Kernel Image Processing with C++, OpenMP and CUDA

Francesco Gradi
francesco.gradi@stud.unifi.it

Federico Màlato
federico.malato@stud.unifi.it

Abstract

In Kernel Image Processing, heavy convolutional operations are carried out between an input image and a computation mask. It is possible to parallelize these calculations to enhance a performance speed up compared to the performances of a traditional sequential program. This relationship takes into account the idea to parallelize the convolution with some technologies for the shared memory parallelism. More specifically, after the sequential program, a processor-level parallelism was implemented through the OpenMP framework; after that, thanks to the CUDA language, the calculation was tested also on a GPU. Some diversified tests were provided, with variable dimension images as input. The results showed a huge speed up in every situation: a boost of 3x was achieved with OpenMP, while with CUDA it was even greater, up to 8x.

Future Distribution Permission

1. Introduction

Kernel Image Processing (KIP) is an analysis and editing process for digital images performed whenever a task like photo editing, classification, features extraction, pattern recognition is required [1]. More specifically, with "Kernel Image Processing" one refers to the set of image filtering techniques implemented through a kernel (i.e. a convolution mask) and a convolution operation between a mask and an input image. Convolution for KIP is performed between two matrices: the first one is an arbitrarily large matrix, while the second one is required to be smaller and squared (rule of thumb: 3x3 or 5x5). During the convolution, the smaller matrix (which is usually called 'convolution mask' or 'kernel') will work as a sliding window bounded by the bigger matrix dimensions. Starting from the top left element of the bigger matrix, the mask is centered in that position and every element of the matrix is multiplied by the corresponding element of the mask (if there's one). Then, we compute the arithmetic mean of those values as the new value for the top left element of the convolved matrix. After that, the mask shifts following a certain rule (often one element to the right) and the whole procedure is repeated until the last element. For KIP, we refer to the bigger matrix as the "input image" and to the kernel as the "filter". In formula, with w the mask function and $f(x, y)$ the image, the convolution function is:

$$w \circ f(x, y) = \sum_{s=-a}^{a} \sum_{t=-b}^{b} w(s, t) f(x - s, y - t)$$

There are different class of filters (depending on the values of the elements of the convolution mask), each of which leads to a different output matrix. For example, if we consider an identity filter (that is, every element of the mask is 0, except the centered one, which is set to 1), the output image will be an exact copy of the input image. Another class of kernels is the smoothing kernels, like the gaussian kernel (the more centered the pixel with respect to the convolution mask, the bigger the weight assigned to it, resulting in a more blurred image). All those filters can be referred as the "Sobel filters" class, in which the sum of the elements of any filter of the class is 1 and implies that the final values of the pixels of the image don't need to be normalized.

There are also other class of filters, whose goal is to compute the first order derivative (in order to perform the detection of the edges of the image) or to sharpen the input image. Depending on the dimension of the input image, convolution can be a very expensive operation and, because of this, a lot of techniques have been developed in order to reduce that cost. The most effective approach consists in parallelizing the computation using a multicore CPU or a modern GPU.

In this paper, we will cover the following topics:

- dealing with KIP's computation problem, processing a .ppm image and coding this using C++;
- convolving two objects using a sequential ap-

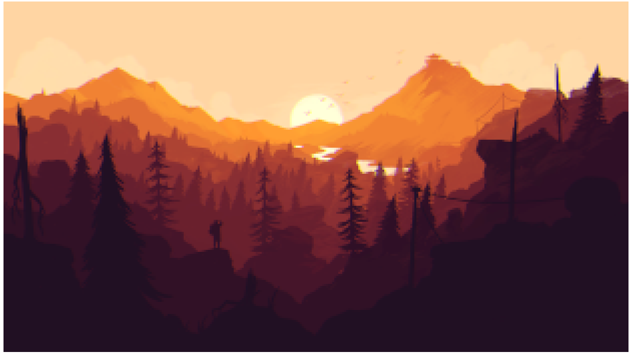

Figure 1. An original test image.

