Statistical Measure of Similarity between Protein 3-D Structures

Rashida Hasan
CLID# rxh5385, rxh5385@louisiana.edu

Titli Sarkar
CLID# txs7980, txs7980@louisiana.edu

Supervised By:

Dr. Vijay Raghavan

A Project Submitted in Partial Fulfillment of the Requirements for the Course
CSCE-561: Information Storage and Retrieval

Project URL: https://github.com/TitliSarkar/CSCE-561-Final-Project

The Center for Advanced Computer Studies

University of Louisiana at Lafayette, USA

December 2016

Table of content

I. INTRODUCTION

Proteins are macromolecules or natural polymers, formed by repeating units of amino acids. There are 20 naturally occurring amino acids that can repeat in a protein several hundred times resulting into extremely complex structures. The comparative analysis of protein data, can have breakthrough contributions to understanding protein functions, drug design and in determining the cause of several diseases. The greatest challenge in representing protein structures is the complexity of the protein macromolecule in addition to its size. On an average, there are 300 amino acids in each protein and each protein has three structural levels-primary, secondary and tertiary. The lack of reference frame makes comparison of two proteins computationally expensive and approximate. Popular 3-D structure comparison algorithms are computationally expensive, distance-based methods that require translation and rotation to achieve precise alignment before calculating similarity the root mean square distance between the two macromolecules. Our goal in this project is to find the similarities between all protein-protein pairs from a given data set using Generalized Jaccard Coefficient Measure. We have achieved a significant speedup by using parallel algorithm.

This work is inherited from Sumi Singh, former Ph.D student of Dr. Vijay Raghavan [1].

II. METHODOLOGY

This section describes the problem formulation for finding similarity between all combination of protein pairs, details description of our methodology, tools and languages we used, design, algorithms and methods.

A Problem Formulation

Proteins are defined by key-value pairs and not just by the set of keys. Therefore, the weight vector of two protein structures $p_1 and\ p_2$ are taken into consideration. Equivalence ϵ for a given key k_i in two different proteins p_1 and p_2 is defined by Equation 1 and the difference z for a given key k_i in a pair of proteins is given by Equation 2.

$$\epsilon_i = k_i^{p_1} \cap k_i^{p_2} \quad \text{Equation 1}$$

$$z_i = k_i^{p_1} \cup k_i^{p_2} \quad \text{Equation 2}$$

where

∩ is define by the *minimum* count of similar keys
U is defined by the *maximum* count of similar keys

The Generalized Jaccard coefficient measure is used to calculate the similarity between two proteins represented. The Generalized Jaccard similarity coefficient is given by Equation 3.

$$\text{Jac}_{gen} = \sum_{i=1}^{n} \epsilon_i / \sum_{i=1}^{n} z_i \quad \text{Equation 3}$$

where

n is the total number of unique keys in proteins $p_1 and\ p_2$
ϵ_i and z_i are obtained from Equation 1 & 2

B Specifications

Dataset: We use Sumi Singh's raw dataset of proteins with key-value pairs. Each protein is represented as a '*. keys' file. Inside the file the data are stores as "key-value" pairs in each line.
System Specification: 2.30 GHz Intel core i5, 8 GB RAM
Graphics Card: NVIDIA Geforce GTX 960M (Compute Capability: 5.0)
Operating System*:* Windows 10 amd64
Tools: Anaconda 3.5 (python dsiribution), Spyder 3.5
Programming Language**:** Python, pyCUDA (for GPU implementation)

