D4.4

First report on co-design actions

Fabio Affinito, Pietro Bonfà, Pietro D. Delugas, Federico Ficarelli, Alberto Garcia, Conrad Hillairet, Anton Kozhevnikov, Fabrizio Magugliani, Simone Tinti, Simone Marocchi, Giacomo Rossi, and Carlo Cavazzoni

Due date of deliverable:	31/05/2020
Actual submission date:	31/05/2020
Final version:	31/05/2020
Revised version:	19/10/2020
Revised version submission:	19/02/2021

Lead beneficiary:	CINECA (participant number 8)
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:	www.max-centre.eu
Deliverable No.:	D4.4

Authors: F. Affinito, P. Bonfà, P. D. Delugas, F. Ficarelli, A. Garcia, C. Hillairet, A. Kozhevnikov, F. Magugliani, S. Marocchi, G. Rossi, S. Tinti, and C. Cavazzoni

To be cited as: F. Affinito et al. (2020): First report on co-design actions. Deliverable D4.4 of the H2020 project MAX (final version as of 29/05/2020). EC grant agreement no: 824143, CINECA, Casalecchio di Reno (BO), Italy.

Disclaimer:

D4.4 First report on co-design actions

Content

Changes and contribution history

	Change	Author	Notes
1	Section 8.2.1	F. Affinito, D. Cesarini	Improved explanation of CCPI and low levels of efficiency.
2	New chapter 10: "Co-design feedback to HPC technology providers"	F. Affinito, F. Magugliani, A. Kozhevnikov, S. Mohr, A. Ferretti	Improved explanation of the co-design cycle, the role of MaX in the procurement of HPC pre-exascale systems and the plans for the future.
3	Section 11: "Conclusions"	F. Affinito	Description of the role of MaX in the co-design cycle on the software side, considering also the interactions with the HPC technology providers.

1 Executive Summary

The MAX codes are widely used all over the world, from personal workstations to high-end supercomputers, and quite often these codes are at the top in resource usage. They receive attention both from HW vendors and HPC centre experts: as a consequence, the opportunities for co-design are more than those we can afford with MAX WP4 resources. We thus decided to be selective and concentrate on those serving the strategic objective of MAX and EuroHPC. First of all, we focus on three main classes of co-design actions: programming paradigms, hardware co-design, and math libraries co-design. Secondly, we give priority to what is expected to be the EuroHPC ecosystem, with a particular focus on EPI (European Processor) characteristics, so that the MAX codes will be ready to run on future European processors.

For the near term (pre-exascale EuroHPC systems) it was of fundamental importance to continue with the work on programming paradigms for accelerators, as these devices will be present in the majority of the EuroHPC systems. At the same time, we invested significant WP4 resources to explore co-design opportunities on ARM architecture, with SVE (Scalable Vector Extension) instructions, since these represent the main feature of the EPI processor. In this respect it was important to evaluate ARM+GPU and ARM+SVE, as we have run PoC with NVIDIA (GPU side) and Marvell (ARM thunder X2), Fujitsu for ARM+SVE, and ARM itself with CPU simulator, emulator and profiler. So far we are not able to disclose all the details of this activity, since at the time of writing many HW features are still not public, or at least results of real tests cannot be disclosed yet.

In general, co-design actions allow the WP4 team to perform good explorative actions and provide feedback to the application developers about how to refactor and rewrite applications. On the software side (see also deliverable D4.3), one of the most important challenges to the MaX codes is represented by finding a good GPU-aware substitute of the ScaLAPACK library (distributed linear algebra solvers). In this respect, a number of options are under evaluation, as also detailed in the present deliverable.

2 Introduction

This document reports about the activities of the different proof-of-concepts performed on MAX codes for the selected co-design actions. In particular, we consider several examples of co-design targeting the utilisation of accelerator devices, discussing proof-of-concepts arising from different programming paradigms (CudaFortran, OpenACC, OpenMP) or through the adoption of libraries (ELPA, HPX). Furthermore, we explore new computing architectures: in particular the porting to emulator of ARM SVE instructions and the benchmarking on the ARMIDA cluster are discussed in detail, showing some use cases coming from the MAX flagship codes. Finally, an example of exploitation of new non-volatile

memory, such as the Intel Optane "Apache Pass", is discussed, giving an opportunity for future utilisation in high-end computing facilities.

3 Overview of GPU-oriented paradigms

One of the clear trends in this technological era is the affirmation of the hybrid paradigm in the HPC facilities. The limits in the density of silicon chips and the necessity to deal with economically sustainable systems inevitably led to the success of systems accelerated with GPUs. Such systems are, with some notable exceptions, driving the HPC market and dominating the Top500 lists in the last months. One of the most important and difficult challenges consists in bringing the HPC software to a readiness level able to get the most advantage from the new CPU+GPU systems. In this respect, the most relevant issue is the absence of a recognized standard programming paradigm. In addition to CUDA, which is a proprietary language of NVIDIA, many options are emerging, with different probabilities of getting rid of the other concurrents. A very similar approach to the standard CUDA, but more useful when dealing with legacy codes, is represented by CUDAFortran. CUDAFortran allows for porting codes to GPUs adopting the same approach of standard CUDA (i.e. through CUDA kernels) and with the same efficiency. Moreover, CUDAFortran offers the possibility of defining "CUF kernels" which permits to identify with directives the portions of the codes to be executed on the GPU. This is similar to the "directive approach" used in OpenACC and OpenMP. The former has been developed in the recent years, but for the moment has been implemented only in the PGI compilers (it was also supported by the CRAY compiler and it has a partial and ongoing implementation in the GNU suite). OpenMP, similarly, has defined a set of directives for external devices from the 4.5 release of the standard. Remarkably, OpenMP5 is part of oneAPI, the programming approach adopted by Intel for their next generation of devices (PonteVecchio). In OneAPI, together with OpenMP, there is also another possible choice: DPC++, which adopts and develops the standard defined by Khronos SYCL. In the following sections of this deliverable, we will discuss some early experience where we have experimented with some of the above described approaches in the context of the MAX flagship codes and related libraries.

4 OpenMP tasking and offload: the Intel software stack

4.1 Intel oneAPI and PVC

Intel oneAPI is a unified and cross-architecture programming model that permits to deploy applications and solutions across CPUs, GPUs, and FPGAs, simplifying programming. The core of oneAPI is DPC++, an open and cross-architecture language built upon the ISO C++ and Khronos SYCL standards; DPC++ extends these standards and provides explicit parallel constructs and offload interfaces, to support a broad range of computing architectures. DPC++ allows for code reuse across hardware targets, while permitting custom tuning.

Moreover, oneAPI supports Fortran and C codes on accelerators, using directive-based programming style by means of the OpenMP API: the support for accelerators has been introduced in OpenMP from version 4.0. Current OpenMP API 5.0 specifications introduced support for unified shared memory between host and devices.

Intel announced also its flagship discrete GPU, codename Ponte Vecchio, based on the forthcoming Xe architecture: Ponte Vecchio will be manufactured on Intel's 7nm technology and will leverage Intel's Foveros 3D and EMIB - Embedded Multi-die Interconnect Bridge packaging, including high-bandwidth memory.

4.2 Prototype with tasking in QE FFT: leveraging on granularity tasks

The current configuration of FFTXlib in Quantum ESPRESSO provides two main procedures for FFTs evaluation: the simplest one relies on "scalar" routines, that use vendor specific drivers, as Intel DFTI from MKL, IBM ESSL DCFFT, and ARM ZFFT1MX, or the generic driver from the FFTW library; inside these routines, 1D, 2D and 3D FFTs are implemented, so it is possible to perform 1D FFT along X direction, then 1D FFT along Y direction, followed by 1D FFT along Z direction, or a single 2D FFT in the XY plane followed by a 1D FFT along Z direction, or a single 3D FFT.

When the number of processors exceeds the number of FFT planes available, Task Groups (TGs) are used: FFT data is redistributed to all the task groups, so that each group can process several wave functions at the same time. TGs rely on the scalar 1D FFTs, because any task group performs three 1D FFTs in a row.

The Task Groups algorithm is the following:

- FFT along Z
- scatter Z -> Y
- FFT along Y
- scatter Y -> X

- FFT along X

for the reciprocal space to real space case.

In order to work properly, TGs need dedicated routines and data management, so the whole code is more verbose and during FFTXlib development or modification, both standard and TGs approaches need to be changed.

To overcome this limitation, a new approach has been developed, based on OpenMP Tasks. Tasking concept has been introduced in OpenMP 4.0 and extended in OpenMP 4.5, where taskloop and taskloop simd constructs have been introduced. OpenMP 5.0 introduced task reductions, detachable tasks and task-to-data affinity. Using OpenMP tasks, it is possible to easily parallelize algorithms with an irregular and runtime dependent execution flow.

The new approach, called "manyFFTs", is based on FFT batching: the number of FFTs is increased (and consequently, their size is decreased), so each processor can perform its own FFT; moreover, it is possible to introduce an overlap between computation and communication. The number of FFT planes available does not represent anymore a limitation in the number of parallel processes that can be launched: the FFTs are grouped together, so at the same time a total number of (MPI processes times FFTs bands) FFTs can be performed.

In this context, the use of OpenMP Tasks is really helpful, because the parallelization of the FFT execution through the different bands becomes pretty straightforward: using taskloop, each generated task will perform an FFT on a different band; the number of bands should be larger than the number of OpenMP threads, in order to take full advantage of the band decomposition: otherwise, there will be inactive OpenMP threads.

The algorithm is not different from the Task Group one, just slightly modified: now there are bands FFTs along each direction, and the fft_scatter_yz routine has been replaced by fft_scatter_many_yz routine, that takes into account the different data layout derived from FFT batching. The code, in the reciprocal to real space case, is reported in the following:

```
!$omp parallel default(none)                                        &
!$omp          private(i, j)                                        &
!$omp          shared(howmany, f, nnr_, isgn, dfft)   &
!$omp          shared(nsticks_z, n3, nx3)   &
!$omp          shared(nsticks_y, n2, nx2)   &
!$omp          shared(nsticks_x, n1, nx1)   &
!$omp          shared(nsticks_zx, nsticks_yx)
     !
!$omp do
     DO i = 0, howmany-1
        CALL cft_1z( f(i*nnr_+1:), nsticks_z, n3, nx3, isgn, dfft%aux(nx3*nsticks_zx*i+1:) )
     ENDDO
!$omp end do
     !
!$omp single
     CALL fft_scatter_many_yz( dfft, dfft%aux, f, isgn, howmany )
     !
     DO i = 0, howmany-1
!$omp task depend(out:f(i*nnr_+1:(i+1)*nnr_))
        CALL cft_1z( f(i*nnr_+1:), nsticks_y, n2, nx2, isgn, dfft%aux(i*nnr_+1:), in_place=.true. )
!$omp end task
!$omp task depend(in:f(i*nnr_+1:(i+1)*nnr_)) depend(out:dfft%aux(i*nnr_+1:(i+1)*nnr_))
        CALL fft_scatter_xy( dfft, f(i*nnr_+1:), dfft%aux(i*nnr_+1:), nnr_, isgn )
!$omp end task
!$omp task depend(in:dfft%aux(i*nnr_+1:(i+1)*nnr_))
        CALL cft_1z( dfft%aux(i*nnr_+1:), nsticks_x, n1, nx1, isgn, f(i*nnr_+1:) )
!$omp end task
!$omp task depend(in:f(i*nnr_+1:(i+1)*nnr_))
        if (nsticks_x*nx1 < nnr_) then
           do j=nsticks_x*nx1+1, nnr_
               f(j+i*nnr_) = (0.0_DP,0.0_DP)
           end do
        endif
!$omp end task
     ENDDO
!$omp end single
!$omp end parallel
```

Using inplace FFT along Y direction, any FFT band can be processed independently from the others, maximizing concurrency.

Unfortunately, OpenMP Tasks are not yet fully supported by the GNU compiler suite or PGI compiler, so in order to ensure the broadest compatibility, the code above has been modified:

```
!$omp parallel default(none)                                        &
!$omp          private(i, j)                                        &
!$omp          shared(howmany, f, nnr_, nsticks_z, n3, nx3, isgn)   &
!$omp          shared(nsticks_zx, dfft)
        !
!$omp do
        DO i = 0, howmany-1
           DO j=1, nsticks_z*nx3
              dfft%aux(j+i*nnr_) = f(j+i*nnr_)
           ENDDO
        ENDDO
!$omp end do
        !
!$omp do
        DO i = 0, howmany-1
           CALL cft_1z( dfft%aux(i*nnr_+1:), nsticks_z, n3, nx3, isgn, f(nx3*nsticks_zx*i+1:) )
        ENDDO
!$omp end do
!$omp end parallel
        !
        CALL fft_scatter_many_yz( dfft, f, dfft%aux, isgn, howmany )
        !
!$omp parallel default(none)                                        &
!$omp          private(i)                                           &
!$omp          shared(howmany, f, nnr_, isgn, nsticks_y, n2, nx2)   &
!$omp          shared(nsticks_yx, dfft)
!$omp do
        DO i = 0, howmany-1
           CALL cft_1z( dfft%aux(i*nnr_+1:), nsticks_y, n2, nx2, isgn, f(nx2*nsticks_yx*i+1:) )
        ENDDO
```

```
!$omp end do
!$omp end parallel
        !
        CALL fft_scatter_many_xy ( dfft, f, dfft%aux, isgn, howmany )
        !
!$omp parallel default(none)                                        &
!$omp          private(i, j)                                        &
!$omp          shared(howmany, f, nnr_, isgn, nsticks_x, n1, nx1)   &
!$omp          shared(dfft)
!$omp do
        DO i = 0, howmany-1
           CALL cft_1z( dfft%aux(i*nnr_+1:), nsticks_x, n1, nx1, isgn, f(i*nnr_+1:) )
        ENDDO
!$omp end do
        !
!$omp do
        DO i = 0, howmany-1
           if (nsticks_x*nx1 < nnr_) then
              do j=nsticks_x*nx1+1, nnr_
                  f(j+i*nnr_) = (0.0_DP,0.0_DP)
              end do
           endif
        END DO
!$omp end do
!$omp end parallel
```

The manyFFTs algorithm has been tested against TaskGroups using the FFT miniapp on a single node, dual socket, Intel Cascade Lake 8260L (24 cores @ 2.4GHz), equipped with 192GB RAM @ 2666 MHz, varying the number of MPI tasks and OpenMP threads accordingly, in order to keep all 48 cores occupied; the number of task groups and FFT bands varied from 1 to 32.

