Northwestern University, Dept. of ECE Technical Report, 1997.

Mapping Methods for the Chimaera Reconfigurable Functional Unit

Katherine Nelson, Scott Hauck
Northwestern University
Evanston, IL 60208 USA
kati@nwu.edu, hauck@ece.nwu.edu

Abstract

In this paper we examine how the Chimaera Reconfigurable Functional Unit can be used to speed up program execution. We investigate a graphical skeletonization (object thinning) algorithm for this purpose. First, we will show how an assembly language post-processor or very primitive compiler could automate the use of the Chimaera system. Second, we will demonstrate how a more sophisticated compiler can achieve superior performance though more complicated hardware mappings. Last, we will present the results of an implementation of the algorithm designed specifically for Chimaera to obtain the greatest speedups.

Introduction

A reconfigurable hardware co-processor, working in conjunction with a host processor, can perform computations that the main processor would ordinarily be required to calculate. This is advantageous for several reasons. First, if a set of instructions that is normally in software is converted into a "configuration" for the reconfigurable co-processor, the overhead of reading and decoding these instructions is eliminated. Second, since these computations have been mapped to hardware, they are performed much more quickly. Third, because this co-processor is reconfigurable, it may perform one set of functions at one point of the program, and a different set of functions elsewhere in the program. Thus the system can adapt not only to one program, but to different sections of a program.

The Chimaera System

The Chimaera system is a reconfigurable hardware co-processor (sometimes referred to as a "Reconfigurable Functional Unit" or RFU) which can perform computations at the bit level [1]. It is comprised of cells arranged in rows. To initialize the hardware to perform a specific computation, that computation's "configuration" is loaded into Chimaera. This configuration sets up the logic in the cells and the routing between them. The configurations themselves are row-based, with each configuration (multiple configurations are allowed in the RFU at a time) using as many complete rows as necessary. Each of these configurations is also known as an RFUOP (RFU Operation).

The number of cells per row is dependent upon the number of bits per word in the host processor, i.e. 32 cells per row for a 32 bit processor. The flow of information is from the topmost row in a configuration to the bottommost row.

Data is input to the RFU through up to nine of the CPU registers. In this paper we are using a 32-bit MIPS processor as an example, and the registers used by the RFU will be assumed to be t0 through t8. Up to two registers may be read by any one cell. Each cell can only read the bit of the register corresponding to its column number, i.e., the cells in column 4 can read bit number 4 from any two of the nine registers.

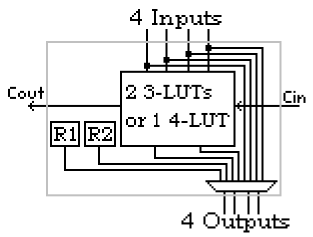


Figure 1: Chimaera Cell

The architecture of a cell itself can be thought of as either a set of 3-LUTs or (by using the two 3-LUTs in conjunction) a single 4-LUT, as shown in Figure 1. Each cell can also read from up to two registers in the CPU. The cell itself cannot use these values, but it can send them to a cell in the row below it.

Inputs to the cell consist of 4 main inputs plus a carry chain input from the cell to the right (LSB is rightmost). The four main inputs are of 3 different types. Two of these can only receive values from the cell directly above. Another input can receive values from any cell above it that is up to one column away, or from one of two longlines that span the length of the row. The last input can receive values from cells above that are up to three columns away, or from the other longline.

There are four main outputs from each cell, in addition to the carry chain output. The carry chain output is attached to one of the two 3-LUTs. The other four outputs mirror the four main inputs. Two of the outputs can only pass values to the cell below it, one can communicate with cells below it up to one column away or place its value on one of the longlines, and the other can communicate with cells below it up to three columns away or place its value on the other longline.

The four main outputs of a cell can each output one of the following: one of the four inputs, one of the two values read from the registers, the one result of a 4-LUT configuration, or one of the two results of a 2*3LUT configuration. This allows for great flexibility in computation, the passing through of values from earlier rows, and the reading of registers--especially considering the varied abilities of the output lines in communicating to cells in other columns.

Finally, the value generated in the last row of the configuration which is being executed is written back into the register specified when the RFU instruction was called. The RFU instruction itself is executed through the assembly instruction "RFUOP <configuration #> <destination register>". For example, "RFUOP 3 t6" would execute configuration 3 in the Chimaera system and place the result in register t6.

It is possible for a single configuration to contain more than one output row, as long as only one output row is used per execution of the configuration. This is accomplished through a flag value output by the most significant column in each of the output rows. Whichever output row contains a flag with a value of true is the row which is actually written to the destination register. Also, one configuration can share some or all of the computation rows of another configuration provided that the output row of each configuration is labeled with the configuration number that it belongs to. This is accomplished through the use of a content-addressable-memory (CAM), which holds the configuration number for each row. Therefore, both the CAM value must match the configuration number called and the flag value must be true for a given row to provide output to the register file.

It should also be noted that the architecture of the RFU is such that all configurations are constantly being executed with the values in the registers that they are configured to use. Calling a specific RFU instruction simply causes the result to be written to the destination register. Therefore, as long as the registers required by a particular RFU instruction are stable for long enough before the RFU instruction is called, the result is available immediately. This method of execution helps to avoid delays caused by propagating though many rows in a large configuration, provided that the registers are initialized well before the instruction is called.

The Skeletonization Problem

The skeletonization problem has already been implemented on the DISC reconfigurable system [2]. This algorithm is somewhat well suited to reconfigurable co-processors which work at the bit level, due to the bit-specific nature of the algorithms used. For this paper we will use the Zhang-Suen thinning algorithm [3], and the implementation presented in Practical Computer Vision Using C [4]. The relevant portions of the original source code from the book appear in Appendix A of this paper. The skeletonization algorithm takes in a picture with two colors, one of which is the "background color" and the other is the "foreground color". All areas which are in the "foreground color" are made narrower at each iteration of the algorithm until all of the foreground areas are only one pixel wide, as seen in Figure 2. The general shape and direction of lines are preserved. This is useful in Optical Character Recognition (OCR) software, in which a text character is recognized mainly by the relative positions of the pixels in the character, a task which is much simpler when the character in question is only composed of lines that are one pixel wide.

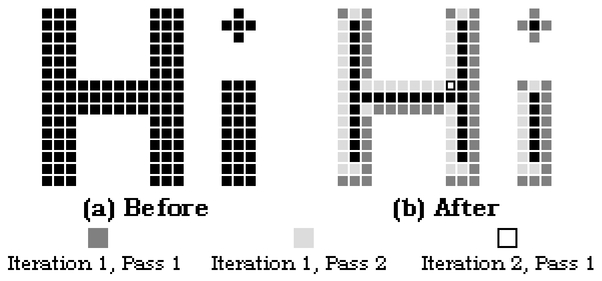


Figure 2: Skeletonization Example
Black pixels in (b) indicate result image. The other colors indicate the stage of the algorithm that the pixel was removed.

