









The MPI + CUDA Gaia AVU–GSR Parallel Solver Toward Next-generation Exascale Infrastructures

Valentina Cesare^{1,2,6} , Ugo Becciani^{1,2} , Alberto Vecchiato^{2,3} , Mario Gilberto Lattanzi³ , Fabio Pitari⁴,
Marco Aldinucci^{2,5} , and Beatrice Bucciarelli³ 

¹ INAF, Astrophysical Observatory of Catania, via Santa Sofia 78, 95123 Catania, CT, Italy; valentina.cesare@inaf.it, ugo.becciani@inaf.it

² ICSC—Centro Nazionale di Ricerca in High Performance Computing, Big Data and Quantum Computing alberto.vecchiato@inaf.it

³ INAF, Astrophysical Observatory of Turin, via Osservatorio 20, 10025 Pino Torinese, TO, Italy; mario.lattanzi@inaf.it, beatrice.bucciarelli@inaf.it

⁴ CINECA, via Magnanelli 6/3, 40033 Casalecchio di Reno, BO, Italy; f.pitari@cineca.it

⁵ University of Turin, Computer Science Department, corso Svizzera 185, 10149 Turin, TO, Italy; marco.aldinucci@unito.it

Received 2023 February 27; accepted 2023 June 16; published 2023 August 1

Abstract

We ported to the GPU with CUDA the Astrometric Verification Unit–Global Sphere Reconstruction (AVU–GSR) Parallel Solver developed for the ESA Gaia mission, by optimizing a previous OpenACC porting of this application. The code aims to find, with a [10, 100] μ arcsec precision, the astrometric parameters of $\sim 10^8$ stars, the attitude and instrumental settings of the Gaia satellite, and the global parameter γ of the parametrized Post-Newtonian formalism, by solving a system of linear equations, $\mathbf{A} \times \mathbf{x} = \mathbf{b}$, with the LSQR iterative algorithm. The coefficient matrix \mathbf{A} of the final Gaia data set is large, with $\sim 10^{11} \times 10^8$ elements, and sparse, reaching a size of ~ 10 –100 TB, typical for the Big Data analysis, which requires an efficient parallelization to obtain scientific results in reasonable timescales. The speedup of the CUDA code over the original AVU–GSR solver, parallelized on the CPU with MPI + OpenMP, increases with the system size and the number of resources, reaching a maximum of $\sim 14\times$, $>9\times$ over the OpenACC application. This result is obtained by comparing the two codes on the CINECA cluster Marconi100, with 4 V100 GPUs per node. After verifying the agreement between the solutions of a set of systems with different sizes computed with the CUDA and the OpenMP codes and that the solutions showed the required precision, the CUDA code was put in production on Marconi100, essential for an optimal AVU–GSR pipeline and the successive Gaia Data Releases. This analysis represents a first step to understand the (pre-)Exascale behavior of a class of applications that follow the same structure of this code. In the next months, we plan to run this code on the pre-Exascale platform Leonardo of CINECA, with 4 next-generation A200 GPUs per node, toward a porting on this infrastructure, where we expect to obtain even higher performances.

Unified Astronomy Thesaurus concepts: [Astronomy software \(1855\)](#); [Astrometry \(80\)](#); [Computational methods \(1965\)](#); [Galaxy kinematics \(602\)](#)

Online material: color figures

1. Introduction

In this epoch of technological evolution, the size of the problems to solve in several contexts is rapidly increasing and can also require up to ~ 10 –100 PB of storage. To allow the analysis of these *Big Data*, novel parallelization techniques have to be continuously defined to find solutions in human-size timescales. The architecture of the infrastructures is also consequently changing, becoming increasingly heterogeneous

(Carpenter et al. 2022), to accomplish the necessity of optimally computing data of these sizes, going toward the (pre-)Exascale era. The supercomputers will require an increasing number of computational nodes, which will have, in turn, a RAM memory organized in a multi-levels hierarchy of nonvolatile memories and hosts less performant than the accelerators (such as GPUs or FPGAs) that will be increasingly employed for calculations and will have an increasing memory and number of streaming multiprocessors. This configuration will also be likely to achieve the target of Green Computing, namely to process this amount of Big Data without excessively increasing the energy consumption while obtaining a high performance. Moreover, the infrastructures will need increasingly faster bridges between the CPU and the accelerators, to reduce the host-to-device (H2D) and the

⁶ Corresponding author.

device-to-host (D2H) data transfers bottleneck in the applications, and storage areas defined with parallel filesystems to guarantee a faster access to the data (Carpenter et al. 2022). Computer clusters such as Marconi100 (M100) of CINECA⁷ and JUWELS of Forschungszentrum Jülich⁸ are already going in this direction but a turning point will be provided by next-generation pre-Exascale infrastructures, such as the CINECA platform Leonardo,⁹ which started to be operative last November.

A typical science case that might involve a large amount of data is the *inverse problem*, that consists in estimating the parameters of a model from a set of observational measurements. Two possible approaches for this task are the *frequentist* and the *bayesian* ones. Concerning the former approach, one of the exploited computational techniques is the *LSQR* iterative algorithm, to solve large, ill-posed, overdetermined, and possibly sparse systems of equations (Paige & Saunders 1982a, 1982b). This algorithm is employed in several contexts, such as medicine (Bin et al. 2020; Guo et al. 2021), geophysics (Joulidehsar et al. 2018; Liang et al. 2019a, 2019b), geodesy (Baur & Austen 2005), industry (Jaffri et al. 2020), and astronomy (Borriello et al. 1986; Van der Marel 1988; Becciani et al. 2014; Naghibzadeh & van der Veen 2017; Cesare et al. 2021, 2022a, 2022b, 2022c). For a more in-depth discussion about the LSQR algorithm and other LSQR-based applications and libraries, see Section 2 of Cesare et al. (2022c).

As an example, in the astronomy context, this algorithm is employed by the Gaia Astrometric Verification Unit–Global Sphere Reconstruction (AVU–GSR) Parallel Solver. This code was developed for the ESA Gaia mission (Gaia Collaboration et al. 2022) under the Data Processing and Analysis Consortium (DPAC) (Mignard & Drimmel 2007), i.e., the scientific community of the mission, funded by the national space agencies, in charge of the definition of the data reduction pipelines (Vecchiato et al. 2018). The code has been in production since 2014 on M100 cluster according to an agreement between Istituto Nazionale di Astrofisica (INAF) and CINECA, with the support of the Italian Space Agency (ASI).

The Gaia AVU–GSR code solves with the LSQR algorithm an overdetermined system of linear equations (Becciani et al. 2014; Cesare et al. 2022c),

$$\mathbf{A} \times \mathbf{x} = \mathbf{b}, \quad (1)$$

where \mathbf{A} is the coefficient matrix, and \mathbf{b} and \mathbf{x} are the arrays of the known terms and of the solution, respectively. The matrix \mathbf{A} is sparse and it might contain $\sim 10^{11} \times 10^8$ elements for the expected final data set of Gaia. Even only considering its non-zero coefficients, it will occupy a large amount of memory (~ 10 – 100 TB).

⁷ <https://www.hpc.cineca.it/hardware/marconi100>

⁸ https://fz-juelich.de/ias/jsc/EN/Expertise/Supercomputers/JUWELS/JUWELS_node.html

⁹ <https://www.cineca.it/temi-caldi/Leonardo>

By solving this system, the AVU–GSR code finds, with an accuracy in the range of 10 – $100 \mu\text{arcsec}$ and of 10 – $100 \mu\text{arcsec yr}^{-1}$, the astrometric parameters (parallaxes, R , A , decl., and proper motions along these two directions) of $\sim 10^8$ stars in the Milky Way, the so-called primary stars (Vecchiato et al. 2018). The Gaia AVU–GSR code is a verification module of the same solution found with the software Astrometric Global Iterative Solution (AGIS; O’Mullane et al. 2011; Lindegren et al. 2012) adopting a different algorithm, to make the determination of the astrometric parameters more robust. Besides the astrometric parameters, the Gaia AVU–GSR solver finds the attitude and instrumental specifications of the Gaia satellite, and the global parameter γ of the Parametrized Post-Newtonian (PPN) formalism, with the same precision around $[10, 100] \mu\text{arcsec}$. The high accuracy of these parameters is essential to properly investigate the formation and the evolution of the Milky Way (e.g., Giammaria et al. 2021; Krolkowski et al. 2021) and to test Einstein’s theory of General Relativity (e.g., Vecchiato et al. 2003; Hees et al. 2018; Crosta et al. 2020; Butkevich et al. 2022).

The LSQR algorithm is the bulk of the AVU–GSR solver and it works by calculating, at each iteration, the iterative estimates of the known terms and of the solution arrays with the *aprod 1* and *aprod 2* functions:

$$\mathbf{b}^i + = \mathbf{A} \times \mathbf{x}^{i-1}, \quad (2)$$

and

$$\mathbf{x}^i + = \mathbf{A}^T \times \mathbf{b}^i, \quad (3)$$

which are the most computational demanding parts of the LSQR procedure, representing more than 90% of the entire calculation.

The last official in-production version of the Gaia AVU–GSR code was entirely parallelized on the CPU with a hybrid MPI + OpenMP approach. In Cesare et al. (2022c), we explored the feasibility of a GPU porting of the application by adopting a preliminary approach, where we replaced the OpenMP directives with the OpenACC ones. With this porting, the speedup of the OpenACC code over the OpenMP code, both run on M100, was of ~ 1.5 . In this paper, we present an optimization of the GPU parallelization of the code starting from the results of our first porting, where we replace the high-level parallelization approach, using OpenACC, with a low-level one, using CUDA (Cesare et al. 2022a). This implied a reorganization of several parts of the code but a substantial performance boost of $\sim 14\times$ over the OpenMP version, as tested on M100. This speedup might further improve on Leonardo, with GPUs having a larger memory and number of streaming multiprocessors than on M100, which is an optimistic estimate in perspective of a future porting on this platform. The CUDA code also showed to achieve a great numerical stability and to obtain parameters with the required accuracy, reasons for which it was put in production on M100.

The paper develops across the following sections. Section 2 summarizes the general structure of the Gaia AVU–GSR code, Section 3 describes the previous versions of the code, i.e., the MPI + OpenMP (Section 3.1) and the MPI + OpenACC (Section 3.2) ones, and Section 4 details the CUDA porting of the Gaia AVU–GSR code. A performance comparison with the OpenACC porting is presented throughout Section 4. Section 5 compares the performance of the MPI + CUDA and the MPI + OpenMP codes on M100 for a set of systems with increasing size and Section 6 compares the solutions of these systems to verify their consistency and quantify the accuracy of the obtained solutions. At last, Section 7 discuss the main results of the paper and presents the future analyses to be developed.

2. The Structure of the Gaia AVU–GSR Code

The black part of Algorithm 1 summarizes the general structure of the Gaia AVU–GSR code, which is common to the OpenMP, OpenACC, and CUDA versions. The preparatory phase consists in importing from binary files the quantities necessary to solve the system (e.g., the coefficient matrix and the known terms; line 1). To accelerate the convergence speed of the iterative procedure, the system is preconditioned before the starting of the LSQR algorithm. Specifically, we normalized the parameters of each column by the norm of the column itself (lines 2–3). The normalization factors of all the columns are stored in a 1D array, \mathbf{p} . The solution is re-multiplied by \mathbf{p} after the end of the LSQR algorithm (line 27). Then, the initial guess of the solution to be iteratively found with the LSQR algorithm is computed through the *aprod 2* function (see Equation (3); line 13). Each MPI process calculates a part of the solution that is then reduced among all the MPI processes (line 15).

