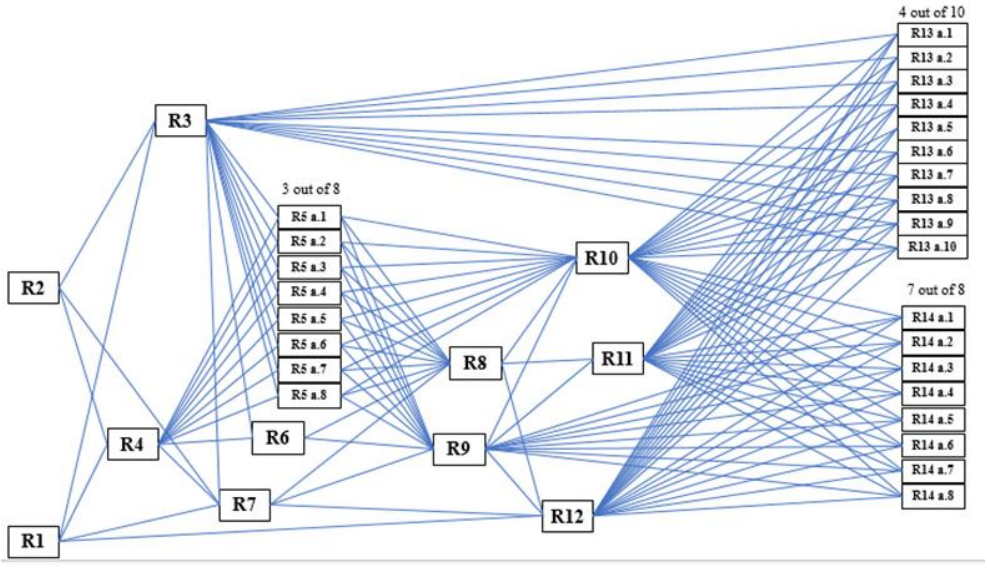
Yeshiva University - Katz School of Science
Data Analytics and Visualization MS
Work done in collaboration with A.O. Olivera,
Mechanical & Electrical Engineering Student at
University of Puerto Rico – Mayaguez Campus

Algorithm Optimization

This document describes the essential techniques used when optimizing the comb () algorithm. The document covers the use of key features aimed at solving the algorithm. More specifically, the note describes the optimization process step by step to show the whole development flow. The document assumes some experience with AWS and Python programing language.

Contents



Introduction

To effectively optimize the MATLAB algorithm, the programmer must become intimately familiar with the features of the MATLAB programing language and the target processor that will eventually execute the algorithm. Knowing the functional units available and the parallelism they offer, the data types and sizes that are supported, the paths to data and instruction memory will allow the developer to write Python code that is tuned for that architecture. The programmer must take advantage of the full core data bandwidth and use proper intrinsic to move data into or from memory, resulting in efficient code.

A general procedure of writing and optimizing MATLAB code is given below.

1. Start with MATLAB code.
    - Compiling with debug mode before optimization to verify its functionality first was impossible due to the lengthy indefinite processing time.
2. Translate MATLAB code to Python code
    - Use a Python compiler and optimization skills to provide information to the compiler and improve the performance based on compiler feedback.
3. Update Python Code
    - Identify code sections that need to be further optimized using profiling tools

