# Branchless Vectorized Median Filtering

Marc Kachelrieß, *Member, IEEE*

***Abstract*—Median filtering is an important tool in signal or image processing. Based on the vector capabilities of modern hardware, which allows for vectorized min, max and mask operations, we provide a median algorithm of complexity $O(NM)$ that is both branchless and vectorized. In contrast to conventional fast median filters, whose run–time is data–dependent and that can operate only on scalar data, its runtime is predictable and data–independent and it can simultaneously operate on several one–dimensional signals thereby making use of data–level parallelism.**

**Our branchless vectorized median (BVM) filter keeps track of a sorted array from which values are deleted and to which new values are inserted. As a spin–off effect we could also use our work to provide a data sorting algorithm that is branchless and vectorized. Although it is of $O(M^2)$ computational complexity while other sort algorithms are of $O(M \ln M)$ computational complexity, at least for typical data, it may outperform other implementations for small $M$. This is mainly due to the fact that we have no unexpected branches which would stall the instruction pipeline and that we can simultaneously operate on vectors of data. Here, however, we focus on median filtering.**

**BVM is compared to a median filter based on doubly linked lists (DLL) and to the median implementation of the Intel Performance Primitives (IPP) library. The comparison uses constant data and linear data, which are two unrealistic settings, and random data, which is more realistic. For constant data IPP is up to five times faster than BVM whose run–time does not depend on the data themselves. However, regarding the realistic case of median filtering noisy data BVM outperforms DLL by about a factor of 1.5 and IPP by about a factor of 2 in our CPU–based implementations, which is far more than we did expect initially.**

Listing 1: Reference deletion algorithm.

```
void DelRef(float * const Array, int const M,
            float const Del)
    {
    float const Inf=Infinity;

    float A=Array[0];
    for(int m=0; m<M-1; m++)
        {
        float B, C;

        if(A==Del) A=Inf;

        B=Array[m+1];
        C=min(A, B);
        A=max(A, B);
        Array[m]=C;
        }
    }
```

Listing 2: Reference insertion algorithm.

```
void InsRef(float * const Array, int const M,
            float const Ins)
    {
    float A=Ins;
    for(int m=0; m<M-1; m++)
        {
        float B, C;

        B=Array[m];
        C=min(A, B);
        A=max(A, B);
        Array[m]=C;
        }
    Array[M-1]=A;
    }
```

## I. Introduction

MEDIAN filtering is regarded superior to linear filtering in many cases. Especially outliers can be efficiently removed using a median filter. However, the median filter is not linear and cannot be implemented in Fourier domain. It rather requires to sort the data that lie in the filter range and to select the central element from this sorted array or at least to recursively loop over the data until the median is found [1], [2]. The run–time of performant implementations is therefore typically data–dependent: they are fast for constant data and slow for noisy data.

Our interest in median filtering is to remove spurious noise from CT rawdata by filtering each detector row with a median filter. We further make use of median filtering during ring artifact reduction in reconstructed CT images [3]. Especially the latter may become very time consuming when the median filter steps are not optimized thoroughly. Nevertheless, the method provided will certainly be useful for other applications in the field of medical imaging and in signal processing in general.

Prof. Dr. Marc Kachelrieß: Institute of Medical Physics (IMP), University of Erlangen–Nürnberg, Henkestr. 91, 91052 Erlangen, Germany.
Corresponding author: marc.kachelriess@imp.uni–erlangen.de

In general, we know that the $m$–th smallest element of a set of $M$ elements can be determined in $O(M)$ time [4], [5]. Since we want to perform median filtering of an array of $N$ entries, i.e. a running median, and not just the selection of one median value, we know that the set of $M$ elements nearly remains the same when incrementing the position of the filter from $n$ to $n+1$. The only thing to do is to delete one element from the list and to insert another element into the list.

Several book–keeping strategies are available to perform median filtering on one–dimensional signals [1]. For example, median filtering can be done by keeping track of a sorted list of $M$ elements that are arranged linear in memory, combined with deletion and insertion operations. At any time, the median is the element at position $m = M/2$ in the list. The deletion and insertion process itself requires $O(\ln M)$ binary search comparison operations to find the list position of the element to delete and the list position of the element to insert plus $O(M/4)$ copy operations to move list elements lying between the delete and insert positions by one position to the left or right. Our branchless vectorized median (BVM) filter is of a sorted list type.

