MPI command:

install
sudo apt update
sudo apt install openmpi-bin openmpi-common libopenmpi-dev

g++ install(optional)
sudo apt update
sudo apt install build-essential

Check Installation
mpic++ --version
mpirun --version

Compile and run
mpic++ hello_mpi.cpp -o hello_mpi
mpirun -np 4 ./hello_mpi

MPI basic function:

MPI Initialization
MPI_Init(&argc, &argv);

Get process info
MPI_Comm_rank(MPI_COMM_WORLD, &rank);
MPI_Comm_size(MPI_COMM_WORLD, &size);

Example usage:
if(rank == 0) cout << "I am the master process\n";
else cout << "I am worker " << rank << "\n";

Broadcasting data
MPI_Bcast(&K, 1, MPI_INT, 0, MPI_COMM_WORLD);

Sends a **single value** (or array) from **root process** (`rank = 0`) to **all other processes**
Parameters:

1. `&K` → pointer to data to send
2. `1` → number of elements
3. `MPI_INT` → data type
4. `0` → root process (source)
5. `MPI_COMM_WORLD` → communicator

Scatter (distribute data)
MPI_Scatter(A, localK*M*N, MPI_INT,
localA, localK*M*N, MPI_INT,
0, MPI_COMM_WORLD);

Divides an array from root into chunks and sends each chunk to a different process

Parameters:

1. `A` → root's array to scatter
2. `localK*M*N` → number of elements per process
3. `MPI_INT` → datatype
4. `localA` → destination buffer for current process
5. `localK*M*N` → how many elements each process receives
6. `MPI_INT` → datatype
7. `0` → root process
8. `MPI_COMM_WORLD` → communicator

MPI_Barrier(Forces all processes to wait here until everyone reaches this point)
MPI_Barrier(MPI_COMM_WORLD);

Measuring time
double start = MPI_Wtime();
double end = MPI_Wtime();
double localTime = end - start;

Reduce (combine results)
MPI_Reduce(&localTime, &totalTime, 1, MPI_DOUBLE, MPI_MAX, 0, MPI_COMM_WORLD);

Combines values from all processes into one value on the root

Parameters:

1. `&localTime` → local process value
2. `&totalTime` → result on root
3. 1 → number of elements
4. `MPI_DOUBLE` → datatype
5. `MPI_MAX` → operation (here maximum of all processes)
6. `0` → root process receives result
7. `MPI_COMM_WORLD` → communicator

Gather (collect results - Opposite of scatter**)**
MPI_Gather(localR, localK*M*P, MPI_INT,
R, localK*M*P, MPI_INT,
0, MPI_COMM_WORLD);

Finalize MPI
MPI_Finalize();

MPI_Send

```
MPI_Send(
    void* buf,
    int count,
    MPI_Datatype datatype,
    int dest,
```

```
    int tag,
    MPI_Comm comm
);
```

MPI_Recv

```
MPI_Recv(
    void* buf,
    int count,
    MPI_Datatype datatype,
    int source,
    int tag,
    MPI_Comm comm,
    MPI_Status* status
);
```

C++ string operations:

Count number of occurrences of a pattern in a string

```
int countOccurrences(const string& text, const string& pat) {
    int count = 0;
    size_t pos = 0;

    while ((pos = text.find(pat, pos)) != string::npos) {
        count++;
        pos += pat.length();   // move forward (non-overlapping)
    }
    return count;
}
```

---

check if a pattern exists and its index

```
size_t pos = text.find(pattern);

if (pos != string::npos) {
    cout << "Found at index " << pos;
}
```

- Returns **first occurrence**
- `string::npos` means *not found*

Check if a pattern exists from a specific range

```
size_t pos = text.find(pattern, startIndex);

if (pos != string::npos) {
    cout << "Found at index " << pos;
}
```

Split strings from a big string by a character or string

`stringstream` (char delimiter)

```
vector<string> split(const string& s, char delim) {
    vector<string> tokens;
    string token;
    stringstream ss(s);

    while (getline(ss, token, delim)) {
        tokens.push_back(token);
    }
    return tokens;
}
```

Example

```
split("Alice,0171,CS", ',');
```

Split by string delimiter (manual)

```
vector<string> split(const string& s, const string& delim) {
    vector<string> tokens;
    size_t start = 0, pos;

    while ((pos = s.find(delim, start)) != string::npos) {
        tokens.push_back(s.substr(start, pos - start));
        start = pos + delim.length();
    }
    tokens.push_back(s.substr(start));
    return tokens;
}
```

//file upload in colab:

```
from google.colab import files

files.upload()
```

Read and write from file

write

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

ofstream fout;
fout.open("output.txt");

fout << "Hello World\n";
fout << 100 << " " << 3.14 << endl;

fout.close();
```

read

```
ifstream fin("input.txt");
string word;

while (fin >> word) {
    cout << word << endl;
}

fin.close();
```

Read line by line

```
ifstream fin("input.txt");
string line;

while (getline(fin, line)) {
    cout << line << endl;
}

fin.close();
```

MPI code:

Matrix

```
#include <mpi.h>
#include <iostream>
#include <vector>
#include <cstdlib>  // for rand()
#include <ctime>    // for srand(time(0))
using namespace std;

int main(int argc, char* argv[]) {
    MPI_Init(&argc, &argv);  // Initialize MPI environment
```