In Fig. 1 are reported the lowest FFTW wall times for 6, 12 and 24 MPI tasks (8, 4 and 2 OpenMP threads respectively): when the number of MPI tasks increases, manyFFTs performed better than Task Groups.

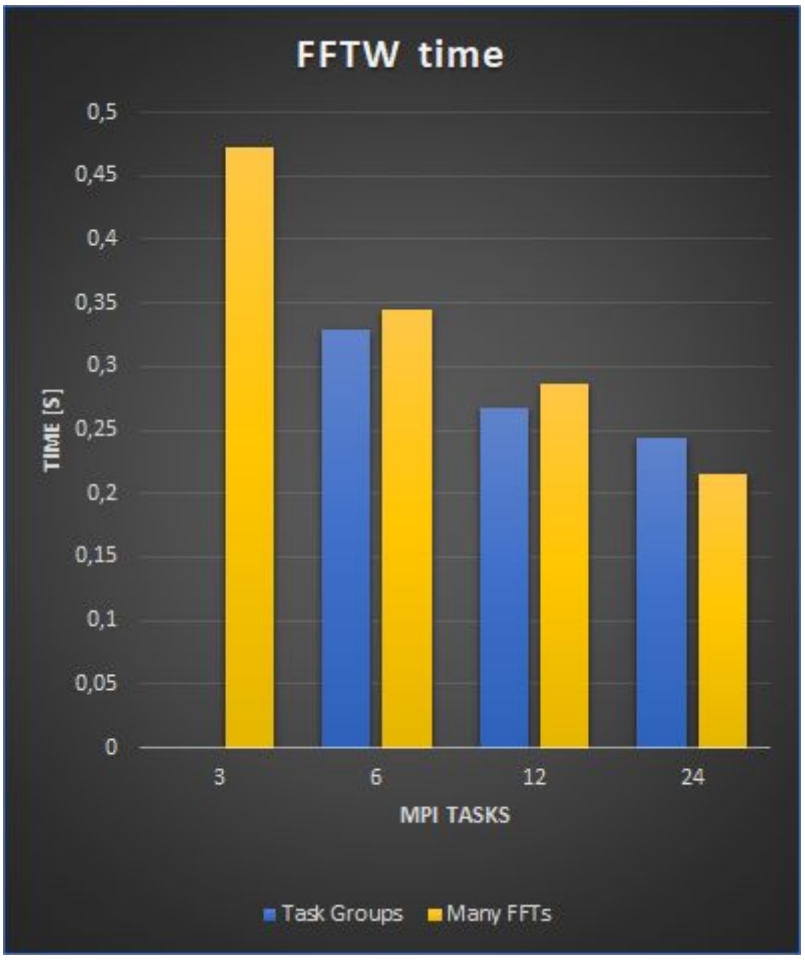

Figure 1: lowest FFTW wall times for 6, 12 and 24 MPI tasks (8, 4 and 2 OpenMP threads respectively): when the number of MPI tasks increases, manyFFTs perform better than Task Groups.

A second test (Fig.2), using the same system but on two nodes, has been performed using the DyOtBuClTHF_100K test case, using 48 MPI tasks and varying the number of OpenMP threads: in this case, the manyFFTs implementation performs always better than Task Groups.

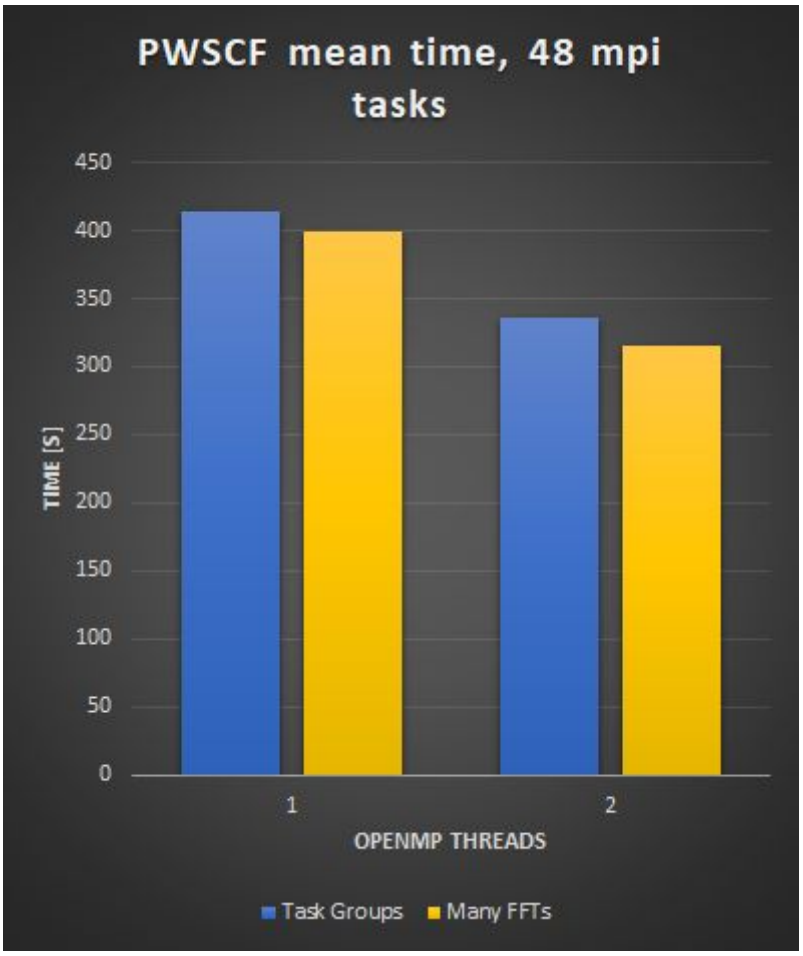

Figure 2: DyOtBuClTHF_100K test case, using 48 MPI tasks and varying the number of OpenMP threads

OpenMP introduced support for accelerators (devices) from version 4.0: a device, in OpenMP syntax, is a computational resource where a region of code can execute; there's no difference between CPUs, GPUs, FPGAs or other specialized processors. The pragma omp target/omp target directive defines the region that has to be executed on the device.

When a host thread encounters a target construct, the target region is executed by a new thread running on the accelerator: before the new thread starts the execution, the device data environment is created for the marked region and the marked code region is mapped to the device; after that, the region is executed.

The OpenMP heterogeneous execution model is host-centric: the program starts running on the host device; each target region acts as an OpenMP subprogram where an initial thread begins the execution of the target region; the initial thread may encounter other parallel constructs and spawn teams of threads. By default, the thread that encounters the target

construct waits for the execution of the target region to complete and then continues to execute the code after the target construct.

The target construct is a task generating construct: when a thread encounters a target construct, it generates an explicit task that manages the execution of the target region, the so-called target task; this task is generated on the host and when executes, the target region executes in the context of an implicit task, called initial task, on the accelerator.

OpenMP supports heterogeneous architectures with both distributed and shared memory by mapping variables from the host to an accelerator. When host and accelerator don't share memory, a mapped variable is copied from host memory into accelerator memory.

The following code snippet shows a portion of the FFTXlib scatter_mod.f90 file:

```
     DO iproc2 = 1, nproc2
        it0 = ( iproc2 - 1 ) * sendsize
        j_end = nr1p_(iproc2) * desc%my_nr3p
        nr1p_2 = nr1p_(iproc2)
!$omp target teams
!$omp distribute parallel do collapse(3)
        DO k=0, howmany-1
           DO j = 1, j_end
                 DO i = 1, my_nr2p
                    it = it0 + (j - 1) * nr2px + k * ncpx * nr2px
                    m2 = (j - 1) / nr1p_2 + 1
                    i1 = mod((j - 1),nr1p_2)+1
                    m1 = indx(i1,iproc2)
                    icompact = m1 + (m2 - 1) * nr1x * my_nr2p + nr1x * (i - 1)
                    f_in( i + it ) = f_aux( icompact + k*nnr )
                 ENDDO
              ENDDO
           ENDDO
!$omp end distribute parallel do
!$omp end target teams
        ENDDO
     IF( nproc2 == 1 ) GO TO 20
     !
     !
     !  step two: communication
     !
!$omp target update from (f_in)
     CALL mpi_alltoall (f_in(1), sendsize, MPI_DOUBLE_COMPLEX, f_aux(1), &
                        sendsize, MPI_DOUBLE_COMPLEX, desc%comm2, ierr)

     IF( abs(ierr) /= 0 ) CALL fftx_error__ ('fft_scatter', 'info<>0', abs(ierr) )
     !
     !  step one: store contiguously the columns
     !
!$omp target update to (f_aux)
     DO iproc2 = 1, nproc2
```

```
        j_end = nr1p_(me2) * desc%my_nr3p - 1
        i_end = desc%nr2p( iproc2 )
        kdest = ( iproc2 - 1 ) * sendsize
        kfrom = desc%nr2p_offset(iproc2)
!$omp target teams
!$omp distribute parallel do collapse(3)
        DO k = 0, howmany-1
           DO j = 0, j_end
              DO i = 1, i_end
                 f_in ( kfrom + nr2x * (j + k*ncpx) + i ) =  f_aux ( kdest + nr2px * (j + k*ncpx) +
i )
              ENDDO
           ENDDO
        ENDDO
!$omp end distribute parallel do
!$omp end target teams
     ENDDO
     ! clean extra array elements in each stick
     IF( nr2x /= desc%nr2 ) THEN
        j_end = nr1p_(me2)*desc%my_nr3p
!$omp target teams
!$omp distribute parallel do collapse(3)
        DO k = 0, howmany-1
           DO j = 1, j_end
              DO i = desc%nr2+1, nr2x
                 f_in( k*ncpx*nr2x + (j-1)*nr2x + i) = 0.0d0
              END DO
           END DO
        ENDDO
!$omp end distribute parallel do
!$omp end target teams
     ENDIF
```

The entire QE FFT library (FFTXlib) has been ported to Fortran OpenMP offloading, with full functionality and has successfully passed all the tests; the port will be publicly released soon.

From this code snippet, it is possible to see some fundamental OpenMP offloading constructs: the target construct, that is often joined to the teams construct, in order to generate, on the device, a league of teams; each of these teams is a single initial thread executing in parallel the subsequent code statement.

The distribute parallel do construct is a composite accelerated worksharing construct that distributes the iterations of a loop across two levels of parallelism: the loop iterations are divided into chunks, which are first distributed to the initial threads; the subset of loop iterations assigned to the initial (master) thread are then again distributed to the threads in the team.

The target update construct is a data-mapping construct: its role is to make the value of a mapped variable the same on the host and on the accelerator. It either assigns the value of the host original variable to the accelerator corresponding variable - omp target update (to: variable) or vice-versa - omp target update (from: variable).

4.3 DevXlib: a language agnostic library for GPU acceleration

The many diverse approaches presented in the previous section reflect the plethora of technological solutions currently proposed in the market as well as the lack of a more definite roadmap towards any transparent standard which may be confidently adopted.

In order to efficiently work on these different threads we have nonetheless to rely on the expectation that all of these approaches may be made accessible to high level programmers through a unique set of interfaces. For this reason, this task has collaborated with other work packages (WP1, WP2, WP3) to design an agnostic data manager intended to provide the programmer with APIs for: (i) memory handling; (ii) perform basic linear algebra; (iii) management and synchronization of multiple address spaces; (iv) mask and optimise data movement between host and device. Such agnostic data manager will hide the diverse implementations adopted by different backends and will be distributed as an autonomous MAX library called DevXlib (see also D1.3). The APIs as well as the specific implementations are at the moment in a developmental stage and are already available for some backends on MAX-Components GitLab repository[1].

Although at this early stage, the more mature features provided by this library are important key resources for the CUDA GPU porting of Quantum ESPRESSO and Yambo using the CUDA-Fortran backend. As demonstrated below in the section about the cp.x porting to GPU the reuse of such features streamlines significantly the porting of the code to new backends.

One of the key actions recently started is the development of the agnostic manager for multiple separated address spaces. The current, experimental version features a Fortran data type that collects the replica of the data in a single endpoint and provides a Fortran pointer, pointing either to the data on the GPU memory or the RAM, as specified by the developer. During pointer assignment, synchronization is automatically performed, but only if data residing on the RAM and on the GPU memory are found to be out of sync. This optimizes data movement operations by avoiding redundant copies.

The PoF, written in Fortran and adopting either OpenMP, OpenACC or CUDA Fortran programming languages for data offloading, has been successfully compiled and run with GNU and PGI implementations of OpenACC and OpenMP 4.5, and with PGI's implementation of CUDA Fortran. In addition, the library was used to port a few subroutines of the pwscf code from the Quantum ESPRESSO suite in order to investigate the advantages and the

[1] https://gitlab.com/max-centre/components/devicexlib

limitations of this approach, which imposes restrictions on memory accesses and data slicing with respect to the standard Fortran syntax.

```
program main
  use offloadable, only : data_flags, offloadable_complex,offloadable_real, &
                          allocate_offloadable, deallocate_offloadable, system_descriptor, dp,
                          initialize_ptr, associate_ptr
  use dxl_blas, only : dxl_ZGEMM

  implicit none
  type(offloadable_complex) :: a, b, c
  integer, parameter :: n =10
  type(data_flags) :: desc
  complex(dp), pointer, contiguous :: ptr(:,:)
  complex(dp), pointer, contiguous :: ptr_d(:,:)
#if defined(__CUDA)
  attributes(device) :: ptr_d
#endif
  integer :: i, j
  desc = system_descriptor(.true.)

  ! Allocate a 10x10 matrix on both CPU and GPU
  CALL allocate_offloadable(a, [ n, n ], desc)
  CALL allocate_offloadable(b, [ n, n ], desc)
  CALL allocate_offloadable(c, [ n, n ], desc)

  ! Associate pointer ptr to allocatable 'a', for writing.
  CALL initialize_ptr(a, 'H', ptr, desc)
  ptr(:,:)=(1.d0,1.d0)

  ! Now associate pointer ptr to allocatable 'b', for writing.
  CALL initialize_ptr(b, 'H', ptr, desc)
  ptr(:,:)=(1.d0,2.d0)

  CALL dxl_ZGEMM('n','n',n,n,n,(1.0d0,0.d0),a,n,b,n,(0.d0, 0.d0),c,n, desc)

  ! Associate ptr to allocatable 'c' to read the result of zgemm.
  CALL associate_ptr( c, 'roh', ptr, desc)
  !
  print *, "ZGEMM on GPU gives: "
  print *, ptr
  !
  CALL associate_ptr( c, 'rwd', ptr_d, desc)

ACC data present(ptr_d) if(desc%use_gpu)
ACC parallel loop if(desc%use_gpu)
OMP target data map(tofrom:ptr_d)
OMP target teams
OMP distribute parallel do simd collapse(2)
```

```
CUF kernel do
  do i=1,10
    do j=1,10
      ptr_d(j,i) = 2.d0*ptr_d(j,i)
    end do
  end do
ACC end parallel
ACC end data
OMP end distribute parallel do simd
OMP end target teams
OMP end target data


  ! Associate ptr to allocatable 'c' to read the result of OpenACC kernel.
  CALL associate_ptr( c, 'roh', ptr, desc)
  !
  print *, "2x previous result"
  print *, ptr

  CALL deallocate_offloadable(a, desc)
  CALL deallocate_offloadable(b, desc)
  CALL deallocate_offloadable(c, desc)

end program main
```

Figure 3: A code snippet showing the usage of the Fortran based data manager with different directive based programming paradigms for GPU computation offloading.

