Hierarchical Histogram-based Median Filter for GPUs

Péter Szántó, Béla Fehér

Budapest University of Technology and Economics, Department of Measurement and Information Systems, Magyar tudósok krt. 2, 1117 Budapest, Hungary, {szanto, feher}@mit.bme.hu

Abstract: Median filtering is a widely used non-linear noise-filtering algorithm, which can efficiently remove salt and pepper noise while it preserves the edges of the objects. Unlike linear filters, which use multiply-and-accumulate operation, median filter sorts the input elements and selects the median of them. This makes it computationally more intensive and less straightforward to implement. This paper describes several algorithms which could be used on parallel architectures and propose a histogram based algorithm which can be efficiently executed on GPUs, resulting in the fastest known algorithm for medium sized filter windows. The paper also presents an optimized sorting network based implementation, which outperforms previous solutions for smaller filter window sizes.

Keywords: median; filter; GPGPU; CUDA; SIMD

1 Introduction

Median filtering is a non-linear filtering technique primarily used to remove salt-and-pepper noise from images. Compared to convolution-based filters, median filter preserves hard edges much better, therefore being a very effective noise removal filter used before edge detection or object recognition. For example, it is widely used in medical image processing to filter CT, MRI and PET images; in image capturing pipelines to remove the sensors' digital noise; or in biological image processing pipelines [14] [15] [17].

Median filter works basically replaces the input pixel with the median of the N=k*k surrounding pixels (Figure 1), which can be done by sorting these pixels and selecting the middle one. The figure also shows one of the possibilities for optimization: as the filter window slides with one pixel, only k pixels change in the k*k array.

The generalization of the median filter is the rank order filter [3], where the output is not the middle value, but the nth sample from the sorted list. Rank order filter

can be even more efficient in noise removal, also used in medical image processing [17]. Due to space limitation, rank order filtering is not discussed.

The main drawback of the median filter is the computational complexity. Linear filtering is based on sequential data access and multiply-and-accumulate operation that can be efficiently executed using CPUs, GPUs or FPGAs. On the other hand, implementing median filtering without data dependent operations is considerably less straightforward and most architectures are not tailored towards the efficient implementation of it.

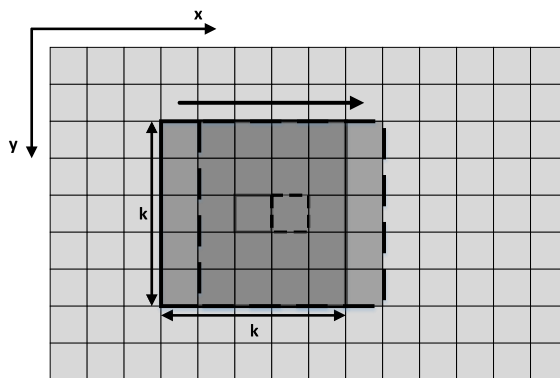

Figure 1
2D median filtering: computation of two consecutive pixels

Median filter implementations differ from each other by the applied sorting (or selection) algorithm. For small window sizes, $O(N^2)$ algorithms are good candidates, for larger window sizes O(N) solutions are typically faster. Several research papers also presented O(1) algorithms [5] [7], but these are typically highly sequential and only offer lower execution time than O(N) algorithms if the window size is quite large (~25x25 pixels). We can conclude that there is no "best algorithm", the optimal solution depends on the window size.

This paper presents several algorithms, which can be efficiently used to execute low- and medium-sized median filtering on highly parallel architectures, like GPUs.

2 Architectures

Although there are many different computing architectures, the key in order to achieve high performance is parallelization. FPGAs offer fine-grained parallelism, where the computing elements and memory structures can be intensely tailored to the given application. Multi-core CPUs and many-core architectures, like GPUs,

share the basic idea of sequential instruction execution, but the level of parallelization is vastly different.

2.1 Parallel Execution on CPUs

Irrespectively of being embedded, desktop, or server products, CPUs offer two kinds of parallelization: the multi-threading and the vectorization. To be able to exploit the multi-core nature of processors, the algorithm should be able to run on multiple, preferably highly independent threads. For image processing, especially in case the resolution is high or there are lots of images in sequence, this is not an issue, as the number of cores in a single CPU is typically low (~32 for the largest server products). It is possible to separate input data temporally (by processing each image of the input sequence on different core) or within frames (by processing smaller parts of a single input image on different cores). This kind of multi-threading requires a minimum amount of communication between threads, and uniform memory access model – used by single multi-core CPUs – makes the data handling straightforward. Larger scale homogeneous CPU systems (such as High-Performance Computing centers) are quite different and are out of the scope of this paper.

The other parallelization method offered by CPUs is SIMD execution. Each modern CPU instruction set has its own SIMD extension: NEON for ARM [13], SSE/AVX for Intel [12] and AltiVec for IBM [9]. An important common feature of all SIMD instruction sets is the limited data loading/storing: one vector can be only loaded with elements from consecutive memory addresses and the elements of a vector can be written only to consecutive memory addresses. Beyond this limitation, in order to achieve the best performance, it is also beneficial to have proper address alignment. Regarding execution, obviously, it is not possible to have data dependent branches within one vector. The typical vector size is 128 or 256 bit and vector lanes can be 8, 16 or 32-bit integers or floats, so when processing 8-bit images 16 or 32 parallel computations can be done with one instruction. The nature of SIMD execution greatly reduces the type of algorithms which can be parallelized this way – with the notable exception of [5], most of them are based on the minimum and maximum instructions [8] [9].

