Radial Lens Distortion Correction

NVIDIA CUDA Parallel Processing

CUDA Toolkit 9.1 - December 2017 Release
NVIDIA GeForce 840M (Performance Score 5.0x)

Student: Davide Stocco [194832]
davide.stocco@studenti.unitn.it
Tutor: Professor Nicola Conci
nicola.conci@unitn.it

October 9, 2018

Contents

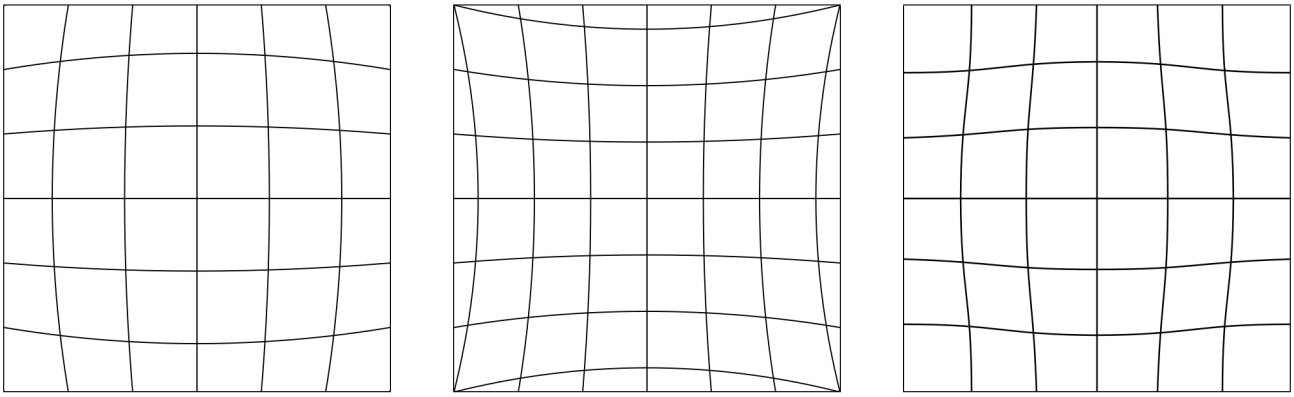
1 Introduction

This project is focused on speeding up a radial distortion correction of a camera capture. The reason behind this is to reach an high computational speed so that further operations can also be performed in real-time while video capturing. To reach such a speed we will *parallelize undistortion operations* by means of a NVIDIA GPU (Graphics Processing Unit). This GPU is provided with a CUDA (Compute Unified Device Architecture) accelerator that consists in a high number of threads in which to split and complete the task in less time.
Starting from a frame, OpenCV will convert it in a $M \times N$ `Mat` class dense array. Then the `Mat.data` substructure will be allocated in the CUDA device memory as a $1 \times (M \times N)$ array and undistorted by the CUDA kernel. When the undistortion process is done the data will be sent back to the host device memory (RAM) and converted again on a $M \times N$ `Mat` class dense array.

2 Radial Distortion

Radial distortion is a phenomenon that deforms an image. This unwanted but unavoidable phenomenon is due to a deviation from rectilinear projection and it is a form of optical aberration. Radial distortion is actually a big set of different kind of distortions that can occur. Thus, it can be furtherly classified as:

- *barrel* distortion;
- *pincushion* distortion;
- *mustache* distortion.



(a) Barrel distortion. (b) Pincushion distortion. (c) Mustache distortion.

Figure 1: Different types of radial distortion.

Mathematically, barrel and pincushion distortion are *quadratic*, meaning they increase as the square of distance from the center. In mustache distortion the *quartic* term is significant: in the center, the degree two barrel distortion is dominant, while at the edge the degree four distortion in the pincushion direction dominates. In this discussion we will only focus on barrel and pincushion distortion.

