# Topics of Day-9 hands-on session

Parallel execution on CPUs and GPUs. Covered topics are:

- basic description of GPU acceleration,
- optimise CPU only runs,
- efficiently run on accelerated systems.

**Nota bene:** this set of exercises will be performed on Marconi100, the HPC system installed at CINECA.

Please login on the cluster with the credentials you have been provided and collect the exercises with:

```
git clone \
https://gitlab.com/QEF/materials-for-max-qe2021-online-school.git
```

**Exercise 0:** Basic information about GPU acceleration.

```
cd example0.intro/
```

**Exercise 1:** Setting up QE on CPU and GPU systems.

```
cd example1.setup/
```

**Exercise 2:** Parallel options – improve performance with npool and ndiag

```
cd example2.CPU/
```

**Exercise 3:** Accelerated systems – how to run with NVidia GPUs

```
cd example3.GPU/
```

# Exercise 0: (very) basic concepts about GPUs

This exercise is to provide some basic practical notions about how GPU acceleration works.

The source file `code_cpu.f90` is a minimal program to perform a matrix-matrix product on the CPU using the DGEMM subroutine from the BLAS libraries.

In order to compile the code, you first purge all modules with

```
module purge
```

and then load the nvfortran compiler with

```
module load autoload hpc-sdk
```

Then, you can compile the code appending the -lblas flag in order to link the BLAS libraries. In this case you will be using the BLAS libraries provided with the nvfortran compiler in the hpc-sdk package.

```
nvfortran -o code_cpu.x code_cpu.f90  -lblas
```

The source file `code_gpu.f90` does the same calculation as `code_cpu.f90`, but on the GPU, using the cuDGEMM subroutine from the cuBLAS libraries. In order to compile the code you load the CUDA module

```
module load autoload cuda
```

and then compile the code specifying that you want to use CUDA (-Mcuda) and that you want to link the cuBLAS libraries (-Mcudalib=cublas)

```
nvfortran -o code_gpu.x code_gpu.f90 -Mcuda -Mcudalib=cublas
```

1. Take a look inside the CPU and GPU code, to have an idea of the CUDA Fortran directives.
2. Launch the two executables with varying the SIZE (substitute SIZE with an integer) of the matrices and compare the elapsed time

```
./code_cpu.x SIZE

./code_gpu.x SIZE
```

Unfortunately in Quantum ESPRESSO things are a bit more complicated than this, because often the matrices are inizialized on the CPU and then need to be moved to the GPU in order to perfom the computations. Sometimes also the result of the computation needs to be moved back to the CPU memory.

This operations are usually referred to as “off-loadings” or “data transfer” between host and device memories. The source code `code_mix.f90` shows this in a very simplified manner.

1. Have a look at the `code_mix.f90` file and find the data transfers between host and device memories.
2. Launch `code_mix` and `code_gpu` for large matrix sizes, and compare the elapsed times. What can you say?

NOTE: As a reference, for a matrix size of 120000, the times should be something around:

```
code_cpu.x
  Full time:       199
  Product time:    193
code_gpu.x
  Full time:         1
  Product time:      1
code_mix.x
  Full time:         7
  Product time:      2
```

# Exercise 1: preparing QE

We will first prepare an HPC ready installation of QE. This exercise will show how to compile QE and check for relevant libraries in the context of standard and accelerated systems.

## CPU version

Download the last release, extract it and rename it with the commands below:

```
wget
↪  https://gitlab.com/QEF/q-e/-/archive/qe-6.7MaX-Release/q-e-qe-6.7Ma
tar xjf q-e-qe-6.7MaX-Release.tar.bz2
mv q-e-qe-6.7MaX-Release qe-cpu
cd qe-cpu
```

*Note:* for the copy-paste friendly version, open the `README.md` file in each directory. Alternatively you can click here to jump to the web-page with QE releases.

For the CPU version we will use hpc-sdk and SpectrumMPI which are a good combination on OpenPower machines. The FFTW library is also required. The environment is setup using the following modules.

```
module purge
module load hpc-sdk/2020--binary spectrum_mpi/10.3.1--binary
↪  fftw/3.3.8--spectrum_mpi--10.3.1--binary
```

Configure QE with the following option, that will select PGI compilers (now rebranded hpc-sdk) and SpectrumMPI

```
./configure  CC=pgcc F77=pgf90 FC=pgf90 F90=pgf90
↪  MPIF90=mpipgifort
```

1. **. . . check that relevant libraries have been detected**, namely on this system, blas, lapack from hpc-sdk and fftw3:

```
The following libraries have been found:
   BLAS_LIBS=-lblas

   ↪  LAPACK_LIBS=-L/cineca/prod/opt/compilers/hpc-sdk/2020/binary
   ↪  -llapack -lblas
   FFT_LIBS= -lfftw3
```

*Note:* we did not enable OpenMP in this case since we will be dealing with small input file. If you plan to run large simulations or you happen to run with accelerators, OpenMP is important and we will indeed enable it in the next section.

