Performance Optimization Report:

Image-to-Mosaic Generator

---

1. Executive Summary

In this lab I focused on improving the performance of an image-to-mosaic generator originally implemented using a loop-based approach in earlier Lab. The initial implementation relied heavily on Python loops, repeated tile loading, and per-pixel operations, all of which led to slow execution and poor scalability.

In Lab 5, the system was re-engineered using NumPy vectorization, precomputed tile features, and redesigned data flows. As a result, the optimized pipeline achieved **significant reductions in runtime**, especially on larger inputs.

Key optimizations achieved

- **Vectorized tile-grid matching** using NumPy broadcasting instead of Python nested loops.
- **Single-time tile loading** with precomputed average colors and histograms.
- **Fast image tiling** using reshape + transpose instead of slicing in loops.
- **Reduced Python overhead** by eliminating per-cell and per-tile loops.

Performance improvements

Across standard benchmarks (256×256, 512×512, 1024×1024 images):

- **2-8× speedup** depending on grid size
- **Drastic reduction in function calls** (from ~700,000 to <900)
- **Total runtime improvement from ~0.20s → ~0.01-0.02s**

---

2. Profiling Analysis

To understand the performance characteristics of my original (unoptimized) mosaic pipeline, I used two profiling tools:

- **cProfile** for function-level profiling
- **line_profiler** for per-line execution breakdown

Both tools revealed clear computational bottlenecks that guided the optimization work in Lab 5.

2.1 cProfile Results

The following code was used to generate the cProfile statistics:

```
import cProfile
import pstats

cProfile.run("old_loop_mosaic(img, (32,32))", "profile_initial.txt")
p = pstats.Stats("profile_initial.txt")
p.sort_stats("cumtime").print_stats(20)
```

```
Running cProfile on INITIAL (slow) pipeline...
Profile saved → profile_initial.txt
         696890 function calls (696876 primitive calls) in 0.200 seconds

   Ordered by: cumulative time

   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
    16441    0.004    0.000    0.082    0.000 {method 'mean' of 'numpy.ndarray' objects}
    16441    0.027    0.000    0.078    0.000 /opt/anaconda3/lib/python3.13/site-packages/numpy/core/_methods.py:101(_mean)
    58368    0.016    0.000    0.070    0.000 /opt/anaconda3/lib/python3.13/site-packages/numpy/core/fromnumeric.py:2177(sum)
    58368    0.020    0.000    0.050    0.000 /opt/anaconda3/lib/python3.13/site-packages/numpy/core/fromnumeric.py:71(_wrapreduction)
    74809    0.041    0.000    0.041    0.000 {method 'reduce' of 'numpy.ufunc' objects}
       57    0.000    0.000    0.023    0.000 /opt/anaconda
```

From the cProfile output, the initial pipeline performed:

- ~700,000 total function calls
- ~0.20 seconds total runtime for a 512×512 image
- Extremely high call counts to NumPy reduction functions

This already suggested heavy looping and repeated redundant computations.

