Project assignment

Parallel and distributed systems and algorithms

Image Compression using K-Means Clustering

Ivan Nikolov

January 2022

1. Motivation

Image compression with K-means clustering is a techicque that uses unsupervised learing in order to reduce the colors present in the image. All the different colors are grouped in K number of clusers and each cluser is represented by its centroid. With this we have K colors that preserve the distribution of the color spectrum on the image so the information loss is minor.

2. Serial algorithm

The serial algorithm is implemented in the C programming language It is composed of one main for loop that goes through all the iterations and two inner for loops. Before we go into the for loop, we initialize the centroids at random values.

The first inner for loop iterates through all the pixels of the image and finds the nearest cluster. We keep the cluster indexes for each pixel in an array with the index representing the pixel index and a value representing the cluster which the pixel belongs to. Another thing we update in this loop is the number of pixels that are present in the cluster.

Because this is an optimization algorithm, it can often happen that some clusters remain empty, causing the number of colors to be below the desired number. To avoid this, we assign a random value to the empty cluster.

The second for loop goes through all the centroid indexes and calculates the mean for each centroid.

After this, the only thing that is left is to write the colors in the result image. This is accomplished by iterating trough the pixels, seeing to which cluster they belong to and writing the value of that centroid to the new image.

3. OpenMP algorithm

For implementation of this algorithm, we used the OpenMP library. The foundation of this algorithm is the serial algorithm. First, we set up the number of threads that we want to use.

The outer for iteration space is small and the result of the next operation is dependent on the previous, so we cannot parallelize this section.

The first inner for loop is parallelized using the ‘#pragma omp parallel for’ macro that divides the iteration space between the threads. Because multiple threads update the array that holds the number of pixels for each sample, we must mark this as an atomic operation.

The same applies for the second for loop, we mark which variables are private and shared and add the atomic instructions.

The writing back to the result image is also parallelized. Because these are the last instructions of the function and there is no need for synchronization, we add the ‘nowait’ key word in the pragma instruction. With this the threads no longer wait each other at the end of each iteration.

4. OpenCL algorithm

For implementation of this algorithm, we used the OpenCL library, which can be utilized for GPU programming.

The main for loop remains on the CPU and calls the two kernels that run of the GPU.

The first kernel is a substitute for the first for loop. The work here is divided by pixels. Each pixel has its own version of the kernel. Because there are many accesses in the global memory for the centroids, we transfer them in the local memory. Before continuing to the next instruction, the loops need to wait each other and finish the transfer to the local memory. This is done by using a barrier that synchronizes the threads on one compute unit. All the updates of the arrays are done directly in the global memory since doing them in first in local and then transferring to the global just extends the execution time.

The second kernel calculates the means for each cluster. The work is divided by centroids. At the end of the second kernel, we also reinitialize the values to zero.

One of the changes that we made in the GPU algorithm is that we changed the order in which the RGB values are written.

The centroid values in the serial and the OpenMP algorithm are written in the following order.

B_0	G_0	R_0	B_1	G_1	R_1	B_2	G_2	R_2	B_i

With this, the values of the neighboring threads are not next to each other, and more memory accesses are needed to get them. This leads to poorer performance of the algorithm.

To solve this problem we change the order of the RGB values, so the blue values are first, followed by the green and red values.

B_0	B_1	B_2	G_0	G_1	G_2	B_0	B_1	B_2

By using this method, the treads get more data from the global memory with one reading access.

The third kernel is responsible for writing the results of the compression in the new image.

5. Results

The algorithm was tested on multiple pictures with the smallest being 640x480px and the biggest being 3840x2160px.

The images (Image 1, Image 2) display the input and the output of the algorithm with K=64 and 50 iterations.

