HPC-3 :-

User

**You**

# Check CUDA version !nvcc --version . explain the code

ChatGPT

**ChatGPT**

This code snippet is using the command-line interface (CLI) to execute a command. Let's break it down:

1. **!**: This symbol indicates that the following text should be interpreted and executed as a command in the command-line interface. It's commonly used in Jupyter notebooks or similar environments to run shell commands from within the notebook.
2. **nvcc**: This is the NVIDIA CUDA Compiler (nvcc). It's a compiler for NVIDIA CUDA code, which is used to compile CUDA programs written in languages like CUDA C/C++.
3. **--version**: This is a command-line option passed to **nvcc**. It tells **nvcc** to print out its version information.

So, when you run this code, it will execute **nvcc --version**, which will output the version information of the CUDA compiler installed on your system. This is useful for checking which version of CUDA is installed, which is important for compatibility with CUDA-enabled software and libraries.

User

**You**

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

ChatGPT

**ChatGPT**

This code snippet is attempting to install a CUDA package from a GitHub repository using pip, Python's package installer. Let's break it down:

1. **!**: Similar to the previous explanation, this symbol indicates that the following text should be interpreted and executed as a command in the command-line interface. It's commonly used in Jupyter notebooks or similar environments.
2. **pip install**: This is the command to install Python packages using pip.
3. **git+https://github.com/afnan47/cuda.git**: This is the URL of the Git repository containing the CUDA package. By prefixing it with **git+**, pip knows to clone the repository and install the package from there.

So, when you run this code, pip will attempt to install the CUDA package from the specified GitHub repository.

User

**You**

# Load nvcc plugin %load\_ext nvcc\_plugin

ChatGPT

**ChatGPT**

This code snippet is using a Jupyter magic command **%load\_ext** to load the **nvcc\_plugin** extension. Let's break it down:

1. **%**: This symbol denotes a Jupyter magic command. Magic commands are special commands provided by IPython (the interactive Python shell used in Jupyter) that allow you to perform various tasks, such as loading extensions or changing configurations.
2. **load\_ext**: This is a magic command used to load Jupyter extensions.
3. **nvcc\_plugin**: This is the name of the extension being loaded. In this case, it's likely a custom Jupyter extension designed to enable the use of CUDA code within Jupyter notebooks. The extension probably provides functionality to compile and execute CUDA code cells directly in the notebook environment.

So, when you run this code, the **nvcc\_plugin** extension will be loaded, allowing you to work with CUDA code in your Jupyter notebook.

User

**You**

%%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() { int size; std::cout << "Enter the size of the array: "; std::cin >> size; std::vector<int> arr(size); for (int i = 0; i < size; ++i) { std::cout << "Enter element " << i << ": "; std::cin >> arr[i]; } 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 on CPU side double average = static\_cast<double>(result\_sum) / size; std::cout << "Average: " << average << std::endl; // Free device memory cudaFree(d\_arr); cudaFree(d\_result); return 0; } . explain the code line by line

ChatGPT

**ChatGPT**

This code is a CUDA program written in C++ for performing reduction operations (min, max, sum, and average) on an array of integers. Let's break it down line by line:

cppCopy code

#include <iostream> #include <vector> #include <climits>

* These are standard C++ header files for input/output operations (**iostream**), working with dynamic arrays (**vector**), and providing constants for integer types (**climits**).

cppCopy code

\_\_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]); } }

* This is a CUDA kernel function for performing minimum reduction. It takes three arguments: the array (**arr**), its size (**size**), and a pointer to store the result (**result**). Each thread calculates the minimum value within its assigned portion of the array and updates the result using the **atomicMin** function.

cppCopy code

\_\_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]); } }

* Similar to the previous kernel, this one performs maximum reduction by calculating the maximum value within each thread's assigned portion of the array and updating the result using the **atomicMax** function.

cppCopy code

\_\_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]); } }

* This kernel performs sum reduction by adding up all the elements in the array using the **atomicAdd** function.

cppCopy code

\_\_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]); } }

* This kernel is designed to be used for calculating the sum, which is then used to compute the average on the CPU side.

