1 Root finding problem

We consider a polynomial of degree n having coefficients in the complex C and zeros α_i, *i=1,...,n*.

$$p(x) = \sum a_i x^i = a_n \prod (x - \alpha_i), a_0 a_n \neq 0 \tag{1}$$

the root finding problem consist to find all n root of *p(x)*. the problem of finding a root is equivalent to the problem of finding a fixed-point. To see this consider the fixed-point problem of finding the n-dimensional vector x such that

$$x = g(x).$$

where $g : C^n \longrightarrow C^n$. Note that 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

$$g(x) = f(x) - x$$

Often it will not be possible 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 numerically can be divided into two main groups: direct methods and iterative methods.
Direct methods exist only for $n \leqslant 4$,solved in closed form by G. Cardano in the mid-16th century. However, N.H. Abel in the early 19th century showed that polynomials of degree five or more could not be solved by directs methods. Since then researchers have concentrated on numerical (iterative) methods such as the famous Newton s method, Bernoulli s method of the 18th, and Graeffe s. With the advent of electronic computers, different methods has been developed such as the Jenkins-Traub method, Larkin s method, Muller s method, and several methods for simultaneous approximation of all the roots, starting with the Durand-Kerner method:

$$Z_i = Z_i - \frac{P(Z_i)}{\prod_{i \neq j}(z_i - z_j)} \tag{2}$$

This formula is mentioned for the first time from Weiestrass [15] as part of the fundamental theorem of Algebra and is rediscovered from Ilieff [17], Docev [13], Durand [7], Kerner [12]. Another method discovered from Borsch-Supan [29] and also described and brought in the following form from Ehrlich [30] and Aberth [2].

$$Z_i = Z_i - \frac{1}{\frac{P'(Z_i)}{P(Z_i)} - \sum_{i \neq j}(z_i - z_j)} \tag{3}$$

Aberth, Ehrlich and Farmer-Loizou [9] have proved that the above method has cubic order of convergence for simple roots.

Iterative methods raise several problem when implemented e.g. specific sizes of numbers must be used to deal with this difficulty.Moreover,the convergence time of iterative methods drastically increase like the degrees of high polynomials. The parallelization of these algorithms will improve the convergence time.

Many authors have treated the problem of parallelization of simultaneous methods. Freeman [26] has tested the DK method, EA method and another method of the fourth order proposed from Farmer and Loizou [9],on a 8- processor linear chain, for polynomial of degree up to 8. The third method often diverges, but the first two methods have speed-up 5.5 (speed-up=(Time on one processor)/(Time on p processors)). Later Freeman and Bane [27] consider asynchronous algorithms, in which each processor continues to update its approximations even although the latest values of other $z_i((k))$ have not received from the other processors, in difference with the synchronous version where it would wait. in [4]proposed two methods of parallelization for architecture with shared memory and distributed memory,it able to compute the root of polynomial degree 10000 on 430 s with only 8 pc and 2 communications per iteration. Compare to the sequential it take 3300 s to obtain the same results.

After this few works discuses this problem until the apparition of the Compute Unified Device Architecture (CUDA) [1],a parallel computing platform and a programming model invented by NVIDIA. the computing ability of GPU has exceeded the counterpart of CPU. It is a waste of resource to be just a graphics card for GPU. CUDA adopts a totally new computing architecture to use the hardware resources provided by GPU in order to offer a stronger computing ability to the massive data computing.

Indeed, [8]proposed the implementation of the Durand-Kerner method on GPU (Graphics Processing Unit). The main result prove that a parallel implementation is 10 times as fast as the sequential implementation on a single CPU for high degree polynomials that is greater than about 48000.

The mean part of our work is to implement the Aberth method on GPU and compare it with the Durand Kerner implementation.................To be continued..................

2 Aberth method and difficulties

A cubically convergent iteration method for finding zeros of polynomials was proposed by O.Aberth [2].The Aberth method is a purely algebraic derivation.To illustrate the derivation, we let $w_i(z)$ be the product of linear factor $w_i(z) = \prod_{j=1,j\neq i}^{n}(z - x_j)$

and rational function $R_i(z)$ be the correction term of Weistrass method [15]:

$$R_i(z) = \frac{p(z)}{w_i(Z)}, i = 1, 2, ..., n. \tag{4}$$

Differentiating the rational function $R_i(z)$ and applying the Newton method, we have

$$\frac{R_i(z)}{R_i'(z)} = \frac{p(z)}{p'(z) - p(z)\dfrac{w_i(z)}{w_i'(z)}} = \frac{p(z)}{p'(z) - p(z)\sum_{j=1,j\neq i}^{n}\dfrac{1}{z - x_i}}, i = 1, 2, ..., n \tag{5}$$

Substituting x_j for z we obtain the Aberth iteration method

Let present the means stages of Aberth's method.

