Efficient high degree polynomial root finding using GPU

Article *in* Journal of Computational Science · January 2017

DOI: 10.1016/j.jocs.2016.12.004

4 authors:

Kahina Ghidouche
Université de Béjaïa
3 PUBLICATIONS **2** CITATIONS
SEE PROFILE

Abderrahmane Sider
Université de Béjaïa
7 PUBLICATIONS **8** CITATIONS
SEE PROFILE

Raphaël Couturier
University Bourgogne Franche-Comté
178 PUBLICATIONS **873** CITATIONS
SEE PROFILE

Christophe Guyeux
University of Franche-Comté
184 PUBLICATIONS **736** CITATIONS
SEE PROFILE

Efficient high degree polynomial root finding using GPU

Kahina Ghidouche[a], Abderrahmane Sider[a], Raphaël Couturier[b,*], Christophe Guyeux[b]

[a] *Laboratoire LIMED, Faculté des sciences exactes, Université de Bejaia, 06000 Bejaia, Algeria*
[b] *FEMTO-ST Institute, Univ. Bourgogne Franche-Comté (UBFC), France*

Abstract

Polynomials are mathematical algebraic structures that play a great role in science and engineering. Finding the roots of high degree polynomials is computationally demanding. In this paper, we present the results of a parallel implementation of the Ehrlich-Aberth algorithm for the root finding problem for high degree polynomials on GPUs using CUDA and on multi-core processors using OpenMP. The main result we achieved is to solve high degree polynomials (up to 1,000,000) efficiently. We also compare the Ehrlich-Aberth method and the Durand-Kerner one on both full and sparse polynomials. Accordingly, our second result is that the first method is much faster and more efficient. Last, but not least, an original proof of the convergence of the asynchronous implementation for the EA method is produced.

Keywords: Polynomial root finding, Iterative methods, Ehrlich-Aberth, Durand-Kerner, GPU

1. The problem of finding the roots of a polynomial

Polynomials are mathematical algebraic structures used in science and engineering to capture physical phenomena and to express any outcome in the form of a function of some unknown variables. Formally speaking, a polynomial $p(x)$

*Corresponding author
Email addresses: kahina.ghidouche@univ-bejaia.dz (Kahina Ghidouche), ar.sider@univ-bejaia.dz (Abderrahmane Sider), raphael.couturier@univ-fcomte.fr (Raphaël Couturier), christophe.guyeux@univ-fcomte.fr (Christophe Guyeux)

of degree n having n coefficients in the complex plane $\mathbb{C}$ is :

$$p(x) = \sum_{i=0}^{n} a_i x^i, a_0 \neq 0. \tag{1}$$

The root finding problem consists in finding all the n values of the variable x for which *p(x)* is nullified. Such values are called zeros of p. If zeros are α_i, *i=1,...,n*, the $p(x)$ can be written as :

$$p(x) = a_n \prod_{i=1}^{n} (x - \alpha_i), a_n \neq 0. \tag{2}$$

The problem of finding a root is equivalent to that of solving a fixed-point problem. To observe this, consider the fixed-point problem of finding the n-dimensional vector X such that :

$$X = g(X)$$

where $g : \mathbb{C}^n \longrightarrow \mathbb{C}^n$. We can easily rewrite this fixed-point problem as a root-finding problem by setting $f(X) = X - g(X)$ and likewise we can recast the root-finding problem into a fixed-point problem by setting :

$$g(X) = f(X) - X.$$

It is often impossible to solve such nonlinear equation root-finding problems analytically. When this occurs, we turn to numerical methods to approximate the solution. Generally speaking, algorithms for solving problems can be divided into two main groups: direct methods and iterative methods.

Direct methods only exist for $n \leq 4$, solved in closed form by G. Cardano [1] in the mid-16th century. However, N. H. Abel [2] in the early 19th century proved that polynomials of degree five or more could not, in general, be solved by direct methods. Since then, mathematicians have focused on numerical (iterative) methods such as the famous Newton [3], and the Graeffe one [4].

Later on, with the advent of electronic computers, other methods have been developed such as Jenkins-Traub [5], Larkin [6], Muller [7], and several others

for the simultaneous approximation of all the roots, starting with the Durand-Kerner (DK) method [8, 9]:

$$DK : z_i^{k+1} = z_i^k - \frac{p(z_i^k)}{\prod_{i \neq j}(z_i^k - z_j^k)}, i = 1, ..., n, \tag{3}$$

where z_i^k is the i^{th} root of the polynomial p at the iteration k.

This formula was mentioned for the first time by Weiestrass [10] as part of the fundamental theorem of Algebra and was rediscovered by Ilieff [11], Docev [12], Durand [8], and Kerner [9]. Another method, discovered by Borsch-Supan [13], and also described and brought in the following form by Ehrlich [14] and Aberth [15], uses a different iteration formula given as:

$$EA : z_i^{k+1} = z_i^k - \frac{1}{\frac{p'(z_i^k)}{p(z_i^k)} - \sum_{i \neq j} \frac{1}{(z_i^k - z_j^k)}}, i = 1, ..., n, \tag{4}$$

where $p'(z)$ is the polynomial derivative of p evaluated in the point z.

Aberth, Ehrlich, and Farmer-Loizou [16] have proven that the Ehrlich-Aberth method (EA) has a cubic order of convergence for simple roots whereas the Durand-Kerner has a quadratic order of convergence. Moreover, the convergence time of iterative methods drastically increases like the degrees of high polynomials, while it is expected that the parallelization of these algorithms will reduce the execution times.

Many authors have dealt with the parallelization of simultaneous methods, *i.e.*, that find all the zeros simultaneously. Freeman [17] implemented and compared DK, EA, and another method of the fourth order proposed by Farmer and Loizou [16], on an 8-processor linear chain, for polynomials of degree 8. The third method often diverges, but the first two methods have a speed-up factor equal to 5.5. Later, Freeman and Bane [18] considered asynchronous algorithms, in which each processor continues to update its approximations even though the latest values of other roots have not yet been received from the other processors. In contrast, synchronous algorithms wait for the computation of all roots at a given iterations before making a new one. Couturier *et al.* [19] proposed two methods of parallelization for a shared memory architecture and

for a distributed memory one. They were able to compute the roots of sparse polynomials of degree 10,000 in 430 seconds with only 8 personal computers and 2 communications per iteration. Compared to sequential implementations where it takes up to 3,300 seconds to obtain the same results, the authors' work experiment shows an interesting speedup.

To our knowledge, no other work has been published regarding the parallelization of this method or other ones before the emergence of the Compute Unified Device Architecture (CUDA) [20], a parallel computing platform and a programming model invented by NVIDIA. The computing power of GPUs (Graphics Processing Units) has exceeded that of CPUs. However, CUDA adopts a totally new computing architecture to use the hardware resources provided by a GPU in order to offer a stronger computing ability to the massive data computing. First, Ghidouche *et al.* [21] proposed an implementation of the Durand-Kerner method for sparse polynomials on GPU. Their main result shows that a parallel CUDA implementation is much faster than the sequential implementation on a single CPU.

In this paper, we report on our ongoing research aiming at proposing, implementing, and improving the EA iterative function and the implementation of the Ehrlich-Aberth method to solve high degree polynomials accurately and rapidly on GPUs. The main contributions of this research work are:

- An adaptation of the exponential logarithm to improve the classical Ehrlich-Aberth iterative method, in order to be able to solve sparse and full polynomials of high degree.
- A parallel implementation of Ehrlich-Aberth method on GPU for sparse and full polynomials of high degree up to $1,000,000$. This parallel implementation finds roots quite rapidly.
- An original proof of the convergence of the asynchronous implementation for the EA method.

The article is organized as follows. Initially, we recall the Ehrlich-Aberth

method in Section 2. Improvements for the Ehrlich-Aberth method are proposed in Section 3. Our convergence proof of the EA asynchronous method is presented in Section 4. Research works related to the implementation of simultaneous methods using a parallel approach are presented in Section 5. In Section 6, we propose a parallel implementation of the Ehrlich-Aberth method on GPU and we discuss it. Section 7 presents and investigates our implementation and experimental study results. Section 8 presents a data analysis collected in the experiments. Finally, Section 9 concludes this article and gives some hints for future research directions in this topic.

