PML | Publishing

Improving the performance of National Centre for Earth Observation (NCEO) code using GPUs

Nicola Martin, Daniel Clewley, Steve Groom

January 2023

PML | Plymouth Marine Laboratory

Executive Summary

- The aim of this study was to investigate the advantages of different tools designed for running code on Graphics Processing Units (GPUs) and specify the types of code best suited for GPU acceleration.
- Three examples of code were obtained from NCEO scientists for three distinct applications and ported to GPUs on the Natural Environment Research Council (NERC) Earth Observation Data Acquisition and Analysis Service (NEODAAS) MAGEO computing cluster.
- Comparisons were made between the time taken to run the code on the original Central Processing Unit (CPU) and the MAGEO CPU and GPU.
- Accelerations of between x30 and x1800 were achieved: more details are provided below.
- In terms of energy saving this relates to an estimated 93.9% to 99.8% reduction in electricity usage.
- The study highlighted the value of expertise in GPUs and coding such as provided by NEODAAS

Outputs of the study

In addition to the speed improvements made to the specific code examples used, key outputs of this study are:

- This report, which contains details of lessons learned and recommendations and provides a useful starting point for future projects wishing to utilise GPUs for processing large Earth Observation datasets.
- Well documented code examples in the form of Jupyter notebooks, demonstrating the methods used to port the code to run on GPUs. The notebooks can be used as a tool for understanding these methods, which can be applied to similar problems in a larger codebase.
- Training material in the form of a Jupyter notebook, aimed at new users as a gentle introduction to the methods used during the study. Available from: https://github.com/NEODAAS/Porting-Python-code-to-GPUs
- Knowledge transfer to the NCEO researchers involved in the study, who are already starting to utilise some of the methods to improve the efficiency of their code

All the new code in this study was developed using MAGEOHub[1] which provides users with web-based access to the NEODAAS MAGEO (MAssive GPU Cluster for Earth Observation) via Jupyter[2] notebooks. The examples considered make use of:

- Numba[3], an open source just-in-time (JIT) compiler for speeding up Python code;
- CuPy[4], an open-source array library for GPU-accelerated computing with Python;
- *gpu_hist,* the GPU algorithm from XGBoost[5], an optimized distributed gradient boosting library.

Code was developed using a containerized approach to ensure portability to other GPU clusters. Training materials are available via the NEODAAS GitHub repository[6].

Lessons learned from the NCEO code examples

Three examples were considered in this study. One looked at reprojecting a Landsat 8 scene, a second was based on self-organizing maps (SOM), and the final example was code for training a model using XGBoost. Full details of the approaches used for porting the examples to GPUs can be found in the appendix. The three examples helped to identify the types of problems that will work well on GPUs as well as a problem that was less well suited to GPU acceleration. Due to the short nature of the study, finding researchers at short notice with time available to optimize code for GPUs was difficult at the start of the project as many were unsure what code was suitable and the improvements they could expect. However, explaining the problems GPUs work best for, and providing real-world examples of typical performance gains, led to more requests.

The key finding was **GPUs work best for repetitive and highly parallel tasks.** The initial SOM code was not easily parallelizable, however, the alternative algorithm allowed for the processing within each mini batch to run in parallel, making it ideally suited to run on a GPU. In all three examples, the **changes to make the code suitable for running on GPUs also led to improved performance on the CPU**. When writing new code, it would be useful to structure it in such a way that it is easy to switch between the CPU and GPUs. This would likely lead to improved performance on the CPU with the option to run the code even faster when GPUs are available.

When porting code to run on GPUs, it is possible to make use of existing libraries such as XGBoost that already have GPU support, or to replace functions using popular libraries such as NumPy or Pandas[7] with GPU enabled alternatives that are designed to provide most of the same functionality, e.g. CuPy or CuDF[8]. However, for more specialized code where neither of these two approaches can be applied, it is likely that the code will not be easily ported to run on GPUs. For example, the library functions being used may need to be re-written in pure Python to make use of tools like Numba for porting your code. This was the case for the reprojection example, and here a pure Python alternative was available. Depending on the library it can be possible to extract the function from the source code and make a few minor changes to remove external dependencies.

