Received: 19 January 2018 | Revised: 20 May 2018 | Accepted: 29 June 2018
DOI: 10.1002/cpe.4905

WILEY

SPECIAL ISSUE PAPER

Accelerating the computation of FLAPW methods on heterogeneous architectures

Davor Davidović[1] | **Diego Fabregat-Traver**[2] | **Markus Höhnerbach**[2] | **Edoardo Di Napoli**[2,3]

[1]Ruđer Bošković Institute, 10000 Zagreb, Croatia
[2]Aachen Institute for Advanced Study in Computational Engineering Science, RWTH Aachen, 52062 Aachen, Germany
[3]Jülich Supercomuting Center and JARA, 52428 Jülich, Germany

Correspondence
Davor Davidović, Ruđer Bošković Institute, Bijenička cesta 54, 10000 Zagreb, Croatia.
Email: davor.davidovic@irb.hr

Funding information
Ministry of Science and Education of the Republic of Croatia; Deutsche Akademische Austauschdienst (DAAD); Bundesministeriums für Bildung und Forschung (BMBF) through project "PPP Kroatien", Grant/Award Number: 57216700; Jülich Aachen Research Alliance-High Performance Computing Deutsche Forschungsgemeinschaft (DFG), Grant/Award Number: GSC 111

Summary

Legacy codes in computational science and engineering have been very successful in providing essential functionality to researchers. However, they are not capable of exploiting the massive parallelism provided by emerging heterogeneous architectures. The lack of portable performance and scalability puts them at high risk, ie, either they evolve or they are destined to be executed on older platforms and small clusters. One example of a legacy code which would heavily benefit from a modern redesign is FLEUR, a software for electronic structure calculations. In previous work, the computational bottleneck of FLEUR was partially reengineered to have a modular design that relies on standard building blocks, namely, BLAS and LAPACK libraries. In this paper, we demonstrate how the initial redesign enables the portability to heterogeneous architectures. More specifically, we study different approaches to port the code to architectures consisting of multi-core CPUs equipped with one or more coprocessors such as Nvidia GPUs and Intel Xeon Phis. Our final code attains over 70% of the architectures' peak performance, and outperforms Nvidia's and Intel's libraries. On JURECA, the large tier-0 cluster where FLEUR is often executed, the code takes advantage of the full power of the computing nodes, attaining 5× speedup over the sole use of the CPUs.

KEYWORDS

FLAPW, FLEUR, hybrid BLAS, multiGPU, Phi, portability, scalability

1 | INTRODUCTION

Modern software is typically designed with modularity, portability, and scalability in mind. Following these three principles allows codes to run efficiently on a variety of large parallel clusters. At the other end of the spectrum, many legacy codes present a rigid structure that often is the result of the direct implementation of mathematical formulas. Such a structure limits the efficient execution of these codes on emerging computing architectures and constitutes a substantial portability challenge. One example of such legacy codes is FLEUR, a software for electronic structure calculations developed during the last three decades at the Forschungszentrum Jülich (Germany).[1] In FLEUR, the generation of the Hamiltonian and Overlap matrices depends on a complex series of operations originally encoded in multiple nested loops distributed over a number of separate routines. In a recent work, it was shown how these operations can be rearranged in terms of matrix linear algebra kernels opening the way to a greatly enhanced performance portability of the code.[2] In this paper, we close the gaps still open in the previous work, and demonstrate that restructuring a complex mathematical formulation in terms of simple linear algebra algorithms indeed facilitates quick performance portability to complex heterogeneous architectures consisting of multi-core shared-memory CPUs in combination with either GPUs or Phi co-processors.

Nowadays, massively-parallel heterogeneous architectures have become ubiquitous, and legacy code must be re-engineered and modernized to make an efficient use of such architectures. A critical factor in writing long-lasting scientific code is to have a modular design where, at the bottom layers, the computational bottlenecks are written in terms of efficient kernels.[3-5] Examples of such kernels are fast Fourier transforms, matrix-matrix products, and eigensolvers, which are provided by a number of commercial and academic libraries, such as Intel MKL,[6] MAGMA,[7] cuBLAS,[8] cuFFT, and ELPA,[9,10] just to name a few. Over time, standardized libraries include routines implementing the most critical kernels,

which are highly-optimized for a variety of computing platforms. Applications codes designed to rely of such routines can automatically benefit from them across distinct architectures. The most outstanding example is the Basic Linear Algebra Subprograms (BLAS) library. In reality, BLAS is an interface that defines a number of common building blocks (including matrix-matrix products and linear systems solves), which was born after the realization that standardization was critical to increase productivity and portability. Today, BLAS is the first library ported and optimized for each new architecture, and is used as a building block for many other higher-level libraries. Therefore, writing software on top of BLAS is a guarantee for performance portability and scalability.

In order to demonstrate the benefits of modern engineering and portability, Di Napoli et al underwent a major effort to reformulate one of the computational bottlenecks of the FLEUR code,[2] ie, the construction of the so-called Hamiltonian and Overlap matrices. The concepts of abstraction and encapsulation were lacking in the original code, and the different modules were tightly coupled. Before re-writing the code from scratch, the authors analyzed the mathematical model behind the code and rewrote the formulas at a higher level, resulting in a series of simple matrix-based linear operations. These matrix operations were then cast into kernels provided by the BLAS and LAPACK libraries. The resulting new algorithm (HSDLA) attains speedups ranging from 1.5× to 2.5× with respect to the corresponding FLEUR code on multi-core CPUs despite it executes one order of magnitude more FLOPs. More importantly, the new code enables performance portability with limited programming effort which was demonstrated in our previous work.[11] In the latter, we analyzed the performance portability of the HSDLA algorithm on heterogeneous architectures consisting of multi-core shared-memory CPUs and one or more Graphical Processing Units (GPUs). In particular, we showed that the implementation of the HSDLA algorithm can be ported to emerging architectures with minor or relatively straightforward code changes by employing highly-optimized libraries such as NVIDIA's CUBLASXT or Intel's MKL. However, this approach attained only up to 69% of the theoretical system's peak performance, thus still significantly underutilizing the available resources.

In this paper, we close the performance gaps and demonstrate the worth of the initial HSDLA algorithm by porting it to complex heterogeneous architectures consisting of multi-core shared-memory CPUs together with GPUs or Xeon Phi co-processors. We first attempt a (almost) plug-and-play solution and quantify the benefits and limitations to make the best of these architectures. Then, we study alternative approaches to leverage existing optimized BLAS kernels for each component to obtain efficient hybrid routines. Finally, we identify some scalability bottlenecks in sequential and poorly-scalable sections of the HSDLA algorithm and propose solutions to limit their impact in the overall scalability. Our experimental tests, obtained on a number of distinct architectures, illustrate that the resulting code makes an efficient use of the computing resources and has the potential to provide a consistent boost for the FLEUR code.

1.1 | Contributions

Our main contributions can be divided in few steps. First, once rewritten in terms of a standardized interface, we perform a portability study of the code and isolate a number of remaining issues limiting a successful port to heterogeneous architectures. In order to overcome these limitations and quantify the improvements, several different approaches are considered, implemented, and tested. Finally, we assembled an improved high-performance and scalable implementation of the HSDLA code and tested it to compute Hamiltonian and Overlap matrices derived from realistic Density Functional Theory problems.

1.2 | Organization of this paper

The remainder of this paper is organized as follows. Section 2 provides the background on Density Functional Theory and the math behind the computation to generate the Hamiltonian and Overlap matrices. Section 3 gives an overview of the HSDLA algorithm for the generation of these matrices and Section 4 provides overview of the related work. The improvements of the HSDLA algorithm as well as our implementation on the heterogeneous architectures are described in Section 5. Section 6 presents experimental results for a collection of test cases and hybrid architectures. Finally, Section 7 draws conclusion and discusses future research directions.

2 | COMPUTING THE HAMILTONIAN AND OVERLAP MATRICES IN FLEUR

FLEUR is a scientific computing code, well-known in the community of condensed matter physicists, for calculating ground-state and excited-state properties of solids. This is one of few Density Functional Theory (DFT) codes, which computes the electron structure of an atomic compound using the Full-potential Linearized Augmented Plane Wave (FLAPW) discretization approach. In this section, we give a brief introduction of the fundamental mathematical aspects of DFT, and its practical realization by the FLAPW method. The material presented does not require a special background knowledge in quantum physics and allows the non-specialist to understand the origin of one of the most computational intensive parts of the FLEUR code; the construction of the Hamiltonian and the Overlap matrices.

