Ref. Ares(2022)1116652 - 15/02/2022

D2.3

Third release of MAX software: Report on the release of documentation of the performance optimised parts

Daniel Wortmann, Stefano Baroni, Augustin Degomme, Pietro Delugas, Stefano de Gironcoli, Andrea Ferretti, Alberto Garcia, Luigi Genovese, Paolo Giannozzi, Anton Kozhevnikov, and Nicola Spallanzani

Due date of deliverable	31/01/2022 (**month 38**)
Actual submission date	15/02/2022
Final version	15/02/2022
Lead beneficiary	JUELICH (participant number 4)
Dissemination level	PU - Public

Document information

Project acronym	MAX
Project full title	Materials Design at the Exascale
Research Action Project type	European Centre of Excellence in materials modelling, simulations and design
EC Grant agreement no.	824143
Project starting/end date	01/12/2018 (month 1) / 31/05/2022 (month 42)
Website	http://www.max-centre.eu
Deliverable no.	D2.3
Authors	Daniel Wortmann, Stefano Baroni, Augustin Degomme, Pietro Delugas, Stefano de Gironcoli, Andrea Ferretti, Alberto Garcia, Luigi Genovese, Paolo Giannozzi, Anton Kozhevnikov, and Nicola Spallanzani.
To be cited as	Wortmann et al. (2021): Third release of MAX software: Report on the release of documentation of the performance optimised parts. Deliverable D2.3 of the H2020 CoE MAX (final version as of 15/02/2022). EC grant agreement no: 824143, JUELICH, Germany.

Disclaimer

This document's contents are not intended to replace consultation of any applicable legal sources or the necessary advice of a legal expert, where appropriate. All information in this document is provided "as is" and no guarantee or warranty is given that the information is fit for any particular purpose. The user, therefore, uses the information at its sole risk and liability. For the avoidance of all doubts, the European Commission has no liability in respect of this document, which is merely representing the authors' view.

Contents

1 Executive Summary **4**

2 Introduction **5**

3 Performance and portability of MAX flagship codes, libraries, and components **5**

3.1 FLEUR . . . 5
3.2 QUANTUM ESPRESSO . . . 9
3.3 Siesta . . . 14
3.4 YAMBO . . . 18
3.5 CP2K . . . 26
3.6 SIRIUS . . . 27
3.7 BigDFT . . . 30

4 Conclusions and ongoing work **32**

Acronyms **32**

References **33**

1 Executive Summary

Ensuring an efficient use of the upcoming pre- and exascale architectures is one of the key targets of the MAX project. In this context WP2 focuses on efforts to provide the MAX flagship codes with performance portable computational kernels. Together with the activities reported in WP4, a trend analysis clearly shows that forthcoming HPC architecture (including those relevant for EuroHPC) will be largely heterogeneous and mostly dominated by GPU accelerators. Therefore, we focused our attention on the corresponding programming frameworks and challenges.

Our work is mostly focused on the specific needs of our codes and methods, and we employed a variety of software engineering approaches and programming models for GPU accelerators. Beyond the use of Cuda(-Fortran) to extend the CPU performance of the codes to NVidia-based GPU hardware, the directive based programming models OpenACC and OpenMP were employed to exploit the capabilities of hardware across vendors.

As a main outcome of these activities, all MAX -flagship codes are now able to exploit the compute power of at least the most widespread (Intel and AMD) CPU and (NVidia) GPU architectures. These efforts are well documented by corresponding performance figures as obtained on some of the most recent supercomputers in Europe. We not only achieve high single node performance or satisfactory utilisation of a single GPU/node, but also focused on parallelization efficiency and demonstrate in multiple benchmarks and use-cases the parallel scaling and efficiency over many nodes. Throughout this deliverable, these achievements are demonstrated by numerical example and benchmarks for each MAX flagship code.

Overall, this shows a clear path to the use of a significant share of the computational resources available on current supercomputers, as well as on future (pre-)exascale machines. Many kernels and computationally relevant parts of the codes and libraries have been redesigned to be able to provide performance using portable frameworks. Thereby, we prepare for future architectures like AMD-based GPUs (as e.g. used in the coming pre-exascale machine LUMI) or Intel GPUs (expected to be released soon). However, due to the lack of hard- and software support results are still sparse for these efforts, and we expect to be able to obtain convincing numbers only after these systems are out into production and their compilers and low-level software is released.

2 Introduction

The goal of providing our codes with performance on current and future supercomputer architectures is the driving force behind the efforts reported here. All codes made significant progress in the last three years of the MAX project and this report will both include new developments and an overview of the achievements made.

A key objective of our work is to cover all relevant computing architectures, with special focus on the EuroHPC program. This amounts to ensure that our codes perform well on CPU-only machines (homogeneous architectures) as well as on supercomputers with accelerators (heterogeneous architectures). After the stop of the Intel MIC kind of processors this currently means machines equipped with GPUs.

The coverage of different computational kernels, different programming paradigms and different computer systems was a clear target of our work. In this sense, we favoured the full coverage of the flagship codes with their specific requirements over the consistency of the approaches and their final presentation. The different results presented in this report therefore give a good impression of both the complexity of the task as well as of the progress made. The related measures and achievements for all codes are reported in the following sections.

Together with WP4 we also explored possibilities of a common presentation and metric. However, as most of the efforts reported here are difficult to compare systematically due to their very different scope, we focused on a more detailed, activity-specific presentation style. With the performance of the flagship codes as our ultimate goal, we report the achievements mainly in these terms.

The measures reported here are also strongly interconnected to the work reported in WP1 and WP3, focusing mostly on the software design and the functionality, respectively. For example the separation of libraries as pushed forward in WP1 directly translates into specific computational kernels relevant for efforts to provide performance portability. Similarly, some functional advances of WP3 are relevant for the performance tuning pursued here.

3 Performance and portability of MAX flagship codes, libraries, and components

After the identification of performance portability challenges, possibilities and bottlenecks as reported in D2.1, we continued to work on improving the performance of the flagship codes. In the following we report a summary of the latest achievements on a code-basis.

3.1 FLEUR

The work on FLEUR in WP2 mainly focused on the implementation of a performance portable code to evaluate hybrid functionals. This work should be seen in conjuncture with the efforts of WP1 aiming at a modularization of this code section. All code relevant for this work is included in the final release MAX -R6.0 of the FLEUR code available at the FLEUR webpage (https://www.flapw.de).

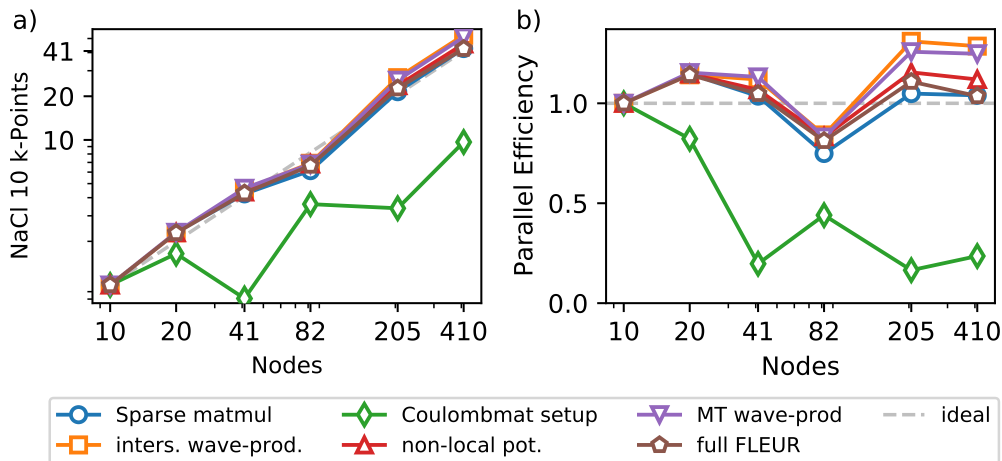

Figure 1: Scaling (a) and parallel efficiency (b) of the MPI parallelisation of the hybrid functional code. The extremely high scalability of most code parts (with the exception of the construction of the Coulomb matrix, which however is not dominating the total run-time) enables efficient use of the resources. The slight over-ideal performance increase can be tracked down to a better memory access for the parallel cases.

Identifaction of relevant kernels for hybrid functionals

Due to the nature of the LAPW method, the evaluation of the exchange interaction as key part of the hybrid functionals can be split into different computational kernels. Basically, main steps of the algorithm can be identified as: (i) the evaluation of the Coulomb kernel in a special mixed product basis (MPB) set, (ii) the projection of wavefunction products onto the MPB set and (iii) the transformation of the Coulomb matrix into the LAPW basis using the calculated projections. Out of these three steps the latter two are the computationally most relevant and thus the majority of effort was focused on these code parts. The projection operations have to be performed in the interstitial region as well as in the MT spheres. While the interstitial projects can be mapped on a series of FFT applied to the wavefunctions and the MPB, the contribution from the atomic MT spheres has to evaluated in a self-written kernel. The third step is actually a linear algebra operation which can be rewritten as a series of matrix-matrix multiplications and this refactoring leads to a highly efficient code due to the possibility of using appropriate third party libraries.

Parallelization over kq-pairs

The usage of supercomputers with their huge number of compute nodes requires a suitable distribution of the computational tasks. In the case of the hybrid functional implementation of `FLEUR`, this has been achieved on two levels. On the one hand, the different kq-pairs for which the parallelization is very efficient as the resulting kernels are independent and minimal communication is needed, and on the other hand a distribution over the different eigenstates that does induce more communication and also the need to recalculate the same quantities on several nodes in some cases. Both parallelization strategies combined can be adopted to efficiently parallelise the calculation of the hybrid func-

tionals over many independent nodes and due to the low communication requirements this is very portable between different supercomputing architectures. No specific high-bandwidth network is required, only a synchronous execution and distribution of the data must be ensured. We would like to stress here, that this parallelism is already available with rather few k-points, and can thus be exploited even for large setups.

Single Node Performance, GPUs

The key cornerstone to achieve performance portability on various architectures besides a suitable parallelization strategy over many nodes as outlined above is of course a suitable approach to single node performance. Our approach was based on:

- Identification of standard math problems which are solved with very high efficiency by vendor/external libraries and to which the performance relevant parts of our algorithms can be mapped to.
- The construction of custom kernels using both OpenMP and OpenACC programming paradigms to enable performance portable implementations of parts of the algorithm that can not be treated as standard math problems.

