Subject: Efficient pattern-matching in a large array
Posted by Chris O'Dell on Tue, 02 Aug 2011 18:05:01 GMT
View Forum Message <> Reply to Message

I have a byte array of tens of millions values (read from a binary file). I want to find an 8-byte pattern where ever it occurs. Similar to where, but 8 values at a time instead of one. The problem is finding a way to do this that is as fast as possible. The pattern is: [234, 203, 138, 216, 21, 52, 117, 39].

Right now, I have this in a big loop and it takes a while. We tried "match2" but it ran out of memory.

Thanks,
Chris

Subject: Re: Efficient pattern-matching in a large array
Posted by Foldy Lajos on Tue, 02 Aug 2011 20:04:19 GMT
View Forum Message <> Reply to Message

On Tue, 2 Aug 2011, Chris O'Dell wrote:

> I have a byte array of tens of millions values (read from a binary
> file). I want to find an 8-byte pattern where ever it occurs. Similar
> to where, but 8 values at a time instead of one. The problem is
> finding a way to do this that is as fast as possible. The pattern is:
> [234, 203, 138, 216, 21, 52, 117, 39].
>
> Right now, I have this in a big loop and it takes a while. We tried
> "match2" but it ran out of memory.
>
> Thanks,
> Chris
>
>

Hint: use ULONG64 (8 byte) comparison:

```
n=1000
barr=bindgen(n)
bpat=[42b, 43b, 44b, 45b, 46b, 47b, 48b, 49b]
lpat=fix(bpat, 0, type=15)

for j=0,7 do begin
    larr=fix(barr, j, j eq 0 ? n/8 : n/8-1, type=15)
    w=where(larr eq lpat, count)
```

```
    if count gt 0 then print, 'found:', 8*w+j
endfor
```

regards,
Lajos

Subject: Re: Efficient pattern-matching in a large array
Posted by ez569x on Fri, 12 Aug 2011 12:18:30 GMT
View Forum Message <> Reply to Message

Lajos,
can you explain the line:

larr=fix(barr, j, j eq 0 ? n/8 : n/8-1, type=15)

it looks like you're trying to form the ULONG64 but the ? and : are tripping me up.

Thanks,
Sam

Subject: Re: Efficient pattern-matching in a large array
Posted by Brian Daniel on Fri, 12 Aug 2011 14:38:52 GMT
View Forum Message <> Reply to Message

On Aug 12, 8:18 am, SpinMan <ez5...@gmail.com> wrote:
> Lajos,
> can you explain the line:
>
> larr=fix(barr, j, j eq 0 ? n/8 : n/8-1, type=15)
>
> it looks like you're trying to form the ULONG64 but the ? and : are tripping me up.
>
> Thanks,
> Sam

Sam,

Lajos is using a Conditional Expression. Its listed in the help under
Other Expressions. IF j EQ 0 THEN the value is n/8 ELSE the value is
n/8-1.

-B

Subject: Re: Efficient pattern-matching in a large array
Posted by guillermo.castilla.ca on Tue, 16 Aug 2011 16:41:15 GMT
View Forum Message <> Reply to Message

> Hint: use ULONG64 (8 byte) comparison:...

Lajos, this is an awesome trick, hats off!!
It took me a while to understand it, so assuming there are more people out there who are intrigued by this (and also for myself, to understand it at a later date), here's a layman explanation:
Chris has a large 1-D array of Byte type (i.e., numbers ranging from 0-255). He wants to find occurrences of a specific sequence (what he calls 'pattern') of eight numbers (in his case [234, 203, 138, 216, 21, 52, 117, 39]). It happens that ULONG64 numbers are stored as a sequence of eight Byte numbers (in his case his 8 byte sequence = 2843236008186268650). Therefore you can convert each sequence of eight consecutive numbers in the array to a ULONG64 number, and then the problem is reduced to finding occurrences of that number in the 8 shifted,reduced (by a factor of 1/8) ULONG64 versions of the original array (where the shift is performed using the offset argument in the FIX function). Brilliant!

Lajos, I was wondering if your solution can be generalized to patterns of any length, could it?