2.1 Polynomials Initialization

The initialization of polynomial P(z) with complex coefficients are given by:

$$p(z) = \sum a_i z^{n-i}.where a_n \neq 0, a_0 = 1, a_i \subset C \tag{6}$$

2.2 Vector $Z^{0)}$ Initialization

The choice of the initial points $z_i^{(0)}, i = 1, ..., n$, from which starting the iteration (2) or (3), is rather delicate since the number of steps needed by the iterative method to reach a given approximation strongly depends on it. In [2]the Aberth iteration is started by selecting n equispaced points on a circle of center 0 and radius r, where r is an upper bound to the moduli of the zeros. After, [5] performs this choice by selecting complex numbers along different circles and relies on the result of [3].

$$\sigma_0 = \frac{u + v}{2}; u = \frac{\sum_{i=1}^{n} u_i}{n.max_{i=1}^{n} u_i}; v = \frac{\sum_{i=0}^{n-1} v_i}{n.min_{i=0}^{n-1} v_i}; u_i = 2.|a_i|^{\frac{1}{i}}; v_i = \frac{|\frac{a_n}{a_i}|^{\frac{1}{n-i}}}{2} \tag{7}$$

2.3 Iterative Function Hi

The operator used with Aberth's method is corresponding to the following equation which will enable the convergence towards polynomial solutions, provided all the roots are distinct.

$$H_i(z) = z_i - \frac{1}{\frac{P'(z_i)}{P(z_i)} - \sum_{j \neq i} \frac{1}{z_i - z_j}} \tag{8}$$

2.4 Convergence condition

determines the success of the termination. It consists in stopping the iterative function $H_i(z)$ when the are stable,the method converge sufficiently:

$$\forall i \in [1, n]; \frac{z_i^{(k)} - z_i^{(k-1)}}{z_i^{(k)}} < \xi \tag{9}$$

3 Difficulties and amelioration

the Aberth method implementation suffer of overflow problems. This situation occurs, for instance, in the case where a polynomial having positive coefficients and large degree is computed at a point ξ where $|\xi| > 1$.Indeed the limited number in the mantissa of floating takings the computation of P(z) wrong when z is large. for example $(10^{50})+1+(-10_{50})$ will give result 0 instead of 1 in reality.consequently we can't compute the roots for large polynomial's degree. This problem was discuss in [14] for the Durand-Kerner method, the authors propose to use the logratihm and the exponential of a complex:

$$\forall (x, y) \in R^{*2}; \ln(x + i.y) = \ln(x^2 + y^2)2 + i.\arcsin(y\sqrt{x^2 + y^2})_{]-\pi,\pi[} \tag{10}$$

$$\begin{aligned} \forall (x, y) \in R^{*2}; \exp(x + i.y) &= \exp(x).\exp(i.y) && (11) \\ &= \exp(x).\cos(y) + i.\exp(x).\sin(y) && (12) \end{aligned}$$

The application of logarithm can replace any multiplications and divisions with additions and subtractions; consequently it manipulates lower absolute values and can be compute the roots for large polynomial's degree exceed [14].

Applying this solution for the Aberth method we obtain the iteration function with logarithm:

$$H_i(z) = z_i^k - \exp\left(\ln\left(p(z_k)\right) - \ln\left(p(z_k^{'})\right) - \ln\left(1 - Q(z_k)\right)\right) \tag{13}$$

where:

$$Q(z_k) = \exp\left(\ln(p(z_k)) - \ln(p(z_k^{'})) + \ln\left(\sum_{k\neq j}^{n} \frac{1}{z_k - z_j}\right)\right) \tag{14}$$

