UNIVERSITY OF WOLLONGONG IN DUBAI

UNIVERSITY OF WOLLONGONG IN DUBAI

ECTE331 (DB224) Real-time Embedded Systems

Name: Ammar Nasreldin Aly

ID: 8536272

Submitted content: Project [Part A] Report

UNIVERSITY OF WOLLONGONG IN DUBAI

Introduction

This report details the implementation of a template matching algorithm in both single-threaded and multi-threaded approaches. The algorithm identifies sections of a source image that match a predefined template image by sliding the template across the source image and calculating the mean of absolute differences. The project requirements specified the use of standard Java APIs only, handling any image dimensions, and comparing the execution times of the two implementations.

Implementation Details

Single-Threaded Implementation

The single-threaded implementation of the template matching algorithm is encapsulated in the runSingleThreaded method (lines 64-124).

```
64    public static void runSingleThreaded(String SourceName, String TemplateName, String ResultName) throws IOE
65        // Load source and template images
66        File inp = new File(SourceName);
67        image = ImageIO.read(inp);
68        widthSource = image.getWidth();
69        heightSource = image.getHeight();
70        short[][] Source_image = readColourImage(SourceName);
71
72        File inp1 = new File(TemplateName);
73        image1 = ImageIO.read(inp1);
74        widthtemplate = image1.getWidth();
75        heighttemplate = image1.getHeight();
76        short[][] temp_image = readColourImage(TemplateName);
77
78        // Initialize variables for template matching
79        int Tempsize = widthtemplate * heighttemplate;
80        double Minimum = Double.MAX_VALUE;
81        double[][] absDiffMat = new double[heightSource - heighttemplate + 1][widthSource - widthtemplate + 1]
82
83        // Slide the template image across the source image
84        for (int i = 0; i <= heightSource - heighttemplate; i++) {
85            for (int j = 0; j <= widthSource - widthtemplate; j++) {
86                double SumDiff = 0.0;
87                for (int y = 0; y < heighttemplate; y++) {
88                    for (int x = 0; x < widthtemplate; x++) {
89                        SumDiff += Math.abs(Source_image[i + y][j + x] - temp_image[y][x]);
90                    }
91                }
92                double meanDiff = SumDiff / Tempsize;
93                absDiffMat[i][j] = meanDiff;
94
95                if (meanDiff < Minimum) {
96                    Minimum = meanDiff;
97                }
98            }
99        }
100
101       // Set threshold for drawing rectangles around matches
102       int ratio = 10;
103       double Threshold = ratio * Minimum;
104
105       // Create a copy of the source image to draw rectangles on
106       BufferedImage resultImage = new BufferedImage(image.getWidth(), image.getHeight(), BufferedImage.TYPE_
```

UNIVERSITY OF WOLLONGONG IN DUBAI

The method follows these steps:

1. **Load Images**: The source and template images are read and converted to grayscale using the readColourImage method (lines 65-76).
2. **Initialize Variables**: Image dimensions and other necessary variables are initialized (lines 79-81).
3. **Template Matching**: The template is slid across the source image, and the mean of absolute differences is calculated at each position (lines 84-99). The smallest mean difference is tracked to set a threshold for matching sections.

