Performance portability via C++ PSTL, SYCL, OpenMP, and HIP: the Gaia AVU-GSR case study

Giulio Malenza*, Valentina Cesare†, Marco Edoardo Santimaria*, Robert Birke*, Alberto Vecchiato‡, Ugo Becciani†, and Marco Aldinucci*

*Department of Computer Science, University of Turin, Italy, Email: name.lastname@unito.it
†Astrophysical Observatory of Catania, National Institute for Astrophysics, Italy, Email: name.lastname@inaf.it
‡Astrophysical Observatory of Turin, National Institute for Astrophysics, Italy, Email: name.lastname@inaf.it

Abstract—**Applications that analyze data from modern scientific experiments will soon require a computing capacity of ExaFLOPs. The current trend to achieve such performance is to employ GPU-accelerated supercomputers and design applications to exploit this hardware optimally. Since each supercomputer is typically a one-off project, the necessity of having computational languages portable across diverse CPU and GPU architectures without performance losses is increasingly compelling. Here, we study the performance portability of the LSQR algorithm as found in the AVU-GSR code of the ESA Gaia mission. This code computes the astrometric parameters of the $\sim 10^8$ stars in our Galaxy. The LSQR algorithm is widely used across a broad range of HPC applications, elevating the study's relevance beyond the astrophysical domain. We developed different GPU-accelerated ports based on CUDA, C++ PSTL, SYCL, OpenMP, and HIP. We carefully verified the correctness of each port and tuned them to five different GPU-accelerated platforms from NVIDIA and AMD to evaluate the performance portability (Ⴔ) in terms of the harmonic mean of the application's performance efficiency across the tested hardware. HIP was demonstrated to be the most portable solution with a 0.94 average Ⴔ across the tested problem sizes, closely followed by SYCL coupled with AdaptiveCpp (ACPP) with 0.93. If we only consider NVIDIA platforms, CUDA would be the winner with 0.97. The tuning-oblivious C++ PSTL achieves 0.62 when coupled with vendor-specific compilers.**

Index Terms—**High-Performance Computing, Performance portability, Portable languages, GPU programming, CPU and GPU architectures, Astrometry**

I. Introduction

The scientific data produced by experiments in different domains are quickly increasing in size, approaching the $\sim$10-100 PB range. Processing such amounts of data in a reasonable time requires Peta- and Exa- FLOP/s of computing capacity. Today such compute capacities are achieved using accelerated supercomputers comprising O(10^3) computational nodes having one or more accelerators, such as GPUs. The inclusion of GPU accelerators adds a new dimension of heterogeneity that further exacerbates the long-standing problem of code and performance portability across successive generations of supercomputers typical of the HPC domain. 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 re-adapting the application structure to fully exploit GPU-accelerated compute node capabilities. While it might not be as effective as co-design for achieving record performance, it can improve the scalability 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. This kind of code porting and tuning is typical, e.g., with CUDA [1] and HIP [2], the respective low-level native programming languages for NVIDIA and AMD GPUs that allow 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 re-tuning on different boards, 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, for industrial users, the ability to preserve investment in code development and tuning across different generations of platforms significantly affects their entire value chain linked to supercomputers. The advent of GPU-accelerated supercomputers makes this need evermore compelling.

To address code portability several programming frameworks that aim to decouple the application from specific hardware platforms with no (or limited) performance loss, have been recently proposed. 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 models raises the concept of *performance portability*, which is becoming a crucial research point 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].

Differently 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 Solver module which implements a customized and parallelized version of the iterative LSQR algorithm [20], [21] to be able to deal with the problem specific structure of the data. This code has been uses in production since 2014 at the CINECA supercomputing center. The LSQR solver is the crucial computational unit in the pipeline and the object of your analysis. For sake of simplicity, hereon we use the term "AVU-GSR" for the LSQR solver of the pipeline alone.

Malenza et al. [22] studied the weak scalability of two ports of the AVU-GSR code, written in CUDA [23], and C++ PSTL, on up to 256 nodes of Leonardo[1] with NVIDIA A100 GPUs. Here, we add further programming frameworks, notably HIP, OpenMP with GPU offload (OpenMP-GPU), and SYCL, verify their correctness, and study their performance portability using Pennycoock's Φ score across five different compute architectures from both NVIDIA and AMD.

Our main contributions can be summarized as follows:

- We study the parallelization of ESA's AVU-GSR solver and port it to HIP, OpenMP-GPU, and SYCL. The choice stems from the promising results and properties described in literature [2], [3], [5], [9].
- We validated the correctness of each port using reference datasets from production and tune each port for optimal performance obtaining a speedup of up to $\sim$ 2x over the CUDA version currently used in production.
- We experimentally determine the Φ performance portability score of eight programming frameworks and compiler combinations across five hardware platforms, four based on NVIDIA, and one based on AMD, using three problem sizes.

Paper roadmap. In §II, we present literature works related to our analysis. In §III, we illustrate the main scientific motivation of our study and the structure of the Gaia AVU-GSR code. In §IV, we describe the parallelization and optimization strategies of the code version. In §V, we present our results in terms of performance portability and code validation. §VI concludes the paper and presents future works.

II. Background and Related works

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:

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

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. 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 worse performance on the first set of platforms compared to the second one. Φ measures the imbalance of platform support across different code versions, not their absolute efficiency. If an application is not able to run on all platforms in H, the Φ is 0 by definition.

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 [24], a reimplementation of McCalpin's STREAM benchmark [25], parallelized with eight programming models (McCalpin, SYCL, RAJA, KOKKOS, OpenMP-C++, OpenACC, CUDA, and OpenCL) and 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. [26] test the performance portability of three heterogeneous high-performance computing (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. [27] 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. [28] added the Alpaka parallelization language to the previous study. Other works study performance portability from the perspective of the (pre-)Exascale era. Some of them are [29]–[31]. Such studies only address benchmarks or mini-apps or lack the use of a wider-spread metric. Differently from them, we use a real-world application using the recognized Φ metric.

III. Gaia Mission and AVU-GSR Code

A. Scientific relevance of the Gaia mission

The Gaia mission was launched on December 19th 2013, and it is expected to end its life at the beginning of 2025,

[1]https://www.hpc.cineca.it/systems/hardware/leonardo/

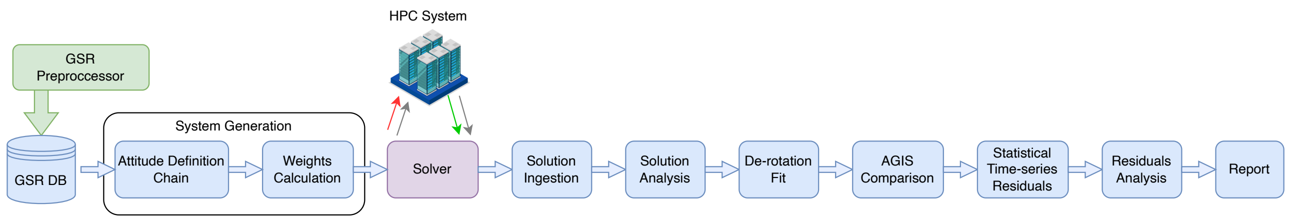


Fig. 1: Description of the pipeline of the AVU-GSR mission code. The solver (in purple) is the main computation bottleneck offloaded to an HPC system.

