Comparative Timings for a Neuron Recognition Program on Serial and Pyramid Computers

by

Ze-Nian Li and Leonard Uhr

Computer Sciences Technical Report #665

September 1986

COMPARATIVE TIMINGS FOR A NEURON RECOGNITION PROGRAM ON SERIAL AND PYRAMID COMPUTERS

Ze-Nian Li and Leonard Uhr
Department of Computer Sciences
University of Wisconsin-Madison

ABSTRACT

This paper examines the time needed by a program that recognizes neurons in photomicrographs when it is executed on a conventional serial computer (a VAX 11/750), and compares this with the time that would be needed if it were executed on an appropriate parallel-serial pyramid multi-computer. As the size of the image array increases from 64x64 to 4,096x4,096 the estimates of the pyramid's increases in speed grow from 350 times as fast as the VAX to 1,276,300 times as fast.

INTRODUCTION

Evaluations and benchmarks should compare different systems on simple algorithms and also on whole programs that accomplish significant tasks. This paper evaluates and compares two architectures - a conventional serial computer (exemplified by the VAX) and a pyramid multi-computer. It examines their speed at executing a pattern perception program that successfully recognizes neurons in photomicrographs, and it examines the individual sub-algorithms that combine into the total program. Relatively few programs have been coded and tested for their ability to recognize biological images like neurons that are appropriate for execution on parallel computers. The program we have developed [1] for perceptual recognition of objects, and used for neuron recognition, was able to recognize a high percentage of the different neurons in an image. It makes use of a number of micro-modular local transforming operations that are executed in a massively parallel hierarchical fashion. It uses a relatively small set of relatively simple transforms, but one that is sufficient to give good results on the neuron recognition task. When given a larger

number of each more powerful transforms it will become capable of recognizing a wider variety of more complex objects. Since this program was coded with a pyramid structure, it is amenable to parallelization. This paper shows how that can be done, and analyzes and presents timings on a serial computer and on a parallel-serial pyramid.

NEURON RECOGNITION PROGRAM

Our program runs on a simulated pyramid on a VAX 11-750. The pyramid used in this example has 10 levels (0 - 9). Each level k ($0 \leqslant k \leqslant 9$) has $2^k * 2^k$ nodes (simple processors with their own registers and local memories). Each node at level k is hard-wired to its 13 neighbors, i.e. 1 parent, 8 siblings and 4 children (except that level 9 nodes have no child node, level 0 nodes have no parent node).

The recognition process consists of the following steps:

(1) Get Micro-Edges. A pyramidal median filtering operation is first applied to preserve edges while reducing noise. An input image whose resolution is 64x64, will be stored at level 6. Each node at level 5 takes the median intensity value of its 3x3 set of children as its intensity value. After filtering, eight edge masks (encoded by 0,1,...,7) are applied to the filtered image array (which is now at level 5). At each node of the array, convolutions are computed using these eight masks. The result of this step is a micro-edge map of the filtered image.

(2) Gather Short Curves. The concept of transforms in [2] is used for feature extraction in the recognition of neurons. Basically, transforms are procedures that extract or combine features. Often a parent executes some transform on its set of specified children. When an operation involves processors in more than one level, it is named 'pyramid operation'. With 'pyramid operations', the transforms are executed in a hierarchical manner. Sometimes an 'array operation' is also used, in which only processors in the same level are involved. The detection of micro-edges is one example of 'array operation'.

It was found that a micro-edge map of any ellipse-shaped object can be divided into eight segments. The places where micro-edges change directions, e.g. 0 → 1, 1 → 2, etc, are very important. For nearly round or elliptical objects, there are always eight such places. This phenomenon is invariant to size, orientation and elongation of the cell; moreover it can always be observed in a very small window, e.g. 3x3. Based on these observations, a transform was built for short curves. At level 4, each node examines its 3x3 child set. It will claim that it finds SHORTCURVE0 if there are a 'micro-edge0' at the upper left and a 'micro-edge1' at the lower right of its window, each with considerable weight. In the same way, it finds SHORTCURVE1,..., SHORTCURVE7.

