

A parallel root finding polynomial on GPU

Elsevier¹

Radarweg 29, Amsterdam

Ghidouche Kahina^{a,*}, Couturier Raphael^{b,*}, Abderrahmane Sider^{a,*}

^aDepartment of informatics, University of Bejaia, Algeria

^bFEMTO-ST Institute, University of Franche-Compt

Abstract

In this article we present a parallel implementation of the Aberth algorithm for the problem root finding for high degree polynomials on GPU architecture (Graphics Processing Unit).

Keywords: root finding of polynomials, high degree, iterative methods, Durand-Kerner, GPU, CUDA, CPU, Parallelization

1. Root finding problem

We consider a polynomial of degree n having coefficients in the complex C and zeros $\alpha_i, i=1, \dots, n$.

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

The root finding problem consists to find all n roots 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

$$x = g(x).$$

*Corresponding author

Email addresses: kahina.ghidouche@gmail.com (Ghidouche Kahina), raphael.couturier@univ-fcomte.fr (Couturier Raphael), ar.sider@univ-bejaia.dz (Abderrahmane Sider)

¹Since 1880.

Where $g : C^n \rightarrow C^n$. 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 $n \leq 4$, 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 direct 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 have 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:

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

This formula is mentioned for the first time from Weierstrass [1] as part of the fundamental theorem of Algebra and is rediscovered from Ilieff [2], Docev [3], Durand [4], Kerner [5]. Another method discovered from Borsch-Supan [6] and also described and brought in the following form from Ehrlich [7] and Aberth [8].

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

¹⁰ Aberth, Ehrlich and Farmer-Loizou [9] have proved that the above method has cubic order of convergence for simple roots.

Iterative methods raise several problems 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 polynomi-
als. The parallelization of these algorithms will improve the convergence time.

Many authors have treated the problem of parallelization of simultaneous
methods. Freeman [10] 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 [11] consider
asynchronous algorithms, in which each processor continues to update its ap-
proximations 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 [12] proposed two methods of parallelization for architecture with
shared memory and distributed memory, it able to compute the root of polyno-
mial 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 discuss this problem until the apparition of the Com-
pute Unified Device Architecture (CUDA) [13], 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 graph-
ics 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, [14] proposed the implementation of the Durand-Kerner method on
GPU (Graphics Processing Unit). The main result prove that a parallel imple-
mentation is 10 times as fast as the sequential implementation on a single CPU
for high degree polynomials that is greater than about 48000.

. The main part of our work is to implement the Aberth method for the prob-
lem root finding for high degree polynomials on GPU architecture (Graphics
Processing Unit). Initially we present the Aberth method in section 1. Ame-
lioration of Aberth method was proposed in section 2. A related works for the

implementation of simultaneous methods in a parallel computer was discuss in
 45 section 3. Section 4 we propose a parallel implementation of Aberth method on
 GPU. Section 5, we present our result and discuss it. Finally, in Section 6, we
 present our conclusions and future research directions.

2. Aberth method

A cubically convergent iteration method for finding zeros of polynomials
 50 was proposed by O.Aberth [8]. The Aberth method is a purely algebraic deriva-
 tion.To illustrate the derivation, we let $w_i(z)$ be the product of linear factor

$$w_i(z) = \prod_{j=1, j \neq i}^n (z - x_j) \quad (4)$$

And rational function $R_i(z)$ be the correction term of Weistrass method [1]:

$$R_i(z) = \frac{p(z)}{w_i(z)}, i = 1, 2, \dots, n \quad (5)$$

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

$$\frac{R_i(z)}{R_i'(z)} = \frac{p(z)}{p'(z) - p(z) \frac{w_i'(z)}{w_i(z)}} = \frac{p(z)}{p'(z) - p(z) \sum_{j=1, j \neq i}^n \frac{1}{z - x_j}}, i = 1, 2, \dots, n \quad (6)$$

55 Substituting x_j for z we obtain the Aberth iteration method

Let present the means stages of Aberth method.

