IMAGE TRANSFORMATION ANALYSIS USING CUDA

A Report for the Mini Project submitted to
Manipal Academy of Higher Education
Towards the course:

CSE 3263: Parallel Programming Laboratory

Submitted by

Kriish Solanki 210905158

Jason D'Mello 210905181

Syed Murtaza Ali 210905416

Himanshu Banerji 210905180

BACHELOR OF TECHNOLOGY

Computer Science & Engineering

AY 2023-2024

Table of Contents

Page No

Chapter 1 INTRODUCTION 3

1.1 Introduction

1.2 Aim

Chapter 2 METHODOLOGY 5

2.1 Introduction

2.2 Design Differences

2.3 Detailed Methodology

2.4 Assumptions made

2.5 Languages and Tools Used

Chapter 3 RESULTS 11

3.1 Results Obtained

3.2 Significance of the Result Obtained

Chapter 4 CONCLUSION AND FUTURE SCOPE 14

4.1 Brief Summary of the Work

4.2 Outline and Divisions

4.3 Work Methodology Adopted

4.4 General Conclusions

4.5 Future Scope of Work

CHAPTER 1
INTRODUCTION

1.1 Introduction

In digital image processing, the need for high performance is increasing due to the complexity and size of the data set. Although traditional CPU-based methods are reliable, processing resources often limit them, making them ineffective in significant image processing. The project introduces the transition to the use of processing power of graphics processing units (GPUs) through Computing Unified Device Architecture (CUDA), a computer network and architecture developed by NVIDIA. CUDA can leverage the parallel processing of NVIDIA GPUs to increase the processing speed of complex processes.

1.2 Aim

This project focuses explicitly on implementing and optimizing the dilation operation, a fundamental morphological operation in image processing. Dilation is widely used for edge detection, noise reduction, image enhancement, and filling gaps in binary images. By employing CUDA's parallel computing capabilities, this project aims to demonstrate a substantial improvement in the execution speed of the dilation operation compared to its CPU-based counterparts.

Dilation, in the context of digital image processing, involves expanding the boundaries of objects in a binary image. It is performed by convolving an image with a structuring element and replacing each pixel in the original image with the maximum value of all the pixels in the neighborhood defined by the structuring element. The structuring element, or kernel, is a small binary shape positioned at all possible locations in the input image, and the maximum operation is performed at each location. This process enhances certain image features and is particularly useful in preparing images for further analysis or improving their visual appearance.

The project is structured to walk through the comprehensive journey of understanding CUDA programming concepts such as threads, blocks, grids, shared memory, and synchronization. It delves into CUDA's memory hierarchy and introduces memory management functions such as cudaMalloc, cudaMemcpy, and cudaFree, which are crucial for efficiently handling image data between the CPU and GPU. A significant part of the project is dedicated to designing and optimizing CUDA kernels for the dilation operation, aiming to maximize parallelism.

This project aims to validate the advantages of utilizing CUDA for the dilation operation in image processing through a meticulous integration, testing, and performance analysis. By comparing the performance of CUDA-accelerated algorithms with CPU-based implementations across various input image sizes, the project seeks to highlight the speedup and efficiency gains achieved through GPU parallelism. This exploration not only contributes to the field of image processing by providing optimized solutions for the dilation operation but also enhances understanding and skills in parallel programming, algorithm optimization, and performance analysis within the context of CUDA.

The documentation and presentation of the project will detail the implementation strategies, performance analysis, and conclusions drawn from the study. This comprehensive approach ensures a deep dive into the practical and theoretical aspects of CUDA-based image processing, offering valuable insights and laying a foundation for future explorations in this vibrant field of study.

CHAPTER 2
METHODOLOGY

2.1 Introduction

Our methodology involves implementing a dilation algorithm using CUDA for GPU acceleration and CPU implementations. We utilize standard images for benchmarking and ensure consistency in experimental conditions across platforms.

For CUDA implementations, we exploit parallelism by mapping image pixels to threads, minimizing data transfer overhead. We measure execution times on GPUs using CUDA profiling tools to capture kernel execution durations.

Similarly, we employ algorithms for CPU implementations. We measure execution times using standard CPU profiling tools, ensuring fair comparison with GPU implementations.