this solution is applying when it is necessary

4 The implementation of simultaneous methods in a parallel computer

The main problem of the simultaneous methods is that the necessary time needed for the convergence is increased with the increasing of the degree of the polynomial. The parallelization of these algorithms will improve the convergence time. Researchers usually adopt one of the two following approaches to parallelize root finding algorithms. One approach is to reduce the total number of iterations as implemented by Miranker [31, 32], Schedler [10] and Winogard [24]. Another approach is to reduce the computation time per iteration, as reported in [18, 22, 23, 25]. There are many schemes for simultaneous approximations of all roots of a given polynomial. Several works on different methods and issues of root finding have been reported in [11, 16, **?**, 28, 34, 33]. However, Durand-Kerner and Ehrlich methods are the most practical choices among them [6]. These two methods have been extensively studied for parallelization due to their following advantages. The computation involved in these methods has some inherent parallelism that can be suitably exploited by SIMD machines. Moreover, they have fast rate of convergence (quadratic for the Durand-Kerner method and cubic for the Ehrlich). Various parallel algorithms reported for these methods can be found in [19, 26, 27, **?**, 21, 23]. Freeman and Bane [27] 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. Therefore they cause a high degree of memory conflict. Recently the author in [32] proposed two versions of parallel algorithm for the Durand-Kerner method, and Aberth method on an on model of Optoelectronic Transpose Interconnection System (OTIS).The algorithms are mapped on an OTIS-2D torus using N processors. This solution need N processors to compute N roots, that it is not practical (is not suitable to compute large polynomial's degrees). Until then, the related works are not able to compute the root of the large polynomial's degrees (higher then 1000) and with small time.

Finding polynomial roots rapidly and accurately it is our objective, with the apparition of the CUDA(Compute Unified Device Architecture), finding the roots of polynomials becomes rewarding and very interesting, CUDA adopts a totally new computing architecture to use the hardware resources provided by GPU in order to offer a stronger computing ability to the massive data computing.in [8] we proposed the first implantation of the root finding polynomials method on GPU (Graphics Processing Unit),which is the Durand-Kerner method. The main result prove that a parallel implementation is 10 times as fast as the sequential implementation on a single CPU for high degree polynomials that is greater than about 48000. Indeed, in this paper we present a parallel implementation of Aberth's method on GPU, more details are discussed in the following of this paper.

5 A parallel implementation of Aberth's method

5.1 Background on the GPU architecture

A GPU is viewed as an accelerator for the data-parallel and intensive arithmetic computations. It draws its computing power from the parallel nature of its hardware and software architectures. A GPU is composed of hundreds of Streaming Processors (SPs) organized in several blocks called Streaming Multiprocessors (SMs). It also has a memory hierarchy. It has 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 as a grid of thread blocks. Therefore, each SM of the GPU executes one or more thread blocks in 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). Indeed at any given clock cycle, the 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. Hence, the data must be transferred in and out of the GPU. However, the speed of memory copy between the GPU and the CPU is slower than the memory bandwidths of the GPU memories and, thus, it dramatically affects the performances of GPU computations. Accordingly, it is necessary to limit data transfers between the GPU and its CPU during the computations.

5.2 Background on the CUDA Programming Model

The CUDA programming model is similar in style to a single program multiple-data (SPMD) softwaremodel. The GPU is treated as a coprocessor that executes data-parallel kernel functions. CUDA provides three key abstractions, a hierarchy of thread groups, shared memories, and barrier synchronization. Threads have a three level hierarchy. A grid is a set of thread blocks that execute a kernel function. Each grid consists of blocks of threads. Each block is composed of hundreds of threads. Threads within one block can share data using shared memory and can be synchronized at a barrier. All threads within a block are executed concurrently on a multi-threaded architecture.The programmer specifies the number of threads per block, and the number of blocks per grid. A thread in the CUDA programming language is much lighter weight than a thread in traditional operating systems. A thread in CUDA typically processes one data element at a time. The CUDA programming model has two shared read-write memory spaces, the shared memory space and the global memory space. The shared memory is local to a block and the global memory space is accessible by all blocks. CUDA also provides two read-only memory spaces, the constant space and the texture space, which reside in external DRAM, and are accessed via read-only caches

5.3 A parallel implementation of the Aberth's method

5.3.1 A sequential Aberth algorithm

The means steps of Aberth's method can expressed as an algorithm like:

Algorithm 1: Algorithm to find root polynomial with Aberth method

```
Input: $Z^0$(Initial root's vector),$\varepsilon$ (error tolerance
       threshold),P(Polynomial to solve)
Output: Z(The solution root's vector)

1  Initialization of the parameter of the polynomial to solve;
2  Initialization of the solution vector $Z^0$;
3  while $\Delta z_{max} \succ \epsilon$ do
4      Let $\Delta z_{max} = 0$;
5      for $j \leftarrow 0$ to $n$ do
6          $ZPrec[j] = Z[j]$;
7          $Z[j] = H(j, Z)$;
8      for $i \leftarrow 0$ to $n-1$ do
9          $c = \frac{|Z[i]-ZPrec[i]|}{Z[i]}$;
10         if $c \succ \Delta z_{max}$ then
11             $\Delta z_{max}$=c;
```

