```
# Check CUDA version
!nvcc --version

# Install CUDA package
!pip install git+https://github.com/afnan47/cuda.git

# Load nvcc plugin
%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-y3zqntv
  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-cecdsfcj/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
```

```
#Addition of Two Large Vectors

%%writefile add.cu
#include <iostream>
#include <cstdlib> // Include <cstdlib> for rand()
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);

    // Allocate host memory
    A = new int[vectorSize];
    B = new int[vectorSize];
    C = new int[vectorSize];

    // Initialize host arrays
    initialize(A, vectorSize);
    initialize(B, vectorSize);
    cout << "Vector A: ";
    print(A, N);
    cout << "Vector B: ";
    print(B, N);

    int* X, * Y, * Z;
    // Allocate device memory
    cudaMalloc(&X, vectorBytes);
    cudaMalloc(&Y, vectorBytes);
    cudaMalloc(&Z, vectorBytes);

    // Check for CUDA memory allocation errors
    if (X == nullptr || Y == nullptr || Z == nullptr) {
        cerr << "CUDA memory allocation failed" << endl;
        return 1;
    }

    // Copy data from host to device
    cudaMemcpy(X, A, vectorBytes, cudaMemcpyHostToDevice);
    cudaMemcpy(Y, B, vectorBytes, cudaMemcpyHostToDevice);

    int threadsPerBlock = 256;
    int blocksPerGrid = (N + threadsPerBlock - 1) / threadsPerBlock;

    // Launch kernel
    add<<<blocksPerGrid, threadsPerBlock>>>(X, Y, Z, N);

    // Check for kernel launch errors
    cudaError_t kernelLaunchError = cudaGetLastError();
    if (kernelLaunchError != cudaSuccess) {
        cerr << "CUDA kernel launch failed: " << cudaGetErrorString(kernelLaunc
        return 1;
    }

    // Copy result from device to host
    cudaMemcpy(C, Z, vectorBytes, cudaMemcpyDeviceToHost);
```

```
    // Check for CUDA memcpy errors
    cudaError_t memcpyError = cudaGetLastError();
    if (memcpyError != cudaSuccess) {
        cerr << "CUDA memcpy failed: " << cudaGetErrorString(memcpyError) << er
        return 1;
    }

    cout << "Addition: ";
    print(C, N);

    // Free device memory
    cudaFree(X);
    cudaFree(Y);
    cudaFree(Z);

    // Free host memory
    delete[] A;
    delete[] B;
    delete[] C;

    return 0;
}
```

```
Writing add.cu
```

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

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

```
#Matrix multiplication usin CUDA
%%writefile matrix_mult.cu
#include <iostream>
#include <cuda.h>
using namespace std;

#define BLOCK_SIZE 2

__global__ void gpuMM(float *A, float *B, float *C, int N) {
    int row = blockIdx.y * blockDim.y + threadIdx.y;
    int col = blockIdx.x * blockDim.x + threadIdx.x;
    float sum = 0.f;
    for (int n = 0; n < N; ++n)
        sum += A[row * N + n] * B[n * N + col];
    C[row * N + col] = sum;
}

int main(int argc, char *argv[]) {
    int N;
    float K;

    // Perform matrix multiplication C = A*B
```

```
    // where A, B and C are NxN matrices
    // Restricted to matrices where N = K*BLOCK_SIZE;

    cout << "Enter a value for size/2 of matrix: ";
    cin >> K;
    K = 1;
    N = K * BLOCK_SIZE;
    cout << "\nExecuting Matrix Multiplication" << endl;
    cout << "Matrix size: " << N << "x" << N << endl;

    // Allocate memory on the host
    float *hA, *hB, *hC;
    hA = new float[N * N];
    hB = new float[N * N];
    hC = new float[N * N];

    // Initialize matrices on the host with random values
    srand(time(NULL)); // Seed the random number generator
    for (int j = 0; j < N; j++) {
        for (int i = 0; i < N; i++) {
            hA[j * N + i] = rand() % 10; // Generate random value between 0 anc
            hB[j * N + i] = rand() % 10; // Generate random value between 0 anc
        }
    }

    // Allocate memory on the device
    int size = N * N * sizeof(float);
    float *dA, *dB, *dC;
    cudaMalloc(&dA, size);
    cudaMalloc(&dB, size);
    cudaMalloc(&dC, size);

    dim3 threadBlock(BLOCK_SIZE, BLOCK_SIZE);
    dim3 grid(K, K);

    // Copy matrices from the host to device
    cudaMemcpy(dA, hA, size, cudaMemcpyHostToDevice);
    cudaMemcpy(dB, hB, size, cudaMemcpyHostToDevice);

    // Execute the matrix multiplication kernel
    gpuMM<<<grid, threadBlock>>>(dA, dB, dC, N);

    // Copy the GPU result back to CPU
    cudaMemcpy(hC, dC, size, cudaMemcpyDeviceToHost);

    // Display the result
    cout << "\nResultant matrix:\n";
    for (int row = 0; row < N; row++) {
        for (int col = 0; col < N; col++) {
            cout << hC[row * N + col] << " ";
        }
        cout << endl;
    }
```

```
    // Free device memory
    cudaFree(dA);
    cudaFree(dB);
    cudaFree(dC);

    // Free host memory
    delete[] hA;
    delete[] hB;
    delete[] hC;

    cout << "Finished." << endl;
    return 0;
}
```

```
Overwriting matrix_mult.cu
```

