**OptimizeRasters – Users Guide**

Applicable for OptimizeRasters Version 20151206 (see Header of OptimizeRasters.py)

***Note: Optimize Rasters and the Associated MRF Drivers for GDAL are provides as a Prototype and testing mode only. The functionality has not been exhaustively tested and are not currently covered under ArcGIS Support. Questions or suggestions related to the running of these workflows should be addressed to the forum associated with the download or sent to***

[***ImageManagementWorkflows@esri.com***](mailto:ImageManagementWorkflows@esri.com)

**Introduction**

OptimizeRasters is a command line tool that converts raster from one format to another, and can also be used to move rasters and other data to and from cloud storage. During data conversion the output format is optimized so as to improve read performance. This is primarily achieved by ensuring that the data is internally tiled and includes appropriate pyramids. Options are also included to compress the imagery, so as to save storage space.

OptimizeRasters supports two output format: TIF and MRF. TIF is a very popular format that can contain pixels in different layouts. TIF files can be optimized for access if the pixels internally broken into tiles. Many TIF files are not tiled. For example the standard products Digital Globe and USGS are not tiled, so it is often advantageous to run OptimizeRasters to copy the data from one set of directories to another and so restructuring the files and enable faster access. MRF is a simple format for the storage of rasters as collections of tiles and is optimized for cloud storage and access. MRF also includes additional compression options not supported in TIF. In many cases it is advantageous to optimize rasters by converting them to MRF.

One of the primary users of OptimizeRasters is to copy rasters into cloud storage such as Amazon S3 so as to provide lower cost storage, yet provide scalable and elastic data access. OptimizeRasters includes special functionality to work with imagery stored in Amazon S3 buckets. It also has the capability to write the intermediate data on fast disk during the conversion process if the input is from a slower disk or S3.

There are many parameters that can be configured based on the output raster to be written and what compression and pyramids are to be created. These configuration parameters are defined in configuration files so as to simplify the repeated calling of the command on different datasets.

OptimizeRasters is implemented as open source Python code accessing the GDAL\_translate and GDALaddo tools. The Open Source code enables user to modify the code and further customize it if required. By utilizing GDAL OptimizeRasters can read as input a large number of formats. If the output format does not support the associated metadata then it is written to an .AUX.XML files along with the resulting output files.

**Overview of MRF (MetaRasterFormat)**

MRF exists as a format and a data access driver in ArcGIS. The MRF format is a very simple raster format optimized for cloud storage, that typically breaks a rasters into 3 files: A small MRF XML file that contains metadata and properties of the raster including references to the data and tile files. A single data file consisting of the data and any pyramid stored as tiles. A single index file that is used by applications to quickly extract the required tiles needed to cover an area of interest at the appropriate scale. Splitting a raster into these three files provides advantages for cloud based access.

The MRF raster driver in ArcGIS enables fast reading of the MRF formatted rasters, but also enables caching of other file formats so as to speed up data access from slower disks. Currently the installation of the MRF driver in ArcGIS requires the replacement of GDAL18.DLL within both ArcGIS for Desktop and Server. Once the MRF driver is installed, ArcGIS sees an MRF file as another raster dataset. One of the advantages of the MRF driver is that it can reference imagery stored both on a file system, but also on web accessible storage such as S3.

The MRF driver can work in 4 modes:

StaticMRF – In this mode the MRF file acts in a similar way to a standard raster dataset eg TIF and the driver reads the tiles as required.

SplitMFR – One advantage of fact that MRF are split into three files, it is possible to store the large tiled data file on slower tiered storage, while keeping the metadata and index files on faster storage. This can speed up access by reducing requests to slower storage devices.

CloneMRF – In this mode the MRF is modified to define a local cache location for the storage of tiles. When the driver accesses a tile from the source MRF data file, it keeps a copy of the tile on a local storage. Subsequent request for the same tile result in use of the cache speeding up access. An optimum way of scaling image access is to store imagery as MRF format on cloud storage and then use CloneMRF to access.

CachingMRF – This is similar to CloneMRF, but the data source can be nearly any raster format stored on a file system or cloud storage. The rasters stored on the cloud storage will only be read when required and will be cached locally. In this way it is possible to get ArcGIS to directly read a range of formats stored on cloud storage (or slow tiered storage) and have performance improved by tile caching. The cache is typically stored in compressed format using LERC compression.

Note that the MRF driver works on both ArcGIS for Desktop and ArcGIS server. It is possible to directly read MRF files as rasters in ArcGIS for desktop with the data stored on Amazon S3 storage. Similarly one can create a Mosaic Dataset from large collections of MRF files. The MRF files are stored locally to desktop or server, but can reference rasters stored on slower storage. As the MRF files are all small, the data management process can be much faster. To serve the imagery the same MRF files can be transferred to a server, which will then reference the same source data. For best performance the server should be located as close as possible to data store. In the Amazon AWS cloud this is done by putting the server in the same region as the S3 storage.

