# A Fast Median Filter using AltiVec

Priyadarshan Kolte  Roger Smith  Wen Su
*Motorola, Inc., Austin, Texas*
*pkolte,rogers,wens@ncct.sps.mot.com*

## Abstract

*This paper describes the design and implementation of a median filter for graphics images on the Motorola AltiVec architecture. The filter utilizes 16-way SIMD parallelism to filter images at rates of 1.15 cycles/pixel for $3 \times 3$ squares and 6.6 cycles/pixel for $5 \times 5$ squares.*

*The median filter is based on a new sorting network which sorts $N^2$ numbers (arranged in an $N \times N$ square) by sorting all columns, rows, and diagonal lines in the square. This paper also describes a scheme for efficient testing of the sorting network.*

## 1. Introduction

Two dimensional graphics images are rectangles of pixels, where each pixel has a discrete value. A *median filter* computes the output value of each pixel in an image as the median value of pixels in a small neighborhood such as the 9 pixels in the $3 \times 3$ square centered around the pixel [8]. This type of non-linear filtering is effective in removing random noise from images without causing the blurring that is typical of linear low–pass filters, and it is offered by popular image processing programs. However, median filtering for images containing many pixels is a CPU intensive task, and there is a need to improve execution time.

A dominant factor in the execution time of the filter is performing compares of pixel values to find the median value. There are three ways of improving the time required by these compare operations. The first is to reduce the number of compares needed. The second is to reuse the results of compares performed within one square for finding the medians of adjacent overlapping squares. The third way is to use the parallelism offered by modern CPU architectures to compare multiple pixel values simultaneously. We have developed a median filter which uses all three ways to improve the execution time.

Our implementation of the median filter targets the Motorola AltiVec architecture [1, 4]. This architecture extends the PowerPC architecture with vector instructions that can process 16 8-bit data values simultaneously. We have measured image processing rates of 1.15 cycles/pixel for $3 \times 3$ squares and 6.6 cycles/pixel for $5 \times 5$ squares using our median filter implementation. A comparable implementation of the median finding algorithm by Paeth [8] would require at least 2 cycles/pixel for $3 \times 3$ squares and 10.9 cycles/pixel for $5 \times 5$ squares. Thus, our implementation improves processing rates by at least 65% over prior work.

Our median filter is based on a new sorting network for numbers arranged in squares. This sorting network for squares operates by successively sorting columns, rows, and diagonal line segments of varying slopes.

Section 2 describes the scalar median algorithm and sorting network. Section 3 describes our technique for testing the correctness of the median algorithm and sorting network. Section 4 provides the details of the AltiVec implementation and explains how we parallelized the scalar median algorithm. Section 5 presents related work and Section 6 concludes.

## 2. Scalar median algorithm

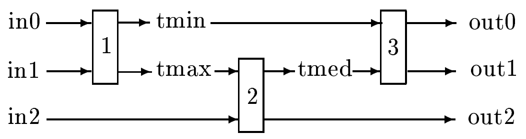
We first discuss a scalar algorithm for finding the median of a $3 \times 3$ square and then generalize the median algorithm for an $N \times N$ square, where $N$ is odd. We also describe sorting networks for completely sorting an $N \times N$ square, first for the case where $N$ is odd, and then for the case where $N$ is even.

### 2.1. Background on sorting networks

A `compare-swap` of two elements `(a,b)` compares and exchanges `a` and `b` so that we have `a <= b` after the operation. A *sorting network* is a sequence of `compare-swap` operations that depends only on the number of elements to be sorted, not on the values of the elements [5]. Bubblesort is an example of a sorting network whereas quicksort is not a sorting network.

Although a sorting network for a fixed number of elements usually requires a greater number of compare operations than a sorting method such as quicksort, the advantage of the sorting network is that the sequence of comparisons is fixed, so that the compare operations to be performed so not depend on the outcome of previously performed compare operations [6]. This absence of control–dependences in the sorting network makes it well suited to utilizing the data–parallelism offered by Single Instruction Multiple Data (SIMD) processors, since multiple independent arrays can be sorted in parallel using a fixed sequence of data–parallel `compare-swap` operations.

**Example sorting network.** The following figure shows an example of a sorting network for 3 elements (`in0, in1, in2`) such that the output (`out0, out1, out2`) is in non-decreasing order. The boxes implement `compare-swap` operations such that the lesser input value appears on the upper output and the greater input value appears on the lower output. For example, box 1 implements `compare-swap(in0,in1)` so that `tmin = min(in0,in1)` and `tmax = max(in0,in1)`.



**Notation.** We shall use the term *extremum* to mean a `min` or a `max` operation so that we can say that completely sorting 3 elements requires 6 extrema.

