Kernel Image Proccesing

Elmir Šut

UP FAMNIT
E-mail:*elmirsutisz@gmail.com*

25.12.2019

This report will present a framework for calculating convolution of each pixel in an image using convolution matrix(kernel). In particular, the main focus was on using edge detection convolution matrices when solving calculations sequentially, in parallel using multiple threads and with message passing between multiple single-threaded processes (MPI). Measurements were recorded and played key role in deciding our final solution.

Using simple GUI (graphical user interface) results will be shown interactively with a final image and time taken to process it using the aforementioned programs.

1 Introduction

The smallest segment of an image is a pixel, depicted as square. Every single image is two-dimensional graphics determined by width and height, measured in pixels. Each pixel has strictly determined place and it is defined by number of bits used to save it in memory.

Pixels in monochrome images consist of one value which represents light intensity or specific shade of grey, but in color images three values (Red, Green and Blue) are needed to create one pixel. Therefore color image is consisted of three same-size matrices, each representing one color among aforementioned color channels.

Convolution is a process where we add value of each pixel weighted by convolution matrix to its local neighbours where as a result we get new value for current pixel.

The general expression of a convolution is:

$$g(x,y) = \omega * f(x,y) = \sum_{s=-a}^{a} \sum_{t=-b}^{b} \omega(s,t) f(x-s, y-t)$$

where *g(x,y)* is the convoluted image, *f(x,y)* is the image to be convoluted, ω is the convolution matrix. Every element of the convolution matrix is considered by $-a \leq s \leq a - a \leq s \leq a$ and $-b \leq t \leq b - b \leq t \leq b$. [1]

[1] WIKIPEDIA: https://bit.ly/2EKU1dh

Algorithm 1 Sequential Convolution

```
1:  function CONVOLUTE(INPUT, WIDTH, HEIGHT, KERNEL, KERNELW, KERNELH)
2:      smallWidth ← width - kernelWidth + 1
3:      smallHeight ← height - kernelHeight+ 1
4:      output ← heightXwidth
5:      for i ← 0 to smallWidth do
6:
7:          for j ← 0 to smallHeight do
8:              output(i + 1, j + 1) ← SPC(input, i, j, kernel, kernelWidth, kernelHeight)
9:          end for
10:
11:     end for
12:
13:     return output
14:
15:     function SPC(INPUT, X, Y, KERNEL, KERNELW, KERNELH)
16:         accumulator ← 0
17:         for i ← 0 to kernelWi do
18:
19:             for j ← 0 to kernelH do
20:                 accumulator ← accumulator + input(x + i, y + j) ∗ kernel(i, j)
21:             end for
22:         end for
23:         return accumulator
24:
25:             ▷ Total complexity of this algorithm will be defined by smallWidth, smallHeight, kernelW and kernelH so we can represent it as: O(smallWidth ∗ smallHeight ∗ kernelW ∗ kernelH)
                                                ▷ Considering special case when image is square-shaped(width == height), and kernel is square-shaped(kernelW == kernelH), then our complexity would be O((N ∗ M)^2)
```

2 Parallelization

Parallelization of **Algorithm 1** (Sequential Convolution) can be done in many ways. Everything depends on what type of data we receive as input *(in our case image/matrix)*. In order to parallelize our algorithm, we must decide how to distribute work among selected threads. Parallel algorithm must adapt to current number of available threads/processors.

Best possible solution is when each thread can access only data slice it was passed to process. This means we dynamically slice our input image/matrix and assign each region to separate thread for convolution. In this way each thread can process only the part it received as input. Considering the fact we are not copying any data in new variables, it means all threads work on the same memory instance and save filtered data to output matrix without collecting all created data in separate variables.

Our class implementing runnable interface takes ordinal number of each thread in order to dynamically designate image slice.

