Efficient Non-Maximum Suppression

Conference Paper · January 2006

DOI: 10.1109/ICPR.2006.479 · Source: DBLP

Efficient Non-Maximum Suppression

Alexander Neubeck
ETH Zurich, Switzerland
Computer Vision Lab
aneubeck@vision.ee.ethz.ch

Luc Van Gool
ETH Zurich, Switzerland
Computer Vision Lab
vangool@vision.ee.ethz.ch

Abstract

In this work we scrutinize a low level computer vision task – non-maximum suppression (NMS) – which is a crucial preprocessing step in many computer vision applications. Especially in real time scenarios, efficient algorithms for such preprocessing algorithms, which operate on the full image resolution, are important. In the case of NMS, it seems that merely the straightforward implementation or slight improvements are known. We show that these are far from being optimal, and derive several algorithms ranging from easy-to-implement to highly-efficient.

1. Introduction

Non-Maximum Suppression (NMS) can be positively formulated as Local Maximum Search, where a *local maximum* is greater than all its neighbors (excluding itself). For a given n, the *neighborhood* of any pixel consists in the 1D case of the n pixels to its left and right side (referred to as $(2n+1)$-neighborhood) and in the 2D case of the quadratic $(2n+1) \times (2n+1)$ region centered around the pixel under consideration. The same value may appear several times in an image, such that the question arises, which pixel should be suppressed in case of a tie. In practice, either all or all but one are suppressed according to some ordering. Since both variants can be easily integrated into the presented algorithms, we didn't clutter the pseudocode with these steps.

NMS is part of several computer vision algorithms. A good recent example is the extraction of interest points [3, 4, 2]. A point saliency measure is extracted throughout the image, or even the whole scale space, and local maxima are selected. Efficiency is crucial in many of their applications, e.g. tracking, data mining, 3D reconstruction, object recognition, and texture analysis.

Closely related to NMS is the maximum filter which computes the maximum value of each pixel's neighborhood. An efficient, mask-size independent algorithm for the maximum filter was proposed by Gil and Werman [1]. Given the maximum filter response in each pixel, the NMS reduces to an additional comparison of each pixel value with its maximum neighbor. Although this scheme can outperform a straightforward implementation of NMS, it extracts more information than needed by NMS.

We will present two concepts which enable the efficient computation of NMS: *partial maximum sequences* (PMS) and block-wise processing. The partial maximum sequence is described in Section 2, whereas the idea of the block-wise processing is given in Section 3. We print for both algorithms pseudo-code in order to ease their implementation. In the second part of Section 3, the two concepts are combined into an optimal algorithm. Since this algorithm is merely of theoretical interest due to its complicated control flow, this last part is merely sketched to provide the important steps to an inclined reader. The analysis results of other combinations are summarized in Fig. 3 without derivation.

In order to analyze the complexity of each algorithm, we count the number of input dependent comparisons in the worst case and average case. For the latter, assumptions about the data distribution are needed. For simplicity, we consider an image as a random permutation of unique values. This assumption approximates the data distribution for small neighborhoods sufficiently well. Experimentally, we got a deviation from the number of comparisons of at most 1% for a 21-neighborhood and at most 10% for a 21×21-neighborhood between a random permutation of unique values and a random image with 256 gray scales.

2. NMS in 1 Dimension

2.1. Straightforward Implementation

The straightforward implementation of NMS consists of two nested loops, where the outer loop iterates over all pixels and the inner loop tests a candidate of the outer loop against all its neighbors. As soon as a neighbor intensity exceeds the current candidate, the inner loop is aborted. Obviously, the algorithm needs $2n$ comparisons per pixel without the early abort. In the worst-case, the abort cannot decrease the complexity. This can be seen by applying the algorithm to an intensity trend. Since pixels on one side are always smaller than the candidate pixel, the inner loop aborts at the $(n+1)$-th iteration leading to exactly $n+1$ comparisons per pixel. Hence, the worst-case number of comparisons per pixels is $O(n)$.

