Number	164
Publication Year	2022
Acceptance in OA@INAF	2022-07-05T14:56:33Z
Title	The MPI+CUDA Gaia AVU-GSR Parallel Solver in perspective of next-generation Exascale Infrastructures and new Green Computing milestones
Authors	CESARE, VALENTINA; BECCIANI, Ugo; VECCHIATO, Alberto
Affiliation of first author	O.A. Catania
Handle	http://hdl.handle.net/20.500.12386/32452; https://doi.org/10.20371/INAF/TechRep/164

Technical Report

The MPI+CUDA Gaia AVU-GSR Parallel Solver in perspective of next-generation Exascale Infrastructures and new Green Computing milestones

Cesare Valentina
Becciani Ugo
Vecchiato Alberto

Abstract

We ported on the GPU with CUDA the Gaia Astrometric Verification Unit-Global Sphere Reconstruction (AVU-GSR) Parallel Solver. The code aims to find the astrometric parameters of $\sim 10^8$ stars in the Milky Way, the attitude and the instrumental settings of the Gaia satellite, and the global parameter γ of the PPN formalism, by solving a system of linear equations, $\mathbf{A} \times \boldsymbol{x} = \boldsymbol{b}$, with the LSQR iterative algorithm. The coefficient matrix $\mathbf{A}$ is large, having $\sim 10^{11} \times 10^8$ elements, and sparse. The CUDA code accelerates $\gtrsim 14$ times compared to the current version of the AVU-GSR code, parallelized on the CPU with MPI+OpenMP and in production since 2014. This acceleration factor is ~9.2 times larger than the one obtained with a preliminary GPU porting with OpenACC, equal to ~1.5. We obtained this result by running the codes on the CINECA SuperComputer Marconi100, that has 4 NVIDIA Volta V100 GPUs per node, where the MPI+CUDA application has been recently put in production. This analysis represents a first step to understand the exascale behaviour of a class of applications that follow the same structure of this code, employed in several contexts. In the next months, we plan to run this code on the pre-exascale platform Leonardo of CINECA, with 4 next-generation A100 GPUs per node, to better investigate this behaviour. Computing on highly parallel devices, such as GPUs, might imply a consistent power saving, which might go towards the achievement of a Green Computing milestone.

Sommario

1. Introduction *4*

2. General structure of the application *5*

3. Previous parallelizations: MPI+OpenMP and MPI+OpenACC *6*

4. The CUDA porting of the Gaia AVU-GSR parallel solver *8*

4.1 Performance comparison with the MPI+OpenMP code **10**

4.2 Numerical stability of the AVU-GSR code **11**

5. Conclusions and future perspectives *12*

References and websites *14*

1. Introduction

In this era of technological development, the size of the data concerning several contexts, from astrophysics to medicine, from economics to industry, is rapidly growing up. Consequently, these data need increasingly efficient parallelization techniques and High Performance Computing (HPC) infrastructures to be processed in reasonable timescales. As a test case of application involving Big Data, in perspective of (pre-)exascale systems, we consider the Gaia Astrometric Verification Unit-Global Sphere Reconstruction (AVU-GSR) parallel solver. This code is developed for the European Space Agency (ESA) Gaia mission [1], under an agreement between Istituto Nazionale di Astrofisica (INAF) and CINECA and with the support of Agenzia Spaziale Italiana (ASI). The application is currently employed by the Coordination Unit 3 (CU3) of the Data Processing and Analysis Consortium (DPAC) for the AVU-GSR tasks. The complete AVU-GSR process is managed by the Data Processing Center of Turin (DPCT), which is in turn supervised by the Aerospace Logistics Technology Engineering Company (ALTEC) and by the Astrophysics Observatory of Turin of INAF (INAF-OATO).

The main purpose of this code is to find the positions and the proper motions of ~10^8 stars in our galaxy, with a micro-arcsecond precision, along with the attitude and instrumental specifications of the Gaia spacecraft and the global parameter γ of the Parametrized Post Newtonian (PPN) formalism. To accomplish this target, the code solves a system of linear equations:

$$\mathbf{A} \times \boldsymbol{x} = \boldsymbol{b}, \tag{1}$$

where (I) the coefficient matrix, $\mathbf{A}$, is large, containing ~10^{11}x10^8 elements, and sparse, following a peculiar sparsity scheme; (II) the solution array, $\boldsymbol{x}$, contains ~10^8 x10^1 elements; (III) the known terms array, $\boldsymbol{b}$, contains ~10^{11}x10^1 elements. The rows of the coefficient matrix contain the observations of the stellar parameters, where we have ~10^8 stars observed on average ~10^3 times each. To decrease the computation time, only the non-zero elements of $\mathbf{A}$ are considered for the calculations, reducing its number of elements from ~10^{11}x10^8 to ~10^{11}x10^1. In this way, we obtain a dense coefficient matrix $\mathbf{A}_\mathrm{d}$. The system is iteratively solved with a customized implementation of the LSQR algorithm [2,3].

The AVU-GSR code is written in C and C++ and a version of this application parallelized on the CPU, with a hybrid MPI+OpenMP paradigm, has been in production on the CINECA SuperComputer Marconi100 and on all the other Tier0 platforms of CINECA since 2014 [4]. A typical execution of this code, with ~1.8x10^9 observations and ~8.4x10^6 stars, lasts ~166 hours and this elapsed time is going to increase for larger datasets, like the one expected at the end of the Gaia mission (~10^{11} observations and ~10^8 stars). To accelerate this code, we performed a preliminary porting of it on a GPU environment, when the OpenMP language was replaced by OpenACC [5,6,7]: with this porting we achieved a speedup of ~1.5 (the execution taken as example passes in this way from ~166 hours to ~111 hours).

