D2.2

Second release of MaX software: Report on performance achieved

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	30/11/2020 (**month 24**)
Actual submission date	30/11/2020
Final version	30/11/2020
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) / 30/11/2021 (month 36)
Website	http://www.max-centre.eu
Deliverable no.	D2.2
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. (2020): 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.

Disclaimer

Contents

1 Executive Summary 4

2 Introduction 5

3 Performance assessment activities 5

4 Performance and portability of MaX flagship codes, libraries and components 6
- 4.1 FLEUR 6
- 4.2 Quantum ESPRESSO 9
- 4.3 Siesta 10
- 4.4 YAMBO 13
- 4.5 CP2K 20
- 4.6 SIRIUS 20
- 4.7 BigDFT 20

5 Conclusions and ongoing work 23

References 24

1 Executive Summary

This report on the second release of the MAX software provides detailed information on the activities and resulting advancements in performance optimisation on various computing platforms. A special section is devoted to our effort to introduce a more homogeneous performance metric useful to monitor the progress of our work. We outline the general challenges we face and our strategy for the presentation of the performance of complex codes.

The work on the codes is presented afterwards taking some of these ideas already into account. In comparison to the last report, the adaptation of our codes to new, heterogeneous in particular, HPC systems like e.g. Marconi-100 has been a main target of our work. This should be considered as an important step towards similar activities ready to start on the upcoming European pre-exascale machines as soon as they become available. While each code deals with slightly different challenges, the common goal to increase the performance on these different computing architectures is a unifying theme throughout this report.

Much of our activity can be seen as providing and implementing the necessary performance to the modules and libraries of WP1 and the new functionality of WP3. Consequently, the release of the codes which this report partly documents is a joint effort with these WPs.

2 Introduction

All MAX flagship codes aim at providing excellent performance on a large set of computing platforms to enable our users to perform cutting-edge simulations in their scientific fields. Hence, the effort reported in this WP focuses on providing this performance in a portable fashion.

While we reported significant advances of the porting of codes to several computing architectures already in D2.1, we now put a stronger emphasis on the performance achieved on these architectures and aim at tuning the performance of our codes. This includes the continuation of work on the different implementations for GPUs by both expanding their coverage as well as modifying and improving the programming. Different programming frameworks are still being investigated and used to tackle the challenge of creating high performance portable codes.

In the following we will present the activities in a per-code fashion most appropriate to document the achievements included in the corresponding public releases.

3 Performance assessment activities

In order to monitor our progress, to document achievements, and to identify the remaining challenges and bottlenecks, we try to introduce a common metric and common presentation of the results applicable to all codes and all computing platforms. Due to the inherent differences of the machines as well as the complexity of the codes and possible use cases, this effort must still be considered very preliminary and we expect to improve it in the future. The differences in the underlying algorithms, the huge zoo of possible physical properties to be calculated, with correspondingly many input parameters and possible different usage scenarios, make it impossible to define a transferable set of reference calculations, usable to compare different codes on different machines. We have opted to explore the following idea: on the one hand, the time-to-solution is the key quantity we have to optimise, since this is ultimately determining the scientific progress enabled by our codes. We thus use the time-to-solution as the basic quantity to characterise our performance tests. On the other hand, this quantity has to be evaluated together with a measure characterising the amount of used computational resources. Hence, we decided to investigate the relation between the resources used and the time-to-solution, and we propose this as the key for determining the progress made in our efforts to boost the performance and applicability of our codes on various architectures.

Different possible measures for the computational effort or the computational resources used can be adopted within this approach providing different views. The most basic approach, but for some aspects most relevant for actual users, is to measure the computational effort of a run the way the users are "charged" for computing resources on PRACE machines. While we will use this idea due to its relevance in actual HPC projects and simple application, we also show its limitations (see the Siesta-related section of this report, Sec. 4.3). Other, more fundamental, hardware centric data could be e.g. the theoretical peak flops the hardware would be able to perform during the run or the energy used.

However, this more fundamental metric of the energy consumption of a given calculation also has some drawbacks. While this seems to be a physics-based, absolute and transportable measure of performance, there are at least three relevant issues to note:

- It is not very easy to instrument and run the codes to obtain the appropriate measurements. We are in contact with hardware experts to try to obtain at least some approximate measure of the energy consumption.
- The energy consumption has to do not only with intrinsic code optimisation but with the characteristics of the architecture. This is another reason why a multi-dimensional presentation of performance, that includes also the time-to-solution, is important.
- From the economics point of view, energy is just one element of the marginal cost of operation, and there are other important fixed costs. We can then expect that, from the user point of view, the choice of code, platform, and algorithm will be guided (apart from time-to-solution) by the resource cost measured with respect to and in the units of its computer centre allocation.

Hence, while the energy used for a simulation can provide a further metric to consider, it also does not provide a picture of uniform applicability and will have to be augmented with other quantities. We will continue to explore such possible ways to quantify the performance of our codes and foresee that this will remain a challenging subject to consider also in future work.

4 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.

4.1 FLEUR

Work on the FLEUR code focused on two independent lines of development. On the one hand we used our large scale demonstrator simulations as performed in the context of WP6 to identify scaling issues and bottlenecks for such large setups. On the other hand we continued to port FLEUR for GPU accelerated machines.

FLEUR performance on large HPC clusters

Using our experiences from the simulation of large setups we identified some further bottlenecks of the code which limit scalability and applicability on architectures with many nodes. In particular, the redistribution of the Hamiltonian and overlap matrix after the setup proved to be problematic. Due to algorithmic constraints we choose to generate these matrices in a cyclic distribution in one matrix dimension and hence have to redistribute it to a block cyclic distribution used by the diagonalization routines. While the SCALAPACK library provides corresponding redistribution routines, we found that

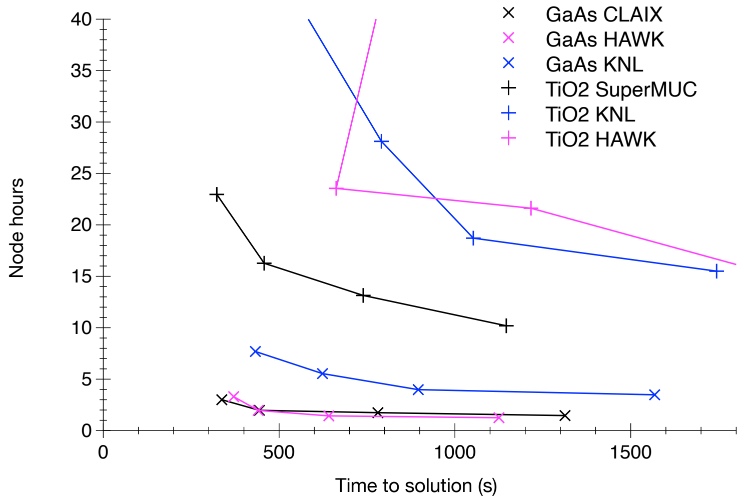


Figure 1: Computational resources needed for two example systems on different computing architectures. While the machines used are rather similar (CLAIX,Super-MUC: Intel Xeon, HAWK:AMD, JURECA-Booster:Intel-KNL) we can observe significant differences in performance and scalability.

some of these turned out to be very ineffective for larger systems and we therefore replaced them with our own algorithm taking the explicit forms of the matrices into account. This removed the specific bottleneck and allowed for significant better scaling of the code. Further modifications and improvements included extensive modifications of the treatment of non-collinear magnetism. Here, the calculations performed showed that some of the setup routines did not include the same level of optimisation on some of the machines and had to be adjusted.