This study has identified the following key questions to help identify if a piece of code is suitable for GPU acceleration:

- Can the code be run in parallel? Has it been vectorized? Does the code contain loops?
- Will the code need to be restructured to allow it to be parallelized?
- How often will data need to be copied to and from the GPU?
- Does the code use any libraries with existing GPU support?
- Are there any existing GPU libraries designed to carry out the processing used in the code?

If the code can be run in parallel and requires a lot of mathematical operations, it is likely that it will run faster on a GPU than the CPU. For pure Python code, Numba can easily be tested to see whether it can speed up the code. If the code already uses NumPy, a useful starting point might be to try replacing NumPy with CuPy.

In the right circumstances, it can be extremely quick and easy to get code running on GPUs. However, often code will need a few modifications to make it efficient at running on a GPU. To benefit from the many available tools for porting software to GPUs, researchers could find basic

training materials useful. To get started using the tools it is helpful to see them being applied to simple examples, understand the types of problem that they can be used for and have an idea of when code needs to be modified to work with these tools.

Electricity Savings

The electricity used by the NCEO code examples was estimated based on their run time and the maximum power for the MAGEO processors. This was compared with the estimated electricity used when running on a GPU, calculated with the new run times and maximum power for the GPU and CPU. These estimates are shown in Table 1, along with the percentage reduction in energy usage based on these figures.

	Electricity used by CPU running original code (kWh)	Electricity used by CPU+GPU running modified code (kWh)	Electricity usage for CPU+GPU as a proportion of electricity usage for CPU only	Percentage reduction
Reprojection	0.032	0.00013	0.4%	99.6%
SOM	0.0051	7.9×10^{-5}	0.2%	99.8%
XGBoost	1.2	0.075	6.1%	93.9%

Table 1 Comparison of electricity usage for original CPU and GPU optimised code

These figures are based on only the CPU and CPU+GPU power usage, not other components or data centre cooling and it is assumed both were drawing the maximum power.

Training materials

As part of this project a Jupyter notebook, *Getting started with Numba and CuPy*, was created as a training tool based on some of the lessons learned from the NCEO code examples. The notebook provides a gentle introduction to Numba and CuPy using some simple examples to demonstrate the basic methods. It also shows how the methods might be applied to real life problems and how Numba and CuPy can be used together. Timings are compared for CPU and GPU to demonstrate the benefits of the methods. Recommended resources for further learning are included at the end of the notebook.

This resource may be used by researchers looking for an introduction to Numba and CuPy. It has been designed to run on MAGEOHub. It can be run on any machine with access to a CUDA-enabled NVIDIA GPU but may need some additional setting up outside of MAGEO.

Suggestions for future work

As this was only a small scoping study, relatively small and self-contained examples were chosen. Porting larger more complicated codebases to running on GPUs would be a logical next step, and the output of this project can help identify those projects most likely to benefit and the steps required. The examples in this study focused on Python software due to the libraries available for GPU

acceleration and preference of the language by many researchers in NCEO for smaller pieces of code. Many of the more complicated models used in NCEO such as the radiative-transfer (RT) models, or typical marine physical-biogeochemical models, and data assimilation code, are written in Fortran. Exploring methods for utilizing GPUs in Fortran would benefit these.

Richard Siddans (Rutherford Appleton Laboratory) suggested trying to port multiple-scattering code to run on GPUs, which can easily be the limiting step in analysing atmospheric remote sensing data. Moving large RT code to GPUs may be a challenge so a first step might be to focus on solving the band matrix inversion at the core of the scattering code. The cuBLAS[9] library provides a GPU-accelerated implementation of the basic linear algebra subroutines (BLAS). Simple performance tests could also be done using the Python bindings for cuBLAS.

