Implicitly Dealiased Convolutions: Example Applications and Performance Comparison

Malcolm Roberts
IRMA, University of Strasbourg
7 rue Ren-Descartes
67084 Strasbourg Cedex
Email: malcolm.i.w.roberts@gmail.com

John C. Bowman
University of Alberta
CAB 632, University of Alberta
Edmonton, Alberta
Canada T6G 2G1
Email: bowman@ualberta.ca

***Abstract*—Implicitly dealiasing is a recently-developed technique which improves upon conventional zero padding to compute linear convolutions via fast Fourier transforms. For one-dimensional inputs, the memory requirements and performance are similar to conventional zero-padded convolutions, but implicitly dealiased convolutions are faster and require less memory when the data is multi-dimensional. We show how implicit dealiased convolutions can be used in some common applications and evaluate their performance using the `FFTW++` library relative to the standard technique and in other implementations.**

I. Introduction

The convolution of $f = \{f_k\}_{k\in\mathbb{Z}}$ and $g = \{g_k\}_{k\in\mathbb{Z}}$ is denoted $f * g = \{(f * g)_k\}_{k\in\mathbb{Z}}$, where

$$(f * g)_k = \sum_{\ell\in\mathbb{Z}} f_\ell g_{k-\ell}. \tag{1}$$

The convolution is an important operator in a variety of applications such as statistics, signal processing, data and image processing, and the numerical approximation of solutions to nonlinear partial differential equations. In practical applications, the inputs $f = \{f_k\}_{k=0}^{N-1}$ and $g = \{g_k\}_{k=0}^{N-1}$ are of finite length N, and the linear convolution $f * g$ is given by

$$(f * g)_k = \sum_{\ell=0}^{k} f_\ell g_{k-\ell}, \quad k = 0, \ldots, N-1. \tag{2}$$

Computing equation (2) directly requires $\mathcal{O}(N^2)$ operations, and rounding error is a significant problem when N is large. It is therefore generally preferred to compute a convolution using the convolution theorem and a fast Fourier transform (FFT): the Fourier transform maps the convolution to a component-wise multiplication in Fourier space. The FFT requires $\mathcal{O}(N \log N)$ [1], [2] operations and the multiplication requires $\mathcal{O}(N)$ operations, reducing the computational complexity to $\mathcal{O}(N \log N)$ and greatly improving the numerical accuracy.

Since the FFT considers the inputs f and g to be periodic, the direct application of the convolution theorem results in a circular convolution, due to the indices in equation (2) being computed as modulo N. Removing these extra aliased terms from the periodic convolution to produce a linear convolution is called dealiasing. We give a brief overview of dealiasing requirements for different types of convolutions in section II.

The standard method for dealiasing FFT-based convolutions is to zero-pad the inputs to an appropriate length so that the aliased terms are all zero. In section III we review the method of implicit dealiasing [3] which reduces the memory requirements of dealiasing multidimensional convolutions. The implicitly dealiased convolution routines are available in the open-source software library `FFTW++` [4], which, in addition to reducing memory requirements, are generally faster than convolutions that are dealiased using explicit zero-padding [3].

In this paper, we demonstrate the use of implicitly dealiased convolutions using the `FFTW++` implementation in various example applications and the performance of `FFTW++` is compared with other implementations. In section IV, we use the convolution to compute the autocorrelogram of a time series in order to test for and determine the presence of periodicity and compare the speed of `FFTW++`'s autocorrelation routine with one based on `Python`'s `numpy` library. Section V shows the use of the convolution for filtering data, which we apply to a simple example of image analysis. We also make a comparison between the performance of `FFTW++` and the `Python scipy` library for image convolution filtering and other realistic data types. In section VI we compare implicit and conventional dealiasing techniques in the context of pseudospectral simulations of solutions to the Navier–Stokes equations. Conclusions and future directions for research are given in section VII.

II. Dealiasing requirements for convolutions

The type of input data and application determines important properties of how the convolution is computed. The elements of f and g can be real or complex. If the Fourier transform of f is real valued, then f exhibits a Hermitian symmetry that can be used to reduce storage requirements. Moreover, complex-to-real/real-to-complex Fourier transforms can make use of Hermitian symmetry to reduce computational complexity and memory requirements by about a factor of two.

