Performance Portability Assessment in Gaia

Giulio Malenza, *Student Member, IEEE*, Valentina Cesare, Marco Edoardo Santimaria, Robert Birke, *Senior Member, IEEE*, Alberto Vecchiato, Ugo Becciani, and Marco Aldinucci

***Abstract*—Modern scientific experiments produce ever-increasing amounts of data, soon requiring ExaFLOPs computing capacities for analysis. Reaching such performance requires purpose-built supercomputers with $O(10^3)$ nodes, each hosting multicore CPUs and multiple GPUs, and applications designed to exploit this hardware optimally. Given that each supercomputer is generally a one-off project, the need for computing frameworks portable across diverse CPU and GPU architectures without performance losses is increasingly compelling. We investigate the performance portability ($\mathcal{P}$) of a real-world application: the solver module of the AVU–GSR pipeline for the ESA Gaia mission. This code finds the astrometric parameters of $\sim 10^8$ stars in the Milky Way using the LSQR iterative algorithm. LSQR is widely used to solve linear systems of equations across a wide range of high-performance computing applications, elevating the study beyond its astrophysical relevance. The code is memory-bound, with six main compute kernels implementing sparse matrix-by-vector products. We optimize the previous CUDA implementation and port the code to further six GPU-acceleration frameworks: C++ PSTL, SYCL, OpenMP, HIP, KOKKOS, and OpenACC. We evaluate each framework's performance portability across multiple GPUs (NVIDIA and AMD) and problem sizes in terms of application and architectural efficiency. Architectural efficiency is estimated through the roofline model of the six most computationally expensive GPU kernels. Our results show that C++ library-based (C++ PSTL and KOKKOS), pragma-based (OpenMP and OpenACC), and language-specific (CUDA, HIP, and SYCL) frameworks achieve increasingly better performance portability across the supported platforms with larger problem sizes providing better $\mathcal{P}$ scores due to higher GPU occupancies.**

***Index Terms*—High-performance computing, performance portability, portable languages, GPU programming, CPU and GPU architectures, astrometry.**

Received 8 March 2025; revised 16 June 2025; accepted 14 July 2025. Date of publication 22 July 2025; date of current version 14 August 2025. This work was supported in part 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, in part by the DYMAN project funded by the European Union - European Innovation Council under Grant 101161930, in part by the Italian Space Agency (ASI) under Grant 2018-24-HH.0 as part of the Gaia mission, and in part by the INAF minigrant awarded to Dr. Valentina Cesare. Recommended for acceptance by D. Tao. *(Corresponding author: Giulio Malenza.)*

Giulio Malenza, Marco Edoardo Santimaria, Robert Birke, and Marco Aldinucci are with the Department of Computer Science, University of Turin, 10124 Torino, Italy (e-mail: giulio.malenza@unito.it; marcoedoardo.santimaria@unito.it; robert.birke@unito.it; marco.aldinucci@xunito.it).

Valentina Cesare and Ugo Becciani are with the Astrophysical Observatory of Catania, National Institute for Astrophysics, 95123 Catania, Italy (e-mail: valentina.cesare@inaf.it; ugo.becciani@inaf.it).

Alberto Vecchiato is with the Astrophysical Observatory of Turin, National Institute for Astrophysics, 10025 Pino Torinese, Italy (e-mail: alberto.vecchiato@inaf.it).

Digital Object Identifier 10.1109/TPDS.2025.3591452

I. Introduction

THE data produced by scientific experiments coming from different domains are quickly increasing in size, approaching the 10–100PB range. Providing Peta- and ExaFLOP/s of computing power, high-performance computers are the systems on which such amounts of data are usually processed. Today, such compute capacities are achieved using accelerated supercomputers comprising O(10^3) computational nodes having one or more GPUs. The inclusion of GPU accelerators adds a new dimension of heterogeneity that further exacerbates the long-standing problem of code and performance portability ($\mathcal{P}$) across successive generations of supercomputers. Indeed, the top supercomputers are often one-off projects with HPC applications relying on end-to-end hardware/software co-design using low-level code to achieve extreme compute scalability and efficiency. Co-design intercepts thriving trends, such as, hardware functional specialization, the need for power capping, and the expanding HPC application space. Still, it requires suitable abstractions to avoid the explosion of time-to-solution, which is utterly needed by the growing community of industrial supercomputing users.

An alternative method is adapting legacy applications to fully exploit GPU acceleration. While it might not be as effective as co-design for achieving record performance, it can improve the performance of legacy parallel codes. Code adaptation also aims to express some of the application's functional steps as algorithmic patterns that map well into available hardware features. As an example, this kind of code porting and tuning is typical for CUDA [1] and HIP [2], the low-level native programming languages for NVIDIA and AMD GPUs, respectively, which allows programmers to access GPU-specific hardware features, such as scratchpad memory and tensor cores. As for any vendor-specific low-level language, the optimized code is confined to run on specific hardware platforms and might require rewriting and re-tuning for optimal performance on different GPUs, even from the same vendor.

Both end-to-end co-design and vendor-specific low-level languages are useful for producing record-breaking applications. Still, they face sustainability and cost limitations for many applications of significant industrial interest. Indeed, especially for industrial users, the ability to preserve investment in code development and tuning across different generations of computing platforms significantly affects their value chain linked to supercomputers. The advent of GPU-accelerated supercomputers makes this need evermore compelling.

Several programming frameworks aim to decouple the application from specific hardware platforms with no (or limited)

© 2025 The Authors. This work is licensed under a Creative Commons Attribution 4.0 License. For more information, see https://creativecommons.org/licenses/by/4.0/

performance loss. Examples are C++ Parallel Standard Template Library (PSTL) [3], HIP [2], KOKKOS [4], OpenMP with GPU offload [5], OpenACC [6], RAJA [7], OCCA [8], SYCL [9], Alpaka [10], and Fastflow [11]. The use of such frameworks makes performance portability a crucial research topic in HPC [12]. While this term has undergone various definitions [4], [13], [14], at large, it aims to capture *an application's performance efficiency for a given problem that can be executed correctly on all platforms in a given set* [15].

Different from related work studying the performance portability of specific tasks (e.g., [16]) or synthetic benchmarks (e.g., [17]), our study dwells into the performance characteristics of a scientifically and computationally relevant real-world application: the Astrometric Verification Unit–Global Sphere Reconstruction (AVU–GSR) pipeline of the European Space Agency's (ESA) Gaia mission. This pipeline aims to find the astrometric parameters of $\sim 10^8$ primary stars [18] in the Milky Way with a 10–100 micro-arcseconds accuracy [19]. At the core, this requires solving a system of linear equations done via a customized and parallelized version of the iterative LSQR algorithm [20].

The LSQR solver is the crucial computational unit in the pipeline and the object of our study. We port the solver to the HIP, OpenMP with GPU offload, SYCL, OpenACC, and KOKKOS programming framework adding to the previous CUDA [21] and C++ PSTL [22] ports, verify their correctness and study their performance portability using Pennycook's Φ score [15] for both application and architectural efficiency. Application efficiency captures the performance in terms of execution time, whereas architecture efficiency compares to the theoretical hardware performance estimated using the roofline model. Overall, we compute two Φ scores for seven programming frameworks across five hardware platforms and up to three problem sizes.

This work is an extension to paper [23], where the following new points are addressed:

1) We extend the ports of the AVU–GSR LSQR solver to two new frameworks (OpenACC and KOKKOS);
2) We build the roofline model of the six most computationally demanding GPU kernels of the solver for each framework, platform, and considered problem size, comparing the achieved performance to the theoretical peak performance of each GPU architecture.
3) Besides performance portability calculated with application efficiency (Φ_{App}), we report the performance portability with architectural efficiency (Φ_{Arch}), leveraging the roofline model for each solver port, hardware platform and problem size.
4) The low-level ports of the solver, written in CUDA, HIP, and SYCL, are further optimized compared to the correspondent versions in [23].
5) To enhance reproducibility and support further investigation, we open source our code and experiments[1].

[1]https://github.com/alpha-unito/PP-Gaia

II. Background and Related Works

Parallel programming frameworks. Several frameworks have been proposed to ease parallel programming with different ease-of-use versus performance tradeoffs. CUDA [1], HIP [2], and SYCL [9] are low-level specific frameworks, meaning they have explicit run-time functions that users can use to manage memory and write fine-tuned kernels [24]. OpenMP [5], similarly to OpenACC [6], is a directive-based API [25]; this means the user can add `pragma` directives that instruct the compiler to generate assembly code that runs on multicores systems and, eventually, accelerators. Pragma-based programming languages might be easier to use compared to language-specific frameworks as users are not required to re-write the entire application [26], [27], [28], [29]), but allow only limited tuning and memory management.

Another way to execute applications on GPUs is developing code with specific abstraction libraries that translate user code into CUDA, HIP or OpenMP code. Examples of such libraries are KOKKOS [4], RAJA [7], Alpaka [10], Thrust [30], OCCA [8], FastFlow [11], and C++ PSTL [3]. Most noteworthy, C++ PSTL is particularly interesting since it is an open standard that only requires the standard library, completely masking any low-level parallel runtime library. Starting from C++17, standard algorithms can be executed in parallel by specifying the execution policy [31].

Performance portability metric: Performance portability aims to capture two critical aspects: the capability of applications to run across a designated set of heterogeneous hardware platforms and the achieved performance. Various metrics have been used [4], [13], [14]. While a universally accepted metric is still lacking, Φ introduced by Pennycook et al. [15] has rapidly gained traction. Φ is defined as follows:

$$\Phi(a,p,H) = \begin{cases} \frac{|H|}{\sum_{i \in H} \frac{1}{e_i(a,p)}} & \text{if } i \text{ is supported } \forall i \in H \\ 0 & \text{otherwise,} \end{cases} \quad (1)$$

where a is a specific application, p is a given problem that the specific application a computes and $e_i(a,p)$ is the efficiency of the application a for the problem p on a platform i from a given set of H of size $|H|$. In other words, Φ is the harmonic mean of the application's efficiency over the set of platforms H. Φ measures the performance imbalance on different platforms across different code versions, not their absolute efficiency. As such, if the Φ of a specific application across a set of platforms is lower than the Φ across a different set of platforms, it does not necessarily mean that this application has a worse absolute performance on the first set of platforms compared to the second one. Depending on the main performance focus, $e_i(a,p)$ can be either an *application efficiency* capturing the variations in execution time or *architectural efficiency* capturing how well the underlying hardware is exploited (see Section V for details).

