A parallel implementation of the Durand-Kerner algorithm for polynomial root-finding on GPU

Kahina Ghidouche[1], Raphaël Couturier[2] and Abderrahmane Sider[1]

[1]LIMED Laboratory, University A-Mira of Bejaia,
Targa Ouzemour streets, Bejaia, Algerie
Kahina.ghidouche@gmail.com, ar.sider@univ-bejaia.dz

[2]FEMTO-ST Institute, University of Franche Comte, IUT
Belfort-Montbéliard, 19 Av. du Maréchal Juin, BP 527,
90016 Belfort CEDEX, France
raphael.couturier@univ-fcomte.fr

***Abstract*— In this article we present a parallel implementation of the Durand-Kerner algorithm to find roots of polynomials of high degree on a GPU architecture (Graphics Processing Unit). We have implemented both a CPU version in C and a GPU compatible version with CUDA. The main result of our work is a parallel implementation that is 10 times as fast as its sequential counterpart on a single CPU for high degree polynomials that is greater than about 48,000.**

Keywords*— *polynomial root-finding, high degree, iterative methods, Durant-Kerner method, GPU, CUDA, Parallelization.

I. INTRODUCTION

The root finding problem consists in retrieving all values of a real or complex variable x verifying $p(x) = 0$. p is a real/complex function in R^n *or* C^n. In this article we suppose that p is a polynomial of degree n, that is to say, p is of the form $p = \sum a_i x_i$ i=0,…,n-1 where a_i is a real/complex constant called the i^{-th} coefficient and $a_{n-1} \neq 0$. It is well known that the number of roots of a polynomial of degree n is exactly n.

The issue of finding the roots of polynomials of very high degrees arises in many complex problems in various fields, such as algebra, biology, finance, physics or climatology [1]. In algebra for example, finding eigenvalues or eigenvectors of any real/complex matrix amounts to finding the roots of the so-called characteristic polynomial.

Different methods of resolution exist for polynomial root-finding and they usually are classified in direct and iterative methods. Direct methods for finding root of polynomials only exist for $n=1$ and $n=2$. But for larger degrees, approximation methods are the only way to solve them. An approximation or iterative method usually starts with an initial solution (the initial guess) that is successively evolved until the roots are approximated with a certain precision. Traditionally, the Newton method serves to iteratively solve fixed point problems of the form $x = f(x)$. For the FRP case, The Newton-Raphson method is used by simply transforming the problem of solving $p(x)=0$ into one of form $x=f(x)$.

However, the extraction of roots of polynomial is a very expensive process in execution time. For example, in [6] authors reported execution times of 3,300s for a polynomial of degree 40,000.

Graphics Processing Units (GPU), equipping personal computers, once used primarily for image processing operations have, nevertheless, seen a tremendous evolution of their computation power that resulted in scientists and engineers turning to them to benefit from their huge capabilities in always higher performance demanding applications. This led to a new programming context called General Processing GPU Computing (GP-GPU). Consequently, very important savings of time with many scientific applications have been successfully obtained [3,4].

The main objective of this work is to parallelize a well known algorithm for the computation of polynomial zeros named the Durand-Kerner method, and to experimentally study its performance on a GPU architecture using various high degree polynomials.

The following of this paper is organized as follows. In Section II, we recall the mathematical description of the Durand-Kerner method. Then we present GPUs and their programming in section III. Afterwards, we detail the parallelization of the algorithm and its implementation in the IVth section. Finally we analyze experimental results, allowing us to conclude our work.

II. THE DURAND-KERNER METHOD

An iterative method proceeds by successively refining initial solution x^0 until it converges. The solution of an iterative method, usually noted x^* verifies $x^* = f(x^*)$. Generally, an iterative algorithm has the following form:

$$x^{k+1} = f(x^k) \quad for\, k = 1,2,\cdots \tag{1}$$

where x^k is the solution at iteration k, x^{k+1} the solution at iteration $k+1$ and f is the iterative function. The Durand-Kerner method consists of four principal phases [5,6] that are : initialization of the polynomial, initialization of the solution, applying the iterative function H_i and a termination condition.

