Generating Optimized Fourier Interpolation Routines for Density Functional Theory using SPIRAL

Doru Thom Popovici*, Francis P. Russell†, Karl Wilkinson‡, Chris-Kriton Skylaris‡
Paul H. J. Kelly† and Franz Franchetti*
*Electrical and Computer Engineering, Carnegie Mellon University, Pittsburgh, U.S.A.
†Department of Computing, Imperial College London, London, UK
‡School of Chemistry, University of Southampton, Southampton, UK

***Abstract*—Upsampling of a multi-dimensional data-set is an operation with wide application in image processing and quantum mechanical calculations using density functional theory. For small upsampling factors as seen in the quantum chemistry code ONETEP, a time-shift based implementation that shifts samples by a fraction of the original grid spacing to fill in the intermediate values using a frequency domain Fourier property can be a good choice. Readily available highly optimized multidimensional FFT implementations are leveraged at the expense of extra passes through the entire working set. In this paper we present an optimized variant of the time-shift based upsampling. Since ONETEP handles threading, we address the memory hierarchy and SIMD vectorization, and focus on problem dimensions relevant for ONETEP. We present a formalization of this operation within the SPIRAL framework and demonstrate auto-generated and auto-tuned interpolation libraries. We compare the performance of our generated code against the previous best implementations using highly optimized FFT libraries (FFTW and MKL). We demonstrate speed-ups in isolation averaging 3x and within ONETEP of up to 15%.**

I. Introduction

The Fourier interpolation of a multi-dimensional data-set is an operation with applications to image processing, quantum mechanical calculations using density functional theory and image acquisition algorithms like synthetic aperture radar. Upsampling is the special case of filling in the signal at intermediate locations, which is usually performed using a three step process: 1) compute the frequency domain representation of the data cube through a fast Fourier transform (FFT) 2) zero-pad the spectrum appropriately, 3) compute the inverse FFT. Variants exist for non-uniform up-sampling based interpolation, and pruned FFTs [1] can be used to avoid operating on known zero entries. The major issue with this approach is that the inverse FFT has to operate on a much larger data set since the zero-padded data set is 8 times the original data set for an upsampling factor of two in three dimensions, and much larger for higher dimensions and larger upsampling factors.

For small upsampling factors (for instance, two in each dimension) an alternative approach is often a more efficient way to compute the finer-resolution signal. Using the Fourier property that time shift is equivalent to pointwise multiplication of the spectrum with appropriately chosen complex roots of unity [2], one can compute a signal shifted by *one half* of the grid spacing, in effect computing the intermediate values through two FFTs of the same size and a pointwise scaling. This saves considerable operations but requires the interleaving of the original and shifted signal which can be costly on modern machines.

Our work was inspired by the quantum chemistry package ONETEP [3] which performs such a two-fold upsampling on a three-dimensional data cube, and where recent work has shown that a time shift based approach indeed has performance advantages [4]. The remaining issues addressed in this paper are:

1) If one implements the time shift-based interpolation using the best available FFT packages to compute the forward and inverse FFTs, one is left with an expensive data reorganization stage at the end of the computation that needs to interleave the original and shifted signal in all three dimensions.
2) ONETEP requires relatively small 3D FFTs (data cube edges no larger than 200), however, the problem sizes need to be *odd* due to properties of the basis functions used.

High performance FFT kernels need to leverage single instruction, multiple data (SIMD) vector instruction sets such as Intel's SSE and AVX vector extensions. Implementing medium sized odd FFT kernels using these instructions is difficult due to the complexities introduced by alignment and vector length divisibility. In order to address these two challenges, we perform optimizations at two levels:

Firstly, we perform data layout optimizations and rearrange the traversals through the data set. We manipulate the data access patterns to perform the entire operation in two memory round trips, and never have to interleave the data explicitly. This cannot be implemented using standard FFT libraries, therefore we have to build our own high performance odd sized SIMD vectorized FFT kernels.

Secondly, we carefully cross-optimize the multiple FFT stages that exist in convolutions underlying upsampling and prime size FFTs. We perform vector-aware zero-padding that inserts the minimum number of extra zeroes required by the vector extension at the initial data loading and drops them as the data set grows throughout the interpolation. This is a very involved process that requires non-standard partial FFT kernels that are not available in any standard FFT library.

Since both optimizations require non-standard high performance FFT kernels, we used the automatic performance tuning and program generation system SPIRAL [5] to auto-generate and auto-tune these kernels from the algorithm and data layout specifications we derive to perform the various optimizations.

In our experimental evaluation we establish that our optimized kernel is providing substantial performance improvements over the best other upsampling implementation available to ONETEP. We show that the performance is obtained both from the memory level optimizations and the kernel level SIMD vector optimizations. We are able to auto-tune the upsampling operation since we generate the kernel from a high level representation that allows for code variants.

Contribution. The paper makes these main contributions:

- We present a highly memory-optimized version of the time-shift upsampling algorithm that computes the final output with only two memory round trips and without explicit interleaving.
- We present novel SIMD vectorization techniques for prime size FFTs and for convolution like FFT-based multi stage algorithms.
- We present a full formal specification of the vectorized upsampling algorithm in the Kronecker product formalism. Using SPIRAL we demonstrate the autogeneration of highly optimized upsampling implementations from these specifications.

In isolation we see that our interpolation kernels outperform IMKL and FFTW by factors of 2x and 3x on average and up to 6x and 5x, respectively. ONETEP runs for a number of quantum chemistry relevant problem sizes show performance gains of up to 15%.

II. ONETEP and Upsampling

A. The ONETEP Package

Background on ONETEP. ONETEP [3] (Order-N Electronic Total Energy Package) is a software package for performing quantum-mechanical calculations using density functional theory. ONETEP achieves linear-scaling in the system size through approximations which exploit the "nearsightedness of electronic matter" [6].

In density functional theory, a fictitious system of non-interacting particles can be completely described by the set of Kohn-Sham orbitals $\{\psi_i(\mathbf{r})\}$. ONETEP reformulates density functional theory in terms of the one-particle density matrix as follows:

$$\rho(\mathbf{r}, \mathbf{r}') = \sum_{\alpha} \sum_{\beta} \phi_{\alpha}(\mathbf{r}) K^{\alpha\beta} \phi_{\beta}(\mathbf{r}'), \tag{1}$$

where the "density kernel" $\mathbf{K}$ is the density matrix expressed in the duals of a set of localized functions, $\{\phi_{\alpha}(\mathbf{r})\}$, called non-orthogonal generalized Wannier functions (NGWFs). NGWFs are constrained to be strictly localized within spherical regions (localization spheres) and are expressed in a psinc basis set. Psinc functions are centered on the points of a regular real-space grid and are related to a plane-wave basis through Fourier transforms. The density matrix $\rho(\mathbf{r}, \mathbf{r}')$ differs significantly from zero only for points r' within a finite volume around r, a property ONETEP exploits.

FFT-based Interpolation in ONETEP. The computation of quantities involving operators in momentum space requires a Fourier transform of the real-space grid on which the psinc functions are centered. Rather than transform the entire simulation cell, ONETEP performs these transforms over subsets of the grid called FFT boxes. FFT boxes are boxes of grid points centered on an NGWF, and large enough to contain any overlapping NGWF localization spheres in their entirety. The relationship between NGWFs, FFT boxes and the simulation cell is illustrated in Figure 1a.

