OPENCL CONVOLUTION

CPU vs GPU

ABSTRACT

An OpenCL solution for image convolution performed on CPU and GPU

Tinu Toader

GPGPU

Contents

1. Convolution

"*Convolution is a simple mathematical operation which is fundamental to many common image processing operators. Convolution provides a way of <<multiplying together>> two arrays of numbers, generally of different sizes, but of the same dimensionality, to produce a third array of numbers of the same dimensionality. This can be used in image processing to implement operators whose output pixel values are simple linear combinations of certain input pixel values.*" (Fisher, Perkins, Walker, & Wolfart., 2013)

2. The problem

Given an input image and a sample code and that can generate a sharper version of the input using a serial execution on a Central Processing Unit, there is a requirement to parallelise and optimise the execution of the application to run on a Central Processing Unit or a General Purpose Graphic Processing Unit using OpenCL API.

3. The serial code

The serial CPU application is simple and quite limited: the user can input the path of a ppm image to be processed, the path where to save the sharpened image and another parameter, the radius, which will affect the output.

3.1 The logic

The image basically uses a negated average blur filter. Based on the radius that sets the area, for each pixel, the neighbours channels values are being added and divided by the number of the participants in the addition. For example, a radius 1 filter, we will have radius * 2 + 1 pixels on each direction, resulting a 3 x 3 grid, in the middle of which lies the current pixel that needs to be blurred. So, the value of all reds is divided by 8 and stored in the red channel of the central pixel, the value of the all greens and blues are also divided by 8 and the result is stored in the central pixel.

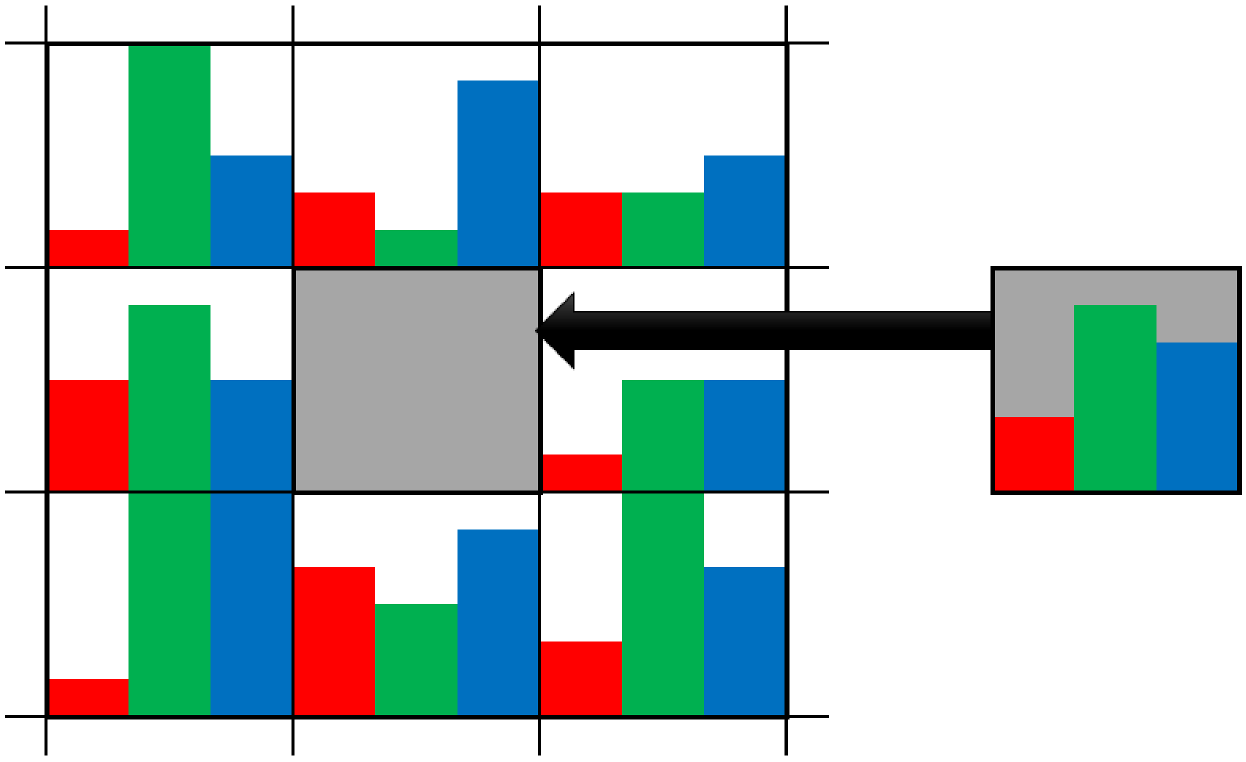


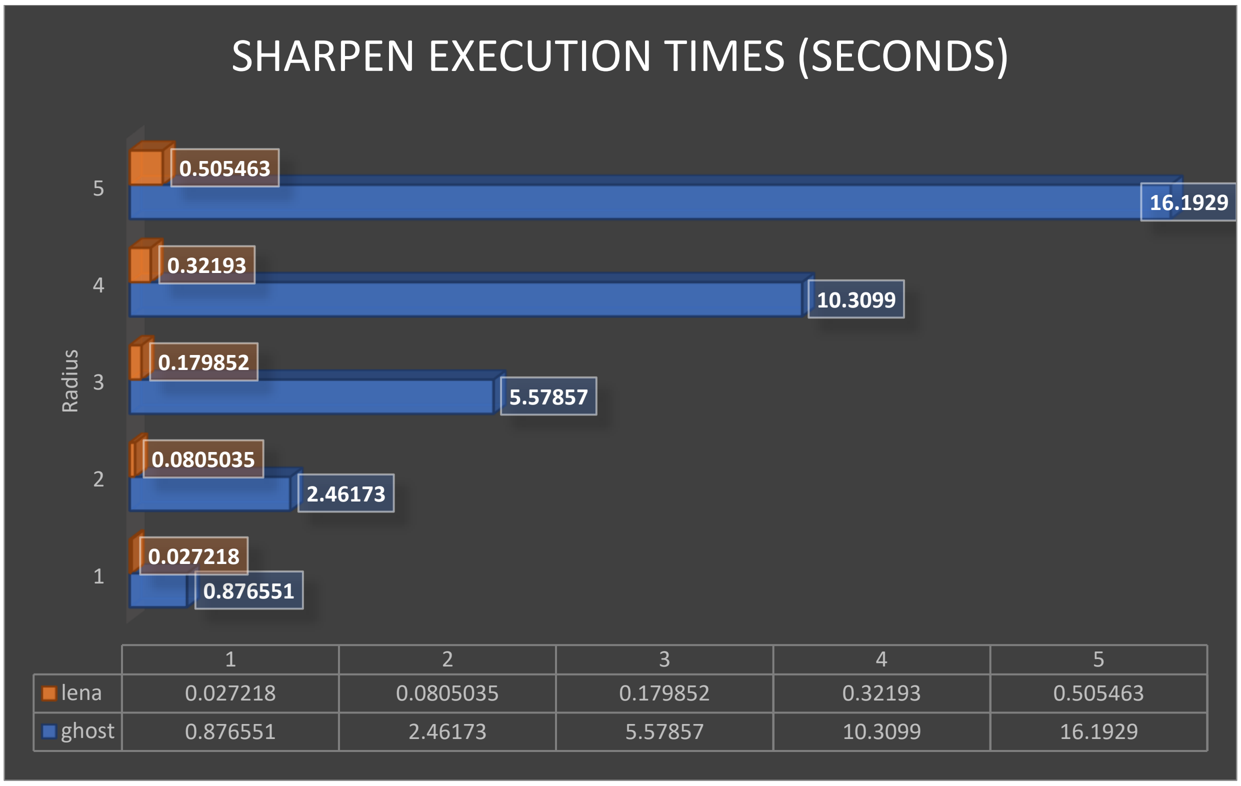
Figure 1. Average blur, 3x3 kernel

This process is repeated for every other pixel, saving all values in a separate result image. For radius 2, a bigger 5 x 5 grid is considered, so a higher number of pixels need to be averaged (24). Here we can already highlight a first observation, that the bigger the radius, the slower the application and the bigger the variation compared to the original pixel as more noise is considered when generating the average values.

The resulted image is blurred again and these two blurred images are combined and multiplied by some correction values which also influence the final output image: alpha (1.5), beta (-0.5) and gamma (0.0). In fact, without this final step, the resulted image would be blurred not sharpened, but thanks to the negative value of beta, the blur effect is inversed, resulting in a sharping effect.

3.2 The performance

Because the execution times differ from a computer to another, it is important to specify that the times described in this report are obtained on a machine with an Intel(R) Core(TM) i7-4790 CPU @ 3.60GHz processor and an AMD Radeon R9 270x GPU. Two images have been used to obtain timings for the serial code and the results are displayed in the following chart.



	1	2	3	4	5
🟧 lena	0.027218	0.0805035	0.179852	0.32193	0.505463
🟦 ghost	0.876551	2.46173	5.57857	10.3099	16.1929

Figure 2. Times obtained with the serial CPU application for two images

The first tested image, lena.ppm, is a 512x512 image with only 262,144 pixels and therefore it has faster execution times. The latter tested image, ghost.ppm, is a 3840x2160 image with 8,294,400, and has about 32 times more pixels to process and this is almost perfectly reflected in the execution times, as visible in the above chart (16/0.5 = 32).

