Subject: Can I do this without using loops?
Posted by Peter Clinch on Wed, 05 Jun 1996 07:00:00 GMT
View Forum Message <> Reply to Message

Can't quite see how, but I have a feeling there's a Better Way...

I have a couple of images, same size, and I need to compare the two and take a result which has the highest value from either of the inputs. Ay the mo. I'm just using loops through all the pixels using an if greater than comparison, and it takes ages... :(

Pointers for a cleaner approach would be much appreciated!

Cheers, Pete.
--
Peter Clinch Dundee University & Teaching Hospitals
Tel 44 1382 660111 ext. 3637 Medical Physics, Ninewells Hospital
Fax 44 1382 640177 Dundee DD1 9SY Scotland UK
net p.j.clinch@dundee.ac.uk http://www.dundee.ac.uk/MedPhys/

Subject: Re: Can I do this without using loops?
Posted by David Ritscher on Mon, 10 Jun 1996 07:00:00 GMT
View Forum Message <> Reply to Message

S Bhattacharyya wrote:
>
> Regarding loops, I am kinda in the same boat. My advisor
> keeps complaining about how slow our code runs...We don't seem to
> know any better around here :-)
>
> Q1) I have a generic array foo(x,y). I'd like to divide each column
> by its max. Can this be done without looping ?
>
> Q2) I have a generic array foo=fltarr(a,b). I'd like to copy findgen(b)
> into every column. Any way of doing this without loops ?

Can these be done without loops???
No. Unfortunately, the IDL/PVWave subscript syntax is relatively powerful, but not that powerful. I keep hoping that one or both of the two languages (i.e., companies) will decide to improve in this area, providing a syntax that specifies 'loop over' and 'extract as vector', i.e., something that communicates to the interpreter which dimensions of an array should be extracted and passed in entirety to something, such as a function, and over which dimensions looping should be performed. As an example syntax, the following would extract the greatest value of each column of the array A:

```
 column_maxes = fltarr( 1, n_elements(A(0,*)))
 column_maxes(loop_over:i) = max( A(loop_over:i, extract_as_vector) )
```
where 'i' is a dummy variable that synchronizes the looping.

Back to the real world: The following are possible solutions under the current syntax limitations. They concur with and expand upon the comments of Prof. Kenneth P. Bowman.

> Q1) I have a generic array foo(x,y). I'd like to divide each column
> by its max. Can this be done without looping ?

Take, for example:

```
  xsize = 3
  ysize = 5
  foo = findgen(xsize, ysize)
```

Here's the simplest solution:

```
  for i = 0L, xsize-1 do foo(i, 0) = foo(i, *) / max(foo(i, *))
```

However, this accesses the arrays column-by-column, which can lead to slow operation with large arrays. Then the following would be more efficient, and could reduce page faulting to ~1/3 since it is necessary to search through the columns only once, not three times:

```
  maxes = fltarr(xsize)
  for i = 0L, xsize-1 do maxes(i) = max(foo(i, *))
  for j = 0L, ysize-1 do foo(0, j) = foo(*, j) / maxes
```

Note that it might also be necessary to check if the max of a column is zero.

> Q2) I have a generic array foo=fltarr(a,b). I'd like to copy findgen(b)
> into every column. Any way of doing this without loops ?

Still no. :-(

simple and efficient:

```
  foo = fltarr(3, 5)
  for j = 0L, ysize-1 do foo(*, j) = float(j)
```

If you were doing it over the columns instead of the rows, (i.e., copying findgen(a) into each row), the following would then be more efficient, since again it would work row-by-row:

```
  one_row = findgen(xsize)
  for j = 0L, ysize-1 do foo(0, j) = one_row
```

For cases where a similar operation is to be performed many times, it can be useful to create an indexing array that aids in carrying out the procedure. For example, if the procedure of Q1 were to be

repeated a number of times on different arrays of the same dimension,
the result of Q2 could be used as an indexing array to make the
repititions more efficient (but note the additional demand on
memory!).

```
 ; for example, use the following 'foo':
 xsize = 3
 ysize = 5
 foo = findgen(xsize, ysize)
 ;
 ; Create an index for accessing inv_maxes vector repeatedly:
 inv_maxes_index = make_array(size=size(foo))
 for i = 0L, xsize-1 do inv_maxes_index(i, *) = i
 ;
 ; Repeat the following for each array to be processed (foo, foo2, etc.):
 maxes = fltarr(xsize)
 for i = 0L, xsize-1 do maxes(i) = max(foo(i, *))
 ; check for a column max of '0', that will cause divide problems:
 if ((where(maxes EQ 0.0))(0) NE -1) then $
     message, 'column found with max = 0'
 inv_maxes = 1. / maxes
 ; Now perform the scaling, scanning through the elements of 'foo' and
 ; repeatedly through the inv_maxes vector:
 foo = foo * inv_maxes(inv_maxes_index)
```

Thus the scaling is performed as a single matrix multiply.

