Solving Secular Equations stably and Efficiently,

Ren-Cang Li, April 1993 [1]

Dimitri Bouche, Gauthier Schweitzer

1 Problem presentation

Our objective is to solve the following secular equation :

$$f(x) = \rho + \sum_{i=1}^{n} \frac{\zeta_i^2}{\delta_i - x} \tag{1}$$

Its roots correspond the eigenvalues of a diagonal matrix $D = diag(\delta_1, ...\delta_n)$ with a rank-1 perturbation :

$$D + \frac{zz^T}{\rho}$$

We set $z = (\zeta_1, ...\zeta_n)$. Without loss of generality, we assume that $(\delta_1, ...\delta_n)$ is such that

$$\delta_1 < \delta_2 < ... < \delta_n$$

and every $\zeta_i \neq 0$.

f has n roots, $n-1$ are in $[\delta_k, \delta_{k+1}]$ while one is in $[\delta_n, +\infty]$ (if $\rho > 0$ and we can assume it without loss of generality). The purpose of our work is to find numerically these roots by using Gragg's aglorithm. All of our formulas are derived from [1]. We will first present the algorithm and the main formulas that we have been using. After that, we will discuss its implementation using CUDA. Finally, we will show some results comparing the performance of the following procedure on CPU or on GPU.

2 Algorithm

2.1 Iteration Formulas

All of the following formulas come directly from the article by Li [1].

2.1.1 Interior roots

For a given y, we want to perform a second order equation to determine η so that $y+\eta$ is closer to the root than y. We set :

$$\begin{aligned}\Delta_k &= \delta_k - y \\ \Delta_{k+1} &= \delta_{k+1} - y \\ a &= (\Delta_k + \Delta_{k+1})f(y) - \Delta_k\Delta_{k+1}f'(y) \\ b &= \Delta_k\Delta_{k+1}f(y)\end{aligned}$$

η is given by :

$$\eta = \frac{a - \sqrt{a^2 - 4bc}}{2c} \quad if \quad a \leq 0 \tag{2}$$

$$\eta = \frac{2b}{a + \sqrt{a^2 - 4bc}} \quad if \quad a > 0 \tag{3}$$

c is a degree of freedom that we can use for the interpolation. With Gragg's procedure, c is chosen so that the second order derivative of the interpolation coincides with the second order derivative of the secular equation in y. Therefore, we obtain the following formula for c :

$$c = f(y) - (\Delta_k + \Delta_{k+1})f'(y) + \Delta_k\Delta_{k+1}\frac{f''(y)}{2} \tag{4}$$

Combining (2) and (4), we can compute the increment η that should be added to y.

2.1.2 Exterior root ($k = n$)

We need to adapt the procedure to find the exterior root. The interpolation procedure is similar to the case $k = n-1$, so :

$$\begin{aligned}\Delta_{n-1} &= \delta_{n-1} - y \\ \Delta_n &= \delta_n - y \\ a &= (\Delta_{n-1} + \Delta_n)f(y) - \Delta_{n-1}\Delta_n f'(y) \\ b &= \Delta_{n-1}\Delta_n f(y) \\ c &= f(y) - (\Delta_{n-1} + \Delta_n)f'(y) + \Delta_{n-1}\Delta_n\frac{f''(y)}{2}\end{aligned}$$

The increment η is given by a modified version of (2) :

$$\eta = \frac{a + \sqrt{a^2 - 4bc}}{2c} \quad si \quad a \geq 0 \tag{5}$$

$$\eta = \frac{2b}{a - \sqrt{a^2 - 4bc}} \quad si \quad a < 0 \tag{6}$$

2.2 Initilization

2.2.1 Educated guess

In the article [1], the authors suggest initial values and give theoretical reasons why they should lead to a nice convergence rate.

