First Experiences with Google Earth Engine

José A. Navarro
Centre Tecnològic de Telecomunicacions de Catalunya (CTTC/CERCA), Av. Carl Friedrich Gauss, 7. Building B4, 08860 Castelldefels, Spain

Keywords: Google Earth Engine, Distributed Processing, Parallel Processing, Image Processing.

Abstract: This paper presents the first experiences of the author with GEE (Google Earth Engine). A C++ image processing algorithm, still under development, was migrated to this new environment using GEE's web interface and the JavaScript language. The idea is to discover the problems that might arise when migrating to this environment as well as to assess the presumable performance boost that should be achieved. A reduced—more didactic—version of the aforementioned algorithm is presented in a step-by-step way along with a brief description of the advantages and drawbacks—from the authors standpoint—of GEE.

1 INTRODUCTION

Google Earth Engine (GEE) (Google, 2015a) is a planetary-scale repository of satellite imagery and geospatial datasets as well as a powerful service offered to scientists and researchers to analyze such data. This paper describes the lessons learned by the author while migrating an image processing algorithm relying on Sentinel-1 imagery to GEE. The idea was to assess both the advantages and inconveniences of using such environment, as well as to evaluate the presumable performance boost that should be expected. GEE is available for the Python and JavaScript programming languages. This paper focuses on the browser-based JavaScript version available online, so the conclusions presented here should not be extrapolated to the Python counterpart.

The actual algorithm that motivated this work is too complicated to be didactic. A simplified—and *useless*—example is used instead. Such example, however, poses the same problems that had to be faced to migrate the original algorithm. It is presented in equations 1 and 2. Figure 1 depicts a very schematic C implementation of equation 1. Equation 2 is not shown in Figure 1. Figure 5 depicts the full JavaScript code implementing the example algorithm.

In a few words, for each value of n, ranging between 1 and the total number of images in a dataset minus 1 ($t-1$), equation 1 is applied, creating a set of $t-1$ intermediate results (images). Finally, these $t-1$ images are merged into a single one (equation 2), keeping the maximum pixel value for each of these.

```
for (n = 1; n <= t-1; n++)
  for (m = 1; m <= n; m++)
    for (r = 1; r <= rows; r++)
      for (c = 1; c <= columns; c++)
        q(n,r,c)+=pow(p(m,r,c),2.0)/pow(n,2.0);
  for (m = n + 1; m <= t; m++))
    for (r = 1; r <= rows; r++)
      for (c = 1; c <= columns; c++)
        q(n,r,c)+= p(m,r,c) / n;
```

Figure 1: C version of the algorithm shown in equation 1.

$$q_{n,r,c} = \frac{\sum_{m=1}^{n} p_{m,r,c}^2}{n^2} + \frac{\sum_{m=n+1}^{t} p_{m,r,c}}{n}, n \in [1..t-1] \quad (1)$$

$$i_{r,c} = \max(q_{n,r,c}), n \in [1..t-1] \quad (2)$$

where

n,r,c	Image, row and column indices.
t	Total number of images to process.
$p_{n,r,c}$	Value of the pixel for input image n, row r, column c.
$q_{n,r,c}$	Value of the pixel for intermediate output image n, row r, column c.
$i_{r,c}$	Values of the pixel for row r, column c in the results image.

2 SOME IMPORTANT FINDINGS

The following subsections describe some features, advantages and drawbacks of GEE that may not be explained using the example algorithm but that are important enough to be taken into account.

Navarro, J.
First Experiences with Google Earth Engine.
DOI: 10.5220/0006352702500255
In *Proceedings of the 3rd International Conference on Geographical Information Systems Theory, Applications and Management (GISTAM 2017)*, pages 250-255
ISBN: 978-989-758-252-3

```
// server-side list and boolean
var myList = ee.List([1, 2, 3]);
var serverBoolean = myList.contains(5);
// Will print "false".
print(serverBoolean);
// Client-side conditional. It will fail.
var clientConditional;
if (serverBoolean) {
  clientConditional = true;
} else {
  clientConditional = false;
}
// Should print false, but will print true.
print('Should be false:', clientConditional);
//Server-side conditional. It will work.
var serverConditional =
  ee.Algorithms.If(serverBoolean,
                   'True!', 'False!');
print('Should be false:', serverConditional);
```

Figure 2: Client-side versus server-side computing.

2.1 Client vs. Server Computation

The JavaScript incarnation of GEE clearly distinguishes between client and server worlds. When running a JavaScript GEE script many operations take place in the browser (client side) itself, but some others are executed by the Google's servers. It must be noted that these two worlds do not mix well. GEE provides a set of Earth Engine objects easily recognizable by their `ee.` prefix (as, for instance, `ee.Image`) that are handled exclusively by the server. On the other side, JavaScript provides variables which are only understood with the browser (Google, 2016a).

