A GPU approach to distance geometry in 1D: an implementation in C/CUDA

Simon Hengeveld, Antonio Mucherino

A GPU approach to distance geometry in 1D: an implementation in C/CUDA

Simon B. Hengeveld* A. Mucherino,*
*IRISA, University of Rennes 1, Rennes, France.
simon.hengeveld@irisa.fr, antonio.mucherino@irisa.fr

***Abstract*—We present a GPU implementation in C and CUDA of a matrix-by-vector procedure that is particularly tailored to a special class of distance geometry problems in dimension 1, which we name "paradoxical DGP instances". This matrix-by-vector reformulation was proposed in previous studies on an optical processor specialized for this kind of computations. Our computational experiments show that a consistent speed-up is observed when comparing our GPU implementation against a standard algorithm for distance geometry, called the Branch-and-Prune algorithm. These results confirm that a suitable implementation of the matrix-by-vector procedure in the context of optic computing is very promising. We also remark, however, that the total number of detected solutions grows with the instance size in our implementations, which appears to be an important limitation to the effective implementation of the optical processor.**

THIS IS AN EXTENDED VERSION

This is an extended version of the conference paper published on IEEE proceedings, where we added some code snippets. This was requested by the three colleagues that have reviewed the paper, but unfortunately we could not include any additional material in the original publication for lack of space.

I. INTRODUCTION

The Distance Geometry Problem (DGP) asks whether a simple weighted undirected graph $G = (V, E, d)$ can be realized in the Euclidean space $\mathbb{R}^K$, with $K > 0$, so that the distance constraints

$$\forall \{u, v\} \in E, \quad ||x_u - x_v|| = d_{u,v},$$

are satisfied [5]. When this is the case, we say that the mapping $x : v \in V \to x_v \in \mathbb{R}^K$ is a *valid* realization of the graph G. Depending on the DGP application at hand, the vertices $v \in V$ can represent different kinds of *objects*, for which possible positions in $\mathbb{R}^K$ are searched. The edge set E encodes the information about the distance between vertex pairs, and the numerical value of these distances is given by the associated weight. Notice that $||\cdot||$ is the Euclidean norm. We suppose that the available distance values are *exact*, i.e. extremely precise.

In this work, we focus our attention of DGPs in dimension 1. In 1979, Saxe proved that the DGP is NP-complete when K is set to 1 [10]. The main DGP application in this dimension is the clock synchronization problem: given a set of clocks (represented by the vertices $v \in V$), and a subset of offset measurements between pairs of clocks (encoded by the edges $\{u, v\} \in E$ and the associated weight $d_{u,v}$), the problem asks whether it is possible to know the precise time indicated by all clocks [11]. This problem is fundamental for the synchronization of events in distributed systems [1], [12], as for example in wireless sensor networks [14].

The Branch-and-Prune (BP) algorithm was proposed in [4] for a subclass of DGP instances that admit the discretization of their search space. In the 1-dimensional case, this algorithm can be employed under the much weaker assumption that the graph G is connected [9]. In this case, in fact, a vertex order on V, which ensures that every vertex v has at least one predecessor u (exception made for the first vertex in the order), can be easily constructed [8]. This vertex order is indicated by the subscripts associated to the vertices in the discussion below.

We are interested in a particular subclass of DGPs in dimension 1: the class of *paradoxical* DGP instances [2]. These instances are represented by graphs G that are cycle graphs, for which a vertex order on its vertex set can be trivially identified. The paradoxical character of these instances is given by the two following observations. On the one hand, the construction of solutions to these instances appears to be relatively *easy*, because most vertices $v_k \in V$ only depend on one predecessor in the vertex order, so that, for each x_{k-1}, the two new positions $x_{k-1} - d_{k-1,k}$ and $x_{k-1} + d_{k-1,k}$ can be easily computed for v_k. Notice that this procedure allows us to build up a binary tree collecting, on each of its layers, the possible positions for each vertex v_k. On the other hand, however, the absence of any other distance information (apart the distance to the predecessor $d_{k-1,k}$) up to the layer n, where $n = |V|$, makes this class of instances actually *very hard*. In fact, it is only at layer n that the distance between the first vertex v_1 and the last vertex v_n can be exploited to select the only two valid realizations out of a set of 2^{n-1} potential solutions [6].