2.1 | DFT and the FLAPW method

In the last two decades, DFT[12,13] has become the "standard model" in Materials Science. Within the DFT framework, it is possible to simulate the physical properties of complex quantum mechanical systems made of few dozens up to few hundreds of atoms. The core of the method relies on the

simultaneous solution of a set of partial differential equations. These equations are determined by a Hamiltonian operator $\hat{H}$ containing a so-called effective potential $V_0[n]$, where $n(\mathbf{r})$ is a function of the radial coordinate $\mathbf{r}$ known as the one-particle electron density.

In turn, the solutions of the equations $\psi_i(\mathbf{r})$ determine the one-particle electron density $n(\mathbf{r})$ used in calculating the effective potential V_0. Because of the dependence of the Hamiltonian on the set of $\psi_i(\mathbf{r})$ through $n(\mathbf{r})$, the governing equations are highly non-linear and cannot be solved directly

$$\begin{aligned} \hat{H}\psi_i(\mathbf{r}) &= \left(-\frac{\hbar^2}{2m}\nabla^2 + V_0(\mathbf{r})\right)\psi_i(\mathbf{r}) = \epsilon_i\psi_i(\mathbf{r}) \quad ; \quad \epsilon_1 \le \epsilon_2 \le \dots \\ n(\mathbf{r}) &= \sum_i^N f_i|\psi_i(\mathbf{r})|^2. \end{aligned} \tag{1}$$

In practice, the equations above, also known as Kohn-Sham (KS),[14] are solved by self-consistent iteration; an initial guess for $n_0(\mathbf{r})$ is used to calculate the effective potential V_0 which, in turn, is inserted in Equation (1) whose solutions, $\psi_i(\mathbf{r})$, are used to compute a new charge density $n_1(\mathbf{r})$. Convergence is checked by comparing the new density to the starting one. When convergence is not reached, an opportune mixing of the two densities is selected as a new guess, and the process is repeated.

In principle, the theory only requires as input the quantum numbers and the positions of the atoms that are part of the investigated system. In practice, there is a plethora of DFT methods, which depends on the discretization used to parameterize the KS equations. The discretization in the Full-potential Linearized Augmented Plane Wave (FLAPW) method[15,16] is based on a plane wave expansion of $\psi_{\mathbf{k},\nu}(\mathbf{r})$, where the momentum vector $\mathbf{k}$ and the band index ν replace the generic index i. The $\mathbf{k}$-point wave function $\psi_{\mathbf{k},\nu}(\mathbf{r}) = \sum_{|\mathbf{G}+k|\le \mathbf{K}_{max}} c^{\mathbf{G}}_{\mathbf{k},\nu}\varphi_{\mathbf{G}}(\mathbf{k},\mathbf{r})$ is expanded in terms of a finite basis set $\varphi_{\mathbf{G}}(\mathbf{k},\mathbf{r})$ indexed by the vectors $\mathbf{G}$ lying in the lattice reciprocal to configuration space up to a chosen cut-off value $\mathbf{K}_{max}$. In FLAPW, the physical (configuration) space of the simulation cell is divided into spherical regions, called Muffin-Tin (MT) spheres, centered around atomic nuclei, and interstitial areas between the MT spheres. The basis set $\varphi_{\mathbf{G}}(\mathbf{k},\mathbf{r})$ takes a different expression depending on the region

$$\varphi_{\mathbf{G}}(\mathbf{k},\mathbf{r}) \propto \begin{cases} e^{i(\mathbf{k}+\mathbf{G})\mathbf{r}} & \text{Interstitial} \\ \sum_{l,m}\left[A^{a,\mathbf{G}}_{l,m}(\mathbf{k})u^a_l(r) + B^{a,\mathbf{G}}_{l,m}(\mathbf{k})\dot{u}^a_l(r)\right]Y_{l,m}(\hat{\mathbf{r}}_a) & a^{th}\ \text{Muffin Tin}, \end{cases} \tag{2}$$

where the coefficients $A^{a,\mathbf{G}}_{l,m}(\mathbf{k})$ and $B^{a,\mathbf{G}}_{l,m}(\mathbf{k})$ are determined by imposing continuity of $\varphi_{\mathbf{G}}(\mathbf{k},\mathbf{r})$ and its derivative at the boundary of the MTs. Due to this expansion, the KS equations naturally translate to a set of generalized eigenvalue problems $\sum_{\mathbf{G}'}[H_{\mathbf{G},\mathbf{G}'}(\mathbf{k}) - \lambda_{\mathbf{k}\nu}S_{\mathbf{G},\mathbf{G}'}(\mathbf{k})]c^{\mathbf{G}'}_{\mathbf{k},\nu} = 0$ for the coefficients of the expansion $c^{\mathbf{G}'}_{\mathbf{k},\nu}$, where the Hamiltonian and Overlap matrices H and S are given by multiple integrals and sums

$$\{H(\mathbf{k}), S(\mathbf{k})\}_{\mathbf{G},\mathbf{G}'} = \sum_a \iint \varphi^*_{\mathbf{G}}(\mathbf{k},\mathbf{r})\{\hat{H}, I\}\varphi_{\mathbf{G}'}(\mathbf{k},\mathbf{r})\mathrm{d}\mathbf{r}. \tag{3}$$

Since the set of basis functions in Equation (2) is implicitly labeled by the values, the variable $\mathbf{k}$ takes in the Brillouin zone, there are multiple Hamiltonian and Overlap matrices, one for each independent $\mathbf{k}$-point.

2.2 | Building H and S

Without loss of generality, we can abstract from the $\mathbf{k}$-point index and recover an explicit formulation of the $H_{\mathbf{G},\mathbf{G}'}$ and $S_{\mathbf{G},\mathbf{G}'}$ matrices by substituting Equation (2) in Equation (3) and carrying out the multiple integration. The computation is particularly complex within the MT regions where the initialization of the Hamiltonian and Overlap matrices is by far the most computationally intensive task. By exploiting the properties of the basis functions, the H and S matrices are directly expressed as functions of the set of A and B coefficients

$$(S)_{\mathbf{G}',\mathbf{G}} = \sum_{a=1}^{N_A}\sum_{l,m}\left(A^{a,\mathbf{G}'}_{l,m}\right)^* A^{a,\mathbf{G}}_{l,m} + \left(B^{a,\mathbf{G}'}_{l,m}\right)^* B^{a,\mathbf{G}}_{l,m}\|\dot{u}^a_l\|^2 \tag{4}$$

$$\begin{aligned} (H)_{G',G} = &\sum_{a=1}^{N_A}\sum_{L',L}\left(\left(A^{a,\mathbf{G}'}_{L'}\right)^* T^{[AA]}_{L',L;a} A^{a,\mathbf{G}}_{L}\right) + \left(\left(A^{a,\mathbf{G}'}_{L'}\right)^* T^{[AB]}_{L',L;a} B^{a,\mathbf{G}}_{L}\right) \\ &+ \left(\left(B^{a,\mathbf{G}'}_{L'}\right)^* T^{[BA]}_{L',L;a} A^{a,\mathbf{G}}_{L}\right) + \left(\left(B^{a,\mathbf{G}'}_{L'}\right)^* T^{[BB]}_{L',L;a} B^{a,\mathbf{G}}_{L}\right). \end{aligned} \tag{5}$$

Notice that, in Equation (5) for convenience, we have compacted the indexes l, m into L, and expressed the range of the index a over all the distinct atom types N_A. The new matrices $T^{[\cdots]}_{L',L;a} \in \mathbb{C}^{N_L\times N_L}$ are dense and their computation involves multiple integrals between the basis functions and the non-spherical part of the potential V_0 (See appendix A.2 in the work of Di Napoli et al[2] for details). Due to the non-orthornormality of the basis function set (2), the matrix S is non-diagonal, dense, and generically positive definite with the exception of having few very small singular values. On the opposite, H is always non-definite and both matrices are either complex Hermitian or real symmetric.

