Computer Physics Communications 187 (2015) 8–19

Contents lists available at ScienceDirect

Computer Physics Communications

journal homepage: www.elsevier.com/locate/cpc

Optimised three-dimensional Fourier interpolation: An analysis of techniques and application to a linear-scaling density functional theory code

Francis P. Russell[a,*], Karl A. Wilkinson[b], Paul H.J. Kelly[a], Chris-Kriton Skylaris[b]

[a] *Department of Computing, Imperial College London, London, SW7 2AZ, UK*

[b] *School of Chemistry, University of Southampton, Southampton, SO17 1BJ, UK*

ARTICLE INFO

Article history:
Received 7 April 2014
Received in revised form
25 September 2014
Accepted 29 September 2014
Available online 7 October 2014

Keywords:
Trigonometric interpolation
Fast Fourier Transform
Density functional theory
ONETEP

ABSTRACT

The Fourier interpolation of 3D data-sets is a performance critical operation in many fields, including certain forms of image processing and density functional theory (DFT) quantum chemistry codes based on plane wave basis sets, to which this paper is targeted. In this paper we describe three different algorithms for performing this operation built from standard discrete Fourier transform operations, and derive theoretical operation counts. The algorithms compared consist of the most straightforward implementation and two that exploit techniques such as phase-shifts and knowledge of zero padding to reduce computational cost. Through a library implementation (TINTL) we explore the performance characteristics of these algorithms and the performance impact of different implementation choices on actual hardware. We present comparisons within the linear-scaling DFT code ONETEP where we replace the existing interpolation implementation with our library implementation configured to choose the most efficient algorithm. Within the ONETEP Fourier interpolation stages, we demonstrate speed-ups of over 1.55×.

1. Introduction

Fourier interpolation of discretely sampled data has a wide variety of uses across various domains—these include postprocessing of magnetic resonance imaging data [1], quantum-mechanical calculations using density functional theory [2] (DFT—not to be confused with the discrete Fourier transform which we do not abbreviate in this paper) and imaging for synthetic aperture radar [3].

Many DFT codes are based upon plane-waves and, as a result, depend on efficient FFTs and Fourier interpolation. A number of developments have been implemented to improve the efficiency of Fourier transform algorithms within plane-wave codes.

In many cases, entire rows or columns of the region being transformed may be zero-valued; this can occur for many reasons including the inputs representing highly localised functions [4], in order to support certain boundary conditions [5] or the need to expand quantities in plane waves with a higher cutoff frequency [6]. Since three-dimensional Fourier transforms can be expressed as a sequence of one-dimensional ones, it is possible to omit some transforms entirely when they operate only on zeros, enabling higher performance.

When performing a three-dimensional transform of a grid containing a sphere of non-zero values, PARATEC avoids transforms on the zero-valued rectangular regions around a sphere of non-zeros as it delocalises in each dimension [4]. The CPMD software package uses this property for transforms of the local potential and charge density between real and reciprocal space [6]. Dugan et al. [5] also perform this optimisation for a three-dimensional Poisson solver in the BigDFT package. This technique is used in ONETEP in the existing Fourier interpolation implementation, and is the one we compare against.

In some instances, parts of the Fourier transform output may not be required. Goedecker et al. describe an algorithm for maximising the communication–computation overlap of a three-dimensional Fast Fourier Transform in which the upper-half of the frequency spectrum is discarded in all three dimensions [7]. The development of efficient parallel FFTs for use in DFT calculations have also been addressed elsewhere [8]. The nature of the parallelism within ONETEP (1 core per FFT operation) contrasts with those in plane-wave codes (many cores per FFT operation)

* Corresponding author. Tel.: +44 7900 647 895.
E-mail addresses: francis.russell02@imperial.ac.uk (F.P. Russell), k.wilkinson@soton.ac.uk (K.A. Wilkinson), p.kelly@imperial.ac.uk (P.H.J. Kelly), c.skylaris@soton.ac.uk (C.-K. Skylaris).

http://dx.doi.org/10.1016/j.cpc.2014.09.019
0010-4655/© 2014 The Authors. Published by Elsevier B.V. This is an open access article under the CC BY license (http://creativecommons.org/licenses/by/3.0/).

meaning that the issue of communication of data between cores is not relevant in this work.

Pruned FFTs [1,9] reduce computational cost by omitting unnecessary calculation on zeros in the input and avoiding calculation of unused outputs. They are so-named since the unnecessary expressions are "pruned" from the butterfly graph of the FFT. Pruned FFTs have been used to reduce the cost of interpolating image data [1]. Pruned FFTs require a custom implementation that can be specialised to the zero/non-zero pattern of the input data and the pattern of needed output values. In order for pruned implementations to provide competitive performance, they must be optimised to a similar extent as available FFT libraries. Efficient code generation for these transforms has been explored [9], achieving modest speed-ups over vendor libraries.

The Fourier interpolation algorithm is necessary in many DFT codes such as CASTEP [10], VASP [11], QUANTUM ESPRESSO [12] CPMD [13] and ABINIT [14]. It is also used within the ONETEP [15] (Order-N Electronic Total Energy Package) linear-scaling code for quantum-mechanical calculations based on DFT, which we use as a case study here.

In this paper, we explore three algorithms for interpolating a signal sampled over a regular three-dimensional grid, although these techniques are also applicable to signals sampled over different numbers of dimensions. All three algorithms make use of existing Fast Fourier Transform (FFT) implementations, enabling existing and future optimisations in these routines to be leveraged. The algorithms we analyse have different operation counts. We present derivations of these operation counts and stand-alone benchmarks in order to analyse the relationship between operation count and actual performance. Within each algorithm choice, we also explore different implementation variations. These variations do not change the operation count, but may result in different performance characteristics over different problem sizes and architectures.

To demonstrate the utility of our techniques, we benchmark their effect within the ONETEP code. A central quantity in DFT is the *charge density*. ONETEP performs Fourier interpolation during construction of the charge density [2], which forms one of the most performance critical operations during the DFT calculation [16].

In Section 2 we provide an outline of the ONETEP theory, focusing on the parts that use Fourier interpolation. In Section 3 we discuss the Fourier interpolation algorithms and their implementation on different architectures and platforms. Section 4 contains results and discussion of synthetic benchmarks whilst Section 5 contains the results obtained from our implementation of the algorithms within ONETEP. Finally, we present our conclusions in Section 6.

2. ONETEP

ONETEP [15] (Order-N Electronic Total Energy Package) is a linear-scaling quantum chemistry software package for DFT calculations. The linear-scaling of computational cost with respect to the number of atoms within ONETEP is achieved through the exploitation of the "nearsightedness of electronic matter" principle [17,18]. The theoretical details of the ONETEP methodology are discussed in detail elsewhere [15] and are only summarised here. The ONETEP program is based on a reformulation of DFT in terms of the one-particle density matrix, $\rho\left(\mathbf{r},\mathbf{r}'\right)$, which is the basis of many linear-scaling DFT approaches [19] where the memory and processing requirements increase linearly with N. This is achieved by taking advantage of the exponential decay of the density matrix in systems with a band gap.

In ONETEP the density matrix is expressed in the following form:

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

where the "density kernel" K is the density matrix expressed in the duals of $\{\phi_{\alpha}\left(\mathbf{r}\right)\}$, the set of non-orthogonal generalised Wannier functions (NGWFs) [20]. The NGWFs are constrained to be strictly localised within spherical regions centred on atoms and their shape is optimised self-consistently by expressing them in a psinc basis set [21,22].

Psinc functions are centred on the points of a regular real-space grid and are related to a plane-wave basis through Fourier transforms. As a result ONETEP is able to achieve linear-scaling computational cost whilst retaining the large basis set accuracy characteristics of plane-wave codes [23].

2.1. Rationale for Fourier interpolation in ONETEP

In order to maintain numerical stability it is important to only perform operations that are compatible with the psinc basis set which is connected to plane waves via a unitary transformation. For example, we have shown in the past [24], that calculation of the kinetic energy by using finite differences is equivalent to an arbitrary switch to a different basis set (e.g. polynomials) and limits the accuracy of the calculation [24]. The same applies for the computation of the electronic density and local potential integrals which we are examining in this work in terms of their requirements for Fourier interpolation. Thus, quantities such as the kinetic energy operator and the Hartree potential have to be calculated in reciprocal space but the products of local orbitals for building the electronic density and local potential matrix elements have to be done with Fourier interpolation. In a technical context, a significant difference between ONETEP and plane-wave codes is the size of the datasets: The localisation of the NGWFs in ONETEP means that FFT operations are performed over relatively small datasets whilst FFT operations in plane wave codes are performed over the entire simulation cell. This difference means that it is feasible to perform atom-localised FFTs in ONETEP on a single core, removing the overheads associated with parallel FFTs from these operations. Parallel FFTs are also used within ONETEP, but only for the calculation of the Hartree potential which is not discussed here.

2.2. Use of Fourier interpolation in ONETEP

The form of the electronic energy in ONETEP is:

$$\begin{aligned}E=&\sum_{\alpha\beta}K^{\alpha\beta}\Big[\langle\phi_{\beta}|\hat{T}|\phi_{\alpha}\rangle+\langle\phi_{\beta}|\hat{V}_{\mathrm{ps,loc}}|\phi_{\alpha}\rangle\\&+\langle\phi_{\beta}|\hat{V}_{\mathrm{ps,nonloc}}|\phi_{\alpha}\rangle\Big]+E_{\mathrm{H}}[n]+E_{xc}[n]\end{aligned} \quad (2)$$

where the first three terms contain the kinetic integrals, local pseudopotential integrals and non-local pseudopotential integrals respectively, and the next two terms are the Hartree and exchange–correlation energy functionals respectively [20]. We can clearly see the explicit dependence of the electronic energy on the density kernel, K, and NGWFs $\{\phi_{\alpha}\left(\mathbf{r}\right)\}$ and, most importantly, the electronic density $n(\mathbf{r})$. The calculation of the electronic energy within ONETEP takes the form of two nested loops: the density kernel, and NGWFs are optimised within the inner and outer loops respectively [25].

