BIDIMENSIONAL MEDIAN FILTER FOR PARALLEL COMPUTING ARCHITECTURES

Ricardo M. Sánchez *Paul A. Rodríguez*

Department of Electrical Engineering
Pontificia Universidad Católica del Perú
Lima, Perú

ABSTRACT

The median filter is a non-linear filter used for removal of salt and pepper noise from images. Each pixel of the image is replaced by the median of its surrounding elements, the median value is calculated by sorting the data. The complexity of the sorting algorithms used on the median filters are $O(n^2)$ or $O(n)$, depending on the kernel size. Those algorithms were formulated for scalar single processor computers, with few of them successfully adapted and implemented for computer with a parallel architecture.

In this paper we present a novel sorting algorithm, with $O(n)$ computational complexity and a highly parallelizable structure, based on the Complementary Cumulative Distribution Function. Furthermore, a 2D median filter based on our proposed sorting algorithm can achieve $O(1)$ complexity. We have implemented our proposed algorithm in two parallel architectures: SIMD Intel and CUDA, which have a throughput of 12.8 and $35 \sim 57$ megapixels per second respectively.

Index Terms— Nonlinear filters, Parallel Algorithms

1. INTRODUCTION

The median filter is a basic operation for digital image processing, as it removes salt and pepper noise while preserving the edges of the image. This type of noise is usually generated by lost packets on digital transmission, noisy digital channels, or errors on the acquisition system (dust on the lens, defective pixels on the sensor) [1]. However, the usage of the 2D median filter (originally proposed in 1974 [2]) is restricted by its high computational cost and its non-linear nature.

The algorithms for median filtering differ one from each other by the sorting algorithm that each one use. For large kernels it is typical to use sorting algorithms with $O(n)$ computational complexity, where n is the kernel size. Those algorithms require additional memory for data structures or containers. Also, they can reduce memory access by keeping the previous results and use them to calculate de median of the next pixel [3, 4, 5]. Furthermore, $O(1)$ computational complexity can be obtained by storing partial results of the containers needed to get the median value [6, 7]. For small kernels, $O(n^2)$ sorting algorithms are faster than the $O(n)$ ones, and some of them make use of vector processors to achieve a very high performance [8, 9, 10]. This improvement is due to the data parallelism of the underlaying sorting algorithm, as the Sorting Network [8], that allows to sort consecutive datasets with common elements. Parallel sorting algorithms, for multiprocessors systems, have been proposed, but they have high communication cost [11] or unbalanced computational load [12].

In this paper we propose a novel sorting algorithm that can be implemented efficiently on parallel multiprocessors systems. This algorithm is based on the Complementary Cumulative Distribution Function ($ccdf$) [13] and it has similar properties to the ones based on probability mass function (pmf or histograms). With this algorithm we develop a new median filter algorithm that is efficient on parallel systems, such as CUDA-enabled graphics cards, and is benefited by the optimizations originally proposed for the histogram (pmf) based algorithms (i.e.: Constant-Time Median Filter [6]). The implementation of the new median filter has been developed for CUDA-enabled graphics card and for Intel processors, as each one offer different parallel architectures.

This paper is organized as follows: in Section 2 the novel sorting algorithm and the median filter algorithms are described. In Section 3 we present the complexity analysis and evaluate its parallel capabilities of our proposed algorithm. We describe the median implementations used and show the computational results on Section 4. On Section 5 we discuss the results and give our concluding remarks

2. ALGORITHMS DESCRIPTION

2.1. CCDF-sorting

This new algorithm for sorting data needs to generate a vector with the complementary cumulative distribution function (ccdf) from the data set to be sorted. From the generated auxiliary vector we can obtain the kth biggest number in the data set.

Given the vector $\mathbf{x} = [x_0, x_1, \ldots, x_{n-1}]$ with n elements, where $x_i \in \mathbb{N}$ and $a \leq x_i \leq b$, the complementary cumulative distribution function, or reliability function [13], is defined by:

$$\bar{F}_{\mathbf{x}}(j) = 1 - F_{\mathbf{x}}(j) = 1 - Pr(\mathbf{x} \leq j) = Pr(\mathbf{x} > j)$$

where $F_{\mathbf{x}}(j)$ is the cumulative distribution function of the vector $\mathbf{x}$ and $Pr(\mathbf{x} > j)$ is the probability of $x_i > j$. For the sorting algorithm we replace the probability function $Pr(\mathbf{x} > j)$ with a counting function $C_j(\mathbf{x})$:

$$C_j(\mathbf{x}) = \sum_{i=0}^{n-1} I_{[x_i > j]} \tag{1}$$

where $I_{[x_i > j]}$ is the indicator function. It is straightforward to show that $\bar{F}_{\mathbf{x}}(j)$ monotonically decreases to zero.