We will only benchmark `pw.x`. Let's compile it with the command

```
make -j pw
```

Now enjoy tea or coffe while you wait 3 minutes or so.

## GPU version

Now go back to the folder of example1 and download the last release of the GPU accelerated version of QE

```
wget
↪  https://gitlab.com/QEF/q-e-gpu/-/archive/qe-gpu-6.7/q-e-gpu-qe-gpu-
tar xjf q-e-gpu-qe-gpu-6.7.tar.bz2
mv q-e-gpu-qe-gpu-6.7 qe-gpu
cd qe-gpu
```

*Note:* for the copy-paste friendly version, open the `README.md` file in each directory. Alternatively you can click here to jump to the web-page with QE-GPU releases.

For the GPU version you *must* use the HPC-SDK which provides a CUDA Fortran compiler. The other libraries remain the same, except for cuda

```
module purge
module load    hpc-sdk/2020--binary
↪  spectrum_mpi/10.3.1--binary
↪  fftw/3.3.8--spectrum_mpi--10.3.1--binary  cuda/11.0
```

Configure with

```
./configure CC=pgcc F77=pgf90 FC=pgf90 F90=pgf90
↪  MPIF90=mpipgifort --enable-openmp --with-cuda=$CUDA_HOME
↪  --with-cuda-runtime=11.0 --with-cuda-cc=70
```

1. . . . **check that relevant libraries have been detected**, DFLAGS show that *CUDA*, *CUSOLVER and MPI* will be activated:

```
setting DFLAGS... -D__PGI -D__CUDA -D__USE_CUSOLVER -D__FFTW
↪  -D__MPI
[...]
The following libraries have been found:
  BLAS_LIBS=-lblas
  LAPACK_LIBS=-L/cineca/prod/opt/compilers/hpc-sdk/2020/
  ↪  binary/Linux_ppc64le/2020/profilers/Nsight_Systems/
  ↪  host-linux-ppc64le -llapack -lblas
  FFT_LIBS=
```

You'll notice that the code is using the internal version of FFTW (`-D__FFTW` instead of `-D__FFTW3`). This is not an issue in this case since 99% of the FFTs will be performed on the GPU with optimized CUDA libraries.

Compile again the code

```
make -j pw
```

Congratulations, now you have both a "standard" and an "accelerated" version of `pw.x` to be used in the following exercises.

# Exercise 2: optimize CPU execution

In this section we only make use of CPUs and try to optimize the time to solution keeping the amount of compute power fixed.

## 1. Pool parallelism

Optimize the number of kpoint pools, starting with 1 up to 8 (what are the admissible values for this option?). The jobscript file to be used on Marconi100 is already available in this folder and is also reported below for your convenience.

```
#!/bin/bash
#SBATCH --nodes=1                  # number of nodes
#SBATCH --ntasks-per-node=16       # number of MPI per node
#SBATCH --cpus-per-task=4          # number of HW threads per task
#SBATCH --mem=230000MB
#SBATCH --time 00:30:00             # format: HH:MM:SS
#SBATCH -p m100_usr_prod
#SBATCH -J qeschool

module load    hpc-sdk/2020--binary
↪  spectrum_mpi/10.3.1--binary
↪  fftw/3.3.8--spectrum_mpi--10.3.1--binary

export QE_ROOT=../example1.setup/qe-cpu/

export PW=$QE_ROOT/bin/pw.x

# This sets OpenMP parallelism, in this case we do a pure MPI
export OMP_NUM_THREADS=1

# Run pw.x with default options for npool and ndiag
mpirun  ${PW} -npool 1 -ndiag 1 -inp pw.CuO.scf.in | tee
↪  no_options
```

