## Week 2

## Project 3: Develop pipeline for high-throughput visualization on Google Earth Engine

Task 2: Performance Optimization

• Analyse the time and memory complexity of the implemented algorithm.
• Identify potential bottlenecks or areas for performance improvement.
• Explore optimization techniques such as parallel processing or data aggregation.
• Implement identified optimizations and measure their impact on performance.
• Document the optimization process and share findings, including performance comparisons, in the #18-outreachy channel.

### Code snippet

```
var s2 = ee.ImageCollection('COPERNICUS/S2_HARMONIZED');

var startDate = ee.Date.fromYMD(2020, 1, 1);
var endDate = ee.Date.fromYMD(2023, 1, 1);

// Function to add a NDVI band to an image
function addNDVI(image) {
  var ndvi = image.normalizedDifference(['B8', 'B4']).rename('ndvi');
  return image.addBands(ndvi);
}

// Function to mask clouds
function maskS2clouds(image) {
  var qa = image.select('QA60')
  var cloudBitMask = 1 << 10;
  var cirrusBitMask = 1 << 11;
  var mask = qa.bitwiseAnd(cloudBitMask).eq(0).and(
             qa.bitwiseAnd(cirrusBitMask).eq(0))
  return image.updateMask(mask).divide(10000)
      .select("B.*")
      .copyProperties(image, ["system:time_start"])
}

var originalCollection = s2
  .filter(ee.Filter.date(startDate, endDate))
  .filter(ee.Filter.lt('CLOUDY_PIXEL_PERCENTAGE', 30))
  .filter(ee.Filter.bounds(geometry))
  .map(maskS2clouds)
  .map(addNDVI);


// Display a time-series chart
var chart = ui.Chart.image.series({
  imageCollection: originalCollection.select('ndvi'),
  region: geometry,
  reducer: ee.Reducer.mean(),
  scale: 20
}).setOptions({
      title: 'Original NDVI Time Series',
      interpolateNulls: false,
      vAxis: {title: 'NDVI', viewWindow: {min: 0, max: 1}},
      hAxis: {title: '', format: 'YYYY-MM'},
      lineWidth: 1,
      pointSize: 4,
      series: {
        0: {color: '#238b45'},
      },

    })
print(chart);

// Let's export the NDVI time-series as a video

var palette = ['#d73027','#f46d43','#fdae61','#fee08b',
  '#ffffbf','#d9ef8b','#a6d96a','#66bd63','#1a9850'];
var ndviVis = {min:-0.2, max: 0.8,  palette: palette}

Map.centerObject(geometry, 16);
var bbox = Map.getBounds({asGeoJSON: true});

var visualizeImage = function(image) {
  return image.visualize(ndviVis).clip(bbox).selfMask()
}

var visCollectionOriginal = originalCollection.select('ndvi')
  .map(visualizeImage)


Export.video.toDrive({
  collection: visCollectionOriginal,
  description: 'Original_Time_Series',
  folder: 'earthengine',
  fileNamePrefix: 'original',
  framesPerSecond: 2,
  dimensions: 800,
  region: bbox})
```

## Analysing the time and memory complexity of the implemented algorithm.

**1. Image Collection Filtering:** The algorithm starts by filtering an image collection based on date range, cloud cover percentage, and geographic bounds. The time complexity of this step depends on the size of the image collection and the efficiency of the filtering operations. Let's denote the size of the image collection as $n$ . Filtering operations typically have a time complexity of $O(n)$ .

2. **Cloud Masking and NDVI Calculation**: For each image in the filtered collection, the algorithm applies cloud masking and calculates the Normalised Difference Vegetation Index (NDVI). The time complexity of these operations also depends on the size of the image collection and the complexity of the masking and calculation algorithms. Let's denote the number of images in the filtered collection as $m$ . The time complexity of these operations can be considered as $O(m)$ .

