Image Denoising

Alexander Kharlamov
akharlamov@nvidia.com

Victor Podlozhnyuk

vpodlozhnyuk@nvidia.com

September 2013

Document Change History

Version	Date	Responsible	Reason for Change
0.9	04/16/2007	akharlamov	Initial release
1.0	05/16/2007	vpodlozhnyuk	Edited CUDA-specific parts

Abstract

Image denoising algorithms may be the oldest in image processing. Many methods, regardless of implementation, share the same basic idea – noise reduction through image blurring. Blurring can be done locally, as in the Gaussian smoothing model or in anisotropic filtering; by calculus of variations; or in the frequency domain, such as Weiner filters. However a universal "best" approach has yet to be found.

NVIDIA Corporation
2701 San Tomas Expressway
Santa Clara, CA 95050
www.nvidia.com

Motivation

White noise is one of the most common problems in image processing. Even a high-resolution photo is bound to have some noise in it. For a high-resolution photo a simple box blur may be sufficient, because even a tiny features like eyelashes or cloth texture will be represented by a large group of pixels. Unfortunately, this is not the case with video where real-time noise reduction is still a subject of many researches. However, current DirectX 10 class hardware allows us to implement high quality filters that run at acceptable frame rates. Starting from GeForce 8 series graphic cards we can also benefit from using CUDA – a general purpose GPU programming system. Features such as shared memory and sync points combined together with flexible thread control allow us to speed up algorithms considerably.

How It Works

The main idea of any neighborhood filter is to calculate pixel weights depending on how similar their colors are. We describe two such methods: the *K Nearest Neighb*ors and *Non Local Means* filters

K Nearest Neighbors Filter

The K Nearest Neighbors filter was designed to reduce white noise and is basically a more complex Gaussian blur filter. Let **u(x)** be the original noisy image, and $\mathbf{KNN_{h,r}u(x)}$ be the result produced by the **KNN** filter with parameters **h** and **r**. Let Ω**(p)** be the spatial neighborhood of a certain size surrounding pixel **p**. We will consider it to be a block of pixels of size **NxN**, where **N = 2M + 1** – so that **p** is the central pixel of Ω**(p)**. Then let

$$\mathrm{KNN}_{h,r}u(x) = \frac{1}{C(\mathbf{x})}\int_{\Omega(\mathbf{x})} u(\mathbf{y})e^{-\frac{|\mathbf{y}-\mathbf{x}|^2}{r^2}}e^{-\frac{|u(\mathbf{y})-u(\mathbf{x})|^2}{h^2}}d\mathbf{y}$$

where C(x) is the normalizing coefficient.

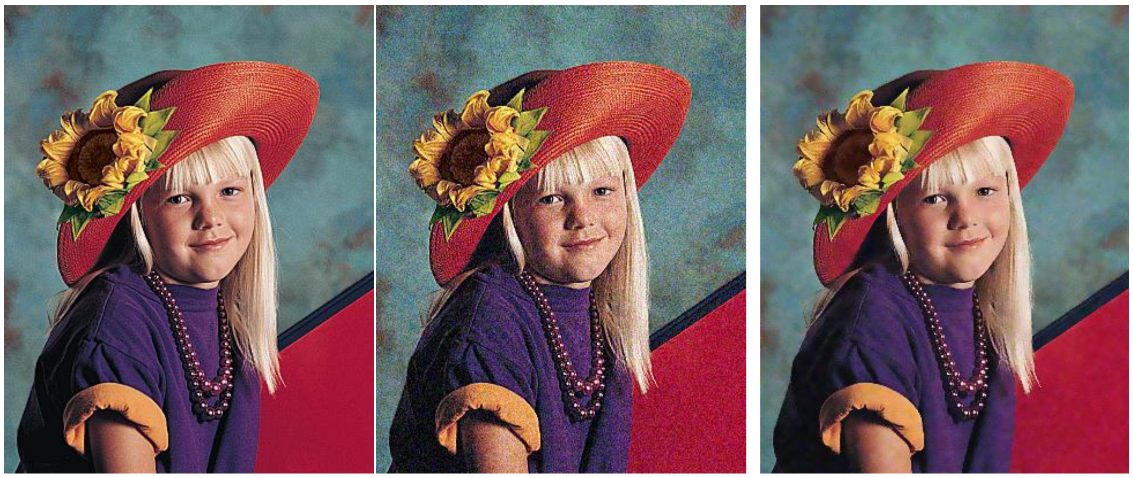
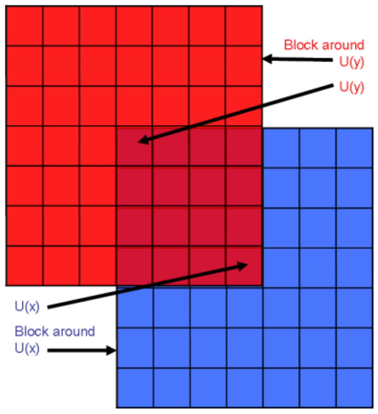