In order to perform operations involving NGWFs, the FFT box technique is used [24]. An FFT box is a box of grid points centred on the atom associated with an NGWF and large enough to contain any overlapping NGWF localisation spheres in their entirety. This representation permits the use of plane-wave methodology to perform momentum space operations with a computational cost that is independent of the size of the simulation cell (i.e. the number of FFT box operations performed in a calculation is proportional to the number of NGWFs, the exact value being determined by the sparsity of the density kernel).

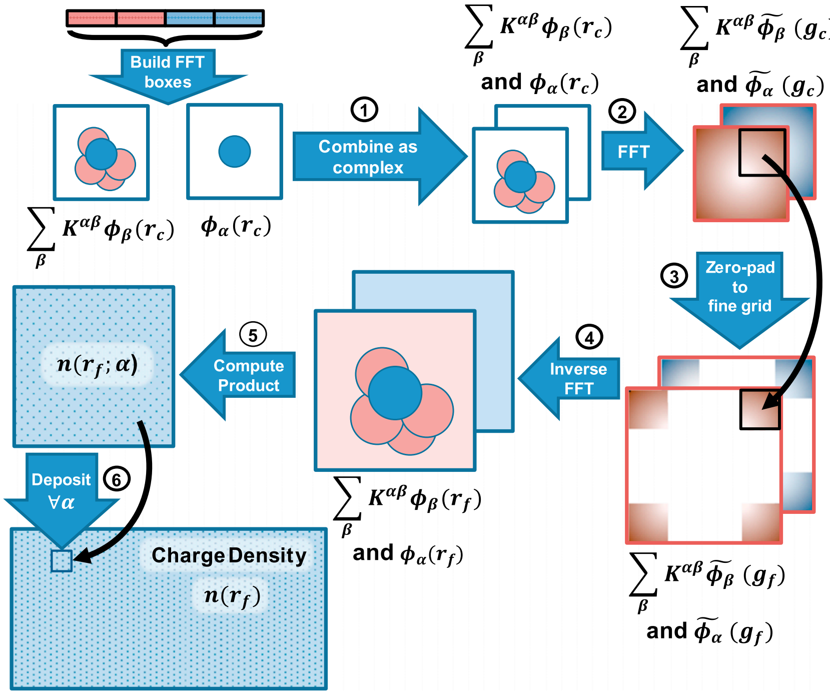
Build FFT boxes
$\sum_\beta K^{\alpha\beta}\phi_\beta(r_c)$
$\phi_\alpha(r_c)$
1 Combine as complex
$\sum_\beta K^{\alpha\beta}\phi_\beta(r_c)$ and $\phi_\alpha(r_c)$
2 FFT
$\sum_\beta K^{\alpha\beta}\widetilde{\phi_\beta}(g_c)$ and $\widetilde{\phi_\alpha}(g_c)$
3 Zero-pad to fine grid
$\sum_\beta K^{\alpha\beta}\widetilde{\phi_\beta}(g_f)$ and $\widetilde{\phi_\alpha}(g_f)$
4 Inverse FFT
$\sum_\beta K^{\alpha\beta}\phi_\beta(r_f)$ and $\phi_\alpha(r_f)$
5 Compute Product
$n(r_f;\alpha)$
Deposit $\forall\alpha$ 6
Charge Density
$n(r_f)$

Fig. 1. Operations performed for the calculation of the charge density in ONETEP. FFT box sizes roughly correspond to data-sizes rather than physical relationships. We use the expressions r_c, r_f, g_c and g_f to denote vectors on the coarse-real, fine-real, coarse-reciprocal and fine-reciprocal grids, respectively. Step (1) combines both real-valued FFT boxes into a complex one; this eases implementation but has no mathematical impact. Steps (2)–(4) correspond to the "padding-aware" technique of Fourier interpolation (Section 3.1.2). Step (5) computes the product $\sum_\beta K^{\alpha\beta}\phi_\beta(r)\phi_\alpha(r)$ which due to the interpolation, avoids aliasing. Step (6) accumulates the α-specific contribution into the charge density (which is distributed across nodes). The steps are repeated for each α to construct the full charge density.

The FFT box operations [24] that utilise the Fourier interpolation algorithm are used in the calculation of quantities such as the charge density and local potential. The calculation of these quantities contribute significantly to the bottlenecks within ONETEP; the exact fraction of runtime depends upon the method used and the scale of the calculation and can range from 60% to 90%.

The charge density $n(\mathbf{r})$, the central quantity in DFT, is given by the diagonal elements of the density matrix

$$n(\mathbf{r}) = \rho(\mathbf{r},\mathbf{r}) = \sum_{\alpha\beta}\phi_\alpha(\mathbf{r})K^{\alpha\beta}\phi_\beta(\mathbf{r}). \quad (3)$$

The local pseudopotential contribution to the energy, is calculated in terms of local potential integrals $\langle\phi_\alpha|\hat{V}_{\mathrm{ps,loc}}|\phi_\beta\rangle$ which, due to the orthogonality and cardinality of the psinc basis are simply the dot product of psinc functions common to both $\phi_\alpha(\mathbf{r})$ and $\hat{V}_{\mathrm{ps,loc}}\phi_\beta(\mathbf{r})$. The computational algorithms used to calculate the charge density and local potential integrals are illustrated in Figs. 1 and 2, respectively, and are discussed in detail in reference [16].

Typically, an algorithm performing operations upon FFT boxes contains these steps:

1. The real-space FFT box is transformed to reciprocal space.
2. The reciprocal space FFT box is "upsampled" to a fine grid. This doubles the number of points in each dimension of the box. In reciprocal space this corresponds to adding new zero-valued components.
3. An operator may be applied to the fine grid.
4. A second FFT is performed to return the FFT box to a real space representation. By construction, most values on the reciprocal space fine grid are zero. The techniques discussed in this paper exploit this to reduce computational cost.

The case where no operator is applied corresponds to an "upsampling" of the real-space FFT box to a finer representation. This operation is used in the construction of the electronic density, which is a computationally intensive part of ONETEP. The process is necessary as ONETEP performs point-wise multiplications between FFT boxes during all calculations and the interpolation process serves to prevent aliasing errors [2].

The ONETEP subroutine `fourier_interpolate` performs the interpolation of an FFT box (actually two due to implementation details) and is used during the calculation of quantities such as the local potential integrals. In the computation of the charge density, two interpolated FFT boxes are immediately subject to a point-wise multiply. The ONETEP routine `fourier_interpolate_product` implements this operation.

We only explore techniques for the "upsampling" of a real-space representation since these can be easily isolated from the rest of the ONETEP code and represent a significant portion of ONETEP's execution time. However, all techniques we explore still involve computing a reciprocal-space representation of the FFT box and therefore could be used to apply operators in reciprocal space. For the "phase-shift" approach, we would require that the operator is applicable point-wise to the grid and is linearly separable. This is the case for all operators used in ONETEP.

3. Adaptation to hardware

ONETEP is capable of running on a number of different architectures and platforms. Obviously, different platforms have different performance characteristics which means code optimised for one platform may work sub-optimally on another. In addition, ONETEP makes use of libraries for linear algebra and Fourier transform operations that may have different performance characteristics.

Here, we explore the factors that affect how the FFT box operations in ONETEP perform on different platforms and develop a mechanism whereby ONETEP can adapt the implementations it uses for improved performance across multiple platforms.

We have focused our analysis on the "upsampling" operation (Fourier interpolation) described previously. This operation takes a signal discretised on a regular three-dimensional grid and using

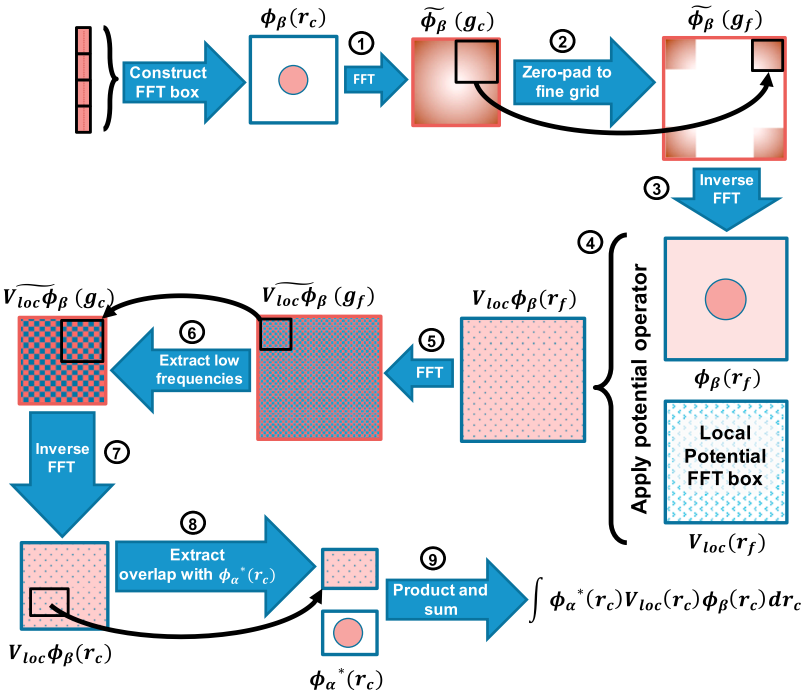

Fig. 2. Operations performed for the calculation of local potential integrals in ONETEP. FFT box sizes roughly correspond to data-sizes rather than physical relationships. We use the expressions r_c, r_f, g_c and g_f to denote vectors on the coarse-real, fine-real, coarse-reciprocal and fine-reciprocal grids, respectively. Steps (1)–(3) correspond to the "padding-aware" technique of Fourier interpolation (Section 3.1.2). Step (4) applies the local potential operator to $\phi_\beta(r_f)$ through a product with local potential operator in the form of an FFT box. Steps (5)–(7) transfer this result back to the coarse-grid by discarding higher-frequency components. Steps (8) and (9) extract the subpart of the result which overlaps with $\phi_\alpha^*(r_c)$ and use it to compute the final integral.