```
int rank, size;
MPI_Comm_rank(MPI_COMM_WORLD, &rank); // Process ID
MPI_Comm_size(MPI_COMM_WORLD, &size); // Total number of processes

int K, M, N, P;

if (rank == 0) {
    // Root process reads input
    cin >> K >> M >> N >> P;
}

// Broadcast input to all processes
MPI_Bcast(&K, 1, MPI_INT, 0, MPI_COMM_WORLD);
MPI_Bcast(&M, 1, MPI_INT, 0, MPI_COMM_WORLD);
MPI_Bcast(&N, 1, MPI_INT, 0, MPI_COMM_WORLD);
MPI_Bcast(&P, 1, MPI_INT, 0, MPI_COMM_WORLD);

// Determine which matrices this process will compute
int per_proc = K / size;
int start = rank * per_proc;
int end = (rank == size - 1) ? K : start + per_proc;

// Initialize matrices A, B, C (flattened 1D arrays)
vector<double> A(M * N), B(N * P), C(M * P);

// Seed random generator differently for each process
srand(time(0) + rank);

// Fill A and B with random numbers
for (int i = 0; i < M * N; i++)
    A[i] = rand() % 10 + 1; // random 1-10
for (int i = 0; i < N * P; i++)
    B[i] = rand() % 10 + 1; // random 1-10

MPI_Barrier(MPI_COMM_WORLD); // synchronize before timing
double t1 = MPI_Wtime();     // start local timer

// Multiply matrices for assigned K matrices
for (int k = start; k < end; k++) {
    for (int i = 0; i < M; i++) {
        for (int j = 0; j < P; j++) {
            C[i * P + j] = 0;
            for (int x = 0; x < N; x++) {
                C[i * P + j] += A[i * N + x] * B[x * P + j];
            }
        }
    }
}

double t2 = MPI_Wtime();
double local_time = t2 - t1; // local computation time
double total_time;

// Reduce to find max time among all processes (total time)
```

```
    MPI_Reduce(&local_time, &total_time, 1, MPI_DOUBLE, MPI_MAX, 0, MPI_COMM_WORLD);

    // Print local time for each process
    cout << "Process " << rank << " time = " << local_time << " seconds\n";

    if (rank == 0) {
        cout << "\nMPI Total Time Taken: " << total_time << " seconds\n";

        // Print A[0]
        cout << "\nA[0]:\n";
        for (int i = 0; i < M; i++) {
            for (int j = 0; j < N; j++)
                cout << A[i * N + j] << " ";
            cout << endl;
        }

        // Print B[0]
        cout << "\nB[0]:\n";
        for (int i = 0; i < N; i++) {
            for (int j = 0; j < P; j++)
                cout << B[i * P + j] << " ";
            cout << endl;
        }

        // Print C[0]
        cout << "\nC[0] = A[0] × B[0]:\n";
        for (int i = 0; i < M; i++) {
            for (int j = 0; j < P; j++)
                cout << C[i * P + j] << " ";
            cout << endl;
        }
    }

    MPI_Finalize(); // Finalize MPI
    return 0;
}
```

//phonebook

```
#include <mpi.h>
#include <iostream>
#include <fstream>
#include <vector>
#include <string>

using namespace std;

int main(int argc, char* argv[]) {
    MPI_Init(&argc, &argv);

    int rank, size;
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);
```

```
string searchName;
vector<string> lines;

// ---------- ROOT READS INPUT ----------
if (rank == 0) {
    cin >> searchName;

    ifstream file("phonebook.txt");
    string line;
    while (getline(file, line)) {
        lines.push_back(line);
    }
    file.close();
}

// ---------- BROADCAST SEARCH NAME ----------
int nameLen;
if (rank == 0) nameLen = searchName.size();
MPI_Bcast(&nameLen, 1, MPI_INT, 0, MPI_COMM_WORLD);

searchName.resize(nameLen);
MPI_Bcast(&searchName[0], nameLen, MPI_CHAR, 0, MPI_COMM_WORLD);

// ---------- BROADCAST TOTAL LINES ----------
int totalLines;
if (rank == 0) totalLines = lines.size();
MPI_Bcast(&totalLines, 1, MPI_INT, 0, MPI_COMM_WORLD);

// ---------- DIVIDE WORK ----------
int perProc = totalLines / size;
int start = rank * perProc;
int end = (rank == size - 1) ? totalLines : start + perProc;

MPI_Barrier(MPI_COMM_WORLD);
double t1 = MPI_Wtime();

// ---------- LOCAL SEARCH ----------
vector<string> localMatches;
for (int i = start; i < end; i++) {
    if (rank == 0) {
        if (lines[i].find(searchName) == 0)
            localMatches.push_back(lines[i]);
    }
}