Performance portability studies: Several studies address performance portability. Pennycook et al. [15] describe some real-life examples of the usage of the Φ metric. They present the GPU-STREAM benchmark [32], a reimplementation of McCalpin's STREAM benchmark [33], parallelized with eight programming models (McCalpin, SYCL, RAJA, KOKKOS,

OpenMP-C++, OpenACC, CUDA, and OpenCL) evaluated on 12 different platforms, either CPU or GPU-based. Since none of the benchmarks is able to run on all platforms, the Φ is calculated for different sets of code versions and platforms.

Lin et al. [34] test the performance portability of three heterogeneous HPC mini-applications (BabelStream, miniBUDE, and CloverLeaf) on both CPU and GPU platforms using different implementations of the C++17 PSTL. The three mini-applications cover both compute-bound and memory bandwidth-bound cases. The authors prove the ports to be competitive with previous versions written in OpenMP, CUDA, SYCL, and KOKKOS across different platforms, but they avoid evaluating the performance portability of these mini-applications with an objective metric.

Bhattacharya et al. [35] investigate different portable parallel programming models (KOKKOS, SYCL, OpenMP, C++ STL) for high energy physics use cases (FastCaloSim, ACTS, Wire Cell, Patatrack, P2R, and Random Number Generators) highlighting their benefits and challenges. Atif et al. [36] added the Alpaka parallelization language to the previous study. Other works study performance portability from the perspective of the (pre-)Exascale era [37], [38], [39]. Hammond and Mattson [40] evaluated data parallelism in C++ on GPU using the Parallel Research Kernels[2], a set of application skeletons simpler than mini-applications. Such studies only address benchmarks or mini-apps or lack the use of a wider-spread metric. Different from previous works, we use a real-world application using the recognized Φ metric.

III. Gaia Mission and AVU–GSR Code

The Gaia mission was launched on December 19th 2013, and is ending its data collection at the beginning of 2025[3]. Two further data releases are planned, Data Release 4, for mid-2026, and Data Release 5, for end-2030[3], considered as the Gaia mission end date. Gaia mission is producing an extremely accurate astrometric map of more than 10^9 stars in the Milky Way, measuring their parallaxes, right ascension, declination, and proper motions, besides their luminosity, temperature, and composition [41]. The accuracy of the astrometric measurements can go down to the 10–100 micro-arcseconds level or below [41]. High-resolution astrometry is a powerful instrument for exploring essential questions related to our Galaxy's origin, structure, and evolutionary history. High-resolution astrometry, in synergy with high-resolution spectroscopy and photometry, allows to build extremely precise kinematic profiles of our Galaxy (rotation curves and vertical velocity dispersions) [42], [43] and to study the formation and evolution of the Milky Way (e.g., [44], [45], [46]). Moreover, it can shed light on disentangling between General Relativity, the current most explored theory of gravity which assumes that galaxies are surrounded by large dark matter halos, and other theories of modified gravity, which do not resort to dark matter (e.g., [43], [47], [48], [49], [50], [51]).

[2]https://github.com/ParRes/Kernels

[3]https://www.esa.int/Science_Exploration/Space_Science/Gaia

A. The Gaia AVU–GSR Code

The use-case analyzed in this work is the solver module of the AVU–GSR pipeline (Fig. 1) of the Gaia mission. The solver's purpose is to derive, with a 10–100 micro-arcseconds accuracy, the astrometric parameters of the $\sim 10^8$ primary stars in our Galaxy, and the attitude and instrumental settings of the Gaia satellite. This code represents a verification module of the same solution derived with a different algorithm through the Astrometric Global Iterative Solution (AGIS) software [52], [53] crossvalidating the results.

The solver finds these parameters by solving an overdetermined and highly sparse system of linear equations,

$$\mathbf{A} \times \vec{x} = \vec{b}, \tag{2}$$

with a customized and preconditioned version of the iterative LSQR algorithm, up to the LSQR convergence with a least-square condition or to a maximum number of iterations set at runtime. The system is solved under the fully relativistic astrometric models of [54] and [55]. In (2), $\mathbf{A}$ is the large and highly sparse coefficient matrix, with $\sim 10^{11}$ rows, representing the equations of the system and the stellar observations, and $\sim 10^8$ columns, expected at the end of the Gaia mission. $\vec{b}$ and $\vec{x}$ are the known terms and solution arrays, with a number of elements equal to the number of rows and columns of $\mathbf{A}$, and, when complete, they will occupy a maximum memory of ∼800GB and ∼4GB, respectively, being of double-precision type.

The LSQR represents ∼95% of the solver's entire computational time. The most computational intensive operations are two matrix-by vector products: *aprod_1*,

$$\vec{b}^i += \mathbf{A} \times \vec{x}^{i-1}, \tag{3}$$

iteratively estimating the known terms, and *aprod_2*,

$$\vec{x}^i += \mathbf{A}^T \times \vec{b}^i, \tag{4}$$

iteratively estimating the solution.

The coefficient matrix $\mathbf{A}$ is vertically divided into three sections: astrometric, attitude, and instrumental, highlighted in Fig. 2 with different colors. The blue boxes represent the non-zero coefficients of $\mathbf{A}$ (not in scale). The astrometric section in green represents ∼90% of the unknowns and its non-zero coefficients follow a block-diagonal structure, allowing a regular data access pattern. The non-zero coefficients in the attitude section in yellow are divided into three blocks of four elements separated by a stride depending on the considered input dataset. Finally, the non-zero coefficients in the instrumental section in red follow an irregular pattern. In total, we have, at maximum, $N_{\text{Astro}} = 5$, $N_{\text{Att}} = 3$, and $N_{\text{Instr}} = 6$ non-zero coefficients per row.

Having an expected number of double-precision type elements of $\sim 10^{11} \times 10^8$ at the end of the mission, the coefficient matrix is likely to occupy ∼80EB, which implies a system not solvable in human-sized timescales even on post-Exascale infrastructures. Since $\gtrsim$99% of the elements of $\mathbf{A}$ are zeros, we perform calculations with a dense matrix $\mathbf{A}_{\text{r}}$ only containing the non-zero coefficients of $\mathbf{A}$. Therefore, the total number of elements of $\mathbf{A}_{\text{r}}$ is expected to be $\sim 10^{11} \times 24$, reducing the

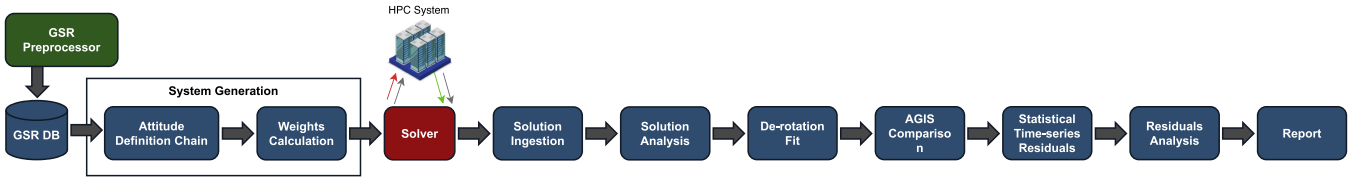


Fig. 1. Description of the pipeline of the AVU–GSR mission code. The solver (in red) is the main computational bottleneck offloaded to an HPC system.

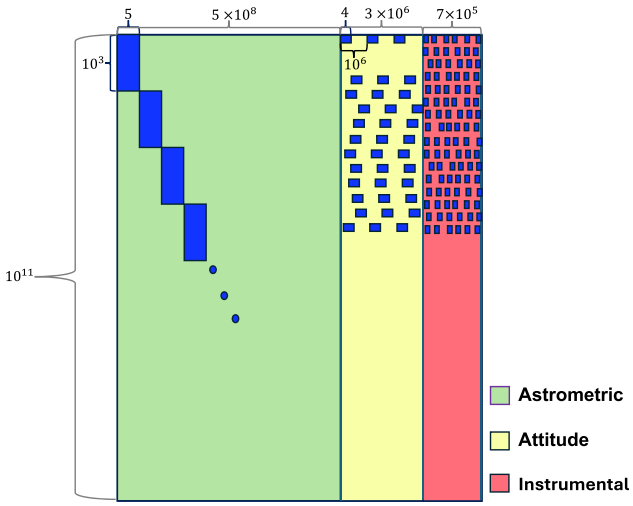


Fig. 2. Structure of matrix $\mathbf{A}$. Non-zero parameters are highlighted as blue blocks.

problem size of ∼7 orders of magnitude and the size of $\mathbf{A}$ to ∼19TB. The matrix $\mathbf{A}$ is further split across multiple nodes to fit each subproblem in the memory available on a single node/GPU.

We stored the reduced matrix $\mathbf{A}_\mathrm{r}$ in memory in a manner that allows as many contiguous data accesses as possible. $\mathbf{A}_\mathrm{r}$ is divided into three submatrices, one for each of the problem sections ($\mathbf{A}_{\mathrm{r,Astro}}$, $\mathbf{A}_{\mathrm{r,Att}}$, and $\mathbf{A}_{\mathrm{r,Instr}}$). Each submatrix has the same number of rows as $\mathbf{A}_\mathrm{r}$ and a number of columns equal to N_{Astro}, N_{Att}, and N_{Instr}. Given the regular block-diagonal structure of the non-zero astrometric coefficients in $\mathbf{A}$, we only need to save the starting column index of the astrometric non-zero elements for each row of $\mathbf{A}$ in an array called `matrixIndexAstro` to recover the original position of the astrometric elements in each row of $\mathbf{A}$. Similarly, the attitude non-zero coefficients follow a regular pattern of three blocks of four elements, each separated by a given stride. To retrieve the original position of the attitude elements in each row of $\mathbf{A}$, we only need to store the index of the first attitude non-zero element in each row of $\mathbf{A}$ in an array named `matrixIndexAtt`. Given the irregular pattern of the instrumental coefficients, we cannot use a similar trick to recover the original position of the instrumental elements. Therefore, we store, for each row of $\mathbf{A}$, the column indexes of all instrumental non-zero elements in an array called `instrCol`.