A. *Phase 1 : initialization of the polynomial P(Z)*

The initialization of the polynomial with complex coefficients $P(z)$ is carried out as follows:

$$\forall a_i \in C, P(z) = \sum_{i=0}^{n} a_i . z^{n-i} \text{ with } (a_0 = 1, a_n \neq 0) \tag{2}$$

B. Phase 2 : initialization of the vector $Z^{(0)}$

The second phase of the method consists in initializing the vector $Z^{(0)}$. This initialization is important because the components of the vector must be different from each other. To achieve this, the Gugenheimer method is used in this work. A radius σ is determined from the polynomial coefficients such that initial roots are placed at equidistance on a circle of radius σ. The computation of σ is carried according to Equation (3) where *u* and *v* are in turn computed as shown in Equation (4) where each u_i and each v_i is the result of Equation (5).

$$\sigma = \frac{u+v}{2} \tag{3}$$

$$u = \frac{\sum_{i=1}^{n} u_i}{n.\max_{i=1} u_i} \quad v = \frac{\sum_{i=0}^{n-1} v_i}{n\text{-}1.\max_{i=0} v_i} \tag{4}$$

$$u_i = 2.|a_i|^{\frac{1}{i}} \quad v_i = \frac{1}{2}.\left|\frac{a_n}{a_i}\right|^{\frac{1}{n-i}} \tag{5}$$

Then the initial guesses for the *n* roots are evenly placed around the circle of radius σ:

$$z_j^{(0)} = (\cos\theta_j + i\sin\theta_j).\sigma \quad j = 0,..., n-1, \tag{6}$$

Where

$$\theta_j = j\frac{2\pi}{n}. \tag{7}$$

C. Phase 3 : Applying the Hi iterative function

The third phase of the method is to apply the iterative function *Hi* which will make it possible to converge to roots solution of the polynomial, provided that all the roots are different.

$$\forall i \in [1,n]; H_i(z) = z_i - \frac{P(z_i)}{\prod_{\substack{j=1 \\ j\neq i}}^{j=n}(z_i - z_j)} \tag{8}$$

D. Phase 4 : A termination condition

At the end of each application of H_i, a stop condition is verified. We have two possibilities to implement it.

- The first solution consists in stopping the iterative process when the whole of the modules of the roots are lower than a fixed value ε, that is :

$$\forall i \in [1,n]; \ |P(z_i)| < \varepsilon. \tag{9}$$

- In the second solution, we stop the iterative function when the roots are stable, i.e. the method converges sufficiently:

$$\forall i \in [1,n]; \frac{z_i^{(k)} - z_i^{(k-1)}}{z_i^{(k)}} < \varepsilon. \tag{10}$$

It should be noticed that our algorithm follows the principle of improvement of the method of Durand-Kerner described in [5,6]. Thus, when the evaluation of a polynomial leads to numbers exceeding the storage capacity of the double type, we evaluate the logarithm of the polynomial. This solution is interesting to find the roots of high degree polynomials.

III. PARALLELISATION OF THE DURAND-KERNER ALGORITHM

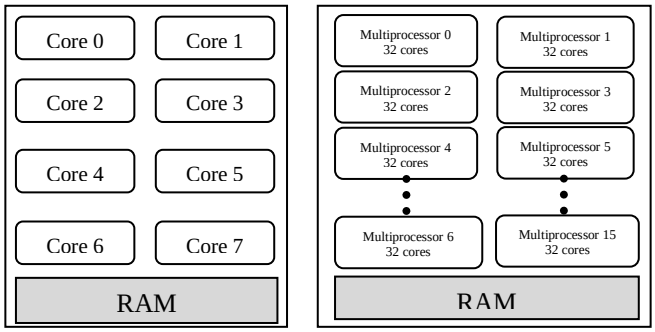
Before dwelling on the parallelization, we present the GPU architecture and the CUDA (Compute Unified Device Architecture) platform which are the tools we have used to parallelize the Durand-Kerner algorithm.

A. The GPU architecture

The graphics processor of GPUs was initially devised to process graphic applications and 3D displays; that is say, to ensure visualization functions. For example, products like GeForce and Quadro, two ranges of GPUs proposed by nVIDIA, are respectively intended for the graphics general public and professional visualization.

A GPU is a graphic processor connected with a traditional processor (CPU) via a PCI-Express interface (see Fig 1.). It is often considered as an accelerator of intensive arithmetic operations of an application executed on a CPU. It derives its computing power from its massively parallel architecture. Indeed, unlike CPU architecture, a GPU is composed of hundreds or even thousands of streaming processors (SP), also called cores, and organized in several blocks of processors called multi-processors (SM or SMX). Fig2 shows a comparison between the architecture of a CPU and that of a Fermi GPU [7].

Fig .1. Example of CPU equipped with GPU.



A CPU with 8 Cores A Fermi GPU with 512 Cores

Fig. 2. Comparison of the number of cores in a CPU and a GPU

The massively parallel architecture of GPUs offers performances and very interesting computing capacities to solve new complex problems of ever increasing sizes. In order to use these GPUs, several platforms have been developed to facilitate their programming and usage. CUDA from nVIDIA and OpenCL from the consortium Khronos group are two very popular development tools that make it easier to write parallel programs on those target GPUs. In this work, we use CUDA,

which is the reason for which the next subsection recalls some of its basic properties.