Although the future of this product line seems to be questionable now, we should also mention Intel's many-core design: the Xeon Phi. In many ways, it is similar to standard Intel processors, as it contains x86 cores with hyper-threading and vector processing capabilities. The main differences are the number of cores in a single chip (up to 72); the wider vectors (512-bit); the mesh interconnect between the cores; and the presence of the on-package high-bandwidth MCDRAM memory.

2.2 Parallel Execution on GPUs

The computing model of GPUs is quite different from CPUs. First, the number of cores is much larger – e.g. 2560 in the high-end NVIDIA GTX 1080 [11]. To allow the efficient usage of these cores, the number of concurrently executed threads should be much larger (tens of thousands) than it is for a CPU. Execution is also different: basically, a GPU is a MIMD array of SIMD units.

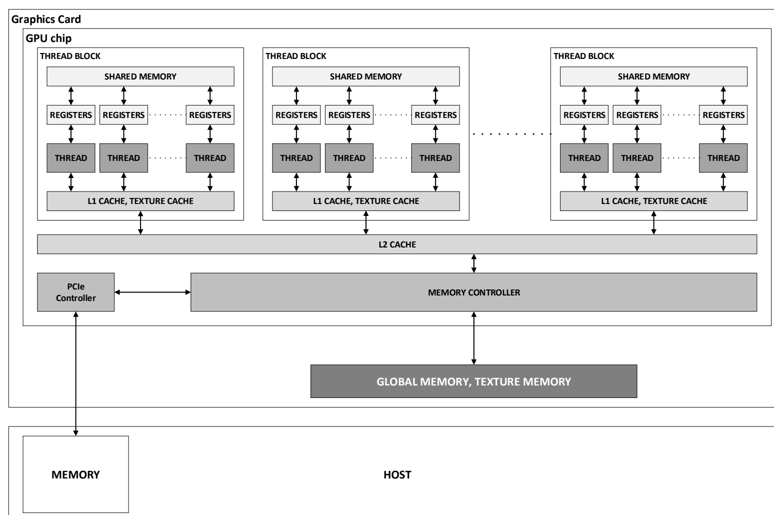

Figure 2

GPU thread hierarchy and accessible memories

Threads are grouped hierarchically, the lowest level being the warp (NVIDIA) or wavefront (AMD), which contains 32 or 64 threads, respectively. The threads within a warp always execute exactly the same instruction; therefore, they are SIMD in this respect. If threads within a warp execute different branches, all threads within the warp execute both branches – therefore data dependent branch divergence should be avoided within a warp unless one of the branches is empty. Memory access, however, is much less constrained compared to a CPU's SIMD unit: theoretically, there is no limitation on where the warp's threads read from or where they write to; practically to get good performance there are constraints, but it is still much freer than it is on CPUs. All threads have their own register space, which is the fastest on-chip storage that the GPU has. The number of registers allocated to a single thread is defined during compile time: the hardware maximum is 255, which is much more than the number of registers available in a CPU (though the number of concurrently scheduled threads decreases as the number of registers per thread increases, which may affect the performance).

The next hierarchy level is the Thread Block (TB): threads within a TB can exchange data through a local (on-chip) Shared Memory (SM) and they can be synchronized almost without overhead. Hardware-wise, threads within a TB are executed on the same Streaming Multiprocessor (SMP). If resource constraints – SM and register usage per TB – allow it, threads from multiple, independent TBs are scheduled on a single SMP.

TBs are grouped into Thread Grid, which is executed independently of each other. Data exchange between TBs is only possible through cached, off-chip Global Memory (GM) and synchronization is quite expensive in terms of performance.

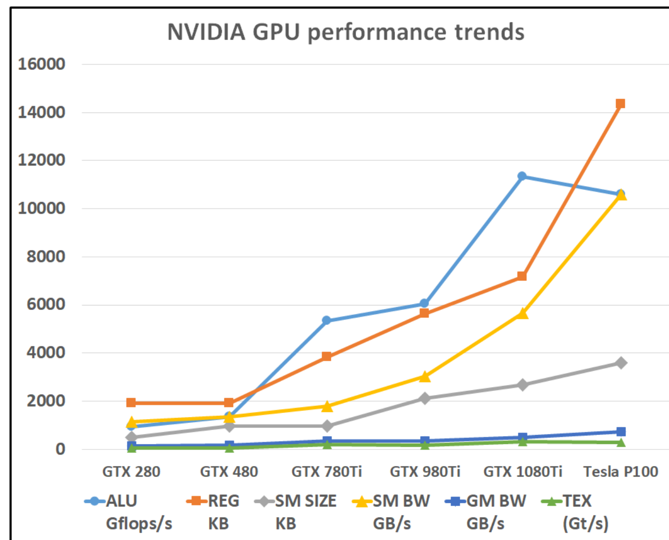

Figure 3
GPU performance trends

Beyond the computational capacity, another important factor that has a huge impact on performance is memory bandwidth. Just as in case of CPUs, the fastest storage is the register array. The second fastest memory is the on-chip SM, but in order to get the most performance out of it, there are restrictions to keep. SM is divided into 32 parallel accessible banks; each consecutive 32-bit word belongs to a different bank. As long as threads within a warp access different banks (or multiple threads access exactly the same word within a bank), SM access is parallel. However, if different threads within a warp access different words from the same bank, memory access is serialized, which heavily affects performance. GM bandwidth is considerably lower (see Figure 3). Since the Fermi architecture, access to off-chip memory is cached, thus performance impact of non-ideal transactions is considerably decreased, but it is still important to design for ideal read/write bursts.

Figure 3 shows the trends of the theoretical calculated performance of current and past NVIDIA GPUs. ALU performance is the number of floating point operations per second (other instructions may execute much slower, but it depends on GPU architecture); REG is the size of the register space on the whole chip; SM size is the size of the SM on the whole chip; SM BW is the cumulative bandwidth of all SM blocks; GM BW is the bandwidth of the off-chip GM; TEX is the texture read performance in GTexel/s.