In this work, we consider the matrix-by-vector reformulation of paradoxical DGPs in dimension 1 (recently proposed in [2] for solving paradoxical DGP instances on a new optical processor) and we implement it on a GPU device. The presented computational experiments show a consistent speed-up when our GPU implementation is compared against the BP algorithm, as well as when the comparison is performed against the sequential implementation of the matrix-by-vector procedure itself. These results confirm therefore that

the matrix-by-vector reformulation is promising in the context of optic computing. Our experiments also point out, however, a possible limit in the actual implementation of the optical processor.

The rest of this paper is organized as follows. In Section II, we will describe the matrix-by-vector reformulation of our paradoxical DGP instances in dimension 1. In Section III, we will present our GPU implementation for the matrix-by-vector multiplication, which will benefit of some simplifications implied by the particular problem we aim to solve. We will present and discuss our computational experiments in Section IV (this section also includes some code snippets in this extended version of the paper), and finally draw our conclusions in Section V.

II. A MATRIX-BY-VECTOR REFORMULATION

When the BP algorithm mentioned in the Introduction is employed for the solution of paradoxical DGP instances in dimension 1 [9], a binary tree containing all possible vertex positions can be recursively constructed, and the valid realizations can be selected at the very end when positions are computed for the last vertex $v_n \in V$. Our paradoxical instances have the particularity of solely executing the branching phase of the algorithm until a leaf node of the tree is reached; it is only at this point that the pruning mechanism is invoked, where the only distance not used for branching, the distance related to the edge $\{1, n\}$, is verified. If the distance is satisfied by the current position for v_n, then the path from the tree root to the current node is a valid realization; it can be discarded otherwise.

As remarked in [2], it is possible to replace, for our paradoxical instances, the pruning phase occurring only at layer n with an additional branching phase, which is performed over the fictive vertex v_{n+1} that is introduced in the original graph. The fictive vertex is connected to its predecessor v_n by an edge having the same weight as the original edge $\{1, n\}$. The edge from v_1 to v_n is thereafter removed, breaking in this way the original cycle structure. The main reason for making this manipulation on G is that now the distance information is equally distributed over the vertices of the graph, and the solver of paradoxical instances can perform exactly the same operation when stepping from one vertex to its successor. In order to identify the valid realizations, it is finally necessary to verify that $x_1 = x_{n+1}$.

The introduction of the fictive vertex allows us to reformulate the paradoxical DGP in dimension 1 as a matrix-by-vector multiplication [2]. We introduce the matrix

$$M_{ij} = \begin{cases} -1 & \text{if } (i-1)/2^{j-1} \bmod 2 = 0, \\ 1 & \text{otherwise,} \end{cases}$$

and the vector $y_j = d_{j,j+1}$, which contains the distance information related to our paradoxical instance. Thus, the vector $r = My$ contains all possible positions x_{n+1} for all possible solutions. Notice that the index i varies from 1 to 2^n, whereas the index j varies from 1 to n. The feasible solutions

