Report for HPC LAB

Name: Bhaskar R
Roll No: CED18I009
Programming Environment: CUDA (Google Colab)
Problem: Vector Dot Product
Date: 20th 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;

int main()
{
   int i, n = 100000;
   double a[n], b[n], sum = 0;
   for (i = 0; i < n; i++)
   {
       a[i] = b[i] = i;
       sum += a[i] * b[i];
   }
}
```

NOTE : Here, single precision floating numbers are used as there is no support for double precision floating numbers in atomicAdd.

Parallel Code :

```
%%cu
#include <bits/stdc++.h>
using namespace std;
#define N 1500
#define M 1024

__global__ void dot_product(float *a, float *b, float *c)
{
   __shared__ float temp[M];

   int index = blockIdx.x * blockDim.x + threadIdx.x;
   if (index < N)
   {
       temp[threadIdx.x] = a[index] * b[index];
       __syncthreads();
       if (threadIdx.x == 0)
       {
           float sum = 0.0;
           if (blockIdx.x < N / blockDim.x)
           {
               for (int i = 0; i < (int)blockDim.x; i++)
                   sum += temp[i];
           }
           else
           {
               for (int i = 0; i < N % blockDim.x; i++)
                   sum += temp[i];
           }
           atomicAdd(c, sum);
       }
   }
}

int main()
```

```
{
    srand(time(0));
    int blocks[] = {1, 1, 1, 1, 1, 1, 1, 10, 20, 30, 40, 50, M / 8, M /
4, M / 2, M, M, M, M, M};
    int threads[] = {1, 10, 20, 30, 40, 50, M, 10, 10, 10, 10, 10, M,
M, M, M / 8, M / 4, M / 2, M};
    float a[N], b[N], c[N];
    float *d_a, *d_b, *d_c;
    for (int i = 0; i < N; i++)
    {
        a[i] = i + 0.250;
        b[i] = i + 0.248;
    }

    cudaMalloc((void **)&d_a, N * sizeof(float));
    cudaMalloc((void **)&d_b, N * sizeof(float));
    cudaMalloc((void **)&d_c, N * sizeof(float));

    cudaMemcpy(d_a, a, N * sizeof(float), cudaMemcpyHostToDevice);

    for (int k = 0; k < 19; k++)
    {
        float elapsed = 0;
        cudaEvent_t start, stop;
        cudaEventCreate(&start);
        cudaEventCreate(&stop);
        cudaEventRecord(start, 0);
        dot_product<<<blocks[k], threads[k]>>>(d_a, d_b, d_c);

        cudaError err = cudaMemcpy(c, d_c, N * sizeof(float),
cudaMemcpyDeviceToHost);
        if (err != cudaSuccess)
            cout << "CUDA Error copying to Host: " <<
cudaGetErrorString(err);
        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);
    }
    cudaFree(d_a);
    cudaFree(d_b);
    cudaFree(d_c);

    return 0;
}
```

Output :