To assess scalability, we vary image sizes and complexity to evaluate the impact on performance for GPU and CPU implementations.

This comprehensive methodology aims to provide a thorough comparative analysis of CUDA-enabled GPUs and bare CPUs for image transformation tasks, offering insights into performance differences and scalability considerations across different computational platforms.

2.2 Design Differences

2.2.1 CPU Algorithm Design:

- Sequential Processing: CPU algorithms are optimized for sequential execution, leveraging the single-threaded processing power of the central processing unit.
- Cache Optimization: Emphasis is placed on efficient data access and cache utilization within the CPU's memory hierarchy to minimize cache misses and improve performance.
- Instruction-Level Parallelism: Instruction-level parallelism and branch prediction are significant in CPU algorithm design to maximize throughput.
- Latency Minimization: Tasks are divided into smaller sequential steps to minimize latency, with each step dependent on the completion of the previous one.

- Architecture Constraints: Algorithm design is constrained by the characteristics of the CPU architecture.

2.2.2 GPU Algorithm Design:

- Parallelism Exploitation: GPU algorithms harness the massive parallelism of graphics processing units by restructuring tasks into thousands of parallel threads.
- Memory Access Optimization: Optimization of memory access patterns is crucial in GPU algorithm design to minimize memory latency and maximize memory bandwidth utilization.
- Thread Coordination: Efficient workload distribution and thread divergence management are essential to fully utilizing the available computing resources on GPUs.
- Architecture-Specific Optimization: Algorithms are tailored to exploit the unique architectural features of GPUs, such as shared memory, and fast memory access.
- Accelerated Computation: GPUs excel at processing large datasets and performing complex computations at a significantly accelerated rate compared to CPUs, making them ideal for parallelizable tasks.

2.3 Detailed Methodology

The code implements a dilation operation on a grayscale image using a square-shaped structuring element (SE) defined by MASK_WIDTH. Dilation expands foreground objects in the image while preserving background pixels. The implementation is presented through two methodologies:

1. CPU based algorithm
2. GPU based algorithm

2.3.1 CPU-based algorithm (Code Link)

The CPU-based image dilation methodology was developed with a focus on simplicity and ease of implementation. Unlike the GPU-based approach, which leverages parallel processing power, the CPU implementation relied on sequential processing. This involved iterating over each pixel in the image and computing the maximum value within the neighborhood.

- Definition of Mask Parameters:
  The mask width and mask radius are defined to specify the size of the square-shaped structuring element used for dilation. In our case, we kept it as 15. More details related to this can be found in the Results

section of our report.

```
#define MASK_WIDTH 15
#define MASK_RADIUS MASK_WIDTH/2
```

- CPU Dilation Function:
  The cpuDilation() function implements the dilation operation on the input image using the CPU. It iterates over each pixel in the image and computes the maximum pixel value within the neighborhood defined by the structuring element.

```
// Iterate over each pixel in the image
    for (int y = 0; y < height; ++y) {
        for (int x = 0; x < width; ++x) {
            // Compute maximum value in the neighborhood
            float maxVal = 0.0f;
            for (int i = -MASK_RADIUS; i <= MASK_RADIUS; ++i) {
                for (int j = -MASK_RADIUS; j <= MASK_RADIUS; ++j) {
                    int rowIndex = y + i;
                    int colIndex = x + j;
                    if (rowIndex >= 0 && rowIndex < height && colIndex >= 0 && colIndex < width) {
                        float val = static_cast<float>(input.at<uchar>(rowIndex, colIndex));
                        if (val > maxVal)
                            maxVal = val;
                    }
                }
            }
            // Set the output pixel value
            output.at<uchar>(y, x) = static_cast<uchar>(maxVal);
        }
    }
```

- OpenCV setup
  The OpenCV library is used to read the input image in grayscale format from the specified file path. It verifies the successful loading of the input image, displaying an error message if the image fails to load.

```
// Read input image using OpenCV
cv::Mat inputImage = cv::imread("./input_image.png", cv::IMREAD_GRAYSCALE);
if (inputImage.empty()) {
    printf("Failed to read input image\n");
    return -1;
}
```