// ---------- SEND LINES TO OTHER PROCESSES ----------
if (rank == 0) {
    for (int p = 1; p < size; p++) {
        int s = p * perProc;
        int e = (p == size - 1) ? totalLines : s + perProc;

        for (int i = s; i < e; i++) {
            int len = lines[i].size();
            MPI_Send(&len, 1, MPI_INT, p, 0, MPI_COMM_WORLD);
```

```
            MPI_Send(lines[i].c_str(), len, MPI_CHAR, p, 0, MPI_COMM_WORLD);
        }
    }
}

// ---------- OTHER PROCESSES RECEIVE & SEARCH ----------
if (rank != 0) {
    for (int i = start; i < end; i++) {
        int len;
        MPI_Recv(&len, 1, MPI_INT, 0, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);

        string line(len, ' ');
        MPI_Recv(&line[0], len, MPI_CHAR, 0, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);

        if (line.find(searchName) == 0)
            localMatches.push_back(line);
    }
}

double t2 = MPI_Wtime();
double localTime = t2 - t1;

// ---------- SEND MATCHES BACK TO ROOT ----------
int localCount = localMatches.size();
MPI_Send(&localCount, 1, MPI_INT, 0, 1, MPI_COMM_WORLD);

for (auto &s : localMatches) {
    int len = s.size();
    MPI_Send(&len, 1, MPI_INT, 0, 1, MPI_COMM_WORLD);
    MPI_Send(s.c_str(), len, MPI_CHAR, 0, 1, MPI_COMM_WORLD);
}

// ---------- ROOT COLLECTS ALL MATCHES ----------
if (rank == 0) {
    vector<string> allMatches = localMatches;

    for (int p = 1; p < size; p++) {
        int cnt;
        MPI_Recv(&cnt, 1, MPI_INT, p, 1, MPI_COMM_WORLD, MPI_STATUS_IGNORE);

        for (int i = 0; i < cnt; i++) {
            int len;
            MPI_Recv(&len, 1, MPI_INT, p, 1, MPI_COMM_WORLD, MPI_STATUS_IGNORE);

            string s(len, ' ');
            MPI_Recv(&s[0], len, MPI_CHAR, p, 1, MPI_COMM_WORLD, MPI_STATUS_IGNORE);

            allMatches.push_back(s);
        }
    }

    cout << "\nMatched Contacts:\n";
    for (auto &s : allMatches)
        cout << s << endl;
```

```
    }

    // ---------- TOTAL TIME ----------
    double totalTime;
    MPI_Reduce(&localTime, &totalTime, 1, MPI_DOUBLE, MPI_MAX, 0, MPI_COMM_WORLD);

    if (rank == 0)
        cout << "\nTotal Time Taken: " << totalTime << " seconds\n";

    MPI_Finalize();
    return 0;
}
```

// substring search

```
#include <mpi.h>
#include <iostream>
#include <fstream>
#include <vector>
#include <string>
#include <sstream>
#include <algorithm>

using namespace std;

// ----------------- MPI UTILS -----------------
void broadcast_string(string &text, int root = 0) {
    int length;
    int rank;
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    if(rank == root) length = text.size();
    MPI_Bcast(&length, 1, MPI_INT, root, MPI_COMM_WORLD);
    if(rank != root) text.resize(length);
    MPI_Bcast(&text[0], length, MPI_CHAR, root, MPI_COMM_WORLD);
}

string vector_to_string(const vector<string> &vec, int start = 0, int end = -1, const string &sep = "\n") {
    if(end == -1 || end > vec.size()) end = vec.size();
    string text;
    for(int i=start;i<end;i++) text += vec[i] + sep;
    return text;
}

vector<string> string_to_vector(const string &text, const string &sep = "\n") {
    vector<string> vec;
    stringstream ss(text);
    string temp;
    if(sep=="\n") {
        while(getline(ss,temp)) vec.push_back(temp);
    } else {
        while(getline(ss,temp,sep[0])) vec.push_back(temp);
    }
    return vec;
}
```

```
pair<int,int> get_work_range(int total_items, int rank, int size) {
    int per_proc = total_items / size;
    int start = rank * per_proc;
    int end = (rank == size-1) ? total_items : start + per_proc;
    return {start,end};
}

int longestMatch(const string &line, const string &pattern) {
    int maxLen = 0;
    int n = pattern.size();
    for(int i=0;i<n;i++){
        for(int len=1;i+len<=n;len++){
            if(line.find(pattern.substr(i,len)) != string::npos)
                maxLen = max(maxLen,len);
        }
    }
    return maxLen;
}

// ----------------- MAIN -----------------
int main(int argc, char** argv){
    MPI_Init(&argc,&argv);

    int rank, size;
    MPI_Comm_rank(MPI_COMM_WORLD,&rank);
    MPI_Comm_size(MPI_COMM_WORLD,&size);

    MPI_Barrier(MPI_COMM_WORLD);
    double t_start = MPI_Wtime(); // start timer

    vector<string> lines;
    string pattern;

    // ----------------- PROCESS 0 READS FILE -----------------
    if(rank==0){
        ifstream fin("phonebook.txt");
        string line;
        while(getline(fin,line)) lines.push_back(line);
        fin.close();

        cout << "Enter pattern to search: ";
        cin >> pattern;
    }

    // ----------------- BROADCAST PATTERN -----------------
    broadcast_string(pattern);

