Experiment 4:

```
!nvcc --version
nvcc: NVIDIA (R) Cuda compiler driver
Copyright (c) 2005-2022 NVIDIA Corporation
Built on Wed_Sep_21_10:33:58_PDT_2022
Cuda compilation tools, release 11.8, V11.8.89
Build cuda_11.8.r11.8/compiler.31833905_0
```

Code : MatrixMul.cu

```
%%writefile matrixMul.cu
#include <cmath>
#include <cstdlib>
#include <iostream>
using namespace std;

// Matrix multiplication Cuda
__global__ void matrixMultiplication(int *a, int *b, int *c, int n) {
    int row = threadIdx.y + blockDim.y * blockIdx.y;
    int col = threadIdx.x + blockDim.x * blockIdx.x;
    int sum = 0;

    if (row < n && col < n)
        for (int j = 0; j < n; j++) {
            sum = sum + a[row * n + j] * b[j * n + col];
        }

    c[n * row + col] = sum;
}
int main() {
    int *a, *b, *c;
    int *a_dev, *b_dev, *c_dev;
    int n = 3;

    a = new int[n * n];
    b = new int[n * n];
    c = new int[n * n];
    int *d = new int[n * n];
    int size = n * n * sizeof(int);
    cudaMalloc(&a_dev, size);
    cudaMalloc(&b_dev, size);
    cudaMalloc(&c_dev, size);

    // Array initialization
    for (int i = 0; i < n * n; i++) {
        a[i] = 2;  // rand()%n;
        b[i] = 1;  // rand()%n;
        // d[i]=a[i]+b[i];
    }
    cudaEvent_t start, end;

    cudaEventCreate(&start);
    cudaEventCreate(&end);

    cudaMemcpy(a_dev, a, size, cudaMemcpyHostToDevice);
    cudaMemcpy(b_dev, b, size, cudaMemcpyHostToDevice);

    dim3 threadsPerBlock(n, n);
    dim3 blocksPerGrid(1, 1);

    if (n * n > 512) {
        threadsPerBlock.x = 512;
        threadsPerBlock.y = 512;
        blocksPerGrid.x = ceil((double)n / (double)threadsPerBlock.x);
        blocksPerGrid.y = ceil((double)n / (double)threadsPerBlock.y);
    }
    // GPU Multiplication
    cudaEventRecord(start);
    matrixMultiplication<<<blocksPerGrid, threadsPerBlock>>>(a_dev,
b_dev, c_dev, n);

    cudaEventRecord(end);
    cudaEventSynchronize(end);

    float time = 0.0;
    cudaEventElapsedTime(&time, start, end);

    cudaMemcpy(c, c_dev, size, cudaMemcpyDeviceToHost);

    // CPU matrix multiplication
    int sum = 0;
    for (int row = 0; row < n; row++) {
        for (int col = 0; col < n; col++) {
            sum = 0;
            for (int k = 0; k < n; k++) sum = sum + a[row * n + k] *
b[k * n + col];
            d[row * n + col] = sum;
        }
    }
    int error = 0;
    for (int i = 0; i < n * n; i++) {
        error += d[i] - c[i];
        // cout<<" gpu "<<c[i]<<" CPU "<<d[i]<<endl;
    }

    cout << "Error : " << error;
    cout << "\nTime Elapsed:  " << time;

    return 0;
}
```

Output:

```
!nvcc -dc matrixMul.cu
!nvcc *.o -o ./matrixMul && ./matrixMul
!rm -rf *.o
Error : 0
Time Elapsed:  0.02608
```

