GBSAR-Proc
Documentation

James Elgy
Daniel André

Cranfield University
Centre for Electronic Warfare, Information, and Cyber

August 20, 2019

Contents

Contact: James.D.Elgy@Cranfield.ac.uk

1 Overview

This package, herein referred to as `GBSAR-Proc`, is a Python package designed to load and process data gathered from Cranfield University's ground based Synthetic Aperture Radar (SAR) system. Included in the package are a series of classes designed to manipulate raw data, process it into range profiles and finally use the Backprojection Algorithm to plot high quality near-field SAR images.

2 Motivation

In the case of volumetric SAR, a three-dimensional volume is formed, as oppose to more common two-dimensional planar images. As a consequence, significantly more computational work needs to be performed.

The addition of an extra dimension to the image formation process is, at least in Backprojection, a trivial change from a software perspective. However, the additional number of elements can cause the image formation to have a duration orders of magnitude longer. To this end, parallel execution is a practical necessity. In the Matlab programming environment, parallel processing is relatively easy with regards to the CPU, unfortunately, the GPU equivalent is not straight forward.

In Matlab, GPU programming is achieved via the `gpuArray` data type. Many functions in Matlab, although not all, are designed to accept `gpuArray` objects as input arguments, and as such operate on a graphics card. For in-built functions, such as `fft` and `exp`, this is sufficient. However, for more complex and/or custom functions, such as Backprojecton, this approach begins to become cumbersome and inefficient.

The prevailing advice from Mathworks is to either use `arrayfun` to execute a single input function over multiple threads, or to write the program in C and compile the code as a `mex` function. However, due to time constraints, a Python based implementation using the well known scientific NumPy [1] and Numba [2] packages has been written instead. The standard Python interpreter, CPython, has much of its underlying code written in C, the difference in execution time between the two languages should therefore be minimal.

3 Requirements

The `GBSAR-Proc` Python package has few rigid requirements. To make use of the core functionality NumPy, SciPy, and Numba are required. To make use of less important functionality, such as plotting, Matplotlib and Mayavi are needed.

The Matlab function `PythonBP` and the Python module `MSAR` are included as part of the `GBSAR-Proc` software. These custom functions are designed to be called from Matlab. To make use of either one, a 64bit version of Python 3.6 must be installed along with the packages NumPy and Numba, both of which are required. In addition, one should also obtain the Parallel Computing Toolbox for Matlab.

4 Workflow

4.1 The `Params` Class

As part of the image formation process, it is necessary to define some parameters ahead of time. For example, the size of the resultant image or how many pixels to consider. These attributes are contained within the `Params` class.

The `Params` class contains the following attributes:

1. `SC`

   The `SC` attribute is a NumPy array containing the cartesian coordinates of the scene centre.

2. `Lcr`, `Lr`, and `Lel`

   These attributes are the extent in metres of the imaging window along the X, Y, and Z axes respectively.

3. `CRDensity`, `RDensity`, and `ZDensity`

   These are the pixel spacing in metres for the X, Y, and Z dimensions. Combined with the window extent, the total number of pixels can be calculated.

4. `DynamicRange`

   This value, in dB, defines intensity threshold for any plotted images. I.e. a value of 30 will give a intensity range of $max(Im) \geq Im \geq max(Im) - 30$.

5. `Plot3D`

   This boolean controls whether the SAR image formation operates in a volumetric or planar modality. Setting the value to False will cause the image formation and the plotting routines to generate a two-dimensional image. Setting the value to True will cause the image formation to generate three-dimensional image.

6. `UpsampleFactor`

   This option controls how much the range profiles are up-sampled by during the formation process.

7. `CableLength`

   The cable length (m) defines a range by which the range profiles can be uniformly shifted.

8. `GeometryIndex`

   This attribute defines the ground-truth positions of the antennas. For example, setting the attribute to 0 generates a monostatic modality, whereas setting it to 3 generates a fixed receiver and a moving transmitter. This is done via the `geometrygrabber` switch.

9. `RadarHeight`

   The physical height of the SAR system in metres. It sets the origin of the coordinate frame at $O = [0, 0, H]$.