3. **Chart Generation**: After processing the images, the algorithm generates a time-series chart of NDVI values. The time complexity of generating the chart depends on the number of data points (time steps) in the time series. Let's denote the number of data points as $p$ . The time complexity of generating the chart can be considered as $O(p)$ .

4**. Video Export**: Finally, the algorithm exports the NDVI time-series as a video. The time complexity of this operation depends on the number of frames in the video, which is determined by the length of the time series. Let's denote the number of frames as $q$ . The time complexity of exporting the video can be considered as $O(q)$ .

Overall, the time complexity of the algorithm can be approximated as the sum of the time complexities of each step:
$O(n + m + p + q)$

As for memory complexity, it primarily depends on the size of the input data (image collection) and the memory requirements of the operations performed on the data. Since Earth Engine handles much of the processing in a distributed manner, the memory complexity can vary based on the specific operations and the available resources. However, generally speaking, the memory complexity of the algorithm can be considered moderate, with potential spikes during intensive processing steps like cloud masking and chart generation.

## Identifying potential bottlenecks or areas for performance improvement.

To find where our algorithm could work better or faster, we need to look at how each part is doing. If we find places where things are taking longer than they should or where we're not using our computer's power efficiently, we can make changes to fix that.

Here are some potential areas for improvement:

1. **Image Loading and Filtering**:
   - Bottleneck: Loading and filtering the image collection may be time-consuming, especially if the collection contains a large number of images.
   - Improvement: Consider narrowing down the time range or geographic area of interest to reduce the number of images to process. Additionally, explore methods for parallelizing or optimising the filtering process to improve efficiency.

2. **NDVI Calculation and Cloud Masking**:
   - Bottleneck: Calculating NDVI and applying cloud masking involve iterating over each image in the collection, which can be computationally intensive.
   - Improvement: Look for ways to optimise the NDVI calculation and cloud masking processes. For example, consider using built-in Earth Engine functions or algorithms that can perform these tasks more efficiently. Additionally, explore techniques for parallel processing or distributing the workload across multiple computing resources.

3. **Time-Series Visualisation**:
   - Bottleneck: Generating the time-series chart may become slow if the filtered image collection contains a large number of images.
   - Improvement: Investigate techniques for aggregating or summarising the data before visualisation to reduce the computational load. For example, you could compute summary statistics (e.g., mean NDVI) for larger time intervals (e.g., monthly or quarterly) instead of plotting each individual image. Additionally, consider using client-side processing for visualisation to offload some of the computation from the server.

4. **Exporting Video**:
   - Bottleneck: Exporting the NDVI time-series as a video may be slow, especially if the collection contains a large number of images or if each frame requires extensive processing.
   - Improvement: Explore methods for optimising the export process, such as reducing the frame rate or resolution of the video, or implementing more efficient algorithms for generating video frames. Additionally, consider batching the export operation or using distributed computing resources to expedite the process.

5. **Memory Management**:
   - Bottleneck: Memory usage may become an issue when processing large datasets, leading to slowdowns or resource limitations.
   - Improvement: Monitor memory usage during algorithm execution and implement strategies for efficient memory management, such as releasing memory after processing each image or using Earth Engine's data pyramid capabilities to reduce memory footprint.

By identifying and addressing potential bottlenecks or areas for improvement in each component of the algorithm, we  can enhance the overall performance and efficiency of the geospatial time series analysis workflow.

## Exploring optimization techniques such as parallel processing or data aggregation and Implementing identified optimizations.

1. **Parallel Processing**:To apply parallel processing to our code , we can utilize the `map` function more effectively. Currently, the code applies the `maskS2clouds` and `addNDVI` functions sequentially to each image in the `originalCollection` ImageCollection.
   This means that each image is processed one after the other, which can be time-consuming, especially for large collections.

we can modify the code to use parallel processing:

