On Enhancing 3D-FFT Performance in VASP

Florian Wende⋆, Martijn Marsman⋆⋆, Thomas Steinke⋆
⋆Zuse Institute Berlin, Takustrasse 9, 14195 Berlin, Germany: {wende, steinke}@zib.de
⋆⋆Universität Wien, Sensengasse 8, 1090 Wien, Austria: martijn.marsman@univie.ac.at

***Abstract*—We optimize the computation of 3D-FFT in VASP in order to prepare the code for an efficient execution on multi- and many-core CPUs like Intel's Xeon Phi. Along with the transition from MPI to MPI+OpenMP, library calls need to adapt to threaded versions. One of the most time consuming components in VASP is 3D-FFT. Beside assessing the performance of multi-threaded calls to FFTW and Intel MKL, we investigate strategies to improve the performance of FFT in a general sense. We incorporate our insights and strategies for FFT computation into a library which encapsulates FFTW and Intel MKL specifics and implements the following features: reuse of FFT plans, composed FFTs, and the use of high bandwidth memory on Intel's KNL Xeon Phi. We will present results on a Cray-XC40 and a Cray-XC30 Xeon Phi system using synthetic benchmarks and with the library integrated into VASP.**

I. Introduction

The Fast Fourier Transform (FFT) is relevant to many scientific codes. Besides those shipping with their own, maybe problem specific, FFT implementation, many codes draw on library based solutions like the widely used FFTW [1]—included within Cray's LibSci—or Intel's MKL. As FFT code optimization is close to the hardware, the integration of FFT into codes using external libraries can be expected the "more portable" way in the sense of maintaining the code base over longer periods of architectural changes of the computer hardware. However, with the transition from multi-core to many-core CPUs as well as the integration of additional layers in the memory hierarchy, like the high bandwidth memory on Intel's upcoming Knights Landing (KNL) platform, the way how the FFT library itself is called from within the program needs to be reconsidered.

For (legacy) MPI-only codes, adaptions towards MPI+X, e.g., MPI+OpenMP, require to either call into the FFT library from within a multi-threaded program context or to use a multi-threaded version of the library. In both cases the scaling of the FFT computation with the number of threads will affect the partitioning of the execution streams into MPI ranks and threads per rank. On Intel's KNL with its 60+ cores and up to 4 hardware threads per core, the minimization of MPI ranks together with thread count maximization should be targeted to leverage its compute resources most effectively. In the VASP application (as a representative of a widely used HPC code), it is challenging to scale FFT computations beyond significantly more than a handful of threads per MPI rank, resulting in more than 1 rank per KNL node—with its rather low per-core performance and its low core frequency, MPI on the KNL can be expected to slow down with an increasing number of MPI ranks. As 3D FFT computation consumes massive amounts of time for many VASP inputs, optimizing its usage within VASP is necessary in order to make the code ready not just for KNL, but for future computer platforms as well.

This paper contains the following contributions in the context of FFT computation on modern computer platforms:

- We evaluate the threading performance of widely used FFTW (Cray LibSci) and Intel's MKL on a current Cray-XC40 with Intel Haswell CPUs and a Cray-XC30 Xeon Phi (Knights Corner, KNC) system.
- We investigate strategies to improve the FFT performance including plan reuse, composed FFT computation, skipping of data transpose operations between successive FFTs, and the use of high bandwidth memory (as a preparation for Intel's KNL).
- We introduce the C++ template library FFTLIB which encapsulates our findings and provides them to the user in a transparent way by intercepting calls to FFTW.

Related Work: Being used in a large variety of computational workflows and data processing, FFT has been subject to optimizations for a long time already. It is well-known that for Density Functional Theory (DFT) computations with plane-wave basis sets, the 3D-FFT is one of the time-critical computational steps [8]. For computations on moderately sized grids, as is the case for many VASP inputs, for instance, domain-specific optimized FFTs exist. Our work focuses on optimizing the 3D-FFT for plane-wave DFT methods. With peta-scale compute systems available, most of the related work is focused on scalable, distributed 3D-FFT implementations [4], [6]. In this paper, we focus on on-node 3D-FFT computation. Our findings regarding composed FFT computation, however, can be used in the context of distributed 3D-FFT computation as well.

II. FFT Computation with FFTW (LibSci) and MKL

A. The Fast Fourier Transform

The FFT is an efficient way to calculate the Fourier Transform [9]

$$\hat{f}(\mathbf{k}) : \hat{f}(k_n) = \sum_{j=0}^{N-1} f(x_j)\, \mathrm{e}^{\frac{2\pi i n j}{N}} \,,\; k_n = \frac{2n\pi}{L} \,,\; n = 0, \ldots, N-1 \tag{1}$$

of a periodic function $f(\mathbf{x})$ with period L, defined on N discrete points $\mathbf{x} = \{x_j = j\frac{L}{N} \,|\, j = 0, \ldots, N-1\}$, with computational complexity $\mathrm{O}(N \log_2(N))$ opposed to $\mathrm{O}(N^2)$ in case of doing it straightforwardly. FFT uses the fact that e^{ikx} has the same

value for different combinations of x and k, thereby reducing the number of computational steps.

Note that Eq. 1 describes the one-dimensional FFT of $f(\mathbf{x})$ (or 1D-FFT for short). However, n-dimensional FFTs can be thought of as being composed of n successive 1D-FFTs.

B. FFTW and MKL

Both FFTW [1] and Intel MKL provide to the user a common C and Fortran interface for 1D-, 2D- and 3D-FFT computation on complex and real data. On Cray machines, FFTW (possibly with some Cray specific optimizations) is accessible through LibSci. Additionally, MKL provides an extended feature set that is available through its DFTI interface, which is a superset of what is accessible through its FFTW compatibility layer. All of these libraries will be available on Intel's KNL platform due to its x86 compatibility which allows for non-Intel software stacks. Users hence will have the choice to either use FFTW (through LibSci) or MKL on KNL.

In this paper, we use the following FFT library versions: a self-compiled version of FFTW,[1] a Cray-optimized version of FFTW (rev. 3.3.4) through LibSci 13.2.0, and FFT through Intel's FFTW wrappers coming with MKL 11.3.2. General code compilation happened with the Intel 16.0.2 (20160204) C/C++/Fortran compiler. We use Cray's `iobuf` library (rev. 2.0.6) and `craype-hugepages8M`. Our hardware configuration comprises: Cray-XC40 compute nodes with dual-socket Intel Xeon E5-2680v3 Haswell CPUs (24 cores per node, and cores clocked at 1.9 GHz—AVX base frequency), and a Cray-XC30 test system with Intel Xeon Phi 5120D coprocessors (59 cores per device, and 4-way hardware multi-threading per core).