cppCopy code

int main() { int size; std::cout << "Enter the size of the array: "; std::cin >> size; std::vector<int> arr(size); for (int i = 0; i < size; ++i) { std::cout << "Enter element " << i << ": "; std::cin >> arr[i]; }

* This is the main function where the program starts execution. It first prompts the user to input the size of the array and then reads the elements of the array from the user.

cppCopy code

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);

* This section declares pointers for device memory (**d\_arr** and **d\_result**) and variables to store the results of reduction operations (**result\_min**, **result\_max**, and **result\_sum**). It then allocates memory on the device for the array and the result, and copies data from the host (CPU) to the device (GPU).

cppCopy code

// 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;

* These sections execute the minimum and maximum reduction kernels on the GPU, and then copy the results back to the host to print them.

cppCopy code

// 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 on CPU side double average = static\_cast<double>(result\_sum) / size; std::cout << "Average: " << average << std::endl;

* These sections execute the sum reduction kernel on the GPU and then calculate the average on the CPU side using the sum obtained from the GPU.

cppCopy code

// Free device memory cudaFree(d\_arr); cudaFree(d\_result); return 0; }

* Finally, the code frees the allocated device memory and returns 0 to indicate successful execution of the program.

User

**You**

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

ChatGPT

**ChatGPT**

These commands are running the CUDA compiler (**nvcc**) to compile the CUDA program **sum.cu** into an executable named **sum**, and then executing that executable.

* **!nvcc sum.cu -o sum**: This command compiles the CUDA program **sum.cu** into an executable named **sum** using the CUDA compiler (**nvcc**). The **-o** flag specifies the output file name.
* **!./sum**: This command executes the compiled program **sum** in the current directory. The **./** indicates that the program is located in the current directory, and **sum** is the name of the executable.

When you run these commands, the CUDA program will be compiled and then executed, performing the reduction operations on the array as specified in the code, and printing out the results.

User

**You**

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; explain the code

ChatGPT

**ChatGPT**

This portion of the code performs reduction operations for finding the minimum and maximum values in the array using CUDA kernels. Let's break it down:

cppCopy code

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;

* **min\_reduction\_kernel<<<(size + 255) / 256, 256>>>(d\_arr, size, d\_result);**: This line launches the CUDA kernel **min\_reduction\_kernel** with a grid configuration determined by **(size + 255) / 256** blocks and 256 threads per block. The number of blocks is calculated to ensure all elements of the array are covered by the threads. The kernel is passed the device pointer **d\_arr** (pointing to the array), the size of the array **size**, and a pointer **d\_result** to store the minimum value found by the kernel.
* **cudaMemcpy(&result\_min, d\_result, sizeof(int), cudaMemcpyDeviceToHost);**: After the kernel execution, this line copies the result from device memory (**d\_result**) to host memory (**&result\_min**), allowing the CPU to access the minimum value found by the kernel.
* **std::cout << "Minimum value: " << result\_min << std::endl;**: This line prints out the minimum value found by the CUDA kernel.

Similarly, the code block below performs the same operations for finding the maximum value:

cppCopy code

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;

Here, the only difference is the use of **max\_reduction\_kernel** instead of **min\_reduction\_kernel** to find the maximum value. After launching the kernel and copying the result back to the host, the maximum value is printed out.

HPC -4(1) :-

CudaMalloc() – is a function that is used to allocation the memory to pointers/variables on the GPU.

blockIdx.x – gives block id

threadIdx.x – gives the thread id

device to host means- GPU to CPU

CudaMemcpy(destination, source, memory to copy, constant)

Watch vedio youtube.

