Non-Local Means for Image Denoising using CUDA

Theodoros Katzalis 9282, Filosidis Stavros 9456

Parallel & Distributed Systems @ ECE AUTh

Abstract

Image denoising is a heavily studied field in computer science, with methods applied varying from simple blurring convolutions to neural networks. **Non Local Means** is a widely used algorithm for image denoising. It is assumed as computationally expensive and not a strictly serial task, thus a great opportunity for parallelism.

For our analysis, the basic principle of this algorithm is to calculate the new filtered pixel with a **weighted average** of all the pixels. The weight of two pixels is determined by the euclidean distance of their patches. The euclidean distance is also weighted, applying a gaussian patch, to emphasize the values of the center rather than the edges of the patch.

We represent three implementations of this algorithm. The first version (v0) that utilizes only one **CPU** core and performs the task in a serialized manner, the second version (v1) that introduces parallelism using **CUDA** with an *NVIDIA GPU (Tesla T4 Offered by Google Colab)* utilizing only the **global/unified** memory, and finally the third version (v2) that is an optimized CUDA algorithm making use of **shared memory**.

We will attempt to show the performance difference between the versions as well as the actual results of the denoising using that particular method.

1 Algorithm

Workflow, Memory usage, matlab, CPU and GPU approach! dynamic shared memory allocation

Contrary to *Local Means*, *Non-Local Means* calculates the average of all the pixels in an image, weighted with the similarity value with respect to the reference pixel. This way more details are preserved, since we are not simply blurring the image in each local pixel neighborhood.

The method relies on finding similar such "neighborhoods" across all the image and calculates the denoised one like so:

$$\hat{f}(\mathbf{x}) = \sum_{\mathbf{y}\in\Omega} w(\mathbf{x}\cdot\mathbf{y}) f(\mathbf{y}), \qquad \forall \mathbf{x} \in \Omega \tag{1}$$

where $\Omega \subset \mathbb{R}^2$ the image domain, $f : \Omega \to \mathbb{R}$ the initial noised image, and $\hat{f} : \Omega \to \mathbb{R}$ the predicted denoised image.

The weights are calculated using the formulas 2 and 3.

$$w(i,j) = \frac{1}{Z(i)} e^{-\frac{\|f(N_i)-f(N_j)\|^2_{G(a)}}{\sigma^2}} \tag{2}$$

$$Z(i) = \sum_j e^{-\frac{\|f(N_i)-f(N_j)\|^2_{G(a)}}{\sigma^2}} \tag{3}$$

with N_k denoting a square pixel neighborhood of fixed size and center pixel k.

1.1 Naïve approach - CPU

Below, we describe the simplest form of the algorithm for the serial version, however in our code we use the slightly improved version that takes the problem symmetry into consideration. This way, the total calculations are reduced in half, by saving the calculated weight for both pixels - p_i, p_j, and slightly changing the second for loop (line 5).

Algorithm 1 Non-Local Means

```
1:  procedure FILTERIMAGE(Image im, PatchSize p)
2:      for p_i in im do
3:          w ← 0                                  ▷ Weight for each patch
4:          w_sum ← 0
5:          for p_j in im do
6:              w ← PATCHDISTANCE(p_i, p_j, p)
7:              w ← e^(-w^2/σ)
8:              w_sum ← w_sum + w
9:              p_i ← p_i + p_j · w
10:         end for
11:         p_i ← p_i / w_sum
12:     end for
13: end procedure
```

2 Parallelism - GPU

In the parallel implementation, we make use of the multitudes of CUDA cores provided by the GPU, as well as two different memory levels in the GPU memory hierarchy.

In hardware, CUDA cores are split into groups inside the so called **Streaming Multiprocessors (SMs)**. Calculations are distributed into **blocks** and **threads**. Each block contains a multiple of 32 threads, called warps, which all share the same block memory called **shared memory**. Multiple blocks form a grid, which is then distributed among all the SMs, based on thread number, needed memory, and GPU capabilities.

We use the **unified memory**, and **shared memory**, provided privately to each block and having much faster speeds, analogous to RAM/Cache in a CPU. When a portion of data is used multiple times by the same threads inside a block, this portion should be copied into the much faster shared memory from the slower, global memory.

2.1 Work distribution

For our problem purposes, we chose to split the image into **multiple blocks of pixels**, and assigned each pixel inside a block to a specified thread. For example, an image with a resolution of *32x32* pixels could be split into 16 blocks, each containing 64 threads. This means that the work is distributed by groups of two image pixel rows (32x2 pixels, same as thread number).

