Laboratory exercises (GPU)
Eigenvalues and Eigenvectors: Power Method

Introduction HPC, TU Delft, course 2019–2020
Designed by Xiaohui Wang and Cong Xiao
December 9, 2019

1 Introduction

1.1 Eigenvalues and Eigenvectors

Let A be an n×n matrix, A non-zero vector x is an eigenvector of A if there exists a scalar λ such that

$$Ax = \lambda x \tag{1}$$

The scalar λ is called the eigenvalue of the matrix A, corresponding to the eigenvector x.

Let λ_1,λ_2,λ_3,...,λ_n be a set of the eigenvalues of an n×n matrix A. If $|\lambda_1| > |\lambda_i|$, (i=2,3,...,n), then λ_1 is called the **dominant eigenvalue of A** and the eigenvectors corresponding to λ_1, is called **dominant eigenvector of A**.

Let us consider a system,

$$Ax = \lambda x \tag{2}$$

for which we want to find the eigenvalues and eigenvectors. The standard method for that is to solve for the roots of λ of the characteristic equation

$$|A - \lambda I| = 0 \tag{3}$$

when A is large, this method is totally impractical. Evaluating the determinant of an n×n matrix is a huge task, when n is large and solving the resulting n-th degree polynomial equation for λ is another additional task on top of that.

The power method is a simple iteration method that can be used to find λ_1 and x_1 for a given matrix, where λ_1 is the largest eigenvalue and x_1 is the corresponding eigenvector. Similarly, the inverse power method can be used to find the smallest eigenvalue and its corresponding eigenvector, which is very similar to power method.

1.2 The Power Method

In mathematics, the power method (also known as power iteration) is an eigenvalue algorithm: given a matrix A, the algorithm will produce a number λ, which is the greatest (in absolute value) eigenvalue of A, and a nonzero vector x, the corresponding eigenvector of λ, such that Ax=λx. The algorithm is also known as the Von Mises iteration[1]. The power iteration is a very simple algorithm, but it may converge slowly. It does not compute a matrix decomposition, and hence it can be used when A is a very large sparse matrix.

We first assume that the matrix A has a dominant eigenvalue with the corresponding dominant eigenvectors. As stated before, the power method for approximating eigenvalues is iterative. Hence, we start with an initial approximation

[1] Richard von Mises and H. Pollaczek-Geiringer, Praktische Verfahren der Gleichungsauflösung, ZAMM - Zeitschrift für Angewandte Mathematik and Mechanik 9, 152-164 (1929)

x_0 of the dominant eigenvector of A, which must be non-zero. Thus, we obtain a sequence of eigenvectors given by the recursive formula as follows,

$$x_{k+1} = \frac{Ax_k}{||Ax_k||} \tag{4}$$

So, at every iteration, the vector x_k is multiplied by the matrix A and normalized. If we assume A has an eigenvalue that is strictly greater in magnitude than the other eigenvalues and the starting approximation x_0 of the dominant eigenvector has a nonzero component in the direction of an eigenvector associated with the dominant eigenvalue, then a subsequence x_k converges to an eigenvector associated with the dominant eigenvalue. Without the two assumptions above, the sequence x_k does not necessarily converge.

After obtaining the dominant eigenvector using power method, ideally, one can use Rayleigh quotient[2] to get the associated eigenvalue. This algorithm is the one used for problems such as the *GooglePageRank*[3]. The method can also be used to calculate the spectral radius (or the largest eigenvalue of a matrix) by computing the Rayleigh quotient

$$\frac{x_k^T Ax_k}{x_k^T x_k} = \frac{x_k^T x_{k+1}}{x_k^T x_k} \tag{5}$$

Here, we will give some explicit descriptions about how to implement the aforementioned power method, including the iteration formula and one simple example. As have been mentioned above, a sequence of eigenvectors of matrix A can be calculated by

$$\begin{aligned}
&x_1 = Ax_0\\
&x_2 = Ax_1 = A(Ax_0) = A^2x_0\\
&x_3 = Ax_2 = A(A^2x_0) = A^3x_0\\
&\dots\dots\dots\dots\dots\dots\\
&\dots\dots\dots\dots\dots\dots\\
&x_{k+1} = Ax_k = A(A^{k-1}x_0) = A^kx_0 \qquad (6)
\end{aligned}$$

When k is large, we can obtain a good approximation of the dominant eigenvector of A by properly scaling the sequence.

