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

February 15, 2013

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 power of noise while minimizing edge blurring. Currently, its existing algorithms and implementations have proved quite efficient but leave room for improvement as far as processing speed is concerned, which has lead 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 1.8 billion pixels per second. 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 [8], 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 adressed these issues and designed, for example, efficient histogram-based median filters featuring predictable runtimes [4, 9]. More recently, authors managed to take advantage of the newly opened perspectives offered by modern GPUs, to develop such CUDA-based filters such as the Branchless Vectorized Median filter (BVM) [5, 2] which allows very interesting runtimes and the histogram-based, PCMF median filter [6] which was the fastest median filter implementation known to us.

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. In the following section, we propose the overall code structure to be used with our median kernels. 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 General structure

Algorithm 1 describes how data is handled in our code. Input image data is stored in the GPU's texture memory so as to benefit from the 2-D caching mechanism. After kernel execution, copying output image back to CPU memory is done by use of pinned memory, which drastically accelerates data transfer.

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

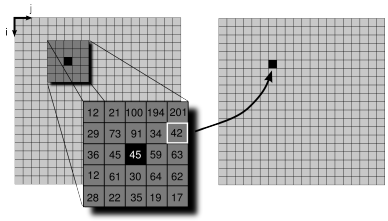
1: allocate and populate CPU memory **h_in**;
2: allocate CPU pinned-memory **h_out**;
3: allocate GPU global memory **d_out**;
4: declare GPU texture reference **tex_img_in**;
5: allocate GPU array **array_img_in**;
6: bind **array_img_in** to texture **tex_img_in**;
7: copy data from **h_in** to **array_img_in**;
8: kernel<<< gridDim,blockDim>>> `/* to d_out */`;
9: copy data from **d_out** to **h_out** ;

3 Implementing a fast median filter

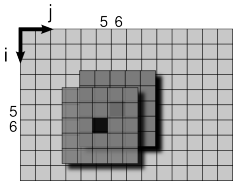
3.1 Basic principles

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 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 $H(i,j)$.



(a) 5×5 median filtering applied on pixel of coordinates (5,6)



(b) window overlapping in 5×5 median filtering

Figure 1: Illustration of 5×5 median filtering

3.2 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 turn to the use of registers as a means to store temporary data, 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. Consequently, it is important to use registers sparingly in order to preserve high pixel throughput values: to do so, we use the iterative *forgetfull selection* algorithm. Its principle is, at each iteration, to identify then to eliminate (forget) both elements showing the maximum and the minimum values among the current list, then to add the next candidate element left in the original list, till none is left; the last value remaining actually is the global median value. Figure 2 illustrates the *forgetfull selection* process applied to a 3×3 pixel median filter. For clarity reasons, the nine values have been represented in a row. The minimum register count R_n needed to start this iterative

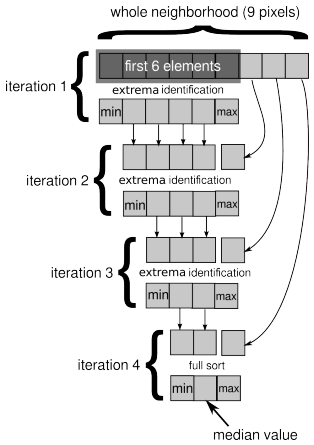


Figure 2: Determination of the Median value by the Forgetfull selection process, applied to a 3×3 neighborhood window.

selection process among $k \times k$ values is given by:

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

The selection of both *extrema* is implemented through an basic 2-element swapping function. This ensures that the GPU kernel code is free of divergent branches liable to severely impact performance.

3.3 Hiding Latencies

Optimizing a GPU kernel also means hiding latencies. The offered massive thread parallelism helps in doing so transparently but, considering the actual computation performed by each thread, optimization may be taken a few steps further:
First, we maximized the Instruction Level Parallelism inside the *forgetfull selection* method by re-arranging the instruction sequence of an incomplete *bitonic sort* [1, 3] so as to reduce the data dependency of consecutive instructions.
second, we divided thread block size by 2, having each thread perform the computation of two neighbor input pixels instead of just one, thus keeping grid size unchanged while reducing the effect of global memory access latency. Additionally, window overlapping also had to be managed, as illustrated by Figure 1, in order to minimize the increase of register count per thread brought by the new computing organization. Two $k \times k$ consecutive neighboring windows share $S_n = n - \sqrt{n}$ pixels, which is more than the number of registers needed to perform the median selection, i.e. R_n (or equal for 3×3 median filter). The $(S_n - R_n + 1)$ first selection stages can then be considered common to both windows, leaving only the k non-shared pixels of each window to be processed separately. The above 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 3 illustrates this by representing the different classes of pixels in the 5×5 median example.

