A Systematic Approach to Improving Data Locality Across Fourier Transforms and Linear Algebra Operations

Doru Thom Popovici, Andrew Canning, Zhengji Zhao, Lin-Wang Wang, John Shalf

Lawrence Berkeley National Lab

USA

{dtpopovici,acanning,zzhao,lwwang,jshalf}@lbl.gov

ABSTRACT

The performance of most scientific applications depends on efficient mathematical libraries. For example, scientific applications like the plane wave based Density Functional Theory approach for electronic structure calculations uses highly optimized libraries for Fourier transforms, dense linear algebra (orthogonalization) and sparse linear algebra (non-local projectors in real space). Although vendor-tuned libraries offer efficient implementations for each standalone mathematical kernel, the partitioning of those calls into sequentially invoked kernels inhibits cross-kernel optimizations that could improve data locality across memory bound operations. In this work we show that, by expressing these kernels as an operation on high dimensional tensors, cross-kernel dataflow optimizations that span FFT, dense and sparse linear algebra, can be readily exposed and exploited. We outline a systematic way of merging the Fourier transforms with the linear algebra computations, improving data locality and reducing data movement to main memory. We show that compared to conventional implementations, this streaming/dataflow approach offers 2x speedup on GPUs and 8x/12x speedup on CPUs compared to a baseline code that uses vendor-optimized libraries. Although we use Density Functional Theory to demonstrate the value of our approach, our methodology is broadly applicable to other applications that use Fourier transforms and linear algebra operations as building blocks.

CCS CONCEPTS

• **Mathematics of computing** → **Mathematical software performance**; • **Computing methodologies** → **Shared memory algorithms**; **Massively parallel algorithms**.

KEYWORDS

Fourier Transforms, Linear Algebra Operations, High Dimensional Tensors, Loop Optimizations

ACM Reference Format:

Doru Thom Popovici, Andrew Canning, Zhengji Zhao, Lin-Wang Wang, John Shalf. 2021. A Systematic Approach to Improving Data Locality Across Fourier Transforms and Linear Algebra Operations. In *2021 International Conference on Supercomputing (ICS '21)*, June 14–17, 2021, Virtual Event, USA. ACM, New York, NY, USA, 13 pages. <https://doi.org/10.1145/3447818.3460354>

This work is licensed under a Creative Commons Attribution International 4.0 License.

ICS '21, June 14–17, 2021, Virtual Event, USA

© 2021 Copyright held by the owner/author(s).

ACM ISBN 978-1-4503-8335-6/21/06.

<https://doi.org/10.1145/3447818.3460354>

1 INTRODUCTION

Many scientific applications in fields like physics, chemistry, materials science or even machine learning, depend on efficient and high-performance mathematical libraries. One such application is Density Functional Theory, which is widely used in materials science and chemistry to simulate the electronic structure of materials. Density Functional Theory codes account for approximately 25% [3, 12] of the total compute cycles consumed at large scientific supercomputing centers such as the National Energy Research and Scientific Computing Center (NERSC) at the Lawrence Berkeley National Laboratory. Among the most widely deployed implementations of the Density Functional Theory method is the plane wave pseudopotential implementation, which uses a plane wave basis set to represent the electron wavefunctions. This approach requires a mix of operations that span multiple vendor-optimized libraries. For example, it requires Fourier transformations to perform basis changes between frequency and real space representation of the wavefunctions, it uses dense linear algebra operations to compute projections and orthogonalizations, and sparse linear algebra computations for real space projections of the nonlocal pseudopotentials. Currently, most of these steps are implemented as sequential function calls to libraries like FFTW [15], MKL [1], AMD-FFTW [2], BLAS [11, 26, 37], cuFFT [24] and cuBLAS [25].

The aforementioned libraries provide stable interfaces to the various hand-coded implementations of mathematical kernels like Fourier transforms, linear algebra operations. In most cases, these libraries offer efficient and high-performance implementations for each of these computations. However, simply invoking a sequence of optimized library calls does not always translate to efficient utilization of the underlying hardware. Performance can still be improved if optimizations are applied across library calls in a dataflow execution pattern. This becomes more relevant when the sequence of computations are well known memory bound problems like Fourier transforms, point-wise operations, some linear algebra operations, as seen in one of the components required in the plane wave-based Density Functional Theory codes. For such components it becomes important to focus on improving data locality across the computation steps, rather than to just optimize the computation in isolation. However, applying optimizations across compute stages requires one to break the library barriers, and expose and express computation using a common high level representation. We are taking the first steps towards a common mathematical language to express a systematic way of applying optimizations across Fourier transforms and linear algebra operations.

In this work, we propose a systematic way to expose computation and data movement between multiple components, and to apply optimizations to improve data locality and reduce roundtrips to main

memory. We will use the plane wave Density Functional Theory application as one example to illustrate our procedure. More specifically, we will merge the Fourier computation with linear algebra operations in the critical application of the Hamiltonian operator to the wave function step. We use the idea proposed in Popovici *et al* [29], and cast the Fourier computation as a multi-linear algebra operation. Using this notation, we provide a step-by-step guide to outline the different ways of fusing the Fourier computation and the linear algebra steps, showing the trade-offs between on chip storage requirement and overall execution time. We provide both CPU and GPU implementations, and show that our approach can achieve significant improvements compared to conventional approaches that use vendor-tuned numerical libraries. While the optimization steps are specific to the Density Functional Theory codes, we argue that the mathematical notation of representing the Fourier computation and the techniques of merging the Fourier computation with linear algebra can be applied to other applications that use similar building blocks such as [21, 38, 39].

Contributions. The work makes the following contributions:

- We extend a common mathematical notation to allow for optimizations between Fourier transforms and multi-linear algebra operations.
- We show a systematic way of applying optimizations across the Fourier computation and sparse tensor contractions in the planewave Density Functional Theory codes.
- We provide a CPU and GPU implementation of our approach.
- We show performance improvements of our dataflow approach against conventional library-based implementations.

2 BACKGROUND

In this section, we give a brief overview of the plane wave Density Functional Theory codes. We outline the components that may benefit from the optimizations tackled in this paper. We then describe the Fourier computation using the multi-linear algebra notation proposed in [29].

2.1 Density Functional Theory

First-principles Density Functional Theory calculations in the Kohn-Sham formalism are widely used to study the electronic structure of materials. The Kohn-Sham single particle equation is expressed as

$$\hat{H}\psi_i(r) = \left[-\frac{1}{2}\nabla^2 + V(r) + \sum_l |\phi_l \rangle \langle \phi_l| \right] \psi_i(r) = \epsilon_i \psi_i(r), \quad (1)$$

where $\psi_i(r)$ is the i -th electron orbital (wavefunction) in the system and ϵ_i represents the electron orbital's eigen energy. Each wavefunction spans the entire 3D real space r . The first term in the Hamiltonian operator $\hat{H}$ is the kinetic energy operator $-\frac{1}{2}\nabla^2$. The second term $V(r)$ represents the local potential. In the local density approximation (LDA) using pseudopotentials to represent the effects of nuclei and core electrons, the V term is expressed as

$$V = \sum_{R_I} v_{ion}(r - R_I) + \int \frac{\rho(r')}{|r - r'|} d^3r' + \mu_{xc}(\rho(r)), \quad (2)$$

where $v_{ion}(r - R_I)$ represents the ionic pseudopotential of ion I , R_I is the position of the ion, $\rho(r) = \sum_i \psi_i(r)\psi_i^*(r)$ is the total

Figure 1: Schematic depiction of the self consistent field (SCF) method implemented in electronic structure codes.

charge density, and $\mu_{xc}(\rho(r))$ is the LDA exchange correlation potential. The last term $\sum_l |\phi_l \rangle \langle \phi_l|$ is the non-local pseudopotential projector, where the index l represents the projectors. l can take values between 9 and 18, and it depends on the atomic type. In real space, the projector representation $\phi_l(r)$ only exists within a R_c sphere surrounding each atom, and the operation of this term can be represented more explicitly as: $[\sum_l |\phi_l \rangle \langle \phi_l|] \psi_i(r) = \sum_l \phi_l(r) \int \phi_l^*(r') \psi_i(r') d^3r'$. The operation can be viewed as a projection of the wavefunctions on the non-local pseudopotentials. The solution to the Kohn-Sham equation gives the wavefunction of each orbital. Since the potential V depends on the total charge density $\rho(r)$, which in turn depends on the wavefunctions $\psi_i(r)$, the Kohn-Sham equation is non-linear.

