Fine-tuned high-speed implementation of a GPU-based median filter.

Gilles Perrot Stéphane Domas
Raphaël Couturier
FEMTO-ST institute
Rue Engel Gros, 90000 Belfort, France.

April 9, 2014

Abstract

Median filtering is a well-known method used in a wide range of application frameworks as well as a standalone filter, especially for *salt-and-pepper* denoising. It is able to highly reduce the power of noise while minimizing edge blurring. Currently, existing algorithms and implementations are quite efficient but may be improved as far as processing speed is concerned, which has led us to further investigate the specificities of modern GPUs. In this paper, we propose the GPU implementation of fixed-size kernel median filters, able to output up to 1.85 billion pixels per second on C2070 Tesla cards. Based on a Branchless Vectorized Median class algorithm and implemented through memory fine tuning and the use of GPU registers, our median drastically outperforms existing implementations, resulting, as far as we know, in the fastest median filter to date.

1 Introduction

First introduced by Tukey in [6], the median filter is a simple process which consists, for each pixel of an image, in replacing its gray-level value by the median value of its neighbors, taken in a $n = k \times k$ window centered on this very pixel. Median filtering has been widely studied since then, and many researchers have proposed efficient implementations of it, adapted to various hypothesis, architectures and processors. Originally, its main drawbacks were its compute complexity, its non linearity and its data-dependent runtime. Several researchers have addressed these issues and designed, for example, efficient histogram-based median filters featuring predictable runtimes [3, 7]. More recently, authors have managed to take advantage of the newly opened perspectives offered by modern GPUs, to develop CUDA-based filters such as the Branchless Vectorized Median filter (BVM) [2, 4] which allows very interesting runtimes and the histogram-based, PCMF median filter [5] which was the fastest median filter implementation to our knowledge.

The use of a GPU as a general-purpose computing processor raises the issue of data transfers, especially when kernel runtime is fast and/or when large data sets are processed. In certain cases, data transfers between GPU and CPU are slower than the actual computation on GPU, even though global GPU processes can prove faster than similar ones run on CPU. So as to obtain high throughput values, it is therefore critical to address both sides of the problem: data transfers and GPU kernel intrinsic performance.

In the following section, we detail our implementation of the median filter, called PRMF for Parallel Register-only Median Filter. For more concision and readability, our coding will be restricted to 8 or 16 bit gray-level input images whose height (H) and width (W) are both multiples of 512 pixels. Let us also point out that the following implementation, targeted on Nvidia Tesla GPU (Fermi architecture, compute capability 2.x), may easily be adapted to other models e.g. those of compute capability 1.3.

2 Implementing a fast median filter

2.1 Basic principles

Designing a 2-D median filter basically consists in defining a square window $H(i, j)$ for each pixel $I(i, j)$ of the input image, containing $n = k \times k$ pixels and

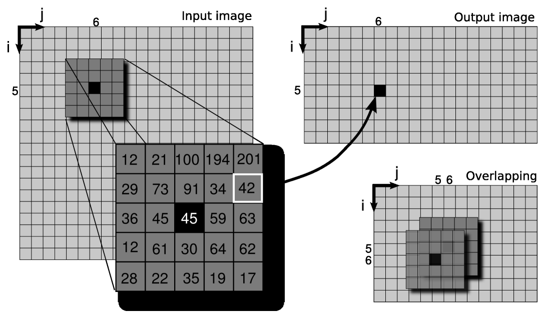


Figure 1: Illustration of 5×5 median filtering, applied on pixel of coordiantes (5,6). Bottom right: window overlapping.

centered on $I(i,j)$. The output value $I'(i,j)$ is the median value of the gray level values of the $k \times k$ pixels of $H(i,j)$. Figure 1 illustrates this principle with an example of a 5x5 median filter applied on pixel $I(5,6)$.

Obviously, one key issue is the selection method that identifies the median value, which can be done using either histogram-based or sorting methods. But, as shown in figure 1, since two neighboring pixels share part of the values to be sorted, a second key issue is how to rule redundancy between consecutive positions of the running window $H(i,j)$.