10. `RotationAngle`

    The angle (deg) of the SAR system with respect to the X axis. For example, setting the value to 11 degrees will place the system at 11 degrees with respect to the scene.

It is possible to assign each of these attributes individually within a Python environment, however it is generally easier and more consistant if the actual `Params.py` file is directly edited.

4.2 Data Loading

The `Data` class is the main repository for the manipulation and loading of the raw phase history data. The loading of the data is handled by the `Data.LoadFile()` method. Once this is done, further corrections to the data can be applied.

1. `Data.LoadFile()`

   This function loads the raw data from either a .scn or .cal file. The function works by searching for specific tags within the file. For example, it will look for 'Positions' before reading the line and grabbing the value. It is worth noting that there is no check for units. I.e. the function assumes that all units of distance are specified in mm before converting them to m.

2. `Data.SubtractMean()`

   This function subtracts the mean frequency response from each pulse of data. It is useful for subtracting constant signatures, e.g. DC coupling, from the phase history.

3. `Data.SubtractCL(CableLength)`

   This function applies a phase ramp to `Data.RawData` such that the time domain representation is shifted by `CableLength` (m).

4. `Data.GetGeometry(Params)`

   This function is designed to calculate and return the cartesian coordinates of a bistatic pair of antennas. The function queries a switch to obtain ground-truth measurements for a fixed antenna with respect to the .scn file, this is then used to generate `Ant1` and `Ant2`, both of which are returned. To add a ground-truth measurement, simply add to the `geometrygrabber` switch. This commented switch is located at the bottom of `Data.py`.

5. `Data.CorrectSC(Params)`

   This function applies a phase ramp to the raw data such that the ranges to the targets are with respect to the scene centre coordinate, rather than the antennas.

4.2.1 File Format

The format for .scn files should be as follows:

1. The file should have the extension .scn for SAR collections and the extension .cal for single pulse calibration measurements.
2. The file should have the following parameters in order:
   (a) 'Positions'
   (b) "HorizontalAperture'
   (c) 'HorizontalStartPosition'
   (d) 'HorizontalIncrement'
   (e) VerticalAperture'
   (f) 'VerticalStartPosition'
   (g) 'VerticalIncrement'
   (h) 'StartFrequency'
   (i) 'StopFrequency'
   (j) 'NumberOfPoints'
3. The raw phase history data should be formatted in two columns (Real Imag) and labeled by a header detailing which pulse. E.g. [Reading1of176].

The `Data.LoadFile` function is designed specifically for this format and will not work if it is deviated from.

A specimen file is attached in Appendix A.

4.2.2 Windowing

Within the `Data` class there is the option to apply window functions over the top of the raw data. These windows are generated in the `Window` class using the `scipy.signal` module. Window is applied via:

1. `Data.ApplyWindow(WinType, Dims=1)`

   By default this function multiplies the window specified in `WinType` and the raw data, `Data.RawData`, replacing the dataset in the process. The variable `Dims` specifies how many dimensions to apply the window. E.g. `Dims=2` will apply the window in both range and cross range.

2. `Data.FormDataCube()`

   This function is called by `Data.ApplyWindow` to reshape would be 2D SAR collections into a cube. This makes it significantly easier to apply windows.

3. `Data.UnFormDataCube()`

   This function does the opposite of `Data.FormDataCube`. It takes a 3D representation of the raw data and converts it back into the 2D representation.

The range of windows that can be applied are defined in the `scipy.signal.windows` module.

4.3 Range Profile Formation

The range profile formation aspect of the `GBSAR-Proc` package is handled in the `RangeProfiles` class. It requires that the `Data` and `Params` classes be passed to it during its construction.

Within the class, the main methods are:

1. `RangeProfiles.FormRangeProfiles()`

   This function converts the raw data into a time domain representation via a Fourier Transform. The complex phase histroy data is zero padded to a length defined by `Params.UpsampleFactor` and circshifted such that the centre frequency is the first element in the array..

2. `RangeProfiles.PlotRangeProfiles()`

   This is simply a function to plot the range profiles as a 2D image of range verses pulse number.