Solving the non-linear problem is typically done using the self-consistent field (SCF) method, shown in Figure 1. The method starts with an initial wavefunction guess, from which the charge density and the Hamiltonian are computed. A new set of wavefunctions is determined by solving Equation 1, which in turn are used to compute a new charge density. The steps are repeated until convergence. Solving Equation 1 requires the wavefunctions to be discretized on a real space grid or expressed in some basis set. One of the most widely used basis sets is the plane wave basis set [8]. For the simple case of a gamma point calculation, the discretization of the $\psi_i(r)$ can be expressed as a Fourier series expansion such that

$$\psi_i(r) = \sum_g c_i(g) e^{jgr}, \quad (3)$$

where g represents the reciprocal lattice vectors of the supercell to be studied and $j = \sqrt{-1}$. The $c_i(g)$ terms represent the plane wave coefficients. It is conventional to truncate the plane wave basis set within an energy cutoff E_{cut} , thus only the g vectors within a cut-off sphere are used in the above expansion as shown in Figure 5.

Equation 1 can be solved for all ψ_i wavefunctions using a Conjugate Gradient (CG) algorithm, as seen in Figure 2. The same CG

$$\begin{aligned}
h(i, j) &= \langle \psi_i | H | \psi_j \rangle \\
P_i &= H \psi_i - \epsilon_i \psi_i \\
P_i &= A \left(P_i - \frac{\lambda_i}{\lambda_i^0} P_i^0 \right) \\
P_i &= P_i - \sum_{j=1, i} \langle P_i | \psi_j \rangle \psi_j \\
\psi_i &= \psi_i \cos \theta_i + P_i \sin \theta_i \\
\psi_i &= \psi_i - \sum_{j < i} \langle \psi_i | \psi_j \rangle \psi_j \\
h(i, j) &= \langle \psi_i | H | \psi_j \rangle
\end{aligned}$$

Figure 2: The Conjugate Gradient algorithm used to solve Equation 1 for ψ_i . Because the same algorithm is applied on all the wavefunctions, the algorithm is batched.

steps are applied for all the wavefunctions. Therefore, the wavefunctions can be batched together such that

$$\Psi = [\psi_0 \mid \psi_1 \mid \dots \mid \psi_{N_b-1}], \quad (4)$$

where N_b represents the number of wavefunctions or the number of bands. The all-band algorithm converts most of the computations from matrix vector multiplications to matrix matrix multiplications. However, the $H\Psi$ is never computed as an actual matrix-matrix multiplication, because storing the full matrix H requires prohibitively large amounts of memory. In reality, the $H\Psi$ computation is decomposed following Equation 1 and Equation 2, which splits the computation into point-wise operations, Fourier transforms and projections onto the non-local pseudopotential projectors. Since the projectors in real space exist within a small sphere surrounding each atom, the projection on the non-local projectors $\phi_l(r)$ is performed in real space. Doing the computation in frequency space would require more computation and more storage.

In a nutshell, the expensive matrix computation is replaced with memory bound operations, more specifically batched memory bound operations. Since the computation of this variant of the $H\Psi$ can take from 20 to 40% of the total execution time for given systems [7], in this work we focus on optimizing across the pointwise, Fourier and projector related operations. To expose the opportunities for optimization, in the following paragraphs we briefly present the Fourier transform using a multi-linear algebraic notation. We will use this notation to outline the systematic approach of optimizing the Fourier transform with the surrounding computation, as emphasized in the thesis work by Popovici [30].

2.2 The Fourier Transform

Any multi-dimensional Fourier transform is a linear transform that can be expressed as a matrix operation. For example, the 1D discrete Fourier transform (we use the DFT abbreviation for the discrete Fourier transform) is a matrix-vector multiplication, where given

Figure 3: Decomposition of the 1D DFT of size 16 using the Cooley-Tukey algorithm (data stored as column major). The algorithm requires four steps, two batched DFTs (a) and (d) of size 4, a point-wise operation (b) and a transposition (c).

the input x of size n , the output y is obtained as

$$y = DFT_n \cdot x. \quad (5)$$

The DFT_n is the $n \times n$ Fourier dense matrix, defined as

$$DFT_n = [\omega_n^{lk}]_{0 \leq l, k < n}, \quad (6)$$

with $\omega_n = e^{-j \frac{2\pi}{n}}$ being the complex root of unity. Typically, the computation of the DFT is implemented using the Fast Fourier Transform (FFT), where instead of performing $O(n^2)$ complex arithmetic operations by doing the matrix-vector multiplication, a recursive decomposition of the DFT matrix is performed to obtain an $O(n \log(n))$ algorithm. The most widely known of these algorithms is the Cooley-Tukey algorithm [9], which can be described concisely as a factorization of the DFT_n matrix for $n = n_0 \times n_1$.

In linear algebra notation, the Cooley-Tukey algorithm for a problem size $n = n_0 \times n_1$ is expressed as:

$$\tilde{y} = (Twid_{n_0 \times n_1} \odot (\tilde{x} \cdot DFT_{n_1}))^T \cdot DFT_{n_0}, \quad (7)$$

where $\tilde{x}$ and $\tilde{y}$ are the two-dimensional matrix representations of size $n_0 \times n_1$ and $n_1 \times n_0$ of the one dimensional input x and output y , respectively. The columns of matrix $\tilde{x}$ are n_1 contiguous sub-vectors (i.e. x_i , $0 \leq i < n_0$) of the input vector x , and are obtained by splitting x into sub-vectors of size n_0 as follows

$$\tilde{x} = [x_0 \mid x_1 \mid \dots \mid x_{n_1-1}].$$

The matrix $\tilde{y}$ is similarly constructed.

Equation 7 clearly describes the steps of the Cooley Tukey algorithm. The first step applies the DFT_{n_1} on the rows of the matrix $\tilde{x}$ (I), assuming data is stored in column major order. The elements in each column of the $\tilde{x}$ matrix are in consecutive locations in memory. This implies that in the first DFT stage, elements are accessed at a stride of n_0 . The result of the first stage is then point-wise multiplied ($\odot$) with the twiddle matrix (II), defined as

$$Twid_{n_0 \times n_1} = [\omega_n^{kl}]_{0 \leq l < n_0 \text{ and } 0 \leq k < n_1}, \quad (8)$$

where ω_n represents the complex roots of unity. The result is then transposed (III) and the second DFT of size n_0 is applied in the rows (IV). The last DFT stage requires data at a stride of n_1 , and because of the transposition it cannot proceed until all previous stages are complete. Pictorially, the four stages of the Cooley Tukey algorithm can be seen in Figure 3.

The notation can be extended to multi-dimensional DFTs. For example, a 3D DFT of size $n_0 \times n_1 \times n_2$ is expressed as

$$y = DFT_{n_0 \times n_1 \times n_2} \cdot x, \quad (9)$$

Figure 4: Algorithms for computing the 3D DFT: (a) the slab-pencil algorithm decomposes the 3D DFT into 2D and 1D DFTs. (b) the pencil-pencil-pencil algorithm decomposes the DFT into three 1D DFTs in the corresponding dimensions.

where the $DFT_{n_0 \times n_1 \times n_2}$ represents the complex DFT matrix. Any multi-dimensional DFT is expressed in terms of multiple 1D DFTs and/or multi-dimensional DFTs of lower dimensions. For example, Figure 4 shows 1. the slab-pencil algorithm, that decomposes the 3D DFT into a batch of 2D DFTs followed by a batch of multiple 1D DFTs, and 2. the pencil-pencil-pencil algorithm, that splits the 3D DFT into three batches of 1D DFTs, where each 1D DFT is applied in the corresponding three dimensions.

The slab-pencil algorithm sees the input (output) column vectors x (y) as 2D matrices $\tilde{x}$ ($\tilde{y}$) of size $(n_0 \times n_1) \times n_2$. Mathematically the decomposition is expressed as

$$\tilde{y} = (DFT_{n_0 \times n_1} \cdot \tilde{x}) \cdot DFT_{n_2} \quad (10)$$

Data is stored in column major, hence the 2D DFT is applied on the columns, while the 1D DFT is applied on the rows. The order of the operations can be permuted, such that the 1D DFT is applied first, followed by the 2D DFT.