3 | THE HSDLA ALGORITHM

In the original FLEUR code, the construction of matrices H and S was a direct implementation of mathematical formulae with the focus on the functionality rather than scalability. The performance scalability was significantly improved in the HSDLA algorithm,[2] in which the authors reformulated

Equations (4) and (5) in terms of the coefficients A and B. As a result, the entire construction of matrices H and S is now cast in terms of matrix operands, as shown in Equations (6) and (7)

$$H = \sum_{a=1}^{N_A} \underbrace{A_a^H T^{[AA]} A_a}_{H_{AA}} + \underbrace{A_a^H T^{[AB]} B_a + B_a^H T^{[BA]} A_a + B_a^H T^{[BB]} B_a}_{H_{AB+BA+BB}} \tag{6}$$

$$S = \sum_{a=1}^{N_A} A_a^H A_a + B_a^H U_a^H U_a B_a, \tag{7}$$

where A_a and $B_a \in \mathbb{C}^{N_L \times N_G}$, $T_a^{[\dots]} \in \mathbb{C}^{N_L \times N_L}$, H and $S \in \mathbb{C}^{N_G \times N_G}$ are Hermitian, and $U \in \mathbb{C}^{N_L \times N_L}$ is a diagonal matrix. Typical values for the matrix sizes are $N_A \sim \mathcal{O}(100)$, $N_G \sim \mathcal{O}(1000)$ to $\mathcal{O}(10000)$, and $N_L \sim \mathcal{O}(100)$.

The HSDLA algorithm[2] takes advantage of the matrix formulation to cast the computation in terms of the highly optimized and portable BLAS and LAPACK libraries. The algorithm is illustrated in Algorithm 1. The main ideas behind it are 1) exploiting the symmetries in the operations to reduce the computational cost, 2) casting the computation in terms of efficient BLAS and LAPACK kernels, and 3) combining multiple operations on small matrices together to increase performance and scalability. Although the algorithm exploits symmetry in matrices to reduce computational costs, extra flops are introduced compared to the original version. However, these are fast flops, run in highly-optimized BLAS-3 kernels, leading to significantly lower execution time when compared to the original code.

Algorithm 1 : Computation of the H and S matrices in HSDLA.

1: Backup $\hat{A} = A, \hat{B} = B$
2: // $H_{AB+BA+BB}$
3: **for** $a := 1 \rightarrow N_A$ **do**
4: $Z_a = T_a^{[BA]} A_a$ ▷ (`zgemm`: $8N_L^2 N_G$ Flops)
5: $Z_a = Z_a + \frac{1}{2} T_a^{[BB]} B_a$ ▷ (`zhemm`: $8N_L^2 N_G$ Flops)
6: Stack Z_a to A
7: Stack B_a to B
8: **end for**
9: $H = A^H B + B^H A$ ▷ (`zher2k`: $8N_A N_L N_G^2$ Flops)
10: // S
11: Restore $A = \hat{A}, B = \hat{B}$
12: $S = A^H A$ ▷ (`zherk`: $4N_A N_L N_G^2$ Flops)
13: $B = UB$ ▷ (`scaling`: $2N_A N_L N_G$ Flops)
14: $S = S + B^H B$ ▷ (`zherk`: $4N_A N_L N_G^2$ Flops)
15: // H_{AA}
16: **for** $a := 1 \rightarrow N_A$ **do**
17: **try:**
18: $C_a = Cholesky(T_a^{[AA]})$ ▷ (`zpotrf`: $\frac{4}{3} N_L^3$ Flops)
19: **success:**
20: $Z_a = C_a^H A_a$ ▷ (`ztrmm`: $4N_L^2 N_G$ Flops)
21: Stack Z_a to B_T
22: **failure:**
23: $Z_a = T_a^{[AA]} A_a$ ▷ (`zhemm`: $8N_L^2 N_G$ Flops)
24: Stack Z_a to B_B
25: Stack A_a to A
26: **end for**
27: $H = H + B_T^H B_T$ ▷ (`zherk`: $4N_{A_{\text{HPD}}} N_L N_G^2$ Flops)
28: $H = H + A^H B_B$ ▷ (`zgemm`: $8N_{A_{\neg\text{HPD}}} N_L N_G^2$ Flops)

For implementation purposes, the computation of H is split into two parts, $H_{AB+BA+BB}$ and H_{AA} (Equation (6), which can be cast completely in terms of BLAS-3 operations. The key idea behind the calculation of $H_{AB+BA+BB}$ (lines 3 to 9) is to rewrite the expression as

$$\sum_{a=1}^{N_A} B_a^H \left(T^{[BA]} A_a\right) + \left(A_a^H T^{[AB]}\right) B_a + \frac{1}{2} B_a^H \left(T^{[BB]} B_a\right) + \frac{1}{2} \left(B_a^H T^{[BB]}\right) B_a,$$

where $T^{[BA]}$ is the Hermitian transpose of $T^{[AB]}$, and $T^{[BB]}$ is Hermitian. By doing so, the common expressions in parentheses can be grouped together and substituted with

$$Z_a = T^{[BA]} A_a + \frac{1}{2} T^{[BB]} B_a,$$

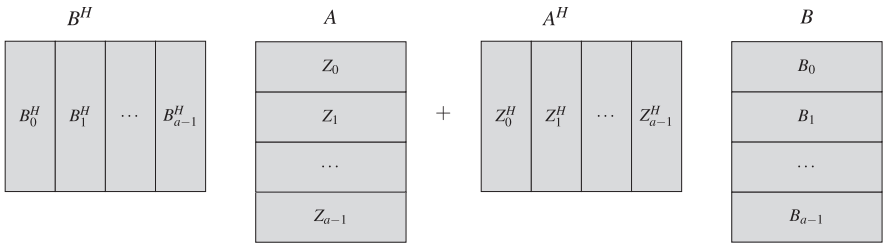

FIGURE 1 Stacking of multiple small matrices Z_a and B_a into larger matrices A and B, respectively. As a result, multiple small matrix products are combined into a larger better performing product

for each atom a (lines 3 to 5). The matrices Z_a and B_a are then stacked together into larger matrices A and B (lines 6 and 7), allowing the remaining computation

$$\sum_{a=1}^{N_A} B_a^H Z_a + Z_a^H B_a$$

to be performed as one single large matrix product (line 9), as depicted in Figure 1.

To compute the second part of the matrix H (lines 16 to 28), the algorithm again takes advantage of the existing symmetries in H_{AA}. Since the $T^{[AA]}$ matrix is Hermitian positive definite (HPD), HSDLA first attempts to compute the Cholesky factorization of $T^{[AA]}$ ($C_a C_a^H = T^{[AA]}$) (line 18), which is then substituted in H_{AA} yielding

$$H_{AA} = \sum_{a=1}^{N_A} A_a^H T^{[AA]} A_a \quad \Rightarrow \quad \sum_{a=1}^{N_A} \underbrace{\left(C_a^H A_a\right)}_{Z_a}^H \left(C_a^H A_a\right).$$

However, while in theory $T^{[AA]}$ is HPD, in practice, due to numerical considerations, the factorization may fail. The algorithm thus divides the computation of H_{AA} in two parts. In case the factorization succeeds, the matrix Z_a is computed (line 20) and stacked at the top part of B (B_T) (line 21). If, on the other hand, the factorization fails, the matrix Z_a (line 23) is stacked at the bottom of matrix B (B_B), and the matrix A_a stacked in A. At the completion of the loop, H is updated with the operation $B_T^H B_T + A^H B_B$ (lines 27 and 28), where the first term ($B_T^H B_T$) exploits the symmetry and computes only half of the output via the BLAS routine `herk`, while the second term ($A^H B_B$) is computed via the BLAS routine `gemm`, which computes a full matrix.

Finally, the computation of S (lines 12 to 14) is more straightforward. First, the product $A^H A$ is computed as one single large product. Then, B is updated with the norms stored in U and then second large product $B^H B$ completes the computation of S.

4 | RELATED WORK

Nowadays, there is an abundance of BLAS-like computational libraries. However, up to our knowledge, none of these libraries does implement BLAS kernels capable of automatically redistribute the computational load between different computational units, such as CPU, GPU and/or Phi. The most popular multi-GPU commercial BLAS library, CUBLASXT, supports hybrid CPU-GPU computation, but so far, only `gemm` kernels are implemented as fully hybrid, thus capable of concurrently divide the computational load between the CPUs and GPUs. An academic alternative to CUBLASXT, called BLASX,[17] requires minor changes to the calls to the BLAS routines, and similar to the CUBLASXT library, takes care of data transfers between CPU and GPU transparently. The BLASX developers reported a significant speedup and a communication volume decrease compared to CUBLASXT, MAGMA, and some other libraries. Despite such gains, only the `dgemm` routine is genuinely hybrid, ie, it has the possibility to schedule and share execution on both CPUs and GPUs. In fact, upon a closer inspection of the BLASX library, one finds that some kernels, which are essential for our algorithm (eg, `zherk`), are implemented by calling the corresponding CUBLASXT kernels. The remaining kernels pertinent to this work have either CPU-only or (multi)GPU-only implementations; only of the two can be exclusively selected at runtime.

