Implementation of Non Local Means Filter in GPUs*

Adrián Márques and Alvaro Pardo

Universidad Catolica del Uruguay, Montevideo 11600, Uruguay
adrian.marques@gmail.com, apardo@ucu.edu.uy

Abstract. In this paper, we review some alternatives to reduce the computational complexity of the Non-Local Means image filter and present a CUDA-based implementation of it for GPUs, comparing its performance on different GPUs and with respect to reference CPU implementations. Starting from a naive CUDA implementation, we describe different aspects of CUDA and the algorithm itself that can be leveraged to decrease the execution time. Our GPU implementation achieved speedups of up to 35.8x with respect to our reduced-complexity reference implementation on the CPU, and more than 700x over a plain CPU implementation.

Keywords: Image denoising, Non-Local Means, GPU, CUDA.

1 Introduction

In this work we focus on the implementation in GPU of the Non-Local Means (NLM) image filter[3] which proposes to compute the output pixels as a weighted average of all pixels in the image (in practice for all pixels inside a given search region). The weights reflect the similarity between pixels and the novelty of the method is that this similarity is based on the distance between patches centered at pixels being processed. If $I(x)$ is the value of the input image at pixel x and S_x is a rectangular search region centered at pixel x the output of the NLM filter is computed with the following equation: $\hat{I}(x) = \frac{\sum_{y \in S_x} w(x,y) I(y)}{\sum_{y \in S_x} w(x,y)}$ where the weights $w(x, y)$ measure the similarity between patches N_x and N_y of size $(2W + 1) \times (2W + 1)$ centered at x and y respectively. This similarity is computed as: $w(x, y) = \exp(-||N_x - N_y||_2^2/h^2)$ with h a parameter that controls the aperture of the weighting function. We assume a search region with range $[-S, S]^2$. The computational cost of a naive implementation of NLM is $O(N^2(2S + 1)^2(2W + 1)^2))$ where N is the size of the image (N rows and columns), $(2S + 1)^2$ is the number of pixels in S_x and $(2W+1)^2$ is the number patch pixels. To alleviate the computational cost of the NLM filter several authors proposed different strategies to speed up the algorithm. These strategies can be classified into two categories. On the one hand the ones that propose approximations to the original NLM

* ANII FMV200913042 and SticAmsud MMVPSCV. Thanks to P. Ezzatti and E. Dufrechou from Univ. de la Republica for discussions and running our code on their machines.

J. Ruiz-Shulcloper and G. Sanniti di Baja (Eds.): CIARP 2013, Part I, LNCS 8258, pp. 407–414, 2013.

that allow the reduction of the computation cost [8,7]. On the other hand, there are solutions that reduce the computational cost while implementing the same filter [5,4]. Here, we review these references that inspired our work for the GPU implementation.

In [4] Condat proposes an elegant solution to lower the computational cost using convolutions. The first observation is the following. If the pixel y is expressed using a displacement vector starting from pixel x as $y = x + dx$ then the weights fulfill $w(x, x + dx) = w(y, y - dx)$. Therefore, there is no need to compute both weights. The second modification involves swapping the loops in x and dx and dividing the computation of the weights in two steps. First, compute an image with square differences: $u(x; dx) = (I(x) - I(x + dx))^2$ Second, using the image $u(x; dx)$, the weights are expressed using convolutions as:

$$w(x, x + dx) = \exp(-v(x)/h^2), \;\; v(dx) = \sum_{x \in N} u(x; dx) = u(x; dx) * g$$

where g is a square kernel of size $(2W + 1)^2$. Condat's algorithm is:

$\hat{I}(.), \; C(.) \; = \; 0$
`for all` dx `in halved search region`
  `compute the image` $u(x) \; = \; (\; I(x) \; - \; I(x + dx) \;)^2$
  `compute the exponents` $v(x) \; = \; u(x) * g$
  `for all pixels` x
    $w(x + dx) \; = \; exp(\; -v(x)/h^2 \;)$
    $\hat{I}(x) \; += \; w(x + dx)I(x + dx)$; $\hat{I}(x + dx) \; += \; w(x + dx)I(x)$
    $C(x) \; += \; w(x + dx)$; $C(x + dx) \; += \; w(x + dx)$
`for all pixels` x
  $\hat{I}(x) \; = \; \hat{I}(x)/C(x)$

The computational cost of this algorithm is $O(N^2(2S + 1)^2(2W + 1))$ which implies a reduction of $(2W + 1)$. If the convolution with g is implemented with a IIR filter this cost can be further reduced to $O(N^2(2S + 1)^2)$. A similar solution was presented in [5] by Darbon et. al., where they also express the differences between patches as a convolution and calculate them using integral images. This alternative has a computational cost $O(4N^2S^2)$ which is independent of the patch size (does not depend on W).