**Usage Patterns for OptimizeRasters**

OptimizeRasters can be used in various ways to assist in Image Management and Sharing. The following are some example workflows

**Optimizing access to data collections from a data provider to a network attached storage**

You may have a set of directories that contain data from a source, but the data is not optimized for reading. Typical examples would be data from most satellite imagery vendors who deliver imagery as TIF files that are not tiled and do not have pyramids, but do have other metadata associated with the files. You can use OptimizeRasters to copy the all the data from one directory (possibly on an external hard disk)to a second device (eg your organizations shared file storage. OptimizeRasters will copy all files including metadata, but will also convert the TIF files into TiledTIF. The resulting file names will be the same, but the TIF files will be faster to access. Nearly all applications that access TIF files, can access TiledTIF without modification, so most traditional data access methods should work. Using OptimizeRasters in this way is similar to using the ArcGIS Copy Rasters command, but it ensures that all the data files and not only the rasters are copied and converted.

A variation of the same workflow is to have the format of the data converted during copying. For example the source data may be delivered in a flavor of JP2 that is slow to read. OptimizeRasters can convert the format to TIF which is faster to read (although larger in size). Programs written using GDAL use the first bytes of a file to identify the file format and not the extension. Therefor it is often possible to convert the files and keep the original extension the same. This is sometimes required then a product includes metadata files that reference the raster data by name. To enable different extension renaming OptimizeRasters includes an option to rename the extension of the rasters or not.

Another variation of the same workflow is to include a compression option to the conversion process so that the resulting data is compressed reducing storage space. This compression can be lossless (eg using Deflate) or lossy (eg using JPEG). When using MRF the compression can be LERC which is a lossless and controlled lossy compression algorithm especially valuable for higher bit depth data such as newer satellite imagery and elevation models.

**Copying data to Cloud storage**

A common workflow is to use OptimizeRaster to copy raster data to cloud storage so that it can be accessed using elastic compute while reducing storage costs. By defining the destination (output directory) to be cloud storage (such as S3), OptimizeRasters can be used to move data to cloud storage. This transfer typically includes conversion of the raster format to MRF, but OptimizeRaters can also be used to transfer data with the output as TiledTIF or without conversion.

**Creating CloneMRF or Caching MRF files**

ArcGIS inherently can only access rasters by referencing a file on a local or shared file system. This raises the issue on how to access imagery that is stored on cloud storage. The solution is to create a CloneMRF or CachingMFR file that references the source on the cloud storage (or tiered file storage). OptimizeRasters can be used to copy to a local (preferably fast direct access) storage a directory of rasters including auxiliary files that may contain specific metadata. If the source data is a MRF file then a CloneMRF file can be created on the local drive. If the source is another format then a CachingMRF file is created. The resulting directory will contain the all the same files as the source, but the volume will be considerably smaller as the original raster files will be replaced by small MRF files.

**Installing Optimize Rasters**

Download the zip file from the Resource Center on ArcGIS Online or Github

Unzip the zip file to C:\ Image\_Mgmt\_Workflows\

If uploading to s3 or downloading from s3; boto is required to be installed.

First download pip.py from <https://pip.pypa.io/en/latest/installing.html#python-os-support>

Place it in c:\Python27\ArcGIS10.3 and from the windows command prompt in the same directory enter the following command:

python.exe get-pip.py

Then go to the folder where pip is installed (i.e c:\Python27\ArcGIS10.3\Scripts) and in the command prompt run the following

pip install boto

(Be sure the command is run from the same path where the pip scripts is present)

Note that OptimizeRasters is distributed with the necessary GDAL binaries for its operation. The default path for GDAL binaries are located at (tools/bin) folder relative to the OptimizeRasters package root.

**Installing MRF driver on ArcGIS Desktop and Server**

The following installation should only be attempted on ArcGIS for Desktop or Server 10.3.1

Not that a hotfix for ArcGIS is expected to be released in Summer 2015, that will include the appropriate DLLs so removing the following manual installation requirement.

**For ArcGIS for Desktop 10.3.1.**

Ensure all instances of ArcGIS for Desktop are closed

Rename the file *c:\Program Files (x86)\ArcGIS\Desktop10.3\bin\gdal18.dll* to *gdal18.dll.org*

If there are restriction (eg access rights) to changing the DLLs then these need to first be resolved.