Using the work already completed during this study, researchers at NCEO could apply the methods to similar pieces of code. Work has already started in applying the Numba tools to the larger Landsat imagery codebase. A key part of porting code to GPUs is being able to identify code that is suitable for GPU acceleration and understanding when modifications can be made to make the code suitable for running on GPUs. The “getting started” guide along with the example notebooks may be enough to instruct how the tools can be used in similar code. Where the code is more complex, further assistance or training may be needed to apply the methods in practice.

Conclusions

During this study, three NCEO code examples were explored to determine the potential of using GPUs to speed up run times. It was shown that the reprojection example was suitable for running on GPUs and significant performance gains were achieved after modifying the code. The code ran approximately 700 times faster than the original code once ported to the GPU and used an estimated 99.6% less electricity (based on CPU and CPU+GPU power usage only). The SOM example initially showed little potential for acceleration based on the size of the applications that would use this code. By understanding the types of problem best suited to GPUs, an alternative batch-based approach was identified. After restructuring the code, it became a good fit for GPU acceleration, running about 1800 times faster than the original code (for 8x8 grid and 10 iterations) and reducing electricity usage by an estimated 99.8%. The XGBoost example showed potential for running the NCEO machine learning code on GPUs and demonstrated how libraries with GPU support could allow for a smooth switch between running on the CPU or GPU. The training example ran over 30 times faster on the GPU than the original code, with an estimated 93.9% reduction in electricity usage.

All the methods used in the study may be applied to similar parallelizable code to benefit from the types of improvement shown in the examples. The introductory guide to Numba and CuPy Jupyter notebook can also help with applying the tools to other pieces of code. As a result of this study NCEO researchers at the University of Reading are already benefitting from the Numba based methods, where these have been applied to the Landsat imagery processing chain.

Though only a small scoping study, this project has been successful in demonstrating the benefits of using GPUs to accelerate code, and the importance of skilled analysts in advising on how to improve existing code, but this has just scratched the surface of what is possible. Suggestions for future work in this area have been presented, which could lead to significant performance improvements to help

NCEO process increasingly larger volumes of data while moving towards UKRI's NetZero aim for Digital Research Infrastructure.

References

1. https://mageohub.neodaas.ac.uk/

2. https://jupyter.org/index.html

3. https://numba.pydata.org/

4. https://cupy.dev/

5. https://xgboost.readthedocs.io/en/stable/gpu/

6. https://github.com/NEODAAS/Porting-Python-code-to-GPUs

7. https://pandas.pydata.org/

8. https://docs.rapids.ai/api/cudf/stable/

9. https://docs.nvidia.com/cuda/cublas/index.html

Appendix

The NCEO code examples

Example 1: Code for reprojecting a Landsat 8 scene

The first example was provided by Niall McCarroll (University of Reading) who identified reprojection code that could benefit from acceleration. A short Python program, reproject_scene.py, was extracted from a processing chain for Landsat 8 data. The program has two main steps:

1. Apply a geotransform to compute the i,j coordinates of each pixel centre in the scene to x,y locations in the coordinate reference system used by Landsat 8 (UTM).
2. Reproject from the UTM coordinates to WGS84 (lat/lon).

The first step is pure Python. The second step includes C code from within the Proj library. Running the program on the CPU takes around 12 minutes.

The program contains a double for-loop to apply the reprojection to each pixel. This type of problem can be sped up using the JIT compiler Numba, which works best on code that uses NumPy arrays and functions, and loops. Python has a useful feature called decorators, which are used to add functionality to existing code. Using Numba typically involves adding various decorators to existing Python functions, that tell Numba to compile them to machine code. Numba can be used with CPUs and GPUs.

Initial checks showed that most of the processing time was spent on step 1 (roughly 8 minutes). Step 1 was pulled out of the for-loops and re-written as 2 simple functions for calculating the x,y locations. The *numba.vectorize* decorator was applied to the functions. To test this change, the rest of the code remained unchanged. The modified program ran on a GPU in about 3 mins rather than the original 12 minutes. This step actually ran faster on the CPU using Numba (GPU about 1 second, CPU less than 1 second), which is due to the copying of data to and from the device. However, when including step 2, the time taken to copy the data becomes less significant.