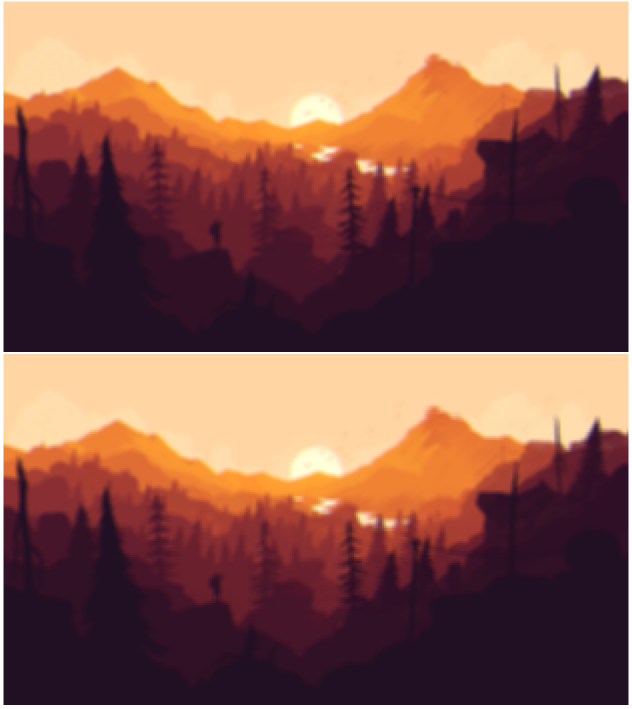

Figure 2. Filtered image with *blur* and *box blur* filters.

proach;

- parallelizing the CPU's work through the OpenMP library;
- working with a GPU as general purpose calculator in order to parallelize the convolution, using a bunch techniques for the GPU's memory management;
- evaluating and comparing the results of the experiments.

2. Methods

2.1. Sequential Approach

The whole program was written in C++, except for CUDA files. Our first purpose was to understand how to get the image's matrix from a *.ppm* file, that is, a simple pixel memorization method that uses char triples stored in a text file composed by a payload (the pixel values) and a header that contain some useful metadata, such as the dimension of the image or the number of its channels. Each line of the .ppm file is extracted and read as the RGB values of each pixel. We have decided to save those values as float instead of 8-bit unsigned int or char, so that we could avoid several casts in other methods of our program, despite of a little increase of the required memory. Furthermore, we have decided to normalize the float values in a [0, 1] range. The values are stored in a 1-dimensional vector that stands for a 3-dimensional matrix of dimension $width * height * channels$. This vector is defined as an attribute of the *Image* class.

The convolution and the mask storing methods are provided by a base class *Kernel*. Then, a derived class is defined for each type of kernel, providing a specific constructor that requires the kernel dimension as an argument. A generic kernel is instantiated on the $KernelFactory :: createKernel()$ followed by a call to the right derived class constructor. Also, a kernel per type can be instantiated with a call to $Kernel :: createAllKernels()$. The program core is the $Kernel :: applyFiltering()$ method, which takes the float matrix that represents the data of the input image as argument and stores a new image derived by the convolution between the matrix and the specific filter that calls the method. It returns a double, for performance analysis purpose.

Inside the method's scope, three for loops cycle on the *pixels* vector with respect to width, height and channels and for each pixel the matrix product between the kernel and the corresponding window of the image matrix centered in the current pixel is computed, and the result is stored in a temporary variable called *sum*. If the filter elements haven't be normalized, it is necessary to normalize the result (this is the case of the edge detection and the sharpening filters) and store it to another vector. Then, the Image class provides a method for exporting the result vector to a *.ppm* file. The convolution is the most expensive part of the whole project, and that's why it is indeed the most suitable module for some fine tuning and optimization, using parallelism.