For input data of length N which is real-valued or complex and does not have Hermitian symmetry, the data is padded with N zeroes for a total FFT length of $2N$, which we refer to as "1/2 padding". If the input data is multidimensional with size $N_1 \times \cdots \times N_d$, then the data must be zero padded to $2N_1 \times \cdots \times 2N_d$, increasing the problem size by a factor of 2^d.

In the case where the complex input data has Hermitian symmetry, it is useful to shift the zero-index of the data so that it is in the middle of the array. In this case, the inputs are $f = \{f_k\}_{k=-N+1}^{N-1}$ and $g = \{g_k\}_{k=-N+1}^{N-1}$, and their convolution is

$$(f * g)_k = \sum_{\ell=k-N+1}^{N-1} f_\ell g_{k-\ell}, \quad k = -N+1, \ldots, N-1. \quad (3)$$

This reduces the amount of zero padding that is necessary: data of length $2N-1$ need be padded only to length $3N$, which is referred to as "$2/3$ padding" [5]. In this case, the problem size for d-dimensional data is increased by a factor of $(3/2)^d$.

The binary convolution in equation (2) can be extended to a general n-ary operator $*(f_1, \ldots, f_n)$, with

$$*(f_1, \ldots, f_n)_\ell = \sum_{\ell_1, \ldots, \ell_n} f_{\ell_1} \cdots f_{\ell_n} \delta_{\ell_1 + \cdots + \ell_n, \ell}, \quad (4)$$

where δ is the Kronecker delta. In the case of $1/2$ padding, this may be computed via a series of binary convolutions, but one must compute the $2/3$ padding case all at once, which increases the problem size by a factor of $\left(\frac{n+1}{2}\right)^d$.

In certain cases the convolution operator must be thought of as being an operator with A inputs and B outputs, and where the multiplication performed after the FFT can be an arbitrary component-wise operation. One performs A FFTs to transform the inputs, performs an appropriate operation on the transformed data, and then performs B inverse FFTs to produce the final output, for a total of $A + B$ FFTs. For the standard binary convolution, there are $A = 2$ inputs and $B = 1$ output and the multiplication operation is

$$(x, y) \to (xy), \quad (5)$$

which is applied element-by-element. An autoconvolution can be computed with $A = B = 1$ (requiring just two transforms), using the operation

$$x \to x^2; \quad (6)$$

computing an autoconvolution via a binary convolution with the same argument repeated requires three transforms. To compute the nonlinear term of the two-dimensional Navier–Stokes equation in vorticity formulation for a pseudospectral simulation, the operation is

$$\left(u_x, u_y, \frac{\partial \omega}{\partial x}, \frac{\partial \omega}{\partial y}\right) \to \left(u_x \frac{\partial \omega}{\partial x} + u_y \frac{\partial \omega}{\partial y}\right), \quad (7)$$

where $\boldsymbol{u} = (u_x, u_y)$ is the 2D velocity, $\omega = \hat{\boldsymbol{z}} \cdot \boldsymbol{\nabla} \times \boldsymbol{u}$ is the z-component of the vorticity, and a total of 5 FFTs are required. For three-dimensional magnetohydrodynamic (MHD) flows the operation is

$$(\boldsymbol{u}, \boldsymbol{\omega}, \boldsymbol{B}, \boldsymbol{j}) \to (\boldsymbol{u} \times \boldsymbol{\omega} + \boldsymbol{j} \times \boldsymbol{B}, \boldsymbol{u} \times \boldsymbol{B}), \quad (8)$$

where $\boldsymbol{u}$ is the (3D) velocity, $\boldsymbol{\omega} = \boldsymbol{\nabla} \times \boldsymbol{u}$ is the vorticity, $\boldsymbol{B}$ is the magnetic field, and $\boldsymbol{j}$ is the current density. Computing the nonlinear term of the MHD equations in this fashion requires 12 forward 6 backward FFTs. For the Navier–Stokes and MHD equations, the operation is quadratic in its arguments, implying that the convolution is binary, ie $n = 2$. Since the wavevectors ares symmetric about the Fourier origin, $2/3$ padding can be used.

Fig. 1: Schematic diagram for computing a multi-dimensional convolution using FFTs.

Fig. 2: Schematic diagram for a computing a multi-dimensional convolution using an FFT in the last dimension and a series of sub-dimensional convolutions.

