Acta Cybernetica **23** (2017) 491–501.

A Parallel Interval Arithmetic-based Reliable Computing Method on a GPU

Zsolt Bagóczki[a] and Balázs Bánhelyi[b]

Abstract

Video cards have now outgrown their purpose of being only a simple tool for graphic display. With their high speed video memories, lots of maths units and parallelism, they can be very powerful accessories for general purpose computing tasks. Our selected platform for testing is the CUDA (Compute Unified Device Architecture), which offers us direct access to the virtual instruction set of the video card, and we are able to run our computations on dedicated computing kernels. The CUDA development kit comes with a useful toolbox and a wide range of GPU-based function libraries. In this parallel environment, we implemented a reliable method based on the Branch-and-Bound algorithm. This algorithm will give us the opportunity to use node level (also called low-level or type 1) parallelization, since we do not modify the searching trajectories; nor do we modify the dimensions of the Branch-and-Bound tree [5]. For testing, we chose the circle covering problem. We then scaled the problem up to three dimensions, and ran tests with sphere covering problems as well.

Keywords: CUDA, parallel Branch-and-Bound, circle covering

1 Introduction

Finding platforms for parallel computations is not at all easy and straightforward, but depending on the task CUDA can be a great choice. With its computational power and easy implementation, it not only lowers the runtime of our applications, but speeds up the development process itself. The platform developed by NVIDIA provides a powerful tool for GPGPU (General-Purpose computing on Graphics Processing Units).

Two advantages of CUDA are that it provides direct access to the GPU's virtual instruction set and that our processes are computed on dedicated, computing kernels. CUDA is an API, meaning that it is not a new programming language one has to learn, but rather a tool that can be paired with different widely known

[a]University of Szeged, E-mail: Bagoczki.Zsolt@stud.u-szeged.hu
[b]University of Szeged, Institute of Informatics, E-mail: banhelyi@inf.u-szeged.hu

DOI: 10.14232/actacyb.23.2.2017.4

and frequently used programming languages, and it is implementable in several well-known development environments (e.g. Matlab, MS Visual Studio). When it comes to number representation, it is important to know that while every video card supports the integer number representation, the implementation of the floating point number representation was not realized until the appearance of DirectX 9. Double precision on video cards was not present until 2008, before the GT200 series of NVIDIA and before the HD3000 and HD4000 series of AMD. In the more recent GPU architectures there is a way to implement it, and many of the recent function libraries include the double precision version of the functions, just like that in the interval arithmetic function libraries by NVIDIA that we use. For efficiency purposes it is highly advisable to use vector-based containers in our application, since the GPU itself works with vectors too, and it is capable of performing operations simultaneously on the elements of the vectors.

The GPU's architecture, however, limits the usability of the card on various types of computing, since it is specialized on executing operations on multiple data flows at the same time. This design ensures the possibility of parallelization, so the processes based on this principle are easy to parallelize. That is, in these cases, the GPU can ensure computations many times faster than the CPU could. The GPU was our primary choice, since we have to perform the same process on all of the elements of a dataflow.

Our program uses the parallel version of the Branch-and-Bound method. During the iterations of the algorithm, intervals bounding the input problem or a part of the original problem will be divided into subintervals, and this process will be executed on multiple subintervals simultaneously. The power of parallelizing will be significant when handling a large number of input circles, and when the boundaries of the circles are near the boundaries of our intervals, because in such cases the division of the input problem has to be repeated very often, hence the number of subintervals will be large. The possibilities of parallelizing the algorithm will be discussed in Section 2.3.

2 The approximating method based on Branch-and-Bound and interval arithmetic

2.1 Reliable methods

In numerical computations, most of the time it is not enough to give a rounded up or rounded down result for our problems since we often need the most accurate solution possible, within a minimal error range, if possible, with a very accurate error estimation that can be used in further calculations. Lots of significant mathematical proofs require this kind of reliability, but this requirement can be extended to more realistic problems, such as different economic problems, where a miscalculation, or an incorrectly calculated error range can result in the loss of a great deal of money.

