SRC MAPstation™ Image Processing: Edge Detection

David Caliga, Director Software Applications
SRC Computers, Inc.
dcaliga@srccomputers.com

Motivations

The purpose of detecting sharp changes in image brightness is to capture important events and changes in properties of the world. It can be shown that under rather general assumptions for an image formation model, discontinuities in image brightness are likely to correspond to:

- discontinuities in depth;
- discontinuities in surface orientation;
- changes in material properties; and
- variations in scene illumination.

This algorithm is used in the medical imaging, SAR processing and target recognition domain areas, just to name a few.

The computational result of applying an edge detector to an image may lead to a set of connected curves that indicate the boundaries of objects, the boundaries of surface markings, as well curves that correspond to discontinuities in surface orientation. Thus, applying an edge detector to an image may significantly reduce the amount of data to be processed and may therefore filter out information that may be regarded as less relevant, while preserving the important structural properties of an image.

The SRC-7 Series H MAP® processor can yield more than two orders of magnitude performance improvements over a 3GHz Xeon microprocessor using Intel IPPLIB v5.1 image processing functions. This paper will discuss various implementation methods and optimizations of an edge detection application.

1. Approaches to Edge Detection

The edge detection methods that have been published mainly differ in the types of smoothing filters that are applied and the way the measures of edge strength are computed. While many edge detection methods rely on the computation of image gradients, they differ in the types of filters used for computing gradient estimates in the x- and y-directions.

1.1 Prewitt Edge Detection

The Prewitt edge detection filter uses the two 3x3 templates to calculate the gradient value. These two templates are shown below.

```
-1   0   1          1   1   1
-1   0   1          0   0   0
-1   0   1         -1  -1  -1
 X Template         Y Template
```

Consider the following 3x3 stencil out of the image array:

```
+------------+
| a00 a01 a02|
| a10 a11 a12|     <---   3 X 3 out of the larger 2D image array
| a20 a21 a22|
+------------+
```

where:

```
a00 .. a22   are the values of each pixel in the stencil window
```

When we do a matrix multiply of the X and Y templates on the stencil in the image we get the following:

```
X = -1*a00 + 1*a02 - 1*a10 + 1*a12 - 1*a20 + 1*a22
Y =  1*a00 + 1*a01 + 1*a02 - 1*a20 - 1*a21 - 1*a22
```

The Prewitt Gradient calculation is then:

```
Prewitt gradient = SQRT (X*X + Y*Y) / 4
```

1.2 Sobel Edge Detection

The Sobel edge detection filter uses these two 3x3 templates to calculate the gradient value.

```
-1   0   1          1   2   1
-2   0   2          0   0   0
-1   0   1         -1  -2  -1
 X Template         Y Template
```

Applying the templates to the image 3x3 stencil yields the following expression.

```
X = -1*a00 + 1*a02 - 2*a10 + 2*a12 - 1*a20 + 1*a22
Y =  1*a00 + 2*a01 + 1*a02 - 1*a20 - 2*a21 - 1*a22
```

The Sobel Gradient calculation is then:

```
Sobel gradient = SQRT (X*X + Y*Y)/4
```

1.3 Median Filter

Both the Prewitt and Sobel edge detectors start to break down if there is "noise" present in the image. A median filter can be used to effectively filter pixels that are much brighter or darker than their neighbors. As such, a median filter is a good method to remove noise from an image. The median filter will be applied to the 3x3 image stencil prior to the application of the edge detection filter.

2. Implementations

This section will review various optimization techniques on this image processing algorithm. The optimizations are easy to understand and straightforward to code.

2.1 Initial Implementation:

In order to run on the MAP, we will need to use a general subroutine structure used for MAP algorithms and add the median filter and edge detect codes to it. The code below shows the subroutine call, allocation of local arrays in the MAP On_Board Memories, and DMA calls to move data to and from the microprocessor.