III. Implicitly dealiased convolutions

The method of implicit dealiasing allows one to dealias convolutions via zero padding without having to explicitly write, read, and multiply by zero values. This is accomplished by implicitly incorporating the zero values into the top level of a decimated-in-frequency FFT. The extra memory previously used for padding now appears as a decoupled work buffer. One dimensional implicitly dealiased convolutions therefore have the same memory requirements as explicitly padded convolutions. They also have similar performance: the advantage of implicit dealiasing in one dimension is that one need not copy the input to a separate zero-padded buffer before performing the FFT.

A d-dimensional convolution is performed by performing an FFT of size $N_1 \times \cdots \times N_d$, performing the appropriate multiplication on the transformed data, and then inverting the FFT, as shown in Figure 1. Alternatively, one can perform an FFT in dimension d, perform n_d $(d-1)$-dimensional convolutions, and then invert the FFT in dimension d. This is shown in Figure 2. Using this decomposition of convolutions and the implicitly zero-padded FFT, one can re-use work buffers for sub-convolutions, thereby reducing the total memory requirement. This is shown schematically for a two-dimensional, $1/2$ padded convolution in Figures 3 and 4. In Figure 3 the input buffer f is implicitly padded in the y-direction. The x-direction FFT is then performed column-by-column using a one-dimensional work buffer, as shown in Figure 4 for the left-most column. At this point, the x-direction FFT can be inverted on the left-most column, completing the x-convolution. The process shown in Figure 4 is repeated on the remaining columns one at a time. Once all the columns have been x-convolved, the y-FFT in Figure 3 is inverted, completing the xy-convolution. Re-using work memory for sub-convolutions allows the convolution to be computed using less total memory: for $1/2$ padded convolutions, the memory requirements per input are about twice the non-dealiased

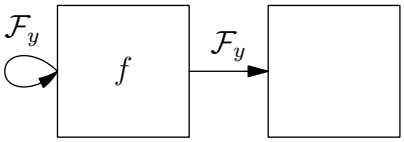

Fig. 3: Diagram showing memory use for an implicitly "1/2" zero-padded FFT in the y-direction, denoted $\mathcal{F}_y$. The input buffer containing f is implicitly zero padded in the outer loop of the y-direction FFT, with the output shared between the input buffer and and additional buffer of the same size.

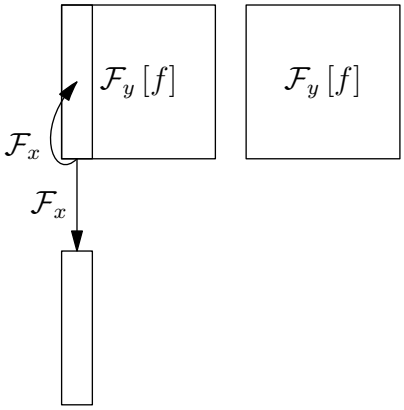

Fig. 4: Diagram showing an implicitly zero-padded FFT in the x-direction, denoted $\mathcal{F}_x$, performed on the left-most column of the output of an implicitly zero-padded FFT in the y-direction. The multiplication can now be performed on data in the left-most column.

version; this represents a memory savings of a factor of 2^{d-1} as compared to explicit padding. For $2/3$ padded convolutions, the memory savings factor is $(3/2)^{d-1}$ as compared to explicit zero-padding. In addition to reduced memory requirements, the implicitly dealiased multi-dimensional convolutions are significantly faster than their explicit counterparts, thanks in part to allowing for better cache management and the fact that the implicitly dealias convolutions automatically skip transforms on data which is guaranteed to be zero.

The implicitly dealiased convolutions in `FFTW++` use the FFTs from `FFTW`. Additional tests and functions were included in `FFTW++` to improve parallel performance on shared memory architectures using `OpenMP`.

IV. Period detection for signal analysis

In this section, we look at the use of convolutions in the application of signal analysis. Consider the cross-correlation of f and g which we denote $f \star g$, with

$$(f \star g)_k = \sum_{\ell=0}^{k} f_\ell^* g_{k+\ell}, \quad k = 0, \ldots, N-1, \tag{9}$$

where f_ℓ^* denotes the complex conjugate of f_ℓ. The cross-correlation is useful in application such as statistics and signal processing. The cross-correlation can be computed via a convolution by making use of the fact that the Fourier transform

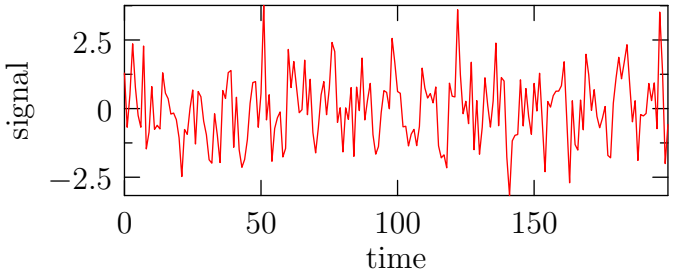

Fig. 5: An example of a time series. A sine wave is hidden by noise.

