GDAL Virtual Format Tutorial

Contents

Introduction

The VRT driver is a format driver for GDAL that allows a virtual GDAL dataset to be composed from other GDAL datasets with repositioning, and algorithms potentially applied as well as various kinds of metadata altered or added. VRT descriptions of datasets can be saved in an XML format normally given the extension .vrt.

The VRT format can also describe **warping operations** and **pansharpening operations**.

An example of a simple .vrt file referring to a 512x512 dataset with one band loaded from utm.tif might look like this:

```
<VRTDataset rasterXSize="512" rasterYSize="512">
  <GeoTransform>440720.0,  60.0,  0.0,  3751320.0,  0.0,  -60.0</GeoTransform>
  <VRTRasterBand dataType="Byte" band="1">
    <ColorInterp>Gray</ColorInterp>
    <SimpleSource>
      <SourceFilename relativeToVRT="1">utm.tif</SourceFilename>
      <SourceBand>1</SourceBand>
      <SrcRect xOff="0" yOff="0" xSize="512" ySize="512"/>
      <DstRect xOff="0" yOff="0" xSize="512" ySize="512"/>
    </SimpleSource>
  </VRTRasterBand>
</VRTDataset>
```

Many aspects of the VRT file are a direct XML encoding of the GDAL Data Model which should be reviewed for understanding of the semantics of various elements.

VRT files can be produced by translating to VRT format. The resulting file can then be edited to modify mappings, add metadata or other purposes. VRT files can also be produced programmatically by various means.

This tutorial will cover the .vrt file format (suitable for users editing .vrt files), and how .vrt files may be created and manipulated programmatically for developers.

.vrt Format

A XML schema of the GDAL VRT format is available.

Virtual files stored on disk are kept in an XML format with the following elements.

VRTDataset: This is the root element for the whole GDAL dataset. It must have the attributes rasterXSize and rasterYSize describing the width and height of the dataset in pixels. It may have a subClass attributes with values VRTWarpedDataset (**Warped VRT**) or VRTPansharpenedDataset (**Pansharpened VRT**). It may have SRS, GeoTransform, GCPList, Metadata, MaskBand and VRTRasterBand subelements.

```
<VRTDataset rasterXSize="512" rasterYSize="512">
```

The allowed subelements for VRTDataset are :

- **SRS**: This element contains the spatial reference system (coordinate system) in OGC WKT format. Note that this must be appropriately escaped for XML, so items like quotes will have the ampersand escape sequences substituted. As as well WKT, and valid input to the SetFromUserInput() method (such as well known GEOGCS names, and PROJ.4 format) is also allowed in the SRS element.

```
<SRS>PROJCS[&quot;NAD27 / UTM zone
        11N&quot;,GEOGCS[&quot;NAD27&quot;,DATUM[&quot;North_American_Datum_1927&quot;,SPHEROID[&
        quot;Clarke
        1866&quot;,6378206.4,294.9786982139006,AUTHORITY[&quot;EPSG&quot;,&quot;7008&quot;]],AUTH
        ORITY[&quot;EPSG&quot;,&quot;6267&quot;]],PRIMEM[&quot;Greenwich&quot;,0],UNIT[&quot;degr
        ee&quot;,0.0174532925199433],AUTHORITY[&quot;EPSG&quot;,&quot;4267&quot;]],PROJECTION[&qu
        ot;Transverse_Mercator&quot;],PARAMETER[&quot;latitude_of_origin&quot;,0],PARAMETER[&quot
        ;central_meridian&quot;,-117],PARAMETER[&quot;scale_factor&quot;,0.9996],PARAMETER[&quot;
        false_easting&quot;,500000],PARAMETER[&quot;false_northing&quot;,0],UNIT[&quot;metre&quot
        ;,1,AUTHORITY[&quot;EPSG&quot;,&quot;9001&quot;]],AUTHORITY[&quot;EPSG&quot;,&quot;26711&
        quot;]]</SRS>
```

- **GeoTransform**: This element contains a six value affine geotransformation for the dataset, mapping between pixel/line coordinates and georeferenced coordinates.

```
<GeoTransform>440720.0,   60,   0.0,   3751320.0,   0.0,  -60.0</GeoTransform>
```

- **GCPList**: This element contains a list of Ground Control Points for the dataset, mapping between pixel/line coordinates and georeferenced coordinates. The Projection attribute should contain the SRS of the georeferenced coordinates in the same format as the SRS element.

```
<GCPList Projection="EPSG:4326">
  <GCP Id="1" Info="a" Pixel="0.5" Line="0.5" X="0.0" Y="0.0" Z="0.0" />
  <GCP Id="2" Info="b" Pixel="13.5" Line="23.5" X="1.0" Y="2.0" Z="0.0" />
</GCPList>
```

- **Metadata**: This element contains a list of metadata name/value pairs associated with the VRTDataset as a whole, or a VRTRasterBand. It has <MDI> (metadata item) subelements which have a "key" attribute and the value as the data of the element. The Metadata element can be repeated multiple times, in which case it must be accompanied with a "domain" attribute to indicate the name of the metadata domain.

```
<Metadata>
  <MDI key="md_key">Metadata value</MDI>
</Metadata>
```

- **MaskBand**: (GDAL >= 1.8.0) This element represents a mask band that is shared between all bands on the dataset (see GMF_PER_DATASET in RFC 15). It must contain a single VRTRasterBand

child element, that is the description of the mask band itself.

```
<MaskBand>
  <VRTRasterBand dataType="Byte">
    <SimpleSource>
      <SourceFilename relativeToVRT="1">utm.tif</SourceFilename>
      <SourceBand>mask,1</SourceBand>
      <SrcRect xOff="0" yOff="0" xSize="512" ySize="512"/>
      <DstRect xOff="0" yOff="0" xSize="512" ySize="512"/>
    </SimpleSource>
  </VRTRasterBand>
</MaskBand>
```

- **VRTRasterBand**: This represents one band of a dataset. It will have a dataType attribute with the type of the pixel data associated with this band (use names Byte, UInt16, Int16, UInt32, Int32, Float32, Float64, CInt16, CInt32, CFloat32 or CFloat64) and the band this element represents (1 based). This element may have Metadata, ColorInterp, NoDataValue, HideNoDataValue, ColorTable, **GDALRasterAttributeTable**, Description and MaskBand subelements as well as the various kinds of source elements such as SimpleSource, ComplexSource, etc. A raster band may have many "sources" indicating where the actual raster data should be fetched from, and how it should be mapped into the raster bands pixel space.

  The allowed subelements for VRTRasterBand are :

  - **ColorInterp**: The data of this element should be the name of a color interpretation type. One of Gray, Palette, Red, Green, Blue, Alpha, Hue, Saturation, Lightness, Cyan, Magenta, Yellow, Black, or Unknown.

    ```
    <ColorInterp>Gray</ColorInterp>:
    ```

  - **NoDataValue**: If this element exists a raster band has a nodata value associated with, of the value given as data in the element.

    ```
    <NoDataValue>-100.0</NoDataValue>
    ```

  - **HideNoDataValue**: If this value is 1, the nodata value will not be reported. Essentially, the caller will not be aware of a nodata pixel when it reads one. Any datasets copied/translated from this will not have a nodata value. This is useful when you want to specify a fixed background value for the dataset. The background will be the value specified by the NoDataValue element.

    Default value is 0 when this element is absent.

    ```
    <HideNoDataValue>1</HideNoDataValue>
    ```

  - **ColorTable**: This element is parent to a set of Entry elements defining the entries in a color table. Currently only RGBA color tables are supported with c1 being red, c2 being green, c3 being blue and c4 being alpha. The entries are ordered and will be assumed to start from color table entry 0.

