ELPA: A Parallel Solver for the Generalized Eigenvalue Problem[1]

Hans-Joachim BUNGARTZ [a], Christian CARBOGNO [b], Martin GALGON [c], Thomas HUCKLE [a,2], Simone KÖCHER [a], Hagen-Henrik KOWALSKI [b], Pavel KUS [d], Bruno LANG [c], Hermann LEDERER [d], Valeriy MANIN [c], Andreas MAREK [d], Karsten REUTER [a], Michael RIPPL [a], Matthias SCHEFFLER [b] and Christoph SCHEURER [a]

[a] *Technical University of Munich*
[b] *Fritz Haber Institute, MPG*
[c] *University of Wuppertal*
[d] *Max Planck Computing and Data Facility*

Abstract. For symmetric (hermitian) (dense or banded) matrices the computation of eigenvalues and eigenvectors $Ax = \lambda Bx$ is an important task, e.g. in electronic structure calculations. If a larger number of eigenvectors are needed, often direct solvers are applied. On parallel architectures the ELPA implementation has proven to be very efficient, also compared to other parallel solvers like EigenExa or MAGMA. The main improvement that allows better parallel efficiency in ELPA is the two-step transformation of dense to band to tridiagonal form. This was the achievement of the ELPA project. The continuation of this project has been targeting at additional improvements like allowing monitoring and autotuning of the ELPA code, optimizing the code for different architectures, developing curtailed algorithms for banded A and B, and applying the improved code to solve typical examples in electronic structure calculations. In this paper we will present the outcome of this project.

Keywords. ELPA-AEO, eigensolver, parallel, electronic structure calculations

1. Introduction

The ELPA-AEO project is a continuation of the ELPA project [1,2] where mathematicians, computer scientists, and users collaborate in order to develop parallel software for the Generalized Symmetric Eigenvalue Problem (GSEP) $AX = BX\Lambda$. The project partners are the Max-Planck Computing and Data facility in Garching, the University of Wuppertal, the Departments of Computer Sci-

[1]This work was supported by the Federal Ministry of Education and Research within the project "Eigenvalue soLvers for Petaflop Applications – Algorithmic Extensions and Optimizations" under Grant No. 01IH15001.

[2]Corresponding author; E-mail: huckle@in.tum.de

ence and of Chemistry of the Technical University of Munich, and the Fritz Haber Institute in Berlin. The former ELPA project developed a basic parallel GSEP solver and provided the software library `https://elpa.mpcdf.mpg.de/about`. Objective of the follow-up ELPA-AEO project is to include useful tools like monitoring and automatic performance tuning, to optimize the software for certain architectures, and to develop a special solver for banded GSEP.

The ELPA GSEP solver works in the following way:

- compute the Cholesky factorization $B = U^H U$ and the related standard eigenvalue problem (SEP) $\tilde{A}\tilde{X} = \tilde{X}\Lambda$ with $\tilde{A} = U^{-H} A U^{-1}$;
- reduce the SEP in $\tilde{A}$ directly (ELPA1) or with an intermediate banded matrix (ELPA2) to tridiagonal form;
- apply a divide-and-conquer solver to the tridiagonal SEP;
- transform the derived eigenvectors back to the original GSEP according to the previous steps.

In electronic structure computations a whole sequence of eigenproblems has to be solved with changing A.

In the following sections we will present the performance improvements included in ELPA-AEO, namely

- monitoring, autotuning, and optimization;
- improved matrix multiplication in the transformations via Cannon's algorithm;
- taking advantage of banded structure in A and B via Crawford's method;
- solving huge important GSEPs in electronic structure computations.

2. Optimization, Monitoring, and Autotuning

Clearly the most obvious change in the recent ELPA releases from the user perspective is the complete redesign of the library API. The new API requires the user to first create the ELPA object, then allows various manipulations with it in order to influence the library performance and finally to call one of the solution routines. Examples (shortened) of a program using ELPA in Fortran and C can be seen in Figures 1 and 2, respectively. The new API brought many benefits for the library users, whilst keeping the user-code changes on very reasonable level. Not only are the calling commands more elegant, but many new options and functionalities have been implemented. One of the most important is the introduction of autotuning. An example code, showing a possible use of this functionality is shown in Figure 3.

The autotuning works as follows. First of all, a set of parameters that should be tuned is selected (either by choosing the level of autotuning or manually). Each of the parameters can attain a limited number of values (e.g. all the different kernel implementations, or different values of certain block sizes, etc.). To alleviate the user from the need to wait too long and to avoid the necessity of wasting the valuable computer time, the autotuning can be performed during the production run with repeating calls (e.g. during the SCF cycle) to the solution routine, each time with one of the possible parameter combinations with the possibility to

```
use elpa
class(elpa_t), pointer :: e
integer :: success
e => elpa_allocate(success)
if (success \= ELPA_OK) ... !handle error
! set the matrix size
call e%set("na", na, success)
if (success \= ELPA_OK) ... !checks further omitted
! set in the same way all the required parameters
! describing the matrix and its MPI distribution.
call e%set("nev", nev, success)
call e%set("local_nrows", na_rows, success)
call e%set("local_ncols", na_cols, success)
call e%set("nblk", nblk, success)
call e%set("mpi_comm_parent", mpi_comm_world, success)
call e%set("process_row", my_prow, success)
call e%set("process_col", my_pcol, success)
success = e%setup()
! if desired, set other run-time options
call e%set("solver", elpa_solver_2stage, success)
! values of parameters can be retrieved
call e%get("stripewidth_real", stripewidth, success)
! call one of the solution methods
! the data types of a, ev, and z determine whether
! it is single/double precision and real/complex
call e%eigenvectors(a, ev, z, success)
! or, in the case of generalized EVP
call e%generalized_eigenvectors(a, b, ev, z, ...
    is_already_decomposed, success)
! cleanup
call elpa_deallocate(e)
call elpa_uninit()
```

Figure 1. Example use of the ELPA object. In the old API, all parameters were passed in one function call, which, with increasing number of customization parameters and options, became too inflexible and error prone since the signature of the function became too long and each newly introduced parameter would change the library API. With the new API, arbitrary large number of parameters can be added in the future. A new API for generalized EVP has been added, allowing the user to specify, whether he or she has already called the function with the same matrix B (using the is_already_decomposed parameter) and wants to re-use its factorizations, which is useful during the SCF cycle.

interrupt and resume the process and finally to store the optimal setting for future use, as it is suggested in Figure 3.

Apart from the previously mentioned changes, a lot of effort has been put into classical HPC optimizations of the code with respect to different architectures. This includes optimizations for the new CPU architectures, GPUs and interconnects. One of the recent HPC architectures, where ELPA has been successfully deployed is the supercomputer cobra at MPCDF, which comprises of skylake-based compute nodes, partially equipped with NVIDIA Volta V100 GPUs and the OmniPath interconnect. The performed optimizations included writing hand-tuned AVX-512 kernels (using compiler intrinsics), addressing MPI performance issues (finally solved by using Intel MPI 2019.3 or higher) and various GPU-related optimizations.

```c
#include <elpa/elpa.h>
elpa_t handle;
handle = elpa_allocate(&error);
elpa_set(handle, "na", na, &error);
elpa_get(handle, "solver", &value, &error);
printf("Solver is set to %d \n", value);
elpa_eigenvectors(handle, a, ev, z, &error);
elpa_deallocate(handle);
elpa_uninit();
```

Figure 2. Example use of the C interface. The object-oriented approach is implemented using the handle pointer. Apart from this, the library use is very similar as through the Fortran interface (as presented in Figure 1), and the C example is thus kept very short for brevity.

As ELPA originated as a replacement for the ScaLAPACK routines P?SYEVR and P?SYEVD, it is natural to compare its performance with the best available implementation of this widely used and de-facto standard library for a given architecture, as it has been done in the past ([1], [3]) on Intel-based machines and also recently by independent authors in [4] using the Cray system. Such comparison can be seen in Figure 4, comparing the performance of the ELPA library with Intel MKL 2019.5 for a matrix of the size 20000. Scaling curves for larger matrices including a cross-island run can be seen in Figure 5. It is obvious, that the performance of the ELPA library, especially its implementation of the two-stage algorithm, exceeds the performance of the MKL routines significantly, as it is consistent with other reports.

