Course Project- High Performance Scientific Computing

Parallelizing Power Method Algorithm for finding Eigenvalue using Open MP and CUDA

Submitted by

Basu Parmar(130010003)

Micky Droch (140010059)

Prashanth Darawath

Theory

In linear algebra, an eigenvector or characteristic vector of a linear transformation is a non-zero vector that does not change its direction when that linear transformation is applied to it. In other words, if v is a vector that is not the zero vector, then it is an eigenvector of a linear transformation *T* if *T*(v) is a scalar multiple of v. This condition can be written as the equation

$$T(\mathbf{v}) = \lambda\mathbf{v},$$

where λ is a scalar known as the eigenvalue or characteristic value associated with the eigenvector v. If the linear transformation *T* is expressed as a square matrix *A*, then the equation can be expressed as the matrix multiplication

$$A\mathbf{v} = \lambda\mathbf{v},$$

where *v* is a column vector. There is a correspondence between *n* by *n* square matrices and linear transformations from an *n*-dimensional vector space to itself. For this reason, it is equivalent to define eigenvalues and eigenvectors using either the language of matrices or the language of linear transformations

Eigen values are very important in any field of science. However finding them exactly is very difficult especially when matrix size is very large.

In this project we write a code of power method for calculating dominant eigenvalue (the largest one) and the eigenvector of any matrix. We will then parallelize it using Open MP and CUDA.

Power method

The power method is explained briefly below:

The power iteration algorithm starts with a vector b_0, which may be an approximation to the dominant eigenvector or a random vector. The method is described by the recurrence relation

$$b_{k+1} = \frac{Ab_k}{\|Ab_k\|}.$$

So, at every iteration, the vector b_k is multiplied by the matrix *A* and normalized.

If we assume *A* has an eigenvalue that is strictly greater in magnitude than its other eigenvalues and the starting vector b_0 has a nonzero component in the direction of an eigenvector associated with the dominant eigenvalue, then a subsequence $\mathbf{b_k}$ converges to an eigenvector associated with the dominant eigenvalue.

Now we look at the C code of power method.

C++ code:

```
// Code for Finding Eigenvalue of a matrix Serial version//

#include <iostream>
#include <math.h>
#include <stdlib.h>
#include <time.h>
#include <iomanip>
#define n 15000     //The matrix size is defined here

using namespace std;
int main(){
double a[n][n],x[n],c[n],o=0,temp=0,d=2;
srand(time(NULL));

//Random number generator used to get average execution time for different
matrices
for(int i=0;i<n;i++)
{
      for(int j=0;j<n;j++)
      {
            a[i][j]=rand()%100+(i+j)*3.149568298;

            //cin>>a[i][j];                 //generating the matrix a[n][n]
            //cout<<" "<<a[i][j]<<endl;
      }
}

// The initial guess for the eigenvector is given here
for(int i=0;i<n;i++)
{
      x[i]=0.5;
}
x[n-1]=1;

//The power method algorithm
 while(fabs(d-temp)>0.0000000000001) //Tolerance factor
    {

        for(int i=0;i<n;i++)
        {
            c[i]=0;
            for(int j=0;j<n;j++){
                c[i]+=a[i][j]*x[j];
                                              }
        }
        for(int i=0;i<n;i++){
            x[i]=c[i];
            }
        temp=d;
        d=0;

        for(int i=0;i<n;i++)
        {
            if(fabs(x[i])>fabs(d))
                d=x[i];
```

```
}
        for(int i=0;i<n;i++){
            x[i]/=d;
            }
    }

    //cout<<d<<endl;   //d is the dominant eigenvalue
    //for(int i=0;i<n;i++){
            cout<<setprecision(20)<<x[i]/d<<endl; // The eigenvector
      //}

return 0;
}
```

The power method gives different execution times for different matrices. (Timing function included in the uploaded code).

Now the code has been parallelized using Open MP and CUDA.

In Open MP the no of threads used are 4. CUDA used threads equal to the number of elements in the matrix. (More are created but less are used).

The codes are times using CUDA timing function for serial and CUDA code, Open MP timing function for Open MP code.

The timings are compared for different versions of the algorithm. The timings are taken on an average as they vary for different matrices so the maximum over some iterations is taken for each matrix size.

The matrix size is limited to 13000*130000 due the limitation on the memory of GPU.

The execution time is tabulated in the table:

N	100	500	1000	2000	5000	10000	11000	13000
GPU	1.74432	6.26704	31.8551	508.247	1394.53	20959.5	92615	181841
serial(ms)	0.467136	15.1033	78.6301	211.291	1275.38	14565	17589	35684
Open MP (s)	0.000965	0.017924	0.049823	0.221487	0.964733	3.758343	4.823648	6.394727

From the execution times we can see that Open MP code runs faster than serial one. However the CUDA code slows down because for each while loop iteration too many threads are created

and destroyed and also too much data is being transferred (copy from device to host and host to device). This causes the overhead to become significant hence the code slows down.