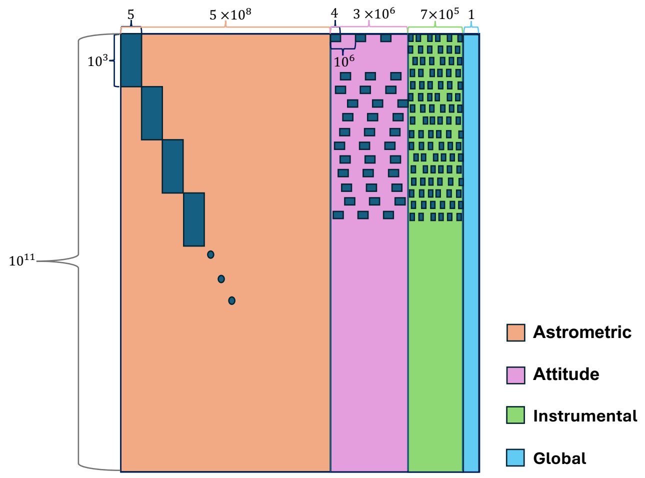


Fig. 2: Structure of matrix $\mathbf{A}$. Non-zero parameters are highlighted as dark blue blocks.

after about seven-years extension from its initial planned end (2018)[2]. This is due to the relevant scientific worth provided by the mission. 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 [32]. The accuracy of the astrometric measurements can go down to the 10-100 micro-arcseconds level or below [32]. 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) [33], [34] and to study the formation and evolution of the Milky Way (e.g., [35]–[37]). 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., [34], [38]–[42]).

B. The Gaia AVU-GSR code

The Gaia AVU-GSR code determines the astrometric parameters of the primary stars in the Milky Way ($\sim10^8$ stars), as well as the attitude and instrumental specifications of the Gaia satellite and the global parameter γ of the Parametrized Post-Newtonian (PPN) formalism, with a high precision around $10-100$ micro-arcseconds. To find these parameters, the Gaia AVU-GSR code solves an overdetermined system of linear equations [19], [22], [23], [43]–[46], choosing between two fully-relativistic astrometric models [47], [48]:

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

where $\mathbf{A}$ is a large coefficient matrix with a high degree of sparsity, $\vec{b}$ is the vector of known terms and $\vec{x}$ is the array of unknowns. The number of rows of $\mathbf{A}$ represents the number of equations which is equal to the number of stars times the number of observations per star, i.e., $\mathrm{O}(10^{8+3})$. The number of columns of $\mathbf{A}$ represents the number of unknowns dominated by the 5 astrometric parameters per star, i.e., $\mathrm{O}(10^8)$. Fig. 2 depicts the structure of the matrix. The matrix is highly sparse with a block diagonal structure for the astrometric parameters, whereas the structure of the attitude parameters depends on a stride stemming from the measurement campaign, and the instrumental part has an irregular pattern. The global part has, at maximum, one parameter per row, different from zero. Saving only the nonzero elements of $\mathbf{A}$ allows to reduce the problem by seven orders of magnitude. The obtained reduced matrix, $\mathbf{A}_\mathrm{r}$, has the same number of rows as $\mathbf{A}$ and contains at most $\sim (10^{11}) \times 24$ elements, i.e., 5 astrometric, 12 attitude, 6 instrumental, and 1 global parameters per row. The dark blue blocks in Fig. 2 (not in scale) illustrate how the non-zero parameters distribute in each row of the astrometric, attitude and instrumental parts.

To store the matrix in memory, we leverage the inherent properties of the matrix. Specifically, we split A_r into four submatrices (one for the astrometric, attitude, instrumental, and global parameters) differently treated according to their structure. The astrometric submatrix is structured with a block diagonal pattern, with each block row only containing five contiguous non-zero elements (Figure 2). To efficiently store this system, we only need to save the starting column index of the astrometric non-zero elements for each row of $\mathbf{A}$ in an

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

array called `matrixIndexAstro`. For the attitude parameters system, each row of $\mathbf{A}$ contains 12 non-zero elements arranged in three blocks of four elements each, with a defined stride (Figure 2). Knowing this pattern allows us to only store the index of the first attitude non-zero element in each row of $\mathbf{A}$ in an array named `matrixIndexAtt`. Unlike the other submatrices, the instrumental parameters lack a predictable pattern (Figure 2). As a result, we store, for each row of $\mathbf{A}$, the column indexes of all instrumental non-zero elements in an array called `instrCol`. Overall, the astrometric submatrix represents $\sim$90% of the memory footprint while the other parts (attitude, instrumental, and global) the remaining $\sim$10%. Overall, $\mathbf{A}_{\mathrm{r}}$, $\vec{b}$ and $\vec{x}$ occupy $\sim$19 TB, $\sim$800 GB and $\sim$4 GB, respectively.

The system solution is iteratively obtained through a customized and preconditioned version of the LSQR algorithm [20], [21], which approximately takes the $\sim$95% of the AVU-GSR solver. The algorithm stops when it reaches convergence or the maximum number of iterations. The most intensive computation of the single LSQR iteration is the execution of the *aprod1*,

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

and *aprod2*,

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

functions. In (2), the number of equations is larger than the number of unknowns, which makes the system overdetermined. Therefore, some constraint equations must be set to derive a univocal solution.

IV. Parallelization of the Gaia AVU-GSR Code

The Gaia AVU-GSR 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 [23], [46]. In particular, the two most intensive computations are implemented as four kernels (one for each submatrix) respectively named as *aprod{1,2}_Kernel_astro()*, *aprod{1,2}_Kernel_att()*, *aprod{1,2}_Kernel_instr()* and *aprod{1,2}_Kernel_glob()*.

Kernel parallelization. The *aprod 1* kernels result in rows of matrix times column vector multiplications, which can be easily parallelized. Instead, due to the irregular structure of transposed $\mathbf{A}$, the indexes used by *aprod2* can collide (with the exception of the astrometric parameters due to their block diagonal structure), and hence the updates require atomic operations. To minimize the collision probability, we redesigned the code to reduce the number of blocks and GPU threads per block in the regions where atomic operations are performed. 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 their 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.

Parallel programming frameworks. In addition to an optimized CUDA implementation leveraging the observations above, we parallelize the LSQR solver using four different additional parallel programming frameworks: HIP, SYCL, OpenMP, and C++ PSTL. CUDA, HIP and SYCL are language-specific, meaning they have explicit run-time functions that users can use to manage memory and write fine-tuned kernels [49]. OpenMP, similarly to OpenACC, is a directive-based API [50]; this means the user can add `pragma` directives that instruct the compiler to generate assembly code that runs on multicores systems and, eventually, accelerators. Generally, pragma-based programming languages are easier to use compared to language-specific frameworks as users are not required to re-write the entire application, and do not require explicit tuning and memory management to offload computation on GPUs or multicore systems. 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, RAJA, Alpaka, Thrust, and C++ PSTL. Here, we used the C++ PSTL as 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 specifying the execution policy [51]. Offloading computations to GPU requires implicit memory mapping mechanics between host and device and a back-end that is able to generate GPU assembly code.

a) The CUDA code: The CUDA version allocates the host variables via *cudaHostMalloc* to use pinned memory. All variables needed by the GPU are allocated using *cudaMalloc*. The four submatrices are copied asynchronously to GPUs, using *cudaMemcpyAsync*. The same is done with all relevant quantities, constraints, and other unknowns of the astrometric problem. CUDA streams are used to overlap *aprod2* 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 frameworks, allowing explicit memory management.

b) Porting from CUDA to HIP: Since HIP is a programming language designed to be syntactically similar to CUDA, porting was relatively easy. Initially, we used HIPIFY to translate our code into HIP, and then we optimized the code, tuning kernel parameters for AMD and NVIDIA architectures. In particular, *cudaMalloc*, *cudaMemcpyAsync*, and *cudaStreamCreate* were replaced with *hipMalloc*, *hipMemcpyAsync* and *hipStreamCreate*. One thing worth noting is that, same as with C++ PSTL, we allocate the memory using *hipMemAdvise* and forcing coarse grain. This is done for performance reasons as we observed experimentally that fine-grain coherence led to performance degradations due to the atomic operations contained in the *aprod2* kernels.

c) Porting from CUDA to SYCL: The SYCL implementation uses order queues and Unified Shared Memory allocators.