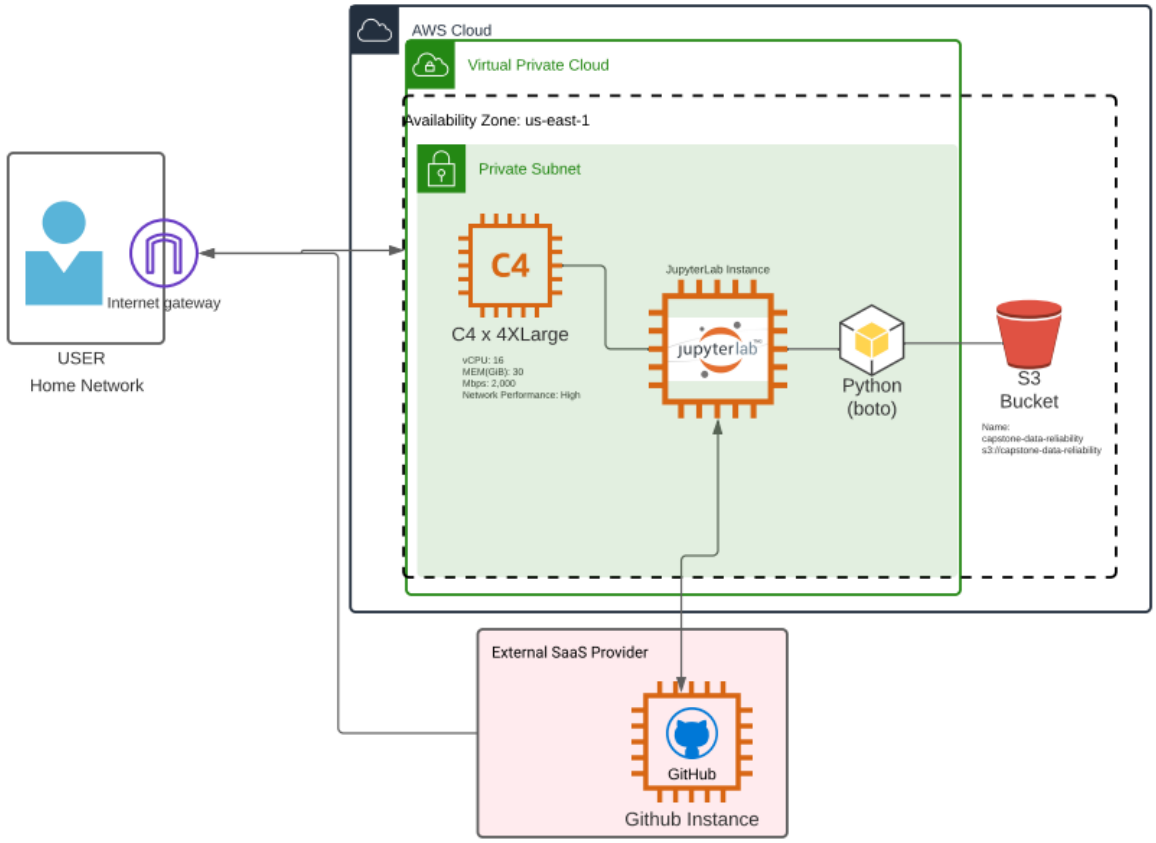
The goal of this document is to guide users on how to optimize the algorithm code. A general flow of the optimization is proposed and given to demonstrate how to follow the flow in real implementation. This document assumes that users are familiar with the following:

- MATLAB and Python programing languages
- AWS Architecture /products and services

Optimization Target

The optimization target is to solve the algorithm. It is very important to know and understand what the resources are needed, and costs associated with solving this algorithm. For the purpose of this project costs will not be discussed. Full pricing can be obtained in AWS website.

The challenge facing programmers was to measure the resources needed available and the extra resources required in this advanced architecture. The implementation of parallelism and data flows is dependent on a unique architecture and access to virtual machines with specific processing and memory units available.



Optimization Tool: Compiler

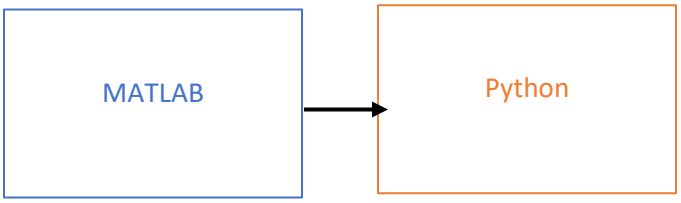
Python code programming is common for developers compared with assembly code thus accelerates your time to market. The Jupyter-Lab environment work as the interface between the virtual machine and the programmers. To fully use the virtual machine resources, the programmer need knowledge on how Jupyter-Lab works.

Optimization Techniques

In this section optimization techniques that were applied will be discussed.

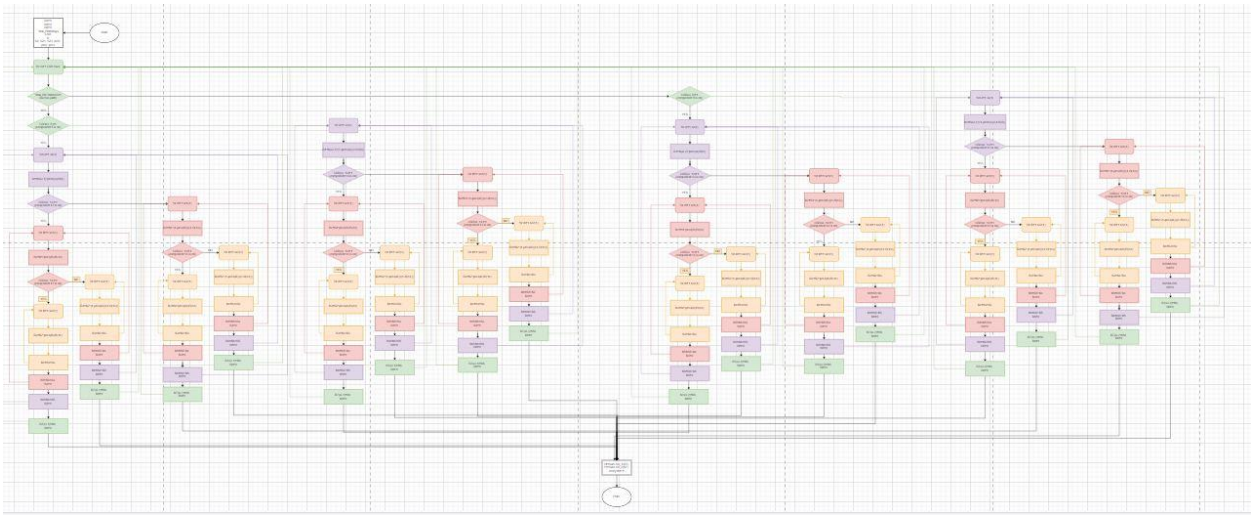
A. Translating MATLAB code to Python Code

