Subject: Chunk Array Decimation
Posted by JD Smith on Tue, 01 Oct 2002 19:32:54 GMT
View Forum Message <> Reply to Message

Histogram lovers (and haters alike):

Carsten put this question to me, and I've wasted too much time on it not to report the results. It's a deceptively simple-seeming problem.

You have a vector of indices, `inds', and a data vector `data' of the same length. The `inds' vector contains a list of many repeated indices, e.g.:

```
[4,2,6,4,1,5,6,6,3,1,7,2]
```

The job is to generate a result vector `vec' of length max(inds)+1, such that for each index i:

```
vec[i]=total(data[where(inds eq i)])
```

or zero otherwise. That is, gather all data with a given corresponding index, and total them together into the result vector at that index. If no indices are repeated, you can of course just use simple assignment, but that's a much simpler problem. Here's a very straightforward implementation based on this idea:

```
mx=max(inds)
vec1=fltarr(mx+1)
for j=0L,mx do begin
  wh=where(inds eq j,cnt)
  if cnt eq 0 then continue
  vec1[j]=total(data[wh])
endfor
```

This is slow. Very slow. Verrrrrrrrrry slow. And wasteful. You search through the index vector using where() many, many times. Beware of constructs like this. Surely we can do better. OK, how about a very literal, straight loop approach, just like we'd write in C?

```
mx=max(inds)
vec2=fltarr(mx+1)
for j=0L,n_elements(data)-1L do $
  vec2[inds[j]]=vec2[inds[j]]+data[j]
```

Accumulate based on the index. Not too bad. Runs much faster than using where(), but its run-time scales with the number of elements of data, independent to how many repeated indices there are.

Of course, anyone familiar at all with histogram() would realize
there's a better route when many indices are repeated:

```
mx=max(inds)
vec3=fltarr(mx+1)
h=histogram(inds,reverse_indices=ri,OMIN=om)
for j=0L,n_elements(h)-1 do if ri[j+1] gt ri[j] then $
  vec3[j+om]=total(data[ri[ri[j]:ri[j+1]-1]])
```

This taps into the ever-so useful reverse indices vector to pick out
those elements of data which fall in each "bin" of the index
histogram. Notice I'm using OMIN to save time in case the minimum
index is greater than 0. This is much faster than the where() method,
and can be a factor of 2 or 3 faster than the literal loop approach,
if indices are repeated at least a few times on average (a few drops
in each histogram bin). If indices are never repeated, or especially
if many indices are skipped (a *sparse* set), the literal loop method
can be much faster than histogram.

This is all well and good, but Carsten was amazed that I'd offered a
loop in a solution. I wondered whether a loop-free and possibly
faster version existed.

Given the initial set of reverse indices, the problem reduces to one
of: is there a loop-free way to decimate an array using a variable
width "chunk" decimation? E.g.: total the first 5, then the next 3,
then the next 0, then the next 7, ..., elements of an array. I was
encouraged by the histogram(total(/CUMULATIVE)) method which solved
Pavel's chunk fill problem of years past, and came up with:

```
mx=max(inds)
h1=histogram(inds,OMIN=om,REVERSE_INDICES=ri1)
col=ri1[n_elements(h1)+1:*]
 h2=histogram(total(h1,/cumulative)-1,MIN=0,reverse_indices=r i2)
row=ri2[0:n_elements(h2)-1]-ri2[0]+om ; chunk indices = row number
sparse_array=sprsin(col,row,replicate(1.,n_ind),(mx+1)>n_ind)
if mx ge n_ind then $
  vec4=sprsax(sparse_array,[data,replicate(0.,mx+1-n_ind)]) $
else vec4=(sprsax(sparse_array,data))[0:mx]
```

I use sparse arrays to solve the chunk decimation problem, with the
chunk fill method generating the row numbers of non-zero (unity
actually) entries, and the original reverse indices generating the
column numbers. Unfortunately, RSI's Numerical Recipes-based sparse
array routines demand square arrays (which seems unnecessary to me),
so you either have to pad the data or truncate the result, depending
on whether you have more repeated indices than skipped indices. Even

so, this method is at least 10 times faster than the former histogram() method for relatively dense (many repeated index) mappings! For sparse sets of indices, it still works (thanks to that if statement at the end), and, amazingly, can still beat the literal loop method! Such is the penalty for looping at all using IDL variables, that you're better off going to elaborate lengths creating sparse array structures and histograms just to get all your looping to occur in real compiled code.

