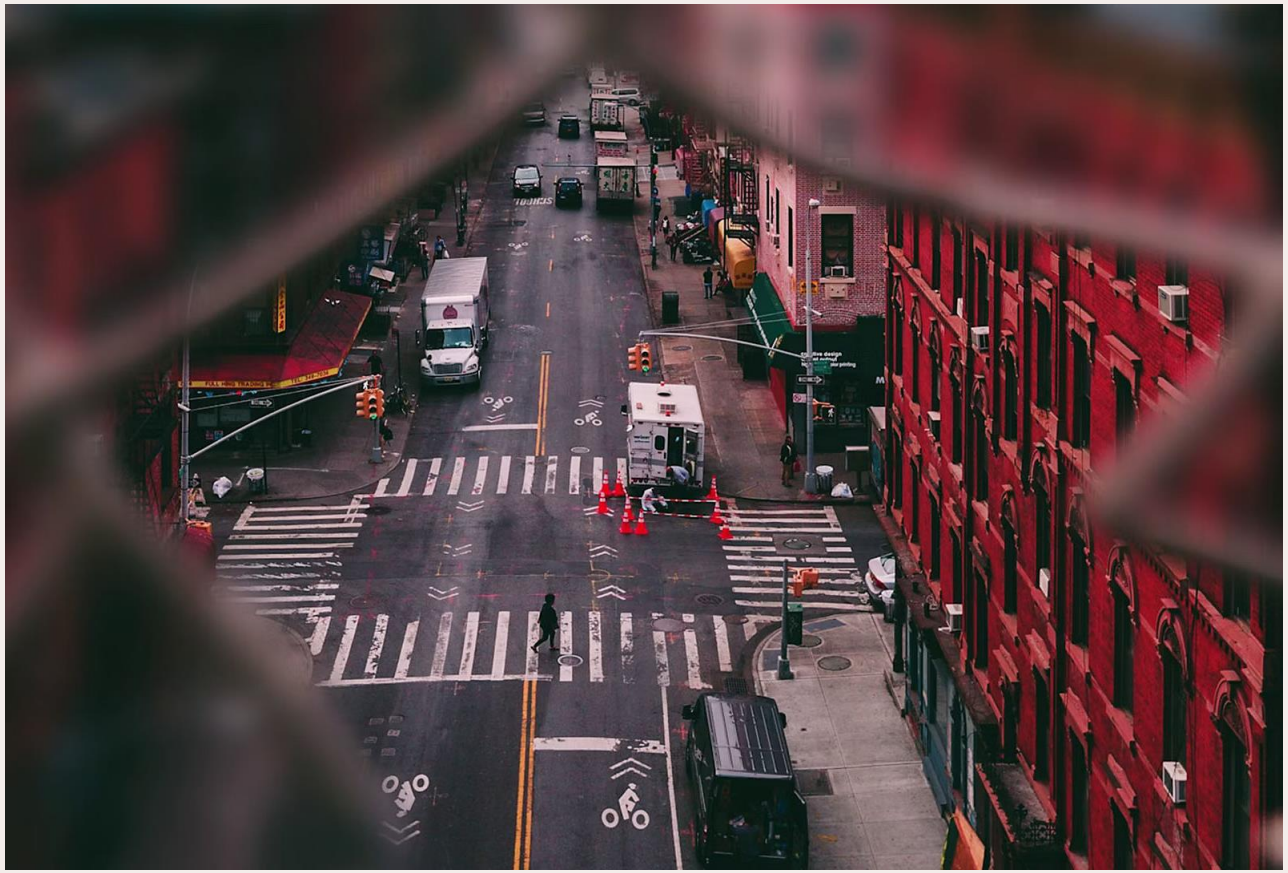
Performance Comparison of Serial vs. Parallel Gaussian Blur Implementation

- **Author**: Ramakrishna Gali
- **Id:** 02149393
- **Affiliation**: University of Massachusetts Dartmouth

Abstract

This study compares the execution performance of serial and parallel implementations of the Gaussian blur technique, commonly used in image processing. We measure execution time, speedup, and efficiency across multiple core counts (2, 4, and 8 cores) to evaluate the benefits of parallelization in reducing processing time.



