HIGH PERFORMANCE COMPUTING PRACTICE

EXPERIMENT - 8

CUDA:

1) Matrix addition

2) Matrix multiplication column major order

3) Matrix multiplication block based approach

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

The size of all the matrices used for below programs is 1,000x1,000.

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.

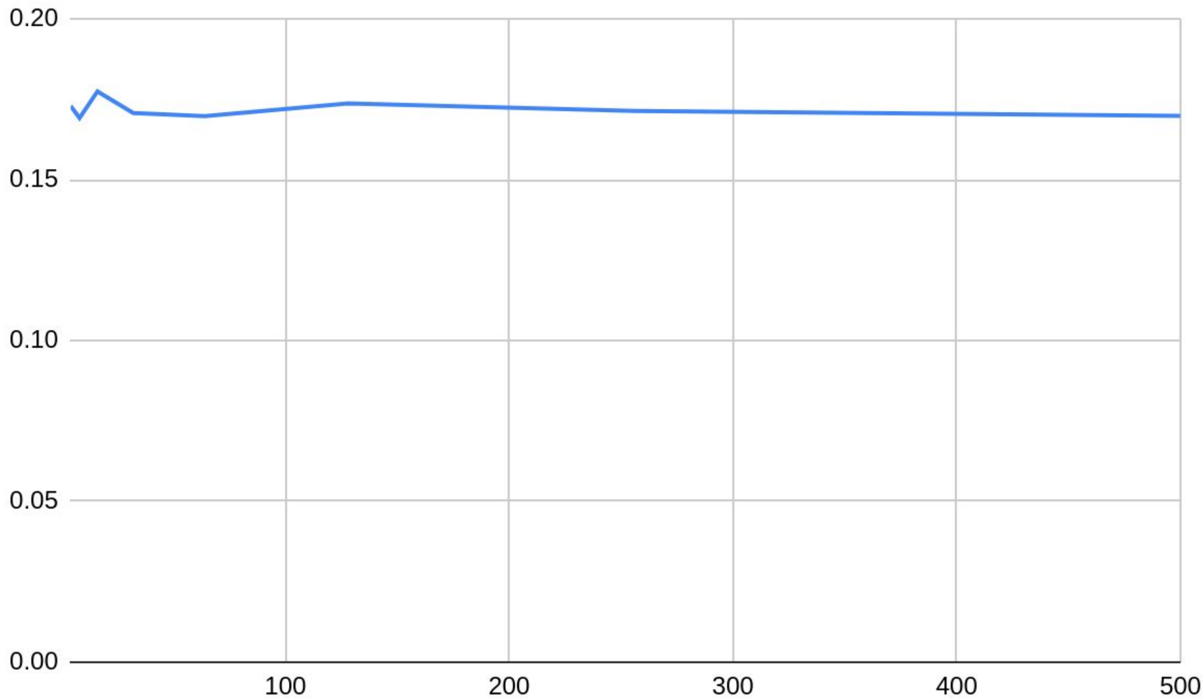
1. Matrix addition

This program contains matrices of size 1,000x1,000.

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	0.175272	1	#DIV/0!
2	0.169077	1.036640111	0.07069012734
4	0.172974	1.013285234	0.01748140034
8	0.169208	1.035837549	0.03954017592
16	0.177450	0.98772612	-0.01325482678
32	0.170749	1.026489174	0.02663804387
64	0.169748	1.032542357	0.03201699381
128	0.173706	1.009015233	0.009005036222
256	0.171361	1.022823163	0.02240139472
500	0.169900	1.031618599	0.03071092662

