Report for HPC LAB

Name: Bhaskar R
Roll No: CED18I009
Programming Environment: CUDA (Google Colab)
Problem: Vector Multiplication
Date: 17th November 2021

Hardware Configuration:

CPU NAME : Intel(R) Xeon(R) CPU @ 2.30GHz
RAM : 12.69 GB

Serial Code:

```
#include <bits/stdc++.h>
using namespace std;
#define N 1024
int main()
{
   srand(time(0));
   int a[N], b[N], c[N];
   for (int i = 0; i < N; i++)
   {
       a[i] = rand() % 100 + i;
       b[i] = rand() % 100 * i;
       c[i] = a[i] * b[i];
   }

   for (auto i : c)
       cout << i << endl;
   return 0;
}
```

Parallel Code :

```
%%cu
#include <bits/stdc++.h>
using namespace std;
#define N 25
#define M 1024

__global__ void vector_mul(double *a, double *b, double *c)
{
    int id = blockIdx.x * blockDim.x + threadIdx.x;
    if (id < N)
        c[id] = a[id] * b[id];
}

int main()
{
    srand(time(0));
    int blocks[] = {1, 1, 1, 1, 1, 1, 1, 10, 20, 30, 40, 50, M / 2, M /
4, M / 8, M, M, M, M, M};
    int threads[] = {1, 10, 20, 30, 40, 50, M, 10, 10, 10, 10, 10, M, M,
M, M / 2, M / 4, M / 8, M};
    double a[N], b[N], c[N];
    double *d_a, *d_b, *d_c;
    double size = N * sizeof(double);

    // Allocate space for device copies of a, b, c
    cudaMalloc((void **)&d_a, size);
    cudaMalloc((void **)&d_b, size);
    cudaMalloc((void **)&d_c, size);

    for (int i = 0; i < N; i++)
    {
        a[i] = rand() % 100 + i + 0.250;
        b[i] = rand() % 100 * i + 0.248;
    }

    cudaMemcpy(d_a, &a, size, cudaMemcpyHostToDevice);
    cudaMemcpy(d_b, &b, size, cudaMemcpyHostToDevice);
```

```
    for (int k = 0; k < 19; k++)
    {
        float elapsed = 0;
        cudaEvent_t start, stop;
        cudaEventCreate(&start);
        cudaEventCreate(&stop);

        cudaEventRecord(start, 0);
        vector_mul<<<blocks[k], threads[k]>>>(d_a, d_b, d_c);

        // Copy result back to host
        cudaError err = cudaMemcpy(&c, d_c, size,
cudaMemcpyDeviceToHost);
        if (err != cudaSuccess)
            cout << "CUDA Error copying to Host :" <<
cudaGetErrorString(err) << endl;

        cudaEventRecord(stop, 0);
        cudaEventSynchronize(stop);

        cudaEventElapsedTime(&elapsed, start, stop);

        cudaEventDestroy(start);
        cudaEventDestroy(stop);

        printf("Blocks = %4d and Threads per Block = %4d Time = %.5f\n",
blocks[k], threads[k], elapsed);
    }
    cout << "\nProduct of Vectors " << endl;
    for (int i = 0; i < N; i++)
        cout << a[i] << " * " << b[i] << " = " << c[i] << endl;

    // Cleanup
    cudaFree(d_a);
    cudaFree(d_b);
    cudaFree(d_c);
    return 0;
}
```

