Final term - Integral image

Author
Marten Schuitemaker
E-mail address
marten.schuitemaker@etu.univ-nantes.fr
Computer
Macbook air M1 (2020)

Abstract

Integral images, also known as summed-area tables, are widely used in computer vision and image processing for efficient computation of local image features and operations. This report presents an overview of integral images, their properties, and applications. Furthermore, it discusses sequential and parallel implementations of integral image computation, along with performance analysis and optimization strategies. Through experimentation and analysis, insights into the effectiveness and limitations of parallel computation techniques are provided.

Future Distribution Permission

The author(s) of this report give permission for this document to be distributed to Unifi-affiliated students taking future courses.
GitHub link : https://github.com/Martenren/final-term-pp.
The computer used on has for specs :
8-core CPU with 4 performance cores and 4 efficiency cores
Max CPU clock rate 3.2 GHz

1. Introduction to Integral Images

In computer vision and image processing, an integral image, also known as a summed-area table, is a data structure commonly used for efficient computation of local image features and operations.

1.1. Definition

An integral image of an input grayscale or color image $I(x, y)$ of size $W \times H$ is defined as:

$$S(x, y) = \sum_{i=0}^{x} \sum_{j=0}^{y} I(i, j)$$

where $S(x, y)$ represents the integral image and $I(x, y)$ represents the pixel intensity at coordinates (x, y).

1.2. Visual representation

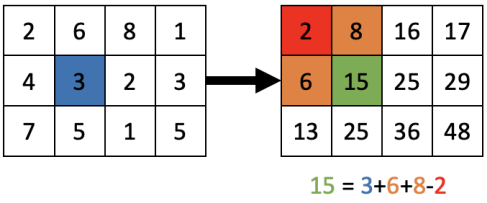

Figure 1. Example of the computation of an integral image

1.3. Properties

Integral images possess the following properties:

- They enable fast computation of the sum of pixel intensities within any rectangular region of an image.
- Computing the sum of pixel intensities over a rectangular region using an integral image requires only four array references, regardless of the size of the region.
- They facilitate the efficient computation of various image features, such as mean intensity, variance, and Haar-like features, which

are fundamental in object detection and recognition tasks.

1.4. Applications

Integral images find applications in various computer vision tasks, including but not limited to:

- **Object Detection:** Integral images enable rapid computation of features used in object detection algorithms such as Viola-Jones.
- **Image Filtering:** They facilitate efficient computation of box filters, Gaussian filters, and other neighborhood operations.
- **Feature Extraction:** Integral images are utilized for extracting features like local binary patterns (LBP), histogram of oriented gradients (HOG), and corner detectors.
- **Stereo Matching:** They aid in computing matching costs for stereo correspondence by efficiently evaluating local similarity measures.

In summary, integral images serve as a cornerstone in accelerating various image processing tasks, offering computational efficiency without compromising accuracy.

2. Sequential Code Implementation

To implement the computation of integral images for a color image, we utilize Python programming language along with NumPy library for efficient array manipulation. In the code found on the GitHub we have a function, `integral_image_color`, takes an input color image and returns the integral images for each color channel separately.

2.1. Function Overview

The `integral_image_color` function performs the following steps:

1. Converts the input image into a NumPy array (`image_array`).
2. Defines an inner function `calculate_integral_image` responsible for computing the integral image of a single color channel.
3. Initializes arrays for each color channel (red, green, and blue) by extracting them from the input image array.
4. Computes the integral images for each color channel using the `calculate_integral_image` function.
5. Returns the integral images for each color channel separately.

2.2. Algorithm Description

The algorithm for computing the integral image ($S(x, y)$) of a given color channel follows the integral image computation formula. It iterates over each pixel in the image and calculates the sum of pixel intensities within the rectangular region from the top-left corner to the current pixel.

1. Initialize an empty array for the integral image.
2. Traverse through each pixel in the input image.
3. For each pixel, compute the cumulative sum of pixel intensities from the top-left corner to the current pixel position.
4. Update the integral image array with the calculated sum.
5. Repeat steps 2–4 for all pixels in the image.