Fourier interpolation, doubles the number of sample points in all three dimensions.

In a manner similar to the "plan" step used by many FFT libraries, we adapt to different hardware through the evaluation of different implementation techniques at runtime. This allows us to choose the best-performing implementation for a particular target.

The implementation choices we explore for the upsampling operation fall into two different categories:

Algorithmic These consist of variations to the algorithm choice we use to perform the upsampling. The different algorithms we explore have different operation counts. Since operation count is not a direct predictor of performance, the best choice of algorithm may vary across platforms.

Platform-dependent These consist of implementation variations that do not change the underlying algorithm. They may exploit different performance properties of the underlying hardware and libraries, but do not change the number of operations that need to be performed.

We now describe three algorithms that can be used to perform the upsampling operation. All produce the same result, but employ Fourier transforms in different ways. Each have different operation counts but due to platform-specific factors, these do not directly correlate with performance.

3.1. Upsampling algorithms

The trigonometric interpolation of a signal is often referred to as "discrete sinc interpolation" in the literature [26,27]. This is due to the fact that interpolation of a signal via convolution with the sinc function is equivalent to interpolation via the Fourier series [26].

We have evaluated three different algorithmic choices for implementing upsampling in ONETEP. The first is the most commonly used approach to trigonometric interpolation, the other two reduce computation cost by omitting calculations on values known to be zero.

Although it is possible to interpolate a signal of size n to any size $m \geq n$, our requirements for standard ONETEP calculations are that $m = 2n$, so our operation count derivations and benchmark results are for this case. We also note that the "phase-shift" approach (Section 3.1.3) can only generalise to the case where m is an integer multiple of n whereas the other two techniques are applicable to any case where $m \geq n$.

The algorithms we present make use of the discrete Fourier transforms as a building block, enabling implementations to exploit the extensive performance optimisation that both vendor-supplied and third party libraries have been subject to.

This work shares the commonality that it avoids unnecessary computation on input or production of output from a multi-dimensional FFT by decomposing into one-dimensional FFTs and discarding unnecessary work and/or data. However, it is often not possible to apply these savings in every dimension, as to do so would require a customised Fourier transform implementation. We examine how it is possible to perform a Fourier interpolation operation with the computation and production of no redundant data whatsoever using only standard Fourier transform implementations. In doing so, we hope to bridge the gap towards achieving maximum possible hardware performance.

3.1.1. Naïve interpolation

This technique relies on zero-padding in the frequency domain to add new zero-amplitude higher frequency components. Such a strategy is described by Yaroslavsky [27] as the "commonly used method". We describe the steps for a 1D input signal of n samples:

1. Perform a Fourier transform to convert the input signal to its frequency domain representation.

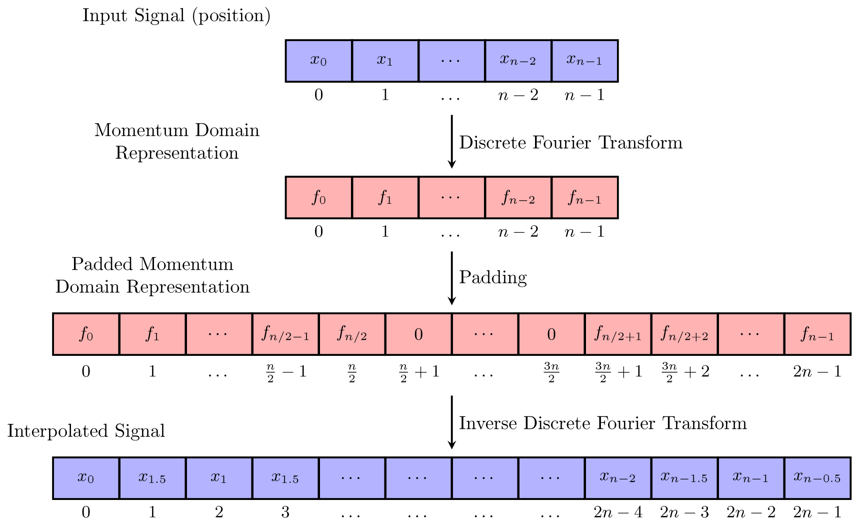
Input Signal (position)
x_0 x_1 ... x_{n-2} x_{n-1}
0 1 ... $n-2$ $n-1$
Momentum Domain Representation
Discrete Fourier Transform
f_0 f_1 ... f_{n-2} f_{n-1}
Padded Momentum Domain Representation
Padding
f_0 f_1 ... $f_{n/2-1}$ $f_{n/2}$ 0 ... 0 $f_{n/2+1}$ $f_{n/2+2}$ ... f_{n-1}
0 1 ... $\frac{n}{2}-1$ $\frac{n}{2}$ $\frac{n}{2}+1$ ... $\frac{3n}{2}$ $\frac{3n}{2}+1$ $\frac{3n}{2}+2$ ... $2n-1$
Inverse Discrete Fourier Transform
Interpolated Signal
x_0 $x_{1.5}$ x_1 $x_{1.5}$ ... ... x_{n-2} $x_{n-1.5}$ x_{n-1} $x_{n-0.5}$
0 1 2 3 ... ... $2n-4$ $2n-3$ $2n-2$ $2n-1$

Fig. 3. Application of $2\times$ upsampling to a 1D input signal with an odd number of samples. The even case is similar, but requires special handling of the coefficient $f_{n/2}$ (Nyquist frequency component).

2. Zeros are inserted into the frequency representation to add new zero-amplitude high frequency components. A Discrete Fourier Transform produces both *positive* and *negative* frequencies which must both be augmented. For $2\times$ upsampling, the frequency domain representation now has $2n$ components. The addition of these zeros is typically called "padding".
3. An inverse Fourier transform is applied to the padded signal, giving an output signal of $2n$ samples.

We illustrate this algorithm for odd numbers of samples in Fig. 3. The frequency components $f_{n/2}$ to f_{n-1} of the transform are in fact the *negative* frequencies produced by the Fourier transform arranged in reverse order. Consequently, the padding step which augments both positive and negative frequencies with zeros corresponds to inserting zeros into the middle of the frequency representation.

In the case when n is even, a single coefficient ($f_{n/2}$) represents both the positive and negative Nyquist frequency. In this case, the padding step duplicates this component with half its magnitude on either side of the zero padding. Understanding of this detail is not necessary for comprehension of these techniques, but we include it for completeness.

The one-dimensional algorithm generalises to the multidimensional case as is illustrated for the 3D case in Fig. 4.

In the three-dimensional case, the inverse FFT step operates on input data that contains 87.5% zeros due to the introduced padding. The next two techniques aim to improve efficiency by avoiding numerical operations performed on the padding values.

3.1.2. Padding-aware interpolation

The interpolation technique we call "padding-aware" is the currently employed method for performing three-dimensional interpolation in ONETEP. A similar strategy has been explored by Dugan et al. [5] for three-dimensional Poisson solvers. Like the "naïve" approach it relies on zero-padding in the frequency domain. However the computation cost is reduced through the following two observations:

1. The multi-dimensional Fourier transform (and its inverse) is *separable* (i.e. a multi-dimensional Fourier transform can be expressed in terms of a number of one-dimensional transforms).

   Specifically, a discrete Fourier transform must be performed for each one-dimensional slice through the dataset in each dimension. For a three-dimensional dataset of dimensions $n \times n \times n$, n^2 one-dimensional transforms (the area of a face)

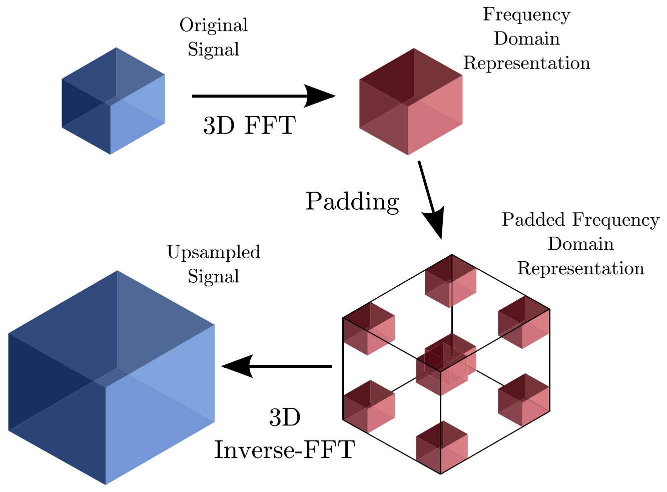
Original Signal
3D FFT
Frequency Domain Representation
Padding
Padded Frequency Domain Representation
Upsampled Signal
3D Inverse-FFT

Fig. 4. Transforms and data movement for the naïve approach (Section 3.1.1) to Fourier interpolation in the three-dimensional case. Transparent regions denote zero-valued coefficients. The input data-set is transformed to the frequency domain, padded with zero-magnitude frequency components, then transformed back using a single three-dimensional inverse Fourier Transform. The input to the inverse transform contains 87.5% zeros.

   must be applied over three faces. This corresponds to $3n^2$ one-dimensional Fourier transforms of size n to implement the three-dimensional one.
2. When the three-dimensional inverse-FFT of the naïve approach is expressed in terms of one-dimensional inverse-Fourier transforms, many of these transforms operate on inputs which are entirely zero. As the output of such a transform is also entirely zero, there is no need to perform these transforms.

