D3.2

Interim report on programming models solutions for exascale efficiency

Pietro Delugas, Angela Acocella, Francesco Andreucci, Louis Beal, Laura Bellentani, Ivan Carnimeo, Fabrizio Ferrari Ruffino, Andrea Ferretti, Luigi Genovese, Ivan Girotto, Mariella Ippolito, Nicola Spallanzani, Sergio Orlandini, Fabio Affinito

Due date of deliverable	30/06/2024 (**month 18**)
Actual submission date	28/06/2024
Final Version	28/06/2024
Lead beneficiary	SISSA (participant number 2)
Dissemination level	PU - Public

Document information

Project acronym	MAX
Project full title	Materials Design at the Exascale
Research Action Project type	Centres of Excellence for HPC applications
EuroHPC Grant agreement no.	101093374
Project starting/end date	01/01/2023 (month 1) / 31/12/2026 (month 48)
Website	http://www.max-centre.eu
Deliverable no.	D3.2
Authors	Pietro Delugas, Angela Acocella, Francesco Andreucci, Louis Beal, Laura Bellentani, Ivan Carnimeo, Fabrizio Ferrari Ruffino, Andrea Ferretti, Luigi Genovese, Ivan Girotto, Mariella Ippolito, Nicola Spallanzani, Sergio Orlandini, Fabio Affinito
To be cited as	Delugas et al. (2024): Interim report on programming models solutions for exascale efficiency. Deliverable D3.2 of the HORIZON-EUROHPC-JU-2021-COE-01 project MAX (final version as of 28/06/2024). EC grant agreement no: 101093374, SISSA, Trieste, Italy.

Disclaimer

Contents

Executive Summary

This interim report presents the status and achievements of the T3.2 task within WP3. We first discuss three different strategies for transitioning from CUDA to programming models more suited for achieving functional and performance portability with all families of accelerators.

- In QUANTUM ESPRESSO , developers have adopted OpenMP and OpenACC as an alternative back end for offloading, gradually phasing out the previous CUDA-Fortran implementation. Several proposed interfaces have been adopted for the whole high-level code layer to maintain the source code uniqueness and transparency concerning the two backends.
- YAMBO developers present the use of the `deviceXlib` MAX library to integrate multiple offload back-ends inside large Fortran codes.
- In BIGDFT, crucial kernels have been implemented using the SYCL C++ programming model with significant results both in portability and performance.

Another part of this report is then dedicated to the work done in T3.2 on the `FFTXlib` of QUANTUM ESPRESSO . This latter has been successfully offloaded with the OpenMP backend. To improve the performance and scalability of this porting, T3.2 has taken charge of implementing a batched version of the library for the Cray/HIP toolchain. To understand the intrinsic scalability limits of this kernel, we are also performing a comparative analysis of FFTXlib performance versus analogous FFT libraries for accelerators (CuFFT_MP, heFFTe). We present the work done to design and implement an optimised band parallelization scheme that can efficiently side FFTXlib parallelism in QUANTUM ESPRESSO , overcoming its current limits in strong and weak scalability. We also present the work done by the BIGDFT group as an appendix for streamlining and optimizing the benchmarking process. We conclude with some final remarks and point out two main topics of oncoming activities in T3.2: data exchange within workflows and the improvement of check-pointing and code resilience.

1 Introduction

This interim report presents the status and achievements of the activities of task 2 of WP3. This task analyses the main challenges that the exascale transition poses to MaX code developers.

In the first part, this task of WP3 has helped the code developers achieve functional and performance portability of the codes to all the primary HPC machines and the many types of accelerators. The next challenge is the performance and scalability of the codes, which are massively parallel in the current pre-exascale and even more so in the oncoming exascale machines.

Regarding the porting, the outstanding problem was adapting these Fortran codes to the now-prevailing heterogeneous machines with accelerators. This first technical question has been substantially solved, and the chosen technical solutions are also presented in WP1 deliverables [1, 2]. In collaboration with WP1, we discuss in this document three adopted strategies that can guarantee the codes' functionality and performance for all families of accelerators. In section 2, we report briefly on the work done in QUANTUM ESPRESSO Fortran APIs to improve the code portability and its transparency to the various back-ends used for the offloading.

This work is going to be seamlessly integrated with the `deviceXlib` APIs presented in the previous WP1 reports [1, 2] to improve code maintainability and reusability. In Section 2.2 YAMBO developers discuss how the new features of the library presented in D1.2 [2] have been indeed beneficial for streamlining the porting to AMD (and later INTEL) accelerators. In Section 3, BIGDFT developers present their work with the C++ `SYCL` programming model, purposely conceived for portability and performance on generic accelerators. This Section also starts switching the focus from the porting strategies towards the analysis and the strategy towards performance.

The subject further passes from the porting to the optimization of performance-crucial kernels: In Section 4.1 we present the work done for the implementation of the batched operation of `FFTXlib` in the Cray-HIP software tool-chain used for the OpenMP offloaded version of `pw.x` on the LUMI/G partition. Even if this is still a portability issue regarding the HIP implementation of an analogue CUDA kernel, this work has paved the way for a more thorough analysis of the FFTXlib performance with accelerators and its scalability and communication overheads, particularly with an increasing number of ranks. The discussion continues in the following Section, where the performance of FFTXlib on CUDA accelerated nodes is compared with those of some native libraries to envisage the room for improvement that can be achieved by optimising the communications and synchronization within the library.

Notwithstanding all possible optimizations, it is well-known that, as for all latency-oriented parallelization schemes, the efficiency of FFTXlib parallelism is intrinsically limited and must be sided by a reciprocal throughput parallelization scheme where instead of speeding up every single task splitting its executions, we increase the number of tasks which are executed concurrently. The analysis and implementation plan for such a solution is presented in Section 5.

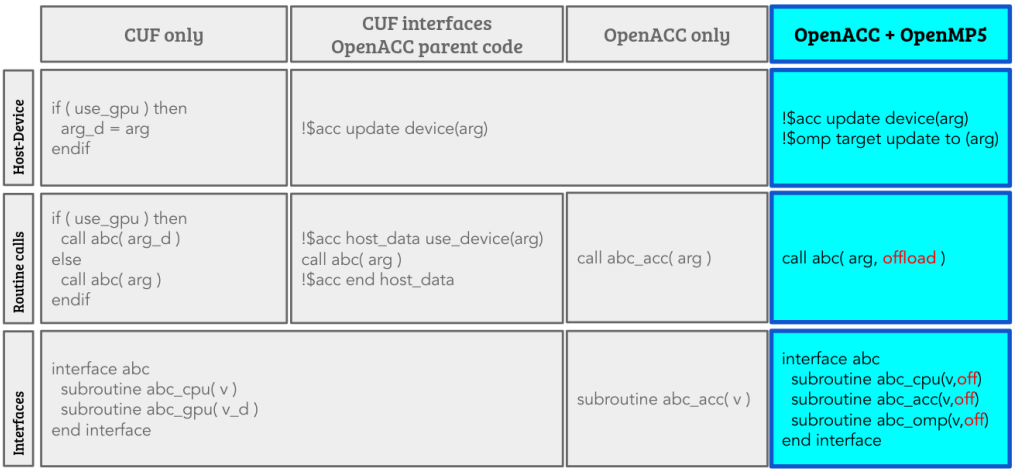

Figure 1: Schematic representation of different GPU code implementation schemes.

