COMP 529 **Shukhrat Khuseynov (0070495)** **28.04.2020**

Assignment 2 (Report)

The CUDA implementations were developed for the image denoise problem. All the parallel versions specified in the assignment description were completed and saved with relevant names: noise_remover_v1.cu, noise_remover_v2.cu, and noise_remover_v3.cu for the three required versions. Two additional optional versions namely noise_remover_v4.cu and noise_remover_v5.cu were included in the submission folder to optimize the first computation kernel and reduction kernel, respectively, both using shared memory. Similarly, each version was implemented on top of the previous one. Consequently, CUDA 10.1 on be01 node with tesla k20m GPU were employed in the cluster jobs. The comparison and best version in terms of Gflops rate will be discussed in the performance study section below [vers2 – 1.947 Gflop/s].

The first, naïve, version consisted of converting the computation part of the serial code to GPU parallel programming using Cuda, primarily requiring the correct allocation of the memory for the variables, host to device transfers and vice versa. Moreover, three main parts: reduction, computation 1 and computation 2 were rewritten as device functions. The shared memory was not used as stated in the assignment description; therefore, the sum reduction involved atomicAdd function to sum up the elements without data races. Other kernels were quite similar to the serial code just without loops, since two dimensional thread blocks resemble its operations in parallel.

In the second version, many repeated variables, especially in the computation 1 and 2 kernels, were replaced with an according temporary variable to reduce the data transfers from the global memory, which are slow.

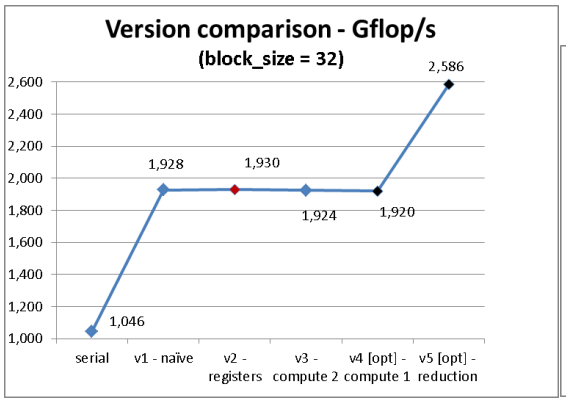
In the third version, the shared memory of thread blocks was used to optimize the compute 2 kernel. Smaller tiles of whole input data were loaded for each thread block with a creation of ghost cell (halo) layers at the end of x and y dimensions, since computations required not only elements of current position but also one step forth for both dimensions. After loading the input data, the synchronization was utilized before proceeding to the similar computations as before.

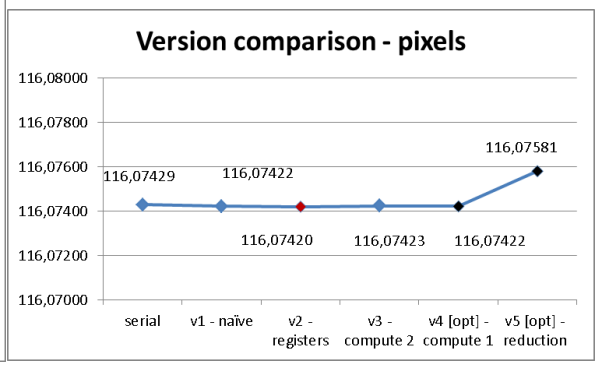
Additionally, my two optional versions were aimed to optimize the two remaining kernels as well, using shared memory. The optimization of compute 1 kernel, the fourth version, was similar to compute 2, just required more operations and few difficulties, such as accessing all four directions in the input data, similar to 2d stencils example in the class, being resolved with halo layers.

In the fifth version, the optimization of reduction kernel was also similar to sum reduction examples that we studied, just two dimensional here: firstly, all sums are calculated within each array of each thread block in one dimension; then, the sums are summed in another dimension, resulting in single sum for each thread block. To sum all final summations, atomicAdd was deployed. In addition, all four previous versions had same floating operations

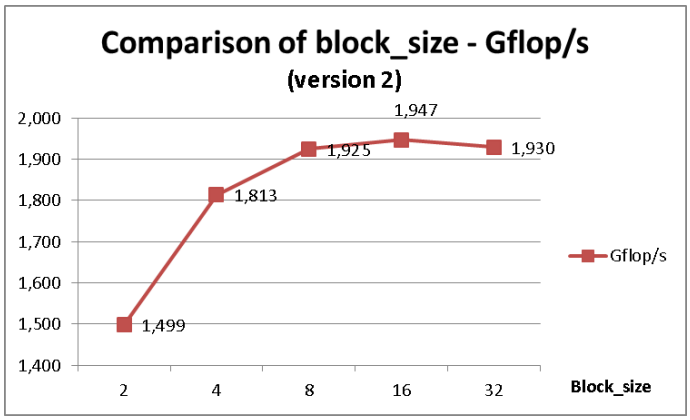
and, therefore same formula for Gflops rate. Since reduction in Cuda is a tricky method, the more relevant numbers for floating operations were considered in its formula.

The correctness of these versions was ensured by checking the average pixels of given example of coffee.pgm image, compared to the result of its serial implementation. All but my fifth version were quite accurate (see the plot of pixels below). The fifth version was also close enough but not all 4 decimal points, its inaccuracy might be due to Cuda floating operations, which are less accurate than CPU's or some small error from my side might be present. Here are the plots of performance studies:





It can be seen that among the required 3 versions in the assignment, slightly the fastest in terms of Gflop/s rate is the second version with the registers usage. Among all proposed, the fifth version seems to be the fastest (in terms of time as well), but it has some inaccuracy, which has to be either explained or corrected.



Moreover, after tuning block sizes for the second version (see the upper plot), the optimal tile dimension or block size is 16. So, the assumed winner among 3 versions is the second one with 16 block_size parameter, which has the Gflop/s rate almost as twice as the serial implementation (the numbers can be seen on the plots, 1.947 Gflop/s here).

In conclusion, it was a very interesting assignment for learning and applying the Cuda parallel implementations. We can see here some optimization techniques which sometimes are more useful and sometimes are less, but as a result GPU calculated the computation part twice as faster as CPU, which is fascinating.