Algorithm 2 Parallel Convolution

```
1:  function RUN()
2:      for i ← 0 to smallWidth do
3:
4:          for j ← (t ∗ slice) to ((t + 1) ∗ slice + 2 ∗ t) do
5:              output(i + 1, j + 1) ← (SPC(input, i, j, kernel, kernelWidth, kernelHeight))
6:          end for
7:
8:      end for
9:
10:     return output
```

In our Parallel class we are submitting Runnable class instance with thread ordinal number parameter. This parameter is crucial in determining which data slice will each thread process and where in our output matrix it will be saved. Algorithm is also using function **SPC** *(Single Pixel Convolution)* previously defined in Sequential Convolution Algorithm. Number of threads is determined by number of processors. More processors exist in certain machine more threads will be created, consequentially image matrix will be sliced in more data ranges. If there was only one processor, then program runs sequentially where one thread processes entire data.

3 Distribution

When it comes to processing images in distributed program, we must be aware that it will not be easy for processors to use shared memory because each one of them will be running its own copy of the program (*MPJ Express)*. This means when dealing with large amount of data, we risk to waste our time on copying chunks of data over and over in order to process them.

Due to fact that logically image is consisted as two-dimensional array, which in our case is not contiguous block of memory, we had to find a way to allocate new one-dimensional array which will contain same image matrix data in order to achieve property of contiguousness. Each color image consists of three separate matrices which set light intensities for Red, Green and Blue. In order to fully utilize this fact, our Distributed MPI algorithm by using MPJ Express (*message passing library*) passed each matrix in form of one-dimensional array from zero rank processor (*root*) to other processors (*non-zero*).

Algorithm 3 Distributed Convolution

```
1:  function MAIN(ARGS)
2:      MPI.Init(args)
3:      comm ← MPI.InitCommunicator(communicator)
4:
5:      numberOfProcessors ← comm.Size()
6:      currentProcessor ← comm.Rank()
7:
8:      rootProcessor ← 0
9:
10:     if (currentProcessor == rootProcessor) then
11:         ROOTACTION()
12:     else
13:         NONROOTACTION()
14:  MPI.Finalize()
```

In this algorithm we see that in our *Main method* as an argument,we will get number of processors to initialize. That means also how many times *Main method* will be executed, every time by different processor with different argument.

To exchange messages we will use combination of two functions. To send actual data we will use MPI Send function, non-blocking version(*Isend*). It will send data immediately even if communication is not finished yet. On the other hand, to receive data we will use *MPI Recive* function, blocking version, meaning that it does not return any data as long as communication is not finished(*Recv*). When sending data from non-root to root we will use blocking version *Send*. Recv and Send will play important role of barrier, because they will receive and send data, respectively, only when the buffer is filled with valid data.

Algorithm 4 Send arrays and recive results

```
1: function ROOTACTION(R, G, B, HEIGHT, WIDTH, KERNEL, KERNELW, KERNELH)
2:
3:     A1 ← convertMatrixToArray(R)
4:     A2 ← convertMatrixToArray(G)
5:     A3 ← convertMatrixToArray(B)
6:     for i ← 1 to numberOfProcessors do
7:         if (currentProcessor == 1) then
8:             comm.Isend(A1, start, size, type, i, flag)
9:         if (currentProcessor == 2) then
10:            comm.Isend(A2, start, size, type, i, flag)
11:        if (currentProcessor == 3) then
12:            comm.Isend(A3, start, size, type, i, flag)
       end for
13:    comm.Recv(temp1, start, size, type, 1, flag)
14:    comm.Recv(temp2, start, size, type, 2, flag)
15:    comm.Recv(temp3, start, size, type, 3, flag)
16:    for i ← 0 to length do
17:        arrayImage[i]=temp1[i]+temp2[i]+temp3[i]
       end for
18:
19:    return convertArrayToMatrix(arrayImage, width, height)
```

Conversion of matrix to array when sending it to *NonRootAction()* and when data collected in *rootAction()* takes huge amount of time, which I select as greatest downside of this algorithm. Creating and using temporary variables comes at a price, too. Without using and processing same shared memory(*each core gets its own copy of the program and its own data to work with, which then is sent to managerial processor*)

