Subject: Re: Accelerating a one-line program doing matrix multiplication
Posted by on Thu, 30 Sep 2010 11:05:36 GMT
View Forum Message <> Reply to Message

On Sep 30, 11:41 am, Axel M <axe...@gmail.com> wrote:
> On Sep 30, 10:39 am, Axel M <axe...@gmail.com> wrote:
>
>
>
>> On Sep 29, 6:57 pm, Karl <karl.w.schu...@gmail.com> wrote:
>
>>> On Sep 29, 10:05 am, Paolo <pgri...@gmail.com> wrote:
>
>>>> On Sep 29, 11:55 am, Axel M <axe...@gmail.com> wrote:
>
>>>> > On 29 Sep., 17:45, Paulo Penteado <pp.pente...@gmail.com> wrote:
>
>>>> > > On Sep 29, 12:24 pm, Axel M <axe...@gmail.com> wrote:
>
>>>> > > > Great, I did not know about this construction, and honestly I do not
>>>> > > > understand how it works (is there any documentation about it?).
>>>> > > > Anyways, I tried it, and unfortunately I saw that it needed ~20%
>>>> > > > longer (the complete function, not the rebin only). So, it is not
>>>> > > > faster.. but it is great though.
>
>>>> > > It is replicating a structure of a single field which contains the
>>>> > > array input ({temp:input}), then selecting only a single field (the
>>>> > > first, 0) of the resulting structure array. Documentation for this
>>>> > > would be on creation and use of structures.
>
>>>> > Ok, I got it. Thanks! Then probably it is the memory allocation for
>>>> > the array of structures which takes so long... it would be great if
>>>> > the ITT people would develop a _fast_ vector replicate, I fear
>>>> > rebinning is not the best option.
>
>>>> > In any case, based on the answers, I assume that my problem is rather
>>>> > on the matrix multiplication part, so I can probably do nothing for
>>>> > that.
>
>>>> > Thanks a lot
>
>>>> well considering your original problem - you need to apply
>>>> a linear transformation to N vectors v_i=(x_i,y_i,z_i),
>>>> for i going from 0 to a large N, right?
>
>>>> I would just explicitely compute the transformed vectors
>
>>>> z_i=(xx_i,yy_i,zz_i)

```
>
>>>>  by just writing out in the program the computation for every
>>>>  component,
>>>>  i.e.
>
>>>>  xx=x*c1+y*c2+z*c3+c4
>>>>  and same for yy,zz with appropriate constant coefficients c1,c2,c3,c4
>>>>  (that are the same for all i).
>
>>>>  But then maybe i misunderstood the problem...
>
>>>>  Ciao,
>>>>  Paolo
>
>>>  Yeah, I think you are right.
>
>>>  Another way to see it:
>
>>>  FUNCTION vc2rc, v0,v1,v2,v3,vc
>>>         xform = [[v1],[v2],[v3]]
>>>         n = <number of points in vc>
>>>         for i=0, n-1
>>>            temp = vc[*,i]
>>>            temp = temp # xform + v0
>>>            vc[*,i] = temp
>>>         end
>>>  END
>
>>>  This assumes that you can change vc itself and that v0 is a 3-vector.
>>>  In this case, there is only one copy of the point array, as it is
>>>  being transformed in place.  In other schemes, there may have been as
>>>  many as three or four copies.  If it is not OK to change vc, then this
>>>  function would have to make a vr array of the same shape as vc and
>>>  return it.  But it is still the best solution as far as memory goes.
>
>>>  Yeah, the for loop is going to be slow, but a test will tell if it is
>>>  faster than other approaches.  If the program causes paging to disk
>>>  with the original approach, then the for loop may be faster.  If speed
>>>  is really, really important, then the above can be implemented in a C
>>>  DLM.
>
>>>  And yes, the three lines with "temp" can be collapsed into one, but
>>>  IDL will make small temps anyway here and so a single line may not be
>>>  much faster.  I left it as three lines for clarity.
>
>>  Hi,
>
>>  Thanks for the idea. I tried it, below is the function code (original
```

