IRIS AperTO

UNIVERSITÀ DEGLI STUDI DI TORINO

AperTO - Archivio Istituzionale Open Access dell'Università di Torino

The Gaia AVU-GSR parallel solver: Preliminary studies of a LSQR-based application in perspective of exascale systems

This is a pre print version of the following article:

Original Citation:

Availability:

This version is available http://hdl.handle.net/2318/1890597 since 2023-02-06T09:57:06Z

Published version:

DOI:10.1016/j.ascom.2022.100660

Terms of use:

Open Access

Anyone can freely access the full text of works made available as "Open Access". Works made available under a Creative Commons license can be used according to the terms and conditions of said license. Use of all other works requires consent of the right holder (author or publisher) if not exempted from copyright protection by the applicable law.

(Article begins on next page)

03 May 2024

Rapporti Tecnici INAF
INAF Technical Reports

Number	163
Publication Year	2022
Acceptance in OA@INAF	2022-07-05T14:55:01Z
Title	The Gaia AVU-GSR parallel solver: preliminary porting with OpenACC parallelization language of a LSQR-based application in perspective of exascale systems
Authors	CESARE, VALENTINA; BECCIANI, Ugo; VECCHIATO, Alberto; PITARI, FABIO; RACITI, MARIO; TUDISCO, GIUSEPPE; Aldinucci, Marco
Affiliation of first author	O.A. Catania
Handle	http://hdl.handle.net/20.500.12386/32451; https://doi.org/10.20371/INAF/TechRep/163

Technical Report

The Gaia AVU-GSR parallel solver: preliminary porting with OpenACC parallelization language of a LSQR-based application in perspective of exascale systems

Cesare Valentina
Becciani Ugo
Vecchiato Alberto
Pitari Fabio
Raciti Mario
Tudisco Giusepe
Aldinucci Marco

Abstract

The Gaia Astrometric Verification Unit-Global Sphere Reconstruction (AVU-GSR) Parallel Solver aims to find the positions and the proper motions for ~10^8 stars in our galaxy, besides the attitude and the instrumental settings of the Gaia satellite, and the global parameter γ of the post Newtonian formalism. To find these parameters, the code solves a system of linear equations, $\mathbf{A} \times \boldsymbol{x} = \boldsymbol{b}$, where the coefficient matrix $\mathbf{A}$ is large, containing ~10^{11}x10^8 elements, and sparse. The system of equations is solved with a customized implementation of the iterative preconditioned (PC)-LSQR algorithm and is parallelized on the CPU with MPI+OpenMP, where the computation related to different horizontal portions of the coefficient matrix is assigned to different MPI processes and it is further parallelized on the OpenMP threads. To improve the code performance, we explored the feasibility of a porting of this application on a GPU environment, by replacing the OpenMP directives with the OpenACC correspondent ones. In this preliminary porting, the ~95% of the data is copied from the host (CPU) to the device (GPU) before the entire cycle of iterations, making the code *compute bound* rather than *data-transfers bound*. The OpenACC code accelerates of a factor of ~1.5 compared to the OpenMP code. The OpenACC application runs on multiple GPUs and it was tested on the CINECA SuperComputer Marconi100, with 4 V100 GPUs per node having 16 GB of memory each. A following porting, where the OpenACC language is replaced with CUDA, was performed, optimizing the preliminary porting with OpenACC. The CUDA code has just been put into production on Marconi100 and we plan to run it on the future pre-exascale platform Leonardo of CINECA, with 4 next-generation A100 GPUs per node.

Sommario

1. Introduction 4

2. Structure of the entire AVU-GSR application and of the coefficient matrix 5

3. The OpenACC porting of the Gaia AVU-GSR parallel solver 7

3.1 Multi-GPU computation 7

3.2 Data transfers 8

3.3 Parallelization with OpenACC directives 9

4. Performance comparison with the MPI+OpenMP code 10

4.1 Detailed analysis of the speedup of the OpenACC code over the OpenMP code 11

5. GPU utilization 13

6. Conclusions and future perspectives 13

References and websites 15

1. Introduction

The European Space Agency (ESA)'s Gaia mission [1] has provided, since its launch occurred on 19th December 2013, a catalogue of astrometric parameters (parallaxes, sky positions, and proper motions) of ~10^9 stars in the Milky Way, ~1% of its total content, with an accuracy at the micro-arcsecond level [1,2,3]. The third data release of Gaia (DR3) has just been published on 13th June 2022 [3].

In this report, we present the Gaia Astrometric Verification Unit-Global Sphere Reconstruction (AVU-GSR) Parallel Solver, an application developed under the Data Processing and Analysis Consortium (DPAC) that aims to find these astrometric parameters for the primary stars of the global astrometric sphere of the Gaia mission, i.e. for ~10^8 stars. Besides the astrometric parameters, this solver will constrain the attitude and the instrumental specifications of the Gaia spacecraft and the parameter γ of the Parametrized Post-Newtonian (PPN) formalism of relativistic gravity theories to model space-time and to test general relativity against alternative theories of gravity [4].

