Parallel Computing : Kernel image processing

Alessio Venturi
University of Florence
alessio.venturi@stud.unifi.it

Abstract

In image processing and analysis, the Kernel Processing or Kernel Convolution is a very used technique. It is usually employed for blurring, embossing sharpening, edge detection and more. As this method is widely used and images are increasing in their dimensions, a parallelized approach is needed. In fact, the sequential version could take a long time with large images. My project is based on two main task: ”Sequential” method and ”Parallel” method through CUDA technology.

1 Introduction

The technique of kernel processing, also known as convolution[1], consists in the application of a small matrix, kernel or mask, on a second matrix, the image. In the context of image processing a convolution filter is just the scalar product of the filter weights with the input pixels within a window surrounding each of the output pixels. This scalar product is a parallel operation that is well suited to computation on highly parallel hardware such as the GPU. Kernel matrix is mostly odd-sized, since in convolution it is important to identify the center of the kernel matrix, which is easily done with odd dimensions; for example, it can be 3x3, 5x5, 7x7, and so on. Kernel matrices are hardly large.
In this project, I will use the mask 5x5 in all implementation. The convolution follows these steps:

- The mask matrix is put on the image matrix so that the center of the mask matches the pixel of the matrix that we want to elaborate
- Then the value assigned to this pixel in the output image is computed as the sum of the products of every element of the kernel (with rows and columns flipped) with the corresponding image pixel

This operation is accomplished for all the pixels of the image.

$$\left(\begin{bmatrix}1 & 2 & 3\\4 & 5 & 6\\7 & 8 & 9\end{bmatrix} * \begin{bmatrix}a & b & c\\d & e & f\\g & h & i\end{bmatrix}\right)$$

The matrices above are the kernel and an image piece respectively.For example, the element is position $[2,2]$ would be a weighted combination of all the entries of the image matrix, with weights given by the kernel, so :

$[2,2] = (i*1)+(h*2)+(g*3)+(f*4)+(e*5)+(d*6)+(c*7)+(b*8)+(a*9)$

As you can see, just only one operation requires a lot calculations. For this reason, we have to consider some techniques as parallelization to improve the performances of the algorithm.

2 Techniques Used

2.1 CUDA

CUDA is a parallel computing platform and application programming interface (API) model created by Nvidia. With CUDA, developers are able to dramatically speed up computing applications by harnessing the power of GPUs. A CUDA program is made of two parts: the host code and the device code; the host code runs on the CPU and the device code on the GPU. I will use CUDA to compare the speed up of sequential and parallel method.

2.2 Image

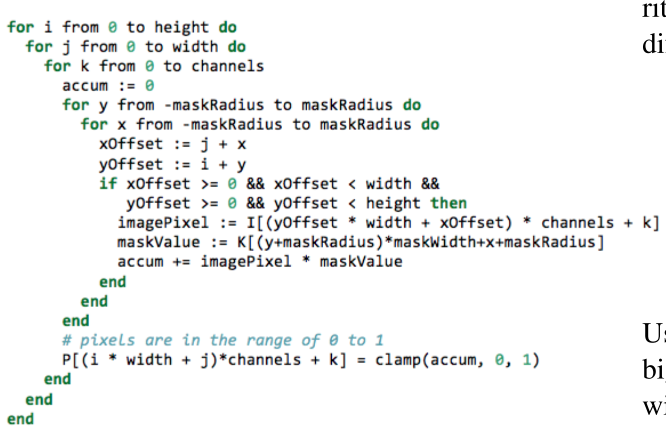
To test the speed up of the different techniques, we are going to use 4 identical images of different sizes, in order to have:

- SD: 640x426
- HD: 1280x853
- HD: 1920x1280
- 4K: 5184x3456

2.3 PPM

To read the images to use, we used the PPM format. The portable pixmap format (PPM) is an image file format designed to be easily exchanged between platforms. Each file starts with a two-byte magic number (in ASCII) that identifies the type of file it is and its encoding. The magic number is a capital P followed by a single-digit number (”P3” and ”P6” are PPM).

