An efficient algorithm for Gaussian blur using finite-state machines

Frederick M. Waltz[a] and John W. V. Miller[b]

[a]2095 Delaware Avenue, Mendota Heights, MN 55118-4801 USA
[b]ECE Department, Univ. of Michigan-Dearborn, Dearborn, MI 48128-1491 USA

Abstract

Two-dimensional Gaussian blur operations are used in many image processing applications. The execution times of these operations can be rather long, especially where large kernels are involved. Proper use of two properties of Gaussian blurs can help to reduce these long execution times:

1. Large kernels can be decomposed into the sequential application of small kernels.
2. Gaussian blurs are separable into row and column operations.

This paper makes use of both of these characteristics and adds a third one:

3. The row and column operations can be formulated as finite-state machines (FSMs) to produce highly efficient code and, for multi-step decompositions, eliminate writing to intermediate images.

This paper shows the FSM formulation of the Gaussian blur for the general case and provides examples. Speed comparisons between various implementations are provided for some of the examples. The emphasis is on software implementations, but implementations in pipelined hardware are also discussed. Straightforward extensions of these concepts to three- and higher-dimensional image processing are also presented. Implementation techniques for DOG (Difference-of-Gaussian filters) are also provided.

Keywords: Gaussian blur, separation, decomposition, finite-state machines, efficient code

1. Introduction

A wide range of low-pass filtering operations are used in image processing – low pass in the sense of passing low spatial frequencies and rejecting high spatial frequencies. These are usually given names suggesting smoothing, averaging, or blurring. One of the most widely-used operations of this type is the so-called Gaussian blur, which has the advantages of being very "smooth" and also circularly symmetric, so that edges and lines in various directions are treated similarly. For digital image processing, blurring operators are often defined on small neighborhoods (e.g., 3x3, 11x11), and a small finite number of grey levels (e.g., 256). In such cases, the ideal Gaussian "bell curve" must be approximated by a few integer values. The values usually used are based on Pascal's triangle (the binomial coefficients. These approach the true Gaussian curve more and more closely as the number of points increases:

Index N	Coefficients	Sum of coefficients = 2^N
0	1	1
1	1 1	2
2	1 2 1	4
3	1 3 3 1	8
4	1 4 6 4 1	16
5	1 5 10 10 5 1	32
6	1 6 15 20 15 6 1	64
7	1 7 21 35 35 21 7 1	128
8	1 8 28 56 70 56 35 8 1	256
9	1 9 36 84 126 126 84 36 9 1	512
10	1 10 45 120 210 252 210 120 45 10 1	1024
11	1 11 55 165 330 462 462 330 165 55 11 1	2048

These coefficients have the very useful property that the set for N = k can be obtained by convolving the set for N = i with the set for N = j, where k = i + j. For example, the set for N = 7 can be obtained by convolving [1 3 3 1] with [1 4 6 4 1]. The absolutely crucial implication of this is that repeated applications of small-neighborhood Gaussian blurs can achieve large-neighborhood Gaussian blurs. These results are exactly equivalent, except for the accumulated rounding errors due to the successive operations. This is true for one-dimensional Gaussian operators as well as for two-dimensional operators and all higher dimensions.

Two-dimensional Gaussian blur operations are used in many image processing applications. The execution times of these operations can be rather long, especially where large kernels are involved. Proper use of two properties of Gaussian blurs can help to reduce these long execution times:

1. Large kernels can be decomposed into the sequential application of small kernels, as indicated in the paragraph above.
2. Gaussian blurs are separable into independent row and column operations.
   For example, the 3x3 Gaussian blur operator

$$\begin{vmatrix} 1 & 2 & 1 \\ 2 & 4 & 2 \\ 1 & 2 & 1 \end{vmatrix} \text{ is exactly equivalent to } \begin{vmatrix} 1 & 2 & 1 \end{vmatrix} \text{ followed by } \begin{vmatrix} 1 \\ 2 \\ 1 \end{vmatrix}.$$

   Alternatively, the row and column operators can be interchanged, with identical results.

This paper makes use of both of these characteristics and adds a third one:

3. The row and column operations can be formulated as finite-state machines (FSMs) to produce highly efficient code. Furthermore, for multi-step sequential decompositions, the necessity of writing results to intermediate images and then fetching these results for the next operation is eliminated.

2. The basic 2x2 operation

Figure 1 shows the simplest possible two-dimensional operator, the 2x2 blur, which has rows [1 1] and [1 1], implemented using the SKIPSM (Separated-Kernel Image Processing using finite-State Machines) paradigm. (Please see References 1 through 20 for background information on SKIPSM.)

This implementation requires one memory location or register for the state of the row machine SR0 and a column state buffer SC0[i], i = 1, 2, … NPixels, where NPixels is the number of pixels in an image row. The column state buffer is set to zero at the start of the overall operation, and the row state buffer is set to zero at the start of each row.

For this example and all the others to follow, each image pixel is fetched *once and only once*. For each pixel, the main loop code shown below is executed once. There are no other steps. The output can be written back onto the input image, if desired, because all the information needed to compute the output is contained in the state buffers.