Using the FFTW Interface: The usual pattern when calling FFTW (or MKL through its FFTW interface) is as follows:

0. (optional) Initialize threading via
   `fftw_init_threads()` and
   `fftw_plan_with_nthreads(nthreads)`.
1. Create plans for FFT computations, e.g., via
   `fftw_plan p=fftw_plan_dft_3d(..)`.
2. Perform FFT computation using that plan (as often as needed) via
   `fffw_execute(p)` or
   `fftw_execute_dft(p,in,out)`.
3. Clean up plans via
   `fftw_plan_destroy(p)` and
   `fftw_cleanup()` or
   `fftw_cleanup_threads()`.

Before the actual FFT computation, a plan `p` needs to be created. Any plan `p` can be used as often as needed until either `fftw_plan_destroy(p)`, `fftw_cleanup()` or `fftw_cleanup_threads()` is called. It is recommended to use plans as often as possible to amortize for their creation costs.

Plan creation with FFTW can happen with differently expensive planner schemes:

`FFTW_ESTIMATE` (cheap),
`FFTW_MEASURE` (expensive),
`FFTW_PATIENT` (more expensive) and
`FFTW_EXHAUSTIVE` (most expensive).

Except for `FFTW_ESTIMATE` plan creation involves testing different FFT algorithms together with runtime measurements to achieve best performance on the target platform. With MKL, these schemes can be specified for compatibility reasons, but none of them affect the internal planner.

To speed up the plan creation step, FFTW implements the so-called "wisdom" feature which is used when planning with `FFTW_MEASURE` or any of the more expensive schemes. For each plan requested from FFTW, an internal data structure is created that holds a "minimal" set of information for fast plan recreation in case of compatible successive requests. When doing lots of FFTs for certain grid geometries, it is recommended for each of them to first plan with any of the expensive schemes and then to use the same scheme again or `FFTW_ESTIMATE`. Wisdom then will be used to get the "good" plans much faster:

```
// first planner call: expensive
p=fftw_plan_dft_3d(64,72,68,in,out,
                   FFTW_FORWARD,FFTW_MEASURE)
// successive planner call(s): wisdom is used
p=fftw_plan_dft_3d(64,72,68,in,out,
                   FFTW_FORWARD,FFTW_MEASURE).
```

C. 3D-FFT in VASP (Initial Setup)

In the VASP application the following pattern is used for all kinds of FFT computation: "create plan – execute$^{(n)}$ – destroy plan," where $n = 1$ in most cases. Table. I summarizes for one particular input (PdO2: Palladiumdioxide on a Paladium serface) the total execution time of VASP using 24 MPI ranks on 4 Cray-XC40 compute nodes. Each MPI rank uses either 1 or 4 OpenMP threads, with all MPI ranks and threads evenly distributed across the two CPU sockets per node.

For runs with MKL, the 3D-FFT performance increases by about factor 2.2 when using 4 instead of 1 OpenMP thread. With FFTW, we observe the opposite, that is, the time spent in 3D-FFT computation increases with the number of threads. The effective performance loss can be traced to the planner of the FFTW library—VASP strictly follows the proposed proceeding: plan with `FFTW_MEASURE` in the first instance, and then use `FFTW_ESTIMATE` [1]. The 3D-FFT execution time, on the other hand, decreases with the number of threads.

[1] The source code has been taken from the official FFTW GitHub repository `https://github.com/FFTW/FFTW`, and has been compiled using Intel's C/C++/Fortran compiler 16.0.2 (20160204) with configure options `--enable-avx2 --enable-fma --enable-openmp` for Haswell CPU, and `--enable-kcvi --enable -fma --enable-openmp` for Intel Xeon Phi (KNC). Compile options: `-O3` and `-xcore-avx2` for Haswell and `-mmic` for Xeon Phi (KNC). For Xeon Phi, we additionally built the official FFTW library (rev. 3.3.4) due to significantly reduced planner performance of FFTW from GitHub—not so on Haswell. However, the official FFTW library (rev. 3.3.4) does not incorporate Xeon Phi optimizations and hence gives poor performance.

TABLE I
VASP PROGRAM EXECUTION TIME FOR THE PDO2 SETUP ON 4 CRAY-XC40 COMPUTE NODES WITH INTEL XEON E5-2680V3 (HASWELL) CPUS. IN ALL CASES 24 MPI RANKS ARE USED, EACH WITH T=1 OR T=4 OPENMP THREADS. ADDITIONALLY, THE TIME SPENT IN 3D-FFT PLANNING AND EXECUTION IS GIVEN.

	Setup: PdO2					
	MKL 11.3.2		FFTW		FFTW (LibSci)	
	T=1	T=4	T=1	T=4	T=1	T=4
Total	146.6s	78.0s	162.1s	122.5s	162.3s	121.9s
3D-FFT	23.4s	10.7s	38.2s	43.3s	38.6s	41.9s
+planner	1.0s	1.0s	10.0s	32.9s	9.8s	31.1s
+execute	22.4s	9.7s	28.2s	10.4s	28.8s	10.8s

FFTW Planner: To further investigate our observations regarding the FFTW planner, we determine the time spent in the first and successive planner call(s) using `FFTW_MEASURE`. The costs for the FFTW planner step on an Intel Haswell 12-core CPU using 1 MPI rank and up to 12 OpenMP threads are given in Fig. 1 for complex 3D-FFTs on three differently sized grids—even though MKL does not support the different FFTW planner schemes, we included the respective data for comparison reasons. Costs for the FFTW planner step on KNC are given in Fig. 2 for up to 56 OpenMP threads spread across the physical CPU cores in "scatter" fashion.

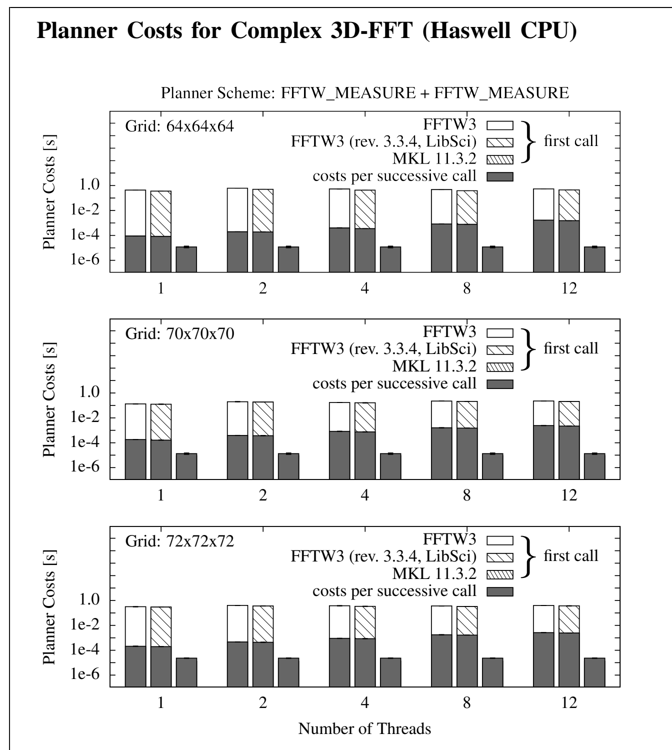


Fig. 1. Planner costs for 3D-FFT computation for different grid sizes and different numbers of threads on an Intel Xeon E5-2680v3 Haswell CPU. The patterned bars display the costs for the first planner call. All successive calls, represented by the light gray bars, make use of wisdom in case of FFTW (no equivalent for MKL) and are much faster. For all plan creations FFTW_MEASURE is used.

