Kernel Image Processing

Zhao William
E-mail address
william.zhao@stud.unifi.it

Abstract

This paper talks about kernel image processing. The purpose of this paper is to see the speed-up of parallel version of the code that executes convolution compared with the sequential version's code. The parallel version of code was used cuda programming. different kinds of kernel matrix was used (Box Blur, Gaussian Blur and Sharpen) to process image.

1. Introduction

The kernel image processing is a convolution between the image and the kernel matrix.

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

Figure 1. Convolution Formula

where $g(x,y)$ is the filtered image, $f(x,y)$ is the original image, ω is the filter kernel. Every element of the filter kernel is considered by $-a <= dx <= a$ and $-b <= dy <= b$

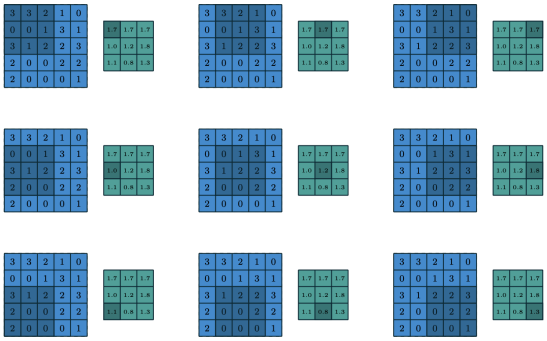


Figure 2. Convolution Operation from paperswithcode.com

The kernel matrix is a small matrix used for blurring, sharpening, embossing, edge detection, and so on so far.

Unfortunately, this will reduce the dimension of the image of width and height by kernel width + 1, kernel height + 1 respectively, one solution for this is to adding a padding with value of 0.

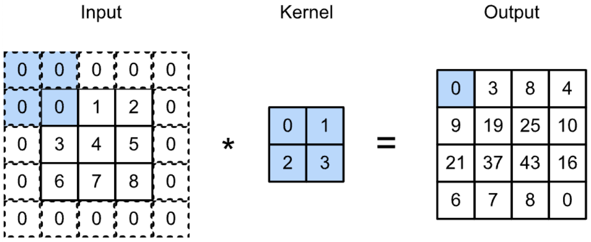


Figure 3. Convolution - Same Padding

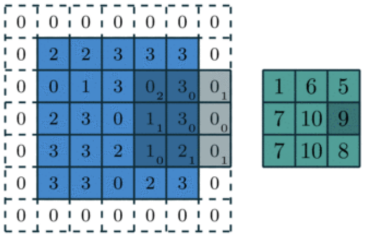
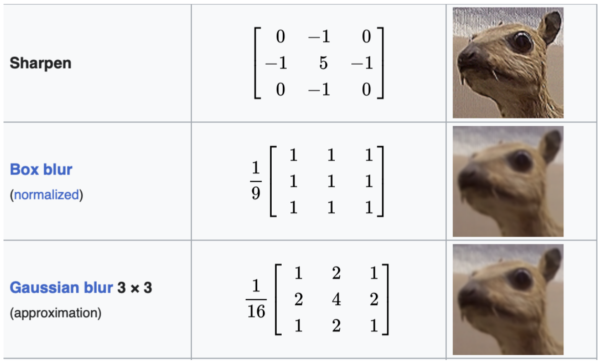


Figure 4. Convolution - Same Padding

As you can see, with this border added to the image, the filtered image can be with same height and width as the original image. [1]

2. Kernel Matrix

Three kind of kernel matrix was used, the gaussian blur, box blur and sharpen in figure 5.



Sharpen	$\begin{bmatrix} 0 & -1 & 0 \\ -1 & 5 & -1 \\ 0 & -1 & 0 \end{bmatrix}$	
Box blur (normalized)	$\frac{1}{9}\begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix}$	
Gaussian blur **3 × 3** (approximation)	$\frac{1}{16}\begin{bmatrix} 1 & 2 & 1 \\ 2 & 4 & 2 \\ 1 & 2 & 1 \end{bmatrix}$	

Figure 5. Kernel Matrixs

In figure 6 is shown a cat with no filter applied, it is 200x200 height and width.

