# A Performance Study of Quantum ESPRESSO's PWscf Code on Multi-core and GPU Systems

Joshua Romero[1(✉)], Everett Phillips[1], Gregory Ruetsch[1], Massimiliano Fatica[1], Filippo Spiga[2], and Paolo Giannozzi[3]

[1] NVIDIA Corporation, Santa Clara, USA
joshr@nvidia.com
[2] Research Computing Service, University of Cambridge, Cambridge, UK
[3] Dip. Scienze Matematiche Informatiche e Fisiche, University of Udine, Udine, Italy

**Abstract.** We describe the porting of PWscf (Plane-Wave Self Consistent Field), a key component of the Quantum ESPRESSO open-source suite of codes for materials modeling, to GPU systems using CUDA Fortran. Kernel loop directives (CUF kernels) have been extensively used in order to have a single source code for both CPU and GPU implementations. The results of the GPU version have been carefully validated and the performance of the code on several GPU systems (both x86 and POWER8 based) has been compared with traditional Intel multi-core (CPU only) systems. This current GPU version can reduce the time-to-solution by an average factor of 2–3 running two different input cases widely used as benchmarks on small and large high performance computing systems.

**Keywords:** DFT · Materials science · Eigensolver · GPU computing CUDA Fortran

## 1 Introduction

Computer simulations of materials, in particular *first-principle* simulations based on density-functional theory [9,13], pseudo-potentials, and plane-wave basis sets [14], have become widespread in many fields of science as well as in industry. These applications are run on a variety of computing systems, from desktop PCs to very large parallel machines, depending on the physical system under investigation and the property to be computed. The search for better methodologies and for better algorithms is a very active field of research.

Among the various packages implementing first-principle techniques, we focus on Quantum ESPRESSO (QE) [6], an integrated suite of open-source software released under the terms of the GNU General Public License (GPL). Programs included in QE can perform many different kinds of calculations. The complete distribution consists of approximately 520,000 lines of Fortran 95 source code, some additional code written in C, auxiliary scripts, and Python utilities.


S. Jarvis et al. (Eds.): PMBS 2017, LNCS 10724, pp. 67–87, 2018.
https://doi.org/10.1007/978-3-319-72971-8_4

Due to accuracy requirements in electronic-structure computations, double precision floating point arithmetic is always used. In this study, we will concentrate on the PWscf code which solves self-consistently the Kohn-Sham equations arising in density-functional theory.

QE is designed to work on a variety of computing architectures and has evolved into a complex application with multiple layers of parallelism and key dependencies on mathematical libraries. The suite is able to run in serial and in parallel, targeting multi-core systems via multi-threaded libraries and explicit OpenMP and distributed systems using the Message Passing Interface (MPI) [12] and parallel libraries such as ScaLAPACK [2] or ELPA [11]. QE also supports modern approaches to effectively exploit multi-core and many-core architectures via hybrid parallelism based on MPI and OpenMP combined [17].

The need to accelerate time to discovery and tackle bigger and more challenging problems has motivated the first porting of QE to the programmable Graphics Processing Unit (GPU). GPUs are remarkable pieces of technology that have evolved into highly parallel many-core processors with floating-point performance and memory bandwidth that far exceed that of today's central processing units (CPUs). GPUs are especially well suited to address problems that can be expressed as data-parallel computations, where the same program is executed on different data elements in parallel. The CUDA programming model developed by NVIDIA has become the de-facto standard in GPU computing.

Today, the highest performing GPUs available on the market, suitable for scientific computation in fields like materials science, computational fluid dynamics, astrophysics and many others, are those within the NVIDIA Pascal family. In this paper, we will focus our evaluation on several computing platforms based on NVIDIA Pascal P100. This GPU is available in both PCI and SMX2 form-factors, with slightly different technical specifications (such as peak memory bandwidth and peak floating-point throughput). It is now possible to program GPUs in several languages, from the original CUDA C to the new OpenACC directive based compilers. QE is written in Fortran 90, so the natural choices for a GPU port are either CUDA Fortran or OpenACC. We decided to use CUDA Fortran as the structure of the code allows for the extensive use of CUF kernels, making the effort comparable to an OpenACC port, while also retaining the possibility of using explicit CUDA kernels when needed. In addition, the explicit nature of data movement in CUDA Fortran allows us to better optimize the CPU/GPU data movement and network traffic.

An initial GPU version of QE was developed several years ago [18] written in CUDA C and bundled with the original Fortran source code. This version, still available for reference and performance comparison [16], has been discontinued due to the complexity of managing and maintaining a large code base of mixed Fortran and CUDA C. This original version offloaded only limited portions of the workload to GPUs. A brand new version compatible with QE version 6 has been developed from the ground-up based on CUDA Fortran, focused on delivering performance on both large-scale and dense GPU system configurations, with all significant computation carried out on GPUs. As a consequence, unlike the

original plugin, this new version requires the complete dataset to fit in GPU memory.

The following section will first introduce the CUDA programming model and then provide an overview of CUDA Fortran and some specific features used in the porting effort. A detailed guide of the CUDA Fortran language extensions and features used can be found in [4].

## 2 CUDA Programming Model and CUDA Fortran

CUDA-enabled GPUs can contain anything from a few to thousands of processor cores which are capable of running tens of thousands of threads concurrently. To allow for the same CUDA code to run efficiently on different GPUs with varying specifications, a hierarchy of resources exists both in physical hardware, and in available programming models. In hardware, the processor cores on a GPU are grouped into multiprocessors. The programming model mimics this grouping: a subroutine run on the device, called a kernel, is launched with a grid of threads grouped into thread blocks. Within a thread block, data can be shared between threads, and there is a fine-grained thread and data parallelism. Thread blocks run independently of one another, which allows for scalability in the programming model: each block of threads can be scheduled on any of the available multiprocessors within a GPU, in any order, concurrently or sequentially, so that a compiled CUDA program can execute on a device with any number of multiprocessors. This scheduling is performed behind the scenes, the CUDA programmer needs only to partition the problem into coarse sub-problems that can be solved independently in parallel by blocks of threads, where each sub-problem is solved cooperatively in parallel by all threads within the block.

The CUDA platform enables hybrid computing, where both the host (CPU and its memory) and device (GPU and its memory) can be used to perform computations. A typical sequence of operations for a simple CUDA Fortran code is:

- Declare and allocate host and device memory
- Initialize host data
- Transfer data from the host to the device
- Execute one or more kernels
- Transfer results from the device to the host

From a performance perspective, the bandwidth of the PCIe bus is over an order of magnitude less than the bandwidth between the device's memory and GPU, and therefore a special emphasis needs to be placed on limiting and hiding PCIe traffic. For MPI applications, data transfers between the host and device are required to transfer data between MPI processes. Therefore, the use of asynchronous data transfers, i.e. performing data transfers concurrently with computation, becomes mandatory. This will be discussed in detail in Sect. 5.