After these passages, the LSQR procedure starts. The LSQR algorithm is a while loop (lines 17–26) that iterates the solution up to a convergence condition or until a maximum number of iterations set at runtime is reached. At each iteration, the two main steps are the execution of the *aprod* function in the modes 1 and 2 (lines 18 and 22). The *aprod 1* (Equation (2)) provides the iterative estimate of the known terms \mathbf{b} for each equation of the system and for a set of constraints equations (Vecchiato et al. 2018), required since the system is overdetermined. After the calculation of the *aprod 1*, the \mathbf{b} array is reduced among the MPI processes. Then, the *aprod 2* (Equation (3)) provides the iterative estimate of the solution array \mathbf{x} . Also for this step, the constraints equations are defined and the solution is reduced among the MPI processes. At the end of each iteration, the errors (variances) on the unknowns and the covariances between the different couples of unknowns are calculated (line 26). The convergence condition is achieved in the least-squares sense, when the residuals $r^i = b^i - A \times x^i$, estimated

at the i th iteration, go below a given tolerance, set to the machine precision ($\sim 10^{-16}$ on M100).

The coefficient matrix of the system \mathbf{A} is large and has a high sparsity degree. Specifically, for the expected final data set of Gaia, the matrix might contain $\sim 10^{11} \times 10^8$ elements (see Section 1). The rows of \mathbf{A} , i.e., the equations of the system, represent the observations of the Milky Way stars, where each star is observed $N_{\text{Obsperstar}} \sim 10^3$ times, besides the constraints equations. The number of the columns of \mathbf{A} is instead the number of unknowns to solve.

For each row, the coefficients are divided in their astrometric, attitude, instrumental, and global sections. The astrometric part of \mathbf{A} contains $N_{\text{Astro}} \times N_{\text{Stars}}$ coefficients per row. N_{Stars} is the number of stars considered in the system, in the range of $[10^6, 10^8]$, and $0 \leq N_{\text{Astro}} \leq 5$ is the number of astrometric coefficients per star and the number of non-zero astrometric coefficients per row. The total number of non-zero astrometric parameters is of $N_{\text{Obsperstar}} \times N_{\text{Astro}} \times N_{\text{Stars}} \in [10^9, 10^{12}]$ and they represent the $\sim 90\%$ of the coefficient matrix \mathbf{A} , where they follow a block-diagonal structure of N_{Stars} blocks. The $N_{\text{Obsperstar}}$ rows of each block are the astrometric parameters observed for the same star and the number of columns of each block is equal to N_{Astro} . In our current modelization, the attitude part has $N_{\text{Att}} = 12$ nonzero coefficients per row, organized in $N_{\text{Axes}} = 3$ blocks of $N_{\text{ParAxis}} = 4$ elements separated by N_{DFA} zeros, where $N_{\text{Axes}} = 3$ is the number of axes of the satellite attitude, $N_{\text{ParAxis}} = 4$ is the number of nonzero coefficients per axis, and N_{DFA} is the number of degrees of freedom of each axis. In the instrumental part, we have $0 \leq N_{\text{Instr}} \leq 6$ nonzero coefficients per row, distributed without a particular scheme. So far, we have considered in the global part only $N_{\text{Glob}} = 1$ coefficient, the γ parameter of the PPN formalism, or we have run without computing a global part.

To operate in human-size timescales, the calculations are performed with a dense coefficient matrix \mathbf{A}_d that only contains the nonzero coefficients of \mathbf{A} for each section. Therefore, the number of coefficients per row passes from $\sim 10^8$ to a maximum of $N_{\text{par}} = 24$, in our current modelization, and the total number of elements of \mathbf{A}_d is of $\sim 10^{11} \times 10^1$. The indexes that the astrometric, the attitude, and the instrumental coefficients of \mathbf{A}_d had in the original matrix \mathbf{A} are stored in two one-dimensional integer arrays, M_i (for the astrometric and attitude parts) and I_c (for the instrumental part), to map the correct positions of these parameters in the matrix \mathbf{A} . For further details about the structure of the coefficient matrix, see Sections 3 and 4 of Cesare et al. (2022c).

The system of equations is parallelized over the MPI processes such that different subsets of the total number of observations, n , are assigned to each MPI process. The one-dimensional integer array $N[\text{nproc}]$ stores the number of observations assigned to each MPI process, where *nproc* is the

Algorithm 1
Structure of the entire Gaia AVU–GSR application in CUDA

```

1 Import data (e. g.,  $\mathbf{A}$ ,  $\mathbf{b}$ ) from files
2 Calculate preconditioning array  $\mathbf{p}$ 
3 Normalization of  $\mathbf{A}$  by  $\mathbf{p}$ 
  // Get the number of the devices in the node
4 cuda_error = cudaGetDeviceCount(&deviceCount)
  // Set the number of the device in the node
5 cuda_error = cudaSetDevice(pid % deviceCount)
6 cuda_error = cudaMalloc(&Ad,dev,length(Ad))
7 ...
8 cuda_error = cudaMemcpy(Ad,dev,Ad,length(Ad),cudaMemcpyHostToDevice)
9 cuda_error = cudaMemcpy(Mi,dev,Mi,length(Mi),cudaMemcpyHostToDevice)
10 cuda_error = cudaMemcpy(Ic,dev,Ic,length(Ic),cudaMemcpyHostToDevice)
11 cuda_error = cudaMemcpy(bdev,b,length(b),cudaMemcpyHostToDevice)
12 cuda_error = cudaMemcpy(xdev,x,length(x),cudaMemcpyHostToDevice)
  // Calculation of the initial solution  $\mathbf{x}_0$ 
13 aprod 2 call (in each MPI process)
14 cuda_error = cudaMemcpy(x,xdev,length(x),cudaMemcpyDeviceToHost)
  // Reduction of the initial solution among the MPI processes
15 MPI_Allreduce(x0)
16 cuda_error = cudaMemcpy(xdev,x,length(x),cudaMemcpyHostToDevice)
  // LSQR algorithm
17 while (conv. cond. || max itn. reached) do
  // Iterative estimate of the known terms array  $\mathbf{b}$ 
18 aprod 1 call (in each MPI process)
19 cuda_error = cudaMemcpy(b,bdev,length(bConstraints),cudaMemcpyDeviceToHost)
  // Reduction of  $\mathbf{b}$  among the MPI processes
20 MPI_Allreduce(b)
21 cuda_error = cudaMemcpy(bdev,b,length(bConstraints),cudaMemcpyHostToDevice)
  // Iterative estimate of the solution array  $\mathbf{x}$ 
22 aprod 2 call (in each MPI process)
23 cuda_error = cudaMemcpy(x,xdev,length(x),cudaMemcpyDeviceToHost)
  // Reduction of  $\mathbf{x}$  among the MPI processes
24 MPI_Allreduce(x)
25 cuda_error = cudaMemcpy(xdev,x,length(x),cudaMemcpyHostToDevice)
26 Variances and covariances computation
27 Re-multiplication of the solution and of its variance by  $\mathbf{p}$ 
28 Print the solution to files

```

number of MPI processes defined at runtime. Figure 1 represents the system of equations parallelized on four MPI processes in one node of a computer cluster, where different colors refer to diverse MPI processes.

Concerning the rows of \mathbf{A} , whereas the observation equations are distributed among the MPI processes through the N array, the constraints equations, placed at the bottom of

the system, are replicated on each MPI process. For this reason, after the execution of the *aprod 1* function, the only part of the known terms array \mathbf{b} that has to be reduced among the MPI processes is the one related to the constraints equations. Given that the constraints equations represent a negligible fraction of the total number of equations, their replica was more convenient compared to their distribution among the MPI

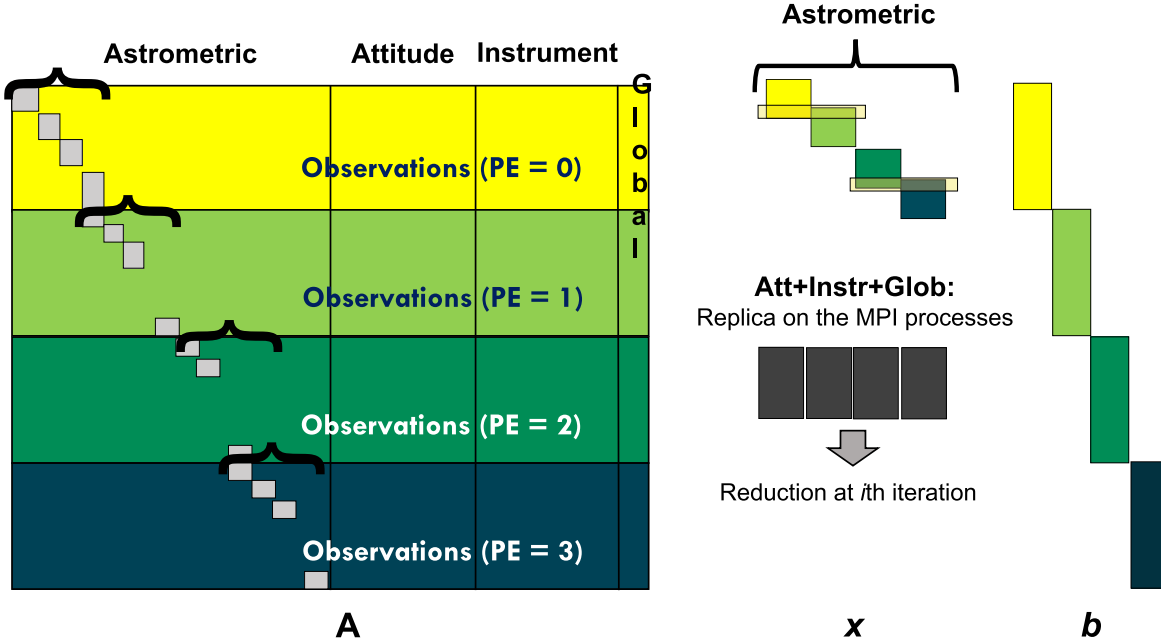


Figure 1. Parallelization scheme of the system of equations (Equation (1)) on four MPI processes in a single node of a computer cluster. Left panel: coefficient matrix A . Middle panel: unknowns array x . Right panel: known terms array b . Different colors (yellow, light green, dark green, and blue) refer to different MPI processes or processing elements (PE). The block-diagonal part in the left side of the coefficient matrix illustrates its nonzero astrometric section. In the middle panel, the four square blocks diagonally placed, and labeled as “Astrometric” represent the astrometric part of the solution array, distributed among the MPI processes. Instead, the four dark gray aligned blocks, labeled as “Att+Instr+Glob”, represent the attitude, instrumental, and global portions of the solution array, replicated on each MPI process, as written above. At the end of each iteration i , a reduction of the replicated portions of x is performed.

processes, which would have implied a rearrangement of the code.

Concerning the columns of A , the astrometric part is distributed among the MPI processes whereas the other three parts are replicated on them. Therefore, after the execution of the *aprod 2* function, only the attitude + instrumental + global parts of the solution array x are reduced among the MPI processes. The regular block-diagonal structure of the astrometric parameters made their distribution among the MPI processes more intuitive. Instead, the other three sections do not follow a regular pattern, which would have made their distribution on the MPI processes less trivial. Since, the attitude + instrumental + global parts only represent the 10% of the total system, their replica does not imply a substantial slowdown of the code.