2 Unified Fortran interfaces for different GPU-offload schemes

A significant challenge hindering the support of the different HPC architectures presently available is the variety of software stacks adopted by GPU vendors. An efficient porting must cope with the details of each vendor specific technologies, such as compilers, libraries, and profilers. An overview of performance comparison among different GPU porting approaches on various HPC architectures can be found in a paper by Davis et al. [3]. MAX codes have also adopted slightly different approaches but always keep in view the goal of designing and adopting a unified programming paradigm for the various MaX codes. In this section, we analyze the strategies of QUANTUM ESPRESSO and YAMBO that have been able to support both CUDA and HIP back-ends.

QUANTUM ESPRESSO has concentrated most of the effort in transitioning to an OpenACC/OpenMP directive base model for the high-level parts of the code, and refactoring the APIs of its computational kernels so that they are transparent to the different back-ends and implementations. YAMBO has instead implemented the multiple back-end support using the MaX library `deviceXlib`. In the rest of this section, we present these two converging approaches in detail.

2.1 Strategies for multiple back-ends support in QUANTUM ESPRESSO

In QUANTUM ESPRESSO , the OpenMP offload has been implemented with directives using the OpenMP API 5.1 standard [4], which from here onwards will be referred to as the "OpenMP5" porting model. Currently, the development of `PWSCF` is about to be completed (branch `develop_omp5` in the official QUANTUM ESPRESSO repository [5]). This new porting is designed to seamlessly accommodate OpenACC and OpenMP5 directives in a single source code, enabling deployment on various hardware architectures (CPU, NVIDIA GPUs, and Intel/AMD GPUs) determined at compile time. This approach aims at achieving consistent performance across different GPU cards at runtime.

To target a diverse range of architectures, particularly based on Intel and AMD hard-

ware and software stack, a new porting of the QUANTUM ESPRESSO suite based on OpenMP offload has been initiated. The OpenMP offload has been developed using directives based on the OpenMP API 5.1 standard [4]. To reduce maintenance burden and improve code development, OpenMP directives are included together with the preexisting OpenACC directives [6], resulting in a single source code that can be deployed on various hardware architectures (CPU, NVIDIA GPUs and Intel/AMD GPUs) determined at compile time. This approach aims at achieving consistent performance across different GPU cards at runtime.

A critical point in developing code for the mixed OpenACC/OpenMP5 porting model was tailoring execution for CPU, CUF/OpenACC, and OpenMP5 paths, especially when each path necessitates routines specifically designed for a particular architecture. For instance, let us consider the case where distinct specialized algorithms are employed for CPU and GPU executions, and GPU execution is then further customized with a CUDA Fortran (or OpenACC) implementation tailored for NVIDIA architectures and an OpenMP5 implementation tailored for Intel/AMD architectures. Additionally, since variables can be stored in host or device memories, the CPU execution should also be kept accessible when the code is compiled with GPU flags. In the QUANTUM ESPRESSO suite, this scenario typically occurs, for example, in the FFTXlib library [7], for 3d FFTs, and in the `becmod` module, that contains many numerically intensive procedures with heavy matrix-matrix multiplications.

In Figure 1, schematic pseudo-codes illustrate different strategies to cope with this issue. In the old purely CUDA-Fortran approach [8], Fortran interfaces could recognize the `DEVICE` attribute of variables, invoking the right subroutine procedure accordingly. For instance, in the leftmost block of Figure 1, `v_d` is declared `DEVICE` in `abc_gpu`, and the type of the `arg`/`arg_d` variable in the parent code is sufficient to determine whether to invoke `abc_gpu` or `abc_cpu`. To resolve Fortran interfaces with OpenACC [6] and OpenMP5 [9] based models (rightmost block of Figure 1), we have defined three distinct derived types, one for each different execution path (CPU, CUF/OpenACC, OpenMP5). The `off` argument in `abc_cpu`, `abc_acc`, `abc_omp` is then declared with the specific type. Consequently, depending on the type of the `offload` flag in the parent code, a particular execution path is chosen.

2.2 Multiple back-end support YAMBO with `deviXlib`

The strategy adopted by the YAMBO team for portability on accelerated machines with different architectures has been extensively described in deliverable D1.2[1] [2], and is largely based on the use of the `deviceXlib` library. Several features of `deviceXlib` have shown to be key for a quick porting on AMD and Intel accelerators:

- **Broadened compiler compatibility:** the library effectively recognizes and works with Cray compilers and ROCm environments, expanding its reach to a wider range of systems. Similar extensions have been put in place for Intel environment.
- **integration of linear algebra subroutines**: besides NVIDIA cuBLAS, `deviceXlib` integrates with rocBLAS and MKL-GPU libraries, offering a subset of BLAS routines that can be transparently utilized through `deviceXlib` wrappers. This

[1] https://max-centre.eu/max-2023-2026

eliminates the need for developers to interact with rocBLAS/MKL-GPU directly, simplifying the development process.

- **Enhanced profiling on AMD/Intel architectures:** New APIs with ROCm ranges improve profiling readability on AMD architectures for Perfetto, a tool for visualizing rocprof traces. These APIs can also support other backends like Score-P and TAU.

In summary: by utilizing the `deviceXlib` library, particularly for managing data transfers between the CPU and GPU, we have successfully integrated support for three different back-ends for GPU offloading into a single source: CUDA-Fortran, OpenACC, and OpenMP-GPU. When specialized kernels are needed, extensive use of directives decorating the main loops is employed. These directives are preceded by pre-compiler macro's that enable the activation of one back-end at a time, selected during the configuration phase.

In recent months, we have continued to clean up the YAMBO source code, as this strategy facilitates more readable code. Additionally, with the support of CINECA HPC experts, we have been profiling the YAMBO code on accelerated systems, comparing the different back-ends. This has allowed us to identify unexpected metadata movements in the OpenACC porting, which we are currently correcting. Further testing is necessary for this activity.

3 SYCL in BigDFT

To enhance performance and tackle increasingly large problems, BigDFT has been GPU-enabled since 2009 [10] using Nvidia's CUDA language [11]. CUDA has been used to accelerate the Fock operator calculations (which is a key operation of highly accurate computational methods in electronic structure calculations), leveraging intensive CuFFT [12] calls in the *interpolating scaling function Poisson solver* of the code. As demonstrated in Ref. [13], the computationally expensive evaluation of the EXX in the cubic-scaling PBE0 approximation can be offloaded to GPUs, reducing computing time and making it competitive with the less accurate PBE approximation.

However, relying solely on CUDA for GPU acceleration is no longer sufficient, as it only supports Nvidia GPUs. With the advent of other GPGPUs, such as AMD's Instinct series [14] and Intel's Max series [15], used in several of the world's fastest supercomputers [16, 17], a broader approach is needed. Fortunately, CUDA and SYCL are quite similar, enabling a smooth transition from CUDA code to SYCL. Figure 2 (also presented in the SDP) illustrates the similarities and differences between the two language models by showing the same kernel written in both.

To facilitate the migration from CUDA, Intel provides the DPC++ Compatibility Tool (dpct) [18], which automates the code migration process. Indeed, the SYCL code depicted in Figure 2 was automatically generated using this tool. The drawback of this automated approach is the potential addition of a dpct interface layer, which can increase code complexity and affect performance. The SYCL implementation presented here is based on code automatically generated by the dpct tool[2] and manually refined and opti-

[2]Intel DPC++ Compatibility Tool version 2023.1.0. Codebase: (89a0192e122343c2a13cec7dc6d57cab899c7b64)

mized to achieve the performance demonstrated below.