Copy the file ***OptimizeRasters.zip****\OptimizeRasters\tools\bin\gdal18.dll* to

*c:\Program Files (x86)\ArcGIS\Desktop10.3\bin\gdal18.dll*

Note: Prior to running a hot fix on ArcGIS, please rename gdal18.dll.org back to gdal18.dll

**For ArcGIS for Server 10.3.1.**

Ensure all instances of ArcGIS for Server are closed. One way to achieve this is to run the command “services.msc” and stop the *‘ArcGIS Server’* service

Rename the file *c:\Program Files\ArcGIS\Server\bin\gdal18.dll* to *gdal18.dll.org*

If there are restriction (eg access rights) to changing the DLLs then these need to first be resolved.

Copy the file ***OptimizeRasters.zip****\OptimizeRasters\tools\bin\****64****\gdal18.dll* to

c:\Program Files (x86)\ArcGIS\Desktop10.3\bin\gdal18.dll

Note: The 64bit version for ArcGIS server is in the \64 subdirectory. Prior to running a hot fix on ArcGIS, please rename *gdal18.dll.org* back to *gdal18.dll*

Restart the *‘ArcGIS Server’* service.

Note: Prior to running a hot fix on ArcGIS, please rename gdal18.dll.org back to gdal18.dll

**Running Optimize Rasters**

Optimize Rasters is a command line tool. Currently there is no graphical user interface or GPTool (although these tools may soon be developed).

Help for the command can be obtained by using the following command:

**<path\_to\_python.exe> <path\_to\_optimizerastes.py> --help**

A common command line usage is:

<path to python.exe> <path to optimizerasters.py> -input=<path to input folder> -output=<path to outputfolder> -mode=mrf

The following command line arguments can be used:

-input = Input directory path. Note all files in a directory will be processed.

-output = Output directory path

-config = Configuration file with default settings. All parameters not defined on the command line will be taken from this file. If undefined then OptimizeRasters.xml stored in the same location as OptimizeRasters.py will be used. The configuration file can be used to define often used configuration setting so reducing the command line parameters required. Parameter defined in the command line override configuration parameters.

-mode = Processing mode/output format, value should be mrf, cachingmrf, clonemrf, tif. Refer below to the meaning of modes

-cache = cache output directory path. Location to embed into MRF files for where to create the cache files. If not defined then will be the same location as the MRF files. (see below for information on cache management)

-quality =JPEG quality if compression is jpeg

-prec = LERC precision

-pyramids =If to generate pyramids- value should be true/false. Typically pyramids should always be created. Used to override the BuildPyramids parameter in config file

-clouddownload = If the input is a S3/Azure cloud storage folder this value should be true else false

-cloudupload = If the input is an S3 folder this value should be true else false

-subs = Include sub-directories - value should be true/false. If true all sub directories of the input directory will also be processed

-tempinput=Path to copy -input raters before conversion. If the input is S3 then this is a required parameter. The input rasters will be temporarily copied to this directory before processing.

-tempoutput=Path to copy converted rasters before moving to output. If the output is S3 then this is a required parameter. The input rasters will be temporarily copied to this directory before being uploaded to S3. It is optimum if this is on a faster drive as pyramids will be generated by reading and writing to this locaiton.

All the above arguments can be defined in the configuration file, reducing the need for command line parameters. The configuration file contains additional parameters, with explanations of their use.

Some of the important configuration parameters the may need to be change include:

RasterFormatFilter – Defined the file extension of the rasters that should be converted. Typically this is set to “tif,TIF,mrf” and defines that files with these extensions are rasters that should be converted. All other files (eg JPEG file that may contain a Logo) do not get converted, but get copied from source to destination. Note the file extensions are case sensitive hence the definition of tif and TIF.

ExcludeFilter – Defines the extension of files that should not be copied as they are not needed. Typically if data is being converted say to TIF with internal pyramids or MRF files then existing ovr or rrd files are not required. Similarly there are other files that would typically be replaced with new files such as idx, lrc, mrf\_cache, pjp, ppng, pft, pzp. In some workflows other file extension from the source are also not required such as tfw aux.xml and can in included in this list.

KeepExtension – Defines if the output raster extensions should be changed or not. If true the extension will remain the same as the input even if the format was changed (eg to MRF). If false then the extension will be that most appropriate for the output format (typically .tif or .mrf)

PyramidFactor – Defines the factors to build. Typically this is 2 4 8 16 32 64. It can be set to a factor of 3 such as 3 9 27 81

PyramidSampling – Defines sampling to use for pyramids such as average or nearest. If undefined then average will be used except for some specific cases such as quality files for Landsat8

NoDataValue – Defines if there is a NoDataValue in the source. This is important to define if rasters have 0 as NoData so that the pyramid generation will not include this value.