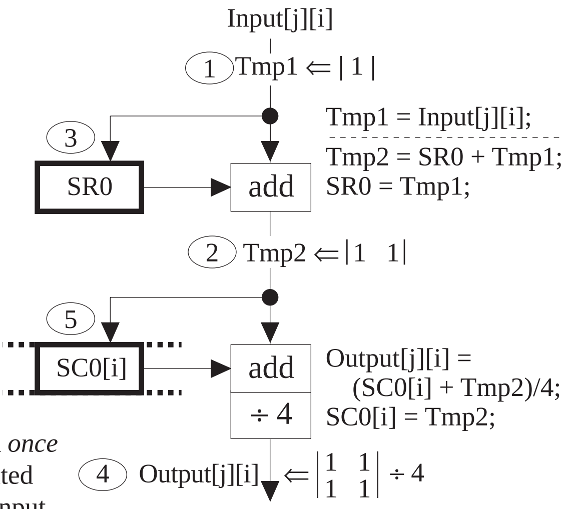


Figure 1. SKIPSM implementation of the 2x2 Gaussian blur operator.

Step	Main Loop Code	Comments
1	Tmp1 = Input[j][i];	// Fetch the next input pixel
2	Tmp2 = SR0 + Tmp1;	// Form the row machine output
3	SR0 = Tmp1;	// Update the row state buffer
4	Output[j][i] = (2 + SC0[i] + Tmp2)/4;	// Form and scale the output
5	SC0[i] = Tmp2;	// Update the column state buffer

This 2x2 SKIPSM implementation uses five steps (or six, if one counts step 4 as two steps), whereas a pure “brute force” implementation would use nine (four dual-index pixel fetches, three additions, one scaling step, and a dual-index “write” to the output image). In this case, the SKIPSM result is not particularly impressive. The real advantages of SKIPSM increase as the size of the operator increases. And, of course, nobody is likely to use something as cumbersome as the brute force approach. (Note: The addition of the value 2 in Step 4 provides for rounding to the nearest integer.)

3. Larger blurs: 3x3 and 5x5

This implementation requires two temporary variables, one memory location or register for the state of the row machine (SR0), for images with NPixels in each row, a column state buffer SC0[i], i = 1, 2, … NPixels. The 3x3 implementation requires two temporary variables, two memory locations or registers for the state of the row machine, SR0 and SR1, and two column state buffers, SC0[i] and SC1[i]. The column state buffers are set to zero at the start of the overall operation, and the row state buffers are set to zero at the start of each row. The main loop code is given below. The output is written to address [j-1][i-1] because this is the center of the current 3x3 neighborhood.

```
#  Main Loop Code for 3x3 Gaussian blur
   //  Row machine
1  Tmp1 = Input[j][i];                          // Fetch the next input pixel
2  Tmp2 = SR0 + Tmp1;                           // Form the intermediate value
3  SR0 = Tmp1;                                  // Update 1st row state buffer
4  Tmp1 = SR1 + Tmp2;                           // Form the row machine output
5  SR1 = Tmp2;                                  // Update 2nd row state buffer
   //  Column machine
6  Tmp2 = SC0[i] + Tmp1;                        // Form the intermediate value
7  SC0[i] = Tmp1;                               // Update 1st column state buffer
8  Output[j-1][i-1] = (8 + SC1[i] + Tmp2)/16;   // Form the output
9  SC1[i] = Tmp2;                               // Update 2nd column state buffer
```

Thus, the 3x3 SKIPSM implementation requires 9 steps. The brute-force approach for this case requires 24 steps (9 dual-index pixel fetches, 5 multiplies, 8 additions, one scaling step, and a dual-index "write" to the output image).

The 5x5 implementation requires two temporary variables, four memory locations or registers for the state of the row machine, SR0 through SR3, and four column state buffers. The column state buffers are set to zero at the start of the overall operation, and the row state buffers are set to zero at the start of each row. The main loop code is given below. The output is written to address [j-2][i-2] because this is the center of the current 5x5 neighborhood. See Figure 3.

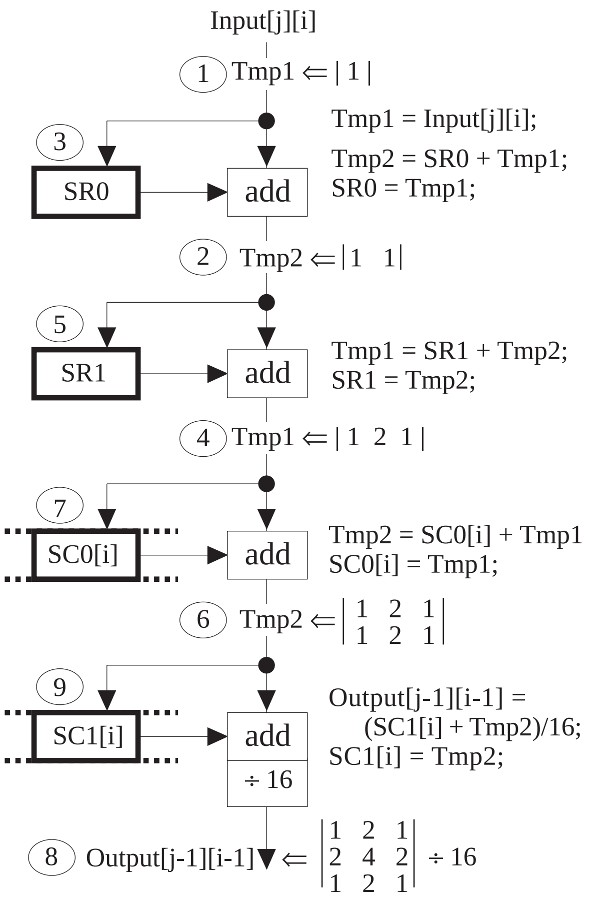


Figure 2. SKIPSM implementation of the 3x3 Gaussian blur operator.

