GPU-Accelerated XOR Analysis: Efficient Image Differencing with CUDA

Luigi Galluccio
University of Naples ”Parthenope”
HPC: Final project

Contents

1 Introduction

Detecting differences between two images is a fundamental task in computer vision, with applications in motion detection, image forensics, medical imaging, and industrial quality control. Traditional methods for image comparison often rely on pixel-wise subtraction, which computes the absolute difference between corresponding pixels in two images. While effective in many cases, this approach can be highly sensitive to variations in lighting conditions, shadows, and sensor noise, leading to false positives or irrelevant differences being highlighted.

An alternative technique involves using the XOR (exclusive OR) logical operation, which provides a simple yet efficient way to identify changes between two images. XOR operates at the binary level, comparing corresponding pixels and setting the output to 1 (or a nonzero value) only if the pixel values differ. This means that identical regions in both images remain unchanged (black in a binary representation), while areas with differences stand out clearly. Unlike subtraction-based methods, XOR is less affected by uniform changes in brightness and can provide a more reliable way of detecting structural differences.

1.1 Project aim

In this project, we implement and analyze the effectiveness of XOR-based image differencing, first using a sequential approach in C++ with OpenCV and then accelerating the process with CUDA to leverage GPU parallelism. By offloading computations to the GPU, we aim to significantly improve processing speed, making the technique more suitable for real-time applications such as surveillance, object tracking, and automated inspection systems.

1.2 Why use CUDA for acceleration?

In real-time applications, even a single second of execution time can introduce unacceptable latency. Therefore, by using the CUDA platform to leverage the computational power of available machines, we can address these issues, reducing execution times to the order of nanoseconds.

2 Frameworks and technologies

In this section, we discuss the frameworks and technologies used in the project, explaining their role in the implementation and how they contribute to the overall performance improvement.

2.1 OpenCV for image processing

OpenCV (Open Source Computer Vision Library) is a powerful library widely used for image processing tasks. It provides efficient tools for image manipulation, filtering, and transformations. In this project, OpenCV is used for image loading, grayscale conversion, thresholding, and the saving of results.

2.2 CUDA for Parallel Computing

CUDA (Compute Unified Device Architecture) is a parallel computing platform developed by NVIDIA that allows leveraging the massive parallelism of GPUs. By using CUDA, implemented algorithms in the project can be executed much faster compared to a sequential CPU implementations, significantly reducing computation time.

2.3 Development Environment

The development environment includes the tools and hardware used to build and run the project. The code was written in C++ using Visual Studio Code, with OpenCV and CUDA for implementation, and compiled with NVCC (NVIDIA CUDA Compiler). The experiments were conducted on REDJEANS, a machine equipped with an NVIDIA GPU, ensuring optimal performance and parallel execution capabilities.

3 Sequential Implementation

3.1 Implemented Algorithms

In this section, we will look in detail at all the algorithms used in the sequential code, which will then be subsequently parallelized.

3.1.1 Binarization

Before performing the XOR difference, since it requires pixel values to be strictly binary, I implemented a binarization function that, given an input threshold, converts the image from RGB to a binary mask, ensuring that the XOR logic is correctly applied.

```
void binarization(Mat src, int th){

    for (int i = 0; i < src.rows; i++){
        for (int j = 0; j < src.cols; j++){
                int pixel = src.at<uchar>(i, j);
                if (pixel >= th)
                    src.at<uchar>(i, j) = 255;
                else
                    src.at<uchar>(i, j) = 0;
                }
        }
}
```

3.1.2 XOR difference

This function takes two binary images as input and aims to calculate the result of the XOR operation at the pixel level between the two images. The XOR operation is useful for highlighting the differences between two binary images, making visible the points where the two images do not match.

```
Mat makeXORdifference(Mat img1,Mat img2){

    Mat xor_result = Mat::zeros(img1.size(), CV_8UC1);

    for (int i = 0; i < img1.rows; i++) {
        for (int j = 0; j < img1.cols; j++) {
            uchar pixel1 = img1.at<uchar>(i, j);
            uchar pixel2 = img2.at<uchar>(i, j);
            xor_result.at<uchar>(i, j) = pixel1 ^ pixel2;
        }
    }

    return xor_result;
}
```