In order to analyze the average case complexity, the probability $p(i)$ that the inner loop breaks at its i-th iteration is needed. For it, two conditions must be satisfied. First, the i-th neighbor must be the largest among the first i neighbors plus the current candidate, which happens with probability $\frac{1}{i+1}$. Second, the loop must not have been aborted before the i-th iteration, i.e. the current candidate must be larger than the already tested neighbors, which happens with probability $\frac{1}{i}$. Altogether, we get:

$$p(i) = \frac{1}{(i+1)\cdot i}. \tag{1}$$

In each iteration exactly one comparison between the candidate pixel and the i-th neighbor takes place, which sums up to i comparisons until the i-th iteration. Thus, the expected number of comparisons per pixel (CpP) is

$$E(\text{CpP}) = \frac{2n}{2n+1} + \sum_{i=1}^{2n} p(i)\cdot i \tag{2}$$

$$= \frac{2n}{2n+1} + \sum_{i=1}^{2n} \frac{1}{i+1} \approx 1 + \ln(2n), \tag{3}$$

where n is the number of left and right neighbors, and the first term captures the case when the candidate pixel is a local maximum.

2.2. 3-Neighborhood

In order to get a better insight into NMS, we consider the smallest feasible neighborhood next, where the central pixel needs only be compared to its direct neighbors. **Algorithm 1** reaches the theoretical optimum of 1 comparison per pixel in the worst case, whereas the straightforward solution requires already 1.5 comparisons per pixel on average. Amazingly, the expected number of comparisons performed by **Algorithm 1** is with 0.815 less than 1.

Algorithm 1 starts with the left-most candidate of the input sequence $I[0], \ldots, I[W-1]$ located at $i = 1$ (line 1). If the current candidate i exceeds its right and left neighbors, a local maximum has been found (lines 3–5). Since it is already known that pixel $i+1$ is smaller than its left neighbor i, it cannot be an local maximum. Hence, processing can directly continue with pixel $i+2$ (line 12). If the current candidate i fails to exceed its right neighbor, this right neighbor stands in as candidate (line 7). By applying this rule iteratively, the candidate climbs up a monotonically increasing subsequence (line 8–9). When the top is reached,

```
1  $i \leftarrow 1$;
2  while $i+1 < W$ do
3      if $I[i] > I[i+1]$ then
4          if $I[i] >= I[i-1]$ then
5              MaximumAt($i$);
6      else
7          $i \leftarrow i+1$;
8          while $i+1 < W$ AND $I[i] \leq I[i+1]$ do
9              $i \leftarrow i+1$;
10         if $i+1 < W$ then
11             MaximumAt($i$);
12     $i \leftarrow i+2$;
```

Algorithm 1: 1D NMS for 3-Neighborhood

i.e. the right neighbor is even less than the candidate i, a maximum is found (line 11). One should notice that i is automatically greater than its left neighbor in this case.

The `while`-loop of **Algorithm 1** partitions the input sequence into subsequences. A subsequence begins with the position to which i points at the beginning of an iteration and ends just before the beginning of the next subsequence. In order to analyze the complexity of **Algorithm 1**, the probability $p(B)$ that a subsequence of size B occurs and the number of comparisons $c(B)$ performed in order to find the local maximum within such a subsequence are computed. In the first case (lines 3–5), the subsequence has a fixed size of 2 elements for which 2 comparisons are performed. In the other cases (lines 7–11), the algorithm increases i for each comparison (line 2&7 resp. 8&9). When the last test fails, i is incremented by 2 (line 8&12). Hence, the number of comparisons $c(B)$ is

$$c(B) = \begin{cases} 2 & \text{if } B = 2 \\ B-1 & \text{if } B > 2 \end{cases}. \tag{4}$$

The worst-case occurs for $B = 2$, where one comparison per pixel is performed. For the average case complexity, additionally the probability $p(B)$ is needed that a block of a certain length B occurs. Since all occurring subsequences are monotonically increasing until the element before the end, the last element can take any value except the largest one leading to $B-1$ possibilities. Since the remaining $B-1$ elements must be in ascending order, there is no choice left. Thus, $p(B)$ is the ratio of these $B-1$ occurring permutations and all possible permutations:

$$p(B) = \frac{B-1}{B!}. \tag{5}$$

Given the probability and the costs of each subsequence, the expected number of comparisons per pixel can be computed as the expected number of comparisons per subsequence divided by the expected subsequence length:

$$E(\mathrm{CpP}) = \frac{\sum_{B=2}^{\infty} p(B)c(B)}{\sum_{B=2}^{\infty} p(B)B} = \frac{1+\sum_{B=3}^{\infty}\frac{(B-1)^2}{B!}}{\sum_{B=2}^{\infty}\frac{(B-1)\cdot B}{B!}} \quad (6)$$

$$= \frac{1+\sum_{B=3}^{\infty}\frac{B^2}{B!} - 2\frac{B}{B!} + \frac{1}{B!}}{e} = 1 - \frac{1}{2e} \approx 0.816. \quad (7)$$