3.3 The results

Following the execution of the application, several images were generated. For the rest of the project we will not consider using the ghost image to show the visual results, but only for benchmark. Using lena.ppm, we generated 1 image for each radius used, but because the difference between a result with radius 1 and a result with radius 2 is almost unnoticeable, in the following figure we present the results from a resulted image generated with a radius 1 kernel and a resulted image generated with a radius 5 kernel:

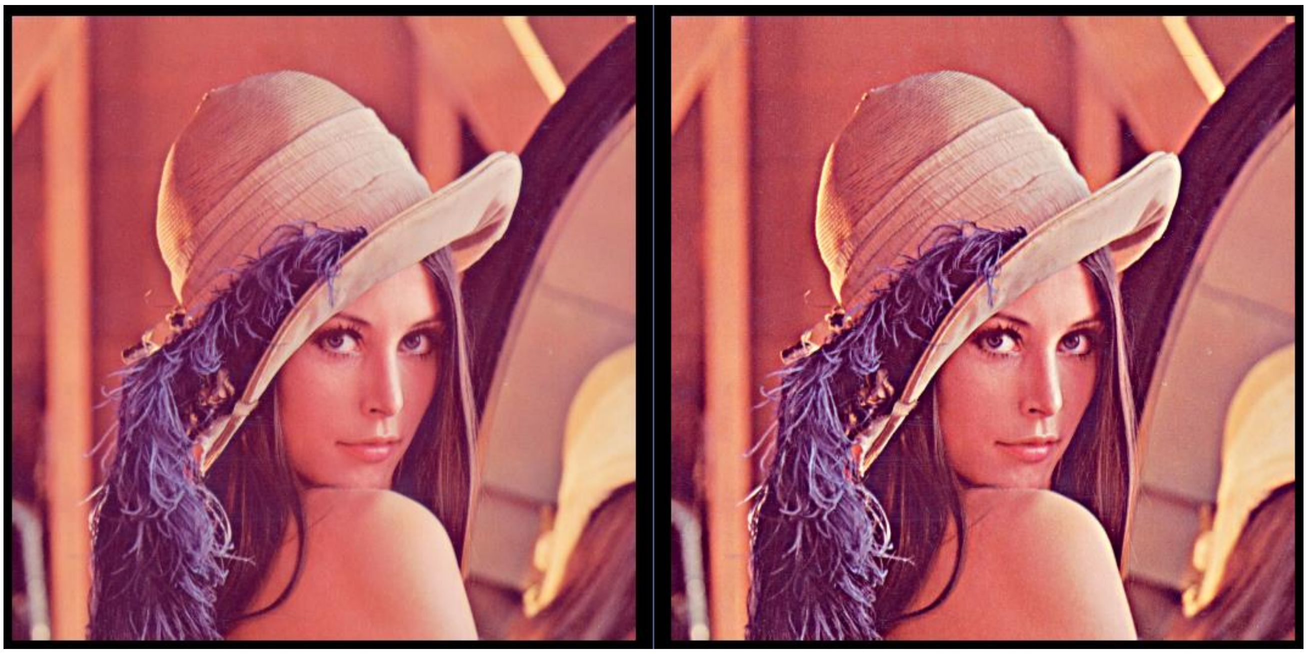


Figure 3. Sharp results: Lena radius 1 (left) and Lena radius 5 (right) CPU serial.

We can observe that increasing the radius, we obtain a sharper output.

3.4 Preliminary conclusions

- The serial code uses the average blur, and inverts the filter to obtain a sharp effect.
- Increasing the radius yields a sharper output
- Using a filter with default value (5) for radius, a 512x512 image was sharpened in 0.5 seconds, while a 3840x2160 image was sharpened in 16 seconds (32 times slower).

4. The solution

To improve the time and the application overall, the OpenCL API will be used. A parallel version of the serial code will be created and will be executed using OpenCL on the CPU and GPU. Two different blur kernels will be left available: the average and Gaussian blur. The solution should be faster than the serial code when running both on CPU and GPU using OpenCL.

If time will allow, the solution will be extended to allow real-time image manipulation by loading an image, taking input from the user, updating the image using OpenCL and displaying it in a window using OpenGL. To make the application fluid and to be rendered at minimum 24 frames per second, we need to have all OpenCL and Input updates completed in less than 41 milliseconds. We hope that using the OpenCL, we'll beat this time and create a general purpose editor which only for particular cases - huge kernels, or huge images – the application will not be very useful.

5. The plan

To build the solution, in simple steps we had to read an image, send it to an OpenCL device, run parallelized program for each pixel on device, retrieve the image data and save it or display it. To do this and to refresh the OpenCL knowledge, we followed the following plan:

1. Create basic OpenCL application with identified platform(s) and device(s)
2. Create an application with a working empty kernel
3. Create an application where some data is sent to the device and back to host

4. Successfully load the ppm with 4 channels
5. Successfully save back to disk a ppm from a 4 channels image
6. Create parametrised Gaussian filter
7. Create working kernel for sharpening
8. Encapsulate sharp execution in a parametrised function or a class
9. Add execution tracking functionality, and dump timings to excel format to ease chart generation
10. Create OpenGL window
11. Create a mean of communication between OpenGL and OpenCL
12. Get input and change Gaussian filter values based on input
13. Display image on OpenGL window
14. Update image shown on OpenGL window using OpenCL manipulation after input

5.1 The warm-up

First refreshing exercises were done in parallel with Youtube, following Shillingford's (2016) videos. After working with vectors and buffers, we came about read_Imageui, and we changed the approach. Instead of sending Buffers of data, we tried to send Image2Ds. Our first attempt was to send the image data in the natural way in the RGB format, but we were greeted with a compiler error -10. This is when we started to create a Util.h file where we stored Selmar's (2014) translations for the compiler errors. That's also the moment when we found out that -10 means `CL_IMAGE_FORMAT_NOT_SUPPORTED`, so my device doesn't like CL_RGB format. This meant that we continued to extend the Util.h.

5.2 The conversion

We have modified the given ppm image loader and added another value for alpha, after every 3 pixels read:

```
while (ss >> u) {
        result.push_back(u); // Yes, pushing an uint into a vector of uchars
        if (++counter % 3 == 0) {
                result.push_back(max);
        }
}
```

Seeing that we are reading in 4 channels, we also had to create a way to save back to 3 channels from 4. Of course, we modified the given ppm image writer:

```
unsigned skipper = 0;
for (const unsigned col : data) { // Yes, reading a uchar as a uint
        if (++skipper % 4) {
                ss << col << ' ';
        }
}
```

Meanwhile, we've also encapsulated the OpenCL program creation and moved it in Util. Now we could create a program without all the boilerplate code, specifying mainly the device type.

After we had image data in a vector in RGBA format, we sent it to the GPU.

```
deviceInputImage = Image2D(context, CL_MEM_READ_ONLY, IMAGE_FORMAT, width, height, 0, imagePixels.data());
```

Unfortunately, on our device, this does not compile as we were always getting `CL_INVALID_HOST_PTR`. Luckily the Khronos Group (2007) explains when this error occurs:

"CL_INVALID_HOST_PTR if host_ptr is NULL and CL_MEM_USE_HOST_PTR or CL_MEM_COPY_HOST_PTR are set in flags ***or if host_ptr is not NULL but CL_MEM_COPY_HOST_PTR or CL_MEM_USE_HOST_PTR are not set in flags****". (Khronos Group, 2007)*

We were in the last case, so we only had to add the blamed flag:

```
deviceInputImage = Image2D(context, CL_MEM_READ_ONLY | CL_MEM_COPY_HOST_PTR, IMAGE_FORMAT, width, height, 0, imagePixels.data());
```

5.3 First Image2D

Now, we could reserve memory for images on the selected device. And it worked! When the selected device was the CL_DEVICE_TYPE_CPU, we could actually printf the values. This is when we started to work more on the kernel. For some unknown reason, we were obsessed about doing the blur using the Gaussian Kernel, instead of the average kernel (where basically all elements are 1). Probably too much reading about how to do the blur effect. So, we adapted some code and created a function to generate a Gaussian Kernel based on a radius and a sigma parameter and saved its pointer.

Because using convolution kernels accesses the neighbours of a pixels, we had to consider what we do when the suspect pixel is on the edge of the image. To solve this easily we created a sampler and send that to the device as well (along with the buffer for the Gaussian filter):

```
deviceSampler = Sampler(context, CL_FALSE, CL_ADDRESS_CLAMP_TO_EDGE, CL_FILTER_NEAREST);

deviceGrid = Buffer(context, CL_MEM_READ_ONLY | CL_MEM_COPY_HOST_PTR, sizeOfGrid, grid, &status);
```

Next, we sent the values for alfa, beta, gamma, radius, and sigma. We sent these as arguments, and not as compiler flags (even though it would have been faster), because we had in mind the interactivity feature that we wanted to add, so with compiler flags we would have needed to recompile the kernel at every frame, which would not be convenient, nor fast. So, the modifiers were sent as kernel arguments.

5.4 The kernel signature

Finally, we have the first complete kernel signature, with all the parameters sent to the device:

```
__kernel void gaussian(
                   __read_only        image2d_t input,
                   __write_only       image2d_t output,
                                      sampler_t sampler,
                   __constant         float*    grid,
                                      int       radius,
                                      float     alfa,
                                      float     beta,
                                      float     gama
)
```

Obviously, the input image is read_only, the output image is used to save the sharpened result, so we'll only write to that, the sampler has no qualifier because it complains if we add any, and the Gaussian grid is read-only for primitive types, so it is __constant and cached in the private memory. The rest of the values are changeable in the interactive mode, so no other qualifiers are needed.

5.5 The kernel logic

At this point, we were ready to write the kernel logic. Basically, the Gaussian Kernel is similar to the Average Kernel, but the neighbour pixels are weighted with their corresponding value from the Gaussian grid. In the Average Kernel, that value would be always 1.

So, after we get the offset position of the pixel to be processed, we compute the weighted sum of the neighbours and divide it by the number of participating pixels: the grid's size = (radius x 2 +1) x (radius x 2 +1).

```
const int x = get_global_id(0);
const int y = get_global_id(1);
const float4 ALFA = alfa;
const float4 BETA = beta;
const float4 GAMA = gama;
float4 sum = 0.0f;
uint index = 0;
int2 position;
for (int row = -radius; row <= radius; row++) {
         position.y = y + row;
         for (int column = -radius; column <= radius; column++) {
                  position.x = x + column;
                  sum += convert_float4(read_imageui(input, sampler, position)) * grid[index++];
         }
}
sum = sum / ((radius * 2 + 1)*(radius * 2 + 1));
```

At the end, we get the current value of the processed pixel, we add the weighted sum and save it, with the same position, on the output image.

```
position.x = x;
position.y = y;
const uint4 original = read_imageui(input, sampler, position);
const uint4 sharp = convert_uint4(convert_float4(original) * ALFA + sum * BETA + GAMA);
write_imageui(output, position, sharp);
```

5.6 The *other* kernel logic

The Average Kernel does not need the filter, nor the index. So, the highlighted text need to be removed:

```
__kernel void gaussian(
                   __read_only        image2d_t input,
                   __write_only       image2d_t output,
                                      sampler_t sampler,
                   __constant         float*    grid,
................................................................................
uint index = 0;
................................................................................
                   sum += convert_float4(read_imageui(input, sampler, position)) * grid[index++];
```

Also, when setting the kernel arguments for the Average Kernel, we need to skip setting the pointer for the Gaussian grid. Both kernels work, so we finally have an output image from the device. Now that everything is working, we need to encapsulate the functionality, so the Sharper class has been created and it is built based on the arguments passed from the command line. Of course, if these are missing, there are default values to instantiate the class.