To further improve the performances of this application, we optimized this code and we replaced the OpenACC language with CUDA [8]. This resulted in a speedup $\gtrsim$14 compared to the CPU code, which implies that the typical execution passes from ~166 hours to ~12 hours. These results are not only important for the ESA Gaia mission but also to investigate the (pre-)exascale behaviour of a class of codes following a structure similar to the one of the AVU-GSR application and based on the LSQR algorithm, currently employed in several contexts (e.g. geophysics [9,10], medicine [11], tomography [12], industry [13], and astronomy [14,15]). Moreover, this substantial reduction of the computation time might result in a decrease of the power consumption due to the application, going towards a Green Computing policy.

The CUDA code has been recently put in production on the Marconi100 platform. This implied a strong increase of the ability of the AVU-GSR pipeline to produce solutions and tests in timescales never seen before, being the acceleration factor of the CUDA code compared to the MPI+OpenMP code equal to ~14. The CUDA code has to be put in a GitLab repository, under a license held by INAF, that explicitly states that the re-use and the copy of the code is strictly prohibited by third parties unless expressly authorized by the authors of the code. The code is proprietary and confidential and it is reserved for the Gaia International Collaboration.

2. General structure of the application

As already specified in Section 1, the system of equations is solved in an iterative way with the LSQR algorithm, which represents the bulk of the AVU-GSR code. Figure 1 represents the general structure of the AVU-GSR application:

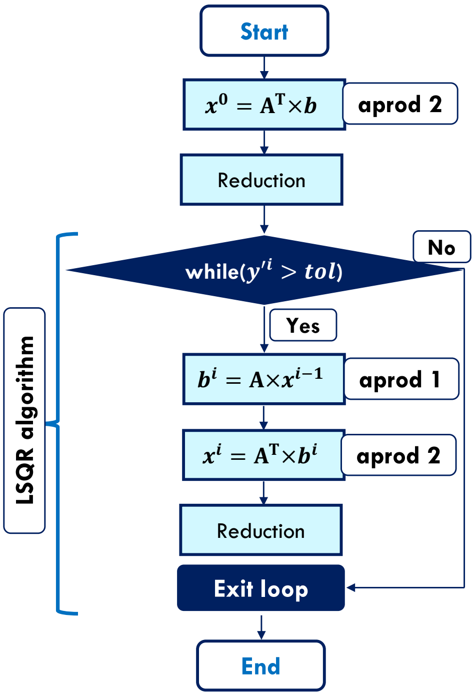


Figure 1: General structure of the Gaia AVU-GSR application.

The LSQR algorithm consists in a while loop, where the iterations stop when either a convergence condition or a maximum number of iterations set at runtime are reached. The convergence is considered achieved in the least-square sense, when the vector of the residuals $\boldsymbol{y}'^i = \boldsymbol{b}^i - \mathbf{A} \times \boldsymbol{x}^i$, given by the difference between the iterative estimate of the known terms array, $\boldsymbol{b}^i$, and the product between the coefficient matrix $\mathbf{A}$ and the iterative estimate of the solution, $\boldsymbol{x}^i$, goes below a pre-defined tolerance, tol, set to the machine precision. The solving of the system in the least-square sense is necessary for an overdetermined system like this, where the number of equations (i.e., the number of stellar observations) is much larger than the number of unknowns to solve.

At each step of the LSQR algorithm the iterative estimates of the known terms, $\boldsymbol{b}^i$, and of the solution, $\boldsymbol{x}^i$, are determined by the call of the aprod 1 and 2 functions, which perform the operations indicated in Figure 1. The aprod 2 function is also called before the starting of the LSQR iterations, to set the initial solution of the system. At the bottom of the coefficient matrix, an additional number of constraints equations are necessarily present to solve of this kind of overdetermined system. For further details about the general structure of the AVU-GSR code consult [4,6,7].

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

The AVU-GSR code is parallelized with MPI. Specifically, each MPI process deals with a subset of the total number of observations. The numbers of observations assigned to each MPI process are stored in a 1D single precision array, $\boldsymbol{N}$, whose index goes from 0 to the number of MPI processes set at runtime minus 1. Figure 2 illustrates the system of equations parallelized on 4 MPI processes in a single node of a computer cluster.

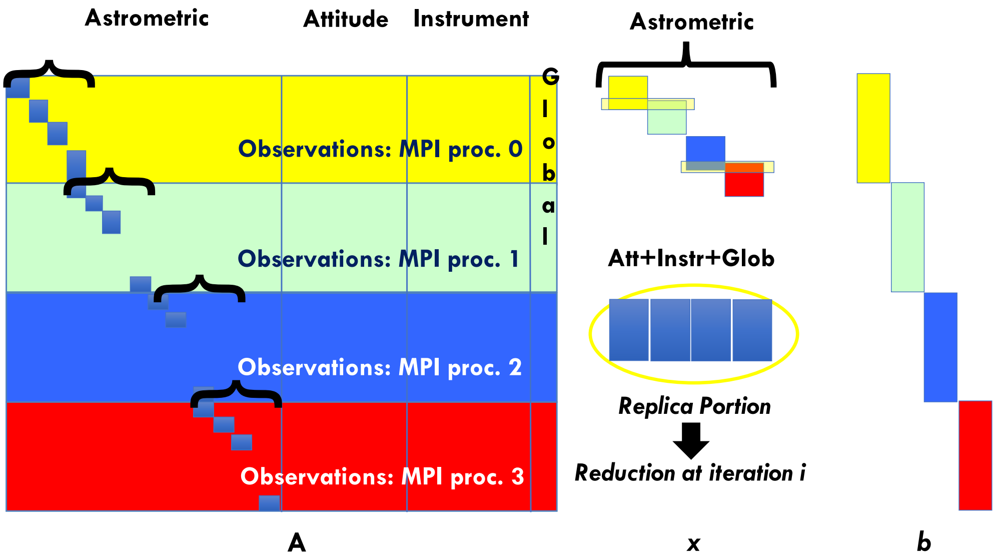


Figure 2: System of equations (Eq. (1)) parallelized on 4 MPI processes in one node of a computer cluster. Each coloured portion represents one MPI process.