The pencil-pencil-pencil algorithm views the input (output) vectors as 3D cubes $\hat{x}$ ($\hat{y}$) of size $n_0 \times n_1 \times n_2$. The input (output) column vector x (y) is decomposed into n_2 groups of n_1 subgroups of size n_0 . Hence, the 3D cube $\hat{x}$ is viewed as a matrix of matrices such as

$$\hat{x} = [\tilde{x}_0 \mid \tilde{x}_1 \mid \dots \mid \tilde{x}_{n_2-1}], \quad (11)$$

where each $\tilde{x}_i$ is the 2D matrix of size $n_0 \times n_1$ for all values $0 \leq i < n_2$. Mathematically, the pencil-pencil-pencil algorithm is expressed as

$$\tilde{t}_i = (DFT_{n_0} \cdot \tilde{x}_i) \cdot DFT_{n_1}, \forall 0 \leq i < n_2 \quad (12)$$

$$\hat{y} = [\tilde{t}_0 \mid \tilde{t}_1 \mid \dots \mid \tilde{t}_{n_2-1}] \cdot DFT_{n_2}$$

where the DFT_{n_0} is applied in the depth dimension, the DFT_{n_1} is applied in the column dimension and finally the DFT_{n_2} is applied in the row dimension. Data is stored in column major order and therefore the dimension corresponding to n_0 is laid out in the fastest dimension in memory, while the dimension corresponding to n_2 is laid out in the slowest dimension in memory. The order of applying the 1D pencils can be permuted and this is important when merging the forward and the inverse Fourier transforms. We describe these aspects in the following sections.

Algorithm 1: Computing $H\Psi$ as outlined in Equations 1 and 2.

Input:

- 1 $\Psi^{(b)}(v_0 v_1 v_2)$ - input planewaves
- 2 $\nabla^{(b)}(v_0 v_1 v_2)$ - kinetic energy
- 3 $\Phi^{(lr)}(\mu_0 \mu_1 \mu_2)$ - non-local projectors
- 4 $V(\mu_0 \mu_1 \mu_2)$ - ionic potential

Output:

- 5 $\Psi^{(b)}(v_0 v_1 v_2)$ - output planewaves

begin

- 7 $T_0^{(b)}(\mu'_0 \mu'_1 \mu'_2) = \Psi^{(b)}(v_0 v_1 v_2) \cdot P^{(v_0 v_1 v_2)}(\mu'_0 \mu'_1 \mu'_2)$
- 8 $T_1^{(b)}(\mu_0 \mu_1 \mu_2) = T_0^{(b)}(\mu'_0 \mu'_1 \mu'_2) \cdot iDFT_{3D}^{(\mu'_0 \mu'_1 \mu'_2)}(\mu_0 \mu_1 \mu_2)$
- 9 $T_2^{(b)}(lr) = T_1^{(b)}(\mu_0 \mu_1 \mu_2) \cdot \left(\Phi^{(lr)}(\mu_0 \mu_1 \mu_2) \right)^T$
- 10 $T_1^{(b)}(\mu_0 \mu_1 \mu_2) = T_1^{(b)}(\mu_0 \mu_1 \mu_2) \odot V(\mu_0 \mu_1 \mu_2)$
- 11 $T_1^{(b)}(\mu_0 \mu_1 \mu_2) + = T_2^{(b)}(lr) \cdot \Phi^{(lr)}(\mu_0 \mu_1 \mu_2)$
- 12 $T_0^{(b)}(\mu'_0 \mu'_1 \mu'_2) = T_1^{(b)}(\mu_0 \mu_1 \mu_2) \cdot DFT_{3D}^{(\mu_0 \mu_1 \mu_2)}(\mu'_0 \mu'_1 \mu'_2)$
- 13 $\Psi^{(b)}(v_0 v_1 v_2) = \Psi^{(b)}(v_0 v_1 v_2) \odot \nabla^{(b)}(v_0 v_1 v_2)$
- 14 $\Psi^{(b)}(v_0 v_1 v_2) + = T_0^{(b)}(\mu'_0 \mu'_1 \mu'_2) \cdot U^{(\mu'_0 \mu'_1 \mu'_2)}(v_0 v_1 v_2)$
- 15 **end**

3 MERGING THE FOURIER AND LINEAR ALGEBRA OPERATIONS

In this section, we first talk about the $H\Psi$ computation in more detail. We use the multi-linear algebra notation to represent both Fourier transforms and linear algebra operations. We then show the step-by-step approach to merging the computation steps.

3.1 The $H\Psi$ Computation

The Algorithm 1 outlines the steps required to compute the $H\Psi$ as outlined in Equations 1 and 2. Note that the inputs and the outputs are multi-dimensional tensors. We use capital letters for the tensor names, and lower-case letters as super-scripts to denote the tensor dimensions. For example, the planewave representation of the wavefunctions is a four dimensional tensor, denoted by $\Psi^{(b)}(v_0 v_1 v_2)$. The b dimension represents the batch or band dimension, and the v_0, v_1, v_2 dimensions represent the frequency space dimensions. The kinetic energy $\nabla^{(b)}(v_0 v_1 v_2)$, the ionic potential $V^{(\mu_0 \mu_1 \mu_2)}$ are three dimensional tensors. The non-local pseudopotential projector $\Phi^{(lr)}(\mu_0 \mu_1 \mu_2)$ is a five dimensional tensor, where the l dimension corresponds to the number of projectors per atom, the r dimension corresponds to the number of atoms, and finally μ_0, μ_1 and μ_2 represent the real space dimensions. V is dense, and the other three are block-sparse tensors.

The steps of the algorithm are straightforward. First, the input Ψ is padded with zeroes in the $(v_0 v_1 v_2)$ (Line 7) using the $P^{(v_0 v_1 v_2)}(\mu'_0 \mu'_1 \mu'_2)$ operator. P contains ones and zeros, and embeds the sphere corresponding to each planewave into a larger cube, as shown in Figure 5.a. The padding operation is batched since the same P is applied on all planewaves stored in Ψ . Once padding is done, a 3D inverse DFT is applied on the newly padded dimension

Figure 5: The Fourier computation requires each wavefunction to be padded with zeroes. Version (a) does the full packing and then applies the 3D DFT on the padded cube. Version (b) splits the padding and the 3D DFT and interleaves the operations.

$(\mu'_0 \mu'_1 \mu'_2)$ (Line 8). Recall that the 3D DFT is a two-dimensional matrix and also a six-dimensional tensor, since any multi-dimensional DFT is decomposed as multiple one dimensional DFTs. The same 3D DFT computation is applied on all the planewaves in the b dimension. Therefore, the Fourier computation can be batched. Although the size of the dimension does not change, the dimensions $(\mu'_0 \mu'_1 \mu'_2)$ are changed into $(\mu_0 \mu_1 \mu_2)$ to specify the change from frequency to real space. The data stored in $T_1^{(b)}(\mu_0 \mu_1 \mu_2)$ represents the real space representation of the planewaves.

In real space, T_1 is first projected onto the non-local projectors stored in Φ (Line 9). The projection can be viewed as a matrix-matrix multiplication in the common dimension $(\mu_0 \mu_1 \mu_2)$. Note that on Line 9, the two dimensional view of the Φ tensor is transposed. This transposition does not occur, and the projection of T_1 just requires strided access into Φ . Meanwhile, the same data points in T_1 are point-wise scaled with the ionic potential V (Line 10), where the same V is used for all the data points in the b dimension. Finally, the projectors in Φ are scaled by the results in T_2 and used to update T_1 (Line 11). Similar to Line 9, the scaling of Φ with T_2 can be viewed as a matrix-matrix multiplication in the common (lr) dimension. The two matrix-matrix operations are block-sparse times dense operations. Φ is block-sparse, but T_1 and T_2 are dense. The last step of the computation converts the data in T_1 back to frequency space by applying a batched 3D DFT in the $(\mu_0 \mu_1 \mu_2)$ dimension. In frequency space, the initial Ψ tensor is point-wise scaled by the kinetic ∇ (Line 13) and then updated with the corresponding elements from T_0 (Line 14). The data in T_0 is unpacked using the same U tensor. Like P , U contains ones and zeroes, and extracts each planewave sphere from T_0 and updates the corresponding locations in Ψ .

