Kernel Image Processing

Oleksandr Babenko

https://github.com/MrPatrek

In this paper, I am going to present a kernel image processing, in particular, what it is, how this works, how we can run this in sequential, parallel and distributive modes, and finally discuss the results obtained from testing it in different modes.

1 Introduction

Kernel image processing is a way to put the filters on the images and obtain a new image with affected colors on them. This is done via convolution, which means that we have some input image (matrix consisting of pixels of our image) and some input kernel matrix (filter we desire to apply for this image). What we do is simply for each pixel in our image, we overlap this pixel with the central element in our kernel matrix keeping in mind that other pixels are overlapped with corresponding kernel matrix elements. What we do next is we multiply the corresponding element of kernel matrix with the pixel that is overlapped with this kernel element, do this for every element in our kernel matrix, and simply sum all these multiplications. This sum is then written to the output image matrix to the position of the pixel that we chose before.

```
for each image row in input image:
    for each pixel in image row:

        set accumulator to zero

        for each kernel row in kernel:
            for each element in kernel row:

                if element position  corresponding* to pixel position then
                    multiply element value  corresponding* to pixel value
                    add result to accumulator
                endif

         set output image pixel to accumulator
```

Figure 1: Pseudocode for the convolution

It is also worth mentioning the change of the brightness level here. If the sum of all kernel matrix elements gives us 1, then the brightness of the image will not change despite changing the color of the pixel. Otherwise, it will decrease or increase. In my case I have some kernel matrices that change brightness level (e.g., Edge Detection filter), but we do not change anything here as it is intentionally done to obtain a desired result of image filtering.

2 Parallelization

In order to parallelize our computation, we must decide on a way of how to distribute the work among several workers. In both parallel and distributive modes, I decided to simply give each worker the specific region of the image that it will process. In our case each worker gets ***from*** and ***to*** coordinates that correspond to the starting point and finishing point of the **width of the region of the image** that it is going to process. In other words, each worker will get its own area that it will need to process, and this area will be defined by the from and to parameters of this particular area (we do not consider height here as each worker will need to **process the whole height**, so the area is defined by the whole height with starting and ending width positions).