Another method is to maintain a doubly linked list (DLL) where the array value is stored together with its predecessor

9781-4244-3962-1/09/$25.00 ©2009 IEEE

and its successor. To find the position of a new element to insert into the list requires traversing parts of the DLL which may be of $O(M)$ or even of $O(\ln M)$ complexity [1]. Insertion and deletion itself is then only of $O(1)$ complexity since fast deletion and insertion is the main feature of doubly linked lists. In total, a run–time of $O(N \ln M)$ may be achieved with such an approach. We will see that the DLL implementation available to us performs significantly slower than $O(N \ln M)$.

Implementations based on heaps have the potential to perform in $O(N \ln M)$ as well [1]. They keep track of a min and a max heap whose common root is the median value. Since the heap is a tree structure, insertion and deletion can be done in $O(\ln M)$ time.

This paper describes a simple median filter whose run–time is data–independent and which makes use of the vector capabilities of modern hardware. We are well aware of the fact that its theoretical run–time, measured in algorithmic complexity, is inferior to the methods based on more sophisticated data structures, at least for large filter sizes $M$. Nevertheless, the complexity of modern hardware which requires to take into account the main memory bandwidth, the cache structure and latency, as well as the capabilites of the CPU to run several execution pipelines in parallel, did motivate us to implement and evaluate the branchless vectorized median filter.

## II. Method

The BVM algorithm makes use of the fact that $\{\min(a,b), \max(a,b)\}$ sorts the two–element list $\{a,b\}$ and that modern architectures often implement the minimum and maximum functions in hardware which perform with a throughput of one clock cycle. Since no if–statements are involved in the hardware implementation the instruction flow through the processor is not interrupted and the performance of an algorithm based on these operations is highly predictable and data–independent. Further on, the vector units available perform these minmax operation element–wise on complete vectors such that $\{\min(\boldsymbol{a},\boldsymbol{b}), \max(\boldsymbol{a},\boldsymbol{b})\}$ sorts the vectors $\{\boldsymbol{a},\boldsymbol{b}\}$ with one clock cycle throughput. Given that we are filtering single precision floating point data (four bytes per float) such a vector comprises four elements on the Cell broadband engine (CBE) and on standard central processing units (CPU) while it is 16 elements on the Larrabee (LRB) architecture [6].

For convenience, we restrict ourselves to the sorting and median filtering of floating point values (four bytes per datum). The algorithm can be easily generalized to double precision, to integer values and other data types given that the hardware supports corresponding minmax operations.

Listing 1 shows how we can delete a value from an array that is already sorted. $A$ holds the current datum of the array (at position $m$). Whenever it equals the value to be deleted it is set to infinity which has the effect that this value moves to the very end of the array. The minmax statements simply sort adjacent elements and write them back from the registers into memory. The if–statement of the reference code will be replaced by a compare and mask instruction in the optimized and vectorized listings and therefore is no branch.

Listing 3: Reference deletion and insertion algorithm.

```
void DelInsRef(float * const Array, int const M,
               float const Del,
               float const Ins)
    {
    float const Inf=Infinity;

    float A=Array[0];
    float B=Ins;
    for(int m=0; m<M-1; m++)
        {
        float C, D;

        if(A==Del) A=Inf;

        C=Array[m+1];
        D=min(A, C);
        A=max(A, C);
        C=min(B, D);
        B=max(B, D);
        Array[m]=C;
        }
    Array[M-1]=B;
    }
```

Listing 4: Reference BVM algorithm.

```
void BVMRef(float const * const Src, float * const Dst,
            int const N, int const M)
    {
    CArray<float> Array(M, 0); Array=Src[0];

    for(int n=0, dM=M/2; n<N; n++)
        {
        float const Del=Src[n-dM-1];
        float const Ins=Src[n+dM  ];
        DelInsRef(Array, M, Del, Ins);
        Dst[n]=Array[dM];
        }
    }
```

The insertion algorithm of listing 2 is more simple since we simply add the value to be inserted to the sorting process and carry it through the array while sorting with the minmax instructions.

The deletion and insertion process can be combined in one loop as shown in listing 3. Depending on the architecture it may be of advantage to use the two separate loops of listings 1 and 2 or to use the combined loop of listing 3. In our case the latter turned out to be slightly faster and therefore we will show the vectorized and processor specific listings corresponding to the combined reference listing.

The median filter itself is of size $M$ and acts on data of size $N$ (listing 4). It is realized by shifting data values through the sorted array using the delete and insert process described above. Note that listing 4 does not correctly handle boundary conditions or boundary effects because it rather intends to illustrate the general idea of how to carry out the filtering.