Now, in order to get the sorted vector $\mathbf{y}$, we need to set the auxiliary vector $\boldsymbol{\tau} = \{\tau_j\} = ccdf(\mathbf{x}), j \in [a, b], \tau_j = C_j(\mathbf{x}) \in [0, n]$. Then $\mathbf{y} = \{y_k\} = ccdf(\boldsymbol{\tau}), k \in [0, n-1], y_k = C_k(\boldsymbol{\tau}) \in [a, b]$ is the sorted vector of $\mathbf{x}$. For example, given the set $\mathbf{x} = [4, 6, 2, 9, 8]$, with $n = 5, a = 0$ and $b = 10$, we can compute $\boldsymbol{\tau} = [5, 5, 4, 4, 3, 3, 2, 2, 1, 0, 0]$ and the sorted vector $\mathbf{y} = [9, 8, 6, 4, 2]$.

2.2. Median filter with CCDF-sorting

To obtain the kth biggest element of $\mathbf{x}$ from the auxiliary vector $\boldsymbol{\tau}$ we only need to calculate $C(\boldsymbol{\tau} > k)$. Following this, if we require the minimum, maximum and median value we need to set $k = n-1$, $k = 0$ and $k = (n-1)/2$ respectively, we assume that n is odd.

An important property of the *ccdf* is the separability. Given the set $\mathbf{x}$, with $n_{\mathbf{x}}$ elements and the subsets $\mathbf{a}, \mathbf{b}$ with $n_{\mathbf{a}}, n_{\mathbf{b}}$ elements respectively, such that $n_{\mathbf{x}} = n_{\mathbf{a}} + n_{\mathbf{b}}$, $\mathbf{x} = \mathbf{a} \cup \mathbf{b}$, it can be shown that

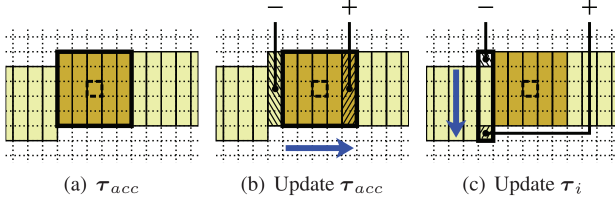
$$\boldsymbol{\tau}^{(\mathbf{x})} = \boldsymbol{\tau}^{(\mathbf{a})} + \boldsymbol{\tau}^{(\mathbf{b})}.$$

Given an additional dataset $\mathbf{y}$, with $n_{\mathbf{y}}$ elements and the subset $\mathbf{c}$ with $n_{\mathbf{c}}$ elements, such that $n_{\mathbf{y}} = n_{\mathbf{b}} + n_{\mathbf{c}}$, $\mathbf{y} = \mathbf{b} \cup \mathbf{c}$, then it is easy to show that:

$$\boldsymbol{\tau}^{(\mathbf{y})} = \boldsymbol{\tau}^{(\mathbf{x})} + \boldsymbol{\tau}^{(\mathbf{c})} - \boldsymbol{\tau}^{(\mathbf{a})}.$$

This property allows us to reduce memory access by keeping a $\boldsymbol{\tau}_{acc}$ and updating it for the next pixel. The update is done by adding the *ccdf* of the new pixels and subtracting the *ccdf* of the old ones.

This approach was originally proposed for the histogram-based median filter [3] and it can be applied for the *ccdf*-based median filter as well. Another approach is to generate partial histograms for a whole row of the image and getting the kernel's histogram using those partial histograms only. For the next row the partial histograms are updated (Fig. 1) [6].



(a) $\boldsymbol{\tau}_{acc}$ (b) Update $\boldsymbol{\tau}_{acc}$ (c) Update $\boldsymbol{\tau}_i$

Fig. 1. $O(1)$ complexity approach (originally proposed by [6]).

This procedure can be also applied for the ccdf-based median filter. The computational complexity is lowered and the memory access are reduced, but additional memory is required. Algorithm 1 describes the proposed algorithm for median filtering.