Each coloured portion in Figure 2 represents one MPI process. The astrometric parameters in the coefficient matrix are organized in a block-diagonal structure. Given this regular pattern, the astrometric part of the solution array is distributed among the MPI processes. Instead, the other three parts, attitude, instrumental, and global, are replicated on the MPI processes. However, this does not imply a substantial slowdown of the code since these three sections only represent the ~10% of the total. At the end of each iteration, the solution computed in the different MPI processes is reduced in a unique value (see Figures 1 and 2). In the CPU code, the observations in each MPI process are further parallelized on the OpenMP threads.

In our first preliminary porting of the AVU-GSR code on a GPU environment, we substantially replaced the OpenMP directives with their OpenACC counterparts. This consisted in a simple exercise, to explore the feasibility of the porting of the AVU-GSR code on a GPU environment. In the MPI+OpenACC code, the MPI processes allocated on each node of a computer cluster are assigned to the GPUs of the node in a round-robin fashion, as illustrated in Figure 3.

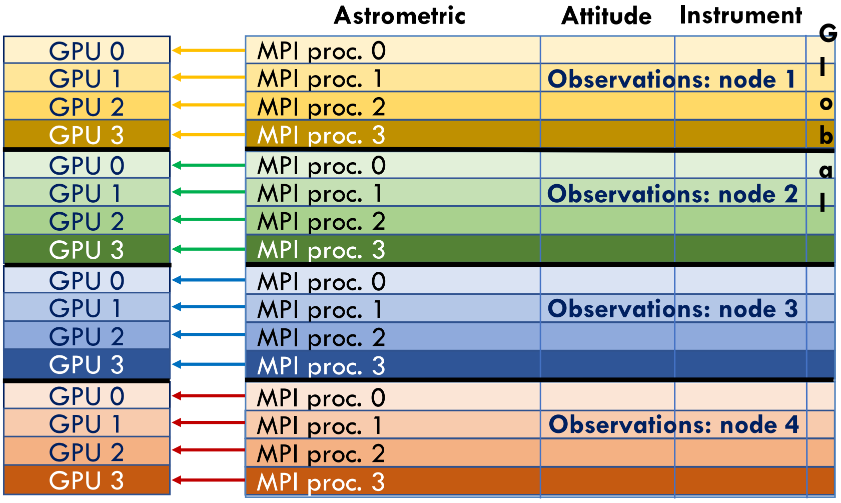


Figure 3: Parallelization scheme of the coefficient matrix on 4 nodes of a computer cluster with 4 MPI processes assigned per node and 4 GPUs per node.

In GPU programming, part of the execution time of the code is due to the data transfers between the host (CPU) and the device (GPU). Data transfers has to be managed with care because if they dominate over GPU kernels computation, the performance of the code might even worsen compared to the CPU version.

We tested the MPI+OpenACC code on the CINECA SuperComputer Marconi100 [16], that has 4 NVIDIA Volta V100 GPUs per node with 16 GB of memory each. The V100 GPUs on Marconi100 have 80 streaming multiprocessors (SMs) with a maximum number of threads allocable per SM equal to 2048. This means that a maximum number of threads equal to 80 x 2048 = 163840 can concurrently run on each GPU.

The top panel of Figure 4 shows the output of the NVIDIA Nsight Systems Profiler [17] for one iteration of a run of the MPI+OpenACC code parallelized on four MPI processes on one node of the Marconi100. The solved system occupies a memory of 50 GB.

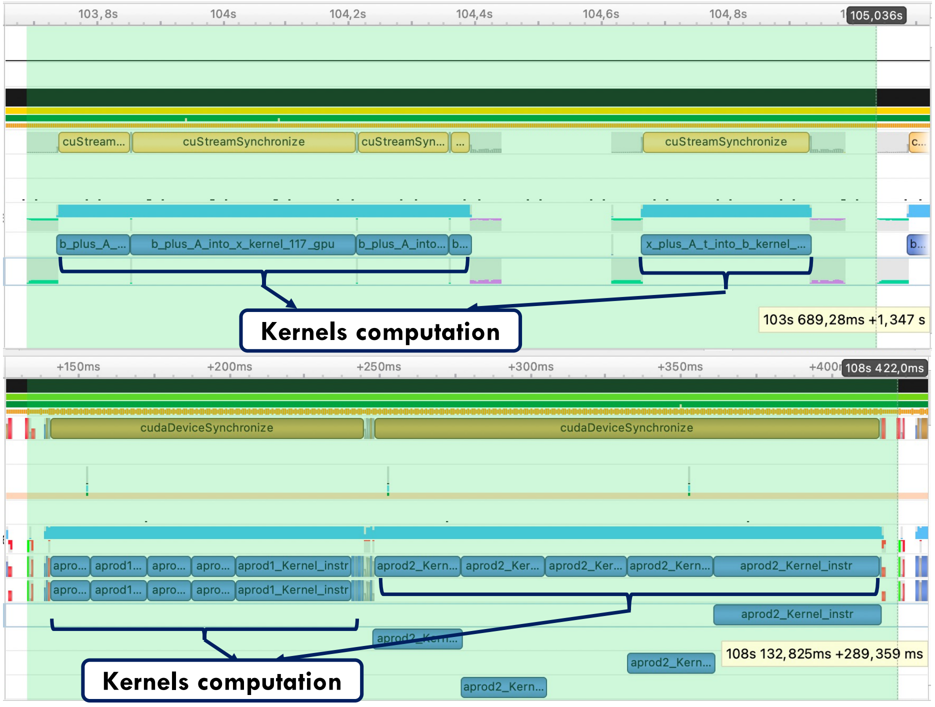


Figure 4: Output of the NVIDIA Nsight Systems Profile for one iteration of a run of the MPI+OpenACC (top panel) and MPI+CUDA (bottom panel) codes, parallelized on four MPI processes on one node of Marconi 100. The two runs occupy 50 GB of memory each.