Example: Use power method to approximate a dominant eigenvalue and the corresponding eigenvector of $\begin{pmatrix} 0 & 11 & -5 \\ -2 & 17 & -7 \\ -4 & 26 & -10 \end{pmatrix}$ after 10 iterations.

Solution: We begin with an initial non-zero approximation of dominant eigenvector x_0 as x_0=$\begin{pmatrix} 1 \\ 1 \\ 0 \end{pmatrix}$ and obtain the following approximation as

$$x_1 = Ax_0 = \begin{pmatrix} 0 & 11 & -5 \\ -2 & 17 & -7 \\ -4 & 26 & -10 \end{pmatrix} \begin{pmatrix} 1 \\ 1 \\ 0 \end{pmatrix} = \begin{pmatrix} 6 \\ 8 \\ 12 \end{pmatrix} = 12.00 \begin{pmatrix} 0.5000 \\ 0.6667 \\ 1.0000 \end{pmatrix} \tag{7}$$

As explained in Example 1, we take out the dominant element 12 out of the resultant matrix and the corresponding vector $\begin{pmatrix} 0.5000 \\ 0.6667 \\ 1.0000 \end{pmatrix}$ will be the new vector

[2] Horn, R. A. and C. A. Johnson. 1985. Matrix Analysis. Cambridge University Press. pp. 176–180.

[3] Langville, Amy N.; Meyer, Carl D. (2003). "Survey: Deeper Inside PageRank". Internet Mathematics. 1(3).

for the next approximation. Proceeding in this manner we obtain a series of approximations as follows,

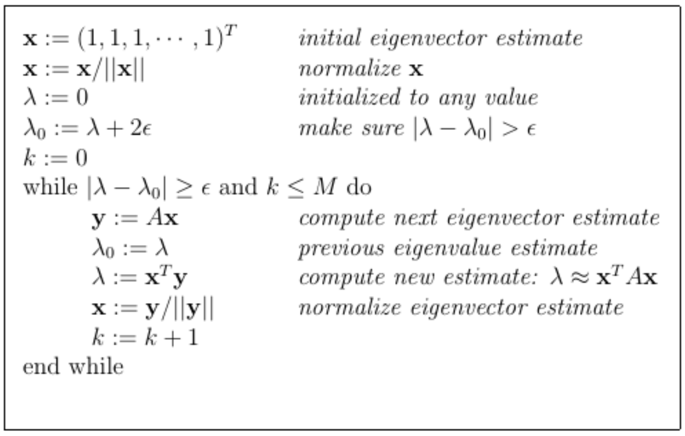
$$x_2 = Ax_1 = \begin{pmatrix} 0 & 11 & -5 \\ -2 & 17 & -7 \\ -4 & 26 & -10 \end{pmatrix} \begin{pmatrix} 0.5000 \\ 0.6667 \\ 1.0000 \end{pmatrix} = \begin{pmatrix} 2.3337 \\ 3.3339 \\ 5.3342 \end{pmatrix} = 5.3342 \begin{pmatrix} 0.4375 \\ 0.6250 \\ 1.0000 \end{pmatrix}$$

$$x_3 = 4.5000 \begin{pmatrix} 0.4167 \\ 0.6111 \\ 1.0000 \end{pmatrix}, x_4 = 4.222 \begin{pmatrix} 0.4079 \\ 0.6053 \\ 1.0000 \end{pmatrix}, x_5 = 4.1050 \begin{pmatrix} 0.4038 \\ 0.6026 \\ 1.0000 \end{pmatrix}$$

$$x_6 = 4.0510 \begin{pmatrix} 0.4019 \\ 0.6013 \\ 1.0000 \end{pmatrix}, x_7 = 4.025 \begin{pmatrix} 0.4009 \\ 0.6006 \\ 1.0000 \end{pmatrix} \tag{8}$$

Therefore, after 7 iteration steps, the recursive procedure is convergent. The dominant eigenvalue is approximately 4.00 and the corresponding eigenvector is $\begin{pmatrix} 0.4000 \\ 0.6000 \\ 1.0000 \end{pmatrix}$.