BlockSize – Define the size of the tiles (sometimes called blocks) in the output image. A value of 512 is recommended for most datasets. Smaller values that are powers of 2 (eg 128,256) may be used in dataset where the generation of temporal profiles is common.

**Resume interrupted workflows.**

OptimizeRasters has a built in resume functionality to help continue a workflow from a point of interruption. This is immensely helpful in cases where only a few of the input files have failed to process and to avoid redoing the entire workflow each time something goes wrong and not to forget the cost associated with accessing data hosted on a Cloud FS. OptimizeRasters by default takes care of processing only those files that have failed and there’s no extra effort is necessary to get the workflow restarted to complete only the failed ones. This therefore increases the productivity by freeing the user to work with something else.

By default OptimizeRasters creates a (Job) file each time it’s run. A Job file is a simple text file which has a header block at the start, a series of key value pairs each starting with ‘#’ and followed by a list of files to process at the source input location. The naming convention for the (Job) file is OR\_YYYYMMDDTHHMMSSssssss.orjob. The syntax followed by the underscore is the DATATIME format in the ISO format and denotes the date and time when the workflow was first run. However, it’s possible to name the job files using the command-line flag –job along with other parameter that are required to process a workflow. The default .orjob extension will be added if omitted.

The resume functionality or the creation of the Job file by OptimizeRasters was designed to work in silence and behind the scene hence shouldn’t require any attention by the user. On completion of processing a workflow successfully, the Job file will be moved to the Job folder for archive purpose or is left behind at the OptimizeRasters.py location in case of any errors. Users can resume job files that are left behind by using the –input command line flag pointing at the Job file.

E.g. python OptionizeRasters.py –input=OR\_20151211T024329630000.orjob

To inspect any errors however, the user may manually inspect the Job file to see the list of files that have reported errors or the processing stage at which each file has failed.