(3) Combine to build long curves and whole cells. Our transforms detect such long curves much as they detect short curves. Every node at level 3 examines its 3x3 window at level 4. If, for example, SHORTCURVE0 is at the upper left and SHORTCURVE1 at the lower right of the window and their combined weights exceed certain threshold, they get compounded to LONGCURVE0 (SW) at level 3. Similarly the transforms that combine long curves into cells first find the pairs of matched LONGCURVEs SW-NE and SE-NW respectively, and then try to combine two such corresponding pairs into a whole cell at level 2. Coordinates of the curve components are checked to reduce erroneous compoundings.

See [1] for more description of this recognition program and its performance.

EXECUTION TIMES ON THE VAX 11/750

When running under the UNIX operating system, the time used for executing our neuron recognition program on a VAX 11/750 can be acquired from execution time measures provided by the UNIX system. With an input 64x64 image at level 6, the execution time for each procedure is shown in Fig. 1.

task of procedure	level	type	time (ms)
median filtering	5	P	5,420
micro-edge	5	A	17,390
thresholding	5	A	220
short curve	4	P	9,670
cleaning	4	A	1,260
long curve	3	P	2,030
cell (3x3)	2	P	1,240
cell (4x4)	2	P	2,850
total			40,080

A -- array operation

P -- pyramid operation

Fig. 1 Execution Time Measure on All Procedures

It is well understood that, for implementing the algorithms described in the previous section, a 1-CPU computer has to iterate its same operation $2^k \times 2^k$ (k -- level) times at the corresponding levels, whereas a parallel multi-computer (with a sufficient number of processors and memory) need only do it once. Furthermore, a 1-CPU computer does the following additional operations which are usually not needed for a parallel multi-computer system. (a) Loop control: Taking care of branches of loops through 2D arrays at each level. (b) Array indexing: Each time a node is accessed, the address of an array element is calculated. For many compilers (including our C compiler), this requires several instructions. Since a tremendous amount of array indexing is usually encountered in an image understanding program, this becomes appreciable. (c) Boundary checking: The program checks boundaries for every iteration of a window operation. If an operand's row or column address is beyond the boundary, the neighboring border pixel is used instead. (d) Procedure calls: For a better style, our program uses many procedure calls. In particular, almost every node indexing is a procedure call. For example, if a node at row x, column y of level l wants to get the value of its number c child, instead of doing some in-line index calculation, a procedure will be called. This means thousands of procedure calls for the neuron recognition program.

Since the program is written in C, we use assembly codes generated by the C compiler and timing data available for each assembly instruction to get a rough estimation of its execution time.

Many instructions (e.g., movl) have different addressing modes, and different timings (e.g. movl reg,reg; movl reg,mem). For simplicity only one average time is used. The averages were determined taking into consideration the number of occurrences of different addressing modes in the program. They are not necessarily accurate, but they suffice our requirement. Also the instructions were counted dynamically, i.e., only those actually executed at the run time are gathered.

The estimated times were compared to the actual execution times as measured on our VAX 11/750 to ascertain their accuracy. The purpose of doing this is to understand which portion of the total execution time is due to operations common to the serial and parallel approach, and which portion is due to one of the above additional operations needed exclusively by serial computers.

Since the procedures for 'micro-edge' and 'short curve' extraction take 27,060 ms, which is more than 67% of the total execution time, and they represent typical 'array operations' and 'pyramid operations' respectively, the analysis on the VAX 11/750 in this paper examines these two procedures.

Analysis of Micro-Edge Extraction

This procedure does 'array operations' at level 5 (with a resolution 32x32). Eight 3x3 masks are used to detect micro-edges with eight different directions (0°, 45°, 90°, ..., 315°). Our analysis will start with a very simple example.