Step 2 was a little more complicated. Within the loops a reprojection from UTM to WGS84 was applied to each point with the 'TransformPoint' method. Numba does not understand this method so the decorator could not be simply applied to this step. A pure Python implementation of the transformation was needed. A modified version of the *conversion.py* module in the Python library *utm* was used (https://pypi.org/project/utm/). It was decided that this implementation gave results at the required accuracy for the problem, with the results using this method giving latitudes and longitude values matching the original results to 6 decimal places.

A Numba JIT decorator was applied to the pure Python function for converting UTM to lat/lon so that it would be compiled as machine code before running. Two simple functions were used to call this JITed function and both of these functions were decorated with *numba.vectorize*.

Running both steps together with the new modifications showed a significant speed up using CPU and GPU. With CPU alone the program ran in less than 20 seconds. On a GPU the program ran in approximately 2 seconds.

Further changes were made using CuPy (NumPy for GPU). This simplified some of the copying to and from the GPU. The data types were also updated so that arrays used float32 data instead of float64

data, which runs faster on GPUs. With these small changes the time running on the GPU came down to just over 1 second compared to the original 12 minutes.

If this code is moved into production, the improvements in efficiency are going to be important in minimising the resources used to run the code. Niall has taken the Numba-based improvements to the reprojection code and added them into the larger program that is part of the Landsat imagery processing chain. Researchers using this are already able to benefit from the work. Niall has started to apply the approaches used in the notebook to other bottlenecks in the code and has seen an improvement in timings being reduced from 20 minutes to 3-4 minutes. He has shared the new code with other members of the group in Reading and has suggested that some training using simple examples of Numba would help researchers to benefit from this work further.

This particular problem was very well suited to running on the GPU. The GPU's ability to run the individual pixel calculations in parallel rather than looping over each point, enabled the processing to be completed much more quickly. For any problems that loop over elements in large arrays, the methods used for this example could be applied in the same way. Problems that involve many mathematical operations on each element in the array are likely to see significantly reduced timings when run on a GPU.

Example 2: Python implementation of self-organizing maps

The next example also came from Niall McCarroll, but this time looking at self-organizing maps (SOM). The code provided was a Python script, *som.py*, that uses an example dataset from the NCEO ACSIS project containing sea level anomalies in the N Atlantic averaged by month and cell. Self-organizing maps (SOM) is applied here to try to do data reduction to identify areas which have similar patterns of seasonal variation in sea level anomaly. The code uses NumPy for working with the data arrays (https://numpy.org/). It can work on small maps (say 10 x 10) on CPU but slows down according to the square of the map size, making larger maps impractical.

Summary of algorithm (https://en.wikipedia.org/wiki/Self-organizing_map):

1. Randomize the map's nodes' weight vectors
2. Traverse each input vector in the input data set
    a. Traverse each node in the map
        i. Use the Euclidean distance formula to find the similarity between the input vector and the map's node's weight vector
        ii. Track the node that produces the smallest distance (this node is the best matching unit, BMU)
    b. Update the nodes in the neighbourhood of the BMU (including the BMU itself) by pulling them closer to the input vector
        i. $W_v(s+1) = W_v(s) + \theta(u,v,s) \cdot \alpha(s) \cdot \big(D(t) - W_v(s)\big)$
3. Increase s and repeat from step 2 while $s < \lambda$

Because of the way this algorithm works there are many parts that must run in series since a weight array is updated for each instance using the weights calculated from the previous instance. In this specific example the number of instances is 216941 (lat:401 x lon:541).

Although a loop over the instances is still required, within the loop the BMU must be calculated for each node in the map. This is the part that slows down according to the square of the map size and provides the potential for running this part in parallel on a GPU. Also within the loop is the operation to update the weights. It is applied to all weights in the neighbourhood of the BMU so there is the option to parallelize the loop here, however there are only a few nodes that get updated.