Finally, we observed significant differences on the machines we investigated (Fig. 1). In particular, the JURECA booster in JSC with its KNL processors did not provide the same level of performance and scalability compared to Intel-Xeon and AMD based systems. Some work relating to the optimal choice of computational setups like thread placement and different math-libraries lead to an improvement of the performance on these architectures. We also observe rather irregular run-times of the code on the HAWK supercomputer (see Fig.1). This could be tracked down to the matrix diagonalization and needs to be investigated further and fixed.

To start investigating the energy footprint of the calculations we use the typical average power consumption data available for some of the machines we use. As we do not have actual power usage for the runs available, the resulting Fig. 2 displays only a different scaling of the performance data presented in Fig. 1) but allows to compare the different machines in terms of their computational efficiency. Again we can observe that the data for the KNL architecture is significantly worse compared with Intel-Xeon and AMD data.

GPU-aware implementation

The deployment of FLEUR on GPU bases HPC systems is still limited by the difficulties to obtain reliably multi-GPU solvers for the generalized Hermitian matrix diagonaliza-

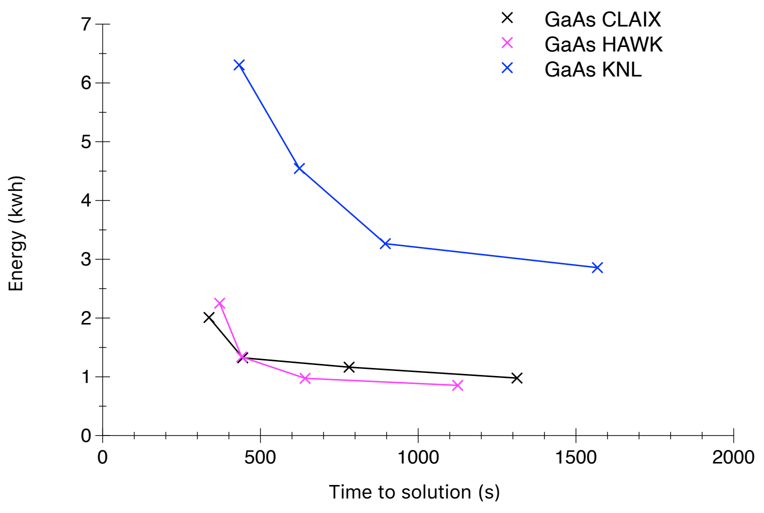


Figure 2: Estimate of electrical footprint of FLEUR calculations. These values have been obtained using the typical average power consumption of the machines and the runtime. No dedicated measurement was performed.

tion. As now such libraries become available (e.g. ELPA), we intensified our effort to improve the GPU-aware implementation of the remaining computational relevant routines of FLEUR. In particular, we focused on the following:

- **Replacement of remaining CUDA-Fortran implementations with corresponding OpenACC code.**
  While CUDA-Fortran sometimes provides more flexibility and fine-grained access to the GPU, the lack of portability of this programming model motivated us to abandon this approach. Hence, we by now replaced all corresponding code by OpenACC without observing any relevant performance degradation.

- **Extension of the GPU code to further subroutines.**
  While we so far have mainly concentrated on the Hamiltonian setup as the most relevant computational intensive part (neglecting the diagonalization) we now also started to implement GPU accelerated versions of the code for the charge-density generation.

- **Minimisation of Data transfer.**
  By including more computations on the GPU even for code sections not in itself relevant for the performance of the calculation we managed to reduce the required data-movement between the GPU and the CPU.

- **MPI parallelism in the GPU version.**
  We made the GPU version compatible with the MPI parallelization of the code. We aim at a usage model in which a single MPI task per GPU is used in order to maximise the available memory and to balance the performance of the GPU/CPU implementations.

4.2 QUANTUM ESPRESSO

Currently, the performance portability effort in QUANTUM ESPRESSO is mostly concentrated on the high-level layers of the code (see deliverable D1.4 for more details on the code structure) while lower-level layers still retain some specific code parts, adopting though architecture agnostic APIs.

The current GPU version of the code may be compiled for both hybrid MPI/openMP and heterogeneous MPI + CUDA/GPU accelerators. It maintains on hybrid systems the same performance of the main version. Work is ongoing towards unification of the code base of the GPU version with the official trunk. Effort on this side are mostly dedicated to automatise and hide the allocation and synchronisation of data between the host and the device. This is needed because the high-level code parts are meant to be used and developed by electronic-structure specialists who may not know the details of architecture-specific data organisation and data movement.

The data management simplification is at the moment pursued by the usage of the `devXlib` MAX library, also adopted by YAMBO. The library has just been added to the official QUANTUM ESPRESSO code release in order to streamline such development and also to allow us to collect more user experience. An alternative route under consideration for the code unification, is the usage at all code levels of a portable programming paradigm such as OpenACC or OpenMP5, or libraries such as `kokkos` and alike. This choice has in our view the inconvenience that it would require a specific knowledge of such paradigms by the whole developers community, making the code less accessible. A more viable choice on our side is to use these paradigms only for the development of low-level libraries and modules which can be kept transparent to developers of high-level code.

One important challenge for performance portability has been to organise the computation and the data distribution so as to have a good performance on different kind of nodes. We have mainly distinguished two types:

- *Homogeneous regular nodes*: the computational capacity is totally provided by a certain number of cores and we are able to efficiently run a large number of MPI task using one or more threads per task. We use MPI for distributing the data and the computational load. The performance may scale up to a very large number of MPI tasks.