To solve these kinds of tasks, we will utilize so-called reliable numerical methods. To understand what we mean by a method being reliable, we need to be aware of the cause of these expectations, namely the error. It may originate from different problems, which might for instance be the inaccuracy of the input data, the inaccuracy of the formulas used; or, in our case, the inaccuracy of the numeric representation we use. These errors may accumulate throughout the whole computation process, and significantly distort the results. It is easy to see that because of this distortion, when it is necessary to have exact results, the results must be handled differently.

In short, we can call a method reliable if it gives us the upper and lower bounds of our results, with a guaranteed error estimation at the end of the process.

2.2 Naive interval arithmetic, and its usage

In numerical analysis due to the finite precision of computers and estimations, we can lose significant digits and this will leave us with inexact results and possibly huge errors. Because of this, it is reasonable to demand that a method should be able to give interval bounds of our computational results instead of a rounded up or rounded down result. It provides both a lower and upper bound to the results, hence instead of exact values, we have to compute with intervals. Whenever we encounter a source of error like rounding, our intervals will keep expanding, but the reliable method will give us an estimation of the error that we can include in our future computations in order to get increasingly accurate results [2, 1].

On the video cards, to be able to utilize the GPU in our computations as much as possible, we will have to define both intervals, and interval arithmetic operations directly. For this, we will use the `cuda_interval_rounded_arith.h` and `cuda_interval_lib.h` function libraries provided by NVIDIA, in which we find all the required interval operations [6].

However, using one-dimensional intervals on their own will not be sufficient for our two-dimensional test cases. Implementing these is not a hard task with the mentioned function libraries, and the way to create multidimensional intervals is fairly obvious.

The shapes used will be represented using two-dimensional intervals. The unit square to cover will be stored in a two-dimensional interval, namely $[[x1, y1], [x2, y2]]$, where $x = (0, 1)$ and $y = (0, 1)$. With these two points, the unit square can be extended in the plane. For the circles, a class is used to store the data structure. The origin of the circles is stored in one, two-dimensional interval with an (x, y) coordinate pair in the plane, where both x and y are intervals. Then we store the square of the length of the radius as a simple interval (r^2). For an illustration of this, see Figure 1.

The most significant operation that had to be implemented using intervals is the checking method itself, which is used to determine the covering of a subproblem – a slice of the unit square – with a given circle. We check if the distance of the points in the square from the origin of the circle are smaller than the radius of the circle itself. In the Euclidean plane, the distance between two points can be found

via the formula

$$distance(p_1, p_2) = \sqrt{((p_{2,1} - p_{1,1})^2 + (p_{2,2} - p_{1,2})^2)}, \tag{1}$$

where $p_1 = (p_{1,1}, p_{1,2})$ and $p_2 = (p_{2,1}, p_{2,2})$ are two-dimensional points in the Euclidean plane. It is easy to see that it is not efficient to take the root of the sum for each iteration, so the square of the radius of the circles will be stored in advance, and only the square of the distance will be computed without taking the square root each time. All we have to do now is to compare the squares of the two values with the following formula:

$$sup(distance^2) < inf(r^2), \tag{2}$$

where sup is the supremum value of the interval and inf is the infimum value.

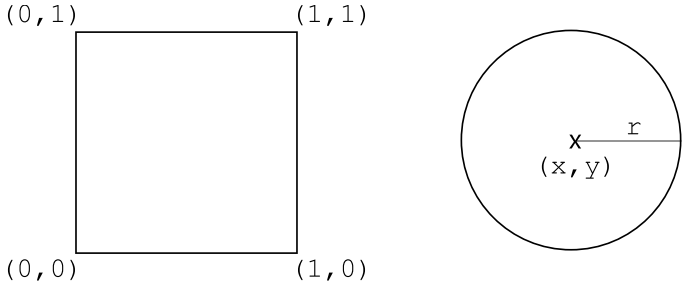

Figure 1: Two-dimensional shapes

2.3 The Branch-and-Bound algorithm

The implementation of our reliable method is based on a simplified Branch-and-Bound algorithm. The algorithm is well known and it is used widely in mathematical proofs, mathematical or computational optimization and in solving linear programming tasks.