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 2. In this case-

Execution time = 0.169077

Speed-up = 1.036640111

Parallel fraction = 0.07069012734

Matrix addition code

```
%%cu

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

#define M 1000
#define N 1000

__global__ void matrices_add(double *d_result, double *d_a,
double *d_b, int thread_count)
{
   int elementsReceived = M/thread_count;

   if(thread_count != 1)
   {
       int start_index, end_index;

       start_index = (threadIdx.x * elementsReceived);

       if(threadIdx.x == thread_count-1)
           end_index = M;
       else
           end_index=(threadIdx.x + 1)*elementsReceived;

       for(int i=start_index; i<end_index; i++)
       {
           for(int j=0; j<N; j++)
           {
               d_result[i*N+j] = d_a[i*N+j] + d_b[i*N+j];
           }
```

```
        }
    }
    else
    {
        for(int i=0; i<M; i++)
        {
            for(int j=0; j<N; j++)
                d_result[i*N+j] = d_a[i*N+j] + d_b[i*N+j];
        }
    }
}

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

    int size = sizeof(double)*M*N;
    a = (double*)malloc(size);
    b = (double*)malloc(size);
    out = (double*)malloc(size);

    cudaMalloc((void**)&d_a, size);
    cudaMalloc((void**)&d_b, size);
    cudaMalloc((void**)&d_result, size);
    gettimeofday(&t1, 0);

    for(int i = 0; i < M; i++)
    {
        for(int j=0;j<N;j++)
        {
            a[i*N+j] = pow(2,15)+rand()+0.13246549884;
            b[i*N+j] = pow(2,15)+rand()+0.13246549884;
```

```
            // a[i*N+j] = i;
            // b[i*N+j] = j;
        }
    }

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

    int thread_count = 500;

    matrices_add<<<1,thread_count>>>(d_result, d_a, d_b,
thread_count);

    cudaMemcpy(out, d_result, size, cudaMemcpyDeviceToHost);

    gettimeofday(&t2, 0);
     double time = (float)(1000000.0*(t2.tv_sec-t1.tv_sec) +
t2.tv_usec-t1.tv_usec)/1000000.0;
    printf("time taken : %lf sec\n", time);
     return 0;
}
```

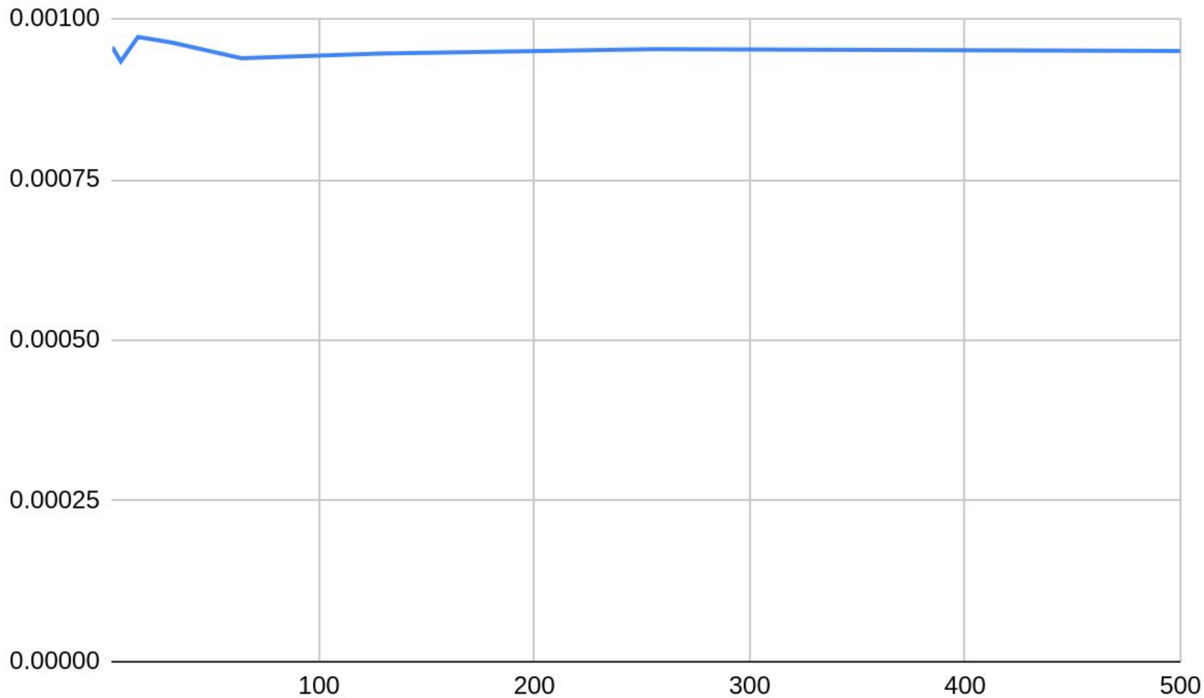
2. Matrix multiplication column major order

This program contains matrices of size 1,000x1,000.

