Parallel Distributed Computing

Assignment #03

Sahrish Mustafa 22i0977

Table of Contents

Question# 01: 2D Edge Detection Using OpenCL

Problem

Convolution is an operation in image processing that transforms an image by applying a kernel (filter) to extract specific features such as edges, textures, and patterns. It is widely used in edge detection, blurring, sharpening, and feature extraction in computer vision applications.

In mathematical terms, convolution of an image with a kernel is defined as:

$$O(x, y) = \sum_{i=-k}^{k} \sum_{j=-k}^{k} I(x+i, y+j) \,.\, K(i, j)$$

where:

- *O is the output image,*
- *I is the input grayscale image,*
- *K is the convolution kernel (or filter),*
- *k is the half-size of the kernel (e.g., for a* 3×3 *kernel* = 1)

This kernel detects vertical edges by computing intensity differences between left and right neighboring pixels. The resulting image enhances edges while suppressing uniform intensity regions.

Edge Detection Using Convolution

Edge detection is a technique used to identify regions in an image where pixel intensity changes abruptly. A vertical edge detection kernel highlights edges where horizontal intensity transitions occur. A common 3×3 vertical edge detection filter is the **Left Sobel Filter,** as shown below:

$$\begin{bmatrix} 1 & 0 & -1 \\ 1 & 0 & -1 \\ 1 & 0 & -1 \end{bmatrix}$$

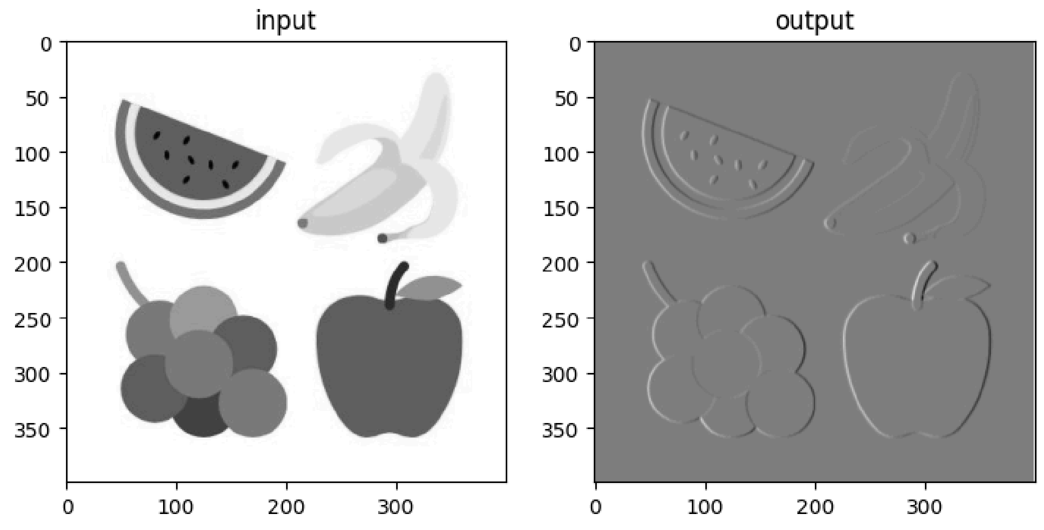
This kernel is used in all the following implementations. Changing the values inside the kernel will allow different types of edges to be detected.

An example from an article posted on [8th Light]demonstrating how it works is given below:

$$\overset{\textbf{Input with padding}}{\begin{array}{|c|c|c|c|c|c|}\hline 0&0&0&0&0&0\\\hline 0&2&3&1&4&0\\\hline 0&3&1&3&2&0\\\hline 0&3&0&1&3&0\\\hline 0&0&2&0&1&0\\\hline 0&0&0&0&0&0\\\hline\end{array}} \quad \text{conv} \quad \overset{\textbf{kernel}}{\begin{array}{|c|c|c|}\hline 1&0&-1\\\hline 1&0&-1\\\hline 1&0&-1\\\hline\end{array}} \quad = \quad \overset{\text{Output of original size}}{\begin{array}{|c|c|c|c|}\hline -4&1&-2&4\\\hline -4&\boxed{3}&-5&5\\\hline -3&2&-3&4\\\hline -2&2&-2&1\\\hline\end{array}}$$

```
I[x + i , y + j] x K[i ,  j] =  (2 x 1) + (0 x 3) + (1 x -1) + (3 x
1) + (1 x 0) + (3 x -1) + (3 x 1) + (0 x 0) + (1 x -1) = 3 = O[x ,
y]
```