A vectorized version of listing 3 optimized for CPU–based architectures that makes use of SSE intrinsics is given in listing 5. $L = 4$ data arrays are simultaneously median filtered with this version of the algorithm. The inner loop consists of nine instructions. Note, how the if–statement was implemented branchless by using a compare–equivalent instruction followed by masking the array of infinity values followed by adding the masked infinities to the vector $A$. The corresponding median algorithm that calls the function is very similar to the reference median algorithm (listing 4) and therefore not shown here.

The LRB instructions are more powerful than the SSE

Listing 5: CPU–based deletion and insertion algorithm.

```
void DelInsCPU(float * Array, int const M,
               float const * const Del,
               float const * const Ins)
    {
    __m128 const Inf=_mm_set1_ps(Infinity);
    __m128 const Del=_mm_load_ps(Del);

    __m128 A=_mm_load_ps(Array);
    __m128 B=_mm_load_ps(Ins);
    for(int m=0; m<M-1; m++, Array+=4)
        {
        __m128 C, D;

        C=_mm_cmpeq_ps(A, Del);
        C=_mm_and_ps(Inf, C);
        A=_mm_add_ps(A, C);

        C=_mm_load_ps(Array+4);
        D=_mm_min_ps(A, C);
        A=_mm_max_ps(A, C);
        C=_mm_min_ps(B, D);
        B=_mm_max_ps(B, D);
        _mm_store_ps(Array, C);
        }
    _mm_store_ps(Array, B);
    }
```

Listing 6: LRB–based deletion and insertion algorithm.

```
void DelInsLRB(float * Array, int const M,
               float const * const Del,
               float const * const Ins)
    {
    _M512 const Inf=_mm512_set_1to16_ps(Infinity);
    _M512 const Del=_mm512_loadd(Del);

    _M512 A=_mm512_loadd(Array);
    _M512 B=_mm512_loadd(Ins);
    for(int m=0; m<M-1; m++, Array+=16)
        {
        _M512 C, D;

        __mmask Mask=_mm512_cmpeq_ps(A, Del);
        A=_mm512_mask_add_ps(A, Mask, A, Inf);

        C=_mm512_loadd(Array+16);
        D=_mm512_min_ps(A, C);
        A=_mm512_max_ps(A, C);
        C=_mm512_min_ps(B, D);
        B=_mm512_max_ps(B, D);
        _mm512_vstored(Array, C);
        }
    _mm512_vstored(Array, B);
    }
```

instructions. LRB has the possibility to use the result of the comparison operation as a mask that can be directly used as an additional argument to the vector operations. This simplifies the comparison statements for the delete value. LRB's code uses $L = 16$ and is shown in listing 6. Only eight instructions are needed in the inner loop.

## III. Results

Performance was evaluated running the median filters on two platforms, a PC and a server. The PC is a Celsius R670 workstation (Fujitsu Technology Solutions) equipped with two Xeon W5590 quad core processors running at $3.33 \cdot 10^9$ Hz with $4 \cdot 256$ kB L2 cache and 8 MB L3 cache per processor and 48 GB RAM. Neither its two NVIDIA Tesla C1060 boards nor its NVIDIA Quadro NVS 290 GPU were used in this study. This PC has a total of 8 compute cores.

The server is a Caneland platform (Intel Corporation) equipped with four Xeon X7460 hexa core processors running at $2.66 \cdot 10^9$ Hz with $3 \cdot 3$ MB L2 cache and 16 MB L3 cache per processor and 32 GB RAM. This server has a total of 24 compute cores.

Our 64 bit code was compiled using the Microsoft Visual Studio 2008 and is running on the Windows XP operating system. To optimize performance we make use of loop unrolling to reduce dependencies between adjacent instructions. For multi–threading we use OpenMP to simultaneously filter several vector arrays. The number of threads used equals the number of cores available (8 for the PC, 24 for the server).

BVM is benchmarked against Intel's IPP median algorithm and against another proprietary median algorithm based on doubly linked lists (DLL). The IPP and the DLL contain data–dependent branches and are therefore not vectorizable. Hence $L = 1$ for these two implementations.

To quantify performance we compute the operation count. The number of updates required to filter one array of size $N$ with a median filter of size $M$ with our BVM algorithm is $NM$. Since we use the data–level parallelism of the data we do not filter a single array but rather $L$ arrays in a vectorized manner. As stated above $L = 1$ for the IPP and DLL, and $L = 4$ for BVM running on CPU–based systems. To improve the accuracy of our timing measurements we actually do not only filter $L$ arrays but rather $KL$ arrays with $K$ chosen such that $KL$ is constant and large. The values we use for our investigations are $N = 2048$ and $KL = 16384$ while the filter size $M$ ranges from 3 to 513 in increments of 2. The update count is given as $NMLK$. It is divided by $1024^3$ and the resulting number is called giga updates (GU). Performance is measured in how many giga updates can be performed per second, i.e. in giga updates per second (GUPS).

