```
!nvcc --version
!pip install git+https://github.com/afnan47/cuda.git
%load_ext nvcc_plugin
```

```
nvcc: NVIDIA (R) Cuda compiler driver
Copyright (c) 2005-2023 NVIDIA Corporation
Built on Tue_Aug_15_22:02:13_PDT_2023
Cuda compilation tools, release 12.2, V12.2.140
Build cuda_12.2.r12.2/compiler.33191640_0
Collecting git+https://github.com/afnan47/cuda.git
  Cloning https://github.com/afnan47/cuda.git to /tmp/pip-req-build-un13y4o
  Running command git clone --filter=blob:none --quiet https://github.com/a
  Resolved https://github.com/afnan47/cuda.git to commit aac710a35f52bb78ab
  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
  Stored in directory: /tmp/pip-ephem-wheel-cache-37nq9loz/wheels/aa/f3/44/
Successfully built NVCCPlugin
Installing collected packages: NVCCPlugin
Successfully installed NVCCPlugin-0.0.2
created output directory at /content/src
Out bin /content/result.out
```

```
%%writefile add.cu
#include <iostream>
#include <cuda_runtime.h>
#include <cstdlib>

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];
    }
}

int main() {
    int N;
    cout << "Enter the size of the vectors: ";
    cin >> N;

    int* A, * B, * C;
    size_t vectorBytes = N * sizeof(int);

    A = new int[N];
    B = new int[N];
    C = new int[N];

    for (int i = 0; i < N; i++) {
        A[i] = rand() % 10;
        B[i] = rand() % 10;
```

```
    }

    int* X, * Y, * Z;
    cudaMalloc(&X, vectorBytes);
    cudaMalloc(&Y, vectorBytes);
    cudaMalloc(&Z, vectorBytes);

    cudaMemcpy(X, A, vectorBytes, cudaMemcpyHostToDevice);
    cudaMemcpy(Y, B, vectorBytes, cudaMemcpyHostToDevice);

    add<<<1, N>>>(X, Y, Z, N);

    cudaMemcpy(C, Z, vectorBytes, cudaMemcpyDeviceToHost);

    cout << "Vector A:";
    for (int i = 0; i < N; i++) {
        cout << " " << A[i];
    }
    cout << endl;

    cout << "Vector B:";
    for (int i = 0; i < N; i++) {
        cout << " " << B[i];
    }
    cout << endl;

    cout << "Addition:";
    for (int i = 0; i < N; i++) {
        cout << " " << C[i];
    }
    cout << endl;

    delete[] A;
    delete[] B;
    delete[] C;

    cudaFree(X);
    cudaFree(Y);
    cudaFree(Z);

    return 0;
}
```

```
Writing add.cu
```

```
!nvcc add.cu -o add
!./add
```

```
Enter the size of the vectors: 2
Vector A: 3 7
Vector B: 6 5
Addition: 0 0
```

```
%%writefile matrix_multi.cu
```

```
%%writefile matrix_multi.cu
#include <iostream>
#include <cuda_runtime.h>

using namespace std;

const int N = 2;

__global__ void matrixMultiply(int* A, int* B, int* C) {
    int row = blockIdx.y * blockDim.y + threadIdx.y;
    int col = blockIdx.x * blockDim.x + threadIdx.x;

    if (row < N && col < N) {
        int sum = 0;
        for (int i = 0; i < N; ++i) {
            sum += A[row * N + i] * B[i * N + col];
        }
        C[row * N + col] = sum;
    }
}

int main() {
    int* A, * B, * C;
    size_t matrixBytes = N * N * sizeof(int);

    A = new int[N * N];
    B = new int[N * N];
    C = new int[N * N];

    auto input = [&](int* matrix) {
        cout << "Enter elements of Matrix (" << N << "x" << N << "):" << endl;
        for (int i = 0; i < N * N; ++i) cin >> matrix[i];
    };

    input(A);
    input(B);

    int* X, * Y, * Z;
    cudaMalloc(&X, matrixBytes);
    cudaMalloc(&Y, matrixBytes);
    cudaMalloc(&Z, matrixBytes);

    cudaMemcpy(X, A, matrixBytes, cudaMemcpyHostToDevice);
    cudaMemcpy(Y, B, matrixBytes, cudaMemcpyHostToDevice);

    matrixMultiply<<<1, dim3(N, N)>>>(X, Y, Z);

    cudaMemcpy(C, Z, matrixBytes, cudaMemcpyDeviceToHost);

    cout << "Output- Matrix size: " << N << "x" << N << endl;
    cout << "Input Matrix 1:" << endl;
    for (int i = 0; i < N; ++i) {