The technique consists of two major substeps, namely the branching and the bounding. In the branching process, we divide our problem into two (dichotome branching) or more (polytome branching) subproblems, building an enumeration tree. Every division creates a new branch in this tree. The bounding process will give us the lower bound of the feasible solutions in the set of the solutions.

A simplified version of the algorithm is used with the goal to reliably prove any given quality of a two dimensional interval. A checking method is used to help in deciding whether the given quality is met. We discussed the checking method in Section 2.2 above. To improve the runtime, and to avoid infinite loops due to the infinite bisection of the subproblems when we bisect precisely on an interval boundary, we give a constraint for the length of the new subintervals. Let us call this constraint ϵ, the limit of the error. It should be mentioned that we cannot reliably

decide whether the quality is not actually met. This problem originates from the interval arithmetic we use; namely, there might be cases where the subintervals created by our bisections are already smaller than the given error limit ϵ, so a reliable proof only can be given if the given quality is actually met; otherwise it can not be proven whether the problem did not meet the quality or if in the bisection process, the result is already smaller than the given constraint.

The Branch-and-Bound algorithm is an easily parallelizable method [4]. We use node level parallelization, which means that we evaluate the nodes (which are the subproblems) simultaneously, and after all the evaluations are complete, we compare the results and continue the process based on the results [9]. This kind of parallelization is called low-level, or type 1 parallelization, since we will not modify the search trajectories or the dimensions of the Branch-and-Bound tree [7]. For this purpose, we shall use the computing cores of the CUDA supporting NVIDIA video cards, since they contain significantly more cores than a standard CPU does.

Algorithm 1 A parallel B&B algorithm.

Funct PIBB($param, \epsilon$)

1: Set the working buffer $\mathcal{B}_W := \{param\}$ and the result set $\mathcal{L} := \{\}$
2: Calculate current thread ID: tid
3: **while** $\mathcal{B}_W$ is not empty **do**
4: Pop v from $\mathcal{B}_W$ and split along all sides, and take the subproblem which belongs to our current thread: v_{tid}
5: **if** $maxWidth(v_{tid}) < \epsilon$ **then** *Termination rule*
6: $\mathcal{L}^{tid} = false$
7: **else if** $isMetQuality(v_{tid})$ is false **then**
8: Push v_{tid} into $\mathcal{B}_W$
9: **break**
10: **end if**
11: Synchronize threads
12: **if** $tid = 0$ **then**
13: **for** $i = 1, \ldots, sizeOf(\mathcal{L})$ **do**
14: **if** $\mathcal{L}^i$ = false **then**
15: **print** Cannot decide whether the quality is not met or the result is smaller than ϵ
16: **return** false
17: **end if**
18: **end for**
19: **end if**
20: **end while**
21: **print** The quality is met
22: **return** true

The correctness of the single core, non-parallel version of the algorithm has already been proven (see [3]).

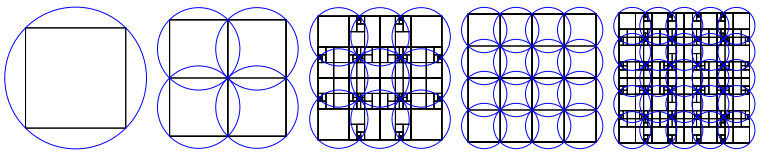

Figure 2: Test cases of circle covering

3 Test cases

3.1 Circle covering

The problem of circle covering [8] is the dual of the circle packing problem [12], where our goal is to give the densest covering of different shapes with n congruent circles. There are numerous proofs for both locally and globally optimal circle covering cases (see [11]).