The MATLAB code was translated to python to increase portability, to reduce runtime and to increase overall efficiency. The complete process of translating the code took approximately two weeks, close to 30 hours.

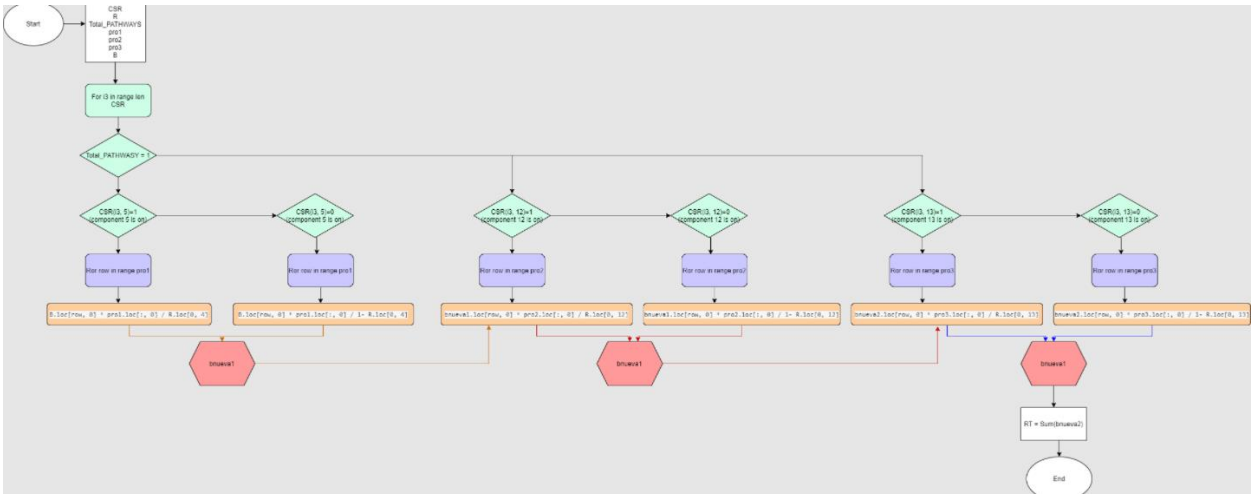


B. **Loop Optimization and Data Alignment:** For Loops reduction leveraging Dataframes

- Original MATLAB code flow diagram:



- Updated Python code flow diagram:



C. **Data Alignment:** Removal of 0s from original B input. By removing the 0s from the computation the performance runtime for block 1 went from 12 hours to 3 hours. Example below.

```
bnueva = bnueva.append(pd.DataFrame(a), ignore_index=True).replace(0, np.nan).dropna(how='all', axis=0)
```

D. **Profiling and Parallelism**: Splitting the output from block 1 and feeding it in batches to block 2. Processing the data in this way achieved a runtime of ~ 33 hours. Example

```
bnueva_A = bnueva.iloc[0:173040,:].copy().reset_index(drop=True)
```