    // ----------------- BROADCAST LINES -----------------
    string allText = vector_to_string(lines);
    broadcast_string(allText);

    // Reconstruct lines on all processes
    lines = string_to_vector(allText);
```

```
    // ----------------- COMPUTE LOCAL RANGE -----------------
    auto [start,end] = get_work_range(lines.size(), rank, size);

    int localBestLen = 0;
    string localBestLine;

    for(int i=start;i<end;i++){
        int matchLen = longestMatch(lines[i], pattern);
        if(matchLen > localBestLen){
            localBestLen = matchLen;
            localBestLine = lines[i];
        }
    }

    // ----------------- REDUCE TO FIND GLOBAL BEST -----------------
    struct { int len; int rank; } localData{localBestLen, rank}, globalData;
    MPI_Reduce(&localData,&globalData,1,MPI_2INT,MPI_MAXLOC,0,MPI_COMM_WORLD);

    if(rank == globalData.rank){
        cout << "\nBest Matching Line:\n" << localBestLine << endl;
        cout << "Match Length: " << localBestLen << endl;
    }

    // ----------------- TIME PRINT -----------------
    double t_end = MPI_Wtime();
    double local_time = t_end - t_start;

    // Each process prints its own time
    printf("Process %d time: %f seconds\n", rank, local_time);
    MPI_Barrier(MPI_COMM_WORLD);
    // Compute total (max) time across all processes
    double total_time;
    MPI_Reduce(&local_time, &total_time, 1, MPI_DOUBLE, MPI_MAX, 0, MPI_COMM_WORLD);
    if(rank==0) printf("Total parallel time: %f seconds\n", total_time);

    MPI_Finalize();
    return 0;
}
```

Word count

```
#include <mpi.h>
#include <bits/stdc++.h>
using namespace std;

int main(int argc, char* argv[]) {
    MPI_Init(&argc, &argv);

    int rank, size;
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);

    double start = MPI_Wtime();
```

```
vector<string> lines;
string filename = "input.txt";

// ---------- Rank 0 reads file ----------
if(rank == 0) {
    ifstream fin(filename);
    string line;
    while(getline(fin, line)) lines.push_back(line);
    fin.close();
}

// ---------- Broadcast number of lines ----------
int totalLines;
if(rank == 0) totalLines = lines.size();
MPI_Bcast(&totalLines, 1, MPI_INT, 0, MPI_COMM_WORLD);

// ---------- Broadcast all text ----------
string allText;
if(rank == 0) {
    for(auto &l : lines) allText += l + "\n";
}

int textSize;
if(rank == 0) textSize = allText.size();
MPI_Bcast(&textSize, 1, MPI_INT, 0, MPI_COMM_WORLD);

if(rank != 0) allText.resize(textSize);
MPI_Bcast(&allText[0], textSize, MPI_CHAR, 0, MPI_COMM_WORLD);

// ---------- Rebuild lines ----------
lines.clear();
stringstream ss(allText);
string temp;
while(getline(ss, temp)) lines.push_back(temp);

// ---------- Divide work ----------
int perProc = totalLines / size;
int startLine = rank * perProc;
int endLine = (rank == size - 1) ? totalLines : startLine + perProc;

map<string,int> localCount;

// ---------- Local word count ----------
for(int i = startLine; i < endLine; i++) {
    stringstream ls(lines[i]);
    string word;
    while(ls >> word) {
        localCount[word]++;
    }
}

// ---------- Convert local map to string ----------
string localData;
```

```
    for(auto &p : localCount)
        localData += p.first + " " + to_string(p.second) + "\n";

    int localSize = localData.size();
    MPI_Gather(&localSize, 1, MPI_INT, NULL, 0, MPI_INT, 0, MPI_COMM_WORLD);

    // ---------- Rank 0 merges ----------
    map<string,int> globalCount;

    if(rank == 0) {
        globalCount = localCount;
        for(int p = 1; p < size; p++) {
            int sz;
            MPI_Recv(&sz, 1, MPI_INT, p, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
            string buf(sz, ' ');
            MPI_Recv(&buf[0], sz, MPI_CHAR, p, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);

            string w; int c;
            stringstream s(buf);
            while(s >> w >> c) globalCount[w] += c;
        }
    } else {
        MPI_Send(&localSize, 1, MPI_INT, 0, 0, MPI_COMM_WORLD);
        MPI_Send(&localData[0], localSize, MPI_CHAR, 0, 0, MPI_COMM_WORLD);
    }

    double end = MPI_Wtime();
    double localTime = end - start;
    double totalTime;

    MPI_Reduce(&localTime, &totalTime, 1, MPI_DOUBLE, MPI_MAX, 0, MPI_COMM_WORLD);

