Gaussian Blur

Mold Florian, Karakaya Aytac

Gaussian function in two dimensions:

$$G(x,y) = \frac{1}{2\pi\sigma^2} e^{-\frac{x^2+y^2}{2\sigma^2}}$$

Problem explanation

- A Gaussian blur is the result of blurring an image by a Gaussian function.
- It is a widely used effect in graphics software, typically to reduce image noise and reduce detail.
- The Gaussian blur is a type of image-blurring filter that uses a Gaussian for calculating the transformation to apply to each pixel in the image.
- The Gaussian outputs a 'weighted average' of each pixel's neighborhood, with the average weighted more towards the value of the central pixels.
- The original pixel's value receives the heaviest weight (having the highest Gaussian value) and neighboring pixels receive smaller weights as their distance to the original pixel increases.
- Parameters:
  - Sigma: This defines how much blur there is. A larger number is a higher amount of blur.
  - Radius: The size of the kernel in pixels

Workload

- The **xl.bmp** image has a resolution of 2000px x 2000px which has 4.000.000 pixels in total. This results in 1.760.000.000 total iterations that need to be made to calculate the blurred image with a radius of **10**.

- This huge amount of workload comes from computing the kernel for every pixel.

Example kernel: $\frac{1}{16}\begin{array}{|c|c|c|}\hline 1 & 2 & 1 \\ \hline 2 & 4 & 2 \\ \hline 1 & 2 & 1 \\ \hline \end{array}$

Environment

- Technology: C++, OpenMP
- 8 Core CPU
- 16GB RAM
- Shell script for executing the program and collecting the runtimes of the program into a .csv file.

```
for j in $(seq 1 $maxThreads);
do
  for i in ${images[@]}; do
    for r in ${radius[@]}; do
      ./make-test.sh -i $maxIterations -t ${j} -n cmake-build-debug/images/${i}.bmp -r ${r} -f results/radius-${r}/image-${i}/result-t${j}.txt
    done
  done
done
```

- For all measurements, the program was executed 30 times and the mean of the execution times was taken for the calculations.
- We only measured calculating the values of the pixels. Not the reading/writing of the image to the disk.

Parallelized Part

$$G(x,y) = \frac{1}{2\pi\sigma^2} e^{-\frac{x^2+y^2}{2\sigma^2}}$$

Gaussian function in two dimensions:

```
double square = (col - j) * (col - j) + (row - i) * (row - i);
double sigma = radius * radius;
double weight = exp( x: -square / (2 * sigma)) / (M_PI * 2 * sigma);
```

```
#pragma omp parallel for default(none) shared(height, width, radius, red, green, blue) collapse(2) schedule(guided)
    // Loop every pixel
    for (int i = 0; i < height; i++) {
        for (int j = 0; j < width; j++) {
            int row, col;
            double redSum = 0, greenSum = 0, blueSum = 0, weightSum = 0;

            // Sum the value of every pixel in the radius of the target pixel
            for (row = i - radius; row <= i + radius; row++) {...}
            // Change the value of the pixels by the calculated weight
            red[i * width + j] = round( x: redSum / weightSum);
            green[i * width + j] = round( x: greenSum / weightSum);
            blue[i * width + j] = round( x: blueSum / weightSum);
        }
    }
```

Approach

- The two nested loops which compute the new blurred value of every pixel can be executed in parallel.
- Every iteration of the loop is independent from the previous iteration, so the work can be distributed evenly for every thread.
- We chose **schedule(guided),** because not every thread has the same amount of work. Pixels at the edge of the image do not have to calculate a kernel that is as large as a kernel in the center of the image.
- The two nested loops can be collapsed, because the loops are independent from each other.
- We went for **data parallelism**, because the computation for every pixel is the same and it can be easily parallelized with a for loop.

Problems

- We tried to parallelize the inner loops, which calculate the kernel for every pixel.
- But this slowed down the program, because every thread of the outer loop spawn's new threads for the inner loop, which must wait at the end of the **parallel for** section, because there is an implicit barrier. We used reductions, so that every thread has its own copy of *redSum*, *greenSum*, *blueSum* and *weightSum*.
- We do not know the specific reason, why the program performed slightly worse, we think **oversubscription** happens.

Problems

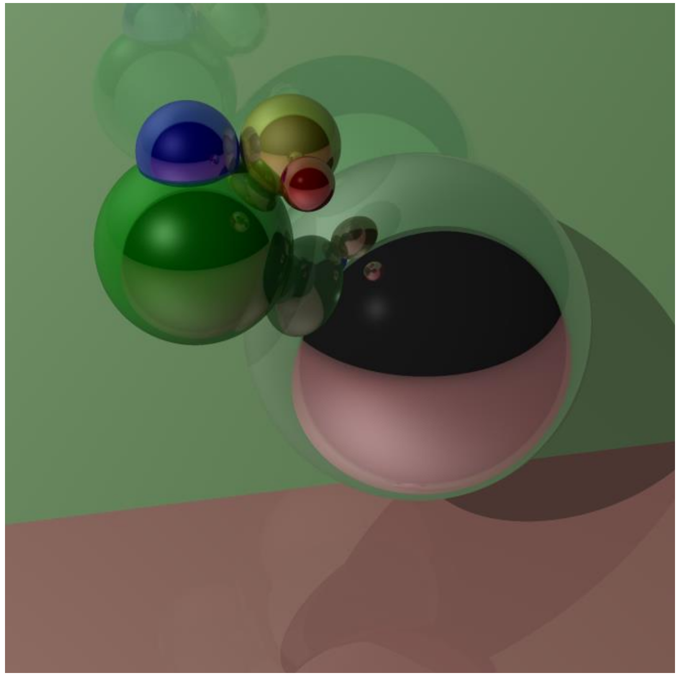
```
            // Sum the value of every pixel in the radius of the target pixel
#pragma omp parallel for default(none) shared(height, width, radius, red, green, blue, i, j) private(col) reduction (+:redSum, blueSum, greenSum, weightSum)
            for (row = i - radius; row ⩽ i + radius; row++) {
                for (col = j - radius; col ⩽ j + radius; col++) {
                    int x = clampIndex( index: col,  min: 0,  max: width - 1);
                    int y = clampIndex( index: row,  min: 0,  max: height - 1);

                    int tempPos = y * width + x;

                    double square = (col - j) * (col - j) + (row - i) * (row - i);
                    double sigma = radius * radius;
                    double weight = exp( x: -square / (2 * sigma)) / (M_PI * 2 * sigma);

                    redSum += red[tempPos] * weight;
                    greenSum += green[tempPos] * weight;
                    blueSum += blue[tempPos] * weight;
                    weightSum += weight;
                }
```