2. The Ehrlich-Aberth method

It is a cubically convergent iterative method to find zeros of polynomials as proposed by O. Aberth [15] whose iterative function is:

$$EA2 : z_i^{k+1} = z_i^k - \frac{\frac{p(z_i^k)}{p'(z_i^k)}}{1 - \frac{p(z_i^k)}{p'(z_i^k)} \sum_{j=1, j\neq i}^{j=n} \frac{1}{(z_i^k - z_j^k)}}, i = 1, ..., n \quad (5)$$

It can be noticed that this equation is equivalent to Eq. 4, but we prefer the latter one, because we can use it to improve the Ehrlich-Aberth method and find the roots of high degree polynomials. More details are given in Section 3.

As for any iterative method, a convergence criterion must be checked after each iteration to decide whether to perform another step or to terminate the computations. When the termination happens, it means that the roots are sufficiently stable, *i.e.*, very close to the actual zeros. In the following, we consider that the method converges sufficiently when:

$$\forall i \in [1, n]; \left| \frac{z_i^k - z_i^{k-1}}{z_i^k} \right| < \xi \quad (6)$$

where $|.|$ stands for the absolute value and ξ is the error threshold.

The definition of a polynomial $p(z)$ is done by setting each of the n complex coefficients a_i. According to the *sparse* or *full* setting, some or all of the coefficients are set deterministically and not randomly so as to have reproducible and comparable results. More details are given in the Experiments section.

Finally, as for any iterative method, we need to choose n initial guess points $z_i^0, i = 1, ..., n$. The initial guess is very important since the number of steps needed by the iterative method to reach a given approximation strongly depends on it. In [15] the Ehrlich-Aberth iteration is started by selecting n equi-spaced points on a circle of center 0 and radius σ, where σ is an upper bound to the moduli of the zeros. Later, Bini *et al.* [22] improved this choice by selecting complex numbers along different circles which relies on the result of [23]:

$$\sigma = \frac{u+v}{2}; u = \frac{\sum_{i=1}^{n} u_i}{n.max_{i=1}^{n} u_i}; v = \frac{\sum_{i=0}^{n-1} v_i}{n.min_{i=0}^{n-1} v_i}; \tag{7}$$

where:

$$u_i = 2.|a_i|^{\frac{1}{i}}; v_i = \frac{|\frac{a_n}{a_i}|^{\frac{1}{n-i}}}{2}. \tag{8}$$

We build on this latter work and adopt it for the starting zeros for our implementation.

3. Improving the Ehrlich-Aberth method for high degree polynomials with the exp-log formulation

With high degree polynomials, the Ehrlich-Aberth method implementation suffers from overflow problems. This situation occurs, for instance, in the case where a polynomial, having positive coefficients and a large degree, is computed at a point ξ where $|\xi| > 1$ ($|\xi|$ stands for the complex modulus of ξ). Indeed, the limited number in the mantissa of floating point representations makes the computation of $p(z)$ wrong when z is large. For example $(10^{50}) + 1 + (-10^{50})$ will give the wrong result of 0 instead of 1. Consequently, one cannot compute the roots for high degree polynomials. This problem was discussed earlier in [24] for the Durand-Kerner method. The authors proposed to use the logarithm and the exponential of a complex in order to compute the power at a high exponent. We noticed also that floats are exploited rapidly, when the arithmetic operations are performed by a processor which has the following characteristics:

- **Float**: real in the $[-3.40282e+38, +3.40282e+38]$ interval. The mantissa contains 6 decimal numbers.

- **Double**: real in the $[-1.79769e+308, +1.79769e+308]$ interval. The mantissa contains 15 decimal numbers.

- **Long Double**: real in the $[-1.18973e+4932, +1.18973e+4932]$ interval. The mantissa contains 33 decimal numbers.

Using the logarithm (Eq. 9) and the exponential (Eq. 10) operators, we can replace any multiplications and divisions with additions and subtractions. Consequently, computations manipulate lower absolute values and the roots for large polynomial degrees can be looked for successfully [24].

$$\forall (x,y) \in \mathbb{R}^{*2}; \ln(x+i.y) = \ln(x^2+y^2)2 + i.\arcsin(y\sqrt{x^2+y^2})_{]-\pi,\pi[} \quad (9)$$

$$\forall (x,y) \in \mathbb{R}^{*2}; \exp(x+i.y) = \exp(x).\exp(i.y) \quad (10)$$

$$= \exp(x).\cos(y) + i.\exp(x).\sin(y) \quad (11)$$

Applying this solution to the iteration function Eq. 5 of Ehrlich-Aberth method, we obtain the following iteration function with exponential and logarithm:

$$EA.EL : z_i^{k+1} = z_i^k - \exp\left(\ln\left(p(z_i^k)\right) - \ln\left(p'(z_i^k)\right) - \ln\left(1 - Q(z_i^k)\right)\right), \quad (12)$$

where:

$$Q(z_i^k) = \exp\left(\ln(p(z_i^k)) - \ln(p'(z_i^k)) + \ln\left(\sum_{i \neq j}^{n} \frac{1}{z_i^k - z_j^k}\right)\right) i = 1, ..., n, \quad (13)$$

This solution is applied when the root excepts the circle unit, represented by the radius R evaluated in C language as :

$$R = exp(log(DBL\_MAX)/(2 * n)); \quad (14)$$

where `DBL_MAX` stands for the maximum representable `double` value.

4. Asynchronous convergence proof for the Ehrlich-Aberth method

Let us introduce the fixed point application T associated to the Ehrlich-Aberth method, as follows:

$$\forall i \in \{1, \ldots, n\}, T_i(z^k) = z_i^k - \frac{\frac{p(z_i^k)}{p'(z_i^k)}}{1 - \frac{p(z_i^k)}{p'(z_i^k)} \sum_{j=1, j\neq i}^{j=n} \frac{1}{(z_i^k - z_j^k)}}.$$

Let us denote by $||z|| = \max_{1\leq i\leq n} |z_i|$ on $\mathbb{C}^n$, and z^* the roots vector of the polynomial p. Let us first establish the following lemma [24].

Lemma 1 *In case of single roots, the fixed point application T associated to the Ehrlich-Aberth method is a contraction for $||.||$, at least in a close neighborhood of z^*.*

PROOF In case of single roots, we can establish at least a quadratic convergence. Indeed, let us consider $z \in \mathbb{C}^n$, then we have:

$$\begin{aligned} T_i(z) - T_i(z^*) &= T_i(z) - z_i^* \\ &= z_i - z_i^* - \frac{\frac{p(z_i^k)}{p'(z_i^k)}}{1 - \frac{p(z_i^k)}{p'(z_i^k)} \sum_{j=1, j\neq i}^{j=n} \frac{1}{(z_i^k - z_j^k)}} \\ &= z_i - z_i^* - \frac{1}{\frac{p'(z_i^k)}{p(z_i^k)} - \sum_{j=1, j\neq i}^{j=n} \frac{1}{(z_i^k - z_j^k)}}. \end{aligned}$$

We now define

$$f(x) = \frac{p(x)}{\prod_{j=0;\, j\neq k}^{n} (x - z_j)},$$

which is such that:

$$\begin{aligned} &\frac{p'(x)}{p(x)} - \sum_{j=0;\, j\neq k}^{n} \frac{1}{x - z_j} \\ &= \frac{d}{dx} \big(\ln |p(x)| - \sum_{j=0;\, j\neq k}^{n} \ln |x - z_j| \big) \\ &= \frac{d}{dx} \ln |f(x)| \\ &= \frac{f'(x)}{f(x)}. \end{aligned}$$