The GPU-accelerated computations in BigDFT involve three-dimensional FFT-based Poisson equation solvers, integral to the Fock operator evaluation. As presented in Ref. [19], rather than using two large zero-padded three-dimensional FFTs (one forward and one backward transform) per iteration, BigDFT's implementation leverages multiple batched one-dimensional FFTs across the three spatial dimensions. The CUDA implementation employs the CuFFT library [12] for these FFTs. Conversely, the SYCL implementation utilizes the double-batched FFT (dbfft) library [20]. An alternative approach for the SYCL implementation could be the use of FFT algorithms provided by the Intel oneMKL [21], owing to its straightforward availability as part of Intel's oneAPI suite [22]. However, oneMKL's FFTs have several disadvantages compared to dbfft. First, the double-batching feature inherent in dbfft avoids explicit data transpositions (cf. Section 3 in [19]). Additionally, dbfft includes features for defining "callback functions" for load and store operations. These callback functions enable on-the-fly zero-padding, eliminating the need for explicit zero-padding required for solving problems with free boundary conditions (cf. Section 3 in Ref. [19]). Figure 3 presents two pseudo-codes outlining the two different algorithms.

Thus, utilizing the dbfft library approximately halves the GPU memory requirements compared to the CUDA implementation based on CuFFT or an alternative implementation using oneMKL FFTs, and reduces the required memory bandwidth by avoiding data transpositions. This results in significant performance gains, especially on the CPU. The downside of using dbfft is that the BigDFT-SYCL implementation cannot run on Nvidia or AMD GPUs due to the lack of support from dbfft. Extending the SYCL code to fully support various non-Intel backends and to leverage SYCL's cross-platform capabilities remains future work.

We have evaluated the performance of the new SYCL implementation on the Ponte Vecchio (PVC) Intel GPU and CPU against the existing CUDA and CPU (OpenMP) implementations using the "Fock miniapp". The Fock miniapp is a small test program designed to assess the performance of computing the Fock operator mirroring the actual evaluation in the complete BigDFT suite. Extensive tests have been conducted on several runtimes within a PVC-based architecture, details of which are under non-disclosure and will be presented in an upcoming publication. The results provided here highly represent the full suite's performance in many scenarios, as the Fock operator evaluation is the most time-intensive computation in the full suite when employing the PBE0 approximation. The initial workload involves a single Fock operator evaluation for a grid size of 256 points in each of the three dimensions (totaling 256^3 points), 64 orbitals, and free boundary conditions (resulting in a Poisson solver grid size of 512^3 points).

The results of three different code versions, namely, the original CPU implementation, the SYCL implementation on the CPU, and the SYCL implementation on Intel Max 1550 GPUs, are shown in Fig. 4 in red, green, and blue, respectively. One can observe that: (i) the SYCL implementation is significantly faster on the CPU than the original CPU implementation by approximately a factor of two, (ii) the SYCL implementation on the Intel GPU significantly outperforms the other implementations, and (iii) the scaling of the GPU implementation levels off from 1 to 8 nodes (i.e., 4 to 32 GPUs) due to the increasingly small computing times which fail to hide the communications. For the GPU runs, 1 MPI rank per GPU stack was used (i.e., 2 MPI ranks per GPU, 8 MPI ranks

```
//CUDA
__global__ void kernel(int nx, int ny, int nz,
double *rho, double *data1, int shift1,
double *data2, int shift2, double hfac) {

  int tj = threadIdx.x;
  int td = blockDim.x;
  int blockData = (nx*ny*nz)/(gridDim.x*gridDim.y);
  int jj = (blockIdx.y*gridDim.x + blockIdx.x)*
    blockData;

  for (int k=0; k<blockData/td; k++) {
    int idx = jj + tj + k*td;
    data1[idx+shift1] = data1[idx+shift1] +
      hfac*rho[idx]*data2[idx+shift2];
  }
}

//SYCL
void kernel(int nx, int ny, int nz,
  double *rho, double *data1, int shift1,
  double *data2, int shift2, double hfac,
  const sycl::nd_item<3> &item) {

  int tj = item.get_local_id(2);
  int td = item.get_local_range(2);
  int blockData = (nx*ny*nz) /
    (item.get_group_range(2)*
    item.get_group_range(1));
  int jj = (item.get_group(1)*
    item.get_group_range(2) +
    item.get_group(2))*blockData;

  for (int k=0; k<blockData/td; k++) {
    int idx = jj + tj + k*td;
    data1[idx+shift1] = data1[idx+shift1] +
      hfac*rho[idx]*data2[idx+shift2];
  }
}
```

Figure 2: Example of one of the BigDFT CUDA kernels compared to the SYCL equivalent to demonstrate the commonalities between SYCL and CUDA. The SYCL code was automatically generated using the Intel DPC++ Compatibility Tool version 2023.1.0. Note that there is a manually translated version of the above SYCL code, which is simpler and which is used in the latest version of BigDFT.

```
if (<Boundary condition in x is free>)
  <zero-pad in x direction>
1DFFT_X(Ny*Nz, Sx) //real-to-complex FFT
if (<Boundary condition in y is free>)
  <zero-pad in y direction>
<transpose data>
1DFFT_Y((Sx/2+1)*Nz, Sy)
if (<Boundary condition in z is free>)
  <zero-pad in z direction>
<transpose data>
1DFFT_Z((Sx/2+1)*Sy, Sz)

<Convolution kernel multiplication>

inverse_1DFFT_Z((Sx/2+1)*Sy, Sz)
if (<Boundary condition in z is free>)
  <remove padding in z direction>
<inverse transpose>
inverse_1DFFT_Y((Sx/2+1)*Nz, Sy)
if (<Boundary condition in y is free>)
  <remove padding in y direction>
<inverse transpose>
inverse_1DFFT_X(Ny*Nz, Sx) //complex-to-real
if (<Boundary condition in z is free>)
  <remove padding in x direction>
```

```
if (<Boundary condition in x is free>)
  <set callback to add zeros>
1DFFT_X(Ny*Nz, Sx) //real-to-complex FFT
if (<Boundary condition in y is free>)
  <set callback to add zeros>
1DFFT_Y((Sx/2+1)*Nz, Sy)
if (<Boundary condition in z is free>)
  <set callback to add zeros>
1DFFT_Z((Sx/2+1)*Sy, Sz)

<Convolution kernel multiplication>

inverse_1DFFT_Z((Sx/2+1)*Sy, Sz)
inverse_1DFFT_Y((Sx/2+1)*Nz, Sy)
inverse_1DFFT_X(Ny*Nz, Sx) //complex-to-real
```

Figure 3: Pseudocode to demonstrate the simplification yielded by utilizing the double-batched FFT library. Removing the transpose operations and the explicit zero – padding also increases the performance. The first arguments in the FFT calls above are the batch size, the second argument the size of each FFT. The variables Nx, Ny, and Nz denote the domain size in x, y, and z directions, respectively. The variables Sx, Sy, and Sz denote the padded problem size and is either equal to the domain size Nx, Ny or Nz, (in case of boundary conditions differing from free boundary conditions) or twice the domain size in the respective direction.