Given an n×n matrix A, and a tolerance ϵ >0, and the prescribed maximum allowed number of iterations M<0, the general power method algorithm can be formulated as follows,



```
$\mathbf{x} := (1, 1, 1, \cdots, 1)^T$      initial eigenvector estimate
$\mathbf{x} := \mathbf{x}/||\mathbf{x}||$          normalize $\mathbf{x}$
$\lambda := 0$                  initialized to any value
$\lambda_0 := \lambda + 2\epsilon$           make sure $|\lambda - \lambda_0| > \epsilon$
$k := 0$
while $|\lambda - \lambda_0| \geq \epsilon$ and $k \leq M$ do
    $\mathbf{y} := A\mathbf{x}$              compute next eigenvector estimate
    $\lambda_0 := \lambda$              previous eigenvalue estimate
    $\lambda := \mathbf{x}^T\mathbf{y}$             compute new estimate: $\lambda \approx \mathbf{x}^T A\mathbf{x}$
    $\mathbf{x} := \mathbf{y}/||\mathbf{y}||$        normalize eigenvector estimate
    $k := k + 1$
end while
```

Fig. 1: **The illustration of the power method algorithm**.

If the while-loop terminates with $k \leqslant M$, then we conclude the algorithm has terminated successfully. In this case λ is the dominant eigenvalue and x is the corresponding normalized eigenvector.

The rate of convergence of the power method depends on the difference between the magnitude of the dominant eigenvalue and other eigenvalues. Also, the power method will fail if the matrix does not have any real eigenvalues.

2 Parallel Implementation of Power Method in GPUs

2.1 Description of Parallelized Power Method

Suppose we have a parallel machine with p processors. Following the previous parallelized solution of Possion equation, we begin by partitioning the problem into many small individual tasks. We can partition the matrix A into individual rows where a_i is a 1×n row vector that is the i^{th} row of A. Then the product of Ax becomes

$$y = Ax = \begin{bmatrix} a_1 \\ a_2 \\ a_3 \\ ... \\ a_n \end{bmatrix} x = \begin{bmatrix} a_1 x \\ a_2 x \\ a_3 x \\ ... \\ a_n x \end{bmatrix} \tag{9}$$

So the i^{th} entry of y is computed as

$$y_i = a_i x = \sum_{j=1}^{n} a_{ij} x_j \tag{10}$$

for each entry y_i in the result vector. Each of these is a scalar since it is the product of a 1× $n$vector and a n ×1 vector (this is called an *inner product*), and each of these products can be computed in parallel. Since n will usually be larger than the number of processors p in a given parallel machine, we can assign multiple rows to each processor during the agglomeration and mapping phases.

Each task must have access to a row of A and the entire vector of x in order to compute a single y_i. Every task must then communicate its y_i value to all other tasks since the entire vector y is normalized to become the vector x for the next iteration.

One obvious way to agglomerate and map is to group tasks together based on the rows of A they use. The rows may be interleaved or consecutive, but it's probably most natural to consider consecutive rows of A. In the case of p processes the matrix A is blocked into individual blocks A_k, k=1,...,p, where each block has approximately n/p rows. The products $y = Ax$ can be depicted:

$$y = \begin{bmatrix} A_1 \\ A_2 \\ A_3 \\ ... \\ A_p \end{bmatrix} x = \begin{bmatrix} A_1 x \\ A_2 x \\ A_3 x \\ ... \\ A_p x \end{bmatrix} \tag{11}$$

so each process computes the portion of y given by $y_k = A_k x$, k=1,...,p. Each process must have the entire vector y before it can compute the estimate of λ and create a normalized eigenvector estimate. Since every process must share its portion of y with every other process, an *allgather* operation is in order.

The evolution of GPU technology has outperformed the traditional multi-core paradigms, introducing a new opportunity in the field of scientific computing. GPU based solutions have outperformed the corresponding sequential implementations by leaps and bounds. Particularly, for scientific computing applications, they provide a very valuable resource to parallelize and achieve high performance. This exercise focuses on analyzing its performance on Cuda, and the comparison of power method between CPU and GPU. The details of the CPU and GPU formats and parts of pseudo-code are described in the following section.

2.2 Useful Reference

GPUs (Graphics Processing Units) originally designed for graphics applications become popular for scientific computing and simulations. GPUs consist of multiprocessor elements that run under the shared-memory threads model. GPUs can run hundreds or thousands of threads in parallel and has its own DRAM. Therefore, GPUs are good at data-parallel processing. That is to say, GPU is suitable for addressing problems that can be expressed as the same operation executed on many data elements in parallel with high arithmetic intensity.