In this work we evaluate a GPU implementation of Condat's algorithm and study different optimizations at the GPU level. For comparison purposes we also implemented CPU versions of Condat's and Darbon's proposals.

2 GPUs

Modern GPUs are very efficient in parallel processing of computer graphics data but also with any other type of data that can take advantage of the parallel nature of the GPUs. The manufacturers of the GPUs realized the power of this technology in fields beyond computer graphics and introduced programming

models that transform the GPU units into more general computing devices. Image and video processing are two examples where the application of GPUs gives many benefits and great reductions in computational time. Since GPUs are basically consumer electronics products they are very competitive in terms of price. The programming models provided by the manufacturers are transparent to the hardware specifications to allow the end user to upgrade the hardware to increase computational power without the need to modify the software. NVIDIA was the first company to introduce a general-purpose programming model with the release of CUDA and recently other companies joined efforts around the OpenCL standard.

Several authors have proposed NLM implementations for GPUs. In [2] the authors divide the image in blocks and calculate weights only for the central pixel, assigning that weight to all pixels within the block. Although their proposed method does reduce the computational complexity of the algorithm, it does so by sacrificing denoising performance, since this coarse weight approximation can introduce artifacts at the edges in the image. This same implementation is evaluated in [9].

In [10], a CUDA implementation of NLM for CT scans is presented that takes no steps to reduce the computational complexity of the algorithm. The authors' main contribution towards runtime optimization is exploiting the shared memory space to prefetch and then access the image data rather than reading from global memory multiple times, since the former can be accessed much faster than the latter. However, shared memory is a limited resource that restricts the number of thread blocks that can be run concurrently on a streaming multiprocessor, and the proposed approach does not extend well when processing color images or video. In [6], the authors present a DirectX implementation that, just as Condat's and Darbon's, exploits the fact that the differences between patches can be calculated as a convolution.

3 Proposed Implementation

There are many resources available to learn CUDA programming, and coding an initial version of a parallel application can be very easy. However, to get the most of the GPU, a deeper understanding of the underlying architecture is usually required and at this point the learning curve grows steeper. In this article we describe each of these improvements so that they may serve as an introductory guide to others that may be getting started with implementing image processing applications in CUDA.

Naive Implementation: This consists in a straightforward implementation of Condat's algorithm. Host code controls the iteration through the search region while GPU kernel functions are invoked for displaced image subtraction, separable convolution, addition of weighted pixel contribution and finally division of the contributions by the total summed weights. The pseudo code for this approach is described below. For the separable convolution, we used the CUDA Toolkit

sample code described in [1]. The other kernels are straightforward implementations of their CPU counterparts. The only addition is that, since a thread with linearized index x updates $\hat{I}(x)$ and $C(x)$ as well as $\hat{I}(x+dx)$ and $C(x+dx)$, we introduced another pair of accumulation and summed weight images $\hat{I}_{sym}$ and C_{sym} to store the symmetric contributions and thus eliminate concurrency overwrite issues between threads.