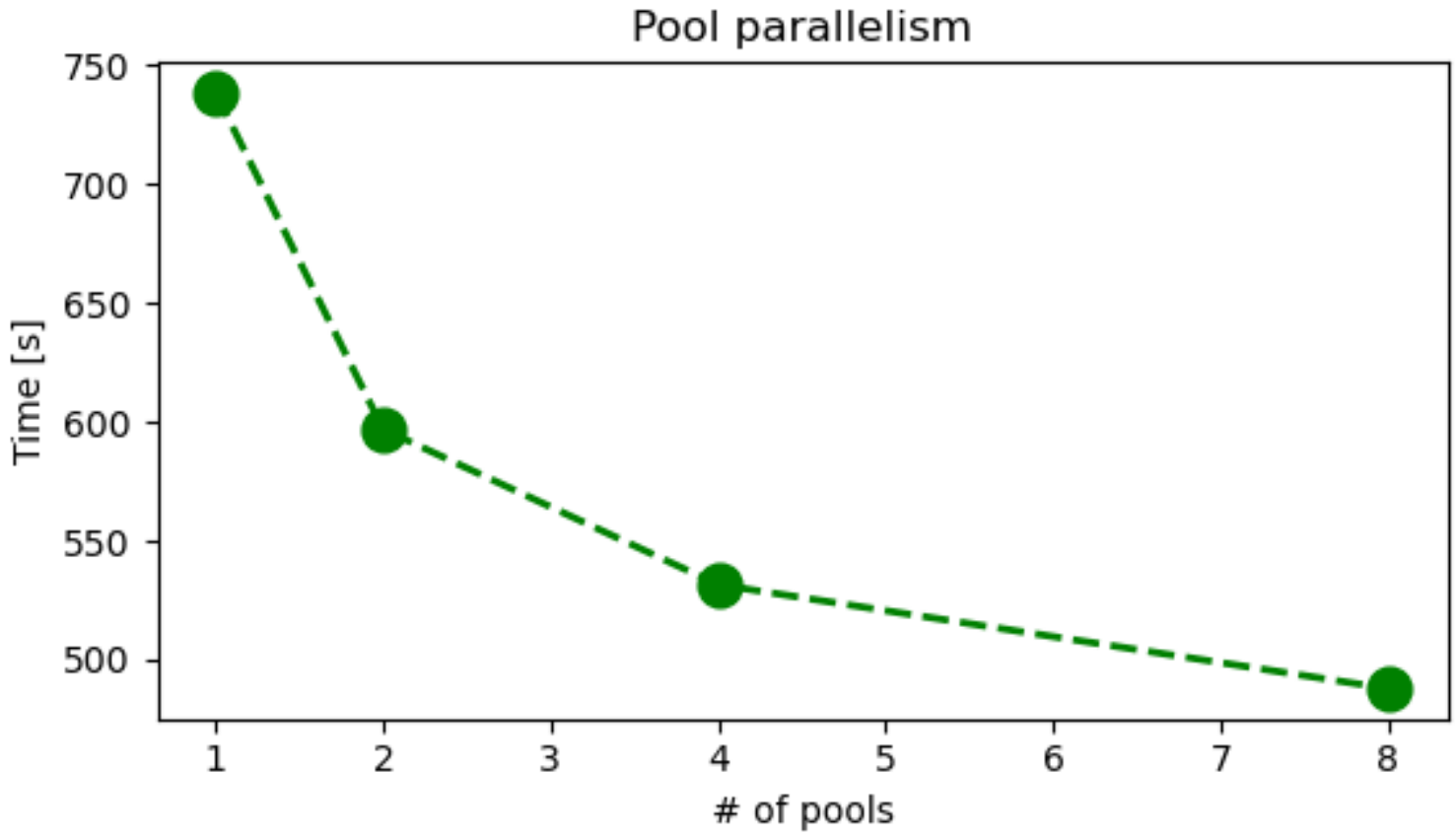
1. First, **submit the job as is**, with npool set to 1.
2. Second, **open the job-script file** (`job.sh`) and **change the number of pools to be used `-npool X`**, with X={2,4,8}. Don't forget to rename the output file as well.
3. **Collect the time** taken by the code as a function of the number of k point pools.

The execution time can be obtained by looking at one of the last lines of the output, that reads for example

```
PWSCF        :    5m53.84s CPU    5m58.18s WALL
```

the WALL time is the value you want to note down (the CPU time is the amount of time spent by the CPU processing `pw.x` instructions, which is a considerable portion of the whole execution time, but neglects, for example, I/O. For more details check wikipedia).

You should be able to produce a plot similar to this one:




Congrats! With the same computational resources, the time to solution is reduced by 1/3!

Pool parallelism can be actually much better than what you obtained in this example. Indeed for this small input file the parallelization on plane waves is good enough, especially because all our MPI processes reside on a single node and inter-process communication is fast.

## 2. Parallel diagonalization

In this second part we want to speedup the code by solving the dense eigenvalue problem using more than one core.