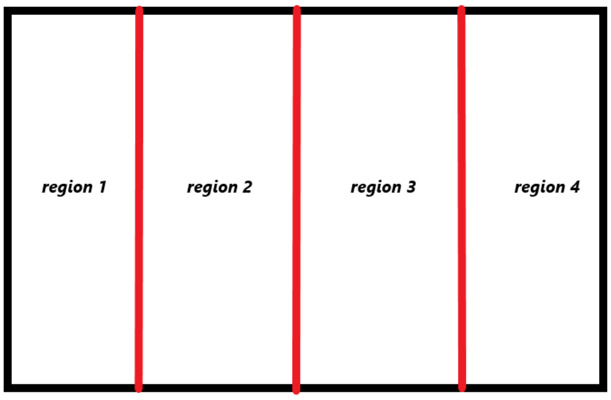


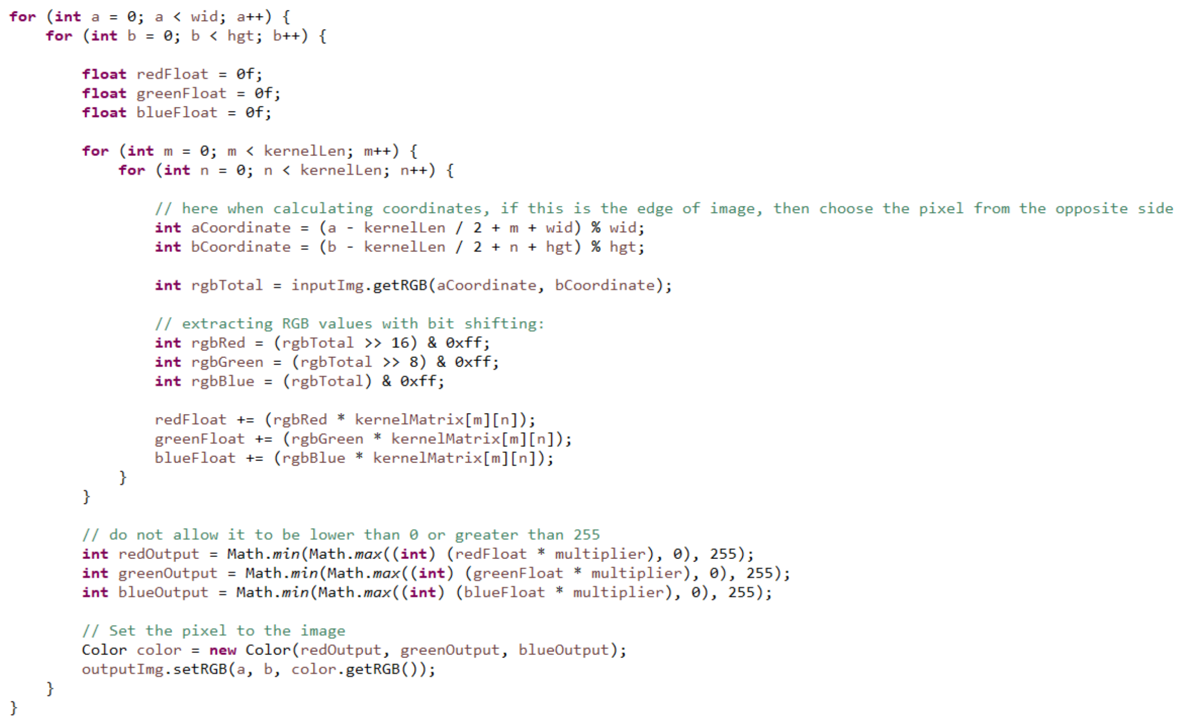
Figure 2: If we have, for example, 4 workers, then each worker gets its own region. That's how these regions look like

In both the parallel and distributive modes, I assign as equal width region values between the workers as it is possible, so that the greatest difference between some region widths is at most 1. The parallelization is quite easy to achieve here as each pixel transformation is completely independent of other pixel transformations, which means workers do not need to communicate with each other while doing their job. Due to this, no complex synchronization needed and all workers work independently of each other while computing new pixels.

3 Problem description

Describing the problem depends on the mode that it is being solved in, so that's why there will be three different descriptions for sequential, parallel and distributed modes, although they share the same base code.

- ***Sequential mode:*** This is the most primitive mode as it uses one thread of execution for everything. Here we first load our image to the BufferedImage variable (we are using Java language where BufferedImage class represents one of the ways to work with images). Then, as described above, we simply go over each pixel in our image, then for every pixel that is overlapped by kernel, we extract red, green and blue values of these colors, we multiply them by the corresponding kernel value. In the end, when all these values are summed, we create a new color by combining these three RGB colors and simply set this color to the corresponding image. When all the pixels are processed, we obtain a new filtered image.