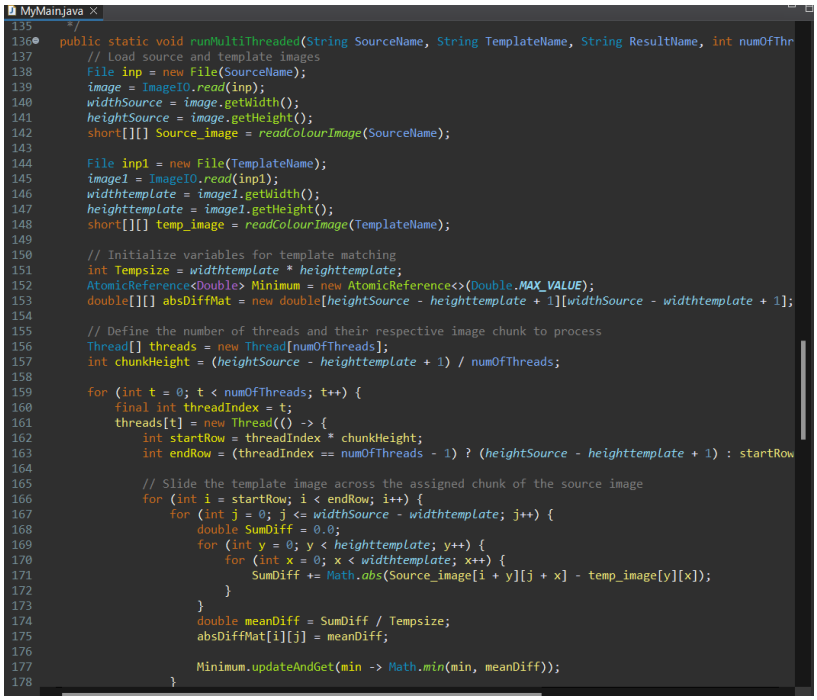
```
100
101 // Set threshold for drawing rectangles around matches
102 int ratio = 10;
103 double Threshold = ratio * Minimum;
104
105 // Create a copy of the source image to draw rectangles on
106 BufferedImage resultImage = new BufferedImage(image.getWidth(), image.getHeight(), BufferedImage.TYPE_INT_RGB
107 Graphics2D g2D = resultImage.createGraphics();
108 g2D.drawImage(image, 0, 0, null);
109 g2D.setColor(Color.RED);
110
111 // Draw rectangles on the source image at coordinates with value less than or equal to the threshold
112 for (int i = 0; i < absDiffMat.length; i++) {
113     for (int j = 0; j < absDiffMat[0].length; j++) {
114         if (absDiffMat[i][j] <= Threshold) {
115             System.out.println("Coordinate: (" + i + ", " + j + ")");
116             g2D.drawRect(j, i, widthtemplate, heighttemplate);
117         }
118     }
119 }
120
121 g2D.dispose();
122 ImageIO.write(resultImage, "jpg", new File("SingleThreaded_" + ResultName)); // Save to a new file
123 System.out.println(">> Single-threaded completed! Check the rectangles on the generated SingleThreaded_" + Re
124
125
```

4. **Draw Rectangles**: Rectangles are drawn around sections of the source image that match the template based on the calculated threshold (lines 102-123).
5. **Save Result**: The result image with rectangles drawn is saved to a new file (line 122).