Convolution example: left_sobel



The values in the output are normalized back to the range between 0-255.

Dataset Details

The dataset consists of two folders named “512” and “1024”. The first contains 2,281 images from the AFHQ dataset of cats and dogs, of size 512x512. The latter consists of 1,000 images of human faces from the SLHQ dataset, as well as another 1,000 images of landscapes from the LHQ dataset, both of which are of size 1024x1024.

These images have been converted into grayscale images using a python script (which inturn uses OpenCV). This is so that a simple 2D convolution can be applied onto the images. Using coloured images requires #D convolutions as a coloured image comprises of 3 different channels.

Part A - Scalar Implementation of Image Convolution

Scalar Implementation Overview

The scalar (non-SIMD) implementation iterates over each pixel in the image and apply the convolution kernel serially. The algorithm follows these steps:

- Load Image: Read a grayscale image and store pixel intensities in a 2D floating-point array.
- Apply Zero-Padding: Extend the image borders to handle edge cases.

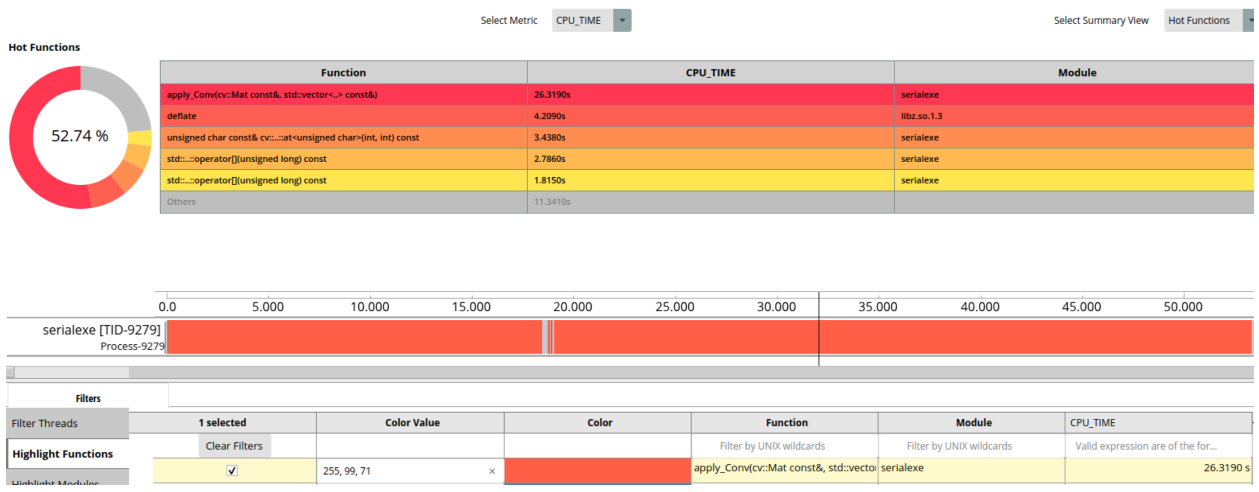
When performing convolution, the kernel (filter) slides over the image, computing a new pixel value for each position. However, when the kernel reaches the borders of the image, there are fewer surrounding pixels to use in the computation.Padding helps to maintain the original image dimensions by adding extra pixels around the border.