3.4 Compute complexity

Arithmetic instructions and texture fetches are easy to count but the number of element swaps needed to select the median value is data dependent and only its maximum can be evaluated. The incomplete sorting (*forgetful selection*) and the redundancy management performed when outputing two pixels per thread lead to the instruction count below:

- 5 integer multiplications and 5 integer additions to compute thread indexes and ouput pixel coordinates.
- $2n-1$ additions to compute neighbor pixel coordinates.
- $n - \sqrt{n}$ texture fetches to load gray-level values.
- within a m-element vector and according to our selection method, the maximum

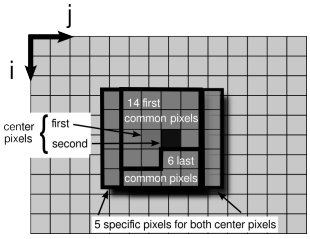


Figure 3: Reducing register count in a 5×5 register-only median kernel processing 2 input pixels. The first 7 forgetful selection stages are common to both processed center pixels.

number of element swaps needed to move the minimum value at first position and the maximum at last is given by:

$$sc(m) = -2 + \lceil \frac{3.m}{2} \rceil$$

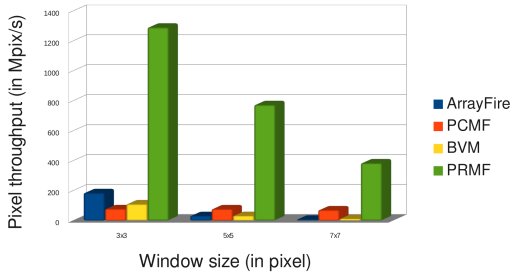
Consequently, the number of element swaps needed by the entire selection of two median values performed by one thread is inferior or equal to:

$$\left(\sum_{i=\lceil \frac{n}{2} \rceil + 1}^{n-\sqrt{n}} sc(i) \right) + 2 \left(\sum_{i=n-\sqrt{n}+1}^{n} sc(i) \right)$$

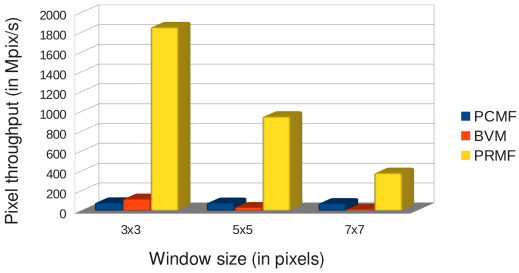
The above sum equals 42,156 and 474 for typical window sizes of 3×3, 5×5 and 7×7. The total instruction count is thus kept near a $O(nlog(n))$ rule.

4 Experiments

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 images of sizes 512×512, 1024×1024, 2048×2048 and 4096×4096. Like many authors, we computed the pixel throughput value as our main performance indicator. It includes kernel runtime as well as transfer times to and from the GPU. We also measured the maximum effective pixel throughput that our GPU/host couple is able to achieve, by running an identity kernel that fetches the gray-level of each pixel in texture memory and outputs it into global memory exclusive of any other instruction. Knowing this peak value allows us to evaluate the potential for improvement of each kernel and helps in deciding on further investigation. Those peak values are detailed in Table 2, which shows that the larger the image, the higher the expected throughput.



(a) 512×512 pixel input image.



(b) 4096×4096 pixel input image

Figure 4: Pixel throughput value comparison of several implementation against our PRMF.

5 Results

The main contribution of this work is to show how to tune a CUDA GPU implementation in order to achieve highest pixel throughput values. Runtimes, as well as pixel throughput values are gathered in Table 3. Figure 4

Gray level format	**8 bits**			**16 bits**		
time costs→ image size (pixels)↓	to GPU (ms)	from GPU (ms)	Total (ms)	→GPU (ms)	→GPU (ms)	Total (ms)
512×512	0.08	0.06	0.14	0.14	0.10	0.24
1024×1024	0.24	0.19	0.43	0.45	0.35	0.80
2048×2048	0.85	0.68	1.53	1.59	1.32	2.91
4096×4096	3.27	2.61	5.88	6.21	5.21	11.42

Table 1: Time cost of data transfer for each image size and gray-level format on C2070 GPU.

compares the throughput values of several implementations against ours, labeled PRMF for Parallel Register-only Median Filter. Due to the lack of available source code, comparison is based on the most recent results published in [7], 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*, what makes the difference is our fine tuning of the implementation 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.

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 2: 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 - 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 3: Runtime and pixel throughput of fast median kernels processing 8 and 16 bit-coded gray-level images and run by C2070 GPU.

References

[1] K. E. Batcher. Sorting networks and their applications. In *Proceedings of the April 30–May 2, 1968, spring joint computer conference*, AFIPS '68 (Spring), pages 307–314, New York, NY, USA, 1968. ACM.

[2] Wei Chen, M. Beister, Y. Kyriakou, and M. Kachelries. High performance median filtering using commodity graphics hard-

ware. In *Nuclear Science Symposium Conference Record (NSS/MIC), 2009 IEEE*, pages 4142–4147, 24 2009-nov. 1 2009.

[3] Thomas H Cormen, Charles E Leiserson, Ronald L Rivest, and Clifford Stein. *Introduction to algorithms*. MIT press, 2001.

[4] Thomas S. Huang. *Two-Dimensional Digital Signal Processing II: Transforms and Median Filters*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1981.

[5] M. Kachelriess. Branchless vectorized median filtering. In *Nuclear Science Symposium Conference Record (NSS/MIC), 2009 IEEE*, pages 4099 –4105, 24 2009-nov. 1 2009.

[6] R.M. Sanchez and P.A. Rodriguez. Bidimensional median filter for parallel computing architectures. In *Acoustics, Speech and Signal Processing (ICASSP), 2012 IEEE International Conference on*, pages 1549 –1552, march 2012.

[7] RicardoM. Snchez and PaulA. Rodrguez. Highly parallelable bidimensional median filter for modern parallel programming models. *Journal of Signal Processing Systems*, pages 1–15, 2012.

[8] John Wilder Tukey. *Exploratory Data Analysis*. Addison-Wesley, 1977.

[9] Ben Weiss. Fast median and bilateral filtering. In *ACM SIGGRAPH 2006 Papers*, SIGGRAPH '06, pages 519–526, New York, NY, USA, 2006. ACM.