General Infrastructure for MAP codes

```
#include "libmap.h"

#define REF(_A,_i,_j) ((_A)[(_i)*px+(_j)])

void edge(int px, int py, int sz,
                int64_t image_in[], int64_t image_out[],
                int64_t *t0, int64_t *t1,int64_t *t2, int64_t *t3, int mapnum)
{

// Store Arrays in On-Board Memory
  OBM_BANK_A(AL,int64_t,MAX_OBM_SIZE)
  OBM_BANK_B(BL,int64_t,MAX_OBM_SIZE)
  OBM_BANK_C(CL,int64_t,MAX_OBM_SIZE)


// DMA input image
    read_timer (t0);
    buffered_dma_cpu (CM2OBM, PATH_0, AL, MAP_OBM_stripe(1,"A"), image_in, 1, sz);
    read_timer (t1);

// ***********************************************
// Add computational Median Filter code here
// ***********************************************

// ***********************************************
// Add computational Edge Detect code here
// ***********************************************

// DMA output image
    read_timer (t2);
    buffered_dma_cpu (OBM2CM, PATH_0, CL, MAP_OBM_stripe(1,"C"), image_out, 1, sz);
    read_timer (t3);
}
```

The initial implementation of the median filter and edge detection codes are shown below. The function median_9 is finding the median value of nine supplied values.

Median Filter Code

```
for (i=0; i<py-2; i++)
for (j=0; j<px-2; j++) {
    a00 = REF (AL, i, j);
    a01 = REF (AL, i, j+1);
    a02 = REF (AL, i, j+2);

    a10 = REF (AL, i+1, j);
    a11 = REF (AL, i+1, j+1);
    a12 = REF (AL, i+1, j+2);

    a20 = REF (AL, i+2, j);
    a21 = REF (AL, i+2, j+1);
    a22 = REF (AL, i+2, j+2);

    median_9 (a00, a01, a02,
              a10, a11, a12,
              a20, a21, a22,
              &px);

    REF (BL, i, j) = px;
```

Edge Detect Code

```
for (i=0; i<py-2; i++)
for (j=0; j<px-2; j++) {
    b00 = REF (BL, i, j);
    b01 = REF (BL, i, j+1);
    b02 = REF (BL, i, j+2);

    b10 = REF (BL, i+1, j);
    b12 = REF (BL, i+1, j+2);

    b20 = REF (BL, i+2, j);
    b21 = REF (BL, i+2, j+1);
    b22 = REF (BL, i+2, j+2);

    hz = (b00 + b01 + b02) -
         (b20 + b21 + b22);
    vt = (b00 + b10 + b20) -
         (b02 + b12 + b22);

    px = sqrtf (hz*hz+vt*vt);
```

}
 	
	if ((i>=2) & (j>=2))
	REF (CL, i-2, j-2) =
	px > 255 ? 255 : px;
	}

Read or write access to an individual On-Board Memory can support one value every clock. When we compile the code above for the MAP, the Carte™ compiler states that the median filter loop is slowed down by eight clocks and the edge detect code is slowed down by 7 clocks due to all memory accesses using a single On-Board Memory Bank.

The performance timing for runs on the Series H MAP includes data movement (DMAs) of input and output data. As a result of these conflicts the following results are achieved compared to an AMD Dual Core Opteron 275 2.2 GHz with 4GB memory per dual core using Intel image processing functions from IPPLIB.

Performance Gain over Microprocessor	
Image 640 x 480	Image 1024 x 1024
.56x	.69x

2.2 Optimization 1: Flatten Loops

The advantage of flattening the loops is that the "cost of pipelining" is paid only once, and not on each time in a nested loop situation. The cg_count_ceil functions (SRC function) are used to generate the i and j indices of the code above. The code below shows our next optimization with the modified code in bold text.

Median Filter Code

```
for (n=0; n<((px-2)*(py-2)); n++) {
   cg_count_ceil_32 (1, 1, n==0, px-2,
                     &i);
   cg_count_ceil_32 (i==1, 1, n==0,
                      0xffffffff, &j);
    a00 = REF (AL, i, j);
    a01 = REF (AL, i, j+1);
    a02 = REF (AL, i, j+2);

    a10 = REF (AL, i+1, j);
    a11 = REF (AL, i+1, j+1);
    a12 = REF (AL, i+1, j+2);

    a20 = REF (AL, i+2, j);
    a21 = REF (AL, i+2, j+1);
    a22 = REF (AL, i+2, j+2);

    median_9 (a00, a01, a02,
              a10, a11, a12,
              a20, a21, a22,
              &px);

    REF (BL, i, j) = px;
}
```

