i * 3;
greenIdx = i * 3 + 1;
blueIdx = i * 3 + 2;

outBuf[redIdx] = 255 - buffer[redIdx];
outBuf[greenIdx] = 255 - buffer[greenIdx];
outBuf[blueIdx] = 255 - buffer[blueIdx];
```

- **Gaussian Blur:**

```
global function gaussianBlur(buffer, outBuf):
        i = blockDim.x * blockIdx.x + threadIdx.x;

redIdx = i * 3;
greenIdx = i * 3 + 1;
blueIdx = i * 3 + 2;

halfKernelSize = kernelSize / 2;

for y in range(-halfKernelSize, halfKernelSize+1):
        for x in range(-halfKernelSize, halfKernelSize+1):
                idx = (i + y * width + x) * 3
                        if idx >= 0 and idx < width * height * 3:
                                sumRed += buffer[idx]
                                sumGreen += buffer[idx + 1]
                                sumBlue += buffer[idx + 2]
                                count += 1
```

- **Adjust Contrast:**

```
global function adjustContrast(buffer, outBuf,contrast):
    i = blockDim.x * blockIdx.x + threadIdx.x;

    redIdx = i * 3;
    greenIdx = i * 3 + 1;
    blueIdx = i * 3 + 2;

    r = ((float)buffer[redIdx] / 255.0 - 0.5) * contrast + 0.5;
    g = ((float)buffer[greenIdx] / 255.0 - 0.5) * contrast + 0.5;
    b = ((float)buffer[blueIdx] / 255.0 - 0.5) * contrast + 0.5;

    outBuf[redIdx] = (unsigned char)r;
    outBuf[greenIdx] = (unsigned char)g;
    outBuf[blueIdx] = (unsigned char)b;
```

- **Adjust Brightness:**

```
global function adjustBrightness(buffer, outBuf,brightness):
        i = blockDim.x * blockIdx.x + threadIdx.x;

redIdx = i * 3;
greenIdx = i * 3 + 1;
blueIdx = i * 3 + 2;

r = buffer[redIdx] + brightness;
g = buffer[greenIdx] + brightness;
b = buffer[blueIdx] + brightness;

