Image Compression with Genetic Algorithms - Population Rendering

Libor Novak
novakli2@fel.cvut.cz

A4M39GPU final report, fall 2016/2017
https://github.com/libornovax/eoa_image_compression

***Abstract*—This semestral work tackles the problem of image compression using simple geometric shapes. Genetic optimization algorithms are used to find the best layout of simple geometric shapes, which minimizes the difference of the pixel colors of the prototype and the approximation image. The focus is put on the rendering of the approximation images (extraction of phenotypes) from the chromosomes and computing the fitness of the approximations. A sequential (CPU) and a parallel (GPU) solution to the fitness computation are presented and compared. The parallel GPU solution clearly wins the race approximately 10-fold.**

I. Problem description

Image compression is a task of approximation of an image with an image, which is as similar to the original (target) one as possible in terms of some evaluation function, while keeping a given compression rate. Say how well can we approximate an image if we only have 1% of the storage needed to store the raw original image.

A. Representation

To represent the approximation (compressed image) we chose to use a list of simple shapes - circles (the algorithm is anyway easily extensible to other simple shapes). The list of circles will be called a *chromosome* as we are using genetic optimization algorithms. Each circle is represented by its RGB color, opacity α, (x, y) center coordinates, and radius r, i.e. 7 integer numbers as shown in Fig. 1.

R	G	B	α	x	y	r

Fig. 1. Circle representation.

B. Quality assessment

In order to evaluate the quality of the compression we define a fitness function. The fitness function represents the channel-wise difference between pixel colors of the original O and the approximation A:

$$\sum_i \sum_{c \in \{R,G,B\}} (p_{ic}^O - p_{ic}^A)^2,$$

where i represents a pixel in an image and p_{ci}^* is the value of the c channel of the pixel i. However, while carrying out tests we discovered that the approximation lacks details, which are important for a human observe to recognize the object in the image. Therefore we added weights to the image - one set of weights is on detected edges and the other is user defined (see Fig. 3).

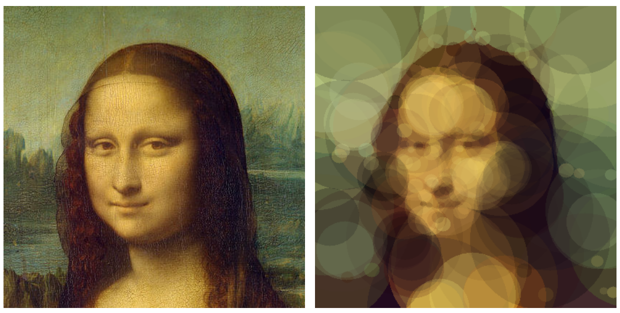
Fig. 2. The original image and its approximation with 200 circles.

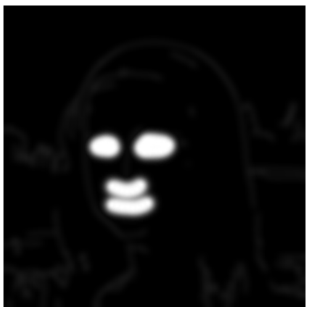
Fig. 3. The original image and its approximation with 200 circles.

The final used fitness function is therefore a weighted sum of squared differences over all pixels:

$$fitness = \sum_i \sum_{c \in \{R,G,B\}} w_i (p_{ic}^O - p_{ic}^A)^2,$$

where w_i is the weight of the difference in pixel i. Since the fitness function is a difference, it will be minimized.