TABLE I: Software Versions on NVIDIA architectures

	T4 & V100	A100	H100
CUDA	12.3	11.8	12.3
NVC++	24.3	24.3	24.3
AdaptiveCpp	24.06	24.06	24.06
HIP	5.7.3	5.7.3	5.7.3
Clang	17.0.6	17.0.6	17.0.6
DPC++	19.0.0	19.0.0	19.0.0

In particular, `malloc_device` allocates data directly on GPU. We used `parallel_for` to declare parallel code regions and `NDrange` to fine-tune kernels.

d) Porting from CUDA to OpenMP: We used the OpenMP `#pragma omp enter data` 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`.

e) Porting from CUDA to C++ PSTL: The first porting of the code to C++ PSTL was done in [22]. It is worth noting that, in this case, there is no specific directive to tune the number of threads and blocks to be used by the kernels.

V. Results

A. Software stack and hardware platforms

We measured the Ψ metric considering only runs using a single GPU. We used the following hardware platforms:

- **GraceHopper**: CPU: 1x NVIDIA Grace CPU (72 Arm Neoverse V2 Cores @ 3.0 GHz); GPU 1x NVIDIA HOPPER H100, 96 GB HBM3e; MEM: 574 GB LPDDR5X; Driver Version: 545.23.08; CUDA Version: 12.3;
- **EpiTo**: CPU: 1 Ampere Altra Q80-30 CPU (80-core Arm Neoverse M1); GPU: 2 x NVIDIA A100 GPU, 40 GB HBM2 each; Driver Version: 515.65.01; CUDA Version: 11.7;
- **CascadeLake**: CPU: 1 Intel(R) Xeon(R) Gold 6230 CPU; GPU: 1 x NVIDIA V100S GPU, 32 GB HBM2; 1 x NVIDIA T4 GPU, 15 GB GDDR6; Driver Version: 550.54.15 ; CUDA Version: 12.4;
- **Setonix**: CPU: 1x AMD EPYC 7742 (64 cores@2.25 GHz); GPU: 4x AMD MI250x, 512 GB HBM2; MEM: 512 GB DDR4;

Since we mainly target the performance using accelerators and each platform has a different GPU, hereon we identify each platform via the name of the equipped GPU. We compare the performances of the different implementations using the LSQR iteration time. Since the AVU-GSR code is based on an iterative algorithm, we believe that the iteration time is a good estimator of the application performance. We used code profilers from NVIDIA and AMD to verify that most of the time of this code is spent computing the matrix-by-vector products of *aprod1* and *aprod2*.

TABLE II: Compilation Flags on NVIDIA architecture

P. Frameworks	Compilation Flags
CUDA	`-gencode=arch=compute_XX,code=sm_XX`
HIP	`--gpu-architecture=sm_XX`
SYCL	
acpp	`--acpp-platform=cuda` `--acpp-targets=cuda:sm_XX` `--acpp-gpu-arch=sm_XX`
DPC++	`-fsycl -fsycl-targets=nvptx64-nvidia-cuda` `-Xsycl-target-backend` `--cuda-gpu-arch=sm_XX`
OpenMP	
clang++	`-fopenmp -fopenmp-targets=nvptx64-nvidia` `-cuda -Xopenmp-target=nvptx64-nvidia-cuda` `-march=sm_XX`
nvc++	`-mp=gpu -gpu=ccXX,sm_XX`
PSTL	
acpp	`--acpp-platform=cuda --acpp-stdpar` `--acpp-targets=cuda:sm_XX` `--acpp-stdpar-unconditional-offload` `--acpp-gpu-arch=sm_XX`
nvc++	`-stdpar=gpu -gpu=ccXX,sm_XX`

a) Software stack on NVIDIA architecture: We built the compilers from source for each platform using `gcc-12.2.0` as base compiler. Table I and Table II summarize the compilers and compiler flags used for each framework. CUDA and HIP versions were compiled with `nvcc` and `hipcc`. We compiled mostly all code with the flag `-std=c++20 -O3`. For CUDA and HIP versions on EpiTo we used `-std=c++17` for default system setting reasons. The same C++ standard version was used for the SYCL code when compiled with `DPC++` compiler.

b) Software stack on AMD architecture: On AMD, we used the `rocm-5.7.3` toolkit installed on the Setonix system to compile HIP and OpenMP code. OpenMP code was also compiled with native clang compiler version `17.0.6`. We installed the AdaptiveCpp compiler version `24.06` to compile SYCL and C++ PSTL versions. We installed `rocm-stdpar` compiler version `18.0.0` for C++ PSTL and `DPC++` compiler `18.0.0` for SYCL. Table III summarizes all compilation flags used. All codes were compiled using the additional `-std=c++20 -O3` flags.

B. Performance portability

We measure the Ψ of all code versions across the platforms listed in §V-A using the application efficiency based on the average iteration time of the solver. To ensure that our ports are representative of the application's best performance, we did a preliminary comparison of our optimized CUDA version against the production version of the code, obtaining a speed-up of 2.0x on Leonardo on a 42 GB problem.

The considered accelerators vary in their available device RAM capacity. Since the GPU memory occupancy is closely related to the size of the **A** matrix (copied only once before the main iteration cycle), we tested problems of different sizes close to the limit of the different GPUs, in particular: 10 GB (on all devices), 30 GB (all except Tesla T4) and 60 GB (only

TABLE III: Compilation Flags on AMD architecture

P. Frameworks	Compilation Flags
HIP	`--offload-arch=gfx90a -munsafe-fp-atomics`
SYCL	
acpp	`--acpp-platform=rocm --acpp-targets=generic --acpp-gpu-arch=gfx90a -munsafe-fp-atomics`
DPC++	`-fsycl -fsycl-targets=amdgcn-amd-amdhsa -Xsycl-target-backend --offload-arch=gfx90a`
OpenMP	
clang++	`-fopenmp -fopenmp-targets=x86_64, -fopenmp-targets=amdgcn-amd-amdhsa -Xopenmp-target=amdgcn-amd-amdhsa -march=gfx90a`
amdclang++	`-fopenmp --offload-arch=gfx90a -munsafe-fp-atomics`
PSTL	
acpp	`--acpp-platform=rocm --acpp-stdpar --acpp-targets=hip:gfx90a --acpp-stdpar-unconditional-offload --acpp-gpu-arch=gfx90a -munsafe-fp-atomics`
clang++	`--hipstdpar --hipstdpar-path=$(HIPSTDAR_ROOT) --offload-arch=gfx90a -munsafe-fp-atomics`

on H100 and MI250X)[3]. We report the average iteration time over 100 iterations, and repeat each experiment 3 times to enhance its statistical robustness.