2.3. Code Overview

The provided Python code utilizes nested loops to traverse through each pixel in the input image. The `calculate_integral_image` function handles the computation of the integral image for a single color channel. The main function then iterates over each color channel (red, green, and blue) and computes the integral image for each.

2.4. Efficiency Considerations

While the sequential implementation provides a straightforward approach to computing integral images, it may suffer from performance limitations, especially for large images. The nested loops iterate over each pixel individually, resulting in a time complexity of $O(W \times H)$, where W and H represent the width and height of the image, respectively. This can lead to suboptimal performance for high-resolution images.

2.5. Conclusion

The sequential code implementation successfully computes integral images for a color image, providing a foundation for further optimization and parallelization. However, to improve computational efficiency, parallelization techniques or optimized algorithms may be explored.

3. Parallel Integration: Large Array of Images

To leverage the computational power of multiple CPU cores, we employ parallel processing techniques for computing integral images of a large array of images. The Python `multiprocessing` module is utilized to distribute the workload across available CPU cores.

3.1. Approach

The parallel integration process involves the following steps:

1. Iterate over the number of CPU cores from 1 to the maximum available.
2. For each CPU core count, create a pool of processes using the `Pool` class from the `multiprocessing` module.
3. Distribute the array of images (`image_array`) across the processes using the `map` function of the `Pool` object.
4. Measure the execution time for the parallel computation.
5. Store the execution time for each CPU core count for analysis.
6. Close the process pool to release system resources.

3.2. Code Implementation

The parallel integration is implemented using the following Python code found on the GitHub repository.

In this code snippet, the `max_cpu` variable represents the maximum number of CPU cores available for parallel processing. The computation time is measured for each CPU core count, and the results are stored in the `times` dictionary for further analysis.

3.3. Discussion

The parallel integration approach aims to reduce the overall computation time by distributing the workload across multiple CPU cores. By utilizing parallel processing, we expect to achieve significant speedup, especially for large arrays of images. However, the effectiveness of parallelization may be influenced by factors such as the number of available CPU cores, the size of the image array, and the computational complexity of the integral image computation algorithm. In the next section, we will analyze the performance results obtained from the parallel integration approach and compare them with the sequential implementation.

4. Results: Sequential vs. Parallel Computation

In this section, we present the results of comparing the performance of sequential and parallel computation for batches of images. We vary the number of images in the batch and evaluate the execution time for different numbers of CPU cores.

4.1. Experimental Setup

We utilize a batch of color images, each representing a distinct computational workload. The batch size ranges from 3 to 19 images, with each image being identical. The experiments are conducted on a system with a maximum of N CPU

cores, where N represents the number of available CPU cores.

4.2. Performance Comparison

We measure the execution time for both sequential and parallel computation approaches across varying batch sizes and CPU core counts. The following figures illustrate the comparison of execution time for different batch sizes using sequential and parallel computation.

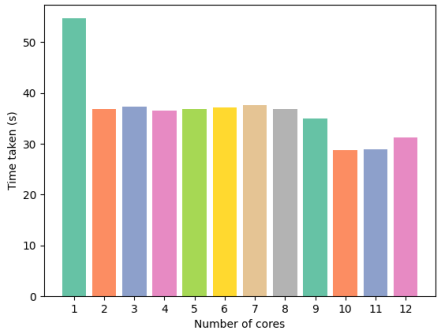

Figure 2. Comparison of execution time for sequential and parallel computation, batch size = 2

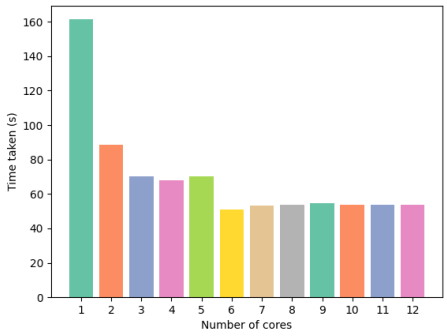

Figure 3. Comparison of execution time for sequential and parallel computation, batch size = 6

4.3. Discussion

As depicted in the Figures 2 3 4, the performance of the parallel computation approach exhibits notable improvements over sequential computation as the batch size increases. This is particularly evident when a larger number of CPU cores are utilized.