Firstly, we can conclude: the increase in GM bandwidth is notably slower than the increase in computational performance (even for the P100, which uses HBM2 High Bandwidth Memory). Therefore, often used data should fit into on-chip memory otherwise the algorithm is not the best candidate for GPU acceleration. Second, SM bandwidth increases more or less in line with the computing performance, but this is not true for the size of this memory. Thus, memory bandwidth intensive algorithms may be relatively good candidates, but the algorithms requiring a large size of on-chip memory are not ideal.

2.3 Special Instructions

Creating branchless algorithms is a requirement for vectorization and it is also preferred for GPU acceleration. For median filtering, the most important instructions are the vector comparison instructions, the data selection instruction, and the minimum/maximum instructions.

All SIMD CPU and GPU architectures have comparison instructions for all data types: these instructions set a register either to a predefined value (if the comparison is true) or to zero. E.g. for Intel SSE the result is 1.0f and 0.0f for floating point data types and -1 and 0 for integer data types. For ARM NEON, the output is -1 or 0, irrespective of the input data type. NVIDIA GPUs has a more versatile SEL instruction: the result type can be set independently of the source operand type, therefore both (1.0f or 0.0f) and (-1 or 0) are supported. A slightly different version of the SEL instruction is SELP: instead of setting a general-purpose register, this instruction writes a special predicate register. Please note that comparison instructions are not always executed at full-speed: unlike the Fermi generation, in case of newer NVIDIA Maxwell and Pascal GPUs, the throughput is 0.5 operations per clock per ALU. The same is true for x86 SSE2, throughput is 0.5 or 1 depending on data type and CPU generation.

Another important instruction type is the selection. ARM NEON and IBM AltiVec natively supports 3-operand selection instruction. NVIDIA GPUs has a SELP instruction where the selection is based on the value of a predicate register and an SLCT instruction where selection between two operands is based on the sign of the third operand. SSE2 does not have dedicated selection instruction, but it can be implemented with several logical operations.

Minimum and maximum instructions are widely supported, all SIMD CPU instruction sets and all GPUs can execute them on all types of data, though – just like comparison and selection – throughput is typically lower than 1.

3 Grayscale Image Processing

In this chapter, we present several algorithms. Binary search (BS) is an optimized CUDA implementation of NVIDIA's OpenCL SDK sample [10]; Batcher's Odd-Even Merge Sort (BMS) implements the known algorithm [1] in CUDA; BMS2 is an optimized version of BMS; the counter based method (CNT) is the GPU implementation of our previous FPGA-based solution [3]; and finally hierarchical histogram (HH) is a novel histogram based solution for GPUs. Exactly the same implementations can be used to handle multiple color planes independently, e.g. to filter R, G and B channels independently. For biomedical image processing pipelines where the planes show different properties of the cell (e.g. fluorescence microscopy [16]) this is the required processing method. However, for natural color images, like photos, this is not an advisable method, as independent filtering may generate RGB combinations on the output, which were not present on the input. Chapter 4 presents a luminance based modification which can be used to filter photographs.

3.1 General Architecture

Similarly to most image processing functions, input data usage is very redundant in case of the median filter. For a N=k*k window size, N-k pixels are the same for two neighboring output pixel calculations (see Figure 1), which means that every input pixel is read several times. Therefore, appropriate caching is very important. Modern GPUs offer three alternatives: cached read access from GM; cached read from Texture Memory; software managed buffering in SM. Although every memory access is cached, efficiency is not the same – albeit it requires more instructions, typically an SM-based solution is considerably faster, thus this is the method used for the implementations discussed below.

Unless otherwise noted, the basic principle is that every thread computes one output pixel, which means that a thread block with a size of TX*TY requires (TX+k-1)*(TY+k-1) input pixels. The 8-bit pixels are stored line by line in SM, so consecutive byte addresses belong to adjacent pixels, therefore one SM word stores four adjacent pixels. Using horizontal thread block size of 32, access to the adjacent pixels within a warp is free of bank conflict: at any given time the 32 threads of a warp access 32 adjacent pixels which reside in 32/4=8 banks. If a single thread computes S horizontally adjacent output pixels, the ith thread of a warp reads byte address i*S, which is bank i*S/4. To have a bank conflict-free

access, (i*S/4)%32 should be different for i=0…31, which limits the possible numbers to O={1,2,3,4,12,…}. Computing vertically adjacent pixels is free of bank conflict: at any given time the threads within a warp reads horizontally consecutive pixels from a single line. The drawback is GM read access: when loading input data into SM only 32+k-1 bytes can be read from consecutive addresses.

3.2 Binary Search

Binary search (BS) is the algorithm employed by the 3x3 median filter example in NVIDIA's OpenCL SDK, therefore it is included as a reference [10]. It finds the median by counting the number of elements which are greater than the current median candidate (highcount); if the result is greater than (N-1)/2, there are more larger elements and less smaller elements than the median candidate, thus the candidate is increased with interval halving. Figure 4 shows the first three steps of the algorithm: the initial median candidate (128) is half of the absolute interval ([0….255]). In the first step, there are 5 larger and 4 smaller elements than the median estimate. Therefore, the median estimate is moved to the middle of the upper half-interval: (128+255)/2=192. The new interval is [128…255]. The maximum number of steps required to find the median is $b=\log_2 I$, where I is the range of the input data, e.g. 256 for 8-bit input.