Note that in IDL, the handy TEMPORARY function can make the last line
above more efficient, by not making a second copy of 'foo':

```
 foo = temporary(foo) * inv_maxes(inv_maxes_index)
```

Be careful with the above indexes - with an interesting 'enhanced
feature' of PVWave and IDL, when you index an array with an array,
instead of the normal array checking, elements going out of bounds are
simply 'truncated' to the last element. For example:

```
 test = indgen(2)
 test_index = indgen(9)
 print, test(test_index)
       0       1       1       1       1       1       1       1       1
```

--

David Ritscher
Raum 47.2.401
Zentralinstitut fuer Biomedizinische Technik
Albert-Einstein-Allee 47
Universitaet Ulm
D-89069 ULM
Germany

Tel: ++49 (731) 502 5313
Fax: ++49 (731) 502 5315
internet: david.ritscher@zibmt.uni-ulm.de

Subject: Re: Can I do this without using loops?
Posted by steinhh on Tue, 11 Jun 1996 07:00:00 GMT
View Forum Message <> Reply to Message

In article <4pcm41$64r@vixen.cso.uiuc.edu>, santanu@glibm5.cen.uiuc.edu (S Bhattacharyya) writes:
|> Regarding loops, I am kinda in the same boat. My advisor
|> keeps complaining about how slow our code runs...We don't seem to
|> know any better around here :-)
|>
|> Q1) I have a generic array foo(x,y). I'd like to divide each column
|>   by its max. Can this be done without looping ?
|>
|> Q2) I have a generic array foo=fltarr(a,b). I'd like to copy findgen(b)
|>   into every column. Any way of doing this without loops ?
|>

Q2 can actually be solved without loops, quite fast
(if I've understood your question correctly)

```
  foo = rebin(findgen(1,b),a,b,/sample)
```

The "solution" to Q1 is that the "array reduction" operations min/max
(and others!) should get the same functionality as TOTAL, where you can
choose which dimension to total over. If you have a MAX() like that, you
could use something like (foo = foo(a,b))

```
   foo = foo / rebin(reform(max(foo,2),a,1),a,b,/sample)
```

or perhaps (to save some space):

```
   maxfoo = max(foo,2) ;; Max along each column
   foo = temporary(foo) / rebin(reform(maxfoo,a,1),a,b,/sample)
```

Stein Vidar

Subject: Re: Can I do this without using loops?
Posted by Trygve Sparr on Tue, 11 Jun 1996 07:00:00 GMT
View Forum Message <> Reply to Message

In Article<4pcm41$64r@vixen.cso.uiuc.edu>, <santanu@glibm5.cen.uiuc.edu>
writes:

> Q2) I have a generic array foo=fltarr(a,b). I'd like to copy findgen(b)
> into every column. Any way of doing this without loops ?
>

This one is not to difficult. Try:
foo = (fltarr(a) + 1.) # findgen(b)

If you want the rows findgened instead, you get:
foo = findgen(a) # (fltarr(b) + 1.)

Good luck!

--Trygve

Subject: Re: Can I do this without using loops?
Posted by steinhh on Fri, 14 Jun 1996 07:00:00 GMT
View Forum Message <> Reply to Message

In article <31C17DE4.50C@zibmt.uni-ulm.de>, David Ritscher <david.ritscher@zibmt.uni-ulm.de>
writes:

[nice review deleted]

```
|> PRO TEST8, rows, columns
|> strt = systime(1)
|> ;
|> array = rebin(findgen(1, columns), rows, columns)
|> ;
|> print, 'elapsed time: ',systime(1) - strt
|> return
|> ;
|> END
|>
```

[..snip..]

```
|>
|> PRO TEST8_rows, rows, columns
|> strt = systime(1)
|> ;
|> array = rebin(findgen(rows), rows, columns)
|> ;
|> print, 'elapsed time: ',systime(1) - strt
|> return
```

|> ;
|> END
|>

Have you tried these with the /SAMPLE keyword to REBIN as well?
And also: rebin(findgen(rows,1),rows,columns,/sample) for
TEST8_rows?

I should of course try out every piece of this on the local
machines, but that'll wait till later.