Using the p3-analysis-library [52], we plot the application efficiency and Ⴔ for all platforms and frameworks for each problem size in Fig. 3. In each subfigure, the plot on the top left shows how efficiency varies across frameworks and platforms. Lines identify the AVU-GSR solver implemented using different framework-compiler pairs. In particular, the first value on the x-axis describes the maximum efficiency on the best-performing hardware for a given framework. The hardware platform itself is identified by the letter in the plot below on the row with the line of the same color. The right top plot shows instead the performance portability computed using Eq. (1) across all platforms. The green and violet lines illustrate the efficiency of OpenMP code when compiled with the base clang (OMP+LLVM) and the vendor compilers (OMP+V), respectively. The brown and gray lines depict the efficiency of C++ PSTL when using AdaptiveCpp (PSTL+ACPP) and vendor compilers (PSTL+V). Lastly, the yellow and cyan lines represent the efficiency of SYCL when compiled with AdaptiveCpp (SYCL+ACPP) and DPC++ (SYCL+DPCPP)[4].

To dive deeper into the results, we also report, in Fig. 4 and 5, the average iteration time and application efficiency of all frameworks on the different platforms. Each figure comprises three subfigures that show the results for the three problem sizes.

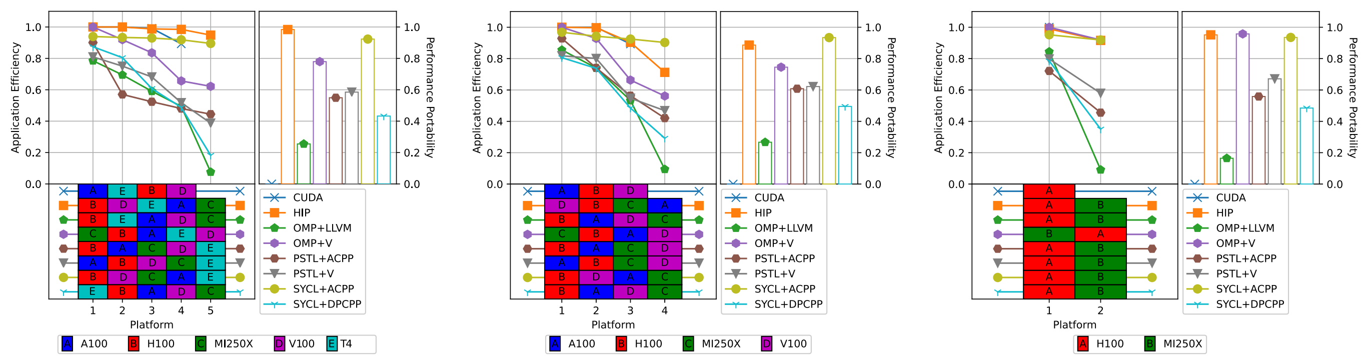
Fig. 3a, Fig. 3b and Fig. 3c show the results for a problem size of 10 GB, 30 GB, and 60 GB, respectively. For a 10 GB problem, the highest Ⴔ was achieved by HIP (0.98) followed by SYCL+ACPP (0.92). Apart from CUDA's Ⴔ value, which is zero by definition since it cannot be executed on AMD GPUs, the worst value is 0.25 obtained by OMP+LLVM, and all others lying somewhere in between. Interestingly, the best efficiency is obtained on the most recent NVIDIA hardware, i.e., H100, for 4 out of 8 frameworks, including even HIP. MI250X is, instead, the best platform for OMP+V. Surprisingly, T4 is the best platform for SYCL+DPCPP. Only V100 has never been the best platform for any of the frameworks. When considering the results on a bigger problem, i.e., 30 GB (shown in Fig. 3b), the overall application efficiencies vary more leading to generally lower performance portability scores. Here the best score is 0.93 by SYCL+ACPP which surpasses HIP with a score of 0.88. The framework with the worst drop in efficiency is OMP+LLVM which goes from 0.85 on H100 to 0.53 on V100. The best platform remains H100, which is the first choice for half the frameworks and the second best for the remaining half. Unsurprisingly, the MI250X is the best platform for HIP. Only two GPUs have enough memory to support the 60 GB problem: H100 and MI250X. The trends shown before are confirmed here too, but overall, more frameworks obtain high scores due to the low number of hardware platforms. Not considering the MI250X, CUDA would achieve a Ⴔ score of 0.97 and 0.96 for the 10 GB and 30 GB problem sizes, respectively. Note that there is no meaning to compute Ⴔ from the 60 GB problem since we have only one NVIDIA GPU supporting that size in our test platforms.

The figures clearly show that HIP and SYCL+ACPP (and CUDA when considering only NVIDIA GPUs) obtain the best Ⴔ scores stemming from high application efficiency scores. This is partially confirmed by the results shown in Figs. 4 and 5. While, as expected, newer and more performant platforms clearly deliver lower average iteration times across all model sizes, given a platform, the fastest time is typically given by CUDA (mostly on T4 and A100) or HIP (mostly on V100 and H100) frameworks. Surprisingly, the best framework on MI250X is OMP+V, however it is clearly outperformed on the NVIDIA-accelerated platforms by the other frameworks leading to a lower Ⴔ score (between 0.95 and 0.45 across the three problem sizes). Instead, SYCL+ACPP, while not being the best on any platform, achieves similar application efficiencies across all the tested hardware.

CUDA, HIP, and SYCL allow hand-tuning of the GPU kernels. Indeed, different numbers of blocks and threads of the kernels produce different performance results. In our experiments, we tuned the parameters of the CUDA, HIP, and SYCL kernels for each platform, achieving up to 40% reduction in iteration time. This testifies how relevant tuning such frameworks can be. Unfortunately, different platforms often require different tuning. The best SYCL performance is

[3]Bigger problems can be addressed using multiple GPUs eventually on multiple nodes which is out of scope of this paper.

[4]Hereon, for simplicity we refer to framework as specific framework plus compiler combinations



(a) 10 GB Problem (b) 30 GB Problem (c) 60 GB Problem

Fig. 3: Performance portability for problems of size: a) 10GB, b) 30GB, and c) 60GB.

obtained using the AdaptiveCpp compiler. SYCL code compiled with the DPC++ compiler offers lower performance. This can be due to incorrect compilation or suboptimal parameter tuning. We kept the same tuning configurations adopted for AdaptiveCpp.

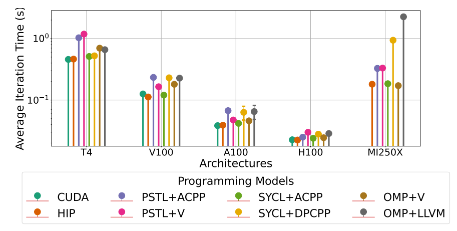
The other frameworks do not allow easy tuning of the GPU kernels. The default compiler tuning produced a code that, on H100, achieved 91% and 84% of the CUDA performance, when compiled with `nvc++` and standard `clang++`, respectively, approximately on all problem sizes. On other platforms, OpenMP performed slightly less but still between 83% and 59% of the best-achieved performance with comparable performance on both compilers.