- Perform Convolution
  - Iterate over each pixel in the image.
  - Extract a local neighborhood corresponding to the kernel size.
  - Perform element-wise multiplication with the kernel.
  - Sum the results and assign them to the output image.
- Write the Output Image: Save the convolved image.

Details on results are verification can be found here.

Computationally Expensive Sections (Identified via Profiling)

(Image size 512x512)



Function	CPU_TIME	Module
apply_Conv(cv::Mat const&, std::vector<..> const&)	26.3190s	serialexe
deflate	4.2090s	libz.so.1.3
unsigned char const& cv::..::at<unsigned char>(int, int) const	3.4380s	serialexe
std::..::operator[](unsigned long) const	2.7860s	serialexe
std::..::operator[](unsigned long) const	1.8150s	serialexe
Others	11.3410s	

1 selected	Color Value	Color	Function	Module	CPU_TIME
Clear Filters			Filter by UNIX wildcards	Filter by UNIX wildcards	Valid expression are of the for...
✓	255, 99, 71 ×	🟥	apply_Conv(cv::Mat const&, std::vecto	serialexe	26.3190 s

Although it has already been mentioned in the assignment that the convolution has to be parallelized, it is routine to check the computational expense of each function part of the program to identify it.

Part B - SIMD-Optimized OpenCL Implementation

Accelerator (GPU) Specifications

Integrated AMD Radeon GPU	5000 series
Architecture	RDNA 3
Max Clock Speed	1800MHz
Compute Units	7
Max Work Group Size	1024
Architecture	RDNA 3

OpenCL Implementation Overview

Currently, there are two implementations that exist which utilize the built in GPU using OpenCL. The first one uses the global memory of the GPU only, whereas the second one uses global and shared (local) memory as well. Both implementations differ slightly in both the host and device code.

For simplicity, let's refer to the first implementation as Global, and the second as Shared. Both implementations use the same convolution type for vertical edges.

OpenCL implementation #01: Global Memory usage

1. Image is loaded into the memory.
2. The image is padded **on the HOST.**
3. The kernel is created and built.
4. Initializes OpenCL platform, device, and command queue.
5. Allocates memory buffers on the device for images and kernel.
6. Sets kernel arguments and executes the convolution kernel using global memory.
7. Reads back the processed output from the device.
8. Uses NDRange (globalSize/localSize) to distribute computations to threads amongst work-groups.
9. Each work item (thread) processes a portion of the image, generating a single pixel of the output image.
10. Optimizes performance using BLOCKSIZE (set to 16x16).
11. Saves the processed images in a designated output folder.
12. Measures and reports the total processing time.

Threads working on pixels at the boundary of the image will take advantage of the padded image and calculate the convolution.

OpenCL implementation #02: Shared(local) Memory usage

Most implementational details are the same, except for:
The image is loaded into the memory and copied into buffers allocated. It is **not padded** in the host code, rather the padding is implemented in parts for every work-group inside the kernel where it uses shared memory.

Each work group loads a certain subset of the image from the global memory into the shared memory of equal size. These are tiled-regions. This reduces **global memory reads** by **reusing shared memory** among multiple work-items.

```
__local float sharedTile[BLOCKSIZE + 2][BLOCKSIZE + 2];
```

Since this transfer is also done in parallel (one thread per element), a barrier is used to synchronize the threads inside the same work-group before the computational work begins.

```
barrier(CLK_LOCAL_MEM_FENCE);
```

The memory is reusable because the stride of the kernel is 1, meaning that consecutive threads will be using most of the same memory to calculate the convolution for its centered pixel.

Padding is implemented during the transfer of data by each thread in a work-group from the global memory to the shared (local/tiled) memory.
In theory, shared memory access is a lot faster than global memory access in a GPU. However, this is only evident when the GPU itself has considerable shared memory, and generally contains more SMPs (Streaming Multi-Processors).