The profiler illustrates the time dedicated to GPU kernels computation (blue), to host to device (H2D) and device to host (D2H) data copies (green and purple, respectively), and to CPU calculation (white). The time of one iteration (1.347 s) is superimposed to the figure. The kernels computation represents the 68.5% of the iteration time, whereas the data copies and the CPU computation only represents the 31.5%. This means that the code is *compute-bound*, even if further optimization is certainly possible. With this parallelization, the MPI+OpenACC code gains a factor of ~1.5 in performance compared to the MPI+OpenMP code. Going more in detail, the aprod 2 region accelerates of ~3.6 times, whereas the aprod 1 region looses in performance, of a factor of ~0.8 [6,7].

4. The CUDA porting of the Gaia AVU-GSR parallel solver

Our main purpose was to optimize the parallelization of the AVU-GSR solver starting from our preliminary porting to obtain a higher gain in performance. To accomplish this task we decided to pass from a high-level parallelization model (with OpenACC directives) to a low-level parallelization model (with CUDA) [8]. This implied a re-engineering of the great part of the code but in this way it was possible to manually define the GPU threads hierarchy to match the GPU hardware and the topology of the problem to solve, which entailed a performance boost.

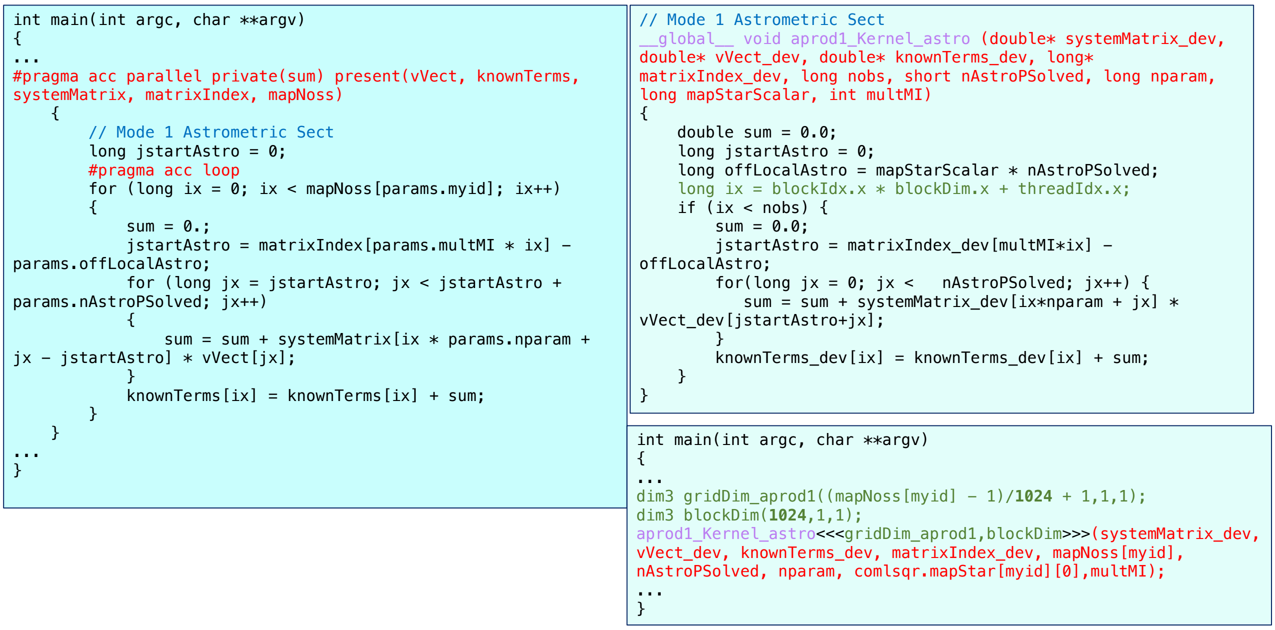
Figures 5 and 6 represent the astrometric section of the aprod 1 and 2 functions parallelized on the GPU with OpenACC (left panel) and with CUDA (right panel). The OpenACC code is substantially identical to the original OpenMP code, with the only difference that the parallelization directives refer to the OpenACC paradigm rather than to the OpenMP paradigm. This is what is intended for "high-level" parallelization language. Instead, the structure of the CUDA code completely change. The CUDA kernel is defined outside the main scope. The GPU threads topology is defined through the system variables "`blockIdx.x`" (index of the block in the grid of blocks of threads), "`blockDim.x`" (number of blocks of the grid), and "`threadIdx.x`" (local index of the thread inside each block). The global index of the thread inside the grid of blocks is given by the combination "`ix = blockIdx.x`

`* blockDim.x + threadIdx.x`". The "`.x`" specification indicates that the grid is 1D-dimensional and defined along the "x" direction. With these specifications, we can manually define the hierarchy of the grid of threads to directly match the topology of the problem to obtain better performances. This is what is intended for "low-level" parallelization language. It is important that the global index `ix` does not exceed the dimension of the array that has to be computed by the kernel, which is controlled by the "`if`" instruction.

