Integral Images with Java Threads and CUDA

Giulia Bertazzini
giulia.bertazzini@stud.unifi.it

Niccolò Guiducci
niccolo.guiducci@stud.unifi.it

Abstract

The aim of this project is to calculate the integral image of a given RGB image. We propose two sequential and two parallel implementations, in order to measure how the parallel versions can improve the performances.
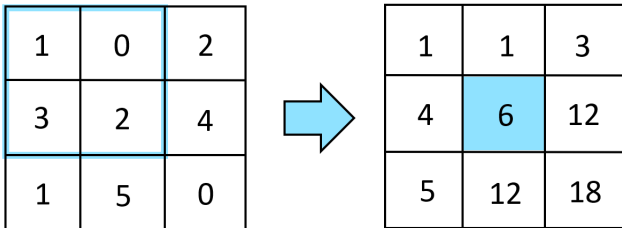
1 Introduction

Given an RGB image as input, in this project, we implemented two sequential programs e two parallel ones in order to calculate the corresponding integral image: the first parallel version we propose uses **Java Threads** [4], while the second one uses C++ **CUDA** [1], as regards the two sequential versions, these are nothing more than implementations in different languages, C ++ and Java, of the same algorithm. In this way, we can estimate how the parallel implementations can improve the performances, measured in terms of mean execution time and speedup.

2 Integral Image

An integral image [3] is a data structure and algorithm for quickly and efficiently generating the sum of values in a rectangular subset of a matrix; it means that in an integral image the value at any point (x, y) is the sum of all the pixels above, to the left of (x, y) and the pixel value itself. If $i(x, y)$ represents the value of the pixel (x, y) of the

Figure 1: Example of integral image.

original image, then the value of the pixel (x, y) in the integral image I is obtained as follow:

$$I(x, y) = \sum_{\substack{x' < x \\ y' < y}} i(x', y')$$

However, this can be computed efficiently in a single pass over the image, as the value in the integral image at (x, y) is

$$I(x, y) = i(x, y) + I(x, y - 1)$$
$$+I(x - 1, y) - I(x - 1, y - 1)$$

3 Implementation

In this section, we describe the sequential and parallel implementations of integral images. In particular, we propose a sequential version in Java and the corresponding parallel version with Java Threads, and a second parallel version in CUDA: in this last case, we also propose another sequential implementation in C++, for a better comparison between the sequential and parallel implementations.

In all cases, since an integral image is calculated from a grayscale one, it is needed initially to convert the color space of the input image into a grayscale one.

3.1 Sequential Implementation

First of all we implemented two sequential programs to calculate an integral image, based on the same algorithm, given an input RGB image. In particular, we propose a sequential implementation in Java and another one in C++, in order to better compare the parallel versions, described later, with the corresponding sequential one. In both cases, the primary step is to convert the color input image, since that an integral image is calculated starting from a grayscale one. After that, we can fill the matrix which represents the integral image getting pixels' value from the original image (in grayscale) and properly summing them.

3.1.1 Java sequential implementation

Regarding the Java implementation, we defined a class named "**GreyImage**" that contains the following methods:

- the **constructor** of the class, which, given the path to an image file, converts it into a grayscale image and calculates the corresponding matrix;

- **getPixel(int x, int y)** which simply return the grey level of pixel at position (x,y);

- **calculateIntegralImage()** which compute the integral image with the sequential algorithm: to do this, as we can see from the code below, it copies the value of the pixel (0,0) of the original image in a matrix representing the integral image (which has the same dimensions of the original image), fills the first column and the first row in the matrix with the relatives subsequent sums and then it properly calculates the other cells values;

```java
public int[][]
    calculateIntegralImage(){
int[][] integralImage = new
    int[height][width];

//Fill the first cell
integralImage[0][0] =
    this.getPixel(0,0);

//Fill the first column
for(int i = 1; i < height ; i++)
   integralImage[i][0] =
       integralImage[i - 1][0] +
       this.getPixel(0,i);

//Fill the first row
for(int j = 1; j < width ; j++)
   integralImage[0][j] =
       integralImage[0][j-1] +
       this.getPixel(j, 0);

for(int i=1; i < height; i++)
   for(int j=1; j < width ; j++)
      integralImage[i][j] =
          this.getPixel(j,i) +
          integralImage[i][j-1] +
          integralImage[i-1][j] -
          integralImage[i-1][j-1];

   return integralImage;
}
```

- **calculateParallelIntegralImage(int numThreads)** which, as suggested by the name, calculates the integral image in a parallel way. We discuss more in details about this method in section [3.2.1].