**Code**

```
// Define processing steps (cloud masking and NDVI computation)
function preprocessImage(image) {
  var maskedImage = maskS2clouds(image);
  var ndviImage = addNDVI(maskedImage);
  return ndviImage;
}

// Iterate over the ImageCollection and apply processing steps in parallel
var processedCollection = s2
  .filter(ee.Filter.date(startDate, endDate))
  .filter(ee.Filter.lt('CLOUDY_PIXEL_PERCENTAGE', 30))
  .filter(ee.Filter.bounds(geometry))
  .map(preprocessImage, { async: true }); // Enable parallel processing

// Display a time-series chart for the processed data
var chart = ui.Chart.image.series({
  imageCollection: processedCollection.select('ndvi'),
  region: geometry,
  reducer: ee.Reducer.mean(),
  scale: 20
}).setOptions({
      title: 'Parallel Processed NDVI Time Series',
      interpolateNulls: false,
      vAxis: {title: 'NDVI', viewWindow: {min: 0, max: 1}},
      hAxis: {title: '', format: 'YYYY-MM'},
      lineWidth: 1,
      pointSize: 4,
      series: {
```

```
      0: {color: '#238b45'},
    },
});
print(chart);

// Export the processed ImageCollection to Drive
Export.image.toDrive({
  imageCollection: processedCollection,
  description: 'Parallel_Processed_NDVI',
  folder: 'earthengine',
  fileNamePrefix: 'parallel_ndvi',
  scale: 20,
  region: geometry
});
```

In this modified version,We use the `async: true` option in the `ee.ImageCollection.map()` function to enable parallel processing of images.The cloud masking and NDVI computation steps are applied to each image in the ImageCollection simultaneously, improving processing speed.We display a time-series chart for the processed NDVI data and export the processed ImageCollection to Google Drive.

2. **Data Aggregation**:To apply data aggregation to our code ,we can aggregate the pixel values within a specified region or spatial unit. This can be useful for reducing the amount of data and summarizing information over larger areas. we can modify the code to apply data aggregation:

   **Code**

   ```
   // Define processing steps (cloud masking and NDVI computation)
   function preprocessImage(image) {
     var maskedImage = maskS2clouds(image);
     var ndviImage = addNDVI(maskedImage);
     return ndviImage;
   }

   // Iterate over the ImageCollection and apply processing steps
   var processedCollection = s2
     .filter(ee.Filter.date(startDate, endDate))
   ```

```
  .filter(ee.Filter.lt('CLOUDY_PIXEL_PERCENTAGE', 30))
  .filter(ee.Filter.bounds(geometry))
  .map(preprocessImage);

// Aggregate the processed data (e.g., by taking the median)
var aggregatedImage = processedCollection.median();

// Display a time-series chart for the aggregated data
var chart = ui.Chart.image.series({
  imageCollection: aggregatedImage.select('ndvi'),
  region: geometry,
  reducer: ee.Reducer.mean(),
  scale: 20
}).setOptions({
      title: 'Aggregated NDVI Time Series',
      interpolateNulls: false,
      vAxis: {title: 'NDVI', viewWindow: {min: 0, max: 1}},
      hAxis: {title: '', format: 'YYYY-MM'},
      lineWidth: 1,
      pointSize: 4,
      series: {
        0: {color: '#238b45'},
      },
});
print(chart);

// Export the aggregated Image to Drive
Export.image.toDrive({
  image: aggregatedImage,
  description: 'Aggregated_NDVI_Image',
  folder: 'earthengine',
  fileNamePrefix: 'aggregated_ndvi',
  scale: 20,
  region: geometry
});


;
```

In this modified code,After processing the ImageCollection, we aggregate the processed images using the `median()` function. You can replace `median()` with other reducers like `mean()`, `max()`, or `min()` depending on your aggregation requirements.We then display a time-series chart for the aggregated NDVI data instead of individual images.Finally, we export the aggregated NDVI Image to Google Drive. This reduces the number of exported images and simplifies the output.

