Performance Tuning for CUDA-Accelerated Neighborhood Denoising Filters

Ziyi Zheng, Wei Xu and Klaus Mueller

Abstract—**Neighborhood denoising filters are powerful techniques in image processing and can effectively enhance the image quality in CT reconstructions. In this study, by taking the bilateral filter and the non-local mean filter as two examples, we discuss their implementations and perform fine-tuning on the targeted GPU architecture. Experimental results show that the straightforward GPU-based neighborhood filters can be further accelerated by pre-fetching. The optimized GPU-accelerated denoising filters are ready for plug-in into reconstruction framework to enable fast denoising without compromising image quality.**

Index Terms—**GPU, Denoising, Bilateral Filter, non-local mean filter, CUDA, Computed Tomography**

I. Introduction

There has been growing concern about the high radiation dose delivered to patients in cone-beam X-ray CT, and thus recently low dose CT has gained substantial interests in research. It usually involves lowering X-ray energy or reducing the number of projections, or both. In traditional scenarios, these approaches suffer from low signal-noise-ratio (SNR). To lower the radiation doses without compromising image quality, recent research proposes to use iterative reconstruction methods together with neighborhood denoising filters [1] as regularization steps interleaved with reconstruction steps.

Neighborhood filters are ubiquitous in image de-noising. The bilateral filter (BF) [2] and the non-local means filter (NLM) [3] are two of the most popular neighborhood filters and there has been widespread coverage of research on using these two filters [1] [4-6]. The advantages of these denoising filters are that they can help to reduce substantial noise and in the meantime preserve edges. However, to achieve the high-quality denoising effects, these filters require extensive neighborhood search which results in long running times. There are two major approaches to speed up the denoising procedure. One approach focuses on how to approximate exact filtering computations [2][3]. Another approach turns to parallel computing devices for solutions, most dominantly high-performance graphics processing units (GPUs). Our earlier works [1] shows that a straightforward GPU implementation offered better speed than filtering via TVM (Total Variation Minimization). In that study we focused on both the quality and speed performance of regularized iterative CT reconstruction. In the current paper, we focus on advanced accelerating techniques for the various neighborhood filters. We show that pre-computation along with a pre-fetching scheme is quite effective for denoising filters, especially for large neighborhood sizes.

In this paper, Section 2 presents related work and background. Section 3 introduces the overall methodology and Section 4 shows results, followed by conclusion in Section 5.

Ziyi Zheng and Klaus Mueller are with the Center for Visual Computing, Computer Science, Stony Brook University, Stony Brook, NY 11790 USA (phone: 631-632-1524; e-mail: {zizhen, mueller}@cs.sunysb.edu).
Funding was provided by NSF grant EAGER 1050477.

II. Background

In this paper, we take the NVIDIA GeForce GTX 480 GPU as an example to discuss the GPU architecture. A GTX 480 GPU card contains 480 processors. These 480 processors are grouped into 15 *streaming multi-processor* (SMP) which can perform tasks independently from each other. Each SMP contains 32 processors, which allow 32 threads (a warp) to execute concurrently. Thus each SMP is inherently based on single instruction multiple data (SIMD) design. In the best case, the GTX 480 has theoretical computational power reaching 1.3 Tera-floating point operations per second (TFLOPS) in single floating-point precision which largely outperform the CPU computational power.

GPU device memory is an off-chip memory that stores the input data and receives the output from the processors. The GTX 480 has 1.5GB DDR5 device memory with peak bandwidth 177.4 GB/s. Although the bandwidth of GPU memory is much faster than that of the CPU memory, it has several limitations. First, each off-chip memory (also called device/global memory) access instruction takes several hundreds of clock cycle. This latency needs to be alleviated by issuing a large amount of threads which will automatically enable hardware context switching. Second, the memory instructions should better to be coalesced or at least have a specified granularity (128 bytes). The maximum GPU global bandwidth can only be achieved by issuing 1 memory instruction for 128 bytes data. This implies 32 neighbouring threads (a warp) should read/write within a 128-byte-aligned segment. With proper alignment, sequential mapping of threads to memory address will yield a coalesced memory access pattern.