Choosing the shape and size of the FFT box so that it encloses any overlapping pair of support functions ensures that operators applied inside remain Hermitian and that certain quantities have a unique and consistent representation throughout the calculation. As a consequence, all FFT boxes used in a simulation are identically sized, and posses an odd number of grid points in each dimension. It is acceptable in ONETEP for an FFT box to be larger than the minimum determined size. ONETEP often enlarges FFT boxes so that their size reflects the transform sizes most efficiently computed by the FFT library in use (typically products of small primes), but the size in each dimension must remain odd. In the majority of the cases, ONETEP's enlargement process will create boxes that will be smaller than 130 in each dimension.

The calculation of quantities in ONETEP such as the charge density and and local potential involves expressions that contain products between NGWFs. These products introduce high frequency components that cannot be represented with the existing psinc basis. Since a simple point-wise multiplication of FFT boxes would introduce aliasing, both are interpolated onto a psinc grid with twice as many points in each dimension before the product is formed. Upsampling of FFT boxes is an extremely computationally intensive part of ONETEP.

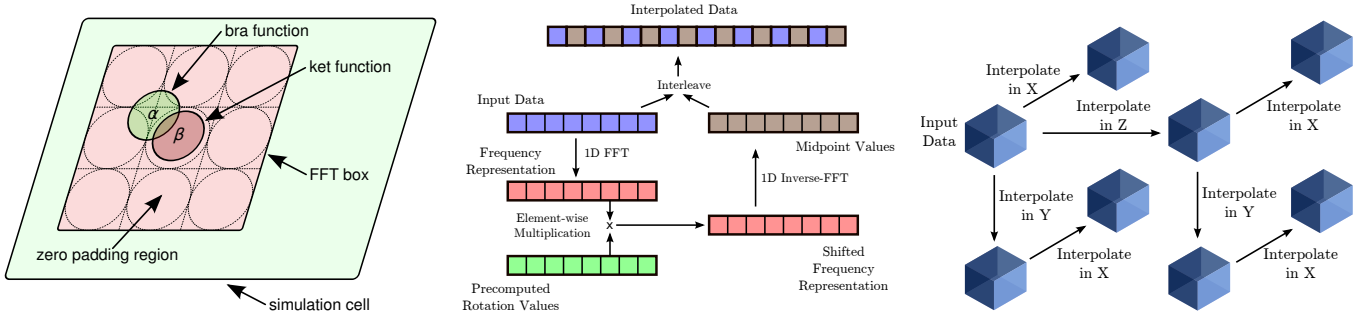
B. Upsampling Through a 1/2 Sample Shift

In order to improve ONETEP's performance, we construct a high-performance implementation of the interpolation operation. Specifically, we trigonometrically interpolate a function sampled on a three-dimensional regular grid to a grid where the number of grid-points has been doubled in each dimension.

Baseline: Standard FFT Upsampling. It is possible to implement this operation using existing discrete Fourier transform implementations. The theoretical and actual performance of different algorithms have been investigated in other work [4]. Typically, the data-set to be interpolated is transformed to frequency space, followed by a padding step which introduces new zero-amplitude high-frequency components. The signal is transformed back to the original domain using an inverse Fourier transform, producing the upsampled version [2].

Although it is possible to implement this operation using two three-dimensional Fourier transforms (a forward and a backward), there is benefit to implementing the inverse transform using one-dimensional transforms; this is possible due to the fact that the multi-dimensional Fourier transform is a separable operation. The introduction of the zero-padding causes many of the one-dimensional transforms to operate only on zeros and can be omitted entirely [7].

Although this "padding-aware" approach eliminates transforms that operate on entirely zero-valued input, there is further redundancy due to the fact that the remaining transforms operate on inputs that contain 50% zeros.



(a) An FFT box centered on NGWF β, which overlaps NGWF α. FFT box size remains constant for the simulation and is chosen such that when centered on an arbitrary NGWF, it contains all overlapping NGWFs.

(b) One-dimensional interpolation via phase shift [4]. The original signal is phase-shifted in the frequency domain and the two versions are then interleaved.

(c) Phase-shift interpolation in 3 dimensions [4]. 8 versions of the input signal are interleaved to produce the final output.

Fig. 1: FFT box and interpolation operation in ONETEP.

Half-sample shift based upsampling. An alternative approach eliminates the zero-padding step entirely. In the one-dimensional case, the algorithm (illustrated in Figure 1b) proceeds as follows:

1) Perform a discrete Fourier transform of the input dataset.
2) Multiply each frequency coefficient by a phase-shift value that delays (or advances) that component by $\frac{1}{2}$ a sample.
3) Perform an inverse discrete Fourier transform. The resulting signal has the same frequency components as the original signal, but is sampled at the *midpoints* between samples of the original signal.
4) Interleave the original input with the interpolated midpoint values to form the upsampled signal.

The "phase-shift" algorithm generalizes to the multi-dimensional case, with a "tree" of interpolations required to construct the newly introduced points. The interpolations required in the three-dimensional case are illustrated in Figure 1c.

The interpolation of each block requires multiple one-dimensional interpolations in a specific dimension. Once all blocks are constructed, these are interleaved to form the interpolated signal in a similar manner to the one-dimensional case.

The baseline, "padding-aware" and "phase-shift" algorithms have been implemented in the TINTL library [4] and used to accelerate ONETEP. These implementations rely on an underlying FFT library such as FFTW or Intel's MKL. We test our generated code against these implementations within ONETEP in Section VII.

III. Formal Framework

A. Kronecker Product Formalism and FFTs

In this paper we use the matrix-vector product representation of the discrete Fourier transform (DFT) and fast Fourier transform (FFT) [8]. In this section we briefly introduce the Kronecker product formalism used to describe fast algorithms as matrix factorizations. This framework is used by the SPIRAL code generation and auto-tuning system [5] which we use to automatically generate our interpolation kernels.

Discrete Fourier transform. The DFT of n input samples $x_0, \ldots, x_{n-1}$ is defined as the matrix vector product $y = \mathrm{DFT}_n\, x$ with

$$\mathrm{DFT}_n = [\omega_n^{k\ell}]_{0 \le k,\ell < n} \quad \text{with} \quad \omega_n = \exp(-2\pi j/n). \tag{2}$$

In (2) we are stacking the x_ℓ and y_k into vectors x and y. To derive a declarative problem specification we drop x and y and simply think of the matrix DFT_n as the transform. Throughout the entire paper we use the DFT_n notation for the forward Fourier transform and the iDFT_n for the inverse Fourier transform.

Matrix formalism and SPL language. The fast Fourier transform is obtained by factorizing the matrix DFT_n into a product of structured sparse matrices. In the SPIRAL system the SPL language is used to capture data-flow at a higher level of abstraction. The SPL language is essentially a language that describes matrix factorization using mathematical formulas. The entire language starts from the notion of the Kronecker product or tensor product which can be expressed as follows:

$$A \otimes B = [a_{k,\ell} B], \quad \text{for } A = [a_{k,\ell}].$$

The direct sum operation $\oplus$ is defined as usual. Besides the tensor product and direct sum, we also make use of some additional elements that will help us in later derivations. Permutations or shuffle operations are additional elements used in the derivation of both the FFT and the 3D interpolation algorithms. The permutations are represented as sparse matrices such as the *strided permutation* L_m^{mn} matrix or the *Rader permutation* W_p matrix, defined as:

$$L_m^{mn} : in + j \mapsto jm + i, \quad 0 \le i < m,\ 0 \le j < n$$
$$W_p : i \mapsto g^i \bmod n, \quad 0 \le i < p.$$