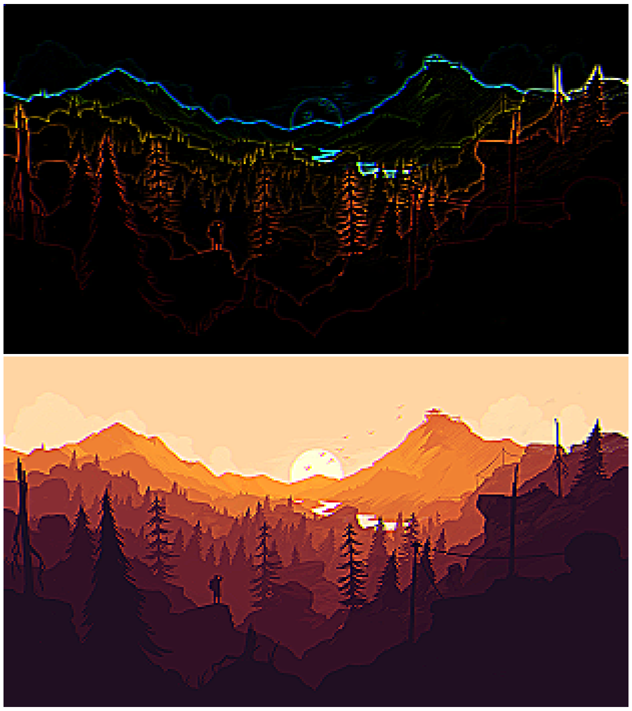

Figure 3. Filtered image with *edge* and *sharpen* filters.

2.2. OpenMP Parallel Computing

The first step in order to achieve parallelism in a relatively simple way is using the C/C++ framework for shared memory parallelism called OpenMP [3]. Its aim is to split a sequential program in multiple chunks to be executed on different threads. This is a pretty fast way to proceed, because you don't need to rewrite the whole code, but it is enough to change some specific lines, such as the for loops, with other directives already defined in the OpenMP framework. In our program we have changed the outermost for loop with a parallel for OpenMP directive, that splits the input array into chunks and executes the for loop on a different thread per chunk, of the $Kernel :: applyFiltering()$ method. We have chosen not to parallelized all of the for loops because the inner cycles were not worth parallelizing if compared to the management costs of the threads. Moreover, some of the variables were defined as private in order to reduce the management costs, which are one of the cons of this approach: whenever a thread is created, you have to deal with the cost of the threads' scheduling and the shared memory access and management. Also, a suitable number of threads must be chosen: with respect to the CPU's maximum number of threads, too few threads wouldn't take full advantage of parallelism, while too many threads would result in an unreasonable raise of the management costs. A rule of thumb is to declare a number of threads equal to or slightly superior than the sum of the physical and virtual cores of the computer.

2.3. CUDA Parallelism

The next step in our work consisted of implementing a harder parallelism paradigm, taking advantage of the modern NVidia GPUs' CUDA cores technology [4]. Thanks to this architecture we were able to achieve significant improvements with respect to the traditional sequential program. CUDA parallelism needs two "actors": the computer where the program is compiled and linked, referred as the host, and the device, where the calculation is done. The general approach to follow in these cases is to execute the C ++ code as already seen to read the *.ppm* file, generate the starting image and the filter matrices, pass this data to our device (that is, the GPU), perform the calculations on the device (calling a device method called kernel method) and return the results to the host.

The kernel method basically contains the convolution calculus. When it is called from the host, the size of the grid and the blocks containing the threads to be executed simultaneously are specified. What happens is that each thread executes the kernel code in the same way and each of those threads is characterized by an index that identifies it which is used as the index of the for loop of the sequential code.

We have used a first kernel method to write a parallel version of the $applyFiltering()$ method without the outer for loops, where the column and row indexes were identified from the thread and block executing that part of the code using their IDs, like this:

- $row = blockIdx.y * blockDim.y + threadIdx.y$;
- $col = blockIdx.x * blockDim.x + threadIdx.x$;

Apart from this new implementation, the code is identical to the previous one. Thanks to the better hardware management that comes with the CUDA paradigm, we have tried to improve this naive kernel by improving the memory management. This can be done in two different ways: using the constant memory or with the shared memory.