In this sequential algorithm one thread CPU execute all steps, let see the step 3 the execution of the iterative function , 2 instructions are needed, the first instruction *save* the solution vector for the previous iteration, the second instruction *update* or compute a new values of the roots. We have two manner to execute the iterative function, taking a Jacobi iteration who need all the previous value $z_i^{(k)}$ to compute the new value $z_i^{(k+1)}$we have:

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

Or with the Gauss-seidel iteration, we have:

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

In formula(16) the iteration function use the z_i^{k+1} computed in the current iteration to compute the rest of the roots, which take him to converge more quickly compare to the jacobi iteration (it's well now that the Gauss-seidel iteration converge more quickly because they used the most fresh computed root, so we used Gauss-seidel iteration.)

The steps 4 of the Aberth's method compute the convergence of the roots, using(9) formula. Both steps 3 and 4 use 1 thread to compute N roots on CPU, which is faster and hard, it make the algorithm faster and hard for the large polynomial's roots finding.

The execution time Let $T_i(N)$: the time to compute one new root's value of the step 3,T_i depend on the polynomial's degrees N, when N increase T_i increase to.We need $N.T_i(N)$ to compute all the new root's value in one iteration on the step 3.

Let T_j: the time to compute one root's convergence value of the step 4, we need $N.T_j$ to compute all the root's convergence value in one iteration on the step 4.

The execution time for both steps 3 and 4 can see like:

$$T_{exe} = N(T_i(N) + T_j) + O(n). \tag{17}$$

Let Nbr_iter the number of iteration necessary to compute all the roots,so the total execution time $Total\_time_{exe}$ can give like:

$$Total\_time_{exe} = \left[N\left(T_i(N) + T_j\right) + O(n)\right].Nbr\_iter. \tag{18}$$

The execution time increase with the increasing of the polynomial's root, which take necessary to parallelize this step to reduce the execution time. In the following paper you explain how we parrallelize this step using GPU architecture with CUDA platform.

5.3.2 Parallelize the steps on GPU

On the CPU Aberth algorithm both steps 3 and 4 contain the loop `for` , it use one thread to execute all the instruction in the loop N times.Here we explain how the GPU architecture can compute this loop and reduce the execution time. The GPU architecture affect the execution of this loop to a groups of parallel threads organized as a grid of blocks each block contain a number of threads. All threads within a block are executed concurrently in parallel. the instruction are executed as a kernel.

Let nbr_thread be the number of threads executed in parallel, so you can easily transform the (18)formula like this:

$$Total\_time_{exe} = \left[\frac{N}{nbr\_thread}\left(T_i(N) + T_j\right) + O(n)\right].Nbr\_iter. \tag{19}$$

In theory, the $Total\_time_{exe}$ on GPU is speed up nbr_thread times as a $Total\_time_{exe}$ on CPU. We show more details in the experiment part.

In CUDA platform, All the instruction of the loop `for` are executed by the GPU as a kernel form. A kernel is a procedure written in CUDA and defined by a heading `__global__`, which means that it is to be executed by the GPU.the following algorithm see the Aberth algorithm on GPU:

Algorithm 2: Algorithm to find root polynomial with Aberth method

```
Input: $Z^0$(Initial root's vector),$\varepsilon$ (error tolerance
       threshold),P(Polynomial to solve)
Output: Z(The solution root's vector)
1 Initialization of the parameter of the polynomial to solve;
2 Initialization of the solution vector $Z^0$;
3 Allocate and fill the data in the global memory GPU;
4 while $\Delta z_{max} \succ \epsilon$ do
5     Let $\Delta z_{max} = 0$;
6     $kernel\_save(d\_Z^{k-1})$;
7     $kernel\_update(d\_z^k)$;
8     $kernel\_testConverge(d_?z_{max}, d_Z^k, d_Z^{k-1})$;
```

After the initialization step, all data of the root finding problem to be solved must be copied from the CPU memory to the GPU global memory, because the GPUs only work on the data filled in their memories. Next, all the data-parallel arithmetic operations inside the main loop `(do ... while(...))` are executed as kernels by the GPU. The first kernel *save* in line(6, Algorithm 2) consist to save the vector of polynomial's root found at the previous time step on GPU memory, in order to test the convergence of the root at each iteration in line (8, Algorithme2).