(1) Analysis on Sample Program 1

A simplified program for performing one 3x3 mask operation on a 32x32 array is given in Fig. 2. To avoid boundary checking in this example, the mask operation is only on nodes within row 1 to 30 and column 1 to 30.

```
/* Sample program 1 for micro-edge extraction.  */
main ()
{ int   I, J, K, L;
  short image [32][32], result [32][32];
  int   mask[3][3];
  int   temp;

  getru0 ();          /* start measuring time. */
  for (I = 1; I <= 30; I++)
    for (J = 1; J <= 30; J++)
      { temp = 0;
        for (K = 0; K < 3; K++)
          for (L = 0; L < 3; L++)
            temp = temp + image [I + K - 1][J + L - 1]
                        * mask [K][L];
        result [I][J] = temp;
      }
  getru ();           /* get time measurement. */
}
```

Fig. 2 Sample Program 1

The assembly code of this sample program is in Fig. A1 in the appendix. The number of assembly language instructions within the innermost loop (from L28 to L27 in Fig. A1) is 23. Among these 23, 15 instructions are used for generating addresses for 'image' and 'mask' array elements. Beside these 15 instructions, four instructions are used for loop control. The remaining four are used for 'mov', 'add' and 'mul' ..., which are instructions needed for both the VAX 11/750 and any parallel multi-computers to accomplish the desired mask operation. We name them 'core instructions'. The estimated times for 'loop control', 'core instructions' and 'array indexing' are shown in Fig. 3.

Since this innermost loop will be executed $30 \times 30 \times 3 \times 3 = 8{,}100$ times, the total number of the innermost loop instructions is $23 \times 8{,}100 = 186{,}300$; the total time spent in the innermost loop is about $53.4\,\mu s \times 8{,}100 = 432.5$ ms. Taking all the instructions outside this innermost loop into account, the total number of assembly instructions is 218,913. With a rough estimation (ignoring the fact that different instructions have different execution time), the total time for executing one 3x3 mask operation on 30x30 nodes is about $432.5 \times \frac{218{,}913}{186{,}300} = 508.2$ ms.

Loop Control		
Instruction	No. of inst.	Time / inst. (μs)
cmpl	1	1.5
incl	1	1.2
jgeq	1	1.5
jbr	1	1.0
Subtotal	4	5.2
Core Instructions		
Instruction	No. of inst.	Time / inst. (μs)
addl2	1	1.2
cvtwl	1	1.5
movl	1	1.5
mull2	1	8.8
Subtotal	4	13.0
Array Indexing		
Instruction	No. of inst.	Time / inst. (μs)
addl2	4	1.2
addl3	2	2.0
ashl	3	3.0
decl	2	1.2
movl	1	1.5
mull3	1	9.5
subl3	2	2.0
Subtotal	15	35.2
Total	23	53.4

Fig. 3 Estimated Execution Time for Sample Program 1

The actual execution time measured by UNIX system for this sample program 1 was 540 ms. Thus the relative error of our analysis is $\frac{540 - 508.2}{540} = 5.9\%$.

(2) Analysis on Sample Program 2

Fig. A2 in the appendix, sample program 2, is a slightly simplified version of a micro-edge detection procedure. This program is written in the style of the neuron recognition program. Node accessing uses procedure calls. Boundary checking is also invoked. To make it comparable to sample program 1, mask operations are still only on nodes with $0 < \text{row} < 31$ and $0 < \text{col} < 31$. This means that no array index will be 'out-of-bound' in this sample program, but notice that the code for 'if-condition' checking is always executed.

Loop Control		
Instruction	No. of inst.	Time / inst. (μs)
cmpl	1	1.5
incl	1	1.2
jbr	1	1.0
jgeq	1	1.5
Subtotal	4	5.2
Core Instructions		
Instruction	No. of inst.	Time / inst. (μs)
addl2	1	1.2
cvtwl	1	1.5
movl	1	1.5
mull2	1	8.8
Subtotal	4	13.0
Array Indexing		
Instruction	No. of inst.	Time / inst. (μs)
addl2	5	1.2
addl3	2	2.0
ashl	1	3.0
decl	3	1.2
movl	6	1.5
mull2	1	8.8
mull3	2	9.5
subl3	1	2.0
Subtotal	21	55.4
Boundary Checking		
Instruction	No. of inst.	Time / inst. (μs)
cmpl	2	1.5
movl	2	1.5
jgeq	2	1.5
jlss	2	1.5
tstl	2	1.2
Subtotal	10	14.4
Procedure Calls		
Instruction	No. of inst.	Time / inst. (μs)
call+ret	2	20.9
jbr	4	1.0
movl	4	1.5
pushl	8	1.5
subl2	2	1.0
Subtotal	20	65.8
Total	59	153.8