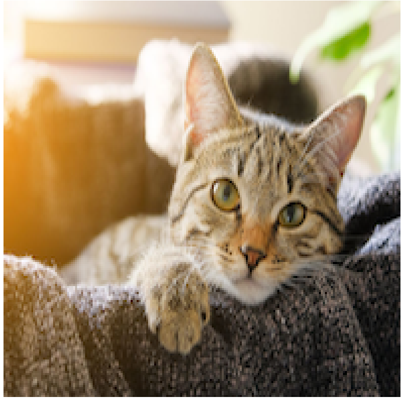


Figure 6. 200x200 Original Image

After apply the Gaussian blur kernel matrix the result is shown in figure 7

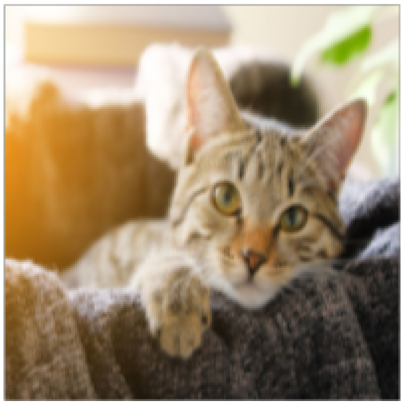


Figure 7. Gaussian Blur

Another filter is the Box blur applied and shown in figure 8

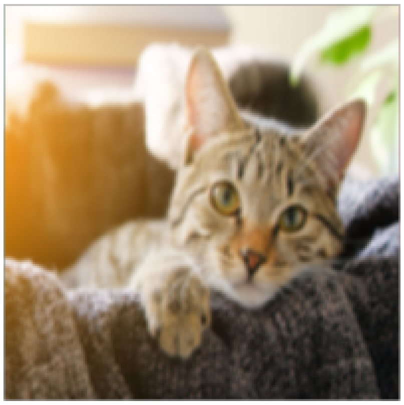


Figure 8. Box Blur

The interesting kernel matrix is sharpen which will evidence the main object of the image as shown in figure 9.

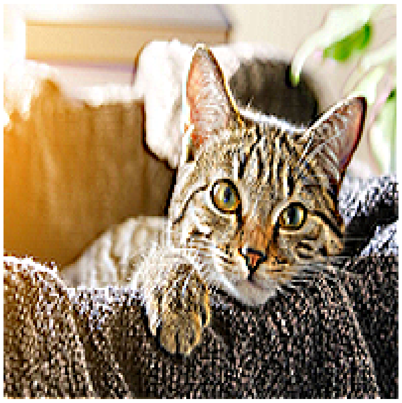


Figure 9. Sharpen

3. Setup and Environment

The environment used to create these applications is Ubuntu 22.04.1 LTS x64 bit, it is a VM borrowed from the Professor Marco Bertini. This VM has a 2 very powerful graphic cards NVIDIA RTX A2000 12GB.

CLion was used as IDE, a Cross-Platform IDE for C and C++, downloadable in URL `https://www.jetbrains.com/clion/download/#section=linux`.

The lib used for parallel programming is CUDA lib.

Figure 10. NVIDIA Cuda

Png++ lib is used for image processing (read, write) of png format image files, installable using the command sudo apt install png++, it will auto install also the fundamental libpng.