- *Heterogeneous fat/GPU nodes*: the computational capacity is distributed between the regular cores and a few GPGPU devices per node. The latter usually provide most of the computational power of the node. The choice in this case has been to offload all compute-intensive parts to the GPU. Each MPI task typically uses one accelerator, but depending on the size of the calculation more tasks may offload to the same accelerator. MPI distribution is, in this case, used mainly to reduce the GPU allocated memory per task within the device capabilities. As for the computational load, apart for **k**-point pool parallelism [1, 2], it is usually convenient to limit the number of used MPI tasks to those required by memory load distribution. Although this has the side advantage of inducing the user to minimise the computation cost in term of node-hours (see fig. 3) we have been improving the scalability mainly working on the band parallelism. In the framework of WP3,

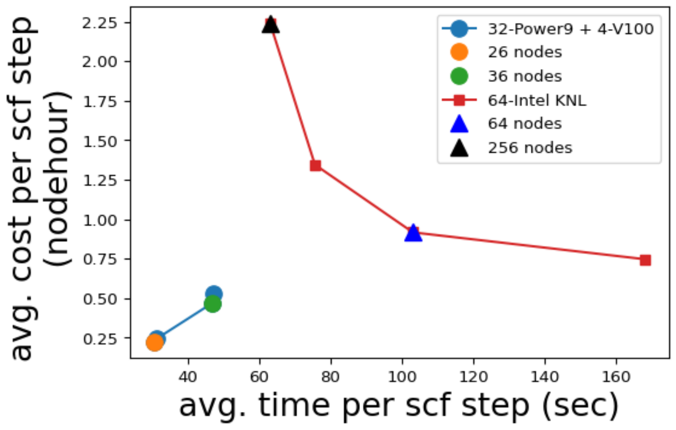


Figure 3: Computational cost for CNTPOR benchmark case (see https://gitlab.com/max-centre/benchmarks/-/tree/master/Quantum_Espresso/PW/CNT10POR8) in different platforms (see [1, 2] for more details). The different behavior for the two curves reflects the different choice in distributing the calculation.

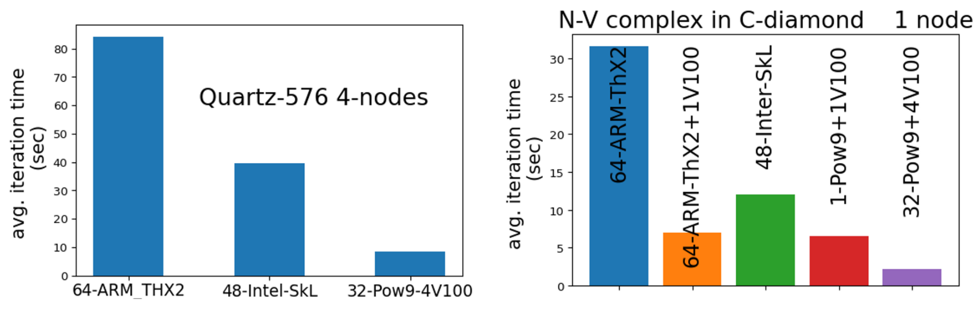


Figure 4: Comparison of averaged computational times for small calculations on few nodes of different platforms. The calculation on the left is a quartz supercell of 576 atoms. The calculation on the right is is Vacancy + N impurity complex in a 216 cell of C-diamond. ARM Thunder-X2 has been used with 16 tasks and 8 thread per task. The SkL node with 12 task and 4 threads, the Power9 with four task, 8 threads per task and one NVIDIA Volta card per task. The Power9-V100 architecture has also been tested in serial mode, its node cost should be 25% of the whole node.

for instance, we have implemented more parallelizable algorithms [3]. Those have been introduced in the last release and we expect a sensibly improved scaling.

The current approach is able to achieve good performance in terms of node-hours for both type of nodes (see Figs. 3,4) with an implementation that is completely transparent for the high-level developers.

4.3 Siesta

The main performance breakthrough in SIESTA since the previous software release has been the addition of GPU support for the electronic-structure solver based on diagonalization. This has been achieved through the use of the GPU-enabled version of the ELPA library, available in stand-alone form or through the ELSI interface layer implemented in SIESTA as part of the WP1 activities.

We will see below that benchmarks carried out on the new Marconi-100 system at CINECA show sizeable speedups when using GPUs. The cost of the non-solver part of the execution (the setup of the Hamiltonian) is typically very small compared to the solver part (less than 5% initially). Hence, speedups in the solver are basically speedups in the overall calculation.

This is a very significant development, and a milestone for SIESTA. However, it is important to take into account that diagonalization is not the only solver option available in the program. The use of strictly localised basis sets, implying the sparsity of the Hamiltonian and overlap matrices, enables the use of alternative, very efficient and scalable algorithms to compute the electronic-structure. Indeed, the SIESTA solver of choice for massively parallel calculations for large systems has traditionally been PEXSI (Pole Expansion and Selected Inversion), due to its favourable size scaling, multi-level parallelization scheme, and smaller memory needs. SIESTA was the first mainstream code to offer an interface to the PEXSI library, and now further enhancements are available through the ELSI interface layer recently implemented in SIESTA, including an extra level of parallelization over chemical-potential trial points, and the ability to expand the Fermi-Dirac function with much fewer poles. With the availability of GPU-enabled diagonalization in SIESTA, it is relevant to revisit the issue of the placement of the "break-even" point, also taking into account the fuller context of the user needs. Hence in this report, which is based on the benchmarking data presented in D4.3 [2], we offer also a comparison of the time-to-solution and resource usage for both methods.

We present execution data on two architectures: MareNostrum IV at BSC, and the new Marconi-100 computer at CINECA. MareNostrum IV is a cpu-only machine, with two 24-core Intel Xeon processors per node. For our benchmarks we used the 2017.4 Intel compilation and MPI environment, including the MKL library. Marconi-100 has nodes composed of two 16-core Power9 processors, and four Volta GPUs. In the benchmarks reported in this section, we have used the Spectrum MPI library, CUDA version 10.1, and IBM's optimised ESSL library, with the GNU 8.4 compilers. We do not take advantage of the 4 extra hyperthreads per core offered by the Power architecture.

We first offer some details about the GPU acceleration achieved in the diagonalization solver, using a first benchmark system based on a hydrogen-saturated Si quantum dot. The base system contains 1359 atoms, and for the purposes of the benchmark we replicate it 8 times. With a minimal basis, this results in a problem with around 35000 orbitals. It is interesting to decompose the overall timings to see the contribution of the individual phases of the computation (see Fig. 5). 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 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). We note that the Cholesky and DM-building steps have not yet been enabled for GPU acceleration, but those steps that have been ported show very significant speedups.