```
Blocks =    1 and Threads per Block =    1 Time = 0.12976
Blocks =    1 and Threads per Block =   10 Time = 0.03354
Blocks =    1 and Threads per Block =   20 Time = 0.02992
Blocks =    1 and Threads per Block =   30 Time = 0.03069
Blocks =    1 and Threads per Block =   40 Time = 0.02986
Blocks =    1 and Threads per Block =   50 Time = 0.03283
Blocks =    1 and Threads per Block = 1024 Time = 0.05066
Blocks =   10 and Threads per Block =   10 Time = 0.03648
Blocks =   20 and Threads per Block =   10 Time = 0.03891
Blocks =   30 and Threads per Block =   10 Time = 0.03014
Blocks =   40 and Threads per Block =   10 Time = 0.03034
Blocks =   50 and Threads per Block =   10 Time = 0.03430
Blocks =  128 and Threads per Block = 1024 Time = 0.07501
Blocks =  256 and Threads per Block = 1024 Time = 0.05901
Blocks =  512 and Threads per Block = 1024 Time = 0.05024
Blocks = 1024 and Threads per Block =  128 Time = 0.03840
Blocks = 1024 and Threads per Block =  256 Time = 0.03853
Blocks = 1024 and Threads per Block =  512 Time = 0.04534
Blocks = 1024 and Threads per Block = 1024 Time = 0.05882
```

Observations:

Number of Blocks	Threads per Block	Execution Time	Speed-up	Parallelization Fraction
1	1	0.129	1.0	
1	10	0.033	3.9091	82.6874
