# Ass. 1 DFS

#include <iostream> #include <vector> #include <stack> #include <omp.h> using namespace std;

const int MAX = 100000; vector<int> graph[MAX]; bool visited[MAX];

void dfs(int node)

{

stack<int> s; s.push(node); while (!s.empty())

if (temp->right == NULL)

{

temp->right = new node; temp->right->left = NULL; temp->right->right = NULL; temp->right->data = data; return root;

}

else

{

q.push(temp->right);

}

}

}

{

int curr\_node = s.top(); if (!visited[curr\_node])

{

visited[curr\_node] = true; s.pop();

cout << curr\_node << " "; #pragma omp parallel for

for (int i = 0; i < graph[curr\_node].size(); i++)

{

int adj\_node = graph[curr\_node][i]; if (!visited[adj\_node])

{

s.push(adj\_node);

}

}

}

}

}

int main()

{

int n, m, start\_node;

cout << "Enter no. of Node,no. of Edges and Starting Node of graph:\n"; cin >> n >> m >> start\_node;

// n: node,m:edges

cout << "Enter pair of node and edges:\n"; for (int i = 0; i < m; i++)

{

int u, v;

cin >> u >> v;

// u and v: Pair of edges graph[u].push\_back(v); graph[v].push\_back(u);

}

#pragma omp parallel for for (int i = 0; i < n; i++)

{

visited[i] = false;

}

dfs(start\_node); return 0;

}

# BFS

#include <iostream> #include <stdlib.h> #include <queue> using namespace std; class node

{

public:

node \*left, \*right; int data;

};

class Breadthfs

{

public:

node \*insert(node \*, int); void bfs(node \*);

};

node \*insert(node \*root, int data)

// inserts a node in tree

{

if (!root)

{

root = new node; root->left = NULL; root->right = NULL; root->data = data; return root;

}

queue<node \*> q; q.push(root); while (!q.empty())

{

node \*temp = q.front(); q.pop();

if (temp->left == NULL)

{

temp->left = new node; temp->left->left = NULL; temp->left->right = NULL; temp->left->data = data; return root;

}

else

{

q.push(temp->left);

}

void bfs(node \*head)

{

queue<node \*> q; q.push(head);

int qSize;

while (!q.empty())

{

qSize = q.size(); #pragma omp parallel for

// creates parallel threads for (int i = 0; i < qSize; i++)

{

node \*currNode; #pragma omp critical

{

currNode = q.front(); q.pop();

cout << "\t" << currNode->data;

} // prints parent node #pragma omp critical

{

if (currNode->left) // push parent's left node in queue q.push(currNode->left);

if (currNode->right) q.push(currNode->right);

} // push parent's right node in queue

}

}

}

int main()

{

node \*root = NULL; int data;

char ans; do

{

cout << "\n enter data=>"; cin >> data;

root = insert(root, data);

cout << "do you want insert one more node?"; cin >> ans;

} while (ans == 'y' || ans == 'Y'); bfs(root);

return 0;

}

# ASS>2 MERGE

#include <iostream> #include <vector> #include <omp.h>

void merge(std::vector<int>& arr, int left, int mid, int right) { int i, j, k;

int n1 = mid - left + 1; int n2 = right - mid;

std::vector<int> leftArr(n1), rightArr(n2); for (i = 0; i < n1; i++)

leftArr[i] = arr[left + i]; for (j = 0; j < n2; j++)

rightArr[j] = arr[mid + 1 + j]; i = 0;

j = 0;

k = left;

while (i < n1 && j < n2) {

if (leftArr[i] <= rightArr[j]) { arr[k] = leftArr[i];

i++;

} else {

arr[k] = rightArr[j]; j++;

} k++;

}

while (i < n1) { arr[k] = leftArr[i]; i++;

k++;

}

while (j < n2) { arr[k] = rightArr[j]; j++;

k++;

}

}

void parallelMergeSort(std::vector<int>& arr, int left, int right) { if (left < right) {

int mid = left + (right - left) / 2; #pragma omp parallel sections

{

#pragma omp section

{

parallelMergeSort(arr, left, mid);

}

#pragma omp section

{

parallelMergeSort(arr, mid + 1, right);

}

}

merge(arr, left, mid, right);

