

# Radial Undistortion of Images

## NVIDIA CUDA Parallel Processing

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

Davide Stocco

February 21, 2023

## Contents

<b>1</b>	<b>Introduction . . . . .</b>	<b>2</b>
<b>2</b>	<b>Radial Distortion . . . . .</b>	<b>2</b>
2.1	Software Correction . . . . .	3
2.2	Inverse Model Approach for Radial Distortion Correction . . . . .	3
2.3	Advantages of the Inverse Algorithm . . . . .	4
2.4	Bilinear Interpolation . . . . .	4
<b>3</b>	<b>CUDA Background . . . . .</b>	<b>5</b>
3.1	History of CUDA . . . . .	5
3.2	CUDA GPUs . . . . .	6
3.3	Advantages and Limitations . . . . .	7
<b>4</b>	<b>CUDA Coding . . . . .</b>	<b>7</b>
4.1	CUDA Error Checking . . . . .	7
4.2	CUDA Kernel . . . . .	8
4.3	Main Function . . . . .	9
4.3.1	System Variables Allocation . . . . .	9
4.3.2	Kernel Settings, Launch and Output Error Checking . . . . .	10
4.3.3	Copy Data from GPU Memory to RAM . . . . .	11
<b>5</b>	<b>Code Execution and Results . . . . .</b>	<b>11</b>
<b>6</b>	<b>Conclusions . . . . .</b>	<b>12</b>

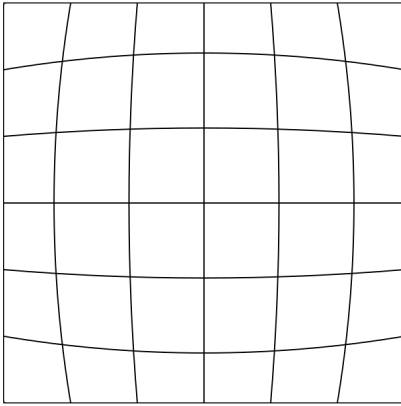
# 1 Introduction

This project is focused on speeding up a radial distortion correction of a camera capture. The reason behind this is to reach a high computational speed so that further operations can also be performed in real-time while video capturing. To reach such a target we will parallelize undistortion operations by means of an NVIDIA GPU (*Graphics Processing Unit*). This GPU is provided with a CUDA (*Compute Unified Device Architecture*) accelerator that consists of a high number of threads in which to split and complete the task in less time. In other words, starting from an image frame, OPENCV will convert it into 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 an OPENCV  $M \times N$  `Mat` class dense array.

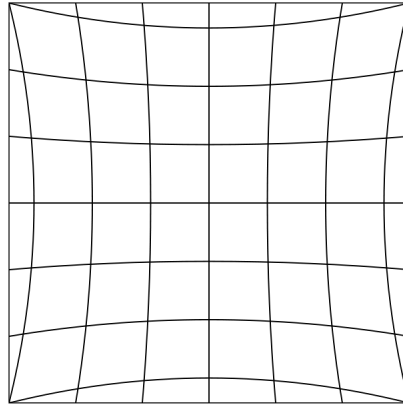
## 2 Radial Distortion

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

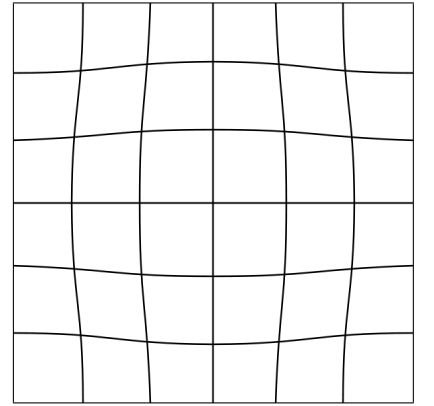
- *barrel* distortion;
- *pincushion* distortion;
- *mustache* distortion.



(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 the distance from the centre. In moustache distortion the *quartic* term is significant: in the centre, the degree two barrel distortion is dominant, while at the edge the degree four distortions in the pincushion direction dominate. 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 software is known as *decentering distortion model*, and it can be expressed in mathematical form as:

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

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

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 value of the  $K_n$  and  $P_n$  terms, hence they can be characterized accordingly.