**Best sorting networks.** The table below (derived from the table on page 227 of Knuth's book [5]) shows values of $E(N)$, where $E(N)$ represents the number of extrema required by the best sorting networks (i.e., those which require the least number of extrema for sorting $N$ elements). We use these best sorting networks as subcomponents of our sorting networks.

| $N =$ | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
|---|---|---|---|---|---|---|---|---|
| $E(N) =$ | 0 | 2 | 6 | 10 | 18 | 24 | 32 | 38 |
| $N =$ | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 |
| $E(N) =$ | 50 | 58 | 70 | 78 | 92 | 102 | 112 | 120 |

**Dead code elimination.** In the sorting network for 3 elements shown above, if we need just the median of 3 elements, we omit the 2 extremum instructions for the non-median elements (`out0` and `out2`), and the number of extrema needed reduces from 6 to 4.

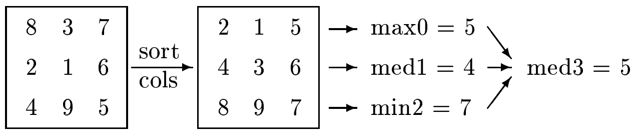
This optimization of omitting extremum instructions for results that are not needed is similar to the "dead code elimination" transformation performed by optimizing compilers. We use it to reduce the number of extrema in cases where a complete sort of all elements is not needed and a partial sort is sufficient.

### 2.2. $3 \times 3$ median algorithm

Algorithm `median_3` shown below finds the median of the 9 values in array $A[3][3]$ by first sorting each 3 element column, then finding the maximum element of the first row, the median of the second row, and the minimum of the third row, and finally finding the median of the three intermediate results.

```
Algorithm: median_3
Input: A is an unsorted 3x3 array
Output: median of the 9 values
  for c = 0 to 2
    sort column c so that A[r-1,c] <= A[r,c]
  let max0 = max (A[0,0], A[0,1], A[0,2])
  let med1 = med (A[1,0], A[1,1], A[1,2])
  let min2 = min (A[2,0], A[2,1], A[2,2])
  let med3 = med (max0, med1, min2)
```

The following figure shows an example to illustrate the operation of the `median_3` algorithm.



Using a sorting network to sort 3 columns of array $A$ requires $3 \times 6 = 18$ extremum operations, computing `max0` requires 2 extrema, `med1` requires 4 extrema, `min2` requires 2 extrema, and `med3` requires 4 extrema. Thus, `median_3` requires a total of 30 extrema.

### 2.3. $N \times N$ median algorithm (odd N)

We generalize `median_3` to work for all odd values of $N$ as shown below in algorithm `median_odd_N`.

```
Algorithm: median_odd_N
Input: A is an unsorted NxN array
Output: median of the NxN values
  let M = (N-1)/2
  /* Sort columns. */
  for c = 0 to N-1
    sort column c so that A[r-1,c] <= A[r,c]
  /* Partially sort rows. */
  for r = 0 to N-1
    sort row r so that A[r,c-1] <= A[r,c]
  for k = 1 to M
    /* Partially sort diagonals of slope k */
    for s = k*(M+1) to k*(M-1)+(N-1)
      sort line (k*r + c = s) so that
        A[r-1, s-k*(r-1)] <= A[r, s-k*r]
  let median = A[M,M]
```

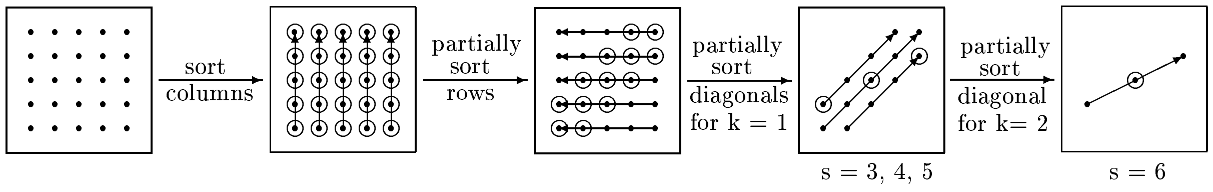


Figure 1. Lines sorted by `median_odd_N` for $N = 5$. `k` specifies slopes and `s` specifies needed diagonals.

Figure 1 illustrates the lines that are sorted by algorithm `median_odd_N` for $N = 5$. A dot denotes an array element and an arrow denotes a line segment that is sorted so that the least element is placed at the head and the greatest element is placed at the tail of the arrow. A dot with a circle around it denotes an array element that is computed by a partial sort because it is needed for a subsequent sort. A dot without a circle denotes an element that is an input value for a partial sort but which need not be computed as an output because it no longer affects the value of the median element.