The first approach for running the code on a GPU was to use CuPy as a drop-in replacement for NumPy to see if there was any improvement to the timings. To do this CuPy was imported as 'np' so that all NumPy calls would now be CuPy ones. There were a couple of extra modifications to the code that were required to do this due to subtle differences between the NumPy and CuPy interfaces.

The default settings for testing the script use an 8x8 grid size with 10 iterations, which takes about 2 minutes to run. Ideally with some optimization, it would run for a grid size of 100x100 with 100 iterations. Using CuPy as a straight drop-in replacement for NumPy took nearly half an hour with the default settings! Much slower than the NumPy equivalent. It is not surprising that it took longer than the CPU code as the grid size is still quite small. Table 1 shows the results of a quick comparison using 8x8 vs 100x100 for 1 iteration.

	8x8 grid	**100x100 grid**
NumPy	26 seconds	1 minute 52s
CuPy	4 minutes 20s	5 minutes 27s

Table 1 Comparison of timings for 1 iteration using NumPy and CuPy on an 8x8 and a 100x100 grid

Although the timings of the NumPy solution grow at a faster rate for larger grid size, the CuPy approach still takes much longer to run for the grid sizes up to 100x100. To determine the grid sizes where the GPU would become useful for this SOM algorithm, the small part of the code that calculates the BMU was investigated further. CuPy's *ReductionKernel* class was found to be the fastest way to calculate the sum of squares in this part of the code. Figure 1 summarizes the times taken to run a single BMU calculation as the grid length is increased.

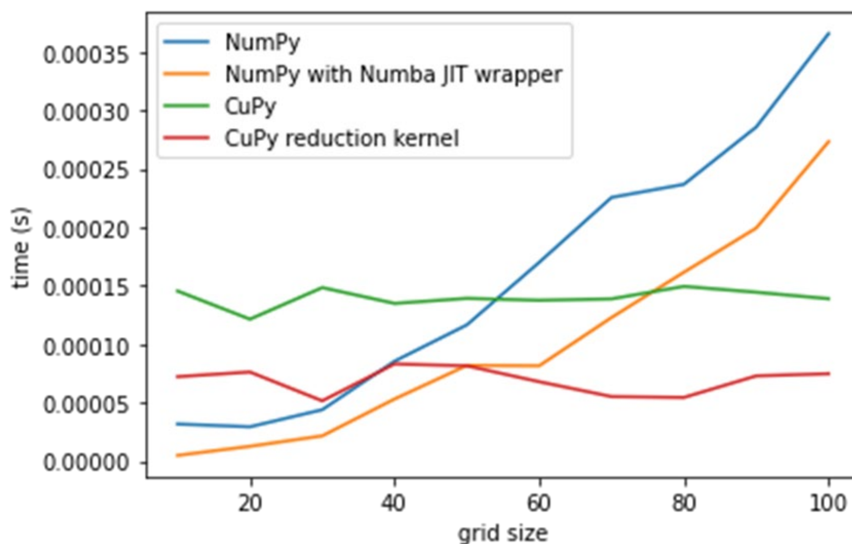

Figure 1 Time taken using NumPy and CuPy to run a single BMU calculation for different grid sizes

The results suggest that using the GPU will be faster for the larger maps (grid size more than about 50x50). However, the weight updates also need to be run on the GPU and it turns out these are very slow on the GPU. Table 2 shows the timings for a single iteration for a selection of grid sizes.

	8x8	**100x100**	**200x200**
Original code	21.8s	117s	419s
CPU Numba JIT	5s	73s	260s
GPU reduction and GPU vectorized network update	180s	181s	184s

Table 2 Comparison of timings for 1 iteration using CPU and GPU solutions for different grid sizes

For grid sizes of around 200x200 using a GPU would be beneficial, but for smaller grid sizes it is likely to be faster running on the CPU instead. Using Numba on the CPU does make this implementation more practical for real world examples.