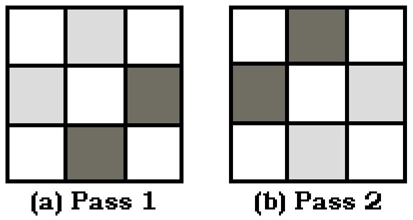
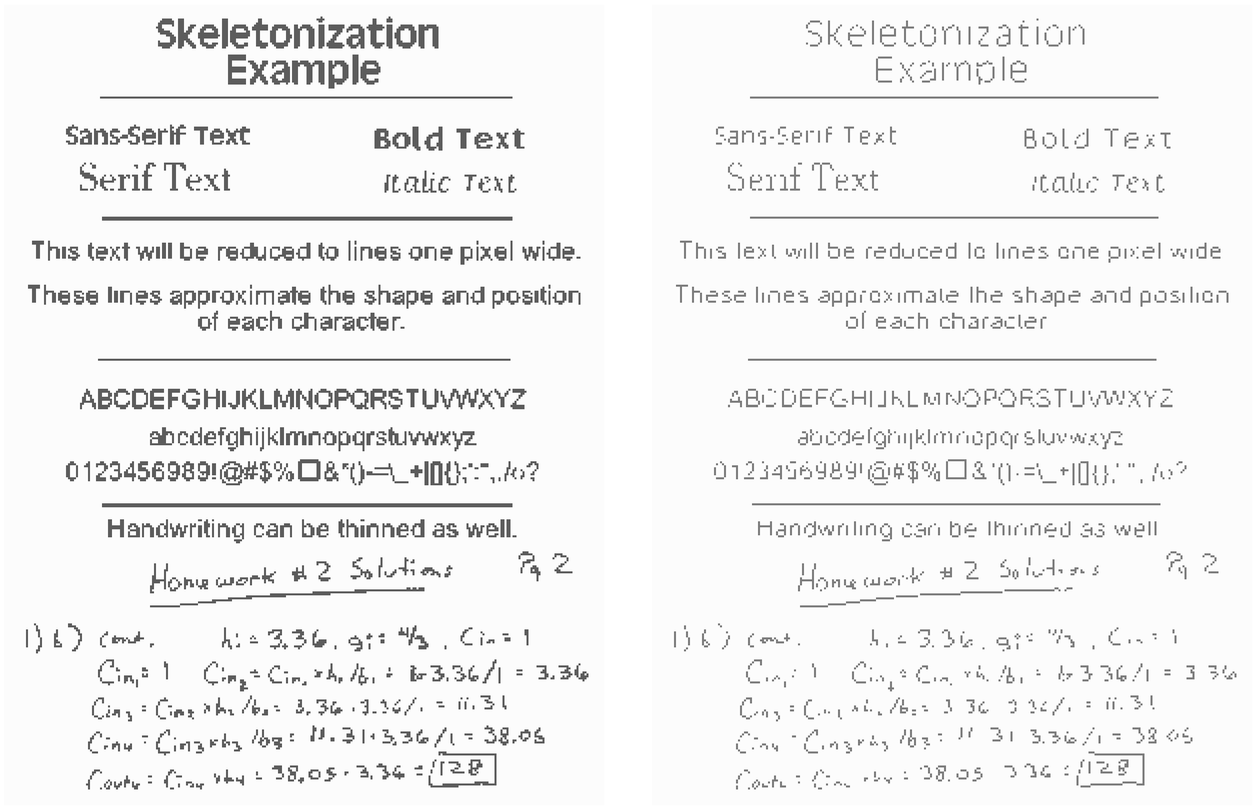


Figure 3: Pass Requirements
One of the requirements to "erase" a pixel is if either one of the dark grey pixels or both of the light grey pixels are the background value as shown in the diagram for the corresponding pass.

The Zhang-Suen algorithm for skeletonization is more complex than a simple erosion technique that merely removes successive layers of pixels from an object until the object is only one pixel wide at any given point. This algorithm ensures that the thinned object is at the center of the original object, that areas of the object which are connected remain connected, and that the end points of lines remain approximately intact. To accomplish this, foreground pixels are examined for "removal" (setting to the background color) based upon three criteria.

First, the number of surrounding foreground pixels must be appropriate. We do not want to remove pixels at the end of a line (1 neighboring foreground pixels), or which are in the middle of an object (totally surrounded by foreground pixels). Second, if we move clockwise around the pixel, there should be two locations where adjacent pixels differ in value. This is to ensure that we are at an edge of the object rather than at an intersection of lines or areas. Finally, we check if certain strategic pixels are already set to the background color.

There are two passes to the algorithm. In the first pass, one of the conditions in Figure 3a must be fulfilled. In the other pass, one of the conditions in Figure 3b must be fulfilled. This causes the picture to be eroded in a fashion which places the resulting thinned object at the center of the original object. Following these guidelines, we obtain a thinned object which is true to the general shape and location of the original object.


Skeletonization Example
Sans-Serif Text
Bold Text
Serif Text
Italic Text
This text will be reduced to lines one pixel wide.
These lines approximate the shape and position of each character.
ABCDEFGHIJKLMNOPQRSTUVWXYZ
abcdefghijklmnopqrstuvwxyz
Handwriting can be thinned as well.
Homework #2 Solutions
Pg 2

(a) Original Version (b) Thinned Version

Figure 4: The Test Image

Skeletonization With Chimaera

We will examine three different ways to adapt this algorithm to the Chimaera RFU hardware. The first is an example of how either a minimally Chimaera-aware C compiler or a Chimaera assembly code post-processor might make use of the hardware's capabilities. The second is an example of how a more intelligent C compiler might utilize the RFU. The last method to be discussed is how a knowledgeable programmer might re-write the algorithm with the Chimaera system in mind. This last method obviously requires more work on the part of the programmer, but also provides much larger speedups.

Throughout the remainder of this paper we will examine timing results based upon running the program on a "test image". The test image used appears in Figure 4a, and the thinned version appears in Figure 4b. These pictures have been reduced in size in the interests of space within this paper, but the original (and final) size of the picture when used was 600 pixels wide by 800 pixels tall.

We will show various Chimaera configurations using 8-cell wide rows, though the actual configurations are 32 cells wide. The narrower rows simplify the diagrams, and are easily extendable to 32 bits.

Speedup calculations are given based on total number of cycles, including the reading and writing of the .xbm file used, spent during execution as well as for just the thinning portion of the program. Including this overhead gives a more realistic view of the performance that we actually get using the Chimaera system.

Method I

A minimally Chimaera-adapted C compiler or an assembly language post-processor would examine the projected or existing assembly language of the program in order to find sections of instructions that can be easily mapped to the Chimaera system. For example, additions, subtractions, static-length-shifts and logic statements all function well within the Chimaera system. Variable length shifts, multiplications and divisions do not, as can be seen by an examination of the hardware's capabilities. It is by combining multiple Chimaera-friendly operations into one hardware configuration that we achieve good speedups.

Looking through the disassembled code for the skeletonization problem, we find only one section of code that looks promising at all. In the assembly source of the nay8() function are the instructions appearing in Figure 5a. This is a section of code for which there is only one way to enter (from the beginning), and one way to exit (from the end). Therefore, if we do not preserve the contents of the "t" registers, there will be no conflicts. First, we reorder the instructions to that seen in Figure 5b. Next we convert the marked instructions to an RFU configuration as shown in Figure 6. Finally, we replace those three instructions in the code with the single instruction "RFUOP 1, t7", as seen in Figure 5c, signifying that the first RFU configuration should be executed and the result written into the destination register t7.

```
lw    t2, 52(sp)        lw    t6, 48(sp)
lw    t3, 44(sp)        lw    t2, 52(sp)
lw    t6, 48(sp)        lw    t3, 44(sp)        lw    t6, 48(sp)
addu  t4, t2, t3        lw    t8, 8(t6)         lw    t2, 52(sp)
lw    t8, 8(t6)         lw    t0, 56(sp)        lw    t3, 44(sp)
sll   t5, t4, 2         lw    t1, 40(sp)        lw    t8, 8(t6)
lw    t0, 56(sp)        addu  t4, t2, t3        lw    t0, 56(sp)
lw    t1, 40(sp)        sll   t5, t4, 2         lw    t1, 40(sp)
addu  t7, t8, t5        addu  t7, t8, t5        RFUOP 1, t7
lw    t9, 0(t7)         lw    t9, 0(t7)         lw    t9, 0(t7)
```

(a) Original Code **(b) Reordered Code** **(c) Chimaera Code**

Figure 5: Selected nay8() Assembly Code

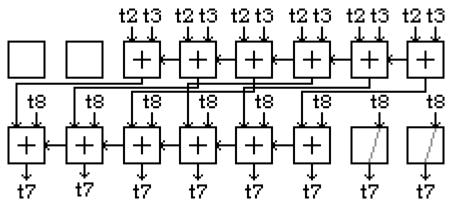


Figure 6: Method I, RFUOP 1 Configuration
Replacement for a portion of the nay8() assembly code.