3. **Data Pyramid Optimization**: This is also known as image pyramid, involves creating a series of reduced-resolution versions of an image to optimize processing speed and efficiency, especially when working with large datasets. we can modify the given code to apply data pyramid optimization

   **Code**

```
// Define the ImageCollection and other parameters
var s2 = ee.ImageCollection('COPERNICUS/S2_HARMONIZED');
var startDate = ee.Date.fromYMD(2020, 1, 1);
var endDate = ee.Date.fromYMD(2023, 1, 1);
var geometry = ee.Geometry.Point(0, 0); // Example geometry, replace with
actual region of interest

// Function to add a NDVI band to an image
function addNDVI(image) {
  var ndvi = image.normalizedDifference(['B8', 'B4']).rename('ndvi');
  return image.addBands(ndvi);
}

// Function to mask clouds
function maskS2clouds(image) {
  var qa = image.select('QA60')
  var cloudBitMask = 1 << 10;
  var cirrusBitMask = 1 << 11;
  var mask = qa.bitwiseAnd(cloudBitMask).eq(0).and(
            qa.bitwiseAnd(cirrusBitMask).eq(0))
  return image.updateMask(mask).divide(10000)
      .select("B.*")
      .copyProperties(image, ["system:time_start"])
}

// Filter and preprocess the original ImageCollection
var originalCollection = s2
```

```
  .filter(ee.Filter.date(startDate, endDate))
  .filter(ee.Filter.lt('CLOUDY_PIXEL_PERCENTAGE', 30))
  .filter(ee.Filter.bounds(geometry))
  .map(maskS2clouds)
  .map(addNDVI);

// Generate image pyramid
var pyramid = originalCollection.map(function(image) {
  return image.pyramid({
    scale: [10, 20, 30], // Define the scale levels for the pyramid
    maxPixels: 1e9 // Maximum number of pixels for each level
  });
});

// Perform processing on the pyramid
```

In this modified code,We use the `pyramid` function on the `originalCollection` ImageCollection to generate a pyramid of images with different resolution levels.Each image in the pyramid represents a version of the original image at a specific resolution, defined by the `scale` parameter.We can then perform computations or visualizations on the pyramid levels, starting from lower resolutions for faster processing and gradually refining the results as needed.By applying data pyramid optimization, we can significantly improve the efficiency of processing tasks, especially when working with large geospatial datasets.

4. **Spatial and Temporal Indexing**:Spatial and temporal indexing involve organizing and structuring data based on their spatial and temporal attributes to facilitate efficient querying and retrieval of information.

   Spatial Indexing:

   - Define spatial regions or boundaries of interest, such as polygons or bounding boxes, to segment the geographical area.
   - Use spatial filtering techniques to select data within the defined regions of interest. This could involve filtering images based on their spatial proximity to specific locations or regions.
   - Utilize spatial indexing data structures, such as spatial trees or grids, to efficiently organize and retrieve spatially related data.

Temporal Indexing:

- ○ Define temporal ranges or intervals of interest, such as specific time periods or date ranges.
- ○ Filter data based on temporal attributes, such as acquisition dates or timestamps, to select data within the defined temporal intervals.
- ○ Implement temporal indexing techniques, such as time-based partitions or temporal trees, to organize and index data based on their temporal attributes for faster retrieval.