This leads to situations that are suprising enough. For instance, index-based iteration *is not recommended* when Earth Engine objects are involved, since this would mix client-side, JavaScript variables (the index) and server-side objects—the Earth Engine ones—leading to unpredictable (but always incorrect) results. *Map* (Google, 2016c) and *reduce* (Google, 2016f) operations are recommended instead. This poses some problems on the way algorithms are designed.

The same happens to conditional statements; server side (`ee.`) objects may offer boolean methods that, apparently, could be used by the client-side conditional statements. Since the value returned by those server objects is not a client-side boolean, client-side comparisons will also fail. In this particular case, however, it is possible to use server-side conditional statements. Figure 2, excerpted from (Google, 2016a), depicts this problem.

```
var i1 =
  ee.Image('MOD09GA/MOD09GA_005_2012_03_09');
var i2 =
  ee.Image('MOD09GA/MOD09GA_005_2012_03_08');
var result = ee.Image(i1 - i2);
result = result.multiply(0.0001);
```

Figure 3: Typical operations on images.

2.2 On-demand vs. Batch Computation

There are two computing modes in GEE: on-demand and batch. On-demand tasks are run interactively, immediately; on the other hand, background tasks are executed in a batch queue. The main difference between on-demand and batch tasks is a time limit. On-demand (interactive) tasks cannot run for more than five minutes; batch tasks may run indefinitely *as long as it is clear that the job is continuing to make progress* (Google, 2016g). Long computations, therefore, should be executed in batch mode.

By default, all tasks are run in on-demand mode; to execute a batch task, command `Export` (Google, 2016g) must be used.

2.3 Limits

One of the most frequent problems found when running an program in the GEE environment is the fact that limits *do* exist. Time limits and the way to avoid these have been commented in section 2.2. Memory and storage limits must also be taken into account. Memory limit problems normally arise when running some commands on big images; for instance the computation of the maximum and minimum values performed by the example algorithm will not finish until the extent of the output image is restricted by a clipping operation (section 3.5). Storage (to save results in Google Drive or Google Cloud) is also a problem; although GEE may be accessed using a free Google account, the standard 15 Gb storage limit is an important constraint to keep in mind.

2.4 Images, Channels and Pixels

The algorithm in Figure 1 implements the traditional pixel-based approach to image processing. Images, rows and columns are accessed by means of indices that let the programmer access every individual pixel to process. GEE, on the contrary, forgets about pixels and works directly with images and channels at once. For instance, Figure 3 shows how two images are loaded and a third one is computed.

Therefore, equations 1 and 2 must be rewritten as follows to adapt these to GEE's philosophy:

```
#pragma omp parallel for
for(int x=0; x < width; x++)
  for(int y=0; y < height; y++)
    result[x][y] = computeSomething(x,y);
```

Figure 4: Explicit parallelization directive in OpenMP.

$$q_n = \frac{\sum_{m=1}^{n} p_m^2}{n^2} + \frac{\sum_{m=n+1}^{t} p_m}{n}, n \in [1..t-1] \quad (3)$$

$$i = \max(q_n), n \in [1..t-1] \quad (4)$$

where

n	Image index.
t	Total number of images to process.
p_n	Input image n.
q_n	Intermediate output image n.
i	Results image.

In this context, for instance, p_m^2 stands for "square the values of all pixels in image p_m".

In short, the maximum granurality of the operations available in GEE is the image channel. It is not possible to set the value of a single pixel. This, of course, poses problems on the way algorithms are devised.

2.5 Transparent Parallelization

Parallelization is automatic and transparent in GEE (Google, 2016b). This means that no actions need to be taken to make the algorithm parallel or to indicate the parts that may be parallelized, as it happens with other parallelization frameworks (OpenMP, 2016). The `#pragma` directive in the first line of Figure 4 is an example showing how C / C++ code may be parallelized using OpenMP. The loop in the figure would not be parallelized if such directive would not exist. The effort to identify all the parallelizable parts in a complex algorithm should be obvious to the reader. GEE, as stated above, makes this task transparent to the programmer, so, in this case, all the advantages of parallelization come without the usual drawbacks. Parallelization is the key to high performance.