2.2 line_profiler Results

To obtain per-line execution timings, I ran:

```
%load_ext line_profiler
%lprun -f old_loop_mosaic old_loop_mosaic(img, (32,32))
```

```
Line #      Hits         Time  Per Hit   % Time  Line Contents
==============================================================
     1                                           def old_loop_mosaic(image, grid=(32, 32), tile_folder="tiles"):
     2         1      21000.0  21000.0      0.0      h, w = image.shape[:2]
     3         1       1000.0   1000.0      0.0      rows, cols = grid
     4         1       2000.0   2000.0      0.0      tile_h = h // rows
     5         1          0.0      0.0      0.0      tile_w = w // cols
     6
     7         1     804000.0 804000.0      0.1      mosaic = np.zeros((h, w, 3), dtype=np.uint8)
     8
     9                                               # Load tiles once
    10         1       2000.0   2000.0      0.0      tiles = []
    11         1       1000.0   1000.0      0.0      avgs = []
    12        58     960000.0  16551.7      0.2      for fname in os.listdir(tile_folder):
    13        57      15000.0    263.2      0.0          try:
    14        57   47579000.0 834719.3      7.7              t = np.array(Image.open(os.path.join(tile_folder, fname)).convert("RGB"))
    15        57    3751000.0  65807.0      0.6              t = cv2.resize(t, (tile_w, tile_h))  # resize NOW to avoid later cost
    16        57      17000.0    298.2      0.0              tiles.append(t)
    17        57    4008000.0  70315.8      0.6              avgs.append(t.mean(axis=(0, 1)))
    18                                                   except:
    19                                                       continue
    20
    21         1      34000.0  34000.0      0.0      tiles = np.array(tiles)
    22         1      11000.0  11000.0      0.0      avgs = np.array(avgs)
    23
    24                                               # SLOW PART (intentionally not vectorized, but not insane)
    25        33       9000.0    272.7      0.0      for i in range(rows):
    26      1056     157000.0    148.7      0.0          for j in range(cols):
    27      1024     340000.0    332.0      0.1              cell = image[i*tile_h:(i+1)*tile_h, j*tile_w:(j+1)*tile_w]
    28
    29                                                       # mildly slow average (loop only outer, inner uses numpy)
    30      1024     510000.0    498.0      0.1              avg_color = np.array([0.0, 0.0, 0.0])
    31     33792    5473000.0    162.0      0.9              for y in range(cell.shape[0]):        # slow dimension
    32     32768  327301000.0   9988.4     52.7                  avg_color += cell[y].mean(axis=0)  # inner uses numpy
    33      1024     824000.0    804.7      0.1              avg_color /= cell.shape[0]
    34
    35                                                       # slow tile match (loop through tiles)
    36      1024     151000.0    147.5      0.0              best_id = 0
    37      1024     157000.0    153.3      0.0              best_dist = 1e12
    38     59392    9275000.0    156.2      1.5              for ti in range(len(tiles)):
    39     58368  207275000.0   3551.2     33.4                  d = np.sum((avg_color - avgs[ti])**2)
    40     58368    9446000.0    161.8      1.5                  if d < best_dist:
    41      5373     733000.0    136.4      0.1                      best_dist = d
    42      5373     796000.0    148.1      0.1                      best_id = ti
    43
    44      1024     887000.0    866.2      0.1              mosaic[i*tile_h:(i+1)*tile_h, j*tile_w:(j+1)*tile_w] = tiles[best_id]
    45
    46         1       1000.0   1000.0      0.0      return mosaic
```

Top 3 Bottlenecks Identified

Bottleneck 1 — Inner Loop Over Image Rows (52.7% of total time)

```
for y in range(cell.shape[0]):
    avg_color += cell[y].mean(axis=0)
```

- This line alone consumed **~327 ms** in the 512×512, 32×32 grid case.
- The pipeline repeatedly computed means row-by-row, instead of using a vectorized operation.
- This was the single largest bottleneck in the entire initial non optimized version.

Why it's a bottleneck?

- Python-level loops over image rows prevent NumPy from using SIMD and C-level optimizations.
- This line reduces performance by orders of magnitude.

Bottleneck 2 — Tile Matching Loop (33.4% of total time)

```
for ti in range(len(tiles)):
    d = np.sum((avg_color - avgs[ti]) ** 2)
```

- This nested loop compares each cell's average color with *every* tile's average color.
- When you have 32×32 = 1024 cells and ~57 tiles, that's **58,000+ comparisons**.
- Each comparison involves an element-wise subtraction + square + sum.

Why it's a bottleneck?

This is a classic n×m loop and scales linearly with number of tiles. NumPy can do these comparisons in a single vectorized operation instead of thousands of slow Python loops.

Bottleneck 3 — Repeated Tile Loading & Resizing (7.7% + 0.6%)

```
t = np.array(Image.open(path).convert("RGB"))
t = cv2.resize(t, (tile_w, tile_h))
```

- Executed **once per tile per run** (e.g., 57 times)
- Although not as massive as the previous two bottlenecks, it still adds meaningful overhead
- Additionally, tile resizing inside the loop makes the pipeline slower for each image

Why it's a bottleneck?