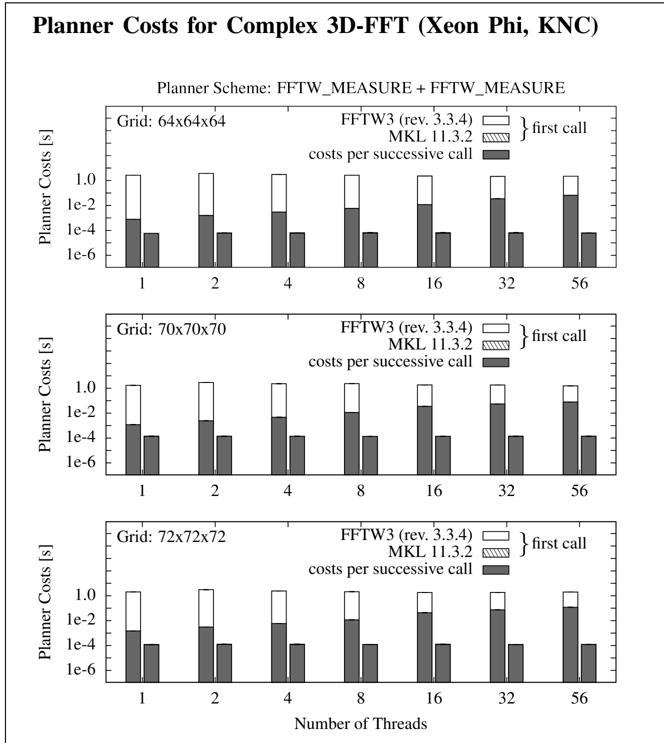


Fig. 2. Same as for Fig. 1, but for an Intel Xeon Phi 5120D instead.

For FFTW, the initial planner costs are the most expensive ones (patterned boxes), whereas all successive planner calls (gray boxes) requesting a plan for exactly the same grid are significantly faster by at least two orders of magnitude. Thus, we conclude that wisdom is functioning as expected. However, comparing the cost for any of the successive planner calls against those for MKL's planner, there is still up to two orders of magnitude discrepancy to the disfavor of FFTW. Additionally, the costs seem to increase with the number of threads to be use for the FFT computation.

Unlike FFTW, the planner costs for MKL remain constant almost independently of the problem size. As the MKL library is closed source, it is not clear to us what actually happens within MKL at this point: it might be possible that there is no equivalent to FFTW's planner step, and MKL just suffices the FFTW interface—explaining the constant costs. If the latter proves right, MKL, however, seems not to be at a disadvantage compared to FFTW. On the contrary, looking at the FFT execution times listed in Tab. I, MKL uses an FFT algorithm that performs better than FFTW.

The question that might arise in that context is "does it pay off to spend that much time in the planner phase when using FFTW?" Instead of using `FFTW_MEASURE`, we could switch to `FFTW_ESTIMATE` so as to have costs close to those for the successive calls in Fig. 1 and 2 right from the beginning.

Effectiveness of Expensive Planner Schemes (FFTW only): To assess the effect of planning with `FFTW_MEASURE` instead of `FFTW_ESTIMATE`, we simply run a synthetic kernel that creates plans with the different planner schemes and then

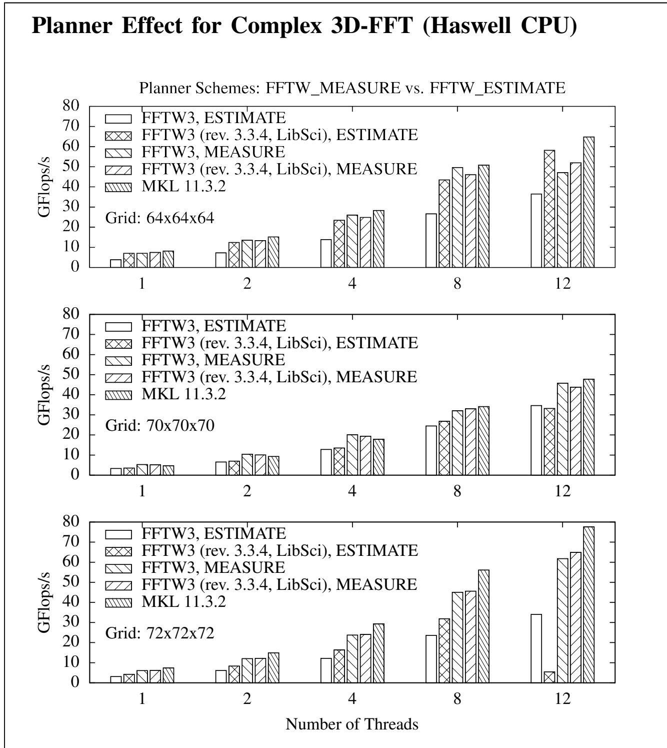


Fig. 3. Performance of a complex 3D-FFT computation for different grid sizes and planner schemes, and different numbers of threads on an Intel Xeon E5-2680v3 Haswell CPU.

