The application of SKIPS to various 3x3 image processing operations

Frederick M. Waltz
2095 Delaware Avenue, Mendota Heights, MN 55118-4801 USA

ABSTRACT

Most of the published work about SKIPS (Separated-Kernel Image Processing using finite-State Machines) has concentrated on large-neighborhood operations (e.g., binary morphology, Gaussian blur), because the speed improvements are the most dramatic in such cases. However, there are many frequently-used 3x3 operations that could also benefit from speed improvements that arise from separability and the use of finite-state machines.

This paper shows SKIPS implementations for an extensive list of 3x3 operations, including various edge detectors, connectivity detectors, direction of brightest neighbor, largest gradient, and direction of largest gradient. A generic implementation applicable to all 3x3 binary operations, whether separable of non-separable, is also given.

Keywords: image processing, linear, nonlinear, separability, finite-state machines

1. INTRODUCTION

The SKIPS (Separated-Kernel Image Processing using finite-State Machines) paradigm has been described in numerous previous publications1-22. Most of these have concentrated on large-neighborhood operations (e.g., binary morphology, Gaussian blur), because the speed improvements are the most dramatic in such cases. However, there are many frequently-used 3x3 operations that could also benefit from speed improvements that arise from separability and the use of finite-state machines.

This paper shows SKIPS implementations for an extensive list of 3x3 operations, including various edge detectors, connectivity detectors, direction of brightest neighbor, largest gradient, and direction of largest gradient. A generic implementation applicable to all 3x3 binary operations, whether separable of non-separable, is also given.

2. ALL 3X3 BINARY OPERATIONS

First, a brief review of SKIPS principles: Two finite-state machines are used, usually but not always a row machine and a column machine. The input image pixels are read in the usual video raster-scan order, left to right and top to bottom. Each pixel is read once and only once. All information needed for later computations is contained (often in highly compact form) in the state buffers of the finite-state machines. The column machine state buffer(s) must be initialized at the start of the operation, and the row machine state register(s) must be initialized at the beginning of each row. SKIPS often provides speedups and simpler code, in comparison to traditional implementations. These improvements stem mainly from the separation of 2-dimensional operations into two 1-dimensional operations, but the pipelining and finite-state machines also provide some speedup in many cases.
The basic implementation shown in Figure 1 can be used for all possible binary image processing operations defined on a 3x3 neighborhood, simply by choosing the appropriate output lookup table. In addition, a thresholding operation can be built into the input (row machine) lookup table, so that grey-scale inputs images can be accepted in addition to binary images. Possible operations include erosion and dilation with arbitrary 3x3 structuring elements, isolated white-point removal and/or black-point removal, binary edge, connectivity detection, directional edge detection, and many others. This is not one of the more impressive implementations of SKIPSM, being roughly equivalent to a serpentine memory. However, it does allow a common approach to all operations, making the development of new operations very easy. A paper on connected component analysis is now in preparation.

The row and column LUTs are defined as 2-dimensional arrays, thus: R_LUT_State[4][2], R_LUT_Out[4][2], and C_LUT_State[64][8], and C_LUT_Out[64][8]. The main-loop code for executing the lookup operations then becomes

\[
R_{\text{State}} = R_{\text{LUT State}}[R_{\text{State}}][R_{\text{In}}]; \quad C_{\text{State}[i]} = C_{\text{LUT State}}[C_{\text{State}[i]}][R_{\text{Out}}];
\]

\[
R_{\text{Out}} = R_{\text{LUT Out}}[R_{\text{State}}][R_{\text{In}}]; \quad C_{\text{Out}} = C_{\text{LUT Out}}[C_{\text{State}[i]}][R_{\text{Out}}];
\]

The optional output LUT is defined as 1-dimensional array, thus: Output[512]. The code for executing this lookup operation then becomes

\[
\text{Output}[j][i] = \text{Output}_LUT[C_{\text{Out}}].
\]

Some sample output lookup tables:

- isolated white and black point removal — Set these entries to “white” (nominally 255): 17…31, 48…63, 80…95, 112…127, 144…159, 179…191, 208…223, 240…255, 272…287, 304…319, 336…351, 368…383, 400…415, 432…447, 464…479, 495…511. Set all other entries to “black.”
- 8-connected binary erosion — Set all entries to zero except entry 511, which is set to the grey level corresponding to “white” (nominally 255).