2.2 Data transfers

CUDA-enabled devices offer several memory types, each with its own levels of latency and speed. The most versatile is the generic global memory, but it is also the one with the highest latency value (around 400 clock cycles) and its transfer rate is subject to access pattern constraints. Among other memory types, only constant memory and texture memory are usable to store data from CPU memory. On CPU side, CUDA langage extension features a memory allocation function able to allocate non-pageable memory called *pinned-memory*, which is an efficent alternative to classical allocation as it allows more direct access to stored data. The drawback is that it has to be use sparingly in order to avoid an early memory overflow. Benchmarking all possible combinations led us to adopt the memory management described in Algorithm 1. Input image data is stored in the GPU's texture memory so as to benefit from the 2-D caching mechanism which transparently preloads neighbor pixel values when fetching one particular pixel. It reduces memory access latency. After kernel execution, copying output image back to CPU memory is done by use of pinned memory, which drastically accelerates data transfer (see Tables 1 and 2 for precise timings).

Algorithm 1: Global memory management on CPU and GPU sides.

```
1 allocate and populate CPU memory h_img_in;
2 allocate CPU pinned-memory h_img_out;
3 allocate GPU global memory d_img_out;
4 allocate GPU texture memory tex_img_in;
5 copy data from h_img_in to tex_img_in;
6 kernel<<< gridDim,blockDim>>>   /* to d_img_out */;
7 copy data from d_img_out to h_img_out ;
```

2.3 Using registers

As register access is at least 20 times faster than all the other memory types available on the GPU, it is natural to try to use them as a mean to store temporary data inside our kernels, keeping in mind that on the *fermi* architecture, each individual thread can use a maximum of 63 registers within the limit of 32K per thread block. However, it must be noticed that a high register usage, though below the above-cited limits, may result in a loss of performance due to a lower parallelism level inside each block, *i.e.* less threads actually run in parallel. Consequently, it remains important to use registers sparingly in order to preserve high pixel throughput values: to do so, we use the *forgetful selection* algorithm. Its principle is to construct a list of R_n pixels values, taken among the $n = k \times k$ ones of the window. Then we identify and eliminate (forget) both elements showing the maximum and the minimum values in the list. Finally, we include one of the values left apart of the original list. This process is repeated until no more value can be included in the list. The remaining element in the list is the global median value. Two important points should be notice:

- this algorithm has a fixed number of steps, equals to $\left(n - \lceil \frac{n}{2} \rceil\right)$, which implies that all threads have almost the same workload, despite the data dependency of the extrema identification step.
- for small windows, the whole original list can be put in registers.

The number R_n of elements in the initial list is chosen as the minimum element count which allows to identify the global median value through the above process. It is obtained by considering the constraint of keeping the global median in the list at each elimination step. This lead to:

$$R_n = \lceil \frac{n}{2} \rceil + 1$$

Which represents the minimum register count needed to perform the forgetful selection (one register per element). It is also noticeable that the second and following elimination steps use less registers than R_n as two elements are eliminated and one element added at each step.

Figure 2 illustrates the *forgetful selection* process applied to a 3×3 pixel median filter. For clarity reasons, the nine values have been represented in a row. The process begins with $R_9 = 6$ elements and ends after 4 iterations, when there is no more candidate element. This also corresponds to the state where there is only one element in the list: the median value. The selection of both *extrema* is implemented through a basic 2-element swapping function, which will be detailed in the following. This ensures that the GPU kernel code is free of divergent branches liable to severely impact performances.

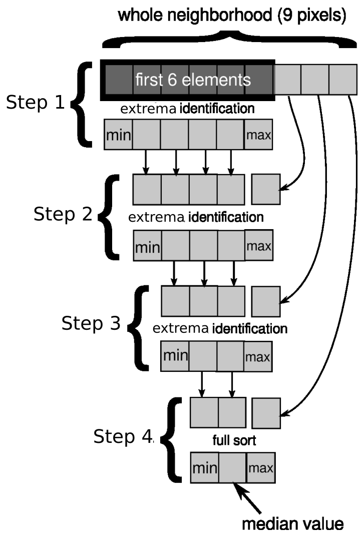


Figure 2: Determination of the Median value by the forgetful selection process, applied to a 3×3 neighborhood window.