A lot of effort has been put into GPU related optimizations of ELPA, since the number of GPU-equipped HPC systems is on the rise. We have already reported this effort and the obtained results in the previous papers [5] and [3], so let us here only present a typical performance output (see Table 1) and reiterate some conclusions:

- ELPA 1-stage can run significantly faster using GPUs, which is not the case for ELPA 2-stage, where the speed-up is moderate to none at the moment.
- In order to benefit from the GPUs, there has to be enough data to saturate them. It is thus beneficial to use them for setups, where there are large local matrices (possibly up to the memory limits), thus for large matrices and/or moderate number of GPU equipped nodes.
- To fully utilize both the CPUs and GPUs, ELPA is run as a purely MPI application with one MPI rank per core and the efficient use of the GPU cards is achieved through the NVIDIA MPS daemon.

We can thus conclude (see Table 1), that the GPU implementation of ELPA 1-stage is utilizing the GPUs well and given a suitable problem setup, it can be very efficiently used to reduce the total application runtime.

3. Reduction of Full Generalized Eigenvalue Problems

The solution of a GSEP $AX = BX\Lambda$ with A hermitian and B hermitian positive definite typically proceeds in four steps.

```
use elpa
class(elpa_t),           pointer :: e
class(elpa_autotune_t), pointer :: tune_state
e => elpa_allocate()
! set all the required fields, omitting others
call e%set("na", na, error)
! alternatively exclude some parameters from autotuning by ...
    setting them
call e%set("gpu", 0)
!set up the ELPA object and create the autotuning object
success = e%setup()
tune_state => e%autotune_setup(level, domain, error)

if(done_with_autotuning) then
  call e%load_all_parameters("autotuned_pars.txt")
elseif(autotuning_in_progress) then
  call e%autotune_load_state(tune_state, "atch.txt")
endif
iter=0
! application-specific cycle, where multiple similar
! EVP problems are solved, e.g. the SCF cycle
do while (continue_calculation)
  if(.not. done_with_autotuning) &
    finished = .not. e%autotune_step(tune_state)
  if(finished) then
    ! set and print the autotuned-settings
    call e%autotune_set_best(tune_state)
    ! the current values of the parameters can be saved
    call e%save_all_parameters("autotuned_pars.txt")
    done_with_autotuning = .true.
  endif
  ! do the actual calculation
  call e%eigenvectors(a, ev, z, error)
  ! do whatever needed with the result
end do
if (.not. done_with_autotuning) then
  ! the status of the autotuning can be saved
  call e%autotune_save_state(tune_state, "atch.txt")
endif
! de-allocate autotune object
call elpa_autotune_deallocate(tune_state)
```

Figure 3. A sketch of a code, which performs autotuning during a production run of a program which calls the ELPA library repeatedly. It also shows how to split the autotuning process into multiple calls of the program by saving the autotuning state into a checkpoint file atch.txt. Each actual library call is performed with slightly different settings. After all combinations have been exhausted, the optimal settings are saved to the autotuned_pars.txt file, the autotuning is not performed any more and the optimal setting is used ever since.

i) Compute Cholesky decomposition $B = U^H U$.
ii) Reduce the GSEP to an equivalent SEP $\tilde{A}\tilde{X} = \tilde{X}\Lambda$, where $\tilde{A} = U^{-H} A U^{-1}$.
iii) Solve the SEP.
iv) Back-transform the eigenvectors via $X = U^{-1}\tilde{X}$.

Since one key application of ELPA is electronic structure theory, where often a

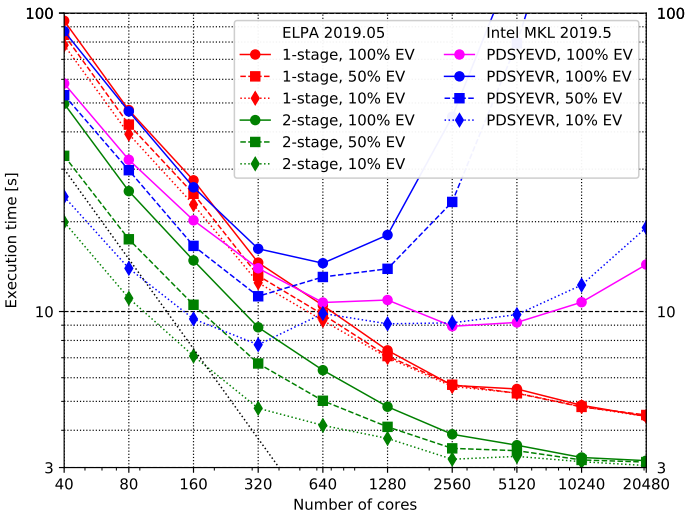

Figure 4. Scaling results from the most recent skylake-based supercomputer at MPCDF using real double precision matrix of the size 20000. Compared is the performance of the two relevant MKL 2019.5 routines and ELPA one and two stage. Where possible, also results for 10% and 50% of eigenvectors are shown. The MKL routines offer comparable or superior performance to the ELPA one-stage algorithm for small and moderate number of cores, but do not scale for larger core counts. ELPA scaling is generally much better in the investigated region. For the very large core counts and thus very small local matrices, the speed-up with growing number of cores is slowing down, but the performance is not deteriorating, which can be very beneficial when coupled with a well scaling application. The ELPA two-stage solver clearly outperforms all the other routines in this setup.

sequence of GSEPs $A^{(k)}X^{(k)} = BX^{(k)}\Lambda^{(k)}$ *with the same matrix* B have to be solved during a *self consistent field* (SCF) cycle, ELPA's approach for the above step ii) is to explicitly compute B^{-1} and then to do (triangular) matrix multiplications to obtain $\tilde{A}$. Alternative approaches use the inverse only implicitly; cf. the routines `PDSYNGST` and `TwoSidedTrsm` in the ScaLAPACK [6] and ELEMENTAL [7] libraries, resp.

With the inverse U^{-1} available explicitly (again upper triangular, denoted as $\hat{U}$ in the following), a computationally efficient way to implement the above step ii), $\tilde{A} = \hat{U}^H A \hat{U}$, is as follows [8].

ii.a) Compute the upper triangle M_u of $M := A\hat{U}$.
ii.b) Transpose M_u to obtain the lower triangle M_l of $M^H = \hat{U}^H A^H = \hat{U}^H A$.
ii.c) Compute the lower triangle of $\tilde{A} = M_l \hat{U}$.
ii.d) If the whole matrix $\tilde{A}$ is needed then reflect its lower triangle along the diagonal.

During the ELPA-AEO project, new algorithms have been developed for the multiplications in steps ii.a) (*Multiplication 1*: compute upper triangle of "hermitian × upper triangular") and ii.c) (*Multiplication 2*: compute lower triangle of "lower triangular × upper triangular"). Compared to these multiplications, the transpositions in steps ii.b) and d) are inexpensive [9].

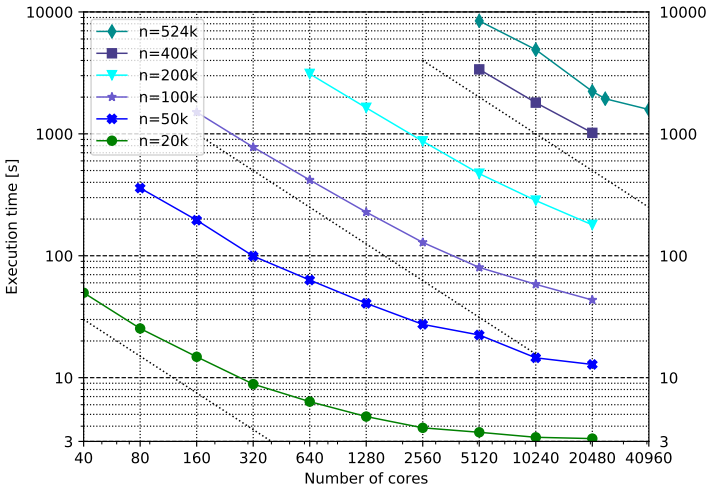

Figure 5. Strong scaling graphs for ELPA 2-stage computing all eigenvectors with different matrix sizes n. The line for n=20k corresponds to the line of the same color from Figure 4. The results shown in both figures were obtained on the supercomputer cobra at MPCDF, comprising of compute nodes containing two Intel Xeon Gold 6148 processors (Skylake with 20 cores (each) at 2.4 GHz) connected through a 100 Gb/s OmniPath interconnect. Most of the calculations shown were performed within a single island with a non-blocking, full fat tree network topology. The blocking factor among islands is 1:8. The only cross-island run is for the largest matrix n=524k and 40960 cores showing reasonable performance despite the weaker network between the islands.