IV. Parallelization of the Gaia AVU–GSR Solver Code

The Gaia AVU–GSR solver code leverages distributed systems via MPI, where each MPI rank processes a subset of the observations. To fully exploit the compute accelerators, the code is further parallelized by offloading the main arithmetic computations on GPUs using CUDA [21], [56]. In particular, the two most intensive computations are implemented as six kernels: the two matrix-by vector products from (3) and (4) applied on the three submatrices, respectively named as *a1Astro*, *a2Astro*, *a1Att*, *a2Att*, *a1Instr*, and *a2Instr*.

A. Parallel Programming Frameworks

We optimize the original CUDA implementation, and parallelize the LSQR solver using six additional parallel programming frameworks: HIP, SYCL, OpenMP, OpenACC, KOKKOS, and C++ PSTL, in combination of different compilers. Table I presents an overview of all framework-compiler combinations.

1) CUDA: The CUDA version allocates the host variables via *cudaHostMalloc* to use pinned memory to improve performance and use streams. All GPU variables are allocated using *cudaMalloc*. The three submatrices are copied to GPUs asynchronously, using *cudaMemcpyAsync*. The same is done with all relevant quantities, constraints, and other unknowns of the astrometric problem. CUDA streams are used to overlap *aprod 2* kernel computations. It is worth noting that the matrices are copied to the GPU before the main loop and remain there until the end of the algorithm, avoiding costly GPU-CPU data exchanges during loop iterations. We force the same behavior on all the frameworks that allow explicit memory management, i.e., CUDA, HIP, and SYCL. We use NVIDIA's nvcc compiler.

2) HIP: HIP is a programming language designed to be syntactically similar to CUDA. Hence, porting from CUDA is relatively easy. We use HIPIFY to translate our code into HIP, and then manually optimized the code, tuning kernel parameters for AMD and NVIDIA architectures. In particular, *cudaMalloc*, *cudaMemcpyAsync*, and *cudaStreamCreate* are replaced with *hipMalloc*, *hipMemcpyAsync* and *hipStreamCreate*. One thing worth noting is that we allocate the memory forcing coarse-grain coherence using *hipMemAdvise*. This is done for performance reasons as we experimentally observed that fine-grain coherence led to performance degradation due to the atomic operations in the *aprod_2* kernels. We use AMD's hipcc compiler.

3) SYCL: The SYCL implementation uses order queues and Unified Shared Memory allocators. In particular, `malloc_device` allocates data directly on GPU. We used `parallel_for` to declare parallel code regions and `NDrange` to fine-tune the number of threads per block and blocks per grid used by the kernels. We compile SYCL both with AdaptiveCpp (SYCL+A) and Intel's DCPP (SYCL+I) compilers.

TABLE I
FRAMEWORK DETAILS FOR NVIDIA AND AMD

Framew.	Comp.	Compilation Flags (NVIDIA)
CUDA	`nvc++`	`-gencode=arch=compute_XX,code=sm_XX`
HIP	`hipcc`	`-gpu-architecture=sm_XX`

SYCL+A	`acpp`	`-acpp-platform=cuda`
		`-acpp-targets=cuda:sm_XX`
		`-acpp-gpu-arch=sm_XX`

SYCL+I	`clang++`	`-fsycl -fsycl-targets=nvptx64-nvidia-cuda` `-Xsycl-target-backend -cuda-gpu-arch=sm_XX`
OpenACC	`nvc++`	`-acc -gpu=ccXX`
OMP+V	`nvc++`	`-mp=gpu -gpu=ccXX,sm_XX`
OMP+LLVM	`clang++`	`-fopenmp -fopenmp-targets=nvptx64-nvidia-\` `cuda -march=sm_XX`
KOKKOS	`nvcc`	`-offload-arch=sm_XX -fopenmp`

PSTL+A	`acpp`	`-acpp-platform=cuda -acpp-stdpar`
		`-acpp-targets=cuda:sm_XX`
		`-acpp-stdpar-unconditional-offload`
		`-acpp-gpu-arch=sm_XX`

PSTL+V	`nvc++`	`-stdpar=gpu -gpu=ccXX,sm_XX`
Framew.	**Comp.**	**Compilation Flags (AMD)**
HIP	`hipcc`	`-offload-arch=gfxXXX -munsafe-fp-atomics`
SYCL+A	`acpp`	`-acpp-platform=rocm -acpp-targets=hip` `-acpp-gpu-arch=gfxXXX -munsafe-fp-atomics`

SYCL+I	`clang++`	`-fsycl -fsycl-targets=amdgcn-amd-amdhsa`
		`-Xsycl-target-backend -offload-arch=gfxXXX`
		`-mllvm -amdgpu-oclc-unsafe-fp-atomics=true`

OMP+V	`hipcc`	`-fopenmp -offload-arch=gfxXXX` `-munsafe-fp-atomics`
OMP+LLVM	`clang++`	`-fopenmp -fopenmp-targets=x86_64,amdgcn-\` `amd-amdhsa -march=gfxXXX`
KOKKOS	`hipcc`	`-offload-arch=gfxXXX -fopenmp` `-munsafe-fp-atomics`

PSTL+A	`acpp`	`-acpp-platform=rocm -acpp-stdpar`
		`-acpp-targets=hip:gfxXXX`
		`-acpp-stdpar-unconditional-offload`
		`-acpp-gpu-arch=gfxXXX -munsafe-fp-atomics`

PSTL+V	`hipcc`	`-hipstdpar -hipstdpar-path=$(HIPSTDAR_ROOT)` `-offload-arch=gfxXXX -munsafe-fp-atomics`

All cases use additionally `-Std=c++20 -O3`.

4) OpenMP: We use the OpenMP `#pragma omp enter data map` directive to allocate GPU data and the OpenMP `#pragma omp target update` directive to update it. Data are processed on GPU using `#pragma omp target teams distribute parallel for`. Kernels can be fine-tuned by setting the number of teams `num_teams` and the thread limit `thread_limit`. We compile with LLVM (OMP+LLVM) or the vendor's own (OMP+V) compilers.

5) OpenACC: Similarly to OpenMP we use `#pragma acc enter data copyin` and `#pragma acc update host/device` to transfer and update data between CPU and GPU. We parallelize the data loop using the directive `#pragma acc parallel loop gang vector`. Kernels can be fine-tuned by setting the number of gangs (`num_gangs`) and the number of threads within each gang (`vector_length`). We compile the code on NVIDIA GPUs using nvc++.

6) C++ PSTL: The first porting of the code to C++ PSTL was done in [22]. It is worth noting that, in this case, there are no specific tuning directives for the number of threads and blocks. Due to the better performance on AMD, same as with HIP we additionally force coarse-grain coherence using *hipMemAdvise*.

TABLE II
MAIN FEATURES OF PARALLEL FRAMEWORKS

	Low-level	Pragma-based	C++ Abstraction
Kernel Tuning	Manual	Automatic/Manual	Automatic
Memory Management	Manual	Automatic/Manual	Automatic
Shared Memory	Manual	Automatic	Automatic
Synchronization	Manual	Automatic	Automatic
GPU Streams	Available	Not Available	Not Available
Code lines	High	Medium	Low

This implementation is compiled using AdaptiveCpp (PSTL+A) or the vendor specific (PSTL+V) compilers.

7) KOKKOS: KOKKOS code is implemented starting from the creation of `Kokkos::View` to define the arrays needed on the GPUs. Data is copied from CPU to GPU and vice versa using `Kokkos::deep_copy`. KOKKOS `parallel_for` and `parallel_reduce` are used in conjunction with `KOKKOS_LAMBDA` to parallelize the necessary operations. KOKKOS comes with its own compiler.

Table II summarizes key features of the low-level, pragma-based, and C++ abstraction parallel programming models. Low-level programming offers the greatest flexibility but requires significant manual effort for kernel writing, tuning, memory management, shared memory usage, and synchronization. It also provides access to GPU streams, which are not generally available in the higher-level models. Pragma-based frameworks strike a balance between automation and manual control; by default, kernel tuning and memory management are automatically handled, but users can still specify them. However, they do not explicitly expose specific APIs to use shared memory, GPU streams, and synchronization. No GPU kernel code can be directly written; users write code compiled into assembly that can be executed on the GPU. C++ abstraction frameworks aim to maximize programmability and portability by fully automating kernel tuning, memory management, shared memory handling, and synchronization, at the cost of reduced control and flexibility. They are based on general algorithms that can be executed parallel on GPUs. As in the pragma-based case, users can not directly write GPU kernels, but the compiler generates GPU assembly code. They also do not provide direct access to GPU streams. In terms of code complexity, low-level approaches require the highest number of code lines, pragma-based models reduce this burden, and C++ abstractions yield the most concise code.

B. Kernel Parallelization

The *aprod_1* kernels result in multiplications of matrix rows times column vector, which can be easily parallelized assigning rows to threads. Conversely, due to the rectangular structure of $\mathbf{A}^T$, the indexes used by *aprod_2* on the attitude and instrumental submatrices can collide, and hence the updates require atomic operations. To minimize the collision probability, we redesign the code to reduce the number of blocks and GPU threads per block in the regions where atomic operations are performed.

Algorithm 1: *a2Astro* for Pragma-Based Frameworks.

```
#pragma omp target teams distribute parallel for
for ix ← 0 to N_star do
    // Set start and end indexes of M_I,Astro array
    start = startend[ix];
    end = startend[ix + 1];
    // Set local index to astrometric section
    tid = M_I,Astro[start] − offLocalAstro;
    for i ← start to end do
        tmp = b_dev[i];
        // Iterate over non-zero astrometric indices
        for j ← 0 to N_Astro do
            x_dev[tid+j] += A_dev[i × N_Astro + j] × tmp
        end
    end
end
```

Algorithm 2: *a2Astro* for Low-Levels Frameworks.

```
// Threads identification
ix = blockIdx.x * blockDim.x + thIdx.x;
if ix < N_obs then
    // Indexes initialization
    start = startend[ix];
    end = startend[ix + 1];
    tid = M_I,Astro[start];
    // Shared memory initialization
    shared double shSum[TxRow][TxCol][5];
    for j ← 0 to N_Astro do
        shSum[thIdx.x][thIdx.y][j] = 0.0;
    end
    __syncthreads();
    // Calculation of partial results
    for i ← start + thIdx.y to end by TxCol do
        shSum[thIdx.x][thIdx.y][j] =
          A_dev[i × N_Astro + j] × b_dev[i];
    end
    // Parallel reduction
    for offset ← TxCol/2 to 0 by offset ≫ 1 do
        if thIdx.y < offset then
            for j ← 0 to N_Astro do
                shSum[thIdx.x][thIdx.y][jx] +=
                  shSum[thIdx.x][thIdx.y +
                  offset][jx];
            end
        end
        // Writing of final result
        if thIdx.y == 0 then
            for j ← 0 to N_Astro do
                x_dev[tid − offLocalAstro + j] =
                  shSum[thIdx.x][0][j];
            end
        end
    end
end
```

However, this can imply that, in some cases, the GPU occupancy is not maximally exploited. To limit stalling times, we execute the kernels in streams, allowing asynchronous overlap. Since the atomic operations in each submatrix target different subsections of $\vec{x}$, the asynchronous execution of the kernels does not increase the execution cost of the atomic operations. An exception to this is the astrometric submatrix. Due to its block diagonal structure, its *aprod_2* kernel, i.e., *a2Astro*, can be parallelized differently, avoiding using atomic operations. In particular, by introducing some index lists in which the start and end row indexes of two successive star observations are saved, we can assign the set of observations from a given star to a thread and independently execute the matrix by vector products on different stars. Algorithm 1 describes how this kernel was parallelized with pragma-based frameworks. Each star ix is assigned to a thread that computes all the star's matrix-by-vector products saved in the region from $startend[ix]$ to $startend[ix+1]$. With low-level frameworks, we optimize the kernel further (see Algorithm 2) improving coalescing access pattern. Specifically, we further use shared memory to execute all the matrix-by-vector products of a star by a group of threads and assign thread groups to different stars.