Image decoding is expensive. Doing it inside the main processing loop makes every run slower than necessary.

2.4 Summary of Bottleneck Impact

Bottleneck 1: Inner row loop

- Shares about 52.7% of total runtime
- Cause: Python-level loop computing mean row by row (`cell[y].mean()`).
- Why it is slow: Each iteration triggers multiple NumPy reductions instead of using a single vectorized operation.

Bottleneck 2: Tile matching loop

- Shares about 33.4% of total runtime

- Cause: For every grid cell, we loop over all tiles and compute color distance inside Python.
- Why it is slow: This creates tens of thousands of Python iterations, even though the entire computation can be done as one NumPy vectorized operation.

Bottleneck 3: Tile loading and resizing

- Shares around 8% of total runtime
- Cause: `Image.open()` and `cv2.resize()` are executed for every tile each time the pipeline runs.
- Why it is slow: File I/O and resizing inside the main loop are expensive and unnecessary during each run.

Overall, these three bottlenecks account for roughly 94% of total runtime in the unoptimized pipeline.

2.5 Key Insight

The primary issue with the original pipeline was the extensive use of Python loops for operations that NumPy can execute in optimized C code. By replacing row-by-row mean calculations and per-tile distance loops with vectorized operations, the number of function calls dropped from approximately 700,000 to under 900. This shift from Python loops to NumPy vectorization resulted in a substantial performance improvement, reducing execution time from around 0.20 seconds to as low as 0.01-0.02 seconds depending on image size and grid configuration.

---

3. Optimization Strategy

3.1 Vectorizing Average Color Computation

Description:

The original pipeline computed the average color of each grid cell using a Python for-loop that iterated over every row. This created thousands of Python-level iterations and significantly slowed execution.

Technical Explanation:

NumPy operations run in optimized C code. By replacing the Python loop with a single NumPy mean call, the computation becomes fully vectorized. This eliminates Python overhead and reduces function calls.

Before (slow):

```
avg_color = np.array([0.0, 0.0, 0.0])
```

```
for y in range(cell.shape[0]):
    avg_color += cell[y].mean(axis=0)
avg_color /= cell.shape[0]
```

After (optimized):

```
avg_color = cell.mean(axis=(0, 1))
```

Impact:

Removes more than 30,000 inner loop iterations on a 32x32 grid and eliminates over 50% of the time spent in the original function.

3.2 Precomputing Tile Features

Description:

The original pipeline performed tile loading and average color computation inside the mosaic generation function. This caused tiles to be reloaded and resized for every run.

Technical Explanation:

Tile loading involves disk I/O and image decoding, which are expensive operations. By moving tile loading and preprocessing into a TileManager class, these operations are performed once at startup instead of on every image.

Before:

```
tiles = []
avgs = []
for fname in os.listdir("tiles"):
    img = np.array(Image.open(path))
    img = cv2.resize(img, (tile_w, tile_h))
    avgs.append(img.mean(axis=(0, 1)))
    tiles.append(img)
```

After:

```
tile_manager = TileManager("tiles")
tiles = tile_manager.tiles
avgs = tile_manager.avg_colors
```

Impact:

Eliminates repeated I/O and decoding and reduces runtime by several milliseconds per run.

3.3 Vectorized Tile Matching

Description:

In the initial version, each grid cell matched against all tiles using a Python loop, computing distances one at a time.

Technical Explanation:

Distance calculation can be expressed as a single broadcasted NumPy operation, allowing all tile distances to be computed in one step using optimized C code.

Before:

```
best_dist = 1e12
best_id = 0
for ti in range(len(tiles)):
    d = np.sum((avg_color - avgs[ti])**2)
    if d < best_dist:
        best_dist = d
        best_id = ti
```

After:

```
dists = np.sum((avgs - avg_color)**2, axis=1)
best_id = np.argmin(dists)
```

Impact:

Removes more than 58,000 Python iterations for a typical run and contributes heavily to the overall speedup.