Details on other permutation matrices and how they are used within the SPIRAL framework can be found in [9]. We also provide a generalization for the identity matrix I_n, namely

$$I_{m\times n} = \begin{bmatrix} I_n \\ O_{m-n\times n} \end{bmatrix}, \quad m \ge n,$$
$$I_{m\times n} = \begin{bmatrix} I_m & O_{m\times n-m} \end{bmatrix}, \quad m < n.$$

Matrix formula	Matlab pseudo code
$y = (A_n B_n)x$	`t[0:1:n-1] = B(x[0:1:n-1]);` `y[0:1:n-1] = A(t[0:1:n-1];)`
$y = (I_m \otimes A_n)x$	`for (i=0; i<m; i++)` `  y[i*n:1:i*n+n-1] = A(x[i*n:1:i*n+n-1]);`
$y = (A_m \otimes I_n)x$	`for (i=0; i<n; i++)` `  y[i:n:i+m*n-n] = A(x[i:n:i+m*n-n]);`
$y = D_n x$	`for (i=0; i<n; i++)` `  y[i] = Dn[i]*x[i];`

$y = L^{mn}_m x$	`for (i=0; i<m; i++)`
	`  for (j=0; j<n; j++)`
	`  y[i+m*j] = x[n*i+j];`

TABLE I: From matrix formulas to code, in Matlab notation.

Here, $O_{m\times n}$ is the $m \times n$ all-zero matrix. $I_{n\times n}$ is the identity matrix I_n.

There are various identities connecting these constructs [10]. For example,

$$A_m \otimes B_n = L^{mn}_m (B_n \otimes A_m) L^{mn}_n \text{ and } L^{mn}_m L^{mn}_n = I_{mn}. \quad (3)$$

Formulas can be translated into code, following Table I. Details can be found in [5].

Fast Fourier transform. We now summarize how the Kronecker product formalism is used to capture the fast Fourier transform. For $n = 2$ the discrete Fourier transform is the *butterfly matrix*

$$\mathrm{DFT}_2 = \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix}. \quad (4)$$

For larger sizes $N = mn$ that can be factorized into two numbers, the Cooley-Tukey FFT algorithm

$$\mathrm{DFT}_{mn} = (\mathrm{DFT}_m \otimes I_n) D_{m,n} (I_m \otimes \mathrm{DFT}_n) L^{mn}_m \quad (5)$$

breaks a larger DFT into a number of smaller DFTs. For prime size FFTs, Rader's algorithm

$$\mathrm{DFT}_p = W_p^{-1} (I_1 \oplus \mathrm{DFT}_{p-1}) E_p (I_1 \oplus \mathrm{DFT}_{p-1}) W_p \quad (6)$$

breaks an DFT of prime size p into a convolution (and thus two DFTs) of size $p-1$, which is guaranteed to be a composite number for $p > 3$, and thus (5) applies. For $p = 3$, the base case (4) applies. In (5) and (6), the L^{mn}_m and W_p notations represent the two different permutation matrices. Furthermore, I_n is the $n \times n$ identity matrix, and $D_{m,n}$ is a diagonal matrix containing the so-called twiddle factors and the E_p matrix is "almost" a diagonal matrix with 2 additional off diagonal elements. More details on the different FFT algorithms can be found in [9].

Inplace computation. We will use a $\underline{A}$ to indicate that the operation $y = Ax$ is to be performed *inplace*, i.e., the output is being written back to the input vector. If the matrix $A_{k\times m}$ is rectangular ($m \mid k$) then inplace implies that the input is embedded in the output at stride k/m. Further, a product $\underline{A}\,B$ for rectangular A implies that B scatters the data to embed it into the output vector of A. The symbol $\overline{1}$ means that the corresponding vector element will not be written as it remains unchanged throughout the formula.

B. SIMD Vectorization of FFTs

SIMD instructions. Modern CPUs feature SIMD vector extensions to speed up arithmetic-intensive computation kernels. Such SIMD vector extensions add vector registers (from 2-way for double precision SSE2 available since the Intel Pentium 4 to 16-way single precision for on Xeon Phi). Current Intel server processors typically feature AVX, which provides 4-way double precision vectors 8-way single precision vectors. AVX includes an SSE compatibility mode and thus support for 2-way double precision and 4-way single precision. Subsequent generations of SSE and AVX introduced ever more complex instructions to support complex arithmetic, partial vector loads/stores, and in-register data reorganization.

Translating SPL to vector code. SIMD vectorization of FFT algorithms relies on a simple but powerful observation: A Kronecker product formula that is a product of the following terms,

$$A \otimes I_\nu, \quad D_n, \quad L^{\nu^2}_\nu, \quad \text{and} \quad L^{2\nu}_\nu \quad (7)$$

can be implemented efficiently with vector instructions [11], [12]. Note that 4-way double precision AVX leads to $\nu = 2$ since we have to pack complex numbers into the vector. For instance, the two-way vectorized butterfly operation $y = (\mathrm{DFT}_2 \otimes I_2)x$ can be implemented using AVX instructions and the Intel C++ compiler *intrinsics* interface as follows:

```
__m256d x[2], y[2];
y[0] = _mm256_add_pd(x[0], x[1]);
y[1] = _mm256_sub_pd(x[0], x[1]);
```

The AVX implementation of $y = L^4_2 x$ is given below:

```
__m256d x[2], y[2];
 y[0] = _mm256_permute2f128_pd(x[0], x[1], 0x20);
 y[1] = _mm256_permute2f128_pd(x[0], x[1], 0x31);
```

C. SPIRAL

SPIRAL is a program generation and optimization system for transforms including one and higher-dimensional DFTs [5]. For a given DFT of size N, Spiral expands DFT_N recursively using (5) and (6) as well as other FFT algorithms until base cases ($N = 2$) are reached. The resulting tensor product expression is converted to highly efficient C code that utilizes SIMD vector instructions (through the intrinsics/builtin interface) and supports multithreading [12]. SPIRAL performs empirical search to find good recursive decomposition choices for m and n, traversing a large search space.

In this paper we extend Spiral to support the SIMD vectorization of a large class of odd-sized FFTs, including sizes that require loop code and Rader's algorithm. We then extend SPIRAL to support time shift-based interpolation and the necessary optimization steps. In Section VII, all our results are fully auto-generated and auto-tuned using the extended SPIRAL system we build based on the formal algorithm description and optimization derived in Sections V and VI.

IV. Formalization of 3D Upsampling

We now discuss how to formalize the 3D upsampling algorithm in the Kronecker product formalism. This is a prerequisite to perform optimizations through formula manipulation.

Upsampling through 1/2 sample time shift. The time shift property of the DFT shows that shifting the signal in the time domain is equivalent to multiplying the signal with properly chosen complex roots of unity in the frequency domain. Therefore, a time shifted signal can be computed by first computing its Fourier transform, then performing a pointwise multiplication with the complex roots of unity, and finally computing an inverse Fourier transform. This is essentially the convolution of the original signal with a signal consisting of complex exponentials [2]. Specifically, the shift operator Z_n^k that computes a time shifted signal by k samples (k not necessarily integer) can be written as

$$Z_n^k = \text{iDFT}_n \; D_n^k \; \text{DFT}_n \tag{8}$$

with

$$D_n^k = \text{diag}\left(1, \omega_n^k, \omega_n^{k^2} \ldots, \omega_n^{k^{n-1}}\right) \text{ and } \omega_n = \exp(-2\pi j/n).$$

Note that for integer k the matrix Z_n^k is the *cyclic shift matrix* and consists of two all-one off-diagonals that rotate the data by an integer sample increment. The 2-fold upsampling operator $\mathcal{U}_n^2$ can be expressed as the interleaving of the original signal with a signal time-shifted by $k = 1/2$, or formally,

$$\mathcal{U}_n^2 = L_n^{2n} \begin{bmatrix} I_n \\ Z_n^{1/2} \end{bmatrix}. \tag{9}$$

This is the upsampling kernel we will be using for the remainder of the paper.

