Efficient filter computation with symmetric matrix kernels

Manfred Kopp
Institute of Computer Graphics, Visualization and Animation Group
Technical University of Vienna,
Karlsplatz 13/186-2, A-1040 Wien
m.kopp@ieee.org , http://www.cg.tuwien.ac.at/~kopp/

Abstract

This paper presents an algorithm for filter calculations using symmetric matrix kernels. This algorithm outperforms traditional methods for kernels larger than or equal to 5x5 on machines based on RISC designs, where the time needed to calculate an addition equals the time needed for a multiplication. The algorithm is based on a decomposition of the kernel matrix into several kernel matrices of decreasing size, which can be computed very fast because of spatial coherence. A comparison with traditional methods shows the efficiency of the presented approach.

Keywords: image processing, anti-aliasing, filtering, symmetric matrix kernels, spatial coherence

1 Introduction

Filtering using matrix kernels is a common technique in image processing [Russ92][Habe89] and anti-aliasing [Fole90][Watt92]. The filtered pixel is the weighted sum of the original and its neighbouring pixels within a rectangle defined by a maximum distance in x- and y-direction which is determined by the kernel size. The weights form a (M x N) matrix called the kernel matrix. The dimensions M and N of the kernel matrix are usually odd, which guarantees that the maximum distance of accounted neighbouring pixels from the original pixel in one dimension is the same in positive and in negative direction. Equation (1) describes the filter computation using a kernel matrix:

$$\overline{p}_{x,y} = \sum_{i=1}^{M}\sum_{j=1}^{N} K_{i,j} \cdot p_{x+i-m,y+j-n}; \quad m = \frac{M+1}{2}; \quad n = \frac{N+1}{2}; \qquad (1)$$

$\overline{p}_{x,y}$ ... filtered pixel at position (x,y)

$p_{x,y}$ ... original pixel at position (x,y)

K ... (M x N) kernel matrix

According to Equation (1), a straight forward implementation of a general kernel filter would require (M·N) multiplications and (M·N-1) additions for each filtered pixel. Since a picture usually contains several hundreds of thousands to several millions of pixels, efficient filtering is important to get the results in resonable time.

In this paper we want to cope with kernel filters, which are symmetric with respect to the x-axis and the y-axis. Given a (M x N) kernel matrix K, K is symmetric with respect to the x-axis, iff

$$K_{i,j} = K_{M-i+1,j}; \quad 1 \le i \le M; \quad 1 \le j \le N \qquad (2a)$$

and symmetric with respect to the y-axis, iff

$$K_{i,j} = K_{i,N-j+1}; \quad 1 \le i \le M; \quad 1 \le j \le N \qquad (2b)$$

Most of the frequently used filter kernels are symmetric with respect to the x- and y-axis, including Gauss kernels [Russ92], Box kernels [Watt92], and Bartlett kernels [Crow81].

A simple improvement for symmetric kernels to the general kernel filter algorithm could make use of the symmetry to reduce the number of multiplications: The four symmetric pixels are added and the common weight $K_{i,j} = K_{M-i+1,j} = K_{M-i+1,N-j+1} = K_{i,N-j+1}$ is multiplied afterwards. This improved method needs only 1/4 (M·N) multiplications if M and N are even and a little bit more, if M or N are odd, but still (M·N-1) additions.

2 Basic algorithm

2.1 Concept of our proposed algorithm

Our goal is to reduce the number of operations, consisting of additions and multiplications, needed in the filtering. This goal will be reached through the reduction of additions, but the number of multiplications will be larger than or equal to those in traditional methods. Therefore we are assuming, that the calculation time for a multiplication equals the calculation time for an addition, which is usually true on RISC based machines.

The efficiency of our algorithm is based on spatial coherence [Gröl93], the use of information gained from the filtering of neighbouring pixels for the actual filtering.

To achieve the goal we will recursively decompose the kernel matrix into the sum of a smaller kernel matrix and two less computational expensive kernel matrices of the same size. Equation (3) shows this decomposition of the kernel into three simpler to compute kernels. From (1) it can be derived, that the sum of the filtering with the kernels ${}^{i}F$, i=1,2,...,anz , is equal to the filtering with kernel $\sum_{i=1}^{anz} {}^{i}F$.