2.3. Dynamic Block Algorithm

The generalization of **Algorithm 1** to the n-neighborhood is printed in **Algorithm 3**. As before, the `while`-loop partitions the input sequence into subsequences – referred to as *dynamic blocks*. At each time of the construction, a block satisfies the following invariance: within a block there is exactly one local maximum i where only neighbors within the block are considered and, consequently, neighbors outside the block are ignored. The algorithm keeps track of this truncated local maximum i within the current block and extends the block to the right, until the block contains the full right neighborhood of i. Each time the block grows, further neighbors of i have to be tested. Thereby, either i passes all the tests – and the block stops growing – or i's largest neighbor exceeds i and becomes the block's new local maximum.

If a full-grown block, ranging from a to b, contains the left neighborhood of its local maximum i (i.e. $a \leq i - n$), then i is obviously a local maximum of the whole sequence without further testing. Otherwise ($a > i - n$), it must be tested against the partially excluded left neighborhood ranging from $i - n$ to $a - 1$. Instead of testing each of these elements one by one, we present a way, where the already performed comparisons of the previous block can be used, to get the testing to the left done with at most two additional comparisons, i.e. independent on the neighborhood size.

```
CompPartialMax(from, to)
    $pmax[to] \leftarrow I[to]$;
    $best \leftarrow to$;
    while $to > from$ do
        $to \leftarrow to - 1$;
        if $I[to] \leq I[best]$ then
            $pmax[to] \leftarrow I[best]$;
        else
            $pmax[to] \leftarrow I[to]$;
            $best \leftarrow to$;
    return $best$;
```

Algorithm 2: Compute Partial Maximum

To achieve this goal, we introduce the concept of the *partial maximum sequence* ($pmax$), which we define for a subsequence $I[l], \cdots, I[r]$ of the input image I as:

$$pmax[i] = \max\{I[i], I[i+1], \cdots, I[r-1], I[r]\}. \quad (8)$$

It is straightforward to see that the partial maximum sequence can be computed with $r - l$ comparisons from right to left (see **Algorithm 2**). This kind of query is what we need to efficiently get the maximum of the excluded left neighborhood. However, the block size is not known in advance and therefore, the partial maximum sequence for the whole block can not be incrementally computed while the block grows. Instead, the partial maximum sequence of the appendix by which a block grows is computed. Since each appendix ranges from the old block end b plus 1 to i's rightmost neighbor $i + n$, it is guaranteed that two consecutive appendices span at least $n+1$ elements. Since the left neighborhood of any element of the next block extends into the previous block by at most n elements, the partial maximum sequences of the last two appendices enable the left testing with at most two comparisons. The separator of the two appendices is traced in the variable $chkpt$ of the pseudocode.

```
$i \leftarrow n$;
CompPartialMax($0, i-1$);
$chkpt \leftarrow -1$;
while $i < W - 2n$ do
    $j \leftarrow$ CompPartialMax($i, i+n$);
    $k \leftarrow$ CompPartialMax($i+n+1, j+n$);
    if $i == j$ OR $I[j] > I[k]$ then
        if ($chkpt \leq j - n$ OR $I[j] \geq pmax[chkpt]$)
        AND ($j - n = i$ OR $I[j] \geq pmax[j-n]$) then
            MaximumAt($j$);
        if $i < j$ then
            $chkpt \leftarrow i + n + 1$;
        $i \leftarrow j + n + 1$;
    else
        $i \leftarrow k$;
        $chkpt \leftarrow j + n + 1$;
        while $i < W - n$ do
            $j \leftarrow$ CompPartialMax($chkpt, i+n$);
            if $I[i] > I[j]$ then
                MaximumAt($i$);
                $i \leftarrow i + n - 1$;
                break;
            else
                $chkpt \leftarrow i + n - 1$;
                $i = j$;
```

Algorithm 3: 1D NMS for $(2n+1)$-Neighborhood

The cost function $c(B)$ counts the comparisons within a block of size B and the eventual comparisons into the previous block. The local maximum of a full-grown block is located n elements before the block's end, such that the left neighborhood of the local maximum shares $\max\{2n +$