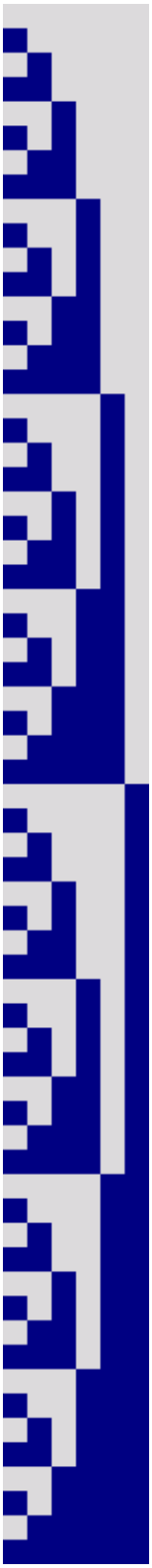

Fig. 1. The pattern given by the signs of the elements of the matrix M. In dark blue, the elements that have positive sign; in light gray the ones having negative sign.

```
__global__ void gpusumrows(int n,long two_n,float *d,long *sol)
{
    int j;
    int id,nthreads;
    long ilstart,ilend;
    long il,chunk;
    float sum;

    // getting the number of involved threads
    nthreads = blockDim.x * gridDim.x;

    // getting the unique id for this thread (1-dim grid)
    id = blockIdx.x * blockDim.x + threadIdx.x;

    // computing the portion of optic matrix rows to treat
    chunk = (two_n / nthreads) + 1L;
    ilstart = id*chunk;
    ilend = ilstart + chunk;

    // iterating over the assigned long integers
    sol[id] = 0L;
    il = ilstart;
    while (il < ilend)
    {
        // first element
        sum = (-1 + ((il & 1L) << 1))*d[0];

        // second element
        sum = sum + (-1 + (il & 2L))*d[1];

        // other elements
        for (j = 2; j < n; j++)  sum = sum + (-1 + ((il >> (j - 1)) & 2L))*d[j];

        // verifying feasibility
        if (sum < 0.0f)  sum = -sum;
        if (sum < gpu_eps)  sol[id] = il;

        // preparing for the next row
        il++;
    };
};
```

Fig. 2. The kernel in CUDA.

to our paradoxical instances are the ones for which $r_i = 0$ (because x_1 is here implicitly set to 0).

We notice that performing the matrix-by-vector multiplication gives an answer to the original decision problem (does it exist a realization such that...) but it does not directly provide *the* realizations, i.e. the sequences of positions on the real line for the vertices of G. In order to construct one selected valid realization, as for example the realization encoded by the i^{th} row of the matrix M, the value of each position x_k^i for the vertex $v_k \in V$ (we added a superscript to x to specify the matrix row) can be obtained by performing the following partial sum:

$$x_k^i = \sum_{j=1}^{k} M_{ij} y_j.$$

The next section describes an ad-hoc GPU implementation of this matrix-by-vector multiplication.

III. A GPU IMPLEMENTATION

Our GPU implementation does not perform *generic* matrix-by-vector multiplications. For this general problem, the reader can refer to some recent (see for example [7], [13]) and very recent (see [3]) publications on the topic. Differently from the cited papers, our implementation takes advantage of the structure of our matrix M to optimize the computations.

First of all, since the elements of our matrix M are either -1 or 1, we can trivially "move" all distance values from the vector y to the matrix, by paying only attention to the sign to consider for each distance value when placed in a particular row of the matrix. We define therefore this new matrix:

$$M'_{ij} = \begin{cases} -d_{j,j+1} & \text{if } (i-1)/2^{j-1} \bmod 2 = 0, \\ d_{j,j+1} & \text{otherwise,} \end{cases}$$

from which the vector r can be simply computed by summing

```
// fragment of the main

cudaMemcpy(gpu_d,d,n*sizeof(float),cudaMemcpyHostToDevice);  // copying distance values on GPU
t0 = clock();  // resetting the wall clock
gpusumrows<<<nblocks,threads_per_block>>>(n,nrows,gpu_d,gpu_sol);  // launching kernel
cudaThreadSynchronize();  // synchronizing threads belonging to different blocks
t1 = clock();  // measuring the time
sec = (double)(t1 - t0) / CLOCKS_PER_SEC;
cudaMemcpy(sol,gpu_sol,nthreads*sizeof(long),cudaMemcpyDeviceToHost);

// ...
```

Fig. 3. A fragment of the **main**.

up all row elements:

$$r_i = \sum_{j=1}^{n} M'_{ij}.$$

As a consequence, our GPU implementation will only perform sums, and not products of real numbers.

Another important point in our implementation is the procedure to construct the matrix M', and in particular for the choice of the sign for each matrix element. The rule given in the definition (involving the modulus operator) is simple to understand and to apply, but it can be computationally very expensive to perform for every element of the matrix. For our implementation, we found another, and more efficient, method to identify the sign of every matrix element.