Algorithm 2 : Refinement of the computation of H_{AA}.

15: // H_{AA}
16: **for** $a := 1 \rightarrow N_A$ **do**
17: $X_a = T_a^{[AA]} A_a$ ▷ (`zhemm`: $8N_L^2 N_G$ Flops)
18: Stack X_a to X
19: Stack A_a to A
20: **end for**
21: $H = H + A^H X$ ▷ (`zherkx`: $4N_A N_L N_G^2$ Flops)

In the work of Blas et al,[18] the authors reported recent work focused on porting Matlab applications on CPU/GPU system for which they used Armadillo and ArrayFire libraries. Similar to our approach, the authors rely on calling the GPU and CPU optimized kernels implemented in libraries such as Intel MKL, OpenBLAS and NVBLAS. In contrast with our solutions, they select the opportune GPU or CPU kernels at runtime, based on the problem size. Depending on such choice, the resulting computation is executed exclusively on either CPUs or GPUs and not on all available resources.

Further research in hybridization of codes has been done in the work of Haidar et al[19] in which authors developed a lightweight scheduler to offload tasks between different computational units inside a single compute node. The schedule is based on task parallelism and, as was demonstrated in the case of Cholesky, the factorization tasks are executed on the CPU while updating is done on the GPUs. The authors also demonstrated the portability and the scalability of their approach across different accelerators, including multi-GPU and multi-Phi systems. However, there is no report on hybrid BLAS-3 kernels required by HSDLA.

At the time of writing this paper, only one research[20] reported on hybrid GPU-based algorithms for the generation of Hamiltonian and Overlap matrices in FLAPW methods. In that work, the authors aimed at 1000+ atoms systems, too large to be executed on a single compute node thus a distributed block-cyclic setup and distribution of the Hamiltonian and Overlap matrices were implemented. However, the authors based the generation of H and S only on distributed versions of `zgemm` and did not exploit symmetries to decrease computational cost. In addition, the generation is a task-based heterogeneous implementation where particular tasks are scheduled between CPU and the GPUs. In our approach to heterogeneity, we aim at further exploiting data parallelism, not task parallelism.

5 | HSDLA ON HETEROGENEOUS ARCHITECTURES

In this section, we concentrate on the behavior of the algorithm when executed on hybrid architectures consisting of shared-memory CPUs and one or more graphic processor units (GPUs) or Intel Xeon Phi coprocessors. As discussed in the work of Di Napoli et al,[2] once HSDLA is cast in terms of BLAS and LAPACK routines, it attains high performance on multi-core architectures by simply linking to a multi-threaded implementation of these libraries. However, as we demonstrate in this section, attaining such performance levels on hybrid architectures is not straightforward.

We first refine the original HSDLA algorithm, and significantly reduce its computational cost as well as its memory footprint. Then, we identify the limitations of a straightforward port to hybrid architectures and the shortcomings of the existing solutions, and study alternative approaches to attain satisfactory performance levels on such architectures.

5.1 | HSDLA refined

In the effort of exploiting symmetries from the problem, the designers of HSDLA overlooked the fact that matrix H is Hermitian and used the `zgemm` routine (general matrix-matrix product) in Algorithm 1 (line 28) instead of a routine that takes advantage of the symmetry and computes only one triangle of the output matrix. As a result, the algorithm performs $4_{\neg HPD} N_A N_L N_G^2$ redundant flops for computing the upper triangle of the matrix.

With that in mind, and given that some vendor libraries such as NVIDIA's CUBLASXT and Intel's MKL provide a specialized routine to compute only one half of a general matrix product, lines 16 through 28 can be greatly simplified, avoiding the Cholesky factorization altogether, by replacing them with the computation in Algorithm 2. As a result, the total cost of computing H_{AA} is significantly reduced and is dominated by the specialized routine with a cost of $4N_A N_L N_G^2$.

By further examining Algorithm 1 one observes two sequential steps, a backup and a restore of matrices A and B, lines 1 and 11, respectively. Since A and B are overwritten (lines 3 to 9), one copy of each are stored in two additional storage spaces $\hat{A}$ and $\hat{B}$ (line 1) and reused to compute S (lines 12–14) and the rest of H (H_{AA}) (lines 16 to 28). However, the need for additional storage spaces $\hat{A}$ and $\hat{B}$ can be significantly reduced by simply reordering the execution flow as described in Algorithm 3. The computation of S, which only requires one temporary buffer, is moved to the beginning of the algorithm (lines 2 to 4); the result of computing UB is stored in the temporary buffer X, and the original values A and B and are so far preserved. The buffer X is then reused as auxiliary storage in the computation of $H_{AB+BA+BB}$ for stacking Z_a (line 9); note that overwriting B now in line 10 is harmless since its original contents are not needed anymore. Finally, in the computation of H_{AA}, the result of line 15 is also stacked in X, and A is compressed in the A buffer itself. Since all of A, B, and X are of size $16N_A N_L N_G$ bytes (in the order of gigabytes), cutting this requirements in half has a major impact on the memory footprint of the entire algorithm.

5.2 | Limitations of the straightforward approach

As we reported in our previous work,[11] with very limited effort, HSDLA can be easily ported to various computing architectures such as multi-core and multi-GPU systems. To achieve that goal, wrappers have to be implemented around calls to optimized architecture-specific libraries such as cuBLAS, CUBLASXT, or Intel MKL, and function calls changed with those of the optimized libraries for a particular architecture. Although this straightforward approach can quickly bring reasonable performance, it cannot utilize the underlying computational system at their full potential. For example, if CUBLASXT is used on a multi-GPU system, most of the BLAS operations will be offloaded to the GPUs while the CPUs remain idle. Thus, simply using the existing optimized libraries cannot yield the performance increase that is expected by combining the computational power of all CPUs and GPUs of a system.

In order to overcome this issue, we developed heterogeneous BLAS kernels for the routines `zher2k`, `zherk`, and `zgemm/zherkx` that can efficiently divide the workload between the CPUs and the accelerators (GPUs and Phis).

Algorithm 3 : The refined HSDLA without Cholesky and with only one temporary buffer.

1: // S
2: $S = A^H A$ ▷ (`zherk`: $4N_A N_L N_G^2$ Flops)
3: $X = UB$ ▷ (`scaling`: $2N_A N_L N_G$ Flops)
4: $S = S + X^H X$ ▷ (`zherk`: $4N_A N_L N_G^2$ Flops)
5: // $H_{AB+BA+BB}$
6: **for** $a := 1 \rightarrow N_A$ **do**
7: $Z_a = T_a^{[BA]} A_a$ ▷ (`zgemm`: $8N_L^2 N_G$ Flops)
8: $Z_a = Z_a + \frac{1}{2} T_a^{[BB]} B_a$ ▷ (`zhemm`: $8N_L^2 N_G$ Flops)
9: Stack Z_a to X
10: Stack B_a to B
11: **end for**
12: $H = X^H B + B^H X$ ▷ (`zher2k`: $8N_A N_L N_G^2$ Flops)
13: // H_{AA}
14: **for** $a := 1 \rightarrow N_A$ **do**
15: $X_a = T_a^{[AA]} A_a$ ▷ (`zhemm`: $8N_L^2 N_G$ Flops)
16: Stack X_a to X
17: Stack A_a to A
18: **end for**
19: $H = H + A^H X$ ▷ (`zherkx`: $4N_A N_L N_G^2$ Flops)

In the rest of this section, we describe two different designs on how to implement the BLAS kernels on heterogeneous architectures, required by the refined HDSLA. We denoted these two approaches as follows.

- *Static* - The computation between the CPUs and GPUs is divided by pre-computing the number of rows/columns that will be offloaded to the GPUs and then a highly-tuned multi-GPU library is used to compute it.
- *Dynamic* - The computation is based on dynamic scheduling depending on available on-device memory to determine block size and per-device task queuing.