C. *Optimization algorithms*

The goal of the optimization is to find the optimal positions and colors of the circles. The optimizer is randomly initialized with a given number of solutions with a given number of circles. Then, the properties of the circles are iteratively improving by applying changes, which improve the solution as much as possible. Such changes are explored by a population of candidate solutions, which are created by random perturbations - mutations. In total we explored three such algorithms, here we present only a brief, but their detailed descriptions can be found in [1], [2].

a) Steepest Ascent Hill Climber: Works with one best-so-far solution. In each iteration it generates a pool of candidate solutions by applying mutation, the whole population is evaluated and the best solution from the whole pool is chosen and replaces the current best-so-far solution if it improves upon it.

b) Steady-State Evolutionary Algorithm: Works with a population of solutions. In each epoch for each individual it chooses with tournament selection another individual for crossover. If the child, which emerged from the crossover is superior to the parent, then the parent is replaced by the child.

c) Hybrid Evolutionary Algorithm: For a given number of epochs it runs the steady-state evolutionary algorithm, which provides the optimization with crossover in between the individuals. Then, the optimization switches to hill climbing and each individual is separately evolved for a given number of epochs. These two types of optimization are interleaving during the whole evolution.

D. *Fitness computation*

This is the main focus of this report. It is apparent that in all of the described algorithms we will need to evaluate the fitness function millions of times in order to converge to an optimum. Therefore, the time it takes to evaluate the fitness of an individual or a population of individuals will be the main bottleneck of the optimization algorithm.

To evaluate the fitness of an individual we first need to render the whole image (phenotype) from the list of circles (chromosome) and compute the weighted sum of pixel differences from the target image. Essentially, we need to take the circles one by one and add them to the image. I considered two different approaches to the rendering from which each yields a slightly different result because it treats the transparency of the circles differently - see Fig. 4.

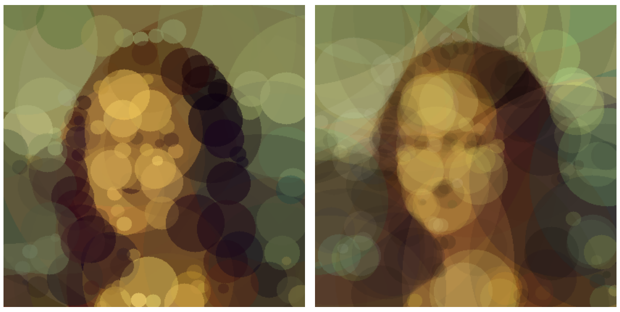
Fig. 4. Difference between rendering approaches - layered rendering (left), averaging (right).

d) Layered rendering: The image is constructed iteratively, meaning that when a circle with a given α is added to the canvas, the opacity considers only the so-far-rendered circles, which lie below.

e) Averaging: Or better weighted averaging considers the α channel as weight of the circle. One can imagine it in a way that we compute a weighted sum of contributions from all circles for each pixel.

As the rendering is the main concern of this report we will not be giving too many details on how the algorithms themselves are implemented and we will spend most of the time on rendering.

II. Sequential CPU Implementation

For the fitness computation on CPU we conveniently wanted to use the OpenCV library because it contains functions for drawing circles and other shape. Unfortunately, we realized that the drawing functions do not support drawing of transparent circles, each circle would thus have to be rendered into a separate canvas and the canvas' then added together. This would be very slow, and therefore we wrote our own rendering function, which supports transparency of circles.

As well as OpenCV, we used the midpoint circle algorithm [3], which can render 4 pixel rows of a circle at once. This makes it very fast and suitable for our purpose. Interestingly, one does not have to compute any distance from the center of the circle during the whole run of the algorithm as the circle is being cleverly plotted line by line, see Fig. 5 for code sample.