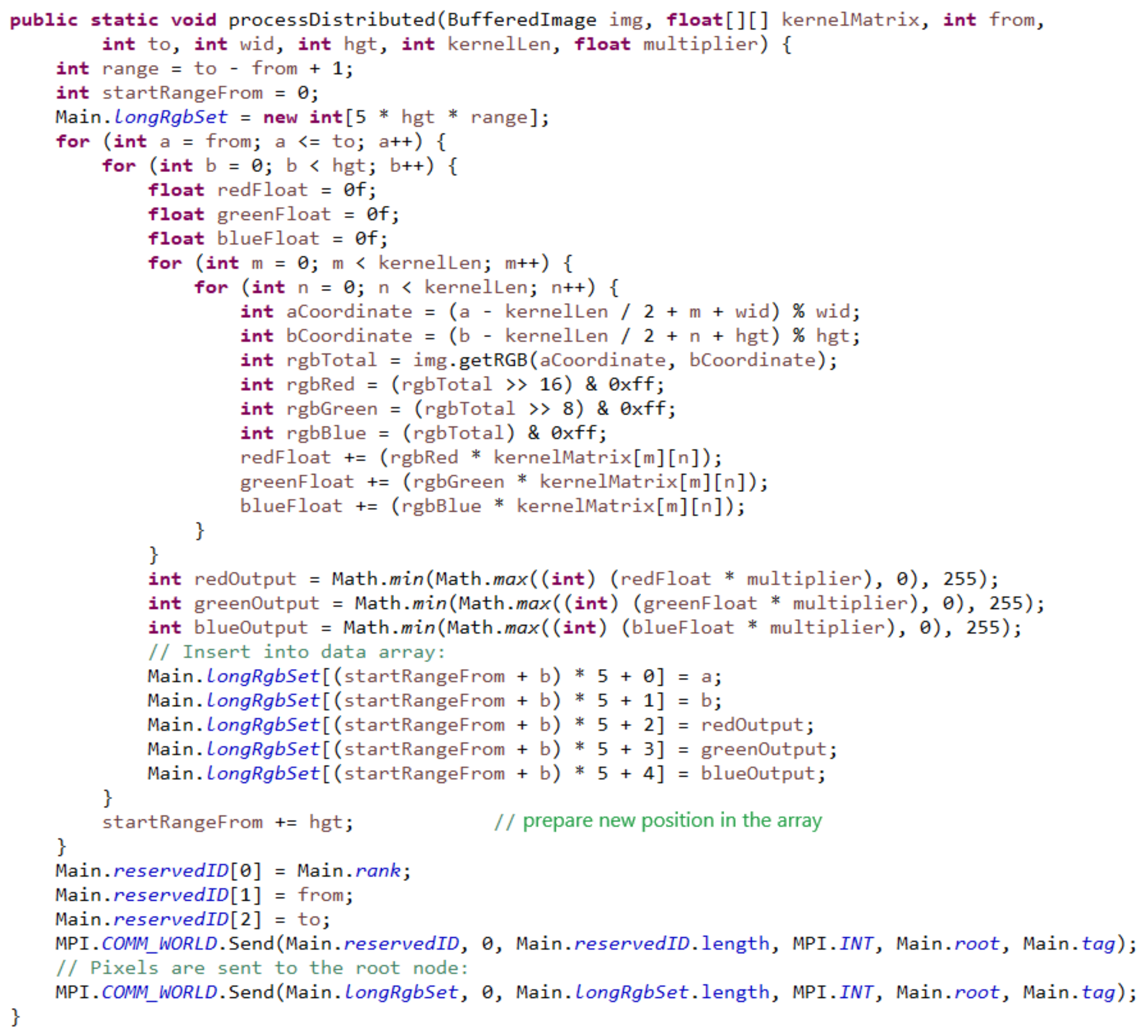


```java
for (int a = 0; a < wid; a++) {
    for (int b = 0; b < hgt; b++) {

        float redFloat = 0f;
        float greenFloat = 0f;
        float blueFloat = 0f;

        for (int m = 0; m < kernelLen; m++) {
            for (int n = 0; n < kernelLen; n++) {

                // here when calculating coordinates, if this is the edge of image, then choose the pixel from the opposite side
                int aCoordinate = (a - kernelLen / 2 + m + wid) % wid;
                int bCoordinate = (b - kernelLen / 2 + n + hgt) % hgt;

                int rgbTotal = inputImg.getRGB(aCoordinate, bCoordinate);

                // extracting RGB values with bit shifting:
                int rgbRed = (rgbTotal >> 16) & 0xff;
                int rgbGreen = (rgbTotal >> 8) & 0xff;
                int rgbBlue = (rgbTotal) & 0xff;

                redFloat += (rgbRed * kernelMatrix[m][n]);
                greenFloat += (rgbGreen * kernelMatrix[m][n]);
                blueFloat += (rgbBlue * kernelMatrix[m][n]);
            }
        }

        // do not allow it to be lower than 0 or greater than 255
        int redOutput = Math.min(Math.max((int) (redFloat * multiplier), 0), 255);
        int greenOutput = Math.min(Math.max((int) (greenFloat * multiplier), 0), 255);
        int blueOutput = Math.min(Math.max((int) (blueFloat * multiplier), 0), 255);

        // Set the pixel to the image
        Color color = new Color(redOutput, greenOutput, blueOutput);
        outputImg.setRGB(a, b, color.getRGB());
    }
}
```