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	0.000947	1	#DIV/0!
2	0.000950	0.9968421053	-0.006335797254
4	0.000956	0.9905857741	-0.01267159451
8	0.000934	1.01391863	0.01568864082
16	0.000972	0.9742798354	-0.02815909891
32	0.000963	0.9833852544	-0.01744047416
64	0.000939	1.008519702	0.00858182062
128	0.000946	1.001057082	0.001064280904
256	0.000953	0.9937040923	-0.006360643518
500	0.000950	0.9968421053	-0.003174247121

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 = 0.000934

Speed-up = 1.01391863

Parallel fraction = 0.01568864082

Matrix multiplication column major order code

```
%%cu

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

#define M 1000
#define N 1000

#define thread_count 500

__global__ void mat_mul(double *d_result, double *d_a, double
*d_b)
{
   int elementsReceived = M/thread_count;
   if(thread_count != 1)
   {
       int start_index, end_index;

       start_index=(threadIdx.x * elementsReceived);

       if(threadIdx.x==thread_count-1)
           end_index = M;
       else
           end_index=(threadIdx.x + 1)*elementsReceived;

       for(int i=start_index; i<end_index; i++)
       {
           for(int j=0;j<N; j++)
           {
```

```
                for(int k=0; k<N; k++)
                {
                    d_result[i+j*M] = d_result[i+j*M] +
d_a[i+k*M] * d_b[k+j*N];   // Take row of a and column of b
                }
            }
        }
    }
    else
    {
        for(int i=0; i<M; i++)
        {
            for(int j=0; j<N; j++)
            {
                for(int k=0; k<N; k++)
                {
                    d_result[i+j*M] = d_result[i+j*M] +
d_a[i+k*M] * d_b[k+j*N];   //Take row of a and column of b
                }
            }
        }
    }
 }

int main(void)
{
    double *a, *b, *out, *d_a, *d_b, *d_result;
    struct timeval t1, t2;
     int size = sizeof(double)*M*N;
    a = (double *)malloc(size);
    b = (double *)malloc(size);
    out = (double *)malloc(size);
```

```
    cudaMalloc((void **)&d_a, size);
    cudaMalloc((void **)&d_b, size);
    cudaMalloc((void **)&d_result, size);

    for(int i = 0; i < M; i++)
    {
        for(int j=0;j<N;j++)
        {
            a[i+j*M] = pow(2,15)+rand()+0.13246549884;
            b[i+j*M] = pow(2,15)+rand()+0.13246549884;
            // a[i+j*M] = i;
            // b[i+j*M] = j;
        }
    }
    gettimeofday(&t1, 0);
    cudaMemcpy(d_a, a, size, cudaMemcpyHostToDevice);
    cudaMemcpy(d_b, b, size, cudaMemcpyHostToDevice);
     mat_mul<<<1,thread_count>>>(d_result, d_a, d_b);

    cudaMemcpy(out, d_result, size, cudaMemcpyDeviceToHost);
    gettimeofday(&t2, 0);
    double time = (float)(1000000.0*(t2.tv_sec-t1.tv_sec) +
t2.tv_usec-t1.tv_usec)/1000000.0;
    printf("time taken : %lf sec\n", time);
    return 0;
}
```

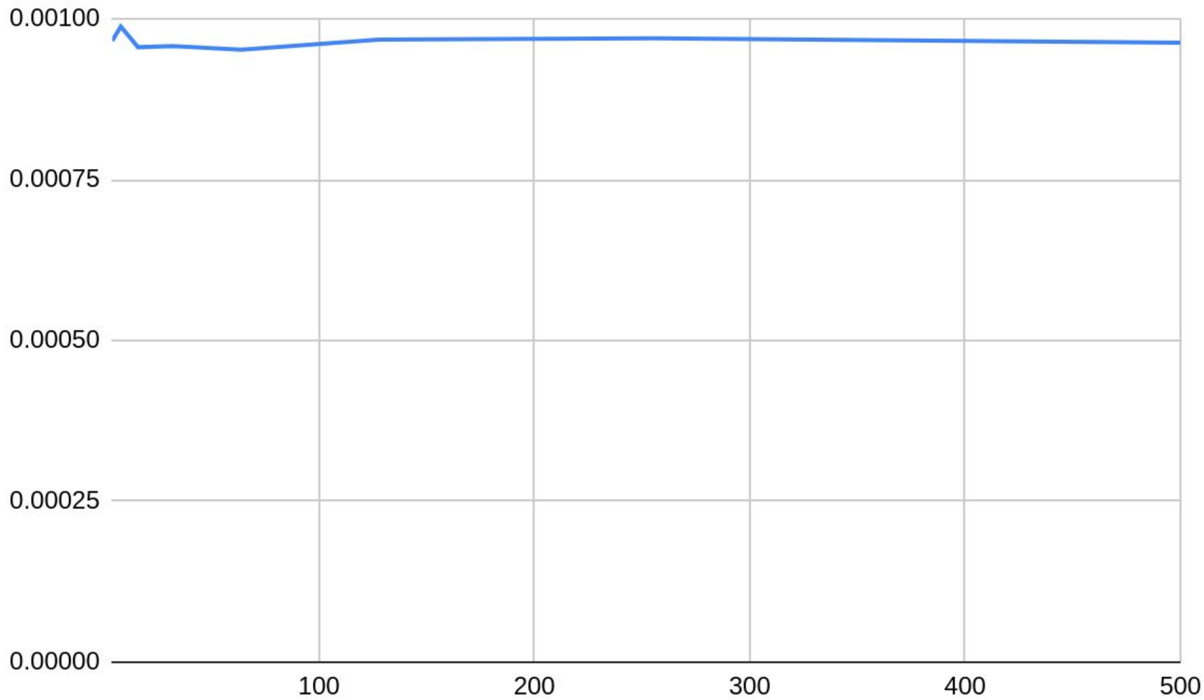
3. Matrix multiplication block based approach