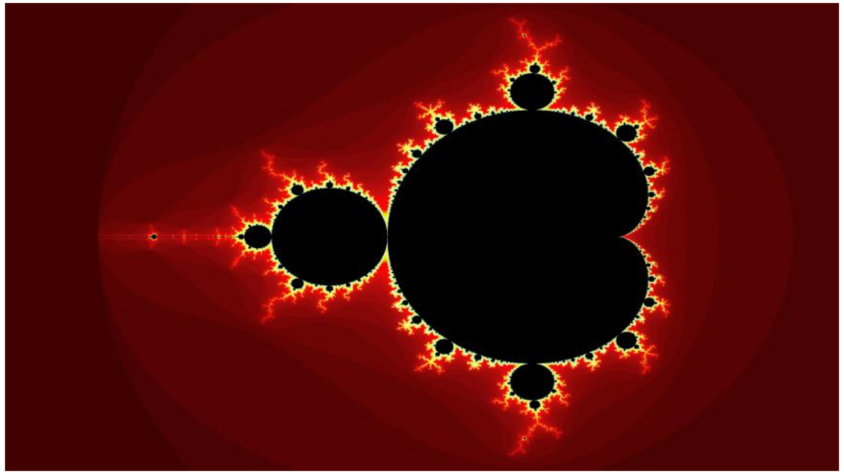
Test images



Large image: xl.bmp
2000px x 2000px

Small image: delta.bmp
800px x 600px

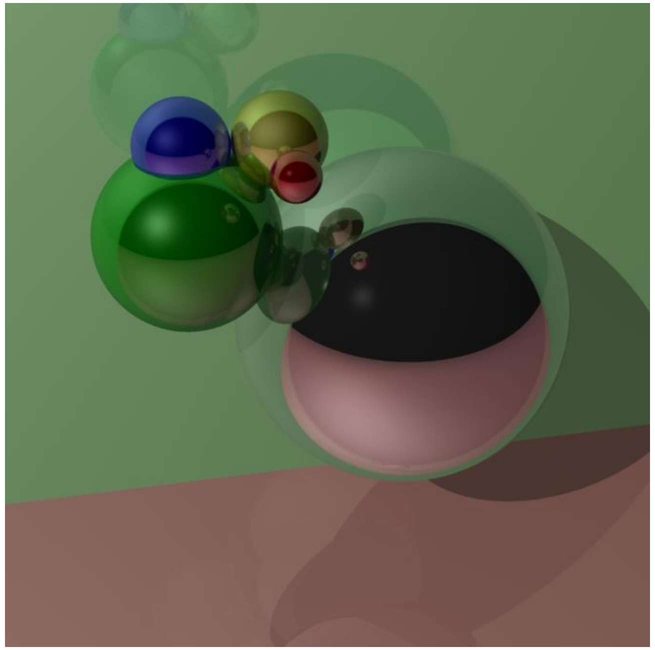
Medium Image: balls.bmp
1400px x 1000px



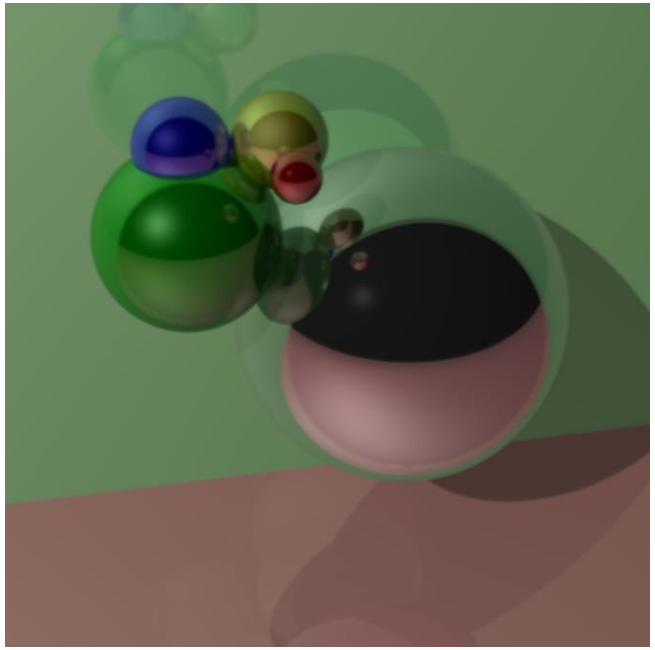
Medium image: mandelbrot.bmp
1920px x 1080px