Chapter 2.2 describes our decomposition scheme, chapter 2.3 explains the filter algorithm based on the decomposition and appendix B shows an example of the proposed algorithm with a 5x5 kernel filter.

$$
{}^{k}K = \begin{bmatrix} {}^{k}K_{1,1} & {}^{k}K_{1,2} & \cdots & {}^{k}K_{1,N-1} & {}^{k}K_{1,N} \\ {}^{k}K_{2,1} & {}^{k}K_{2,2} & & {}^{k}K_{2,N-1} & {}^{k}K_{2,N} \\ \vdots & & \cdots & & \vdots \\ {}^{k}K_{M-1,1} & {}^{k}K_{M-1,2} & & {}^{k}K_{M-1,N-1} & {}^{k}K_{M-1,N} \\ {}^{k}K_{M,1} & {}^{k}K_{M,2} & \cdots & {}^{k}K_{M,N-1} & {}^{k}K_{M,N} \end{bmatrix} = {}^{k}A + {}^{k}B + {}^{k+1}K = \tag{3}
$$

$$
\begin{bmatrix} {}^{k}A_{1,1} & {}^{k}A_{1,2} & \cdots & {}^{k}A_{1,N-1} & {}^{k}A_{1,N} \\ {}^{k}A_{2,1} & {}^{k}A_{2,2} & & {}^{k}A_{2,N-1} & {}^{k}A_{2,N} \\ \vdots & & \cdots & & \vdots \\ {}^{k}A_{M-1,1} & {}^{k}A_{M-1,2} & & {}^{k}A_{M-1,N-1} & {}^{k}A_{M-1,N} \\ {}^{k}A_{M,1} & {}^{k}A_{M,2} & \cdots & {}^{k}A_{M,N-1} & {}^{k}A_{M,N} \end{bmatrix} + \begin{bmatrix} S_k & 0 & \cdots & 0 & S_k \\ 0 & 0 & & 0 & 0 \\ \vdots & & \ldots & & \vdots \\ 0 & 0 & & 0 & 0 \\ S_k & 0 & \ldots & 0 & S_k \end{bmatrix} + \begin{bmatrix} 0 & 0 & \cdots & 0 & 0 \\ 0 & {}^{k+1}K_{1,1} & \cdots & {}^{k+1}K_{1,N-2} & 0 \\ \vdots & \vdots & \cdots & \vdots & \vdots \\ 0 & {}^{k+1}K_{M-2,1} & \cdots & {}^{k+1}K_{M-2,N-2} & 0 \\ 0 & 0 & \cdots & 0 & 0 \end{bmatrix}
$$

2.2 Matrix decomposition

The first step in the algorithm is a decomposition of the (M x N) kernel matrix ${}^{1}K$ in a scalar value S_1, a (M x N) matrix ${}^{1}A$ and a ((M-2)x(N-2)) matrix ${}^{2}K$. S_1, ${}^{1}A$ and ${}^{2}K$ are computed according to the following equations with k=1:

$$S_k = {}^{k}K_{1,1} - 1 \tag{4}$$

$$ {}^{k}A_{1,1} = {}^{k}A_{M-2\cdot k+2,1} = {}^{k}A_{1,N-2\cdot k+2} = {}^{k}A_{M-2\cdot k+2,N-2\cdot k+2} = 1 \tag{5a}$$

$$ {}^{k}A_{i,1} = {}^{k}A_{i,N} = {}^{k}K_{i,1}; \quad 1 < i < M - 2\cdot k + 2 \tag{5b}$$

$$ {}^{k}A_{1,j} = {}^{k}A_{M,j} = {}^{k}K_{1,j}; \quad 1 < j < N - 2\cdot k + 2 \tag{5c}$$

$$ {}^{k}A_{i,j} = {}^{k}A_{i,1} \cdot {}^{k}A_{1,j}; \quad 1 < i < M - 2\cdot k + 2; \quad 1 < j < N - 2\cdot k + 2 \tag{5d}$$

$$ {}^{k+1}K_{i,j} = {}^{k}K_{i+1,j+1} - {}^{k}A_{i+1,j+1}; \quad 1 \le i \le M - 2\cdot k; \quad 1 \le j \le N - 2\cdot k \tag{6}$$