C Design

a. Logic Design

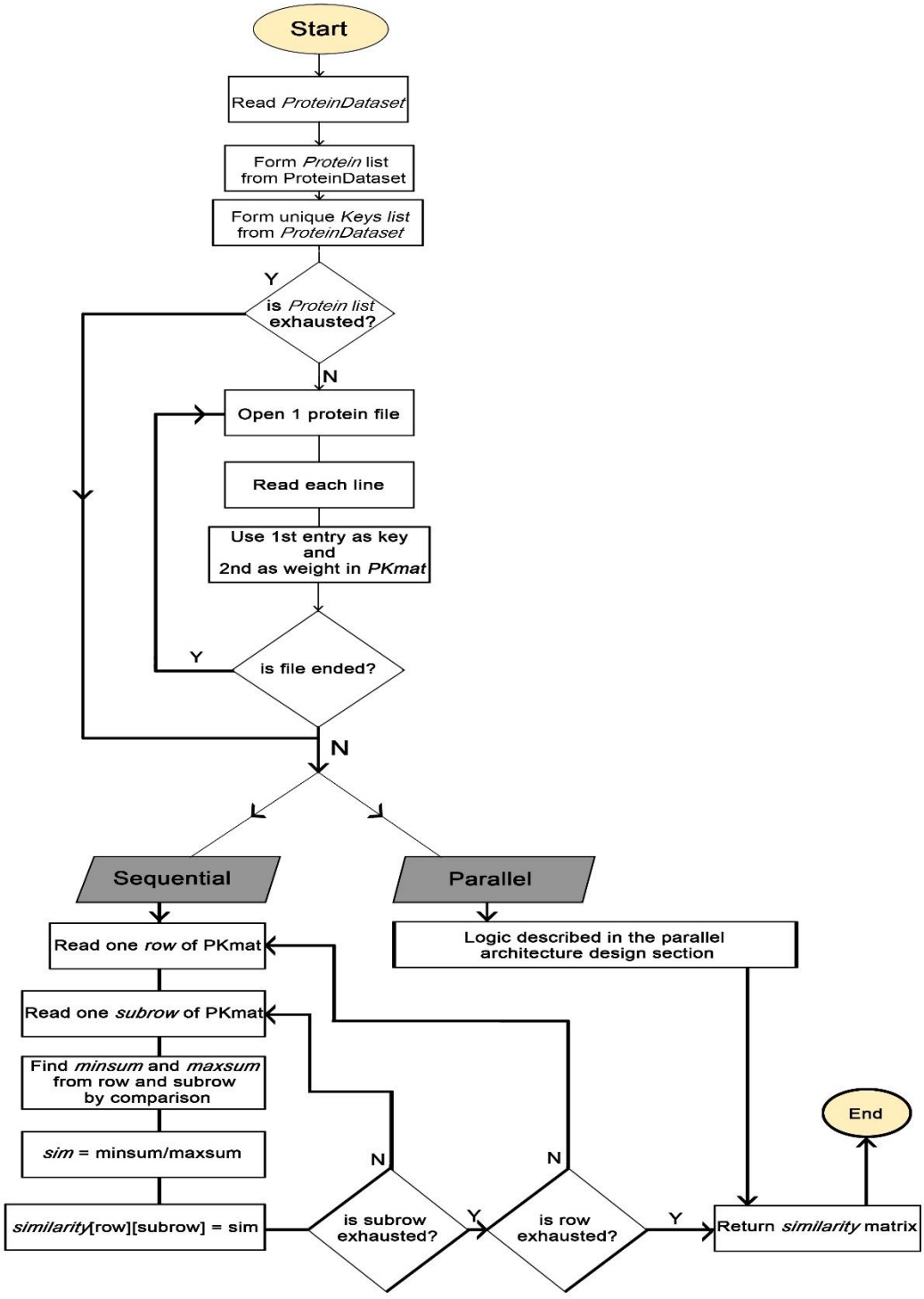


Fig 1: Flowchart of program Logic

b. Parallel Architecture

We use power of CUDA which is a parallel computing platform (data-level parallelism, not process level) and application programming interface(API) invented by NVIDIA. It enables dramatic increases in computing performance by harnessing the power of the graphics processing unit (GPU). The process flow of CUDA architecture is shown below:

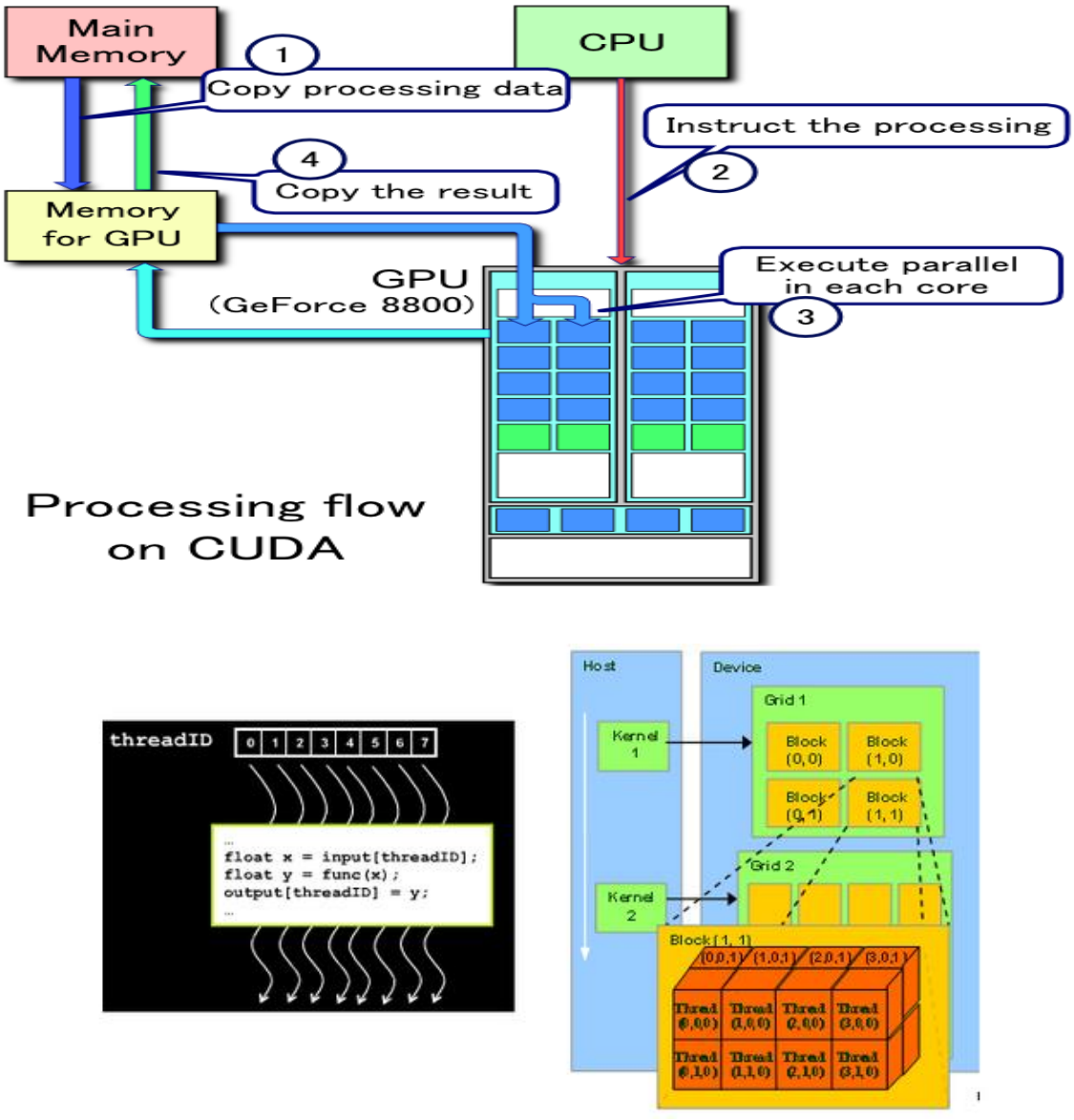
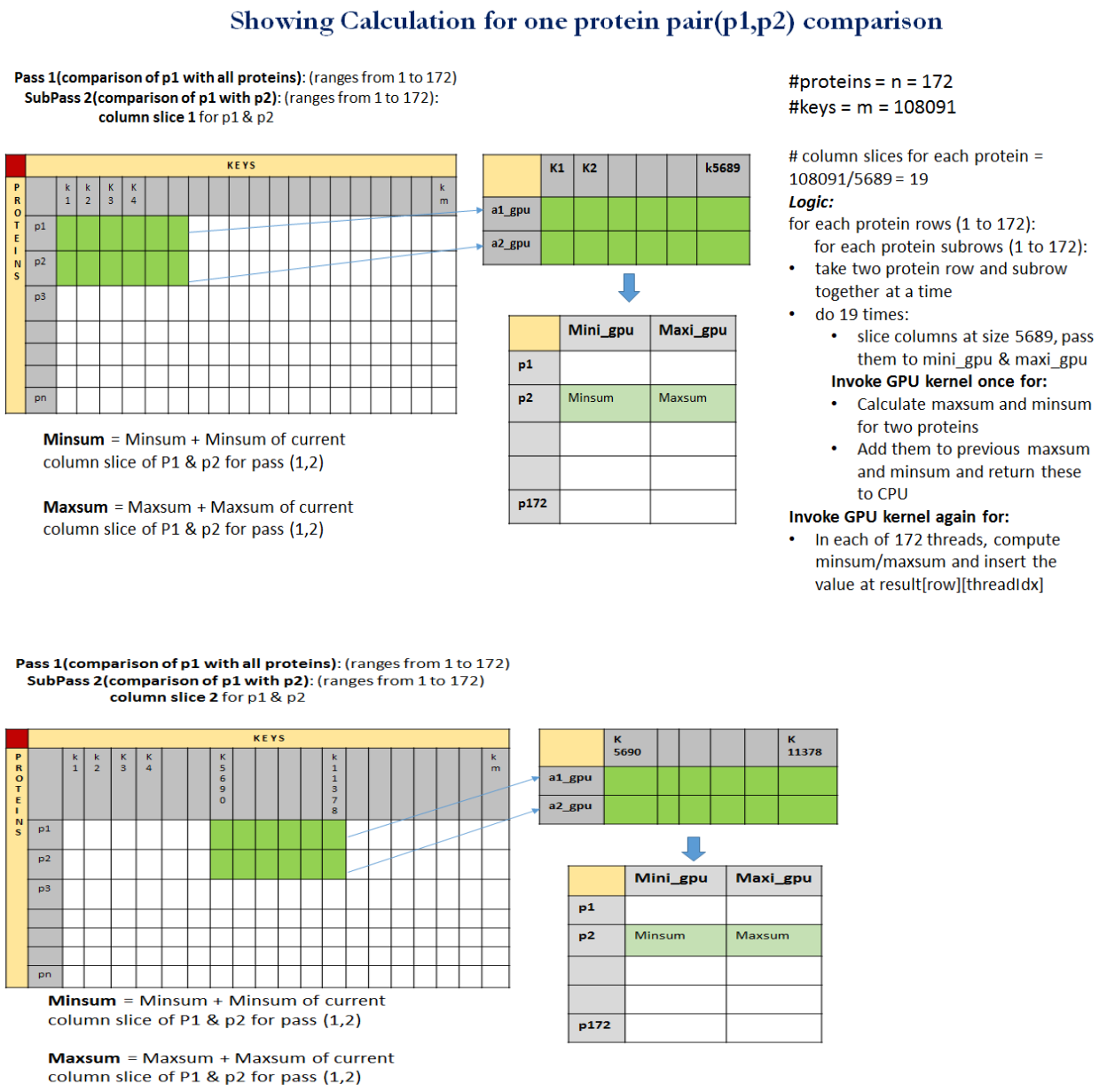
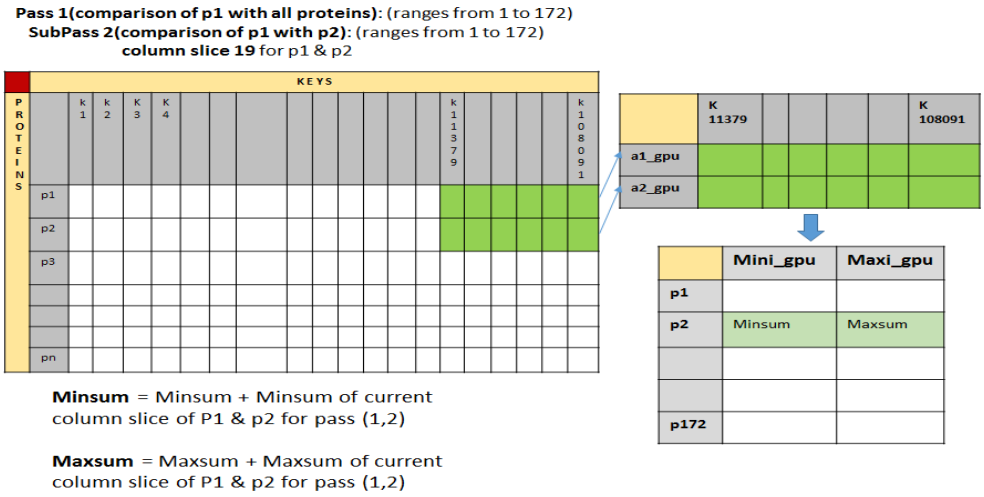