5 Strategies for integrating GPU and CPU code in a single source tree: porting the Car-Parrinello kernel of Quantum ESPRESSO to CudaFortran

Together with open standards OpenACC, and OpenMP, another option for Fortran codes like Quantum ESPRESSO (QE) to be ported on heterogeneous architecture, is to use vendor specific language extensions. In particular for NVIDIA accelerator, the language extension is called CUDAFortran.

The goal of this activity is twofold: on one side, the porting to GPUs of the Car-Parrinello kernel of QE using CUDAFortran (in order to complete the porting of the QE main kernels); on the other side, the opportunity of field testing the strategy to integrate - in the main trunk of QE - all the work done so far in terms of libraries, functionalities, and modules for GPUs.

The CUDAFortran extension was made available only recently, and actually QE was one of the first codes to adopt it. Indeed, QE was adopted by NVIDIA itself as a test platform to develop the language extension, the compiler, and some libraries, like the eigenvalue solver.

Within MAX, also thanks to proof-of-concept and activities of this co-design work package, we develop the strategy to encapsulate functionalities, factor out libraries to be made independent from the rest of the code, and develop layers of software to implement high level APIs to hide self contained code components that may become hardware specific.

The two first libraries we identified and actually produced were LAXlib and FFTXlib. These two were also the first two components to be ported in CUDAFortran and actually becoming the first target for any new porting, as happened for OpenMP5 and ARM SVE capable processors. Other libraries have followed, and in particular one with GPU specific wrappers and helper subroutines, called DevXlib, with the aim of hiding, from the main kernel codes, GPU specific syntax constructs and functions. DevXlib is still under development and improvement, and already used by different MAX codes, like QE and Yambo.

Regarding QE, this is not a monolithic code, it is rather a suite of codes compiled and built in different executables and libraries, sharing as much code as possible. The QE code base is huge and the porting to GPUs has been implemented starting from the low level components, as summarised above, and then propagated up to the main kernels. The first kernel to be ported has been PWscf, already released for GPU in a separate development branch, but kept synchronised with the main trunk of QE.

The next step will be to move forward and start the integration of the GPU branch in the main trunk to have a single source tree, which represents a fundamental milestone in the evolution of the code towards heterogeneous architectures and exascale.

The porting of the Car-Parrinello (CP) kernel of QE has given us also the opportunity to test the maturity of the GPU branch, and to demonstrate the readiness for a full integration of all the porting efforts in the main trunk.

In what follows, we report the steps after the porting of CP (all the details can be easily recovered from the Gitlab[2]).

Firstly, we studied the best candidate to start with. The obvious option was the FFTXlib loops, since these are implemented in almost the same way both in PWscf and CP, but we instead chose the orthogonalization set of subroutines, since they as well are based on a common library (LAXlib), but they are not present in PWscf, so they gave us also the best opportunity to experiment the implementation of a single source code for both CPU and GPU.

One after the other, we moved all the data structures involved in the orthogonalization to the GPU, at the beginning by splitting the subroutines in two, a CPU and a GPU version. When all data were ported, they could be either all on GPU or all on CPU (for a CPU build). At this point, the need for splitting the subroutines disappeared, and we then were able to collapse all the code for both CPU and GPU in a single tree. The only difference is given by the type of pointer from CPU and GPU. (See for example the subroutine “ortho_iterate” in the above mentioned repository.[3])

This routine executes the same procedure for GPUs and CPUs, and the code is quite lean and clear. This is the kind of results we would like to obtain on the final convergent source tree. Actually, we were quite pleased that all the efforts and the strategies we developed so far for the porting finally led us to this important result.

After orthogonalization, we ported the FFTXlib loops, and everything worked fine, as it required only changing the pointer from host to device and implementing the data remapping on the GPU. This was expected because it was already extensively tested on PWscf. Even in this case the GPU code is lean and the source code can be merged with the CPU code without adding duplication and exceptions. At present, some low level drivers are still present in two versions (GPU and not CPU), but it will not be an issue to merge them, just a matter of adding few data manipulation wrappers, masking GPU and CPU code.

Then we moved the subroutines handling the pseudopotentials projectors. This part of the code was already containing unnecessary duplication of the functionalities, so when we started adding GPU code, at the beginning the duplication worsened, with a lot of subroutines doing similar things for CPU and GPU. Reducing all this duplication and ending up with a set of subroutines smaller than before the porting cost quite some time but was eventually achieved. Note that the code base of QE is really huge and some parts date back

[2] https://gitlab.com/QEF/q-e-gpu/-/tree/cp-gpu
[3] https://gitlab.com/QEF/q-e-gpu/-/blob/cp-gpu/CPV/src/ortho_base.f90

up to decades, there may be subroutines that have not been reviewed since many years, and may contain less modern code.

Finally we optimised some subroutines left on the CPU that had become a bottleneck with the acceleration of those ported on the GPU. Optimisation here was mostly on the OpenMP parallelization, since usually when running on GPU you need only few MPI tasks on the host (one or two for each GPU), opening up opportunity to use more OpenMP parallelization (especially in many core hosts as those of modern supercomputers).

We stopped porting other functionalities to GPU (for the time being) when the ratio of the performance of CP kernel on GPU nodes versus CPU nodes became comparable to the ratio of the respective peak performances. As an example in Tab. 1 we report the performance of 10 steps of Car-Parrinello molecular dynamics on three different nodes, with the ratio of code performances and peek performances.

Node	Time to solution Tts	ratio of Tts	ratio of Peak Perf
Node BROADWELL (36 core)	51.76s	1	1
Node SKYLAKE (48 core)	27.09s	1.88	2
Node BROADWELL + 1xNVIDIA K80	20.10s	2.57	2.68

Table 1: Car-Parrinello 128 Water molecules benchmark.

In conclusion, with this activity we demonstrate that the strategies adopted so far in porting the code to heterogeneous, GPU-based, architecture were effective in separating concerns. In particular such strategies allowed us for porting in an almost transparent way large high level kernels, and to do this within a single source tree.

6 HPX implementation of tall-and-skinny matrix multiplication

A tall-and-skinny dense matrix-matrix multiplication problem arises in plane-wave DFT codes when we compute the inner product of the wave-functions. As a proof of concept we implemented this compute intensive operation using HPX runtime.

General matrix-multiplication involves matrices of compatible dimensions: $\mathbf{A}_{mk} * \mathbf{B}_{kn} = \mathbf{C}_{mn}$ where **m** is the number of rows of **A** and **C**, **n** is the number of columns of **B** and **C** and **k** is the number of columns of **A** and the number rows of **B**. A tall-and-skinny multiplication (TSM) is characterized by one of **m**, **n** or **k** being much bigger than the other two. Distribution of the matrices among processes and nodes has a significant impact on the performance of TSM. In plane-wave and related electronic structure codes, **A** and **B** are the wave-functions and **C** is the resulting overlap or Hamiltonian matrix. The **k** dimension is related to the coefficients of basis expansion. The **m** and **n** dimensions are related to the electronic Kohn-Sham states. Both **A** and **B** are distributed contiguously along the **k** dimension: each process list of global row indices forms an interval and is strictly increasing. Each interval is of approximately the same length, but is often not exactly equal (the distribution of **k** is constrained by the distribution of **G**-vector sticks of the parallel FFT). If the subsequent diagonalization or Cholesky factorization of the **C** matrix is parallel then **C** has a 2D block-cyclic distribution (i.e. ScaLAPACK compatible). On the other hand, if the subsequent diagonalization of **C** is sequential, the matrix is global to each MPI process. The common case is the former (parallel diagonalization), rather than the latter.

The current implementation of the inner-product kernel in the SIRIUS library for the fully parallel case on CPUs works as follows:

1. Tile-up the process local portions of **A** and **B** with a fixed tile size **ts** (default is 1024) and iterate in column-major order along tiles.

2. Multiply the first **A** and **B** tiles locally into separate **C** tiles.

3. Overlap a non-blocking MPI Allreduce communication on the first **C** tile with a local **AB** multiplication on the second **C** tile.

4. Issue a non-blocking MPI Allreduce for the second **C** tile then wait to complete communication for the first **C** tile and offload the parts needed at the local process.

5. Repeat 2-5 where the first tile is now the second tile and the second tile is the next tile for **A**, **B** and **C** respectively (round-robin).

The GPU implementation is similar, except for the following:

- the round robin process described above happens for 4 tiles at a time instead of 2

- the mechanism to overlap the communication and computation is different, 2 OpenMP threads are used: the first issues local GEMMs and copies data to the host, the second waits for the local GEMMs to complete, issues a blocking MPI Allreduce and offloads the parts needed at the local process.

There are a few observations that can be made regarding the outlined algorithms:

- The communication volume for P number of processes stands at ~O(log(P) * **m'** * **n'**), where **m'** >= **m** and **n'** >= **n** are adjusted for **ts** that doesn't fit **m** and **n** perfectly. The log(P) factor comes from the MPI Allreduce call. The operation communicates more data than needed as each **C** tile stores multiple blocks of the final 2D block-cyclic distribution, most of which belong to other processes (note step 4, only parts of the received data are stored). The log(P) factor can be eliminated by only communicating data to processes that need it.
- In general, overlapping communication and computation brings about at most 2x speedup compared to a non-overlapped version of the algorithm. Notice that in step 3, there are two possible outcomes: a) the local GEMM on the second **C** tile takes longer than the communication of the first **C** tile or b) the opposite is the case and the processes idle waiting for communication to complete. In either case, we have complete communication/computation overlap excluding 1) the local GEMM on the very first tile and 2) the MPI Allreduce on the very last tile. To reduce the overhead of 1) and 2) a smaller **ts** can be used. To improve the overall execution of scenario a), a more efficient local GEMM implementation is needed. That is unlikely as SIRIUS already uses highly-optimized vendor BLAS libraries such as MKL/cuBLAS. To improve scenario b), the time spent communicating must be reduced, that is possible as discussed in the previous bullet point.
- The tile size (**ts**) has to be large enough to ensure the local GEMM call efficiently utilizes all cores, while being small enough to minimize the communication/computation overlap overhead (see previous bullet point). In particular, if **ts** >= **n** and **ts** >= **m** there is no communication/computation overlap.

The current implementation of the algorithm uses a standard fork-join parallel model. An implementation using a task-based parallel model opens up opportunities for more efficient processor utilization by allowing tasks from other SIRIUS kernels to execute while tasks in the SIRIUS inner product kernel are idling. The dependency graphs of algorithms are described in terms of dependencies between operations on tiles of **A**, **B** and **C**. This represents a fine-grained approach to parallel computation.

The goal of the new prototype is to explore efficient implementations of SIRIUS inner product kernel utilizing the task-based parallel framework HPX. HPX is written in modern C++, closely following and contributing to the C++ standardization efforts in the areas of parallelism and concurrency. HPX returns a future object when a task is scheduled. The object is used to build a dependency graph among the tasks which the scheduler uses to order the execution pipeline.

The prototype reduces the communication volume by a factor of log(P) compared to SIRIUS implementation. However, currently, only a CPU version is available.

Tasks represent operations on tiles of **A**, **B** and **C** such as GEMM, offloading, sending, receiving. The tile size (**ts**) is now the same as the block size for **C**. The local GEMM call is no longer parallelized internally using MKL's OpenMP backend, instead, each GEMM task executes a sequential MKL GEMM call and the parallelization is handled via HPX.

The overall structure of the algorithm is as follows where all schedule calls are asynchronous (non-blocking). For each tile in **A**, **B** and **C**:

1. schedule local gemm operation
2. schedule offload and send operation depending on 2.
3. schedule a receive and load operation
4. wait for all receives and sends to complete

Note that step 4 is only needed to benchmark the kernel; in production code, there will be no wait statement: instead, the futures associated with these tasks will be propagated to other kernels.

There are a few variants of the prototype based on how MPI communication is handled. In all variants different threads may make MPI calls, the required thread support is `MPI_THREAD_MULTIPLE`. The send and receive tasks issue matching `MPI_Isend` and `MPI_Irecv` tasks. Each process often issues multiple send/receive calls to each of all other processes. The calls are matched with tags. The variants are as follows:

- **v1** – based on experimental API for handling non-blocking MPI communications in HPX. The API wraps `MPI_Request` objects in futures and integrates well with existing HPX infrastructure. All pending requests from all tasks are collected into a global array and polled for completion with `MPI_Testany` whenever a scheduled task completes or yields.
- **v2** – uses existing facilities to periodically call `MPI_Test` in send and receive tasks and yields the tasks if communication hasn't completed. There are 3 sub variants of `v2`:
  - **v2** – uses the default HPX scheduler for all tasks with all tasks having the same priority
  - **v2_priority** – uses the default HPX scheduler for all tasks but schedules send and receive tasks with high priority
  - **v2_pool** – uses a custom scheduler with a single CPU core dedicated exclusively to send and receive tasks (MPI), gemm tasks execute on the default scheduler with the remaining cores.