}

}

int main() {

std::vector<int> numbers = {5, 2, 8, 12, 3}; std::cout << "Before sorting: ";

for (int num : numbers) { std::cout << num << " ";

}

std::cout << std::endl; parallelMergeSort(numbers, 0, numbers.size() - 1); std::cout << "After sorting: ";

for (int num : numbers) { std::cout << num << " ";

}

std::cout << std::endl; return 0;

}

BUbble

#include <iostream> #include <vector> #include <omp.h>

void parallelBubbleSort(std::vector<int>& arr) { int n = arr.size();

bool swapped; #pragma omp parallel

{

do {

swapped = false; #pragma omp for

for (int i = 0; i < n - 1; i++) { if (arr[i] > arr[i + 1]) { std::swap(arr[i], arr[i + 1]); swapped = true;

}

}

} while (swapped);

}

}

int main() {

std::vector<int> numbers = {5, 2, 8, 12, 3}; std::cout << "Before sorting: ";

for (int num : numbers) { std::cout << num << " ";

}

std::cout << std::endl; parallelBubbleSort(numbers); std::cout << "After sorting: "; for (int num : numbers) { std::cout << num << " ";

}

std::cout << std::endl; return 0;

}

# ASS. 3

**Min max sum** #include <iostream> #include <vector> #include <omp.h> #include <climits> using namespace std;

void min\_reduction(vector<int> &arr)

{

int min\_value = INT\_MAX;

#pragma omp parallel for reduction(min : min\_value) for (int i = 0; i < arr.size(); i++)

{

if (arr[i] < min\_value)

{

min\_value = arr[i];

}

}

cout << "Minimum value: " << min\_value << endl;

}

void max\_reduction(vector<int> &arr)

{

#pragma omp parallel for reduction(+ : sum) for (int i = 0; i < arr.size(); i++)

{

sum += arr[i];

}

cout << "Sum: " << sum << endl;

}

void average\_reduction(vector<int> &arr)

{

int sum = 0;

#pragma omp parallel for reduction(+ : sum) for (int i = 0; i < arr.size(); i++)

{

sum += arr[i];

}

cout << "Average: " << (double)sum / arr.size() << endl;

}

int main()

{

vector<int> arr; arr.push\_back(5); arr.push\_back(2); arr.push\_back(9); arr.push\_back(1); arr.push\_back(7); arr.push\_back(6); arr.push\_back(8); arr.push\_back(3); arr.push\_back(4); min\_reduction(arr); max\_reduction(arr); sum\_reduction(arr); average\_reduction(arr);

}

# Ass. 6

# bouston housing

import numpy as np

from tensorflow.keras.models import Sequential from tensorflow.keras.layers import Dense, Dropout from tensorflow.keras.optimizers import RMSprop from tensorflow.keras.datasets import mnist

import matplotlib.pyplot as plt from sklearn import metrics

# Load the OCR dataset

# The MNIST dataset is a built-in dataset provided by Keras.

(x\_train, y\_train), (x\_test, y\_test) = mnist.load\_data()

# X\_train and X\_test are our array of images while y\_train and y\_test are our array of labels for each image.

plt.imshow(x\_train[0], cmap='gray') plt.show()

print(x\_train[0])

print("X\_train shape", x\_train.shape) print("y\_train shape", y\_train.shape) print("X\_test shape", x\_test.shape) print("y\_test shape", y\_test.shape)

x\_train = x\_train.reshape(60000, 784) x\_test = x\_test.reshape(10000, 784) x\_train = x\_train.astype('float32') x\_test = x\_test.astype('float32') x\_train /= 255

x\_test /= 255

# Convert class vectors to binary class matrices num\_classes = 10

y\_train = np.eye(num\_classes)[y\_train] y\_test = np.eye(num\_classes)[y\_test]

# Define the model architecture model = Sequential()

model.add(Dense(512, activation='relu', input\_shape=(784,))) # The input\_shape argument is passed to the foremost layer. It comprises of a tuple shape, model.add(Dropout(0.2)) # DROP OUT RATIO 20%

model.add(Dense(512, activation='relu')) #returns a sequence of vectors of dimension 512

model.add(Dropout(0.2)) model.add(Dense(num\_classes, activation='softmax'))

int max\_value = INT\_MIN;

#pragma omp parallel for reduction(max : max\_value) for (int i = 0; i < arr.size(); i++)