    // ---------- Print result ----------
    if(rank == 0) {
        vector<pair<int,string>> v;
        for(auto &p : globalCount)
            v.push_back({p.second, p.first});

        sort(v.rbegin(), v.rend());

        cout << "\nTop 10 words:\n";
        for(int i = 0; i < min(10,(int)v.size()); i++)
            cout << v[i].second << " -> " << v[i].first << endl;

        cout << "\nTotal MPI Time: " << totalTime << " seconds\n";
    }

    MPI_Finalize();
    return 0;
}
```

//count pattern

```
#include <mpi.h>
#include <bits/stdc++.h>
```

```
using namespace std;

// Count substring occurrences in a string
int countOccurrences(const string& text, const string& pat) {
    int count = 0;
    size_t pos = text.find(pat);
    while (pos != string::npos) {
        count++;
        pos = text.find(pat, pos + 1);
    }
    return count;
}

int main(int argc, char* argv[]) {
    MPI_Init(&argc, &argv);

    int rank, size;
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);

    string paragraph, pattern;

    double start = MPI_Wtime();

    // -------- Rank 0 reads input --------
    if (rank == 0) {
        ifstream fin("input.txt");
        string line;
        while (getline(fin, line))
            paragraph += line + " ";
        fin.close();

        cout << "Enter pattern (like %x%): ";
        cin >> pattern;

        // remove % from %x%
        pattern = pattern.substr(1, pattern.size() - 2);
    }

    // -------- Broadcast pattern --------
    int patLen;
    if (rank == 0) patLen = pattern.size();
    MPI_Bcast(&patLen, 1, MPI_INT, 0, MPI_COMM_WORLD);

    pattern.resize(patLen);
    MPI_Bcast(&pattern[0], patLen, MPI_CHAR, 0, MPI_COMM_WORLD);

    // -------- Broadcast paragraph --------
    int textLen;
    if (rank == 0) textLen = paragraph.size();
    MPI_Bcast(&textLen, 1, MPI_INT, 0, MPI_COMM_WORLD);

    paragraph.resize(textLen);
    MPI_Bcast(&paragraph[0], textLen, MPI_CHAR, 0, MPI_COMM_WORLD);
```

```
    // -------- Divide work --------
    int n = paragraph.size();
    int perProc = n / size;
    int startIdx = rank * perProc;
    int endIdx = (rank == size - 1) ? n : startIdx + perProc + pattern.size();

    string localText = paragraph.substr(startIdx, endIdx - startIdx);

    // -------- Local count --------
    int localCount = countOccurrences(localText, pattern);

    // -------- Reduce counts --------
    int totalCount;
    MPI_Reduce(&localCount, &totalCount, 1, MPI_INT, MPI_SUM, 0, MPI_COMM_WORLD);

    double end = MPI_Wtime();
    double localTime = end - start;
    double totalTime;

    MPI_Reduce(&localTime, &totalTime, 1, MPI_DOUBLE, MPI_MAX, 0, MPI_COMM_WORLD);

    if (rank == 0) {
        cout << "\nTotal Occurrences: " << totalCount << endl;
        cout << "Total MPI Time: " << totalTime << " seconds\n";
    }

    MPI_Finalize();
    return 0;
}
```

Cuda code

Matrix

```
%%writefile sadnur.cu
#include <iostream>
#include <cuda_runtime.h>
using namespace std;

__global__ void matrixMul(float* A, float* B, float* C, int M, int N, int P, int offset) {
    int k = threadIdx.x + offset;

    float* a = A + k * M * N;
    float* b = B + k * N * P;
    float* c = C + k * M * P;

    for(int i = 0; i < M; i++) {
        for(int j = 0; j < N; j++) {
            for(int l = 0; l < P; l++) {
                //c[i][l] += a[i][j] * b[j][l];
                c[i * P + l] += a[i * N + j] * b[j * P + l];
            }
        }
    }
```

```
}

int main(int argc, char *argv[]) {

    int T = atoi(argv[1]); //koyta thread use korte parbo
    int K = atoi(argv[2]); //koita matrix gun

    //100 gun, thread 10,

    int M = 400, N = 400, P = 400;

    int SizeA = M * N * K;
    int SizeB = N * P * K;
    int SizeC = M * P * K;

    //memory alocate (cpu allocate)
    float *h_A = new float[SizeA];
    float *h_B = new float[SizeB];
    float *h_C = new float[SizeC];


    //malloc (gpu allocate)
    float *d_A;
    cudaMalloc(&d_A, SizeA * sizeof(float));
    float *d_B;
    cudaMalloc(&d_B, SizeB * sizeof(float));
    float *d_C;
    cudaMalloc(&d_C, SizeC * sizeof(float));

    //data initialize
    for (int i = 0; i < SizeA; i++) {
        h_A[i] = rand();
    }
    for(int i = 0; i < SizeB; i++) {
        h_B[i] = rand();
    }


    //copy from host to device
    cudaMemcpy(d_A, h_A, SizeA * sizeof(float), cudaMemcpyHostToDevice);
    cudaMemcpy(d_B, h_B, SizeB * sizeof(float), cudaMemcpyHostToDevice);