2.4 Hiding Latencies

Optimizing a GPU kernel also means hiding latencies potentially generated by memory accesses and data dependent instruction calls. Indeed, modern GPUs are able to pipeline instruction processes so as to reduce the average latency of an instruction sequence: this capability is called ILP (Instruction Level Parallelism). As for global memory accesses, when two or more consecutive arithmetic operators manipulate (read or write) independant variables, only the first access generates latency. The massive thread parallelism of CUDA-enabled devices helps in hiding those latencies transparently but, analysing the actual computation performed by each thread, optimization may be taken a few steps further:
First, we maximize the Instruction Level Parallelism inside the *forgetful selection* method by re-arranging the instruction sequence of an incomplete sorting network [1] so as to reduce the data dependency of consecutive instructions and thus preventing frequent empty pipelines Figure 3 shows the scheduling of the first extrema identification step of a 5×5 median filter, carried out with $R_{25} = 14$ elements. Each arrow represents one call to the 2-element swapping function: after the call, the starting point symbolizes the lowest value element and the ending arrow points out the highest one. In addition, horizontal dashed lines separate packs of independants instructions.This clearly maximizes the ILP.
Second, in order to reduce the effect of global memory access latency, each thread performs the computation of two neighbor input pixels instead of just one. Additionally, window overlapping is exploited in order to minimize the increase of register count per thread

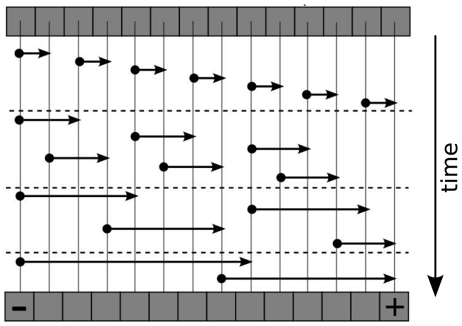


Figure 3: First extrema identification step of the forgetful selection, applied to a 5×5 median filter. It begins with $R_{25} = 14$ unsorted elements and ends with the minimum value at the first position (left) and the maximum at the last position (right).

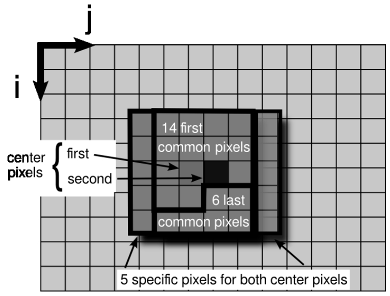


Figure 4: Reducing register count in a $5{\times}5$ register-only median kernel processing 2 input pixels. The first 7 forgetful selection steps are common to both processed center pixels: the first one needs 14 pixels, leaving 6 more pixels to be processedone after another.

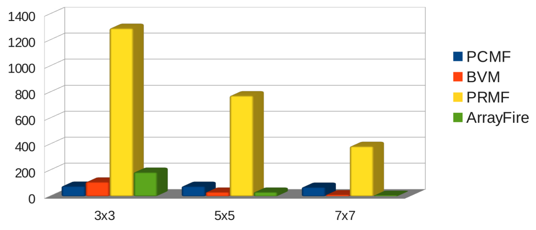
brought by this *2 pixels per thread* rule. The register count per thread block is easily kept unchanged by dividing the block size by 2, while preserving the grid size. Trying to benefit from overlapping cannot be achieved by additional computation after identification of the first median values, as it can be done with histogram-based solutions. Instead, both selections have to be carried out in parallel.
Considering that the R_n elements of the first selection step can be taken anywhere in the window, we begin the selection with R_n elements, choosen among those shared by both windows. This only makes sense if two consecutive windows share at least R_n elements, which obviously is the case as they actually share $S_n = n - k = n - \sqrt{n}$ pixels, which is always greater than R_n (or equal for the 3×3 median filter ($n = 9$). The $(S_n - R_n + 1)$ first selection steps can then be considered common to both windows, leaving only the k non-shared pixels of each window to be processed separately. This technique saves $k + 1$ registers for each pair of input pixels, which means that each thread block now uses fewer registers while processing the same pixel count, thus allowing a higher level of parallelism. Figure 4 illustrates this by representing the different classes of pixels in the 5×5 median example: the first $R_{25} = 14$ common pixels are used to generate the vector to be sorted at the first step, 6 more steps are carried out with the remaining common pixels before entering into separate sorting processes.