Beyond that, we defined another class named "**sequentialTest**" which is responsible to test the performances of the sequential implementation, measured in terms of mean, maximum and minimum execution time. We refer to section [4] and [5] for more details about the experiments made and the results obtained.

3.1.2 C++ sequential implementation

As mentioned before, we propose another sequential implementation in C++, to better compare the results obtained with the parallel implementation in CUDA [3.2.2]. Even in this case, we implemented a class named "**GreyImage**" with a constructor that, given the color image, reads and converts it in a grayscale one (to do this we used the open-source library **OpenCV**), and other methods to get the attributes of this class; in particular, also in this case, we implement the method **getPixel(int x, int y)** to return the grey level value of pixel at position (x, y) (subsequently, as we will see in the code, since the image is treated by OpenCV as a matrix it was necessary to invert the indices, so the pixel in position (x, y) in the original image will be stored inside the matrix in position [y][x], since the coordinate x of the pixel represents the column's index and the coordinate y the row's index).
Module "**SequentialIntegralImage.h**" contains two methods:

- the method **sequentialIntegralImage**, which, given as parameter a GreyImage, sequentially calculates the integral image, in the same way described for Java sequential implementation;

- **timeSequentialIntegralImage** which is responsible to calculate the execution time of this sequential implementation; more details are presented in sections [4] and [5].

3.2 Parallel Implementation

To measure how parallelism can improve the performances, in this project we propose two parallel implementations: the first one make use of Java Threads while the second one make use of CUDA. In both cases, the naive approach was to decompose the double summation which is the basis of the formula of the integral image, i.e. we proceed to sum the elements of the matrix first by columns and then by rows; however, this approach is not cache friendly as the elements of the matrix on the same column are located in non-consecutive memory locations. To overcome this problem, which would have affected
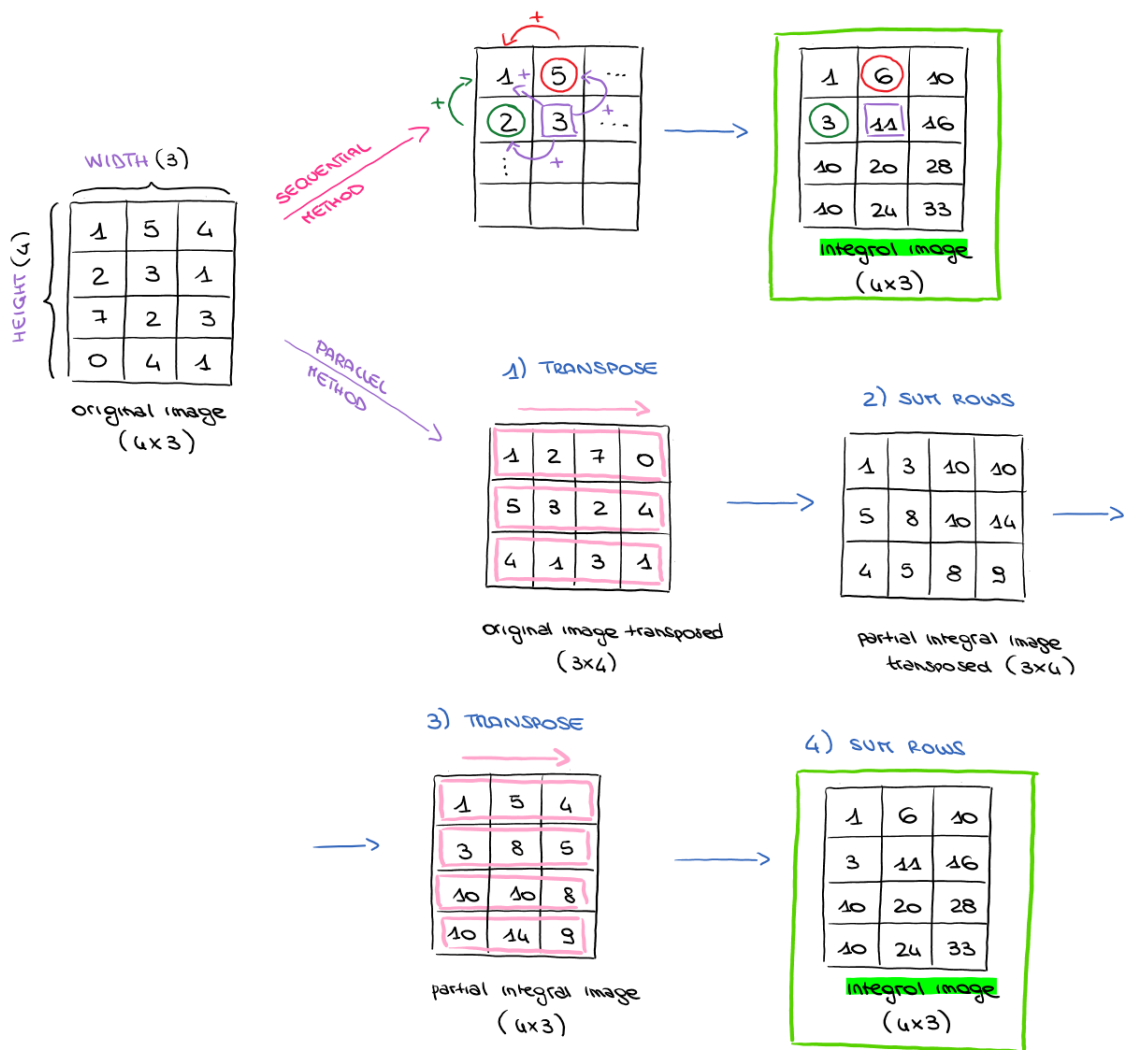
Figure 2: Explanation of the parallelization implemented with Java Threads and CUDA.