Preliminary results of all variants are promising. However, there are certain parameter combinations that pose problems. In particular, if the number of tiles/blocks in **C** is large (i.e.

block size is small compared to the dimensions of the matrix), each process issues a large number of small non-blocking receives which start to slow the underlying MPI library to a crawl. To alleviate the issue, there are two approaches: a) increasing the block/tile size, as this will reduce the number of issued non-blocking communications; b) reducing the available parallelism by introducing dependencies between tasks thus imposing a specific order of execution such that there are no more than a given number (batch size) of outstanding non-blocking communications at a given time. Approach b) is generic, as it can adapt to accommodate the excess of communication tasks, whereas approach a) is easy but may not be available if a specific block size is required by a subsequent algorithm.

Note also that one may use collective operations, such as `MPI_Ireduce` to communicate each tile/block instead of matching multiple `MPI_Isend` and `MPI_Irecv`. However, since `MPI_Ireduce` doesn't have a tag argument, the calls to `MPI_Ireduce` will either have to a) use different communicators, b) execute in the same order across all processes. Approach a) is quite expensive as the number of tiles grows and more communicators have to be created. Approach b) is more palatable but requires introducing even more dependencies among tasks to ensure the order is consistent.

The Fig. 4 graph compares the various approaches to implementing SIRIUS inner product kernel discussed in the previous section. The code was executed on Piz Daint XC40 compute nodes equipped with Intel Xeon E5-2695 (Broadwell (v4), 2 x 18 @ 2.1 GHz). The benchmark used 32 nodes, 2 processes per node, the skinny dimensions **m** and **n** were set to 1000, the tall dimension, **k** was set to 1000000. Each variant was tested with two tile/block sizes: 100 and 200.

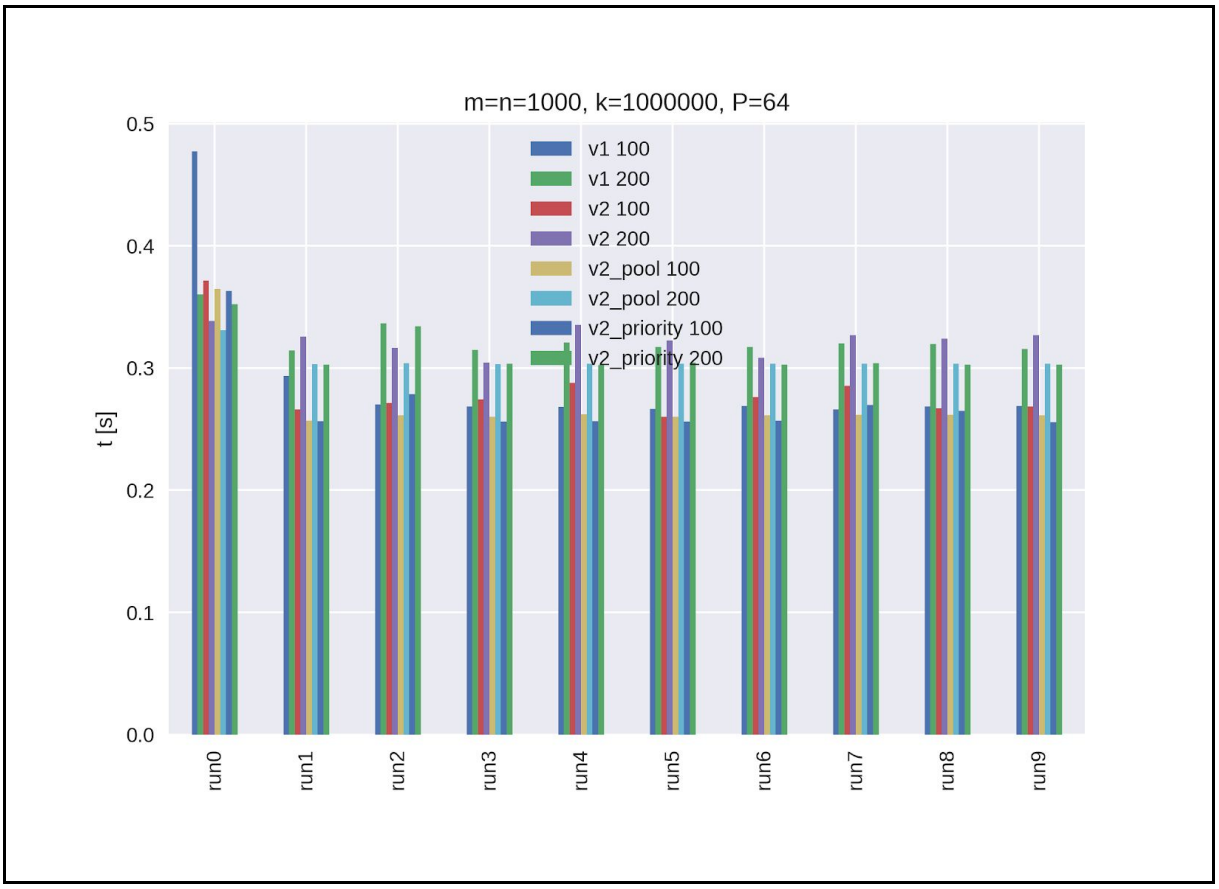

Figure 4: benchmark with Piz Daint 32 nodes, 2 processes per node, the skinny dimensions m and n were set to 1000, the tall dimension, k was set to 1000000. Each variant was tested with two tile/block sizes: 100 and 200.

The 0th iteration is slower due to MKL initialization overhead. The rest of the iterations show that smaller tile/block size is generally beneficial as the available parallelism increases. All variants perform well and no clear winner can be observed in that particular benchmark. Other benchmarks exploring the parameter space suggest slight advantages of **v1** and **v2_priority** over **v2** and **v2_pool**, however a more comprehensive study is needed.

7 Notes on the GPU acceleration of the ELPA library for diagonalization

The ELPA (Eigenvalue SoLvers for Petaflop-Applications) library has had GPU support for a while (for the 'one-stage' flavour of the solver[4]) and has been recently enhanced to add GPU support to the 'two-stage' flavour[5]. These developments are available already in the candidate for the 2020.05.001 release (the ELPA development team makes two releases per year, in May and November), which is in the master branch at https://gitlab.mpcdf.mpg.de/elpa/elpa.

The GPU acceleration is directly available to codes by simply linking to the appropriately compiled version of the ELPA library. This is the case of SIESTA, for which a detailed benchmark, showing sizable speedups, is provided in deliverable D4.3. At present, the implementation in Quantum ESPRESSO is also undergoing.

In view of its potential for performance portability for other MaX codes, we provide here some more details about the optimisations in ELPA.

The 1-stage solver is based on the traditional conversion of a full matrix to tridiagonal form, which is then diagonalized. The final step is the back-transformation of the eigenvectors. In some cases (as in SIESTA), one has to solve a generalized eigenvalue problem, as the basis orbitals are not orthogonal. This implies two extra stages of conversion and back-transformation to and from the simple problem.

The 2-stage solver introduces an extra step: the conversion of the full matrix to banded form, which is then made tri-diagonal. This seems like a roundabout way, but the first step allows for the usage of highly optimized BLAS level 3 functions, so the two-step conversion from full to tridiagonal form is almost always faster. One also needs, however, to transform each eigenvector twice. This makes ELPA 2 competitive when only a small fraction of the eigenvectors are needed. When most or all of the eigenvectors are needed, the best choice might depend on other parameters (matrix size, hardware specifics, etc). GPU acceleration of the 2-stage solver again involves replacing local BLAS calls with the corresponding cuBLAS functions. This is done for the full-to-banded transformation, the solution of the tridiagonal system, and the banded-to-full back-transformation. The tridiagonal-to-banded back-transformation is GPU accelerated by a hand-crafted CUDA implementation of the

[4] P. Kuˇs, A. Marek, S. Koecher, H.-H. Kowalski, C. Carbogno, C. Scheurer, K. Reuter, M. Scheffler, and H. Lederer, "Optimizations of the eigen- solvers in the elpa library," Parallel Comput. 85, 167 – 177 (2019)

[5] Victor Wen-zhe Yu, Jonathan Moussa, Pavel Kůs, Andreas Marek, Peter Messmer, Mina Yoon, Hermann Lederer, Volker Blum, "GPU-Acceleration of the ELPA2 Distributed Eigensolver for Dense Symmetric and Hermitian Eigenproblems,", arXiv:2002.10991 (https://arxiv.org/abs/2002.10991)

corresponding CPU algorithm. The banded-to-tridiagonal transformation has not been ported to GPUs since it has a low computational cost.

For small node counts, GPU-ELPA1 is marginally faster than GPU-ELPA2, which becomes faster than GPU-ELPA1 as the node count increases. The crossover point depends on the size of the problem. In general, GPU speedup is larger for (1) larger matrix size, (2) fewer nodes, and (3) solving a complex problem instead of a real one. Typically, the strong scaling of the GPU solvers is never as good as that of the CPU solvers. When using a large number of nodes or solving a small matrix, the workload on each node becomes so little that the many GPUs cannot be saturated, and the cost of CPU-GPU communications cannot be amortised. In contrast, when solving a large matrix or using a small number of nodes, a large amount of local work is offloaded to the GPUs, resulting in a significant speedup.

The size of the speedup that can be obtained by GPU offloading depends on a number of hardware factors, including the quality of the interconnects and the availability in the CPU of special instructions (e.g., AVX512 vector ops) which can affect the baseline.

Given that the optimal performance that can be obtained with GPU-ELPA1, GPU-ELPA2, or CPU-ELPA2, depends very much on the specifics of the problem and the architecture, users are faced with the complicated task of exploring the best options for their problem. This is a non trivial issue that is going to affect all accelerated codes, and it should be addressed to take full advantage of the porting efforts. In the case of the ELPA library, new versions offer an auto-tuning feature that can alleviate the burden.

Early results of the implementation in SIESTA are presented and discussed in the D4.3, reporting some experience on Marconi100, a Power9+V100 system, where the efficient exploitation of the accelerators in the solution of the eigenvalue problems is crucial. The obtained results are quite promising and suggest the tentative adoption of the ELPA libraries also in other MaX flagship codes (for example Quantum ESPRESSO and FLEUR) where the linear algebra showed to be a significant bottleneck.

8 Exploration of ARM based architectures

Best known for having developed the architecture and instruction sets that power the smartphone world, Arm is today attracting growing interest in the Cloud and HPC spaces. In fact, Arm recently emerged in the landscape of high end HPC solutions for designing the new Post-K Fugaku system at Riken, and implementing the A64FX processor from Fujitsu, which features high core count, advanced SIMD instruction set (SVE), low power consumption and high bandwidth memory with HBM2.

In Europe, the European Commission is considering the Arm Neoverse technology inside the EPI project,[6] which aims at building an independent architecture for the future exascale machines. Within MaX we had the opportunity to test the readiness and the efficiency of the flagship codes with respect to the most recent Arm architectures (with particular attention to the SVE features) and to extract valuable insights about the next technological developments.

8.1 Compilation and porting efforts

The readiness of MaX flagship codes for Arm-based hardware in the coming years is a key milestone to get aligned with the objectives and strategy of the European commission.

To achieve the readiness of MaX flagship codes for Arm-based hardware the porting of the applications can be divided into two main steps:

- the codes compile and run on Arm based CPUs
- the codes are optimised for these platforms and can take advantage of their key features.

We will present the results for the first point in this section and results for the latter in the next section.

We have started to port MAX codes to Arm-based hardware. We focused our efforts on testing the commercial Arm Compiler for Linux (formerly known as the Arm HPC Compiler) as well as the GNU compiler provided by Arm in its Arm Allinea Studio bundle. Depending on the silicon provider other compilers are available in the Arm software ecosystem: Cray compiler and Fujitsu compiler for instance. We will test them when possible.

[6] https://www.european-processor-initiative.eu/project/epi/

For this first report we focused on three applications out of the six MaX flagship codes: BigDFT, CP2K, and Yambo.

8.1.1 BigDFT

The porting of BigDFT to Arm has been relatively smooth so far. Indeed, the code compiles out of box with the GNU 9.2.0 compiler. Whilst it does not with the Arm Compiler for Linux 20.0, workarounds have been found quickly which enabled us to work with both binaries.

The current build recipes have been published online:

https://gitlab.com/arm-hpc/packages/-/wikis/packages/bigdft

We can divide the compilation issues faced with BigDFT into three main categories:

- the compiler crashes when compiling a part of the code;
- the compilation fails due to the compiler behaviour being slightly different wrt the one of the GNU compilers;
- the produced binary does not give correct results.

We have isolated the issues and made reproducers whenever possible to file them to the Arm teams. They have already provided a fix for one of these issues that is integrated in the latest release of their compiler 20.1. We will therefore continue doing this work for the remaining issues.

8.1.2 CP2K

CP2K also compiles out of box with the GNU 9.2.0 compiler on Arm systems which is nice. It does not compile with the Arm Compiler for Linux. This is something that was a bit expected since it is known that CP2K does not compile with the PGI compiler and Flang for instance:

https://www.cp2k.org/dev:compiler_support

In a “software co-design” approach, we worked on isolating as many compiler issues as we could and making reproducers for them. This enabled us to file seven compilation issues to the Arm teams, three of them concerning compilation issues of libDBCSR. We have already seen progress from Arm since the 20.1 release of their compiler now provides support for the “Fortran Block statement” which was one of the issues being faced.

We have not been able to work around the remaining issues so far. Therefore, we will use a GNU build of CP2K for future work on Arm platforms.

The build recipes of CP2K for Arm are available on:

https://gitlab.com/arm-hpc/packages/-/wikis/packages/cp2k

8.1.3 ELPA

The Eigenvalue SoLvers for Petaflop-Applications (ELPA) library can be used in several MAX flagship codes (including CP2K, SIESTA, Fleur, Quantum ESPRESSO, and Yambo) to accelerate the computation of eigenvalues. We have therefore started to port it onto Arm hardware. We currently have serial and OpenMP, NEON versions of the library with both GNU and Arm compilers. We will focus on the MPI version in the forthcoming future.

8.1.4 Yambo

We have not faced any compilation issues, neither with the GNU compiler nor with the Arm Compiler for Linux. The only obstacles that we had to work around are configuration issues. But overall it went smoothly. We have been able to run the code on small examples and tutorials.