The second kernel executes the iterative function and update Z(k),as formula (), we notice that the kernel update are called in two forms, separated with the value of R which determines the radius beyond which we apply the logarithm formula like this:

Algorithm 3: A global Algorithm for the iterative function

```
if ($\left|Z^{(k)}\right| <= R$) then
    $kernel\_update(d\_z^k)$;
else
    $kernel\_update\_Log(d\_z^k)$;
```

The first form execute the formula(8) if all the module's ($|Z(k)| <= R$), else the kernel execute the formulas(13,14).the radius R was computed like:

$$R = \exp(\log(DBL\_MAX)/(2 * (double)P.degrePolynome))$$

The last kernel verify the convergence of the root after each update of $Z^{(k)}$, as formula(), we used the function of the CUBLAS Library (CUDA Basic Linear Algebra Subroutines) to implement this kernel.

The kernels terminates its computations when all the root are converged. Finally, the solution of the root finding problem is copied back from the GPU global memory to the CPU memory. We use the communication functions of CUDA for the memory allocations in the GPU (`cudaMalloc()`) and the data transfers from the CPU memory to the GPU memory (`cudaMemcpyHostToDevice`) or from the GPU memory to the CPU memory (`cudaMemcpyDeviceToHost)`).

5.4 Experimental study

5.4.1 Definition of the polynomial used

We use a polynomial of the following form for which the roots are distributed on 2 distinct circles:

$$\forall \alpha_1 \alpha_2 \in C, \forall n_1, n_2 \in N^*; P(z) = (z^{n^1} - \alpha_1)(z^{n^2} - \alpha_2) \qquad (20)$$

This form makes it possible to associate roots having two different modules and thus to work on a polynomial constitute of four non zero terms. An other form of the polynomial to obtain more non zero terms is:

$$\forall \alpha_i \in C, \forall n_i \in N^*; P(z) = \sum_{p}^{i=1} (z^{n^i} - \alpha_i) \qquad (21)$$

with this formula, we can have until 2p non zero terms.

5.4.2 The study condition

In order to have representative average values, for each point of our curves we measured the roots finding of 10 different polynomials.

The our experiences results concern two parameters which are the polynomial degree and the execution time of our program to converge on the solution. The polynomial degree allows us to validate that our algorithm is powerful with high degree polynomials. The execution time remains the element-key which justifies our work of parallelization. For our tests we used a CPU Intel(R) Xeon(R) CPU E5620@2.40GHz and a GPU Tesla C2070 (with 6 Go of ram)

5.4.3 Comparative study

We initially carried out the convergence of Aberth algorithm with various sizes of polynomial, in second we evaluate the influence of the size of the threads per block....

Polynomial's degrees	T_{exe} on CPU	T_{exe} on GPU	CPU iteration	GPU iteration
5000	1.90	0.40	18	17
50000	172.723	3.92	21	18
500000	–	497.109	–	24
1000000	–	1524,51	–	24

Table 1: the convergence of Aberth algorithm

The convergence of Aberth algorithm

Tread's numbers	Execution time	Number of iteration
1024	523	27
512	449.426	24
256	440.805	24
128	456.175	22
64	472.862	23
32	830.152	24
8	2632.78	23

Table 2: The impact of the thread's number into the convergence of Aberth algorithm

The impact of the thread's number into the convergence of Aberth algorithm

Polynomial's degrees	Aberth T_{exe}	D-Kerner T_{exe}	Aberth iteration	D-Kerner iteration
5000	0.40	3.42	17	138
50000	3.92	385.266	17	823
500000	497.109	4677.36	24	214

Table 3: Aberth algorith compare to Durand-Kerner algorithm

A comparative study between Aberth and Durandkerner algorithm

References

[1] *Compute Unified Device Architecture Programming Guide Version 3.0.*

[2] O. Aberth. Iteration methods for finding all zeros of a polynomial simultaneously. *Mathematics of Computation*, 27(122):339–344, 1973.

[3] A.Ostrowski. On a theorem by j.l. walsh concerning the moduli of roots of algebraic equations,bull. a.m.s. *Algorithmes itératifs paralléles et distribués*, 1(47):742–746, 1941.