V. Evaluation Metrics

Our main evaluation metric is the performance portability defined in [15]. This metric depends on the definition of efficiency. Here, we consider two efficiencies: the application efficiency (e^{App}), which *represents the ability of an application to use the most appropriate implementation and algorithm for each platform* and architectural efficiency (e^{Arch}), *which represents the ability of an application to utilize hardware efficiently* [15]. More in detail, we estimate the architectural efficiency through the roofline model. With these efficiencies, we estimate the performance portability with the application (Φ_{App}) and architectural (Φ_{Arch}) efficiency.

A. Performance Portability With Application Efficiency

e^{App} is the *achieved performance as a fraction of best observed performance* [15]. For each framework, we measure the average iteration time $\bar{t}$ of the LQSR solver, which essentially contains the execution of the six main application kernels on GPU: *a1Astro*, *a1Att*, *a1Instr*, *a2Astro*, *a2Att*, and *a2Instr*. We calculate the application efficiency of each software framework j on each hardware platform i as follows:

$$e^{\text{App}}_{i,j} = \frac{\bar{t}^{\min}_i}{\bar{t}_{i,j}}, \tag{5}$$

where $\bar{t}_{i,j}$ is the average iteration time of framework j on platform i and $\bar{t}^{\min}_i$ is the minimum average iteration time among all the frameworks on platform i. By construction, the most performant framework on a given platform has $e^{\text{App}} = 1$. To calculate the Φ_{App} of framework j over the set of considered platforms H, we plug this application efficiency into (1) summing over all platforms $i = 1, \ldots, |H|$.

TABLE III
GPU ARCHITECTURES CONSIDERED

Attribute	H100	A100	V100	T4	MI250X
GPU	NVIDIA Hopper	NVIDIA Ampere	NVIDIA Volta	NVIDIA Tesla	AMD Instinct CDNA 2
Global Memory	96 GB HBM3	64 GB HBM2e	32 GB HBM2	16 GB GDDR6	64 GB HBM2e (per GCD)
L2 Cache	60 MB	32 MB	6 MB	4 MB	8 MB (per GCD)
Max Clock Rate	1980 MHz	1395 MHz	1597 MHz	1590 MHz	1700 MHz
CUDA Cores/ Compute Units	16896 Cores	7936 Cores	5120 Cores	2560 Cores	110 CU (per GCD)
Memory Clock Rate	2619 MHz	1593 MHz	1107 MHz	5001 MHz	1593 MHz
Bus Width	6144-bit	4096-bit	4096-bit	256-bit	4096-bit
Memory Channels	6	4	4	6	4
Driver Version	545.23.08	530.30.02	550.54.15	550.54.15	ROCm 6.0.3
CUDA/HIP Version	CUDA 12.3	CUDA 12.1	CUDA 12.4	CUDA 12.4	HIP 6.0.32831-204d35d16
Peak memory bandwidth [TB/s]	4.02	1.63	1.13	0.32	1.60
Theoretical fp64 peak perf. [TFLOP/s]	33.5	11.1	8.18	4.09	23.9

B. Performance Portability With Architectural Efficiency

e^{Arch} is the *achieved performance as a fraction of peak theoretical hardware performance*. To calculate the theoretical hardware performance $P^{\text{peak}}_{\text{fp64}}$, we refer to the *Roofline Model* [57]. This model represents the theoretical hardware performance, measured in $FLOP/s$, as a function of the arithmetic intensity AI, measured in $FLOP/byte$. It is typically depicted as a plane, where the x-axis represents AI and the y-axis indicates performance (P). As such, the roofline model of a certain architecture "relates processor performance to off-chip memory traffic and ties together floating-point performance, operational intensity, and memory performance in a 2D graph" (cit. by [58]). The *memory bandwidth roof*, i.e., the limit imposed by the maximum memory bandwidth ($BW_{\max}$) of the considered architecture, is represented by a line passing from the origin in the (AI, P) plane, whose angular coefficient is $BW_{\max}$, measured in $byte/s$ [58]:

$$P^{\text{peak}}_{\text{fp64}} = BW_{\max} \times AI. \tag{6}$$

We calculate the $BW_{\max}$ as [59]:

$$BW_{\max} = \frac{2 \times f^{\text{clock}}_{\text{mem}} \times W_{\text{bus}}}{8}, \tag{7}$$

where $f^{\text{clock}}_{\text{mem}}$ is the memory clock frequency in Hz and W_{bus} is the bus width in bit.

Line (6) intersects for a certain value of AI a horizontal line with equation:

$$P^{\text{peak}}_{\text{fp64}} = N_{\text{cores}} \times f^{\text{clock}}_{\text{core}} \times FLOP^{\text{cycle}}_{\text{core}}, \tag{8}$$

where N_{cores} is the number of cores, $f^{\text{clock}}_{\text{core}}$ is the clock speed, and $FLOP^{\text{cycle}}_{\text{core}}$ is the number of FLOP per core at each clock cycle. The RM sets an upper bound on performance of a kernel according to its AI. If a certain kernel has an AI falling at the left of the crossing point between the *memory bandwidth roof* line (6) and the horizontal line (8), its performance is limited by memory bandwidth of the architecture and its behavior is *memory bound*, which means the code spends more time accessing data from memory than performing arithmetic computations. Conversely, if the AI of the kernel falls at the right of the crossing point, its AI is sufficiently large to saturate the power computation of the architecture, and the kernel becomes *compute bound* [59].

Considering the relative AI of a set of different kernels, spanning from *sparse-matrix-by-vector products* (SpMV) to *N-body problems*, SpMV kernels have the smallest AI [59]. Indeed, the RM of the SpMV kernels in the AVU–GSR code all fall at the left of the cross point.

Table III reports the $BW_{\max}$ of each GPU considered in our study, given by (7), along with the theoretical fp64 peak performance $P^{\text{peak}}_{\text{fp64}}$ given by (8). In our experiments, we only used 1 of the 2 Graphics Compute Die (GCD) of the MI250X, halving the theoretical maximum bandwidth and peak performance.

Calculating the AI of an entire LSQR iteration is not trivial because there is not one kernel but multiple kernels interleaved by minor CPU computations. Since the sum of the times of the six main GPU kernels is $\geq 89\%$ of the average LSQR iteration time, we calculate the AI_k of each GPU kernel k. To calculate AI_k, we count the number of arithmetic operations performed in each kernel k among double precision arrays ($FLOP_k$), and we divide $FLOP_k$ by the number of accesses to each array multiplied by 8 to account for the bytes needed to store fp64 values. Since SpMV kernels are *memory bound*, to calculate the theoretical peak performance $P^{peak}_{fp64,k}$ of each GPU kernel k, we evaluate (6) for the calculated AI_k:

$$P^{\text{peak}}_{\text{fp64},k} = BW_{\max} \times AI_k. \tag{9}$$

We compute the measured peak performance P_k of each kernel k as the calculated $FLOP_k$ divided by the kernel execution time (t_k):

$$P_k = \frac{FLOP_k}{t_k}. \tag{10}$$

On NVIDIA platforms, we verify the consistency between the calculated AI_k and P_k of each kernel k with the values obtained via NVIDIA Nsight Compute profiler[4]. For each considered platform i, we calculate the architectural or roofline efficiency of each kernel k and each framework j as:

$$e^{\text{Arch}}_{i,j,k} = \frac{P_{i,j,k}}{P^{peak}_{\text{fp64},i,k}}. \tag{11}$$

To obtain a unique estimation of architectural efficiency per framework j on platform i, we combine the architectural efficiencies of the six different kernels with this formula:

$$e^{\text{Arch}}_{i,j} = \frac{\sum_{k=0}^{6} w_{i,j,k} e_{\text{Arch},i,j,k}}{\sum_{k=0}^{6} w_{i,j,k}}, \tag{12}$$