In our case two important computational math-kernels can be identified. On the one hand the FFT can be used to solve the challenge to calculate the interstitial contribution to the wavefunction projection onto the auxiliary basis set and matrix-matrix multiplies can be used to project the Coulomb basis onto the LAPW basis set in the final step of preparing the non-local potential.

When porting the hybrid functionals to the GPU (see Fig. 2) one can observe significant performance improvements in some parts of the code, most notably in the projection of the wavefunction products in the interstitial regions and the sparse-matrix multiply for the final projections. Other parts show little performance gain or might even be slightly faster on a CPU architecture. This can be understood in terms of their lower computational complexity and their lower degree of exposable parallelism. However, these code parts are significantly less relevant for the total runtime and it has proven to be much less efficient to calculate these contributions on the CPU as this would require significant data transfer between the CPU and GPU memory. Hence, we concluded that even though some of our kernels are not well suited for GPU computing the aspect of removing data transfer is more important for performance. At the same time this observation might also be used to adjust the distribution of the computations and the data in other systems to increase performance portability.

Single kq-pair

While we demonstrated excellent scalability for many nodes and good performance on a single GPU in the previous sections, the applicability of the code ultimately is determined by the possibility to calculate a single kq-pair efficiently on a nodes with more than a single GPU as such a computational setup seems to be the currently dominating architecture. As we have demonstrated, the additional scaling on many nodes using the parallelism over the kq-points is then a straight forward extension. Hence, we studied the scalability of the code using up to four GPU per node and up to 16 total GPU for this problem. As seen from Fig. 3 we can very efficiently utilises such an architecture.

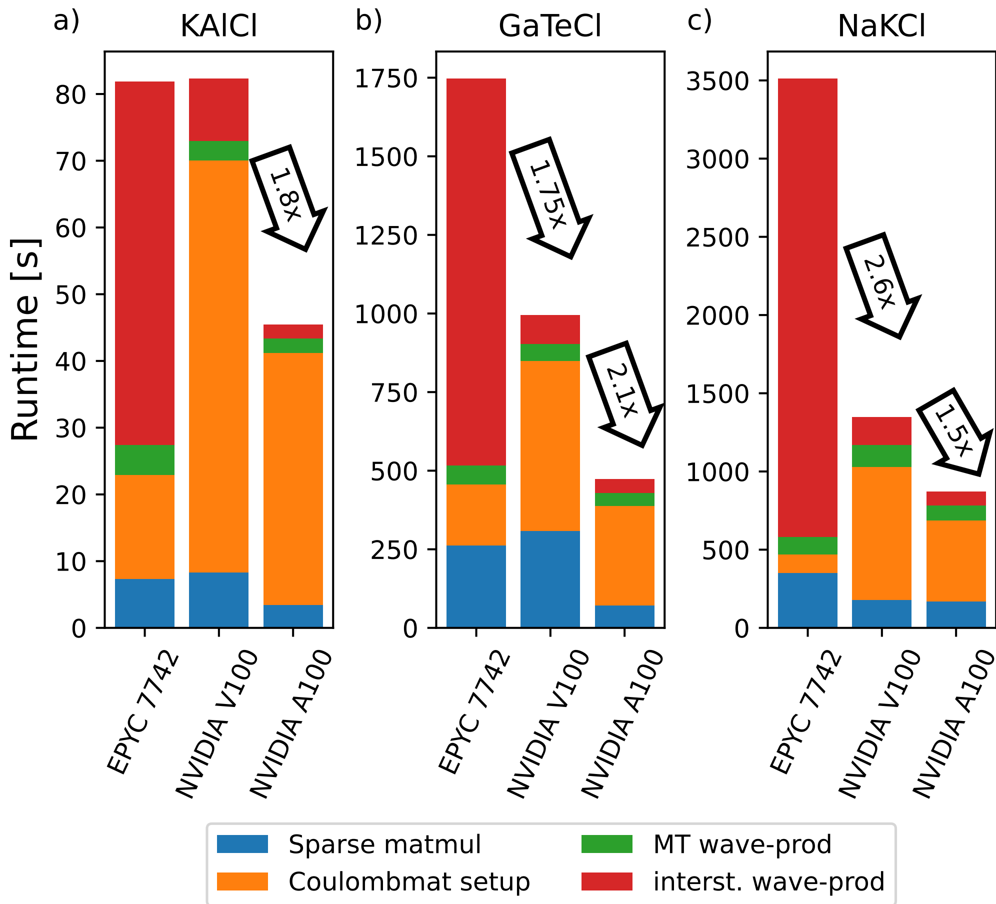

Figure 2: Comparison of the FLEUR runtime of the non-local potential calculation of three test systems on three different architectures. A 24 atom KAlCl, the 44 atom GeTeCl and the 64 atom NaKCl setup calculated on three systems: On an AMD EPYC 7742 CPU, an NVIDIA V100 and an NVIDIA A100 card.

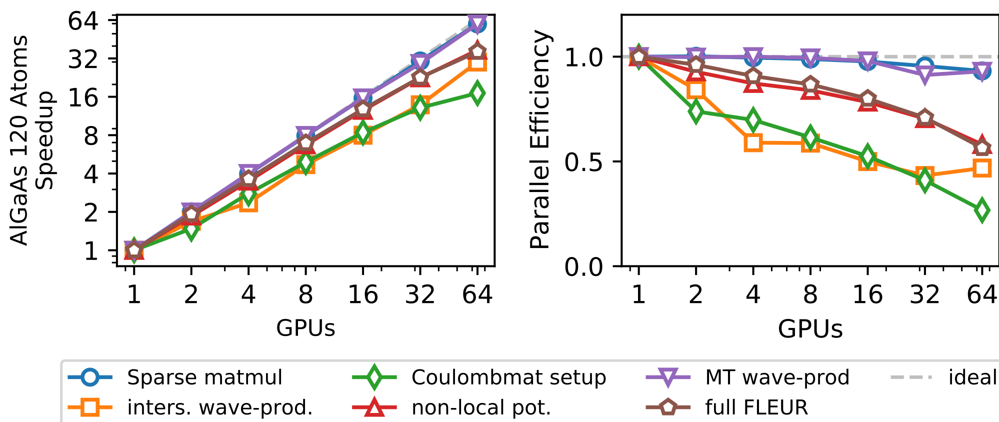

Figure 3: Scaling behaviour for a single k-point on the JURECA-DC GPU supercomputer featuring 4 A100 GPU cards per node. In the left plot the speedup on a double logarithmic scale is shown, while the right columns features the corresponding parallel efficiency.

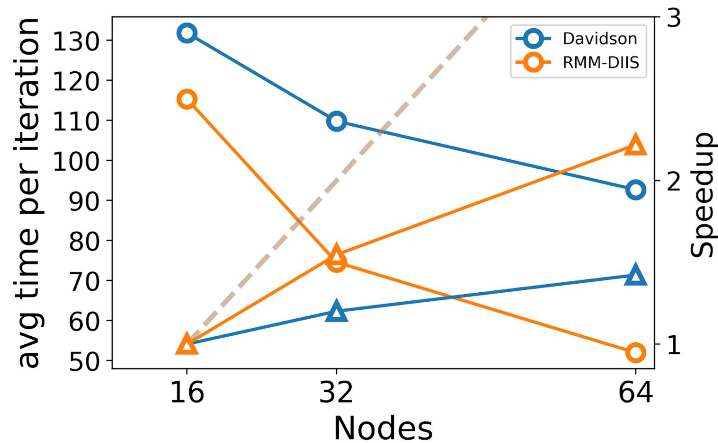

Figure 4: Improved scalability in pw.x. Field test with the CNTPOR benchmark (see D4.5 [3]) run on Marconi-A3@CINECA (Intel Skylake 48 cores per node). The new RMM-DIIS solver trades most of the computation time spent in poorly scaling large dense matrices' diagonalization for more computation time spent in the efficiently parallelized kernels, whose workload is either distributed in the R&G MPI group ranks, or offloaded into the accelerator. The dashed reference line represents the speedup expected at full parallel efficiency.

Concluding, we can claim that the hybrid functional implementation in FLEUR can demonstrate very high performance on CPU as well as GPU architectures with a scalability that easily extends into the high PetaFlop range.

3.2 QUANTUM ESPRESSO

As shown in other reports [1, 2, 3, 4], QUANTUM ESPRESSO has reached a good performance in many traditional homogeneous CPU-based machines as well as on heterogeneous architectures based on CPUs + NVIDIA GPGPUs.

Figures 5 and 6 make evident the differences in the parallelization strategies used on different architectures. For homogeneous systems most of the performance comes from the computation distribution among the MPI ranks. The main work-load division is done among the R&G group ranks, with the distribution of the plane-wave coefficients and of the scalar FFTs. At this level we also use openMP multithreading, that is used inside the FFT kernel and to parallelize the loops over the plane-wave coefficients. On top of this fundamental level, the MPI parallelism exploits the band group to distribute the work on wave functions; and the pool group to distribute the work on the Hamiltonian blocks.

In heterogeneous systems most of the parallelization is done among the accelerators' threads instead. The loops on plane wave-coefficients and the FFT kernels are all offloaded within the accelerator and the the FFTXlib workflow has been specifically optimized for the most efficient usage of the accelerators (see FFTXlib paragraph below). The MPI R&G parallelism has the function of distributing the data structure, so as to avoid memory issues. The heterogeneous implementation is apparently more efficient as it yields significantly shorter times-to-solution; and a much lower cost in terms of node-hours expended by the calculations.

The MPI scalability in the heterogeneous architectures is obtained mostly by exploiting the auxiliary levels of parallelism (bands groups and pools). The pool parallelization

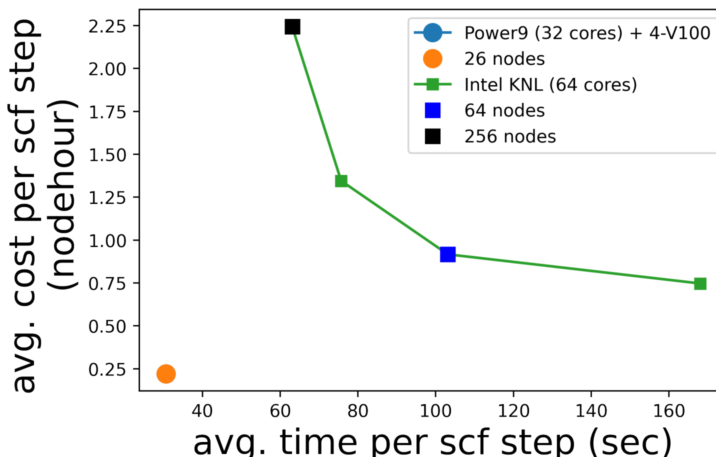

Figure 5: Computational cost for the CNTPOR benchmark (see https://gitlab.com/max-centre/benchmarks/-/tree/master/Quantum_Espresso/PW/CNT10POR8) in different platforms: (**circle**) heterogeneous nodes with Power9 CPU (32 cores) and 4 NVidia V100 GPUs; (**squares**) homogeneous nodes with Intel Knights Landing CPU (64 cores) (see [1, 2] for more details).