Output :

```
Blocks =    1 and Threads per Block =    1 Time = 0.20787
Blocks =    1 and Threads per Block =   10 Time = 0.02902
Blocks =    1 and Threads per Block =   20 Time = 0.03014
Blocks =    1 and Threads per Block =   30 Time = 0.02794
Blocks =    1 and Threads per Block =   40 Time = 0.03050
Blocks =    1 and Threads per Block =   50 Time = 0.03357
Blocks =    1 and Threads per Block = 1024 Time = 0.03370
Blocks =   10 and Threads per Block =   10 Time = 0.03389
Blocks =   20 and Threads per Block =   10 Time = 0.02954
Blocks =   30 and Threads per Block =   10 Time = 0.02973
Blocks =   40 and Threads per Block =   10 Time = 0.02966
Blocks =   50 and Threads per Block =   10 Time = 0.03325
Blocks =  512 and Threads per Block = 1024 Time = 0.03936
Blocks =  256 and Threads per Block = 1024 Time = 0.03094
Blocks =  128 and Threads per Block = 1024 Time = 0.03014
Blocks = 1024 and Threads per Block =  512 Time = 0.04256
Blocks = 1024 and Threads per Block =  256 Time = 0.03722
Blocks = 1024 and Threads per Block =  128 Time = 0.04090
Blocks = 1024 and Threads per Block = 1024 Time = 0.04982

Product of Vectors
6.25 * 0.248 = 1.55
66.25 * 72.248 = 4786.43
66.25 * 134.248 = 8893.93
78.25 * 132.248 = 10348.4
69.25 * 272.248 = 18853.2
27.25 * 175.248 = 4775.51
29.25 * 522.248 = 15275.8
101.25 * 175.248 = 17743.9
91.25 * 328.248 = 29952.6
101.25 * 108.248 = 10960.1
14.25 * 240.248 = 3423.53
26.25 * 0.248 = 6.51
22.25 * 1128.25 = 25103.5
37.25 * 208.248 = 7757.24
25.25 * 1078.25 = 27225.8
78.25 * 270.248 = 21146.9
99.25 * 448.248 = 44488.6
59.25 * 799.248 = 47355.4
113.25 * 324.248 = 36721.1
62.25 * 228.248 = 14208.4
58.25 * 1300.25 = 75739.4
69.25 * 273.248 = 18922.4
74.25 * 2068.25 = 153567
61.25 * 805.248 = 49321.4
60.25 * 1968.25 = 118587
```

Observations:

Number of Blocks	Threads per Block	Execution Time	Speed-up	Parallelization Fraction
1	1	0.207	1.0	
1	10	0.029	7.1379	95.5448
1	20	0.030	6.9	90.0076
1	30	0.027	7.6667	89.9551
1	40	0.030	6.9	87.6997
1	50	0.033	6.2727	85.7734
1	1024	0.033	6.2727	84.1401
10	10	0.033	6.2727	93.3977
20	10	0.029	7.1379	95.5448
30	10	0.029	7.1379	95.5448
40	10	0.029	7.1379	95.5448
50	10	0.033	6.2727	93.3977
512	1024	0.039	5.3077	81.2388
256	1024	0.030	6.9	85.5908
128	1024	0.030	6.9	85.5908
1024	512	0.042	4.9286	79.8663
1024	256	0.037	5.5946	82.4477
1024	128	0.040	5.175	81.3116
1024	1024	0.049	4.2245	76.4032

Speed up can be found using the following formula,

S(n)=T(1)/T(n)

where, S(n) = Speedup for thread count ‘n’