We choose four different data models: constant, linear, constant plus noise, linear plus noise. In the constant case all data values were set to zero, in the linear case entry $n$ was set to $n$ in 50% of the $KL$ data arrays processed and it was set to $-n$ in the remaining 50% of the arrays. For the noise case we realized a normal random number of expectation 0 and variance 1 for each datum (i.e. $NLK = 2^{25}$ realizations in total). The linear plus noise model is given by the sum of the linear and of the noise model. The reason why we chose two different slopes for the linear model is that we observed that IPP performs up to 30% better for the positive slope than for the negative slope. Our choice therefore yields an average performance value.

### A. Comparison with other Implementations

Results for BVM in comparison to the IPP median and in comparison to the DLL median are shown in table I using a filter size of $M = 257$. For the unrealistic case of constant data IPP shows an enormous performance. Whenever we simulate realistic data by using data linear in $n$ or by filling the array with Gaussian noise the IPP performance is significantly impaired. Comparing with BVM, whose run–time is independent of the data, shows a performance benefit by roughly 100% when switching to the BVM code.

The DLL median performs similar like IPP and its performance is also data–dependent. In the realistic case of noisy data it is outperformed by BVM by a factor of about 2.

| Code | Data | PC | Server |
|---|---|---|---|
| IPP | const | 74 GUPS | 47 GUPS |
| IPP | linear | 4.8 GUPS | 12 GUPS |
| IPP | const plus noise | 7.9 GUPS | 19 GUPS |
| IPP | linear plus noise | 7.8 GUPS | 18 GUPS |
| DLL | const | 45 GUPS | 17 GUPS |
| DLL | linear | 41 GUPS | 17 GUPS |
| DLL | const plus noise | 10 GUPS | 17 GUPS |
| DLL | linear plus noise | 10 GUPS | 17 GUPS |
| BVM | any | 16 GUPS | 34 GUPS |

TABLE I
PERFORMANCE RESULTS FOR MEDIAN FILTERING WITH $M = 257$.

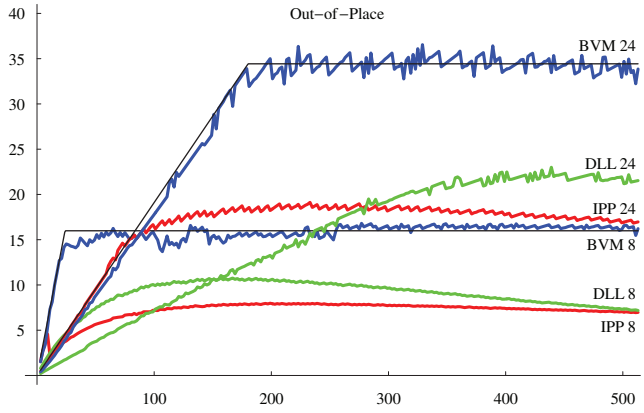


Fig. 2. IPP (red), DLL (green) and BVM (blue) performance in GUPS as a function of $M$ ranging from 3 to 513 in steps of 2 for the linear plus noise data model measured using 8 threads on the PC (lower curves) and 24 threads on the server (upper curves). The solid curves show a fit using an improved performance model for the BVM algorithm.

To give a numerical example regard BVM on the dual Quad core PC which achieves 16 GUPS. Having a median filter of 257 elements this means that the filter returns nearly 67 million values per second on a standard PC.

Instead of focussing on a single filter size let figure 1 illustrate the data–dependency for a large range of filter sizes ranging from 3 through 513. In all realistic cases, i.e. those data models that contain noise, BVM outperforms IPP as well as DLL.

Figure 2 plots the median filter performance using the linear plus noise data model for all three algorithms IPP, DLL, and BVM for both architectures, the PC and the server, as a function of the filter size $M$. As expected, the server with its 24 compute cores outperforms the PC which only has 8 compute cores. The peak performance of BVM is about 35 GUPS on the server and about 16 GUPS on the PC.