The "padding-aware" approach performs the interpolation in the same manner as the "naïve" approach except for the final inverse three-dimensional Fourier transform. This step is performed using one-dimensional inverse Fourier transforms, applied in a per-dimension manner. We illustrate this technique in Fig. 5.

The inverse transform is applied to a dataset of dimensions $2n \times 2n \times 2n$. Inverting an arbitrary input would require $4n^2$ one-dimensional transforms of size $2n$ in each dimension, resulting in $12n^2$ transforms in total.

However, exploiting knowledge of padding, the number of transforms that need to be performed in each dimension becomes

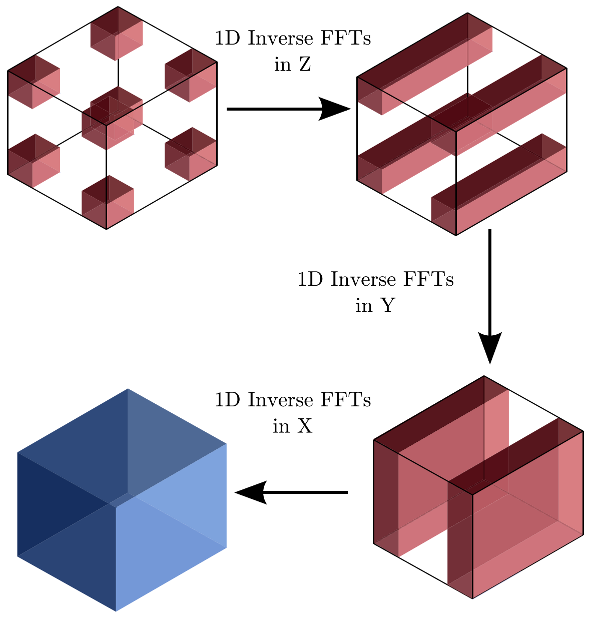

Fig. 5. Transforms and data movement for "padding-aware" interpolation (Section 3.1.2). Transparent regions denote zero-valued coefficients. The "padding-aware" algorithm transforms input data to the frequency domain and pads it in the same manner as the "naïve" algorithm. We show only the backward transform in which the three-dimensional inverse Fourier transform is implemented using one-dimensional inverse Fourier transforms, applied in a per-dimension manner. Transforms on entirely zero one-dimensional slices through the dataset can be omitted entirely. The remaining one-dimensional transforms operate on inputs containing 50% zeros.

different:

1. In the z dimension, only $1/4$ of the $4n^2$ slices through the z face contain non-zero values. The slices containing non-zero values are located on square shaped regions of area $n^2/4$ at the corner of each face. Hence, the inverse transform in this dimension only requires n^2 one-dimensional inverse FFTs to compute.
2. In the y dimension, $1/2$ of the $4n^2$ slices through the y face contain non-zero values. The slices containing non-zero values are located on two rectangular shaped regions of area n^2 at the highest and lowest values of x. The inverse transform in this dimension requires $2n^2$ one-dimensional inverse FFTs.
3. In the x dimension, all the $4n^2$ slices through the face contain non-zero values. The inverse transform in this dimension requires $4n^2$ one-dimensional inverse FFTs and so no saving is made by exploiting sparsity.

This technique can be considered as a form of pruning (Section 3.1) that does not require a pruned FFT implementation. However, it cannot reduce operation costs in the one-dimensional case nor reduce costs when the underlying one-dimensional transforms operate on a mixture of zero and non-zero values.

To implement the inverse transform, the "padding-aware" approach requires $7n^2$ one-dimensional inverse Fourier transforms in total as opposed to the $12n^2$ that would be required without exploiting padding.

This approach only exploits padding applied in different dimensions to the one being transformed. Consequently, this scheme cannot exploit padding in the one-dimensional case. Each one-dimensional transform still operates on 50% zeros, so it is apparent that it is possible to reduce computation further.

3.1.3. *Phase-shift interpolation*

The final algorithm we explored uses Fourier transforms in such a way as to avoid ever operating on zeros introduced by padding. The algorithm for efficient sinc interpolation described by Yaroslavsky [27] closely resembles our "phase-shift" approach. Yaroslavsky's method uses shifted discrete Fourier transforms which can be parameterised by shifts in both the time and frequency domains. Yaroslavsky describes the algorithm in a more abstract sense whereas we explore concerns that affect efficient implementation and analyse and compare performance for concrete implementations of these algorithms on a real-world application.

We describe our technique for the one-dimensional case first, then generalise to the multi-dimensional one. In Fig. 3 we can see that the inverse discrete Fourier transform step produces a signal whose odd numbered samples are identical to the original input. Since we possess the original input, there is no need to recalculate these values. We only need to calculate the values of the new interpolated samples.

In order to calculate the interpolated samples, we construct a new signal which consists of the original signal shifted in space by half a sample. The points at which this new signal is sampled are the midpoints between samples of the original signal. By interleaving the samples of these two signals, we produce the interpolated signal.

We illustrate this technique for the one-dimensional case in Fig. 6. The steps of interpolating a one-dimensional signal are as follows:

1. Perform a Discrete Fourier transform on the input signal.
2. Multiply each coefficient in the momentum representation by a pre-calculated phase shift. The multiplication applies a phase shift that shifts each frequency component by $1/2$ the sampling interval.

   For a signal with n samples, the phase shift for frequency f can be computed as $e^{if\pi/n}$.

   As before, the case where the input signal has an even number of samples requires special handling. Shifting the Nyquist frequency by half the sampling interval causes it to become zero at all sample points. Consequently, this coefficient is always set to zero.
3. An inverse Discrete Fourier transform is applied to the modified Fourier coefficients. The result corresponds to the values at the midpoints between the sample points of the original signal.
4. The original input signal and the midpoint values are interleaved to produce the interpolated signal.

As with the "naïve" approach, we can generalise by repeating the process in multiple dimensions. We illustrate this in Fig. 7. Interpolation in each dimension doubles the number of data points, until we have $8\times$ the number of original samples. We choose to interpolate in the least cache-efficient dimension first so that the most executed interpolations are more cache efficient.

3.1.4. *Operation count analysis*

We provide an analysis of the operation count of the three presented algorithms. We consider the relative operation counts for each when applied to a 3D grid with n points along each dimension. Forward and backward (inverse) Fourier transforms are assumed to have the same operation count.

We consider the cost of a one-dimensional Fourier transform of n points to have a cost of $n \log n$ operations. Our derived operation counts will have constant factors relative to the cost of single transforms.

A multi-dimensional Fourier transform can be expressed in terms of a set of 1D transforms applied to each pencil-shaped slice through the data-set in each dimension.

For a grid of size $n \times n \times n$, we perform n^2 transforms in each dimension, each having an operation count of $n \log n$. This gives an operation count of:

$$3n^2 n \log n \tag{4}$$

$$= 3n^3 \log n. \tag{5}$$

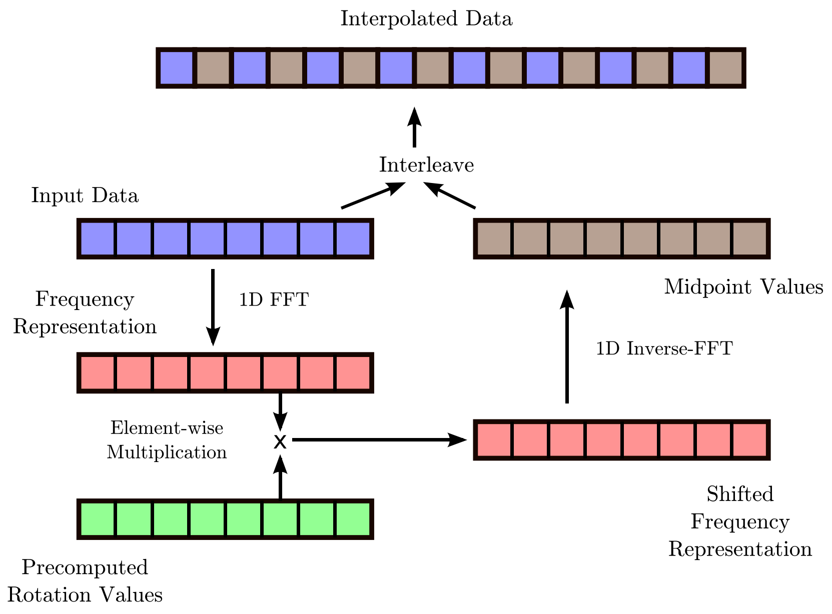

Fig. 6. The interpolation of a one-dimensional signal via the application of a phase shift (Section 3.1.3). The original signal is transformed via a correlation in the frequency domain to produce sample values at the midpoints of the original sampling locations. These are interleaved with the original signal to produce the interpolated signal. Zero padding is not required during any step.