The code finds these parameters by solving a linearized system of equations, of the form:

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

where $\mathbf{A}$ is the coefficient matrix, $\boldsymbol{x}$ the array of the unknowns to solve, and $\boldsymbol{b}$ the array of the known terms. The matrix $\mathbf{A}$ is large, including ~10^{11}x10^8 elements, and sparse, obeying to a peculiar sparsity scheme. The rows of the matrix represent the observations of the stellar parameters, where each of the ~10^8 stars is observed ~10^3 times on average, whereas the number of columns of $\mathbf{A}$ is the number of unknowns to solve. To reduce the computation time, only the elements of $\mathbf{A}$ different from zero are considered during calculations, passing from a sparse to a dense matrix, $\mathbf{A_d}$. The dense matrix $\mathbf{A_d}$ contains ~10^{11}x10^1 elements. This system of equations is overdetermined, being the number of equations larger than the numbers of unknowns. For this reason, it is solved in the least-square sense, adopting a customized version of the iterative LSQR algorithm [5,6], whose iterations stop when either a convergence condition or a maximum number of iterations set at runtime is reached.

The version of the AVU-GSR code in production on the CINECA SuperComputer Marconi100 [7] since 2014 is written in C and C++ and it is parallelized on the CPU with a hybrid MPI+OpenMP approach [8]. This application was developed for the ESA Gaia mission, under an agreement between Istituto Nazionale di Astrofisica (INAF) and CINECA and with the support of Agenzia Spaziale Italiana (ASI) and it is currently employed by the Coordination Unit 3 (CU3) of the Data Processing and Analysis Consortium (DPAC). The complete AVU-GSR process is managed by the Data Processing Center of Turin (DPCT), which is supervised by the Aerospace Logistics Technology Engineering Company (ALTEC) and by the Astrophysics Observatory of Turin of INAF (INAF-OATO).

To accelerate of the AVU-GSR code, we ported it on a GPU environment. This report presents a preliminary porting of this application, where the OpenMP high-level directives are replaced by their OpenACC counterparts, finalized to understand the performance improvements due to the parallelization on GPU devices in perspective of a more optimized CUDA porting on pre-exascale systems [9,10,11,12]. With this first porting, we achieve a speedup of ~1.5 over the OpenMP code, result obtained by comparing the performances of the OpenMP and of the OpenACC codes on the CINECA infrastructure Marconi100.

The OpenACC code is 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. Structure of the entire AVU-GSR application and of the coefficient matrix

As said in Section 1, to solve the system of equations the code employs a customized implementation of preconditioned (PC)-LSQR algorithm, an iterative conjugate-gradient type algorithm that can solve an overdetermined system (number of equations > number of unknowns) of linear equations in the least-square sense. To find a unique solution, an additional number of constraints equations is set at the end of the system. For *preconditioning technique* it is intended a proper normalization of the coefficient matrix to accelerate the convergence speed of the LSQR. The main part of the LSQR procedure is the calling, at each iteration, of the *aprod* function, either in mode 1 or in mode 2. The aprod 1 computes the operation $\mathbf{A} \times x^i$ and the aprod 2 computes the operation $\mathbf{A}^{\mathrm{T}} \times y^i$, where x^i is the i-th iterative estimate of the unknowns array and y^i is proportional to the array of residuals $y'^i = \boldsymbol{b} - \mathbf{A} \times x^i$. The convergence is accomplished when y'^i goes below a pre-defined tolerance tol, set to the machine precision.

Figure 1 summarizes the entire structure of the AVU-GSR application and details how the aprod 1 and 2 operations are actually implemented in the LSQR algorithm. The aprod 2 function is also employed to calculate the initial solution for the LSQR procedure.

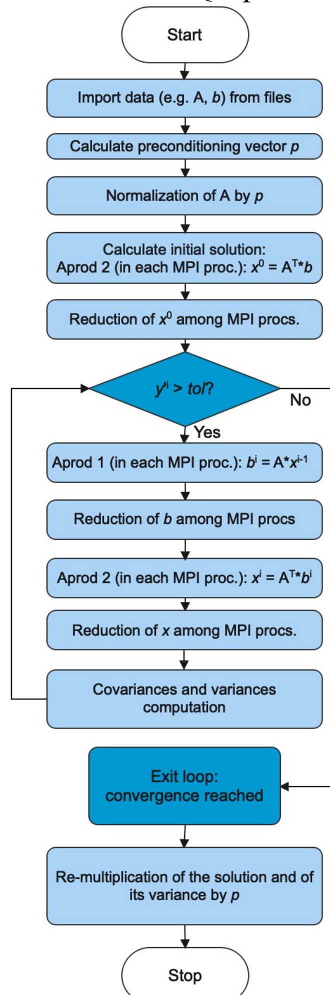

Figure 1: Structure of the entire Gaia AVU-GSR application with details of the aprod 1 and 2 operations.

Instead, Figure 2 illustrates the MPI+OpenMP parallelization scheme of the system of equations. In particular, it shows a system parallelized on four MPI processes allocated on one node of a computer cluster, where each MPI process is represented with a different colour. Each horizontal portion of the coefficient matrix, representing a subset of the total number of observations, is assigned to a different MPI process and it is further parallelized over the OpenMP threads. The number of observations assigned to each MPI process is stored in a 1D array, $\boldsymbol{N}$, whose index goes from 0 to the number of MPI processes set at runtime minus 1. The for loop iterating on the observations related to each MPI process is parallelized with OpenMP with the `#pragma omp for` directive, placed within the `#pragma omp parallel` directive that defines a parallel region of the code on the CPU.