3D upsampling. 3D upsampling interpolates a 3D signal by filling in the missing data in all three dimensions. It represents an increase of the resolution in each dimension, i.e. finds a smooth signal that goes through the grid points and has the same energy, and sample that signal according to the new sample rate. The 3D upsampling can be represented as Kronecker product of 1D upsampling operations,

$$\mathcal{U}_{k\times m\times n}^{2\times 2\times 2} = \mathcal{U}_k^2 \otimes \mathcal{U}_m^2 \otimes \mathcal{U}_n^2. \tag{10}$$

Higher-dimensional FFT-based upsampling is a separable operation, due to the properties of the FFT. Therefore, performing 3D FFT-based upsampling reduces itself to applying the 1D upsampling operation to all *pencils* in each dimension, similar to the row-column FFT algorithm.

Row-column algorithm. 3D upsampling is best done using the 3D row-column algorithm that first works within contiguous data planes (the xy plane), and then does a final upsampling along the z dimension. Formally, this can be described by the matrix factorizations

$$\mathcal{U}_{k\times m\times n}^{2\times 2\times 2} = (\mathcal{U}_k^2 \otimes I_{4mn})(I_k \otimes \mathcal{U}_{m\times n}^{2\times 2}) \text{ and} \tag{11}$$

$$\mathcal{U}_{m\times n}^{2\times 2} = (\mathcal{U}_m^2 \otimes I_{2n})(I_m \otimes \mathcal{U}_n^2). \tag{12}$$

This approach is similar to the *slab* decomposition for 3D FFTs and ensures maximum locality and cache reuse for problem sizes where a full plane fits into a cache level. This situation is applicable in our case, as k, m, and n are relatively small such that a data plane fits into cache, but the full upsampled cube of memory footprint $8kmn$ becomes too big for the last level cache for many of the sizes of interest to ONETEP.

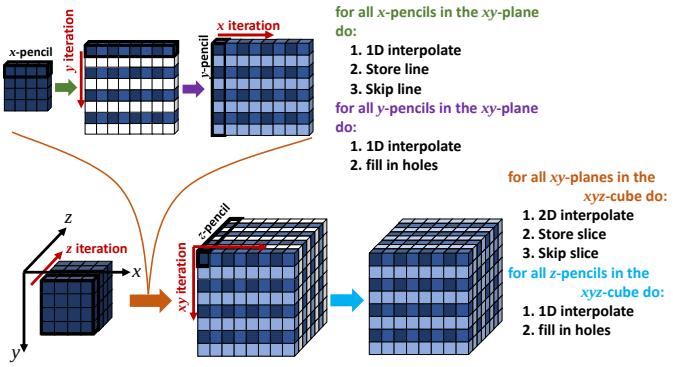


Fig. 2: Two-pass upsampling approach in SPIRAL-generated code. Original data is touched only once. Subsequent upsmpling pencils fill in the holes only through 1/2 sample shift.

V. Locality and Memory Hierarchy Optimization

In this section we discuss the memory hierarchy optimizations. We perform two main optimizations: Firstly, we perform a data layout transformation that writes all the interpolated data immediately to its final location, avoiding a final data shuffle stage. Secondly, we perform loop merging that merges FFT stages from adjacent forward and inverse FFT transforms and pointwise scaling to minimize the times the data set needs to be traversed.

A. Locality and Data Layout Optimization

The original upsampling approach discussed in Section II-B computes 8 data blocks separately that in the end need to be interleaved in one big copy operation that involves inefficient strided access. In our formalization, the 1D upsampling operation (9) implies that the input data first time-shifted by $1/2$ sample and then the original data is interleaved with the time shifted data, requiring strided memory access and an inefficient copy operation. The corresponding 2D and 3D upsampling operations (11) and (12), respectively, imply that the whole data set has to be interleaved three times, once for each dimension. Either approach is inefficient.

Main idea. Careful analysis shows that the correct breakdown of the 3D interpolation together with a data layout transformation allows us to write the data at the final location as each time shift kernel produces the data. The first stage in addition needs to copy the original data to its final location. Correctly picking the order of upsampling dimensions as given by (11) and (12) makes the writes cache friendly as every cache line that gets touched is written completely, and touched only once.

We show the procedure in Fig. 2. We first upsample the xy plane, followed by upsampling the z pencils. Together with copying the original data, the x pencils upsampling output is written to memory so that original and shifted data are interleaved. Between two x pencils space for the y direction upsampling is left. The y pencil upsampling fills in the blanks left by the x upsampling. Between two adjacent xy planes, an empty xy plane is left, that will be filled in by the z pencils.

The x direction is the fastest varying dimension, thus this approach is compatible with SIMD vectorization and cache

lines. In addition, the last level cache can hold at least one xy plane, and proper traversal of the xy iteration space when computing the z pencils ensures cache friendliness of the z pencil stage.

We now express this optimization formally using the Kronecker product formalism.

x pencils. The first stage computes the x pencils, i.e., in x direction the shifted version of the input data. It takes the original and shifted input, interleaves the two data sets and finally scatters the entire data to the final destination in the output array. The full sweep over an xy plane where all the x pencils are computed and empty pencils are interleaved between upsampled x pencils to spread the data out properly in the xy plane is given by

$$\mathcal{U}_n^x = I_m \otimes \begin{bmatrix} \mathcal{U}_n^2 \\ O_{2n \times n} \end{bmatrix}. \tag{13}$$

y pencils. The second stage reads the non-zero values of y pencils from their final location, computes the shifted intermediate values and inserts them between the values already computed by the x traversal. The interpolation kernel

$$\mathcal{U}_m^y = \left(\begin{bmatrix} 0 \\ 1 \end{bmatrix} \otimes I_m \right) \left(Z_m^{1/2} \otimes [1\,0] \right)$$

gathers data from a y pencils at stride 2 (starting at element 0), computes the shifted version, and writes them in y pencil at stride 2 (starting at element 1).

z pencils. The third stage reads the non-zero values of z pencils from their final location, computes the shifted intermediate values and inserts them between the values already computed by the x and y traversals. The interpolation kernel $\mathcal{U}_n^z = \mathcal{U}_n^z$ gathers data from a z pencils at stride 2 (starting at element 0), computes the shifted version, and writes them in z pencil at stride 2 (starting at element 1).

Full upsampling algorithm. We now give the full upsampling algorithm as SPL formula. The timeshift operation that gathers data from a x or y pencil and inserts them between the original values can be written as

$$\mathcal{Z}_\ell = \underline{L_n^{2n} \begin{bmatrix} I_n \otimes [\overline{1}\,0] \\ Z_\ell^{1/2} \otimes [1\,0]. \end{bmatrix}} \tag{14}$$

We now express the operation on an xy plane as

$$\mathcal{U}_{m \times n}^{xy} = \left(\underline{\mathcal{Z}_m \otimes I_{2n}} \right) (I_m \otimes \mathcal{U}_n^x). \tag{15}$$

The full algorithm can now be expressed as

$$\mathcal{U}_{k \times m \times n}^{2 \times 2 \times 2} = \left(\underline{\mathcal{Z}_k \otimes I_{4mn}} \right) \left(I_k \otimes \begin{bmatrix} \mathcal{U}_{m \times n}^{xy} \\ O_{4mn} \end{bmatrix} \right). \tag{16}$$

B. Merging of FFT Stages in Convolutions

The memory level optimization so far ensures that data is written exactly once, at the right final location, all data access patterns are compatible with cache-based memory hierarchies. It takes advantage of the relatively small edge length of the data cube by employing the slab decomposition to break down the computation into two phases, instead of 3 phases that would result from the standard row-column approach. The remaining memory level optimizations are targeting data reuse inside the cache by merging of loops that traverse the data set in multiple passes.

