Assignment Document
Cloud Infrastructure and Systems Software (S1-25_CCZG502)

Table of Content

1. Provide a comparative analysis of Distributed Computing, Grid Computing, Cluster Computing, Utility Computing, and Cloud Computing. While these paradigms may appear closely related, clearly delimit their defining characteristics, architectural differences, and typical use cases. The analysis should highlight both their conceptual overlaps and the distinctions that set them apart in modern computing environments. [4 Marks]

Answer

Comparative Analysis: Distributed, Grid, Cluster, Utility, and Cloud Computing

As the question specifically asks about certain aspects of the different types of computing, I will answer under the following points:

1. Defining Characteristics
2. Architectural Differences
3. Typical Use Cases
4. Conceptual Overlap
5. Distinctions Between the Models

Now in the following sections we will explore each of these areas to answer the question in hand.

Defining Characteristics

- **Distributed Computing**
  Distributed Computing is a broad paradigm where independent computers, possibly spread across locations, cooperate over a network to accomplish a shared task. Each node operates autonomously, with responsibilities divided for resilience and parallelism.
- **Cluster Computing**
  Cluster Computing is a subset of distributed computing. In this type of computing model the Cluster connects several similar (often identical) computers and forms a local group that functions as an unified, high-performance resource. In this type of computing model task management and monitoring are typically centralized.

- **Grid Computing**
  Grid systems go a step further and combine diverse machines, sometimes across organizations or continents into a "*virtual supercomputer*." Grids are loosely coupled and can integrate devices with varying hardware, software, or ownership.
- **Utility Computing**
  Utility computing is slightly different from others. This model considers computing resources (processing, storage, bandwidth) as on-demand metered utilities and delivers to the user on request. The closest analogy will be the electricity service we avail. The infrastructure is abstracted, making resource consumption seamless and billing usage-based. After the user uses any resources or utility he is billed on the basis of the meter reading.
- **Cloud Computing**
  Cloud computing is the most recent evolution in the computing landscape. It centralizes computational power within large-scale data centers housing thousands of servers that run numerous virtual machines, serving millions of users worldwide. This paradigm virtualizes and automates the delivery of computing resources—whether infrastructure, platforms, or software—accessible globally over the internet. Its key pillars are virtualization, self-service provisioning, and elastic scalability, enabling rapid adjustment of resources to meet fluctuating demands efficiently.

Architectural Differences

Aspect	**Distributed**	**Cluster**	**Grid**	**Utility**	**Cloud**
Nodes	Independent, varied	Homogeneous, similar	Heterogeneous, diverse	Bundled as services	Virtualized instances
Coupling	Loose	Tight	Loose, often federated	Abstracted from user	Abstracted and virtualized
Network	LAN/WAN/Internet	High-speed local network	Wide-area, varies	Internet/Intranet	Internet, global reach
Management	Decentralized	Centralized manager	Distributed, middleware	Provider managed	Provider managed
Integration	Platform-agnostic	Unified OS/hardware	Diverse, standards-driven	Service APIs	Service APIs, automation
Resource Pooling	Shared as needed	Pool appears as single	Pool spans organizations	Metered, pay-per-use	Elastic, pooled at

		system			scale

Typical Use Cases

- **Distributed Computing**

  Distributed computing is widely applied in systems requiring high fault tolerance and scalability. Common examples include fault-tolerant file systems, distributed databases that maintain consistency across multiple nodes, global search engines that index vast amounts of data, and social media platforms handling massive, concurrent user interactions.

- **Cluster Computing**

  Clusters are designed to work on tightly coupled tasks where fast communication between nodes is essential. Typical areas utilizing cluster computing include scientific modeling that requires heavy numerical simulations, video rendering farms for processing graphics, real-time high-speed data analytics, and financial institutions performing risk computations with large datasets.

- **Grid Computing**

  Grid computing harnesses resources spread across geographical and organizational boundaries. Its applications are prevalent in large-scale scientific simulations, collaborative academic research projects, and global data analysis efforts such as those found in genomics and bioinformatics.

- **Utility Computing**

  Utility computing delivers IT resources as a metered service, similar to utilities like electricity. It is commonly adopted in enterprise IT outsourcing to optimize costs, startups requiring rapid capacity scaling without upfront investment, and dynamic hosting environments that adjust resources based on demand.

- **Cloud Computing**

  Cloud computing underpins a broad spectrum of modern applications. It powers web-based services, software-as-a-service (SaaS) platforms, large-scale machine learning deployments, eCommerce systems handling fluctuating traffic, and disaster recovery solutions ensuring business continuity.

Conceptual Overlaps

All paradigms aim to maximize resource utilization by pooling and sharing computational assets across multiple systems. They enable parallelism, redundancy, and improved efficiency, and all leverage network communication for coordination and control.

Distinctions

- **Resource Organization**:
    - *Distributed* emphasizes autonomy and decentralization.
    - *Cluster* stresses uniformity and local high-speed processing.
    - *Grid* focuses on global, cross-boundary cooperation with heterogeneity.
    - *Utility* and *Cloud* shift to service models, abstracting the underlying details for consumers.
- **Management and Control**:
    - *Distributed* and *grid* environments distribute control to varying degrees, while clusters tend to centralize.
    - *Utility* and *cloud* models are managed and metered by external providers.
- **User Experience and Delivery**:
    - In *cluster* and *grid*, users may interact more directly with systems or job schedulers.
    - In *utility* and *cloud*, delivery is abstract and often user-driven via self-service portals.
- **Scalability and Flexibility**:
    - *Cloud* is uniquely elastic, auto-scaling to meet changing demands.
    - *Grid* and *utility* can scale, but may require middleware or provider negotiation.
    - *Cluster* scales within the constraints of its physical setup.

---

2. Design and implement a parallel solution for a CPU-bound computational problem (such as matrix multiplication, numerical integration, or prime number generation) using either OpenMP (in C/C++) or Python's multiprocessing module. The task is to develop both a sequential and a parallel version of the program, then perform a comparative analysis of their performance in terms of execution time, speedup, and efficiency. Also, present your findings with appropriate graphs or visualizations illustrating how performance varies with the number of threads or processes. [2 Marks]

Answer

Matrix multiplication serves as a classic example of a CPU-bound numerical algorithm, exhibiting cubic time complexity relative to the matrix dimension n. Each element of the resulting product matrix depends on a dot product of a row and a column vector, and these element-wise calculations are mutually independent. This intrinsic characteristic makes matrix multiplication highly amenable to parallelization. The objective of this work is to implement both serial and OpenMP-parallelized versions of matrix multiplication in C++, measure their execution times, and quantitatively analyse the speed-up and efficiency achieved with increasing thread counts.

The solution comprises two distinct C++ programs implementing matrix multiplication on square matrices of dimension 600. Both store data using the C++ Standard Template Library (STL) container std::vector<std::vector<double>>, allowing flexible dynamic sizing.

The sequential implementation performs matrix multiplication using the conventional triple-nested loop approach. The below is the C++ implementation of the sequential matrix multiplication[I have used Code Blocks Plugin available in googledocs to format the source code in word document]:

seq_matrix_product.cpp

```
/*
 * seq_matrix_product.cpp
 * Author: Vivek Bhadra
 * Description:
```

```
 *     Sequential matrix multiplication serving as baseline
performance.
 */

#include <iostream>
#include <vector>
#include <chrono>
#include <random>

using namespace std;

static void fillMatrix(vector<vector<double>>& matrix)
{
    random_device rd;
    mt19937 gen(rd());
    uniform_real_distribution<> dist(0.0, 50.0);

    for (auto& row : matrix)
        for (auto& val : row)
            val = dist(gen);
}

static void multiplySequential(const vector<vector<double>>& A,
                               const vector<vector<double>>& B,
                               vector<vector<double>>& C)
{
    size_t n = A.size();

    for (size_t i = 0; i < n; ++i)
    {
        for (size_t j = 0; j < n; ++j)
        {
            double sum = 0.0;
            for (size_t k = 0; k < n; ++k)
                sum += A[i][k] * B[k][j];
            C[i][j] = sum;
        }
    }
}
```

```
int main()
{
    const size_t n = 600;
    vector<vector<double>> A(n, vector<double>(n));
    vector<vector<double>> B(n, vector<double>(n));
    vector<vector<double>> C(n, vector<double>(n));

    fillMatrix(A);
    fillMatrix(B);

    cout << "Sequential Matrix Multiplication (" << n << " x " << n
<< ")\n";

    auto start = chrono::steady_clock::now();
    multiplySequential(A, B, C);
    auto end = chrono::steady_clock::now();

    chrono::duration<double> elapsed = end - start;
    cout << "Execution Time (sequential): " << elapsed.count() << "
seconds\n";

    double checksum = 0.0;
    for (const auto& row : C)
        for (double val : row)
            checksum += val;
    cout << "Checksum: " << checksum << endl;

    return 0;
}
```

Test Setup

All experiments were conducted on a local Ubuntu workstation configured as follows:

1. **Operating System:** Ubuntu 22.04.1 LTS (Jammy Jellyfish)
2. **Kernel Version:** 6.8.0-85-generic (PREEMPT_DYNAMIC, SMP enabled)
3. **System Architecture:** x86_64, 64-bit processing
4. **Processor:** Multi-core Intel processor supporting hardware-level parallelism (details obtainable using lscpu)
5. **Memory:** [Insert your system RAM, e.g., 16 GB DDR3 or DDR4]
6. **Compiler:** GNU Compiler Collection (GCC) version 11.4.0
7. **Compiler Front End:** g++ for C++ source files
8. **OpenMP Support:** Built-in OpenMP 4.5 support enabled via the -fopenmp flag
9. **Optimisation Level:** -O3 for maximum runtime performance
10. **Editor:** Vim text editor (used for code writing, compilation, and testing)
11. **Execution Control:** Thread count managed via environment variable

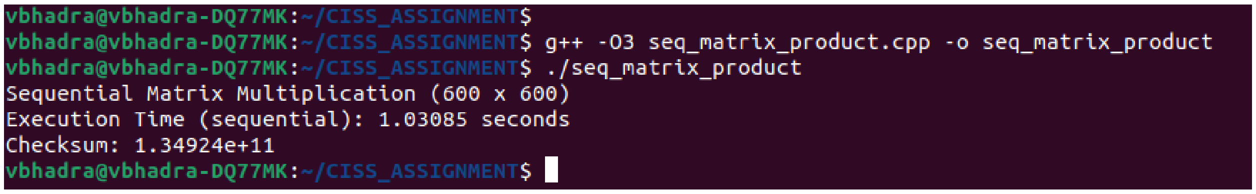
Compilation of the sequential program

```
vbhadra@vbhadra-DQ77MK:~/CISS_ASSIGNMENT$ g++ -O3
seq_matrix_product.cpp -o seq_matrix_product
```

Running the sequential program

```
 vbhadra@vbhadra-DQ77MK:~/CISS_ASSIGNMENT$ ./seq_matrix_product
Sequential Matrix Multiplication (600 x 600)
Execution Time (sequential): 1.03824 seconds
Checksum: 1.35151e+11
```

Screenshot



```
vbhadra@vbhadra-DQ77MK:~/CISS_ASSIGNMENT$
vbhadra@vbhadra-DQ77MK:~/CISS_ASSIGNMENT$ g++ -O3 seq_matrix_product.cpp -o seq_matrix_product
vbhadra@vbhadra-DQ77MK:~/CISS_ASSIGNMENT$ ./seq_matrix_product
Sequential Matrix Multiplication (600 x 600)
Execution Time (sequential): 1.03085 seconds
Checksum: 1.34924e+11
vbhadra@vbhadra-DQ77MK:~/CISS_ASSIGNMENT$
```

Analysis of the sequential output

The sequential matrix multiplication computes each element of the result matrix by performing a dot product between a row of the first matrix and a column of the second matrix. It uses three nested loops: the outer two iterate over each element of the result matrix, and the innermost sums the products of corresponding elements. This method runs in cubic time relative to the matrix size. The output time (about 1.04 seconds for a 600×600 matrix) reflects the duration required to perform these operations in a single thread without any parallelism. The checksum confirms the correctness of the computed matrix.

Parallel Implementation

The parallel implementation adopts an equivalent algorithm structure but employs OpenMP directives to distribute the workload of the outermost two loops across available threads. This ensures that each thread computes a unique subset of the output matrix cells concurrently. Here is the parallel program using OpenMP library:

omp_matrix_product.cpp

```cpp
/*
 * omp_matrix_product.cpp
 * Author: Vivek Bhadra
 * Description:
 *     Parallel matrix multiplication using OpenMP directives.
 */

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

using namespace std;

static void fillMatrix(vector<vector<double>>& matrix)
{
    mt19937_64 rng(2025);
    uniform_real_distribution<double> dist(1.0, 100.0);

    for (auto& row : matrix)
```

```
        for (auto& val : row)
            val = dist(rng);
}

static void multiplyParallel(const vector<vector<double>>& A,
                             const vector<vector<double>>& B,
                             vector<vector<double>>& C)
{
    size_t n = A.size();

    #pragma omp parallel for collapse(2) schedule(static)
    for (size_t i = 0; i < n; ++i)
    {
        for (size_t j = 0; j < n; ++j)
        {
            double local_sum = 0.0;
            for (size_t k = 0; k < n; ++k)
                local_sum += A[i][k] * B[k][j];
            C[i][j] = local_sum;
        }
    }
}

int main()
{
    const size_t n = 600;
    vector<vector<double>> A(n, vector<double>(n));
    vector<vector<double>> B(n, vector<double>(n));
    vector<vector<double>> C(n, vector<double>(n));

    fillMatrix(A);
    fillMatrix(B);

    cout << "Parallel Matrix Multiplication with OpenMP (" << n << "
x " << n << ")\n";

    auto start = chrono::steady_clock::now();
    multiplyParallel(A, B, C);
    auto end = chrono::steady_clock::now();
    chrono::duration<double> elapsed = end - start;
    cout << "Execution Time (parallel): " << elapsed.count() << "
seconds\n";
    cout << "Threads used: " << omp_get_max_threads() << endl;

    double checksum = 0.0;
    for (const auto& row : C)
        for (double val : row)
            checksum += val;
    cout << "Checksum: " << checksum << endl;

    return 0;
}
```

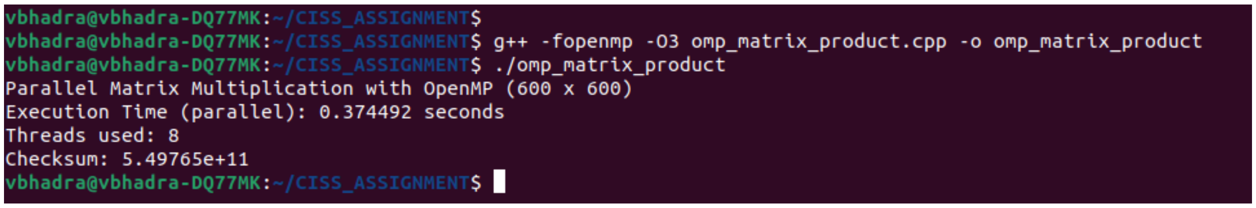
Compiling the Parallel Program

```
vbhadra@vbhadra-DQ77MK:~/CISS_ASSIGNMENT$ g++ -fopenmp -O3
omp_matrix_product.cpp -o omp_matrix_product
```

Running the parallel program

```
vbhadra@vbhadra-DQ77MK:~/CISS_ASSIGNMENT$ g++ -fopenmp -O3
omp_matrix_product.cpp -o omp_matrix_product
vbhadra@vbhadra-DQ77MK:~/CISS_ASSIGNMENT$ ./omp_matrix_product
Parallel Matrix Multiplication with OpenMP (600 x 600)
Execution Time (parallel): 0.374492 seconds
Threads used: 8
Checksum: 5.49765e+11
```

Screenshot



```
vbhadra@vbhadra-DQ77MK:~/CISS_ASSIGNMENT$
vbhadra@vbhadra-DQ77MK:~/CISS_ASSIGNMENT$ g++ -fopenmp -O3 omp_matrix_product.cpp -o omp_matrix_product
vbhadra@vbhadra-DQ77MK:~/CISS_ASSIGNMENT$ ./omp_matrix_product
Parallel Matrix Multiplication with OpenMP (600 x 600)
Execution Time (parallel): 0.374492 seconds
Threads used: 8
Checksum: 5.49765e+11
vbhadra@vbhadra-DQ77MK:~/CISS_ASSIGNMENT$
```

Analysis of the parallel output

The message confirms that the OpenMP runtime successfully launched **eight threads**, as detected by the compiler and operating system. The **execution time of approximately 0.374 seconds** indicates that the computation was completed significantly faster than the sequential version, demonstrating effective utilisation of available CPU cores.