### 3. The Laplacian Operator

The 3x3 Laplacian operator is a symmetrical second-derivative operator used to highlight edges. Figure 2 shows a SKIPSM implementation for this operation. Also shown is sample code for the main loop. To save space in this paper and to make the code easier to read and understand, many of the unnecessary carriage returns and “curly brackets” have been omitted.

#### The Laplacian Code

<table>
<thead>
<tr>
<th>Step</th>
<th>Laplacian code</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td></td>
<td></td>
</tr>
<tr>
<td>9</td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

#### Convolution Coefficients

<table>
<thead>
<tr>
<th>-1</th>
<th>-1</th>
<th>-1</th>
</tr>
</thead>
<tbody>
<tr>
<td>-1</td>
<td>8</td>
<td>-1</td>
</tr>
<tr>
<td>-1</td>
<td>-1</td>
<td>-1</td>
</tr>
</tbody>
</table>

divide by 8

#### Step 0

- \( Tmp1 = \text{Input}[j][i] = M \)

#### Step 1

- \( RS0 = Tmp1; \)

#### Step 2

- \( Tmp2 = Tmp1 + RS1; \)

#### Step 3

- \( Tmp3 = 9*RS0; \)

#### Step 4

- \( RS1 = RS0 + Tmp1; \)

#### Step 5

- \( \text{L} = Tmp1 - \text{RS1}; \)

#### Step 6

- \( \text{L} + \text{M} = Tmp2; \)

#### Step 7

- \( 9*G - (F+G+H) = \text{CS1}[i] \)

#### Step 8

- \( 9*L - (K+L+M) = \text{CS0}[i] \)

Figure 2. SKIPSM implementation of the Laplacian.

### 4. Roberts Cross

The Roberts Cross edge-detection operation is widely used, in part because it uses a 2x2 neighborhood in its computation, which presumably makes it faster than edge detectors such as the Sobel which use a 3x3 neighborhood. But because the neighborhood has no “center pixel” at which to place the result, the output image is “shifted” a half-pixel in the x and y directions. This section presents a version of the Roberts Cross defined symmetrically on a 3x3 neighborhood. Two options are presented for combining the two diagonal differences: average or maximum. Figure 3 shows two versions of the SKIPSM implementation: a conventional one in which the same code is used for all rows of the image, and a “ping-ponged” approach in which different code is used for alternate image lines, in order to eliminate one state update step. The (slight) extra overhead involved in running the “ping-pong” scheme might be regarded as not worth the trouble, if only one step is eliminated. But the extra step involved in running the “ping-pong” scheme happens once per image row, whereas the eliminated
step happens once per pixel. Thus, unless the image rows are very short, the “ping-pong” scheme will be about 11 per cent faster than the “standard” implementation (8 steps per pixel instead of 9). Speed tests should be made comparing these two implementations. Sample code is given for the main loop for both cases.

This is not one of the better applications for SKIPSM, because it requires that raw (i.e., unprocessed) pixel values be saved rather than processed results. In many other applications, it is the saving of partially-computed results in the state buffers that results in much of the speed improvement. It is not even clear that the SKIPSM implementation is faster than simply fetching, as needed, the pixel values from the input image. If there is any time saving here it all, it involves the fact that the SKIPSM buffer reads and writes involve one index i, whereas the image fetches involve two indices, i and j, which may be slower on some processors. On the other hand, the SKIPSM implementation (as always) allows the output to be written back into the same image buffer, without loss of information. This might be an advantage in some applications.