Stein Vidar

Subject: Re: Can I do this without using loops?
Posted by David Ritscher on Fri, 14 Jun 1996 07:00:00 GMT
View Forum Message <> Reply to Message

--

David Ritscher
Raum 47.2.401
Zentralinstitut fuer Biomedizinische Technik
Albert-Einstein-Allee 47
Universitaet Ulm
D-89069 ULM
Germany

Tel: ++49 (731) 502 5313
Fax: ++49 (731) 502 5315
internet: david.ritscher@zibmt.uni-ulm.de

This is a comparison of different for-loop and for-loopless approaches
to an indexing problem. A lot can be learned from the timing results.

Once more, regarding the original posting from S Bhattacharyya:

> Q2) I have a generic array foo=fltarr(a,b). I'd like to copy findgen(b)
> into every column. Any way of doing this without loops ?

I stand corrected - there are a couple of ways to accomplish the
second of the two examples (Q2) without using for-loops. I wrote the
various methods into subroutines, and testing the times for each
method, using first a small size that easily fit inside memory, and
then using large arrays where my machine was forced to page a lot. I
ran them on an HP 715 / 64, with 96 Mbyte RAM and 320 Mbyte swapspace.
I ran them both under IDL and PV-Wave, versions: IDL. Version 3.1.0

and PV-WAVE v6.01. Sometimes one was faster, sometimes the other (often by a factor of two or more!). I repeated the tests several times, and the results were quite consistent. I didn't perform any averaging, and didn't do anything to make it all particularly accurate. This should not be considered a benchmark between the two programs. My goal was just to get some feel which methods were more effective.

I compared both the problem of inserting columns (that of Q2), plus the problem of inserting rows of findgen into an array. These two problems turn out to be quite different, with different optimal solutions.

The methods compared are those proposed by:
1. Kenneth P. Bowman
2. me
3. by both David Fanning and Dan Carr (plus some recent news posters)
4. Paul C. Sorenson (paulcs@netcom.com)
5. a brute-force, double-for-loop method (to demonstrate what not to do)
6. exactly like 5 except that the long in the loop variable that is
   assigned to each element not explicitly converted to float (this
   happens explicitly).
7. as in 5., but with the loops reversed, causing thrashing
8., 9. Robert Cannon <rcc@hera.neuro.soton.ac.uk>

I wrote them each into a similar form (see the listing at the end of this note). Here is the algorithm for each. The version written to insert a row instead of a column is similar in form to these. Compare, for example:

```
 insert rows:
  one_column = findgen(1, columns)
  for i = 0L, rows-1 do array(i, 0) = one_column
```

with

```
 insert columns:
  one_row = findgen(rows)
  for j = 0L, columns-1 do array(0, j) = one_row
```

All routines are listed at the end of this note.

The algorithms:

1.
```
array = fltarr(rows, columns, /nozero)
  for j = 0L, columns-1 do array(*, j) = float(j)
```

2.
```
array = fltarr(rows, columns, /nozero)
  one_row = findgen(rows)
  for j = 0L, columns-1 do array(0, j) = one_row
```

3.
```
array = FINDGEN(rows) # REPLICATE(1, columns)
```

4. array = (findgen(1,columns))(bytarr(rows),*)

5. array = fltarr(rows, columns, /nozero)
```
   for j = 0L, columns-1 do $
      for i = 0L, rows-1 do $
  array(i, j) = float(i)
```
6. as in 5., but '= i' instead of '= float(i)'
7. as in 5., but reversed order of the two for statements

8. array = rebin(findgen(1, columns), rows, columns)

9. array = transpose(findgen(columns, rows) mod columns)

Here are the execution times, in seconds, under IDL and PV-Wave, for the specified array sizes, for each of the algorithms above. The left two columns show the times for inserting columns, the second two columns involve inserting rows.

Tested using the following number of rows and columns: 3000, 3000

	Insert columns		Insert rows	
	IDL	PV-Wave	IDL	PV-Wave
1.	22.84	17.58	30.48	21.31
2.	22.77	30.81	4.45	1.48
3.	3.88	9.73	3.87	9.70
4.	12.05	11.93	5.67	7.17
5.	89.24	164.56	87.51	163.15
6.	141.93	385.72	140.45	389.53
7.	92.75	166.44	92.82	169.71
8.	1.30	4.96	9.78	10.29
9.	30.75	28.71	18.40	21.60

Tested using the following number of rows and columns: 1000, 30000

	Insert columns		Insert rows	
	IDL	PV-Wave	IDL	PV-Wave
1.	84.95	71.09	Inf	Inf
2.	Inf	Inf	28.52	22.69
3.	27.65	42.95	30.88	50.10
4.	129.53	138.84	124.39	126.26
5.	300.62	547.51	296.74	556.55
6.	-	1284.83	-	1300.17
7.	Inf	Inf	Inf	Inf
8.	20.07	28.76	Inf	Inf
9.	-	3100.02	158.07	161.23

Where here I poetically define "Inf" to be anything that takes longer
than a day (I suspect these would all end up taking about 10^4 seconds).