Figure 1. Original, Noisy and KNN Restored Picture

Non Local Means Filter

The Non Local Means filter is a more complex variation of the KNN filter. Using the same notation as for KNN, let $\mathbf{NLM_{h,r,B}u(x)}$ be the restored image, Let **B(q)** be the spatial neighborhood of a certain size surrounding pixel **q**. We will consider it to be a block of pixels of size **KxK**, where **K = 2L + 1** – so that **q** is the central pixel of **B(q)**. Then let

$$\mathrm{NLM}_{h,r,B}u(\boldsymbol{x}) = \frac{1}{C(\mathbf{x})}\int_{\Omega(\mathbf{x})} u(\mathbf{y})e^{-\frac{|\mathbf{y}-\mathbf{x}|^2}{r^2}}e^{-\frac{ColorDistance(\boldsymbol{B}(x),\boldsymbol{B}(y))}{h^2}}d\mathbf{y}$$



where C(**x**) is the normalizing coefficient, and

$$ColorDistance(\mathbf{B}(\mathbf{x}),\mathbf{B}(\mathbf{y})) = \frac{1}{S(\mathbf{B})}\int_{\mathbf{B}(\mathbf{x})} |u(\mathbf{y}+(\mathbf{x}-\boldsymbol{\alpha}))-u(\boldsymbol{\alpha})|^2\, d\boldsymbol{\alpha}$$

where **`S(B)`** is **B**'s area. Thus **`ColorDistance(B(x),B(y))`** represents a normalized sum of the absolute differences between blocks around pixel **`u(x)`** and around pixel **`u(y)`**.

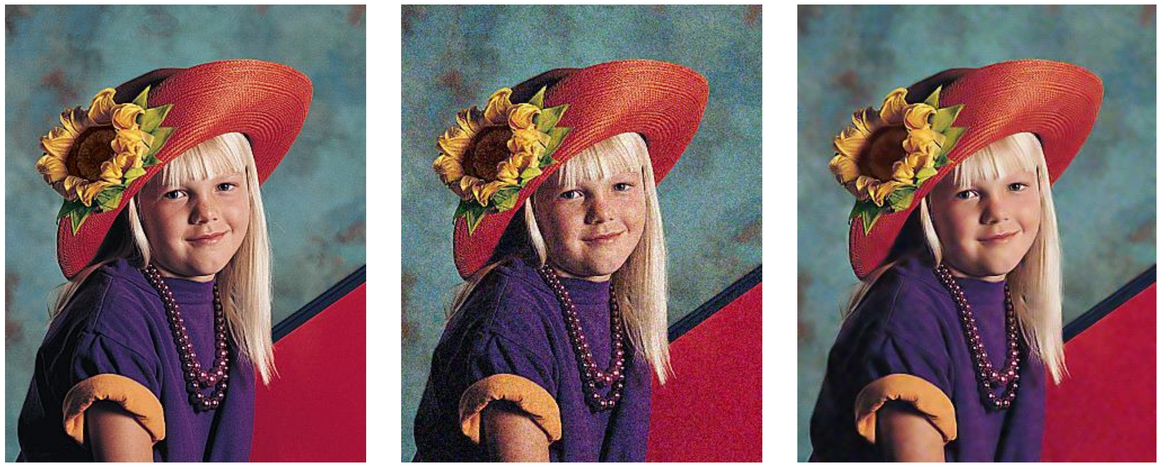


Figure 2. Original, Noisy and NLM Restored Picture.