**Data Declaration, Allocation, and Transfers.** The first Fortran extension we discuss is the variable attribute `device` used when declaring data that resides in GPU memory. Such declarations can be allocatable. The `allocate()` command has been overloaded so allocation occurs on the device when the argument is declared with the `device` attribute. Similarly, the assignment operator has been overloaded to perform data transfers between host and device memory spaces.

The Fortran 2003 `sourced` allocation construct, `allocate(lhs, source=rhs)`, is also supported and extended. When `allocate` is invoked with the optional `source=` argument, `lhs` becomes a clone of `rhs`: it is allocated with the same shape of `rhs` and each element of `rhs` is copied into the corresponding element of `lhs`. In CUDA Fortran, if the `lhs` array was defined as a device array, `lhs` will be a GPU array and the content from the CPU array `rhs` will be copied over the PCIe bus to GPU memory.

The above methods of data transfer are all blocking transfers, in that control is not returned to the CPU thread until the transfer is complete. This is sufficient in many cases, but prevents the possibility of overlapping data transfers with computation on both the host and device. The CUDA API function `cudaMemcpyAsync()` and its variants can be used to perform asynchronous transfers between host and device which allows concurrent computation.

**Kernels.** Kernels, or subroutines that are executed on the device, are denoted using the `attributes(global)` function attribute. Kernels are typically invoked in host code just as any subroutine is called, with the exception that an additional *execution configuration* specifying the number of thread blocks and number of threads per thread block to be used is included. In the device code itself, the automatically defined variables `threadIdx`, `blockIdx`, `blockDim`, and `gridDim` can be used to map threads to data elements. Aside from this, kernel code looks similar to the subroutines in the host code. The difference is that the kernel code is executed by many threads in parallel.

**CUF Kernels.** CUDA Fortran can automatically generate and invoke kernel code from a region of host code containing tightly nested loops. Such code is referred to as a CUF kernel. A simple example of a CUF kernel is:

```
!$cuf kernel do <<<*,*>>>
 do i=1, n
  a_d(i) = a_d(i) + b
 enddo
```

where the directive indicates that the following loop has to be performed on the device. One can specify the execution configuration in the chevrons. In the example above we use wild-cards and let the runtime system determine these parameters. The arrays in CUF kernels, such as `a_d` above, are required to be device arrays; however, the scalar `b` can be a host variable which will be passed as a kernel argument by value.

One can port host code to the device using CUF kernels without modifying the contents of the loops using the following programming convention. If the arrays used in the loops are declared in a module, along with a device equivalent:

```
module m
 ...
 real :: a(n)
 real,device :: a_d(n)
 ...
end module
```

then the rename option to the `use` statement can be invoked to allow conditional execution of the code either on the host or device:

```
subroutine update
#ifdef USE_CUDA
 use m, only: a => a_d
#else
 use m, only: a
#endif
...
!$cuf kernel do <<<*,*>>>
 do i=1, n
  a(i) = a(i) + b
 enddo
...
```

If the arrays used in the loops are explicitly passed to the subroutine, the only change required is to add the `device` attribute:

```
subroutine update(a,n)
real:: a(n)
#ifdef USE_CUDA
attributes(device) :: a
#endif
...
!$cuf kernel do <<<*,*>>>
 do i=1, n
  a(i) = a(i) + b
 enddo
...
```

Note that here the contents of the loop are unaltered. The only changes to the host code are the conditional renaming of module variables or the additional `device` attribute and the CUF kernel directive. The directive will appear as a comment to the compiler if GPU code generation is disabled or if the compiler does not support them (similar to the OpenMP directives that are ignored if OpenMP is not enabled).

## 3 Profiling Using NVTX

Profiling is an essential tool to identify parts of the code that may require additional tuning. When dealing with GPU codes, profiling is even more important as new opportunities for better interactions between the CPUs and the GPUs can be discovered. The standard profiling tools in CUDA, `nvprof` and `nvvp`, are able to show the GPU timeline but do not present CPU activity. The NVIDIA Tools Extension (NVTX) is a C-based API (application program interface) to annotate the profiler time line with events and ranges and to customize their appearance and assign names to resources such as CPU threads and devices [10]. We have written a Fortran module to instrument CUDA/OpenACC Fortran codes using Fortran ISO C bindings [3]. Using this module is very simple: once the NVTX module is included, the developer only needs to mark the region of interest with `nvtxRangePush` and `nvtxRangePop` calls. Calls to `nvtxStartRange("text")` with a single argument will insert green markers with a *text* label in the timeline. Different colors can be selected using an optional integer parameter and the regions of interest can be nested.

Since QE already has a built-in performance report that summarizes the time spent in the important parts of the code, we added the NVTX calls to the timing functions. This allowed a minimal code change.

To eliminate profiling overhead during production runs, we use a preprocessor variable to make the profiling calls return immediately. During the runs, one or more MPI processes generate the traces that are later imported and visualized with `nvvp`, the NVIDIA Visual Profiler.

Figure 1 shows a typical output for a PWscf run (when the mouse rolls over the markers, it will indicate the name of the marker and information on the kernel configurations).

## 4 Structure of the PWscf Code

As noted in the introduction, QE is not a monolithic program but a modular suite of codes sharing common libraries and data structures. The two major packages that are the foundation of every material science simulation work-flow are PWscf (Plane-Wave Self-Consistent Field) and `CP` (Car-Parrinello).

In this GPU porting effort, PWscf has been the main focus. The basic computations of the PWscf code involve the calculation of the Kohn-Sham (KS) orbitals and energies for isolated or extended/periodic systems and the complete structural optimizations of the microscopic (atomic coordinates) and macroscopic (unit cell) degrees of freedom. The KS orbitals are quantum-mechanical states of electrons under an effective Kohn-Sham potential. The solution is *self-consistent*: the KS potential depends upon the KS orbitals via the charge density (the sum of the square moduli of Kohn-Sham orbitals). This non-linear problem can be solved with an iterative procedure (see [6], Appendix A.2). Figure 2 illustrates the main activities performed in a typical execution of PWscf, where both high-level structural optimization and self-consistency [8] are explored.