5.7 The benchmark tools

When this step was completed, we prepared for measuring the results. We took a function that measures the execution time of a function from previous project and enriched it to create average time, by adding number of executions and some logging:

```
Time averageMeasure(bool verbose, unsigned int times, M method, P&&... parameters)
```

Also, the logStats function was reused with minor touching related to formatting of the headers and data.

```
bool logStats(string statsFileName, const vector<Time> stats, unsigned int cycles, TimeUnit unit = NANOSECOND)
```

Then we start measuring the execution times. Because many scenarios were passionately tested, we created command line arguments that can trigger the benchmarks and set other application settings. This lead to the creation of a primitive arguments parser that returns a Settings struct which holds default values for the Sharper class:

```
struct Settings {
         cl_device_type deviceType = CL_DEVICE_TYPE_GPU;
         string imageFileName = "lena.ppm";
         string outputFileName;
         Save what = SHARP;
         bool interactive = false;
         bool verbose = false;
         bool showHelp = false;
         bool saveToDisk = true;
         bool useAverageKernel = true;
         unsigned int radius = 3;
         unsigned int cycles = 1;
         unsigned int maxRadii = 0;
         float alfa = 1.5f;
         float beta = -0.5f;
         float gama = 0.0f;
         float sigma = 1.0f;
};
```

5.8 The glut refresh

Now that we had the results (presented further in Chapter 6), we looked over past projects where OpenGL, glut, and glew were used (Lighthouse3d.com, 2015), and successfully rendered, after a while, our first OpenGL window, white triangle, and a black background.

```
void runGL(int argc, char **argv, int width, int height) {
	glutInit(&argc, argv);
	glutInitDisplayMode(GLUT_DOUBLE | GLUT_RGBA);
	glutInitWindowPosition(0, 0);
	glutInitWindowSize(width, height);
	windowID = glutCreateWindow("Sharp");
	glutKeyboardFunc(press);
	glutKeyboardUpFunc(release);
	glutDisplayFunc(GLrender);
	glutMainLoop();
}
```

5.9 To share or not to share

Existing example code (Enjalot, 2010; forums.khronos.org, 2011; Lake, & Robert, 2014; Shevtsov, 2014) for combining OpenGL and OpenCL functionality always recommends using a shared context, and to make use of information associated with the OpenGL context through **cl_khr_gl_sharing** extension. We did not manage to find any attempt, where the OpenCL result is simply returned to the Host and bound directly to a texture to be used in OpenGL. So, we thought that we should try it.

Our Sharpen class now has a GLuint texture handle that we update with every keypress, when OpenCL queue finishes:

```
void Sharper::run()
{
	queue.enqueueNDRangeKernel(kernel, NullRange, NDRange(width, height));
	queue.finish();
}
```

In the texture getter, we read it from the device before we return it to the caller:

```
GLuint Sharper::getTexture()
{
	queue.enqueueReadImage(deviceOutputImage, CL_BLOCKING, origin, region, 0, 0, imagePixels.data());
	glGenTextures(1, &texture);
	glBindTexture(GL_TEXTURE_2D, texture);
	glTexEnvf(GL_TEXTURE_ENV, GL_TEXTURE_ENV_MODE, GL_MODULATE);
	glTexParameterf(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR_MIPMAP_NEAREST);
	glTexParameterf(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR);
	glTexParameterf(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_REPEAT);
	glTexParameterf(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_REPEAT);
	gluBuild2DMipmaps(GL_TEXTURE_2D, 4, width, height, GL_RGBA, GL_UNSIGNED_BYTE, imagePixels.data());

	return texture;
}
```

This is being called in the render function, immediately after we enable GL_TEXTURE_2D and prepare to draw a textured quad:

```
glEnable(GL_TEXTURE_2D); {
	// bind texture
	glBindTexture(GL_TEXTURE_2D, sharper.getTexture());
	// draw textured quad
	glBegin(GL_QUADS); {
		glTexCoord2f(0.0f, 0.0f); glVertex2f(0.0f, 0.0f);
		glTexCoord2f(0.0f, 1.0f); glVertex2f(0.0f, 1.0f);
		glTexCoord2f(1.0f, 1.0f); glVertex2f(1.0f, 1.0f);
		glTexCoord2f(1.0f, 0.0f); glVertex2f(1.0f, 0.0f);
	}
	glEnd();
}
glDisable(GL_TEXTURE_2D);
```

The output is a resizable window with the same size as the loaded image, visible in the following Figure:

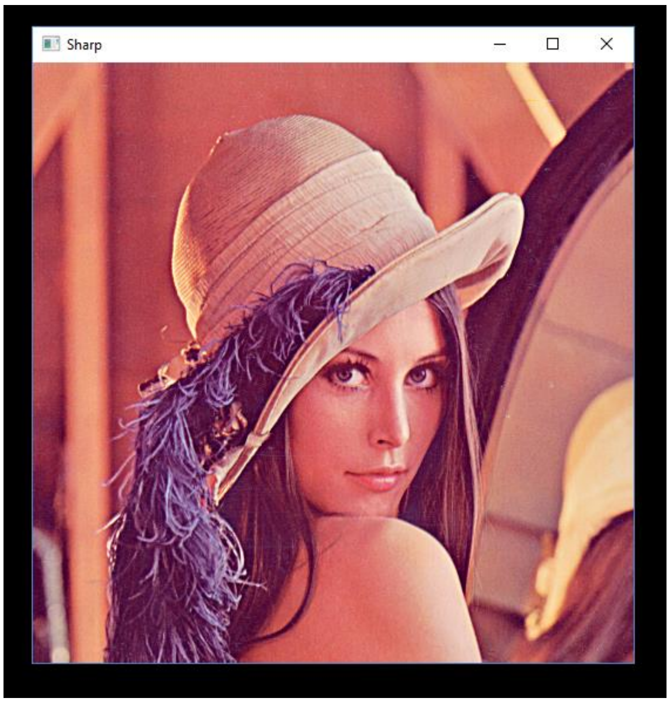


Figure 4. Window output using Glut

5.10 Let's move it!

The only thing left to do at this stage is to apply variations to the output image in the GL window. We have defined the `glutKeyboardFunc(press)` and the `glutKeyboardUpFunc(release).` In these functions, after each meaningful key press or key release the Sharper class updates the kernel arguments using this function call:

```
sharp.update(true, radius, alfa, beta, gama, sigma);
```

In the update, the class will decide if the arguments should be updated in the Average Kernel or in the Gaussian Kernel:

```
void Sharper::update(unsigned int r, float a, float b, float g, float s)
{
         radius = r;
         alfa = a;
         beta = b;
         gama = g;
         sigma = s;
         useAverageKernel ? updateAverageKernel() : updateGaussianKernel();
}
```

The difference between the two updates is that the Gaussian one also creates a buffer for the Gaussian grid:

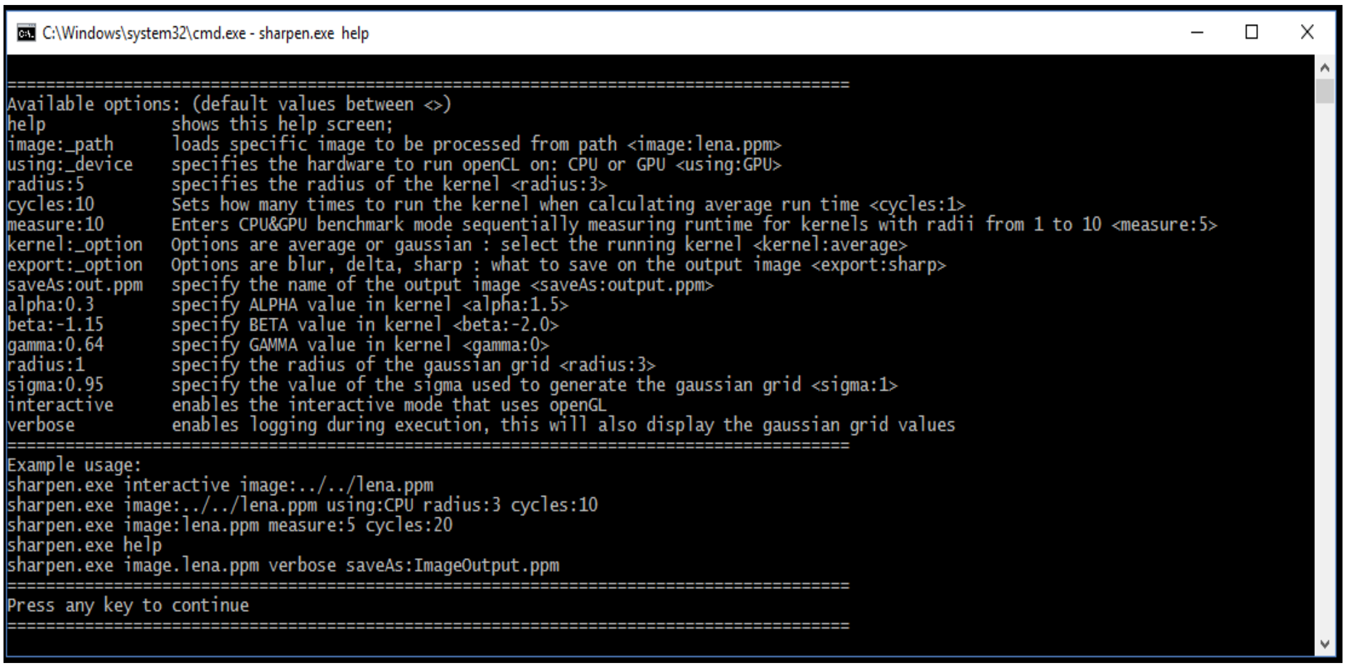
```
void Sharper::updateGaussianKernel() {
         grid = getGaussianFilter(logging, radius, sigma);
         unsigned int sizeOfGrid = (radius * 2 + 1) * (radius * 2 + 1) * sizeof(float);
         deviceGrid = Buffer(context, CL_MEM_READ_ONLY | CL_MEM_COPY_HOST_PTR, sizeOfGrid, grid, &status);
         onError(status, "Could not create Buffer for device grid", QUIT);

         onError(kernel.setArg(0, deviceInputImage), "Could not set argument on kernel for variable input", QUIT);
         onError(kernel.setArg(1, deviceOutputImage), "Could not set argument on kernel for variable output", QUIT);
         onError(kernel.setArg(2, deviceSampler), "Could not set argument on kernel for variable sampler", QUIT);
         onError(kernel.setArg(3, deviceGrid), "Could not set argument on kernel for variable mask ", QUIT);
         onError(kernel.setArg(4, sizeof(int), &radius), "Could not set the argument on kernel for variable radius", QUIT);
         onError(kernel.setArg(5, sizeof(float), &alfa), "Could not set argument on kernel for variable alfa", QUIT);
         onError(kernel.setArg(6, sizeof(float), &beta), "Could not set argument on kernel for variable beta", QUIT);
         onError(kernel.setArg(7, sizeof(float), &gama), "Could not set argument on kernel for variable gama", QUIT);
}
```

5.11 Help me up!

With all these commands, functions, parameters and arguments, we had to create some help messages to describe the usage. These are the hints that the application displays:

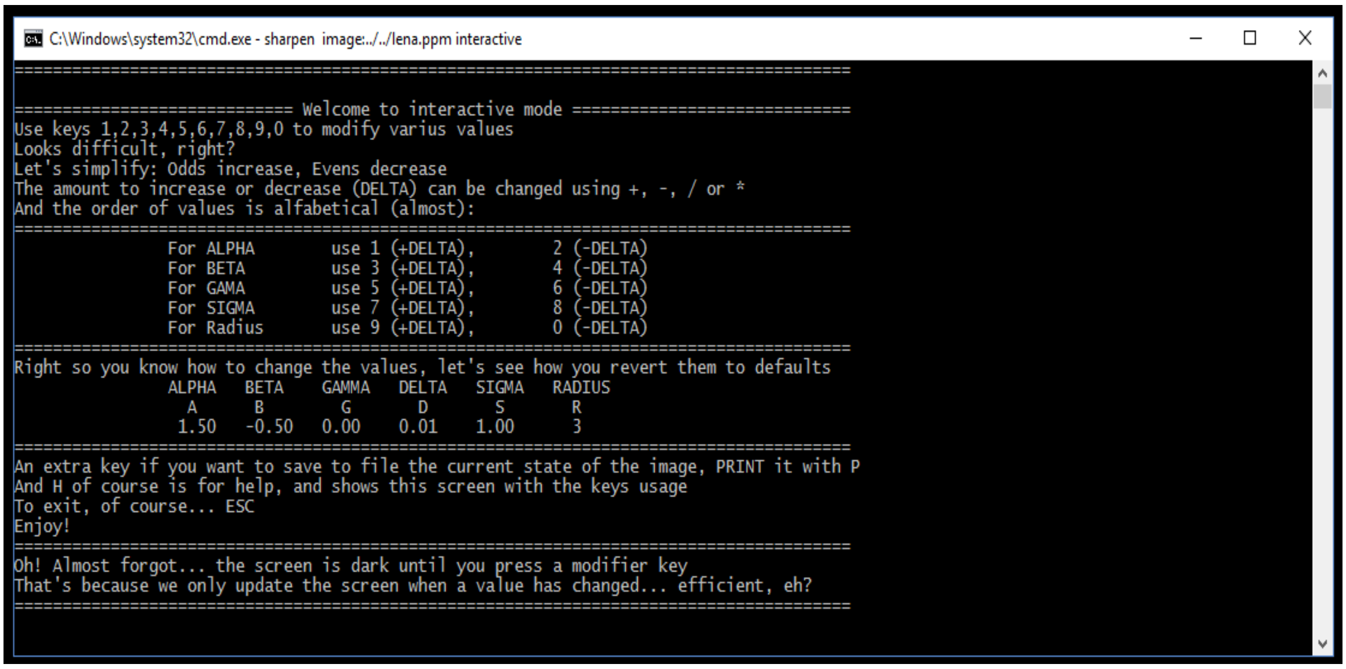
The Usage help



```
=====================================================================================
Available options: (default values between <>)
help            shows this help screen;
image:_path     loads specific image to be processed from path <image:lena.ppm>
using:_device   specifies the hardware to run openCL on: CPU or GPU <using:GPU>
radius:5        specifies the radius of the kernel <radius:3>
cycles:10       Sets how many times to run the kernel when calculating average run time <cycles:1>
measure:10      Enters CPU&GPU benchmark mode sequentially measuring runtime for kernels with radii from 1 to 10 <measure:5>
kernel:_option  Options are average or gaussian : select the running kernel <kernel:average>
export:_option  Options are blur, delta, sharp : what to save on the output image <export:sharp>
saveAs:out.ppm  specify the name of the output image <saveAs:output.ppm>
alpha:0.3       specify ALPHA value in kernel <alpha:1.5>
beta:-1.15      specify BETA value in kernel <beta:-2.0>
gamma:0.64      specify GAMMA value in kernel <gamma:0>
radius:1        specify the radius of the gaussian grid <radius:3>
sigma:0.95      specify the value of the sigma used to generate the gaussian grid <sigma:1>
interactive     enables the interactive mode that uses openGL
verbose         enables logging during execution, this will also display the gaussian grid values
=====================================================================================
Example usage:
sharpen.exe interactive image:../../lena.ppm
sharpen.exe image:../../lena.ppm using:CPU radius:3 cycles:10
sharpen.exe image:lena.ppm measure:5 cycles:20
sharpen.exe help
sharpen.exe image.lena.ppm verbose saveAs:ImageOutput.ppm
=====================================================================================
Press any key to continue
=====================================================================================
```

Figure 5. Usage help screen. To arrive here run: sharpen.exe help

The interactive help


C:\Windows\system32\cmd.exe - sharpen image:../../lena.ppm interactive

```
=====================================================================================

============================ Welcome to interactive mode ============================
Use keys 1,2,3,4,5,6,7,8,9,0 to modify varius values
Looks difficult, right?
Let's simplify: Odds increase, Evens decrease
The amount to increase or decrease (DELTA) can be changed using +, -, / or *
And the order of values is alfabetical (almost):
=====================================================================================
                For ALPHA       use 1 (+DELTA),       2 (-DELTA)
                For BETA        use 3 (+DELTA),       4 (-DELTA)
                For GAMA        use 5 (+DELTA),       6 (-DELTA)
                For SIGMA       use 7 (+DELTA),       8 (-DELTA)
                For Radius      use 9 (+DELTA),       0 (-DELTA)
=====================================================================================
Right so you know how to change the values, let's see how you revert them to defaults
                ALPHA   BETA    GAMMA   DELTA   SIGMA   RADIUS
                  A       B       G       D       S       R
                 1.50   -0.50   0.00    0.01    1.00      3
=====================================================================================
An extra key if you want to save to file the current state of the image, PRINT it with P
And H of course is for help, and shows this screen with the keys usage
To exit, of course... ESC
Enjoy!
=====================================================================================
Oh! Almost forgot... the screen is dark until you press a modifier key
That's because we only update the screen when a value has changed... efficient, eh?
=====================================================================================
```

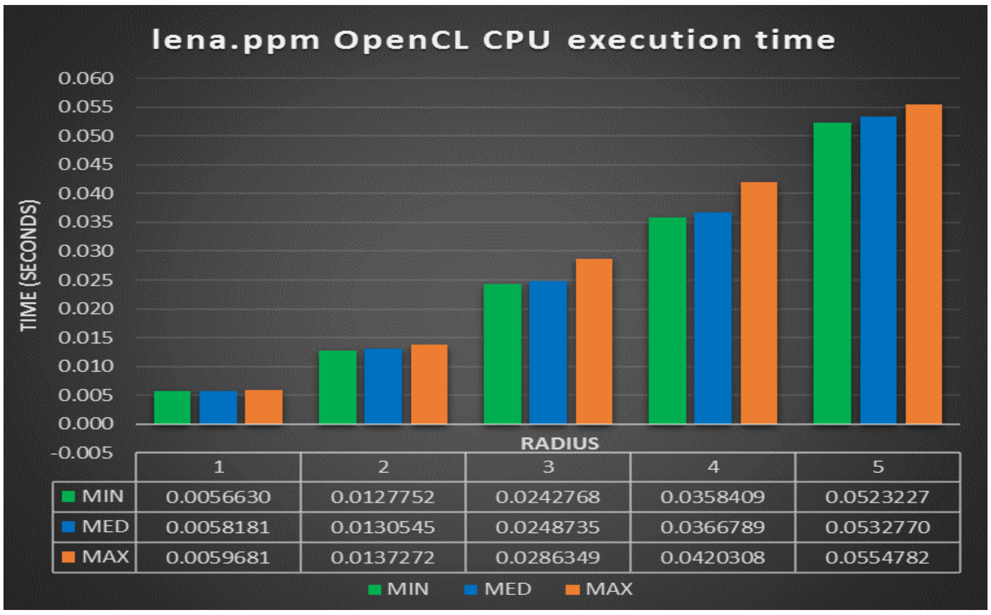
Figure 6. Interactive screen. To arrive here run: sharpen.exe interactive

6. Benchmark results

To measure the effectiveness of the OpenCL approach, we need to run the following command line:

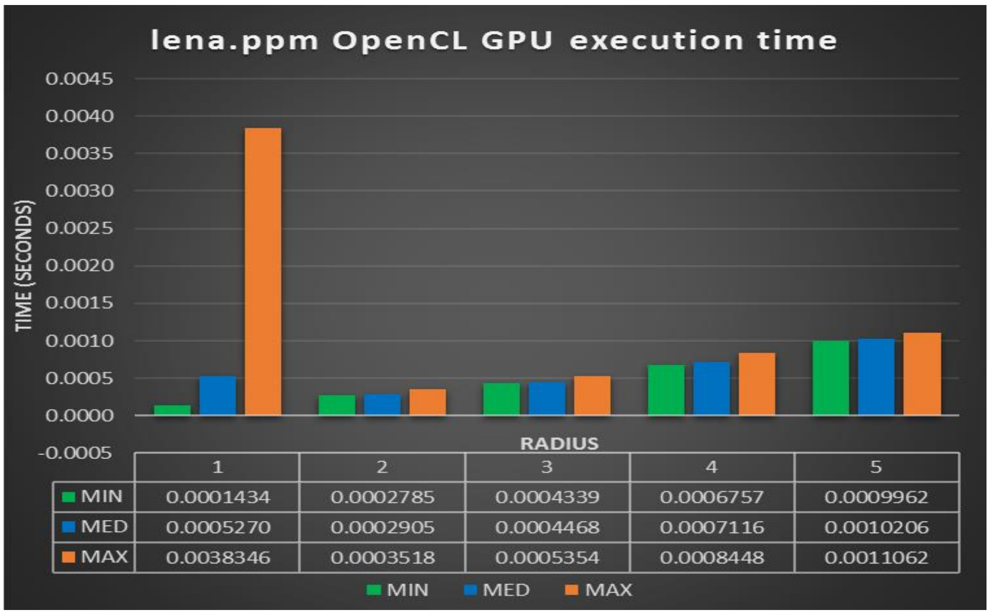
```
sharpen lena.ppm measure:5 cycles:10
```

This will create two Sharp instances, one to run in parallel on CPU and one on GPU. The *measure* argument will run the instances for radii between 1 to 5, and each instance will run 10 times to calculate an average time, a fastest time and a slowest execution time. The results are saved in two separate csv files: CPU-times.csv and GPU-times.csv. The following results are generated after running benchmarks on lena.ppm. For larger images, it is recommended to also add *verbose* argument, to see progress of the application (the images will take longer to load and will give the impression of a frozen application).