2.1 Software Correction

Radial distortion can be corrected by using the *Brown-Conrady model*. This model corrects both radial and tangential distortions caused by misaligned elements in a lens. The most commonly

used distortion correction model in softwares is known as *decentering distortion model*. It can be expressed in mathematical form as:

$$\begin{aligned} x_u = x_d + \dots \\ (x_d - x_c)(K_1 r^2 + K_2 r^4 + \dots) + \dots \\ (P_1(r^2 + 2(x_d - x_c)^2) + 2P_2(x_d - x_c)(y_d - y_c))(1 + P_3 r^2 + P_4 r^4 \dots \end{aligned} \tag{1}$$

$$\begin{aligned} y_u = y_d + \dots \\ (y_d - y_c)(K_1 r^2 + K_2 r^4 + \dots) + \dots \\ (2P_1(x_d - x_c)(y_d - y_c)P_2(r^2 + 2(y_d - y_c)^2))(1 + P_3 r^2 + P_4 r^4 \dots \end{aligned} \tag{2}$$

where:

- (x_d, y_d) is the distorted image point;
- (x_u, y_u) is the undistorted image point;
- (x_c, y_c) is the distortion center;
- K_n is the n-th *radial distortion coefficient*;
- P_n is the n-th *tangential distortion coefficient*;
- $r = \sqrt{(x_d - x_c)^2 + (y_d - y_c)^2}$.

Each kind of distortion has a typical sign and behaviour of the K_n and P_n terms and they can be characterized with them.

2.2 Inverse Model Approach for Radial Distortion Correction

The Brown-Conrady model is non-linear, thus also not invertible. To invert such non-linear model we would need a numeric approach, which is quite precise but not as fast as we would it to be. Pierre Drap and Julien Lefèvre in [1] proposed an *original approach* which aims us to compute the inverse transformation of a pure radial distortion model[1].
Given a model of distortion or correction with parameters $(k_1, k_2, k_3, \dots)$, they expressed the inverse transformation on the same form of the direct transformation, i.e. with parameters $(k'_1, k'_2, k'_3, \dots)$. Therefore they expressed each k'_i as a function of all the k_j. More practically, given the generic transformation T, they demonstrate that

$$T : \begin{pmatrix} x_u \\ y_u \end{pmatrix} \rightarrow \begin{pmatrix} x_d \\ y_d \end{pmatrix} = Q(r') \begin{pmatrix} x_u \\ y_u \end{pmatrix} \tag{3}$$

where $r' = \sqrt{(x_u - x_c)^2 + (y_u - y_c)^2}$ and $Q(r')$ can be expressed as a power series that authors have demonstrated to be

$$Q(r') = \sum_{n=0}^{+\infty} b_n r'^{2n} \tag{4}$$

Drap and Lefèvre also found a recursive formula to compute b_n terms as function of distortion or correction parameters $(-k_1, -k_2, -k_3, \dots)$. For this application we will stop at the first four terms, which are:

- $b_1 = -k_1$;
- $b_2 = 3k_1^2 - k2$;
- $b_3 = -12k_1^3 + 8k_1k_2 - k_3$;
- $b_4 = 55k_1^4 - 55k_1^2k_2 + 5k_2^2 + 10k_1k_3 - k_4$.

[1] Pure radial distortion model is characterized by $P_n = 0$ with $n = 1, \dots, n$.

2.3 Advantages of the Inverse Algorithm over Brown-Conrady Model

An example of the final result of distortion or correction process using Brown-Conrady model is shown in Figure 2. Notice that black lines appear on the most stretched sides of the image. This phenomenon occurs when the remapping operation shift pixels by a distance which in modulus is bigger than a unit. Unluckily, this kind of visual artifact cannot be avoided and a further software corrections are needed in order to interpolate pixel values in the surroundings of holes.

(a) Original image. (b) Barrel distortion. (c) Pincushion distortion.

Figure 2: Distortion of a sample image using Brown-Conrady model. Notice the visual artifacts that appear not only when applying pincushion distortion (Figure 2b) but also when applying barrel distortion with a light tangential distortion (Figure 2c).

The main advantage of the inverse approach is that starting from the destination or corrected image and by the inverse transformation, we can compute the a priori position of each pixel. With this method every pixel of the destination image is mapped to its original position in the source or distorted image, the result is that no empty spaces or visual artifacts are left in the corrected image.

(a) Original image. (b) Barrel distortion. (c) Pincushion distortion.