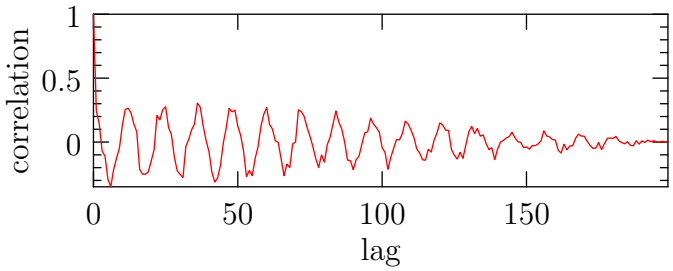

Fig. 6: The normalized autocorrelation of the signal in Figure 5, revealing its periodic nature.

of a cross-correlation is a component-wise multiplication of the complex conjugate of the Fourier transform of the first input with the Fourier transform of the second input. That is,

$$\mathcal{F}[f \star g] = (\mathcal{F}[f])^* \mathcal{F}[g]. \tag{10}$$

This can be computed via a convolution with the post-transform multiplication operator replaced by

$$(x, y) \to x^* y. \tag{11}$$

The autocorrelation of the f is is simply $f \star f$, and can be computed via a convolution using the post-transform operator

$$x \to |x^2|. \tag{12}$$

The autocorrelation is useful for detecting hidden periods in noisy data. For example, the signal in Figure 5 is composed of a Gaussian noise on top of a sine wave, which is revealed in its autocorrelation shown in Figure 6.

We now compare the performance of the `FFTW++` implementation of the one-dimensional autocorrelation with various `Python` implementations. The input data is real valued, which means that one must first copy the signal to a complex buffer before using the Python-wrapped autocorrelation `FFTW++` routine. This copy operation in the performance analysis. The `Python` implementation involves first explicitly padding the input real-valued input array before using a real-to-complex FFT, performing the multiplication, and inverting the transform with a complex-to-real FFT. The FFT transforms used were from the `Python` library `numpy`. The use of real/complex transforms reduces computation time by about

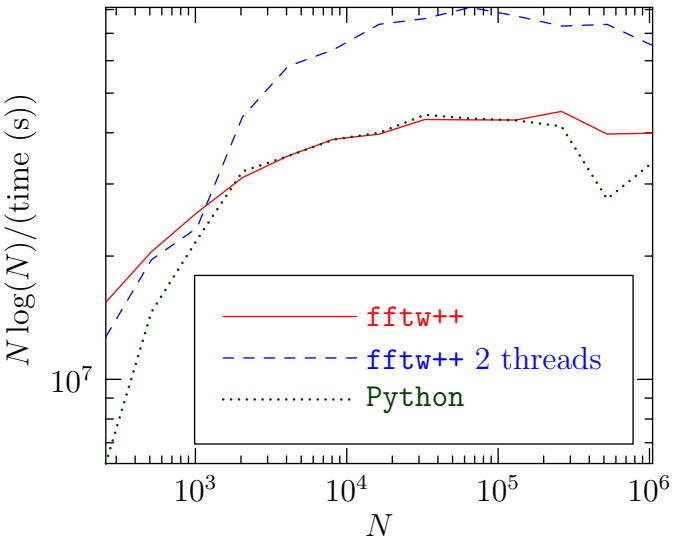

Fig. 7: Speed test for autocorrelations on real data: a comparison between FFTW++'s complex-data convolution using one or two threads with a numpy-based complex/real FFT autocorrelation using one thread. The horizontal axis shows the problem size N and the vertical axis shows the execution speed normalized by the problem size; higher values are better.

half as compared to using complex-to-complex FFTs. As can be seen in Figure 7, the FFTW++ and Python implementations have similar performance, despite the fact that the Python implementation is able to make use of real-to-complex FFTs which, a priori, reduces the computational cost by a factor of two. The FFTW++ implementation is also able to take use of all multi-threading on shared memory architectures, and there is a clear performance improvement for the cases that we considered using two cores. The problem size N was chosen to be a power of two in order to show peak performance of the implementations.