This program contains matrices of size 1,000x1,000.

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	0.000962	1	#DIV/0!
2	0.000969	0.9927760578	-0.01455301455
4	0.000966	0.9958592133	-0.005544005544
8	0.000988	0.9736842105	-0.03088803089
16	0.000956	1.006276151	0.006652806653
32	0.000958	1.004175365	0.004292133324
64	0.000952	1.010504202	0.01056001056
128	0.000968	0.9938016529	-0.006286116522
256	0.000970	0.9917525773	-0.008348620113
500	0.000963	0.9989615784	-0.001041584208

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 64. In this case-

Execution time = 0.000952

Speed-up = 1.010504202

Parallel fraction = 0.01056001056

Matrix multiplication block based approach code

```
%%cu

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

#define M 1000
#define N 1000

#define num_threads 500

__global__ void matmul(double *a, double *b, double *c)
{
   int B=5;
   int elementsReceived = N/num_threads;
   int mini,minik;
   double r;

   if(num_threads != 1)
   {
       int start_index,end_index;
       start_index=(threadIdx.x * elementsReceived);

       if(threadIdx.x==num_threads-1)
           end_index=N;//Num of rows
       else
           end_index=(threadIdx.x + 1)*elementsReceived;

       for(int jj=0; jj<N; jj=jj+B)
       {
```

```
            for(int kk=0; kk<N; kk=kk+B)
            {
                for(int i=start_index; i<end_index; i++)
                {
                    if (jj+B<N)
                        mini=jj+B;
                    else
                        mini=N;

                    for(int j=jj;j<mini;j++)
                    {
                        r=0;
                        if (kk+B<N)
                            minik=kk+B;
                        else
                            minik=N;

                        for(int k=kk;k<minik;k++)
                            r+=a[i*N+k]* b[k*N+j];

                        c[i*N+j]+=r;
                    }
                }
            }
        }
    }
    else
    {
        for(int jj=0; jj<N; jj=jj+B)
        for(int kk=0; kk<N; kk=kk+B)
        for(int i=0; i<N; i++)
        {
            if (jj+B<N)
```

```
                mini=jj+B;
            else
                mini=N;
            for(int j=jj;j<mini;j++)
            {
                r=0;
                  if (kk+B<N)
                   minik=kk+B;
                else
                   minik=N;

                for(int k=kk;k<minik;k++)
                    r+=a[i*N+k]* b[k*N+j];

                c[i*N+j]+=r;
            }
        }
    }
 }

int main(void)
{
   double *a, *b, *c;   // host copies of a, b, c
   double *d_a, *d_b, *d_c;  // device copies of a, b, c
    int size = sizeof(double)*N*N;
   double start,end;
    cudaMalloc((void **)&d_a, size);    // Allocate space for
device copies of a, b, c
   cudaMalloc((void **)&d_b, size);
   cudaMalloc((void **)&d_c, size);

   a = (double *)malloc(size);
   b = (double *)malloc(size);
```

```
   c = (double *)malloc(size);

    for(int i = 0; i < N; i++)          // Setup input values
   {
       for(int j=0;j<N;j++)
       {
           a[i+j*N] = pow(2,15)+rand()+0.13246549884;
           b[i+j*N] = pow(2,15)+rand()+0.13246549884;
       }
   }
   cudaMemcpy(d_a, a, size, cudaMemcpyHostToDevice);     // Copy
inputs to device
   cudaMemcpy(d_b, b, size, cudaMemcpyHostToDevice);

   matmul<<<1,num_threads>>>(d_a, d_b, d_c);       // Launch
matmul() kernel on GPU

   cudaMemcpy(c, d_c, size, cudaMemcpyDeviceToHost);

   return 0;
}
```