We derive operation counts for each algorithm as follows:

Naïve Interpolation. This technique (Section 3.1.1) requires one forward three-dimensional Fourier transform of size n and one backward three-dimensional transform of size $2n$. This is shown for the one-dimensional case in Fig. 3 and the generalisation to the three-dimensional case in Fig. 4. The operation count can be derived as:

$$3n^3 \log n + 24n^3 \log 2n \tag{6}$$

$$= 27n^3 \log n + 24n^3 \log 2. \tag{7}$$

Padding-Aware Interpolation. This technique (Section 3.1.2) performs a 3D forward transform of size n. It then performs 1D backward transforms of size $2n$ in each dimension with counts of n^2, $2n^2$ and $4n^2$. The effect of the application of each set of transforms is shown in Fig. 5. The operation count can be derived as:

$$3n^3 \log n + (n^2 + 2n^2 + 4n^2)(2n \log 2n) \tag{8}$$

$$= 3n^3 \log n + 14n^3 \log 2n \tag{9}$$

$$= 17n^3 \log n + 14n^3 \log 2. \tag{10}$$

Phase-Shift Interpolation. This technique (Section 3.1.3) performs seven interpolations, each requiring n^2 1D convolutions. Each convolution consists of a forward and backward transform of size n, and a point-wise multiplication of size n. Fig. 6 illustrates application of the convolution to calculate the "midpoint values". Fig. 7 illustrates why seven interpolations are necessary for the three-dimensional case.

The convolutions incur an additional $7n^3$ multiplies for the entire interpolation. We multiply this term by the unknown factor c since we do not know the cost of these multiplies relative to the whole interpolation (though c will be ≤ 1). The total operation count of the Fourier transforms is:

$$7n^2(2n \log n + cn) \tag{11}$$

$$= 14n^3 \log n + 7cn^3. \tag{12}$$

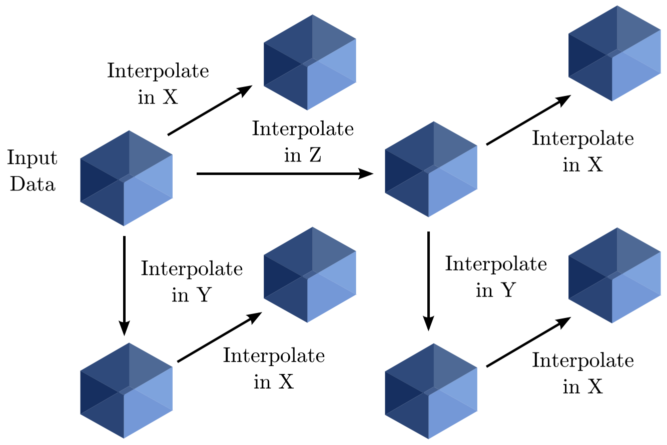

Fig. 7. Application of phase-shift interpolation (Section 3.1.3) in one dimension to compute sample points for a three-dimensional input signal. The process results in 8 versions of the input signal subjected to different phase shifts (one of which is the original). These are then interleaved to produce the final interpolated output.

We conclude that in terms of operation count, the phase-shift interpolation technique, should be most efficient for sufficiently large sizes of n. This is to be expected given that it never operates on zeros and therefore should not perform redundant work.

Operation counts allow us to compare the theoretical performance of each algorithm for large values of n. Assuming no other limiting factors, the padding-aware and phase-shift algorithms are capable of providing throughput improvements of $1.59\times\ (\frac{27}{17})$ and $1.93\times\ (\frac{27}{14})$ over the naïve algorithm, respectively. Over the padding-aware algorithm, the phase-shift approach may provide a $1.21\times\ (\frac{17}{14})$ improvement in throughput.

In practice, operation counts are not an effective predictor of performance for problems for similar magnitude. Hardware characteristics such as cache size and memory bandwidth significantly influence achievable performance. For this reason, we have explored adaptation as a mechanism by which we choose the most appropriate implementation for a given problem size, hardware platform and underlying FFT library.

3.2. Platform-dependent implementation choices

We have implemented the algorithms described in Section 3.1 in a small C library: TINTL (Trigonometric INTerpolation Library) [28]. For each algorithm, we have implemented different variations, intended to exploit the performance characteristics of different platforms.

3.2.1. Real versus complex interpolation

ONETEP performs interpolation on real-valued data-sets. However, it always performs the interpolation of two data-sets simultaneously. If one of the data-sets is multiplied by i and added to the other, a complex-valued data-set with the same number of elements is formed. Interpolation may be performed on this data-set and the real and imaginary parts of the result separated afterwards to retrieve the two interpolated real-valued data-sets.

This provides us with two implementation choices for the interpolation of two real-valued data-sets. We can either interpolate each one individually, or combine them to form a complex-valued data-set, interpolate that, then split to form the result.

The Fourier transform is inherently complex-to-complex, so interpolation on the combined data-set is much simpler to implement and requires only complex-to-complex transforms. Complex-to-complex transforms can easily be performed in-place, since the input and output consist of the same number of complex-valued elements.

Interpolation of a real-valued data-set requires complex-to-real, real-to-complex, and possibly complex-to-complex transforms (depending on the algorithm used). The real-to-complex and complex-to-real transforms typically have slightly differing input and output data sizes, preventing them from being performed in place.

Either strategy may have different performance behaviour on different architectures since:

- The working set when interpolating two data-sets simultaneously will be twice as large as interpolating a single one individually. This may lead to reduced cache utilisation on some architectures.
- Utilising only complex-to-complex transforms allows many operations to be performed in-place, reducing memory requirements.
- The complex-to-complex, real-to-complex and complex-to-complex transforms may have undergone differing levels of performance optimisation.
- Combining, and in particular separating (since the interpolated data is 8× larger) involves access to additional memory regions, increasing the size of the working set. Additionally, this data movement costs time and bandwidth but performs no mathematical operations, leading to reduced efficiency in terms of FLOPs per byte moved.

We have implemented interpolation variants for all three upsampling algorithms that interpolate each pair of real data-sets separately and combined as a complex-valued data-set.

3.2.2. Batched FFTs versus individual FFTs

The padding-aware and phase-shift implementations perform a large number of one-dimensional Fourier transforms. For all dimensions except one, performing a one-dimensional Fourier transform on a pencil-shaped slice through an array will result in data being accessed from non-contiguous locations. On processors with caches, each access will typically read a larger amount of data, called a cache-line. If the other data in this cache-line is unused, bandwidth has been wasted.

Performing Fourier transforms in batch has a number of benefits. For input data arranged in the appropriate manner, it is possible to vectorise across corresponding elements in the different inputs and reduce control flow overhead [29]. Use of vector instructions (such as SSE and AVX on Intel processors) enables reductions in both the number of instructions required for computation and for loads and stores.

Despite the advantages of batch transforms, we have also explored the approach of using individual Fourier transforms. When using individual transforms, we reduce temporary storage requirements to a pencil-shaped region of the array being interpolated (as opposed to another array of the same size). Also, since the data being operated on will undergo other processing after the first Fourier transform (multiplication by phase factors and an inverse transform), performing individual FFTs ensures that the data is highly likely to be in cache after the transform. In contrast, batched transforms operate on larger amounts of data, so later operations may require data that has since been evicted from the cache.

4. Synthetic results and discussion

We implemented the three interpolation algorithms described in our C library, TINTL (Trigonometric INTerpolation Library) [28]. For each interpolation algorithm, the library measures execution times for the different platform-specific variations in order to select the highest performing one. Hence, our results use the highest-performing code variants for each algorithm.

In this section, we present results exploring the impact of our platform-specific variations in isolation, hence the term "synthetic". In Section 5, we look at the effectiveness of our techniques in a real application context.

4.1. Algorithmic

We compared the performance of the three algorithmic choices. For our benchmarking, we used a system containing 2.0 GHz Intel Xeon E5-2620 "Sandy Bridge" CPUs. All interpolations were performed single-threaded. Version 11.0 of Intel's Math Kernel Library [30] (IMKL) was used as the underlying library for performing FFTs. These results are shown in Fig. 8. A problem size of n corresponds to the upsampling of a split-format complex-valued $n \times n \times n$ data-set.

The performance of the FFT routines such as those available in FFTW [31] and Intel's Math Kernel library [30] are typically higher with transform sizes that can be expressed as a product of small primes (smooth numbers). The "Cooley–Tukey" algorithm which forms the basis for many FFT implementations [31] can decompose a transform of composite size, but other algorithms must be employed to handle prime-sized cases. As a consequence, it is difficult to see a consistent performance trend for an FFT implementation across a range of transform sizes unless the results have been filtered to k-smooth numbers for some appropriately chosen k.

ONETEP increases the size of its FFT-boxes in order to avoid choosing transform sizes that give unfavourable performance. We show our performance results filtered to problem sizes of the form $2^a3^b5^c7^d11^e13^f$ where $e + f < 2$. This is both more representative of the transform sizes used by ONETEP and makes the performance trends for these sizes clearer.

We find that on our "Sandy Bridge" system, for problem sizes between 75 and 120 (those most relevant to ONETEP) that the padding-aware and phase-shift approaches provide average throughput improvements of 1.49× and 1.84× against the naïve approach, respectively. For the same range of problem size, the phase-shift approach provides an average throughput improvement of 1.22× against the padding-aware approach. The relative performance of all three algorithms is close to that derived in Section 3.1.4 using operation counts.

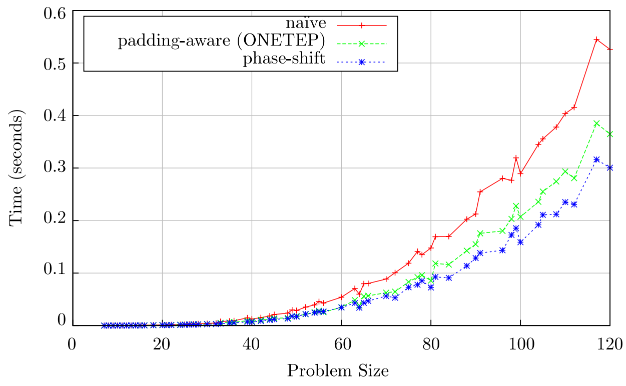

Fig. 8. Time taken to interpolate three-dimensional complex-valued data-sets in split format using a single thread running on a 2.0 GHz Intel Xeon E5-2620 CPU with Intel MKL 11.0 as the underlying FFT library. Results have been filtered to only include sizes that are of the form $2^a3^b5^c7^d11^e13^f$ where $e + f < 2$.

