Assignment 5

Introduction to OpenCL programming.

David Padilla Orenga
Ignacio Pastore Benaim

Environment Setup

The OpenCL environment was set up on a macOS system with the following specifications:

Specification	**Value**
Platform Name	Apple
Platform Vendor	Apple
Platform Version	OpenCL 1.2 (Feb 10 2024 00:43:19)
Platform Host Timer Resolution	Not defined in OpenCL 1.2
Number of Available Devices	1
Device Name	Apple M1
Device Max Compute Units	8
Device Global Memory Cache Size	0 KB
Device Global Memory Size	5460 MB
Device Local Memory Size	32 KB
Device Max Work Group Size	256
Device Profiling Timer Resolution	1.0×10^3 nanoseconds

Table 1: OpenCL platform and device specifications

Sobel Edge Detection

Sobel Edge Detection is a fundamental image processing algorithm that detects edges by computing intensity gradients in horizontal and vertical directions. This assignment implements the algorithm in OpenCL to showcase the advantages of parallel processing.

(a) Input Image

(b) Sobel Edge Detection Output

Figure 1: Comparison of the input image and the result of Sobel Edge Detection.

Implementation Details

The implementation involves two main components:

- **Host Program:** The host program prepares input data, sets up the OpenCL environment, and manages kernel execution. It also logs performance metrics and results.
- **Kernel Program:** The kernel performs convolution on image data, leveraging global and local memory for optimization. Shared memory was used to load blocks of the image into faster memory for efficient access during computation.

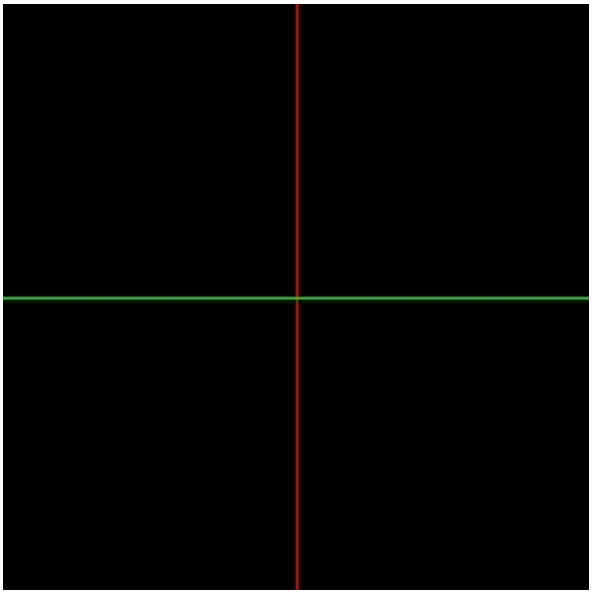
The kernel was designed to process rectangular images, dynamically adjusting the global work size to match the image dimensions while ensuring compatibility with the local size. Halo regions were carefully handled to prevent edge artifacts during convolution.

Experiment Design

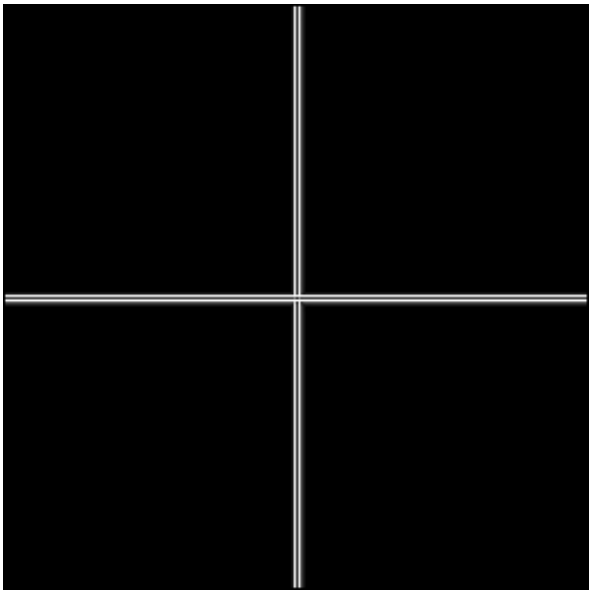
The aim of this experiment was to implement Sobel Edge Detection and analyze its performance on both GPU and CPU platforms, with the CPU serving as a baseline for comparison. The following steps outline the experimental design:

- **Input Images:** RGB images of various dimensions (256×256; 256×512; 512×512; 720×1280; 1024×1024; and 2048×4096) were generated using the CImg library. These images were converted to grayscale as input for the Sobel filter.
- **CPU Implementation:** The CPU implementation calculates the Sobel gradients sequentially, measuring both the total execution time and computation-only time for each run.
- **GPU Implementation:** The GPU implementation leverages OpenCL to parallelize the Sobel filter. Tests were conducted using different local sizes (2×2; 4×4; 8×8 and 16×16), chosen based on the device's local memory size of 32 KB, to optimize performance.