	1	2	3	4	5
🟩 MIN	0.0056630	0.0127752	0.0242768	0.0358409	0.0523227
🟦 MED	0.0058181	0.0130545	0.0248735	0.0366789	0.0532770
🟧 MAX	0.0059681	0.0137272	0.0286349	0.0420308	0.0554782

Figure 7. OpenCL CPU run time for different radii. Average calculated for 10 cycles using lena.ppm



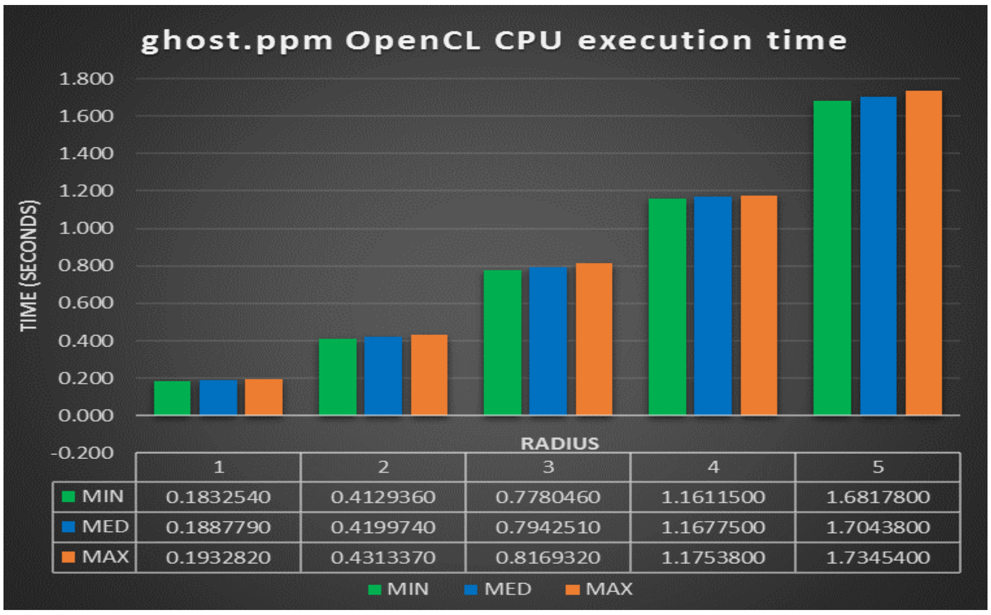
	1	2	3	4	5
🟩 MIN	0.0001434	0.0002785	0.0004339	0.0006757	0.0009962
🟦 MED	0.0005270	0.0002905	0.0004468	0.0007116	0.0010206
🟧 MAX	0.0038346	0.0003518	0.0005354	0.0008448	0.0011062

Figure 8. OpenCL GPU run time for different radii. Average calculated for 10 cycles using lena.ppm

The improvement brought by OpenCL is noticeable in the execution time when sharpening a 512x512 image. Compared to the CPU-serial execution time, the CPU-CL code is ~4 times faster for radius 1, ~6 times faster for radius 2, ~7 times faster for radius 3, ~8 times faster for radius 4 and ~9 times faster for radius 5.

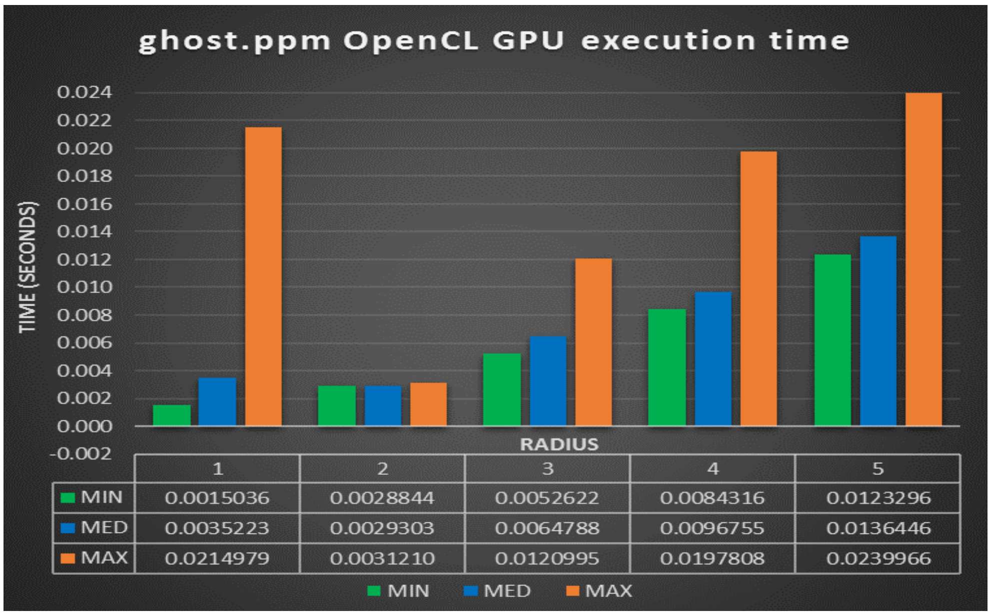
On the other hand, the GPU-CL code is ~50 times faster for radius 1, ~250 times for radius 2, ~400 times for radius 3, ~450 times for radius 4, ~500 times for radius 5.

Between the two OpenCL implementations, for a 512x512 image, the GPU-CL performance is roughly 40 times faster than the CPU-CL for radius 1, and roughly 50 times for the other radii.



RADIUS	1	2	3	4	5
🟩 MIN	0.1832540	0.4129360	0.7780460	1.1611500	1.6817800
🟦 MED	0.1887790	0.4199740	0.7942510	1.1677500	1.7043800
🟧 MAX	0.1932820	0.4313370	0.8169320	1.1753800	1.7345400

Figure 9. OpenCL CPU run time for different radii. Average calculated for 10 cycles using ghost.ppm



RADIUS	1	2	3	4	5
🟩 MIN	0.0015036	0.0028844	0.0052622	0.0084316	0.0123296
🟦 MED	0.0035223	0.0029303	0.0064788	0.0096755	0.0136446
🟧 MAX	0.0214979	0.0031210	0.0120995	0.0197808	0.0239966

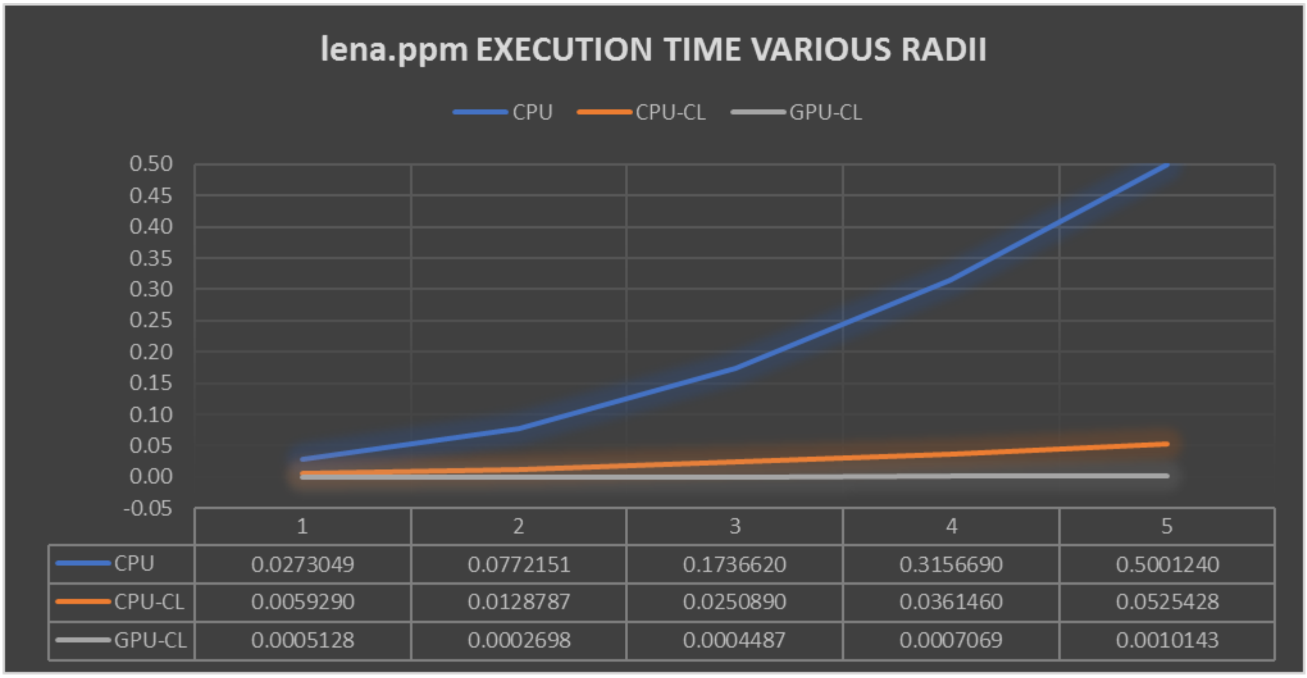
Figure 10. OpenCL CPU run time for different radii. Average calculated for 10 cycles using ghost.ppm

When comparing the execution times for sharpening a bigger, 3840x2160 image the results are interesting. Compared to the CPU-serial execution time, the CPU-CL has a similar performance to the one obtained when sharpening a smaller image.

On the other hand, the GPU-CL code is ~250 times faster for radius 1, ~850 times for radius 2, ~860 times for radius 3, ~1050 times for radius 4, ~1200 times for radius 5.

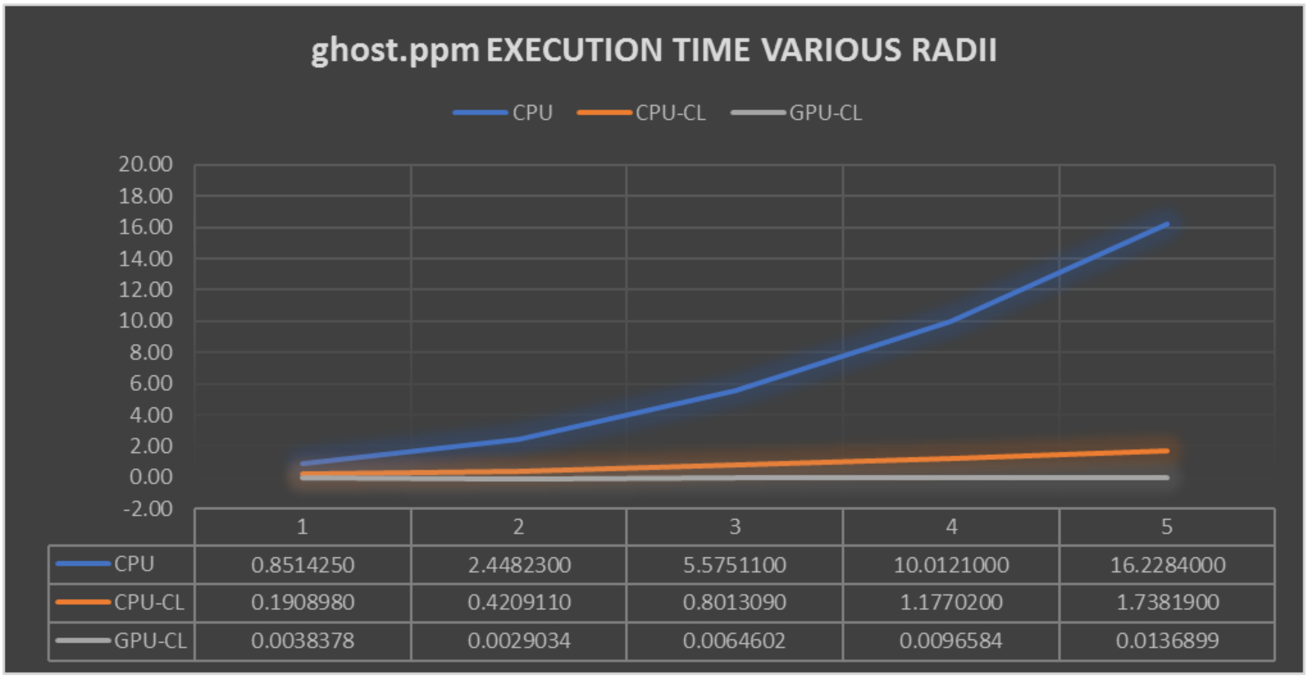
Between the two OpenCL implementations, for a 3840x2160 image, the GPU-CL performance is roughly 50 times faster than the CPU-CL for radius 1, and roughly 140 times for the other radii.