FFT stage merging. We leverage the observation that the core operation in upsampling is a FFT-based convolution, as shown in (8). Expanding the DFT in (8) with (5) and the iDFT with the transposed rule, we obtain

$$\begin{aligned} Z_{mn}^k = L_n^{mn} (I_m \otimes \mathrm{iDFT}_n) D'_{m,n} (\mathrm{iDFT}_m \otimes I_n) \\ D_{mn}^k (\mathrm{DFT}_m \otimes I_n) D_{m,n} (I_m \otimes \mathrm{DFT}_n) L_m^{mn}. \end{aligned} \tag{17}$$

For DFT sizes where DFT_m and DFT_n can be implemented as basic block but DFT_{mn} requires loop code, (17) can be implemented using 4 passes through the data by merging the permutations and diagonals with the tensor products [9]. Using a parameterized version of the tensor product, $\otimes_i$, that allows the non-identity matrix factor to be dependent on the location of the 1 in the identity matrix [5], we can merge the last stage of the forward FFT with the diagonal and the first stage of the inverse FFT,

$$\begin{aligned} (\mathrm{iDFT}_m \otimes I_n) D_{mn}^k (\mathrm{DFT}_m \otimes I_n) \\ = (\mathrm{iDFT}_m \, D_{mn}^{k,i} \, \mathrm{DFT}_m) \otimes_i I_n, \end{aligned} \tag{18}$$

which leads to a 3 stage implementation for (17). The key idea is to expand the inverse FFT and the forward FFT so that their data flow graphs are the mirror image of each other, which can always be accomplished.

Stage merging in Rader's FFT. A similar pattern is seen in the Rader algorithm (6) that breaks down an DFT of prime size p into a convolution of size $p-1$. Except for the two off-diagonal elements in E_p the same approach as for merging loops in (17) can be applied. This way Rader's FFT can be implemented with three stages as well when DFT_p is too big for unrolled code but DFT_m and DFT_n for $m, n \mid p-1$ can be unrolled.

The same idea can now be applied to convolutions of prime size that require Rader's FFT for the DFT/iDFT. In this case three neighboring DFT/iDFT stages need to be merged: merging the DFT/iDFT stages in the two Rader FFT kernel for the convolution DFT and iDFT, respectively, and merging of the DFT/iDFT stages for the convolution due to (8). The full prime size 1D shift operation Z_p^k then requires 5 stages.

VI. Vectorization

In the previous section we optimized the upsampling kernel for the memory hierarchy. The remaining task is to optimize the 1D pencil kernels for the SIMD vector instructions present on all modern microprocessors. This task involves three sub-steps: 1) SIMD vectorization of the odd-sized 1D FFT kernels through vector-aware local zero-padding, 2) cross-block optimization of the FFT and iFFT in the convolution kernels to reduce zero-padding overhead at the boundaries of the 1D FFT kernels, and 3) optimization of the three upsampling phases of the 3D upsampling kernel to take advantage of the fill-in and 3D anisotropy of the memory linear memory viewed as data cube.

A. Vectorization of Odd-Sized FFTs

The quantum chemistry package ONETEP has somewhat unusual FFT requirements: 1) the FFTs need to be odd-sized

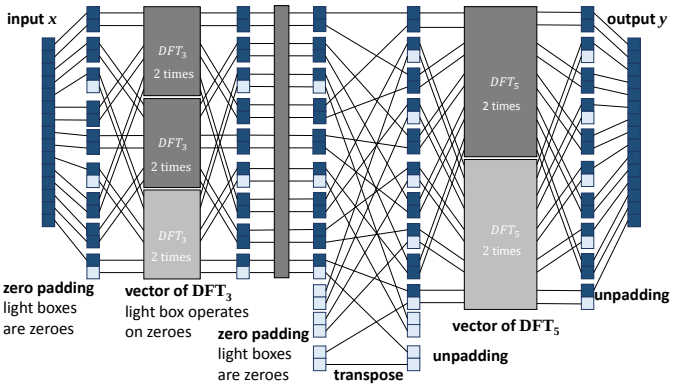


Fig. 3: DFT decomposition with zero-padding for size $N = 15$ and vector length $\nu = 2$. Light boxes represent inserted zero value cells, and staggered pairs of cells represent SIMD vectors or SIMD vectorized DFT kernels. Data flows from right to left.

due to the properties of the basis functions used, and 2) the FFT sizes of interest are smaller than 130, thus medium-sized, i.e., too large for unrolled basic blocks but small enough that they can be computed with two stages. Therefore, neither the approach taken by FFTW's `genfft` [13] nor by SPIRAL's odd sized vectorized FFT kernels [1] are applicable.

In the remainder of this section we extend SPIRAL's approach from unrolled odd-sized FFTs to loopable medium-sized FFTs. The following two novel techniques were necessary: 1) a loopable *odd-size short vector Cooley Tukey FFT variant*, and 2) a loopable *short vector Rader FFT variant* and *convolution*.

Approach. (7) in Section III-B collects the formula constructs that can be translated into SIMD vector instructions efficiently. In particular, for arbitrary A, $A \otimes I_\nu$ can be implemented solely with vector instructions with vector length ν. Constructs $A \otimes I_n$ with ν not dividing n cannot be mapped directly to vector instructions in an efficient way. The idea of vector-aware zero-padding [1] provides a solution: as data gets first loaded, zero-extend the vectors, and as the final result is written to memory, un-pad, i.e., only store the elements that correspond to originally loaded elements. Formally, this is expressed as

$$A_m \otimes I_n = (I_m \otimes I_{n \times \nu\lceil n/\nu \rceil})(A_m \otimes I_{\nu\lceil n/\nu \rceil})(I_m \otimes I_{\nu\lceil n/\nu \rceil \times n}).$$

This is the only new construct needed to describe very sophisticated zero-padding operations necessary to implement loopable odd-sized FFT algorithms.

Algorithm. We now derive the loopable short vector Cooley Tukey FFT for odd sizes. We start from (5), and use (3) to flip the right tensor product and move the stride permutation between the two stages, leading to the *Four Step FFT algorithm*,

$$(\mathrm{DFT}_m \otimes I_n) D_{m,n} L^{mn}_m (\mathrm{DFT}_n \otimes I_m).$$

Next we carefully zero-pad to the minimum extent necessary. We define $m' = \nu\lceil m/\nu \rceil$, $n' = \nu\lceil n/\nu \rceil$. (19) in Fig. 4 shows the final formula.

Discussion. (19) has two stages, each of which performs iterations of $\mathrm{DFT}_m \otimes I_\nu$ or $\mathrm{DFT}_n \otimes I_\nu$. The loops are extended to a trip count that is divisible by ν. The zero-padding via $I_{m' \times m}$ and unpadding via $I_{n \times n'}$ is minimized, and only for the central stride permutation $L^{m'n'}_{m'}$ that is factorized into three terms full padding is needed. The twiddle diagonal $D_{m,n}$ is zero-padded as well and moved to the right FFT stage.