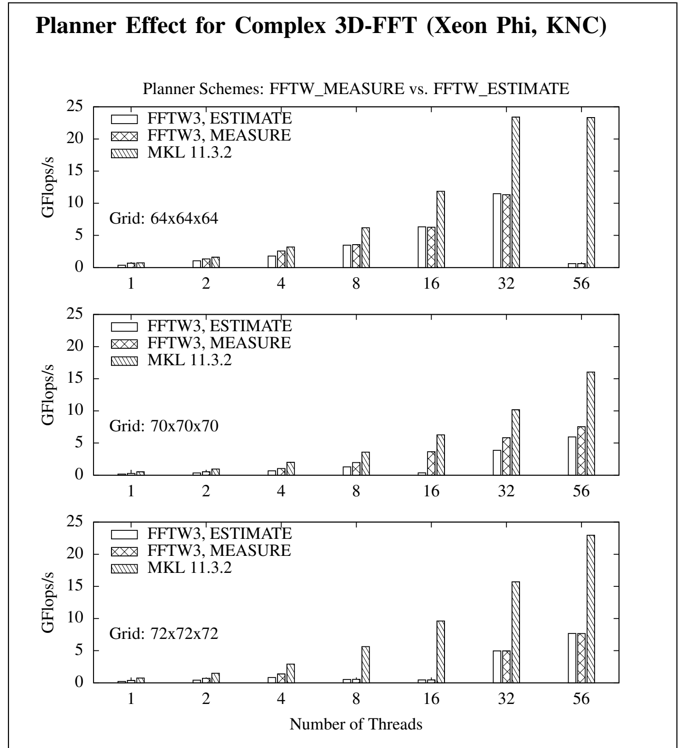


Fig. 4. Same as for Fig. 3, but for an Intel Xeon Phi 5120D instead.

measures the time for the complex 3D-FFT computation. Figures 3 and 4 give for different grid sizes and thread counts the approximated Flops/s according to what is proposed on the FFTW webpage for FFTs on complex data [1]:

$$\mathrm{Flops/s} = 5\,n \log_2(N)/t\,, \tag{2}$$

where N is the number of grid points and t is the execution time in seconds.

While on the Xeon Phi planning with `FFTW_MEASURE` has only minor effect on the FFT performance (at least for the grid sizes considered), the execution on the Haswell CPU can benefit significantly from the more expensive planner scheme. Up to a factor 2 performance gain over planning with `FFTW_ESTIMATE` can be noted for both single- and multi-threaded execution.

Figures 3 and 4 additionally show the scaling of the complex 3D-FFT computation with the number of threads. For both FFTW and MKL, the scaling is quite acceptable for up to 8 threads on Haswell, and up to 32 threads on the Xeon Phi. However, on the Xeon Phi the FFTW performance is significantly behind MKL, and in some cases shows elusive drops, e.g., 8 and 16 thread processing of the 72^3 FFT, or 56 thread processing of the 64^3 FFT. We assume that FFTW optimizations for the Xeon Phi do not come into effect for the small problem sizes considered in this paper. This might be different for larger grids.

Remark: Do not wonder about the low performance on the Xeon Phi. Running with 12 threads on Haswell, one entire CPU socket is used. With 56 threads on the Xeon Phi, however, it is just 1/4th of the device. Due to the 2-cycle instruction fetch issue on the Xeon Phi, it can be expected to get at least a factor 2 gain over what is given in Fig. 4 when using 2 or more threads per CPU core. Further performance gains can be achieved when using multiple MPI ranks per device—each of which doing its own FFT computation: throughput oriented approach—, and by placing OpenMP threads not in scatter, but in "compact" fashion. The latter results in better per-core cache utilization. However, the same also applies to FFT computation on the Haswell compute node.

TABLE II
OVERALL PERFORMANCE (IN FLOPS/S) FOR M COMPLEX 64^3 FFTS EXECUTED BY M MPI RANKS, EACH OF WHICH USING n THREADS.

	Grid: 64x64x64					
	Haswell CPU			Xeon Phi		
MPI/OpenMP (M/n) $\rightarrow$	2/12	6/4	12/2	7/32	14/16	56/4
MKL 11.3.2	129	171	182	76	76	83
FFTW	92	155	155	70	71	69
FFTW (rev. 3.3.4, LibSci)	100	154	161	–	–	–

Table II lists the overall performance for M complex 64^3 FFTs executed by M MPI ranks, each of which using n threads. For each "M/n"-combination the "entire" compute node or device is used.[2] Compared to Fig. 4, we could gain the overall

[2]On the Xeon Phi we use only 56 of the 59 CPU cores, as it allows for a better partitioning of the device and because one CPU core is reserved for OS processes and Cray services running on the Xeon Phi.