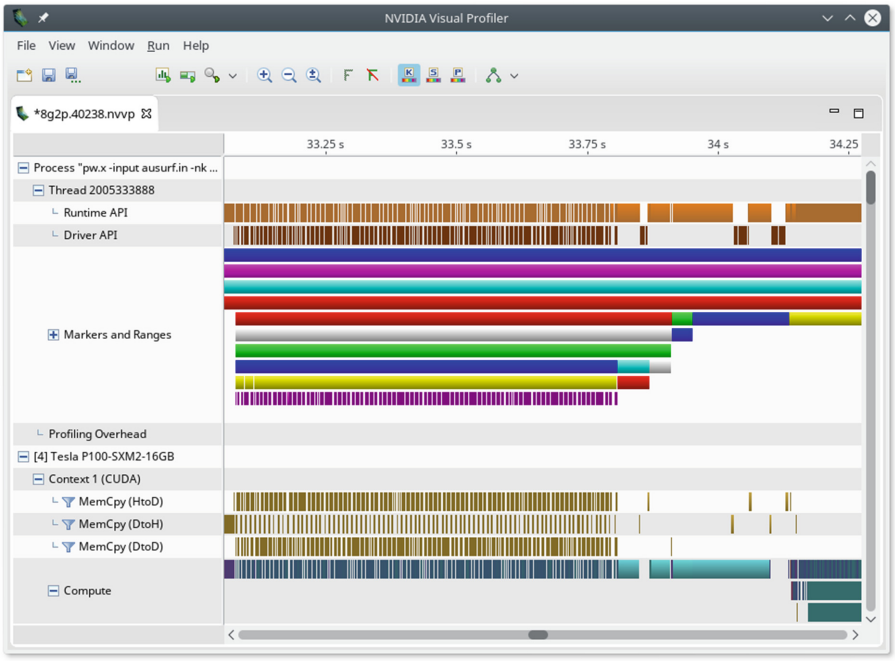


**Fig. 1.** Segment of `nvvp` output for AUSURF112 case on the DGX-1 system with 8 GPUs and no GPUDirect (GDR) features enabled. "Markers and Ranges" section contains colored markers corresponding to various NVTX ranges.

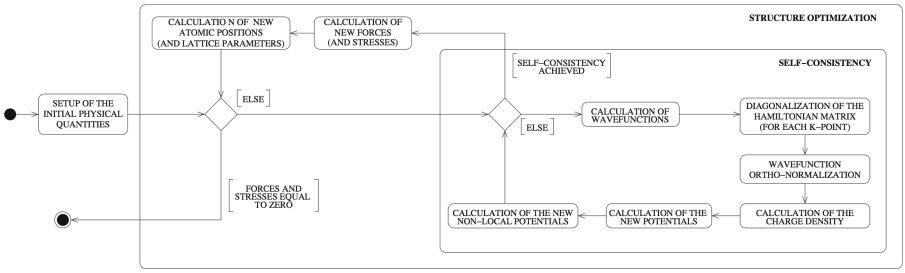


**Fig. 2.** Schematic view of PWscf internal steps.

In a plane-wave basis set, each KS orbital, $\psi$, is represented by a vector of plane-wave coefficients. The self-consistency loop is an iteration over the charge density, until input and output charge densities are the same within a predefined threshold. The output charge density is computed from KS orbitals, obtained by diagonalizing the matrix of the Hamiltonian operator, $H_{KS}$, which depends on the KS potential. By default, iterative diagonalization is completed using a block Davidson method. The calculation of the charge density requires all

occupied KS orbitals in the system. In a crystal, KS orbitals are classified by a Bloch vector, or "$k$-point", and by a "band" index. In practice, a discrete number of $k$-points, ranging from one to a few tens or hundreds at most, is needed [5]. The diagonalization is separately performed for each $k$-point. The number of occupied KS orbitals is determined by the number of electrons in the unit cell.

The iterative diagonalization and computation of charge density account for the majority of the time spent in the solver, with the remaining cost attributed to initialization and post processing routines. In the iterative diagonalization, the time-consuming step is the direct calculation of products $H_{KS}\psi$. Note that those products are *not* computed as matrix-vector products: the $H_{KS}$ matrix would be far too large for all but the simplest systems. Using the so-called dual-space technique, all computationally expensive terms can be expressed in terms of the following basic operations:

- 3-dimensional Fast Fourier Transforms (FFT);
- basic linear-algebra operations on vectors and matrices, in particular matrix-matrix multiplications (Level-3 BLAS);
- dense matrix diagonalization (LAPACK or ScaLAPACK).

The code offers a number of run-time options that affect the parallelization and enable distributed operation. A list of options used in this study includes:

- *k-point parallelization* using `-npool`: distributes the $k$-points into $N_K$ pools, allowing embarrassingly parallel execution of the iterative diagonalizations. If $N$ is the total number of MPI processes, there are $N_P = \frac{N}{N_K}$ processes per pool.
- *linear-algebra parallelization* using `-ndiag`: distributes the solution of the subspace diagonalization, needed by the block Davidson algorithm, to $N_D \leq N_P$ processes, enabling usage of ScaLAPACK or similar distributed linear solver library.

These options can be applied simultaneously, resulting in a wide array of possible combinations, not all valid or equally effective. The $k$-point parallelization takes precedence, splitting all available processes into equal pools. Within each pool, plane waves are distributed (this is also referred as *plane-wave* or *g-parallelism*). This distribution of plane waves across multiple MPI processes results in the need to perform parallel distributed 3D FFTs in order to transform physical quantities (KS orbitals, charge density and potentials) between reciprocal and real space. The FFT grids are generally of modest size (with dimensions in the hundreds); however, the FFT computation is repeated many times throughout the course of the calculation.

## 5 GPU Porting of Key Routines

While the full GPU porting effort involved the translation of a number routines in the original CPU code to GPU either by the use of CUF directives or CUDA kernels, we focus our discussion here on the routines that are considered most

performance critical. Without delving too deep into the specifics, it is informative to breakdown the major components of the PWscf iteration and identify the key computational operations involved. The iterative diagonalization involves the heavy use of three main computational components: a dense generalized eigensolver to diagonalize the subspace projected linear system, double-precision complex GEMMs which are mostly used to process the approximated eigenvalues and eigenvectors and expand the basis, and distributed forward and inverse 3D FFTs used in the procedure to compute the local potential term in $H_{KS}\psi$ for each unconverged band using the dual-space technique. The computation of the symmetrized charge density is dominated by the accumulation of wave-function contributions to the charge from each $k$-point which involves numerous distributed forward 3D FFT computations, one for each band.

Of the operations identified, the matrix-matrix multiplications are the most straightforward and can be easily computed on GPU using the CUBLAS library. The porting effort of the other computational components is more involved and requires further discussion.

### 5.1 Forward and Inverse 3D FFTs

Forward and inverse 3D FFTs are required in both the iterative diagonalization process and the computation of charge. As such, they account for a large share of the total computational load. While the component-wise 1D FFT computations can be carried out on GPU using simple calls to the CUFFT library, the complete computation is typically distributed among a number of processes, requiring transposition and communication of data across processes.

Currently, QE uses a 1D decomposition of the domain to distribute the 3D FFTs. With this decomposition, a typical 3D FFT computation of dimension $NX \times NY \times NZ$, distributed across $N_P$ processes in the pool, is completed in the following steps:

1. Begin with contiguous columns of data along $z$-dimension. Each process contains a $NX/N_P \times NY \times NZ$ sized chunk of the domain. Perform 1D FFTs on the $z$-columns.
2. Transpose result into planes representation via `MPI_Alltoall` or similar communication pattern. After communication, each process contains a $NX \times NY \times NZ/N_P$ sized chunk of the domain.
3. Perform 2D FFTs on the $xy$-planes.

This process also occurs in reverse within the solver, but the forward description is sufficient for this discussion.

