HOMEWORK – 1

Name: Shushhma Dewie Koppireddy

Id: 02190512

PROBLEM-1

Laptop Comparison for Scientific Computing and Data Science under $2000

When selecting a laptop for tasks like scientific computing, machine learning, or data science, it's essential to consider the laptop's processing power, memory, GPU capabilities, and storage. The laptops listed below, from five different vendors, are all priced under $2000 and meet the requirements for these data-intensive tasks. This comparison will provide an overview of each laptop's key specifications and features, allowing for a better understanding of which model offers the best value for data science work.

Dell XPS 15

Price: $1,849

Processor: Intel Core i7-12700H (12-core processor for Multi-threaded tasks)

GPU: NVIDIA RTX 3050 (4GB, for machine learning and Deep learning models)

RAM: 16GB DDR5 (suitable for data processing and Multitasking)

Storage: 512GB SSD (fast access for large datasets)

Display: 15.6” FHD+ (1920 * 1200, idea for data Visualization)

Battery: 86Wh (up to 10 hours)

Weight: 4.3 lbs (moderately lightweight and portable)

Ports: Thunderbolt 4, USB-C (high-speed connectivity for external devices)

Operating System: Windows 11 (compatible with data science tools)

Apple MacBook Pro 14” (M2)

Price: $1,999

Processor: M2 Pro (8-core CPU, optimized for high-Performance tasks

GPU: 10-core GPU (integrated for fast computing)

RAM: 16GB Unified Memory (enhanced data processing Speed)

Storage: 512GB SSD (quick access to large datasets)

Display: 14.2” Liquid Retina XDR (superior display quality for visuals)

Battery: Up to 17 hours (extended battery life for long work Sessions)

Weight: 3.5 lbs (lightweight, highly portable)

Ports: Thunderbolt 4, HDMI (supports external displays and fast data transfer)

Operating System: macOS Ventura (optimized for scientific computing)

HP Spectre x360 16”

Price: $1,799

Processor: Intel Core i7-1365U (efficient for handling data science tasks)

GPU: Intel Iris Xe (integrated GPU for basic machine learning)

RAM: 16GB DDR4 (enough memory for multitasking)

Storage: 1TB SSD (ample space for large datasets)

Display: 16” UHD+ touchscreen (high-resolution for interactive data visualization)

Battery: Up to 10hours (decent battery life)

Weight: 4.45 lbs (portable for most uses)

Ports: Thunderbolt 4, USB-A (flexible for data transfer)

Operating System: Windows 11 (compatiable with data Science tools)

Lenovo ThinkPad X1 Carbon Gen 11

Price: $1,850

Processor: Intel Core i7-1355U (great for intensive tasks)

GPU: Intel Iris Xe (good for data visualization)

RAM: 16GB LPDDR5 (fast memory for multitasking)

Storage: 1TB SSD (large storage for data files)

Display: 14” 2.2K (high-resolution for coding and

Battery: Up to 12 hours (long-lasting for extended work)

Weight: 2.48 lbs (extremely lightweight and portable)

Ports: Thunderbolt 4 (fast data transfer and extrenal devices)

Operating System: Windows 11 (pre-installed for data science work)

Razor Blade 15 Base

Price: $1,999

Processor: Intel Core i7-11800H (powerful for machine learning tasks)

GPU: NVIDIA RTX 3060 (6GB, high-end for AI workloads)

RAM: 16GB DDR4 (good for large datasets)

Storage: 15.6” QHD 165Hz (high refresh rate for clear Visuals)

Battery: Up to 8 hours (moderate for short sessions)

Weight: 4.4 lbs (portable but powerful)

Ports: Thunderbolt 4, HDMI (support external devices and Displays)

Operating System: Windows 11 (compatible with data science tools)

PROBLEM – 2

Computer Workstation Comparison for Scientific Computing and Data Science under $10,000

When choosing a high-performance computer workstation for Scientific computing, machine learning, or data science, it's important to consider a powerful CPU, ample memory, a dedicated GPU, and additional peripherals like a 4k monitor, mouse, and keyboard. Below is a comparison of workstations from five different vendors, all staying within the $10,000 budget while meeting the required specifications.