5.3 | Static

This approach targets hybrid architectures with one or more CUDA-based GPU devices, but can be easily extended to Intel Xeon Phi as well as other accelerators, as long as optimized BLAS libraries for these architectures exist. The key idea of this approach is to re-use the existing, highly optimized multi-GPU and multi-threaded libraries in building the hybrid code, with the goal to significantly reduce the programming effort. As such, the code can be quickly tuned for new and emerging GPU architectures, thus improving performance portability on different platforms, while at the same time explores concurrent execution on all available CPUs and GPUs of the system.

For the sake of simplicity, only the `zher2k` routine will be described in details, while the same approach can be easily extended to the other two routines. The `zher2k` performs the Hermitian rank 2k update of the given matrix C in one of the following operations:

$$C := \alpha A B^H + \overline{\alpha} B A^H + \beta C,$$
$$C := \alpha A^H B + \overline{\alpha} B^H A + \beta C,$$

where α and β are scalars, $C \in R^{n \times n}$ is a Hermitian matrix, and A and B are general matrices of size $n \times k$ in the first case and $k \times n$ in the second case. Hereafter, we will observe only the first case; the same algorithm applies for the second, but with A and B of transposed dimensions.

Matrix C is split into four blocks and updated as described in Figure 2. The workload is divided such that the principal submatrix C_{00} with dimension $n_g \times n_g$ is updated on the GPU(s), while the blocks C_{10} and C_{11} (C_{01} is not required since C is Hermitian) are updated on the CPU. The entire update of C can then be performed with only 4 function calls

$$C_{00} = \alpha A_0 B_0^H + \overline{\alpha} B_0 A_0^H + \beta C_{00} \longrightarrow \text{zher2k (GPU)}$$
$$C_{11} = \alpha A_1 B_1^H + \overline{\alpha} B_1 A_1^H + \beta C_{11} \longrightarrow \text{zher2k (CPU)}$$
$$C_{10} = \overline{\alpha} B_1 A_0^H + \beta C_{10} \longrightarrow \text{zgemm (CPU)}$$
$$C_{10} = \alpha A_1 B_0^H + C_{10} \longrightarrow \text{zgemm (CPU)}.$$

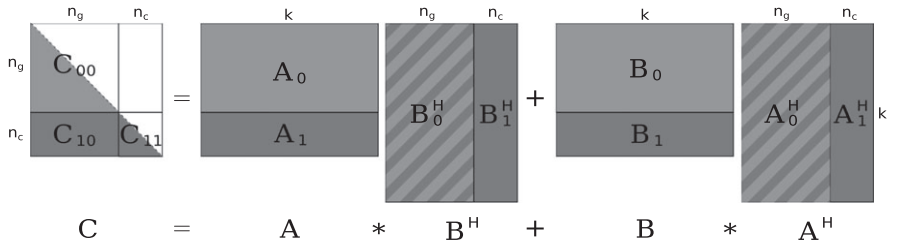

FIGURE 2 Rank 2k update of matrix C (`zher2k`). The blocks are statically split between the CPUs (dark gray) and GPUs (light gray). n_g is the size of principal submatrix offloaded to the GPUs

Since these operations are independent from each other, they can be performed in parallel. For scheduling the workload between CPU and GPU devices, OpenMP with nested parallelism is used. Two OpenMP threads are created, ie, one manages the computation on the CPU and invokes multi-threaded MKL kernels, the other manages the computation on the GPU and invokes CUBLASXT kernels on the GPU.

Although this approach is simple, the main challenge that remains is how to optimally set the size of the principal submatrix that will be offloaded to the GPU. In the ideal case, the CPU and GPU execution times should be completely overlapped. Thus, knowing that the total number of flops for `zher2k` is $8kn^2$, with n the number of rows and columns of C and k number of columns of matrices A and B, n_g, the size of the principal submatrix C_{00}, may be computed as

$$\begin{aligned}
\text{time}_{gpu} &= \text{time}_{cpu} \\
\frac{\text{flops}_{gpu}}{\text{Gflops}_{gpu}} &= \frac{\text{flops}_{cpu}}{\text{Gflops}_{cpu}} \\
\frac{8kn_g^2 + 4}{\text{Gflops}_{gpu}} &= \frac{8k(n-n_g)^2 + 4 + 16kn_g(n-n_g)}{\text{Gflops}_{cpu}}.
\end{aligned}$$

Note that the updating block C_{10} requires two additional calls to `zgemm` which adds extra flops to the count. If $m = \frac{\text{Gflops}_{gpu}}{\text{Gflops}_{cpu}}$, then the size of matrix offloaded to the GPU is

$$n_g^2 = \frac{mn^2 + 4m}{m+1}.$$

As we illustrate in Section 6, the optimal offload between CPU and GPU devices can be computed analytically using this approach or the performance ratio between CPU and GPU can be estimated on smaller problem sizes.

5.4 | Dynamic

While a static split of the matrices and a static assignment of blocks to devices leads to the lowest overheads and highest efficiency, it lacks the flexibility to adapt easily to new circumstances. An alternative is the use of a dynamic scheduling that queries the devices to determine the available on-device memory, calculates appropriate block sizes, and uses per-device queues to distribute the work. To maximize portability, the implementation is split into a device-specific and a device-independent part. This allows the dynamic code to target both the Xeon Phi coprocessor and CUDA GPUs. The device-independent part contains all the workload distribution, scheduling, the BLAS operation and hybrid calculation logic. As for the device-dependent code, the Xeon Phi implementation builds on the hStreams library and MKL, while the GPU implementation relies on CUDA and cuBLAS.

The device-independent logic works as follows. First, each matrix is split into square blocks. The block size is chosen to split the matrices evenly, fit multiple blocks into GPU memory, and follow vendor recommendations (eg, for the first generation Xeon Phi accelerators the block size should be divisible by 64 but not 256). Next, the scheduler determines the individual operations to be performed on each of the submatrices, and schedules them to the devices in a round-robin fashion. Each block is packed into contiguous buffers and streamed to the devices; the results are unpacked once the calculation and the reverse transfer complete. The packing of the blocks is necessary for the Xeon Phi, since it lacks support for copies from 2D arrays, but it is avoided on GPUs. If the scheduler notices that a device is not yet ready to accept work because its queue is full, it skips that devices. Once the memory of the devices are filled up, and the CPU is not needed to drive the devices, the CPU starts computing block operations using its own computational resources.

The remaining challenge is to improve utilization at the very start of a calculation, and at the very end. There, it is desirable to reduce block sizes and sacrifice kernel efficiency for load balance. However, implementing such a scheme requires sophisticated models of transfer and compute efficiency, and it is left to future work.

TABLE 1 Problem sizes for NaCl, AuAg, and TiO_2, for a variety of K_{max} values. The value of N_G varies with K_{max}

Test case	N_A	N_L	N_G: $K_{max} = 2.5$	$K_{max} = 3.0$	$K_{max} = 3.5$	$K_{max} = 4.0$
NaCl	512	49	2 256	3 893	6 217	9 273
AuAg	108	121	3 275	5 638	8 970	13 379
TiO_2	384	81	7 094	12 293	19 553	29 144

6 | EXPERIMENTAL RESULTS

We turn now our attention to experimental results. We compare the performance of the *original* multi-core (CPU only) HSDLA algorithm[2] against our *refined* CPU implementation from Section 5.1 and the two hybrid CPU-GPU implementations based on that refined algorithm. As mentioned in Section 5, these include an implementation that offloads large matrix-matrix products to the multi-GPU using *CUBLASXT*, an implementation using *static* work assignment (Section 5.3), and an implementation using *dynamic* work assignment (Section 5.4). Since the implementation using dynamic work assignment supports offloading to Intel Xeon Phi accelerators, we also present results on that platform.

As test cases, we use three input systems that describe distinct physical systems. We refer to them as NaCl, AuAg, and TiO_2, respectively. These systems represent a heterogeneous sample (including both insulators and conductors) with different physical properties. In our tests, the code generates the matrices H and S for one single $\mathbf{k}$-point, and different K_{max} values. The actual problem sizes, ie, the values for N_A, N_L, and N_G for each case are given in Table 1.

6.1 | Experimental setup

We ran our experiments on a range of compute nodes. We used two computed nodes hosted by the IT center of the RWTH Aachen University. One of these nodes (RWTH-GPU) consists of two eight-core Sandy Bridge E5-2680 processors, running at a frequency of 2.7 GHz. The node is equipped with 64 GBs of RAM and 2 Nvidia Tesla K20Xm GPUs. The peak performance of the 16 CPU cores in double precision is 345 GFlops/s, while the peak performance of each GPU is 1.3 TFlops/s, for a combined peak of almost 3.0 TFlops/s. The second RWTH node (RWTH-Phi) consists of two eight-core Sandy Bridge E5-2650 processors, running at a frequency of 2.0 GHz. The node is also equipped with 64 GBs of RAM and 2 Intel Xeon Phi 5110p accelerators (Knight's Corner), The peak performance of the 16 CPU cores in double precision is 256 GFlops/s, while the peak performance of each Xeon Phi is (about) 1 TFlops/s, for a combined peak of 2.3 TFlops/s.

