Report for HPC LAB

Name: Bhaskar R
Roll No: CED18I009
Programming Environment: CUDA (Google Colab)
Problem: Sum of N Numbers
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 = 1, n = 100000;
   double a[n], sum = 0;
   for (i = 1; i < n; i++)
   {
       a[i] = i;
       sum += a[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 N_sum(float *a, float *b)
{
   __shared__ float temp[M];
   int index = blockIdx.x * blockDim.x + threadIdx.x;
   if (index < N)
   {
       temp[threadIdx.x] = a[index];
       __syncthreads();
       if (threadIdx.x == 0)
       {
           float sum = 0;
           for (int i = 0; i < M; i++)
               sum += temp[i];
           atomicAdd(b, 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] = {0.0};
   float *d_a, *d_b;
   for (int i = 0; i < N; i++)
       a[i] = i + 1.2564;
   cudaMalloc((void **)&d_a, N * sizeof(float));
```

```
    cudaMalloc((void **)&d_b, 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);
        N_sum<<<blocks[k], threads[k]>>>(d_a, d_b);

        cudaError err = cudaMemcpy(b, d_b, 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);
    }

    printf("\nNumbers : ");
    for (int i = 0; i < N; i++)
        cout << a[i] << "  ";

    printf("\nSum      : ");
    cout << b[0] / 19.0 << endl;
    cudaFree(d_a);
    cudaFree(d_b);

    return 0;
}
```

Output :

```
Blocks =    1 and Threads per Block =    1 Time = 0.16790
Blocks =    1 and Threads per Block =   10 Time = 0.04554
Blocks =    1 and Threads per Block =   20 Time = 0.04202
Blocks =    1 and Threads per Block =   30 Time = 0.03878
Blocks =    1 and Threads per Block =   40 Time = 0.04013
Blocks =    1 and Threads per Block =   50 Time = 0.04147
Blocks =    1 and Threads per Block = 1024 Time = 0.04256
Blocks =   10 and Threads per Block =   10 Time = 0.04902
Blocks =   20 and Threads per Block =   10 Time = 0.04131
Blocks =   30 and Threads per Block =   10 Time = 0.04157
Blocks =   40 and Threads per Block =   10 Time = 0.04173
Blocks =   50 and Threads per Block =   10 Time = 0.04221
Blocks =  128 and Threads per Block = 1024 Time = 0.03738
Blocks =  256 and Threads per Block = 1024 Time = 0.04202
Blocks =  512 and Threads per Block = 1024 Time = 0.04877
Blocks = 1024 and Threads per Block =  128 Time = 0.04278
Blocks = 1024 and Threads per Block =  256 Time = 0.04858
Blocks = 1024 and Threads per Block =  512 Time = 0.04374
Blocks = 1024 and Threads per Block = 1024 Time = 0.04838