The number of extrema required for the median of an $N \times N$ array depends on the number of extrema required to sort $N$ elements. Using the values of $E(N)$ from the table for "best sorting networks," the number of extrema required by algorithm `median_odd_N` (without using partial sorts) is $M(N^2)$ as shown below.

| $N$ | 3 | 5 | 7 | 9 | 11 | 13 | 15 |
|---|---|---|---|---|---|---|---|
| $N^2$ | 9 | 25 | 49 | 81 | 121 | 169 | 225 |
| $M(N^2)$ | 42 | 224 | 596 | 1236 | 2186 | 3484 | 5042 |

Using partial sorts (i.e., applying the dead-code elimination optimization to the row and diagonal sorts) reduces the required number of extrema as shown in the following table. The row labeled "Dead" shows the number of extrema eliminated from the previous count of extremum instructions, and $M(N^2)$ is the number of extrema required by algorithm `median_odd_N` after dead-code elimination (i.e., when using partial sorts).

| $N^2$ | 9 | 25 | 49 | 81 | 121 | 169 | 225 |
|---|---|---|---|---|---|---|---|
| Dead | 12 | 40 | 79 | 136 | 211 | 315 | 423 |
| $M(N^2)$ | 30 | 184 | 517 | 1100 | 1975 | 3169 | 4619 |

Note that the number of extrema $M(N^2)$ for $N = 3$ matches the count of 30 previously calculated for algorithm `median_3`.

## 2.4. $N \times N$ sorting algorithm (odd N)

Algorithm `median_odd_N` is extended to yield algorithm `sort_odd_N` that completely sorts all $N^2$ elements in an $N \times N$ square (for odd $N$) in non-decreasing order. The extension is to completely sort all rows and all diagonal line segments instead of just those line segments that affect the median value as shown below.

```
Algorithm: sort_odd_N
Input: A is an unsorted NxN array
Output: A is sorted in row-major order
  for r = 0 to N-1
    sort row r so that A[r,c-1] <= A[r,c]
  for k = 0 to N-1
    for s = 0 to (k+1)*(N-1)
      sort line (k*r + c = s) so that
        A[r-1, s-k*(r-1)] <= A[r, s-k*r]
```

The number of extrema required by `sort_odd_N` for small $N$ is $S(N^2)$ as shown in the following table.

| $N^2$ | 9 | 25 | 49 | 81 | 121 | 169 | 225 |
|---|---|---|---|---|---|---|---|
| $S(N^2)$ | 50 | 288 | 820 | 1804 | 3346 | 5572 | 8494 |

## 2.5. $N \times N$ sorting algorithm (even N)

Applying algorithm `sort_odd_N` to an array with even $N$ sometimes fails. For example, the following input $4 \times 4$ array is sorted into an output array that has 4 placed before 3 as shown in the circled elements.

$$\text{in} = \boxed{\begin{matrix} 1 & 1 & 1 & 4 \\ 1 & 2 & 5 & 6 \\ 3 & 6 & 6 & 6 \\ 6 & 6 & 6 & 6 \end{matrix}} \qquad \text{out} = \boxed{\begin{matrix} 1 & 1 & 1 & 1 \\ 2 & \textcircled{4} & \textcircled{3} & 6 \\ 6 & 6 & 6 & 6 \\ 6 & 6 & 6 & 6 \end{matrix}}$$

For the case of even $N$, we introduce additional `compare-swap` operations to correct the incorrect sort that was demonstrated by the example above. The following additional operations are done at the end of the body of the `k` loop of algorithm `sort_odd_N` to yield algorithm `sort_all_N`.

```
if (k > 1 and k evenly divides N)
  for w = 1 to N/k-1
    let c = k * w
    for r = N/k-w to N-1-w
      compare-swap (A[r,c-1], A[r,c])
```

The number of extrema required by the additional `compare-swaps` is in the row labeled "Extra" in the table below, which also shows $S(N^2)$, the total number of extrema required by `sort_all_N`.

| $N^2$ | 16 | 36 | 64 | 100 | 144 | 196 | 256 |
|---|---|---|---|---|---|---|---|
| Extra | 4 | 20 | 36 | 56 | 164 | 108 | 212 |
| $S(N^2)$ | 128 | 520 | 1232 | 2522 | 4400 | 7050 | 10348 |

Although `sort_all_N` also sorts arrays with odd $N$, when compared to `sort_odd_N`, algorithm `sort_all_N` requires 24 extra extremum instructions for $N = 9$ and 128 extra extremum instructions for $N = 15$, so we use `sort_all_N` only for even $N$.

The sorting algorithm as well as the median algorithm for $N \times N$ squares trivially extend to rectangles of size $N_1 \times N_2$ where $N_1 \leq N_2$.