the results considerably (especially in CUDA since data readings are not coalesced), it was decided to always add by rows by transposing the matrix of grey levels.

This is a summary of the parallelization method (figure 2 tries to make the idea of this approach):

- first step is to **transpose** the image grayscale matrix (if the input image is *width*height*, the transposed one is *height*width*);

- second step is to properly **sum each row**: it means that each pixel in a row is the subsequent sum of all the pixel to the left of it (the previous ones);

- third step is to **transpose** the partial integral image obtained from step two, to get back image back to being *width*height*;

- last step is to **sum again each row**, same as the second step.

As we can see from the figure 2 the result obtained is the same as the integral image obtained using the sequential approach, previously explained.

3.2.1 Java Threads

Regarding the parallel implementation with **Java Threads**, the code below is the one that calculates the integral image using the method previously explained.

```
public int[][]
   calculateParallelIntegralImage(int
   numThreads) throws InterruptedException {
      int[][] integral_image;
      int[][] transpose;
```

```java
int columns_per_thread = (int)
    Math.floor(width / numThreads);
int rows_per_thread = (int)
    Math.floor(height / numThreads);

// 1) transpose the original image
transpose =
    transpose(this.image_matrix,
    height, width);

// 2) sum each row on the transposed
    image
SumRowIntegralImage[] threads = new
    SumRowIntegralImage[numThreads];

for (int i = 0; i < numThreads - 1;
    i++)
    threads[i] = new
        SumRowIntegralImage(transpose, i
        * columns_per_thread, (i *
        columns_per_thread)+
    columns_per_thread, height);
threads[numThreads-1] = new
    SumRowIntegralImage(transpose,
    (numThreads - 1) *
    columns_per_thread, width, height);

for (SumRowIntegralImage thread :
    threads)
    thread.start();

for (SumRowIntegralImage thread:
    threads)
    thread.join();

// 3) transpose the partial integral
    image
integral_image = transpose(transpose,
    width, height);

// 4) sum each rows on the partial
    integral image
threads = new
    SumRowIntegralImage[numThreads];

for (int i = 0; i < numThreads - 1;
    i++)
    threads[i] = new
        SumRowIntegralImage(integral_image,
        i * rows_per_thread, (i *
        rows_per_thread)+
        rows_per_thread, width);
threads[numThreads-1] = new
    SumRowIntegralImage(integral_image,
    (numThreads - 1) * rows_per_thread,
    height, width);

for (SumRowIntegralImage thread :
    threads)
    thread.start();

for (SumRowIntegralImage thread:
    threads)
    thread.join();
```

```java
    return integral_image;
}
```