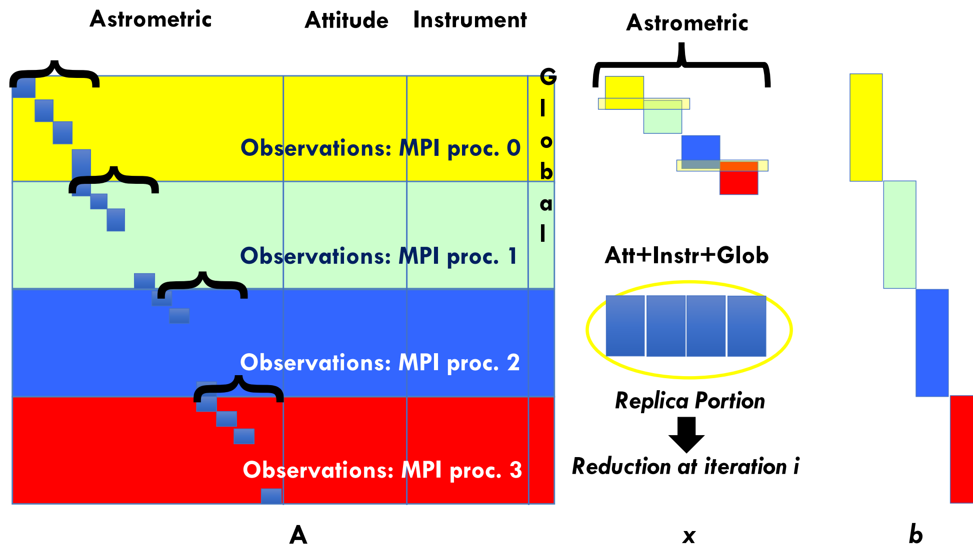

Figure 2: Parallelization scheme of the system of equations (Eq. (1)) on 4 MPI processes in one node of a computer cluster. The MPI processes are represented with four different colours.

Each row of the coefficient matrix, i.e. each observation equation, contains four sections, the astrometric, the attitude, the instrumental, and the global ones, divided by vertical lines in Figure 2. The 90% of the coefficients are contained in the astrometric section and we have 5 astrometric coefficients per star, where the number of stars is typically in the range of [10^6-10^8]. The astrometric coefficients follow a regular pattern and they are organized in a block-diagonal structure (see the blue blocks in the Astrometric part of the coefficient matrix shown in Figure 2). Given this regular pattern, the computation related to the astrometric part of the solution array is easily distributed among the MPI processes, whereas the computation related to the other three parts is replicated on the MPI processes, which does not imply a substantial slowdown of the code since the other three parts only represent the 10% of the total. The replicated parts of the solution array on the different MPI processes are reduced in a single value at the end of each iteration.

Each row of the coefficient matrix contains 5, 12, 6, and 1 astrometric, attitude, instrumental, and global parameters different from zero, respectively. As said in Section 1, to perform the computation in reasonable timescales, only the non-zero parameters are considered in the coefficient matrix during calculations. An appropriate map containing the indexes that these parameters had in the complete coefficient matrix is defined in two 1D arrays, one for the astrometric and the attitude portions and one for the instrumental portion. The global portion always occupies the last column of the coefficient matrix.

3. The OpenACC porting of the Gaia AVU-GSR parallel solver

We ported the AVU-GSR solver on a GPU environment, by replacing the OpenMP part with OpenACC (see [13] for the description of a semi-automatic methodology to parallelize scientific applications with the OpenACC parallelization language).

3.1 Multi-GPU computation

The MPI+OpenACC code runs on multiple GPUs according to the number of MPI processes set at runtime, as specified by the instruction at line 3 of Figure 3, which represents the AVU-GSR code parallelized in OpenACC.

```
1  Read parameters from files
2  Preconditioning
3  acc_set_device_num(pid%acc_get_num_devices(acc_device_default),acc_device_default)
4  #pragma acc enter data copyin(A_d,M_i,I_c,N)
5  #pragma acc enter data copyin(x,b)
6  aprod 2 call (calculation of the initial solution)
7  #pragma acc exit data copyout(x,b)
8  MPI reduce
   // LSQR algorithm
9  while (conv. cond. || max itn. reached) do
10     #pragma acc enter data copyin(x,b)
11     aprod 1 call
12     #pragma acc exit data copyout(x,b)
13     MPI reduce
14     #pragma acc enter data copyin(x,b)
15     aprod 2 call
16     #pragma acc exit data copyout(x,b)
17     MPI reduce
18     Covariances and variances computation
```

Figure 3: Structure of the entire MPI+OpenACC Gaia AVU–GSR application.

The MPI processes in each node of a computer cluster are assigned to the GPUs of the node in a round-robin fashion, as also illustrated in Figure 4, that represents the coefficient matrix of the system of equations parallelized on 4 nodes of a computer cluster, with 4 MPI processes allocated per node and 4 GPUs per node.