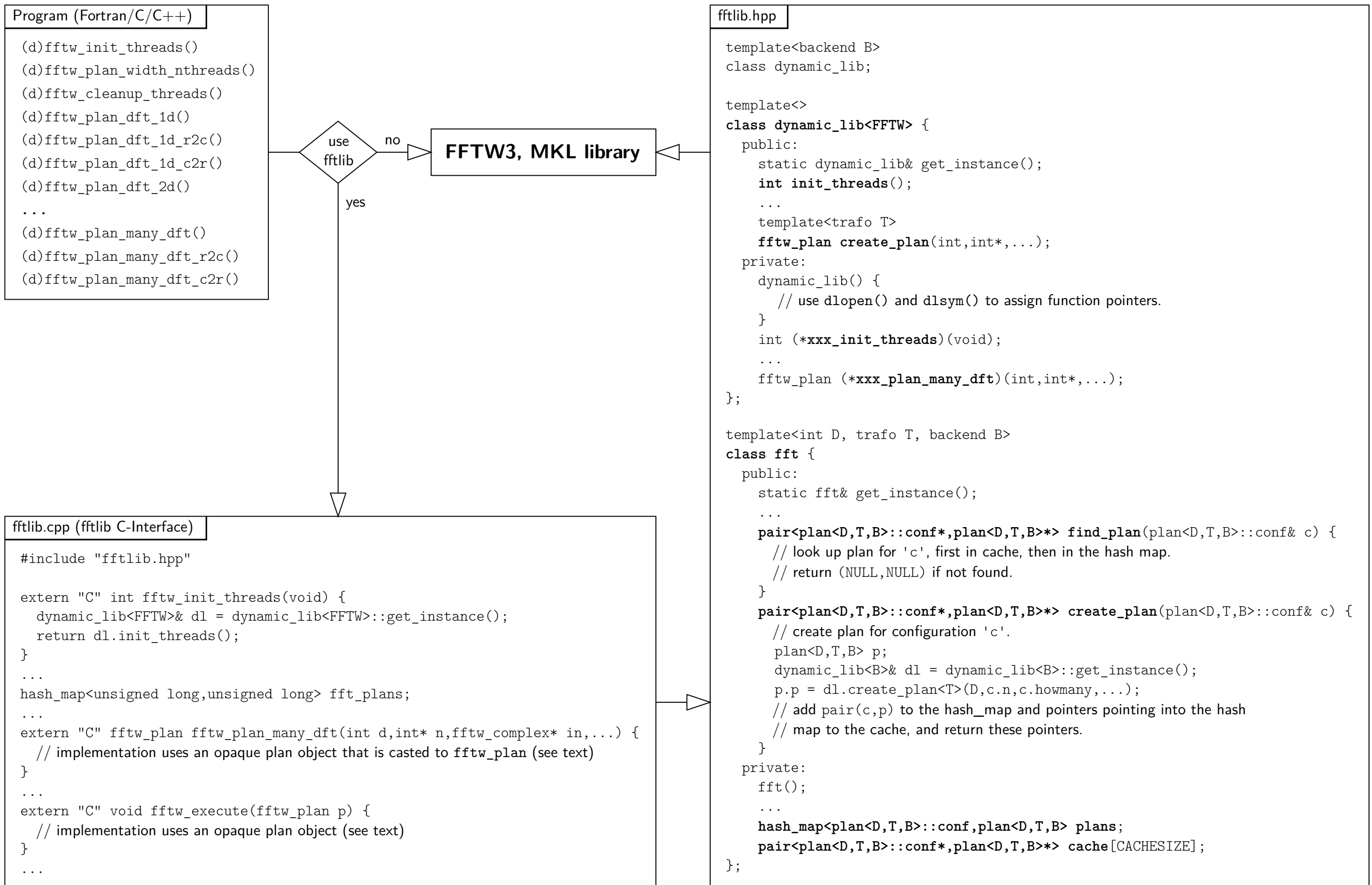


Fig. 5. FFTLIB call interception scheme.

performance for the complex 64^3 FFT by about a factor 4 when using 56 MPI processes together with 4 threads per process. However, there is still a gap of at least 2x between Haswell and Xeon Phi. Seemingly, MKL's FFT optimization for the Xeon Phi does not target small problem sizes. This might change with upcoming MKL releases.

III. FFTLIB

The main issue when using FFTW in VASP remains to be the planning phase. In the previous section, we convinced ourselves of the meaningfulness of planning at least with `FFTW_MEASURE`. Using plans for a certain grid geometry hundreds or thousands of times can compensate for the time spent in plan creation. However, we also observed that requesting plans again and again also consumes an amount of time which is of the order "tens of seconds," as demonstrated in Tab. I. The main issue with FFTW wisdom is the circumstance that plans are not permanently stored as objects within FFTW, but are created every time they are requested. Using wisdom, the plan creation happens much faster, but it is still very expensive.

Our approach within FFTLIB—a C++ template library, developed at ZIB—is to permanently hold the plans (or pointers to the plans) in an internal data structure that allows for fast access. The general idea is illustrated in Fig. 5. From the user perspective nothing changes, that is, in case of using FFTW, all calls `(d)fftw_xxx()` remain the same. When compiling the application with FFTLIB, however, a subset of these calls is intercepted and uses our library.

Dynamic Library (Loader): To eventually redirect FFT calls to the dynamic FFTW or MKL/DFTI library `libxxx.so` from within FFTLIB, we need to manually load that library at runtime, and read those symbols for which FFTLIB provides an alternative/enhanced implementation—otherwise, we would end up in a name conflict. This can be easily done with the `dlopen()` and `dlsym()` system calls.

The `dynamic_lib` template class (Fig. 5) implements that functionality using a singleton pattern. The latter guarantees that at every point in time there is just one instance of that class within the process context. It is instantiated within the static `get_instance()` method and can be accessed/used as follows (with the FFTW library as back-end, for instance):

```
// request instance of dynamic_lib:
dynamic_lib<FFTW>& dl=
      dynamic_lib<FFTW>::get_instance();
// create a plan for a complex 3D-FFT:
dl.create_plan<C2C_64>(3,..);
```

Beside the template parameter `FFTW`, it is planned to provide specializations for `DFTI` (with some MKL specifics). Within

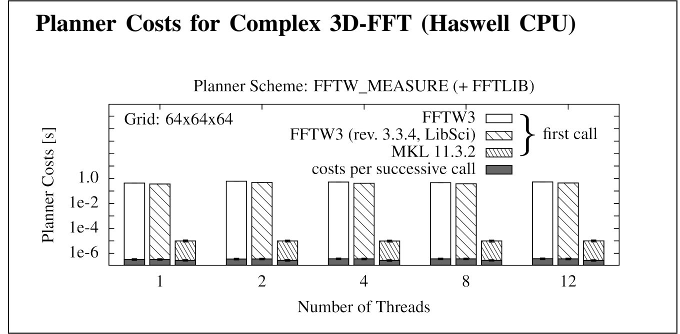
the `dynamic_lib` class, we provide a set of template functions that use the dynamically loaded symbols, and if needed implement different behavior depending on which kind of FFT should be performed. Our current implementation includes `R2R_64`, `R2C_64`, `C2R_64` and `C2C_64` covering any combination of real and complex FFTs with 64-bit double precision.

Plan Reuse: The actual functionality of FFTLIB is implemented in the `fftlib` template class (also using the singleton pattern for instantiation). Template parameters are the dimensionality `D` of the FFT, the type `T` (=`R2R_64|...`) of the FFT, and the FFT back-end `B` (=`FFTW|DFTI`). One of the library's core components is a hash-map (we use STL's unordered map) that stores plans (template class `plan<D,T,B>`) that were created using either the FFTW or MKL/DFTI back-end. For each entry in the hash-map, we use the configuration `c` of the FFT (template class `plan<D,T,B>::conf`) as the key and the plan `p` as the value. The hash value is deduced from the configuration. In addition to the hash-map, FFTLIB uses a cache that holds pointers to those configurations and plans in the map that were recently accessed—pointers to keys and values in the hash-map are not invalidated when altering the map. The size of the cache can be configured.

Every time a plan is requested from within the user program, FFTLIB creates the configuration `c` that is looked up first in the cache, and afterwards in the hash-map if not found in the cache already. The lookup is implemented in `find_plan(plan<D,T,B>::conf& c)`. If the plan cannot be found in any of the two, a new hash-map entry is generated, requesting the plan from the back-end library using `create_plan(plan<D,T,B>::conf& c)`.

Plans `p` returned to the user are equivalent to the address of an opaque plan object, which, beside the actual plan `pp`—residing in FFTLIB's hash-map—holds additional information such as the input and output pointers `in` and `out` for the FFT computation. This is conform to what is returned by the FFTW library and MKL. When executing the plan `p`, for instance, via `(d)fftw_execute(p)`, FFTLIB detects whether `p` has been created by FFTLIB or not. If so, it recovers `pp` as well as `in` and `out`, and calls `fftw_execute_dft(pp,in,out)`. If the plan has not been created by FFTLIB, it is forwarded to the back-end FFT library using the `dynamic_lib` class.