2.1. Polynomials Initialization

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

$$p(z) = \sum a_i z^{n-i}, a_n \neq 0, a_0 = 1, a_i \in C \quad (7)$$

2.2. Vector $Z^{(0)}$ Initialization

60 The choice of the initial points $z_i^{(0)}, i = 1, \dots, n$. 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 [8] 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. 65 After, [15] performs this choice by selecting complex numbers along different circles and relies on the result of [16].

$$\sigma_0 = \frac{u+v}{2}; u = \frac{\sum_{i=1}^n u_i}{n \cdot \max_{i=1}^n u_i}; v = \frac{\sum_{i=0}^{n-1} v_i}{n \cdot \min_{i=0}^{n-1} v_i}; \quad (8)$$

Where:

$$u_i = 2 \cdot |a_i|^{\frac{1}{i}}; v_i = \frac{|\frac{a_n}{a_i}|^{\frac{1}{n-i}}}{2}. \quad (9)$$

2.3. Iterative Function H_i

The operator used with Aberth method is corresponding to the following equation which will enable the convergence towards polynomial solutions, provided all the roots are distinct. 70

$$H_i(z) = z_i - \frac{1}{\frac{P'(z_i)}{P(z_i)} - \sum_{j \neq i} \frac{1}{z_i - z_j}} \quad (10)$$

2.4. Convergence condition

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

$$\forall i \in [1, n], \frac{z_i^{(k)} - z_i^{(k-1)}}{z_i^{(k)}} < \xi \quad (11)$$

3. Amelioration of Aberth method

75 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
80 of 1 in reality. Consequently we can not compute the roots for large polynomial's degree. This problem was discuss in [17] for the Durand-Kerner method, the authors propose to use the logarithm and the exponential of a complex:

$$\forall(x, y) \in R^{*2}; \ln(x + i.y) = \ln(x^2 + y^2)2 + i. \arcsin(y\sqrt{x^2 + y^2})_{]-\pi, \pi[} \quad (12)$$

$$\forall(x, y) \in R^{*2}; \exp(x + i.y) = \exp(x). \exp(i.y) \quad (13)$$

$$= \exp(x). \cos(y) + i. \exp(x). \sin(y) \quad (14)$$

The application of logarithm can replace any multiplications and divisions with additions and subtractions. Consequently, it manipulates lower absolute
85 values and can be compute the roots for large polynomial's degree exceed [17].

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

$$H_i(z) = z_i^k - \exp \left(\ln(p(z_k)) - \ln(p(z'_k)) - \ln(1 - Q(z_k)) \right) \quad (15)$$

Where:

$$Q(z_k) = \exp \left(\ln(p(z_k)) - \ln(p(z'_k)) + \ln \left(\sum_{k \neq j}^n \frac{1}{z_k - z_j} \right) \right) \quad (16)$$

This solution is applying when it is necessary

4. The implementation of simultaneous methods in a parallel computer

90 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
95 of iterations as implemented by Miranker [18, 19], Schedler [20] and Winog-
ard [21]. Another approach is to reduce the computation time per iteration, as
reported in [22, 23, 24, 25]. There are many schemes for simultaneous approxi-
mations of all roots of a given polynomial. Several works on different methods
and issues of root finding have been reported in [26, 27, 28, 29, 30, 31]. How-
100 ever, Durand-Kerner and Ehrlich methods are the most practical choices among
them [32]. 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
105 method and cubic for the Ehrlich). Various parallel algorithms reported for
these methods can be found in [33, 10, 11? , 34, 24]. Freeman and Bane [11]
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
110 at the end of each iteration. Therefore they cause a high degree of memory con-
flict. Recently the author in [19] proposed two versions of parallel algorithm for
the Durand-Kerner method, and Aberth method 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
115 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
120 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 com-
puting. In [14] we proposed the first implantation of the root finding polyno-
mials method on GPU (Graphics Processing Unit),which is the Durand-Kerner

125 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 poly-
nomials that is greater than about 48000. Indeed, in this paper we present a
parallel implementation of Aberth method on GPU, more details are discussed
in the following of this paper.

130 **5. A parallel implementation of Aberth method**

5.1. Background on the GPU architecture

A GPU is viewed as an accelerator for the data-parallel and intensive arith-
metic computations. It draws its computing power from the parallel nature of
its hardware and software architectures. A GPU is composed of hundreds of
135 Streaming Processors (SPs) organized in several blocks called Streaming Mul-
tiprocessors (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 [35]

On a CPU equipped with a GPU, all the data-parallel and intensive functions
140 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)
145 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
150 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.