Regarding the IPP and the BVM curve for the 24 core server we observe a sawtooth behaviour in the plateau region of both curves. It appears to be more regular for the IPP curve (red plot) than for the BVM curve (blue plot). The distance of the sawtooth peaks is 4 increments in $M$ for the IPP while it is 8, 16 or 20 for the BVM. Probably, the behaviour results from cache misses when array data need to be loaded from the RAM into the CPU. The plots for the 8 core PC do not show such a sawtooth. This may be due to either requiring less memory bandwidth (since only 8 cores are active) or due to different

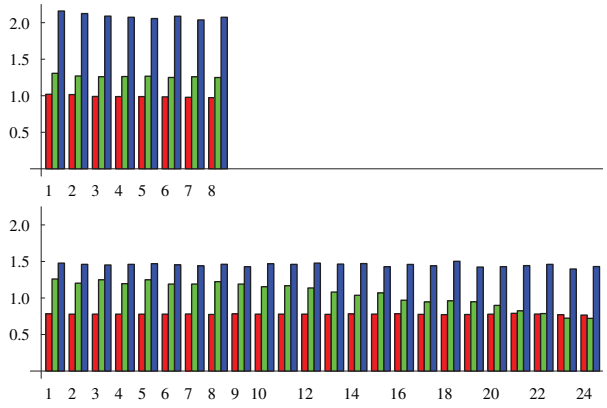


Fig. 3. IPP (red), DLL (green) and BVM (blue) performance in GUPST (GUPS per thread) as a function of the number of threads simultaneously running, plotted for $M = 257$ using the realistic data model linear plus noise. For the PC (top) we have up to 8 threads while the server (bottom) allows to use as much as 24 threads.

cache prefetching strategies on the PC architecture.

The small cusp apparent in IPP's PC curve for small filter sizes indicates that IPP switches to a different algorithm whenever $M$ is small. To be more quantitative, the cusp's peak is at $M = 9$ with a height of 4.6 GUPS.

To finish our analysis on the three median filters available to us let us regard the scalability of the approaches. Figure 3 shows the performance per thread, given in giga updates per thread (GUPST), achieved when increasing the number of threads from 1 to the maximum number of threads available (which equals the number of cores in our case). IPP as well as BVM show a perfect scalability since the performance per thread is constant for all levels of parallelization. The DLL algorithm however shows a decreasing performance per thread when more than 12 threads are running simultaneously on the 24 core server. The reason may be that the doubly linked lists that are used in the DLL median require increased memory bandwidth since each element in the list of $M$ elements is linked twice (one predecessor and one successor link).

### B. Improved Performance Model for CPU–based BVM

Apart from the sawtooth variation, that we observed in figure 2, we also find from figure 2 that the BVM performance increases linear in $M$ for small $M$ while it saturates into a plateau for large $M$. The plateau is the expected behaviour: our operation count $NMLK$ reflects the design of BVM and therefore we expect the performance measure (our GUPS value) to be independent of $M$. This, however, is not the case for small $M$. The computation time rather seems to contain a constant offset $O$ that reflects the time required to load all $NLK$ data values from the main memory into the processors and to store them back to RAM after median filtering. Whenever $M$ is greater than this offset the computation time is given by the time needed to carry out the $M$ operations necessary to find one median value.

Hence, a better model would be $NLK(O \vee M)$ with $O$ being the offset required to describe the memory latency. We do not want to promote this model since the model itself

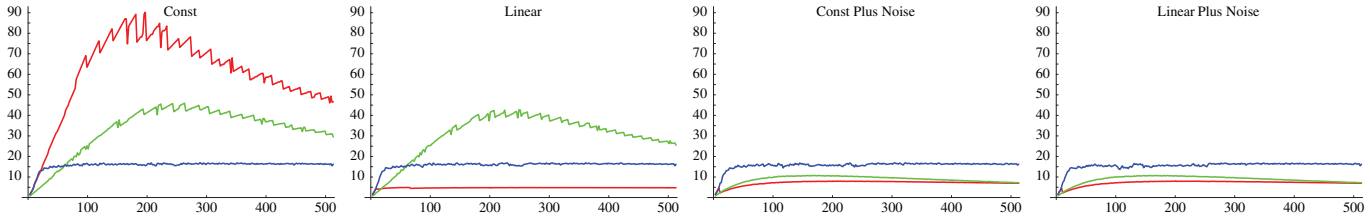


Fig. 1. IPP (red), DLL (green) and BVM (blue) performance in GUPS as a function of $M$ for the four data models constant, linear, constant plus noise, and linear plus noise. For the two realistic models (constant plus noise and linear plus noise) the BVM is significantly outperforming the other median filters. The timing was measured using 8 threads on the PC.

is architecture–dependent and it introduces a new parameter $O$ that is dependent on the compute architecture as well. However, we can determine $O$ for the performance data used so far. By fitting we find $O = 23.9$ for the 8 core PC and $O = 180$ for the 24 core server. The corresponding fit curves are shown as solid black curves in figure 2. Their plateau is at $SO = 16.0$ GUPS for the PC and at $SO = 34.4$ GUPS for the server. The plateau value is the mean performance of BVM when operating in the compute–limited region $M > O$. The slope $S$ for the linear region can be computed by dividing the plateau performance value by the offset value and yields $S = 0.67$ GUPS for the PC and $S = 0.19$ GUPS for the server. This indicates that the memory access is much better for the PC.