A few conclusions can be drawn from the timings:

The most important thing is to differentiate between problems that
fit fully within physical memory (RAM) and those that require the use of
virtual memory. For example, on my 96 Mbyte machine the first example
of using arrays of 3000x3000 floats (= ~36 Mbytes) fit within my
machine. However, increasing the problem to 1000x30000 (= ~120 Mbytes)
makes it exceed my machines RAM, and it becomes necessary for my machine
to swap the array in and out of memory as it works through the elements.
This is a very slow operation, that will almost always be the dominant
factor in determining how long the routine takes. Although this
problem is only about three times larger than the original, the fastest
solutions were 10 times longer. Some solutions, that had taken 20
seconds with the first problem, were still not finished after the
machine had run for a full day.

For problems that do not fit into physical memory it is crucial that all
operations scan over the leftmost dimension first (i.e. rows) before
later dimensions (columns, etc.). The statement array(i, *) = i inside
a for-loop causes the machine to scan through the second dimension
before the first dimension. That turned this seemingly simple statement
into a problem that was not finished a day later. The slow approach of
using double for-loops was many times faster than this more elegant
approach when the for-loops were done in the proper order:
    for j=0L, rows-1 do for i=0L, columns-1 do array(i, j) = ...
but reversing the order of the for-loops turned it into another
problem that was not finished a day later. The best solution of
problems that are inherently row-oriented (such as inserting something
into each row of a matrix) are different from the best solution of
column-oriented problems. These optimal solutions ended up being the
same solutions that were optimal for the problem that fit within
physical memory.

The only solution that performed pretty well for both row-oriented and
column-oriented problems was the elegant cross-product approach
(approach 3.), despite the fact that it requires an extra multiply for
each array element: FINDGEN(rows) # REPLICATE(1, columns)

Comparing the times of solutions 5. and 7., one notes that placing the
for-loops in the wrong order has very little effect on execution time
as long as the problem fits within physical memory, but has a drastic
effect when that is not the case.

The optimal solution to the problem of inserting findgen into rows of

the array ended up being the for-loop looping over the columns, solution 2.:

```
one_row = findgen(rows)
for j = 0L, columns-1 do array(0, j) = one_row
```

The optimal solution for the problem of inserting findgen into the columns was accomplished with rebin, solution 8.:

```
array = rebin(findgen(1, columns), rows, columns)
```

The clever vector indexing approach, solution 4.:

```
(findgen(1,columns))(bytarr(rows),*)
```

performed surprisingly poorly. I can't explain why; it's simply a function of how the interpreter functions with such an indexing scheme.

It was interesting to note the burden that a function call creates when it is inside an inner for-loop. In solution 5. an implicit type conversion is made from long integer to float: array(i, j) = i when this is changed to array(i, j) = float(i) (solution 6.), the execution times are much slower, sometimes more than doubly slow.

One can see from the execution times that at least part of the code for IDL and PVWave are diverging (presumably the memory administration and indexing systems, and perhaps the interpreter); often one language is twice as fast as the other in a given solution, but then the other language is twice as fast on another problem. Often the times are essentially identical. It's clear that neither company has set a goal of perfecting execution speed under all conditions.

As I mentioned, when one does operations along a higher dimension first for a problem that doesn't fit into memory, the problem becomes very slow. I was curious what was happening, because after two days I still hadn't gotten the result from one of these. So I added an extra print statement to my test routine test1_rows.pro to look at the timing as it inserts each row. The best time for this example was about 20 seconds, but each of the 1000 rows took longer than this, due to the thrashing to swap memory in and out. Here are the timings, in seconds for the first few row computations. The time per row keeps growing, but always slower:

```
test1_rows, 1000, 30000

 row    elapsed time    average time/row
   1      13.296000       13.327000
   2      81.090000       40.560500
   3      150.33500       50.121667
   4      218.95900       54.750000
   5      294.14200       58.832600
   6      365.92100       60.992000
   7      435.88800       62.274143
   8      505.99300       63.253000
```

9	579.73700	64.418556
10	649.93200	64.997300
11	721.52000	65.595636
12	790.99900	65.919167
13	863.28500	66.408923
14	936.90500	66.923929
15	1008.2590	67.219333
16	1080.0630	67.505813
17	1154.3780	67.906412
18	1227.2110	68.180111
19	1297.6890	68.301053
20	1369.7120	68.487150
21	1444.2740	68.776429
22	1517.4780	68.977636
23	1589.7140	69.119348

I'm guessing that completion will take something like 100,000 seconds, or about 5,000 times longer than the best time for this solution.