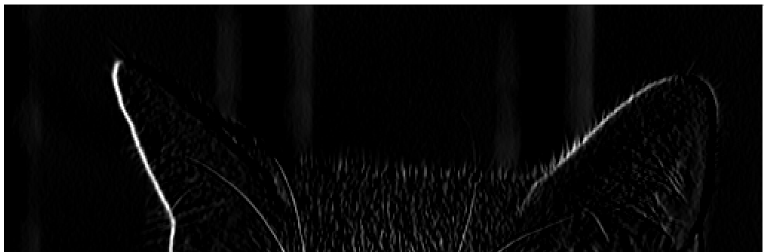
Another optimization is that the memory is accessed in a **coalesced** manner, which allows more cache hits and better performance, since the memory locations being accessed are consecutive.

Result Verification

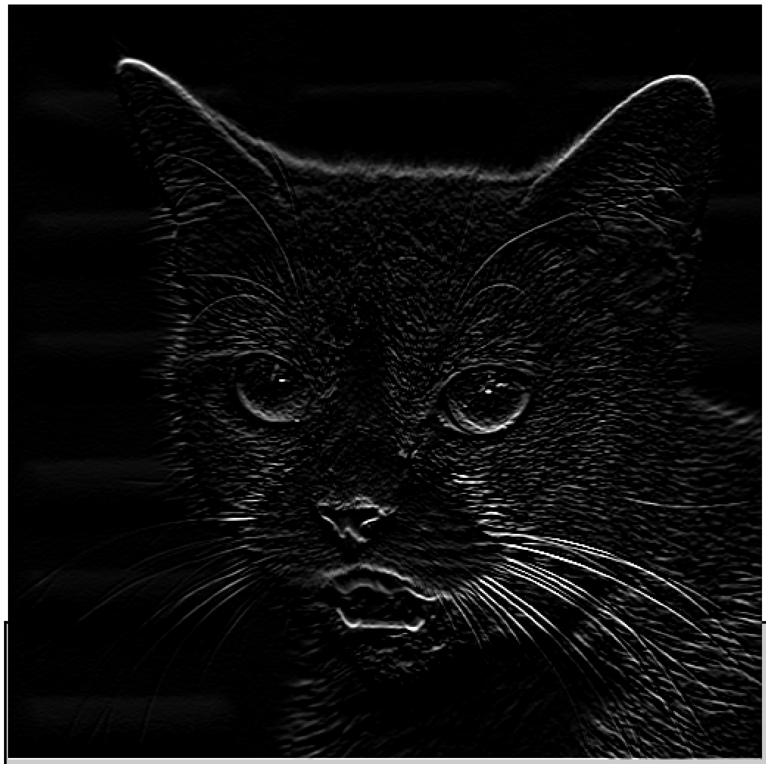
A python script uses OpenCV's and Scipy built-in functions to calculate the convolutions and compare them to my own results. The comparison is done by finding the Mean-Square Error. MSE measures the average squared difference between the pixels of the two images. Lower values indicate that the images are more similar.

Here is an example:
Original Image:

Serial - Result



SIMD - Result



```
Comparison with
OpenCV:
 - MSE: 0.012931
Comparison with SciPy:
 - MSE: 0.012518
```

Performance Analysis

It must be taken into consideration that the times are measured s**olely for convolution computation in all serial and parallel versions**. The performance, and therefore, the speedups are all **kernel-level.**

SIMD Implementation Kernel Launch

```
     // Start timing
     auto start = chrono::high_resolution_clock::now();
     err = clEnqueueNDRangeKernel(queue, clKernel, 2, NULL,
globalSize, localSize, 0, NULL, NULL);
     CHECK_ERROR(err, "Enqueuing kernel failed");
     // End timing
     auto end = chrono::high_resolution_clock::now();
     chrono::duration<double> elapsed = end - start;
     total_elapsed += elapsed;
```

All times are measured in seconds.