$\hat{I}(x),\ \hat{I}\_sym\ =\ 0,\ C,\ C\_sym\ =\ 0$
`for all` dx `in halved search region`
  $u\ =\ displaced\_image\_substraction\_kernel(I,\ dx)$
  $v\ =\ separable\_convolution\_kernel(u)$
  $(\hat{I},\ \hat{I}\_sym,\ C,\ C\_sym)\ +=\ add\_weighted\_pixel\_contributions\_kernel(I,\ v,\ dx)$
$\hat{I}(x)\ =\ weight\_normalization\_kernel(\hat{I}(x),\ \hat{I}\_sym,\ C(x),\ C\_sym)$

Coalescing Memory Access: On many GPU applications, memory access can have a great impact on performance. Reads and writes to global memory can be coalesced (meaning grouped into a single transaction) when the threads in a warp access the memory addresses in predefined patterns. These patterns can vary depending on the CUDA architecture, with 1.0 and 1.1 being the most restrictive and relaxing into more permissive models from 1.2 to 2.x and 3.x versions. In CUDA 1.0 and 1.1, successive threads in a half-warp must access consecutive 4, 8, or 16-byte words, with the first word located in a memory address aligned to the size of the transaction. In order to coalesce most global memory reads, we allocated the memory for our images using the function $cudaMallocPitch()$ and $cudaMemcpy2D()$ rather than $cudaMalloc()$ and $cudaMemcpy()$. The former pads (if necessary) the allocation to ensure that the addresses of the rows of 2D arrays will meet the alignment requirements for coalescing. Since we replicate the border of the processed images, we also had to make sure that the size of the replicated border was a multiple of 16 for our card with CUDA 1.1 and of 32 for our cards with CUDA 2.0 or higher, in order to assure memory alignment when working within the border. After coalescing global memory access in this manner, a speedup of 1.5x over the naive GPU implementation was obtained.

Using 2D Textures for Remaining Unaligned Reads: After the modifications described above, all reads and writes of threads with linearized index x to pixels with the same index will be coalesced. However, the kernels that compute image subtraction and addition of weighted pixel contributions also perform accesses that remain uncoalesced to pixels indexed as $x+dx$. We therefore explored using textures to accelerate these read operations. The texture memory space is read-only and resides in device memory but is cached, so a texture fetch will cost one memory read from the texture cache rather than global memory unless a cache miss occurs, in which case the cost will then be a read from global memory. Since this cache is optimized for 2D spatial locality, higher bandwidth can be achieved by using textures if memory reads by threads in the same warp do not follow the access patterns required for memory transaction coalescing but the read addresses are close together in 2D. By using textures to read displaced pixel values, the speedup factor over the previous implementation was of 1.2x.

As an alternative to texture fetches for unaligned memory access, we experimented with prefetching the data to shared memory using a coalesced memory access pattern to then operate on the data in shared memory. However, due to the overhead introduced by the prefetching code and that we only used the prefetched data once per kernel, using textures remained the faster option.

Coalescing Remaining Write Operations: At this point, writes of the contributions and weights of displaced pixels to $\hat{I}_{sym}(x+dx)$ and $C_{sym}(x+dx)$ still remained uncoalesced. However, if instead of each thread using $w(x)$ to update indexes x and $x+dx$ we change to updating only x using $w(x)$ and $w(x-dx)$ as noted in [6], all writes can be coalesced. Furthermore, the need for a second set of images to keep track of symmetric weighted contributions and weights dissapears as well, since each thread will now update a single image index. Under this strategy, in order to avoid reevaluations of the exponential function, the convolution kernel has to be trivially modified to calculate the weights of each pixel as a last step. This modification resulted in a further speedup factor of 2.3x.

4 Results

All of the reported NLM implementations operate on color float images. Table 1 details the execution times and speedups obtained for each of the GPU implementation variants mentioned in section 3, with 4.1x being the final speedup factor obtained over the naive GPU implementation. These results were obtained on a Quadro FX 770M card with compute capability 1.1. Following CUDA versions introduced global memory caching that may provide a higher bandwidth than texture fetches if the accessed elements are present on the cache, which may yield different speedup factors than these.

Table 2 lists execution times for the different algorithm variants we implemented on the CPU and our current GPU version. Since the purpose of the CPU implementations was to provide easily reproducible and comparable baseline execution times, straightforward implementations with no particular code optimizations were employed. The fastest implementation on the CPU was Condat's alternative, which represented a 11.8x improvement over the implementation that only exploits weight symmetry. In turn, the GPU version was 32.4x faster than its CPU counterpart (Condat) and an impressive 717.9x faster than the naive CPU implementation, but the latter is hardly a fair comparison.