The GPU kernel is called in the main scope of the program with the "<<<>>>" instruction. The first argument inside the "<<<>>>" instruction indicates the number of blocks in the grid of threads, whereas the second argument indicates the number of threads in each block, where 1024 blocks is an optimal solution for a V100 GPU, present on Marconi100. In the call of the aprod 2 astrometric kernel, two additional arguments are present within the "<<<>>>" instruction: the third one indicates the amount of shared memory allocated on the GPU (0 bytes in this case) and the fourth one indicates the number of the "stream" for asynchronous computation among different GPU kernels. Indeed, as we can also see from the bottom panel of Figure 4, the GPU kernels of the aprod 2 function run asynchronously with one another. For reasons of dependencies, this is not possible for the aprod 1 function.

In the different GPU kernels the quantities "`systemMatrix`", "`vVect`", "`knownTerms`", and "`mapNoss`" refer to "$\mathbf{A}_\mathrm{d}$", "$\boldsymbol{x}$", "$\boldsymbol{b}$", and "$\boldsymbol{N}$".

We note that, in the aprod 2 kernel, the `#pragma acc atomic` directive in the OpenACC code and the `atomicAdd()` instruction in the CUDA code are necessary to prevent different GPU threads to concurrently write the same elements of the unknowns array $\boldsymbol{x}$.

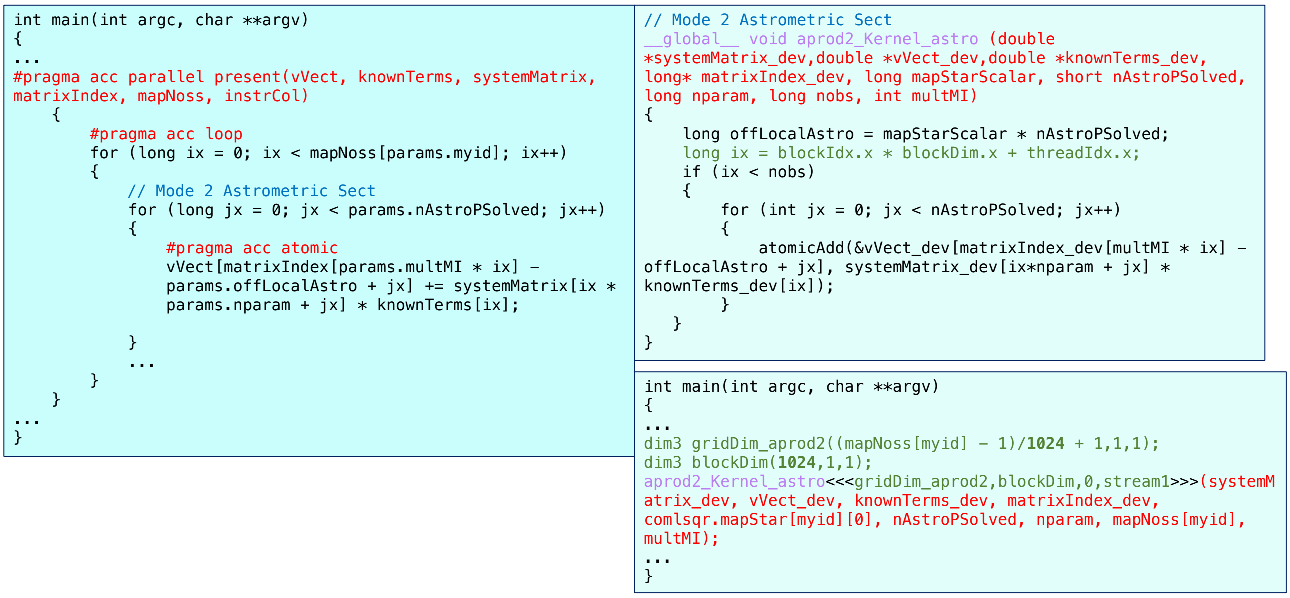


```
int main(int argc, char **argv)
{
...
#pragma acc parallel private(sum) present(vVect, knownTerms,
systemMatrix, matrixIndex, mapNoss)
    {
        // Mode 1 Astrometric Sect
        long jstartAstro = 0;
        #pragma acc loop
        for (long ix = 0; ix < mapNoss[params.myid]; ix++)
        {
            sum = 0.;
            jstartAstro = matrixIndex[params.multMI * ix] -
params.offLocalAstro;
            for (long jx = jstartAstro; jx < jstartAstro +
params.nAstroPSolved; jx++)
            {
                sum = sum + systemMatrix[ix * params.nparam +
jx - jstartAstro] * vVect[jx];
            }
            knownTerms[ix] = knownTerms[ix] + sum;
        }
    }
...
}
```

```
// Mode 1 Astrometric Sect
__global__ void aprod1_Kernel_astro (double* systemMatrix_dev,
double* vVect_dev, double* knownTerms_dev, long*
matrixIndex_dev, long nobs, short nAstroPSolved, long nparam,
long mapStarScalar, int multMI)
{
    double sum = 0.0;
    long jstartAstro = 0;
    long offLocalAstro = mapStarScalar * nAstroPSolved;
    long ix = blockIdx.x * blockDim.x + threadIdx.x;
    if (ix < nobs) {
        sum = 0.0;
        jstartAstro = matrixIndex_dev[multMI*ix] -
offLocalAstro;
        for(long jx = 0; jx <   nAstroPSolved; jx++) {
           sum = sum + systemMatrix_dev[ix*nparam + jx] *
vVect_dev[jstartAstro+jx];
        }
        knownTerms_dev[ix] = knownTerms_dev[ix] + sum;
    }
}
```

```
int main(int argc, char **argv)
{
...
dim3 gridDim_aprod1((mapNoss[myid] - 1)/1024 + 1,1,1);
dim3 blockDim(1024,1,1);
aprod1_Kernel_astro<<<gridDim_aprod1,blockDim>>>(systemMatrix_dev,
vVect_dev, knownTerms_dev, matrixIndex_dev, mapNoss[myid],
nAstroPSolved, nparam, comlsqr.mapStar[myid][0],multMI);
...
}
```