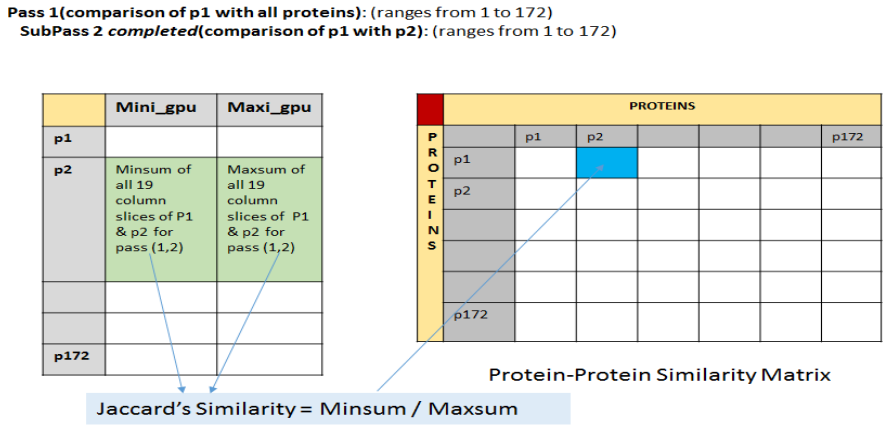
Fig 2: Process flow of CUDA architecture with GPU kernel and Thread Model

c. *Model of Parallel Design*

We have designed our code for doing comparison for comparison of all proteins with others, by taking each protein and comparing it to all others, by invocation of GPU. This logic applies for all proteins and subsequently gives result for all protein pairs comparison. As the upper triangular and lower triangular matrices of the similarity matrix is same, only upper triangular matrix is calculated, therefore saving computation time. The comparison logic by GPU for p1 with all other proteins are shown below:







Showing Result accumulation for one protein-all (p1-[p1 to p172]) comparison

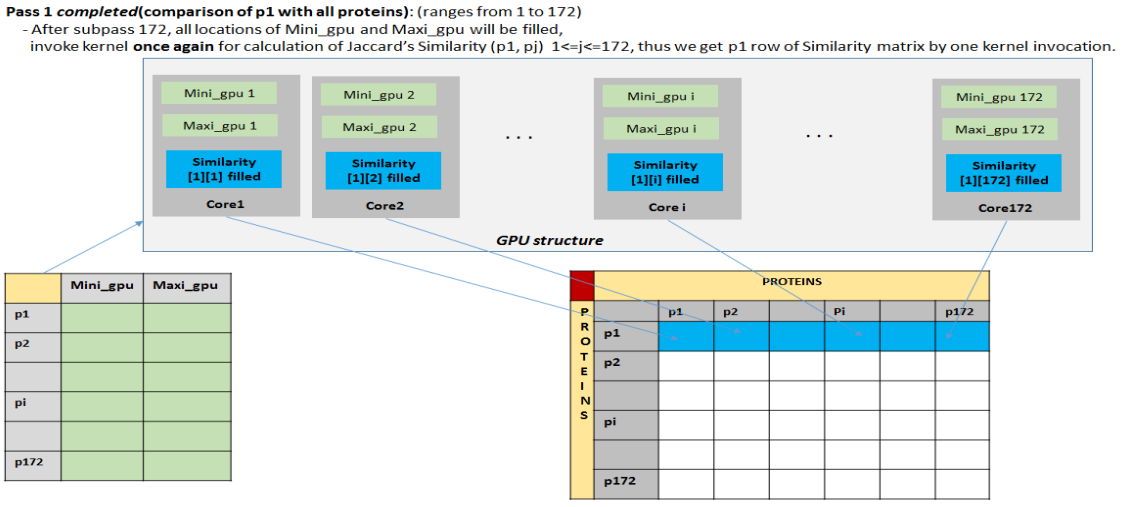


Fig 3: Flow design of p1-all pairs similarity calculation by GPU

D Algorithm

a. Protein-Key Matrix Formulation

Input: '*ProteinDataset'*
Output: a (n x m) matrix *PKmat* with Protein-Key weights

1. Declare *PKmat(n x m)* to hold the weights/ values of each key in each protein.
2. Get all protein files from the *ProteinDataset* and keep them in an indexed list *Protein* [P_1, P_2 ,…,P_n]
3. Get unique keys from the protein files and keep them in an indexed list *Keys* [K_1, K_2,…,K_m]
4. Formulate *PKmat* with the weights from the protein files corresponding to each key; PKmat [i][j] = k means "k is the value of key K_j in protein file P_i" as follows :
   For each protein file in *Protein* :
   a. read each line and split it in two parts.
   b. Use first part as key and second part as *weight* of that key in corresponding protein.
   c. Find index of corresponding protein and key in *Protein* and *Keys* files and use them as *row* and *column* index of *PKmat.*
   d. *PKmat[row][column] = weight.*
5. Save the result in a file and output time for calculation.

b. Similarity Calculation by CPU

Input: a (n x m) matrix *PKmat* with Protein-Key weights
Output: a (n x n) matrix *Similarity* with Protein-Protein similarity values

1. Declare *Similarity(n x n)* to hold the values of each Protein-Protein similarity.
2. For i=1 to n
   For j=1 to n
   Initialize *minsum* & *maxsum* to 0
   For k=1 to m
   If *PKmat*[i][k] less equals *PKmat*[j][k] :
   minsum = *minsum* + *PKmat*[i][k]
   maxsum = *maxsum* + *PKmat*[j][k]
   Else:
   minsum = *minsum* + *PKmat*[j][k]
   maxsum = *maxsum* + *PKmat*[i][k]
   result = *minsum*/*maxsum*
   similariy[i][j] = *result*
   similariy[j][i] = *result*
3. Save the result in a file and output time for calculation.
4. Return similarity matrix.

c. Similarity Calculation by GPU

For parallel implementation, we need to divide the matrix in submatrix. Our input matrix size is n ×m. Since the column size is a large number, we divide the each column into subcolumns and run GPU code on them. Then accumulate the result of submatrices to get the final result. This design principle is illustrated in figure 3.

Input: a (n x m) matrix *PKmat* with Protein-Key weights
Output: a (n x n) matrix *Similarity_gpu* with Protein-Protein similarity values

1. Declare *Similarity(n x n)* as empty to hold the values of each Protein-Protein similarity and copy it to *Similarity_gpu(n x n)* to be passel to kernel call.
2. Declare two temporary column vectors *mini*[n] & *maxi*[n] to store the *minsum* and *maxsum* for each protein-protein pair accumulated from each kernel call.
3. Initialize *mini* & *maxi* to 0
4. Copy them to *mini_gpu* & *maxi_gpu* for passing to kernel call.
5. Declare two temporary row vectors *a1*[p] & *a2*[p] to store one slice of the column of the *PKmat*.
6. Initialize *a1*& *a2* to 0
7. Copy them to *a1_gpu* & *a2 _gpu* for passing to kernel call.
8. For k=1 to n *#For each Protein row*
9. Set a variable x = k

```
Set device (mini_gpu) matrix with 0's
Set device (maxi_gpu) matrix with 0's
For l=: x to n   #For calculation only Upper Triangular Matrix
    Set a variable y = l
    Set pos = 0          #For calculate the Column number
    For i=1 to 19:       #Break each row into 19 Parts for kernel call
        For j=1 to 5689:  #For storing each part with 5689 columns into a1 and a2 Temp array
            if(pos<108091):
                a1[j] = PKmat_gpu[x][pos]
                a2[j] = PKmat_gpu[l][pos]
            else:
                a1[j] = 0
                a2[j] = 0
            pos = pos + 1

    drv.memcpy_htod(a1_gpu, a1) #Copying data of (a1) temp host matrix to (a1_gpu) device memory
    drv.memcpy_htod(a2_gpu, a2) #Copying data of (a2) temp host matrix to (a2_gpu) device memory
```

MinMax = mod.get_function("MinMax") *#Create MinMax function in host that call the MinMax Kernel on GPU or Device*

MinMax(a1_gpu, a2_gpu, mini_gpu, maxi_gpu, y, block = (1,1,1), grid = (1,1)) *#Send MinMax Kernel Arguments and Kernel Structure(as block, grid)*

result = mod.get_function("result") *#Create result function in host that call the result kernel on GPU or Device*

result(mini_gpu, maxi_gpu, Similarity_gpu, x, y, block = (172,1,1), grid = (1,1)) *#Send result Kernel Arguments and Kernel Structure(as block, grid)*

drv.memcpy_dtoh(Similarity, Similarity _gpu) *#Copy data of (Similarity _gpu) device memory to (Similarity) host memory*

10. Save the result in a file and output time for calculation.
11. Return Similarity matrix.

MinMax_calculation kernel:

Input: a1_gpu[5689], *a2_gpu* [5689] with Key column slices

Output: mini_gpu[172], *maxi_gpu* [172] with minsum and maxsum for a1_gpu and a2_gpu

1. Calculate thread index
2. If threadindex < 1
    a. Loop through each of a1_gpu and a2_gpu for elementwise comparison :
        i. maxsum = maxsum + larger of a1_gpu and a2_gpu
        ii. minsum = minsum + smaller of a1_gpu and a2_gpu
3. mini_gpu = mini_gpu + minsum
   maxi_gpu = maxi_gpu + maxsum
4. Return mini_gpu and maxi _gpu.