The CUDA code is profiled using nvprof for N=1000.

Elapsed Time = 34.1454 ms==5700== Profiling result:

Time(%)	Time	Calls	Avg	Min	Max	Name
93.94%	25.810ms	15	1.7207ms	2.1760us	5.1765ms	[CUDA memcpy HtoD]
5.90%	1.6210ms	5	324.20us	323.33us	324.64us	Matrix_Product(double*, double*, double*)
0.16%	42.688us	5	8.5370us	8.5120us	8.5760us	[CUDA memcpy DtoH]

==5700== API calls:

Time(%)	Time	Calls	Avg	Min	Max	Name
87.72%	248.40ms	2	124.20ms	4.1740us	248.39ms	cudaEventCreate
10.21%	28.915ms	20	1.4458ms	16.844us	4.5147ms	cudaMemcpy
1.01%	2.8481ms	15	189.87us	10.299us	389.20us	cudaMalloc
0.67%	1.9065ms	15	127.10us	13.877us	256.38us	cudaFree
0.25%	721.82us	83	8.6960us	110ns	352.70us	cuDeviceGetAttribute
0.08%	234.32us	5	46.864us	37.444us	58.780us	cudaLaunch
0.02%	63.530us	1	63.530us	63.530us	63.530us	cuDeviceTotalMem
0.02%	44.607us	1	44.607us	44.607us	44.607us	cuDeviceGetName
0.01%	16.872us	15	1.1240us	268ns	7.0330us	cudaSetupArgument
0.01%	16.552us	2	8.2760us	7.5630us	8.9890us	cudaEventRecord
0.00%	9.5210us	5	1.9040us	1.3770us	2.6230us	cudaConfigureCall
0.00%	7.3790us	1	7.3790us	7.3790us	7.3790us	cudaEventSynchronize
0.00%	2.6790us	1	2.6790us	2.6790us	2.6790us	cudaEventElapsedTime
0.00%	1.1290us	2	564ns	206ns	923ns	cuDeviceGetCount
0.00%	390ns	2	195ns	138ns	252ns	cuDeviceGet

The max time here is taken by CUDA Memcpy (The cudaEventcreate overhead is constant for all and Memcpy exceeds it for higher N). Thus we see why the code slows down for CUDA due to massive overhead by CUDA Memcpy.

The open_mp and CUDA codes are profiled using valgrind. The cachegrind tool is used for calculating hits and misses.

Open Mp:

Instruction:

N	Reads	Misses	Hits	Hit Rate	Miss Rate
1000	6.3E+08	4285	6.3E+08	0.99999	6.81E-006

Data Reads:

N	Total	Misses	Hits	Hit Rate	Miss Rate
1000	4.2E+08	4771370	4.2E+08	0.98864	0.011360063

Data Writes:

N	Total	Misses	Hits	Hit Rate	Miss Rate
1000	3E+07	260774	3E+07	0.99138	8.62E-003

The size of instructions is very less thus instruction misses are very less. The miss rate is 1% for data reads and of the order of 10^{-3} for data writes. Thus the code is well optimized

Serial:

Instruction

N	Total	Misses	Hits	Hit Rate	Miss Rate
1000	2556838440	48438	2.56E+09	0.999981	1.89E-005

Data Reads

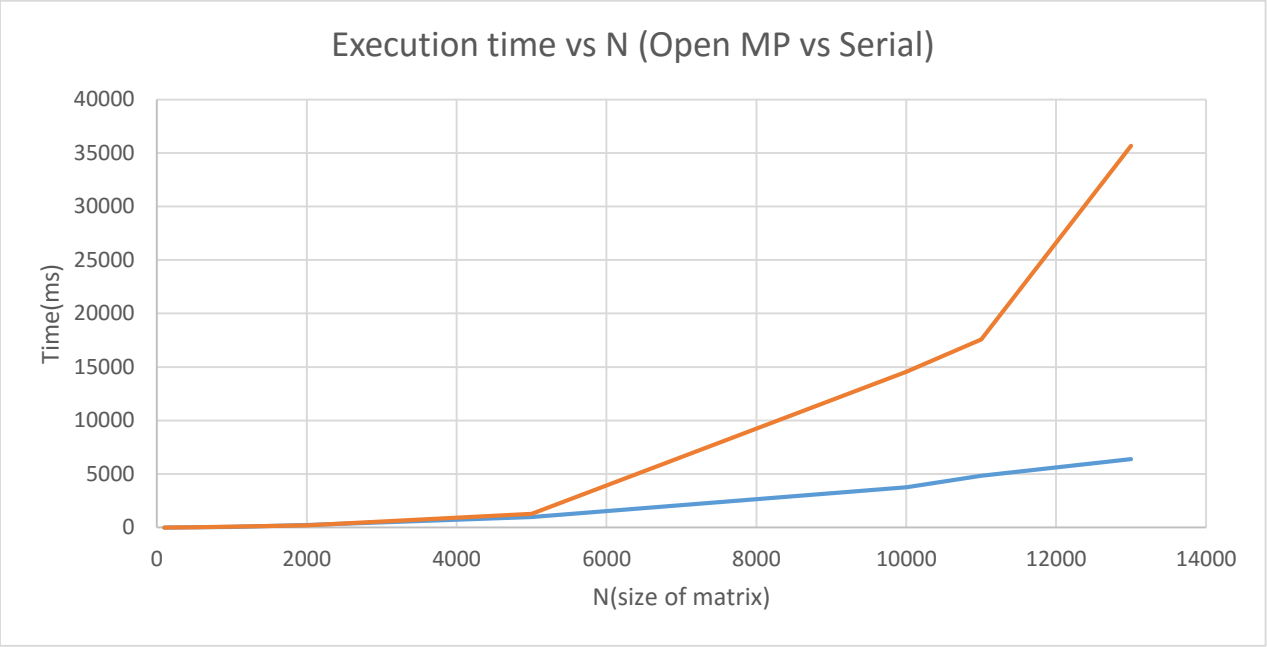
N	Total	Misses	Hits	Hit Rate	Miss Rate
1000	1199470730	29438555	1.17E+09	0.975457	0.024542954