For a random list of 20,000 indices drawn from 0-2000 (a dense sampling: each index repeated 10 times, on average), the methods time to (average, sec):

```
0.48   ; where()
0.024  ; literal loop
0.011  ; histogram() loop
0.0096 ; sparse array method
```

And for 20,000 indices drawn from 0-40,000 (a sparse sampling: only 1 out of 2 indices present on average), you get:

```
5.839  ; where()
0.0257 ; literal loop
0.1047 ; histogram() loop
0.0207 ; sparse array method
```

Yes, it's ugly, but the numbers speak for themselves. For those of you not too squeamish to look closely, you'll see that I've used a histogram of a histogram.

Does anyone begin to feel like the looping penalty in IDL is a bit much? In any case, it looks like I'm going to add the spr* functions to my rebin/reform/histogram/## power tool list. They seem to be quite fast.

JD

Subject: Re: Chunk Array Decimation
Posted by JD Smith on Fri, 04 Oct 2002 22:07:45 GMT
View Forum Message <> Reply to Message

One more very depressing timing to report. I compiled a C version of the literal accumulate loop, which consists entirely of:

```
for(i=0;i<=n_elts-1;i++) vec[inds[i]]+=data[i];
```

(omitting 10 lbs of DLM cruft). Here's a test run with 1,000,000 elements, each index repeated 20 times on average:

Literal Accumulate Loop: 1.2411
Reverse Indices Loop: 0.7217
Loop-Free with Sparse Arrays: 1.1401
FDDRIZZLE Loop: 0.7815
Dual Histogram Loop: 0.5490
Thinned WHERE Histogram Loop: 0.8422
Literal Accumulate: Compiled DLM : 0.0288

Yes, that's right, 20 times faster than the fastest pure IDL method. What's really amusing is to compare the compiled and uncompiled Literal Accumulate Loop, which uses precisely the same logic: 43 times faster, which is the approximate penalty you pay for loops in IDL vs. loops in C. This is optimized C (whereas IDL is not heavily optimized), but it only uses one processor. Threads are not enough to recover the tremendous difference between native compiled and IDL code.

I discover this disparity about once every year, and then conveniently forget about it, lest I should spend too much time writing function declarations ;).

JD

Subject: Re: Chunk Array Decimation
Posted by Jaco van Gorkom on Mon, 07 Oct 2002 15:51:21 GMT
View Forum Message <> Reply to Message

"JD Smith" <jdsmith@as.arizona.edu> wrote in message news:pan.2002.10.04.22.07.45.664757.24772@as.arizona.edu...
> ... Here's a test run with 1,000,000
> elements, each index repeated 20 times on average:
>
> Literal Accumulate Loop: 1.2411
> Reverse Indices Loop: 0.7217
> Loop-Free with Sparse Arrays: 1.1401
> FDDRIZZLE Loop: 0.7815
> Dual Histogram Loop: 0.5490
> Thinned WHERE Histogram Loop: 0.8422
> Literal Accumulate: Compiled DLM : 0.0288

JD, we mortals need a tutorial or two in order to even begin to understand that Sparse Array trick you pulled there. My humble contribution to this thread is to propose an additional and hopefully simpler (or, more intuitive)

algorithm: let us first sort the data array based on the index
array, and then use a cumulative total to do the chunking.
In pseudo-code:

```
  sortInds = SORT(inds)
  totData = TOTAL(data[sortInds], /CUMULATIVE)
  uniqInds = UNIQ(inds[sortInds])
  subTotData = [0., totData[uniqInds]]
  vec7 = subTotData[1:*] - subTotData[0:*-1]
```

leaving some minor issues like non-occurring indices
unresolved.