3.1.3 Marking differences

This function highlights the areas of change between two images, using the colors of the original image to make these differences stand out. In this way, we get an output that is much more visually interesting, even for those who are not familiar with the technique used.

```
Mat markDifferences(Mat xor_result,Mat color_src){

    Mat evidence;
    cvtColor(xor_result, evidence, COLOR_GRAY2BGR);
    for (int i = 0; i < xor_result.rows; i++) {
        for (int j = 0; j < xor_result.cols; j++) {
            if (xor_result.at<uchar>(i, j) > 0) {
                evidence.at<Vec3b>(i, j) = color_src.at<Vec3b>(i, j);
            }
        }
    }

    return evidence;
}
```

3.2 Performance Analysis

3.2.1 Test I

Test n°1 was conducted on three different definitions of the same image, showing in all three cases a good recognition of the difference between the two images, namely the absence of one of the two flower vases.

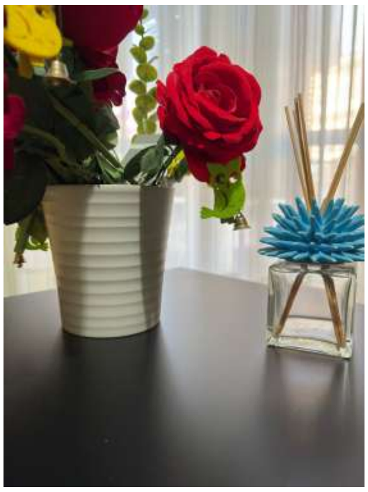
Figure 1: Input 1

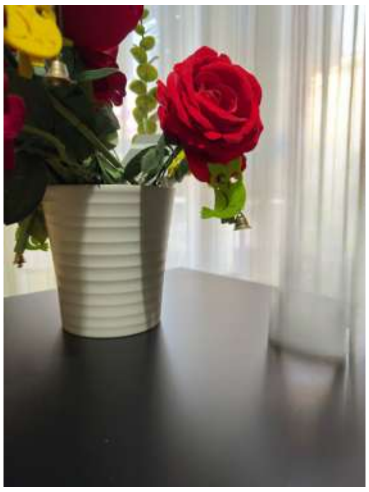
Figure 2: Input 2

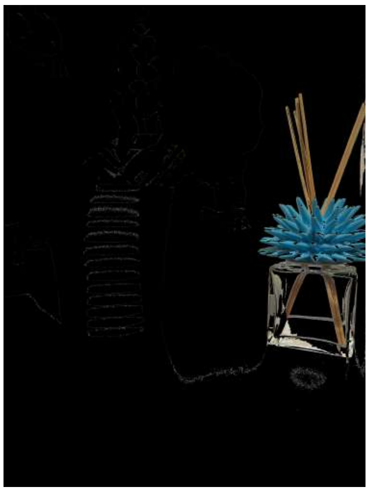
Figure 3: Output

EXECUTION TIME: 3.242795s (Max quality)

3.2.2 Test II

Test n°2 has similar dynamics and is based on recognizing the absence of an eyeglass case in the photo.