$1-B,0\}$ elements with the previous block. Exploiting the partial maximum sequences, at most 2 of these elements need to be compared with the local maximum. Together with the $B-1$ comparisons, to find the local maximum within a block, the worst-case costs are:

$$c(B) = \min\{\max\{2n+1-B, 0\}, 2\} + B - 1. \quad (9)$$

The costs per pixel decrease down to 1 in the range $n+1 \leq B \leq 2n$ and stays below 1 for greater B. Thus, splitting the input sequence in blocks of minimal size ($B = n+1$) seems to be the worst-case. However, a block of size $n+1$ has only a single partial maximum sequence, reducing the worst-case costs of the next block by one. With blocks of size $n+2$ ($n > 2$) this behavior can be avoided and a corresponding sequence (which exists) requires the worst-case per pixel cost of $1 + \frac{1}{n+2}$.

3. NMS in 2 Dimensions

The difficulty of 2D NMS is its non-separability. Therefore, another concept is needed to get an efficient solution. We start with the introduction of such a concept and its realization and then fill in ideas of the 1D case until we end up with a highly efficient solution.

3.1. $(2n+1) \times (2n+1)$-Block Algorithm

We observe that two local maxima are at least $n+1$ pixels in each direction apart. Vice versa, within each block of size $(n+1) \times (n+1)$ there can be at most one local maximum. Thus, the algorithm partitions the input image into such blocks and searches within each block for the greatest element, which is its only possible local maximum candidate. Then, the full neighborhood of this candidate is tested. Hereby, elements of the block itself can be skipped, because they are by construction already smaller than the candidate. The pseudocode is shown in **Algorithm 4**.

```
forall $(i,j) \in \{n, 2n+1, \dots\}^2 \cap [0, W-n] \times [0, H-n]$ do
    $(mi, mj) \leftarrow (i, j)$;
    forall $(i2, j2) \in [i, i+n] \times [j, j+n]$ do
        if $img(i2, j2) > img(mi, mj)$ then
            $(mi, mj) \leftarrow (i2, j2)$;
    forall $(i2, j2) \in [mi-n, mi+n] \times [mj-n, mj+n] - [i, i+n] \times [j, j+n]$ do
        if $img(i2, j2) > img(mi, mj)$ then
            goto failed;
    MaximumAt (mi,mj);
    failed:
```

Algorithm 4: 2D $(n+1) \times (n+1)$-Block NMS

The worst-case occurs, when a block's candidate is in fact a local maximum of the input image, in which case all the neighbors have to be tested. The algorithm does this with $(2n+1)^2 - 1$ comparisons per block, which limits the number of comparisons per pixel by

$$\mathrm{CpP} \leq \frac{(2n+1)^2 - 1}{(n+1)^2} = 4 - \frac{4}{n+1}. \quad (10)$$

The average case analysis is similar to the one of the straightforward implementation, except that the maximum can be positioned anywhere within the block and that the testing starts with the $(n+1)^2$-th neighbor instead of the first:

$$E(\mathrm{CpP}) = \frac{(2n+1)^2 - 1}{(2n+1)^2} + \sum_{i=(n+1)^2+1}^{(2n+1)^2} \frac{1}{i} \quad (11)$$

$$\approx 1 - \frac{1}{(2n+1)^2} + 2\ln\left(2 - \frac{1}{n+1}\right) \leq 1 + \ln 4. \quad (12)$$

Thus, the algorithm easily outperforms the straightforward implementation because the asymptotic behavior is already independent of the neighborhood size: in the average case the number of comparisons per pixel is limited by 2.39 and in the worst-case by 4. Although this is still far from being optimal, the algorithm requires no additional memory and each block can be processed independently.

3.2. Partial Maxima in 2D

The main disadvantage of this first 2D NMS algorithm is that no information between blocks is shared. By incorporating the previously introduced concept of partial maxima, the worst-case complexity can be improved to $2 + O(1/n)$.

Since the algorithm can compute the maximum within each block in arbitrary order, one can first compute the maximum of each column within a block and then compute the maximum over all column maxima. These column maxima can then be reused in the testing phase, when the left and right parts of the full neighborhood must be tested (see black regions in Fig. 1). As a consequence, the total number of comparisons is reduced by almost one quarter.

Furthermore, the maximum of the upper half of a column can be computed with a partial maximum sequence from top to bottom and the maximum of the lower half with a partial maximum sequence from bottom to top (indicated by the arrows in Fig. 1). This allows to reuse computations also for the upper and lower neighborhood regions (see dark gray regions). In the worst-case, there is either an upper or a lower region, such that only half of this region is covered by reusable partial maximum sequences.

