A tutorial on the implementations of linear image filters in CPU and GPU

Alvaro Pardo
apardo@ucu.edu.uy

Facultad de Ingeniería y Tecnologías
Universidad Católica del Uruguay

Abstract. This article presents an overview of the implementation of linear image filters in CPU and GPU. The main goal is to present a self contained discussion of different implementations and their background using tools from digital signal processing. First, using signal processing tools, we discuss different algorithms and estimate their computational cost. Then, we discuss the implementation of these filters in CPU and GPU. It is very common to find in the literature that GPUs can easity reduce computational times in many algorithms (straightforward implementations). In this work we show that GPU implementations not always reduce the computational time but also not all algorithms are suited for GPUs. We beleive this is a review that can help researchers and students working in this area. Although the experimental results are not meant to show which is the best implementation (in terms of running time), the main results can be extrapolated to CPUs and GPUs of different capabilities.

1 Introduction to Linear Filtering

Image filtering is one of the most studied problems in the image processing community. Image smoothing, sharpening, feature detection and edge detection are some of the applications of image filtering. In the literature we can find two broad categories of image filters: linear and non linear. More recently, non local methods attracted the attention of researchers in the area. In fact, several of the state of the art algorithms are both non local and non linear (see [4] for more details). In this tutorial we will focus on the analysis and implementation, both in CPU and GPU, of linear filtering methods. The approach will be strongly connected to the theory of linear systems and digital signal processing. We refer the interested reader to [9] and [1] for further details on these areas. First we recall that a filter, or system, that takes an input image to produce and output one, is said to be linear if for all linear combinations of inputs produce a linear combination of outputs with the same weighting coefficients. Before analyzing linear image filter using tools from linear systems we will describe linear image filters in their most basic form using sliding windows (convolution masks).

We start with a simple linear averaging filter in which each pixel $x = (i, j)$ of the output image is computed as the average of all pixels in a 3×3 window centered at x in the input image. Processing the whole image can be expressed

with a sliding window algorithm. Given the pixel x the average filter can be implemented moving a 3×3 window with weights $1/9$ across the input image. Mathematically this can be formulated as:

$$g(i,j) = \sum_{i'=-1}^{1} \sum_{j'=-1}^{1} w(i',j') f(i+i', j+j') \tag{1}$$

where $f(.,.)$ and $g(.,.)$ are the input and output images and $w(.,.)$ is the window containing the filter weights. In the previous example of linear averaging $w(m,n) = 1/9$ for all (m,n). Changing the values of $w(m,n)$ different filters can be obtained. Inspecting equation (1) we can see that is very similar to a two dimensional convolution. Recalling the theory of linear systems we know that the output of a linear and invariant system can be obtained convolving the input f with the impulse response of the system h: $g = f * h$ [1]. The impulse response of an image filter can be obtained as the output of the filter when the input image is a discrete impulse. If $\mathcal{N}$ is a neighborhood of the same size of the sliding window centered at pixel (i,j) and $h(.,.)$ is the impulse response of the filter, equation (1) can be rewritten as a discrete convolution:

$$g(i,j) = \sum_{(m,n)\in\mathcal{N}} h(i-m, j-n) f(m,n). \tag{2}$$

The main difference between equations (1) and (2) is the range of indexes (i,j) and (m,n). Both formulations convey useful information; the first one is more suited for interpretation while the second one enables us to connect linear image filtering with convolution and the frequency response of the filter.

The equation (2) can be reformulated interchanging the role of h and f; instead of moving h across f we move f and leave h fixed. To do that we center h around the origin and extend it filling it with zeros outside the original window range and extend the input image outside the original range $[0, M-1] \times [0, N-1]$. In this way the equation (2) turns into: $g(i,j) = \sum_{(m,n)} h(m,n) f(i-m, j-n)$. In the next section we will use this formulation to obtain the frequency response of linear image filters.