3. Previous Parallelizations: MPI + OpenMP and MPI + OpenACC

3.1. The OpenMP Parallelization

In the in-production code, the observations assigned to each MPI process are further parallelized over the OpenMP threads. The left panels of Algorithms 2 and 3 highlights in boldface the regions of the code parallelized with OpenMP, namely the *aprod 1* and *2* functions. In the *aprod 1*, we parallelized the for

loop that iterates on the number of observations in each MPI process, $N[pid]$, with the `#pragma omp for` directive, where *pid* identifies the rank of the MPI process. Instead, in the *aprod 2* the most external for loop iterates from $N_i[tid][0]$ to $N_i[tid][1]$, where *tid* is the ID number of the OpenMP thread, that goes from 0 to *n*th, the total number of threads, and N_i is a one-dimensional integer array that contains the observations computed by each thread *tid*. Specifically, $N_i[tid][0]$ and $N_i[tid][1]$ are the first and the last observation computed by the thread *tid*.

3.2. The OpenACC Parallelization

In our preliminary porting to a GPU environment, the OpenMP parallelization model is replaced by OpenACC (Cesare et al. 2021, 2022a, 2022c). The middle panels of Algorithms 2 and 3 highlights in boldface the correspondent parts of the left panels, parallelized with OpenACC instead of OpenMP. For reasons of optimization, we divided the *aprod 1* function in four parallel regions, one for each section of the system, and we organized the *aprod 2* in a single parallel region. Each parallel region is enclosed within a `#pragma acc parallel` directive, which starts a parallel execution on the current device. In the *aprod 1*, the variable *sum* is defined within the `private` clause (lines 4, 14, 27, and 37 of Algorithm 2), which ensures each GPU thread to have a local

Algorithm 2
aprod 1 with OpenMP, OpenACC, and CUDA.

aprod 1 with OpenMP

```

0.1 int main(int argc, char **argv)
0.2 {
0.3 ...
0.4 #pragma omp parallel private(pid,sum)
    shared(N,w,Ad,b)
0.5 {
0.6     #pragma omp for
0.7     for i ← 0 to N[pid] do
0.8         sum = 0.0
0.9         // Astrometric sect.
0.10        k = i × Npar
0.11        for j ← 0 to NAstro do
0.12            sum = sum + Ad[k]w[j + offset[i]]
            k++
0.13        // Attitude sect.
0.14        k = i × Npar + NAstro
0.15        for j1 ← 0 to NAxis do
0.16            k2 = j1 × NDFA + offset[j1]
0.17            for j2 ← 0 to NParAxis do
0.18                sum = sum + Ad[k]w[j2 + k2]
                k++
0.19        // Instrumental sect.
0.20        k = i × Npar + NAstro + NAtt
0.21        for j ← 0 to NInstr do
0.22            sum = sum + Ad[k]w[ $\mathcal{F}(i, j)$  + offset]
            k++
0.23        // Global sect.
0.24        k = i × Npar + NAstro + NAtt + NInstr
0.25        for j ← 0 to NGlob do
0.26            sum = sum + Ad[k]w[j + offset]
            k++
0.27        b[i] = b[i] + sum
0.28    }
0.29    Constraints computation
0.30    MPIAllreduce(b)
0.31    ...
0.32 }

```

aprod 1 with OpenACC

```

a.1 int main(int argc, char **argv)
a.2 {
a.3 ...
a.4 // Astrometric sect.
a.5 #pragma acc parallel private(sum)
a.6 {
a.7     #pragma acc loop
a.8     for i ← 0 to N[pid] do
a.9         sum = 0.0
a.10        k = i × Npar
a.11        for j ← 0 to NAstro do
a.12            sum = sum + Ad[k + j]w[j + offset[i]]
            b[i] = b[i] + sum
a.13    }
a.14 // Attitude sect.
a.15 #pragma acc parallel private(sum)
a.16 {
a.17     #pragma acc loop
a.18     for i ← 0 to N[pid] do
a.19         sum = 0.0
a.20         k = i × Npar + NAstro
a.21         for j1 ← 0 to NAxis do
a.22             k2 = j1 × NDFA + offset[j1]
a.23             for j2 ← 0 to NParAxis do
a.24                 sum = sum + Ad[k + j2 + k1]w[j2 + k2]
            b[i] = b[i] + sum
a.25    }
a.26 // Instrumental sect.
a.27 #pragma acc parallel private(sum)
a.28 {
a.29     #pragma acc loop
a.30     for i ← 0 to N[pid] do
a.31         sum = 0.0
a.32         k = i × Npar + NAstro + NAtt
a.33         for j ← 0 to NInstr do
a.34             sum = sum + Ad[k + j]w[ $\mathcal{F}(i, j)$  + offset]
            b[i] = b[i] + sum
a.35    }
a.36 // Global sect.
a.37 #pragma acc parallel private(sum)
a.38 {
a.39     #pragma acc loop
a.40     for i ← 0 to N[pid] do
a.41         sum = 0.0
a.42         k = i × Npar + NAstro + NAtt + NInstr
a.43         for j ← 0 to NGlob do
a.44             sum = sum + Ad[k + j]w[j + offset]
            b[i] = b[i] + sum
a.45    }
a.46 }
a.47 Constraints computation
a.48 MPIAllreduce(b)
a.49 ...
a.50 }

```

aprod 1 with CUDA

```

// Astrometric sect.
c.1 _global_ void aprod1_KernelAstro
    (double* Ad,dev, double* wdev,
    double* bdev, long n,
    short NAstro, long Npar, ...)
c.2 {
c.3     i = blockIdx.x*blockDim.x+threadIdx.x
c.4     if (i < n)
c.5     {
c.6         sum = 0.0
c.7         k = i × Npar
c.8         for j ← 0 to NAstro do
c.9             sum += Ad,dev[k + j]wdev[j + offset[i]]
c.10        bdev[i] = bdev[i] + sum
c.11    }
c.12 }
// Attitude sect., axis 0
c.13 _global_ void aprod1_KernelAtt0
    (double* Ad,dev, double* wdev,
    double* bdev, long n,
    short NDFA, short NParAxis, long Npar, short
    NAstro ...)
c.14 {
c.15     k1 = 0 × NParAxis
c.16     i = blockIdx.x*blockDim.x+threadIdx.x
c.17     if (i < n)
c.18     {
c.19         sum = 0.0
c.20         k = i × Npar + NAstro
c.21         k2 = 0 × NDFA + offset[i]
c.22         for j2 ← 0 to NParAxis do
c.23             sum += Ad,dev[k + j2 + k1]wdev[j2 + k2]
c.24         bdev[i] = bdev[i] + sum
c.25     }
c.26 }
// Attitude sect., axis 1
c.27 _global_ void aprod1_KernelAtt1(...)
c.28 {...}
// Attitude sect., axis 2
c.29 _global_ void aprod1_KernelAtt2(...)
c.30 {...}
// Instrumental sect.
c.31 _global_ void aprod1_KernelInstr
    (double* Ad,dev, double* wdev,
    double* bdev, long n,
    short NAstro, short NAtt, short NInstr, long
    Npar, ...)
c.32 {
c.33     i = blockIdx.x*blockDim.x+threadIdx.x
c.34     if (i < n)
c.35     {
c.36         sum = 0.0
c.37         k = i × Npar + NAstro + NAtt
c.38         for j ← 0 to NInstr do
c.39             sum = sum + Ad,dev[k + j]wdev[ $\mathcal{F}(i, j)$  + offset]
c.40         bdev[i] = bdev[i] + sum
c.41     }
c.42 }
// Global sect.
c.43 _global_ void aprod1_KernelGlob
    (double* Ad,dev, double* wdev,
    double* bdev, long n, short NAstro,
    short NAtt, short NInstr, short NGlob, long
    Npar, ...)
c.44 {
c.45     i = blockIdx.x*blockDim.x+threadIdx.x
c.46     if (i < n)
c.47     {
c.48         sum = 0.0
c.49         k = i × Npar + NAstro + NAtt + NInstr
c.50         for j ← 0 to NGlob do
c.51             sum = sum + Ad,dev[k + j]wdev[j + offset]
c.52         bdev[i] = bdev[i] + sum
c.53     }
c.54 }
c.55 Constraints kernels
c.56 ...
c.57 int main(int argc, char **argv)
c.58 {
c.59 ...
c.60 TW = 1024
c.61 dim3 gridDim ((N[pid] - 1)/TW + 1,1,1)
c.62 dim3 blockDim (TW,1,1)
c.63 aprod1_KernelAstro<<<gridDim,blockDim>>
    (Ad,dev,wdev,bdev,N[pid],NAstro,Npar, ...)
c.64 aprod1_KernelAtt0<<<gridDim,blockDim>>
    (Ad,dev,wdev,bdev,N[pid],NDFA,NParAxis,
    Npar,NAstro ...)
c.65 aprod1_KernelAtt1<<<gridDim,blockDim>>
    (Ad,dev,wdev,bdev,N[pid],NDFA,NParAxis,
    Npar,NAstro ...)
c.66 aprod1_KernelAtt2<<<gridDim,blockDim>>
    (Ad,dev,wdev,bdev,N[pid],NDFA,NParAxis,
    Npar,NAstro ...)
c.67 aprod1_KernelInstr<<<gridDim,blockDim>>
    (Ad,dev,wdev,bdev,N[pid],NAstro,NAtt,
    NInstr,Npar, ...)
c.68 aprod1_KernelGlob<<<gridDim,blockDim>>
    (Ad,dev,wdev,bdev,N[pid],NAstro,NAtt,
    NInstr,NGlob,Npar, ...)
c.69 Constraints kernels<<<...>>(...)
c.70 cudaDeviceSynchronize()
c.71 MPIAllreduce(b)
c.72 ...
c.73 }