Edge Detect Code

```
for (n=0; n<((px-2)*(py-2)); n++) {
   cg_count_ceil_32 (1, 1, n==0, px-2,
                     &i);
   cg_count_ceil_32 (i==1, 1, n==0,
                      0xffffffff, &j);
    b00 = REF (BL, i, j);
    b01 = REF (BL, i, j+1);
    b02 = REF (BL, i, j+2);

    b10 = REF (BL, i+1, j);
    b12 = REF (BL, i+1, j+2);

    b20 = REF (BL, i+2, j);
    b21 = REF (BL, i+2, j+1);
    b22 = REF (BL, i+2, j+2);

    hz = (b00 + b01 + b02) -
         (b20 + b21 + b22);
    vt = (b00 + b10 + b20) -
         (b02 + b12 + b22);

    px = sqrtf (hz*hz+vt*vt);

    if ((i>=2) & (j>=2))
       REF (BL, i-2, j-2) =
         px > 255 ? 255 : px;
}
```

The result of this optimization is only slightly better than the original.

Performance Gain over Microprocessor	
Image 640 x 480	Image 1024 x 1024
.56x	.70x

However, the flattening of the loop positions the code for the next optimization.

2.3 Optimization 2: Remove multiple accesses to On-Board Memories

The loop slow downs previously mentioned effectively slow the compute loops down by 8x and 7x respectively.

SRC has a set of macros that will use a set of FIFOs to create a "cache" of a set of data. The macro, delay_queue_8_var will take in a pixel every loop iteration and delay it by the width of a row (px). We will use these macros to cache two rows of pixels at a time. We will have to read two rows plus three values of row three before we have the values for 3x3 stencil. We can at this point compute our first median filter output. Each subsequent read will allow us to compute a new output point. Notice that the edge detect also utilizes the same technique to build up its "caching" of values going into the edge detect 3x3 stencil.

Median Filter Code

```
for (n=0; n<((px-2)*(py-2)); n++) {
   cg_count_ceil_32 (1, 1, n==0, px-2,
                     &i);
   cg_count_ceil_32 (i==1, 1, n==0,
                      0xffffffff, &j);

   mvalue = REF (AL, i, j);

   a20 = a21;
   a21 = a22;
   a22 = mvalue;

   a10 = a11;
   a11 = a12;
   delay_queue_8_var (a22,1,n==0,px,
                      &a12);

   a00 = a01;
   a01 = a02;
   delay_queue_8_var (a12,1,n==0,px,
                      &a02);

   median_9 (a00, a01, a02,
             a10, a11, a12,
             a20, a21, a22, &px);

// continued on column to right
```

Edge Detect Code

```
// continued from column on left

   b20 = b21;
   b21 = b22;
   b22 = px;

   b10 = b11;
   b11 = b12;
   delay_queue_8_var (b22,1,n==0,px,
                      &b12);

   b00 = b01;
   b01 = b02;
   delay_queue_8_var (b12,1,n==0,px,
                      &b02);

    hz = (b00 + b01 + b02) -
         (b20 + b21 + b22);
    vt = (b00 + b10 + b20) -
         (b02 + b12 + b22);

    px = sqrtf (hz*hz+vt*vt);

    if ((i>=2) & (j>=2))
       REF (BL, i-2, j-2) =
         px > 255 ? 255 : px;


}
```

Notice that this performance improvement is roughly 8x the previous optimization performance.

Performance Gain over Microprocessor	
Image 640 x 480	Image 1024 x 1024
5.02x	6.21x

2.4 Optimization 3: Streaming DMAs and Streams Processing

The Carte Programming Environment has support for "streams." A stream is a data structure that provides an extremely efficient communication mechanism between concurrent producer and consumer loops. A producer can

be a computation loop or a DMA and consumers can be a compute loop or a DMA. The performance advantage is that the consumer loop can consume a value in the stream as soon as it is produced. This provides the ability to overlap computation loops that are inherently sequential to be computed simultaneously. An additional benefit is the overlapping of DMAs and computation loops.

The buffered_dma_cpu call is replace by the streamed_dma_cpu. This will create a stream of 64b data values into stream S0. Carte uses parallel compute sections to create the "overlap" of the producer and consumer loops. The compute loop will get data values from the stream by calling get_stream on S0. Notice this replaced the read from On-Board Memory in the previous optimization step.