The C++ PSTL efficiency increases from T4 to H100, reaching a value of 90% application efficiency on H100 coupled with ACPP and problem sizes of 10 GB and 30 GB. `nvc++` compiler performs slightly better than ACPP on H100 for the 60 GB problem size reaching 79%. As for OpenMP, both compilers achieve comparable performance, even if ACPP does not explicitly use the system unified shared memory mechanism [53] while it is required by `nvc++`. The increasing performance trend mirrors that of more performant hardware, so we hypothesize that it is due to the different technical characteristics of the considered GPUs. As said in §IV, with C++ PSTL we cannot explicitly set the kernels parameters yet. Using the `nsys` profiler, we saw that the default parameter tuning spans 256 threads per block on each architecture. While this number of threads efficiently optimizes the kernel's execution on H100 and A100, it is less efficient on the weaker T4 and V100, where, as seen from the kernel's tuning of the CUDA code, the number of threads that give best performance is 32. The C++26 proposal [54] aims to include executors in the STL. This feature will potentially allow to set explicit kernel parameters and, hence, reduce the observed performance gap among the platforms.

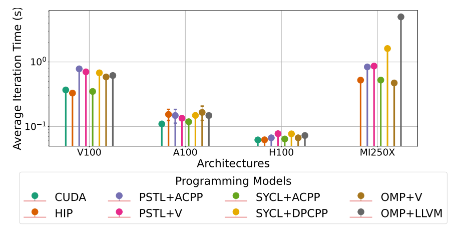
A special note goes to the performance on MI250X. Even with a powerful GPU the observed performance is slower than on the NVIDIA architectures, especially A100 and H100. This is mainly due to how the `aprod{1,2}` kernels were originally written. To verify this hypothesis, we take similar SpMV kernels, implemented using HIP by amd-lab-notes [55], and test them on matrix sizes similar to our own. We tested them on A100 and MI250X architectures. Indeed, the performance was similar to the one obtained by our AVU-GSR solver. We hypothesize that the lower performance is due to noncoalescent memory accesses by threads. We tried to maximize performance by kernel tuning and noted that low numbers of threads and blocks offer the best performance. OpenMP code compiled with `amdclang++` is the one that achieves the best performance on all considered problem sizes. We tuned the OpenMP kernels with parameters similar to the ones used by HIP and SYCL as the default compiler tuning brought lower performance. HIP and SYCL perform similarly to OpenMP when compiled with `hipcc` and `acpp`, respectively. While SYCL code compiled with `DPC++` compiler and OpenMP code compiled with base `clang++` compiler gives lower performance. This is due to the fact that some compilers could not generate code that uses atomic read-modify-write (RMW). They probably generate code in which atomic operations are performed with a compare-and-swap (CAS) loop. In our case, this degrades performance. Specifying the flag `-munsafe-fp-atomics` on MI250X, we are specifying to generate assembly code with atomic RMW instructions, but it is not supported on all compilers. Finally, C++ PSTL code achieved an application efficiency of 0.45-0.6 with both `clang++` and `acpp`. This is because we could not properly tune the kernel parameters. Again, we believe that the new executors feature proposed for C++26 could help solve this problem.

C. Code validation

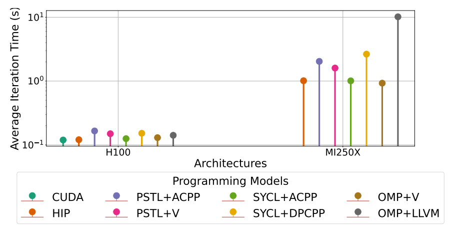
We verified the correctness of the different code versions across all considered frameworks and H100, A100, and AMD MI250x architectures, by comparing the solution of the system from Eq. (2) with its standard error against the solution found by the CUDA code currently in production on the Leonardo supercomputer. For this verification, we consider two real, well-studied datasets of 42 GB and 306 GB. For both datasets, the solution includes the astrometric, attitude, and instrumental



(a) 10 GB Problem

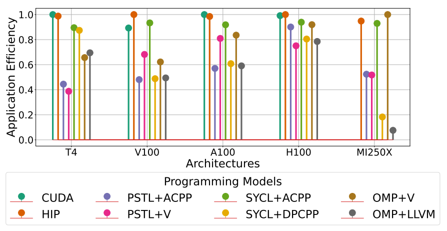


(b) 30 GB Problem

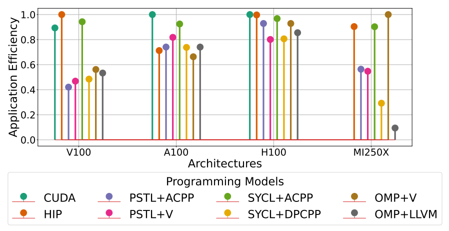


(c) 60 GB Problem

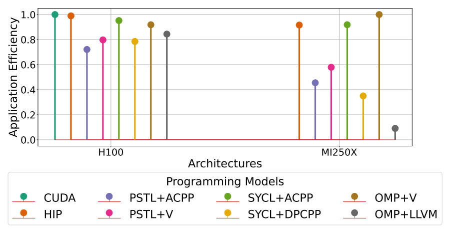
Fig. 4: Average iteration time across architectures and programming models for problems of size: a) 10 GB, b) 30 GB, and c) 60 GB.



(a) 10 GB Problem



(b) 30 GB Problem



(c) 60 GB Problem

Fig. 5: Application efficiency across different platforms and programming frameworks for problems of size: a) 10 GB, b) 30 GB, and c) 60 GB.

sections and no global section, which has not been computed yet in production runs.

The two left panels of Fig. 6 compare the astrometric solution (red) and its standard error (blue) obtained for the 42 GB dataset with the HIP code run on Leonardo and the same quantities obtained with the CUDA-production code on Leonardo as well. The two right panels show the same quantities with the HIP code run on Setonix. The black dashed line testifies the one-to-one relation between the two solutions. Similar results have been obtained on all other considered frameworks for both datasets and, therefore, are not shown. From a visual inspection, we can see that the different quantities are consistent with each other and this agreement is also quantitatively verified, being the different couples of solutions in agreement within 1σ. Moreover, the Gaia mission aims to determine the astrometric parameters with a $10-100$ micro-arcseconds = $(0.48-4.8)\times10^{-10}$ rad precision. We verified that the mean and standard deviation of the differences between the standard errors obtained with the CUDA production code and all other versions always stay below the 10 micro-arcseconds threshold.

.

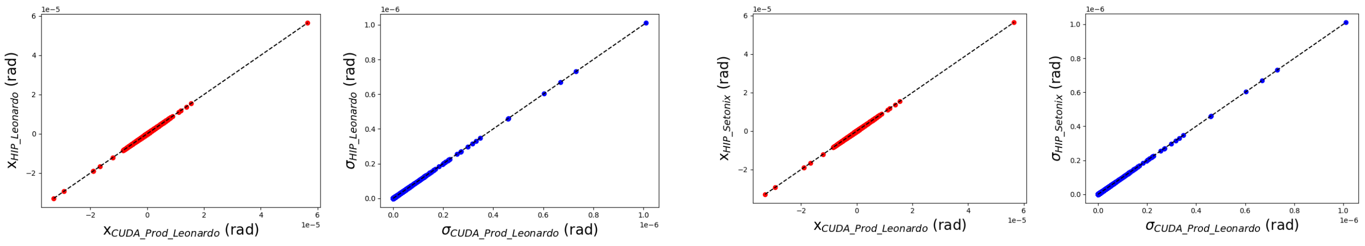
VI. Conclusions and Future Works

Performance portability (Φ) is an increasingly impelling necessity in HPC to lower the time to solutions on new supercomputers. For this purpose, we present a study taking the real-world application AVU-GSR solver, based on the LSQR algorithm, of the ESA Gaia mission as a test case. The main operations are two sparse matrix-by-vector products, a well-known, highly memory-bound operation. Specifically, we optimized the original CUDA version and re-implemented it using HIP, SYCL, OpenMP-GPU, and C++ PSTL, with the purpose of measuring the Φ of each framework across multiple hardware platforms. We considered all the platforms available to us: NVIDIA Tesla T4, Volta V100, Ampere A100 and Hopper H100, and AMD MI250X. We compiled and tested the code using different compilers and verified the correctness of the solutions.

