Solving Multiple Quartic Equations on the GPU using Ferrari's Method

Sándor Szénási
John von Neumann Faculty of Informatics
Óbuda University
Budapest, Hungary
Email: szenasi.sandor@nik.uni-obuda.hu

Ákos Tóth
Donát Bánki Faculty of Mechanical and Safety Engineering
Óbuda University
Budapest, Hungary
Email: toth.akos@bgk.uni-obuda.hu

***Abstract*—As known, quartics are the highest degree polynomials which can be solved analytically in general by the methods of radicals. There are several problems based on not only one but more equations independently, in case of simulations, the number of equations can be very high. For this reason, it is worth examining the runtime of the solver algorithms implemented for multi-core systems, especially graphics accelerators. In this paper, we discuss the runtime and numerical stability of the Ferrari's method using GPUs. It is worth to port an application to the graphics card, if the number of calculations is relatively high and the number and volume of memory accesses is relatively small. Based on the results, it is clear, that running multiple equation solvers based on the given method is clearly meets these conditions.**

I. Introduction

Several problems need quartic equations to be solved. Beyond computer graphics [1]–[4], there are several scientific areas where these equations play an important role. Like astronomy (transformation of geocentric to geodetic coordinates without approximations [5]), biochemistry (theoretic analysis of monocyclic cascade models [6]), physics (analysis of chiral perturbation theory [7], analytical solution of tt dileption equations [8]), geography (quantitative analysis of land surface topography [9], [10]), and cryptography [11].

It is important to note, that most of these problems need to solve more than one equation independently [12]. In case of simulations, the number of equations can be very high. For this reason, it is worth examining the runtime of the solver algorithms implemented for multiprocessor systems, especially graphics accelerators [13].

There are various techniques to solve these problems; however, in this paper, we discuss only analytic solutions. We will examine the runtime and numerical stability of the Ferrari method using NVIDIA GPUs. In mathematics, a quartic function, or equation of the fourth degree, is a function of the form:

$$f(x) = ax^4 + bx^3 + cx^2 + dx + e \tag{1}$$

where a is nonzero; or in other words, a polynomial of degree four.

As known, quartics are the highest degree polynomials which can be solved analytically in general by the methods of radicals, i.e., operating on the coefficients with a sequence of operators from the set: sum, difference, product, quotient, and the extraction of an integral order root [14]. This means that these equations can be solved by analytic algorithms, and need no iterative implementations.

All existing analytic methods (e.g. Descartes-Euler-Cardano's, Ferrari-Lagrange's, Neumark's, Christianson-Brown's, and Yacoub-Fraidenraich-Brown's) are particular versions of the same universal method [9]. Many algorithms are based on the idea of previously solving a particular cubic equation, the coefficients of which are derived from those of the original quartic. The root of this cubic equation is then used to factorise the quartic into quadratics, which can then be solved by this way [14].

II. Ferrari method

Lodovico Ferrari is known as the person, who found the first solution for this problem in 1540. However, this method (like all algebraic variants) requires the solution of a cubic to be found, therefore it was published later in the book Ars Magna [15]. A short overview about the Ferrari method:

Given the following quartic equation:

$$Ax^4 + Bx^3 + Cx^2 + Dx + E = 0 \tag{2}$$

First, we have to calculate the following intermediate results:

$$\alpha = -\frac{3B^2}{8A^2} + \frac{C}{A} \tag{3}$$

$$\beta = \frac{B^3}{8A^3} - \frac{BC}{2A^2} + \frac{D}{A} \tag{4}$$

$$\gamma = -\frac{3B^4}{256A^4} + \frac{CB^2}{16A^3} - \frac{BD}{4A^2} + \frac{E}{A}. \tag{5}$$

The next step is based on the value of β. In the case when β is zero, we can easily calculate the final results (Eq. 6)

$$x = -\frac{B}{4A} \pm_s \sqrt{\frac{-\alpha \pm_t \sqrt{\alpha^2 - 4\gamma}}{2}} \qquad \text{(for } \beta = 0 \text{ only)} \tag{6}$$

Otherwise, we have to continue the calculations with the followings:

$$P = -\frac{\alpha^2}{12} - \gamma \tag{7}$$

$$Q = -\frac{\alpha^3}{108} + \frac{\alpha\gamma}{3} - \frac{\beta^2}{8} \tag{8}$$

$$R = -\frac{Q}{2} \pm \sqrt{\frac{Q^2}{4} + \frac{P^3}{27}} \tag{9}$$

$$U = \sqrt[3]{R} \tag{10}$$

$$y = \begin{cases} -\frac{5}{6}\alpha + U - \frac{P}{3U} & \text{if } U \neq 0 \\ -\frac{5}{6}\alpha + U - \sqrt[3]{Q} & \text{if } U = 0 \end{cases} \tag{11}$$

$$W = \sqrt{\alpha + 2y} \tag{12}$$

$$x = -\frac{B}{4A} + \frac{\pm_s W \mp_t \sqrt{-\left(3\alpha + 2y \pm_s \frac{2\beta}{W}\right)}}{2} \tag{13}$$

Some further remarks about the calculations:

- At Eq. 9, any of the two square roots will do.
- At Eq. 10, any of the three complex roots will do.