Figure 5: Astrometric part of the aprod 1 function parallelized with OpenACC (left panel) and CUDA (right panel).



```
int main(int argc, char **argv)
{
...
#pragma acc parallel present(vVect, knownTerms, systemMatrix,
matrixIndex, mapNoss, instrCol)
    {
        #pragma acc loop
        for (long ix = 0; ix < mapNoss[params.myid]; ix++)
        {
            // Mode 2 Astrometric Sect
            for (long jx = 0; jx < params.nAstroPSolved; jx++)
            {
                #pragma acc atomic
                vVect[matrixIndex[params.multMI * ix] -
                params.offLocalAstro + jx] += systemMatrix[ix *
                params.nparam + jx] * knownTerms[ix];

            }
            ...
        }
    }
...
}
```

```
// Mode 2 Astrometric Sect
__global__ void aprod2_Kernel_astro (double
*systemMatrix_dev,double *vVect_dev,double *knownTerms_dev,
long* matrixIndex_dev, long mapStarScalar, short nAstroPSolved,
long nparam, long nobs, int multMI)
{
    long offLocalAstro = mapStarScalar * nAstroPSolved;
    long ix = blockIdx.x * blockDim.x + threadIdx.x;
    if (ix < nobs)
    {
        for (int jx = 0; jx < nAstroPSolved; jx++)
        {
            atomicAdd(&vVect_dev[matrixIndex_dev[multMI * ix] -
offLocalAstro + jx], systemMatrix_dev[ix*nparam + jx] *
knownTerms_dev[ix]);
        }
    }
}
```

```
int main(int argc, char **argv)
{
...
dim3 gridDim_aprod2((mapNoss[myid] - 1)/1024 + 1,1,1);
dim3 blockDim(1024,1,1);
aprod2_Kernel_astro<<<gridDim_aprod2,blockDim,0,stream1>>>(systemM
atrix_dev, vVect_dev, knownTerms_dev, matrixIndex_dev,
comlsqr.mapStar[myid][0], nAstroPSolved, nparam, mapNoss[myid],
multMI);
...
}
```

Figure 6: Astrometric part of the aprod 2 function parallelized with OpenACC (left panel) and CUDA (right panel).

We also tested the MPI+CUDA code on Marconi100, to compare the behaviour of the MPI+OpenACC and of the MPI+CUDA codes on the same platform. The bottom panel of Figure 4 shows the output of the profiler for the MPI+CUDA code, correspondent to the one for the MPI+OpenACC code, illustrated in the top panel of the same figure. With the CUDA approach, we obtained a gain factor of ~6.4, for the aprod 1 kernels region, and of ~1.6, for the aprod 2 kernels region, with respect to the OpenACC code. We can see that the majority of the gain was due to the aprod 1 part that in the OpenACC code was slower than in the OpenMP code.

To continue the optimization, we analyzed the output of the profiler for the MPI+OpenACC code, to locate the data-transfers and the CPU regions, which we aimed to reduce as much as possible. From the bottom panel of Figure 4, we can see that these regions almost disappeared. Indeed, we took care to transfer only the array portions that needed to be moved and we ported on the GPU more functions that in the MPI+OpenMP code were not ported. With this optimization, the fraction of the average time of one LSQR iteration due to kernels computation increases from 68.5% to 93.5% and the fraction of the average time of one LSQR iteration due to data copies and CPU calculation reduces from 31.5% to 6.46%, indicating that the GPU hardware is better exploited than in the MPI+OpenACC code.

4.1 Performance comparison with the MPI+OpenMP code

To compare the performances of the MPI+CUDA and of the MPI+OpenMP codes, we ran the two applications on the Marconi100 platform, for different input datasets having different sizes. These inputs were provided by ALTEC and consisted in real datasets, employed for the in-production CPU version of the AVU-GSR code. Specifically, we ran systems occupying 29 GB, 97 GB, 229 GB, and 303 GB of memory. Figure 7 illustrates the ratio between the average times of one iteration of the CUDA and the OpenMP codes, as a function of the system size.

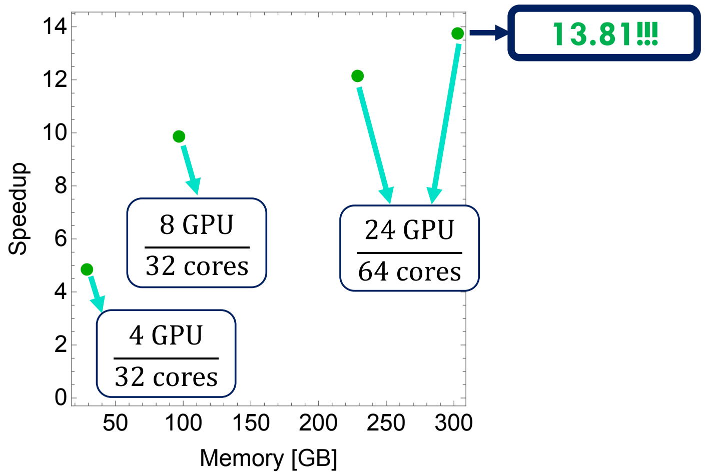


Figure 7: Speedup of the CUDA code over the OpenMP code as a function of the memory occupied by the system. For every point, it is indicated the number of GPUs employed by the CUDA code and the number of physical cores employed by the OpenMP code. For the last point, the speedup is explicitly highlighted.