1.1 Z Transform and Frequency Response

The Z transform is a very useful tool in the context of linear and invariant systems, signal processing and discrete control theory [1]. To justify the Z transform we will first deduce it starting from the convolution product. If we consider an input image $f(i,j) = z_x^i z_y^j$ with z_x and z_y arbitrary complex numbers, the output signal is: $z_x^i z_y^j \sum_m \sum_n h(m,n) z_x^{-m} z_y^{-n}$. This simple result shows that $z_x^i z_y^j$ are eigenfunctions of linear and invariant filters with corresponding eigenvalues $H(z_x, z_y) = \sum_m \sum_n h(m,n) z_x^{-m} z_y^{-n}$. This expression is known as the Z transform of $h(m,n)$ or transfer function of the filter. One of the most important properties of the Z transform states that the convolution of two signals is the

[1] The filter impulse response h is sometimes referred as filter kernel.

product of their respective Z transforms, see [1] for details. Hence, if f and g are the input and output signals related by $g = f * h$, their relationship in the Z space is: $G(z_x, z_y) = H(z_x, z_y).F(z_x, z_y)$. If we evaluate the Z transform in the unit sphere we obtain the Fourier transform. The Fourier transform of h is $H(\theta_x, \theta_y) = \sum_{(m,n)} h(m,n)exp(-j(\theta_x m + \theta_y n))$, the frequency response of the filter. In the following section we will use the Z transform to study image filters and propose alternative formulations for some of them. The interested reader can obtain more information about the Z transform in [1].

2 Implementation of linear image filters

In this section we discuss the implementation of linear image filters using the tools presented in previous sections. We will describe the implementation details and address the computational complexity of each approach. One of the goals of the following analysis is to determine which are the best implementations given the filter characteristics (window size, symmetry, etc,). First we show how to implement the filters in their traditional sequential form used for CPU algorithms. Later on, we study the parallel versions of the same algorithms suited to GPU architectures.

2.1 Convolution

The implementation of equations (1) and (2) is straightforward. Basically, the idea is to visit every pixel in the image and apply the corresponding equations. Typically, the sliding window approach, equation (1), is the first option since is very easy to understand and code. The trickiest part of the implementation is the management of the border conditions. That is, how to process pixels close to the image borders where part of the filter window falls out of the image.

Convolution computational cost To conclude the description of this method we will estimate the number of operations needed to implement it. To simplify the estimation we will assume that the image size is $N \times N$ and the window filter size is $(2W+1)\times(2W+1)$. It can be easily seen that each pixel demands $(2W+1)^2$ operations and therefore the total number of operations is of order $N^2(2W+1)^2$. To avoid confusions we distinguish computational cost from computational time.

2.2 Separable convolution

A filter is said to be separable if its kernel can be broken into two one-dimensional vectors that multiplied give the original filter response: $w(i,j) = u(i)v(j)$. The convolution of an image with a separable kernel can be implemented with two one-dimensional convolutions. First, each row in the image is convolved with v, then the result is processed across columns convolving it with u. The mathematical justification can be easily obtained substituting $w(i,j) = u(i)v(j)$ into (2): $g(i,j) = \sum_m u(m) \left(\sum_n v(n) f(i-m, j-n)\right)$.

Separable convolution computational cost Convolution with separable kernels requires $N(N(2W+1)) + N(N(2W+1)) = 2N^2(2W+1)$ operations. The reduction in the number of operations is $(2W+1)/2$ compared to the traditional two-dimensional convolution. Therefore, the larger the kernel the better speed-up can be obtained with this approach.