```

Algorithm 3
aprod 2 with OpenMP, OpenACC, and CUDA.

```

aprod 2 with OpenMP
o.1 int main(int argc, char **argv)
o.2 {
o.3 ...
o.4 #pragma omp parallel private(pid,tid,nth)
    shared(N,α,Ad,b)
o.5 {
o.6 // ID number of the OpenMP thread
    tid = omp_get_thread_num()
o.7 // Number of OpenMP threads
    nth = omp_get_num_threads()
o.8 for i ← Nt[tid][0] to Nt[tid][1] do
o.9     // Astrometric sect.
    k = i × Npar
o.10 for j ← 0 to NAstro do
o.11     α[j + offset[i]] = α[j + offset[i]] + Ad[k]b[i]
o.12     k++
o.13 // Attitude sect.
    k = i × Npar + NAstro
o.14 for j1 ← 0 to NAxes do
o.15     k2 = j1 × NDFA + offset[i]
o.16     for j2 ← 0 to NParAxis do
o.17     α[j2 + k2] = α[j2 + k2] + Ad[k]b[i]
o.18     k++
o.19 // Instrumental sect.
    k = i × Npar + NAstro + NAtt
o.20 for j ← 0 to NInstr do
o.21     α[F(i, j) + offset] =
o.22     α[F(i, j) + offset] + Ad[k]b[i]
o.23 // Global sect.
    k = i × Npar + NAstro + NAtt + NInstr
o.24 for j ← 0 to NGlob do
o.25     α[j + offset] = α[j + offset] + Ad[k]b[i]
o.26     k++
o.27 }
o.28 Constraints computation
o.29 MPI_Allreduce(α)
o.30 ...
o.31 }

aprod 2 with OpenACC
a.1 int main(int argc, char **argv)
a.2 {
a.3 ...
a.4 #pragma acc parallel
    a.5 {
a.6 #pragma acc loop
    for i ← 0 to N[pid] do
a.7     // Astrometric sect.
    k = i × Npar
a.8 for j ← 0 to NAstro do
a.9     #pragma acc atomic
a.10     α[j + offset[i]] =
a.11     α[j + offset[i]] + Ad[k + j]b[i]
a.12 // Attitude sect.
    k = i × Npar + NAstro
a.13 for j1 ← 0 to NAxes do
a.14     k1 = j1 × NParAxis
a.15     k2 = j1 × NDFA + offset[i]
a.16     for j2 ← 0 to NParAxis do
a.17     #pragma acc atomic
a.18     α[j2 + k2] =
a.19     α[j2 + k2] + Ad[k + j2 + k1]b[i]
a.20 // Instrumental sect.
    k = i × Npar + NAstro + NAtt
a.21 for j ← 0 to NInstr do
a.22     #pragma acc atomic
a.23     α[F(i, j) + offset] =
a.24     α[F(i, j) + offset] + Ad[k + j]b[i]
a.25 // Global sect.
    k = i × Npar + NAstro + NAtt + NInstr
a.26 for j ← 0 to NGlob do
a.27     #pragma acc atomic
a.28     α[j + offset] = α[j + offset] + Ad[k + j]b[i]
a.29 }
a.30 Constraints computation
a.31 MPI_Allreduce(α)
a.32 ...
a.33 }

aprod 2 with CUDA
// Astrometric sect.
c.1 __global__ void aprod2_Kernel_astro
(double* Ad,dev, double* αdev,
 double* bdev, long n,
 short NAstro, long Npar, ...)
c.2 {
c.3     i = blockIdx.x*blockDim.x+threadIdx.x
c.4     if (i < n)
c.5     {
c.6         k = i × Npar
c.7         for j ← 0 to NAstro do
c.8         atomicAdd(&α[j + offset[i]], Ad[k + j]b[i])
c.9     }
c.10 }
// Attitude sect., axis 0
c.11 __global__ void aprod2_Kernel_att0
(double* Ad,dev, double* αdev,
 double* bdev, long n,
 short NDFA, short NParAxis, long Npar, short
 NAstro ...)
c.12 {
c.13     k1 = 0 × NParAxis
c.14     i = blockIdx.x*blockDim.x+threadIdx.x
c.15     if (i < n)
c.16     {
c.17         k = i × Npar + NAstro
c.18         k2 = 0 × NDFA + offset[i]
c.19         for j2 ← 0 to NParAxis do
c.20         atomicAdd(&α[j2 + k2], Ad[k + j2 + k1]b[i])
c.21     }
c.22 }
// Attitude sect., axis 1
c.23 __global__ void aprod2_Kernel_att1(...)
c.24 {...}
// Attitude sect., axis 2
c.25 __global__ void aprod2_Kernel_att2(...)
c.26 {...}
// Instrumental sect.
c.27 __global__ void aprod2_Kernel_instr
(double* Ad,dev, double* αdev,
 double* bdev, long n,
 short NAstro, short NAtt, short NInstr, long
 Npar, ...)
c.28 {
c.29     i = blockIdx.x*blockDim.x+threadIdx.x
c.30     if (i < n)
c.31     {
c.32         k = i × Npar + NAstro + NAtt
c.33         for j ← 0 to NInstr do
c.34         atomicAdd(&α[F(i, j) + offset], Ad[k + j]b[i])
c.35     }
c.36 }
// Global sect., part 1
c.37 __global__ void aprod2_Kernel_globSum1(double* Ad,dev,
 double* αdev, double* bdev, long n, short
 NAstro, short NAtt, short NInstr, long Npar,
 double* αdev,sum, int j, ...)
c.38 {...}
// Global sect., part 2
c.39 __global__ void aprod2_Kernel_globSum2(double* αdev,
 long arraySize, short NAstro, double* garr, int j,
 ...)
c.40 {...}
Constraints kernels
c.42 ...
c.43 static const int blockSize = 1024
c.44 static const int gridSize = 24*16
c.45 int main(int argc, char **argv)
c.46 {
c.47 ...
c.48 TW = 1024
c.49 dim3 gridDim ((N[pid] - 1)/TW + 1,1,1)
c.50 dim3 blockDim (TW,1,1)
c.51 aprod2_Kernel_astro<<<gridDim,blockDim,0,stream1>>>
(Ad,dev,αdev,bdev,N[pid],NAstro,Npar, ...)
c.52 aprod2_Kernel_att0<<<gridDim,blockDim,0,stream2>>>
(Ad,dev,αdev,bdev,N[pid],NDFA,NParAxis,
 Npar,NAstro ...)
c.53 aprod2_Kernel_att1<<<gridDim,blockDim,0,stream3>>>
(Ad,dev,αdev,bdev,N[pid],NDFA,NParAxis,
 Npar,NAstro ...)
c.54 aprod2_Kernel_att2<<<gridDim,blockDim,0,stream4>>>
(Ad,dev,αdev,bdev,N[pid],NDFA,NParAxis,
 Npar,NAstro ...)
c.55 aprod2_Kernel_instr<<<gridDim,blockDim,0,stream5>>>
(Ad,dev,αdev,bdev,N[pid],NAstro,NAtt,
 NInstr,Npar, ...)
c.56 for j ← 0 to NGlob do
c.57     aprod2_Kernel_globSum1<<<gridSize,blockSize>>>
(Ad,dev,αdev,bdev,NAstro,NAtt,
 NInstr,Npar,αdev,sum,j, ...)
c.58     aprod2_Kernel_globSum2<<<1,blockSize>>>
(αdev,gridSize,NAstro,αdev,sum,j, ...)
c.59 Constraints kernels<<<...>>>(...)
c.60 cudaDeviceSynchronize()
c.61 MPI_Allreduce(α)
c.62 ...
c.63 }

```

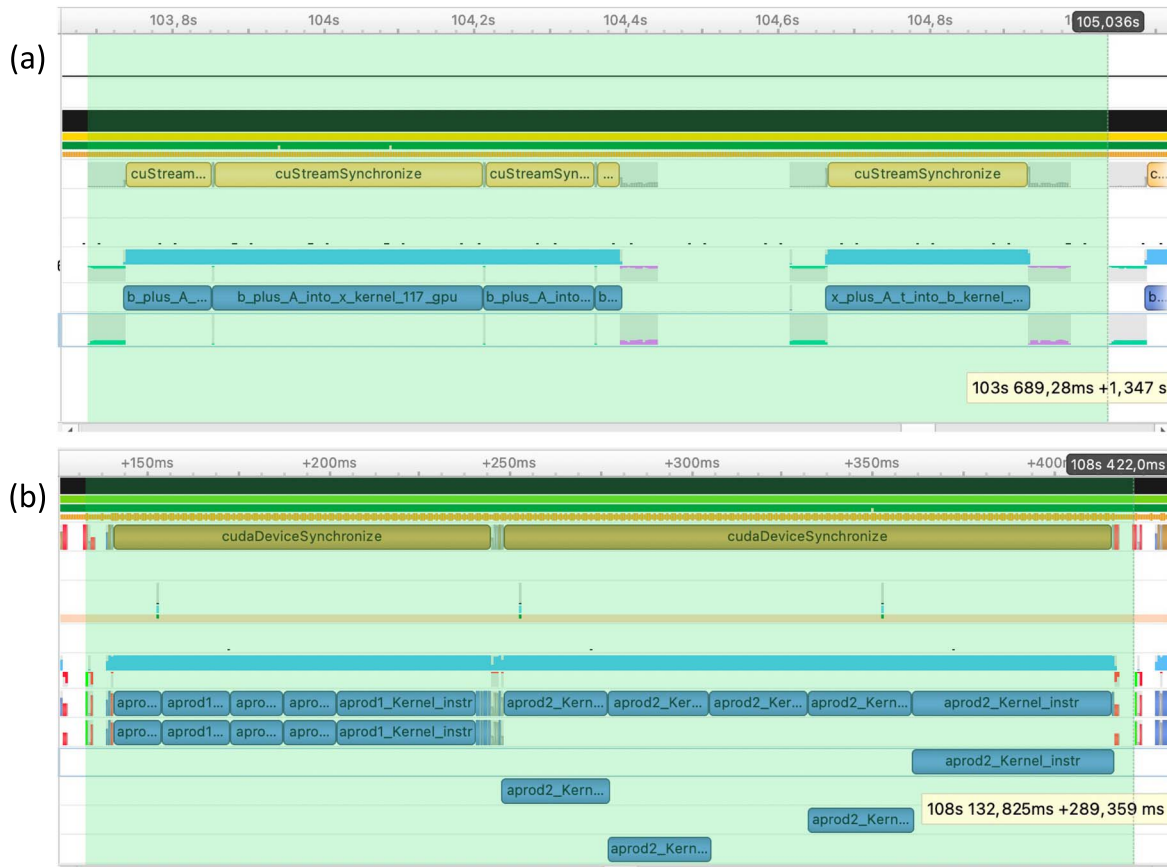


Figure 2. Result of the NVIDIA Nsight Systems profiler for a run of the OpenACC code (Figure 2(a)) and of the CUDA code (Figure 2(b)) parallelized on 4 MPI processes in one node of M100, for a system that occupies 50 GB of memory. The two outputs show a zoom-in of a single iteration of the LSQR algorithm, better highlighted with a large transparent light green box, and refer to one of the 4 MPI processes. Within the light green box of each panel, the blue regions represent the sections of the code parallelized on the GPU, the green and the purple/red regions illustrate the H2D and D2H data movements (very small in panel 2(b)), and the blank gaps between different regions are related to the sections of code still running on the CPU. The timescale above each panel shows the absolute time from the beginning of the programs execution and the small yellow rectangles at the bottom-right corner of each transparent light green box shows the iteration time (1.347 s and 0.289359 s for the OpenACC and the CUDA codes, respectively).

copy of it. In both *aprod 1* and *aprod 2*, we parallelized the most external for loop in each parallel region with the `#pragma acc loop` directive (lines 6, 16, 29, and 39, in Algorithm 2, and line 6, in Algorithm 3). These for loops iterate on the observations related to each MPI process from 0 to N [pid], also in the *aprod 2* function, where the array N_i is no more needed since we do not use any longer the OpenMP threads. In the *aprod 2*, the `#pragma acc atomic` directive (lines 10, 17, 21, and 25, of Algorithm 3) prevents the GPU threads to simultaneously overwrite the same elements of the array x , i.e., it avoids a data race condition.

We tested the performance of the MPI + OpenACC code on M100, with 4 NVIDIA Volta V100 GPUs per node having 16 GB of memory each. Figure 2(a) shows the output of the NVIDIA Nsight System profiler¹⁰ correspondent to

one iteration of the LSQR algorithm, highlighted with a large transparent light green box, for a system occupying 50 GB of memory. The system was run on four MPI processes in one node of M100 and the shown profiler output refers to one of the four processes. The timescale at the top of the panel shows the absolute time from the beginning of the program execution and the small yellow rectangle at the bottom-right corner of the light green box shows the iteration time, equal to 1.347 s. Within the light green box, the profiler shows the code regions parallelized on the GPU (blue), dedicated to data transfers (green and purple, for the H2D and D2H directions), and to calculation on the CPU (blank spaces between different regions). The blue regions labeled as “*b_plus...*” and as “*x_plus...*” show the *aprod 1* and *2* functions, respectively. The time fractions of one iteration due to GPU computation is $\sim 70\%$, whereas the ones due to data transfers

¹⁰ <https://developer.nvidia.com/nsight-systems>

and CPU computation are of $\sim 15\%$. This shows that the code is *compute bound*, namely data copies are subdominant compared to computation. This is essential for a GPU-ported application that, if data movements are not properly managed, can result in an even worse performance than the correspondent CPU version.

With this parallelization, the OpenACC code accelerates of $\sim 1.5\times$ over the OpenMP version. Specifically, the speedup is due to the porting of the *aprod 2* region, that accelerates of $\sim 3.6\times$, whereas the *aprod 1* region loses in performance of $\sim 0.8\times$ (Cesare et al. 2022c). Further optimizations are possible, as better detailed in the following sections.