Interior roots For interior roots, we set :

$$\Delta = \delta_{k+1} - \delta_k$$

Then, if $f\left(\frac{\delta_k+\delta_{k+1}}{2}\right) \geq 0$:

$$a = c\Delta + \left(\zeta_k^2 + \zeta_{k+1}^2\right), \quad b = \zeta_k^2\Delta$$

and if $f\left(\frac{\delta_k+\delta_{k+1}}{2}\right) < 0$:

$$a = -c\Delta + \left(\zeta_k^2 + \zeta_{k+1}^2\right), \quad b = -\zeta_k^2\Delta$$

Then, if we denote y the initial value, it is obtained by setting :

$$y = \begin{cases} \delta_k + \frac{a-\sqrt{a^2-4bc}}{2c} \text{ if } a \leq 0 \\ \delta_k + \frac{2b}{a+\sqrt{a^2-4bc}} \text{ if } a > 0 \end{cases}$$

Exterior root ($k = n$) We first define

$$g(x) = \rho + \sum_{j=1, j\neq k, k+1}^{n} \frac{\zeta_j^2}{\delta_j - x}, \quad h(x) = \frac{\zeta_k^2}{\delta_k - x} + \frac{\zeta_{k+1}^2}{\delta_{k+1} - x}$$

There are two cases :

1. If $f\left(\frac{\delta_n+\delta_{n+1}}{2}\right) \leq 0$
   - If $g\left(\frac{\delta_k+\delta_{k+1}}{2}\right) \leq -h(\delta_{n+1},$ then $y = \delta_n + z^T z/\rho$
   - else, then :

$$y = \begin{cases} \delta_n + \frac{a+\sqrt{a^2-4bc}}{2c} \text{ if } a \geq 0 \\ \delta_n + \frac{2b}{a-\sqrt{a^2-4bc}} \text{ if } a < 0 \end{cases} \tag{7}$$

     where

$$\Delta = \delta_n - \delta_{n-1}, \ a = -c\Delta + \left(\zeta_{n-1}^2 + \zeta_n^2\right), \ b = -\zeta_n^2\Delta, \ c = g\left(\frac{\delta_k + \delta_{k+1}}{2}\right)$$

2. If $f\left(\frac{\delta_n+\delta_{n+1}}{2}\right) > 0$, then y is computed as in (7).

2.2.2 Random guess

We also tried a method consisting in pseudo-randomly picking a number in $[\delta_k, \delta_{k+1}]$. On the exterior part, the upper bound δ_{n+1} of the segment on which we draw randomly is the same as the one given for the 'educated guess'.

3 Implementation in C

We have developped two main algorithms, one using the CPU and one using the GPU. We directly used the formulas highlighted previously, coming from [1]. One difference is the stopping criterion. We have used a simpler criterion than the one stated in the article : we use a threshold on the absolute value of the spectral function computed at the estimated root.

3.1 CPU

We won't go very deep into details for this implementation. One should however mention that we have built two sub-algorithms using CPU. The first one uses *float* variables while the second one has double precision and uses *double*. We will see in the results part that these specificities have strong implications, both for the running time and for the precision of the estimated roots.

3.2 GPU

Why using a GPU Secular equations solving is a task that has two features that make it very interesting for GPU computing :

— it requires important computational power ;

— it is by essence highly parallelizable.

The second point is more interesting to discuss. To solve secular questions, one has to solve n different problems, each one corresponding to the computation of one root. All of these subproblems are independent since one does not need the root on one interval to compute a root on another one. Therefore, it appears natural to parallelize these computations. Actually, parallelization goes even further.

__global__ functions We have four distinct kernels, each of them corresponding to a task to parallelize. These kernels are launched one after another :

1. The first kernel *square_kernel* computes the square of each ζ_i and the squared norm of the ζ vector. It uses a standard grid dimension $<< 1024, 512 >>$ to take full profit of the NVIDIA 1080 GPU. Therefore, each thread is computing the square of one coordinate and is doing an AtomicAdd (to prevent concurrent writing) to obtain the square ;
2. The second kernel *initialize_x0_kernel* is used to initialize the root-finding process on each of the intervals. Therefore, each thread has a given interval on which it performs calculations to set the initial value of λ. Once more, the grid is $<< 1024, 512 >>$;
3. Finally, the last kernel is the main one *find_roots_kernel*. Starting from the different inital values, it performs Gragg's algorithm to obtain the roots. As explained, each thread computes one root. Grid size is $<< 1024, 512 >>$.

