Subject: Re: 2d filters and large images (continued from ASSOC)
Posted by JD Smith on Thu, 27 May 2004 03:19:17 GMT
View Forum Message <> Reply to Message

On Wed, 26 May 2004 17:53:04 -0700, Peter Mason wrote:

> Jonathan Greenberg wrote:
>> So I probably should have asked a more general question (thank you for
>> the feedback on ASSOC, by the way). I was hoping to get some feedback
>> on how to apply 2-d filters of various sizes to a large 2-d image (too
>> big to load the entire file into memory) -- which ways would work the
>> best? Right now, if my filter is, say, 5 x 5, I grab 5 lines of the
>> image and run through the center pixels (row 3 in the subset) and apply
>> the filter, then grab the next line and create a new 5 line x number of
>> samples chunk.
>>
>> Are there better/quicker/easier ways of applying filter like this?
>
>
> Hi again Jonathan,
>
> Sounds nasty. Here are a few ideas. (I haven't been faced with a
> problem quite like this so these ideas will be somewhat abstract.)
>
> As you are grinding through the image sequentially, I would say that
> you'd get the best performance out of plain old READU (and WRITEU - you
> are writing a separate output, right?). A little better than ASSOC
> (which always does file-pointer manipulation) and maybe even better than
> proper memory mapping (which will page-fault when it feels the need and
> might end up bloating your working-set size).
>
> The immediate inclination is to maintain a 5-line buffer for your image
> data, shifting lines up by 1 and plastering the new line into the bottom
> of the buffer as you go through the image. But that's a lot of memory
> manipulation and it's certainly not going to go unnoticed. A faster
> approach is to maintain a cyclic image buffer. (No shifting. The
> new-line insertion index cycles round and round, and the current line
> index tracks behind it accordingly.) Here you do the line-shifting on
> the *filter* - a much quicker task as the filter is small.
>
> Another idea is to use proper memory mapping, mapping only 5 image lines
> at a time and bumping the mapping's offset by one line's worth to step
> through the image. The beauty of this is that there's no shifting
> involved at all, and *hopefully* the OS's caching will be smart enough
> to avoid repeated reads. The downside is the time taken by the OS to
> "bump the mapping's offset". With IDL's implementation I think that
> you have to close the old mapping entirely and open a new one in order
> to achieve this. I don't have a feel for the performance consequences

> here.

It seems to me you don't need to change the offset in the memory mapped file if you treat it as one large array, and keep track of the indices yourself. In fact, I've had success performing interpolations and other nearest neighbor operations on very large memory mapped arrays, just working with memory-sized chunks at once (e.g. 100MB). I would try this as your next step: it should certainly offer a speedup over the shift and read method, and is very simple to implement. The basic recipe is:

```
IDL> shmmap,'map_name',FILENAME=file,[25000L,25000L],/BYTE
IDL> arr=shmvar('map_name')
IDL> print,size(arr,/DIMENSIONS)
       25000       25000
IDL> print,arr[15000,15000]
  0
```

now you can access whatever size chunk of arr you like, e.g.:

```
arr[0,0]=convol(arr[0:4999,0:4999],$
 psf_gaussian(NPIXEL=5,FWHM=1.75,NDIMENSION=2,/NORMALIZE))
```

Don't try arr[*] though: IDL can't maintain such a large list of indices. And remember arr=blah doesn't work: it breaks the association of the variable `arr' with the memory mapped file.

Loop through all the relevant sized such chunks to apply the filter to the entire array. If you care about tile overlap, you'll have to extract a somewhat larger sized tile than you want (larger by the filter width), apply the filter, and set only the inner portion back to disk. I just tried this on my 1/2GB of data, and it took ~400s. The maximum speed that it could possibly have read and then written a total of 1GB on my (slow laptop) disk is about 100s, so this is only a factor of 4 overhead to do all the math, and perform all the non-serial I/O. I could probably tune the chunk size to get faster performance.

While the memory subsystem of your OS is at a disadvantage because it doesn't know beforehand the precise pattern of reads and writes you intend to make, it offers some advantages over READ, including direct I/O without intermediate copying, and no repeated system calls. And (depending on OS) it is backed by lots of research performed by clever people, and designed to keep unnecessary I/O to a minimum: it usually succeeds at this quite well. It also seems a more flexible technique than "rolling your own": just throw the pile of data in whatever format you have at the virtual memory device, and access away.

I'm sure I'm not alone in saying I'd love to see some test results on the relative efficiencies of these various methods for different sized memory-mapped arrays, if you ever get that far. Has anyone else had good or bad experiences with the new memory mapping routines?

Good luck,

JD