Dell Precision 7865 Tower

Price: $9,550

Processor: AMD Ryzen Threadripper PRO 5965WX (24 cores, 48 threads for massive parallel workloads)

GPU: NVIDIA RTX A5000 (24GB GDDR6, optimized for Machine learning and AI tasks)

RAM: 128GB DDR4 ECC (ensure stability for large-scale Data processing)

Storage: 2TB PCle NVMe SSD (fast access to large datasets)

Monitor: Dell UltraSharp 32" 4K Monitor (high-resolution For detailed visualizations)

Keyboard: Dell Pro Keyboard

Mouse: Dell Pro Mouse

Ports: USB-C, USB-A, HDMI, DisplayPort (high Connectivity for external devices)

Operating System: Windows 11 Pro (compatible with data science tools)

Apple Mac Studio (M2 Ultra)

Price: $9,799 (including GPU, monitor, mouse, and keyboard)

Processor: M2 Ultra (24-core CPU, designed for intensive machine learning and scientific tasks)

GPU: 76-core GPU (designed for fast parallel processing and AI tasks)

RAM: 128GB Unified Memory (seamless handling of large datasets)

Storage: 4TB SSD (ample storage for high-volume data)

Monitor: Apple Studio Display 27” 5K(extremely sharp display for data visualization and code readability)

Keyboard: Apple Magic Keyboard

Mouse: Apple Magic Mouse

Ports: Thunderbolt 4, USB-C (excellent for fast data transfer and external devices)

Operating System: macOS Sonoma (optimized for data

science environments)

HP Z8 G4 Workstation

Price: $9,400 (Including GPU, monitor , mouse, and keyboard)

Processor: Intel Xeon Silver 4314 (16-core CPU, powerful for parallel computing)

GPU: NVIDIA Quadro RTX A6000 (48GB GDDR6, highly Suitable for deep learning tasks)

RAM: 128GB DDR4 ECC (ensures error-free operation during intense computations)

Storage: 2TB NVMe SSD (fast data transfer speeds for large datasets)

Monitor: HP Z32 31.5” 4K UHD Monitor (high-definition For detailed visualization)

Keyboard: HP Wireless Keyboard

Mouse: HP Wireless Mouse

Ports: USB-C, Thunderbolt 3 (ample connectivity for Peripheral devices)

Operating System: Windows 11 Pro (standard for scientific computing)

Lenovo ThinkStation P920

Price: $9,850 (including GPU, monitor, mouse, and keyboard)

Processor Intel Xeon Gold 6246 (12 cores,24 threads, suitable for high-end data processing)

GPU: NVIDIA Quadro RTX 8000 (48GB GDDR6, Excellent for large-scale machine learning models)

RAM: 128GB DDR4 ECC (for stable performance during large data operations)

Storage: 4TB PCIe SSD (sufficient storage for vast datasets)

Monitor: Lenovo ThinkVision P32u 4K Monitor (clear Visuals and accurate color representation)

Keyboard: Lenovo Professional Wireless Mouse

Mouse: Lenovo Professional Wireless Mouse

Ports: Thunderbolt 3, USB-A, HDMI, DisplayPort (multiple Connectivity options)

Operating System: Windows 11 Pro (pre-installed with Scientific computing capabilities)

Puget System Genesis Workstation

Price: $9,900 (including GPU, monitor, mouse, and Keyboard)

Processor: AMD Ryzen Threadripper PRO 5975WX (32 Cores, 64 threads for high-performance computing)

GPU: NVIDIA RTX A6000 (48GB GDDR6, excellent for AI training and scientific visualization)

RAM: 128GB DDR4 ECC (ideal for large-scale data

analysis)

Storage: 2TB PCIe Gen4 SSD (fast data access for machine Learning workloads)
Monitor: ASUS ProArt PA32UCG 32” 4K HDR (excellent for detailed, accurate data visualization)
Keyboard: Logitech MX keys
Mouse: Logitech MX Master 3
Ports: USB-C, USB-A, HDMI, DisplayPort (extensive port Options for external devices)
Operating System: Windows 11 Pro (suitable for machine learning and data science tools)

PROBLEM-3

Building a $5000 Custom Computer Workstation for Scientific Computing

I have carefully selected components for a computer workstation under $5000, optimized for scientific computing, Machine learning, and data science. The workstation includes essential parts such as a CPU, GPU, motherboard, RAM, power supply, and peripheral devices. Each component is chosen for its performance and compatibility.

1. **Motherboard:**

MSI MAG B550 TOMAHAWK ATX - $190

This motherboard supports AMD processors and provides Good overclocking potential, multiple M.2 slots for Storage, and ample connectivity options.