A single step of an iteration is a comparison of the input data and the modification of the high-counter. This should be repeated for every input pixel, thus the complexity is O(b*N), where b is the bit width of the input values. The algorithm is a good choice because it does not contain divergent branches and does not require an extensive amount of registers or memory (interval halving can be implemented branchless with comparison and selection instructions).

NVIDIA's original implementation was created for RGBA input images where every thread computes one pixel. The input components were stored in SM as 8-bit values per component, but computation (median estimate, interval ends) was done using float values. This is acceptable when the filter window is small because the compiled code reads input values from the SM only once and stores them in registers – conversion from uint8 to float takes place during the read. However, as the window size increases, there are not enough registers to store all N input pixels in registers, therefore every pixel is read and converted multiple times, which makes type conversion redundant (not to mention that this type of instruction is slow anyway). Although it requires more memory, performance wise it is a better approach to do the conversion when the SM is loaded – the speed of this process is limited by the GM bandwidth, so the relatively slow type conversion is almost free in terms of execution time.

By checking the generated assembly code, it can be noticed that the NVIDIA compiler generates a SETP and a SELP instruction from the C code which

implements the basic compare-increment functionality. This is unnecessary, as the SEL instruction can set a register to 1.0f or 0.0f based on the result of the comparison, thus one instruction can be saved. With the data type modification and inline assembly, the performance of the binary search method can be increased by 60% compared to the original code.

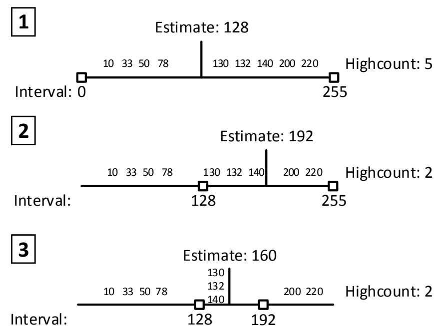

Figure 4
First three steps of the binary search

As the algorithm can be realized without branches, it can be vectorized when being implemented for CPUs. When SSE2 is used, a single vector register can be used to calculate the counter for 16 adjacent pixels. As the comparison returns -1 when true, the addition should be replaced with subtraction for the "larger than" counter and the selection operation should be implemented with logical operators.

3.3 Batcher Odd-Even Merge Sort

The Batcher Odd-Even Merge Sort (BMS) is a general sorting network introduced by Ken Batcher [1]. Although its $O(N*(\log N)^2)$ complexity is not optimal, for reasonable input size it is better than any other sorting network [2]. Like other sorting networks it is based on comparison and element swapping, which is equivalent to executing min()/max() functions – therefore it can be implemented branchless. The algorithm completely sorts the N input elements, which is not necessary in the case of a median filter, so complexity can be slightly decreased by removing the unnecessary comparisons. Figure 4b shows the network for N=9. Generally, for $N=2^t$ inputs, the number of data swaps required to generate a sorted list [2]:

$$c(2^t) = (t^2 - t + 4) * 2^{t-2} - 1 \quad (1)$$

Therefore, complexity is $O(N*(\log N)^2)$. Because of the relatively bad scaling and high register usage, the expectation is that a sorting network-based implementation can only be used for smaller (3x3 or 5x5) window sizes.

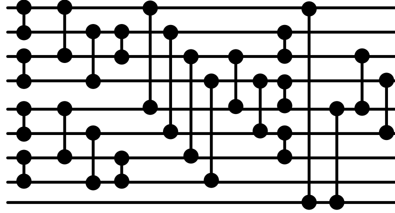

Figure 5
Batcher sorting network for N=9 median

The performance can be considerably increased by taking advantage of the common pixels in two adjacent filter windows: in a window of size N=k*k, there are CP=k*(k-1) common pixels. In the sorted list of the common pixels, the only median candidates are the (k+1) middle values. To compute the final median these candidates should be merged with the sorted list of the remaining k pixels, separately for each window. Therefore, to compute two outputs, we have to employ one CP-input sorting network and two additional merging steps using the (k+1) and k element sorted lists. Compared to performing two N-input sorting networks independently, this method greatly decreases the number of required instructions. Performance numbers achieved with this optimized version are denoted as BMS2 in Chapter 5.

3.5 Counter-based Method

The third algorithm (denoted with CNT) originates from our optimized FPGA implementation [3], but a slightly similar (less efficient) method was also used in [6]. The basic idea is to count the number of elements, which are greater than the current one for every pixel. At the end of the process, there will be N different counter values ranging from 0 to N-1; the median is the element which counter equals to (N-1)/2.

For the ith processed element, the initial value of its counter is set to i. Then the new element is compared with all the older elements (new<old); if the comparison is true, the counter of the older element is increased with one and the counter of the new element is decreased with one. If the comparison result is false, counters do not change. Figure 6 shows the steps of the algorithm for 5 elements.

The median element can be found by comparing all counter values with (N-1)/2 – the index of the median element is the index of the counter where the comparison is true. The already mentioned SETP and SELP instructions can be used to step through all the counter values and select the one with the appropriate value. As the GPU's registers cannot be indexed with a register, this should be done in an unrolled loop. A slightly faster version can be created using SEL and FMAD

(floating point multiply and add): the result of the SEL (1.0f or 0.0f) is multiplied with the index of the counter register and added to an accumulator. As all counter values are different, SEL returns 1.0f only once, thus the final accumulator value equals to the index. This is faster because the throughput of the FMAD instruction is 1.

