1. Introduction

In this project, we aim to implement and compare serial (CPU-only) versus parallel (GPU/CUDA) algorithms for a specific image-processing task involving 16-bit grayscale images. The assignment is designed to exercise both the CPU (in a straightforward single-threaded approach) and the GPU (in a massively parallel approach) and then compare their performance under increasing image sizes.

The problem itself is structured into three main steps:

1. ***Gradient Calculation*** – Each pixel checks its 8 neighbors to determine a direction from 1 to 8, or 0 if it is a local extremum.
2. ***7×7 Neighborhood Update*** – Each pixel's direction is refined by averaging direction vectors within a 7×7 window.
3. ***Path Tracing*** – Every pixel repeatedly follows its direction until it reaches a boundary or a zero-direction pixel, storing only the start and end coordinates of that path.

After implementing both solutions, I tested their performance under increasing image dimensions (from 16×16 up to 16384×16384) and also compared two different tile sizes (256 vs. 1024) for the CUDA version. This report details the design decisions, implementation approaches, and performance results.

2. Design and Implementation

2.1 Overall Algorithm

Data Input and Processing

I load a **16-bit grayscale image** of dimension N×N. The algorithm then performs:

1. ***Step 1: Calculate Gradient Directions***
   - For each pixel, if its intensity is lighter (i.e., smaller numerical value) than all 8 neighbors, assign 0. Otherwise, choose 1..8 depending on which neighbor has the smallest intensity.
2. ***Step 2: 7×7 Neighborhood Computation***
   - For each pixel, accumulate the directions of pixels within a 7×7 window centered on (x,y). Direction codes (1..8) are converted into (dx, dy) unit vectors and summed.
   - The vector (sumX,sumY) is converted to an angle via atan2. That angle is mapped to the nearest direction code 1..8, or 0 if the vector magnitude is negligible.

3. ***Step 3: Path Tracing***
    - Each pixel (x, y) starts at (startX, startY) = (x, y). I follow the direction codes step-by-step until:
        1. direction = 0, or
        2. out of image bounds, or
        3. a step limit (to prevent infinite cycles).
    - Only (startX, startY, endX, endY) is stored for each pixel, reducing memory usage compared to storing every step in the path.

Software Environment

- **Operating System**: Windows 11
- **IDE**: Visual Studio 2022
- **CUDA Version**: 12.6
- **Compiler**: nvcc (for the CUDA parts), MSVC for the C++ host code.

We also rely on OpenCV for basic image I/O (loading 16-bit images, etc.).

2.2 Serial Implementation Details

The serial (CPU-only) version is written in C++ without multithreading:

1. **calculateGradientSerial**
    - I iterate over each pixel from (1,1) to (N-2, N-2).
    - For each pixel, I compare its 16-bit intensity against 8 neighbors. If it is lighter than all neighbors, direction = 0. Otherwise, direction is set to 1..8 for the neighbor with the smallest intensity.
        - Compare its 16-bit intensity to the intensities of 8 neighbors (N, NE, E, SE, S, SW, W, NW).
        - If the pixel is lighter than all neighbors, assign 0. Otherwise, store 1..8 for whichever neighbor has the smallest intensity.
    - This step takes O(N^2 * N^2) = O(N^4) time because each pixel checks 8 neighbors.
2. **calculate7x7NeighborhoodSerial**

- For each pixel (x,y) from 3 to (N-4) in both dimensions:
    - I examine a 7×7 window centered on (x,y). Each direction code is converted to (dx, dy). Summing them yields (sumX,sumY).
    - The angle = atan2(sumY,sumX) is mapped to 1..8 or 0.
- Each pixel does a constant 49 neighbor checks, so overall O(N^2).

3. **findEndpointsSerial**
    - For each of the N^2 pixels, I follow its direction field step-by-step, up to a limit of N^2 steps to avoid infinite loops.
    - This path approach can lead to O(N^2 * N^2) = O(N^4) in the worst case. As N grows, the CPU time quickly becomes very large.

