```
!pip install git+https://github.com/afnan47/cuda.git
```

```
  Collecting git+https://github.com/afnan47/cuda.git
    Cloning https://github.com/afnan47/cuda.git to /tmp/pip-req-build-nkywa1y3
    Running command git clone --filter=blob:none --quiet https://github.com/afn
    Resolved https://github.com/afnan47/cuda.git to commit aac710a35f52bb78ab34
    Preparing metadata (setup.py) ... done
  Building wheels for collected packages: NVCCPlugin
    Building wheel for NVCCPlugin (setup.py) ... done
    Created wheel for NVCCPlugin: filename=NVCCPlugin-0.0.2-py3-none-any.whl si
    Stored in directory: /tmp/pip-ephem-wheel-cache-_zu52z5t/wheels/aa/f3/44/e1
  Successfully built NVCCPlugin
  Installing collected packages: NVCCPlugin
  Successfully installed NVCCPlugin-0.0.2
```

```
%load_ext nvcc_plugin
```

```
  created output directory at /content/src
  Out bin /content/result.out
```

```
%%cu
#include <iostream>
int main(){
  std::cout << "Hello World\n";
  return 0;
}
```

```
Hello World
```

```
%%writefile ass1.cu

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

using namespace std;

class Graph {
    int V; // Number of vertices
    vector<vector<int>> adj; // Adjacency list
public:
    Graph(int V) : V(V), adj(V) {}
    // Add an edge to the graph
    void addEdge(int v, int w) {
        adj[v].push_back(w);
    }
    // Parallel Depth-First Search
    void parallelDFS(int startVertex) {
        vector<bool> visited(V, false);
        double startTime = omp_get_wtime();
        parallelDFSUtil(startVertex, visited);
        double endTime = omp_get_wtime();
        cout << "\nExecution Time (DFS): " << endTime - startTime << " seconds" <
    }
    // Parallel DFS utility function
    void parallelDFSUtil(int v, vector<bool>& visited) {
        visited[v] = true;
        cout << v << " ";
        #pragma omp parallel for
        for (int i = 0; i < adj[v].size(); ++i) {
            int n = adj[v][i];
            if (!visited[n])
                parallelDFSUtil(n, visited);
        }
    }
    // Parallel Breadth-First Search
    void parallelBFS(int startVertex) {
        vector<bool> visited(V, false);
        queue<int> q;
        double startTime = omp_get_wtime();
        visited[startVertex] = true;
        q.push(startVertex);
        while (!q.empty()) {
            int v = q.front();
            q.pop();
            cout << v << " ";
            #pragma omp parallel for
            for (int i = 0; i < adj[v].size(); ++i) {
                int n = adj[v][i];
                if (!visited[n]) {
                    visited[n] = true;
                    q.push(n);
                }
```

```
            }
        }
        double endTime = omp_get_wtime();
        cout << "\nExecution Time (BFS): " << endTime - startTime << " seconds" <
    }
};

int main() {
    // Create a graph
    Graph g(7);
    g.addEdge(0, 1);
    g.addEdge(0, 2);
    g.addEdge(1, 3);
    g.addEdge(1, 4);
    g.addEdge(2, 5);
    g.addEdge(2, 6);
    cout << "Depth-First Search (DFS): ";
    g.parallelDFS(0);
    cout << endl;
    cout << "Breadth-First Search (BFS): ";
    g.parallelBFS(0);
    cout << endl;
    return 0;
}
```

```
Writing ass1.cu
```

```
!nvcc -o ass1 ass1.cu -Xcompiler -fopenmp
!./ass1
```

```
Depth-First Search (DFS): 0 2 1 3 4 5 6
Execution Time (DFS): 0.00201439 seconds

Breadth-First Search (BFS): 0 1 2 4 3 6 5
Execution Time (BFS): 8.4151e-05 seconds
```

```
# Assignment 2 parallel bubble sort

%%writefile ass2parallelbubble.cu

#include<iostream>
#include<omp.h>