III. GPU implementation of the algorithm

The first reason why it is worth executing algorithms in the GPU is the enormous computing capacity of these devices. The peak performance of the graphics card is about two magnitudes higher than the already available multi-core CPUs. The main reasons for this high peak performance are as follows:

- High number of processing units (cores).
- Complex and very effective memory architecture.

Similar to CPUs, in the case of newer GPUs, there are increasingly more processing units. It is worthwhile noticing that in this case we talk about quite different dimensions, the number of CUDA cores in a Kepler based Tesla K40 graphics accelerator is 2880, compared to the common 8 cores of traditional CPUs. It is obvious that we have to start at least as many threads as the number of processing units to utilize the graphics card. The latency hiding mechanism of the GPU is based on the fast context switching, therefore we have to run 3-4 times more threads parallel to reach the peak performance.

Another limitation of GPU development is the separated CPU-GPU memory hierarchy. Both the GPU and the CPU have their own memory area, and neither of them can work in the other's memory area. This means that we have to copy data from host memory to device memory and vice versa. This coping can degrade performance.

There are several cases when it is hard to decide whether an algorithm will be faster in the GPU than in the CPU, or not. This depends on several hardware (number/size/speed of memory/bus/execution units) and software (number of threads, independency of threads) aspects. However there is a general rule that can help in this decision: the ratio of calculations per data movement is as the higher GPU will perform better. From this point of view, porting the Ferrari method to the GPU promises good results. Solving the equation is quite calculation intensive (there are several floating point operations), but the volume of the input-output data is not significant.

Based on the foregoing, we developed a GPU kernel using CUDA C language to solve quartic equations. The system is able to solve multiple equations in parallel. We developed the same application for the CPU using the standard C language. The next section contains the comparison of the two applications. The Ferrari method uses complex values, therefore both of the implemented algorithms are based on these data types. Fortunately, in both environments we can use similar instructions for complex number management.

IV. Test results

We used the following configurations for the tests:

- CPU configuration
  - Processor: Intel® Core™ i7-2600
  - Architecture: Sandy Bridge
  - Number of cores: 4
  - Memory: 16GB DDR2
- GPU configuration
  - Graphics accelerator: NVIDIA Tesla K40c
  - Architecture: Kepler (GK110B)
  - Number of shaders: 2880
  - SMX Count: 15
  - Memory: 12GB GDDR5
  - Host: the same as the CPU config

A. *Single-precision floating points*

In the first test, we checked the runtime of the algorithm in case of floating point operations. The main steps of the procedure:

1) Get a new N value, it is the number of the equations.
2) Generate a_1, b_1, c_1, d_1 random single float numbers [16]. Calculate coefficients A, B, C, D, E for a_1, b_1, c_1, d_1 using $(x - a_1)(x - b_1)(x - c_1)(x - d_1) = 0$ formula.
3) Checkpoint T_1
4) Calculate the root values from the coefficient values (single-precision) on CPU
5) Checkpoint T_2
6) Copy input data to the GPU
7) Checkpoint T_3
8) Calculate the root values from the coefficient values (single-precision) on GPU
9) Checkpoint T_4
10) Copy results back to the CPU
11) Checkpoint T_5
12) Enter the quartic equation coefficients directly to check the accuracy.

We repeated the whole measurement process 32 times, because the standard deviation of the runtime values (mainly in the case of small N values) was quite high. Table I and Fig. 1 show the average results of these tests.

As you can see, the CPU is faster in the case of small N values. When we increase the number of equations, the GPU becomes faster. That is what we expected, because we need to start several threads to utilize the processing power of the GPU.