155 *5.2. Background on the CUDA Programming Model*

The CUDA programming model is similar in style to a single program multiple-data (SPMD) software model. 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 multithreaded 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.

160
165
170

5.3. The implementation of Aberth method on GPU

175 5.3.1. A sequential Aberth algorithm

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

Algorithm 1: Algorithm to find root polynomial with Aberth method

Input: Z^0 (Initial root's vector), ϵ (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
180 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:

$$H(i, z^{k+1}) = \frac{p(z_i^{(k)})}{p'(z_i^{(k)}) - p(z_i^{(k)}) \sum_{j=1, j \neq i}^n \frac{1}{z_i^{(k)} - z_j^{(k)}}}, i = 1, \dots, n. \quad (17)$$

Or with the Gauss-seidel iteration, we have:

$$H(i, z^{k+1}) = \frac{p(z_i^{(k)})}{p'(z_i^{(k)}) - p(z_i^{(k)}) \sum_{j=1}^{i-1} \frac{1}{z_i^{(k)} - z_j^{(k+1)}} + \sum_{j=i+1}^n \frac{1}{z_i^{(k)} - z_j^{(k)}}}, i = 1, \dots, n. \quad (18)$$

185 In formula(16), the Gauss-seidel iteration converge more quickly because they used the most fresh computed root z_i^{k+1} , at this reason we used Gauss-seidel iteration.

The steps 4 of the Aberth method compute the convergence of the roots, using(9) formula. Both steps 3 and 4 use 1 thread to compute N roots on CPU, 190 which is harmful 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.

195 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:

$$T_{exe} = N(T_i(N) + T_j) + O(n). \quad (19)$$

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

$$Total_time_{exe} = [N(T_i(N) + T_j) + O(n)] .Nbr_iter \quad (20)$$

200 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

205 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 assign the execution of this loop to a groups of parallel threads organized as a grid of blocks each block contain a number of threads. 210 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:

$$Total_time_{exe} = \left[\frac{N}{nbr_thread} (T_i(N) + T_j) + O(n) \right] .Nbr_iter. \quad (21)$$

In theory, the $Total_time_{exe}$ on GPU is speed up `nbr.thread` times as a 215 $Total_time_{exe}$ 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.

220 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), ϵ (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 $kernel_testConverge(d_Z^k, d_Z^{k-1})$;
-

After the initialization step, all data of the root finding problem to be solved
 225 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
 230 memory, in order to test the convergence of the root at each iteration in line (8,
 Algorithm 2).

The second kernel executes the iterative function and update $Z(k)$, as formula
 (1), 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

235 formula like this:

Algorithm 3: A global Algorithm for the iterative function

```

if ( $|Z^{(k)}| \leq R$ ) then
    | kernel_update( $d \cdot z^k$ );
else
    | kernel_update_Log( $d \cdot z^k$ );

```

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

$$R = \exp(\log(DBL_MAX)/(2 * (double)P.degreePolynome))$$

240 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 245 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:

$$\forall \alpha_1 \alpha_2 \in C, \forall n_1, n_2 \in N^*; P(z) = (z^{n_1} - \alpha_1)(z^{n_2} - \alpha_2) \quad (22)$$

250 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 a full polynomial is:

$$\forall a_i \in C; p(x) = \sum_{i=1}^{n-1} a_i \cdot x^i \quad (23)$$

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

5.4.2. The study condition

255 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
 260 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
 265 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
10000	172.723	0.59	21	24
20000	172.723	1.52	21	25
30000	172.723	2.77	21	33
50000	172.723	3.92	21	18
500000	>1h	497.109		24
1000000	>1h	1,524.51		24

Table 1: the convergence of Aberth algorithm

Aberth algorithm on CPU and GPU.

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

Thread'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

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 algorithm compare to Durand-Kerner algorithm

A comparative study between Aberth and Durand-kerner algorithm.

- 270 [1] 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.
- [2] L. Ilieff, On the approximations of newton, Annual Sofia Univ (46) (1950) 167–171. doi:10.1016/0003-4916(63)90068-X.
- 275 [3] 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.