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

TABLE IV
APPLICATION AND ARCHITECTURAL EFFICIENCIES ACROSS ARCHITECTURES EXPRESSED AS PERCENTAGES

	Application Efficiency (Φ_{App}) [%]						Architectural Efficiency (Φ_{Arch}) [%]					
	NVIDIA + AMD			NVIDIA			NVIDIA + AMD			NVIDIA		
Name	10 GB	30 GB	60 GB	10 GB	30 GB	60 GB	10 GB	30 GB	60 GB	10 GB	30 GB	60 GB
CUDA	0.00±0e-0	0.00±0e-0	0.00±0e-0	100.00±2.7e-2	100.00±7.8e-2	99.91±7.6e-3	0.00±0e-0	0.00±0e-0	0.00±0e-0	33.52±1.0e-2	36.57±1.8e-2	41.65±3.2e-3
HIP	99.85±8.1e-2	99.76±1.2e-1	99.56±1.6e-1	99.13±2.8e-2	99.04±9.4e-2	98.84±8.5e-3	24.13±3.6e-2	23.68±2.9e-2	22.44±3.7e-2	33.44±5.6e-3	36.48±2.1e-2	41.61±2.7e-3
SYCL+A	95.71±8.9e-2	96.77±1.1e-1	96.85±1.1e-1	93.85±7.1e-2	94.80±8.2e-2	94.46±1.3e-2	23.88±2.9e-4	23.47±2.4e-2	22.22±1.8e-2	32.49±9.8e-3	35.38±1.8e-2	39.86±6.6e-3
SYCL+I	91.83±8.3e-2	92.20±9.0e-2	92.20±1.3e-1	92.42±8.1e-2	93.40±5.4e-2	94.67±2.9e-2	22.05±3.0e-3	21.44±5.7e-3	20.17±4.3e-3	31.55±5.5e-3	34.52±1.5e-2	39.97±1.4e-2
OpenACC	0.00±0e-0	0.00±0e-0	0.00±0e-0	82.00±3.7e-2	79.18±6.7e-2	85.61±2.5e-2	0.00±0e-0	0.00±0e-0	0.00±0e-0	26.90±9.8e-3	27.99±1.1e-2	35.10±1.1e-2
OMP+V	78.37±1.5e-1	76.44±7.4e-2	79.50±1.0e-1	78.34±1.4e-1	76.12±6.1e-2	81.46±3.9e-2	19.08±1.4e-2	18.06±1.7e-2	17.56±2.3e-2	26.38±3.1e-2	27.58±1.2e-2	34.58±1.4e-2
OMP+LLVM	64.59±2.7e-1	62.71±6.4e-2	68.04±1.0e-1	65.28±2.1e-1	63.23±4.4e-2	72.57±8.8e-2	15.95±8.1e-3	14.62±2.9e-3	14.58±1.7e-3	22.86±1.1e-2	22.88±9.0e-3	31.19±1.1e-2
KOKKOS	50.67±3.5e-2	52.54±4.7e-2	55.73±8.6e-2	53.07±1.5e-2	57.90±3.1e-2	68.34±2.1e-2	11.00±3.3e-3	10.86±1.7e-3	10.62±1.4e-3	16.21±4.7e-3	19.51±5.8e-3	28.52±9.1e-3
PSTL+A	50.60±1.8e-1	52.33±9.7e-2	53.42±3.2e+0	53.55±3.0e-2	59.35±6.5e-2	65.43±2.4e-2	10.73±6.0e-2	10.67±2.2e-2	10.20±8.3e-1	15.92±8.4e-3	20.42±1.5e-2	27.41±1.1e-2
PSTL+V	46.10±7.5e-1	51.40±1.8e-1	55.33±5.4e-1	45.00±1.7e-2	55.84±2.9e-2	68.25±1.2e-2	10.88±2.6e-1	10.91±8.1e-2	10.48±1.4e-1	13.96±4.2e-3	19.36±3.3e-3	28.06±3.7e-3

where the weights $w_{i,j,k}$ are given by:

$$w_{i,j,k} = \frac{t_{i,j,k}}{t_{\mathrm{AvIter},i,j}}, \tag{13}$$

and $t_{i,j,k}$ is the execution time of kernel k for framework j on platform i. To calculate Φ_{Arch} of each framework j across the set H of employed platforms, we use (1) with $e_{\mathrm{Arch},i,j}$ (12) at denominator, summing over the considered platforms, where i goes from 1 to $|H|$.

C. Error Propagation

We calculate the error on Φ by applying the error propagation formula on a quantity $a(\{b\})$ depending on a set of N parameters with error $\{b\}$:

$$\sigma_{a(\{b\})} = \sqrt{\sum_{i=1}^{N} \left(\frac{\partial a}{\partial b_i}\right)^2 \sigma_{b_i}^2}, \tag{14}$$

where $\sigma_{a(\{b\})}$ is the error on $a(\{b\})$, $\frac{\partial a}{\partial b_i}$ is the partial derivative of a with respect to b_i, and σ_{b_i} is the error on b_i. In our case, quantity a and parameters b_i are Φ and e_i in (1), and (14) becomes:

$$\sigma_{\Phi} = \frac{|H|}{\left(\sum_{l=1}^{|H|} \frac{1}{e_l}\right)^2} \sqrt{\sum_{i=1}^{|H|} \left(\frac{\sigma_{e_i}}{e_i^2}\right)^2} \tag{15}$$

where e_i can be either the application (5) or the architectural (12) efficiency (the framework index j is omitted for simplicity). The error on e_i is in turn derived with (14), by propagating the errors on the only quantities with errors on which e_i depends, i.e., the LSQR iteration and kernel times, entering in (5), (11), and (12).

VI. RESULTS

We test the seven framework implementations on the five platforms detailed in Table III, using three subproblem sizes, 10, 30, and 60 GB, to adapt to the available GPU memory. Specifically, we execute the 10 GB tests on all platforms, the 30 GB tests on all platforms but T4, and the 60 GB tests on all platforms but T4 and V100 due to GPU memory limits. Each run is executed on a single GPU, i.e., a single GCD in the case of the MI250X. For each test, we measure the execution time of the six main compute kernels (see Section IV) and of the whole LSQR iteration. Φ_{App} uses the application efficiency derived from the iteration times and (5). Φ_{Arch} leverages the FLOP and execution times of each kernel versus their theoretical peak performance using (11) and (12). We average all measure times across 100 iterations and repeat each test ten times for statistical significance. We use the mean and standard deviation across the ten runs to compute Φ_{App} and Φ_{Arch} and their estimation errors.

A. Performance Portability

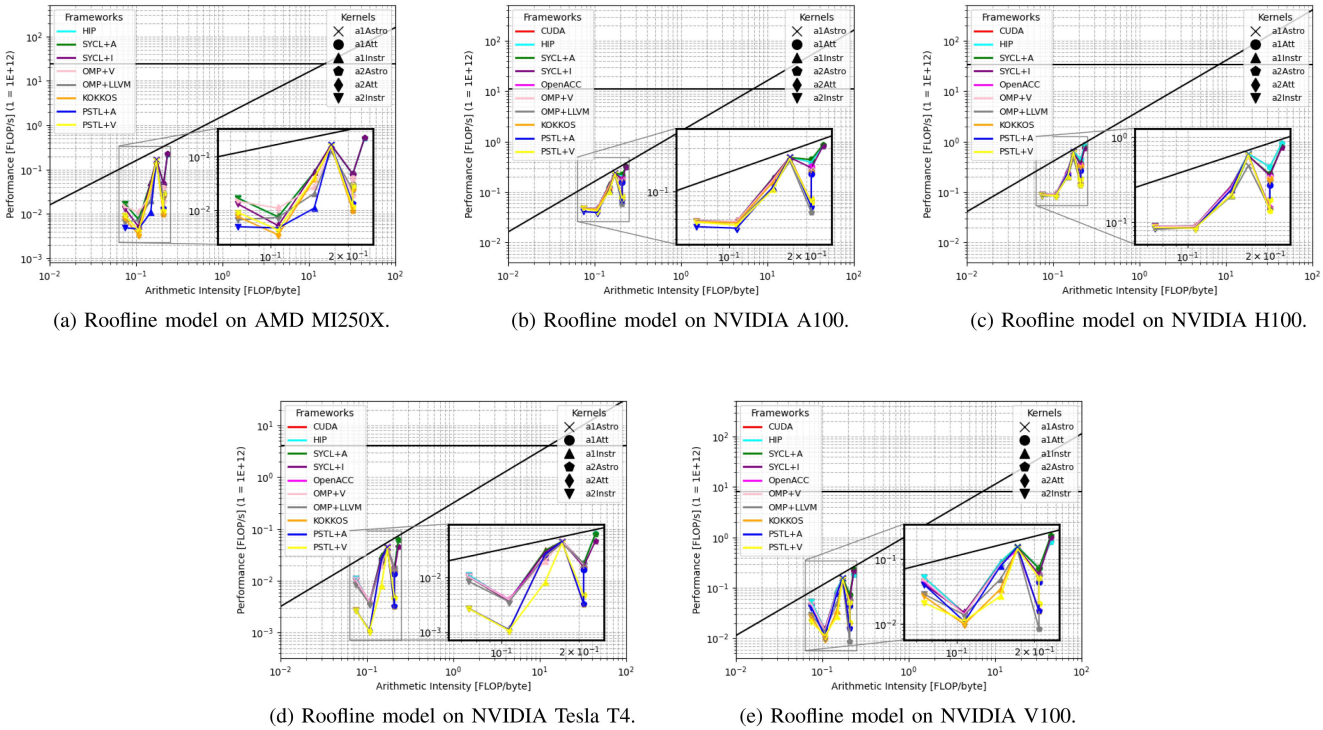
We calculate Φ_{App} and Φ_{Arch} for all (NVIDIA+AMD) and only NVIDIA platforms, i.e., excluding MI250X.

The values of Φ_{App} and Φ_{Arch} are summarized in Table IV, from which we can draw some general considerations. Note that CUDA cannot be executed on platforms different from NVIDIA; hence, when considering also AMD GPUs, CUDA's Φ_{Arch} and Φ_{App} are zero by definition.

1) Low-Level Frameworks: These frameworks consistently achieve the highest Φ scores across all problem sizes, Φ types, GPU architectures, and compilers. Considering only NVIDIA platforms, CUDA is the most performant. The maximum Φ_{Arch} (41.65%) is obtained on a problem size of 60 GB, while the minimum is on the problem of 10 GB (33.52%). Φ_{App} is ∼100% for all problem sizes.

HIP and SYCL also provide Φ values comparable to CUDA. For HIP, the minimum and maximum Φ_{Arch} values are 22.44% (60 GB, all platforms) and 41.61% (60 GB, NVIDIA platforms). Instead, the minimum and maximum Φ_{App} values are 98.84% (60 GB, NVIDIA platforms) and 99.85% (10 GB, all platforms). Differently from CUDA, HIP is also portable to AMD GPUs but not to Intel ones.

SYCL is the most portable, including Intel GPUs. Its performance is comparable across different compilers: generally, the highest Φ_{App} and Φ_{Arch} values are achieved with the AdaptiveCpp compiler (SYCL+A) but they overcome the correspondent values obtained with DPC++ compiler (SYCL+I) by at most 4%. Considering all platforms, the maximum performance is obtained with the problem size of 10 GB for Φ_{Arch} (23.88%) and with the problem size of 60 GB for Φ_{App} (96.85%) by SYCL+A. Considering NVIDIA platforms alone, the maximum performance is obtained with the problem size of 60 GB for both Φ_{Arch} (39.97%) by SYCL+I and with the problem size of 30 GB for Φ_{App} (94.80%) by SYCL+A. We set the AdaptiveCpp compiler to use HIP and CUDA backends on AMD and NVIDIA GPUs, respectively. Therefore, the performance is similar to CUDA and HIP codes.