Each **thread** is then responsible for the initially assigned pixel, and calculates the weights for every other pixel in the image. All memory accesses are performed using the unified memory.

Even though this approach is multiple times faster than the CPU based one, the main **performance bottleneck** is the memory access. To further improve the execution times, we make use of the **shared memory** inside each SM. Now, the **patches** that correspond to the pixels inside the block are first copied into the shared memory, and then accessed through it by the threads. The reason we only load patches of a block is that the shared memory is very limited, and we cannot load the whole image in it.

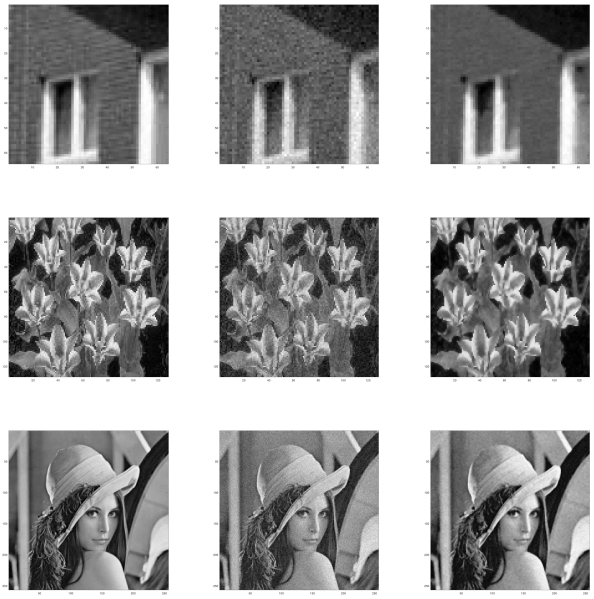
After all threads of a block finish calculating (using **synching**), the next block is loaded into the shared memory, and replaces the old one. This is done until we load all blocks of the image, and thus calculating all needed weights for the initial block.

3 Image Denoising

For these three test images, except the performance difference between our implementations, we will attempt to show the effectiveness of the algorithm by visually inspecting the results. The noise applied/added is Gaussian white with $mean = 0$ and $variance = 0,002$. Regarding the algorithm, there are two weighted functions. One for the euclidean distance between the patches and the other for the importance of pixels per patch with respect to the center. The impact of these weights is determined by their sigma values named as *filtSigma* and *patchSigma*.

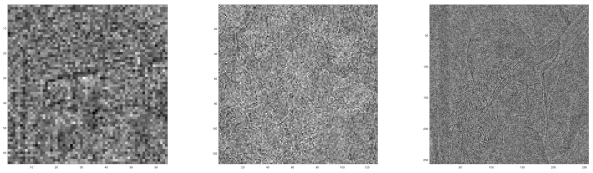
It is worth mentioning that decreasing the patchSigma will emphasize the centre region of the window and decreasing the filtSigma will increase the denoising effect. The patch size, another internal parameter that determines the region that we test the similarity between the pixels, could potentially influence the performance. For our datasets, we couldn't spot a significant visual improvement but in general it is assumed that a larger patch size is more suitable for smooth areas and a smaller patch size is better for (rough?) textured areas. Additionally, images with patterns and self-similarities are highly likely to be filtered more efficiently (see tulips residual).

For visual validation, we show additionally the residual per image that is the difference between the filtered and the noised pixels. The following experiments have been performed using these fixed values: filtSigma = 0.02, patchSigma = $5/3$ and patchSize = 3 (window 3x3).



(a) Original image **(b)** Noisy image **(c)** Denoised image

Figure 1: House (64x64), Tulips (128x128), Lena (256x256)



(a) House **(b)** Tulips **(c)** Lena

Figure 2: Residuals (filtered values)

4 Benchmarking/Results

Using the Tesla T4 GPU, equipped with **2,560** CUDA cores, we run each test 5 times to avoid and take into account any runtime variations that may occur.

4.1 Benchmarking Methodology

CPU based implementation was run for each patch size and dataset, and the other two parallel CUDA implementations were run using a combination of grid size (powers of 2) from 16 to 1024 and thread numbers (multiples of 32 - warps) from 32 to 1024. However, not all runs are displayed in the final benchmarking plots due to hardware limitations concerning limited shared memory size and maximum thread count or grid size.