Result_accumulation kernel:

Input: mini_gpu[172], *maxi_gpu* [172] with minsum and maxsum for a1_gpu and a2_gpu

Output: Similarity_gpu(172 x 172), one row filled

1. Calculate thread index.
2. Similarity_gpu[protein_id][idx] = mini_gpu[idx]/maxi_gpu[idx]
   Similarity_gpu[idx][protein_id] = mini_gpu[idx]/maxi_gpu[idx]
3. Return Similarity_gpu.

d. CPU and GPU Result Comparison

Input: Similarity_gpu(172 x 172), Similarity_gpu(172 x 172), filled

Output: Equal or not Equal

1. cpu_sum = sum of all elements of Similarity matrix
2. gpu_sum = sum of all elements of Similarity_gpu matrix
3. if cpu_sum equals gpu_sum :
   results are Equal
   else
   results are not Equal

III. RESULTS & DISCUSIONS

a. Similarity Calculation by CPU result:

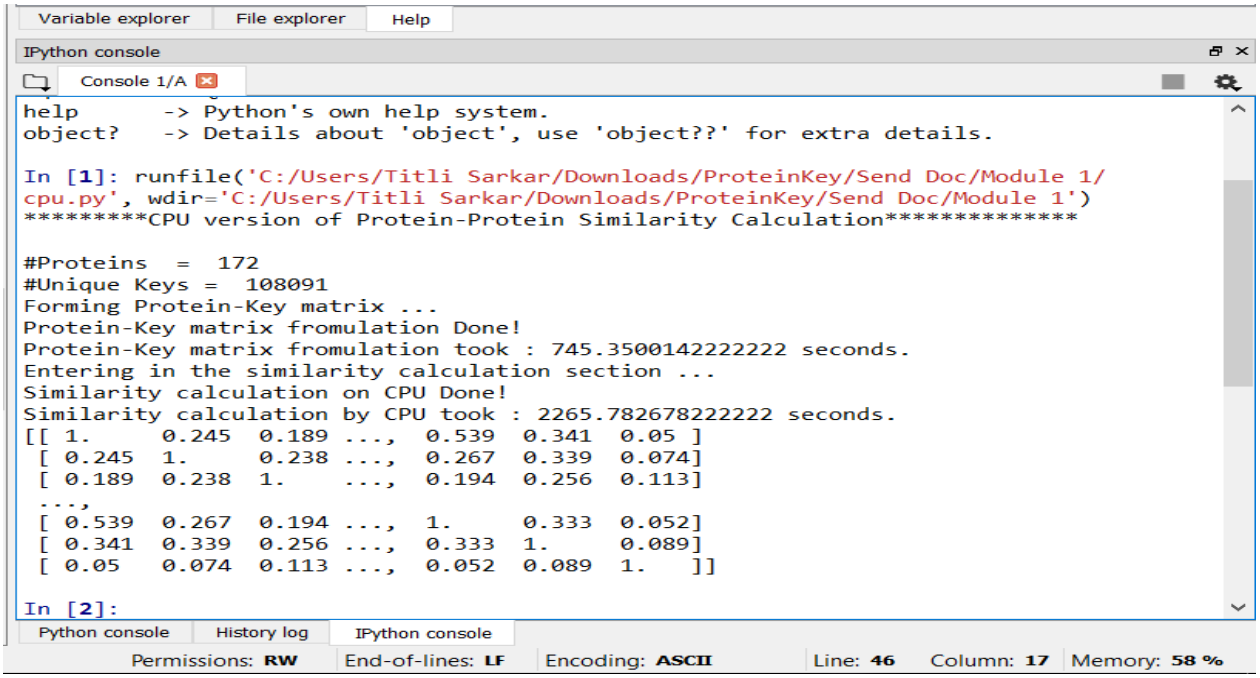


Fig 4: Result of similarity calculation on CPU with execution time

b. Similarity Calculation by GPU result:

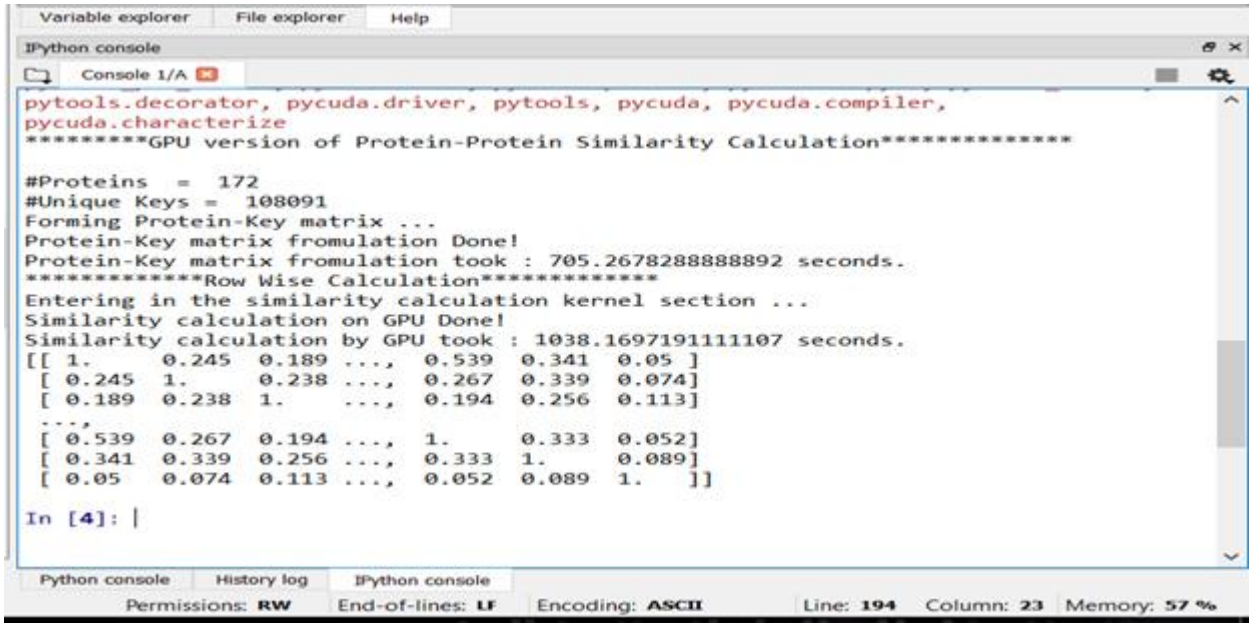


Fig 5: Result of similarity calculation on GPU with execution time

c. CPU and GPU results equal:

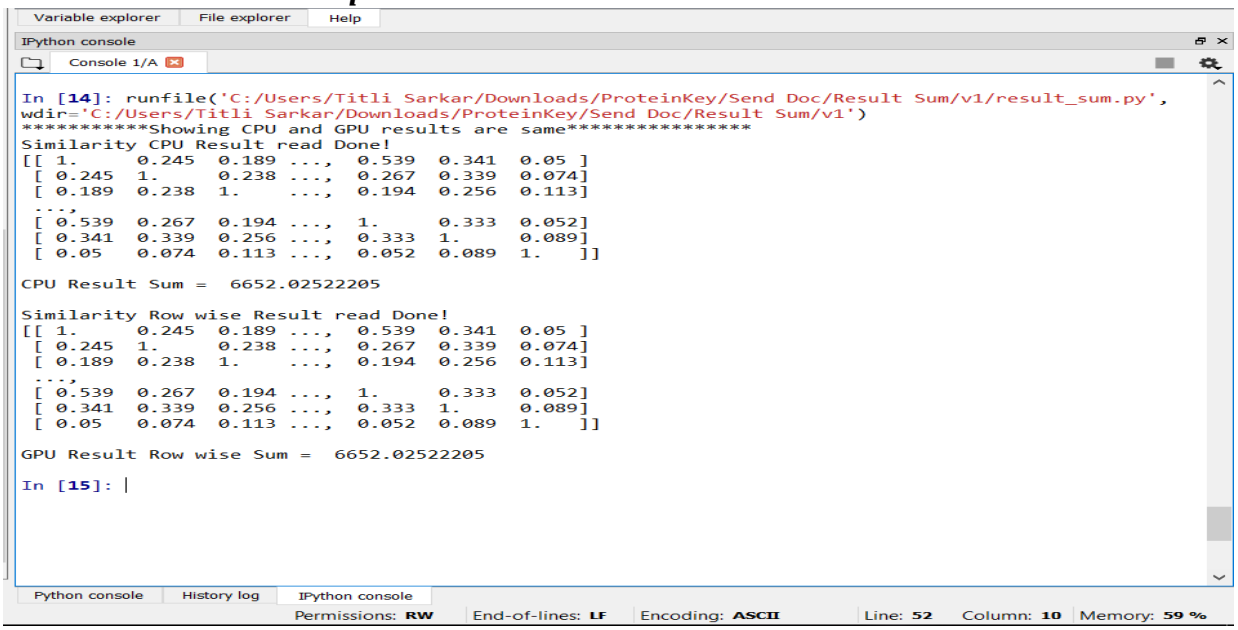


Fig 6: Showing all corresponding elements of CPU & GPU calculation are same by adding up all of them and printing sum

d. Performance Comparison:

Version	Similarity Calculation Time (seconds)
Sequential Implementation (CPU)	2265.78
Parallel Implementation(GPU)	1038.17

Fig 7: Speedup in terms of GPU

From the table, it is clear that GPU boosts the performance of calculation by running the application much faster with parallel processing. Here, GPU runs the code almost *2 times faster* than CPU code. It takes the help of GPU multi-cores which scales the computation in much lesser time than sequential one. The results of both version are stored in '*similarity_cpu.csv/txt*' and '*similarity_gpu_1.csv/txt*' files which can be easily compared manually. To be sure, all elements of the similarity matrices for each of CPU and GPU version are added up and found results are equal.

IV. CONCLUSION

In this project, we implement both sequential implementation and parallel implementation for finding the similarity between all combination of protein pairs for our data set. The challenge is to address huge dataset in an efficient manner. Forming a matrix from given dataset is difficult in handling in terms of storage but makes parallel implementation easier. The results show that parallel implementation scales the computation in much lesser time than sequential implementation.

V. FUTURE SCOPE

Protein structure studies assist in the investigation of protein-protein interactions and give researchers insight into the biological processes of the cell. Through the structure comparison of protein, it is possible to understand how similar they are, which information can be used further for different purposes.

The proposed algorithm of parallel invocation gives almost double speedup on sequential version. Different approach of parallel algorithms can be used to invoke kernel more efficiently and better performance.

References:

[1] Sing, Sumi, "*Protein 3-D Structure Representation for Alignment-Free Comparison and Structural Motif Discovery of Proteins, and Hierarchical Protein Classification*", University of Louisiana at Lafayette, ProQuest Dissertations Publishing, 2015. 10003582

[2] https://developer.nvidia.com/cuda-education-training, Last accessed Dec 01, 2016

[3] https://docs.python.org/3/tutorial/, Last accessed Dec 01, 2016

[4] https://www.ibm.com/developerworks/community/blogs/jfp/entry/Installing_PyCUDA_On_Anaconda_For_Windows?lang=en

List of Figures

Appendix:

cpu.py

```
# -*- coding: utf-8 -*-
'''
This code calculates protein-protein similarity among 172 proteins with
108091 keys, in sequential batch mode.

Formula used for similarity calculation: Jaccard's Similarity Coefficient.

The input is available as a folder/directory of '.keys' files each of which
contains a set of key-value pairs.

First, a Protein-Key matrix is formed from these files with Protein files as
rows and list of unique keys as columns.
Then similarity among all Protein(i,j) pairs are calculated using Jaccard's
similarity coefficient and the outputs are
stored in a  Protein-Protein similarity matrix.
'''

import sys
sys.path.append('c:\\program files\\anaconda3\\lib\\site-packages')
import glob, os
import os, os.path
import csv
import operator
import numpy as np
import pandas as pd
import re
import time

start_time = time.clock()
#Getting all Protein files and saving them in a list'Protein[]' ---------->
Protein =[]
os.chdir("F:\\Studies\\Ph.D\\Ph.D Work\\ProteinDataSet")
for file in glob.glob("*.keys"):
        Protein.append(file)
no_of_proteins = len(Protein)
print("#Proteins = ",no_of_proteins)

#Sorting all Protein files ------------->
for p in Protein:
        with open(p, "r") as p_file:
                filename = "F:\\Studies\\Ph.D\\Ph.D
Work\\ProteinDataSet\\"+p_file.name
                f = open(filename, "r")
                lines = f.readlines()
                lines.sort(key=lambda a_line: a_line.split()[0])
                f.close()
#print("File sorting Done!")

#Getting all unique keys in a  sorted list 'Keys[]' ---------------->
Keys = []
```

```
for pt in Protein:
        with open(pt, "r") as p_file:
                filename = "F:\\Studies\\Ph.D\\Ph.D
Work\\ProteinDataSet\\"+p_file.name
                f = open(filename, "r")
                for line in f:
                        cntnt = line.split()
                        res = list(map(int, cntnt))
                        item = res[0]
                        Keys.append(item) #add only 1st number of each line
of the files as keys
                f.close()

Keys = np.unique(Keys)
no_unq_keys = Keys.shape[0]
np.save("F:\\Studies\\Ph.D\\Ph.D Work\\ProteinDataSet\\keys", Keys);
print("#Unique Keys=",no_unq_keys)
#print(" Getting unique keys Done!")

#Forming Protein-Key matrix ------------------------------------------------>
PKmat = np.zeros(shape=(no_of_proteins,no_unq_keys))
#np.save("F:\\Studies\\Ph.D\\Ph.D Work\\ProteinDataSet\\PKmat", PKmat);
for p in Protein:
        with open(p, "r") as pt_file:
                fname = "F:\\Studies\\Ph.D\\Ph.D
Work\\ProteinDataSet\\"+pt_file.name
                fl = open(fname, "r")
                for line in fl:
                        content = line.split()
                        results = list(map(int, content))
                        r = Protein.index(p)
                        var=np.where(Keys==results[0])
                        c=var[0][0]
                        PKmat[r][c] = results[1]
                fl.close()

print ("Protein-Key matrix formulation took :",time.clock() - start_time,
"seconds.")
np.savetxt("F:\\Studies\\Ph.D\\Ph.D Work\\ProteinDataSet\\PKmat.txt", PKmat,
delimiter=' ')
df = pd.DataFrame(PKmat, columns=Keys)
df.to_csv('F:\\Studies\\Ph.D\\Ph.D Work\\ProteinDataSet\\pkmat.csv')
#print("PK matrix formulation done!")

#Calculation of similarity for each protein -------------------------->
start_time = time.clock()
similariy = np.zeros(shape=(no_of_proteins,no_of_proteins))
i = 0
for row in PKmat:
    j = 0
    for subrow in PKmat:
        max = np.maximum(row , subrow)
        maxsum = np.sum(max)
        min = np.minimum(row , subrow)
        minsum = np.sum(min)
        sim = maxsum/minsum
        similariy[i][j] = sim
```

```
        j += 1
    i += 1
#print (similariy)
dfsim = pd.DataFrame(similariy, columns=Protein)
dfsim.to_csv('F:\\Studies\\Ph.D\\Ph.D Work\\ProteinDataSet\\Similarity.csv')
print ("Similarity calculation Done!")
print ("Similarity calculation took :",time.clock() - start_time, "seconds.")
```

