MANIPAL INSTITUTE OF TECHNOLOGY
MANIPAL
(A constituent unit of MAHE, Manipal)

PCAP Lab Mini Project Report on

Image Filtering

SUBMITTED BY

A.Jaswanth	12	210905050
Sai Mokshith	17	210905080
S.V.Pramod	54	210905332
V.Lishitha	59	210905398

Section B

Under the Guidance of:

Dr. Radhika Kamath

Department of Computer Science and Engineering
Manipal Institute of Technology, Manipal, Karnataka – 576104

April 2024

TABLE OF CONTENTS

INTRODUCTION

Image filtering is a fundamental task in image processing and computer vision, playing a crucial role in tasks such as noise reduction, edge detection, and feature extraction. As images continue to grow in size and complexity, the need for efficient algorithms capable of processing them in a timely manner becomes increasingly important. Parallel computing architectures offer a promising solution to this challenge by leveraging the computational power of GPUs. Among the various parallel computing platforms available, CUDA stands out as a robust framework for developing parallel applications on NVIDIA GPUs. This report focuses on leveraging CUDA to accelerate image filtering tasks, demonstrating its effectiveness in improving processing speeds and overall performance.

PROBLEM STATEMENT

The objective of this report is to explore the implementation of image filtering algorithms using CUDA, with a specific focus on a vivid filter technique. The primary challenge lies in efficiently parallelizing the image filtering algorithm to leverage the massively parallel architecture of GPUs while ensuring accuracy and maintaining image quality. By addressing these challenges, we aim to demonstrate the potential of CUDA in accelerating image processing tasks and overcoming the limitations of traditional CPU-based approaches.

.

METHODOLOGY :

To implement the vivid filter using CUDA, we first load the input JPEG image using the libjpeg library, extracting its width, height, and color channels. We then allocate memory for both the input and output images on both the host and device. The CUDA kernel function vivid_filter is designed to operate on individual pixels of the input image in parallel, applying the vivid filter technique to increase the intensity of each color channel. Memory management is crucial in CUDA programming, involving the allocation of device memory, data transfer between the host and device, and memory deallocation after processing.

Technology used : CUDA (Compute Unified Device Architecture)
which is a parallel computing platform and programming model developed by NVIDIA.