2.6 Deferred Processing

An important fact to note is that an algorithm in GEE may do nothing at all if no information is displayed on the screen or exported to disk files. GEE analyzes and optimizes the algorithm in such way that actual computations will take place when these are *absolutely* necessary. This feature allows, for instance, loading the *full* collection of Sentinel-1 images without running into memory problems. Obviously, if such collection is not filtered using any criteria, the algorithm will run into problems as soon as actual output is produced. The key point here is that computations that do not lead to actual results will never take place.

3 THE ALGORITHM IN GEE

The following sections describe how the example algorithm was implemented. The problems that arose during its implementation are also highlighted.

3.1 Retrieving the Images

Lines 1–5 of Figure 5 show how a few variables are initialized. Lines 6–21 of the same figure are much more interesting, since they show a powerful mechanism to fetch the input images. Line 6 declares a variable—`only1BandCollection`—to hold them. In line 7 the *full* Sentinel-1 collection is selected; line 8 reduces the number of images so only those captured between January 1st, 2014 and September 1st, 2016 are taken into account. In lines 9–10 the collection is filtered a bit more, keeping only those images covering the point whose coordinates are provided; the filtering process goes a little bit further in lines 11–13, where it is stated that the images captured along some specific orbital path of satellite should only be selected. Lines 14–16 state that the images still in the collection must be filtered once more to keep only those with a given pixel resolution, and lines 17–18 request that the instrument used to capture the imagery must be the one specified there.

Then a specific single band for every image in the collection is selected (line 20), so all the other bands are discarded. Finally, line 21 sorts the remaining images in a chronological order.

The point to stress here is that such a powerful image selection and filtering mechanism avoids the need to manually download the different images involved in the process, thus reducing substantially the time needed to get started.

3.2 Preparing the Images

Although equations 3 and 4 already assume the handling of full images instead of pixels—which is compatible with GEE's approach—these still rely on the use of index-based iteration—which is not (section 2.1). Consequently, the algoritm had to be adapted to GEE's phylosophy.

The first change was to limit the number of input images to process to a certain value (variable `max_images`). This is done in lines 22–23 of Figure 5. The key point is that, knowing the actual number

```
var orbit_number = 81;
var max_images = 5;
var band_to_extract = 'VV';
var resolution_meters = 10;
var instrumentMode = 'IW'
var only1BandCollection =
  ee.ImageCollection('COPERNICUS/S1_GRD')
  .filterDate('2014-01-01', '2016-09-1')
  .filterBounds(ee.Geometry.Point
    ([-3.714566, 40.425326]))
  .filter(ee.Filter.eq
    ('relativeOrbitNumber_start',
   orbit_number))
  .filter(ee.Filter.eq
    ('resolution_meters',
   resolution_meters))
  .filter(ee.Filter.eq
    ('instrumentMode',
   instrumentMode))
  .select(band_to_extract)
  .sort('system:time_start', false);
only1BandCollection =
  only1BandCollection.limit(max_images);
var mergeChannels =
  function(image, outputImage)
{
  outputImage =
    ee.Image(outputImage);
  outputImage =
    outputImage.addBands(image);
  return outputImage;
};
var mergedImage =
  ee.Image(only1BandCollection
  .iterate(mergeChannels,
           ee.Image(0).select([])));
mergedImage = mergedImage.double();
var logRemovedImage =
  ee.Image(10)
  .pow(mergedImage.divide(ee.Image(10)));
logRemovedImage =
  logRemovedImage
  .rename(['1','2','3','4','5']);
var bandMap =
{
  'a': logRemovedImage.select('1'),
  'b': logRemovedImage.select('2'),
  'c': logRemovedImage.select('3'),
  'd': logRemovedImage.select('4'),
  'e': logRemovedImage.select('5')
}
// n = 1
var intermediateResult =
  logRemovedImage.expression
(
'(a*a) +' +
'((b + c + d + e))',
  bandMap
);
var the_result =
  intermediateResult;
// n = 2
intermediateResult =
  logRemovedImage.expression
(
'((a*a + b*b) / 4) +' +
'((c + d + e) / 2)',
  bandMap
);
the_result =
  intermediateResult.max(the_result);
// n = 3
intermediateResult =
  logRemovedImage.expression
(
'((a*a + b*b + c*c) / 9) +' +
'((d + e) / 3)',
 bandMap
);
the_result =
  intermediateResult.max(the_result);
// n = 4
intermediateResult =
  logRemovedImage.expression
(
'((a*a + b*b + c*c + d*d) / 16) +' +
'(e / 4)',
  bandMap
);
the_result =
  intermediateResult.max(the_result);
var limits =
  ee.Geometry.Rectangle
  (-3.747899, 40.458659, -3.681233, 40.391993);
the_result = the_result.clip(limits);
var stats = the_result.reduceRegion({
  reducer: ee.Reducer.max()
           .combine(ee.Reducer.min(),
    null, true),
  maxPixels: 1e9,
  tileScale: 16});
var min_image_value =
  ee.Number(stats.get('1_min'));
var max_image_value =
  ee.Number(stats.get('1_max'));
var the_normalized_result =
  the_result.subtract(min_image_value)
  .divide(ee.Image.constant(max_image_value)
  .subtract(min_image_value))
Map.setCenter(-3.714566, 40.425326, 8);
Map.addLayer(the_normalized_result,
             {'min': 0, 'max': 1},
             'The normalized result');
Export.image.toDrive({
  image: the_normalized_result,
  description: 'The_normalized_result',
  scale: 10,
  maxPixels: 1e13
});
```