While this structure is very direct and clarifies how each pixel is processed, it can be extremely slow for large N.

2.3 Parallel Implementation (CUDA)

To address the high computational cost, I implemented the same three steps on the GPU using CUDA. Each step is handled by a separate kernel. Below are the main components and program flow.

2.3.1 Memory Management & Tiling

- I can either upload the entire image to the GPU if it fits in GPU memory, or split the image into tiles of size tileSize (e.g., 256 or 1024).
- Each tile extracts a sub-rectangle from the original image data on the CPU side. To account for the 7×7 neighborhood at tile boundaries, I expand that sub-rectangle by an overlap of 3 pixels in each direction (where available).
- On the GPU, I allocate arrays for:
    - the 16-bit input tile (d_image),
    - an 8-bit array for directions (d_gradient),
    - an 8-bit array for updated directions (d_newLabels),
    - and a 4-integer-per-pixel array (d_endpoints) to store (startX, startY, endX, endY).

For each tile in the host code (processTileParallel function):

1. I call cudaMalloc for these buffers and copy the sub-rectangle from the CPU to d_image.

2. I launch three kernels (gradient, 7×7 update, endpoints) in sequence, synchronizing after each launch with cudaDeviceSynchronize(). This ensures the data from one step is finalized before the next step reads it.
3. Finally, I copy d_endpoints and d_newLabels back to the CPU, interpret the results, and free GPU memory.

2.3.2 Gradient & 7×7 Kernels

- **calculateGradientCUDA**:
  A CUDA thread is assigned to each pixel (x, y). It reads the 16-bit center and its 8 neighbors in d_image, then writes direction 0..8 to d_gradient[y * width + x].
- **calculate7x7NeighborhoodCUDA**:
  Each thread sums direction vectors in a 7×7 window from d_gradient and computes an angle via atan2f(sumY, sumX). The result is stored in d_newLabels.
  A boundary check ensures x >= 3 && x < width - 3 and similarly for y.

2.3.3 Revised findEndpointsCUDA

- Originally, the path kernel stored every step in a huge global array, which was impractical for large images. The original approach wrote every step of every pixel path (O(N^3) or even O(N^4) memory usage), leading to extreme slowness and memory exhaustion.
- Now, each thread processes exactly one pixel path (just like the serial code). It follows the direction field up to maxSteps = width * height.
- Once it finishes, it writes four integers: (startX, startY, endX, endY). This drastically reduces memory usage to O(4 * N^2).

2.3.4 Synchronization & Program Flow

- **Host side**: After each kernel call (gradient, 7×7, endpoints), I call cudaDeviceSynchronize() to ensure the kernel is completed and the results are accessible for the next step.
- **Tile Integration**: If I am using multiple tiles, I must then copy the relevant portion of d_newLabels back to a global output image or matrix. Also, I parse the (startX,startY,endX,endY) pairs from the GPU's endpoint array, adjusting their coordinates by adding the tile's offset.
- The final result is appended into an output file as (startX, startY) (endX, endY) pairs.

This parallel approach largely mirrors the serial logic, but distributes the work across thousands of GPU threads, making it significantly faster for large images.

3. Performance Evaluation

3.1 Test Methodology

I tested both serial and CUDA implementations on images ranging from 16×16 up to 16384×16384. For the GPU, I tried two tile sizes 256 vs. 1024 to see how memory overhead and kernel launch patterns might affect performance. Overlap was fixed at 3 pixels in each tile dimension to cover the 7×7 boundary areas.

- **Repeated Runs**: For short tests (like 16×16 or 64×64), I ran each scenario 2 or 3 times, writing slash-separated results. For long tests, I sometimes ran only once (since they can take minutes or hours).
- **Units**: I measured total runtime in milliseconds (ms).
- **Skipped Serial for Large**: For images beyond 512×512 or 1024×1024, the serial approach could take hours or even days (due to O(N^4) worst-case), so I omitted those results.