    ```
    <ColorTable>
      <Entry c1="0" c2="0" c3="0" c4="255"/>
      <Entry c1="145" c2="78" c3="224" c4="255"/>
    </ColorTable>
    ```

- **GDALRasterAttributeTable**: (GDAL >=2.3) This element is parent to a set of FieldDefn elements defining the columns of a raster attribute table, followed by a set of Row element defining the values of the columns of each row.

```
<GDALRasterAttributeTable>
  <FieldDefn index="0">
    <Name>Value</Name>
    <Type>0</Type>
    <Usage>0</Usage>
  </FieldDefn>
  <FieldDefn index="1">
    <Name>Red</Name>
    <Type>0</Type>
    <Usage>6</Usage>
  </FieldDefn>
  <FieldDefn index="2">
    <Name>Green</Name>
    <Type>0</Type>
    <Usage>7</Usage>
  </FieldDefn>
  <FieldDefn index="3">
    <Name>Blue</Name>
    <Type>0</Type>
    <Usage>8</Usage>
  </FieldDefn>
  <Row index="0">
    <F>-500</F>
    <F>127</F>
    <F>40</F>
    <F>65</F>
  </Row>
  <Row index="1">
    <F>-400</F>
    <F>154</F>
    <F>168</F>
    <F>118</F>
  </Row>
</GDALRasterAttributeTable>
```

- **Description**: This element contains the optional description of a raster band as its text value.

```
<Description>Crop Classification Layer</Description>
```

- **UnitType**: This optional element contains the vertical units for elevation band data. One of "m" for meters or "ft" for feet. Default assumption is meters.

```
<UnitType>ft</UnitType>
```

- **Offset**: This optional element contains the offset that should be applied when computing "real" pixel values from scaled pixel values on a raster band. The default is 0.0.

```
<Offset>0.0</Offset>
```

- **Scale**: This optional element contains the scale that should be applied when computing "real" pixel values from scaled pixel values on a raster band. The default is 1.0.

```
<Scale>0.0</Scale>
```

- **Overview**: This optional element describes one overview level for the band. It should have a child SourceFilename and SourceBand element. The SourceFilename may have a relativeToVRT boolean attribute. Multiple elements may be used to describe multiple overviews.

```
<Overview>
```

```
  <SourceFilename relativeToVRT="1">yellowstone_2.1.ntf.r2</SourceFilename>
  <SourceBand>1</SourceBand>
</Overview>
```

- **CategoryNames**: This optional element contains a list of Category subelements with the names of the categories for classified raster band.

```
<CategoryNames>
  <Category>Missing</Category>
  <Category>Non-Crop</Category>
  <Category>Wheat</Category>
  <Category>Corn</Category>
  <Category>Soybeans</Category>
</CategoryNames>
```

- **SimpleSource**: The SimpleSource indicates that raster data should be read from a separate dataset, indicating the dataset, and band to be read from, and how the data should map into this bands raster space. The SimpleSource may have the SourceFilename, SourceBand, SrcRect, and DstRect subelements. The SrcRect element will indicate what rectangle on the indicated source file should be read, and the DstRect element indicates how that rectangle of source data should be mapped into the VRTRasterBands space.

  The relativeToVRT attribute on the SourceFilename indicates whether the filename should be interpreted as relative to the .vrt file (value is 1) or not relative to the .vrt file (value is 0). The default is 0.

  The shared attribute, added in GDAL 2.0.0, on the SourceFilename indicates whether the dataset should be shared (value is 1) or not (value is 0). The default is 1. If several VRT datasets referring to the same underlying sources are used in a multithreaded context, shared should be set to 0. Alternatively, the VRT_SHARED_SOURCE configuration option can be set to 0 to force non-shared mode.

  Some characteristics of the source band can be specified in the optional SourceProperties tag to enable the VRT driver to differ the opening of the source dataset until it really needs to read data from it. This is particularly useful when building VRTs with a big number of source datasets. The needed parameters are the raster dimensions, the size of the blocks and the data type. If the SourceProperties tag is not present, the source dataset will be opened at the same time as the VRT itself.

  Starting with GDAL 1.8.0, the content of the SourceBand subelement can refer to a mask band. For example mask,1 means the mask band of the first band of the source.

```
<SimpleSource>
  <SourceFilename relativeToVRT="1">utm.tif</SourceFilename>
  <SourceBand>1</SourceBand>
  <SourceProperties RasterXSize="512" RasterYSize="512" DataType="Byte" BlockXSize="128"
        BlockYSize="128"/>
  <SrcRect xOff="0" yOff="0" xSize="512" ySize="512"/>
  <DstRect xOff="0" yOff="0" xSize="512" ySize="512"/>
</SimpleSource>
```

  Starting with GDAL 2.0, a OpenOptions subelement can be added to specify the open options to apply when opening the source dataset. It has <OOI> (open option item) subelements which have a "key" attribute and the value as the data of the element.

```
<SimpleSource>
  <SourceFilename relativeToVRT="1">utm.tif</SourceFilename>
  <OpenOptions>
```

```
      <OOI key="OVERVIEW_LEVEL">0</OOI>
  </OpenOptions>
  <SourceBand>1</SourceBand>
  <SourceProperties RasterXSize="256" RasterYSize="256" DataType="Byte" BlockXSize="128"
        BlockYSize="128"/>
  <SrcRect xOff="0" yOff="0" xSize="256" ySize="256"/>
  <DstRect xOff="0" yOff="0" xSize="256" ySize="256"/>
</SimpleSource>
```

Starting with GDAL 2.0, a resampling attribute can be specified on a SimpleSource or ComplexSource element to specified the resampling algorithm used when the size of the destination rectangle is not the same as the size of the source rectangle. The values allowed for that attribute are : nearest,bilinear,cubic, cubicspline,lanczos,average,mode.

```
<SimpleSource resampling="cubic">
  <SourceFilename relativeToVRT="1">utm.tif</SourceFilename>
  <SourceBand>1</SourceBand>
  <SourceProperties RasterXSize="256" RasterYSize="256" DataType="Byte" BlockXSize="128"
        BlockYSize="128"/>
  <SrcRect xOff="0" yOff="0" xSize="256" ySize="256"/>
  <DstRect xOff="0" yOff="0" xSize="128" ySize="128"/>
</SimpleSource>
```

- **AveragedSource**: The AveragedSource is derived from the SimpleSource and shares the same properties except that it uses an averaging resampling instead of a nearest neighbour algorithm as in SimpleSource, when the size of the destination rectangle is not the same as the size of the source rectangle. Note: starting with GDAL 2.0, a more general mechanism to specify resampling algorithms can be used. See above paragraph about the 'resampling' attribute.

- **ComplexSource**: The ComplexSource is derived from the SimpleSource (so it shares the SourceFilename, SourceBand, SrcRect and DestRect elements), but it provides support to rescale and offset the range of the source values. Certain regions of the source can be masked by specifying the NODATA value.

  Starting with GDAL 1.11, alternatively to linear scaling, non-linear scaling using a power function can be used by specifying the Exponent, SrcMin, SrcMax, DstMin and DstMax elements. If SrcMin and SrcMax are not specified, they are computed from the source minimum and maximum value (which might require analyzing the whole source dataset). Exponent must be positive. (Those 5 values can be set with the -exponent and -scale options of gdal_translate.)

  The ComplexSource supports adding a custom lookup table to transform the source values to the destination. The LUT can be specified using the following form:

  ```
  <LUT>[src value 1]:[dest value 1],[src value 2]:[dest value 2],...</LUT>
  ```

  The intermediary values are calculated using a linear interpolation between the bounding destination values of the corresponding range.

  The ComplexSource supports fetching a color component from a source raster band that has a color table. The ColorTableComponent value is the index of the color component to extract : 1 for the red band, 2 for the green band, 3 for the blue band or 4 for the alpha band.

  When transforming the source values the operations are executed in the following order:

1. Nodata masking
2. Color table expansion
3. For linear scaling, applying the scale ratio, then scale offset
4. For non-linear scaling, apply (DstMax-DstMin) * pow((SrcValue-SrcMin) / (SrcMax-SrcMin), Exponent) + DstMin
5. Table lookup

```
<ComplexSource>
  <SourceFilename relativeToVRT="1">utm.tif</SourceFilename>
  <SourceBand>1</SourceBand>
  <ScaleOffset>0</ScaleOffset>
  <ScaleRatio>1</ScaleRatio>
  <ColorTableComponent>1</ColorTableComponent>
  <LUT>0:0,2345.12:64,56789.5:128,2364753.02:255</LUT>
  <NODATA>0</NODATA>
  <SrcRect xOff="0" yOff="0" xSize="512" ySize="512"/>
  <DstRect xOff="0" yOff="0" xSize="512" ySize="512"/>
</ComplexSource>
```

Non-linear scaling:

```
<ComplexSource>
  <SourceFilename relativeToVRT="1">16bit.tif</SourceFilename>
  <SourceBand>1</SourceBand>
  <Exponent>0.75</Exponent>
  <SrcMin>0</SrcMin>
  <SrcMax>65535</SrcMax>
  <DstMin>0</DstMin>
  <DstMax>255</DstMax>
  <SrcRect xOff="0" yOff="0" xSize="512" ySize="512"/>
  <DstRect xOff="0" yOff="0" xSize="512" ySize="512"/>
</ComplexSource>
```