Figure 3: Distortion of a sample image using inverse model proposed by Pierre Drap and Julien Lefèvre in [1]. Notice that visual artifacts disappear.

2.4 Bilinear Interpolation

In computer vision and image processing, bilinear interpolation is one of the basic resampling techniques. When an image needs to be scaled up, each pixel of the original image needs to be

moved in a certain direction based on the scale constant. However, when scaling up an image by a non-integral scale factor, there are pixels that are not assigned appropriate pixel values. In this case, those holes should be assigned appropriate RGB or grayscale values so that the output image does not have non-valued pixels.
Bilinear interpolation can be used where perfect image transformation with pixel matching is impossible, so that one can calculate and assign appropriate intensity values to pixels. Unlike other interpolation techniques such as nearest-neighbor interpolation and bicubic interpolation, bilinear interpolation uses values of only the four nearest pixels, located in diagonal directions from a given pixel, in order to find the appropriate color intensity values of that pixel.
Bilinear interpolation considers the closest 2×2 neighborhood of known pixel intensity values surrounding the unknown pixel's computed location as

$$\begin{aligned} I(x, y_1) &\approx \frac{x_2 - x}{x_2 - x_1} I_{11} + \frac{x - x_1}{x_2 - x_1} I_{21} \\ I(x, y_2) &\approx \frac{x_2 - x}{x_2 - x_1} I_{12} + \frac{x - x_1}{x_2 - x_1} I_{22} \end{aligned} \tag{5}$$

It then takes a weighted average of these four pixels to arrive at its final, interpolated value

$$I(x, y) \approx \frac{y_2 - y}{y_2 - y_1} I(x, y_1) + \frac{y - y_1}{y_2 - y_1} I(x, y_2) \tag{6}$$

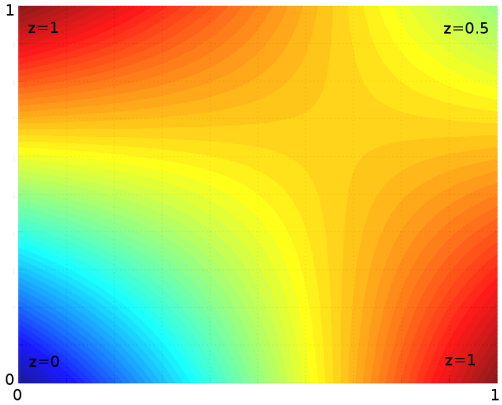


Figure 4: Example of bilinear interpolation on the unit square with the z values 0, 1, 1 and 0.5 as indicated. Interpolated values in between represented by color.

3 CUDA Background

3.1 History of CUDA

Driven by the market demand for real-time and high-definition graphics, the programmable GPU has evolved into a highly parallel, multi-threaded and many core processor with high computational power and large memory bandwidth. More specifically, the GPU is especially well suited to address problems that can be expressed as *data-parallel computations*[2] with *high arithmetic intensity*[3]. Because the same program is executed for each data element, there is a lower requirement for sophisticated flow control, and because it is executed on many data

[2] The same program is executed on many data elements in parallel.
[3] The ratio of arithmetic operations to memory operations.

elements and has high arithmetic intensity, the memory access latency can be hidden with calculations instead of big data caches. Data-parallel processing maps data elements to parallel processing threads. Many applications that process large data sets can use a data-parallel programming model to speed up the computations.
CUDA was introduced by NVIDIA in 2006 as a general purpose parallel computing platform and programming model. It uses the parallel compute engine in NVIDIA GPUs to solve many complex computational problems in a more efficient way than on a CPU.

3.2 CUDA GPUs

The advent of multi-core CPUs and many-core GPUs means that mainstream processor chips are now parallel systems. The challenge is to develop application software that scales its parallelism to exploit the increasing number of processor cores. The CUDA parallel programming model is designed to overcome this challenge while maintaining a low learning curve for programmers familiar with standard programming languages such as C. At its core there are three key abstractions[4] that are simply exposed to the programmer as a minimal set of language extensions. These abstractions provide fine-grained *data parallelism* and *thread parallelism*, nested within coarse-grained *data parallelism* and *task parallelism*. They guide the programmer to partition the problem into coarse sub-problems that can be solved independently in parallel by blocks of threads, and each sub-problem into finer pieces that can be solved cooperatively in parallel by all threads within the block.
This decomposition preserves language expressivity by allowing threads to cooperate when solving each sub-problem, and at the same time enables automatic scalability. Indeed, each block of threads can be scheduled on any of the available multiprocessors within a GPU, in any order, *concurrently* or *sequentially*, so that a compiled CUDA program can execute on any number of multiprocessors, and only the runtime system needs to know the physical multiprocessor count.