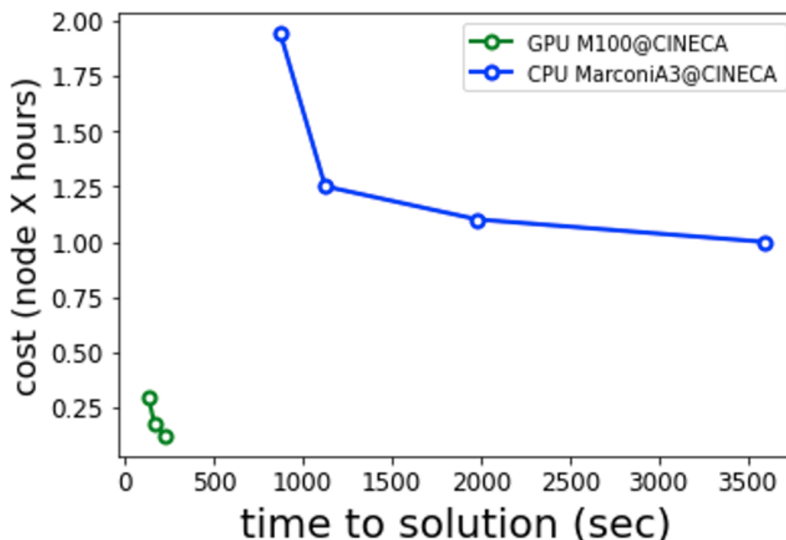

Figure 6: Computational cost for ZrO2 benchmark case with `cp.x` (see https://gitlab.com/max-centre/benchmarks/-/tree/master/Quantum_Espresso/CP/ZrO2/supercell_11layer) in different platforms (see [3] for more details). The different behavior for the two curves reflects the different choice in distributing the calculation.

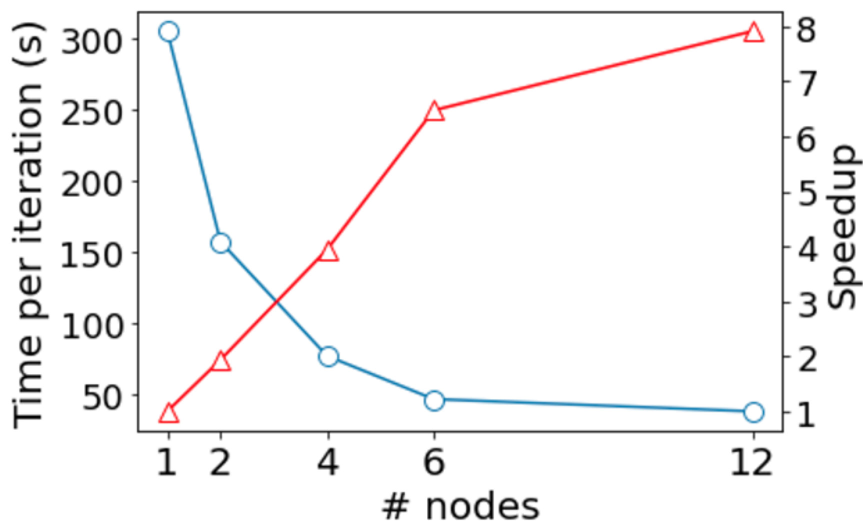

Figure 7: Average time per iteration (**circles**) and speedup (**triangles**) for the GrCoIr benchmark run on an A100 cluster. Each node is provided with 8 A100 NVidia Cards. The calculation can be distribute up to 12 pools-groups. After 6 nodes the plot show the speedup of 4 vs. 8 GPUs per pool.

is already very efficient, as reported in Figure 7. More work is ongoing for the enhancement of the band-group parallelism so to take full advantage of the new iterative solvers developed in WP3 (see D3.4 report). The new implementation of the RMM-DIIS iterative solver (for more details see the Q. ESPRESSO section of D3.4 [5]) has improved the scalability of `pw.x` for large calculations. The improvement shown in Fig. 4 is given by the significant reduction of the calculation time spent in poorly scaling large dense diagonalizations, traded with an increase of the time spent in operations efficiently parallelised in the `R&G` MPI group or with the accelerator threads.

For the portability on systems based on CPUs, we mostly rely upon the efficiency of the software stack (`Fortran` compiler, FFT and linear algebra libraries), with very few machine-specific adjustments. In the heterogeneous case, the portability is more problematic, due to the lack of well-established standards. Ad-hoc programming models are generally needed for each type of device. It seems, however, that such issue will soon be mitigated by the foreseen adoption of either `openACC` or `openMP` standards by all main vendors of accelerated hardware. We are thus currently recasting the high-level code layers of QUANTUM ESPRESSO towards a comprehensive adoption of these directive-based approaches. The first outcomes of this action are already included into the latest release (`qe-7.0`), where most of the offloading implementation in quantum engines is done with `openACC` directives. This action will be continued by completing the `openACC` porting and by adding the `openMP` implementation.

In the rest of the section we will first summarise the optimisation actions undertaken during the MAX phase-2 project. In the last paragraph we will then present the ongoing actions on the performance portability of QUANTUM ESPRESSO.

High-level parts of the code. At the higher level, we have been able to maintain the same organisation and workflow in all the architectures. With the modularization, most of the compute-intensive parts have been moved to the mathematical libraries. The remaining load is constituted by domain specific loops such as those over the plane-wave coefficients. These loops are distributed among the ranks of the main FFTXlib MPI group (called `R&G`) and, within each process, on many threads. In simulations of systems containing a large number of atoms, the nested loops on the atomic indices also become increasingly heavy (e.g. the DFT-D3 dispersion correction term) and it has been necessary to parallelise them. For accelerated machines these loops are offloaded to accelerators.

FFTXlib. This library performs distributed 3D FFT transforms using basis sets defined by a spherical cutoff. This is a domain-specific, compute-intensive, and recurrent task in the QUANTUM ESPRESSO workflows. In reciprocal space, data are distributed among MPI ranks in sticks along the z direction. In real space data are distributed as slabs and slab-slices. The 3D transform is computed as a succession of 1D FFTs on stick, or 2D FFTs on slabs. The 3D data distribution is transposed from the z-stick distribution to the z-slice via a sequence on MPI all-to-all calls. As the 3D-FFTs are executed on a large number of wave-functions, the optimisation actions have targeted the enhancement of the capability to process many wavefunctions concurrently. Different solutions are used depending on the system. For homogeneous machines with a large number of cores, the most efficient solution is usually to split the R&G MPI group in many different task groups, each acting on a different set of wave-functions. In nodes equipped with accelerators, many local 1D or 2D FFTs are performed on the GPU on a batch overlapped with MPI all-to-all calls for another batch.

LAXlib. Parallel linear algebra represents the other most compute-intensive kernel calls by QUANTUM ESPRESSO quantum-engines. Optimal performance for this kernel is obtained by the usage of specialised libraries. The `LAXlib` library provides wrappers for initialisation and usage of the main specific libraries. Distributed linear algebra is currently used for homogeneous machines (Scalapack, ELPA), while GPU specific libraries (e.g. cublas) are used for GPUs.

The `LAXlib` library provides also some auxiliary routines for splitting or collecting the distributed matrices, and for performing some general matrix operations. These have also been enabled and specifically for operating on matrices allocated on the accelerators.

XClib. These kernels compute the exchange-correlation functional energy contribution and their derivatives on a 3D data grid. By construction, they are ideal for multi-thread vectorisation or, in machines with accelerators, for offloading. For what concerns the latest version of QUANTUM ESPRESSO, the whole library has been accelerated with `openACC`. The GPU porting is in particular extremely efficient with up a $10\times$ (see fig. 8) with respect to the CPU-only version, depending on the grid size.

Ongoing and future work. Most of the work currently ongoing aims at expanding the performance portability of the code for what concerns the heterogeneous systems equipped with non-CUDA GPGPUs. As mentioned above part of this effort is the total conversion of the offloading specification in the higher-level code layers to `openACC` and

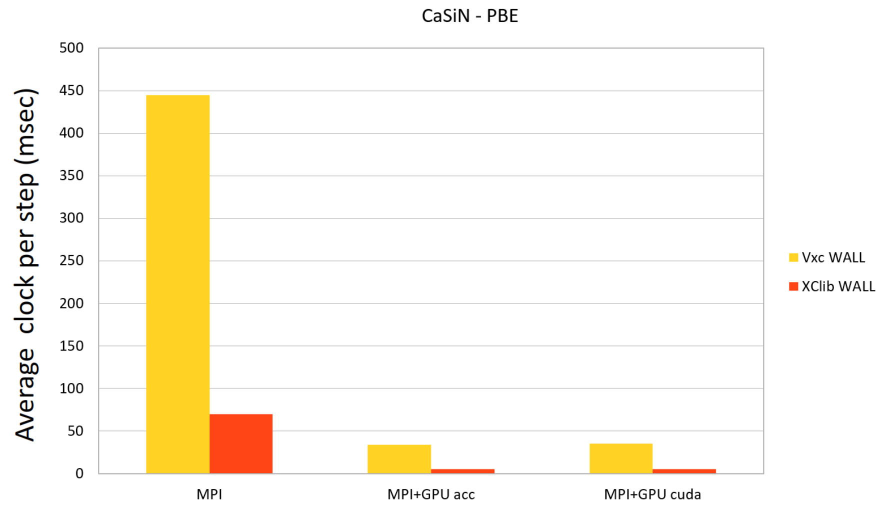

Figure 8: GPU optimization of the XClib library with a $10\times$ acceleration for the evaluation of exchange-correlation functionals in distributed real space grids. The new more portable openACC refactoring maintains the same performance as the first CUDA specific version.

openMP. The usage of these directive-based programming models is also speeding up the acceleration of other applications of the QUANTUM ESPRESSO suite. In particular, the openACC porting of the phonon code (ph.x) is currently underway and part of this work could be directly reused for all the other applications related to the DFPT formalism. For what concerns the performance portability of the main kernels we are currently experimenting in LAXlib the wrappers for HIP and ROCm libraries on AMD GPUs. This work will be finalised in the MAX hackathon that will be held in February 2022. Similar experimentation in LAXlib has been done with the MKL libraries for GPUs on the Intel devcloud platform.