The choice of these dimensions ensures coverage of a variety of image sizes, simulating small and large-scale input data. Testing with increasing local sizes enables evaluation of GPU memory usage and processing efficiency. Input images were created with the CImg library as shown in Figure 2.



(a) Input for profiling (256x256)



(b) Sobel output for profiling (256x256)

Figure 2: Example input and output images for Sobel Edge Detection.

CPU Execution Times

CPU execution times were measured for each input image. Both the total execution time, including data preparation, and the computation-only time for applying the Sobel filter were logged.

GPU Measures

The GPU implementation was evaluated for each image size across multiple local sizes.

Metrics recorded include:

- **Total Execution Time:** Time from data preparation to result output. This is the overall time measured for the entire process, including any overhead.
- **Kernel Execution Time:** Time taken for the kernel to process the input image. This excludes the data transfer time and focuses solely on the computation.
- **Throughput:** The number of pixels processed per second, calculated using the formula:

$$\text{Throughput} = \frac{\text{Width} \times \text{Height}}{\text{Kernel Execution Time (s)}}$$

  Where:

  - Width × Height is the total number of pixels in the image.
  - Kernel Execution Time (s) is the time taken by the kernel in seconds, obtained by converting milliseconds to seconds as Kernel Execution Time (ms) $\times 10^{-3}$.

- **Memory Bandwidth:** The amount of data transferred per second during kernel execution, calculated as:

$$\text{Bandwidth} = \frac{\text{Width} \times \text{Height} \times \text{sizeof(float)}}{\text{Kernel Execution Time (s)}}$$

  Where:

  - Width × Height is the total number of pixels.
  - sizeof(float) represents the size of each pixel in bytes.
  - Kernel Execution Time (s) is the execution time in seconds.

Results and Analysis

GPU Execution Times

The figures 3 and 4 show the kernel and program execution times for different local sizes and image dimensions.

In all cases the kernel execution times decreases as the local size increases. Also, there is a clear tendency for the program execution time to decrease as the local size increases. This is expected because larger local sizes reduce the overhead of managing smaller workgroups and improve resource utilization on the GPU.

Certain configurations, such as the 1024×1024 image at a local size of 4×4 and the 256×512image at a local size of 4×4, show notable peaks in execution time. These peaks likely represent overhead introduced by suboptimal partitioning of the workload or inefficient use of GPU resources at these specific local sizes.

CPU Execution Times

The total execution time and kernel execution time for CPU implementation are presented in Figure 5.

First, we can see that the kernel execution time is the dominant factor in the total execution time, indicating that the computation is the bottleneck in the CPU implementation. Moreover, in comparison with the GPU implementation, the CPU execution times are significantly higher than the optimal GPU execution times for large scle images, highlighting the superior performance of the GPU for parallel processing tasks. We can also observe that for small images like 256×256 or 256×512, the CPU execution times are comparable to the GPU execution times, suggesting that the CPU is more efficient for smaller datasets.

Performance Analysis

Since the selected metrics only vary with the kernel execution time, we decided to display only the performance metrics for the 16×16 local size. Table 2 summarizes the performance metrics for various image dimensions.

From the table, it is evident that larger images, such as 2048×4096, exhibit higher throughput and bandwidth, highlighting the GPU's capability to efficiently handle large datasets. This demonstrates the scalability of the GPU for parallel processing tasks as the dataset size increases. In contrast, smaller images, such as 256×256, have significantly lower memory requirements but still maintain notable throughput levels. However, the bandwidth utilization is comparatively reduced for these smaller datasets, indicating that the GPU is not fully utilized under such conditions. This suggests that the GPU performs optimally with larger workloads, where its parallel processing capabilities can be fully leveraged.