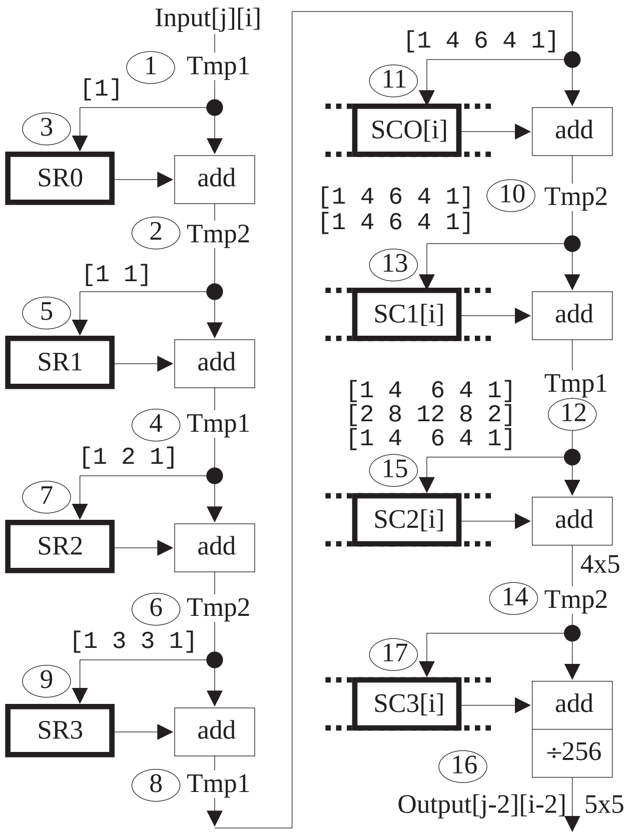


Figure 3. SKIPSM implementation of the 5x5 Gaussian blur operator.

Step	Main Loop Code for 5x5 Gaussian blur
1	Tmp1 = Input[j][i];
2	Tmp2 = SR0 + Tmp1;
3	SR0 = Tmp1;
4	Tmp1 = SR1 + Tmp2;
5	SR1 = Tmp2;
6	Tmp2 = SR2 + Tmp1;
7	SR2 = Tmp1;
8	Tmp1 = SR3 + Tmp2;
9	SR3 = Tmp2;
10	Tmp2 = SC0[i] + Tmp1;
11	SC0[i] = Tmp1;
12	Tmp1 = SC1[i] + Tmp2;
13	SC1[i] = Tmp2;
14	Tmp2 = SC2[i] + Tmp1;
15	SC2[i] = Tmp1;
16	Output[j-2][i-2] = (128 + SC3[i] + Tmp2)/256;
17	SC3[i] = Tmp2;

Thus, the 5x5 SKIPSM implementation requires 17 steps. In comparison, the brute-force approach for this case requires 72 steps (25 dual-index pixel fetches, 21 multiplications, 24 additions, 1 scaling step, and 1 "write" to the output image).

Six implementations will now be compared with respect to the number of code steps required for odd values of N. The “brute-force” approach is included for comparison, ever though nobody would use it. Instead, they would use either a decomposition or repeated applications of a smaller blur – typically 3x3.

method	pixel fetches	multiplications	additions	scaling	write	total
NxN brute force	N^2	$N^2 - 4$	$N^2 - 1$	1	1	$3*N^2 - 3$
3x3 brute force, (N - 1)/2 times	—	—	—	—	—	12*N-12
Nx1 and 1xN decomposition	2*N	2*(N - 2)	2*(N - 1)	2	2	6*N - 2
decomposed 3x3, (N - 1)/2 times	—	—	—	—	—	8*N - 8
NxN SKIPSM	—	—	see sample code	—	—	4*N - 3
3x3 SKIPSM, (N - 1)/2 times	—	—	—	—	—	9*(N-1)/2

Here are the results for NxN Gaussian blurs, for various odd values of N:

Number of main-loop steps for N =	3	5	7	9	11	13	15	17	19	21	23
NxN brute force	24	72	144	240	360	477	672	864	1080	1320	1584
3x3 brute force, (N - 1)/2 times	24	48	72	96	120	144	168	192	216	240	264
Nx1 and 1xN decomposition	16	28	40	52	64	76	88	100	112	124	136
decomposed 3x3, (N - 1)/2 times	16	32	48	64	60	96	112	128	144	160	176
NxN SKIPSM	9	17	25	33	41	49	57	65	73	81	89
3x3 SKIPSM, (N - 1)/2 times	9	18	27	36	45	54	63	72	81	90	99

It can be seen that the repeated application of the small SKIPSM operator is almost as good as integrating all the steps into one SKIPSM operator, and that either one is much better that either the brute-force implementations or the decompositions. In fact, the one-step SKIPSM algorithm is even better than this makes it appear, because not only does it eliminate the writing to and reading from intermediate images (which are included in this tally), but it eliminates the *extra* overhead required by the separate double main loops used for each individual pass (which are not included in this tally).