3 Results

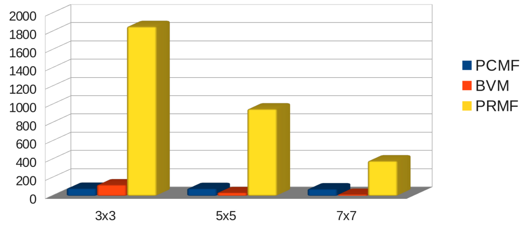
Runtimes have been obtained by averaging 1000 executions on a C2070 GPU card hosted by a system with one Xeon E56202.40GHz processor running a linux kernel 2.6.18x86_64 and CUDA v4.0. Each kernel has been run on 8 bit and 16 bit images of sizes $512{\times}512$, $1024{\times}1024$, $2048{\times}2048$ and $4096{\times}4096$. As mentioned in section 2.2, our implementation optimizes data transfers: Tables 1, 2 and 3 detail times and relative costs of data transfers between CPU and GPU. Transfers into texture memory are a bit slower than those done through pinned-memory but, as said above, the associated 2D-caching mechanism allow a great performance improvement of the later data fetching. The rightmost columns of Tables 1 and 2 allow to compare our way of transferring data against standard global memory transfers. It reveals that our choices make transfers 15% to 75% faster than naive ones.
In addition, Table 3 shows the relative costs of data transfers against total process times. Analysing these values confirms the relevance of our approach: data transfers between CPU and GPU represents at

least 13% of the total runtime, for 8 bit large image and window sizes, but up to 82% for 16 bit small image and window size. Consequently, it had to be minimized as much as possible.

Like many authors, we have used the pixel throughput value as our main performance indicator. It includes kernel runtime as well as transfer times to and from the GPU. To evaluate the absolute performance of our implementations, we have also measured the maximum effective pixel throughput that our GPU/host couple is able to achieve. Knowing such a peak value helps in deciding on further investigation. We performed this measurement by running a dummy kernel that fetches the gray-level of each pixel in texture memory and outputs it into global memory exclusive of any other instruction. Running the dummy kernel on all image sizes and depths brought the peak values gathered in Table 4, which shows that the larger the image is, the higher the expected throughput is. Kernel runtimes and throughtput values are presented in Table 5, with separate global throughput values for 8 and 16 bit depths (T_8 and T_{16}) as transfer time costs vary, while kernel runtime is not influenced by the gray-level depth. Though our implementation does not show a constant runtime, but follows a classical $n.log(n)$ law, it proves from 5.3 to 10 times faster than the one in second position and can achieves up to 1850 Mpix/s. It is confirmed by Figure 5 which compares the throughput values of several implementations against ours for common small window sizes. Moreover, focusing on the 3×3 median filter, the actual pixel throughput achieved by our implementation reaches more than 75% of the absolute peak throughput value.
It is also noticeable that our kernel algorithm is quite similar to the one implemented in *ArrayFire* (at least for 3 median filter). That led us to try and find out what had brought such high speedup in our implementation. For this purpose, we inserted the 3 *ArrayFire* median filter in our own coding structure in order to benefit from the optimal data transfers. Little kernel modifications had also been done to allow the fetching of data from texture memory. This setup allows *ArrayFire* kernel to achieve 670 Mpix/sec, *i.e.* 3.7 times higher than the original. The remaining x2.7 speedup is then brought by our kernel implementation itself.



(a) 512×512 pixel input image.



(b) 4096×4096 pixel input image

Figure 5: Pixel throughput value comparison, in million pixels per second, of several implementation against our PRMF. From left to right: PCMF, BVM, PRMF, ArrayFire (impossible with 4096×4096)

time costs→ image size↓	to GPU (ms)	from GPU (ms)	**Total** (ms)	**Gmem** (ms)
512×512	0.08	0.06	**0.14**	0.23
1024×1024	0.24	0.19	**0.43**	0.81
2048×2048	0.85	0.68	**1.53**	2.15
4096×4096	3.27	2.61	**5.88**	7.10

Table 1: Time cost of data transfer for each image size in 8 bit gray-level format on C2070 GPU. In column Gmem, simple global-memory-only transfers times are shown for comparison.

4 Conclusion

We proposed a very high speed, small window median filter which makes it possible to process, for example, almost 900 high definition (1080p) images per second. Due to the lack of available source code, our

time costs→ image size↓	to GPU (ms)	from GPU (ms)	**Total** (ms)	**Gmem** (ms)
512×512	0.14	0.10	**0.24**	0.42
1024×1024	0.45	0.35	**0.80**	1.23
2048×2048	1.59	1.32	**2.91**	3.83
4096×4096	6.21	5.21	**11.42**	13.16

Table 2: Time cost of data transfer for each image size in 16 bit gray-level format on C2070 GPU. In column Gmem, simple global-memory-only transfers times are shown for comparison.