The above algorithm is by far not fast enough because of the
very slow SORT() operation. If this is replaced by a sorting
routine which is more optimised for the problem at hand, such
as, well, I'll let you guess;) , then things get much better:

```
  h = HISTOGRAM(inds, REVERSE_INDICES=ri)
  nh = N_ELEMENTS(h)
  sortData = data[ inds[ri[nh+1:*]] ]
  totSortData = [0., TOTAL(sortData, /CUMULATIVE)]
  vec8 = totSortData[ri[1:nh]-nh-1] - $
        totSortData[ri[0:nh-1]-nh-1]
```

On my machine this seems to be always slightly faster than the
double histogram loop.

Cheers,
  Jaco

P.S. Some timings:

For 1,000,000 elements, each index repeated 20 times on
average, on a single PIII 1GHz, W2K:

Literal Accumulate Loop: 1.2778

Reverse Indices Loop: 0.9794

Loop-Free with Sparse Arrays: 1.2589

FDDRIZZLE Loop: 0.9543

Dual Histogram Loop: 0.6329

Thinned WHERE Histogram Loop: 1.0506

Simple sorting plus cumulative total: 2.6347

Histogram plus cumulative total: 0.6119

And for each index repeated only once on average:

Literal Accumulate Loop: 1.3920

Reverse Indices Loop: 8.1998 (?)

Loop-Free with Sparse Arrays: 1.8647

FDDRIZZLE Loop: 1.7135

Dual Histogram Loop: 1.3129

Thinned WHERE Histogram Loop: 1.7996

Simple sorting plus cumulative total: 2.8701

Histogram plus cumulative total: 1.2488

Subject: Re: Chunk Array Decimation
Posted by JD Smith on Thu, 10 Oct 2002 00:18:45 GMT
View Forum Message <> Reply to Message

On Mon, 07 Oct 2002 08:51:21 -0700, Jaco van Gorkom wrote:

> "JD Smith" <jdsmith@as.arizona.edu> wrote in message
> news:pan.2002.10.04.22.07.45.664757.24772@as.arizona.edu...
>
> JD, we mortals need a tutorial or two in order to even begin to
> understand that Sparse Array trick you pulled there. My humble
> contribution to this thread is to propose an additional and hopefully
> simpler (or, more intuitive) algorithm: let us first sort the data array
> based on the index array, and then use a cumulative total to do the
> chunking.

Well, think of an array multiplication of a very large array on a very long data vector. Consider only the first row. If almost everything in the first column is 0., but a few choice 1.'s, that is the same as selecting those elements of the data vector and totaling them. If you really had to do all the null operations like:

 ...+0.*data[12]+0.*data[13]+...

then it would do you no good at all: all those useless
multiply-by-zeroes would waste far too much time to be efficient.
Fortunately, "sparse" arrays were invented for just this problem: you
specify only the non-zero elements, and, when multiplying using
sprsax, they alone are computed. If you adjust the array values, you
could obviously use this to do much more than totaling selected data
elements.

> The above algorithm is by far not fast enough because of the very slow
> SORT() operation. If this is replaced by a sorting routine which is more
> optimised for the problem at hand, such as, well, I'll let you guess;) ,
> then things get much better:
> h = HISTOGRAM(inds, REVERSE_INDICES=ri) nh = N_ELEMENTS(h) sortData =
> data[inds[ri[nh+1:*]]]
> totSortData = [0., TOTAL(sortData, /CUMULATIVE)] vec8 =
> totSortData[ri[1:nh]-nh-1] - $
> totSortData[ri[0:nh-1]-nh-1]