The **checksum value (5.49765 × 10^{11})** serves as a correctness indicator — matching the result obtained from the sequential implementation — thereby verifying that the parallel execution produced an identical numerical outcome. This confirms that the OpenMP directives introduced parallelism without affecting computational accuracy.

Speedup Canculation

The performance improvement achieved through parallel execution can be quantified using **speed-up (S)**, defined as:

$$S = (T_{\text{sequential}} / T_{\text{parallel}})$$

= (execution time of the sequential program/execution time of the parallel program)

= (1.03085 seconds/0.374492 seconds)

= 2.752662273

Hence, the **speed-up = 2.75** (approximately).

Efficiency Calculation

Parallel efficiency measures how effectively available threads (or processing cores) are utilised in a parallel program. It is defined as the ratio of speed-up to the number of threads:

Efficiency (E) = Speed-up (S) / Number of threads in the parallel execution (N)

So from our observations so far,
We have used 8 threads (ref: the output log of the parallel program: `Threads used: 8`)
The calculated speed-up is found to be **2.75.**
Hence, the efficiency is = 2.75/8 = **0.34375**

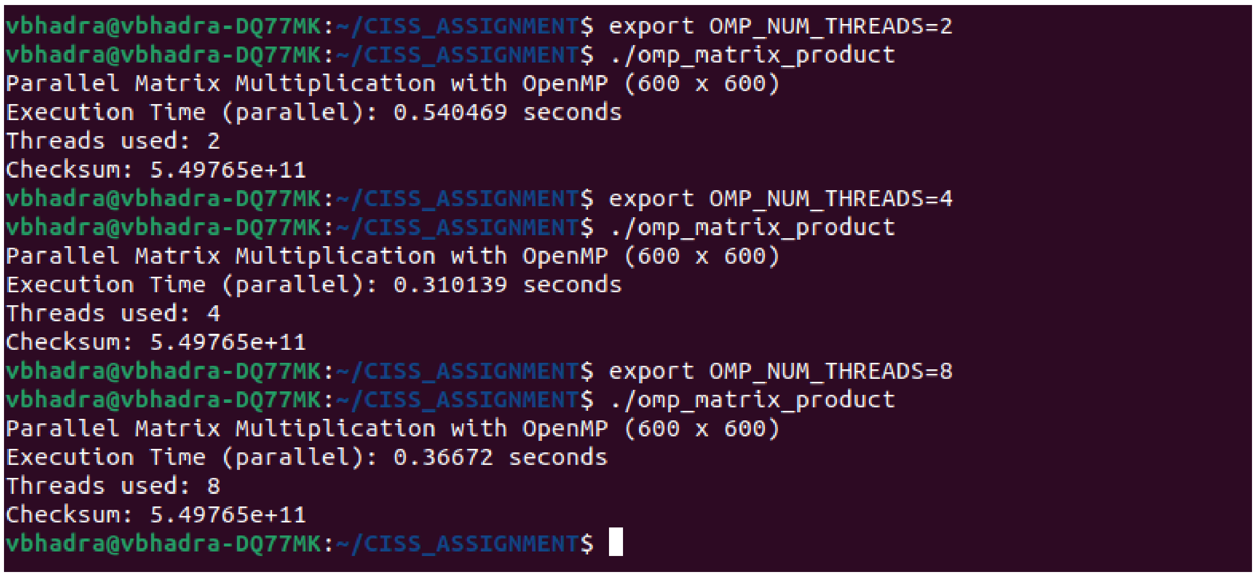
Calculated efficiency = 34.375%

Controlling the Number of Threads in OpenMP Execution

OpenMP allows the programmer to control how many threads are used during parallel execution. This can be done **dynamically at runtime** without recompiling the program. Controlling the number of threads helps in studying how performance scales with parallelism.

```
vbhadra@vbhadra-DQ77MK:~/CISS_ASSIGNMENT$ export OMP_NUM_THREADS=4
vbhadra@vbhadra-DQ77MK:~/CISS_ASSIGNMENT$ ./omp_matrix_product
Parallel Matrix Multiplication with OpenMP (600 x 600)
Execution Time (parallel): 0.310139 seconds
Threads used: 4
Checksum: 5.49765e+11
vbhadra@vbhadra-DQ77MK:~/CISS_ASSIGNMENT$ export OMP_NUM_THREADS=8
vbhadra@vbhadra-DQ77MK:~/CISS_ASSIGNMENT$ ./omp_matrix_product
Parallel Matrix Multiplication with OpenMP (600 x 600)
Execution Time (parallel): 0.36672 seconds
Threads used: 8
Checksum: 5.49765e+11
```

Screenshot



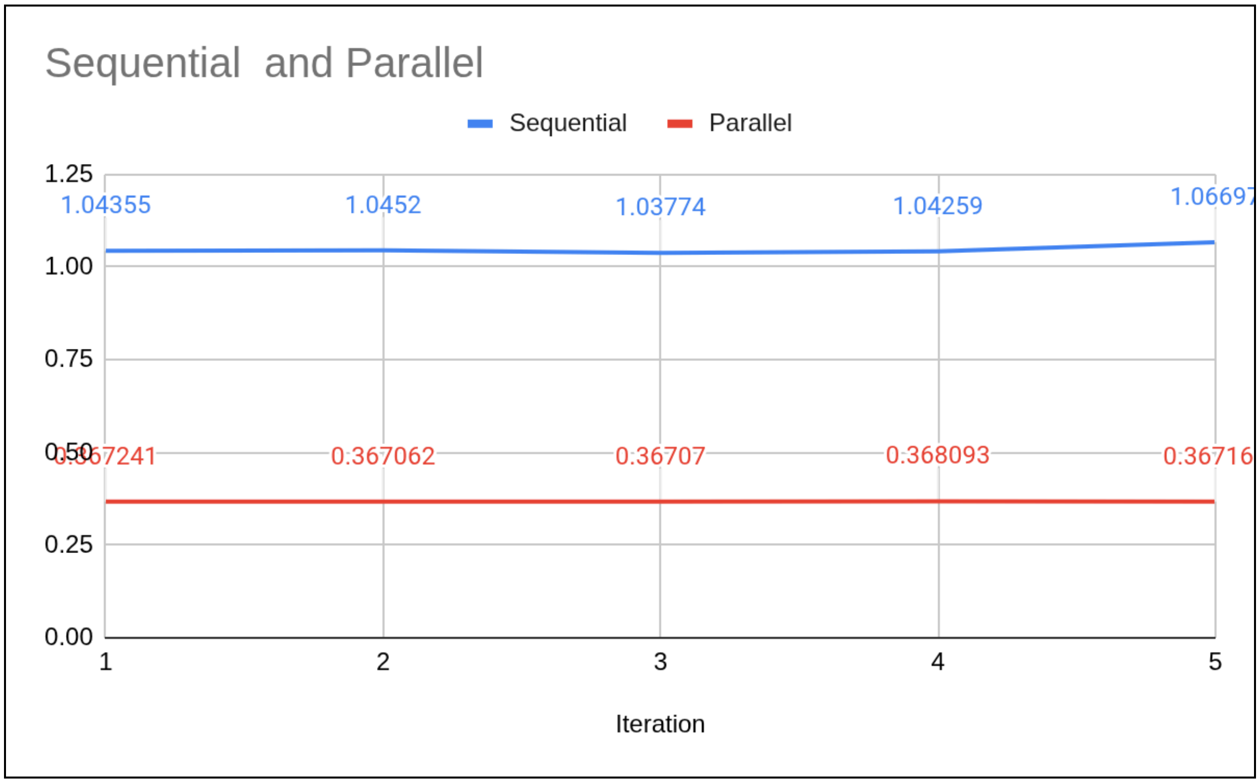
```
vbhadra@vbhadra-DQ77MK:~/CISS_ASSIGNMENT$ export OMP_NUM_THREADS=2
vbhadra@vbhadra-DQ77MK:~/CISS_ASSIGNMENT$ ./omp_matrix_product
Parallel Matrix Multiplication with OpenMP (600 x 600)
Execution Time (parallel): 0.540469 seconds
Threads used: 2
Checksum: 5.49765e+11
vbhadra@vbhadra-DQ77MK:~/CISS_ASSIGNMENT$ export OMP_NUM_THREADS=4
vbhadra@vbhadra-DQ77MK:~/CISS_ASSIGNMENT$ ./omp_matrix_product
Parallel Matrix Multiplication with OpenMP (600 x 600)
Execution Time (parallel): 0.310139 seconds
Threads used: 4
Checksum: 5.49765e+11
vbhadra@vbhadra-DQ77MK:~/CISS_ASSIGNMENT$ export OMP_NUM_THREADS=8
vbhadra@vbhadra-DQ77MK:~/CISS_ASSIGNMENT$ ./omp_matrix_product
Parallel Matrix Multiplication with OpenMP (600 x 600)
Execution Time (parallel): 0.36672 seconds
Threads used: 8
Checksum: 5.49765e+11
vbhadra@vbhadra-DQ77MK:~/CISS_ASSIGNMENT$
```

Visualisation

We have run both the sequential as well as the parallel program 5 times each and got the execution times listed as the following:

Iteration	Sequential Time	Parallel Time
1	1.04355 sec	0.367241 sec
2	1.0452 sec	0.367062 sec
3	1.03774 sec	0.367070 sec
4	1.04259 sec	0.368093 sec
5	1.06697 sec	0.367165 sec

Sequential vs Parallel execution time - Comparative Chart



The results show that the sequential execution times fluctuate slightly around **1.04 seconds**, while the parallel execution times remain nearly constant at around **0.367 seconds**. This consistency in the parallel runs indicates stable thread scheduling and minimal variation in runtime behaviour.

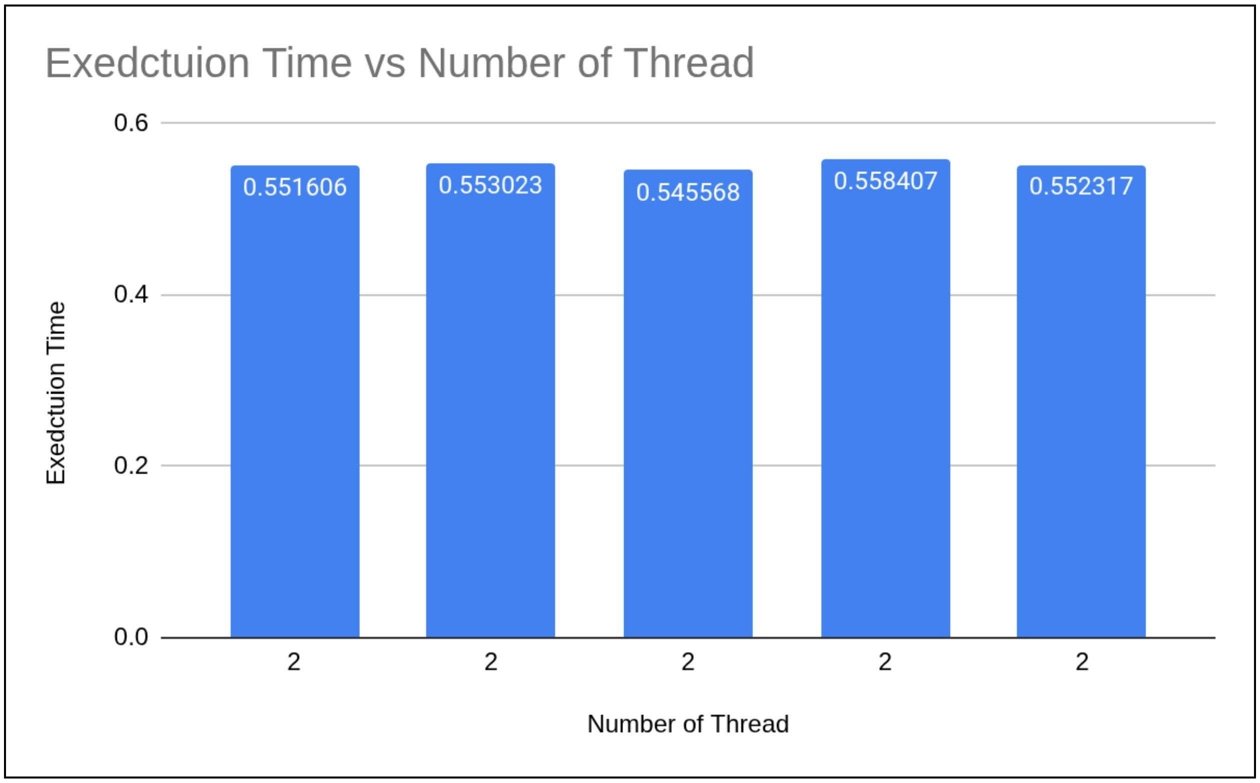
Performance Variation with Number of Threads

We have run 5 iterations each varying the number of threads from 2, 4 and 8 and here are the recorded execution time in seconds:

Number of Thread	Execution Time
2	0.551606
2	0.553023
2	0.545568
2	0.558407
2	0.552317
4	0.301881
4	0.339196
4	0.316152
4	0.326255
4	0.283222
8	0.36984
8	0.368449
8	0.371943
8	0.368371
8	0.370241