The existing CPU implementation of this distributed 3D FFT procedure is fairly basic, with a few characteristics making a direct translation to GPU low performing. The first of these characteristics is that the FFT computation is carried out in a loop over bands, with relatively small FFT computations for each band. These small FFT computations are problematic on GPUs due to the lack of available concurrent work to fully saturate the GPU resources,

leading to inefficient device utilization and possible losses due to latency. The second characteristic is that the existing procedure does not make any attempt to overlap MPI communication with computation. This is especially problematic for a GPU implementation where, when direct peer-to-peer access between GPUs is unavailable, MPI communication buffers must be staged through CPU memory. Therefore, in addition to efficiency losses due to non-overlapped MPI communication, there are additional losses attributed to data movement of the communication buffers between host and device memory.

To address these issues, a new batched FFT strategy was implemented for GPUs which processes the 3D FFTs for several bands together. By processing multiple 3D FFTs at a time, there is naturally more concurrent work available to fully saturate the GPUs which addresses the first issue with the original implementation. In addition to this, further separation of batches into smaller sub-batches yields an opportunity for pipelining data movement and computation between sub-batches which we leveraged in our implementation. As a further optimization, the all-to-all communication pattern was carried out using non-blocking `MPI_Isend` and `MPI_Irecv`. This is of particular importance on GPU systems with fully-connected subsets of GPUs via NVLink, like the DGX-1, where numerous peer-to-peer transfers can occur simultaneously via GPUDirect (GDR). A simple method to enable these concurrent peer-to-peer transfers is through the use of a CUDA-aware MPI distribution. With that being said, early experimentation indicated that several issues arise in a number of available MPI implementations of these features, leading to suboptimal utilization of available peer-to-peer bandwidth on systems with numerous peer-to-peer links. To address this, an explicit handling of peer-to-peer communication was implemented using CUDA inter-process communication (IPC) features, with non-peer transfers handled by the linked MPI library. Lastly, by finely controlling the all-to-all communication, self-to-self buffer transfers on the GPU can be handled specifically to avoid any unneeded use of host resources. It should be noted that batching the FFT computation does increase memory requirements, as multiple FFT domains must be resident in device memory. For the benchmark cases tested in this study, this was not a limiting factor; however, for larger cases, the batch size can be adjusted to fit within available memory.

### 5.2 Solving the Eigenproblem

The final major computational component to discuss is the dense eigensolver, which is used to solve the subspace projected problem generated through the Davidson iteration process. In the existing CPU implementation, the dense eigensolve can either be computed sequentially, using one process in a $k$-point pool group, or distributed across $N_D$ processes in the pool group using ScaLAPACK or a similar distributed linear algebra package.

The initial GPU port targets only the serial path, using a custom developed GPU eigensolver. A custom solver was chosen in lieu of several existing GPU-enabled eigensolvers, like those available in MAGMA [7]. The custom GPU eigensolver was developed to specifically limit dependencies on CPU resources, using

the CPU only for the solution of a reduced tridiagonal eigensystem using available functionality from Intel MKL or other LAPACK implementations. This is in contrast to implementations available in MAGMA, where many more operations are offloaded to the CPU, with a complex pipelining of CPU computation, GPU computation, and data movement between the host and device. This is especially beneficial on "fat" GPU nodes, nodes with a high ratio of GPU to CPU sockets, where available CPU resources (host memory bandwidth, PCIe bandwidth between host and device, available CPU FLOPS) per GPU can be limited. By limiting the use of CPU resources, the custom eigensolver can achieve more consistent performance across these types of node topologies, with less sensitivity to available CPU resources per GPU. Even with node topologies with one full CPU socket available per GPU, limiting these CPU dependencies has been shown to improve performance of the custom solver relative to MAGMA and MKL [15].

While only the serial eigensolver path has been ported, the results of several benchmark cases to be discussed in later sections will show that our custom eigensolver, even operating on a single GPU, provides competitive performance relative to high-performance distributed CPU solvers, like the ELPA solver [1].

## 6 Performance Comparison

Performance results were obtained on a number of GPU systems ranging in size from a small workstation containing only two GPUs up to several large GPU accelerated clusters, with reference CPU performance results obtained on a private development cluster.

The reference CPU system (labeled "Broadwell" in the results) is a private development system of a few hundred nodes fully based on Intel technology. Each node has dual socket 18-core Intel Xeon E5-2697 v4 (Broadwell) CPUs, 128 GB of system memory and one single Intel Omni-Path interconnect to provide 100 Gb/s connectivity for both parallel jobs and I/O.

The small systems used in this study were a workstation with a 6-core Intel Core i7-5930K CPU with two 16 GB NVIDIA P100 GPUs and an NVIDIA DGX-1 system. The DGX-1 contains dual socket 20-core Intel Xeon E5-2698 v4 (Broadwell) CPUs with eight 16 GB NVIDIA P100 GPUs, with fully-connected clusters of four GPUs with NVLink associated with each CPU socket.

The large GPU systems used in this study were Piz Daint at the Swiss National Supercomputing Centre (CSCS), SummitDev at the Oak Ridge National Laboratory (ORNL) and Wilkes-2 cluster at the University of Cambridge.

Piz Daint is a Cray XC50 with 5,272 nodes, each with a 12-core Intel Xeon E5-2690 v3 (Haswell) CPU, 64 GB of system memory and a 16 GB NVIDIA P100 GPU. The network uses Aries routing and communications ASICs and a dragonfly network topology. Piz Daint is currently number three on the June 2017 Top500 list with 19.59PF and is one of the most efficient petaFLOP class machines in the world: in the Green 500 list published in June 2017, the machine was able to achieve 10398 MFLOP/s/W with level 3 measurements, the most accurate available.

The SummitDev system is an early access system that is one generation removed from ORNL's next big supercomputer, Summit. The system has 54 IBM POWER8 S822LC nodes. Each node has dual socket IBM POWER8 CPUs, each with 10 cores and 80 HW threads, 256 GB of system memory, and four 16 GB NVIDIA P100 GPUs, with two NVLink connected GPUs per socket. In contrast to the Intel based systems, the GPUs on SummitDev are connected to the CPUs by NVLink 1.0 at 80 GB/s. The nodes are connected in a full fat-tree via EDR InfiniBand. SummitDev has access to Spider 2, the OLCF's center-wide Lustre parallel file system, and also local NVMe disks.

Wilkes-2 is a new GPU cluster at the University of Cambridge composed of 90 Dell PowerEdge C4130 compute nodes. Each node has a single socket 12-core Intel Xeon CPU E5-2650 v4 (Broadwell) CPU, 96 GB of system memory and four 16 GB NVIDIA P100 GPUs all connected to the same PCIe root complex. One single Mellanox Infiniband EDR card provides 100 Gb/s connectivity for both parallel jobs and access to the Lustre storage. Wilkes-2 is completely based on commodity hardware and it is currently number 100 on the June 2017 Top500 list with 1.193 PF and number 5 on the Green500 list with 10428 MFLOP/s/W.