Figure 5: The algorithm.

of images to work with, it is possible to rewrite the equations *exactly for the specific number of selected images* (five in the example, line 2, Figure 5). Equations 3 and 4 may thus be rewritten as follows:

$$q_1 = (p_1^2) + (p_2 + p_3 + p_4 + p_5) \quad (5)$$

$$q_2 = ((p_1^2 + p_2^2)/4) + ((p_3 + p_4 + p_5)/2) \quad (6)$$

$$q_3 = ((p_1^2 + p_2^2 + p_3^2)/9) + ((p_4 + p_5)/3) \quad (7)$$

$$q_4 = ((p_1^2 + p_2^2 + p_3^2 + p_4^2)/16) + (p_5/4) \quad (8)$$

$$i = \max(q_1, q_2, q_3, q_4) \quad (9)$$

Obviously, these new equations will change depending on the actual number of selected images, which has direct implications on the maintainability of the code. However, this modification makes possible to implement the algorithm using the so-called GEE image *expressions* (Google, 2016d). An expression is a method of the `ee.Image` class that is able to parse a textual representation of a math operation and then apply it to the *channels* in the image. The problem is that expressions cannot involve several *images, but a single one*.

Since the algorithm has stored the single-channel `max_images` images in an image collection, expressions may not be used to compute the equations: expressions are able to work with the channels in a single image only. To solve this problem, the algorithm was adapted again to transform the `only1BandCollection` collection into an unique image made of these single channel images. This mutation made possible the use of expressions.

The solution is to iterate through the whole `only1BandCollection` image collection, calling a function as many times as images it contains. The function will append the (current) single-channel image in the collection (first parameter) as a new channel to an output, results image (second parameter). Once the iterator has invoked the function for each of the images in the collection, the results is the sought multichannel, merged image. To iterate through image collections their method `iterate` may be used. It takes two parameters: the name of the aforementioned function (`mergeChannels` in the example) and the results image, which must be empty. The function `mergeChannels` is defined in lines lines 24–31 of Figure 5. The iterator itself is invoked in lines 33–36. The result is assigned to `mergedImage`.

Line 37 changes the kind of values stored in `mergedImage` to double to avoid precision problems when computing the equations. In lines 38–40, the logarithmic scale affecting Sentinel-1 imagery (Google, 2015b) is removed using a simple arithmetic operation, so the original values are restored. The resulting image is `logRemovedImage`.

3.3 The Tailored Equations

Now it is possible to implement the tailored version of the algorithm as shown in equations 5 to 9 using expressions. The first thing to do is to rename the channels in `logRemovedImage` (lines 41–43 of Figure 5) to refer to these easily—the previous operations baptized the channels with rather weird names. Note that this command is fragile since it depends on the number of images selected at the beginning of the algorithm.

Expressions need to refer to the different channels in an image using *labels*. A dictionary (`bandMap`) is defined in lines 44-51 of the example. It allows the reduction of the amount of code required for each expression. It defines, respectively, channels 1 to 5 in image `logRemovedImage` as `a`, `b`, `c`, `d` and `e`, the actual labels used in the expressions. Then equations 5, 6, 7 and 8 are implemented as expressions in lines 52–59, 62–69, 72–79 and 82–89 respectively. Equation 9 is implemented sequentially, computing the partial maximum just after the evaluation of each expression (lines 70–71, 80–81 and 90–91). Note how the expressions mimetize the equations. The result of this process is stored in an image, `the_result`. To finish, it is worth to remark that the expressions used by the algorithm directly depend on the number of images to be processed, set at the beginning of the code, which again compromises code maintainability.