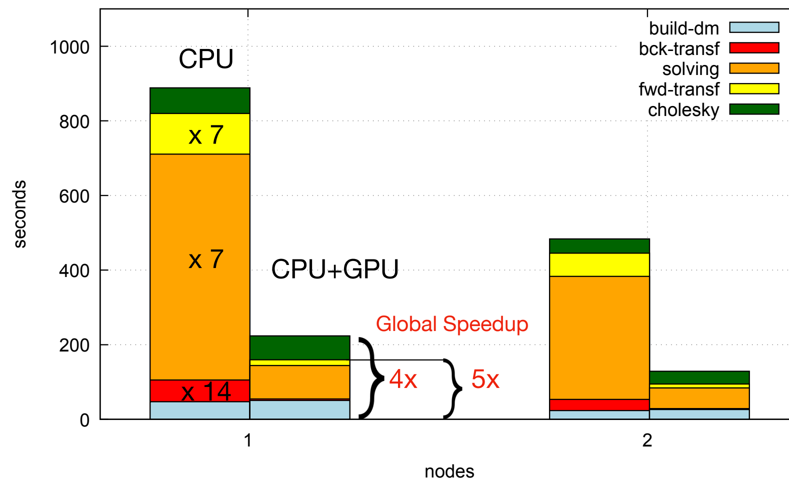

Figure 9: Speedup of the diagonalization comparing the CPU only version with the CPU+GPU implementation. The contributions of the different steps are shown individually highlighting the different level of GPU porting.

3.3 Siesta

In the past year, the performance enhancements for the SIESTA program have come mainly from the implementation of new algorithms. Further details are given as part of the report of the WP3 activities, but we can summarise the enhancements in the following list:

- Implementation of a basis-contraction scheme to reduce the cardinality of the basis set without compromising the accuracy of the calculations.
- Refactoring of the linear-scaling sub-system to abstract the underlying matrix operation and take advantage of accelerated back-ends.

Global achievements in MaX-phase2

In SIESTA, the performance-portability is almost completely linked to the use of appropriate external solver libraries, as the lion's share of the CPU time spent by the program is associated to the solver part. We have considerably extended the performance enhancement possibilities of the code by the completion of the interface to the ELSI library of solvers, and by taking advantage of the improvements in the ELPA library. The performance enhancements come in three significant fronts:

GPU acceleration. The GPU acceleration of the electronic-structure solver based on diagonalization has been achieved through the use of GPU-enabled versions of the ELPA library, available in stand-alone form or through the ELSI interface layer implemented in SIESTA as part of the WP1 activities.

The acceleration improvements are shown in detail in Fig. 9. The Cholesky step factorises the overlap matrix, a prerequisite for the transformation of the generalised eigenvalue problem into a standard one (which is the second step). The main phase is the

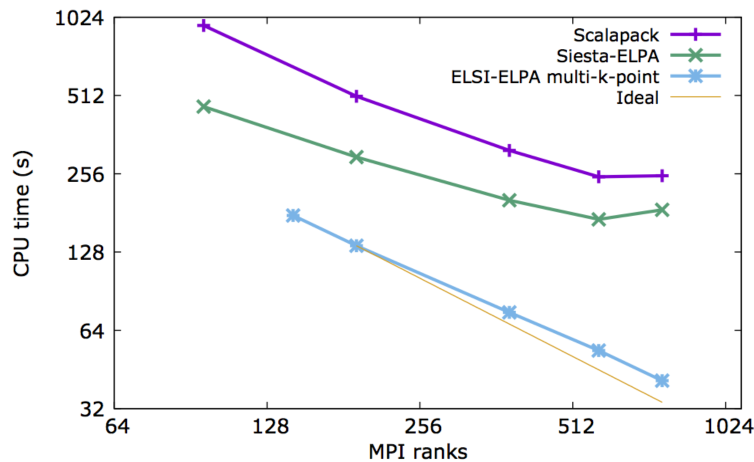

Figure 10: Performance improvement from the use of the extra level of parallelisation over **k**-points in Siesta using the ELSI interface with the ELPA solver, compared to the previous diagonalisation scheme (using both the standard Scalapack solver and the existing ELPA interface in Siesta). The system is bulk Si with H impurities, with 1040 atoms, 13328 orbitals, and a sampling of 8 k-points. The multi-k scheme is able to stay closer to ideal scalability for larger numbers of MPI processes.

solving of this standard problem. The original eigenvalue problem is formally completed after the back-transformation of the eigenvectors, but the full solution of the electronic-structure problem still needs the building of the density matrix (DM). The accceleration of the Cholesky and DM-building steps is being worked on by the ELPA and ELSI developers. Once these improvements are incorporated in the respective libraries, SIESTA will immediately benefit.

More levels of parallelization. A feature common in principle to all solvers is that the SIESTA-ELSI interface is fully parallelised over **k**-points and spins (support for non-collinear spin is in the works). This means that these calculations can use two extra levels of parallelisation (beyond the standard one of parallelisation over orbitals and real-space grid), see eg Fig. 10. In addition, the PEXSI solver, beyond a reduced scaling (at most $O(N^2)$ for dense systems, and $O(N)$ for quasi-one-dimensional systems) offers *two* further levels of parallelisation: over poles, and over trial points for chemical-potential bracketing. It can be used for large systems with very high numbers of processors.

Use of mixed precision. The ELPA solver can be invoked in single-precision mode, which can speed up the initial steps of the electronic self-consistent-field (scf) cycle. In fact, it has been shown that in SIESTA one just needs to perform one or two final scf steps in double precision to maintain the standard level of precision. This leads to substantial CPU-time savings (see Fig. 11).

Outlook for further improvements

As mentioned above, the PEXSI solver within ELSI offers several extra levels of parallelization: in addition to orbitals and **k**-points (when relevant), it can parallelize over

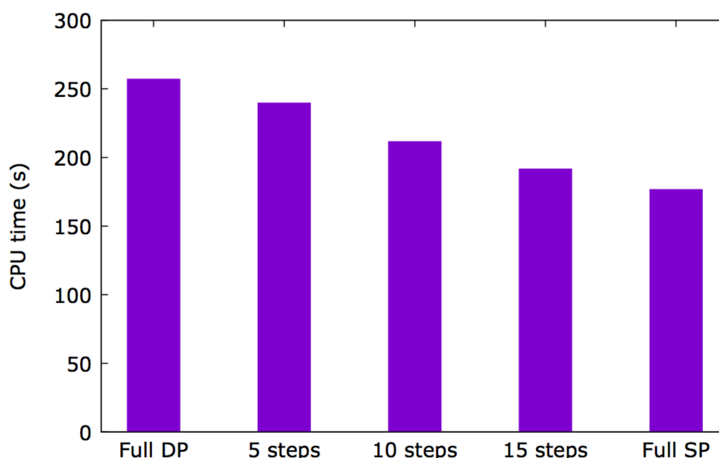

Figure 11: Performance improvement coming from the use of the mixed-precision mode in the ELPA solver. The system is a H-terminated Si quantum dot, with 1359 atoms and 14691 orbitals, and convergence of the scf cycle takes 16 steps. CPU time savings of approximately 30% can be obtained.

poles, and over chemical potential interpolation points. In Fig. 12 (taken from the D4.3 deliverable) we see that this scaling reserve of the PEXSI method means that, by simply increasing the number of tasks per pole `tpp`, it can use effectively many more tasks to provide much lower times-to-solution than the GPU-accelerated diagonalizer. If minimisation of the time-to-solution is the main goal, then the more favourable scaling of the PEXSI solver is key.

While very relevant for many projects, minimum time-to-solution is not the only possible goal. Users might want to maximise the return of their supercomputer allocation by carrying out as many jobs as possible, without regard (within limits) to the time involved. In this case, minimising the total cost (in node*hours) of a calculation is the relevant objective (this is also related to the optimization of the energy-to-solution). A good way to look at this balance of objectives is provided by Fig. 13. Here proximity to the lower-left corner represents the overall "goodness" of the method, but a user can choose to give an arbitrary relative weight to the two objectives. This plot encodes extra useful information: the (negative) slope of a line reflects the marginal cost of diminishing the time-to-solution, which is lower in the PEXSI method (for further details, recall the discussion in the report for D4.3). A further opportunity for performance enhancement is then the GPU acceleration of the PEXSI solver.

Further performance improvements and portability to new architectures are in the pipeline in several of the libraries used by Siesta. ELPA is adding support for AMD and Intel GPUs, and improving data communication patterns with the accelerators. Libxc is adding GPU support. DBCSR and Psolver, both part of the MaX effort, are also actively developed.

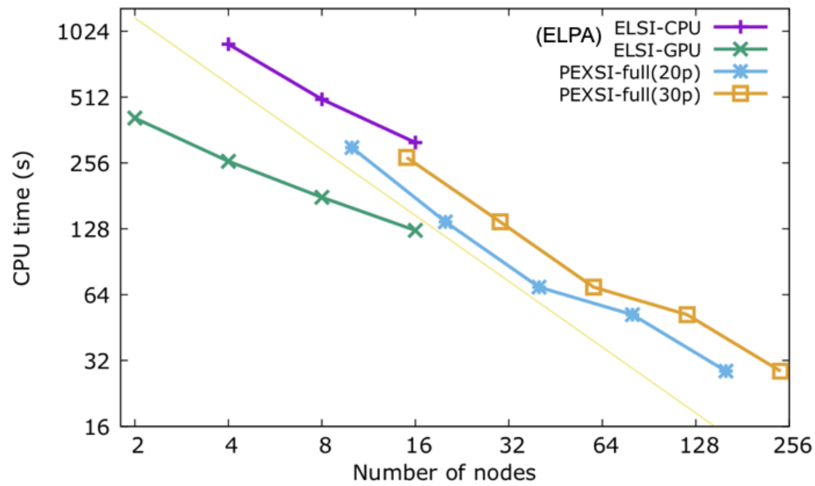

Figure 12: Time to solve the diagonalization problem corresponding to a piece of virus protein surrounded by water molecules, with approximately 58000 orbitals. Two sets of PEXSI results (for 20 and 30 poles) are shown, each for different numbers of tasks per pole (tpp) (from left to right: tpp=8, 16, 32, 64, 128). The thin line shows the ideal scalability behaviour. The calculations were done on Marconi-100, with 32 CPUs and 4 GPUs per node.