We also ran experiments on the JURECA supercomputer at the Jülich Supercomputing Centre (JSC). More specifically, we used one JURECA node consisting of two twelve-core Haswell E5-2680v3 processors, running at a nominal frequency of 2.5 GHz, and 2 NVIDIA K80 GPUs (each of which consists of two GK210 devices). The node is equipped with 128 GBs of RAM. The combined peak performance for the 24 CPU cores in double precision is 960 GFlops/s, while the theoretical peak performance in double precision of each GPU device is about 1.45 TFlops/s, for a total of 6.7 TFlops/s.

In the following sections, when presenting efficiency of the achieved system with a varying number of accelerators, we take as reference value the peak performance corresponding to the same number of accelerators. For example, when presenting the results on JURECA with 1, 2 and 3 GPUs, the system peak performance is 2.4, 3.8, and 5.3 TFlops, respectively.

We compare several different improved HSDLA versions. We refer to the base HSDLA algorithm, as implemented in the work of Di Napoli et al,[2] with *original*. Our algorithmically refined multiCPU HSDLA version is denoted as *refined*, while the same version but for multi-GPU system is named *cuBlasXt*. The accelerated versions, denoted as *static* and *dynamic*, are the version described in Sections 5.3 and 5.4, respectively. In all cases, the code was linked to Intel MKL version 11.3.2 for the BLAS routines on the CPU and Xeon Phis; the GPU code was linked to NVIDIA CUBLASXT version 7.5 and 8.0 on the RWTH-GPU nodes and on the JURECA nodes, respectively.

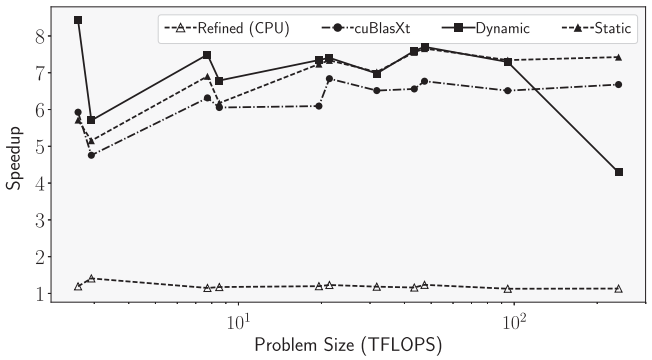

FIGURE 3 Speedup on RWTH-GPU for all implementations, relative to original code

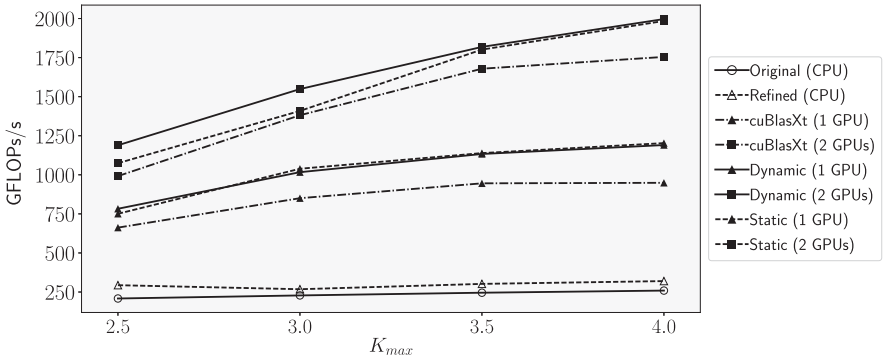

FIGURE 4 Performance on RWTH-GPU for all implementations, for the AuAg test case

TABLE 2 GFlops/s and efficiency (in parentheses) for the AuAg test case and $K_{max} = 4.0$. Results for the RWTH node using 2 GPUs

Implementation	zherk	zher2k	zherkx/zgemmt
CPU	358 (11.93%)	304 (10.13%)	343 (11.43%)
cuBlasXt	2041 (68.30%)	2012 (67.06%)	2021 (67.07%)
Dynamic	2332 (77.73%)	2367 (78.90%)	2302 (76.73%)
Static	2337 (77.91%)	2315 (77.16%)	2315 (77.16%)

6.2 | RWTH-GPU

We provide two experiments for the RWTH-GPU system. Figure 3 presents the speedup over the original HSDLA algorithm attained by the refined, cuBlasXt, dynamic, and static implementations, for each of the different test cases (showed on x-axis) sorted by the increasing number of floating point operations (in TFLOPS) required by each of them. The results show an average speedup of 1.13 of the refined algorithm over the original, when run on the CPU only. That clearly demonstrates how small changes in the code, such as exploiting symmetries and reordering of the execution flow (requiring limited programming effort), can lead to higher performance, even without using the accelerators.

In addition, the figure illustrates how the GPU-based codes (cuBlasXt, static, and dynamic) achieve seizable speedups no matter how large the the problem size is. The speedup obtained by using only the GPUs is up to 7*X* for larger problems (the theoretical peak performance of both GPUs is 7.5*X* higher than those of CPUs). Furthermore, in most cases, our custom hybrid implementations outperform the cuBlasXT, thanks to both better tuning of block sizes and especially a heavier usage of the host CPU computational power by offloading a part of the cuBlasXT computation to the CPUs. The dip at the end of the dynamic code is due to TiO_2 case run out of memory when 2 GPUs are used; instead result using 1 GPU is presented.

Figure 4 showcases the total performance achieved by the different implementations for the AuAg test case and a range of values for K_{max}. Again, the advantages of enabling the use of the entire compute nodes (CPU + GPUs) become apparent. There is a clear separation between CPU-only and accelerated code, and also between GPU-only cuBlasXt and the hybrid static and dynamic codes. Both our hybrid codes outperform GPU-only (cuBlasXt), for large enough problems, by more than 200 Gflops, which is very close to the peak performance of the CPU-only version, Table 2. This clearly demonstrates that the all CPUs are fully utilized in our hybrid approach.

Furthermore, for large problems that allow for a reasonable utilization of the multiple GPUs, we achieve about 2 TFlops/s compared to a peak performance of 3 TFlops/s, ie, 66% efficiency. The efficiency is even higher for larger problems such as TiO_2 with $K_{max} = 4.0$, where our code attains about 2.5 TFlops/s (83% of the peak).

The presented results include timings from the CPU-only parts of the code (Algorithm 3 lines 6 to 11 and 14 to 18), during which the GPUs are in idle state, thus decreasing total performance and efficiency. The efficiency attained by our hybrid BLAS routines is even higher, as presented in Table 2. The attained efficiency of the BLAS-only kernels is up to 79% for AuAg case and slightly better for TiO_2 case, up to 81% of the peak system performance.

6.3 | RWTH-Phi

For the RWTH-Phi system, we again present a speedup plot (Figure 5) and a performance plot (Figure 6). In this case, we present results for the refined algorithm and the dynamic version of the hybrid code, which is the only one with current support for the Xeon Phi (through the hStreams API). While MKL can offload calculations automatically to the device, this is only true for the GEMM operation. Therefore, the automatic offload is not applicable to our algorithm, which is dominated by the `zherk`, `zher2k`, and `zherkx` routines.

Figure 5 shows how our hybrid code consistently achieves a speedup between 4 and 5 with respect to the original HSDLA algorithm. This is consistent with both the relative computational power of the two accelerator cards and the CPU, as well as the expectation that the Phi accelerators do not get as close to their theoretical peak performance as GPUs do.