2.3 Special case: Box Filtering

The box filter is an average filter with uniform weights; the output at pixel $x = (i, j)$ is the average of all pixels in the filtering window:

$$g(i,j) = \sum_{i'=-W}^{W} \sum_{j'=-W}^{W} \frac{1}{(2W+1)^2} f(i+i', j+j')$$

The beauty of this filter is that it can be implemented using integral images to reduce the number of operations. The integral image of an image f at pixel (i, j), denoted as $Sf(i, j)$, is the sum of all elements in the rectangular region with upper-left and lower-right vertices $(0, 0)$ and (i, j): $Sf(i,j) = \sum_{i' \leq i, j' \leq j} f(i,j)$ Given the pixel (i, j), the output of the box filter can be obtained using the integral image as follows: sum the pixels in the square region defined by the points $(i - W, j - W)$ and $(i + W, j + W)$ and divide it by the number of pixels in the window (recall that the filter window is $[-W, W] \times [-W, W]$. This can be easily implemented with integral images [3]:

$$g(i,j) = \frac{Sf(i+W, j+W) - Sf(i+W, j-W) - Sf(i-W, j+W) + Sf(i-W, j-W)}{(2W+1)^2}. \tag{3}$$

The computation of the box filter implies two steps: the computation of the integral image and, after that, the computation of the output using equation (3). This formulation is especially useful when we need to filter the image at different scales, i.e. with different filter sizes, because the first step can be reutilized and regardless the filter size, the second step has always the same cost in terms of operations. In order to estimate the cost, in terms of operations, we have to estimate the cost of computing the integral image and the actual filter, from equation (3) It can be easily seen that the computation of the integral image requires N^2 operations. On the other hand, the filtering requires only four operations per pixel so the total number of operations to apply the filter is $4N^2$. Therefore, the total number of operations for the case of the box filter using integral images is $5N^2$. As said before, this does not depend on the filter size. This is why integral images are very attractive to filter the same image at different scales (the work of Viola and Jones popularized this idea [10]).

Moving Average Filter Moving Average Filters are in fact an implementation of the Box Filter. Using the two-dimensional Z transform it can be shown that the relationship between the Z transforms of input and output images is:

$$G(z_x, z_y) = \frac{F(z_x, z_y)}{(2W+1)^2} \sum_{i=-W}^{W} \sum_{j=-W}^{W} z_x^i z_y^j = \frac{F(z_x, z_y)}{(2W+1)^2} \frac{z_x^{W+1} - z_x^{-W}}{1 - z_x} \frac{z_y^{W+1} - z_y^{-W}}{1 - z_y}.$$

Taking the inverse Z transform the previous equation gives:

$$g(i+1,j+1) = g(i+1,j) + g(i,j+1) - g(i,j) + \frac{(f(i-W,j-W) - f(i+W+1,j-W) - f(i-W,j+W+1) + f(i+W+1,j+W+1))}{(2W+1)^2}.$$

The computation cost to process an $N \times N$ image in this case is $7N^2$. Comparing to the separable alternative there is no big difference in this case in terms of number of operations.

3 Introduction to GPU programming using CUDA

In this section we review the main concepts behind GPUs and the parallel implementation of algorithms using this technology. In particular we will use CUDA. For a more detailed presentation we refer to [8].

GPUs are highly parallel processors with many cores and the ability to run multiple threads that provide high performance computing. The architecture of the GPUs, traditionally optimized for graphic applications, has some limitations; less cache and flow control limitations. GPUs provide advantages in applications where the same computations can be applied in parallel to may data elements. However, memory transfers from main memory to device memory (GPU) have to be considered. A GPU implementation pays off if its computation cost it s higher that memory access cost. To process data with an algorithm implemented in the GPU the data must be transferred from main memory to the device, process it in the device and transfer back to main memory. Therefore, the computation cost must be higher enough to pay the overhead introduced by memory transfers. CUDA (Compute Unified Device Architecture) is a programming language by nVidia that allows programming the GPUs abstracting the code from the actual hardware details (OpenCL is another option). Provides the user a high level interface so that he can take advantage of the capabilities of GPUs without having to directly handle the hardware. The CUDA programming model allows the user to use GPU capabilities from a simple interface similar to C language (C language extension). CUDA proposes three abstractions: a hierarchy of thread groups, shared memory and synchronization [6]. These abstractions provide an easy way to understand and handle parallelization. These abstractions are designed so the actual implementation does not need to know the details of the hardware (number fo cores, etc.) (see Figure 1). The idea is to divide the problem in blocks of threads. Then each block of threads works cooperatively to solve the problem. In this way scalability is easily achieved, see Figure 1. To understand image filtering implementations on GPUs, and to make this article self-contained, we first review the basics using a simple examples of vectors and matrices addition (This section is based on [6]).