Algorithm 1: Parallel Ccdf-based Median Filter (PCMF)

Input: I: $N \times M$ matrix to be filtered.
Kernel size: $k \times k$, k odd.
Output: O: $N \times M$ filtered matrix
begin
  foreach *i-th Column in I* **do**
    $x_{i,l} \leftarrow \{I_{i,n} : |n - l| \leq k\}$
    $\boldsymbol{\tau}_i \leftarrow$ `CCDF(`$\mathbf{x}_i$`)`
  foreach *j-th Row in I* **do**
    $\boldsymbol{\tau}_{acc}^{(i,j)} \leftarrow \sum \boldsymbol{\tau}_{i,j-n}, \quad n = -k : k$
    foreach *i-th Pixel in j-th Row* **do**
      $O_{i,j} \leftarrow$ `MedianFromCCDF(`$\boldsymbol{\tau}_{acc}^{(i,j)}$`)`
      `UpdateCCDFfromTempVectors(`$\boldsymbol{\tau}_{acc}^{(i,j)}$`)`
    foreach *i-th Column in I* **do**
      `UpdateCCDFfromMatrix(`$\boldsymbol{\tau}_i$`)`

---

3. ALGORITHM ANALYSIS

3.1. Computational Complexity

In what follows we analyze the computational complexity of the functions `CCDF(x)` and `MedianFromCCDF(`$\boldsymbol{\tau}_{\mathbf{x}}$`)` (see Algorithm 1), since our proposed algorithm is based on this two basic operations.

- `CCDF(x)`: From Eq (1), to get the *ccdf* of a vector $\mathbf{x}$ of n elements, with $a \leq x_i \leq b$, the number of operations needed is $\boldsymbol{\tau}_{\mathbf{x}} = (b-a)((n-1)$ additions$+n$ comparisons$) = O(n)$
- `MedianFromCCDF(`$\boldsymbol{\tau}_{\mathbf{x}}$`)`: Given $\boldsymbol{\tau}_{\mathbf{x}}$, the number of operations needed to get the median value is $median(\boldsymbol{\tau}_{\mathbf{x}}) = (b-a-1)$ additions $+ (b-a)$ comparisons $= O(1)$.

  The complexity to get the median value of a vector $\mathbf{x}$ is $median(\mathbf{x}) = O(n) + O(1) = O(n)$.

With this results, and given I, an $N \times M$ image, the computational complexity of the PCMF (Algorithm 1) for the whole image is:

$$\underbrace{NO(n^2)}_{\text{Initialize } \boldsymbol{\tau}} + \underbrace{NMO(1)}_{\text{Median Values}} + \underbrace{NMO(1)}_{\text{Update } \boldsymbol{\tau}} = O(n^2)$$

As the initialization stage takes place only one time per row, a better representation for the asymptotical behavior of the algorithm is to find its complexity per pixel. This value is:

$$\frac{\text{Complexity}}{\text{\# of Pixels}} = \frac{NO(n^2) + 2NMO(1)}{MN} = 2O(1) + O(n^2)/M$$

Then, for a big image (large M), the factor $\frac{O(n^2)}{M}$ tends to 0. This consideration allows us to say that the complexity of the PCMF is $O(1)$.

Regarding memory usage, our proposed algorithm, the PCMF, needs $(b-a+n-1)N = O(n)$ additional memory. This is costly for parallel architectures with limited memory, such CUDA-enabled graphics cards, and it can make prevent the condition $\frac{O(n^2)}{M} \to 0$ to be held.

3.2. Parallel Capabilities and Memory Access

In this subsection we are going to compare the *ccdf* based median filter against the histogram based median filters. We will focus on the parallel capabilities and memory access patterns of both methods. We will consider a parallel computing model with many concurrent threads available, concurrent read (for aligned data or *SIMD* access pattern) and penalized exclusive read and writes access (for random memory access).

In that context, calculating and updating $\boldsymbol{\tau}_{\mathbf{x}}$ needs the same number of steps as calculating and updating the histogram (Eq. 1). The difference is the memory access pattern. For the histograms, random access is used, while for the *ccdf* an ordered access pattern is used. For the *ccdf* we can use $(b-a)$ threads per $\boldsymbol{\tau}_{\mathbf{x}}$, and we can get concurrent read access, but for the histogram we are restricted to only one thread per histogram and all memory access is penalized.