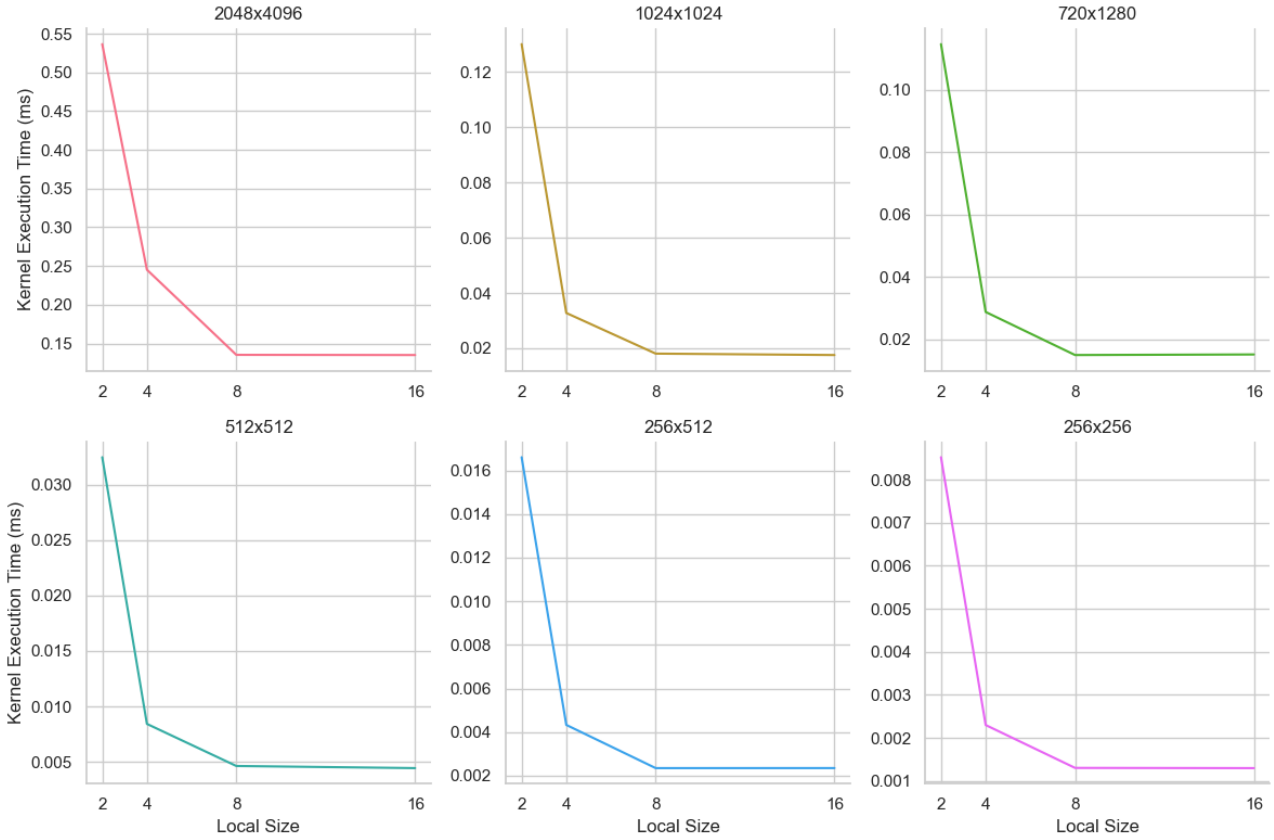


Figure 3: Kernel Execution Time vs Local Size for various image dimensions.