We can observe then that the GPU-CL performance compared to the CPU-CL one increases with the amount of work to be done. So, for higher images, GPU-CL gets better and better compared to the CPU-CL and the GPU-CL execution time is insignificant compared to the CPU-serial running time. The difference is just too high.



	1	2	3	4	5
CPU	0.0273049	0.0772151	0.1736620	0.3156690	0.5001240
CPU-CL	0.0059290	0.0128787	0.0250890	0.0361460	0.0525428
GPU-CL	0.0005128	0.0002698	0.0004487	0.0007069	0.0010143

Figure 11. Comparison of execution time (seconds) between serial CPU, CPU-CL and GPU-CL for lena.ppm



	1	2	3	4	5
CPU	0.8514250	2.4482300	5.5751100	10.0121000	16.2284000
CPU-CL	0.1908980	0.4209110	0.8013090	1.1770200	1.7381900
GPU-CL	0.0038378	0.0029034	0.0064602	0.0096584	0.0136899

Figure 12. Comparison of execution time (seconds) between serial CPU, CPU-CL and GPU-CL for ghost.ppm

From Figure 11 and Figure 12 we can see that the execution time is directly proportional with the number of pixels it has to process, but the performance ratios between the three approaches do not vary significantly and do not depend on the image size.

7. The interactive mode

The interactive mode is enabled by adding the argument interactive when executing the application. By default, the kernel used in this mode is the Gaussian Kernel, because we have more control over the convolution kernel by altering the sigma value too, not only the radius. For example, a 3x3 normalized kernel with sigma 1 has different kernel values compared to a 3x3 unnormalized kernel. Also, these values change when sigma changes, as we can notice in the following figure:

3x3 Kernel - unnormalised

Radius	1	
Sigma	1.00	
0.1467627	0.2419707	0.1467627
0.2419707	0.3989423	0.2419707
0.1467627	0.2419707	0.1467627

3x3 Kernel - normalised

Radius	1	
Sigma	1.00	
0.0751136	0.1238414	0.0751136
0.1238414	0.2041800	0.1238414
0.0751136	0.1238414	0.0751136

3x3 Kernel - unnormalised

Radius	1	
Sigma	20.00	
0.0198973	0.0199222	0.0198973
0.0199222	0.0199471	0.0199222
0.0198973	0.0199222	0.0198973

3x3 Kernel - normalised

Radius	1	
Sigma	20.00	
0.1110185	0.1111574	0.1110185
0.1111574	0.1112964	0.1111574
0.1110185	0.1111574	0.1110185

Figure 13. Radius 1 Kernel with different settings

In a normalized kernel, the sum of all values is 1 (or very close to 1). With a sigma of 1, the neighbour pixels will have the highest weight. A higher radius will spread the weight values towards the marginal pixels. With a sigma less than one, the weights will go towards 0, increasing the value of the central pixel. This means that the blur effect fades away. With a sigma higher than 1, the higher the sigma, the more even the weights.

To see the normalised values of the Gaussian Kernel, run the application with the following arguments:

```
sharpen image:lena.ppm kernel:gaussianNormalised sigma:20 radius:1 verbose
```

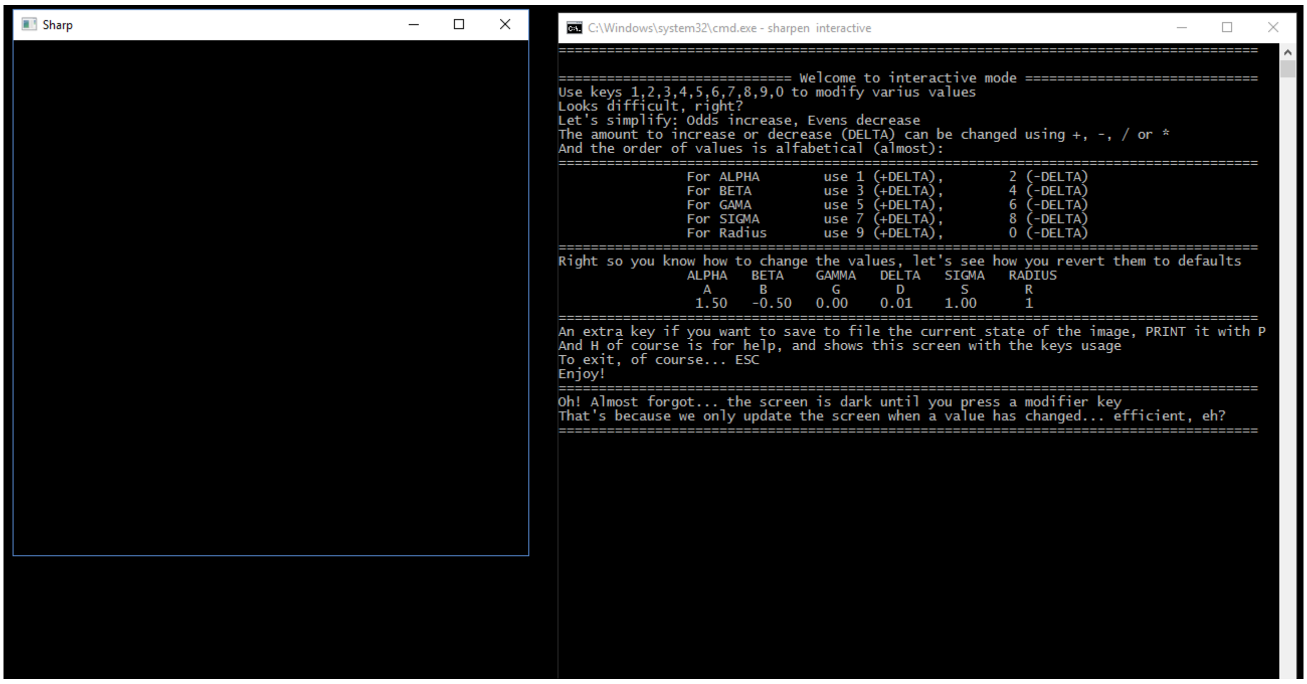
or for the unnormalized kernel run with the appropriate kernel argument:

```
sharpen image:lena.ppm kernel:gaussian sigma:20 radius:1 verbose
```

Let's start the application in interactive mode:

```
sharpen image:lena.ppm interactive
```

The application will display text in console, and image output in a new black window like in figure 14. The reason the windows is black is because we update the texture only when some variable changes, and this change comes from the user input.



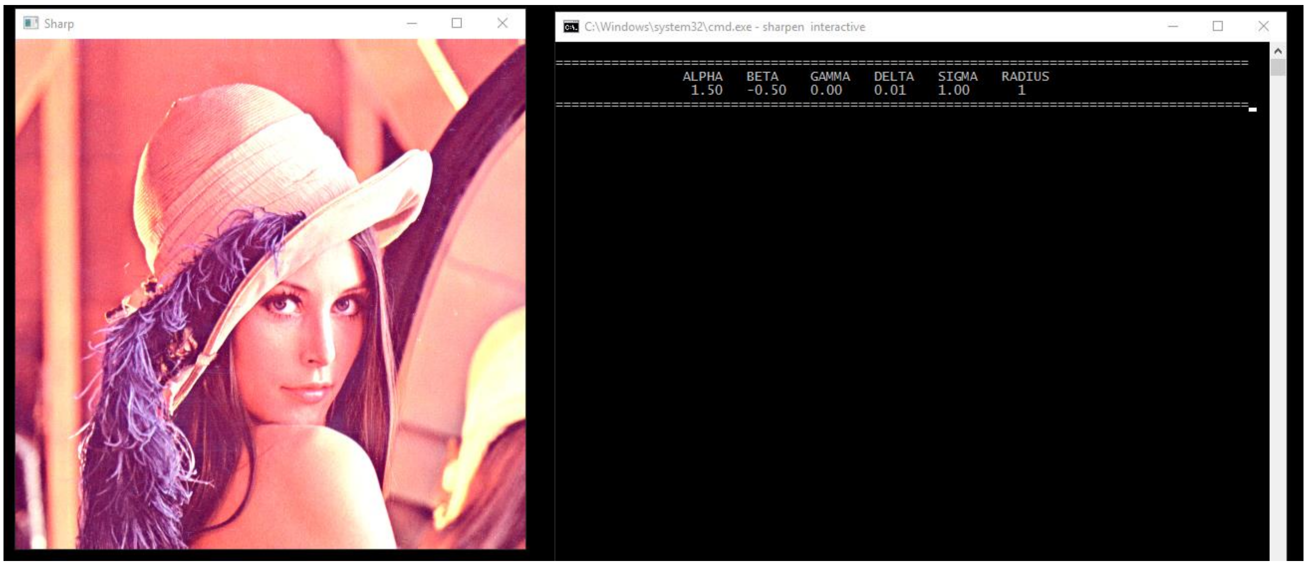
```
================================================================================
========================== Welcome to interactive mode ==========================
Use keys 1,2,3,4,5,6,7,8,9,0 to modify varius values
Looks difficult, right?
Let's simplify: Odds increase, Evens decrease
The amount to increase or decrease (DELTA) can be changed using +, -, / or *
And the order of values is alfabetical (almost):
================================================================================
                 For ALPHA        use 1 (+DELTA),        2 (-DELTA)
                 For BETA         use 3 (+DELTA),        4 (-DELTA)
                 For GAMA         use 5 (+DELTA),        6 (-DELTA)
                 For SIGMA        use 7 (+DELTA),        8 (-DELTA)
                 For Radius       use 9 (+DELTA),        0 (-DELTA)
================================================================================
Right so you know how to change the values, let's see how you revert them to defaults
                 ALPHA    BETA    GAMMA    DELTA    SIGMA    RADIUS
                   A        B       G        D        S        R
                  1.50   -0.50    0.00     0.01     1.00       1
================================================================================
An extra key if you want to save to file the current state of the image, PRINT it with P
And H of course is for help, and shows this screen with the keys usage
To exit, of course... ESC
Enjoy!
================================================================================
Oh! Almost forgot... the screen is dark until you press a modifier key
That's because we only update the screen when a value has changed... efficient, eh?
================================================================================
```

Figure 14. Initial view in interactive mode

To apply modifiers for ALPHA, BETA, GAMMA, DELTA, SIGMA or RADIUS use the 0-9 keys and mathematical operators to manipulate DELTA. DELTA is the value added or subtracted from ALPHA, BETA, GAMA, SIGMA variables. Here is how the operators change the DELTA:

- Pressing + will make the variable delta +=0.1f
- Pressing - will make the variable delta -=0.1f
- Pressing * will make the variable delta *=2
- Pressing / will make the variable delta /=2f

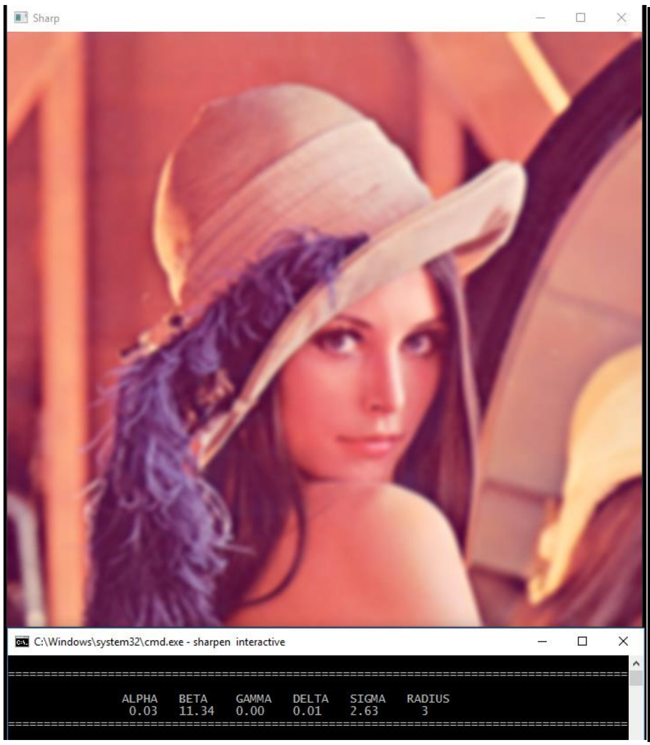
Once a variation is recorded, the image will be displayed. The default image uses ALPHA = 1.5, BETA = -0.5, GAMMA = 0, SIGMA = 1 and RADIUS 1. Also, the console will now show the current value of the variables, as visible in Figure 15.



```
================================================================================
                 ALPHA    BETA    GAMMA    DELTA    SIGMA    RADIUS
                  1.50   -0.50    0.00     0.01     1.00       1
================================================================================
```

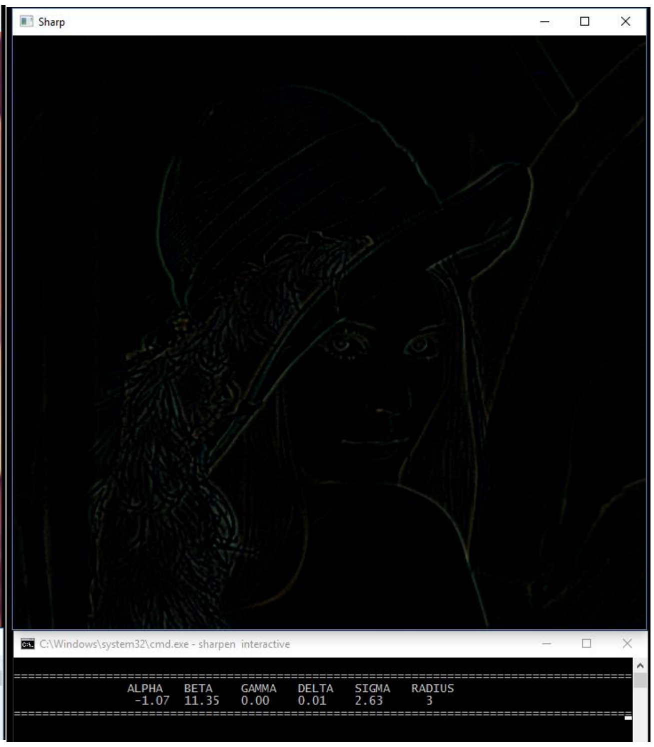
Figure 15. Active interactive mode

Tweaking these variables can output different image effects. These outputs do not represent the real effects, but a similar output:



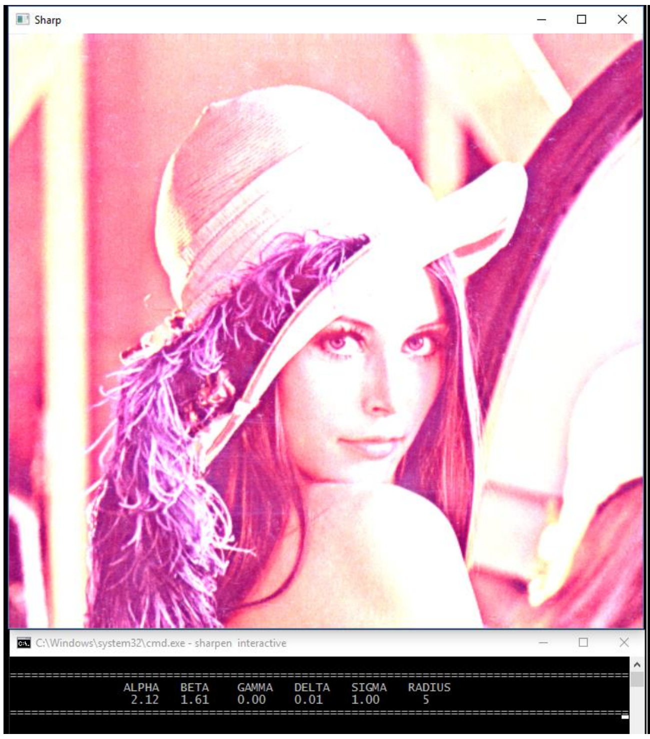
```
ALPHA   BETA    GAMMA   DELTA   SIGMA   RADIUS
 0.03   11.34   0.00    0.01    2.63      3
```

Blur



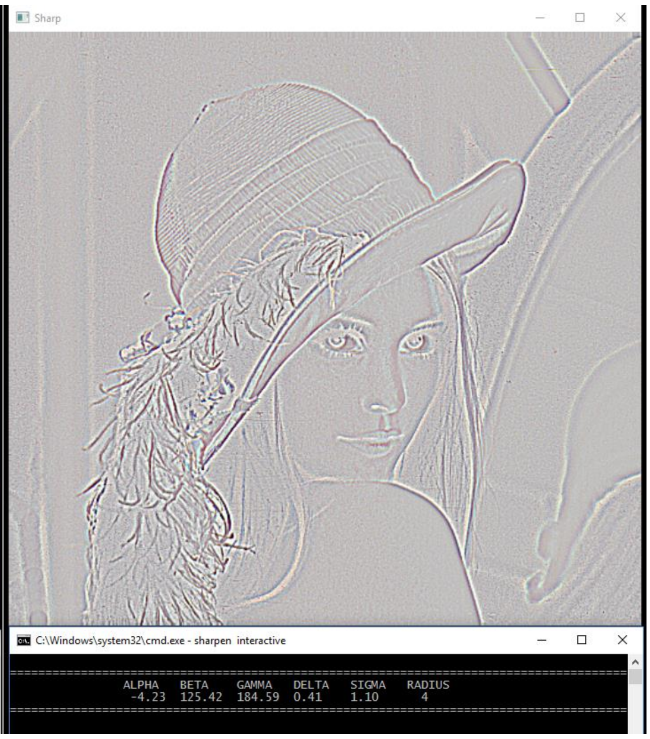
```
ALPHA   BETA    GAMMA   DELTA   SIGMA   RADIUS
-1.07   11.35   0.00    0.01    2.63      3
```

Edges



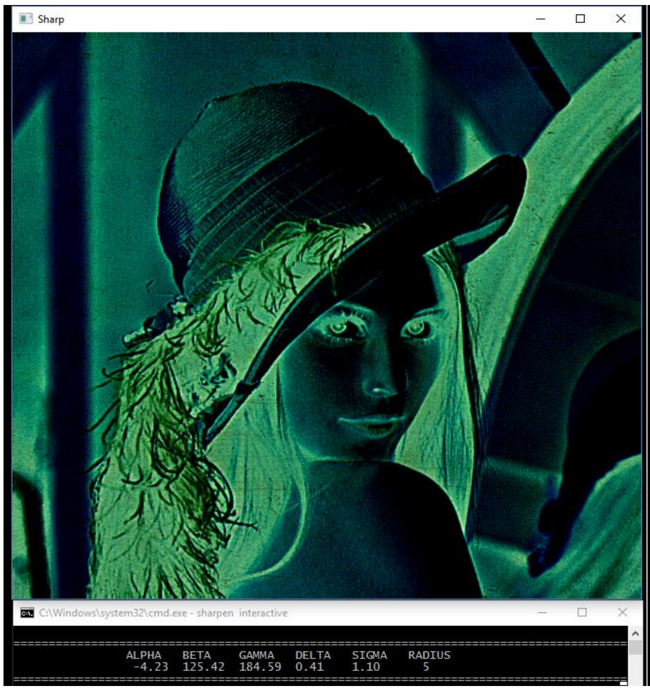
```
ALPHA   BETA    GAMMA   DELTA   SIGMA   RADIUS
 2.12   1.61    0.00    0.01    1.00      5
```

Old Photo



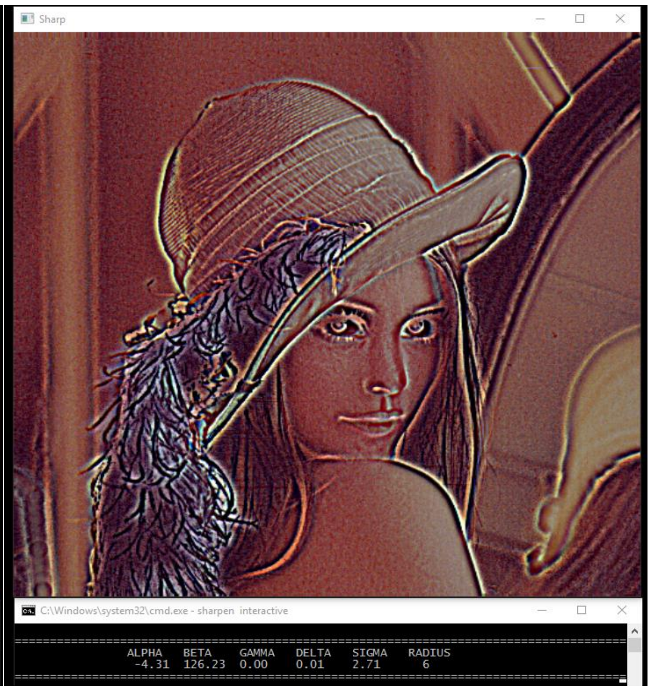
```
ALPHA   BETA    GAMMA   DELTA   SIGMA   RADIUS
-4.23   125.42  184.59  0.41    1.10      4
```