While the previous point give us a hint about the superior parallel capabilities of the *ccdf*, the main difference is the actual computation of the median value. For the histogram, we need to iterate over the histogram vector, accumulating its contents until an element makes a condition met. The median value is the index of that element. This procedure is made just by one thread, and the number of steps required for the search depends on the dataset, giving us unbalanced loads on the threads and data dependency.

For the $ccdf$, we compare each element of the $\boldsymbol{\tau}_{\mathbf{x}}$ vector with the median index. This operation can be done in parallel, thus we need only one step to compare all the data. Finally, to get the median value we do a summation of the result of the previous comparisons. In the many concurrent threads context, the summation can be done in $\log_2(b-a)$ steps (assuming $(b-a)$ a power of two). The whole procedure to get the median value from the $\boldsymbol{\tau}_{\mathbf{x}}$ is independent of the data set. From this analysis we can say that the $ccdf$ based median filter is better suited for parallel computing systems than the histogram based ones.

Given the vector $\mathbf{x} = [6, 6, 1, 2, 7]$, with $0 \leq x_i \leq 7$, its histogram $\mathbf{h}$, with eight elements, is calculated by one thread in five steps. Each element of the vector $\boldsymbol{\tau} = \tau_j = ccdf(\mathbf{x}), j \in [0, 7]$ can be computed by one thread (8 threads in total) in 5 steps. To get the median value from $\mathbf{h}$ only one thread can be used, and it requires six steps (each of them include conditional branches and additions). For the $ccdf$ we can use eight threads to apply the indicator function to $\boldsymbol{\tau}$ in one step, and then we can use a binary reduction of three levels to solve the summation. The total steps required to get the median value of $\mathbf{x}$ is eleven with the histogram based algorithm, whilst we need only nine steps with the $ccdf$ based one.

4. PERFORMANCE

The following test were executed on a PC with an Intel Core i7-930 CPU (2.8GHz, 8MB Cache memory, SEE4.2 Instruction set), 4GB DDR3 RAM memory. A nVidia GeForce GTX260 graphics card (1.3 computational capability, 16KB shared memory, 216 CUDA cores, 1.3 GHz GPU clock speed) is connected to a PCI Express 2.0 slot. This is a medium performance graphics card. The operative system is Kubuntu 10.4, with Linux kernel 2.6.28 x86_64.

Each condition tested (kernel size, input image, algorithm used) is repeated 1000 times and the result shown is the median value. The time measures for the CUDA implementations include memory transfer operations. It is important to note that, for a kernel size k, the elements to be sorted are $n = k^2$. For the tests we use $k = \{3, 5, 7, 9, 11, 13, 15\}$.

4.1. Description of Median Filter Implementations

We test six implementations, three for Intel processor and the other three for CUDA graphics cards. For each platform two implementations are reference algorithms and the third one is a PCMF implementation.

For CUDA, the first reference implementation is the Branchless Vector Median (BVM) algorithm [14]. This $O(n^2)$ algorithm is based on sorting networks; and exploits some of its properties to insert and remove elements from a previously sorted dataset. Sorting networks are greatly benefited of the data parallelism (offered by the SIMD units [8, 10]), but the CUDA approach offers a different type of parallelism (Single Instruction Multiple Thread - *SIMT*). The second CUDA reference implementation is a closed-source commercial library for CUDA, named libJacket [15]. This library implements many general algorithms and it include an implementation of the median filter for images. The algorithm it uses is not mentioned on its documentation.

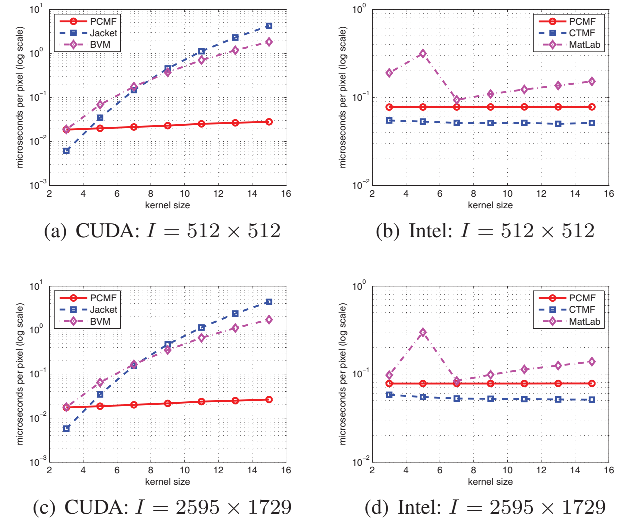
For the Intel processor, the first reference implementation is the Constant Time Median Filter (CTMF) algorithm [6]. It is a histogram based median filter with $O(1)$ computational complexity. It uses the scalar approach to generate all the histograms required, and to get the median value. The accumulated histogram is calculated and updated with vector operations on the SIMD (Single Instruction Multiple Data) unit. The implementation needs the size of the cache memory as input parameter, and it uses this information to limit the quantity of histograms used. The second refence implementation is provided by the MatLab function `medfilt2` [16]. We use MatLab version 7.9.0.529 (R2009b) 64-bit (glnxa64). The algorithm is based on histograms and has $O(n)$ computational complexity.