3. `RangeProfiles.RangeGate(Min, Max)`

   This function crops the range profiles to a specific range extent. Any value with a range less than `Min` or greater that `max` is scrapped. The result of this function is smaller range extent and subsequently, a smaller dataset.

4.4 Backprojection

For a single pulse of data, the Backprojection algorithm calculates the bistatic range from antennas `A1` and `A2` to the pixel grid. The complex range profile data, `RP`, is then interpolated onto this range to produce a projection of the range profile onto the imaging surface. In SAR this is repeated for each antenna position.

Within the `SAR` class, there are two methods of performing this Backprojection operation:

1. `SAR.BackProjection_CPU()`.

   This method runs exclusively on the CPU and returns `SAR.ComplexImage`.

2. `SAR.BackProjection_GPU()`

   This method exports the main work to the GPU, returning `SAR.ComplexImage`. Consequently, it is significantly more complex and requires the CUDA Toolkit be pre-installed on the computer.

4.4.1 CUDA Backprojection

CUDA is a NVIDIA framework for exporting computational load onto a GPU in order to leverage the high core count and multi-threading optimisations built into modern graphics cards. While CUDA is developed and designed for C, it is possible to use Numba to compile python code into a CUDA compatible format. The general format for doing so is as follows:

Listing 1: Simple CUDA Example

```
from numba import cuda

@cuda.jit()
def CudaKernel(args, var1, var2)
    args = var1 + var2
```

The `cuda.jit` function decorator performs a just-in-time compilation of the `CudaKernel` function into GPU compatible machine code. Note that CUDA functions cannot return variables, therefore the desired output, `args`, must be provided as an input, then overwritten. While this example is extremely simple, it follows the basic guidelines laid out on the Numba website [3].

Since all the variables, in this example `args`, `var1`, and `var2`, must be transferred to the device, it is in the users best interest to transfer as little data as possible, since doing so takes a significant amount of time. With this in mind, the CUDA implementation of Backprojection aims to do all the required steps with as little data transfer as possible.

The Host Function transfers the minimum required variables onto the GPU, this includes the pixel grid, `Pixel` and an array of zeros that will eventually become the final image. The final job of the Host Function, before calling the kernel, is to set up the thread hierarchy alluded to earlier. This is done via the `Initalise` function.

Listing 2: Initalise Function

```
def Initalise(imsize):
    # Set the number of threads in a block
    threadsperblock = 32
    # Calculate the number of thread blocks in the grid
    blockspergrid = (imsize+(threadsperblock-1))//threadsperblock

    return (threadsperblock, blockspergrid)
```

The variables `threadsperblock` and `blockspergrid` define the so-called thread hierarchy and must be provided to the kernel when it is called.

With that said, the Backprojection function itself operates as follows: First the imaging grid is transferred onto the GPU. This could be segmented, however keeping it as one contiguous array helps to keep the code simple. Next the amount of free memory is queried, and the range profile data segmented appropriately. Finally, each segment of the range profile data is sent to the GPU and the Backprojection kernel is executed. The data is transferred in this way to minimise the number of separate transfers than need to be executed, and thus improve the execution time

4.5 Plotting

Within the `SAR` class, there are three functions related to plotting and saving the data.

1. `SAR.PlotSAR()`

   This function does not return any variable, however it does produce a plot of the Back-projected image. 2D planar images require the Matplotlib package, whereas volumetric images are plotted using Mayavi.

2. `SAR.PlotPhase()`

   A function to plot the spatial frequency support of either 2D or 3D Backprojected images. The spatial frequency is obtained by performing a N-dimensional Fourier Transform of the complex image data. As with the `SAR.PlotSAR` function, planar images are plotted by Matplotlib and 3D images by Mayavi.

3. `SAR.SaveVolumeImage(Dir)`

   This function forms and saves a volumetric SAR image as a multipage tiff file, specified by the string `Dir`.

All of these functions operate on a dB scale, with a colourmap specified by the peak intensity and the dynamic range specified in the attribute `Params.DynamicRange`.