The `renderCircle()` function is then used to render circles into each color channel of the image approximation. The `Renderer` class takes care of rendering the whole chromosome. It is implemented as the visitor pattern because this way we can easily accommodate rendering of different shape types (as they all inherit from the same abstract class `IShape`).

```
void renderCircle (cv::Mat &image, const cv::Point &center, int radius, int alpha_color,
                   double alpha)
{
    // The plotting of the circle is done with the "Midpoint circle algorithm" as described
    // here: https://en.wikipedia.org/wiki/Midpoint_circle_algorithm. There are some
    // alternations to the provided code to prevent a line being processed several times, which
    // happens in the provided implementation of the article

    int x_prev = 9999999;
    int x = radius;
    int y = 0;
    int err = 0;

    while (x >= y)
    {
        if (x != y)
        {
            if (center.y+y >= 0 && center.y+y < image.rows)
                drawLine(image, center.y + y, center.x - x, center.x + x, alpha_color, alpha);

            if (y != 0 && center.y-y >= 0 && center.y-y < image.rows)
                drawLine(image, center.y - y, center.x - x, center.x + x, alpha_color, alpha);
        }
        // We only want to plot this line if the x coordinate changed
        if (x_prev != x)
        {
            if (center.y+x >= 0 && center.y+x < image.rows)
                drawLine(image, center.y + x, center.x - y, center.x + y, alpha_color, alpha);

            if (center.y-x >= 0 && center.y-x < image.rows)
                drawLine(image, center.y - x, center.x - y, center.x + y, alpha_color, alpha);
        }

        x_prev = x;
        err += 1 + 2*y;
        y += 1;
        if (2*(err-x) + 1 > 0)
        {
            x -= 1;
            err += 1 - 2*x;
        }
    }
}
```

Fig. 5. Circle rendering function used in the CPU implementation of the fitness computation. Circles are added directly to the image canvas, therefore avoiding the need for expensive summing of OpenCV matrices.

A population of solutions is represented as a `std::vector` of chromosomes. The code sample in Fig. 6 shows how we distinguish between the CPU and GPU rendering in the code. In the case of CPU, each chromosome is processed separately by the `Renderer` and the fitness is then computed with basic operations on `cv::Mat` class. All these tasks run sequentially in one thread.

Note: Multi-threading on the CPU level is not used for rendering itself, but it is extensively used in the hybrid evolutionary algorithm, where we need to evolve each individual in the population with the hill climber. Therefore, a thread pool is created, which processes each individual in a separate thread and therefore allows to use the full CPU capacity for this task.

III. Parallel GPU Solution

In the GPU fitness computation we focus on fast computation of the fitness of the whole population of individuals (chromosomes). As shown in code snippet in Fig. 6, the rendering function gets a `std::vector` of chromosomes. It was designed specifically this way in order to support the population rendering by GPU. The advantage of GPU is that it can render multiple chromosomes at once and also multiple pixels in those chromosomes at once.

The fitness computation is split among different blocks and threads as shown in Fig. 7. We render one chromosome per block. The reason for this is mainly the size of shared memory. If the whole population has N individuals, we need to run N blocks. Each block then takes care of rendering of the whole image and computing of the fitness of its assigned individual.

A. *Memory*

The plotting canvas, where the image is being rendered has to be stored in memory of the GPU. The obvious choice for storing the plotting canvas is shared memory because it is accessed many times. However, since the size of the shared memory per block is limited to 48 kB in our case (see Tab. I), the whole canvas, which has to be an array of integers, cannot fit in the shared memory (e.g. $400 \times 400 \times 3 \times 4 \approx 2$ MB). The solution is to introduce one more layer of "inception", i.e. split the whole image into a grid of cells, which can fit into the shared memory of a block and will be rendered separately, as shown in Fig. 8.

Each block processes its own chromosome. It splits the task of fitness computation of the whole image into tasks of fitness computation in the separate grid cells. Each cell is rendered and its fitness is computed (fitness on this part of the image). The rendering can therefore happen in the shared memory only. Also, the chromosome description is copied into the shared memory (we limited the chromosome length to 250 shapes (10000 B) for this reason) in order to provide faster access since it is accessed many times. The grid cell size is 55×55 (36300 B) for the layered rendering and 46×46 (33856 B) for the averaging rendering as the averaging needs one more channel for computing the sum of α (weight) for each pixel.

B. *Representation*

The whole population is represented as one array of integers, where each circle takes 10 integers, each chromosome therefore $10 \cdot length$.[1] The whole population description is copied over to the global memory, from where each block copies its own part to its shared memory.