As can be seen from Figure 6, the hardware configuration first adds together t2 and t3, the output of which is sent to the cell two columns to the left in the next row, where it is added with the value in t8. The result is written out to the t7 register, which is later compared to zero in software as a branch condition.

This gives us a saving of two instructions (3 instructions -> 1 RFU instruction) per execution of this section of code. In our test case, this section of code was executed 1,597,086 times, giving us a total savings of 2*1,597,086 = 3,194,172 cycles at one cycle per instruction. With a total number of 254,092,695 instructions executed by the thinning portion of the program and 508,155,578 total instructions executed in the original program, this optimization provides a speedup by a factor of only 1.013 over the relevant sections and 1.006 over the entire program. This lack of significant improvement is due to the memory-intensive aspect of this specific implementation of the algorithm, and the unsophisticated nature of the type of improvement allowed.

Method II

Using a more intelligent and powerful C compiler than the previous example, we believe that the C source itself can be examined for its intention, which could then be translated into a Chimaera RFU instruction. The clearest example of this approach is illustrated by the source code for the range() procedure, as it appears in Figure 7. This function simply returns a zero if either n or m is smaller than zero (negative), or larger than their particular boundaries in the picture structure (number of rows, number of columns). This code is only called from the nay8() function, whose code appears in Figure 8.

```
int range(struct image*x, int n, int m)
{  /* Return 1 if (n,m) are legal
      (row,column) indices for image X */
   if (n < 0 || n >= x->nr) return 0;
   if (m < 0 || m >= x->nc) return 0;
   return 1;
}
```

Figure 7: range() C Source Code

```
int nay8(struct image *x, int i, int j, int val)
{  /* return the number of 8-neighbors of (i,j) */
   int n,m,k;
   if (x->data[i][j] != val) return 0;
   k = 0;
   for (n=-1; n<=1; n++) {
       for (m=-1; m<=1; m++) {
           if (range(x, i+n, j+n))
               if (x->data[i+n][j+m] == val) k++;
   }   }
   return k-1;
}
```

Figure 8: nay8() C Source Code

Since the range() code is so simple, and is only ever called as it is from nay8, we can remove the function range() entirely, and replace the calling of it in nay8() with an RFUOP. Looking at the way that the range() function is called, we see that it uses n+i and m+j as its parameters. We can perform this addition in the configuration as well. Our input parameters are therefore i, n, j, m, nr, and nc, and our output value is zero if it is valid and non-zero if it is not valid. The relevant original assembly code of nay8() is listed in Figure 9a and the optimized version of the code appears in Figure 9b.

```
lw    t7, 52(sp)
lw    t9, 44(sp)
lw    t0, 56(sp)
addu  a1, t7, t9
lw    t9, -32620(gp)
lw    t1, 40(sp)
lw    a0, 48(sp)
jalr  ra, t9
addu  a2, t0, t1
lw    gp, 24(sp)
beq   v0, zero, 0x402634
nop
```

(a) Original Code

```
lw    t2, 48(sp)
lw    t7, 52(sp)
lw    t9, 44(sp)
lw    t3, 0(t2)
lw    t0, 56(sp)
lw    t1, 40(sp)
lw    t4, 4(t2)
RFUOP 1 t5
bne t5, zero, 0x402634
nop
```

(b) Chimaera Code

Figure 9: Selected nay8() Assembly Code
This section of code calls the range() function.

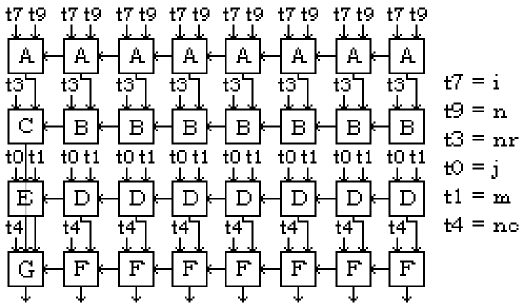


Figure 10: Method II, RFUOP 1
Replacement for the range() function.

We wish to test if i+n is less than zero or greater than nr, and if j+m is less than zero or greater than nc. We can accomplish this by first adding i and n, and checking the sign bit. If it is 1, then i+n < 0. Then, we subtract the sum from nr. If the resulting sign bit is 1, then i+j > nr. The same logic can be used for the j+m case. Essentially, we want to output the OR of the sign bits of these four computations. Then, the output of the configuration will be 0x0000 for valid values and 0x8000 for values which are not valid. This configuration, labeled here as RFUOP 1, is shown in Figure 10.

The hardware version of nay8 adds i and n from registers t7 and t9 in the first row (cells labeled "A"). In the second row, all B cells subtract the value from the row above from the nr value stored in register t3. They do not actually output the sum value, they just send the carry value along the carry chain. In the most significant cell of the same row, cell C, the same computation as the B cells is computed, and the resulting sign bit is checked to see if it is 1.

Also, the sign bit of the result (the most significant bit) computed in the A cells is checked. If either of these sign bits is a one, then the i+n value is out of range, and a 1 is output by this cell, otherwise a zero is output.

In the next row, the D cells compute the addition of the j and m values from the t0 and t1 registers. Cell E performs the addition of the most significant bit of the j and m values as well as passing through the result of the C cell untouched. The row below it is similar to the row containing the B cells and the C cell. Here, the F cells subtract the result of the D cells from nc, which is contained in register t4. The difference results are not output, but the carry results are sent along the carry chain. The actual output from each of these cells is 0. In cell G, we compute the same subtraction as the F cells, and examine the result, as well as the two outputs of the E cell. If the sum value outputted by cell E is 0 and the difference value calculated in cell G is 0, then the two word values which correspond to those bits are both positive and j+m is within range. The value generated by cell C which is passed through cell E indicates if i+n is within range. The output of cell F is the two outputs of cell E and the subtraction result from G itself OR'd together. Therefore, if any of these values are a 1, then there is some value, either i+n or j+m, which is out of range. If (i+n, j+m) is within range a zero is returned. Otherwise, a 1 is returned in the most significant bit.

Replacing an entire function call with a single RFU instruction removes a great deal of overhead in addition to reducing the amount of calculation that the CPU is required to perform. Before this optimization, this section of code in nay8() was 12 instructions long, and the range() function was required. After the optimization, the section of code in nay8() becomes 10 instructions long, and range() is eliminated. This is a savings of 2 instructions per time that the nay8() code is executed plus any cycles spent in the range() function, 2*1,597,086 + 25,553,376 = 28,747,548.