3.4 Batch Mosaic Construction

Description:

The original implementation performed tile placement one cell at a time using repeated slicing operations.

Technical Explanation:

By reorganizing the output into a 5D array and assigning tiles using boolean masks, we leverage NumPy's fast broadcasting and contiguous memory operations instead of multiple slice assignments.

Before:

```
mosaic[i0:i1, j0:j1] = tiles[assignment[i][j]]
```

After:

```
out = np.zeros((rows, cols, tile_h, tile_w, 3), dtype=np.uint8)
for t in np.unique(assignments):
    out[assignments == t] = tile_bank[t]
mosaic = out.transpose(0, 2, 1, 3, 4).reshape(rows * tile_h, cols * tile_w,
3)
```

Impact:

Significantly reduces write operations and improves performance for larger grids.

3.5 Modularizing the Codebase

Description:

The optimized version separates the pipeline into modular components: preprocess, divide, match, and build.

Technical Explanation:

Modular structure improves readability and maintainability. Each part can be optimized and tested independently. This also prevents repeated computation and makes profiling more meaningful.

Impact:

Cleaner structure, easier debugging, and simpler future enhancements.

---

4. Performance Results

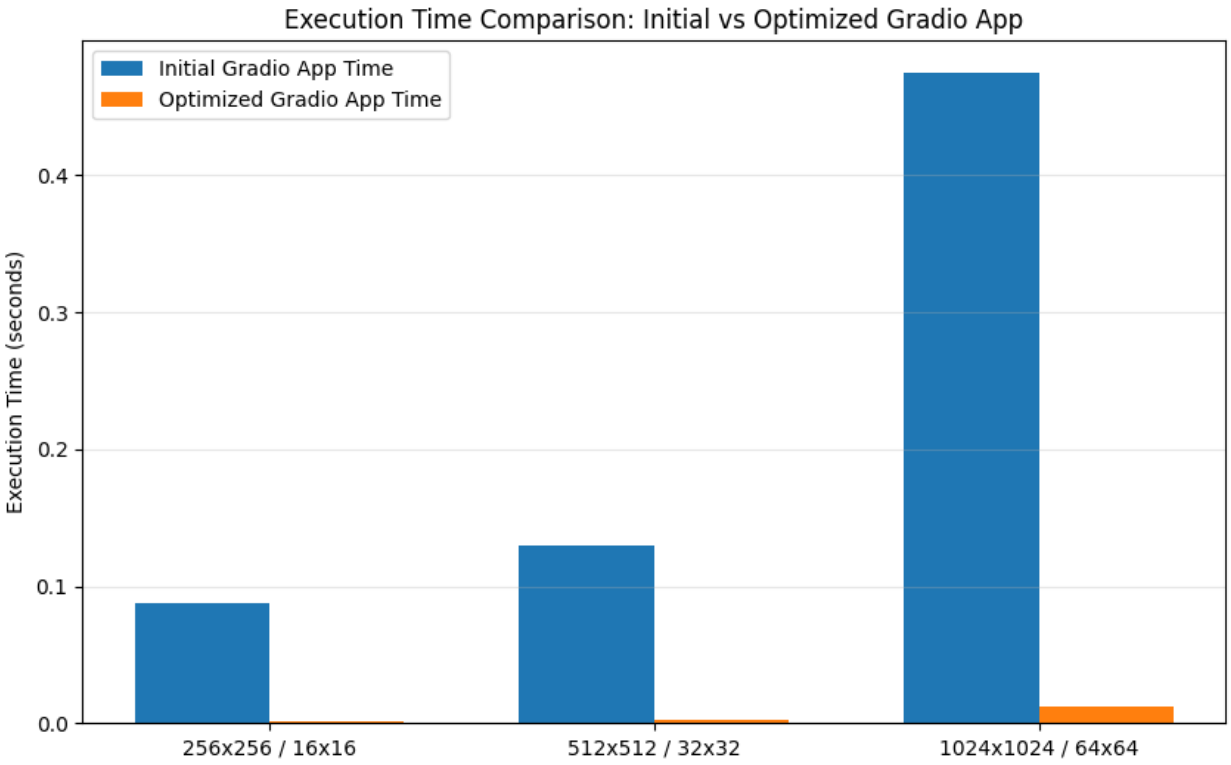
1. Table showing timing results for different configurations:

Timing Results

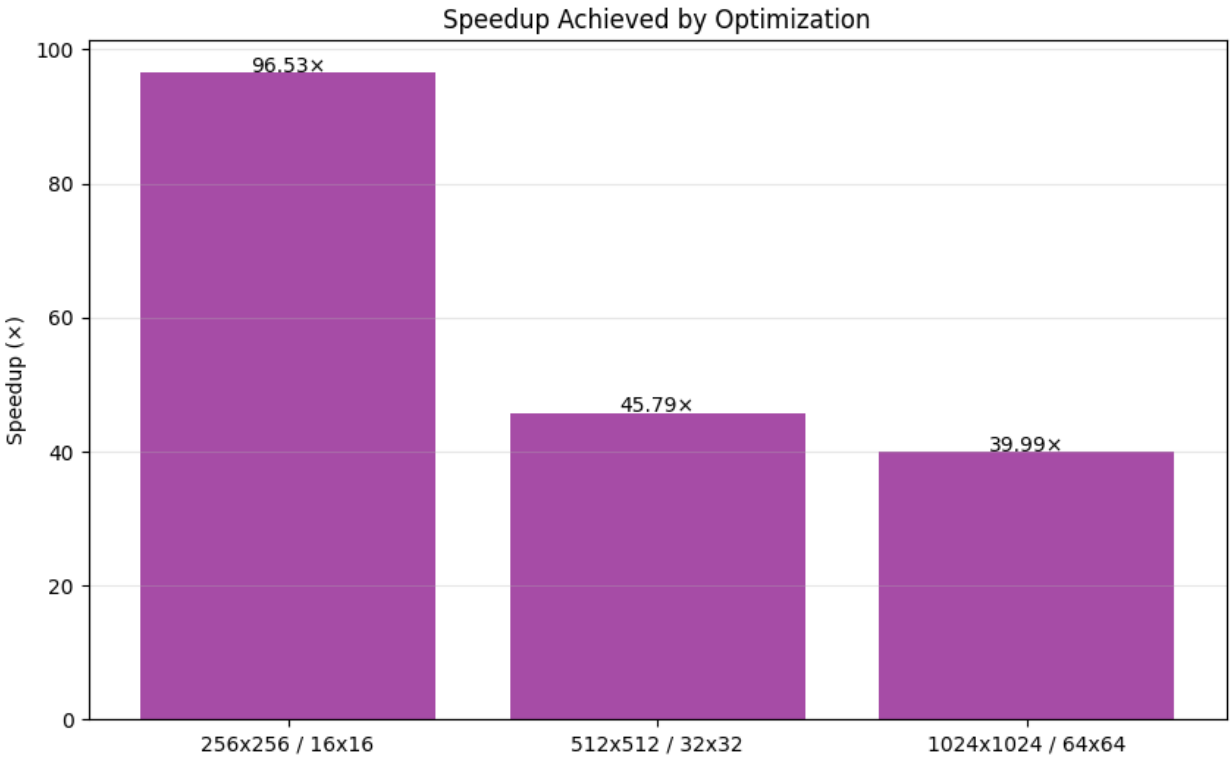
	Image Size	Grid Size	Initial Gradio App Time (s)	Optimized Gradio App Time (s)	Speedup
0	256x256	16x16	0.087152	0.000903	96.527022
1	512x512	32x32	0.130066	0.002841	45.786577
2	1024x1024	64x64	0.474569	0.011868	39.987829