As we can see, each thread is responsible to make the sums of a certain number of rows; threads can be run by creating an instance of the class *SumRowIntegralImage* (that extends the Java class *Thread*) and calling the start() method. Then, calling on each thread the join() method, they wait until all the other thread complete their work. This waiting is necessary, since step 3 and 4 of the parallelization method proposed can't be executed until the partial integral image is completed; once all thread have finished are executed step 3 and 4, in the same way of the previous two.

Notice that the calculation of the transpose is also parallelized with IntStream that can be associated with a C++ for loop with OpenMP directives, as we can see from the code below.

```java
public int[][] transpose(int[][] matrix, int
    height, int width){
    int[][] transpose = new
        int[width][height];
    IntStream.range(0, width *
        height).parallel().forEach(i ->
    {
        int m = i / height;
        int n = i % height;
        transpose[m][n] = matrix[n][m];
    });
    return transpose;
}
```

Classes *sequentialTest* and *parallelTest* are the ones responsible to implement, respectively, the sequential and the parallel test, which are discussed more in details in sections [4] and [5].

3.2.2 CUDA

CUDA (or Compute Unified Device Architecture), created by NVidia, is a parallel computing platform and API that allows software to use certain types of GPUs for general purpose processing. In this project, we used CUDA to implement an additional parallel version of integral image, following the same method explained in section [3.2].

A typical CUDA program structure consists of five main steps:

- allocate CPU and GPU memories;

- copy data from CPU memory (host) to GPU memory (device);

- invoke the CUDA functions, called **kernel**, to perform program-specific computation;

- copy data back from GPU memory to CPU memory;

- destroy GPU memories.

As we can see from the code below, the typical structure has been respected. We can notice that we added a warm up section, in which we execute one time the kernels, discarding the results. This was made besides the GPU being in a power saving state, there are other reasons why the first launch of a kernel could be slower than further runs, such as just-in-time compilation, transfer of kernel to GPU memory, cache content and so on. In this way, we measure the effective time of kernels' execution, excluding the memory allocation and data transfer.

```cpp
double timeParallelIntegralImage(GreyImage
    image, const int thread_per_block){
  int height = image.getHeight();
  int width = image.getWidth();

  // Allocate host memory
  int* original_image = image.getImage();
  int size = height*width;
  int* parallel_integral_image = new
      int[size];

  int* d_original_image;
  int* d_integral_image;
  int* d_transpose;

  int block_num_for_cols = (width +
      thread_per_block)/thread_per_block;
  int block_num_for_rows =
      (height+thread_per_block)/thread_per_block;

  cudaHostRegister(&original_image, size,
      cudaHostRegisterDefault);
  cudaHostRegister(&parallel_integral_image,
      size, cudaHostRegisterDefault);

  dim3 grid((width + BLOCK_DIM)/ BLOCK_DIM,
      (height+BLOCK_DIM)/ BLOCK_DIM, 1);
  dim3 threads(BLOCK_DIM, BLOCK_DIM, 1);
  dim3 grid_after_trasp((height+BLOCK_DIM)/
      BLOCK_DIM, (width +
      BLOCK_DIM)/BLOCK_DIM, 1);

  // Allocate device memory
  cudaMalloc((void**)&d_original_image,
      sizeof(int) * width * height);
  cudaMalloc((void**)&d_integral_image,
      sizeof(int) * width * height);
  cudaMalloc((void**)&d_transpose,
      sizeof(int) * width * height);

  // Transfer data from host to device
      memory
  cudaMemcpy(d_original_image,
      original_image, sizeof(int) * width *
      height, cudaMemcpyHostToDevice);

  //warm-up
  transpose<<<grid,threads>>>(d_transpose,
      d_original_image, width, height);
  CUDASumRowIntegralImage<<<block_num_for_cols,
      thread_per_block>>>(height, width,
      d_transpose);
  transpose<<<grid_after_trasp,threads>>>
  (d_integral_image,d_transpose, height,
      width);
  CUDASumRowIntegralImage<<<block_num_for_rows,
      thread_per_block>>>(width, height,
      d_integral_image);
  cudaDeviceSynchronize();

  // Executing kernels
  auto start =
      std::chrono::system_clock::now();

  transpose<<<grid,threads>>>(d_transpose,
      d_original_image, width, height);
  CUDASumRowIntegralImage<<<block_num_for_cols,
      thread_per_block>>>(height, width,
      d_transpose);
  transpose<<<grid_after_trasp,threads>>>
  (d_integral_image,d_transpose, height,
      width);
  CUDASumRowIntegralImage<<<block_num_for_rows,
      thread_per_block>>>(width, height,
      d_integral_image);
  cudaDeviceSynchronize();

  std::chrono::duration<double> diff{};
  diff = std::chrono::system_clock::now() -
      start;

  // Transfer data back to host memory
  cudaMemcpy(parallel_integral_image,
      d_integral_image, sizeof(int) * width
      * height, cudaMemcpyDeviceToHost);

  // Deallocate device memory
  cudaFree(d_original_image);
  cudaFree(d_integral_image);
  cudaFree(d_transpose);

  delete[] parallel_integral_image;
  delete[] original_image;

  return diff.count();
}
```