Median Filter Code

```
#pragma src parallel sections
{
#pragma src section
{
  streamed_dma_cpu (&S0, PORT_TO_STREAM,
               PATH_0, image_in, 1, sz);
}
#pragma src section
{

for (n=0; n<((px-2)*(py-2)); n++) {
   cg_count_ceil_32 (1, 1, n==0, px-2,
                     &i);
   cg_count_ceil_32 (i==1, 1, n==0,
                      0xffffffff, &j);

   get_stream (&S0, &mvalue);

   a20 = a21;
   a21 = a22;
   a22 = mvalue;

   a10 = a11;
   a11 = a12;
   delay_queue_8_var (a22,1,n==0,px,
                      &a12);

   a00 = a01;
   a01 = a02;
   delay_queue_8_var (a12,1,n==0,px,
                      &a02);

    median_9 (a00, a01, a02,
              a10, a11, a12,
              a20, a21, a22,
              &px);

// continued on column to right
```

Edge Detect Code

```
// continued from column on left

   b20 = b21;
   b21 = b22;
   b22 = px;

   b10 = b11;
   b11 = b12;
   delay_queue_8_var (b22,1,n==0,px,
                      &b12);

   b00 = b01;
   b01 = b02;
   delay_queue_8_var (b12,1,n==0,px,
                      &b02);

    hz = (b00 + b01 + b02) -
         (b20 + b21 + b22);
    vt = (b00 + b10 + b20) -
         (b02 + b12 + b22);

    px = sqrtf (hz*hz+vt*vt);

    if ((i>=2) & (j>=2))
       REF (BL, i-2, j-2) =
         px > 255 ? 255 : px;


}
```

Performance Gain over Microprocessor	
Image 640 x 480	Image 1024 x 1024
6.69x	8.28x

This optimization has effectively eliminated the cost of the DMA of the input data by overlapping it with the computation.

2.5 Optimization 4: Loop unrolling by eight using packed 8bit pixels in 64bit OBM access

A major advantage of MAP is that the Carte Programming Environment can "create" or instantiate as much computational parallelism in a pipelined loop as defined by the C or Fortran code. In this optimization step, we will unroll or replicate eight copies of the median filter and edge detect code. We utilize the function split_64to8 to unpack the 64b data value into eight pixel values. The computation for the eight unrolled copies will take place in basically the same amount of time as the computation of the previous code. We will then pack the eight computed pixel values into a 64b value using comb_8to64 and this 64b value will be put into a stream to be picked up by the edge detection computation. This code also utilizes code inlining for the edge detection functionality. In addition, inlining helps for code clarity.

Median Filter Code

```
#pragma src parallel sections
{
#pragma src section
{
  streamed_dma_cpu (&S0, PORT_TO_STREAM,
              PATH_0, image_in, 1, nbytes);
}
#pragma src section
{
px8 = px/8;
for (n=0; n<((px8)*(py-2)); n++) {
 cg_count_ceil_32 (1, 1, n==0, px8, &i);
 cg_count_ceil_32 (i==1, 1, n==0, py, &j);

 get_stream (&S0, &w1);

/*  | row
    || word
    ||| byte
    vvv
   v111  */

 a210=v0; a211=v1; a212=v2; a213=v3;
 a214=v4; a215=v5; a216=v6; a217=v7;
 split_64to8 (w1, &v7, &v6, &v5, &v4,
                  &v3, &v2, &v1, &v0);
 a218=v0; a219=v1;

 a110=v0; a111=v1; a112=v2; a113=v3;
 a114=v4; a115=v5; a116=v6; a117=v7;
 delay_queue_64_var (w1,1,n==0,px8, &w2);
 split_64to8 (w2, &v7, &v6, &v5, &v4,
                  &v3, &v2, &v1, &v0);
 a128=v0; a129=v1;

 a010=v0; a011=v1; a012=v2; a013=v3;
 a014=v4; a015=v5; a016=v6; a017=v7;
 delay_queue_64_var (w2,1,n==0, px8, &w3);
 split_64to8 (w3, &v7, &v6, &v5, &v4,
                  &v3, &v2, &v1, &v0);
 a018=v0;  a019=v1;

 median_8_9 (a010, a011, a012, a110, a111,
             a112, a210, a211, a212, &p0);
 median_8_9 (a011, a012, a013, a111, a112,
             a113, a211, a212, a213, &p1);
 median_8_9 (a012, a013, a014, a112, a113,
             a114, a212, a213, a214, &p2);
```