**User parameters**

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| Name in configuration file | | Command-line equivalent | Description | Default | Value accepted | Is value case-sensitive | Notes |
| <mode> | | -mode | Processing mode/output format |  | tif  tif\_lzw  tif\_jpeg  mrf  mrf\_jpeg  cachingmrf  clonemrf  splitmrf | No |  |
| <In\_S3\_ParentFolder> | | -input | Input raster folder | The default path in <In\_S3\_ParentFolder> is used if the  -clouddownload is set to any valid boolean (true) value. | Any valid folder path. | Yes if pointed at an AWS key path. | -input can be a UNC/local folder path or a AWS/S3 path.  <In\_S3\_ParentFolder> applicable only if the  -clouddownload is set to a boolean (true) value  Valid boolean values are [true, yes, t, 1, y, false, no, f, 0, n] |  |
| <Out\_S3\_ParentFolder> | | -output | Output to store converted rasters and any associated metadata files. | The default path in <Out\_S3\_ParentFolder> is used if the command-line flag –cloudupload or the <CloudUpload> in the parameter file is set to a valid boolean (true) value. |  | Yes if pointed at an AWS key path. | -output can be a UNC/local folder or a valid cloud storage path [Amazon, Azure].  <Out\_S3\_ParentFolder> is applicable only if  - cloudupload or < CloudUpload> is set to a boolean (true) value |
| N/A | | -config | Path to the configuration file with default user settings | The default configuration file (OptimizeRasters.xml) is used unless overridden by the –config flag. |  |  | Cannot point at a S3 key. |
| <IncludeSubdirectories> | | -subs | Include sub-directories in –input | False | Boolean |  | Boolean flag to enable scanning for subfolders |
| N/A | | -tempinput | Folder to copy -input raters before conversion |  |  |  | Tip. Processing can be moved to a drive for performance reasons. Supported only UNC and local file paths.  - |
| N/A | | -tempoutput | Folder to output converted rasters before moving to (-output) folder. |  |  |  | Tip: Moving output machine which offers better upload bandwidth support. Supported only UNC and local file paths. |
| N/A | | -cache | Cache output directory |  |  |  |  |
| <Quality> | | -quality | JPEG quality if compression is jpeg. | 85 | Numeric value |  | Applicable only if the  –mode is related to (jpeg) compression. |
| <LERCPrecision> | | -prec | LERC precision to apply for LERC compression |  | 0.5 |  |  |
| <BuildPyramids> | | -pyramids | Generate pyramids |  | True | Yes if pointed at an AWS key path. | Valid boolean values are [true, yes, t, 1, y, false, no, f, 0, n]  or  ‘only’.  In case of ‘only’ the –input and –output must be pointed at the same folder to generate just the pyramids. |
| <RasterFormatFilter> | | N/A | File extensions considered as (Rasters). These files will not be copied from the input path. | tif,TIF,mrf | Comma separated values. | Yes. | Case sensitive for local as well as for S3. |
| <ExcludeFilter> | | N/A | File extensions to ignore complete while copying files/data from the  -input path | tmp, ovr, rrd, aux.xml, lrc, mrf\_cache, pjp, ppng, pft, pzp | Comma separated values. | Yes | Case sensitive for local as well as for S3. |
| <Interleave> | | N/A | Value for the GDAL Translate (INTERLEAVE) key. |  |  |  | Applicable only if the  –mode is related to (jpeg) compression. |
| <PyramidFactor> | | N/A | Pyramid levels to create | 2 | Comma separated values. |  |  |
| <PyramidSampling> | | N/A | Sampling to use for pyramids | average |  | nearest,  average,  gauss,  cubic,  cubicspline  lanczos,  average\_mp,  average\_magphase,  mode | gdaladdo.exe –r |
| <PyramidCompression> | | N/A | Pyramid compression | Jpeg |  | jpeg,  lzw,  deflate | Gdaaddo.exe –config COMPRESS\_OVERVIEW |
| <NoDataValue> | | N/A | No data value |  |  |  | gdal\_translate.exe -a\_nodata |
| <BlockSize> | | N/A | Output title size | 512 | Numeric value |  | gdal\_translate.exe –co –BLOCKSIZE |
| <Scale> | | N/A | Uniform scale factor for rotation scaling. | 2 |  |  | gdal\_translate.exe –co -UNIFORM\_SCALE |
| <KeepExtension> | | N/A | Keep the output raster extension same as the input raster file. If set to a valid (false) boolean value, the output raster extension will be renamed to ‘mrf’ | False | Boolean |  | Valid boolean values are [true, yes, t, 1, y, false, no, f, 0, n] |
| <Threads> | | N/A | Simultaneous threads to use for parallel processing /instances of gdal\_translate/gdal\_addo/etc. | 10 | Numeric value |  |  |
| <LogPath> | | N/A | Folder to store log files. | Log folder at the OptimizeRasters root. |  |  | Must be a UNC or a local folder. S3 keys not supported |
| <In\_S3\_AWS\_ProfileName> | | N/A | Input AWS profile name. |  |  | No | Refer to [Working with S3/AWS credentials usage |
| <In\_S3\_ID> | | N/A | Input AWS access key ID |  |  | Yes | Refer to [Working with S3] |
| <In\_S3\_Secret> | | N/A | Input AWS access secret key |  |  | Yes | Refer to [Working with S3] |
| <In\_S3\_Bucket> | | N/A | Input AWS bucket name |  |  | Yes |  |
| <Out\_S3\_AWS\_ProfileName> | | N/A | Output AWS profile name. |  |  | No | Refer to [Working with S3/AWS credentials usage |
| <Out\_S3\_ID> | | N/A | Output AWS access key ID |  |  | Yes | Refer to [Working with S3] |
| <Out\_S3\_Secret> | | N/A | Output AWS access secret key |  |  | Yes | Refer to [Working with S3] |
| <Out\_S3\_Bucket> | | N/A | Output AWS bucket name |  |  | Yes |  |
| <Out\_S3\_DeleteAfterUpload> | | N/A | Delete locally generated output files once successfully uploaded to Amazon/Azure cloud storage. | False | Boolean |  | Default depends on the value in the configuration file. |
| *<Out\_S3\_Upload>* | *-s3output. See [notes]* | |  |  |  |  | ***Deprecated****.* ***See <CloudUpload>.***  *–s3output*  *command-line flag and the node <Out\_S3\_Upload> in the parameter file*  *have been deprecated.* |
| <CloudUpload> | | -cloudupload | Is -output a cloud storage? | False | Boolean | No |  |
| <Out\_Cloud\_Type> | | -clouduploadtype | Select upload Cloud Type | See [Notes] | From list | No | No default. Value must be preset in the configuration file. |
| <Out\_Azure\_ParentFolder> | | -output | Root path to upload files. Works similar to **<Out\_S3\_ParentFolder>** |  |  | Yes |  |
| <Out\_Azure\_AccountName> | |  | Account name. Similar to **<Out\_S3\_ID>** |  |  | Yes |  |
| <Out\_Azure\_AccountKey> | |  | Account Key. Works similar to **<Out\_S3\_Secret>** |  |  | Yes |  |
| <Out\_Azure\_Container> | |  | Container Name. Similar to **<Out\_S3\_Bucket>** |  |  | Yes |  |
| <Out\_Azure\_Access> | |  | access type. Similar to **<Out\_S3\_ACL>** for S3.  Valid values are value values are [**private**, **blob**, **container**] |  | From list. | No | **private:** accessible only to the user  **blob:** Files within the container are publicly accessible.  **Container**:  Same as (blob) plus container metadata are publicly accessible. |
| <In\_Cloud\_Type> | | -clouddownloadtype | Select the download cloud storage type. [**Azure**, **Amazon**] |  | From list | No |  |
| <In\_Azure\_ParentFolder> | | -input | Root path to download files. Works similar to **<In\_S3\_ParentFolder>** |  |  | Yes | Use forward slash in path for clarity. |
| <In\_Azure\_AccountName> | |  | Account name. Similar to **<Out\_Azure\_AccountName>** |  |  | Yes |  |
| <In\_Azure\_AccountKey> | |  | Account Key. Works similar to **<Out\_Azure\_AccountKey>** |  |  | Yes |  |
| <In\_Azure\_Container> | |  | Container Name. Similar to **<Out\_Azure\_Container>** |  |  | Yes |  |
| <In\_Azure\_Access> | |  | access type. Similar to **<Out\_Azure\_Access>**. Valid values are [**private**, **blob**, **container**] |  | From list. | No | **private:** accessible only to the user  **blob:** Files within the container are publicly accessible.  **Container**:  Same as (blob) plus container metadata are publicly accessible. |