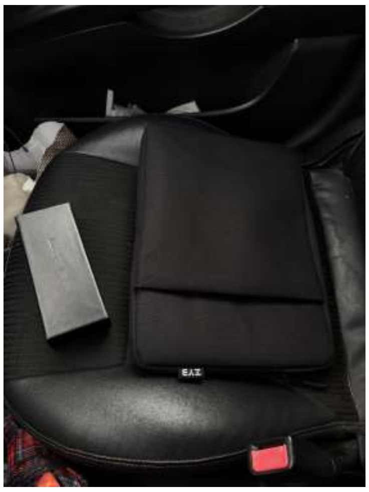
Figure 4: Input 1

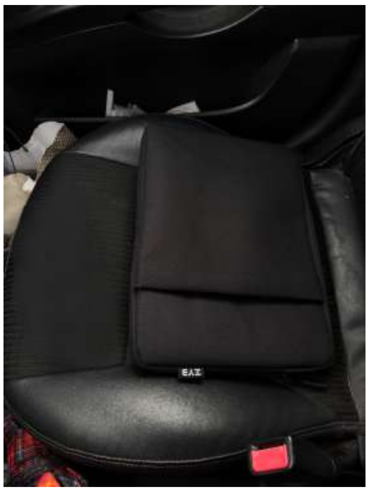
Figure 5: Input 2

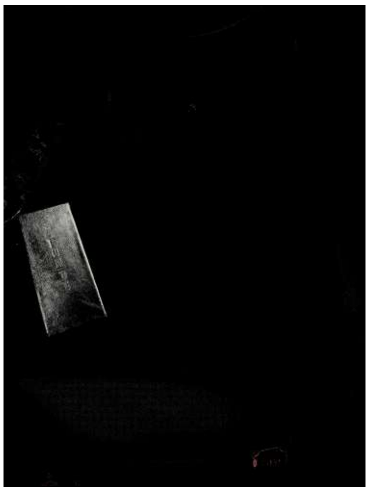
Figure 6: Output

EXECUTION TIME: 0.555316s

3.2.3 Test III

Test n°3 recognizes the absence of some balls in the background, including one that appears to be half-covered by another.

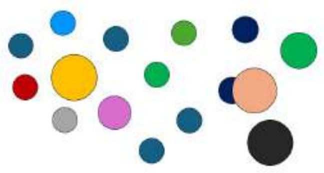
Figure 7: Input 1

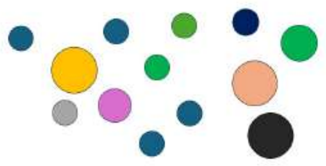
Figure 8: Input 2

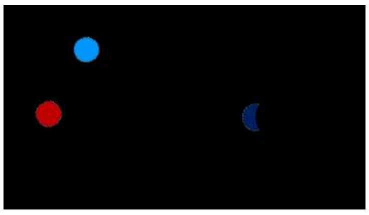
Figure 9: Output

EXECUTION TIME: 0.025380s

4 Parallel Implementation with CUDA

4.1 GPU Kernel Design

CUDA kernels are designed to distribute the computational load across multiple threads and take advantage of the architecture provided by the GPU. Each thread processes a subset of the data, thus reducing the processing time compared to a sequential execution on the CPU. Each thread executes a portion of the work and calculates its own index (`indexRow` and `indexCols`) based on the block ID and the thread ID. To prevent out-of-bounds memory access, the indices are checked to ensure they are valid with respect to the input image dimensions. Now let us look at the kernels that were used in the parallel implementation of the code.

4.1.1 Parallel binarization

```
__global__ void binarization_kernel(uchar* img,uchar* bin_result,int
    rows, int cols,int th){

    int indexRow = threadIdx.y + blockIdx.y * blockDim.y;
    int indexCols = blockIdx.x * blockDim.x + threadIdx.x;

    if(indexRow < rows && indexCols < cols){
        if(img[indexRow*cols+indexCols] >= th)
            bin_result[indexRow*cols+indexCols] = 255;
        else
            bin_result[indexRow*cols+indexCols] = 0;
    }
}

binarization_kernel<<<nBlocks, NumThreadsPerBlock>>>(dst_img1,dst_bin1,
    img1.rows,img1.cols,th);
```

4.1.2 Parallel XOR difference

```
__global__ void xor_kernel(uchar* img1, uchar* img2, uchar* xor_result,
     int rows, int cols) {

    int indexRow = threadIdx.y + blockIdx.y * blockDim.y;
    int indexCols = blockIdx.x * blockDim.x + threadIdx.x;

    if (indexRow < rows && indexCols < cols){
        int idx = indexRow*cols+indexCols;
        xor_result[idx] = img1[idx] ^ img2[idx];
    }
}

xor_kernel<<<nBlocks, NumThreadsPerBlock>>>(dst_bin1, dst_bin2,
    dst_xor_result, img1.rows, img1.cols);
```