Results

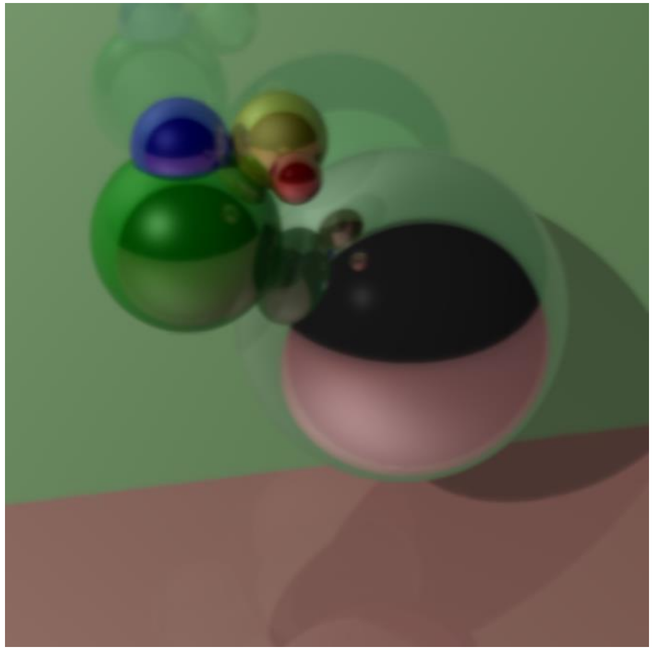
- Large Image (2000px x 2000px)