Other functions The rest of the functions that we are using are either standard host functions (called from the host to be performed on the host), or __*device*__ ones (called from the device to be performed on the device).

Access to memory For many workloads, the performance benefits of parallelization are hindered by the large and often unpredictable overheads of launching GPU kernels and of transferring data between CPU and GPU. Therefore, we tried to limit as much as possible communications between the host and the device. We mainly used registers memory, that is the fastest one. These variables stored in registers are local. We have also used shared variables for values that are accessed by multiple threads.

4 Results

4.1 The scripts that we have used

4.1.1 Testing the different algorithms individually

— main_cpu_double.cu : CPU is used, with a double precision to compute the n roots of a secular equation, n being given by the user.

— main_cpu_float.cu : CPU is used, with a single precision to compute the n roots of a secular equation, n being given by the user

— main_gpu.cu : GPU is used, with a single precision to compute the n roots of a secular equation, n being given by the user. For each n being tested, the GPU is warmed-up before the first iteration (method discussed later on). No shared variables are used in this version.

4.1.2 Comparing the performance

Directly in the terminal

— comp_console.cu : Both CPU and GPU with single precision are used to compute the n roots of a secular equation, n being given by the user. The differential in performance can be seen immediately in the console (running time and magnitude of the loss). This version does not include shared variables.

Generating a csv that can be graphically read in a Python notebook

— comp_table.cu : Both CPU and GPU with single precision are used to compute the n roots of a secular equation. A range of n is given by the user and performance (running time and magnitude of the error) is stored on a csv file ('result.csv'). To compare them on a fair basis, the user can choose to run the test several time. For each n being tested, the GPU is warmed-up before the first iteration. To try with high values of n, the user can also choose not to compute the roots with the CPU (only GPU). This version of the GPU procedure does not use shared memory.

— double.cu : Performs the same task with double precision for the CPU (output is 'result_double.csv')

— memory.cu : Performs the same task with the GPU, in a version using shared memory (output is 'result_mem.csv')

— initialization.cu : Performs the same task with CPU, the algorithm being initialized pseudo randomly as described before (output is 'result_init.csv')

4.2 Results

4.2.1 Method

As described before, we have built programs that generate a csv file to track the performance of the algorithm. We could not run all of these programms once for all with a large set of n to test. We we tried to do so, our job has been killed several times, resulting in a loss of all of our results. Therefore, we have decided to run the programms several times on smaller spans of n and then aggregate the results in a Python Notebook ('Graphs'). In the end, we have obtained 27 csv files, corresponding to 2537 iterations of an algorithm.
To make sure that the roots that we had found were the right ones, we built a Jupyter Notebook ('PythonCheckForLi') that implements Li algorithm, plots the function and computes the roots using Gragg's algorithm. Our results were very similar.
To compare GPU to CPU on a fair basis, we have used the 'wakeup' kernel that we designed. Without using it, the first iteration of the GPU always took much longer than the other, no matter the n. This kernel simply uses 1024 threads to perform the addition of a constant integer to each element of a size-1024 array.

4.2.2 Graphs

On the graphs, the label correspondence is the following :

— GPU : use of GPU for computation, only registered memory used (no shared memory), single precision

— CPU : use of CPU for computation, single precision

— CPU double precision : use of CPU for computation, double precision

— GPU shared memory : use of GPU computation, use of registered memory and shared memory for values that are used by multiple threads

— CPU random initialisation : use of CPU for computation, but instead of setting the initialization with the article formulas, it is set pseudo randomly on each of the intervals.