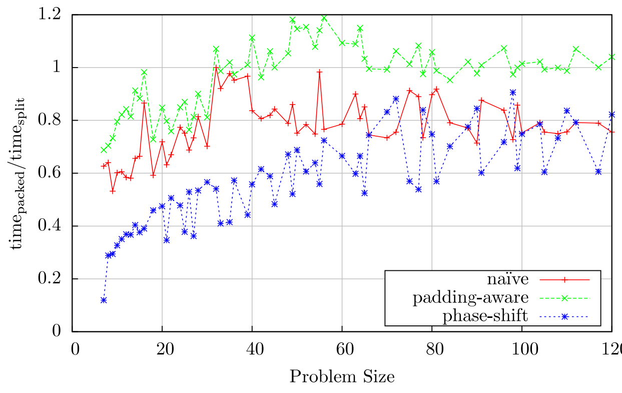

Fig. 9. Cost of interpolating two three-dimensional real-valued data-sets separately relative to combining both into a complex-valued data-set, interpolating, then splitting into two real-valued results. Benchmarking was done using a single thread running on a 2.0 GHz Intel on E5-2620 CPU with Intel MKL 11.0 as the underlying FFT library. Results have been filtered to only include sizes that are of the form $2^a3^b5^c7^d11^e13^f$ where $e + f < 2$.

4.2. Platform-dependent

In this section, we look at the performance impact of our platform-dependent implementation variations. These do not change the underlying algorithm or operation count, but represent different ways of mapping the algorithm onto the underlying FFT libraries.

For our implementation, we benchmark different implementations at runtime and select the best-performing variant, allowing us to take advantage of platform-dependent variation.

4.2.1. Complex versus real interpolation

ONETEP, our application of interest, applies Fourier interpolation to real-valued data-sets. It typically interpolates two real-valued data-sets simultaneously. This can either be implemented as two interpolations of real-valued data-sets or the single interpolation of a complex-valued data-set (the real data-sets are combined as the real and imaginary components of a complex-valued input and split afterwards).

We note that FFTW (and IMKL, which supports the FFTW interface) has the ability to accept input and output data in *split* format, which represents complex-valued data as two separate arrays containing the real and imaginary components. This offers the possibility of avoiding the steps of combining and separating real-valued data-sets when interpolating both simultaneously. However, this functionality is not suitable for our purposes since an FFTW plan requires a fixed offset in memory between the real and imaginary arrays. The data-sets we interpolate are at varying locations in memory, so this requirement is too restrictive.

The interpolation of complex-valued and real-valued data-sets have a number of differences including:

- If interpolation is performed using complex-valued data, the real data-sets must be combined to form a complex data-set and split apart after the interpolation.
- Interpolation of real-valued data often requires additional storage since it may involve real-to-complex or complex-to-real transforms that cannot be performed in-place.

We plot results for interpolating two real-valued data-sets in Fig. 9 for each of our three algorithms. Each plot represents the ratio of the execution time of an implementation performing two real-valued interpolations relative to the same interpolations performed using an implementation combining and separating two real-valued data-sets and interpolating purely complex-valued data.

On this platform, we observe that the best choice of data format for the "padding-aware" algorithm depends strongly on the problem size. For the "naïve" and "phase-shift" strategies, operating on a combined data-set is typically more efficient than performing separate interpolations (though we note that this is inherently a platform-specific result).

4.2.2. Batched FFTs versus individual FFTs

In the "phase-shift" implementation, multiple interpolations are performed in which the input and output data-sets are the same size (Fig. 7). Each interpolated block represents a new set of interpolated samples. All blocks are later interleaved to form the higher-resolution representation of the input.

Interpolation of each block in the multi-dimensional case is performed using the technique shown in Fig. 6 (excluding the interleave step). The one-dimensional process is performed on each one-dimensional "pencil" through the block in the dimension being interpolated.

Each one-dimensional FFT may access data from non-contiguous areas of memory (depending on the stride of the dimension) giving poor cache utilisation. Batched FFTs allow FFT implementations to improve memory system utilisation by performing multiple FFTs simultaneously whose data is co-located in memory.

Batched FFTs can be expected to be at least as fast as performing the same set of transforms individually. However, for our "phase-shift" implementation, we wish to optimise memory utilisation across a forward and backward FFT and a correlation (the element-wise multiplication of phase-shift coefficients).

We compare two different implementations:

- The "batched" implementation in which the forward and backward FFTs are applied to entire blocks. The element-wise multiplication is also applied to the entire block.
- The "individual" implementation in which the forward transform is applied to a single "pencil", the pencil undergoes element-wise multiplication then the backward transform is applied.

The first implementation enables leveraging the performance improvements afforded by using batched FFTs. The second improves temporal locality of memory accesses by moving all manipulation of each "pencil" closer together in the algorithm. We present the timings of interpolating a given data-block using the "individual" approach relative to the "batched" approach in Fig. 10.

We observe that for smaller problem sizes, using batched FFTs is usually more efficient for all three dimensions. For larger problem sizes, it becomes more efficient to use the "individual" approach in the x dimension (in which values are contiguous) but more unpredictable for the other two dimensions. Hence, for many larger

Table 1
Properties of problems on which ONETEP benchmarks were run.

Abbrev.	Psinc energy cutoff (eV)	NGWF radius (Bohr)	FFT-box size	Atoms	Description
amyloid	600	7.0	$75 \times 75 \times 75$	1712	Amyloid fibril
nanotube	800	8.0	$99 \times 99 \times 99$	700	Carbon nanotube
cellulose	800	8.0	$99 \times 99 \times 99$	1881	Cellulose
lysozyme	1000	8.0	$99 \times 99 \times 99$	2615	Lysozyme complex
tbl600	600	8.0	$91 \times 91 \times 91$	181	"Tennis ball" monomer
tbl800	800	8.0	$99 \times 99 \times 99$	181	"Tennis ball" monomer
tbl1000	1000	8.0	$117 \times 117 \times 117$	181	"Tennis ball" monomer
tbl1200	1200	8.0	$117 \times 117 \times 125$	181	"Tennis ball" monomer

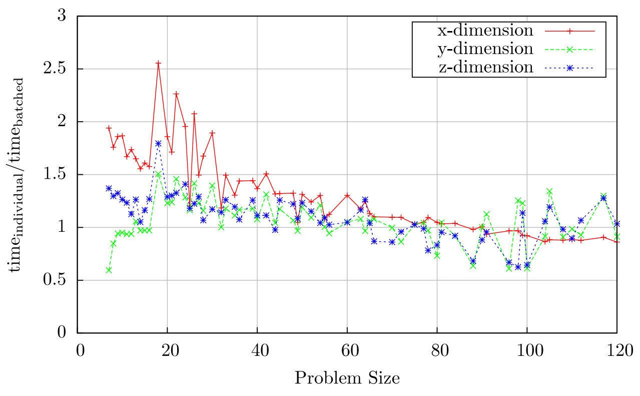

Fig. 10. Within the "phase-shift" implementation, cost of interpolating using individual transforms relative to the use of batched FFTs. Results are shown for interpolation in the x, y, and z directions, each of which has different data strides. Benchmarking was done using a single thread running on a 2.0 GHz Intel on E5-2620 CPU with Intel MKL 11.0 as the underlying FFT library. Results have been filtered to only include sizes that are of the form $2^a3^b5^c7^d11^e13^f$ where $e+f < 2$.

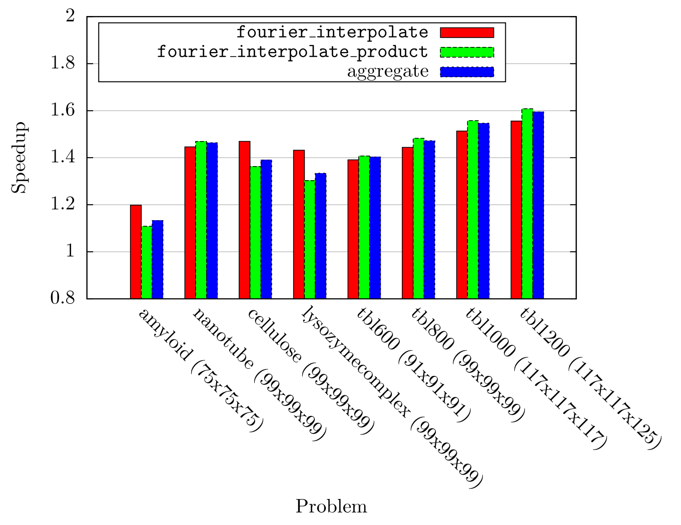

Fig. 11. Performance impact of optimal algorithm selection on Fourier interpolation routines inside ONETEP on different test problems. FFT box sizes are shown in parentheses. Results are shown for the `fourier_interpolate` and `fourier_interpolate_product` routines and for the aggregated timings of both routines.

problem sizes, TINTL will use choose to perform phase-shift interpolation using a mixture of batched and individual transforms.

Having presented the performance of our interpolation implementation in isolation, we now look at effectiveness for a real application context in the following section.