radius 1, xl.bmp



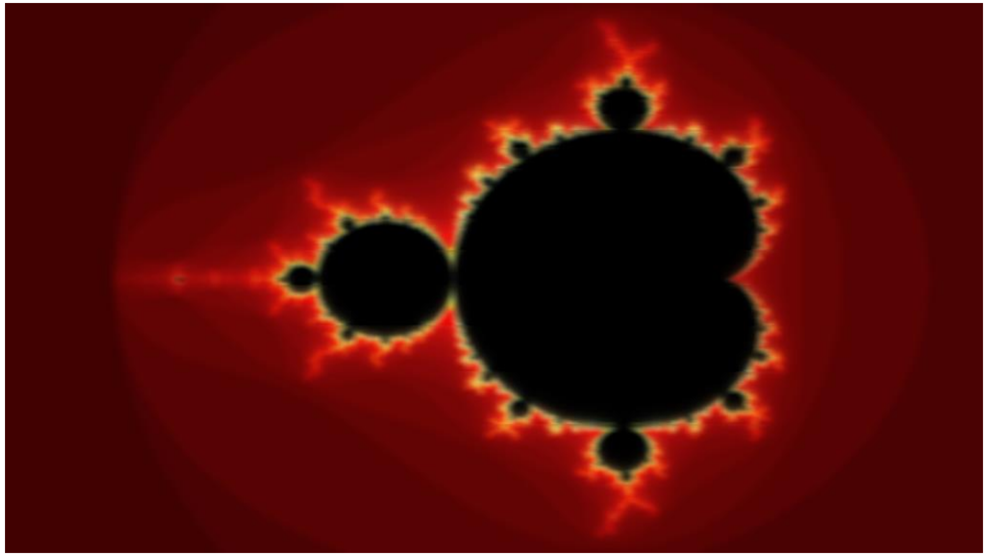
radius 5, xl.bmp



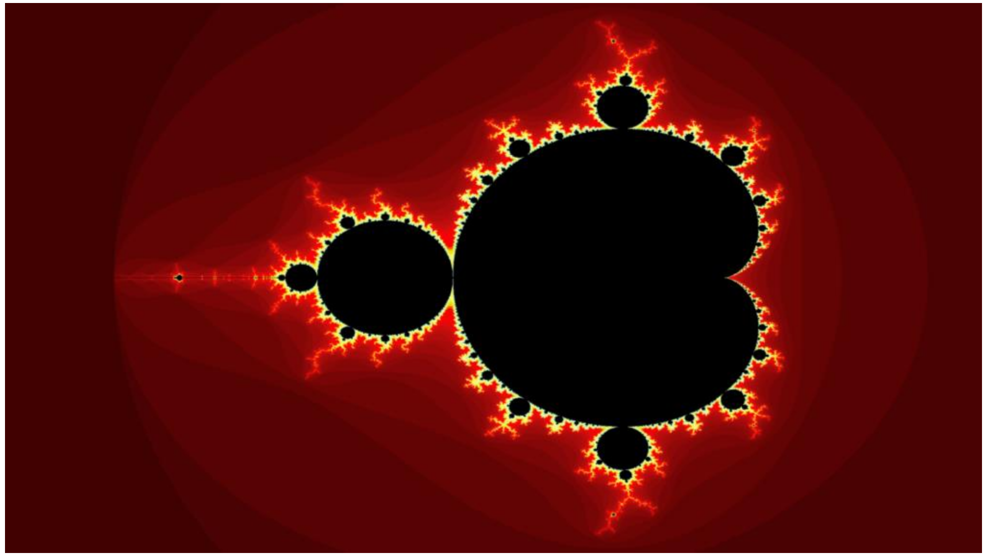
radius 10, xl.bmp

Results

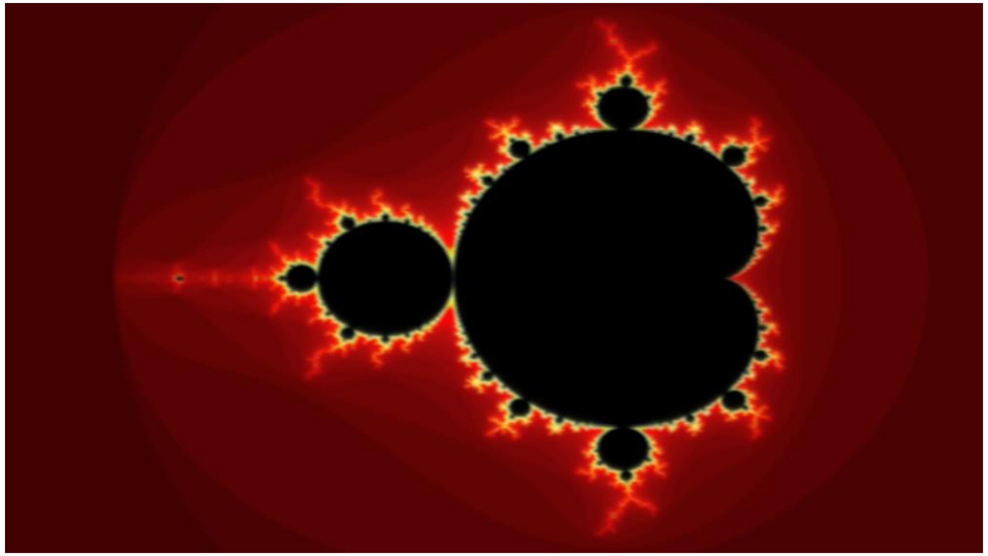
- Medium Image (1920px x 1080px)



