```cpp
#include <iostream>
using namespace std;
// CUDA code to multiply matrices
__global__ void multiply_gpu(int* A, int* B, int* C, int size) {
    // Uses thread indices and block indices to compute each element
    int row = blockIdx.y * blockDim.y + threadIdx.y;
    int col = blockIdx.x * blockDim.x + threadIdx.x;

    if (row < size && col < size) {
        int sum = 0;
        for (int i = 0; i < size; i++) {
            sum += A[row * size + i] * B[i * size + col];
        }
        C[row * size + col] = sum;
    }
}
__global__ void matrix_multiplication_cpu(int *a, int *b, int *c, int size){
    for(int i = 0; i <size; i++){
        for(int j = 0; j < size; j++){
            int sum = 0;
            for(int k = 0; k < size; k++){
                sum += a[i*size + k] * b[k*size + j];
            }
            c[i*size + j] = sum;
        }
    }
}
void initialize(int* matrix, int size) {
    for (int i = 0; i < size * size; i++) {
        matrix[i] = rand() % 10;
    }
}
void print(int* matrix, int size) {
    for (int row = 0; row < size; row++) {
        for (int col = 0; col < size; col++) {
            cout << matrix[row * size + col] << " ";
        }
        cout << '\n';
    }
    cout << '\n';
}
int main() {
    int* A, * B, * C;
    int N = 3;
    int blockSize =  16;
    int matrixSize = N * N;
    size_t matrixBytes = matrixSize * sizeof(int);
    A = new int[matrixSize];
```

```
    B = new int[matrixSize];
    C = new int[matrixSize];
    initialize(A, N);
    initialize(B, N);
    cout << "Matrix A: \n";
    print(A, N);
    cout << "Matrix B: \n";
    print(B, N);
    int* X, * Y, * Z;
    // Allocate space
    cudaMalloc(&X, matrixBytes);
    cudaMalloc(&Y, matrixBytes);
    cudaMalloc(&Z, matrixBytes);
    // Copy values from A to X
    cudaMemcpy(X, A, matrixBytes, cudaMemcpyHostToDevice);
        cudaMemcpy(Y, B, matrixBytes, cudaMemcpyHostToDevice);
 // Threads per CTA dimension
    int THREADS = 2;
    // Blocks per grid dimension
    int BLOCKS = N + THREADS - 1 / THREADS;
    cudaEvent_t start,stop;
    float elapsedTime;
    // Use dim3 structs for block  and grid dimensions
    dim3 threads(THREADS, THREADS);
    dim3 blocks(BLOCKS, BLOCKS);
     cudaEventCreate(&start);
    cudaEventCreate(&stop);
    cudaEventRecord(start,0);
    // Launch kernel
    multiply_gpu<<<blocks, threads>>>(X, Y, Z, N);
    cudaEventRecord(stop,0);
    cudaEventSynchronize(stop);
    cudaEventElapsedTime(&elapsedTime,start,stop);
    cudaEventDestroy(start);
    cudaEventDestroy(stop);
    cudaMemcpy(C, Z, matrixBytes, cudaMemcpyDeviceToHost);
    cout << "Multiplication of matrix A and B: \n";
    print(C, N);
cout<<"Elapsed Time : "<<elapsedTime<<endl;
    cudaEventCreate(&start);
    cudaEventCreate(&stop);
    cudaEventRecord(start,0);
    // Launch kernel
    matrix_multiplication_cpu<<<blocks, threads>>>(X, Y, Z, N);
    cudaEventRecord(stop,0);
    cudaEventSynchronize(stop);
    cudaEventElapsedTime(&elapsedTime,start,stop);
```

```
    cudaEventDestroy(start);
    cudaEventDestroy(stop);
    cudaMemcpy(C, Z, matrixBytes, cudaMemcpyDeviceToHost);
    cout << "Multiplication of matrix A and B: \n";
    print(C, N);
cout<<"Elapsed Time : "<<elapsedTime<<endl;
    delete[] A;
    delete[] B;
    delete[] C;
    cudaFree(X);
    cudaFree(Y);
    cudaFree(Z);
    return 0;
}
```

Vector addition

```
#include <iostream>
using namespace std;

__global__void add(int* A, int* B, int* C, int size) {
    int tid = blockIdx.x * blockDim.x + threadIdx.x;

    if (tid < size) {
        C[tid] = A[tid] + B[tid];
    }
}
void initialize(int* vector, int size) {
    for (int i = 0; i < size; i++) {
        vector[i] = rand() % 10;
    }
}
void print(int* vector, int size) {
    for (int i = 0; i < size; i++) {
        cout << vector[i] << " ";
    }
    cout << endl;
}
int main() {
    int N = 4;
    int* A, * B, * C;
    int vectorSize = N;
    size_t vectorBytes = vectorSize * sizeof(int);
    A = new int[vectorSize];
    B = new int[vectorSize];
    C = new int[vectorSize];
    initialize(A, vectorSize);
    initialize(B, vectorSize);
    cout << "Vector A: ";
    print(A, N);
```

```
    cout << "Vector B: ";
    print(B, N);
    int* X, * Y, * Z;