5. ONETEP results and discussion

Details of the problems chosen for the tests within ONETEP are given in Table 1. The problems were chosen to be representative of typical ONETEP use. The 181 atom "tennis ball" monomer is a very small system by ONETEP standards and is used as a reference in which we consider the effect of varying the size of the FFT boxes (by changing the kinetic energy cutoff) whilst maintaining the same molecular structure. The 700 atom carbon nanotube represents a structurally well-ordered system containing only a single atom type, permitting an ideal distribution of the workload across multiple cores. The 1881 atom cellulose system also has a well ordered structure but contains multiple atom types. The 1712 atom beta-amyloid fibril and 2615 atom lysozyme complex are both larger, non-crystalline biomolecular systems and approach the problem size for which ONETEP is designed. However, it is of note that the lysozyme system has a relatively small FFT-box size compared to the other test systems.

We modified ONETEP to use TINTL and compared performance against standard ONETEP. Version 3.5.9.1 of ONETEP was used for the benchmarks and was compiled with version 13.1.3 of the Intel Fortran compiler. Version 11.0 update 5 of Intel's MKL was used as the underlying library for performing FFTs.

Each ONETEP benchmark was run with MPI parallelisation with a maximum of 8 atoms assigned to each MPI process. All of the benchmarks except for the "Tennis Ball" problems were parallelised across multiple nodes.

Benchmarking was performed on the "ARCHER" UK national supercomputing service [32]. Each node contained two 2.7 GHz, 12-core Intel Xeon E5-2697 v2 "Ivy Bridge" processors. One ONETEP MPI process was assigned to each of the 24 physical cores of each node (12 per socket).

In a manner similar to our automatic selection of the best platform-specific implementation variations, TINTL times all three algorithmic choices at runtime then provides the fastest implementation to ONETEP. For our benchmarks, the "phase-shift" implementation is typically chosen.

Standard ONETEP uses the "padding-aware" approach (Section 3.1.2), interleaving two real-valued data-sets so that interpolation can be performed on complex-valued data.

We present speedup results in Fig. 11 restricted to the ONETEP routines that perform Fourier interpolation. We present speedup values for the `fourier_interpolate` and `fourier_interpolate_product` routines (see Section 2) and also for the aggregated timings of both. These routines are used in the calculation of the local potential and charge density respectively. The unmodified code is seen to spend between 15% and 53% of its entire runtime in these two routines. The lowest value of 15% is seen for the amyloid fibril problem which has a particularly small FFT box, all other problems spend more than 29% of their runtime in these routines.

Both `fourier_interpolate` and `fourier_interpolate_product` interpolate two real-valued data-sets but the latter routine performs a point-wise multiply between the two resulting data-sets, giving a single real-valued data-set. The library we have written also has functionality to handle this case since performing the product within the interpolation routines enables reducing the amount of data written compared to if the product was performed

Table 2
Performance counter values for the `fourier_interpolate` routine in ONETEP using the native ONETEP "padding-aware" implementation the TINTL implementation of the same algorithm. Results were collected from the `tbl1000` benchmark problem which has an FFT box size of 117 × 117 × 117.

Performance Counter	Unmodified ONETEP	Modified ONETEP
L1 Data cache miss	3.87*e*7	2.95*e*7
L2 Data cache miss	1.74*e*7	9.20*e*6
L3 Cache miss	1.22*e*7	6.62*e*6
Instructions	1.16*e*9	1.08*e*9
Floating point operations	1.55*e*9	1.56*e*9
Clock cycles	2.03*e*9	1.41*e*9

later. The `fourier_interpolate_product` routine is typically called 3–4 times more often than `fourier_interpolate` which is why the aggregate speedup tends towards the same values as `fourier_interpolate_product`.

We find that on realistic benchmark problems, we are able to improve the throughput of ONETEP's interpolation routines by over 1.55×. We observe that we typically achieve greater performance improvements with larger FFT-box sizes. We also observe performance variation in the `nanotube`, `cellulose`, `lysozyme` and `tbl800` problems. Each of these problems has the same FFT-box size (99 × 99 × 99) but exhibits different speedup values for `fourier_interpolate_product` (though not `fourier_interpolate`). We have determined that both standard ONETEP and ONETEP modified to use TINTL exhibit performance variations in the `fourier_interpolate_product` routine across these problems despite identical FFT box sizes. Changing the number of nodes the problem is parallelised across does not appear to change this result suggesting that this behaviour is not due to MPI communication.

Standard ONETEP uses its own implementation of the "padding-aware" algorithm whereas our interpolation library typically chooses to use the "phase-shift" approach. Across many benchmark problems, the performance improvement we observe for ONETEP's Fourier interpolation routines is much larger than the 1.22× we observed in our synthetic benchmarks comparing these two algorithms (Section 4.1). This suggests that the "padding-aware" implementation in TINTL must be operating more efficiently than ONETEP's native "padding-aware" implementation, despite using the same algorithm.

We compared the performance characteristics of the "padding-aware" implementation in ONETEP and in TINTL using the Performance Application Programming Interface (PAPI) library [33] to monitor processor hardware performance counters. Both implementations were analysed during a ONETEP execution on the `tbl1000` problem. Results are shown in Table 2.

We note that both "padding-aware" implementations have similar instruction and floating point operation counts. However, the native ONETEP implementation incurs significantly more cache misses. Since we perform the same FFT calls, we attribute this to the code that stages data to and from the FFT routines in the native ONETEP implementation.

We present results in Fig. 12 that show the impact of our optimisations on the overall running time of ONETEP. In this case, the extent to which performance is improved is highly problem dependent, since different problems spend different proportions of time performing Fourier interpolation.

6. Conclusions

We have presented three algorithms for implementing Fourier upsampling built from Discrete Fourier transforms. For each of these algorithms we have derived operation counts that show that these algorithms have different costs (although the same asymptotic complexity). We have presented performance results

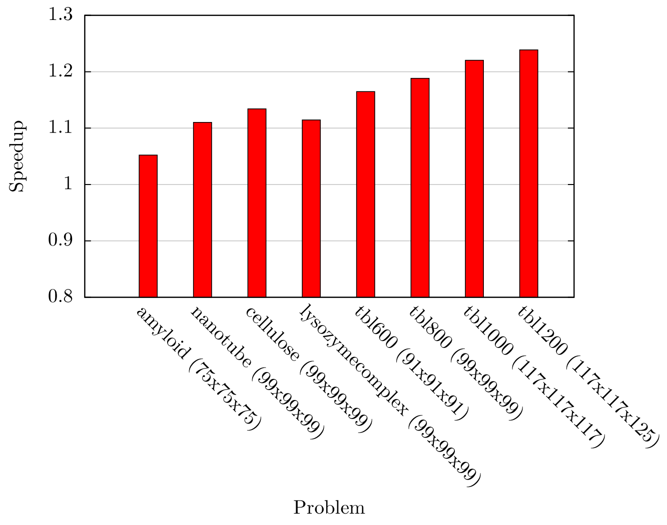

Fig. 12. Performance impact of optimal algorithm selection on total ONETEP execution time on different test problems.

showing that the differences in operation count have a measurable impact when implemented on top of an existing high-performance Fast Fourier Transform implementation.

As well as varying the underlying algorithm, we have explored platform-dependent optimisations that do not affect the operation count but instead the efficiency with which the code can be executed by a given processor.

The performance of implementation variations across software platform, hardware platform and problem size is hard to predict. In response, we have developed a system that can choose the appropriate algorithm and platform-dependent optimisations to employ at runtime. We have evaluated the performance of this library both standalone and when called by the linear-scaling quantum chemistry code ONETEP.

We have hand-crafted our implementation of these algorithms, which are currently restricted to 2× upsampling for the three-dimensional case. We would like to support multiple dimensions and different resampling resolutions. However, code that generalises efficiently to multiple dimensions can be hard to write correctly by hand. An approach using code-generation to automatically generate implementations of the algorithms described would enable this.

Our approach in this paper has been to leverage existing FFT libraries in order to achieve high performance. An alternative approach would be to use code-generation to generate parts (or the entirely of) high-performance interpolation implementations directly. Pruned FFT implementations [9] also offer the possibility of reducing the operation count of the "padding-aware" approach. However, as interpolation is expressible as a linear transform, the SPIRAL [34] framework should be capable of generating the entirety of an interpolation implementation.

In conclusion, we have shown how both choice of algorithm and variations in algorithm implementation to exploit specific platforms can be used to improve performance through the selection of efficient implementations at runtime. As a real test case, we have presented performance results from the linear-scaling quantum chemistry code ONETEP. The implementation of the algorithms within the TINTL library means that an extensive code rewrite is not required. Code additions required to call the TINTL interpolation implementation are minimal, and removal of the existing interpolation implementation results in a code decrease. Extensive benchmarks with ONETEP illustrate that our techniques can be used to provide performance increases of over 1.55× for individual routines and over 1.2× for the entire calculation across a number of non-trivial benchmark problems.

Acknowledgements

This work was supported by the Engineering and Physical Sciences Research Council (EPSRC). Francis Russell was supported under grants EP/I006761/1 and EP/I00677X/1. Karl Wilkinson was supported under grant EP/I006613/1. We would like to thank Dr. Nicholas Hine for useful discussions during the early stages of this work and Professor Peter Haynes for his feedback on this paper. This work used the ARCHER UK National Supercomputing Service (http://www.archer.ac.uk).

References

[1] T. Smit, M.R. Smith, S.T. Nichols, Efficient sinc function interpolation technique for center padded data, IEEE Trans. Acoust. Speech Signal Process. 38 (9) (1990) 1512–1517. http://dx.doi.org/10.1109/29.60071.

[2] C.-K. Skylaris, P.D. Haynes, A.A. Mostofi, M.C. Payne, Implementation of linear-scaling plane wave density functional theory on parallel computers, Phys. Status Solidi b 243 (5) (2006) 973–988. http://dx.doi.org/10.1002/pssb.200541328.