In the Intel implementation of the PCMF, the SIMD unit is used for almost all the operations, and in the generation of all $\boldsymbol{\tau}$ vectors. This implementation is expected to have a $O(1)$ complexity, as the condition $\frac{O(n^2)}{M} \rightarrow 0$ is held. For the CUDA implementation, the image is split in 32×64 pixels subimages. Each subimages is processed by one CUDA block with 512 threads per block. The structure of the algorithm allow us to keep all threads working most of the time, and help us to avoid the bank conflicts. The dimensions of the subimages are limited by the size of the shared memory per block, and this limit prevents the condition $\frac{O(n^2)}{M} \rightarrow 0$ to be held. For this reason, the CUDA implementation is expected to not have a constant computational complexity.

4.2. Computational Results

In the first test we measure the execution time of the median filter implementations. The size of the test image used are 512×512 (0.26MP), 640×480 (0.3MP), 1280×720 (0.9MP), 1920×1080 (2.1MP), 2592×1729 (4.5MP) and 4000×4000 (16MP). It is important to note that for the image of 4000×4000 and kernels bigger than 5, the CUDA BVM and the CUDA Jacket implementations caused CUDA runtime errors and the test failed.

From the results (Fig 2) we can see that the CUDA reference implementations (Jacket and BVM) has a $O(n^2)$ behavior (in logarithmic scale), whilst our CUDA PCMF behaves almost constant. For the Intel implementations we can confirm the expected computational complexity: $O(n)$ for MatLab (for $k \geq 7$) and $O(1)$ for the CTMF and PCMF algorithms.



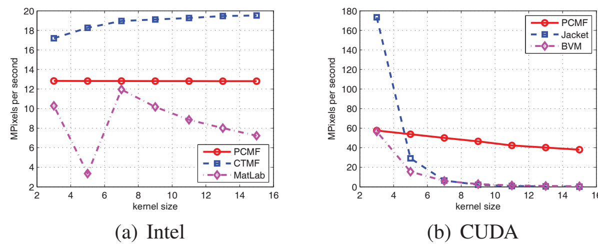
(a) CUDA: $I = 512 \times 512$ (b) Intel: $I = 512 \times 512$

(c) CUDA: $I = 2595 \times 1729$ (d) Intel: $I = 2595 \times 1729$

Fig. 2. Computational results in time (μsec per pixel) for images of 0.26 and 4.5 Megapixels

A better measurement for the performance of the implementation is how many pixels per second they can process. Then we calculate the Megapixels per second (MP/sec) that each implementa-

tion achieves. The results (Fig. 3) show that for Intel, the PCMF process at $12.8 MP/sec$, the Intel CTMF do it in a range of 17.1 to $19.5 MP/sec$ and the MatLab performance drops linearly. For CUDA, the PCMF can process in the range of 35 to $57 MP/sec$, the other two algorithms have a poor performance for $k \geq 7$.



(a) Intel (b) CUDA

Fig. 3. Comparison of computational results in Megapixels per second

Next, the data dependence of the algorithms is tested. We use two images: Boat (512×512) and Goldhill (1600×1200), with different level of salt and pepper noise ($0\%, 25\%$ and 50% of pixels corrupted). This test is made because most of the sorting algorithms are data dependent, and the worst case performance of those algorithm are frequently obtained when the data is corrupted by salt and pepper noise. For this test we use $k = 15$ and the CPU cycles required to process one pixel is measured. The results (Table 1) shows that the MatLab and CTMF implementatios are data dependent.