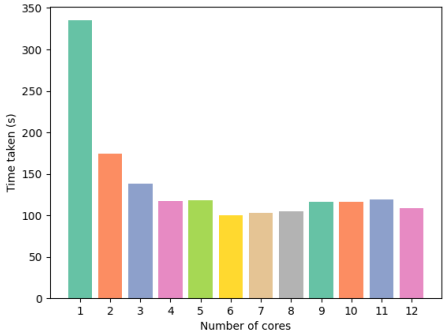

Figure 4. Comparison of execution time for sequential and parallel computation, batch size = 12

With smaller batch sizes, the overhead of parallelization and inter-process communication may outweigh the benefits of parallel computation, resulting in comparable or slightly worse performance compared to sequential computation. However, as the batch size grows, the parallelization overhead becomes negligible, and the benefits of parallel computation become more pronounced.

Furthermore, increasing the number of CPU cores leads to significant reductions in execution time, demonstrating the scalability of the parallel computation approach. However, diminishing returns are observed beyond a certain threshold, as the parallelization efficiency may decrease due to factors such as resource contention and communication overhead.

Overall, the results highlight the effectiveness of parallel computation for accelerating image processing tasks, particularly with larger batches and optimal CPU core utilization.

5. Parallelizing Computation of a Single Image

To further enhance the parallelization strategy, an attempt is made to parallelize the computation of integral images for a single image. This approach aims to exploit parallelism at a finer granularity level within the integral image computation algorithm.

5.1. Approach

The parallelization of a single image's integral image computation involves the following steps:

1. Divide the computation into two phases: column-wise and row-wise computation.
2. For each phase, create separate processes to handle parallel computation.
3. After completing column-wise and row-wise computation, combine the results.
4. Parallelize the computation of diagonal elements in the integral image.
5. Utilize multiple processes to compute diagonal elements concurrently.

5.2. Code Implementation

The Python code found on the GitHub repository demonstrates the implementation of parallel computation for a single image's integral image: In the provided code, the `compute_integral_image_parallel` function orchestrates the parallel computation process for a single image's integral image. It divides the computation into column-wise, row-wise, and diagonal computation phases, utilizing multiprocessing techniques to achieve parallelism.

5.3. Visual representation

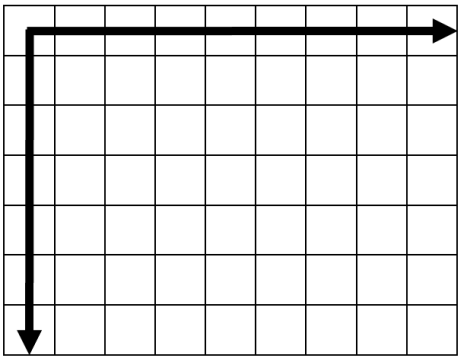

Figure 5. Parallel computation of first row and first column

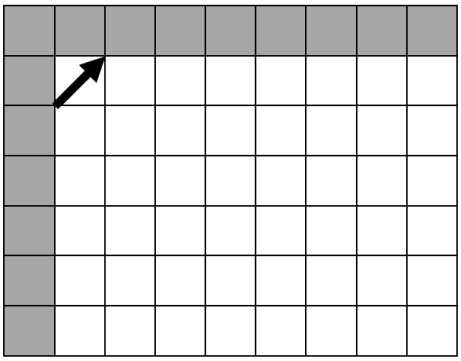

Figure 6. Example of computation of diagonal (t=1)