Fig. 4 Estimated Execution Time for Sample Program 2

The estimated execution time for 'loop control', 'core instruction', 'array indexing', 'boundary checking' and 'procedure calls' is in Fig. 4.

Fig. 4 illustrates that in one innermost loop of sample program 2, the VAX 11/750 used 3.4% of the total execution time for 'loop control', 8.5% for 'core instructions', 36% for 'array indexing', 9.4% for 'boundary checking' and 42.8% for 'procedure calls'.

The total number of innermost loop instructions is $59 \times 8{,}100 = 477{,}900$, the total time spent in the innermost loop is about $153.8\,\mu s \times 8{,}100 = 1245.8$ ms. The total number of assembly instructions is 504,217. An estimate of total time for executing sample program 2 is $1245.8 \times \frac{504{,}217}{477{,}900} = 1{,}314.4$ ms.

The execution time for sample program 2 measured by the UNIX system was 1,550 ms. Hence the relative error of our analysis on sample program 2 is $\frac{1{,}550 - 1{,}314.4}{1{,}550} = 15.2\%$.

Sample program 2 does only one mask operation on a 30x30 subarray. For eight mask operations and a 32x32 array, it would be $\frac{1{,}550 \times 8 \times 32 \times 32}{30 \times 30} = 14{,}108$ ms. Considering this is still a slightly simplified version of our micro-edge extraction routine, the execution time of 17,390 ms in Fig. 1 is now reasonably well approximated and understood.

Analysis of Short Curve Extraction

The main part of the procedure for finding one type of short curves (SHORTCURVE4) is extracted as sample program 3 (code not shown in this paper). All the nodes at level 4 examine their 3x3 child nodes at level 5 ('pyramid operation'), to combine micro-edges arranged in a certain way into possible short curves. Apparently, not all the nodes will succeed in finding curves. In other words, the code related to processing possible combinations will only occasionally be executed. In our neuron example, the chance is about 10%. This part of the execution time can be prorated into the final estimates. The estimated execution time for 'loop control', 'core instruction', 'array indexing', 'boundary checking' and 'procedure calls' is in Fig. A3 in the appendix.

Similar analysis shows that in one innermost loop, VAX 11/750 spends 1.4% of the total execution time for 'loop control', 5.9% for 'core instructions', 20.6% for 'array indexing', 5.2% for 'boundary checking' and 66.8% for 'procedure calls'. An estimate of total time for executing this program is 666.9 ms.

The real time measure from the UNIX system for this sample was 770 ms. Hence the relative error of our analysis on sample program 3 is $\frac{770 - 666.9}{770} = 13.4\%$.

The above time estimates on three sample programs all have lower values with relative error from 5% to 15%. The timing measure is higher on the VAX 11/750 because of the overhead of the UNIX multi-user environment. Therefore these sample programs were also run on a bare VAX 11/750 (one of the nodes of our department's multi-computer network), where the extra overhead from the multi-user environment is eliminated. The timing measures are 494 ms, 1,430 ms and 709 ms as opposed to 540 ms, 1,550 ms and 770 ms respectively. Accordingly, the relative errors become 2.9%, 8.1% and 5.9%.