3.3 Advantages and Limitations

CUDA has advantages over traditional General-Purpose computation on GPUs (GPGPU) using graphics APIs:

- scattered reads[5];
- unified virtual memory;
- unified memory;
- fast shared memory among threads;
- faster downloads and read-backs to and from the GPU;
- full support for integer and bitwise operations, including integer texture lookups.

But it has some disadvantages too:

- limited interoperability [6];
- copying between host and device memory may incur a performance hit due to system bus bandwidth and latency[7];
- threads should be running in groups of at least 32 for best performance.

[4] A hierarchy of thread groups, shared memories, and barrier synchronization.
[5] Code can read from arbitrary addresses in memory.
[6] Interoperability with other languages such as OpenGL is one-way, with OpenGL having access to registered CUDA memory but CUDA not having access to OpenGL memory.
[7] This can be partly alleviated with asynchronous memory transfers.

4 CUDA Coding

In this section the CUDA kernel code and related parts to undistort or correct images will be analyzed.

4.1 CUDA Error Checking

Checking the results returned by CUDA API functions is important to ensure getting `cudaSuccess`, which means that the API call returned with no errors.

```
// Define CUDA Error Check
#define gpuErrchk(ans) { gpuAssert((ans), __FILE__, __LINE__); }
inline void gpuAssert(cudaError_t code, const char *file, int line, bool abort=false)
{
   if (code != cudaSuccess)
   {
      fprintf(stderr,"GPUassert: %s %s %d\n", cudaGetErrorString(code), file, line);
      if (abort) exit(code);
   }
}
```

As it can be seen, the error checking function `gpuErrchk()` only needs the CUDA command as input and it will print on the command window a string of the form "`GPUassert: #Type_of_error #File #Line_number`". It also can be chosen whether to abort or continue the execution of the program if an error occurs.

4.2 CUDA Kernel

The CUDA kernel is the core of the program, it defines the GPU tasks and how to perform these tasks. CUDA kernels must be declared before any host function.

```
// CUDA Kernel Undistortion
__global__ void undistort(uchar *image, uchar *image_out,
                          float k1, float k2, float k3, float k4,
                          int width, int height)
{
   // Setting texture memory
   uint i = blockIdx.x * blockDim.x + threadIdx.x;
   uint j = blockIdx.y * blockDim.y + threadIdx.y;

   // Define distortion center and polar coordinates
   float x_c = width / 2;
   float y_c = height / 2;
   float r = sqrtf(powf(i-x_c,2)+powf(j-y_c,2));

   // Calculate bn coefficents and Inverse transformation Q
   // Notice that b0 = 1
   float b1 = -k1;
   float b2 = 3*powf(k1,2) - k2;
   float b3 = 8*k1*k2 - 12*powf(k1,3) - k3;
   float b4 = 55*powf(k1,4) + 10*k1*k3 - 55*powf(k1,2)*k2 + 5*powf(k2,2) - k4;
   float Q = 1 + b1*powf(r,2) + b2*powf(r,4) + b3*powf(r,6) + b4*powf(r,8);

   // Final x,y coordinates
   float x = (i-x_c)*Q;
   float y = (j-y_c)*Q;
   int offset = x_c + y_c * width;
```

```
60
61    // Bilinear Interpolation Algorithm
62    // Ensure that the undistorted point sits in the image
63    if(i<=width && j<=height)
64    {
65       // Define intermediate points
66       int x1 = floorf(x);
67       int y1 = floorf(y);
68       int x2 = x1 + 1;
69       int y2 = y1 + 1;
70
71       // Ensure that the mean value can be computable
72       if(x1>=-x_c && y1>=-y_c && x2<(width/2) && y2<(height/2))
73       {
74          float val1 = image[x1 + y1 * width + offset];
75          float val2 = image[x2 + y1 * width + offset];
76          float val3 = image[x1 + y2 * width + offset];
77          float val4 = image[x2 + y2 * width + offset];
78
79          float color1 = (x2-x)*val1 + (x-x1)*val2;
80          float color2 = (x2-x)*val3 + (x-x1)*val4;
81          float color = (y2-y)*color1 + (y-y1)*color2;
82
83          image_out[i + j * width] = color;
84       }
85    }
86 }
```