Table 1. ELPA runtimes (s) on a full Skylake node (40 cores in total) equipped with two NVIDIA Volta V100 GPUs. As it is usually the case, ELPA is running as purely MPI application (thus using 40 MPI ranks). In the GPU case, each of the MPI ranks is offloading compute intensive kernels to one of the GPUs (through the NVIDIA MPS for efficiency). As it can be seen from the results, even using one particular architecture, it is not possible to determine the generally best option. In this particular case, ELPA 1-stage CPU is the best option for very small matrices, ELPA 2-stage CPU for larger and ELPA 1-stage GPU for the largest. The ELPA 2-stage GPU is not listed, since its performance is almost never the best possible and is thus currently not recommended.

	CPU		**GPU**
matrix size	ELPA 1	ELPA 2	ELPA 1
1024	**0.11**	0.13	0.93
8192	10.7	**5.57**	8.45
20000	110	52.7	**37.0**
65536	5795	2551	**733**

Our algorithms are based on Cannon's method [10]; they exploit the triangular structure to save on arithmetic operations and communication, and they have been extended to work on non-square $p_r \times p_c$ grids with integer aspect ratio $p_c : p_r$. In this case, they take p_r phases, which improves over the p_c phases of the approach described in [11] for full matrices.

Here we only point out the main ideas for Multiplication 1. Assume that

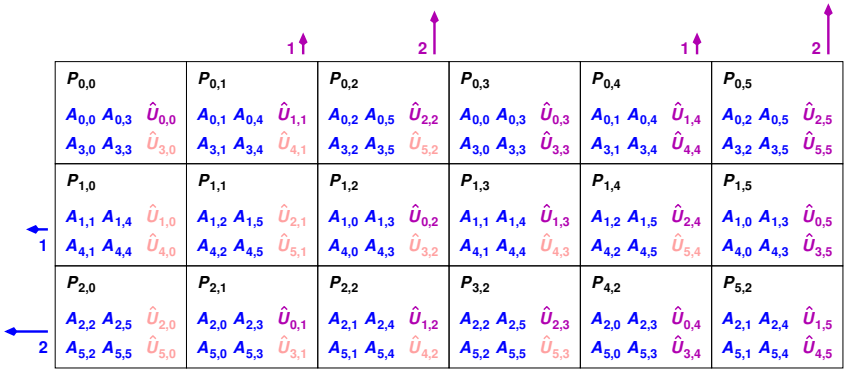

Figure 6. Distribution of the matrices for Multiplication 1 *after* the initial skewing and the sharing of A blocks. The numbers next to the arrows indicate the distance of the skewing shifts within rows/columns of the process grid. See the main text for a description.

$A \in \mathbb{C}^{n\times n}$ and $\hat{U} \in \mathbb{C}^{n\times n}$ have been partitioned into $N \times N$ blocks $A_{i,j}$ ($\hat{U}_{i,j}$, resp.) of size $n_b \times n_b$, where $N = \lceil n/n_b \rceil$, and that they are distributed over the process grid in a *block torus wrapped* manner, i.e., process $P_{k,\ell}$ holds exactly those blocks $A_{i,j}$ and $\hat{U}_{i,j}$ such that $i \equiv k \mod p_r$ and $j \equiv \ell \mod p_c$. This is also the default distribution in ScaLAPACK and ELPA. Considering the case $N = 6$, $p_r = 3$, $p_c = 6$ as an example (cf. also Figure 6), process $P_{1,5}$ would hold the blocks $A_{1,5}$, $A_{4,5}$, $\hat{U}_{1,5}$, and $\hat{U}_{4,5}$. Next we do a Cannon-type *initial skewing*: In row k of the process grid, $k = 0, \ldots, p_r - 1$, the local portions of A are shifted by k positions to the left, and in column ℓ, $\ell = 0, \ldots, p_c - 1$, the local portions of $\hat{U}$ are shifted by $\ell \mod p_r$ positions upwards (with cyclic connections along rows and columns). Therefore, $P_{1,5}$ now has $P_{1,5+1} \equiv P_{1,0}$'s original blocks from A (i.e., $A_{1,0}$ and $A_{4,0}$) and $P_{1+2,5} \equiv P_{0,5}$'s original blocks from $\hat{U}$ (i.e., $\hat{U}_{0,5}$ and $\hat{U}_{3,5}$). Finally, groups of p_c/p_r processes that are p_r positions apart in the same row, share their portion of A. In our example, $P_{1,5}$ shares the A blocks with $P_{1,2}$, such that both hold the same blocks $A_{1,0}$, $A_{4,0}$, $A_{1,3}$, $A_{4,3}$ from A, but different blocks from $\hat{U}$, cf. Figure 6. Note that the blocks $\hat{U}_{i,j}$ in the strict lower triangle of $\hat{U}$ are zero and therefore need not be stored and sent; they bear a light color in Figure 6.

After these preparations, the computation proceeds in p_r phases. In each phase, every process multiplies its current local A with the current local $\hat{U}$. In our example, taking into account the structure of $\hat{U}$ and the fact that we compute only the lower triangle of the product $M = A\hat{U}$, in the first phase $P_{1,5}$ would update $\begin{bmatrix} M_{1,5} \\ M_{4,5} \end{bmatrix} = \begin{bmatrix} M_{1,5} \\ M_{4,5} \end{bmatrix} + \begin{bmatrix} A_{1,0} & A_{1,3} \\ A_{4,0} & A_{4,3} \end{bmatrix} \cdot \begin{bmatrix} \hat{U}_{0,5} \\ \hat{U}_{3,5} \end{bmatrix}$, whereas the update in $P_{1,3}$ reads $\begin{bmatrix} M_{1,3} \\ M_{4,3} \end{bmatrix} = \begin{bmatrix} M_{1,3} \\ M_{4,3} \end{bmatrix} + \begin{bmatrix} A_{1,1} \\ A_{4,1} \end{bmatrix} \cdot \hat{U}_{1,3}$, and $P_{1,1}$ performs no computation at all in this phase. At the end of each phase, the local A blocks are shifted by one position to the left in the process grid, and the $\hat{U}$ blocks are shifted by one position up. It is not hard to verify that, after p_r such phases, $P_{1,5}$ has computed "its" blocks

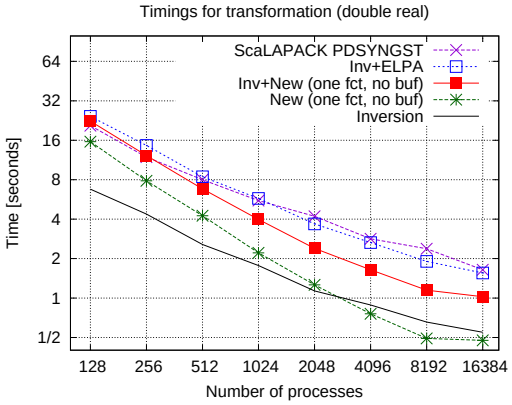

Figure 7. Timings on HYDRA for the complete transformation $A \to \tilde{A}$ with the ScaLAPACK routine `PDSYNGST` (one step, inverse of the Cholesky factor used only implicitly) and with the "invert and multiply" approach (multiplication routines from ELPA or the new Cannon-based implementations). $n = 30{,}000$, $n_b = 64$, 16 single-threaded processes per node.

$M_{1,5}$ and $M_{4,5}$ of the block torus wrapped-distributed product M, and similarly for the other $P_{k,\ell}$.

For a description of Multiplication 2 and a discussion of possible savings from combining the two multiplications in one function and buffering some data the reader is referred to [12].

In Figure 7 we present timings obtained on the HYDRA system at the Max Planck Computing and Data Facility in Garching. Each HYDRA node contains two 10-core Intel Ivy Bridge processors running at 2.8 GHz. All matrices were double precision real of size $n = 30{,}000$, and the block size was $n_b = 64$. We observe that explicit inversion, combined with our Cannon-based matrix multiplications, can be highly competitive even for solving a single generalized eigenproblem (red curve, including the time for inverting U). For sequences of GSEPs with the same B, where the inversion can be skipped in most cases, the new reduction according to steps ii.a) to d) is significantly faster (green curve).

In [12] we have considered only MPI parallelization, using p processes for utilizing a total of p cores. Alternatively, one can reduce the number of processes and enable multithreaded execution. This may or may not be beneficial, depending on several factors. In particular, while multithreading reduces the size of the process grid and therefore leads to savings in communication, it also can cause a loss of computational performance if running a process' computations with φ threads does not speed them up by a factor of φ.