**Working with S3**

**AWS standards to manage S3 credentials**

OptimizeRasters supports the AWS standards to manage credentials. This means credentials can use the default environment variables (AWS\_ACCESS\_KEY\_ID, AWS\_ACCESS\_KEY\_ID) or by using the default AWS credential file located at

%USERPROFILE%\.aws\credentials.

Please note credentials are a text/INI file without the extension.

One primary advantage of using AWS credential file to store S3 keys is for the default user access security offered by the OS. The credentials file will only be accessible to the user who already has the read/write access to the profile location. This will ensure that when copying the OptimizeRasters package onto another machine, that credentials otherwise recoded in the OptimizeRasters config file are not inadvertently exposed.

**Reading from public AWS buckets**

The following configuration keys need to be left empty to enforce reading from a public AWS bucket.

 <In\_S3\_AWS\_ProfileName></In\_S3\_AWS\_ProfileName>  
 <In\_S3\_ID></In\_S3\_ID>  
 <In\_S3\_Secret></In\_S3\_Secret>

**Overriding AWS credentials**

To The bypass the default AWS standard to manage credentials, the OptimizeRasters config file can be edited to include the necessary information as shown below to have the credentials added to S3 input and S3 output storage respectively.

<In\_S3\_Secret>\_IN\_S3\_SECRET\_KEY\_</In\_S3\_Secret>  
<In\_S3\_Bucket> \_IN\_S3\_BUCKET\_NAME\_</In\_S3\_Bucket>

<Out\_S3\_ID>\_OUT\_S3\_SECRET\_KEY\_</Out\_S3\_ID>  
<Out\_S3\_Secret>\_OUT\_S3\_BUCKET\_NAME\_</Out\_S3\_Secret>

The keys in the config file take the precedence over the AWS standard credential manager.

**AWS credentials usage**

If using the AWS credential manager for S3 bucket authentication, the entries related to AWS profiles need to be updated in the OptimizeRasters parameter file to reference the matching profile names in the AWS credentials file.

For e.g. In the OptimizeRasters config file you can have the AWS manager specific entries updated to match the AWS profile names,

<In\_S3\_AWS\_ProfileName>OptimizeRaster\_S3In</In\_S3\_AWS\_ProfileName>  
<Out\_S3\_AWS\_ProfileName>OptimizeRaster\_S3Out</Out\_S3\_AWS\_ProfileName>

For the above entries to work, AWS profile file (credentials) has to be updated to reflect the profile names in the parameter file as shown below.

[OptimizeRaster\_S3In]  
aws\_access\_key\_id=XXX\_YOUR\_ACCESS\_KEY\_ID\_XXX  
aws\_secret\_access\_key=XXX\_SECRET\_ACCESS\_KEY\_XXX

[OptimizeRaster\_S3Out]  
aws\_access\_key\_id=XXX\_YOUR\_ACCESS\_KEY\_ID\_XXX   
aws\_secret\_access\_key=XXX\_SECRET\_ACCESS\_KEY\_XXX

**Setting parameters to Read and Write from S3**

Various parameters need to be set in the configuration file to enable data to be uploaded to an S3 bucket. For example to upload data to the following http://mydata.s3.amazonaws.com/abc/pqr/t then

The following additional changes must be made to the config file

Out\_S3\_Upload – Need to be defined as True

Out\_S3\_Bucket - Specify S3 bucket name where the data should be upload to. I.e. mydata

Out\_S3\_ParentFolder - Specify the s3 folder where the data is to be uploaded to. IE You need to exclude the bucket name. The output folder path in the above example would be abc/pqr/t

Out\_S3\_ID – Define the access ID which will be used to make a connection

Out\_S3\_Secret – Define the secret key required to access the bucket

Out\_S3\_DeleteAfterUpload – Should be defined as true, as the process will create intermediate data files that should be deleted after uploading.