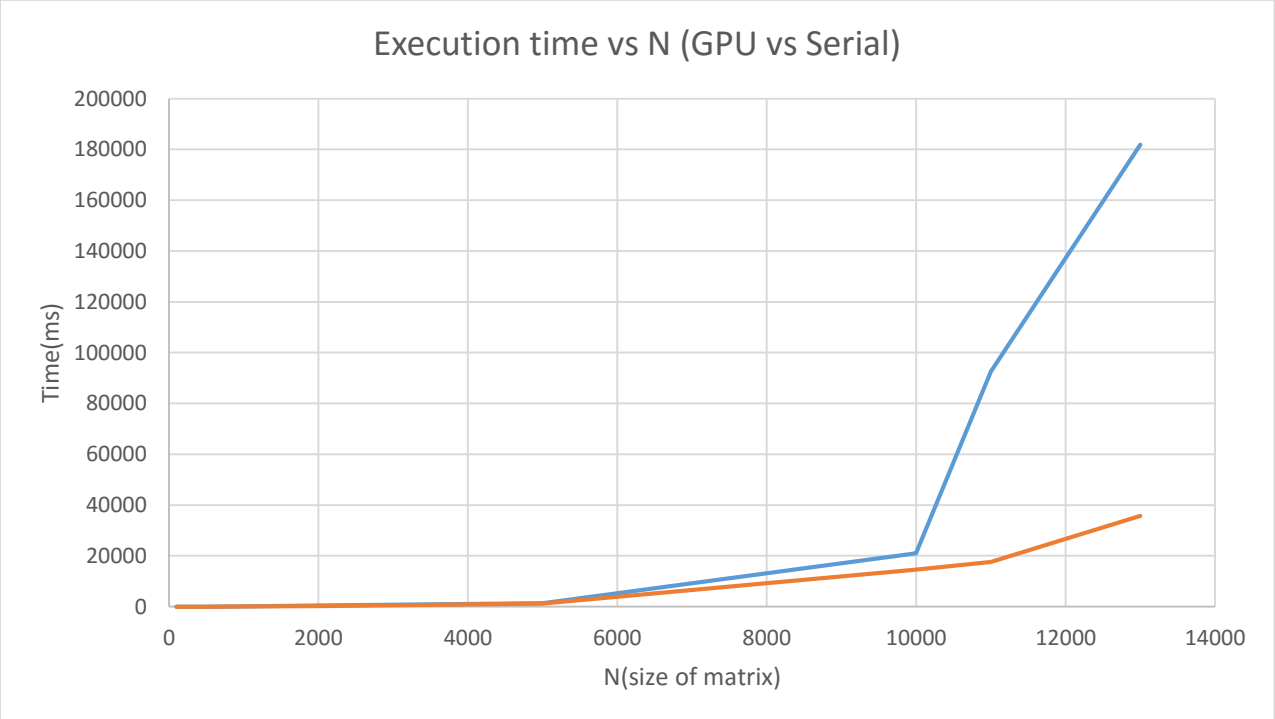
Data writes

N	Total	Misses	Hits	Hit Rate	Miss Rate
1000	128073131	515448	1.28E+08	0.995975	4.02E-003

We can see that the serial code has more data misses than the Open MP code. This is evident as each thread has less data thus less total misses.

Now we can see the difference in execution times with simple graphs.





We can infer from the plots the speedup involved with Open MP threads when compared to the serial code. The speed up is less compared for low N due to thread creation overhead involved. The GPU time is more due to the reasons mentioned above.

Conclusion

From the project we conclude that by parallelizing the code we can improve the execution time significantly. However care has to be taken of the overhead involved as it can slow down the code. The miss rates are less than serial code for parallel codes as expected.