UNIVERSITAT POLITÈCNICA DE VALÈNCIA

etsinf

Parallel Computing (CPA)

Lab 2: Image scaling with Lanczos interpolation

Year 2019/20

Contents

Warning

The report and source code to be delivered in this laboratory unit must be the outcome of the original personal work of the corresponding group (two students at most). Copying all or part of the report or the solutions from any other source will imply a zero mark in this work, apart from any disciplinary measures that could be produced.

1 Image scaling

It is common to have an image in digital forma in a different size from what is required, which can be larger or smaller than the original one. In such cases it is necessary to apply a scaling algorithm to the image that allows computing the color of the pixels of the scaled image from the pixels of the original image.

There exist many methods to carry out this process. In this lab unit we are going to focus on the scaling based on Lanczos interpolation It consists in an extension to the 2D domain of the same technique that is commonly applied to obtain more data from the sampling of a 1D signal (in one dimension).

In one dimension, the value of each point is computed as the weighted sum of the values of the original points (samples) close to it. The influence or weight that a sample has in the computation of a neighboring point x is given by the *Lanczos kernel*, which is a function defined as

$$L(x) = \begin{cases} 1 & \text{if } x = 0 \\ \dfrac{a \cdot \sin(\pi x) \cdot \sin(\pi x/a)}{\pi^2 x^2} & \text{if } x \neq 0 \quad \text{and} \quad -a \leq x \leq a \\ 0 & \text{otherwise} \end{cases}$$

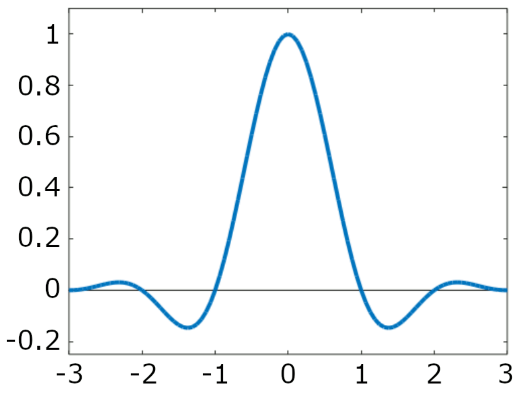


Figure 1: Lanczos kernel with window size $a = 3$

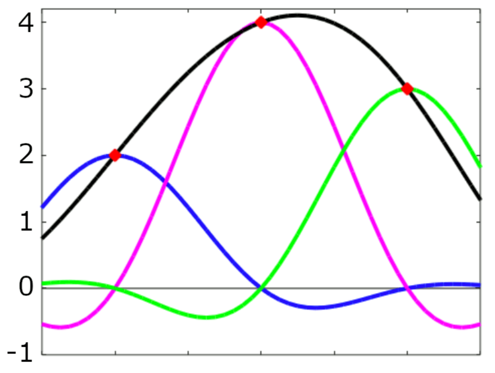


Figure 2: Example of Lanczos interpolation with 3 samples

where a is a parameter that indicates the size of the window to be used. For a value a, the function takes into account a original points on the left and right for each destination point that is computed. Usual values of a are 2 and 3.

In Figure 1 we can see the function of the Lanczos kernel, with window size $a = 3$.

Finally, the Lanczos interpolation is the result of adding the Lanczos kernels each of them multiplied by the value of the corresponding sample. As an example, see Figure 2 where, given the samples with values 2, 4 and 3, for each sample its scaled Lanczos kernel is represented, along with the result of the sum (in black). The values between samples are the points where we wanted to approximate the original function. Note that for each sample, the Lanczos kernel scaled by it has a zero value in the abscissas corresponding to the other samples, and this guarantees that when adding all the scaled kernels the interpolation coincides with the samples at the known points.

If this interpolation procedure is applied in one dimension and the in the other dimension, it can be used to interpolate values in two dimensions from a known sampling. For instance, for an image this technique can be applied in each column of the image, allowing to “stretch” the image vertically, and then repeat this for each row widening the image again.

In the case of image interpolation, this bidimensional interpolation is carried out three times, one per each plane of color.

2 Sequential code

The source code file `lan.c` is provided, corresponding to a program that implements the Lanczos interpolation method for image scaling, which must be parallelized. By default, the program scales a given image, increasing the size by a factor of 4 in both width and height.