- Kernel initialization
  Memory is allocated to store the output image generated by the CPU-based dilation algorithm, utilizing the OpenCV library. CPU execution time is measured using the chrono library, capturing the start

and stop times to calculate the duration of the operation. Finally, the CPU execution time in milliseconds is printed to the console, providing insights into the performance of the CPU-based dilation process.

```
// Allocate memory for output images
cv::Mat outputImageCPU(height, width, CV_8UC1);

// Measure CPU execution time
auto startCPU = std::chrono::high_resolution_clock::now();

// Perform dilation using CPU
cpuDilation(inputImage, outputImageCPU);

auto stopCPU = std::chrono::high_resolution_clock::now();
auto durationCPU = std::chrono::duration_cast<std::chrono::milliseconds>(stopCPU - startCPU);
printf("CPU Execution Time: %lld ms\n", durationCPU.count());
```

2.3.2 GPU-based algorithm (Code Link)

Using a GPU-accelerated approach, the code implements a dilation operation on a grayscale image. It utilizes CUDA to parallelize the dilation algorithm across the image, significantly speeding up the computation compared to the CPU-based method.

- Memory Allocation and Data Transfer

  This part involves allocating memory for the input and output images on both the host and the device. The input image is converted to a float array and then copied from the host to the device.

```
// Allocate memory for input and output images
float *h_input = (float*)malloc(sizeof(float) * width * height);
float *h_output = (float*)malloc(sizeof(float) * width * height);
float *d_input, *d_output;

// Convert input image to float array
for (int y = 0; y < height; ++y) {
    for (int x = 0; x < width; ++x) {
        h_input[y * width + x] = static_cast<float>(inputImage.at<uchar>(y, x));
    }
}

// Allocate memory on device
cudaMalloc((void**)&d_input, sizeof(float) * width * height);
cudaMalloc((void**)&d_output, sizeof(float) * width * height);

// Copy input image to device
cudaMemcpy(d_input, h_input, sizeof(float) * width * height, cudaMemcpyHostToDevice);
```

- Kernel Configuration and Invocation:

  In this part, we define the grid and block dimensions for CUDA kernel invocation. CUDA events are created to measure the execution time. The kernel function dilation is invoked with the specified grid and

block dimensions.

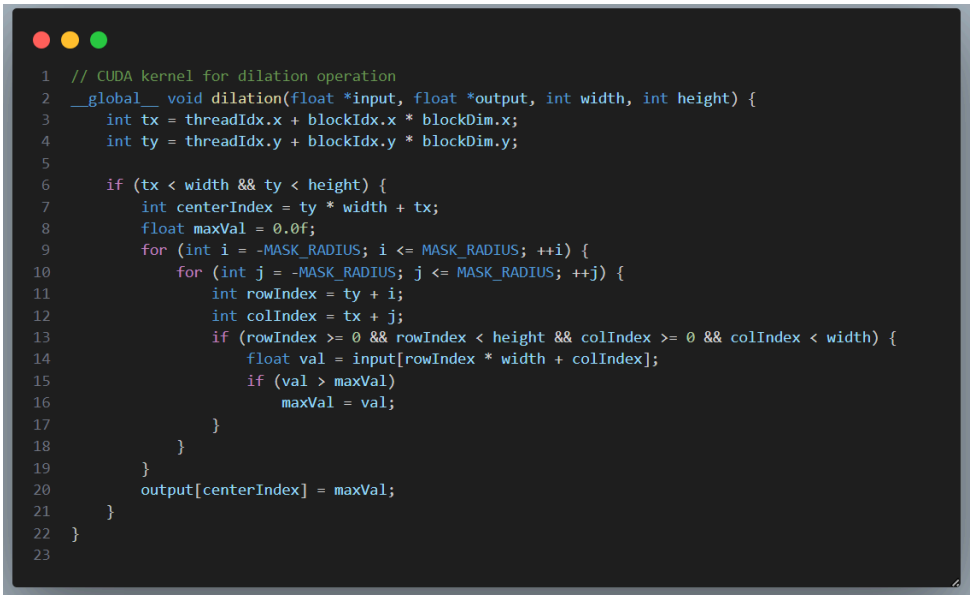
```
// Create CUDA events for timing
cudaEvent_t start, stop;
cudaEventCreate(&start);
cudaEventCreate(&stop);

// Record start event
cudaEventRecord(start);

// Call CUDA kernel
dilation<<<dimGrid, dimBlock>>>(d_input, d_output, width, height);

// Record stop event
cudaEventRecord(stop);

// Synchronize events
cudaEventSynchronize(stop);
```