Aha, a very interesting and compact submission. I think there's one
error there. Should it not be data[ri[nh+1:*]] without the inds? I
translated as:

```
h = histogram(inds, REVERSE_INDICES=ri)
nh = n_elements(h)
tdata = [0.,total(data[ri[nh+1:*]],/CUMULATIVE)]
vec9 = tdata[ri[1:nh]-nh-1]-tdata[ri[0:nh-1]-nh-1]
```

The biggest problem with this method, though, is roundoff error, which
accumulates like mad in a cumulative total of any length. If you do
it in floating point, as you've written, I find unacceptably large
roundoff errors for as few as 500 individual indices. If I convert
the addition to DOUBLE, this mitigates (but does not eliminate) the
roundoff errors, but then it erases the slight time advantage this
method has over the prior champ (the dual histogram). Neither, of
course, come close to the compiled DLM :(.

Thanks for the example.

JD

Subject: Re: Chunk Array Decimation
Posted by JD Smith on Thu, 10 Oct 2002 00:19:32 GMT

View Forum Message <> Reply to Message

On Wed, 09 Oct 2002 17:18:45 -0700, JD Smith wrote:

> On Mon, 07 Oct 2002 08:51:21 -0700, Jaco van Gorkom wrote:
>
>> "JD Smith" <jdsmith@as.arizona.edu> wrote in message
>> news:pan.2002.10.04.22.07.45.664757.24772@as.arizona.edu...
>>
>> JD, we mortals need a tutorial or two in order to even begin to
>> understand that Sparse Array trick you pulled there. My humble
>> contribution to this thread is to propose an additional and hopefully
>> simpler (or, more intuitive) algorithm: let us first sort the data
>> array based on the index array, and then use a cumulative total to do
>> the chunking.
>
> Well, think of an array multiplication of a very large array on a very
> long data vector. Consider only the first row. If almost everything in
> the first column is 0., but a few choice 1.'s, that is the same as
> selecting those elements of the data vector and totaling them. If you
> really had to do all the null operations like:

Sorry, meant to say:

"if almost everything in the first *row* is 0.,..."

JD

Subject: Re: Chunk Array Decimation
Posted by Jaco van Gorkom on Thu, 10 Oct 2002 14:26:46 GMT
View Forum Message <> Reply to Message

"JD Smith" <jdsmith@as.arizona.edu> wrote in message
news:pan.2002.10.10.00.18.45.172062.5209@as.arizona.edu...
>
> Aha, a very interesting and compact submission. I think there's one
> error there. Should it not be data[ri[nh+1:*]] without the inds? I
> translated as:
>
> h = histogram(inds, REVERSE_INDICES=ri)
> nh = n_elements(h)
> tdata = [0.,total(data[ri[nh+1:*]],/CUMULATIVE)]
> vec9 = tdata[ri[1:nh]-nh-1]-tdata[ri[0:nh-1]-nh-1]

Ok, this is how I had coded it originally. I tested it on some ten-element sample vectors, got confused, and convinced myself that the extra inds had to be in. Reading the original problem again, I suppose you are right.

> The biggest problem with this method, though, is roundoff error, which
> accumulates like mad in a cumulative total of any length. If you do

> it in floating point, as you've written, I find unacceptably large
> roundoff errors for as few as 500 individual indices.

Oops. Ok, so we should use /DOUBLE, indeed. But a quick test of
TOTAL(REPLICATE(1., 30000000), /CUMULATIVE) reveals roundoff errors only
after roughly 17,000,000 elements on my machine. If you tested the routine
with a data vector data = FINDGEN(100000) then the average value for each data element would be 50,000 , so
after 500 indices the cumulative total is
already 25,000,000. Realistic data might well have a lower average value,
or even zero average for data spread around zero, so that the cumulative
total of 25,000,000 is only reached after many more elements or never.

And what exactly do you mean by 'accumulates like mad in a cumulative
total'? Suffers a cumulative total more from roundoff error than a single
total? That should not be, should it? The way I see it, not having any
computer science background, the error in a cumulative total would be the
(very small) error in each sum (accumulating many, many times), plus the
(much larger) roundoff errors that occur when we get to very high values.
So as long as we do not get to high values we should be ok, especially
since we take differences out of subelements of the total, which will not
be very far apart, probably.