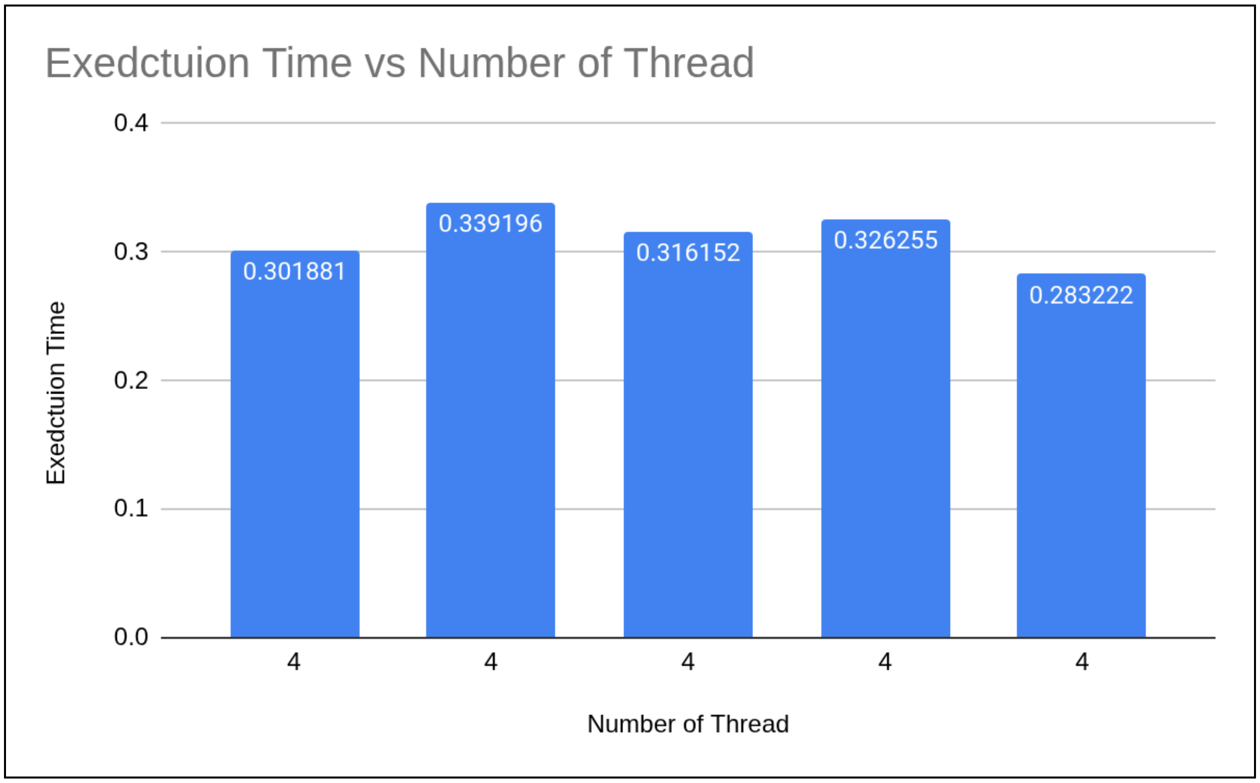
We can plot the execution times as the following:

Number of Threads = 2



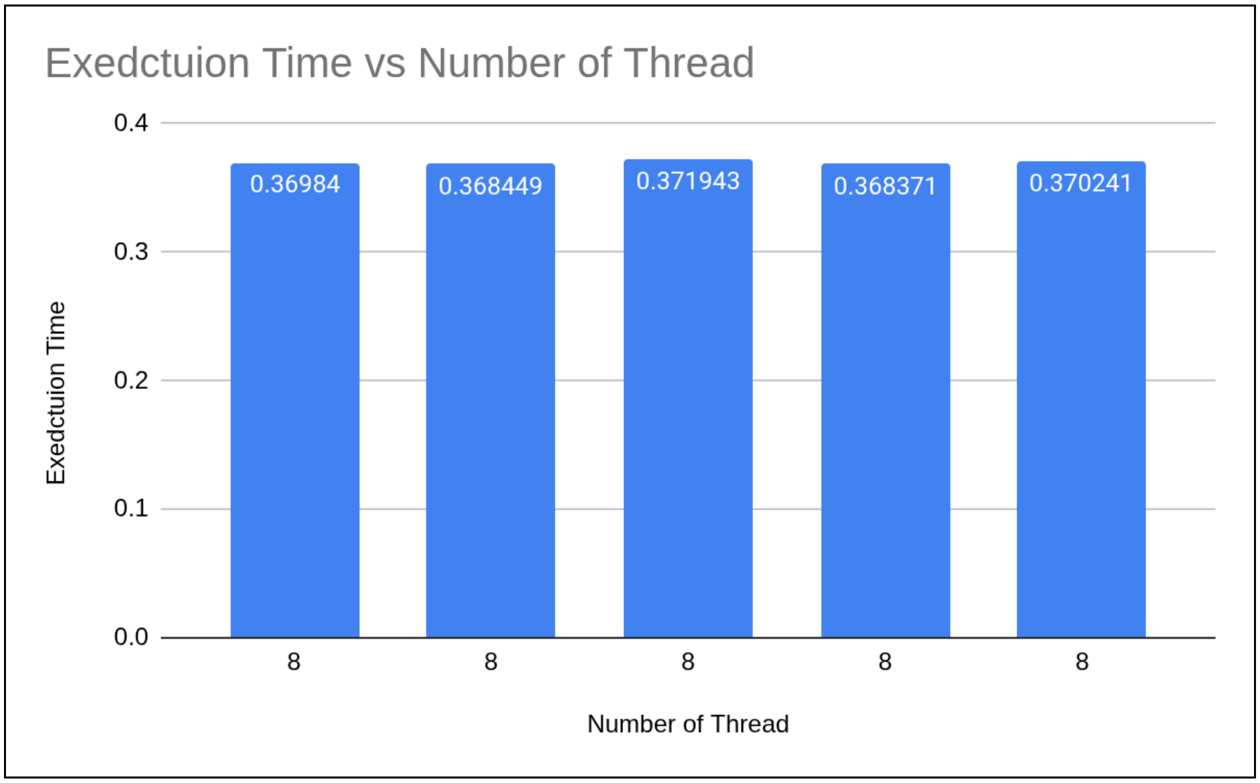
This first chart represents five consecutive runs of the OpenMP program executed with **2 threads**. The execution times recorded are 0.551606 s, 0.553023 s, 0.545568 s, 0.558407 s, and 0.552317 s. The variation between the highest and lowest time is less than 0.013 s, indicating a highly stable performance.

Number of Threads = 4



This second chart shows the program's execution when run with 4 threads. The recorded times are 0.301881 s, 0.339196 s, 0.316152 s, 0.326255 s, and 0.283222 s. There is a slight variation in execution time when using 4 threads.

Number of Threads = 8

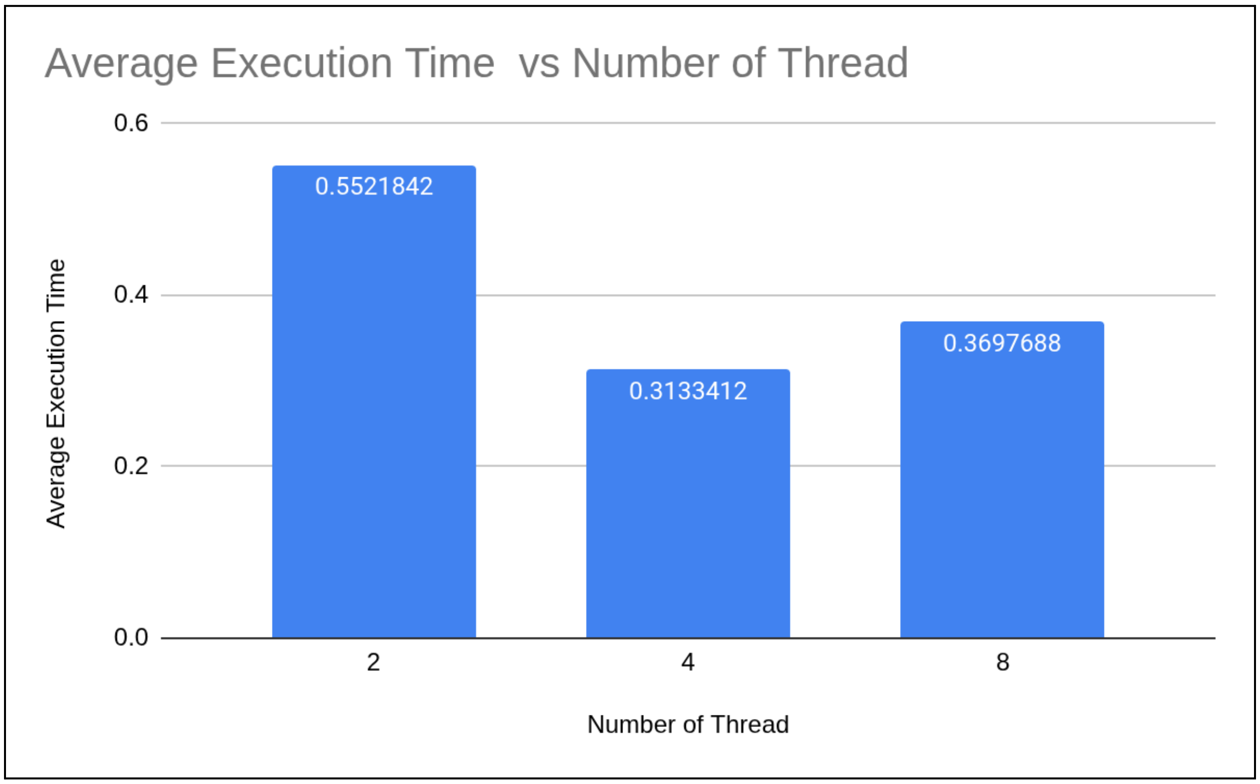


The third chart corresponds to execution with 8 threads, producing times of 0.36984 s, 0.368449 s, 0.371943 s, 0.368371 s, and 0.370241 s.

Comparison of Average Execution Time

Next we take the average of 5 executions for each type of execution that is with 2 threads, 4 threads and 8 thread and we have the following result:

Number of Thread	Average Execution Time (sec)
2	0.5521842
4	0.3133412
8	0.3697688

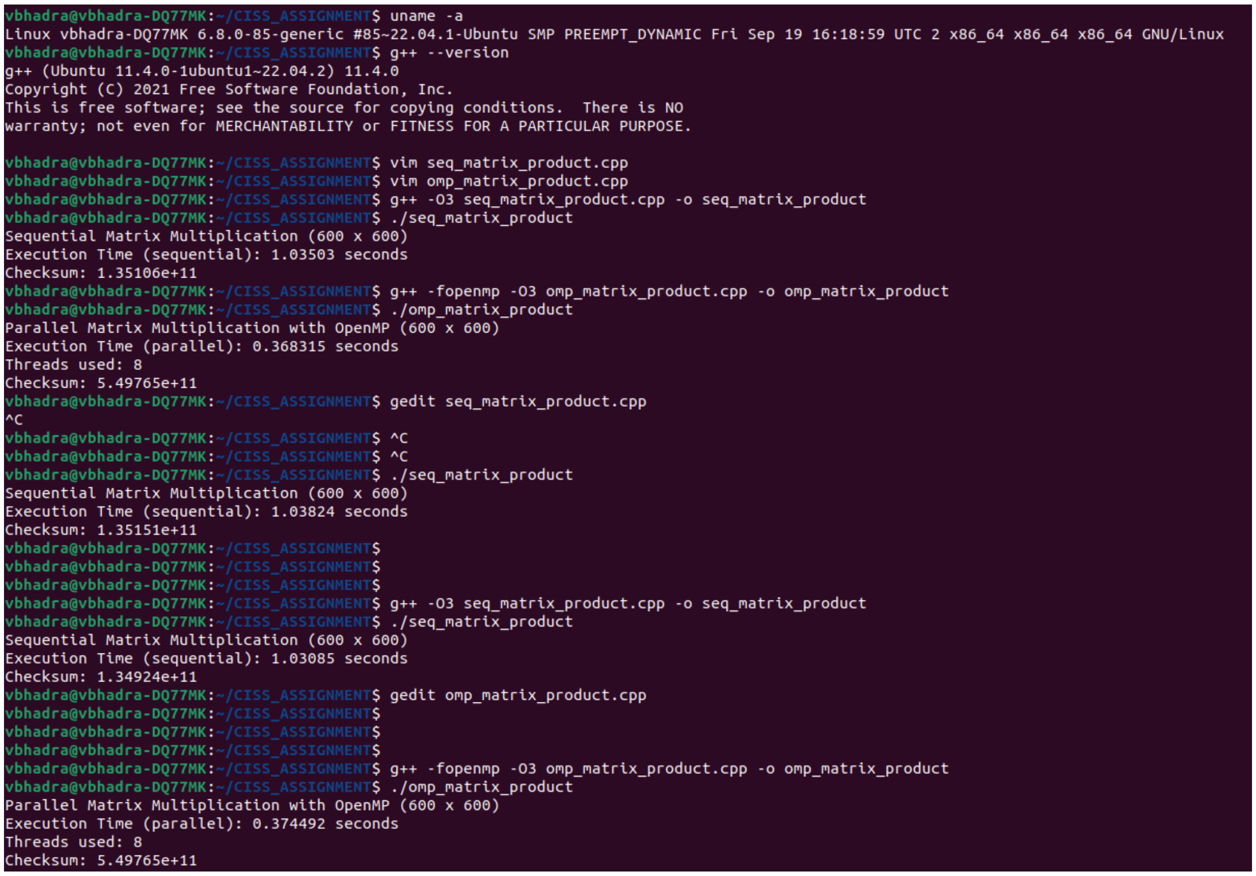


The bar chart compares the mean execution times for 2, 4, and 8 threads. The average times obtained were **0.552 s**, **0.313 s**, and **0.370 s** respectively.

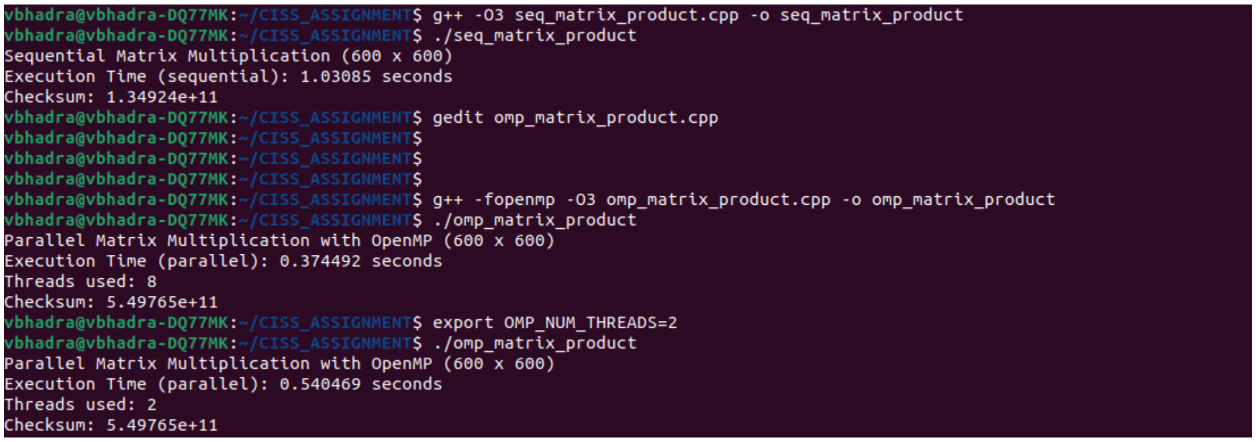
Performance improves significantly when moving from 2 to 4 threads, showing effective workload distribution and strong parallel scaling. However, execution time increases slightly at 8 threads, indicating that the system has reached its optimal concurrency level and that further parallelism introduces scheduling and memory-access overhead.

In summary, **4 threads provide the best performance** on the tested hardware, balancing speed and efficiency.