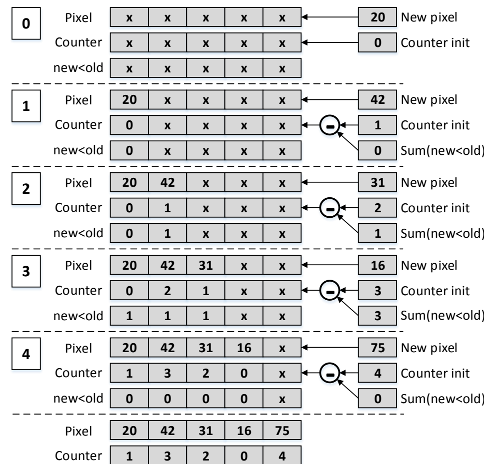

Figure 6

Counter based algorithm example

The basic step of the algorithm contains four arithmetic instructions: set initial counter value; compare the new value with the old one; increment and decrement the corresponding counters. The number of steps required for the full process equals to

$$c = \sum_{i=1}^{N-1} \sum_{j=0}^{i-1} 4 * j \qquad (2)$$

so the algorithm is $O(N^2)$, meaning it is only a candidate for a window size of 3x3 pixels. Register usage also confirms this assumption: the number of counter registers required equals to the window size.

It is possible to exploit the benefits of the overlapping kernel windows: for two adjacent outputs, there are N-k common input pixels and k different pixels. This means that the first N-k computations are common. That is, the number of steps per output pixel can be decreased from 36 to 34 for the 3x3 window, and from 300 to 205 for the 5x5 window. On the other hand, register usage cannot be decreased, as all processed pixel should have their own counter.

3.6 Histogram-based Method

Histogram-based median filters are widely used in CPU implementations because of the achievable O(N) or O($N^{1/2}$) complexity. Moreover, several papers discussed O(1) implementations [5] [7], but such algorithms are typically sequential, require a large amount of memory and, although O(1) complexity is true, the large multiplication factor makes them a viable option only for extremely large window sizes. A conceptually similar method was also implemented for GPUs [5] with the conclusion that although O(1) complexity can be approached, the O(N) version is faster for reasonably sized windows.

Creating a simple histogram based median filter for uint8 data (referred as H256) is quite straightforward. After clearing the histogram, each pixel in the filter window is processed and the histogram bin corresponding to the given pixel's value is incremented. After all pixels are processed, the summation of the histogram bins equals to N. The median value can be found by accumulating histogram bin values starting with bin 0 – the median value is in the bin where the accumulator reaches (N+1)/2, because there are (N+1)/2-1 pixels which are smaller than the current bin index. Exploiting the advantages of a sliding window (see Figure 1) is also quite straightforward: as the filter window steps one column right, k pixels should be removed and k new pixels should be added to the histogram. For GPU implementation, the main drawback of the simple histogram algorithm is memory usage. Every thread should have its own histogram of the pixel(s) it processes. This limits the number of concurrently scheduled threads on an SMP to 384 (Maxwell and Pascal architectures) when using uint8 bins and to 192 when using uint16 bins (for windows which contain more than 255 pixels). Such a low level of occupancy can seriously decrease performance.

To reduce memory usage at the expense of increasing the instruction count, we propose a novel method, the Hierarchical Histogram (HH). The basic idea is to first create a histogram based on the MSB bits of the input and then create a histogram based on the LSB bits using the inputs whose MSB equals to the computed MSB median. Figure 7 demonstrates the operation using the following hexadecimal input values: 0x14, 0x44, 0x42, 0xA3, 0xA6, 0xAB, 0xC0, 0xE4, 0xFF. The MSB histogram is created from the 4 MSB bits of the input data set, then the median is found by accumulating the bin values until (N+1)/2 is reached – in this example this is bin 10, where the number of already counted elements equals to 6. The number of elements before the selected bin is 3 (bin 1 and bin 4). The LSB histogram is created using input data where MSB equals to 0xA. To find the LSB median, bin value accumulation should be started from 3, what was the number of elements before bin 10 in the MSB histogram. The process reaches (N+1)/2 at bin 6, therefore the median LSB value is 6, thus the full median value is 0xA6. For 8 bit inputs separating MSB and LSB processing requires only 16 bytes of histogram memory, which greatly increases GPU occupancy.

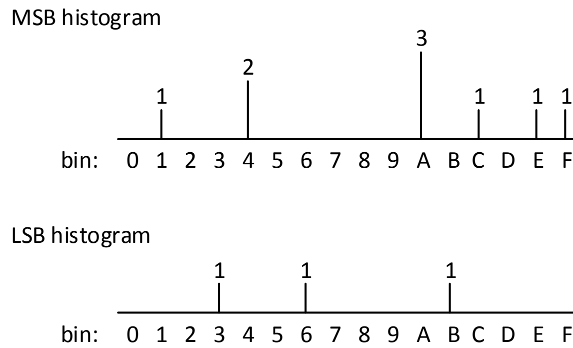

Figure 7
Median filtering using Hierarchical Histogram

The main disadvantage is that in worst case every input pixel should be processed twice, which almost doubles SM bandwidth usage. The efficiency increase from incrementally processing adjacent pixels also decreases: although the MSB histogram can be updated by removing k old pixels and adding k new ones, the LSB histogram has to be cleared and recomputed for every new output. To be able to do this, the two histograms have to be stored in separate memories, which doubles the required space.