B. *CUDA architecture*

CUDA has been developed by NVIDIA and it enables developers to increase the performances of their computing programs by exploiting the huge computing power of the graphic processors.

It is based on the C/C++ programming language with some extensions that admit the expression of dense and complex data in a context of parallelism. An application written in CUDA is a heterogeneous program that executes on a processor (CPU) equipped with a graphics board (GPU). Indeed, in a CUDA program, the codes to be executed by the CPU are separately defined from those to be executed by the GPU. All the intensive arithmetic operations 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. In addition, the CPU executes all the sequential operations that cannot be executed in parallel and controls the execution of the kernels on the GPU as well as data communication between the CPU memory and the GPU memory.

CUDA is based on the model of parallel programming single instruction multiple threads SIMT (Single Instruction, Multiple Thread) model. So, each kernel is executed in parallel by thousands, even millions, of threads. At the level of a GPU, the threads of the same kernel are organized in grids of several blocks of threads which are distributed more or less equitably, on the whole of the multiprocessors of the GPU (see Fig .3) [3].

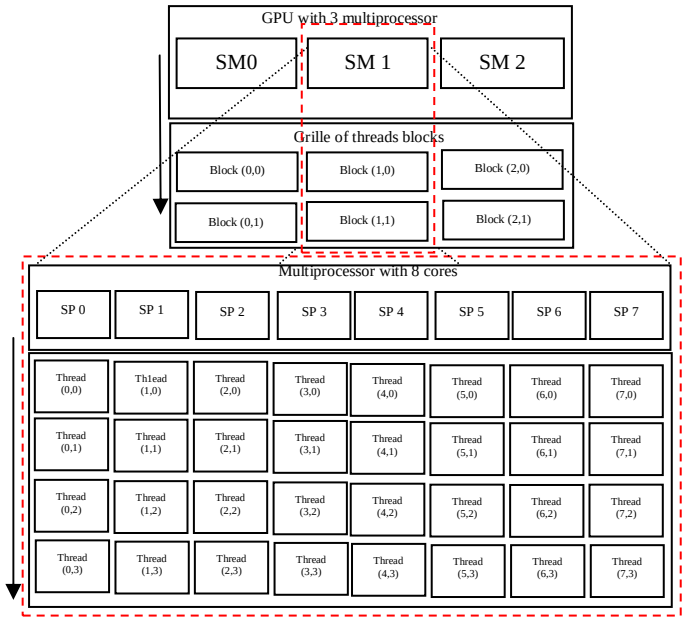


Fig. 3. Example block execution threads on a two-dimensional on GPU with 3 multiprocessors (8 cores)

C. *Parallel version of the Durand-Kerner algorithm*

Like any parallel code, a GPU parallel implementation first requires to determine the sequential tasks and the parallelizable parts of the sequential version of the program/algorithm. In our case, all the operations that are easy to execute in parallel must be made by the GPU to accelerate the execution of the application. On the other hand, all the sequential operations and the operations that have data dependencies between threads or recursive computations must be executed by only one CUDA or CPU thread. In general all the data must stay on the GPU because memory transfers are expensive.

In our case we parallelized phase 3 and phase 4 of the Durand-Kerner method. For phase 3 we have two kernels, the first named *save* is used to save vector Z^{K-1} and the kernel *update* is used to update the Z^k vector. In phase 4 a kernel is created to test the convergence of the method. In order to compute function H, we have two possibilities: either to use the Jacobi method, or the Gauss-Seidel method which uses the most recent computed roots. It is well known that the Gauss-Seidel mode converges more quickly. So, for both versions of the algorithm we used the Gauss-Seidel mode of iteration. To parallelize the code, we created kernels and many functions to be executed on the GPU for all the operations dealing with the computation on complex numbers and the evaluation of the polynomials. As said previously, we managed both functions of evaluation of a polynomial: the normal method, based on the method of Horner and the method based on the logarithm of the polynomial. All these methods were rather long to implement, as the development of corresponding kernels with CUDA is longer than on a CPU host. This comes in particular from the fact that it is very difficult to debug CUDA running threads like threads on a CPU host. In the following paragraph Algorithm 1 shows a sequential CPU implementation whereas Algorithm 2 shows the GPU parallel version.

