HW6: CUDA Image Processing

R. Kumar*
*Department of Mechanical Engineering, University of Tennessee at Chattanooga Chattanooga, TN USA

Abstract

This report presents the implementation of fundamental image processing operations accelerated using CUDA on GPU. Input images are processed in the PPM format, and four operations are demonstrated: Gaussian blur, edge detection, sine-wave image generation, and a custom vignette effect. Each operation is parallelized by assigning one CUDA thread per pixel, enabling efficient computation for high-resolution images. The report details the workflow from reading and writing PPM images, launching GPU kernels, and converting the results to standard formats such as PNG for visualization. Results on sample images demonstrate the effectiveness of GPU acceleration in producing visually distinct outputs while highlighting the advantages of parallel processing for real-time or large-scale image manipulation.

I. Introduction

Image processing is now a critical technology, driving advancements in fields from medical diagnostics and autonomous systems to social media and satellite analysis. However, the increasing resolution and complexity of digital imagery demand immense computational power, often exceeding the capabilities of traditional CPU-based processing, especially for real-time video or large batch operations. The parallel architecture of GPUs offers a powerful solution. Designed to execute thousands of operations simultaneously, GPUs are ideally suited for image processing tasks where each pixel can be processed by an independent thread. This enables dramatic speedups for computationally intensive operations like filtering, edge detection, and feature extraction.

This report explores the application of GPU computing via CUDA to accelerate fundamental image processing techniques. Using the simple Portable Pixmap (PPM) format for reading and writing, we implement a suite of operations—including Gaussian blur, gradient-based edge detection, sine-wave pattern generation, and vignette effects—through custom CUDA kernels. The results are then converted to standard formats like PNG for visualization. This study not only demonstrates practical image manipulation but also serves as a foundation for understanding the principles of GPU parallelism and its transformative impact on visual computing.

II. Methodology

The workflow for this report consists of reading or generating image data, launching CUDA kernels with one thread per pixel, and writing the processed results back to a PPM file. All kernels use two-dimensional grid and block configurations such that each CUDA thread corresponds to a pixel at coordinates (x, y). The host code is responsible for memory allocation, CPU–GPU data transfer, kernel execution, and saving the final output. Four distinct image-processing operations were implemented as separate CUDA kernels, with a menu-driven main program determining which operation to execute.

A. Gaussian Blur

For the Gaussian blur operation, an input PPM image is loaded and copied to the GPU. Each thread computes the blurred intensity of its assigned pixel using a weighted average of its neighboring pixels within a configurable window. A Gaussian-like kernel assigns larger weights to central pixels and smaller weights to surrounding pixels. The blur strength and the number of iterations are user-controlled, and multiple blur passes are performed by re-launching the blur kernel and feeding each output back as input. The final blurred image is written to a PPM file. The GPU kernel for gaussian blur is given below

```
__global__ void blur_kernel(int w, int h, float weight,
                            const float* r_in,const float* g_in,const float* b_in,
                            float* r_out,float* g_out,float* b_out)
{
    int x = blockIdx.x * blockDim.x + threadIdx.x;
    int y = blockIdx.y * blockDim.y + threadIdx.y;
    if(x>=w || y>=h) return;

    int idx = y*w + x;

    float wc = weight;
    float wn = (1.0f - weight)/4.0f;

    int xm1 = max(x-1,0);
    int xp1 = min(x+1,w-1);
    int ym1 = max(y-1,0);
    int yp1 = min(y+1,h-1);
```

```
    r_out[idx] = wc*r_in[idx] + wn*(r_in[y*w+xm1] + r_in[y*w+xp1] + r_in[ym1*w+x] + r_in[yp1*w+
        x]);
    g_out[idx] = wc*g_in[idx] + wn*(g_in[y*w+xm1] + g_in[y*w+xp1] + g_in[ym1*w+x] + g_in[yp1*w+
        x]);
    b_out[idx] = wc*b_in[idx] + wn*(b_in[y*w+xm1] + b_in[y*w+xp1] + b_in[ym1*w+x] + b_in[yp1*w+
        x]);
}
```

The `blur_kernel` function implements a Gaussian blur on an image using CUDA, where each GPU thread processes a single pixel. The kernel parameters include the image dimensions (`w`, `h`), a weight factor (`weight`), input color channels (`r_in`, `g_in`, `b_in`), and output color channels (`r_out`, `g_out`, `b_out`).