Note: NLM can even fix some flaws in original images (Figure 4)

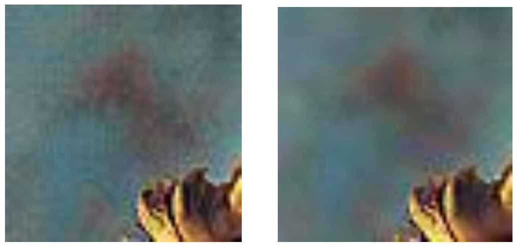


Figure 3. Original and NLM Restored Picture.

Speeding up NLM.

Whereas KNN runs in real-time (~500 fps on 8800 GTX) NLM is much slower. The main reason for that is the great amount of texture fetches:

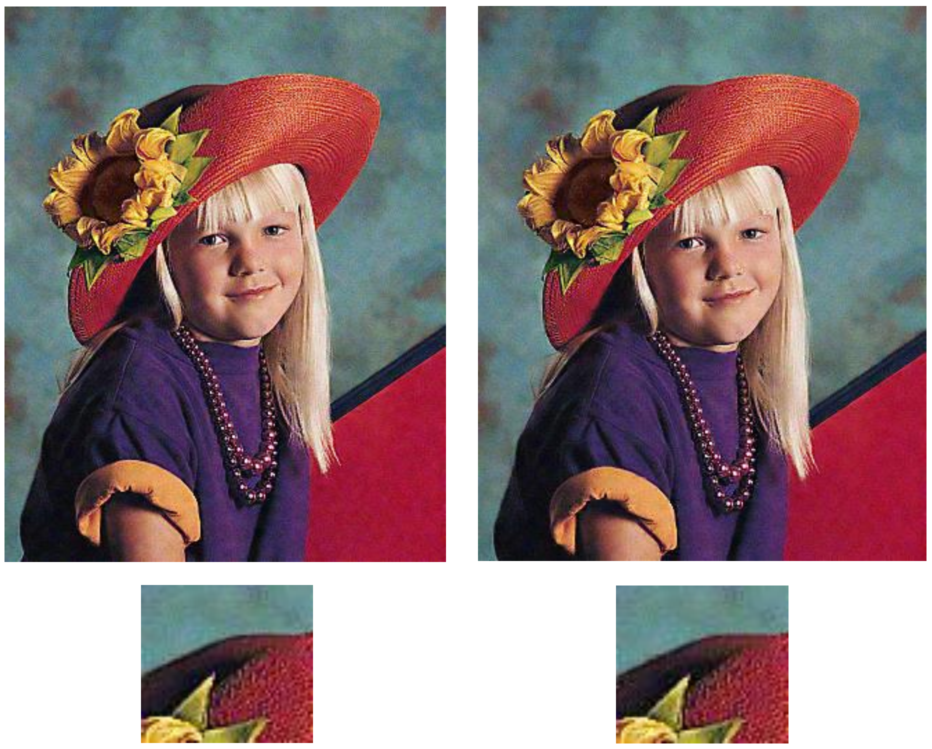
- For every pixel $\mathbf{N}^2$ number of weights need to be count where (**`N = 2M + 1`** then **`M`** = Window Radius).

- To calculate each weight additional $\mathbf{K}^2$ number of weights need to be count where (**`K = 2L + 1`** then **`L`** = Block Radius) texture fetches are done to calculate *ColorDistance()* function.