In the left picture of Figure 8 we see that using multiple threads per process may extend the range of scalability. More details are exposed in the right picture, which shows the relative timings for the same runs. We see that for numbers of cores p that are not squares and therefore would lead to a non-square grid when using single-threaded processes, using 2 threads per process (leading to a square process grid) reduced the time for multiplication 1 by roughly 10%, whereas it increased the time for those p that are square and not very large. This indicates

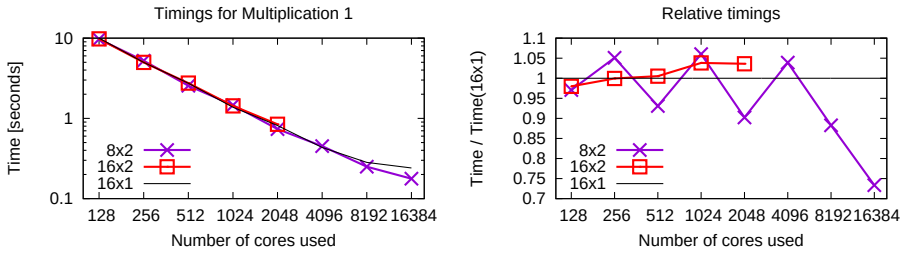

Figure 8. (Left) Time of Multiplication 1 on HYDRA for $n = 30{,}000$, $n_b = 64$, if 16 cores per node are used for 16 single-threaded processes per node ("16×1"), 8 processes with 2 threads each ("8×2"), and 16 processes with 2 threads each (hyperthreading, "16×2"). (Right) Times relative to the baseline of 16 single-threaded processes per node.

a slight preference of that routine for square process grids. If the number of cores is large enough that communication contributes significantly to overall time then multithreading pays independently of the grid's shape because it leads to a smaller process grid and therefore reduces communication.

The effect of multithreading also differs between the routines in the ELPA library; see Table 2. Some of them contain explicit OpenMP directives or pragmas for controlling thread parallelism. Others rely exclusively on multithreaded BLAS, and the efficiency of the latter depends on the size and shape of the involved matrices, which may be rather different; this is the case, e.g., with the `GEMM` calls in Multiplication 1 vs. Multiplication 2 [12].

A detailed discussion of these issues is not within the scope of this work, but note that even the decision whether to use multithreading for the complete solution of an eigenproblem (considering all routines involved) may depend on whether it is part of a whole sequence of eigenproblems, as common in SCF cycles, or just a single eigenproblem; cf. the last two lines in Table 2. See Section 2 on support in ELPA for taking such decisions in a partially or fully automated way.

Table 2. Timings (in seconds) on HYDRA for $n = 30{,}000$, $n_b = 64$, with different setups of 4096 cores (256 nodes with 16 cores each): 16 single-threaded processes per node, 8 processes with 2 threads, and 4 processes with 4 threads.

	16×1	8×2	4×4
Cholesky decomposition	1.804	1.078	0.878
Invert	0.885	0.804	0.781
New transformation	0.759	0.709	0.705
Solution of standard eigenproblem	8.189	8.668	8.758
Back-transformation 33%	0.247	0.235	0.203
Overall without Chol. & Invert	9.195	9.612	9.665
Overall including Chol. & Invert	11.883	11.494	11.324

4. Eigenvalue Solver for Banded Matrices

If, additionally, the two matrices A, $B = U^H U$ in the GSEP $AX = BX\Lambda$ are banded, the procedure described in Section 3 is not optimal as it leads to a full matrix $\tilde{A}$ (the Cholesky factor of B is still banded, but the inverse of the Cholesky factor is in general a full matrix and hence $\tilde{A}$ becomes a full matrix).
The two-stage solver in ELPA however, first transfers a full matrix C of a SEP $CY = Y\Lambda$ to a banded matrix $\hat{C}$ and then further transforms it to a tridiagonal matrix $\bar{C}$ which is solved for the eigenvalues and eigenvectors. Subsequently, the eigenvectors undergo two backtransformation steps to obtain the eigenvectors of the SEP. In this framework, by maintaining the band while transforming the GSEP to a SEP, the first step (transformation of the full matrix to the banded matrix) can be omitted as well as the second step of the backtransformation.

Crawford proposed an algorithm for maintaining the band in [13]. His algorithm stepwise applies the Cholesky factorization of B and removes the occurring fill-in outside the band by a series of QR factorizations. Lang extended the algorithm in [14]. His version offers more flexibility for blocksizes and bandwidth and utilizes a twisted factorization for B instead of a standard Cholesky factorization. The latter allows to reduce computational work when removing the occurring fill-in drastically. In the following we will briefly describe our parallel implementation of Lang's algorithm including the backtransformation of the eigenvectors. A more detailed description can be found in [15].

For the parallel implementation we use a unified blocksize $n_b = \max(b_A, b_B)$ (as in the original Crawford algorithm) to get an efficient pipelining algorithm. The matrices A and B can therefore be subdivided into $N \times N$ blocks with $N = \lceil \frac{n}{n_b} \rceil$. The case when n is not a multiple of n_b can be covered by adding an incomplete block at the end.

The matrix U originates from the twisted factorization of B with twist position p and twist block P (the twist position p is chosen such that it is the end of a block; this block is referred to as twist block). U can itself be factorized as

$$U = U_P \cdot U_{P-1} \cdots U_1 \cdot U_{P+1} \cdots U_{N-1} \cdot U_N.$$

Each of the factors U_i has the shape of an identity matrix besides one block row between the rows $(i-1)n_b + 1$ and in_b. These rows contain the same values as in the matrix U at the same place. Figure 9 gives an illustration of the matrix shapes and the block structure.
The transformation $\tilde{A} = U^{-H} A U^{-1}$ can therefore be reformulated to the stepwise application

$$\tilde{A} = U_P^{-H} \cdot U_{P-1}^{-H} \cdots U_1^{-H} \cdot U_{P+1}^{-H} \cdots U_{N-1}^{-H} \cdot U_N^{-H} \cdot$$
$$A \cdot U_N^{-1} \cdot U_{N-1}^{-1} \cdots U_{P+1}^{-1} \cdot U_1^{-1} \cdots U_{P-1}^{-1} \cdot U_P^{-1}.$$

As it can be seen from Figure 9 every U_i consists of one block row that differs from the identity matrix. In this block row, we will denote the diagonal block as $U_{i,i}$ and the other non-zero block as $U_{i,i-1}$ or $U_{i,i+1}$, depending on the position in the lower or upper matrix half. When inspecting the inverse of the matrix,

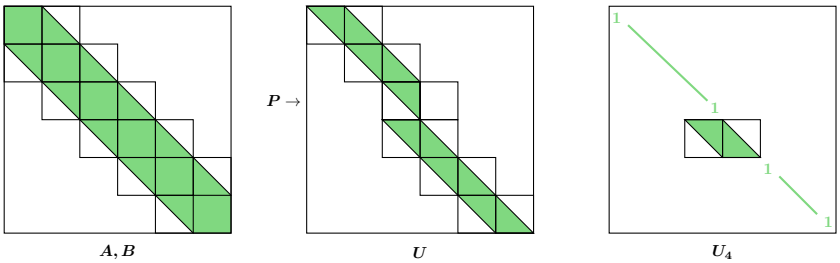

Figure 9. Block structure of the matrices A and B (left), the twisted factorization of B, U with its twist block P (middle), and one of its factors, U_4 (right).

U_i^{-1}, it can be seen that it has the same structure as U_i. The diagonal block of the inverse matrix turns out to be the inverse of the diagonal block of U_i. In the further text we use $D_i := U_{i,i}^{-1}$ as abbreviation for this block. The other block of the inverse matrix is denoted as E_i and can be described by $E_i := -D_i U_{i,i-1}$ in the lower matrix half or $E_i := -D_i U_{i,i+1}$ in the upper matrix half, respectively.

Applying one factorization step U_i in the lower matrix half hence takes the i-th block column of A, multiplies it from the right with E_i and adds it to block column $i-1$ (upper matrix half: block column $i+1$). Afterwards, block column i is multiplied from the right with D_i. The same procedure is rolled out for the multiplication from the left with U_i^{-H}. Block row i times E_i^H is added to block row $i-1$ (upper matrix half: block row $i+1$) and subsequently block row i is multiplied by D_i^H.