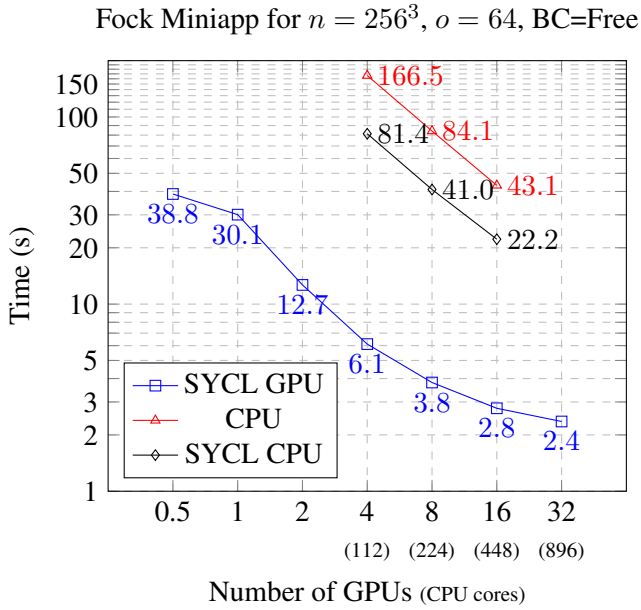

Figure 4: Computing time of the Fock miniapp for free boundary conditions, a grid size of $n = 256^3$ and $o = 64$ orbitals. In the GPU case, one MPI rank was pinned to each stack of a GPU up to a maximum of 8 MPI ranks per compute node. In the CPU cases 16 MPI ranks per node were used. Note that the CPU tests on 8 nodes are missing since the number of MPI ranks cannot exceed the number of orbitals. The first data point was performed on one stack of the two stacks of the Intel GPU.

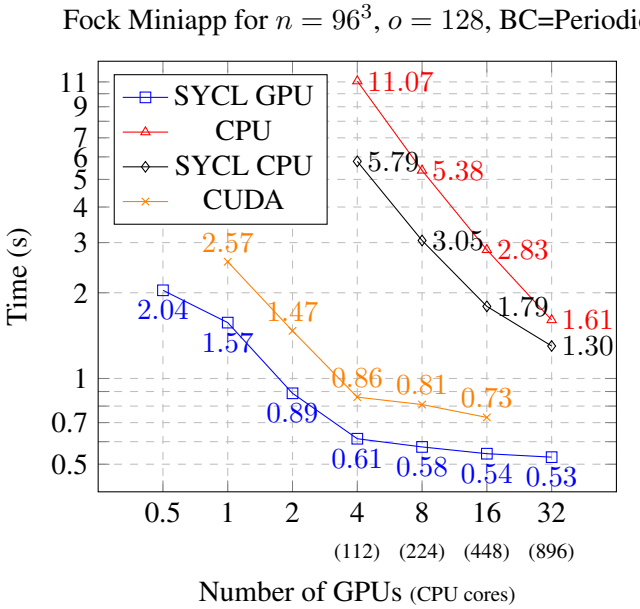

Figure 5: Computing time of the Fock miniapp for periodic boundary conditions, a grid size of $n = 96^3$ and $o = 128$ orbitals. In the SYCL GPU case, one MPI rank was pinned to each stack of a GPU up to a maximum of 8 MPI ranks per compute node. In the CPU cases 16 MPI ranks per node were used. Note that this workload coincides with the exact exchange computation performed during a full solve of the H2O-32 case presented in Figure **??**. Note that in the CUDA case, two GPUs constitute a node whereas in all other cases it is four GPUs per node. The first data point was generated on one of the two stacks of the Intel GPU.

per node), whereas 16 MPI ranks per node were used for the CPU runs.

The second workload shows a Fock operator evaluation for a grid size of 96 points in each of the three dimensions (for a total of 96^3 points), 128 orbitals, and periodic boundary conditions (which does not require zero-padding and thus maintains a grid size of 96^3 points for the Poisson solver).

In addition to the previous three configurations, the graph also shows results achieved on NVIDIA A100 GPUs with the CUDA implementation. The CUDA results were generated by pinning two MPI ranks to each A100 GPU to utilize the same number of MPI ranks per GPU as in the SYCL tests. One can observe that the SYCL implementation on the Intel Max GPU is highly competitive in terms of computing times. Similarly to the workload shown above, the scaling in the SYCL GPU case degrades significantly for more than a single node and the computing times even increase from four nodes to eight nodes due to increasing time required for the communication. The different CPUs in the SYCL-GPU and CUDA cases have minimal impact on the presented execution times since the majority of the timed code is executed on the GPUs.

Table 1 compares the average HBM and L3 bandwidths during the single-stack executions on the Intel GPU for the workloads described above. It is clear from the table that the smaller workload ($n = 96^3$) fits into the L3 cache, thereby generating minimal HBM traffic. Conversely, the larger workload exceeds the L3 cache capacity, resulting

Case	Read (GB/s)		Write (GB/s)		Read+Write	
	L3	HBM	L3	HBM	L3	HBM
$n = 256^3, o = 64$	0.02	468	336	315	336	783
$n = 96^3, o = 128$	1172	66	1744	48	3516	114

Table 1: Comparison of the average read and write L3- and HBM-bandwidths achieved by the Fock miniapp on a single stack of the Intel GPU. The smaller case ($n = 96^3$, $o = 128$) fits in L3 cache and therefore hardly utilizes HBM. The larger case ($n = 256^3$, $o = 64$) does not fit in L3 and induces heavy HBM traffic.

in significant HBM traffic. For the larger workload, the code utilizes an average of 783 GB/s of HBM bandwidth, equating to approximately 48% of the theoretical peak bandwidth of a single stack. Notably, the complex-to-complex FFT with the highest average bandwidth among all complex-to-complex FFTs achieved a bandwidth of 938 GB/s.

4 Optimization and multiple back-end support in FFTXlib

Most of the computational cost of a typical `PWSCF` calculation is due to Fast Fourier Transform (FFT) operations. Three-dimensional FFTs in the QUANTUM ESPRESSO suite are executed using FFTXlib [7], a specialized library tailored to accommodate the internal data distribution of the wavefunction (and charge density). It leverages the properties of DFT-related datasets, such as band structure, cutoff, and dual parameters, and supports slab and pencil decompositions. The former method involves a slab-based partition of the direct space, where the input function transforms the entire x-y domain for a subset of values along the z-axis. Each subgroup is allocated to a specific processor, and the Fourier transform along the z-direction is carried out based on the 'z-stick' distribution of the reciprocal space.

For each x-y point falling within the cutoff circle, the function is transformed across the entire z-range. This algorithm entails one 2d FFT operation for x-y slab transforms, one stage of collective communication (typically MPI all to all), which involves distributing the z-sticks among processors, and one final 1d FFT operation on the z-sticks. The uniqueness of the QUANTUM ESPRESSO version of this algorithm lies in the z-stick decomposition of the reciprocal space and the utilization of the energy cutoff. This mapping of the square grid covering the direct space into a smaller one inscribed within the cutoff sphere allows for memory and data movement optimizations.

Pencil decomposition works similarly to slab decomposition but transforms along the x and y directions separately, involving only 1d FFT operations. While it enables more efficient memory distribution, it requires an additional stage of communication compared to slab decomposition. Consequently, with the increasing memory availability of modern GPU devices, slab decomposition has become the preferred method in recent years.

Panel (a) of Figure 6 provides a schematic overview of the main steps of an inverse FFT computation, performed with slab decomposition on 4 MPI processes: even on small systems, the time spent in communication and data movement significantly exceeds that related to actual 1d and 2d FFT computations.