4.1.3 Parallel marking of differences

```
__global__ void mark_differences_kernel(uchar* xor_result, uchar* color_src, uchar* evidence, int rows, int cols) {

    int indexRow = threadIdx.y + blockIdx.y * blockDim.y;
    int indexCols = blockIdx.x * blockDim.x + threadIdx.x;
    int idx = indexRow*cols+indexCols;

    if (indexRow < rows && indexCols < cols) {
        if (xor_result[idx] > 0) {
            evidence[3 * idx] = color_src[3 * idx]; //R channel
            evidence[3 * idx + 1] = color_src[3 * idx + 1]; //G channel
            evidence[3 * idx + 2] = color_src[3 * idx + 2]; //B channel
        }
    }
}

mark_differences_kernel<<<nBlocks, NumThreadsPerBlock>>>(dst_xor_result, dst_display, dst_evidence, img1.rows, img1.cols);
```

5 Comparison between Sequential and Parallel

5.1 Time Comparison

Test	Image dimension (px)	sequential (s)	Parallel (s)	Speedup
Test 1	8000 × 10666	3.24	0.000081	40000x
	1440 × 1920	0.141511	0.000114	1241x
	450 × 600	0.020964	0.000131	160x
Test 2	3072 × 4096	0.484674	0.000086	5634x
Test 3	720 × 405	0.026293	0.000097	271x

Table 1: Comparison between sequential and parallel (8 x 8 threads)