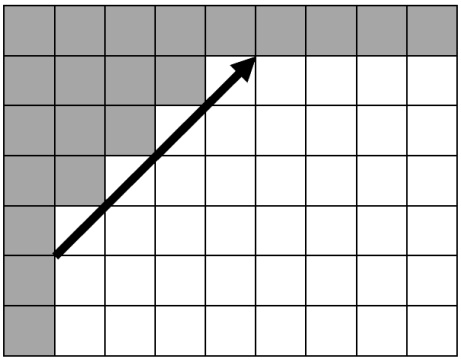

Figure 7. Example of computation of diagonal (t=4)

5.4. Discussion

The attempt to parallelize the computation of a single image's integral image aims to further exploit parallelism within the integral image computation algorithm. By dividing the computation into finer-grained tasks and utilizing multiple processes, we expect to achieve improved performance and scalability, particularly for large images.

In the subsequent section, we will analyze the performance results obtained from this enhanced parallelization approach and compare them with the previous implementations.

6. Results: Parallel Computation Analysis

In this section, we analyze the performance of the parallel computation approach compared to sequential computation for a single image. Despite the parallelization attempt, the parallel computation exhibits slower performance compared

to sequential computation. Several factors contribute to this observation:

1. **Overhead of Process Creation**: Creating and managing multiple processes incurs overhead, including process creation, context switching, and inter-process communication. This overhead can outweigh the benefits of parallelization, especially for small computational tasks (which is our case since our tasks are small sums).
2. **Inter-Process Communication (IPC) Overhead**: In the provided implementation, interprocess communication is used to exchange data between processes. This communication overhead can become significant, particularly when transferring large data structures like the integral image between processes.
3. **Granularity of Parallelization**: The granularity of parallelization may not be optimal, leading to synchronization overhead and limited parallelism efficiency.
4. **Overhead of Subprocess Management**: Managing a subprocess pool involves additional overhead for distributing tasks and collecting results.
5. **GIL (Global Interpreter Lock)**: While multiprocessing avoids the GIL limitation of Python threads, it introduces additional overhead compared to multithreading.

Overall, these factors contribute to the slower performance of parallel computation compared to sequential computation for the given workload. Further optimization of the parallelization strategy and consideration of alternative parallelization approaches may be necessary to achieve improved performance.

7. Divide and Conquer Approach

Another strategy for parallelizing the computation of integral images involves a ”divide and conquer” approach. In this method, the image is partitioned into smaller chunks, and each chunk’s integral image is computed independently. Finally, the integral images of all chunks are combined to obtain the integral image of the entire image.

7.1. Chunk Partitioning

The first step in the divide and conquer approach is to partition the image into smaller chunks. Each chunk is processed independently, allowing for parallel computation across multiple cores or processes. The partitioning process divides the image into rectangular regions of equal size, ensuring balanced workload distribution.

7.2. Chunk Computation

Once the image is partitioned into chunks, the integral image of each chunk is computed separately. This computation follows a different less optimized algorithm but operates only on the pixels within the chunk. By processing smaller regions independently, parallelization efficiency is improved, as each chunk can be assigned to a separate processing unit.

7.3. Algorithm

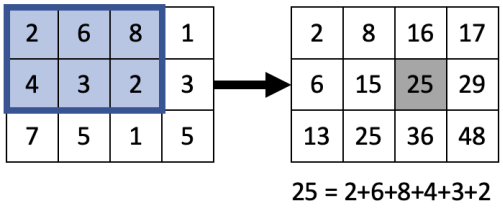

Figure 8. Integral sum

As shown in 8 value for the cell in our integral image is the sum of all the numbers in the shaded area. The sum of these numbers is 51, so let’s go ahead and put that in our Integral Image.

While this is way slower than our more optimized way of calculating the interal sum at a given point. It does not require precedent values to compute, as such we can parallelize without the need of data sharing. This might lead to an increase in speed compared to the sequential value.

7.4. Visual representation

As we can see in 9, multiple processes can work at the same time, calculating different val-

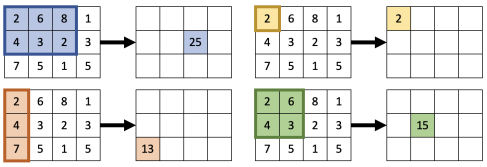

Figure 9. Example parallel computation

ues without the need to share data.