Edge Detect Code

```
// continued from column on left



#pragma src section
{
px8 = px/8;
for (n=0; n<((px8)*(py-2)); n++) {
 cg_count_ceil_32 (1, 0,  n==0, px8, &j);
 cg_count_ceil_32 (j==0, 0, n==0, py, &i);

 get_stream (&S1, &w1);

/*  | row
    || word
    ||| byte
    vvv
   v111  */

 a210=v0; a211=v1; a212=v2; a213=v3;
 a214=v4; a215=v5; a216=v6; a217=v7;
 split_64to8 (w1, &v7, &v6, &v5, &v4,
                  &v3, &v2, &v1, &v0);
 a218=v0; a219=v1;

 a110=v0; a111=v1; a112=v2; a113=v3;
 a114=v4; a115=v5; a116=v6; a117=v7;
 delay_queue_64_var (w1, 1, n==0, px8, &w2);
 split_64to8 (w2, &v7, &v6, &v5, &v4,
                  &v3, &v2, &v1, &v0);
 a128=v0; a129=v1;

 a010=v0; a011=v1; a012=v2; a013=v3;
 a014=v4; a015=v5; a016=v6; a017=v7;
 delay_queue_64_var (w2, 1, n==0, px8, &w3);
 split_64to8 (w3, &v7, &v6, &v5, &v4,
                  &v3, &v2, &v1, &v0);
 a018=v0;  a019=v1;

 edge_detect_8 (a010, a011, a012, a110, a111,
                a112, a210, a211, a212, &p0);
 edge_detect_8 (a011, a012, a013, a111, a112,
                a113, a211, a212, a213, &p1);
 edge_detect_8 (a012, a013, a014, a112, a113,
                a114, a212, a213, a214, &p2);
```

```
 median_8_9 (a013, a014, a015, a113, a114,
             a115, a213, a214, a215, &p3);
 median_8_9 (a014, a015, a016, a114, a115,
             a116, a214, a215, a216, &p4);
 median_8_9 (a015, a016, a017, a115, a116,
             a117, a215, a216, a217, &p5);
 median_8_9 (a016, a017, a018, a116, a117,
             a118, a216, a217, a218, &p6);
 median_8_9 (a017, a018, a019, a117, a118,
             a119, a217, a218, a219, &p7);

 comb_8to64 (p7,p6,p5,p4,p3,p2,p1,p0,&b1);

 put_stream (&S1, b1, 1);

}  // end parallel section

// continued on column to right
```

```
 edge_detect_8 (a013, a014, a015, a113, a114,
                a115, a213, a214, a215, &p3);
 edge_detect_8 (a014, a015, a016, a114, a115,
                a116, a214, a215, a216, &p4);
 edge_detect_8 (a015, a016, a017, a115, a116,
                a117, a215, a216, a217, &p5);
 edge_detect_8 (a016, a017, a018, a116, a117,
                a118, a216, a217, a218, &p6);
 edge_detect_8 (a017, a018, a019, a117, a118,
                a119, a217, a218, a219, &p7);

 comb_8to64 (p7,p6,p5,p4,p3,p2,p1,p0,&b1);

 ix = (i-4)*px8 + j-2;
 if ((i>=4) & (j>=2))  DL[ix] = b1;

} // end parallel section


} // end parallel region
```

The performance results for this optimization are:

Performance Gain over Microprocessor	
Image 640 x 480	Image 1024 x 1024
51.6x	65.66x

This optimization dramatizes the large performance improvements that can be achieved in MAP. We have put eight times more computation in the loop with virtually no increase in time to solution. The compiler will build all of the necessary operators and support logic to make this computation pipelined. The advantage that pipelining gives is that the latency to get the first value out is larger, but each subsequent loop iteration will produce eight computed pixel values.

2.6 Optimization 5: Loop unrolling by 16 using Streaming DMA of two 64bit values

We can extend the unrolling technique in the previous optimization and now have 16 sets of median filter and edge detection execution at the same time. In this case we are streaming in from Global Common Memory (GCM) and streaming out to microprocessor. This level of unrolling will come close to saturating the bandwidth to the MAP. We utilize a form of streams that can have two 64b data values "moving" in the stream. Note the new form of DMA at get_stream and put_stream in the code below.