Algorithm 5 Recive arrays, process and send results

```
1: function NONROOTACTION()
2:     comm.Recv(arrayData, start, size, type, origin, flag)
3:     arrayData←(CONVOLUTE(arrayData, width, height, kernel, kernelW, kernelH))
4:     if (currentProcessor == 1) then
5:         comm.Send(arrayData, start, size, type, 0, flag)
6:     if (currentProcessor == 2) then
7:         comm.Send(arrayData, start, size, type, 0, flag)
8:     if (currentProcessor == 3) then
9:         comm.Send(arrayData, start, size, type, 0, flag)
```

Each non-root processor sends its data only to root processor, denoted as Rank 0 in our algorithm. They send data back using MPI Send() function, which will not return until communication is finished.

4 Implementation

All three types of executions were programmed in Java according to previously described algorithms. Access to those implementations is enabled through simple JAVA GUI (Graphical User Interface) which enables user to specify desired image from local resource repository (*as specified in report requirements paper*) or to select any photo from the system. When selection is completed, user can select type of convolution to perform. Available matrices are those able to perform edge detection. We can divide them in two groups. First group detects vertical edges and second one detects horizontal edges. In first group we have: *Vertical Filter*, *Sobel Vertical Filter* and *Scharr Vertical Filter*. In second group we have: *Horizontal Filter*, *Sobel Horizontal Filter* and *Scharr Vertical Filter*.

Having image and matrix selected, we can perform sequential, parallel or distributed implementation and obtain results immediately visible as new filtered image. After image is processed GUI will present us with time taken in milliseconds to perform certain implementation we have chosen previously.

Resource folder is filled up with 13 different-sized images. They are used as source images to perform measurements and compare afterwards. Every image is created from same source image and is of same type.

4.1 Sequential algorithm implementation

Problem Description: Using single thread we need to filter source image. Every new pixel of our result image will be calculated one after another. Each image is consisted of three matrices (Red color matrix, Green color matrix), which means that they will processed one after another.

Solution: Each matrix (*R,G,B*) will be processed independently and saved in temporary variables, which then will be used to create output matrix by traversing through each of aforementioned temporary matrices and for each pixel position values will be summed up and saved to output matrix.

4.2 Parallel algorithm implementation

Problem Description: Using as many as available threads we need to filter image. Considering the fact that in this case when we have independently(*partially*) structured data, which means we can slice our image in as many parts as exists threads.

Solution: Each matrix (*R,G,B*) will be processed independently and saved in temporary variables, but this time when having our algorithm parallelized(*as described in* ***Algorithm 2***) matrices will be sliced depending on

threads number and processed independently. Processed data will be saved immediately to shared output matrix and returned. If there is just one thread available, then it works same as sequential algorithm.

4.3 Distributed algorithm implementation

Problem Description: Taking in account that processors between themselves work independently and they do not have access to same memory (*MPJ Express shares workload by running on each processor*) only pragmatic decision was to distribute each matrix (*R, G, B*) to processors and collect them in root processor (*Rank 0*).

Solution: Each matrix (*R,G,B*) will be processed on its own processor and sent to root processor. Root processor will collect final matrices, sum them up . Matrices will be send in form of 1-D contiguous array, processed, sent back, summed up and converted back to 2-D array (*matrix*)

5 Measurements

Measurements were obtained manually, in order to get real insight of algorithm capabilities. It was done in two ways:

First group of measurements was performed on 13 different-sized images and for each image the program was run from beginning (nothing could be cached). So this group of measurements contains time needed to set up environment and data to be copied in RAM.

Second group of measurements was performed when operating system already established the program. Measurements were also performed on 13 different-sized images, but this time data was cached and in RAM to certain extent.

6 Hardware/Software

Hardware:The programs were run on processor 2,3 GHz Dual-Core Intel Core i5. 8GB of LPDDR3 RAM memory was available. Three levels of cache: L2 Cache (per Core): 256 KB, L3 Cache: 4 MB.