Image 1: Input image

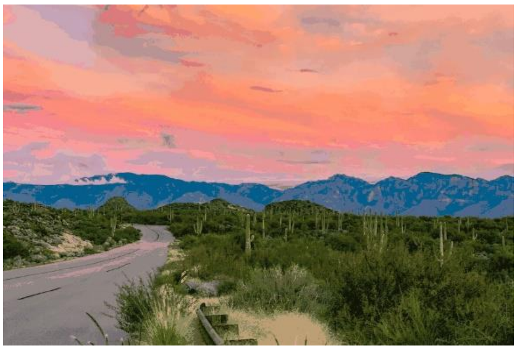


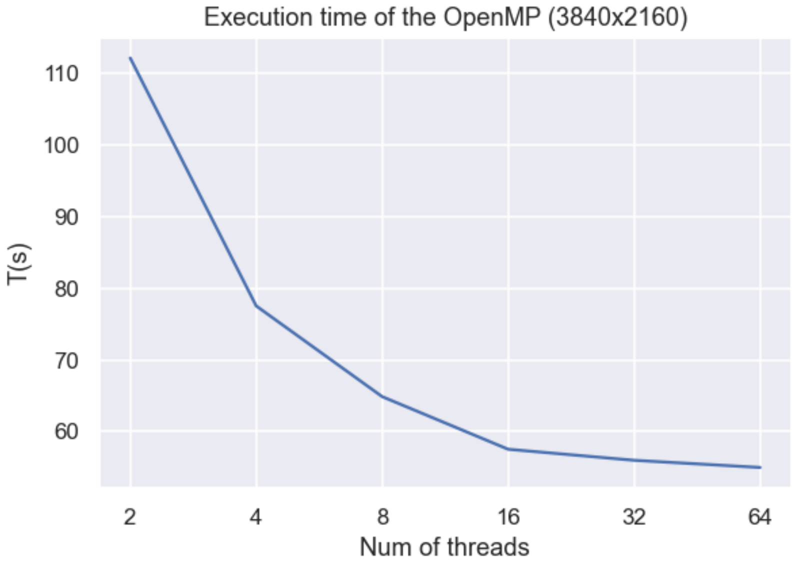
Image 2: Output image

The testing was done on the nsc server at the Institute Jozef Stefan. All algorithms were tested with cluster number K = 64 and 50 iterations.

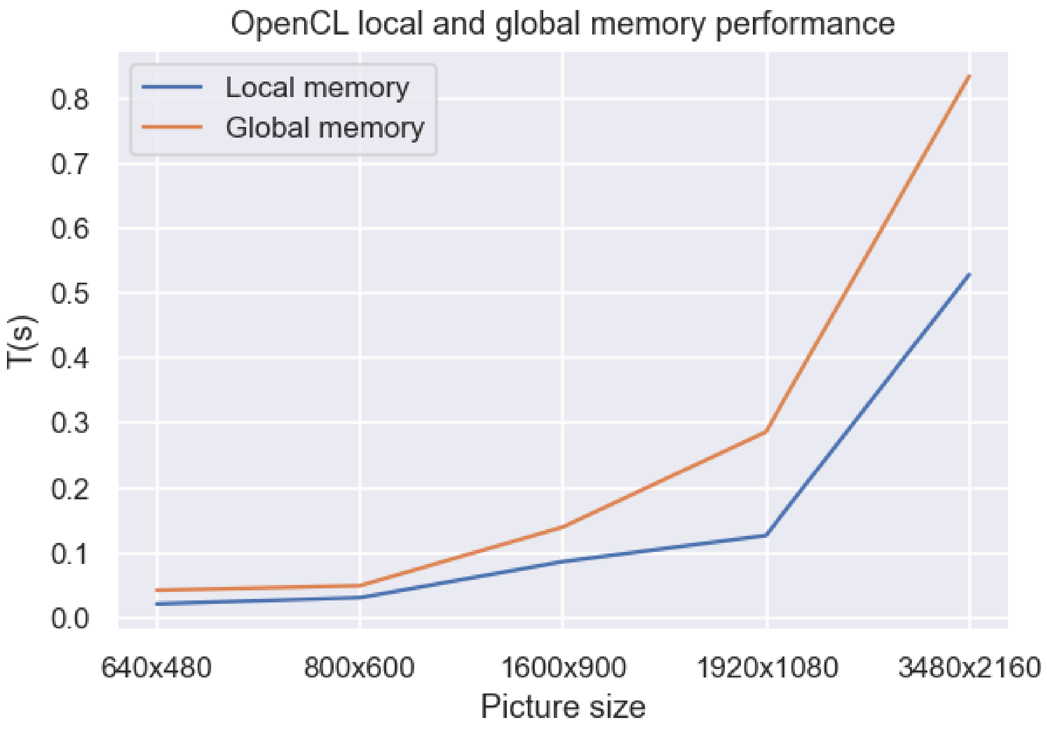
Image size	Serial time (s)	OpenMP time (s)	OpenMP speedup	OpenCL time (s)	OpenCL speedup
640x480	4.68864	1.590741	2,947	0.021011	223,152
800x600	7.153839	2.667327	2.682	0.030793	232,320
1600x900	21.644183	8.134683	2,661	0.086459	250,340
1920x1080	25.486731	9.51534	2,678	0.12668	201,190
3840x2160	146.499173	55.617677	2,634	0.529111	276,878

From the table above we can see that OpenCL has a speedup around 250 while the for the OpenMP algorithm the speedup is around 2.5.
The OpenMP version is slower because we could not resolve the critical sections and there are multiple instructions that are atomic. We tried resolving this problem by using reduction, but because it was too complex to have a separate table for each cluster, the attempt was unsuccessful.