Figure 10 shows in the left picture the application of a factorization step in the lower matrix half and the occurring fill-in (left two block columns). The fill-in is created in the blocks $A_{i,i-1}$, $A_{i+1,i-1}$ and $A_{i+1,i}$. Due to symmetry we restrict the description to the lower triangle of the matrix A. On the blocks $A_{i,i-1}$ and $A_{i+1,i-1}$ a QR decomposition is computed and the block rows i and $i+1$ are multiplied with the obtained Q from the left as well as the block columns i and $i+1$ from the right. The symmetric application of Q shifts the fill-in by one block row and one block column towards the lower end. By repeating the QR step, the fill-in can be completely evicted from the matrix and the next factorization step can be applied. The procedure for the upper matrix half is the same, only the QR factorization is replaced by a QL factorization and the fill-in moves stepwise towards the top left of the matrix.

Denoting the Qs following the application of U_i with $Q_i^{(k)}$, the series $U_i, Q_i^{(1)}, Q_i^{(2)}, \ldots, Q_i^{(\nu_i)}$ applies one step of the factorization and restores the band. Hence, using

$$\tilde{U}^{-1} = U_N^{-1} \cdot Q_N^{(1)} \cdots Q_N^{(\nu_N)} \cdots U_{P+1}^{-1} \cdot Q_{P+1}^{(1)} \cdots$$
$$Q_{P+1}^{(\nu_{P+1})} \cdot U_1^{-1} \cdot Q_1^{(1)} \cdots Q_1^{(\nu_1)} \cdots U_P^{-1} \cdot Q_P^{(1)} \cdots Q_P^{(\nu_P)},$$

the overall transformation with restoring the band can be described as $\hat{A} = \tilde{U}^{-H} A \tilde{U}^{-1}$.

The eigenvalues of the SEP $\hat{A}\hat{X} = \hat{X}\Lambda$ are the same as the eigenvalues of the GSEP $AX = BX\Lambda$, but for the eigenvectors a backtransformation step has to be

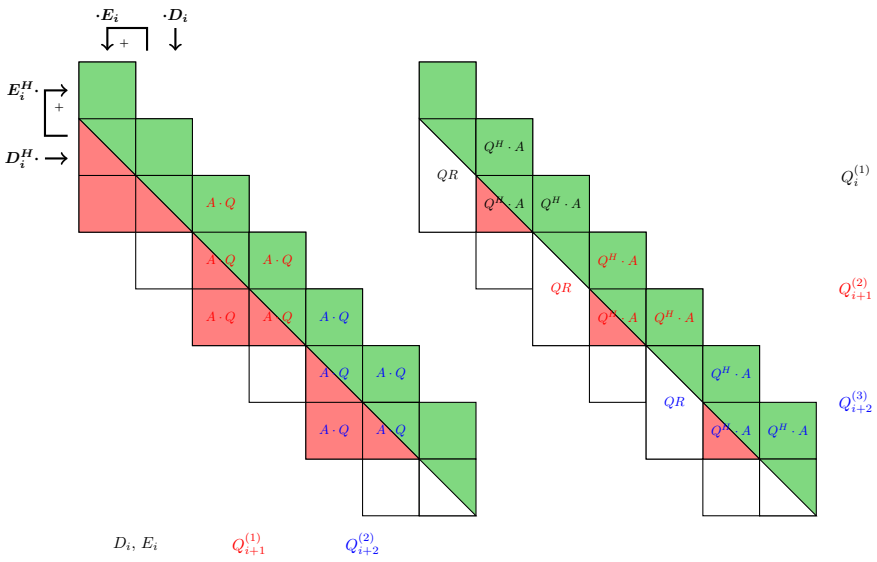

Figure 10. The two steps of the algorithm (in the lower matrix half, upper matrix half similar but mirrored): 1. (left) Applying the factorization (D_i, E_i) and performing the right sided update of the two-sided Q-application. Since operating on different block columns, Q of different factorization steps can be applied in parallel. Red indicates the bulge of newly created non-zeros outside the band (in green). 2. (right) Eliminating the fill-in by generating the QR decomposition and applying it from the left to the matrix. Since operating on different block rows, the QR decomposition and the left sided application of Q of different factorization steps can be applied in parallel.

applied. These eigenvectors of the GSEP can be found by applying $\tilde{U}^{-1}$ to the eigenvectors of the SEP: $X = \tilde{U}^{-1}\hat{X}$.

Contrary to the computation of $\hat{A}$ the eigenvectors are multiplied from the left with $\tilde{U}^{-1}$ and not with the hermitian of it. Therefore the order of the operations is reverse: $Q_i^{(\nu_i)}, \ldots, Q_i^{(2)}, Q_i^{(1)}, U_i$. Figure 11 gives an illustration of the updating scheme. In the lower matrix half the applications of $Q_i^{(k)}$ update the block rows $i+k-1$ and $i+k$. After having applied the Qs, D_i multiplies the block row i from the left and to this block row the $i-1$st block row multiplied from the left with E_i is added. The block rows to update in the upper matrix half are slightly different. $Q_i^{(k)}$ update the block rows $i-k+1$ and $i-k$ and instead of multiplying the $i-1$st block row with E_i, the $i+1$st block row is added to block row i (which has been multiplied by D_i).

Having a closer look on the application of the factorization and the generation and application of the Q, it can be seen that by splitting the two-sided application of Q and by interchanging the order of the $Q_i^{(k)}$ a pipelining structure can be obtained. It is based on the fact that a left sided update with $Q_i^{(k)}$ updates only two consecutive block rows and a right sided update and the application of the factorization only update two consecutive block columns. The order of execution has to be kept within a factorization step, meaning $Q_i^{(k)}$ has to be executed before $Q_i^{(k+1)}$, but $Q_i^{(k)}$ can be executed at the same time as $Q_{i+1}^{(k+1)}$. Details on the

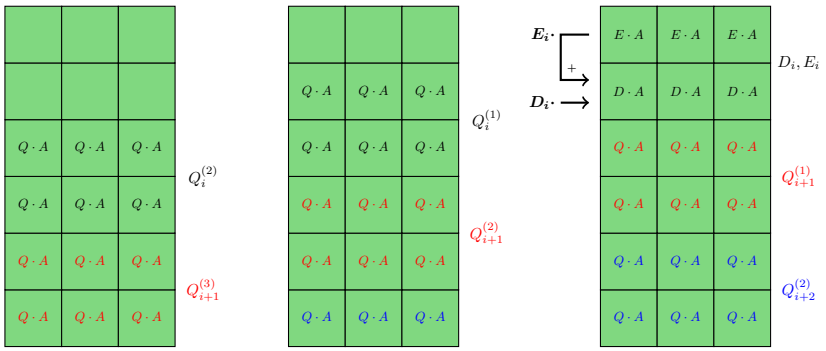

Figure 11. Three consecutive steps in the backtransformation: The applications of different Q can be done in parallel in the same way the Q have been created (see Figure 10, right picture). Finally, the D_i and E_i are applied.

interchangeability can be found in [15]. Figure 10 shows the pipelining structure in the lower matrix half in the computation of $\hat{A}$.

A similar pipelining scheme can be obtained for the backtransformation step: All $Q_{i+j}^{(k+j)}$ can be applied to the eigenvectors simultaneously. Additionally, the application of D_i and E_i can be decoupled from the application of the $Q_i^{(k)}$ and can be executed afterwards. Figure 11 shows the application of different $Q_i^{(k)}$ which can be executed concurrently.

Besides the pipelining scheme, $\tilde{U}^{-1}$ offers another parallelization layer which comes by the twisted factorization. The operations in $\tilde{U}^{-1}$ first process the lower matrix half and afterwards the upper matrix half. These operations, however, do not overlap besides the twist block P that is updated by the upper and the lower matrix half of the factorization. Therefore they can be run in parallel with a synchronization point at the twist block. Concluding, the algorithm provides three parallelization layers: parallel execution of the upper and lower matrix half, parallel execution of the independent steps in the pipeline and parallelization of the operations in the single blocks. Additionally, the use of a threaded BLAS library can provide a fourth layer of parallelization.

The process setup is hence chosen in a way to exploit the parallelization layers. The available processes are separated in processes for the upper and the lower matrix half. Processes of a matrix half are further subdivided into groups which compute the operations of a block. These groups are ordered in a grid and if not enough groups are available to fill all blocks, repeated cyclically. All operations involve communication between the processes of two groups. Due to the constant neighbourhood, local communicators are used to perform these operations efficiently. In the backtransformation step the process setup is used in the same way, exploiting to have the Householder vectors already in place.