Finally, another real advantage of using the combined SKIPSM approach is in the elimination of rounding errors. By postponing any rounding/scaling step until the end of the operation, full precision is maintained throughout, the Gaussian ideal is approached more closely, and the rather crude approximations involved in small kernels are avoided completely.

4. NON-SQUARE GAUSSIAN BLURS

There is no reason to limit oneself to square operators, or to odd-numbered ones either. Any size row machine can be used with any size column machine. (The only difficulty comes with deciding where to put the result when N is even.) For a SKIPSM blur operator with N columns [(N - 1) row states] and M rows [(M - 1) column state buffers], the number of main loop code steps becomes 1 + 2*(N - 1) + 2*(M - 1) = (2*N + 2*M - 3).

Main loop code for 3x5 Gaussian Blur

```
Tmp1 = Input[j][i]
Tmp2 = SR0 + Tmp1;
SR0 = Tmp1;
Tmp1 = SR1 + Tmp2;
SR1 = Tmp2;
Tmp2 = SR2 + Tmp1;
SR2 = Tmp1;
Tmp1 = SR3 + Tmp2;
SR3 = Tmp2;
Tmp2 = SC0[i] + Tmp1;
SC0[i] = Tmp1;
Output[j-1][i-2] = (32 + SC1[i] + Tmp2)/64;
SC1[i] = Tmp2;
```

Coefficients

$$\left|\begin{array}{ccccc}1&4&6&4&1\\2&8&12&8&2\\1&4&6&4&1\end{array}\right| \div 64$$

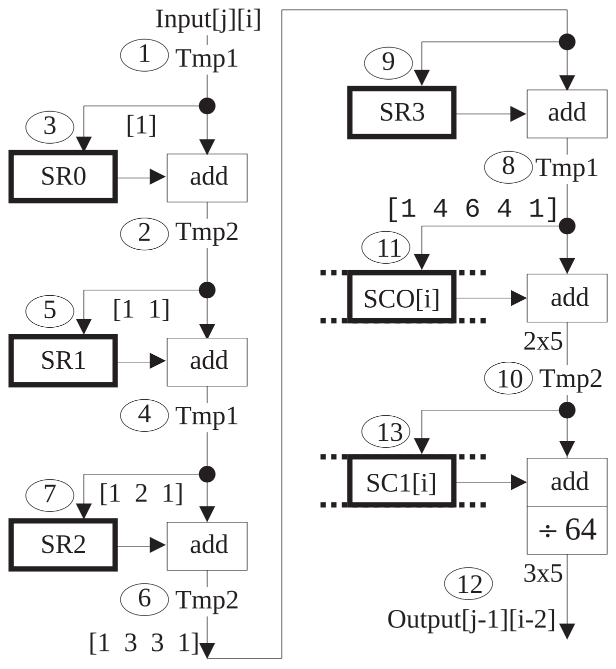


Figure 4. 5 column-by-3 row Gaussian blur operator

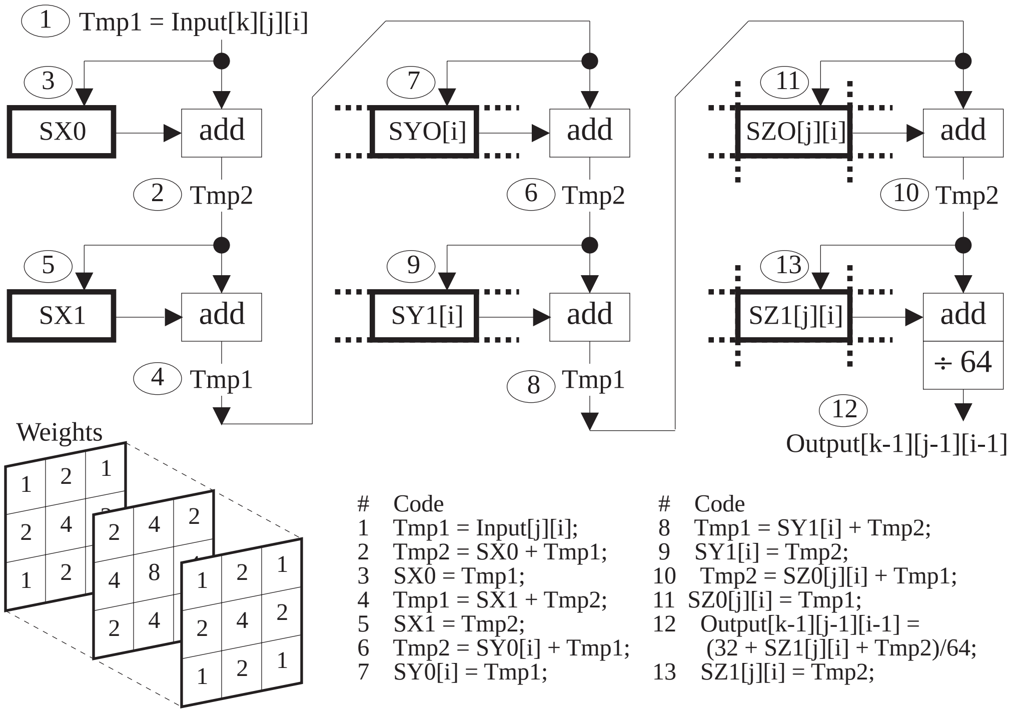
Figure 4 shows a blur operator with 5 columns and 3 rows. As predicted, it has 13 main loop steps. Generalization to other sizes and shapes is very straightforward.