GPU-accelerated diagonalization vs PEXSI

To benchmark the scalability and general performance properties of the diagonalization and PEXSI solvers, we have chosen a larger system: a piece of a protein of the Sars-Cov2

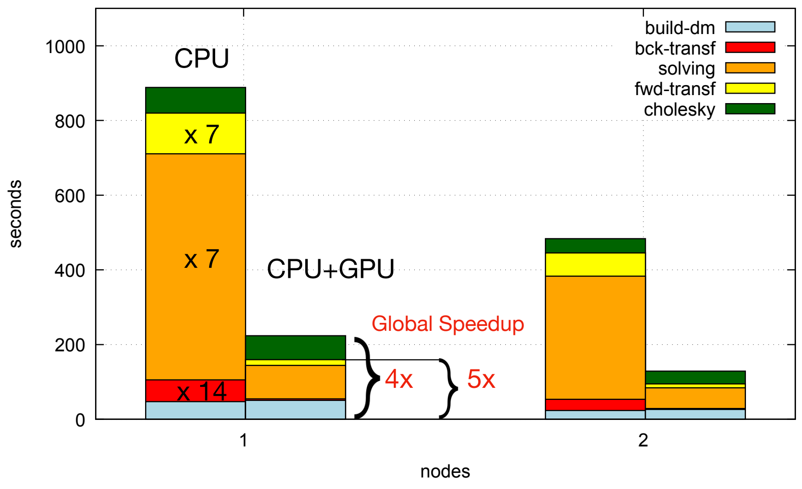


Figure 5: Speedup of the diagonalization comparing the CPU only version with the CPU+GPU implementation. The contributions of the different steps are shown individually highlighting the different level of GPU porting.

virus in water, with around 9000 atoms and 58000 basis orbitals. In Fig. 6 we show the time-to-solution versus node use in Marconi 100. While exhibiting a strong improvement over the CPU version, the scalability of the GPU-accelerated solver is rather degraded already at 16 nodes, so we cannot expect much further decrease in the time-to-solution for this problem size.

In contrast, the PEXSI solver offers several levels of parallelization: over orbitals, over poles, and over chemical potential interpolation points (just two in this case). The first level is represented by the number of MPI tasks per pole (`tpp`), which ranges from 8 to 128 in the figure. For `tpp=8` (the first points to the left in the PEXSI curves) we can achieve full parallelization with 10 nodes for the case of 20 poles, and with 15 for the case of 30 poles. In Fig. 6 we see that this scaling reserve of the PEXSI method means that, by simply increasing `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. A good way to look at this balance of objectives is provided by Fig. 7. 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, but note that there is a sharp drop in efficiency when going from `tpp=32` to `tpp=64`. This obviously reflects the fact that the intra-pole parallelization now needs to perform communications with other nodes, with higher latency. In this benchmark we did not go beyond `tpp=128`, but it would be interesting to try larger systems and see if they can

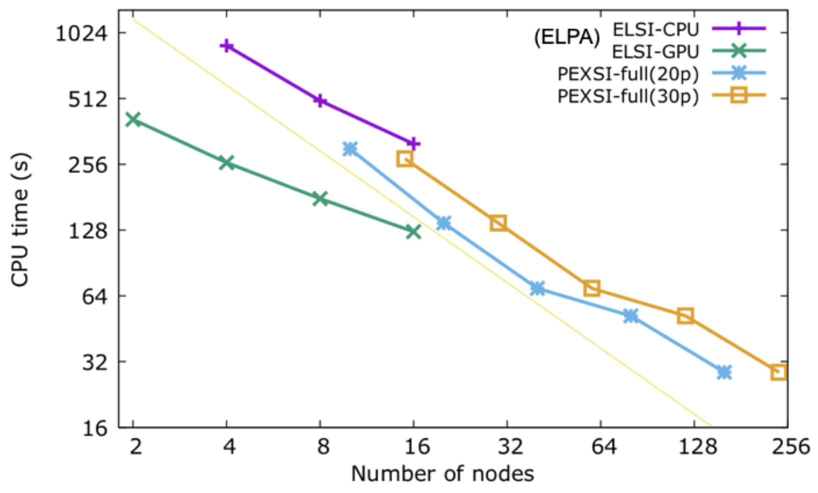


Figure 6: Time to solve the diagonalization problem corresponding to a piece of virus protein surrounded by water molecules, with approximately 58000 orbitals. Two sets of PEXSI results (for 20 and 30 poles) are shown, each for different numbers of tasks per pole (from left to right: 8, 16, 32, 64, 128). The thin line shows the ideal scalability behaviour.

maintain a good scaling at those levels (the original benchmarks of the selected-inversion algorithm show that they should).)

The problem of choosing a metric for resource costs

The node*hour unit used in the vertical axis in the two-dimensional performance representation in the last section is a stand-in for the general concept of "resource cost", and is indeed widely used by supercomputer centres as a unit for allocations and user charges. However, while useful to check the evolution of the performance of a given code in a given platform, this "unit" is not portable across machines. To illustrate this, we have extended the benchmark data with calculations on MareNostrum IV with the PEXSI solver (see Fig. 8). It would seem that calculations on MareNostrum IV are much cheaper than in Marconi-100, but obviously this does not take into account the relative cost of a Marconi-100 node vs a MareNostrum IV node. In fact, it can be argued that we cannot meaningfully compare the true resource costs within Marconi-100 itself, since in the PEXSI case we are not actually using the GPUs.

A more fundamental metric could be the energy consumption of a given calculation, but, as discussed in Sec. 3, its implementation is not trivial and its full implications not completely clear.

4.4 YAMBO

As of M12 (Nov 2019), a first CUDA-aware version of YAMBO was released. During the second year of MAX phase 2 (Dic 2019 – Nov 2020), the YAMBO development team has worked on the extension of the GPU-porting of the code. Significant effort has been devoted to **actions along the following lines**:

- Porting of new kernels of the code to GPUs using CUDA-Fortran;

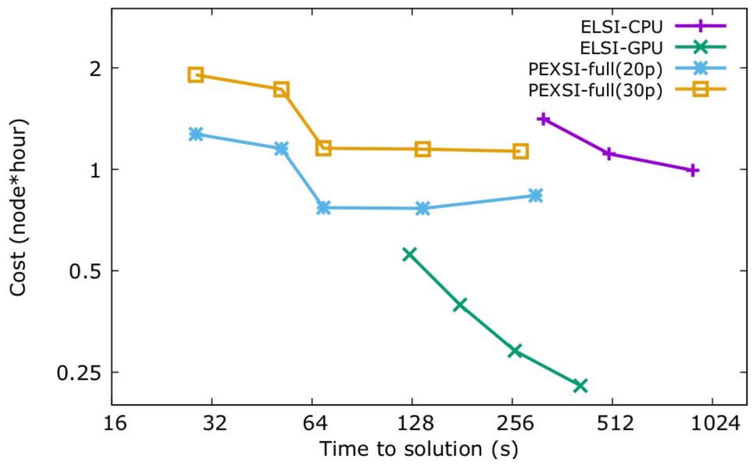


Figure 7: Total cost (per scf step) vs time-to-solution for the virus protein problem, with approximately 58000 orbitals. Details as in previous figure.