The common programming environments for GPUs are CUDA and OpenCL. In this lab experiment, we use CUDA as a parallel programming environment for GPUs. CUDA (Compute unified device architecture, see the reference on the course website for more details) is a general-purpose parallel computing platform and programming model for managing computations on the GPU. CUDA designed by NVIDIA allows developers to use C as a programming language on their GPUs. A CUDA program given in a file *.cu consisting of a host program and kernel functions is compiled by the NVIDIA-C-compiler (nvcc), which separates both program parts. The execution of a CUDA program starts by executing the host program that calls kernel functions to be processed in parallel on the GPU. After invoking a kernel function, the CPU continues to process the host program that might lead to the call of other kernel function. CUDA extends the C function declaration syntax to distinguish host and kernel functions. In particular, the CUDA-specific keyword **__global__** indicates that the function is a kernel that can be called from a host function to be executed on a GPU. The keyword **__device__** indicates that the function is a kernel to be called from another kernel or device function. A host function is declared using the keyword **__host__**. All functions without any keyword are host function by default.

To begin with, similar to the introduction of parallelization using MPI, we give some descriptions about how to compile the codes and run it in CUDA by a *HelloWorld*. And then, parts of codes about implementation of parallelized power method separately in CPU and GPUs also will be given. Some additional code should be added into our provided code by the student themselves to do the analysis.

Code for *HelloWorld* This code is complete, the students can preliminarily understand how to implement the CUDA codes.

```
// parallel HelloWorld using GPUs
// Simple starting example for CUDA program : this only works on
   arch 2 or higher
// Cong Xiao and Senlei Wang, Modified on Sep 2018

#include <stdio.h>
#include <stdlib.h>
#include <cuda.h>

#define N_THRDS    4 // Nr of threads in a block (blockDim)
#define N_BLKS    4 // Nr of blocks in a kernel (gridDim)

void checkCudaError(const char *error)
{
   if (cudaGetLastError() != cudaSuccess)
   {
      fprintf (stderr, "Cuda : %s\n",error);
      exit(EXIT_FAILURE);
   }
}

void checkCardVersion()
{
   cudaDeviceProp prop;
   cudaGetDeviceProperties(&prop, 0);
   checkCudaError("cudaGetDeviceProperties failed");

```

```
    fprintf(stderr,"This GPU has major architecture %d, minor %d
     \n",prop.major,prop.minor);
    if(prop.major < 2)
    {
       fprintf(stderr,"Need compute capability 2 or higher.\n");
       exit(1);
    }
}

___global___ void HelloworldOnGPU(void)
{
    int myid = (blockIdx.x * blockDim.x) + threadIdx.x;
    // Each thread simply  prints it's own string :
    printf( "Hello World, I am thread %d in block with index %d,
     my thread index is %d \n",
      myid, blockIdx.x, threadIdx.x);
}

int main(void)
{
    checkCardVersion();
    HelloworldOnGPU <<< N_BLKS, N_THRDS >>> ();
    cudaDeviceSynchronize(); // without using synchronization,
     output won't be shown

    return 0;
}
```

To make the task easier, we also provide a file to automatically complete the whole procedure, including the code compilation, code running in CUDA and outputs of the final results.

CUDA is supported by Nvidia GPUs. The current CUDA 8.0 implementation can be added to your environment as follows:
module load cuda80/toolkit.
An SGE job script **HelloWorld.job** to submit a CUDA application on a host with a GTX480 GPU could then look like this:

```
#!/bin/sh
#SBATCH --time=00:15:00
#SBATCH -N 1
#SBATCH -C GTX480
#SBATCH --gres=gpu:1

. /etc/bashrc
. /etc/profile.d/modules.sh
module load cuda80/toolkit
file='HelloWorld'
nvcc $file.cu -o $file
./$file
```

Submit the job by command:
sbatch HelloWorld.job,
the result will be in output file **slurm******.out**.

The option "SBATCH -C GTX480" specifies a node with a GTX480 GPU, while the option "SBATCH –gres=gpu:1" lets SLURM allocate the GPU for the job by setting environment parameter CUDA_VISIBLE_DEVICES to 0. Note that without the "–gres" option, SLURM by default sets CUDA_VISIBLE_DEVICES to value NoDevFiles, which causes the CUDA runtime system to ignore the GPU. More information about the implementation of GPU code on Das4 system can be found on the Das4 website: *https://www.cs.vu.nl/das4/gpu.shtml.*

Description of the sequential code: power_cpu.cu As a starting point we consider the code *power_cpu.cu* that solves the Power Method problem sequentially in CPU. It is accessible as *https://brightspace.tudelft.nl/d2l/le/content/36523/Home.* All the routine in this program are put together in one file. Some modes or functions are explained as follows,

1. *UploadArray* and *InitOne*. In these two routines, the specific matrix and the initial eigenvector are given, respectively.