Furthermore ${}^{1}A$ can be written as the product of a vertical vector ${}^{1}v$ and a horizontal vector ${}^{1}h$:

$$ {}^{k}h = {}^{k}A_{1,j}; \quad {}^{k}v = {}^{k}A_{i,1}; \quad {}^{k}A = {}^{k}v \cdot {}^{k}h; \tag{7}$$
$$ 1 \le i \le M - 2\cdot k + 2; \quad 1 \le j \le N - 2\cdot k + 2 $$

With this recursive calculation scheme, S_k, ${}^{k}v$, ${}^{k}h$ and ${}^{k+1}K$ are calculated from ${}^{k}K$, k≥1, until one of the dimensions of the matrix ${}^{k+1}K$ reaches one or two. The result of this recursive decomposition are q scalar values S_k, q vertical vectors ${}^{k}v$, and q horizontal vectors ${}^{k}h$ and the remaining kernel matrix ${}^{q+1}K$ with $q = \min\left(\left\lfloor \frac{M-1}{2} \right\rfloor, \left\lfloor \frac{N-1}{2} \right\rfloor\right)$.

2.3 Filter computations with the components

First we can sum up the pixels affected by the scalar values S_i, 1≤i≤q. This leads to the first intermediate sum:

$$\mathrm{sum}_1 = \sum_{i=1}^{q} S_i \cdot \left(\hat{p}_{i,i} + \hat{p}_{M-i+1,i} + \hat{p}_{M-i+1,N-i+1} + \hat{p}_{i,N-i+1}\right) \tag{8}$$

$$\hat{p}_{i,j} = p_{x+i-m,y+i-n}; \quad 1 \le i \le M; \quad 1 \le j \le N$$

The next step is to compute the kernel filtering with the kernel ${}^{k}A$, $1 \le k \le q$ and add these filterings to the intermediate sum. But instead of applying the whole matrix ${}^{k}A$ on the window of neighbouring pixels, we apply the horizontal filter ${}^{k}h$ on each line of this window, followed by the vertical filter ${}^{k}v$ applied on the result of the horizontal filtering. From (1) and (7) it can be proved, that this method produces the same results as applying ${}^{k}A$ itself. Note that because of the symmetry of ${}^{k}v$ and ${}^{k}h$, only one multiplication has to be performed in the filtering for each pair of opposite pixels. The kernel calculations with the horizontal filters can be reused in filtering of the neighbouring vertical pixels. Therefore only one new horizontal filter has to be applied for the calculation of one kernel ${}^{k}A$ on a new pixel window except in the calculation of the first lines of the picture.

Finally the kernel matrix filtering with ${}^{q+1}K$ is computed and added to the previous sum using the conventional method with symmetry enhancements described in chapter 1. Note that the size of the remaining kernel ${}^{q+1}K$ is less than or equal to ((M-N+2) x 2) or $(2 \text{ x } (N - M + 2)$, respectively. For square kernels the remaining kernel ${}^{q+1}K$ is (1x1) or (2x2), respectively.

3 Optimizations

One obvious optimization is to stop the recursive decomposition, if the last computed intermediate kernel ${}^{k}K$ equals the null matrix O. Note that the Box kernel and the Bartlett kernel need only one decomposition step, because these kernels can directly be written as the product of a vertical and a horizontal vector with integer values. Also obvious, scalar values S_k can be ignored, if they are 0.

Our algorithm outperforms traditional ones, if the size of the kernel is big and is beaten, if the size is small. Therefore one optimization is to alter the decomposition scheme to stop decomposing, if the remaining kernel ${}^{k+1}K$ can be computed faster with a traditional algorithm.

Often integer arithmetic is used for the calculation of the filtering with integer values in the kernel and the result is divided by a scaling value. This is because the frame buffer only deals with integer color values and the filtering is made for direct display. Therefore floating point arithmetic would cause some additional conversion overhead. If floating point arithmetic is used because of the needed precision of the result, the decomposition of the kernel matrix can be modified to get rid of most of the scalar values S_k:

```
IF ($^kK_{1,1} = 0$) DO
    calculate decomposition according to (4),(5),(6),(7)
ELSE DO
    $S_k = 0$                                                        (9)
    $^kA_{1,1} = {}^kA_{M-2\cdot k+2,1} = {}^kA_{1,N-2\cdot k+2} = {}^kA_{M-2\cdot k+2,N-2\cdot k+2} = {}^kK_{1,1}$    (10)
    calculate rest of matrix $^kA$ and $^{k+1}K$ according to (5b)-(5d), (6)
    $^kh = {}^kA_{1,j};\quad {}^kv = \frac{^kA_{i,1}}{^kA_{1,1}};\ 1 \le i \le M-2\cdot k+2;\quad 1 \le j \le N-2\cdot k+2$    (11)
```