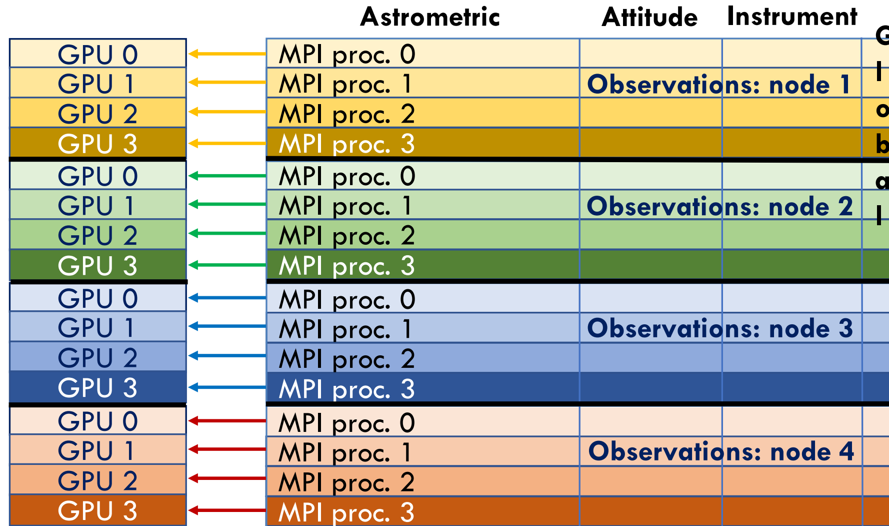

Figure 4: Parallelization scheme of the coefficient matrix of the system of equations on 4 nodes of a computer clusters with 4 MPI processes allocated per node and 4 GPUs per node.

Making another example, if a system is parallelized on 8 MPI processes per node on a platform with 4 GPUs per node, the processes with rank 0 and 4 are assigned to GPU 0, the processes with rank 1 and 5 to GPU 1, the processes with rank 2 and 6 to GPU 2, and the processes with rank 3 and 7 to GPU 3.

3.2 Data transfers

The management of the data transfers between the host (CPU) and the device (GPU) has to be taken with particular care in GPU programming. Indeed, the potential performance gain due to a parallelization on a device with a number of cores ~10^2 larger than on the CPU might be cancelled by a bad management of these data movements. Properly managing the data copies is especially important in iterative algorithms, where it is essential to transfer as much data as possible before and/or after the entire iteration cycle, and to reduce the transfers at every step of the algorithm.

These data copies are defined through specific directives placed in strategical points of the code, highlighted in red in Figure 3. The data-copy regions are also highlighted in green (host-to-device – H2D transfers) and purple (device-to-host – D2H transfers) in Figure 5, that represents the output of the NVIDIA Nsight Systems profiler [14] for a run with 4 iterations of the LSQR algorithm parallelized on 4 MPI processes, and consequently on 4 GPUs, in one node of the CINECA SuperComputer Marconi100. Further details of this figure are provided in Section 4.1.

In this porting, the ~95% of the data are moved H2D before the LSQR algorithm (see line 4 of Figure 3 and the large green region in the left part of Figure 5). Instead, only the solution and the known terms arrays, which represent the ~5% of the memory occupied by the system, are moved at each iteration of the LSQR algorithm. Specifically, at each LSQR step, these two arrays are moved both H2D and D2H, in correspondence of both the aprod 1 and 2 functions, as highlighted at lines 10, 12, 14, and 16 of Figure 3 and in Figure 5.

Figure 5: Output of the NVIDIA Nsight Systems profiler [14] for a run of the MPI+OpenACC code with 4 iterations of the LSQR algorithm. The code is parallelized on 4 MPI processes on one node of Marconi100, and thus on 4 GPUs, and the system occupies a memory of 10 GB. This output shows the portion of the OpenACC code from line 4 to line 17 of Figure 3.

3.3 Parallelization with OpenACC directives

We ported with OpenACC both the aprod 1 and the aprod 2 functions. In Section 2, we said that the for loop iterating on the number of observations assigned to each MPI process was parallelized with the `#pragma omp for` directive, placed within a `#pragma omp parallel` directive, that defines a parallel region of the code on the CPU. We substantially replaced these two directives with the `#pragma acc parallel` directive, which starts a parallel execution on the GPU, and with the `#pragma acc for` directive. To be effective, the `#pragma acc parallel` directive requires an analysis by the programmer to ensure safe parallelism of the region of the code that is enclosed within the directive scope. For the aprod 1, we defined four distinct parallel regions, called also GPU kernels, for the astrometric, the attitude, the instrumental, and the global sections of the system, whereas we defined the aprod 2 in a single parallel region. In the aprod 2 region, we also employed the `#pragma acc atomic` directive, to prevent different GPU threads to concurrently write the same elements of the unknowns array $\boldsymbol{x}$. Figures 6 and 7 compare the OpenMP and the OpenACC pseudocodes related to the astrometric part of the aprod 1 and 2 functions, where the OpenMP directives are highlighted in blue and the OpenACC directives are highlighted in red. In the figure, "$N_{\texttt{Astro}}$=5" is the number of non-zero astrometric coefficients per observation equation. The other three sections, attitude, instrumental, and global, follow a similar structure. For more details concerning the parallelization structure of the MPI+OpenMP and MPI+OpenACC versions of the AVU-GSR parallel solver consult [9,10].