    //cuda process suru
    int gunKorteHobe = K;
    int offset = 0;
    while(gunKorteHobe > 0){

        int currentBatch = min(gunKorteHobe, T);

        matrixMul<<<1,currentBatch>>>(d_A, d_B, d_C, M, N, P, offset);
        cudaDeviceSynchronize();

        gunKorteHobe -= currentBatch;
        offset += currentBatch;
```

```
    }

    //let's copy back to cpu
    cudaMemcpy(h_C, d_C, SizeC * sizeof(float), cudaMemcpyDeviceToHost);

    cout << "All operation done" << endl;

}


/*
!nvcc -arch=sm_75 sadnur.cu -o sadnur

!time ./sadnur 20 10 && sleep 2

*/
```

Matrix mul

```
%%writefile fixed_matmul.cu
#include <iostream>
#include <cuda_runtime.h>
using namespace std;

__global__ void matMulKernel(
    float* A, float* B, float* C,
    int M, int N, int P, int K)
{
    int k   = blockIdx.z;
    int row = blockIdx.y * blockDim.y + threadIdx.y;
    int col = blockIdx.x * blockDim.x + threadIdx.x;

    if (k < K && row < M && col < P) {
        float sum = 0.0f;
        for (int j = 0; j < N; j++) {
            sum += A[k*M*N + row*N + j] *
                   B[k*N*P + j*P + col];
        }
        C[k*M*P + row*P + col] = sum;
    }
}

int main() {
    int M = 4, N = 4, P = 4, K = 2;

    int sizeA = M*N*K;
    int sizeB = N*P*K;
    int sizeC = M*P*K;

    float *h_A = new float[sizeA];
    float *h_B = new float[sizeB];
    float *h_C = new float[sizeC];

    // Initialize A and B, initialize C = 0
```

```
for (int i = 0; i < sizeA; i++) h_A[i] = 1;
for (int i = 0; i < sizeB; i++) h_B[i] = 1;
for (int i = 0; i < sizeC; i++) h_C[i] = 0;

float *d_A, *d_B, *d_C;
cudaMalloc(&d_A, sizeA*sizeof(float));
cudaMalloc(&d_B, sizeB*sizeof(float));
cudaMalloc(&d_C, sizeC*sizeof(float));

cudaMemcpy(d_A, h_A, sizeA*sizeof(float), cudaMemcpyHostToDevice);
cudaMemcpy(d_B, h_B, sizeB*sizeof(float), cudaMemcpyHostToDevice);
cudaMemcpy(d_C, h_C, sizeC*sizeof(float), cudaMemcpyHostToDevice);

dim3 threads(16,16);
dim3 blocks((P+15)/16, (M+15)/16, K);

cudaEvent_t start, stop;
cudaEventCreate(&start);
cudaEventCreate(&stop);
cudaEventRecord(start);

matMulKernel<<<blocks, threads>>>(d_A, d_B, d_C, M, N, P, K);

cudaEventRecord(stop);
cudaEventSynchronize(stop);

cudaError_t err = cudaGetLastError();
if (err != cudaSuccess)
    cout << "CUDA Error: " << cudaGetErrorString(err) << endl;

float ms;
cudaEventElapsedTime(&ms, start, stop);

cudaMemcpy(h_C, d_C, sizeC*sizeof(float), cudaMemcpyDeviceToHost);

// Print A[0]
cout << "\nA[0]:\n";
for(int i=0;i<M;i++){
    for(int j=0;j<N;j++)
        cout << h_A[i*N + j] << " ";
    cout << endl;
}

// Print B[0]
cout << "\nB[0]:\n";
for(int i=0;i<N;i++){
    for(int j=0;j<P;j++)
        cout << h_B[i*P + j] << " ";
    cout << endl;
}

// Print C[0]
cout << "\nC[0] = A[0] × B[0]:\n";
for(int i=0;i<M;i++){
```

```
        for(int j=0;j<P;j++)
            cout << h_C[i*P + j] << " ";
        cout << endl;
    }

    cout << "\nGPU Time: " << ms << " ms\n";

    // Cleanup
    delete[] h_A;
    delete[] h_B;
    delete[] h_C;
    cudaFree(d_A);
    cudaFree(d_B);
    cudaFree(d_C);

    return 0;
}
```

Phonebook

```
%%writefile phonebook_search.cu
#include <bits/stdc++.h>
using namespace std;
#include <cuda.h>

struct Contact{
    char name[65];
    char phone_number[65];
};

// Removed the old getInput function as it's no longer needed

__device__ bool check(char* str1, char* str2){
    for(int i = 0; str1[i] != '\0'; i++){
        int flag = 1;
        for(int j = 0; str2[j] != '\0' ; j++){
            if(str1[i + j] != str2[j]){
                flag = 0;
                break;
            }
        }
        if(flag == 1) return true;
    }
    return false;
}


__global__ void myKernel(Contact* phoneBook, char* pat, int offset){
    int threadNumber = threadIdx.x + blockDim.x * blockIdx.x + offset; // Corrected thread numbering for multiple blocks
    // Added a bounds check to prevent out-of-bounds access if n is not a multiple of blockSize
    if (threadNumber < 10000) { // Limit to max contacts read
        if(check(phoneBook[threadNumber].name, pat)){
            printf("%s %s\n", phoneBook[threadNumber].name, phoneBook[threadNumber].phone_number);
        }
```