Fig. 1 shows the sign distribution over the matrices M and M': all positive elements correspond to the dark pixels, while all negative elements correspond to the light gray pixels. More than one pattern can be identified in these matrices, but one in particular turns out to be very useful for our GPU implementation. If in fact we interpret every gray pixel as a 0 (instead of a -1), whereas the dark pixels still represent 1's, then we can see every matrix row (notice that the matrix is transposed in Fig. 1) as the binary representation of integer numbers spanning from 0 to $2^n - 1$. Moreover, if we consider the big-endian convention for the bit ordering (which is, the less significant bit is on the left side, differently from our standard convention with decimal numbers), then the integer number at row i is simply the predecessor (in integer arithmetic) of the one at row $i+1$, and it is the successor of the one at row $i-1$. If the bits of an integer ℓ encode therefore the signs at row i, the bits of the integer $\ell+1$ simply encode the signs at row $i+1$.

In our GPU implementation, every thread is in charge of computing the sums for a subset of matrix rows. This subset forms a block of contiguous matrix rows, so that, once each thread has found out its starting value for ℓ, it simply needs to increase it by one unit per time for treating all subsequent rows. Naturally, all row blocks are supposed to have the same size in order to better exploit the power of the GPU device.

After the computation of every row sum, the thread verifies whether this sum is *close enough* to 0. In the case it is true, the thread keeps this information aside (in binary format) and it sends it back to the CPU at the end of the computations. Notice that this information is binary (a valid realization was found or not), because the symmetry properties [6] of our paradoxical instances indicate that the only chance to have two valid realizations treated by the same thread is when all matrix rows are assigned to one unique thread.

	BP algorithm		CPU matrix-by-vector		GPU matrix-by-vector	
$\|V\|$	#sols	time	#sols	time	#sols	time
20	3	0.012189	3	0.022815	3	0.000437
21	4	0.021719	4	0.048202	4	0.000849
22	8	0.036419	8	0.099945	8	0.001699
23	16	0.067282	16	0.208566	16	0.003494
24	44	0.133229	44	0.435459	44	0.007211
25	82	0.260027	82	0.905198	82	0.014951
26	130	0.498371	130	1.886785	130	0.030999
27	271	0.989594	271	3.905493	271	0.064336
28	515	2.025879	515	8.110146	513	0.133360
29	1074	4.186474	1074	16.831842	1074	0.263456
30	2134	8.036184	2134	34.801628	2046	0.509210
31	3638	15.836381	3638	71.677937	3358	1.006642
32	7613	34.954935	7613	147.561225	6547	2.032326

TABLE I
THE COMPUTATIONAL EXPERIMENTS COMPARING THE STANDARD BP ALGORITHM AGAINST THE SEQUENTIAL AND THE PARALLEL IMPLEMENTATIONS OF OUR MATRIX-BY-VECTOR PROCEDURE. COMPUTATIONAL TIMES ARE GIVEN IN SECONDS. ALL USED INSTANCES WERE RANDOMLY GENERATED AND THEY BELONG TO THE CLASS OF PARADOXICAL INSTANCES. SIMILAR RESULTS CAN BE OBTAINED WITH LARGER INSTANCES.

IV. COMPUTATIONAL EXPERIMENTS

This section presents some computational experiments where we compare the standard BP algorithm (see Introduction) against our matrix-by-vector procedure, executed both in sequential and in parallel. In this extended version of our contribution, we initially present the two main parts of our C/CUDA implementation.

Fig. 2 shows the kernel in CUDA to be executed by all threads on the GPU device. It basically performs the following steps. First of all, it computes the unique identifier of the running thread, which is thereafter used for determining the chunk of matrix rows the thread is supposed to work on. Then, the main part of the kernel consists in a **while** loop (over all matrix rows assigned to this thread) where the sums of all row elements are computed. Notice the use of the bits in the integer **il** to choose the sign for every available distance. The

feasibility check is performed at the end of every iteration of the **while** loop, and in case of a positive answer, the information is stored in the element of the array **sol** devoted to this thread. A fragment of the **main** function is shown in Fig. 3, with the CUDA call to the kernel.