Introduction

Objective:
This research aims to compare the performance of serial and parallel implementations of a Gaussian blur image processing technique. We evaluate execution time, speedup, and economy as we expand the task across multiple core counts (2, 4, and 8).

Gaussian Blur:
Gaussian blur is a key image processing technique for smoothing and removing noise in photographs. It is implemented using a convolution process that replaces the value of each pixel with a weighted average of its neighboring pixels, using a Gaussian kernel.

Parallelization:
Parallel computing can accelerate Gaussian blur operations by breaking tasks into smaller portions and processing them concurrently. This project makes use of MATLAB's Parallel Computing Toolbox, which supports multicore execution with functions such as parfor for parallel loops

Research Focus:
The research seeks to determine the best number of cores to balance performance and efficiency when parallelizing the Gaussian blur technique. The analysis includes calculating speedup (how much faster the parallel version is than the serial one) and efficiency (how well the system uses the available cores).

Methodology

•**Serial Implementation**:
The Gaussian blur algorithm is applied pixel by pixel using nested loops. Each pixel is updated by convolving it with the Gaussian kernel, resulting in a time complexity of $O(n^2)$.

•**Parallel Implementation**:
The parfor loop in MATLAB divides the image into rows and processes them in parallel across multiple cores. We tested with 2, 4, and 8 cores using MATLAB's Parallel Computing Toolbox.

•**Performance Metrics**:

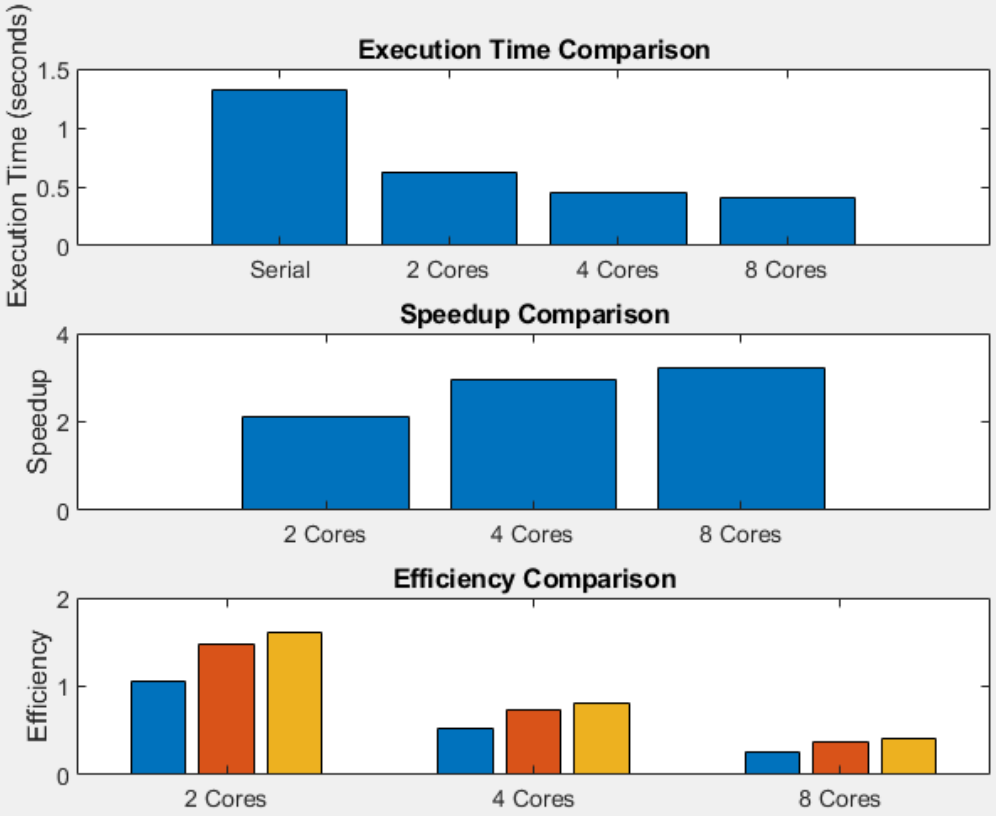
- •**Speedup**: Ratio of serial time to parallel time.
- •**Efficiency**: Speedup divided by the number of cores, showing how well the parallelism scales.