Running times In terms of running times (figure 1), several comments can be made :

— General remarks :

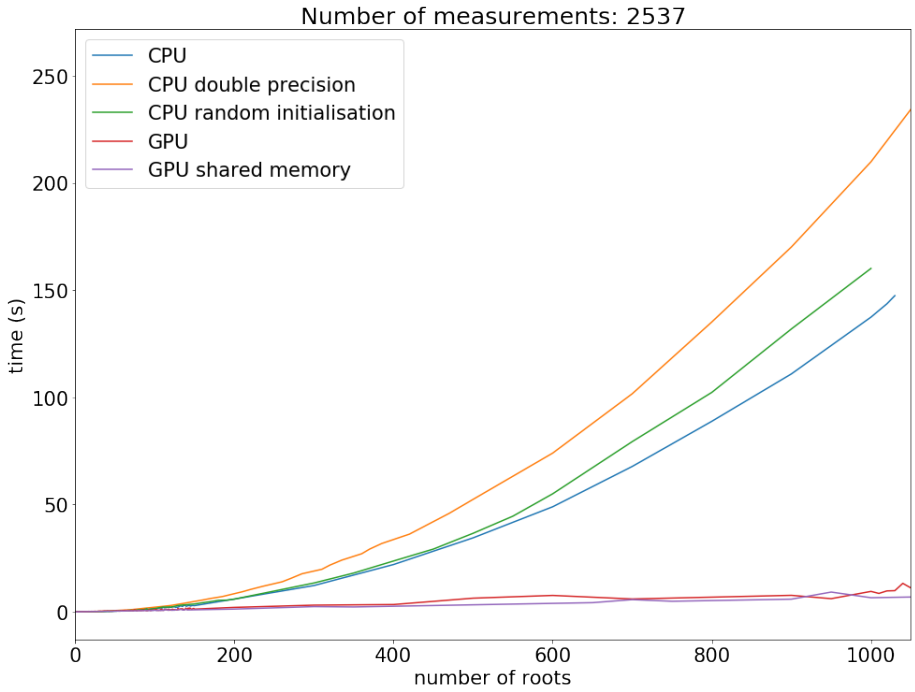
  ⋆ No matter if we use shared memory or not, GPU is faster than CPU in the long run. For $n = 1000$, it appears that the CPU takes around 100 seconds to find all the roots while the GPU needs less than 10 seconds, generating a considerable speed-up. For very small n, however ($n \leq 50$), CPU is faster than GPU, due to overhead costs.

— Focus on the CPU :

  ⋆ Running time increases with n at a speed that is higher than the linear one (not a line on the graph (a)) but lower than exponential (not a line on the semilog graph (d)) ;

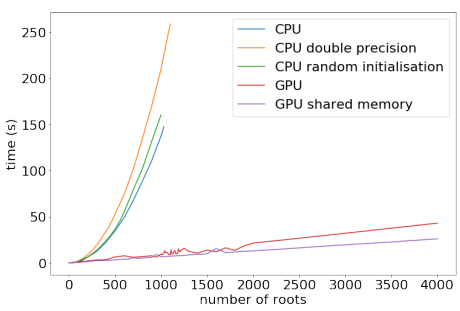
  ⋆ Double-precision increases running time, but does not double it (+62% running time compared to the standard CPU)