3.4 Clipping

Lines 92–94 in the example define a rectangle of interest. It will be used to clip the results image, so it will cover the area stated by such rectangle only. Clipping is *essential*; otherwise, the algorithm will try to compute a result for a very big area— as big as a full Sentinel-1 image—taking, for this example, about 20 hours of elapsed time to complete. Taking into account that the algorithm is run in parallel in several Google servers such amount of time is not negligible.

The clipping operation takes place in line 95. Note that the algorithm has not yet shown nor stored the result, so according to the deferred processing approach (section 2.6) no computations have taken place neither; that is why it is possible to clip the results image *after* evaluating the expressions at no computational cost.

3.5 Normalizing, Visualizing, Exporting

Although not shown in equations 5 to 9, the result had to be interpreted as a *probability map*; thus, the results image had to be rescaled to store values in the range

[0..1]. To do it, the minimum and maximum pixel values in `the_result` had to be computed. This was done by means of a *reduce* operation (Google, 2016e), that is, one that takes a full image and returns a single result (in this case, a set of statistics). The computation may be found in lines 96–101 of the example. Parameters `maxPixels` and `tileScale` are optional, but have been explicitly passed to reduce the chances of running into memory problems. Lines 102–105 copy the values of the minimum and maximum pixel values into floating point variables to use them later. Then the image is normalized by means of an arithmetic image operation (lines 106–109); the result is stored in the new image, `the_normalized_result`.

Note *again* that no actual processing has taken place until now because of the deferred processing feature (section 2.6). In lines 110-113 the map is displayed on the screen, so the actual processing can be deferred no more. The amount of work (pixels) to compute will depend on the scale of the map shown, so when using big scales the computation is almost *instantaneous* (only a few pixels will be actually computed). Decreasing the scale will increase the computational cost. Lines 114–118 export the result to Google Drive (other export methods exist). Exporting implies that the *whole* result will be computed and that the batch mode will be used (section 2.2); the computational cost will depend again on the actual `scale` selected (in meters per pixel).

4 CONCLUSIONS

GEE is a good tool for image-processig related research. The servers offered by Google plus the parallel processing approach and the availability of constantly updated on-line imagery are key points. The web-based interface is ideal to quickly test new ideas. On the contrary, the map / reduce / iterate approach that forgets about pixel-based operations, plus the absence of index-based server iterations are, at least in the beginning, serious obstacles to surmount: it implies a radical change in the way algorithms are implemented—which may be difficult to maintain due to tailored solutions (as setting a limited number of input images to process). Limitations do exist. The on-demand work mode restricts the maximum run time to 5 minutes. This means that the (complex) process of big areas must take place using the batch mode. Memory limitations also exist; these apply to both on-demand and batch modes. One solution to overcome these seems to be the clipping of the output area. To finish, performance is exceptional in on-demand mode. No results are available yet to compare the batch mode with the non-GEE implementation, although a performance boost is foreseable due to parallel, distributed execution in GEE.

ACKNOWLEDGEMENTS

The author wants to thank Noel Gorelick for the invaluable help provided when answering a noticeable number of questions in the Google Earth Engine developer's forum.

REFERENCES

Google (2015a). Google Earth Engine. Available online: https://earthengine.google.com/. Accessed: 2017-01-10.

Google (2015b). Sentinel-1: C-band Synthetic Aperture Radar (SAR) ground range data, log scaling. Available online: https://explorer.earthengine.google.com/#detail/COPERNICUS Accessed: 2016-12-28.

Google (2016a). Google Earth Engine API. Client vs. server. Available online: https://developers.google.com/earth-engine/client_server. Accessed: 2017-01-10.

Google (2016b). Google Earth Engine API. Introduction. Available online: https://developers.google.com/earth-engine. Accessed: 2017-01-10.

Google (2016c). Google Earth Engine API. Mapping over an image collection. Available online: https://developers.google.com/earth-engine/ic_mapping. Accessed: 2017-01-10.

Google (2016d). Google Earth Engine API. Mathematical operations. Available online: https://developers.google.com/earth-engine/image_math. Accessed: 2017-01-10.

Google (2016e). Google Earth Engine API. Reducer overview. Available online: https://developers.google.com/earth-engine/reducers_intro. Accessed: 2017-01-10.

Google (2016f). Google Earth Engine API. Reducing an image collection. Available online: https://developers.google.com/earth-engine/ic_reducing. Accessed: 2017-01-10.

Google (2016g). Google Earth Engine developer's forum; thread: Batch versus interactive. Available online: https://groups.google.com/forum/#!topic/google-earth-engine-developers/dIsa00Sm9e8. Accessed: 2017-01-10.

OpenMP (2016). The OpenMP API specification for parallel programming. Available online: http://www.openmp.org/. Accessed: 2016-12-28.