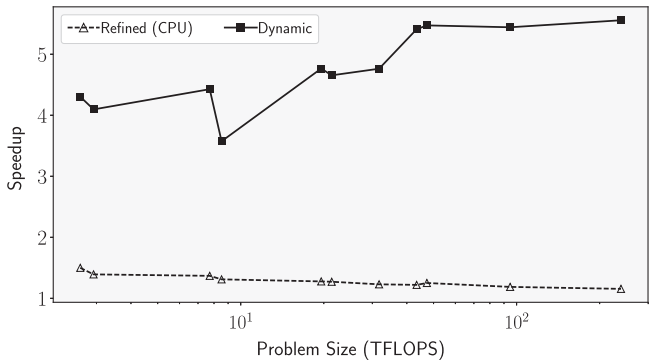

FIGURE 5 Speedup on RWTH-Phi for the refined and the hybrid dynamic implementations, relative to original code

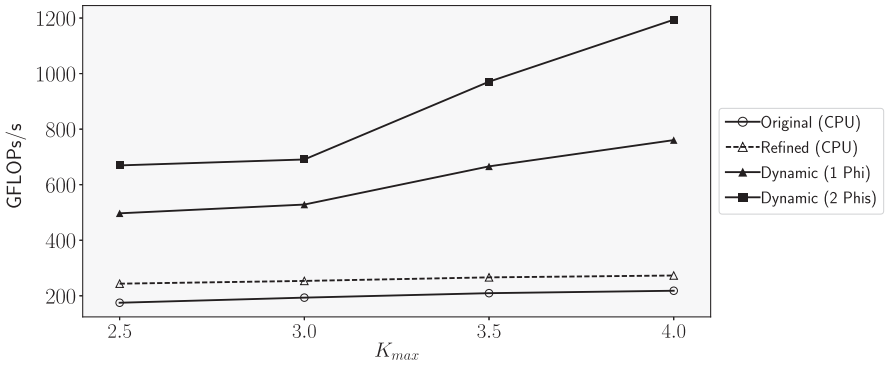

FIGURE 6 Performance on RWTH-Phi for the original, refined, and dynamic implementations, for the AuAg test case

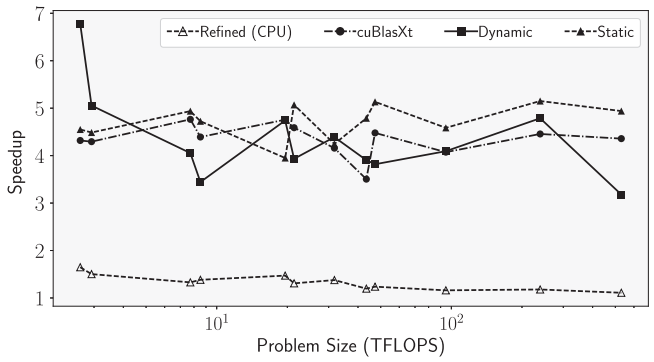

FIGURE 7 Speedup on JURECA for all implementations, relative to original code

In Figure 6, we illustrate the performance of the original, refined, and dynamic codes for AuAg test case. For the larger case ($K_{max} = 4.0$), the dynamic versions attain about 700 GFlops/s, using one Phi, and 1200 GFlops/s, using two Phis, which correspond to 55% and 52% of the system peak performance, respectively. For the TiO_2 test case, we observed even higher performances of up to 950 GFlops/s (one Phi) and 1600 GFlops/s (two Phis) corresponding to an efficiency of 75% and 69%.

Similar to the previous experiments in the RWTH-GPU system, the larger the problem size, the larger the efficiency attained, which shows that even higher efficiency is to be expected when larger systems are simulated.

6.4 | JURECA-GPU

The JURECA-GPU nodes comprise two NVIDIA K80 GPUs, each of which consists of two devices. This allows us not only to not evaluate speedups (Figure 7) and performance (Figure 8), but also scalability of the different implementations (Figure 9). Figure 7 illustrates the speedup of the various GPU implementations relative to the original HSDLA. The speedup ranges most of the times between 3.5 and 5.5 (starting from a maximum of about 7). The fact that the speedup of the dynamic code varies widely indicates that the scheduler struggles to fully utilize all four GPUs. With regards to the comparison among the three GPU-hybrid implementations, there is again a clear separation between the static code and the cuBlasXt implementation.

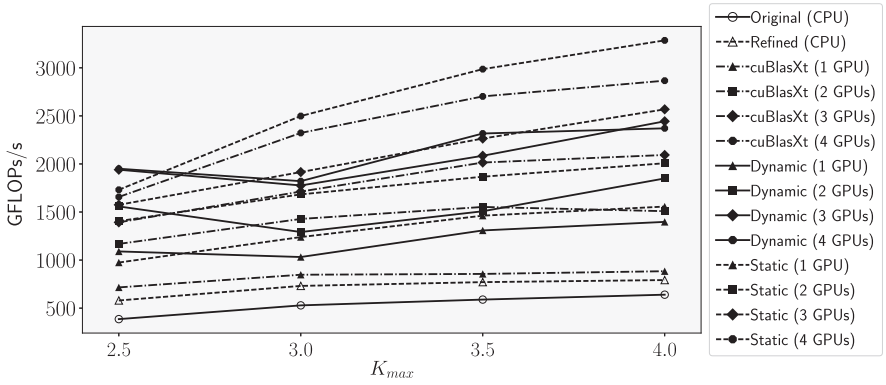

FIGURE 8 Performance on JURECA for all implementations, for the AuAg test case

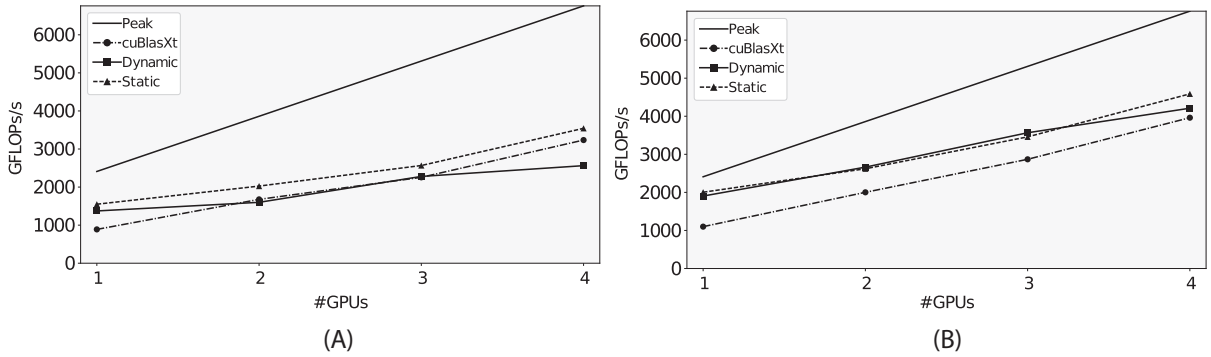

FIGURE 9 Scalability of the hybrid BLAS routines on JURECA going from one to many GPUs, for both codes, for the AuAg (A) and TiO_2 (B) 3.5 test case

The performance plot in Figure 8 illustrates the flop rates achieved by each implementation. The clear separation between the static and the cuBlasXt implementation is again very pronounced. However, as observed in Figure 7, the dynamic implementation struggles to utilize many GPUs. It seems like further improvements would be necessary to properly adapt it to this use case. Overall, our static implementation achieves up to 3.2 TFlops/s for the AuAg test case, ie,, a 15% higher performance than the cuBlasXt implementation. Similar results are achieved for the TiO_2 case, in which we observe a performance of up to 4.3 TFlops/s on 4 GPUs, resulting in 16% higher performance compared to the cuBlasXt implementation. As the system has a peak performance of 6.7 TFlops/s (4 GPU included), the results for AuAg and TiO_2 cases correspond to 47% and 61% peak performance utilization, respectively. These much lower results compared to RWTH-GPU are the result of the surprisingly low performance of the CUBLASXT implementations (only 55% utilization) of the key BLAS-3 kernels, on which our accelerated HSDLA versions depend on.

The plot in Figure 9 captures the scalability behavior of our code for the AuAg and TiO_2, $K_{max} = 3.5$ case, for 1, 2, 3, and 4 GPUs. In this plot, we focus solely on our hybrid implementation of the off-loaded BLAS routines, and consider only this part of the runtime. We extract three important messages from the graph. First, the measurements for static and dynamic code nearly coincide. This is not surprising, since both codes aim to achieve the same goal, ie, hybrid execution. Second, the measurements for all three implementations roughly share the same slope. This indicates that all codes utilize the GPUs similarly well as additional devices are added. Third, there is a gap between the hybrid codes and the GPU-only code of almost 1 TFlop/s, which is roughly the computational power of the CPU. This reflects that not only the hybrid codes reach a utilization of the GPUs similar to the vendor optimized CUBLASXT, but they also fully utilize the CPU cores. This is particularly of interest because modern CPUs provide high computational power (eg, 960 GFlops on JURECA per node), even when compared to that of the coprocessors, and thus should not be neglected nor underutilized. Table 3 shows that the performance gain of combining CPUs with GPUs in the BLAS-3 kernels in the HSDLA algorithm can improve performance up to 19% compared to the state-of-the-art cuBlasXt implementations using 4 GPUs.