Then

$$\begin{aligned} T_i(z) - T_i(z^*) &= z_i - z_i^* - \frac{f(z_i)}{f'(z_i)} \\ &= \frac{f'(z_i)(z_i - z_i^*) - f(z_i)}{f'(z_i)} \\ &= \frac{f'(z_i)(z_i - z_i^*) - (f(z_i) - f(z_i^*))}{f'(z_i)} \end{aligned}$$

By using the second order Taylor polynomial for f, we have

$$|f'(z_i)(z_i - z_i^*) - (f(z_i) - f(z_i^*))\,| \leq \gamma_i |z_i - z_i^*|^2,$$

where $\gamma_i = \max_{z \in U_{z_i^*}} \dfrac{|f''(z)|}{2}$, in a given neighborhood $U_{z_i^*}$ of z^*. Let γ be the minimum of these $\gamma_i, 1 \leq i \leq n$. Additionally, f being a polynomial, this is the case too for f', and as we can consider that p has at least one root, then f' has a finite number of roots. As a consequence, we can find an open ball centered on z^* of radius r_i and a constant $\rho_i > 0$ such that, for z_i in $\mathcal{B}_i(z_i^*, r_i) \cap U_{z_i^*}$, we have $\rho_i \leq |f'(z_i)|$. On this ball, we thus have:

$$|T_i(z) - z_i^*| \leq \frac{\gamma}{\rho_i} |z_i - z_i^*|^2,$$

and so, for all z in $\mathcal{B}(z^*, r) \cap \Pi_{1 \leq i \leq n} U_{z_i^*}$, $r = \min_{1 \leq i \leq n} r_i$, and for $\rho = \max_{1 \leq i \leq n} \rho_i$, we have:

$$||T(z) - z^*|| \leq \frac{\gamma}{\rho} ||z - z^*||^2,$$

and the fixed point application will be a contraction mapping if

$$\frac{\gamma}{\rho} ||z - z^*|| \leq 1 \Leftrightarrow ||z - z^*|| \leq \frac{\rho}{\gamma}.$$

As a conclusion, T is a contraction mapping in the open ball $\mathcal{B}(z^*, \dfrac{\rho}{\gamma})$ intersected by the neighborhood $\Pi_{1 \leq i \leq n} U_{z_i^*}$ of z^*.

We can now deduce that:

Theorem 1 *All asynchronous algorithms associated to the fixed point application T defined previously, and starting at a sufficiently low distance to z^*, will converge to the roots of the polynomial p in the single roots case.*

PROOF Due to Lemma 1, the conditions for asynchronous convergence of fundamental theorem presented page 329 in [25] are satisfied.

5. Implementation of simultaneous methods in a parallel computer

The main problem of simultaneous methods is that the time needed for convergence is increased when the degree of the polynomial is increased. The parallelization of these algorithms is expected to improve the convergence time. Authors usually adopt one of the two following approaches to parallelize root finding algorithms. The first approach aims at reducing the total number of iterations as in Miranker [26, 27], Schedler [28], and Winograd [29]. The second approach aims at reducing the computation time per iteration, as reported, *e.g.*, in [30, 31, 32, 33].

There are many schemes for the simultaneous approximation of all roots of a given polynomial. Several works on different methods and issues of root finding have been reported in [34, 35, 36, 37, 38]. However, the Durand-Kerner and the Ehrlich-Aberth methods are the most practical choices among them [39]. These two methods have been extensively studied for parallelization due to their intrinsic parallelism, *i.e.*, the computations involved in both methods have some inherent parallelism that can be suitably exploited by SIMD machines. Moreover, they have a fast rate of convergence (quadratic for the Durand-Kerner and cubic for the Ehrlich-Aberth). Various parallel algorithms reported for these methods can be found in [40, 17, 18, 41, 32]. Freeman and Bane [18] presented two parallel algorithms on a local memory MIMD computer with the compute-to communication time ratio O(n). However, their algorithms require each processor to communicate its current approximation to all other processors at the end of each iteration (synchronous). Therefore they cause a high degree of memory conflict. Recently the author in [31] proposed two versions of parallel algorithm for the Durand-Kerner method, and the Ehrlich-Aberth method on a model of Optoelectronic Transpose Interconnection System (OTIS). The algorithms are mapped on an OTIS-2D torus using N processors. This solution needs N processors to compute N roots, which is not practical to find roots of high degree polynomials.

Finding polynomial roots rapidly and accurately is the main objective of

our work. With the advent of CUDA (Compute Unified Device Architecture), finding the roots of polynomials receives a new attention because of the new possibilities to solve higher degree polynomials in less time. In [21] we already proposed the first implementation of a root finding method on GPUs, that of the Durand-Kerner method. Their main result shows that a parallel CUDA implementation is 10 times faster than the sequential one, on a single CPU and for high degree polynomials of 48,000.

6. Implementation of the Ehrlich-Aberth method on GPU

In the following, we describe the parallel implementation on GPU of the Ehrlich-Aberth method, to find the roots of high degree polynomials. But first, we need to present the hardware architecture of GPUs and their programming model. Then, we show the main features of our GPU implementation.

6.1. Background on the GPU architecture

A GPU can be viewed as an accelerator for the data-parallel and intensive arithmetic computations. It draws its computing performances from the massive parallelism of its hardware and software architecture. Indeed, a GPU is composed of hundreds of Streaming Processors (SPs) organized in several blocks called Streaming Multiprocessors (SMs). It also has a memory hierarchy. A private read-write local memory per SP, fast shared memory and read-only constant and texture caches per SM, and a read-write global memory shared by all its SPs [20]. On a CPU equipped with a GPU, all the data-parallel and intensive functions of an application running on the CPU are off-loaded onto the GPU in order to accelerate their computations. A similar data-parallel function is executed on a GPU as a kernel by thousands or even millions of parallel threads, grouped together in a grid of thread blocks. Therefore, each SM of the GPU executes one or more thread blocks in a SIMD fashion (Single Instruction, Multiple Data) and in turn each SP of a GPU SM runs one or more threads within a block in SIMT fashion (Single Instruction, Multiple threads). With

the SIMT model, at any given clock cycle, the parallel threads execute the same instruction of a kernel, but each of them operates on different data. GPUs only work on data filled in their global memories and the final results of their kernel executions must be communicated to their CPUs.

6.2. Background on the CUDA Programming Model

CUDA, an acronym for Compute Unified Device Architecture, is a parallel computing architecture developed by NVIDIA [20] for GPUs. The unit of execution in CUDA is called a thread. Each thread executes the same kernel by running on the streaming processors in parallel. In CUDA, a group of threads that are executed together is called a thread block, and the computational grid consists of a grid of thread blocks. Each GPU multiprocessor executes one or more thread blocks in SIMD fashion and in turn each core of the multiprocessor executes one or more threads within a block. Additionally, threads in the same thread block may use shared memory and coordinate their execution through synchronization points. In contrast, within a grid of thread blocks, there is no synchronization at all between blocks. The GPU only works on data filled in the global memory and the final results of the kernel executions must be transferred out of the GPU. In the GPU, the global memory has lower bandwidth than the shared memory associated to each multiprocessor. Thus, as a rule of thumb, with CUDA programming, it was long thought necessary to design carefully the arrangement of the thread blocks in order to ensure a low latency and a proper use of the shared memory, but this has been recently downplayed. As for the global memory accesses, it should be minimized.

6.3. Parallel implementation with CUDA

In Algorithm 1 we show the key points for finding roots with the Ehrlich-Aberth method on GPU. P, $P^{'}$, and Z stand for the polynomial to solve, the derivative of P, and the root's solution vector, respectively.