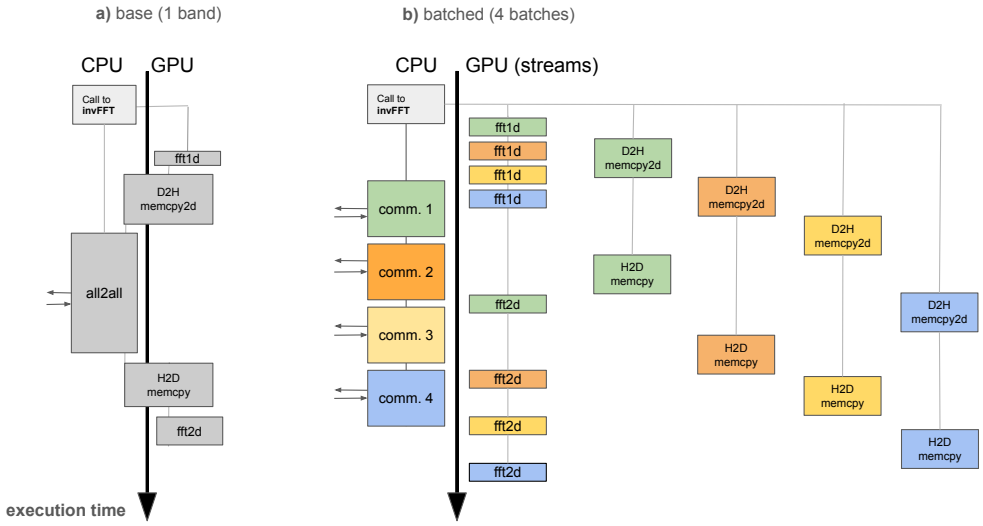

Figure 6: Illustrative scheme of FFT with slab decomposition: a) base accelerated inverse FFT algorithm; b) asynchronous batched inverse FFT algorithm with 5 streams and 4 sub-batches (one for each colour). The main data movements are done with memcpy2d and memcpy routines. MPI communications without GPU-aware MPI are performed among CPUs and require host-device synchronizations, whereas when GPU-aware MPI is enabled, they involve GPU-direct device-device communications. The time scale of the two diagrams is not the same.

4.1 Asynchronous batching in FFTXlib in the HIP/OpenMP CrayFCE software stack

The GPU porting of FFTs for NVIDIA-based architectures [8] leveraged the extensive internal parallelism of GPU cards to concurrently process many bands at a time within each FFT operation. This was achieved by segmenting the wavefunction data structures into batches and sub-batches, each containing a fixed number of bands, and using streams to process sub-batches asynchronously. This algorithm was initially developed using CUDA Fortran and has not undergone rewriting in OpenACC during the recent refactoring [6] of the QUANTUM ESPRESSO suite. It has now been expanded [9] to support AMD-based architectures using the HIP language, since OpenMP does not allow one to work on multiple GPU streams within a single CPU task. For Intel architectures, GPU acceleration is entirely based on OpenMP directives, and only the base FFTXlib algorithm (shown in Figure 6a) has been ported [9]. Work is currently underway to include the asynchronous streams.

In the batched algorithm [8, 9], schematically summarized in Figure 6(b), all the 1d and 2d FFTs, needed to fully Fourier transform a batch, are performed on one dedicated stream (usually stream 0), while data movement and copies are performed on different streams, one for each sub-batch. Other small operations are performed on stream 0 too, by means of explicit HIP kernels. MPI communications are called for each sub-batch asynchronously with respect to the computation and the data movements related to all

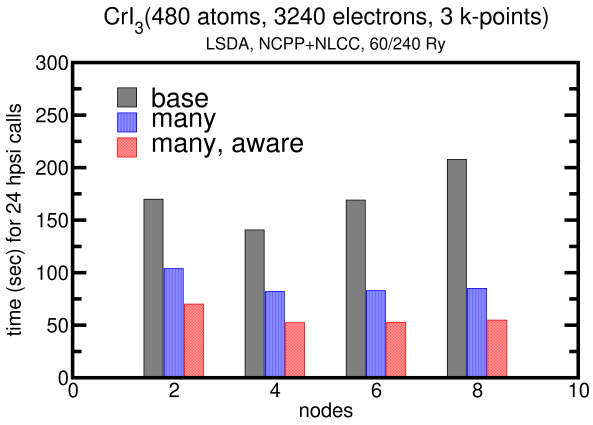

Figure 7: *R*&*G* scaling of `h_psi` over the number of nodes in LUMI cluster. “many" refers to the batched FFT algorithm, “aware" refers to GPU-aware MPI, “base" is without neither “many" nor “aware".

the other sub-batches. In this way, the overlap in time between communication and computation is brought out at its maximum, taking into account the constraints coming from the finite bandwidth of CPU and GPU communications.

In Figure 7, the scaling over plane waves (*R*&*G*) on LUMI, at fixed number of pools, is shown for the same system, and it is noteworthy that using the batched FFT algorithm and GPU-aware MPI it is possible to decrease communication bottlenecks and scale the calculations beyond 2 nodes.

4.2 Alternative interGPU communication layers for FFTXlib

In WP3, we aim to provide technical analysis and insight into the main performance-relevant kernels of MAX codes. Our strategy for performing this analysis is implementing mini-apps, which extract and encapsulate the targeted core libraries and software modules from the main applications to ease their performance analysis, testing, and optimization.

Regarding QUANTUM ESPRESSO (QE), this opens up the possibility to deeper investigate the performance of its domain-specific library for FFT operations, FFTXlib, and to make comparisons with well-known vendor APIs, highly optimized on hardware, but not “DFT tailored”. For example, cuFFTMp [23] demonstrates scalability up to many processes and implements new communication protocols (NCCL) and GPU memory management (NVSHMEM) optimized for NVIDIA GPUs. On the other side, it does not allow one to exploit the DFT cutoff radius in reciprocal space and to efficiently set the distribution of the input and output data to match the one adopted by QE. Moreover, the typical sizes of the FFT grids in QE lie just before the lower bounds of the cuFFTMp available benchmarks, so it would be interesting to expand the lower side of such benchmarks, considering the comparison with FFTXlib.

The library cuFFTMp has been integrated into the FFTXlib mini-app during the MAX Hackathon of April 8-12th, 2024, as an experimental alternative for 3D FFTs.

Unlike the GPU driver currently implemented in FFTXlib, cuFFTMp uses symmetric GPU memory to distribute data, one-sided communications to exchange buffers and it enables standard pencil or slab decompositions along with an optimized built-in default distribution.

The integration of the cuFFTMp API was performed by explicitly allocating the GPU buffers on the "symmetric heap", a memory layer shared across multiple devices through the new technology provided by NVSHMEM. These allocations are done explicitly by overcoming specific descriptors of the cuFFTMp library to streamline the eventual integration of NVSHMEM parallel programming interface only - beyond its use in cuFFTMp APIs - in the future.

The current stage of implementation achieved after the hackathon allows the user to select a custom slab/pencil distribution in real (R) and reciprocal (G) space, which ensures good flexibility but not enough to match the peculiar distribution of data in G space of the FFTXlib library, which is based on z-sticks confined into a G-cutoff sphere and uniformly distributed among the MPI processes. Interfacing cuFFTMp to the QE data distribution would require additional effort, which will be considered only if the cuFFTMp library proves competitive enough concerning the current FFTXlib performance.

All the cuFFTMp initialization, execution and destruction routines have been experimentally integrated within the FFTXlib library by following its native scheme based on interfaces. The next steps will be focused on the accurate assessment of cuFFTMp and cuFFT performance in the grid size range of interest for QE, as well as the production of an overall validated benchmark reference for the FFT library. Results will be summarized and reported via the Jülich Benchmarking Environment (JUBE) application. Documentation will also be improved.