We experimented with running the algorithm on different cards, obtaining the same execution times for a Quadro FX 770M and a GT 430, in spite of the latter having 3 times as many cores as the former. This is caused by the algorithm being bandwidth-bound rather than compute-bound and both cards having the same memory bandwidth of 25.6 GB/sec. For the GTX 480, with 133.9 GB/sec, and the GTX 680, with 192.2 GB/sec, execution times were 8.7x and 10.7x respectively faster than with the previous cards. Thread block dimensions were set to maximize occupancy. In order to comply with memory access coalescing

Table 1. Execution times and speedup factors over the naive GPU implementation for all tested GPU implementation alternatives. Listed results correspond to 512x512 images with a 21x21 search window and 9x9 patches.

GPU implementation alternative	Execution Time (sec)	Speedup vs. Naive Implementation
Naive Implementation	3.09	1x
Coalescing memory access	2.06	1.5x
Using 2D textures for unaligned reads	1.72	1.8x
Coalescing remaining write operations	0.73	4.1x

Table 2. Execution times and speedup factors for a 720x480 image with a 21x21 search window and 9x9 patches. The CPU used was an Intel Core Intel Core i7 2600 CPU @ 3.40GHz. The GPU implementation was run on a NVIDIA GeForce GTX 680 card.

NLM implementation alternative	Execution Time (ms)	Speedup vs. Naive Implementation
CPU 1: Naive Implementation	63,180	1x
CPU 2: Using symmetric weights	33,688	1.9x
CPU 3: Darbon (integral images)	4,789	13.2x
CPU 4: Condat (separable convolution)	2,851	22.2x
GPU Final implementation	88	717.9x

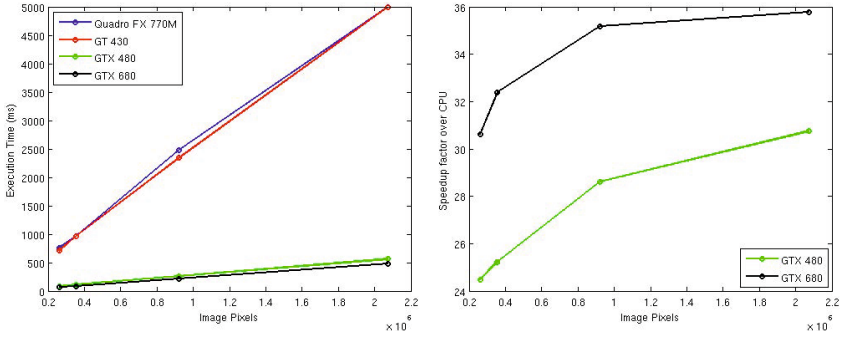

Fig. 1. Left: Execution times per image pixels for different GPUs on images of size 512x512, 720x480, 1200x720 and 1920x1080. **Right:** GPU speedup factor over an Intel Core i7 2600 @ 3.40GHz CPU for the different test image sizes.

patterns, the block width was set to 16 for the Quadro FX 770M, which has compute capability 1.1, and to 32 for the other cards.

Figure 1 (left) illustrates execution times for NLM running on images of different sizes over the different GPU cards. For these image sizes and the best two cards, the speedup factor obtained over the CPU implementation of Condat's method is plotted on the right. The speedup factor increases with image size, leveling off for larger images, and varies between 24.5x and 30.7x for the GTX 480 and between 30.6x and 35.8x for the GTX 680.

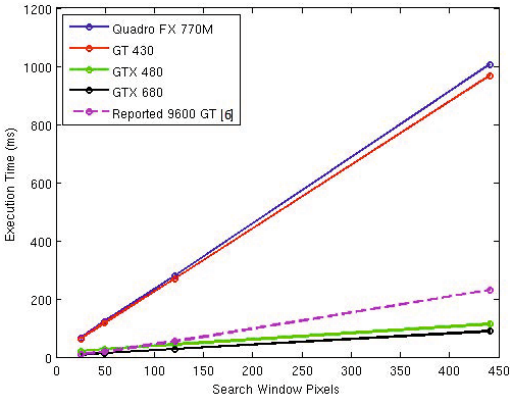

Fig. 2. Execution times per pixels in search window for windows of size 5x5, 7x7, 11x11 and 21x21. Image size remained fixed at 720x480 and patch size at 9x9.