## 3. Testing the sorting network

Exhaustive testing can verify the correctness of the median and sorting algorithms, but the challenge is to minimize the number of test inputs to be verified so that the time required for testing is manageable. An $N \times N$ array has $(N^2)!$ permutations as test inputs to the sorting network, but we can test using only $2^{(N^2)}$ permutations as test inputs by taking advantage of the well known 0-1 principle, which is given below.

**0-1 Principle:** If a sorting network with $n$ input lines sorts all $2^n$ sequences of 0's and 1's into nondecreasing order, it will sort any arbitrary sequence of $n$ numbers into nondecreasing order.

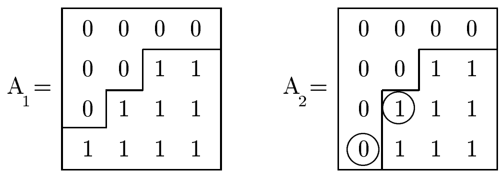
The use of the 0-1 principle dramatically reduces the number of test cases. For example, testing `sort_odd_N` for $N = 9$ requires $2^{81} = 2.4 \times 10^{24}$ inputs (instead of $81! = 5.8 \times 10^{120}$ inputs). However, even $10^{24}$ is huge, so we reduce the number of test inputs by using the following property of the sorting network that we proved (we omit details of the proof here).

**Two Conditions (TC):** If the sorting network used in `sort_odd_N` is given an input array $A[N][N]$, which contains only 0 and 1 values, then the following two conditions hold for array $A$ after completing the row sorts, the column sorts, and the sorts of all the diagonal lines for `k = 1`:

1) All rows are sorted and

2) All diagonal lines of slope 1 are sorted.

We say that an array $A$ satisfies TC if and only if it satisfies both the conditions given above.



Consider the example arrays $A_1$ and $A_2$ shown above. Array $A_1$ satisfies TC, but diagonal $r + c = 3$ of array $A_2$ violates the second condition because $A_2[2][1] > A_2[3][0]$ as shown in the circled elements.

If the diagonal line segment sorts for `k > 1` correctly sort all arrays $A$ that satisfy TC, then algorithm `sort_odd_N` correctly sorts all input arrays that contain 0 and 1 values.

The question is: "How many $N \times N$ arrays satisfy TC?" Consider the array $Step[N]$ which contains only 0 and 1 values, and the integer array $Zeros[N]$ derived from $Step[N]$ such that $Zeros[c] = \sum_{i=c}^{N-1} Step[i]$. $Zeros[i]$ represents the count of 0 values in column $i$ of array $A$, and $Step[i]$ represents the presence of a "step" between column $i$ and column $i + 1$. Any test input array $A$ that is generated from array $Zeros$ (which is generated from array $Step$) satisfies TC.

For example, the array $Step_1[4] = \{1, 1, 0, 1\}$ results in $Zeros_1[4] = \{3, 2, 1, 1\}$, which generates the array $A_1$ that satisfies TC as shown above. But, there is no array $Step_2[4]$ to generate the $Zeros_2[4] = \{4, 2, 1, 1\}$ that is needed to generate the array $A_2$.

Since the transformation from an array $Step$ to the array $Zeros$ to the array $A$ that satisfies TC is one-to-one, we can generate all cases of $A$ that satisfy TC by generating all cases of $Step$, transforming them to $Zero$ and then to cases of $A$. Since $Step$ has only $2^N$ distinct cases, there are only $2^N$ distinct cases for $A$ that need to be tested.

Thus, we are able to test the $9 \times 9$ sorting network using $2^9 = 512$ input cases of $A$, which is quite practical. Using the reduction just described, we have tested the sorting network as well as the median algorithms for array sizes up to $31 \times 31$.

It is possible to generalize the value of `k` in the TC property to improve testing efficiency. For example, testing networks with odd $N$ would be more efficient with `k = 2` because we would assume that the column and row sorts and the diagonal sorts for `k = 1` and `k = 2` were done, and then only test the diagonal sorts for `k > 2`. The number of possibilities for $Step$ (which contains only 0 and 1 values such that no two adjacent columns have 1 values) leads to $F_{N+2}$ input cases of $A$. $F_n$ represents the Fibonacci numbers, and $F_n$ is equal to the integer closest to $\phi^n/\sqrt{5}$, where $\phi = 1.618$ is the golden ratio [3]. Using this generalization of TC with `k = 2`, testing the $9 \times 9$ sorting network would require only $F_{11} = 89$ input tests.

## 4. Parallel $3 \times 3$ median filter

The scalar `median_3` algorithm forms the basis of the AltiVec median filter implementation described in this section. We first provide the necessary background on the AltiVec instruction set and notation and then show

how to parallelize the scalar algorithm. We describe additional optimizations that improve the execution time of the parallel median filter. Finally, we present measurements of the execution time of the parallel median filter on AltiVec simulators and hardware.

