Aleksandar Aleksandrov

ECEC 413

Box Blur Filter

Introduction

The goal of this project is to implement a parallel version of a box blur filter. A blur filter is an example of a low-pass filter. In matrix form, a 3 x 3 box blur can be written as:

$$\frac{1}{9}\begin{bmatrix}1 & 1 & 1\\ 1 & 1 & 1\\ 1 & 1 & 1\end{bmatrix}$$

The blur result is the average value of all its neighboring pixels. The "blur_filter_gold.cpp" file contains a serial version of the algorithm which is performed on the CPU. The goal of this assignment is to write a parallel version of this code on the GTX1080 GPU and compare the performance of both implementations. The serial code loops over all the pixels and then for every pixel it goes over its neighbors, finds the total blur value and divides it by the number of pixels in the corresponding tile.

Discussion

For the parallel version of the code, all of the calculation will be performed on the GTX1080 GPU. For this purpose, a few helper functions are created:

- **"allocate_matrix_on_device"** – this function gets as an argument the input image matrix and then uses "cudaMalloc" to allocate the necessary memory for the input matrix on the GPU.
- **"copy_matrix_to_device"** – after the memory is allocated, this function copies all of the values of the input matrix from the CPU to the global memory of the GPU.
- **"copy_matrix_from_device" –** this function is responsible to copy the resulting matrix from the GPU back to the CPU using "cudaMemcpy".

After the helper functions are created, the CUDA implementation first allocates memory on the GPU for both the input and output images and populates the values of the input image to the GPU.

Then the algorithm proceeds to set up the appropriate execution grid. For this assignment, a tile block of size 32x32 is used. The grid size depends on the size of the image. The following calculation sets the dimensions of the grid:

(image_size + tile_size - 1)/tile_size

This ensures that the grid covers the whole image with tiles. For example, for an image of size 8192, the algorithm will create a grid of size 256x256 tile blocks, each of which being 32x32 threads.

Then the algorithm calls the kernel "blur_filer_kernel" and supplies it with the elements of the input and output images which were allocated on the device as arguments as well as the size of the image.

The Kernel for this assignment uses only global memory. Therefore, the code is straightforward. Every thread obtains the column and row index of the pixel on which it has to operate:

Column = blockDim.x*blockIdx.x + threadIdx.x

Row = blockDim.y*blockIdx.y + threadIdx.y

Then the kernel proceeds to operate on the corresponding pixel by iterating over the horizontal and vertical neighbors and adding their values to the "blur_value" variable. For every neightbor, the "num_neighbors" variable is incremented. After the loop, the thread writes the new value of that pixel to the output matrix allocated on the device.

After all threads have done operating on the image and have written their values to the allocated output image, the devices wait on each other "cudaDeviceSynchronize()" and then the resulting matrix is copied from the device to the output matrix on the CPU.

Speedups

Since the GPU provides with a thread for every pixel, then the amount of speed-up is significant if we disregard the communication between the CPU and the GPU.

Image Size	CPU performance	GPU – including CPU-CPU communications	GPU kernel performance
1024x1024	0.038365s	0.141826s	0.000067s
2048x2048	0.129833s	0.149121s	0.000069s
4096x4096	0.529893s	0.181356s	0.000084s
8192x8192	1.812001s	0.420638s	0.000080s

From the results above, we can see that we get significant performance improvements for all image sizes if we measure the kernel's performance. This shows the advantage of performing such operations on a GPU over the CPU. However, if we include the CPU – GPU communication – transfer and allocation of data on the device, the improvements become less significant. This is due to the major bottleneck of memory transfer between the CPU and the GPU. Even with this bottleneck, however, the GPU outperforms the CPU over 3 times for an image size of 8192. We can also see that for small image sizes, the significant cost of transferring data to the GPU outweighs the performance improvement gained from the GPU – the kernel performs fast but the over performance is lower than the CPU implementation.