Kernel Image Processing

A Parallel Computing Final-term Project

Filippo Bianco

Matricola
6442543
E-mail address
filippo.bianco@stud.unifi.it

Abstract

In this project a sequential and two parallel solutions are presented for the problem of Kernel Image Processing.

A brief explanation of the problem will be presented, which is a convolution operation with a kernel, or convolution matrix, to an image. Finally the experiments conducted will be illustrated with the three solutions with different inputs.

1. Introduction

1.1. Kernel

A kernel, convolution matrix, or mask is a small matrix. It is used for blurring, sharpening, embossing, edge detection, and more. This is accomplished by doing a convolution between a kernel and an image.

In this project were used boxblur kernels of different dimensions: 3, 5 and 7. An example of boxblur kernel 3x3 is shown below:

$$\begin{bmatrix} 1/9 & 1/9 & 1/9 \\ 1/9 & 1/9 & 1/9 \\ 1/9 & 1/9 & 1/9 \end{bmatrix}$$

1.2. Convolution

Convolution is the process of adding each element of the image to its local neighbors, weighted by the kernel. This is related to a form of mathematical convolution. It should be noted that the matrix operation being performed - convolution - is not traditional matrix multiplication, despite being similarly denoted by *.

The mathematical equation for 2D convolution is represented below:

$$y[i,j] = \sum_{m=-\infty}^{\infty} \sum_{n=-\infty}^{\infty} h[m,n] \cdot x[i-m, j-n]$$

Particular attention is needed to the edge handling. In this project it was used the extend method: the nearest border pixels are conceptually extended as far as necessary to provide values for the convolution. Corner pixels are extended in 90 degrees wedges. Other edge pixels are extended in lines. This method is shown in Figure 1.

Figure 1 Extend method for edge handling with a border of 3

A simple algorithm to compute a convolution on a single image pixel is represented in Figure 2.

```
int sum = 0;
for (int i = 0; i < kernelSize; i++) {
    for (int j = 0; j < kernelSize; j++) {
        sum += img[i * kernelSize + j] *
               kernel[(kernelSize - 1 - i) *
                      kernelSize + (kernelSize - 1 - j)];
    }
}
```

Figure 2 Convolution on a single pixel

1.3. Portable Pixmap Image

A PPM file is a 24-bit color image formatted using a text format. It stores each pixel with a number from 0 to 65536, which specifies the color of the pixel. PPM files also store the image height and width, whitespace data, and the maximum color value.

The Portable Pixmap format uses an uncompressed and inefficient format compared to other bitmap image formats

but it is useful in our case to get red, green and blue channel values for every pixel.

This made the convolution easier but we had to implement a method to manage the format in Java, in fact PPM consists of:

- a "magic number" for identifying the file type. A ppm image's magic number is the two characters "P6".
- whitespace (blanks, TABs, CRs, LFs).
- a width, formatted as ASCII characters in decimal.
- whitespace.
- a height, again in ASCII decimal.
- whitespace.
- the maximum color value (*Maxval*), again in ASCII decimal. Must be less than 65536 and more than zero.
- a single whitespace character (usually a newline).
- a raster of height rows, in order from top to bottom. Each row consists of width pixels, in order from left to right. Each pixel is a triplet of red, green, and blue samples, in that order. Each sample is represented in pure binary by either 1 or 2 bytes. If the *Maxval* is less than 256, it is 1 byte. Otherwise, it is 2 bytes. The most significant byte is first.

1.4. Languages

- *Sequential solution*: `C++ 11`
- *OpenMP Parallel solution*: `C++ 11` with OpenMP framework
- *Java Parallel solution*: `Java` using native thread management

2. Main Idea

In the sequential solution, intuitively, an iteration through every image pixel is needed to perform the convolution computation with the algorithm mentioned.

While, in the parallel solutions, a good and straightforward way to parallelize the problem is to split the image into as many slices as threads we choose to use.

Every thread can work on a slice and the output image is sliced in the same way the input image is, so every thread can work on his separate portion of memory for saving results.

More details will be exposed in the following paragraphs.

3. Sequential solution

We need to iterate through every image pixel, so we need two *for* cycles (to iterate horizontally and vertically).