using namespace std;
void bubble(int array[], int n){
for (int i = 0; i < n - 1; i++){
for (int j = 0; j < n - i - 1; j++){
if (array[j] > array[j + 1]) swap(array[j], array[j + 1]);
}
}
}
void pBubble(int array[], int n){
//Sort odd indexed numbers
for(int i = 0; i < n; ++i){
#pragma omp for
for (int j = 1; j < n; j += 2){
if (array[j] < array[j-1])
{
swap(array[j], array[j - 1]);
}
}
// Synchronize
#pragma omp barrier
//Sort even indexed numbers
#pragma omp for
for (int j = 2; j < n; j += 2){
if (array[j] < array[j-1])
{
swap(array[j], array[j - 1]);
}
}
}
}
void printArray(int arr[], int n){
for(int i = 0; i < n; i++) cout << arr[i] << " ";
cout << "\n";
}
int main(){
// Set up variables
int n = 10;
int arr[n];
int brr[n];
double start_time, end_time;
// Create an array with numbers starting from n to 1
for(int i = 0, j = n; i < n; i++, j--) arr[i] = j;
// Sequential time
start_time = omp_get_wtime();
bubble(arr, n);
end_time = omp_get_wtime();
cout << "Sequential Bubble Sort took : " << end_time - start_time << " seconds.\n
printArray(arr, n);
// Reset the array
```

```
for(int i = 0, j = n; i < n; i++, j--) arr[i] = j;
// Parallel time
start_time = omp_get_wtime();
pBubble(arr, n);
end_time = omp_get_wtime();
cout << "Parallel Bubble Sort took : " << end_time - start_time << " seconds.\n";
printArray(arr, n);
}
```

```
Writing ass2parallelbubble.cu
```

```
!nvcc -o ass2parallelbubble ass2parallelbubble.cu -Xcompiler -fopenmp
!./ass2parallelbubble
```

```
ass2parallelbubble.cu(42): warning #177-D: variable "brr" was declared but ne
  int brr[n];
      ^

Remark: The warnings can be suppressed with "-diag-suppress <warning-number>"

Sequential Bubble Sort took : 9.63e-07 seconds.
1 2 3 4 5 6 7 8 9 10
Parallel Bubble Sort took : 2.513e-06 seconds.
1 2 3 4 5 6 7 8 9 10
```

```
%%writefile ass2parallelmergesort.cu

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

void merge(int arr[], int low, int mid, int high) {
    // Create arrays of left and right partitions
    int n1 = mid - low + 1;
    int n2 = high - mid;
    int left[n1];
    int right[n2];

    // Copy all left elements
    for (int i = 0; i < n1; i++)
        left[i] = arr[low + i];

    // Copy all right elements
    for (int j = 0; j < n2; j++)
        right[j] = arr[mid + 1 + j];

    // Compare and place elements
    int i = 0, j = 0, k = low;
    while (i < n1 && j < n2) {
        if (left[i] <= right[j]) {
            arr[k] = left[i];
            i++;
        } else {
            arr[k] = right[j];
            j++;
        }
        k++;
    }

    // If any elements are left out
    while (i < n1) {
        arr[k] = left[i];
        i++;
        k++;
    }
    while (j < n2) {
        arr[k] = right[j];
        j++;
        k++;
    }
}

void parallelMergeSort(int arr[], int low, int high) {
    if (low < high) {
        int mid = (low + high) / 2;
#pragma omp parallel sections
        {
#pragma omp section
            {
                parallelMergeSort(arr, low, mid);
```

```
            }
#pragma omp section
            {
                parallelMergeSort(arr, mid + 1, high);
            }
        }
        merge(arr, low, mid, high);
    }
}

void mergeSort(int arr[], int low, int high) {
    if (low < high) {
        int mid = (low + high) / 2;
        mergeSort(arr, low, mid);
        mergeSort(arr, mid + 1, high);
        merge(arr, low, mid, high);
    }
}

int main() {
    int n = 1000;
    int arr[n];
    double start_time, end_time;

    // Create an array with numbers starting from n to 1.