For now, let us put aside the optimization problem for the covering cases. Our choice of testing was to place $n \times n$ circles on the Euclidean plane in such a way that they will certainly cover the unit square. The most obvious idea for this is to divide the diagonal of the square into n pieces, so the radius of a circle will be $r = \sqrt{\frac{2}{n}}$. In order to ensure the reliability of our computation, we need to add an error to the radius, which has to be bigger than our constraint ($\epsilon_r > \epsilon$). The reason is that near the tangential points of the circles, the distance should be 0 – which is smaller than our error range, causing the program to return with an undecidable result. During the testing phase, we used the values $\epsilon = 10^{-5}$ and $\epsilon_r = 10^{-4}$.

As we mentioned earlier, the method we use is a simple Branch-and-Bound based algorithm. The behaviour of it on the circle covering case is demonstrated by the first five test cases shown in Figure 2. We divide the sides of an interval into subintervals, and we check whether the given interval-piece is covered; then we continuously divide and check as explained above, until the result is positive or undecidable. The intervals will be divided up until the length of the intervals attain the length of the ϵ constraint, and below that point the method returns with an undecidable result.

When the number of the input circles is to the power of 2, the covering is unequivocal, meaning that it is decidable with a minimal amount of subproblems - in this case subboxes. For example, the $n = 1$, $n = 2$ and the $n = 4$ cases shown in Figure 2 illustrate the clarity and simplicity of the decision. The reason is that in these cases the dividing point of the intervals will be exactly on the intersections of the circles. Within a few divisions, we will reach a state where all the subproblems are the largest squares inside the circles, hence the circles will obviously cover the squares.

3.2 Expanding the circle covering problem to sphere covering

While solving circle covering problems in the Euclidean plane has a certain mathematical attraction, the world around us is three dimensional, hence we found it more useful to extend the problem to higher dimensions.

Firstly, we will have to rescale the input data. To store the spheres, we will use a class similar to the one we used to store circles, with a small modification to store the third dimension coordinates as well. The figure we wish to cover in this case is the unit cube, which is defined as the following: $[[x_1, y_1], [x_2, y_2], [x_3, y_3]]$, where $[x_i, y_i] = [0, 1]$, if $i = 1, 2, 3$. For data storage, we can use the stack buffer we have implemented earlier, which is accessible for both the CPU and GPU.

The n^3 spheres are placed along the space diagonal of the cube, and along lines parallel to this, which will ensure that they are tangential to each other. The radius in this case can be calculated from the 3D-space diagonal.

When rescaling the checking method, the distance between two points in a three-dimensional space is found using the usual distance metric in three dimensions. For a (p, q) pair of points, where $p = (p_1, p_2, p_3)$, and $q = (q_1, q_2, q_3)$, the following formula applies:

$$distance(p, q) = \sqrt{((q_1 - p_1)^2 + (q_2 - p_2)^2 + (q_3 - p_3)^2)}. \tag{3}$$

The covering is true when the distance between the points of an interval from the centre of the given sphere are less than the radius of the sphere (for which we store the squared value in the program). In this case, the subintervals will be cubes.

3.3 The test results

We ran the test cases first on 1, then on 2 CPU cores [10] as a benchmark for the GPU runtime testing, which was conducted on 256 CUDA cores. The runtime dependency on the number of circles can be clearly seen after a few cases, so we ended up running 20 test cases on all three of the core numbers.

We conducted the tests on the following hardware: AsRock Anniversary H97 motherboard, Intel Pentium G3450 CPU, 2*4GB DDR3 Kingston KVR16N11S8/4 memory, ASUS STRIX GTX950 2GB GPU and the Windows 10 Pro operating system.

The results of the runtime tests of the two-dimensional have been plotted in Figure 3. The results we got completely matched our expectations. By increasing the number of the circles, the runtimes also increased, and when we use more and more cores for the computations, the runtimes display an inversely proportional relation.

