GDAL Warp API Tutorial

Overview

The GDAL Warp API (declared in **gdalwarper.h**) provides services for high performance image warping using application provided geometric transformation functions (GDALTransformerFunc), a variety of resampling kernels, and various masking options. Files much larger than can be held in memory can be warped.

This tutorial demonstrates how to implement an application using the Warp API. It assumes implementation in C++ as C and Python bindings are incomplete for the Warp API. It also assumes familiarity with the GDAL Data Model, and the general GDAL API.

Applications normally perform a warp by initializing a **GDALWarpOptions** structure with the options to be utilized, instantiating a **GDALWarpOperation** based on these options, and then invoking the **GDALWarpOperation::ChunkAndWarpImage()** method to perform the warp options internally using the **GDALWarpKernel** class.

A Simple Reprojection Case

First we will construct a relatively simple example for reprojecting an image, assuming an appropriate output file already exists, and with minimal error checking.

```
#include "gdalwarper.h"

int main()
{
    GDALDatasetH  hSrcDS, hDstDS;

    // Open input and output files.

    GDALAllRegister();

    hSrcDS = GDALOpen( "in.tif", GA_ReadOnly );
    hDstDS = GDALOpen( "out.tif", GA_Update );

    // Setup warp options.

    GDALWarpOptions *psWarpOptions = GDALCreateWarpOptions();

    psWarpOptions->hSrcDS = hSrcDS;
    psWarpOptions->hDstDS = hDstDS;

    psWarpOptions->nBandCount = 1;
    psWarpOptions->panSrcBands =
        (int *) CPLMalloc(sizeof(int) * psWarpOptions->nBandCount );
    psWarpOptions->panSrcBands[0] = 1;
    psWarpOptions->panDstBands =
        (int *) CPLMalloc(sizeof(int) * psWarpOptions->nBandCount );
    psWarpOptions->panDstBands[0] = 1;

    psWarpOptions->pfnProgress = GDALTermProgress;

    // Establish reprojection transformer.

    psWarpOptions->pTransformerArg =
        GDALCreateGenImgProjTransformer( hSrcDS,
                                         GDALGetProjectionRef(hSrcDS),
                                         hDstDS,
                                         GDALGetProjectionRef(hDstDS),
                                         FALSE, 0.0, 1 );
    psWarpOptions->pfnTransformer = GDALGenImgProjTransform;

    // Initialize and execute the warp operation.
    GDALWarpOperation oOperation;

    oOperation.Initialize( psWarpOptions );
    oOperation.ChunkAndWarpImage( 0, 0,
                                  GDALGetRasterXSize( hDstDS ),
                                  GDALGetRasterYSize( hDstDS ) );

    GDALDestroyGenImgProjTransformer( psWarpOptions->pTransformerArg );
    GDALDestroyWarpOptions( psWarpOptions );

    GDALClose( hDstDS );
    GDALClose( hSrcDS );

    return 0;
}
```

This example opens the existing input and output files (in.tif and out.tif). A **GDALWarpOptions** structure is allocated (**GDALCreateWarpOptions()** sets lots of sensible defaults for stuff, always use it for defaulting things), and the input and output file handles, and band lists are set. The panSrcBands and panDstBands lists are dynamically allocated here and will be free automatically by **GDALDestroyWarpOptions()**. The simple terminal output progress monitor (GDALTermProgress) is installed for reporting completion progress to the user.

GDALCreateGenImgProjTransformer() is used to initialize the reprojection transformation between the source and destination images. We assume that they already have reasonable bounds and coordinate systems set. Use of GCPs is disabled.

Once the options structure is ready, a **GDALWarpOperation** is instantiated using them, and the warp actually performed with **GDALWarpOperation::ChunkAndWarpImage()**. Then the transformer, warp options and datasets are cleaned up.

Normally error check would be needed after opening files, setting up the reprojection transformer (returns NULL on failure), and initializing the warp.