The target image is passed as an array of `uchar` of size $w \times h \times 3$ and the weights of the pixels as an array of `float` of size $w \times h \times 1$. Since each block accesses each pixel in those arrays only once, we keep them in the global memory.

```
// Computes fitness of the given chromosomes
void computeFitness (/*vector*/ &chromosomes)
{
#ifdef USE_GPU
    computeFitnessGPU(chromosomes);
#else
    computeFitnessCPU(chromosomes);
#endif
}
```

Fig. 6. Fitness computation split. Compile time option `USE_GPU` defines, whether GPU will be used for fitness computation or CPU. *Note: The code is invalid!*

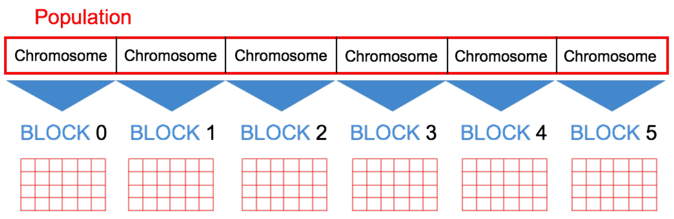

Fig. 7. GPU population rendering scheme. Each chromosome is assigned its own block of threads, which will then render it and compute its fitness.

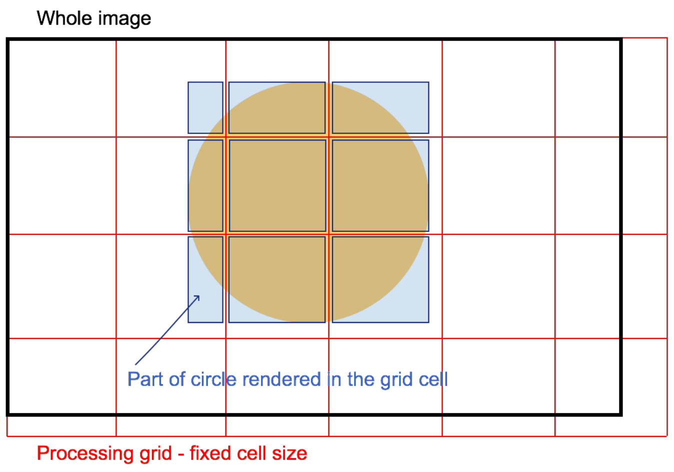
Whole image
Part of circle rendered in the grid cell
Processing grid - fixed cell size

Fig. 8. GPU image rendering grid. Image is split into a grid of cells, which can be rendered in the shared memory of the GPU. Fitness from each cell is computed separately.

C. *Kernels*

There are two different kernels (see Fig. 9). Since usually we do not need to copy the rendered images out from the GPU, there is a faster version of the fitness computing kernel, which does not store the rendered parts or the image. The slower kernel takes an extra input argument - array of `int` for the plotted images. After a cell is processed, its current canvas is copied to the output array. The array of `int` is then copied one by one image into a OpenCV `cv::Mat` and can be then

[1]We intentionally omitted the region of interest description, which is in the code as we did not want to go into detail about it here.

```
/**
 * @brief Kernel that computes the fitness values of all chromosomes in the given population
 *
 * Each block processes one chromosome (individual) from the population. Each block therefore
 * needs canvas for rendering the images in the shared memory. Since the shared memory is quite
 * small one needs to split the whole population into blocks. Also each chromosome is copied to
 * the shared memory, which needs to be shared between canvas and the chromosome description.
 *
 * This kernel also copies out the rendered images into the global memory g_all_canvas.
 *
 * @param g_target Data from cv::Mat of the target image that we want to compare (w x h x 3)
 * @param g_weights Weights of different parts of the target image (w x h x 1)
 * @param width Width of the target image (w)
 * @param height Height of the target image (h)
 * @param g_population Description vector of the whole population
 * @param population_size Number of chromosomes in the population
 * @param chromosome_length Number of shapes in each chromosome
 * @param g_out_fitness Output array for the computed fitness values (length = population_size)
 * @param g_all_canvas Array of cv::Mat canvas for rendering of the images
 */
__global__
void populationFitness (__uint8_t *g_target, float *g_weights, int width, int height,
                        int *g_population, int population_size, int chromosome_length,
                        float *g_out_fitness, int *g_all_canvas);

/**
 * @brief Same as previous kernel, but this one does not copy out the rendered images
 */
__global__
void populationFitness (__uint8_t *g_target, float *g_weights, int width, int height,
                        int *g_population, int population_size, int chromosome_length,
                        float *g_out_fitness);
```

Fig. 9. Two different GPU kernels, which can be used to compute the fitness function of each chromosome in the population. The `g_` prefix of variables denotes pointers to the global memory.

saved. A pseudo-code of kernel execution is provided in Fig. 10.