### 6.1 Performance Analysis

**Benchmark Cases and Details.** For testing, two benchmark test cases were used which span a range of typical use cases for the PWscf solver. The cases used were:

- AUSURF112: computation of a surface of 112 gold atoms with two $k$-points. Small case suitable for testing on workstations and small distributed systems.
- Ta2O5: computation of tantalum pentoxide with 96 atoms and 26 $k$-points. Large case suitable for scaling from small to large distributed systems.

Detailed input specifications for these benchmark cases can be found in Table 1.

For cases run on GPU systems with Intel CPUs, multithreaded MKL was used for any BLAS and LAPACK routines computed using the CPU, including the tridiagonal eigensolve offloaded from the custom GPU eigensolver. On SummitDev, multithreaded ESSL was used in place of MKL; however, due to the lack of a linkable implementation of `ZSTEDC`, the CPU tridiagonal eigensolver routine we require for our GPU eigensolver, the program was linked against the LAPACK implementation provided with PGI, with underlying BLAS routines computed using ESSL.

For all runs on the reference CPU system, the eigenproblem is solved using the distributed ELPA library, with $N_D$ set to the closest square number to half the available MPI processes per pool group. Note that the number of available MPI processes per pool group is reduced if OpenMP threads are enabled. The results on the reference CPU system reported are the best-case results achieved using a variety of possible configurations of OpenMP threads and $N_D$ values.

For all runs on the GPU systems, $N_D$ is always set to one since only the serial eigensolver path was ported to GPU. For systems using Intel CPUs, OpenMP

**Table 1.** Benchmark case input specifications

| Parameter | Benchmark case | |
|---|---|---|
| | AUSURF112 | Ta2O5 |
| Number of atomic species | 1 | 2 |
| Number of atoms | 112 | 96 |
| Number of electrons | 1,232 | 544 |
| Number of Kohn-Sham states | 739 | 326 |
| Number of k-points | 2 | 26 |
| Number of plane waves | 100,747 | 477,247 |
| Kinetic energy cutoff | 25 Ry | 130 Ry |
| Charge density cutoff | 200 Ry | 520 Ry |
| Dimension of dense FFT grid | {180, 90, 288} | {198, 168, 220} |

threading was enabled to improve the offloaded CPU tridiagonal eigensolve using multithreaded MKL; as such, threads were distributed so that a larger portion of available cores were bound to processes within pool groups performing the serial eigensolve. On SummitDev, a similar thread distribution strategy was utilized with multi-threaded ESSL; however, OpenMP was disabled elsewhere in the code due to existing compatibility issues between the PGI and IBM OpenMP runtimes.

On GPU systems with available peer-to-peer connections between GPUs, the test cases were run both with and without using GPUDirect (GDR) features. For all communication except the all-to-all in the distributed FFTs, these features were enabled implicitly through the use of CUDA-aware MPI distributions, typically Open MPI or Cray MPICH on Piz Daint. On SummitDev, due to poor performance of the CUDA-aware features of Spectrum MPI, all MPI communication is staged through the host. For the all-to-all communication, peer-to-peer transfers were handled explicitly using our explicit CUDA IPC implementation, with non-peer transfers handled by the linked MPI library.

**Results and Discussion.** Performance results for the AUSURF112 test case can be found in Table 2, with timing breakdowns for the cases run with 4 GPUs or CPUs and cases run with 8 GPUs or CPUs plotted in Figs. 3 and 4 respectively. For accuracy considerations, the final converged total energy results on the reference CPU system for this test case were within the range −11427.08997421 Ry to −11427.08997363 Ry. This compares well with the converged total energy results obtained on the GPU systems, which ranged from −11427.08997417 Ry to −11427.08997388 Ry.

Similarly, performance results for the Ta2O5 test case across the tested systems can be found in Table 3, with timing breakdowns for the cases using 8 GPUs or CPUs, 104 GPUs or CPUs, and 208 GPUs or CPUs plotted in Figs. 5, 6 and 7 respectively. For this test case, the final converged total energy results

**Table 2.** PWscf time in seconds for AUSURF112 testcase

| System | $N_K$ | Number of CPUs or GPUs used | | | | |
|---|---|---|---|---|---|---|
| | | 2 | 4 | 8 | 16 | 32 |
| Broadwell (CPU) | 1 | 1142.24 | 642.03 | 369.66 | 272.00 | 266.20 |
| | 2 | 1190.13 | 586.84 | 335.00 | 196.54 | **144.07** |
| Piz Daint | 1 | 286.24 | 219.91 | 171.80 | – | – |
| | 2 | – | 149.21 | **115.87** | – | – |
| DGX-1 | 1 | 347.82 | 271.37 | 210.67 | – | – |
| | 2 | – | 184.10 | 142.15 | – | – |
| DGX-1, GDR | 1 | 270.21 | 190.12 | 174.75 | – | – |
| | 2 | – | 142.43 | **100.54** | – | – |
| Summit Dev | 1 | 321.69 | 234.32 | 187.69 | – | – |
| | 2 | – | 176.50 | 128.85 | – | – |
| Summit Dev, GDR | 1 | 308.52 | 227.74 | 188.39 | – | – |
| | 2 | – | 169.60 | **124.22** | – | – |
| Wilkes-2 | 1 | 395.26 | 326.71 | 227.61 | – | – |
| | 2 | – | 226.89 | 167.80 | – | – |
| Wilkes-2, GDR | 1 | 300.03 | 226.13 | 203.59 | – | – |
| | 2 | – | 164.63 | **116.50** | – | – |
| Workstation | 1 | 334.23 | – | – | – | – |
| Workstation, GDR | 1 | **279.54** | – | – | – | – |

on the reference CPU system were within the range −2370.63541806 Ry to −2370.63541801 Ry. This also compares well with the converged total energy results obtained on the GPU systems, which ranged from −2370.63541805 Ry to −2370.63541804 Ry.

Considering the tabulated performance results in Tables 2 and 3, several observations can be made. First, across most results for this case, it can be noted that for a fixed number of CPU or GPU resources, increasing $N_K$ provides a performance improvement. This indicates that the program on both CPU and GPU is more efficiently utilizing compute resources when there are fewer resources assigned per pool. If the program scaled perfectly with the number of resources per pool, the PWscf time, assuming the computation outside the scope of the pool parallelization is negligible, should remain nearly fixed if the number of pools is doubled. This is because the doubling of performance associated with processing more $k$-points concurrently would be counteracted by a halving in performance due to halving the number of compute resources per pool.

This reduction in efficiency can largely be attributed to the scaling characteristics of the distributed 3D FFT computations and the eigensolver. This can be observed clearly in the timing breakdowns plotted in Figs. 3, 4, 5, 6 and 7 when comparing the results for the different $N_K$ values on each system.