        for (int j = 0; j < N; ++j) cout << A[i * N + j] << " ";
        cout << endl;
    }
```

```
    }

    cout << "Input Matrix 2:" << endl;
    for (int i = 0; i < N; ++i) {
        for (int j = 0; j < N; ++j) cout << B[i * N + j] << " ";
        cout << endl;
    }

    cout << "Resultant matrix:" << endl;
    for (int i = 0; i < N; ++i) {
        for (int j = 0; j < N; ++j) cout << C[i * N + j] << " ";
        cout << endl;
    }

    cout << "Finished." << endl;

    delete[] A;
    delete[] B;
    delete[] C;

    cudaFree(X);
    cudaFree(Y);
    cudaFree(Z);

    return 0;
}
```

```
Writing matrix_multi.cu
```

```
!nvcc matrix_multi.cu -o matrix_multi
!./matrix_multi
```

```
Enter elements of Matrix (2x2):
```

```
%%writefile smma.cu
#include <iostream>
#include <vector>
#include <climits>

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

__global__ void max_reduction_kernel(int* arr, int size, int* result) {
    int tid = blockIdx.x * blockDim.x + threadIdx.x;
    if (tid < size) {
        atomicMax(result, arr[tid]);
    }
}
```

```
__global__ void sum_reduction_kernel(int* arr, int size, int* result) {
    int tid = blockIdx.x * blockDim.x + threadIdx.x;
    if (tid < size) {
        atomicAdd(result, arr[tid]);
    }
}

int main() {
    std::vector<int> arr = {5, 2, 9, 1, 7, 6, 8, 3, 4};
    int size = arr.size();
    int* d_arr;
    int* d_result_min, * d_result_max, * d_result_sum;
    int result_min = INT_MAX, result_max = INT_MIN, result_sum = 0;

    cudaMalloc(&d_arr, size * sizeof(int));
    cudaMemcpy(d_arr, arr.data(), size * sizeof(int), cudaMemcpyHostToDevice);

    cudaMalloc(&d_result_min, sizeof(int));
    cudaMalloc(&d_result_max, sizeof(int));
    cudaMalloc(&d_result_sum, sizeof(int));

    cudaMemcpy(d_result_min, &result_min, sizeof(int), cudaMemcpyHostToDevice);
    cudaMemcpy(d_result_max, &result_max, sizeof(int), cudaMemcpyHostToDevice);
    cudaMemcpy(d_result_sum, &result_sum, sizeof(int), cudaMemcpyHostToDevice);

    min_reduction_kernel<<<(size + 255) / 256, 256>>>(d_arr, size, d_result_mi
    max_reduction_kernel<<<(size + 255) / 256, 256>>>(d_arr, size, d_result_ma
    sum_reduction_kernel<<<(size + 255) / 256, 256>>>(d_arr, size, d_result_su

    cudaMemcpy(&result_min, d_result_min, sizeof(int), cudaMemcpyDeviceToHost);
    cudaMemcpy(&result_max, d_result_max, sizeof(int), cudaMemcpyDeviceToHost);
    cudaMemcpy(&result_sum, d_result_sum, sizeof(int), cudaMemcpyDeviceToHost);

    std::cout << "Minimum value: " << result_min << std::endl;
    std::cout << "Maximum value: " << result_max << std::endl;
    std::cout << "Sum: " << result_sum << std::endl;
    std::cout << "Average: " << static_cast<double>(result_sum) / size << std:::

    cudaFree(d_arr);
    cudaFree(d_result_min);
    cudaFree(d_result_max);
    cudaFree(d_result_sum);

    return 0;
}
```

```
!nvcc smma.cu -o smma
!./smma
```

```
%%writefile bfsdfs.cu
#include <iostream>
#include <vector>
#include <queue>
```

```
#include <queue>
#include <stack>

__global__ void bfs_kernel(int* adjList, int* visited, int* queue, int* queueSi
    int tid = blockIdx.x * blockDim.x + threadIdx.x;
    if (tid < *queueSize) {
        int u = queue[tid];
        if (!visited[u]) {
            visited[u] = 1;
            for (int i = adjList[u]; i < adjList[u + 1]; ++i) {
                int v = adjList[i];
                if (!visited[v]) {
                    int idx = atomicAdd(queueSize, 1);
                    queue[idx] = v;
                }
            }
        }
    }
}

__global__ void dfs_kernel(int* adjList, int* visited, int* stack, int* stackSi
    int tid = blockIdx.x * blockDim.x + threadIdx.x;