The command line would then be as follows:

<path to python.exe> <path to optimizerasters.py> -input=<path to input folder> -output=<path to s3 outputfolder> -tempoutput=<path\_to-a\_folder\_on\_localdisk> -cloudupload=true –mode=mrf

**Caution on Case Sensitivity**

Note that file names in S3 bucket are case sensitive. It is therefore very important not to change the case of the file names or their extensions when copying data to S3. References to these files must also maintain the case sensitivity.

**Working with Cloned MRF from S3**

If the rasters have been stored as MRF formatted rasters in S3 then it is recommend to use cloneMRF mode to access them.

In the ‘CloneMRF’ mode, a copy of the input directory is created including all auxiliary files, but excluding MRF data and index files. The MRF files are copied and modified to include appropriate links back to the original index and data files and appropriate cache files locations are defined.

For example if the input MRF files are in http://mydata.s3.amazonaws.com/abc/pqr/t

Following configuration changes are needed:

In\_S3\_Bucket - Name of the input bucket mydata

In\_S3\_ParentFolder - The s3 folder where the data needs to be downloaded. abc/pqr/t

In\_S3\_ID – Define the access ID which will be used to make a connection

In\_S3\_Secret - Required secret key

The command line would then be as follows:

<path to python.exe> <path to optimizerasters.py> -input=<path to s3 input folder> -output=<path to outputfolder> -clouddownload=true –mode=clonemrf

**Using Caching MRF with rasters stored in S3**

If the rasters in a non MRF format e.g.(jpeg, jp200 ) on S3, then the following can be used to access the data within the ArcGIS framework by caching MRF. In the ‘CachingMRF’ mode, a copy of the input directory is created excluding all raster files such as TIF and JP2. These are substituted with CachingMRF files. CachingMRF files point back to the source data and appropriate cache files are defined. This has the advantage of not duplicating data but providing faster access and having requests cached to local machines. Prior to using CachingMRF one must ensure that pyramids exist on the source data. If the source rasters are large (>5000cols) and the data not tiled then there can be a considerable performance degradation.

The same changes to the configuration file as for cloned MRF are required.

The command line would then be as follows:

<path to python.exe> <path to optimizerasters.py> -input=<path to s3 input folder> -output=<path to outputfolder> -clouddownload=true –mode=cachingmrf

The following example shows how to create a Caching MRF file of a Scene on Landsat on AWS.

<path to python.exe> <path to optimizerasters.py> -config = LandatOnAWS.xml -input=xxxxxxxxxx -output=<path to outputfolder> -clouddownload=true –mode=cachingmrf

**Working with Azure**

Similar to working with S3, the values of few configuration parameters in the parameter file must be set to prepare OptimizeRasters to work with the Azure FS from Microsoft. Please note on comments in *Italic*.

 <Out\_Cloud\_Type>azure</Out\_Cloud\_Type>

*Cloud type, valid values for now are [****azure****,****amazon****]. if empty no default, must be set in the configuration parameter file.*

 <Out\_Azure\_ParentFolder>folder/output</Out\_Azure\_ParentFolder>

*Root destination folder where files must be uploaded.*

 <Out\_Azure\_AccountName></Out\_Azure\_AccountName>

*Account name. This is similar to Account ID for S3*

<Out\_Azure\_AccountKey></Out\_Azure\_AccountKey>

*Account key. Similar to the secret key for S3*

 <Out\_Azure\_Container>tiffs</Out\_Azure\_Container>

*Container name. Similar to Bucket name for S3*

 <Out\_Azure\_Access>blob</Out\_Azure\_Access>

*Access type. Similar to <Out\_S3\_ACL> for S3.*

*Value values are [****private****,* ***blob****,* ***container****]*

*private: accessible only to the user*

*blob:  Files within the container are publicly accessible.*

*container: Same as type blob plus container metadata are publicly accessible.*

**Azure usage**

python OptimizeRaster.py -input=c:/date/rasters -clouduploadtype=azure -cloudupload=true -tempoutput=c:/temp/tempoutput

python OptimizeRaster.py -input=c:/date/rasters -clouduploadtype=amazon -cloudupload=true -tempoutput=c:/temp/tempoutput

**Internal Info**

Parallel upload support: Yes. Similar to Amazon/S3 uploads. The upload file size does not matter.