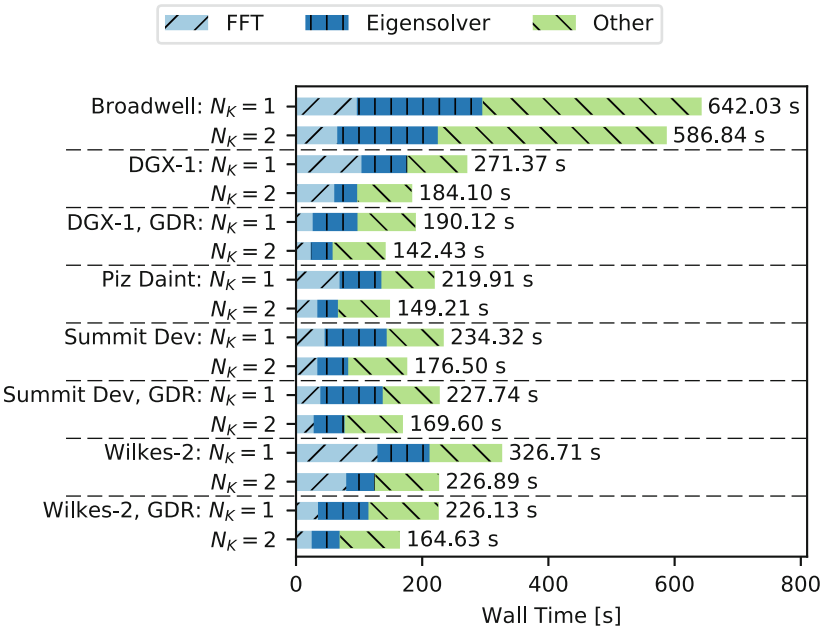


**Fig. 3.** Breakdown of PWscf time for AUSURF112 using 4 GPUs or CPUs by system and pool size.

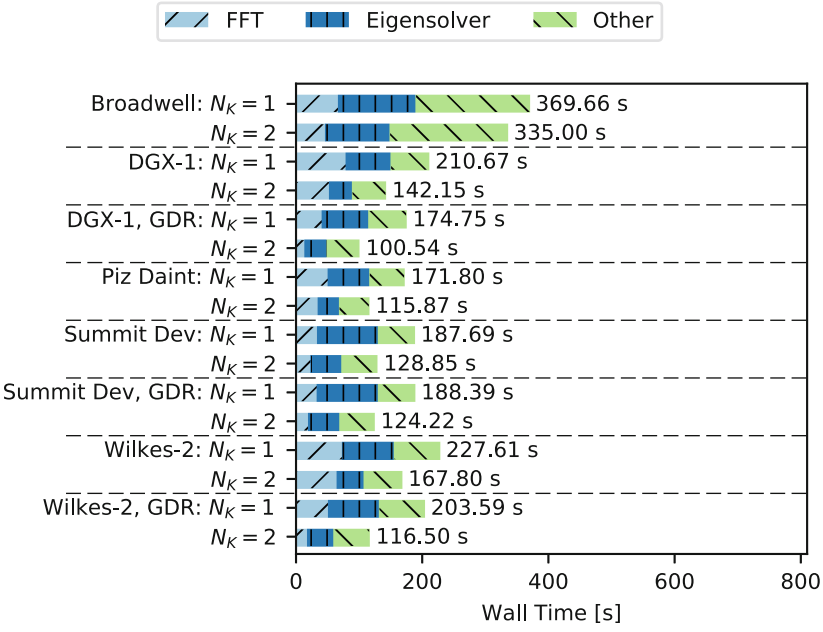


**Fig. 4.** Breakdown of PWscf time for AUSURF112 using 8 GPUs or CPUs by system and pool size.

**Table 3.** PWscf time in seconds for Ta2O5 testcase

| System | $N_K$ | Number of CPUs or GPUs used | | | | |
|---|---|---|---|---|---|---|
| | | 8 | 26 | 52 | 104 | 208 |
| Broadwell (CPU) | 13 | – | – | 1374.26 | 809.36 | 540.64 |
| | 26 | – | 3055.46 | 1566.95 | 682.05 | **378.73** |
| Piz Daint | 1 | 5273.93 | – | – | – | – |
| | 2 | 3602.07 | – | – | – | – |
| | 13 | – | – | 617.58 | 419.39 | 330.85 |
| | 26 | – | – | – | 315.60 | **217.29** |
| DGX-1 | 1 | 7253.06 | – | – | – | – |
| | 2 | 5008.94 | – | – | – | – |
| DGX-1, GDR | 1 | 4139.18 | – | – | – | – |
| | 2 | 2701.00 | – | – | – | – |
| Summit Dev | 1 | 4122.03 | – | – | – | – |
| | 2 | 3236.12 | – | – | – | – |
| | 13 | – | – | 581.15 | 394.62 | 289.30 |
| | 26 | – | – | – | 305.66 | 216.95 |
| Summit Dev, GDR | 1 | 3994.21 | – | – | – | – |
| | 2 | 2959.70 | – | – | – | – |
| | 13 | – | – | 544.83 | 398.91 | 292.87 |
| | 26 | – | – | – | 284.90 | **207.37** |
| Wilkes-2 | 1 | 7394.40 | – | – | – | – |
| | 2 | 6111.83 | – | – | – | – |
| | 13 | – | – | 1035.20 | 656.85 | – |
| | 26 | – | – | – | 515.78 | – |
| Wilkes-2, GDR | 1 | 5032.51 | – | – | – | – |
| | 2 | 3264.26 | – | – | – | – |
| | 13 | – | – | 572.43 | 460.16 | – |
| | 26 | – | – | – | **273.86** | – |

First, on both the CPU and GPU systems, the distributed FFT operates more efficiently when distributed across fewer processes. This is because distribution across fewer processes on the tested systems generally resulted in improved all-to-all communication performance. One reason for this performance improvement is that a smaller group of processes can maintain better locality, resulting in a larger percentage of communication occurring over higher bandwidth intra-node connections, either within local CPU memory, or through more direct NVLink or PCIe connections when using GDR features on GPU systems. In addition to this, with fewer processes, the self to self buffer involved in the all-to-all, which is a fast local memory movement, comprises a larger portion of the total communication volume.