    if (tid < *stackSize) {
        int u = stack[tid];
        if (!visited[u]) {
            visited[u] = 1;
            for (int i = adjList[u]; i < adjList[u + 1]; ++i) {
                int v = adjList[i];
                if (!visited[v]) {
                    int idx = atomicAdd(stackSize, 1);
                    stack[idx] = v;
                }
            }
        }
    }
}

int main() {
    int n, m;
    std::cout << "Enter the number of vertices: ";
    std::cin >> n;
    std::cout << "Enter the number of edges: ";
    std::cin >> m;

    // Assuming graph is represented as an adjacency list
    std::vector<std::vector<int>> adjList(n + 1);
    std::cout << "Enter the edges (format: u v):\n";
    for (int i = 0; i < m; ++i) {
        int u, v;
        std::cin >> u >> v;
        adjList[u].push_back(v);
        adjList[v].push_back(u); // Assuming an undirected graph
    }

    // Allocate memory on the GPU
```

```
    // Allocate memory on the GPU
    int* d_adjList, * d_visited, * d_queue, * d_queueSize, * d_stack, * d_stack
    cudaMalloc(&d_adjList, (2 * m) * sizeof(int)); // Each edge is stored twice
    cudaMalloc(&d_visited, n * sizeof(int));
    cudaMalloc(&d_queue, n * sizeof(int));
    cudaMalloc(&d_queueSize, sizeof(int));
    cudaMalloc(&d_stack, n * sizeof(int));
    cudaMalloc(&d_stackSize, sizeof(int));

    // Initialize data on the GPU

    // Perform BFS traversal
    int start;
    std::cout << "Enter the starting vertex for BFS: ";
    std::cin >> start;
    cudaMemcpy(d_queue, &start, sizeof(int), cudaMemcpyHostToDevice);
    cudaMemcpy(d_queueSize, &start, sizeof(int), cudaMemcpyHostToDevice);
    int queueSize = 1;
    while (queueSize > 0) {
        bfs_kernel<<<(queueSize + 255) / 256, 256>>>(d_adjList, d_visited, d_qu
        cudaMemcpy(&queueSize, d_queueSize, sizeof(int), cudaMemcpyDeviceToHost
    }

    // Perform DFS traversal
    std::cout << "Enter the starting vertex for DFS: ";
    std::cin >> start;
    cudaMemcpy(d_visited, &start, sizeof(int), cudaMemcpyHostToDevice);
    cudaMemcpy(d_stack, &start, sizeof(int), cudaMemcpyHostToDevice);
    cudaMemcpy(d_stackSize, &start, sizeof(int), cudaMemcpyHostToDevice);
    int stackSize = 1;
    while (stackSize > 0) {
        dfs_kernel<<<(stackSize + 255) / 256, 256>>>(d_adjList, d_visited, d_st
        cudaMemcpy(&stackSize, d_stackSize, sizeof(int), cudaMemcpyDeviceToHost
    }

    // Copy visited array back to host
    int* h_visited = new int[n];
    cudaMemcpy(h_visited, d_visited, n * sizeof(int), cudaMemcpyDeviceToHost);

    // Print BFS traversal result
    std::cout << "BFS traversal starting from vertex " << start << ":\n";
    for (int i = 0; i < n; ++i) {
        if (h_visited[i]) {
            std::cout << i << " ";
        }
    }
    std::cout << std::endl;

    // Print DFS traversal result
    std::cout << "DFS traversal starting from vertex " << start << ":\n";
    for (int i = 0; i < n; ++i) {
        if (h_visited[i]) {
            std::cout << i << " ";
        }
    }
```

```
    std::cout << std::endl;

    delete[] h_visited;

    // Free memory on the GPU
    cudaFree(d_adjList);
    cudaFree(d_visited);
    cudaFree(d_queue);
    cudaFree(d_queueSize);
    cudaFree(d_stack);
    cudaFree(d_stackSize);

    return 0;
}

!nvcc bfsdfs.cu -o bfsdfs
!./bfsdfs

%%writefile bubblesort.cu
#include <iostream>
#include <vector>
#include <chrono>

__global__ void bubbleSortParallel(int* arr, int n) {
    int idx = blockIdx.x * blockDim.x + threadIdx.x;
    if (idx < n - 1) {
        if (arr[idx] > arr[idx + 1]) {
            int temp = arr[idx];
            arr[idx] = arr[idx + 1];
            arr[idx + 1] = temp;
        }