4. Sequential Version

The sequential version reads the input images in the directory images and extracts 3 channels of color bits, next it is passed as a parameter in the method of the object convolute along with the kernel matrix and the same padding boolean value.

```
1  Image KernelImageProcessing::convolute(Image& image, Matrix& filter, bool samePadding){
2
3      int height = image[0].size();
4      int width = image[0][0].size();
5      int filterHeight = filter.size();
6      int filterWidth = filter[0].size();
7
8      int output_height = height - filterHeight + 1;
9      int output_width = width - filterWidth + 1;
10
11     int padHeight = filterHeight / 2;
12     int padWidth = filterWidth / 2;
13     Image paddedImage(3, Matrix(height+2*padHeight, Array(width+2*padWidth)));
14
15     if(samePadding){
16
17
18         for (int d=0 ; d<3 ; d++) {
19             for (int i=padHeight ; i<height+padHeight ; i++) {
20                 for (int j=padWidth ; j<width+padWidth ; j++) {
21
22                     paddedImage[d][i][j] = image[d][i-padHeight][j-padWidth];
23                 }
24             }
25         }
26
27         output_height = height;
28         output_width = width;
29     }
30
31     std::cout << "Captured Height Image: " << height << std::endl;
32     std::cout << "Captured Width Image: " << width << std::endl;
33     std::cout << "Captured Height Kernel: " << filterHeight << std::endl;
34     std::cout << "Captured Width Kernel: " << filterWidth << std::endl;
35
36     Image newImage(3,Matrix(output_height, Array(output_width)) );
37
38     double tempSum = 0;
39
40     for(int m = 0; m < 3; m++){
41         for(int i = 0; i < output_height; i++){
42             for(int j = 0;j < output_width; j++){
43
44
45                 for(int k = 0; k < filterHeight; k++){
46                     for(int z = 0; z < filterWidth; z++){
47                         if(samePadding){
48                             tempSum += paddedImage[m][i+k][j+z] * filter[k][z];
49
50                         }else{
51                             tempSum += image[m][i+k][j+z] * filter[k][z];
52                         }
53
54                     }
55                 }
56                 newImage[m][i][j] = std::max((double)0., std::min((double)255., tempSum));
57                 tempSum = 0;
58             }
59         }
60     }
61
62     return newImage;
63 }
```

Listing 1. Sequential Convolution

Lines 3-6 get the width and height of the image and kernel matrix. Lines 8-9 calculate the output height and width of the filtered image. Lines 15-29 start to fill the image bits at padHeight and padWidth into the variable paddedImage and change the output height and width into the original image height and width. Lines 31-63 start to convolute in the first position of 3x3 picked on the center as you can see in the picture 2

5. Parallelized Version (Cuda)

A parallel algorithm is designed to execute multiple operations at the same step. The parallelism in an algorithm can yield improved performance on many different kinds of computers. [2] Giving a portion of the work to multiple blocks of threads, will exponentially speed up the execution runtime of the application especially when it is related to kernel image processing, it is exactly suitable for this kind of task.