radius 10, mandelbrot.bmp



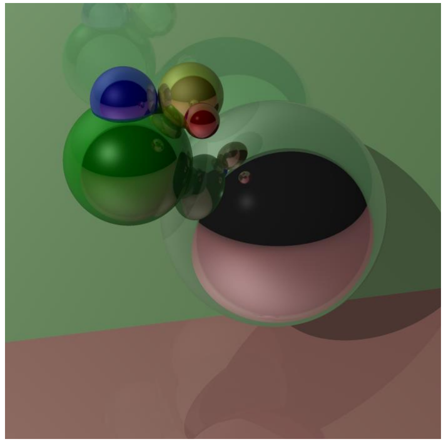
radius 1, mandelbrot.bmp

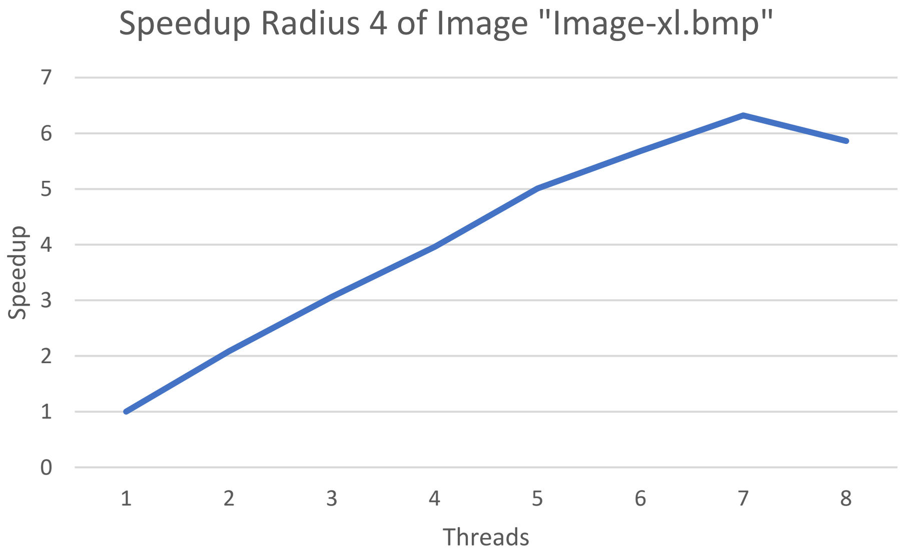


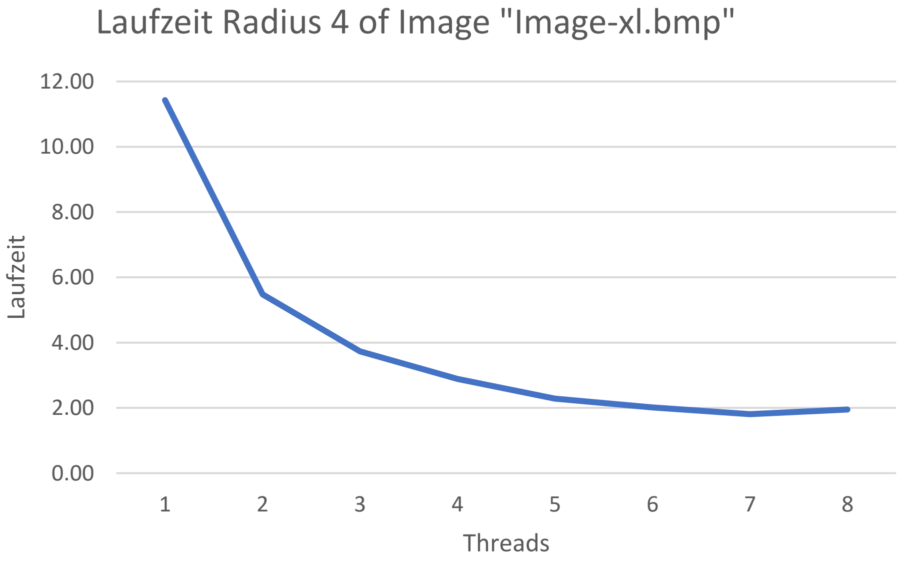
radius 5, mandelbrot.bmp

Speedup: Large Image

- Large Image: xl.bmp, radius 4

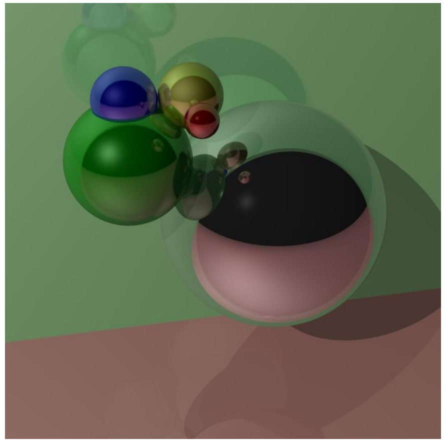


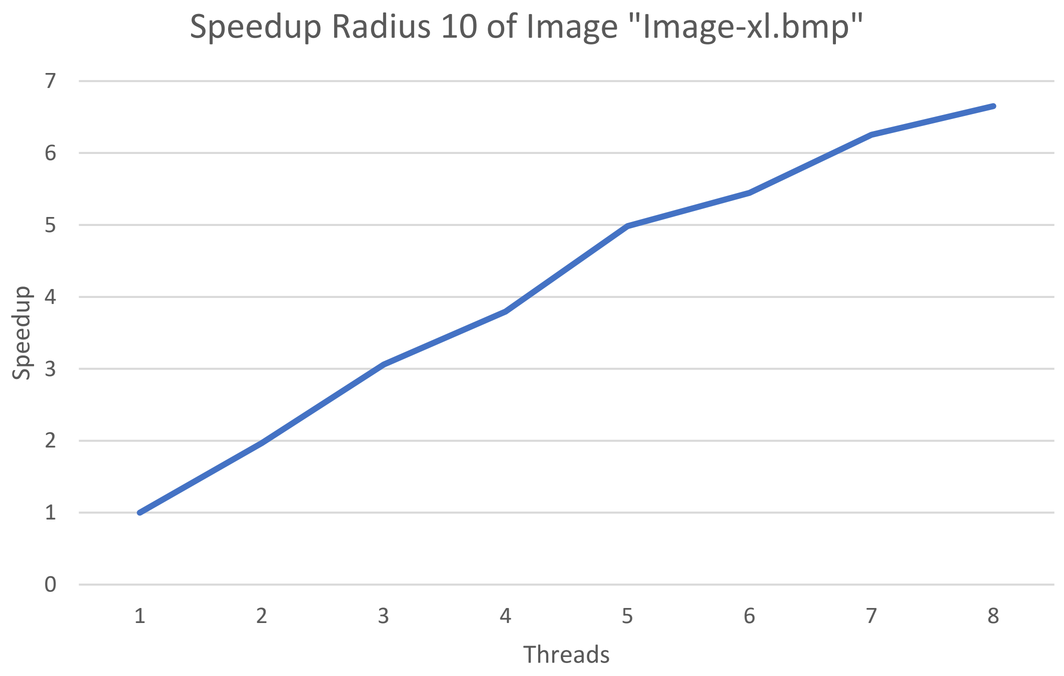


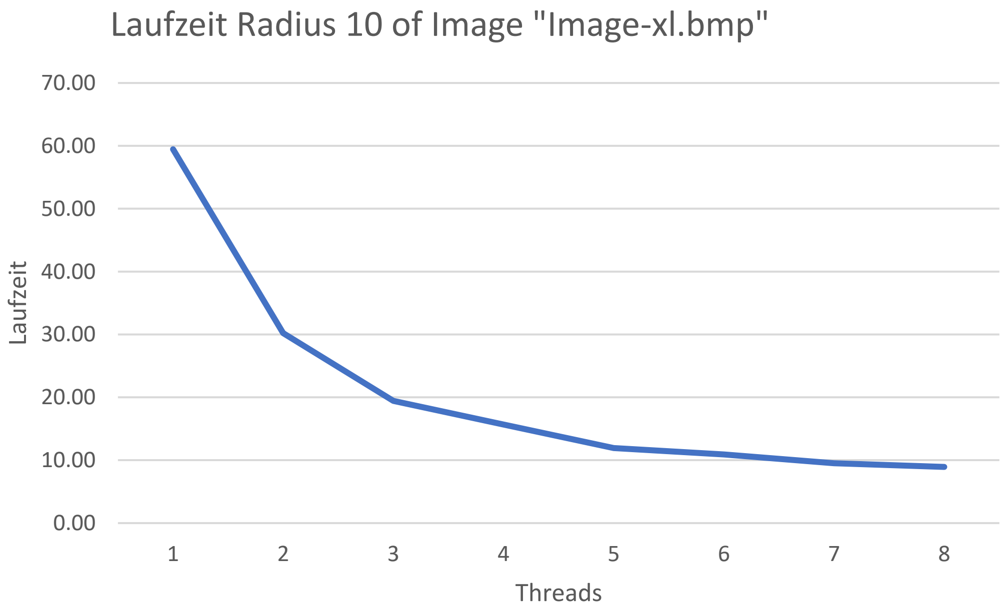


Speedup: Large Image