The application of SKIPSM to various 3x3 image processing operations

```c
// Roberts Cross code -- conventional
if (CS0[i] >= RS1) Tmp1 = CS0[i] - RS1;
else Tmp1 = RS1 - CS0[i];
R0 = Input[i][i];;
if (CS0[i-2] >= R0) Tmp2 = CS0[i-2] - R0;
else Tmp2 = R0 - CS0[i-2];
Output[i-1][i-1] = (Tmp1 + Tmp2)/2;
// or use Output = Max(Tmp1, Tmp2);

// Roberts Cross -- “ping-pong” EVEN rows
if (CS0[i] >= RS1) Tmp1 = CS0[i] - RS1;
else Tmp1 = RS1 - CS0[i];
R0 = Input[i][i];;
if (CS0[i-2] >= R0) Tmp2 = CS0[i-2] - R0;
else Tmp2 = R0 - CS0[i-2];
Output[i-1][i-1] = (Tmp1 + Tmp2)/2;
// or use Output = Max(Tmp1, Tmp2);
```

```c
// Roberts Cross -- “ping-pong” ODD rows
if (CS1[i] >= RS1) Tmp1 = CS1[i] - RS1;
else Tmp1 = RS1 - CS1[i];
R0 = Input[i][i];;
if (CS1[i-2] >= R0) Tmp2 = CS1[i-2] - R0;
else Tmp2 = R0 - CS1[i-2];
Output[i-1][i-1] = (Tmp1 + Tmp2)/2;
// or use Output = Max(Tmp1, Tmp2);
```
5. Horizontal and Vertical Gradient Operations

These are probably the most commonly used edge detectors in their own right, as well as being the basis for the Sobel edge detector. They use the familiar 1, 2, 1 Gaussian weighting pattern. See Figure 4a) and b). Notice the similarity in form between the row machine of one of these and the column machine of the other. Sample code for the main loops is given on the next page.

6. Direction of Brightest Neighbor

Finding the brightest pixel in a 3x3 neighborhood is relatively easy, with or without SKIPS. This operation is covered in 19 as a ranked filter of rank 9, and is not discussed further here. In contrast, finding the direction relative to the center pixel is both a useful and a rather difficult operation. Figure 5 shows a SKIPS implementation of this operation.
Finding the brightest neighbor involves a maximization or sorting operation. The real difficulty lies not in finding the maximum but in keeping track of where it came from during the sorting process. To that end, a “trick” is used here: To each grey-level value, a direction code is appended as the lowest bits. For the example shown here, it is assumed that the input pixels have 8 bits or fewer. In the first step, these are moved to the high byte of a 16-bit word, leaving the low byte equal to zero. Then, as the sorting progresses, appropriate direction codes are added to the low byte and carried along from then on. These do not affect the sorting process except in the case of “ties” in the high byte, in which case it doesn’t matter which direction “wins.” Preference is given in this implementation to the pixel with the higher direction code value. Finally, the low byte, containing the “winning” direction code, is taken as the output. Of course, other direction-value codes could be used instead of these.

A comment on the notation for this figure: To simply the figure, an abbreviated notation is used. Thus “PQR,012” means “the maximum of the values P, Q, and R in the high byte, and the corresponding direction 0, 1, or 2 in the low byte.” The row machine outputs RA and RB are shown with little three-box figures. These are reminders to the reader as to the pixels over which the maximization is taken. Sample code for the main loop is given below.

7. DIRECTION OF LARGEST GRADIENT

Finding the direction of the largest gradient in a 3x3 neighborhood is even more difficult than finding the direction of the brightest neighbor. Figure 6 shows a SKIPSM implementation of this operation. In this version, a “ping-pong” approach is used to reduce the number of “move” steps required to update the state buffers. As with the previous operation, the low byte is used to carry the direction information through the sorting process. A simplified definition of the gradient is used here – just the difference between symmetrically-placed pixels. Sample code for the main loop is given with the figure.