Table 1. Computational Results (CPU cycles per pixel)

Algorithm	Boat			Goldhill		
	0%	25%	50%	0%	25%	50%
Intel						
MatLab [16]	417.74	513.38	580.25	354.85	492.56	563.95
CTMF [6]	143.03	146.12	146.74	139.56	143.70	145.17
PCMF	218.56	218.57	218.58	219.19	219.29	219.25
CUDA						
Jacket [15]	11.75k	11.80k	11.76k	12.23k	12.22k	12.23k
BVM [14]	5.10k	5.10k	5.10k	4.93k	4.93k	4.93k
PCMF	78.21	78.22	78.25	74.53	74.45	74.44

5. CONCLUSION

In this paper we propose a new algorithm for sorting and a bidimensional median filter based on it, the Parallel Ccdf-based Median Filter (PCMF). The structure of the algorithm allows us to get an efficient and simple implementation for parallel systems. The computational results show that the CUDA implementation of the proposed median filter algorithm is efficient and can outperform other generic median filters for CUDA.

The behavior of the implementation Intel PCMF is clearly $O(1)$ (see Table 1). The reference algorithm, the CTMF, also has $O(1)$ computational complexity and has a better performance than our implementation.The MatLab median filter has linear complexity and it has the worst performance of the Intel implementations tested.

Finally, the CUDA implementation of the PCMF has the best overall performance of the tested algorithms, and has a lower computational complexity than the reference CUDA implementations. $O(n^2)$ algorithms are usually faster than $O(n)$ and $O(1)$ for small n, but our implementation has a better performance than algorithms with that computational complexity (except for the commercial solution Jacket, and only for $k = 3$).

6. REFERENCES

[1] A. Bovik, *Handbook of Image and Video Processing*, Academic Press, 2000.

[2] J. Tukey, "Nonlinear (nonsuperimposable) methods for smoothing data," in *IEEE Electronics and Aerospace Conference (EASCON), Conference Records*, 1974, p. 673.

[3] T. Huang, G. Yang, and G. Tang, "A fast two-dimensional median filter algorithm," *IEEE Trans. Acoust., Speech, Signal Process.*, vol. 27, no. 2, pp. 13–18, 1979.

[4] B. Chaudhuri, "An efficent algoritm for running window pel gray level ranking 2-d images," *Pattern Recognition Lett.*, vol. 11, no. 2, pp. 77–80, 1990.

[5] J. Gil, "Computing 2-d min, median and max filters," *IEEE Trans. Pattern Anal. Mach. Intell.*, vol. 15, no. 5, pp. 504–507, 1993.

[6] S. Perreault and P. Hébert, "Median filter in constant time," *IEEE Trans. on Image Processing*, vol. 16, no. 9, pp. 2389 – 2394, 2007.

[7] D. Cline, K. B. White, and P. K. Egbert, "Fast 8-bit median filtering based con separability," *International Conference on Image Processing*, 2007.

[8] R. Sánchez, "Diseño e implementación del filtro mediano de dos dimensiones para arquitecturas SIMD," *Bacherlor's Degree Thesis*, 2011.

[9] T. Furtak, J. N. Amaral, and R. Niewiadomsk, "Using simd register and instructions to enable instruction-level parallelism in sorting algorithms," *SPAA '07: Proceedings of the nineteenth annual ACM symposium on Parallel algorithms and architectures*, pp. 348–357, 2007.

[10] P. Kolte, R. Smith, and W. Su, "A fast median filter using altivec," *International Conference on Computer Design (ICCD'99)*, pp. 384–391, 1999.

[11] Y. Li, S. Peng, and W. Chu, "An efficient parallel sorting algorithm on metacube multiprocessors," *9th International Conference on Algorithms and Architectures for Parallel Processing*, pp. 372–383, 2009.

[12] S. Chen, J. Qin, Y. Xie, J. Zhao, and P. Heng, "A fast an flexible sorting algorithm with cuda," *9th International Conference on Algorithms and Architectures for Parallel Processing*, pp. 281–290, 2009.

[13] T. Ryan, *Modern Engineering Statistics*, chapter 14, p. 468, Wiley-Interscience, 2007.

[14] W. Chen, M. Beister, Y. Kyriakou, and M. Kachelries, "High performance median filtering using commodity graphics hardware," in *IEEE Nuclear Science Symposium Conference Record (NSS/MIC)*, 2009.

[15] AccelerEyes, "Jacket library," *http://www.accelereyes.com/products/libjacket*, 2011.

[16] MatLab, "medfilt2," *http://www.mathworks.com/help/toolbox/images/ref/medfilt2.html*, 2011.