Regarding the approach to FFTXlib profiling on NVIDIA GPUs, we integrated into the library a module containing specific APIs to start and stop NVTX ranges. NVTX is a lightweight library tool provided by NVIDIA that can be used to wrap source code regions and assign them a label and further attributes. In addition to enhancing the readability of profiles, these ranges can be used for filtering and extracting the time spent by the runtime on GPUs, without adding synchronizations with the CPU, which is different from clocks. This will enable us to extract the time for a range projected on the GPU and distinguish between time spent in kernel computation and time spent in GPU to GPU data movement overhead of kernel launches, MPI library, etc.

5 Analysis and proof of concept implementation of band distribution of the QE workload

As the above Section shows, FFT operations on the many wave functions generally constitute the primary workload of QUANTUM ESPRESSO executables. The memory and workload of these operations may be mitigated by the R&G parallelism distributing workload and data among the R&G MPI ranks, but as all latency-targeted parallelizations that aim at reducing the operations time per task, this has an intrinsically limited efficiency. Also, the distribution of each wavefunction on each R&G rank poses significant limits in terms of weak scalability, particularly with GPUs.

Band parallelization is conceived in QUANTUM ESPRESSO to provide a complementary throughput-targeted parallelization scheme that distributes the wave functions

among independent MPI R&G groups. The efficient and scalable implementation of this distribution —both in the strong and weak sense— is crucial for the many-nodes performance of all the applications in the suite. Current implementations of band parallelization are either limited to a few code parts or are limited in scalability by their too-large memory footprint. These limitations are even more significant when the code works with GPUs because the single-device memory amount poses a stronger limit.

For this reason, in WP3 Task 2 we have started analyzing in detail the performances and requirements of the alternative implementations for the band parallelism. Within WP3, we will apply and compare the different technical solutions into a simplified mini-app that reproduces the main functions of the QUANTUM ESPRESSO applications, mimicking the main interactions and data exchange between the most significant kernels of these codes. In the following Section, we first present the different ways in which the band parallelization is currently implemented in QUANTUM ESPRESSO and then briefly describe the mini-app organization, how this can characterize band-parallelization performance for QUANTUM ESPRESSO and the changes that are necessary for each mini-app part to experiment with the various schemes.

QUANTUM ESPRESSO band-parallelism. Several somewhat incompatible band parallelization schemes are already present in QUANTUM ESPRESSO . Of these, the band groups and the task-group parallelization work using different replicas of the R&G group and operate separately from different groups of bands. The main difference between the two schemes is that the task-group distribution is implemented internally in FFTXlib and leverages its data-structure descriptors to work, while the bands-group parallelism uses distinct R&G groups created externally to the FFTXlib data structure and thus works autonomously in each MPI group. They keep the whole set of bands in memory, with a significant overhead in communication and memory occupation. A third parallelization scheme is implemented to calculate the Fock potential. In this case, we need to compute an $\mathcal{O}(N^2)$ number of independent terms. It is thus more efficient to store entire wave functions within each R&G rank and distribute the calculation of such independent terms accordingly.

Mini-app description

Our test application embeds two main kernels, which are presented below.

The `h_psi` kernel. The first kernel implements a toy Hamiltonian. It is made by operations on single wave functions such as the kinetic energy operator in reciprocal (G) space and applying a local potential in direct (R) space. In such a way, this kernel mimics the single band operators that can be found in QUANTUM ESPRESSO quantum-engines and other applications, involving subsequent operation in R and G space, with, in the middle, the many wave functions FFTXlib calls to pass from R to G representations. Being made of autonomous single band operations, this kernel is naturally band-parallelizable with very few adaptations. The parallelization can be done by distributing the band or maintaining shared band data among the band groups in the same way as currently implemented in QUANTUM ESPRESSO . This second approach requires further collective communication to synchronize the data band structure, while with the second approach,

no further work is needed in this kernel. In figure 8, we show a snippet of how the main driver of the `h_psi` kernel.

Iterative Solvers. The second main kernel used in the mini-app performs the iterative diagonalization of the single-band operator described above. For this part, we will work on the `KS_solver` library provided in QUANTUM ESPRESSO , which implements several of these iterative solvers. In this part, the band data distribution becomes non-trivial because these solvers need to access and operate successively on all the pairs of wave functions, which implies that each band group must access the other band group data. The iterative solvers will be refactored to work in blocked mode to avoid sharing the whole band data structure. Each band group will thus be assigned its own intra-block elements plus a set of interblock couples. Each band group will successively store only the elements from another band group. Appropriate strategies for improving the work balance among the band groups can be implemented (see figure 9). Moreover, operating in blocked mode allows for the adoption of asynchronous data exchange strategies that will enable overlapping communication and computation within the iterative solvers and in any other code part where it is necessary to perform data exchange between two band groups.

Other Libraries. This test application also involves the usage of other important libraries of the QUANTUM ESPRESSO suite that may need to be adapted and expanded with the support for the new band parallelism schemes.

- `FFTXlib`: this library is heavily used by the single-band kernel of the mini-app. Its work distribution in the R&G groups is completely transparent to the band group distribution, and thus, none or very small adaptations will be needed for this library. The main consequence, when the band parallelization scheme is fully implemented, will be the possibility to simplify the library further, removing the task group distribution functionality that rather complicates the library and its maintainability.

- `LAXlib`: this library is used for all the distributed linear algebra operations occurring during the iterative solvers' operations. Its APIs distribute the data within the diagonalization group and give transparent access to the accelerated eigensolvers. We are studying and testing major adaptations to the new band parallelization scheme. LAXlib needs to collect and redistribute data to all band groups, either gathering them on one process to operate the accelerated eigensolver or redistributing them among the group that works the distributed solver. To implement the blocked Davidson algorithm, it is also necessary to extend the API that redistributes the matrices from the band groups to the ortho groups; this will also be instrumental to the possibility of generic BLACS block-cycling distributions that allow the optimal performance of distributed solvers.

- `UtilXlib`: MPI initialization, error handling timing and profiling in the mini-app are fully compliant with QUANTUM ESPRESSO conventions and performed using UtilXlib APIs. This has allowed us to test the library can manage the new parallelization schemes.

```
!
! Copyright (C) 2002-2016 Quantum ESPRESSO group
! This file is distributed under the terms of the
! GNU General Public License. See the file `License'
! in the root directory of the present distribution,
! or http://www.gnu.org/copyleft/gpl.txt .
!--------------------------------------------------------------------------
SUBROUTINE cb_h_psi( lda, n, m, psi, hpsi )
  !----------------------------------------------------------------------------
  ! ... This routine computes the product of the Hamiltonian
  ! ... matrix with m wavefunctions contained in psi
  !
  ! ... input:
  ! ...    lda   leading dimension of arrays psi, spsi, hpsi
  ! ...    n     true dimension of psi, spsi, hpsi
  ! ...    m     number of states psi
  ! ...    psi
  !
  ! ... output:
  ! ...    hpsi  H*psi
  !
  ! --- Wrapper routine: performs bgrp parallelization on non-distributed bands
  ! --- if suitable and required, calls old H\psi routine as cb_h_psi_
  !
  ! band parallelization with non-distributed bands is performed if
  ! 1. enabled (variable use_bgrp_in_hpsi must be set to .T.)
  ! 2. there is more than one band, otherwise there is nothing to parallelize
  !
  call start_clock('h_psi')
  IF (use_bgrp_in_hpsi .AND. m > 1) THEN
     ! use band parallelization here
     CALL divide(inter_bgrp_comm,m,m_start,m_end)
     hpsi(:,:) = (0.d0,0.d0) ! array must be initializad to zero to prevent adding garbage
     ! call the routine if there at least one band in this band group
     IF (m_end >= m_start) CALL cb_h_psi_( lda, n, m_end-m_start+1, psi(1,m_start), hpsi(1,
     CALL mp_sum(hpsi,inter_bgrp_comm) ! collect the result across the band group partners
  ELSE
     ! don't use band parallelization here
     CALL cb_h_psi_( lda, n, m, psi, hpsi )
  END IF
  call stop_clock('h_psi')
  RETURN
  !
END SUBROUTINE cb_h_psi
```

Figure 8: Snippet of the `h_psi` part of the mini-app, this can operate in parallel with the distributed and the shared band data set.