- **KernelFilteredSource**: This is a pixel source derived from the Simple Source (so it shares the SourceFilename, SourceBand, SrcRect and DestRect elements, but it also passes the data through a simple filtering kernel specified with the Kernel element. The Kernel element should have two child elements, Size and Coefs and optionally the boolean attribute normalized (defaults to false=0). The size must always be an odd number, and the Coefs must have Size * Size entries separated by spaces. For now kernel is not applied to sub-sampled or over-sampled data.

```
<KernelFilteredSource>
  <SourceFilename>/debian/home/warmerda/openev/utm.tif</SourceFilename>
  <SourceBand>1</SourceBand>
  <Kernel normalized="1">
    <Size>3</Size>
    <Coefs>0.11111111 0.11111111 0.11111111 0.11111111 0.11111111 0.11111111 0.11111111
        0.11111111 0.11111111</Coefs>
  </Kernel>
</KernelFilteredSource>
```

Starting with GDAL 2.3, a separable kernel may also be used. In this case the number of Coefs entries should correspond to the Size. The Coefs specify a one-dimensional kernel which is applied along each axis in succession, resulting in far quicker execution. Many common image-processing filters are separable. For example, a Gaussian blur:

```
<KernelFilteredSource>
  <SourceFilename>/debian/home/warmerda/openev/utm.tif</SourceFilename>
  <SourceBand>1</SourceBand>
  <Kernel normalized="1">
    <Size>13</Size>
    <Coefs>0.01111 0.04394 0.13534 0.32465 0.60653 0.8825 1.0 0.8825 0.60653 0.32465
        0.13534 0.04394 0.01111</Coefs>
  </Kernel>
</KernelFilteredSource>
```

- **MaskBand**: (GDAL >= 1.8.0) This element represents a mask band that is specific to the VRTRasterBand it contains. It must contain a single VRTRasterBand child element, that is the description of the mask band itself.

Overviews

GDAL can make efficient use of overviews available in the sources that compose the bands when dealing with RasterIO() requests that involve downsampling. But in the general case, the VRT bands themselves will not expose overviews.

Except if (from top priority to lesser priority) :

1. The **Overview** element is present in the VRTRasterBand element. See above.
2. or external .vrt.ovr overviews are built
3. (starting with GDAL 2.1) if the VRTRasterBand are made of a single SimpleSource or ComplexSource that has overviews. Those "virtual" overviews will be hidden by external .vrt.ovr overviews that might be built later.

.vrt Descriptions for Raw Files

So far we have described how to derive new virtual datasets from existing files supports by GDAL. However, it is also common to need to utilize raw binary raster files for which the regular layout of the data is known but for which no format specific driver exists. This can be accomplished by writing a .vrt file describing the raw file.

For example, the following .vrt describes a raw raster file containing floating point complex pixels in a file called *l2p3hhsso.img*. The image data starts from the first byte (ImageOffset=0). The byte offset between pixels is 8 (PixelOffset=8), the size of a CFloat32. The byte offset from the start of one line to the start of the next is 9376 bytes (LineOffset=9376) which is the width (1172) times the size of a pixel (8).

```
<VRTDataset rasterXSize="1172" rasterYSize="1864">
  <VRTRasterBand dataType="CFloat32" band="1" subClass="VRTRawRasterBand">
    <SourceFilename relativetoVRT="1">l2p3hhsso.img</SourceFilename>
    <ImageOffset>0</ImageOffset>
    <PixelOffset>8</PixelOffset>
    <LineOffset>9376</LineOffset>
    <ByteOrder>MSB</ByteOrder>
  </VRTRasterBand>
</VRTDataset>
```

Some things to note are that the VRTRasterBand has a subClass specifier of "VRTRawRasterBand". Also, the VRTRawRasterBand contains a number of previously unseen elements but no "source" information. VRTRawRasterBands may never have sources (i.e. SimpleSource), but should contain the following elements in addition to all the normal "metadata" elements previously described which are still supported.

- **SourceFilename**: The name of the raw file containing the data for this band. The relativeToVRT attribute can be used to indicate if the SourceFilename is relative to the .vrt file (1) or not (0).
- **ImageOffset**: The offset in bytes to the beginning of the first pixel of data of this image band. Defaults to zero.

- **PixelOffset**: The offset in bytes from the beginning of one pixel and the next on the same line. In packed single band data this will be the size of the **dataType** in bytes.
- **LineOffset**: The offset in bytes from the beginning of one scanline of data and the next scanline of data. In packed single band data this will be PixelOffset * rasterXSize.
- **ByteOrder**: Defines the byte order of the data on disk. Either LSB (Least Significant Byte first) such as the natural byte order on Intel x86 systems or MSB (Most Significant Byte first) such as the natural byte order on Motorola or Sparc systems. Defaults to being the local machine order.

A few other notes:

- The image data on disk is assumed to be of the same data type as the band **dataType** of the VRTRawRasterBand.
- All the non-source attributes of the VRTRasterBand are supported, including color tables, metadata, nodata values, and color interpretation.
- The VRTRawRasterBand supports in place update of the raster, whereas the source based VRTRasterBand is always read-only.
- The OpenEV tool includes a File menu option to input parameters describing a raw raster file in a GUI and create the corresponding .vrt file.
- Multiple bands in the one .vrt file can come from the same raw file. Just ensure that the ImageOffset, PixelOffset, and LineOffset definition for each band is appropriate for the pixels of that particular band.

Another example, in this case a 400x300 RGB pixel interleaved image.

```
<VRTDataset rasterXSize="400" rasterYSize="300">
  <VRTRasterBand dataType="Byte" band="1" subClass="VRTRawRasterBand">
    <ColorInterp>Red</ColorInterp>
    <SourceFilename relativetoVRT="1">rgb.raw</SourceFilename>
    <ImageOffset>0</ImageOffset>
    <PixelOffset>3</PixelOffset>
    <LineOffset>1200</LineOffset>
  </VRTRasterBand>
  <VRTRasterBand dataType="Byte" band="2" subClass="VRTRawRasterBand">
    <ColorInterp>Green</ColorInterp>
    <SourceFilename relativetoVRT="1">rgb.raw</SourceFilename>
    <ImageOffset>1</ImageOffset>
    <PixelOffset>3</PixelOffset>
    <LineOffset>1200</LineOffset>
  </VRTRasterBand>
  <VRTRasterBand dataType="Byte" band="3" subClass="VRTRawRasterBand">
    <ColorInterp>Blue</ColorInterp>
    <SourceFilename relativetoVRT="1">rgb.raw</SourceFilename>
    <ImageOffset>2</ImageOffset>
    <PixelOffset>3</PixelOffset>
    <LineOffset>1200</LineOffset>
  </VRTRasterBand>
</VRTDataset>
```

Programmatic Creation of VRT Datasets

The VRT driver supports several methods of creating VRT datasets. As of GDAL 1.2.0 the **vrtdataset.h** include file should be installed with the core GDAL include files, allowing direct access to the VRT classes. However, even without that most capabilities remain available through standard GDAL interfaces.

To create a VRT dataset that is a clone of an existing dataset use the CreateCopy() method. For example to clone utm.tif into a wrk.vrt file in C++ the following could be used:

```
GDALDriver *poDriver = (GDALDriver *) GDALGetDriverByName( "VRT" );
GDALDataset *poSrcDS, *poVRTDS;

poSrcDS = (GDALDataset *) GDALOpenShared( "utm.tif", GA_ReadOnly );

poVRTDS = poDriver->CreateCopy( "wrk.vrt", poSrcDS, FALSE, NULL, NULL, NULL );

GDALClose((GDALDatasetH) poVRTDS);
GDALClose((GDALDatasetH) poSrcDS);
```

Note the use of **GDALOpenShared()** when opening the source dataset. It is advised to use **GDALOpenShared()** in this situation so that you are able to release the explicit reference to it before closing the VRT dataset itself. In other words, in the previous example, you could also invert the 2 last lines, whereas if you open the source dataset with **GDALOpen()**, you'd need to close the VRT dataset before closing the source dataset.