This allows us to divide into chunks our image and assign each process it's own chunk to work on (cf. 10).

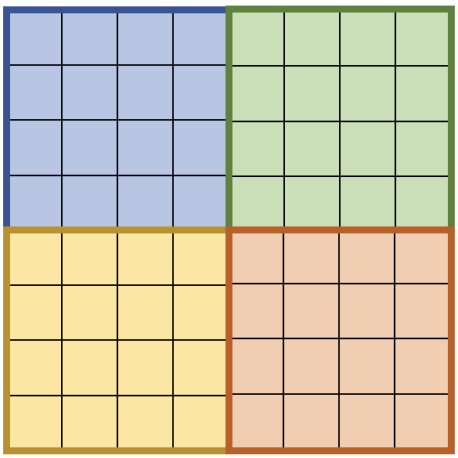

Figure 10. Divide and conquer (4 processes)

7.5. Combining Results

After computing the integral image for each chunk, the results are combined to obtain the integral image of the entire image.

7.6. Result

Despite the potential for parallelization offered by the divide and conquer approach, empirical evaluation reveals that it may not always outperform sequential computation. In fact, in many cases, the divide and conquer approach may even be slower due to various factors.

One significant factor contributing to the slower performance is the overhead associated with partitioning the image into chunks and combining the results. While parallelization allows for concurrent computation of individual chunks, the overhead of managing multiple processes, inter-process communication, and result aggregation can outweigh the benefits, especially for smaller images or computational tasks.

Furthermore, the chunk-based computation may introduce inefficiencies in workload distribution, leading to imbalanced processing times for different chunks. In scenarios where certain chunks require more computational resources or processing time than others, idle processors or uneven utilization of CPU cores may occur, diminishing the overall parallelization efficiency.

Moreover, the divide and conquer approach may not fully leverage the optimization opportunities available in sequential computation algorithms. While it eliminates the need for data sharing between processes, it may result in suboptimal utilization of computational resources and lack of exploitation of algorithmic optimizations.

In conclusion, while the divide and conquer approach offers a promising strategy for parallelizing the computation of integral images, its performance may be hindered by overheads, workload distribution inefficiencies, and suboptimal algorithmic utilization. Careful consideration of these factors is essential when evaluating the suitability of the divide and conquer approach for specific image processing tasks.

8. Vectorization

Vectorized operations revolutionize computational efficiency by performing computations on entire arrays (or vectors) of data simultaneously, rather than operating on individual elements sequentially. This approach harnesses the capabilities of modern CPUs and GPUs, which support SIMD (Single Instruction, Multiple Data) instructions. SIMD instructions enable processors to execute the same operation on multiple data elements in parallel, effectively exploiting parallelism at the hardware level.

8.1. How Vectorized Operations Work

Vectorized operations typically work through the following mechanisms:

- **Single Instruction, Multiple Data (SIMD):**

SIMD instructions allow processors to apply the same operation to multiple data elements in a single instruction cycle. For example, a SIMD instruction might add together corresponding elements from two arrays in a single operation, rather than performing separate additions for each pair of elements.

- **Data Parallelism:** Vectorized operations leverage data parallelism, where the same operation is simultaneously applied to multiple elements of an array. This enables efficient parallelization of computations across large datasets.

- **Optimized Implementations:** Libraries like NumPy provide optimized implementations of common mathematical operations using SIMD instructions and other low-level optimizations. These optimized implementations ensure that vectorized operations run efficiently on modern hardware.

- **Implicit Looping:** Vectorized operations eliminate the need for explicit loops over array elements, which can be slow in interpreted languages like Python. Instead, the operation is applied to the entire array at once, often using highly optimized, compiled code under the hood.