For our testing, we used three grayscale images with resolutions of 64x64, 128x128 and 256x256 as seen in fig. 1 and each one of them is represented as a 2D array. For each test image and window (patch) sizes of 3x3, 5x5 and 7x7, we test the performance of the aforementioned algorithm versions.

4.2 Results

A summarized version of the results can be seen in table 1. This contains the best times for each combination of parameters, which can vary based on the number of total CUDA cores,

	64x64			128x128			256x256		
	3	**5**	**7**	**3**	**5**	**7**	**3**	**5**	**7**
v0	0.3172	0.6117	1.0348	4.9412	9.6417	16.5384	78.1390	154.8578	261.8866
v1	0.0027	0.0083	0.0129	0.0147	0.0647	0.1872	0.1991	1.0617	2.9879
v2	**0.0024**	**0.0052**	**0.0086**	**0.0132**	**0.0214**	**0.0478**	**0.1053**	**0.2969**	**0.6501**
$t_{v0/v1}$	117.48	73.69	80.21	336.13	149.02	88.34	**392.46**	145.85	87.64
$t_{v0/v2}$	132.16	117.63	120.32	374.33	450.54	345.99	**742.06**	521.58	402.84
$t_{v1/v2}$	1.12	1.59	1.5	1.11	3	3.91	1.89	3.57	**4.59**

Table 1: Execution Time (s)

GPU architecture and much more. The most important results are the relative speedups. A more detailed view of the effect of different thread count and grid count is also displayed in figures 3 to 8.

We observe a huge improvement when comparing the CPU version to the GPU one, reaching a maximum speedup of **392** for unified memory, **742 times** for shared memory implementations.

For the relative speedup between the v1 and v2 variations, we achieve a maximum speedup of **4.59 times**, when using shared memory.

5 Tuning performance

Regarding the performance table 1 we should mention some attempts of kernel optimization that has been integrated and produced the above results.

5.1 Kernel launch

Except the usage of shared memory as an optimization technique to improve the read and write speed, configuring the launching of the kernel regarding the number of threads (block size) and the number of blocks (grid size) plays a significant role in time execution. Things that we need to consider are the following:

- Block size should be a multiple of 32 due to the concept of warps (256 is used as a rule of thumb).
- Choose a block size that **optimizes occupancy**, which is also dependent on **register usage**.
- **Grid stride loop** for flexible kernel launching. Grid and block sizing independent of data - two threads can be assigned to more than 1 pixel (works with non-multiples of 2 for input size).
- **Increase thread load**. Sometimes a thread mapped to a single pixel will perform worse than letting threads handle more than singular pixels. Indeed this behavior is observed in our benchmarking (see figs. 3, 4, 5). *Keeping the pipeline loaded performed better.*

In order to do all of that we need to also consider the specs of the GPU that we are currently using. In newer CUDA toolkits there are a number of utilities, as well as a **spreadsheet** (provided by NVIDIA) that offer a very nice overview for determining the best block size for balanced occupancy, based on a number of factors such as the **number of registers** that are used, shared memory, threads per block, **compute capability** and CUDA version (**nvidia-smi**).

Based on this spreadsheet along with the help of appropriate nvcc flags (-Xptxas=-v) that can give us the information we want, we attempt to optimize or even maximize occupancy per SM, for optimal performance, tailored to the particular GPU we used. However, higher occupancy **does not always guarantee** faster execution times, as there are more factors like **memory speed**, **memory access patterns** and **data alignment**, **bank conflicts**, and others that should be taken into account.

Practically, a naïve approach to determine block and grid size is to launch a kernel for an image of MxN pixels, using M blocks and N threads. Another way including the aforementioned considerations is to use a block size of 256 and the grid size to be as many needed to fulfill the size of the data (rounding up).

Since we want to explore all possible ways to launch the kernel, with varying combinations of grid size and thread count per block, we run multiple variations, and plotted them compactly in section 8.

5.2 Discussion - Further optimization

When copying data to shared memory, each thread copies the data corresponding to it based on the index it was assigned. This means that each thread accesses it's own portion of data and no other thread accesses the same memory chunk, avoiding bank conflicts.

However, this results in much more memory being used, and thus to a limited total thread count, on top of more unified memory accesses. Although for our particular implementation, using overlapping portions of data would increase indexing complexity, and we decided not to explore that option.

6 Conclusions

Extraordinary speedup of the CUDA versions with respect to the serial version.

Shared memory, although still requires global memory access, still improved time execution in the long run because of data reusability. For larger images and windows, the effect of shared memory has a bigger impact, improving performance.