4. The CUDA Porting of the Gaia AVU–GSR Code

To further improve the performance of the Gaia AVU–GSR solver, we decided to change the parallelization approach. Instead of optimizing the high-level OpenACC parallelization, which might be possible, for example by manually defining the grid of the GPU threads on which each parallel region is run rather than leaving this task to the compiler, we decided to adopt a low-level model, that is the NVIDIA native language CUDA. This choice was driven by the fact that, in future further optimizations, this approach will better lead us to manually tune some parameters directly related to the device. In the CUDA code, we manually allocated the GPU threads where to run each parallel region in a grid of threads blocks, each one customized to match the GPU architecture and the topology of the problem to solve.

Whereas the high-level OpenACC porting implied a minimal re-design of the application, ideal for beginner users (Cesare et al. 2020; Aldinucci et al. 2021), at the expense of a possible performance loss, the low-level parallelization with CUDA required a substantial re-engineering of the code structure. This can be seen from the left, middle, and right columns of Algorithms 2 and 3, that represent the parallelization of the *aprod 1* and 2 functions with OpenMP, OpenACC, and CUDA, respectively. Comparing the left and the middle columns, we can see that the structure of the OpenMP and of the OpenACC *aprod 1* and 2 functions, both parallelized through high-level directives, are very similar, whereas the right columns of the same algorithms show that the structure of the CUDA *aprod* functions is different.

In the below sections, we detail the parallelization of the CUDA code on multiple GPUs (Section 4.1), the definition of the CUDA kernels for the *aprod 1* and 2 functions (Section 4.2), the GPU porting of regions that in the OpenACC code were still running on the CPU (Section 4.3), the management of the data-transfers between the host and the device (Section 4.4), and the compilation of the application (Section 4.5).

4.1. Multi-GPU parallelization

As the OpenACC code (Cesare et al. 20221, 2022a, 2022c), the CUDA code runs on multiple GPUs, according to the number of MPI processes set at runtime. Specifically, the MPI processes are scheduled on the GPUs of the node in a round-robin fashion. This operation is performed with the commands at lines 4 and 5 of Algorithm 1 highlighted in gray. The optimal configuration to run the code is to set the number of MPI processes per node to the number of the GPUs of the node (4 on M100), since it allows to obtain the best performance by exploiting all the GPUs of the node and simultaneously employing the minimal number of MPI resources, as also shown in Section 7.1 of Cesare et al. (2022c).

4.2. CUDA Kernels Definition in *aprod 1* and *aprod 2* Functions

The right columns of Algorithms 2 and 3 show, in boldface, the definition of the CUDA kernels for the astrometric, the attitude, the instrumental, and the global sections of the *aprod 1* and 2 functions (lines c.1–55, in Algorithm 2, and lines c.1–41, in Algorithm 3) and their call in the main scope of the program (lines c.63–68, in Algorithm 2, and lines c.51–59, in Algorithm 3). The index $i = \text{blockIdx.x} * \text{blockDim.x} + \text{threadIdx.x}$ defined within the kernels, is the global index of the GPU thread within the grid of blocks of threads, where blockIdx.x is the index of the block inside the grid, blockDim.x is the size of the block in threads unit, and threadIdx.x is the thread index local to each block. The arrays involved in the calculations in the GPU kernels, such as the dense system matrix \mathbf{A}_d , the solution array \mathbf{x} , and the known terms array \mathbf{b} , have to be first allocated and then copied on the GPU. In Algorithm 1, only the device allocation of \mathbf{A}_d is highlighted gray, as an example (line 6), whereas all the H2D and D2H copies are reported in gray. The arrays allocated on the device are identified with the “dev” subscript, as we can see in the kernels in the left columns of Algorithms 2 and 3.

Comparing the right and the middle columns of Algorithms 2 and 3, we can see that the content of the CUDA kernels that compute the different sections of the system, except for the global part of the *aprod 2* function, are equivalent to the correspondent parts in the OpenACC code, which are directly defined in the main scope of the program within the for loops iterating on the number of observations assigned to each MPI process pid , parallelized with the `#pragma acc loop` directive.

To parallelize the for loops iterating on the observations assigned to each MPI process, from observation 0 to observation $N[pid]$, in each section of the system, the index of the GPU thread was directly mapped to the index of the observation. In this way, each thread can independently perform the product $\mathbf{b} = \mathbf{A}_d \times \mathbf{x}$, in the *aprod 1* kernels, and $\mathbf{x} = \mathbf{A}_d^T \times \mathbf{b}$, in the *aprod 2* kernels. In these CUDA kernels,

the for loop syntax disappears and it is replaced by the if-condition $i < N[pid]$ (lines c.4, c.17, c.34, and c.46 of Algorithm 2, and lines c.4, c.15, and c.30 of Algorithm 3), to avoid the thread index i to point to a memory address beyond the size of the product arrays.

It is essential to define, in each kernel, the hierarchy of the GPU threads to best match the GPU architecture and the topology of the problem to solve, which implies an efficient exploitation of the hardware of the device and results in an optimal performance. In our case, the topology of the problem is one-dimensional, since the product arrays are 1D. The grid of threads can be defined in a Cartesian coordinate space set by the x , y , and z axes. The three directions are not equivalent to each other: specifically, along the x direction it is possible to allocate more threads than along the y and the z directions. In particular, on a V100 GPU we can allocate $\sim 2 \times 10^9$ threads along the x direction and $\sim 6.5 \times 10^4$ threads along the y and z directions. We thus defined the grids of threads along the x direction, as identified by the `.x` specification (lines c.3, c.16, c.33, and c.45 of Algorithm 2, and lines c.3, c.14, and c.29 of Algorithm 3).

As in the OpenACC code, in the *aprod 2* kernels the operations $\mathbf{x} += \mathbf{A}_d^T \times \mathbf{b}$ are performed atomically. In CUDA, the atomic operation is performed with the `atomicAdd` function, that takes as first argument the memory address of the element of the x array where the result is cumulated and as second argument the quantity that has to be cumulated (lines c.8, c.20, and c.34 of Algorithm 3).

Performing different tests, we verified that defining more kernels allows to save the $\sim 10\%$ – 30% of the computation time compared to perform more operations in the same kernel. For this reason, we parallelized the four sections of the system both in the *aprod 1* and in the *aprod 2* on more kernels, differently from the *aprod 2* in the OpenACC code, which was defined within a single parallel region. Moreover, we also split the calculation of the attitude section, both in the *aprod 1* and *2* regions, in three kernels, one for each attitude axis. In Algorithms 2 and 3, we only report the attitude kernel for axis 0 since the kernels for the other two axes are equivalent.

The parallelization of the global part of the *aprod 2* was less trivial than the other three parts. Looking at the OpenACC column of Algorithm 3, at line a.26, we can see that the index of the element of the x array where the result of the atomic operation is cumulated does not depend on the index of the observation i , differently from the other three sections (lines a.11, a.18, a.22, c.8, c.20, and c.34 of Algorithm 3). This might cause a bottleneck in this point of the code since there is no parallelism over the GPU threads, whereas, in the astrometric, attitude, and instrumental parts of the *aprod 2*, the access to the element of the x array where the result is cumulated occurs in parallel for each thread i matched to the observation i .

So far, for scientific purposes of the current production, we did not derive the γ PPN parameter and, thus, this section does not represent a bottleneck. However, the γ parameter will be calculated in upcoming runs to test General Relativity, and we cannot exclude that future astrometric models will have more global parameters, which makes necessary to properly parallelize this region of code. To verify how leaving the atomic operation in the *aprod 2* global part would affect the performance of the code, we ran a 6 GB and a 50 GB system with 1 and 5 global parameters. Even with one global parameter, the computation of the *aprod 2* global section dominates over the other sections. We, thus, decided to rearrange the parallelization of this part. We removed the atomic operation and we implemented in CUDA the sum at line a.26 of Algorithm 3 with a parallel reduction, which also exploits the GPU shared memory. This reduce sum is organized in two kernels, the former parallelized on a certain number of blocks, each of which computes a partial sum saved in the array $x_{dev,sum}$ (line c.37, Algorithm 3), and the latter parallelized on a single block of threads, which combines all the partial results in x_{dev} (line c.39, Algorithm 3). Adopting this new implementation, we obtain a $\sim 20\times$ speedup for the global part of the *aprod 2* function and a speedup of $[1.5, 3]x$ for the entire *aprod 2* region. A deeper description of this implementation for the *aprod 2* global part will be object of a future work.

The grid of threads on which the kernels are parallelized, except for the *aprod 2* global kernels, are set through the `gridDim` and `blockDim` vectors, defined with the `dim3` integer vector type. The `gridDim` vector contains three elements, i.e., the number of blocks of threads in the grid along the x , y , and z directions. Since we defined a 1D grid, we have 1 block along the y and z directions. Instead, the `blockDim` vector contains the number of threads in each block again along the x , y , and z directions. As in `gridDim`, the second and the third element of `blockDim` are set to 1.

On a V100 GPU, the maximum number of threads per block is 1024. We set the number of threads per block, TW , to 1024, since it allows to obtain the best performance. Since the regions that we parallelized with the CUDA kernels correspond to the for loops that iterate from observation 0 to observation $N[pid]$ and that the index i of the observation is directly mapped to the index i of the GPU thread, the number of blocks should be $N[pid]/TW$ to properly fit the problem. If $N[pid]$ were an exact multiple of TW , this would result in a grid of $N[pid]$ threads. Yet, this is not necessary the case, and to avoid defining a grid with less threads than required, the number of blocks is set to $N_{bl} = (N[pid] - 1)/TW + 1$. Since the total number of threads $N_{th} = N_{bl} \times TW = N[pid] + TW - 1 > N[pid]$, the control condition $ifi < n$ defined in the kernels (lines c.4, c.17, c.34, and c.46 of Algorithm 2 and lines c.4, c.15, and c.30 of Algorithm 3) is necessary to avoid memory overflows. The scalar n coincides with $N[pid]$, as specified by the parameter $N[pid]$ passed to the kernel at lines c.63–c.68 of Algorithm 2 and

at lines c.51–c.55 of Algorithm 3. For the *aprod 2* global kernels, we defined a different grid (lines c.43 and c.44 of Algorithm 3), where the number of threads per block is always set to 1024 and the number of blocks employed for the reduction operation is 24×16 , empirically obtained such that it provided the best performance.

In the kernels call within the main scope of the program, the parameters passed in the angle brackets are the number of blocks and of threads per block. In the *aprod 2*, two additional arguments are passed within the angle brackets of the astrometric, the attitude, and the instrumental kernels. The third parameter sets the amount of the GPU shared memory to be used by the kernel, in this case 0 bytes, and the fourth argument is the identifier of the execution queue, called stream, on the GPU. This allows to execute these kernels asynchronously, which is possible without impairing the code correctness due to the atomic operations. The overlapping execution regions between the different *aprod 2* kernels represent a very minor fraction of the kernels execution, since the number of threads that can concurrently run on the GPU is limited to the number of GPU cores, much smaller than the number of threads in the grid. Yet, the asynchronous computation is useful to reduce the latencies between the successive kernels calls, essential when a large number of iterations is required to reach the convergence of the LSQR algorithm.

After the call of all the CUDA kernels of the *aprod 1* and *aprod 2* regions, a `cudaDeviceSynchronize()` barrier is set to wait all the kernels to end their computation (line c.70 of Algorithm 2 and line c.60 of Algorithm 3). This is necessary since, as soon as a CUDA kernel is called, the control returns immediately to the host and the CPU operations defined immediately after the kernels calls, i.e., the MPI reduction operations that combine the partial results obtained from the *aprod 1* and *aprod 2* (line c.71 of Algorithm 2 and line c.61 of Algorithm 3), would run concurrently to the calculations performed by the kernels.