    cudaMalloc(&X, vectorBytes);cudaMalloc(&Y, vectorBytes);cudaMalloc(&Z, vectorBytes);
    cudaMemcpy(X, A, vectorBytes, cudaMemcpyHostToDevice);cudaMemcpy(Y, B, vectorBytes,
cudaMemcpyHostToDevice);
    int threadsPerBlock = 256;
    int blocksPerGrid = (N + threadsPerBlock - 1) / threadsPerBlock;
    add<<<blocksPerGrid, threadsPerBlock>>>(X, Y, Z, N);
    cudaMemcpy(C, Z, vectorBytes, cudaMemcpyDeviceToHost);
    cout << "Addition: ";
    print(C, N);delete[] A;delete[] B;delete[] C; cudaFree(X);cudaFree(Y);cudaFree(Z);
    return 0;
}

Fashion Minists
from tensorflow.keras.datasets import fashion_mnist
import numpy as np
(train_x, train_y), (test_x, test_y) = fashion_mnist.load_data()
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, Flatten, MaxPooling2D, Conv2D
model = Sequential()
model.add(Conv2D(filters=64,kernel_size=(3,3),activation='relu',input_shape=(28, 28, 1)))
# Adding maxpooling layer to get max value within a matrix
model.add(MaxPooling2D(pool_size=(2,2)))
model.add(Flatten())
model.add(Dense(128, activation = "relu"))
model.add(Dense(10, activation = "softmax"))
model.summary()
model.compile(optimizer = 'adam', loss = 'sparse_categorical_crossentropy', metrics = ['accuracy'])
model.fit(train_x.astype(np.float32), train_y.astype(np.float32), epochs = 5, validation_split = 0.2)
loss, acc = model.evaluate(test_x, test_y)
labels = ['t_shirt', 'trouser', 'pullover', 'dress', 'coat', 'sandal', 'shirt', 'sneaker',   'bag',
'ankle_boots']
predictions = model.predict(test_x[:1])
label = labels[np.argmax(predictions)]
import matplotlib.pyplot as plt
print(label)
plt.imshow(test_x[:1][0])
plt.show

Google stock
import pandas as pd
df= pd.read_csv("goog1.csv")
closed= df['Close']
from sklearn.preprocessing import MinMaxScaler
sc= MinMaxScaler()
closed= sc.fit_transform(closed.values.reshape(-1,1))
```

```
train_data= closed[:int(len(closed)*0.8)]
train_data
test_data= closed[int(len(closed)*0.8):]
test_data
import numpy as np
def createxy(data):
    x=[]
    y=[]
    for i in range(0,len(data)-6):
        x.append(data[i:i+6,0]);
        print(data[i:i+6,0])
        print(data[i+6,0])
        y.append(data[i+6,0]);
    return np.array(x), np.array(y)
x_train,y_train= createxy(train_data)
x_test,y_test=createxy(test_data)
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import LSTM , Dense
model= Sequential()
model.add(LSTM(units=50,return_sequences=True,input_shape=(6,1)))
model.add(LSTM(units=50))
model.add(Dense(units=1))
model.summary()
model.compile(optimizer='adam',loss="mse",metrics=['mae'])
model.fit(x_train,y_train,epochs=10)
y_predict= model.predict(x_test)
y_predict= sc.inverse_transform(y_predict)
y=[]
for i in y_test:
    y.append([i])
y_test= y
y_test= sc.inverse_transform(y_test)
import matplotlib.pyplot as plt
plt.plot(y_test,label="test")
plt.plot(y_predict,label="predicted data")
plt.legend();
plt.show()
from sklearn.metrics import mean_absolute_error,mean_squared_error
mae= mean_absolute_error(y_test,y_predict)
mse= mean_squared_error(y_test,y_predict)


Boston Housing
import numpy as np
import pandas as pd
df = pd.read_csv('HousingData.csv',na_values='NA')
df.head()
df.columns
```

```
df.isna().sum()
df.describe()
for i in df.columns:
    mean_value = df[i].mean()
    df[i].fillna(mean_value, inplace=True)
    df.isna().sum()
df.head()
from sklearn.model_selection import train_test_split
X = df.loc[:, df.columns != 'MEDV']
y = df.loc[:, df.columns == 'MEDV']
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3, random_state=123)
from sklearn.preprocessing import MinMaxScaler
mms = MinMaxScaler()
mms.fit(X_train)
X_train = mms.transform(X_train)
X_test = mms.transform(X_test)
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense
model = Sequential()
model.add(Dense(128, input_shape=(13, ), activation='relu', name='dense_1'))
model.add(Dense(64, activation='relu', name='dense_2'))
model.add(Dense(1, activation='linear', name='dense_output'))

model.compile(optimizer='adam', loss='mse', metrics=['mae'])
model.summary()
history = model.fit(X_train, y_train, epochs=100, validation_split=0.05, verbose = 1)
mse_nn, mae_nn = model.evaluate(X_test, y_test)

print('Mean squared error on test data: ', mse_nn)
print('Mean absolute error on test data: ', mae_nn)
```