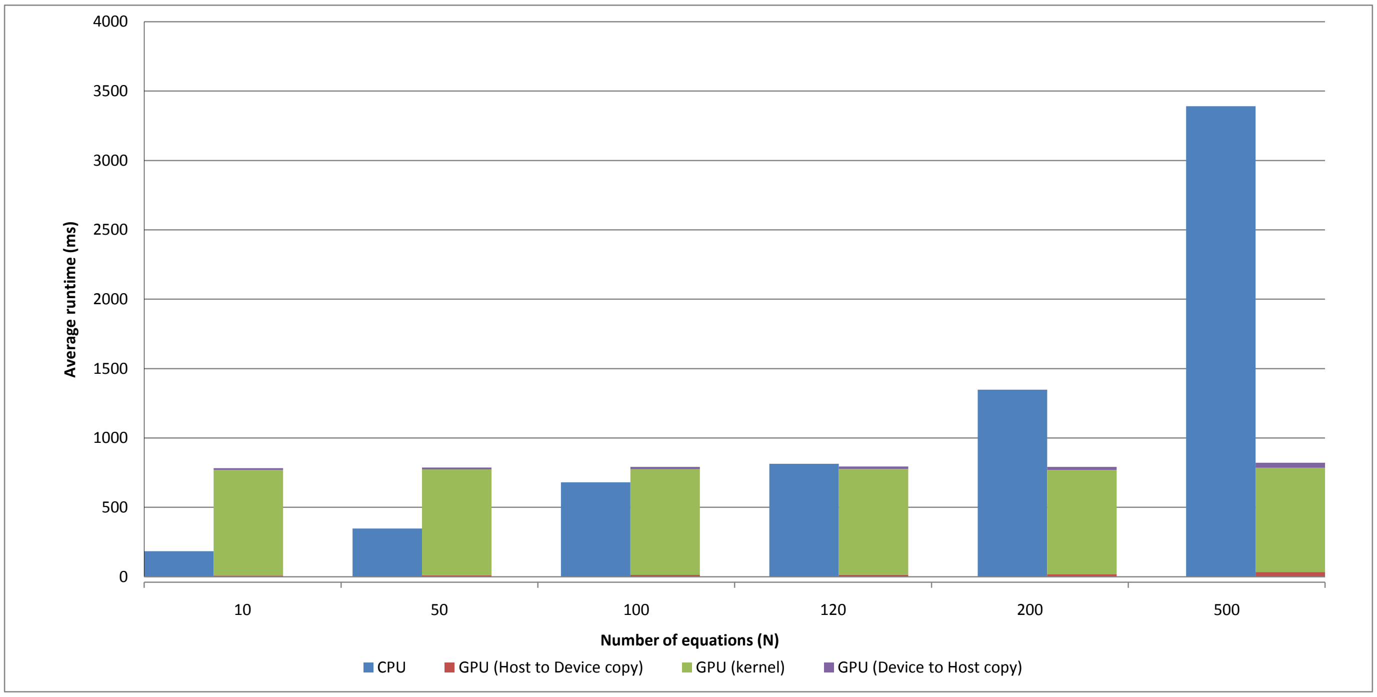

Fig. 1. Runtime of CPU and GPU algorithm in the case of 32bit variables. The stacked bars show the detailed runtime values for the GPU.

TABLE I
RUNTIME OF CPU AND GPU ALGORITHM IN THE CASE OF 64 BIT VARIABLES. THE LAST THREE COLUMNS SHOW THE DETAILED GPU RUNTIMES: MEMORY COPY FROM CPU TO GPU, KERNEL EXECUTION, MEMORY COPY FROM GPU TO CPU

			GPU details (ms)		
N	CPU (ms)	GPU (ms)	CPU-GPU	Kernel	GPU-CPU
10	184	782	9	761	12
50	348	787	11	762	14
100	681	792	14	761	17
120	814	794	15	761	18
200	1348	791	19	750	22
500	3391	822	34	751	37
1000	6767	890	58	771	61
1500	10162	955	81	787	87
2000	13565	1043	104	828	111

B. Double-precision floating points

The main steps of the test:

1) Get a new N value, it is the number of the equations.
2) Generate a_1, b_1, c_1, d_1 random double-precision numbers. Calculate coefficients A, B, C, D, E for a_1, b_1, c_1, d_1 using $(x-a_1)(x-b_1)(x-c_1)(x-d_1)=0$ formula.
3) Checkpoint T_1
4) Calculate the root values from the coefficient values (double-precision) on CPU
5) Checkpoint T_2
6) Copy input data to the GPU
7) Checkpoint T_3
8) Calculate the root values from the coefficient values (double-precision) on GPU
9) Checkpoint T_4
10) Copy results back to the CPU
11) Checkpoint T_5
12) Enter the quartic equation coefficients directly to check the accuracy.

The results are very similar as in the first test. It is obvious that both the CPU and the GPU are slower in the case of double-precision arithmetic. It is worth noting, that the double-precision speed of the GPU is very attractive. We did similar tests some years ago, and at that time, the double-precision performance of the GPU's was quite poor.