After the initialization step, all data of the root finding problem must be copied

Algorithm 1: CUDA Algorithm to find roots with the Ehrlich-Aberth method

Input: Z^0 (Initial roots vector), ε (Error tolerance threshold), P (Polynomial to solve), $P^{'}$ (Derivative of P), n (Polynomial degree), $Error$ (Maximum value of stop condition)

Output: Z (Solution roots vector), Z^{Prev} (Previous solution roots vector)

1 Initialization of the parameters of roots finding problem (P, $P^{'}$, Z^0);
2 Allocate and copy initial data to the GPU global memory;
3 **while** $Error > \varepsilon$ **do**
4 $\quad Z^{Prev} = save(Z)$;
5 $\quad Z = Update(P, P^{'}, Z)$;
6 $\quad \Delta Z = Test\_Converge(Z, Z^{Prev})$;
7 $\quad Error = Max(\Delta Z)$;
8 **end**
9 Copy results from GPU memory to CPU memory;

from the CPU memory to the GPU global memory, because a GPU can only access and work on data present in its memories. Next, the algorithm uses an iterative method for finding roots, defined in the function *Update()* (in Algorithm 1, line 5). The iterative method used in this algorithm is the Ehrlich-Aberth method corresponding to Eq. 4. Before each iteration, the previous vector solution Z^{Prev} is saved using the *Save()* function (line 4. in Algorithm 1) because it is needed to measure the convergence of roots after each iteration (line 7). The iterative function terminates its computations when the error tolerance threshold ε has been reached, and/or all the roots have converged, which is checked in the function *Test_Converge()* in (Algorithm 1, line 6). Finally, the solution of the root finding problem is copied back from the GPU global memory to the CPU memory. All the data-parallel arithmetic operations inside the main loop `(while(...) do)` are executed as *kernels* by the GPU.

The Ehrlich-Aberth is based on arithmetic vector operations that are easy to implement on parallel computers and, thus, on GPU. Indeed, the GPU executes the vector operations as kernels and the CPU executes the sequential operations, launches the kernels, and supplies the GPU with data.

In order to implement the iterative Ehrlich-Aberth method in CUDA, it is possible to use the Jacobi scheme or the Gauss-Seidel one. With the Jacobi iteration, at iteration $k+1$ we need all the previous values z_i^k to compute the new values z_i^{k+1}, that is :

$$EAJ : z_i^{k+1} = z_i^k - \frac{\frac{p(z_i^k)}{p'(z_i^k)}}{1 - \frac{p(z_i^k)}{p'(z_i^k)} \sum_{j=1, j\neq i}^{j=n} \frac{1}{(z_i^k - z_j^k)}}, i = 1, ..., n. \tag{15}$$

With the Gauss-Seidel iteration, we have:

$$EAGS : z_i^{k+1} = z_i^k - \frac{\frac{p(z_i^k)}{p'(z_i^k)}}{1 - \frac{p(z_i^k)}{p'(z_i^k)} (\sum_{j=1}^{i-1} \frac{1}{z_i^k - z_j^{k+1}} + \sum_{j=i+1}^{j=n} \frac{1}{(z_i^k - z_j^k)})}, i = 1, ..., n \tag{16}$$

Using Eq. 16 to update the vector solution Z, we expect the Gauss-Seidel iteration to converge faster because, just as any Jacobi algorithm (for solving

linear systems of equations), it uses the freshest computed roots z_i^{k+1}. Following the proof presented in 4, we specify that we have implemented the asynchronous version of Ehrlich-Aberth method using Eq. 16.

Algorithm 2 shows the key points of the "Update" iterative function of Algorithm 1 line 4, implemented as a kernel.

Algorithm 2: Update kernel for the iterative function

```
if (‖Z‖ <= R) then
    kernel_Classical_update(Z, P, P');
else
    kernel_update_ExpoLog(Z, P, P');
```

Experimentally speaking, overflows are frequent if we try to solve a high degree polynomial with the classical Ehrlich-Aberth method as we explained before. To avoid this problem, we check if the modulus of the roots is greater than the circle (R), in this case the `kernel_update_ExpoLog()` is called and the EA.EL function Eq. 12 is used (with Eq. 9 and Eq. 10), in order to take into account the limited capacity of floats represented in processors. If the modulus of the root is not greater then the classical form of the EA function (Eq. 5) is executed. We should notice here that we used the cuda `cuDoubleComplex` native type already available in CUDA.

The last kernel $Test\_Converge()$ in Algorithm 1, line 6, checks the convergence of the roots after each update of z^k, according to formula Eq. 6. Here again, we used two functions of the CUBLAS Library (CUDA Basic Linear Algebra Subroutines): `cublasGetVector` to transfer the *Error* vector from host to device and `cublasIdamax()` to compute the maximum of the *Error* vector in Algorithm 1 line 7.

Listing 1 shows a simplified version of the second kernel code (some parameters in the kernels have been simplified in order to increase the readability). As can be seen, this kernel calls other multiple kernels. All the kernels for complex numbers and kernels for the evaluation of a polynomial are not detailed.

Listing 1: Kernels to update the roots

```
//classical version of the Ehrlich-Aberth method
__device__
cuDoubleComplex Classical_update_EA(int i,cuDoubleComplex *Z,polynomial P,
polynomial Pu){

  cuDoubleComplex  result;
  cuDoubleComplex  C,F,Fp;
  int              j;
  cuDoubleComplex  sum;
  cuDoubleComplex  un;

  //evaluate the polynomial
  F = Fonction(Z[i],P);
  //evaluate the derivative of the polynomial.
  Fp=FonctionD(Z[i],Pu);

  sum.x=0;sum.y=0;
  un.x=1;un.y=0;
  C=cuCdiv(F,Fp);        //P(z)/P'(z)

  //for all roots, compute the sum
  //for the Ehrlich-Aberth iteration
  for ( j=0 ; j<P.PolyDegre ; j++ )
  {
    if ( i != j )
    {
      sum=cuCadd(sum,cuCdiv(un,cuCsub(Z[i],Z[j])));
    }
  }
  sum=cuCdiv(C,cuCsub(un,cuCmul(C,sum)));    //C/(1-Csum)
  result=cuCsub(Z[i], sum);
  return (result);
}


//Exp-Log version of the Ehrlich-Aberth method
__device__
```

```
cuDoubleComplex ExpoLog_update_EA(int i,cuDoubleComplex *Z,polynomial P,
polynomial Pu) {

  cuDoubleComplex result;
  cuDoubleComplex F,Fp;
  cuDoubleComplex one,denominator,sum;
  int             j;
  one.x=1;one.y=0;
  sum.x=0;
  sum.y=0;

  //evaluate the polynomial with
  //the Exp-Log version
  Fp = LogFonctionD(Z[i],P);
  //evaluate the derivative of the polynomial
  //with the Exp-Log version
  F = LogFonction(Z[i],Pu);

  cuDoubleComplex FdivFp=cuCsub(F,Fp);

  //for all roots, compute the sum
  //for the Ehrlich-Aberth iteration
  for ( j=0 ; j<P.degrePolynome ; j++ )
  {
    if ( i != j )
    {
      sum=cuCadd(sum,cuCdiv(un,cuCsub(Z[i],Z[j])));
    }
  }

  //then terminate the computation
  //of the Ehrlich-Aberth method
  denominator=cuCln(cuCsub(un,cuCexp(cuCadd(FdivFp,cuCln(sum)))));
  result=cuCsub(FdivFp,denominator);
  result=cuCsub(Z[i],cuCexp(res));

  return   result;
```

```
}

//kernels to update a root i
__global__
void update_EA(int size,cuDoubleComplex *Z,polynomial P,polynomial Pu,
int* finished, double MaxRadius) {
  int i= blockIdx.x*blockDim.x+ threadIdx.x;
  if(i<size) {
    //if the root needs to be updated
    if(!finished[i]) {
      //according to the module of the root
      if (cuCmodule(Z[i])<=maxRadius)
        //selects the classicl version
        Z[i] = Classical_update_EA(i,Z,P,Pu);
      else
        //of the Exp-Log version
        Z[i] = ExpoLog_update_EA(i,Z,P,Pu);
     }
  }
}
/* Function executed en CPU */
void EA_Method(Polynomial P,Polynomial Pu,cuDoubleComplex *Z,cuDoubleComplex *ZPrev,
double * finiched,double MaxRadius)
{int Threads=256;                                          //size of Threads blocks
 int Blocks = (P.degrePolynome + Threads - 1) / Threads; //number of thread blocks
//MaxRadius,Polynomial coefficients of P and Pu, filled in the constant memory
//vector Elements of Z are filled in the global memory
do {
     Save<<<Blocks,Threads>>>(P.degrePolynome, Z, ZPrev);
     Update_EA<<<Blocks,Threads>>>(P.degrePolynome,Z,P,Pu,finished,MaxRadius);
     Test_Convergence_EA<<<Blocks,Threads>>>(P.degrePolynome,Z,ZPrev,d_Error);

     cublasIdamax(handle, P.degrePolynome, d_Error, 1, &index);
     cublasGetVector(1, sizeof(double), &d_Error[index], 1, &ArretMax, 1);

   }  while (ArretMax >= EPSILON );
}
```