CUDA Kernels A kernel is a function that runs N times in parallel on N different threads. In the following code a kernel is used to sum in parallel to vectors of dimension N using N threads.

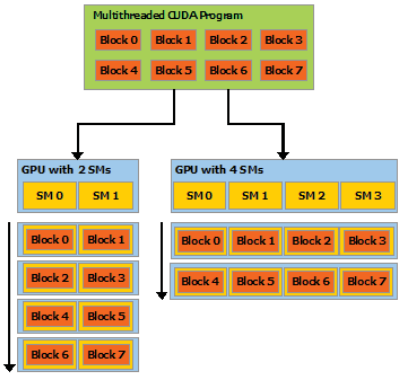

Fig. 1. From [6]: *A multithreaded program is partitioned into blocks of threads that execute independently from each other, so that a GPU with more multiprocessors will automatically execute the program in less time than a GPU with fewer multiprocessors.*

```
// Kernel definition
__global__ void VecAdd(float* A, float* B, float* C){
  int i = threadIdx.x;
  C[i] = A[i] + B[i];
}
// Kernel invocation with N threads
VecAdd<<<1, N>>>(A, B, C);
```

CUDA threads are three dimensional vectors which enable processing blocks up to dimensions three. The following example shows how to add two matrices.

```
// Kernel definition
__global__ void MatAdd(float A[N][N],
                       float B[N][N], float C[N][N]){
  int i = threadIdx.x;
  int j = threadIdx.y;
  C[i][j] = A[i][j] + B[i][j];
}
// Kernel with one block of NxNx1 threads
int numBlocks = 1;
dim3 threadsPerBlock(N, N);
MatAdd<<<numBlocks, threadsPerBlock>>>(A, B, C);
```

Since the number of threads is bounded (in actual GPUs by 1024) and all threads of a block reside in the same core, when dealing with large vectors or matrices, the problem must be organized into several blocks. CUDA blocks can be organized into one, two or three dimensional grids. In this way, the problem can be organized into a number of blocks per grid and threads per block. This allows for flexibility to organize the computations. The following code shows how to add two matrices organizing the computation into blocks of size 16×16. The matrices are divided with a tiling of 16×16. Since there is no guarantee that

N is multiple of 16, inside the kernel we must verify that the pixel (i, j) resided inside the matrices. The choice of blocks of size 16×16 can be modified to take advantage of the GPU capabilities.

```
// Kernel definition
__global__ void MatAdd(float A[N][N],
                       float B[N][N], float C[N][N]){
  int i = blockIdx.x * blockDim.x + threadIdx.x;
  int j = blockIdx.y * blockDim.y + threadIdx.y;
  if (i < N && j < N)
    C[i][j] = A[i][j] + B[i][j];
}
// Kernel invocation
dim3 threadsPerBlock(16, 16);
dim3 numBlocks(N / threadsPerBlock.x, N / threadsPerBlock.y);
MatAdd<<<numBlocks, threadsPerBlock>>>(A, B, C);
```

4 Image Filtering in GPUs

This section discussed the GPU implementation using CUDA of the image filters presented above. We will use the basic notions of GPU programming with CUDA introduced in previous section.