	Word address	Thread, Bin			
Bank 0, Word 0	0	0,0	32,0	64,0	96,0
Bank 1, Word 0	1	1,0	33,0	65,0	97,0
....	....	....	....	....	....
Bank 31, Word 0	31	31,0	63,0	95,0	127,0
Bank 0, Word 1	32	0,1	32,1	64,1	96,1
Bank 1, Word 1	33	1,1	33,1	65,1	97,1
....	....	....	....	....	....
Bank 31, Word 1	63	31,1	63,1	95,1	127,1
....	....	....	....	....	....
....	....	....	....	....	....
Bank 0, Word 15	480	0,15	32,15	64,15	96,15
....	....	....	....	....	....
Bank 31, Word 15	511	31,15	63,15	95,15	127,15

Figure 8
Histogram Shared Memory structure

For all histogram based filters bank conflict-free memory access is also an important design goal. As the threads within a warp access their histogram randomly, the histogram array should be constructed in a way that different bins of a histogram reside in the same bank and only bins from different warps may be placed in the same bank. For N<256 uint8 is enough to store a single bin, for larger N uint16 is necessary. Assuming uint8 bins, the most trivial way to avoid bank conflict is to place the first 4 bins of a thread into word W of bank B, and the ith 4 bin word into word W+i of bank B. The drawback of this method is that

histogram access requires additional bit selection from the input to determine word address and byte address. A better way is to make addressing data-independent and only related to the thread ID so it can be computed once. One general solution is to place 4 bins from 4 different warps into the same word and place the bins of a thread into the same bank, as Figure 8 shows. For window size N>256 only two uint16 bins are placed in a single word, but otherwise, the same memory structure can be used.

When using histogram-based median filters on CPUs, memory size is not the limiting factor – a 256-byte histogram easily fits into any CPU's L1 cache. On the other hand, searching the histogram may take up to 256 steps in the worst case. For reasonable window sizes (<15*15), this can require much more instructions than the histogram update itself, therefore accelerating the search gives a tremendous performance increase. Several SIMD instruction sets offer horizontal addition, which sums the elements of a single vector, e.g. 16 bins when the vector size is 128 bit. The downside is that the resolution of the resulting "sum of bins" is not one bin, but 16. Therefore, after the accumulated sum of 16 bins is larger or equal to (N+1)/2, single bin values should be subtracted until the accumulator becomes smaller than (N+1)/2. The bin subtracted, before this condition becomes true, is the median bin. In the worst case, the horizontally vectorized search requires (256/16) vector adds and 16 single bin subtractions, which is 8 times faster than the linear search. For SSE2, horizontal operations are very limited, only the sum-of-absolute–differences can be used to sum 8-8 elements of a 16 element vector. Most other instruction sets (SSE4, AVX, NEON) offer a wider range of horizontal add instructions.

4 Luminance-based Filtering

As it was mentioned earlier, to avoid mixing color planes from different input pixels, color images should be filtered using the luminance component (or a value similar to it). The additional work to be done compared to the grayscale version are the computation of luminance from RGB, and the selection of the full RGB values based on the luminance median value. The former is quite trivial: the input SM is filled with luminance values computed from the RGB components during the SM write. Storing only this value in SM is an appropriate compromise: the full RGB values are used only for computing the luminance value and during the final output RGB write. This degree of redundant read does not justify the memory size which would be required to store the luminance and all the RGB planes in internal memory. Even if only the luminance value is stored, its accuracy has an effect on the memory size required: storing only 8 bits requires similar memory size to the grayscale version, however using more bits doubles the input storage space required. Furthermore, in the case of the histogram-based algorithms, increasing

the accuracy of the luminance value exponentially increases the size of the required histogram memory.

Selecting the full RGB values from the computed luminance median is quite different for the algorithms discussed above.

The Counter based method does not require too much additional computation – searching is similar to the grayscale version with the only difference being that the output of the process is not the luminance value, but the linear index of the register (and thus the pixel) where the median value resides. This index – together with the thread IDs – clearly defines the input pixel, which should be written as an output.

Batcher Odd-Even Merge Sort (and any other algorithm based on min/max computation) can be easily modified by extending the values with the pixel indices. That is, when the luminance value is loaded from SM, a 32-bit value is generated by concatenating the luminance with the x and y pixel coordinates within the filter window. As the luminance is placed on the MSB, adding the indices does not affect the min/max computation. Final RGB values can be loaded based on the indices in the median value and the thread ID.

For the histogram based algorithms, there are two methods implemented. The first one simply compares all the luminance values of the filter window with the resulting median and stores the index of the pixel where the values are similar. This can be done branchless with SETP and SELP instructions. The other method requires an additional index memory. This index memory has as many bins as the LSB histogram and it is written with the pixel index whenever the LSB histogram is updated. When the LSB histogram update is finished, it is certain that the bin corresponding to the median value contains the index of the last pixel equals to the median. As this memory is never cleared, other bins of this index memory may contain outdated data (if multiple pixels are processed by a single thread), but this does not have any influence on the operation.

For the binary search algorithm, the only available option is the complete search method (the first alternative for the histogram based implementation).

5 Performance Results

For the benchmarks below a real-world 20 MPixel (4992x3774) photo was used – the histogram of the grayscale version is shown in Figure 9. With the exception of the histogram based methods, all other algorithms are data independent, thus execution time does not depend on the input data. For the HH version, the worst-case execution time can be measured when every pixel falls into the last bin. In this case, all pixels have to be processed both for the MS and the LS histograms

and the median search takes as many steps as many bins are used. For the vectorized H256 version the worst case is when all pixels are in bin 240 – this is the first bin of the last 16-bin part, so the median search is the slowest in this case. Measurements show that worst-case execution time is ~25% higher than the execution time presented.