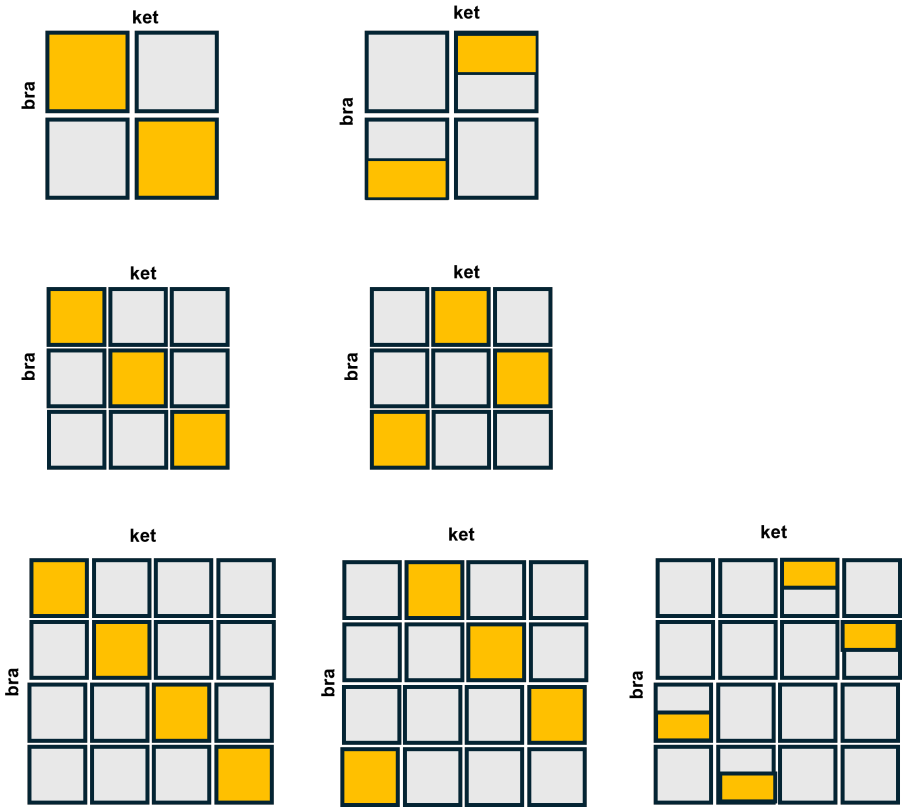

Figure 9: Work distribution with bands groups working in blocked modes inside `KS_Solvers` with band pairs operations. The *bra* group of bands is the one that is assigned to each band group, while the *ket* group changes cyclically. For an even number of band groups, the same final bra-ket pair is always assigned to two band groups, and a further work distribution is possible.

- `DeviceXlib`: The mini-app will be tested in all different types of architectures and targeted back-ends. This will also offer the opportunity to set up and test the code unification strategies in QUANTUM ESPRESSO . These strategies are based on the adoption wherever possible of the new `DeviceXlib` APIs that will be gradually introduced in all libraries involved in the mini-app and will provide a model for the introduction in the main source of QUANTUM ESPRESSO

6 Conclusions

We have presented above our work on analyzing some of the technical problems, with special emphasis on programming models, with which the development of MAX flagship codes must cope.

The first two Sections present the strategies adopted in MAX codes for supporting multiple offload back-ends. In Section 2, we have seen the cases of QUANTUM ESPRESSO and YAMBO, where the high-level parts of the codes have been refactored so that they remain transparent to the underlying offload back-end adopted, with all data-movement and operation on GPUs done either with directives or via the APIs. A significant role in this strategy is played by lower level libraries, such as the multi-purpose library `deviceXlib` that provides general tools for the offloading or the compute-intensive kernels of QUANTUM ESPRESSO . In Section 3, we have seen how, as in the case of some compute-intensive kernels of BIGDFT, performance portability can be achieved by adopting the `SYCL` programming model that, together with the portability, can also provide a visible performance improvement.

In Section 4, we have inspected in detail the work done and ongoing for improving the performance and the portability of the most compute-intensive kernel of QUANTUM ESPRESSO : `FFTXlib`. For this kernel, WP3 Task 2 has worked out the HIP version of the library for AMD GPUs. It is also working at streamlining communications on NVIDIA cards using this vendor's communication libraries. Another part of the work of WP3 in collaboration with WP1 concerns refactoring the band parallelization of QUANTUM ESPRESSO applications. In section 5, we show the analyses and proposed improvements for the band parallelization.

From the experience gathered in working on the subject of the present report, we can draw some general remarks:

- The portability and single-node performance solutions are already well-settled and have, for the most part, been applied to the production codes;
- Regarding the performance challenges, the information gathered by the work reported here provides important guidelines, strategies, and a sound estimate of the achievements regarding efficiency and performance improvements.

During this first part of the work, we also learned an essential methodological point: the adoption of mini-apps, a lesson which came as a plus from the collaboration with WP4. Thanks to their adoption, we could exploit the advantages of separation of concerns due to the modularization and encapsulation that MAX codes have undergone. With their adoption for this task, we have quickly analysed complex issues, identified clear bottlenecks and experimented with technical solutions, avoiding the supplementary complexity of the main codes.

Regarding the technical results, we would like to point out in this closure the thorough characterization and analysis of the technical issues concerning the scalability of QUANTUM ESPRESSO . We have first analysed the main workload constituted by FFTXlib and evaluated the possible technical improvements for this kernel with the estimate of the performance improvements that can be reasonably expected from the optimization of communications and the computation/communication overlap. Once we had defined this improvement strategy and its realistic targets, we turned to optimise the band parallelism, which constitutes its complementary throughput parallelization scheme. Combining these two solutions will enable the expected strong and weak scalability results.

We conclude with an outlook of the endeavours of task T3.2; apart from completing the pending points presented here, the main technical points that will be dealt with in the future will be more related to the realization of the scientific workflows—in collaboration with WP2 for the data exchange between workflow components and with WP1 concerning check-pointing robustness and reliability.

```
[3]: # init with JUBE4MaX template and platform.xml file
     test = JUBETemplate(template="leonardo/submit.job", platform="leonardo.
       ↪xml")
```

Figure 10: Setup of a `JUBEInterop` Template instance.

A JUBE and remotemanager

Benchmarking is an important tool in the HPC world. JUBE is a tool which automates the benchmarking process, with the aim of increasing the reproducibility of benchmarks. In turn, this allows for the extraction of performance metrics and for the detection and study of trends. In the following we present a benchmarking procedure, developed within the BigDFT team, making use of JUBE and `remotemanager`.

As part of the reproducibility goal for benchmarking, JUBE uses a custom input/output format for calculation definitions and results. The machine which is to run the calculation is also defined here, using an xml format "platform" file, along with a plain text job script template. The latter files are those which `remotemanager` is currently able to interact with.

A.1 remotemanager

`remotemanager` is the lightweight submission engine developed by the BigDFT team. "Submission engine", in this context means that `remotemanager` is responsible for the submission of jobs on a remote machine. Similarly to JUBE, job scripts are generated and used for this purpose, however `remotemanager` also has functionality to do this over a network connection.