outBuf[redIdx] = r;
outBuf[greenIdx] = g;
outBuf[blueIdx] = b;
```

EXPERIMENTAL SETUP

Platform: Google Colaboratory (Colab)

CPU: Intel Xeon Processor @ 2.30 GHz (or equivalent)

GPU: NVIDIA Tesla K80 (or equivalent)

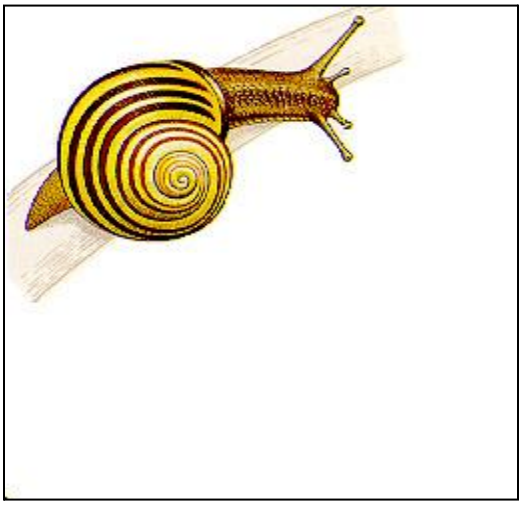
In this setup, we use Google Colab as our platform for running the iFilter project. Colab provides a free environment for running machine learning and other data-intensive applications, and comes with a built-in GPU that can be used for acceleration of CUDA-based applications.

For our CPU, we assume an Intel Xeon processor running at 2.30 GHz or equivalent. This is a typical CPU that can be found in many modern computing systems, and is used for running the host code and managing data transfers between the host and device.

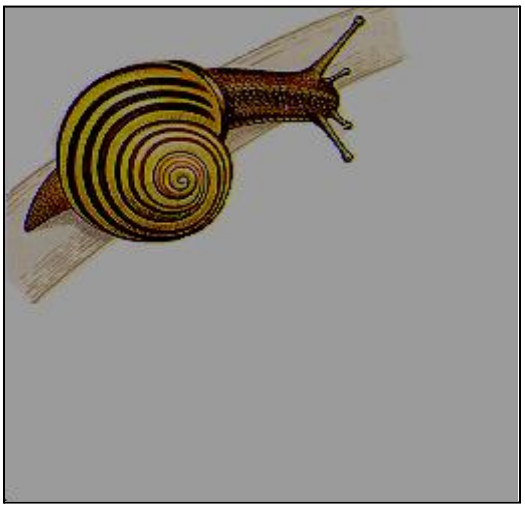
For our GPU, we assume an NVIDIA Tesla K80 or equivalent. This is a high-performance GPU that is commonly used for deep learning and other data-intensive applications, and is capable of providing significant speedup for CUDA-based applications such as iFilter.

To run the iFilter project on this setup, we first upload our input bitmap file to Colab's file system, and then run the project code using the provided kernel configuration syntax. The resulting output bitmap file is then downloaded back to our local system for further analysis or use.

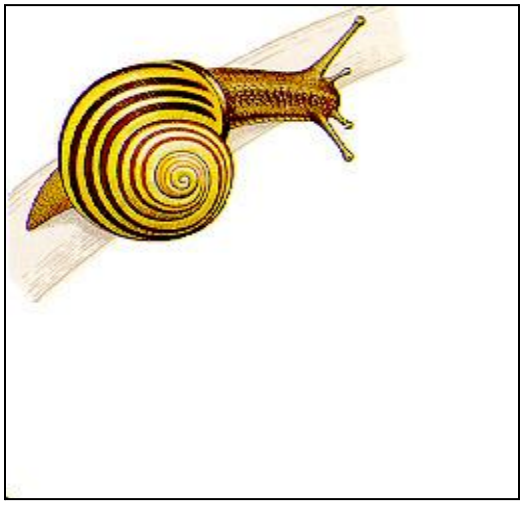
RESULTS



Input Image

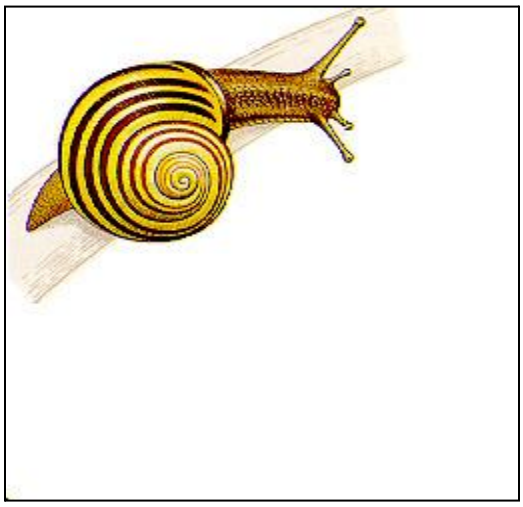


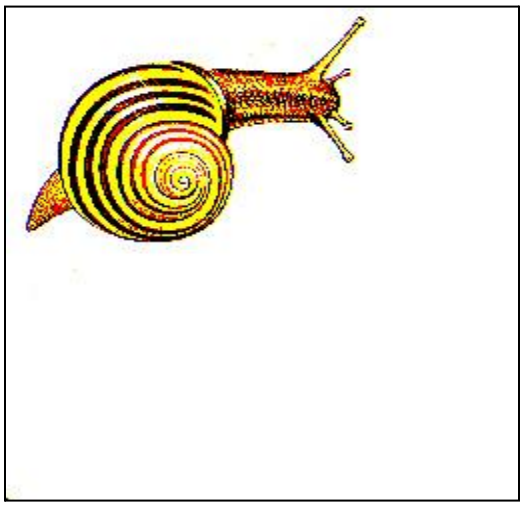
Lowering Brightness



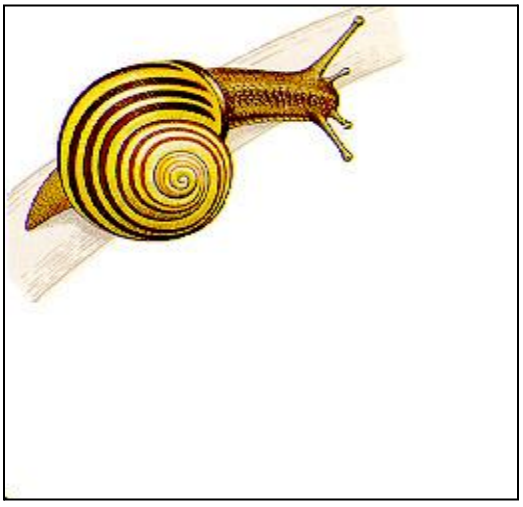
Input Image

Colour Inversion





Input Image **Increasing Contrast**



Input Image **Gaussian Blur**

****