The current build recipes of Yambo for Arm are available on:

https://gitlab.com/arm-hpc/packages/-/wikis/packages/yambo

8.2 Experiences on the ARMIDA cluster

In this section we report performance scalability results and observations obtained on a state-of-the-art 64 bit ARM CPU Cluster.

The system under test is a 8 nodes cluster interconnected by InfiniBand EDR 100 Gb/s fabric. Each node has the following specification:

Hardware details

CPU: 2x Marvell ThunderX2 CN9980, each CPU host 32 physical cores @ 2.0 GHz base frequency

RAM: 16 GiB DDR4-2666, we used 1 DIMM per channel in order to achieve max memory bandwidth

HPC Network: 1x Mellanox ConnectX-5 EDR Infiniband 100 Gb/s port

Extended Marvell ThunderX2 details

Based on– Arm®v8-A architecture, 32 cores 64-bit

Each core integrates two 128-bit FP (Floating-Point) units, 32 KB L1 instruction and data cache, 256KB L2 per core, 32 MB distributed L3 cache. ThunderX2 is capable of Simultaneous MultiThreading – up to 4 per physical core, SMT set to 1 for HPC benchmarking.

The FP units can execute, in a single clock cycle, two double-precision SIMD instructions using the Arm NEON vector extensions, each FP unit can execute two double-precision FMA

(Fused-Multiply-Accumulate) instructions per clock cycle corresponding to a theoretical peak performance of 8 double-precision FLOP/cycle.

Dual socket system, as the ones under test, are connected by Coherent Processor Interconnect (CCPI2™, Speed: 600Gbps)

Each CPU has 8x DDR4-2666 channels providing a theoretical peak bandwidth of ≈ 160 GB/s, dual socket system can therefore exploit 16x DDR4-2666 channels for peak memory bandwidth of ~= 320GB/.

Software Stack

OS: Red Hat Enterprise linux release 8.0

Kernel: 4.18.0-80.el8.aarch64

Openmpi-4.0.2

gcc-8.2.0

ARM performance libraries: armpl 19.3.0

8.2.1 Quantum ESPRESSO Intranode Performance

Using AUSURF112 model we recorded very low efficiency as the number of process increases (see Fig. 5), notably performance do not improve, or get even worse, when process start to be allocated on the second NUMA node: processes 1 to 32 runs on numa-node 0, processes 33 to 64 are allocated on NUMA node 1.

```
NUMA topology:
available: 2 nodes (0-1)
node 0 cpus: 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24
25 26 27 28 29 30 31
node 0 size: 130245 MB
node 0 free: 97087 MB
node 1 cpus: 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50 51 52
53 54 55 56 57 58 59 60 61 62 63
node 1 size: 130903 MB
node 1 free: 109508 MB
node distances:
node   0   1
  0:  10  20
  1:  20  10
```

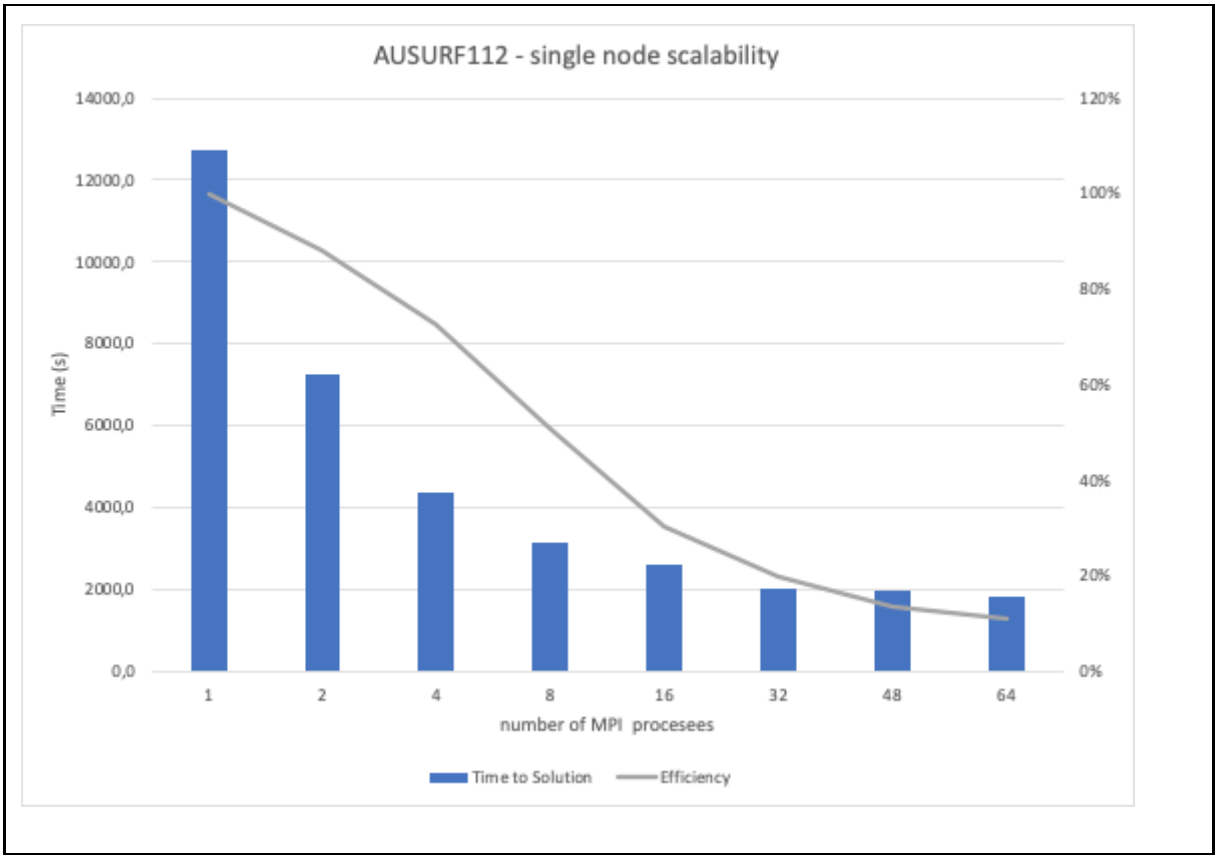

Figure 5: AUSURF112 Quantum ESPRESSO benchmark on the ARMIDA cluster.

This could be, at least partially, explained by an architectural bottleneck of Cavium Coherent Processor Interconnect™ (CCPI). CCPI is an socket interconnection that delivers a theoretical throughput of 600 Gb/s (75 GB/s) but this throughput is further decreased when traffic moves in (RX) and out (TX) from the socket. The throughput is automatically balanced from the CCPI controllers related to the traffic direction, so it can be completely dedicated to a single direction or being splitted for both TX and RX data movement. To evaluate this architectural bottleneck of CCPI, we run a simple measurement conducted using the STREAM memory benchmark. The test has been conducted allocating threads on NUMA node 0 while memory is forced to be allocated on NUMA node 1 (numactl -N 0 -m 1 stream):

OMP_NUM_THREADS	CPU Binding	Mem Binding	Copy	Triad	Peak/Measured	Speedup	Efficiency	Triad BW/thread GB/s	Note
1	skt 0	skt 1	7.309	9.430	12,3%	0,9	86%	9430	all threads on skt 0
2	skt 0	skt 1	14.195	18.351	23,9%	1,7	84%	9176	all threads on skt 0
4	skt 0	skt 1	22.986	23.084	30,1%	2,1	53%	5771	all threads on skt 0
8	skt 0	skt 1	29.732	23.841	31,0%	2,2	27%	2980	all threads on skt 0
16	skt 0	skt 1	32.540	31.801	41,4%	2,9	18%	1988	all threads on skt 0
32	skt 0	skt 1	32.631	32.573	42,4%	3,0	9%	1018	all threads on skt 0

The STREAM benchmark reads and writes data at the same time (for each read operation there is a consequent write operation), this means that the traffic is forced to traverse the CCPI interconnect in both directions. We can see from the table that the CCPI throughput drops to 32 GB/s (almost 50% of the 75GB/s). This throughput is quite low compared to the DDR memory throughput of each socket (158 GB/s for each direction, this means a total throughput of 316 GB/s, almost one order of magnitude higher with respect to CCPI).

While better intranode performance can be obtained using a hybrid configuration (2 MPI process binded respectively on NUMA node 0 and 1, and 32 OpenMP Threads per process), which greatly reduces the memory traffic on the CCPI interconnect, see Fig. 6:

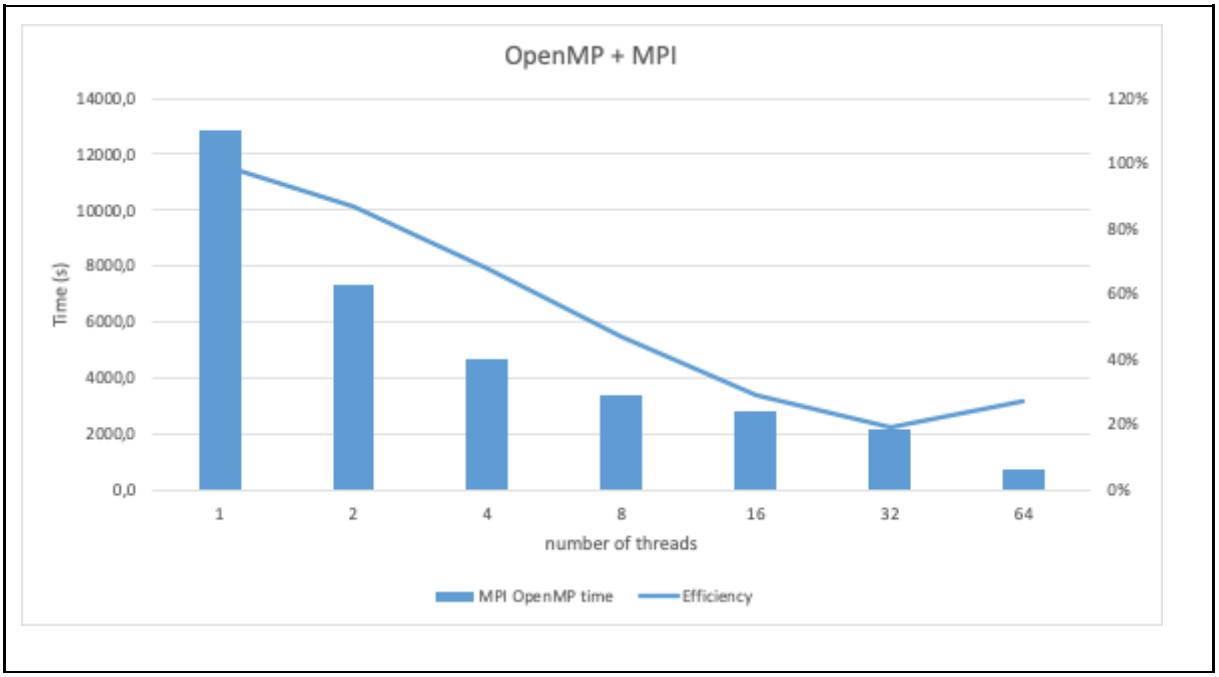

Figure 6: AUSURF112 Quantum ESPRESSO benchmark using a hybrid parallelization on the ARMIDA cluster.

Comparison between MPI Vs Hybrid (MPI + OpenMP) shows a -60% reduction in time to solution for the 64 threads/processes case (Fig. 7).

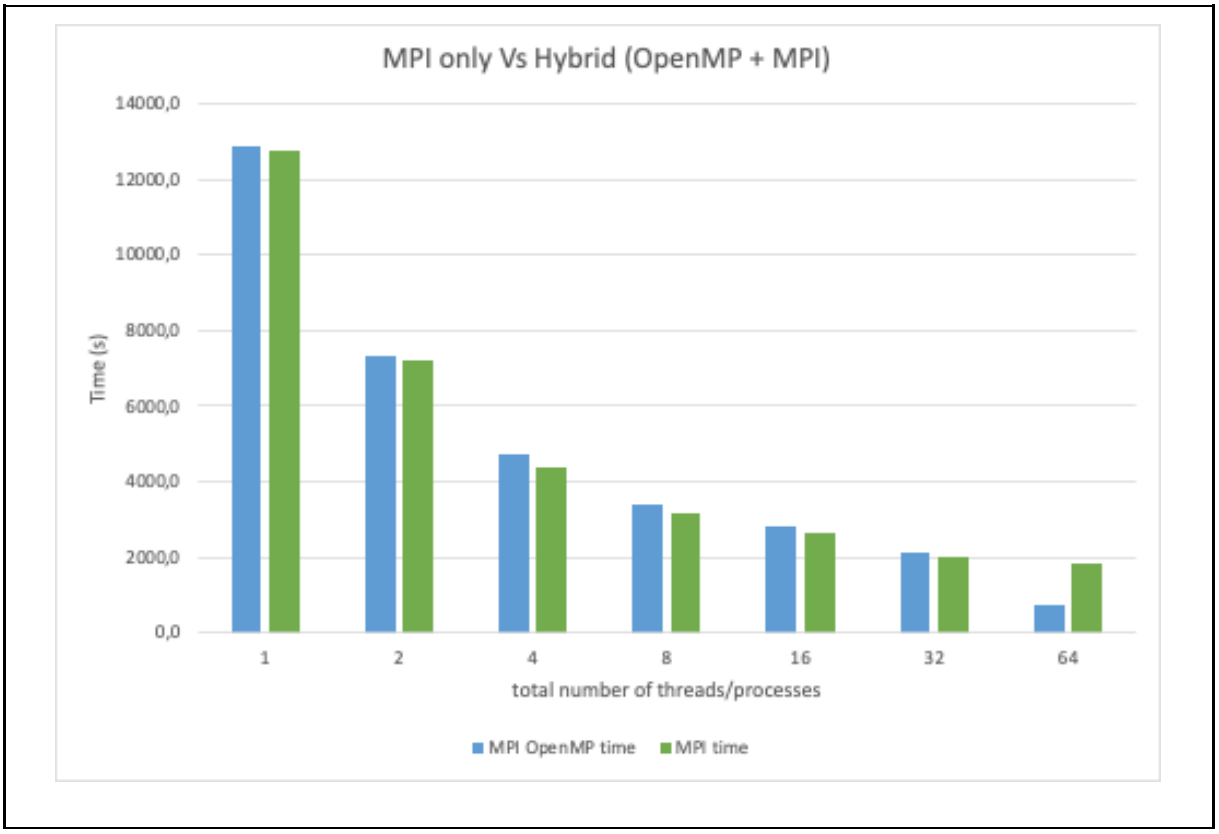

Figure 7: Comparison of performances using pure MPI vs hybrid MPI+OpenMP approaches.

