SHORT VECTOR SIMD PARALLELIZATION OF MAXIMUM FILTER

Rade Kutil and Erich Mraz
University of Salzburg, Department of Computer Sciences
Jakob Haringer-Str. 2, 5020 Salzburg, Austria
Email: rkutil@cosy.sbg.ac.at

Abstract

The maximum filter is used in mathematical morphology and other image and signal processing applications. A naive implementation of the maximum filter would be similar to an FIR filter and could be parallelized for short vector SIMD processor extensions such as SSE in the same way as FIR filters have been parallelized. However, there is a more efficient algorithm called van Herk/Gil-Werman algorithm, whose complexity is independent of the filter length. Therefore, this work investigates ways to parallelize this algorithm in 1D as well as 2D. The developed schemes achieve speedups between 2.8 and 6 on 16-fold SIMD.

1 Introduction

The maximum filter substitutes each sample $a(x)$ of a discrete signal a by the maximum of the signal samples within an interval $[x-k, x+l]$ around the sample's position x:

$$b(x) = \max_{i=-k}^{l} a(x+i) \tag{1}$$

This filter is a primitive operation in mathematical morphology [5], which has applications in object recognition [1], feature extraction [7], edge detection [4], image enhancement [3], and image compression [2].

When the filter is implemented in a straight forward way according to (1), the complexity depends linearly on the filter size

$$s = k + l + 1\,. \tag{2}$$

The complexity is

$$(k+l)n = (s-1)n\,, \tag{3}$$

where n is the data size. However, there exists a more efficient algorithm called van Herk/Gil-Werman (HGW) algorithm [9, 6], whose complexity is independent of the filter length. This algorithm works in blocks of size $s-1$. It uses two intervals of size $s-1$ to the left (c) and right (d) of a position u where the maximum between u and each position in the interval is cumulated, which can be done in linear complexity. Each desired maximum value of the output array b can then be calculated by a single maximum operation of two values, one from the left and one from the right interval. Figure 1 shows the algorithm. The outer loop over u iterates the blocks of size $s-1$, where u points to the center of the block. The first inner loop fills the cumulated maximum-array d to the right of u, the second inner loop fills the corresponding array c to the left of u. Finally, in the last loop a block of the output array b

```
for $u = l \ldots n$ step $s - 1$
    $d(0) = a(u)$
    for $i = 1 \ldots s - 2$
        $d(i) = \max(d(i-1), a(u+i))$
    $c(s-2) = a(u-1)$
    for $i = 1 \ldots s - 2$
        $c(s-i-2) = \max(c(s-i-1), a(u-i-1))$
    for $i = 0 \ldots s - 2$
        $b(u-l+i) = \max(c(i), d(i))$
```

Figure 1: Van Herk/Gil-Werman algorithm. Data outside of the array boundaries is assumed to have zero or minimum value.

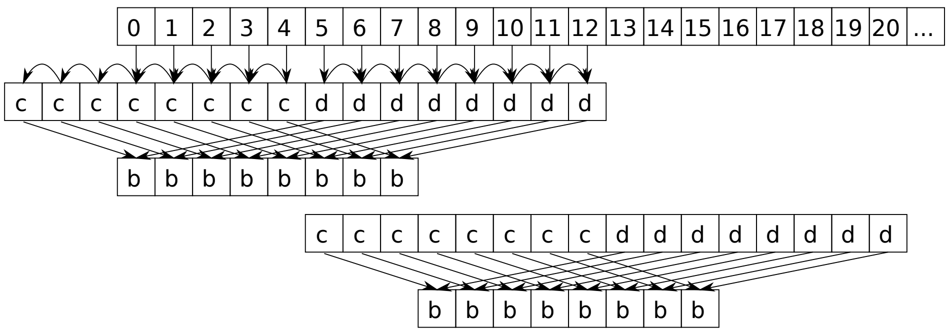

Figure 2: Example for van Herk/Gil-Werman algorithm for $k = 3$, $l = 5$, $s = 9$. c is the cumulative interval to the left, and d the one to the right of u. The first two iterations $u = l = 5$ and $u = l + s - 1 = 13$ are shown.

is calculated from one value of c and one from d for each output value. Figure 2 shows an example. Note that the algorithm in Figure 1 does not show the handling of array bounaries for reasons of comprehensibility. However, if outside-data is assumed to have zero (or negative maximum) values, the algorithm is correct. The actual implementation copes with the first and last u-iteration separately in order not to access outside-data and to avoid if-then-else-constructs in the main loop for performance reasons. The complexity of this algorithm is almost independent of the filter size s:

$$n\frac{2(s-2)+(s-1)}{s-1} = n\left(3 - O\left(\frac{1}{s}\right)\right) . \tag{4}$$