### 4.1. Background on AltiVec

The AltiVec architecture provides 128-bit vector registers and a set of instructions to operate on the vector registers. The `lvx` (Load Vector Indexed) instruction loads 16 contiguous bytes from memory into a vector register. Similarly, the `stvx` instruction, stores a vector register into memory.

The `vminub` (Vector Minimum Unsigned Byte) instruction has two operand vector registers and one result vector register; it treats each operand as a vector of 16 unsigned 8-bit bytes and computes a vector containing the 16 minimum values. For example, if the first operand is register `v1` and the second operand is register `v2`, and the register values are:

```
v1 = {16, 15, 14, 13, ...  4, 3, 2, 1}
v2 = { 5, 10, 15, 20, ...  65, 70, 75, 80}
```

then the instruction `vminub v3,v2,v1` computes

```
v3 = { 5, 10, 14, 13, ...  4, 3, 2, 1}.
```

Similarly, the `vmaxub` instruction computes the maximum of vectors containing 16 unsigned bytes.

AltiVec also provides the `vsldoi` (Vector Shift Left Double by Octet Immediate) instruction which has two vectors and one literal value as operands and one result vector. This instruction concatenates the two operand vectors and shifts the concatenated vector left by the number of bytes given by the literal value. For example, if `v1` and `v2` contains the values shown in the previous paragraph and the literal operand is `14`, the instruction `vsldoi v3,v1,v2,14` computes

```
v3 = {2, 1, 5, 10, 15, 20, ...  65, 70}.
```

### 4.2. Notation

In the rest of this paper, `vec_min` denotes the `vminub` instruction, `vec_max` denotes the `vmaxub` instruction, and `vec_sld` denotes the `vsldoi` instruction. Now, the term *extremum* means either a `vec_min` or a `vec_max` instruction.

### 4.3. AltiVec implementation

In function `vector_3x3_median`, we show the main steps for filtering an image and we omit the code for the boundaries (the median filter just copies input values to the output image for pixels on the boundaries). We assume that the width of the image (W) is a multiple of 16 to match the width of vectors in AltiVec.

The strategy is to use the steps of `median_3`, except that instead of processing one $3 \times 3$ square, we use vector operations to process a *tile* of 16 horizontally overlapping $3 \times 3$ squares. Once a tile is processed, we process the next tile on the same row. Once all tiles on a row are processed, we start with the next row, until all rows are done.

```
Function: vector_3x3_median
Input: IN is a HxW image
Output: OUT is a HxW filtered image
  for r = 0 to H-3
    for c = 0 to W-16 by 16
1:      let in0 = load IN[r][c..c+15]
        let in1 = load IN[r+1][c..c+15]
        let in2 = load IN[r+2][c..c+15]
2:      sort columns in (in0, in1, in2)
          to get (row0, row1, row2)
3:      shift and concatenate (old_row0, row0)
          to get (row0s, row0ss)
        let max0 = max (row0, row0s, row0ss)
4:      shift and concatenate (old_row1, row1)
          to get (row1s, row1ss)
        let med1 = med (row1, row1s, row1ss)
5:      shift and concatenate (old_row2, row2)
          to get (row2s, row2ss)
        let min2 = min (row2, row2s, row2ss)
6:      let med3 = med (max0, med1, min2)
7:      store m3 to OUT[r+1][c..c+15]
8:      let old_row0 = row0
        let old_row1 = row1
        let old_row2 = row2
```

**Details of steps.** The first step loads vectors of 16 pixel values from memory into vector registers `in0`, `in1`, and `in2`.

The second step sorts the columns to produce vectors `row0` (which contains the minimum), `row1` (which contains the median), and `row2` (which contains the maximum) using the following sorting network.

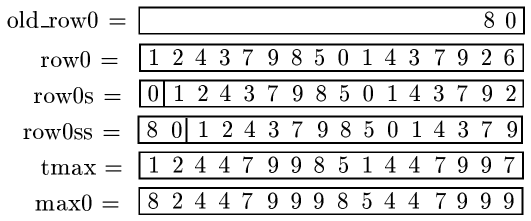
```
2:  let tmin = vec_min (in0,  in1)
    let tmax = vec_max (in0,  in1)
    let tmed = vec_min (tmax, in2)
    let row2 = vec_max (tmax, in2)
    let row0 = vec_min (tmin, tmed)
    let row1 = vec_max (tmin, tmed)
```

The third step computes the vector register `max0` to contain the maximum of every 3 elements in `row0`. In contrast to the case of column sorts, it is not obvious how to perform a parallel sort across a row stored in a vector register, but it is easy once we know the trick of right shifting a vector and performing column sorts.