{

if (arr[i] > max\_value)

{

max\_value = arr[i];

}

}

cout << "Maximum value: " << max\_value << endl;

}

void sum\_reduction(vector<int> &arr)

{

int sum = 0;

# Compile the model

model.compile(loss='categorical\_crossentropy', # for a multi-class classification problem

optimizer=RMSprop(), metrics=['accuracy'])

# Train the model

batch\_size = 128 # batch\_size argument is passed to the layer to define a batch size for the inputs.

epochs = 20

history = model.fit(x\_train, y\_train,

batch\_size=batch\_size, epochs=epochs,

verbose=1, # verbose=1 will show you an animated progress bar eg. [==========]

validation\_data=(x\_test, y\_test))

# Evaluate the model

score = model.evaluate(x\_test, y\_test, verbose=0) print('Test loss:', score[0])

print('Test accuracy:', score[1])

# Ass. 7

#multiclass classification

import numpy as np import pandas as pd

#loading imdb data with most frequent 10000 words from keras.datasets import imdb

(X\_train, y\_train), (X\_test, y\_test) = imdb.load\_data(num\_words=10000) # you may take top 10,000 word frequently review of movies

#consolidating data for EDA(exploratory data analysis: involves gathering all the relevant data into one place and preparing it for analysis )

data = np.concatenate((X\_train, X\_test), axis=0) label = np.concatenate((y\_train, y\_test), axis=0)

print("Review is ",X\_train[5]) # series of no converted word to vocabulory associated with index

print("Review is ",y\_train[5]) # 0 indicating a negative review and 1 indicating a positive review.

vocab=imdb.get\_word\_index() #The code you provided retrieves the word index for the IMDB dataset

print(vocab)

data #data is a numpy array that contains all the text data from the IMDB dataset, both the training and testing sets.

label #label is a numpy array that contains all the sentiment labels from the IMDB dataset, both the training and testing sets.0 indicates a negative review and 1 indicates a positive review.

X\_train.shape X\_test.shape y\_train

y\_test # y\_test is 25000, which indicates that it contains 25000 sentiment labels, one for each review in the testing set.

def vectorize(sequences, dimension = 10000):

# Create an all-zero matrix of shape (len(sequences), dimension) results = np.zeros((len(sequences), dimension))

for i, sequence in enumerate(sequences):

results[i, sequence] = 1 return results

test\_x = data[:10000] test\_y = label[:10000] train\_x = data[10000:] train\_y = label[10000:]

test\_x test\_y train\_x train\_y

print("Categories:", np.unique(label))

print("Number of unique words:", len(np.unique(np.hstack(data))))

length = [len(i) for i in data]

print("Average Review length:", np.mean(length)) print("Standard Deviation:", round(np.std(length)))

print("Label:", label[0]) print(data[0])

index = imdb.get\_word\_index()

reverse\_index = dict([(value, key) for (key, value) in index.items()])

decoded = " ".join( [reverse\_index.get(i - 3, "#") for i in data[0]] ) #The purpose of subtracting 3 from i is to adjust the indice,# to indicate that the index was not found. print(decoded)

index reverse\_index decoded

#Adding sequence to data

# Vectorization is the process of converting textual data into numerical vectors and is a process that is usually applied once the text is cleaned.

data = vectorize(data)

label = np.array(label).astype("float32")

Data label

import seaborn as sns sns.set(color\_codes=True) import matplotlib.pyplot as plt

labelDF=pd.DataFrame({'label':label}) sns.countplot(x='label', data=labelDF)

# For below analysis it is clear that data has equel distribution of sentiments.This will help us building a good model.

# Creating train and test data set

from sklearn.model\_selection import train\_test\_split

X\_train, X\_test, y\_train, y\_test = train\_test\_split(data,label, test\_size=0.20, random\_state=1)

X\_train.shape X\_test.shape

from keras.utils import to\_categorical from keras import models

from keras import layers model = models.Sequential()

model.add(layers.Dense(50, activation = "relu", input\_shape=(10000, )))

model.add(layers.Dropout(0.3, noise\_shape=None, seed=None)) model.add(layers.Dense(50, activation = "relu")) #ReLU" stands for Rectified Linear Unit, and it is a commonly used activation function in neural networks. model.add(layers.Dropout(0.2, noise\_shape=None, seed=None)) model.add(layers.Dense(50, activation = "relu"))