The experiments presented below were performed on a workstation equipped with an Intel(R) Xeon(R) CPU E5-2609 v3 @1.90GHz, Nvidia GPU GeForce GTX TITAN X graphics card, and running Ubuntu Linux operating system. We compiled our programs with the version 5.4.0 of GCC, and with the version 9.0.176 of CUDA. In all experiments, our execution on GPU was launched with a thread grid comprising 64 blocks, having 512 threads each.

Table I presents some computational experiments where the BP algorithm is compared against the sequential and the parallel implementations of our matrix-by-vector procedure. We considered instances of size ranging from 20 to 32 which were randomly generated so that to satisfy the properties of paradoxical instances. The cardinality $|V|$ of the vertex sets is reported on the first column of the table. We omit to report the cardinality of the edge set E because it always corresponds to $|V|$ in our instances. The BP algorithm was run only on CPU; the matrix-by-vector procedure was run on both CPU (the sequential version) and GPU (the parallel version).

While it is expected (see Introduction and Section III) that every instance only admits two valid realizations, the second column of Table I shows that the number of solutions (#sols) found by the BP algorithm is larger, and it tends to increase with the instance size. Our interpretation for this phenomenon is that, the more the search space increases in size (exponentially with n), the more are the chances to find a realization that is *close enough* to feasibility. The verification of the final distance $d_{1,n}$ is performed with tolerance $\varepsilon = 10^{-4}$ in all experiments, which allows us to take into consideration the possible round-off error propagation. However, the use of this tolerance seems to enlarge *too much* the subset of realizations for which this final distance appears to be satisfied. We remark, however, that a generic tuning on the value of ε that would work for all instances is naturally not possible.

The third column of our table gives the time in seconds necessary for the BP algorithm to fully explore the search space of the given instances (see Fig. 3). Recall that the search space has size 2^{n-1}.

The forth and fifth columns of Table I show the results obtained with the implementation of our matrix-by-vector procedure in sequential. While the number of found solutions does not change w.r.t those found by BP, we remark an increase on the computational time. This was expected, because the matrix-by-vector formulation does not exploit the fact that the computations necessary for a given solution can be partially reused for neighboring solutions (i.e. belonging to near matrix rows). This is the reason why the BP algorithm works differently. However, the independence of the matrix rows is an essential feature for our GPU implementation.

Finally, the last two columns of the table show the results obtained with our GPU implementation. We point out that we used the **float** primitive type for the distances and the positions (we did the same in the previous two C implementations, in order to obtain results as uniform as possible). We used **long** types to store the values of the integer ℓ (see Section III). Naturally, this choice limits the instance size n to 64, but has no impact on the experiments we have performed for this work (the decimal representation of 2^{64} is already a quite "large" integer, composed by 20 digits).

The computational time is significantly reduced, when compared to the sequential version of the matrix-by-vector procedure, as well as when the comparison is performed against the original BP algorithm. We notice, however, that the number of found solutions differs with the other implementations for the instances of larger size. This error is due to the way the GPU threads communicate their results to the CPU: this information is in fact binary (solution found / not found) because at most one solution per thread was initially expected to be identified. Apparently, during the computations, more than one solution was instead (wrongly) detected by the same thread, thus leading to a smaller final solution sum.

The reader may wonder why we have decided not to fix this "issue" in our CUDA implementation. Since our work is motivated by the optical processor mentioned in the Introduction, which is supposed to perform the calculations (although in an analog fashion) in a similar way, it seems important to us to point out this drawback of the implementation. This is actually a quite important limitation for the physical implementation of optical processor.

V. Conclusions

We have presented a GPU implementation of a matrix-by-vector procedure that is particularly tailored for the solution of paradoxical DGP instances in dimension 1. The idea to reformulate this problem as a matrix-by-vector multiplication comes from previous studies on an optical processor, which is specialized for this class of problems.

Our computational experiments show that our GPU implementation is already able to take advantage of the matrix-by-vector reformulation. On our randomly generated paradoxical instances, the pattern shown by our table of experiments is very regular: the GPU implementation is about 16 times faster than the standard BP algorithm. Naturally, much better speed-ups have been achieved on GPUs; in our case, however, the reformulation transforms the problem in a harder one (even if the complexity class remains the same). Nevertheless, the GPU implementation is still able "to do better" than the standard sequential algorithm.