Our expectations for the three-dimensional problems were very similar to our expectations for the two dimensional one. The reason we still found it advisable to run the test cases in three dimensions is because of the increasing difference between the CPU and GPU runtimes that will provide an even better way of comparing them.

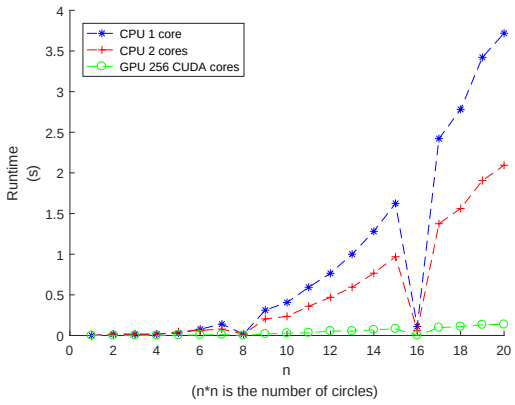

Figure 3: The runtime of the program on various test cases, with a different number of cores

The results of the tests can be seen in Figure 4. The values of the CPU and GPU runtimes do not display any irregularities; in fact they are just as we expected them to be; and the characteristics are almost identical to those we observed in the two dimensional test cases. The interesting part is when we compare the two results. Thanks to the parallelization on the GPU, the runtime of our program is never more than a few seconds, even when the process on the CPU terminates with a runtime of over an hour. This demonstrates the runtime differences much better than the few-minute differences measured in the two-dimensional testcases. Table 1 gives the actual measured runtime values in seconds.

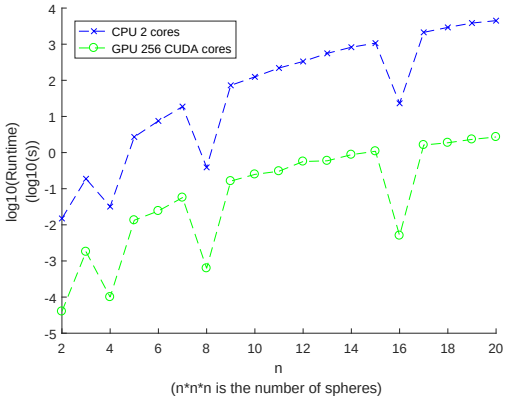

Figure 4: Test cases for sphere covering

As can be seen in both Figure 3 and Figure 4, there are certain exceptional cases. We mentioned previously that when the input number of circles or spheres is to the power of 2, the number of subboxes needed to be generated is minimal, which is the same as the number of the input shapes. In other words, the number of interval divisions made by the algorithm is low, therefore it should terminate rapidly and lead to significantly lower runtimes compared to those cases where the number of the input shapes was not to the power of 2.

Table 1: The results for the circle and the sphere covering test cases with different numbers of circles. The results are given in seconds.

n	Circle			Sphere	
	CPU (s)		GPU (s)	CPU (s)	GPU (s)
	1 core	2 cores	256 cores	2 cores	256 cores
1	0.000	0.000	0.000034	0.000	0.000035
2	0.016	0.016	0.000036	0.015	0.00004
3	0.015	0.015	0.001071	0.188	0.0018
4	0.016	0.016	0.000049	0.032	0.0001
5	0.031	0.046	0.004135	2.703	0.0134
6	0.078	0.063	0.005870	7.516	0.0242
7	0.140	0.078	0.010647	18.750	0.05675
8	0.016	0.016	0.000098	0.390	0.000627
9	0.312	0.204	0.019143	72.609	0.1624
10	0.406	0.235	0.028382	124.656	0.2489
11	0.593	0.359	0.033319	219.141	0.3049
12	0.765	0.469	0.054516	332.390	0.5652
13	1.000	0.594	0.053796	557.468	0.5919
14	1.281	0.766	0.069647	824.157	0.8735
15	1.625	0.969	0.081841	1069.800	1.0883
16	0.110	0.063	0.000297	23.094	0.00503
17	2.422	1.375	0.097214	2127.890	1.6199
18	2.782	1.562	0.109281	2915.480	1.8684
19	3.422	1.906	0.132101	3848.860	2.3156
20	3.719	2.094	0.136277	4481.250	2.7

4 Conclusions

During the testing of our reliable method we had a chance to try it out in a parallel, graphical computational environment. The test proved most informative, and the results once again met our expectations. It was a good illustration of where computing can not only be done fast and efficiently with supercomputers, but the average PC user can also have the opportunity and tools to experiment even if they are on a tight budget.