Numerical Results

- **Serial Execution**:
  - **Average Serial Time**: 1.3190 seconds
- **Parallel Execution (2 cores)**:
  - **Average Parallel Time**: 0.6225 seconds
  - **Speedup**: 2.12
  - **Efficiency**: 1.06
- **Parallel Execution (4 cores)**:
  - **Average Parallel Time**: 0.4454 seconds
  - **Speedup**: 2.96
  - **Efficiency**: 0.53
- **Parallel Execution (8 cores)**:
  - **Average Parallel Time**: 0.4105 seconds
  - **Speedup**: 3.21
  - **Efficiency**: 0.26

Comparison Graph

1. This slide will show graphical representations of execution times, speedup, and efficiency for various core counts. This helps the audience grasp the patterns and relationships between the serial and simultaneous operations.
2. Speedup Comparison: This graph depicts the speedup achieved with two, four, and eight cores. The speedup improves with the number of cores, with 4 and 8 cores producing similar speeds.
3. Efficiency Comparison: This graphic compares the efficiency of two, four, and eight cores. As the number of cores grows, the efficiency falls, with two cores having the best efficiency and eight cores having the lowest.



Conclusion

•**Parallel Computing Improves Performance**:
Parallel execution significantly reduces processing time. The **speedup** improves as more cores are utilized, with a peak speedup of **3.21** at **8 cores**. This demonstrates the potential of parallelism for performance enhancement in computational tasks like Gaussian blur.

•**Diminishing Returns with More Cores**:
While speedup increases with more cores, **efficiency** decreases, indicating diminishing returns. The **efficiency** dropped from **1.06** at **2 cores** to **0.26** at **8 cores**, due to the overhead and synchronization required in parallel tasks.

•**Optimal Core Usage**:
The results suggest that using 2 cores provides a good balance between **speedup** and **efficiency**, while the performance improvement slows down after 4 cores. It's essential to select the number of cores based on the trade-off between performance gains and efficiency losses.

•**Future Work**:
Future optimizations could focus on reducing **parallelization overhead** and improving **load balancing** between cores to achieve higher efficiency even with larger core counts. This would make the parallel implementation more scalable and effective for larger computational tasks.

References

- **https://en.wikipedia.org/wiki/Gaussian_blur**

- Parallel Computing in MATLAB
MATLAB documentation provides detailed resources on parallel computing techniques, including parallel loops and distributed computing.