To further reduce the huge costs associated with off-chip memory access, the cache can be leveraged. Constant memory cache is the simplest type of cache. It is an off-chip memory with the similar bandwidth as device memory. To speed up the constant data access rate, a GTX 480 contains an 8KB cache per 8 processors for constant memory access. Besides the constant cache, 32 processors within one SMP share an L1 cache and a user-controllable cache known as the shared memory. The difference between the L1 cache and the shared memory is that the former is automatically scheduled by the hardware and the latter can be controlled by the user to perform prefetching. The amount of shared memory and L1 cache in one SMP is user-configurable (16KB + 48KB or 48KB + 16 KB).

NVIDIA GPUs can be programmed via a C-like API – CUDA. CUDA is a general purpose API which exposes more control over how a task is computed on the GPU hardware, as compared to graphics-based APIs (CG, GLSL). The task-hardware mapping is enabled by introducing the concept of "*block*". Each *block* is mapped to an SMP.

The key difference between CPU implementation and its GPU counterpart is the parallel programing. While typically CPU program will launch one thread, GPU will launch millions of threads with the same instruction. A large amount of threads are executed in terms of thread *blocks*, whereas the total task is called *grid*. On the hardware level, each *block* is mapped to a single SMP. In the back-projection stage of the CT reconstruction, SMPs are assigned to different regions of the resulting volume sequentially. This enables a mapping where the *grid-block* decomposition in CUDA corresponds to the volumetric reconstructed 3D dataset. To avoid misunderstanding, we use *block* and *grid* in this paper only as terms in CUDA, not for their geometry meaning.

III. METHODOLOGY

A. *Straightforward implementation*

Neighborhood filter CUDA kernels are similar to their CG implementations — fragment programs. If we assume one CUDA kernel function only computes one resulting pixel, a neighborhood filter fragment program can be changed into its CUDA kernel without much modification. In the SIMD architecture, the same kernel/fragment program will replicate itself to all different processors. These threads on different processors have unique two-dimensional IDs *(x, y)* to guide them to read neighborhood data around *(x, y)* and output to the value at *(x, y)*.

Here we list the pseudo-code for a 2D neighborhood filter kernel:

```
Neighborhood_filter_2D
    Obtain the current thread ID (x, y)
    Collect all pixels' values in 2D neighborhood within the mask
    Calculate output pixels value defined by filtering algorithm
    Output results (x,y)at the resulting image
End
```

Figure 1. Pseudo-code for 2D neighborhood filter kernel.

CUDA has more sophisticated controls which are not available in CG. CUDA's execution configuration guides how the parallel computations are assigned on GPU hardware on streaming-multi-processor (SMP) level. This can be done by dividing the 2D image into tiles and assign them to a CUDA *block*. Each of the 2D tiles will be mapped into a SMP.

To achieve maximum bandwidth in reading, the output image is stored in 2D pitched memory and the input is stored in a read-only 2D texture. In addition, to confirm the rule that each warp (32 threads) writes to a 128-byte segment, each thread should output a 4-byte unit. This 4-byte unit can be 4 characters, 2 short integers or 1 single-precision floating-point number.

B. *Pre-computation*

Some of neighborhood filters such as the bilateral filter or the non-local means (NLM) filter involve 2 Gaussian weights: σ_x, σ_y. They define the smoothing parameters in the x, y axis respectively.