Each thread computes its pixel coordinates (x, y) using the block and thread indices. Threads outside the image bounds immediately return to avoid invalid memory access. The linear index `idx` in the row-major pixel arrays is calculated as:

$$\texttt{idx = y * w + x}$$

The kernel computes a weighted average of the pixel's own color and its four immediate neighbors (left, right, top, bottom). The weight of the central pixel is given by `wc = weight`, while the neighbors share the remaining weight equally, `wn = (1.0 - weight)/4.0`.

Boundary conditions are handled by clamping indices to remain within valid ranges. Finally, the weighted sums are stored in the output arrays for each color channel. By launching one thread per pixel, this kernel fully utilizes the parallelism of the GPU, performing the blur operation simultaneously across all pixels for efficient execution.

B. Edge Detection

The edge-detection kernel computes the gradient magnitude at each pixel using central finite differences:

$$G_x = I(x{+}1, y) - I(x{-}1, y), \qquad G_y = I(x, y{+}1) - I(x, y{-}1).$$

The gradient magnitude,

$$\sqrt{G_x^2 + G_y^2},$$

is compared against a user-defined threshold. Pixels with magnitude above the threshold are classified as edges and set to white, while others are set to black. Border pixels are handled by clamping index values. The resulting binary edge map is saved as a PPM image. The GPU kernel for edge detection is described below:

```
    /************ Edge Detection Kernel ************/
__global__ void edge_kernel(int w,int h,float threshold,
                            const float* r_in,
                            float* r_out, float* g_out, float* b_out)
{
    int x = blockIdx.x*blockDim.x + threadIdx.x;
    int y = blockIdx.y*blockDim.y + threadIdx.y;

    if(x>=w || y>=h) return;

    int xm1 = max(x-1,0);
    int xp1 = min(x+1,w-1);
    int ym1 = max(y-1,0);
    int yp1 = min(y+1,h-1);

    float gx = 0.5f*(r_in[y*w+xp1] - r_in[y*w+xm1]);
    float gy = 0.5f*(r_in[yp1*w+x] - r_in[ym1*w+x]);
    float mag = sqrtf(gx*gx + gy*gy);

    int idx = y*w + x;

    if(mag > threshold){
        r_out[idx]=g_out[idx]=b_out[idx]=1.0f;
    }else{
        r_out[idx]=g_out[idx]=b_out[idx]=0.0f;
    }
}
```

The `edge_kernel` function detects edges in a grayscale image using a gradient-based method. Each CUDA thread processes a single pixel, with parameters including the image dimensions (`w`, `h`), a threshold value (`threshold`), input red channel (`r_in`), and output channels (`r_out`, `g_out`, `b_out`).

Each thread computes its pixel coordinates (x, y) using the block and thread indices. Threads outside the image bounds return immediately to prevent out-of-range memory access. Neighboring pixel indices are clamped at the boundaries to handle edge cases:

$$xm1 = \max(x - 1, 0), \quad xp1 = \min(x + 1, w - 1)$$
$$ym1 = \max(y - 1, 0), \quad yp1 = \min(y + 1, h - 1)$$

The horizontal (G_x) and vertical (G_y) gradients are calculated using central differences:

$$G_x = 0.5 \times (r\_in[y * w + xp1] - r\_in[y * w + xm1])$$
$$G_y = 0.5 \times (r\_in[yp1 * w + x] - r\_in[ym1 * w + x])$$

The gradient magnitude is computed as:

$$\text{mag} = \sqrt{G_x^2 + G_y^2}$$

If the magnitude exceeds the threshold, the pixel is considered part of an edge, and its output color channels are set to white (1.0); otherwise, they are set to black (0.0):

- `r_out[idx] = g_out[idx] = b_out[idx] = 1.0f` (edge)
- `r_out[idx] = g_out[idx] = b_out[idx] = 0.0f` (non-edge)

This thread-per-pixel approach ensures all pixels are processed in parallel, leveraging GPU parallelism for fast edge detection.