5. Rotated elliptical Gaussian blurs

It is not even necessary that the axes of non-circular blurs be aligned with the x and y axes of the image. It is possible also to use two diagonal finite-state machines to produce elliptical blurs rotated 45 degrees with respect to these axes. Diagonal machines were used in [15] for grey-scale erosion, and a similar approach could easily be adapted to Gaussian blurs. The details are omitted here.

6. Three-Dimensional and n-dimensional Gaussian blurs

The SKIPSM approach can be extended to 3-dimensional and n-dimensional operations without difficulty. Figure 5 shows the result for a 3x3x3 implementation. For clarity, the notation has been changed from rows and columns to X, Y, and Z. For this example there are three kinds of state buffers: registers SX0 and SX1 for the two X state values, two row-length buffers SY0[i] and SY1[i] for the Y state values, and two image-sized buffers SZ0[j][i] and SZ1[j][i]. All buffers except SX0 must have longer word lengths than the number of bits in the input image, with registers further along in the processing chain requiring progressively more. In this case, and for 8-bit input values, the number of bits for SX0, SX1, etc. are 8, 9, 10, 11, 12, and 13, respectively. Before scaling (dividing by 64), the sum computed in step 12 can have up to 14 bits. In this case, 16 bit buffers could be used for all of these. For larger-sized operators, care must be taken to ensure that the buffer word lengths are sufficiently large, especially toward the end of the processing chain.



#	Code	#	Code
1	Tmp1 = Input[j][i];	8	Tmp1 = SY1[i] + Tmp2;
2	Tmp2 = SX0 + Tmp1;	9	SY1[i] = Tmp2;
3	SX0 = Tmp1;	10	Tmp2 = SZ0[j][i] + Tmp1;
4	Tmp1 = SX1 + Tmp2;	11	SZ0[j][i] = Tmp1;
5	SX1 = Tmp2;	12	Output[k-1][j-1][i-1] = (32 + SZ1[j][i] + Tmp2)/64;
6	Tmp2 = SY0[i] + Tmp1;		
7	SY0[i] = Tmp1;	13	SZ1[j][i] = Tmp2;

Figure 5. SKIPSM implementation of a 3x3x3 Gaussian blur on a 3-D “image”

As with 2-D SKIPSM, the number of main loop steps required increases linearly with operator size N, whereas with conventional approaches the number of steps typically increases proportional to N^n. See [13] for more information about higher-dimensional processing using SKIPSM.

7. Difference-of-Gaussian (DOG) filters

Gaussian low-pass filters (Gaussian blurs) are often used in the computation of high-pass and band-pass filters. High-pass filters are obtained by subtracting a low-pass result from the original image. Band-pass filters are obtained by subtracting Gaussian blurs of two different sizes. These filters are called DOG (Difference of Gaussian) filters, and result in the familiar “Mexican hat” impulse response. They are widely used (and even more widely advocated) for pattern recognition applications. The human retina appears to have a series of DOG filters built into the retinal pre-processing. Crowley [21-23] refined DOG filters to an amazing degree, including the development very efficient computational techniques for a compact invertible multi-band decomposition of images.

If a full multi-band image decomposition is desired, it would be difficult to do better than Crowley has done. But if all that is wanted is one or a few DOG filters applied to an image, or if one can't make use of the particular √2 bandwidth ratios inherent in Crowley's method, then SKIPSM provides an efficient and accurate alternative.

To facilitate discussion, the following notation will be introduced:

Let G_{mxn} be the Gaussian blur operator with m rows and n columns.

Let $G_{1x1} = [1]$ be the identity operator.

Let $G_{1x2} = (1/2)\begin{vmatrix} 1 & 1 \end{vmatrix}$ $\quad G_{2x1} = (1/2)\begin{vmatrix} 1 \\ 1 \end{vmatrix}$ $\quad$ and $G_{2x2} = (1/4)\begin{vmatrix} 1 & 1 \\ 1 & 1 \end{vmatrix}$

Let * denote the convolution operation. Then $G_{mxn} * G_{1x1} = G_{1x1} * G_{mxn} = G_{mxn}$ and

$G_{pxq} * G_{rxs} = G_{rxs} * G_{pxq} = G_{(p+r-1)x(q+s-1)}$. This is the property 1 of Section 1, above.

All the larger Gaussian blurs, square or rectangular, can be defined recursively in terms of the this property, using the definitions of G_{1x2}, G_{2x1}, and G_{2x2} as starting points. All will have a sum of coefficients equal to one, when the scale factor is taken into account.