Pre-computing techniques can be applied on the filter to reduce computational cost. Given the mask size, we can pre-compute a discrete mask for the 2D Gaussian smooth kernel and store it in the GPU's constant memory. Then once cached in SMP, these pre-computed weights will be ready to use which will save a huge amount of exponential computations. However, the Gaussian in the intensity domain which is inherently different from spatial dimension since it is sampled in a continuous domain. Although similar pre-computing method exists, which discretize the continuous intensity domain and lookup the pre-computed weightings, we have not explored the speed-quality trade-off of this approximation technique. We calculate the intensity Gaussian on the fly, therefore let our GPU algorithm is an exact method.

We store the output volume in 2D pitched memory in order to achieve better global memory bandwidth. The output is decoupled from the order of the loops in the CUDA kernel computation. Switching the order of the loops or changing the output storage to YX will result in non-coalesced memory writing patterns that downgrade the performance. Furthermore, this loop order also indicates the pre-computed weights should be organized in XY order.

C. *Prefetching*

We also use the prefetching method to reduce the data-transfer cost, based on huge difference between on-chip and off-chip memory bandwidth. Prefetching is done according to the *apron* which is the image region served as input of a *block* of threads. The size of the 2D preloading apron is:

$$(w_b+2r_w+2r_p)\cdot(h_b+2r_w+2r_p) \tag{4}$$

where w_b,and h_b are width and height of a 2D CUDA block. r_p is the patch radius and r_w is the windows radius in non-local mean filler [3], while for bilateral filter [2] $r_p = 0$.

The apron is usually larger than the output region and thus aprons from different CUDA *blocks* are overlapped. Since neighborhood filters re-use input data in an apron multiple times, shared memory can serve as a user controllable cache to reduce the off-chip memory access. Then a 2D prefetching approach can yield less cache misses which will result in better performance.

The code for loading an apron is listed in Figure 2. The LOCAL_BLOCK_W and LOCAL_BLOCK_H are the two dimensions of the apron defined in Equation (4). The input data is stored in a texture reference *2d_tex*. The data fetching is performed by tex2D(*2d_tex*, *idx* + 0.5f, *idy* + 0.5f) where the current position for the thread is (*idx*, *idy*) and is offset by (0.5f, 0.5f). Essentially, the code shows that each thread will perform at most 4 loading operations, based on the assumption that the neighborhood area cannot be 4 times larger than the CUDA block area. We will illustrate the positions of these 4 preloading data later in this section. Note since all the threads executed in parallel without orderings, we need to ensure the local 2D apron was fully loaded into the shared memory in current SMP before the input readings are directed to the shared memory, The block-level synchronization should occur immediately after the prefetching.

```
__shared__ float LocalBlock [LOCAL_BLOCK_W * LOCAL_BLOCK_H];

        int SharedIndex = threadIdx.y * LOCAL_BLOCK_W;

        LocalBlock [SharedIndex + threadIdx.x]
         = tex2D ( 2d_tex, idx - KERNEL_RADIUS_X + 0.5f,
                       idy - KERNEL_RADIUS_Y + 0.5f  );

        if( threadIdx.x + blockDim.x <  LOCAL_BLOCK_W )
           LocalBlock [ SharedIndex + threadIdx.x + blockDim.x ]
            = tex2D ( 2d_tex, idx + (int) blockDim.x - KERNEL_RADIUS_X + 0.5f,
                          idy - KERNEL_RADIUS_Y + 0.5f );

        if ( threadIdx.y < KERNEL_RADIUS_Y * 2  )
        {
           SharedIndex = (threadIdx.y + blockDim.y) * LOCAL_BLOCK_W;

           LocalBlock [ SharedIndex + threadIdx.x ]
            = tex2D( 2d_tex, idx - KERNEL_RADIUS_X + 0.5f,
                          idy + (int) blockDim.y - KERNEL_RADIUS_Y + 0.5f );

           if( threadIdx.x+ blockDim.x <  LOCAL_BLOCK_W)
              LocalBlock [ SharedIndex + threadIdx.x + blockDim.x ]
              =tex2D( 2d_tex, idx + (int) blockDim.x - KERNEL_RADIUS_X + 0.5f,
                             idy + (int) blockDim.y - KERNEL_RADIUS_Y+ 0.5f );
        }

   __syncthreads();
```

Figure 2. CUDA kernel code for 2D neighborhood prefetching. The type and built-in variables/functions in CUDA is in blue. The input texture reference is shown in orange and the shared memory is shown in gray.