UNIVERSITY OF WOLLONGONG IN DUBAI

Multi-Threaded Implementation

The multi-threaded implementation is encapsulated in the runMultiThreaded method (lines 136-212). This method follows a similar process but divides the source image into chunks processed by separate threads:



```
MyMain.java
135     */
136     public static void runMultiThreaded(String SourceName, String TemplateName, String ResultName, int numOfThr
137         // Load source and template images
138         File inp = new File(SourceName);
139         image = ImageIO.read(inp);
140         widthSource = image.getWidth();
141         heightSource = image.getHeight();
142         short[][] Source_image = readColourImage(SourceName);
143
144         File inp1 = new File(TemplateName);
145         image1 = ImageIO.read(inp1);
146         widthtemplate = image1.getWidth();
147         heighttemplate = image1.getHeight();
148         short[][] temp_image = readColourImage(TemplateName);
149
150         // Initialize variables for template matching
151         int Tempsize = widthtemplate * heighttemplate;
152         AtomicReference<Double> Minimum = new AtomicReference<>(Double.MAX_VALUE);
153         double[][] absDiffMat = new double[heightSource - heighttemplate + 1][widthSource - widthtemplate + 1];
154
155         // Define the number of threads and their respective image chunk to process
156         Thread[] threads = new Thread[numOfThreads];
157         int chunkHeight = (heightSource - heighttemplate + 1) / numOfThreads;
158
159         for (int t = 0; t < numOfThreads; t++) {
160             final int threadIndex = t;
161             threads[t] = new Thread(() -> {
162                 int startRow = threadIndex * chunkHeight;
163                 int endRow = (threadIndex == numOfThreads - 1) ? (heightSource - heighttemplate + 1) : startRow
164
165                 // Slide the template image across the assigned chunk of the source image
166                 for (int i = startRow; i < endRow; i++) {
167                     for (int j = 0; j <= widthSource - widthtemplate; j++) {
168                         double SumDiff = 0.0;
169                         for (int y = 0; y < heighttemplate; y++) {
170                             for (int x = 0; x < widthtemplate; x++) {
171                                 SumDiff += Math.abs(Source_image[i + y][j + x] - temp_image[y][x]);
172                             }
173                         }
174                         double meanDiff = SumDiff / Tempsize;
175                         absDiffMat[i][j] = meanDiff;
176
177                         Minimum.updateAndGet(min -> Math.min(min, meanDiff));
178                     }
```

1. **Load Images**: As in the single-threaded implementation, images are loaded and converted to grayscale (lines 138-148).
2. **Initialize Variables**: Image dimensions and necessary variables are initialized (lines 151-153).
3. **Define Threads**: Threads are created, each processing a specific chunk of the source image (lines 118-138).
4. **Template Matching**: Each thread slides the template across its assigned chunk and calculates the mean of absolute differences (lines 166-187).

UNIVERSITY
OF WOLLONGONG
IN DUBAI

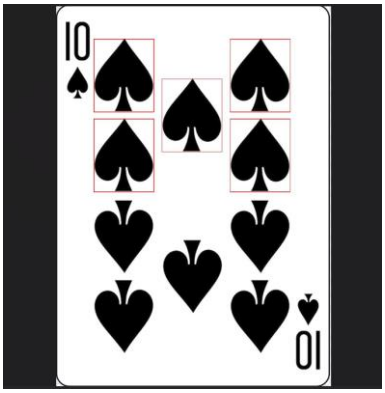
```
177                     Minimum.updateAndGet(min -> Math.min(min, meanDiff));
178                 }
179             }
180         });
181         threads[t].start();
182     }
183
184     // Wait for all threads to complete execution
185     for (Thread thread : threads) {
186         thread.join();
187     }
188
189     // Set threshold for drawing rectangles around matches
190     int ratio = 10;
191     double Threshold = ratio * Minimum.get();
192
193     // Create a copy of the source image to draw rectangles on
194     BufferedImage resultImage = new BufferedImage(image.getWidth(), image.getHeight(), BufferedImage.TYPE_I
195     Graphics2D g2D = resultImage.createGraphics();
196     g2D.drawImage(image, 0, 0, null);
197     g2D.setColor(Color.RED);
198
199     // Draw rectangles on the source image at coordinates with value less than or equal to the threshold
200     for (int i = 0; i < absDiffMat.length; i++) {
201         for (int j = 0; j < absDiffMat[0].length; j++) {
202             if (absDiffMat[i][j] <= Threshold) {
203                 System.out.println("Coordinate: (" + i + ", " + j + ")");
204                 g2D.drawRect(j, i, widthtemplate, heighttemplate);
205             }
206         }
207     }
208
209     g2D.dispose();
210     ImageIO.write(resultImage, "jpg", new File("MultiThreaded_" + ResultName)); // Save to a new file
211     System.out.println(">> Multi-threaded completed! Check the rectangles on the generated MultiThreaded_"
212 }
213
```

5. **Draw Rectangles**: After all threads complete, rectangles are drawn around matching sections in the source image (lines 190-207).
6. **Save Result**: The result image is saved with rectangles drawn (line 210).