Another section of code that could be mapped to the Chimaera system through some analysis of the C code is the crosssing_index() function. This function, as seen in Figure 11, simply examines the eight neighboring pixels of the current pixel, looking at them in a clockwise manner and counting the number of times neighboring pixels differ, then dividing by two. This code is called only by the thinzs() routine. The pass one section of the assembly code for thinzs() calls the crossing_index function is shown in Figure 12a.

```
int crossing_index(struct image *x, int ii, int jj)
{   /* Compute crossing index for pixel X[ii][jj] and return it */
    int i,j,k,count;
    if ((ii<=0)||(ii>=x->nr-1)||(jj<=0)||(jj>=x->nc-1)) return -1;
    count = 0;
    i = ii-1;  j = jj-1;
    k = x->data[i][j];
    /* Move clockwise around the pixel, counting level changes */
    j++; if (k != x->data[i][j]) {k = x->data[i][j]; count++;}
    j++; if (k != x->data[i][j]) {k = x->data[i][j]; count++;}
    i++; if (k != x->data[i][j]) {k = x->data[i][j]; count++;}
    i++; if (k != x->data[i][j]) {k = x->data[i][j]; count++;}
    j--; if (k != x->data[i][j]) {k = x->data[i][j]; count++;}
    j--; if (k != x->data[i][j]) {k = x->data[i][j]; count++;}
    i--; if (k != x->data[i][j]) {k = x->data[i][j]; count++;}
    i--; if (k != x->data[i][j]) {k = x->data[i][j]; count++;}
   return count/2;
}
```

Figure 11: crossing_index() C Source Code

The RFU instruction can behave somewhat differently from the original C function in output. The original function output the number of times the pixel values changed when moving clockwise around the pixel being examined. In the thinzs() code, this value was then tested to see if it was equal to 1. The RFU instruction simplifies the output to 0 if it is a valid crossing_index value, and non-zero if it is not a valid value. Furthermore, since the number of level changes is always even, we can perform only 7 of the 8 comparisons, and allow "good" values to be either 1 or 2. We know that the last two pixels will be different if we have an odd result, and that they will be the same if we have an even result.

It should be noted that with the code given in Practical Computer Vision Using C, the integer value 0 is used for foreground color, and the integer value of 255 is used as the background color. This knowledge was used in the creation of the crossing_index RFU instruction, in that comparisons can be made between two pixels at any bit position lower than 8, given that a background pixel is all 1's and a foreground pixel is all 0's.

```
                            lw    t5, 52(sp)      lw    t2, 8(t9)
                            lw    t4, 36(sp)      addu  t9, t7, t6
lw    t9, -32612(gp)        addiu t5, t5, -1      lw    t7, 0(t9)
lw    a0, 36(sp)            sll   t4, t5, 2       lw    t4, 8(t8)
lw    a1, 52(sp)            lw    t2, 8(t4)       lw    t3, 8(t9)
lw    a2, 48(sp)            lw    t6, 48(sp)      addu  t9, t4, t6
lw    a3, 64(sp)            addu  t8, t2, t4      lw    t6, 0(t9)
jalr  ra, t9                lw    t7, 0(t8)       lw    t5, 4(t9)
nop                         addiu t6, t6, -1      lw    t4, 8(t9)
lw    gp, 24(sp)            addu  t9, t7, t6      RFUOP 2 t9
li    at, 1                 lw    t0, 0(t9)       bne   t9, zero, 0x402da8
bne   v0, at, 0x402da8      lw    t1, 4(t9)       nop
nop                         lw    t7, 4(t8)
```

(a) Original Code **(b) Chimaera Code**

Figure 12: Selected thinzs() Assembly Code

This section of code calls the crossing_index() function. The Chimaera-optimized code for this section of the program is longer than the original because instructions from the crossing_index() function which would read memory locations must be moved into the thinzs() code in order to eliminate the crossing_index function.

To compare pixel values, we XOR them. If the result is a 1, the pixels are different, and if the result is a 0, the pixels are the same. We then test if the sum of these results for each of the 7 comparisons is equal to 1 or 2, in which case we output all zeros. Otherwise, we want to output some non-zero value.

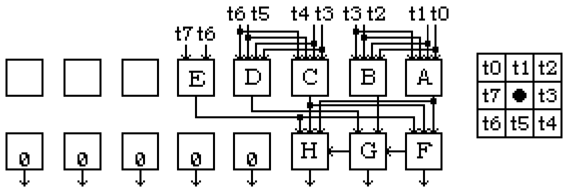


Figure 13: Method II, RFUOP 2

Replacement for the crossing_index() function.

The crossing index hardware, shown in Figure 13 as RFUOP 2, functions by XOR'ing the pixels together which are next to each other, then adding the results of the XORs together. The pixel values contained in t0, t1, t2, and t3 are all compared and the results added together in the cells labeled "A" and "B". The sum bit of the addition of the three values (t0 XOR t1), (t1 XOR t2), and (t2 XOR t3) is the output of cell A, and the carry of the calculation is the output of cell B. The pixel values contained in t3, t4, t5, and t6 are compared and added in the same manner in cells C and D. In cell E, t6 is XOR'd with t7. In the second row, zeros are output to the register file from every cell except the cells labeled "F", "G", and "H".

The output to the register file from cell F is the equation E*C*A. This is a test to see if there are less than three 1's in the sum results of the previous row. Cell F also places the results of the function E*A + C*A + E*C onto the carry chain. This value is the carry result of the addition of the A, C, and E values, which are the least significant bits of the addition of all the XOR'd values. The output of cell G is the equation Cin*B + D*B + Cin*D, where Cin is the carry chain value output from cell F. This function tests to see if there is a carry out of the second-to-least significant bit of the addition of all the XOR'd values., which would indicate a value higher than three, whereas a value higher than two is not allowed. The carry out of cell G is the function D + B, a test for at least one carry value from the first row. Cell H outputs the result of /E*/Cin*/C*/A + E*Cin + A*Cin + C*Cin. Originally, this function also had E*C*A as part of the sum of products calculation, but since this value is checked for in cell F, it is unnecessary here. This function tests to see if the sum of the A, C, and E values are greater than zero if the carry out of cell G is also zero. With these three tests in the second row, the value output to the register file will be zero if the crossing index value that we have calculated is either one or two, but some non-zero value in any other case.

Implementing the entire crossing_index() function as a single RFUOP removes the overhead of a function call in the calling code as well as eliminating the time spent in the function itself, though some of the memory references in the crossing_index() function would need to be moved to the thinzs() routine in order to access the pixel values. The new version of the thinzs code in Figure 12a are shown in Figure 12b. We have increased the number of instructions in this section of code from 11 to 25, and completely removed the call to crossing_index. Although in our test case the number of cycles spent in this particular section of code has increased by 83,228*(25-11) = 1,165,192 per pass, we have eliminated the 34,324,640 cycles spent in the crossing_index function itself for a total savings of 36,655,024 cycles.

Using this "intelligent compiler" approach, we have replaced two function calls with RFU instructions, reducing the total number of cycles spent running the thinning algorithm on the test image from 254,092,695 to 188,690,123. This provides a speedup of 1.347 over the original skeletonization code, and 1.148 over the entire program. This is a 33% improvement over the previous method's speedup in the skeletonization code itself.

Method III

The last method of adapting the skeletonization algorithm to the Chimaera system that we will examine is that of the Chimaera-aware programmer. For this example, the skeletonization code was completely rewritten with the RFU in mind as an available resource. The use of the Chimaera system was simulated by writing a function RFUOP, which performs the calculations intended to be mapped into the Chimaera hardware. This code was compiled and executed with the same test picture as the other examples. The source was then disassembled and partially re-written to reflect the use of the RFU. The relevant portions of the C source for this implementation appear in Appendix B.

We have chosen to store the image in a more compressed format than in <u>Practical Computer Vision Using C</u>, since a pixel can only be one of two values for the skeletonization routine (foreground or background). Assuming a 32 bit host processor, we will store 32 pixels per word in a straight binary format. The Chimaera configurations are shown for 8 bits to conserve space and preserve simplicity, but are all scalable to the 32 bit version used to obtain the timing results.

A static size limit of 2048 x 2048 pixels has been chosen for the picture to be thinned. This limit is sufficiently large for most pictures, and has the advantage that memory address calculation will always be simple. A row of the picture always uses (2048/32)*4 = 256 bytes, so choosing values from neighboring rows only involves using an offset of 256 in the memory address.

