Parallel and Distributed Systems 3rd Assignment

Antonios Ourdas | AEM: 9358 | email: ourdasav@ece.auth.gr

In this assignment we are asked to implement a non-local means filter for removing noise from an image and improve performance using CUDA to run the process in an NVIDIA GPU. The structure of this image processing algorithm fits the GPU architecture and therefore we can achieve an important speed up of the process. Our implementation is structured in 3 code versions each achieving better performance than the previous one:

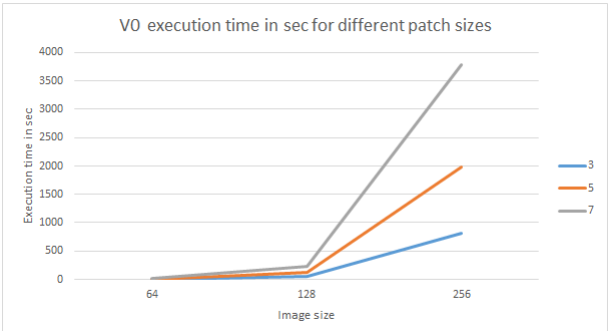
- **V0:** Serial implementation
- **V1:** Parallel implementation with CUDA using only global memory
- **V2:** Parallel implementation with CUDA using shared memory to reduce the amount of reads from global memory and to further speed up the process

Further details and results from simulations run on AUTH HPC facility are given below.

V0: Serial Implementation

In V0 we implement the non-local means algorithm using the well-known Lenna image, an MRI scan brain image, a woman image and a ship image all in 64, 128 and 256 sizes. The images were converted in csv format text files using MATLAB and are used as an input to our C program, which then writes the output images (noisy, denoised and residual) in the same csv format. The csv files are processed by MATLAB using the provided script Read_Image.m to view the results. Our code reads the input image, creates gaussian kernel patch, and pads the image edges symmetrically to fit patches on the edges. Then all pixels of denoised image are calculated using the given formula. Execution times for Lenna image executed on the AUTH HPC facility are reported below (similar times where observed for other images):

Image size	Patch size	Execution time in sec
64	3	3.135921
	5	7.806028
	7	14.088401
128	3	49.489246
	5	128.490104
	7	223.726115
256	3	814.142663
	5	1989.724481
	7	3788.934946

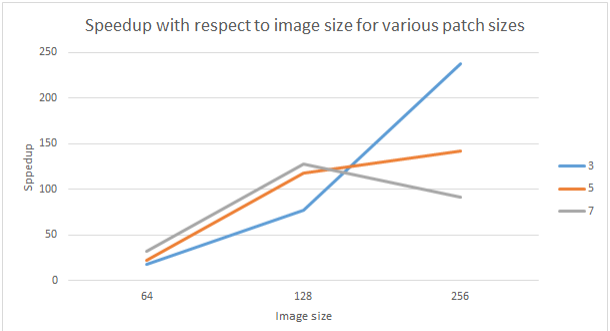


The previous diagrams show the time complexity of non-local means which is $O(N^4)$ for an $N \times N$ image, since we have to search all other patches for every pixel of the image. Also, the patch size has a less significant impact on time than image size ($O(w^2)$ complexity). An example of 256x256 Lenna image before and after denoise using 7x7 patches is shown below (more of them are available on git repository):

V1: Parallel implementation with CUDA using only global memory

In V1 we implement the parallel implementation of the above algorithm using CUDA to run our code on an NVIDIA GPU. At first the CPU creates a gaussian kernel and calculates all the image patches which are then loaded in the GPU memory. One thread is called for every pixel of our image which calculates the final denoised image value using the provided formula (using the pre-calculated patches). We create n blocks with n threads per block to cover all of our image (according to our input values the threads per block will always be a multiple of 32 which is recommended when launching kernels in GPUs). After the kernels have finished the GPU data are transferred back to the CPU. Execution times for Lenna image executed on the AUTH HPC facility are reported below (similar times where observed for other images):

Image size	Patch size	Execution time in ms
64	3	173.450882
	5	341.157532
	7	429.53833
128	3	636.756348
	5	1092.486328
	7	1746.861816
256	3	3424.910645
	5	13963.35547
	7	41308.44531



The previous diagrams show the effect of image and patch size on execution time. The relative speed up increases with size only for patch size 3, while for the other there is not an increase in size. This is probably due to the increased size of the patch cube as the patch size increases so more time is spent transferring data between CPU and GPU. Also execution time increases with patch size (same as V0).

V2: Parallel implementation with CUDA using shared memory

In V2 we take advantage of GPU faster shared memory between threads of the same block to deal with patches overlap which results to unnecessary reads from global memory. Here we don't construct the patch cube like in V1 but instead only copy the image in the GPU and the gaussian kernel. At each block of threads only one thread transfers image pixels in blocks from global to shared memory, while other threads wait for transfer to be complete. Then each thread finds the desired weights without having to load every patch from global memory, but instead reading them from the image block which is now available to the fast shared memory. The process is repeated until all image blocks are accessed. The number of threads per block is 256 for all images sizes and the image is blocked into 16x16 blocks of pixels (actual size is (16+(w-1))*(16+(w-1)) including padded edges). The above has been implemented in V2 source code but wasn't properly tested due to some undefined behavior. The for loops indexes should follow the above description. One last thing to note is that the gaussian which is loaded in the global memory is loaded once in the shared memory of each block of threads which is then accessible to all threads.

The source code with input/output images, MATLAB script used for parsing images, figures and diagrams generated are available on the following GitHub repository:

https://www.github.com/AntoniosOurdas/Parallel-and-Distributed-Systems-3rd-Assignment.git