Number of Images	Size	CPU Execution Time	GPU Execution Time		Speed Up	
			Global Memory Only	Shared Memory	Global Memory	Shared Memory
2,281	512	35.3625	0.0216623	0.0229819	1632.444385	1538.710899
3,001	1024	204.592	0.0273009	0.0275366	7493.9654	7429.820675

Challenges and Solutions

1) Understanding Local vs. Global Memory Usage in Parallel Computing

Understanding how local memory differs from global memory in terms of access speed, usage in kernels, and its impact on performance.
In GPU computing, global memory access is slow (hundreds of clock cycles), while local memory (shared memory in CUDA/OpenCL) is much faster.

- **Global memory** is accessible by all threads but is slow.
- **Local (shared) memory** is **faster** but limited in size.
- **Registers** are the fastest but have very limited storage.

In convolution operations, using **local memory** (shared memory in CUDA) can **improve performance** by reducing redundant global memory accesses.
Tiling techniques used to store frequently used data in shared memory, minimizing global memory reads.

2) Debugging Kernel Execution

Debugging OpenCL kernels was difficult due to limited debugging tools for GPU execution. Used **printf debugging** within kernels and verified intermediate results using CPU-based implementations.

3) Getting OpenCL to Work on AMD GPU in a Virtual Machine

Tried to run OpenCL on an AMD GPU inside a **Kali Linux virtual machine**, but the GPU was not recognized.

- Checked `clinfo` but the AMD GPU was missing.
- Installed AMD OpenCL drivers but faced compatibility issues inside the VM.
- **Realized that VMs do not have direct access to the host GPU unless PCI passthrough is enabled**, which was not feasible for my setup.
- Switched to a native Linux installation instead of a VM to fully utilize OpenCL on AMD hardware.

Question# 02: Circle Construction using OpenMP

Problem

Analyze and compare the performance of serial and parallel implementations of a program that computes and plots a circle using Taylor series approximations for sine and cosine functions. The parallel version utilizes OpenMP to accelerate computations, while the serial version executes the calculations sequentially.

Serial Implementation

- Uses Taylor series approximations for sine and cosine functions.
- Computes (x, y) coordinates sequentially for angles ranging from 0 to 359 degrees.
- Uses SDL2 for graphical rendering.
- Execution time is measured using `clock()` function.

Parallel Implementation

- Employs OpenMP for parallel computation of sine and cosine values.
- Uses `#pragma omp parallel for` to parallelize the computation of:
    - Taylor series terms for sine and cosine.
        - The taylor series function uses `reduction(+:sum)` to perform summation of the terms.
    - (x, y) coordinate points.
        - This loop uses dynamic scheduling - which increases context switching overhead, but it ensures that once a thread completes its assigned task, it can take the next available iteration, leading to better load balancing.
- Execution time is measured using `omp_get_wtime()` function.

Performance Analysis

Expected Output

1. A smooth circle should be drawn using SDL2.

2. A performance comparison should demonstrate the efficiency of the parallel implementation.
3. Parallelization at two levels:
    - Computation of Taylor series terms.
    - Computation of (x, y) points.

Predefined Numbers

These numbers are kept constant across both serial and parallel implementations.

```
#define PI 3.14159265358979323846
#define TERMS 100  // Number of terms in Taylor series
#define COLS 800
#define ROWS 600
#define RADIUS 100
```