>> and "accelerated" with your idea) and the test code. By explicitly
>> applying the linear transformation (_accel version) within a loop it
>> took 15 times longer... I guess IDL does this better with the #
>> operator.
>
>> I still think I can most definitely gain time by using the fact that
>> vc represents just all indexes of an array, but I have to find out how
>> to exploit this property...
>

```
>> FUNCTION vc2rc, v0,v1,v2,v3,vc
>>       RETURN, [[v1],[v2],[v3]] # vc + REBIN(v0, SIZE(vc, /DIMENSIONS))
>> END
>
>> FUNCTION vc2rc_accel, v0,v1,v2,v3,vc
>>       npoints = (SIZE(vc, /DIMENSIONS))[1]
>>       for i=0L, npoints-1 DO BEGIN
>>             vc[*,i] = vc[0,i] * v1 + vc[1,i] * v2 + vc[2,i] * v3 + v0
>>       endfor
>>       RETURN, vc
>> END
>
>> PRO testspeed
>>       dims = [100,100,100]
>>       i = LINDGEN(LONG(dims[0])*dims[1]*dims[2]) ;image dimensions
>>       vc = TRANSPOSE([[(i MOD dims[0])], [((i / dims[0]) MOD
>> (dims[1]))], [(i / (dims[0] * dims[1]))]])
>>       v0=[5,5,5] ;origin
>>       v1=[1.0,0,0] ;vectors
>>       v2=[0,1.0,0]
>>       v3=[0,0,2.0]
>
>>       t0 =SYSTIME(/SECONDS)
>>       rc = vc2rc_accel(v0,v1,v2,v3,vc)
>>       rc = 0 & vc = TRANSPOSE([[(i MOD dims[0])], [((i / dims[0]) MOD
>> (dims[1]))], [(i / (dims[0] * dims[1]))]])
>>       rc = vc2rc_accel(v0,v1,v2,v3,vc)
>>       print, 'Time: ', STRING(SYSTIME(/SECONDS) - t0)
>
>>       rc = 0 & vc = TRANSPOSE([[(i MOD dims[0])], [((i / dims[0]) MOD
>> (dims[1]))], [(i / (dims[0] * dims[1]))]])
>
>>       t0 =SYSTIME(/SECONDS)
>>       rc = vc2rc(v0,v1,v2,v3,vc)
>>       rc = 0 & vc = TRANSPOSE([[(i MOD dims[0])], [((i / dims[0]) MOD
>> (dims[1]))], [(i / (dims[0] * dims[1]))]])
>>       rc = vc2rc(v0,v1,v2,v3,vc)
>>       print, 'Time: ', STRING(SYSTIME(/SECONDS) - t0)
>
```

```
>>        rc = 0 & vc = TRANSPOSE([[(i MOD dims[0])], [((i / dims[0]) MOD
>>  (dims[1]))], [(i / (dims[0] * dims[1]))]])
>
>>        t0 =SYSTIME(/SECONDS)
>>        rc = vc2rc_accel(v0,v1,v2,v3,vc)
>>        rc = 0 & vc = TRANSPOSE([[(i MOD dims[0])], [((i / dims[0]) MOD
>>  (dims[1]))], [(i / (dims[0] * dims[1]))]])
>>        rc = vc2rc_accel(v0,v1,v2,v3,vc)
>>        print, 'Time: ', STRING(SYSTIME(/SECONDS) - t0)
>>  END
>
>  Hi again,
>
>  I found a solution which is ~20% faster, receiving the dimensions of
>  the image directly instead of the "vc" points (since, as I said, it is
>  in this case where speed really becomes an issue). It is doing
>  additions rather than multiplications, which appears to work faster.
>
>  FUNCTION rc_fromimage, v0,v1,v2,v3,dims
>        RETURN, REBIN(v1 # INDGEN(dims[0]), [3, dims]) + REBIN(REFORM(v2 #
>  INDGEN(dims[1]), 3, 1, dims[1], 1), [3, dims]) + REBIN(REFORM(v3 #
>  INDGEN(dims[2]), 3, 1, 1, dims[2]), [3, dims])
>  END
>
>  within testspeed, the tests looks like "rc =
>  rc_fromimage(v0,v1,v2,v3,dims)"
```

Sorry, small correction adding v0 which I forgot before:

```
FUNCTION rc_fromdims, v0,v1,v2,v3,dims
 RETURN, $
  REBIN(v0, [3, dims], /SAMPLE) + $
  REBIN(v1 # INDGEN(dims[0]), [3, dims], /SAMPLE) + $
  REBIN(REFORM(v2 # INDGEN(dims[1]), 3, 1, dims[1], 1), [3, dims], /
SAMPLE) + $
  REBIN(REFORM(v3 # INDGEN(dims[2]), 3, 1, 1, dims[2]), [3, dims], /
SAMPLE)
END
```