In the first part the `undistort` function is introduced as a `__global__` kernel[8]. The inputs are the two `Mat.data` substructure arrays, the first contains the source image while the second is a zero valued array (black image) which will be overwritten in the remapping process. Both `Mat.data` are automatically given in a a $1 \times (M \times N)$ layout. Other inputs are the distortion values and the two dimensions of the image.
Texture memory defines the indexing order, or in other words in which order to perform the remapping process. In this case the kernel will subdivide the image in a 2D texture of `blockDim.x` $\times$ `blockDim.y` windows dimensions and will address each pixel to a specific thread. Final coordinates are defined as integer numbers because they will correspond to a specific pixel position in the final image. Distortion center is initialized as a float for straightforwardness[9]. Then the *inverse remapping process* starts with the calculation of the `r` coefficient. By using the formula given in (4) and b_n coefficients, the original position of each pixel in the undistorted or corrected image is computed. Of course, the position will not be an integer and color intensity value is computed through the previously presented bilinear interpolation algorithm.

4.3 Main Function

The `main` function consists on a `while` loop that will ask for distortion parameters modifications and will return the output image until the exit key is given on the command window. More specifically, a `switch` function will do the job. In this section of the code distortion parameters `k1`, `k2`, `k3` and `k4`, can be increased, decreased, reseted to zero and shown in the command window.

[8] In CUDA there are two types of functions, `__global__` functions (or kernels), which can be called from host code by the call semantics (`<<<...>>>`), and `__device__` functions, which cannot be called from host code.
[9] It would be anyway converted to a float number within `powf` and `sqrtf` functions.

4.3.1 System Variables Allocation

```
186       // Init system variables and read img
187       Mat image = imread("C:\\...", 0);
188       uchar * img;
189
190       Mat image_out = Mat(image.rows, image.cols, CV_8UC1, double(0));
191       uchar * img_out;
192
193       int N = image.rows*image.cols;
194
195       // Init CUDA stream
196       cudaStream_t myStream;
197       cudaError_t resultStream;
198       resultStream = cudaStreamCreate(&myStream);
199
200       //To GPU
201       gpuErrchk( cudaFree(0) );
202       gpuErrchk( cudaMalloc(&img, N*sizeof(uchar)) );
203       gpuErrchk( cudaMemcpyAsync(img, image.data, N*sizeof(uchar), cudaMemcpyHostToDevice,
    myStream) );
204       gpuErrchk( cudaMalloc(&img_out, N*sizeof(uchar)) );
205       gpuErrchk( cudaMemcpyAsync(img_out, image_out.data, N*sizeof(uchar),
    cudaMemcpyHostToDevice, myStream) );
```

The source image is transformed into a one channel `CV_8UC1 Mat` dense class array and an `img` pointer is initialized. The same is done also for the zero valued `Mat`. A CUDA stream is launched and the allocation of data in the GPU starts by means of `cudaMalloc` function. Then the data is copied from the RAM to the GPU memory through `cudaMemcpyAsync` function which specifies the source data, the destination pointer, the direction of data flow (in this case `cudaMemcpyHostToDevice`) and in which stream it operates.