- **Broadcasting:** Vectorized operations can automatically handle operations between arrays of different shapes through a mechanism called broadcasting. Broadcasting allows NumPy to align arrays with different shapes by implicitly replicating elements along certain dimensions, enabling element-wise operations to be performed efficiently.

In summary, vectorized operations provide a powerful and efficient way to perform computations on arrays of data, leveraging hardware parallelism and optimized implementations to achieve high performance. They are a key feature of libraries like NumPy, enabling fast and concise code for numerical computing tasks in Python.

8.2. Performance analysis

In 11 we can see that vectorization is a huge increase in speed compared to sequential:

- 500x500 pixels: increase in speed of 1404%
- 1000x1000 pixels: increase in speed of 3395%
- 2000x2000 pixels: increase in speed of 3171%
- 3000x3000 pixels: increase in speed of 3262%
- 5600x3200 pixels: increase in speed of 2645%

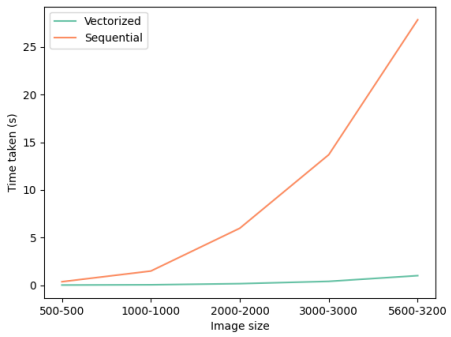

Figure 11. Vectorized computation vs Sequential computation on different image sizes

Also, when we take a batch of images (in this case 10, cf.12), we can also see huge increases of speed compared to batch parallelization: We also tried to parallize the vectorization on multiple images, but as one could expect this is slower, since we are increasing the overhead and the numpy library already uses parallelization to increase the speed.

- 500x500 pixels: increase in speed vectorized vs batch parallel 1722%
- 1000x1000 pixels: increase in speed vectorized vs batch parallel 1952%
- 2000x2000 pixels: increase in speed vectorized vs batch parallel 1592%

- 3000x3000 pixels: increase in speed vectorized vs batch parallel 1851%
- 5600x3200 pixels: increase in speed vectorized vs batch parallel 1354%

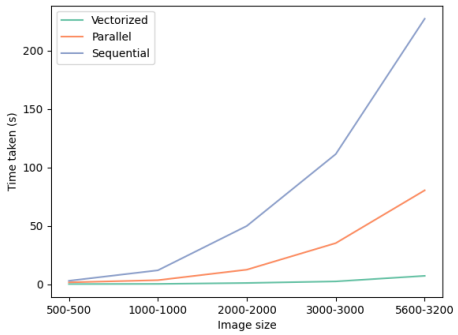

Figure 12. batch of 10 images: Vectorized vs Batch Parallel (8 cores)

9. Conclusion

In conclusion, this report has provided an in-depth exploration of integral images, their computation, applications, and optimization strategies. We began by introducing integral images and their significance in computer vision and image processing tasks. Sequential and parallel implementations of integral image computation were discussed, along with their respective performance analyses and optimization considerations.

Through experimentation and analysis, insights were gained into the effectiveness of parallel computation techniques, including parallel integration of a large array of images and attempts to parallelize computation at a finer granularity level. While parallelization strategies showed promise for improving computational efficiency, factors such as overheads, workload distribution inefficiencies, and algorithmic limitations were identified as challenges that may impact performance.

Furthermore, the report delved into the concept of vectorization and its profound impact on computational efficiency. By leveraging vectorized operations, significant speedups were achieved compared to both sequential and parallel computation approaches, particularly for large datasets and image sizes. The performance analysis demonstrated the remarkable benefits of vectorization, highlighting its role as a key feature in libraries like NumPy for numerical computing tasks in Python.

In summary, this report underscores the importance of efficient computation techniques in image processing and computer vision applications. While traditional sequential and parallel approaches offer valuable insights, the advent of vectorized operations presents a paradigm shift in computational efficiency, enabling faster and more scalable solutions for handling large datasets and complex image processing tasks.