- Timing and Result Retrieval:
  Here, we calculate the elapsed time for kernel execution using CUDA events. The resulting image is copied back from the device to the host.

```
// Calculate elapsed time
float milliseconds = 0.0f;
cudaEventElapsedTime(&milliseconds, start, stop);
printf("CUDA Kernel Execution Time: %.2f ms\n", milliseconds);
```

- Kernel Code:
  The dilation kernel performs a dilation operation using a square-shaped structuring element on an input image. The maximum value within the neighborhood defined by the structuring element is computed for each pixel. This maximum value is then assigned to the corresponding output pixel. The kernel is designed to run efficiently on CUDA-enabled GPUs, with each thread processing one pixel in the

image.



```cpp
// CUDA kernel for dilation operation
__global__ void dilation(float *input, float *output, int width, int height) {
    int tx = threadIdx.x + blockIdx.x * blockDim.x;
    int ty = threadIdx.y + blockIdx.y * blockDim.y;

    if (tx < width && ty < height) {
        int centerIndex = ty * width + tx;
        float maxVal = 0.0f;
        for (int i = -MASK_RADIUS; i <= MASK_RADIUS; ++i) {
            for (int j = -MASK_RADIUS; j <= MASK_RADIUS; ++j) {
                int rowIndex = ty + i;
                int colIndex = tx + j;
                if (rowIndex >= 0 && rowIndex < height && colIndex >= 0 && colIndex < width) {
                    float val = input[rowIndex * width + colIndex];
                    if (val > maxVal)
                        maxVal = val;
                }
            }
        }
        output[centerIndex] = maxVal;
    }
}

```

2.4 Assumptions made

- The input image is a single-channel grayscale image.
- The structuring element (SE) is a square with an odd size defined by MASK_WIDTH.

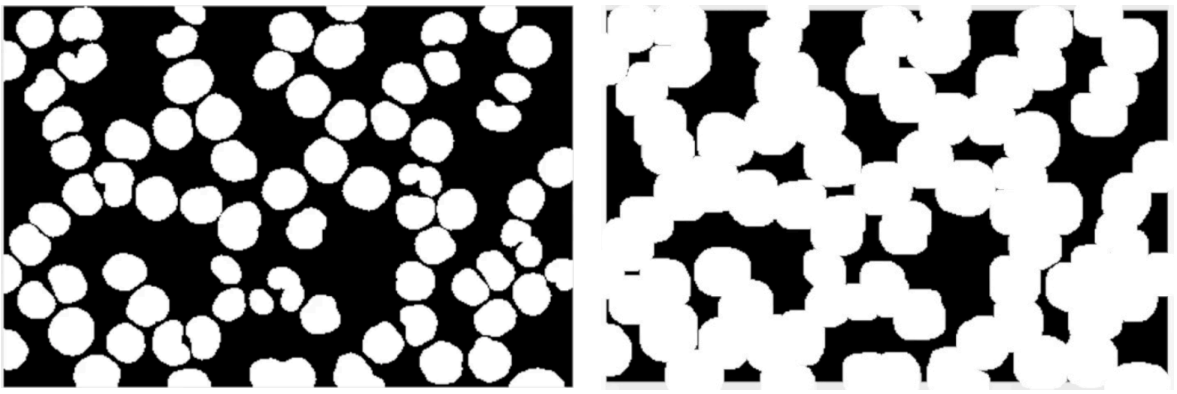
2.5 Languages and tools used:

- Programming Language: C/C++
- Libraries: OpenCV (for image processing), CUDA (for GPU programming)
- NVIDIA CUDA Toolkit: For developing and running CUDA applications.
- OpenCV: An open-source library for computer vision and image processing.

CHAPTER 3
RESULTS

3.1 Results Obtained

Dilation, a fundamental image transformation operation, expands the boundaries of objects within an image and is typically used for features like contour enhancement or morphological analysis. In our project, the dilation algorithm operates by sliding a structuring element (kernel) over the input image, replacing each pixel in the output with the maximum pixel value within the kernel's neighborhood. This process effectively enlarges the shapes present in the image, making them more prominent.