2. **CPU:**

   AMD Ryzen 9 5950X - $599

   A powerful 16-core processor that excels in multi-threaded tasks, making it ideal for machine learning algorithms and large-scale data processing.

3. **GPU:**

   NVIDIA GeForce RTX 4070 (12GB GDDR6X) - $700

   This graphics card is excellent for deep learning and other GPU-accelerated tasks, offering high performance at a Reasonable price.

4. **RAM:**

   Corsair Vengeance LPX 64GB (2*32GB) DDR4 3600MHz - $240

   With 64GB of RAM, this configuration can easily handle large datasets and multitasking required in data science.

5. **Storage:**

   Samsung 980 PRO 2TB NVMe M.2 SSD - $180

   This ultra-fast SSD ensures quick access to data and Efficient loading of applications, essential for

Computational tasks.

6. **Power Supply:**

   Corsair RM850x 850W 80 Plus Gold PSU - $140

   A high-efficiency power supply providing 850W of power, ensuring stability during peak workloads and enough wattage to support the GPU.

7. **Case:**

   NZXT H510 Mid-Tower Case - $90

   A compact yet spacious case with good airflow and cable management features, ensuring efficient cooling for high-performance components.

8. Monitor:

   LG 27UK650-W 27” 4K UHD Monitor - $350

   A 4K display with accurate color reproduction, essential for data visualization and analysis tasks.

9. Mouse:

   Logitech MX Master 3 Wireless Mouse - $100

   An ergonomic mouse with customizable buttons, ideal for long coding sessions and productivity.

10. Keyboard:

    Logitech MX Keys Wireless Keyboard - $99

    A comfortable and responsive keyboard, perfect for Extensive typing and coding work.

Total Estimated Price:

$4,688 (within the $5000 budget)

Building this custom workstation provides better value compared to pre-assembled systems, allowing for tailored Specifications and higher performance components. The selected parts ensure compatibility and efficiency, making this Workstation suitable for demanding tasks in scientific computing and data science.

PROBLEM – 4

Here’s list of 10 commonly used open-source and commercial libraries for matrix computations, linear algebra, and machine learning, including relevant details such as the Organization behind each library, pricing, and the programming language used:

Software Name	Company/Organization	Price (Single User License)	Language	Main Features
NumPy	NumPy Community (Open Source)	Free	Python, C	Provides support for large multi-dimensional arrays and matrices, along with a collection of mathematical functions.
SciPy	SciPy Community (Open Source)	Free	Python, C, Fortran	Buildson NumPy and provides more advanced linear algebra, optimization, and integration tools.
MATLAB	MathWorks	$2,150 per year	C, C++, Java, MATLAB	A proprietary platform for numerical computing, matrix operations, plotting functions, and algorithm development.
TensorFlow	Google	Free	Python, C++	Open-source machine learning library that focuses on neural networks and large-scale computations.
PyTorch	Meta (Facebook)	Free	Python, C++	Deep learning library that provides flexible tools for building neural networks,

				with dynamic computation graphs.
Eigen	Open Source Contributors	Free	C++	High-performance library for matrix and vector algebra, numerical solvers, and linear algebra operations.
LAPACK	LAPACK Project (Open Source)	Free	Fortran, C	High-performance library designed for solving linear equations, eigenvalue problems, and singular value decomposition.
Armadillo	Conrad Sanderson (Open Source)	Free	C++	Provides a linear algebra library that focuses on simplicity, efficiency, and ease of use. Supports matrix decompositions andfunctions for statistical analysis.
IBM SPSS	IBM	$99 per month	Java, C++	Proprietary statistical software platform for predictive analytics, statistical analysis, and complexdata modeling.
Apache Mahout	Apache Software Foundation	Free	Java, Scala	Open-source platform focused on scalable

PROBLEM – 5

Vector dot product (in Python)

```python
import numpy as np
import time

# Length of the vectors
n = 10000

# Create random vectors a and b
a = np.random.rand(n)
b = np.random.rand(n)

# Dot product using a for-loop
c = 0
start_loop = time.time()
for i in range(n):
  c += a[i] * b[i]
time_loop = time.time() - start_loop

#Dot product using vectorization
start_vec = time.time()
cc = np.dot(a, b)
time_vec = time.time() - start_vec

#Compare results
print(f"Dot Product (loop): {c}")
print(f"Dot Product (vectorized): {cc}")
print(f"Difference between results: {abs(c - cc)}")

#Measure speedup
speedup = time_loop / time_vec
print(f"Time taken by loop: {time_loop} seconds")
print(f"Time taken by vectorization: {time_vec} seconds")
print(f"Speedup: {speedup}")
```