4.1 Convolution

The code below is a direct implementation of 3×3 linear image filter. The code is very similar to the one in C used for the CPU. The main difference is that in this case the pixel indices i and j are obtained from the grid and block organization of the computation on the device. This implementation follows the same philosophy of the code seen before to add two matrices. The next snippet of code shows how to organize the memory allocation and kernel invocation.

```
// Kernel definition
__global__ void filter(float *f, float* g, int rows, int cols){
  int i = blockIdx.x * blockDim.x + threadIdx.x;
  int j = blockIdx.y * blockDim.y + threadIdx.y;
  ...
  if ( i>=w && j>=w && i<cols-w && j<cols-w ){
    float h[3][3] = {...};
    float sum = 0;
    for (int ii=-w; ii<=w; ii++){
      for (int jj=-w; jj<=w; jj++){
        float fij = f[ (i+ii)*rows + (j+jj) ];
        sum += h[ii+w][jj+w] * fij;
      }
    }
    // store output. sumh is the sum of all weights h[][]
    g[i*cols + j] = sum/sumh;
        ...
```

```
// HOST memory
float *f = new float[arraySize];
float *g = new float[arraySize];
// DEVICE memory (GPU)
cudaMalloc((void**)&dev_f, size * sizeof(float));
cudaMalloc((void**)&dev_g, size * sizeof(float));
// Copy image f from DEVICE to HOST
cudaMemcpy2D(dev_f, pitch, f, size*sizeof(float)*fils,
             cudaMemcpyHostToDevice);
// Kernel invocation
dim3 block(16, 16);
dim3 grid(fils/16, cols/16);
filter<<<grid,block>>>(dev_f, dev_g, fils, cols);
// Copy result from DEVICE to HOST
cudaMemcpy(g, dev_g, size*sizeof(float), cudaMemcpyDeviceToHost);
```

The first two columns of Table 1 show the results of executing the CPU and GPU version of the sliding window (convolution) method. For small images, the overhead time of memory transfers is higher than the computation cost and therefore the GPU implementation does not give any speed up. For larger images the GPU is an alternative to speed up linear image filtering (the speed ups factors are shown in parenthesis). Although the breakpoint of when GPU outperforms CPU can depend on the hardware (CPU & GPU), the main result holds valid; for small images and filters of low computational demands (small windows) GPUs are not faster than CPUs due to the memory transfer overheads.

CUDA Texture Memory Texture memory is a read-only memory that can be used to improve performance. Optimizing memory access in the GPU provides benefits in terms of computational time [8]. Texture memory is one of the most basic improvements that can be added to the code of the image filter. The only modification in the kernel code is the access to pixel data f[i][j] using float fij = tex2D(texf,i+ii,j+jj) where texf is a texture connected to array f. In Table 1 we can see that the use of texture memory reduces the running time. Once again, we observe that the differences appear for large images.

4.2 Separable Convolution

The GPU implementation of a separable filter needs two kernels; one to filter by rows and the other by columns. Separable convolution can provide speedups around 3 times [2]. According to [7] the use of texture memory and other memory optimizations an additional speedup of factor 2 can be obtained (see [7] for details).

[2] In `https://blog.kevinlin.info/nvidia-cuda-gpu-computing-and-computer-vision/` there is a detailed analysis of the separable implementation

5 Results and Discussion

CPU versus GPU The first result is that when dealing with small images GPU does not provide advantages over CPU (see columns 1 and 2 from Table 1. The actual size of the image where one implementation outperforms the other depends on hardware features. However, the observation holds valid and, as we said before, is due to memory transfers from host to device and backwards. Furthermore, on the CPU side there is still room for improvements using, for example, parallelization with multicores. Therefore, the GPU implementation paysoff for large images on when additional operations will be performed in the GPU with the same data. That is, when other processes will be applied to the same image. In this case, the data transfer cost is shared among several process and makes GPU more attractive. Since in many areas we are seeing an increasing use of high definition images (HDTV, Ultra HDTV), we can expect to have to process large images and therefore GPUs are obviously a good alternative. This is the case of mobile platforms which include a GPU to handle image and video data.