TABLE II
RUNTIME OF CPU AND GPU ALGORITHM IN THE CASE OF 64 BIT VARIABLES. THE LAST THREE COLUMNS SHOW THE DETAILED GPU RUNTIMES: MEMORY COPY FROM CPU TO GPU, KERNEL EXECUTION, MEMORY COPY FROM GPU TO CPU

			GPU details (ms)		
N	CPU (ms)	GPU (ms)	CPU-GPU	Kernel	GPU-CPU
10	241	814	10	790	14
50	993	866	15	834	17
100	1935	925	19	884	22
120	2321	938	22	892	24
200	3971	975	29	914	32
500	9804	1092	58	972	62
1000	19582	1213	105	995	113
1500	29321	1325	151	1012	162
2000	39111	1424	196	1016	212

As it is visible (Table II and Fig. 2), if the value of N is beyond a limit (about N = 50), the GPU becomes faster. The reasons are the same, we have to do enough calculations to use all the processing units of the device.

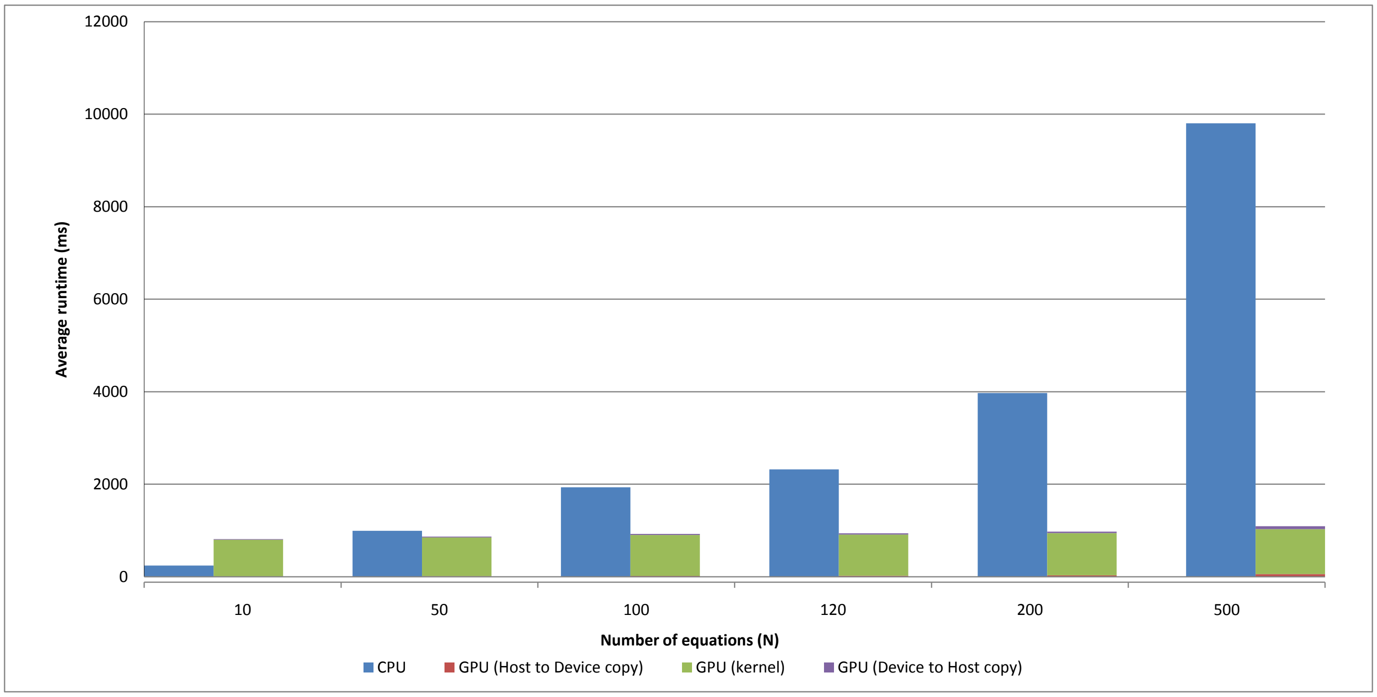

Fig. 2. Runtime of CPU and GPU algorithm in the case of 64bit variables. The stacked bars show the detailed runtime values for the GPU.

C. *Accuracy test*

In our last test, we checked the accuracy of both results. For this, we entered the results into the equation coefficients directly. Based on the equation, the result must be 0, but due to numerical instability, there may be some difference.