When handling processes or computations that are straightforward to parallelize, either at a tree or node level, it is recommended to exploit the power of the GPU, and if one does, one of the best options is CUDA. It provides a friendly, easily operable platform and supports most of the well-known programming languages like Java, C/C++, Python if not directly, then with the help of third party applications. The toolbox offers numerous process monitoring, examining opportunities, and ready made function libraries that contain algorithms and methods specially designed for GPU usage. Functions that are essential or indispensable in the daily work routine of a programmer or a mathematician are available.

A big advantage of CUDA and computing on GPUs overall is the accessibility of the video cards. Entry-level video cards are already able to solve parallel computing tasks and produce satisfactory results, and the 256 CUDA cores we used is roughly equal to the computational power of these cards. Cards supporting the latest version of CUDA are available in the price range of 100€, which makes them affordable even for regular PC users.

With the help of the readily scalable test cases, we got a good idea of the computing power of different architectures and their limitations. Now we can say from experience that when using approximation methods, a resource for computations like a video card is a very efficient tool, especially because it removes a big burden from our processor.

If we can reliably determine the properties for a circle, sphere, and hypersphere covering problem, then the next step is obvious: developing an algorithm which is able to give the optimal solution for the covering problems. In the future, we would like to develop a global optimization method based on our GPU-powered parallel method using interval arithmetic.

References

[1] Alefeld, G. and Herzberger, J. *Introduction to interval computations*. Academic Press Inc, New York, 1983.

[2] Alefeld, G. and Mayer, G. Interval analysis: theory and applications. *J. Comput Appl Math*, 121:421–464, 2000.

[3] Bánhelyi, Balázs, Csendes, Tibor, and Garay, Barna. A verified optimization technique to locate chaotic regions of Hénon systems. *Journal of Global Optimization*, 35:145–160, 2006.

[4] Casado, L.G., Martinez, J.A., Garcia, I., and Hendrix, E.M.T. Branch-and-bound interval global optimization on shared memory multiprocessors. *Optimization Methods Software*, 23:689–701, 2008.

[5] Clausen, Jens. Branch and bound algorithms - principles and examples, 1999.

[6] Collange, S., Daumas, M., and Defour, D. Interval arithmetic in CUDA. In *GPU Computing Gems Jade Edition*. 2012.

[7] Crainic, Teodor Gabriel, Cun, Bertrand Le, and Roucairol, Catherine. Chapter 1. parallel branch-and-bound algorithms, parallel combinatorial optimization, 2006.

[8] Friedman, E. Circles covering squares. `http://www2.stetson.edu/~efriedma/circovsqu/`, 2005.

[9] Garzón, E.M. and Garca, I. Parallel implementation for large and sparse eigenproblems. *Acta Cybernetica*, 15:137–149, 2001.

[10] Palatinus, E. and Bánhelyi, B. Circle covering and its applications for telecommunication networks. In *Proceedings of the 6th International Conference on Applied Informatics (ICAI2010)*. 2011.

[11] Sanders, Jason and Kandrot, Edward. *CUDA by Example: An Introduction to General-Purpose GPU Programming*. Addison-Wesley, United States, Ann Arbor, Michigan, 2010.

[12] Szabó, P.G., Markót, M.Cs., Csendes, T., Specht, E., Casado, L.G., and García, I. *New Approaches to Circle Packing in a Square – With Program Codes*. Springer, Berlin, 2007.