Gray-level format→ image size↓	$\mathbf{T_8}$	$\mathbf{T_{16}}$
512×512	1598	975
1024×1024	2101	1200
2048×2048	2359	1308
4096×4096	2444	1335

Table 4: Maximum effective pixel throughput values for T_8 and T_{16} (in MPixel per second) on C2070, achieved when processing 8 and 16 bit-coded gray-level images.

Window size→ Image size - depth.↓		**3×3**	**5×5**	**7×7**
512^2	8 bits	73%	44%	20%
	16 bits	82%	57%	29%
1024^2	8 bits	68%	37%	15%
	16 bits	80%	53%	25%
2048^2	8 bits	66%	34%	14%
	16 bits	79%	59%	23%
4096^2	8 bits	65%	33%	13%
	16 bits	78%	50%	23%

Table 3: Relative cost of data transfers, in percent of total runtime, for 8 and 16 bit-coded gray-level images and run by C2070 GPU.

Window size→ Image size - perf.↓		**3×3**	**5×5**	**7×7**
512^2	t (ms)	0.05	0.19	0.60
	T_8 (Mpix/s)	1291	773	348
	T_{16} (Mpix/s)	865	607	307
1024^2	t (ms)	0.20	0.74	2.39
	T_8 (Mpix/s)	1644	889	371
	T_{16} (Mpix/s)	1045	692	329
2048^2	t (ms)	0.79	2.95	9.53
	T_8 (Mpix/s)	1805	936	379
	T_{16} (Mpix/s)	1130	729	338
4096^2	t (ms)	3.17	11.77	38.06
	T_8 (Mpix/s)	1854	951	382
	T_{16} (Mpix/s)	1151	738	340

Table 5: Kernel runtimes and global pixel throughput of fast median kernels processing 8 and 16 bit-coded gray-level images and run by C2070 GPU.

comparison is based on the most recent results published in [5], obtained with the same GPU as ours and with 8 bit-coded gray-level images. While the algorithm implemented here is similar to the one in *ArrayFire*, the main difference resides in our fine tuning of the implementation, on both data transfers and kernel side, that leads to the fastest GPU median filter known to date with 1854 MPix/s. Let us also note that such considerable throughput values come very close to the peak effective pixel throughput value of 2444 Mpix/s allowed by our developpement platform. Consequently further investigation would likely bring little performance improvement. Other types of algorithms can benefit from all or at least part of these optimizations, mainly the memory management. As we did with convolution filters, which is the subject of a next publication, all linear or non-linear neighborhhod filters can be successfully treated that way.

References

[1] Batcher KE (1968) Sorting networks and their applications. In: Proceedings of the April 30–May 2, 1968, spring joint computer conference, ACM, New York, NY, USA, AFIPS '68 (Spring), pp 307–314, DOI 10.1145/1468075.1468121, URL http://doi.acm.org/10.1145/1468075.1468121

[2] Chen W, Beister M, Kyriakou Y, Kachelries M (2009) High performance median filtering using commodity graphics hardware. In: Nuclear Science Symposium Conference Record (NSS/MIC), 2009 IEEE, pp 4142–4147, DOI 10.1109/NSSMIC.2009.5402323

[3] Huang TS (1981) Two-Dimensional Digital Signal Processing II: Transforms and Median Filters. Springer-Verlag New York, Inc., Secaucus, NJ, USA

[4] Kachelriess M (2009) Branchless vectorized median filtering. In: Nuclear Science Symposium Conference Record (NSS/MIC), 2009 IEEE, pp 4099 –4105, DOI 10.1109/NSSMIC.2009.5402362

[5] Snchez R, Rodrguez P (2012) Highly parallelable bidimensional median filter for modern parallel programming models. Journal of Signal Processing Systems pp 1–15, DOI 10.1007/s11265-012-0715-1, URL http://dx.doi.org/10.1007/s11265-012-0715-1

[6] Tukey JW (1977) Exploratory Data Analysis. Addison-Wesley

[7] Weiss B (2006) Fast median and bilateral filtering. In: ACM SIGGRAPH 2006 Papers, ACM, New York, NY, USA, SIGGRAPH '06, pp 519–526, DOI 10.1145/1179352.1141918, URL http://doi.acm.org/10.1145/1179352.1141918