1	20	0.029	4.4483	81.5995
1	30	0.030	4.3	79.3905
1	40	0.029	4.4483	79.5072
1	50	0.032	4.0312	76.7281
1	1024	0.050	2.58	61.3002
10	10	0.036	3.5833	80.1031
20	10	0.038	3.3947	78.3804
30	10	0.030	4.3	85.2713
40	10	0.030	4.3	85.2713
50	10	0.034	3.7941	81.8259
128	1024	0.075	1.72	41.9014
256	1024	0.059	2.1864	54.3158
512	1024	0.050	2.58	61.3002
1024	128	0.038	3.3947	71.0978
1024	256	0.038	3.3947	70.819
1024	512	0.045	2.8667	65.2441
1024	1024	0.058	2.2241	55.0918

Speed up can be found using the following formula,

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

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

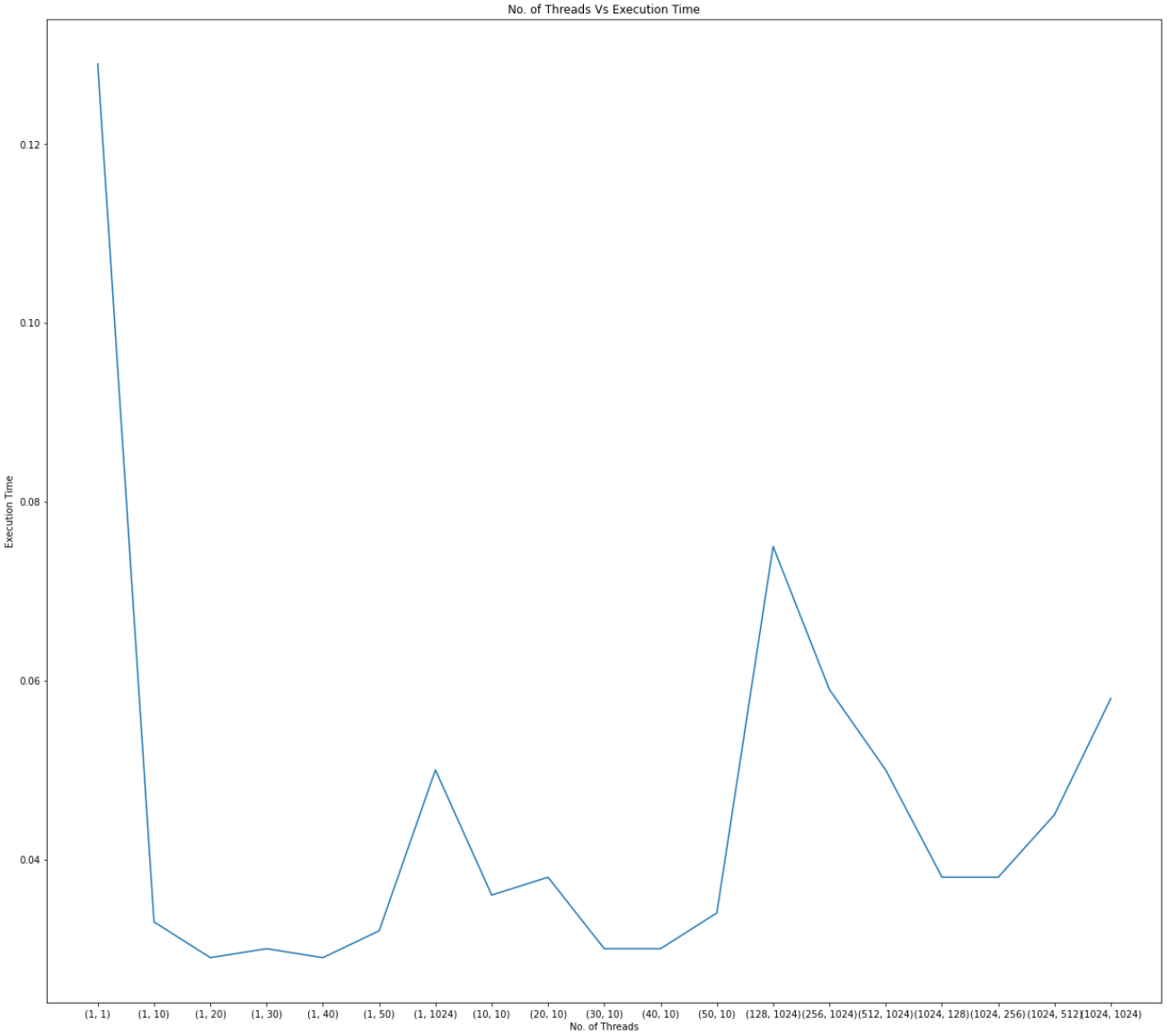
T(1) = Execution Time for Thread count ‘1’ (serial code)

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

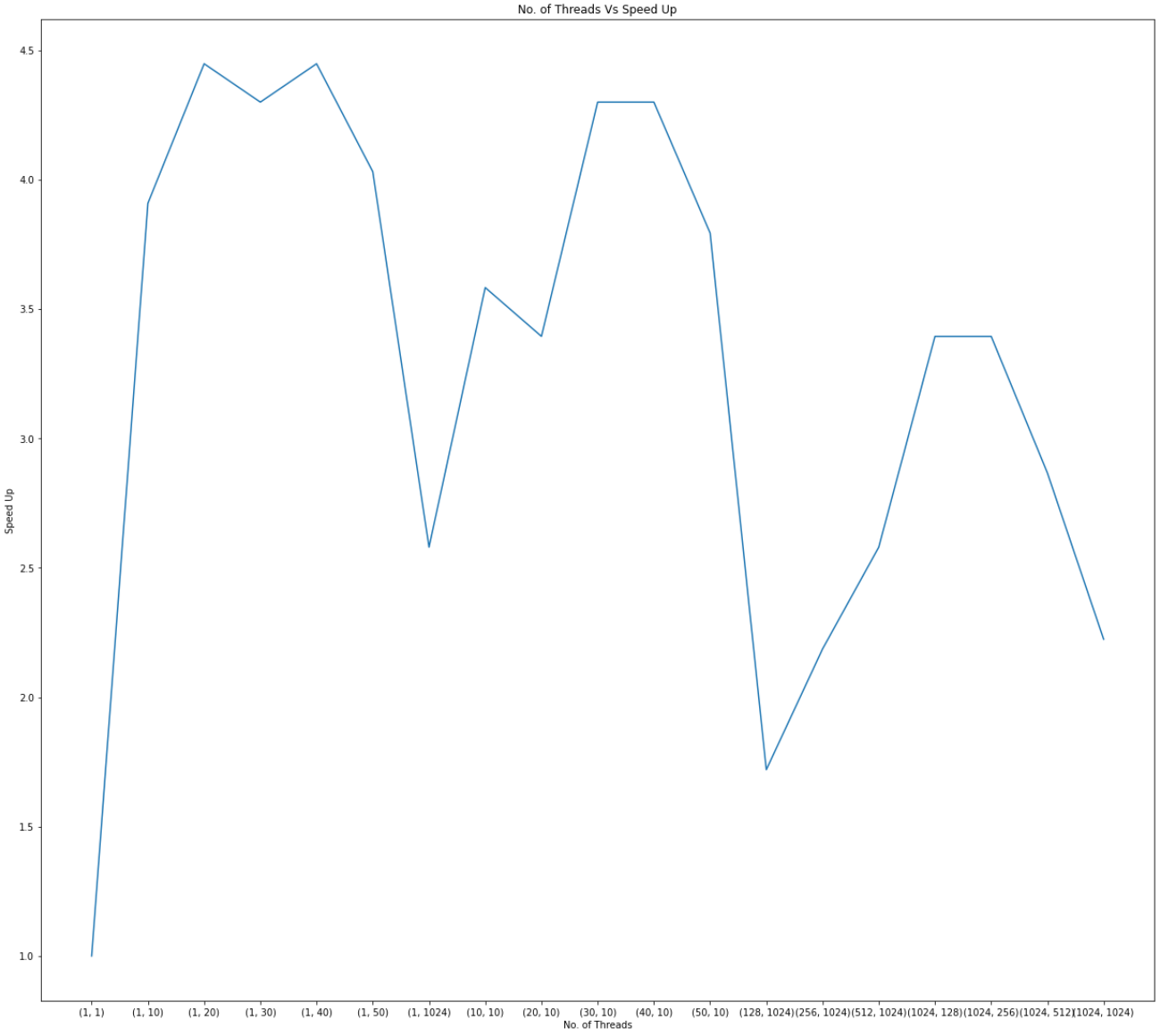
Parallelization Fraction can be found using the following formula, **S(n)=1/((1 - p) + p/n)**

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 dot_product function for the different combinations of no. of blocks and no. of threads.
- The Maximum speedup was for 1 number block with 40 threads per block combination. This is because it has reasonably fewer communication overheads and also a good amount of parallelization