- Large Image: xl.bmp, radius 10

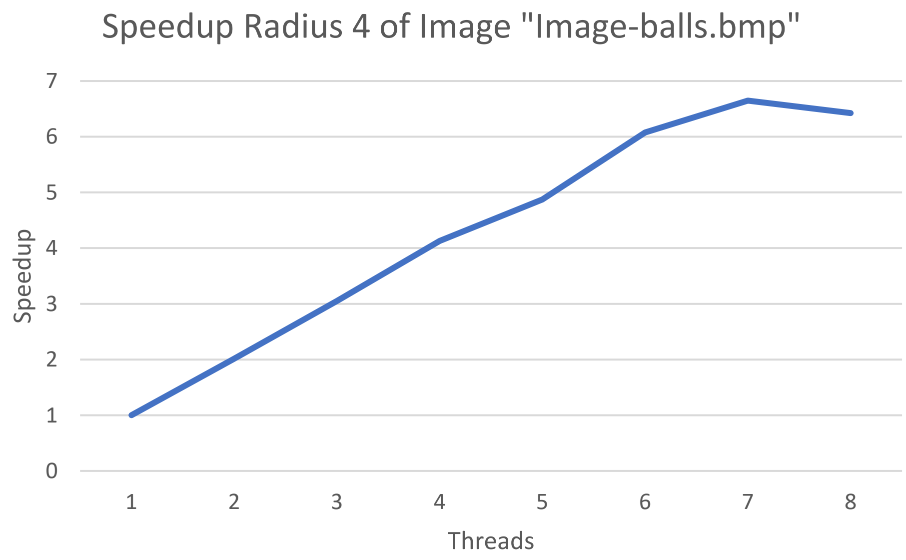


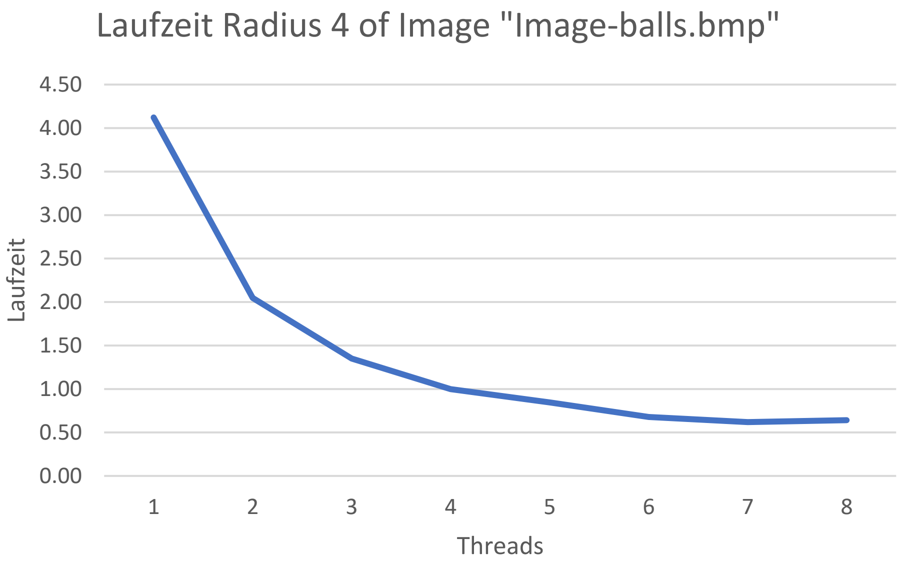




Speedup: Medium Image

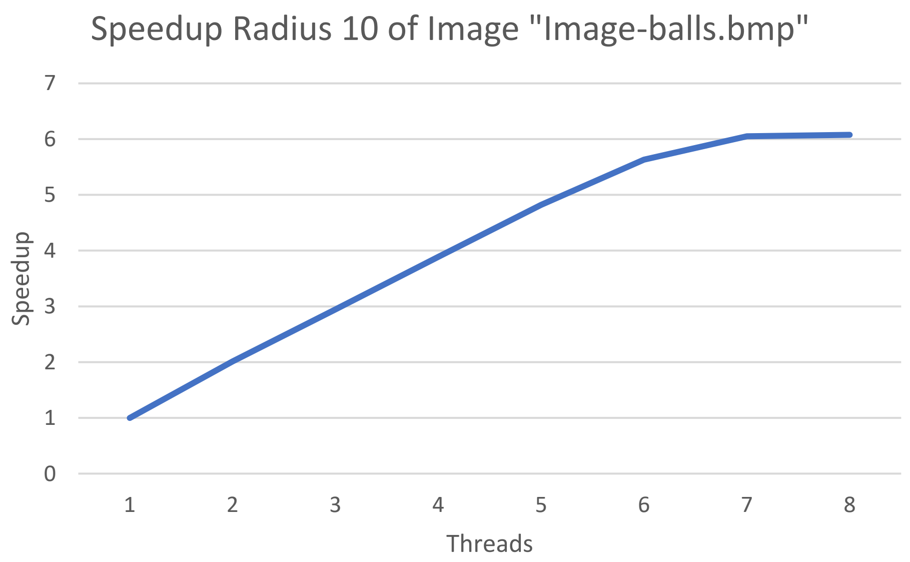
- Medium Image: balls.bmp, radius 4

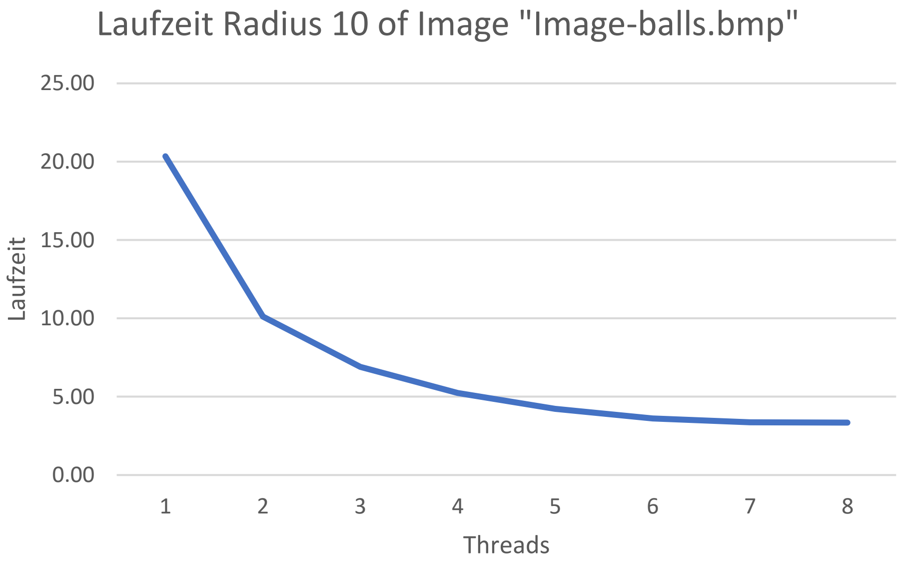




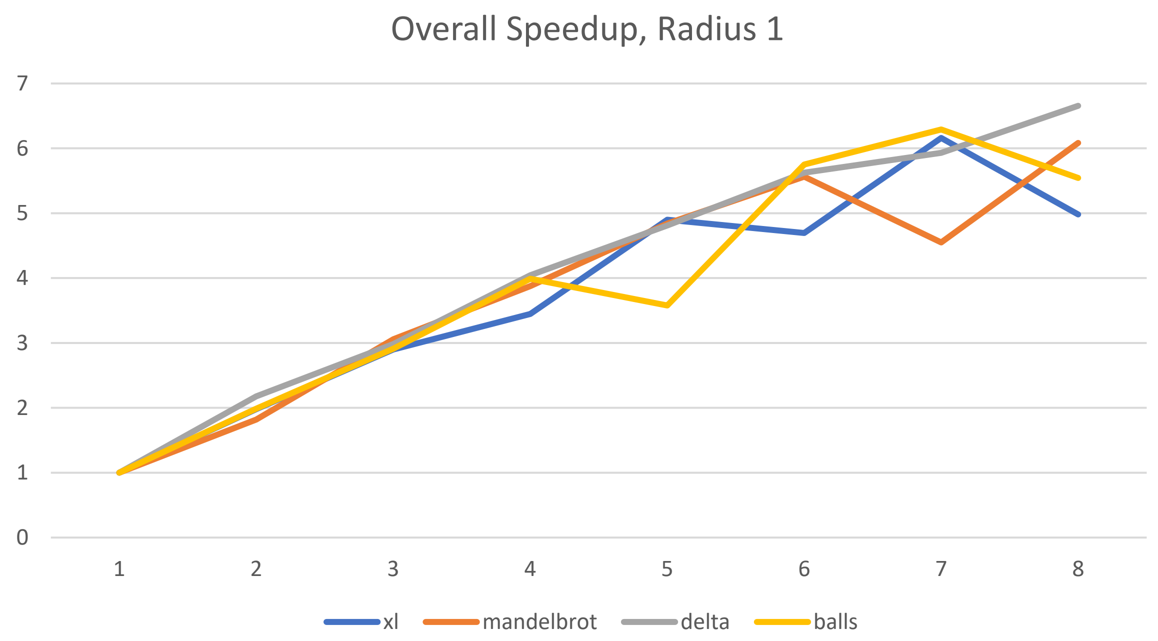
Speedup: Medium Image

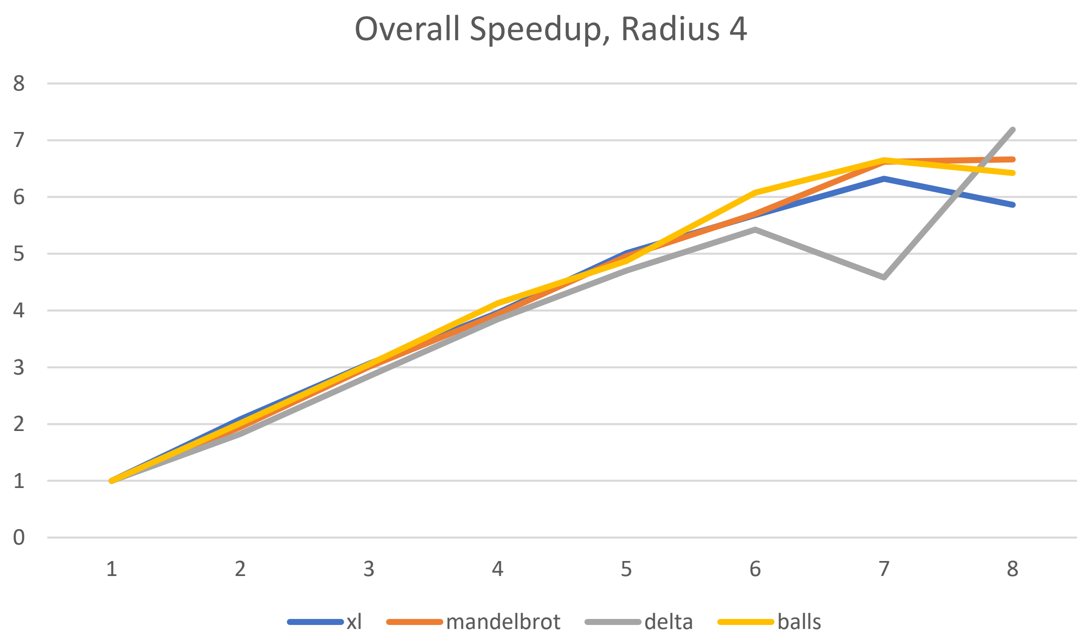
- Medium Image: balls.bmp, radius 10