Emboss or Sketch



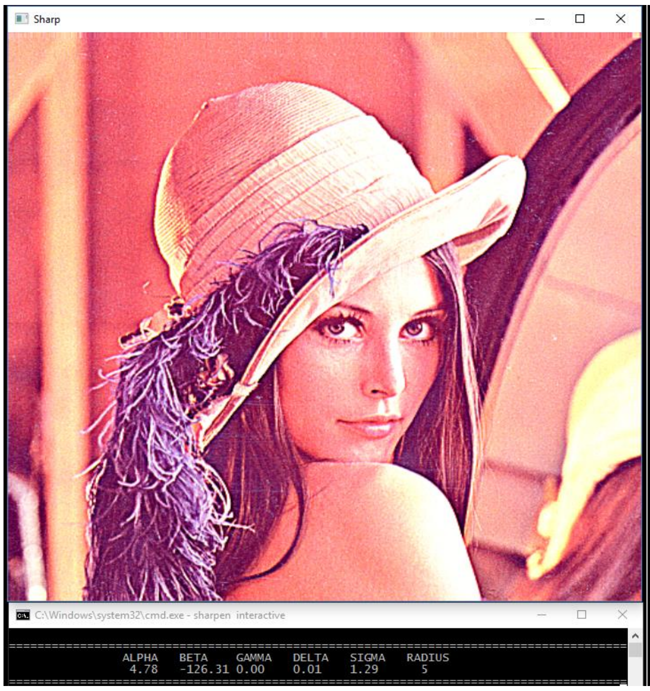
```
ALPHA   BETA    GAMMA   DELTA   SIGMA   RADIUS
 -4.23  125.42  184.59  0.41    1.10      5
```

Negative



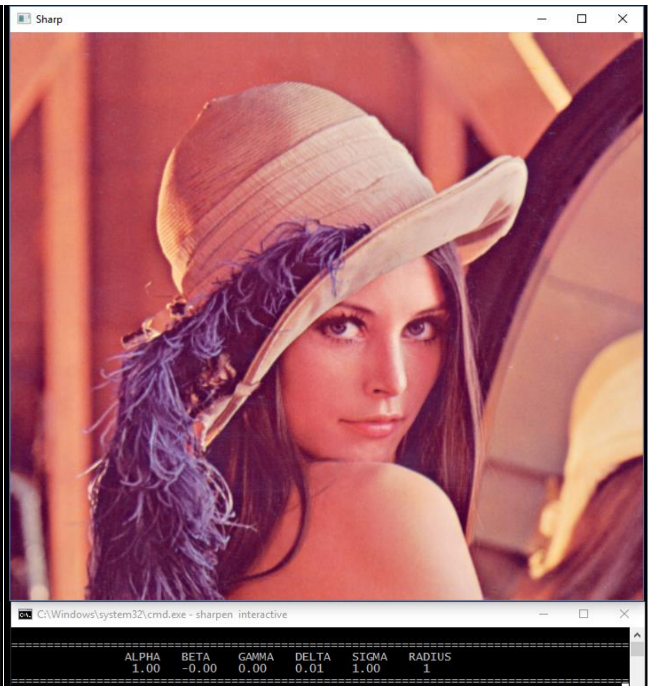
```
ALPHA   BETA    GAMMA   DELTA   SIGMA   RADIUS
 -4.31  126.23  0.00    0.01    2.71      6
```

Glow Edge Colour



```
ALPHA   BETA    GAMMA   DELTA   SIGMA   RADIUS
 4.78   -126.31 0.00    0.01    1.29      5
```

Sharp



```
ALPHA   BETA    GAMMA   DELTA   SIGMA   RADIUS
 1.00   -0.00   0.00    0.01    1.00      1
```

Original

To save any of these pictures, on their current state, just press P to **P**rint them to file. The Image will be saved in the same location as the executable as InteractiveEdit.ppm. To reset the values of the variables, press on their initial letter: **A**(lpha), **B**(eta), **G**(amma)**,** **D**(elta), **R**(adius)**,** **S**(igma). To display the interactive help menu, press **H**(elp). To exit the application, press **ESC**ape.

These images can be reproduced by running the application with the following arguments and adding the value from the console part of the image:

```
sharpen image:lena.ppm kernel:gaussian alpha:4.78 beta:-126.31 gamma:0 sigma:1.29 radius:5 unit:second
```

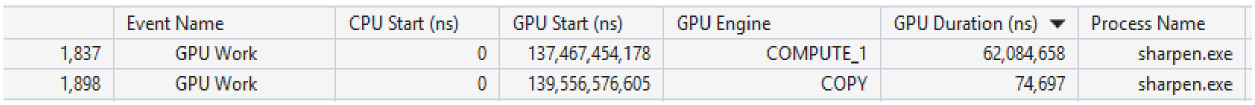
8. Conclusions

The serial code uses the average blur, and inverts the filter to obtain a sharp effect. Using a filter with default value (5) for radius, a 512x512 image was sharpened in 0.5 seconds, while a 3840x2160 image was sharpened in 16 seconds (32 times slower).

Using the OpenCL API, the application runs even 1200 times faster than the serial code. In general, the more pixels to process the higher the gap between the serial code and the GPU-CL approach. The timings are relative to the hardware they are running on, so it is better to measure the performance with percentage or mentioning the number of times.

The Average blur is faster than the Gaussian Blur and we can consider the average blur a particular case of Gaussian blur where all kernel elements are 1. But because the average blur does not multiply the pixels one by one with their grid correspondent, it saves execution time and it is faster.

The real-time application can work smoothly without using the cl_khr_gl_sharing extension, as long as the GPU COMPUTE time and GPU COPY time are not bigger than 40 milliseconds (25 updates per second). Even with 3840x2160 images, the application can work, but it requires approximately 62 milliseconds for a 7x7 kernel effect, yielding 16 frames per second, which is not very smooth, but still acceptable. This has been recorded in the profiler, where the event timing is more accurate (although not too far from the chrono high resolution clock).



	Event Name	CPU Start (ns)	GPU Start (ns)	GPU Engine	GPU Duration (ns)	Process Name
1,837	GPU Work	0	137,467,454,178	COMPUTE_1	62,084,658	sharpen.exe
1,898	GPU Work	0	139,556,576,605	COPY	74,697	sharpen.exe

In general, it is better to run with small kernels, as they are faster and the weight is not so spread across all elements. The effect is harder, sharper, so for a finer weight, higher radii are recommended, but this will come at a cost of performance. For the common images (not 4k), the sharpen interactive application works very well, even if it is using the Gaussian kernel.

We traded the performance of using a faster kernel (that could use compiler variables, local memory, or unrolled for loops if the radius would be constant) for a complete working application that returns the desired result and some extra effects. Even so, the included kernels (average and Gaussian) are fast enough and the obtained performance is more than satisfactory.

Overall OpenCL, like CUDA, brings the ultimate computing power and squeezes the best performance out of our General Purpose Graphic Processing Units. And with OpenCL, also from our CPUs.

Convention

In this report, we calculate the kernel number of elements by the formula:

$$NumberOfElements = (radius \times 2 + 1)^2$$

So, in our convention:

- a radius 1 kernel is a grid of 9 elements, a 3x3 kernel,
- a radius 2 kernel is a grid of 25 elements, a 5x5 kernel,
- a radius 3 kernel is a grid of 49 elements, a 7x7 kernel, et cetera.

Please use care and make sure you convert your notion to our kernel size.

References

Enjalot,. (2010). *Adventures in OpenCL Part 2: Particles with OpenGL | enjalot*. *Enja.org*. Retrieved 6 April 2017, from http://enja.org/2010/08/27/adventures-in-opencl-part-2-particles-with-opengl/

Fisher, R., Perkins, S., Walker, A., & Wolfart., E. (2013). *Glossary - Convolution*. *Homepages.inf.ed.ac.uk*. Retrieved 5 April 2017, from http://homepages.inf.ed.ac.uk/rbf/HIPR2/convolve.htm

forums.khronos.org,. (2011). *OpenCL - OpenGL 2D texture interop*. *Forums.khronos.org*. Retrieved 6 April 2017, from https://forums.khronos.org/showthread.php/7452-OpenCL-OpenGL-2D-texture-interop

Khronos Group,. (2007). *clCreateBuffer*. *Khronos.org*. Retrieved 6 April 2017, from https://www.khronos.org/registry/OpenCL/sdk/1.0/docs/man/xhtml/clCreateBuffer.html

Lake, A., & Robert, I. (2014). *Sharing Surfaces between OpenCL™ and OpenGL* 4.3 on Intel® Processor Graphics using implicit synchronization | Intel® Software*. *Software.intel.com*. Retrieved 6 April 2017, from https://software.intel.com/en-us/articles/sharing-surfaces-between-opencl-and-opengl-43-on-intel-processor-graphics-using-implicit

Lighthouse3d.com,. (2015). *Initialization*. *Lighthouse3d.com*. Retrieved 6 April 2017, from http://www.lighthouse3d.com/tutorials/glut-tutorial/initialization/

Selmar,. (2014). *Convenient way to show OpenCL error codes?*. *Stackoverflow.com*. Retrieved 6 April 2017, from http://stackoverflow.com/questions/24326432/convenient-way-to-show-opencl-error-codes

Shevtsov, M. (2014). *OpenCL™ and OpenGL* Interoperability Tutorial | Intel® Software*. *Software.intel.com*. Retrieved 6 April 2017, from https://software.intel.com/en-us/articles/opencl-and-opengl-interoperability-tutorial

Shillingford, W. (2016). *[OpenCL 1.2 C++ Tutorials 1/9] - What is OpenCL?*. *YouTube*. Retrieved 6 April 2017, from https://youtu.be/YU_pRT-Be0c

Other study materials:

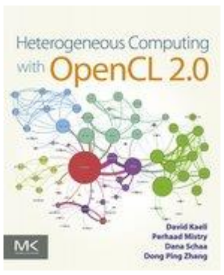
http://developer.amd.com/tools-and-sdks/opencl-zone/opencl-resources/programming-in-opencl/image-convolution-using-opencl/

https://www.khronos.org/registry/OpenCL/specs/opencl-cplusplus-1.1.pdf

From safaribooksonline.com:

OpenCL Programming by Example
By: Ravishekhar Banger; Koushik Bhattachary ya
Publisher: Packt Publishing
Pub. Date: December 23, 2013
Print ISBN-13: 978-1-84969-234-2
Web ISBN-13: 978-1-84969-235-9
Pages in Print Edition: 304
Subscriber Rating: [**0 Ratings**]

Heterogeneous Computing with OpenCL 2.0
By: David R. Kaeli; Perhaad Mistry; Dana Schaa; Dong Ping Zhang
Publisher: Morgan Kaufmann
Pub. Date: June 18, 2015
Web ISBN-13: 978-0-12-801649-7
Print ISBN-13: 978-0-12-801414-1
Pages in Print Edition: 330
Subscriber Rating: [**0 Ratings**]



OpenCL Parallel Programming Development Cookbook
By: Raymond Tay;
Publisher: Packt Publishing
Pub. Date: August 26, 2013
Print ISBN-13: 978-1-84969-452-0
Web ISBN-13: 978-1-84969-453-7
Pages in Print Edition: 302
Subscriber Rating: [**0 Ratings**]