With CUDA, threads are grouped into blocks and the entirety of blocks is called grid. To properly sum the matrix image rows, we implemented a specific kernel that uses one thread for each row of the matrix. In order to have a total number of threads at least equals to the first matrix dimension, we give to this methods the desired threads' number per block as parame-

ter (*thread_per_block* in the code), and then we properly calculate the number of block needed to cover all the matrix rows. More in details, these blocks are organized as a mono dimensional grid, that is a grid of dimension $1 \times ((dimension/thread\_num) + 1)$.

Notice that to execute the kernel **transpose** (which code is subject to NVIDIA ownership [2]), that calculates the transpose matrix of the image, we used a different grid dimension and different number of thread in each block.

4 Tests

All of the following tests are executed on a computer with the following specifications:

- OS: Microsoft Windows 10 Pro

- CPU: 11th Gen Intel(R) Core(TM) i9-11900KF @3.5GHz(TBoost 5.1GHz) Octa-Core

- GPU: GeForce RTX 3070Ti

- RAM: 32GB DDR4 @3600 MHz

- MOBO: Asus PRIME Z490-A

Tests consider mean, maximum and minimum execution times which are calculated by repeating the computation of integral image 100 times for each test image for every implementation proposed. In particular, the test images differs in size: we consider the same image scaled at different resolution (480p, 720p, 1080p, 1440p and 2160p, where "p" means that the image, with 16:9 of aspect ratio, has a vertical resolution of 480, 720, and so on).

Regarding the parallel approaches, we repeat these test also changing the threads' number.

5 Results

In this section we discuss about the results achieved by the tests. As expected, in both cases, the results achieved following the parallel approaches are significantly better than the sequential ones.

5.1 Java Implementation Results

Figure 3 shows the execution times obtained with Java Threads, repeating tests on images with different size and varying the threads' number. As we can notice, without any surprise, the bigger the image, the more the execution time increases.

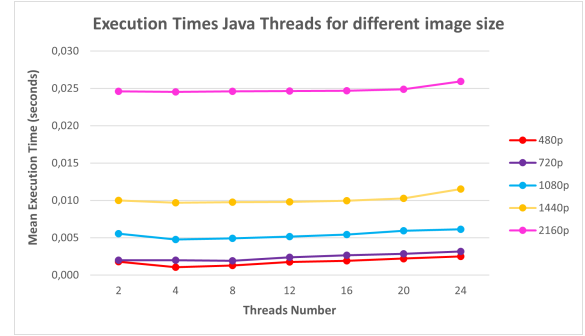Figure 4 compares the execution times obtained with the different image size, using both the sequential and