8.2.2 Multi nodes scalability

Running with a fixed number of processes (64 as the intranode case) spread across multiple nodes did not provide any improvement (Fig. 8).

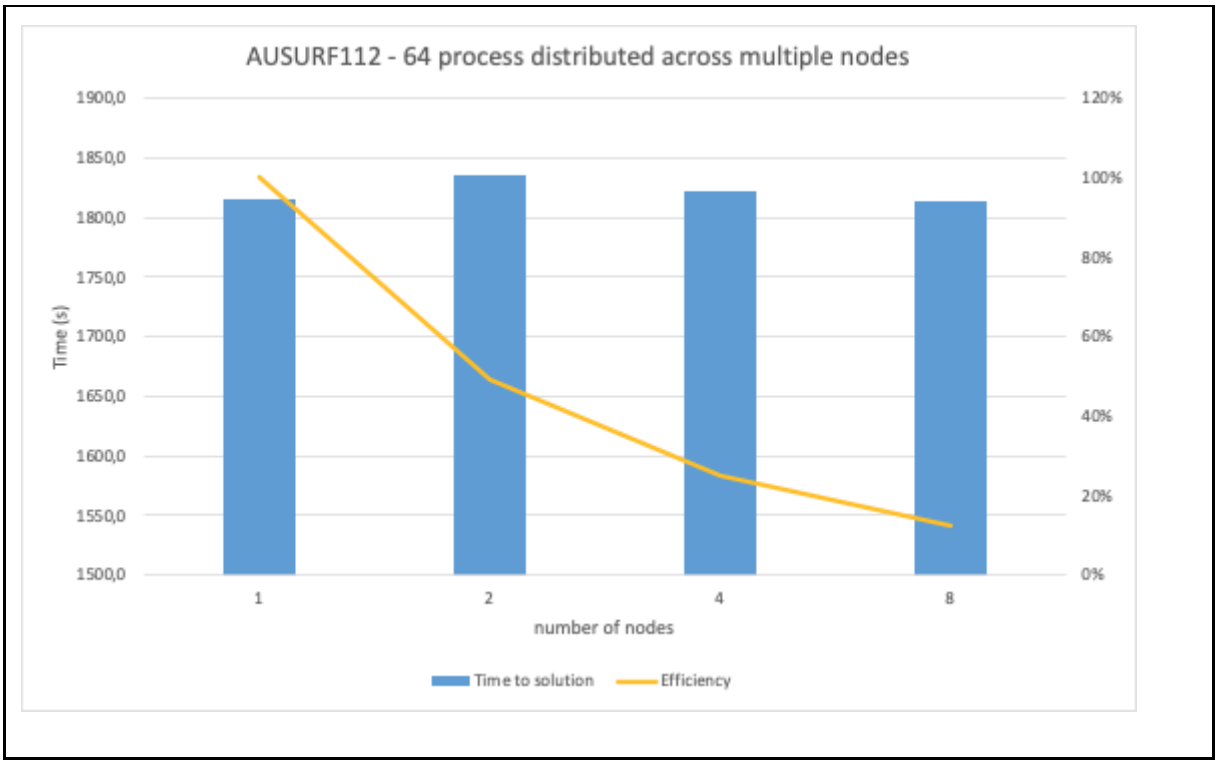

Figure 8: Out-of-node scalability performance on the ARMIDA cluster using AUSURF112 test case.

We run the test on up to 8 nodes with the following process placement schema:

2 nodes - 32 process per node

4 nodes - 16 process per node

8 nodes - 8 process per node

Increasing the parallelization level in order to use all the cores available in the 8 node cluster shows poor scalability for the hybrid run, none for the MPI only version (Fig. 9):

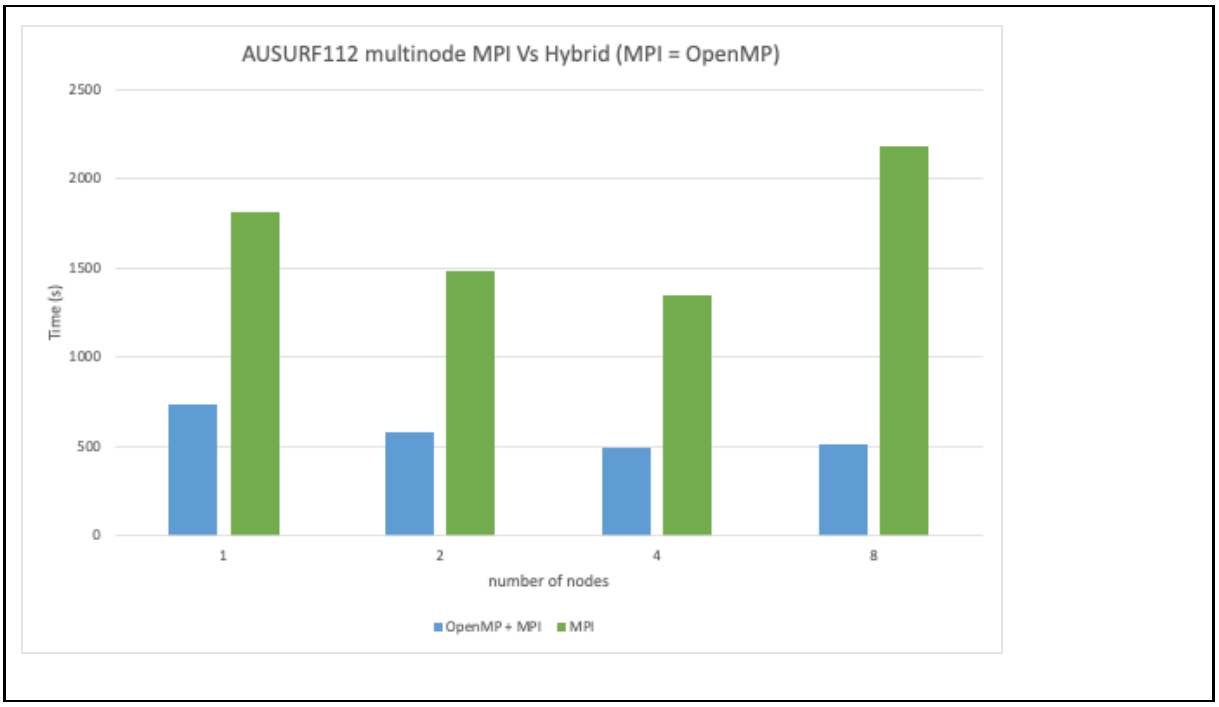

Figure 9: Comparison of pure MPI vs hybrid MPI+OpenMPI approaches in inter-node scaling benchmarks.

Number of cores involved:

#cores	#nodes	OpenMP + MPI Time to solution (s)	MPI Time to solution (s)
64	1	731,3	1816,0
128	2	581,54	1482,6
256	4	492,95	1344,5
512	8	509,5	2180,7

For the hybrid case 2 processes per node are allocated on different NUMA domain, each MPI process runs 32 threads.

8.3 SVE exploration

The current generation of HPC and server class Arm-based CPUs implements the Arm NEON SIMD instruction set. This gives software developers the possibility to use optimisation techniques similar to what they would use on other platforms. NEON registers have a fixed

width of 128 bits, which currently gives the possibility to run four single precision or two double precision floating point operations at the same time. In the same way, exploiting SSE, AVX, AVX2 and AVX-512 is unavoidable on x86 platforms to reach the full performance potential of the hardware, being able to exploit NEON is a must-do on current generation of Arm hardware like the Marvell Thunder X2, the AWS Graviton 2, the Huawei Kunpeng 920 or the Ampere Altra processor.

The Scalable Vector Extension (SVE) is a new advanced SIMD instruction set for Armv8-A AArch64 hardware[7]. HPC and server class CPUs fit into this category. It is not an extension of the current NEON SIMD instruction set but a new independent set of instructions particularly suited for HPC applications, bringing new capabilities that were missing with NEON: Scalable vector length, Per-lane predication, Gather-load and scatter-store, Vector partitioning, Fault-tolerant speculative vectorization, Horizontal vector operations and Serialized vector operations.

The A64FX from Fujitsu that will be used for the pre-exascale class Post-K Fugaku system at Riken will be the first chip implementing SVE. Full of promises no doubt that we will see more and more hardware implementing SVE in the forthcoming future. Ignoring the increasing interest for Arm based CPUs would be a mistake. Our goal here is to make sure that MAX flagship codes compile, run and are able to exploit the full performance potential of current and future Arm hardware targeted for HPC. The recent announcement by SiPearl that the first generations of EPI chips will be based on Arm architecture comforts us in this position, since it is key for us to make sure that the MAX codes will run on future HPC systems in Europe.

8.3.1 SVE results on BigDFT

As we said, SVE is going to play a bigger role in the Arm ecosystem for HPC in the forthcoming future. One of our goals is to make sure that MAX flagship codes can take advantage of this new advanced SIMD instruction set. We have started this work for BigDFT and we present here our first results.

We started by compiling the application for SVE with both compilers using the flags "-march=armv8-a+sve". We did this for several optimization levels : "-O2", "-O3" and "-Ofast".

We started to make sure that such versions of BigDFT were running without issues and producing sensible results by using ArmIE without instrumentation clients. The only issue that we faced was with an "Ofast version" of BigDFT when using the Arm Compiler for Linux. The code was producing "NaNs" during the initialization step. We have been able to identify the source of the issue and found a simple workaround.

[7] Porting and Optimizing HPC Applications for Arm SVE | What is the Scalable Vector Extension?

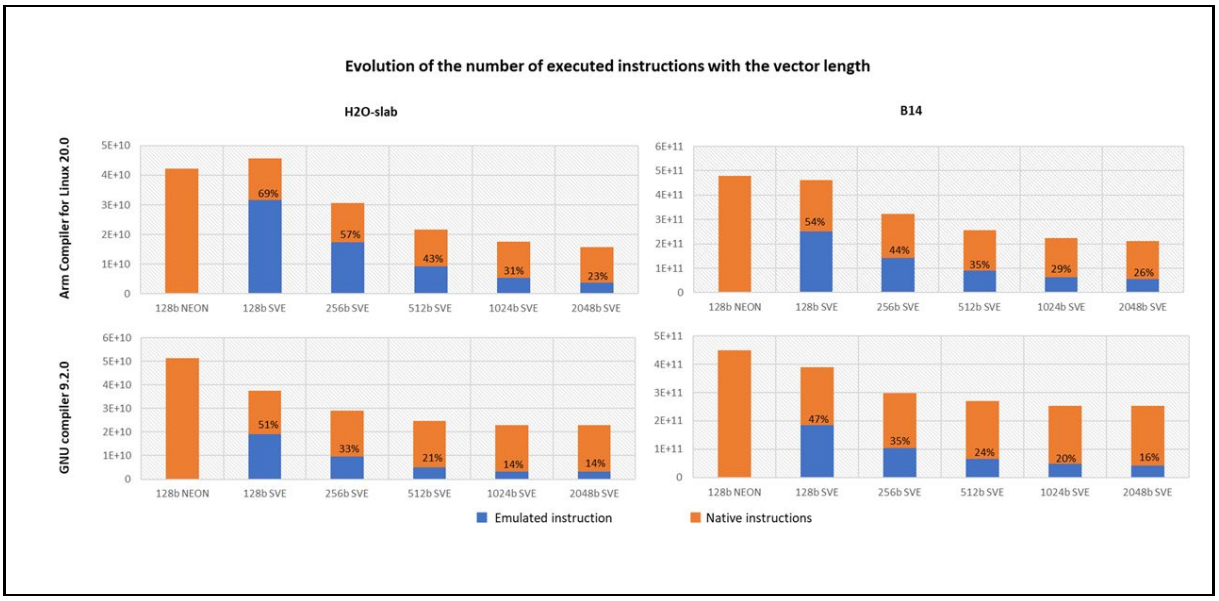

Figure 10: Amount of emulated SVE instructions using H2O-slab and B14 test cases with different vector lengths.

We then had a look at the amount of SVE instructions being executed thanks to the "libinscount_emulated" client provided with ArmIE. These instructions are not available on the current hardware so ArmIE needs to emulate them to execute the application. To know how much SVE instructions are executed we therefore have a look at the amount of emulated instructions. We ran the "libinscount_emulated" client on two different test cases: H2O-slab and B14 with different optimization levels (Fig.10). We have computed the ratio of emulated instructions (SVE instructions) by the total number of executed instructions. This amount of SVE instructions is quite low (below 8%) with the optimization flags "-O2" and "-O3". Moving to "-Ofast" seems to give far better results. This means that by using the right compilation flags, without any code modification, BigDFT is already able to make use of SVE instructions.

On the contrary of other SIMD instruction sets such as SSE, AVX, AVX2 and AVX-512 or NEON, SVE does not come with a fixed vector length. This means that silicon providers can choose to implement their own vector length as long as it is a multiple of 128 bits, the maximum allowed by the architecture being 2048 bits. End-users won't need to recompile their application to use SVE for another vector length. This does not answer the question "Which vector length would be the more beneficial for this set of applications?" though. Simple preliminary studies can be done thanks to ArmIE and its dynamic instrumentation clients.

We ran several experiments with the same binary using different vector lengths and we plotted the number of emulated instructions (SVE instructions) for each experiment. One

can quickly notice that, as expected, the amount of SVE instructions decreases when the vector length increases. Which is likely to be translated to a speed-up when executing on real hardware. An ideal model would be to divide the number of SVE instructions by the same amount by which we multiply the vector length. In other words, if we multiply the vector length by two, ideally, we should divide the number of SVE instructions by two. One can see from the graphs that this is interestingly not the case here. When we increase the vector width by two, the number of SVE instructions decreases by less than two. There is a "flattening" phenomenon when the vector length increases. Let's have a look at different derived metrics from the outputs of ArmIE.