The number of comparisons for the block itself, the

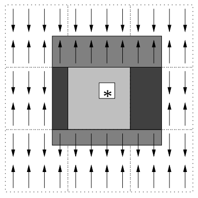

Figure 1. Neighborhood Partitioning of a Local Maximum Candidate.

left/right and top/bottom regions sums up to at most

$$\mathrm{CpP} \le \frac{[(n+1)^2-1]+[2(n+1)]+[(2n+1)\lfloor\frac{n}{2}+1\rfloor]}{(n+1)^2} \tag{13}$$

$$\le 2+\frac{2.5}{n+1}+\frac{0.5}{(n+1)^2}. \tag{14}$$

3.3. Stripe Algorithm

With a fully sequential algorithm, it is possible to drop the worst-case complexity even down to $1+O(\frac{1}{n})$. However, due to its complicated control flow, this algorithm is more of theoretical than of practical interest.

Instead of the block processing of the two previous 2D NMS algorithms, this algorithm partitions the input image into stripes of height $n+1$. The stripes are processed from top to bottom. As before, for each column of such a stripe the maximum is computed with $\frac{n}{n+1}$ comparisons per pixel. The 1D NMS algorithm can be applied to the resulting 1D sequence of maxima. Since the 1D sequence contains only $\frac{1}{n+1}$ many elements compared to the full stripe, the maximum computation of the 1D sequence has at most $\frac{1}{n+1}\cdot\frac{n+3}{n+2}$ comparisons per pixel.

By construction, the local maxima returned by the 1D NMS algorithm passed already the $(2n+1)\times(n+1)$ elements of their neighborhood within the stripe. In a second step, they are tested against the remaining $n\times(n+1)$ elements above and below the stripe. For the part below, partial maximum sequences are columnwise computed and the largest element of each column is tested against the local maximum candidate. When the algorithm moves on to the next stripe, it only needs to compute the partial maximum sequence for the remainder of each column, such that the anticipated partial maximum computations are for free.

Unfortunately, it is slightly more complicated. The bottom neighborhood regions of two consecutive local maxima can overlap (see Fig. 2), such that the corresponding columns need to be split into three partial maximum sequences instead of two. In practice, one would process the partial maxima within a stripe from top to bottom and remember the already processed part of each column.

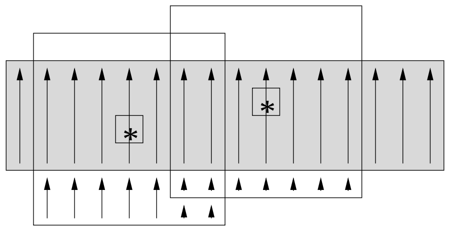

Figure 2. Stripe Algorithm: bottom neighborhoods overlap leading to at most 3 partial maximum sequences per column.

For the neighborhood elements above the stripe, the partial maximum sequences are already computed. Since a column is split into at most three such sequences, at most three comparisons are needed to test the local maximum against a column of the upper neighborhood. In total, the upper bound of per pixel comparisons is

$$\mathrm{CpP} < 1+\frac{8}{n+1}-\frac{3}{(n+1)^2}. \tag{15}$$

4. Results

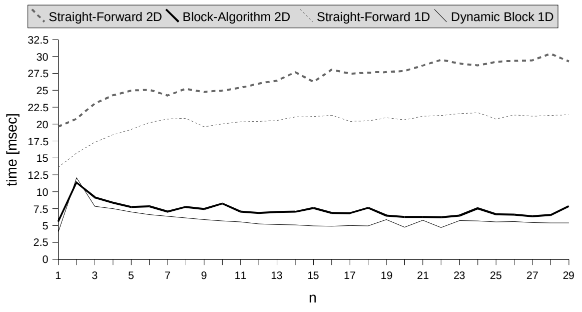

Figure 4. Computation Time in Milliseconds for one Million Pixels.