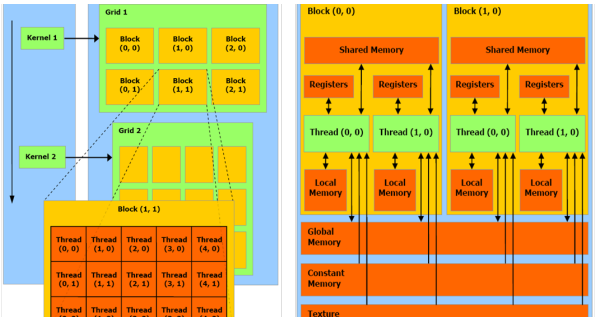


Figure 11. CUDA Architecture

```
__global__ void convolution(unsigned char*  out,
    unsigned char* in, size_t pitch,const char*
```

```
     kernel, const int factor, unsigned int width
    , unsigned int height){
2
3       int x_o = (TILE_SIZE * blockIdx.x) +
    threadIdx.x;
4       int y_o = (TILE_SIZE * blockIdx.y) +
    threadIdx.y;
5
6       //center
7       int x_i = x_o - (FILTER_SIZE/2);
8       int y_i = y_o - (FILTER_SIZE/2);
9       float sum = 0;
10
11      __shared__ unsigned char sMem[BLOCK_SIZE][
    BLOCK_SIZE];
12
13      // inside of image then copy
14      if( (x_i >=0) && (x_i < width) && (y_i >=0)
    && (y_i < height)){
15          sMem[threadIdx.y][threadIdx.x] =  in[(y_i
    *pitch)+x_i];
16      }else{
17          sMem[threadIdx.y][threadIdx.x] =  0;
18      }
19
20      __syncthreads();
21
22      if(threadIdx.x < TILE_SIZE && threadIdx.y <
    TILE_SIZE){
23          for(int r = 0; r < FILTER_SIZE; ++r){
24              for(int c = 0; c< FILTER_SIZE; ++c){
25                  sum += (float)((float)sMem[
    threadIdx.y+r][threadIdx.x+c] * ((float)
    kernel[r*FILTER_SIZE+c]/(float)factor));
26              }
27          }
28
29
30
31          if( (x_o < width) && (y_o < height)){
32              out[(y_o*width)+x_o] = (int)max(0.,
    min(255.,sum));
33          }
34      }
35
36  }
```

Listing 2. Parallel Cuda Convolution

Lines 3-4 calculate output coordinates, Lines 7-8 go to the first position of the matrix, Lines 9-20 declare the sum and a shared memory variable and assign to it the value in the ”in” matrix but checking the boundaries Lines 22-34 do the convolution and assign the value in the position yo and xo with min 0 and max 255.

6. Experiment Results

Running and timing the sequential version of kernel image processing shows this result

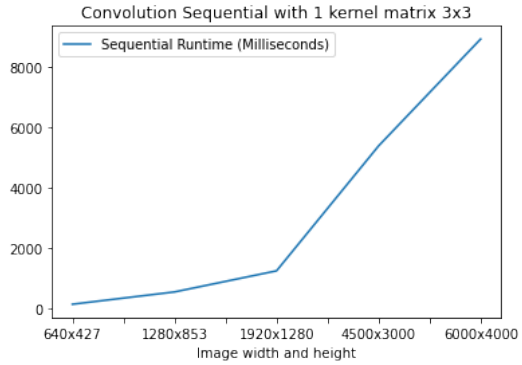


Figure 12. Convolution runtime with 1 filter sequential

As shown in this graph, initially it finishes to convolute very quickly, but after 1920x1280, it grows up linearly.

Let’s see the cuda version and the trend of the graph.

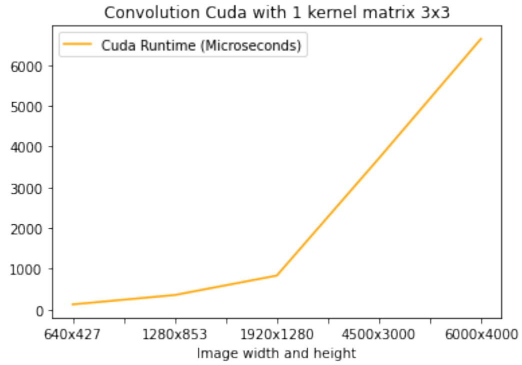


Figure 13. Convolution runtime with 1 filter CUDA version

At the start you will be surprised that the 2 graphs look very similar, but if you look closely, the runtime is in **microseconds**, it is incredibly faster than the sequential version.

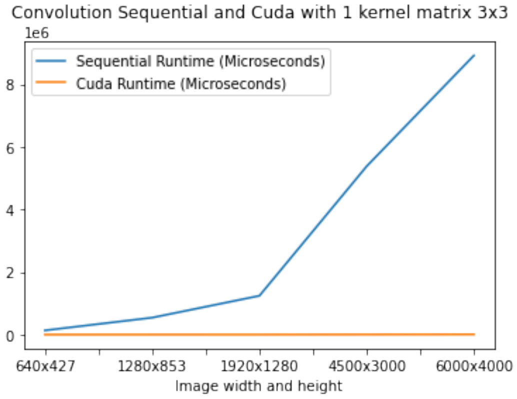


Figure 14. Convolution runtime with 1 filter CUDA and Sequential

Next, I wanted to stress the 2 programs sequential and parallel to see with 3 kernels matrix applied, the total runtime how it will be. As usual, I will start with the sequential version which shows this trend.