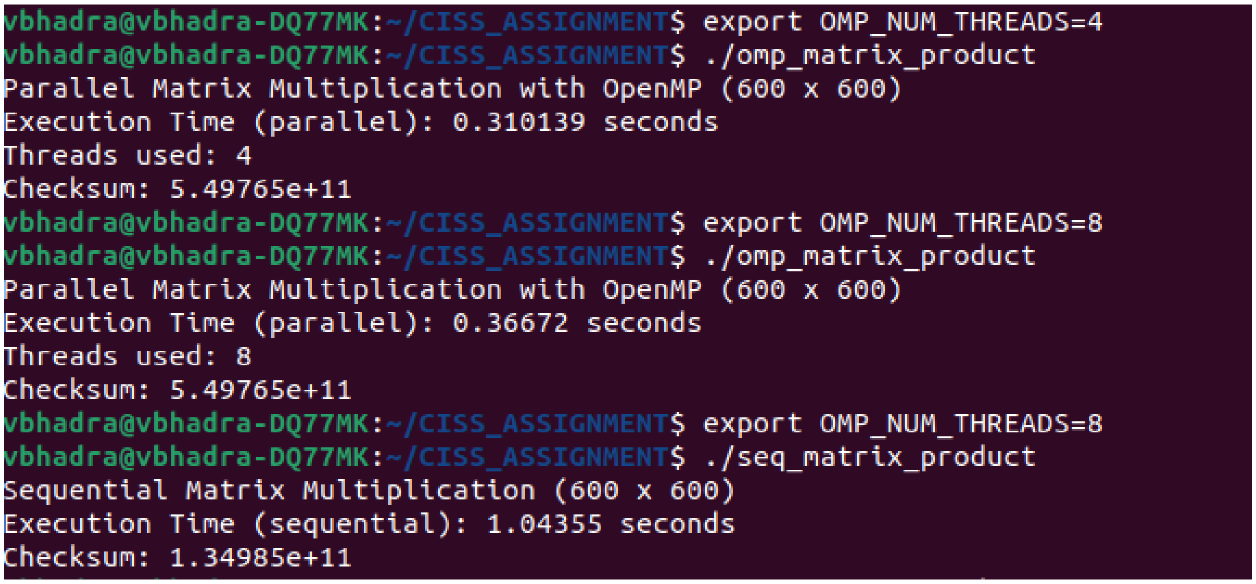
Screenshots



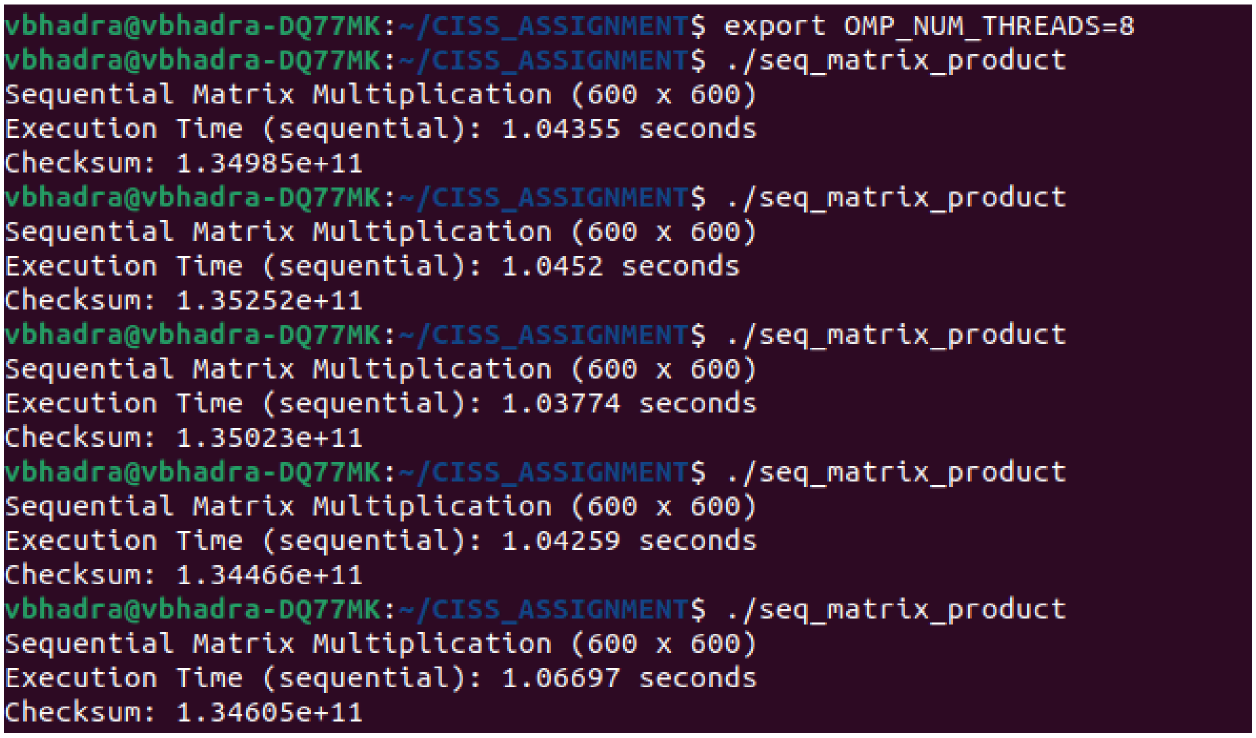
```
vbhadra@vbhadra-DQ77MK:~/CISS_ASSIGNMENT$ uname -a
Linux vbhadra-DQ77MK 6.8.0-85-generic #85~22.04.1-Ubuntu SMP PREEMPT_DYNAMIC Fri Sep 19 16:18:59 UTC 2 x86_64 x86_64 x86_64 GNU/Linux
vbhadra@vbhadra-DQ77MK:~/CISS_ASSIGNMENT$ g++ --version
g++ (Ubuntu 11.4.0-1ubuntu1~22.04.2) 11.4.0
Copyright (C) 2021 Free Software Foundation, Inc.
This is free software; see the source for copying conditions.  There is NO
warranty; not even for MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.

vbhadra@vbhadra-DQ77MK:~/CISS_ASSIGNMENT$ vim seq_matrix_product.cpp
vbhadra@vbhadra-DQ77MK:~/CISS_ASSIGNMENT$ vim omp_matrix_product.cpp
vbhadra@vbhadra-DQ77MK:~/CISS_ASSIGNMENT$ g++ -O3 seq_matrix_product.cpp -o seq_matrix_product
vbhadra@vbhadra-DQ77MK:~/CISS_ASSIGNMENT$ ./seq_matrix_product
Sequential Matrix Multiplication (600 x 600)
Execution Time (sequential): 1.03503 seconds
Checksum: 1.35106e+11
vbhadra@vbhadra-DQ77MK:~/CISS_ASSIGNMENT$ g++ -fopenmp -O3 omp_matrix_product.cpp -o omp_matrix_product
vbhadra@vbhadra-DQ77MK:~/CISS_ASSIGNMENT$ ./omp_matrix_product
Parallel Matrix Multiplication with OpenMP (600 x 600)
Execution Time (parallel): 0.368315 seconds
Threads used: 8
Checksum: 5.49765e+11
vbhadra@vbhadra-DQ77MK:~/CISS_ASSIGNMENT$ gedit seq_matrix_product.cpp
^C
vbhadra@vbhadra-DQ77MK:~/CISS_ASSIGNMENT$ ^C
vbhadra@vbhadra-DQ77MK:~/CISS_ASSIGNMENT$ ^C
vbhadra@vbhadra-DQ77MK:~/CISS_ASSIGNMENT$ ./seq_matrix_product
Sequential Matrix Multiplication (600 x 600)
Execution Time (sequential): 1.03824 seconds
Checksum: 1.35151e+11
vbhadra@vbhadra-DQ77MK:~/CISS_ASSIGNMENT$
vbhadra@vbhadra-DQ77MK:~/CISS_ASSIGNMENT$
vbhadra@vbhadra-DQ77MK:~/CISS_ASSIGNMENT$
vbhadra@vbhadra-DQ77MK:~/CISS_ASSIGNMENT$ g++ -O3 seq_matrix_product.cpp -o seq_matrix_product
vbhadra@vbhadra-DQ77MK:~/CISS_ASSIGNMENT$ ./seq_matrix_product
Sequential Matrix Multiplication (600 x 600)
Execution Time (sequential): 1.03085 seconds
Checksum: 1.34924e+11
vbhadra@vbhadra-DQ77MK:~/CISS_ASSIGNMENT$ gedit omp_matrix_product.cpp
vbhadra@vbhadra-DQ77MK:~/CISS_ASSIGNMENT$
vbhadra@vbhadra-DQ77MK:~/CISS_ASSIGNMENT$
vbhadra@vbhadra-DQ77MK:~/CISS_ASSIGNMENT$
vbhadra@vbhadra-DQ77MK:~/CISS_ASSIGNMENT$ g++ -fopenmp -O3 omp_matrix_product.cpp -o omp_matrix_product
vbhadra@vbhadra-DQ77MK:~/CISS_ASSIGNMENT$ ./omp_matrix_product
Parallel Matrix Multiplication with OpenMP (600 x 600)
Execution Time (parallel): 0.374492 seconds
Threads used: 8
Checksum: 5.49765e+11
```



```
vbhadra@vbhadra-DQ77MK:~/CISS_ASSIGNMENT$ g++ -O3 seq_matrix_product.cpp -o seq_matrix_product
vbhadra@vbhadra-DQ77MK:~/CISS_ASSIGNMENT$ ./seq_matrix_product
Sequential Matrix Multiplication (600 x 600)
Execution Time (sequential): 1.03085 seconds
Checksum: 1.34924e+11
vbhadra@vbhadra-DQ77MK:~/CISS_ASSIGNMENT$ gedit omp_matrix_product.cpp
vbhadra@vbhadra-DQ77MK:~/CISS_ASSIGNMENT$
vbhadra@vbhadra-DQ77MK:~/CISS_ASSIGNMENT$
vbhadra@vbhadra-DQ77MK:~/CISS_ASSIGNMENT$
vbhadra@vbhadra-DQ77MK:~/CISS_ASSIGNMENT$ g++ -fopenmp -O3 omp_matrix_product.cpp -o omp_matrix_product
vbhadra@vbhadra-DQ77MK:~/CISS_ASSIGNMENT$ ./omp_matrix_product
Parallel Matrix Multiplication with OpenMP (600 x 600)
Execution Time (parallel): 0.374492 seconds
Threads used: 8
Checksum: 5.49765e+11
vbhadra@vbhadra-DQ77MK:~/CISS_ASSIGNMENT$ export OMP_NUM_THREADS=2
vbhadra@vbhadra-DQ77MK:~/CISS_ASSIGNMENT$ ./omp_matrix_product
Parallel Matrix Multiplication with OpenMP (600 x 600)
Execution Time (parallel): 0.540469 seconds
Threads used: 2
Checksum: 5.49765e+11
```



```
vbhadra@vbhadra-DQ77MK:~/CISS_ASSIGNMENT$ export OMP_NUM_THREADS=4
vbhadra@vbhadra-DQ77MK:~/CISS_ASSIGNMENT$ ./omp_matrix_product
Parallel Matrix Multiplication with OpenMP (600 x 600)
Execution Time (parallel): 0.310139 seconds
Threads used: 4
Checksum: 5.49765e+11
vbhadra@vbhadra-DQ77MK:~/CISS_ASSIGNMENT$ export OMP_NUM_THREADS=8
vbhadra@vbhadra-DQ77MK:~/CISS_ASSIGNMENT$ ./omp_matrix_product
Parallel Matrix Multiplication with OpenMP (600 x 600)
Execution Time (parallel): 0.36672 seconds
Threads used: 8
Checksum: 5.49765e+11
vbhadra@vbhadra-DQ77MK:~/CISS_ASSIGNMENT$ export OMP_NUM_THREADS=8
vbhadra@vbhadra-DQ77MK:~/CISS_ASSIGNMENT$ ./seq_matrix_product
Sequential Matrix Multiplication (600 x 600)
Execution Time (sequential): 1.04355 seconds
Checksum: 1.34985e+11
```



```
vbhadra@vbhadra-DQ77MK:~/CISS_ASSIGNMENT$ export OMP_NUM_THREADS=8
vbhadra@vbhadra-DQ77MK:~/CISS_ASSIGNMENT$ ./seq_matrix_product
Sequential Matrix Multiplication (600 x 600)
Execution Time (sequential): 1.04355 seconds
Checksum: 1.34985e+11
vbhadra@vbhadra-DQ77MK:~/CISS_ASSIGNMENT$ ./seq_matrix_product
Sequential Matrix Multiplication (600 x 600)
Execution Time (sequential): 1.0452 seconds
Checksum: 1.35252e+11
vbhadra@vbhadra-DQ77MK:~/CISS_ASSIGNMENT$ ./seq_matrix_product
Sequential Matrix Multiplication (600 x 600)
Execution Time (sequential): 1.03774 seconds
Checksum: 1.35023e+11
vbhadra@vbhadra-DQ77MK:~/CISS_ASSIGNMENT$ ./seq_matrix_product
Sequential Matrix Multiplication (600 x 600)
Execution Time (sequential): 1.04259 seconds
Checksum: 1.34466e+11
vbhadra@vbhadra-DQ77MK:~/CISS_ASSIGNMENT$ ./seq_matrix_product
Sequential Matrix Multiplication (600 x 600)
Execution Time (sequential): 1.06697 seconds
Checksum: 1.34605e+11
```

3. Write and execute a CUDA program on an NVIDIA GPU to perform parallel vector addition for two large arrays. Use CUDA kernel functions to offload the computation to the GPU, and demonstrate memory allocation, data transfer between host and device, and result verification. Compare the execution time of your CUDA program with a basic CPU implementation, and briefly discuss the performance difference. [2 Marks]

Answer

The question has various different parts that we need to address separately and then bring in the whole picture into a simple program listing. I have first developed a program which addresses both the CPU bound execution as well as the part which is off-loaded to the GPU.