Strictly speaking, only operators which are of the same size can be added or subtracted. Therefore, the difference G_{1x1} - G_{3x3} is not defined. However, it will be assumed that when such a difference is needed, the smaller operator is "padded" with zeros to make them the same size, and then the subtraction is done. Therefore, let $DOG_{mxn-pxq}$ represent the difference between G_{mxn} and G_{pxq}. The usual practice is to subtract the larger-region blur from the smaller, so that the center value of the operator will be positive.

An example: One of the simplest DOG filters, $G_{1x1} – G_{3x3}$, has the form

$$\begin{vmatrix} 0 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \end{vmatrix} - (1/16)\begin{vmatrix} 1 & 2 & 1 \\ 2 & 4 & 2 \\ 1 & 2 & 1 \end{vmatrix} = (1/16)\begin{vmatrix} -1 & -2 & -1 \\ -2 & 12 & -2 \\ -1 & -2 & -1 \end{vmatrix}$$

The key time-saving idea in the SKIPSM implementation of DOG filters is that the results for the smaller blur, which are obtained in the process of computing the larger blur, are set aside and saved until needed. The only difficulty here is that the two blurs to be subtracted must be centered on the same pixel, whereas the SKIPSM algorithm produces blurs which share the current pixel, which is at the lower right hand corner of both regions. See Figure 6. Thus, it would appear that a separate image buffer must be used to save this result until needed. However, the pipeline timing works out just right so that this is unnecessary, as shown in Figure 6. One word of caution: The temporary storage buffer should have sufficient word length to store the sums involved in the unscaled smaller blur. If this can not be arranged, then the smaller-blur results must be scaled and rounded before storage, resulting in some loss of accuracy (usually slight). Note that, here as always with the SKIPSM approach, the scaled output results can be written back into the input image buffer, if desired, with no loss of data.

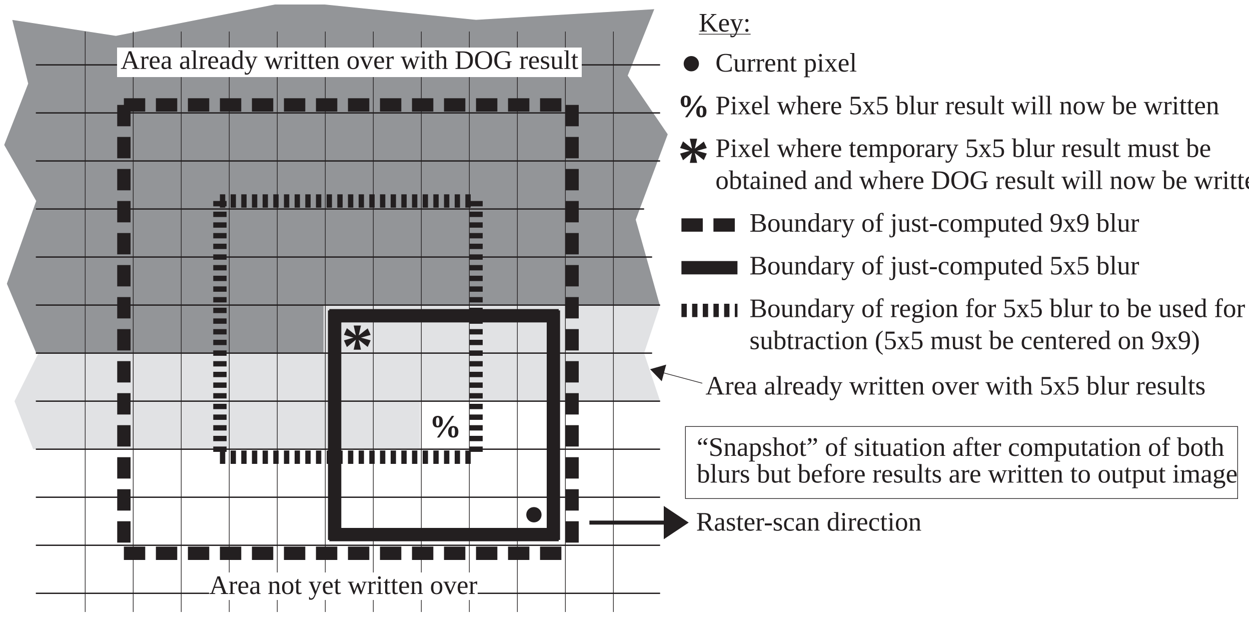

Area already written over with DOG result
Key:
● Current pixel
% Pixel where 5x5 blur result will now be written
✱ Pixel where temporary 5x5 blur result must be obtained and where DOG result will now be written
Boundary of just-computed 9x9 blur
Boundary of just-computed 5x5 blur
Boundary of region for 5x5 blur to be used for subtraction (5x5 must be centered on 9x9)
Area already written over with 5x5 blur results
"Snapshot" of situation after computation of both blurs but before results are written to output image
Raster-scan direction
%
✱
Area not yet written over