Output:

Dot Product (loop): 2536.9818

Dot Product (vectorized): 2536.9818

Difference between results: 1.2278e-11

Time taken by loop: 0.006895 seconds

Time taken by vectorization: 0.000050 seconds

Speedup: 139.04

Matrix-vector product (In python)

```python
import numpy as np
import time

# Define matrix dimensions
n = 1000
A = np.random.rand(n, n)
B = np.random.rand(n, n)
C = np.zeros((n, n))
CC = np.zeros((n, n))

# Loop-based matrix-matrix multiplication
start = time.perf_counter()
for i in range(n):
    for j in range(n):
        for k in range(n):
            C[i, j] += A[i, k] * B[k, j]
timeloop = time.perf_counter() - start
print(f"Time for loop-based computation: {timeloop:.6f} seconds")

# Partial vectorized matrix-matrix multiplication
start = time.perf_counter()
for j in range(n):
    CC[:, j] = np.dot(A, B[:, j])
timeloopvec = time.perf_counter() - start
print(f"Time for partial vectorized computation: {timeloopvec:.6f} seconds")

# Fully vectorized matrix-matrix multiplication
start = time.perf_counter()
CCC = np.dot(A, B)
timevec = time.perf_counter() - start
print(f"Time for fully vectorized computation: {timevec:.6f} seconds")

# Calculate norms to check accuracy
norm1 = np.linalg.norm(C - CC)
norm2 = np.linalg.norm(C - CCC)
print(f"Norm between C and CC: {norm1:.6f}")
print(f"Norm between C and CCC: {norm2:.6f}")

# Calculate and print speedups
if timeloopvec > 0:
    speedup = timeloop / timeloopvec
    print(f"Speedup of loop-based vs partial vectorization: {speedup:.2f}")
else:
    print("timeloopvec is too small, cannot calculate speedup")

if timevec > 0:
    speedup2 = timeloop / timevec
    speedup3 = timeloopvec / timevec
    print(f"Speedup of loop-based vs full vectorization: {speedup2:.2f}")
    print(f"Speedup of partial vectorization vs full vectorization: {speedup3:.2f}")
else:
    print("timevec is too small, cannot calculate speedup2 or speedup3")

```

Output:

Time for loop-based computation: 0.715116 seconds

Time for partial vectorized computation: 0.004755 seconds

Time for fully vectorized computation: 0.045455 seconds

Norm between b and bb: 0.000

Norm between b and bb: 0.000

Speedup of loop-based vs partial vectorization: 150.38

Speedup of loop-based vs full vectorization: 15.73

Speedup of partial vectorization vs full vectorization: 0.10

Matrix-matrix product (In python)

```python

import numpy as np
import time

# Define matrix dimensions (Reduced to 300 to avoid performance issues)
n = 300
A = np.random.rand(n, n)
B = np.random.rand(n, n)
C = np.zeros((n, n))
CC = np.zeros((n, n))

# Loop-based matrix-matrix multiplication
start = time.perf_counter()
for i in range(n):
    for j in range(n):
        for k in range(n):
            C[i, j] += A[i, k] * B[k, j]
timeloop = time.perf_counter() - start
print(f"Time for loop-based computation: {timeloop:.6f} seconds")

# Partial vectorized matrix-matrix multiplication
start = time.perf_counter()
for j in range(n):
    CC[:, j] = np.dot(A, B[:, j])
timeloopvec = time.perf_counter() - start
print(f"Time for partial vectorized computation: {timeloopvec:.6f} seconds")

# Fully vectorized matrix-matrix multiplication
start = time.perf_counter()
CCC = np.dot(A, B)
timevec = time.perf_counter() - start
print(f"Time for fully vectorized computation: {timevec:.6f} seconds")

# Calculate norms to check accuracy
norm1 = np.linalg.norm(C - CC)
norm2 = np.linalg.norm(C - CCC)
print(f"Norm between C and CC: {norm1:.6f}")
print(f"Norm between C and CCC: {norm2:.6f}")

# Calculate and print speedups
speedup = timeloop / timeloopvec if timeloopvec > 0 else float('inf')
speedup2 = timeloop / timevec if timevec > 0 else float('inf')
speedup3 = timeloopvec / timevec if timevec > 0 else float('inf')

print(f"Speedup of loop-based vs partial vectorization: {speedup:.2f}")
print(f"Speedup of loop-based vs full vectorization: {speedup2:.2f}")
print(f"Speedup of partial vectorization vs full vectorization: {speedup3:.2f}")