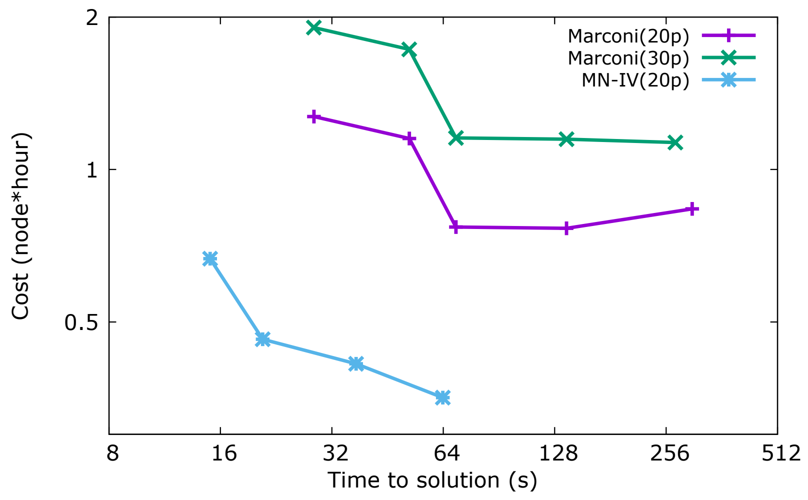


Figure 8: Total cost (per scf step) vs time-to-solution for the virus protein problem using the PEXSI solver on Marconi100 (same details as above) and MareNostrum IV (in this case the tpp values are 24, 48, 96, and 192).

nodes	dipoles	χ^0	χ	Σ_{xc}	other	wall time
YAMBO v4.5						
20	1623.0000	132.1358	7.9136	16.9497	16.0009	1796
40	827.0000	165.7774	6.3793	9.4550	16.3883	1025
YAMBO v5.0						
20	162.775	91.1397	8.3256	18.5777	34.1820	315
40	93.951	47.5844	7.1586	10.4086	42.8974	202

Table 1: Timing comparison of YAMBO v4.5 with v5.0. Data are computed on Marconi100 at CINECA, using GPU-support for the H-Rutile use case. Calculations for with 20 and 40 nodes, each equipped with 4 NVIDIA V100 GPUs, are shown.

- Identification of time- and memory-bottlenecks and related code refactoring;
- Cleanup of the code-base for maintainability and readability, especially in view of further developing the porting to GPU-accelerated machines;
- Evaluation and first proofs of concept of different programming models to add a second back-end for GPU-porting.

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

- Completion of the CUDA-Fortran GPU porting of the code;
- A number of bottlenecks solved (calculation of dipoles, most notably), making the code less memory hungry and faster on GPUs;
- All significant code duplication due to CUDA support removed; code is ready for a second implementation of GPU support (OpenACC and OpenMP5 are being considered).

Completion of the CUDA-Fortran porting

While already advanced at M12, the GPU-porting of YAMBO via usage of the CUDA-Fortran programming model has been the main target of the activities described in this document. On the one side we have **(A) ported extra computational kernels** that were not considered at first as well as identified bottlenecks and addressed them. On the other side we have also consolidated and stabilized the porting by **(B) adding tools dedicated to the management of GPU-accelerated runs** and **reorganizing the code base**. The most relevant tasks accomplished during the period are described in the following.

Refactoring of the DIPOLE kernel porting. The dipole kernel computes matrix elements of the form $\rho_{nm\mathbf{k}}(\mathbf{q},\mathbf{G}) = \langle\psi_{nk}|e^{i(\mathbf{q}+\mathbf{G}\mathbf{r})}|\psi_{m\mathbf{k}-\mathbf{q}}\rangle$ in the limit $\mathbf{q} \to 0$, which enters the calculation of linear response quantities and are basic ingredients of MBPT calculations. Based on detailed benchmarks, we have identified this kernel as a performance bottleneck when executed on GPUs in the presence of a large number of pseudopotential projectors. We have addressed the issue by completely refactoring one of the critical routines relevant for the calculation. The resulting speedup is quite important, as shown by the comparison of data for H-Rutile reported in Tab. 1 (data corresponding to YAMBO v4.5 were also shown in the deliverable D4.3).

Moreover, the dipole matrix elements can be computed in several ways, one of which is called "covariant" (the $\mathbf{q} \to 0$ limit mentioned above is performed by considering small but finite $\mathbf{q}$ vectors). This specific kernel, quite relevant to compute dipoles when e.g. non-local terms beyond the pseudopotentials are present in the Hamiltonian, have also been ported to GPUs following the same strategy adopted in other parts of the code by using CUDA-Fortran.

Device memory tracking. With the inclusion in YAMBO of the memory tracking utility, all `allocate/deallocate` instructions are intercepted by the precompiler and expanded in a few extra lines, able to track memory usage. Given the relevance of memory tracking on GPUs, we have further extended the approach to track separately (as it should) the memory allocated on the host and on the devices. Besides being very useful for the user (immediately notified when a memory issue shows up), this utility allowed also the developers to identify a number of memory leaks and peaks.

GPU memory optimisation. Thanks to the memory tracking, significant GPU-oriented memory optimisation has been put in place, by avoiding memory duplication (or unnecessary GPU allocation) or MPI-distributing large arrays. This action has touched, among other, the routines: `scatter_Gamp` (space integrated Coulomb interaction), `X_PARALLEL_alloc`, `X_redux` (calculation of the distributed reducible response function), `QP_ppa_cohsex` (calculation of GW/COHSEX self-energies). In particular, the distribution of the response matrix χ has been included in the calculation of QP corrections thereby avoiding the allocation of large scale non-distributed arrays on GPUs.

Run-time CPU/GPU switch of linear algebra (linear response). When solving the Dyson equation for the reducible response function χ, a matrix inversion needs to be performed and YAMBO implements parallel linear algebra (LA) on GPU for several years. When working on GPUs, the distributed parallel LA is not yet available (missing libraries). As a workaround, we have implemented the possibility to choose at runtime whether to invert the χ matrix serially on a single GPU or to perform the operation in parallel (ScaLapack style) on CPUs. This is quite useful when the size of the matrix to be inverted becomes large (>20-30000) and dominates the GPU memory usage. Support for parallel distributed dense linear algebra on GPUs is once more identified as a crucial step to further improvements.