(a) Roofline model on AMD MI250X. (b) Roofline model on NVIDIA A100. (c) Roofline model on NVIDIA H100.

(d) Roofline model on NVIDIA Tesla T4. (e) Roofline model on NVIDIA V100.

Fig. 3. Roofline model of each of the six main GPU kernels for all the frameworks for the 10 GB problem sizes. Different colors identify different frameworks and different symbols identify different kernels, as indicated in the legends.

2) Pragma-Based Frameworks: Frameworks using pragmas generally have intermediate Φ scores across all problem sizes, Φ types, GPU architectures, and compilers. Among these frameworks, OpenACC has higher Φ scores than OpenMP on NVIDIA platforms, but it is not supported on AMD platforms. On NVIDIA platforms, Φ_{Arch} maximum and minimum values are achieved on problem sizes of 60 and 10 GB and Φ_{App} maximum and minimum values are achieved on problem sizes of 60 and 30 GB.

Considering OpenMP, the highest values are achieved using vendor compilers (OMP+V), i.e., nvc++ on NVIDIA and amdclang++ on AMD, while the LLVM compiler shows some overhead.

3) Abstraction-Based Frameworks: Frameworks using abstraction libraries generally have the lowest Φ scores across the different problem sizes, Φtypes, GPU architectures, and compilers due to their higher overhead. Among these frameworks, PSTL+V has the worst Φ on NVIDIA-only architectures for 10 and 30 GB sizes. When including also AMD architectures, its Φ scores are comparable to KOKKOS and PSTL+A. This is due to the different number of GPU threads used by the compilers. Specifically, nvc++ spans 256 threads per block, while PSTL+A and KOKKOS use 128 threads per block. In the case of our application, a smaller number of threads per block achieves better performance due to better cache behavior.

4) Φ_{App} versus Φ_{Arch}: Performance portability values computed with architectural efficiency are lower than those obtained using application efficiency, reflecting that the entire application is not able to exploit full GPU peak performance. Considering all the platforms, the Φ_{Arch} values are almost identical across problem sizes, while on NVIDIA platforms, the values increase with the size, reflecting a better utilization of the GPU cache levels hierarchies. We deepen the analysis in the next section, describing the roofline model results.

B. Roofline Analysis

We compute the roofline model of each kernel for each framework, platform, and problem size, using the methodology described in Section V-B. Fig. 3 shows the roofline models for the 10 GB problem size on all five platforms. The oblique and horizontal black lines represent the maximum performance in $\log_{10} - \log_{10}$ space for a given arithmetic intensity stemming from (6) and the platforms' peak performance from Table III. The points identifying the measured performance of the six different kernels are zoomed in on the inner panel of each figure. Based on the position of the points along the x-axis, one can see that all six kernels are memory bound, i.e., limited by the oblique black line. The roofline models for the 30 GB and 60 GB problem sizes are qualitatively similar, and their respective plots can be found in the open repository[1]. Using the roofline model values, we computed the roofline efficiencies using (11). Table V shows the results averaged among all platforms for the 10 GB problem size as it is the only one allowing execution on all considered GPUs. The results for the other problem sizes follow the same trends.

The best-performance tuning of the number of threads per block was empirically determined for low-level frameworks on GH200 and MI250X platforms. The tuning on the other NVIDIA platforms was aligned to GH200 to prevent a

TABLE V
AVERAGE ROOFLINE EFFICIENCIES 10 GB PROBLEM

NVIDIA+AMD	CUDA	HIP	SYCL+A	SYCL+I	OpenACC	OMP+V	OMP+LLVM	KOKKOS	PSTL+A	PSTL+V
a1Astro	–	0.800	0.789	0.781	–	0.764	0.671	0.785	0.795	0.760
a1Att	–	0.374	0.362	0.316	–	0.303	0.283	0.267	0.252	0.142
a1Instr	–	0.488	0.460	0.466	–	0.419	0.337	0.367	0.385	0.254
a2Astro	–	0.784	0.809	0.743	–	0.154	0.144	0.107	0.104	0.142
a2Att	–	0.150	0.150	0.147	–	0.148	0.144	0.118	0.114	0.118
a2Instr	–	0.378	0.345	0.340	–	0.361	0.278	0.231	0.248	0.218
NVIDIA	**CUDA**	**HIP**	**SYCL+A**	**SYCL+I**	**OpenACC**	**OMP+V**	**OMP+LLVM**	**KOKKOS**	**PSTL+A**	**PSTL+V**
a1Astro	0.846	0.846	0.830	0.822	0.808	0.817	0.723	0.836	0.840	0.806
a1Att	0.431	0.431	0.415	0.359	0.361	0.343	0.332	0.314	0.293	0.155
a1Instr	0.557	0.554	0.519	0.526	0.493	0.469	0.398	0.418	0.471	0.281
a2Astro	0.821	0.821	0.853	0.771	0.182	0.178	0.162	0.126	0.120	0.169
a2Att	0.177	0.177	0.176	0.177	0.177	0.176	0.169	0.143	0.136	0.141
a2Instr	0.437	0.438	0.396	0.397	0.408	0.413	0.334	0.272	0.301	0.255

platform-dependent behaviour of the code, reducing portability. We aligned the tuning of pragma-based frameworks to the tuning of low-level frameworks (except for the a2Astro kernel) after verifying a better performance than the tuning forced by the compiler. A manual setting of the tuning was not possible on C++ abstraction-based frameworks (Table II). We describe below, kernel by kernel, the horizontal comparison of the roofline efficiencies among the different frameworks.

1) a1Astro: This kernel, by construction, has the most efficient memory access pattern due to contiguous accesses. Hence, it has the highest roofline efficiency with average values of ~ 0.77 on all platforms and ~ 0.82 on NVIDIA alone. The roofline efficiencies of all the frameworks, except for OMP+LLVM, vary in a range $\leq 5\%$, considering both NVIDIA and all architectures. OMP+LLVM shows less performance, probably due to a different number of blocks of threads and a higher register usage. Specifically, while all the other frameworks generate a number of blocks to cover the entire computation, OMP+LLVM limits this number to 3200, hence forcing the reuse of the same block of threads. Moreover, from profilers, we notice that OMP+LLVM uses more registers compared to other frameworks, not only for this kernel but also for other kernels, increasing register pressure.

2) a1Att: Attitude kernel has a more irregular memory pattern than a1Astro, producing more tuning-sensitive results. CUDA and HIP obtained the best performance, achieving a roofline efficiency of 0.43 on NVIDIA architectures, followed by SYCL+A (0.42). SYCL+I and OpenACC achieve a roofline efficiency of 0.36, due to fewer L1 cache hits, compared to CUDA and HIP on NVIDIA T4 and V100. From profilers, we notice that SYCL+I uses, on average, 20 registers more than CUDA/HIP on T4 and V100.

Pragma-based frameworks perform better than C++ abstraction ones due to kernel tuning. On pragma-based, we set 16 threads per block as in the low-level frameworks, while C++ abstraction creates blocks of 128 and 256 threads for KOKKOS/PSTL+A and PSTL+V, respectively. From profilers, we noticed that this setting mainly affects the L1 cache hit, degrading performance compared to low-level and pragma-based.

3) a1Instr: Instrumental kernel has a memory pattern similar to attitude kernel and, consequently, the values of the roofline efficiency are aligned with those of a1Att. As above, for low-level and pragma-based, we set the number of threads per block to 16 while by default C++ abstraction sets it to 128, except for PSTL+V (256).

4) a2Astro: Originally, this kernel used atomic operations to avoid data races. We remove them, rearranging the code according to Algorithms 1 and 2. Roofline efficiencies achieved by low-level frameworks are ~ 0.78 and ~ 0.82 on all and only NVIDIA architectures, respectively. Shared memory allows creating two-dimensional thread blocks that access contiguous data coalesced, improving the performance. Performance of the other parallel frameworks drastically drops due to high-uncoalesced access performed by each thread. Among them, pragma-based performs slightly better than C++ abstraction, due to a different number of threads per block (16) and blocks (2500). PSTL+V performs slightly better than KOKKOS and PSTL+A. This is due to a higher L1 cache hit rate on T4 and V100 GPUs. PSTL+V uses 256 threads and 79 blocks, while KOKKOS and PSTL+A use 128 threads and 313 blocks across different problem sizes on NVIDIA GPUs.

5) a2Att: Contrary to a2Astro, atomic operations cannot be easily avoided on this kernel. Generally, this slows down the performance. The kernel achieves an average roofline efficiency of ~ 0.16 on NVIDIA architectures. A similar value is obtained by adding the MI250X GPU, where we used the flag `-munsafe-fp-atomics` to force the compiler to emit hardware-based FP atomics, improving performance. Unlike a1Att, we notice a similar L1 hit rate across different architectures for both low-level and pragma-based, resulting in similar performance. The type and the frequency of atomic operations are crucial for performance on GPUs. The SYCL code performs atomic operations using `atomic_ref`, which allows specifying four template parameters: the data type, the DefaultOrder, the DefaultScope, and the AddressSpace. While AdaptiveCpp maintains performance under different settings, DPC++ is sensitive to the choice of DefaultScope. For all tested problem sizes, configuring DefaultScope as `work_group` on NVIDIA H100 and A100 results in a performance degradation of approximately 30% compared to using `device`, whereas performance remains stable on T4 and V100. On the AMD MI250, the performance drop is around 20%. The same performance degradation is noticed on the a2Instr kernel. Moreover, on NVIDIA GPUs, with DefaultScope set to `device`, the performance of DCP++ is in line with the performance of AdaptiveCpp even if the generated code uses differ-

ent types of atomic operations: `atom.global.add.f64` for DPC++ versus `atom.cas.b64` for AdaptiveCpp. On AMD GPUs, the generated assembly contains placeholders, making it difficult to map it to a specific atomic type.

6) a2Instr: This kernel has a similar pattern to the previous one, exhibiting a similar behaviour. Also in this case, we set threads per block to 16 for the low-level and pragma-based frameworks, keeping default compiler values for the C++ abstractions.