To create a virtual copy of a dataset with some attributes added or changed such as metadata or coordinate system that are often hard to change on other formats, you might do the following. In this case, the virtual dataset is created "in memory" only by virtual of creating it with an empty filename, and then used as a modified source to pass to a CreateCopy() written out in TIFF format.

```
poVRTDS = poDriver->CreateCopy( "", poSrcDS, FALSE, NULL, NULL, NULL );

poVRTDS->SetMetadataItem( "SourceAgency", "United States Geological Survey");
poVRTDS->SetMetadataItem( "SourceDate", "July 21, 2003" );

poVRTDS->GetRasterBand( 1 )->SetNoDataValue( -999.0 );

GDALDriver *poTIFFDriver = (GDALDriver *) GDALGetDriverByName( "GTiff" );
GDALDataset *poTiffDS;

poTiffDS = poTIFFDriver->CreateCopy( "wrk.tif", poVRTDS, FALSE, NULL, NULL, NULL );

GDALClose((GDALDatasetH) poTiffDS);
```

In the above example the nodata value is set as -999. You can set the HideNoDataValue element in the VRT dataset's band using SetMetadataItem() on that band.

```
poVRTDS->GetRasterBand( 1 )->SetMetadataItem( "HideNoDataValue" , "1" );
```

In this example a virtual dataset is created with the Create() method, and adding bands and sources programmatically, but still via the "generic" API. A special attribute of VRT datasets is that sources can be added to the VRTRasterBand (but not to VRTRawRasterBand) by passing the XML describing the source into SetMetadata() on the special domain target "new_vrt_sources". The domain target "vrt_sources" may also be used, in which case any existing sources will be discarded before adding the new ones. In this example we construct a simple averaging filter source instead of using the simple source.

```
  // construct XML for simple 3x3 average filter kernel source.
  const char *pszFilterSourceXML  =
"<KernelFilteredSource>"
"  <SourceFilename>utm.tif</SourceFilename><SourceBand>1</SourceBand>"
"  <Kernel>"
"    <Size>3</Size>"
"    <Coefs>0.111 0.111 0.111 0.111 0.111 0.111 0.111 0.111 0.111</Coefs>"
"  </Kernel>"
"</KernelFilteredSource>";

  // Create the virtual dataset.
  poVRTDS = poDriver->Create( "", 512, 512, 1, GDT_Byte, NULL );
```

```
poVRTDS->GetRasterBand(1)->SetMetadataItem("source_0",pszFilterSourceXML,
                                           "new_vrt_sources");
```

A more general form of this that will produce a 3x3 average filtered clone of any input datasource might look like the following. In this case we deliberately set the filtered datasource as in the "vrt_sources" domain to override the SimpleSource created by the CreateCopy() method. The fact that we used CreateCopy() ensures that all the other metadata, georeferencing and so forth is preserved from the source dataset ... the only thing we are changing is the data source for each band.

```
int   nBand;
GDALDriver *poDriver = (GDALDriver *) GDALGetDriverByName( "VRT" );
GDALDataset *poSrcDS, *poVRTDS;

poSrcDS = (GDALDataset *) GDALOpenShared( pszSourceFilename, GA_ReadOnly );

poVRTDS = poDriver->CreateCopy( "", poSrcDS, FALSE, NULL, NULL, NULL );

for( nBand = 1; nBand <= poVRTDS->GetRasterCount(); nBand++ )
{
    char szFilterSourceXML[10000];

    GDALRasterBand *poBand = poVRTDS->GetRasterBand( nBand );

    sprintf( szFilterSourceXML,
      "<KernelFilteredSource>"
      "  <SourceFilename>%s</SourceFilename><SourceBand>%d</SourceBand>"
      "  <Kernel>"
      "    <Size>3</Size>"
      "    <Coefs>0.111 0.111 0.111 0.111 0.111 0.111 0.111 0.111 0.111</Coefs>"
      "  </Kernel>"
      "</KernelFilteredSource>",
      pszSourceFilename, nBand );

    poBand->SetMetadataItem( "source_0", szFilterSourceXML, "vrt_sources" );
}
```

The VRTDataset class is one of the few dataset implementations that supports the AddBand() method. The options passed to the AddBand() method can be used to control the type of the band created (VRTRasterBand, VRTRawRasterBand, VRTDerivedRasterBand), and in the case of the VRTRawRasterBand to set its various parameters. For standard VRTRasterBand, sources should be specified with the above SetMetadata() / SetMetadataItem() examples.

```
GDALDriver *poDriver = (GDALDriver *) GDALGetDriverByName( "VRT" );
GDALDataset *poVRTDS;

poVRTDS = poDriver->Create( "out.vrt", 512, 512, 0, GDT_Byte, NULL );
char** papszOptions = NULL;
papszOptions = CSLAddNameValue(papszOptions, "subclass", "VRTRawRasterBand"); // if not specified,
        default to VRTRasterBand
papszOptions = CSLAddNameValue(papszOptions, "SourceFilename", "src.tif"); // mandatory
papszOptions = CSLAddNameValue(papszOptions, "ImageOffset", "156"); // optional. default = 0
papszOptions = CSLAddNameValue(papszOptions, "PixelOffset", "2"); // optional. default = size of band
        type
papszOptions = CSLAddNameValue(papszOptions, "LineOffset", "1024"); // optional. default = size of band
        type * width
papszOptions = CSLAddNameValue(papszOptions, "ByteOrder", "LSB"); // optional. default = machine order
papszOptions = CSLAddNameValue(papszOptions, "relativeToVRT", "true"); // optional. default = false
poVRTDS->AddBand(GDT_Byte, papszOptions);
CSLDestroy(papszOptions);

delete poVRTDS;
```

Using Derived Bands (with pixel functions in C/C++)

A specialized type of band is a 'derived' band which derives its pixel information from its source bands. With this type of band you must also specify a pixel function, which has the responsibility of generating

the output raster. Pixel functions are created by an application and then registered with GDAL using a unique key.

Using derived bands you can create VRT datasets that manipulate bands on the fly without having to create new band files on disk. For example, you might want to generate a band using four source bands from a nine band input dataset (x0, x3, x4, and x8):

```
band_value = sqrt((x3*x3+x4*x4)/(x0*x8));
```

You could write the pixel function to compute this value and then register it with GDAL with the name "MyFirstFunction". Then, the following VRT XML could be used to display this derived band:

```
<VRTDataset rasterXSize="1000" rasterYSize="1000">
  <VRTRasterBand dataType="Float32" band="1" subClass="VRTDerivedRasterBand">
    <Description>Magnitude</Description>
    <PixelFunctionType>MyFirstFunction</PixelFunctionType>
    <SimpleSource>
      <SourceFilename relativeToVRT="1">nine_band.dat</SourceFilename>
      <SourceBand>1</SourceBand>
      <SrcRect xOff="0" yOff="0" xSize="1000" ySize="1000"/>
      <DstRect xOff="0" yOff="0" xSize="1000" ySize="1000"/>
    </SimpleSource>
    <SimpleSource>
      <SourceFilename relativeToVRT="1">nine_band.dat</SourceFilename>
      <SourceBand>4</SourceBand>
      <SrcRect xOff="0" yOff="0" xSize="1000" ySize="1000"/>
      <DstRect xOff="0" yOff="0" xSize="1000" ySize="1000"/>
    </SimpleSource>
    <SimpleSource>
      <SourceFilename relativeToVRT="1">nine_band.dat</SourceFilename>
      <SourceBand>5</SourceBand>
      <SrcRect xOff="0" yOff="0" xSize="1000" ySize="1000"/>
      <DstRect xOff="0" yOff="0" xSize="1000" ySize="1000"/>
    </SimpleSource>
    <SimpleSource>
      <SourceFilename relativeToVRT="1">nine_band.dat</SourceFilename>
      <SourceBand>9</SourceBand>
      <SrcRect xOff="0" yOff="0" xSize="1000" ySize="1000"/>
      <DstRect xOff="0" yOff="0" xSize="1000" ySize="1000"/>
    </SimpleSource>
  </VRTRasterBand>
</VRTDataset>
```

In addition to the subclass specification (VRTDerivedRasterBand) and the PixelFunctionType value, there is another new parameter that can come in handy: SourceTransferType. Typically the source rasters are obtained using the data type of the derived band. There might be times, however, when you want the pixel function to have access to higher resolution source data than the data type being generated. For example, you might have a derived band of type "Float", which takes a single source of type "CFloat32" or "CFloat64", and returns the imaginary portion. To accomplish this, set the SourceTransferType to "CFloat64". Otherwise the source would be converted to "Float" prior to calling the pixel function, and the imaginary portion would be lost.

```
<VRTDataset rasterXSize="1000" rasterYSize="1000">
  <VRTRasterBand dataType="Float32" band="1" subClass="VRTDerivedRasterBand">
    <Description>Magnitude</Description>
    <PixelFunctionType>MyFirstFunction</PixelFunctionType>
    <SourceTransferType>CFloat64</SourceTransferType>
    ...
```

Default Pixel Functions

Starting with GDAL 2.2, GDAL provides a set of default pixel functions that can be used without writing new code:

- **"real":** extract real part from a single raster band (just a copy if the input is non-complex)
- **"imag":** extract imaginary part from a single raster band (0 for non-complex)
- **"complex":** make a complex band merging two bands used as real and imag values
- **"mod":** extract module from a single raster band (real or complex)
- **"phase":** extract phase from a single raster band [-PI,PI] (0 or PI for non-complex)
- **"conj":** computes the complex conjugate of a single raster band (just a copy if the input is non-complex)
- **"sum":** sum 2 or more raster bands
- **"diff":** computes the difference between 2 raster bands (b1 - b2)
- **"mul":** multiply 2 or more raster bands
- **"cmul":** multiply the first band for the complex conjugate of the second
- **"inv":** inverse (1./x). Note: no check is performed on zero division
- **"intensity":** computes the intensity Re(x*conj(x)) of a single raster band (real or complex)
- **"sqrt":** perform the square root of a single raster band (real only)
- **"log10":** compute the logarithm (base 10) of the abs of a single raster band (real or complex): log10(abs(x))
- **"dB":** perform conversion to dB of the abs of a single raster band (real or complex): 20. * log10(abs(x))
- **"dB2amp":** perform scale conversion from logarithmic to linear (amplitude) (i.e. 10 ^ (x / 20)) of a single raster band (real only)
- **"dB2pow":** perform scale conversion from logarithmic to linear (power) (i.e. 10 ^ (x / 10)) of a single raster band (real only)

Writing Pixel Functions

To register this function with GDAL (prior to accessing any VRT datasets with derived bands that use this function), an application calls GDALAddDerivedBandPixelFunc with a key and a GDALDerivedPixelFunc:

```
GDALAddDerivedBandPixelFunc("MyFirstFunction", TestFunction);
```

A good time to do this is at the beginning of an application when the GDAL drivers are registered.

GDALDerivedPixelFunc is defined with a signature similar to IRasterIO:

Parameters

papoSources A pointer to packed rasters; one per source. The datatype of all will be the same, specified in the eSrcType parameter.

nSources The number of source rasters.

pData The buffer into which the data should be read, or from which it should be written. This buffer must contain at least nBufXSize * nBufYSize words of type eBufType. It is organized in left to right, top to bottom pixel order. Spacing is controlled by the nPixelSpace, and nLineSpace parameters.

nBufXSize The width of the buffer image into which the desired region is to be read, or from which it is to be written.

nBufYSize The height of the buffer image into which the desired region is to be read, or from which it is to be written.

eSrcType The type of the pixel values in the papoSources raster array.

eBufType The type of the pixel values that the pixel function must generate in the pData data buffer.

nPixelSpace The byte offset from the start of one pixel value in pData to the start of the next pixel value within a scanline. If defaulted (0) the size of the datatype eBufType is used.

nLineSpace The byte offset from the start of one scanline in pData to the start of the next.

Returns

CE_Failure on failure, otherwise CE_None.

```
typedef CPLErr
(*GDALDerivedPixelFunc)(void **papoSources, int nSources, void *pData,
                        int nXSize, int nYSize,
                        GDALDataType eSrcType, GDALDataType eBufType,
                        int nPixelSpace, int nLineSpace);
```

The following is an implementation of the pixel function:

```
#include "gdal.h"

CPLErr TestFunction(void **papoSources, int nSources, void *pData,
                    int nXSize, int nYSize,
                    GDALDataType eSrcType, GDALDataType eBufType,
                    int nPixelSpace, int nLineSpace)
{
    int ii, iLine, iCol;
    double pix_val;
    double x0, x3, x4, x8;

    // ---- Init ----
    if (nSources != 4) return CE_Failure;

    // ---- Set pixels ----
    for( iLine = 0; iLine < nYSize; iLine++ )
    {
        for( iCol = 0; iCol < nXSize; iCol++ )
        {
            ii = iLine * nXSize + iCol;
            /* Source raster pixels may be obtained with SRCVAL macro */
            x0 = SRCVAL(papoSources[0], eSrcType, ii);
            x3 = SRCVAL(papoSources[1], eSrcType, ii);
            x4 = SRCVAL(papoSources[2], eSrcType, ii);
            x8 = SRCVAL(papoSources[3], eSrcType, ii);

            pix_val = sqrt((x3*x3+x4*x4)/(x0*x8));

            GDALCopyWords(&pix_val, GDT_Float64, 0,
                          ((GByte *)pData) + nLineSpace * iLine + iCol * nPixelSpace,
                          eBufType, nPixelSpace, 1);
        }
    }

    // ---- Return success ----
    return CE_None;
}
```

Using Derived Bands (with pixel functions in Python)

Starting with GDAL 2.2, in addition to pixel functions written in C/C++ as documented in the **Using Derived Bands (with pixel functions in C/C++)** section, it is possible to use pixel functions written in Python. Both CPython and NumPy are requirements at run-time.

The subelements for VRTRasterBand (whose subclass specification must be set to VRTDerivedRasterBand) are :

- *PixelFunctionType* (required): Must be set to a function name that will be defined as a inline Python module in PixelFunctionCode element or as the form "module_name.function_name" to refer to a function in an external Python module
- *PixelFunctionLanguage* (required): Must be set to Python.
- *PixelFunctionArguments* (optional): It is possible to pass arguments to the Python pixel function by defining attributes in the PixelFunctionArguments element.
- *PixelFunctionCode* (required if PixelFunctionType is of the form "function_name", ignored otherwise). The in-lined code of a Python module, that must be at least have a function whose name is given by PixelFunctionType.
- *BufferRadius* (optional, defaults to 0): Amount of extra pixels, with respect to the original RasterIO() request to satisfy, that are fetched at the left, right, bottom and top of the input and output buffers passed to the pixel function. Note that the values of the output buffer in this buffer zone will be ignored.

The signature of the Python pixel function must have the following arguments:

1. *in_ar*: list of input NumPy arrays (one NumPy array for each source)
2. *out_ar*: output NumPy array to fill. The array is initialized at the right dimensions and with the VRTRasterBand.dataType.
3. *xoff*: pixel offset to the top left corner of the accessed region of the band. Generally not needed except if the processing depends on the pixel position in the raster.
4. *yoff* line offset to the top left corner of the accessed region of the band. Generally not needed.
5. *xsize*: width of the region of the accessed region of the band. Can be used together with out_ar.shape[1] to determine the horizontal resampling ratio of the request.
6. *ysize*: height of the region of the accessed region of the band. Can be used together with out_ar.shape[0] to determine the vertical resampling ratio of the request.
7. *raster_xsize*: total with of the raster band. Generally not needed.
8. *raster_ysize*: total with of the raster band. Generally not needed.
9. *buf_radius*: radius of the buffer (in pixels) added to the left, right, top and bottom of in_ar / out_ar. This is the value of the optional BufferRadius element that can be set so that the original pixel request is extended by a given amount of pixels.
10. *gt*: geotransform. Array of 6 double values.
11. *kwargs*: dictionary with user arguments defined in PixelFunctionArguments

Examples

- VRT that multiplies the values of the source file by a factor of 1.5

```
<VRTDataset rasterXSize="20" rasterYSize="20">
  <SRS>EPSG:26711</SRS>
  <GeoTransform>440720,60,0,3751320,0,-60</GeoTransform>
  <VRTRasterBand dataType="Byte" band="1" subClass="VRTDerivedRasterBand">
    <PixelFunctionType>multiply</PixelFunctionType>
    <PixelFunctionLanguage>Python</PixelFunctionLanguage>
    <PixelFunctionArguments factor="1.5"/>
    <PixelFunctionCode><![CDATA[
import numpy as np
def multiply(in_ar, out_ar, xoff, yoff, xsize, ysize, raster_xsize,
                  raster_ysize, buf_radius, gt, **kwargs):
    factor = float(kwargs['factor'])
    out_ar[:] = np.round_(np.clip(in_ar[0] * factor,0,255))
]]>
```

```
    </PixelFunctionCode>
    <SimpleSource>
      <SourceFilename relativeToVRT="1">byte.tif</SourceFilename>
    </SimpleSource>
  </VRTRasterBand>
</VRTDataset>
```