Since the threads very often read from the shared memory and write into it, in order to avoid bank conflicts the rendering and other loops are designed to access continuous blocks of memory by the same thread. An example of such a loop is in Fig. 11.

```
// Num. of elements to be processed per thread
int n = ceil(float(width*height) / blockDim.x);
for (int i = threadIdx.x*n;
        i < threadIdx.x*n+n; ++i)
{
    if (i < width*height)
    { /* Do stuff with cache[i] */ }
}
```

Fig. 11. Example loop that considers bank conflict avoidance. The solution is of course not perfect as we do not know what the width and height will be.

IV. Testing

The algorithms were tested on 2 different GPUs - Nvidia Tesla K40c and Nvidia GeForce GTX TITAN X, see Tab. I. The row with *Num. concurrent blocks* is provided in order to show how many blocks could be processed at once on the GPU, which for our rendering task is a very important feature. Both GPUs were very powerful, so we could expect nice speed up of the fitness computation.

The tests were focused on the fitness computation, therefore we will show time comparison on 3 different rendering tasks. The visual outputs - the approximations will be shown only to illustrate that the algorithms are working.

A. Test setup

Tests were carried out on 3 different test sets:

- Image 416×416 px, chromosome length 200.
- Image 416×416 px, chromosome length 100.
- Image 105×105 px, chromosome length 200.

```
__global__
void populationFitness (__uint8_t *g_target, float *g_weights, int width, int height,
                        int *g_population, int population_size, int chromosome_length,
                        float *g_out_fitness, int *g_all_canvas)
{
    int chromosome_id = blockIdx.x;
    float fitness = 0;
    Copy chromosome into __shared__ memory;
    rows x cols = span of the cell grid for the height x width image;
    for (each grid cell in rows x cols)
    {
        Initialize cell canvas;
        // Render the chromosome into the current cell
        for (each shape in the chromosome)
        {
            Render shape (or its part) if it intersects current grid cell;
        }
        Compute fitness of the current cell;
        fitness += fitness of the current cell;
    }
}
```

Fig. 10. Pseudo-code of kernel execution.

The image size corresponds to the size of the image that is being approximated - size of the whole canvas that needs to be rendered in order to compute the value of the fitness function. Each of these settings were also ran on 3 different population sizes 10, 100, and 1000.

All tests were carried out on 2 different computers with the same CPU Intel Xeon 2.40 GHz, but with different graphics cards described in Tab. I.

The time measured is the time to render the whole population of chromosomes, the measuring high precision clock was placed inside of the function `computeFitness()` around the `computeFitnessCPU()` and `computeFitnessGPU` calls (see the calls in Fig. 6). As the algorithms have random initialization of the chromosomes and the rendering time depends on the actual content of the chromosomes - larger circles take longer to render, we ran each computation 50 times and made an average from the runs.

Each test was also carried out for both types of renderings described in Section I-D.

	TITAN X	**Tesla K40c**
Compute capability	5.2	3.5
Number of CUDA cores	3072	2880
Multiprocessor count	24	15
Max threads per block	1024	1024
Shared mem. per block	49152	49152
Shared mem. per mult.	98304	49152
Num. concurrent blocks	48	15
Threads per block	128	192

TABLE I
USED GPU PROPERTIES, PROPERTIES IN ITALICS ARE USER DEFINED. (NOTE THAT TITAN X IS A SHORTCUT FOR GEFORCE GTX TITAN X)

V. MEASUREMENTS

The results of the time measuring tests on the three test sets are shown in Tabs. II, III, and IV. Because of the random nature of the algorithms, the measured times are with tolerance approximately $\pm 5\%$.