As mentioned in *Paragraph 1.2.*, we need to pay attention to the edges of the image: in the convolution, the value of elements which go ”outside” the image will be evaluated the same as the edge pixels, using the extend method, visually showed in Figure 1. This is achieved by computing the indexes while evaluating the conditions when the index goes below zero and above the image height and width.

4. Java parallel solution

In this solution the image is ideally sliced vertically, one vertical slice per thread with a special behavior for the last slice.

Every thread will work on his assigned slice of image, in the same way the sequential solution works.

4.1. Implementation

The code threads will run consists in a sort of sequential solution for the image slice assigned to the thread. It is just needed to take care of the indexes in the input and output image data: we must always consider an horizontal offset to reach the right element in the right slice. This is achieved by using an `id`, from `0` to `threadNum-1` assigned to each thread. Multiplying the `id` by `sliceWidth` will give the right offset.

A particular case happen when `imageWidth` isn’t a multiple of `threadNum`: to fix this, the last thread’s image slice width will be `imageWidth/threadNum + imageWidth%threadNum`, so the extra remaining columns will be assigned to the last thread.

Image slices must handle edge elements which, when computing convolution, may result outside the image or interleaving with another image slice (obviously just in the left or right side). While computing indexes for the tiling operation, we must then pay attention to use the extend method for the image edges or ”looking” at the adjacent slices elements in the other cases. When iterating through every pixel of the assigned image slice, the indexes must be updated to check every particular condition mentioned before.

5. OpenMP parallel solution

OpenMP is a set of compiler directives and library functions which take care by itself of all the duties needed to run parallel code. It works best on embarrassingly parallel problems. In this case, all threads will do the same work independently, with no need of any kind of communication: it is the perfect scenario for an OpenMP solution.

In fact, the coding logic is the same of the previous parallel version.

5.1. Implementation

The parallel section of the code starts with the specific OpenMP compiler directive which instructs the compiler where and how the parallel section begins. There is

defined the scope of the previously declared variables: the kernel, the input image and the output image; the `tid` variable used to hold the thread `id` is private, each thread has a private copy of the ”global” `tid` variable.

```
#pragma omp parallel default(none)
shared(kernel,kernelSize,inputImage,
outputImage) private(tid) {
//parallel code
}
```

Figure 3 OpenMP snippet code

There is no need of critical sections since each thread works on a separate input image slice and a separate output image slice.

To get his own `id` each thread call the OpenMP library function `omp_get_thread_num()`. Another OpenMP library function is used: the number of threads to use is passed as argument, with a call to `omp_set_num_threads(thread_num)` this parameter is set.

6. Experiments and results

Experiments were conducted running the three solutions, each execution with different inputs and parameters. It was used an image as input using PPM format (3648x3648 pixels), computed with kernel sizes of 3, 5 and 7. Tests were made also using different number of threads.

In order to obtain more trusted results the code runs 10 times and an average of the results is computed.

In Table 1 the different execution times are presented while in the Table 2 the speedup values are computed.

The code ran on an Intel i7 7th Gen Quad-core processor.

Table 1 *Execution times*

Kernel Size	3	5	7
Sequential	3.744 s	10.923 s	18.193 s
Java 2 t	1.110 s	2.394 s	5.192 s
Java 5 t	0.603 s	1.319 s	3.057 s
Java 10 t	0.481 s	1.145 s	2.021 s
OpenMP 2 t	1.689 s	5.068 s	13.363 s
OpenMP 5 t	0.964 s	2.467 s	4.311 s
OpenMP 10 t	0.813 s	2.102 s	3.799 s

Table 2 *Speedup*

Kernel Size	3	5	7
Java 2 t	3.373	4.563	3.504
Java 5 t	6.209	8.281	5.951
Java 10 t	7.784	9.540	9.002
OpenMP 2 t	2.217	2.155	1.361
OpenMP 5 t	3.884	4.428	4.220
OpenMP 10 t	4.605	5.196	4.789

7. Conclusions

From the experiments results we can see how efficient the parallel solutions are.

The use of OpenMP framework eased the management of threads and shared variables, and kept the code very clean and readable.

On the other side a Java solution outperforms OpenMP resulting in a better performance and intuitive code, thanks to the Java support for multithreading.

Furthermore, the thread management overhead seems similar as you can see from the percentage of performance growth increasing threads number.