gpu.py

```
# -*- coding: utf-8 -*-
'''
This code calculates protein-protein similarity among 172 proteins with
108091 keys,in parallel batch mode.
'''

from __future__ import print_function
from __future__ import absolute_import
from pycuda.compiler import SourceModule

import pycuda.driver as drv
import pycuda.autoinit
import numpy as np
import time
import pandas as pd
import glob, os
import os, os.path
import csv
import operator
import re


mod = SourceModule("""
      __global__ void MinMax(double a1[5689], double a2[5689], double
mini_gpu[172], double maxi_gpu[5689], int y)
      {
            int idx = threadIdx.x;
            double min = 0.0, max = 0.0;
            if(idx < 1)
            {
                  for(int i=0; i<5689; i++)
                  {
                        if( a1[i] <= a2[i])
                        {
                              min = min + a1[i];
                              max = max + a2[i];
                        }
                        else
                        {
                              min = min + a2[i];
                              max = max + a1[i];
                        }
                  }
                  mini_gpu[y] = mini_gpu[y] + min;
                  maxi_gpu[y] = maxi_gpu[y] + max;
            }
```

```
      }
      __global__ void result(double mini_gpu[172], double maxi_gpu[172],
double res_gpu[172][172], int x, int y)
      {
            int idx = threadIdx.x;
            if(idx>=y && idx<172)
            {
                  res_gpu[x][idx] =
(double)mini_gpu[idx]/(double)maxi_gpu[idx];
                  res_gpu[idx][x] =
(double)mini_gpu[idx]/(double)maxi_gpu[idx];

            }
      }
      """)
print("*********GPU version of Protein-Protein Similarity
Calculation*************\n")
#Getting all Protein files and saving them in a list 'Protein[]'
Protein=[]

#Set the ProteinKet data set folder path
os.chdir("F:\\Studies\\Ph.D\\Ph.D Work\\ProteinDataSet")

for file in glob.glob("*.keys"):
      Protein.append(file)

no_of_proteins = len(Protein)
print("#Proteins  = ", no_of_proteins)

#Sorting all Protein files ------------->
for p in Protein:
      with open(p, "r") as p_file:  #open p position file of Protein[] in
Read format
            filename = ""+p_file.name
            #print(" File --->", filename)
            f = open(filename, "r")
            lines = f.readlines()
            #print(lines)
            lines.sort(key=lambda a_line: a_line.split()[0])
            f.close()

#Getting all unique keys in a sorted list 'Keys[]' ------------>
keys = []
for pt in Protein:
      with open(pt, "r") as p_file:
            filename = ""+p_file.name
            #print(filename)
            f = open(filename, "r")
            for line in f:
                  cntnt = line.split()
                  res = list(map(int, cntnt))
                  item = res[0]
                  keys.append(item) #add only 1st number of each line of the
files as keys
            f.close()

keys = np.unique(keys)
```

```
no_unq_keys = keys.shape[0]
np.save("keys", keys)   #save the keys into Keys file
print("#Unique Keys = ", no_unq_keys)

#Forming Protein-Key Matrix ------------------------>
print("Forming Protein-Key matrix ...")
start_time = time.clock()
PKmat_gpu = np.zeros(shape=(no_of_proteins,no_unq_keys))
for p in Protein:
      with open(p, "r") as pt_file:
            fname = ""+pt_file.name
            fl = open(fname, "r")
            for line in fl:
                  content = line.split()
                  results = list(map(int, content))
                  r = Protein.index(p)
                  var = np.where(keys==results[0])
                  c = var[0][0]
                  PKmat_gpu[r][c] = results[1]
            f.close()
print("Protein-Key matrix fromulation Done!")
print ("Protein-Key matrix fromulation took :",time.clock() - start_time,
"seconds.")
np.savetxt("PKmat_gpu.txt", PKmat_gpu, delimiter=' ')
df = pd.DataFrame(PKmat_gpu, columns=keys)
df.to_csv('pkmat_gpu.csv')
print("*************Row Wise Calculation*************")

r = np.shape(PKmat_gpu)[0]          #No. of Proteins
row = np.int32(r)
c = np.shape(PKmat_gpu)[1]          #No. of Keys
col = np.int32(c)

#This is the Device Kernel Section that calculate the Similarity value
#This Machine GPU having following details :
#                              Device Name : GeForce GTX 960
#                              Total Global Memory : 979 MB
#                              maximum Shared memory in a block : 48 KB
#                              Maximum Threads per Block : 1024

#This Kernel Section calculate the Similarity matrix in Row wise

mini = np.zeros(row)          #Create Mini Column martix on Host Memory
mini_gpu = drv.mem_alloc(mini.nbytes)     #Allocate Device memory for Mini
Column matrix
maxi = np.zeros(row)          #Create Maxi Column matrix on Host Memory
maxi_gpu = drv.mem_alloc(maxi.nbytes)     #Allocate Device memory for Maxi
Column matrix
res = np.zeros(shape=(row,row))           #Create Result matrix on Host
Memory to store the Similarirty value
res_gpu = drv.mem_alloc(res.nbytes) #Allocate Device memory for Result matrix

drv.memcpy_htod(res_gpu, res)       #Set Device Result matrix with 0's

#Create two Temp row matrix for GPU calculation and allocate there
corrosponding memory into Device
a1 = np.zeros(5689)
```

```
a2 = np.zeros(5689)
a1_gpu = drv.mem_alloc(a1.nbytes)
a2_gpu = drv.mem_alloc(a2.nbytes)


print("Entering in the similarity calculation kernel section ...")
start_time = time.clock()           #Start the timer
i = 0
val = np.int32(i)
for k in range(0,row):        #For each row
      x = np.int32(k)
      drv.memcpy_htod(mini_gpu, mini)     #For each row Set device (mini_gpu)
matrix with 0's
      drv.memcpy_htod(maxi_gpu, maxi)     #For each row Set device (maxi_gpu)
matrix with 0's

      for l in range(x,row):        #For calculate only Upper Triangular
Matrix
            y = np.int32(l)
            pos = 0                             #For calculate the Column
number
            for i in range(0,19):   #Break rows into 19 Parts for kernel call
                  for j in range(0,5689): #For storing each part with 5689
columns into a1 and a2 Temp array
                        if(pos<108091):
                              a1[j] = PKmat_gpu[x][pos]
                              a2[j] = PKmat_gpu[l][pos]
                        else:
                              a1[j] = 0
                              a2[j] = 0
                        pos = pos + 1

                  drv.memcpy_htod(a1_gpu, a1)         #Copying data of (a1)
temp host matrix to (a1_gpu) device memory
                  drv.memcpy_htod(a2_gpu, a2)         #Copying data of (a2)
temp host matrix to (a2_gpu) device memory

                  MinMax = mod.get_function("MinMax")       #Create MinMax
function in host that call the MinMax Kernel on GPU or Device
                  MinMax(a1_gpu, a2_gpu, mini_gpu, maxi_gpu, y, block =
(1,1,1), grid = (1,1))  #Send MinMax Kernel Arguments and Kernel Structure(as
block, grid)
            #print(y)

            result = mod.get_function("result")             #Create result
function in host that call the result kernel on GPU or Device
            result(mini_gpu, maxi_gpu, res_gpu, x, y, block = (172,1,1), grid
= (1,1))                #Send result Kernel Arguments and Kernel Structure(as
block, grid)
      #print(x)
print ("Similarity calculation on GPU Done!")
print ("Similarity calculation by GPU took :",time.clock() - start_time,
"seconds.")       #Stop timer and Calculate total time

drv.memcpy_dtoh(res, res_gpu)                   #Copy data of (res_gpu)
device memory to (res) host memory
```

```
np.savetxt("similarity_gpu-1.txt", res, delimiter=' ')            #Save
results in similarity_gpu.txt file

np.set_printoptions(precision=3)
print(res)                            #Print Result Matrix

df1 = pd.DataFrame(res)       #Save Result in .csv file
df1.to_csv('similarity_gpu-1.csv')
```

result_sum.py

```
# -*- coding: utf-8 -*-
'''
This code sums up all the elements of similarity matrices generated by
sequential and parallel code separately and compare the sums if they are equal
or not.
'''

import sys
sys.path.append('c:\\program files\\anaconda3\\lib\\site-packages')

import glob, os
import os, os.path
import csv
import operator
import numpy as np
import pandas as pd

print("***********Showing CPU and GPU results are same****************")
os.chdir("F:\\Studies\\Ph.D\\Ph.D Work\\ProteinDataSet\\")
#For Module 1------->
cpu_result_sum = np.loadtxt("similarity_cpu.txt")
print("Similarity CPU Result read Done!")
p = np.shape(cpu_result_sum)[0]
q = np.shape(cpu_result_sum)[1]
cpu_row = np.int32(p)
cpu_col = np.int32(q)

np.set_printoptions(precision=3)
print(cpu_result_sum)

cpu_sum = 0.0
gpu_sum1 = 0.0
gpu_sum2 = 0.0

for i in range(0,cpu_row):
      for j in range(0,cpu_col):
            cpu_sum = cpu_sum + cpu_result_sum[i][j]

print("")
print("CPU Result Sum = ",cpu_sum)

#For Module 2 ------------>
#For Row wise calculation
gpu_result_sum1 = np.loadtxt("similarity_gpu-1.txt")
print("\nSimilarity Row wise Result read Done!")
```

```
r = np.shape(gpu_result_sum1)[0]
s = np.shape(gpu_result_sum1)[1]
gpu_row = np.int32(r)
gpu_col = np.int32(s)

np.set_printoptions(precision=3)
print(gpu_result_sum1)


for i in range(0, gpu_row):
      for j in range(0, gpu_col):
            gpu_sum1 = gpu_sum1 + gpu_result_sum1[i][j]

print("\nGPU Result Row wise Sum = ",gpu_sum1)
```