Figure 2(b) shows the output of the NVIDIA Nsight System profiler for a 50 GB run of the CUDA code correspondent to the one of Figure 2(a) for the OpenACC code, i.e., parallelized on 4 MPI processes in one node of M100. As for Figure 2(a), the top timescale refers to the absolute time from the start of the program execution and the value in the yellow rectangle shows the time for the illustrated iteration (0.289359 s). Comparing the outputs of Figures 2(a) and (b), the *aprod 1* and 2 regions computed with the CUDA code present a $6.4\times$ and $1.6\times$ speedup over the same sections computed with the OpenACC application, respectively, correspondent to a speedup of $5.1\times$ and $5.8\times$ over the OpenMP code for an analogous run (Cesare et al. 2022c, 2022a). Considering one entire iteration, the CUDA code accelerates of $\sim 5\times$ over the OpenACC code and of $\sim 7\times$ over the OpenMP code. However, the speedup increases with the memory occupied by the system and with the

resources employed for the parallelization, as better illustrated in Section 5.

4.3. GPU Porting of the CPU Regions of the Code

Besides optimizing the parallelization of the *aprod 1* and *aprod 2* regions, we also ported to the GPU other sections of code that in the OpenACC version were running on the CPU. Looking at Figure 2(a), we can clearly see that many code regions were still running on the CPU (blank gaps).

First of all, we ported to the GPU the computation of the constraints equations both for the *aprod 1* and the *aprod 2* functions. With this porting, the computation time of these regions remains basically unaffected, since their calculation was already very fast ($\sim 10^{-4}$ s) on the CPU. However, porting these regions is essential to reduce the H2D and D2H data copies, since it avoids to entirely copy the \mathbf{b} and the \mathbf{x} arrays back to the host to perform the same calculations on the CPU.

Second of all, we ported to the GPU the calculation of the quantity to be compared with the tolerance of 10^{-16} that determines the convergence of the LSQR algorithm. This quantity depends on the norms, β and α , of the \mathbf{b} and the \mathbf{x} arrays. In the AVU–GSR code, the norm of these two arrays is calculated, in each MPI process, by square summing the array elements divided by the array maximum local to the MPI process and by multiplying this normalized squared sum by this maximum:

$$|\mathbf{a}|_{\text{loc}} = \max_{\text{loc}}(a) \times \sqrt{\sum_{i=0}^N \frac{a_i^2}{\max_{\text{loc}}(a)^2}}, \quad (4)$$

where \mathbf{a} is either \mathbf{b} or \mathbf{x} . Then, the global norm of the array, $|\mathbf{a}|$, is reduced among the MPI processes and sent back to each MPI process.

This method to compute the norm is adopted not to loose in numerical precision. In the MPI + OpenMP and MPI + OpenACC codes, this norm is computed on the CPU with the `cbblas_dnrn2` function of the `cbblas` libraries (Galassi et al. 2018).¹¹ In the MPI + CUDA version, we ported to the GPU the calculation of this norm by computing the local maximum and the squared and scaled sum of the array elements with a parallel reduce operation. With this porting, the computation of β and α accelerates of $\sim 35\times$ over the CPU implementation.

By porting these calculations to the GPU, the time fraction of one iteration due to CPU computation reduces from $\sim 15\%$ (see Section 3.2), to $\sim 3\%$, and the time fraction due to GPU computation increases from $\sim 70\%$ (see Section 3.2) to $\sim 90\%$. This can be visually seen from Figure 2(b), where the blank regions of the profiler are drastically reduced compared to Figure 2(a).

¹¹ <http://www.gnu.org/software/gsl/>

4.4. H2D and D2H Data Transfers

In the OpenACC code, we copied, at each iteration, both the entire \mathbf{b} and \mathbf{x} arrays, from the host to the device before the beginning of the *aprod 1* and *aprod 2* functions, and from the device to the host, after the end of the same functions. The \mathbf{b} and the \mathbf{x} arrays only represent the 5% of the total memory occupied by the system of equations (Cesare et al. 2022c) and consequently the time fraction of one iteration due to data copies was anyway subdominant compared to the time fraction due to GPU computation (see Section 3.2 and Figure 2(a)). However, some of these copies were unnecessary and they could be further reduced.

In the CUDA code, the first copy of the entire \mathbf{b} and \mathbf{x} arrays on the device is performed before the beginning of the LSQR cycle (lines 11 and 16 of Algorithm 1). Since the *aprod 1* only modifies the \mathbf{b} array, we only copy back to the host this array after the execution of the *aprod 1*, necessary operation since the \mathbf{b} array has to be reduced among the MPI processes. In fact, only the constraints part of the \mathbf{b} array has to be reduced among the MPI processes (see Section 2). For this reason, we only copy back to the host the final portion of the \mathbf{b} array correspondent to the constraints part (see “length($\mathbf{b}_{\text{Constraints}}$)” in the `cudaMemcpy` commands at lines 19 and 21 of Algorithm 1), which represents a minor fraction of the entire array. The same portion of the \mathbf{b} array is again copied to the device after the reduction operation.

For the same reason, since the *aprod 2* only modifies the \mathbf{x} array, the \mathbf{x} array alone is copied back to the host after the execution of the *aprod 2*. After the copy, the \mathbf{x} array is reduced among the MPI processes on the host and then is again copied to the device.

Rearranging the data copies in this way, the time fraction of one iteration due to data copies reduces from $\sim 15\%$, in the OpenACC code, to $\sim 3\%$, in the CUDA code (see Section 3.2 and Figure 2). The data copies in the CUDA code are highlighted in bold gray in Algorithm 1.

4.5. Compilation

To compile the MPI + CUDA code, written both in C and C++, we wrote a Makefile and we employed the `nvcc` CUDA compiler driver for the CUDA release 11.3 and the version 21.5-0 of the `nvc++` compiler. We compile with the `-arch = sm_70` option to target the Volta architecture of the GPU, present on M100.

5. Performance Tests

The MPI + OpenMP code has been in production since 2014 and it has run on all the Tier0 systems of CINECA. It is currently running on M100, which has 980 compute nodes having the following features:

1. 2 sockets of 16 physical cores each, of the type IBM POWER9 AC922, with a processor speed of 3.1 GHz. Each physical core corresponds to 4 virtual cores, with a total of 128 ($2 \times 16 \times 4$) virtual cores per node;
2. 4 GPUs of the type NVIDIA Volta V100, with a memory of 16 GB each, connected with Nvlink 2.0;
3. 256 GB of RAM.

As shown by different runs, such as the performance tests illustrated in Cesare et al. (2022c), the MPI + OpenMP code runs in its optimal configuration when parallelized on 16 MPI processes per node and 2 OpenMP threads per MPI process. For a typical run for the production occupying a memory of 340 GB, parallelized on 2 nodes on 16 MPI processes + 2 OpenMP threads per node, and with a number of observations and of stars equal to $N_{\text{obs}} = 1.8 \times 10^9$ and $N_{\text{stars}} = 8.4 \times 10^6$, respectively, we achieve a convergence after $\sim 141,000$ iterations with an iteration time of ~ 4.23 s, which results in a total elapsed time of $t_{e,\text{OMP}} \simeq 166$ hours, namely about one week. However, this elapsed time is obtained for a system having a number of observations ~ 2 orders of magnitude smaller than the number of observations expected for the final Gaia data set ($\sim 10^{11}$). When we will have to deal with such a large data set, that will occupy $\sim 10\text{--}100$ TB of memory, the time-to-solution would become $\sim 30\text{--}300$ times larger, which will result in a far from optimal production. To manage these data sizes, a properly accelerated code is needed. For this purpose, we compared the performance of the MPI + CUDA and of the MPI + OpenMP codes on M100 for a set of systems with increasing size, measuring the acceleration factor of the CUDA code over the OpenMP code to verify whether the CUDA code was worth to be put in production.

We ran the OpenMP and the CUDA applications for different input data sets provided by the Data Processing Center of Turin (DPCT), which is supervised by the Aerospace Logistics Technology Engineering Company (ALTEC) in collaboration with the Astrophysics Observatory of Turin (INAF-OATO). These inputs are real Gaia data sets and they are employed for the production of the OpenMP code. The data sets have different sizes, occupying a memory of 40, 100, 300, and 350 GB, and each of them only computes some sections of the complete model. The 40 GB and 300 GB systems solve the attitude and the instrumental parts, the 100 GB system solves the astrometric part, and the 350 GB system solves the astrometric, the attitude, and the instrumental parts. As anticipated in Section 4, no system solves the global part.

Figure 3 shows the ratio between the average times of one LSQR iteration of the OpenMP and the CUDA codes, as a function of the system size. We ran the OpenMP and the CUDA codes in their optimal configurations (16 MPI processes + 2 OpenMP threads per node, for the OpenMP code, and 4 MPI processes per node for the CUDA code, see Section 4.1).

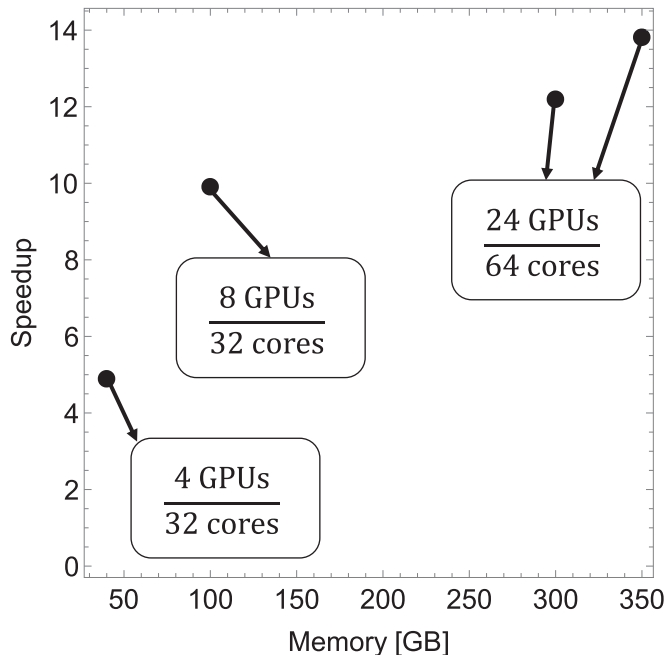


Figure 3. Speedup of the CUDA code over the OpenMP code as a function of the memory occupied by the system. For every point, the number of GPUs employed by the CUDA code and the number of physical cores employed by the OpenMP code is indicated.

We report below each point in Figure 3 the number of GPUs employed by the CUDA code, coincident with the number of MPI tasks, and of physical cores employed by the OpenMP code. Each code runs on the minimum number of nodes needed. For the same amount of memory, the CUDA code might require more nodes than the OpenMP code, since the memory of the four GPUs in each node is smaller than the RAM memory of the node (64 GB versus 256 GB). For example, the 100 GB system is parallelized on 8 GPUs, i.e., 2 nodes, for the CUDA code, and on 32 cores, i.e., 1 node, for the OpenMP code.

The speedup of the CUDA code over the OpenMP code increases with both the system size and the number of employed GPU resources. From the first to the second point, correspondent to the 40 GB and the 100 GB systems, the speedup doubles, passing from ~ 5 to ~ 10 . This might be explained by the fact that whereas the OpenMP code is always parallelized on the same amount of resources, in the CUDA code the number of resources doubles in the second run compared to the first run. Instead, considering the last two points, correspondent to the 300 GB and the 350 GB systems, the speedup does not substantially increase, passing from ~ 12 to ~ 14 . In this case, the two systems are always parallelized on the same amount of resources. The slightly increase of the speedup might be justified by an increase of the GPU occupancy in the 350 GB system compared to the 300 GB

system. In the 300 GB system, the memory assigned per MPI process, and thus per GPU, is of ~ 9.5 GB, which implies a GPU occupancy of $\sim 60\%$. Instead, in the 350 GB system, the memory assigned per MPI process is of ~ 13 GB, which implies a GPU occupancy of $\sim 80\%$.