GPU porting of G-terminator. The terminator technique has been implemented in the past years (see e.g. D1.4 of MAX phase 1) in YAMBO, and allows the user to accelerate convergence with respect to the sum-over-states when computing the polarizability or the self-energy. This technique is quite relevant in a number of cases, e.g. when the absolute value of quasi-particle (QP) corrections are required (at variance with QP energy differences), and turned out to be a bottleneck when used in the context of GPU-accelerated runs. Because of this, during the restructuring of the `QP_ppa_cohsex` routine we have also ported the usage of G-terminators to GPUs. Benefits were already evident in the calculation of IPs of molecules in the GW100 set [4], done in the context of WP6.

Device auto-assignment. When running on accelerated nodes with more than one GPU, the assignment of MPI tasks to GPU cards is not automatic (and unattended mapping may result in all tasks subscribing to a single GPU, with loss of performance). Based on the knowledge of intra-node MPI communicators, we have performed an even assignment of MPI tasks to GPUs, also including the case of GPU over-subscription (more than one MPI task assigned to a single card), and reporting this occurrence to the user in the report file.

Update of compilation with GPU-support. The autotool-based configuration of YAMBO has been updated in order to easily compile on GPU-accelerated machines, mostly using the NVIDIA PGI compiler. Specific updates were included to compile the code out-of-the-box based on the experience gained working on the Marconi100 machine as well as following up on users' reports from the YAMBO forum.

Other optimizations. Quite remarkably, one important issue related to parallel performance was identified earlier in the calculation of the response function. There, an overwhelming MPI communication triggered by the routine `MATRIX_transfer` caused the problem. This issue was addressed and reported in the deliverable D4.3.

Code cleanup and GPU-aware programming style

In order to consolidate and stabilized the GPU porting of M12, all merged into the main development branch with a single source code base, YAMBO had to undergo an important process of cleanup of the source files mostly aimed at **(1) readability** and **(2) maintainability**, with particular emphasis on the removal of CUDA-related code duplication. Both tasks have been achieved with large degree of success, by employing quite standard software engineering techniques.

In particular, we have made large use of (i) simple Fortran data structure to organize cpu and gpu variables, (ii) pre-compiler macros (`DEV_VAR`, `DEV_SUB`, `DEV_ATTR`) to hide differences across cpu and gpu code, e.g. appending `_d` or `_gpu` to variable and subroutine names. As an example:

```
#ifdef _CUDA
#  define DEV_SUB(x)      x##_gpu
#  define DEV_VAR(x)      x##_d
#  define DEV_ATTR        , device
#else
#  define DEV_SUBN(x)      x
#  define DEV_VAR(x)       x
#  define DEV_ATTR


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)
...
```

Importantly, we have also extensively used (*iii*) DevXlib wrappers (named `dev_*` in the following example), to mask memory copies and small computational kernels.

```
     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(DEV_VAR(iscp%rhotw),DEV_VAR(isc%rhotw))
       endif
       !
       DP_Sx_l=dev_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
```

Finally (*iv*), the concurrency of `omp parallel do` regions and CUDA-Fortran `cuf kernels` has been handled with the same techniques, leaving room to further add OpenACC or OpenMP5 directives to the implementation.

```
    !DEV_CUF kernel do(2)
    !DEV_OMP parallel do default(shared), private(ig1,ig2), collapse(2)
    !
    do ig2=X_cols1,X_cols2
    do ig1=X_rows1,X_rows2
      eet_factor(ig1,ig2)=isc_rhotw_p(DEV_VAR(G_m_G)(ig2,ig1))
    enddo
    enddo
```

Performance achieved

Concerning performance, we report here recent data obtained on the Marconi-100 (M100) and Marconi-A3 (M-SKL) machines at CINECA. M100 has IBM Power9 nodes equipped with 4 NVIDIA V100 cards,[1] while Marconi-A3 deploys dual socket Intel Skylake CPUS (24+24 cores/node and Omnipath network).[2] Overall, scaling, performance, node and energy cost data obtained using YAMBO v5.0 for the H-Rutile system are reported in Figs. 9 and 10. In the upper panels of Fig. 9 a comparison of M-100 (left) and M-SKL (right) is reported. The comparison done at fixed number of nodes gives a ratio in the wall-time larger than the nominal ratio of node computational powers, in favour of the GPU run. This can be explained on one side by the significantly larger number of MPI tasks to be used on M-SKL, which naturally has an impact on performance. On the other side, different levels of optimisation performed by different compilers (PGI and Intel here) on sensible routines (such as `QP_ppa_cohsex`) can also be invoked. Further investigation is needed. Cost plots related to the same data are reported in Fig. 10. Left panels shows the node*hour cost measure, where we have renormalized the M-SKL curve with a

[1]Marconi-100, https://www.top500.org/system/179845/
[2]Marconi-A3, https://top500.org/system/179186/

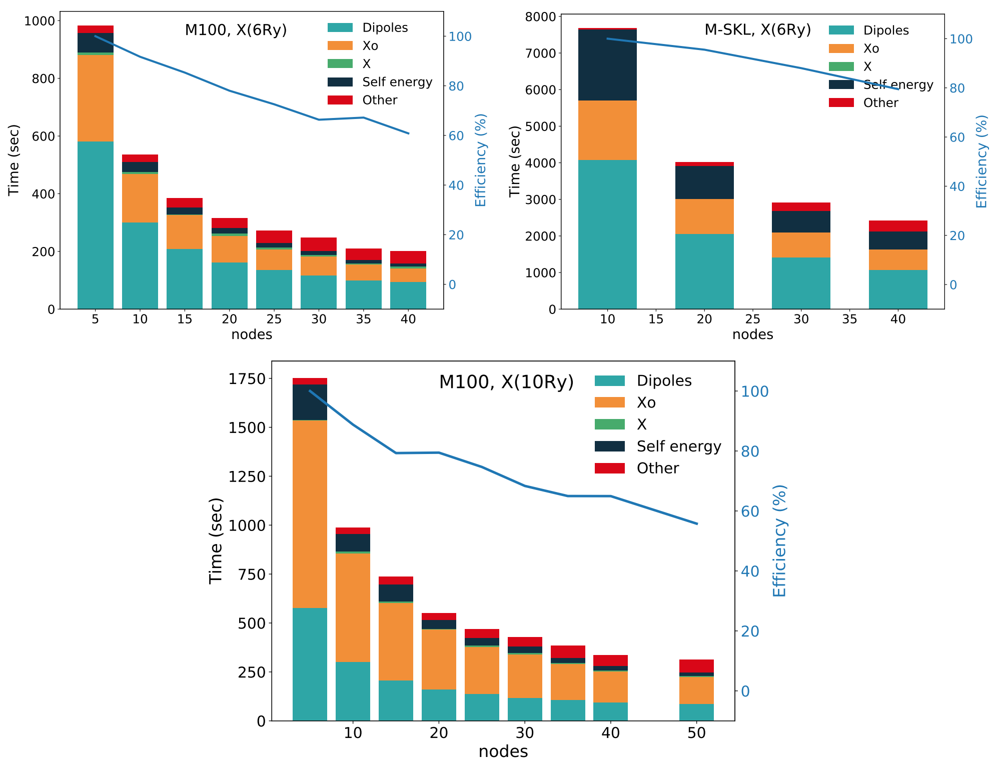


Figure 9: YAMBO parallel scaling and performance taken from runs of the H-Rutile use case of the MAX benchmark set (defined in D4.2 [1]). Upper panels: data from Marconi100 at CINECA (P9+4*V100, left) and Marconi-A3 (INTEL Skylake, right). Lower panel: Data from Marconi100 where the dimension of the response matrix X has been increased to 10 Ry (at variance with 6 Ry used in the upper panels), representing a value more realistic for production runs. All calculations run with YAMBO v5.0.