2. *CPU_AvPoduct*. In this routine a new vector x_k for the k iteration step is calculated by multiplying the previously normalized vector x_{k-1} with the matrix A.

3. *CPU_NormalizedW*. In this routine, the new estimated vector x_k for the k+1 iteration step is normalized as Eq.**??**.

4. *CPU_ComputeLamda*. In this routine, the estimated largest eigenvalue of this matrix at the k iteration step is computed as Eq.5.

And the *RUNCPUPowerMthod* is the main code of the Power Method in CPU. In the next section, a parallel version of Power Method in GPU using CUDA will be described step by step.

2.3 Exercise

Building a parallel program using CUDA The sequential version of the code that solves a Power Method problem is provided as the *power_cpu.cu.* In order to make a parallel version from it that uses the CUDA library, modifications have to be made and code has to be added at several places. You will already be familiar with some of these steps, since they are similar to the exercises in the introduction to CUDA using *HelloWord*. In this part of the lab you will build a working parallel code step by step. Meanwhile you will become familiar with some additional functionality that is offered by the CUDA library. In addition to the initialization of the matrix and vector, e.g, *UploadArray* and *InitOne*, another routines, *CPU_AvPoduct*, *CPU_NormalizedW*, *CPU_ComputeLamda*, should be recoded to a parallellized version using CUDA. Here, four subroutines, e.g, *GPU_AvPoduct*, *GPU_FindNormW*, *GPU_NormalizedW*, and *GPU_ComputeLamda*, are provided to implement the Power Method in GPU. Some tips are given here. The Power method actually is a matrix-vector multiplication which is computed by parallelization in GPU. However, during the procedure of the Power Method, the normalization step for the vector as Eq.**??** forces us to collect all of the elements of the resultant vector k from the threads, which need us to transfer the data from GPU (device) to CPU (host), and then transfer the normalized vector from the CUP to the GPU. Because this transfer involves a small amount of data and the main parts of matrix-vector multiplication is efficiently implemented in GPU. This feature makes the Power Method very suitable for GPU. In this section, you should understand the provided four subroutines and then finish the missing part of the main code, *power_gpu.cu.* The Workflow for parallelization of Power Method on GPUs is summarized by a flow chart in Fig.2.

Performance analysis After building a parallel code you now will perform some "experiments" with it. The main goal of these experiments is to analyze the performance of GPU. This can be done on a variety of ways, i.e., adjust

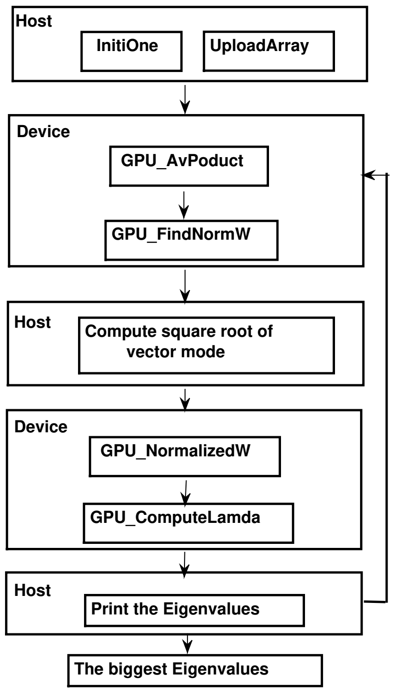


Fig. 2: **Workflow for parallelization of Power Method on GPUs**

the size of matrix or the number of threads per block. This implies that it is important to know how much time the program spends in the various phases, and how this time depends on the various parameters like number of threads and problem size. In addition, some comparisons between the implementation of Power Method in CPU and GPU also should be conducted to further investigate the advantages of the GPU.

1. Step 1: The essence of parallelized Power Method on GPUs is Matrix-Vector multiplication. The provided codes are about doing matrix-vector multiplications using CUDA with **shared** memory, a type of on-chip memory that is much faster than the **global** memory of the device, however, the limited size of share memory requires an overwhelming programming effort for utilizing the shared memory in practice. You should explore the performance by use of those two different memory access and compare the speed differences.
2. Step 2: Measure the execution time for a matrix A with different size, n=50, 500, 2000, and different number threads per block, 32, 64, 128.
3. Step 3: Calculate the speedups, (a) excluding the memory-copy between the CPU and GPU; (b) including the time of memory-copies.
4. Step 4: Explain the different performance results from the previous experiments.