3.2 Merging the Packing with the 3D DFTs

The padding operation and the inverse 3D DFT computation described as

$$\begin{aligned} T_0^{(b)}(\mu'_0 \mu'_1 \mu'_2) &= \Psi^{(b)}(\nu_0 \nu_1 \nu_2) \cdot P^{(\nu_0 \nu_1 \nu_2)}(\mu'_0 \mu'_1 \mu'_2) \\ T_1^{(b)}(\mu_0 \mu_1 \mu_2) &= T_0^{(b)}(\mu'_0 \mu'_1 \mu'_2) \cdot iDFT_{3D}^{(\mu'_0 \mu'_1 \mu'_2)}(\mu_0 \mu_1 \mu_2) \end{aligned} \quad (13)$$

can be merged to reduce redundant computation and data movement. As shown in Figure 5.a, the sphere is embedded into a cube, and then the 3D DFT is applied on the entire cube. The cube surrounding the sphere has a width of at least twice the size of the sphere's diameter. Embedding the sphere into the cube increases

the amount of data by almost 16 times. Because the 3D DFT computation is decomposed as one dimensional DFTs as

$$\begin{aligned} T_0^{(b)}(\mu'_0 \mu'_1 \mu'_2) &= \Psi^{(b)}(\nu_0 \nu_1 \nu_2) \cdot P^{(\nu_0 \nu_1 \nu_2)}(\mu'_0 \mu'_1 \mu'_2) \\ T_{00}^{(b)}(\mu_0 \mu'_1 \mu'_2) &= T_0^{(b)}(\mu'_0 \mu'_1 \mu'_2) \cdot iDFT_{1D}^{\mu'_0 \mu_0} \\ T_{01}^{(b)}(\mu_0 \mu_1 \mu'_2) &= T_{00}^{(b)}(\mu_0 \mu'_1 \mu'_2) \cdot iDFT_{1D}^{\mu'_1 \mu_1} \\ T_1^{(b)}(\mu_0 \mu_1 \mu_2) &= T_{01}^{(b)}(\mu_0 \mu_1 \mu'_2) \cdot iDFT_{1D}^{\mu'_2 \mu_2}, \end{aligned} \quad (14)$$

where the $iDFT_{1D}^{\mu'_0 \mu_0}$, $iDFT_{1D}^{\mu'_1 \mu_1}$ and $iDFT_{1D}^{\mu'_2 \mu_2}$ are the 1D DFTs applied in the μ'_0 , μ'_1 and μ'_2 dimensions, the first two stages move redundant amount of data back and forth main memory.

As shown in the paper by Canning [7], the redundant computation can be eliminated by grouping the padding operation with the DFT stages. As shown in Figure 5.b, the zero-padding operation can be split into three stages, that gradually pad the data with zeros in each dimension. The three padding stages and the three DFT stages can be interleaved such that

$$\begin{aligned} T_{02}^{(b)}(\mu_0 \nu_1 \nu_2) &= \Psi^{(b)}(\nu_0 \nu_1 \nu_2) \cdot (P^{\nu_0 \mu'_0} \cdot iDFT_{1D}^{\mu'_0 \mu_0}) \\ T_{03}^{(b)}(\mu_0 \mu_1 \nu_2) &= T_{02}^{(b)}(\mu_0 \nu_1 \nu_2) \cdot (P^{\nu_1 \mu'_1} \cdot iDFT_{1D}^{\mu'_1 \mu_1}) \\ T_1^{(b)}(\mu_0 \mu_1 \mu_2) &= T_{03}^{(b)}(\mu_0 \mu_1 \nu_2) \cdot (P^{\nu_2 \mu'_2} \cdot iDFT_{1D}^{\mu'_2 \mu_2}), \end{aligned} \quad (15)$$

where $P^{\nu_0 \mu'_0}$, $P^{\nu_1 \mu'_1}$, $P^{\nu_2 \mu'_2}$ represent the padding operations done in each of the dimensions. The padding and the 1D DFT computation are done one after the other, such that the data gets zero padded and then immediately Fourier transformed. The merging operation reduces the amount of data each stage must read and write, and it also reduces the amount of redundant computation. For the remainder of the paper, we refer to the merged operation as the padded DFT. Lines 12 and 14 perform the inverse operation of the one described above. Data is Fourier transformed and the corresponding spheres are selected and store into the output destination buffer. The same optimization steps can be applied there as well. More precisely, the 3D DFT and un-padding operation can be done in mirror image such that

$$\begin{aligned} T_{03}^{(b)}(\mu_0 \mu_1 \nu_2) &= T_1^{(b)}(\mu_0 \mu_1 \mu_2) \cdot (DFT_{1D}^{\mu_2 \mu'_2} \cdot U^{\mu'_2 \nu_2}) \\ T_{02}^{(b)}(\mu_0 \nu_1 \nu_2) &= T_{03}^{(b)}(\mu_0 \mu_1 \nu_2) \cdot (DFT_{1D}^{\mu_1 \mu'_1} \cdot U^{\mu'_1 \nu_1}) \\ \Psi^{(b)}(\nu_0 \nu_1 \nu_2) &= T_{02}^{(b)}(\mu_0 \nu_1 \nu_2) \cdot (DFT_{1D}^{\mu_0 \mu'_0} \cdot U^{\mu'_0 \nu_0}), \end{aligned} \quad (16)$$

Figure 6: Merging the padded 1D iDFT with the projection of the Fourier transformed data points on the non-local projector $\Phi^{(lr)}(\mu_0\mu_1\mu_2)$. The tensors are depicted as matrices using a column major layout. The padded 1D iDFT reads strided batches of data given the size of the dimensions $(\mu_0\mu_1)$, applies zero-padding followed by the batched 1D iDFT (I). The same stride is used to read the corresponding projector data points (II). The contraction is performed, and partial results update the tensor T_2 (III).

where $U^{\mu_0\nu'_0}$, $U^{\mu_1\nu'_1}$, $U^{\mu_2\nu'_2}$ are the corresponding two dimensional matrices required for the un-padding steps. Like the padded DFT, we call the merged 3D forward DFT with the pruning operation as the un-padded DFT. The data is Fourier transformed and only the necessary data points are extracted from the cube and stored into main memory, discarding the unnecessary values. We use the mirror image forward DFT stages to further optimize across the DFT and the real space linear algebra operations, as outlined in the following subsection.

3.3 Merging the DFT with the Linear Algebra

Parts of the DFT computation can be grouped and merged with the projections operations that involve the Φ tensor. First, the forward and inverse padded 3D DFTs must apply their computation in mirror images of each other. If the padded 3D DFT applies the 1D computations in the μ_0 , μ_1 and μ_2 , then the un-padded computation applies its steps in the μ_2 first and μ_0 last. This allows us to re-arrange the computation steps such that parts of the Fourier computation surround the linear algebra operations as

$$\begin{aligned} T_1^{(b)}(\mu_0\mu_1\mu_2) &= T_{03}^{(b)}(\mu_0\mu_1\nu_2) \cdot (P^{\nu_2\mu'_2} \cdot iDFT_{1D}^{\mu'_2\mu_2}) \\ T_2^{(b)}(lr) &= T_1^{(b)}(\mu_0\mu_1\mu_2) \cdot \Phi^{(lr)}(\mu_0\mu_1\mu_2) \\ T_1^{(b)}(\mu_0\mu_1\mu_2) &= T_1^{(b)}(\mu_0\mu_1\mu_2) \odot V^{(\mu_0\mu_1\mu_2)} \\ T_1^{(b)}(\mu_0\mu_1\mu_2) + &= T_2^{(b)}(lr) \cdot \Phi^{(lr)}(\mu_0\mu_1\mu_2) \\ T_{03}^{(b)}(\mu_0\mu_1\nu_2) &= T_1^{(b)}(\mu_0\mu_1\mu_2) \cdot (DFT_{1D}^{\mu_2\mu'_2} \cdot U^{\mu'_2\nu_2}). \end{aligned} \quad (17)$$

Astute readers should immediately identify that the point-wise computation with the ionic potential V , and the DFT stages are part of a DFT-based convolution. Hence, merging the padded DFT stage with the pointwise and the un-padded DFT should be a given optimization step. However, there are extra operations between the DFT stages, namely the projection of the T_1 tensor on the non-local projectors stored in Φ , and the scaling of the projectors in Φ by the result in T_2 tensor and the update of T_1 . T_2 must be fully computed before the update to the T_1 tensor. Because of the dependency, merging of the compute stages is not that straightforward.

