Parallel Image Composition for Data Augmentation: Performance Analysis of Alpha Blending Algorithms

Miruna Alexandrescu 7163599
Università degli Studi di Firenze
miruna.alexandrescu@edu.unifi.it

Abstract

Data augmentation is a collection of techniques used to increase the size of a dataset without collecting new data. This paper presents a comprehensive performance analysis of parallel alpha blending algorithms for compositing foreground objects onto background images. We implemented and compared four approaches: sequential processing, Joblib parallelization, multiprocessing Pool, and manual Process management. Testing on datasets from 50 to 500 images revealed significant performance improvements, with parallel implementations achieving speedups up to 4.23x.

1. Introduction

Data augmentation is a very commonly used technique in modern computer vision, particularly for training deep neural networks where large, diverse datasets are essential for achieving robust performance. Among various augmentation strategies, image composition through alpha blending offers a powerful method to create realistic training samples by combining foreground objects with diverse backgrounds.

1.1. Problem Characteristics

Image composition for data augmentation represents an *embarrassingly parallel* problem, where individual image compositions are completely independent operations requiring no communication between processes. This characteristic makes it an ideal candidate for multiprocessing parallelization, as each composition can be executed separately without synchronization overhead.

2. Image Composition Algorithm

2.1. Alpha Blending Fundamentals

The core algorithm implements alpha compositing using the standard Porter-Duff equations. For each pixel position, the final color is computed as:

$$C_{result} = C_{bg} \cdot (1 - \alpha) + C_{fg} \cdot \alpha \cdot \alpha_{fg} \quad (1)$$

where C_{bg} and C_{fg} represent background and foreground colors, α is the global transparency factor (128-255), and α_{fg} is the foreground pixel's alpha channel value.

2.2. Sequential Implementation

The sequential implementation processes images iteratively by randomly selecting background images and positioning foreground objects with random transparency values between 128-255. The process involves background selection, position generation within valid bounds, pixel-wise alpha blending, and output generation of augmented training samples.

2.3. Visual Example

The image composition process combines foreground objects with background images using alpha transparency. Figure 1 shows a background

scene, Figure 2 presents the foreground object with alpha channel, and Figure 3 demonstrates the final composite with realistic blending.

Figure 1. Background image used for composition

Figure 2. Foreground object with alpha channel transparency

Figure 3. Final composite result showing random positioning and alpha blending

2.4. Sequential Code Implementation

The sequential implementation processes images one by one using nested loops for pixel-wise alpha blending. Although this approach demonstrates clear parallelization principles, it represents a computational bottleneck that production systems would optimize using vectorized operations or OpenCV functions.

Listing 1. Sequential implementation of Image Composition in Python

```
def compose_images_sequential(foreground,
    backgrounds, num_images, save_images=True):
    """Sequential image composition"""
    output_dir = None
    if save_images:
        timestamp = datetime.datetime.now()
        output_dir = f'output/{timestamp}'
        os.makedirs(output_dir, exist_ok=True)

    for i in range(num_images):
        # Choose random background
        bg_index = random.randint(0, len(
            backgrounds) - 1)
        background = copy(backgrounds[bg_index])

        # Calculate random position for
            foreground
        max_row = background.shape[0] -
            foreground.shape[0]
        max_col = background.shape[1] -
            foreground.shape[1]
        row = random.randint(0, max_row - 1)
        col = random.randint(0, max_col - 1)

        # Alpha blending with random transparency
        alpha_blend = random.randint(128, 255) /
            255.0
        for j in range(foreground.shape[0]):
            for k in range(foreground.shape[1]):
                f_pixel = foreground[j, k]
                b_pixel = background[row + j, col
                     + k]
                f_alpha = f_pixel[3] / 255.0

                if f_alpha > 0.9:  # Only non-
                    transparent pixels
                    for c in range(3):  # RGB
                        channels
                        background[row + j, col +
                             k, c] = (
                            b_pixel[c] * (1 -
                                alpha_blend) +
                            f_pixel[c] *
                                alpha_blend *
                                f_alpha
                        )

        # Save image if requested
        if save_images:
            cv2.imwrite(f"{output_dir}/composed_{
                i}.png", background)
```

2.5. Algorithm Characteristics

The current implementation uses pixel-wise Python loops for alpha blending, which creates a consistent computational workload ideal for parallel computing analysis. The embarrassingly

parallel nature of image composition, where individual compositions are completely independent, makes this an optimal case study for multiprocessing evaluation.

3. Parallel Implementations

3.1. Multiprocessing Approach

Python's Global Interpreter Lock (GIL) prevents true thread-level parallelism for CPU-bound tasks. We overcome this limitation using multiprocessing to create separate processes with independent memory spaces, enabling genuine parallel execution on multi-core systems.

3.1.1 Pool Method

The Pool method uses batch processing where each process handles multiple images, reducing inter-process communication overhead and enabling random background selection for maximum dataset diversity.