5.2 Outputs' consistency

5.2.1 Test I

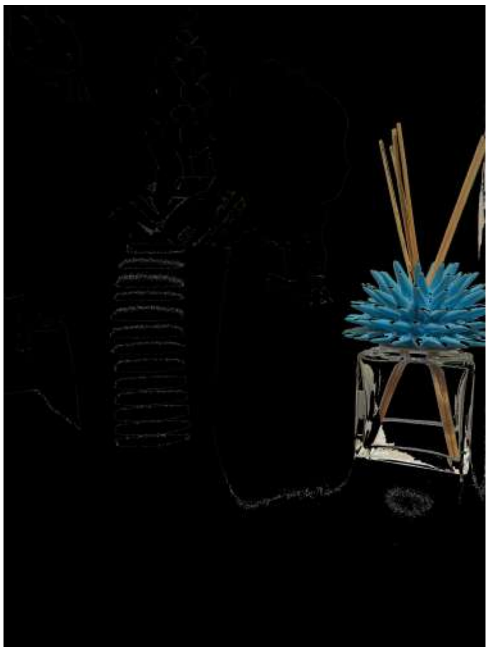

Figure 10: Sequential

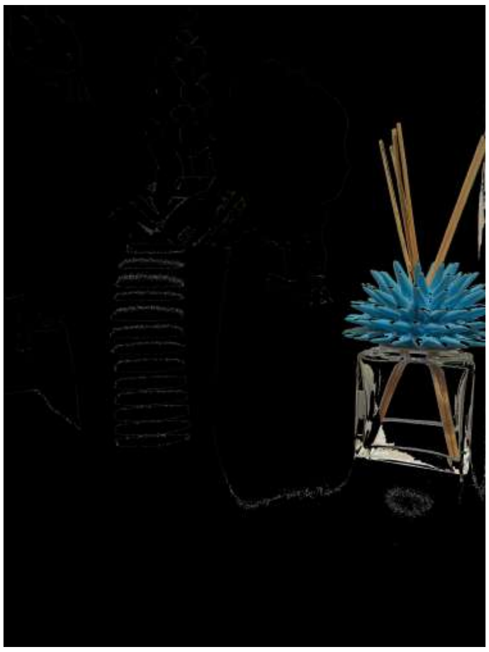

Figure 11: Parallel

5.2.2 Test II

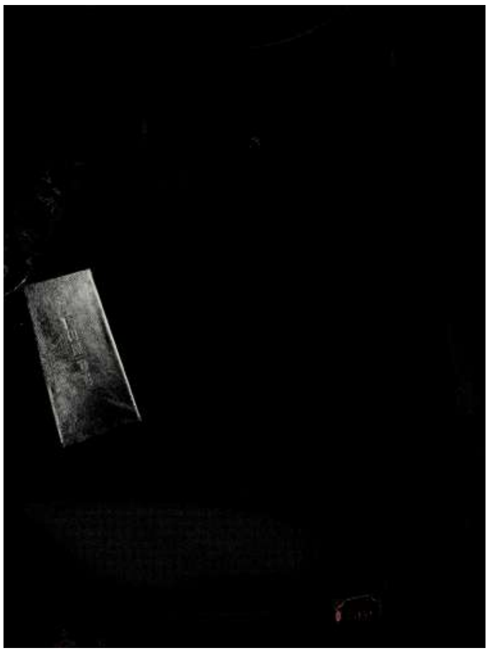
Figure 12: Sequential

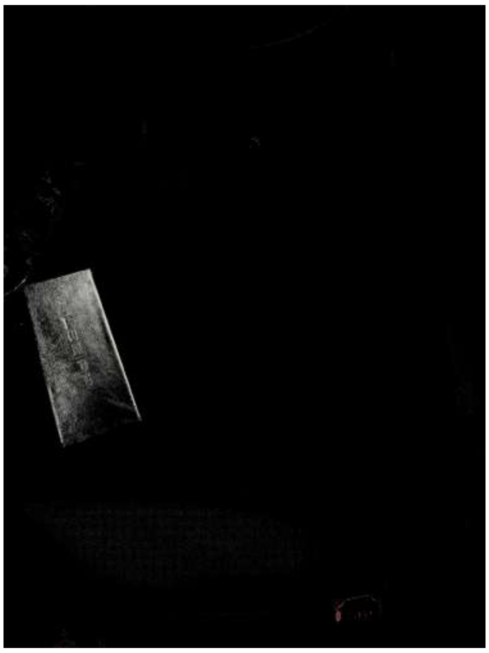
Figure 13: Parallel

5.2.3 Test III

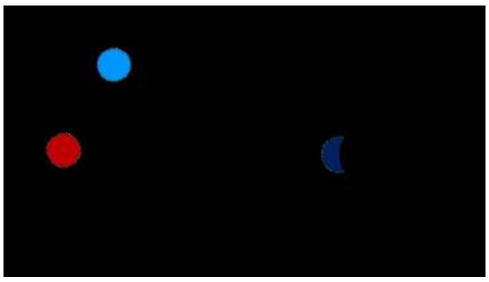
Figure 14: Sequential

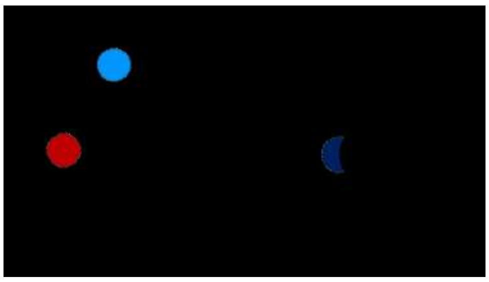
Figure 15: Parallel

6 Conclusions

The results obtained from this project clearly demonstrate the advantages of using CUDA for accelerating image difference computation using XOR logic. The key findings can be summarized as follows:

- The XOR-based difference method effectively highlights changes between images while being less sensitive to uniform lighting variations, making it a reliable approach for structural change detection.
- The parallel implementation using CUDA achieves a remarkable speedup compared to the sequential CPU implementation. As shown in Table 1, the execution time was reduced by factors ranging from 160x to over 40,000x depending on the image resolution.
- The correctness of the parallel implementation was verified by comparing outputs from both sequential and parallel approaches, ensuring that the detected differences remained consistent.
- The scalability of the CUDA implementation allows it to efficiently handle high-resolution images, making it suitable for real-time applications such as object tracking, surveillance, and automated inspection.

Overall, this project highlights the power of GPU computing for image processing tasks, demonstrating that a well-optimized CUDA implementation can provide significant performance benefits while maintaining accuracy. Future work could explore extending this approach to video sequences for real-time change detection and incorporating machine learning techniques for a more intelligent and specific difference analysis.