With this decomposition scheme, S_k can only be -1 or 0, therefore most additions in (8) can be skipped, because S_k=0. Therefore (8) can be replaced with:

$$\mathrm{sum}_1 = -\sum_{S_i \neq 0; 1 \le i \le q} \left(\hat{p}_{i,i} + \hat{p}_{M-i+1,i} + \hat{p}_{M-i+1,N-i+1} + \hat{p}_{i,N-i+1} \right) \tag{12}$$

4 Results

Table 1 shows a comparison of operations needed for the computation of one filtered pixel using square matrix kernels with odd dimensions which are symmetric with respect to the x- and y-axis. We used odd square matrix kernels for our comparison, because these are the most frequently used kernels in image processing and anti-aliasing.

Standard is the brute force implementation of general kernel matrix filtering. *Symmetric* refers to the standard method improved by reducing multiplications through adding up the symmetric pixels before the multiplication. *Base* is the algorithm described in chapter 2 and *Optimized* uses our basic algorithm enhanced with the optimization to stop decomposing, if the remaining kernel ^{k+1}K can be computed faster with the *Symmetric* algorithm. The decomposition threshold for this algorithm with odd square matrix kernels is 3x3.

To calculate the operations needed for the *Base* algorithm with a NxN kernel, we need:

- the operations needed for a (N-2)x(N-2) kernel
- 4 additions and 1 multiplication for the addition of the four corner points multiplied with S_1
- N-1 additions and (N-1)/2 multiplications for the filtering with the horizontal vector
- N-1 additions and (N-1)/2 multiplications for the filtering with the vertical vector
- 1 addition to add the result of the vector filtering to the rest.

Because the first and the last weights in the horizontal and vertical vector are 1, only (N-1)/2 and not (N+1)/2 multiplications are needed for the vector filtering. Therefore to calculate the operations needed for a NxN kernel with odd N we need the operations needed for a (N-2)x(N-2) kernel filter plus $2\cdot N+3$ additions plus N multiplications.

The only difference in the calculation of operations for the *Optimized* algorithm is that it starts from a 3x3 kernel filtering computed with the Symmetric algorithm.

	Standard			Symmetric (Sym)			Base			Optimized		
	Add	Mul	Op	Add	Mul	Op	Add	Mul	Op	Add	Mul	Op
3x3	8	9	17	8	4	12	10	4	14	Sym	Sym	Sym
5x5	24	25	49	24	9	33	23	9	32	21	9	30
7x7	48	49	97	48	16	64	40	16	56	38	16	54
9x9	80	81	161	80	25	105	61	25	86	59	25	84
11x11	120	121	241	120	36	156	86	36	122	84	36	120
13x13	168	169	337	168	49	217	115	49	164	113	49	162
15x15	224	225	449	224	64	288	148	64	212	146	64	210

Table 1 - Comparison of algorithms with kernel symmetric with respect to the x- and y-axis

Usually square kernel filters, which are symmetric with respect to the x- and y-axis are also symmetric with respect to the diagonals. Therefore the *Symmetric* algorithm can be extended to cope with the diagonal symmetry, which reduces the number of multiplications needed for a filter computation to be smaller than that of our algorithm. But the number of operations is still lower in our proposed algorithm, therefore it beats the *Symmetric* algorithm on RISC machines, where the time needed for an addition equals the time needed for a multiplication. Table 2 shows the actual numbers.