Median Filter Code

```
#pragma src parallel sections
{
#pragma src section
{
  streamed_dma_gcm_128 (&S0,
PORT_TO_STREAM, PATH_0, image_in, 1,
nbytes);
}
#pragma src section
{
px16 = px/16;
for (n=0; n<((px16)*(py-2)); n++) {
 cg_count_ceil_32 (1, 1, n==0, px16, &i);
 cg_count_ceil_32 (i==1, 1, n==0, py, &j);

 get_stream_128 (&S0, &w1, &w2);
```

Edge Detect Code

```
// continued from column on left



#pragma src section
{
px16 = px/16;
for (n=0; n<((px16)*(py-2)); n++) {
 cg_count_ceil_32 (1, 0,  n==0, px16, &j);
 cg_count_ceil_32 (j==0, 0, n==0, py, &i);

 get_stream_128 (&S1, &w1, &w2);
```

```
// 16 median_8_9 calls


 put_stream_128 (&S1, b1, b2 1);

}  // end parallel section

// continued on column to right
```

```
// 16 edge_detect calls


 ix = (i-4)*px16 + j-2;
 if ((i>=4) & (j>=2))  {
   CL[ix] = b11;
   DL[ix] = b12;

} // end parallel section


} // end parallel region
```

Resulting performance for this optimization.

Performance Gain over Microprocessor	
Image 640 x 480	Image 1024 x 1024
154x	195x

This optimization is getting almost a 3x performance improvement over the previous optimization. This is arises due to fact that we are streaming the data in and out and overlapping the computation with this data movement.

2.7 Optimization 6: Loop unrolling by 32 using Streaming DMA of four 64bit values

The intent of this optimization is to show image processing capabilities that could be support image data coming in directly from a source into MAP. MAP has a 12 Gbyte/s GPIOX port that can connect to sensors that could input more than four 64b data values every clock.

This optimization is a further step on the previous optimization. We used Streaming DMA from and to Global Common Memory to get the maximum bandwidth with one GCM simulating an external sensor. We are also using streams that support "moving" four 64b data values.

Median Filter Code

```
#pragma src parallel sections
{
#pragma src section
{
  streamed_dma_gcm_256 (&S0,
PORT_TO_STREAM, PATH_0, image_in, 1,
nbytes);
}
#pragma src section
{
px32 = px/32;
for (n=0; n<((px32)*(py-2)); n++) {
 cg_count_ceil_32 (1, 1, n==0, px32, &i);
 cg_count_ceil_32 (i==1, 1, n==0, py, &j);

 get_stream_256 (&S0, &w1, &w2, &w3, &w4);


// 32 median_8_9 calls


 put_stream_256 (&S1, b1, b2, b3, b4, 1);

}  // end parallel section
```

Edge Detect Code

```
// continued from column on left









#pragma src section
{
px32 = px/32;
for (n=0; n<((px32)*(py-2)); n++) {
 cg_count_ceil_32 (1, 0,  n==0, px32, &j);
 cg_count_ceil_32 (j==0, 0, n==0, py, &i);

 get_stream_256 (&S1, &w1, &w2, &w3, &w4);


// 32 edge_detect calls


 ix = (i-4)*px32 + j-2;
 if ((i>=4) & (j>=2))  {
   CL[ix] = b1;
```

```
// continued on column to right
```

```
    DL[ix] = b2;
    EL[ix] = b3;
    FL[ix] = b4;

} // end parallel section


} // end parallel region
```

Performance Gain over Microprocessor	
Image 640 x 480	Image 1024 x 1024
192x	256x

The computation is able to work on four 64bit words every clock. The current level of bandwidth into the MAP from a single source is three 64bit words every clock. This is the reason that the performance gain did not increase by a factor of two compared to Optimization 5. Had a camera actually been connected to the GPIOX Port that could deliver four 64b words per clock, we would have seen a performance increase of 2x.

3. Resource Usage

FPGAs are an excellent medium to use when trying to expose as much parallelism as possible to the compiler. The optimization process is to do the maximum work possible until a resource is exhausted. The resources that could be exhausted are:

1. Memory bandwith to off chip memories or internal RAM blocks
2. Logic space in an FPGA(s) on a MAP.

The logic utlization in this exercise of a single Altera Stratix II S180 FPGA is shown in the following table.