The program receives the names of the input and output files as arguments (in this order), taking default values if the output file is not specified (or both of them).

Among the available options we have:

- `--` to avoid the generation of the output file.
- `-e` followed by the scaling factor to be used.
- `-a` followed by the value to be used as the window size a in Lanczos kernel.

For example, to enlarge an image stored in the file "`example.ppm`" by a factor 2, using a window size of 50 and saving the result in file "`output.ppm`", the command would be:

```
$ ./lan -e 2 -a 50 example.ppm output.ppm
```

If the program is invoked without arguments, the performed operation is enlarge by a factor 4 the image in file "`/labos/asignaturas/ETSINF/cpa/p2/Lenna256.ppm`", using a window size of 3 and leaving the resulting image in file "`out.ppm`":

```
$ ./lan
```

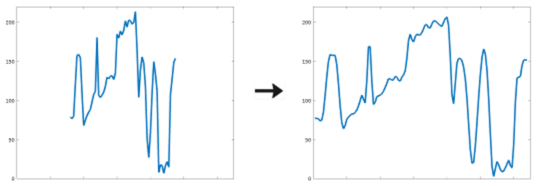
The Lanczos scaling method can be implemented by means of a program very similar to the one used in the previous lab session related to image processing. However, with the goal that the code is not as similar (and hence the parallelization is different) we have used an implementation in which the filter is applied in two sweeps, first vertically and then horizontally. This allows working with a function that applies the method to a vector (`resize1D`), which is called several times to be applied to each column and then each row of the image (`resize2D`). In turn, this will be repeated in each of the color planes of the image (`resizeRGB`). These are the three functions to which most attention must be paid, since they are the objective of the proposed parallelization. A graphical representation of how these functions work can be seen in Figure 3.

Function `resize1D` is where the Lanczos method is applied. The value of each pixel of the final image is computed by adding the different contributions of the corresponding points of the original image. A simplified pseudocode of this function can be seen in Figure 4, where `kernel` corresponds to the Lanczos kernel function $L(x)$. Note that variable `i` traverses the destination points while variable `j` traverses the points of the origin (inside the specified window).

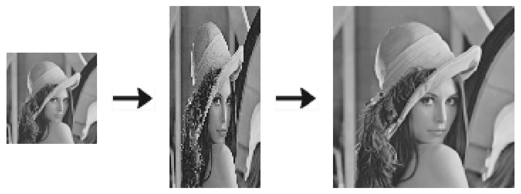
3 Work to be done

We are going to parallelize the code in different (independent) ways. In all of them you must make the program show the execution time of function `resizeRGB` and the number of threads with which the program is being run.

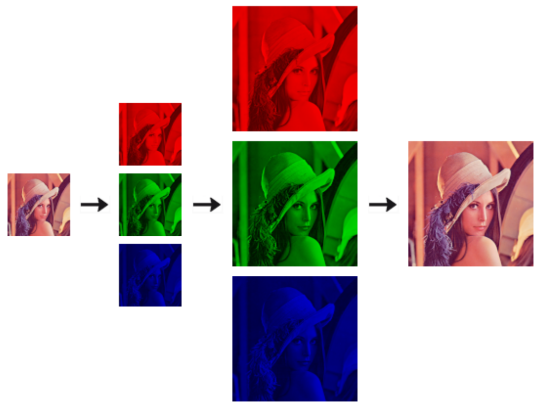
For the tests, you can scale the image "`Lenna256.ppm`" (default input file) from 256×256 to 1024×1024 (i.e., enlarge it by 4). When a parallel version is ready, you should perform several executions with more than one thread and visually check that the generated image is correct, that is, that it coincides with the one obtained with the original program. it is also convenient to use the `cmp` command to compare the output generated by the parallel program with the one generated by the original code.