Fig. 5 summarizes the discussion in this section by listing the statistics on distributions of the execution time for extracting micro-edges and short curves. The data are initially obtained from the analysis on innermost loops. This is sufficient for our purpose, since these procedures spend most of their time in innermost loops.

	sample program 2	sample program 3
loop control	3.4%	1.4%
core instructions	8.5%	5.9%
array indexing	36.0%	20.6%
boundary checking	9.4%	5.2%
procedure calls	42.8%	66.8%

Fig. 5 Statistics of Time distribution

At this point, we want to discuss some optimization issues. (a) Generally the speed of a program depends heavily on the style and skill of the programmer. For instance, since procedure calls (mainly for node indexing) take so much time, people in real application

program development may want to replace most of them by in-line index calculation. (b) The quality of the compiled code contributes significantly to the speed of a program. Our C compiler shows poor performance on many aspects (e.g., register utilization, loop invariant), that produces a larger overhead on the 'array indexing' part. (c) The boundary checking in this program can be replaced by duplicating the border columns and rows, to have larger arrays with expanded borders. (But note that this may not be trivial for VLSI fabrications.)

EXECUTION TIMES ON A PYRAMID

In this section, estimated times are given for the neuron recognition program executed in a pyramid. The assumptions on the pyramid are: (a) its nodes are 1-bit processors with their own registers and local memories; (b) basic instructions are similar to the VAX instructions, i.e. mov, add, mul ..., except they have to be executed bit-serially; (c) if an instruction takes t μs to finish and the word length of its operands is w bits, then it takes $t \times w$ μs to finish in this pyramid. The maximum word length needed for executing our program is 16. Thus we are assuming all the instructions will take 16 times longer to finish, e.g., an 'add' will take $1.2 \times 16 = 19.2$ μs, etc.

As described above, a pyramid node is hard-wired to 13 neighboring nodes (1 parent, 8 siblings and 4 children). If we define the 'distance' between two nodes as the number of links through which the information has to travel from one node to the other, then a node will have distance one to any of its 13 neighbors. We name them 'direct parent', 'direct sibling' and 'direct child' respectively. For a 'pyramid operation', any window size bigger than 2x2 will produce distances greater than one between some of the children nodes and their parent node. If the distance is d, it will basically take d fetches to access a non-direct neighboring node.

Several orders of magnitude speed-up are expected. First, thousands of parallel processors are used. Second, The instructions for 'loop control', 'array indexing' and

'boundary checking' are no longer needed. Numerous 'procedure calls' due to node indexing are also saved. On the other hand, we should point out that, in the last section, some instructions contributed much less to the total execution time than others, because they were in the 'then' part of a if-statement, and the 'if-condition' had very little chance to be true. For example, only 10% of the nodes had a chance to execute the code related to successful short curve combinations. With SIMD multi-computers at each level of the pyramid, while 10% of the nodes are combining curves, the remaining 90% are 'masked out' and sitting idle. Thus these instructions are always executed to completion.

Estimations for 'Micro-Edge' Extraction

Pseudo code for this 'array operation' is given below. This is not optimized code. It is written to imitate the assembly code generated by our C compiler, so as to make more reasonable comparisons on time measures. The main operations for each mask are 9 multiplications, 9 additions, 9 fetches and 9 stores.

```
for each of the eight masks do:
 { mov    #0, result
   mov    #0, weight
   for each of the nine mask elements i, do:
    { clr    reg1                ; or 'set', according to the
                                     value of mask element.
      mov    mem(i), reg2        ; mem (i) -- intensity value
      mul    reg1,   reg2
      add    result, reg2
      mov    reg2, result
    }
   cmp    result, weight         ; result > old weight ?
   jleq   L1
   mov    result, weight
   mov    i, direction           ; store edge direction
L1:
 }
```

The two 'for' loops are stated only for simplicity. They can be expanded. Therefore no count is made for 'loop control' instructions. Fig. 6 illustrates the execution time for one mask operation. Since the distance between a node and its direct sibling is one, a fetch or store ('mov') takes $1 \times 1.5 \times 16 = 24\,\mu$s. ('mov1' is used to indicate a move with

distance one; accordingly 'mov2' will indicate a move between nodes with distance two and takes 48 μs.)