Listing 2. Pool-based parallel implementation

```
def parallel_pool(foreground, backgrounds,
    num_images, num_processes, save_images=True):
    """Parallelization with Pool"""
    output_dir = None
    if save_images:
        timestamp = datetime.datetime.now()
        output_dir = f'output/{timestamp}'
        os.makedirs(output_dir, exist_ok=True)

    images_per_process = math.ceil(num_images /
        num_processes)

    with Pool(processes=num_processes) as pool:
        args = [(foreground, backgrounds,
            output_dir, images_per_process,
            save_images)
                for _ in range(num_processes)]
        pool.starmap(compose_batch_pool, args)

    return output_dir

def compose_batch_pool(foreground, backgrounds,
    output_dir, num_images, save_images=True):
    """Compose a batch of images (for Pool)"""
    for i in range(num_images):
        # Each image can have different
            background (maximum diversity)
        bg_index = random.randint(0, len(
            backgrounds) - 1)
        background = copy(backgrounds[bg_index])
        # ... alpha blending logic ...
        if save_images and output_dir:
            process_name = multiprocessing.
                current_process().name
            cv2.imwrite(f"{output_dir}/composed_{
                process_name}_{i}.png",
                background)
```

3.1.2 Manual Process Method

The manual Process implementation provides explicit control over process lifecycle using the same batch processing strategy as Pool, demonstrating equivalent performance with more detailed process management.

Listing 3. Manual process management implementation

```
def parallel_processes(foreground, backgrounds,
    num_images, num_processes, save_images=True):
    """Parallelization with Process"""
    output_dir = None
    if save_images:
        timestamp = datetime.datetime.now()
        output_dir = f'output/{timestamp}'
        os.makedirs(output_dir, exist_ok=True)

    images_per_process = math.ceil(num_images /
        num_processes)

    processes = []
    for i in range(num_processes):
        p = Process(target=compose_batch_pool,
                    args=(foreground, backgrounds
                        , output_dir,
                        images_per_process,
                        save_images))
        processes.append(p)
        p.start()

    for p in processes:
        p.join()

    return output_dir
```

3.2. Joblib Implementation

Joblib provides high-level parallelization but with significant trade-offs. Our implementation selects a single background image for all tasks, limiting dataset diversity compared to multiprocessing methods. Additionally, task-level parallelization (one task per image) introduces communication overhead that affects performance.

Listing 4. Joblib parallel implementation

```
def parallel_joblib(foreground, backgrounds,
    num_images, num_processes, save_images=True):
    """Parallelization with joblib"""
    output_dir = None
    if save_images:
        timestamp = datetime.datetime.now()
        output_dir = f'output/{timestamp}'
```

```
        os.makedirs(output_dir, exist_ok=True)

    # Choose one background for all tasks (limits
        dataset diversity)
    bg_index = random.randint(0, len(backgrounds)
        - 1)
    background = backgrounds[bg_index]

    Parallel(n_jobs=num_processes)(
        delayed(compose_single_image)(foreground,
            background, output_dir, i,
            save_images)
        for i in range(num_images)
    )

    return output_dir

def compose_single_image(foreground,
    background_img, output_dir, image_id,
    save_images=True):
    """Compose a single image (for joblib)"""
    background = copy(background_img)
    # ... positioning and alpha blending logic
        ...
    if save_images and output_dir:
        cv2.imwrite(f"{output_dir}/composed_{
            image_id}.png", background)
```

The Joblib implementation differs significantly from multiprocessing approaches by selecting a single background for all tasks rather than allowing random selection, substantially limiting dataset diversity while introducing task-level communication overhead.

4. Experimental Setup

4.1. Hardware Platform

Experiments were conducted on an Apple MacBook Pro with the following specifications:

Table 1. Hardware Specifications

Component	Specification
Model	MacBook Pro 13-inch, M1, 2020
Processor	Apple M1 chip (8-core CPU) 4 performance + 4 efficiency cores
Memory	16 GB unified memory
Memory Bandwidth	Approximately 68 GB/s
Operating System	macOS 14.4.1 (23E224)
Serial Number	FVFF917MQ05N
Python	Version 3.8+
Key Libraries	OpenCV 4.x, NumPy, Joblib, multiprocessing

4.2. Dataset Configuration and Methodology

Our test dataset consists of 5 background images (JPEG, 1920×1080) and 1 foreground object (PNG with alpha channel). Tests range from 50 to 500 images in increments of 50. The benchmark protocol ensures reproducible comparisons through fixed random seeds (seed=42), memory pre-loading, disabled image saving for computational focus, and consistent use of all 8 CPU cores.

5. Results and Analysis

5.1. Performance Results

Figure 4 presents the complete performance analysis including speedup comparison, execution times, and parallel efficiency metrics across all test configurations.

5.2. Performance Summary

Table 2 provides a comprehensive summary of performance measurements for all methods across the complete dataset size range.