Figures 12 and 13 show the strong scaling behaviour of the algorithm for matrix sizes of 51200 and 204800. The overall runtime as well as the two main steps are plotted: the backtransformation of the eigenvectors and the application of $\tilde{U}^{-1}$. The bandwidth of the matrices was in both cases 1% of the matrix size. Both matrix sizes show good scaling for a selected number of processes per group. If all groups only hold one block further speedup can be achieved by using more

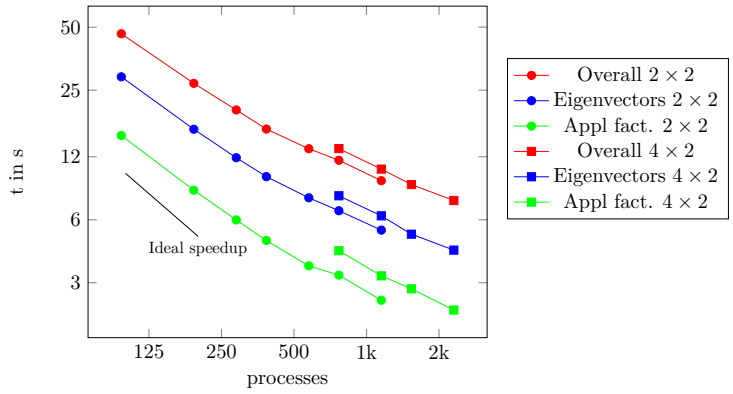

Figure 12. Strong scaling for a matrix of size 51200. The bandwidth (and hence the blocksize) is 512, the twist index is at 25600. The backtransformation is done for 12800 eigenvectors (25%). Per group 2×2 and 4×2 processes have been used. The runs have been carried out on the Cobra Supercomputer.

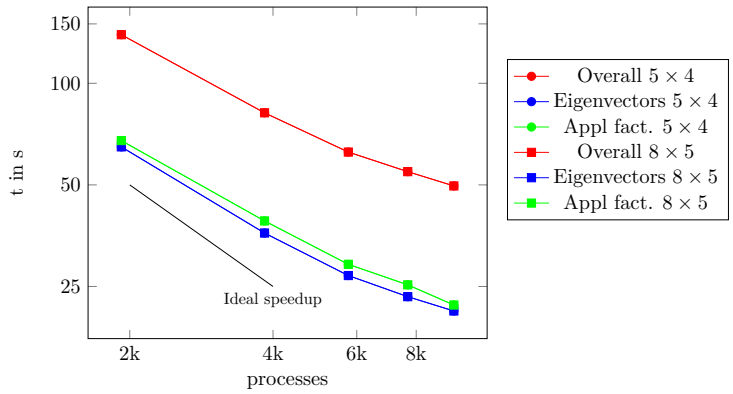

Figure 13. Strong scaling for a matrix of size 204800. The bandwidth (and hence the blocksize) is 2048, the twist index is at 102400. The backtransformation is done for 25600 eigenvectors (12.5%). Per group 5×4 and 8×5 processes have been used. The runs have been carried out on the Cobra Supercomputer.

processes per group. Using more processes per group, however, comes along with a loss in performance compared to the same number of processes with less processes per group. Not shown here is the additional bandreduction step which is necessary for bandwidth of size 512 or 2048. The savings compared to computing the dense eigenvalue problem however, will still be significant.

For solving generalized eigenvalue problems with banded matrices this procedure allows to compute eigenpairs at matrix sizes where the standard procedure with factorizing B, applying B to A and using a standard dense solver for the full resulting matrix C would consume to much memory or result in way more computation. When considering sparse eigenvalue solvers, the computation of

higher percentages of the eigenpairs becomes expensive. This approach, however, provides the possibility to overcome this issue.

5. Applications

First-principles simulations in computational chemistry, solid state physics, and materials science typically involve to determine the interactions between the M nuclei described by $3M$ nuclear positions $\{\vec{R}\}$. Being able to compute the total energy of the system $E_0(\{\vec{R}\})$, i.e., the high dimensional potential energy surface (PES), as a function of $\{\vec{R}\}$ and, ideally, its derivatives such as the forces acting on the nuclei $\vec{F}_I(\{\vec{R}\}) = -\nabla_{\vec{R}_I} E_0(\{\vec{R}\})$, allows to investigate the properties of molecules and materials. For instance, one can systematically map out the PES $E_0(\{\vec{R}\})$ to search for (stable) minima and saddle points between them or explore it dynamically via molecular dynamics (MD) or statistical (e.g. Monte Carlo) sampling. Accordingly, a typical computational study often requires to determine $E_0(\{\vec{R}\})$ for thousands of nuclear configurations $\{\vec{R}\}$.

Computing $E_0(\{\vec{R}\})$ requires to solve the quantum-mechanical electronic-structure problem. In density-functional theory (DFT) [16], the most wide-spread electronic-structure formalism, this requires to identify the electronic density $n(\vec{r})$ that minimizes the convex total-energy functional $E_0 = \min E[n(\vec{r})]$ for a given number of electrons $N = \int d\vec{r} n(\vec{r})$. In Kohn-Sham (KS) DFT [17], this variational problem is mapped onto a series of eigenvalue problems (EVP), the so called self-consistent field (SCF) formalism. In each step of the SCF cycle, the EVP

$$H\left[n(\vec{r})\right]\Psi(\vec{r}) = \varepsilon\Psi(\vec{r}) \quad \text{with} \quad n(\vec{r}) = \sum_{s=1}^{N} |\Psi_s(\vec{r})|^2 \tag{1}$$

is solved to determine the eigenstates Ψ_s. The N eigenstates Ψ_s with the lowest eigenvalues ε_s allow to compute an updated and improved $n(\vec{r})$, for which Equation (1) is then solved again. This procedure is repeated until "self-consistency" is achieved at the end of the so called SCF cycle, i.e., until a stationary solution with minimal $E[n(\vec{r})]$ is found. In practice, a basis set expansion $\Psi_s = \sum_i x_{si}\varphi_i(\vec{r})$, e.g., in terms of Gaussians, plane waves, numerical functions, etc., is used to algebraize and solve Equation (1). By this means, one obtains the generalized EVP

$$A\left[n(\vec{r})\right]x = \lambda B x \,, \tag{2}$$

the size of which is determined by the number of basis functions $\varphi_i(\vec{r})$ employed in the expansion. Here, the Hamiltonian A and the overlap matrix B are given as:

$$A_{ij}[n(\vec{r})] = \int d\vec{r}\varphi_i^*(\vec{r})\, H\left[n(\vec{r})\right]\, \varphi_j(\vec{r}) \,, \quad B_{ij} = \int d\vec{r}\varphi_i^*(\vec{r})\varphi_j(\vec{r}) \,.$$

5.1. Autotuning: The Case of GPU Offloading

Due to the cubic scaling with system size, the generalized EVP (2) quickly becomes the numerical bottleneck in practical DFT calculations. It is thus more than desirable to use optimal ELPA settings (ELPA1 vs. ELPA2, architecture-specific kernels, etc.) to utilize the computational resources in the most efficient way so to obtain the optimal time-to-solution. As discussed above, this is of particular importance in first-principles simulations, which require solving many similar eigenvalue problems, e.g., the 10–100 individual SCF steps in one SCF cycle or the thousands if not millions of SCF steps performed in an iterative exploration of the PES $E_0(\{\vec{R}\})$. ELPA's autotuning feature allows to determine these optimal settings, which depend upon both the inspected physical problem and the used architecture, in an automated way [3].

This is particularly important for new and upcoming architectures featuring GPUs: This is exemplified in Figure 14, which shows calculations performed with the FHI-aims code [18] using ELSI [19] as interface to ELPA and the PBE exchange-correlation functional [20] for periodic Caesium Chloride crystals as function of the number of basis functions used. For this purpose, calculations with different system sizes, i.e., number of atoms, were performed. Since FHI-aims uses local atomic orbitals [18], the number of basis functions increases with the number of atoms: For example, the smallest investigated system contains 16 atoms and thus uses 496 basis functions, while the largest system contains 3,456 atoms and 107,136 basis functions. For all system sizes, we benchmarked ELPA1 and ELPA2 separately; in both cases, CPU only calculations as well as calculations using CPUs and full GPU acceleration (for the tridiagonalization, the solution of the eigenvalue problem, and the back transformation) were performed on four Intel Skylake (Xeon Gold 6138) + nVidia Tesla V100 nodes with two CPUs and GPUs each (20 cores/CPU @ 2.0 GHz).