- [4] E. Durand, Solution numerique des equations algebriques, vol. 1, equations du type $f(x)=0$, racines d'une polynome Vol.1.
- 280 [5] I. Kerner, Ein gesamtstritteverfahren zur berechnung der nullstellen von polynomen (8) (1966) 290–294.
- [6] W. Borch-Supan, A posteriori error for the zeros of polynomials (5) (1963) 380–398.
- [7] L. Ehrlich, A modified newton method for polynomials, Comm. Ass. Comput. Mach. (10) (1967) 107–108.
- 285 [8] O. Aberth, Iteration methods for finding all zeros of a polynomial simultaneously, Mathematics of Computation 27 (122) (1973) 339–344. doi:10.1016/0003-4916(63)90068-X.
- [9] G. Loizon, Higher-order iteration functions for simultaneously approximating polynomial zeros, Intern. J. Computer Math (14) (1983) 45–58.
- 290 [10] T. Freeman, Calculating polynomial zeros on a local memory parallel computer, Parallel Computing (12) (1989) 351–358.
- [11] T. Freeman, R. Brankin, Asynchronous polynomial zero-finding algorithms, Parallel Computing (17) (1990) 673–681.
- 295 [12] R. Couturier, F. Spetiri, Extraction de racines dans des polynomes creux de degres levs.rsrcp (rseaux et systmes rpartis, calculateurs parallles), Algorithmes itratifs parallles et distribus 1 (13) (1990) 67–81.
- [13] Compute Unified Device Architecture Programming Guide Version 3.0.
- [14] K. Ghidouche, R. Couturie, 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.
- 300

- [15] D. Bini, Numerical computation of polynomial zeros by means of aberth s method, Numerical Algorithms 13 (4) (1996) 179–200.
- 305 [16] A. Ostrowski, On a theorem by j.l. walsh concerning the moduli of roots of algebraic equations,bull. a.m.s., Algorithmes itratifs parallles et distribus 1 (47) (1941) 742–746.
- [17] K. Rhofir, F. Spies, J.-C. Miellou, Perfectionnements de la mthode asynchrone de durand-kerner pour les polynmes complexes, Calculateurs Parallles 10 (4) (1998) 449–458.
- 310 [18] W. Mirankar, Parallel methods for approximating the roots of a function, IBM Res Dev 30 (1968) 297–301.
- [19] W. Mirankar, A survey of parallelism in numerical analysis, SIAM Rev (1971) 524–547.
- 315 [20] G. Schedler, Parallel iteration methods in complexity of computer communications, Commun ACM (1967) 286–290.
- [21] S. Winogard, Parallel iteration methods in complexity of computer communications, Plenum, New York.
- [22] M. Ben-Or, E. Feig, D. Kozzen, P. Tiwary, A fast parallel algorithm for determining all roots of a polynomial with real roots, Int: Proc of ACM 320 (1968) 340–349.
- [23] P. Jana, Polynomial interpolation and polynomial root finding on otis-mesh, Parallel Comput 32 (3) (2006) 301–312.
- [24] 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.
- 325 [25] T. Rice, L. Jamieson, A highly parallel algorithm for root extraction, IEEE Trans Comp 38 (3) (2006) 443–449.

- [26] H. Azad, The performance of synchronous parallel polynomial root extraction on a ring multicomputer, *Clust Comput* 2 (10) (2007) 167–174.
330
- [27] L. Gemignani, Structured matrix methods for polynomial root finding., n: Proc of the 2007 Intl symposium on symbolic and algebraic computation (2007) 175–180.
- [28] B. Kalantari, Polynomial root finding and polynomiography., World Scientific, New Jersey.
335
- [29] V. Skachek, Structured matrix methods for polynomial root finding., n: Proc of the 2007 Intl symposium on symbolic and algebraic computation (2008) 175–180.
- [30] 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.
340
- [31] W. Zhu, w. Zeng, D. Lin, an adaptive algorithm finding multiple roots of polynomials, *Lect Notes Comput Sci* (5262) (2008) 674–681.
- [32] D. Bini, L. Gemignani, Inverse power and durand kerner iterations for univariate polynomial root finding, *Comput Math Appl* (47) (2004) 447–459.
345
- [33] M. Cosnard, P. Fraigniaud, Finding the roots of a polynomial on an mimd multicomputer, *Parallel Comput* 15 (3) (1990) 75–85.
- [34] 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.
350
- [35] NVIDIA, NVIDIA CUDA C Programming Guide, Vol. 7 of 001, PG, 2015.