aprod 1 with OpenMP

omp.1 `#pragma omp parallel private(`pid, sum`)` `shared(`$\boldsymbol{N}, \boldsymbol{x}, \mathbf{A_d}, \boldsymbol{b}$`)`
omp.2 {
omp.3 `#pragma omp for`
omp.4 **for** $i \leftarrow 0$ **to** $\boldsymbol{N}[pid]$ **do**
omp.5 $sum = 0.0$
// Astrometric sect.
omp.6 $k = i \times N_{\mathrm{par}}$
omp.7 **for** $j \leftarrow 0$ **to** N_{Astro} **do**
omp.8 $sum = sum + \mathbf{A_d}[k]\boldsymbol{x}[j + \mathrm{offset}[i]]$
omp.9 $k{+}{+}$
…
}

aprod 1 with OpenACC

// Astrometric sect.
acc.1 `#pragma acc parallel private(`sum`)`
acc.2 {
acc.3 `#pragma acc loop`
acc.4 **for** $i \leftarrow 0$ **to** $\boldsymbol{N}[pid]$ **do**
acc.5 $sum = 0.0$
acc.6 $k = i \times N_{\mathrm{par}}$
acc.7 **for** $j \leftarrow 0$ **to** N_{Astro} **do**
acc.8 $sum = sum + \mathbf{A_d}[k + j]\boldsymbol{x}[j + \mathrm{offset}[i]]$
acc.9 $\boldsymbol{b}[i] = \boldsymbol{b}[i] + sum$
acc.10 }

Figure 6: Pseudocode of the astrometric part of the aprod 1 function parallelized with OpenMP (left panel) and OpenACC (right panel). The OpenMP directives are highlighted in blue and the OpenACC directives are highlighted in red.

```
     aprod 2 with OpenMP
omp.1 #pragma omp parallel private(pid,tid,nth)
      shared(N,x,A_d,b)
omp.2 {
     // ID number of the OpenMP thread
omp.3   tid = omp_get_thread_num()
     // Number of OpenMP threads
omp.4   nth = omp_get_num_threads();
omp.5   for i ← N_t[tid][0] to N_t[tid][1] do
          // Astrometric sect.
omp.6     k = i × N_par
omp.7     for j ← 0 to N_Astro do
omp.8         x[j + offset[i]] = x[j + offset[i]] + A_d[k]b[i]
omp.9         k++
     ...
     }
```

```
acc.1 aprod 2 with OpenACC
acc.2 #pragma acc parallel
acc.3 {
acc.4   #pragma acc loop
acc.5   for i ← 0 to N[pid] do
          // Astrometric sect.
acc.6     k = i × N_par
acc.7     for j ← 0 to N_Astro do
acc.8         #pragma acc atomic
acc.9         x[j + offset[i]] = x[j + offset[i]] + A_d[k + j]b[i]
      ...
      }
```

Figure 7: Pseudocode of the astrometric part of the aprod 2 function parallelized with OpenMP (left panel) and OpenACC (right panel). The OpenMP directives are highlighted in blue and the OpenACC directives are highlighted in red.

4. Performance comparison with the MPI+OpenMP code

We compared the performance of the MPI+OpenMP and of the MPI+OpenACC versions of the Gaia AVU-GSR application by running the two codes on the CINECA SuperComputer Marconi100. Each node of this cluster has:

1) 32 physical cores, distributed between 2 sockets that host 16 cores each. Each physical core corresponds to 4 virtual cores with a total of 128 (32 x 4) virtual cores per node;
2) 4 NVIDIA Volta V100 GPUs, with 16 GB of memory each;
3) 256 GB of RAM.

Specifically, we performed three performance tests:

1) On one node, maintaining the memory of the system fixed to 10 GB, on an increasing number of MPI processes (top-left panel of Figure 8);
2) On an increasing number of nodes, with 4 MPI processes per node, maintaining the memory of the system fixed to 40 GB (top-right panel of Figure 8);
3) On an increasing number of nodes, with 4 MPI processes per node, setting the memory of the system proportional to the number of nodes, with 40 GB per node (40 GB on one node, 80 on two nodes, 160 on four nodes, etc.) (bottom panel of Figure 8).

In the top-left panel of Figure 8, we plot the average time of one LSQR iteration as a function of the number of MPI processes, set to 1, 2, 4, 8, 16, and 32. For the OpenMP code, the number of OpenMP threads per MPI process, represented on the top axis, are set such that the product between the number of MPI processes and the number of OpenMP threads is equal to 32, namely the number of physical cores per node.

In the top-right panel of Figure 8, we plot the average time of one LSQR iteration as a function of the number of nodes. We ran on a number of nodes equal to 1, 2, 4, 8, and 16. Since we ran on 4 MPI processes per node, the two codes were parallelized on 4, 8, 16, 32, and 64 MPI processes. The OpenMP code is parallelized on 8 threads per MPI processes (4 MPI processes per node x 8 OpenMP threads per MPI process = 32 = number of physical cores per node). In the bottom panel of Figure 8

the quantities represented on the axes and the number of resources employed are as in the top-right panel.