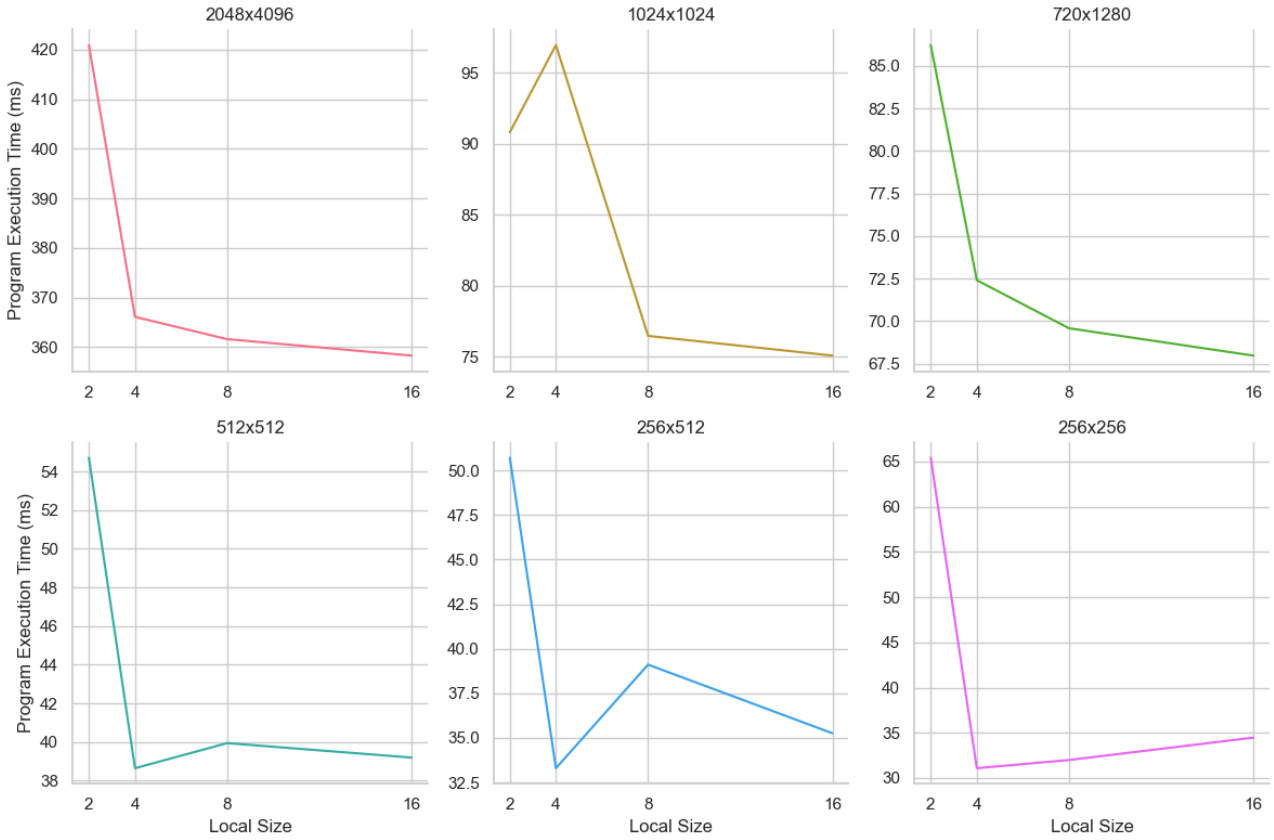


Figure 4: Program Execution Time vs Local Size for various image dimensions.

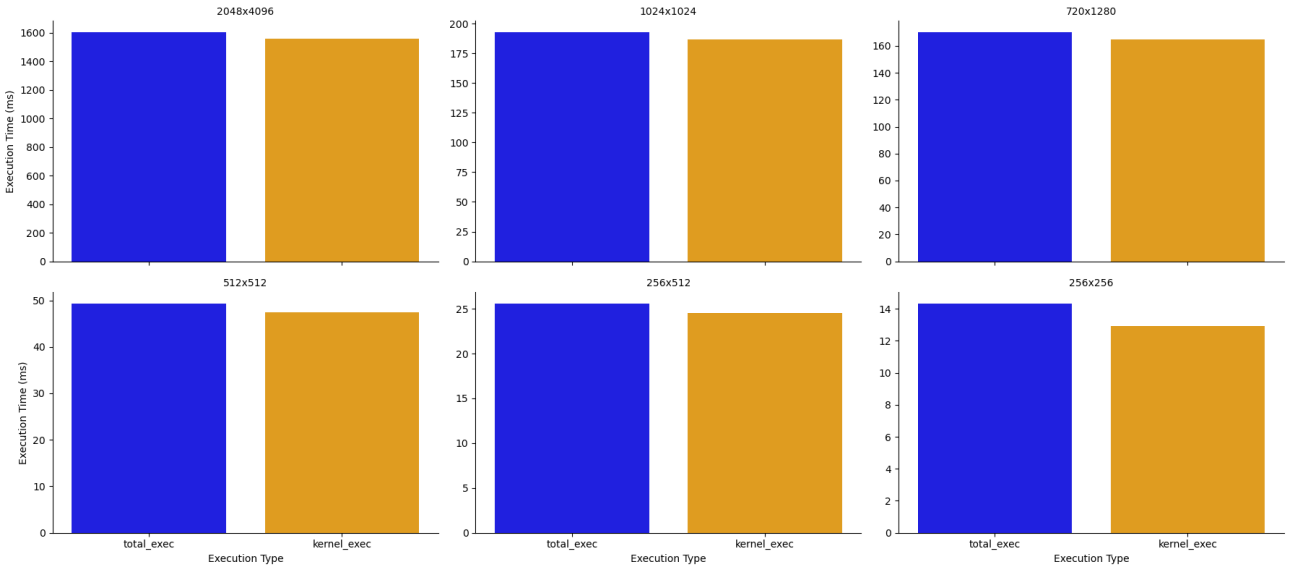


Figure 5: Comparison of Total and Kernel Execution Times for CPU.

Image Dimension	**Bandwidth (GB/s)**	**Throughput ($\times 10^9$ pixels/s)**
2048x4096	248.49	62.12
1024x1024	239.48	59.87
720x1280	242.45	60.61
512x512	236.67	59.17
256x512	223.29	55.82
256x256	202.06	50.52

Table 2: Performance metrics for various image dimensions. Bandwidth is converted to GB/s, throughput is expressed in billions of pixels per second.