Figure 3 illustrates the prefetching scheme for the 2D neighborhood filters. In this case a 16×16 CUDA block (in green) needs to read 32×32 pixels in its neighborhood (all pixels in panel (a)). Performing neighborhood filtering directly on (a) will result in low performance. Panel (b-e) shows the proposed prefetching method will load 4 16×16 tiles into shared memory (in gray) in sequence. Finally in panel (f), the CUDA threads in the green CUDA block can fast access the input in on-chip cache (shared memory). Then applying neighborhood filters on (f) will guarantee there will be no cache miss afterward thus will boost the performance.

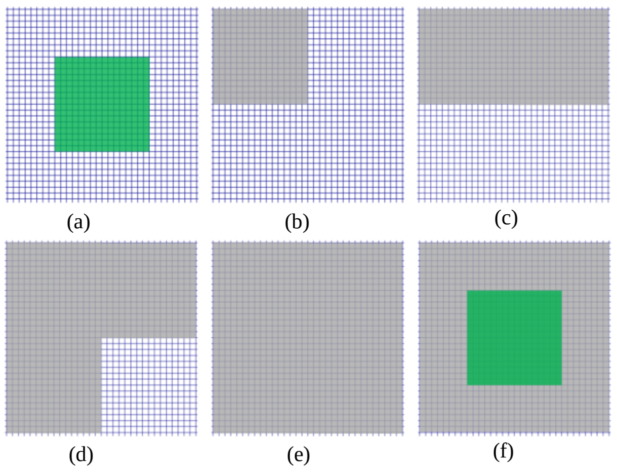


Figure 3. An example to illustrate prefetching procedure. (a) shows the configuration with one CUDA block (in green). (b-e) show the neighborhood pixels are loaded into shared memory (in gray). (f) shows the data configuration after loading all the input into the user shared memory.

IV. Results

Our experiments were conducted on an NVIDIA GTX 480 GPU, programmed with CUDA 3.2 runtime API and with an Intel Core 2 Duo CPU @ 2.66GHz. We built the program in 32bit mode. In the experiment, the size of the CUDA *block* is set to 32×32. The first dimension is chosen to be 32 to conform to the coalescing rule. The second dimension we choose the maximum number as 32 due to the block's size limit 1024 in the NVIDIA Fermi card.

We did a performance and image quality study on one slice of a human head. We simulated 90 parallel beam projections and added Gaussian noise SNR=25 (SNR is computed by the ratio of the mean pixel value to the standard deviation of Gaussian noise) into the projections. Figure 4(a) shows the gold-standard and Figure 4(b) shows the iterative reconstruction results from noisy projections.

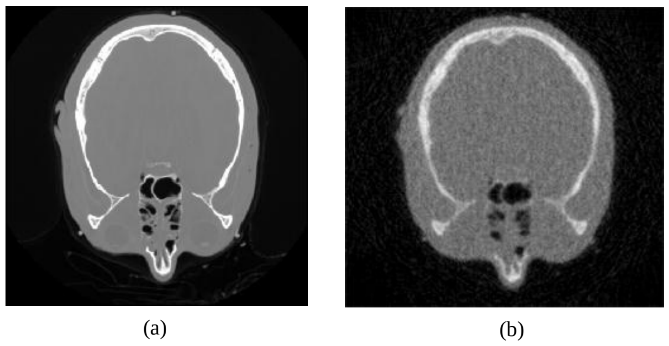
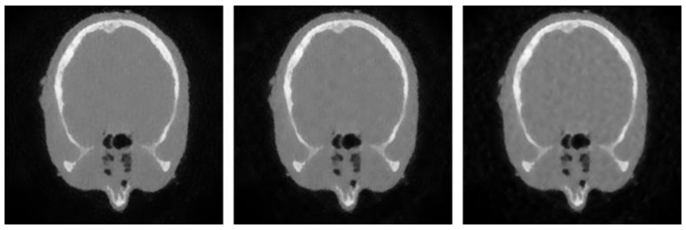


Figure 4. Testing image with size 256^2. (a) shows the gold-standard. (b) shows iterative reconstruction from 90 noisy projections.