3.2 Results

Below is an illustrative table of measured times (in ms):

Image Size	**Serial (ms)**	**CUDA tileSize=1024 (ms)**	**CUDA tileSize=256 (ms)**
16×16	2 / 2 / 2	86 / 85 / 69	85 / 91 / 80
32×32	16 / 15 / 15	77 / 119 / 91	88 / 93 / 81
64×64	322 / 262 / 330	126 / 115 / 105	132 / 115 / 130
128×128	4690 / 5249 / 4941	254 / 234 / 242	258 / 258 / 244
256×256	84769 / 91408 / 92385	860 / 876 / 850	821 / 825 / 818
512×512	1611284	4765 / 4805 / 4752	3133 / 3218 / 3290
1024×1024	(Not tested)	45626 / 35214 / 35579	12602 / 12631 / 12758
2048×2048	(Not tested)	144785 / 145055 / 145883	50486 / 51874 / 5236
4096×4096	(Not tested)	582124	206338 / 204981
8192×8192	(Not tested)	2415321	886821
16384×16384	(Not tested)	9833751	3957881

3.2.1 Observations

1. **Small Sizes**
   - At 16×16 or 32×32, the serial approach is essentially instantaneous. The GPU shows some overhead from memory transfers and kernel launches, but remains in the same ballpark (tens of milliseconds).
2. **Intermediate Sizes**
   - Already at 64×64, the serial code can spike to hundreds of ms because each pixel can follow up to 4096 steps. Meanwhile, the GPU times remain in the low hundreds or below.
   - By 256×256 or 512×512, the serial code can reach minutes or tens of minutes, whereas the GPU finishes in seconds.
3. **Large Sizes**
   - For 1024×1024 or larger, I did not complete most of the serial tests due to extremely long run times (O(N^4)). The GPU still finishes in seconds or minutes.
   - The choice of tile size also has some effect: occasionally tileSize=256 outperforms tileSize=1024, possibly because large tiles can introduce more memory-allocation overhead or suboptimal block scheduling.
4. **Overall Trend**
   - The GPU approach scales much better: it effectively parallelizes each pixel’s path tracing across thousands of threads, so as N grows, it remains far more feasible than the naive serial code.

4. Conclusion

1. **Design**
   - I followed the assignment’s steps exactly in both serial and CUDA versions. The serial code is straightforward but can blow up to O(N^4) run time for large N.
   - The CUDA version uses one thread per pixel for each kernel, with 7×7 accumulation and path tracing all done in parallel. I revised findEndpointsCUDA to store only (startX, startY, endX, endY) rather than every path step, making it vastly more memory-efficient.
2. **Performance**

    - **Small images:** CPU is super fast, GPU overhead is also relatively small.
    - **Medium to Large images:** CPU times explode (minutes, hours, or even days), while GPU remains in seconds to minutes.
    - **Tiling**: Testing tileSize=256 vs. tileSize=1024 shows that bigger tiles do not always mean faster performance. In some cases, smaller tiles reduce overhead and yield better times.

3. **Future Work**
    - Additional optimization (e.g., pointer-jumping or skipping repeated paths) might improve the serial code drastically, but the assignment specifically required a "basic" reference for direct comparison.
    - In practice, the GPU approach demonstrates how parallel computing is far more suitable for massive image-processing tasks—particularly where each pixel can have a potentially long chain of computations.

I will submit the project source code, build artifacts, and the test images (up to 1024×1024 due to file-size constraints) along with this report. These materials should allow anyone to reproduce the measurements or further analyze the code. Overall, this project clearly illustrates the dramatic speedup that GPU parallelism can achieve when each pixel's work is distributed across thousands of threads.

Acknowledgments

- **NVIDIA** for CUDA documentation and tools.
- **OpenCV** for image I/O functionalities.

Prepared by: Yusuf Tahir Orhan

Date: 01/28/2025