HIGH PERFORMANCE COMPUTING PRACTICE

EXPERIMENT - 9

CUDA:

1) Addition of N-numbers

2) Vector Dot Product

Use input as a larger double number (64-bit).

Run experiment for Threads = {1,2,4,8,16,32,64,128,256,500}.

Estimate the parallelization fraction.

Document the report and submit.

Bhaavanaa Thumu - CED17I021

Specifications

All the elements of the matrices are produced randomly and are large double numbers (64-bit).

Formulae used

Speed-up = T(1)/T(P)

Parallel fraction, f = (1 - T(P)/T(1))/(1 - (1/P))

where,

T(1) - time taken for serial execution

T(P) - time taken for parallel execution

P - number of processes/threads/processors

Other functions used

cudaMalloc - allocates a block of memory

cudaMemcpy - copies ‘n’ characters from source to destination memory area.

1. Sum of N numbers

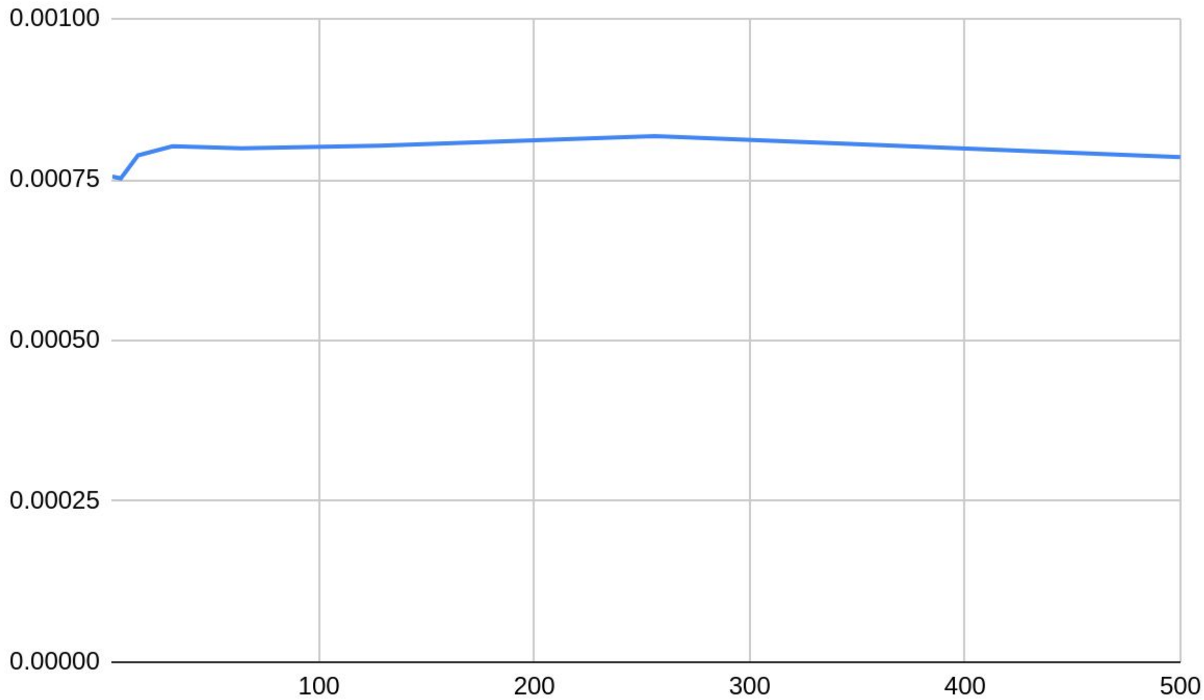
This program contains a vector of size 10,000.

Hence, N = 10,000.

The number of processors initialized and the execution time in each case is as shown in the table.