Algorithms Now we discuss the impact of the algorithms that reduce the computation cost. First we reviewed separable convolution which is a case of interest since many traditional image filters are separable (Gaussian filters, Sobel filters for edge detection, etc.). From the data in Table 1 we can observe and speedup of $\times 1.5$ for a filter of size 3×3. This factor agrees with the estimation in Section 2.2. In this case $W = 1$ so the theoretical computation cost reduction is $3/2$. As we mentioned in Section 4.2 GPU implementation of separable convolution gives an additional speed up (see [7]). To illustrate the benefits of applying the correct algorithms to decrease the computational cost and improve running times, we discussed Box Filters in Section 2.3. Box filters are a special case of linear image filters with many real applications due to their reduced computational cost [9]. The last column of Table 1 shows the obtained running times for a CPU implementation. We must be careful when directly comparing this implementation with the others since this is a special filter (with uniform weights). If we assume that all algorithms implement the same box filter, using a uniform kernel, we can see that the implementation of the box filter (MAF) outperforms all other algorithms. Hence, if the application allows a box image filter then the MAF is a simple and computational efficient algorithm (there is no need for a GPU implementation). Finally, if we need a multiscale version of the box filter, the use of integral images is a good solution. In [5,2] the authors compare GPU and CPU implementation of integral images.

6 Conclusions

In this paper we presented an overview of linear image filtering, its basic results based on the theory of linear and invariant systems, and different algoritms to implement the filter. We reviewed different algorithms to reduce the computational cost and discussed their CPU and GPU implementations. We discussed

pros and cons of algorithms and their implementations. Based on the results presented in in Table 1 we can see that image size must be considered to select the most suited implementation. This paper was intended to understand the basics behind linear image filtering using CPU and GPUs. In real applications, libraries such as NPP (nVidia Performance Primitives) or ArrayFire to name two, must be considered.

N	CPU (1)	GPU (2)	GPU texture (3)	CPU sep. (4)	MAF (5)
512	10	65 (x0.15)	65 (x0.15)	7 (x1.42)	1
1024	42	76 (x0.55)	75 (x0.55)	25 (x1.68)	4
2048	165	105 (x1.58)	91 (x1.83)	105 (x1.58)	17
4096	660	202 (x3.27)	162 (x4.07)	440 (x1.50)	66

Table 1. Running times in mseg for a 3×3 filter. GeForce GT 430 (96 cores, 1400 Mhz). Intel i7-2600 3.4 GHz, 16 GB RAM, Windows 7 64 bits. (1) Standard CPU. (2) Direct GPU. (3) Direct GPU using texture memory. (4) CPU separable convolution. (5) CPU implementation of MAF.

References

1. V Oppenheim Alan, W Schafer Ronald, and RB John. *Discrete-time signal processing*. 1989.
2. Berkin Bilgic, Berthold KP Horn, and Ichiro Masaki. Efficient integral image computation on the gpu. In *Intelligent Vehicles Symposium (IV), 2010 IEEE*, pages 528–533. IEEE, 2010.
3. Scott Krig. *Computer Vision Metrics: Textbook Edition*. Springer, 2016.
4. Peyman Milanfar. A tour of modern image filtering: New insights and methods, both practical and theoretical. *IEEE Signal Processing Magazine*, 30(1):106–128, 2013.
5. Diego Nehab, André Maximo, Rodolfo S Lima, and Hugues Hoppe. Gpu-efficient recursive filtering and summed-area tables. *ACM Transactions on Graphics (TOG)*, 30(6):176, 2011.
6. nVidia. Cuda c programming guide, 2017.
7. Victor Podlozhnyuk. Image convolution with cuda. *NVIDIA Corporation white paper, June*, 2097(3), 2007.
8. Jason Sanders and Edward Kandrot. *CUDA by Example: An Introduction to General-Purpose GPU Programming, Portable Documents*. Addison-Wesley Professional, 2010.
9. Steven W Smith et al. *The scientist and engineer's guide to digital signal processing*. California Technical Pub. San Diego, 1997.
10. Paul Viola and Michael Jones. Rapid object detection using a boosted cascade of simple features. In *Computer Vision and Pattern Recognition, 2001. CVPR 2001. Proceedings of the 2001 IEEE Computer Society Conference on*, volume 1, pages I–I. IEEE, 2001.