5 Matlab Integration

In Matlab, there is functionality for importing and executing Python code. This is done by prefacing python functions with `py`. E.g. the code `py.print('foo')` will run a Python print command. It is important to know that Matlab does not actually compile the Python code itself, rather it delegates that task to a Python interpreter already installed on the computer. to that end, one must specify the location of the Python executable using the `pyversion` command.

In Matlab, the custom function `PythonBP` takes a set of input variables from Matlab and uses them as arguments for the `MSAR` Python code. This is a function that is callable from Matlab and has produced a significant speed improvement when compared to the equivalent Matlab parallel CPU Backprojector.

However, since there is no convenient type matching between the two languages (e.g. a Matlab double is translated to an int in Python) the `MSAR` module has been modified from the original code to include these type conversions. Furthermore, Matlab cannot transfer complex numbers, hence the real and imaginary components of the range profiles are passed as separate arguments. This is inelegant, however it is a working solution

5.1 Installation

To associate a Python installation with Matlab, use the command `pyversion`. By default, Python installs itself at C:\Users\...\appdata\Local\Programs\Python \Python36, ergo, to associate Python with Matlab use:

```
>>>pyversion('C:\Users\...\appdata\Local\Programs\Python\Python36\
python.exe')
```

This command is something that could easily be inserted into a `startup.m` file for convenience in the future.

Note: Matlab does not officially support Python environments such as those used by Anaconda. For this reason they recommend that packages are obtained directly through `pip` rather than a package manager.

Finally, to make use of the GPU, the CUDA Toolkit needs to be associated with the software. This is done by setting the, currently hardcoded, environment variable `CUDA_Home` to the location of the CUDA installation.

5.2 Usage

Both the `PythonBP` function and the `MSAR` module must be placed in the current Matlab directory or in the Python search path.

Unlike the Backprojectors in the `SAR` class, the `PythonBP` function requires that the imaging grid be formed ahead of time, there is also no option to run this code exclusively on the CPU. To use, simply call the `PythonBP` function like any other Matlab function.

1. `PythonBP(xm, ym, zm, RPvec, Ant1, Ant2, Freq, RangeSc, RangeProfile_Real, RangeProfile_Imag)`

   This is the main function needed to call the `MSAR` module. It adds the current directory to the Python search path and imports the module. Secondly, it calls `MSAR.BP_GPU` to run the Backprojection. Finally it reformats the output dataset into a Matlab complex double.

6 Benchmarking

As stated, the GPU version of Backprojection is significantly faster than both single core and parallel execution. However, this only holds true for images with greater than 1 MPixel, otherwise parallel CPU execution is quicker [1].

Figure 1 illustrates this point by showing how the execution time scales with pixel count [2].

[1] These comparisons are generated using an NVIDIA Quadro RTX6000 and 2 Intel Xeon Gold 5118 12 core CPUs

[2] These comparisons are for the `PythonBP` function when compared to single core and parallel execution in Matlab.

There is some time associated with transferring data onto and off of the GPU. Due to this, it is recommended that if the GPU is to be used then the range profiles be made as small as possible, via the `RangeProfiles.RangeGate` function. Doing so, reduces the transfer overhead and accelerates the Backprojection.