As explained in the description of the skeletonization algorithm, there are three steps taken in order to determine if a pixel can be changed from the foreground color to the background color. First, the pixel must have between 2 and 6 (inclusive) neighbors that are foreground pixels. Second, there can be only one set of changes in pixel value as the pixels surrounding the main pixel are examined in a clockwise order (a crossing index of 1). Last, depending on which pass of the algorithm is being executed, certain surrounding pixels must have the background color value. In addition to these main steps is a more general constraint that pixels on the outermost edges of the picture may not be altered.

When we examine the calculations that need to be performed on every pixel, with the values of that pixels surrounding pixels as parameters, we quickly find that a single column per pixel is not enough computation room. We need to count 8 values, add together the results of eight comparisons, and look at four specific surrounding pixels to determine eligibility. The pixels within a byte or word must be separated into 2 or more sections in order to provide enough computation width. We found that an odd/even approach functioned quite well by minimizing the number of sections (1 odd, 1 even), keeping the number of rows per configuration reasonable in size, and reducing the number of memory accesses.

Another challenge in the computation is that of the end bits, the least significant bit and the most significant bit of a word. Each of these requires values from another column of words in order to perform the computations (Figure 14). Splitting the bits into odd and even sections allows us to deal with only one end bit problem per section, the most significant bit for the odd section and the least significant bit for the even section. With this method, we are only required to have six word values in memory at a time instead of nine. For the odd bits, we require the current word, the words above and below it, the word to the left, and the words above and below the word to the left. For the even bits, we require the mirror image of the words for the odd bits, which is the three words in the "center" and the three words on the "right".

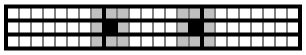


Figure 14: End Bits Problem Illustration
The end bits of each word must access values from each of three other words in order to perform the computations which rely on the values of neighboring pixels.

For each word, the registers are loaded for the odd operations which are then performed. This means that if any of the odd bits qualify, they will be set to the background value. The even bits remain unchanged. Next, the three registers which hold words in the same column as the word being examined are shifted over to the three registers to the left, and new values are loaded into the registers on the right, resulting in the correct register configuration for the even operations, which are then performed. At this point, the registers are already in the proper positions for the odd operations on the next word. The cycle repeats--odd, shift, even, odd, shift, even, odd, shift, even, etc., until all words are finished being processed.

The first and last word of each row are somewhat of a special case, since for the odd operation on the first word and the even operation on the last word there is not a word "next to" the word in question. This is not a problem, however, due to the constraint that the edge pixels of the picture cannot be altered -- these pixels are not candidates, and so should be ignored.

When we map the three aspects of the skeletonization algorithm to the Chimaera architecture, we find that we are able to combine the neighbor count mapping with the pass-specific test for neighboring background pixels. We are not, however, able to also add the crossing index test into this conglomerate, due to the nature of the routing of these mappings. Both the eight neighbor count and the crossing index test require that the cells in one of the rows of the configuration all have four inputs. As a result, one of the configurations could not be placed above (earlier in the computation) than the other, because the results could not be passed through the computations of the other mapping. The result of one of the tests needs to be written out to a register, then input into the other hardware function after the bottleneck.

In order to transfer the results of the crossing index configuration to the neighbor count and background pixel test configuration, we have chosen to use a mask value. The crossing index sets this mask value to 1 for the bits that have a valid crossing index and are foreground pixel. Pixels that either do not have a valid crossing index or are already background have their bit position in the mask set to 0. This value is written out to the register file. When the neighbor count configuration is run, after the neighbors are counted and the correct neighboring pixels are checked for the background value, the results of these tests (1 for TRUE or VALID, 0 for FALSE or NOT VALID) are AND'd together with the correct bit position from the mask register. This final value is a TRUE/FALSE result that indicates if the pixel should be set to the background color.

We can also use this mask to prevent the configurations from altering pixel values at the very edges of the picture. All we need to do is to input to the crossing index an initial mask that is 0x7fff for the first column, 0xffff for the middle columns, and the appropriate value for the end column, depending on the number of pixels in the picture. The crossing index configuration then performs an AND of its result with the initial mask to obtain the final mask value which is output to the register file.

The intent of the crossing index function is to examine the surrounding pixels in clockwise order and add up the number of times the pixel values change from background to foreground or the reverse. Therefore, we wish to find the sum of the XORs of each set of neighboring pixels using the clockwise order, and output a 1 if the value is either 1 or 2 (we are using the 7 comparison approach outlined in Method II). Then, this value is AND'd with the initial mask value and the pixel value (foreground pixels have a value of 1). This computation is shown for one of the even bits in Figure 15.

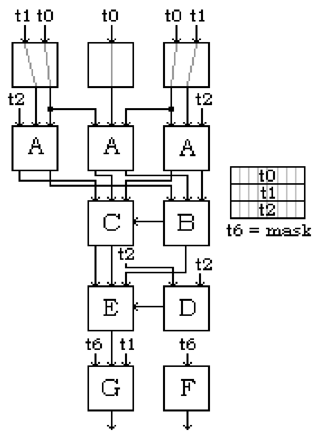


Figure 15: The Crossing Index Mapping for One Even Bit

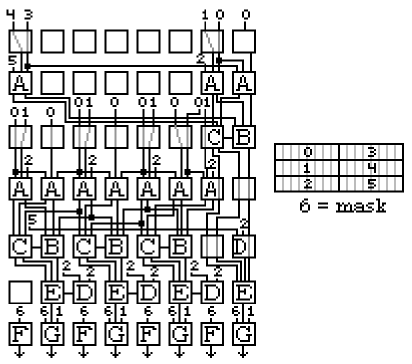


Figure 16: Method III, RFUOP 2
The full crossing index mapping.

In each column, three adjacent pixels to the pixel being tested are compared. In the left column the three bits to the left are examined, in the right column the three bits to the right are examined, and in the center column the three pixels above are examined. The cells labeled "A" compute the function (x XOR y) + (y XOR z), where "+" denotes an addition, y is the middle pixel of the three, and x and z are the pixels to either side. In the next row, cell B computes an addition of the sum bits from the A cells. The C cell computes the addition of the carry bits from the A cells. Cell D inputs the pixel bit below the pixel in question and the bit to the lower right, and performs an exclusive or of these two bits. This value is sent along the carry chain to cell E, which also takes as input the outputs of cells B and C. Cell E computes the function /C2*/C1*Cin + /C2*/C1*B + /C2*C1*/B*/Cin, where Cin is the value passed in from cell D, C2 is the carry bit from cell C, and C1 is the sum bit from cell C. This value is 1 if the sum of all of the XORs is either 1 or 2, or 0 otherwise. This value is then AND'd with the mask value and the pixel value for the bit position that we are looking at in cell G. Cell F simply reads in the value for the mask for that bit position, and outputs that value.

The full crossing index configuration appears in Figure 16 for the even bits. Note that the t1 register holds the set of pixels that we are currently altering. The odd bit version is essentially the same as the even bit version, with the values offset by one, and the extra computation for the most significant bit instead of the least significant bit. This "complete" crossing index configuration is basically the configuration shown in Figure 15 repeated across the columns, with some extra logic to compensate for columns on the edge (column 0 for the even bits, column 7 for the odd bits). We have labeled these configurations in the CAM such that the odd bit computation is RFUOP 1 and the even bit computation is RFUOP 2.

In order to determine the neighbor count, we simply add up the values of the eight neighboring pixels, since foreground pixels have a value of 1. The pass dependent background pixel check tests the appropriate pixels from Figure 3 to see if one of the conditions is fulfilled. These two computations are shown in Figure 17 for one of the even bits. Note that we are using Chimaera’s ability to share logic in different computations. The row labeled "Pass 1" has a different RFUOP number than the row labeled "Pass 2".