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

```
Enter a value for size/2 of matrix: 2

Executing Matrix Multiplication
Matrix size: 2x2

Resultant matrix:
60 28
20 12
Finished.
```

```
#Min,Max, Sum and Average Operations

%%writefile sum.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]);
```

```
    }
}

__global__ void average_reduction_kernel(int* arr, int size, int* sum) {
    int tid = blockIdx.x * blockDim.x + threadIdx.x;
    if (tid < size) {
        atomicAdd(sum, 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;
    int result_min = INT_MAX;
    int result_max = INT_MIN;
    int result_sum = 0;

    // Allocate memory on the device
    cudaMalloc(&d_arr, size * sizeof(int));
    cudaMalloc(&d_result, sizeof(int));

    // Copy data from host to device
    cudaMemcpy(d_arr, arr.data(), size * sizeof(int), cudaMemcpyHostToDevice);
    cudaMemcpy(d_result, &result_min, sizeof(int), cudaMemcpyHostToDevice);

    // Perform min reduction
    min_reduction_kernel<<<(size + 255) / 256, 256>>>(d_arr, size, d_result);
    cudaMemcpy(&result_min, d_result, sizeof(int), cudaMemcpyDeviceToHost);
    std::cout << "Minimum value: " << result_min << std::endl;

    // Perform max reduction
    cudaMemcpy(d_result, &result_max, sizeof(int), cudaMemcpyHostToDevice);
    max_reduction_kernel<<<(size + 255) / 256, 256>>>(d_arr, size, d_result);
    cudaMemcpy(&result_max, d_result, sizeof(int), cudaMemcpyDeviceToHost);
    std::cout << "Maximum value: " << result_max << std::endl;

    // Perform sum reduction
    cudaMemcpy(d_result, &result_sum, sizeof(int), cudaMemcpyHostToDevice);
    sum_reduction_kernel<<<(size + 255) / 256, 256>>>(d_arr, size, d_result);
    cudaMemcpy(&result_sum, d_result, sizeof(int), cudaMemcpyDeviceToHost);
    std::cout << "Sum: " << result_sum << std::endl;

    // Perform average reduction
    cudaMemcpy(d_result, &result_sum, sizeof(int), cudaMemcpyHostToDevice);
    average_reduction_kernel<<<(size + 255) / 256, 256>>>(d_arr, size, d_result
    cudaMemcpy(&result_sum, d_result, sizeof(int), cudaMemcpyDeviceToHost);
    std::cout << "Average: " << static_cast<double>(result_sum) / size << std::

    // Free device memory
    cudaFree(d_arr);
    cudaFree(d_result);

    return 0;
```

```
    return 0;
}
```

```
Writing sum.cu
```