Cesare et al. (2022c) show in Figures 4(b) and 5(a) a strong scaling test for the OpenMP code up to 16 nodes: the strong scaling curve already departs from the ideal linear speedup, tending to a plateau, after ~ 3 nodes (96 physical cores). This means that, for a fixed amount of memory, the performance of the OpenMP code does not substantially improve if we continue to increase the number of physical cores on which it runs. The same figures show that the OpenACC strong scaling behavior is similar to the OpenMP one and that the ratio between the OpenMP and OpenACC mean iteration times is nearly constant and around 1.4. Given that in the CUDA code, as in the OpenACC code, the MPI tasks are assigned to the GPUs of the node in a round-robin fashion, we expect the strong scaling curve to be also similar for the CUDA code. Furthermore, the CUDA code is much more performant than the OpenACC code and, thus, we expect it to accelerate over the OpenMP code even if the latter is run on a larger amount of physical cores.

The maximum speedup of $\sim 14\times$ is obtained for the 350 GB system. With this speedup, the CUDA code is more than $9\times$ faster than the OpenACC code. Given the trend observed in Figure 3, we expect the speedup to continue to increase for systems of larger sizes. This is a remarkable result in perspective of the final data set of Gaia which makes this implementation of the CUDA code a good candidate to be put in production. However, before proceeding, we performed a further test, detailed in the following section, to verify whether the rearrangement of the code required for the CUDA parallelization had impaired the correctness of the application.

6. Numerical Stability

To check if the CUDA parallelization was correctly implemented, we compared the solutions of the systems of equations considered in the previous section and the errors on the solutions, obtained with the OpenMP and the CUDA codes. Figure 4(a) plots the solution of the astrometric section of the 350 GB system found with the CUDA code against the solution of the same system found with the OpenMP code. Figure 4(b) shows the same for the errors on the solutions. The one-to-one relation (black dashed line) is shown as a reference. We do not illustrate the analogous plots for the other sections of the same system and for the other systems, since they show equivalent outputs.

The scatter plots in Figure 4 show that the CUDA and the OpenMP solutions and errors tightly distribute along the one-to-one relation, which suggests an agreement between the two couples of quantities. However, the figures only show a

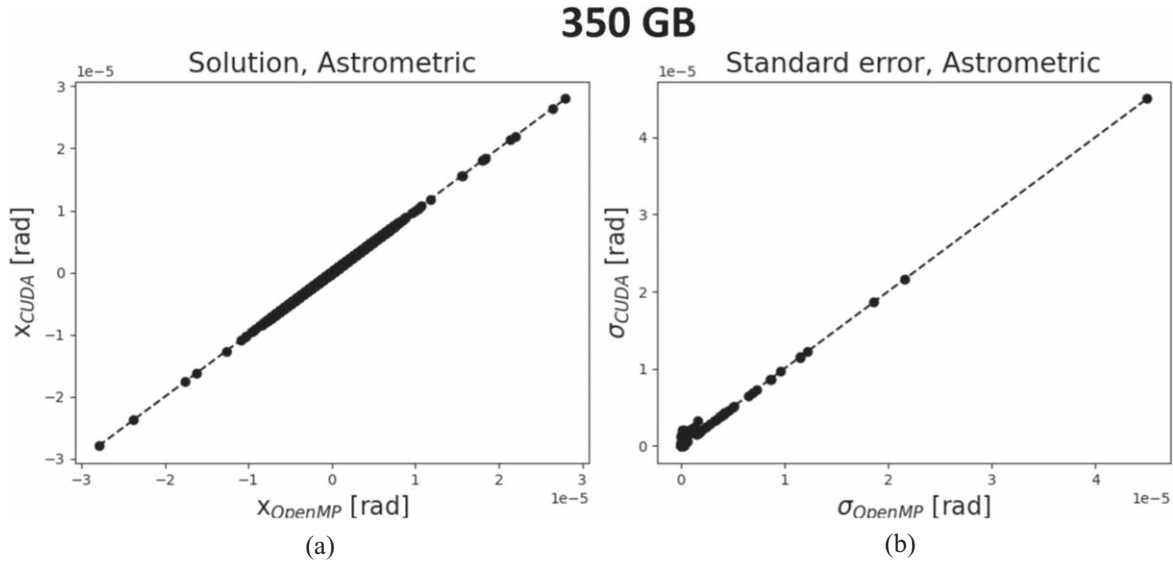


Figure 4. Solution (Figure 4(a)) of the astrometric section of the 350 GB system and its error (Figure 4(b)) computed with the CUDA code against the solution and the error of the same system computed with the OpenMP code. The one-to-one relation is shown as a black dashed line, for comparison.

Table 1
Comparison between the CUDA and OpenMP solutions and errors on the solutions for the four considered systems of equations

Memory (GB)	Section	Mean(d_x) \pm StDev(d_x) (rad)	Mean(d_σ) \pm StDev(d_σ) (rad)
(1)	(2)	(3)	(4)
40	Attitude	$3.1 \times 10^{-20} \pm 7.4 \times 10^{-17}$	$-3.2 \times 10^{-10} \pm 1.5 \times 10^{-8}$
	Instrumental	$7.3 \times 10^{-23} \pm 4.1 \times 10^{-20}$	$-4.5 \times 10^{-11} \pm 1.5 \times 10^{-10}$
100	Astrometric	$-1.5 \times 10^{-20} \pm 4.7 \times 10^{-17}$	$3.1 \times 10^{-14} \pm 2.2 \times 10^{-12}$
300	Attitude	$1.7 \times 10^{-21} \pm 3.7 \times 10^{-17}$	$6.9 \times 10^{-11} \pm 1.2 \times 10^{-8}$
	Instrumental	$-7.6 \times 10^{-23} \pm 1.5 \times 10^{-20}$	$-1.9 \times 10^{-12} \pm 6.4 \times 10^{-12}$
350	Astrometric	$-3.9 \times 10^{-22} \pm 2.2 \times 10^{-17}$	$-4.0 \times 10^{-13} \pm 8.1 \times 10^{-10}$
	Attitude	$1.2 \times 10^{-21} \pm 1.4 \times 10^{-21}$	$-7.2 \times 10^{-13} \pm 5.7 \times 10^{-11}$
	Instrumental	$3.1 \times 10^{-23} \pm 3.4 \times 10^{-20}$	$1.2 \times 10^{-13} \pm 7.4 \times 10^{-13}$

Note. Column 1: Memory occupied by the system of equations; column 2: section solved for the considered system; column 3: mean and standard deviation of the differences between the solutions of the systems of equations found from the CUDA and the OpenMP codes; column 4: mean and standard deviation of the differences between the errors on the solutions found from the CUDA and the OpenMP codes. The quantities d_x and d_σ refer to the differences between a CUDA and an OpenMP quantity.

qualitative result, deduced by a visual inspection. To better quantify the consistency between the solutions and the errors found from the two codes, we calculate the average and the standard deviation of their differences, as reported in Table 1. Table 1 also reports the same quantities for the attitude and instrumental sections of the 350 GB system and for the other systems. We can see that the average differences, both for the

solutions (d_x) and for the errors (d_σ), are very close to 0, spanning a range, in absolute value, from 3.1×10^{-23} rad¹² to 1.5×10^{-20} rad, for the solutions, and from 3.1×10^{-14} rad to

¹² To simplify, from this point on we always write “rad” and “arcsec” and we do not distinguish between “rad” and “rad yr⁻¹” and between “arcsec” and “arcsec yr⁻¹.”

3.2×10^{-10} rad, for the standard errors. The standard deviations of the differences are, in every case, larger than the averages, which implies the agreement of the differences with zero. Moreover, the average differences, for both the solutions and the errors, are sometimes positive and sometimes negative, which suggests the absence of systematic errors.

To better evaluate the agreement between the CUDA and the OpenMP solutions of every system, we also compared their differences with their errors. For each solution and error point, we computed the ratio:

$$q = \frac{|x_{i,\text{CUDA}} - x_{i,\text{OpenMP}}|}{\sqrt{\sigma_{i,\text{CUDA}}^2 + \sigma_{i,\text{OpenMP}}^2}}, \quad (5)$$

where $x_{i,\text{CUDA}}$ and $x_{i,\text{OpenMP}}$ are the solution points found from the CUDA and the OpenMP codes, and $\sigma_{i,\text{CUDA}}$ and $\sigma_{i,\text{OpenMP}}$ are their errors. For every section of all the systems, the ratio q is smaller than 1, which means that the solutions found from the two codes are always consistent within 1σ . These results show that the CUDA and the OpenMP solutions and errors are in agreement with each other for systems of increasing size and prove the numerical stability of the CUDA code.

Besides checking the consistency between the results obtained with the two codes, we also wanted to verify if the solutions were obtained with the accuracy required by the Gaia mission ($\sim[10, 100] \mu\text{arcsec}$ for the parallaxes and the positions and $\sim[10, 100] \mu\text{arcsec yr}^{-1}$ for the proper motions, see Section 1) to achieve a high precision astrometry, in order to properly investigate, e.g., the kinematics and the dynamics of the Galaxy. We converted the uncertainties on the solutions (σ) from radians to arcseconds with the relation:

$$\sigma(\text{arcsec}) = \frac{\sigma(\text{rad})}{4.84814 \times 10^{-6}}. \quad (6)$$

In the 100 GB and 350 GB runs, which compute the astrometric section of the system, the average uncertainties on the astrometric parameters along with their standard deviations are of $\sigma = (4.0 \times 10^{-5} \pm 5.5 \times 10^{-3})$ arcsec and $\sigma = (2.1 \times 10^{-5} \pm 2.1 \times 10^{-4})$ arcsec, both for the OpenMP and the CUDA codes, in agreement with the needed precision. For the 100 GB run, nearly 80% of the astrometric solution points have uncertainties below $10 \mu\text{arcsec}$, and more than 97% and 99% of the astrometric solution points have uncertainties below $100 \mu\text{arcsec}$ and $500 \mu\text{arcsec}$. For the astrometric part of the 350 GB run, the $\sim 99\%$ of the solution points already have uncertainties below $100 \mu\text{arcsec}$. Also the attitude and instrumental parameters are generally obtained with a compatible accuracy.

Given these results, we put the CUDA code in production in Q2 2022. The CUDA solver was also put on a proprietary GitLab repository of CINECA and its copyright is held by INAF.

7. Conclusions and Future Works

We ported to a GPU environment with the CUDA programming language the AVU–GSR parallel solver, developed for the ESA Gaia mission and originally parallelized on the CPU with a hybrid MPI + OpenMP model. The code solves a system of linear equations with the iterative LSQR algorithm to find the astrometric parameters of $\sim 10^8$ stars in the Milky Way, the attitude and the instrumental settings of the Gaia spacecraft, and the global parameter γ of the PPN formalism. To iteratively find the solution up to the convergence of the algorithm defined in the least square sense, the LSQR calls, at each step, the *aprod* function in its modes 1 and 2, which provide an iterative estimate for the known terms and the solution arrays, respectively.