    for (int i = 0, j = n; i < n; i++, j--)
        arr[i] = j;

    // Measure Sequential Time
    start_time = omp_get_wtime();
    mergeSort(arr, 0, n - 1);
    end_time = omp_get_wtime();
    cout << "Time taken by sequential algorithm: " << end_time - start_time << "

    // Reset the array
    for (int i = 0, j = n; i < n; i++, j--)
        arr[i] = j;

    // Measure Parallel time
    start_time = omp_get_wtime();
    parallelMergeSort(arr, 0, n - 1);
    end_time = omp_get_wtime();
    cout << "Time taken by parallel algorithm: " << end_time - start_time << " se

    return 0;
}
```

```
Writing ass2parallelmergesort.cu
```

```
!nvcc -o ass2parallelmergesort ass2parallelmergesort.cu -Xcompiler -fopenmp
!./ass2parallelmergesort
```

```
Time taken by sequential algorithm: 0.000125365 seconds
Time taken by parallel algorithm: 0.00284824 seconds
```

```
%%writefile ass3.cu

#include <iostream>

__global__ void minKernel(int *arr, int *minval, int n) {
    int tid = threadIdx.x + blockIdx.x * blockDim.x;
    if (tid < n) atomicMin(minval, arr[tid]);
}

__global__ void maxKernel(int *arr, int *maxval, int n) {
    int tid = threadIdx.x + blockIdx.x * blockDim.x;
    if (tid < n) atomicMax(maxval, arr[tid]);
}

__global__ void sumKernel(int *arr, int *sum, int n) {
    int tid = threadIdx.x + blockIdx.x * blockDim.x;
    if (tid < n) atomicAdd(sum, arr[tid]);
}

int minval(int arr[], int n) {
    int *d_arr, *d_minval;
    cudaMalloc(&d_arr, n * sizeof(int));
    cudaMalloc(&d_minval, sizeof(int));
    cudaMemcpy(d_arr, arr, n * sizeof(int), cudaMemcpyHostToDevice);
    cudaMemcpy(d_minval, &arr[0], sizeof(int), cudaMemcpyHostToDevice);

    minKernel<<<(n + 255) / 256, 256>>>(d_arr, d_minval, n);

    int minval;
    cudaMemcpy(&minval, d_minval, sizeof(int), cudaMemcpyDeviceToHost);

    cudaFree(d_arr);
    cudaFree(d_minval);

    return minval;
}

int maxval(int arr[], int n) {
    int *d_arr, *d_maxval;
    cudaMalloc(&d_arr, n * sizeof(int));
    cudaMalloc(&d_maxval, sizeof(int));
    cudaMemcpy(d_arr, arr, n * sizeof(int), cudaMemcpyHostToDevice);
    cudaMemcpy(d_maxval, &arr[0], sizeof(int), cudaMemcpyHostToDevice);

    maxKernel<<<(n + 255) / 256, 256>>>(d_arr, d_maxval, n);

    int maxval;
    cudaMemcpy(&maxval, d_maxval, sizeof(int), cudaMemcpyDeviceToHost);

    cudaFree(d_arr);
    cudaFree(d_maxval);

    return maxval;
}
```

```
int sum(int arr[], int n) {
    int *d_arr, *d_sum;
    cudaMalloc(&d_arr, n * sizeof(int));
    cudaMalloc(&d_sum, sizeof(int));
    cudaMemcpy(d_arr, arr, n * sizeof(int), cudaMemcpyHostToDevice);
    cudaMemcpy(d_sum, &arr[0], sizeof(int), cudaMemcpyHostToDevice);

    sumKernel<<<(n + 255) / 256, 256>>>(d_arr, d_sum, n);

    int sum;
    cudaMemcpy(&sum, d_sum, sizeof(int), cudaMemcpyDeviceToHost);

    cudaFree(d_arr);
    cudaFree(d_sum);

    return sum;
}

int average(int arr[], int n) {
    return (double)sum(arr, n) / n;
}

int main() {
    int n = 5;
    int arr[] = {1, 2, 3, 4, 5};

    std::cout << "The minimum value is: " << minval(arr, n) << '\n';
    std::cout << "The maximum value is: " << maxval(arr, n) << '\n';
    std::cout << "The summation is: " << sum(arr, n) << '\n';
    std::cout << "The average is: " << average(arr, n) << '\n';

    return 0;
}
```

```
Overwriting ass3.cu
```

```
!nvcc -o ass3 ass3.cu -Xcompiler -fopenmp
!./ass3
```

```
The minimum value is: 1
The maximum value is: 5
The summation is: 16
The average is: 3
```

```
# ass4 addition of two large vector
%%writefile ass4twolargevector.cu


#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);
```

```
!nvcc -o ass4twolargevector ass4twolargevector.cu -Xcompiler -fopenmp
!./ass4twolargevector
```

```
    Vector A: 3 6 7 5
    Vector B: 3 5 6 2
    Addition: 6 11 13 7

    Overwriting ass4twolargevector.cu
```

```
# ass 4 matrix multiplication
%%writefile ass4matrixmultiplication.cu
```