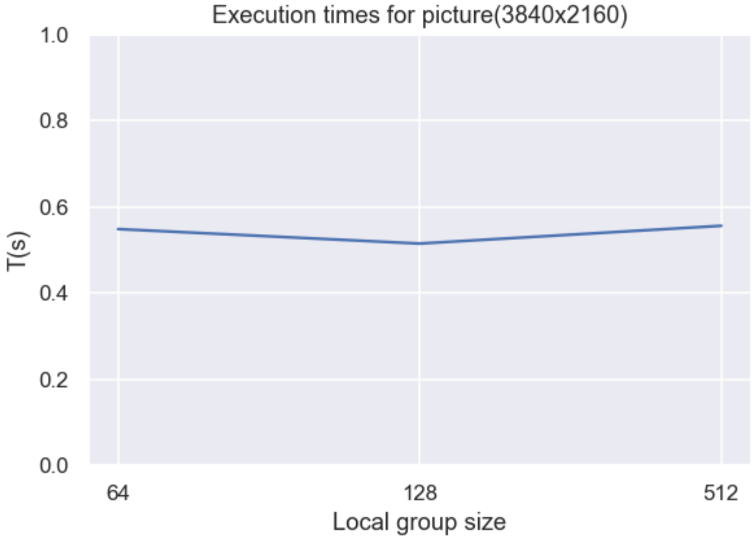
The best time for the OpenMP algorithm is the one with the biggest number of threads, which is intuitive because the dimensions of the picture are large, and the workload is evenly distributed between the treads.



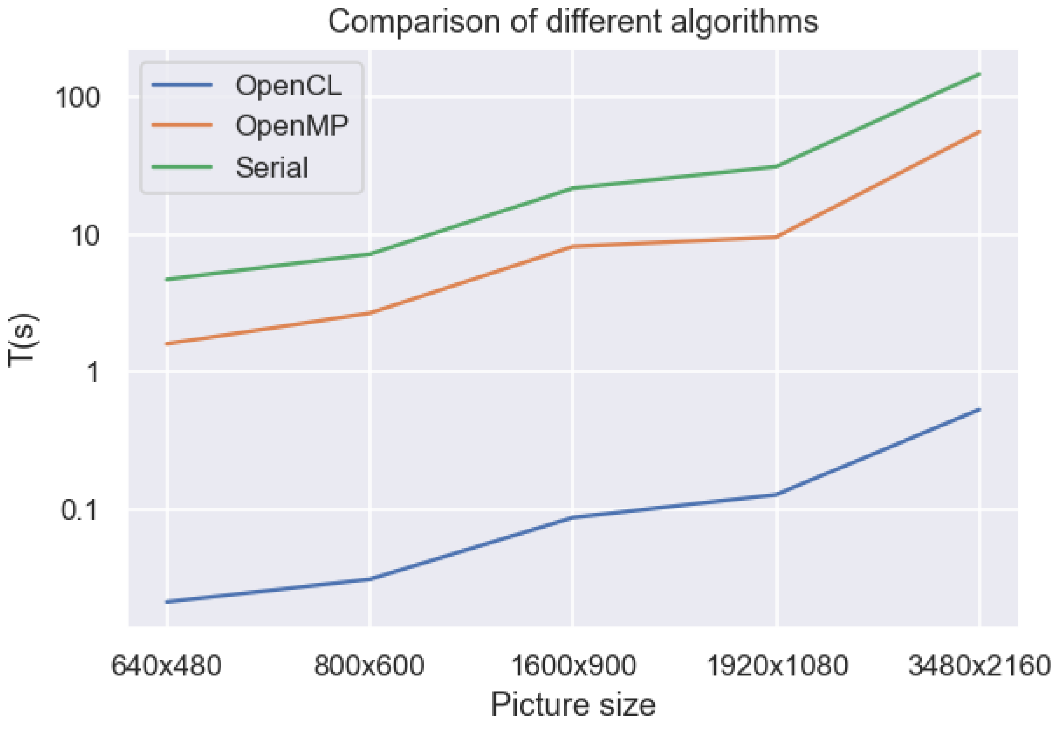
Another interesting optimization is the usage of the local memory in the OpenCL algorithm. As explained above, by using local memory we get better results (shown in the plot below).



We also tried experimenting with different local group sizes, but the results did not vary that much and were nearly the same.



At last, we compare the different algorithms and can see that the best approach is OpenCL, which is expected. OpenMP has numerous critical sections and needs further optimization. We could get better results with OpenMP if we can somehow implement reduction in the algorithm.



6. Conclusion

With this project assignment I applied in practice the knowledge I gathered throughout this course. First, I implemented the serial algorithm. Then redesigned it for OpenMP and OpenCL. If I had more time, I would use it to optimize the OpenMP algorithm and maybe try out using the cluster approach for the GPU to see how much the results differ between the approaches.