Basically an O($\mathbf{N}^2 * \mathbf{K}^2$) number of fetches is done and only reducing that number will increase performance.

The proposed solution is to assume that within every block weights do not change. Thus we calculate weights for central pixel only and use these weights as convolution coefficients. That way the number of texture fetches is reduced to O($\mathbf{K}^2$). Considering that the most common values for **`N = 7`** we make 49 times fewer texture fetches.

The assumption that weights are uniform within a block is fairly true. Most smooth areas are restored with no visual difference from original NLM. However areas with edges can be restored with artifacts.



Original NLM Image Quick NLM Image

Figure 5. Difference between NLM and quick NLM method.

Quick NLM can be used in combination with KNN. The combination of their results will be more accurate than the individual results, and still faster than the original NLM implementation.

Choosing Parameters

There is always the question of choosing the best parameters for KNN and NLM. As evident from the previous equations the weights for surrounding pixels depend on the following parameters

- **`r`** – can be considered to be a traditional Gaussian blur coefficient. If then KNN transforms into Gaussian blur. As with Gaussian blur **`r`** should be equal to **N**.
- **h** – is a lot more tricky to choose. In fact, the best way to select h is to make it user defined as estimation of quality of the image is highly subjective.
- The size of **Ω** depends on the size of the image, but good visual results are usually achieved with a 5x5 or 7x7 blocks of pixels. We have chosen 7x7 for our CUDA implementation.
- NLM has one additional parameter – the size of **B**. We have chosen 7x7 for our CUDA implementation.
- Quick NLM has one additional parameter – the block of pixels that share weights. This parameter is crucial in speeding NLM. We have chosen to share weights among a block of 8x8 pixels. This approach increases performance by almost 18 times but introduces some minor artifacts.

Implementation Details

Both methods can be easily implemented on CUDA. The noisy picture is loaded as a texture and the filter is implemented in a CUDA kernel. The restored image is rendered into a PBO. Quick NLM (NLM2) uses shared memory to precompute pixel weights. Every thread calculates a single weight and stores it into a shared array. After all threads are synced, they use the same weights to average pixels within the block.

Running the Sample

KNN has the following parameters that can be altered by the user:

- Noise Level corresponds to **h** in the formula for $\mathbf{KNN}_{h,r}\mathbf{u(x)}$ and $\mathbf{NLM}_{h,r,B}\mathbf{u(x)}$.
- Gaussian Sigma is a traditional Gaussian blur coefficient. It corresponds to **`r`** in the formula for $\mathbf{KNN}_{h,r}\mathbf{u(x)}$ and $\mathbf{NLM}_{h,r,B}\mathbf{u(x)}$.
- Lerp Coefficient, Weight Threshold and Counter Threshold are used in a simple modification:

  After the weight of a pixel is calculated, it can be compared to **`KNN_WEIGHT_THRESHOLD`**. The percentage of weights that are greater than **`KNN_WEIGHT_THRESHOLD`** is accumulated into the **`fCount`** variable. The working range for **`KNN_WEIGHT_THRESHOLD`** is (0.66f, 0.95f).

  Once all the weights are calcualated, and a restored pixel has been computed, we blend between the original and restored pixels. The blending quotient **`lerpC`**, has working range [0.00, 0.33]. A simple check determines if the block is "smooth": if **`fCount`** exceeds **`KNN_LERP_THRESHOLD`** (having typical value of 66%), the block is considered to be smooth enough, so filtered pixel value should have more weight in the output than the original (noisy) input pixel value. Otherwise the block is presumed to contain edges or small features, and in order to give more weight to the original input pixel value in the output and retain important visual image properties, **`lerpC`** is reverted.

```
float lerpQ = (fCount > KNN_LERP_THRESHOLD) ? lerpC : 1.0f - lerpC;
clr.x = lerpf(clr.x, clr00.x, lerpQ);
clr.y = lerpf(clr.y, clr00.y, lerpQ);
clr.z = lerpf(clr.z, clr00.z, lerpQ);
```

- Window Radius determines the size of Ω. If Ω is a block of pixels of size **`NxN`**, where **`N = 2M + 1`** then **`M`** = Window Radius.