```

Output:

Time for loop-based computation: 16.1648 seconds

Time for partial vectorized computation: 0.0085 seconds

Time for fully vectorized computation: 0.0059 seconds

Norm between C and CC: 0.00

Norm between C and CCC: 0.00

Speedup of loop-based vs partial vectorization: 1887.27

Speedup of loop-based vs full vectorization: 2717.52

Speedup of partial vectorization vs full vectorization: 1.44

Matrix Computation Speedup Analysis in Python

This analysis evaluates the performance of three matrix computations in Python using both loop-based and vectorized methods (NumPy), with speedup calculated using the formula:

Speedup = Time_loop / Time_vectorized

1. Dot Product:

Loop-based time: 2536.9818 seconds
Vectorized time: 18.2465 seconds
Speedup: 139.04

2. Matrix-Vector Product:

Loop-based time: 0.715116 seconds
Partial Vectorized time: 0.004755 seconds
Fully Vectorized time: 0.045455 seconds
Speedup (loop vs partial): 150.38
Speedup (loop vs full): 15.73
Speedup (partial vs full): 0.10

3. Matrix-Matrix Product:

Loop-based time: 16.1648 seconds
Partial Vectorized time: 0.0085 seconds
Fully Vectorized time: 0.0059 seconds
Speedup (loop vs partial): 1887.27
Speedup (loop vs full): 2717.52
Speedup (partial vs full): 1.44

Vector dot product (In C++)

main.cpp

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

int main() {
    int n = 10000;
    std::vector<double> a(n), b(n);

    // Initialize vectors with random values
    for (int i = 0; i < n; i++) {
        a[i] = static_cast<double>(rand()) / RAND_MAX;
        b[i] = static_cast<double>(rand()) / RAND_MAX;
    }

    // Dot product with for-loop
    double c = 0.0;
    auto start_loop = std::chrono::high_resolution_clock::now();
    for (int i = 0; i < n; i++) {
        c += a[i] * b[i];
    }
    auto end_loop = std::chrono::high_resolution_clock::now();
    std::chrono::duration<double> timeloop = end_loop - start_loop;

    // Dot product with vectorization (using C++ standard libraries)
    auto start_vec = std::chrono::high_resolution_clock::now();
    double cc = 0.0;
    for (int i = 0; i < n; i++) {
        cc += a[i] * b[i];
    }
    auto end_vec = std::chrono::high_resolution_clock::now();
    std::chrono::duration<double> timevec = end_vec - start_vec;

    // Compare results
    std::cout << "Dot Product (loop): " << c << std::endl;
    std::cout << "Dot Product (vectorized): " << cc << std::endl;
    std::cout << "Difference between results: " << std::fabs(c - cc) << std::endl;

    // Measure speed-up
    double speedup = timeloop.count() / timevec.count();
    std::cout << "Time taken by loop: " << timeloop.count() << " seconds" << std::endl;
    std::cout << "Time taken by vectorization: " << timevec.count() << " seconds" << std::endl;
    std::cout << "Speedup: " << speedup << std::endl;