```
    }
}



int main(int argc, char* argv[])
{
    if (argc < 3) {
        cerr << "Usage: " << argv[0] << " <search_name> <thread_limit>\n";
        return 1;
    }
    int threadLimit = atoi(argv[2]);

    ifstream myfile("one.txt");
    if (!myfile.is_open()) {
        cerr << "Error: Could not open one.txt\n";
        return 1;
    }
    vector<Contact> phoneBook;

    string line;
    int count = 0;
    while(getline(myfile, line)){
        if(count >= 10000) break; // Apply the same limit as before
        if(line.empty()) continue;

        size_t space_pos = line.find(' ');
        if(space_pos == string::npos) {
            // Handle lines without a space separator, e.g., print a warning or skip
            cerr << "Warning: Skipping malformed line: " << line << "\n";
            continue;
        }

        string name_str = line.substr(0, space_pos);
        string phone_str = line.substr(space_pos + 1);

        Contact c;
        // Ensure strings are null-terminated and do not overflow buffers
        strncpy(c.name, name_str.c_str(), sizeof(c.name) - 1);
        c.name[sizeof(c.name) - 1] = '\0';
        strncpy(c.phone_number, phone_str.c_str(), sizeof(c.phone_number) - 1);
        c.phone_number[sizeof(c.phone_number) - 1] = '\0';

        phoneBook.push_back(c);
        count++;
    }
    myfile.close();

    string search_name = argv[1];
    char pat[65];
    strncpy(pat, search_name.c_str(), sizeof(pat) - 1);
    pat[sizeof(pat) - 1] = '\0';

    char* d_pat;
```

```
    cudaMalloc(&d_pat, 65); //memory allocation
    cudaMemcpy(d_pat, pat, 65, cudaMemcpyHostToDevice); //copying to device

    int n = phoneBook.size();
    Contact* d_phoneBook;
    cudaMalloc(&d_phoneBook, n*sizeof(Contact));
    cudaMemcpy(d_phoneBook, phoneBook.data(), n * sizeof(Contact), cudaMemcpyHostToDevice);

    int bakiAche = n;
    int offset = 0;
    // Determine a reasonable block size for better GPU utilization, e.g., 256 or 512
    // Using threadLimit as blockSize now, as it's the max threads per block.
    int blockSize = threadLimit;

    while(bakiAche > 0){
        int numThreads = min(blockSize, bakiAche);
        int numBlocks = (numThreads + blockSize - 1) / blockSize; // Should always be 1 block if numThreads <= blockSize
        myKernel<<<numBlocks, numThreads>>>(d_phoneBook, d_pat, offset);
        cudaDeviceSynchronize();

        bakiAche -= numThreads;
        offset += numThreads;
    }

    cudaFree(d_pat);
    cudaFree(d_phoneBook);

    return 0;
}


/*
!nvcc -arch=sm_75 phonebook_search.cu -o phonebook_search
!time ./phonebook_search Lily 5 > output.txt
*/
```

Phonebook search

```
%%writefile cuda_phonebook.cu
#include <bits/stdc++.h>
#include <cuda.h>
using namespace std;

#define MAX_LINE 100
#define MAX_NAME 50

__global__ void searchKernel(char phonebook[][MAX_LINE],
                             char* search,
                             int total) {
    int idx = blockIdx.x * blockDim.x + threadIdx.x;

    if (idx < total) {
        bool match = true;
        int i = 0;
        while (search[i] != '\0') {
```

```
            if (phonebook[idx][i] != search[i]) {
                match = false;
                break;
            }
            i++;
        }
        if (match) {
            printf("Match: %s\n", phonebook[idx]);
        }
    }
}

int main() {
    vector<string> lines;
    string line, search;

    ifstream fin("phonebook.txt");
    while (getline(fin, line))
        lines.push_back(line);
    fin.close();

    cout << "Enter name to search: ";
    cin >> search;

    int n = lines.size();

    char h_phonebook[n][MAX_LINE];
    char h_search[MAX_NAME];

    for (int i = 0; i < n; i++)
        strcpy(h_phonebook[i], lines[i].c_str());
    strcpy(h_search, search.c_str());

    char (*d_phonebook)[MAX_LINE];
    char* d_search;

    cudaMalloc(&d_phonebook, n * MAX_LINE);
    cudaMalloc(&d_search, MAX_NAME);

    cudaMemcpy(d_phonebook, h_phonebook, n * MAX_LINE, cudaMemcpyHostToDevice);
    cudaMemcpy(d_search, h_search, MAX_NAME, cudaMemcpyHostToDevice);

    cudaEvent_t start, stop;
    cudaEventCreate(&start);
    cudaEventCreate(&stop);

    cudaEventRecord(start);

    int threads = 256;
    int blocks = (n + threads - 1) / threads;
    searchKernel<<<blocks, threads>>>(d_phonebook, d_search, n);
    cudaDeviceSynchronize();

    cudaEventRecord(stop);
```

```
    cudaEventSynchronize(stop);

    float ms;
    cudaEventElapsedTime(&ms, start, stop);

    cout << "\nCUDA Time: " << ms << " ms\n";

    cudaFree(d_phonebook);
    cudaFree(d_search);
}
```

Word count

```
%%writefile wordcount.cu
#include <bits/stdc++.h>
#include <cuda.h>
using namespace std;