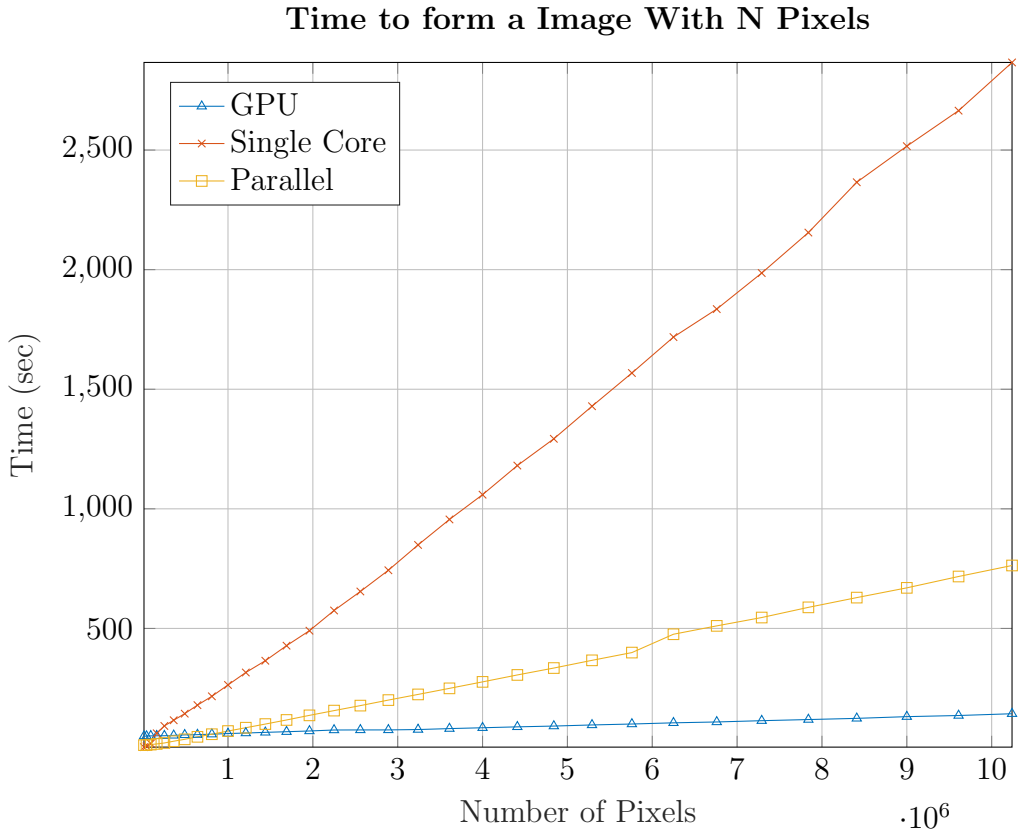

Figure 1: Time comparisons between single core, parallel, and GPU executions of Backprojection. The time increases linearly with pixel count.

7 Considerations and Future Work

The art of utilising a GPU for SAR Backprojection is not new. It has been around for over a decade now. As such there have been a slew of papers published on the topic. The implementation chosen for this Backprojector was chosen due to its simplicity, making it an ideal first foray into using a graphics card for heavy work. The approach taken here follows a scheme described by Hu et al [4]. Small scale optimisations can be made, however the speedup from said optimisations is likely to be minor. A substantial improvement was proposed by Chapman et al in 2012 [5] whereby the image is segmented into blocks. once each block is finished it transfers the data and acquires a new set independent of the other blocks.

In summary, we have achieved a substantial speedup when compared to a Matlab implemen-

tation of Backprojection through leveraging the parallelisable nature of the Backprojection Algorithm. Ideally, the transfer overhead moving the data onto and off of the GPU would be reduced, as this is a significant bottleneck, however rewriting the code would take a large amount of time and minor optimisations are unlikely to yield substantial improvements.

8 References

[1] Travis Oliphant. NumPy: A guide to NumPy. USA: Trelgol Publishing, 2006–. [Online; accessed ¡14/08/2019¿].

[2] Siu Kwan Lam, Antoine Pitrou, and Stanley Seibert. Numba: A llvm-based python jit compiler. In *Proceedings of the Second Workshop on the LLVM Compiler Infrastructure in HPC*, LLVM '15, pages 7:1–7:6, New York, NY, USA, 2015. ACM.

[3] Writing cuda kernels. `https://numba.pydata.org/numba-doc/dev/cuda/kernels.html#introduction`. [Online; accessed ¡14/08/2019¿].

[4] Kebin Hu, Xiaoling Zhang, Wenjun Wu, Jun Shi, and Shunjun Wei. Three GPU-based parallel schemes for SAR back projection imaging algorithm. *Proceedings - 17th IEEE International Conference on Computational Science and Engineering, CSE 2014, Jointly with 13th IEEE International Conference on Ubiquitous Computing and Communications, IUCC 2014, 13th International Symposium on Pervasive Systems,*, pages 324–328, 2014.

[5] William Chapman, Sanjay Ranka, Sartaj Sahni, Mark Schmalz, Linda J. Moore, Uttam Majumder, and Bracy Elton. Backprojection algorithms for multicore and GPU architectures, 2012.