Figure 6. Relative timing information for $DOG_{5x5-9x9}$

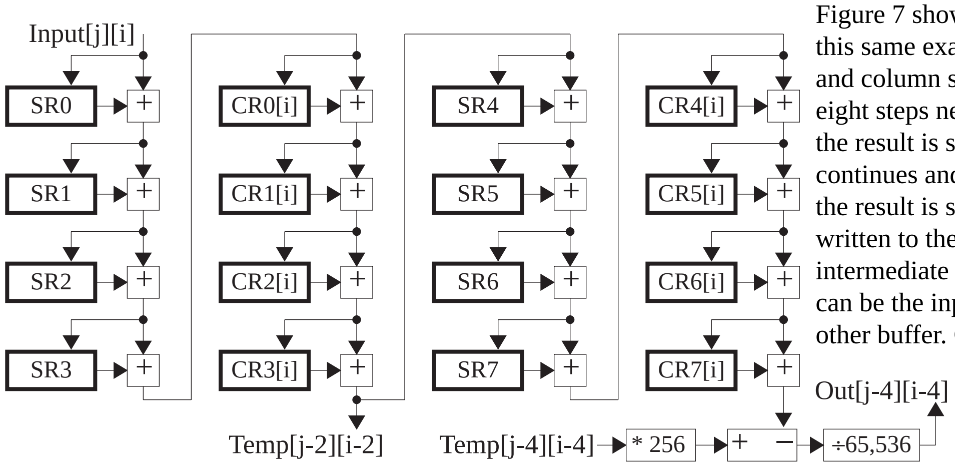


Figure 7. Programming diagram for $\mathrm{DOG}_{5x5-9x9}$

Figure 7 shows the programming diagram for this same example. Because the individual row and column steps can be taken in any order, the eight steps needed for the 5x5 blur are done first, the result is saved, and then the operation continues and the 9x9 blur is computed. Finally, the result is subtracted and scaled before being written to the output image buffer. The intermediate buffer is here named “Temp,” but can be the input buffer, the output buffer, or some other buffer. Other sizes can be obtained similarly. It is even possible to “pick off” three or more blurs of various sizes, so that multiple DOG filters could be applied to a given input image in a single pass.

8. Speed Comparisons

In order to demonstrate the advantage of SKIPSM for implementing Gaussian filters, a comparison was made with two other approaches for implementing this function. The brute force technique used the 5x5 kernel given in Table 1. In order to get a filtered pixel value, 25 multiplications and 24 additions are needed. The other approach used two one-dimensional kernels with values given in Table 2. Here the two-dimensional Gaussian kernel has been decomposed into the two given one-dimensional kernels, which can be applied in either order.

$$\begin{vmatrix} 1 & 4 & 6 & 4 & 1 \\ 4 & 16 & 24 & 16 & 4 \\ 6 & 24 & 36 & 24 & 6 \\ 4 & 16 & 24 & 16 & 4 \\ 1 & 4 & 6 & 4 & 1 \end{vmatrix}$$

Table 1. Standard 5x5 convolution kernel

$$\begin{vmatrix} 1 & 4 & 6 & 4 & 1 \end{vmatrix} \qquad \begin{vmatrix} 1 \\ 4 \\ 6 \\ 4 \\ 1 \end{vmatrix}$$

Table 2. Decomposed convolution kernels

Each program was compiled using the DOS port of GCC (DJGPP) using -O3 for optimization. No attempt was made to use the special Pentium® compiler with -O6 optimization but results would probably be around 10-15% faster with it.

The different approaches were all tested on a PC-compatible computer using a 166 MHz AMD K6 CPU with 512 KBytes of secondary cache and 64 Mbytes of memory under Windows 95®. There were a number of background processes running which may have increased times slightly but would not invalidate the comparisons since all tests were performed under the same conditions. Given that the execution times for a single image were under a second for all of the approaches, processing was repeated on the input image as needed so that the total time was on the order of one minute. The time for each iteration was calculated for the three different methods and is given in Table 3.

Approach	Time/Iteration
5x5 conventional convolution	919 ms
5x5 decomposition	450 ms
5x5 SKIPSM	108 ms

Table 3. Execution times for three implementations of a 5x5 Gaussian blur

As expected, implementing the Gaussian blur filter through standard convolution was relatively slow. Decomposition was about twice as fast. But the best results were obtained with the SKIPSM approach, which was almost nine times as fast as standard convolution. The speed advantages of the SKIPSM implementation should be even greater for larger values of N.

9. Summary and Conclusions

This paper has shown, both by analysis and example, that the SKIPSM implementation of the Gaussian blur operation provides a significant speed increase, in comparison with conventional implementations. Thus, the pipelined recursive finite-state machine aspect of the SKIPSM formulation provides speed increases *in addition to those provided by separation*. These relative speed improvements can be expect to increase further for larger values of N.