C. Sine-Wave Pattern Generation

In the function-generation operation, a new image is created on the GPU, and each thread assigns its pixel value based on a sinusoidal pattern across the horizontal axis. The intensity is computed as

$$I(x, y) = 0.5 \left[1 + \sin\left(\frac{2\pi x}{\text{period}} \right) \right],$$

where the period and amplitude are user-configurable. This produces a smooth sine-wave pattern across the canvas. The generated image is copied back to the CPU and stored in PPM format. The corresponding GPU kernel is described below:

```
    /************ Sine Wave Image Kernel ************/
__global__ void sine_kernel(int w,int h,float amplitude,float period,
                            float* r,float* g,float* b)
{
    int x = blockIdx.x * blockDim.x + threadIdx.x;
    int y = blockIdx.y * blockDim.y + threadIdx.y;

    if(x>=w || y>=h) return;

    float yf = 0.5f*h + amplitude * sinf(2*3.1415926f*(x/period));
    int idx = y*w + x;

    if(fabsf(y-yf) < 2.0f){
        r[idx]=0; g[idx]=1; b[idx]=0;
    } else {
        r[idx]=0; g[idx]=0; b[idx]=0;
    }
}
```

The `sine_kernel` function generates a visual sine-wave pattern on the GPU. Each CUDA thread corresponds to a single pixel at coordinates (x, y). Threads outside the image bounds immediately return to prevent out-of-range memory access.

For each pixel, the vertical position of the sine wave is computed as:

$$yf = 0.5 \cdot h + \text{amplitude} \cdot \sin\left(2\pi \frac{x}{\text{period}} \right)$$

where `amplitude` and `period` are configurable parameters controlling the wave's height and wavelength.

The pixel color is then set based on its proximity to the sine curve:

- If the pixel's vertical position y is within a small range of yf ($|y - yf| < 2$), it is colored green (`r=0, g=1, b=0`).
- Otherwise, it is colored black (`r=0, g=0, b=0`).

This approach uses a thread-per-pixel strategy, allowing all pixels to be processed simultaneously on the GPU. Once the kernel finishes execution, the image arrays for red, green, and blue channels are copied back from device memory to host memory and written to a PPM file. This implementation demonstrates a straightforward way to generate procedural images efficiently using CUDA.

D. Custom Operation: Vignette Effect

The fourth operation applies a vignette effect, where pixel brightness decreases radially toward the image boundaries. Each thread computes the normalized distance of its pixel from the image center and applies a falloff factor:

$$I'(x, y) = I(x, y) \left(1 - k\, r^2\right),$$

where r is the normalized distance from the center, and k controls the vignette strength. This produces a smooth darkening effect near the edges. The processed image is saved as a PPM file.

```
    /************ Vignette Custom Kernel ************/
__global__ void vignette_kernel(int w,int h,
                                float* r,float* g,float* b)
{
    int x = blockIdx.x*blockDim.x + threadIdx.x;
    int y = blockIdx.y*blockDim.y + threadIdx.y;
    if(x>=w || y>=h) return;

    float cx = w/2.0f;
    float cy = h/2.0f;

    float dx = (x - cx) / (w*0.5f);
    float dy = (y - cy) / (h*0.5f);

    float d = sqrtf(dx*dx + dy*dy);
    float factor = fmaxf(1.0f - d, 0.0f);

    int idx = y*w + x;
    r[idx] *= factor;
    g[idx] *= factor;
    b[idx] *= factor;
}
```

The `vignette_kernel` applies a radial darkening effect to an image. Each CUDA thread processes a single pixel at coordinates (x, y). Threads that fall outside the image bounds return immediately to avoid out-of-range memory access.

The implementation first calculates the normalized distance of the pixel from the image center (cx, cy):

$$dx = \frac{x - cx}{w/2}, \quad dy = \frac{y - cy}{h/2}, \quad d = \sqrt{dx^2 + dy^2}$$

where w and h are the image width and height.

A multiplicative factor is computed as:

$$\text{factor} = \max(1.0 - d, 0.0)$$

This factor decreases from the center of the image to the edges, producing the vignette effect.

Finally, each color channel of the pixel is scaled by `factor`:

$$r[x, y] \mathrel{*}= \text{factor}, \quad g[x, y] \mathrel{*}= \text{factor}, \quad b[x, y] \mathrel{*}= \text{factor}$$

This thread-per-pixel approach allows all pixels to be processed concurrently on the GPU. Once the kernel execution is complete, the modified image channels are copied back to host memory and stored as a PPM file.

