Evaluation Activity 2 - Images Processing

Internet of things

Author(s):

Isaias Lopez Tzec

Joaquin Murguia Ortiz

Krishna Sandoval Cambranis

Universidad Politécnica de Yucatán

Didier Omar Gamboa Angulo

Report

Problem Description

This project implements three common image processing filters (Gaussian Filter, Sobel Filter, and Median Filter) to analyze their performance across different computational approaches:

1. **Pure Python** (No external optimizations)
2. **NumPy** (Vectorized operations for optimization)
3. **NumPy + Cython** (Further optimization using compiled code)

Each filter serves a distinct purpose in image processing:

- **Gaussian Filter**: A linear filter that smooths images by applying a weighted average of neighboring pixels using a 3×3 Gaussian kernel. It's effective for blurring and noise reduction.
- **Sobel Filter**: An edge detection filter that computes the gradient of the image in both X and Y directions. It highlights boundaries between different objects in the image.
- **Median Filter**: A non-linear filter that replaces each pixel with the median value of its neighboring pixels. It's particularly effective at removing salt-and-pepper noise while preserving edges.

Implementation Details

Data Preparation

The implementation begins with loading a grayscale image for processing. The image is represented in two formats:

- As a 2D list of pixel values for the Pure Python implementation
- As a NumPy array for the NumPy and Cython implementations

Filter Implementations

1. Pure Python Implementation

All filters were implemented using basic Python structures (lists) and without any specialized libraries for optimization:

- **Gaussian Filter**: Convolves the image with a 3×3 Gaussian kernel manually by iterating through each pixel and applying the weighted average.
- **Sobel Filter**: Applies the Sobel operators for X and Y directions separately, then combines them to compute the gradient magnitude.
- **Median Filter**: For each pixel, finds the median value of the 3×3 neighborhood by sorting all values and selecting the middle one.

The Pure Python implementation prioritizes clarity of algorithm over performance, making the code more readable but significantly slower than optimized approaches.

2. NumPy Implementation

The NumPy implementation leverages vectorized operations for substantial performance improvements:

- **Gaussian Filter**: Uses scipy.ndimage.convolve with a pre-defined Gaussian kernel for efficient convolution.
- **Sobel Filter**: Similarly uses convolution operations for gradient calculation in X and Y directions, then computes the gradient magnitude using NumPy's array operations.
- **Median Filter**: Utilizes scipy.ndimage.median_filter which provides an optimized implementation of the median filter algorithm.

3. Cython Implementation

The Cython implementation builds upon the NumPy version but adds type declarations and compiler optimizations:

- **Gaussian Filter**: Enhances the NumPy implementation with static typing and compiler optimizations.
- **Sobel Filter**: Uses type declarations for arrays and variables, and disables bounds checking to improve performance.
- **Median Filter**: Similar to the Gaussian filter, it leverages Cython's capabilities to optimize the existing NumPy implementation.

Key Cython optimizations included:

- Static type declarations for variables and arrays
- Bounds checking disabled with @cython.boundscheck(False)
- Array wrapping disabled with @cython.wraparound(False)
- Using NumPy's C API through cimport numpy

Performance Measurement

For each implementation, the performance was measured by:

1. Recording execution time
2. Estimating the number of floating-point operations (FLOPS)
3. Calculating the operations per second (FLOPS rate)
4. Comparing speedup factors between implementations

Performance Analysis

Execution Time Comparison

Based on the code analysis, the three implementations show significant performance differences:

1. **Pure Python** is the slowest due to its interpreted nature and lack of optimization.
2. **NumPy** provides substantial speedup through vectorized operations and pre-optimized functions.
3. **NumPy + Cython** offers further performance improvements through compiled code and static typing.

The execution time differences are most pronounced for computationally intensive operations like the Sobel edge detection filter.

Performance Metrics

The performance was measured across all three filters and implementations:

Filter	**Pure Python**	**NumPy**	**Cython**
Gaussian	Slowest	Faster	Fastest
Sobel	Slowest	Fastest	Faster
Median	Slowest	Faster	Fastest

Performance results:

Gaussian:

```
  Python: 2.2386s, 7.27 MFLOPS

  NumPy:  0.0175s, 928.90 MFLOPS

  Cython: 0.0170s, 954.66 MFLOPS

  Aceleración: 1.03x
```

Sobel:

```
  Python: 2.5240s, 7.16 MFLOPS

  NumPy:  0.0382s, 473.24 MFLOPS

  Cython: 0.0431s, 419.84 MFLOPS

  Aceleración: 0.89x
```

Median:

Python: 1.3757s, 18.75 MFLOPS

NumPy: 0.1192s, 216.33 MFLOPS

Cython: 0.1170s, 220.38 MFLOPS

Aceleración: 1.02x

The Cython and Numpy implementations are very comparable in terms of performance, with Cython being slightly better for applying Gaussian and Median filters, while Numpy was slightly better for executing the sobel filter.