Each of the *Save, Update, and Test_Converge* kernels is executed by a large

number of GPU threads such that each thread is in charge of the computation of one component of the iterate vector Z. To maximize the parallel execution between the GPU streaming processors, we set the size of a thread block, *Threads*, to 256 threads and the number of thread blocks, *Blocks*, is computed according to Eq.17 so as to have each GPU thread in charge of one vector element.

$$Blocks = \frac{N + Threads - 1}{Threads}, \tag{17}$$

It should be noticed that, as blocks of threads are scheduled automatically by the GPU, we have absolutely no control on the order of the blocks. Consequently, our algorithm is executed with the asynchronous iteration model, where blocks of roots are updated in a non deterministic way. Another consequence of that, is that several executions of our algorithm with the same polynomial do not necessarily give the same results (but roots have the same accuracy) and the same number of iterations (even if the variation is not very significant). From our point of view, our code is quite optimized, it is normal that some kernels produce branch divergence that cannot be suppressed. For example, to compute a root, all the other roots are used. It is clear that the computation of root j is different from the computation of root $j+1$, because in Eq. 5, there is a sum in which the current root is excluded. This is a cause of branch divergence. The other one is due to the fact that a root can use different routines to be updated: the normal mode or the log-exp mode for the EA method. Finally it should be noticed that this code is quite complex and is written in about 2,000 lines of code.

7. Experimental study

We study two categories of polynomials: sparse polynomials and full ones. *A sparse polynomial* is a polynomial for which only some coefficients are not null. In this paper, we consider sparse polynomials for which the roots are distributed on 2 distinct circles:

$$\forall \alpha_1 \alpha_2 \in \mathbb{C}, \forall n_1, n_2 \in N^*; P(z) = (z^{n_1} - \alpha_1)(z^{n_2} - \alpha_2) \tag{18}$$

A full polynomial is, in contrast, a polynomial for which all the coefficients are not null. A full polynomial is defined by:

$$\forall a_i \in \mathbb{C}, i \in \mathbb{N}; p(x) = \sum_{i=0}^{n} a_i.x^i, a_i \neq 0 \tag{19}$$

For our experiments, a machine, composed of 2 CPU Intel Xeon E5-2660 @ 2.20GHz (with 8 cores each), has been used with OpenMP and a machine composed of one CPU Intel(R) Xeon(R) CPU E5620@2.40GHz, with a NVIDIA GPU K40 (with 6 GB of RAM) has been used for the GPU computation.

We performed a set of experiments on the parallel algorithms with a bi-CPU machine and a single GPU. We measured the execution time and took into account the polynomial size, the number of threads per block and the degree of sparsity of polynomials (sparse and full). Firstly, we discuss the performance behavior of the asynchronous version of Ehrlich-Aberth method implemented on GPU with CUDA vs. on a multi-core CPU using OpenMP. Then, we study the influence of the number of threads per block on the execution time of the EA method to solve (sparse and full) polynomials. Later, we show the contribution of the exp-log solution to compute a high degree polynomial with the EA method. Finally, we compare the performance behaviors of EA method with the Durand-Kerner (DK) method.

All experimental results obtained from the simulations are done in double precision data, the convergence threshold of the methods is set to 10^{-7}. The initialization values of the vector solution of the EA method are given in Eq 7.

7.1. Execution time of the EA method on a 8-cores dual CPU with OpenMP vs. on a single Tesla GPU with CUDA

In the Ehrlich-Aberth method implemented with OpenMP, all the data are shared with the OpenMP threads. The shared data are the solution vector Z, the polynomial to solve P, its derivative P', and the error vector $Error$. The number of cores is fixed or defined by an environment variable: `OMP_NUM_THREADS`. We could use until 16 real cores per node. The gcc 4.4.7 compiler has been used

with the `-O3` option to optimize the code. For the GPU implementation, we fixed the number of threads per blocks to 256, the number of blocks is computed with Eq 17.

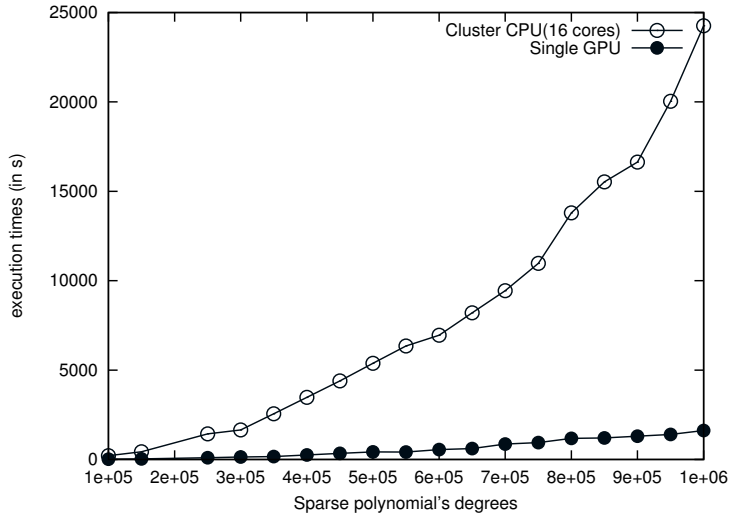

Figure 1: Execution time of the EA method on a 16 cores-node with OpenMP vs. on a single GPU K40.

In Figure 1, we report the execution time of the EA method implemented with OpenMP (with 16 cores) and a GPU K40. We chose different sparse polynomials with high degrees ranging from 100,000 to 1,000,000. Firstly, we can notice that both implementations manage to solve a polynomial of degree 1,000,000. However, it takes about 2,200s for the GPU to solve a one million degree polynomial whereas the CPU implementation only solves a polynomial of degree 300,000 during the same period. It should be noticed that both implementations have approximately the same number of iterations and the same accuracy.

7.2. Influence of the number of threads on the execution time of different polynomials (sparse and full)

In order to maximize the use of GPU cores (maximize the number of threads executed in parallel) according to the execution time consuming, it is interesting

to see the influence of the number of threads per block on the execution time of the Ehrlich-Aberth algorithm. For that, we noticed that the maximum number of threads per block for the Nvidia Tesla K40 GPU is 1,024, so we varied the number of threads per block from 8 to 1,024. We measured the execution time for 10 different sparse and full polynomials of degree 50,000 and of degree 500,000 and the results are presented in Figure 2.

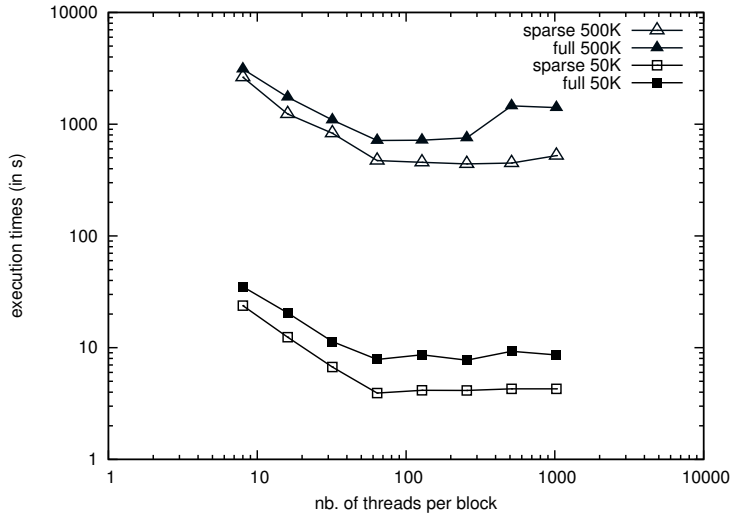

Figure 2: Influence of the number of threads per block on the execution time on sparse and full polynomials.