We can see from Figure 7 that the speedup increases with the system size and that for a system that occupies 303 GB of memory the speedup is ~14. This trend indicates that the speedup might further increase for larger systems.

For every point in Figure 7, the number of GPUs employed by the CUDA code and the number of physical cores employed by the OpenMP code is indicated. For reasons of optimization, we always run the CPU code with 16 MPI processes per node and 2 OpenMP threads per MPI process, and the CUDA code with 4 MPI processes per node [5,6,7,8]. Running on 4 MPI processes per node on a platform with 4 GPUs per node is optimal for this kind of CUDA code since all the GPUs of the node are employed, only allocating one MPI process per GPU [6,7,8]. Since the RAM memory of each node is of 256 GB whereas the total memory of the GPUs in each node is of 64 GB (4 x 16 GB), more resources are required by the CUDA code compared to the OpenMP code to compute a system of equal size. Each node of Marconi100 has 32 physical cores per node. Below the second point, correspondent to a system that occupies 97 GB of memory, we can see that the CPU code ran on 32 cores, namely on one node, whereas the GPU code ran on 8 GPUs, namely on 2 nodes. This is because a system of 97 GB that runs on the GPUs cannot be allocated on a single node, where the GPU total available memory is of 64 GB, whereas the same system running on the CPU can be allocated on a single node, that has a RAM of 256 GB.

Looking at the first and at the second point of Figure 7, correspondent to a system of 29 GB and 97 GB, respectively, we can see that the CPU code runs on the same number of resources, whereas the CUDA code doubles the number of resources. Consequently, the speedup of the CUDA code over the OpenMP code increases of a factor of ~2, passing from ~5 to ~10.

Instead, looking at the third and the fourth point of Figure 7, correspondent to a system of 229 GB and 303 GB, the speedup slightly increases, passing from ~12.21 to ~13.81. The two codes are ran on the same amount of resources but the GPU occupancy is better exploited for the 303 GB system with respect to the 229 GB system (78.9% vs 59.6%), which might justify this slight performance improvement of the 303 GB system over the 229 GB system.

4.2 Numerical stability of the AVU-GSR code

We also tested the numerical stability of the AVU-GSR code, by comparing the solutions, along with their uncertainties, of a set of systems solved with the OpenMP and the CUDA codes. Specifically, we considered the same systems described in Section 4.1. The left panel of Figure 8 compares the solutions found with the CUDA and the OpenMP codes for the astrometric part of a system that occupies 303 GB of memory. The right panel of Figure 8 illustrates the correspondent comparison for the uncertainties on the solutions.

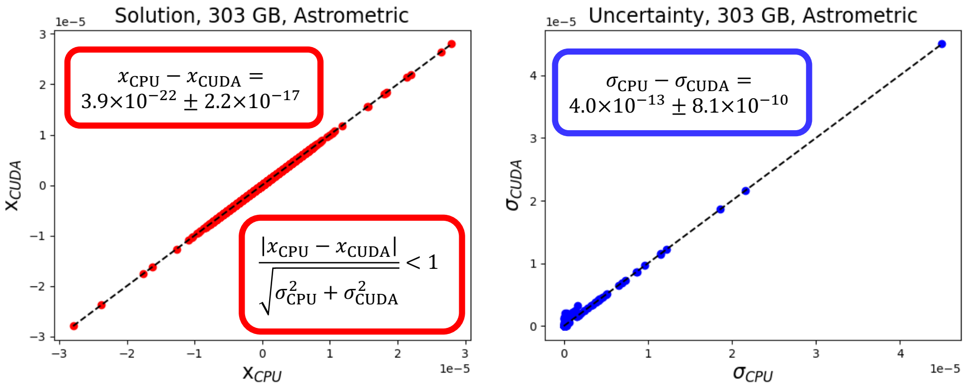


Figure 8: Comparison between the astrometric solutions (left panel) and their uncertainties (right panel) found with the CUDA and the OpenMP codes for a system of 303 GB of memory.

The CUDA and the OpenMP solutions are consistent with each other within 1σ, for each value of the unknowns. The mean and the standard deviation of the differences of each couple of OpenMP- CUDA unknowns and uncertainties on the unknowns are of $d_x = x_{\text{CPU}} - x_{\text{CUDA}} = 3.9 \times 10^{-22} \pm 2.2 \times 10^{-17}$ and $d_\sigma = \sigma_{\text{CPU}} - \sigma_{\text{CUDA}} = 4.0 \times 10^{-13} \pm 8.1 \times 10^{-10}$, fully consistent with zero.

We obtain statistically similar results for the other systems of equations, which indicates a robust numerical stability of the code.

5. Conclusions and future perspectives

In this work, we ported the Gaia AVU-GSR solver on a GPU environment, with the CUDA parallelization language. The MPI+CUDA code has just been put in production on Marconi100 (see Section 1). The MPI+CUDA code presents an acceleration factor of ~14 over the same code fully parallelized on the CPU, with a MPI+OpenMP paradigm, which might increase for systems occupying a larger amount of memory. To obtain this result, we ran both the CUDA and the OpenMP applications on the CINECA SuperComputer Marconi100, with 256 GB of RAM, 32 physical cores, and 4 V100 GPUs of 16 GB of memory each per node. We also compared the solutions and their standard errors obtained with the CUDA and the OpenMP codes for several systems of equations occupying different amounts of memory, verifying the numerical stability of the AVU-GSR code.

The largest system we solved occupies a memory of 303 GB, containing ~1.6x10^9 observations of the stellar parameters and ~7.6x10^6 stars, about two orders of magnitude less than the values expected for the final dataset of Gaia, that might contain ~10^{11} observations and ~10^8 stars. In the next months we aim to perform performance, scalability, and numerical stability tests increasing the size of the system to solve, up to sizes of the expected final dataset produced by the measurements performed with the Gaia satellite, to explore the behaviour of this application in perspective of (pre-)Exascale systems.