Figure 2 illustrates the execution times when varying the search window's size. Since the algorithm calculates the differences between patches using a separable convolution, changing the patch size hardly affects execution time. The execution times reported in [6] for their DirectX-based implementation, which also exploits the use of a convolution to calculate the distances between image patches, are included among the results presented in figure 2. A more direct comparison was not possible, but when adjusting for the maximum memory bandwidth of the cards, the results in 2 seem to be up to 2x faster than our current implementation. The difference may lie in that they mention computing the convolution as a moving average, whereas we compute the separable convolution. We will evaluate whether we can improve on this point.

5 Conclusions

In this paper we have presented a CUDA-based GPU implementation of NLM that reduces its computational complexity by calculating the differences between images patches as a separable convolution. This variant still produces the same result as the original algorithm, as opposed to the CUDA implementation proposed in [2], which calculates weights for only a subset of image pixels and assigns the same weight within image blocks. It is also faster than the alternative described in [10], since in that case the authors do not reduce the algorithm's computational complexity. The implementation that can be more closely compared to our work is the DirectX-based one presented in [6], which seems to suggest that we could still improve upon our convolution computation to achieve higher speedups.

With respect to our CPU reference implementation, a speedup factor of 3.5x was obtained with the Quadro FX 770M and GeForce GT 430 cards, and up to 30x and 35x speedups were obtained with the comparatively more powerful GTX

480 and GTX 680. The main reason behind the difference in performance between cards is the memory bandwidth of each, since the algorithm is bandwidth-limited. The final speedup factor with respect to a naive CPU implementation was of 718x.

One of the contributions we have tried to make with this paper has been to report each step we have taken while optimizing our implementation, starting from the most basic, so that it may serve as a quick reference for people that are just starting to port their image processing algorithms to CUDA.

It should be noted that we have not explored yet all concepts that we believe may lead to further efficiency improvements. In particular, our access to the better-performing cards reported in this work has been recent, and further exploring implementation alternatives on them can probably yield additional optimizations. As future work, we plan to explore these remaining promising modifications and write a revised version of this article more focused on serving as an quick introduction to CUDA optimization for image processing tasks based on the NLM case study.

References

1. Podlozhnyuk, V., Kharlamov, A.: Image convolution with CUDA. Technical report. NVIDIA, Inc., Santa Clara (2007)
2. Podlozhnyuk, V., Kharlamov, A.: Image denoising. Technical report. NVIDIA, Inc., Santa Clara (2007)
3. Buades, A., Coll, B., Morel, J.M.: A non-local algorithm for image denoising. In: CVPR, pp. 60–65 (2005)
4. Condat, L.: A simple trick to speed up the non-local means. Technical report
5. Darbon, J., Cunha, A., Chan, T., Osher, S., Jensen, G.: Fast nonlocal filtering applied to electron cryomicroscopy. In: ISBI, pp. 1331–1334 (2008)
6. Goossens, B., Luong, H., Aelterman, J., Pižurica, A., Philips, W.: A GPU-accelerated real-time NLMeans algorithm for denoising color video sequences. In: Blanc-Talon, J., Bone, D., Philips, W., Popescu, D., Scheunders, P. (eds.) ACIVS 2010, Part II. LNCS, vol. 6475, pp. 46–57. Springer, Heidelberg (2010)
7. Orchard, J., Ebrahimi, M., Wong, A.: Efficient nonlocal-means denoising using the SVD. In: ICIP, pp. 1732–1735 (2008)
8. Tasdizen, T.: Principal neighborhood dictionaries for nonlocal means image denoising. IEEE Trans. on Image Process. 18(12), 2649–2660 (2009)
9. Wu, H., Zhang, W.-H., Gao, D.-Z., Yin, X.-D., Chen, Y., Wang, W.-D.: Fast CT image processing using parallelized non-local means. Journal of Medical and Biological Eng. 31(6), 437–441 (2011)
10. Mueller, K., Zheng, Z., Xu, W.: Performance tuning for CUDA-accelerated neighborhood denoising filters. In: Workshop on High Performance Image Reconstruction (July 2011)