E. Kernel Launch Configuration and Memory Handling

For all operations, the image is stored as a linear RGB array. Device memory is allocated for both input and output buffers, and host–device communication is performed using `cudaMemcpy`. Each kernel is launched using two-dimensional `dim3` grid and block dimensions, ensuring one thread per pixel. After execution, the output is copied back to the host and written using the provided PPM I/O routines. This structure ensures that all operations execute efficiently on the GPU and fully exploit parallelism across image pixels.

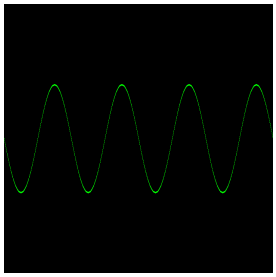
(a) Cat

(b) Dog

Fig. 1: Sample images for image processing operations

(a) Gaussian blur

(b) Edge detection



(c) Sine function generation

(d) Vignette effect

Fig. 2: Image processing operation with cat

F. Compilation and Run

Before running the CUDA program, the input images need to be converted to the PPM format. This can be done using the custom Python script `convert_to_ppm.py`, provided in Appendix A. The conversion is performed as:

```
python3 convert_to_ppm.py input.png
```

This creates an output file named `input.ppm`. Once the image is in PPM format, the CUDA program can be compiled using `nvcc`:

```
nvcc -O2 ppm-cuda.cu -o ppm-cuda
```

The compiled executable supports multiple operations, specified via command-line arguments: `blur` for Gaussian blurring, `edge` for edge detection, `sine` for sine-wave generation, and `custom` for the vignette effect. The corresponding script can be found in Appendix B. Each operation reads the input PPM image, processes it on the GPU, and writes the result to an output PPM file, e.g., `blur.ppm` or `edge.ppm`. These output files can then be converted to standard formats like PNG using ImageMagick or the Python script `ppm_to_png.py` (see Appendix C):

```
python ppm_to_png.py blur.ppm
```

This workflow ensures proper input formatting, efficient GPU execution, and convenient output visualization.

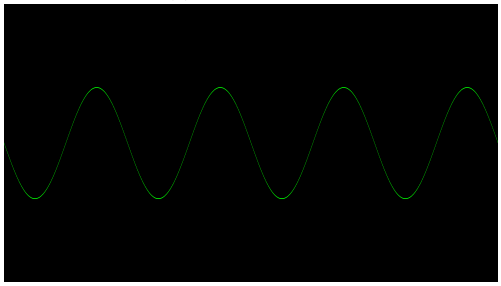
III. Results and Discussion

This section presents the results obtained from the implemented GPU-based image processing operations. Two sample images—a cat and a dog—were sourced from the internet for testing, as shown in Fig. 1. The framework was used to perform several post-processing operations, including Gaussian blur, edge detection, sine-wave pattern generation, and a vignette effect. These results demonstrate the correctness of each kernel and highlight the efficiency of performing pixel-level computations on the GPU.

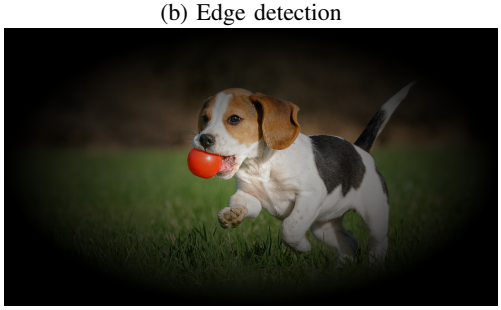
Figure 2 presents the image-processing results for the cat image. The blurred image shows a significant reduction in high-frequency details while preserving the overall structure of the scene. A blur factor of 0.2 with 40 iterations is applied, resulting in softened edges, smoothed textures, and reduced sharp color transitions due to repeated neighborhood averaging. The edge-detected image extracts regions with strong intensity gradients using a binary black-and-white representation. Prominent contours such as the outline of the cat, facial features, and fur boundaries are highlighted in white, while uniform or less textured regions

(a) Gaussian blur

(b) Edge detection



(c) Sine function generation



(d) Vignette effect

Fig. 3: Image processing operation with dog

(a) Threshold cutoff of 0.2

(b) Threshold cutoff of 0.15

(c) Threshold cutoff of 0.1

Fig. 4: Effect of threshold cutoff value on edge detection

appear black, clearly isolating the silhouette of the subject. The threshold value of gradient is set to 0.2. Increasing the threshold value reveals less features, where as lower values result in appearance of more edges.