## 2.2 Inverse Model Approach for Radial Distortion Correction

The Brown-Conrady model is non-linear, and thus also not invertible. To invert such a 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<sup>1</sup>.

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:

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

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:

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

---

<sup>1</sup>Pure radial distortion model is characterized by  $P_n = 0$  with  $n = 1, \dots, n$ .



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

Drap and Lefèvre also found a recursive formula to compute  $b_n$  terms as a 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:

$$\begin{aligned}
 b_1 &= -k_1 \\
 b_2 &= 3k_1^2 - k_2 \\
 b_3 &= -12k_1^3 + 8k_1k_2 - k_3 \\
 b_4 &= 55k_1^4 - 55k_1^2k_2 + 5k_2^2 + 10k_1k_3 - k_4.
 \end{aligned} \tag{5}$$

### 2.3 Advantages of the Inverse Algorithm

An example of the final result of the distortion or correction process using the 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 shifts pixels by a distance which in modulus is bigger than a unit. Unluckily, this kind of visual artefact cannot be avoided and further software corrections are needed in order to interpolate pixel values in the surroundings of holes. 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 artefacts are left in the corrected image.

### 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-integer scale factor, some pixels 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.



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

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-neighbour interpolation and bi-cubic 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 colour intensity values of that pixel.

Bilinear interpolation considers the closest  $2 \times 2$  neighbourhood of known pixel intensity values surrounding the unknown pixel's computed location as:

$$\begin{aligned} I(x, y_1) &\approx \frac{x_2 - x}{x_2 - x_1} I_{11} + \frac{x - x_1}{x_2 - x_1} I_{21} \\ I(x, y_2) &\approx \frac{x_2 - x}{x_2 - x_1} I_{12} + \frac{x - x_1}{x_2 - x_1} I_{22}. \end{aligned} \quad (6)$$

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

$$I(x, y) \approx \frac{y_2 - y}{y_2 - y_1} I(x, y_1) + \frac{y - y_1}{y_2 - y_1} I(x, y_2). \quad (7)$$

## 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 multi-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*<sup>2</sup> with *high arithmetic intensity*<sup>3</sup>. 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 elements and has high

<sup>2</sup>The same program is executed on many data elements in parallel.

<sup>3</sup>The ratio of arithmetic operations to memory operations.

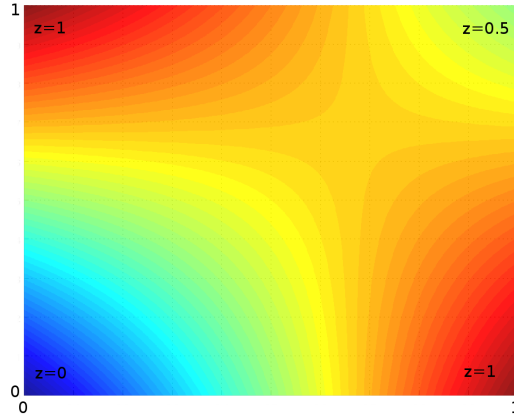


Figure 4: Example of bilinear interpolation on the unit square with  $z$  values equal to 0, 1, 1 and 0.5. Interpolated values in between are represented by colour shades.

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 computing 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<sup>4</sup> 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.

---

<sup>4</sup>A hierarchy of thread groups, shared memories, and barrier synchronization.

### 3.3 Advantages and Limitations

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

- scattered reads<sup>5</sup>;
- 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 <sup>6</sup>
- copying between host and device memory may incur a performance hit due to system bus bandwidth and latency<sup>7</sup>;
- threads should be running in groups of at least 32 for best performance.

## 4 CUDA Coding

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

### 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 is returned with no errors.

```
1 // \Cuda{} error check
2 inline
3 void
4 GPU_Assert(
5     cudaError_t code,          // Error code
6     const char *file,         // File name
7     int line,                 // Line number
8     bool abort = false // Abort program flag
9 )
10 {
11     if (code != cudaSuccess)
12     {
13         fprintf(stderr, "GPUassert: %s %s %d\n", cudaGetErrorString(code), file, line);
14         if (abort == true)
15             exit(code);
16     }
17 }
```

---

<sup>5</sup>Code can read from arbitrary addresses in memory.

<sup>6</sup>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.