Code: matrixVecMul.cu

```
%%writefile matrixVecMul.cu
#include <time.h>
#include <cmath>
#include <cstdlib>
#include <iostream>
using namespace std;

__global__ void matrixVectorMultiplication(int *a, int *b, int *c, int
n) {
    int row = threadIdx.x + blockDim.x * blockIdx.x;
    int sum = 0;

    if (row < n)
        for (int j = 0; j < n; j++) {
            sum = sum + a[row * n + j] * b[j];
        }

    c[row] = sum;
}
int main() {
    int *a, *b, *c;
    int *a_dev, *b_dev, *c_dev;
    int n = 32;

    a = new int[n * n];
    b = new int[n];
    c = new int[n];
    int *d = new int[n];
    int size = n * sizeof(int);
    cudaMalloc(&a_dev, size * size);
    cudaMalloc(&b_dev, size);
    cudaMalloc(&c_dev, size);

    for (int i = 0; i < n; i++) {
        for (int j = 0; j < n; j++) {
            a[i * n + j] = i * n + j + 1;  // rand()%n;
        }

        b[i] = i + 1;  // rand()%n;
        // cout<<a[i]<<" ";
        // d[i]=a[i]+b[i];
    }

    cudaEvent_t start, end;

    cudaEventCreate(&start);
    cudaEventCreate(&end);

    cudaMemcpy(a_dev, a, size * size, cudaMemcpyHostToDevice);
    cudaMemcpy(b_dev, b, size, cudaMemcpyHostToDevice);

    dim3 threadsPerBlock(n, n);
    dim3 blocksPerGrid(1, 1);

    if (n * n > 512) {
        threadsPerBlock.x = 512;
        threadsPerBlock.y = 512;
        blocksPerGrid.x = ceil((double)n / (double)threadsPerBlock.x);
        blocksPerGrid.y = ceil((double)n / (double)threadsPerBlock.y);
    }

    cudaEventRecord(start);
    matrixVectorMultiplication<<<blocksPerGrid,
threadsPerBlock>>>(a_dev, b_dev, c_dev, n);

    cudaEventRecord(end);
    cudaEventSynchronize(end);

    float time = 0.0;
    cudaEventElapsedTime(&time, start, end);

    cudaMemcpy(c, c_dev, size, cudaMemcpyDeviceToHost);
    cout << "\nGPU Time Elapsed:  " << time;

    // CPU matrixVector multiplication
    clock_t t = clock();
    int sum = 0;
    for (int row = 0; row < n; row++) {
        sum = 0;
        for (int col = 0; col < n; col++) {
            sum = sum + a[row * n + col] * b[col];
        }
        d[row] = sum;
    }
    t = clock() - t;
    cout << "\nCPU Time Elapsed:  " <<
((double)t);  //((double)t)/CLOCKS_PER_SEC;

    int error = 0;
    for (int i = 0; i < n; i++) {
        error += d[i] - c[i];
        // cout<<" gpu "<<c[i]<<" CPU "<<d[i]<<endl;
    }

    cout << "Error : " << error;

    return 0;
}
```

Output:

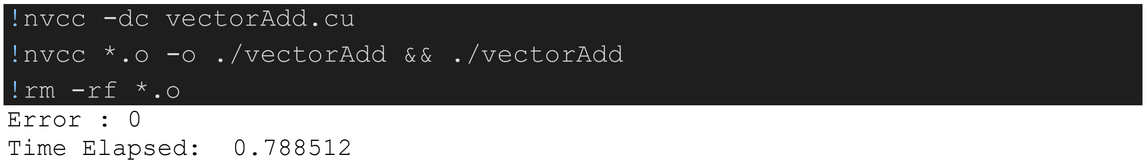
```
!nvcc -dc matrixVecMul.cu
!nvcc *.o -o ./matrixVecMul && ./matrixVecMul
!rm -rf *.o
```

```
GPU Time Elapsed:  0.004096
CPU Time Elapsed:  6Error : 8746496
```

Code : vectorAdd.cu

```
%%writefile vectorAdd.cu
#include <cstdlib>
#include <iostream>

using namespace std;

// VectorAdd parallel function
__global__ void vectorAdd(int *a, int *b, int *result, int n) {
    int tid = threadIdx.x + blockIdx.x * blockDim.x;
    if (tid < n) {
        result[tid] = a[tid] + b[tid];
    }
}
int main() {
    int *a, *b, *c;
    int *a_dev, *b_dev, *c_dev;
    int n = 1 << 24;

    a = new int[n];
    b = new int[n];
    c = new int[n];
    int *d = new int[n];
    int size = n * sizeof(int);
    cudaMalloc(&a_dev, size);
    cudaMalloc(&b_dev, size);
    cudaMalloc(&c_dev, size);

    // Array initialization..You can use Randon function to assign
values
    for (int i = 0; i < n; i++) {
        a[i] = 1;
        b[i] = 2;
        d[i] = a[i] + b[i];  // calculating serial addition
    }

    cudaEvent_t start, end;

    cudaEventCreate(&start);
    cudaEventCreate(&end);

    cudaMemcpy(a_dev, a, size, cudaMemcpyHostToDevice);
    cudaMemcpy(b_dev, b, size, cudaMemcpyHostToDevice);
    int threads = 1024;
    int blocks = (n + threads - 1) / threads;
    cudaEventRecord(start);

    // Parallel addition program
    vectorAdd<<<blocks, threads>>>(a_dev, b_dev, c_dev, n);

    cudaEventRecord(end);
    cudaEventSynchronize(end);

    float time = 0.0;
    cudaEventElapsedTime(&time, start, end);

    cudaMemcpy(c, c_dev, size, cudaMemcpyDeviceToHost);

    // Calculate the error term.
    int error = 0;
    for (int i = 0; i < n; i++) {
        error += d[i] - c[i];
        // cout<<" gpu "<<c[i]<<" CPU "<<d[i];
    }

    cout << "Error : " << error;
    cout << "\nTime Elapsed:  " << time;

    return 0;
}
```

Output:



```
!nvcc -dc vectorAdd.cu
!nvcc *.o -o ./vectorAdd && ./vectorAdd
!rm -rf *.o
Error : 0
Time Elapsed:  0.788512
```