Constant memory is a kind of high frequency memory where you can store data that are not going to be edited in the scope of the kernel method that uses it [5]. This memory is like a cache that can be used to store some constant variables and therefore limit the number of reads from the global memory of the device (which are pretty expensive), boosting the program performances. Compared to the previous code, the only difference of this second kernel implementation is the declaration of the convolution mask variable: it is declared as *constant* outside the body of any method, (just like a define from C/C++) and it has a global scope, so that there's no need to pass it as parameter when the function is called. Then, we copied the mask definitions to the allocated constant memory using *cudaMemcpyFromSymbol*() and made the program use those definition instead of the previous ones.

On the contrary, shared memory is another kind of the NVidia GPU's memory [6]. This memory has a very low latency because it is an on-chip built on each block. Because of this, also, it has a very limited storage capacity and so its management is really important in order to take advantage of it. The management strategy for the shared memory is called tiling: it is based on splitting the variables in chunks, moving a chunk from the host's memory to the constant memory of the device, computing it and repeating for the next chunk. By the fact that the shared memory can be addressed from an arbitrarily large number of threads of the same block, we drastically reduce the number of calls to the global memory. Still, the chunks have to be copied from a slower memory and this leads to a non-avoidable slowdown. Furthermore, for this approach to be effective, the threads must always be synchronized, and this means that we have another slowdown. Anyway, a good usage of shared memory still shows a solid boost with respect to the performances of a sequential program. Basically in the code we have defined a macro, named $TILE\_WIDTH$, which has helped us defining the size of the single chunk. After that a square matrix $N[TILE\_WIDTH * TILE\_WIDTH]$ has been declared as *shared*, all we had to do was to enable copy (using *cudaMemcpy*()) of the chunks from the GPU's global memory to the shared memory, one-chunk-at-a-time, synchronize the threads, make the program calculate using N instead of the data stored in the global memory and store the result back to the global memory.

3. Results

We have tested two cases: for the first one we have created a single filter (a gaussian blur filter) for each implemented approach and we have taken the sequential program result as reference, while for the second case we have instantiated one filter per type. We have tested the program using several images of different resolutions, up to 8K (which needs about 100MB of disk space to be stored). We have run the tests on two hardwares: the first one is equipped with an AMD Ryzen 5 2600 (6c/12t) and a NVidia GTX 1060 6GB, the second one has an AMD FX-8350 Vishera (8c/8t) and an NVidia RTX 2070 8GB.

On the first test, named Single, we have simply applied a filtering using a 3x3 standard gaussian blur mask. We have tested the following routine: read the *.ppm* file and store its data as a matrix, create a filter (the type is irrelevant to the experiment, since the number of floating point operations is the same for each filter), calculate convolution and store the result in a new *.ppm* file. Table 1 shows that OpenMP has an average speed up of 2.5-3x if compared to the sequential program, while CUDA's speed up is proportional to the image resolution (this is clearly shown when using CUDA Naive, which anyway has slightly worse performances if compared to the other CUDA approaches): this probably happens because, when working with low resolution images, data transfers from host to device requires a significant fraction of the total computation time to be completed and therefore they have a higher impact on the global performance, than the impact you have when working with high resolution images, where most of the time is spent to calculate the convolution.

The second test, called Multiple, simulates a real use case, where you need to apply more than one filter to a single image, typically as a cascade, saving the intermediate results. The routine we have tested consists of the following steps: read the image from the *.ppm* file and store it as before, create one object per available filter type, calculate the convolution between the image and each of them and store the result back to a *.ppm* file. CUDA comes out with a better result than the Single test result, because in this case the image matrix is allocated only once, during the first convolution, while it is transferred back and forth for each convolution. This way, Table 1 shows that CUDA has a speed up of about 7-8x, while OpenMP achieves a slightly better result than the previous test.