In view of an implementation of the optical processor in [2], we remark that it is likely to suffer of the same effect on the increased number of detected solutions that we have observed in our computational experiments. This opens new challenging for the conception and development of this kind of alternative computing devices.

Acknowledgments

We are grateful to Caroline Collange for the fruitful discussions and for the authorization to use one of the machines of her research group to perform our computational experiments.

We are also thankful to the three reviewers that provided very helpful comments on this paper. We hope that this extended version of the paper deposited on the HAL's open archives is able to better address all their comments.

This work is partially supported by the international project MULTIBIOSTRUCT funded by the ANR French funding agency (ANR-19-CE45-0019).

REFERENCES

[1] N.M. Freris, S.R. Graham, P.R. Kumar, *Fundamental Limits on Synchronizing Clocks Over Networks*, IEEE Transactions on Automatic Control **56**(6), 1352–1364, 2010.

[2] S.B. Hengeveld, N. Rubiano da Silva, D.S. Gonçalves, P.H. Souto Ribeiro, A. Mucherino, *An Optical Processor for Matrix-by-Vector Multiplication: an Application to the Distance Geometry Problem in 1D*, Journal of Optics **24**(1), 015701, 2022.

[3] K. Isupov, *Multiple-Precision Sparse MatrixVector Multiplication on GPUs*, Journal of Computational Science **61**, 101609, 2022.

[4] L. Liberti, C. Lavor, N. Maculan, *A Branch-and-Prune Algorithm for the Molecular Distance Geometry Problem*, International Transactions in Operational Research **15**, 1–17, 2008.

[5] L. Liberti, C. Lavor, N. Maculan, A. Mucherino, *Euclidean Distance Geometry and Applications*, SIAM Review **56**(1), 3–69, 2014.

[6] L. Liberti, B. Masson, J. Lee, C. Lavor, A. Mucherino, *On the Number of Realizations of Certain Henneberg Graphs arising in Protein Conformation*, Discrete Applied Mathematics **165**, 213–232, 2014.

[7] A. Monakov, A. Lokhmotov, A. Avetisyan, *Automatically Tuning Sparse Matrix-Vector Multiplication for GPU Architectures*, In: "High Performance Embedded Architectures and Compilers", Y.N. Patt, P. Foglia, E. Duesterwald, P. Faraboschi, X. Martorell (Eds.), Lecture Notes in Computer Science **5952**, Springer, 111–125, 2010.

[8] A. Mucherino, *Optimal Discretization Orders for Distance Geometry: a Theoretical Standpoint*, Lecture Notes in Computer Science **9374**, Proceedings of the 10^{th} International Conference on Large-Scale Scientific Computations (LSSC15), Sozopol, Bulgaria, 234–242, 2015.

[9] A. Mucherino, *On the Exact Solution of the Distance Geometry with Interval Distances in Dimension 1*. In: "Recent Advances in Computational Optimization", S. Fidanova (Ed.), Studies in Computational Intelligence **717**, 123–134, 2018.

[10] J. Saxe, *Embeddability of Weighted Graphs in k-Space is Strongly NP-hard*, Proceedings of 17^{th} Allerton Conference in Communications, Control and Computing, 480–489, 1979.

[11] A. Singer, *Angular Synchronization by Eigenvectors and Semidefinite Programming*, Applied and Computational Harmonic Analysis **30**(1), 20–36, 2011.

[12] P. Verissimo, M. Raynal, *Time in Distributed System Models and Algorithms*. In: "Advances in Distributed Systems, Advanced Distributed Computing: From Algorithms to Systems", S.K. Shrivastava, S. Krakowiak (Eds.), Springer, 1–32, 1999.

[13] V. Volkov, J.W. Demmel, *Benchmarking GPUs to Tune Dense Linear Algebra*, IEEE Conference Proceedings, ACM/IEEE conference on Supercomputing (SC08), 11 pages, 2008.

[14] Y-C. Wu, Q. Chaudhari, E. Serpedin, *Clock Synchronization of Wireless Sensor Networks*, IEEE Signal Processing Magazine **28**(1), 124–138, 2011.