Guillermo

Subject: Re: Efficient pattern-matching in a large array
Posted by JDS on Tue, 16 Aug 2011 23:01:39 GMT
View Forum Message <> Reply to Message

Using ULONG64's is quite interesting, but the byte offsetting slows performance compared to what you might expect. In the likely case that the byte pattern occurs rarely, a thinned WHERE approach is at least twice as fast in my testing:

```
w=where(array eq pat[(c=0)],/NULL)
while keyword_set(w) && ++c lt n_elements(pat) do begin
  keep=where(array[w+c<n] eq pat[c],/NULL)
  w=w[keep]
endwhile
```

It also has the advantage on working for any pattern length, and returning the "hits" in sorted order. Finding around 5 8-byte needles in a 100 million byte haystack took about 1/3 sec with this algorithm on my machine (though Lajos' method was close behind at only 0.7s).

JD

Subject: Re: Efficient pattern-matching in a large array
Posted by guillermo.castilla.ca on Wed, 17 Aug 2011 13:41:19 GMT
View Forum Message <> Reply to Message

On Aug 16, 8:01 pm, JDS <jdtsmith.nos...@yahoo.com> wrote:
> Using ULONG64's is quite interesting, but the byte offsetting slows performance compared to what you might expect. In the likely case that the byte pattern occurs rarely, a thinned WHERE approach is at least twice as fast in my testing:

Wow, the thinned WHERE approach rocks!!

You are right, shifting or offsetting takes a lot of time (god knows why). I tried the following approach (looking for sequences of numbers that yield the same sum than the pattern, and then double checking that they actually form the sought pattern):

```
carr=total(array,/cumulative,/integer)
w=where(Shift(carr,-n_elements(pat))-carr eq Total(pat,/integer),count)
+1
if count ne 0 then for j=0,count-1 do $
  if ~array_equal(array[w[j]:w[j]+n_elements(pat)-1] eq pat, 1) then
w[j]=-1
w=w[where(w ge 0, count,/null)]
```

It takes almost 10 times more than JD's method (1.27s vs. 0.13s) when I set array=byte(lindgen(100000000))and pat=byte([1,2,3,4,5,6,7,8]), yikes! Half of time is invested in the 2nd line, and a fifth in the 1st line, meaning that TOTAL and SHIFT suck a lot of time. Btw, in this example, with almost 40k patterns in the array, JD's method is still twice as fast as Lajos (0.55s), so JD's thinned where approach is definitely the method of choice for searching for specific sequences of consecutive numbers in an array.

Guillermo

Subject: Re: Efficient pattern-matching in a large array
Posted by Bob[4] on Wed, 17 Aug 2011 21:14:00 GMT
View Forum Message <> Reply to Message

This is nice but it would seem an incorrect use of keyword_set(w) in the while statement? If the where statement returns [0] (a possibly valid match) then the while statement will not run. Thus one of the standard approaches for detecting a successful where is needed (i.e., check that count gt 0 or pat[0] gt -1 w/o the /NULL).

ASIDE: IDL does not let you subscript a !null variable as a length one array.

IDL> a = !NULL

```
IDL> print, a[0]
% Variable is undefined: A.
% Execution halted at: $MAIN$
```

Why not? Seems to me it should print !NULL. This would be occasionally useful, although it would not help with the above problem.

Subject: Re: Efficient pattern-matching in a large array
Posted by JDS on Thu, 18 Aug 2011 17:00:14 GMT
View Forum Message <> Reply to Message

Right you are, Bob! Thanks for the catch.

I struggled to find a simple logical test with the properties:

1) any array is true
2) !NULL is false

And I didn't remember that KEYWORD_SET looks only at the first entry. I think *any array* should be true on its own, but IDL logic treats scalars and single length arrays as identical, and issues an error when IF-testing longer arrays. This is maximally confusing, since you often don't know the tested array's length in advance, and an error is not thrown when that length drops to 1. I think David has an article somewhere about the resulting confusion.

In any case, you don't have to give up on !NULL for the test. In fact you can take !NULL even further, applying it without testing for WHERE's success at all (one of the main uses of !NULL). This further simplifies the algorithm:

```
w=where(a eq pat[(c=0)],/NULL)
while w ne !NULL && ++c lt n_elements(pat) do w=w[where(a[w+c] eq pat[c],/NULL)]
w=w[where(w le n-n_elements(pat),/NULL)]
```

This will work even if the pattern occurs right away. That last test is new and is for one corner case: if the pattern contains a string of repeating values at its end, and this value matches the very last element of the array, you will otherwise get a spurious match.

By the way, this uses the curious fact that comparison to !NULL is special cased in IDL (since you can't have an array of !NULL's):

```
IDL> a=[1,2,4,5]
IDL> print, a ne 2
  1   0   1   1
IDL> print, a ne !NULL
  1
```

JD