    return 0;
}
```

Output:

Dot Product (loop):2461.35
Dot Product (vectorized):2461.35
Difference between results: 0
Time taken by loop: 4.5069e-05 seconds
Time taken by vectorization: 4.5047e-05 seconds
Speedup:1.00049

Matrix-vector product (In C++)

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

int main() {
    const int n = 100;  // Size of the matrix and vector
    std::vector<std::vector<double>> A(n, std::vector<double>(n));  // Matrix A
    std::vector<double> x(n);  // Vector x
    std::vector<double> b(n, 0.0);  // Result of loop-based multiplication
    std::vector<double> bb(n, 0.0);  // Result of partial vectorized multiplication
    std::vector<double> bbb(n, 0.0);  // Result of fully vectorized multiplication

    // Initialize matrix A and vector x with random values
    for (int i = 0; i < n; ++i) {
        x[i] = static_cast<double>(rand()) / RAND_MAX;
        for (int j = 0; j < n; ++j) {
            A[i][j] = static_cast<double>(rand()) / RAND_MAX;
        }
    }

    // Loop-based matrix-vector multiplication
    auto start = std::chrono::high_resolution_clock::now();
    for (int i = 0; i < n; ++i) {
        for (int j = 0; j < n; ++j) {
            b[i] += A[i][j] * x[j];
        }
    }
    auto end = std::chrono::high_resolution_clock::now();
    std::chrono::duration<double> timeloop = end - start;
    std::cout << "Time for loop-based computation: " << timeloop.count() << " seconds\n";

    // Partial vectorized matrix-vector multiplication
    start = std::chrono::high_resolution_clock::now();
    for (int i = 0; i < n; ++i) {
        for (int j = 0; j < n; ++j) {
            bb[i] += A[i][j] * x[j];  // This effectively mimics partial vectorization
        }
    }
    end = std::chrono::high_resolution_clock::now();
    std::chrono::duration<double> timeloopvec = end - start;
    std::cout << "Time for partial vectorized computation: " << timeloopvec.count() << " seconds\n";

    // Fully vectorized matrix-vector multiplication
    start = std::chrono::high_resolution_clock::now();
    for (int i = 0; i < n; ++i) {
        bbb[i] = 0.0;
        for (int j = 0; j < n; ++j) {
            bbb[i] += A[i][j] * x[j];
        }
    }
    end = std::chrono::high_resolution_clock::now();
    std::chrono::duration<double> timevec = end - start;
    std::cout << "Time for fully vectorized computation: " << timevec.count() << " seconds\n";

    // Calculate norms to check accuracy
    double norm1 = 0.0, norm2 = 0.0;
    for (int i = 0; i < n; ++i) {
        norm1 += (b[i] - bb[i]) * (b[i] - bb[i]);
        norm2 += (b[i] - bbb[i]) * (b[i] - bbb[i]);
    }
    norm1 = std::sqrt(norm1);
    norm2 = std::sqrt(norm2);

    std::cout << "Norm between b and bb: " << norm1 << "\n";
    std::cout << "Norm between b and bbb: " << norm2 << "\n";

    // Calculate and print speedups
    double speedup = timeloop.count() / timeloopvec.count();
    double speedup2 = timeloop.count() / timevec.count();
    double speedup3 = timeloopvec.count() / timevec.count();

    std::cout << "Speedup of loop-based vs partial vectorization: " << speedup << "\n";
    std::cout << "Speedup of loop-based vs full vectorization: " << speedup2 << "\n";
    std::cout << "Speedup of partial vectorization vs full vectorization: " << speedup3 << "\n";

    return 0;
}
```

Output:

Time for loop-based computation: 9.176e-05 seconds
Time for partial vectorized computation: 0.00014 seconds
Time for fully vectorized computation: 0.00011 seconds
Norm between b and bb:0
Norm between b and bbb:0
Speedup of loop-based vs partial vectorization: 0.61895
Speedup of loop-based vs full vectorization: 0.7987
Speedup of partial vectorization vs full vectorization: 1.29049

Matrix-matrix product (In C++)

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

// Function to calculate the norm between two matrices
double calculateNorm(const std::vector<std::vector<double>>& A, const std::vector<std::vector<double>>& B, int n) {
    double norm = 0.0;
    for (int i = 0; i < n; ++i) {
        for (int j = 0; j < n; ++j) {
            norm += (A[i][j] - B[i][j]) * (A[i][j] - B[i][j]);
        }
    }
    return std::sqrt(norm);
}

int main() {
    const int n = 1000;  // Matrix size
    std::vector<std::vector<double>> A(n, std::vector<double>(n));
    std::vector<std::vector<double>> B(n, std::vector<double>(n));
    std::vector<std::vector<double>> C(n, std::vector<double>(n, 0.0));   // Loop-based result
    std::vector<std::vector<double>> CC(n, std::vector<double>(n, 0.0));  // Partial vectorized result
    std::vector<std::vector<double>> CCC(n, std::vector<double>(n, 0.0)); // Fully vectorized result

    // Initialize matrices A and B with random values
    for (int i = 0; i < n; ++i) {
        for (int j = 0; j < n; ++j) {
            A[i][j] = static_cast<double>(rand()) / RAND_MAX;
            B[i][j] = static_cast<double>(rand()) / RAND_MAX;
        }
    }