Morals of the story:
A good optimized C compiler worries about a lot of the details of making these things efficient. IDL / PVWave are not designed to optimize the problem for you, so in the case where time becomes crucial, you have to do the optimizing:

1. Avoiding accessing in order of higher dimension first (i.e., columns before rows).
2. Try to avoid an inner for-loop - vectorize the fastest-changing index when possible.
3. Avoid function calls within an inner for-loop. (but this falls under the last, avoid the inner for-loop in general).
4. Careful choice of approach can then optimize a problem between different fairly effective approaches. Interestingly, these solutions can be different between IDL and PV-Wave. One just has to experiment.

****

The same method used in (4) can be applied to the first question:

```
>  Q1) I have a generic array foo(x,y). I'd like to divide each column
>      by its max. Can this be done without looping ?
>
```

And provides a solution where the main looping can be done without a for-loop. I still see no way to avoid using a for-loop to get the maximums for each column.

```
 ; make an example array:
 rows = 3
```

```
columns = 5
foo = findgen(rows, columns)
; create a vector of 1 / maxes for each column:
inv maxes = fltarr(rows)
for i = 0L, rows-1 do inv_maxes(i) = 1. / max(foo(i, *))
;
; Now perform the scaling, scanning through the elements of 'foo' and
; repeatedly through the inv_maxes vector:
foo = foo * inv_maxes(*, bytarr(columns))
```

I didn't check the times on this, but assume it will unfortunately also, like solution 4., perform slowly.

; Here are the actual routines I used for the time testing:

```
PRO TEST1, rows, columns
strt = systime(1)
;
array = fltarr(rows, columns, /nozero)
for j = 0L, columns-1 do array(*, j) = j
;
print, 'elapsed time: ',systime(1) - strt
return
END


PRO TEST2, rows, columns
strt = systime(1)
;
array = fltarr(rows, columns, /nozero)
one_column = findgen(1, columns)
for i = 0L, rows-1 do array(i, 0) = one_column
;
print, 'elapsed time: ',systime(1) - strt
return
END


PRO TEST3, rows, columns
strt = systime(1)
;
array =  replicate(1.0, rows) # findgen(1, columns)
;
print, 'elapsed time: ',systime(1) - strt
```

```
return
END


PRO TEST4, rows, columns
strt = systime(1)
;
array = (findgen(1,columns))(bytarr(rows),*)
;
print, 'elapsed time: ',systime(1) - strt
return
END


PRO TEST5, rows, columns
strt = systime(1)
;
array = fltarr(rows, columns, /nozero)
for j = 0L, columns-1 do $
   for i = 0L, rows-1 do $
      array(i, j) = j
;
print, 'elapsed time: ',systime(1) - strt
return
;
END


PRO TEST6, rows, columns
strt = systime(1)
;
array = fltarr(rows, columns, /nozero)
for j = 0L, columns-1 do $
   for i = 0L, rows-1 do $
      array(i, j) = float(j)
;
print, 'elapsed time: ',systime(1) - strt
return
;
END


PRO TEST7, rows, columns
strt = systime(1)
;
array = fltarr(rows, columns, /nozero)
   for i = 0L, rows-1 do $
for j = 0L, columns-1 do $
```

```
      array(i, j) = j
;
print, 'elapsed time: ',systime(1) - strt
return
;
END


PRO TEST8, rows, columns
strt = systime(1)
;
array = rebin(findgen(1, columns), rows, columns)
;
print, 'elapsed time: ',systime(1) - strt
return
;
END


PRO TEST9, rows, columns
strt = systime(1)
;
array = transpose(findgen(columns, rows) mod columns )
;
print, 'elapsed time: ',systime(1) - strt
return
;
END



; the following are similar to the above, but insert rows intead of
; columns of 'findgen's:

PRO TEST1_rows, rows, columns
strt = systime(1)
;
array = fltarr(rows, columns, /nozero)
for i = 0L, rows-1 do array(i, *) = i
;
print, 'elapsed time: ',systime(1) - strt
return
END


PRO TEST2_rows, rows, columns
strt = systime(1)
;
```

```
array = fltarr(rows, columns, /nozero)
one_row = findgen(rows)
for j = 0L, columns-1 do array(0, j) = one_row
;
print, 'elapsed time: ',systime(1) - strt
return
END


PRO TEST3_rows, rows, columns
strt = systime(1)
;
array =  findgen(rows) # replicate(1.0, 1, columns)
;
print, 'elapsed time: ',systime(1) - strt
return
END


PRO TEST4_rows, rows, columns
strt = systime(1)
;
array = (findgen(rows))(*, bytarr(columns))
;
print, 'elapsed time: ',systime(1) - strt
return
END


PRO TEST5_rows, rows, columns
strt = systime(1)
;
array = fltarr(rows, columns, /nozero)
for j = 0L, columns-1 do $
   for i = 0L, rows-1 do $
      array(i, j) = i
;
print, 'elapsed time: ',systime(1) - strt
return
;
END


PRO TEST6_rows, rows, columns
strt = systime(1)
;
array = fltarr(rows, columns, /nozero)
for j = 0L, columns-1 do $
```

```
   for i = 0L, rows-1 do $
      array(i, j) = float(i)
;
print, 'elapsed time: ',systime(1) - strt
return
;
END


PRO TEST7_rows, rows, columns
strt = systime(1)
;
array = fltarr(rows, columns, /nozero)
   for i = 0L, rows-1 do $
for j = 0L, columns-1 do $
      array(i, j) = i
;
print, 'elapsed time: ',systime(1) - strt
return
;
END


PRO TEST8_rows, rows, columns
strt = systime(1)
;
array = rebin(findgen(rows), rows, columns)
;
print, 'elapsed time: ',systime(1) - strt
return
;
END


PRO TEST9_rows, rows, columns
strt = systime(1)
;
array = findgen(rows, columns) mod rows
;
print, 'elapsed time: ',systime(1) - strt
return
;
END
```