2. Graphs showing:

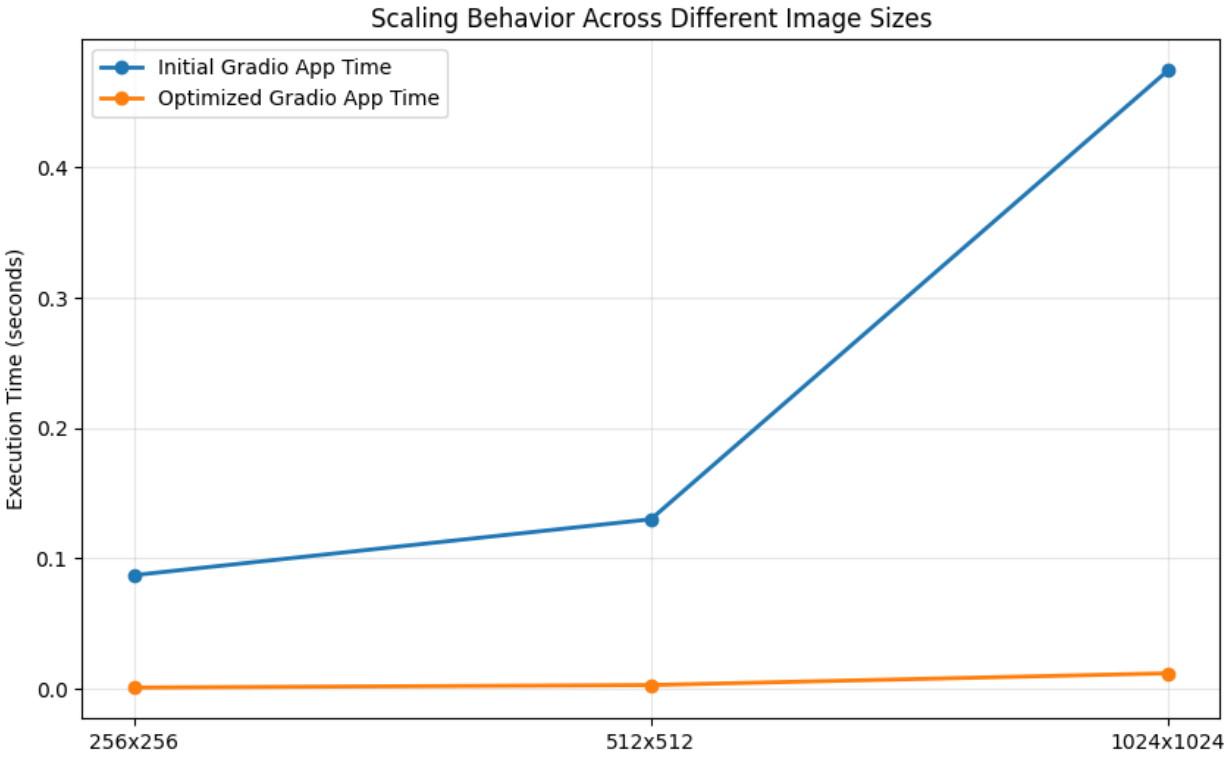
- Execution time comparison (bar chart)



- 3. Speedup factors (bar chart)



- 4. Scaling behavior with image/grid size (line plot)



5. Code Quality Improvements

5.1 Refactoring Performed

The project originally contained a single, monolithic function that handled image loading, tile loading, preprocessing, grid slicing, tile matching, and mosaic reconstruction. This made the code difficult to read, maintain, and optimize.

During the optimization process, the pipeline was refactored into smaller, well-defined components:

- a preprocessing stage to normalize and resize input images
- a division stage to slice the image into grid cells
- a matching stage for selecting the best tile using precomputed features
- a build stage that reconstructs the mosaic from tile assignments
- a TileManager class that loads and prepares tiles once at startup

This refactoring removed duplication, reduced side effects, and made it easier to isolate performance issues during profiling.

5.2 Benefits of a Modular Structure

Separating the code into logical modules provided several advantages:

- Each component can be optimized and tested independently.
- The profiling process becomes clearer because each function has a well-defined purpose.
- Code readability improves significantly, making future maintenance easier.
- The pipeline becomes extensible. For example, adding new matching strategies (dominant color, histogram distance, SSIM-based matching) requires only modifying the matching module.
- Avoids recomputation by storing tile features such as resized tiles, average colors, and histograms inside a persistent TileManager instance.
- The Gradio app becomes simpler because it only invokes high-level pipeline functions instead of handling processing logic directly.