Even though this algorithm is very fast, processing of large images should be as efficient as possible. Short-vector SIMD extensions such as SSE or AltiVec are a convenient possibility for parallelization because they are available in almost all general purpose processors nowadays. They allow to process 16 bytes at a time, so 16 max-operations can be carried out in a single cycle. The bytes have to be arranged consecutively in vector registers, though, which is not always easy to do. There are instructions for almost arbitrary permutations and selections to rearrange the bytes properly. However, these instructions take additional processor time, thus reducing the speedup.

A 2-D maximum filter, in its simple form, consists of the 1-D filtering of all rows of the 2-D source data set, followed by the 1-D filtering of all columns. If a 2-D array of data with row-wise memory layout has to be processed, then sequential 1-D algorithms applied vertically to all columns can be parallelized easily by applying the sequential algorithm on vectors of 16-byte-rows in order to process 16 columns at once. On the other hand, true 1-D SIMD parallelizations, as required for 1-D data and rows of 2-D data, are always more complicated.

This is also true for the HGW algorithm. Recently, a short-vector SIMD parallelization of the HGW algorithm was presented in [8]. However, the horizontal part of the HGW is implemented by transposing the whole image and applying the vertical version. Moreover, results are only presented for the vertical 1-D part of the algorithm.

This work develops a true 1-D SIMD parallelization of the HGW algorithm. The vertical part of the 2-D algorithm is implemented as in [8], and performance results are shown for the combined 2-D algorithm. Intel SSE2 with vectors of 16 bytes is used. Experiments are performed on an Intel Core 2 Duo with 2.66 GHz. C++ code is compiled with gcc 4.2.1. The image size used for experiments is 2560×1024.

2 1-D SIMD parallelization

The 1-D parallel SIMD algorithm is executed in two stages. The first stage performs a maximum filter of length $r < 32$, which can be done in $\log_2(r)$ steps with SIMD operations. The second stage is analogous to the sequential algorithm, but works on vectors.

First, the filter size s has to be decomposed into

$$s = 16(p_k + p_l) + r_k + r_l + q , \tag{5}$$

```
$y_1 = y_2 = y_3 = y_4 = x_1 = x_2 = x_3 = (0, \dots, 0)$        // initialization of vectors
for $u = -2 \dots n/16 - 1$        // start at $-2$ for correct startup
    $x = a(u + 2)$        // read new vector
    if $(q \geq 2)$ $\{y = x;\ x = \max(x, (y_1, x)_{(15,\dots,30)});\ y_1 = y;\}$    // max. with left neighbor
    if $(q \geq 4)$ $\{y = x;\ x = \max(x, (y_2, x)_{(14,\dots,29)});\ y_2 = y;\}$    // max. with 3 left neighbors
    if $(q \geq 8)$ $\{y = x;\ x = \max(x, (y_3, x)_{(12,\dots,27)});\ y_3 = y;\}$    // max. with 7 left neighbors
    if $(q = 16)$ $\{y = x;\ x = \max(x, (y_4, x)_{(8,\dots,23)});\ y_4 = y;\}$    // max. with 15 left neighbors
    // build max. of $q + r_k + r_l$ size intervals
    $a(u) = \max((x_3, x_2, x_1)_{(15-r_k,\dots,30-r_k)}, (x_2, x_1, x)_{(r_l,\dots,15+r_l)})$
    $x_3 = x_2;\ x_2 = x_1;\ x_1 = x$
```

Figure 3: First stage of 1-D SIMD algorithm. $x_\cdot$, $y_\cdot$, and $a(u)$ are vectors of size 16. The handling of data accesses outside of array boundaries is not shown.

where the variables obey the following conditions:

$$q = \begin{cases} 1 & s = 2 \\ 2 & s = 3, 4 \\ 4 & s = 5, 6, 7, 8 \\ 8 & s = 9, \dots, 16 \\ 16 & s > 16 \end{cases} \qquad \begin{array}{l} r_k = k - 16p_k - q \\ r_l = l - 16p_l \\ p_k \geq 0, \quad p_l \geq 0 \\ -16 < r_k \leq 16 \\ 0 \leq r_l < 31 \\ 0 \leq r_k + r_l \leq 16 \end{array} \tag{6}$$

The first stage calculates for each position i the maximum of the interval $[i - q - r_k + 1, i + r_l]$. $a(u)$ shall denote the u-th vector containing 16 bytes. We assume that n is a multiple of 16. Figure 3 shows the algorithm. The first four vector-max-operations calculate the maximum of q-size intervals, where q is a power of two, in $\log_2(q)$ steps. The first operation calculates the maximum of size-2 intervals, the second one calculates the maximum of size-4 intervals based on the size-2 results, and so on. The last max-operation accounts for non-power-of-two size intervals. Variables x_i and y_i are used to pass reusable vectors from one iteration to the next, in order to avoid recalculating or re-reading them from memory.