The sine-wave image is a synthetically generated CUDA output, producing a smooth sinusoidal pattern across the image plane. The amplitude value is set to 20% of the image height and period is set to 25% of the image width. Rendered in green on a black background, the waveform displays amplitude and wavelength values defined by the user, demonstrating GPU-accelerated procedural image generation. Finally, the vignette image applies radial brightness attenuation, gradually darkening the corners while keeping the central region bright. This effect is calculated relative to the image center, generating a smooth, symmetric fall-off that naturally draws visual attention toward the middle of the photograph.

Figure 3 shows the image-processing results for the dog image. The Gaussian blur is applied using the same settings as before, resulting in a softened image where major structures remain visible but fine details are noticeably smoothed out. The edge-detection output is less successful compared to the cat image, as it does not fully capture the dog's body outline. This limitation arises from the low contrast between the dog and its background—because their intensities are similar, the image contains weaker gradients, making it difficult for the edge detector to isolate the animal's contours.

Fig. 4 shows the effect of varying the threshold parameter in edge detection. As the threshold decreases from 0.2 to 0.1, the detector becomes more sensitive, allowing finer details and weaker edges to appear. This illustrates how crucial it is to understand and properly tune such parameters, as they directly influence the clarity, accuracy, and amount of information extracted during image processing. The sine-wave and vignette operations are successfully applied. The sine-wave image displays the expected GPU-generated sinusoidal pattern, while the vignette effect introduces smooth radial darkening toward the corners, emphasizing the central region of the image.

IV. Conclusion

This report implemented Gaussian blur, edge detection, sine-wave generation, and vignette effects using CUDA to demonstrate GPU-accelerated image processing. The results show that CUDA efficiently handles pixel-level parallel operations, producing fast and consistent outputs for both sample images. Blurring and vignette effects were visually smooth, while edge detection highlighted how image contrast affects gradient-based methods. Overall, the study shows that CUDA provides a powerful and scalable platform for real-time image processing and offers a strong basis for extending these techniques to more advanced filters and computer-vision tasks.

Appendix A

```python
import os
import sys

try:
    from PIL import Image
except ImportError:
    print("Pillow is not installed in this environment.")
    print("Try loading Anaconda: module load anaconda/4.9.2")
    sys.exit(1)

def convert_image(input_path, output_path):
    """Convert a single image to binary PPM (P6)"""
    im = Image.open(input_path).convert("RGB")
    im.save(output_path, format="PPM")
    print(f"Saved: {output_path}")

def batch_convert(folder):
    """Convert all .jpg/.png/.jpeg images in folder to .ppm"""
    for fname in os.listdir(folder):
        if fname.lower().endswith(('.png', '.jpg', '.jpeg')):
            input_path = os.path.join(folder, fname)
            output_fname = os.path.splitext(fname)[0] + ".ppm"
            output_path = os.path.join(folder, output_fname)
            convert_image(input_path, output_path)

if __name__ == "__main__":
    if len(sys.argv) < 2:
        print("Usage: python3 convert_to_ppm.py <image_or_folder>")
        sys.exit(1)

    path = sys.argv[1]

    if os.path.isdir(path):
        batch_convert(path)
    elif os.path.isfile(path):
        output_path = os.path.splitext(path)[0] + ".ppm"
        convert_image(path, output_path)
    else:
        print("Error: path does not exist.")
        sys.exit(1)
```