4.3.2 Kernel Settings, Launch and Output Error Checking

```
207       // Kernel settings
208       dim3 dimBlock(256, 540); // for a 16:9 4K image
209       dim3 numBlocks(image.cols/dimBlock.x, image.rows/dimBlock.y);
210
211       // Apply an inplace kernel on img
212       clock_t start_CUDA = clock();
213       undistort <<<dimBlock, numBlocks, 0, myStream>>> (img, img_out, -k1, -k2, -k3, -k4,
    image.cols, image.rows);
214       cudaDeviceSynchronize();
215       gpuErrchk( cudaPeekAtLastError() );
216       clock_t stop_CUDA = clock();
217       double elapsed_CUDA = ((stop_CUDA - start_CUDA) * 1000.0 / (double)CLOCKS_PER_SEC);
218       cout << "Custom CUDA kernel - ";
219       printf("Time elapsed in ms: %.2f\n", elapsed_CUDA);
220       gpuErrchk( cudaPeekAtLastError() );
221
222       // Check for CUDA errors
223       cudaError_t error = cudaGetLastError();
224       if (error != cudaSuccess){
225          fprintf(stderr, "ERROR: %s \n", cudaGetErrorString(error));
226       }
```

Dimensions of block texture in which to subdivide the capture are arbitrarily decided through `dimBlock` function[10], while the calculation of blocks number is automatized. After setting these parameters, the `undistort` is called and inputs are given accordingly.
Further CUDA error checking is performed and `cudaPeekAtLastError` returns the last error that has been produced by any of the runtime calls in the same host thread. Possible errors on stream are also checked.
Execution time of kernel is a key factor for evaluating performance. Thus, it is measured from kernel invoking to the end of `cudaDeviceSynchronize()`, which waits for every thread and block to finish before stopping the timer.

4.3.3 Copy Data from GPU Memory to RAM

```
228       //To CPU
229       uchar * out = (uchar*)malloc(N * sizeof(uchar));
230       gpuErrchk( cudaMemcpyAsync(out, img_out, N * sizeof(uchar), cudaMemcpyDeviceToHost,
      myStream) );
231       Mat out_Mat = Mat(image.rows, image.cols, CV_8UC1, out);
```

In the final part of the code the $1 \times (M \times N)$ output array is copied from GPU memory to RAM through `cudaMemcpyAsync` function and now the data flow is defined as `cudaMemcpyDeviceToHost`. Finally, the data is converted again in a one channel $M \times N$ `Mat` array.

5 Code Execution and Results

Code execution is performed through a proper integrated development environment (Microsoft Visual Studio). Here is an example of what appears in the command window during execution.

```
CUDA KERNEL FOR RADIAL UNDISTORTION
Press key:
'a'=+k1 'z'=-k1 's'=+k2 'x'=+k2
'd'=+k3 'c'=-k3 'f'=+k4 'v'=+k4
'w' = See current distortion values
'r' = Reset distortion values
'q' = Quit program

Key:    a
Custom CUDA kernel - Time elapsed in ms: 16.00
OpenCV Function - Time elapsed in ms: 266.00
Key:    d
Custom CUDA kernel - Time elapsed in ms: 15.00
OpenCV Function - Time elapsed in ms: 2250.00
Key:    w
Current distortion values are:
k1 = -1e-015
k2 = 0
k3 = -1e-015
k4 = 0
Key:    r
Reset distortion values!
```

[10] Actually, the function `dimBlock` specifies the number of threads per block. In this application thread IDs correspond with pixels, so the numbers given in `dimBlock` function correspond to the pixel dimensions of the window in which to subdivide the source frame.