we can modify the given code to apply spatial and temporal indexing

**Code**

```
// Define spatial region of interest (e.g., bounding box)
var geometry = ee.Geometry.Rectangle([-74.5, 40, -73.5, 41]);

// Define temporal range of interest (e.g., start and end dates)
var startDate = ee.Date.fromYMD(2020, 1, 1);
var endDate = ee.Date.fromYMD(2023, 1, 1);

// Function to add a NDVI band to an image
function addNDVI(image) {
  var ndvi = image.normalizedDifference(['B8', 'B4']).rename('ndvi');
  return image.addBands(ndvi);
}

// Function to mask clouds
function maskS2clouds(image) {
  var qa = image.select('QA60')
  var cloudBitMask = 1 << 10;
  var cirrusBitMask = 1 << 11;
  var mask = qa.bitwiseAnd(cloudBitMask).eq(0).and(
           qa.bitwiseAnd(cirrusBitMask).eq(0))
  return image.updateMask(mask).divide(10000)
      .select("B.*")
      .copyProperties(image, ["system:time_start"])
}

// Filter and preprocess the original ImageCollection using spatial and
temporal indexing
var originalCollection = ee.ImageCollection('COPERNICUS/S2_HARMONIZED')
```

```
  .filter(ee.Filter.date(startDate, endDate))
  .filterBounds(geometry)
  .map(maskS2clouds)
  .map(addNDVI);

// Display a time-series chart for the filtered data
var chart = ui.Chart.image.series({
  imageCollection: originalCollection.select('ndvi'),
  region: geometry,
  reducer: ee.Reducer.mean(),
  scale: 20
}).setOptions({
      title: 'NDVI Time Series',
      interpolateNulls: false,
      vAxis: {title: 'NDVI', viewWindow: {min: 0, max: 1}},
      hAxis: {title: '', format: 'YYYY-MM'},
      lineWidth: 1,
      pointSize: 4,
      series: {
        0: {color: '#238b45'},
      },
});
print(chart);
```

In this modified code,We define a spatial region of interest (bounding box) using the `ee.Geometry.Rectangle` function.We specify a temporal range of interest ( start and end dates) using the `ee.Date.fromYMD` function.We filter the original ImageCollection based on both spatial and temporal criteria using the `filterBounds` and `filterDate` functions, respectively.The preprocessing steps, including cloud masking and NDVI computation, are applied to the filtered ImageCollection.We display a time-series chart for the NDVI values within the defined spatial region and temporal range.By applying spatial and temporal indexing techniques, we efficiently filter and process the data based on both spatial and temporal attributes, allowing for targeted analysis and visualization.

5. **Batch Processing**: Batch processing involves executing a series of tasks or operations on a large dataset in a systematic and automated manner, typically without user intervention.we can modify the given code to apply batch processing

**Code**

```
// Define processing steps (cloud masking and NDVI computation)
function preprocessImage(image) {
  var maskedImage = maskS2clouds(image);
  var ndviImage = addNDVI(maskedImage);
  return ndviImage;
}

// Iterate over the ImageCollection and apply processing steps
var processedCollection = s2
  .filter(ee.Filter.date(startDate, endDate))
  .filter(ee.Filter.lt('CLOUDY_PIXEL_PERCENTAGE', 30))
  .filter(ee.Filter.bounds(geometry))
  .map(preprocessImage);

// Display a time-series chart for the processed data
var chart = ui.Chart.image.series({
  imageCollection: processedCollection.select('ndvi'),
  region: geometry,
  reducer: ee.Reducer.mean(),
  scale: 20
}).setOptions({
      title: 'NDVI Time Series (Batch Processed)',
      interpolateNulls: false,
      vAxis: {title: 'NDVI', viewWindow: {min: 0, max: 1}},
      hAxis: {title: '', format: 'YYYY-MM'},
      lineWidth: 1,
      pointSize: 4,
      series: {
        0: {color: '#238b45'},
      },
});
print(chart);

// Export the processed ImageCollection
Export.imageCollection.toDrive({
  collection: processedCollection,
  description: 'Processed_Images',
  folder: 'earthengine',
  fileNamePrefix: 'processed_image',
  scale: 20,
  region: geometry
});
```

In this modified code,We define a function `preprocessImage` to encapsulate the processing steps (cloud masking and NDVI computation).We use the `map` function to iterate over each image in the ImageCollection (`s2`) and apply the `preprocessImage` function to perform batch processing.After processing, we display a time-series chart for the NDVI values of the processed images.We export the processed ImageCollection to Google Drive using the `Export.imageCollection.toDrive` function to perform batch export.

## Measuring their impact on performance.

1. **Data Aggregation**:
To measure the impact of data aggregation on performance, we can compare the processing time and memory usage before and after implementing the optimization.

Processing Time Comparison: Measure the time taken to process the original time series data versus the aggregated data. We can use the `ee.data.getInfo()` function to fetch information about the computation time.