1 Compute initial values $\{z_0;\ldots; z_{n-1}\}$
2 Let k = 1;
3 do
3 *let* $\Delta z_{max} = 0$;
4 for j = 0… n-1
5 $z_j^{k-1} = z_j^k$;
6 $z_j^k = H_i(z^{k-1})$;
7 set $\Delta z_{max} = \dfrac{\left|z_j^k - z_j^{k-1}\right|}{z_j^k}$
8 k=k+1;
9 while $z_{max} > \varepsilon$;

Algorithm 1. A Durand-Kerner algorithm sequential implementation on CPU

```
1 Compute initial values {z0;…; zn-1}
2 do
3   let Δz max = 0;
4 copy Z, Δzmaxi into the device memory
d_Z,d_Δzmax
5 <DimGrid,DimBloc> kernel_save(d_Z^k-1);
6 <DimGrid,DimBloc> kernel_update(d_z^k);
7 <DimGrid,DimBloc> kernel_ testConverge
(d_Δzmax,d_Z^k,d_Z^k-1);
8 k=k+1;
9 while Δzmax>ε;
10 copy the result into the host memory
```

Algorithm 2. A Durand-Kerner algorithm parallel implementation on GPU

For the GPU version, kernels serve to make the computations of lines 5, 6 and 7 of the CPU algorithm. We can notice that the "for loop" does not exist anymore in the parallel version because kernels are executed by all the threads. In each kernel call in the code, we recall that there are indeed DimGrid blocks of threads consisting of DimBloc threads by block. Thus, we use as many threads as the number of roots of the studied polynomial. It results that each thread computes one root at time. To achieve this, we adapt the values of DimGrid and DimBloc according to the studied polynomial, and before launching the kernel.

In what follows, we report the CPU and GPU results of an experimental study carried on different high degree polynomials.

IV. EXPERIMENTS

A. Definition of the polynomial used

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

$$\forall \alpha_1, \alpha_2 \in C, \forall\, m_1, m_2 \in N^*;\ P(z) = (z^{m1} - \alpha_1)(z^{m2} - \alpha_2) \qquad (11)$$

This form allows to associate roots having two different modules and thus to work on a polynomial constituted of four nonzero terms.

B. Study condition

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

In our experiments two parameters are studied: 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 of course 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).

C. Comparative study

We initially carried out a comparative study between the Gauss-Seidel iterations and the Jacobi iterations for both sequential and parallel versions. Then we carried out a test with various sizes of polynomials. Finally we evaluated the influence of the size of the threads blocks.

1) *A comparative study between Gauss seidel iteration and Jacobi iteration*

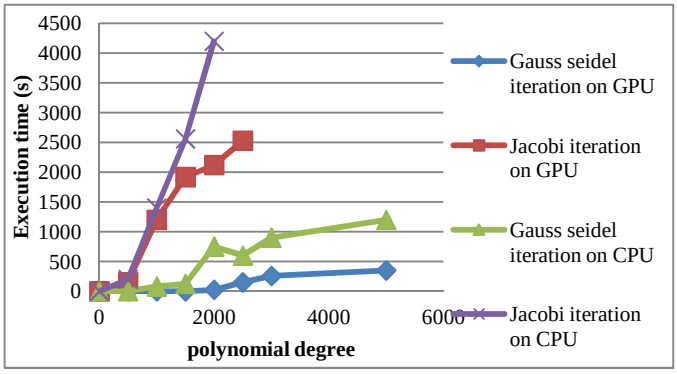


Fig . 4 A comparative study of both the Gauss seidel and the Jacobi iteration for the Durand-Kerner algorithm on GPU and CPU

Figure 4 shows a comparison between the Gauss Seidel iterations and the Jacobi iterations for both the parallel and sequential versions of the algorithms. We clearly see that the Gauss Seidel method converges faster than the Jacobi iteration which has a very slow convergence rate.

2) The Durand-Kerner algorithm with the high degree polynomials.

In this experimentation we compare the sequential and parallel versions of the Durand-Kerner algorithm with high degree polynomials, i.e. more than 20,000. Figure 5 shows the execution times for polynomials of various degrees.