Figure 2 shows that the best execution time for both sparse and full polynomials is obtained when the number of threads per block is between 64 and 256. We also notice that, with small polynomials, the best number of threads per block is 64, whereas, for large polynomials, the best number of threads per block is 256. For this reason, in the following experiments, we set the number of threads per block to 256.

7.3. Influence of exp-log solution to compute high degree polynomials

In this experiment we report the performance of the exp-log solution described in Section 3 to compute high degree polynomials.

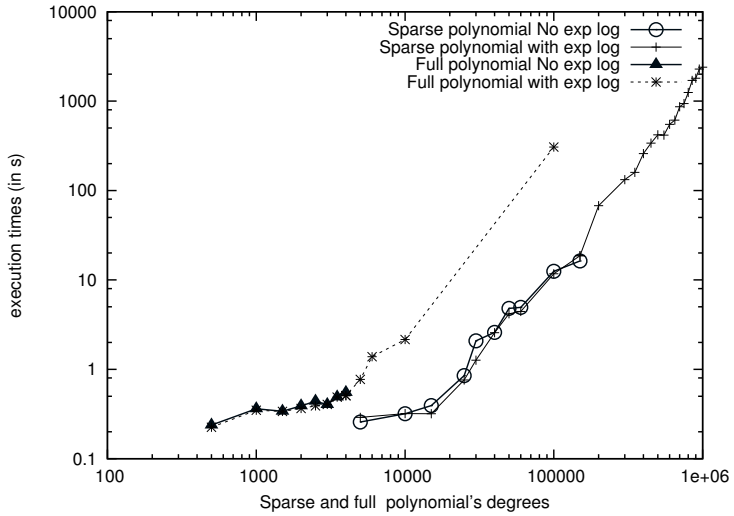

Figure 3: The impact of exp-log solution to compute high degree polynomials

Figure 3 shows the contribution of the *exp-log* solution to solve high degree polynomials. We report the execution time of the Ehrlich-Aberth method using the *exp-log* solution and the execution time of the classical version of the Ehrlich-Aberth method, with full and sparse polynomials. We can first see that the execution times for both (classical, exp-log) versions of the EA algorithm are the same with full polynomials of degree less than 4,000 and with sparse polynomials less than 150,000. We also clearly show that the classical version (without exp-log) of Ehrlich-Aberth algorithm does not converge after these degrees either with sparse or full polynomials. This is due to the limited capacity of double numbers manipulated by processors. However, the new version of the Ehrlich-Aberth algorithm with the *exp-log* solution converges and can accurately solves high degree polynomials.

7.4. Comparison of the Durand-Kerner and the Ehrlich-Aberth methods

In this part, we compare the Durand-Kerner and the Ehrlich-Aberth methods on a GPU. We measure the execution times, the number of iterations, and take into account the polynomials size for both sparse and full polynomials.

7.4.1. The execution time of the DK and EA methods on a GPU

In this experiment, we report the execution times of the EA method and the DK method on GPU, for both (sparse and full) polynomials root of degrees ranging from 100,000 to 1,000,000.

We recall that DK and EA methods have, in theory, respectively quadratic and cubic convergence orders in case of simple zeros. The DK method follows the same steps as the EA method, except for the iterative function of DK which is given in Eq. 3.

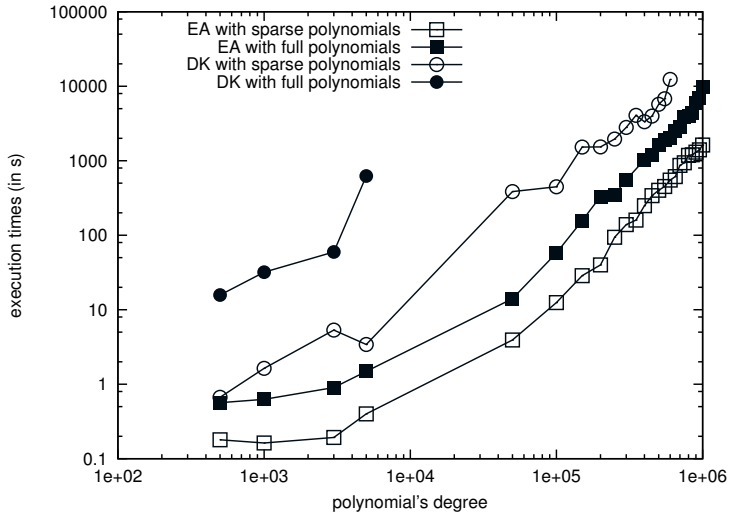

Figure 4: Execution times of the Durand-Kerner and the Ehrlich-Aberth methods on GPU

In Figure 4, we can notice that EA converges indeed more rapidly than DK for both sparse and full polynomials. In addition to its cubic convergence order, the derivative of the polynomial to solve P called in the iterative function of EA, makes it faster to converge to the roots solution even if the computation P' takes more time. Specifically with full polynomials, DK cannot compute polynomials upper than a degree of 5000. This reduces the computational capabilities of the DK method, unlike the method of EA, which has proven its efficiency to compute polynomials of $1,000,000$ degree.

7.4.2. On the number of iterations needed for the EA and the DK methods to converge

In this experiment, we report the number of iterations needed to converge for both the EA and DK methods with sparse polynomial degrees ranging from $1,000$ to $1,000,000$.

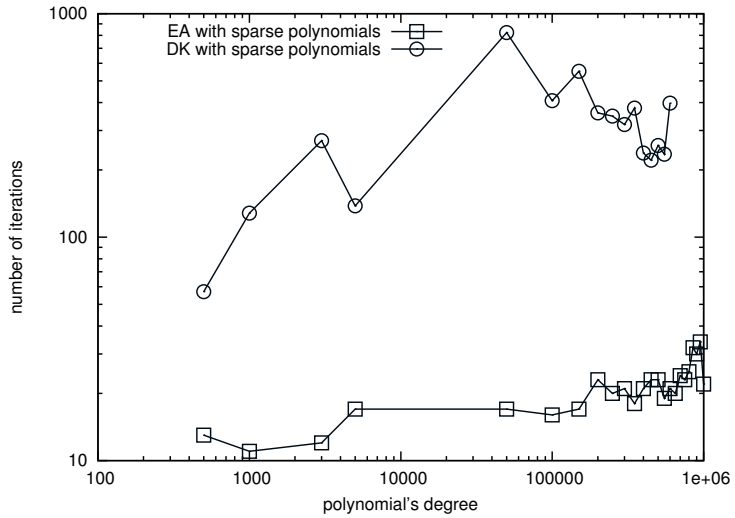

Figure 5: The number of iterations needed to converge for the EA and the DK methods

Figure 5 shows that the number of iterations of DK is of order 100 while EA is of order 10. Indeed the computation of the derivative of P in the iterative function (Eq. 5) executed by EA allows the algorithm to converge faster. On the contrary, the DK operator (Eq. 3) needs low operations and consequently low execution times per iteration, but it needs more iterations to converge.