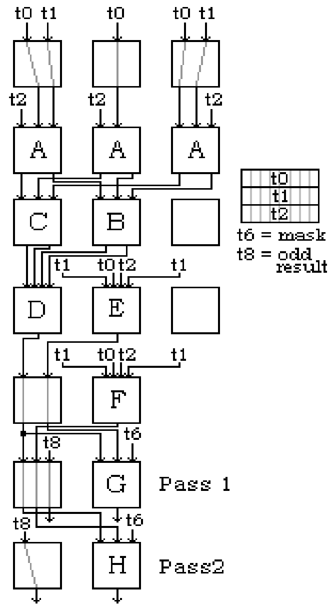


Figure 17: The Neighbor Count and Pass Check for One Even Bit

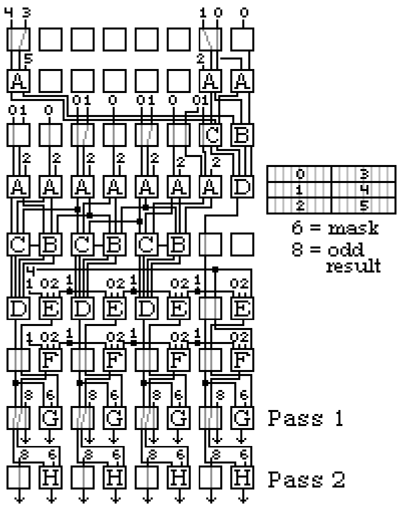


Figure 18: Method III, RFUOP 4 and RFUOP 6
This is the full mapping for the neighbor count and the pass-specific background pixel check. The "Pass 1" row is RFUOP 4 and the "Pass 2" row is RFUOP 6 in the CAM.

In cells labeled "A", the pixel values for the current word are added to the pixel values for the word above and the word below in each bit position. In cell B, the sum bits from the A cells are added together. In cell C, the carry bits from the A cells are added together. Cell D inputs the carry and sum bits from cells B and C, and outputs the function /(/C2*/C1*/B2 + C2*C1*B1 + C2*C1*B2 + C2*C1*B1), where B1 and C1 are the sum bits of the outputs of cells B and C, and B2 and C2 are the carry bits of the outputs of cells B and C. This results in an output of 1 if the number of neighbors, given from the outputs of cells B and C, is greater than or equal to 2 and less than or equal to six.

Cell E computes the pass one eligibility by checking the relevant pixels to see if they are background, and cell F computes the pass two eligibility by checking background pixels. Cell G computes the actual resulting pixel value for the first pass, incorporating the results of cell D and cell E with the mask value (output of the crossing index configuration) and the original value for the pixel we are examining. Cell H computes the actual resulting pixel value for the second pass, using the same parameters as cell G except with cell F (second pass eligibility) instead of cell E. The output row for pass 1 is therefore the row with G and the output row for pass 2 is the row with H. Since this is the result of the even bit computation, which is to be performed after the odd bit computation, we incorporate the results of the odd computation by outputting the odd bits of t8 for the result row of both passes.

The full combined neighbor count and background pixel check configuration for the even bits is shown in Figure 18. The t1 register holds the pixels that we wish to alter. The odd bit version is only a slight variation. Again, this is the one-pixel version repeated across the row with some extra logic for the edges. The odd bit configuration is labeled in the CAM as RFUOP 3 for pass 1 and RFUOP 5 for pass 2. The even bit configuration is labeled RFUOP 4 for pass 1 and RFUOP 6 for pass 2.

The new assembly code for the pass one portion of the code appears in Appendix C, along with indicators as to how many times each section of the code is executed. The entire replacement assembly code (including both passes) uses 4,762,519 cycles in its execution. The non-optimized assembly code used 22,344,055 cycles, an improvement of 22,344,055 - 4,762,519 = 17,581,536 cycles in the skel() code itself. In addition, we eliminate the cycles spent in the RFUOP() function entirely, for a total improvement of 753,145,214 cycles, and a total of 815,888,093 - 753,145,214 = 62,742,879 cycles of execution time for the entire program for our test image. Comparing our total execution time with the execution time of the original software-only implementation of the algorithm, we see a speedup of a factor of 8.099. Looking at just the skeletonization times, the speedup jumps to 53.353. This is a 5167% improvement over the first method, and a 3861% improvement over the second method.

Conclusion

Different methods of optimizing programs to the Chimaera architecture provide varying levels of improvement. As expected, if we rely on a primitive post-processor to sift though already assembled code, the speedup provided is unremarkable. We fare better with a more intelligent compiler which could analyze the code and its context prior to compilation in order to shrink or eliminate larger sections of code. Both of these methods have the advantage that the programmer need not know how to use or even be aware of the Chimaera system in order to use it. However, the situation in which the programmer is fully aware of the reconfigurable hardware and designs the program to make good use of it is the one in which we obtain the most improvement over the non-optimized code.

References

[1] S. Hauck, T. W. Fry, M. M. Hosler, J. P. Kao, "The Chimaera Reconfigurable Functional Unit", *IEEE Symposium on FPGAs for Custom Computing Machines*, pp. 87-96, 1997.

[2] M. J. Wirthlin, B. L. Hutchings, "Sequencing Run-Time Reconfigured Hardware with Software", *ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, pp. 122-128, 1996

[3] T. Y. Zhang, C. Y. Suen, "A Fast Parallel Algorithm for Thinning Digital Patterns", *Communications of the ACM*, pp. 236-239, 1984.

[4] J. R. Parker, Practical Computer Vision Using C, John Wiley & Sons, Inc., 1994