	Symmetric (Sym)			Optimized general case			Optimized Box kernels			Optimized Bartlett kernels		
	Add	Mul	Op	Add	Mul	Op	Add	Mul	Op	Add	Mul	Op
3x3	8	3	11	Sym	Sym	Sym	4	0	4	4	2	6
5x5	24	6	30	21	8	29	8	0	8	8	4	12
7x7	48	10	58	38	15	53	12	0	12	12	6	18
9x9	80	15	95	59	24	83	16	0	16	16	8	24
11x11	120	21	141	84	35	119	20	0	20	20	10	30
13x13	168	28	196	113	48	161	24	0	24	24	12	36
15x15	224	36	260	146	63	209	28	0	28	28	14	42

Table 2 - Comparison of algorithms with kernel symmetric with respect to the x- and y-axis and the diagonals and optimized algorithms for Box and Bartlett kernels

Table 2 also contains the number of operations needed for a highly optimized filtering with Bartlett kernels and Box kernels, described in chapter 3. The Bartlett kernels itself are shown in appendix A. The *Optimized Box kernel* computation does not need a multiplication, because all values in the kernel matrix are 1. Note that in these optimizations the number of operations needed for the filtering of one pixel is only linear to the horizontal or vertical dimension of the kernel.

5 Conclusions

This paper presented an algorithm for efficient computation of matrix kernels filtering with symmetry with respect to the x- and y-axis. Our proposed algorithm clearly beats the brute force method and also the enhanced method using the symmetry to reduce the number of multiplications for kernels larger than or equal to 5x5. In the case of square matrix kernels, which are also symmetric with respect to the diagonals, the proposed algorithm uses less operations, but more multiplications with large kernels than the standard algorithm improved for exploiting these symmetries. Therefore the proposed algorithm also performs better than the symmetry enhanced standard algorithm on RISC machines, where the calculation time for an addition equals the calculation time for a multiplication.

Also optimizations of the basic algorithm using special properties of some specific kernels were exploited. Examples for these include the Box and the Bartlett kernels.

Acknowledgements

I want to thank Martin Feda, Eduard Gröller and Werner Purgathofer for their helpful comments.

Note

This paper is also available online as a Technical Report from the World-Wide Web server of the Institute of Computer Graphics at the Technical University of Vienna at *http://www.cg.tuwien.ac.at/* or via FTP at *ftp://ftp.cg.tuwien.ac.at/pub/TR/94/TR186-2-94-4.ps.gz*

References

[Crow81] F. C. Crow, A Comparison of Anti-aliasing Techniques, IEEE Computer Graphics and Applications 1(1), 40-48, Jan 1981.

[Fole90] James D. Foley, Andries van Dam, Steven K. Feiner and John F. Hughes, Computer Graphics - Principles and Practice, Second Edition, Addison-Wesley Publishing Company 1990, pp 617-646.

[Gröl93] Eduard Gröller, Coherence in Computer Graphics, PhD Thesis at the Technical University of Vienna, Verband der wissenschaftlichen Gesellschaften Österreichs (VWGÖ) 1993.

[Habe89] Peter Haberäcker, Digitale Bildverarbeitung - Grundlagen und Anwendungen, Third Edition, Carl Hanser Verlag 1989, pp 127-160.

[Russ92] John C. Russ, The Image Processing Handbook, CRC Press 1992.

[Watt92] Alan Watt and Mark Watt, Advanced Animation and Rendering Techniques, Addison-Wesley and ACM Press 1992, pp 111-154.

A Bartlett kernels

$$B_3 = \begin{pmatrix} 1 & 2 & 1 \\ 2 & 4 & 2 \\ 1 & 2 & 1 \end{pmatrix} \quad B_5 = \begin{pmatrix} 1 & 2 & 3 & 2 & 1 \\ 2 & 4 & 6 & 4 & 2 \\ 3 & 6 & 9 & 6 & 3 \\ 2 & 4 & 6 & 4 & 2 \\ 1 & 2 & 3 & 2 & 1 \end{pmatrix} \quad B_7 = \begin{pmatrix} 1 & 2 & 3 & 4 & 3 & 2 & 1 \\ 2 & 4 & 6 & 8 & 6 & 4 & 2 \\ 3 & 6 & 9 & 12 & 9 & 6 & 3 \\ 4 & 8 & 12 & 16 & 12 & 8 & 4 \\ 3 & 6 & 9 & 12 & 9 & 6 & 3 \\ 2 & 4 & 6 & 8 & 6 & 4 & 2 \\ 1 & 2 & 3 & 4 & 3 & 2 & 1 \end{pmatrix}$$