[3] D. McFarlin, F. Franchetti, J.M.F. Moura, M. Püschel, High performance synthetic aperture radar image formation on commodity architectures, in: SPIE Conference on Defense, Security, and Sensing, vol. 7337, Proc. SPIE, 2009, p. 733708. http://dx.doi.org/10.1117/12.818399.

[4] A. Canning, Scalable parallel 3d FFTs for electronic structure codes, in: J.M. L.M. Palma, P.R. Amestoy, M. Daydé, M. Mattoso, J.C. Lopes (Eds.), High Performance Computing for Computational Science - VECPAR 2008, in: Lecture Notes in Computer Science, vol. 5336, Springer, Berlin. Heidelberg, 2008, pp. 280–286. http://dx.doi.org/10.1007/978-3-540-92859-1_25.

[5] N. Dugan, L. Genovese, S. Goedecker, A customized 3D GPU Poisson solver for free boundary conditions, Comput. Phys. Comm. 184 (8) (2013) 1815–1820. http://dx.doi.org/10.1016/j.cpc.2013.02.024.

[6] J. Hutter, A. Curioni, Dual-level parallelism for ab initio molecular dynamics: Reaching teraflop performance with the CPMD code, Parallel Comput. 31 (1) (2005) 1–17. http://dx.doi.org/10.1016/j.parco.2004.12.004.

[7] S. Goedecker, M. Boulet, T. Deutsch, An efficient 3-dim FFT for plane wave electronic structure calculations on massively parallel machines composed of multiprocessor nodes, Comput. Phys. Comm. 154 (2) (2003) 105–110. http://dx.doi.org/10.1016/S0010-4655(03)00287-X.

[8] P.D. Haynes, M. Côté, Parallel fast Fourier transforms for electronic structure calculations, Comput. Phys. Comm. 130 (1–2) (2000) 130–136. http://dx.doi.org/10.1016/S0010-4655(00)00049-7.

[9] F. Franchetti, M. Püschel, Generating high performance pruned FFT implementations, in: ICASSP, IEEE, 2009, pp. 549–552. http://dx.doi.org/10.1109/ICASSP.2009.4959642.

[10] M.D. Segall, P.J.D. Lindan, M.J. Probert, C.J. Pickard, P.J. Hasnip, S.J. Clark, M.C. Payne, First-principles simulation: ideas, illustrations and the CASTEP code, J. Phys.: Condens. Matter 14 (11) (2002) 2717. http://dx.doi.org/10.1088/0953-8984/14/11/301.

[11] G. Kresse, J. Furthmüller, Efficient iterative schemes for ab initio total-energy calculations using a plane-wave basis set, Phys. Rev. B 54 (1996) 11169–11186. http://dx.doi.org/10.1103/PhysRevB.54.11169.

[12] P. Giannozzi, S. Baroni, N. Bonini, M. Calandra, R. Car, C. Cavazzoni, D. Ceresoli, G.L. Chiarotti, M. Cococcioni, I. Dabo, A. Dal Corso, S. de Gironcoli, S. Fabris, G. Fratesi, R. Gebauer, U. Gerstmann, C. Gougoussis, A. Kokalj, M. Lazzeri, L. Martin-Samos, N. Marzari, F. Mauri, R. Mazzarello, S. Paolini, A. Pasquarello, L. Paulatto, C. Sbraccia, S. Scandolo, G. Sclauzero, A.P. Seitsonen, A. Smogunov, P. Umari, R.M. Wentzcovitch, QUANTUM ESPRESSO: a modular and open-source software project for quantum simulations of materials, J. Phys.: Condens. Matter 21 (39) (2009) 395502. http://dx.doi.org/10.1088/0953-8984/21/39/395502.

[13] W. Andreoni, A. Curioni, New advances in chemistry and material science with CPMD and parallel computing, Parallel Comput. 26 (2000) 819–842. http://dx.doi.org/10.1016/S0167-8191(00)00014-4.

[14] X. Gonze, B. Amadon, P.-M. Anglade, J.-M. Beuken, F. Bottin, P. Boulanger, F. Bruneval, D. Caliste, R. Caracas, M. Côté, T. Deutsch, L. Genovese, P. Ghosez, M. Giantomassi, S. Goedecker, D.R. Hamann, P. Hermet, F. Jollet, G. Jomard, S. Leroux, M. Mancini, S. Mazevet, M.J.T. Oliveira, G. Onida, Y. Pouillon, T. Rangel, G.-M. Rignanese, D. Sangalli, R. Shaltaf, M. Torrent, M.J. Verstraete, G. Zerah, J.W. Zwanziger, ABINIT: First-principles approach to material and nanosystem properties, Comput. Phys. Comm. 180 (12) (2009) 2582–2615. http://dx.doi.org/10.1016/j.cpc.2009.07.007.

[15] C.-K. Skylaris, P.D. Haynes, A.A. Mostofi, M.C. Payne, Introducing ONETEP: Linear-scaling density functional simulations on parallel computers, J. Chem. Phys. 122 (8) (2005) 084119. http://dx.doi.org/10.1063/1.1839852.

[16] K. Wilkinson, C.-K. Skylaris, Porting ONETEP to graphical processing unit-based coprocessors. 1. FFT box operations, J. Comput. Chem. 34 (28) (2013) 2446–2459. http://dx.doi.org/10.1002/jcc.23410.

[17] W. Kohn, Density functional and density matrix method scaling linearly with the number of atoms, Phys. Rev. Lett. 76 (1996) 3168–3171. http://dx.doi.org/10.1103/PhysRevLett.76.3168.

[18] E. Prodan, W. Kohn, Nearsightedness of electronic matter, Proc. Natl. Acad. Sci. USA 102 (33) (2005) 11635–11638. http://dx.doi.org/10.1073/pnas.0505436102.

[19] S. Goedecker, Linear scaling electronic structure methods, Rev. Modern Phys. 71 (1999) 1085–1123. http://dx.doi.org/10.1103/RevModPhys.71.1085.

[20] C.-K. Skylaris, A.A. Mostofi, P.D. Haynes, O. Diéguez, M.C. Payne, Nonorthogonal generalized wannier function pseudopotential plane-wave method, Phys. Rev. B 66 (2002) 035119. http://dx.doi.org/10.1103/PhysRevB.66.035119.

[21] A.A. Mostofi, P.D. Haynes, C.-K. Skylaris, M.C. Payne, Preconditioned iterative minimization for linear-scaling electronic structure calculations, J. Chem. Phys. 119 (17) (2003) 8842–8848. http://dx.doi.org/10.1063/1.1613633.

[22] D. Baye, P.H. Heenen, Generalised meshes for quantum mechanical problems, J. Phys. A, Math. Gen. 19 (11) (1986) http://dx.doi.org/10.1088/0305-4470/19/11/013.

[23] S.J. Fox, C. Pittock, T. Fox, C.S. Tautermann, N. Malcolm, C.-K. Skylaris, Electrostatic embedding in large-scale first principles quantum mechanical calculations on biomolecules, J. Chem. Phys. 135 (22) (2011) http://dx.doi.org/10.1063/1.3665893.

[24] C.-K. Skylaris, A.A. Mostofi, P.D. Haynes, C.J. Pickard, M.C. Payne, Accurate kinetic energy evaluation in electronic structure calculations with localized functions on real space grids, Comput. Phys. Comm. 140 (3) (2001) 315–322. http://dx.doi.org/10.1016/S0010-4655(01)00248-X.

[25] P.D. Haynes, C.-K. Skylaris, A.A. Mostofi, M.C. Payne, Density kernel optimization in the ONETEP code, J. Phys.: Condens. Matter 20 (29) (2008) 294207. http://dx.doi.org/10.1088/0953-8984/20/29/294207.

[26] T. Schanze, Sinc interpolation of discrete periodic signals, IEEE Trans. Signal Process. 43 (6) (1995) 1502–1503. http://dx.doi.org/10.1109/78.388863.

[27] L.P. Yaroslavsky, Efficient algorithm for discrete sinc interpolation, Appl. Opt. 36 (2) (1997) 460–463. http://dx.doi.org/10.1364/AO.36.000460.

[28] TINTL homepage, https://github.com/FrancisRussell/tintl, last checked 2014-09-22.

[29] S. Goedecker, Rotating a three-dimensional array in an optimal position for vector processing: case study for a three-dimensional fast fourier transform, Computer Phys. Communications 76 (3) (1993) 294–300. http://dx.doi.org/10.1016/0010-4655(93)90057-J.

[30] Intel, Intel$^{®}$ Math Kernel Library, http://software.intel.com/en-us/intel-mkl last checked 2014-09-22.

[31] M. Frigo, S.G. Johnson, The design and implementation of FFTW3, Program Generation, Optimization, and Platform Adaptation, Proceedings of the IEEE 93 (2) (2005) 216–231. (special issue). http://dx.doi.org/10.1109/JPROC.2004.840301.

[32] ARCHER homepage, http://www.archer.ac.uk, last checked 2014-09-22.

[33] D. Terpstra, H. Jagode, H. You, J. Dongarra, Collecting performance data with PAPI-C, in: M.S. Müller, M.M. Resch, A. Schulz, W.E. Nagel (Eds.), Tools for High Performance Computing 2009, Springer, Berlin. Heidelberg, 2010, pp. 157–173. http://dx.doi.org/10.1007/978-3-642-11261-4_11.

[34] M. Püschel, J.M.F. Moura, J. Johnson, D. Padua, M. Veloso, B. Singer, J. Xiong, F. Franchetti, A. Gacic, Y. Voronenko, K. Chen, R.W. Johnson, N. Rizzolo, SPIRAL: Code generation for DSP transforms, Program Generation, Optimization, and Adaptation, Proc. IEEE 93 (2) (2005) 232–275. (special issue). http://dx.doi.org/10.1109/JPROC.2004.840306.