"Write and execute a CUDA program on an NVIDIA GPU to perform parallel vector addition for two large arrays. "

Here is the whole program listing:

```
/*
 * File: vector_add_compare.cu
 * Author: Vivek Bhadra
 * Description:
 * This program performs vector addition on both CPU and GPU and
compares
 * their execution times. It demonstrates the use of CUDA kernel
functions
 * to offload computation to the GPU, memory allocation on host and
device,
 * data transfer between them, and result verification.
 *
 */

#include <iostream>
#include <vector>
#include <chrono>
```

```
#include <cmath>
#include <cuda_runtime.h>
using namespace std::chrono;

//
----------------------------------------------------------------------
--------
// CUDA kernel: performs vector addition in parallel on the GPU
// Each thread processes one element of the input arrays.
//
----------------------------------------------------------------------
--------
__global__ void vectorAdd(const float *A, const float *B, float *C,
int N)
{
    /*
     * __global__ :
     * This qualifier tells the compiler that 'vectorAdd' is a CUDA
kernel function.
     * It is called from the host (CPU) but executes on the device
(GPU).
     * Such functions must have 'void' return type and are launched
using the
     * special CUDA launch syntax with triple angle brackets <<< >>>.
     */

    /*
     * Built-in variables:
     *   blockIdx.x   → Index of the current block within the grid.
     *   threadIdx.x  → Index of the current thread within its
block.
     *   blockDim.x   → Number of threads per block (size of the
block).
     *
     * The combination of these three variables gives each thread a
unique global
     * index value, allowing it to operate on a distinct element of
the array.
     */
    int i = blockIdx.x * blockDim.x + threadIdx.x;

    /*
     * Boundary check:
     * It is possible that the total number of launched threads is
greater than N.
     * This condition ensures that threads with an index beyond the
last element
     * do not access invalid memory locations.
     */
    if (i < N)
    {
        /*
         * Core computation:
         * Each thread adds one pair of corresponding elements from A
and B,
         * and writes the result to C at the same index.
         *
         * Since all threads execute concurrently, the entire vector
addition
         * is completed in parallel on the GPU.
         */
        C[i] = A[i] + B[i];
    }
}

//
---------------------------------------------------------------------
--------
// Function to verify the results between CPU and GPU computations
//
---------------------------------------------------------------------
--------
bool verifyResults(const std::vector<float> &A, const
std::vector<float> &B, const std::vector<float> &C)
{
    for (size_t i = 0; i < A.size(); ++i)
    {
        float expected = A[i] + B[i];
        if (fabs(C[i] - expected) > 1e-5)
            return false;
    }
    return true;
}

//
----------------------------------------------------------------------
--------
// Main function
//
----------------------------------------------------------------------
--------
int main()
{
    // -----------------------------
    // Step 1: Define problem size
    // -----------------------------
    int N = 1 << 24; // 16 million elements
    size_t size = N * sizeof(float);
    std::cout << "Vector size: " << N << " elements (" << size /
(1024.0 * 1024.0)
             << " MB per array)" << "\n";

    // -----------------------------
    // Step 2: Allocate host memory
    // -----------------------------
    std::vector<float> h_A(N, 1.0f);
    std::vector<float> h_B(N, 2.0f);
    std::vector<float> h_C(N); // For GPU result
    std::vector<float> h_C_ref(N); // For CPU result reference

    //
======================================================================
====
    // SECTION A: CPU IMPLEMENTATION (SEQUENTIAL)
    // Performs vector addition using a single CPU thread.
    //
======================================================================
====
    auto cpu_start = high_resolution_clock::now();

    for (int i = 0; i < N; ++i)
        h_C_ref[i] = h_A[i] + h_B[i];

    auto cpu_end = high_resolution_clock::now();
    double cpu_time = duration_cast<milliseconds>(cpu_end -
cpu_start).count();
    std::cout << "CPU Execution Time: " << cpu_time << " ms" <<
std::endl;

    //
======================================================================
====
    // SECTION B: GPU IMPLEMENTATION (PARALLEL)
    // Demonstrates memory allocation, data transfer, kernel launch,
and timing.
    //
======================================================================
====

    // Step 1: Allocate device memory
    float *d_A, *d_B, *d_C;
    cudaMalloc(&d_A, size);
    cudaMalloc(&d_B, size);
    cudaMalloc(&d_C, size);

    // Step 2: Copy input data from host to device
    cudaMemcpy(d_A, h_A.data(), size, cudaMemcpyHostToDevice);
    cudaMemcpy(d_B, h_B.data(), size, cudaMemcpyHostToDevice);

    // Step 3: Configure CUDA kernel launch parameters
    int threadsPerBlock = 256;
    int blocksPerGrid = (N + threadsPerBlock - 1) / threadsPerBlock;

    // Step 4: Use CUDA events for precise GPU timing
    cudaEvent_t startEvent, stopEvent;
    cudaEventCreate(&startEvent);
    cudaEventCreate(&stopEvent);

    cudaEventRecord(startEvent, 0); // Record start time on GPU

    // Launch kernel
    vectorAdd<<<blocksPerGrid, threadsPerBlock>>>(d_A, d_B, d_C, N);

    cudaEventRecord(stopEvent, 0);  // Record stop time on GPU
    cudaEventSynchronize(stopEvent); // Wait for kernel to finish

    // Calculate elapsed time
    float gpu_time = 0.0f;
    cudaEventElapsedTime(&gpu_time, startEvent, stopEvent);

    cudaEventDestroy(startEvent);
    cudaEventDestroy(stopEvent);

    // Step 5: Copy result back to host
    cudaMemcpy(h_C.data(), d_C, size, cudaMemcpyDeviceToHost);

    // Step 6: Free device memory
    cudaFree(d_A);
    cudaFree(d_B);
    cudaFree(d_C);

    std::cout << "GPU Execution Time: " << gpu_time << " ms" << "\n";

    //
======================================================================
====
    // SECTION C: RESULT VERIFICATION AND PERFORMANCE ANALYSIS
    //
======================================================================
====
    bool ok = verifyResults(h_A, h_B, h_C);
    std::cout << "Result Verification: " << (ok ? "PASS" : "FAIL") <<
"\n";
    std::cout << "Sample value check: C[0] = " << h_C[0] << "\n";
    double speedup = cpu_time / gpu_time;
    std::cout << "\nSpeedup = " << speedup << "x faster on GPU" <<
"\n";
    std::cout << "-------------------------------------------" <<
"\n";
    return 0;
}
```

NVIDIA GPU Configuration on AWS EC2

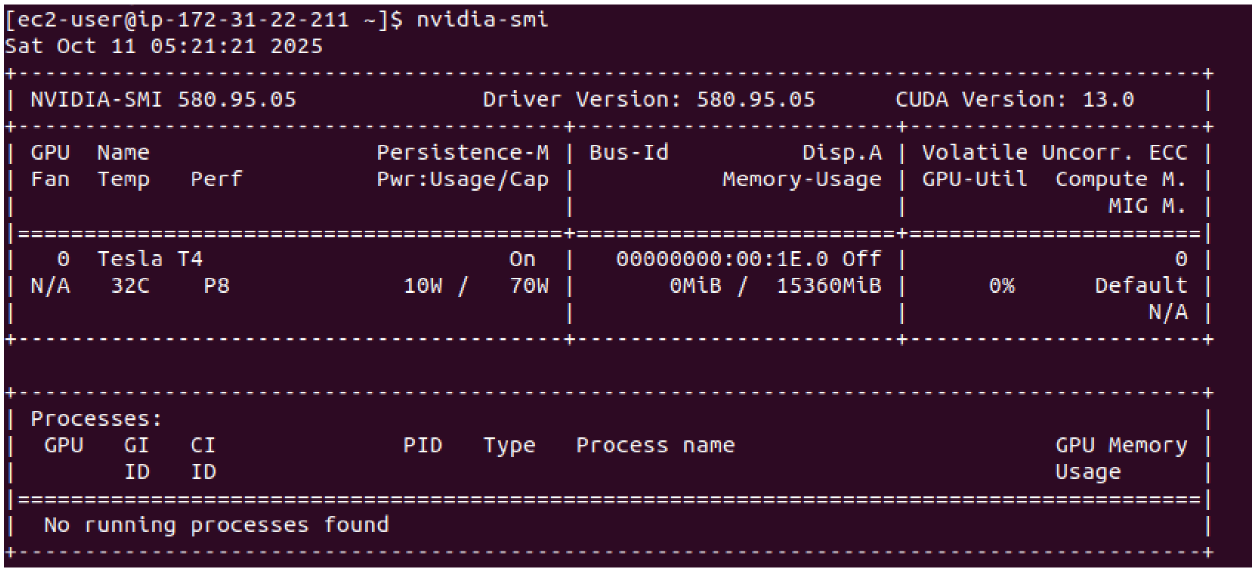
The CUDA program was executed on an **Amazon Web Services (AWS)** EC2 instance configured as follows:

- **Instance Type:** `g4dn.xlarge` (The g4dn.xlarge instance was chosen because it offers a balanced combination of GPU acceleration (NVIDIA T4), adequate CPU resources, and affordable pricing, making it ideal for running and benchmarking moderate-scale CUDA programs.)
- **GPU:** 1 × NVIDIA **Tesla T4** (Turing architecture)
- **vCPUs:** 4 virtual CPUs
- **Memory:** 16 GB RAM
- **Operating System:** Amazon Linux 2023 (Deep Learning Base AMI)
- **NVIDIA Driver Version:** 580.95.05
- **CUDA Toolkit Version:** 12.8 (preinstalled with AMI)
- **Compiler Used:** `nvcc` (NVIDIA CUDA Compiler)
- **Access Method:** SSH connection using key pair and public DNS
- **Verification Command:** `nvidia-smi` used to confirm GPU presence and driver installation
- **Purpose:** Execution and performance comparison of the CUDA vector addition program against its CPU equivalent

Running the CUDA Program

```
[ec2-user@ip-172-31-25-183 GPU_CUDA_PROGRAMMING]$
./vector_add_compare
Vector size: 16777216 elements (64 MB per array)
CPU Execution Time: 124 ms
GPU Execution Time: 0.820064 ms
Result Verification: PASS
Sample value check: C[0] = 3
Speedup = 151.208x faster on GPU
```

Screenshots



```
[ec2-user@ip-172-31-22-211 ~]$ nvidia-smi
Sat Oct 11 05:21:21 2025
+-----------------------------------------------------------------------------------------+
| NVIDIA-SMI 580.95.05              Driver Version: 580.95.05      CUDA Version: 13.0     |
+-----------------------------------------+------------------------+----------------------+
| GPU  Name                 Persistence-M | Bus-Id          Disp.A | Volatile Uncorr. ECC |
| Fan  Temp   Perf          Pwr:Usage/Cap |           Memory-Usage | GPU-Util  Compute M. |
|                                         |                        |               MIG M. |
|=========================================+========================+======================|
|   0  Tesla T4                       On  |   00000000:00:1E.0 Off |                    0 |
| N/A   32C    P8             10W /   70W |       0MiB /  15360MiB |      0%      Default |
|                                         |                        |                  N/A |
+-----------------------------------------+------------------------+----------------------+

+-----------------------------------------------------------------------------------------+
| Processes:                                                                              |
|  GPU   GI   CI              PID   Type   Process name                        GPU Memory |
|        ID   ID                                                               Usage      |
|=========================================================================================|
|  No running processes found                                                             |
+-----------------------------------------------------------------------------------------+
```

The above screenshot was captured while running the CUDA program on the EC2 instance using the nvidia-smi command in the console. The extracted information about the NVIDIA GPU configuration is as follows:

Driver Version: 580.95.05
CUDA Version: 13.0
GPU Name: Tesla T4
GPU Temperature: 32°C
Performance State (Perf): P8
Power Usage: 10 W
Power Capacity: 70 W
Bus ID: 00000000:00:1E.0
Display Active: Off
Memory Usage: 0 MiB / 15360 MiB
GPU Utilisation: 0%
Compute Mode: Default
Running Processes: None

Code Walkthrough

Now, let's have a look at the different parts of the question and try to understand how this program addresses those.

Objective

The goal of this exercise is to write and execute a CUDA program that performs **parallel vector addition** on two large arrays using an NVIDIA GPU. The computation is offloaded from the CPU to the GPU through CUDA kernel functions. The program demonstrates memory allocation on both the host (CPU) and device (GPU), data transfer between them, and result verification. Finally, the GPU version's execution time is compared with a simple CPU implementation to highlight the performance difference.

"Use CUDA kernel functions to offload the computation to the GPU."

Host and Device Setup

The program first defines the problem size and sets up memory for both the CPU and GPU. CUDA programs use a **host–device model**, where the CPU manages the workflow and the GPU performs the parallel computation.

```
int n = 1<<20;  // 1 million elements
size_t bytes = n * sizeof(float);

float *h_A, *h_B, *h_C;           // Host arrays
float *d_A, *d_B, *d_C;           // Device arrays

h_A = (float*)malloc(bytes);
h_B = (float*)malloc(bytes);
h_C = (float*)malloc(bytes);
```

At this stage, memory for the input and output arrays is allocated on the host. Corresponding memory on the GPU is created later using CUDA-specific functions.

“Demonstrate memory allocation, data transfer between host and device, and result verification.”

Initialising Input Data

Before running the kernel on the GPU, the program fills the host arrays `h_A` and `h_B` with known values so that the results can be easily checked later.

```
for (int i = 0; i < n; i++) {
    h_A[i] = 1.0f;
    h_B[i] = 2.0f;
}
```

Every element of the resulting vector `C` should therefore be `3.0f`.

Memory Allocation on GPU and Data Transfer

Once the host data is ready, the program allocates equivalent space on the GPU and transfers the data across.

```
cudaMalloc(&d_A, bytes);
cudaMalloc(&d_B, bytes);
cudaMalloc(&d_C, bytes);

cudaMemcpy(d_A, h_A, bytes, cudaMemcpyHostToDevice);
cudaMemcpy(d_B, h_B, bytes, cudaMemcpyHostToDevice);
```

- cudaMalloc() allocates memory on the GPU.
- cudaMemcpy() handles data transfer between CPU and GPU memory spaces.
  This step is crucial, since the GPU cannot directly access host memory.

CUDA Kernel for Vector Addition

The computation is handled by a **CUDA kernel**—a special function that runs on the GPU across thousands of threads in parallel.

```
int threads = 256;
int blocks = (n + threads - 1) / threads;
```

```
vectorAdd<<<blocks, threads>>>(d_A, d_B, d_C, n);
cudaDeviceSynchronize();
```

Here:

- 256 threads are grouped into each block.
- The number of blocks is calculated to ensure all elements are covered.
- cudaDeviceSynchronize() ensures that the CPU waits for the GPU to finish before continuing.

“Demonstrate result verification.”

Copying Results Back and Verifying Correctness

Once the GPU finishes its work, the results are copied back to the host for verification.

```
cudaMemcpy(h_C, d_C, bytes, cudaMemcpyDeviceToHost);
for (int i = 0; i < n; i++) {
    if (h_C[i] != 3.0f) {
        printf("Error at index %d\n", i);
        break;
    }
}
```

This confirms that every computed value matches the expected sum.

“Compare the execution time of your CUDA program with a basic CPU implementation.”

CPU Version for Comparison

To evaluate performance, the same vector addition is implemented sequentially on the CPU:

```
for (int i = 0; i < n; i++) {
    h_C[i] = h_A[i] + h_B[i];
}
```

The two implementations (CPU and GPU) are timed separately so their performance can be compared.

“Compare the execution time of your CUDA program with a basic CPU implementation.”

Measuring Execution Time

CUDA provides event APIs that accurately measure the time taken for GPU operations.