Language-specific programming frameworks, specifically HIP and SYCL, provide the best performance portability values with an average across different problem sizes of 0.94 and 0.93, respectively. CUDA is the best programming framework on NVIDIA architectures, achieving a Φ value of 0.97. OpenMP is the most performant on AMD MI250X when compiled with `amdclang++`. On NVIDIA, the best performance portability values are 0.91 and 0.84 when the code is compiled with `nvc++` and `clang++` on H100, and it is tested on a problem of 60 GB. C++ PSTL achieves its maximum (0.9) application efficiency on H100 and problem sizes of 30 GB and 60 GB. On MI250X, it reaches the values of 0.45-0.6 with both `clang++` and `acpp` compilers. In the Gaia AVU-GSR case, tuning kernel parameters is fundamental to better exploit GPU computational power. Programming frameworks, such as C++ PSTL, for which this is not possible, usually have lower performance portability values. For this reason, using executors, these performance gaps are expected to be reduced.

Acknowledgments

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



(a) Solution Astrometric (b) Std Error Astrometric (c) Solution Astrometric (d) Std Error Astrometric

Fig. 6: a) Astrometric solution (red) of the system of equations (2), and b) standard error (blue) of the solution obtained on test problem of 42 GB with the HIP code on H100 as a function of the same quantities obtained with the CUDA-production code on Leonardo. The black dashed line represents the one-to-one relation as a reference. c) and d): same plots but with the HIP code executed on MI250X platform.

ationEU; by the Italian Space Agency (ASI) (grant n 2018-24-HH.0) as part the Gaia mission; by the EuroHPC JU under the project EUPEX (grant n. 101033975); and by the INAF minigrant awarded to Dr. Valentina Cesare.

References

[1] D. Kirk, “Nvidia cuda software and gpu parallel computing architecture,” in *Proceedings of the 6th International Symposium on Memory Management*, ser. ISMM ’07. New York, NY, USA: Association for Computing Machinery, 2007, p. 103–104. [Online]. Available: https://doi.org/10.1145/1296907.1296909

[2] N. Kerscher, “Investigating the hip programming model with regards to portability and performance portability,” 2022.

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

[4] H. Carter Edwards, C. R. Trott, and D. Sunderland, “Kokkos: Enabling manycore performance portability through polymorphic memory access patterns,” *Journal of Parallel and Distributed Computing*, vol. 74, no. 12, pp. 3202–3216, 2014, domain-Specific Languages and High-Level Frameworks for High-Performance Computing. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0743731514001257

[5] J. Huber, M. Cornelius, G. Georgakoudis, S. Tian, J. M. M. Diaz, K. Dinel, B. Chapman, and J. Doerfert, “Efficient execution of openmp on gpus,” in *2022 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, 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 *2014 First Workshop on Accelerator Programming using Directives*, 2014, pp. 19–26.

[7] D. A. Beckingsale, J. Burmark, R. Hornung, H. Jones, W. Killian, A. J. Kunen, O. Pearce, P. Robinson, B. S. Ryujin, and T. R. Scogland, “Raja: Portable performance for large-scale scientific applications,” in *2019 IEEE/ACM International Workshop on Performance, Portability and Productivity in HPC (P3HPC)*, 2019, pp. 71–81.

[8] D. S. Medina, A. St-Cyr, and T. Warburton, “OCCA: A unified approach to multi-threading languages,” *arXiv e-prints*, p. arXiv:1403.0968, Mar. 2014.

[9] B. Johnston, J. S. Vetter, and J. Milthorpe, “Evaluating the performance and portability of contemporary sycl implementations,” in *2020 IEEE/ACM International Workshop on Performance, Portability and Productivity in HPC (P3HPC)*, 2020, pp. 45–56.

[10] E. Zenker, B. Worpitz, R. Widera, A. Huebl, G. Juckeland, A. Knüpfer, W. E. Nagel, and M. Bussmann, “Alpaka – an abstraction library for parallel kernel acceleration,” in *2016 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, 2016, pp. 631–640.

[11] M. Aldinucci, S. Ruggieri, and M. Torquati, “Porting decision tree algorithms to multicore using fastflow,” in *Machine Learning and Knowledge Discovery in Databases*, J. L. Balcázar, F. Bonchi, A. Gionis, and M. Sebag, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 7–23.

[12] A. Dubey, L. C. McInnes, R. Thakur, E. W. Draeger, T. Evans, T. C. Germann, and W. E. Hart, “Performance portability in the exascale computing project: Exploration through a panel series,” *Computing in Science & Engineering*, vol. 23, no. 5, pp. 46–54, 2021.

[13] W. Zhu, Y. Niu, and G. Gao, “Performance portability on EARTH: a case study across several parallel architectures,” *Cluster Comput*, vol. 10, no. 2, pp. 115–126, Mar. 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 *Supercomputing*, J. M. Kunkel, T. Ludwig, and H. W. Meuer, Eds. Cham: Springer International Publishing, 2014, pp. 53–75.

[15] S. Pennycook, J. Sewall, and V. Lee, “Implications of a metric for performance portability,” *Future Generation Computer Systems*, vol. 92, pp. 947–958, 2019. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0167739X17300559

[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 *Accelerator Programming Using Directives*, S. Bhalachandra, C. Daley, and V. Melesse Vergara, Eds. Cham: Springer International Publishing, 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 *High Performance Computing*, R. Yokota, M. Weiland, J. Shalf, and S. Alam, Eds. Cham: Springer International Publishing, 2018, pp. 711–723.

[18] A. Vecchiato, B. Bucciarelli, M. G. Lattanzi, U. Becciani, L. Bianchi, U. Abbas, E. Sciacca, R. Messineo, and R. De March, “The global sphere reconstruction (GSR). Demonstrating an independent implementation of the astrometric core solution for Gaia,” *A&A*, vol. 620, p. A40, Nov. 2018.

[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 *2014 International Conference on High Performance Computing Simulation (HPCS)*, 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. (TOMS)*, vol. 8, no. 1, pp. 43–71, 1982a.

[21] ——, “Algorithm 583: Lsqr: Sparse linear equations and least squares problems,” *ACM Trans. Math. Softw. (TOMS)*, vol. 8, no. 2, pp. 195–209, 1982b.

[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. Supercomput*, Mar. 2024.

[23] V. Cesare, U. Becciani, A. Vecchiato, M. Gilberto Lattanzi, F. Pitari, M. Aldinucci, and B. Bucciarelli, “The MPI + CUDA Gaia AVU-GSR parallel solver toward next-generation Exascale infrastructures,” *PASP*, vol. 135, no. 1049, p. 074504, Jul. 2023.