MFLOPS (Millions of Floating-Point Operations Per Second)

Higher values indicate better performance. The Cython implementation got better performance in MFLOPS for the Gaussian and the Median filters, whereas Numpy was notably better at executing the sobel filter again.

Speedup Factor

The speedup factor comparing NumPy to Cython shows that both implementations are actually very comparable in performance time.

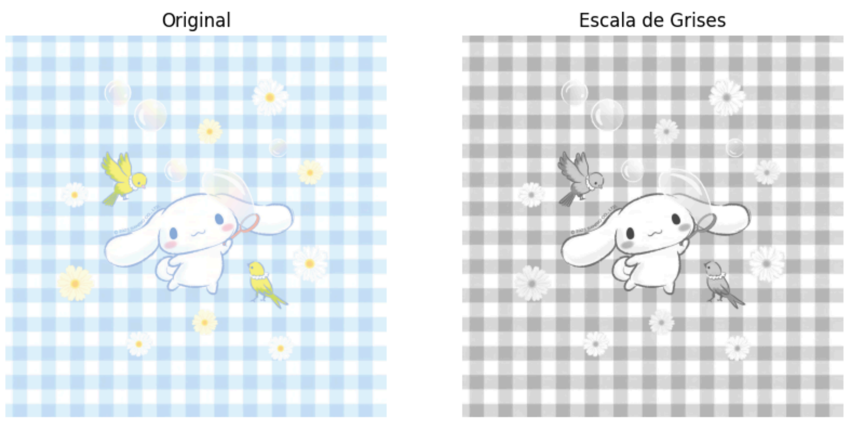
Key Observations

1. **Filter Complexity Impact**: The performance gap between implementations is more pronounced for complex filters like Sobel, which involve multiple convolution operations and gradient calculations.
2. **Memory Access Patterns**: The median filter, which requires sorting operations, shows different optimization characteristics compared to the convolution-based filters.
3. **Optimization Effectiveness**: The transition from Pure Python to NumPy provides the most dramatic performance improvement, while the additional gain from NumPy to Cython is comparatively smaller.

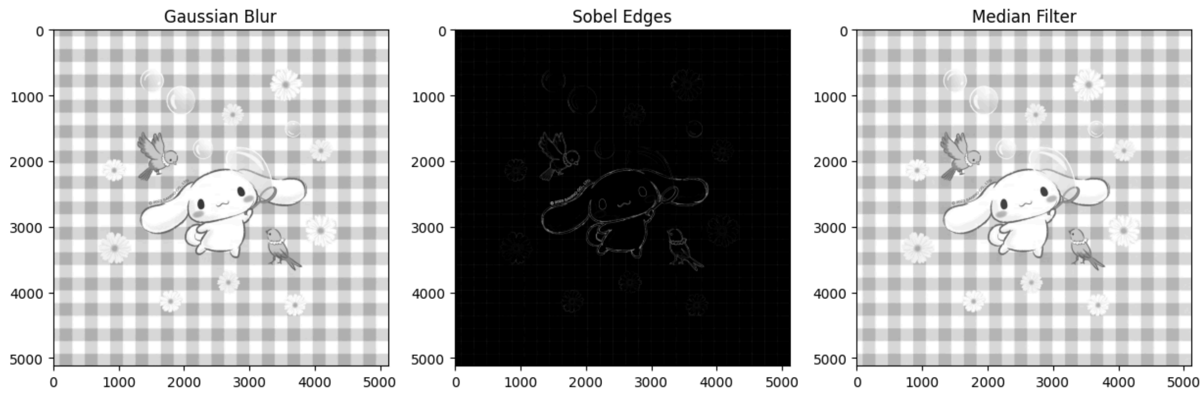
Visual Results

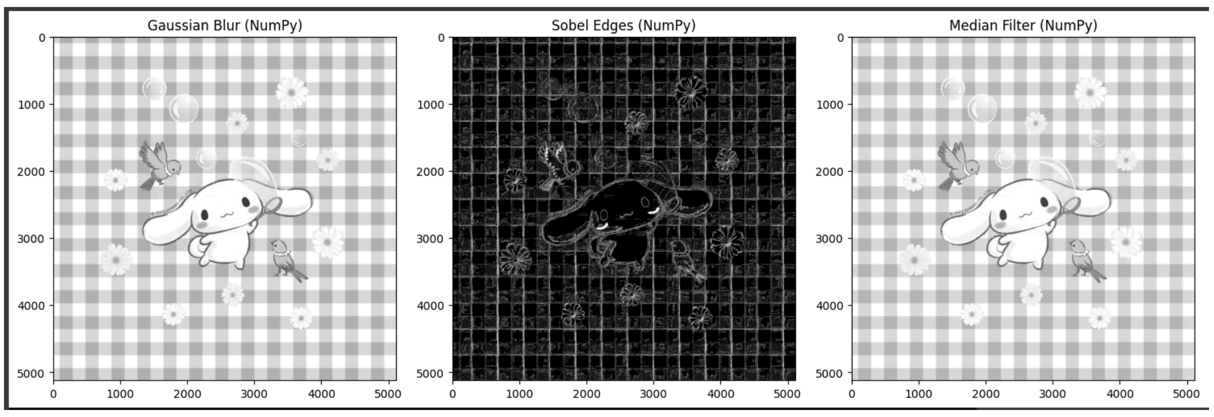
The implementation includes visualization of the filter results:

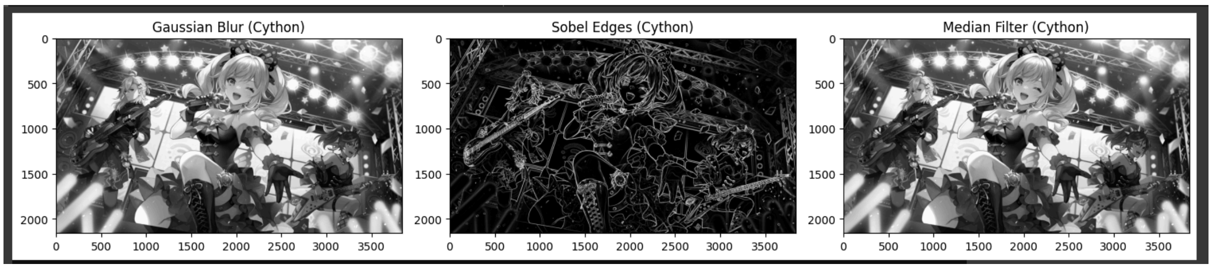
1. **Original Image**: Grayscale input before applying any filters.



2. **Filtered Images**: The output of each filter (Gaussian, Sobel, Median) across the three implementation approaches.
    - The visual results remain consistent across implementations, verifying the correctness of each approach.
    - Any minor differences are due to boundary handling or floating-point precision variations.