1. **Set `-npool` to 4 and activate parallel diagonalization by changing `-ndiag 4` to improve the performance.**
2. Inspect the beginning of the output file and look for this message

```
Subspace diagonalization in iterative solution of
the eigenvalue problem:
one sub-group per band group will be used
custom distributed-memory algorithm
(size of sub-group:  2*  2 procs)
```

3. Check the time to solution. Did you manage to reduce the WALL time?

Unfortunately you'll notice that the simulation is actually **taking longer**.

There are two reasons for this:

1. the eigenvalue is too small to take advantage of parallel diagonalization,
2. we didn't use optimized libraries for this task. The code is using a suboptimal parallel eigensolver. Two common options to improve in this case are linking Scalapack or ELPA libraries.

# Exercise 3: running with GPUs

To run the accelerated version you are supposed to couple **each MPI with a single GPU**. Therefore this time your jobscript is setup to request **two MPI processes and 2 GPUs** with your submission script.

The jobscript file to be used on Marconi100 is already available in this folder and is also reported below for your convenience.

```
#!/bin/bash
#SBATCH --ntasks-per-node=2     # number of MPI per node
#SBATCH --ntasks-per-socket=2   # number of MPI per socket
#SBATCH --cpus-per-task=8      # number of HW threads
#SBATCH --gres=gpu:2            # number of gpus per node
#SBATCH --mem=230000MB
#SBATCH --time 00:10:00         # format: HH:MM:SS
#SBATCH -A cin_QEdevel1_4
#SBATCH -p m100_usr_prod
#SBATCH -J qeschool

module load hpc-sdk/2020--binary spectrum_mpi/10.3.1--binary
↪  fftw/3.3.8--spectrum_mpi--10.3.1--binary

export QE_ROOT=../example1.setup/qe-gpu/
export PW=$QE_ROOT/bin/pw.x
export OMP_NUM_THREADS=1 # This sets OpenMP parallelism

# Run pw.x with default options for npool and ndiag
mpirun  ${PW} -npool 1 -ndiag 1 -inp pw.CuO.scf.in | tee
↪  no_options
```

1. **Analyze the difference with the previous jobscript** and,
2. **submit this jobscript** that will run the same input without any parallel optimization.
3. Once the simulation is complete, **check the output file**.

At the beginning of the output file you will spot

```
GPU acceleration is ACTIVE.
```

Moreover, this run should be much faster than any of the previous CPU tests, **taking slightly less than 2 minutes**.

4. Now try to **exploit the entire CPU with OpenMP**.

Change the environment variable set by the following command

```
export OMP_NUM_THREADS=X
```

with X=2,4,8.

5. You'll notice a small improvement and, eventually a saturation.

Once again, OpenMP is effective only for large simulation, but in this case it is used to take advantage of idle CPU cores as much as possible.

## Pool parallelism

You can improve the previous result with pool parallelism. This time you will be limited by the total number of MPI processes, namely 2.

1. **Modify the original jobscript**, set `-npool 2`, submit the job.
2. **Check the time to solution.**

You should observe a substantial **reduction of the time to solution** which is now about **3/4 of your previous test**. This improvement is actually due to FFTs that are now performed without communications on a single GPU.

## Oversubscription

For small inputs, one can possibly obtain some additional performance by oversubscribing the GPU.
Try to increase the number of MPI processes used to run this job by changing the jobscript as shown below:

```
#!/bin/bash
#SBATCH --ntasks-per-node=4     # number of MPI per node
#SBATCH --ntasks-per-socket=4   # number of MPI per socket
#SBATCH --cpus-per-task=4      # number of HW threads per
↪  task
#SBATCH --gres=gpu:2            # number of gpus per node
#SBATCH --mem=230000MB
#SBATCH --time 00:10:00         # format: HH:MM:SS
#SBATCH -A cin_QEdevel1_4
#SBATCH -p m100_usr_prod
#SBATCH -J qeschool

module load    hpc-sdk/2020--binary
↪  spectrum_mpi/10.3.1--binary
↪  fftw/3.3.8--spectrum_mpi--10.3.1--binary

export QE_ROOT=../example1.setup/qe-gpu/

export PW=$QE_ROOT/bin/pw.x

export OMP_NUM_THREADS=1

mpirun  ${PW} -npool 4 -ndiag 1 -inp pw.CuO.scf.in |
↪  oversubscription
```

## Compare with theoretical performance

The ratio between the peak performance of the GPU and the CPU is about a factor 20.

1. **Evaluate the ratio between the best time to solution of your CPU and GPU tests.** Do your results reproduce the ideal ratio? Why not?