No of parallel upload threads per file: 20

Azure blob type: Block blob

Upload payload size per thread: 4 MB.

**Prerequisites**

Azure module for python.

type pip.exe install azure from {python\_folder}/script to install the required module.

Please note, currently only uploading to Azure FS is supported.

**Using Landsat 8 data provided by Amazon to create caching MRF files**

Amazon provides Landsat 8 data to used/accesses to the users freely as part of their PDS (Public Dataset) program.(see <http://aws.amazon.com/public-data-sets/landsat>) These datasets are in geotiff format, with pyramids build at a factor of 3. Using CahingMRF it is possible to directly use them in within ArcGIS. To access a scene the path row and scene-id are required. A configuration file (OptimizeRasters\_PDScaching.xml)with all required parameters is provided.

Following is a sample command.

c:\Python27\ArcGIS10.3\python.exe c:\Image\_Mgmt\_Workflows\OptimizeRasters\OptimizeRasters.py -config=c:\Image\_Mgmt\_Workflows\OptimizeRasters\**OptimizeRasters\_PDScaching.xml** -input=L8/160/043/LC81600432015109LGN00 -clouddownload=true -output=c:\temp\ landsatpdsdata\L8\160\043\LC81600432015109LGN00

**Creating CachingMRF files from rasters stored on a network attached storage**

CachingMRF can be used to speed up access of data to slower network attached storage, by creating CachingMRF files of the source data on a fast local (or SSD based) drives.

The following command line can be used:

<path to python.exe> <path to optimizerasters.py> -input=<path to source data> -output=<path to fast disk> -mode=cachingmrf

**Using Split MRF with rasters stored on a network attached storage**

MRF files can be used also without the caching option, while still optimizing access. This is typical systems with NAS (Network Attached Storage). When ArcGIS access a file it also access the associated metadata files. One way of optimizing the access it to ensure the metadata files and SpitMRF files are copied to faster local storage, while leaving the large data files on the NAS. In the ‘SplitMRF’ mode, a copy of the input directory is created, but it excludes the MRF data files and instead adds links in the MRF files to point back to them.

The following command line can be used:

<path to python.exe> <path to optimizerasters.py> -input=<path to s3 input folder> -output=<path to outputfolder> -mode=splitmrf

**Cache Management**

When using CloneMRF or CachingMRF the MRF driver creates caches of the rasters tiles and associated indices. The MRF driver will only add to the cache, but not delete the cache for example if the disk gets full. These cache files can grow to become very large and care should be taken to ensure they are correctly managed.

By default the caches will be saved with the extension .mrf\_cache in the same directory as the MRF files. It is often advantageous to define the cache to be stored in a separate location so that the files can be easily deleted. The MRF driver has been designed to be robust to the deletion of these files. If an mrf\_cache is found to be missing the driver will fetch new tiles and start recreating the files. The location to store the mrf\_cache is defined using the –cache option in the command line. Typically one does not want all the cache files in a single directory. As the cache files have the same name as the source raster, it is possible for errors to occur if two files have the same name. It is therefore recommended to have cache follow a similar directory naming as the source data. The MRF driver will attempt to create the directory structure specified for the location of the MRF files. Best practice is to define a drive or directory specifically for the cache. It is then easy to delete the files in this directory and subdirectories when additional space is required.

~~There are cases where it is not advantageous to have cache files. An example of this would be the generation of tilecache, where all the imagery from the source will be read only once. A simple way to achieve this is to point cache location to a temporary directory that can be renamed. If the directory does not exist then the cache is not created. It is therefore possible to turn off caching by renaming a directory.~~

~~By default the MRF driver will not create directories if they do not exist. There is through an exception. If the location for the cache starts with the following from z:\MRFcache (where z: is any drive letter), then the driver will create the specified directory structure if it does not exist. It is therefore recommended to the cache in a directory with this name.~~

OptimizeRasters contains some tools to assist in such MRF cache management.

The XXXX tool can be used to clear cache based on amount of memory required.

Usage is …..

……………..

It is recommended to have this program run at a regular interval using Window Scheduler.

**Special Considerations of ArcGIS Server**

The MRF driver has been designed to enable multiple processes to write the same cache simultaneously without corrupting the cache.

When using ArcGIS for Desktop and Server on the same machine some special considerations are required. By default if ArcGIS Server creates a file then ArcGIS for desktop which is logged in as a different users is not able to read the file. This can cause corrupting in that some files cannot be read.

To resolve this problem it is necessary to …………..

-------------- End of Document --------------