- VRT that adds 2 (or more) rasters

```
<VRTDataset rasterXSize="20" rasterYSize="20">
  <SRS>EPSG:26711</SRS>
  <GeoTransform>440720,60,0,3751320,0,-60</GeoTransform>
  <VRTRasterBand dataType="Byte" band="1" subClass="VRTDerivedRasterBand">
    <PixelFunctionType>add</PixelFunctionType>
    <PixelFunctionLanguage>Python</PixelFunctionLanguage>
    <PixelFunctionCode><![CDATA[
import numpy as np
def add(in_ar, out_ar, xoff, yoff, xsize, ysize, raster_xsize,
                    raster_ysize, buf_radius, gt, **kwargs):
    np.round_(np.clip(np.sum(in_ar, axis = 0, dtype = 'uint16'),0,255),
              out = out_ar)
]]>
    </PixelFunctionCode>
    <SimpleSource>
      <SourceFilename relativeToVRT="1">byte.tif</SourceFilename>
    </SimpleSource>
    <SimpleSource>
      <SourceFilename relativeToVRT="1">byte2.tif</SourceFilename>
    </SimpleSource>
  </VRTRasterBand>
</VRTDataset>
```

- VRT that computes hillshading using an external library

```
<VRTDataset rasterXSize="121" rasterYSize="121">
  <SRS>EPSG:4326</SRS>
  <GeoTransform>-80.004166666666663,0.008333333333333,0,
44.004166666666663,0,-0.008333333333333</GeoTransform>
  <VRTRasterBand dataType="Byte" band="1" subClass="VRTDerivedRasterBand">
    <ColorInterp>Gray</ColorInterp>
    <SimpleSource>
      <SourceFilename relativeToVRT="1">n43.dt0</SourceFilename>
    </SimpleSource>
    <PixelFunctionLanguage>Python</PixelFunctionLanguage>
    <PixelFunctionType>hillshading.hillshade</PixelFunctionType>
    <PixelFunctionArguments scale="111120" z_factor="30" />
    <BufferRadius>1</BufferRadius>
    <SourceTransferType>Int16</SourceTransferType>
  </VRTRasterBand>
</VRTDataset>
```

with hillshading.py:

```
# Licence: X/MIT
# Copyright 2016, Even Rouault
import math

def hillshade_int(in_ar, out_ar, xoff, yoff, xsize, ysize, raster_xsize,
                          raster_ysize, radius, gt, z, scale):
    ovr_scale_x = float(out_ar.shape[1] - 2 * radius) / xsize
    ovr_scale_y = float(out_ar.shape[0] - 2 * radius) / ysize
    ewres = gt[1] / ovr_scale_x
    nsres = gt[5] / ovr_scale_y
    inv_nsres = 1.0 / nsres
    inv_ewres = 1.0 / ewres

    az = 315
    alt = 45
    degreesToRadians = math.pi / 180

    sin_alt = math.sin(alt * degreesToRadians)
    azRadians = az * degreesToRadians
    z_scale_factor = z / (8 * scale)
    cos_alt_mul_z_scale_factor = \
              math.cos(alt * degreesToRadians) * z_scale_factor
```

```
    cos_az_mul_cos_alt_mul_z_scale_factor_mul_254 = \
                254 * math.cos(azRadians) * cos_alt_mul_z_scale_factor
    sin_az_mul_cos_alt_mul_z_scale_factor_mul_254 = \
                254 * math.sin(azRadians) * cos_alt_mul_z_scale_factor
    square_z_scale_factor = z_scale_factor * z_scale_factor
    sin_alt_mul_254 = 254.0 * sin_alt

    for j in range(radius, out_ar.shape[0]-radius):
        win_line = in_ar[0][j-radius:j+radius+1,:]
        for i in range(radius, out_ar.shape[1]-radius):
            win = win_line[:,i-radius:i+radius+1].tolist()
            x = inv_ewres * ((win[0][0] + win[1][0] + win[1][0] + win[2][0])-\
                             (win[0][2] + win[1][2] + win[1][2] + win[2][2]))
            y = inv_nsres * ((win[2][0] + win[2][1] + win[2][1] + win[2][2])-\
                             (win[0][0] + win[0][1] + win[0][1] + win[0][2]))
            xx_plus_yy = x * x + y * y
            cang_mul_254 = (sin_alt_mul_254 - \
                (y * cos_az_mul_cos_alt_mul_z_scale_factor_mul_254 - \
                    x * sin_az_mul_cos_alt_mul_z_scale_factor_mul_254)) / \
                math.sqrt(1 + square_z_scale_factor * xx_plus_yy)
            if cang_mul_254 < 0:
                out_ar[j,i] = 1
            else:
                out_ar[j,i] = 1 + round(cang_mul_254)

def hillshade(in_ar, out_ar, xoff, yoff, xsize, ysize, raster_xsize,
              raster_ysize, radius, gt, **kwargs):
    z = float(kwargs['z_factor'])
    scale= float(kwargs['scale'])
    hillshade_int(in_ar, out_ar, xoff, yoff, xsize, ysize, raster_xsize,
                  raster_ysize, radius, gt, z, scale)
```

Python module path

When importing modules from inline Python code or when relying on out-of-line code (PixelFunctionType of the form "module_name.function_name"), you need to make sure the modules are accessible through the python path. Note that contrary to the Python interactive interpreter, the current path is not automatically added when used from GDAL. So you may need to define the PYTHONPATH environment variable if you get ModuleNotFoundError exceptions.

Security implications

The ability to run Python code potentially opens the door to many potential vulnerabilities if the user of GDAL may process untrusted datasets. To avoid such issues, by default, execution of Python pixel function will be disabled. The execution policy can be controlled with the GDAL_VRT_ENABLE_PYTHON configuration option, which can accept 3 values:

- YES: all VRT scripts are considered as trusted and their Python pixel functions will be run when pixel operations are involved.
- NO: all VRT scripts are considered untrusted, and none Python pixel function will be run.
- TRUSTED_MODULES (default setting): all VRT scripts with inline Python code in their PixelFunctionCode elements will be considered untrusted and will not be run. VRT scripts that use a PixelFunctionType of the form "module_name.function_name" will be considered as trusted, only if "module_name" is allowed in the GDAL_VRT_TRUSTED_MODULES configuration option. The value of this configuration option is a comma separated listed of trusted module names. The '*' wildcard can be used at the name of a string to match all strings beginning with the substring before the '*' character. For example 'every*' will make 'every.thing' or 'everything' module trusted. '*' can also be used to make all modules to be trusted. The ".*" wildcard can also be used to match exact modules or submodules names. For example 'every.*' will make 'every' and 'every.thing' modules trusted, but not 'everything'.

Linking mechanism to a Python interpreter.

Currently only CPython - 2.6, 2.7 and 3.x - is supported. The GDAL shared object is not explicitly linked at build time to any of the CPython library. When GDAL will need to run Python code, it will first determine if the Python interpreter is loaded in the current process (which is the case if the program is a Python interpreter itself, or if another program, e.g. QGIS, has already loaded the CPython library). Otherwise it will look if the PYTHONSO configuration option is defined. This option can be set to point to the name of the Python library to use, either as a shortname like "libpython2.7.so" if it is accessible through the Linux dynamic loader (so typically in one of the paths in /etc/ld.so.conf or LD_LIBRARY_PATH) or as a full path name like "/usr/lib/x86_64-linux-gnu/libpython2.7.so". The same holds on Windows will shortnames like "python27.dll" if accessible through the PATH or full path names like "c:\python27\python27.dll". If the PYTHONSO configuration option is not defined, it will look for a "python" binary in the directories of the PATH and will try to determine the related shared object (it will retry with "python3" if no "python" has been found). If the above was not successful, then a predefined list of shared objects names will be tried. At the time of writing, the order of versions searched is 2.7, 2.6, 3.4, 3.5, 3.6, 3.3, 3.2. Enabling debug information (CPL_DEBUG=VRT) will show which Python version is used.

Just-in-time compilation

The use of a just-in-time compiler may significantly speed up execution times. Numba has been successfully tested. For better performance, it is recommended to use a offline pixel function so that the just-in-time compiler may cache its compilation.