Appendix A: Implementation of the Zhang-Suen skeletonization algorithm from Practical Computer Vision Using C.

```
/*  The IMAGE data structure        */
struct image
{   int             nr, nc;           /* rows, columns */
    unsigned char   **data;           /* Pixel values */
};

void main(int argc, char *argv[])
{   char            input[30], output[30];
    struct image    *x;
    int             error_code;
    int             i, j;

    if (argc != 3) return;
    strcpy(input, argv[1]);
    strcpy(output, argv[2]);
    readimage(&x, input, &error_code);
    if (error_code == 0)
    {  thinzs(x, 0, &error_code);
        if (error_code == 0)
        {  writeimage(x, output, &error_code);
           if (error_code != 0) an_error(error_code);
        }
        else an_error(error_code);
    }
    else an_error(error_code);
}

/*  Return TRUE (1) if (n,m) are legal pixel coordinates
    for the image X, and return FALSE (0) otherwise.        */
int range (struct image *x, int n, int m)
{   if (n < 0 || n >= x->nr) return 0;
    if (m < 0 || m >= x->nc) return 0;
    return 1;
}

/*  Return the number of 8-connected neighbors of (i,j) having value VAL */
int nay8 (struct image *x, int i, int j, int val)
{   int n,m,k;
    if (x->data[i][j] != val) return 0;
    k = 0;
    for (n= -1; n<=1; n++) {
        for (m= -1; m<=1; m++) {
        if (range(x,i+n, j+m))
          if (x->data[i+n][j+m] == val) k++;
        }
    }
    return k-1;
}

/*      Compute the crossing index for pixel (ii, jj)   */
int crossing_index(struct image *x, int ii, int jj, int *error_code)
{   int i,j,k, count;

    *error_code = 0;
    if ( (ii<=0)||(ii>= x->nr-1)||(jj<=0)||(jj>=x->nc-1) ){
        *error_code = NO_RESULT;
        return -1;
    }
    count = 0;
    i = ii-1; j = jj-1; k = x->data[i][j];

/*      Move clockwise around the (II,JJ) Pixel, counting level changes  */
    j++;    /* Move to (i-1,j) */
    if (k != x->data[i][j]) { k = x->data[i][j]; count++; }
    j++;    /* Move to (i-1,j+1) */
    if (k != x->data[i][j]) { k = x->data[i][j]; count++; }
    i++;    /* Move to (i,j+1) */
    if (k != x->data[i][j]) { k = x->data[i][j]; count++; }
    i++;    /* Move to (i+1,j+1) */
    if (k != x->data[i][j]) { k = x->data[i][j]; count++; }
    j--;    /* Move to (i+1,j) */
    if (k != x->data[i][j]) { k = x->data[i][j]; count++; }
    j--;    /* Move to (i+1,j-1) */
    if (k != x->data[i][j]) { k = x->data[i][j]; count++; }
    i--;    /* Move to (i,j-1) */
    if (k != x->data[i][j]) { k = x->data[i][j]; count++; }
    i--;    /* Move to (i-1,j-1) */
    if (k != x->data[i][j]) { k = x->data[i][j]; count++; }
    return count/2;
}

/*      Zhang-Suen type of thinning procedure. Thin the region VAL      */
void thinzs (struct image *x, int val, int *error_code)
{   int i,j,n, again;
    struct image *y;

    y = 0;
    copy (x, &y, error_code);
    if (*error_code) return;

    do {
        again = 0;
        for (i=1; i<y->nr-1; i++)
            for (j=1; j<y->nc-1; j++)  {
                if (y->data[i][j] != val) continue;
                n = nay8(y, i, j, val);
                if ( (n>=2) && (n<=6) ) {
                    if (crossing_index(y,i,j,error_code)==1) {
                        if ( (y->data[i-1][j]==BACKGROUND) ||
                        (y->data[i][j+1]==BACKGROUND) ||
                        (y->data[i+1][j]==BACKGROUND) ) {
                            if ( (y->data[i][j+1]==BACKGROUND) ||
                            (y->data[i+1][j]==BACKGROUND) ||
                            (y->data[i][j-1]==BACKGROUND) ) {
                                x->data[i][j] = BACKGROUND;
                                again = 1;
                            }
                        }
                    } else if (*error_code) { freeimage(y, &again);  return; }
                }
            }

        copy (x, &y, error_code);
        if (*error_code) { freeimage(y,&again);  return; }

        for (i=1; i<x->nr-1; i++)
            for (j=1; j<x->nc-1; j++)  {
                if (x->data[i][j] != val) continue;
                n = nay8(x, i, j, val);
                if ( (n>=2) && (n<=6) ) {
                    if (crossing_index(x,i,j,error_code)==1) {
                        if ( (x->data[i-1][j]==BACKGROUND) ||
                        (x->data[i][j+1]==BACKGROUND) ||
                        (x->data[i][j-1]==BACKGROUND) ) {
                            if ( (x->data[i-1][j]==BACKGROUND) ||
                            (x->data[i+1][j]==BACKGROUND) ||
                            (x->data[i][j-1]==BACKGROUND) ) {
                                y->data[i][j] = BACKGROUND;
                                again = 1;
                            }
                        }
                    } else if (*error_code) { freeimage (y, &again);  return; }
                }
            }
        copy (y, &x, error_code);
        if (*error_code) { freeimage (y, &again);  return; }
    } while (again);
    freeimage (y, error_code);
}
```

Appendix B: Implementation of the Zhang-Suen skeletonization algorithm for Chimaera, using the C function RFUOP to simulate a call to the RFU.

```
/* NOTE: PICTURES MUST BE > 32 PIXELS WIDE */
#define background 0
#define foreground 1
#define N 32
#define FULL (unsigned int) 4294967295
#define PASS1 1
#define PASS2 2
#define MAX 2048

typedef struct picture
{    int nrows;
     int ncols;
     int wordcols;
     unsigned int data[MAX][MAX/N];
} picture;
void main()
{    picture p;
     char infile[50], outfile[50];
     FILE *inf, *outf;

     printf("\nEnter name of input file: ");
     scanf("%s", infile);
     printf("\nEnter name of output file: ");
     scanf("%s", outfile);
     inf = fopen(infile, "r");   outf = fopen(outfile, "w");
     if (read_picture(&p, inf) != -1)
     {   skel(&p);
         write_picture(&p, outf);
     }
     fclose(inf);   fclose(outf);
}

void skel(picture *p)
{    int mask1, mask2;
     int pict[3][3];
     picture newp;
     int nr, nc, row, col;
     int x, change;

     nr = p->nrows;
     nc = p->wordcols;
     mask1 = (unsigned)FULL>>1;
     mask2 = (unsigned)FULL<<(p->wordcols*N - p->ncols);
     newp.nrows = nr;
     newp.wordcols = nc;
     newp.ncols = p->ncols;
     for (col = 0; col < nc; col++)
     {   newp.data[0][col] = p->data[0][col];
         newp.data[nr-1][col] = p->data[nr-1][col];
     }

     while (1)
     {  /* test for change */
        change = 0;
        /* pass 1 */
        for (row = 1; row < nr-1; row++)
        {   for (x = 0; x < 3; x++)
            {   pict[x][0] = 0;
                pict[x][1] = p->data[row+x-1][0];
                pict[x][2] = p->data[row+x-1][1];
            }
            newp.data[row][0] = RFUOP(pict, mask1, PASS1);
            change = change | (newp.data[row][0] ^ p->data[row][0]);
            for (col = 1; col < nc-1; col++)
            {   for (x = 0; x < 3; x++)
                {   pict[x][0] = pict[x][1];
                    pict[x][1] = pict[x][2];
                    pict[x][2] = p->data[row+x-1][col+1];
                }
                newp.data[row][col] = RFUOP(pict, FULL, PASS1);
                change = change | (newp.data[row][col] ^ p->data[row][col]);
            }
            for (x = 0; x < 3; x++)
            {   pict[x][0] = pict[x][1];
                pict[x][1] = pict[x][2];
                pict[x][2] = 0;
            }
         newp.data[row][nc-1] = RFUOP(pict, mask2, PASS1);
         change = change | (newp.data[row][nc-1] ^ p->data[row][nc-1]);
        }
        /* pass 2 */
        for (row = 1; row < nr-1; row++)
        {   for (x = 0; x < 3; x++)
            {   pict[x][0] = 0;
                pict[x][1] = newp.data[row+x-1][0];
                pict[x][2] = newp.data[row+x-1][1];
            }
            p->data[row][0] = RFUOP(pict, mask1, PASS2);
            change = change | (p->data[row][0] ^ newp.data[row][0]);
            for (col = 1; col < nc-1; col++)
            {   for (x = 0; x < 3; x++)
                {   pict[x][0] = pict[x][1];
              pict[x][1] = pict[x][2];
              pict[x][2] = newp.data[row+x-1][col+1];
                }
                p->data[row][col] = RFUOP(pict, FULL, PASS2);
                change = change | (p->data[row][col] ^ newp.data[row][col]);
         }
         for (x = 0; x < 3; x++)
         {   pict[x][0] = pict[x][1];
             pict[x][1] = pict[x][2];
             pict[x][2] = 0;
         }
         p->data[row][nc-1] = RFUOP(pict, mask2, PASS2);
         change = change | (p->data[row][nc-1] ^ newp.data[row][nc-1]);
        }
        /* see if any changes */
        if (!change) break;
     }
}

/* 0 3 6
   1 4 7   for bit access
   2 5 8                   */

int RFUOP(int pict1[3][3], int mask, int pass)
{   int b[9], c[7];
    int m, x;
    int neigh, cross;
    int pict2;
    int nbit;

    pict2 = pict1[1][1];

    /* get middle bits */
    for (nbit = N-1; nbit >= 0; nbit--)
    {   if (nbit != N) for (x = 0; x < 6; x++) b[x] = b[x+3];
        else
        {   for (x = 0; x < 3; x++) b[x] = (pict1[x][0])&1 == foreground;
            m = 1<<nbit;
            for (x = 0; x < 3; x++) b[3+x] = (pict1[x][1]&m)>>nbit == foreground;
        }

        /* see if we're at end of word */
        if (nbit != 0)
        {   m = 1<<(nbit-1);
            for (x = 0; x < 3; x++) b[x+6] = (m&pict1[x][1])>>(nbit-1) == foreground;
        }
        else for (x = 0; x < 3; x++) b[x+6] = (pict1[x][2]>>(N-1))&1 == foreground;

        /* if current focus pixel is foreground, continue */
        if ((b[4]) && (mask&(1<<nbit)))
        {   /* determine level changes */
            c[0] = b[0] ^ b[1];      c[1] = b[1] ^ b[2];      c[2] = b[2] ^ b[5];
            c[3] = b[5] ^ b[8];      c[4] = b[8] ^ b[7];      c[5] = b[7] ^ b[6];
            c[6] = b[6] ^ b[3];
            /* crossing index */
            cross = 0;
            for (x = 0; x < 7; x++) cross += c[x];

            if ((cross == 1) || (cross == 2))
            {   neigh = 0;
                for (x = 0; x < 4; x++) neigh += b[x];
                for (x = 5; x < 9; x++) neigh += b[x];
                if ((neigh >= 2) && (neigh <= 6))
                {   if (pass == PASS1)
                    {   if (!b[5] || !b[7] || (!b[3] && !b[1]))
                        {   pict2 &= ~(1<<nbit);
                            pict2 |= background<<nbit;
                        }
                    }
                    else
                    {   if (!b[1] || !b[3] || (!b[7] && !b[5]))
                        {   pict2 &= ~(1<<nbit);
                            pict2 |= background<<nbit;
                        }
                    }
                }
            }
        }
    }
    return pict2;
}
```