In order to achieve this, `remotemanager` must handle the connection parameters, file transfers, and job script generation. These are automated by the tool after some basic user input. Job scripts are generated similarly to JUBE, in that machines are defined with a template job script, which then exposes parameters to be changed for the workflow. It is thanks to this similarity that it is possible to utilise the JUBE machine specifications to submit jobs using `remotemanager`.

A.2 JUBEInterop

The `JUBEInterop` module is an interoperability module within `remotemanager` that allows it to ingest a JUBE template file and generate a job script which can be used for remote submission. Setup of this interop module is designed to be simple and in the `remotemanager` style. It is enough to import the module, then pass the file paths to `template` and `platform`. When created, this object inherits all of the remote accessing abilities of the internal `remotemanager.URL` module (which handles the machine connection), yet gains the functionality expected of a JUBE platform.

This allows `remotemanager` to directly submit jobs on a machine defined in the JUBE format. We can demonstrate the job script that would be generated if this instance was used by `remotemanager` to submit a calculation by directly calling the `script()` method. See figure 11 for an example output. Within the `platform.xml` file, parameters can be specified such that they depend on other parameters. One such

```
[6]: # RemoteManager compatible script generation
     print(test.script())
```

```
#!/bin/bash
#SBATCH --nodes=1
#SBATCH --ntasks-per-node=1
#SBATCH --cpus-per-task=32
#SBATCH --gres=gpu:1
#SBATCH --time=24:00:00
#SBATCH --exclusive
#SBATCH --account=test_acc
#SBATCH --partition=boost_usr_prod
#SBATCH --qos=normal

module purge

export OMP_NUM_THREADS=1

scontrol show jobid -dd $SLURM_JOB_ID > scontrol.out

##


 mpirun -np 1 --map-by socket:PE=32 --rank-by core  bigdft
```

Figure 11: Interaction of the JUBEInterop module with JUBE specified platform.xml and submit.job template files. `accountno` was set to `test_acc`, `executable` to `bigdft` and `args_executable` to an empty string. These lines have been omitted and the output clipped for brevity.

example is setting the user quality of service (qos) if their node specification is outside of the default. A great deal of care was taken to ensure this feature is not broken when translated from JUBE to `remotemanager` format. See figure 12 for an example output.

References

[1] First report on max software architecture and implementation planning. URL https://www.max-centre.eu/sites/default/files/D1.1_First%20report%20on%20MAX%20software%20architecture%20and%20implementation%20planning_compressed.pdf.

[2] First release of max software: report on performed and planned refactoring. URL https://www.max-centre.eu/sites/default/files/D1.2_.First%20release%20of%20MAX%20software_report%20on%20performed%20and%20planned%20refactoring%20%281%29.pdf.

[3] Davis, J. H. *et al.* Taking GPU programming models to task for performance portability (2024). 2402.08950.

```
[5]: # qos should change to boost_qos_bprod if nodes>64
     test.nodes = 128

     # mapping and args_starter depend on the starter parameter
     test.starter = "srun"

[7]: # RemoteManager compatible script generation
     print(test.script())

#!/bin/bash
#SBATCH --nodes=128
#SBATCH --ntasks-per-node=1
#SBATCH --cpus-per-task=32
#SBATCH --gres=gpu:1
#SBATCH --time=24:00:00
#SBATCH --exclusive
#SBATCH --account=test_acc
#SBATCH --partition=boost_usr_prod
#SBATCH --qos=boost_qos_bprod

...

srun -n 128 --cpu-bind=cores --cpus-per-task=$SLURM_CPUS_PER_TASK  bigdft
```

Figure 12: Recreating the previous script, with some changes to the input parameters. This has set our `qos` to `boost_qos_bprod`, and our run line is now using the `srun` format. The ellipses indicate unchanged output that has been removed.

[4] OpenMP Architecture Review Board. OpenMP application program interface version 5.1 (2020). URL https://www.openmp.org/wp-content/uploads/OpenMP-API-Specification-5-1.pdf.

[5] Omp offload. URL https://gitlab.com/QEF/q-e/-/tree/develop_omp5?ref_type=heads.

[6] Carnimeo, I. *et al.* Quantum espresso: One further step toward the exascale. *Journal of Chemical Theory and Computation* **19**, 6992–7006 (2023).

[7] Wagner, M. *et al.* Performance analysis and optimization of the fftxlib on the intel knights landing architecture. In *2017 46TH International Conference on Parallel Processing Workshops (ICPPW)*, International Conference on Parallel Processing Workshops, 243–250 (CRAY Supercomp Company; Intel; ARM; OCF; SGI; Univ Bristol; VirginiaTech; Int Assoc Comp & Communications, 2017). 46th International Conference on Parallel Processing Workshops (ICPPW), Bristol, England, AUG 14-17, 2017.

[8] Giannozzi, P. *et al.* QUANTUM ESPRESSO toward the exascale. *J. Chem. Phys.* **152**, 1–11 (2020).

[9] Ferrari Ruffino, F. *et al.* QUANTUM ESPRESSO towards performance portability: GPU offload with OpenMP. *Proceedings of the First EuroHPC user day (submitted)* (2024).

[10] Genovese, L. *et al.* Density functional theory calculation on many-cores hybrid central processing unit-graphic processing unit architectures. *The Journal of Chemical Physics* **131**, 034103 (2009).

[11] NVIDIA Corporation. CUDA toolkit (2023). URL https://developer.nvidia.com/cuda-toolkit.

[12] NVIDIA Corporation. cuFFT (2023). URL https://docs.nvidia.com/cuda/cufft/index.html.

[13] Ratcliff, L. E., Degomme, A., Flores-Livas, J. A., Goedecker, S. & Genovese, L. Affordable and accurate large-scale hybrid-functional calculations on gpu-accelerated supercomputers. *Journal of Physics: Condensed Matter* **30**, 095901 (2018).

[14] Advanced Micro Devices, Inc. AMD Instinct GPUs (2023). URL https://www.amd.com/en/graphics/instinct-server-accelerators.

[15] Intel Corporation. Intel Max Series GPUs (2023). URL https://www.intel.com/content/www/us/en/products/details/discrete-gpus/data-center-gpu/max-series.html.

[16] Oak Ridge National Laboratory. Frontier Supercomputer (2023). URL https://www.olcf.ornl.gov/frontier/.

[17] Argonne Leadership Computing Facility. Aurora Supercomputer (2023). URL https://www.alcf.anl.gov/aurora.

[18] Intel Corporation. DPC++ compatibility tool (2023). URL https://www.intel.com/content/www/us/en/developer/tools/oneapi/dpc-compatibility-tool.html.

[19] Dugan, N., Genovese, L. & Goedecker, S. A customized 3d gpu poisson solver for free boundary conditions. *Computer Physics Communications* **184**, 1815–1820 (2013).

[20] Carsten Uphoff. double-batched FFT library (2023). URL https://github.com/intel/double-batched-fft-library.

[21] Intel Corporation. Intel oneAPI Math Kernel Library (2023). URL https://www.intel.com/content/www/us/en/docs/oneapi/programming-guide/2023-2/intel-oneapi-math-kernel-library-onemkl.html.

[22] Intel Corporation. Intel oneAPI (2023). URL https://www.intel.com/content/www/us/en/developer/tools/oneapi/overview.html.

[23] NVIDIA Corporation. cuFFTMp (2023). URL https://docs.nvidia.com/hpc-sdk/cufftmp/index.html.