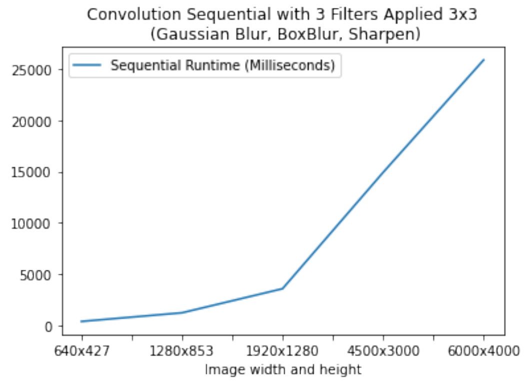


Figure 15. Convolution runtime with 3 filters Sequential

It has the same trend as the first one with only 1 kernel applied, let's see next the cuda version with 3 kernel filters applied.

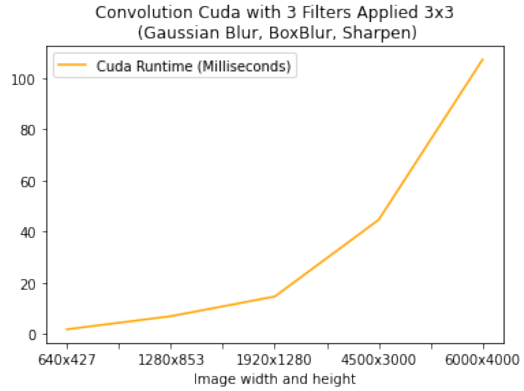


Figure 16. Convolution runtime with 3 filters Cuda

Also CUDA has the same trend as the previous one. Together with the figure 13 it looks completely the same.