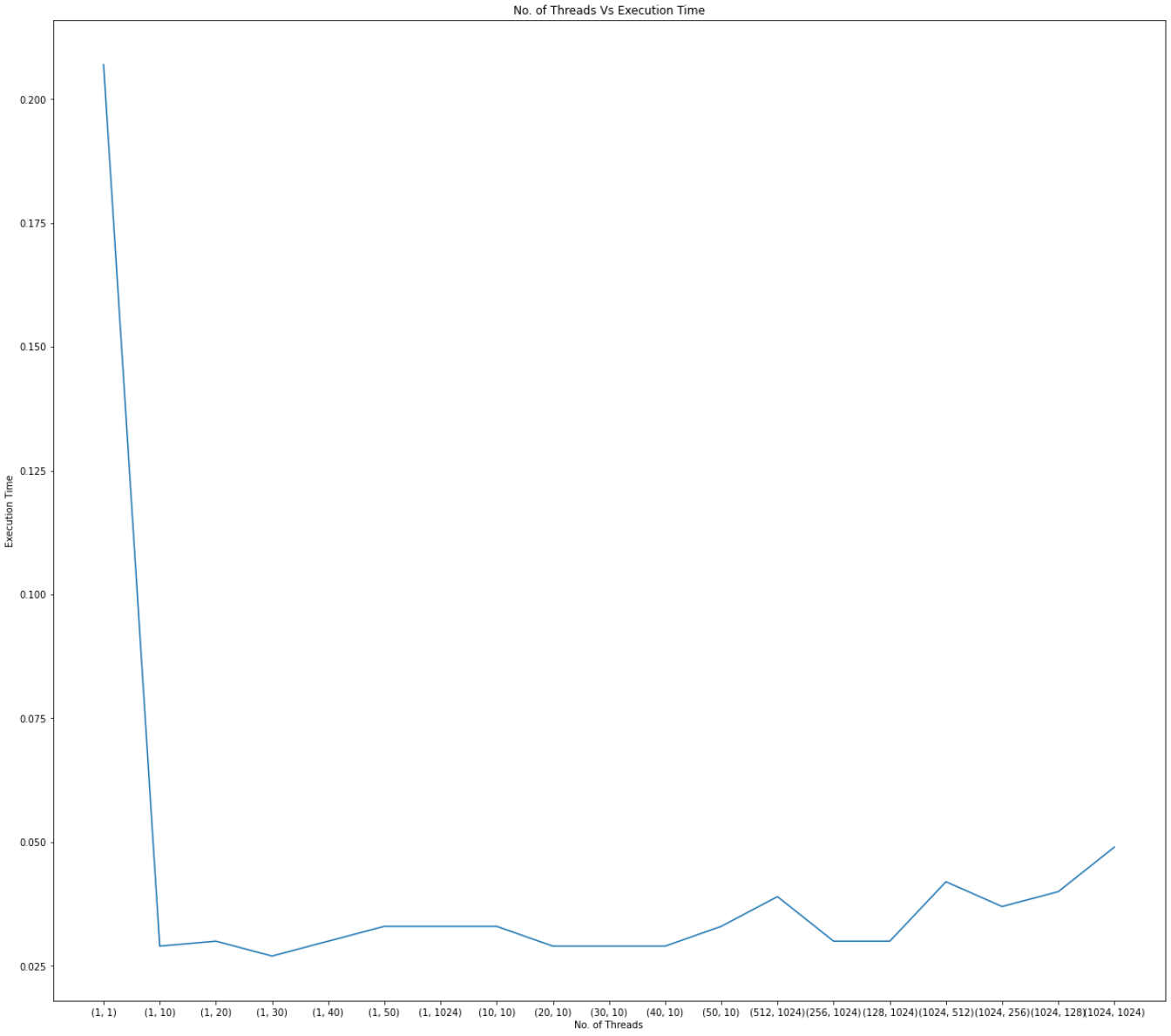
T(1) = Execution Time for Thread count ‘1’ (serial code)

T(n) = Execution Time for Thread count ‘n’ (serial code)