Figure 3: Convolution in the sequential mode

- ***Parallel (multi-threaded) mode:*** Here the job loading part is the same as in sequential mode. After this, we should spawn some worker threads that will process the image by the regions that each of them will receive. As mentioned above, each thread gets its own range of width. The convolution is done the same way as in sequential. Each thread here does not need to wait for anybody, it is completely independent. When some thread calculates the value that it needs to set to some pixel, it simply sets it to the corresponding pixel in the output image that all the threads share between each other. After each thread set all the colors, we finish them and the output image is ready. The code is practically identical to the one in the sequential part with only addition that we add unique ranges of image to process for each thread.

- ***Distributed (MPI) mode:*** In this mode, we have root process and slave processes (we can also have just root process). The root process first sends all the data required to the slaves (due to distributed memory we need to do that), including the BufferedImage, which is converted to the bytes and then sent to the slaves. This is done because it appeared that reading an image in parallel between all the processes took much more time, so the decision was made to read the image for the root process only, which then transforms this image into array of bytes, then sends it to the slaves, who then transform this back into the BufferedImage, which is then quite faster than parallel reading of the image. As in parallel mode, we evenly distribute the regions between all the processes (even the root). I decided for the root process not just to send data and receive results back, but also to **process some area** in the meantime because most of the time it will simply wait for other processes to finish, so that it is why **root also filters image**. In the end, they all are finish at approximately the same time due to proportional distribution of the regions. Processes write the color to the internal array, which is sent to the root process when process has finished its job. Root process receives these arrays and simply sets these values to the output image.



```java
public static void processDistributed(BufferedImage img, float[][] kernelMatrix, int from,
        int to, int wid, int hgt, int kernelLen, float multiplier) {
    int range = to - from + 1;
    int startRangeFrom = 0;
    Main.longRgbSet = new int[5 * hgt * range];
    for (int a = from; a <= to; a++) {
        for (int b = 0; b < hgt; b++) {
            float redFloat = 0f;
            float greenFloat = 0f;
            float blueFloat = 0f;
            for (int m = 0; m < kernelLen; m++) {
                for (int n = 0; n < kernelLen; n++) {
                    int aCoordinate = (a - kernelLen / 2 + m + wid) % wid;
                    int bCoordinate = (b - kernelLen / 2 + n + hgt) % hgt;
                    int rgbTotal = img.getRGB(aCoordinate, bCoordinate);
                    int rgbRed = (rgbTotal >> 16) & 0xff;
                    int rgbGreen = (rgbTotal >> 8) & 0xff;
                    int rgbBlue = (rgbTotal) & 0xff;
                    redFloat += (rgbRed * kernelMatrix[m][n]);
                    greenFloat += (rgbGreen * kernelMatrix[m][n]);
                    blueFloat += (rgbBlue * kernelMatrix[m][n]);
                }
            }
            int redOutput = Math.min(Math.max((int) (redFloat * multiplier), 0), 255);
            int greenOutput = Math.min(Math.max((int) (greenFloat * multiplier), 0), 255);
            int blueOutput = Math.min(Math.max((int) (blueFloat * multiplier), 0), 255);
            // Insert into data array:
            Main.longRgbSet[(startRangeFrom + b) * 5 + 0] = a;
            Main.longRgbSet[(startRangeFrom + b) * 5 + 1] = b;
            Main.longRgbSet[(startRangeFrom + b) * 5 + 2] = redOutput;
            Main.longRgbSet[(startRangeFrom + b) * 5 + 3] = greenOutput;
            Main.longRgbSet[(startRangeFrom + b) * 5 + 4] = blueOutput;
        }
        startRangeFrom += hgt;              // prepare new position in the array
    }
    Main.reservedID[0] = Main.rank;
    Main.reservedID[1] = from;
    Main.reservedID[2] = to;
    MPI.COMM_WORLD.Send(Main.reservedID, 0, Main.reservedID.length, MPI.INT, Main.root, Main.tag);
    // Pixels are sent to the root node:
    MPI.COMM_WORLD.Send(Main.longRgbSet, 0, Main.longRgbSet.length, MPI.INT, Main.root, Main.tag);
}
```