NLM has one additional parameter:

- Block Radius determines the size of **`B`**. If **`B`** is a block of pixels of size **`KxK`**, where **`K = 2L + 1`** then **`L`** = Block Radius.

Performance

Traditionally, image filtering is much faster on the GPU than on the CPU. Both KNN and NLM are not exceptions.

On a sample 320 x 408 image on GeForce 8800GTX all the three modifications run in real time:

- KNN: 500 fps;
- NLM: 26 fps; much slower than KNN, due to increased texture fetch pressure.
- Quick NLM (NLM2): ~470 fps, almost as fast as KNN.

The computational complexity of all the three modifications grows proportionally to the image size.

Original Image	Noisy Image

Figure 6. Noise in an Image

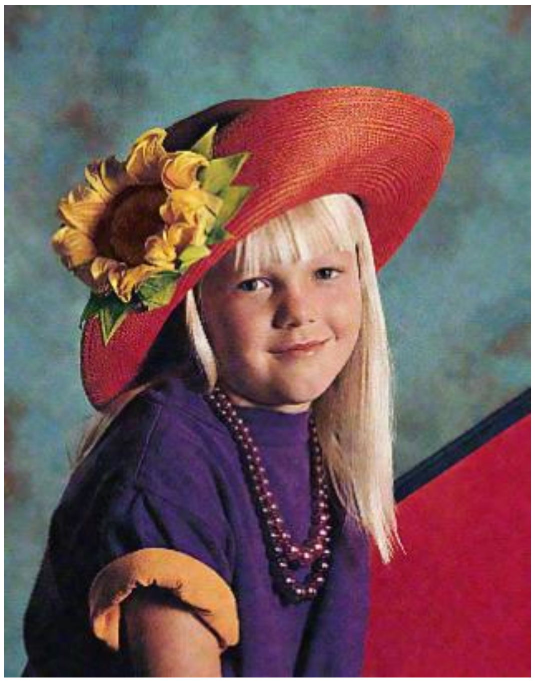
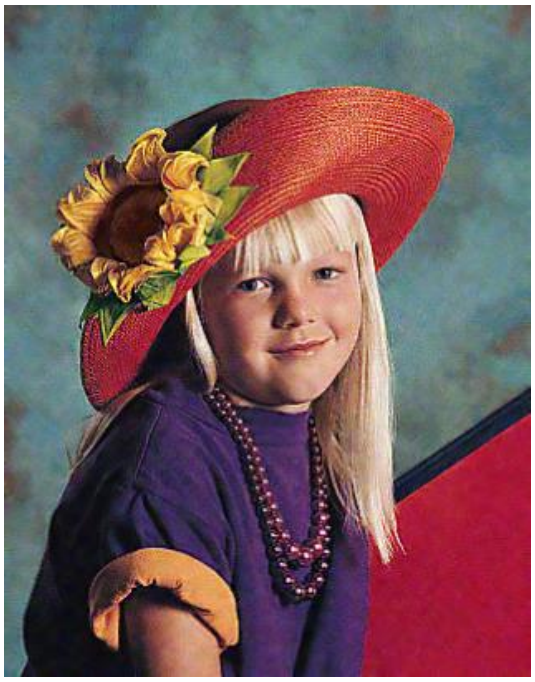
	

Image restored with KNN, runs at ~ 500 FPS with *Window Radius = 3*	Image restored with NLM, runs at ~ 26 FPS with *Window Radius = 3* and *Block Radius = 3*

Figure 7. Restored Images

References

[1] A. Buades, B. Coll, and J. Morel. *"Neighborhood Filters and PDE's"*. Technical Report 2005-04, CMLA, 2005.

[2] L. Yaroslavsky. *"Digital Picture Processing - An Introduction"*. Springer Verlag, 1985.