The overhead is kept to a minimum, and all operations are performed with vector operations of length $\nu = 2$. $L^{\nu^2}_\nu$ is a construct in (7) and can be implemented efficiently with vector instructions. In a high performance implementation the padding is fused into the load operation of the first stage but then requires one loop iteration to be peeled to handle the zero extension. Fig. 3 shows the data flow graph of (19) for DFT_{15} with $\nu = 2$, $m = 3$, and $n = 5$ and the stride permutation L^{24}_4 not factorized.

B. Vectorization of Convolutions and Rader FFT

The core operation in pencil upsampling is a convolution, given by (8). This requires that an DFT and iDFT are computed back-to-back (with an intermediate scaling operation). Similar to the optimizations in Section V-B we can take advantage of this fact to minimize the number of padding and unpadding operations.

Vectorization of convolution. We utilize that for $k > \ell$,

$$I_{k \times \ell} I_{\ell \times k} = I_\ell \oplus O_{k-\ell}. \tag{20}$$

We now write the 1D 2x upsampling operation $\mathcal{U}^2_{mn}$ as

$$A_{mn \times m'n}(I_n \otimes I_{m' \times m}) D^k_{mn} (I_n \otimes I_{m \times m'}) B_{m'n \times mn}, \tag{21}$$

using (17) and (19). We are expanding the iDFT_{mn} using the expansion of DFT_{mn} and apply the identity $\mathrm{iDFT}_n = \mathrm{DFT}^H_n$ with $(.)^H$ denoting conjugate-transpose. In (21) $A_{mn \times m'n}$ represents the full expansion of the iDFT according to (19), except for the zero-padding stage. $B_{m'n \times m'n}$ represents the full expansion of the DFT according to (19), except for the unpadding stage. Padding and unpadding have been made explicit. By applying (20) to (21) we obtained the optimized convolution/upsampling formula

$$\mathcal{U}^2_{mn} = A_{mn \times m'n} D'^k_{mn} B_{m'n \times mn} \tag{22}$$

where the padding and unpadding has been dropped. The diagonal matrix D^k_{mn} is extended from size $mn \times mn$ to $m'n \times m'n$, with zeroes inserted according to the dropped padding/unpadding stages.

Vectorization of Rader's FFT. Since Rader's FFT computes the DFT for a prime size p by convertting it into a convolution of size $p - 1$, the same idea applies. The only additional complication arises from the fact that E_p is not exactly a diagonal matrix and that the permutations W_p cannot be vectorized and are hard to loop. Our solution is to implement a *vector gather/scatter* on complex elements and use an indirection table with the precomputed indices to avoid online computation of integer exponentiation and modulo.

C. Vectorization of the 3D Upsampling Kernel

There are a few more vector optimizations necessary for the full 3D upsmapling operation beyond the optimization of the 1D pencil upsampling.

$$\mathrm{DFT}_{mn} = (I_m \otimes I_{n\times n'})\big((\mathrm{DFT}_m \otimes I_{n''}) \otimes I_\nu\big) D'_{m,n} (I_{n'} \otimes I_{m\times m'})\big(L^{m'n''}_{m'} \otimes I_\nu\big)$$
$$\big(I_{m''n''} \otimes L^{\nu^2}_\nu\big)\big(I_{n''} \otimes L^{m'}_{m''} \otimes I_\nu\big)(I_{m''n'\times m''n} \otimes I_\nu)\big((\mathrm{DFT}_n \otimes I_{m''}) \otimes I_\nu\big)(I_n \otimes I_{m'\times m}) \quad (19)$$

Fig. 4: Loopable Short Vector Cooley Tukey FFT for odd sizes, with $m' = \nu\lceil m/\nu\rceil$, $n' = \nu\lceil n/\nu\rceil$, $m'' = m'/\nu$, $n'' = n'/\nu$.

Zero-padding. The first observation is that the data can stay zero-padded across interpolation stages at very little memory cost, avoiding unpadding and re-padding between x and y pencils and y and z pencils. Due to the fill-in at each stage only for vector lengths $\nu > 4$ unpadding at the end of the z pencils is necessary–in all other cases the 8-fold expansion of data volume makes unpadding unnecessary.

Vectors of DFTs. The second observation is that for the y and z pencils, one can compute *a vector of DFTs*,

$$\mathrm{DFT}_n \otimes I_\nu,$$

instead of having to vectorize the actual DFT. This allows stage 2 and 3 to be computed without vector overhead. The only shuffle operations in these stages are due to shuffles required for complex multiplications. However, since the data cube dimension is odd, zero-padding is happening in the loop traversing the pencils. In stage 2, there are $2m$ iterations and in stage 3 there are $4km$, limiting the zero-padding overhead.

Interleaving. Finally, in the first stage, the original data needs to be merged with the interpolated data. This happens along the fastest varying dimension and thus has to happen within vector registers. By picking the x dimension as the first dimension to work on we are minimizing the overhead of data copying and ensure vector memory access.

VII. Experimental Results

Setup. We now evaluate the performance of our approach. We run experiments across a variety of Intel CPUs that feature the SSE and AVX instruction set extensions: 1) 3.5 GHz Haswell 4770K, 2) 3.3 GHz Ivy Bridge Xeon E3-1230, and 3) 2.1 GHz Sandy Bridge Xeon E5-2620. We use the Intel C++ Compiler 14.0.1, Intel MKL 11.0.0, FFTW 3.3.4, and ONETEP 4.1.5.8. We compare three performance aspects: 1) performance of the interpolation kernel in isolation, 2) performance of the underlying 1D FFTs, and 3) performance impact of our interpolation kernel on a ONETEP NGWF optimization iteration. We compare our implementations to the hand-written TINTL interpolation routine [4]. Finally, we analyze the vector instruction throughput. We show results for both the *split complex* and *interleaved complex* data format.

Metric. Performance is given in *Pseudo Gflop/s* which is computed as $5N\log_2 N$ for an 1D FFT of size N (N an arbitrary positive integer). This leads to an operation count of $80N^3\log_2 N + 32N^3$ for the interpolation operation. This is a performance measure that essentially compares inverse runtime but gives an indication of the performance one sees, and is a standard measure in FFT benchmarking [14].

Interpolation performance. First we investigate the performance of the full interpolation kernel in isolation. In Figs. 5 (a) and (b) we compare across both data formats and two machines. We see that the SPIRAL-generated upsampling kernel for both machines and data formats for most data points outperforms the FFTW and MKL based implementations by a large margin (typically a factor of 2 to 3 except for some large prime numbers). The gain is more pronounced for the interleaved data format. This format is more demanding for optimization as either complex multiplications within a vector are required or data needs to be de- and re-interleaved multiple times. In contrast, the split format is relatively easier to optimize.

FFT performance. In Figs. 5 (c) and (d) we investigate the performance of the underlying 1D FFTs. We compare the *best fit* performance (timing them with hot cache) of 1D FFTs provided by FFTW and MKL to the SPIRAL-generated FFTs. For interleaved input we see that SPIRAL-generated FFTs outperform both FFTW and MKL considerably for most of the sizes. However, for small sizes and numbers with small prime factors, FFTW and MKL are well-optimized and there our generated code is slightly slower that MKL. For the split input format SPIRAL-generated code outperforms both MKL and FFTW for all but 2 sizes, and most of the times the performance gain is between 50% and 2x. Finally, we see that both FFTW and MKL contains highly optimized kernels for small prime sizes and a inefficient catch-all implementation for the unoptimized sizes. SPIRAL's gain stem from providing highly optimized kernels for all sizes.