The algorithm used in this example was not well suited to running on a GPU. The updates needed to the weights array had to be carried out frequently and these had to be run in order since outputs from these updates were used as inputs to the next step. The weight updates performed after each BMU calculation only operated on a small number of points in the array so were not benefitting from the GPU's ability to run many operations in parallel.

There have been some publications in recent years that demonstrate how SOM can be implemented using a batch-based approach, which makes it more suited to running on a GPU, for example in this chapter from the book *High Performance Computing:* Daneshpajouh, Habib & Delisle, Pierre & Boisson, Jean-Charles & Krajecki, Michael & Zakaria, Nordin. (2018). Parallel Batch Self-Organizing Map on Graphics Processing Unit Using CUDA. 10.1007/978-3-319-73353-1_6. Niall wrote an updated version of the Python script that made the code more vectorised using NumPy. In this new version, training instances are batched together and the weights only updated after each "mini" batch. This new version is much more flexible in that it can work with different topologies and neighbourhood masks. The default settings used in the updated script were set with a grid size of 16 x 16 which is suitable for the particular problem. The *minibatch_size* was set to 1000 with an option of passing all the data in a single batch. On the CPU this algorithm was already faster with 10 iterations taking about 40 seconds to run.

The new script was perfectly suited to running on a GPU and could be sped up with minimal effort. Using CuPy as a straight replacement for NumPy enabled the code to run in 460ms, nearly 90 times faster than on the CPU. Making use of the *ReductionKernel* class that showed some success in the original algorithm, reduced the time down to 189ms, over 200 x faster than running with NumPy.

To get the code to run with CuPy there were a few modifications needed as before. There were additional changes needed where CuPy did not have a couple of the methods available that were being used with NumPy. In this case there were alternatives available. It is worth noting that CuPy and its underlying libraries are constantly being updated so it is likely that missing features will be added in the future.

The new algorithm worked better because being in its vectorized form all the instances in a batch could be processed in parallel with the updates to the weights only taking place between the batches.

This example highlighted the fact that there is sometimes a need to rewrite code to make it suitable to run on a GPU. In theory CuPy could be used by anyone wanting to port their code to run on GPUs with a single line of code being changed. However, this example showed that this does not always work without modifications to the code and highlights the need for specialist advice or support. It is therefore useful to understand where there are differences between CuPy and NumPy and where care should be taken. Some basic training could be useful for researchers wanting to get started with CuPy.

Example 3: Training an XGBoost model

The final example was provided by Rob Parker and Cristina Ruiz Villena (University of Leicester). This example took the form of a Jupyter notebook written in Python that loaded a big data frame and trained a model using XGBRegressor. The data frame included the years 2000 to 2020; 2000-2019 was used as training data and 2020 was used as validation data. The training was done using the *exact* tree method. The mean absolute error was used to evaluate the fitted models. The hyper-parameters were set in the notebook based on some earlier testing, including the *max_depth* (maximum depth of a tree) that was set to 15. With these settings the training took about 6.5 hours to run on a single processor. Cross-validation for hyper-parameter tuning would take days/weeks to run or would need to be spread across the HPC cluster.

The *exact* tree method can be very slow and does not support distributed training. Approximated training algorithms are often used instead, with the *exact* method only being used for small datasets. The fastest of these algorithms is the *hist* tree method and there is a GPU implementation of this method called *gpu_hist*. According to the XGBoost documentation "*Most of the time using (gpu)_hist with higher max_bin can achieve similar or even superior accuracy while maintaining good performance*".

Using the same hyper-parameters as the original notebook, the three tree methods were compared. It was found that both the *hist* and *gpu_hist* methods gave equivalent results to the *exact* method when using the *max_depth* of 15. They also ran a lot faster, with the *hist* method running on the CPU taking approximately 45 mins and the *gpu_hist* method running on a GPU taking approximately 12 minutes, compared to the original 6.5 hours run time.