A solution is to create two groups of operations, such that the dependency is preserved. As such, the padded 1D DFT is duplicated and distributed to the operations that use the T_1 tensor. The first of these operations is defined as

$$\begin{aligned} T_1^{(b)}(\mu_0\mu_1\mu_2) &= T_{03}^{(b)}(\mu_0\mu_1\nu_2) \cdot (P^{\nu_2\mu'_2} \cdot iDFT_{1D}^{\mu'_2\mu_2}) \\ T_2^{(b)}(lr) &= T_1^{(b)}(\mu_0\mu_1\mu_2) \cdot (\Phi^{(lr)}(\mu_0\mu_1\mu_2))^T. \end{aligned} \quad (18)$$

The two operations can be merged as shown in Figure 6. The Fourier computation is applied on batches of data from T_{03} . The batches are read at a stride equal to the size of the μ_0 and μ_1 dimensions. Locally the batches are zero-padded, and Fourier transformed. The result is stored in a local buffer (Stage I). The projection on the Φ tensor can be immediately computed. The correct data from Φ must be read (Stage II). Recall that the projection is done in the $\mu_0\mu_1\mu_2$ dimension. In addition, the data from T_{03} is read at a stride given by the μ_0 and μ_1 dimensions. This implies that parts of the Φ tensor must be read at the same stride as the data from T_{03} . Once data is on chip the local contraction can proceed (Stage III). The local operation computes partial results to tensor T_2 . The frequent updates to T_2 suggest T_2 be stored locally as well. The DFT computation and local projector is not computed every time given the sparse nature of data stored in Φ .

The second group of operations where T_1 appears can be re-written as

$$\begin{aligned} T_1^{(b)}(\mu_0\mu_1\mu_2) &= T_{03}^{(b)}(\mu_0\mu_1\nu_2) \cdot (P^{\nu_2\mu'_2} \cdot iDFT_{1D}^{\mu'_2\mu_2}) \\ T_1^{(b)}(\mu_0\mu_1\mu_2) &= T_1^{(b)}(\mu_0\mu_1\mu_2) \odot V^{(\mu_0\mu_1\mu_2)} \\ T_1^{(b)}(\mu_0\mu_1\mu_2) + &= T_2^{(b)}(lr) \cdot \Phi^{(lr)}(\mu_0\mu_1\mu_2) \\ T_{03}^{(b)}(\mu_0\mu_1\nu_2) &= T_1^{(b)}(\mu_0\mu_1\mu_2) \cdot (DFT_{1D}^{\mu_2\mu'_2} \cdot U^{\mu'_2\nu_2}). \end{aligned} \quad (19)$$

Operations can be merged as depicted in Figure 7. The padded 1D DFT is applied on batches of data points in T_{03} , where the batches are accessed at a stride given by the μ_0 and μ_1 dimensions. The batches are read, padded, Fourier transformed and stored into a local buffer (Stage I). The point-wise scaling with V is done locally (Stage II). The data stored in V must be read at the same stride as

Figure 7: Merging the inverse and forward 1D DFTs with the point-wise computation with V and the update of the scaled projector Φ . Batches of the data are read, padded and Fourier transformed (I). The point-wise scaling with the ionic potentials is then performed (II). The corresponding components of the projectors are packed in local memory (III). The projectors are scaled by the data stored in T_2 and the values added to the local buffer (IV). Once all updates are performed, the data points are Fourier transformed and only the relevant data points are stored back into the same locations as the input batches (V).

T_{03} . The access to V are amortized, because the same value from V is used multiple times for all the data in the b dimension. Once the point-wise scaling operation is complete, the scaled projectors can update the local buffer. Parts of projectors from Φ are accessed at the same stride as accessing T_{03} , and packed into a local buffer (Stage III). The projection update (Stage IV) requires data from T_2 , which can also be packed into a local buffer. Because T_2 is read multiple times, the tensor is preferred to reside in the cache hierarchy, not main memory. Finally, the inverse 1D DFT is computed, then $U^{\mu_2'\mu_2}$ filters the necessary data points, and stores only those points back to the corresponding locations in T_{03} (Stage V).

One would argue that the 1D DFT computations are duplicated for the first and second group of operations. Note that the T_1 tensor is not used anymore. Recall that the T_1 tensor is almost 16 times the size of the original Ψ tensor. Merging the operation reduces the amount of data moved to and from main memory, which is more important. The fusion of the compute stages does come at a cost. Buffers are required to store the local computation, which need to be tailored to the DFT size and the batch dimension. In addition, other buffers like T_2 must be kept close due to the frequency of reading and writing into T_2 . However, the frequency of reading and writing to T_2 is tied to the data layout of the projectors as discussed in the following section.

4 DISCUSSION AND RESULTS

In this section, we present the implementation details and the results of our approach. We give a brief description of the data format used to store the wavefunctions and the non-local pseudopotential projectors. Then we describe the architectural choices like SIMD instructions and parallelism that were adopted within the code. Finally, we present results of our approach on both CPU and GPU systems. We compare our results against conventional library-based implementations, but we also provide an analysis of where the performance is coming from. Finally, we present some potential future directions one could take to automate and extend this approach to other applications and other hardware systems.

4.1 Data Storage

The Ψ and Φ tensors are block-sparse tensors. The Ψ tensor is composed of multiple wavefunctions batched together given the b dimension. Recall that in frequency space, the wavefunctions are stored as spheres given a cut-off energy. As such, we use a Compressed Sparse Row (CSR) [6] like format to store Ψ as shown in Figure 8. We project the sphere on the $v_1 v_2$ plane. We then flatten the plane and construct the indexing array. This array is the offset into the planewaves tensor when stored in memory. An array corresponding to the indices in the v_0 is also constructed. This array is used to specify how to pad the Fourier computation. The padded DFT computation starts with the data in v_0 dimension. In addition, the padded DFT does not change from one wavefunction to another. As such, one copy of the indexing arrays is used for all planewaves, since all planewaves have the same size.

The Φ tensor is composed of multiple projectors which are located at the center of each atom. Each atom has l projectors, where l varies between 8 and 18 depending on the atomic types. We use a Compressed Sparse Column (CSC) like format to store the projectors in Φ as shown in Figure 9. Like the planewave sphere, the projector spheres are mapped onto the $\mu_0 \mu_1$ plane. The plane is then flattened to construct the indexing array into Φ . Additional arrays are built for indexing into the μ_2 and r dimensions. We opt for a CSC format for the projectors, because merging the linear algebra operations with the DFT stages requires the projector data to be read at a stride. The DFT computation is applied in the slowest dimension in memory, which requires the projectors to be read at the same stride as discussed in the previous section. We opt to follow the directions from the papers [19, 42], which advocate for changing the data layouts such that computation is done efficiently. In this case, the projectors in Φ are compute once and used multiple times, hence permuting them seems a good solution.

4.2 Code Implementation

We implement each mathematical kernel from scratch, for both the CPU and the GPU. We are merging the computation and we

Figure 8: The CSR-like format to store the planewaves. The sphere is projected on the $v_1 v_2$ plane, and the plane is flattened and used as an offset indexing array.

need to full control over the computation and most important the data movement. We apply a bottom-up approach, where build the smallest computations given the number of registers, and then we loop around them. We tile and parallelize the computation based on the memory hierarchy.

Fourier Transforms. We implement batched DFTs based on the Cooley-Tukey algorithm [10]. We construct codelets that perform batched 1D DFTs on sequential or strided data. The codelets use either a temporary buffer to store intermediate results, or they use the input or the output as temporary buffer and do parts of the computation in-place. We modify the 1D DFTs to allow for the padding or un-padding operations to be done as part of the Fourier computation. As such, each 1D DFT accepts an indexing array to specify which data points are valid data points. We loop over the codelets to construct the stages of the 3D DFTs as needed in our implementation. For the CPU implementation, we use AVX instructions to vectorize the code, we pack the complex data points as block interleaved and perform the computation at cacheline lengths as shown in [28]. In addition, for the stages that write data to main memory we use non-temporal store operations to by-pass the cache hierarchy as suggested in [31]. As for the thread level parallelism, we use OpenMP [27] constructs. We request the user to create a pool of threads before calling the computation. Within the code we use the thread id and omp barrier to control the threads. We offer two levels of parallelism, one on the loops around the codelets, and one within the batch dimension. We do not parallelize within the 1D DFT. The GPU implementation follows the same structure as the CPU code. Because each streaming multi-processor has a fast shared memory and can execute multiple warps concurrently, we also parallelize the 1D DFT among the warps within a thread block.

Linear Algebra Operations. The operations that involve the Φ tensor are block sparse matrix times dense matrix operations. Because the blocks are dense, we use the guidelines from [16, 37] to implement the micro-kernels, the packing routines and the loops around the two. We use the analytical model proposed in [23] to determine the corresponding tiling parameters for the micro-kernel and loops around the micro-kernel. For the matrix multiplications, we provide two variants. One variant has all the loops required to do

Figure 9: The CSC-like format to store the projectors. The spheres are projected on the $\mu_0 \mu_1$ plane, and the plane is flattened and used as an offset indexing array.