The porting presented in this paper is the result of an optimization of a previous GPU porting of this application, performed with the high-level language OpenACC. The OpenACC code showed a moderate speedup of $\sim 1.5\times$ over the OpenMP code. As already pointed out at the beginning of Section 4, further speedups might as well have been obtained with a better optimization of the usage of the OpenACC language. Indeed, the speedup of $\sim 1.5\times$ refers to a quite basic parallelization with OpenACC, where the OpenMP directives were basically replaced by the OpenACC correspondent ones. However, we preferred to adopt the low-level language CUDA for the new porting since it allows to better match the architecture of the device and, thus, to possibly achieve larger performances. On the other hand, this reduces the code portability, since the CUDA parallelization is architecture-dependent. However, since the Gaia mission is expected to end in the following years and only a further porting of this code on Leonardo supercomputer is expected, we aimed to improve the performance rather than to obtain a larger code portability.

With the CUDA porting, we reorganized the structure of the Gaia AVU–GSR solver, by defining the kernels to parallelize different regions of the code, such as the *aprod 1* and *2* functions. In each of the kernels, we manually defined the hierarchy of the grid of threads to match as better as possible the GPU architecture and the topology of the problem to solve. We also ported with CUDA other regions of code that in the OpenACC application were still running on the CPU and we reduced the H2D and D2H data copies with respect to the OpenACC code. With these optimizations, the time fraction of one LSQR iteration due to GPU computation rises from $\sim 70\%$ to $\sim 90\%$, and the time fractions due to CPU calculations and data transfers reduces from $\sim 15\%$ to $\sim 3\%$.

Running the CUDA and the OpenMP applications on M100, the CUDA code presents a speedup over the OpenMP code increasing with the system size and with the employed GPU resources. The speedup reaches a maximum of ~ 14 for a system occupying 350 GB of memory and is expected to increase for systems of larger sizes and by running the codes on next-

generation platforms with GPUs having more memory and streaming multiprocessors, such as the CINECA supercomputer Leonardo. Indeed, the A200 GPUs of Leonardo have $4\times$ more memory and more streaming multiprocessors than the V100 GPUs of M100, which allows to execute more concurrent threads. Since both M100 and Leonardo have 4 GPUs per node, the GPU memory per node on Leonardo is quadrupled with respect to M100. We plan to perform the first tests of the AVU–GSR code on Leonardo in the first half of 2023.

The CUDA code showed great numerical stability, since it provided solutions and uncertainties on the solutions fully consistent within 1σ with the correspondent ones found with the OpenMP code for a set of systems. Moreover, the solutions are obtained with the accuracy of $[10, 100] \mu\text{arcsec}$, as required by the Gaia mission. Given these results, the MPI + CUDA AVU–GSR solver was put in production on M100. This is a fundamental achievement since it provides an optimal production for the AVU–GSR pipeline, allowing to obtain important data for scientific analyses, such as the study of the Milky Way formation and evolution, in reduced timescales.

The increasing trend of the speedup with the system size is a very important result toward the scientific purposes of the upcoming Data Releases of the Gaia mission, from which TBs of data will be produced up to an expected final data set of $\sim 10\text{--}100$ TB. In perspective of these pre-Exascale data products, we will continue the optimization process of the AVU–GSR code, for example by porting to the GPU further sections of code, and the consequent investigation of the performance, scaling, and numerical stability of the code, for systems with an increasing size, up to the sizes expected for the final Gaia data set. These are some of the targets of a two-years project already underway and funded by INAF, an INAF Mini Grant, of which the author VC is the PI and which is performed in collaboration with Prof. Marco Aldinucci of the University of Turin. For this future analysis, we will use Leonardo, to better investigate the behavior of the AVU–GSR code on a next-generation pre-Exascale infrastructure and in perspective of a final porting of this code on Leonardo.

Besides allowing a better performance, this novel arrangement of the hardware, with hosts less performant than the accelerators and GPUs with a larger memory and number of streaming multiprocessors, such as on Leonardo, will imply a low energy consumption for the size of the problems that will need to be computed by HPC GPU-oriented applications. When a code such as the AVU–GSR solver is ported to the GPU resulting in a $\gtrsim 14\times$ speedup over the CPU version, besides obtaining results in a minor time, we also expect to save a substantial amount of energy. This is not obvious, since H2D and D2H data transfers, not present in the CPU application, might be rather energy consuming, but this might be compensated by the high speedup. In a future work, we aim to compare the energy consumption of the CUDA and the OpenMP codes by running systems of increasing size, up to

$\sim 10\text{--}100$ TB, both on M100 and on Leonardo, to verify whether the CUDA code is in fact “greener” than the OpenMP code and whether running on Leonardo allows to save even more energy than on M100. Since the GPU memory per node on Leonardo is $4\times$ the GPU memory per node on M100, a quarter of the resources could be required on Leonardo with respect to M100 to run a system of equal size. This might imply a minor energy consumption on Leonardo compared to M100. Also this analysis is a target of the Mini Grant project.

This research activity has important repercussions in the development, toward a (pre-)Exascale calculation, of other LSQR-based applications involving the solutions of systems with a high sparsity degree, similarly to the Gaia AVU–GSR solver. The parallelization techniques employed in this code could be adapted and exploited in different contexts that adopt the LSQR, such as the reconstruction of images in radioastronomy (Naghizadeh & van der Veen 2017), geophysics (Joulidehsar et al. 2018; Liang et al. 2019a, 2019b), geodesy (Baur & Austen 2005), medicine (Bin et al. 2020; Guo et al. 2021), and industry (Jaffri et al. 2020) (see Section 1). In conclusion, the continue development of efficient parallelization techniques is essential to face the increasingly faster production of data in contexts of different nature, going toward the Big Data era.

Acknowledgments

We sincerely thank the referee, whose comments largely improved and clarified the presentation of our results.

We sincerely thank Dr. Aswin Kumar of NVIDIA for his support in parallelizing this application with CUDA, and the organizers of the CINECA course “Programming paradigms for GPU devices,” held on 2021 June 9th–11th, for providing the material employed to learn the most important notions necessary to parallelize this code with CUDA. We also thank Dr. Massimiliano Guarasi of CINECA, for deepening some CUDA concepts.

This work has been supported by the Spoke 1 “FutureHPC & BigData” of the ICSC—Centro Nazionale di Ricerca in High Performance Computing, Big Data and Quantum Computing-and hosting entity, funded by European Union—Next GenerationEU.

This work was also supported by the Italian Space Agency (ASI) [grant No.: 2018-24-HH.0], in support of the Italian participation to the Gaia mission, and by Consorzio Interuniversitario Nazionale per l’Informatica, under the project EUPEX, EC H2020 RIA, EuroHPC-02-2020 [Grant Agreement: 101033975].

ORCID iDs

Valentina Cesare  <https://orcid.org/0000-0003-1119-4237>

Ugo Becciani  <https://orcid.org/0000-0002-4389-8688>

Alberto Vecchiato  <https://orcid.org/0000-0003-1399-5556>

Mario Gilberto Lattanzi  <https://orcid.org/0000-0003-0429-7748>

Marco Aldinucci  <https://orcid.org/0000-0001-8788-0829>

Beatrice Bucciarelli  <https://orcid.org/0000-0002-5303-0268>

References

- Aldinucci, M., Cesare, V., Colonnelli, I., et al. 2021, *JPDC*, 157, 13
- Baur, O., & Austen, G. 2005, in Proc. Joint CHAMP/GRACE Science Meeting (Potsdam: GeoForschungsZentrum)
- Becciani, U., Sciacca, E., Bandieramonte, M., et al. 2014, in Int. Conf. on High Performance Computing Simulation (HPCS) (Piscataway, NJ: IEEE), 104
- Bin, G., Wu, S., Shao, M., Zhou, Z., & Bin, G. 2020, *J. Electrocardiol.*, 62, 190
- Borriello, L., Dalessandro, F., Murgolo, F., & Prezioso, G. 1986, *MmSAI*, 57, 267
- Butkevich, A. G., Vecchiato, A., Bucciarelli, B., et al. 2022, *A&A*, 663, A71
- Carpenter, P., Utz, U.-H., Narasimhamurthy, S., & Suarez, E. 2022, Heterogeneous High Performance Computing, Zenodo, doi: [10.5281/zenodo.6090425](https://doi.org/10.5281/zenodo.6090425)
- Cesare, V., Becciani, U., Vecchiato, A., et al. 2021, in ASP Conf. Ser., Astronomical Data Analysis Software and Systems XXXI (San Francisco, CA: ASP), in press
- Cesare, V., Becciani, U., Vecchiato, A., et al. 2022a, *INAF Technical Reports* 163
- Cesare, V., Becciani, U., & Vecchiato, A. 2022b, *INAF Technical Reports* 164
- Cesare, V., Becciani, U., Vecchiato, A., et al. 2022c, *A&C*, 41, 100660
- Cesare, V., Colonnelli, I., & Aldinucci, M. 2020, in 28th Euromicro Int. Conf. on Parallel, Distributed and Network-Based Processing (PDP) (Piscataway, NJ: IEEE), 376
- Crosta, M., Giammaria, M., Lattanzi, M. G., & Poggio, E. 2020, *MNRAS*, 496, 2107
- Gaia Collaboration, Vallenari, A., Brown, A. G. A., et al. 2023, *A&A*, 674, A1
- Galassi, M., Davies, J., Theiler, J., et al. 2018, GNU Scientific Library Reference Manual, <https://www.gnu.org/software/gsl/>
- Giammaria, M., Spagna, A., Lattanzi, M. G., et al. 2021, *MNRAS*, 502, 2251
- Guo, H., Zhao, H., Yu, J., et al. 2021, *J. Biophotonics*, 14, e202100089
- Hees, A., Le Poncin-Lafitte, C., Hestroffer, D., & David, P. 2018, in IAU Symp. 330 (Cambridge: Cambridge Univ. Press), 63
- Jaffri, N. R., Shi, L., Abrar, U., Ahmad, A., & Yang, J. 2020, in Proc. 2020 5th Int. Conf. on Multimedia Systems and Signal Processing (New York: ACM), 16
- Joulidehsar, F., Moradzadeh, A., & Doulati Ardejani, F. 2018, *PapGe*, 175, 4389
- Krolikowski, D. M., Kraus, A. L., & Rizzuto, A. C. 2021, *AJ*, 162, 110
- Liang, S.-X., Jiao, Y.-J., Fan, W.-X., & Yang, B.-Z. 2019a, *PrGeo*, 34, 1475
- Liang, S.-X., Wang, Q., Jiao, Y.-J., Liao, G.-Z., & Jing, G. 2019b, *Geophysical and Geochemical Exploration*, 43, 359
- Lindgren, L., Lammers, U., & Hobbs, D. 2012, *A&A*, 538, A78
- Mignard, F., & Drimmel, R. 2007, DPAC: Proposal for the Gaia Data Processing, http://www.rssd.esa.int/doc_fetch.php?id=2720336
- Naghizadeh, S., & van der Veen, A.-J. 2017, in IEEE Int. Conf. on Acoustics, Speech and Signal Processing (ICASSP) (Piscataway, NJ: IEEE), 3385
- O'Mullane, W., Lammers, U., Lindgren, L., Hernandez, J., & Hobbs, D. 2011, *ExA*, 31, 215
- Paige, C. C., & Saunders, M. A. 1982a, *ACM Trans. Math. Softw. (TOMS)*, 8, 43
- Paige, C. C., & Saunders, M. A. 1982b, *ACM Trans. Math. Softw. (TOMS)*, 8, 195
- Van der Marel, H. 1988, PhD thesis, Delft Univ. Technology, Netherlands
- Vecchiato, A., Bucciarelli, B., Lattanzi, M. G., et al. 2018, *A&A*, 620, A40
- Vecchiato, A., Lattanzi, M. G., Bucciarelli, B., et al. 2003, *A&A*, 399, 337