8. INITIALIZATION

Point 1: The column machine state buffers must be initialized at the beginning of the overall operation. The row machine state registers must be initialized at the beginning of each row.

Point 2: Because the output for a 3x3 operation is always written to the center of the 3x3 neighborhood, which is one pixel to the left of and one row above the current pixel, the output must not be written to the output image during the processing of the first row or the first pixel in each row. To do so would mean writing outside the defined image buffer.

These two “difficulties” can be overcome simultaneously by employing small preliminary chunks of code to use the first-row data to initialize the column machine buffers and the first-pixel-in-each-row data to initialize the row state registers. Of course, the output image is not written to during these steps. This code resembles the main loop code, but with all code references to data from the row above (or the pixel to the left) replaced by zeros, after which many of the statements can be deleted because they can be seen to do nothing. For example, because the input pixel values are non-negative, any addition or maximization step involving zero and any other number is equivalent to using that other number. The initialization details vary slightly from case to case, and are omitted here.
The application of SKIPSM to various 3x3 image processing operations

The Direction of largest gradient operation
Step // Inner-loop code for even rows
1 R2 = RS1;
2 RS1 = RS0;
3 RS0 = 256*Input[j][i];
4 if (CS0[i] >= CS2[i])
   A1 = CS0[i] - CS2[i];
   else A1 = 4 + CS2[i] - CS0[i];
5 if (R2 >= CS3[i])
   A2 = 3 + R2 - CS3[i];
   else A2 = 7 + CS3[i] - R2;
6 if (A1 >= A2)
   B1 = A1;
   else B1 = A2;
7 if (RS1 >= CS4[i])
   A1 = 2 + RS1 - CS4[i];
   else A1 = 6 + CS4[i] - RS1;
8 if (RS0 >= CS5[i])
   A2 = 1 + RS0 - CS5[i];
   else A2 = 5 + CS5[i] - RS0;
9 if (A1 >= A2)
   B2 = A1;
   else B2 = A2;
10 if (B1 >= B2)
   PreOut = B1;
   else PreOut = B2;
11 CS5[i] = R2;
12 CS4[i] = RS1;
13 CS3[i] = RS0;
// Set output = low byte of PreOut
14 Out[j-1][i-1] = PreOut & 00FF H
// For odd-numbered rows, replace // steps 4 through 13 with this code.
4 if (CS3[i] >= CS5[i])
   A1 = CS3[i] - CS5[i];
   else A1 = 4 + CS5[i] - CS3[i];
5 if (R2 >= CS0[i])
   A2 = 3 + R2 - CS0[i];
   else A2 = 7 + CS0[i] - R2;
6 if (A1 >= A2)
   B1 = A1;
   else B1 = A2;
7 if (RS1 >= CS1[i])
   A1 = 2 + RS1 - CS1[i];
   else A1 = 6 + CS1[i] - RS1;
8 if (RS0 >= CS2[i])
   A2 = 1 + RS0 - CS2[i];
   else A2 = 5 + CS2[i] - RS0;
9 if (A1 >= A2)
   B2 = A1;
   else B2 = A2;
10 if (B1 >= B2)
   PreOut = B1;
   else PreOut = B2;
11 CS2[i] = R2;
12 CS1[i] = RS1;
13 CS0[i] = RS0;

Figure 6. SKIPSM implementation of the direction of largest gradient operation.
9. SUMMARY AND CONCLUSIONS

This paper has shown that the SKIPSM paradigm can be used to implement a variety of standard binary, grey-scale, linear, and non-linear 3x3 image processing operations. Many operations in addition to those shown could also be implemented. The algorithms shown here give promise of being fast and efficient. Code comparisons with competing methods are difficult, because detailed code lists are usually not available. In the absence of such detailed information, proof of the speed of these implementations relative to conventional implementations can be obtained only by actual running-time tests on real machines.

10. BIBLIOGRAPHY