$$B_9 = \begin{pmatrix} 1 & 2 & 3 & 4 & 5 & 4 & 3 & 2 & 1 \\ 2 & 4 & 6 & 8 & 10 & 8 & 6 & 4 & 2 \\ 3 & 6 & 9 & 12 & 15 & 12 & 9 & 6 & 3 \\ 4 & 8 & 12 & 16 & 20 & 16 & 12 & 8 & 4 \\ 5 & 10 & 15 & 20 & 25 & 20 & 15 & 10 & 5 \\ 4 & 8 & 12 & 16 & 20 & 16 & 12 & 8 & 4 \\ 3 & 6 & 9 & 12 & 15 & 12 & 9 & 6 & 3 \\ 2 & 4 & 6 & 8 & 10 & 8 & 6 & 4 & 2 \\ 1 & 2 & 3 & 4 & 5 & 4 & 3 & 2 & 1 \end{pmatrix} \quad B_{11} = \begin{pmatrix} 1 & 2 & 3 & 4 & 5 & 6 & 5 & 4 & 3 & 2 & 1 \\ 2 & 4 & 6 & 8 & 10 & 12 & 10 & 8 & 6 & 4 & 2 \\ 3 & 6 & 9 & 12 & 15 & 18 & 15 & 12 & 9 & 6 & 3 \\ 4 & 8 & 12 & 16 & 20 & 24 & 20 & 16 & 12 & 8 & 4 \\ 5 & 10 & 15 & 20 & 25 & 30 & 25 & 20 & 15 & 10 & 5 \\ 6 & 12 & 18 & 24 & 30 & 36 & 30 & 24 & 18 & 12 & 6 \\ 5 & 10 & 15 & 20 & 25 & 30 & 25 & 20 & 15 & 10 & 5 \\ 4 & 8 & 12 & 16 & 20 & 24 & 20 & 16 & 12 & 8 & 4 \\ 3 & 6 & 9 & 12 & 15 & 18 & 15 & 12 & 9 & 6 & 3 \\ 2 & 4 & 6 & 8 & 10 & 12 & 10 & 8 & 6 & 4 & 2 \\ 1 & 2 & 3 & 4 & 5 & 6 & 5 & 4 & 3 & 2 & 1 \end{pmatrix}$$

B Example of the algorithm with a 5x5 kernel filter

Given a 5x5 kernel matrix ${}^1K$ and the 5x5 matrix W containing the original and neighbouring pixels:

$${}^1K = \begin{pmatrix} 2 & 3 & 4 & 3 & 2 \\ 0 & 5 & 6 & 5 & 0 \\ 1 & 7 & -1 & 7 & 1 \\ 0 & 5 & 6 & 5 & 0 \\ 2 & 3 & 4 & 3 & 2 \end{pmatrix} \quad W = \begin{pmatrix} 1 & 0 & 2 & 3 & 3 \\ 2 & 0 & 1 & 2 & 2 \\ 2 & 1 & 0 & 1 & 2 \\ 2 & 2 & 0 & 0 & 1 \\ 2 & 2 & 0 & 0 & 0 \end{pmatrix}$$

To check the results of our algorithm, we compute the kernel filtering directly:

$$\overline{p}_{direct} = 2\cdot1+3\cdot0+4\cdot2+3\cdot3+2\cdot3+0\cdot2+5\cdot0+6\cdot1+...=79.$$

According to (4), $S_1 = 2-1 = 1$, ${}^1A$ can be computed from (5a)-(5d) and ${}^2K$ from (6):

$$ {}^1A = \begin{pmatrix} 1 & 3 & 4 & 3 & 1 \\ 0 & 0\cdot 3 & 0\cdot 4 & 0\cdot 3 & 0 \\ 1 & 1\cdot 3 & 1\cdot 4 & 1\cdot 3 & 1 \\ 0 & 0\cdot 3 & 0\cdot 4 & 0\cdot 3 & 0 \\ 1 & 3 & 4 & 3 & 1 \end{pmatrix} = \begin{pmatrix} 1 & 3 & 4 & 3 & 1 \\ 0 & 0 & 0 & 0 & 0 \\ 1 & 3 & 4 & 3 & 1 \\ 0 & 0 & 0 & 0 & 0 \\ 1 & 3 & 4 & 3 & 1 \end{pmatrix} \quad {}^2K = \begin{pmatrix} 5-0 & 6-0 & 5-0 \\ 7-3 & -1-4 & 7-3 \\ 5-0 & 6-0 & 5-0 \end{pmatrix} = \begin{pmatrix} 5 & 6 & 5 \\ 4 & -5 & 4 \\ 5 & 6 & 5 \end{pmatrix} $$