Figure 3: Comparison of execution times with Java Threads for different image size.
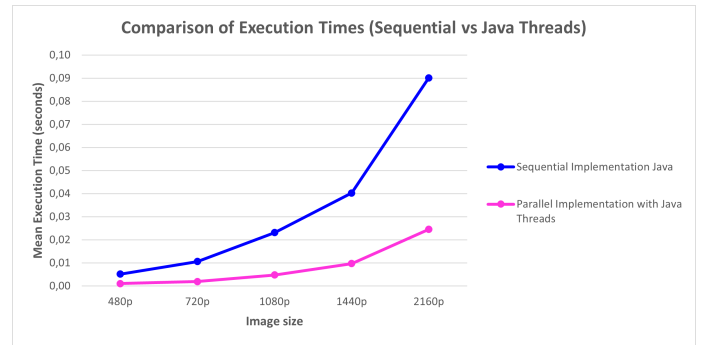

Figure 4: Comparison of execution times for each image size between the sequential and parallel approach in Java. For parallel implementation, for each image size we considered the minimum time obtained varying threads' number.

the parallel approach. In particular, regarding the parallel implementation with Java Threads, in this graphic are represented the minimum execution times obtained from the previous test, varying the number of threads. We can see that we achieved with the parallel approach good improvements mostly with the biggest image, as we expected.
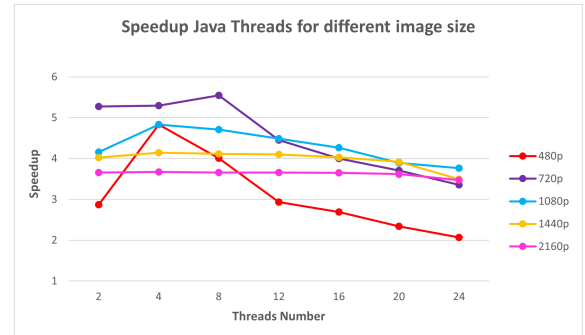

Figure 5: Comparison of speedup with Java Threads for different image size.

Finally, figure 5 compares the different speedup ob-

tained for each image size, varying the number of threads. The speedup is obtained as $S_P = t_S/t_P$, where P is the number of processors, t_S is the completion time of the sequential algorithm and t_P is the completion time of the parallel algorithm. We can see that we achieved the best results with the 720p image, using 8 threads: indeed in this case we obtained a 5.55 of speedup.

5.2 CUDA Implementation Results

Figure 3 shows the execution times obtained with CUDA, repeating tests on images with different size and varying the threads' number. Clearly, even in this case, the bigger the image, the the more the execution time increases, same as with Java Threads.
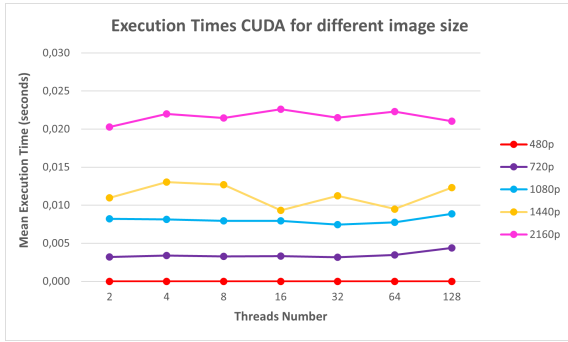

Figure 6: Comparison of execution times with CUDA for different image size.

Figure 7 compares the execution times obtained with the different image size, using both the sequential and the parallel approach. Once again, regarding the parallel implementation with CUDA, in this graphic are represented the minimum execution times obtained from the previous test, varying the number of threads. We can see that, even in this case, we achieved with the parallel approach good improvements mostly with the biggest image, as we expected.

Finally, figure 8 shows the speedup obtained with CUDA. We can notice that, in general, we achieved great results (in fact all the speedups exceed 5), especially with the smallest image size.