<sup>7</sup>This can be partly alleviated with asynchronous memory transfers.

```

16 }
17 }

```

As it can be seen, the error checking function `GPU_Assert` only needs the CUDA command as input and it will print 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.

```

1 // \Cuda{} kernel for undistortion
2 __global__
3 void
4 undistort(
5     uchar *image_in,      // Input image
6     uchar *image_out_CUDA, // Output image
7     double k1,            // Radial distortion coefficient k1
8     double k2,            // Radial distortion coefficient k2
9     double k3,            // Radial distortion coefficient k3
10    double k4,            // Radial distortion coefficient k4
11    int width,            // Image width
12    int height           // Image height
13 )
14 {
15     // Setting texture memory
16     uint i = blockIdx.x * blockDim.x + threadIdx.x ;
17     uint j = blockIdx.y * blockDim.y + threadIdx.y;
18
19     // Define distortion center and polar coordinates
20     double xc = width / 2.0;
21     double yc = height / 2.0;
22     double r = std::sqrt((i-xc)*(i-xc) + (j-yc)*(j-yc));
23
24     double k1_2 = k1*k1;
25     double r_2 = r*r;
26     double r_4 = r_2*r_2;
27
28     // Calculate bn coefficients and Inverse transformation Q
29     // Notice that b0 = 1
30     double b1 = -k1;
31     double b2 = 3.0*k1_2 - k2;
32     double b3 = 8.0*k1*k2 - 12.0*k1_2*k1 - k3;
33     double b4 = 55.0*k1_2*k1_2 + 10.0*k1*k3 - 55.0*k1_2*k2 + 5*k2*k2 - k4;
34     double Q = 1.0 + b1*r_2 + b2*r_4 + b3*r_2*r_4 + b4*r_4*r_4;
35
36     // Final (x,y) coordinates
37     double x = (i-xc) * Q;
38     double y = (j-yc) * Q;
39     int o = xc + yc * width;
40
41     // Bilinear Interpolation Algorithm
42     // Ensure that the undistorted point sits in the image
43     if (i <= width && j <= height)

```



```

44 {
45     // Define intermediate points
46     int x1 = std::floor(x);
47     int y1 = std::floor(y);
48     int x2 = x1 + 1;
49     int y2 = y1 + 1;
50
51     // Ensure that the mean color value is computable
52     if(x1 >= -xc && y1 >= -yc && x2 < (width/2.0) && y2 < (height/2.0))
53     {
54         double val1 = image_in[x1 + y1 * width + o];
55         double val2 = image_in[x2 + y1 * width + o];
56         double val3 = image_in[x1 + y2 * width + o];
57         double val4 = image_in[x2 + y2 * width + o];
58
59         double color1 = (x2 - x) * val1 + (x - x1) * val2;
60         double color2 = (x2 - x) * val3 + (x - x1) * val4;
61
62         image_out_CUDA[i + j * width] = (y2 - y) * color1 + (y - y1) * color2;
63     }
64 }
65 }

```

In the first part the `undistort` function is introduced as a `__global__` kernel<sup>8</sup>. 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  $1 \times (M \times N)$  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 centre is initialized as a float for straightforwardness<sup>9</sup>. 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 the colour intensity value is computed through the previously presented bilinear interpolation algorithm.

## 4.3 Main Function

The `main` function consists of 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, reset to zero and shown in the command window.

### 4.3.1 System Variables Allocation

<sup>8</sup>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.

<sup>9</sup>It would be anyway converted to a float number within `powf` and `sqrtof` functions.

```

1 // Read input image
2 Mat image_in = imread("path-to-image-file", 0);
3 uchar *img_in;
4
5 // Allocate output image
6 Mat image_out_CUDA = Mat(image_in.rows, image_in.cols, CV_8UC1, double(0.0));
7 uchar *img_out;
8
9 // Get image size
10 int N = image_in.rows * image_in.cols;
11
12 // Initialize \Cuda{} stream and error
13 cudaStream_t CUDA_stream;
14 cudaError_t CUDA_error;
15 CUDA_error = cudaStreamCreate(&CUDA_stream);
16
17 // To GPU memory
18 GPU_ERROR(cudaFree(0));
19 GPU_ERROR(cudaMalloc(&img_in, N*sizeof(uchar)));
20 GPU_ERROR(cudaMemcpyAsync(img_in, image_in.data, N*sizeof(uchar), cudaMemcpyHostToDevice,
    CUDA_stream));
21 GPU_ERROR(cudaMalloc(&img_out, N*sizeof(uchar)));
22 GPU_ERROR(cudaMemcpyAsync(img_out, image_out_CUDA.data, N*sizeof(uchar), cudaMemcpyHostToDevice
    , CUDA_stream));