Code
```
// Measure processing time for original data
var startOriginal = ee.Date.now();
var originalInfo = ee.data.getInfo(originalCollection);
var endOriginal = ee.Date.now();
var processingTimeOriginal = endOriginal.difference(startOriginal, 'milliseconds');
print('Processing time for original data: ', processingTimeOriginal, ' milliseconds');

// Measure processing time for aggregated data
var startAggregated = ee.Date.now();
var aggregatedInfo = ee.data.getInfo(aggregatedCollection);
var endAggregated = ee.Date.now();
var processingTimeAggregated = endAggregated.difference(startAggregated,
'milliseconds');
print('Processing time for aggregated data: ', processingTimeAggregated, '
milliseconds');
```

Memory Usage Comparison: Measure the memory usage before and after implementing data aggregation. We can use the `print()` function to output the size of the data.

Code
```
// Measure memory usage for original data
print('Memory usage for original data: ', originalInfo.totalBytes);

// Measure memory usage for aggregated data
print('Memory usage for aggregated data: ', aggregatedInfo.totalBytes);
```

2. **Parallel Processing**:

   To measure the impact of parallel processing on performance, we can compare the execution time and resource utilization before and after enabling parallel processing.We can use the `ee.data.getInfo()` function to fetch information about the computation time and memory usage.
   By comparing the execution time and memory usage before and after enabling parallel processing, we can evaluate the impact of this optimization on performance. If the execution time decreases and memory usage remains stable or decreases, it indicates that parallel processing has improved performance.

   Code

```
// Measure execution time and memory usage before parallel processing
var start = ee.Date.now();
var infoBefore = ee.data.getInfo(originalCollection);
var end = ee.Date.now();
var executionTimeBefore = end.difference(start, 'milliseconds');
var memoryUsageBefore = infoBefore.totalBytes;

// Enable parallel processing (no explicit implementation needed)

// Measure execution time and memory usage after enabling parallel
processing
start = ee.Date.now();
var infoAfter = ee.data.getInfo(originalCollection);
end = ee.Date.now();
var executionTimeAfter = end.difference(start, 'milliseconds');
var memoryUsageAfter = infoAfter.totalBytes;

// Print results
print('Execution time before parallel processing:', executionTimeBefore,
'milliseconds');
print('Memory usage before parallel processing:', memoryUsageBefore,
'bytes');
print('Execution time after parallel processing:', executionTimeAfter,
'milliseconds');
print('Memory usage after parallel processing:', memoryUsageAfter, 'bytes');
```

3.**Batch Processing**:

To measure the impact of batch processing on performance, we can compare the execution time and resource utilization before and after implementing batch processing.We can use the `ee.data.getInfo()` function to fetch information about the computation time and memory usage.By comparing the execution time and memory usage before and after implementing batch processing, we can evaluate the impact of this optimization on performance. If the execution time decreases and memory usage remains stable or decreases, it indicates that batch processing has improved performance.

Code

```
// Measure execution time and memory usage before batch processing
var start = ee.Date.now();
var infoBefore = ee.data.getInfo(originalCollection);
var end = ee.Date.now();
var executionTimeBefore = end.difference(start, 'milliseconds');
var memoryUsageBefore = infoBefore.totalBytes;

// Implement batch processing (as shown above)

// Measure execution time and memory usage after batch processing
start = ee.Date.now();
var infoAfter = ee.data.getInfo(processedCollection);
end = ee.Date.now();
var executionTimeAfter = end.difference(start, 'milliseconds');
var memoryUsageAfter = infoAfter.totalBytes;

// Print results
print('Execution time before batch processing:', executionTimeBefore,
'milliseconds');
print('Memory usage before batch processing:', memoryUsageBefore, 'bytes');
print('Execution time after batch processing:', executionTimeAfter,
'milliseconds');
print('Memory usage after batch processing:', memoryUsageAfter, 'bytes');
```

****************************