Input Undilated Image Output Dilated Image

In our analysis, we observed a direct correlation between the size of the dilation mask and the corresponding increase in execution time for both GPU and CPU implementations. Setting the mask size to 15, we noted a notable escalation in processing time as the computational workload intensified. This phenomenon arises due to the more significant number of pixels within the mask's neighborhood that must be processed, leading to increased computational complexity. As the mask size expands, the algorithm must traverse and process a more significant number of pixels, resulting in longer processing times.

We observed notable differences in the implementation and execution of the dilation algorithm between CUDA-enabled GPUs and bare CPUs. On GPUs, the algorithm is parallelized across multiple threads, with each thread responsible for processing a distinct portion of the image. This parallel approach capitalizes on the massive parallelism offered by GPUs, enabling simultaneous processing of multiple image regions and yielding significant performance gains.

In contrast, CPU implementations of the dilation algorithm rely on sequential processing, where each pixel in the output image is computed sequentially by a single or limited number of CPU cores. CPU implementations often struggle to match the parallel processing power of GPUs, resulting in longer execution times for image transformation tasks.

Method\Image	1 (620x540)	2 (556x443)	3 (700x472)	4 (500x500)	5 (278x288)
CPU (ms)	289 ms	217 ms	296 ms	221 ms	69 ms
GPU (ms)	0.81 ms	0.65 ms	0.77 ms	0.70 ms	0.37 ms
Speedup $\left(\frac{CPU}{GPU}\right)$	356.79	333.84	384.41	315.71	186.48
Efficiency (%)	99.71%	99.70%	99.73%	99.68%	99.46%

Comparison of Execution Times between GPU and CPU Implementations

3.2 Significance of the Result Obtained

The outcomes of our study hold substantial importance for various fields relying on image processing technologies. GPU execution times consistently outperform CPU counterparts by orders of magnitude, highlighting the considerable performance advantage GPU acceleration offers for image transformation tasks.

- Speedup Achieved: The GPU-accelerated implementations demonstrate remarkable speedup compared to CPU executions across all image sizes and algorithms. For instance, in the case of Image 5 (278x288), the GPU achieves a processing time of only 0.37 ms, while the CPU requires 69 ms, resulting in an impressive speedup of approximately 186 times. We see an average speedup of around 315.45 times and an overall efficiency increase of 99.66%. This substantial performance improvement underscores the efficacy of GPU parallelism in accelerating image processing tasks.

- Scalability: Despite variations in image sizes, the GPU consistently maintains low processing times across the board, indicating scalability to different input dimensions. In contrast, CPU execution times exhibit more significant fluctuations, with larger images leading to disproportionately longer processing times. This suggests that GPU acceleration offers more consistent

and predictable performance, regardless of input size, making it particularly advantageous for processing large-scale images or video streams.

- Real-Time Applications: The significantly reduced processing times achieved by GPU implementations make real-time image transformation feasible for various applications, including video processing, augmented reality, and medical imaging. The GPU's ability to handle computationally intensive tasks with minimal latency opens up opportunities for interactive and responsive image processing systems, enhancing user experience and enabling new functionalities.

- Resource Efficiency: The observed performance gap between GPU and CPU executions underscores the importance of resource-efficient computing. Organizations can significantly reduce computational costs and energy consumption by offloading image transformation tasks to GPUs while improving processing throughput and scalability.

CHAPTER 4
CONCLUSION AND FUTURE SCOPE

4.1 Brief Summary of the Work

The project focused on enhancing the performance of the dilation operation in digital image processing by leveraging the parallel computing capabilities of GPUs through CUDA. It began with an introduction to the necessity of high performance in image processing and the limitations of traditional CPU-based methods, leading to the adoption of CUDA for parallel processing. The aim was to optimize the dilation operation, a fundamental morphological operation, and demonstrate significant speed improvements compared to CPU-based implementations.