[24] 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 *High Performance Computing: ISC High Performance 2016 International Workshops, ExaComm, E-MuCoCoS, HPC-IODC, IXPUG, IWOPH, Pˆ 3MA, VHPC, WOPSSS, Frankfurt, Germany, June 19–23, 2016, Revised Selected Papers 31*. Springer, 2016, pp. 489–507.

[25] J. D. McCalpin *et al.*, "Memory bandwidth and machine balance in current high performance computers," *IEEE computer society technical committee on computer architecture (TCCA) newsletter*, vol. 2, no. 19-25, 1995.

[26] W.-C. Lin, T. Deakin, and S. McIntosh-Smith, "Evaluating iso c++ parallel algorithms on heterogeneous hpc systems," in *2022 IEEE/ACM International Workshop on Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems (PMBS)*, 2022, pp. 36–47.

[27] M. Bhattacharya, P. Calafiura, T. Childers, M. Dewing, Z. Dong, O. Gutsche, S. Habib, X. Ju, M. Kirby, K. Knoepfel, M. Kortelainen, M. Kwok, C. Leggett, M. Lin, V. R. Pascuzzi, A. Strelchenko, B. Viren, B. Yeo, and H. Yu, "Portability: a necessary approach for future scientific software," *arXiv e-prints*, p. arXiv:2203.09945, Mar. 2022.

[28] M. Atif, M. Battacharya, P. Calafiura, T. Childers, M. Dewing, Z. Dong, O. Gutsche, S. Habib, K. Knoepfel, M. Kortelainen, K. H. M. Kwok, C. Leggett, M. Lin, V. Pascuzzi, A. Strelchenko, V. Tsulaia, B. Viren, T. Wang, B. Yeo, and H. Yu, "Evaluating portable parallelization strategies for heterogeneous architectures in high energy physics," *arXiv e-prints*, p. arXiv:2306.15869, Jun. 2023.

[29] C. Tanis, K. Sreenivas, J. C. Newman, and R. Webster, "Performance portability of a multiphysics finite element code," in *2018 Aviation Technology, Integration, and Operations Conference*, 2018, p. 2890.

[30] T. Deakin, A. Poenaru, T. Lin, and S. McIntosh-Smith, "Tracking performance portability on the yellow brick road to exascale," in *2020 IEEE/ACM International Workshop on Performance, Portability and Productivity in HPC (P3HPC)*, 2020, pp. 1–13.

[31] C. R. Trott, D. Lebrun-Grandié, D. Arndt, J. Ciesko, V. Dang, N. Ellingwood, R. Gayatri, E. Harvey, D. S. Hollman, D. Ibanez, N. Liber, J. Madsen, J. Miles, D. Poliakoff, A. Powell, S. Rajamanickam, M. Simberg, D. Sunderland, B. Turcksin, and J. Wilke, "Kokkos 3: Programming model extensions for the exascale era," *IEEE Transactions on Parallel and Distributed Systems*, vol. 33, no. 4, pp. 805–817, 2022.

[32] Gaia Collaboration, A. Vallenari, A. G. A. Brown, T. Prusti, and et al., "Gaia Data Release 3. Summary of the content and survey properties," *A&A*, vol. 674, p. A1, Jun. 2023.

[33] Gaia Collaboration, D. Katz, T. Antoja, M. Romero-Gómez, and et al., "Gaia Data Release 2. Mapping the Milky Way disc kinematics," *A&A*, vol. 616, p. A11, Aug. 2018.

[34] M. Crosta, M. Giammaria, M. G. Lattanzi, and E. Poggio, "On testing CDM and geometry-driven Milky Way rotation curve models with Gaia DR2," *MNRAS*, vol. 496, no. 2, pp. 2107–2122, Aug. 2020.

[35] M. Giammaria, A. Spagna, M. G. Lattanzi, G. Murante, P. Re Fiorentin, and M. Valentini, "The formation history of the Milky Way disc with high-resolution cosmological simulations," *MNRAS*, vol. 502, no. 2, pp. 2251–2265, Apr. 2021.

[36] 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 Journal*, vol. 162, no. 3, p. 110, Sep. 2021.

[37] N. Lagarde, C. Reylé, C. Chiappini, R. Mor, F. Anders, F. Figueras, A. Miglio, M. Romero-Gómez, T. Antoja, N. Cabral, J. B. Salomon, A. C. Robin, O. Bienaymé, C. Soubiran, D. Cornu, and J. Montillaud, "Deciphering the evolution of the Milky Way discs: Gaia APOGEE Kepler giant stars and the Besançon Galaxy Model," *A&A*, vol. 654, p. A13, Oct. 2021.

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

[39] A. Büdenbender, G. van de Ven, and L. L. Watkins, "The tilt of the velocity ellipsoid in the Milky Way disc," *MNRAS*, vol. 452, no. 1, pp. 956–968, Sep. 2015.

[40] A. Hees, C. Le Poncin-Lafitte, D. Hestroffer, and P. David, "Local tests of gravitation with gaia observations of solar system objects," *Proceedings of the International Astronomical Union*, vol. 12, no. S330, pp. 63–66, 2018.

[41] Z. Davari and S. Rahvar, "Testing MOdified Gravity (MOG) theory and dark matter model in Milky Way using the local observables," *MNRAS*, vol. 496, no. 3, pp. 3502–3511, Aug. 2020.

[42] 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," *A&A*, vol. 663, p. A71, Jul. 2022.

[43] V. Cesare, U. Becciani, A. Vecchiato, M. G. Lattanzi, F. Pitari, M. Raciti, G. Tudisco, M. Aldinucci, and B. Bucciarelli, "Gaia AVU-GSR parallel solver towards exascale infrastructure," in *Astromical Data Analysis Software and Systems XXXI*, ser. Astronomical Society of the Pacific Conference Series, B. V. Hugo, R. Van Rooyen, and O. M. Smirnov, Eds., vol. 535, May 2024, p. 405.

[44] V. Cesare, U. Becciani, A. Vecchiato, F. Pitari, M. Raciti, and G. Tudisco, "The Gaia AVU-GSR parallel solver: preliminary porting with OpenACC parallelization language of a LSQR-based application in perspective of exascale systems," *INAF Technical Reports*, vol. 163, Jul. 2022.

[45] V. Cesare, U. Becciani, A. Vecchiato, M. G. Lattanzi, F. Pitari, M. Raciti, G. Tudisco, M. Aldinucci, and B. Bucciarelli, "The Gaia AVU-GSR parallel solver: preliminary studies of a LSQR-based application in perspective of exascale systems," *Astronomy and Computing*, vol. 41, p. 100660, Oct. 2022.

[46] 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 Technical Reports*, vol. 164, Jul. 2022.

[47] S. Bertone, A. Vecchiato, B. Bucciarelli, M. Crosta, M. G. Lattanzi, L. Bianchi, M.-C. Angonin, and C. Le Poncin-Lafitte, "Application of time transfer functions to Gaia's global astrometry. Validation on DPAC simulated Gaia-like observations," *A&A*, vol. 608, p. A83, Dec. 2017.

[48] 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, p. 104030, Nov 2017. [Online]. Available: https://link.aps.org/doi/10.1103/PhysRevD.96.104030

[49] J. Nickolls, I. Buck, M. Garland, and K. Skadron, "Scalable parallel programming with cuda," *Queue*, vol. 6, pp. 40–53, 03 2008.