Performance for the $H\Psi$ Computation
AMD Ryzen 7 3800X, 8/16 Cores/Threads, AVX, 3.9 GHz

Figure 10: Results obtained on the AMD Ryzen 7 processor for our approach (red bar), the baseline (blue bar) and the version that optimizes the computation in isolation (yellow bar). The results show relative speedup compared to the execution time of the baseline implementation.

both projections in their entirety. We use this implementation to improve the computation of the baseline implementation, as discussed in the following subsection. The second variant has only some of the loops, because it is merged with the Fourier computation as depicted in Figures 6 and 7. For the CPU implementations, we use AVX instructions to vectorize the micro-kernel, and scalar code to implement the packing routines. The thread level parallelism is handled by OpenMP. In addition, we use the details from [36] to parallelize the loops given the shared levels of the cache hierarchy. We request the user to create a pool of threads, and then within the computation we use the thread ids and barriers to control the computation. Like the Fourier transforms, we port the code to the GPU, and adjust the micro-kernel size as outlined in [4]. The GPU implementation follows the details from the paper, however instead of using bit-wise operations we use the normal double precision floating point instructions.

Figure 11: The top two plots show the execution breakdown for the $H\Psi$ when implemented using vendor-tuned libraries (left plot) and our optimized computations (right plot). The bottom two plots show the performance differences between the padded DFTs (left plot) and the contractions (right plot). All results are obtained on the AMD Ryzen 7 CPU.

4.3 Analysis and Results

In the following paragraphs, we present the methodology and results obtained on various architectures.

Platforms. We run all codes on the following architectures

- AMD Ryzen 7 3800X CPU 8/16/32MB cores/threads/L3 cache with 64 GB of DRAM - personal desktop
- AMD EPYC 7302 CPU 2/16/32/128MB sockets/cores/threads/L3 cache with 2 TB of DRAM - single node on NERSC Cori Supercomputer
- NVIDIA V100 GPU - 80 streaming multi-processors with 16 GB of DRAM - single node on NERSC Cori Supercomputer
- NVIDIA A100 GPU - 108 streaming multi-processors with 40 GB of DRAM - single node on NERSC Cori Supercomputer

On the AMD Ryzen processor, we use the aocl-fftw.2.2 library for computing the Fourier transforms, and the aocl-blis.2.2 library for the BLAS operations. For this system, we disable turbo boosting

and set the performance scaling governor to performance. On the supercomputer nodes we could not configure the turbo boost, so we used the default system configuration. On the AMD EPYC processor we use the fftw.3.5.6 and openblas.0.2.20 libraries provided by Cray. The implementation of the pointwise scaling was done by hand using AVX instructions and the OpenMP parallel for construct to specify the thread level parallelism. On the GPU, we use cuFFT 11.0.2 library for the Fourier transforms. We tried the cuSPARSE library for computing the non-local pseudopotential projections. However, due to the indexing arrays into the projector array, we could not make this variant execute. As such we resorted to implementing the matrix operations by hand using native CUDA.

Baseline Implementation. We build the baseline implementations for the CPU and the GPU, following the details from [18, 40, 41]. For the DFT computation we use the FFTW interface. More specifically, we use the plan_many_dft to perform batched 3D

Figure 12: Performance results on the AMD EPYC CPUs. The left plot shows performance of $H\Psi$ on the AMD EPYC, where we compare our optimized dataflow approach (red) against the baseline (blue). The right plot shows performance of the same $H\Psi$ on the AMD EPYC, where we keep the Fourier transform size fixed, and increase the bands and the number of atoms.

DFTs on the planewaves. Because FFTW does not provide any functionality to do zero-padding or to remove unnecessary data points, we implement the functions outside the FFTW function calls as depicted in Figure 5.a. We tried implementing the other version with the FFTW Guru Interface, however the execution time of both the sequential and the parallel versions were worse than simply expanding the planewaves up front and then applying the batched 3D DFTs. For the non-local pseudopotential projections the paper suggests using the dense matrix-matrix multiplication, by invoking dgemm from the BLAS routines. For each atom the projectors are stored contiguously in memory, as opposed to our way of storing the data points as shown in Figure 9. The projectors are read contiguously, however the corresponding planewave data points are read at a stride. Because BLAS does not allow one to insert packing routines to read strided data, we implement our own packing routines. For each atom, the data is packed, and then corresponding BLAS routine is invoked.

For the GPU, the papers suggest a band-by-band algorithm for the $H\Psi$. Each planewave is zero padded, Fourier transformed, and then updated with the projector values and inverse transformed. The process is repeated for all the planewaves. The papers opt for this variant because of memory constraints. Recall that expanding the planewaves from frequency to real space requires approximately 16 times more storage. We do not choose this variant, and we follow the guidelines from [8] and implement the all band implementation on the GPU as well. For the Fourier computation we use the cuFFT interface to perform batched 3D DFTs. We tried the cuFFT callback functions to perform the zero padding, however the execution time was worse than simply having a CUDA kernel that packs the data before the 3D DFT. As for the non-local pseudopotential operations, we implemented the code from scratch. Given that the planewaves are tall and skinny, the matrix operations are memory bound. We tested the implementations and both contractions achieve 90% bandwidth utilization on both GPU systems.

Parameter Sweep. For all the experiments, we can modify the size of the Fourier transform, the number of planewaves, the number of atoms, and the number of projectors per atom. The diameter of the planewave spheres is always chosen to be half the size of the 1D DFT. We choose the Fourier size to always be a power of two, and the batch dimension to be multiple of eight (cacheline length or eight threads from a warp). The number of atoms can be any positive value greater than one. The number of projectors per atom can be any value between 8 and 18. We set the diameter of the spheres to be equal to 26 real space grid points and for each atom we set the number of projectors to 18. The projector dimensions depend on physical properties and the values can vary. We run experiments with other projector values, however we have not seen any performance changes. The objective of choosing different ranges of parameters is to correspond to a large range of different physical systems with different numbers of atoms and types, although some of the parameters correspond most closely to a bulk silicon system.

Single Socket AMD Ryzen CPU. In Figure 10, we report the performance results obtained on an AMD Ryzen 7 CPU. We report relative speedup against the baseline implementation that uses vendor-optimized libraries (blue bar). The yellow bar represents the speedup over the baseline when optimizations are applied to the individual computation steps, like the DFT and the projections, but called in the same sequence as the conventional library implementation. The red bar represents our approach where we merge the computation as discussed in this work. Even though the optimized DFT and linear algebra kernels perform better than the vendor optimized version, merging the stages offers further improvement. Merging the stages reduces the amount of data moved between CPU and main memory and improves data locality.

We provide an analysis on how the performance changes when the vendor libraries are replaced with our own implementations of the DFT and contraction operations. The top two plots in Figure 11 show the execution breakdown for the $H\Psi$. The left plot shows

Figure 13: Performance results of the $H\Psi$ computation on the NVIDIA V100 (left) and A100 (right) GPUs. We compare our optimized dataflow approach (red bar) against the baseline implementation (blue) that uses cuFFT library calls.

the breakdown for the vendor library implementation. For small number of bands, the execution is dominated by the contractions, however as the number of bands is increased the Fourier transforms become dominant. This breakdown changes when we optimize the library calls. The left plot in Figure 11 shows the breakdown when computation is more efficient. Note that the contraction is within 30% of the overall execution, and that the point-wise computation becomes more prominent and comparable to the Fourier computations. The pointwise operation just reads data from memory, does a simple floating point computation and stores the results back. The implementation streams the data at bandwidth speed.

In the bottom two plots, we show the performance improvements for the DFT and contraction operations. In the left plot, we report the performance of the padded DFT. We compare our approach that merges the padding with the DFT computation against the FFTW variant, where padding is done up front and a 3D batched DFT is applied on the larger input. Our padded DFT moves roughly 2.8x fewer data points and uses the cache hierarchy more efficient, which explains the performance difference between the two implementations. In the right plot, we report performance results for the two contractions. The performance gains are more obvious for these operations. Our implementation packs the data more efficiently and the parallelism is better exploited. All the experiments were executed using eight threads, each thread pinned to its own core. For the baseline implementation, more specifically for the FFTW and BLAS routines, we use the environment variables to specify the number of threads. For the remainder of this section, we do not report the performance of the yellow bar, since it is similar and always provide a boost over the baseline implementations.