As Figure 14 shows, the use of GPU acceleration offers a sizeable performance increase for large systems with respect to CPU-only calculations for both ELPA1 and ELPA2, whereby the gains are more pronounced for ELPA1. The threshold number of basis functions for which GPUs indeed accelerate the calculation is essentially determined by the workload on each CPU and GPU. For too small systems, the time spent transferring the data to the GPU is larger than the actual computational gains due to the GPU. In this particular case, GPUs are thus beneficial for ELPA1 for more than 10,000 basis functions and for ELPA2 for more than 20,000 basis functions. Overall, CPU-only ELPA1 is the fastest solver up to 4,000 basis functions, CPU-only ELPA2 for system between 4,000 up to roughly 20,000 basis functions, and CPU+GPU ELPA1 for all systems with even larger number of basis functions. Note that this might be quite surprising even for well-experienced ELPA users, given that ELPA2 is typically superior to ELPA1 for large system sizes in the CPU only case, as also shown in Figure 14. In practice, switching from CPU-only ELPA2 to CPU+GPU ELPA1 can thus lead to significant savings in computational time around 30%, as it is the case for a system size with 107,136 basis functions.

As shown above, optimal performance can only be achieved if different combinations of ELPA1 and ELPA2 with and without GPU acceleration are chosen

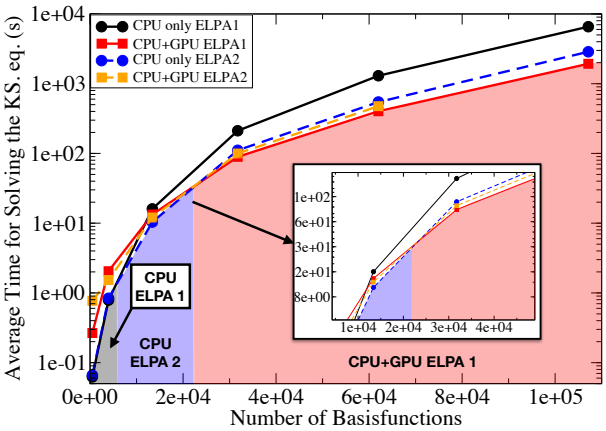

Figure 14. Computational time per SCF step (in seconds) as function of the numbers of basis functions employed. Solid lines denote ELPA1, dashed lines ELPA2 calculations. CPU-only and CPUs+GPU calculations were performed. The shaded areas denote which setup is fastest for different system sizes. The inlet shows the timings for system sizes at which CPU+GPU ELPA1 becomes the fastest solver (border between blue and red marked areas).

depending on the system size. Moreover, the actual threshold at which GPU acceleration becomes beneficial strongly depends on the number of nodes employed in the calculation: For smaller number of nodes, the workload on the individual nodes increases and GPU acceleration becomes beneficial earlier, i.e., for smaller system sizes. Eventually, let us note that for the calculations shown in Figure 14 the GPU acceleration was used for the tridiagonalization, the solution of the eigenvalue problem, and the back transformation. In practice, it can be beneficial to exploit GPUs only for a subset of these steps, as shown below. This particular application thus showcases the importance of ELPA's autotuning functionality, which saves the user from performing tedious benchmark calculations for all different settings and prevents him from choosing sub-optimal settings, e.g., by choosing ELPA2 for large systems based on previous CPU-only experience.

We have explicitly verified this by running calculations with autotuning enabled for two different system sizes with 13,392 and 31,744 basis functions, respectively. As shown in Table 3, the autotuning procedure is able to identify an optimal solution for both cases. In the smaller system with 13,392 basis functions, CPU-only ELPA2 is the optimal solution. Note that in this case the CPU kernel has been fixed to the AVX512-one in all calculations, otherwise also this parameter would have been optimized by the autotuning procedure [3]. For the larger system with 31,744 basis functions, ELPA1 with GPU acceleration is identified as the optimal setup. Compared to the earlier calculations shown in Figure 14, the autotuning procedure found out that it is beneficial to use GPU acceleration only for the tridiagonalization and the back transformation, whereas the solution of the eigenvalue problem is better performed only on the CPUs. The additional gain in computational saving of roughly 1% compared to the next-best solution is not earth-shattering in this case, but still noticeable, given that in actual simulations this 1% can be exploited for thousands if not millions of eigenvalue problems. As already discussed in [3], the cost of the autotuning procedure is well

Table 3. Computational time required for solving the KS equations in seconds for ELPA1 and ELPA2 (CPU-only and CPU+GPU calculations) as well as for the optimal settings found by ELPA's autotuning functionality.

Number of basis functions	ELPA1 CPU-only	ELPA1 CPU+GPU	ELPA2 CPU-only	ELPA2 CPU+GPU	Optimal
13,392	16.03s	13.29s	10.38s	12.05s	10.38s
31,744	211.40s	89.38s	110.90s	99.72s	88.73s

worth the gains in practical calculations. Although some sub-optimal setups such as CPU+GPU ELPA1 and CPU-only ELPA1 are tested during the autotuning for the small and large system, respectively, the benefits outrun these costs in the long term.

5.2. Performance Benefits by Reduced Precision

In DFT simulations, the individual SCF iterations leading up to self-consistency are of no particular interest. Only the final results of the converged SCF cycle have any physical relevance at all. Hence, it is worthwhile to study how a reduction in precision of the SCF procedure from double (DP) to single precision (SP) might accelerate the generalized EVP as the numerical bottleneck of DFT simulations, as long as the final converged result is not altered up to the precision required by the problem at hand. In this section, the precision is independently controlled for the following individual eigensolver steps: the Cholesky decomposition (i), the matrix multiplication in (ii) and (iv), and the solution of the eigenproblem via tridiagonalization (iii) (see Section 3). Since SP in the matrix inversion step $U \rightarrow U^{-1}$ destroys the convergence entirely [3], the inversion of U is always conducted in DP.

To demonstrate the gain in computational performance by both the algorithmic improvements and the readily available SP routines in the new version of ELPA, we have performed DFT calculations with FHI-aims [18] using ELSI [19] as interface to different ELPA versions. The model system chosen for performance comparisons is selected from a class of novel, self-organizing materials, called metal-organic frameworks (MOF). Their electric conductivity can be manipulated and tuned by doping with different metal ions [21]. Due to the low concentration of doping atoms, the theoretical description is challenging and requires the simulation of extensive supercells with a large number of atoms and hence basis functions. Therefore, the iron triazolate MOF doped with a single copper atom [22] is an ideal benchmark system to quantify the speed-up achieved by different precisions by evaluating five SCF cycles and the atomic forces for supercells ranging from 2,405 to 19,343 atoms and 30,167 to 244,529 basis functions, respectively. The calculations were conducted on Intel Xeon 'Skylake' (40 cores @ 2.4 GHz) and compared to the FHI-aims internal ELPA 2013 (only DP available).

As shown in Figure 15, replacing the FHI-aims internal ELPA 2013 by ELPA2018.11 (DP) provides a speed-up of about 1.6 for the solution of the Kohn-Sham eigenvalue problem. For high-level parallelization, where ELPA 2013 does not scale very well, speed-up factors over 2.0 can be achieved. The total computational time is reduced by an average speed-up factor of 1.3, which can go up to 1.7 for large runs. This speed-up comprises all improvements and developments in the

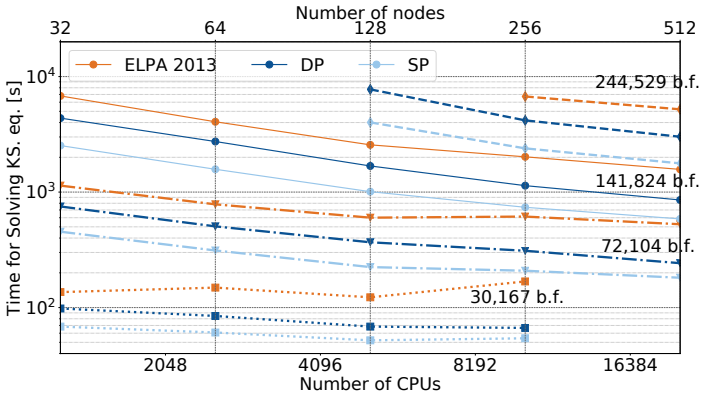

Figure 15. Computational time for solving the Kohn-Sham equation in five SCF iterations (in seconds) as function of the number of CPUs. Calculations conducted with the FHI-aims internal ELPA 2013 are denoted in orange. Application of ELPA2018.11 with DP and SP in steps (i) to (iv) are depicted in dark and light blue, respectively. The different line styles show the timings for different system sizes from 30,167 (dotted) via 72,104 (dash-dotted) and 141,824 (solid) to 244,529 (dashed) basis functions (b.f.).