Appendix B

```cpp
    // nvcc -O2 ppm-cuda.cu -o ppm-cuda

#include <iostream>
#include <fstream>
#include <vector>
#include <cmath>
#include <cstdio>
#include <algorithm>

/*****************************************
 * UTILITY: GPU ERROR CHECKING
 *****************************************/
```

```
#define gpuErrchk(ans) { gpuAssert((ans), __FILE__, __LINE__); }
inline void gpuAssert(cudaError_t code, const char *file, int line, bool abort=true)
{
    if(code != cudaSuccess){
        fprintf(stderr,"GPUassert: %s %s %d\n",
                cudaGetErrorString(code), file, line);
        if(abort) exit(code);
    }
}

/*****************************************
 * PPM IMAGE LOADING
 *****************************************/
char* data;

int read_ppm(std::string filename,
             int& width, int& height,
             std::vector<float>& r,
             std::vector<float>& g,
             std::vector<float>& b)
{
    std::ifstream in(filename.c_str(), std::ios::binary);
    if(!in.is_open()){
        std::cerr<<"Cannot open input "<<filename<<"\n";
        return 0;
    }

    std::string magic;
    in >> magic;
    if(magic != "P6"){
        std::cerr<<"Not a P6 PPM file\n";
        return 0;
    }

    // skip comments
    char c;
    in.get(c);
    while(c=='#'){
        while(in.get(c) && c!='\n');
        in.get(c);
    }
    in.unget();

    int maxcol;
    in >> width >> height >> maxcol;
    in.get(); // skip newline

    data = new char[width*height*3];
    in.read(data, width*height*3);
    in.close();

    r.resize(width*height);
    g.resize(width*height);
    b.resize(width*height);

    for(int i=0;i<width*height;i++){
        r[i] = ((unsigned char)data[3*i+0])/255.0f;
        g[i] = ((unsigned char)data[3*i+1])/255.0f;
        b[i] = ((unsigned char)data[3*i+2])/255.0f;
    }
    delete [] data;
    return 1;
}

/*****************************************
 * PPM IMAGE SAVING
 *****************************************/
```

```
int write_ppm(std::string filename,
              int width,int height,
              const std::vector<float>& r,
              const std::vector<float>& g,
              const std::vector<float>& b)
{
    std::ofstream out(filename.c_str(), std::ios::binary);
    if(!out.is_open()){
        std::cerr<<"Cannot open output "<<filename<<"\n";
        return 0;
    }

    out << "P6\n# CUDA OUTPUT\n" << width << " " << height << "\n255\n";
    for(int i=0;i<width*height;i++){
        out << (unsigned char)(fminf(r[i],1.0f)*255)
            << (unsigned char)(fminf(g[i],1.0f)*255)
            << (unsigned char)(fminf(b[i],1.0f)*255);
    }
    out.close();
    return 1;
}

/*****************************************
 * CUDA KERNELS
 *****************************************/

/************ Gaussian Blur kernel ************/
__global__ void blur_kernel(int w, int h, float weight,
                            const float* r_in,const float* g_in,const float* b_in,
                            float* r_out,float* g_out,float* b_out)
{
    int x = blockIdx.x * blockDim.x + threadIdx.x;
    int y = blockIdx.y * blockDim.y + threadIdx.y;
    if(x>=w || y>=h) return;

    int idx = y*w + x;

    float wc = weight;
    float wn = (1.0f - weight)/4.0f;

    int xm1 = max(x-1,0);
    int xp1 = min(x+1,w-1);
    int ym1 = max(y-1,0);
    int yp1 = min(y+1,h-1);

    r_out[idx] = wc*r_in[idx] + wn*(r_in[y*w+xm1] + r_in[y*w+xp1] + r_in[ym1*w+x] + r_in[yp1*w+
        x]);
    g_out[idx] = wc*g_in[idx] + wn*(g_in[y*w+xm1] + g_in[y*w+xp1] + g_in[ym1*w+x] + g_in[yp1*w+
        x]);
    b_out[idx] = wc*b_in[idx] + wn*(b_in[y*w+xm1] + b_in[y*w+xp1] + b_in[ym1*w+x] + b_in[yp1*w+