Two 'mov1's, one 'clr', one 'mul' and one 'add' are iterated nine times, the rest are just executed once. Notice that the last two 'mov1's do not get executed every time at all the nodes. While some processors are doing these two 'mov1's, others are idle. The time still has to be counted. (Possible overhead for 'masking out' the idle nodes is ignored for this discussion.)

Time for one mask operation		
Instruction	No. of inst.	Time / inst. (μs)
add	1	19.2
clr	1	19.2
mov1	2	24.0
mul	1	140.8
Subtotal	5	227.2
Subtotal x 9	45	2044.8
cmp	1	24.0
jleq	1	24.0
mov1	4	24.0
Total	51	2188.8

Fig. 6 Time for Micro-Edge Operation in a Pyramid

Hence the execution time for 'Micro-Edge' extraction is 2,188.8 $\mu s \times 8 = 17.5$ ms. For simplicity, variable word lengths are not used in this discussion. If 1-bit word were used for holding the mask values, some instructions would not be 16 times as slow as VAX's; words short than 16 bits will be processed commensurately faster.

Estimations for 'Short Curve' Extraction

Pseudo code for this 'pyramid operation' is given below (handling of coordinates is not included).

```
for each of the eight short curves, do
 { mov1   #0, sum1
   mov1   #0, sum2
```

```
mov     mem (DIREC), reg1     ; get edge direction.
mov1    #DIREC1,     reg2     ; direction to compare.
for a subwindow of 6 nodes, do     ; upper left
  { cmp   reg1, reg2
    jneq  L1
    mov   mem (SCORE), reg2      ; get edge weight
    add   sum1, reg2
    mov1  reg2, sum1
  L1:
  }
mov1    #DIREC2,     reg2
for a subwindow of 6 nodes, do     ; lower right
  { cmp   reg1, reg2
    jneq  L2
    mov   mem (SCORE), reg2
    add   sum2, reg2
    mov1  reg2, sum2
  L2:
  }
mov1    sum1, reg1
mov1    sum2, reg2
mul     reg1, reg2                 ; take sum1*sum2 as the
mov1    reg2, mem (CURVE)   ; weight of short curve.

}
```

Fig. 7 illustrates the execution time for one short curve detection.

Time for one short curve detection		
Instruction	No. of inst.	Time / inst. (μs)
add	2	19.2
cmp	2	24.0
jneq	2	24.0
mov	2	37.3
mov1	2	24.0
Subtotal	10	257.0
Subtotal x 6	60	1542.0
mov	1	37.3
mov1	7	24.0
mul	1	140.8
Total	69	1888.1

** 'mov1' is used for local memory access, 'mov' for 3x3 window 'pyramid operations'. Because 4 children have distance 1 to their parent, and 5 children have distance 2, the timing is prorated as $\frac{4 \times 24 + 5 \times 48}{9} = 37.3\,\mu s$.

Fig. 7 Time for Short Curve Detection in a Pyramid

The coordinate handling code will take about 1,368 μs. Consequently one short curve detection takes roughly 3,256 μs. Hence the execution time for 'Short Curve' detection is $3{,}256\,\mu s \times 8 = 26.0$ ms.

Estimations for Other Steps

In this paper, we will not describe a whole detailed analysis on all the steps. According to our analysis, compounding 'long curves' takes 19.4 ms, compounding 'cells' with 3x3 window takes 13.5 ms, with 4x4 window, 24 ms. the median filtering step takes 9.7 ms. Fig. 8 lists estimates on all procedures.

task of procedure	level	type	time (ms)
median filtering	5	P	9.7
micro-edge	5	A	17.5
thresholding	5	A	0.1
short curve	4	P	26.0
cleaning	4	A	4.3
long curve	3	P	19.4
cell (3x3)	2	P	13.5
cell (4x4)	2	P	24.0
total			114.5

Fig. 8 Execution Time Estimation for a Pyramid

Finally Table 1 shows a comparison on the execution time for the neuron recognition program. The VAX execution times up to image size 512x512 are all actual time measures. Table entries with image size bigger than 512x512 are extrapolated from the time measure for a 512x512 image. Not surprisingly, bigger images take more time on a serial computer. Consequently, the pyramid's relative speed-up increases as the image size increases.

image size	VAX 11/750 execution time (sec.)	Pyramid estimated time (sec.)	VAX / Pyramid Ratio
64x64	40.1	0.1145	350
128x128	154.1	0.1145	1,346
256x256	582.5	0.1145	5,087
512x512	2,283.4	0.1145	19,942
1024x1024	9,133 *	0.1145	79,764
2048x2048	36,534 *	0.1145	319,070
4096x4096	146,140 *	0.1145	1,276,300

* extrapolated from the time measure for 512x512 image.