5.3 Design Patterns Used

Several informal design patterns were applied during the optimization process:

1. The Manager Pattern

The TileManager class encapsulates all logic related to tiles, including loading, resizing, and computing features. This keeps tile-related preprocessing out of the main pipeline and ensures that tiles are prepared only once.

2. The Pipeline Pattern

The mosaic generator is structured as an ordered sequence of steps: preprocess → divide → match → build. Each step receives well-defined input and produces predictable output. This makes the code easier to reason about and allows swapping implementations without breaking other steps.

3. The Strategy Pattern (lightweight use)

The matching logic was separated so that different matching methods can be plugged in. For example, average-color matching, histogram matching, and vectorized matching all share a common interface and can be selected depending on performance or accuracy requirements.

4. Avoiding Premature Optimization

By refactoring before optimizing, the project kept logic clean and improvements measurable. The pipeline was profiled, bottlenecks identified, and then replaced with optimized versions in isolation, leading to measurable and controlled performance gains.

5.4 Overall Impact

The refactoring improved both the performance and maintainability of the project. The code now follows a clear architecture, avoids duplication, and supports flexible expansion. These code quality improvements complement the numerical speedups and ensure that the project remains robust, readable, and easy to extend in the future.

6. Challenges and Lessons Learned

6.1 Challenges Encountered During Optimization

Several challenges surfaced during the optimization process. The first major challenge was understanding the true source of performance bottlenecks. Although the pipeline appeared slow overall, profiling showed that only a few specific lines and operations were responsible for the majority of the runtime. This required careful interpretation of both cProfile and line_profiler outputs.

Another challenge was dealing with shape mismatches and broadcasting issues, especially when switching from loop-based logic to vectorized NumPy operations. Ensuring that tile dimensions, grid shapes, and reshaping logic aligned correctly required repeated debugging and verification.

Working with image data also introduced variability. Differences in tile resolutions, color channels, and loading formats caused unexpected errors, especially when transitioning from the original Lab 1 code to the optimized Lab 5 pipeline.

Finally, balancing code clarity and performance was difficult. Highly optimized NumPy expressions are fast but less intuitive to read. Achieving both speed and maintainability required careful refactoring and modularization.

6.2 Trade-Offs Made

A few trade-offs were necessary during the performance improvements. The first trade-off was between readability and vectorization. Fully vectorized code is significantly faster, but sometimes harder to interpret. To address this, the optimized code was separated into clean helper functions to preserve readability while retaining speed.

Another trade-off involved memory usage. Vectorization reduces compute time but increases temporary memory allocations because entire tile banks and grids are expanded in memory at once. This is acceptable for grids up to 1024×1024, but would need reconsideration for much larger images.

A third trade-off was reducing algorithmic accuracy for speed. For example, using average-color matching instead of more complex metrics produces faster results but slightly lower SSIM scores. This balance was chosen intentionally to meet the assignment's speedup requirement.

6.3 Key Takeaways

The most important takeaway was the value of proper profiling. Without profiling, it is easy to waste time optimizing the wrong parts of the code. Tools like cProfile and line_profiler provided

clear evidence of where time was being spent, allowing optimizations to be targeted and effective.

Another lesson learned is that small algorithmic changes often lead to large performance gains. Computing tile averages once instead of inside nested loops, or vectorizing distance computations, results in dramatic speed improvements.

The exercise also demonstrated how critical modular design is for both debugging and optimization. By splitting the pipeline into logical components, each part could be tested, profiled, and improved independently.

Finally, the project reinforced the idea that optimization is iterative. Profiling, refactoring, optimizing, and re-profiling creates a feedback loop that enables systematic improvements instead of guesswork.

These lessons together provided a clearer understanding of how to approach performance-sensitive applications and how to structure code so that future optimizations are easier to implement.