We store the value of `row_0` from the previous iteration of the `c` loop in vector register `old_row0`, so that we can compute the maximum for the first two elements in `row0`. Vector register `row0s` contains `row0` shifted right once, and it is computed by concatenating the last 1 element of `old_row0` with the first 15 elements of row0. Similarly, vector register `row0ss` contains `row0` shifted right twice, and it is computed by concatenating the last 2 elements of `old_row0` with the first 14 elements of `row0`. We use the `vec_sld` instruction to provide the right shift with concatenation. Now, computing `max0` requires 2 extrema as shown below.

```
3: let row0s  = vec_sld (old_row0, row0, 15)
   let row0ss = vec_sld (old_row0, row0, 14)
   let tmax   = vec_max (row0, row0s)
   let max0   = vec_max (tmax, row0ss)
```

The figure below shows an example of the operation of the third step. As expected, the element at position $i$ of `max0` is the maximum of the elements at positions $i-2$, $i-1$, and $i$ of `row0`. Note that the element at position 0 of `max0` includes the maximum of the elements at positions 14 and 15 of `old_row0`.


old_row0 = 8 0
row0 = 1 2 4 3 7 9 8 5 0 1 4 3 7 9 2 6
row0s = 0 1 2 4 3 7 9 8 5 0 1 4 3 7 9 2
row0ss = 8 0 1 2 4 3 7 9 8 5 0 1 4 3 7 9
tmax = 1 2 4 4 7 9 9 8 5 1 4 4 7 9 9 7
max0 = 8 2 4 4 7 9 9 9 8 5 4 4 7 9 9 9

The fourth step is similar to the third step except that vector register `med1` contains the median of every 3 elements in `row1`. This step requires 4 extrema (as described in the example on dead code elimination). The fifth step computes the minimum of every 3 elements in a row using 2 extrema similar to the third step, and the sixth step computes the median of `(max0, med1, min2)` using 4 extrema. The seventh step stores the `med3` register to output memory, and the eighth step copies the current values of the `row` registers for use in the next iteration of the `c` loop.

Thus, the total number of extrema required to find the median of 16 squares is $(6+2+4+2+4) = 18$.

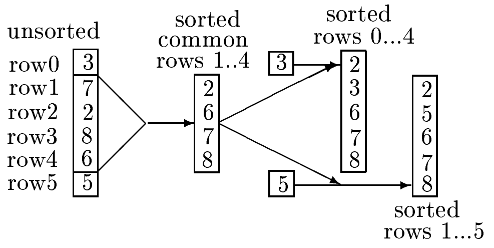
## 4.4. Reusing results of extremum instructions

The number of extrema required by the scalar median algorithm `median_3` is 30, yet function `vector_3x3_median` achieves the same result in just 18 extrema. The reason for the reduction is: When we filter multiple squares across the image, we reuse each sorted column for the 3 horizontally overlapping squares that the column appears in, so that for each square we effectively sort only 1 column instead of 3 columns. Since sorting a 3 element column requires 6 extrema, we save 12 extrema per square.

The regularity of the first step (the column sorts) in the median algorithm permits common sequences of sorts between horizontally overlapping squares to be shared among the squares. This is a significant feature because it saves a large number of extrema, $(N-1) \times E(N)$, in the general case. We show the number of extrema saved in the row labeled "Reused" in the following table for $M(N^2)$, which is the number of extrema required by `median_odd_N` after reusing extremum instructions.

| $N^2$ | 9 | 25 | 49 | 81 | 121 | 169 | 225 |
|---|---|---|---|---|---|---|---|
| Reused | 12 | 72 | 192 | 400 | 700 | 1104 | 1568 |
| $M(N^2)$ | 18 | 112 | 325 | 700 | 1275 | 2065 | 3051 |

In addition to taking advantage of columns that are shared between *horizontally* adjacent squares, we take advantage of rows that are shared between *vertically* adjacent squares to further reduce the number of extrema for column sorts. To sort two overlapping $N$ element columns, we first sort the $N-1$ element column that is common to both columns, and then merge the end element for each $N$ element column using $N-1$ `compare-swaps`. The following figure gives an example for $N=5$.



Rows 1, 2, 3, and 4 are sorted first using $E(4)=10$ extrema. Then, for the column containing rows 0 to 4, row0 is merged using 8 extrema. Similarly, for the column containing rows 1 to 5, row5 is merged using 8 extrema. The total number of extrema required for the two column sorts is $10+8+8=26$, instead of $2E(5)=36$, which saves 5 extrema per column.