Optimization Level	Percentage of Logic Utilization
Initial Implementation	16
Optimization 1 Flattened Loops	16
Optimization 2 Remove multiple accesses to On-Board Memories	16
Optimization 3 Streaming DMA and Stream Processing	16
Optimization 4 Loop Unrolling by 8	37
Optimization 5 Loop Unrolling by 16	47
Optimization 6 Loop Unrolling by 32	75

The use of streams meant that we did not require the use of off-chip memories or many of the internal RAM blocks.

4. Performance Summary

The step-wise performance gains shown in this study come from combinations of the following:

1. Eliminating conflicting On-Board Memory accesses using delay queues
2. Increased computational intensity in a pipelined loop via unrolling
3. Overlapping computations via streams
4. Overlapping DMAs and computation via streams

Optimization Level	Performance Improvement	
	640 x 480	1024 x 1024
Initial Implementation	.56	.69
Optimization 1 Flattened Loops	.56	.70
Optimization 2 Remove multiple accesses to On-Board Memories	5.0	6.2
Optimization 3 Streaming DMA and Stream Processing	6.7	8.3
Optimization 4 Loop Unrolling by 8	51.6	56.7
Optimization 5 Loop Unrolling by 16	154	195
Optimization 6 Loop Unrolling by 32	191	256

Performance gains using the MAP processor are obtained using the same step-by-step techniques already used by programmers on traditional CPUs. Methods such as loop unrolling and code inlining have been in widespread use for years, and the SRC Carte programming environment allows programmers to utilize these familiar techniques to easily obtain significant application performance on SRC systems.

Appendix A

The following code utilizes Intel IPPLIB version 5.1 and used in the timing comparisons.

```
    Ipp8u *pSrc;
    Ipp8u *pDst;

    IppStatus status;
    IppiPoint anchor = {1,1};
    IppiSize img={WA,HEIGHT}, roi = {WIDTH,HEIGHT}, mask={3,3};

    pSrc = (Ipp8u *) AA;    pDst = (Ipp8u *) BB;
    status = ippiFilterMedian_8u_C1R( pSrc, 1, pDst, 1, roi, mask, anchor);

    pSrc = (Ipp8u *) AA;    pDst = (Ipp8u *) BB;
    status = ippiFilterPrewittHoriz_8u_C1R( pSrc, 1, pDst, 1, roi);

    pSrc = (Ipp8u *) AA;    pDst = (Ipp8u *) CC;
    status = ippiFilterPrewittVert_8u_C1R( pSrc, 1, pDst, 1, roi);

    pSrc = (Ipp8u *) BB;    pDst = (Ipp8u *) AA;
    status = ippiSqr_8u_C1RSfs( pSrc, 1, pDst, 1, roi, 5);

    pSrc = (Ipp8u *) CC;    pDst = (Ipp8u *) DD;
    status = ippiSqr_8u_C1RSfs( pSrc, 1, pDst, 1, roi, 5);

    pSrc = (Ipp8u *) AA;    pDst = (Ipp8u *) DD;
    status = ippiAdd_8u_C1RSfs( pSrc, 1, pDst, 1, CC, WA, roi, 1);

    pSrc = (Ipp8u *) CC;    pDst = (Ipp8u *) AA;
    status = ippiSqrt_8u_C1RSfs( pSrc, 1, pDst, 1, roi, -4);

    pSrc = (Ipp8u *) AA;    pDst = (Ipp8u *) CC;
    status = ippiRShiftC_8u_C1R( pSrc, 1, 0,  pDst, 1, roi);
```

```
    res1 = (unsigned char *) CC;
    res2 = (unsigned char *) DD;
    for (i=0;i<HEIGHT*WA;i++) {
        v = (int)res1[i];
        res2[i] = v>255 ? 255 : v;
    }
```

References:

1. J. M. S. Prewitt, “Object enhancement and extraction”, *Picture Processing and Psychopictorics*, Academic Press, New York, 1970.

2. Sobel I (1978), “Neighborhood coding of binary images for fast contour following and general binary array processing”, *Computer Graphics and Image Processing*, 8: 127–135