HPC- 4(2) :-

This CUDA program performs matrix multiplication on the GPU. Let's go through it line by line:

cppCopy code

#include<stdio.h> #include<cuda\_runtime.h>

* These are standard C libraries for input/output operations (**stdio.h**) and CUDA runtime APIs (**cuda\_runtime.h**).

cppCopy code

\_\_global\_\_ void matproduct(int \*l, int \*m, int \*n, int row1, int col1, int row2, int col2) { int x = blockIdx.x; int y = blockIdx.y; int k; n[col2\*y+x] = 0; for(k = 0; k < col1; k++) { n[col2\*y+x] += l[col1\*y+k] \* m[col2\*k+x]; } }

* This is the CUDA kernel function **matproduct**, which performs matrix multiplication. It takes pointers to two input matrices **l** and **m**, and an output matrix **n**, along with their dimensions. It uses **blockIdx.x** and **blockIdx.y** to determine the index of the element in the output matrix that the current thread will compute. It then performs the matrix multiplication and stores the result in the output matrix **n**.

cppCopy code

int main() { int row1, row2, col1, col2; printf("Enter the number of rows in matrix 1: "); scanf("%d", &row1); printf("Enter the number of columns in matrix 1: "); scanf("%d", &col1); printf("Enter the number of rows in matrix 2: "); scanf("%d", &row2); printf("Enter the number of columns in matrix 2: "); scanf("%d", &col2); printf("\n"); int a[row1][col1]; int b[row2][col2]; int c[row1][col2]; int \*d, \*e, \*f; int i, j;

* In the **main** function, the user is prompted to input the dimensions of two matrices, matrix **a** (with dimensions **row1** x **col1**) and matrix **b** (with dimensions **row2** x **col2**). Arrays **a**, **b**, and **c** are declared to store the matrices, and pointers **d**, **e**, and **f** are declared to allocate memory on the GPU.

cppCopy code

printf("\nEnter elements of first matrix:\n"); for(i = 0; i < row1; i++) { for(j = 0; j < col1; j++) { scanf("%d", &a[i][j]); } } printf("\nEnter elements of second matrix:\n"); for(i = 0; i < row2; i++) { for(j = 0; j < col2; j++) { scanf("%d", &b[i][j]); } }

* Here, the user inputs the elements of matrices **a** and **b**.

cppCopy code

cudaMalloc((void \*\*)&d, row1 \* col1 \* sizeof(int)); cudaMalloc((void \*\*)&e, row2 \* col2 \* sizeof(int)); cudaMalloc((void \*\*)&f, row1 \* col2 \* sizeof(int)); cudaMemcpy(d, a, row1 \* col1 \* sizeof(int), cudaMemcpyHostToDevice); cudaMemcpy(e, b, row2 \* col2 \* sizeof(int), cudaMemcpyHostToDevice); dim3 grid(col2, row1); matproduct<<<grid, 1>>>(d, e, f, row1, col1, row2, col2); cudaMemcpy(c, f, row1 \* col2 \* sizeof(int), cudaMemcpyDeviceToHost);

* Memory is allocated on the GPU for matrices **d**, **e**, and **f**, and the data from matrices **a** and **b** is copied from the host to the device. The grid dimensions are set to match the dimensions of the output matrix **c**. The **matproduct** kernel is then launched with the specified grid and block dimensions. After the kernel execution, the result matrix **c** is copied back from the device to the host.

cppCopy code

printf("\nProduct of two matrices:\n"); for(i = 0; i < row1; i++) { for(j = 0; j < col2; j++) { printf("%d\t", c[i][j]); } printf("\n"); } cudaFree(d); cudaFree(e); cudaFree(f); return 0; }

* Finally, the product of the two matrices (**c**) is printed, and memory allocated on the GPU is freed before returning from the **main** function.

HPC -3 (openmp code explanation)

Certainly! Let's delve into the code in detail, line by line:

1. **Header Includes:**

cppCopy code

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

These lines include necessary header files.

* + **<iostream>** is included for basic input/output operations.
  + **<vector>** is included to use the vector data structure for storing elements dynamically.
  + **<omp.h>** is included for OpenMP (Open Multi-Processing) support, which allows for parallel programming.

1. **Namespace Declaration:**

cppCopy code

using namespace std;

This line simplifies the usage of symbols from the **std** namespace, which contains standard C++ library components.

1. **Function Definitions for Parallel Operations:**

cppCopy code

int parallelMin(vector<int> arr) { ... } int parallelMax(vector<int> arr) { ... } int parallelSum(vector<int> arr) { ... } double parallelAvg(vector<int> arr) { ... }

These functions are defined to perform various parallel operations on a vector of integers. Each function takes a vector **arr** as input and returns the respective result.