[50] T. Deakin and T. G. Mattson, *Programming Your GPU with OpenMP: Performance Portability for GPUs*. The MIT Press, 11 2023. [Online]. Available: https://doi.org/10.7551/mitpress/14866.001.0001

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

[52] S. J. Pennycook, J. Sewall, D. Jacobsen, T. Deakin, Y. Zamora, and K. L. K. Lee, "Performance, Portability and Productivity Analysis Library," Mar. 2023. [Online]. Available: https://doi.org/10.5281/zenodo.7733678

[53] A. Alpay and V. Heuveline, "Adaptivecpp stdpar: C++ standard parallelism integrated into a sycl compiler," in *Proceedings of the 12th International Workshop on OpenCL and SYCL*, ser. IWOCL '24. New York, NY, USA: Association for Computing Machinery, 2024. [Online]. Available: https://doi.org/10.1145/3648115.3648117

[54] "A Unified Executors Proposal for C++ — P0443R14 — open-std.org," https://www.open-std.org/jtc1/sc22/wg21/docs/papers/2020/p0443r14.html, [Accessed 13-08-2024].

[55] "Sparse matrix vector multiplication - part 1 - AMD lab notes — gpuopen.com," https://gpuopen.com/learn/amd-lab-notes/amd-lab-notes-spmv-docs-spmv_part1/#gpu-kernel-implementations, [Accessed 13-08-2024].

APPENDIX A
ARTIFACT DESCRIPTION/ARTIFACT EVALUATION

A. *Overview of Contributions and Artifacts*

1) Artifact repository: https://github.com/alpha-unito/gaia-avugsr-p3hpc

2) Paper's Main Contributions: We measured the performance portability (Φ) of five optimized re-implementations of the Gaia AVU-GSR code written in CUDA, HIP, SYCL, C++ PSTL, and OpenMP with GPU offload (§V-B). We also verified the correctness of the different optimized frameworks against a reference solution (§V-C).

3) Computational Artifacts: The artifact presented in this article is the Gaia AVU-GSR code, which executes the iterative LSQR routine to solve a system of equations (Eq. (2)). The LSQR routines for the different frameworks can be found in the `lsqr_hip.cpp`, `lsqr_stdpar.cpp`, `lsqr_openmp_gpu.cpp`, `lsqr_sycl.cpp`, and `lsqr_cuda.cu` source code files. Each of these routines can be executed by `solvergaiaSim.cpp` solver code, depending on how you compile `solvergaiaSim.cpp`.

Inside the repository, the following artifacts are available:

1) `README.md`: file with the compilation and execution instructions of the code.
2) `LICENSE`: license file with the licence under which the code is released.
3) `Makefile.examples`: Some samples Makefiles to show how to compile the source code for all the different architectures.
4) `src` folder containing all the source code.
   - `solvergaiaSim.cpp`: solver source code.
   - `lsqr_*`: source codes with the different LSQR implementations.
   - `gaiasim_includes`: common headers files
   - other source files
5) `include`: common headers files for the Gaia AVU-GSR application
6) `Scripts`: This folder contains, sorted by used architecture, a sample SLURM script to compile and execute the Gaia AVU-GSR application. Those files are merely examples and must be modified according to the local machine configuration (i.e., compilers' paths and source code's paths must be adjusted accordingly). You will find different folders in the `Scripts` folder (one for each GPU architecture). Inside the architectures folders, two more folders are present:
   - `comp`: Sample SLURM scripts to compile the code, one for each programming framework
   - `test`: Sample SLURM scripts to execute the Gaia AVU-GSR application, one for each programming framework

To better help with the understanding of the GPU architectures and cluster name association, Table IV shows how to interpret the content inside the `Script` folder.

GPU - Cluster reference table	
Cluster name	GPU vendor & model
CascadeLake	NVIDIA V100s
TeslaT4	NVIDIA T4
EpiTo	NVIDIA A100
GraceHopper	NVIDIA H100
Setonix	AMD MI250X

TABLE IV: Cluster name to GPU model reference table

B. *Artifact Identification*

1) Relation To Contributions: Two classes of experiments were performed:

1) In §V-B, we tested the Φ of the five optimized implementations of the solver across five different platforms, four based on NVIDIA GPU architecture (H100, A100, V100 and T4) and one based on AMD GPU architecture (MI250X). The Φ is objectively measured with a metric defined by John Pennycook (Eq. (1)). This test aims to evaluate if the different languages provide portable solutions without a significant performance loss. The results of these tests are illustrated in Figure 3.
2) In §V-C, we validate the correctness of the five optimized implementations of the code against the CUDA version in production, by taking as input two benchmark datasets with a size of 42 GB and 306 GB.

2) Expected Reproduction Time (in Minutes): A single execution of `solvergaiaSim.cpp` (100 iterations with a single version of LSQR among HIP, C++ PSTL, SYCL, CUDA, or OpenMP) should not exceed 5 minutes.

C. *Artifact Setup (incl. Inputs)*

The hardware and software employed in our work are described in §V and Table I.

a) Datasets / Inputs: The datasets used by the Φ are synthetic but distributed in the system as the real data. In the `README.md` file, the instructions on how to run the different tests with an input dataset of a given size are present. The size of the dataset to be passed in input to the system to solve is given at runtime in GB to the executing command of the solver. Once the solver reads this parameter, it randomly generates, given a certain seed, a dataset with the specified size and it runs the solution for 100 iterations, as specified in the scripts in the `Scripts/<target_arch>/test` directory inside the code repository). In these cases, it was not important to obtain the solution at convergence but to measure the iteration time, that 100 iterations proved to be sufficiently stable.

The real datasets used in the code validation section are produced by the ESA Gaia mission and cannot be shared due to a non disclosure agreement, hence motivating the need to provide a synthetic (but mathematically equivalent) dataset to perform tests.

b) Installation and Deployment: To compile the code, gcc version ≥12.2.0 is needed. The compilers used on NVIDIA and AMD architectures are detailed in Table I of the article. Remember to check all the paths inside the compilation scripts to reflect your local installation, particularly the environment variables that define the libraries paths (which are documented inside the scripts), as well as the source code relative and absolute paths inside the scripts and Makefiles.

1) Artifact Execution: We briefly summarize the procedure adopted to execute the tests detailed in §V. The two classes of tests are independent with each other, no output from one test are taken as input to the other test.c

In §V-B, we measured the Ɏ of the five optimized code versions on the five considered platforms with the metric defined by John Pennycook, reported in Eq. (1). The Ɏ results are reported in Figure 3. To calculate the Ɏ, the solver was executed on one GPU of Gracehopper (H100), EpiTo (A100), CascadeLake-V100, CascadeLake-T4, and Setonix, with an input memory set to 10 GB. The same test were executed for an input dataset of 30 GB only on H100, A100, V100, and MI250X and for an input dataset of 60 GB only on H100 and MI250X.

The LSQR was run for 100 iterations. To evaluate the Ɏ, we calculated the "application efficiency" that we derived, in turn, from the "best observed performance" of a specific code version among all the considered platforms. We evaluated the "best observed performance" of each code version as the lowest LSQR iteration time among the considered platforms. We measured the iteration time maximized among all MPI processes and averaged among 100 iterations.