An interesting observation is that the FFT and MKL FFT performance has very high spikes for 1D problem sizes that have a simple prime factorization. However, the corresponding 3D interpolation performance is much less spiky. In particular, while the 1D FFTs of FFTW and MKL occasionally outperform the 1D SPIRAL-generated FFTs, the SPIRAL-generated interpolation kernels outperform the handwritten interpolation powered by FFTW and MKL. This shows the impact of the data layout transformations and the loop merging we performed to optimize the 3D upsampling.

Performance portability. Fig. 5 show that our auto-tuning approach provides highest performance across multiple machines. The code run on each machine is specifically tuned to the machine's microarchitecture and memory hierarchy. We would observe a substantial drop in performance if we used the same code on all machines.

Impact on ONETEP. Finally, we evaluate how our optimized kernels perform inside ONETEP for interpolation sizes of practical relevance. For our test problem, we use an 181-atom "tennis-ball" molecule that is used as a model of the ligand binding cavity of a protein. We vary cutoff energies in order to change the resolution of the grid used to describe the NGWFs and influence FFT box size. Due to time constraints, the number of iterations in the NGWF optimization inner and outer loops were limited to 1. Fourier interpolation is used in both outer and inner loops, but 5-10 inner iterations is more typical of a simulation.

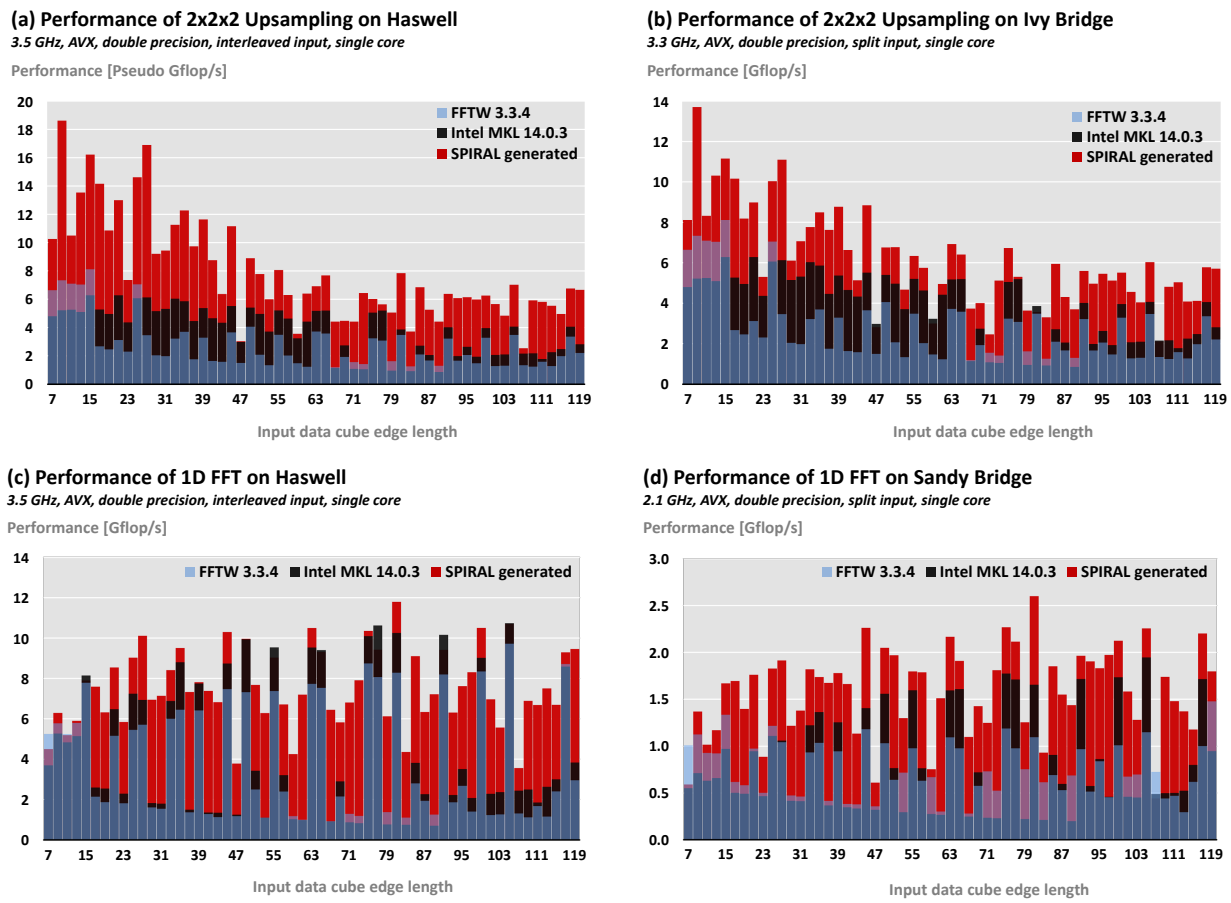


Fig. 5: Performance of 3D upsampling and 1D FFTs on a range of CPUs, for both split and interleaved format. We use purple bars for sizes where FFTW outperforms Intel MKL and dark blue bars for the minimum of FFTW and MKL (typically FFTW).

Fig. 6 shows runtime results for ONETEP's NGWF optimization running on 4 cores with OpenMP, where FFT runtime accounts for approximately 50% of the runtime. ONETEP supports OpenMP and MPI parallelisation, enabling it to scale to thousands of cores [15]. However, for interpolation, we are primarily concerned with intra-node performance optimizations. We compare unmodified ONETEP and two variants of ONETEP modified to use TINTL [4]. TINTL using MKL chooses the best-performing interpolation implementation of three (Sec. II-B), all of which are hand-written and depend on the performance of underlying FFTs. We see that the Intel MKL-based interpolation approach using TINTL [4] outperforms baseline ONETEP by 20% to 30%. TINTL+SPIRAL-generated code is on equal with ONETEP+TINTL for smaller sizes and for larger sizes we see an additional 10% to 15% runtime reduction. Since each benchmark problem uses a specific FFT-box size, overall speed-up is correlated with the upsampling performance obtained for that particular problem size.

Vector instruction throughput. In Fig. 7 we show the throughput of our generated interpolation kernel. We show floating-point throughput (vector adds and vector multiplies per cycle) and vector shuffles (shuffle instructions and load/store operations that require extra shuffles, as used for partial loads and stores). We see that our code sustains between 0.6 and 1.2 vector flop/cycle (2.4 to 4.8 flop/cycle), and there is an overhead of about 0.2 to 0.5 vector shuffles/cycle. This shows the high efficiency of our algorithm, vectorization, and generated code.

VIII. Related Work

Our algorithm is based on the time shift-based interpolation algorithm developed for ONETEP [3], [4]. The memory locality optimizations are inspired by distributed memory FFT libraries that use the slab decomposition [14]. Our vectorization approach extends previous SIMD vectorization inside SPIRAL [11], [1] but adds support for prime numbers and problem sizes that need looped kernels. FFTW's `genfft` [13] auto-generate SIMD vectorized FFT kernels but does not handle zero-padding and global cross-stage optimization. Intel's MKL and IBM's ESSL provide high performance FFT implementation that supports many sizes, albeit at varying performance levels. Other formal specification approaches for numerical algorithms include FLAME [16] and TCE [17]. Important auto-tuning libraries for numerical kernels include ATLAS [18] and OSKI [19].