Appendix C: **New assembly code for pass 1 of the Chimaera-optimized Zhang-Suen skeletonization algorithm described in Method III. The number of executions per section of code while thinning the test image is shown. Comments are given.**

```
sw zero, 40(sp)         change = 0
lw t8, 60(sp)           t8 = nr
li t2, 1                t2 = 1
addiu t4, t8, -1        t4 = nr-1
slti at, t4, 2          is nr-1 < 2?  (should we enter loop)
bne at, zero, loop1end  goto end of loop1
sw t2, 52(sp)           row = 1
^--------executed 4 times
loop1:
lui t1, 0x8             start computing base address of p->data
addu t1, t1, sp         continue computing base address of p->data
lw t9, 52(sp)           t9 = row
lw t1, 120(t1)          t1 = p->data
sll t9, t9, 8           t9 = row * 256
addu t9, t9, t1         t9 = &(p->data[row][0])-12 (-12 because of picture structure format)
lw t3, -244(t9)         t3 = p->data[row-1][0]
lw t4, 12(t9)           t4 = p->data[row][0]
lw t5, 268(t9)          t5 = p->data[row+1][0]
lui t7, 0x7fff          t7 = 0x7fff (value of mask1--first bit is not valid (edge))
li t9, 1                t9 = 1
RFUOP 1 t7              change odd bits mask (t7) w/crossing index op
RFUOP 3 t8              compute odd result for pass 1
lw t2, 56(sp)           t2 = nc
sw t8, 44(sp)           store half result in variable x (unused for this implementation)
addiu t2, t2, -1        t2 = nc-1
slti at, t2, 2          is nc-1 < 2? (should we enter loop)
bne at, zero, loop2end  goto end of loop2
sw t9, 48(sp)           col = 1
^--------executed 3192 times
loop2:
addu t0, t3, zero       shift data over
addu t1, t4, zero       shift data over
addu t2, t5, zero       shift data over
addu t6, t7, zero       shift mask data over
lw t3, 48(sp)           t3 = col
lw t2, 56(sp)           t2 = nc
addiu t3, t3, 1         t3 = col+1
bne t2, t3, notedge     use mask of 0xffff, not mask2 (not at end of row)
lui t7, 0xffff          t7 = 0xffff (starting mask value for non-end words)
^--------executed 54264 times
lui at, 0x30            end of row: start computing address of mask2
addu at, at, sp         end of row: continue computing address of mask
lw t7, -25612(at)       end of row: t7 = mask2
^--------executed 3192 times
notedge:
lui t8, 0x8             start computing base address of p->data
addu t8, t8, sp         continue computing base address of p->data
lw t9, 52(sp)           t9 = row
lw t8, 120(t8)          t8 = p->data
lw t3, 48(sp)           t3 = col
sll t9, t9, 8           t9 = row * 256
sll t3, t3, 2           t3 = col * 4
addu at, t9, t3         at = row*256 + col*4
addu t9, at, t8         t9 = &(p->data[row][col])-12
lw t3, -244(t9)         t3 = p->data[row-1][col]
lw t4, 12(t9)           t4 = p->data[row][col]
lw t5, 268(t9)          t5 = p->data[row+1][col]
lw t8, 44(sp)           t8 = x (results of odd bit computation)
addu t9, sp, 64         compute base address of newp.data
addiu at, at, -4        at = row*256 + (col-1)*4
RFUOP 2 t6              change mask for even bits with crossing index operation
RFUOP 4 t8              compute even result, incorporating odd result (pass 1)
addu t9, t9, at         t9 = &(newp.data[row][col-1])-12
sw t8, 12(t9)           newp.data[row][col-1] = t8 (result of pass1 of thinning)
lw t9, 40(sp)           t9 = change
xor t8, t8, t1          see if any changes were made
or t9, t9, t8           if any changes were made, make sure change is non-zero
sw t9, 40(sp)           and save change back to its memory location
RFUOP 1 t7              change mask for odd bits with crossing index operation
RFUOP 3 t8              compute odd result for pass 1
lw t0, 48(sp)           t0 = col
lw t1, 56(sp)           t1 = nc
sw t8, 44(sp)           half result stored in x (odd bits)
```

```
28  addiu t0, t0, 1            t0 = col+1
29  slt at, t0, t1             see if col+1 < nc
30  bne at, zero, loop2        continue loop2
31  sw t0, 48(sp)              col = col+1
^--------executed 54264 times
loop2end:
0   addu t0, t3, zero          shift data over
1   addu t1, t4, zero          shift data over
2   addu t2, t5, zero          shift data over
3   addu t6, t7, zero          shift data over
4   lw t8, 44(sp)              get odd half result
5   lw t9, 52(sp)              t9 = row
6   lw at, 56(sp)              at = nc
7   RFUOP 2 t6                 change mask for even bits with crossing index operation
8   RFUOP 4 t8                 compute even result, incorporating odd result (pass 1)
9   sll t9, t9, 8              t9 = row * 256
10  sll at, at, 2              at = nc * 4
11  addu t9, t9, at            t9 = row*256 + nc*4
12  addiu t9, t9, -4           t9 = row*256 + (nc-1)*4
13  addu at, sp, 64            compute base address of newp.data
14  addu t9, t9, at            t9 = &(newp.data[row][nc-1])-12
15  sw t8, 12(t9)              newp.data[row][nc-1] = result from thinning
16  lw t9, 40(sp)              t9 = change
17  xor t8, t8, t1             see if any changes were made
18  or t9, t9, t8              if any changes were made, make sure change is non-zero
19  sw t9, 40(sp)              and save change back to its memory location
20  lw t0, 52(sp)              t0 = row
21  lw t1, 60(sp)              t1 = nr
22  addiu t0, t0, 1            t0 = row+1
23  addiu t1, t1, -1           t1 = nr-1
24  slt at, t0, t1             test if done with rows
25  bne at, zero, loop3        continue loop3
26  sw t0, 52(sp)              row = row+1
^--------executed 3192 times
loop1end:
```