Although a similar linear increase is observed for the other median filter implementations it should be noted that it makes no sense to do a similar fit for the IPP and the DLL performance curves since the internal algorithmic structure is proprietary and not known to us.

We may further proceed interpreting the slope values. Since $S$ is given by dividing $NLK$ by the time required to load and store all data ($NLK$ data values are being filtered) the slope reveals information about the memory bandwidth utilized. Since each datum is four bytes and since for each datum we need to perform read and write we find $0.67$ GUPS $\cdot$ 4 B $\cdot$ $2 = 5.35$ GB/s utilized bandwidth on the PC. Can't we do better? The code assessed so far performs out–of–place median filtering, which means that the source and destination arrays are disjunct and need both to be loaded into cache.

Thus, let us switch to the in–place versions of the median filters where the source and destination arrays occupy the same memory. Figure 4 shows the in–place performance as a function of $M$. Compared to figure 2 the slope values of BVM are significantly increased. IPP and DLL seem not to make much differences between in–place and out–of–place computation apart from the larger cusp in the IPP curves for small filter sizes. This cusp, which we also found in the out–of–place curves and which indicates that IPP switches to a different implementation for $M \leq 9$, has its maximum for $M = 9$ at 19.2 GUPS on the PC and at 12.0 GUPS on the server. Note that the slope values of the rising side of the cusps are consistent with the slope values determined by fitting of the BVM curves.

Our observations regarding BVM's in–place and out–of–place versions are summarized in table II. It shows BVM's slope $S$, offset $O$, and plateau values $SO$ together with the

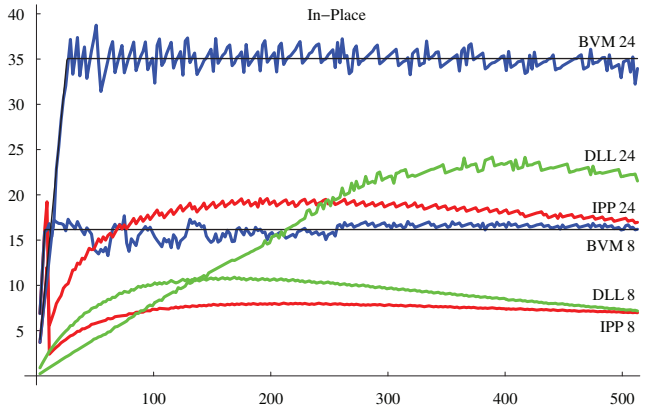


Fig. 4. Performance of the median filters' in–place versions. Same data and layout as figure 2. For small $M$ the slope of the BVM curves is significantly increased compared to the out–of–place code (c.f. figure 2).

| | $S$ | $O$ | $SO$ | Bandwidth |
|---|---|---|---|---|
| out–of–place, PC | 0.67 GUPS | 23.9 | 16.0 GUPS | 5.35 GB/s |
| in–place, PC | 2.29 GUPS | 7.04 | 16.2 GUPS | 18.3 GB/s |
| out–of–place, server | 0.19 GUPS | 180 | 34.4 GUPS | 1.53 GB/s |
| in–place, server | 1.34 GUPS | 26.1 | 35.1 GUPS | 10.8 GB/s |

TABLE II
PERFORMANCE OF BVM FILTERING. THE PLATEAU HEIGHT $SO$ IS THE AVERAGE BVM PERFORMANCE.

memory bandwidth $S \cdot 4$ B $\cdot 2$ provided for median filtering. The in–place version improves the slope and bandwidth by a factor of 3.4 on the PC and by a factor of 7.1 on the server compared to the out–of–place version. The peak performance is, however, nearly identical.

This analysis further shows us that there is only one point where the system is well balanced: whenever BVM operates at the kink in the curve, i.e. using a filter size of $M = O$, both the memory performance and the compute performance are at peak level. For $M < O$ the median filter is bandwidth–limited, and BVM cannot be outperformed by any other median filter, while for $M > O$ the median filter is compute–limited. If $M < O$ one may therefore reduce the number of active BVM threads and use the remaining ressources to perform other computations (given that those do not require signficant bandwidth).