9 Appendix A - Specimen File Format

Attached is a subset of a generic .scn data file:

```
[EMScan]
Origin = Radar Scanner Controller v 2017.2.0 (Build 22)
Type = EMS v 1.02s
File = C:\Users\Keith\Documents\EMScan\EMScan190322_081_0031.scn
Date = 2019-03-22
Time = 17:10:47.327
Polarisation = 'HH' of 'HH'
Positions = 176
PositionCoordinates = (X,Y)
ScanType = Horizontal Scan

[Comment]
No comment
```

```
14
15 [Settings]
16 ScanType = 1
17 HorizontalAperture = 3500.0
18 HorizontalStartPosition = 0.0
19 HorizontalIncrement = 20.0
20 HorizontalReturnToStart = .false.
21 HorizontalHomeBeforeScan = .true.
22 HorizontalHomeDuringScan = .true.
23 HorizontalHomeBeforePolarisation = .false.
24 HorizontalDisableWhenDone = .true.
25 Aperture = 3500.0
26 StartPosition = 0.0
27 Increment = 20.0
28 DisableBoomAxisWhenDone = .true.
29 VerticalAperture = 1500.0
30 VerticalStartPosition = 0.0
31 VerticalIncrement = 25.0
32 VerticalReturnToStart = .false.
33 VerticalHomeBeforeScan = .false.
34 VerticalHomeDuringScan = .false.
35 VerticalHomeBeforePolarisation = .false.
36 VerticalDisableWhenDone = .true.
37 TransverseReturnToStart = .false.
38 TransverseDisableWhenDone = .true.
39 AllowPositionalScan = .true.
40 ReturnToStart = ''
41 HomeBeforeScan = '.false.'
42 HomeBeforePolarisation = ''
43 LevelBeforeScan = .false.
44 LevelBeforePolarisation = .false.
45 WobbleDelay = 0.0
46 VVPolarisation = .false.
47 VHPolarisation = .false.
48 HHPolarisation = .true.
49 HVPolarisation = .false.
50 NumberOfScans = 1
51 TimeBetweenScans = 3910
52 TimeBetweenScansIsDelay = .true.
53 TimeBeforeDisablingAxes = 60
54
55 HorizontalScanSpeed = 40.0
56 HorizontalScanAcceleration = 20.0
57 HorizontalScanDeceleration = 20.0
58 HorizontalHomeSpeed = 40.0
59 HorizontalHomeAcceleration = 20.0
60 HorizontalHomeDeceleration = 20.0
```

```
HorizontalReturnSpeed = 40.0
HorizontalReturnAcceleration = 20.0
HorizontalReturnDeceleration = 20.0

[Units]
HorizontalPosition = mm
HorizontalSpeed = mm/s
HorizontalAcceleration = mm/s2

[PNA]
IdentificationString = Agilent Technologies,N5230A,MY46400279
PNAPowerOnDelay = 0.0

[NormalScanPNAData]
StartFrequency = 1.0E9
StopFrequency = 6.0E9
Averaging = .false.
AveragingFactor = 1
AverageMode = 1
DwellTime = 0.0
IFBandwidth = 3.0E3
NumberOfPoints = 2801
SweepDelay = 0.0
SweepTime = 0.90878445
MeasurementSweepTime = 0.90878445
TestPortPower = 9.0
PowerSlope = 0.0
PowerSlopeState = .false.
MadeChanges = .true.

[Home]
Horizontal axis, home offset = -0.01 mm

[Reading1of176]
0.00 1500.00
'HH'
888.88 0.0
17:10:47.420
17:10:48.372
    2801
  1.50109399E-02 -2.92815547E-02
 -1.63930859E-02 -3.17203850E-02
 -3.58894803E-02 -2.50003883E-03
 -2.02626251E-02  2.65208352E-02
  7.04702875E-03  3.12650539E-02
  3.08915116E-02  1.19293090E-02
  2.48109307E-02 -1.73169635E-02
```

```
 2.10503303E-03 -2.85631847E-02
-2.05997489E-02 -1.93624496E-02
```