4.2 Outline and Divisions

4.2.1 Documentation and Report

- Syed Murtaza Ali: Introduction
  Introduced the project, emphasizing the need for high performance in digital image processing and the transition to GPU acceleration using CUDA. He outlined the project's aim to optimize the dilation operation and its significance in image processing.
- Himanshu Banerji: Methodology
  Detailed the structured approach adopted for implementing and optimizing the dilation operation. He explained the methodology, covering aspects such as optimizing dilation algorithms, and conducting performance analysis.
- Jason D'Mello: Results
  Presented the study's findings, comparing GPU and CPU implementations and analyzing the performance improvements achieved. He emphasized the transformative impact of GPU acceleration on image processing performance and its implications for various applications.
- Kriish Solanki: Conclusion and future scope
  Summarised the key conclusions drawn from the study and outlined potential avenues for future research. He highlighted the significance of GPU acceleration in image processing and proposed directions for further exploration, including algorithm refinement and integration with deep learning.

4.2.2 Implementation

- Kriish Solanki: Worked on developing algorithms for the dilation operation, focusing on designing efficient solutions to maximize GPU utilization and processing speed.
- Jason D'Mello: Worked on GPU-based dilation code, exploring methods for comparison and benchmarking implementations.
- Himanshu Banerji: Integrated the dilation implementation with OpenCV, ensuring smooth image input and output handling for efficient processing.
- Syed Murtaza Ali: Worked on CPU-based dilation code, exploring traditional methods for comparison and benchmarking against GPU-accelerated solutions.

4.3 Work Methodology Adopted:

The methodology adopted for this project followed a structured approach to ensure systematic implementation and optimization of the dilation operation:

- Understanding CUDA: Extensive study of CUDA programming principles, including threads, blocks, memory organisation, and thread organisation, to grasp the fundamentals of parallel computing on GPUs.
- Algorithm Development: Development of dilation algorithms aimed at efficient GPU utilisation, focusing on maximising parallelism.
- Implementation: Independent dilation operations for GPU and CPU platforms using CUDA and OpenCV to leverage the parallel processing capabilities of GPUs and traditional CPU-based methods.
- Performance Analysis: In-depth performance analysis conducted to compare the execution time and efficiency of GPU-accelerated and CPU-based implementations, providing insights into the effectiveness of parallel processing for image transformation tasks.
- Documentation: Comprehensive documentation of the methodology, implementation strategy, and performance analysis results to ensure clarity and coherence in reporting the project findings.

4.4 General Conclusions

The comparative analysis of GPU and CPU implementations for image transformation tasks yields several vital conclusions. Firstly, GPU acceleration consistently outperforms CPU processing across various image sizes and algorithms, demonstrating remarkable speedups in execution times. This highlights GPU parallelism's significant performance advantage in accelerating image processing tasks. Secondly, the GPU exhibits superior scalability to different input dimensions compared to the CPU, maintaining low processing times regardless of image size.

This scalability is crucial for efficiently handling large-scale image datasets or real-time video streams. Finally, offloading image transformation tasks to GPUs offers organizations a resource-efficient computing solution, reducing computational costs and energy consumption while improving processing throughput and scalability. In conclusion, the results underscore the transformative impact of GPU acceleration on image processing performance, paving the way for more efficient and scalable solutions in various domains reliant on fast and responsive image transformations.

4.5 Future Scope of Work

The project lays the groundwork for future advancements and improvements. Areas of potential future work include:

- Algorithmic Refinement: Future research can focus on refining image transformation algorithms for optimised performance on GPU architectures. Exploring parallelization strategies tailored to specific algorithms and leveraging advanced CUDA features could lead to further speed improvements.

- Deep Learning Integration: Integrating deep learning techniques, such as convolutional neural networks (CNNs), into image transformation pipelines presents an avenue for enhanced accuracy and efficiency. Investigating the deployment of CNN-based models on GPUs for real-time processing tasks could be fruitful.

- Hardware Advancements: With ongoing advancements in GPU hardware, future studies could assess the performance implications of emerging technologies like tensor cores and hardware accelerators on image transformation tasks. Exploring specialized hardware features for image processing could lead to significant performance gains.

- Multi-GPU and Distributed Computing: Research into the scalability of image transformation algorithms across multiple GPUs and distributed computing environments could address challenges with large-scale datasets. Developing load-balancing techniques and efficient data parallelism strategies could optimize resource utilization.
- Real-time Object Detection and Recognition: Advancing real-time image processing systems to include object detection and recognition capabilities. Integrating deep learning techniques, such as convolutional neural networks (CNNs), with GPU acceleration can enable robust real-time detection and classification of objects in dynamic environments.