Plan Creation using FFTLIB (Performance): Figure 6 illustrates the costs for requesting a plan for a complex 64^3 FFT computation using FFTLIB with FFTW and MKL as back-end libraries. Compared to Figs. 1 and 2, the costs for the first planner call remain the same except for a small overhead due to accessing the hash-map—as for the first request the plan is not in the map, the call is forwarded to the back-end library, and the returned plan is inserted into the map. All successive plan requests, however, are served by FFTLIB within 0.3 to 0.4 micro seconds on the Haswell CPU, and 3 to 4 micro seconds on the Xeon Phi. We can note a performance gain of about 40x and 20x for MKL on the Haswell CPU and Xeon Phi, respectively, and about three orders of magnitude for FFTW on both of the two platforms.



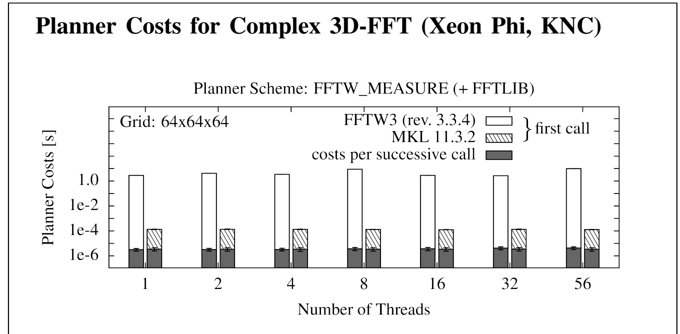


Fig. 6. Planner costs for a complex 64^3 FFT using FFTLIB on Haswell CPU and Intel Xeon Phi. The costs for the first planner call equal those given in Figs. 1 and 2, but all successive planner calls are significantly faster due FFTLIB's cache and hash-map lookup.

All values given in Fig. 6 are for requesting the same plan again and again, hence reflecting the time to access FFTLIB's cache. In order to assess the hash-map lookup costs, we set the cache size to 1, placed $n > 1$ plans in the map, and continuously requested them one after another. For $n = 50$, we measure access times between 0.5 and 0.7 micro seconds on the Haswell CPU, and 5 to 6 micro seconds on Xeon Phi, respectively.

Figure 7 illustrates the fraction of i) the first planner call, ii) all successive planner calls, iii) plan execution, and iv) plan destruction on the total execution time for 100 complex 72^3 FFTs on the Haswell CPU. For plan creation with FFTW, the `FFTW_MEASURE` planner scheme has been used. It can be seen that the time spent in plan creation (first and successive planner calls) dominates if multiple threads are used (total execution times are noted on top of the stacked bars). With FFTLIB, the fraction of the time spent in all successive planner calls—now served by FFTLIB—however, can be reduced significantly, which for the 8 and 12 thread execution gives about 1.3x and 1.45x overall performance gain, respectively. For real-world applications using FFTW together with FFTLIB, the fraction of the planner step can be further reduced when plans are reused several thousands of times. Without FFTLIB, the costs for all successive planner calls would sum up to a non-negligible value.

Using FFTLIB in VASP: Table III lists the VASP program

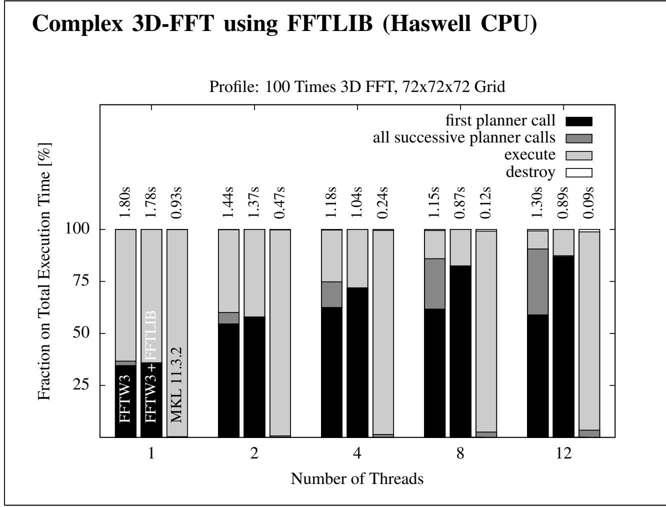


Fig. 7. Fraction of i) the first planner call, ii) successive planner calls, iii) plan execution, and iv) plan destruction on the total execution time for a complex 72^3 FFT w/ and w/o FFTLIB on Haswell CPU.

execution times using FFTW (self-compiled and LibSci) together with FFTLIB. A direct comparison against values given in Tab. I shows that the planner costs now include only those for the first planner call(s). For runs with MKL, the effect is insignificant. In case of using 4 OpenMP threads per MPI rank, it seems that FFTLIB introduces overheads that result in a small performance degradation. However, runs with FFTW show performance gains for the entire application ranging from 1.05x to 1.4x when multiple threads are used. Furthermore, FFTW+FFTLIB runs can close up to runs with MKL.

Ball$\leftrightarrow$Cube-FFT: The central quantities in plane-wave electronic structure codes like VASP are the so-called one-electron orbitals $\psi_n(\mathbf{r})$, and the electronic density $\rho(\mathbf{r}) = \sum_n \psi_n^*(\mathbf{r})\psi_n(\mathbf{r})$. The one-electron orbitals are expressed in terms of a basis set of plane-waves, and their Fourier components $\psi_n(\mathbf{k})$ are stored. This basis set is commonly limited to those Fourier components with a reciprocal space vector below a certain cutoff length G (chosen by the user, as a input parameter of the calculation): $\psi_n(\mathbf{k}) = 0 \,\forall\, |\mathbf{k}| > G$.

TABLE III
VASP PROGRAM EXECUTION TIME FOR THE PDO2 SETUP ON 4 CRAY XC40 COMPUTE NODES WITH INTEL XEON E5-2680V3 (HASWELL) CPUS. 3D-FFT COMPUTATION HAPPENS EITHER WITH FFTW OR MKL TOGETHER WITH FFTLIB. IN ALL CASES 24 MPI RANKS ARE USED, EACH WITH T=1 OR T=4 OPENMP THREADS. COMPARE THE RUNTIMES (AND TIME SPENT IN 3D-FFT COMPUTATION) AGAINST THOSE GIVEN IN TAB. I.

	Setup: PdO2					
	MKL 11.3.2		FFTW		FFTW (LibSci)	
	T=1	T=4	T=1	T=4	T=1	T=4
Total	145.5s	84.8s	152.6s	86.5s	153.3s	90.0s
3D-FFT	23.4s	10.0s	29.0s	11.3s	29.6s	11.7s
+planner	0.3s	0.3s	0.8s	0.9s	0.8s	0.9s
+execute	22.4s	9.7s	28.2s	10.4s	28.8s	10.8s

Consequently, the electronic density has Fourier components with reciprocal space vectors up to a length of $2G$ (the density is constructed from products of ψ_n). This situation is illustrated by Fig. 8A: the ball represents the Fourier components of ψ_n (non-zero only for reciprocal vectors up to a length G), centered at the origin of a $(4G \times 4G \times 4G)$ cube. This cube represents the regular FFT grid that is large enough to contain all non-zero Fourier components of the electronic density. It encompasses a sphere with radius $2G$.