File Attachments

1) idl2.txt, downloaded 92 times

Subject: Re: Can I do this without using loops?
Posted by bowman on Sat, 15 Jun 1996 07:00:00 GMT
View Forum Message <> Reply to Message

In article <31C17DE4.50C@zibmt.uni-ulm.de>, David Ritscher
<david.ritscher@zibmt.uni-ulm.de> wrote:

> The algorithms:
> 1. array = fltarr(rows, columns, /nozero)
> for j = 0L, columns-1 do array(*, j) = float(j)

> 3. array = FINDGEN(rows) # REPLICATE(1, columns)

> Here are the execution times, in seconds, under IDL and PV-Wave, for
> the specified array sizes, for each of the algorithms above. The left
> two columns show the times for inserting columns, the second two
> columns involve inserting rows.
>
> Tested using the following number of rows and columns: 3000, 3000
>
> Insert columns Insert rows
> IDL PV-Wave IDL PV-Wave
> 1. 22.84 17.58 30.48 21.31

> 3. 3.88 9.73 3.87 9.70
>
> Tested using the following number of rows and columns: 1000, 30000
>
> Insert columns Insert rows
> IDL PV-Wave IDL PV-Wave
> 1. 84.95 71.09 Inf Inf

> 3. 27.65 42.95 30.88 50.10

I confess to being surprised by how fast (3) is compared to (1). I always
thought these tricks using the # operator to avoid loops were 'too clever
by half'. At least on the face of it, (3) creates an unnecessary
temporary vector and does useless multiply operations. The multiplies can
probably be perfectly overlapped with the stores on a RISC machine (and so
lead to no additional cost), but even so it seems more complicated. There
may still be aspects of (1) that are interpreted, (the loop probably is),
but it would seem to be the simplest possible operation to write as
optimized code.

Thanks for the lesson.

This does point out how useful a good IDL profiling tool would be.

Ken Bowman

--
Kenneth P. Bowman, Assoc. Prof. 409-862-4060
Department of Meteorology 409-862-4132 fax
Texas A&M University bowman@csrp.tamu.edu
College Station, TX 77843-3150

Subject: Re: Can I do this without using loops?
Posted by David Ritscher on Sat, 15 Jun 1996 07:00:00 GMT
View Forum Message <> Reply to Message

Snipped out from Stein Vidar's post:

> Have you tried these with the /SAMPLE keyword to REBIN as well?
> And also: rebin(findgen(rows,1),rows,columns,/sample) for
> TEST8_rows?

No, I only tested the 18 programs listed.