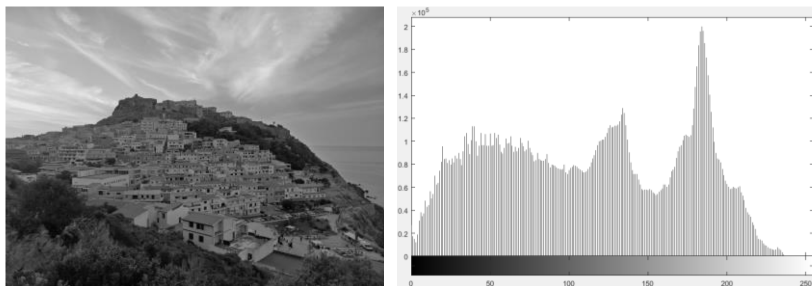

Figure 9
Image and its histogram used for the benchmarks

5.1 GPU Grayscale Results

To our knowledge, the fastest median algorithms for GPUs published to date are the Forgetful selection [4] and Parallel Ccdf-based Median Filter [5] (hereinafter FFUL and PCMF). Although our primary targets are modern desktop and embedded architectures (NVIDIA Maxwell, Pascal), to be comparable with previous work, the grayscale we also benchmarked on a Fermi GPU based Tesla 2070, which is the same card used in the above articles. In their article, the authors benchmarked PCMF for window sizes ranging from 3x3 to 15x15, while FFUL was benchmarked for 3x3, 5x5 and 7x7.

The performance numbers presented in Figure 10 differs from the numbers in the original articles. The mentioned articles measured the time of (host→GPU DMA + kernel runtime + GPU→host DMA) and calculated the MPixel/s numbers from this value, while the results in Figure 10 are calculated from the kernel runtime only. There are multiple reasons to do this. First of all, DMA speed is also affected by the capabilities of the PCIe root complex (chipset). Second, for several GPUs, DMA from/to the host can be overlapped with kernel execution, so as long as data movement takes less time than kernel runtime, it can be eliminated from the full runtime. Third, in the embedded Tegra SoC systems there is no separated GPU memory, therefore there is no need for data copy. For FFUL, kernel runtimes were presented in [4], for PCMF they were calculated by subtracting the two direction DMA times (which were measured with the same board and image size) from the runtimes published in [5].

Using the Tesla 2070, at 3x3 and 5x5 window sizes FFUL is slightly faster than the standard BMS implementation, but the optimized BMS2 has a substantial performance advantage over both. At sizes 7x7 and 9x9 BMS2 is still the fastest, though HH12 (Hierarchical Histogram, one thread computes 12 output pixels) is getting closer because of its lower complexity. The situation changes drastically when the number of registers required by the BMS2 increases too much – performance drops significantly and HH12 becomes the fastest solution for the larger windows. Although it scales worse than PCMF, it is still almost three times as fast even when the window size is increased to 15x15 pixels.

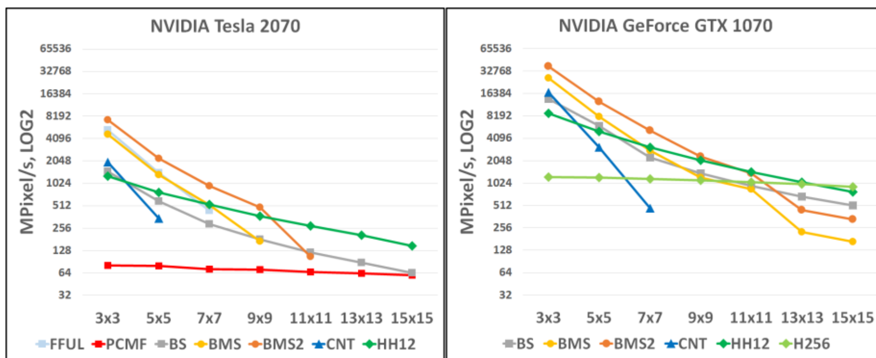

Figure 10

Performance using Tesla 2070 and GeForce GTX 1070

There are notable changes when we use the GTX 1070 with the most recent Pascal architecture (note: FFUL and PCMF were not re-implemented). Due to the increased register count, the larger performance drop of BMS2 happens only at larger window size. On the other hand, SM intensive algorithms also behave better, as the per-SMP SM size is increased from 48 kB to 96 kB, which allows higher occupancy for these implementations. Similarly to the case of the Tesla, the optimized BMS2 implementation is the fastest implementation up to window size 7x7. HH12 already performs on par with it at 9x9, however, the result is data dependent: HH12 can be up to 18% faster than BMS2, or can be up to 17% slower. For larger sizes, HH12 is data-independently faster. Due to the increased SM size, the trivial histogram based implementation (H256) also becomes a viable option – for the largest window size benchmarked, it is even faster than HH12.

Results differ noticeably when the algorithms are executed on NVIDIA's latest embedded GPU which can be found in the Tegra Parker SoC. Similarly to the GTX 1070, this chip is also based on the Pascal GPU architecture, but the SM size was decreased to 64 kB, while the per-SM register size remained the same. The impact is clearly visible in Figure 11: compared to the other solutions, the SM intensive HH4 (Hierarchical Histogram, one thread computes 4 output pixels) algorithm performs slightly worse. BMS2 is clearly the fastest solution up to 9x9

window size and it performs similarly to HH4 at 11x11. As the window size increases further, HH4 becomes the fastest method.