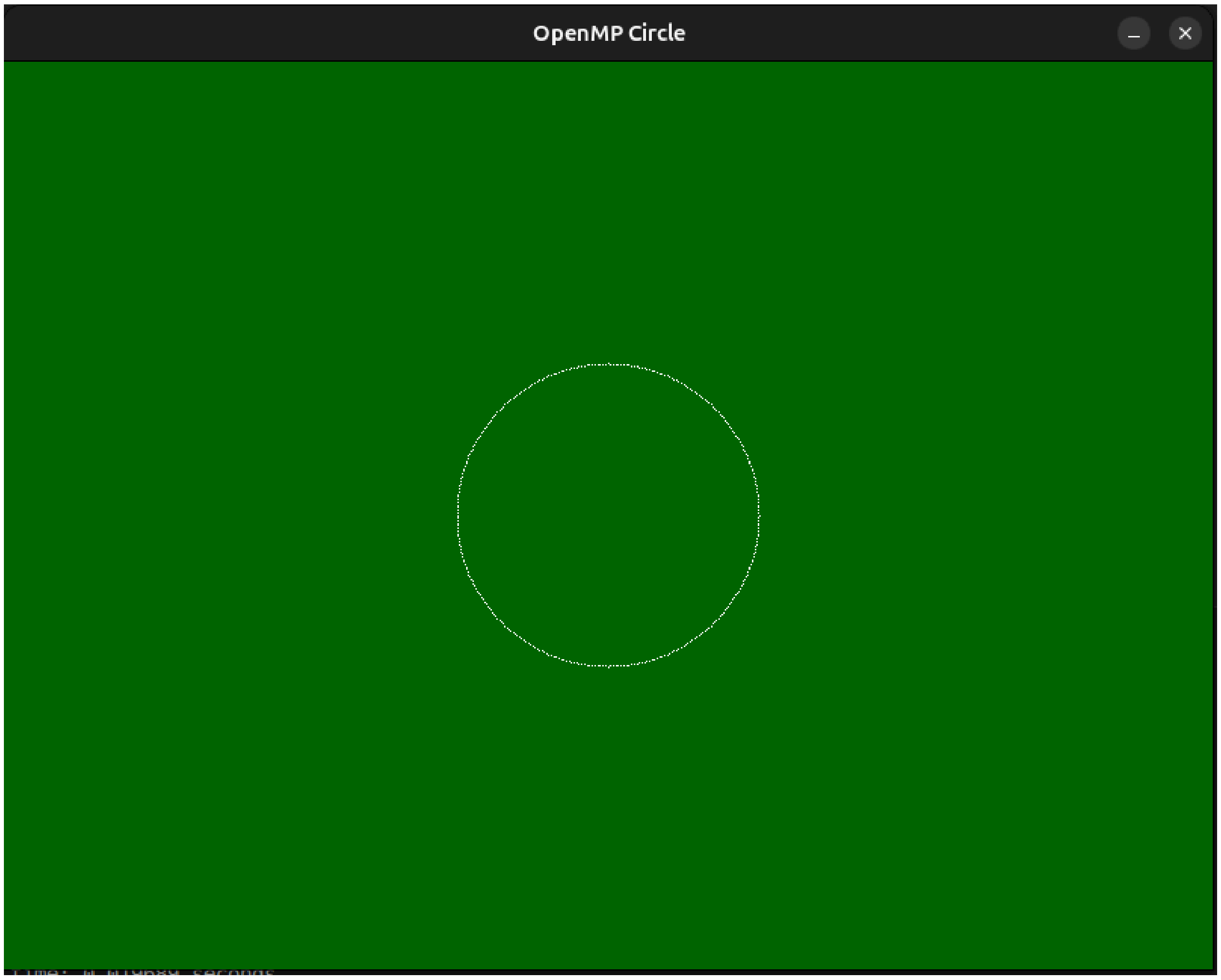
All the times below are measured in seconds.

Implementation	Execution Time (X & Y Computation)
Serial	0.021814
Parallel (OpenMP)	0.019689

SpeedUp = 1.107928285

- The speedup is not very significant. The operations inside the loop are relatively simple (basic arithmetic and memory access). The overhead of creating and managing multiple threads might outweigh or barely improve performance over the serial execution.
- Moreover, the parallelization over the second level, corresponding to each term in the taylor series, includes the `reduction(+:sum)` clause in OpenMP, ensures safe accumulation of Taylor series terms but introduces synchronization overhead, which can negate the benefits of parallel execution.
- In addition, the second parallel loop computing `x[t]` and `y[t]` is very lightweight (just multiplication and addition). This means that even in serial execution, it runs extremely fast, leaving little room for parallel optimization.

Generated Result



```
Time: [illegible] seconds
```

Precomputing factorials in parallel can improve the performance.