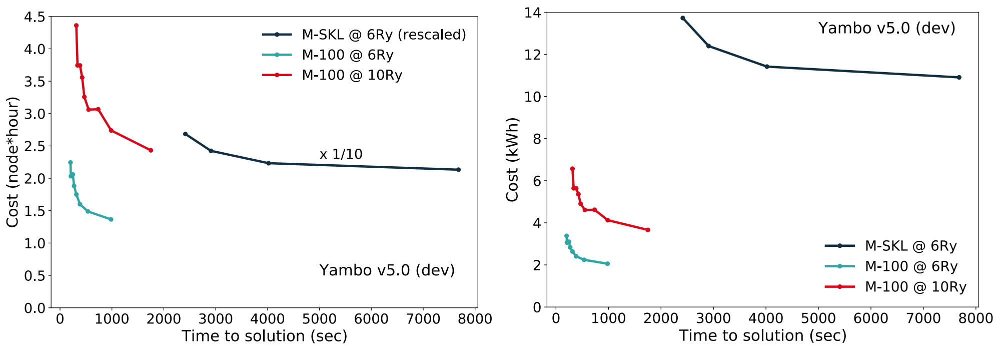


Figure 10: Node and energy costs of YAMBO runs performed on Marconi100 (M100) and Marconi-A3 Skylake (M-SKL) at CINECA. In the left panel, M-SKL data have been scaled dividing the number of node hours by 10 for the sake of visualisation. However it is relevat to notice that this conversion is also quite close to the actual ratio of nominal computational power for the two kind of nodes, namely about 30 and 3.2 TFlops/node for M100 and M-SKL, respectively.

factor 1/10 to improve readability. Being this factor close to the nominal ratio node compute powers (though slightly in favour of Skylake nodes), one can also read the plot as a graphical re-statement of the discussion above. Right panel shows instead the node*hour graph converted to energy cost by multiplying with the average power/node taken from the Top500 listing. Here the difference between M100 and M-SKL opens up further.

As a last comment, after the optimisation described above, the H-Rutile test case (GW run for a defected system with 72+1 atoms) has no longer to be considered as a large scale system, but rather a medium size calculation. Notably, we were able to exploit a parallel partition of Marconi100 of about 1.5 PFlops, with decent parallel performance, which is quite remarkable for a medium class job. We also note that a much larger partition exploitation of Marconi-100 (single runs up to 8 PFlops, with tests up to 20 PFlops) were reported in the deliverable D4.3 [2] for a graphene nanoribbon adsorbed on a graphene layer, a system studied in the context of WP6 as MAX demonstrator.

4.5 CP2K

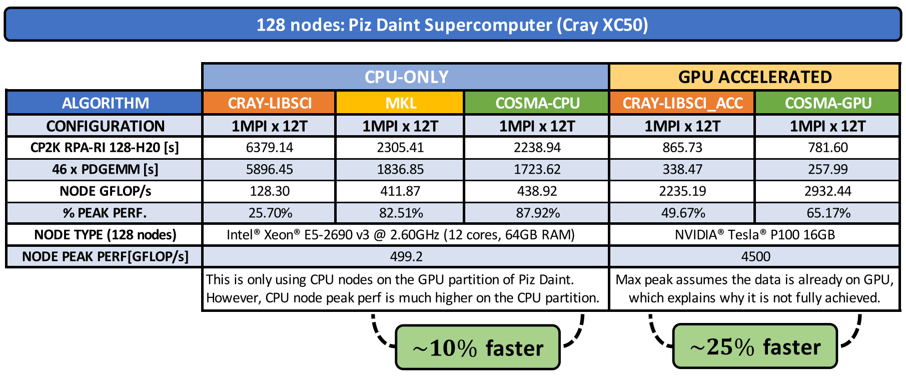
CP2K code is heavily relying on the external libraries for its performance portability as the code itself is written in a generic way without any architecture specific implementations. In particular, DBCSR and ScaLAPACK (as a source of distributed matrix-matrix implementation `pdgemm`) libraries must exhibit a descent performance on a given platform in order for CP2K to achieve a fast execution in O(N) and RPA types of calculation. AS such, the effort of optimising and porting CP2K to new architectures was channelled to the performance tuning of DBCSR and COSMA libraries. COSMA is a new library developed at ETHZ which implements communication-optimal matrix-matrix algorithm and provides a corresponding `pdgemm` wrapper natively used by CP2K. The results of the latest benchmark of RPA calculation of 128 water molecules with CP2K are collected in the Fig. 11.

4.6 SIRIUS

In the M19-M24 of the project the work on the SIRIUS library was focused around the development of the AMD GPU backed and benchmarking of the SIRIUS-enabled Quantum ESPRESSO code on AMD RI MI60 hardware. At the time of writing this report, the NDA agreement on the results is still enforced. However, we can make the following statements:

- (1) There is one last performance issue with the `ztrmm` rocBLAS function that will be fixed in the next releases of ROCm SDK.
- (2) The hipMAGMA implementation of the eigen-solver works and performs well.
- (3) The supported functionality of SIRIUS (total energy, stress, forces) works on AMD cards and thus SCF and lattice relaxation workflows of QUANTUM ESPRESSO also run on AMD GPUs.

4.7 BigDFT



128 nodes: Piz Daint Supercomputer (Cray XC50)

	CPU-ONLY			GPU ACCELERATED	
ALGORITHM	CRAY-LIBSCI	MKL	COSMA-CPU	CRAY-LIBSCI_ACC	COSMA-GPU
CONFIGURATION	1MPI x 12T	1MPI x 12T	1MPI x 12T	1MPI x 12T	1MPI x 12T
CP2K RPA-RI 128-H20 [s]	6379.14	2305.41	2238.94	865.73	781.60
46 x PDGEMM [s]	5896.45	1836.85	1723.62	338.47	257.99
NODE GFLOP/s	128.30	411.87	438.92	2235.19	2932.44
% PEAK PERF.	25.70%	82.51%	87.92%	49.67%	65.17%
NODE TYPE (128 nodes)	Intel® Xeon® E5-2690 v3 @ 2.60GHz (12 cores, 64GB RAM)			NVIDIA® Tesla® P100 16GB	
NODE PEAK PERF[GFLOP/s]	499.2			4500	
	This is only using CPU nodes on the GPU partition of Piz Daint. However, CPU node peak perf is much higher on the CPU partition.			Max peak assumes the data is already on GPU, which explains why it is not fully achieved.	