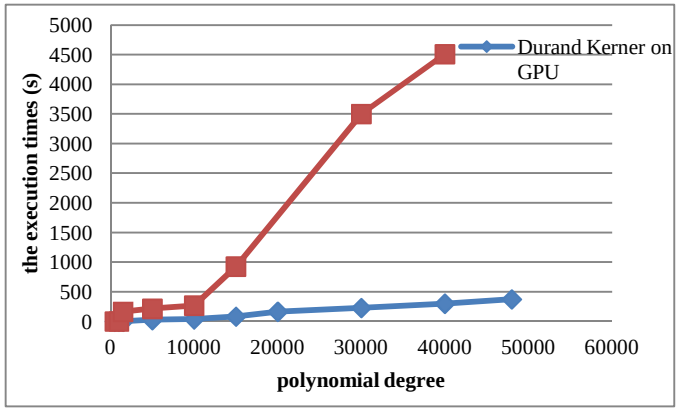


Fig. 5. Comparison of the execution times of the Durand-Kerner algorithm on CPU and GPU

This curve shows a comparison of execution times between the parallel and sequential version of the Durand-Kerner algorithm with polynomial degrees ranging from 500 to 50,000. During our test we noticed that the parallel version executes a polynomial of size 48,000 in 373.944s whereas the sequential version requires 4,510.22s. So, the parallel version is executed approximately 10 times faster than the sequential version. Furthermore, we verified that the number of iterations is the same. This reduction of time allows us to compute roots

of polynomial of more important degrees at the same time than with a CPU.

3) Influence of the number of threads per block

It is also interesting to see the influence of the number of threads per block on the execution time. For that, we chose 10 different polynomials of degree 35,000.

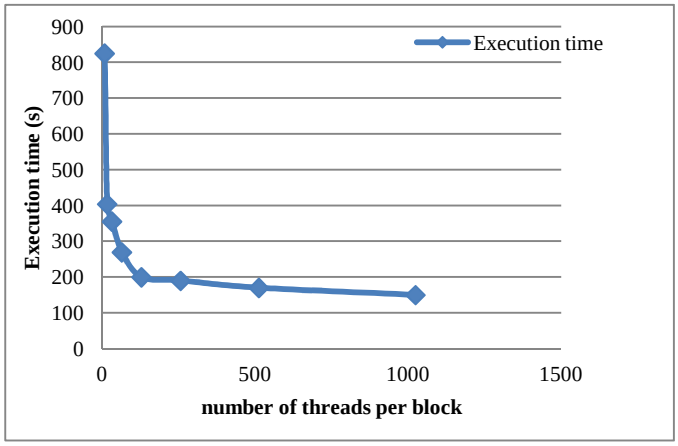


Fig .6. Influence of the number of threads per block on the execution time.

The curve shows the impact of a different number of threads per block over the execution time. The number of threads per block varies from 8 to 1,024. It can be noticed that the larger this one is, the more the execution time decreases. On the graph we can remark that the best execution time is around 150s with 1024 threads per bloc.

V. CONCLUSION AND PROSPECTIVE

The GPU architecture is well adapted to intensive computing. We parallelized the Durand-Kerner algorithm for polynomial roots-finding and we obtained encouraging results. Indeed, the experimental study confirms that our program determines the same roots than the sequential version for high degrees. The contribution of the parallel solution allows us to accelerate the execution time and to study even more important degrees of polynomial. The parallelization of this type of application is thus completely justified and confirmed by a speed up of 10.

As perspective, it is interesting to compute and find more roots so we think to develop this solution on several GPUs with a cluster of GPUs. Finally, we plan to study other algorithms of roots finding polynomials and their implantation on GPUs.

VI. REFERNECES

[1] Bini Dario Andrea. Numerical computation of polynomial zeros by means of Aberth's method. Numerical Algorithms 13 (1996), 179–200.

[2] Sagraloff Michael., and Chee. K Yap. A simple but exact and efficient algorithm for complex root isolation. In Proceedings of ISSAC'2011 (2011), pp. 353–360.

[3] Raphaël Couturier, Designing Scientific Applications on GPUs, Numerical Analysis and Scientific Computing, 2013. Chapman and Hall/CRC

[4] Lilia Ziane Khodja, Ming Chau, Raphaël Couturier, Jacques Bahi, Pierre Spitéri, Parallel solution of American option derivatives on GPU clusters, Computers and Mathematics with Applications 65(11): 1830-1848 (2013)

[5] Karim Rhofir, François Spies, and Jean-Claude Miellou. Perfectionnements de la méthode asynchrone de Durand-Kerner pour les polynômes complexes. Calculateurs Parallèles, 10(4):449--458, 1998.

[6] Raphaël Couturier, François Spies. Extraction de racines dans des polynômes creux de degré élevé. RSRCP (Réseaux et Systèmes Répartis, Calculateurs Parallèles), Numéro thématique : Algorithmes itératifs parallèles et distribués, 13(1):67--81, 2001.

[7] NVIDIA Corporation. NVIDIA CUDA C Programming Guide.Version4.2,2012.http://developer.download.nvidia.com/compute/DevZone/docs/html/C/doc/CUDA C Programming Guide.pdf.