Figure 5 shows images restored bilateral filtering. The image quality of the bilateral filtering depends on smoothing parameters and window sizes. Here σ_x and σ_y control the spatial Gaussian, σ_r controls the range Gaussian and r_w is the window radius. The window size is $2r_w$+1. Here we show approximately the best parameters for each window size. The large window size case (17×17 in Figure 5(a)) generated better results than 11×11 (Figure 5(b)) and 7×7 (Figure 5(c)).



(a) $\sigma_x = \sigma_y = 30$ $\sigma_r = 19$ $r_w = 8$

(b) $\sigma_x = \sigma_y = 38$ $\sigma_r = 19$ $r_w = 5$

(c) $\sigma_x = \sigma_y = 40$ $\sigma_r = 20$ $r_w = 3$

Figure 5. Bilateral filtering result.

We extend the performance test of bilateral filter on larger image sizes. The computation time is listed in Table I (in ms).

Besides the computation timing, we note that the memory transfer time from CPU and GPU is 0.7 ms for 256^2 data, 2.0 ms for 512^2 data, 7.5 ms for 1024^2 data. By using the prefetching scheme, a speedup ratio of 20% is achieved for the bilateral filter.

Next, the NLM filter is applied to the test dataset. Figure 6 demonstrates the image quality of the NLM filter with different window sizes. In the NLM filter, there is a parameter *h* that controls the noise reduction effect. We also find the approximately best parameters for different windows sizes. The large neighborhood size ($(11+17)^2$ in Figure 6(a)) resulted better quality than in the smaller neighborhood case ($(11+11)^2$ in Figure 6(b) and $(11+7)^2$ in Figure 6(c)).

TABLE I
PERFORMANCE IN BILATERAL FILTER (IN MILLISECONDS)

Image size	Neighborhood Size	Bilateral	Optimized Bilateral	Speedup
256^2	7^2	0.192	0.131	1.46
	11^2	0.309	0.246	1.25
	17^2	0.650	0.539	1.21
512^2	7^2	0.411	0.326	1.26
	11^2	0.927	0.705	1.31
	17^2	2.150	1.760	1.22
1024^2	7^2	1.446	1.120	1.29
	11^2	3.374	2.473	1.36
	17^2	8.080	6.545	1.23

The NLM filter's performance is shown in Table II. The prefetching method resulted up to 4× speedup in this filter. This is because the NLM filter has one order of magnitude more neighborhood searching to do than the bilateral filter, which make them clearly a memory bounded problem. Our optimized filters outperformed the bilateral filter and the NLM filter implemented in the CUDA SDK [7] by similar speedups. The performance shows the more neighborhood lookups, the more effective the shared-memory acceleration will be. Based on the fact that the NLM filter is usually one order of magnitude slower than bilateral filter but has better denoising quality, our proposed method would make expensive neighborhood filters more practical while enjoying the superior image quality.

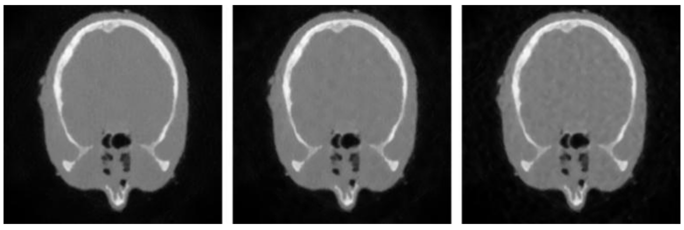

(a) $h = 17$ $r_p = 5$ $r_w = 8$ (a) $h = 17$ $r_p = 5$ $r_w = 5$ (c) $h = 18$ $r_p = 5$ $r_w = 3$

Figure 6. NLM filtering results.

