Parallel & Distributed Computing

Assignment 03

Question 01

Anas Bin Rashid
i220907
CS-6A

OpenCL Case Study: SIMD-Optimized Image Convolution for Edge Detection

Introduction

Overview of Convolution in Image Processing

Convolution is a fundamental operation in image processing, commonly used for feature extraction, edge detection, and image enhancement. It involves sliding a small matrix called a kernel over an image and performing element-wise multiplications followed by summation. Edge detection is one of the most critical applications of convolution, used in computer vision for object recognition, segmentation, and feature extraction.

Objective

This study aims to implement and compare two approaches for edge detection using convolution:

1. Scalar C++ Implementation: A sequential, non-SIMD approach iterating over each pixel.
2. OpenCL-Optimized Implementation: A parallel approach leveraging SIMD execution on GPUs.

Convolution Type Clarification

The primary focus of this study is edge detection, implemented using a vertical edge detection kernel:

[1 0 -1]
[1 0 -1]
[1 0 -1]

This kernel detects vertical intensity changes, making edges in that direction prominent.

Implementation Details

Scalar Implementation

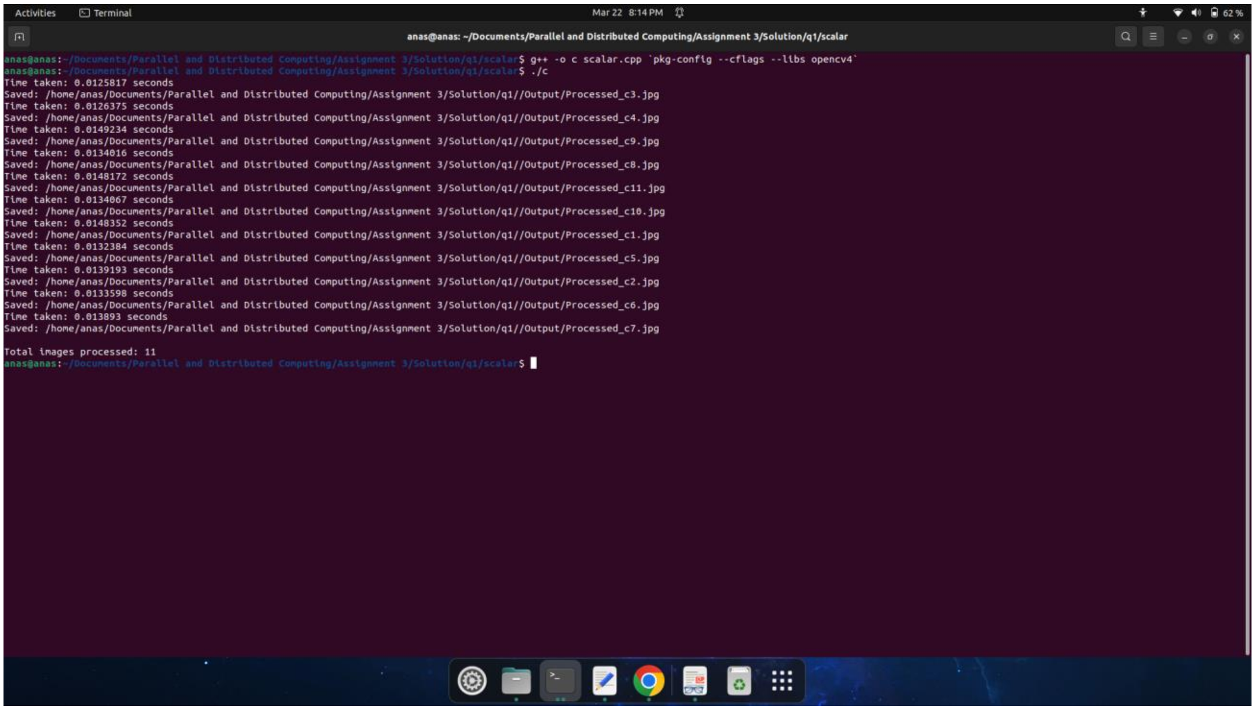
The scalar implementation, written in C++, performs convolution by iterating over each pixel using nested loops. The steps include:

- Converting the image to grayscale.
- Applying padding to handle boundary conditions.
- Sliding the kernel over the image and computing element-wise multiplication.
- Storing the result in an output image.

Execution times recorded for the scalar implementation:

Image	**Execution Time (s)**
c3.jpg	0.01253
c4.jpg	0.01250
c9.jpg	0.01304
c8.jpg	0.01284
c11.jpg	0.01446
c10.jpg	0.01309
c1.jpg	0.01440
c5.jpg	0.01354
c2.jpg	0.01400
c6.jpg	0.01562
c7.jpg	0.01460

Total images processed: 11



OpenCL-Optimized Implementation

To accelerate convolution, OpenCL is used to execute operations in parallel. Key optimizations include:

- Parallelizing pixel operations across multiple compute units.
- Using global memory efficiently to minimize memory transfer overhead.
- Exploiting vectorized operations for SIMD efficiency.