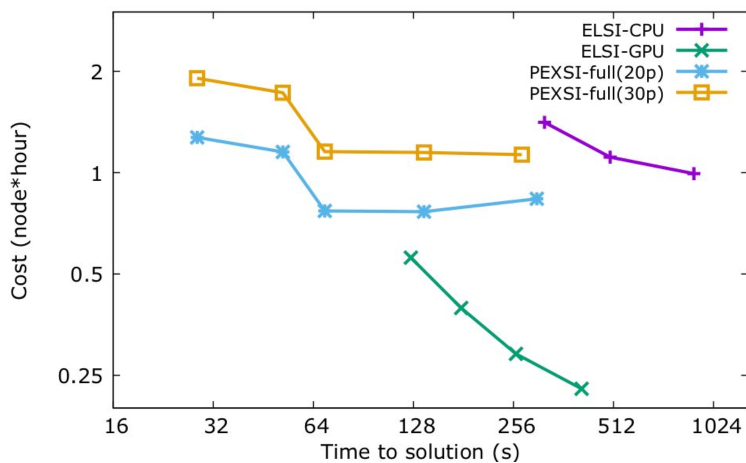

Figure 13: Total cost (per scf step) vs time-to-solution for the virus protein problem, with approximately 58000 orbitals. Details as in previous figure.

3.4 YAMBO

Following the first GPU-aware release of YAMBO at M12 (v4.5, Nov 2019), and the second enhanced release at M24 (v5.0, Nov 2020), here we report about the last year developments concerning performance portability and GPU-support of YAMBO. The most mature features described below are already included in YAMBO v5.1 (the most recent ones, or those still ongoing, being instead located in dedicated development branches, whose continuous integration assessment can be seen here: http://www.yambo-code.org/robots/index.php).
Most of the **development effort** has focused on the following lines:

- Maintenance and consolidation of the GPU-porting of YAMBO based on CUDA-Fortran (NVIDIA-GPUs), including testing and validation of GPU-aware distributed linear-algebra libraries relevant for YAMBO kernels;
- Development of software engineering solutions to support multiple GPU-programming models without disrupting the source base (oriented to support NVIDIA, AMD, and INTEL GPUs).

The **key achievements** accomplished in the last year are:

- Further optimisation of the memory allocation and distribution, with special focus on YAMBO GW and BSE runs on GPUs;
- DeviceXlib has been significantly reorganized to support multiple GPU-hardare and programming models, including CUDA-Fortran, OpenACC, and OpenMP5. Tests have been extended and demonstrated to work for all environments, notably including pre-release INTEL GPUs (in direct collaboration with INTEL personnel);
- Restructuring of the YAMBO source base to allow for multiple GPU-backend. Besides CUDA-Fortran, the OpenACC programming model has been explicitly adopted and demonstrated for selected kernels (including e.g. the Hartree-Fock self-energy).

Consolidation of the CUDA-Fortran porting

The CUDA-Fortran porting of YAMBO is proved to be an efficient tool for taking advantage of NVIDIA GPUs. It was therefore appropriate to consolidate it by porting also kernels and frameworks developed only recently (e.g. MPA, RM_W).

An issue that we faced testing the porting with systems with large memory footprint was the memory limit of some GPU cards (e.g. NVIDIA V100 with 16GB). In order to address the problem, at least partially, we started an optimisation of the memory usage that was in beta phase at M24. Now we can consider it largely tested and validated and so ready to be added at the next stable release.

As a critical point, the complete solution to the memory issue is related to the usage of libraries that allow for distributed linear algebra computations on the GPUs. In fact, when dense linear algebra, as involved in the solution of the Dyson equation for the screened Coulomb potential, becomes memory critical, at present we can only distribute the problem on CPUs via the standard Scalapack routines. Distributing on GPUs would

therefore be extremely beneficial in these cases. In view of this, we have started making experience with distributed linear algebra routines from NVIDIA (cuSOLVER_Mg). At first we focused on the solution of complex linear systems with multiple rhs. Eventually we were able to build an interface and to have the kernel working on multiple GPUs within one node. Currently, we are still in the process of evaluating alternative solutions, including e.g. cuSOLVER_Mp, or SLATE, to address multi-GPU multi-node solutions.

DeviceXlib: Strategy and development

As also discussed in D1.5, `DeviceXlib` is a domain-specific performance-portability library developed within the MAX consortium (mostly by the QUANTUM ESPRESSO and YAMBO teams), with the aim of hiding most of programming-model and vendor-specific instructions, while exposing abstract operations (memcpy to/from/within the GPU memory, equipped with allocation/deallocation/memset utilities, linear algebra interfaces, specialized common kernels, ...). The domain-specific nature of the library stands in the selection of the kernels and operations exposed to the developer.

`DeviceXlib` has been significantly restructured in order to allow for the use of **multiple GPU-oriented programming models** (such as CUDA-Fortran, OpenaCC, OpenMP5), multiple linear algebra libraries (notably, cuBLAS, RocBLAS, MKL_GPU), in turn targeting multiple GPU hardware (including NVIDIA, AMD, and INTEL GPUs).

At the moment **`DeviceXlib` have been positively demonstrated** (by running the internal tests) on NVIDIA architectures using multiple compilers and programming models. The library has also been deployed and run correctly on INTEL GPUs (pre-release hardware, in direct collaboration with INTEL personnel). Moreover, at the time of writing, a MAX Hackathon dedicated to AMD hardware (with special focus on the LUMI EuroHPC machine) is running.

Examples of the **software engineering strategies** adopted in `DeviceXlib` to support the above described programming models, libraries, and hardware are reported below. In particular we have taken advantage of the previous experience done with YAMBO (see e.g. D2.2) to hide CUDA-specific code, largely employing pre-compiler macros and Fortran-interfaces to avoid code-duplication.

```
!
! directive sentinels
!
#if defined __DXL_OPENACC
#  define DEV_ACC $acc
#endif
#if defined __DXL_CUDAF
#  define DEV_CUF $cuf
#endif
#if defined __DXL_OPENMP_GPU
#  define DEV_OMPGPU $omp
#endif
#if defined __DXL_OPENMP && !defined (__DXL_HAVE_DEVICE)
#  define DEV_OMP $omp
#endif

!
! motif for malloc routines
!
subroutine dev_malloc(array, range1 )
    [...]
#if defined __DXL_CUDAF
```

```
    if (.not.allocated(array)) allocate( array(d1s:d1e) )
    !
#elif defined __DXL_OPENACC || defined __DXL_OPENMP_GPU
    !
    !DEV_ACC enter data create( array(d1s:d1e) )
    !DEV_OMPGPU target enter data map(alloc: array )
    !
#endif

!
! motif for linear algebra wrappers
!
subroutine dev_CGEMM_gpu(TRANSA,TRANSB,M,N,K,ALPHA,A,LDA,B,LDB,BETA,C,LDC)
    [...]
    !DEV_ACC data present(A,B,C)
    !DEV_ACC host_data use_device(A,B,C)
    !DEV_OMPGPU target variant dispatch use_device_ptr(A,B,C)
#if defined __DXL_CUBLAS
    call cublasCgemm(transa_,transb_,M,N,K,ALPHA,A,LDA,B,LDB,BETA,C,LDC)
#elif defined __DXL_ROCBLAS
    ierr=rocblas_cgemm(rocblas_handle,transa_,transb_,M,N,K,...)
#elif defined __DXL_MKL_GPU
    call CGEMM(TRANSA,TRANSB,M,N,K,ALPHA,A,LDA,B,LDB,BETA,C,LDC)
#endif
    !DEV_OMPGPU end target variant dispatch
    !DEV_ACC end host_data
    !DEV_ACC end data

!
! motif for custom (automatic) functions/kernels
! here the case of a wrapper for the complex conjugation of device data
!
      !DEV_CUF kernel do(1)
      !DEV_ACC data present(array_inout)
      !DEV_ACC parallel loop collapse(1)
      !DEV_OMPGPU target map(alloc:array_inout)
      !DEV_OMPGPU teams loop collapse(1)
      !DEV_OMP  parallel do
      do i1 = d1s, d1e
        array_inout(i1 ) = &
            conjg (  array_inout (i1 ) )
      enddo
      !DEV_ACC end data
      !DEV_OMPGPU end target
```

Then, `DeviceXlib` exposes a number of APIs corresponding to abstract device-oriented operations that can be used in scientific codes such as YAMBO and QUANTUM ESPRESSO. When possible, single interfaces for device/host, kinds, ranks, precisions are provided (and automatically generated via python scripts when useful). A list of these utilities include:

```
! malloc/free
subroutine dev_malloc
subroutine dev_free
  function dev_allocated     ! logical enquire function
!
! memcpy and memset
subroutine dev_memcpy_h2d
subroutine dev_memcpy_d2h
subroutine dev_memcpy_d2d
subroutine dev_memset
!
! auxiliary functions
subroutine dev_conjg         ! complex conjugations
subroutine dev_vec_upd_remap ! v_out(:) = scal * v_in(map(:))
```

```
[...]
!
! linear algebra interfaces
  function dev_xDOT, dev_xDOTU, dev_xDOTC
subroutine dev_xAXPY
subroutine dev_xGEMV
subroutine dev_xGEMM
[...]
```

DeviceXlib is equipped with a number of unit tests to allow for continuous integration and is made available on the MAX Gitlab repository: https://gitlab.com/max-centre/components/devicexlib.