Given the following mandelbrot.py file :

```
# Trick for compatibility with and without numba
try:
    from numba import jit
    #print('Using numba')
    g_max_iterations = 100
except:
    class jit(object):
        def __init__(self, nopython = True, nogil = True):
            pass

        def __call__(self, f):
            return f

    #print('Using non-JIT version')
    g_max_iterations = 25

# Use a wrapper for the entry point regarding GDAL, since GDAL cannot access
# the jit decorated function with the expected signature.
def mandelbrot(in_ar, out_ar, xoff, yoff, xsize, ysize, raster_xsize,
                       raster_ysize, r, gt, **kwargs):
    mandelbrot_jit(out_ar, xoff, yoff, xsize, ysize, raster_xsize, raster_ysize,
g_max_iterations)

# Will make sure that the code is compiled to pure native code without Python
# fallback.
@jit(nopython=True, nogil=True, cache=True)
def mandelbrot_jit(out_ar, xoff, yoff, xsize, ysize, raster_xsize,
                       raster_ysize, max_iterations):
    ovr_factor_y = float(out_ar.shape[0]) / ysize
    ovr_factor_x = float(out_ar.shape[1]) / xsize
    for j in range( out_ar.shape[0]):
        y0 = 2.0 * (yoff + j / ovr_factor_y) / raster_ysize - 1
        for i in range(out_ar.shape[1]):
            x0 = 3.5 * (xoff + i / ovr_factor_x) / raster_xsize - 2.5
            x = 0.0
            y = 0.0
            x2 = 0.0
            y2 = 0.0
```

```
            iteration = 0
            while x2 + y2 < 4 and iteration < max_iterations:
                y = 2*x*y + y0
                x = x2 - y2 + x0
                x2 = x * x
                y2 = y * y
                iteration += 1

            out_ar[j][i] = iteration * 255 / max_iterations
```

the following VRT file can be used (to be opened with QGIS for example)

```
<VRTDataset rasterXSize="100000000" rasterYSize="100000000">
  <VRTRasterBand dataType="Byte" band="1" subClass="VRTDerivedRasterBand">
    <PixelFunctionLanguage>Python</PixelFunctionLanguage>
    <PixelFunctionType>mandelbrot.mandelbrot</PixelFunctionType>
    <Metadata>
      <MDI key="STATISTICS_MAXIMUM">255</MDI>
      <MDI key="STATISTICS_MEAN">127</MDI>
      <MDI key="STATISTICS_MINIMUM">0</MDI>
      <MDI key="STATISTICS_STDDEV">127</MDI>
    </Metadata>
    <ColorInterp>Gray</ColorInterp>
    <Histograms>
      <HistItem>
        <HistMin>-0.5</HistMin>
        <HistMax>255.5</HistMax>
        <BucketCount>256</BucketCount>
        <IncludeOutOfRange>0</IncludeOutOfRange>
        <Approximate>1</Approximate>
        <HistCounts>0|0|0|0|0|0|0|0|0|0|0|0|0|0|0|0|0|0|0|0|0|0|0|0|0|0|0|0|0|
0|0|0|0|0|0|0|0|0|0|0|0|0|0|0|0|0|0|0|0|0|0|0|0|0|0|0|0|0|0|0|0|0|0|0|0|0|0|0|
0|0|0|0|0|0|0|0|0|0|0|0|0|0|0|0|0|0|0|0|0|0|0|0|0|0|0|0|0|0|0|0|0|0|0|0|0|0|0|
0|0|0|0|0|0|0|0|0|0|0|0|0|0|0|0|0|0|0|0|0|0|0|0|0|0|0|0|0|0|0|0|0|0|0|0|0|0|0|
0|0|0|0|0|0|0|0|0|0|0|0|0|0|0|0|0|0|0|0|0|0|0|0|0|0|0|0|0|0|0|0|0|0|0|0|0|0|0|
0|0|0|0|0|0|0|0|0|0|0|0|0|0|0|0|0|0|0|0|0|0|0|0|0|0|0|0|0|0|0|0|0|0|0|0|0|0|0|
0|0|0|0|0|0|0|0|0|0|0|0|0|0|0|0|0|0|0|0|0|0|0|0|0|0|0|0|0|0|0|0</HistCounts>
      </HistItem>
    </Histograms>
  </VRTRasterBand>
</VRTDataset>
```

Warped VRT

A warped VRT is a VRTDataset with subClass="VRTWarpedDataset". It has a **GDALWarpOptions** element which describe the warping options.

```
<VRTDataset rasterXSize="20" rasterYSize="20" subClass="VRTWarpedDataset">
  <SRS>PROJCS["NAD27 / UTM zone 11N",GEOGCS["NAD27",DATUM["North_American_Datum_1927",SPHEROID["Clarke
        1866",6378206.4,294.9786982138982,AUTHORITY["EPSG","7008"]],AUTHORITY["EPSG","6267"]],PRIMEM["G
        reenwich",0,AUTHORITY["EPSG","8901"]],UNIT["degree",0.0174532925199433,AUTHORITY["EPSG","9122"]
        ],AUTHORITY["EPSG","4267"]],PROJECTION["Transverse_Mercator"],PARAMETER["latitude_of_origin",0]
        ,PARAMETER["central_meridian",-117],PARAMETER["scale_factor",0.9996],PARAMETER["false_easting",
        500000],PARAMETER["false_northing",0],UNIT["metre",1,AUTHORITY["EPSG","9001"]],AXIS["Easting",E
        AST],AXIS["Northing",NORTH],AUTHORITY["EPSG","26711"]]</SRS>
  <GeoTransform>  4.4072000000000000e+05,  6.0000000000000000e+01,  0.0000000000000000e+00,
        3.7513200000000000e+06,  0.0000000000000000e+00, -6.0000000000000000e+01</GeoTransform>
  <Metadata>
    <MDI key="AREA_OR_POINT">Area</MDI>
  </Metadata>
  <VRTRasterBand dataType="Byte" band="1" subClass="VRTWarpedRasterBand">
    <ColorInterp>Gray</ColorInterp>
  </VRTRasterBand>
  <BlockXSize>20</BlockXSize>
  <BlockYSize>20</BlockYSize>
  <GDALWarpOptions>
    <WarpMemoryLimit>6.71089e+07</WarpMemoryLimit>
    <ResampleAlg>NearestNeighbour</ResampleAlg>
    <WorkingDataType>Byte</WorkingDataType>
    <Option name="INIT_DEST">0</Option>
    <SourceDataset relativeToVRT="1">byte.vrt</SourceDataset>
    <Transformer>
      <ApproxTransformer>
        <MaxError>0.125</MaxError>
        <BaseTransformer>
```

```
          <GenImgProjTransformer>
            <SrcGeoTransform>440720,60,0,3751320,0,-60</SrcGeoTransform>

        <SrcInvGeoTransform>-7345.33333333333303,0.0166666666666666664,0,62522,0,-0.0166666666666666664
        </SrcInvGeoTransform>
            <DstGeoTransform>440720,60,0,3751320,0,-60</DstGeoTransform>

        <DstInvGeoTransform>-7345.33333333333303,0.0166666666666666664,0,62522,0,-0.0166666666666666664
        </DstInvGeoTransform>
          </GenImgProjTransformer>
        </BaseTransformer>
      </ApproxTransformer>
    </Transformer>
    <BandList>
      <BandMapping src="1" dst="1" />
    </BandList>
  </GDALWarpOptions>
</VRTDataset>
```

Pansharpened VRT

(Since GDAL 2.1)

A VRT can describe a dataset resulting from a pansharpening operation. The pansharpening VRT combines a panchromatic band with several spectral bands of lower resolution to generate output spectral bands of the same resolution as the panchromatic band.

VRT pansharpening assumes that the panchromatic and spectral bands have the same projection (or no projection). If that is not the case, reprojection must be done in a prior step. Bands might have different geotransform matrices, in which case, by default, the resulting dataset will have as extent the union of all extents.

Currently the only supported pansharpening algorithm is a "weighted" Brovey algorithm. The general principle of this algorithm is that, after resampling the spectral bands to the resolution of the panchromatic band, a pseudo panchromatic intensity is computed from a weighted average of the spectral bands. Then the output value of the spectral band is its input value multiplied by the ratio of the real panchromatic intensity over the pseudo panchromatic intensity.

Corresponding pseudo code:

```
pseudo_panchro[pixel] = sum(weight[i] * spectral[pixel][i] for i=0 to nb_spectral_bands-1)
ratio = panchro[pixel] / pseudo_panchro[pixel]
for i=0 to nb_spectral_bands-1:
    output_value[pixel][i] = input_value[pixel][i] * ratio
```

A valid pansharpened VRT must declare subClass="VRTPansharpenedDataset" as an attribute of the VRTDataset top element. The VRTDataset element must have a child **PansharpeningOptions** element. This PansharpeningOptions element must have a **PanchroBand** child element and one of several **SpectralBand** elements. PanchroBand and SpectralBand elements must have at least a **SourceFilename** child element to specify the name of the dataset. They may also have a **SourceBand** child element to specify the number of the band in the dataset (starting with 1). If not specify, the first band will be assumed.

The SpectralBand element must generally have a **dstBand** attribute to specify the number of the output band (starting with 1) to which the input spectral band must be mapped. If the attribute is not specified, the spectral band will be taken into account in the computation of the pansharpening, but not exposed as an output band.