Other Warping Options

The **GDALWarpOptions** structures contains a number of items that can be set to control warping behavior. A few of particular interest are:

1. **GDALWarpOptions::dfWarpMemoryLimit** - Set the maximum amount of memory to be used by the **GDALWarpOperation** when selecting a size of image chunk to operate on. The value is in bytes, and the default is likely to be conservative (small). Increasing the chunk size can help substantially in some situations but care should be taken to ensure that this size, plus the GDAL cache size plus the working set of GDAL, your application and the operating system are less than the size of RAM or else excessive swapping is likely to interfere with performance. On a system with 256MB of RAM, a value of at least 64MB (roughly 64000000 bytes) is reasonable. Note that this value does **not** include the memory used by GDAL for low level block caching.

2. GDALWarpOpations::eResampleAlg - One of GRA_NearestNeighbour (the default, and fastest), GRA_Bilinear (2x2 bilinear resampling) or GRA_Cubic. The GRA_NearestNeighbour type should generally be used for thematic or color mapped images. The other resampling types may give better results for thematic images, especially when substantially changing resolution.

3. **GDALWarpOptions::padfSrcNoDataReal** - This array (one entry per band being processed) may be setup with a "nodata" value for each band if you wish to avoid having pixels of some

background value copied to the destination image.

4. **GDALWarpOptions::papszWarpOptions** - This is a string list of NAME=VALUE options passed to the warper. See the **GDALWarpOptions::papszWarpOptions** docs for all options. Supported values include:

   - INIT_DEST=[value] or INIT_DEST=NO_DATA: This option forces the destination image to be initialized to the indicated value (for all bands) or indicates that it should be initialized to the NO_DATA value in padfDstNoDataReal/padfDstNoDataImag. If this value isn't set the destination image will be read and the source warp overlaid on it.

   - WRITE_FLUSH=YES/NO: This option forces a flush to disk of data after each chunk is processed. In some cases this helps ensure a serial writing of the output data otherwise a block of data may be written to disk each time a block of data is read for the input buffer resulting in a lot of extra seeking around the disk, and reduced IO throughput. The default at this time is NO.

Creating the Output File

In the previous case an appropriate output file was already assumed to exist. Now we will go through a case where a new file with appropriate bounds in a new coordinate system is created. This operation doesn't relate specifically to the warp API. It is just using the transformation API.

```
#include "gdalwarper.h"
#include "ogr_spatialref.h"

...

    GDALDriverH hDriver;
    GDALDataType eDT;
    GDALDatasetH hDstDS;
    GDALDatasetH hSrcDS;

    // Open the source file.

    hSrcDS = GDALOpen( "in.tif", GA_ReadOnly );
    CPLAssert( hSrcDS != NULL );

    // Create output with same datatype as first input band.

    eDT = GDALGetRasterDataType(GDALGetRasterBand(hSrcDS,1));

    // Get output driver (GeoTIFF format)

    hDriver = GDALGetDriverByName( "GTiff" );
    CPLAssert( hDriver != NULL );

    // Get Source coordinate system.

    const char *pszSrcWKT, *pszDstWKT = NULL;

    pszSrcWKT = GDALGetProjectionRef( hSrcDS );
    CPLAssert( pszSrcWKT != NULL && strlen(pszSrcWKT) > 0 );

    // Setup output coordinate system that is UTM 11 WGS84.

    OGRSpatialReference oSRS;

    oSRS.SetUTM( 11, TRUE );
    oSRS.SetWellKnownGeogCS( "WGS84" );

    oSRS.exportToWkt( &pszDstWKT );

    // Create a transformer that maps from source pixel/line coordinates
    // to destination georeferenced coordinates (not destination
    // pixel line).  We do that by omitting the destination dataset
    // handle (setting it to NULL).
    void *hTransformArg;

    hTransformArg =
        GDALCreateGenImgProjTransformer( hSrcDS, pszSrcWKT, NULL, pszDstWKT,
                                         FALSE, 0, 1 );
    CPLAssert( hTransformArg != NULL );

    // Get approximate output georeferenced bounds and resolution for file.

    double adfDstGeoTransform[6];
    int nPixels=0, nLines=0;
    CPLErr eErr;

    eErr = GDALSuggestedWarpOutput( hSrcDS,
                                    GDALGenImgProjTransform, hTransformArg,
                                    adfDstGeoTransform, &nPixels, &nLines );
    CPLAssert( eErr == CE_None );

    GDALDestroyGenImgProjTransformer( hTransformArg );

    // Create the output file.

    hDstDS = GDALCreate( hDriver, "out.tif", nPixels, nLines,
                         GDALGetRasterCount(hSrcDS), eDT, NULL );

    CPLAssert( hDstDS != NULL );

    // Write out the projection definition.

    GDALSetProjection( hDstDS, pszDstWKT );
    GDALSetGeoTransform( hDstDS, adfDstGeoTransform );

    // Copy the color table, if required.

    GDALColorTableH hCT;

    hCT = GDALGetRasterColorTable( GDALGetRasterBand(hSrcDS,1) );
    if( hCT != NULL )
        GDALSetRasterColorTable( GDALGetRasterBand(hDstDS,1), hCT );

    ... proceed with warp as before ...
```