# Output- Layer

model.add(layers.Dense(1, activation = "sigmoid")) #adds another Dense layer to the model, but with a single neuron instead of 50,i.e. out put layer,it produces the output predictions of the model.

model.summary()

import tensorflow as tfcallback = tf.keras.callbacks.EarlyStopping(monitor='loss', patience=3)

model.compile( optimizer = "adam",

loss = "binary\_crossentropy", metrics = ["accuracy"]

)

results = model.fit( X\_train, y\_train, epochs= 2,

batch\_size = 500,

validation\_data = (X\_test, y\_test), callbacks=[callback]

)

# Let's check mean accuracy of our model print(np.mean(results.history["val\_accuracy"]))

# list all data in history print(results.history.keys())

# summarize history for accuracy plt.plot(results.history['accuracy'])

plt.plot(results.history['val\_accuracy']) #Plots the validation accuracy of the model at each epoch.

plt.title('model accuracy') plt.ylabel('accuracy') plt.xlabel('epoch')

plt.legend(['train', 'test'], loc='upper left') plt.show()

# summarize history for loss plt.plot(results.history['loss']) plt.plot(results.history['val\_loss']) plt.title('model loss') plt.ylabel('loss') plt.xlabel('epoch')

plt.legend(['train', 'test'], loc='upper left') plt.show()

model.predict(X\_test)

# Ass. 8

Fashion dataset

import tensorflow as tf

import matplotlib.pyplot as plt from tensorflow import keras import numpy as np

(x\_train, y\_train), (x\_test, y\_test) = keras.datasets.fashion\_mnist.load\_data() plt.imshow(x\_train[1])

plt.imshow(x\_train[6]) plt.imshow(x\_train[0])

x\_train = x\_train.astype('float32') / 255.0 x\_test = x\_test.astype('float32') / 255.0 x\_train = x\_train.reshape(-1, 28, 28, 1)

x\_test = x\_test.reshape(-1, 28, 28, 1) x\_train.shape

x\_test.shape y\_train.shape y\_test.shape

model = keras.Sequential([

keras.layers.Conv2D(32, (3,3), activation="relu", input\_shape=(28,28,1)), keras.layers.MaxPooling2D((2,2)),

keras.layers.Dropout(0.25), keras.layers.Conv2D(64, (3,3), activation="relu"), keras.layers.MaxPooling2D((2,2)), keras.layers.Dropout(0.25), keras.layers.Conv2D(128, (3,3), activation="relu"), keras.layers.Flatten(),

keras.layers.Dense(10, activation="softmax")

])

model.summary()

model.compile(optimizer="adam", loss="sparse\_categorical\_crossentropy", metrics=['accuracy'])

history = model.fit(x\_train, y\_train, epochs = 20, validation\_data = (x\_test, y\_test)) test\_loss, test\_acc = model.evaluate(x\_test, y\_test)

print("Test accuracy:", test\_acc)

# Ass. 4

Addition

#include <iostream> #include <cuda.h>

#define N 1000000 // Number of elements in the vectors

\_global\_

void vectorAddition(const float\* a, const float\* b, float\* result) { int tid = blockIdx.x \* blockDim.x + threadIdx.x;

if (tid < N) {

result[tid] = a[tid] + b[tid];

}

Multi.

#include <iostream> #include <cuda.h>

#define N 1024 // Matrix size

// Kernel function for matrix multiplication

\_global\_

void matrixMultiplication(const float\* A, const float\* B, float\* C) { int row = blockIdx.y \* blockDim.y + threadIdx.y;

int col = blockIdx.x \* blockDim.x + threadIdx.x;

if (row < N && col < N) { float sum = 0.0f;

for (int k = 0; k < N; k++) {

sum += A[row \* N + k] \* B[k \* N + col];

}

C[row \* N + col] = sum;

}

}

int main() {

// Initialize host matrices

float\* hostA = new float[N \* N]; float\* hostB = new float[N \* N]; float\* hostC = new float[N \* N];

// Initialize input matrices for (int i = 0; i < N \* N; i++) {

hostA[i] = static\_cast<float>(i); hostB[i] = static\_cast<float>(2 \* i);