In the course of an electronic structure calculation both quantities are repeatedly shuffled back and forth between real space and reciprocal space by means of FFTs. To limit the amount of work spent on the FFTs of ψ_n, most plane-wave electronic structure codes do not perform a straightforward 3D-FFT of the cube in Fig. 8A. Instead most codes implement a (1D$\times$1D$\times$1D) ball$\leftrightarrow$cube-FFT. This is illustrated in Fig. 8B-D: the first series of 1D-FFTs is taken along the x-direction and is limited to the space depicted as a red rod in Fig. 8B. This rod with diameter G is just large enough to completely encompass the sphere with non-zero Fourier components of ψ_n. The second series of 1D-FFTs is performed along the y-direction on that part of space that is depicted as a red slab in Fig. 8C. The slab completely encompasses the rod of the previous FFTs. The final series of 1D-FFTs is taken along the last remaining direction, the z-direction, and has to be taken over the complete cube, as shown in Fig. 8D. The progression B$\rightarrow$D is the "forward"-FFT from "real"-space to "reciprocal"-space: $\psi(\mathbf{r}) \rightarrow \psi(\mathbf{k})$. The corresponding "back"-transform would be the progression D$\rightarrow$B.

FFTLIB incorporates this idea by splitting the 3D-FFT into a 2D- and a 1D-FFT that are performed in sequence—breaking the 2D-FFT down into two 1D-FFTs, and implementing the steps B and C in Fig. 8 seems inferior to 2D-FFT computation, as the letter is well optimized already. The 2D-FFT effectively combines steps B and C in Fig. 8 and the 1D-FFT corresponds to step D. In between the data layout has to be adapted so that after the 2D-FFT all components in z-direction are contiguous in memory. The first transpose operation T_{xz} interchanges $x \leftrightarrow z$ before the 1D-FFT, and the seconds transpose T_{zx} after the 1D-FFT works the other way around, recovering the original layout.

Our implementation of the ball$\leftrightarrow$cube-FFT in FFTLIB determines the number of "zero"-layers in z-direction automatically and creates a plan for the 2D-FFT covering the "non-zero"-layers. The respective plans are stored in the internal hash-map and can be reuse for other ball$\leftrightarrow$cube-FFTs having the same x- and y-extent and the same number of "non-zero"-layers in z-direction. The latter does not necessarily mean the same z-extent. Additionally, we allow to skip the T_{zx} transpose operation. In the VASP application, we then need to adapt the computation(s) after the 3D-FFT to operate on the "zyx" instead of the "xyz" layout.

Figure 9 illustrates the performance of FFTLIB's ball$\leftrightarrow$cube FFT using the FFTW back-end and without the last transpose operation. Across all the different grid geometries, it can be noted that the ball$\leftrightarrow$cube approach achieves a higher

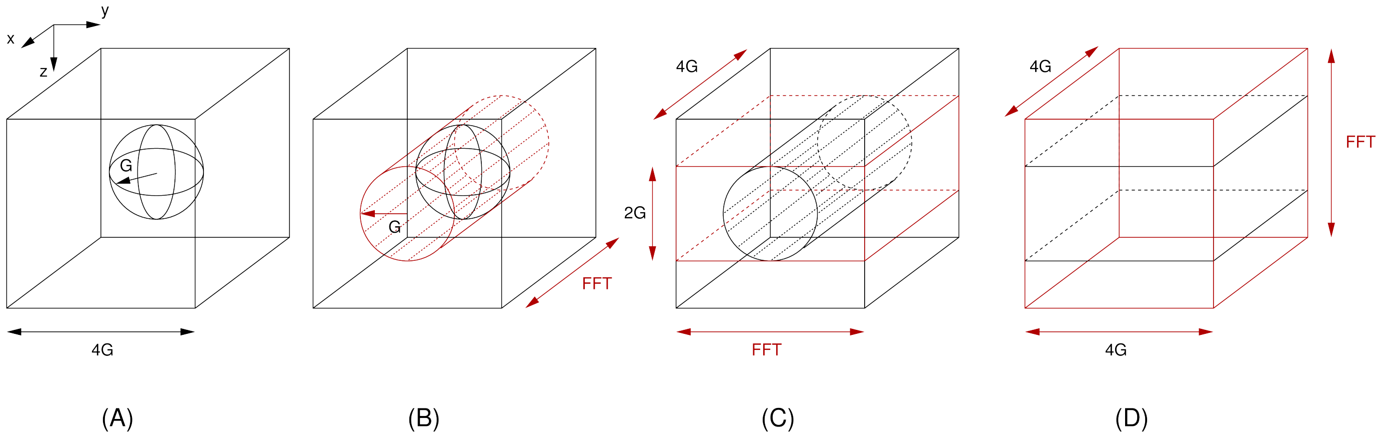


Fig. 8. (A): Fourier components of the one-electron orbitals (ball) inside a cube that represents the regular FFT grid large enough to contain all non-zero Fourier components of the electronic density, and (B)-(D): (1D×1D×1D) ball↔cube-FFT.

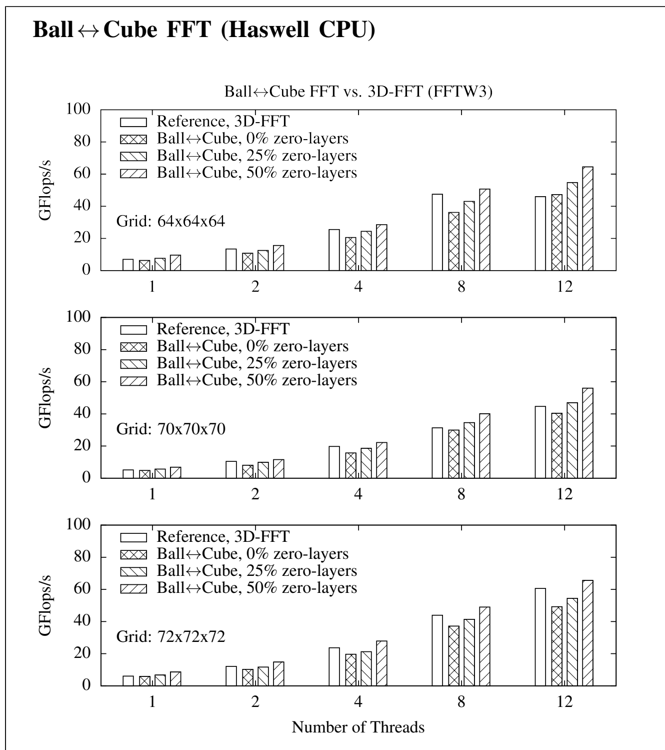


Fig. 9. Performance of FFTLIB's ball↔cube FFT compared against 3D-FFT computation.

compute performance than conventional 3D-FFT computation if a significant fraction of the z-layers contains zeros—in our test cases we consider 0%, 25% and 50% of the z-layers be filled with zeros. A maximum performance gain of 1.4x could be achieved for the complex 64^3 FFT using 12 threads.