My experiences show that there is not any significant differences between the CPU and the GPU results. Older generations of GPUs have some limitations with floating point arithmetic, but it seems that the new architecture eliminates this problem.

V. Conclusions

It is worth porting an application to the GPU, if the number of calculation (arithmetic operations) is relatively high and the number and volume of memory access is relatively small. It is clear, that running multiple equation solvers based on the Ferrari method clearly meets these conditions.

As expected, if the number of equations is small, the CPU gives better results. But if we increase the number of parallel threads, we can reach two limits. First, the execution time of the kernel becomes smaller than the execution time of the traditional CPU code (it is when N = 115 in the single-precision case, and N = 48 in the double-precision case). However, if we check the runtime of the entire algorithm (including the memory transmissions) the CPU is already faster. When we reach the second limit (N = 120 in the case of single-precision values and N = 50 in the case of double-precision values), the GPU will be faster from every point of view. In these cases, it is definitely worth to run the solver in the GPU.

Acknowledgements

The authors would like to thank NVIDIA Corporation for providing graphics hardware for the GPU benchmarks through the CUDA Teaching Center program. The authors also wish to thank the members of the CUDA Teaching Center at Óbuda University for their constructive comments and suggestions. The content of the paper however does not necessarily express the views of these companies and persons, the authors take full responsibility for any errors or omissions.

References

[1] S. Sergyán and L. Csink, "Automatic parametrization of region finding algorithms in gray images," in *4th International Symposium on Applied Computational Intelligence and Informatics (SACI)*, Timisoara, Romania, May 17–18 2007, pp. 199–202.

[2] Z. Vámossy and T. Haidegger, "The rise of service robotics: Navigation in medical and mobile applications," in *2014 IEEE 12th International Symposium on Applied Machine Intelligence and Informatics (SAMI)*, Jan 2014, pp. 11–11.

[3] J. Tick, Z. Kovács, and F. Friedler, "Synthesis of optimal workflow structure," *Journal of Universal Computer Science*, vol. 12, no. 9, pp. 1385–1392, 2006.

[4] K. Par and O. Tosun, "Real-time traffic sign recognition with map fusion on multicore/many-core architectures," *Acta Polytechnica Hungarica*, vol. 9, no. 2, pp. 231–250, 2012.

[5] K. M. Borkowski, "Transformation of geocentric to geodetic coordinates without approximations," *Astrophysics and Space Science*, vol. 139, no. 1, pp. 1–4, 1987.

[6] P. B. Chock and E. R. Stadtman, "Superiority of interconvertible enzyme cascades in metabolic regulation: analysis of monocyclic systems." *Proc. Nati. Acad. Sci.*, vol. 74, no. 7, pp. 2766–2770, July 1977.

[7] G. Munster, "On the phase structure of twisted mass lattice QCD," *JHEP*, vol. 0409, pp. 1–10, 2004.

[8] L. Sonnenschein, "Analytical solution of ttbar dilepton equations," *Phys. Rev.*, vol. D73, pp. 1–8, 2006.

[9] S. L. Shmakov, "A universal method of solving quartic equations." *Int. J. Pure Appl. Math.*, vol. 71, no. 2, pp. 251–259, 2011.

[10] D. Stojcsics, L. Somlyai, and A. Molnar, “Unmanned aerial vehicle guidance methods for 3d aerial image reconstruction,” in *2013 IEEE 9th International Conference on Computational Cybernetics (ICCC)*, July 2013, pp. 321–326.

[11] A. Keszthelyi, “About passwords,” *Acta Polytechnica Hungarica*, vol. 10, no. 6, pp. 99–118, 2013.

[12] M. Takács and E. Tóth-Laufer, “The AHP extended fuzzy based risk management,” in *10th WSEAS International Conference on Artificial Intelligence, Knowledge Engeneering and Data Bases (AIKED)*. Cambridge: WSEAS Press, 2011, pp. 269–272.

[13] G. Windisch and M. Kozlovszky, “Improvement of texture based image segmentation algorithm for he stained tissue samples,” in *2013 IEEE 14th International Symposium on Computational Intelligence and Informatics (CINTI)*, Nov 2013, pp. 273–279.

[14] D. H. Evans, “Solving quartics and cubics for graphics,” University of Sidney, Tech. Rep., 2006.

[15] Wikipedia, “Quartic function — Wikipedia, the free encyclopedia,” 2014, [Online; accessed 11-Nov-2013]. [Online]. Available: http://en.wikipedia.org/wiki/Quartic_function

[16] G. Györök and M. Tóth, “On random numbers in practice and their generating principles and methods,” in *International Symposium on Applied Informatics and Related Areas (AIS)*, 2010, pp. 1–6.