Figure 4: Convolution process in distributed mode and the following sending of results to the root worker

4 Small technical remarks

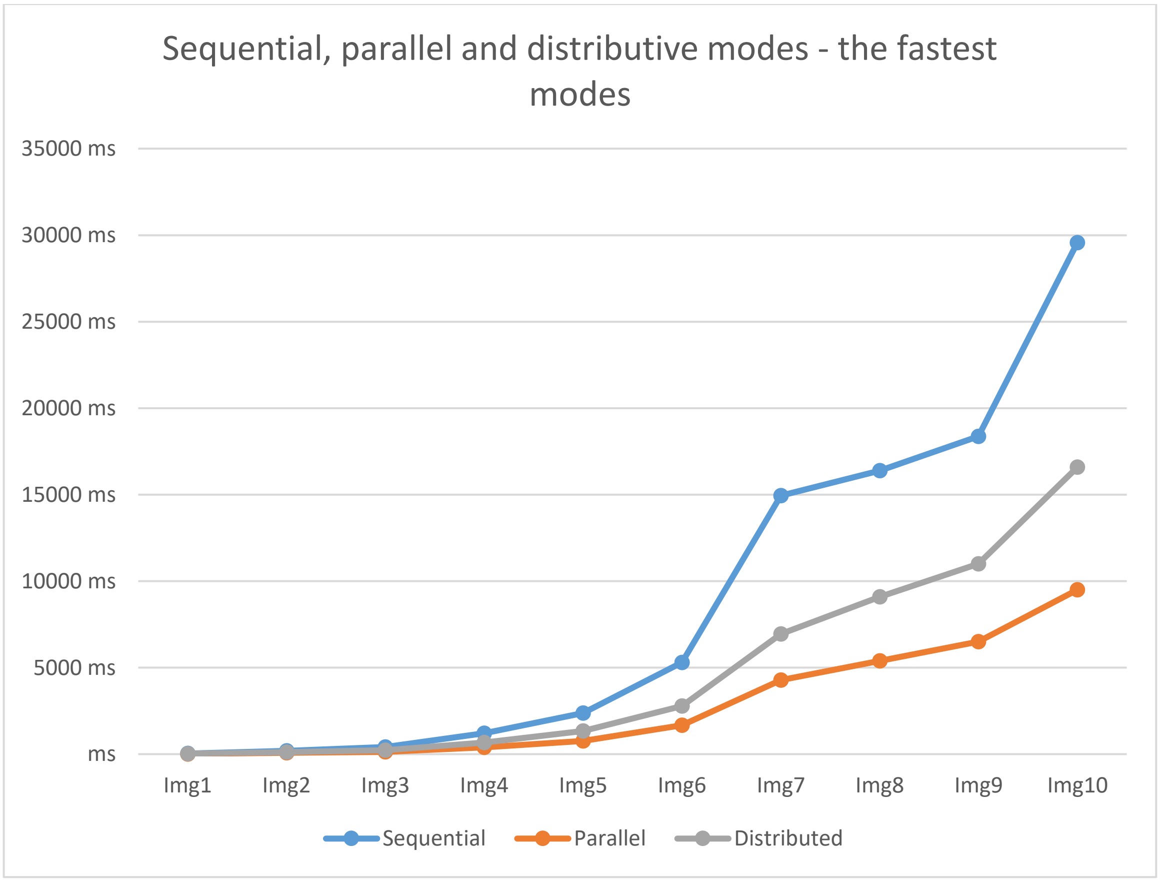
For this project, I used Java programming language in order to implement all the modes, which are sequential, parallel and distributive. In particular, for the distributed (MPI) mode I used MPJ Express, which is an open-source Java message-passing library.