}

// Declare device matrices float\* deviceA;

float\* deviceB; float\* deviceC;

}

int main() {

// Initialize host vectors float\* hostA = new float[N]; float\* hostB = new float[N];

float\* hostResult = new float[N];

// Initialize input vectors for (int i = 0; i < N; i++) {

hostA[i] = static\_cast<float>(i); hostB[i] = static\_cast<float>(2 \* i);

}

// Declare device vectors float\* deviceA;

float\* deviceB; float\* deviceResult;

// Allocate memory on the device cudaMalloc((void\*\*)&deviceA, N \* sizeof(float)); cudaMalloc((void\*\*)&deviceB, N \* sizeof(float)); cudaMalloc((void\*\*)&deviceResult, N \* sizeof(float));

// Copy input vectors from host to device

cudaMemcpy(deviceA, hostA, N \* sizeof(float), cudaMemcpyHostToDevice); cudaMemcpy(deviceB, hostB, N \* sizeof(float), cudaMemcpyHostToDevice);

// Allocate memory on the device cudaMalloc((void\*\*)&deviceA, N \* N \* sizeof(float)); cudaMalloc((void\*\*)&deviceB, N \* N \* sizeof(float)); cudaMalloc((void\*\*)&deviceC, N \* N \* sizeof(float));

// Copy input matrices from host to device

cudaMemcpy(deviceA, hostA, N \* N \* sizeof(float), cudaMemcpyHostToDevice); cudaMemcpy(deviceB, hostB, N \* N \* sizeof(float), cudaMemcpyHostToDevice);

// Define grid and block dimensions dim3 threadsPerBlock(16, 16);

dim3 blocksPerGrid((N + threadsPerBlock.x - 1) / threadsPerBlock.x, (N + threadsPerBlock.y - 1) / threadsPerBlock.y);

// Launch the kernel

matrixMultiplication<<<blocksPerGrid, threadsPerBlock>>>(deviceA, deviceB, deviceC);

// Copy result matrix from device to host

cudaMemcpy(hostC, deviceC, N \* N \* sizeof(float), cudaMemcpyDeviceToHost);

// Print the top-left 10x10 elements of the result matrix for (int i = 0; i < 10; i++) {

for (int j = 0; j < 10; j++) {

std::cout << hostC[i \* N + j] << " ";

}

std::cout << std::endl;

// Define grid and block dimensions int threadsPerBlock = 256;

int blocksPerGrid = (N + threadsPerBlock - 1) / threadsPerBlock;

// Launch the kernel

vectorAddition<<<blocksPerGrid, threadsPerBlock>>>(deviceA, deviceB, deviceResult);

// Copy result vector from device to host

cudaMemcpy(hostResult, deviceResult, N \* sizeof(float), cudaMemcpyDeviceToHost);

// Print the first 10 elements of the result vector for (int i = 0; i < 10; i++) {

std::cout << hostResult[i] << " ";

}

std::cout << std::endl;

// Free device memory cudaFree(deviceA); cudaFree(deviceB); cudaFree(deviceResult);

// Free host memory delete[] hostA; delete[] hostB; delete[] hostResult;

return 0;

}

}

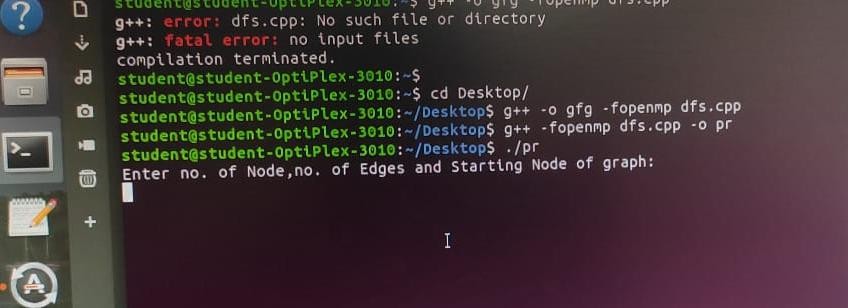
// Free device memory cudaFree(deviceA); cudaFree(deviceB); cudaFree(deviceC);

// Free host memory delete[] hostA; delete[] hostB; delete[] hostC;

return 0;

}

*Import pandas as pd*

*df = pd.read\_csv(“fashion.csv”) Df*