Table 1 Comparative Timings for the VAX and Pyramid

SUMMARY AND DISCUSSION

This paper has examined how fast a relatively simple program (yet one that is complete, modular, and that can be made much more general, by improving and adding to its transforms) that recognizes neurons in photomicrographs will execute on a conventional serial computer, and on a pyramid. The pyramid will execute the neuron recognition program roughly 350 to 1,276,300 times as fast as a VAX 11/750, moving from a 64x64 to a 4,096x4,096 image. The VAX estimates that were checked against actual VAX timings are close enough to conclude that the total set of estimates is reasonably accurate. In any case the margin of error is extremely small and minor compared to the very large differences in time needed by the VAX in contrast to the pyramid.

It is possible that different kinds of programs might be coded for the VAX that executed faster, with as high a level of performance. But the results of this neuron recognition program are at least as good as those of any program of which we are aware, and its speeds appear to be typical for programs of this sort.

The pyramid time estimates may well be too slow, for the following reasons: (a) VAX assembly language instructions are used, whereas an actual pyramid would have its own, for it more efficient, set of instructions (e.g., a single 200 nanosecond fetch-add as in

the DAP). (b) Special image processing instructions (e.g., for 3x3 mask operations) would further reduce timings. (c) The pyramid when coded to process n-bit numbers need take only n steps, whereas we have always assumed it takes 16 steps.

ACKNOWLEDGEMENT

This research was partially supported by NSF Grant DCR-8302397 to Leonard Uhr. The authors would like to thank Peter Sandon for his valuable comments on this paper.

APPENDIX

/* Assembly code for sample program 1. */

```
LL0: .data
     .text
     .align   1
     .globl   _main
_main: .word    L12
     jbr      L14
L15: calls    $0,_getru0
     movl     $1,-4(fp)
L19: cmpl     -4(fp),$31
     jgeq     L18
     movl     $1,-8(fp)
L22: cmpl     -8(fp),$31
     jgeq     L21
     clrl     -4152(fp)
     clrl     -12(fp)
L25: cmpl     -12(fp),$3
     jgeq     L24
     clrl     -16(fp)
L28: cmpl      -16(fp),$3          ; L < 3 ?
     jgeq     L27
     addl3     -16(fp),-8(fp),r0     ; L + J
     decl     r0                  ; L + J - 1
     subl3    $2064,fp,r1
     addl3    -12(fp),-4(fp),r2     ; I + K
     decl     r2                  ; I + K - 1
     ashl     $6,r2,r2
     addl2     r2,r1
     ashl     $1,r0,r0
     addl2     r0,r1          ; *image [I+K-1][J+L-1]
     movl      -16(fp),r0
     subl3     $4148,fp,r2
     mull3     $12,-12(fp),r3
```

```
      addl2    r3,r2
      ashl     $2,r0,r0
      addl2    r0,r2                ; *mask [K][L]
      cvtwl    (r1),r0
      mull2    (r2),r0
      addl2    -4152(fp),r0          ; get 'temp'
      movl     r0,-4152(fp)          ; store 'temp'
L26: incl      -16(fp)
      jbr      L28
L27:
L23: incl      -12(fp)
      jbr      L25
L24: movl      -8(fp),r0
      subl3    $4112,fp,r1
      ashl     $6,-4(fp),r2
      addl2    r2,r1
      ashl     $1,r0,r0
      addl2    r0,r1
      cvtlw    -4152(fp),(r1)
L20: incl      -8(fp)
      jbr      L22
L21:
L17: incl      -4(fp)
      jbr      L19
L18: calls     $0,_getru
      ret
      .set     L12,0x0
L14: movab     -4152(sp),sp
      jbr      L15
      .data
```