Except for the case where the OpenACC code is parallelized on 1 and 2 MPI processes (first two points in the top-left panel of Figure 8), when the performances of the OpenACC code are worse since it is parallelized only on 1 and 2 GPUs, respectively, the time ratio between the OpenMP and the OpenACC codes remains nearly constant. Indeed, in all the other cases the OpenACC code is parallelized on 4 GPUs per node (the maximum number of GPUs per node). The optimal configuration for the OpenACC code is to run on 4 MPI processes per node, since we obtain the best performance employing the smaller number of resources. Instead, as we can see from the top-left panel of Figure 8, the optimal configuration for the OpenMP code is to run on 16 MPI processes per node and 2 OpenMP threads per MPI process. The best performance gain obtained by the OpenACC code over the OpenMP code is of $\eta \sim 1.5$.

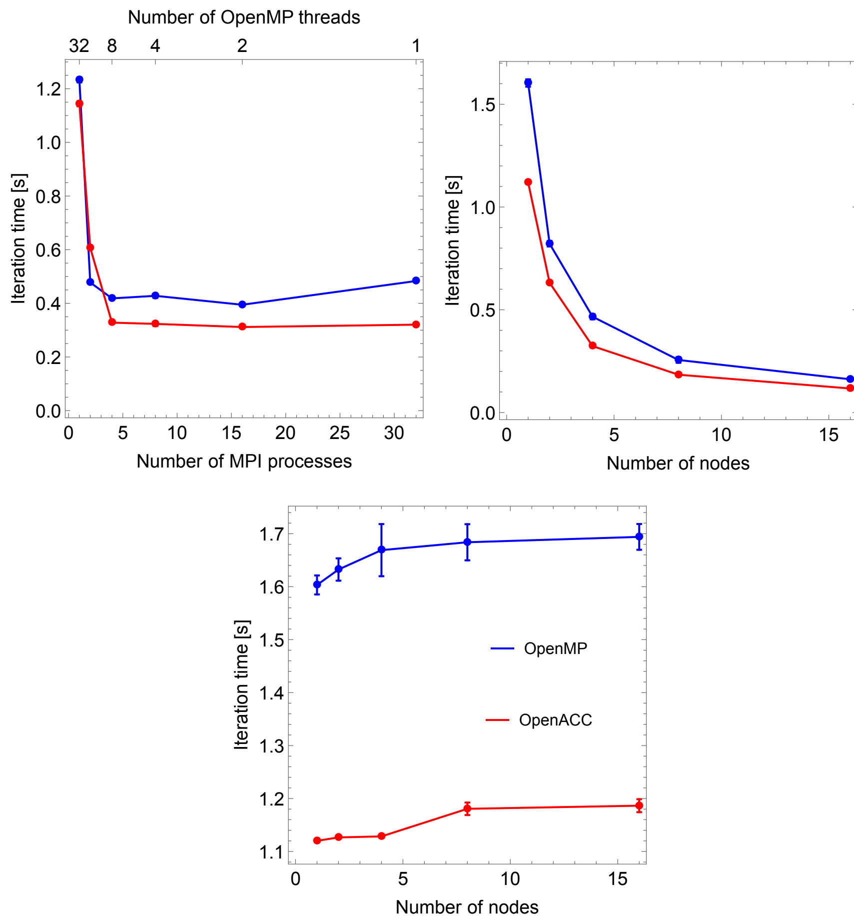

Figure 8: Performance tests for the MPI+OpenMP (blue lines and error bars) and the MPI+OpenACC (red lines and error bars) versions of the Gaia AVU-GSR application. Top-left panel: intra-node fixed memory performance test. Top-right panel: inter-nodes fixed memory performance test. Bottom panel: inter-nodes proportional memory performance test.

4.1 Detailed analysis of the speedup of the OpenACC code over the OpenMP code

We now investigate the origin of the speedup of the OpenACC code over the OpenMP code by comparing the execution times of different regions of the two applications. To perform this study, we compare an execution of the OpenACC and of the OpenMP codes, both run in their optimal configurations on one node of Marconi100 (4 MPI processes for the OpenACC code and 16 MPI processes + 2 OpenMP threads for the OpenMP code, as explained in Section 4). The system occupies 10 GB of memory, as in the top-left panel of Figure 8.

As anticipated in Section 3.2, Figure 5 illustrates the output of the NVIDIA Nsight Systems profiler tool for the run considered in this section. The top panel of Figure 9, shows a zoom-in of one iteration of the run illustrated in Figure 5, where the iteration time equal to $t_{\mathrm{iter,ACC}}$ = 314.730 ms ~ 0.31 s is superimposed to the figure. The output of the NVIDIA Nsight Systems profiler shows the computation regions due to the GPU kernels (blue), the H2D and the D2D copies (green and purple, respectively), and the regions still running on the CPU (white), and it is particularly useful to analyze the regions of code that might be further optimized (see [11]).