YAMBO with multiple GPU programming models

The porting strategy of YAMBO sketched in D2.2 has been consolidated in the source-base, and further developed to allow for openACC support. OpenMP5 support, already included in DeviceXlib, is also under development. The overall strategy is based on the following choices:

- In view of the available software stacks for GPU support, at present YAMBO needs to support multiple programming models in order to address multiple GPU architectures;
- Data representation in GPU memory is obtained either explicitly (as when using CUDA-Fortran) or implicitly via host pointers + memory mapping (as with OpenACC or OpenMP5);
- Extensive use of DeviceXlib is made, allowing YAMBO developers to handle explicitly, though abstractly, data memcpys to/from GPU memory, and to perform basic operations on such data (including linear algebra kernels).
- When specialised kernels are needed, extensive use of directive loop decoration is made (protecting directive sentinels via the pre-compiler macros DEV_OMP, DEV_CUF, DEV_ACC, DEV_OMPGPU, as described above for DeviceXlib). This already covers the largest part of computational loops in the code.
- When performance critical, explicit implementation of custom kernels could in principle be added (not used at present).
- As a result, YAMBO has a single source code supporting CPU (with OpenMP parallelism) and GPU (via CUDA-Fortran and OpenACC).

Currently, CUDA-Fortran porting is fully developed, OpenACC is partly developed (more and more kernels are being ported), while porting via OpenMP5 is planned in the short term (being already present in DeviceXlib). Practical examples of the above strategy are shown below.

```
!
! macros to be included
!
#ifdef _CUDAF
#  define DEV_SUB(x)         x##_gpu
#  define DEV_VAR(x)         x##_d)
#  define DEV_ATTR          , device
```

```
#elif defined _OPENACC || defined _OPENMP5
#  define DEV_SUB(x)        x##_gpu
#  define DEV_VAR(x)        x
#  define DEV_ATTR

#else
#  define DEV_SUB(x)        x
#  define DEV_VAR(x)        x
#  define DEV_ATTR
#endif

!
! From routine src/wf_and_fft/scatter_Bamp.F
!
subroutine DEV_SUB(scatter_Bamp)(isc)
 [...]
 complex(SP), pointer DEV_ATTR :: WF_symm_i_p(:,:), WF_symm_o_p(:,:)
 complex(SP), pointer DEV_ATTR :: rhotw_p(:)
 complex(DP), pointer DEV_ATTR :: rho_tw_rs_p(:)
 logical :: have_cuda_loc
 !
 ! define pointers to enable CUF kernels
 ! when compiling using CUDA-Fortran
 !
 WF_symm_i_p => DEV_VAR(isc%WF_symm_i)
 WF_symm_o_p => DEV_VAR(isc%WF_symm_o)
 rho_tw_rs_p => DEV_VAR(isc%rho_tw_rs)
 rhotw_p     => DEV_VAR(isc%rhotw)
 [...]

 !DEV_ACC data present(rho_tw_rs_p,WF_symm_i_p,WF_symm_o_p)
 !DEV_ACC parallel loop async
 !DEV_CUF kernel do(1) <<<*,*>>>
 !DEV_OMP parallel default(shared), private(ir)
 !DEV_OMP do
 do ir = 1, fft_size
   rho_tw_rs_p(ir) = cmplx(conjg(WF_symm_i_p(ir,1))*WF_symm_o_p(ir,1),kind=DP)
 enddo
 !DEV_OMP end parallel
 !DEV_ACC end data
 [...]
```

The above approach, jointly with the use of `DeviceXlib`, allowed us to implement the OpenACC porting of the HF kernel by leaving the high-lever sources mostly unchanged with respect to the CUDA-Fortran porting. A snippet from the `XCo_Hartree_Fock.F` source file is provided below:

```
do jb=Sx_lower_band,Sx_upper_band
  (...)
  call DEV_SUB(scatter_Bamp)(isc)
  (...)
  if (isc%is(1)/=iscp%is(1)) then
    call DEV_SUB(scatter_Bamp)(iscp)
  else
    ! iscp%rhotw = isc%rhotw
    call dev_memcpy_d2d(DEV_VAR(iscp%rhotw),DEV_VAR(isc%rhotw))
  endif
  !
  DP_Sx_l=DEV_SUB(Vstar_dot_VV)(isc%ngrho,DEV_VAR(iscp%rhotw),&
                        DEV_VAR(isc%rhotw),DEV_VAR(isc%gamp)(:,1))
  DP_Sx=DP_Sx + DP_Sx_l * const
  !
enddo
```

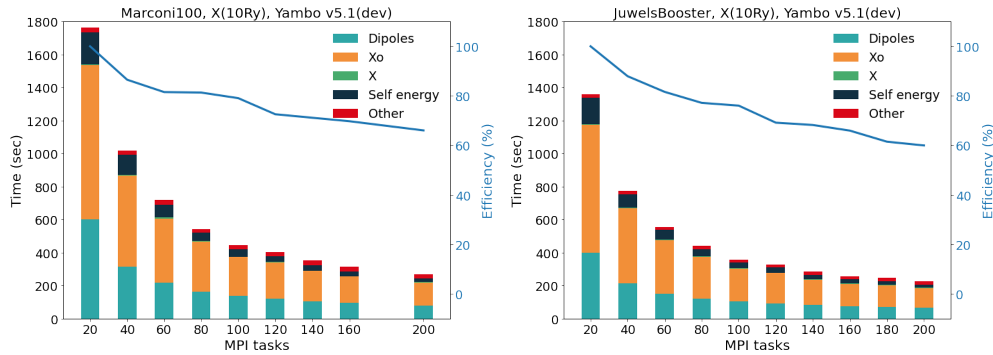

Figure 14: YAMBO parallel scaling and performance taken from runs of the Rutile-H benchmark. Data from Marconi-100 at CINECA (IBM-P9 + 4*V100, left) and Juwels-Booster at JSC (AMD-EPYC-Rome + 4*A100, right). All calculations run with YAMBO v5.1(dev) using 8 threads per MPI task.

MPI	Nodes	Wall time	Dipoles	X_0	X	Self energy	Other
			Marconi 100				
20	5	1763.0	600.0000	938.0000	2.2058	195.7894	27.0048
40	10	1019.0	314.6036	552.2515	2.4054	124.6729	25.0666
60	15	721.0	217.8861	387.6636	7.2033	78.5183	29.7287
80	20	542.0	165.0172	301.9688	2.3379	51.5035	21.1726
100	25	446.0	137.5690	235.7383	2.3202	45.1188	25.2537
120	30	405.0	123.2480	217.8780	2.3611	34.0016	27.5113
140	35	354.0	106.4343	182.5872	2.3460	33.1455	29.4870
160	40	316.0	95.4596	159.8472	2.4356	28.4254	29.8322
200	50	267.0	81.1682	137.8192	2.3841	22.6336	22.9949
			Juwels Booster				
20	5	1361.0	397.1855	777.0000	3.5280	159.3552	23.9313
40	10	774.0	213.2750	455.4877	3.7270	81.1500	20.3603
60	15	556.0	152.4184	324.4857	3.7884	56.4944	18.8131
80	20	441.0	121.5582	254.0378	3.6955	42.6529	19.0556
100	25	358.0	103.3167	197.9958	3.7542	34.7814	18.1519
120	30	328.0	91.2371	183.7931	3.8982	29.8318	19.2398
140	35	285.0	81.9063	153.4539	3.7370	27.4232	18.4796
160	40	258.0	76.2563	133.0946	4.8090	24.2360	19.6041
180	45	246.0	71.4286	130.2942	3.6830	22.0923	18.5019
200	50	227.0	67.0105	116.2699	3.7086	20.3843	19.6267

Table 1: YAMBO parallel scaling taken from runs of the Rutile-H benchmark. Data from Marconi-100 at CINECA (IBM-P9 + 4*V100) and Juwels-Booster at JSC (AMD-EPYC-Rome + 4*A100). All calculations run with YAMBO v5.1(dev) using 8 threads per MPI task.

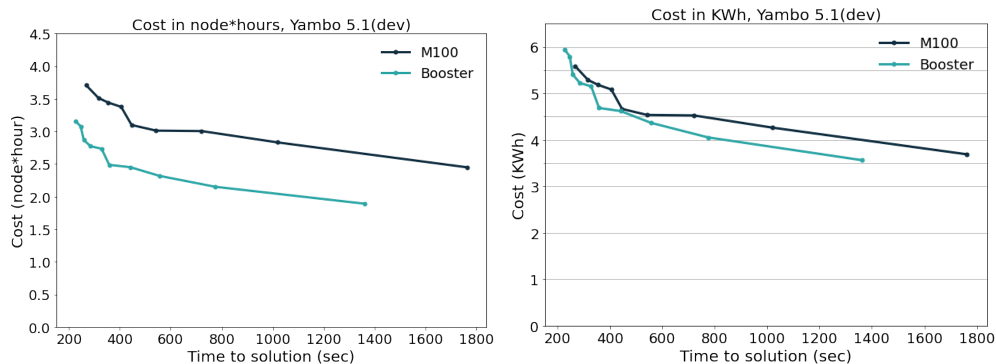

Figure 15: Rutile-H benchmark: node and energy costs of YAMBO v5.1(dev) runs performed on Marconi-100 at CINECA (IBM-P9 + 4*V100, left) and Juwels-Booster at JSC (AMD-EPYC-Rome + 4*A100, right).

Performance achieved

Following the work done in previous deliverable D2.2 [1], below we report the performance results obtained on Marconi-100 (M100) at CINECA, and we add the results obtained on the Juwels Booster module (Booster) at JSC, as a comparison. M100 deploys IBM Power9 nodes equipped with 4 NVIDIA V100 cards,[1] while Booster is built with AMD EPYC Rome 7402 nodes equipped with 4 NVIDIA A100 cards.[2] In this regard, the comparison of the results obtained will be mainly focused on the different generation of NVIDIA GPU mounted on the two machines.

In deliverable D4.5 [3] on profile and bottleneck identification we compared three versions of YAMBO (a pre-release of v4.5, v5.0, and a pre-release of v5.1) across M100 and Marconi-Skylake, using the Rutile-H use case of the MAX benchmark set (defined in D4.2 [6]), a defected $2\times2\times3$ TiO_2 rutile bulk supercell with an interstitial H impurity (72+1 atoms). Working on the same system, here we also add a comparison with the Booster machine, equipped with state-of-the-art NVIDIA cards.

Considering Fig. 14 and at the related Tab. 1, the performance improvement of runs on the newer NVIDIA A100 GPUs is quite evident. By comparing the data, the average speedup of about 1.26. Such performance enhancement is probably somehow limited by the nature of the system where the calculation of dipoles is a relevant part of the run. A deeper investigation in this kernel of the code is on schedule. The left panel of Fig. 15 reports the comparison between the two clusters in terms of the "node hours" cost measure. Then, the right panel shows the node*hour graph converted to energy cost by multiplying with the average power/node taken from the Top500 listing. The comparison shows that the use of newer generation GPUs results in a lower computational cost, across all calculations. In terms of energy cost we have a net advantage for a low level of parallelization, while for high number of nodes the the system studied probably becomes too small to support a high energy efficiency.