5 Results

The program was evaluated on the Intel Core i7-6700HQ processor, which has 4 cores and 2 threads for each of the cores. Each particular image was tested 3 times for each configuration (for every tuple of the image, num. of workers, mode), and then the average of those measurements is taken as a result. All the results are displayed in the ***Table 1***. However, **the fastest among each mode results** are also displayed on the ***Graph 1***. The parameter that influences the speed of convolution is the overall number of pixels in image.

	Seq	Par2	Par4	Par8	Par12	Dis2	Dis4	Dis8	Dis12
Img1	34	23	14	14	21	37	39	32	38
Img2	182	117	61	77	69	139	107	115	117
Img3	406	259	153	130	146	292	234	235	259
Img4	1212	801	476	385	428	869	730	671	866
Img5	2382	1568	961	772	859	1732	1508	1341	1635
Img6	5297	3714	2282	1673	1914	4128	3156	2784	3007
Img7	14951	9234	5577	4274	4434	10561	8090	6944	7682
Img8	16387	10923	7233	5385	5889	12464	10158	9086	9713
Img9	18370	12733	8172	6506	6744	14963	12032	11000	12087
Img10	29569	19734	12662	9499	9777	20462	17822	16584	16641

*Table 1: Time taken for each image for every mode (in ms). **Green** represents the fastest results in some particular mode, **red** represents ineffective results in each mode. **Seq** means sequential mode, **Par** means parallel mode, **Dis** means Distributed mode. The **number that follows in each mode** is the number of workers of specific type (formally, threads for parallel and processes for distributed, e.g. Par4 means 4 threads for parallel mode). The image that has the higher number has the higher overall number of pixels, e.g. Img2 has more pixels than Img4*



Graph 1: The fastest modes (sequential, parallel with 8 threads and distributed with 8 workers)

6 Conclusion

As it can be seen from the results, the convolution works the best when we use all the threads of our CPU that are available. Using more threads than our CPU offers to us takes more time, although it is not that much in our example, but still, this makes no sense to use more threads than we have, which makes sense because we need additional time to switch between threads just because there are more of them than CPU has. Using less threads than our CPU has is giving us faster results than sequential, but still not that fast as when using exactly the same number of threads that our CPU has, which is also expectable since in this case we do not use all the number of threads available.

So, both parallel and distributed modes are faster than sequential, which is already a progress. We can also see that parallel mode appears to be faster up to 1,7x than distributed. This seems quite expectable, since in distributed mode we had to make some preprocessing before the image convolution, this was sending the data to slave workers, including the image itself, and then gathering the results back to the root worker and setting them to the output image, which takes some time. However, in parallel mode we did not need to spend time on sending data including image and returning information back since threads in parallel share the same variables, so this reduced a good amount of time for parallel mode.

If we take the results of the fastest mode (Parallel) with the number of threads as CPU has (in my example this is 8) and compare this to the basic sequential mode, we can achieve up to 3x faster performance using parallel mode, although on only one small image (Img1 which has 300 x 300 = 90 000 pixels) it is faster a bit less than 2x and not 3x, which makes sense because overall the number of pixels is small, and we still need some extra time for division of regions, for creating and finishing the workers in comparison with sequential mode. This means that parallelization plays an important role in boosting our program.

To sum up, the presented image processing is appropriate for image kernel processing at both small and large resolution images. All modes work fine. Parallel mode appears to be the fastest one, launching exactly the same number of threads that our CPU has.