Just my thoughts,
Jaco

P.S. The (floating-point) roundoff error in TOTAL seems to come in
integer steps, how strange! Maybe more later in a new thread...

Subject: Re: Chunk Array Decimation
Posted by JD Smith on Thu, 10 Oct 2002 18:51:39 GMT
View Forum Message <> Reply to Message

On Thu, 10 Oct 2002 07:26:46 -0700, Jaco van Gorkom wrote:

> "JD Smith" <jdsmith@as.arizona.edu> wrote in message
> news:pan.2002.10.10.00.18.45.172062.5209@as.arizona.edu...
>>
>> Aha, a very interesting and compact submission. I think there's one
>> error there. Should it not be data[ri[nh+1:*]] without the inds? I
>> translated as:
>>
>> h = histogram(inds, REVERSE_INDICES=ri) nh = n_elements(h) tdata =
>> [0.,total(data[ri[nh+1:*]],/CUMULATIVE)] vec9 =
>> tdata[ri[1:nh]-nh-1]-tdata[ri[0:nh-1]-nh-1]
>
> Ok, this is how I had coded it originally. I tested it on some

> ten-element sample vectors, got confused, and convinced myself that the
> extra inds had to be in. Reading the original problem again, I suppose
> you are right.
>
>> The biggest problem with this method, though, is roundoff error, which
>> accumulates like mad in a cumulative total of any length. If you do it
>> in floating point, as you've written, I find unacceptably large
>> roundoff errors for as few as 500 individual indices.
>
> Oops. Ok, so we should use /DOUBLE, indeed. But a quick test of
> TOTAL(REPLICATE(1., 30000000), /CUMULATIVE) reveals roundoff errors only
> after roughly 17,000,000 elements on my machine. If you tested the
> routine with a data vector data = FINDGEN(100000) then the average value
> for each data element would be 50,000 , so after 500 indices the
> cumulative total is already 25,000,000. Realistic data might well have a
> lower average value, or even zero average for data spread around zero,
> so that the cumulative total of 25,000,000 is only reached after many
> more elements or never.
>
> And what exactly do you mean by 'accumulates like mad in a cumulative
> total'? Suffers a cumulative total more from roundoff error than a
> single total? That should not be, should it? The way I see it, not
> having any computer science background, the error in a cumulative total
> would be the (very small) error in each sum (accumulating many, many
> times), plus the (much larger) roundoff errors that occur when we get to
> very high values. So as long as we do not get to high values we should
> be ok, especially since we take differences out of subelements of the
> total, which will not be very far apart, probably.

I guess I mean that to get the answer for a single element of the result vector, say the 100th element, you must do *many* more additions with this technique. You are throwing away most of the additions in the accumulation. Each addition has associated with it an intrinsic roundoff error. Compounding these errors many times is what leads to the "mad accumulation" of roundoff in the cumulative sum. This is more or less the same whether performing a regular or cumulative total. However, with all the other techniques in this thread, only the data in that bin are summed (a sum which still has its own roundoff error, but it's much smaller).

Essentially the issue is, your way performs the calculation of v+x+y+z as:

a+b+c+d+e+f+g+h+j+k+l+m+n+o+p+q+r+s+t+u+v+x+y+z -
a+b+c+d+e+f+g+h+j+k+l+m+n+o+p+q+r+s+t+u

An example:

The sum of the first n natural integers is n*(n+1)/2 (a result Gauss discovered as a young schoolboy, much to the consternation of his lazy

teacher, who had assigned her students a full morning to summing the first 100 integers). E.g. for n=1 million:

```
IDL> n=1000000LL & print,n*(n+1)/2
        500000500000

IDL> print,total(1.+findgen(n)),FORMAT='(D18.2)'
  499898744832.00
```

a difference of 101744640, or about .02%. Interestingly, the result is slightly different with accumulation:

```
IDL>  print,(total(1.+findgen(n),/CUMULATIVE))[n-1],FORMAT='(D18.2 )'
  499941376000.00
```

but it's roughly the same magnitude. Try it on your machine and see. Roundoff error is always roughly the same magnitude in terms of the mantissa. This can be a large number or small number, depending on the exponent. The relative roundoff is therefore a more interesting measure.