ELPA library since the FHI-aims internal ELPA 2013 version, such as AVX-512 kernel optimization, autotuning etc. (see Section 2) but without the application of GPUs.

Table 4 summarizes the speed-up factors broken down into the individual steps of the GSEP. The Cholesky decomposition (i) is only conducted in the first SCF iteration of each SCF cycle, i.e., only once in each benchmark calculation. The gain by reduction to SP in the Cholesky step (i) is minimal with a speed-up factor of about 1.1. Whereas for strong-scaling situations with high parallelization ($<$ 20 basis functions / cpu), SP in the Cholesky decomposition can effectively increase the computational time of this step. In contrast, SP in the matrix multiplication of step (ii) and (iv) efficiently reduces the cpu time to 50% of the DP computational time (speed-up factor 2.0). Similarly, SP in the eigensolver (iii) achieves a speed-up of factor 1.9 for the computational time of step (iii). The combination of SP in steps (i), (ii), (iii), and (iv) provides a speed-up of about 1.7 for the solution of the Kohn-Sham eigenvalue problem and of about 1.3 for the total computational time, unless parallelization is high ($<$ 20 basis functions / cpu).

6. Conclusions

We have presented the recent advances in the ELPA eigenvalue solver project. Due to the API changes the autotuning functionality is now available for users. It allows also non-experts to find the best parameter setups for their runs. Especially in the setting of electronic structure theory where many similar eigenvalue problems have to be solved, autotuning is a very powerful instrument. Additional gain in computational time was demonstrated by a mixed-precision approach where certain steps to solve a generalized eigenvalue problem are done in single instead

Table 4. Speed-up factors for SP versus DP (ELPA2018.11) for five SCF iterations and increasing number of basis functions (b.f.) decomposed into each step of the GSEP solver: Cholesky decomposition (i), transformation of GSEP to SEP (ii), solution of the eigenproblem via tridiagonalization (iii), and back-transformation of the eigenvectors (iv). The last two columns summarize the speed-up factors for the computational time required for the solution of the Kohn-Sham equation and for the total computational time.

No. b.f.	(i)	(ii)	(iii)	(iv)	KS	total
30,167	0.9	1.8	1.4	1.7	1.3	1.1
72,104	1.0	1.9	1.7	1.9	1.5	1.2
141,824	1.1	2.0	1.8	2.0	1.6	1.3
244,529	1.2	2.3	2.1	2.2	1.8	1.4

of double precision. The computational kernels and routines have been further optimized and been ported for the newest GPU and CPU Hardware. This allows to accelerate the computation of eigenvalues and eigenvectors and compute even larger matrices. The new algorithmic developments improve the solution of the generalized eigenvalue problems. For the banded and the dense matrix case remarkable savings in computation time have been shown.

References

[1] T. Auckenthaler, V. Blum, H.-J. Bungartz, T. Huckle, R. Johanni, L. Kraemer, B. Lang, H. Lederer, and P. Willems, "Parallel solution of partial symmetric eigenvalue problems from electronic structure calculations," *Parallel Comput.*, vol. 27, no. 12, pp. 783–794, 2011.

[2] A. Marek, T. Auckenthaler, V. Blum, H.-J. Bungartz, H. Ville, R. Johanni, A. Heinecke, B. Lang, and H. Lederer, "Parallel eigenvalue solutions for electronic structure theory and computational science," *J. Phys. Condens. Matter*, vol. 26, no. 21, p. 213201, 2014.

[3] P. Kůs, A. Marek, S. Koecher, H.-H. Kowalski, C. Carbogno, C. Scheurer, K. Reuter, M. Scheffler, and H. Lederer, "Optimizations of the eigensolvers in the ELPA library," *Parallel Comput.*, vol. 85, pp. 167 – 177, 2019.

[4] B. Cook, T. Kurth, J. Deslippe, P. Carrier, N. Hill, and N. Wichmann, "Eigensolver performance comparison on cray xc systems," *Concurr. Comp.-Pract. E.*, vol. 31, no. 16, p. e4997, 2019.

[5] P. Kůs, H. Lederer, and A. Marek, "GPU optimization of large-scale eigenvalue solver," in *Numerical Mathematics and Advanced Applications ENUMATH 2017* (F. A. Radu, K. Kumar, I. Berre, J. M. Nordbotten, and I. S. Pop, eds.), (Cham), pp. 123–131, Springer International Publishing, 2019.

[6] L. S. Blackford, J. Choi, A. Cleary, E. D'Azevedo, J. Demmel, I. Dhillon, J. Dongarra, S. Hammarling, G. Henry, A. Petitet, K. Stanley, D. Walker, and R. C. Whaley, *ScaLAPACK Users' Guide*. Philadelphia, PA: SIAM, 1997.

[7] J. Poulson, B. Marker, R. A. van de Geijn, J. R. Hammond, and N. A. Romero, "Elemental: A new framework for distributed memory dense matrix computations," *ACM Trans. Math. Software*, vol. 39, pp. 13:1–13:24, Feb. 2013.

[8] J. J. Dongarra, L. Kaufman, and S. Hammarling, "Squeezing the most out of eigenvalue solvers on high-performance computers," *Linear Algebra Appl.*, vol. 77, pp. 113–136, 1986.

[9] J. Choi, J. J. Dongarra, and D. W. Walker, "Parallel matrix transpose algorithms on distributed memory concurrent computers," *Parallel Comput.*, vol. 21, pp. 1387–1405, Sept. 1995.

[10] L. E. Cannon, *A Cellular Computer to Implement the Kalman Filter Algorithm*. PhD thesis, Montana State University, Bozeman, MT, 1969.

[11] H.-J. Lee, J. P. Robertson, and J. A. B. Fortes, "Generalized Cannon's algorithm for parallel matrix multiplication," in *Proc. ICS '97, Intl. Conf. Supercomputing, July 7–11, 1997, Vienna, Austria*, pp. 44–51, ACM Press, 1997.

[12] V. Manin and B. Lang, "Cannon-type triangular matrix multiplication for the reduction of generalized HPD eigenproblems to standard form," 2018. Submitted.

[13] C. R. Crawford, "Reduction of a band-symmetric generalized eigenvalue problem," *Comm. ACM*, vol. 16, pp. 41–44, jan 1973.

[14] B. Lang, "Efficient reduction of banded hermitian positive definite generalized eigenvalue problems to banded standard eigenvalue problems," *SIAM J. Sci. Comput.*, vol. 41, no. 1, pp. C52–C72, 2019.

[15] M. Rippl, B. Lang, and T. Huckle, "Parallel eigenvalue computation for banded generalized eigenvalue problems," *Parallel Comput.*, vol. 88, p. 102542, 2019.

[16] P. Hohenberg and W. Kohn, "Inhomogeneous electron gas," *Phys. Rev.*, vol. 136, no. 3B, pp. B864–B871, 1964.

[17] W. Kohn and L. J. Sham, "Self-consistent equations including exchange and correlation effects," *Phys. Rev.*, vol. 140, no. 4A, pp. A1133–A1138, 1965.

[18] V. Blum, R. Gehrke, F. Hanke, P. Havu, V. Havu, X. Ren, K. Reuter, and M. Scheffler, "Ab initio molecular simulations with numeric atom-centered orbitals," *Comput. Phys. Commun.*, vol. 180, no. 11, pp. 2175 – 2196, 2009.

[19] V. W. Yu, F. Corsetti, A. Garcia, W. P. Huhn, M. Jacquelin, W. Jia, B. Lange, L. Lin, J. Lu, W. Mi, A. Seifitokaldani, Álvaro Vázquez-Mayagoitia, C. Yang, H. Yang, and V. Blum, "ELSI: A unified software interface for Kohn–Sham electronic structure solvers," *Comput. Phys. Commun.*, vol. 222, pp. 267 – 285, 2018.

[20] J. P. Perdew, K. Burke, and M. Ernzerhof, "Generalized gradient approximation made simple," *Phys. Rev. Lett.*, vol. 77, pp. 3865–3868, Oct 1996.

[21] F. Schröder and R. A. Fischer, "Doping of metal-organic frameworks with functional guest molecules and nanoparticles," *Top. Curr. Chem.*, vol. 293, pp. 77–113, 2010.

[22] L. S. Xie, L. Sun, R. Wan, S. S. Park, J. A. DeGayner, C. H. Hendon, and M. Dincă, "Tunable Mixed-Valence doping toward record electrical conductivity in a Three-Dimensional Metal–Organic framework," *J. Am. Chem. Soc.*, vol. 140, no. 24, pp. 7411–7414, 2018.