[4] C.Raphael and S.François. Extraction de racines dans des polynômes creux de degrées élevés.rsrcp (réseaux et systèmes répartis, calculateurs parallèles). *Algorithmes itératifs paralléles et distribués*, 1(13):67–81, 1990.

[5] D.A.Bini. Numerical computation of polynomial zeros by means of aberth s method. *Numerical Algorithms*, 13(4):179–200, 1996.

[6] L.Gemignani DA.Bini. Inverse power and durand kerner iterations for univariate polynomial root finding. *Comput Math Appl*, (47):447–459, 2004.

[7] E.Durand. Solution numerique des equations algebriques, vol. 1, equations du type f(x)=0, racines d'une polynome. Vol.1, 1960.

[8] G.Kahina, C.Raphael, and S.Abderrahmane. parallel implementation of the durand-kerner algorithm for polynomial root-finding on gpu. *IEEE. Conf. on advanced Networking, Distributed Systems and Applications*, pages 53–57, 2014.

[9] G.Loizon. Higher-order iteration functions for simultaneously approximating polynomial zeros. *Intern. J. Computer Math*, (14):45–58, 1983.

[10] GS.Schedler. Parallel iteration methods in complexity of computer communications. *Commun ACM*, pages 286–290, 1967.

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

[12] I.O.Kerner. Ein gesamtschritteverfahren zur berechnung der nullstellen von polynomen. (8):290–294, 1966.

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

[14] F.Spies K.Rhofir and Jean-Claude Miellou. Perfectionnements de la méthode asynchrone de durand-kerner pour les polynômes complexes. *Calculateurs Parallèles*, 10(4):449–458, 1998.

[15] 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*, 3:251–269, 1903.

[16] L.Gemignani. Structured matrix methods for polynomial root finding. *n: Proc of the 2007 Intl symposium on symbolic and algebraic computation*, pages 175–180, 2007.

[17] L.Ilieff. On the approximations of newton. *Annual Sofia Univ*, (46):167–171, 1950.

[18] M.Ben-Or, E.Feig, D.Kozzen, and P.Tiwary. A fast parallel algorithm for determining all roots of a polynomial with real roots. *Int: Proc of ACM*, pages 340–349, 1968.

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

[20] NVIDIA. *NVIDIA CUDA C Programming Guide*, volume 7 of *001*. PG, march 2015.

[21] PK.Jana. Finding polynomial zeroes on a multi-mesh of trees (mmt). *In: Proc of the 2nd int conference on information technology*, pages 202–206, 1999.

[22] PK.Jana. Polynomial interpolation and polynomial root finding on otis-mesh. *Parallel Comput*, 32(3):301–312, 2006.

[23] R.Datta Gupta PK.Jana, BP.Sinha. Efficient parallel algorithms for finding polynomial zeroes. *Proc of the 6th int conference on advance computing, CDAC, Pune University Campus,India*, 15(3):189–196, 1999.

[24] S.Winogard. Parallel iteration methods in complexity of computer communications. *Plenum, New York*, 1972.

[25] LH.Jamieson TA.Rice. A highly parallel algorithm for root extraction. *IEEE Trans Comp*, 38(3):443–449, 2006.

[26] T.L.Freeman. Calculating polynomial zeros on a local memory parallel computer. *Parallel Computing*, (12):351–358, 1989.

[27] T.L.Freeman and R.K.Brankin. Asynchronous polynomial zero-finding algorithms. *Parallel Computing*, (17):673–681, 1990.

[28] V.Skachek. Structured matrix methods for polynomial root finding. *n: Proc of the 2007 Intl symposium on symbolic and algebraic computation*, pages 175–180, 2008.

[29] W.Borch-Supan. A posteriori error for the zeros of polynomials. (5):380–398, 1963.

[30] L. W.Ehrlich. A modified newton method for polynomials. *Comm. Ass. Comput. Mach.*, (10):107–108, 1967.

[31] WL.Mirankar. Parallel methods for approximating the roots of a function. *IBM Res Dev*, 30:297–301, 1968.

[32] WL.Mirankar. A survey of parallelism in numerical analysis. *SIAM Rev*, pages 524–547, 1971.

[33] D.Lin W.Zhu, w.Zeng. an adaptive algorithm finding multiple roots of polynomials. *Lect Notes Comput Sci*, (5262):674–681, 2008.

[34] Z.Yi X.Zhanc, 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*, pages 38–41, 2008.