We can happily say that the GPU implementation outperforms the CPU implementation. The difference from the CPU implementation is about 10-fold in the case of TITAN X and about 5-fold for Tesla K40c. As we see from the tables, the difference is bigger for larger populations. If we compare Tabs. II and III, we see that in the case of a shorter chromosome the difference between CPU and GPU is even bigger than for longer chromosomes. On the other hand, when Tabs. II and IV are compared, we see that the difference between CPU and GPU is smaller for smaller images.

Out of the two GPUs the TITAN X clearly wins. The reason is that it has more multiprocessors and also double the shared memory per multiprocessor, which allows it to run 48 blocks at once, whereas in the case of Tesla K40c only 15 blocks can run at once. The fact that the Tesla K40c GPU can use more threads per block does not help it enough to catch the performance of the TITAN X.

As of the different rendering types the results correlate. We see that the layered rendering takes less time

	Layered rendering			Averaging		
Population size	CPU	TITAN X	Tesla K40c	CPU	TITAN X	Tesla K40c
10	61 ms	**11 ms**	17 ms	69 ms	**15 ms**	19 ms
100	553 ms	**34 ms**	127 ms	648 ms	**46 ms**	144 ms
1000	6225 ms	**241 ms**	1179 ms	6422 ms	**294 ms**	1296 ms

TABLE II
COMPARISON OF POPULATION RENDERING TIMES. IMAGE SIZE 416×416 PX, CHROMOSOME LENGTH 200.

	Layered rendering			Averaging		
Population size	CPU	TITAN X	Tesla K40c	CPU	TITAN X	Tesla K40c
10	43 ms	**7 ms**	12 ms	63 ms	**9 ms**	13 ms
100	468 ms	**25 ms**	87 ms	570 ms	**28 ms**	86 ms
1000	4095 ms	**144 ms**	751 ms	5563 ms	**183 ms**	735 ms

TABLE III
COMPARISON OF POPULATION RENDERING TIMES. IMAGE SIZE 416×416 PX, CHROMOSOME LENGTH 100.

	Layered rendering			Averaging		
Population size	CPU	TITAN X	Tesla K40c	CPU	TITAN X	Tesla K40c
10	5 ms	**1 ms**	2 ms	6 ms	**2 ms**	2 ms
100	45 ms	**4 ms**	10 ms	58 ms	**5 ms**	14 ms
1000	443 ms	**31 ms**	99 ms	516 ms	**41 ms**	132 ms

TABLE IV
COMPARISON OF POPULATION RENDERING TIMES. IMAGE SIZE 105×105 PX, CHROMOSOME LENGTH 200.

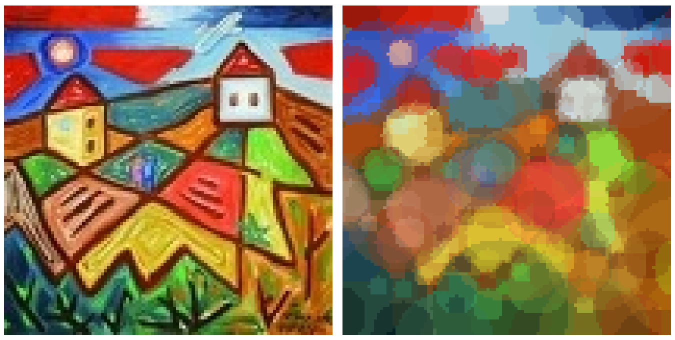
Fig. 12. Sample results of the optimization. Left is the original.

than the averaging, which is not surprising as we have to render one extra channel.

A. Correctness

In order to determine, if the algorithms are giving correct results, we ran several optimizations and here we present the resulting rendered approximations. A sample cubistic painting is shown in Fig. 12 and Mona Lisa in Fig. 13. In the case of Mona Lisa we used the user defined weights from Fig. 3 in order to pronounce more the eye, nose, and mouth regions.