4. Conclusion

Kernel Image Processing is certainly an area where the use of parallelism technologies greatly improves performance. In this project we saw how to improve the calculation of the convolution function, the burdensome part in terms of the number of operations to be performed by the program, with respect to a purely sequential reference function written in C++. Using OpenMP to implement a shared memory parallelism to exploit all the processor's virtual cores would seem to be a good idea: in this way you have a speed up of about 3x in the test configuration, more or less for all types of images taken into consideration. The main advantage in the use of this framework lies in the simplicity of its use (basically we try to parallelize the more external for loops, passing the variables in an appro-

Method	Reference (seconds)	OpenMp	Cuda Naive	Cuda Const. Mem.	Cuda Tiling
Single 0.5k	0.04	2.40x	0.45x	5.11x	3.93x
Multiple 0.5k	0.16	2.57x	7.16x	7.87x	7.85x
Single 1k	0.20	2.55x	1.35x	5.01x	4.77x
Multiple 1k	0.91	3.12x	7.23x	8.07x	8.35x
Single 2k	0.50	2.95x	2.40x	4.44x	4.37x
Multiple 2k	2.31	3.62x	7.56x	7.69x	7.67x
Single 4k	1.11	3.14x	3.50x	4.65x	4.65x
Multiple 4k	5.31	3.87x	7.90x	8.00x	7.83x
Single 8k	4.45	2.85x	4.04x	4.64x	4.63x
Multiple 8k	21.8	2.43x	7.57x	7.76x	7.54x

Table 1. All results and relative speed ups with Ryzen 5 2600 and Nvidia GTX 1060 6GB configuration.

Method	Reference (seconds)	OpenMp	Cuda Naive	Cuda Const. Mem.	Cuda Tiling
Single 0.5k	0.08	2.03x	0.36x	3.95x	4.01x
Multiple 0.5k	0.36	2.84x	5.92x	6.04x	6.30x
Single 1k	0.42	3.02x	1.55x	4.78x	4.90x
Multiple 1k	2.04	4.01x	8.50x	8.84x	8.85x
Single 2k	1.10	2.99x	2.59x	4.74x	4.75x
Multiple 2k	5.08	3.99x	8.13x	8.13x	7.93x
Single 4k	2.49	3.28x	3.30x	4.55x	4.74x
Multiple 4k	11.6	4.04x	8.13x	8.20x	8.18x
Single 8k	9.83	3.30x	4.29x	4.78x	4.78x
Multiple 8k	46.3	4.13x	8.21x	8.27x	8.28x

Table 2. All results and relative speed ups with AMD FX-8350 Vishera and Nvidia RTX 2070 8GB configuration.

priate way, to obtain an immediate speed up).

In heavier contexts, with multiple operations to be performed or with very large images be filtered, GPUs CUDA architecture can be a great advantage. Looking at the results of Table 1, you can see very well its 8x speed up compared to the sequential program, especially in the second test. This is because you save some time by reducing the number of data transfers, which is a disadvantageous part of CUDA: a proof of this lies in the fact that in the Single test with a low resolution image it is easy to see that the speed up is not that great in CUDA Naive, as we have discussed in section *Results*. The two implementations exploiting the constant memory and the tiling with the shared memory have generally achieved better results, thanks to their better usage of the different types of memory of the GPU. One of the disadvantages in the CUDA code is the need to develop a much more complex code written in a much lower-level language (that is, CUDA) than C++14. Compared to the much easier-to-write OpenMP code, CUDA isn't always advantageous: sometimes it could be preferable to achieve a smaller performance boost, avoiding tedious and hard-to-read code.

In conclusion we can affirm that KIP works much better with a parallel program, compared to a sequential program, but it is important to choose the right way to parallelize it: if the input image is a low resolution image, then an OpenMP program should be enough; although if there's the need to work with high resolution images, CUDA is probably the best option available.

References

[1] Mather, P. M. 2004. Computer Processing of Remotely Sensed Images, An Introduction. West Sussex. John Wiley and Sons Ltd.

[2] *https://www.openmp.org/.*

[3] *https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html.*

[4] *https://devtalk.nvidia.com/default/topic/910290/using-constant-memory/.*

[5] *https://devblogs.nvidia.com/using-shared-memory-cuda-cc/.*