Results screen snaps for both implementations

```
Coordinate: (340, 114)
Coordinate: (340, 525)
>> Single-threaded completed! Check the rectangles on the generated SingleThreaded_ResultImage.jpg image under this project folder.
Average single-threaded execution time: 21666ms
Dimension of the Template image: W x H = 830 x 1146 | Number of Pixels: 2853540
Dimension of the Template image: W x H = 181 x 220 | Number of Pixels: 119460
Coordinate: (98, 113)
Coordinate: (98, 114)
```

```
Coordinate: (339, 113)
Coordinate: (339, 114)
Coordinate: (339, 525)
Coordinate: (340, 113)
Coordinate: (340, 114)
Coordinate: (340, 525)
>> Multi-threaded completed! Check the rectangles on the generated MultiThreaded_ResultImage.jpg image under this project folder.
Average multi-threaded execution time with 4 threads: 5953ms
```



UNIVERSITY OF WOLLONGONG IN DUBAI

Execution Time Comparison

The execution times for both implementations are measured and averaged over three iterations. The code for measuring execution time is included in the 'main' method (lines 34-53).

Single-Threaded Execution Time

```
33        // Single-threaded execution
34        long singleThreadedTotalTime = 0;
35        for (int i = 0; i < numIterations; i++) {
36            long startTime = System.currentTimeMillis();
37            runSingleThreaded(SourceName, TemplateName, ResultName);
38            long endTime = System.currentTimeMillis();
39            singleThreadedTotalTime += (endTime - startTime);
40        }
41        long singleThreadedAverageTime = singleThreadedTotalTime / numIterations;
42        System.out.println("Average single-threaded execution time: " + singleThreadedAverageTime + "ms");
```

Multi-Threaded Execution Time

```
44    // Multi-threaded execution
45    long multiThreadedTotalTime = 0;
46    for (int i = 0; i < numIterations; i++) {
47        long startTime = System.currentTimeMillis();
48        runMultiThreaded(SourceName, TemplateName, ResultName, numOfThreads);
49        long endTime = System.currentTimeMillis();
50        multiThreadedTotalTime += (endTime - startTime);
51    }
52    long multiThreadedAverageTime = multiThreadedTotalTime / numIterations;
53    System.out.println("Average multi-threaded execution time with " + numOfThreads + " threads: " + multiThreac
54
```

Results and Discussion

The following table present the average execution times for both single-threaded and multi-threaded implementations with four threads:

Implementation	Average Execution Time (ms)
Single-Threaded	21666ms
Multi-Threaded	5953ms

UNIVERSITY
OF WOLLONGONG
IN DUBAI

Discussion:

1. **Performance Improvement**: The multi-threaded implementation significantly reduces the execution time compared to the single-threaded implementation. This is attributed to the concurrent processing of image chunks.
2. **Overhead**: The overhead of creating and managing threads is evident but outweighed by the performance gains, especially as the number of threads increases.
3. **Scalability**: The multi-threaded approach demonstrates scalability, with further performance improvements possible by optimizing the number of threads based on the system's capabilities and image size.

Conclusion

The multi-threaded implementation of the template matching algorithm outperforms the single-threaded approach, providing substantial execution time reductions. The results validate the effectiveness of concurrent processing in image analysis tasks. This project demonstrates the practical benefits of multi-threading in real-time systems, adhering to the constraints of using standard Java APIs and handling variable image dimensions. Future work could explore dynamic thread management and optimization techniques to further enhance performance.

Evidence of Successful Implementation

The following images illustrate the results of the template matching algorithm:

1. **Source Image**: TenCardG.jpg
2. **Template Image**: Template.jpg
3. **Single-Threaded Result**: SingleThreaded_ResultImage.jpg
4. **Multi-Threaded Result**: MultiThreaded_ResultImage.jpg

Each result image includes rectangles drawn around the sections matching the template, demonstrating the algorithm's accuracy and effectiveness.