#define MAX_WORDS 50000
#define WORD_LEN 32

__global__ void countWords(char* words, int* freq, int totalWords) {
    int id = threadIdx.x + blockIdx.x * blockDim.x;
    if(id < totalWords) {
        atomicAdd(&freq[id], 1);
    }
}

int main() {
    vector<string> words;
    ifstream fin("input.txt");
    string w;
    while(fin >> w) words.push_back(w);
    fin.close();

    int n = words.size();
    if(n > MAX_WORDS) n = MAX_WORDS;

    char* h_words = new char[n * WORD_LEN];
    int* h_freq = new int[n];

    memset(h_freq, 0, n*sizeof(int));

    for(int i=0;i<n;i++)
        strncpy(&h_words[i*WORD_LEN], words[i].c_str(), WORD_LEN);

    char* d_words;
    int* d_freq;
    cudaMalloc(&d_words, n * WORD_LEN);
    cudaMalloc(&d_freq, n * sizeof(int));

    cudaMemcpy(d_words, h_words, n*WORD_LEN, cudaMemcpyHostToDevice);
    cudaMemcpy(d_freq, h_freq, n*sizeof(int), cudaMemcpyHostToDevice);
```

```
    cudaEvent_t start, stop;
    cudaEventCreate(&start);
    cudaEventCreate(&stop);
    cudaEventRecord(start);

    int threads = 256;
    int blocks = (n + threads - 1) / threads;
    countWords<<<blocks,threads>>>(d_words, d_freq, n);
    cudaDeviceSynchronize();

    cudaEventRecord(stop);
    cudaEventSynchronize(stop);

    float ms;
    cudaEventElapsedTime(&ms, start, stop);

    cudaMemcpy(h_freq, d_freq, n*sizeof(int), cudaMemcpyDeviceToHost);

    map<string,int> mp;
    for(int i=0;i<n;i++)
        mp[words[i]]++;

    vector<pair<int,string>> v;
    for(auto &p:mp)
        v.push_back({p.second,p.first});

    sort(v.rbegin(), v.rend());

    cout << "\nTop 10 words:\n";
    for(int i=0;i<min(10,(int)v.size());i++)
        cout << v[i].second << " -> " << v[i].first << endl;

    cout << "\nCUDA Time: " << ms << " ms\n";

    cudaFree(d_words);
    cudaFree(d_freq);
    delete[] h_words;
    delete[] h_freq;
}


/*
nvcc wordcount.cu -o wc_cuda
./wc_cuda

*/
```

Pattern count

```
%%writefile cuda_pattern.cu
#include <bits/stdc++.h>
#include <cuda.h>
using namespace std;
```

```
__global__ void countKernel(char* text, char* pat, int* count, int n, int m) {
    int idx = threadIdx.x + blockIdx.x * blockDim.x;

    if (idx + m <= n) {
        bool match = true;
        for (int j = 0; j < m; j++) {
            if (text[idx + j] != pat[j]) {
                match = false;
                break;
            }
        }
        if (match) atomicAdd(count, 1);
    }
}

int main() {
    ifstream fin("input.txt");
    string paragraph, line;
    while (getline(fin, line))
        paragraph += line + " ";
    fin.close();

    string pattern;
    cout << "Enter pattern (like %x%): ";
    cin >> pattern;
    pattern = pattern.substr(1, pattern.size() - 2);

    int n = paragraph.size();
    int m = pattern.size();

    char *d_text, *d_pat;
    int *d_count;

    cudaMalloc(&d_text, n);
    cudaMalloc(&d_pat, m);
    cudaMalloc(&d_count, sizeof(int));

    int zero = 0;
    cudaMemcpy(d_text, paragraph.c_str(), n, cudaMemcpyHostToDevice);
    cudaMemcpy(d_pat, pattern.c_str(), m, cudaMemcpyHostToDevice);
    cudaMemcpy(d_count, &zero, sizeof(int), cudaMemcpyHostToDevice);

    cudaEvent_t start, stop;
    cudaEventCreate(&start);
    cudaEventCreate(&stop);

    cudaEventRecord(start);

    int threads = 256;
    int blocks = (n + threads - 1) / threads;
    countKernel<<<blocks, threads>>>(d_text, d_pat, d_count, n, m);
    cudaDeviceSynchronize();

    cudaEventRecord(stop);
```

```
    cudaEventSynchronize(stop);

    float ms;
    cudaEventElapsedTime(&ms, start, stop);

    int result;
    cudaMemcpy(&result, d_count, sizeof(int), cudaMemcpyDeviceToHost);

    cout << "\nTotal Occurrences: " << result << endl;
    cout << "CUDA Time: " << ms << " ms\n";

    cudaFree(d_text);
    cudaFree(d_pat);
    cudaFree(d_count);
}
```