Summarizing the observations, it is evident that a more contiguous data access pattern, as in the case of a1Astro, makes the kernel less sensitive to tuning, resulting in smaller performance differences across frameworks. Conversely, when the memory access pattern is irregular, as observed in a1Att and a1Instr, tuning has a greater impact, leading to more pronounced performance disparities and favoring low-level and pragma-based frameworks. In a2Astro, the use of shared memory enables memory coalescing, which significantly enhances the performance of low-level frameworks compared to others. As expected, in a2Att and a2Instr, the presence of atomic operations leads to performance degradation regardless of the framework used.

VII. Key Insights

Kernel tuning is crucial for enhancing performance when memory access patterns are irregular. As a result, programming models that allow it, pragma-based and low-level, perform better. As expected, a regular memory access pattern yields better and more stable performance, and the difference in performance between programming models is almost negligible, even when using high-level C++ abstractions. The use of atomic operations inside a kernel has a significant impact: depending on their frequency and implementation at the hardware level, they can cause significant slowdowns. The explicit use of shared memory offered by low-level frameworks can provide the opportunity to avoid atomic operations and improve memory coalescing, resulting in better performance, as seen in the revised a2Astro kernel (Algorithm 2). However, the resulting code is more verbose and potentially more error-prone. The performance portability metric effectively captures the effects of kernel tuning, memory access patterns, and atomic operations on the different programming models, as it reflects both application-level behaviour and architectural efficiency. Moreover, the performance portability computed using architectural efficiency is lower than the values calculated with application efficiency. This indicates that the application does not fully utilize the GPU's peak performance capabilities.

VIII. Conclusions and Future Works

Given the steeply rising trend of data produced from modern scientific experiments and their consequent need to use HPC facilities, performance portability Φ of software is becoming increasingly impelling. As a test case, we analyse the performance portability of a real application: the solver module of the AVU–GSR pipeline of the Gaia mission. This application requires sparse matrix by vector multiplications to cope with the problem size. We implement the code in seven different parallel programming frameworks, three of which are compiled with two different compilers.

We measure the Φ of each code version, considering both the application efficiency Φ_{App} and the architectural efficiency Φ_{Arch} across five GPU platforms –four NVIDIA architectures and one AMD architecture– and three problem sizes –10, 30, and 60 GB.

Generally, low-level programming frameworks achieve the best performance by tuning kernels and using features not used or masked in the other parallelization frameworks, e.g., shared-memory. CUDA is less portable than SYCL and HIP because it is tied to NVIDIA GPUs. HIP and SYCL achieve performances similar to CUDA on NVIDIA and AMD GPUs. Pragma-based frameworks achieve intermediate Φ scores. Among them, OpenACC scores the highest, but it is only partially supported on AMD GPUs. Abstraction-based frameworks score the lowest Φ due to their higher overheads and limitations in kernel tuning optimizations. We also measure the roofline model of the six most computing-demanding GPU kernels for each code version, platform, and problem size. These kernels are representative of the entire application since the sum of their execution time is between 90% and 99% of the LSQR execution time. We confirm that the structure of the matrix is crucial for performance. The astrometric section of the matrix having the most regular data pattern achieves the closest to peak theoretical performance in every case. For the language-specific frameworks, its transpose also achieves similar performance when leveraging shared memory usage, which is not available in pragma- and abstraction-based frameworks. The attitude and instrumental sections achieve the lowest performance due to their irregular structure, which worsens even further when transposed due to the necessity of using atomic operations to avoid data races.

In future studies, we plan to extend this work to other GPU architectures and other programming frameworks to complete the performance portability investigation of this application and other LSQR-based codes based on SpMV algorithms.

Acknowledgment

The authors acknowledge the EuroHPC Joint Undertaking for awarding us access to LUMI at CSC, Finland, and Leonardo at CINECA, Italy. We acknowledge the Pawsey Supercomputing Research Center and the High-Performance Computing for Artificial Intelligence (HPC4AI) laboratory at the University of Turin for providing high-performance computing resources, support, and collaboration.

References

[1] D. Kirk, "NVIDIA CUDA software and GPU parallel computing architecture," in *Proc. 6th Int. Symp. Memory Manage.*, 2007, pp. 103–104.

[2] N. Kerscher, "Investigating the HIP programming model with regards to portability and performance portability," 2022. [Online]. Available: https://events.gwdg.de/event/243/contributions/506/attachments/142/179/FINAL_Seminar_on_Performance_Portable_Programming_of_HPC_Applications_Topic_HiP.pdf

[3] C. DeLozier, "GPU acceleration for the C++ standard template library," 2012. [Online]. Available: https://api.semanticscholar.org/CorpusID:1464773

[4] H. CarterEdwards, C. R. Trott, and D. Sunderland, "Kokkos: Enabling manycore performance portability through polymorphic memory access patterns," *J. Parallel Distrib. Comput.*, vol. 74, pp. 3202–3216, 2014.

[5] J. Huber et al., "Efficient execution of OpenMP on GPUs," in *Proc. 2022 IEEE/ACM Int. Symp. Code Gener. Optim.*, 2022, pp. 41–52.

[6] J. A. Herdman, W. P. Gaudin, O. Perks, D. A. Beckingsale, A. C. Mallinson, and S. A. Jarvis, "Achieving portability and performance through OpenACC," in *Proc. 1st Workshop Accel. Program. Using Directives*, 2014, pp. 19–26.

[7] D. A. Beckingsale et al., "RAJA: Portable performance for large-scale scientific applications," in *Proc. 2019 IEEE/ACM Int. Workshop Perform. Portability Productiv. HPC*, 2019, pp. 71–81.

[8] D. S. Medina, A. St-Cyr, and T. Warburton, "OCCA: A unified approach to multi-threading languages," 2014, *arXiv:1403.0968*.

[9] B. Johnston, J. S. Vetter, and J. Milthorpe, "Evaluating the performance and portability of contemporary SYCL implementations," in *Proc. IEEE/ACM Int. Workshop Perform. Portability Productiv. HPC*, 2020, pp. 45–56.

[10] E. Zenker et al., "Alpaka – An abstraction library for parallel Kernel acceleration," in *Proc. 2016 IEEE Int. Parallel Distrib. Process. Symp. Workshops*, 2016, pp. 631–640.

[11] M. Aldinucci, S. Ruggieri, and M. Torquati, "Porting decision tree algorithms to multicore using FastFlow," in *Proc. Joint Eur. Conf. Mach. Learn. Knowl. Discov. Databases*, 2010, pp. 7–23.

[12] A. Dubey et al., "Performance portability in the exascale computing project: Exploration through a panel series," *Comput. Sci. Eng.*, vol. 23, no. 5, pp. 46–54, Sep./Oct. 2021.

[13] W. Zhu, Y. Niu, and G. Gao, "Performance portability on EARTH: A case study across several parallel architectures," *Cluster Comput.*, vol. 10, pp. 115–126, 2007.

[14] S. McIntosh-Smith, M. Boulton, D. Curran, and J. Price, "On the performance portability of structured grid codes on many-core computer architectures," in *Proc. Int. Supercomputing Conf.*, 2014, pp. 53–75.

[15] S. Pennycook, J. Sewall, and V. Lee, "Implications of a metric for performance portability," *Future Gener. Comput. Syst.*, vol. 92, pp. 947–958, 2019.

[16] V. R. Pascuzzi and M. Goli, "Achieving near-native runtime performance and cross-platform performance portability for random number generation through SYCL interoperability," in *Proc. Int. Workshop Accel. Program. Using Directives*, 2022, pp. 22–45.

[17] S. Boehm, S. Pophale, V. G. Vergara Larrea, and O. Hernandez, "Evaluating performance portability of accelerator programming models using SPEC ACCEL 1.2 benchmarks," in *Proc. Int. Conf. High Perform. Comput.*, 2018, pp. 711–723.

[18] A. Vecchiato et al., "The global sphere reconstruction (GSR). Demonstrating an independent implementation of the astrometric core solution for Gaia," *Astron. Astrophys.*, vol. 620, 2018, Art. no. A40.

[19] U. Becciani, E. Sciacca, M. Bandieramonte, A. Vecchiato, B. Bucciarelli, and M. G. Lattanzi, "Solving a very large-scale sparse linear system with a parallel algorithm in the Gaia mission," in *Proc. Int. Conf. High Perform. Comput. Simul.*, 2014, pp. 104–111.

[20] C. C. Paige and M. A. Saunders, "LSQR: An algorithm for sparse linear equations and sparse least squares," *ACM Trans. Math. Softw.*, vol. 8, pp. 43–71, 1982.

[21] V. Cesare et al., "The MPI + CUDA Gaia AVU-GSR parallel solver toward next-generation Exascale infrastructures," *Pub. Astronomical Soc. Pacific*, vol. 135, 2023, Art. no. 074504.

[22] G. Malenza, V. Cesare, M. Aldinucci, U. Becciani, and A. Vecchiato, "Toward HPC application portability via C++ PSTL: The Gaia AVU-GSR code assessment," *J. Supercomputing*, vol. 80, pp. 14369–14390, 2024.

[23] G. Malenza et al., "Performance portability via C++ PSTL, SYCL, OpenMP, and HIP: The Gaia AVU-GSR case study," in *Proc. Workshops Int. Conf. High Perform. Comput. Netw. Storage Anal.*, 2024, pp. 1152–1163.

[24] J. Nickolls, I. Buck, M. Garland, and K. Skadron, "Scalable parallel programming with CUDA," in *Proc. ACM SIGGRAPH Classes*, 2008, Art. no. 16.

[25] T. Deakin and T. G. Mattson, *Programming Your GPU With OpenMP: Performance Portability for GPUs*. Cambridge, MA, USA: MIT Press, 2023.

[26] R. Reyes, I. López, J. J. Fumero, and F. de Sande, "Directive-based programming for GPUs: A comparative study," in *Proc. IEEE 14th Int. Conf. High Perform. Comput. Commun. IEEE 9th Int. Conf. Embedded Softw. Syst.*, 2012, pp. 410–417.

[27] V. Cesare, I. Colonnelli, and M. Aldinucci, "Practical parallelization of scientific applications," in *Proc. 28th Euromicro Int. Conf. Parallel Distrib. Netw.-Based Process.*, 2020, pp. 376–384.

[28] M. Aldinucci et al., "Practical parallelization of scientific applications with OpenMP, OpenACC and MPI," *J. Parallel Distrib. Comput.*, vol. 157, pp. 13–29, 2021.