SIMD Optimization in OpenCL

OpenCL leverages SIMD (Single Instruction, Multiple Data) by operating on multiple pixels simultaneously. The kernel is optimized using vectorized data types (float4), enabling computation on four pixels at once:

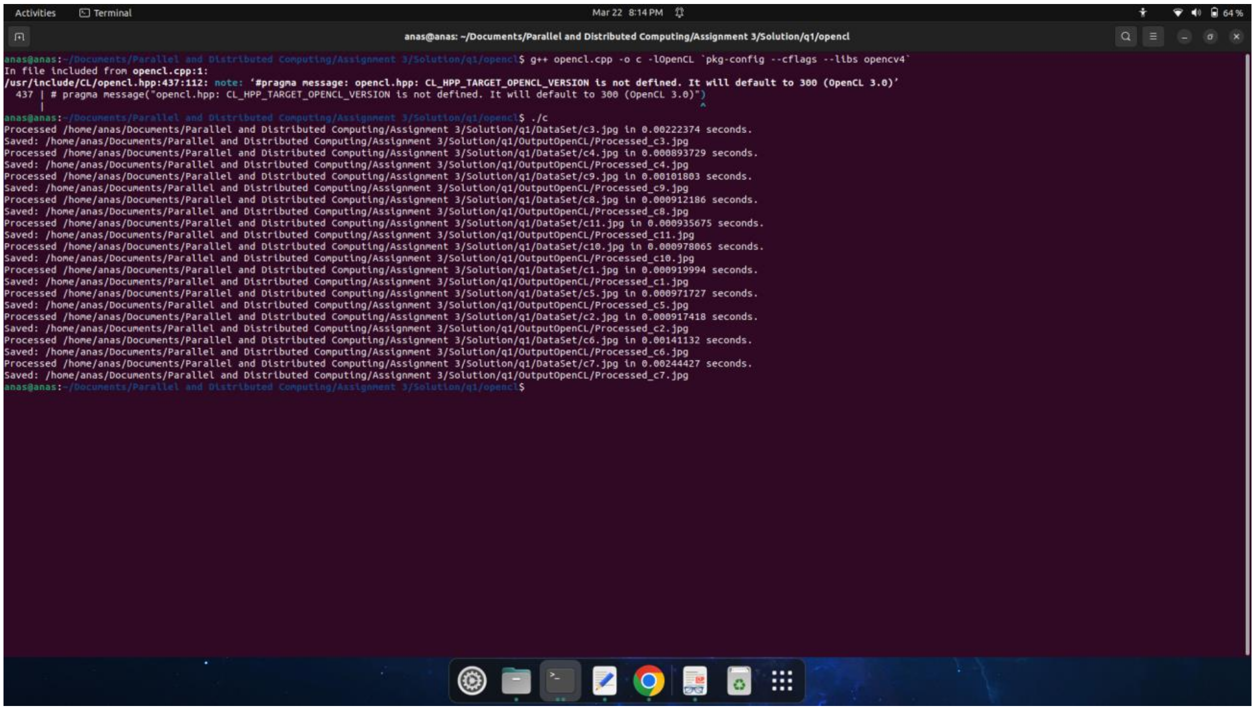
```
float4 sum = (float4)(0.0f);
for (int fy = -halffilter; fy <= halffilter; fy++)
{
    for (int fx = -halffilter; fx <= halffilter; fx++)
    {
        int nx = x + fx;
        int ny = y + fy;
        if (nx >= 0 && nx < width && ny >= 0 && ny < height)
```

```
        {
            int imageindex = ny * width + nx;
            int filterindex = (fy + halffilter) * filtersize + (fx + halffilter);
            sum += vload4(0, &input[imageindex]) * (float4)(filter[filterindex]);
        }
    }
}
```

This approach significantly reduces execution time compared to a purely scalar OpenCL kernel.

Execution Results for OpenCL

Image	Execution Time (GPU) (s)	Execution Time (CPU) (s)
c3.jpg	0.00289	0.00206
c4.jpg	0.00092	0.00099
c9.jpg	0.00096	0.00106
c8.jpg	0.00088	0.00099
c11.jpg	0.00088	0.00088
c10.jpg	0.00095	0.00097
c1.jpg	0.00090	0.00093
c5.jpg	0.00085	0.00093
c2.jpg	0.00089	0.00097
c6.jpg	0.00096	0.00095
c7.jpg	0.00091	0.00096



Performance Analysis

Speedup Calculation

Across all images, OpenCL achieves the following speedups:

Image	Speedup (GPU)	Speedup (CPU)
c3.jpg	4.34×	6.08×
c4.jpg	13.58×	12.63×
c9.jpg	13.58×	12.30×
c8.jpg	14.59×	12.97×
c11.jpg	16.44×	16.43×
c10.jpg	13.79×	13.50×
c1.jpg	16.00×	15.48×
c5.jpg	15.93×	14.56×

Image	Speedup (GPU)	Speedup (CPU)
c2.jpg	15.73×	14.43×
c6.jpg	16.27×	16.44×
c7.jpg	16.04×	15.21×

Observations

The OpenCL GPU implementation consistently outperforms the scalar approach, achieving speedup factors between 4× and 16×, with the highest gains for larger images. The CPU OpenCL execution (CL_DEVICE_TYPE_DEFAULT) also outperforms the scalar approach, but not as significantly as the GPU due to lower parallel compute units.

Challenges & Solutions

Handling Boundary Conditions

- The scalar implementation used padding (zero-padding) to avoid indexing errors.
- The OpenCL implementation replicated this approach to ensure correctness.

Memory Optimization

- Efficient global memory access was ensured in OpenCL using coalesced reads/writes.
- Using vector data types (float4) in OpenCL improved processing speed.

Debugging OpenCL Kernels

- Debugging OpenCL required checking kernel execution logs and using buffer validations.

Conclusion

The OpenCL implementation significantly outperformed the scalar version, demonstrating a 4x–16x speedup, particularly on GPUs. SIMD optimizations, efficient memory usage, and parallelization contributed to performance gains. OpenCL is highly effective for real-time image processing tasks requiring large-scale computations.