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

```
Minimum value: 1
Maximum value: 9
Sum: 45
Average: 10
```

```
#Parallel Bubble Sort

%%writefile bu.cu
#include <iostream>
#include <vector>
#include <chrono>
using namespace std;

__device__ void device_swap(int& a, int& b) {
    int temp = a;
    a = b;
    b = temp;
}

__global__ void kernel_bubble_sort_odd_even(int* arr, int size) {
    bool isSorted = false;
    while (!isSorted) {
        isSorted = true;
        int tid = blockIdx.x * blockDim.x + threadIdx.x;
        if (tid % 2 == 0 && tid < size - 1) {
            if (arr[tid] > arr[tid + 1]) {
                device_swap(arr[tid], arr[tid + 1]);
                isSorted = false;
            }
        }
        __syncthreads(); // Synchronize threads within block

        if (tid % 2 != 0 && tid < size - 1) {
            if (arr[tid] > arr[tid + 1]) {
                device_swap(arr[tid], arr[tid + 1]);
                isSorted = false;
            }
        }
        __syncthreads(); // Synchronize threads within block
    }
}

void bubble_sort_odd_even(vector<int>& arr) {
    int size = arr.size();
    int* d_arr;
    cudaMalloc(&d_arr, size * sizeof(int));
    cudaMemcpy(d_arr, arr.data(), size * sizeof(int), cudaMemcpyHostToDevice);
```

```
    // Calculate grid and block dimensions
    int blockSize = 256;
    int gridSize = (size + blockSize - 1) / blockSize;

    // Perform bubble sort on GPU
    kernel_bubble_sort_odd_even<<<gridSize, blockSize>>>(d_arr, size);

    // Copy sorted array back to host
    cudaMemcpy(arr.data(), d_arr, size * sizeof(int), cudaMemcpyDeviceToHost);
    cudaFree(d_arr);
}

int main() {
    vector<int> arr = {5, 2, 9, 1, 7, 6, 8, 3, 4};
    double start, end;

    // Measure performance of parallel bubble sort using odd-even transposition
    start = chrono::duration_cast<chrono::milliseconds>(chrono::system_clock::no
    bubble_sort_odd_even(arr);
    end = chrono::duration_cast<chrono::milliseconds>(chrono::system_clock::now(

    cout << "Parallel bubble sort using odd-even transposition time: " << end -
    return 0;
}
```

```
Writing bu.cu
```

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

```
Parallel bubble sort using odd-even transposition time: 149 milliseconds
```

```
# BFS
%%writefile breadthfirst.cu
#include <iostream>
#include <queue>
#include <vector>
#include <omp.h>

using namespace std;

int main() {
    int num_vertices, num_edges, source;
    cout << "Enter number of vertices, edges, and source node: ";
    cin >> num_vertices >> num_edges >> source;

    // Input validation
    if (source < 1 || source > num_vertices) {
        cout << "Invalid source node!" << endl;
        return 1;
    }
```

```
    vector<vector<int>> adj_list(num_vertices + 1);
    for (int i = 0; i < num_edges; i++) {
        int u, v;
        cin >> u >> v;
        // Input validation for edges
        if (u < 1 || u > num_vertices || v < 1 || v > num_vertices) {
            cout << "Invalid edge: " << u << " " << v << endl;
            return 1;
        }
        adj_list[u].push_back(v);
        adj_list[v].push_back(u);
    }

    queue<int> q;
    vector<bool> visited(num_vertices + 1, false);
    q.push(source);
    visited[source] = true;

    while (!q.empty()) {
        int curr_vertex = q.front();
        q.pop();
        cout << curr_vertex << " ";

        // Sequential loop for neighbors
        for (int i = 0; i < adj_list[curr_vertex].size(); i++) {
            int neighbour = adj_list[curr_vertex][i];
            if (!visited[neighbour]) {
                visited[neighbour] = true;
                q.push(neighbour);
            }
        }
    }

    cout << endl;
    return 0;
}
```

```
Overwriting breadthfirst.cu
```

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

```
Enter number of vertices, edges, and source node: 5 4 1
1 2
1 3
2 4
3 5
1 2 3 4 5
```