8. Further Data Analysis

Expressing the evolution of execution time as a mathematical function is a very appealing method in order both to analyze and understand any implementation and to compare it to other implementations or other methods. In our case, however, finding an analytic formula proved to be neither trivial nor straightforward and still constitutes an open problem. That is why we adopted

a curve fitting approach in order to have a better insight on the performance of our implementation, especially its speed-up and scalability with respect to execution times. Furthermore, the advantage of such a mathematical formulation of execution times is that it can be used to predict the execution times of very large polynomials degrees still not subject to experiments. In this section, we compute and discuss the curve fitting of EA execution time on a 16-core CPU and on the K40 GPU for sparse polynomials. But the approach can be easily applied to full polynomials as well. We first explain our methodology. The fitting has been carried out using Gnu Octave Mathematical Software and consisted in finding the coefficients a, b, c of a quadratic polynomial $ax^2 + bx + c$ that best fits on a set of what we call fitting data. For both CPU and GPU execution times, we divided the set of experimental data into two distinct subsets: the fitting data and the validation data. We then looked for the the best fit (actually the coefficients) using the fitting data and we assessed the quality of the fitting function both internally to the fitting data and externally with respect to the validation data by producing the relative error between the actual experimental result and the output of the fitting function for each validation point.

8.1. Execution times of the EA method on the CPU

Figure 6 shows the EA OpenMP execution times on the 16-core CPU for sparse polynomial's degrees ranging from 100K to 1M. Fitting data is represented by red points in the figure, validation data in blue and the computed fitting function is sketched in black. Table 3 shows the actual values of a, b and c. The fitting function has a goodness measure SSE[1] value of 1.6×10^{-5} and R_Square[2] value of $0,99$, meaning that the fit explains 99% of the total variation in the fitting data on average. Additionally, we show in Table 1 the actual execution times for validation data and the predicted results of the fitting func-

[1] Sum of Squares Due to Error.

[2] R-square is the square of the correlation between the response values and the predicted response values for the fitting data.

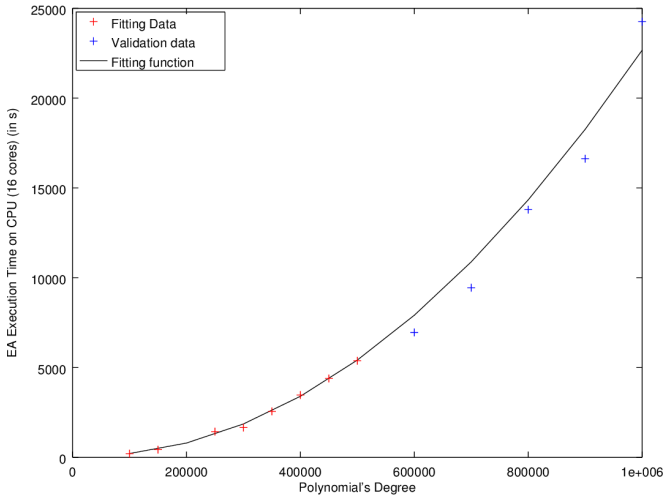

Figure 6: Fitting data, validation data and fitting function in seconds of the OpenMP execution time of EA on the 16-core CPU

tion for each validation point and associated relative error. We can see that the error is generally around 10%, larger errors stem from bias in the measurement of experimental times because, from one experiment to another and depending on various parameters on the machine, the execution times can slightly vary. For larger degrees, the prediction seems even better which can be interpreted as the higher the degree is, the better the function can predict the execution times which are dominated by computations rather than communications (memory access).

8.2. Execution times of the EA method on GPU

Similarly to Figure 6, Figure 7 shows the EA execution times on the K40 GPU for fitting data, validation data and the fitting function whose coefficients are shown in the second raw of Table 3. With respect to fitting data, we have (R_Square=0.9538) SSE value of 1.38×10^{-8} and the fitting method can accurately predict the execution times for large polynomial's degrees with an average relative error of 4.32×10^{-2}. With respect to validation data (cf Table 2), we

Degree /Time (s)	Measured	Predicted	Relative Error (%)
600,000	7,914.1	7,999.40	13.81
700,000	10,442.97	10,887	15.29
800,000	13,794.50	14,337	3.93
900,000	18,630.70	18,265	9.82
1,000,000	24,255.70	22,671	6.53

Table 1: EA 16-core CPU measured execution time, predicted execution times according to the fitting function and actual relative error for each validation point (the lesser is better)

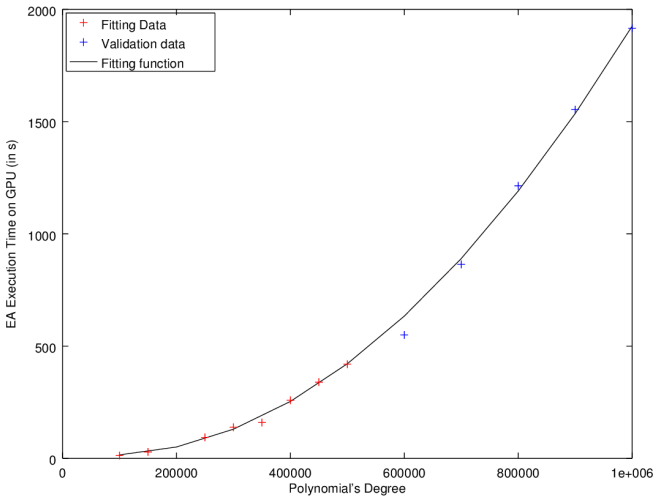

Figure 7: Fitting data, validation data and fitting function in seconds of the CUDA execution time of EA on the K-40 GPU

Degree/Execution time (s)	Measured	Fitted	Relative Error (%)
600,000	549.70	633.70	15.28
700,000	864.21	890.15	3.00
800,000	1,214.82	1,190.87	1.90
900,000	1,553.08	1,535.87	1.10
1,000,000	1,915.6	1,925.15	0.498

Table 2: EA K40 GPU measured execution times, predicted execution times according to the fitting function and actual relative errors for each validation point (the lesser is better).

Time on / Coefficients	a	b	c
16-cores CPU	$2.40\,10^{-8}$	$-13.15\,10^{-4}$	106.54
K40 GPU	$2.20\,10^{-9}$	$-3.135\,10^{-4}$	24.83

Table 3: The coefficients of EA-CPU and EA-GPU fitted quadratic execution time

can see that the fit is around 5% at most, far from the actual experimental results. For a large degree of $1.e6$, the prediction is almost as precise as the actual experimental results. In our opinion, the higher the degree is, the higher the computation times, compared to communication/synchronization times, are.

Finally, based on the actual coefficients in Table 3 and limited by the memory size of the GPU device, one could envision that the speed-up of our GPU implementation is upper bounded by $a_{cpu}/a_{gpu} \approx 11$ for higher polynomial degrees.

Nota bene : we conducted a similar study for the full polynomial setting, the execution times are also quadratic in the size of the problem. We determined the following coefficients for the K40 GPU: $a = 2.8105\,10^{-9}$, $b = 7.4163\,10^{-4}$ and $c = -36.7$. Comparing the sparse to the full setting execution times on the G40 GPU, the evolution of the two curves indicates that, subject to the memory limit of the device, on the long run (for larger degrees), the full polynomials need about 27.75% more time than the sparse polynomial which is the limit when

$n \to \infty$ of the division of the two respective quadratic equations.

9. Conclusion and perspectives

In this paper we have presented the parallel implementation of the Ehrlich-Aberth method on GPU for the polynomial root finding problem. Moreover, we have improved the classical Ehrlich-Aberth method which suffers from overflow problems, the exp-log solution applied to the iterative function allowed us to successfully solve high degree polynomials. We also have proved the convergence of the parallel Ehrlich-method with asynchronous iterations. the results show

Then, we have described the parallel implementation of our modified EA on GPU. We have performed many experiments with the Ehrlich-Aberth method in a 16 cores node and a single GPU. These experiments highlight that this method is more efficient in GPU than all the other implementations. The improvement with the exponential logarithm solution allowed us to solve sparse and full high degree polynomials up to 1,000,000 degree. Hence, it may be possible to consider using polynomial root finding methods in other numerical applications on GPU.

In future works, we plan to investigate the possibility of using several multiple GPUs simultaneously, either with a multi-GPU machine or with a cluster of GPUs. It may also be interesting to study the implementation of other root finding polynomial methods on GPU.