Multi-Socket AMD EPYC CPU. We use the same problem sizes as in the AMD Ryzen experiments, and we run the codes on a two-socket system with a larger L3 cache. In Figure 12, the two plots show the performance of the $H\Psi$ computation on the AMD EPYC CPUs. For larger sizes we see a significant performance boost as shown in the left plot in Figure 12. Our approach achieves almost 12x performance gains compared to the baseline implementation. The dip in performance for the first few configurations is caused

by the data layout used to store the non-local pseudopotential projectors. The data layout interferes with the parallelization. When parallelizing across the local computation, a locking mechanism is needed to ensure that the data in T_2 is correctly updated. If the number of atoms is sufficiently small, then the locking mechanism will influence the overall execution time. We leave as future work the exploration of alternative data layouts [22, 35] that may remove the critical code sections. In the right plot, we show results where we keep the DFT size fixed, but we increase the number of bands and more importantly we increase the number of atoms. The execution of the baseline implementation depends linearly on the number of atoms. As the number of atoms is increased, the amount of data moved through the memory system increases and the baseline execution time goes up. This effect is not seen in our dataflow approach because the projectors are only brought once for the corresponding DFT patches. On the AMD EPYC system we use both sockets and 32 threads each pinned to its own core. We leave the study of how the NUMA domain affects the computation as future work.

Single GPU. In Figure 13 we report results obtained on NVIDIA’s V100 and A100 GPUs. For our dataflow approach we set the threads within a block and the number of thread blocks. Each thread block uses 128 threads, and the number of blocks is set to four times the number of streaming processors. The number of thread-blocks is constrained by the amount of shared memory that is being used by each block. For the baseline implementation, we use cuFFT for the Fourier transforms, for which we do not have control over the threads. However, for the pointwise computation and the contractions we use similar configurations as the ones used for our merged implementation. We perform a sweep of thread and block configurations and choose the one that performs the best. We leave as future work the process of analytically determining an efficient mapping for the threads. For the problem sizes, we report fewer cases as the cases on the CPU, because for some variants cuFFT cannot allocate the necessary temporary arrays. While we show a 2x improvement against the baseline implementation, the performance gains are smaller compared to the gains obtained on the CPU. One possible

explanation is the fact that merging the computation requires multiple writes to the temporary buffer T_2 , which is frequently updated with partial results. T_2 is small enough to fit on the CPU's L3 cache, however it is sufficiently large to overflow from shared memory. As such, the partial updates need to be performed in L2 cache or worse in main memory. In the conventional implementation of the contractions, T_2 is kept in registers and updated once. The result is then written at the end directly to main memory. We leave as future work the development of a more thorough model to outline the possible drawbacks of our approach on GPUs.

Future Opportunities. While we show good results on both CPU and GPU systems, we believe that there is still work to be done. On the CPU side we need to look at other data layouts to represent the block sparse tensors. In the current implementation, we are dealing with locking mechanisms that hinder the parallelism. We need to find a way to make the code more robust. On the GPU side, we need to analyze the algorithmic changes and see whether merging the computation adds any value. Merging the operations improves data locality, however it also requires temporary buffers and changes to the way data is accessed. These modifications may hinder the efficient warp execution and may force the warps to write more to main memory. Besides the improving the current approach, we believe that other applications that use the Fourier transform as a building block may benefit from the notation and the approach developed in this work. The ideas presented in this paper can be incorporated into projects like FTX [14], that provides a high level API to describe and automatically optimize Fourier based applications. In addition, the methodology can be used to target other hardware platforms like SambaNova Cardinal chip [32, 34] or GraphCore. This may require one to automate parts of the approach. The current implementation is built as a standalone package where the optimizations are done manually. However, code generators like Spiral [13], Halide [33], TACO [20] or polyhedral frameworks like Pluto [5], Poly [17] may offer more flexible solutions to test the target the code for other platforms. We leave these aspects as future work.

5 CONCLUSION

In this work, we showed that by casting the Fourier computation as a multi-linear algebra operation, we can create a systematic approach for merging DFT stages with the surrounding sparse and dense linear algebra operations. This enabled us to provide performance improvements to one key component of the planewave Density Functional Theory codes, namely the $H\Psi$ computation that makes use of Fourier transforms, point-wise operations and block sparse linear algebra operations. We implemented the computation on CPUs and GPUs and showed performance gains on both. On CPUs with large last level cache, we showed that the approach of merging the Fourier transforms with the pointwise and sparse operations can provide up to 8x/12x performance improvements against the classical implementation that uses off-the-shelf libraries. The performance boost is attributed to better implementations, but most important to the merging of the computation steps. On GPUs, we used the same technique and showed a performance gain of almost 2x. While the GPU gains are smaller compared to the CPU ones, there is still work to be done to further improve the code

and make it more robust. While the optimizations were applied to Density Functional Theory, we believe that similar steps, analysis, and implementations can be done for other applications that use the Fourier transform as a building block alongside linear algebra operation. In addition, we think that new hardware like static-data-flow architectures with large on chip memory can offer the perfect environment for such codes, and the methodology developed in this paper can help to target such platforms.

ACKNOWLEDGEMENTS

This work was supported by the Laboratory Directed Research and Development Program of Lawrence Berkeley National Laboratory and the National Energy Research Scientific Computing Center Office of Science of the U.S. Department of Energy under Contract No. DE-AC02-05CH11231, and the resources of the Oak Ridge Leadership Computing Facility at the Oak Ridge National Laboratory, which is supported by the Office of Science of the U.S. Department of Energy under Contract No. DE-AC05-00OR22725.

REFERENCES

- [1] 2009. *Intel Math Kernel Library. Reference Manual*. Intel Corporation.
- [2] AMD. 2018. AMD Optimized FFTW. <https://github.com/amd/amd-fftw>.
- [3] Brian Austin. 2018. NERSC NERSC-10 Workload Analysis. https://portal.nersc.gov/project/m888/nersc10/workload/N10_Workload_Analysis.latest.pdf.
- [4] E. Binder, T. M. Low, and D. T. Popovici. 2019. A Portable GPU Framework for SNP Comparisons. In *2019 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. 199–208. <https://doi.org/10.1109/IPDPSW.2019.00041>
- [5] Uday Bondhugula, Albert Hartono, J. Ramanujam, and P. Sadayappan. 2008. A Practical Automatic Polyhedral Parallelizer and Locality Optimizer. In *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation (Tucson, AZ, USA) (PLDI '08)*. Association for Computing Machinery, New York, NY, USA, 101–113. <https://doi.org/10.1145/1375581.1375595>
- [6] Aydin Buluç, Jeremy T. Fineman, Matteo Frigo, John R. Gilbert, and Charles E. Leiserson. 2009. Parallel Sparse Matrix-Vector and Matrix-Transpose-Vector Multiplication Using Compressed Sparse Blocks. In *Proceedings of the Twenty-First Annual Symposium on Parallelism in Algorithms and Architectures (Calgary, AB, Canada) (SPAA '09)*. Association for Computing Machinery, New York, NY, USA, 233–244. <https://doi.org/10.1145/1583991.1584053>
- [7] Andrew Canning. 2008. Scalable Parallel 3d FFTs for Electronic Structure Codes. In *High Performance Computing for Computational Science - VECPAR 2008*, José M. Laginha M. Palma, Patrick R. Amestoy, Michel Daydé, Marta Mattoso, and João Correia Lopes (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 280–286.
- [8] R. Car and M. Parrinello. 1985. Unified Approach for Molecular Dynamics and Density-Functional Theory. *Phys. Rev. Lett.* 55 (Nov 1985), 2471–2474. Issue 22. <https://doi.org/10.1103/PhysRevLett.55.2471>
- [9] James Cooley and John Tukey. 1965. An Algorithm for the Machine Calculation of Complex Fourier Series. *Math. Comp.* 19, 90 (1965), 297–301.
- [10] James W Cooley and John W Tukey. 1965. An algorithm for the machine calculation of complex Fourier series. *Mathematics of computation* 19, 90 (1965), 297–301.
- [11] Jack J. Dongarra, Jeremy Du Croz, Sven Hammarling, and Iain Duff. 1990. A Set of Level 3 Basic Linear Algebra Subprograms. *ACM Trans. Math. Soft.* 16, 1 (March 1990), 1–17.
- [12] B. Driscoll and Z. Zhao. 2020. Automation of NERSC Application Usage Report. In *2020 IEEE/ACM International Workshop on HPC User Support Tools (HUST) and Workshop on Programming and Performance Visualization Tools (ProTools)*. 10–18. <https://doi.org/10.1109/HUSTProTools51951.2020.00009>
- [13] Franz Franchetti, Tze Meng Low, Doru-Thom Popovici, Richard Michael Veras, Daniele G. Spampinato, Jeremy R. Johnson, Markus Püschel, James C. Hoe, and José M. F. Moura. 2018. SPIRAL: Extreme Performance Portability. *Proc. IEEE* 106, 11 (2018), 1935–1968. <https://doi.org/10.1109/JPROC.2018.2873289>
- [14] Franz Franchetti, Daniele G. Spampinato, Anuva Kulkarni, Doru-Thom Popovici, Tze Meng Low, Michael Franasich, Andrew Canning, Peter McCorquodale, Brian Van Straalen, and Phillip Colella. 2018. FTX and SpectralPack: A First Look. In *2018 IEEE 25th International Conference on High Performance Computing Workshops (HiPCW)*. 18–27. <https://doi.org/10.1109/HiPCW.2018.8634111>
- [15] Matteo Frigo and Steven G. Johnson. 1997. *The Fastest Fourier Transform in the West*. Technical Report MIT-LCS-TR-728. Massachusetts Institute of Technology.