CODE :

```
#include <stdio.h>
#include <stdlib.h>
#include <cuda_runtime.h>
#include <jpeglib.h>
#include <time.h>
#define CHANNELS 3
#define TILE_SIZE 16
// CUDA kernel function for vivid filter
_global_ void vivid_filter_cuda(unsigned char *inputImage, unsigned char
*outputImage, int width,
int height) {
int col = blockIdx.x * blockDim.x + threadIdx.x;
int row = blockIdx.y * blockDim.y + threadIdx.y;
if (col < width && row < height) {
int pixelIndex = (row * width + col) * CHANNELS;
// Apply vivid filter by increasing intensity of each color channel
for (int c = 0; c < CHANNELS; ++c) {
float increased_intensity = inputImage[pixelIndex + c] * 1.5; // Increase intensity by
50%
// Clamp the value at 255
if (increased_intensity > 255) {
increased_intensity = 255;
}
outputImage[pixelIndex + c] = (unsigned char)increased_intensity;
}
}
}
// CPU function for vivid filter
void vivid_filter_cpu(unsigned char *inputImage, unsigned char *outputImage, int
width, int height) {
for (int row = 0; row < height; row++) {
for (int col = 0; col < width; col++) {
int pixelIndex = (row * width + col) * CHANNELS;
// Apply vivid filter by increasing intensity of each color channel
for (int c = 0; c < CHANNELS; ++c) {
float increased_intensity = inputImage[pixelIndex + c] * 1.5; // Increase intensity by
50%
// Clamp the value at 255
if (increased_intensity > 255) {
increased_intensity = 255;
```

```
}
outputImage[pixelIndex + c] = (unsigned char)increased_intensity;
}
}
}
}
// Function to load a JPEG image using libjpeg
unsigned char *load_jpeg(const char *filename, int *width, int *height) {
FILE *infile = fopen(filename, "rb");
if (!infile) {
fprintf(stderr, "Error opening file: %s\n", filename);
return NULL;
}
struct jpeg_decompress_struct cinfo;
struct jpeg_error_mgr jerr;
cinfo.err = jpeg_std_error(&jerr);
jpeg_create_decompress(&cinfo);
jpeg_stdio_src(&cinfo, infile);
jpeg_read_header(&cinfo, TRUE);
cinfo.out_color_space = JCS_RGB; // Specify the color space to be RGB
jpeg_start_decompress(&cinfo);
*width = cinfo.output_width;
*height = cinfo.output_height;
unsigned char *buffer = (unsigned char *)malloc((*width) * (*height) *
CHANNELS *
sizeof(unsigned char));
while (cinfo.output_scanline < cinfo.output_height) {
unsigned char *row_pointer = buffer + (cinfo.output_scanline * (*width) *
CHANNELS);
jpeg_read_scanlines(&cinfo, &row_pointer, 1);
}
jpeg_finish_decompress(&cinfo);
jpeg_destroy_decompress(&cinfo);
fclose(infile);
return buffer;
}
// Function to save a processed image as a JPEG using libjpeg
void save_jpeg(const char *filename, unsigned char *image_data, int width, int
height) {
FILE *outfile = fopen(filename, "wb");
if (!outfile) {
fprintf(stderr, "Error opening file: %s\n", filename);
return;
```

```
}
struct jpeg_compress_struct cinfo;
struct jpeg_error_mgr jerr;
cinfo.err = jpeg_std_error(&jerr);
jpeg_create_compress(&cinfo);
jpeg_stdio_dest(&cinfo, outfile);
cinfo.image_width = width;
cinfo.image_height = height;
cinfo.input_components = CHANNELS;
cinfo.in_color_space = JCS_RGB;
jpeg_set_defaults(&cinfo);
jpeg_set_quality(&cinfo, 100, TRUE);
jpeg_start_compress(&cinfo, TRUE);
size_t row_stride = width * CHANNELS;
while (cinfo.next_scanline < cinfo.image_height) {
unsigned char *row_pointer = &image_data[cinfo.next_scanline * row_stride];
jpeg_write_scanlines(&cinfo, &row_pointer, 1);
}
jpeg_finish_compress(&cinfo);
jpeg_destroy_compress(&cinfo);
fclose(outfile);
}
// Main function
int main() {
// File paths for input and output JPEG images
5const char *inputFilePath = "input.jpg";
const char *outputFilePathCPU = "output_cpu.jpg";
const char *outputFilePathGPU = "output_gpu.jpg";
// Variables to hold image width, height, and channels
int width, height;
// Load the input JPEG image using libjpeg
unsigned char *inputImage = load_jpeg(inputFilePath, &width, &height);
if (!inputImage) {
fprintf(stderr, "Failed to load input image.\n");
return 1;
}
// Allocate memory for the output images on the host
unsigned char *outputImageCPU = (unsigned char *)malloc(width * height *
CHANNELS *
sizeof(unsigned char));
unsigned char *outputImageGPU = (unsigned char *)malloc(width * height *
```

```
CHANNELS * sizeof(unsigned char));
if (!outputImageCPU || !outputImageGPU) {
fprintf(stderr, "Failed to allocate memory for output images.\n");
free(inputImage);
return 1;
}
// Measure CPU execution time for vivid filter
clock_t cpu_start = clock();
vivid_filter_cpu(inputImage, outputImageCPU, width, height);
clock_t cpu_end = clock();
double cpu_time = (double)(cpu_end - cpu_start) / CLOCKS_PER_SEC;
printf("CPU Execution Time: %.6f seconds\n", cpu_time);
// Save the CPU processed image as a JPEG using libjpeg
save_jpeg(outputFilePathCPU, outputImageCPU, width, height);
// Allocate memory for the input and output images on the device
unsigned char *d_inputImage, *d_outputImage;
cudaError_t err;
err = cudaMalloc((void **)&d_inputImage, width * height * CHANNELS *
sizeof(unsigned char));
if (err != cudaSuccess) {
fprintf(stderr, "Failed to allocate device memory for input image: %s\n",
cudaGetErrorString(err));
free(inputImage);
free(outputImageCPU);
free(outputImageGPU);
return 1;
}
err = cudaMalloc((void **)&d_outputImage, width * height * CHANNELS *
sizeof(unsigned char));
if (err != cudaSuccess) {
fprintf(stderr, "Failed to allocate device memory for output image: %s\n",
cudaGetErrorString(err));
cudaFree(d_inputImage);
free(inputImage);
free(outputImageCPU);
free(outputImageGPU);
return 1;
}
// Copy the input image from the host to the device
cudaMemcpy(d_inputImage, inputImage, width * height * CHANNELS *
sizeof(unsigned char),
cudaMemcpyHostToDevice);
// Define grid and block dimensions
```

```
dim3 threadsPerBlock(TILE_SIZE, TILE_SIZE);
dim3 numBlocks((width + TILE_SIZE - 1) / TILE_SIZE, (height + TILE_SIZE - 1)
/ TILE_SIZE);
// Measure CUDA kernel execution time
cudaEvent_t start, end;
cudaEventCreate(&start);
cudaEventCreate(&end);
cudaEventRecord(start);
// Launch the CUDA kernel
vivid_filter_cuda<<<numBlocks, threadsPerBlock>>>(d_inputImage,
d_outputImage, width, height);
// Check for kernel launch errors
err = cudaGetLastError();
if (err != cudaSuccess) {
fprintf(stderr, "CUDA kernel launch failed: %s\n", cudaGetErrorString(err));
cudaFree(d_inputImage);
cudaFree(d_outputImage);
free(inputImage);
free(outputImageCPU);
free(outputImageGPU);
return 1;
}
// Synchronize to ensure kernel execution is finished
cudaDeviceSynchronize();
cudaEventRecord(end);
cudaEventSynchronize(end);
float cuda_time;
cudaEventElapsedTime(&cuda_time, start, end);
printf("CUDA Execution Time: %.6f seconds\n", cuda_time / 1000.0);
// Copy the output image from the device back to the host
cudaMemcpy(outputImageGPU, d_outputImage, width * height * CHANNELS *
sizeof(unsigned char),
cudaMemcpyDeviceToHost);
// Free device memory
cudaFree(d_inputImage);
cudaFree(d_outputImage);
// Save the processed GPU image as a JPEG using libjpeg
save_jpeg(outputFilePathGPU, outputImageGPU, width, height);
// Free host memory
free(inputImage);
free(outputImageCPU);
free(outputImageGPU);
printf("Processing complete.\n");
```