It's still a neat technique though.

JD

Subject: Re: Chunk Array Decimation
Posted by Keflavich on Mon, 17 Nov 2008 21:40:45 GMT
View Forum Message <> Reply to Message

Hi IDL group,
I've been using one of the drizzle algorithms posted at David Fanning's site: http://www.dfanning.com/code_tips/drizzling.html (and discussed on this group ~6 years ago) to combine timestream data into maps. It works like a charm, but I need to take it to the next level: cosmic ray removal.
Does anyone know how to implement this algorithm using a median stack of each pixel instead of simply adding / averaging? I'll admit I've been using this code rather blindly - my understanding of the use of all the various indexes is very much incomplete - but it still looks like this line:

```
vec6[vinds]=vec6[vinds]+total(data[ri1[vec_inds]],2)
```

means that you can only add to a running total; replacing 'total' in the above statement with 'median' would not work. So, any ideas?

Thanks,
Adam

Subject: Re: Chunk Array Decimation
Posted by Wout De Nolf on Tue, 18 Nov 2008 10:52:57 GMT
View Forum Message <> Reply to Message

On Mon, 17 Nov 2008 13:40:45 -0800 (PST), Keflavich
<keflavich@gmail.com> wrote:

> Does anyone know how to implement this algorithm using a median
> stack of each pixel instead of simply adding / averaging?

The 'dual histogram loop' approach to drizzling, which you use, loops
over the index-frequencies and not over the index-values like e.g. the
'single histogram' loop does. In the first case, intermediate results
(partial sums) are calculated each iteration while in the second case,
final results (total sums) are calculated each iteration.

As far as I know, there is no 'intermediate' median (i.e. the
equivalent to a partial sum). So maybe the single histogram loop is
what you need:

```
data=[1,2,3,4,5]
inds=[4,4,1,2,1]

mx=max(inds)
vec3=fltarr(mx+1)
h=histogram(inds,reverse_indices=ri,OMIN=om)
for j=0L,n_elements(h)-1 do if ri[j+1] gt ri[j] then $
    vec3[j+om]=median(data[ri[ri[j]:ri[j+1]-1]])
```

Does this help?

Subject: Re: Chunk Array Decimation
Posted by Keflavich on Tue, 18 Nov 2008 15:21:24 GMT
View Forum Message <> Reply to Message

On Nov 18, 3:52 am, Wox <s...@nomail.com> wrote:
> On Mon, 17 Nov 2008 13:40:45 -0800 (PST), Keflavich
>
> <keflav...@gmail.com> wrote:
>> Does anyone know how to implement this algorithm using a median
>> stack of each pixel instead of simply adding / averaging?
>
> The 'dual histogram loop' approach to drizzling, which you use, loops
> over the index-frequencies and not over the index-values like e.g. the
> 'single histogram' loop does. In the first case, intermediate results
> (partial sums) are calculated each iteration while in the second case,
> final results (total sums) are calculated each iteration.

>
> As far as I know, there is no 'intermediate' median (i.e. the
> equivalent to a partial sum). So maybe the single histogram loop is
> what you need:
>
> data=[1,2,3,4,5]
> inds=[4,4,1,2,1]
>
> mx=max(inds)
> vec3=fltarr(mx+1)
> h=histogram(inds,reverse_indices=ri,OMIN=om)
> for j=0L,n_elements(h)-1 do if ri[j+1] gt ri[j] then $
> vec3[j+om]=median(data[ri[ri[j]:ri[j+1]-1]])
>
> Does this help?

It does, thanks. That's remarkably simple; I think it's time for me to understand reverse indices on a deeper level.

Adam
