Real-Time GPU NLMeans for HD Video

Ian Romanick *

Abstract

The non-local means filter (NLMeans) is an effective tool for removing noise from still images and video sequences. However, it remains computationally expensive. Previous work utilized commonly available GPUs and algorithmic improvements to achieve real-time performance on standard definition (480p), grayscale video content. This paper presents extensions to that work for high definition (720p and 1080p), color video content.

1 Introduction

A variety of techniques have been developed for removal of noise artifacts in still images and video. One popular technique is the non-local means (NLMeans) filter. Previous publications [Buades et al. 2005] have shown this technique to be effective for the removal of many types of common artifacts in a variety of images. While this technique is popular, it has a relatively high computational overheard.

For each pixel in the source image, the NLMeans filter compares the $[-A..A] \times [-A..A]$ region immediately surrounding the source pixel with the same-sized region surrounding each pixel within a $[-B..B] \times [-B..B]$ search window around the source pixel. The similarity between the two regions is used to generate a filter weight for the pixel in the search window. Once the weights for all pixels in the search window have been calculated, the resulting kernel is applied to the pixels in the search window to generate the filtered value of the source pixel.

For video, the search window may be extended to include previous or future frames. In this scenario, the search window is $[-B..B] \times [-B..B] \times [t_{past}..t_{future}]$. While this spatial search window is often symmetric, the temporal search window may not be. Considering future frames would add undesirable lag while denoising live video streams, for example, so t_{future} may be 0 even when t_{past} is not.

Formally speaking, the filter is

$$X(p) = \frac{\sum_{q \in \delta} w(p, p+q) Y(p+q)}{\sum_{q \in \delta} w(p, p+q)}$$

$$w(p, p+q) = g\left(\sum_{(\Delta x, \Delta y) \in [-A..A]^2} \parallel r_{p,q}^{(\Delta x, \Delta y)} \parallel^2 \right)$$

$$r_{p,q}^{(\Delta x, \Delta y)} = Y(p_x + \Delta x, p_y + \Delta y) - Y(p_x + q_x + \Delta x, p_y + q_y + \Delta y)$$

where g is a function that remaps distances to weights. Numerous popular remapping functions exists. One function that is used elsewhere[Goossens et al. 2008] and in this experiment is the Bisquare robust weighting function.

$$g(r) = \begin{cases} \left(1 - (r/h)^2\right)^2 & r \leq h, \\ 0 & r > h \end{cases}$$

where h is a tunable parameter.

The project set out with the goal of implementing NLMeans for real-time processing 1080p video sequences on an off-the-shell integrated GPU. To set reasonable bounds, $t_{future} = t_{past} = 0$, a 5×5 search window ($B = 2$), and a 5×5 region size ($A = 2$) was set. The GPU selected was an Intel HD Graphics 4000.

The remainder of this paper is divided into four sections. Section 2 summarizes the previous work on which this study is based. Section 3 presents a series of new optimizations implemented during this study. Section 4 summarizes the preformance results of the study. Finally section 5 concludes the paper and suggests some areas for future work.

2 Previous Work

Previous work optimizing NLMeans for GPUs focused on eliminating algorithmic redundancies[Goossens et al. 2010]. Two key observations led to their algorithmic improvements.

- Two neighboring pixels, (p_x, p_y) and $(p_x + 1, p_y)$, each utilize $Y(p_x + 1 + \Delta x, p_y + \Delta y) - Y(p_x + q_x + 1 + \Delta x, p_y + q_y + \Delta y)$.
- Two pixels separated by q, p and $p - q$, utilize $Y(p) - Y(p - q)$ and $Y(p - q) - Y(p - q + q)$.