No. of processors	*Execution time*	*Speed-up*	*Parallel fraction*
1	0.000792	1	#DIV/0!
2	0.000757	1.046235139	0.08838383838
4	0.000755	1.049006623	0.06228956229
8	0.000752	1.053191489	0.05772005772
16	0.000788	1.005076142	0.005387205387
32	0.000802	0.9875311721	-0.01303356142
64	0.000799	0.9912390488	-0.008978675645
128	0.000803	0.9863013699	-0.01399825022
256	0.000818	0.9682151589	-0.03295702119
500	0.000785	1.008917197	0.00885609603

The plot below is the graph between no. of processors and execution time.



x-axis : no. of processors

y-axis : execution time

Hence, in this case, the optimal number of processors is 8. In this case-

Execution time = 0.000752

Speed-up = 1.053191489

Parallel fraction = 0.05772005772

Addition of N numbers code

```
%%cu

#include<stdio.h>
#include<math.h>
#include <sys/time.h>

#define N 100000
#define thread_count 500


__global__ void vect_add(double *d_result, double *d_a)
{
   int load = N/thread_count;
   __shared__ double s_sum[thread_count];
   s_sum[threadIdx.x] = 0;

   if(threadIdx.x != thread_count-1)
   {
       for(int i=threadIdx.x*load; i<(threadIdx.x+1)*load; i++)
       {
           s_sum[threadIdx.x] += d_a[i];
       }
   }
   else
   {
       for(int i=threadIdx.x*load; i<N; i++)
       {
           s_sum[threadIdx.x] += d_a[i];
       }
   }
   __syncthreads();
```

```
   if(threadIdx.x == 0)
   {
       *d_result = 0;
       for(int i=0; i<thread_count; i++)
       {
           *d_result += s_sum[i];
       }
   }
}

int main()
{
   double *a, out, *d_a, *d_result;
   struct timeval t1, t2;

   a = (double*)malloc(sizeof(double) * N);
   cudaMalloc((void**)&d_a, sizeof(double) * N);
   cudaMalloc((void**)&d_result, sizeof(double));
   for(int i=0; i<N; i++)
   {
       a[i] = pow(2,15)+rand()+0.13246549884;
   }
   cudaMemcpy(d_a, a, sizeof(double) * N,
cudaMemcpyHostToDevice);
   vect_add<<<1,thread_count>>>(d_result, d_a);
   cudaMemcpy(&out, d_result, sizeof(double),
cudaMemcpyDeviceToHost);

   return 0;
}
```

2. Vector dot product

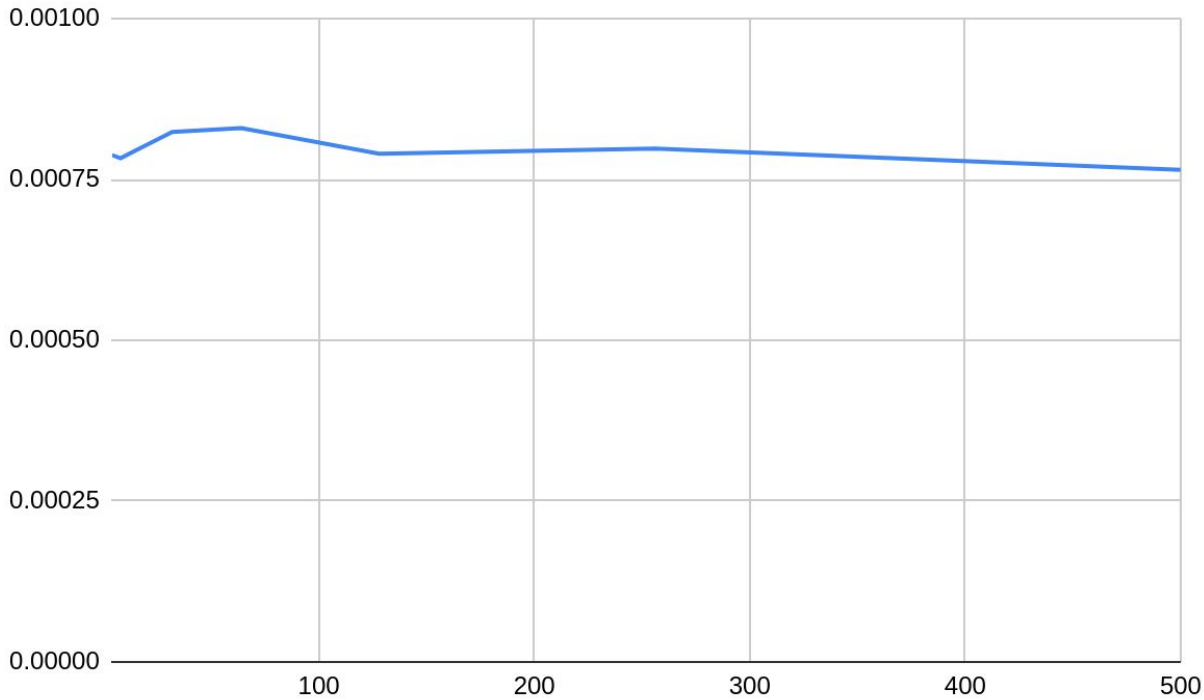
This program contains vectors of size 1,000.