SVE derived metrics for H2O-slab testcase

ACfL 20.0

Vector unit	Total instructions	Emulated instructions	Emulated instructions / Total instructions	Evolution of the number of emulated instructions	Evolution of the total number of instructions	Evolution of the total number of instructions compared to the NEON version	Efficiency / Influence	Comparison with GNU
128b NEON	42308750229	0	0.00%			100.00%	100.00%	82.48%
128b SVE	45744839541	31599049943	69.08%		108.12%	108.12%	92.49%	121.53%
256b SVE	30537463015	17287694741	56.61%	54.71%	66.76%	72.18%	69.27%	104.71%
512b SVE	21559108246	9261446748	42.96%	53.57%	70.60%	50.96%	49.06%	86.91%
1024b SVE	17490139965	5384925448	30.79%	58.14%	81.13%	41.34%	30.24%	76.53%
2048b SVE	15624658121	3595591399	23.01%	66.77%	89.33%	36.93%	16.92%	68.44%

GNU 9.2.0

Vector unit	Total instructions	Emulated instructions	Emulated instructions / Total Instructions	Evolution of the number of emulated instructions	Evolution of the total number of instructions	Evolution of the total number of instructions compared to the NEON version	Efficiency / Influence	Comparison with ACfL
128b NEON	51298483042	0	0.00%			100.00%	100.00%	121.25%
128b SVE	37640513910	19106126463	50.76%		73.38%	73.38%	136.29%	82.28%
256b SVE	29162790177	9605308827	32.94%	50.27%	77.48%	56.85%	87.95%	95.50%
512b SVE	24806657719	5118225648	20.63%	53.29%	85.06%	48.36%	51.70%	115.06%
1024b SVE	22853051055	3231672398	14.14%	63.14%	92.12%	44.55%	28.06%	130.66%
2048b SVE	22829531450	3144509760	13.77%	97.30%	99.90%	44.50%	14.04%	146.11%

SVE derived metrics for B14 testcase

ACfL 20.0

Vector unit	Total instructions	Emulated instructions	Emulated instructions / Total Instructions	Evolution of the number of emulated instructions	Evolution of the total number of instructions	Evolution of the total number of instructions compared to the NEON version	Efficiency / Influence	Comparison with GNU
128b NEON	4.79558E+11	0	0.00%			100.00%	100.00%	106.83%
128b SVE	4.62514E+11	2.50562E+11	54.17%		96.45%	96.45%	103.69%	118.92%
256b SVE	3.23536E+11	1.42376E+11	44.01%	56.82%	69.95%	67.47%	74.11%	108.85%
512b SVE	2.5577E+11	89111021113	34.84%	62.59%	79.05%	53.33%	46.87%	94.85%
1024b SVE	2.23929E+11	63882999390	28.53%	71.69%	87.55%	46.69%	26.77%	88.86%
2048b SVE	2.12201E+11	54620787931	25.74%	85.50%	94.76%	44.25%	14.12%	83.62%

GNU 9.2.0

Vector unit	Total instructions	Emulated instructions	Emulated instructions / Total Instructions	Evolution of the number of emulated instructions	Evolution of the total number of instructions	Evolution of the total number of instructions compared to the NEON version	Efficiency / Influence	Comparison with ACfL
128b NEON	4.48895E+11	0	0.00%			100.00%	100.00%	93.61%
128b SVE	3.88919E+11	1.03334E+11	47.46%		86.64%	86.64%	115.42%	84.09%
256b SVE	2.97223E+11	1.03334E+11	34.77%	55.98%	76.42%	66.21%	75.51%	91.87%
512b SVE	2.69661E+11	65020539387	24.11%	62.92%	90.73%	60.07%	41.62%	105.43%
1024b SVE	2.51999E+11	49250011687	19.54%	75.75%	93.45%	56.14%	22.27%	112.54%
2048b SVE	2.53767E+11	41037368085	16.17%	83.32%	100.70%	56.53%	11.06%	119.59%

The column of interest is the "Evolution of the number of emulated instructions" obtained by the formula:

Evolution of the number of emulated instructions [i] = Emulated instructions [i]/Emulated instructions [i-1]

As we already mentioned, ideally, we should divide by two the number of emulated instructions each time the vector length is multiplied by two. Hence if we follow the ideal model, the closer we are to 50% in this column the closer to the ideal model we are, the better. We can already see that on average, 256 bits and 512 bits are closer to the theoretical model than the 1024 bits and 2048 bits. This is the case for both test cases. But

since we have only taken the number of SVE instructions into account, this does not give us an idea of the influence on the code overall though.

This is the motivation to derive another metric. This time we look at the ratio between the total number of instructions for a given vector length and the total number of instructions for the NEON version of the code. This metric can be found under the name “Evolution of the total number of instructions compared to the NEON version” in the previous tables. If the code contains only SVE instructions, once more, we expect the total number of instructions to be divided by two each time we multiply the vector length by two. Hence, we divide the previous metric by the corresponding ideal ratio. This gives us an idea of the influence of the vector length on the total number of instructions, in the spirit of Amdahl’s law. As well as an idea of the efficiency of using a vector length rather than another one. This metric is available under the name “Efficiency / Influence” in the previous tables. The closer the influence is to 100%, the better. Since it means that using the corresponding vector length plays an important role on the total number of executed instructions. From the tables, we can see that it is likely that BigDFT would benefit from 256 bits wide vector registers with a significant speed-up. The “influence” of 1024 bits and 2048 bits being far below 50%, it is likely that 1024- and 2048-bits wide SVE vectors would not bring significant speed-up. Or it would require a thorough refactoring of the way computations are done. The efficiency of 512-bits vectors being around 50% it may still be worth to use such hardware.

It is also interesting to categorize the emulated instructions. To do so we use the “libinscount_emulated” client provided with ArmIE and some post processing hand-made scripts. We plot the number of instructions per category against the vector width. We present the results obtained with the H2O-slab testcase in the figure “Evolution of the SVE instruction mix with the vector length”.

An interesting point is the evolution of the number of FP reduction instructions: it is constant, it does not depend on the vector width. This participates to the flattening of the number of SVE instructions with the vector length. Another factor that one can also notice is that with the GNU compiler, the number of loads, FP multiplication and adds, Fused FP multiple-add are almost constant between 1024- and 2048-bits vectors. Finally, this also shows us that fused floating point operations are generated and used with both compilers. This is also a key point that is encouraging to take advantage of the full potential of the hardware. If it does not decrease the accuracy of the results.

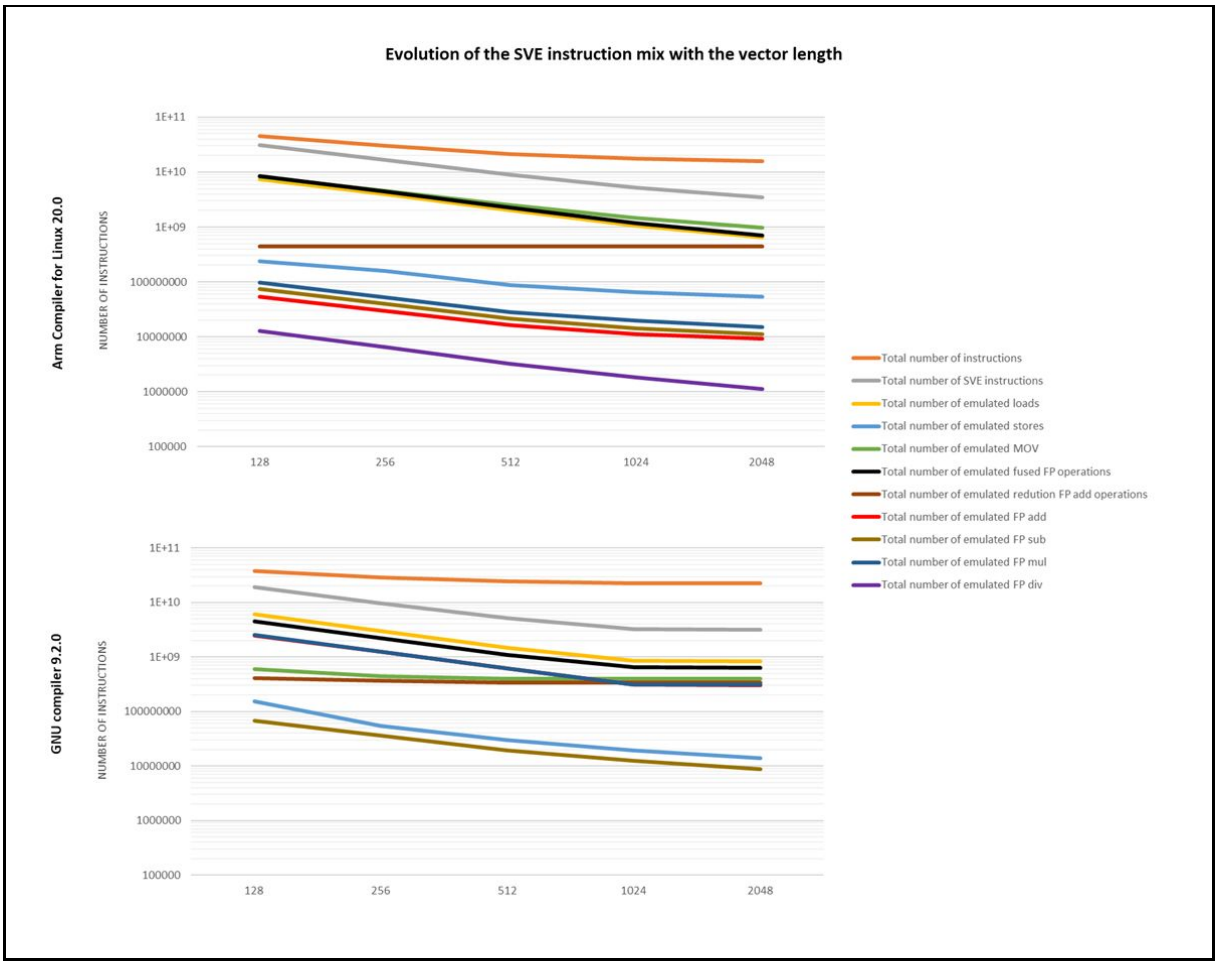

Figure 11: Study of the evolution of SVE instructions with the vector length in BigDFT.

This preliminary SVE study of BigDFT enabled us to have an overview of the usage of SVE by the application (Fig. 11). The results are promising and enabled us to check and identify some key aspects:

- the application runs when compiled for SVE
- the application can use the SVE instruction set to perform computations
- the application is likely to benefit from a substantial speed-up with 256- and 512-bits vector widths
- the application is likely to not benefit from substantial speed-up with 1024- and 2048-bits vector lengths
- SVE reduction instructions look influential on the SVE performances and behaviour of the application.

We will use this methodology on other MAX flagship codes. Future work will also focus on combining profiling work and ArmIE methodology to do some in-depth SVE analysis of the

main kernels of BigDFT. The idea is to isolate micro-kernels. We will also use other ArmIE clients to have a look at the breakdown of memory accesses on one hand and vector lane utilization on the other hand. We could as well consider using gem5 simulation to validate the SVE methodology based on ArmIE. Finally, additional benchmark work could be considered to check the performance of MAX flagship codes on current Arm based hardware.

8.3.2 SVE results on Quantum ESPRESSO

In this initial evaluation phase, we focused on the effectiveness of the Arm compiler toolchain when tasked with LAXLib auto-vectorization. The goal is to assess both the effectiveness of the custom compiler pass shipped by Arm (sve-loop-vectorize, that appears to be a totally different pass than upstream LLVM's loop-vectorize auto-vectorization pass) and the suitability of the LAXLib source code for auto-vectorization at different vector lengths. The steps taken during this activity have been planned according to the Asvie methodology presented by Arm during the SC19 Arm SVE Hackathon (https://jlinford.github.io/sc19-hackathon/).

We focused on LAXLib since it provides tools for distributed dense matrix diagonalization and, while it can rely on well known backends as ELPA or ScaLAPACK, it ships also with a custom algorithm to enable fine tuning and easier portability. LAXLib is of particular interest since it provides Quantum ESPRESSO with all the low-level linear algebra routines, especially those used by the Davidson solver (e.g. the Cannon algorithm for the matrix-matrix product). Besides providing a large number of functionalities to the upper application levels, the LAXlib component comes with a barebone mini-app (shipped in MaX Gitlab repository[8]) designed to enable a swift evaluation of the features both of a cluster network (if MPI is enabled) via block exchange operations and of dense linear algebra efficiency via algebraic operations. This mini-app driver has been the object of our evaluation.

All evaluations have been carried out on Armida (Marvell Thunder X2) using the Arm HPC Compiler 19.0 (*armflang -march=armv8-a+sve -O3 -fvectorize*) and the Arm Instruction Emulator (ArmIE). The latter proved itself as a key tool for SVE evaluations since it is capable of emulating SVE instructions while specifying a custom vector length (via the *-msve-vector-bits* command option), leaving the actual execution of supported Arm instructions to the CPU, an approach that adds no overhead to the actual execution except for the SVE instructions emulation itself. Due to the nature of the tests, no timing evaluations have been carried out, while the ratio between SVE emulated instructions and overall executed instructions is easily obtained from ArmIE reports. Results for LAXLib min-app are presented in Fig. 12.

Reason of the steady decrease of vector utilization given smaller vector units is likely due to the excellent utilization of SVE instructions by the armflang vectorizer and codegen (longer vectors mean less overall instructions executed). Static analysis of generated assembly code

[8] https://gitlab.com/max-centre/components/laxlib

to both prove our initial assumption and to clarify the reason for the plateau observed with 512 and 1024 bit vectors is currently being carried out.