Fig. A1 Assembly Code for Sample Program 1

```
/* Sample program2 for edge extraction.  */

#include "definition.h"

/* get index to array 'memory'. */
NodeIndex (level, row, col, field)
int          level;
int          row,col;
int          field;
{ int index;

   index = InitLvl [level] + (row * SizeLvl [level]
             + col) * MemLvl [level] + field - 1;
   return (index);
}

/* access local memory 'field'. */
int          nget (level, row, col, field)
int          level, row, col, field;
{ int        index;
```

```
  /* index error checking code not shown here. */

  index = NodeIndex (level, row, col, field);
  return (memory [index]);
}

main ()
{ int         I, J, K, L;
  int         mask[3][3];
  int         temp;
  int         row, col;
  int         level = 5;
  int         index1, index2;

  Init ();
  getru0 ();   /* start time measuring. */
  SizeLvl [5] = 32;
  for (I = 1; I < 31; I++)
    for (J = 1; J < 31; J++)
      { temp = 0;
        for (K = 0; K < 3; K++)
          for (L = 0; L < 3; L++)
            { row = I + K - 1;
              col = J + L - 1;
              /* boundary checking. */
                    if (row < 0)  row = 0;
              if (col < 0)  col = 0;
              if (row >= SizeLvl [level])
                row = SizeLvl [level] - 1;
              if (col >= SizeLvl [level])
                col = SizeLvl [level] - 1;
              /* convolution. */
                    temp = temp + nget (LEVEL5, row, col, FIELD1)
                                  * mask [K][L];
                  }
        index2 = NodeIndex (LEVEL5, I, J, FIELD2);
        memory [index2] = temp;       /* store result. */
      }
  getru ();    /* get time measurement. */
}
```

Fig. A2 Sample Program 2

Loop Control		
Instruction	No. of inst.	Time / inst. (μs)
cmpl	1	1.5
incl	1	1.2
jbr	1	1.0
jgeq	1	1.5
Subtotal	4	5.2
Core Instructions		
Instruction	No. of inst.	Time / inst. (μs)
addl3	2	2.0
cmpl	4	1.5
jgtr	1	1.5
jlss	1	1.5
jneq	2	1.5
movl	1	1.5
subl3	2	2.0
Subtotal	13	21.5
Array Indexing		
Instruction	No. of inst.	Time / inst. (μs)
addl2	5	1.5
ashl	2	3.0
cvtwl	1	1.5
decl	3	1.2
divl2	1	9.0
divl3	1	9.7
movl	3	1.5
mull2	2	8.8
mull3	1	9.5
subl3	3	2.0
Subtotal	22	74.9
Boundary Checking		
Instruction	No. of inst.	Time / inst. (μs)
addl3	2	2.0
cmpl	4	1.5
jneq	4	1.5
movl	2	1.5
Subtotal	12	19.0

Procedure Calls		
Instruction	No. of inst.	Time / inst. (μs)
addl3	1	2.0
call+ret	5	20.9
clrl	2	1.5
compl	12	1.5
jbr	18	1.0
jeql	10	1.5
jgeq	1	1.5
jleq	2	1.5
jlss	2	1.5
jneq	3	1.5
movl	15	1.5
pushl	24	1.5
subl2	5	1.0
tstl	6	1.2
Subtotal	106	243.2
Total	157	363.8

Fig. A3 Estimated Execution Time for Sample Program 3

REFERENCES

1. Z.N. Li and L. Uhr, "A Pyramidal Approach for the Recognition of Neurons Using Key Features", *Pattern Recognition* 19, pp. 55-62, 1986.

2. L. Uhr and R.J. Douglass, "A Parallel-Serial Recognition Cone System for Perception: Some Test Results", *Pattern Recognition* 11, pp. 29-39, 1979.