V. Image filtering

Another important application of convolutions is in image and data analysis. Convolution filters for image analysis allow one to, for example, de-noise images and smooth halftone images. As the name suggests, the convolution filter is based upon a convolution, and a popular choice for denoising and smoothing is the a Gaussian filter. The original image in Figure 8 is treated with a Gaussian filter with standard deviation $\sigma = 10$, the result of which is shown in Figure 9. For images, the convolution is two-dimensional, but three-dimensional convolutions are useful in other types of data analysis. The value of the Gaussian filter is effectively zero a few standard deviations away from the peak. When this distance is significantly smaller than the problem size, a direct computation of the convolution, ie not using an FFT, is more efficient. For large filter sizes, it is more efficient to use an FFT-based convolution.

Fig. 8: An image of two galaxies taken by the Hubble telescope. Source: NASA.

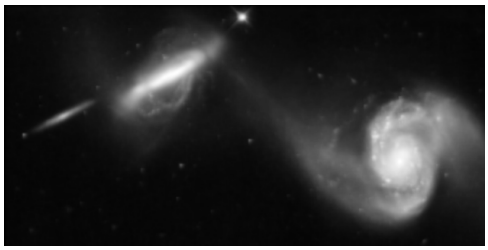

Fig. 9: The image in Figure 8 after applying a Gaussian filter with $\sigma = 10$.

In order to use the FFTW++ two-dimensional, $1/2$ padded complex convolution to apply a Gaussian filter, one must first load the filter and the image to two separate complex arrays. The convolution operates in-place, with the output stored in the first array as complex values. We compare the FFTW++ implicitly dealiased convolution with scipy.signal.fftconvolve. The use of implicit dealiasing reduces the memory requirements of the FFTW++ 2D convolution by a factor of two, but the fact that the real-valued input data is treated as complex eliminates this savings, and, for the 2D case, the two methods have similar memory footprints. A speed comparison is given in Figure 10. Again, we find that the single-threaded FFTW++ implementation is approximately as fast as the Python implementation, despite the a priori advantage for the Python from using real/complex FFTs, and that FFTW++ is able to take advantage of multiple threads in a shared-memory setting. The results are similar when considering three-dimensional convolutions of real data, but one can see in Figure 12 and Figure 13 that FFTW++'s implicitly dealiased convolutions have a clear performance benefit, even with just one thread, when considering complex data in multiple dimensions.

VI. Psuedospectral simulations of Navier–Stokes turbulence

The pseudospectral method is a computational technique to numerically solve nonlinear partial differential equations. Consider the Navier–Stokes vorticity formulation in two dimensions,

$$\frac{\partial \omega}{\partial t} + \boldsymbol{u} \cdot \boldsymbol{\nabla} \omega = \nu \nabla^2 \omega, \quad \boldsymbol{\nabla} \cdot \boldsymbol{u} = 0 \tag{13}$$

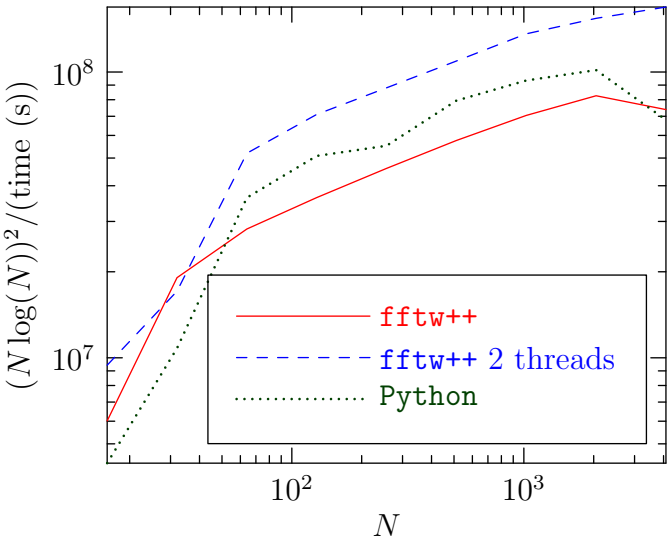

Fig. 10: Speed test for two-dimensional binary convolutions on real data: a comparison between `FFTW++`'s complex-data convolution using one or two threads with `scipy`'s `fftconvolve` using one thread. The horizontal axis shows the problem size N and the vertical axis shows the execution speed normalized by the problem size; higher values are better.