### C. Absolute BVM Performance

How good is the BVM performance on an absolute scale? The dual Quad core PC with its eight cores processing four floats per vector can achieve a peak performance of 32 computations per clock cycle and it has $3.33 \cdot 10^9$ clock cycles per second. Hence, the theoretical peak maximum is 99.2 giga computations per second. Comparing this value to the BVM performance of 16.0 GUPS shows that there are 6.2 computations (clock cycles) per update required on average.

For the hexa core server with its $2.66 \cdot 10^9$ clock cycles per second, where 96 operations (24 cores with four floats per vector) can be performed per clock cycle, we find that BVM on average requires 7.1 computations (clock cycles) per update to achieve the mean performance of 35.1 GUPS.

Note that the innermost loop of the CPU–based BVM implementation (listing 5) consists of 9 instructions. Obviously all instruction latencies can be completely hidden and some instructions can even be issued in parallel on different execution pipelines.

## IV. Discussion

Branchless programming can be useful for many applications. In real–time applications it is of advantage since the run–time of functions is predictable. In other cases, data–dependent branches, that are nearly always unpredictable, yield a significant performance penalty and it may make sense to use code that is branchless (if this is possible) even if the operation count is much higher. This strategy may also be of advantage when designing code for GPUs, FPGAs, or dedicated signal processors.

The BVM algorithm is such an example. Based on minmax sorting we achieve a constant run–time. BVM significantly outperforms other high performance implementations such as the IPP median, while BVM's source code is of striking simplicity.

For small $M$ dedicated hardcoded implementations may be better. For example Basu and Brown proposed a dedicated three tap median filter requiring at most 2.5 comparisons per element [7]. Analyzing listing 7 one finds that it lends itself to a branchless and vectorizable implementation. Listing 8 shows such an implementation based on Basu and Brown's algorithm for $M = 3$ (this implementation, however, was not used for the timing measurements presented in this paper). Since it uses nothing but minmax functions it is vectorizable and branchless. It further requires three instructions per (vector) element only.

Regarding the BVM algorithms we find that the minmax operations always occur in pairs. A sort instruction $\mathrm{sort}(\boldsymbol{a}, \boldsymbol{b})$ that sorts the vectors $\boldsymbol{a}$ and $\boldsymbol{b}$ element–wise could be realized in hardware using the same logics as the minmax functions. To show the high importance of such a sort function regard listing 9 that uses such a hypothetical sort instruction, and compare that listing to listing 3. Note that the availability of such an instruction would reduce the number of instructions by two compared to the minmax–based deletion and insertion algorithms. It further requires less resources since register $D$ is not used anymore.

Modern hardware, such as the Larrabee, provides instructions with three source operands. Therefore, one can even think of instructions $\min(\boldsymbol{a}, \boldsymbol{b}, \boldsymbol{c})$, $\mathrm{med}(\boldsymbol{a}, \boldsymbol{b}, \boldsymbol{c})$, $\max(\boldsymbol{a}, \boldsymbol{b}, \boldsymbol{c})$, and $\mathrm{sort}(\boldsymbol{a}, \boldsymbol{b}, \boldsymbol{c})$ that return the element–wise minimum, the median or the maximum of its three vector operands, or that put the three operands element–wise into ascending order. Since all these new instructions can be designed to have a throughput of one clock cycle and a latency of two or three clock cycles they would further speed up the BVM and related algorithms. Hopefully, the CPU designers will provide such powerful instructions in future hardware.

A final point on this wishlist would be the possiblity to return or sort an index number together with the associated value. This could be realized in hardware as well, namely by treating each pair of vector entries as a pair (value, index). For example consider the vectors $\boldsymbol{a} = (a_0, a_1)$ and $\boldsymbol{b} = (b_0, b_1)$ with the first entry serving as the value according to which to

Listing 7: Basu and Brown's median for $M = 3$.

```
void BnB3Ref(float const * const Src, float * const Dst,
             int const N)
    {
    for(int n=0; n+3<N; n+=2)
        {
        float A, B;
        if(Src[n+1]<Src[n+2]) A=Src[n+1], B=Src[n+2];
        else                  A=Src[n+2], B=Src[n+1];

             if(Src[n+0]<A) Dst[n+1]=A;
        else if(Src[n+0]>B) Dst[n+1]=B;
        else                Dst[n+1]=Src[n+0];

             if(Src[n+3]<A) Dst[n+2]=A;
        else if(Src[n+3]>B) Dst[n+2]=B;
        else                Dst[n+2]=Src[n+3];
        }
    }
```