3 Sequential Processing



```
for i from 0 to height do
  for j from 0 to width do
    for k from 0 to channels
      accum := 0
      for y from -maskRadius to maskRadius do
        for x from -maskRadius to maskRadius do
          xOffset := j + x
          yOffset := i + y
          if xOffset >= 0 && xOffset < width &&
             yOffset >= 0 && yOffset < height then
            imagePixel := I[(yOffset * width + xOffset) * channels + k]
            maskValue := K[(y+maskRadius)*maskWidth+x+maskRadius]
            accum += imagePixel * maskValue
          end
        end
      end
      # pixels are in the range of 0 to 1
      P[(i * width + j)*channels + k] = clamp(accum, 0, 1)
    end
  end
end
```

Figure 1: Pseudo code of Sequential Algorithm

The algorithm of sequential processing consists in the application of a mask on the image. As you can see in the Figure 1 (I is Image and K is Mask), there are some steps :

- Set accumulator to zero
- Cycle on rows and columns of kernel elements
- Multiply each kernel element with the corresponding element of the image and add to the accumulator
- Assign the accumulator value to the image pixel which corresponds to the central elements of the kernel
- Repeat these step for all pixels of the image

Handling of image edges is very important in image processing. There are several ways to handle the edges of the image, in this work the cropping method is performed, because with this technique any pixels in the output image that would require values beyond the edge are skipped.

4 Parallel Processing

The convolution of the kernel with the image is calculated independently for each pixel of the image. As the kernel size grows, the complexity of the convolution computation increases. Since this task is used for each pixel of the image, the whole algorithm is very onerous. Likewise, as the image size increases, the algorithm becomes slower. Therefore, we have decided to parallelize the phases of the convolution, which are independent, in order to decrease the execution time. To parallelize the algorithm, we will use CUDA (described above). The difference between sequential and parallel is:

- **Sequential** is an algorithm that is executed sequentially – once through, from start to finish, without other processing executing
- **Parallel** is an algorithm which can do multiple operations in a given time.

Using a parallel algorithm, we’ll do things faster and bigger. The implementation of parallel processing with CUDA have different steps:

- Allocate device (GPU) memory
- Copy the host memory(Image input, Mask matrix and Image output) to device memory
- Initialize thread block and kernel grid dimension
- Invoke CUDA kernel to compute the convolution
- Copy the results from device memory to host memory

- Deallocate device memory

In the GPU, the threads (extremely lightweight unlike the CPU ones which are heavyweight) are organized in block(the concept of block is important since it represents the entity that gets executed by an SM (streaming multiprocessor)) and blocks constitute a grid.
The structure of device memory is important because there some types of memories:

- Global Memory
- Shared Memory
- Constant Memory

The first, Global memory is the biggest on the device but it's slow (it is accessible for reading and writing by all threads of the grid).
The second, Shared memory, is fast but very small (it is accessible for reading and writing by all threads of the block) and the last, Constant memory, is ideal to store data that can be accessed uniformly and only for reading. The different use of these memories leads to different implementations of the kernel convolution with CUDA. In this project we realized four implementation :

- Global Memory Convolution
- Global Memory and Kernel in Constant Memory Convolution
- Shared Memory Convolution
- Shared Memory and Kernel in Constant Memory Convolution

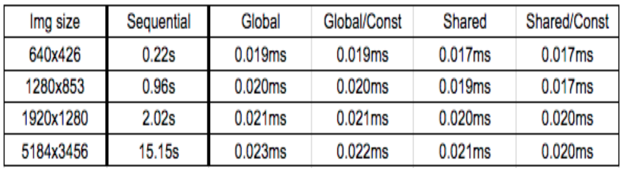
In **Constant Memory** we use **cudaMemcpyToSymbol(dest, src, size)** to copy the variable into the device memory and declare "**__constant__float deviceMaskData [maskRow*maskCol]**"(in our case). This copy function tells the device that the variable will not be modified by the kernel and can be safely cached. Instead, without Constant Memory, we will allocate the deviceMaskData on memory with **CudaMalloc(devPtr, size)**, where devPtr is a pointer to the located memory and size is the requested allocation size in bytes, and then copy bytes from src to dest specifying the direction of the copy with **CudaMemcpy(dest, src, size, kind)**. In order to speed up the input matrix reads, let's put the matrix in shared memory. Shared memory is on-chip, thus it has a much higher bandwidth and much lower latency than local or global memory.It is used similarly to CPU L1 cache, but is also programmable. Each SM has a limited amount of shared memory that is partitioned among thread blocks. Shared memory serves as a basic means for inter-thread communication. Threads within a block can cooperate by sharing data stored in shared memory. The access to the shared memory must to be synchronized using a particular function : **__syncthreads()**. A common is to partition the data into subsets called **tiles**, so that each tile fits into the shared memory. An important criterion is that the kernel computation on these tiles can be done independently of each other (not all data structure can be partitioned into tiles). So with **Tiling** we are going to transform a program to localize memory locations accessed among threads and timing of their access.
There are some step :

- Identify a tile of global memory contents that are accessed by multiple threads
- Load the tile from global memory into on-chip memory
- Use barrier synchronization to make sure that all threads are ready to start the phase
- Have the multiple threads to access their data from the on-chip memory
- Use barrier synchronization to make sure that all thread have completed the current phase
- Move on to the next tile.