        x]);
}

/************ Edge Detection Kernel ************/
__global__ void edge_kernel(int w,int h,float threshold,
                            const float* r_in,
                            float* r_out, float* g_out, float* b_out)
{
    int x = blockIdx.x*blockDim.x + threadIdx.x;
    int y = blockIdx.y*blockDim.y + threadIdx.y;

    if(x>=w || y>=h) return;

    int xm1 = max(x-1,0);
    int xp1 = min(x+1,w-1);
    int ym1 = max(y-1,0);
    int yp1 = min(y+1,h-1);
```

```
144
145     float gx = 0.5f*(r_in[y*w+xp1] - r_in[y*w+xm1]);
146     float gy = 0.5f*(r_in[yp1*w+x] - r_in[ym1*w+x]);
147     float mag = sqrtf(gx*gx + gy*gy);
148
149     int idx = y*w + x;
150
151     if(mag > threshold){
152         r_out[idx]=g_out[idx]=b_out[idx]=1.0f;
153     }else{
154         r_out[idx]=g_out[idx]=b_out[idx]=0.0f;
155     }
156 }
157
158 /************ Sine Wave Image Kernel ************/
159 __global__ void sine_kernel(int w,int h,float amplitude,float period,
160                             float* r,float* g,float* b)
161 {
162     int x = blockIdx.x * blockDim.x + threadIdx.x;
163     int y = blockIdx.y * blockDim.y + threadIdx.y;
164
165     if(x>=w || y>=h) return;
166
167     float yf = 0.5f*h + amplitude * sinf(2*3.1415926f*(x/period));
168     int idx = y*w + x;
169
170     if(fabsf(y-yf) < 2.0f){
171         r[idx]=0; g[idx]=1; b[idx]=0;
172     } else {
173         r[idx]=0; g[idx]=0; b[idx]=0;
174     }
175 }
176
177 /************ Vignette Custom Kernel ************/
178 __global__ void vignette_kernel(int w,int h,
179                                 float* r,float* g,float* b)
180 {
181     int x = blockIdx.x*blockDim.x + threadIdx.x;
182     int y = blockIdx.y*blockDim.y + threadIdx.y;
183     if(x>=w || y>=h) return;
184
185     float cx = w/2.0f;
186     float cy = h/2.0f;
187
188     float dx = (x - cx) / (w*0.5f);
189     float dy = (y - cy) / (h*0.5f);
190
191     float d = sqrtf(dx*dx + dy*dy);
192     float factor = fmaxf(1.0f - d, 0.0f);
193
194     int idx = y*w + x;
195     r[idx] *= factor;
196     g[idx] *= factor;
197     b[idx] *= factor;
198 }
199
200 /*****************************************
201  * OPERATIONS
202  *****************************************/
203
204 void gaussian_blur_op()
205 {
206     int w,h;
207     std::vector<float> r,g,b;
208     read_ppm("input.ppm",w,h,r,g,b);
209
210     float *drA,*dgA,*dbA, *drB,*dgB,*dbB;
```

```
    size_t N = w*h*sizeof(float);

    cudaMalloc(&drA,N); cudaMalloc(&dgA,N); cudaMalloc(&dbA,N);
    cudaMalloc(&drB,N); cudaMalloc(&dgB,N); cudaMalloc(&dbB,N);

    cudaMemcpy(drA,r.data(),N,cudaMemcpyHostToDevice);
    cudaMemcpy(dgA,g.data(),N,cudaMemcpyHostToDevice);
    cudaMemcpy(dbA,b.data(),N,cudaMemcpyHostToDevice);

    dim3 tpb(16,16), bpg((w+15)/16,(h+15)/16);

    float weight = 0.2f;
    int iterations = 40;

    for(int i=0;i<iterations;i++){
        blur_kernel<<<bpg,tpb>>>(w,h,weight, drA,dgA,dbA, drB,dgB,dbB);
        cudaDeviceSynchronize();
        std::swap(drA,drB);
        std::swap(dgA,dgB);
        std::swap(dbA,dbB);
    }

    cudaMemcpy(r.data(),drA,N,cudaMemcpyDeviceToHost);
    cudaMemcpy(g.data(),dgA,N,cudaMemcpyDeviceToHost);
    cudaMemcpy(b.data(),dbA,N,cudaMemcpyDeviceToHost);

    write_ppm("blur.ppm",w,h,r,g,b);

    cudaFree(drA); cudaFree(dgA); cudaFree(dbA);
    cudaFree(drB); cudaFree(dgB); cudaFree(dbB);
}

void edge_detection_op()
{
    int w,h;
    std::vector<float> r,g,b;
    read_ppm("input.ppm",w,h,r,g,b);

    float *dr,*dg,*db,*dr2,*dg2,*db2;
    size_t N = w*h*sizeof(float);