In the next months, we plan to repeat these tests on the pre-exascale platform Leonardo of CINECA that will be operative by the end of this year, rearranging the CUDA kernels to match the architecture of the GPUs of Leonardo. Leonardo will have 4 next-generation A100 GPUs per node, with a larger memory (64-80 GB) and a larger number of SMs, that allows a larger number of concurrent threads per GPU, compared to the V100 GPUs present on Marconi100, and less performant CPUs than on

Marconi100. We thus expect an additional boost of the performances of the CUDA code when it will be run on this platform.

Besides these tests, we also aim to investigate how much power we can save with the CUDA code compared to the CPU code, increasing the system size. Evaluating this power gain might not be trivial: the data transfers between the host and the device might be quite power-consuming but this might be compensated by a $\gtrsim 14$ acceleration factor. We aim to measure the power consumption by the node and by the GPUs for the OpenMP and the CUDA codes with the countdown library developed by the University of Bologna in collaboration with CINECA [18,19]. We plan to perform these measurements both on Marconi100 and on Leonardo. We expect to obtain a higher power saving of the CUDA code on Leonardo than on Marconi100, since, having larger GPUs, less nodes would be required to solve the same system compared to Marconi100. This trend might be inverted for the OpenMP code, since the CPU on Leonardo will be less performant than on Marconi100. For this reason we also expect the OpenMP code performance to worsen when run on Leonardo compared to Marconi100. Indeed, next-generation (pre-)Exascale platforms will be increasingly oriented towards calculation on GPU devices, at the expense of CPU computing. This study would be important in terms of Green Computing, since increasingly efficient parallelization methods on next-generation infrastructures while maintaining a low environmental impact is of growing interest for the national and international community.

This analysis is not only finalized to study the properties of the Gaia AVU-GSR code but to investigate the behaviour towards (pre-)Exascale systems of a class of applications following a structure similar to the one of the Gaia AVU-GSR code, based on the LSQR algorithm, currently employed in several contexts (e.g. geophysics, medicine, tomography, industry, and astronomy).

References and websites

[1] https://sci.esa.int/web/gaia

[2] Paige, C.C., and Saunders, M.A., 1982a, ACM Trans. Math. Softw. (TOMS) 8, 43-71.

[3] Paige, C.C., and Saunders, M.A., 1982b, ACM Trans. Math. Softw. (TOMS) 8, 195-209.

[4] Becciani, U., Sciacca, E., Bandieramonte, M., Vecchiato, A., Bucciarelli, B., and Lattanzi, M. G., 2014, in: 2014 Int. Conference on HPCS, 104-111, DOI: https://doi.org/10.1109/HPCSim.2014.6903675.

[5] Cesare, V., Becciani, U., Vecchiato, A., Lattanzi, M. G., Pitari, F., Raciti, M., Tudisco, G., Aldinucci, M., and Bucciarelli, B., in ADASS XXXI, in press.

[6] Cesare, V., Becciani, U., Vecchiato, A., Lattanzi, M. G., Pitari, F., Raciti, M., Tudisco, G., Aldinucci, M., and Bucciarelli, B., submitted to Astron. Comput..

[7] Cesare, V., Becciani, U., Vecchiato, A., Pitari, F., Raciti, M., Tudisco, G., and Aldinucci, M., INAF Technical Report, submitted.

[8] Cesare, V., Becciani, U., Vecchiato, A., Lattanzi, M. G., Pitari, F., Raciti, M., Tudisco, G., Aldinucci, M., and Bucciarelli, B., in preparation.

[9] Liang, S. X., Jiao, Y. J, Fan, W. X., Yang, B. Z., 2019a, Progress in Geophysics, 34, 1475-1480, DOI: https://doi.org/10.6038/pg2019CC0275.

[10] Liang, S. X., Wang, Q., Jiao, Y. J, Liao, G. Z., Jing, G., 2019b, Geophysical and Geochemical Exploration, 43, 359-366, DOI: https://doi.org/10.11720/wtyht.2019.1261.

[11] Bin, G., Wu, S., Shao, M., Zhou, Z., and Bin, G., 2020, J. Electrocardiol., 62, 190-199, DOI: https://doi.org/10.1016/j.jelectrocard.2020.08.017.

[12] Guo, H., Zhao, H., Yu, J., He, X., He, X., Song, X., 2021, J. Biophotonics, e202100089.

[13] Jaffri, N. R., Shi, L., Abrar, U., Ahmad, A., and Yang, J., 2020, in: Proceedings of the 2020 5th Int. Conference on Multimedia Systems and Signal Processing, 16-20.

[14] Borriello, L., Dalessandro, F., Murgolo, F., Prezioso, G., 1986, Mem. Soc. Astron. Ital., 57, 267-289

[15] Naghibzadeh, S., and van der Veen, A. J., 2017, in: 2017 IEEE Int. Conference on Acoustics, Speech, and Signal Processing (ICASSP), IEEE, 3385-3389.

[16] https://www.hpc.cineca.it/hardware/marconi100

[17] https://developer.nvidia.com/nsight-systems

[18] Cesarini, D., Bartolini, A., Borghesi, A., Cavazzoni, C., Luisier, M., and Benini, L., 2020, IEEE Trans. Parallel Distrib. Syst., 31, 11, 2696-2709, DOI: https://doi.org/10.1109/TPDS.2020.3000418.

[19] Cesarini, D., Bartolini, A., Bonfa, P., Cavazzoni, C., and Benini, L., 2021, IEEE Trans. Comput., 70, 5, 682-695, DOI: https://doi.org/10.1109/TC.2020.2995269.