- **https://www.mathworks.com/help/parallel-computing/**

Appendix

```
function main()
    % Specify the image path
    imagePath = "C:\Users\galir\Downloads\Gaussian_Blur.jpg";

    % Read the image
    input_image = imread(imagePath);

    % Convert to grayscale if necessary (optional)
    if size(input_image, 3) == 3
        input_image = rgb2gray(input_image);
    end

    % Set the sigma value for Gaussian blur
    sigma = 1;

    % Define number of repetitions for timing
    num_repeats = 10;

    % Number of cores to test
    core_counts = [2, 4, 8]; % Adjust based on your system's capabilities

    % Initialize arrays to store execution times and results
    serial_times = zeros(num_repeats, 1);
    parallel_times = zeros(num_repeats, length(core_counts));

    % Measure execution time for serial implementation using manual Gaussian blur
    for i = 1:num_repeats
        tic;
        output_image_serial = gaussian_blur_manual(input_image, sigma);
        serial_times(i) = toc;
    end

    % Measure execution time for parallel implementations with different core counts
    for c = 1:length(core_counts)
        num_workers = core_counts(c);

        % Check if a parallel pool already exists; if not, create one
        pool = gcp('nocreate'); % Get current parallel pool (if it exists)
        if isempty(pool)
            parpool(num_workers); % Start a parallel pool with specified number of workers
        end

        for i = 1:num_repeats
            tic;
            output_image_parallel = gaussian_blur_parallel_manual(input_image, sigma, num_workers);
            parallel_times(i, c) = toc;
        end

        % Optionally delete the pool after use (uncomment if desired)
        % delete(gcp('nocreate'));
    end
```

```
    % Calculate average times
    avg_serial_time = mean(serial_times);
    avg_parallel_times = mean(parallel_times);

    % Calculate speedup and efficiency for each core count
    speedup = avg_serial_time ./ avg_parallel_times;
    efficiency = speedup ./ core_counts'; % Divide by number of workers

    % Display results
    fprintf('Average Serial Time: %.4f seconds\n', avg_serial_time);

    for c = 1:length(core_counts)
        fprintf('Average Parallel Time (%d cores): %.4f seconds\n', core_counts(c), avg_parallel_times(c));
        fprintf('Speedup (%d cores): %.2f\n', core_counts(c), speedup(c));
        fprintf('Efficiency (%d cores): %.2f\n', core_counts(c), efficiency(c));
    end

    % Visualization of results
    figure;

    % Combine average times into a single array for plotting
    combined_avg_times = [avg_serial_time; avg_parallel_times(:)];

    % Execution Time Comparison Plot
    subplot(3, 1, 1);
    bar(combined_avg_times);
    set(gca, 'XTickLabel', [{'Serial'}, arrayfun(@(x) sprintf('%d Cores', x), core_counts, 'UniformOutput', false)]);
    ylabel('Execution Time (seconds)');
    title('Execution Time Comparison');

    % Speedup Plot
    subplot(3, 1, 2);
    bar(speedup);
    set(gca, 'XTickLabel', arrayfun(@(x) sprintf('%d Cores', x), core_counts, 'UniformOutput', false));
    ylabel('Speedup');
    title('Speedup Comparison');

    % Efficiency Plot
    subplot(3, 1, 3);
    bar(efficiency);
    set(gca, 'XTickLabel', arrayfun(@(x) sprintf('%d Cores', x), core_counts, 'UniformOutput', false));
    ylabel('Efficiency');
    title('Efficiency Comparison');
end

function output_image = gaussian_blur_manual(input_image, sigma)
   kernel_size = 2 * ceil(3 * sigma) + 1; % Calculate size of Gaussian kernel
   kernel_half_size = floor(kernel_size / 2);
   [X, Y] = meshgrid(-kernel_half_size:kernel_half_size, -kernel_half_size:kernel_half_size);
```

```
    % Create Gaussian kernel
    kernel = exp(-(X.^2 + Y.^2) / (2 * sigma^2));
    kernel = kernel / sum(kernel(:)); % Normalize the kernel

    [rows, cols] = size(input_image);
    output_image = zeros(rows, cols);

    % Apply convolution using the Gaussian kernel manually
    for i = 1:rows
        for j = 1:cols
            for kx = -kernel_half_size:kernel_half_size
                for ky = -kernel_half_size:kernel_half_size
                    if (i + kx > 0 && i + kx <= rows && j + ky > 0 && j + ky <= cols)
                        output_image(i,j) = output_image(i,j) + input_image(i + kx,j + ky) * kernel(kx + kernel_half_size + 1, ky + kernel_half_size + 1);
                    end
                end
            end
        end
    end

    output_image = uint8(output_image); % Convert back to uint8 if needed
end

function output_image = gaussian_blur_parallel_manual(input_image, sigma, num_workers)
    [rows, cols] = size(input_image);
    output_image = zeros(rows, cols);

    parfor i = 1:rows
        output_image(i,:) = gaussian_blur_manual(input_image(i,:), sigma);
    end

    output_image = uint8(output_image); % Convert back to uint8 if needed
end
```