    cudaMalloc(&dr,N); cudaMalloc(&dg,N); cudaMalloc(&db,N);
    cudaMalloc(&dr2,N); cudaMalloc(&dg2,N); cudaMalloc(&db2,N);

    cudaMemcpy(dr,r.data(),N,cudaMemcpyHostToDevice);

    dim3 tpb(16,16), bpg((w+15)/16,(h+15)/16);

    edge_kernel<<<bpg,tpb>>>(w,h,0.2f, dr, dr2,dg2,db2);

    cudaMemcpy(r.data(),dr2,N,cudaMemcpyDeviceToHost);
    cudaMemcpy(g.data(),dg2,N,cudaMemcpyDeviceToHost);
    cudaMemcpy(b.data(),db2,N,cudaMemcpyDeviceToHost);

    write_ppm("edge.ppm",w,h,r,g,b);

    cudaFree(dr); cudaFree(dg); cudaFree(db);
    cudaFree(dr2); cudaFree(dg2); cudaFree(db2);
}

void sine_wave_op()
{
    int w, h;
    std::vector<float> r, g, b;

    // Read input image to get dimensions
    read_ppm("input.ppm", w, h, r, g, b);
```

```
278
279     size_t N = w * h * sizeof(float);
280
281     float *dr, *dg, *db;
282     cudaMalloc(&dr, N);
283     cudaMalloc(&dg, N);
284     cudaMalloc(&db, N);
285
286     dim3 tpb(16,16);
287     dim3 bpg((w + 15) / 16, (h + 15) / 16);
288
289     // Automatically scale sine parameters based on image size
290     float amplitude = 0.20f * h;   // 20% of image height
291     float period    = 0.25f * w;   // 25% of image width
292
293     // Launch kernel
294     sine_kernel<<<bpg, tpb>>>(w, h, amplitude, period, dr, dg, db);
295
296     // Copy result back
297     cudaMemcpy(r.data(), dr, N, cudaMemcpyDeviceToHost);
298     cudaMemcpy(g.data(), dg, N, cudaMemcpyDeviceToHost);
299     cudaMemcpy(b.data(), db, N, cudaMemcpyDeviceToHost);
300
301     // Save image
302     write_ppm("sine.ppm", w, h, r, g, b);
303
304     cudaFree(dr);
305     cudaFree(dg);
306     cudaFree(db);
307 }
308
309
310 void custom_effect_op()
311 {
312     int w,h;
313     std::vector<float> r,g,b;
314     read_ppm("input.ppm",w,h,r,g,b);
315
316     size_t N = w*h*sizeof(float);
317
318     float *dr,*dg,*db;
319     cudaMalloc(&dr,N); cudaMalloc(&dg,N); cudaMalloc(&db,N);
320
321     cudaMemcpy(dr,r.data(),N,cudaMemcpyHostToDevice);
322     cudaMemcpy(dg,g.data(),N,cudaMemcpyHostToDevice);
323     cudaMemcpy(db,b.data(),N,cudaMemcpyHostToDevice);
324
325     dim3 tpb(16,16), bpg((w+15)/16,(h+15)/16);
326
327     vignette_kernel<<<bpg,tpb>>>(w,h, dr,dg,db);
328
329     cudaMemcpy(r.data(),dr,N,cudaMemcpyDeviceToHost);
330     cudaMemcpy(g.data(),dg,N,cudaMemcpyDeviceToHost);
331     cudaMemcpy(b.data(),db,N,cudaMemcpyDeviceToHost);
332
333     write_ppm("vignette.ppm",w,h,r,g,b);
334
335     cudaFree(dr); cudaFree(dg); cudaFree(db);
336 }
337
338 /*****************************************
339  * MAIN DISPATCH FUNCTION
340  *****************************************/
341 int main(int argc,char** argv)
342 {
343     if(argc<2){
344         std::cerr<<"Usage: ./ppm-cuda <blur|edge|sine|custom>\n";
```

```
        return 1;
    }

    std::string op = argv[1];
    if(op == "blur")       gaussian_blur_op();
    else if(op == "edge") edge_detection_op();
    else if(op == "sine") sine_wave_op();
    else if(op == "custom") custom_effect_op();
    else {
        std::cerr << "Unknown operation: " << op << "\n";
        return 1;
    }
    return 0;
}
```

APPENDIX C

```
    from PIL import Image
import sys
import os

if len(sys.argv) < 2:
    print("Usage: python ppm_to_png.py <input.ppm>")
    sys.exit(1)

input_file = sys.argv[1]

# Create output filename by replacing extension with .png
base_name = os.path.splitext(input_file)[0]
output_file = base_name + ".png"

try:
    # Open PPM file
    im = Image.open(input_file)

    # Save as PNG
    im.save(output_file)
    print(f"Converted {input_file} -> {output_file}")
except Exception as e:
    print(f"Error: {e}")
```