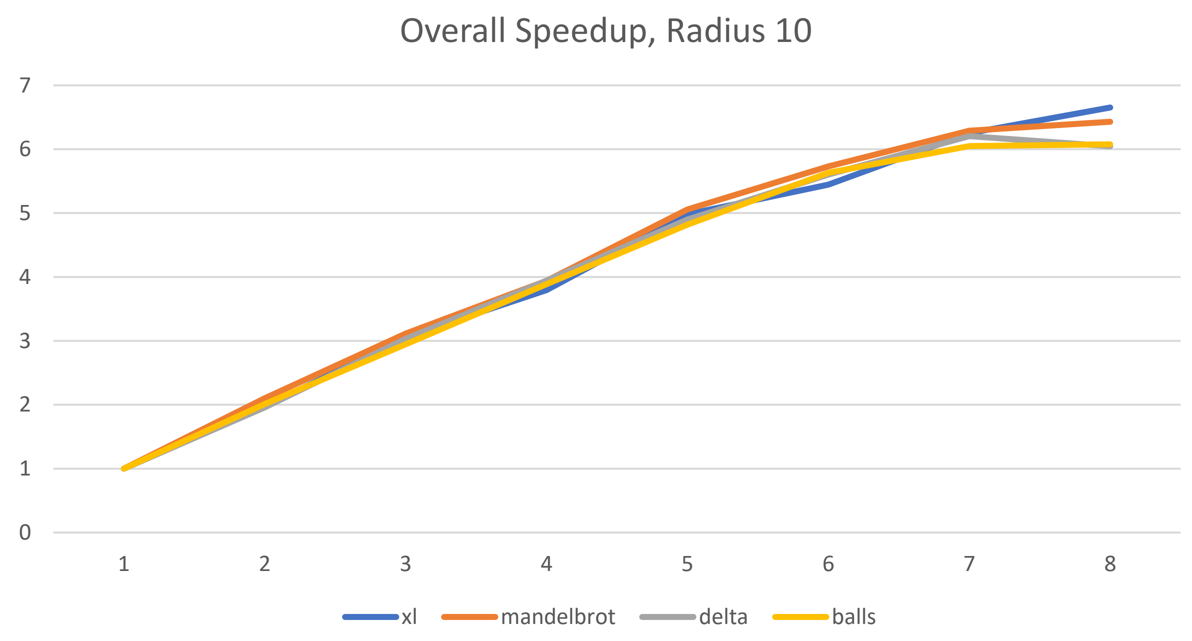


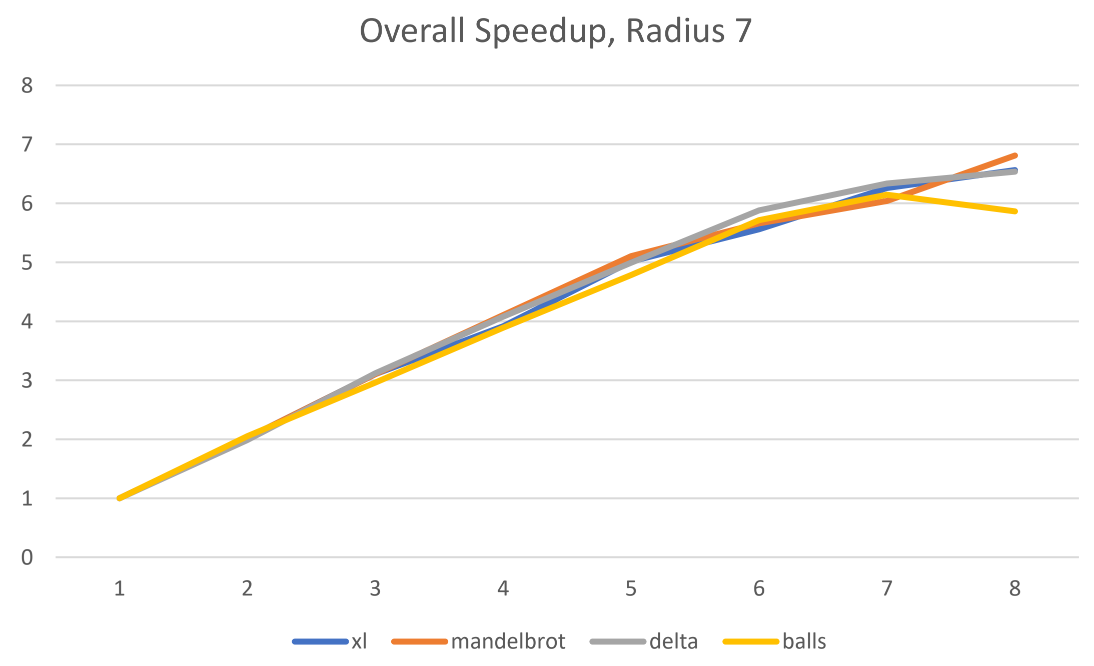
Overall Speedup





Overall Speedup





Conclusion

- At first our expected speedup with 8 threads was 6. This expectation was due to having many nested loops, which couldn’t be parallelized quite well.
- For high radius values and large images, we got a speedup of about 7, which is very good.
- Lower radius values seem to have not as much speedup. Maybe the workload is too small for the number of threads. For smaller images, the speedup seems to stagnate with 8 threads.
- Lower radiuses seem to have a problem speeding up the program, because the speedup plummets sometimes with a higher threadcount.
- Overall, our speedup exceeded our expectations, with the highest average speedup we achieved being **~7,18.**