[29] J. Đukić and M. Mišić, "An evaluation of directive-based parallelization on the GPU using a parboil benchmark," *Electronics*, vol. 12, 2023, Art. no. 4555.

[30] N. Bell and J. Hoberock, "Thrust: A productivity-oriented library for CUDA," in *GPU Computing Gems Jade Edition*, W.-M. Hwu, Ed. Amsterdam, The Netherlands: Elsevier, 2012, ch. 26.

[31] A. Williams, *C++ Concurrency in Action*. Manning, 2019. [Online]. Available: https://books.google.it/books?id=BzgzEAAAQBAJ

[32] T. Deakin, J. Price, M. Martineau, and S. McIntosh-Smith, "GPU-STREAM v2.0: Benchmarking the achievable memory bandwidth of many-core processors across diverse parallel programming models," in *Proc. Int. Conf. High Perform. Comput.*, 2016, pp. 489–507.

[33] J. D. McCalpin, "Memory bandwidth and machine balance in current high performance computers," *IEEE Comput. Soc. Tech. Committee Comput. Archit. Newslett.*, vol. 2, pp. 19–25, 1995.

[34] W.-C. Lin, T. Deakin, and S. McIntosh-Smith, "Evaluating ISO C++ parallel algorithms on heterogeneous HPC systems," in *Proc. 2022 IEEE/ACM Int. Workshop Perform. Model. Benchmarking Simul. High Perform. Comput. Syst.*, 2022, pp. 36–47.

[35] M. Bhattacharya et al., "Portability: A necessary approach for future scientific software," 2022, *arXiv:2203.09945*.

[36] M. Atif et al., "Evaluating portable parallelization strategies for heterogeneous architectures in high energy physics," 2023, *arXiv:2306.15869*.

[37] C. Tanis, K. Sreenivas, J. C. Newman, and R. Webster, "Performance portability of a multiphysics finite element code," in *Proc. 2018 Aviation Technol. Integration Operations Conf.*, 2018, Art. no. 2890.

[38] T. Deakin, A. Poenaru, T. Lin, and S. McIntosh-Smith, "Tracking performance portability on the yellow brick road to exascale," in *Proc. IEEE/ACM Int. Workshop Perform. Portability Productiv. HPC*, 2020, pp. 1–13.

[39] C. R. Trott et al., "Kokkos 3: Programming model extensions for the exascale era," *IEEE Trans. Parallel Distrib. Syst.*, vol. 33, no. 4, pp. 805–817, Apr. 2022.

[40] J. R. Hammond and T. G. Mattson, "Evaluating data parallelism in C++ using the parallel research Kernels," in *Proc. Int. Workshop OpenCL*, 2019, Art. no. 14.

[41] Gaia Collaboration et al., "Gaia data release 3. Summary of the content and survey properties," *Astron. Astrophys.*, vol. 674, 2023, Art. no. A1.

[42] Gaia Collaboration et al., "Gaia data release 2. Mapping the Milky Way disc kinematics," *Astron. Astrophys.*, vol. 616, 2018, Art. no. A11.

[43] M. Crosta, M. Giammaria, M. G. Lattanzi, and E. Poggio, "On testing CDM and geometry-driven Milky Way rotation curve models with Gaia DR2," *Monthly Notices Roy. Astronomical Soc.*, vol. 496, pp. 2107–2122, 2020.

[44] M. Giammaria, A. Spagna, M. G. Lattanzi, G. Murante, P. ReFiorentin, and M. Valentini, "The formation history of the Milky Way disc with high-resolution cosmological simulations," *Monthly Notices Roy. Astronomical Soc.*, vol. 502, pp. 2251–2265, 2021.

[45] D. M. Krolikowski, A. L. Kraus, and A. C. Rizzuto, "Gaia EDR3 reveals the substructure and complicated star formation history of the greater Taurus-Auriga star-forming complex," *Astronimical J.*, vol. 162, 2021, Art. no. 110.

[46] N. Lagarde et al., "Deciphering the evolution of the Milky Way discs: Gaia APOGEE Kepler giant stars and the Besançon Galaxy Model," *Astron. Astrophys.*, vol. 654, 2021, Art. no. A13.

[47] A. Vecchiato, M. G. Lattanzi, B. Bucciarelli, M. Crosta, F. de Felice, and M. Gai, "Testing general relativity by micro-arcsecond global astrometry," *Astron. Astrophys.*, vol. 399, pp. 337–342, 2003.

[48] A. Büdenbender, G. van de Ven, and L. L. Watkins, "The tilt of the velocity ellipsoid in the Milky Way disc," *Monthly Notices Roy. Astronomical Soc.*, vol. 452, pp. 956–968, 2015.

[49] A. Hees, C. Le Poncin-Lafitte, D. Hestroffer, and P. David, "Local tests of gravitation with Gaia observations of solar system objects," in *Proc. Int. Astronomical Union*, vol. 12, no. S330, pp. 63–66, 2017, doi: 10.1017/S1743921317005907.

[50] Z. Davari and S. Rahvar, "Testing modified gravity (MOG) theory and dark matter model in Milky Way using the local observables," *Monthly Notices Roy. Astronomical Soc.*, vol. 496, pp. 3502–3511, 2020.

[51] A. G. Butkevich, A. Vecchiato, B. Bucciarelli, M. Gai, M. Crosta, and M. G. Lattanzi, "Post-Newtonian gravity and Gaia-like astrometry. Effect of PPN γ uncertainty on parallaxes," *Astron. Astrophys.*, vol. 663, 2022, Art. no. A71.
[52] W. O'Mullane, U. Lammers, L. Lindegren, J. Hernandez, and D. Hobbs, "Implementing the Gaia astrometric global iterative solution (AGIS) in Java," *Exp. Astron.*, vol. 31, pp. 215–241, 2011.
[53] L. Lindegren, U. Lammers, D. Hobbs, W. O'Mullane, U. Bastian, and J. Hernández, "The astrometric core solution for the Gaia mission. Overview of models, algorithms, and software implementation," *Astron. Astrophys.*, vol. 538, 2012, Art. no. A78.
[54] S. Bertone et al., "Application of time transfer functions to Gaia's global astrometry. Validation on DPAC simulated Gaia-like observations," *Astron. Astrophys.*, vol. 608, 2017, Art. no. A83.
[55] M. Crosta, A. Geralico, M. G. Lattanzi, and A. Vecchiato, "General relativistic observable for gravitational astrometry in the context of the Gaia mission and beyond," *Phys. Rev. D*, vol. 96, 2017, Art. no. 104030.
[56] V. Cesare, U. Becciani, and A. Vecchiato, "The MPI+CUDA Gaia AVU-GSR parallel solver in perspective of next-generation exascale infrastructures and new green computing milestones," *INAF Tech. Rep.*, 2022.
[57] C. Yang et al., "An empirical roofline methodology for quantitatively assessing performance portability," in *Proc. 2018 IEEE/ACM Int. Workshop Perform. Portability Productiv. HPC*, Tech. Rep. 164, 2018, pp. 14–23, doi: 10.20371/INAF/TechRep/164.
[58] S. Williams, A. Waterman, and D. Patterson, "Roofline: An insightful visual performance model for multicore architectures," *Commun. ACM*, vol. 52, pp. 65–76, 2009.
[59] J. L. Hennessy and D. A. Patterson, *Computer Architecture: A Quantitative Approach*. San Mateo, CA, USA: Morgan Kaufmann, 2017.

Giulio Malenza (Student Member, IEEE) received the BE degree in physics and the MS degree in mathematical engineering from the University of Padua, and the master's degree in high-performance computing from Trieste, in 2021. He is currently working toward the PhD degree with the University of Turin. His research focuses on the performance portability of different parallel programming frameworks used to port and optimize scientific codes on accelerators.

Valentina Cesare received the PhD degree in physics and astrophysics from the Physics Department, University of Turin, in March 2021, with a thesis about the study of the dynamics of galaxies with the theory of modified gravity "Refracted Gravity". She is a fixed-term technologist with INAF-Astrophysical Observatory of Catania, where she has worked since December 2020. She is actually working on GPU porting of applications related to Gaia space mission within ICSC–National Center of Research in HPC, Big Data and Quantum Computing–PNRR in Future Computing.

Marco Edoardo Santimaria received the bachelor's and master's degrees from the Computer Science Department, University of Turin's, where he is currently working toward the PhD degree. His research primarily focuses on developing middleware to aid workflow execution (in particular, scientific, file-based in-situ workflows). He is also active in HPC, distributed programming, and developing HPC tools for large-scale network analysis.

Robert Birke (Senior Member, IEEE) is a tenured assistant professor with the University of Turin, Italy. He has been a visiting researcher with IBM Research Zurich, Switzerland, and a principal scientist with ABB Corporate Research, Switzerland. He has published more than 120 papers. He is currently an associate editor of the IEEE TRANSACTIONS ON NETWORK AND SERVICE MANAGEMENT and COMPUTER COMMUNICATIONS. His research interests include the broad area of virtual resource management, including network design, workload characterization, and AI and big-data application optimization.

Alberto Vecchiato received the master thesis in physics and the PhD degree in physics from the University of Padova, in 1996 and 2001. He is currently working in software development as responsible of the AVU–GSR pipeline within the Gaia mission at INAF-Astrophysical Observatory of Torino, where he has held a permanent position since 2007. Generally, he is mainly working in the fields of astrometry, physics of gravitation, and tests of gravity physics theories. Since 2012, he has developed an interest for archaeoastronomy and the history of astronomy.

Ugo Becciani received the master's degree in electrotechnic engineering from the University of Catania, in 1986, and he got his first position with INAF-Catania, in 1988. He is a lead technologist and research director with INAF-Astrophysical Observatory of Catania. He is responsible for INAF's participation to ICSC–National Center of Research in HPC, Big Data, and Quantum Computing, where serves as a Spoke Leader of Spoke 3: Astrophysics and Cosmos Observation. He is also a professor with the University of Catania, author of numerous papers in international journals, and a member of several technical and scientific boards.

Marco Aldinucci is a full professor and the head with the Parallel Computing Research Group, University of Turin. He has authored more than 150 scientific articles and received several prestigious awards, including the HPC Advisory Council University Award in 2011, the NVidia Research Award in 2013, the IBM Faculty Award in 2015, and the Autodesk Award in 2021. His research interests lie in the broad area of parallel and distributed systems: from languages and programming models to tools for HPC, cloud, and large AI systems.