Vertical sharing saves $E(N)-2(N-1)-E(N-1)/2$ extremum instructions per $N \times N$ square. The table below shows the number of extrema reused and $M(N^2)$, which is the number of extrema required by `median_odd_N` after vertical sharing.

| $N^2$ | 9 | 25 | 49 | 81 | 121 | 169 | 225 |
|---|---|---|---|---|---|---|---|
| Reused | 1 | 5 | 8 | 15 | 21 | 29 | 33 |
| $M(N^2)$ | 17 | 107 | 317 | 685 | 1254 | 2036 | 3018 |

To take advantage of vertical sharing, we modify `vector_3x3_median` so that the inner loop processes two vectors of medians in a tile. The loop body now computes the medians of 32 squares using 34 extrema.

### 4.5. Performance on AltiVec simulators

An implementation of the AltiVec architecture provides a latency as well as throughput of 1 cycle for all but one of the instructions used in the median filter implementation. (The exception is vector load, which has a latency of 2 cycles for hits in the level-1 cache and a greater latency for misses.) The implementation has one execution unit for performing extremum instructions, so the peak performance of the inner loop of `vector_3x3_median` is $34/32 = 1.06$ cycles/pixel.

For performance testing, we used an input image size of $128 \times 128$ pixels (chosen so that the image fits in level-1 cache) and simulated the operation of the median filter on a cycle-level simulator of a non-superscalar AltiVec implementation. The number of cycles obtained was 32K, which yielded a processing rate of 2 cycles/pixel. The overhead of the additional instructions such as `vec_load, vec_sld`, and `vec_store` prevented the non-superscalar implementation from achieving the peak processing rate.

A superscalar implementation of the AltiVec architecture executes the overhead instructions in parallel with the extremum instructions. So we scheduled the instructions of the median filter for a dual-issue processor and simulated it on a dual-issue AltiVec to obtain a cycle count of 20.6K cycles (a rate of 1.26 cycles/pixel).

The superscalar AltiVec simulation revealed that the inner loop executed the 34 extrema in 37 cycles. In order to increase parallelism in the inner loop, we used a technique similar to Software-Pipelining [7] to schedule the instructions of the loop body so that instructions from each loop iteration were executed in parallel with those from the subsequent iteration. The inner loop after software-pipelining executed in 34 cycles, and the complete filter executed in 18.9K cycles (a rate of 1.15 cycles/pixel).

### 4.6. Performance on AltiVec hardware

Execution of the software-pipelined $3 \times 3$ median filter on a hardware implementation of a dual-issue AltiVec was measured at 18.9K cycles for a $128 \times 128$ image, which exactly matches the simulator results.

In order to measure "parallel speedup," we implemented three non-AltiVec $3 \times 3$ median filters, the first based on bubblesort, the second based on quicksort [2], and the third based on our sorting network (which uses 18 extrema for computing 1 median). The number of cycles measured on AltiVec hardware for processing the $128 \times 128$ image for the bubblesort filter was 5.5M, for the quicksort filter was 3.25M cycles, and for the scalar sorting network filter was 950K cycles. Thus, the best scalar median filter required 950K cycles (a rate of 60 cycles/pixel).

The parallel speedup of $60/1.15 = 52$ is greater than 3 times the expected speedup of 16 because the AltiVec hardware offers the vector filter the advantage of executing an extremum in 1 cycle using an arithmetic unit. In contrast, the scalar filter requires approximately 3.5 cycles to execute an extremum using a compare instruction with a dependent conditional branch that is often mispredicted by the hardware.

In addition to the $3 \times 3$ median filter, we also implemented an optimized $5 \times 5$ vector median filter, which processes a $128 \times 128$ image in 108K cycles on the dual-issue AltiVec hardware (a rate of 6.6 pixels/cycle).

## 5. Related work

The problem of finding a sorting network for a specific number of elements is discussed in Knuth's book [5]. Figure 49 on page 228 of this book shows a sorting network for 9 elements from which it is possible to derive the scalar $3 \times 3$ median algorithm.

Knuth's book discusses the best known sorting networks as well as Batcher's merge-exchange sorting network. The following table compares the number of extrema needed by `sort_all_N` with those needed by Batcher's method and the best known networks.

| $N^2$ | 4 | 16 | 256 |
|---|---|---|---|
| Best known | 10 | 120 | 7302 |
| Batcher | 10 | 126 | 7678 |
| `sort_all_N` | 10 | 128 | 10348 |

It is clear from the table above that the sorting network we developed is not optimal for large $N$, but our sorting network requires a low number of extrema for small values of $N$, which are of greatest interest for median filtering of graphics images. Further, in contrast to most sorting networks, the regularity of the column sorts permits our sorting network to reduce the effective number of extrema required per square.