Listing 8: Reference BVM for $M = 3$.

```
void BVM3Ref(float const * const Src, float * const Dst,
             int const N)
    {
    for(int n=0; n+3<N; n+=2)
        {
        float const A=min(Src[n+1], Src[n+2]);
        float const B=max(Src[n+1], Src[n+2]);

        Dst[n+1]=min(max(Src[n+0], A), B);
        Dst[n+2]=min(max(Src[n+3], A), B);
        }
    }
```

Listing 9: Deletion and insertion with sort instruction.

```
void DelInsRef(float * const Array, int const M,
               float const Del,
               float const Ins)
    {
    float const Inf=Infinity;

    float A=Array[0];
    float B=Ins;
    for(int m=0; m<M-1; m++)
        {
        float C;

        if(A==Del) A=Inf;

        C=Array[m+1];
        sort(C, A); // now C<=A
        sort(C, B); // now C<=B
        Array[m]=C;
        }
    Array[M-1]=B;
    }
```

sort and the second entry corresponding to the index. Then,

$$\min(\boldsymbol{a}, \boldsymbol{b}) = \begin{cases} \boldsymbol{a} & \text{if } a_0 < b_0 \\ \boldsymbol{b} & \text{else} \end{cases}$$

would yield the desired behaviour. A corresponding max and a corresponding sort function could be realized as well, together with the three–operand versions. With today's hardware branchless vectorized sorting of value–index–pairs can only be done with a dirty workaround: use the most significant bits of a data type to store the value and the least significant bits to hide the index.

## V. Summary

The branchless vectorized median (BVM) proposed in this paper significantly outperforms other median filtering algorithms under realistic conditions. Since its performance is data–independent and therefore highly predictable it can be useful not only for the field of medical imaging but also for real–time applications such as stream processing. Due to avoiding generating or searching data–dependent decision trees the BVM filter is inherently vectorizable, i.e. it can simultaneously operate on many data streams. If only one data array is to be median filtered one can still make use of the vectorization property by splitting the one array into $L$ subarrays that can be simultaneously median filtered. Another nice feature of the BVM algorithm is its simplicity: a few lines of source code suffice to implement BVM.

We believe that our approach of using minmax sorting to gain data–independent run–time can be also useful to improve the speed of data sorting. We further see the potential to generalize BVM from one dimension to two– or multi–dimensional filtering, and to use the concepts to implement other types of rank filters. And last but not least a related paper shows that the BVM approach is advantageous for the use on graphics processing untis, too [8].

## Acknowledgments

This study was supported by RayConStruct GmbH, Nürnberg, Germany. The high performance hardware was provided by the Intel Corporation and by Fujitsu Technology Solutions GmbH. We thank Sven Steckmann for valuable discussions and hints regarding the improved performance model.

## References

[1] M. Juhola, J. Katajainen, and T. Raita, "Comparison of algorithms for standard median filtering," *IEEE Transactions on Signal Processing*, vol. 39, no. 1, pp. 204–208, Jan. 1991.

[2] W. H. Press, S. A. Teukolsky, W. T. Vetterling, and B. P. Flannery, *Numerical Recipes in C*. Cambridge University Press, 1992.

[3] D. Prell, Y. Kyriakou, and W. A. Kalender, "Comparison of ring artifact correction methods for flat–detector CT," *Phys. Med. Biol.*, vol. 54, pp. 3881–3895, 2009.

[4] M. Blum, R. Floyd, V. Pratt, R. Rivest, and R. Tarjan, "Time bounds for selection," *Journal of Computer and System Sciences*, vol. 7, pp. 448–461, 1972.

[5] D. E. Knuth, *The Art of Computer Programming*, 3rd ed. Addison Wesley Longman, 1997.

[6] L. Seiler, D. Carmean, E. Sprangle, T. Forsyth, M. Abrash, P. Dubey, S. Junkings, A. Lake, J. Sugerman, R. Cavin, R. Espase, E. Grochowski, T. Juan, and P. Hanrahan, "Larrabee: A many–core x86 architecture for visual computing," *ACM Transactions on Graphics*, vol. 27, no. 3, pp. 18:1 – 18:15, Aug. 2008.

[7] A. Basu and C. Brown, "Algorithms and hardware for efficient image smoothing," *Computer Vision, Graphics and Image Processing*, vol. 40, pp. 131–146, Feb. 1987.

[8] W. Chen, M. Beister, Y. Kyriakou, and M. Kachelrieß, "High performance median filtering using commodity graphics hardware," *IEEE Medical Imaging Conference Program*, p. submitted, Oct. 2009.