5.2.1 Comparison between parallel implementations

In the end, we can compare the parallel approaches proposed in this project. Although the minimum execution times are not too different, observing the graphic in figure 9, we can say that the approach which achieved the best results is the one with Java Threads. This is due also to the fact that if we look at the graphic 10, we notice that
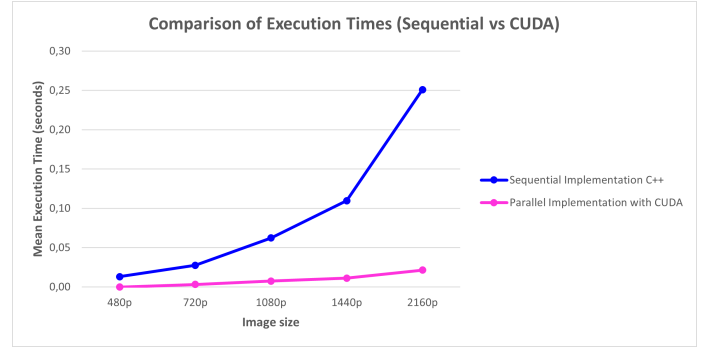

Figure 7: Comparison of execution times for each image size between the sequential and parallel approach in CUDA. For parallel implementation, for each image size we considered the minimum time obtained varying threads' number.
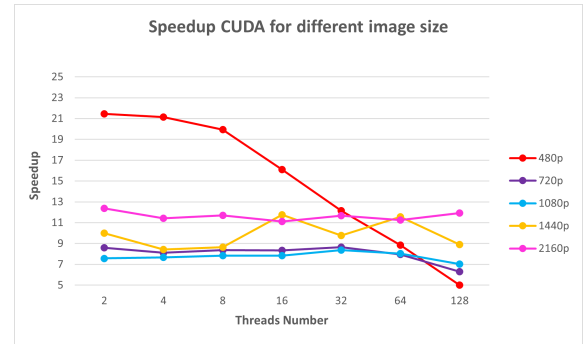

Figure 8: Comparison of speedup with CUDA for different image size.

the sequential implementation in Java has smaller mean execution times than the one in C++.
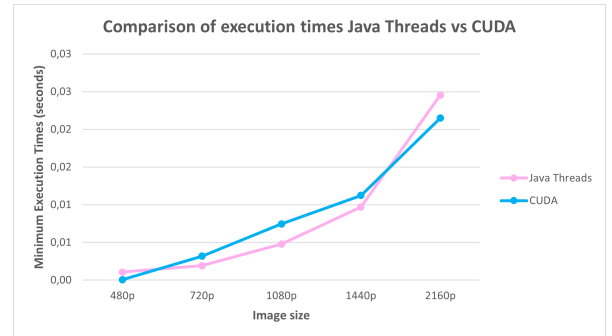

Figure 9: Comparison of mean execution times between Java Threads and CUDA.

6 Conclusion

Without any surprise, both parallel approaches improve the performances in terms of execution time. From the results obtained, the best approach turned out to be the one with Java Threads, since even the sequential implemen-
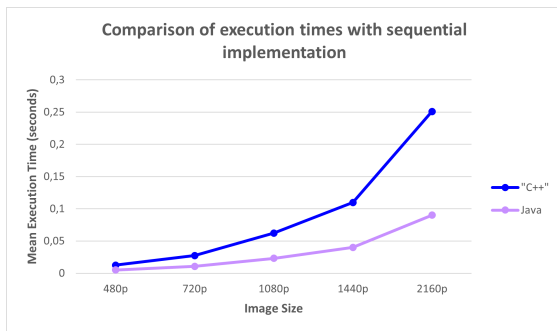
Figure 10: Comparison of mean execution times between sequential implementation in Java and in C++.

tation gave better results with respect to C++ ones. For a better overall consideration, these conclusions derived from tests that calculate the integral image on the same image at different resolution, once a time: actually, doing the test using more images at the same time, we would probably achieve better results with CUDA, since that could be used asynchronous memory's copies and kernels in different streams, taking advantage of all CUDA's potential.

References

[1] CUDA. https://developer.nvidia.com/blog/easy-introduction-cuda-c-and-c/.

[2] CUDA NVidia matrix transpose. https://github.com/JonathanWatkins/CUDA/blob/master/NvidiaCourse/Exercises/transpose/transpose.cu.

[3] Integral Image. https://it.wikipedia.org/wiki/Immagine_integrale.

[4] Java Threads. https://docs.oracle.com/javase/7/docs/api/java/lang/Thread.html.