Leighton's book [6] describes Shearsort, a sorting network for $N^2$ elements arranged in an $N \times N$ square. Shearsort uses alternate phases of row and column sorts, such that the row sorts sort consecutive rows in opposite directions (to produce a "snakelike" order). The target computer is a two-dimensional mesh of processors, where each processor holds one array element. The metric used for evaluating the sorting algorithm is *timestep*, where a single timestep contains all the independent extremum instructions that can be done in parallel. The simplest variant of shearsort requires $N(2\log_2 N + 1)$ timesteps and a complex variant requires $N(\log_2 N + 3)$ timesteps.

Using shearsort for the median filter on AltiVec is not appropriate because the number of extrema needed by shearsort is very high compared to the parallelism offered by AltiVec (a small constant like 16). Although our sorting network requires fewer extrema than shearsort, using it on a mesh of processors (with one array element per processor) is not a clear winner because the number of time steps required for large $N$ would be greater than $N(2 + \log_e N + \gamma)$, where $\gamma = 0.577$ is Euler's constant. The problem is that our sorting network uses limited parallelism whereas the parallelism offered by a mesh of processors is high ($N^2$). Also, the communication in our sorting network when sorting diagonal line segments is not to the nearest neighbors in a mesh. Thus, our sorting network is well matched to the parallelism offered by AltiVec, but may not be suitable for other parallel architectures.

Before we developed our median filter, the median filters implemented for AltiVec were based on the median finding algorithms presented by Paeth [8]. The $3 \times 3$ algorithm by Paeth requires 40 extrema, and dead-code elimination reduces the number of extrema to 32. In contrast, our $3 \times 3$ algorithm requires 17 extrema. The $5 \times 5$ algorithm by Paeth requires 198 extrema, which again can be optimized to 174. (Our $5 \times 5$ algorithm requires 107 extrema.) One shortcoming of Paeth's algorithms is that it is not obvious how these can take advantage of sharing of extremum instructions between overlapping squares and still be vectorizable. Another shortcoming is that the generalization for arbitrary $N$ is not known.

## 6. Conclusions

This paper describes a new sorting network and a median filter based on it. The sorting network has the following desirable properties:

1. It requires a low number of extrema, especially for small squares, which are of most interest for median filtering of graphics images.
2. The regular structure of the column sorts allows extensive reuse of extremum operations across horizontally overlapping squares. Limited reuse across vertically overlapping squares also exists.
3. It is well matched to the parallelism offered by the target architecture.

The median filter makes effective use of the sorting network and the parallelism offered by the Motorola AltiVec architecture to deliver the fastest processing rates that we know of. It is scalable so that if one wanted to operate on 16-bit pixel values (instead of 8-bit values), very little change to the algorithm would be needed, but the processing rate per square would be cut in half. Similarly, if the architecture were changed to offer 256-bit vectors (instead of 128-bit vectors), the processing rate per square would double.

This paper also describes a practical method of testing the correctness of the sorting network and provides experimental measurements of the performance of the median filter. Our current work addresses analytical proofs of correctness and analyses of the asymptotic complexity of the median and sorting algorithms.

## Acknowledgements

We thank Manas Mandal for help with AltiVec hardware, and Kate Stewart and the anonymous reviewers for their reviews and suggestions for improvement.

PowerPC is a trademark of International Business Machines used by Motorola under license from IBM. AltiVec is a trademark of Motorola.

## References

[1] *AltiVec Technology Programming Environments Manual,* Motorola, 1998. (www.mot.com/AltiVec)

[2] T. H. Cormen, C. E. Leiserson, R. L. Rivest, "Medians and Order Statistics," *Introduction to Algorithms,* MIT Press, 1990, pp. 187–189.

[3] R. L. Graham, D. E. Knuth, O. Patashnik, "Fibonacci Numbers," *Concrete Mathematics,* 2nd ed., Addison-Wesley, 1994, pp. 290–301.

[4] L. Gwennap, "AltiVec Vectorizes PowerPC," *Microprocessor Report,* vol. 12, no. 6, May 1998.

[5] D. E. Knuth, "Networks for sorting," *The Art of Computer Programming, Vol. 3: Sorting and Searching,* Addison-Wesley, 1973, pp. 220–229.

[6] F. T. Leighton, "Sorting revisited," *Introduction to Parallel Algorithms and Architectures: Arrays, Trees, Hypercubes,* Morgan Kaufmann, 1992, pp. 139–148.

[7] S. S. Muchnick, "Window scheduling," *Advanced Compiler Design and Implementation,* Morgan Kaufmann, 1997, pp. 551–555.

[8] A. W. Paeth, "Median Finding on a $3 \times 3$ Grid," in *Graphics Gems*, edited by Andrew S. Glassner, Academic Press, 1990, pp. 171–175.