Running the same setups on the Xeon Phi, however, gave poor performance. We assume that our implementation of the transpose operation is somehow unfortunate for execution on the Xeon Phi—the majority of time is spent in the transpose. We therefore do not list performance values for the Xeon Phi.

High Bandwidth Memory: Our implementation of the transpose operation uses one additional buffer (we implement the transpose out-of-place) that is managed within FFTLIB. On the upcoming Intel Xeon Phi KNL platform, an additional memory layer, the high bandwidth memory (HBM), will be available. Application tuning for KNL thus involves moving selected data (-structures) to the HBM when it is used as a "scratchpad."[3]

We extended our implementation using `hbw_malloc()` to allocate memory directly in the HBM—this functionality is accessible through the `memkind` library [2]. Running the program on a dual-socket CPU system, we place all processes on socket 0 using `numactl --cpunodebind=0` and allocate memory on socket 1 using `numactl --membind=1`. By setting the environment variable `MEMKIND_HBW_NODES=0`, we can use the local memory on socket 0 as HBM. In this way, we can emulate the presence of the HBM. On the Cray-XC40 compute nodes, we measure about 10% performance gain for the transpose operation(s) when using HBM instead of the distant memory on socket 1.

IV. Summary and Outlook

We have demonstrated for two relevant aspects in the context of 3D-FFT computation—plan reuse and composed 3D-FFT—how to effectively approach them on current computer platforms with inherent need for multi-threading.

One of the main issues in VASP (and maybe other DFT codes, too) when using the FFTW library is the significant amount of time spent in planning the FFT computation. The latter increases with the number of threads to be used for the computation. We introduced a plan caching scheme as a component of our FFTLIB—a C++ template library that intercepts FFTW calls—that reduces these times measurably. For the VASP application, we were able to gain the multi-threaded application performance by up to a factor 1.4x for a selected input using 4 Cray-XC40 compute nodes.

[3]The HBM can also be used as an additional cache between the on-chip last level cache and the DDR4 RAM.

Furthermore, FFTLIB provides additional functionality like composed FFT computation with "ball↔cube" optimization, and support for the high bandwidth memory (HBM) on the upcoming Intel Xeon Phi Knights Landing platform. Using synthetic benchmark kernels, we achieved up to 1.4x performance gain over a conventional 3D-FFT computation on a Cray-XC40 compute node, and measured up to 10% performance increase for an out-of-place transpose operation as part of the "ball↔cube" FFT when using HBM.

One of our next steps is the integration of FFTLIB's "ball↔cube" FFT into VASP, which currently did not happen as it requires adaptions of the data layout in the code. We also plan to implement an auto-tuning mechanism into FFTLIB, that allows for a re-compilation of some components of FFTLIB using runtime information. First results on tuning the transpose operation, used for the "ball↔cube" FFT, show a performance gain of up to 10%.

Disclaimer

All our measurements have been carried out on the described hardware and with the latest libraries available at the time of this writing. We used the libraries to our best knowledge. For FFTW we incorporated suggestions and recommendations given on the official FFTW webpage: `http://www.fftw.org`. The VASP version used for benchmarking is a prototype version that is not publicly available at the current time. The integration of FFTLIB into VASP happened in agreement with the VASP developers and might become part of VASP in upcoming releases.

Acknowledgment

This work is partially supported by Intel Corporation within the "Research Center for Many-core High-Performance Computing" (IPCC) at ZIB, and by Cray Computer Deutschland within a joint research project.

References

[1] FFTW—Fastest Fourier Transform in the West. Official webpage: `http://www.fftw.org`, 2016.

[2] Memkind library. `https://github.com/memkind/memkind`, 2016.

[3] Constantine Bekas, Alessandro Curioni, and Wanda Andreoni. *Applied Parallel Computing. State of the Art in Scientific Computing: 8th International Workshop, PARA 2006, Umeå, Sweden, June 18-21, 2006, Revised Selected Papers*, chapter New Scalability Frontiers in Ab Initio Electronic Structure Calculations Using the BG/L Supercomputer, pages 1026–1035. Springer Berlin Heidelberg, Berlin, Heidelberg, 2007.

[4] Andrew Canning, John Shalf, Lin-Wang Wang, Harvey J. Wasserman, and Manisha Gajbe. A comparison of different communication structures for scalable parallel three dimensional ffts in first principles codes. In Chapman et al. [5], pages 107–116.

[5] Barbara M. Chapman, Frédéric Desprez, Gerhard R. Joubert, Alain Lichnewsky, Frans J. Peters, and Thierry Priol, editors. *Parallel Computing: From Multicores and GPU's to Petascale, Proceedings of the conference ParCo 2009, 1-4 September 2009, Lyon, France*, volume 19 of *Advances in Parallel Computing*. IOS Press, 2010.

[6] Manisha Gajbe and Andrew Canning and John Shalf and Lin-Wang Wang and Harvey Wasserman and Richard Vuduc. Auto-tuning distributed-memory 3-dimensional fast Fourier transforms on the Cray XT4. In *Proc. Cray User's Group (CUG) Meeting*, Atlanta, GA, USA, May 2009.

[7] S. Song and J. K. Hollingsworth. Scaling parallel 3-d fft with non-blocking mpi collectives. In *Latest Advances in Scalable Algorithms for Large-Scale Systems (ScalA), 2014 5th Workshop on*, pages 1–8, Nov 2014.

[8] Andrew Sunderland, Stephen Pickles, Milos Nikolic, Aleksandar Jovic, Josip Jakic, Vladimir Slavnic, Ivan Girotto, Peter Nash, and Michael Lysaght. An Analysis of FFT Performance in PRACE Application Codes. Technical report, PRACE Whitepaper, 2012.

[9] J.M. Thijssen. *Computational Physics*. Cambridge University Press, 1999.

[10] Weber, Valery and Bekas, Costas and Laino, Teodoro and Curioni, Alessandro and Bertsch, Adam and Futral, Scott. Shedding Light on Lithium/Air Batteries Using Millions of Threads on the BG/Q Supercomputer. In *IPDPS*, pages 735–744. IEEE Computer Society, 2014.