From ${}^1A$ ${}^1h$ and ${}^1v$can be computed via (7):

$$ {}^1h = \begin{pmatrix} 1 & 3 & 4 & 3 & 1 \end{pmatrix} \quad {}^1v = \begin{pmatrix} 1 \\ 0 \\ 1 \\ 0 \\ 1 \end{pmatrix} $$

Now we enter the second level of the recursion and compute S_2, ${}^2A$, ${}^3K$, ${}^2h$ and ${}^2v$:

$S_2 = 5-1 = 4$

$$ {}^2A = \begin{pmatrix} 1 & 6 & 1 \\ 4 & 4\cdot 6 & 4 \\ 1 & 6 & 1 \end{pmatrix} = \begin{pmatrix} 1 & 6 & 1 \\ 4 & 24 & 4 \\ 1 & 6 & 1 \end{pmatrix} \quad {}^3K = \begin{pmatrix} -5-24 \end{pmatrix} = \begin{pmatrix} -29 \end{pmatrix} \quad {}^2h = \begin{pmatrix} 1 & 6 & 1 \end{pmatrix} \quad {}^2v = \begin{pmatrix} 1 \\ 4 \\ 1 \end{pmatrix} $$

Since the dimension of ${}^3K$ is one, the decomposition is halted. Let us look at the filter computations with the components: First we compute the sum of the pixels affected by the scalar values S_1 and S_2 using (8):

$\text{sum}_1 = 1\cdot(1+3+0+2) + 4\cdot(0+2+0+2) = 22$

The next step to do is to compute the kernel filters ${}^1h$ and ${}^1v$ on W, ${}^2h$ and ${}^2v$ on the inner 3x3 window of W and add it to the sum.

$$ {}^1v\left({}^1h(W)\right) = {}^1v \begin{pmatrix} 1\cdot 1 + 3\cdot 0 + 4\cdot 2 + 3\cdot 3 + 1\cdot 3 \\ 1\cdot 2 + 3\cdot 0 + 4\cdot 1 + 3\cdot 2 + 1\cdot 2 \\ 1\cdot 2 + 3\cdot 1 + 4\cdot 0 + 3\cdot 1 + 1\cdot 2 \\ 1\cdot 2 + 3\cdot 2 + 4\cdot 0 + 3\cdot 0 + 1\cdot 1 \\ 1\cdot 2 + 3\cdot 2 + 4\cdot 0 + 3\cdot 0 + 1\cdot 0 \end{pmatrix} = {}^1v \begin{pmatrix} 21 \\ 14 \\ 10 \\ 9 \\ 8 \end{pmatrix} = 1\cdot 21 + 0\cdot 14 + 1\cdot 10 + 0\cdot 9 + 1\cdot 8 = 39 $$

$$ {}^2v\left({}^2h\begin{pmatrix} 0 & 1 & 2 \\ 1 & 0 & 1 \\ 2 & 0 & 0 \end{pmatrix}\right) = {}^2v \begin{pmatrix} 1\cdot 0 + 6\cdot 1 + 1\cdot 2 \\ 1\cdot 1 + 6\cdot 0 + 1\cdot 1 \\ 1\cdot 2 + 6\cdot 0 + 1\cdot 0 \end{pmatrix} = {}^2v \begin{pmatrix} 8 \\ 2 \\ 2 \end{pmatrix} = 1\cdot 8 + 4\cdot 2 + 1\cdot 2 = 18 $$

$$ \text{sum}_2 = \text{sum}_1 + 39 + 18 = 79 $$

Finally we have to add the 1x1 kernel filter ${}^3K$ on the inner point of W to the sum:

$$ \overline{p} = \text{sum}_2 + {}^3K(0) = 79 + (-29)\cdot 0 = 79 $$

This yields to the same result as with the direct kernel filter computation.