(a) Running times of the different algorithms as a function of n

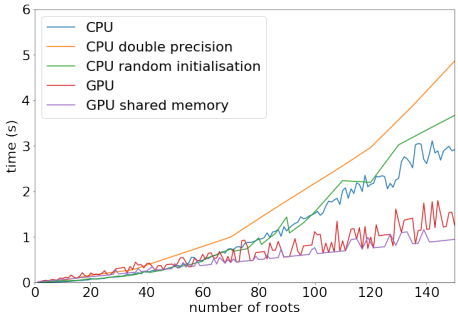


(b) Larger scale

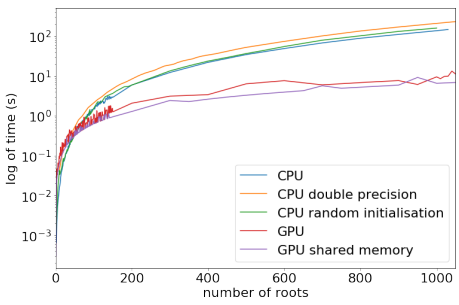
(c) Focus on $n \in [0, 150]$



(d) Semilog scale



(e) Focus on the GPU



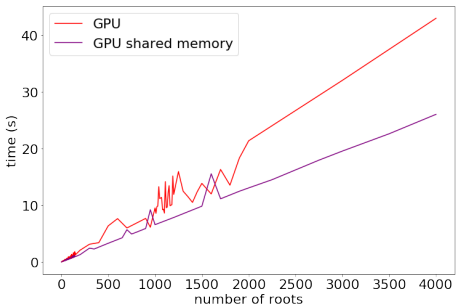


FIGURE 1 – Analysis of running times for the different algorithms

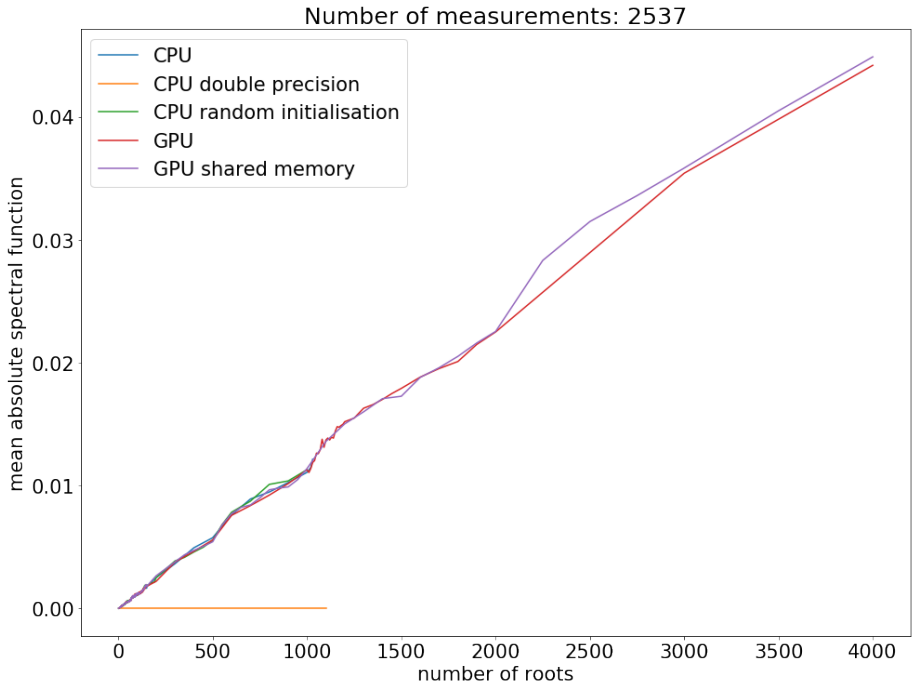
⋆ A random initialisation increases running time (+15% running time compared to the standard CPU)

— Focus on the GPU :

⋆ Running time increases with n at a speed that is linear. We do not observe any particular threshold at the values of n that are key to our hardware, namely the powers of 2 : 512 or 1024.

⋆ Using the shared memory decreases running time. When one uses shared memory, running time is on average 0.79 the time without this use, keeping n constant.

FIGURE 2 – Absolute value of spectral function for the different algorithms as a function of n



Final value of the spectral function Here we focus on the precision of the estimated roots. For that, we compute the mean absolute value of the spectral function computed at each estimated root. Therefore, the smallest the best.
The CPU with double precision is the only algorithm that is able to compute the roots very precisely, even when n is big. However, as we have seen earlier, this comes at the price of a higher running time and an increased use of memory. The other algorithm evolve in the same way, with a mean absolute error increasing linearly with the number of roots to compute. This is a direct consequence of the simple precision that is used. The diminishing performance of simple precision algorithm with increasing n might be due to the fact that the spectral function is getting flatter around the root when n increases.
One can also observe that the CPU initialized from random values is as good as

the others. Therefore, in this setting, a random generation mostly leads to an increased running time, but not to worse prediction.

Références

[1] Ren-Cang Li. “Solving secular equations stably and efficiently”. In : (1993).