1. **Explanation of Parallel Operations Functions:**
   * **parallelMin Function:**
     + Finds the minimum value in the input vector **arr**.
     + Initializes **min\_val** with the first element of **arr**.
     + Utilizes OpenMP directives for parallel execution (**#pragma omp parallel for**) and a reduction operation (**reduction(min:min\_val)**) to find the minimum value across multiple threads.
   * **parallelMax Function:**
     + Similar to **parallelMin**, but it finds the maximum value in the input vector **arr**.
     + Initializes **max\_val** with the first element of **arr**.
     + Uses a reduction operation **reduction(max:max\_val)** to find the maximum value across multiple threads.
   * **parallelSum Function:**
     + Calculates the sum of all elements in the input vector **arr**.
     + Initializes **sum** to zero.
     + Employs a reduction operation **reduction(+:sum)** to calculate the sum across multiple threads.
   * **parallelAvg Function:**
     + Computes the average of all elements in the input vector **arr**.
     + Utilizes the **parallelSum** function to obtain the sum of elements and divides it by the size of the vector to calculate the average.
2. **Main Function:**

cppCopy code

int main() { ... }

* + This is the main entry point of the program.
  + It begins the execution of the program.

1. **User Input of Vector Elements:**

cppCopy code

int n; cout << "No of elements: "; cin >> n; vector<int> arr(n); cout << "Enter elements: "; for(int i = 0; i < n; i++) cin >> arr[i];

* + It prompts the user to input the number of elements (**n**) to be entered.
  + Creates a vector **arr** of size **n**.
  + Prompts the user to enter **n** elements, which are stored in the vector.

1. **Displaying Results:**

cppCopy code

cout << "Minimum: " << parallelMin(arr) << endl; cout << "Maximum: " << parallelMax(arr) << endl; cout << "Sum: " << parallelSum(arr) << endl; cout << "Average: " << parallelAvg(arr) << endl;

* + Finally, it displays the minimum, maximum, sum, and average of the elements in the vector **arr** by calling the respective functions defined earlier.

HPC-2 (openmp):-

This code demonstrates parallel implementations of two sorting algorithms: Bubble Sort and Merge Sort, using OpenMP for parallelism. Let's break it down step by step:

1. **Header Includes:**

cppCopy code

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

These lines include necessary header files:

* + **<algorithm>**: For the **swap** function used in Bubble Sort.
  + **<iostream>**: For input/output operations.
  + **<omp.h>**: For OpenMP support.
  + **<vector>**: For using the vector data structure to store elements.

1. **Namespace Declaration:**

cppCopy code

using namespace std;

This line simplifies the usage of symbols from the **std** namespace.

1. **Function Definitions:**
   * **parallelBubbleSort**: Performs the Bubble Sort algorithm in parallel.
   * **merge**: A helper function for Merge Sort to merge two sorted subarrays.
   * **parallelMergeSort**: Performs the Merge Sort algorithm in parallel.
2. **Parallel Bubble Sort:**
   * It repeatedly steps through the list, compares adjacent elements, and swaps them if they are in the wrong order.
   * The process is performed in parallel using OpenMP directives (**#pragma omp parallel for**) to distribute the work among threads.
3. **Parallel Merge Sort:**
   * It recursively divides the array into two halves until each subarray contains only one element.
   * Then, it merges the sorted subarrays to produce the final sorted array.
   * Parallelism is achieved using OpenMP sections (**#pragma omp parallel sections**) to divide the work of sorting and merging between threads.
4. **Main Function:**
   * Reads the number of elements (**n**) and the elements themselves from the user.
   * Makes a copy of the data for parallel sorting.
   * Performs parallel Bubble Sort and Merge Sort on the data copies.
   * Prints the sorted lists and the time taken for each sorting algorithm to complete.
5. **Output:**
   * Displays the sorted lists after each sorting algorithm's execution.
   * Shows the time taken for each sorting algorithm to complete its execution in seconds.

Overall, this code demonstrates how to implement and parallelize sorting algorithms like Bubble Sort and Merge Sort using OpenMP, which can improve performance on multi-core processors by utilizing parallelism effectively.