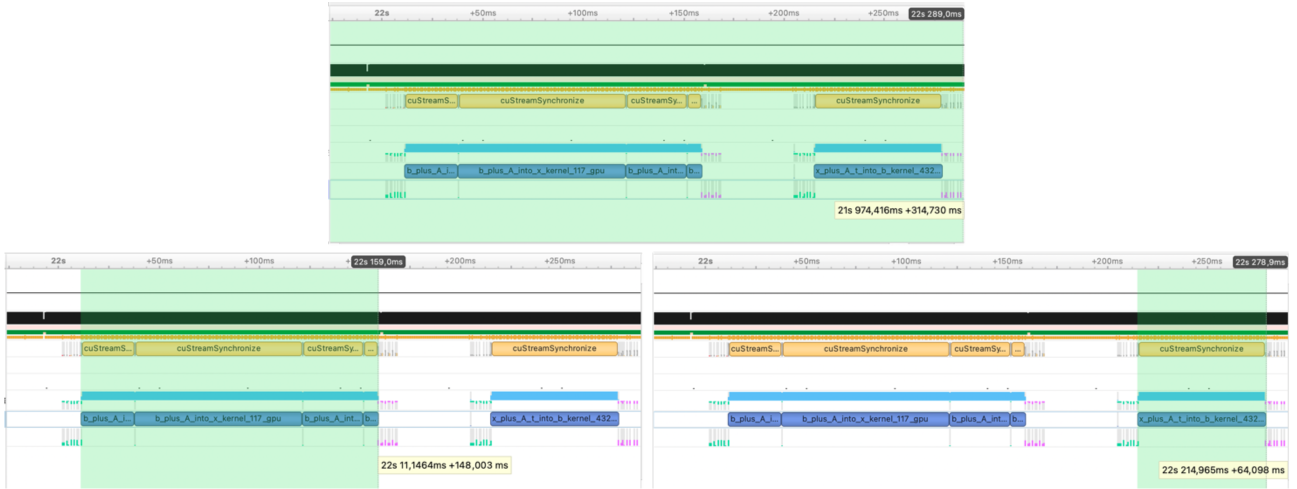

Figure 9: Zoom-in of one iteration of the LSQR algorithm for the run illustrated in Figure 5. In the top, bottom-left, and bottom-right panels the times of one complete iteration of the LSQR cycle, of the only aprod 1 region, and of the only aprod 2 region are highlighted, respectively.

The regions that we ported on the GPU are the aprod 1 and 2 functions, represented by the blue areas labelled as “b_plus...” (aprod 1) and “x_plus...” (aprod 2) in Figure 9. The bottom-left and the bottom-right panels of Figure 9 are equal to the top panel of the same figure with highlighted the times of the aprod 1 (148.003 ms) and 2 (64.098 ms) regions. Whereas the execution times of the aprod 1 and 2 regions in the OpenACC code are equal to $t_{a1,\mathrm{ACC}}$ ~ 0.15 s and $t_{a2,\mathrm{ACC}}$ ~ 0.064 s, the correspondent times in the OpenMP code are of $t_{a1,\mathrm{OMP}}$ ~ 0.12 s and of $t_{a2,\mathrm{OMP}}$ ~ 0.23 s, respectively. This means that, while aprod 1 region actually decelerates in the OpenACC code compared to the OpenMP code of a factor of ~0.8, the aprod 2 region accelerates to a factor of ~3.6. From this result we can conclude that the speedup of the OpenACC code over the OpenMP code is due to the aprod 2 region.

To calculate the global speedup, also the memory transfers regions and the CPU regions have to be taken into account. Given that the time due to data copies in the OpenACC code is equal to t_{Mem} ~ 0.04 s and that the time of the CPU regions on both codes is equal to t_{CPU} ~ 0.064 s, the speedup results equal to:

$$\eta' = \frac{t_{a1,\mathrm{OMP}} + t_{a2,\mathrm{OMP}} + t_{\mathrm{CPU}}}{t_{a1,\mathrm{ACC}} + t_{a2,\mathrm{ACC}} + t_{\mathrm{Mem}} + t_{\mathrm{CPU}}} \sim 1.3, \qquad (2)$$

which is consistent with the value $\eta \sim 1.5$ found in Section 4.

The GPU kernels computation, due to the aprod 1 and 2 functions, represents the $\frac{t_{a1,\mathrm{ACC}}+t_{a2,\mathrm{ACC}}}{t_{\mathrm{Iter,ACC}}} \times 100 \sim 69.0\%$ of the iteration time, whereas the data movements only represent the $\frac{t_{\mathrm{Mem}}}{t_{\mathrm{Iter,ACC}}} \times 100 \sim 12.9\%$ of the iteration time. As we can see from the left panel in Figure 5, the data movements

only represent the 11% of the entire execution time. This result indicates that the code is *compute bound* and not *data-transfers bound*.

5. GPU utilization

At compile time of the MPI+OpenACC code, we set a number of GPU registers equal to 32. This is a logical choice for a NVIDIA V100 GPU, since its architecture is organized such that groups of 32 registers see the same cache memory and are subject to the same operation in a Single Instruction Multiple Data (SIMD)-like fashion. On the software side, this is encoded in the size of a warp, a logical block of 32 threads that always perform the same operations concurrently. Each warp is directly mapped on each block of 32 registers.

To verify if 32 registers correspond to the optimal choice, we exploited the NVDIA Nsight Compute profiler tool [15], which shows the percentage of occupancy (SM %) of the GPU compared to the maximum occupancy. Figure 10 shows this percentage for a 32-registers compilation of the OpenACC code (green), compared to three other configurations, compiled with 64 (light blue), 128 (purple), and 42 (orange) registers. The figure clearly shows that the GPU occupancy is better exploited in the 32-registers case (~78%) compared to the three other cases (~46%). The code compiled with 32 registers provides the smaller iteration time of the LSQR algorithm.