To take advantage of these observations, the Goossens et al. convert the weight summation portion of the algorithm to a separable filter. They perform a separate pass for each possible value of q. In each pass, they perform four sub-passes. The first three sub-passes are:

- Calculate $\parallel r_{p,q}^{(0,0)} \parallel^2$.
- Calculate $\sum_{\Delta y \in [-A..A]} \parallel r_{p,q}^{(0,\Delta y)} \parallel^2$
- Calculate $g\left(\sum_{\Delta x \in [0..A]} \left(\sum_{\Delta y \in [-A..A]} \parallel r_{p,q}^{(0,\Delta y)} \parallel^2\right)\right)$

Notice that the final summation only covers half the range. The final pass takes advantage of the symmetry in distance calculations. If the output of the third sub-pass is an image K, the final sub-pass calculates

$$X(p) = X(p) + \begin{cases} K(p)I(p+q) & q = 0, \\ K(p)I(p+q) + K(p-q)I(p-q) & \text{otherwise} \end{cases}$$

[Goossens et al. 2010] use single compontent, 16-bit floating-point surfaces for the results of all sub-passes.

3 Additional Optimizations

While previous work focused on algorithmic optimizations, this work has focused on implementation micro-optimizations. The primary focus has been reducing the number of passes, and a secondary focus has been reducing overall memory bandwidth requirements.

Even with the previous optimizations, generating the filter weights for a 5×5 search window requires 15 passes, and each

*e-mail: idr@cs.pdx.edu

pass requires four sub-passes. Further optimization is possible by reducing the number of passes. The number of passes can be halved by modifying the first sub-pass to calculate $\| r_{p,q}^{(0,0)} \|^2$ and $\| r_{p,q+1}^{(0,0)} \|^2$. Each of these is output to separate components of a two component surface. The modified sub-pass generates two weights from three texture accesses compared to four texture accesses previously (the value of $I(p)$ is re-used).

The second and third sub-passes are modified to operate on pairs of values. Since the values are stored in separate components of a single image, both values are read using a single texture look-up. While the total memory bandwidth is the same, each access to the texture unit, even for cache-hot accesses, carries significant overhead.

The final pass is also modified, but the modifications are somewhat more complex. $K(p)$ stores the weights for the pixels at $I(p+q)$ and $I(p_x+q_x+1, p_y+q_y)$. For the symmetric portion of the of the weight calculation, $K_x(p-q)$ stores the weight for the pixel at $I(p-q)$, and $K_y(p_x-q_x-1, p_y+q_y)$ stores the weight for the pixel at $I(p_x-q_x-1, p_y+q_y)$. The modified final sub-pass accumulates two weights using seven texture accesses compared to eight texture accesses previously (a single access of $K(p)$ supplies two weights).

It is possible to extend this optimization to generate three weights, (p_x, p_y), (p_x+1, p_y), and (p_x+2, p_y) per pass. As before, this reduces the total number of passes, and saves a single texture access in the first sub-pass (3 weights for 4 accesses) and the final sub-pass (3 weights for 10 accesses).

Extending to four or more weights per pass poses challenges. For a 5×5 search window, the added weight cannot come from (p_x+q_x+3, p_y+q_y) as this lies outside the search window. It is also not possible to use (p_x+q_x, p_y+q_y+1) because the symmetric weight value, $K(p_x-q_x, p_y-q_y-1$, does not contain a weight involving $I(p)$. The symmetric weight is not available anywhere in K. The symmetric weight is only available in K if both (p_x+q_x, p_y+q_y+n) and (p_x+q_x, p_y+q_y-n) are calculated per pass.

Simply extending the mask to (p_x+q_x+1, p_y+q_y), (p_x+q_x+2, p_y+q_y), (p_x+q_x, p_y+q_y+1), and (p_x+q_x, p_y+q_y-1) seems plausible, but the irregular shape of the mask makes covering the entire $[0..2] \times [-2..2]$ search window difficult. Multiple masks would need to be used to cover the full search window, and not all of the masks would be able to generate four weights.

The most plausible extension of the technique is to cover the entire search window in a single pass. This would generate 15 weights per pass. This would require four sepearate render targets. This may carry some additional cache penalties, and this is left as future work.