Panchromatic and spectral bands should generally come from different datasets, since bands of a GDAL dataset are assumed to have all the same dimensions. Spectral bands themselves can come from one or several datasets. The only constraint is that they have all the same dimensions.

An example of a minimalist working VRT is the following. It will generates a dataset with 3 output bands corresponding to the 3 input spectral bands of multispectral.tif, pansharpened with panchromatic.tif.

```
<VRTDataset subClass="VRTPansharpenedDataset">
  <PansharpeningOptions>
    <PanchroBand>
      <SourceFilename relativeToVRT="1">panchromatic.tif</SourceFilename>
      <SourceBand>1</SourceBand>
    </PanchroBand>
    <SpectralBand dstBand="1">
      <SourceFilename relativeToVRT="1">multispectral.tif</SourceFilename>
      <SourceBand>1</SourceBand>
    </SpectralBand>
    <SpectralBand dstBand="2">
      <SourceFilename relativeToVRT="1">multispectral.tif</SourceFilename>
      <SourceBand>2</SourceBand>
    </SpectralBand>
    <SpectralBand dstBand="3">
      <SourceFilename relativeToVRT="1">multispectral.tif</SourceFilename>
      <SourceBand>3</SourceBand>
    </SpectralBand>
  </PansharpeningOptions>
</VRTDataset>
```

In the above example, 3 output pansharpend bands will be created from the 3 declared input spectral bands. The weights will be 1/3. Cubic resampling will be used. The projection and geotransform from the panchromatic band will be reused for the VRT dataset.

It is possible to create more explicit and declarative pansharpened VRT, allowing for example to only output part of the input spectral bands (e.g. only RGB when the input multispectral dataset is RGBNir). It is also possible to add "classic" VRTRasterBands, in addition to the pansharpened bands.

In addition to the above mentioned required PanchroBand and SpectralBand elements, the PansharpeningOptions element may have the following children elements :

- **Algorithm**: to specify the pansharpening algorithm. Currently, only WeightedBrovey is supported.
- **AlgorithmOptions**: to specify the options of the pansharpening algorithm. With WeightedBrovey algorithm, the only supported option is a **Weights** child element whose content must be a comma separated list of real values assigning the weight of each of the declared input spectral bands. There must be as many values as declared input spectral bands.
- **Resampling**: the resampling kernel used to resample the spectral bands to the resolution of the panchromatic band. Can be one of Cubic (default), Average, Near, CubicSpline, Bilinear, Lanczos.
- **NumThreads**: Number of worker threads. Integer number or ALL_CPUS. If this option is not set, the GDAL_NUM_THREADS configuration option will be queried (its value can also be set to an integer or ALL_CPUS)
- **BitDepth**: Can be used to specify the bit depth of the panchromatic and spectral bands (e.g. 12). If not specified, the NBITS metadata item from the panchromatic band will be used if it exists.
- **NoData**: Nodata value to take into account for panchromatic and spectral bands. It will be also used as the output nodata value. If not specified and all input bands have the same nodata value, it will be implicitly used (unless the special None value is put in NoData to prevent that).
- **SpatialExtentAdjustment**: Can be one of *Union* (default), *Intersection*, *None* or *NoneWithoutWarning*. Controls the behaviour when panchromatic and spectral bands have not the same geospatial extent. By default, Union will take the union of all spatial extents. Intersection

the intersection of all spatial extents. None will not proceed to any adjustment at all (might be useful if the geotransform are somehow dummy, and the top-left and bottom-right corners of all bands match), but will emit a warning. NoneWithoutWarning is the same as None, but in a silent way.

The below examples creates a VRT dataset with 4 bands. The first band is the panchromatic band. The 3 following bands are than red, green, blue pansharpened bands computed from a multispectral raster with red, green, blue and near-infrared bands. The near-infrared bands is taken into account for the computation of the pseudo panchromatic intensity, but not bound to an output band.

```
<VRTDataset rasterXSize="800" rasterYSize="400" subClass="VRTPansharpenedDataset">
  <SRS>WGS84</SRS>
  <GeoTransform>-180, 0.45, 0, 90, 0, -0.45</GeoTransform>
  <Metadata>
    <MDI key="DESCRIPTION">Panchromatic band + pan-sharpened red, green and blue bands</MDI>
  </Metadata>
  <VRTRasterBand dataType="Byte" band="1" >
    <SimpleSource>
      <SourceFilename relativeToVRT="1">world_pan.tif</SourceFilename>
      <SourceBand>1</SourceBand>
    </SimpleSource>
  </VRTRasterBand>
  <VRTRasterBand dataType="Byte" band="2" subClass="VRTPansharpenedRasterBand">
      <ColorInterp>Red</ColorInterp>
  </VRTRasterBand>
  <VRTRasterBand dataType="Byte" band="3" subClass="VRTPansharpenedRasterBand">
      <ColorInterp>Green</ColorInterp>
  </VRTRasterBand>
  <VRTRasterBand dataType="Byte" band="4" subClass="VRTPansharpenedRasterBand">
      <ColorInterp>Blue</ColorInterp>
  </VRTRasterBand>
  <BlockXSize>256</BlockXSize>
  <BlockYSize>256</BlockYSize>
  <PansharpeningOptions>
    <Algorithm>WeightedBrovey</Algorithm>
    <AlgorithmOptions>
      <Weights>0.25,0.25,0.25,0.25</Weights>
    </AlgorithmOptions>
    <Resampling>Cubic</Resampling>
    <NumThreads>ALL_CPUS</NumThreads>
    <BitDepth>8</BitDepth>
    <NoData>0</NoData>
    <SpatialExtentAdjustment>Union</SpatialExtentAdjustment>
    <PanchroBand>
      <SourceFilename relativeToVRT="1">world_pan.tif</SourceFilename>
     <SourceBand>1</SourceBand>
    </PanchroBand>
    <SpectralBand dstBand="2">
      <SourceFilename relativeToVRT="1">world_rgbnir.tif</SourceFilename>
      <SourceBand>1</SourceBand>
    </SpectralBand>
    <SpectralBand dstBand="3">
      <SourceFilename relativeToVRT="1">world_rgbnir.tif</SourceFilename>
      <SourceBand>2</SourceBand>
    </SpectralBand>
    <SpectralBand dstBand="4">
      <SourceFilename relativeToVRT="1">world_rgbnir.tif</SourceFilename>
      <SourceBand>3</SourceBand>
    </SpectralBand>
    <SpectralBand> <!-- note the absence of the dstBand attribute, to indicate
                        that the NIR band is not bound to any output band -->
      <SourceFilename relativeToVRT="1">world_rgbnir.tif</SourceFilename>
      <SourceBand>4</SourceBand>
    </SpectralBand>
  </PansharpeningOptions>
</VRTDataset>
```

Multi-threading issues

The below section applies to GDAL <= 2.2. Starting with GDAL 2.3, the use of VRT datasets is subject to the standard GDAL dataset multi-threaded rules (that is a VRT dataset handle may only be used by a

same thread at a time, but you may open several dataset handles on the same VRT file and use them in different threads)

When using VRT datasets in a multi-threading environment, you should be careful to open the VRT dataset by the thread that will use it afterwards. The reason for that is that the VRT dataset uses GDALOpenShared when opening the underlying datasets. So, if you open twice the same VRT dataset by the same thread, both VRT datasets will share the same handles to the underlying datasets.

The shared attribute, added in GDAL 2.0.0, on the SourceFilename indicates whether the dataset should be shared (value is 1) or not (value is 0). The default is 1. If several VRT datasets referring to the same underlying sources are used in a multithreaded context, shared should be set to 0. Alternatively, the VRT_SHARED_SOURCE configuration option can be set to 0 to force non-shared mode.

Performance considerations

A VRT can reference many (hundreds, thousands, or more) datasets. Due to operating system limitations, and for performance at opening time, it is not reasonable/possible to open them all at the same time. GDAL has a "pool" of datasets opened by VRT files whose maximum limit is 100 by default. When it needs to access a dataset referenced by a VRT, it checks if it is already in the pool of open datasets. If not, when the pool has reached its limit, it closes the least recently used dataset to be able to open the new one. This maximum limit of the pool can be increased by setting the GDAL_MAX_DATASET_POOL_SIZE configuration option to a bigger value. Note that a typical user process on Linux is limited to 1024 simultaneously opened files, and you should let some margin for shared libraries, etc... As of GDAL 2.0, gdal_translate and gdalwarp, by default, increase the pool size to 450.