    }
}

void bubbleSortSerial(std::vector<int>& arr) {
    int n = arr.size();
    bool swapped = true;
    while (swapped) {
        swapped = false;
        for (int i = 0; i < n - 1; i++) {
            if (arr[i] > arr[i + 1]) {
                std::swap(arr[i], arr[i + 1]);
                swapped = true;
            }
        }
    }
}

int main() {
    int n = 10000;
    int block_size = 256;
    int num_blocks = (n + block_size - 1) / block_size;
```

```
    std::vector<int> arr(n);

    // Initialize array with random values
    for (int i = 0; i < n; i++) {
        arr[i] = rand() % 10000;
    }

    // Measure serial Bubble Sort performance
    auto start = std::chrono::high_resolution_clock::now();
    bubbleSortSerial(arr);
    auto stop = std::chrono::high_resolution_clock::now();
    auto durationSerial = std::chrono::duration_cast<std::chrono::milliseconds>

    std::cout << "Serial Bubble Sort took " << durationSerial.count() << " mill

    // Reset array for parallel sort
    for (int i = 0; i < n; i++) {
        arr[i] = rand() % 10000;
    }

    int* d_arr;
    cudaMalloc(&d_arr, n * sizeof(int));
    cudaMemcpy(d_arr, arr.data(), n * sizeof(int), cudaMemcpyHostToDevice);

    // Measure parallel Bubble Sort performance
    start = std::chrono::high_resolution_clock::now();
    for (int i = 0; i < n; i++) {
        bubbleSortParallel<<<num_blocks, block_size>>>(d_arr, n);
        cudaDeviceSynchronize();
    }
    stop = std::chrono::high_resolution_clock::now();
    auto durationParallel = std::chrono::duration_cast<std::chrono::millisecond

    std::cout << "Parallel Bubble Sort took " << durationParallel.count() << "

    cudaMemcpy(arr.data(), d_arr, n * sizeof(int), cudaMemcpyDeviceToHost);
    cudaFree(d_arr);

    return 0;
}

!nvcc bubblesort.cu -o bubblesort
!./bubblesort

%%writefile mergesort.cu
#include <iostream>
#include <vector>
#include <chrono>

// Serial merge sort implementation
void merge(int* arr, int l, int m, int r) {
    // Merge logic
}
```

```
void mergeSort(int* arr, int l, int r) {
    // Merge sort logic
}

// Parallel merge sort implementation
__global__ void mergeSortParallel(int* arr, int l, int r) {
    // Merge sort logic
}

int main() {
    int n = 10000;
    int block_size = 256;
    int num_blocks = (n + block_size - 1) / block_size;

    std::vector<int> arr_serial(n);
    std::vector<int> arr_parallel(n);

    // Initialize arrays with random values
    // Copy values from arr_serial to arr_parallel for comparison

    // Serial merge sort
    auto start_serial = std::chrono::high_resolution_clock::now();
    mergeSort(arr_serial.data(), 0, n - 1);
    auto end_serial = std::chrono::high_resolution_clock::now();

    // Parallel merge sort
    int* d_arr;
    cudaMalloc(&d_arr, n * sizeof(int));
    cudaMemcpy(d_arr, arr_parallel.data(), n * sizeof(int), cudaMemcpyHostToDevi

    auto start_parallel = std::chrono::high_resolution_clock::now();
    mergeSortParallel<<<num_blocks, block_size>>>(d_arr, 0, n - 1);
    cudaDeviceSynchronize();
    auto end_parallel = std::chrono::high_resolution_clock::now();

    cudaMemcpy(arr_parallel.data(), d_arr, n * sizeof(int), cudaMemcpyDeviceToHo
    cudaFree(d_arr);

    // Print timing information
    std::chrono::duration<double, std::milli> duration_serial = end_serial - sta
    std::cout << "Serial Merge Sort took " << duration_serial.count() << " milli

    std::chrono::duration<double, std::milli> duration_parallel = end_parallel -
    std::cout << "Parallel Merge Sort took " << duration_parallel.count() << " m

    return 0;
}
```

```
!nvcc mergesort.cu -o mergesort
!./mergesort
```

Start coding or generate with AI.