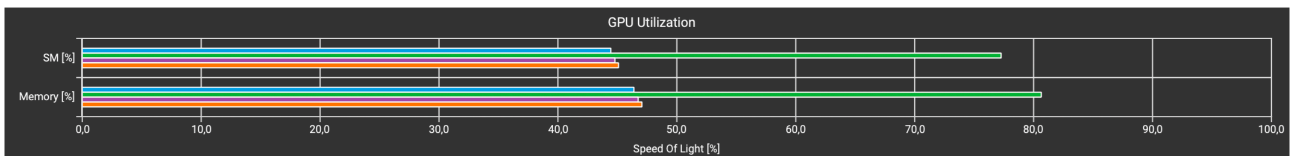

Figure 10: Percentage of utilization of the compute (SM) and of the memory resources of the GPU compared to the theoretical maximum (Speed Of Light metric) when the OpenACC code is run on 32 (green bar), 64 (light blue bar), 128 (purple bar), and 42 (orange bar) GPU registers, as set during compilation. The plot is performed with the NVIDIA Nsight Compute profiler tool.

6. Conclusions and future perspectives

In this report, we present a preliminary porting of the Gaia AVU-GSR code on a GPU environment with the OpenACC parallelization language. This analysis is propaedeutic to a more optimized porting with the CUDA parallelization model, that aims to exploit at most the GPU architecture, which is object of a paper in preparation [12] and of a submitted INAF technical report [11].

The MPI+OpenACC code is *compute bound*, that is the execution time is dominated by GPU kernel computation and not by memory transfers between the host and the device. Indeed, we managed to transfer H2D the ~95% of the data before the entire LSQR cycle.

We compared the performances of the OpenMP and of the OpenACC codes by running the two applications on the CINECA SuperComputer Marconi100, that has 32 physical cores per node, equally distribuited between two sockets, 256 GB of RAM and 4 NVIDIA Volta V100 GPUs per node of 16 GB of memory each. To perform these tests, we ran the two codes (I) on an increasing number of MPI processes in a single node of the cluster, considering a system with a fixed amount of memory, (II) on an increasing number of nodes of the cluster, considering a system with a fixed amount of memory, and (III) on an increasing number of nodes of the cluster, considering a system with a memory proportional to the number of nodes.

The performance tests demonstrate that if parallelized on 4 MPI processes per node, equal to the number of GPUs per node, the OpenACC code runs in its optimal configuration and it accelerates with respect to the OpenMP code of a factor of ~1.5. This speedup is mainly due to the porting of the aprod 2 function. To obtain the best performance of the OpenACC code and to obtain an optimal exploitation of the GPU occupancy, the OpenACC code has to be compiled on 32 GPU registers.

Even if with this OpenACC porting we obtained a moderate gain in performance compared to the CPU parallel code, further optimizations are possible. Despite the application is not *data-transfers bound*, we can see from Figure 9 that the code regions due to data copies are not negligible and they can be further reduced. Moreover, a not negligible section of the code is still running on the CPU and it can be ported on the GPU. However, a substantial gain in performance can be obtained by a change of parallelization model, from OpenACC, which provides a high-level parallelization approach, with directives, that does not change the structure of the original code, to CUDA, which provides a low-level parallelization approach that requires the rearranging of some parts of the code to manually define the GPU threads hierarchy to properly match the GPU architecture and the topology of the system to solve. The CUDA code is already in production on the Marconi100 infrastructure and it provides a speedup of ~14 over the OpenMP code but a further speedup is expected when the CUDA code will be rearranged to match the architecture of the next-generation A100 GPUs that will be present on the future pre-exascale infrastructure of CINECA Leonardo, which will be operational by the end of this year.

However, this simple porting has already provided essential information about the potential performance behaviour of the Gaia AVU-GSR code in perspective of (pre-)Exascale systems. This behaviour can be extended to other codes that obey to its same structure, namely that are 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] Gaia Collaboration: Brown, A. G. A., Vallenari, A., Prusti, T., et al., 2021, A&A, 650, C3, DOI: https://doi.org/10.1051/0004-6361/202039657e.

[3] Gaia Collaboration: Vallenari, A., Brown, A. G. A., Prusti, T., et al., in press, A&A, DOI: https://doi.org/10.1051/0004-6361/202243940.

[4] Vecchiato, A., Lattanzi, M. G., Bucciarelli, B., Crosta, M., de Felice, F., Gai, M., 2003, A&A, 399, 337-342, DOI: https://doi.org/10.1051/0004-6361:20021785, arXiv:astro-ph/0301323.

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

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

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

[8] 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.

[9] 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.

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

[11] Cesare, V., Becciani, U., and Vecchiato, A., INAF Technical Report, submitted.

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

[13] Aldinucci, M., Cesare, V., Colonnelli, I., Martinelli, A. R., Mittone, G., Cantalupo, B., Cavazzoni, C., and Drocco, M., 2021, JPDC, 157, 13-29, DOI: https://doi.org/10.1016/j.jpdc.2021.05.017.

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

[15] https://developer.nvidia.com/nsight-compute