I just did a quick check on rebin using the /sample keyword. Here are the
comparison of routines test8 and test8_rows, relative execution times
(i.e., with compared to without the /sample keyword):

Ratios of execution times of rebin(/sample) / rebin():

problem with 3000, 3000 rows and columns:

	IDL	PV-Wave
test8	1.22997	0.676364
test8_rows	0.999363	0.956496

problem with 1000, 30000 rows and columns:

	IDL	PV-Wave
test8	1.06719	0.936516

> I should of course try out every piece of this on the local

> machines, but that'll wait till later.

I'll be interested to hear what you get for timings on all of these.
Thinks could perform very differently under different conditions.

--

David Ritscher
Raum 47.2.401
Zentralinstitut fuer Biomedizinische Technik
Albert-Einstein-Allee 47
Universitaet Ulm
D-89069 ULM
Germany

Tel: ++49 (731) 502 5313
Fax: ++49 (731) 502 5315
internet: david.ritscher@zibmt.uni-ulm.de

Subject: Re: Can I do this without using loops?
Posted by steinhh on Sun, 16 Jun 1996 07:00:00 GMT
View Forum Message <> Reply to Message

Here are some timing results for an Alphaserver 2100 with 4 CPUs installed and about 1 GB internal memory (which means that the large test didn't swap, I think. However, the .SIZE command in IDL only takes unsigned integers for the number of Kilobytes to reserve, so some tests bombed because of internal memory limits in IDL). I made sure that only 2 other jobs were executing concurrently, so one CPU was more or less dedicated to running this test. (With the fourth doing housekeeping).

I'm not sure about the cache size or processor speed, though.

I added methods 10 and 11, (included at the end of the article) to test the /sample keywords, and to try a combination of the fast column insertion (method 8) with a transpose.

I don't have pv-wave, but just for fun I tested both Idl 3.6.1c and Idl 4.0, and I got a surprise! Unless I've made some strange mistakes, the comparison is quite counterintuitive.

Rows and columns: 3000 3000

IDL 3.6.1c 4.0 | 3.6.1c 4.0

```
 1.   4.64    7.16  |      8.16   10.10
 2.   6.92    7.60  |      0.71    0.74
 3.   0.96    1.39  |      0.92    1.36
 4.   2.25    2.90  |      2.20    2.98
 5.  17.58   21.84  |     17.56   21.83
 6.  32.78   38.85  |     32.22   38.94
 7.  18.98   21.38  |     18.96   21.67
 8.   0.73    0.75  |      3.64    3.69
 9.   9.40   11.51  |      5.49    7.22
10.   0.69    0.69  |      3.65    3.70
11.   0.69    0.69  |      4.75    5.38
```

Rows and columns: 1000 30000

```
    3.6.1c    4.0   |    3.6.1c     4.0

 1.  15.88   24.00  |     17.98    24.87
 2.  14.21   16.70  |      2.77     2.81
 3.   2.97    4.52  |      3.02     4.52
 4.******* *******  |   *******  *******
 5.  58.70   72.90  |     58.67    72.88
 6. 109.13  129.85  |    107.16   129.93
 7.  60.89   69.90  |     60.94    71.47
 8.   2.75    2.78  |      5.35     5.71
 9.******* *******  |     19.64    24.75
10.   2.71    2.72  |      5.42     5.48
11.   2.70    2.79  |    *******  *******
```

Note that version 4.0 is actually *always* slower than version 3.6.1c. Is the "upgrade" to 4.0 coincident with the switch from fortran to C source code? In that case, maybe RSI should consider switching back (at least for some core routines), if these numbers are correct.

Here are the 3.6.1c results relative to the best 3.6.1c results, and 4.0 results relative to the corresponding 3.6.1c results:

Rows and columns: 3000 3000

```
    3.6.1c    4.0   |    3.6.1c    4.0

 1.   6.72    1.54  |     11.49    1.24
 2.  10.03    1.10  |      1.00    1.04
 3.   1.39    1.45  |      1.30    1.47
 4.   3.26    1.29  |      3.10    1.35
 5.  25.48    1.24  |     24.73    1.24
 6.  47.51    1.19  |     45.38    1.21
```

```
 7.  27.51    1.13  |     26.70    1.14
 8.   1.06    1.03  |      5.13    1.01
 9.  13.62    1.22  |      7.73    1.32
10.   1.00    1.00  |      5.14    1.01
11.   1.00    1.00  |      6.69    1.13
```

Rows and columns: 1000 30000

```
    3.6.1c    4.0   |    3.6.1c    4.0

 1.   5.88    1.50  |      6.49    1.38
 2.   5.26    1.17  |      1.00    1.01
 3.   1.10    1.52  |      1.09    1.50
 4.******* *******  |   *******  *******
 5.  21.74    1.24  |     21.18    1.24
 6.  40.42    1.19  |     38.69    1.21
 7.  22.55    1.14  |     22.00    1.17
 8.   1.02    1.01  |      1.93    1.07
 9.******* *******  |      7.09    1.26
10.   1.00    1.00  |      1.95    1.01
11.   1.00    1.03  |   *******  *******
```

For a slower alpha machine (front cover says DEC 3000) I only bothered to time the small test. Here the differences between version 3.6 and 4.0 are slightly less systematic.