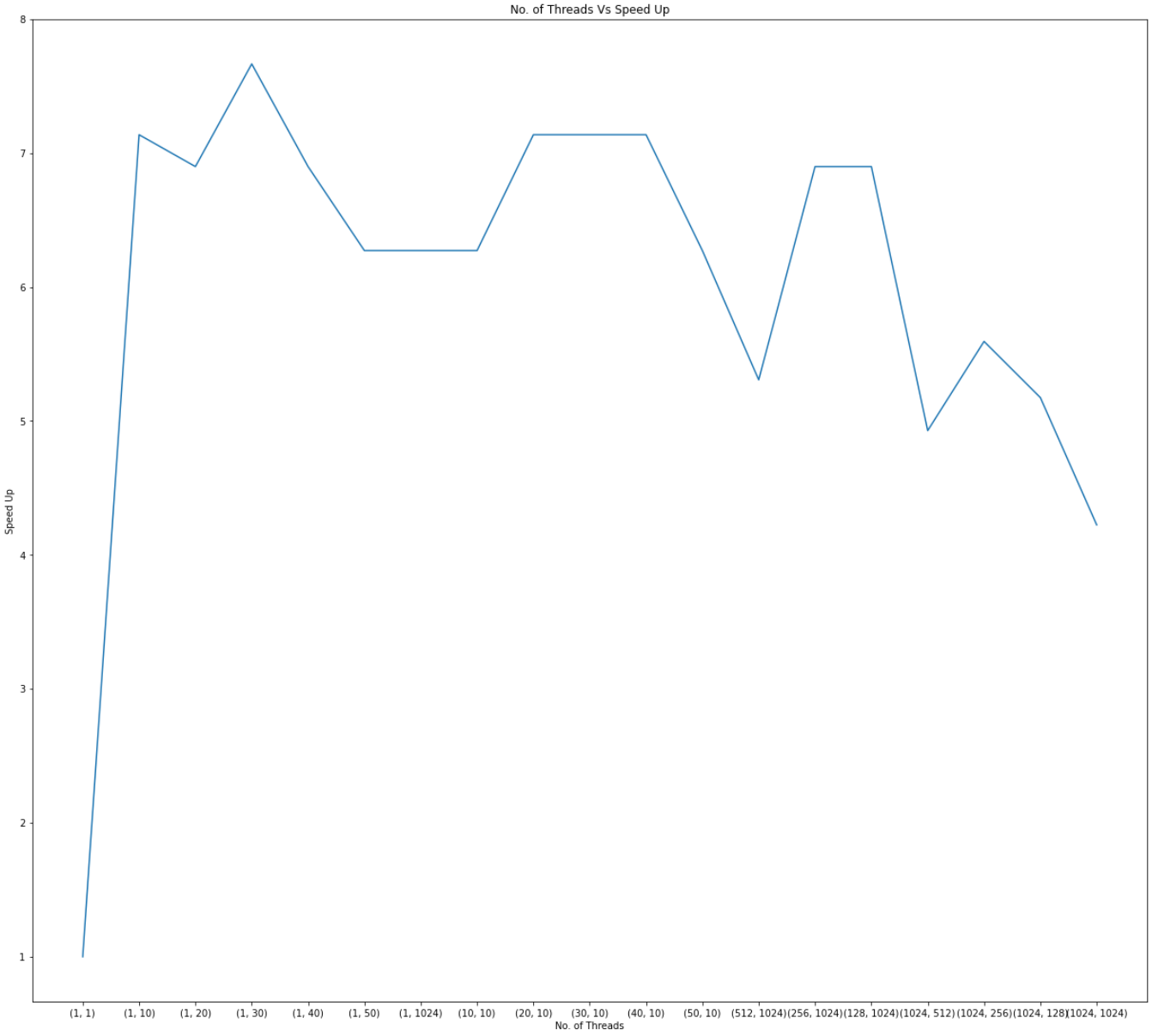
Parallelization Fraction can be found using the following formula, **S(n)=1/((1 - p) + p/n)**

where, S(n) = Speedup for thread count ‘n’
n = Number of threads
p = Parallelization fraction

No. of Threads Vs Execution Time



No. of Threads Vs Speed Up



Inference:

- For (1,1) the execution time is maximum, i.e poor performance. This is because there is no parallel execution.
- The Striding technique was used in the vector_mul function for the different combinations of no. of blocks and no. of threads.
- The Maximum speedup was for 1 number block with 30 threads per block combination. This is because it has reasonably fewer communication overheads and also a good amount of parallelization