Expressions of the form $(x, y)_{(s_0,\dots,s_{15})}$ denote shift- and permutation operations, where elements with indices $s_0, \dots, s_{15}$ are selected from the concatenation of the vectors x and y, and assembled into the resulting vector. Such an operation must be implemented in SSE as (at least) two shift- and an or-instruction. A shift instruction shifts one source vector and fills remaining elements with zeros. For two source vectors, two shifts have to be applied and the non-overlapping results have to be combined by an or-instruction. For instance, $(x, y)_{2,\dots,17} = \mathrm{shl}(x, 2) \mid \mathrm{shr}(y, 14)$, which takes three instructions. Each of these instructions will take one clock cycle if the instructions can be scheduled optimally.

Values q, r_k, r_l, p_k and p_l must be computed at compile-time for two reasons. First, shift-instructions need immediate arguments for the shift distance. Second, the if-branches should be resolved at compile-time in order to avoid time consuming jumps in the inner loop. Also, a few iterations at the beginning and the end of the u-loop would access data outside of the

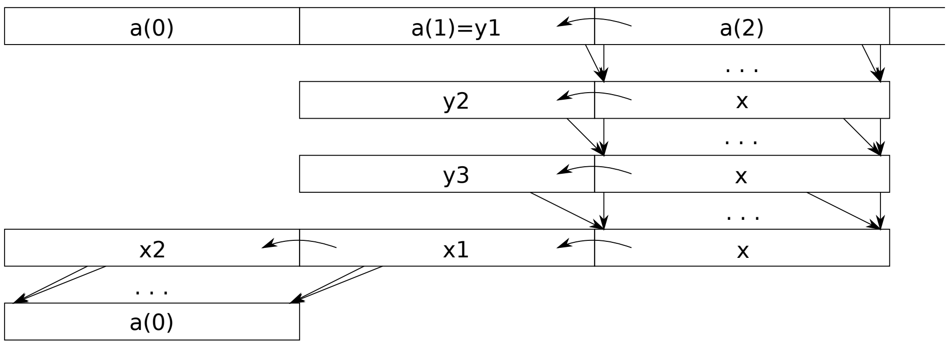

Figure 4: Iteration $u = 0$ of the first stage of the 1-D SIMD algorithm for $k = 3$, $l = 5$, $s = 9$, $p_k = p_l = 0$, $r_k = 3 - 8 = -5$, $r_l = 5$. Each box represents a vector of 16 values. Arched arrows indicate the passing of a vector from a previous iteration.

a-array. To avoid that, these iterations are implemented separately, leaving out unnecessary operations. Figure 4 illustrates an iteration of the 1-D SIMD algorithm.

The second stage of the 1-D SIMD algorithm is analogous to the sequential algorithm in Figure 1. The only differences are that a, b, c, and d are now arrays of vectors, where u and i denote vector-indices, and p_k and p_l substitute k and l.