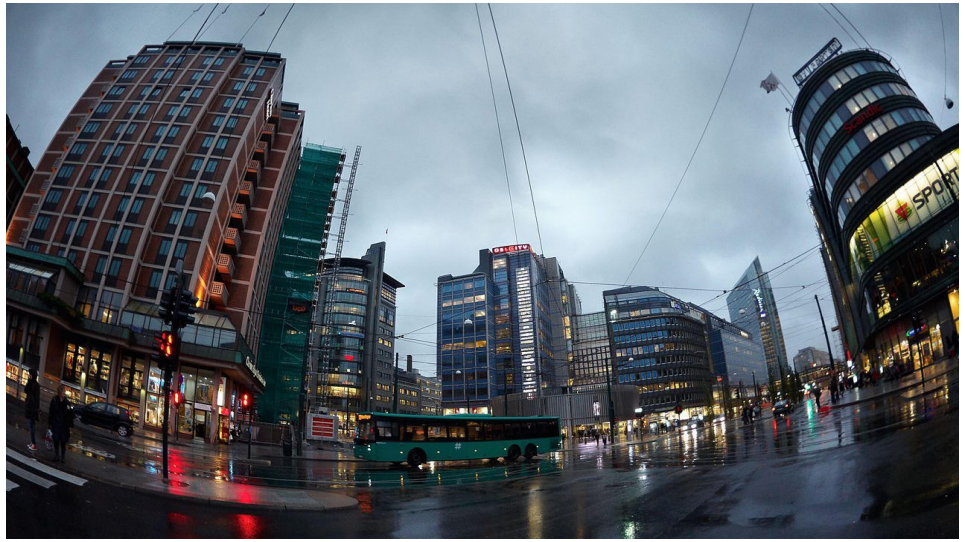
(a) Original image.

(b) `k1 = 1.16861e-015`, `k2 = 1e-014`, `k3 = 9e-021`, `k4 = 0`.

(c) `k1 = 0`, `k2 = 1e-014`, `k3 = 2e-021`, `k4 = 0`.

(d) `k1 = 1e-007`, `k2 = -1e-014`, `k3 = -3e-021`, `k4 = 0`.

Figure 5: Different types of radial distortion - Sample image 4K 16:9.



(a) Original image.

(b) `k1 = 0`, `k2 = 1e-014`, `k3 = 3e-021`, `k4 = 0`.

(c) `k1 = -1e-007`, `k2 = 1e-014`, `k3 = 3e-21`, `k4 = 0`.

(d) `k1 = 0`, `k2 = -1e-014`, `k3 = 0`, `k4 = -8e-028`.

Figure 6: Different types of radial distortion - Sample image 5K 16:9.

6 Conclusions

This code can perform the undistortion or correction of a image but there are some drawbacks. The first is the loss of information due to the conversion of the source image into a one channel `Mat` array. The second drawback consists in the limited capability of images handling. More specifically, the program can only handle frames which dimensions are multiple of (`blockDim.x`, `blockDim.y`) and they should be tuned manually depending on the image dimensions.
An important observation on distortion parameters magnitudes should be done. These parameters are really small in general, and depending on the size of the source image they can vary by orders of magnitude. Thus, the increasing/decreasing step in the distortion parameters could be not suitable depending on the size of source image.
The simplicity of the code is evident and the only bottleneck that could make the code slow down is the data transfer between RAM to GPU memory and vice versa. For a big data set this is not that important since the execution time in GPU is much faster that how much should be in CPU. The time loss in the data transferring is in fact greatly counterbalanced by a fast execution. For a 4K image with a 16:9 aspect ratio, execution time for this CUDA kernel takes about $15 \div 30$ milliseconds. This means that the program can process a 4K video in real-time with frame rate up to 30 fps in the worst case.
Last but not least, the approach proposed by Pierre Drap and Julien Lefèvre in [1] is mathematically exact (based on a power series). Since the expansion has been stopped at the first four terms the approach should be considered an *approximation* of the Brown-Conrady model.

References

[1] *An Exact Formula for Calculating Inverse Radial Lens Distortions*
Pierre Drap & Julien Lefèvre
Aix-Marseille Université, CNRS, ENSAM, Université De Toulon, LSIS UMR 7296, Domaine Universitaire de Saint-Jérôme, Bâtiment Polytech, Avenue Escadrille Normandie-Niemen, Marseille 13397, France

[2] *CUDA C Programming Guide* - PG-02829-001_v9.1 - March 2018 - Design Guide

[3] *CUDA C Best Practices Guide* - DG-05603-001_v9.1 - March 2018 - Design Guide

[4] *OpenCV Documentation* - `https://docs.opencv.org`

[5] *Distortion (optics)* - Wikipedia, the free encyclopedia - `https://en.wikipedia.org/wiki/Distortion_(optics)`

[6] *Bilinear interpolation* - Wikipedia, the free encyclopedia - `https://en.wikipedia.org/wiki/Bilinear_interpolation`