Parallel Programming

Project, Data Parallelism – Parallel Gaussian Blur

Team Members

Aljoscha Alquati - se21m019
Michael Goll - se21003

Approach

The programming language chosen was C/C++. First, a “kernel” is generated that is used by the actual algorithm to calculate the blurring of the pixels within a area of the image. The processing of the images with a gaussian blur algorithm was implemented using 5 loops. Two loops iterating over the rows and column pixels of the image. Inside these two loops (a nested loop) iterates over the 3 channels (RGB). Inside this loop, two more loops iterate over the x- and y-axis of the “kernel” manipulating the pixels to produce the blur effect.

Parallelization was achieved by utilizing the `# pragma omp for` statement provided by the OpenMP library:

First, we tried to parallelize the gaussian blur algorithm’s outermost loop (iterating over the rows). This resulted in the highest speedup. Parallelizing the nested loops further did not yield significant a speedup, so we decided to omit this step in the final solution.

We also tried to parallelize the generation of the kernel, which did also not result in a significant speedup. We also tried to use static scheduling, which resulted in worse execution times. Changing the chunk sizes decreased execution times as well. The final solution, therefore, is set to use dynamic scheduling with a default chunk size of 1.

We used OpenCV to load the image for further manipulation. This library also provides an implementation of the gaussian blur algorithm. This implementation produced the best results (execution time of around 50 ms).

Performance

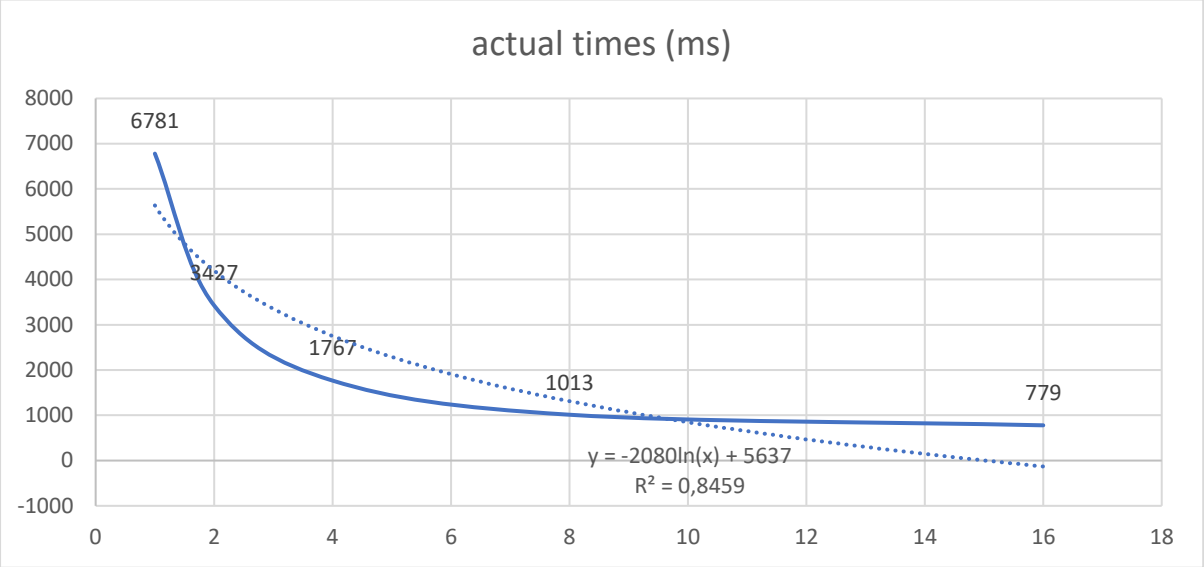
Parameters for the gaussian blur algorithm:

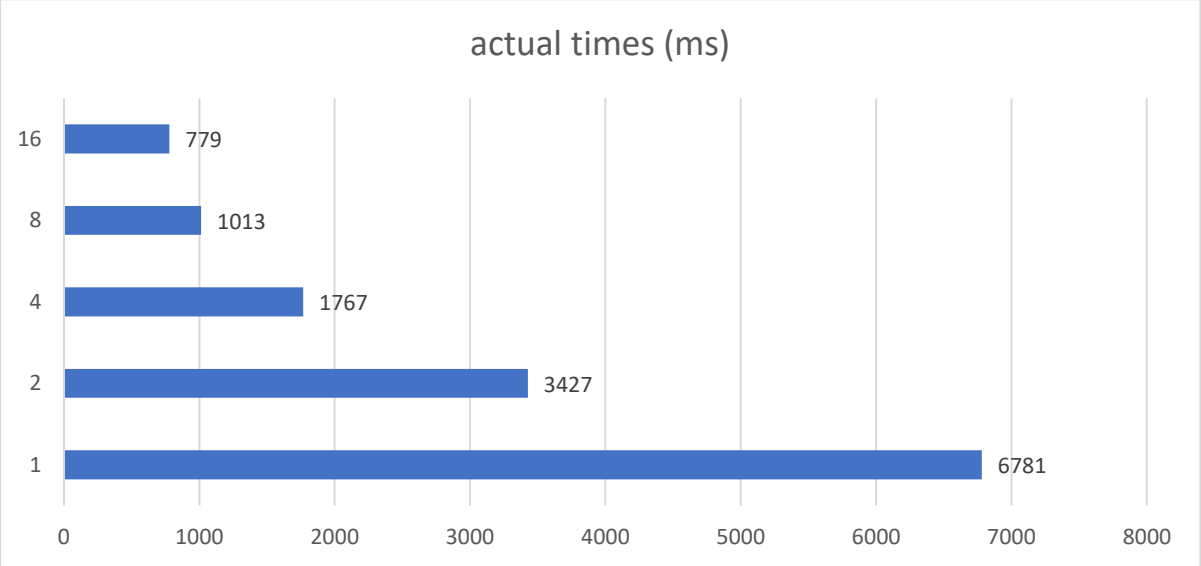
Radius: 9
Sigma: 1
Picture size: 951 x 588 pixel
CPU: AMD Ryzen 7 4800H CPU with 8/16 Cores and a base clock of 2900 Mhz and a max boost of 4300 Mhz

The benchmark consisted of 10 consecutive runs of the gaussian blur algorithm. The following table shows the average execution times achieved for each number of threads, and the corresponding speedup quotient:

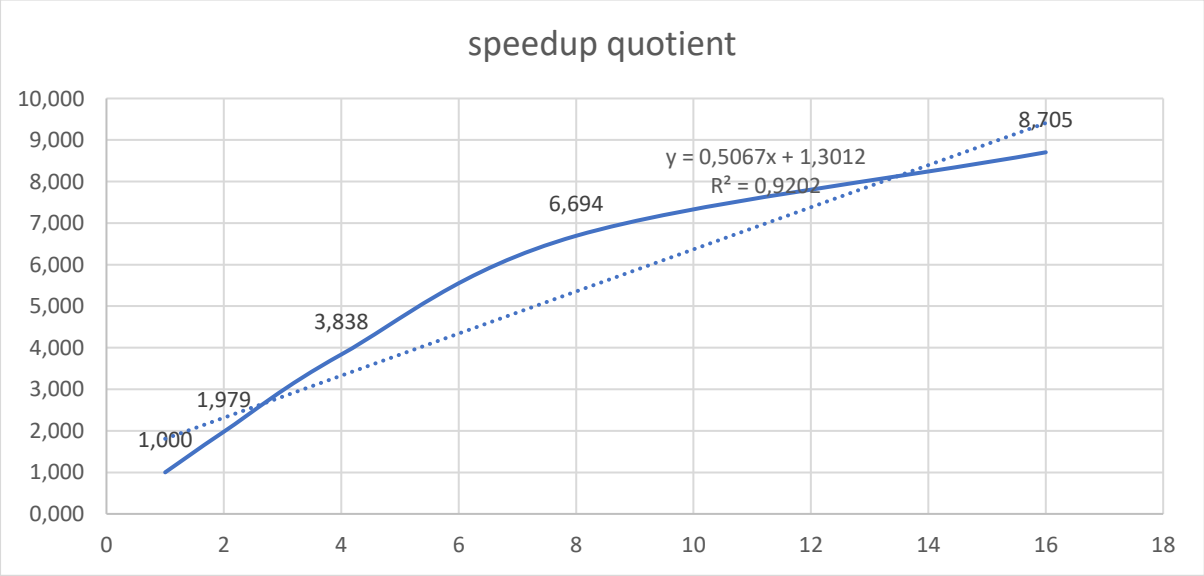
number of threads	1	2	4	8	16
actual times (ms)	6781	3427	1767	1013	779
speedup quotient	1,000	1,979	3,838	6,694	8,705

The actual times included the generation of the kernel and excluded the display of the finished image. The image is not stored to disk.





These results show a steep decline of execution times at 2 to 4 threads. Increasing the number of threads further has a limited effect on execution times, however, the best result can be achieved by utilizing the total number of available threads (16).



The speedup quotient shows a steady increase until the total number of available threads is utilized. Thereafter, no speedup was recorded (and therefore omitted in the plot).