Rows and columns: 3000 3000

```
IDL 3.6.1c    4.0    |    3.6.1c    4.0
 1.  11.96   11.63   |     14.27   14.62
 2.   7.83    9.88   |      0.85    0.88
 3.   1.71    2.83   |      1.71    2.83
 4.  19.18   21.98   |     20.64   17.82
 5.  39.89   43.95   |     40.09   43.98
 6.  81.61   98.88   |     80.17  101.32
 7.  42.68   50.47   |     41.92   48.96
 8.   0.92    1.43   |      3.91    3.88
 9.  41.61   48.65   |      9.87   13.07
10.   0.85    0.85   |      3.92    3.85
11.   0.91    1.39   |     42.91   29.58
```

Relative results:

```
 1.  14.07   0.97  |     16.79       1.02
 2.   9.21   1.26  |      1.00       1.04
 3.   2.01   1.65  |      2.01       1.65
 4.  22.56   1.15  |     24.28       0.86
 5.  46.93   1.10  |     47.16       1.10
```

```
 6.  96.01   1.21  |      94.32        1.26
 7.  50.21   1.18  |      49.32        1.17
 8.   1.08   1.55  |       4.60        0.99
 9.  48.95   1.17  |      11.61        1.32
10.   1.00   1.00  |       4.61        0.98
11.   1.07   1.53  |      50.48        0.69
```

For a DECstation 5000/240 I only ran an even smaller test
(1000,1000), with the following results:

```
Rows and columns:    1000    1000
IDL 3.6.1c    4.0   |    3.6.1c     4.0
 1.   4.66   12.90  |     4.84     11.67
 2.   2.60    2.80  |     0.11     0.09
 3.   0.91    0.93  |     0.90     0.98
 4.   1.72    1.66  |     1.22     1.25
 5.  13.10   12.53  |    13.34    12.40
 6.  28.32   26.28  |    21.76    25.91
 7.  13.82   12.93  |    13.98    12.93
 8.   0.40    0.33  |     0.70     0.71
 9.   4.02    4.07  |     3.11     3.11
10.   0.19    0.24  |     0.58     0.66
11.   0.32    0.33  |     1.11     1.12
```

3.6.1c results relative to the best 3.6.1c results,
and 4.0 results relative to the corresponding 3.6.1c results:

```
 1.  24.53   2.77  |      44.00        2.41
 2.  13.68   1.08  |       1.00        0.82
 3.   4.79   1.02  |       8.18        1.09
 4.   9.05   0.97  |      11.09        1.02
 5.  68.95   0.96  |     121.27        0.93
 6. 149.05   0.93  |     197.82        1.19
 7.  72.74   0.94  |     127.09        0.92
 8.   2.11   0.83  |       6.36        1.01
 9.  21.16   1.01  |      28.27        1.00
10.   1.00   1.26  |       5.27        1.14
11.   1.68   1.03  |      10.09        1.01
```

In other words, even less systematic differences. I'm still a bit
surprised that *any* of these numbers go the "wrong" way.

```
.....................................................
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
```

```
PRO TEST10,rows,columns
  strt = systime(1)
;
```

```
  array = rebin(FINDGEN(1, columns), rows, columns,/sample)
;
  time = systime(1)-strt
  PRINT, 'elapsed time: ',time
  RETURN
;
END


;; Identical to number 8 (just filling in for completeness)
;; The row version uses this plus a transpose
PRO test11,rows,columns
  strt = systime(1)
;
  array = rebin(FINDGEN(1, columns), rows, columns)
;
  time = systime(1)-strt
  PRINT, 'elapsed time: ',time
  RETURN
END


PRO test10_rows,rows,columns
  strt = systime(1)
;
  array = rebin(FINDGEN(rows,1), rows, columns,/sample)
;
  time = systime(1)-strt
  PRINT, 'elapsed time: ',time
  RETURN
;
END

PRO test11_rows,rows,columns
  strt = systime(1)
;
  array = TRANSPOSE(rebin(FINDGEN(1,rows), columns, rows,/sample))
;
  time = systime(1)-strt
  PRINT, 'elapsed time: ',time
  RETURN
;
END
```