IX. Conclusions

In this paper we presented the scope of effort necessary to optimize a mathematical kernel function for which for the most part highly optimized libraries exist, but where a small part of the algorithm must be implemented by hand. Upsampling

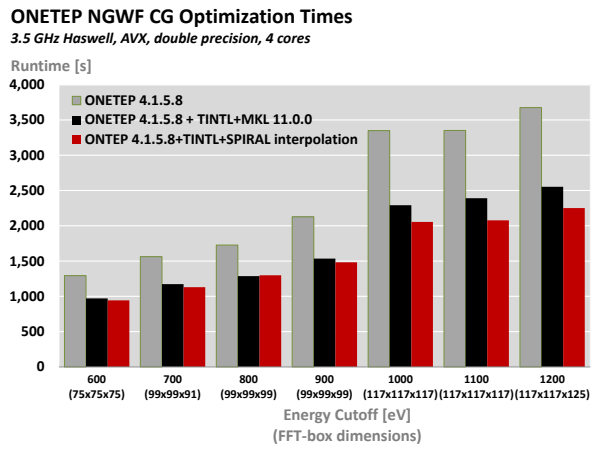


Fig. 6: Runtime of unmodified ONETEP, ONETEP using TINTL [4] with IMKL based hand-written interpolation kernels and ONETEP using TINTL with SPIRAL-generated interpolation kernels. Different cutoff energies are used resulting in a range of FFT box sizes.

of a 3D data cube of odd edge lengths through a frequency-domain method as required by the quantum chemistry package ONETEP is an example of a kernel that is unusual in many respects (odd size, small size, small upsampling factor). This translates into a lack of optimization in standard libraries. In addition, many necessary optimizations cannot be realized by optimizing FFTs but only when the whole operation optimized. We demonstrate that through data layout transformations and SIMD vectorization tricks a speedup of up to 6x (2x on average) over the best other available code (that uses FFTW and Intel's MKL) can be obtained for the kernel in isolation. Inside ONETEP, this translates into up to 15% speed-up.

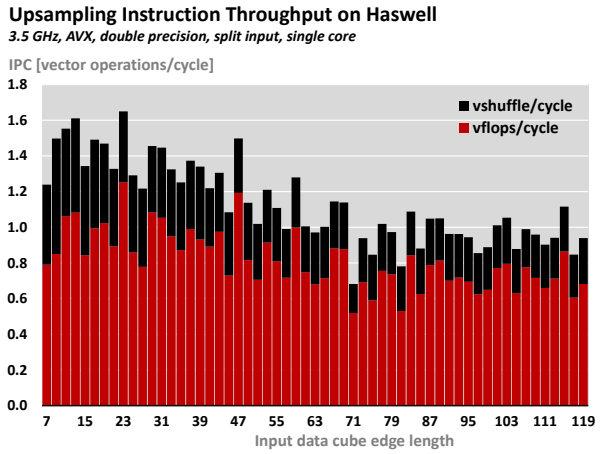


Fig. 7: Vector floating-point throughput and shuffle overhead.

X. Acknowledgement

This work was sponsored by EPSRC under grants EP/I006761/1, EP/I00677X/1 and EP/I006613/1, NSF through award 1116802 and DARPA under agreement HR0011-13-2-0007. The content, views and conclusions presented in this document do not necessarily reflect the position or the policy of DARPA.

References

[1] F. Franchetti and M. Püschel, "SIMD vectorization of non-two-power sized FFTs," in *International Conference on Acoustics, Speech, and Signal Processing (ICASSP)*, vol. 2, 2007, pp. II–17.

[2] L. P. Yaroslavsky, "Efficient algorithm for discrete sinc interpolation," *Appl. Opt.*, vol. 36, no. 2, pp. 460–463, Jan. 1997.

[3] C.-K. Skylaris, P. D. Haynes, A. A. Mostofi, and M. C. Payne, "Introducing ONETEP: Linear-scaling density functional simulations on parallel computers," *J. Chem. Phys.*, vol. 122, no. 8, p. 084119, 2005.

[4] F. P. Russell, K. Wilkinson, P. H. J. Kelly, and C.-K. Skylaris, "Optimised three-dimensional fourier interpolation: An analysis of techniques and application to a linear-scaling density functional theory code," *Computer Physics Communications*, vol. 187, pp. 8–19, 2015.

[5] M. Püschel *et al.*, "SPIRAL: Code generation for DSP transforms," *Proc. of the IEEE, special issue on "Program Generation, Optimization, and Adaptation"*, vol. 93, no. 2, pp. 232–275, 2005.

[6] W. Kohn, "Density functional and density matrix method scaling linearly with the number of atoms," *Phys. Rev. Lett.*, vol. 76, pp. 3168–3171, Apr. 1996.

[7] N. Dugan, L. Genovese, and S. Goedecker, "A customized 3D GPU poisson solver for free boundary conditions," *Computer Physics Communications*, vol. 184, no. 8, pp. 1815–1820, 2013.

[8] C. V. Loan, *Computational Framework of the Fast Fourier Transform*. SIAM, 1992.

[9] F. Franchetti, Y. Voronenko, and M. Püschel, "Loop merging for signal transforms," in *Proc. Programming Language Design and Implementation (PLDI)*, 2005, pp. 315–326.

[10] J. R. Johnson, R. W. Johnson, D. Rodriguez, and R. Tolimieri, "A methodology for designing, modifying, and implementing FFT algorithms on various architectures," *Circuits Systems Signal Processing*, vol. 9, pp. 449–500, 1990.

[11] F. Franchetti and M. Püschel, "Short vector code generation for the discrete Fourier transform," in *Proc. IEEE Int'l Parallel and Distributed Processing Symposium (IPDPS)*, 2003, pp. 58–67.

[12] F. Franchetti *et al.*, "Discrete Fourier transform on multicore," *IEEE Signal Processing Magazine, special issue on "Signal Processing on Platforms with Multiple Cores"*, vol. 26, no. 6, pp. 90–102, 2009.

[13] M. Frigo, "A fast Fourier transform compiler," in *Proc. ACM SIGPLAN conference on Programming Language Design and Implementation (PLDI)*, 1999, pp. 169–180.

[14] M. Frigo and S. G. Johnson, "The design and implementation of FFTW3," *Proc. of the IEEE, special issue on "Program Generation, Optimization, and Adaptation"*, vol. 93, no. 2, pp. 216–231, 2005.

[15] K. A. Wilkinson, N. D. M. Hine, and C.-K. Skylaris, "Hybrid MPI-openMP parallelism in the ONETEP linear-scaling electronic structure code: Application to the delamination of cellulose nanofibrils," *Journal of Chemical Theory and Computation*, vol. 10, no. 11, pp. 4782–4794, 2014.

[16] J. A. Gunnels, F. G. Gustavson, G. M. Henry, and R. A. van de Geijn, "FLAME: Formal linear algebra methods environment," *TOMS*, vol. 27, no. 4, pp. 422–455, Dec. 2001.

[17] G. Baumgartner *et al.*, "Synthesis of high-performance parallel programs for a class of ab initio quantum chemistry models," *Proc. of the IEEE*, vol. 93, no. 2, 2005, special issue on "Program Generation, Optimization, and Adaptation".

[18] R. C. Whaley and J. Dongarra, "Automatically Tuned Linear Algebra Software (ATLAS)," in *Proc. Supercomputing*, 1998, math-atlas.sourceforge.net.

[19] J. Demmel *et al.*, "Self adapting linear algebra algorithms and software," *Proc. of the IEEE*, vol. 93, no. 2, pp. 293–312, 2005, special issue on "Program Generation, Optimization, and Adaptation".