Figure 11: Performance comparison of the CP2K with ScaLAPACK and COSMA back ends. On 128 nodes, we compared the performance of CP2K using the following algorithms for multiplying matrices (pdgemm routine): MKL (version: 19.0.1.144), Cray-libsci (version: 19.06.1), Cray-libsci_acc (version: 19.10.1, GPU accelerated) and COSMA (both CPU-only and GPU-accelerated versions) libraries. The version with COSMA was the fastest on both CPU and GPU. The CPU version of COSMA achieved the peak performance, whereas the GPU version achieved more than 65% of the peak performance of GPUs. Keep in mind that the peak performance of GPUs assumes the data is already residing on GPUs which is not the case here, since matrices were initially residing on CPU. This is one of the reasons why the peak performance is not achieved with the GPU version. Still, the GPU version of COSMA was 25-27% faster than the second best in this case.

Performance portability

The libconv convolution library, using BOAST autogeneration and optimisation framework, is soon to be integrated in BigDFT. Stabilisation process is still ongoing, and efficient use of new architectures, such as ARM platforms with SVE vector instructions, is the focus of performance work on the BOAST side. Exploratory work has been performed this year using newly released A64FX processors (which are used on the RIKEN Fugaku supercomputer, the new first place holder of Top500) and non-SVE ThunderX2 processors (NEON instruction set), to evaluate effectiveness of ARM vector instructions on representative BigDFT kernels. This work was performed in collaboration with ARM, which granted used of their in-house compilers. Fujitsu compiler was also tested and compilation of BigDFT was performed with both compilers and the vendor's linear algebra tuned libraries. As these compilers are still under development to offer better support for these new processors, options for efficient optimisation were not easy to find, some of them being counterproductive. Using the ARM instruction emulator helped finding the amount of vectorised instructions for each setup, and converge to a set of compilation instructions yielding the best performance. These can now be used as a basis for an efficient use of BigDFT and other flagship codes. Only small modifications in BigDFT were needed to compile and run with these new compilers. Preliminary results on such kernels show that the large amount of bandwidth available on these processors allow an important increase in performance for our memory bound kernels, sometimes reaching performance only seen on GPU systems. This work will be continued in the next months when Fugaku will reach public availability and a more stable environment. Furthermore, an ARM-based docker container for BigDFT is now available to users, providing an easy

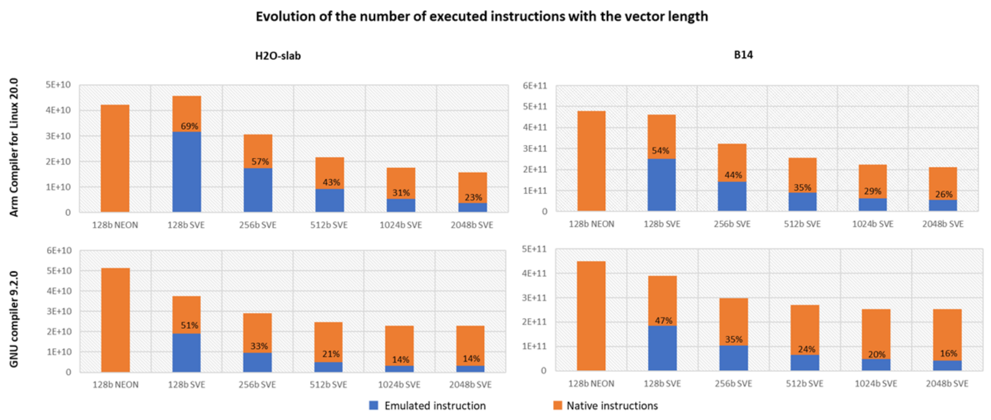


Figure 12: ARM Instruction Emulator estimation of the amount of instructions using H2O-slab and B14 test cases of BigDFT with different vector lengths, showing BigDFT's potential for gains with future ARM platforms with large vector sizes. A64FX supports SVE with up to 512-bit vectors.

way of using and testing BigDFT on ARM systems (with CUDA and OpenCl support as well). Preliminary results are shown in Fig. 12.

Performance prediction

Performance prediction with SimGrid for the libconv convolution library, discussed in D2.1, was successfully implemented, as reported in D4.3 report. The libconv library can now generate self-benchmarking reports on a real run, and use these reports to accurately predict timings in a different-sized simulated run. The timing is then injected directly inside the simulation by the library, skipping the costly real computation, to provide fast and reliable simulation on a single processor system, such as a laptop, for the behaviour of a large run on a large HPC system. This work was presented at SIAMPP20 in Seattle in February 2020.

5 Conclusions and ongoing work

The M24 release of the MAX flagship codes includes significant performance increases on essentially all currently available HPC platforms including GPU based machines and ARM CPUs. This was achieved by the exploration and implementation of different programming paradigms enabling us to be prepared for the deployment of our codes on current and future European HPC systems.

While the fact our codes are actually running on these different architectures already demonstrates portability and many examples also show satisfactory performance for many scientifically relevant examples, we have not yet been able to identify a suitable metric to compare code performance across very different machines in a universally acceptable way. Therefore, we will continue to evaluate different such metrics and options and will probably continue to use a number (possibly small) of different, application- and problem-adjusted views to guide our work.

The work done in this WP is and will be closely connected to the activities of the two other code-centric WPs (1 and 3). This is ensured by the fact that the work in these WPs is mostly carried out by the same set of people and correspondingly, the regular discussions are usually carried out in joint teleconferences, the joint MAX hackathon and similar exchange.

Concluding, we can assure that the work in this WP essentially proceeds as planned in these difficult times. Common activities covering more than a single code and in particular the exchange of ideas and experiences are slightly hindered by the lack of in-person meetings which we can only partly compensate with increased virtual discussions.

References

[1] 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).

[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] Genovese, L. *et al.* Second release of MaX software: Report on the evolution actions taken in each of the codes, and software release. Deliverable D3.3 of the H2020 CoE MaX (final version as of 30/11/2020). EC grant agreement no: 824143, CEA, France. (2020).

[4] van Setten, M. J. *et al.* Gw100: Benchmarking g0w0 for molecular systems. *Journal of Chemical Theory and Computation* **11**, 5665–5687 (2015).