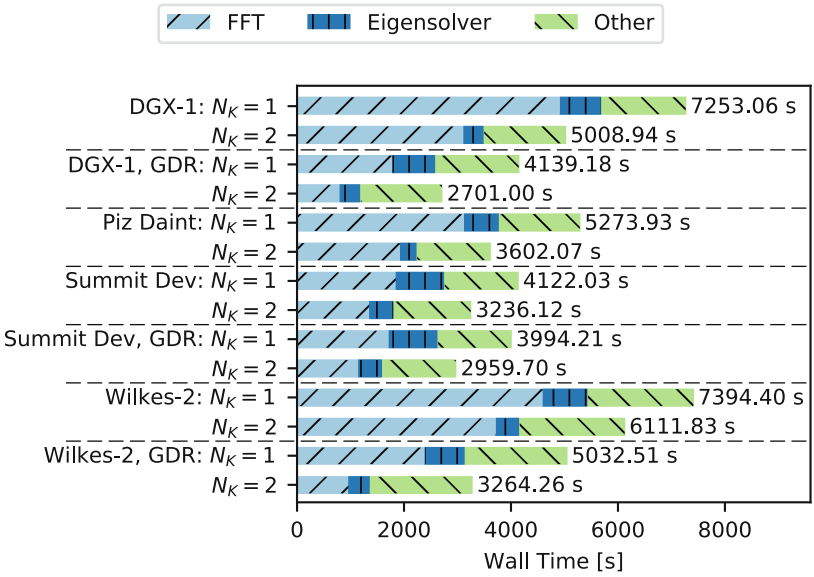


**Fig. 5.** Breakdown of PWscf time for Ta2O5 using 8 GPUs or CPUs by system and pool size.

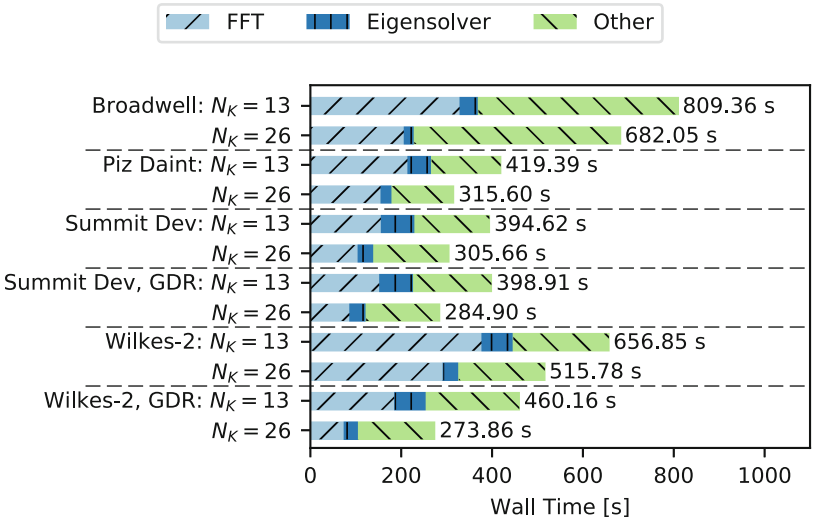


**Fig. 6.** Breakdown of PWscf time for Ta2O5 using 104 GPUs or CPUs by system and pool size.

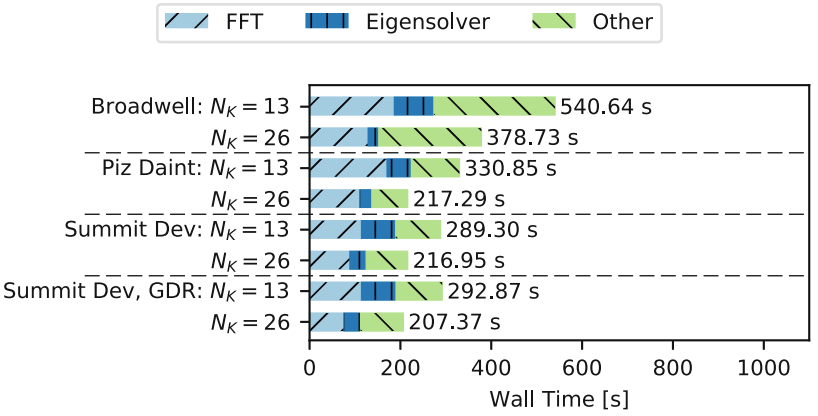


**Fig. 7.** Breakdown of PWscf time for Ta2O5 using 208 GPUs or CPUs by system and pool size.

Continuing on this point, the benefits of using GDR on the GPU systems with available peer-to-peer access can be substantial, with improved performance in most cases on systems with GDR features utilized. As expected, systems with more available peer-to-peer links between GPUs, like the DGX-1 and Wilkes-2 with fully connected clusters of four GPUs, benefit the most from these features; in contrast, SummitDev, which has only connected pairs of GPUs, benefits less in this case. Comparing plots in Figs. 3, 4, 5, 6 and 7 with and without GDR enabled indicates that the use of GDR primarily improves the performance of the distributed 3D FFTs. Additionally, it can be noted that on the DGX-1 and Wilkes-2, the FFT performance improves dramatically when the number of pools results in pool groups with four GPUs, where all communication within the all-to-all occurs over peer-to-peer connections.

Considering the eigensolver on the CPU, the scaling behavior aligns more closely with what is expected, with a small edge in efficiency when distributed across fewer processes. On the GPU systems, due to the use of a serial eigensolver, increasing the number of pools from one to two results in a halving of the eigensolve time. Since the serial eigensolver is always computed using a single GPU per pool group, the eigensolve time scales proportionally with the number of pools. This trend can be observed in Figs. 3, 4, 5, 6 and 7 on all the GPU systems. Consequently, for a given number of pools, the eigensolver performance will remain fixed regardless of the number of GPUs assigned to the pool group, leading to some loss in efficiency. Despite this, the serial GPU eigensolver outperforms the distributed ELPA library used on the reference CPU system for the AUSURF112 cases, while maintaining competitive performance in the Ta2O5 cases.

Comparing the reference CPU system to the GPU system results, the GPU systems are outperforming the reference CPU system in all tested configurations, when comparing single CPU socket performance to single GPU performance,

with relative speedups ranging from 2 to 4. Figures 3, 4, 5, 6 and 7 illustrate where to attribute these gains in performance. In all cases, a large portion of the improvement can be attributed to faster ZGEMM and DGEMM performance on the GPU systems. This is clear, since on the GPU systems, the GEMM dominated portion of the runtime outside of the FFT and eigensolve is significantly reduced on the GPU systems relative to the reference CPU system. Beyond this, additional performance improvements of varying degree can be attributed to the FFT and eigensolver.

Comparing GPU system results, there is some observed variability in the performance between the systems, which can be attributed to differences in node topology (how many GPUs are associated with each CPU socket and how are they connected) and node architecture (IBM POWER8 with host-to-device connections via NVLink compared to Intel Xeon with host-to-device connections via PCIe). As a first example, the slowest GPU system results occur on DGX-1 and Wilkes-2 when GDR features are disabled. These two systems have the highest ratio of GPUs to CPU sockets, with each system having four GPUs per CPU socket. In addition to this, the GPUs on these system share PCIe lanes, with two GPUs per PCIe root complex on the DGX-1, and four GPUs per PCIe root complex on Wilkes-2. Thus, with GDR disabled, the all-to-all communication during the distributed 3D FFTs become bottlenecked by a lack of PCIe bandwidth for transfer of communication buffers between the host and device and CPU memory bandwidth to handle all the MPI traffic. With GDR features enabled however, these bottlenecks are alleviated due to the substantial increase in device-to-device bandwidth offered via peer-to-peer connections, freeing up the CPU to handle only out of socket MPI traffic. This results in these systems showing the highest performance of all the systems tested when GDR features are enabled, demonstrating the importance of exploiting these peer-to-peer connections when possible.