By using single precision floating point data, the largest amount of required shared memory is $(32+2\times(8))^2\times4 = 9216$ byte for the bilateral and the NLM filter. They are below 48KB as the limit of shared memory per SMP in NVIDIA's Fermi card. With the development of more advanced GPU hardware, we can expect that larger preloads such as 64×64 in 32bit floating point data will be supported in the future.

TABLE II
PERFORMANCE IN NON-LOCAL MEANS FILTER (IN MILLISECONDS)

Image size	Neighborhood Size	NLM	Optimized NLM	Speedup
256^2	$(7+7)^2$	8.708	2.097	4.15
	$(7+11)^2$	23.927	5.034	4.75
	$(7+17)^2$	51.095	12.703	4.02
512^2	$(7+7)^2$	31.026	7.339	4.23
	$(7+11)^2$	76.494	17.756	4.31
	$(7+17)^2$	182.497	42.066	4.34
1024^2	$(7+7)^2$	118.831	28.041	4.24
	$(7+11)^2$	292.970	67.727	4.33
	$(7+17)^2$	699.231	161	4.34

V. CONCLUSION

In this paper, we showed that advanced acceleration techniques (such as pre-computation, prefetching) can further speedup straightforward GPU implementations of nearest neighborhood filters. The speedup can be up to 4 times for the large window size case, which can bring high-quality denoising filters real-time performance. Our optimized filter lends itself as an independent component which can be plugged into any iterative reconstruction frameworks and boost the performance of the entire pipeline. While we have only shown 2D filtering results here (it appeared to yield better results for our reconstructions) similar techniques will also apply to 3D filtering.

Finally, although the quality that can be achieved with BLF and NLM appears rather similar in the results presented here, this is mainly due to the relatively low level of noise we experimented with (the main objective of this paper was the GPU acceleration scheme). Companion work [8] presented elsewhere shows that the NLM scheme does significantly better with higher noise and streak artifact levels. Thus, when artifact levels are low, the BLF is preferred due to its higher speed, but with greater artifacts the NLM is a better choice.

REFERENCES

[1] W. Xu, K. Mueller, "A performance-driven study of regularization methods for GPU-accelerated iterative CT," In *Workshop on High Performance Image Reconstruction*, 2009.

[2] C. Tomasi, R. Manduchi, "Bilateral filtering for gray and color images," IEEE International Conference on Computer Vision, pp. 839-846, 1998.

[3] A. Buades, B. Coll, , J.M. Morel, "A non-local algorithm for image denoising," Computer Vision and Pattern Recognition, pp. 60-65, 2005.

[4] T. Pham and L. van Vliet. "Separable bilateral filtering for fast video preprocessing," *IEEE International Conference on Multimedia and Expo*, pp. 4, 2005.

[5] S. Paris, F. Durand. "A fast approximation of the bilateral filter using a signal processing approach," *International Journal of Computer Vision*, vol. 81, no. 1, pp. 24-52, 2009

[6] M. Howison. "Comparing GPU implementations of bilateral and anisotropic diffusion filters for 3D biomedical datasets," *SIAM Conference on Imaging Science*, 2010.

[7] A. Kharlamov, V. Podlozhnyuk, "Image Denoising," http://developer.download.nvidia.com/compute/cuda/1_1/Website/projects/imageDenoising/doc/imageDenoising.pdf, 2007.

[8] W. Xu, K. Mueller, "Evaluating popular non-linear image processing filters for their use in regularized iterative CT," *Conference Record IEEE Medical Imaging Conference (MIC)*, Knoxville, TN, 2010.