We started this Section by reporting the results of the Rutile-H benchmark for continuity with the previous deliverables. However, given above the considerations, we have also decided to study a new benchmark system (graphene/Co interface), that we plan to

[1]Marconi-100, https://www.top500.org/system/179845/
[2]Juwels Booster module https://www.top500.org/system/179894/

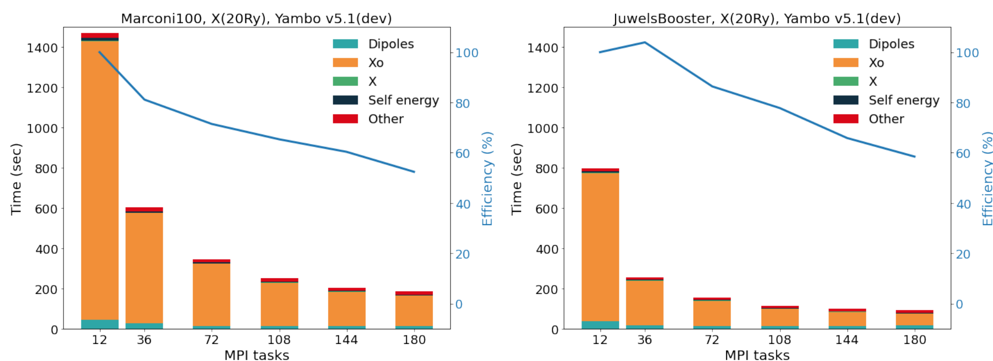

Figure 16: YAMBO parallel scaling and performance taken from runs of the GrCo benchmark. Data from Marconi-100 at CINECA (IBM-P9 + 4*V100, left) and Juwels-Booster at JSC (AMD-EPYC-Rome + 4*A100, right). All calculations run with YAMBO v5.1(dev) using 8 threads per MPI task.

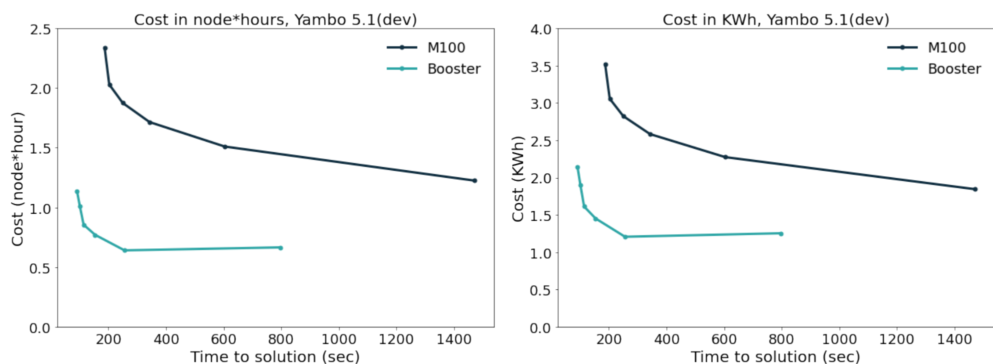

Figure 17: GrCo benchmark: node and energy costs of YAMBO v5.1(dev) runs performed on Marconi-100 at CINECA (IBM-P9 + 4*V100, left) and Juwels-Booster at JSC (AMD-EPYC-Rome + 4*A100, right).

add to reference cases for the YAMBO code. This graphene/Co interface (GrCo) is modelled as a graphene sheet adsorbed on a Co slab with 4 Co layers (2 Carbon atoms on top of a 4 layer Co slab terminated by an H atom). The simulations include a vacuum layer as large as the thickness of the slab, as customary for GW calculations on slabs. The structural optimisation was performed at the PBE-DFT level, followed by the band structure calculations including a large number of empty states, to be used as a starting point for the GW calculations. Figures 16 and Tab. 2 report the timing of the GrCo benchmark run on both the M100 and Booster clusters. The comparison of the wall-time shows and average speedup of 2.11 when using the newer A100 GPUs. This result is in agreement with data obtained by other software in the same scientific field.[3]

[3] https://developer.nvidia.com/blog/nvidia-ampere-architecture-in-depth/

MPI	Nodes	Wall time	Dipoles	X_0	X	Self energy	Other
				Marconi 100			
12	3	1470.0	45.5295	1384.0000	1.5958	14.2920	24.5827
36	9	604.0	26.5707	547.8331	1.7455	7.9777	19.8730
72	18	343.0	12.8779	311.0127	1.7116	5.4037	11.9941
108	27	250.0	12.8607	216.0491	1.7434	3.4225	15.9243
144	36	203.0	11.8446	171.8870	1.7350	2.5992	14.9342
180	45	187.0	13.6641	150.2193	1.7749	2.4651	18.8766
				Juwels Booster			
12	3	798.0	35.8163	737.0000	2.3405	9.0889	13.7543
36	9	256.0	17.9305	220.4115	2.3435	4.6583	10.6562
72	18	154.0	14.3519	123.9023	1.5550	3.3195	10.8713
108	27	114.0	12.4867	85.5235	2.1499	1.8856	11.9543
144	36	101.0	14.2410	68.8225	2.2279	1.7019	14.0067
180	45	91.0	15.1135	59.3384	1.6792	1.5501	13.3188

Table 2: YAMBO parallel scaling taken from runs of the GrCo benchmark. Data from Marconi-100 at CINECA (IBM-P9 + 4*V100) and Juwels-Booster at JSC (AMD-EPYC-Rome + 4*A100). All calculations run with YAMBO v5.1(dev) using 8 threads per MPI task.

3.5 CP2K

The performance portability of CP2K comes solely from the libraries on top of which CP2K is built. One of the key and performance critical libraries for CP2K is DBCSR. DBCSR performs distributed sparse matrix-matrix multiplications and is used in linear-scaling O(N) method. The work on DBCSR was finished at M18 of the project and reported in the previous deliverable. Here the short overview is summarised.

DBCSR performs MPI data exchange and small local matrix-matrix multiplications. The sizes of the matrices are really small – each of the [m,n,k] indices is in the range $\sim 1 - 100$ as defined by the atomic Gaussian basis. To optimise such small matrix-matrix multiplications DBCSR had to perform an exhaustive search of the optimal kernel parameters for each of the [m,n,k] combinations. This was inconvenient for two reasons: 1) for each flavour of the basis user had to generate the optimal GPU kernels for DBCSR before running CP2K 2) this couldn't have been done during the run-time and the process was slow because of the large number of combinations of internal GPU kernel parameters. With the introduction of the new ML-based framework developed during the MaX project, the optimal kernel parameters are found automatically for each new combination of [m,n,k], then the GPU kernels for NVIDIA or AMD are generated and compiled during the run time. This is called just-in-time compilation (JIT) and is available for both NVIDA and AMD GPU cards. The full documentation for users and developers is available at https://cp2k.github.io/dbcsr/develop/ and the predictive modelling framework is described at https://cp2k.github.io/dbcsr/develop/page/3-developer-guide/3-programming/2-accelerator-backend/2-libsmm_acc/4-predict.html.

3.6 SIRIUS

The SIRIUS library was developed with the CUDA backend from day zero. When AMD released the ROCm development toolkit, it was straightforward to generalise the SIRIUS code to work with both the CUDA and ROCm frameworks. The following minimal changes were introduced:

- Wrap API prefix, stream and error handlers

```
#if defined(SIRIUS_CUDA) || defined(SIRIUS_ROCM)
#define CALL_DEVICE_API(func__, args__)              \
{                                                     \
    if (acc::num_devices()) {                         \
        acc_error_t error;                            \
        error = GPU_PREFIX(func__) args__;            \
        if (error != GPU_PREFIX(Success)) {           \
            char nm[1024];                            \
            gethostname(nm, 1024);                    \
            std::printf("hostname: %s\n", nm);        \
            std::printf("Error in %s at line %i of file %s: %s\n",\
              #func__, __LINE__, __FILE__,            \
              GPU_PREFIX(GetErrorString)(error));\
            stack_backtrace();                        \
        }                                             \
    }                                                 \
}
#else
#define CALL_DEVICE_API(func__, args__)
#endif

#if defined(SIRIUS_CUDA)
#define GPU_PREFIX(x) cuda##x
#elif defined(SIRIUS_ROCM)
#define GPU_PREFIX(x) hip##x
#endif

#if defined(SIRIUS_CUDA)
using acc_stream_t = cudaStream_t;
#elif defined(SIRIUS_ROCM)
using acc_stream_t = hipStream_t;
#else
using acc_stream_t = void*;
#endif

#if defined(SIRIUS_CUDA)
using acc_error_t = cudaError_t;
#elif defined(SIRIUS_ROCM)
using acc_error_t = hipError_t;
#else
using acc_error_t = void;
#endif
```

- Wrap complex numbers arithmetic

```
#if defined(SIRIUS_CUDA)
using acc_complex_float_t = cuFloatComplex;
using acc_complex_double_t = cuDoubleComplex;
#define make_accDoubleComplex make_cuDoubleComplex
#define make_accFloatComplex make_cuFloatComplex
#define accCadd cuCadd
#define accCsub cuCsub
...
#define ACC_DYNAMIC_SHARED(type, var) extern __shared__ type var[];
```

```
#elif defined(SIRIUS_ROCM)
using acc_complex_float_t = hipFloatComplex;
using acc_complex_double_t = hipDoubleComplex;
#define make_accDoubleComplex make_hipDoubleComplex
#define make_accFloatComplex make_hipFloatComplex
#define accCadd hipCadd
#define accCsub hipCsub
...
#define ACC_DYNAMIC_SHARED(type, var) HIP_DYNAMIC_SHARED(type, var)
#endif
```

- Wrap API calls

```
/// Get current device ID.
inline int get_device_id()
{
    int id{0};
    CALL_DEVICE_API(GetDevice, (&id));
    return id;
}
```