- [16] Kazushige Goto and Robert van de Geijn. 2008. Anatomy of high-performance matrix multiplication. *ACM Trans. Math. Soft.* 34, 3 (May 2008), 12:1–12:25.
- [17] Tobias Grosser, Hongbin Zheng, Ragheesh Aloor, Andreas Simbürger, Armin Größlinger, and Louis-Noël Pouchet. 2011. Polly-Polyhedral optimization in LLVM. In *Proceedings of the First International Workshop on Polyhedral Compilation Techniques (IMPACT)*, Vol. 2011. 1.
- [18] Weile Jia, Zongyan Cao, Long Wang, Jiyun Fu, Xuebin Chi, Weiguo Gao, and Lin-Wang Wang. 2013. The analysis of a plane wave pseudopotential density functional theory code on a GPU machine. *Computer Physics Communications* 184, 1 (2013), 9–18. <https://doi.org/10.1016/j.cpc.2012.08.002>
- [19] Kyungjoo Kim, Timothy B. Costa, Mehmet Deveci, Andrew M. Bradley, Simon D. Hammond, Murat E. Guney, Sarah Knepper, Shane Story, and Sivasankaran Rajamanickam. 2017. Designing Vector-Friendly Compact BLAS and LAPACK Kernels. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (Denver, Colorado) (SC '17)*. Association for Computing Machinery, New York, NY, USA, Article 55, 12 pages. <https://doi.org/10.1145/3126908.3126941>
- [20] Fredrik Kjolstad, Shoaib Kamil, Stephen Chou, David Lugato, and Saman Amarasinghe. 2017. The Tensor Algebra Compiler. *Proc. ACM Program. Lang.* 1, OOPSLA, Article 77 (October 2017), 29 pages. <https://doi.org/10.1145/3133901>
- [21] Ricardo A Lebensohn, Anand K Kanjarla, and Philip Eisenlohr. 2012. An elastoviscoplastic formulation based on fast Fourier transforms for the prediction of micromechanical fields in polycrystalline materials. *International Journal of Plasticity* 32 (2012), 59–69.
- [22] Jiajia Li, Jimeng Sun, and Richard Vuduc. 2018. HiCOO: Hierarchical Storage of Sparse Tensors. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis (Dallas, Texas) (SC '18)*. IEEE Press, Article 19, 15 pages. <https://doi.org/10.1109/SC.2018.00022>
- [23] Tze Meng Low, Francisco D. Igual, Tyler M. Smith, and Enrique S. Quintana-Orti. 2016. Analytical Modeling Is Enough for High-Performance BLIS. *ACM Trans. Math. Softw.* 43, 2, Article 12 (August 2016), 18 pages. <https://doi.org/10.1145/2925987>
- [24] Nvidia. 2018. NVidia Cuda Based FFT Library. <https://developer.nvidia.com/cufft>.
- [25] NVIDIA, Péter Vingelmann, and Frank H.P. Fitzek. 2020. CUDA, release: 10.2.89. <https://developer.nvidia.com/cuda-toolkit>
- [26] OpenBLAS 2012. <http://xianyi.github.com/OpenBLAS/>.
- [27] OpenMP Architecture Review Board. 2020. OpenMP Application Program Interface Version 5.0.
- [28] Doru-Thom Popovici, Franz Franchetti, and Tze Meng Low. 2017. Mixed data layout kernels for vectorized complex arithmetic. In *2017 IEEE High Performance Extreme Computing Conference, HPEC 2017, Waltham, MA, USA, September 12-14, 2017*. 1–7.
- [29] Doru-Thom Popovici, Martin D. Schatz, Franz Franchetti, and Tze Meng Low. 2020. A Flexible Framework for Multidimensional DFTs. *SIAM J. Sci. Comput.* 42, 5 (2020), C245–C264. <https://doi.org/10.1137/19M1288401>
- [30] Doru Thom Popovici. 2018. *An Approach to Specifying and Automatically Optimizing Fourier Transform Based Operations*. Ph.D. Dissertation. Electrical and Computer Engineering, Carnegie Mellon University.
- [31] Doru-Thom Popovici, Tze-Meng Low, and Franz Franchetti. 2018. Large Bandwidth-Efficient FFTs on Multicore and Multi-Socket Systems. In *IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE.
- [32] Raghu Prabhakar, Yaqi Zhang, David Koeplinger, Matthew Feldman, Tian Zhao, Stefan Hadjis, Ardavan Pedram, Christos Kozyrakis, and Kunle Olukotun. 2018. Plasticine: A Reconfigurable Accelerator for Parallel Patterns. *IEEE Micro* 38, 3 (2018), 20–31. <https://doi.org/10.1109/MM.2018.032271058>
- [33] Jonathan Ragan-Kelley, Connelly Barnes, Andrew Adams, Sylvain Paris, Frédo Durand, and Saman P. Amarasinghe. 2013. Halide: a language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines. In *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '13, Seattle, WA, USA, June 16-19, 2013*, Hans-Juergen Boehm and Cormac Flanagan (Eds.). ACM, 519–530. <https://doi.org/10.1145/2491956.2462176>
- [34] SambaNova Inc. 2020. SambaNova home. <http://sambanova.ai>.
- [35] Shaden Smith and George Karypis. 2015. Tensor-Matrix Products with a Compressed Sparse Tensor. In *Proceedings of the 5th Workshop on Irregular Applications: Architectures and Algorithms (Austin, Texas) (IA3 '15)*. Association for Computing Machinery, New York, NY, USA, Article 5, 7 pages. <https://doi.org/10.1145/2833179.2833183>
- [36] Tyler M. Smith, Robert van de Geijn, Mikhail Smelyanskiy, Jeff R. Hammond, and Field G. Van Zee. 2014. Anatomy of High-Performance Many-Threaded Matrix Multiplication. In *IPDPS '14: Proceedings of the International Parallel and Distributed Processing Symposium*. To appear.
- [37] Field G. Van Zee and Robert A. van de Geijn. 2013. BLIS: A Framework for Rapid Instantiation of BLAS Functionality. *ACM Trans. Math. Soft.* (2013). In review.
- [38] Nicolas Vasilache, Jeff Johnson, Michael Mathieu, Soumith Chintala, Serkan Piantino, and Yann LeCun. 2014. Fast convolutional nets with fbfft: a GPU performance evaluation. *arXiv preprint arXiv:1412.7580* (2014).
- [39] J-L Vay, A Almgren, J Bell, L Ge, DP Grote, M Hogan, O Kononenko, R Lehe, A Myers, C Ng, et al. 2018. Warp-X: A new exascale computing platform for beam-plasma simulations. *Nuclear Instruments and Methods in Physics Research Section A: Accelerators, Spectrometers, Detectors and Associated Equipment* (2018).
- [40] Long Wang, Yue Wu, Weile Jia, Weiguo Gao, Xuebin Chi, and Lin-Wang Wang. 2011. Large scale plane wave pseudopotential density functional theory calculations on GPU clusters. In *SC'11: Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 1–10.
- [41] Lin-Wang Wang, Byounghak Lee, Hongzhang Shan, Zhengji Zhao, Juan Meza, Erich Strohmaier, and David H Bailey. 2008. Linearly scaling 3D fragment method for large-scale electronic structure calculations. In *SC'08: Proceedings of the 2008 ACM/IEEE Conference on Supercomputing*. IEEE, 1–10.
- [42] Jiyuan Zhang, Franz Franchetti, and Tze Meng Low. 2018. High performance zero-memory overhead direct convolutions. In *International Conference on Machine Learning*. PMLR, 5776–5785.