VI. CONCLUSION

The project showed promising results of the GPU implementation, even though in the beginning we were skeptical about if it can actually have some advantage because of the copying of data to and from GPU. However, we solved this in an elegant way, where we copy the target image to the GPU only once an then render the whole population with one kernel call instead of rendering the individuals one by one.

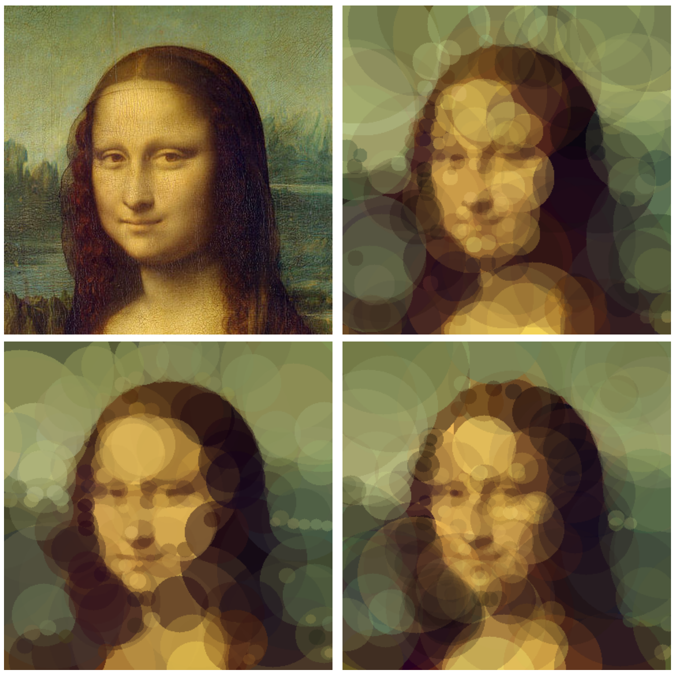
Fig. 13. Sample results of the optimization. Top left is the original.

The GPU implementation is from $5-20\times$ faster than the CPU version, which is a nice achievement. There was a lot of room to learn what to use shared memory for, how to solve the problem of insufficient shared memory and how to avoid copying too much data from and to GPU.

The whole code base is publicly available on Github at https://github.com/libornovax/eoa_image_compression.

References

[1] S. Luke, *Essentials of metaheuristics*. Lulu Com, 2013.
[2] "Genetic programming: Evolution of Mona Lisa," https://rogerjohansson.blog/2008/12/07/genetic-programming-evolution-of-mona-lisa/, 2008.
[3] Wikipedia, "Midpoint circle algorithm," https://en.wikipedia.org/wiki/Midpoint_circle_algorithm, 2008.

Appendix

Here we describe how to **compile and run the code**. In order to do that, the following C++ packages are needed:

- OpenCV 2.4 or newer
- yaml-cpp

Clone the project from the Github page. To compile the code create a folder `build` and `bin` inside the project folder. Then `cd` to `build` and run:

```
cmake ..
make
```

Eventually you might have to specify your CUDA capability or your gcc compiler in the `CMakeLists.txt` and run the previous two commands again.

You should now see 4 executables in your `bin` folder. Each executable does different kind of rendering. The ones without the *_gpu* appendix do not need GPU at all to run.

Then, to run the optimization choose one of the configuration files in the `config` folder and edit the `path_image`, `path_image_weights` and `path_out` entries to correspond with your setup. After that is done `cd` to the `bin` folder and run one of the following commands:

```
./compress ../config/hill_climber_config.yaml             # CPU layered rendering
./compress_gpu ../config/hill_climber_config.yaml         # GPU layered rendering
./compress_average ../config/hill_climber_config.yaml     # CPU averaging rendering
./compress_average_gpu ../config/hill_climber_config.yaml # GPU averaging rendering
```

Or with a different configuration file that you have chosen. The algorithm saves intermediate results every 200 epochs so soon you should see images appearing in your `path_out` folder.