Some notes on this logic:

- We need to create the transformer to output coordinates such that the output of the transformer is georeferenced, not pixel line coordinates since we use the transformer to map pixels around the source image into destination georeferenced coordinates.
- The **GDALSuggestedWarpOutput()** function will return an adfDstGeoTransform, nPixels and nLines that describes an output image size and georeferenced extents that should hold all pixels from the source image. The resolution is intended to be comparable to the source, but the output pixels are always square regardless of the shape of input pixels.
- The warper requires an output file in a format that can be "randomly" written to. This generally limits things to uncompressed formats that have an implementation of the Create() method (as opposed to CreateCopy()). To warp to compressed formats, or CreateCopy() style formats it is necessary to produce a full temporary copy of the image in a better behaved format, and then CreateCopy() it to the desired final format.
- The Warp API copies only pixels. All color maps, georeferencing and other metadata must be copied to the destination by the application.

Performance Optimization

There are a number of things that can be done to optimize the performance of the warp API.

1. Increase the amount of memory available for the Warp API chunking so that larger chunks can be operated on at a time. This is the **GDALWarpOptions::dfWarpMemoryLimit** parameter. In theory the larger the chunk size operated on the more efficient the I/O strategy, and the more efficient the approximated transformation will be. However, the sum of the warp memory and the GDAL cache should be less than RAM size, likely around 2/3 of RAM size.

2. Increase the amount of memory for GDAL caching. This is especially important when working with very large input and output images that are scanline oriented. If all the input or output scanlines have to be re-read for each chunk they intersect performance may degrade greatly. Use **GDALSetCacheMax()** to control the amount of memory available for caching within GDAL.

3. Use an approximated transformation instead of exact reprojection for each pixel to be transformed. This code illustrates how an approximated transformation could be created based on a reprojection transformation, but with a given error threshold (dfErrorThreshold in output pixels).

```
hTransformArg =
    GDALCreateApproxTransformer( GDALGenImgProjTransform,
                                 hGenImgProjArg, dfErrorThreshold );
pfnTransformer = GDALApproxTransform;
```

4. When writing to a blank output file, use the INIT_DEST option in the **GDALWarpOptions::papszWarpOptions** to cause the output chunks to be initialized to a fixed value, instead of being read from the output. This can substantially reduce unnecessary IO work.

5. Use tiled input and output formats. Tiled formats allow a given chunk of source and destination imagery to be accessed without having to touch a great deal of extra image data. Large scanline oriented files can result in a great deal of wasted extra IO.

6. Process all bands in one call. This ensures the transformation calculations don't have to be performed for each band.

7. Use the **GDALWarpOperation::ChunkAndWarpMulti()** method instead of **GDALWarpOperation::ChunkAndWarpImage()**. It uses a separate thread for the IO and the actual image warp operation allowing more effective use of CPU and IO bandwidth. For this to work GDAL needs to have been built with multi-threading support (default on Win32, default on Unix since GDAL 1.8.0, for previous versions –with-threads was required in configure).

8. The resampling kernels vary is work required from nearest neighbour being least, then bilinear then cubic. Don't use a more complex resampling kernel than needed.

9. Avoid use of esoteric masking options so that special simplified logic case be used for common special cases. For instance, nearest neighbour resampling with no masking on 8bit data is highly optimized compared to the general case.

Other Masking Options

The **GDALWarpOptions** include a bunch of esoteric masking capabilities, for validity masks, and density masks on input and output. Some of these are not yet implemented and others are implemented but poorly tested. Other than per-band validity masks it is advised that these features be used with caution at this time.

Id