```

The source image is transformed into one channel CV\_8UC1 Mat dense class array and an img pointer are 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

```

1 // Kernel settings
2 dim3 dimBlock(256, 540);
3 dim3 numBlocks(image_in.cols/dimBlock.x, image_in.rows/dimBlock.y);
4
5 // Apply kernel on input image
6 clock_t CUDA_start_clock = clock();
7 undistort<<<dimBlock, numBlocks, 0, CUDA_stream>>>(
8     img_in, img_out, k1, k2, k3, k4, image_in.cols, image_in.rows
9 );
10 cudaDeviceSynchronize();
11 GPU_ERROR(cudaPeekAtLastError());
12 clock_t CUDA_stop_clock = clock();
13 double CUDA_elapsed_time =
14     (CUDA_stop_clock - CUDA_start_clock) * 1000.0 / (double)CLOCKS_PER_SEC;
15
16 std::cout
17     << "Custom \Cuda{} kernel - Elapsed time:"
18     << CUDA_elapsed_time << " (ms)" << std::endl;
19 GPU_ERROR(cudaPeekAtLastError());

```

```

20
21 // Check for \Cuda{} errors
22 cudaError_t CUDA_past_error = cudaGetLastError();
23 if (CUDA_past_error != cudaSuccess)
24 {
25     fprintf(stderr, "\Cuda{} ERROR: %s \n", cudaGetErrorString(CUDA_past_error));
26 }

```

Dimensions of block texture in which to subdivide the capture are arbitrarily decided through `dimBlock` function<sup>10</sup>, 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 the stream are also checked.

The execution time of the 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

```

1 // To CPU
2 uchar * out = (uchar*)malloc(N * sizeof(uchar));
3 GPU_ERROR(cudaMemcpyAsync(out, img_out, N * sizeof(uchar), cudaMemcpyDeviceToHost, CUDA_stream)
4 );
5 Mat out_Mat = Mat(image_in.rows, image_in.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 one channel  $M \times N$  `Mat` array.

## 5 Code Execution and Results

Code execution is performed through a properly 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 - Elapsed time: 16.00 (ms)
\OpenCV{} Function - Elapsed time: 266.00 (ms)
Key:    d

```

<sup>10</sup>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.

```

Custom \Cuda{} kernel - Elapsed time: 15.00 (ms)
\OpenCV{} Function - Elapsed time: 2250.00 (ms)
Key:    r

Reset distortion values!

Key:    x
Custom \Cuda{} kernel - Elapsed time: 22.00 (ms)
\OpenCV{} Function - Time elapsed in ms: 3062.00
Key:    z
Custom \Cuda{} kernel - Elapsed time: 15.00 (ms)
\OpenCV{} Function - Elapsed time: 563.00 (ms)
Key:    w

Current distortion values are:
k1 = -1e-015
k2 = -1e-015
k3 = 0
k4 = 0

```

## 6 Conclusions

This code can perform the undistortion or correction of an image but there are some drawbacks. The first is the loss of information due to the conversion of the source image into one channel `Mat` array. The second drawback consists of the limited capability of image handling. More specifically, the program can only handle frames whose dimensions are multiple of (`blockDim.x`, `blockDim.y`) and they should be tuned manually depending on the image dimensions.

An important observation on distortion parameter 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 the 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 than how much should be in CPU. The time loss in the data transfer 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 video in real-time with a frame rate of 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, Mar-  
seille 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\\_\(optics\)](https://en.wikipedia.org/wiki/Distortion_(optics))
- [6] *Bilinear interpolation* - Wikipedia, the free encyclopedia - [https://en.wikipedia.org/wiki/  
Bilinear\\_interpolation](https://en.wikipedia.org/wiki/Bilinear_interpolation)