TABLE 3 GFlops/s and speedup (in parentheses) for *static* and *dynamic* implementations compared to *cuBlasXt* for the TiO_2 test case and $K_{max} = 3.5$ on JURECA using 4 GPUs

Implementation	zherk	zher2k	zherkx/zgemmt
cuBlasXt	3439 (5.72×)	3258 (6.64×)	3257 (5.89×)
Dynamic	3862 (1.12×)	3590 (1.11×)	3266 (1.01×)
Static	4012 (1.19×)	3832 (1.19×)	3686 (1.13×)

7 | CONCLUSIONS AND FUTURE WORK

In this paper, we have demonstrated how legacy codes, such as the FLEUR code, can be modernized in order to exploit the massive parallelism of modern computing architectures, and to improve code performance as well as portability and scalability. The starting point of our work was the HSDLA algorithm. HSDLA encodes the computation of the Hamiltonian and the Overlap matrices (*H* and *S*), one of the two computational bottlenecks of the FLEUR code, in terms of multi-dimensional operations that map well onto BLAS-3 operations. However, the straightforward porting of these BLAS operations to heterogeneous architectures consisting of multi-core CPU and one or more GPUs or Phi accelerators does not attain expected performances. The reason behind the limited performance is that vendor optimized BLAS libraries do not provide support for such hybrid architectures and exploit either the CPUs or the accelerators, but not both.

We used the original BLAS-based HSDLA code as a starting point to explore both existing and custom implementations in order to take advantage of the accelerators. First, we improved the original HSDLA algorithm by introducing changes in the execution flow; the result was a much lower memory footprint and a reduced computational cost. Then, we showed that custom hybrid implementations can boost performance compared to accelerator-only implementations, especially when CPUs themselves provide large computational power.

We presented two approaches to implement hybrid BLAS, ie, a dynamic approach that schedules chunks of calculation on the devices using buffers and queues, and a static approach that splits matrices based on prescribed ratios. The dynamic approach is more generic, as demonstrated by targeting both GPUs and Phis. Because the static code needs to be tuned through the ratios, it gives better control for tuning. Both implementation strategies considerably boost performance due to their hybrid nature.

However, there is still room for improvement. By using mathematical equalities, the objects *A* and *B* can be compressed into smaller objects, leading to less but more complex computations. More specifically, the compressed form of *A* and *B* leads to tensor contractions that do not naturally map onto `gemm`-like operations and require the development of specialized routines. If efficient mappings and implementations of new kernels are found, speedups of up to one order of magnitude may be achieved. Furthermore, in order to solve even larger problems (systems of up to 1000 atoms), the presented approaches present certain limitations such as the memory requirements and the limited size of the on-device memory. To efficiently solve larger problems, new memory and communication-avoiding approaches on distributed multi-CPU and multi-GPU systems are required, including the redesign of the most time-consuming kernels.

ACKNOWLEDGMENTS

This work was partially funded by the Ministry of Science and Education of the Republic of Croatia and the Deutsche Akademische Austauschdienst (DAAD) from funds of the Bundesministeriums für Bildung und Forschung (BMBF) through project "PPP Kroatien" ID 57216700. Financial support from the Jülich Aachen Research Alliance-High Performance Computing and the Deutsche Forschungsgemeinschaft (DFG) through grant GSC 111 is also gratefully acknowledged. Furthermore, the authors thank the RWTH IT Center and the Jülich Supercomputing Centre for the computational resources.

ORCID

Davor Davidović http://orcid.org/0000-0003-2649-9236
Edoardo Di Napoli http://orcid.org/0000-0001-5821-5897

REFERENCES

1. FLEUR: The Jülich FLAPW code family. http://www.flapw.de/pm/index.php. Accessed December 10, 2017.
2. Di Napoli E, Peise E, Hrywniak M, Bientinesi P. High-performance generation of the Hamiltonian and overlap matrices in FLAPW methods. *Comput Phys Commun*. 2017;211:61-72. https://doi.org/10.1016/j.cpc.2016.10.003
3. Deserno M, Holm C. How to mesh up Ewald sums. I. A theoretical and numerical comparison of various particle mesh routines. *J Chem Phys*. 1998;109(18):7678-7693.
4. Johnson J, Douze M, Jégou H. Billion-scale similarity search with GPUs. Http://arxiv.org/abs/1702.08734. Accessed February 2017.
5. Fabregat-Traver D, Bientinesi P. Computing petaflops over terabytes of data: the case of genome-wide association Studies. *ACM Trans Math Softw*. 2014;40(4):27.
6. Intel Math Kernel Library. https://software.intel.com/en-us/mkl. Accessed December 10, 2017.
7. Tomov S, Dongarra J, Baboulin M. Towards dense linear algebra for hybrid GPU accelerated manycore systems. *Parallel Comput*. 2010;36(5-6):232-240.
8. NVIDIA cuBLAS. Dense Linear Algebra on GPUs. https://developer.nvidia.com/cublas. Accessed December 12, 2017.
9. Auckenthaler T, Blum V, Bungartz HJ, et al. Parallel solution of partial symmetric eigenvalue problems from electronic structure calculations. *Parallel Comput*. 2011;37(12):783-794.
10. Marek A, Blum V, Johanni R, et al. The ELPA library: scalable parallel eigenvalue solutions for electronic structure theory and computational science. *J Phys: Condens Matter*. 2014;26(21):213201.
11. Fabregat-Traver D, Davidović D, Höhnerbach M, Di Napoli E. Hybrid CPU-GPU generation of the Hamiltonian and overlap matrices in FLAPW methods. In: Di Napoli E, Hermanns MA, Iliev H, Lintermann A, Peyser A, eds. *High-Performance Scientific Computing*. Cham, Switzerland: Springer; 2016:200-211. *Lecture Notes in Computer Science*; Vol. 10164.

12. Fiolhais C, Nogueira F, Marques MAL. *A Primer in Density Functional Theory*. Berlin, Germany: Springer; 2003. *Lecture Notes in Physics*.
13. Sholl D, Steckel JA. *Density Functional Theory: A Practical Introduction*. Hoboken, NJ: John Wiley & Sons; 2011.
14. Kohn W, Sham LJ. Self-consistent equations including exchange and correlation effects. *Phys Rev*. 1965;140:A1133-A1138.
15. Wimmer E, Krakauer H, Weinert M, Freeman AJ. Full-potential self-consistent linearized-augmented-plane-wave method for calculating the electronic-structure of molecules and surfaces: O2 molecule. *Phys Rev B*. 1981;24(2):864-875.
16. Jansen HJF, Freeman AJ. Total-energy full-potential linearized augmented-plane-wave method for bulk solids - electronic and structural-properties of tungsten. *Phys Rev B*. 1984;30(2):561-569.
17. Wang L, Wu W, Xu Z, Xiao J, Yang Y. Blasx: a high performance level-3 BLAS library for heterogeneous multi-GPU computing. In: Proceedings of the 2016 International Conference on Supercomputing (ICS'16); 2016; Istanbul, Turkey.
18. Blas JG, Dolz MF, Garcia JD, et al. Porting MATLAB applications to high-performance C++ codes: CPU/GPU-accelerated spherical deconvolution of diffusion MRI data. Paper presented at: International Conference on Algorithms and Architectures for Parallel Processing; 2016; Granada, Spain.
19. Haidar A, Cao C, Yarkhan A, et al. Unified development for mixed multi-GPU and multi-coprocessor environments using a lightweight runtime environment. In: Proceedings of the IEEE 28th International Parallel and Distributed Processing Symposium (IPDPS'14) 2014; Phoenix, AZ.
20. Solcà R, Kozhevnikov A, Haidar A, Tomov S, Dongarra J, Schulthess TC. Efficient implementation of quantum materials simulations on distributed CPU-GPU systems. In: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis; 2015; Austin, TX.

How to cite this article: Davidović D, Fabregat-Traver D, Höhnerbach M, Di Napoli E. Accelerating the computation of FLAPW methods on heterogeneous architectures. *Concurrency Computat Pract Exper*. 2018;30:e4905. https://doi.org/10.1002/cpe.4905