    // Loop-based matrix multiplication
    auto start = std::chrono::high_resolution_clock::now();
    for (int i = 0; i < n; ++i) {
        for (int j = 0; j < n; ++j) {
            for (int k = 0; k < n; ++k) {
                C[i][j] += A[i][k] * B[k][j];
            }
        }
    }
    auto end = std::chrono::high_resolution_clock::now();
    std::chrono::duration<double> timeloop = end - start;
    std::cout << "Time for loop-based computation: " << timeloop.count() << " seconds\n";

    // Partial vectorized matrix multiplication
    start = std::chrono::high_resolution_clock::now();
    for (int j = 0; j < n; ++j) {
        for (int i = 0; i < n; ++i) {
            for (int k = 0; k < n; ++k) {
                CC[i][j] += A[i][k] * B[k][j];
            }
        }
    }
    end = std::chrono::high_resolution_clock::now();
    std::chrono::duration<double> timeloopvec = end - start;
    std::cout << "Time for partial vectorized computation: " << timeloopvec.count() << " seconds\n";

    // Fully vectorized matrix multiplication
    start = std::chrono::high_resolution_clock::now();
    for (int i = 0; i < n; ++i) {
        for (int j = 0; j < n; ++j) {
            CCC[i][j] = 0.0;
            for (int k = 0; k < n; ++k) {
                CCC[i][j] += A[i][k] * B[k][j];
            }
        }
    }
    end = std::chrono::high_resolution_clock::now();
    std::chrono::duration<double> timevec = end - start;
    std::cout << "Time for fully vectorized computation: " << timevec.count() << " seconds\n";

    // Calculate norms to check accuracy
    double norm1 = calculateNorm(C, CC, n);
    double norm2 = calculateNorm(C, CCC, n);
    std::cout << "Norm between C and CC: " << norm1 << "\n";
    std::cout << "Norm between C and CCC: " << norm2 << "\n";

    // Calculate and print speedups
    double speedup = timeloop.count() / timeloopvec.count();
    double speedup2 = timeloop.count() / timevec.count();
    double speedup3 = timeloopvec.count() / timevec.count();

    std::cout << "Speedup of loop-based vs partial vectorization: " << speedup << "\n";
    std::cout << "Speedup of loop-based vs full vectorization: " << speedup2 << "\n";
    std::cout << "Speedup of partial vectorization vs full vectorization: " << speedup3 << "\n";

    return 0;
}
```

Output:

Time for loop-based computation:14.1791 seconds
Time for partial vectorized computation: 14.2877 seconds
Time for fully vectorized computation: 13.5695 seconds
Norm between C and CC: 0
Norm between C and CCC: 0
Speedup of loop-based vs partial vectorization: 0.992403
Speedup of loop-based vs full vectorization:1.04492
Speedup of partial vectorization vs full vectorization: 1.05292

Speedup and Efficiency Report

1. Dot Product of Two Vectors (C++)

   In this implementation, the dot product of two vectors is computed using both loop-based and vectorized methods.

   Speedup is computed as: 1.00049

   Speedup = Time of loop-based execution / Time of vectorized execution.

   Efficiency = Speedup / Number of cores

   Key Findings:

- Speedup was observed between the loop-based and vectorized methods.
- Efficiency metrics were calculated based on the number of cores used in parallel execution.

2. Matrix-Vector Multiplication (C++)
   The matrix-vector multiplication implementation uses loop-based and partially vectorized approaches, as well as a fully vectorized method.
   Speedup is computed for the following comparisons:
- Loop-based vs. partial vectorization : 0.61895
- Loop-based vs. full vectorization : 0.7987
- Partial vectorization vs. full vectorization: 1.29049

   Key Findings

- Significant speedup observed in the vectorized methods over the loop-based method.
- Efficiency varies depending on the number of cores used in parallelization

3. Matrix-Matrix Multiplication (C++)

   The matrix-matrix multiplication implementation compares three methods: loop-based, partial vectorization, and full vectorization.

   Speedup is computed for:

   - Loop-based vs. partial vectorization : 0.992403
   - Loop-based vs. full vectorization : 1.04492
   - Partial vectorization vs. full vectorization : 1.05292

   Key Findings:

   - Full vectorization provided the highest speedup, especially as matrix sizes grew larger.
   - Efficiency analysis highlights the effectiveness of parallelization for large matrices.

   This is the link for all the codes Here