- Wrap kernel calls

```
/*
 * CUDA runtime calls and definitions
 */
#ifdef SIRIUS_CUDA
#define accLaunchKernel(kernelName, nb, nt, memperblock, streamId, ...) \
do {                                                                    \
    kernelName<<<nb, nt, memperblock, streamId>>>(__VA_ARGS__);         \
} while (0)

#define hipThreadIdx_x threadIdx.x
#define hipThreadIdx_y threadIdx.y
#define hipThreadIdx_z threadIdx.z

#define hipBlockIdx_x blockIdx.x
#define hipBlockIdx_y blockIdx.y
#define hipBlockIdx_z blockIdx.z

#define hipBlockDim_x blockDim.x
#define hipBlockDim_y blockDim.y
#define hipBlockDim_z blockDim.z

#define hipGridDim_x gridDim.x
#define hipGridDim_y gridDim.y
#define hipGridDim_z gridDim.z
#endif

/*
 * ROCM runtime calls and definitions
 */
#ifdef SIRIUS_ROCM
#define accLaunchKernel(...)                  \
    do {                                      \
        hipLaunchKernelGGL(__VA_ARGS__);      \
    } while (0)

#endif
```

- Write GPU kernels in a generic way

```
template <>
__global__ void
add_pw_ekin_gpu_kernel<double>(int num_gvec__, double alpha__,
```

```
                                double const* pw_ekin__,
                                acc_complex_double_t const* phi__,
                                acc_complex_double_t const* vphi__,
                                acc_complex_double_t* hphi__)
{
    int ig = blockIdx.x * blockDim.x + threadIdx.x;
    if (ig < num_gvec__) {
        acc_complex_double_t z1 =
          accCadd(vphi__[ig],
            make_accDoubleComplex(alpha__ * pw_ekin__[ig] * phi__[ig].x,
                                  alpha__ * pw_ekin__[ig] * phi__[ig].y));
        hphi__[ig] = accCadd(hphi__[ig], z1);
    }
}

extern "C" void
add_pw_ekin_gpu_double(int num_gvec__, double alpha__,
                       double const* pw_ekin__,
                       acc_complex_double_t const* phi__,
                       acc_complex_double_t const* vphi__,
                       acc_complex_double_t* hphi__)
{
    dim3 grid_t(64);
    dim3 grid_b(num_blocks(num_gvec__, grid_t.x));

    accLaunchKernel((add_pw_ekin_gpu_kernel<double>), dim3(grid_b),
                    dim3(grid_t), 0, 0, num_gvec__, alpha__, pw_ekin__,
                    phi__, vphi__, hphi__);
}
```

- Wrap linear algebra calls

The same type of modifications were also introduced in the SpFFT and SPLA libraries and the MAGMA[4] team ported MAGMA to ROCm using a similar strategy. With this software stack it is now possible to run SIRIUS on both NVIDIA and AMD hardware. The result of the first benchmark of QUANTUM ESPRESSO on AMD GPU cards is present on the Fig. 18.

[4]MAGMA is a library that provides LAPACK functionality on GPUs

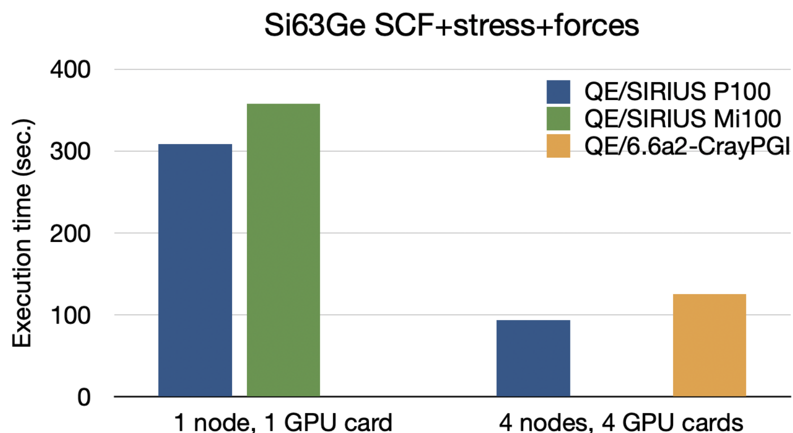

Figure 18: Benchmark of the GPU-accelerated QUANTUM ESPRESSO using NVIDIA P100 and AMD Mi100 cards. Benchmark runs the SCF ground state of $Si_{63}Ge$ and computes stress tensor and atomic forces. Blue: QE/SIRIUS runs on 1 and 4 nodes of Piz Daint. Green: QE/SIRIUS run on 1 node of AMD EPYC 7662 and Mi100 GPU card (single Mi100 node was the only resource available to us). Orange: cuda-Fortran enabled QE-6.6a2 compiled with PGI. The "out of GPU memory" error did not allow to run it on a single node. Please note that AMD Mi200 cards are not yet available for the public access and benchmarking.

3.7 BigDFT

The WP2 activities associated to performance portability related to BigDFT are in tight connection with the actions performed in the framework of WP4, in particular with respect to the use of advanced programming models. The main reason for this overlap is related to the architecture of the `libconv` library (designed in the context of WP1).

The auto-tuned `libconv` convolution library has been proven to be a useful and powerful addition to BigDFT. Its integration is challenging, as a lot of convolutions calls are already scattered in BigDFT, and the new convolution APIs introduced with `libconv` require changes in several places at the computation-kernel level. Memory requirements for vectorization are also important, as properly aligned memory must be used. This can be done within the futile1 library, but must be handled with care.

In order to minimise the impact on existing code, and to provide `libconv` as an option instead of a complete replacement of existing convolutions, a new interface library is being developed. `Liborb` (library for orbital manipulations) aims at handling in a single place all the orbital-related tasks. The aim of this library is to provide an API for the flexible definition of self-consistent field (SCF) operations at a high level. The goal is to separate the implementation of the orbitals representation from the SCF algorithms using it, so that the user is kept isolated from the basis sets employed for the discretisation. `Liborb` handles the specification of the orbital, which can be discretised in wavelet bases, Gaussian bases, or real space.

The centralisation of these representations simplifies the wrapping of all calls to `libconv`, which can be used mostly for the wavelet basis. `Liborb` indeed handles the scalar products between sets of orbitals, the communication of these orbitals between processes, the application of Hamiltonian operators on orbitals, and I/O for these bases. Conversion between all basis representations is also possible within the library, and is

OpenMP Threads	1	48
Ref (Fortran, no OpenMP) + GCC autovec	52961	52961
libconv – no vectorization + GCC autovec	17525	549
libconv – NEON vectorization + GCC autovec	15930	511
libconv – SVE vectorization + GCC no autovec	13604	455
libconv – SVE vectorization + GCC autovec	13689	456

Figure 19: Preliminary timings for 300 full applications of a « MagicFilter » convolution, double precision with sizes 124*132*130 on an A64FX processor. Compiler used is GCC10. Ref is just for results checkings and not the current BigDFT implementation. We can see that the generation of SVE instructions directly from the libconv library using BOAST outperforms the auto vectorized versions of the same convolutions and the NEON version, while the compiler cannot further optimize the resulting convolutions through vectorization. All tested libconv-generated convolutions are using unrolling/dimensions reordering and other optimizations automatically and generated in C, with wrappers created for C and Fortran interfacing. OpenMP scaling over 48 cores is 30 for each variant, showing that the scaling should not be affected by the extended use of SVE instructions. Flags for neon auto vectorization were -O3 -ftree-vectorize -march=armv8-a+sve -msve-vector-bits=512.

useful in most of the current developments of BigDFT.

The wrapping of all `libconv` operations in a single point will make the performance evaluation simpler, in turn allowing for an easy selection of the most efficient flavour of convolutions. `Libconv` itself has also been improved, with the support of SVE instructions within the BOAST framework, to generate automatically SVE-vectorised convolutions, making the best possible use of modern ARM processors, such as the A64FX used in the Fugaku supercomputer.

In the last year, extensive comparison of the performance of BigDFT on ARM architectures has been performed. We refer the reader to the description provided in the WP4 document D4.5 [3] about the various benchmarks performed.

4 Conclusions and ongoing work

Computationally relevant kernels of all MAX flagship codes have been ported to a hybrid (heterogeneous) computing paradigm utilising both CPUs and GPUs. Depending on the availability of hardware for testing and performance verification, as well as on the required software stack for the applications, different computing architectures are supported.

- Besides the dominating x86 CPU architecture, we have also explored ARM based CPUs.
- All codes work on the currently dominating GPUs from NVIDIA, and can utilize these GPUs efficiently. The performance achieved using this architecture is demonstrated in this and previous reports, and these developments generally are incorporated in the code versions publicly released and used for cutting edge science.
- AMD GPUs (and also INTEL GPUs) have also been addressed, with results at different levels of maturity. While in some cases readiness has already been achieved, in other cases the work is still ongoing. This is mostly a direct consequence of the limitations in the available compiler support and hardware accessibility (though notably we were able to run some of our software on pre-release INTEL hardware).

Summarising, the MAX flagship codes achieved and managed to demonstrate performance portability to a wide range of state-of-the-art HPC computational architectures, notably including those relevant for the EuroHPC program. Overall, such progress in performance and scalability paves the way for a successful application of our codes on the upcoming (pre-) exascale machines.

Acronyms

CPU Central Processing Unit. 9–11

DFPT Density Functional Perturbation Theory. 13

GPGPU General Purpose GPU. 9, 12

References

[1] Wortmann, D. *et al.* Second release of max software: Report on performance achieved.Deliverable D2.2 of the H2020 CoE MaX (final version as of 30/11/2020). EC grant agreement no: 824143, JUELICH, Germany. (2020).

[2] Affinito, F. *et al.* Second report on codeprofiling and bottleneck identification. deliverable d4.3 of the h2020 project max(final version as of 31/05/2020). ec grant agreement no:824143, cineca, casalecchio di reno (bo), italy. (2020).

[3] Affinito, F. *et al.* Final report on code profiling and bottleneck identification. deliverable d4.5 of the h2020 project max (final version as of 30/11/2021). ec grant agreement no: 824143, cineca, casalecchio di reno (bo), italy. (2021).

[4] Cesarini, D. *et al.* Final report on co-design activities.

[5] Genovese, L. *et al.* Third release of MAX software: report on the evolution actions taken in each of the codes. Deliverable D3.4 of the H2020 CoEMAX (2022).

[6] Cavazzoni, C. *et al.* First report on code profiling and bottleneck identification, structured plan of forward activities. Deliverable D4.2 of the H2020 CoE MaX (final version as of 30/06/2019). EC grant agreement no: 824143, CINECA, Bologna, Italy (2019).