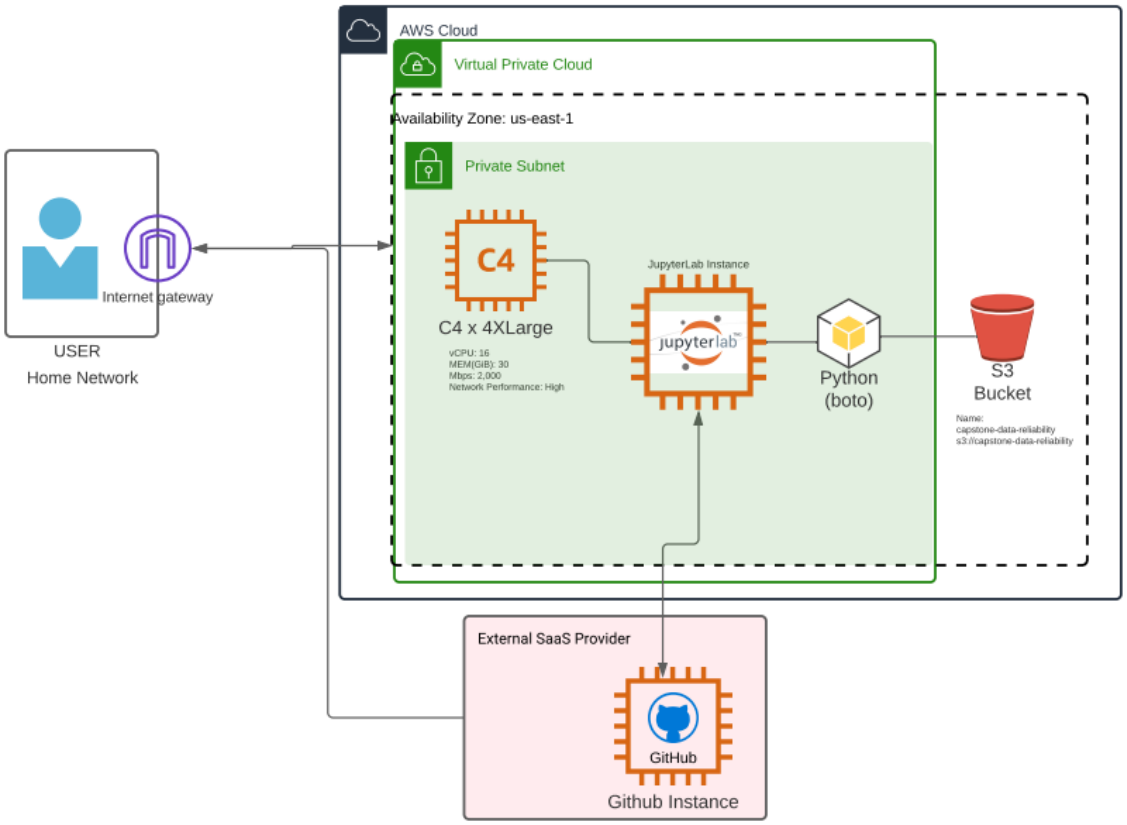
```
bnueva_A1 = bnueva.iloc[0:1000,:].copy().reset_index(drop=True)
```

E. **Profiling and Parallelism**: Splitting the ouput from block 2, feeding it in batches to block 3, and only performing final division to final output. Processing the data in this way achieved a runtime of ~ 20 per batch. Example below.

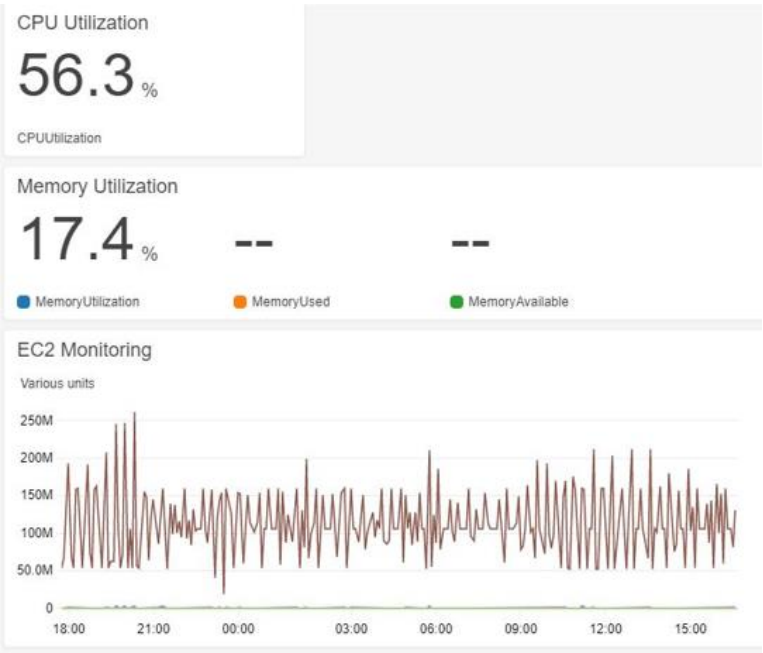
```
bnueva1 = bnueva.iloc[0:5424301,:].copy().reset_index(drop=True)
bnueva2 = bnueva.iloc[5424301:10848602,:].copy().reset_index(drop=True)
```

```
tot = tot_1 + tot_2 / R.loc[0, 13]
```

Optimized Computational Environment:



High Peak Performance



Minimum Resources Requirements (peak)

vCPU: 9

MEM(GiB): 5.1

NetworkOut (Bytes): 110,444,655

NetworkIn (Bytes): 5,706,704