5 Results

Before talking about the results, all the implementations were done through a remote server with a GeForce GTX 980 GPU and an i7 860 as CPU. The CUDA version is 11.2 and C ++ was used as support code.
As mentioned above, we have used images of different size. The kernel's size has been set to 5x5 for all implementations. With these experiments we will see not only the difference between sequential

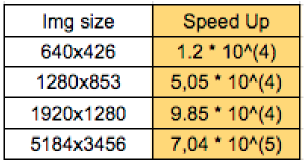
and parallel implementation, but we also compare the CUDA versions. We have also, measured the speed up ($S_p = t_s/t_p$, where t_s is the completion time of sequential algorithm and t_p is the completion time of parallel algorithm) for all CUDA versions.



Img size	Sequential	Global	Global/Const	Shared	Shared/Const
640x426	0.22s	0.019ms	0.019ms	0.017ms	0.017ms
1280x853	0.96s	0.020ms	0.020ms	0.019ms	0.017ms
1920x1280	2.02s	0.021ms	0.021ms	0.020ms	0.020ms
5184x3456	15.15s	0.023ms	0.022ms	0.021ms	0.020ms

Figure 2: Execution times of sequential and CUDA versions using a 5x5 kernel

As you can easily see in Figure 2, the parallel methods are much faster than the sequential method. The time was calculated from when only convolution starts to when it ends, for all five implementations. We observe that the versions which used kernel in constant memory (global + const and shared + const) improved by little the execution time of the versions without constant memory.



Img size	Speed Up
640x426	1.2 * 10^(4)
1280x853	5,05 * 10^(4)
1920x1280	9.85 * 10^(4)
5184x3456	7,04 * 10^(5)

Figure 3: Speed up table

In Figure 3 we see the speed up. It was calculated using the formula above, where the t_s is the sequential execution time and t_p is the average between the all parallel execution time. As you can see, parallel methods increase their usefulness as the image size increases.

The fastest version is with shared memory and kernel in constant memory, but the difference between them is really small that we barely notice it.

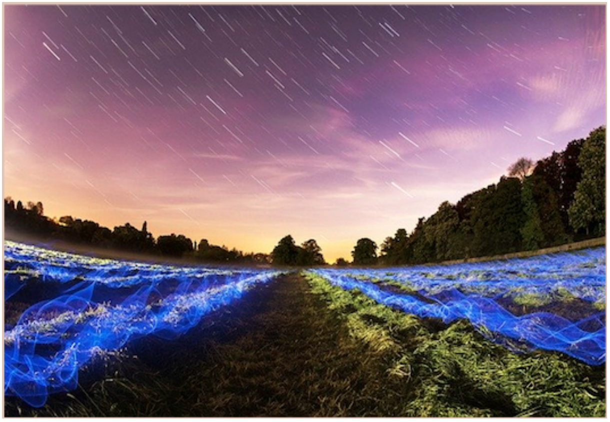


Figure 4: Photo SD without convolution

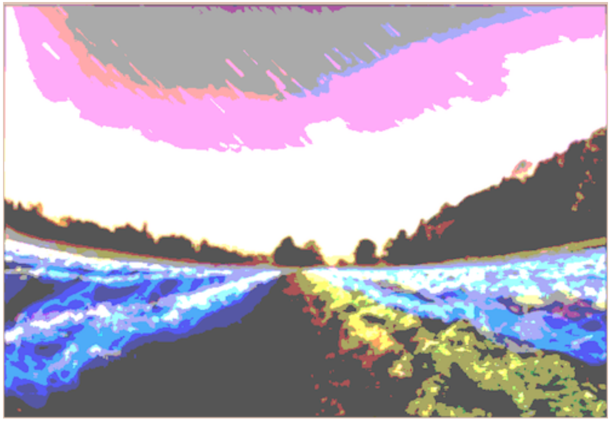


Figure 5: Photo SD with convolution

6 Conclusion

Convolution operations are an essential tool in signal and image processing and are typically responsible for a large fraction of the application's execution time. In this paper, we have proposed a sequential method and four parallel methods.

We have analyzed the execution time, noting very small differences between the various parallel methods, obtaining the best result with the "shared+constant" combination, and large differences between the sequential and parallel method. By increasing the size of the image, the effects of parallelization are noticed as there are low processing times.

6.1 Future Development

Future work may implement a different version of the algorithm using another technique for handling image edges, using different sized kernels in existing versions.

References

[1] V. Podlozhnyuk. Image convolution with CUDA.