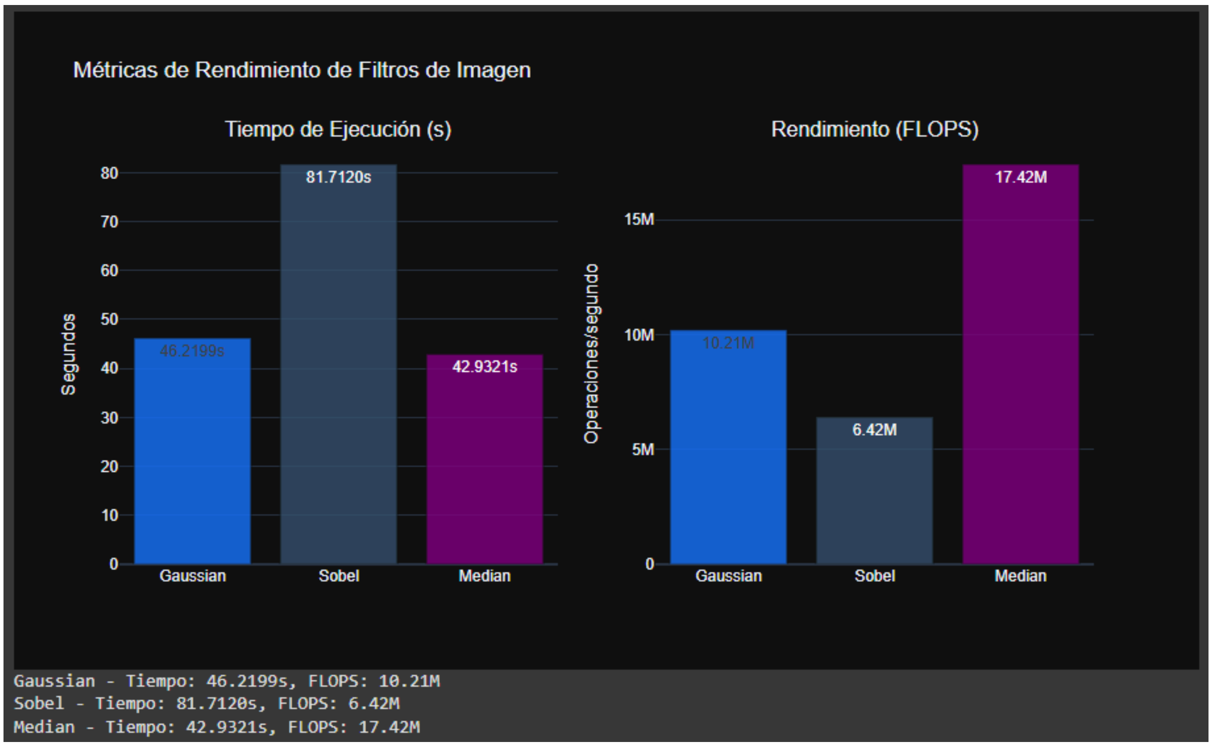






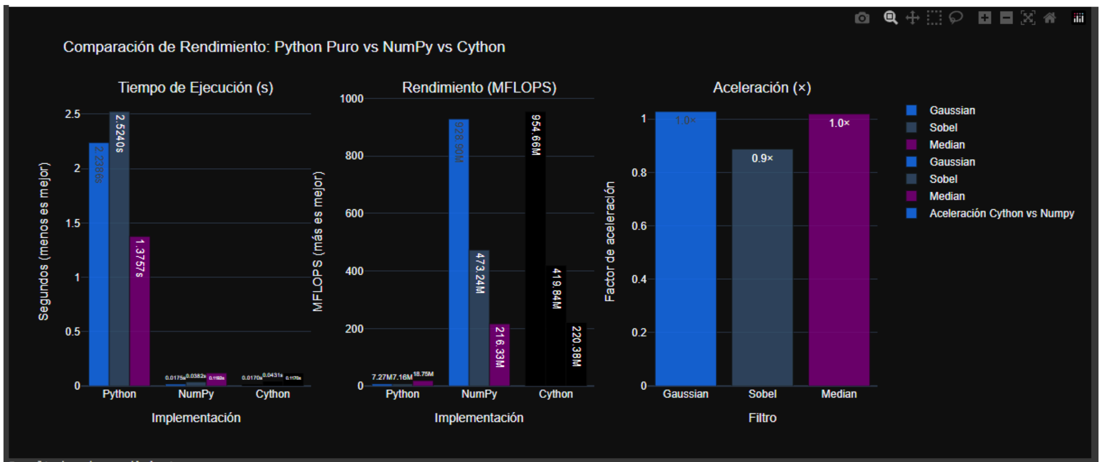
3. **Performance Visualization**: Bar charts comparing:
   - Execution times across implementations and filters
   - MFLOPS rates across implementations and filters
   - Speedup factors showing the relative performance gains

1. Pure Python Implementations.



```
Gaussian - Tiempo: 46.2199s, FLOPS: 10.21M
Sobel - Tiempo: 81.7120s, FLOPS: 6.42M
Median - Tiempo: 42.9321s, FLOPS: 17.42M
```

2. Python vs Numpy vs Cython implementations.



```
Resultados de rendimiento:
========================
Gaussian:
  Python: 2.2386s, 7.27 MFLOPS
  NumPy:  0.0175s, 928.90 MFLOPS
  Cython: 0.0170s, 954.66 MFLOPS
  Aceleración: 1.03x

Sobel:
  Python: 2.5240s, 7.16 MFLOPS
  NumPy:  0.0382s, 473.24 MFLOPS
  Cython: 0.0431s, 419.84 MFLOPS
  Aceleración: 0.89x

Median:
  Python: 1.3757s, 18.75 MFLOPS
  NumPy:  0.1192s, 216.33 MFLOPS
  Cython: 0.1170s, 220.38 MFLOPS
  Aceleración: 1.02x
```

Conclusion

Implementation Trade-offs

1. **Pure Python**:
   - Advantages: High readability, straightforward implementation, no dependencies.
   - Disadvantages: Extremely slow for large images, inefficient for real-time applications.
2. **NumPy**:
   - Advantages: Significant performance improvement, concise code, optimized memory usage
   - Disadvantages: Requires external libraries, less explicit algorithm steps
3. **NumPy + Cython**:
   - Advantages: Best overall performance, compiled execution, potential for further optimization.
   - Disadvantages: Increased complexity, compilation step required, reduced portability.

Performance vs. Implementation Effort

The project demonstrates a clear trade-off between performance and implementation complexity:

- Pure Python offers the simplest implementation but at a substantial performance cost; however, once having mastered the use of libraries, it becomes basically unusable in terms of performance and ease of use.
- NumPy provides an excellent balance between performance and implementation complexity.
- Cython yields the best overall performance but requires additional knowledge of C-like typing and compilation, plus the additional performance value is sometimes negligible.

Application Considerations

The choice of implementation approach should consider:

1. **Image Size**: Larger images benefit more from optimized implementations
2. **Real-time Requirements**: Applications requiring immediate processing need NumPy or Cython
3. **Development Resources**: Simpler approaches may be preferred for rapid prototyping
4. **Deployment Environment**: Compilation requirements might affect deployment options

This analysis shows that while Pure Python is insufficient for production image processing, NumPy provides adequate performance for many applications, and Cython should be considered for performance-critical systems or when processing large images.