On a related note, due to higher memory bandwidth offered by the POWER8 CPU and greater host-device bandwidth through NVLink, SummitDev is less impacted by these issues, leading to high distributed FFT performance even without GDR. The higher host-to-device bandwidth also gives SummitDev an improvement in distributed FFT performance over Piz Daint, due to faster transfer of communication buffers between host and device.

While SummitDev maintains an edge in the distributed FFT performance in non-GDR enabled cases, the eigensolver performance on this system lags behinds that of the other GPU systems. As a generic LAPACK implementation of the offloaded tridiagonal eigensolver was used for this system, the benefits of multi-threading from ESSL was limited to the underlying BLAS calls, leading to a loss in performance relative to a fully multi-threaded implementation. Otherwise, the eigensolver performance is generally more consistent across the GPU systems using Intel CPUs with MKL, even with a varied number of cores available to the GPUs performing the eigensolve.

## 7 Conclusions

This paper presented development details and performance of PWscf on CPU and GPU systems. The new GPU version produces accurate results and can reduce the time-to-solution by an average factor of 2–3 relative to a reference CPU system.

The custom GPU eigensolver developed for this code is very competitive with both ScaLAPACK and ELPA, with little sensitivity to available host resources. Improvements to performance via distribution over multiple GPUs and removing existing CPU dependencies are being considered for future development.

The performance results in this study illustrate the importance of exploiting peer-to-peer connectivity between GPUs when available, implicitly via CUDA-aware MPI or explicitly using CUDA IPC or similar mechanisms. These features, when properly utilized, can provide a substantial performance boost, particularly on systems with high GPU to CPU socket ratios. The upcoming generation of NVIDIA GPUs, Volta, with a faster memory subsystem and double precision performance higher than 7 TeraFLOP/s, will help push the performance of this code even further.
The code is available for download at https://github.com/fspiga/qe-gpu.

**Acknowledgments.** This research used resources of the Oak Ridge Leadership Computing Facility at the Oak Ridge National Laboratory, which is supported by the Office of Science of the U.S. Department of Energy under Contract No. DE-AC05-00OR22725. This work was also supported by a grant from the Swiss National Supercomputing Centre (CSCS) under project ID g33. Wilkes-2 is part of the *Cambridge Service for Data Driven Discovery* (CSD3) system operated by the University of Cambridge Research Computing Service funded by EPSRC Tier-2 capital grant EP/P020259/1, the STFC DiRAC HPC Facility (BIS National E-infrastructure capital grant ST/K001590/1, STFC capital grants ST/H008861/1 and ST/H00887X/1, Operations grant ST/K00333X/1) and the University of Cambridge. CSD3 and DiRAC are part of the UK National e-Infrastructure. Paolo Giannozzi also acknowledges support from the European Union through the MaX Centre of Excellence (Grant No. 676598).

## References

1. Auckenthaler, T., Blum, V., Bungartz, H.J., Huckle, T., Johanni, R., Krämer, L., Lang, B., Lederer, H., Willems, P.R.: Parallel solution of partial symmetric eigenvalue problems from electronic structure calculations. Parallel Comput. **37**(12), 783–794 (2011)
2. Blackford, L.S., Choi, J., Cleary, A., D'Azeuedo, E., Demmel, J., Dhillon, I., Hammarling, S., Henry, G., Petitet, A., Stanley, K., Walker, D., Whaley, R.C.: ScaLAPACK User's Guide. Society for Industrial and Applied Mathematics (1997)
3. Fatica, M.: Customize CUDA Fortran Profiling with NVTX (2015). https://devblogs.nvidia.com/parallelforall/customize-cuda-fortran-profiling-nvtx
4. Fatica, M., Ruetsch, G.: CUDA Fortran for Scientists and Engineers. Morgan Kaufmann, Burlington (2014)
5. Froyen, S.: Brillouin-zone integration by Fourier quadrature: special points for superlattice and supercell calculations. Phys. Rev. B **39**, 3168–3172 (1989)

6. Giannozzi, P., Baroni, S., Bonini, N., Calandra, M., Car, R., Cavazzoni, C., Ceresoli, D., Chiarotti, G.L., Cococcioni, M., Dabo, I., et al.: QUANTUM ESPRESSO: a modular and open-source software project for quantum simulations of materials. J. Phys. Condensed Matter **21**(39), 395502 (2009)
7. Dongarra, J., Gates, M., Haidar, A., Kurzak, J., Luszczek, P., Tomov, S., Yamazaki, I.: Accelerating numerical dense linear algebra calculations with GPUs. In: Kindratenko, V. (ed.) Numerical Computations with GPUs, pp. 3–28. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-06548-9_1
8. Johnson, D.D.: Modified Broyden's method for accelerating convergence in self-consistent calculations. Phys. Rev. B **38**, 12807–12813 (1988)
9. Kohn, W.: Fundamentals of density functional theory. In: Joubert, D. (ed.) Density Functionals: Theory and Applications, pp. 1–7. Springer, Heidelberg (1998). https://doi.org/10.1007/BFb0106731
10. Kraus, J.: CUDA Pro Tip: generate custom application profile timelines with NVTX (2013). https://devblogs.nvidia.com/parallelforall/cuda-pro-tip-generate-custom-application-profile-timelines-nvtx
11. Marek, A., Blum, V., Johanni, R., Havu, V., Lang, B., Auckenthaler, T., Heinecke, A., Bungartz, H.J., Lederer, H.: The ELPA library: scalable parallel eigenvalue solutions for electronic structure theory and computational science. J. Phys. Condensed Matter **26**(21), 213201 (2014)
12. Message Passing Interface Forum: MPI: A Message-Passing Interface Standard, Version 2.2. Technical report (2009). http://www.mpi-forum.org/docs/mpi-2.2/mpi22-report.pdf
13. Parr, R.G., Yang, W.: Density-Functional Theory of Atoms and Molecules (International Series of Monographs on Chemistry). Oxford University Press, New York (1994)
14. Pickett, W.E.: Pseudopotential methods in condensed matter applications. Comput. Phys. Rep. **9**(3), 115–197 (1989)
15. Romero, J.: Developing an Improved Generalized Eigensolver with Limited CPU Offloading. In: GPU Technology Conference, San Jose, CA (2017). http://on-demand.gputechconf.com/gtc/2017/presentation/s7388-joshua-romero-developing-an-improved-generalized-eigensolver.pdf
16. Spiga, F.: Plug-in code to accelerate Quantum ESPRESSO v5 using NVIDIA GPU. https://github.com/fspiga/qe-gpu-plugin
17. Spiga, F.: Implementing and testing mixed parallel programming model into Quantum ESPRESSO. In: Science and Supercomputing in Europe - Research Highlights 2009, CINECA Consorzio Interuniversitario, Bologna, Italy (2010)
18. Spiga, F., Girotto, I.: phiGEMM: a CPU-GPU library for porting Quantum ESPRESSO on hybrid systems. In: 2012 20th Euromicro International Conference on Parallel, Distributed and Network-based Processing, pp. 368–375 (2012)