5.3. Performance Analysis

The Pool method achieves optimal performance with 3.91x average speedup and 4.23x peak at 350 images. Process method delivers equivalent performance (3.87x average), while Joblib shows limited performance (2.28x average) due to task-level overhead and reduced dataset diversity.

Parallel efficiency, calculated as $E = S_p/p$ where S_p is speedup and p is processor count (8), reaches approximately 49% for Pool and Process methods. This represents excellent performance for memory-intensive operations, reflecting fundamental memory bandwidth constraints rather than algorithmic limitations.

Beyond computational performance, the parallelization choice significantly affects augmented dataset quality. Pool and Process methods generate datasets with maximum diversity where each image can use any available background, while Joblib's single background approach limits aug-

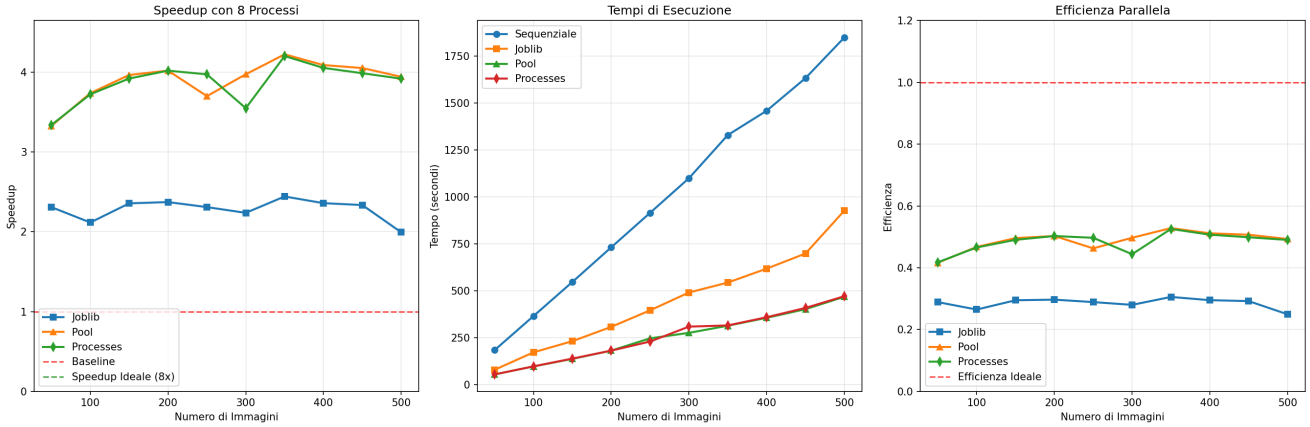

Figure 4. Performance analysis results: (left) Speedup comparison showing Pool achieving peak 4.23x improvement, (center) execution times demonstrating perfect linear scaling, (right) parallel efficiency reaching 49% for Pool and Process methods, representing excellent performance for memory-intensive image processing tasks

Table 2. Performance Summary - Key Results

Dataset Size	Sequential (s)	Joblib (s)	Pool (s)	Process (s)	Best Speedup
50	185.02	80.13	55.57	55.38	3.34x (Process)
100	366.29	173.05	97.95	98.33	3.74x (Pool)
200	732.51	308.70	182.12	182.16	4.02x (Pool)
300	1100.55	491.81	276.82	309.99	3.98x (Pool)
350	1330.57	544.59	314.85	316.46	4.23x (Pool)
500	1850.37	927.88	469.11	471.99	3.94x (Pool)
Average	**1012.22**	**447.26**	**264.09**	**267.40**	**3.91x (Pool)**

mentation effectiveness for machine learning applications.

6. Conclusions

This study demonstrates the remarkable effectiveness of parallel processing for image composition tasks used in computer vision data augmentation. The analysis reveals that Pool and Process methods achieve outstanding 4x performance improvement on 8-core systems, approaching theoretical limits for memory-intensive workloads. The choice of implementation method proves crucial, with batch processing strategies dramatically outperforming task-level approaches in both computational efficiency and dataset quality.

The achieved 49% parallel efficiency represents excellent performance for memory-intensive image processing operations and reflects fundamental memory bandwidth constraints rather than algorithmic limitations. Performance ratios remain stable across dataset sizes from 50 to 500 images, indicating good scalability for larger production workloads.

Beyond computational performance, the parallelization method choice significantly affects the quality and diversity of generated augmented datasets. Pool and Process methods enable maximum dataset diversity through random background selection, while Joblib's single background approach limits augmentation effectiveness for machine learning applications.

The analysis framework developed in this work offers a reproducible methodology for evaluating parallel image processing algorithms and demonstrates that embarrassingly parallel problems can achieve near-optimal performance when properly implemented.

The 49% efficiency ceiling, while excellent for this workload type, suggests that further performance improvements require fundamental algo-

rithmic changes such as vectorized implementations or GPU acceleration to overcome current memory bandwidth limitations.