Software: Operating system version and type was macOS 10.15 (19A603), Java version was Java(TM) SE Runtime Environment (build 13.0.1+9).

7 Results

Results are presented in chart *Figure 1* and *Figure 2*. Charts show how much time it takes for execution types (*Sequential, Parallel, Distributed*) to process(*filter*) images according to their number of pixels.

Red lines represent execution flow of Distributed algorithm. **Blue lines** represent execution flow of Sequential algorithm and **Green lines** represent execution flow of Parallel algorithm.

Figure 1 shows results when for every execution and each image program was executed on first run (*after image was proccesed,* ***program was terminated*** *and run again for new image*)

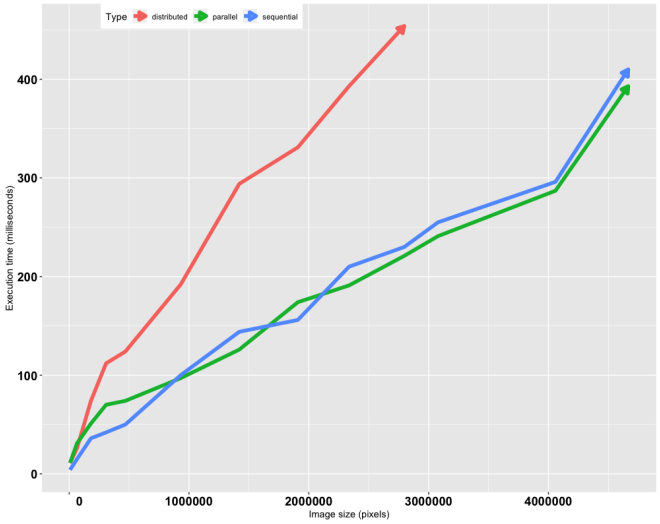

Figure 1: *Execution time compared to image size*

Figure 2 shows results when for every execution and each image program was executed after first run (*after image was proccesed,* ***program was not terminated*** *for new image*)

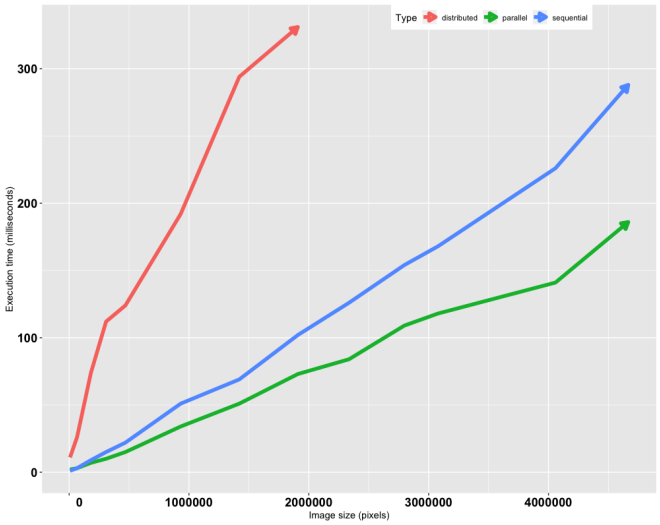

Figure 2: *Execution time compared to image size*

8 Conclusion

Drawing a conclusion from *Figure 1* and *Figure 2* we can state that Parallel implementation is far the best option in our case. It does take advantage of slicing the data and processing it independently, but does not allocate new memory. It runs much faster then sequential and distributed execution.

Referencing to *Figure 1*, we can realize that actual running up the whole program takes time and also impacts first program execution of each implementation (*Sequential, Parallel and Distributed*).

Chances are that Distributed implementation stumbles due to waste of memory and conversions of data types, which could be improved if we changed all three algorithms to work with contiguous arrays, instead matrices. This would lead to overall slower algorithms, but distributed would run faster, surely.

9 Repository

Complete project repository including this report is available on my profile on Gitlab and Github:

Gitlab: *https://gitlab.com/elmirisz/Programiranje3*
Github: *https://github.com/elmirisz/Programiranje3*