In Fig. 4, the running time of the straightforward implementation of 2D NMS and **Algorithm 4** are shown. We have run both algorithms on a Pentium 4 with 3.0 GHz. We experienced that `gcc-3.4.1` could not fully optimize the generic algorithms, although they were written with C++ templates. For instance a handcrafted version for the 3×3

Algorithm	#Processors per Pixel	Memory	Worst-Case Complexity per Pixel	Avg.-Case Complexity per Pixel
1D Straightforward	1	$O(1)$	$O(n)$	$\sim 1 + \ln(n) + \ln(2)$
1D $(n+1)$-Block	$\frac{1}{n+1}$	$O(1)$	$2 - \frac{1}{n+1}$	$\sim 1 + \ln(2 - \frac{1}{n+1}) < 1.69$
1D Dynamic Block	0	$O(n)$	$1 + \frac{1}{n+2}$	~ 1
2D Straightforward	1	$O(1)$	$O(n^2)$	$\sim 1 + 2\ln(n) + \ln(2)$
2D $(n+1) \times (n+1)$-Block	$\frac{1}{n^2}$	$O(1)$	$4 - \frac{4}{n+1}$	$\sim 1 + 2\ln(2 - \frac{1}{n+1}) < 2.39$
2D $(n+1)$-Stripe	$\frac{1}{n \cdot width}$	$O(n)$	$3 - O(\frac{1}{n+1})$	$\sim 1.5 + \frac{3}{4n+2} + \ln(2 + \frac{1}{n})$
2D $(n+1) \times (n+1)$-Block + Partial-Max	$\frac{1}{n^2}$	$O(pixels)$	$2 + O(\frac{1}{n+1})$	—
2D $(n+1)$-Stripe + Partial-Max	0	$O(width)$	$1 + O(\frac{1}{n+1})$	~ 1

Figure 3. Summary of Possible Variations for 1D/2D NMS.

neighborhood outperformed the generic template version by a factor of 2. The straightforward implementation is about 4 to 5 times slower than the proposed method.

The dramatically increasing speed of GPUs and their parallel processing structure make them an interesting platform for computer vision algorithms. Thus, we have adopted **Algorithm 4** for the GPU, where each fragment unit processes one $(n + 1) \times (n + 1)$-block, writing either the position of the local maximum or an invalid position to the output buffer. Due to the block structure, only a size of $\frac{W}{n+1} \times \frac{H}{n+1}$ is required for the output buffer reducing the data transfer back to the CPU significantly. On an NVidia 6800 graphics card, we measured for the 3×3 neighborhood size 2.2 milliseconds pure processing time, when applied to a 1000×1000 image. The back-transfer of the quarter image took about 1 millisecond. Compared to the 5.6 milliseconds of the CPU implementation, the GPU can outperform the CPU by a factor of about 2.5, if subsequent processing stages can also be performed on the graphics card.

We also tested a variation of **Algorithm 4**, where the columnwise maxima of left and right blocks are utilized. To share the maximum information between blocks, a two pass algorithm is needed for the GPU. The first pass, which computes the column maxima, takes about 1 millisecond, whereas the second pass takes about 2 milliseconds. Although this variation performs the NMS computations with one comparison less, it does not compensate for the more complicated control flow, at least for the small 3-neighborhood. We observe a similar behavior on the CPU, where this variation takes about 6.6 milliseconds.

In Fig. 4, the performance of **Algorithm 3** and the straightforward implementation of 1D NMS are drawn. Again, the proposed method is about 3 to 4 times faster than the straightforward implementation.

5. Conclusions

We have proposed several ways to speed up 1D and 2D NMS. Thereby, it was not only possible to get rid of the dependency on the neighborhood size, but to push the number of comparisons down to almost 1 comparison per pixel in the worst-case. It is also remarkable that the average-case complexity can drop below 1 comparison for small neighborhood sizes.

In some applications NMS of higher dimensional input data is needed. Since the presented concepts generalize to higher dimensions, also these applications benefit from the presented work.

Fig. 3 summarizes the different possible variations for 1D/2D NMS. There, we also give the degree of parallelism and memory requirements. The former is expressed by the number of processors per pixel, where 1 means that all pixels can be processed in parallel and 0 means that no parallelization can take place.

Acknowledgment: Support by the EC Network of Excellence PASCAL and the Swiss SNF NCCR project IM2 is gratefully acknowledged.

References

[1] J. Gil and M. Werman. Computing 2-d min, median, and max filters. *IEEE Trans. Pattern Anal. Mach. Intell.*, 15(5):504–507, 1993.

[2] D. G. Lowe. Distinctive image features from scale-invariant keypoints. *IJCV*, 60(2):91–110, 2004.

[3] K. Mikolajczyk and C. Schmid. Scale & affine invariant interest point detectors. *IJCV*, 60(1):63–86, 2004.

[4] T. Tuytelaars and L. V. Gool. Matching widely separated views based on affine invariant regions. *IJCV*, 59(1):61–85, 2004.