10. Bibliography

1. F. M. Waltz, "SKIPSM: Separated-Kernel Image Processing using finite-State Machines," *Proc. SPIE Conf. on Machine Vision Applications, Architectures, and Systems Integration III, Vol. 2347, Paper No. 36,* Boston, Nov. 1994
2. F. M. Waltz and H. H. Garnaoui, "Application of SKIPSM to binary morphology," *Proc. SPIE Conf. on Machine Vision Applications, Architectures, and Systems Integration III, Vol. 2347, Paper No. 37,* Boston, Nov. 1994
3. F. M. Waltz and H. H. Garnaoui, "Fast computation of the Grassfire Transform using SKIPSM," *Proc. SPIE Conf. on Machine Vision Applications, Architectures, and Systems Integration III, Vol. 2347, Paper No. 38,* Boston, Nov. 1994
4. F. M. Waltz, "Application of SKIPSM to binary template matching," *Proc. SPIE Conf. on Machine Vision Applications, Architectures, and Systems Integration III, Vol. 2347, Paper No. 39,* Boston, Nov. 1994
5. F. M. Waltz, "Application of SKIPSM to grey-level morphology," *Proc. SPIE Conf. on Machine Vision Applications, Architectures, and Systems Integration III, Vol. 2347, Paper No. 40,* Boston, Nov. 1994
6. F. M. Waltz, "Application of SKIPSM to the pipelining of certain global image processing operations," *Proc. SPIE Conf. on Machine Vision Applications, Architectures, and Systems Integration III, Vol. 2347, Paper No. 41,* Boston, Nov. 1994
7. A. A. Hujanen & F. M. Waltz, "Pipelined implementation of binary skeletonization using finite-state machines," *Proc. SPIE Conf. on Machine Vision Applications in Industrial Inspection, Vol. 2423, Paper No. 2,* San Jose, Feb. 1995
8. A. A. Hujanen & F. M. Waltz, "Extending the SKIPSM binary skeletonization implementation," *Proc. SPIE Conf. on Machine Vision Applications, Architectures, and Systems Integration IV, Vol. 2597, Paper No. 12,* Philadelphia, Oct. 1995
9. F. M. Waltz, "Application of SKIPSM to binary correlation," *Proc. SPIE Conf. on Machine Vision Applications, Architectures, and Systems Integration IV, Vol. 2597, Paper No. 11,* Philadelphia, Oct. 1995
10. F. M. Waltz, "SKIPSM implementations: morphology and much, much more," *Proc. SPIE Conf. on Machine Vision Applications, Architectures, and Systems Integration IV, Vol. 2597, Paper No. 14,* Philadelphia, Oct. 1995
11. F. M. Waltz, "Automated generation of finite-state machine lookup tables for binary morphology," *Proc. SPIE Conf. on Machine Vision Applications, Architectures, and Systems Integration V, Boston,* Nov. 1996
12. F. M. Waltz, "Binary openings and closings in one pass using finite-state machines," *Proc. SPIE Conf. on Machine Vision Applications, Architectures, and Systems Integration V*, Boston, Nov. 1996
13. F. M. Waltz, "Implementation of SKIPSM for 3-D binary morphology,"*Proc. SPIE Conf. on Machine Vision Applications, Architectures, and Systems Integration VI, Vol. 3205, Paper No. 13,* Pittsburgh, Oct. 1997
14. F. M. Waltz, "Binary dilation using SKIPSM: Some interesting variations," *Proc. SPIE Conf. on Machine Vision Applications, Architectures, and Systems Integration VI, Vol. 3205, Paper No. 15,* Pittsburgh, Oct. 1997
15. J. W. V. Miller and F. M. Waltz, "Software implementation of 2-D grey-level dilation using SKIPSM," F. M. Waltz and J. W. V. Miller, "An efficient algorithm for Gaussian blur using finite-state machines," *Proc. SPIE Conf. on Machine Vision Systems for Inspection and Metrology VII, Vol. 3521, Paper No. 18,* Pittsburgh, Oct. 1997
16. R. Hack, F. M. Waltz, & B. G. Batchelor, "Software implementation of the SKIPSM paradigm under PIP," *Proc. SPIE Conf. on Machine Vision Applications, Architectures, and Systems Integration VI, Vol. 3205, Paper No. 19,* Pittsburgh, Oct. 1997
17. F. M. Waltz, "The application of SKIPSM to various 3x3 image processing operations," *Proc. SPIE Conf. on Machine Vision Systems for Inspection and Metrology VII, Vol. 3521, Paper No. 30,* Boston, Nov. 1998
18. F. M. Waltz and J. W. V. Miller, "Fast, efficient algorithms for 3x3 ranked filters using finite-state machines," *Proc. SPIE Conf. on Machine Vision Systems for Inspection and Metrology VII, Vol. 3521, Paper No. 31,* Boston, Nov. 1998
19. F. M. Waltz, "Automated generation of efficient code for grey-scale image processing," *Proc. SPIE Conf. on Machine Vision Systems for Inspection and Metrology VII, Vol. 3521, Paper No. 32,* Boston, Nov. 1998
20. F. M. Waltz, "Image processing operations in color space using finite-state machines," *Proc. SPIE Conf. on Machine Vision Systems for Inspection and Metrology VII, Vol. 3521, Paper No. 33,* Boston, Nov. 1998
21. J. L. Crowley, *A Representation of Visual Information,* PhD dissertation, Carnegie-Mellon University, 1981
22. J. L. Crowley and A. C. Parker, "A representation for shape based on peaks and ridges in the Difference-of-Low-Pass transform," *IEEE Transactions on PAMI,* March 1984
23. J. L. Crowley and A. M. Lowrie, *Multiple Resolution and Matching of Stereo Scan Lines,* Technical Report, The Robotics Institute, Carnegie-Mellon University, January 1985