These methods also make any tuning of hyper-parameters more practical. For example, GridSearchCV (https://scikit-learn.org/stable/modules/generated/sklearn.model_selection.GridSearchCV.html) for 7 *max_depth* options and 5 folds with *n_estimators* set to 100 took 137 minutes. Even if n_estimators=500 was used, the expected timings would be around 12 hours rather than the days/weeks required previously.

Values in the range 5-20 for the *max_depth* parameter were compared when training with the *gpu_hist* method. Using a smaller *max_depth* value speeds things up significantly, but the accuracy is not as good. Figure 2 shows how the fit time changes with increasing *max_depth* and how the mean absolute error changes with *max_depth*.

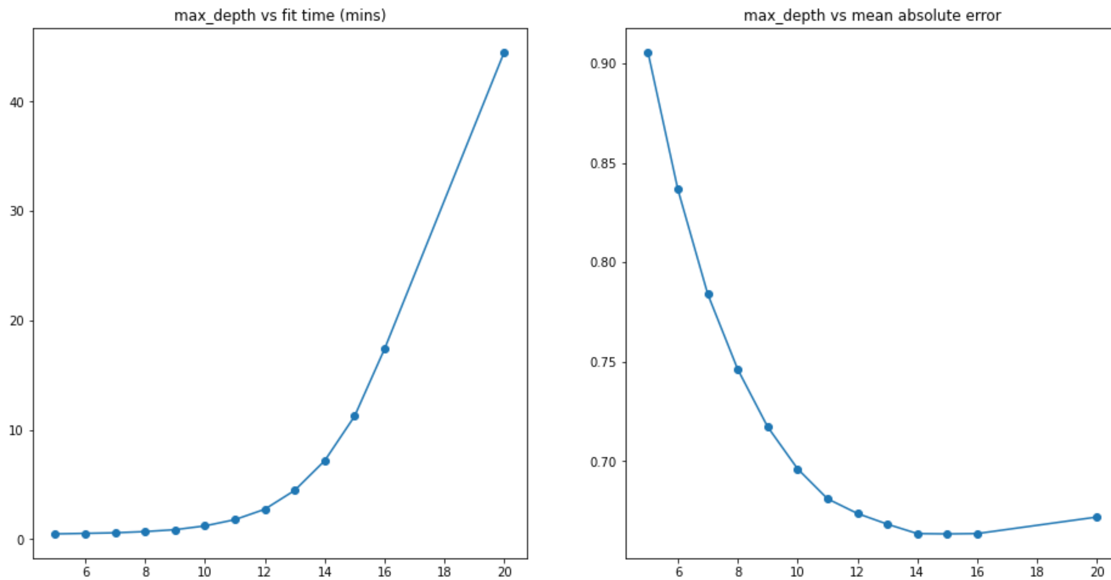

Figure 2 Left: changes in fit time with increasing max_depth, right: changes in mean absolute error with max_depth

max_depth	Fit time	Mae
10	1.2mins	0.6960
14	7.16mins	0.6635
15	11.2mins	0.6633

Table 3 Fit time and mean absolute error using different max_depth values

Table 3 shows the fit times and mean absolute error for different choices of *max_depth*. Reducing the *max_depth* to 14 would give very similar results and save a lot of time.

This example showed that there was the potential to speed up the machine learning work being carried out at the University considerably by using GPU systems. Although the training times have not yet caused a major problem for existing work, this will likely become a significant factor in the future as NCEO does more in this domain. Rob and Cristina have access to MAGEO to try out this example on a GPU and experiment with similar examples.

This example was a good fit for GPU acceleration because it was using the XGBoost library that has built-in GPU support. For this example, there was no need to write additional code to run it on a GPU, only the tree method needed to be changed. XGBoost is one of the open-source projects that the RAPIDS team collaborates with on the GPU accelerated methods (https://rapids.ai/community.html). These projects along with the RAPIDS suite of open-source software libraries allow users to run their code on GPUs whilst using a familiar API.