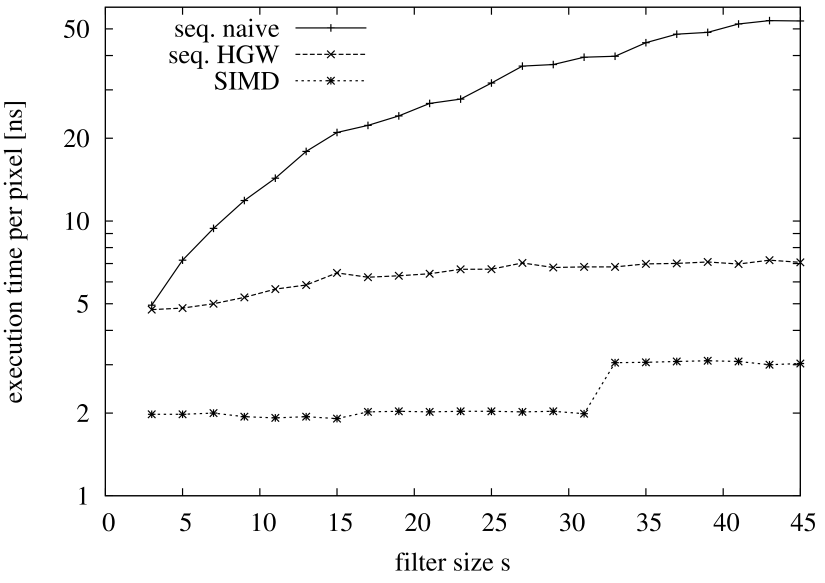

Figure 5: Execution time per pixel of 1-D horizontal algorithms

Figure 5 shows the performance results. In the naive implementation, the execution time grows linearly with the filter size, whereas the other implementations yield almost constant times. The SIMD algorithm has a jump of about one nanosecond at filter size 32. This is the filter size that first requires the second stage of the SIMD algorithm, which takes some extra

```
for $j = 0 \dots m$
    for $u = l \dots n$ step $s - 1$
        [block as in Figure 1 for column $j$]
```

(a) not cache-optimized

```
for $u = l \dots n$ step $s - 1$
    for $j = 0 \dots m$
        [block as in Figure 1 for column $j$]
```

(b) cache-optimized

Figure 6: Loop transposition for cache optimization in the vertical part of the 2-D van Herk/Gil-Werman algorithm.

time. Speedups range between 2.2 and 3.5 compared to the sequential HGW algorithm.

3 2-D SIMD parallelization

The vertical part of the 2-D SIMD parallelization can be implemented simply as the sequential algorithm in Figure 1, where all data variables are vectors of 16-byte-rows. Thus, 16 columns are processed in one run.

However, it is known that such implementations violate the demand for data locality because of step sizes of one or more image rows between subsequent memory accesses. As a consequence, their performance suffers from poor cache usage. Therefore, we suggest a simple modification of the vertical 2-D algorithm, as shown in Figure 6. By transposing the loop for image columns and the u-loop over the $s-1$-size blocks of the HGW algorithm, the 2-D image is processed in rows of height $s-1$, or, to be precise, height $2(s-1)$ with overlap of $s-1$. This improves the data locality significantly, both for the sequential and the SIMD implementation.

Figure 7 shows the performance results. Cache-optimization gains a speedup between 1.5 and 2. The cache-optimized algorithms are a bit more sensitive to the filter size because higher filter sizes increase the height of block rows, which worsens the data locality. The SIMD algorithms exhibit a speedup between 4.5 and 6.

By applying the horizontal 1-D algorithm followed by the vertical implementation, a 2-D max-filter is achieved. The total execution times are shown in Figure 8. We get overall speedups between 2.8 and 3.

4 Conclusion

This work shows that the van Herk/Gil-Werman algorithm for the 1-D maximum filter, as heavily used in mathematical morphology, can be parallelized with short-vector SIMD extensions. The easier case of vertical filtering of 2-D data can be combined with horizontal 1-D filtering to get a SIMD implementation of a 2-D maximum filter.

For 16-fold SIMD, the vertical filtering shows a speedup of about 6 and the horizontal filtering a speedup of about 3. Together, the speedup is about 3, which shows that the horizontal filtering dominates the execution time. Cache-optimization of the vertical algorithm achieves an additional speedup between 1.5 and 2.

