Porting IDL *for* loops to C

Andrei T. Savici (saviciat@ornl.gov)

One unfortunate disadvantage of IDL is the low speed of any implementation of loops (for … do, repeat … until, while … do) over large amounts of iterations. Sometimes these operations are necessary, like in the case of drizzling (http://www-int.stsci.edu/~fruchter/dither/drizzle.html). A straightforward implementation of this problem in IDL is discussed for example in http://www.dfanning.com/code_tips/drizzling.html. This is exactly the algorithm used for rebinning neutron scattering data in DCS_Mslice. As mentioned in this webpage, "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. " The C implementation is in fact about 20 times faster than the best IDL implementation.

With the large amounts of data in the output of inelastic time-of-flight neutron scattering instruments at SNS, the need for fast rebinning implementations become immediately evident. The following describes the port of IDL for loops in dm_step_bin.pro, dm_stepgrid_bin.pro, and dm_volumegrid_bin.pro to the same algorithms implemented in C.

We have for example the following piece of code in dm_stepgrid_bin.pro:

```
 for i=0L,nxdat-1L do begin
  ix = value_locate(xlookup,xdat[i])
  iy = value_locate(ylookup,ydat[i])
  tmp_x[ix] = tmp_x[ix]+xdat[i] & num_x[ix] = num_x[ix]+1L
  tmp_y[iy] = tmp_y[iy]+ydat[i] & num_y[iy] = num_y[iy]+1L
  tmp_z[ix,iy] = tmp_z[ix,iy]+zdat[i] & num_z[ix,iy] = num_z[ix,iy]+1L
  if ok_eint then tmp_w[ix,iy] = tmp_w[ix,iy]+ewid[i]
  if ok_zerr then tmp_e[ix,iy] = tmp_e[ix,iy]+zerr[i]*zerr[i]
endfor
```

For the case where xlookup and ylookup are regular (no uniq_xval,uniq_yval are defined), we can write

```
ix = (xdat[i] – xmin)/xstep
```

and similarly for the other dimensions. The full implementation of the above loop in C looks like:

```
EXPORTED int rebin2d_nat_zew(IDL_LONG* ndat,float* x, float* y, double* z,
    double* zerr, double* ewid, IDL_LONG *nx, IDL_LONG *ny,
    float *xmin, float *xstep, float* ymin, float *ystep,
    IDL_LONG* num_x, float* tmp_x, IDL_LONG* num_y, float* tmp_y,
    IDL_LONG* num_z, float* tmp_z, float* tmp_e, float* tmp_w)
{
    long i, ix, iy;
    for (i = 0L; i < (*ndat); i++){
        ix = (long)((x[i] - (*xmin)) / (*xstep));
        iy = (long)((y[i] - (*ymin)) / (*ystep));
        if ((ix < 0) || (ix >= (*nx)) || (iy < 0) || (iy >= (*ny))) return 1;
        num_x[ix]++;
        num_y[iy]++;
        tmp_x[ix] += x[i];
        tmp_y[iy] += y[i];
        tmp_z[ix + iy * (*nx)] += z[i];
        num_z[ix + iy * (*nx)]++;
        tmp_e[ix + iy * (*nx)] += zerr[i]*zerr[i];
        tmp_w[ix + iy * (*nx)] += ewid[i];
        }
    return 0;
}
```

It is easier (and much faster) to write separate functions to be called from IDL for each individual combination of ok_eint and ok_zerr (total of 4 cases) than to test inside the C implementation.

IDL's CALL_EXTERNAL function is using a calling convention that passes an argument count (argc) and an array of arguments (argv) to a C function. We therefore need to have some wrapper routines in C that transform these parameters into the arguments of the functions we just wrote:

```
 EXPORTED int rebin2d_zew(int argc, void* argv[])
{
     return rebin2d_nat_zew((IDL_LONG *) argv[0],(float *) argv[1], (float *) argv[2],
     (double *) argv[3], (double *) argv[4], (double *) argv[5], (IDL_LONG *) argv[6],
     (IDL_LONG *) argv[7],(float *) argv[8], (float *) argv[9], (float *) argv[10],
     (float *) argv[11],(IDL_LONG *) argv[12], (float *) argv[13], (IDL_LONG *)
     argv[14], (float *) argv[15],(IDL_LONG *) argv[16], (float *) argv[17], (float *)
     argv[18], (float *) argv[19]);
}
```

To use these routines, one can employ MAKE_DLL procedure in IDL to create some dynamically loaded libraries (note that you need a C compiler that is compatible with your IDL). We used a naming convention for the dlls that contains the operating sytem version (Win32, linux, darwin), and the memory bits.

```
pro make_rebinnd,input_dir=input_dir,output_dir=output_dir
  if (N_elements(input_dir) ne 1) then $
     input_dir="dave\programs\modules\DCS\dcs_mslice\"
  source="rebinnd"
  export_rtns=["rebin1d_nat_yew","rebin1d_yew","rebin1d_nat_ye","rebin1d_ye",$
    "rebin1d_nat_yw","rebin1d_yw","rebin1d_nat_y","rebin1d_y", $
    "rebin2d_nat_zew","rebin2d_zew","rebin2d_nat_ze","rebin2d_ze",$
    "rebin2d_nat_zw","rebin2d_zw","rebin2d_nat_z","rebin2d_z",$
     "rebin3d_nat_iew","rebin3d_iew","rebin3d_nat_ie","rebin3d_ie",$
    "rebin3d_nat_iw","rebin3d_iw","rebin3d_nat_i","rebin3d_i"]
  if (N_elements(output_dir) ne 1) then $
     output_dir=file_dirname(ROUTINE_FILEPATH('dave'),/MARK_DIRECTORY)
  if strlen(output_dir) le 3 then output_dir=input_dir
  filename='rebinnd'+'_'+!version.os+'_'+strtrim(string(!VERSION.MEMORY_BITS),2)
  MAKE_DLL,source,filename, export_rtns, $
    INPUT_DIR=input_dir, DLL_PATH=shlib, output_dir=output_dir,$
    EXTRA_CFLAGS='-O2';,/verbose,/NOCLEANUP

end
```

We used gcc for the linux implementation and Visual C++ 2010 RC for the Windows implementation. The libraries were tested using IDL 7.1 in both 32 and 64 bits modes, on RHEL5 and Windows 7.

NOTE: there is very limited testing of parameters passed inside the libraries

In the IDL code of DCS_Mslice, the binning routines will search for the appropriate library, corresponding to the operating system and memory bits used. If it does not find the file, it will revert to the old IDL code (much slower). In case there are errors in the C implementation, you can revert to the old DCS_Mslice routine by just removing the rebind_xxxx_xx.dll or rebind_xxxx_xx.so files.