(a)



(b)



(c)

Figure 3: Scaling functions: `resize1D` (a), `resize2D` (b) and `resizeRGB` (c).

```
/* Given a vector $u$ of n points, scale it to a vector
   $v$ of m points using Lanczos with window $\pm$a */
e $\leftarrow$ n / m
For i = 1, 2, ..., m
  s $\leftarrow$ 0
  j $\leftarrow$ $\lceil$ e $\cdot$ i $\rceil$
  For j = j-a, ..., j+a-1
    x $\leftarrow$ e $\cdot$ i - j
    s $\leftarrow$ s + $u_j$ $\cdot$ kernel(x,a)
  End_for
  $v_i$ $\leftarrow$ s
End_for
```

Figure 4: Pseudocode of function `resize1D`.

Exercise 1: Start by modifying the original program so that it prints the execution time of the call to the function that performs the full scaling of the image. Save this version with the name `lanSeq.c`. It will be used to be able to compare the time with the parallel programs.

Although the program is sequential, use the appropriate OpenMP function to get the time.

Exercise 2: Parallelize function `resizeRGB`. Save this version with the name `lanRGB.c`

Pay special attention to variable "`aux`". It is an array that `resize2D` uses to store temporary data. Each call to `resize2D` overwrites the whole array, without caring about its value before the call or after the call.

Exercise 3: Parallelize function `resize2D`. Save this version with the name `lan2D.c`

Pay special attention to variable "`v`".

Exercise 4: Parallelize function `resize1D`. Write two different versions: one of them parallelizing the outer loop `i` (call this version `lan1Di.c`) and another one parallelizing the inner loop `j` (`lan1Dj.c`), assuming that in both cases the loops are parallelizable.

Check that the parallel versions done in the exercises above work correctly. If you measure execution times you will notice that there is one parallel version that is very inefficient, being even slower than the sequential program. Discard this version for the next exercise.

Exercise 5: Obtain execution times, speedups and efficiencies of the different parallel versions with different number of threads. Even though it is not realistic, in order to better appreciate the benefits of parallelization, use a window of $a = 100$ for measuring times (you will have to run the program with the option `-a100`).

In order to limit the number of executions, we recommend to use powers of 2 for the values of number of threads (2, 4, 8...), reaching the number of threads that you consider appropriate (justify why you choose that maximum number of threads).

All the times included in the report must have been obtained in the `kahan` cluster.

Generate tables and plots for times, speedup and efficiency. Analyze the results and draw conclusions.

Exercise 6: In the parallelization of the outer loop of `resize1D`, would it make sense to change the default scheduling? Justify your answer. Try different schedules and check experimentally whether the algorithm behaves in the expected way.

Exercise 7 (optional): We want to study how the number of calls to function `resize1D` varies for each execution thread when using dynamic scheduling. For this, it would be interesting to know, for each of the three colors, the number of calls to this function made by the thread that performs the largest number of calls, as well as by the thread that performs the smallest number of calls.

That is, we have to count the times that the function is called in each thread, then compute and print the minimum and maximum of this value together with then identifiers of the threads that produced the minimum and maximum values.

Starting from the code `lan2D.c` (call it now `lan2Dx.c`), change the scheduling of all parallelized loops to dynamic schedule and modify the code so that it performs the requested functionality.

The output on the screen should be similar to this:

```
Red plane:   146 (thread 1) - 174 (thread 2) calls.
Green plane: 146 (thread 1) - 174 (thread 6) calls.
Blue plane:  141 (thread 1) - 178 (thread 2) calls.
```

4 Delivery

There are **two tasks** in the PoliformaT platform for the delivery of this laboratory exercise:

- In one of the tasks you should upload a file **in PDF format** with the report of the exercise. No other format will be accepted.
- In the other task, you should upload a single compressed file with all the source code of the programs you have developed. **Only the source code, do not pack the executable files** (they are large and unnecessary as they can be obtained from the source code). The file must be in `.tgz` or `.zip` compressed format.

  Please double check that all source code files compile correctly, and that they have the name indicated in this document.

When delivering the report and the source, take into account the following recommendations:

- You should produce a report that describes the employed source code and the obtained results. The report should have a reasonable size (neither a couple of pages nor several tens of pages).
- Exercises 1-6 are compulsory. Exercise 7 is optional, although it should be submitted if one wants to be eligible for the 100% of the mark.
- Do not include the whole source code of the programs in the report. You can include the parts of the code you have modified.
- Pay special attention when preparing the report. We expect that the report is well structured, with a description of the implementations. Do not merely include a listing of the results.