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

cudaEventRecord(start);
vectorAdd<<<blocks, threads>>>(d_A, d_B, d_C, n);
cudaEventRecord(stop);

cudaEventSynchronize(stop);
float milliseconds = 0;
cudaEventElapsedTime(&milliseconds, start, stop);
```

This measures the GPU kernel execution time in milliseconds, excluding initial data transfers. A similar timing method (e.g., clock()) is used for the CPU version.

Cleanup and Memory Deallocation

```
cudaFree(d_A);
cudaFree(d_B);
cudaFree(d_C);
free(h_A);
free(h_B);
free(h_C);
```

After all computations, both GPU and host memory are released to prevent memory leaks.

Compare the execution time of your CUDA program with a basic CPU implementation, and briefly discuss the performance difference.

Comparison of Execution Time and Performance Discussion

For a vector size of **16,777,216 elements (64 MB per array)**, the **CPU implementation** completed the computation in **124 milliseconds**, whereas the **GPU version** took only **0.820 milliseconds**, resulting in a **speed-up of about 151**×.

This performance difference arises from the architectural contrast between CPUs and GPUs. A CPU executes tasks sequentially or across a small number of general-purpose cores, while a GPU comprises thousands of lightweight cores capable of handling many threads simultaneously. This enables highly efficient parallel computation, particularly for large, data-parallel workloads such as vector addition. Although data transfer overhead can reduce GPU gains for small vectors, its impact diminishes with larger datasets, where the GPU's parallel throughput becomes the decisive factor.

4. Write a simple "Hello, World!" program in C and compile it on a Linux system. Using tools such as strace, trace and analyze the sequence of system calls invoked during its execution. Provide a detailed explanation of the system calls observed. Additionally, discuss how these calls illustrate the role of the operating system in program execution, particularly in process creation, I/O operations, and program termination. [2 Marks] [In case of Windows 11, use WSL (Windows Subsystem for Linux)]

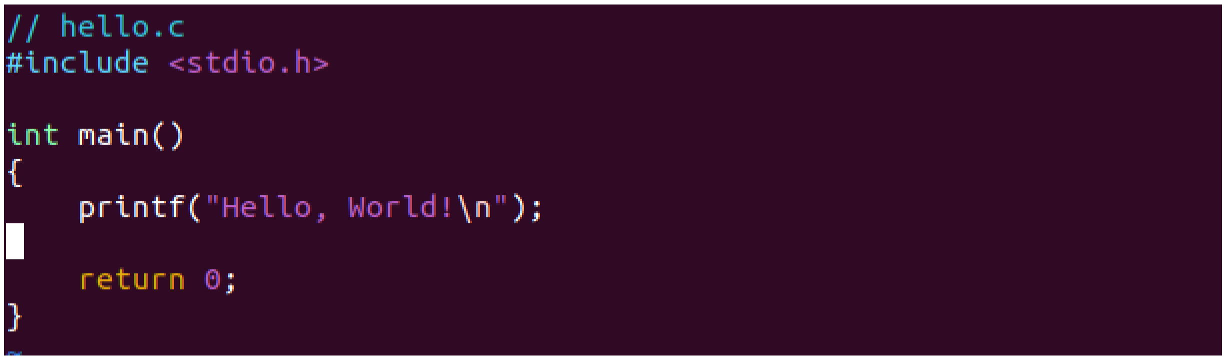
Answer

Writing the Hello World program in C

The following is a simple "Hello World" program written in C:

```c
// hello.c
#include <stdio.h>

int main()
{
    printf("Hello, World!\n");

    return 0;
}
```

Screenshot



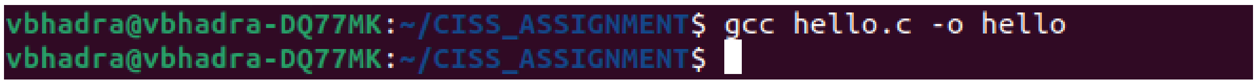
```
// hello.c
#include <stdio.h>

int main()
{
    printf("Hello, World!\n");

    return 0;
}
```

Compile the program in Linux

The following command was used to compile the program on a Ubuntu Linux console:

```
vbhadra@vbhadra-DQ77MK:~/CISS_ASSIGNMENT$ gcc hello.c -o hello
```

Screenshot



```
vbhadra@vbhadra-DQ77MK:~/CISS_ASSIGNMENT$ gcc hello.c -o hello
vbhadra@vbhadra-DQ77MK:~/CISS_ASSIGNMENT$
```

Running Hello World program on the console

```
vbhadra@vbhadra-DQ77MK:~/CISS_ASSIGNMENT$ ./hello
Hello, World!
```

As we can see the Hello World program is printing the string “Hello, World!” on the console. The output confirms that the program executes correctly.

Running hello world program with strace

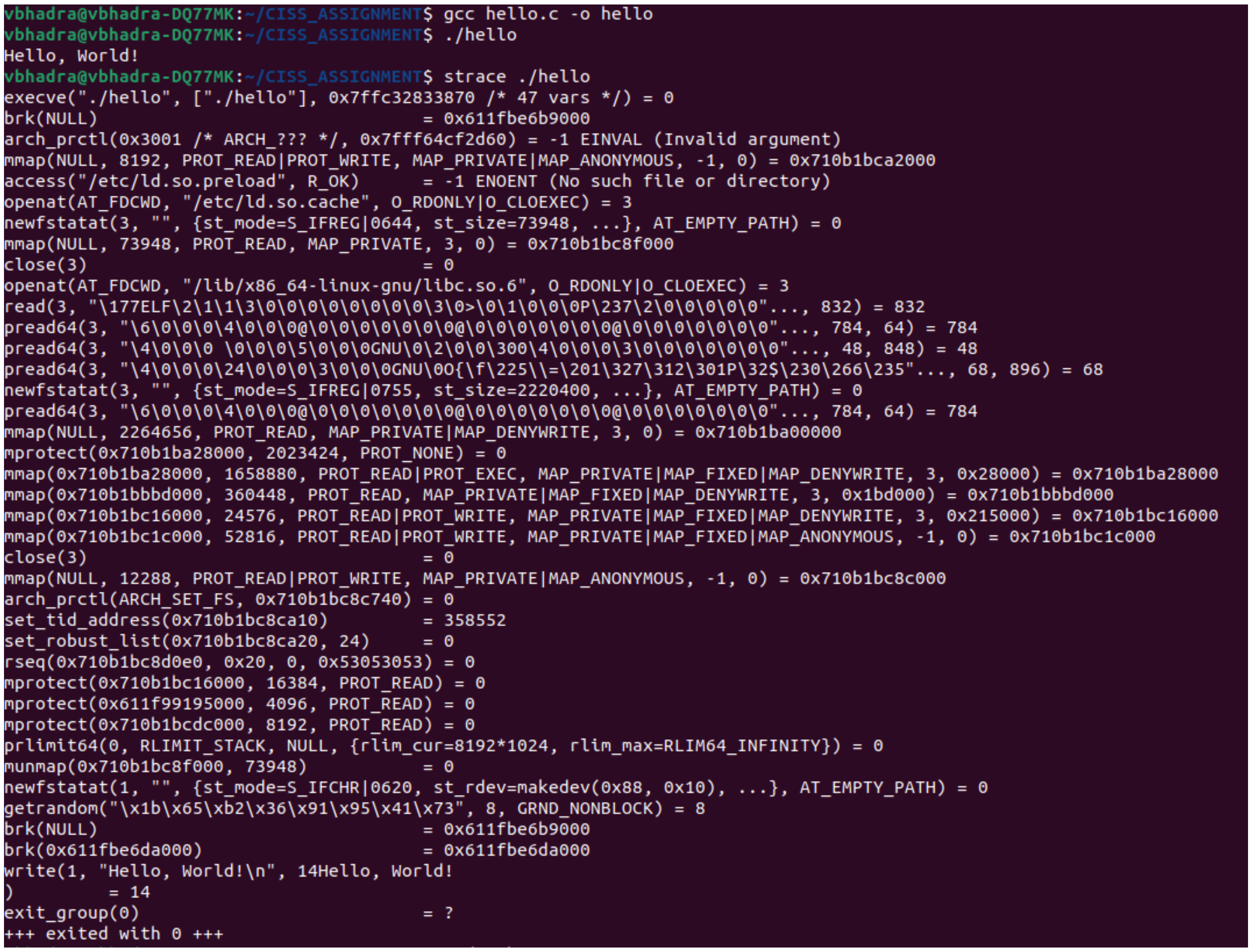
Next, the hello world program was run with strace on an Ubuntu Linux console. Here is the output from strace:

```
vbhadra@vbhadra-DQ77MK:~/CISS_ASSIGNMENT$ strace ./hello
execve("./hello", ["./hello"], 0x7ffc32833870 /* 47 vars */) = 0
brk(NULL)                               = 0x611fbe6b9000
arch_prctl(0x3001 /* ARCH_??? */, 0x7fff64cf2d60) = -1 EINVAL
(Invalid argument)
```

```
mmap(NULL, 8192, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1,
0) = 0x710b1bca2000
access("/etc/ld.so.preload", R_OK)      = -1 ENOENT (No such file or
directory)
openat(AT_FDCWD, "/etc/ld.so.cache", O_RDONLY|O_CLOEXEC) = 3
newfstatat(3, "", {st_mode=S_IFREG|0644, st_size=73948, ...},
AT_EMPTY_PATH) = 0
mmap(NULL, 73948, PROT_READ, MAP_PRIVATE, 3, 0) = 0x710b1bc8f000
close(3)                                = 0
openat(AT_FDCWD, "/lib/x86_64-linux-gnu/libc.so.6",
O_RDONLY|O_CLOEXEC) = 3
read(3,
"\177ELF\2\1\1\3\0\0\0\0\0\0\0\0\3\0>\0\1\0\0\0P\237\2\0\0\0\0\0"...,
832) = 832
pread64(3,
"\6\0\0\0\4\0\0\0@\0\0\0\0\0\0\0@\0\0\0\0\0\0\0@\0\0\0\0\0\0\0"...,
784, 64) = 784
pread64(3, "\4\0\0\0
\0\0\0\5\0\0\0GNU\0\2\0\0\300\4\0\0\0\3\0\0\0\0\0\0\0"..., 48, 848) =
48
pread64(3,
"\4\0\0\0\24\0\0\0\3\0\0\0GNU\0O{\f\225\\=\201\327\312\301P\32$\230\2
66\235"..., 68, 896) = 68
newfstatat(3, "", {st_mode=S_IFREG|0755, st_size=2220400, ...},
AT_EMPTY_PATH) = 0
pread64(3,
"\6\0\0\0\4\0\0\0@\0\0\0\0\0\0\0@\0\0\0\0\0\0\0@\0\0\0\0\0\0\0"...,
784, 64) = 784
mmap(NULL, 2264656, PROT_READ, MAP_PRIVATE|MAP_DENYWRITE, 3, 0) =
0x710b1ba00000
mprotect(0x710b1ba28000, 2023424, PROT_NONE) = 0
mmap(0x710b1ba28000, 1658880, PROT_READ|PROT_EXEC,
MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE, 3, 0x28000) = 0x710b1ba28000
mmap(0x710b1bbbd000, 360448, PROT_READ,
MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE, 3, 0x1bd000) = 0x710b1bbbd000
mmap(0x710b1bc16000, 24576, PROT_READ|PROT_WRITE,
MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE, 3, 0x215000) = 0x710b1bc16000
mmap(0x710b1bc1c000, 52816, PROT_READ|PROT_WRITE,
MAP_PRIVATE|MAP_FIXED|MAP_ANONYMOUS, -1, 0) = 0x710b1bc1c000
```

```
close(3)                                = 0
mmap(NULL, 12288, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS,
-1, 0) = 0x710b1bc8c000
arch_prctl(ARCH_SET_FS, 0x710b1bc8c740) = 0
set_tid_address(0x710b1bc8ca10)         = 358552
set_robust_list(0x710b1bc8ca20, 24)     = 0
rseq(0x710b1bc8d0e0, 0x20, 0, 0x53053053) = 0
mprotect(0x710b1bc16000, 16384, PROT_READ) = 0
mprotect(0x611f99195000, 4096, PROT_READ) = 0
mprotect(0x710b1bcdc000, 8192, PROT_READ) = 0
prlimit64(0, RLIMIT_STACK, NULL, {rlim_cur=8192*1024,
rlim_max=RLIM64_INFINITY}) = 0
munmap(0x710b1bc8f000, 73948)           = 0
newfstatat(1, "", {st_mode=S_IFCHR|0620, st_rdev=makedev(0x88, 0x10),
...}, AT_EMPTY_PATH) = 0
getrandom("\x1b\x65\xb2\x36\x91\x95\x41\x73", 8, GRND_NONBLOCK) = 8
brk(NULL)                               = 0x611fbe6b9000
brk(0x611fbe6da000)                     = 0x611fbe6da000
write(1, "Hello, World!\n", 14Hello, World!
)         = 14
exit_group(0)                           = ?
+++ exited with 0 +++
```

Screenshot



```
vbhadra@vbhadra-DQ77MK:~/CISS_ASSIGNMENT$ gcc hello.c -o hello
vbhadra@vbhadra-DQ77MK:~/CISS_ASSIGNMENT$ ./hello
Hello, World!
vbhadra@vbhadra-DQ77MK:~/CISS_ASSIGNMENT$ strace ./hello
execve("./hello", ["./hello"], 0x7ffc32833870 /* 47 vars */) = 0
brk(NULL)                               = 0x611fbe6b9000
arch_prctl(0x3001 /* ARCH_??? */, 0x7fff64cf2d60) = -1 EINVAL (Invalid argument)
mmap(NULL, 8192, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0x710b1bca2000
access("/etc/ld.so.preload", R_OK)      = -1 ENOENT (No such file or directory)
openat(AT_FDCWD, "/etc/ld.so.cache", O_RDONLY|O_CLOEXEC) = 3
newfstatat(3, "", {st_mode=S_IFREG|0644, st_size=73948, ...}, AT_EMPTY_PATH) = 0
mmap(NULL, 73948, PROT_READ, MAP_PRIVATE, 3, 0) = 0x710b1bc8f000
close(3)                                = 0
openat(AT_FDCWD, "/lib/x86_64-linux-gnu/libc.so.6", O_RDONLY|O_CLOEXEC) = 3
read(3, "\177ELF\2\1\1\3\0\0\0\0\0\0\0\0\3\0>\0\1\0\0\0P\237\2\0\0\0\0\0"..., 832) = 832
pread64(3, "\6\0\0\0\4\0\0\0@\0\0\0\0\0\0\0@\0\0\0\0\0\0\0@\0\0\0\0\0\0\0"..., 784, 64) = 784
pread64(3, "\4\0\0\0 \0\0\0\5\0\0\0GNU\0\2\0\0\300\4\0\0\0\3\0\0\0\0\0\0\0"..., 48, 848) = 48
pread64(3, "\4\0\0\0\24\0\0\0\3\0\0\0GNU\0O{\f\225\\=\201\327\312\301P\32$\230\266\235"..., 68, 896) = 68
newfstatat(3, "", {st_mode=S_IFREG|0755, st_size=2220400, ...}, AT_EMPTY_PATH) = 0
pread64(3, "\6\0\0\0\4\0\0\0@\0\0\0\0\0\0\0@\0\0\0\0\0\0\0@\0\0\0\0\0\0\0"..., 784, 64) = 784
mmap(NULL, 2264656, PROT_READ, MAP_PRIVATE|MAP_DENYWRITE, 3, 0) = 0x710b1ba00000
mprotect(0x710b1ba28000, 2023424, PROT_NONE) = 0
mmap(0x710b1ba28000, 1658880, PROT_READ|PROT_EXEC, MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE, 3, 0x28000) = 0x710b1ba28000
mmap(0x710b1bbbd000, 360448, PROT_READ, MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE, 3, 0x1bd000) = 0x710b1bbbd000
mmap(0x710b1bc16000, 24576, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE, 3, 0x215000) = 0x710b1bc16000
mmap(0x710b1bc1c000, 52816, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_FIXED|MAP_ANONYMOUS, -1, 0) = 0x710b1bc1c000
close(3)                                = 0
mmap(NULL, 12288, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0x710b1bc8c000
arch_prctl(ARCH_SET_FS, 0x710b1bc8c740) = 0
set_tid_address(0x710b1bc8ca10)         = 358552
set_robust_list(0x710b1bc8ca20, 24)     = 0
rseq(0x710b1bc8d0e0, 0x20, 0, 0x53053053) = 0
mprotect(0x710b1bc16000, 16384, PROT_READ) = 0
mprotect(0x611f99195000, 4096, PROT_READ) = 0
mprotect(0x710b1bcdc000, 8192, PROT_READ) = 0
prlimit64(0, RLIMIT_STACK, NULL, {rlim_cur=8192*1024, rlim_max=RLIM64_INFINITY}) = 0
munmap(0x710b1bc8f000, 73948)           = 0
newfstatat(1, "", {st_mode=S_IFCHR|0620, st_rdev=makedev(0x88, 0x10), ...}, AT_EMPTY_PATH) = 0
getrandom("\x1b\x65\xb2\x36\x91\x95\x41\x73", 8, GRND_NONBLOCK) = 8
brk(NULL)                               = 0x611fbe6b9000
brk(0x611fbe6da000)                     = 0x611fbe6da000
write(1, "Hello, World!\n", 14Hello, World!
)         = 14
exit_group(0)                           = ?
+++ exited with 0 +++
```

Below is the **summary of the system call trace** captured on the Ubuntu system:

```
execve("./hello", ["./hello"], 0x7ffc32833870 /* 47 vars */) = 0
brk(NULL)                               = 0x611fbe6b9000
arch_prctl(0x3001 /* ARCH_??? */, 0x7fff64cf2d60) = -1 EINVAL
(Invalid argument)
mmap(NULL, 8192, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1,
0) = 0x710b1bca2000
access("/etc/ld.so.preload", R_OK)      = -1 ENOENT (No such file or
directory)
openat(AT_FDCWD, "/etc/ld.so.cache", O_RDONLY|O_CLOEXEC) = 3
...
write(1, "Hello, World!\n", 14Hello, World!
)         = 14
exit_group(0)                           = ?
```

```
+++ exited with 0 +++
```

Key System Calls and Their Roles

Even a tiny executable like this goes through several distinct stages, each represented by different system calls from Linux's kernel interface.

Program Launch – execve()

The very first thing that happens is the shell calls execve(). This system call instructs the kernel to throw out whatever code was running in this process before and load our compiled program's instructions and data into memory. The runtime environment: arguments, environment variables, memory layout is set up, and the processor's execution point jumps to our program's main() function. In short, execve() is the doorway through which our binary steps into execution.

Memory Setup – brk() and mmap()

Next, the kernel starts shaping the memory space the program will need.

- brk() nudges the end of the heap, creating space for variables and any dynamic allocations that might happen.
- mmap() brings in new blocks of virtual memory. Some of these are for our own use; others are reserved for loading shared libraries such as libc.so.6.

These calls underscore that even the smallest program depends on careful memory management, with the kernel partitioning space for code, data, heap, and stack.

Loading Libraries – openat(), read(), close()

Before your own code executes in earnest, the dynamic linker has to find and load dependent libraries. That's when we see openat() reading files like /etc/ld.so.cache, followed by requests to open and read /lib/x86_64-linux-gnu/libc.so.6. After the required data is loaded into memory — via a series of read() calls — the files are shut with close(). This part happens automatically; the program doesn't manually fetch its libraries. The operating system ensures all dependencies are where they need to be.

Sending Output – write()

When printf() is used in C, the actual low-level work is done by write().
The trace line:

```
write(1, "Hello, World!\n", 14)
```

reveals exactly how the text leaves your program for the terminal. Here, 1 is the file descriptor representing stdout. The kernel takes the string and safely passes it to the console — no direct hardware poking is allowed from user space.

Clean Exit – exit_group()

Finally, the program ends. The call to exit_group(0) signals a graceful termination. At this point, the kernel recovers any resources linked to the process — memory, file handles, stacks — ensuring no leftover debris can affect system stability. The trace closes with:

```
+++ exited with 0 +++
```

which confirms a normal, successful end.

The Role of OS

The operating system plays a central role in every phase of program execution — from loading the executable to handling I/O and ensuring clean termination. A system call trace of a simple Hello, World! program makes this dependency explicit, revealing the OS's orchestration behind seemingly simple user actions.

Process Creation

When a user runs the program, the execve() system call is the first major interaction. It replaces the current shell process image with that of the new executable, setting up virtual memory regions, stack, and file descriptors required for execution. To support dynamic memory management and library loading, calls such as brk() and mmap() are issued. These extend the heap and map shared libraries into the process's address space. Together, they create an isolated, protected environment in user space, ensuring that one process cannot interfere with another.

I/O Operations

At the point where the program calls printf(), the actual printing to the terminal is performed by the write() system call. The OS here acts as a secure mediator between the user process and physical devices. The function writes 14 bytes — the string "Hello, World!\n" — to the standard output file descriptor (1). Through this mechanism, all forms of I/O, whether writing to files, sockets, or screens, are subject to kernel-managed access control and device abstraction layers, shielding user programs from hardware complexity.

Program Termination

Finally, when the main function completes, the program calls exit_group(0). This system call instructs the OS to reclaim allocated memory, close file descriptors, update process tables, and register the exit status (0 indicating success). The trace line +++ exited with 0 +++ confirms that the OS has finalized the process cleanly. This orderly termination ensures that no resources remain locked or orphaned in the system, maintaining overall stability and resource efficiency.

In essence, even the simplest program relies on the operating system at every step — to initialize its environment, mediate hardware access, and cleanly end its execution — illustrating how the OS provides a controlled, secure foundation for all user-level computation.

Appendix

Setting up the NVIDIA CUDA environment in AWS

The CUDA program was executed in an AWS setup. To set up the AWS I have followed the following steps to setup the NVIDIA GPU based environment before executing the program.

Launching the Right EC2 Instance

To get started with GPU-based CUDA programming on AWS, set up a new EC2 instance. Ensure it has the right AMI (software environment). Confirm the hardware configuration includes GPU support. Follow these steps:

Instance Name

In the *Name and Tags* section, enter a clear name such as CUDA-GPU_EC2.

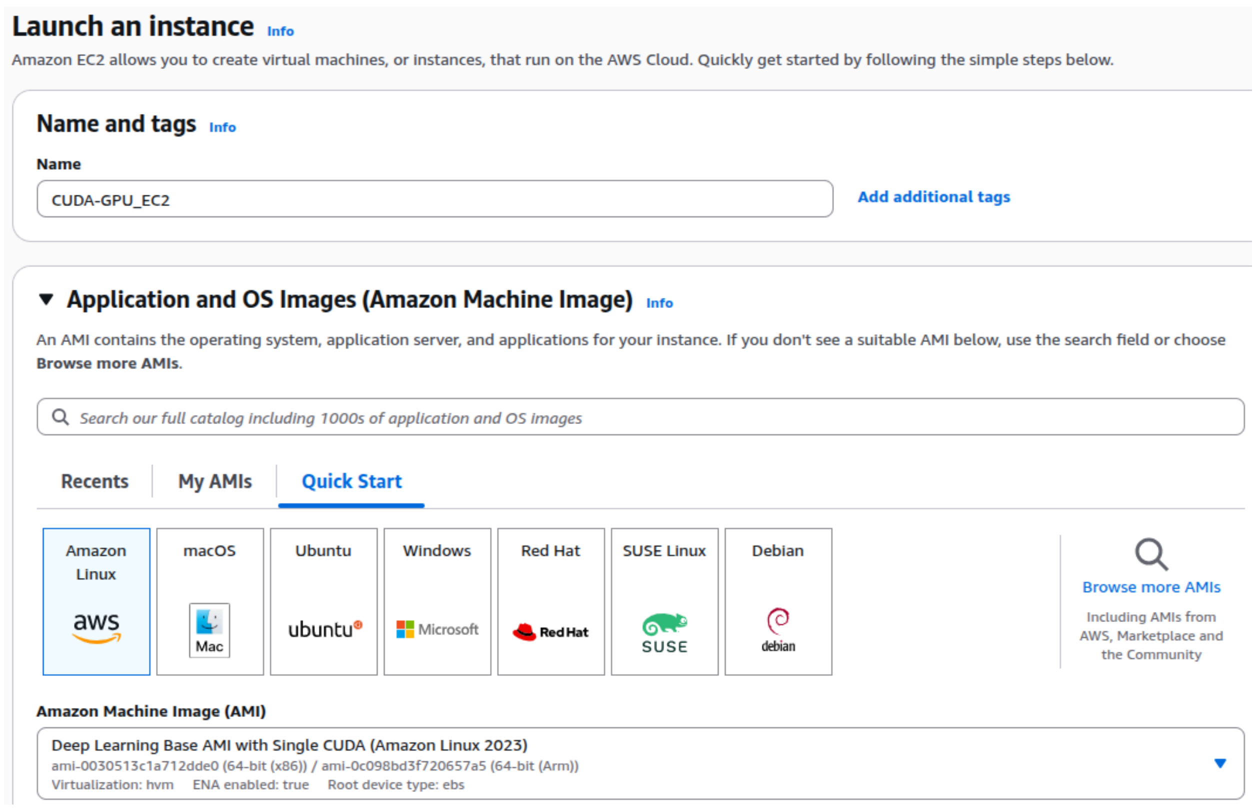
Select the Application and OS Image (AMI)

Under the Quick Start tab, choose Amazon Linux. It's lightweight and stable. Additionally, it is well-supported for CUDA development.
From the available options, select:
Deep Learning Base AMI with Single CUDA (Amazon Linux 2023). This AMI is useful for the following reasons:

- It already comes with the CUDA Toolkit and NVIDIA drivers pre-installed and correctly configured.
- It's clean and minimal — designed for compiler-based CUDA development rather than large AI frameworks.
- It saves setup time because you don't need to install or configure drivers manually.

Architecture

Select 64-bit (x86).


Launch an instance Info
Amazon EC2 allows you to create virtual machines, or instances, that run on the AWS Cloud. Quickly get started by following the simple steps below.
Name and tags Info
Add additional tags
Application and OS Images (Amazon Machine Image) Info
Search our full catalog including 1000s of application and OS images
Recents
My AMIs
Quick Start
Amazon Linux
macOS
Ubuntu
Windows
Red Hat
SUSE Linux
Debian
Browse more AMIs
Including AMIs from AWS, Marketplace and the Community

An AMI contains the operating system, application server, and applications for your instance. If you don't see a suitable AMI below, use the search field or choose **Browse more AMIs**.

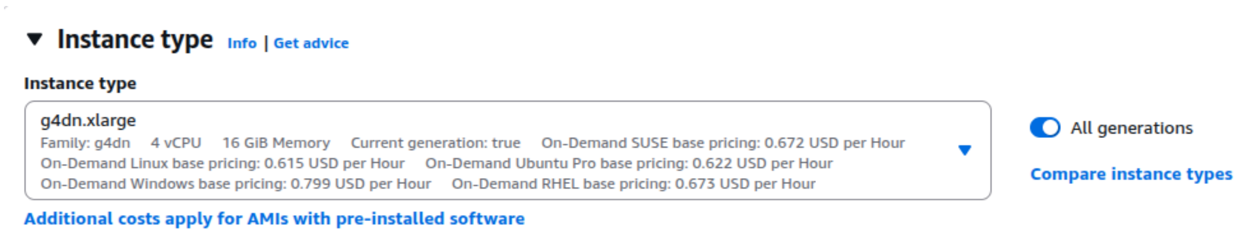
Name: CUDA-GPU_EC2

Amazon Machine Image (AMI): Deep Learning Base AMI with Single CUDA (Amazon Linux 2023)
ami-0030513c1a712dde0 (64-bit (x86)) / ami-0c098bd3f720657a5 (64-bit (Arm))
Virtualization: hvm ENA enabled: true Root device type: ebs

Instance Type

Next, choose an instance type that provides GPU hardware. The most reliable and cost-effective option for testing CUDA programs is:
Instance Type: g4dn.xlarge

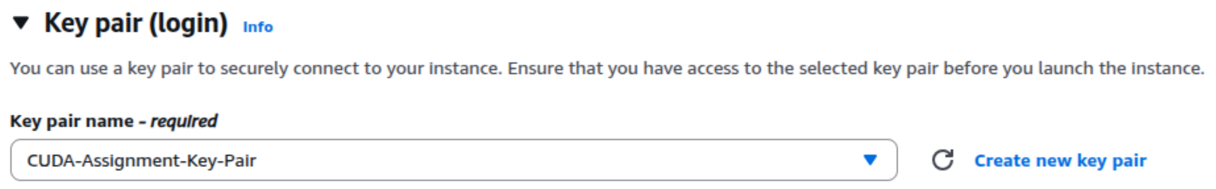
- GPU: 1 × NVIDIA T4
- vCPU: 4
- Memory: 16 GB


Instance type Info | Get advice
All generations
Compare instance types
Additional costs apply for AMIs with pre-installed software

Instance type: g4dn.xlarge
Family: g4dn 4 vCPU 16 GiB Memory Current generation: true On-Demand SUSE base pricing: 0.672 USD per Hour
On-Demand Linux base pricing: 0.615 USD per Hour On-Demand Ubuntu Pro base pricing: 0.622 USD per Hour
On-Demand Windows base pricing: 0.799 USD per Hour On-Demand RHEL base pricing: 0.673 USD per Hour

Key Pair

Choose an existing Key Pair or create a new key pair.


▼ Key pair (login) Info

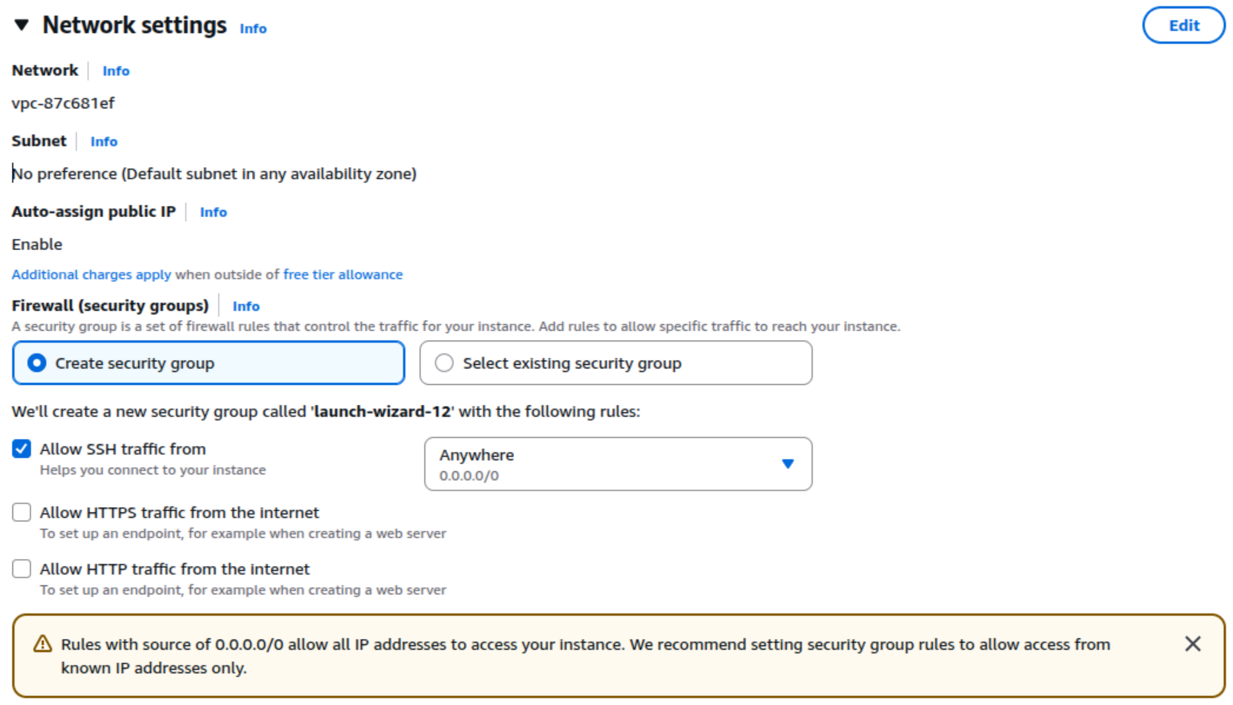
You can use a key pair to securely connect to your instance. Ensure that you have access to the selected key pair before you launch the instance.

Key pair name - *required*

CUDA-Assignment-Key-Pair

Network Settings

Leave the Network Settings as default.


▼ Network settings Info

Network Info

vpc-87c681ef

Subnet Info

No preference (Default subnet in any availability zone)

Auto-assign public IP Info

Enable

Additional charges apply when outside of free tier allowance

Firewall (security groups) Info

A security group is a set of firewall rules that control the traffic for your instance. Add rules to allow specific traffic to reach your instance.

[✓ Create security group] Select existing security group

We'll create a new security group called '**launch-wizard-12**' with the following rules:

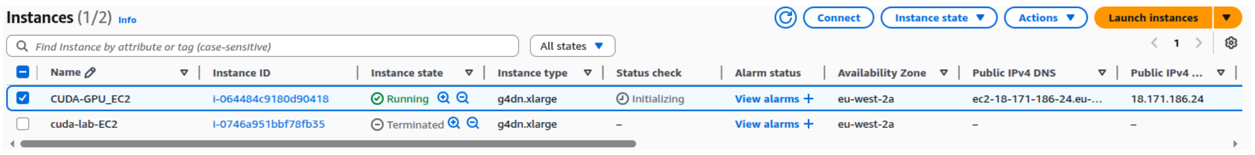
- [✓] Allow SSH traffic from — Anywhere 0.0.0.0/0
  Helps you connect to your instance
- [ ] Allow HTTPS traffic from the internet
  To set up an endpoint, for example when creating a web server
- [ ] Allow HTTP traffic from the internet
  To set up an endpoint, for example when creating a web server

Rules with source of 0.0.0.0/0 allow all IP addresses to access your instance. We recommend setting security group rules to allow access from known IP addresses only.

And leave the rest of the settings to default and launch the instance.

Log in to the EC2

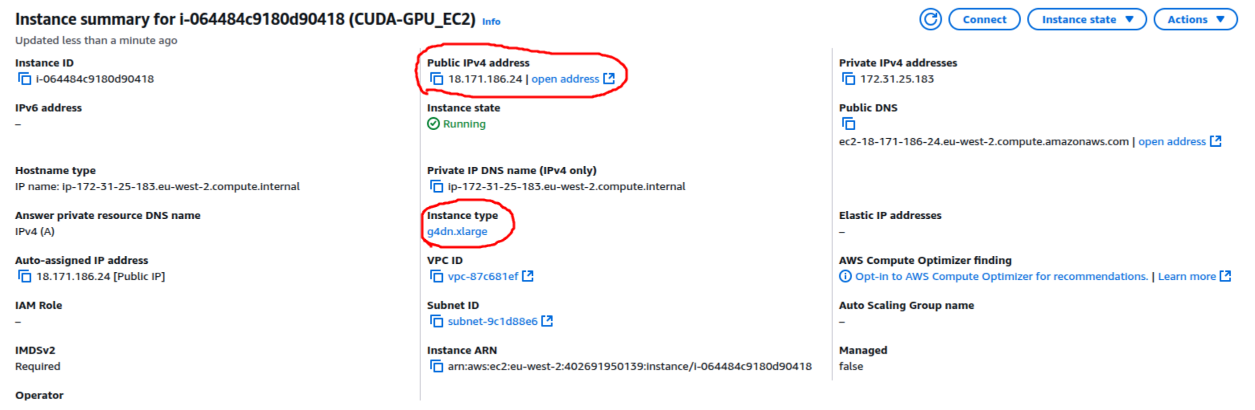
From the EC2->Instances menu you should be able to see your just created instance is running after a few minutes.


Instances (1/2) Info

Find Instance by attribute or tag (case-sensitive) | All states

	Name	Instance ID	Instance state	Instance type	Status check	Alarm status	Availability Zone	Public IPv4 DNS	Public IPv4 ...
✓	CUDA-GPU_EC2	i-064484c9180d90418	Running	g4dn.xlarge	Initializing	View alarms +	eu-west-2a	ec2-18-171-186-24.eu-...	18.171.186.24
	cuda-lab-EC2	i-0746a951bbf78fb35	Terminated	g4dn.xlarge	–	View alarms +	eu-west-2a	–	–

Click on your instance ID and go the the instance details page. Check everything as expected. And then note the public IP of the instance. We have to log in to the instance and start doing the CUDA programming for our GPU.


Instance summary for i-064484c9180d90418 (CUDA-GPU_EC2) Info

Updated less than a minute ago

Instance ID
i-064484c9180d90418

IPv6 address
–

Hostname type
IP name: ip-172-31-25-183.eu-west-2.compute.internal

Answer private resource DNS name
IPv4 (A)

Auto-assigned IP address
18.171.186.24 [Public IP]

IAM Role
–

IMDSv2
Required

Operator

Public IPv4 address
18.171.186.24 | open address

Instance state
Running

Private IP DNS name (IPv4 only)
ip-172-31-25-183.eu-west-2.compute.internal

Instance type
g4dn.xlarge

VPC ID
vpc-87c681ef

Subnet ID
subnet-9c1d88e6

Instance ARN
arn:aws:ec2:eu-west-2:402691950139:instance/i-064484c9180d90418

Private IPv4 addresses
172.31.25.183

Public DNS
ec2-18-171-186-24.eu-west-2.compute.amazonaws.com | open address

Elastic IP addresses
–

AWS Compute Optimizer finding
Opt-in to AWS Compute Optimizer for recommendations. | Learn more

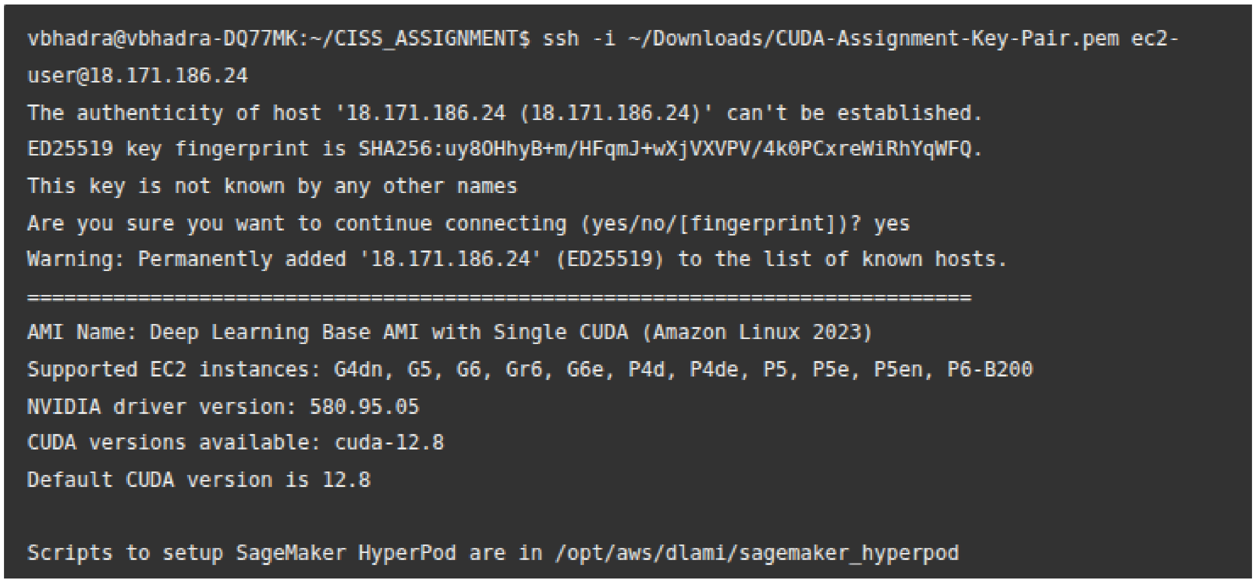
Auto Scaling Group name
–

Managed
false

To log into the instance type in the following from a Linux console or any maybe Putty:

```
ssh -i ~/Downloads/CUDA-Assignment-Key-Pair.pem
ec2-user@18.171.186.24
```

I have my Key Pair located at ~/Downloads/CUDA-Assignment-Key-Pair.pem, hence I have used it. You have to locate your key pair .pem file and use the appropriate PATH in the above command. Also, for this instance type the default user name is ec2-user. Once you issue the above command on a Linux console you should able to log in to the EC2 instance. In my case I see the below when login:



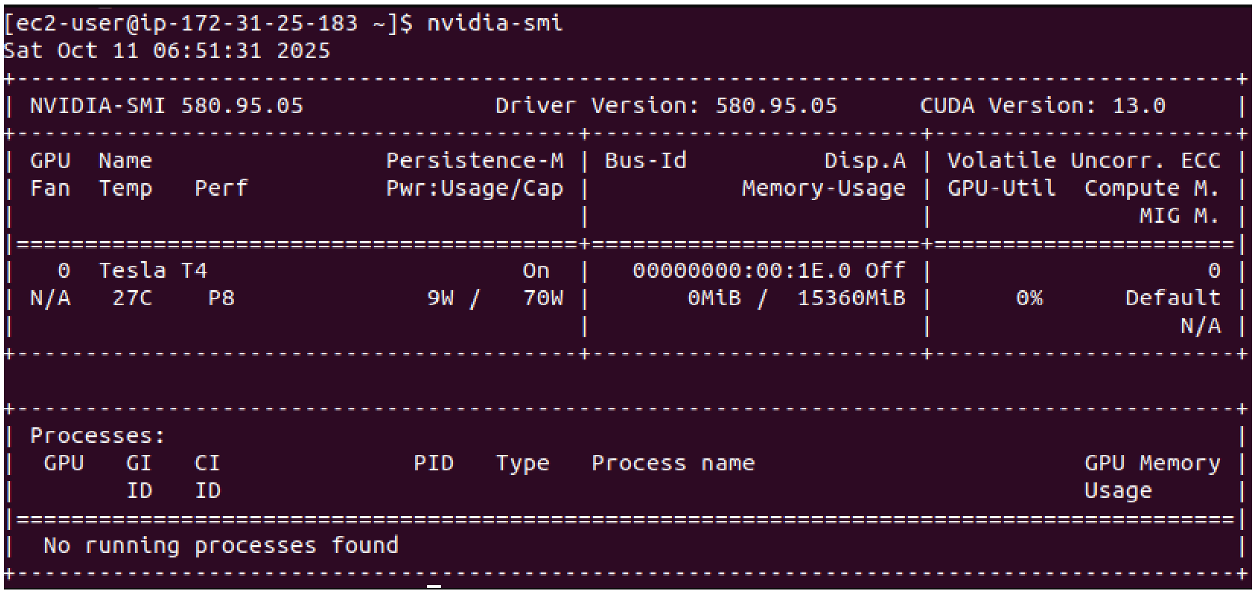
```
vbhadra@vbhadra-DQ77MK:~/CISS_ASSIGNMENT$ ssh -i ~/Downloads/CUDA-Assignment-Key-Pair.pem ec2-
user@18.171.186.24
The authenticity of host '18.171.186.24 (18.171.186.24)' can't be established.
ED25519 key fingerprint is SHA256:uy8OHhyB+m/HFqmJ+wXjVXVPV/4k0PCxreWiRhYqWFQ.
This key is not known by any other names
Are you sure you want to continue connecting (yes/no/[fingerprint])? yes
Warning: Permanently added '18.171.186.24' (ED25519) to the list of known hosts.
============================================================================
AMI Name: Deep Learning Base AMI with Single CUDA (Amazon Linux 2023)
Supported EC2 instances: G4dn, G5, G6, Gr6, G6e, P4d, P4de, P5, P5e, P5en, P6-B200
NVIDIA driver version: 580.95.05
CUDA versions available: cuda-12.8
Default CUDA version is 12.8

Scripts to setup SageMaker HyperPod are in /opt/aws/dlami/sagemaker_hyperpod
```

Check GPU status and driver information

Once you have logged into your EC2 instance you need to verify if you are good to go. Check the GPU driver status using the following command on the console:

```
nvidia-smi
```

It should give a similar output as the following:



```
[ec2-user@ip-172-31-25-183 ~]$ nvidia-smi
Sat Oct 11 06:51:31 2025
+-----------------------------------------------------------------------------------------+
| NVIDIA-SMI 580.95.05              Driver Version: 580.95.05      CUDA Version: 13.0     |
+-----------------------------------------+------------------------+----------------------+
| GPU  Name                 Persistence-M | Bus-Id          Disp.A | Volatile Uncorr. ECC |
| Fan  Temp   Perf          Pwr:Usage/Cap |           Memory-Usage | GPU-Util  Compute M. |
|                                         |                        |               MIG M. |
|=========================================+========================+======================|
|   0  Tesla T4                       On  |   00000000:00:1E.0 Off |                    0 |
| N/A   27C    P8              9W /   70W |       0MiB /  15360MiB |      0%      Default |
|                                         |                        |                  N/A |
+-----------------------------------------+------------------------+----------------------+

+-----------------------------------------------------------------------------------------+
| Processes:                                                                              |
|  GPU   GI   CI              PID   Type   Process name                        GPU Memory |
|        ID   ID                                                               Usage      |
|=========================================================================================|
|  No running processes found                                                             |
+-----------------------------------------------------------------------------------------+
```

Check if the GPU is detected at the hardware level

To confirm that the GPU hardware itself is visible to the operating system, use the lspci command. It lists all connected devices, and you can filter for NVIDIA entries like this:

```
[ec2-user@ip-172-31-25-183 ~]$ lspci | grep -i nvidia
00:1e.0 3D controller: NVIDIA Corporation TU104GL [Tesla T4]
(rev a1)
```

This confirms that the system has detected the NVIDIA Tesla T4 GPU.
At this point, you know that your hardware, driver, and CUDA runtime are all aligned. The environment is ready for compiling and running CUDA programs.

Connecting Your EC2 Instance to GitHub Using SSH

I usually keep all my code in Guthub and then close the code from the Github repository and use it. First, I need to add my RSA key of the EC2 instance to my Github.

Add the key to your GitHub account

- Go to GitHub → Settings → SSH and GPG keys
- Click New SSH key
- Give it a name, e.g., AWS EC2 CUDA
- Paste the public key you just copied
- Click Add SSH key

Clone the CUDA repository

Clone the CUDA code from the repository: GPU_CUDA_PROGRAMMING

Compiling and Running Your CUDA Program

Your CUDA source code is now on the EC2 instance. The next step is to compile it using NVIDIA's CUDA compiler (nvcc). After that, run it on the GPU. The Deep Learning Base AMI with Single CUDA image already comes with nvcc pre-installed. You can compile directly without any extra setup. To compile the CUDA source file, use:

```
[ec2-user@ip-172-31-25-183 GPU_CUDA_PROGRAMMING]$ nvcc
vector_add_compare.cu -o vector_add_compare
nvcc warning : Support for offline compilation for architectures
prior to '_75' will be removed in a future release (Use
-Wno-deprecated-gpu-targets to suppress warning).
[ec2-user@ip-172-31-25-183 GPU_CUDA_PROGRAMMING]$
```

When you compile your CUDA program using nvcc, the NVIDIA CUDA compiler translates the CPU (host) and GPU (device) code. It creates an executable that can run directly on the GPU. To supress the warning above you may like to use the following command:

```
nvcc -Wno-deprecated-gpu-targets vector_add_compare.cu -o
vector_add_compare
```

Running the CUDA Program

After compiling your CUDA program successfully, you can now run it directly on the GPU. This is where you'll see the difference between CPU and GPU execution times in action. Run the program using:

```
[ec2-user@ip-172-31-25-183 GPU_CUDA_PROGRAMMING]$
./vector_add_compare
Vector size: 16777216 elements (64 MB per array)
CPU Execution Time: 124 ms
GPU Execution Time: 0.820064 ms
Result Verification: PASS
Sample value check: C[0] = 3
Speedup = 151.208x faster on GPU
```