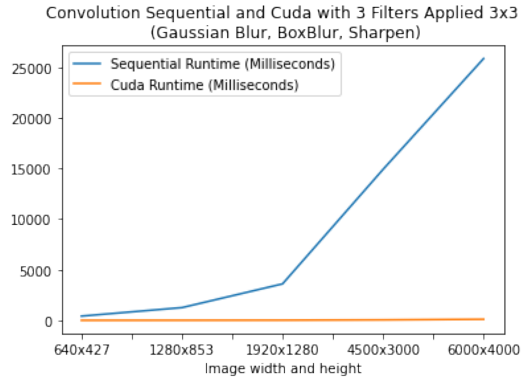


Figure 17. Conv runtime with 3 filters Cuda and Sequential

Let's see their speed-up

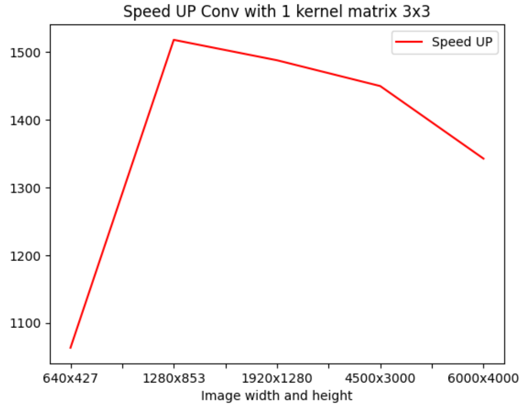


Figure 18. SpeedUP Conv with 1 kernel 3x3 dimension

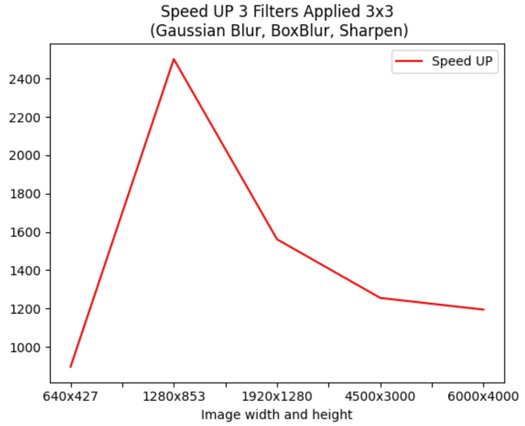


Figure 19. SpeedUP Conv with 3 kernel 3x3 dimension

As shown in 2 graphs of this type, CUDA overcomes the Sequential version in this case of kernel image processing, and CUDA version of code compared to the sequential version of code is impressive fast.

Taking into account the memory read/write access, the speed-up decrease a lots, as you can see in figure **??**

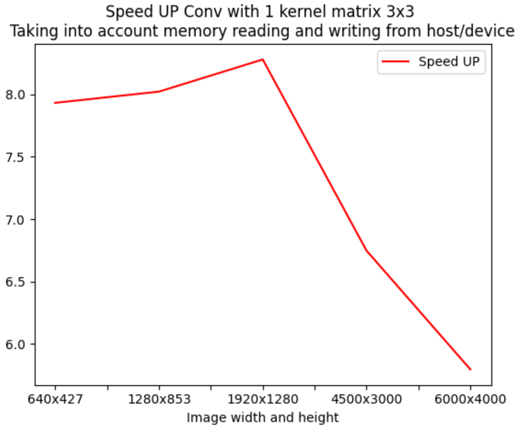


Figure 20. SpeedUP Conv with 1 kernel 3x3 dimension, and taking in account memory read/write

Adding the kernel matrix into constant memory performs better than the previous one.

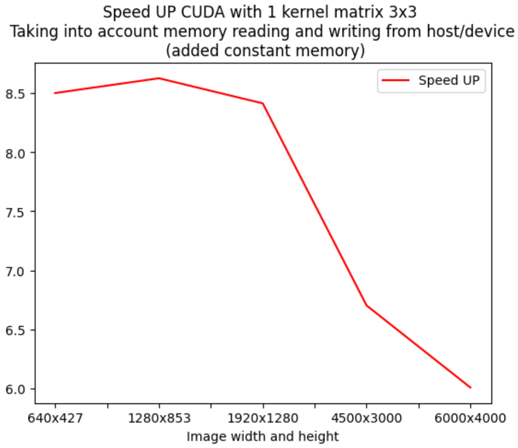


Figure 21. SpeedUP Conv with 1 kernel 3x3 dimension, and taking into account memory read/write and kernel matrix into constant memory

Another experiment is to see varying the block size, and how is it the speed-up.

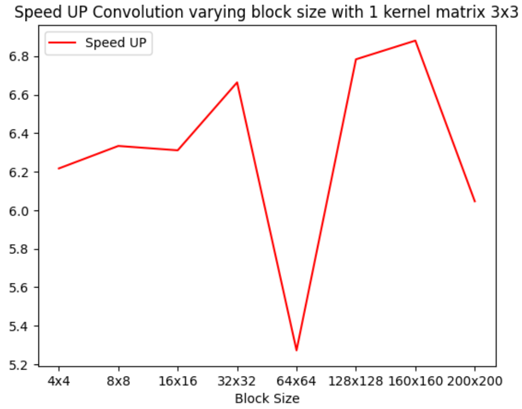


Figure 22. SpeedUP Conv with 1 kernel 3x3 dimension, varying the block size

References

[1] Kernel (image processing) — Wikipedia, the free encyclopedia. Url: `https://en.wikipedia.org/wiki/Kernel_(image_processing).`

[2] Prof. Bertini Marco. Parallel algorithm — pdf file course.