Acknowledgment

This article is partially funded by the Labex ACTION program (ANR-11-LABX-01-01 contract) and the Franche-Comté regional council. We would like to thank NVIDIA for hardware donation under CUDA Research Center 2014 and the Mésocentre de calcul de Franche-Comté for the use of the GPUs.

[1] G. Cardano, Ars Magna or The Rules of Algebra, 1545, MIT, 1968.

[2] N. H. Abel, Beweis der unmöglichkeit, algebraische gleichungen von höheren graden als dem vierten allgemein aufzulösen, J. reine angew, Math 1 (1) (1826) 65–84.

[3] I. Newton, Tractatus de methodis serierum et fluxionum, in: D. T. Whiteside (Ed.), The Mathematical Papers of Isaac Newton, III, Cambridge University Press, Cambridge, 1670–71?, pp. 32–353.

[4] A. A. Grau, Modified Graeffe method, Commun. ACM 8 (6) (1965) 379–380.

[5] J. A. Ford, A Generalization of the Jenkins–Traub method, Mathematics of Computation 31 (137) (1977) 193–203.

[6] F. M. Larkin, Root-Finding by Fitting Rational Functions, Mathematics of Computation 35 (151) (1980) 803–816.

[7] D. E. Muller, A Method for Solving Algebraic Equations using an Automatic Computer, Mathematical Tables and Other Aids to Computation 10 (1956) 208–215.

[8] E. Durand, Solutions numériques des équations algébriques. Tome I: Équations du type $F(x) = 0$; racines d'un polynôme, Masson, Paris, 1960.

[9] I. O. Kerner, Ein Gesamtschrittverfahren zur Berechnung der Nullstellen von Polynomen. (German) [A complete step method for the computation of zeros of polynomials], Numerische Mathematik 8 (3) (1966) 290–294.

[10] K. Weierstrass, Neuer Beweis des Satzes, dass jede ganze rationale function einer veranderlichen dagestellt werden kann als ein product aus linearen functionen derselben veranderlichen, Ges. Werke 3 (1903) 251–269.

[11] L. Ilieff, On the approximations of Newton, Annual Sofia Univ 46 (1950) 167–171.

[12] K. Docev, An alternative method of Newton for simultaneous calculation of all the roots of a given algebraic equation, Phys. Math. J 5 (1962) 136–139.

[13] W. Boersch-Supan, A Posteriori Error Bounds for the Zeros of Polynomials, Numerische Mathematik 5 (1963) 380–398.

[14] L. W. Ehrlich, A modified Newton method for polynomials, Commun. ACM 10 (2) (1967) 107–108.
URL http://doi.acm.org/10.1145/363067.363115

[15] O. Aberth, Iteration Methods for Finding all Zeros of a Polynomial Simultaneously, Mathematics of Computation 27 (122) (1973) 339–344.

[16] G. Loizou, Higher-order iteration functions for simultaneously approximating polynomial zeros, Intern. J. Computer Math 14 (1) (1983) 45–58.

[17] T. L. Freeman, Calculating polynomial zeros on a local memory parallel computer, Parallel Computing 12 (3) (1989) 351–358.
URL http://dx.doi.org/10.1016/0167-8191(89)90093-8

[18] T. Freeman, R. Brankin, Asynchronous polynomial zero-finding algorithms, Parallel Computing 17 (1990) 673–681.

[19] R. Couturier, F. Spies, Extraction de racines dans des polynômes creux de degrées élevés. RSRCP (réseaux et systèmes répartis, calculateurs parallèles), Algorithmes itératifs paralléles et distribués 1 (13) (1990) 67–81.

[20] CUDA C programming guide.
URL http://docs.nvidia.com/cuda/pdf/CUDA_C_Programming_Guide.pdf

[21] K. Ghidouche, R. Couturier, A. Sider, Parallel implementation of the Durand-Kerner algorithm for polynomial root-finding on GPU, IEEE. Conf. on advanced Networking, Distributed Systems and Applications (2014) 53–57.

[22] D. Bini, Numerical computation of polynomial zeros by means of Aberth's method, Numerical Algorithms 13 (2) (1996) 179–200.
URL http://dx.doi.org/10.1007/BF02207694

[23] A. Ostrowski, On a Theorem by J. L. Walsh Concerning the Moduli of Roots of Algebraic Equations.A.M.S., Algorithmes itératifs paralléles et distribués 1 (47) (1941) 742–746.

[24] K. Rhofir, F. Spies, J.-C. Miellou, Perfectionnements de la méthode asynchrone de Durand-Kerner pour les polynômes complexes, Calculateurs Parallèles 10 (4) (1998) 449–458.

[25] M. N. El Tarazi, Some convergence results for asynchronous algorithms, Numerische Mathematik 39 (3) (1982) 325–340. doi:10.1007/BF01407866. URL http://dx.doi.org/10.1007/BF01407866

[26] W. Mirankar, Parallel methods for approximating the roots of a function, IBM Res Dev 13 (1968) 297–301.

[27] W. Mirankar, A survey of parallelism in numerical analysis, SIAM Rev 13 (1971) 524–547.

[28] G. Schedler, Parallel Numerical Methods for Solution of Equations, Commun ACM 10 (1967) 286–290.

[29] S. Winograd, Parallel Iteration Methods, in: R. E. Miller, J. W. Thatcher (Eds.), Complexity of Computer Computations, The IBM Research Symposia Series, Plenum Press, New York, 1972, pp. 53–60.

[30] Ben-Or, Feig, Kozen, Tiwari, A Fast Parallel Algorithm for Determining All Roots of a Polynomial with Real Roots, SICOMP: SIAM Journal on Computing 17.

[31] P. Jana, Polynomial interpolation and polynomial root finding on OTIS-Mesh, Parallel Comput 32 (3) (2006) 301–312.

[32] P. Jana, B. Sinha, R. D. Gupta, Efficient parallel algorithms for finding polynomial zeroes, Proc of the 6th int conference on advance computing, CDAC, Pune University Campus,India 15 (3) (1999) 189–196.

[33] T. Rice, L. Jamieson, A highly parallel algorithm for root extraction, IEEE Trans Comp 38 (3) (2006) 443–449.

[34] H. Azad, The performance of synchronous parallel polynomial root extraction on a ring multicomputer, Clust Comput 2 (10) (2007) 167–174.

[35] L. Gemignani, Structured matrix methods for polynomial root-finding, in: C. W. Brown (Ed.), Proceedings of the 2007 International Symposium on Symbolic and Algebraic Computation, July 29–August 1, 2007, University of Waterloo, Waterloo, Ontario, Canada, ACM Press, pub-ACM:adr, 2007, pp. 175–180. doi:http://doi.acm.org/10.1145/1277548.1277573.

[36] B. Kalantari, Polynomial root finding and polynomiography, World Scientifict, 2008.

[37] X. Zhanc, Z. M. Wan, A constrained learning algorithm for finding multiple real roots of polynomial, In: Proc of the 2008 intl symposium on computational intelligence and design (2008) 38–41.

[38] W. Zhu, Z. Zeng, D. Lin, An Adaptive Algorithm Finding Multiple Roots of Polynomials, in: F. Sun, J. Z. 0001, Y. Tan, J. Cao, W. Y. 0001 (Eds.), ISNN (2), Vol. 5264 of Lecture Notes in Computer Science, Springer, 2008, pp. 674–681.
URL http://dx.doi.org/10.1007/978-3-540-87734-9_77

[39] D. Bini, L. Gemignani, Inverse power and Durand Kerner iterations for univariate polynomial root finding, Comput Math Appl 47 (2004) 447–459.

[40] M. Cosnard, P. Fraigniaud, Finding the roots of a polynomial on an MIMD multicomputer, Parallel Comput 15 (3) (1990) 75–85.

[41] P. Jana, Finding polynomial zeroes on a Multi-mesh of trees (MMT), In: Proc of the 2nd int conference on information technology (1999) 202–206.