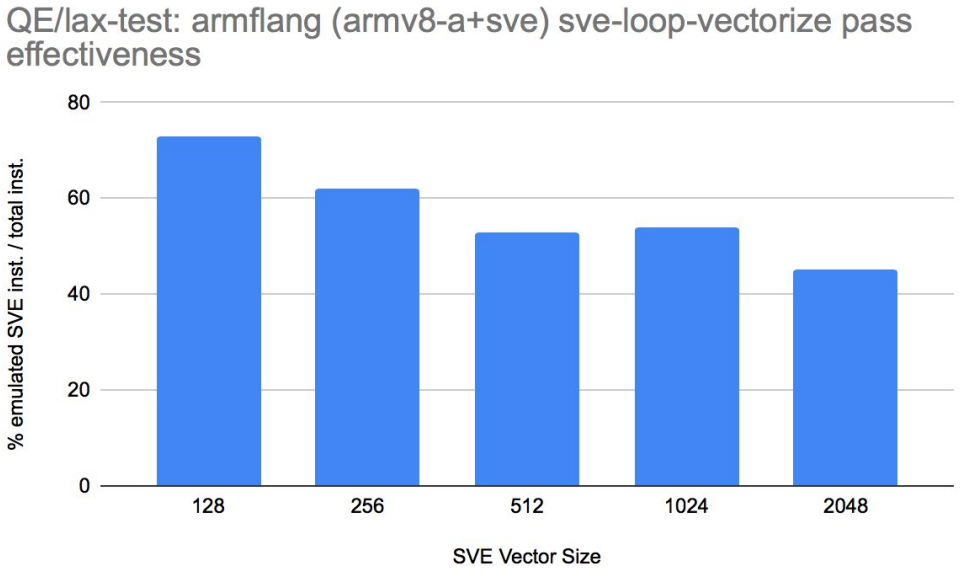

Figure 12: Fraction of the emulated SVE instructions over the total number of instructions as a function of the SVE vector size for the LAXlib mini-app in QuantumESPRESSO

8.4 ARM Hackathon

Arm-based processor technologies are now offering competitive performance and are on the path towards providing even much better solutions compared to what is otherwise available. To leverage these new technologies and to ease the porting of applications, it is necessary to familiarize users, code developers in the 64-bit Arm®v8 programming environment and development tools. The Open Edge and HPC Initiative (www.openedgehpcinitiative.org), CINECA (www.cineca.it), E4 Computer Engineering SpA (www.e4company.com) with the support of representatives of MaX have organized on October 28th/29th at CINECA the CODES@OEHI hackathon on 64-bit Arm®v8 based clusters. E4 Computer Engineering provided the ARMIDA (ARM Infrastructure for the Development of Applications) cluster, featuring 8 nodes connected via high-speed network. The goals of the 2-day hands-on workshop was to familiarise users with the 64-bit Arm®v8 programming environment by porting their applications and to further optimise the already ported applications. The hackathon was attended by about 20 people from academia, research institutions, and industry. Several codes were ported and a preliminary optimisations was performed on these codes using the development tools available on the system. CINECA and E4 are planning to organise a follow-on of the hackathon in fall 2020, using a GPU-powered version of the ARMIDA cluster, if the pandemic conditions will permit.

Figure 13: Daniele Cesarini and Federico Ficarelli (CINECA) attending the E4 ARM Hackathon in CINECA.

9 Intel Optane as additional RAM module

In this section, we are presenting some experiences with Quantum ESPRESSO and the Intel Optane memory devices. In Quantum ESPRESSO, similarly to other DFT based codes, one of the most limiting factors remains the amount of RAM memory per node. RAM dimensions narrow the size of materials that can be simulated or their accuracy. On the technological side, the RAM represents a bottleneck because its speed struggles to keep up with the CPU evolution and also because a large memory capacity is almost never obtained at an affordable price.

A new technology recently introduced by Intel called Optane Memory, codename ”Apache Pass”, aims to fill this gap in performance capacity. Optane is a new class of persistent memory. In particular, “Apache pass” offers greater capacity than traditional RAM and faster performance than storage for a fraction of the RAM price.

Figure 14: Intel Optane “Apache pass” module.

“Apache pass” can be configured both as :

1. additional RAM that can be added to the general memory pool; this setup is called Memory Mode.

2. storage from Permanent Memory Aware (PM Aware) software; this setup is called Application Direct Mode.

When a fraction or the whole persistent memory module capacity is set to Memory Mode, RAM (DDR4) capacity is hidden from the OS and becomes the highest level cache. In this case the Apache Pass memory is not persistent.

We had the chance to test this exciting technology on a couple of twin nodes borrowed from intel.

Each of the twin node we had the opportunity to test, presented the following hardware:

- CPU: Intel Xeon Gold 6252
- RAM: 384 GB of DDR4
- Apache Pass: 1.5 TB

When we set all the amount of 1 node with the Apache Pass in Memory Mode, the memory pool went from 384 GB up to approximately 1.2 TB (a small fraction of Apache pass turn to memory registry).

When DDR4 and Apache pass are coupled together data is loaded inside the DDR4 devices until they can fit (DRAM Cache Hit), otherwise they go to the Apache Pass DRAM Cache Miss).

The Memory Controller manages the data placement so that Apache Pass appears as fast as DDR4.

We performed a Quantum ESPRESSO benchmark to test the efficiency of Apache Pass (Fig. 14).

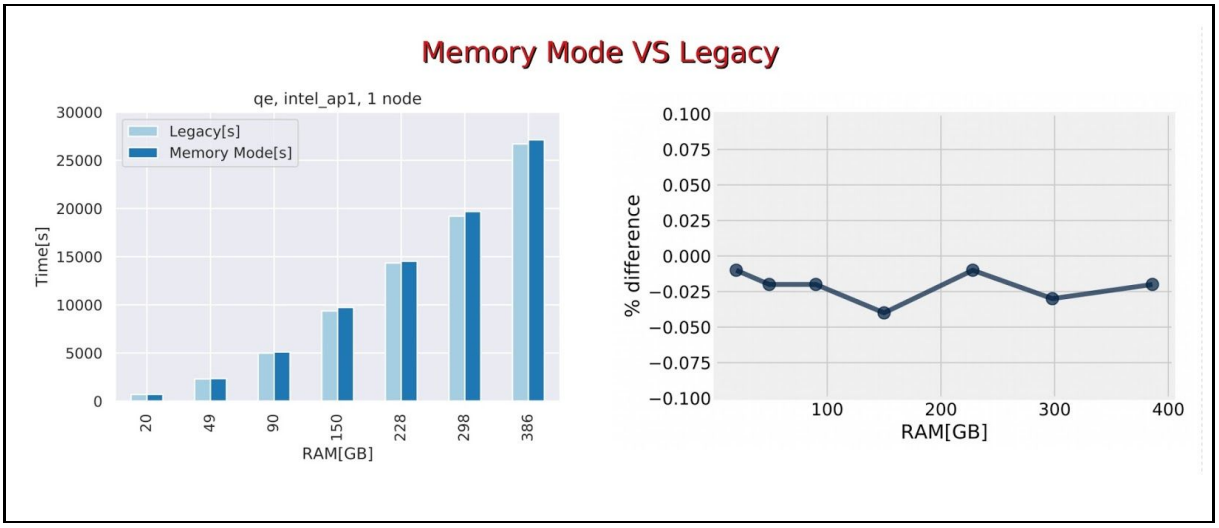

Figure 15: Performance comparison for Quantum ESPRESSO increasing workloads. Legacy represents the standard DDR4 RAM.

In Fig. 15, we show the performance comparison of Quantum ESPRESSO for increasing workloads, where "Legacy" represents the standard DDR4 RAM and the Apache Passe set in memory mode.

The material that we employed as an input file is the Zirconium Silicide, this material is part of a large benchmark dataset of Quantum ESPRESSO (https://github.com/electronic-structure/benchmarks).

In our test we used 2 different slabs with the same crystalline structure but different sizes:

1. Small slab of 24 atoms in total with a K-point mesh of 6 x 19 x 13.
2. Large slab of 108 atoms in total with a K-point mesh of 6 x 6 x 5.

We switched among the 2 input files and we tuned the energy cutoff to uniformly sample a wide interval of memory usage.

In the ordinates of the Figure 14there are respectively the total time of the self-consistent cycle of the material (left panel) and % difference between the total times of Legacy and Memory Mode (right panel) obtained as:

*% difference = (run_time_Legacy – run_time_Memory_Mode)/ run_time_Legacy * 100*

In the memory range analyzed, the differences between Memory Mode and DDR4 are quite small. This confirms the equivalent performances of a system contained entirely by DDR4 RAM and one with Apache Pass.

The Memory Mode is the easiest way to exploit the advantages of the Apache Pass technology but the Application Direct Mode promise to be even more exciting in terms of performance; this is because in Application Direct Mode the page cache can be removed from I/O path allowing a direct mappings of the persistent memory media.

In order to use the Application Direct Mode it is necessary to modify the standard application's code, changing the common malloc() and free() functions in their pmem-aware version.

At present some pmem-aware programs are already available (e.g. Flexible IO).

The future development of this project could involve writing the pmem-aware version of commonly used codes to analyze their performance's gain.

10 Co-design feedback to HPC technology providers

In the co-design cycle, discussed in the D4.1, we exposed our approach which is actually much focused on the SW side of co-design, following the evolution of the technology. In this sense, the main action of MaX is exercised in the "inner loop" of the co-design cycle, where, starting from the microbenchmarks and from the related performance models, new architectural features are evaluated and the codes are accordingly modified in order to maximize their performances. MaX is looking forward to playing a role in the interaction

towards the world of HPC technology providers (vendors and integrators). While MaX cannot make trade-off decisions, since this can only be performed by hardware architects, the aim of MaX is to influence trade-off decisions by creating results that could be used for such trade-off decisions.

MaX has two strategic assets: the flagship codes of MaX, already running on european Tier-0 systems, and the developers of such codes, with significant programming and system-level expertise in exploiting the architectural features of these systems. This combination of codes and developers represent an asset and an opportunity to engage with HPC technology providers designing the current and next generation of HPC systems. The strategy to port, optimize and use these codes on pre-exascale and exascale systems will stress these systems, because the codes will exercise all components and features of these systems. The results and outcome of the runs on the pre-exascale and exascale systems, and the comparison with current-generation systems, will enable MaX to provide data points and pinpoint strengths and weaknesses of the design of the next architectures. Therefore, we see the role of MaX in reporting application-oriented (not simply and not only ad-hoc microkernels) and comparative (having run the codes of multiple systems of different architectures) data points to technology providers, and support their decision-making process when designing new architectures.

Through the engagement of CINECA, BSC, CEA and Juelich in many different EU funded projects, MaX has a voice in the design of the next HPC architectures within the EuroHPC framework. Quantum ESPRESSO has been used in the EPI's co-design process, where the outcome of the key features of the code (e.g. use of the GPUs, and the exploitation of the SVE) has been factored-in the design of the first-generation processor, along with the requirements of other codes. It is planned that when the system/testbed based on the first-generation EPI processor will be available, MaX will have access to it to provide a proof of the results of the co-design process.

EuroHPC issued two "Pilot" calls, whose projects will start in the second half of 2021 and span for 3 or 4 years. Pilot 1 will be focused on the first generation of the processor designed by EPI and marketed by SIPEARL. Pilot 2 will be instead focused on the RISC V architecture. These Pilots are intended to be end-to-end, proof-of-concept platforms, built to enable users and developers to test their applications on fully-functional systems representative of the exascale and post-exascale systems. MaX codes are represented in Pilot 1 with the presence of BigDFT. This MaX code has been identified to represent the exascale material science applications. BigDFT is part of the official benchmark suite of Pilot 1 because it has been recognized to implement real use-cases and complex workflows that characterize typical exascale applications. Its usage in Pilot 1 will impact the architecture codesign of the prototype of the first European exascale computing platform.

The role in the procurement of HPC pre-exascale systems.

CP2K with DBCSR back-end and Quantum ESPRESSO were among the codes used as benchmarks for the procurement of the LUMI, Leonardo and MareNostrum V

supercomputers, the three pre-exascale EuroHPC systems in Europe, with the requirement that contributions from vendors needed to meet performance projections will be upstreamed to the open source codes. Quantum ESPRESSO, DBCSR and SIRIUS developers are currently collaborating with different vendors to provide support for next generation hardware, and will try to work closely with vendors on the pre-exascale architectures when the outcome of the procurement will be announced. Siesta, CP2K and Quantum ESPRESSO, on the other hand, also appear to be the target codes for the new Fugaku supercomputer by Fujitsu. The benchmarks on these codes, accurately ported and optimized, were used as reference when testing and assessing the operations of the new japanese system.

The plans for the future: guidelines on gathered experience.

The sub-optimal performance reported in Section 8.2.1 of this deliverable is a typical example of the need to have in place a process that MaX should apply when any performance- or features-related issues are met during the development, porting, optimization and execution of a code. Many partners of MaX have already fruitful interactions with key technology providers and developers of system-level tools (compilers, SDK, math libraries) and interactions mostly take place at sort of one-to-one level. The technology providers are usually extremely eager to get realistic and reliable data points with respect to bottlenecks, strengths, weaknesses, because that data points provided by the MaX applications enable them to make decisions based on applications relevant to the scientific communities and industrial enterprises, assessing the trade-offs among different architectural options for the design of the systems. MaX is currently working to better formalize the process of sharing data points through an open-data portal. In addition to the submission of data points acquired on existing architectures, and based on the findings gathered comparing different architectures, MaX will compile a list of recommendations about system-level architectural features (about the processor, the memory and its bandwidth, the on-board/off-board storage and the interconnect fabric) and submit it to technology providers to support their planning strategy.

11 Conclusions

In this deliverable we have reported about the experience made with software co-design and proofs-of-concept aimed at defining exploitation cases of new hardware architectures.

The work presented here shows that many different aspects are in fast evolution, in particular because of the exascale challenge. Different programming approaches are competing to be, at the same time, efficient and user-friendly. This competition is, however, far to be concluded, and at the moment no model is clearly prevailing over the others. At the same time, some domain specific issues have not yet found a mature solution that can be considered as a standard. Most remarkably, the absence of a highly performant library for distributed linear algebra, replacing the standard ScaLAPACK, is affecting the performance of the materials science codes tested here.

In this deliverable we have also discussed a significant set of MaX activities related to the experience made by working with MaX codes or SW components on ARM-based HPC technology, that we deem particularly relevant in view of the EuroHPC evolution.

The experience with Intel Optane showed how important it will be in the future to rely on auxiliary memory-persistent devices. In order to get benefit from this hardware, it will be necessary to rethink the way applications use the memory hierarchies and how to leverage on new computing workflows.

Besides the main focus on the software side of the co-design cycle, we are also considering the interactions with the HPC technology providers as one of the aims of the MaX work. Our target, for the moment, is to influence the trade-off decisions by providing data figures helping in the choices of the hardware designers and integrators. One example in this sense is represented by the fact that some of MaX flagship codes have been included in the benchmark-suite for the tender of the three new EuroHPC pre-exascale machines.

In summary, this deliverable shows how much benefit the materials research community can get from the co-design and the experimentation of new technologies, both software and hardware. The more the modernisation of the flagship codes will profit from these experiences, the more such software will be able to provide innovative and productive scientific workflows to materials research and innovation.