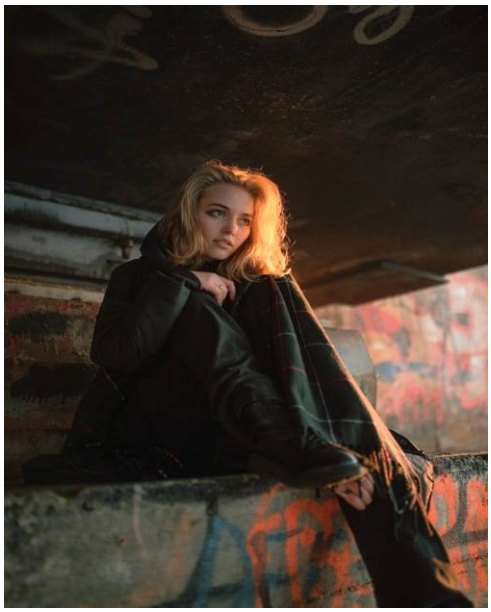
```
printf("Output CPU image saved as: %s\n", outputFilePathCPU);
printf("Output GPU image saved as: %s\n", outputFilePathGPU);
return 0;
}
```

RESULTS & SNAPSHOTS:

The results of applying the vivid filter using CUDA are visually represented through snapshots comparing the original input image with the filtered output image. These snapshots demonstrate the effectiveness of the CUDA-accelerated image filtering algorithm in enhancing the visual appearance of the image while maintaining its overall structure and details. Performance benchmarks indicate significant speedup compared to a sequential CPU-based implementation, highlighting the efficiency gains achieved through parallelization.

INPUT IMAGE:

OUTPUT IMAGE:



LOGIC FOR EXECUTION :

Speedup = Execution time of CPU/Execution time of GPU

EXECUTION TIME :

```
%%script bash
nvcc -o kernels ./src/new.cu -ljpeg
./kernels

CPU Execution Time: 0.015813 seconds
CUDA Execution Time: 0.000432 seconds
Processing complete.
```

CONCLUSION :

In conclusion, this report has explored the implementation of image filtering algorithms using CUDA, focusing on the vivid filter technique as a case study. By harnessing the parallel computing capabilities of NVIDIA GPUs, CUDA enables substantial improvements in processing speeds and performance for image filtering tasks. Despite the challenges encountered, CUDA proves to be a valuable tool for accelerating image processing workflows and unlocking new possibilities for real-time applications. Looking ahead, further research and optimization efforts in CUDA-based image filtering algorithms hold promise for advancing the field of parallel image processing.