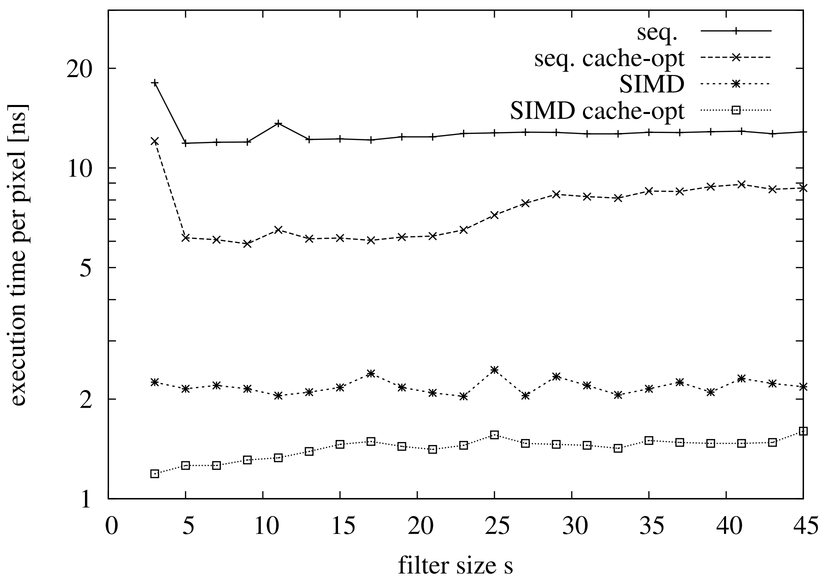

Figure 7: Execution time per pixel of 1-D vertical algorithms

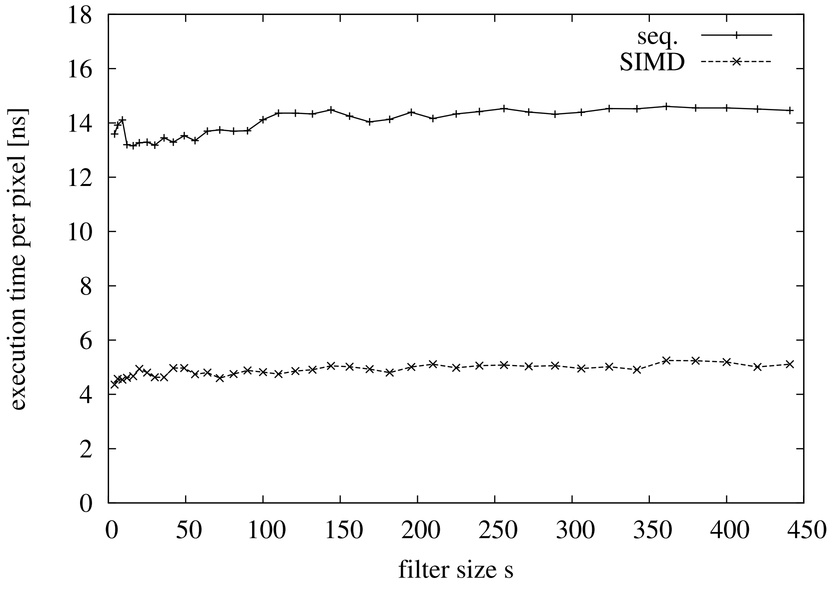

Figure 8: Execution time per pixel of 2-D algorithms

As future work, in order to possibly improve the performance of the implementation, horizontal and vertical cache-optimized filtering could be interleaved, which would increase cache reusage. Furthermore, the transposition approach that enables to use the faster vertical algorithm for horizontal filtering should be compared to the scheme of this work. On-the-fly block transposition using SIMD shuffle instructions could be used to reduce the computation time for the image transposition.

References

[1] Michael R. Bullock, David L. Wang, Scott R. Fairchild, and Tim J. Patterson. Automated training of 3D morphology algorithm for object recognition. In *Automatic Object Recognition IV*, volume 2234 of *Proc. SPIE*, pages 238–251, April 1994.

[2] Yen-Yu Chen and Shen-Chuan Tai. Compressing medical images by morphology filter voting strategy and ringing effect elimination. *Electronic Imaging*, 14:013007–14, March 2005.

[3] P. Deng-Wong, Fulin Cheng, and Anastasios N. Venetsanopoulos. Adaptive morphological filters for color image enhancement. In *Visual Communications and Image Processing*, volume 1818 of *Proc. SPIE*, pages 358–365, November 1992.

[4] Shan Duan and Qian-Qing Qin. Edge detection based on dynamic morphology. In *MIPPR 2005: Image Analysis Techniques*, volume 6044 of *Proc. SPIE*, pages 19–25, 2005.

[5] Charles R. Giardina and Edward R. Dougherty. *Morphological methods in image and signal processing.* Prentice Hall, Englewood Cliffs, NJ, 1988.

[6] Joseph Gil and Michael Werman. Computing 2-D min, median and max filters. *IEEE Trans. on Pattern Analysis and Machine Intelligence*, 15(5):504–507, May 1993.

[7] Xiaojing Jin and Curt H. Davis. New applications for mathematical morphology in urban feature extraction from high-resolution satellite imagery. In *Applications of Digital Image Processing XXVII*, volume 5558 of *Proc. SPIE*, pages 137–148, August 2004.

[8] Ram Saran and Anil K. Sarje. Vectorization of constant-time gray-scale morphological processing algorithm using AltiVec. In *2011 National Conference on Communications (NCC)*, pages 1–5, Bangalore, January 2011.

[9] Marcel van Herk. A fast algorithm for local minimum and maximum filters on rectangular and octagonal kernels. *Pattern Recognition Letters*, 13(7):517–521, July 1992.