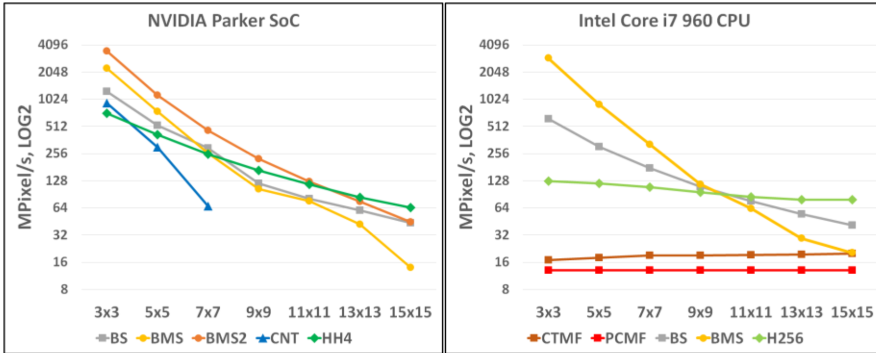

Figure 11
Tegra Parker and Intel Core i7 results

5.2 CPU Grayscale Results

For a complete picture, it is unavoidable to compare GPU filtering performance with CPU filtering performance. In [5] PCMF was also implemented for CPUs and compared to a histogram based O(1) algorithm (Constant Time Median Filtering - CTMF [7]), so these results are included as a reference. As these algorithms are more suitable for very large window sizes, to have comparable results, we have also implemented a multithreaded and vectorized version of BS, BMS, and H256 which takes advantage of Intel's SSE2 instruction set. The benchmarks were run on an Intel Core i7 960 CPU (4 cores, 8 threads, 3.2 GHz frequency, 17 GB/s memory bandwidth), which is similar to the CPUs used in [5], so the results are comparable.

As Figure 11. shows, the trends are quite similar to the GPU results. For window sizes below 11x11, the sorting network based BMS offers the best performance, while for larger windows H256 has the best throughput – at 15x15 it is 4 times as fast as CTMF and 6 times as fast as PCMF.

Conclusions

To give a comprehensive overview about GPU accelerated median filtering, we implemented several algorithms in CUDA and compared them to the fastest published solutions (FFUL, PCMF). BS and BMS are known branchless algorithms, which are good candidates for GPU implementation; CNT is a software implementation of our parallel method developed for FPGAs; BMS2 is our optimized version of the sorting network, while Hierarchical Histogram is our novel algorithm for GPUs. Based on the performance measurements done, we can

make the conclusion: the implementations we presented in the article are the fastest median filter solutions for GPUs. BMS2 offers the highest performance for smaller window sizes; it is faster than the FFUL [4] method even for a window size of 3x3. HH is suitable for larger windows, where it substantially outperforms previously published implementations.

To be able to compare the GPU performance to CPU performance, we also created optimized CPU-based solutions. We can conclude that the easily vectorizable sorting network-based method (BMS) is the fastest implementation up to 9x9 window size, but beyond that our partially vectorized histogram based algorithm becomes faster – at 15x15 window size it is 4 times as fast as the fastest O(1) algorithms published [5] [7].

We should also emphasize the benefits of GPU acceleration in this application. Compared to the CPU, not only the absolute performance is much higher, but also the energy efficiency is outstanding. Even if considering the actual CPU generation approximately doubles the performance per watt ratio, the GTX 1070 board is more than 5 times as efficient as a CPU. The situation is similar in case of the Tegra Parker. Whilst its performance is comparable to the Core i7 960 CPU, there is a huge difference in power consumption: the TDP of the CPU is 130 W, whereas the Tegra consumes less than 10 W.

References

[1] K. Batcher: Sorting networks and their applications, AFIPS Spring Joint Computer Conference 32, 1968, pp 307-314

[2] D. E. Knuth. The Art of Computer Programming, Volume 3: Sorting and Searching, Third Edition. Addison-Wesley, 1998. ISBN 0-201-89685-0. Section 5.3.4: Networks for Sorting, pp. 219-247

[3] P. Szántó, G. Szedő, B. Fehér, Wilson Chung: Scalable architecture for rank order filtering, US Patent 8005881

[4] G. Perrot, S. Domas, R. Couturier: Fine-tuned high-speed implementation of a GPU-based median filter, Journal of Signal Processing Systems, June 2014, Volume 75, Issue 3, pp. 185-190 [FFUL]

[5] R. M. Sánchez, P. A. Rodríguez: Bidimensional median filter for parallel computing architectures, 2012 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)

[6] P. S. Battiato: High Performance Median Filtering Algorithm Based on NVIDIA GPU Computing, International Symposium for Young Scientists in Technology, Engineering and Mathematics, 2016, pp. 1-10

[7] S. Perreault, P. Hebert: Median Filtering in Constant Time, IEEE Transactions on Image Processing (Vol. 16, Issue 9, Sept. 2007), pp 2389-2394

[8] M. Kachelrieß: Branchless Vectorized Median Filtering, Nuclear Science Symposium Conference Record (NSS/MIC), 2009 IEEE

[9] P. Kolte, R. Smith, W. Su: A Fast Median Filter using AltiVec, Computer Design, 1999 IEEE International Conference on Computer Design: VLSI in Computers and Processors

[10] NVIDIA OpenCL SDK, https://developer.nvidia.com/opencl

[11] NVIDIA GeForce GTX 1080 Whitepaper, http://international.download.nvidia.com/geforce-com/international/pdfs/GeForce_GTX_1080_Whitepaper_FINAL.pdf

[12] Intel Intrinsics Guide, https://software.intel.com/sites/landingpage/IntrinsicsGuide/

[13] ARM NEON Intrinsics Reference, ARM Infocenter, Document IHI 0073A

[14] B. Shinde, D. Mhaske, A. R. Dani: Study of Noise Detection and Noise Removal Techniques in Medical Images, International Journal of Image, Graphics and Signal Processing, 2012, 4(2):51-60

[15] H. Zaidi: PET/MRI: Advances in Instrumentation and Quantitative Procedures, An Issue of PET Clinics, Volume 11-2, Elsevier 2016

[16] H. M. Lodhi, S. H. Muggleton: Elements of Computational Systems Biology, 2010, Wiley

[17] H. Pinkard, K. Corbin, M. F. Krummel: Spatiotemporal Rank Filtering Improves Image Quality Compared to Frame Averaging in 2-Photon Laser Scanning Microscopy, 2016, PLoS ONE 11(3)