As a final optimization, the format temporary surfaces was changed from 3-component, 16-bit floating-point[1] to 3-component, 10-bit normalized integers[2]. The distances generated by the first sub-pass have a maximum magnitude of $\sqrt{3}$, so the output of that sub-pass must be scaled to fit in the $[0, 1]$ storable in the surface. The output of the second sub-pass must be further scaled, and the values must be restored when read by the third sub-pass. The result of the Bisquare robust weighting function naturally lie in the range $[0, 1]$, so no further changes are necessary. This reduces the memory bandwidth requirements by approximately 50%.

[1]The OpenGL implementation stores the surface as `GL_RGBA16F`.

[2]The internal format requested is `GL_RGB10_A2_EXT`.

q values per pass	480p	720p	1080p	# passes
1	17ms		109ms	15
2	14.7ms		87ms	10
3	9.4ms	25ms	56ms	5
3 (10-bit surface)	7.8ms	21ms	46ms	5
Pixel increase	1x	2.66x	6x	

Table 1: *Comparison of optimization techniques. All use 5x5 region size and 5x5 search window on a single frame.*

4 Results

Table 1 compares the performance of the three methods implemented in this study. As the number of pixels in the frame increases, there is a linear increase in the processing time. This suggests that there are not adverse cache behaviors involved at these resolutions.

At 1080p, the 2-sample method improves performance 20%, but the 3-sample method improves performance by a further 36%. The super-linear improvement is due to the 2-sample method processing too many q values. The search window is 5×5, so for each q_y value, three q_x values should be processed. Since the 2-sample method processes values two at a time, it processes four q_x values.

By comparison, [Goossens et al. 2010] achieved 9ms per frame for 480p grayscale content[3]. While direct comparisons with their implementation was not possible, their presented work should be very similar to the 1-sample method presented here.

5 Conclusion

Real-time performance for 480p and 720p color video is easily achievable with the existing implementation on current generation integrated GPUs. Real-time performance for 1080p color video, however, is just out of reach. Additional implementation improvements may yield somewhat better performance, and driver improvements may also help. However, it is not expected that this will yield sufficient gains.

In addition to improving the performance of the $\{A=3, B=3, t_{past}=0, t_{future}=0\}$ NLMeans filter, the optimizations investigated in this project could be applied to larger values of A, larger values of B, and larger values of t_{past}. Each of which should improve the quality of the final result.

Both this study and [Goossens et al. 2010] ignore the effects of the video decoder. The data being denoised must come from somewhere, and this data requires some resources to process. In addition, this work uses RGB encoded data. Typical video processing pipelines provide data in planar YUV encodings, and the chromatic components are frequently subsampled. Utilizing both factors may allow some small performance improvements. The caching and memory bandwidth effects of the video decoder on the denoising algorithm may have additional penalties that should be studied.

References

BUADES, A., COLL, B., AND MOREL, J.-M. 2005. A non-local algorithm for image denoising. In *Proceedings of the*

[3]On a different GPU and different CPU. Identical hardware was not available during this study.

2005 IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR'05) - Volume 2 - Volume 02, IEEE Computer Society, Washington, DC, USA, CVPR '05, 60–65.

GOOSSENS, B., LUONG, Q., PIZURICA, A., AND PHILIPS, W. 2008. An improved non-local denoising algorithm. In *2008 International Workshop on Local and Non-Local Approximations in Image Processing*, Tampere International Center for Signal Processing, 143. invited paper.

GOOSSENS, B., LUONG, H., AELTERMAN, J., PIURICA, A., AND PHILIPS, W. 2010. A GPU-accelerated real-time NLMeans algorithm for denoising color video sequences. In *Advanced Concepts for Intelligent Vision Systems*, J. Blanc-Talon, D. Bone, W. Philips, D. Popescu, and P. Scheunders, Eds., vol. 6475 of *Lecture Notes in Computer Science*. Springer Berlin Heidelberg, 46–57.