Numbers : 1.2564  2.2564  3.2564  4.2564  5.2564  6.2564  7.2564  8.2564  9.2564  10.2564  11.25
Sum     : 1.66638e+06
```

Observations:

Number of Blocks	Threads per Block	Execution Time	Speed-up	Parallelization Fraction
1	1	0.167	1.0	
1	10	0.045	3.7111	81.1709
1	20	0.042	3.9762	78.7899
1	30	0.038	4.3947	79.9089
1	40	0.040	4.175	77.9979
1	50	0.041	4.0732	76.9891

1	1024	0.042	3.9762	74.9235
10	10	0.049	3.4082	78.51
20	10	0.041	4.0732	83.8325
30	10	0.041	4.0732	83.8325
40	10	0.041	4.0732	83.8325
50	10	0.042	3.9762	83.1671
128	1024	0.048	3.4792	71.3274
256	1024	0.042	3.9762	74.9235
512	1024	0.037	4.5135	77.9203
1024	128	0.043	3.8837	74.836
1024	256	0.048	3.4792	71.5372
1024	512	0.042	3.9762	74.9968
1024	1024	0.048	3.4792	71.3274

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)

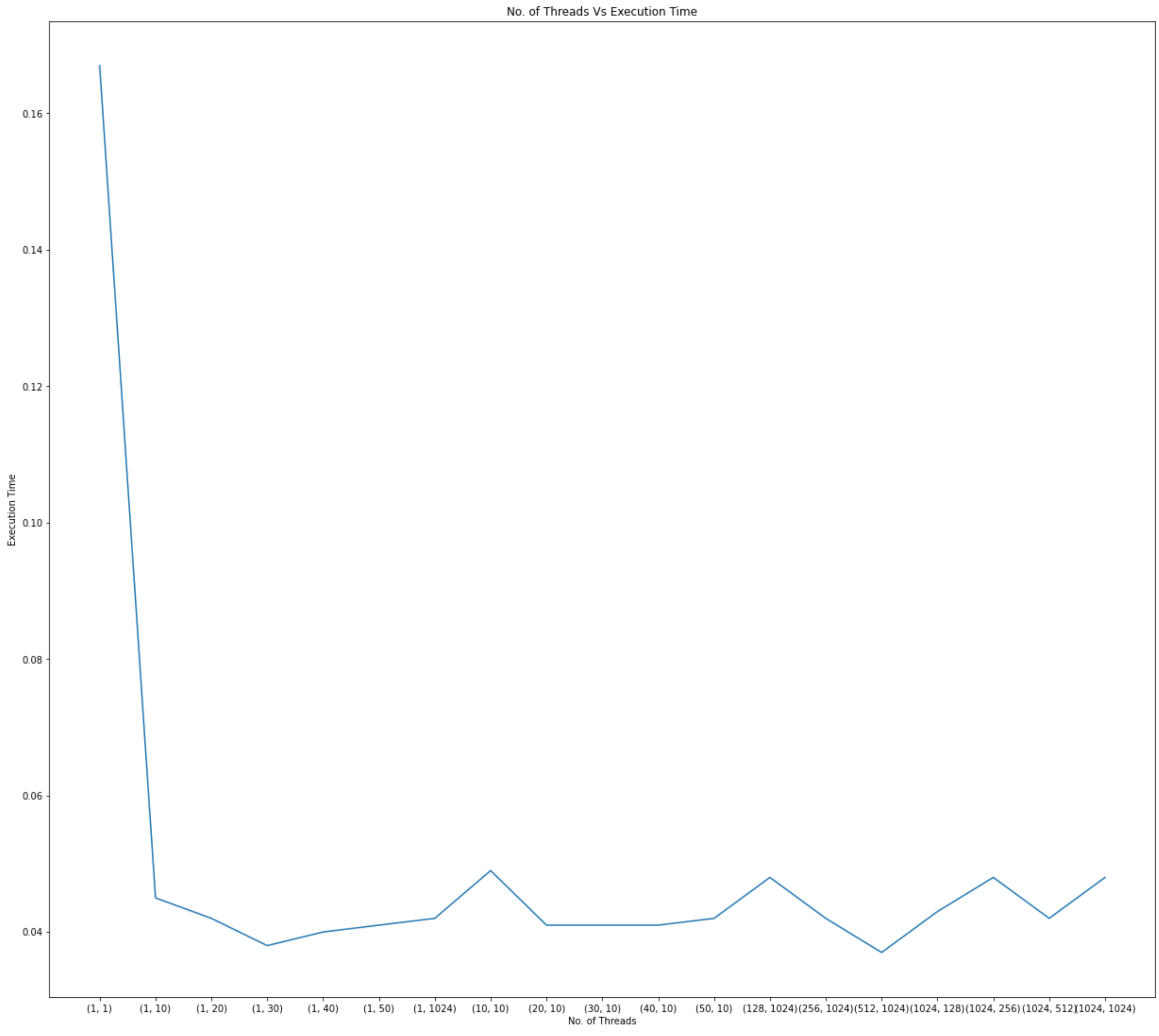
Parallelization Fraction can be found using the following formula, **S(n)=1/((1 - p) + p/n)**

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

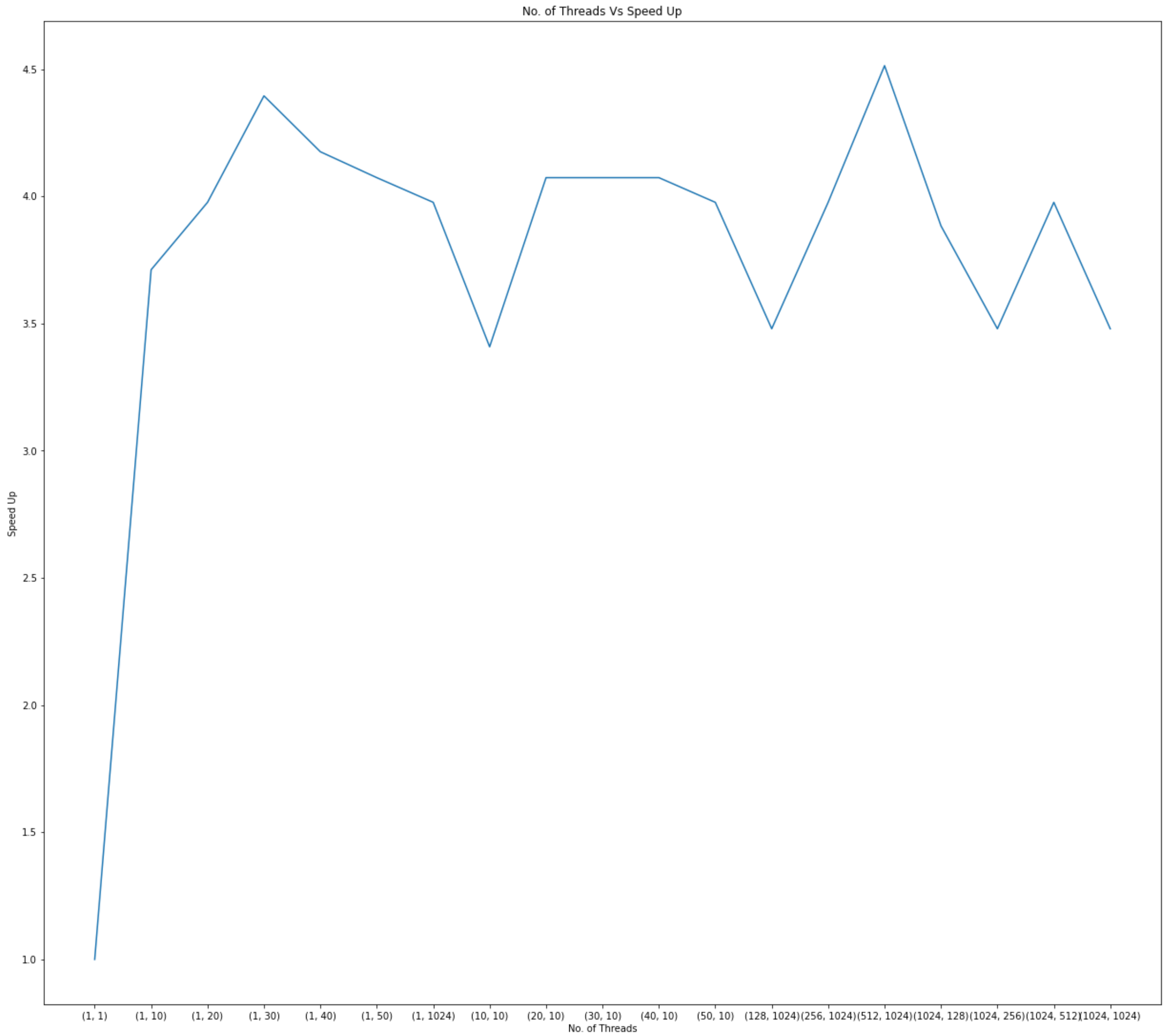
n = Number of threads

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 N_sum function for the different combinations of no. of blocks and no. of threads.
- The Maximum speedup was for 512 number blocks with 1024 threads per block combination. This is because it has reasonably fewer communication overheads and also a good amount of parallelization