6.1 Speedup

Looking at the generated plots, we observe that for **v1**, increasing the **thread count** not only does not improve performance, but at most times worsens it, no matter the block count. In **v2**,

we see no such behavior, with thread count having small to no improvements in speedup.

Increasing the **block count** seems to help speedup in most cases.

Overall, increasing the number of both thread and block count seems to reach a point of **diminishing returns** (particularly in fig. 7, for the house dataset), and the best performance is observed in different combinations depending on input and patch size.

7 Output Validation

In early stages of development we tried to implement the version 0, as a starting/reference point, in such a way to produce identical results with a given reference Matlab version. For this to happen, we validated:

- Wrapped image creation
- Patch creation
- Application of the gaussian patch
- Weights of each pixel with respect to all others
- Filtered values

Ensuring a valid CPU version, we proceeded for the CUDA implementations. The heavy computational part of our algorithm was the weight calculation so this is where the kernel would be launched. For this reason, our test case between the CPU and the GPU will eventually only be the filtered values.

Due to floating point arithmetic, we assume a valid pixel when the subtraction of the corresponding filtered values between the CPU and the GPU is less than 10^{-4}.

Eventually, all three versions produced identical results with a slight difference due that floating point error. This error could also be produced due to the usage of the compiler option *(-use_fast_math)*, which uses floating point approximations as a tradeoff between speed and accuracy.

8 Appendix

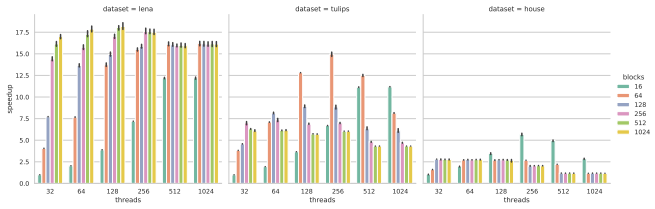


Figure 3: V1, p=3

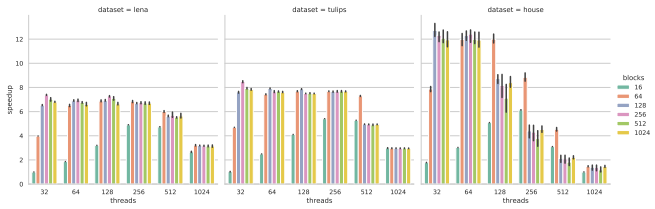


Figure 4: V1, p=5

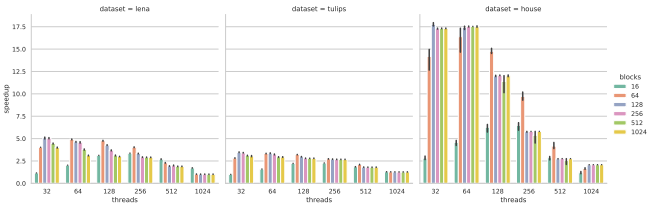


Figure 5: V1, p=7

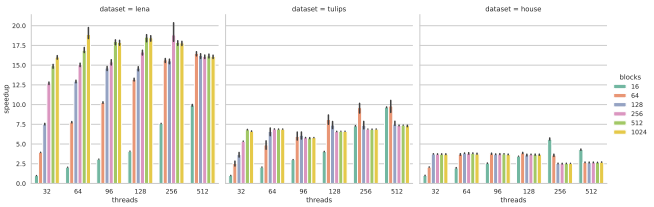


Figure 6: V2, p=3

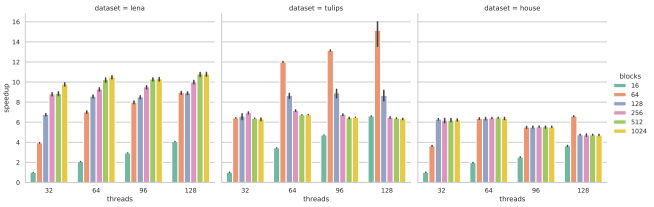


Figure 7: V2, p=5

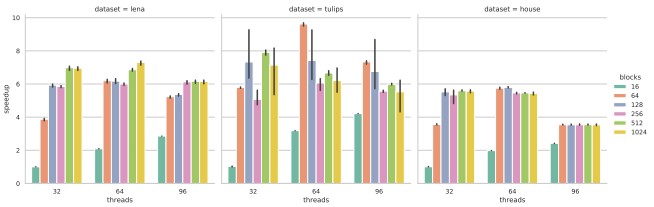


Figure 8: V2, p=7