Therefore, here, N = 1,000.

The number of processors initialized and the execution time in each case is as shown in the table.

No. of processors	*Execution time*	*Speed-up*	*Parallel fraction*
1	0.000815	1	#DIV/0!
2	0.000805	1.01242236	0.0245398773
4	0.000788	1.034263959	0.04417177914
8	0.000783	1.040868455	0.04487291849
16	0.000797	1.022584693	0.02355828221
32	0.000824	0.9890776699	-0.01139916881
64	0.000830	0.9819277108	-0.01869704937
128	0.000790	1.03164557	0.03091638085
256	0.000798	1.021303258	0.0209406953
500	0.000765	1.065359477	0.06147263853

The plot below is the graph between no. of processors and execution time.



x-axis : no. of processors

y-axis : execution time

Hence, in this case, the optimal number of processors is 500. In this case-

Execution time = 0.000765

Speed-up = 1.065359477

Parallel fraction = 0.06147263853

Vector dot product code

```
%%cu

#include<stdio.h>
#include<math.h>
#include <sys/time.h>

#define N 1000
#define thread_count 500

__global__ void vect_add(double *d_result, double *d_a, double
*d_b)
{
   int load = N/thread_count;

   __shared__ double s_sum[N];

   if(threadIdx.x != thread_count-1)
   {
       for(int i=threadIdx.x*load; i<(threadIdx.x+1)*load; i++)
       {
           s_sum[i] = d_a[i] * d_b[i];
       }
   }
   else
   {
       for(int i=threadIdx.x*load; i<N; i++)
       {
           s_sum[i] = d_a[i] * d_b[i];
       }
   }
```

```
    __syncthreads();
    if(threadIdx.x == 0)
    {
        *d_result = 0;
        for(int i=0; i<N; i++)
        {
            *d_result += s_sum[i];
        }
    }
}
int main()
{
    double *a, *b, out, *d_a, *d_b, *d_result;

    a = (double*)malloc(sizeof(double) * N);
    b = (double*)malloc(sizeof(double) * N);
    cudaMalloc((void**)&d_a, sizeof(double) * N);
    cudaMalloc((void**)&d_b, sizeof(double) * N);
    cudaMalloc((void**)&d_result, sizeof(double));
    for(int i=0; i<N; i++)
    {
        a[i] = pow(2,15)+rand()+0.13246549884;
        b[i] = pow(2,15)+rand()+0.13246549884;
    }
    cudaMemcpy(d_a, a, sizeof(double) * N,
cudaMemcpyHostToDevice);
    cudaMemcpy(d_b, b, sizeof(double) * N,
cudaMemcpyHostToDevice);
    vect_add<<<1,thread_count>>>(d_result, d_a, d_b);
    cudaMemcpy(&out, d_result, sizeof(double),
cudaMemcpyDeviceToHost);
    return 0;
}
```