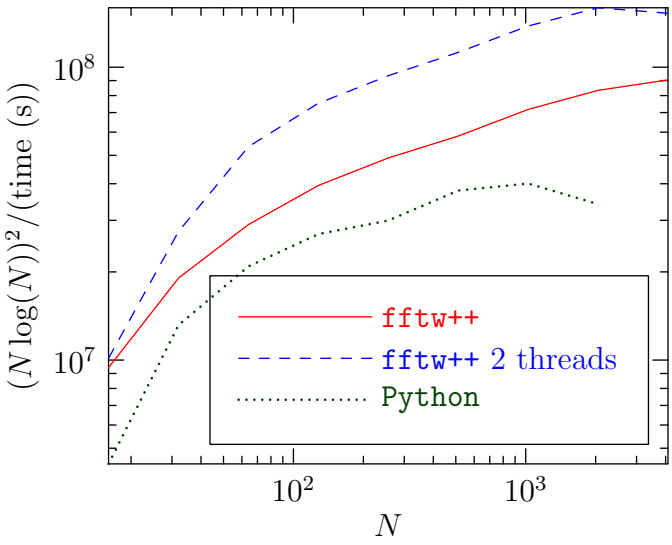

Fig. 12: Speed test for two-dimensional binary convolutions on complex data: a comparison between `FFTW++`'s complex-data convolution using one or two threads with `scipy`'s `fftconvolve` using one thread. The horizontal axis shows the problem size N and the vertical axis shows the execution speed normalized by the problem size; higher values are better.

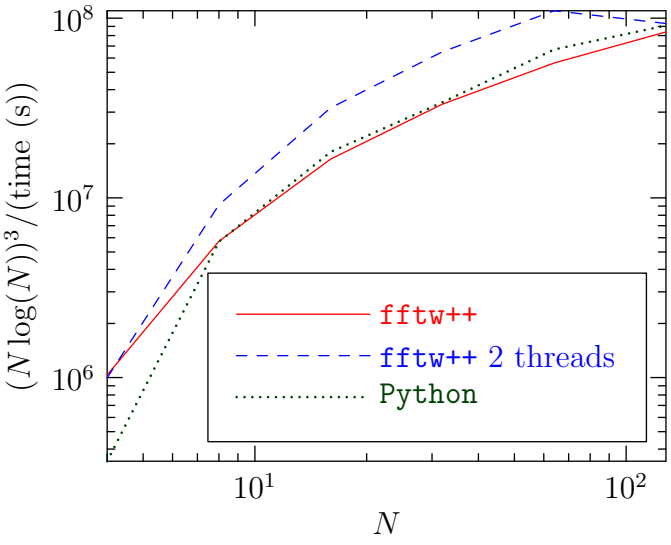

Fig. 11: Speed test for three-dimensional binary convolutions on real data: a comparison between `FFTW++`'s complex-data convolution using one or two threads with `scipy`'s `fftconvolve` using one thread. The horizontal axis shows the problem size N and the vertical axis shows the execution speed normalized by the problem size; higher values are better.

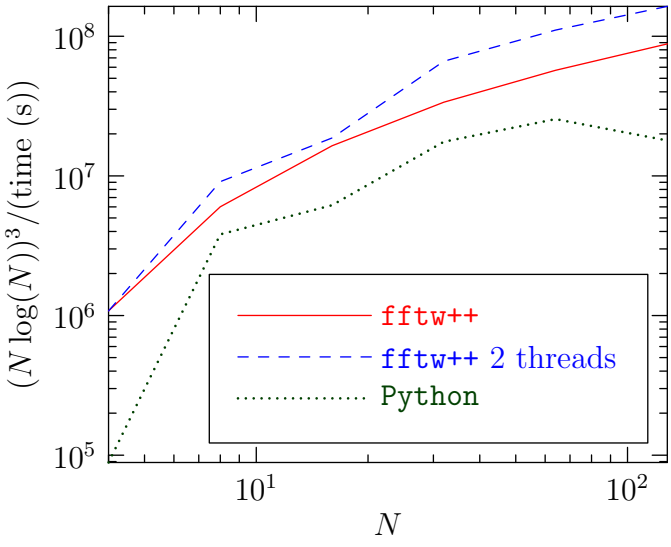

Fig. 13: Speed test for three-dimensional binary convolutions on complex data: a comparison between `FFTW++`'s complex-data convolution using one or two threads with `scipy`'s `fftconvolve` using one thread. The horizontal axis shows the problem size N and the vertical axis shows the execution speed normalized by the problem size; higher values are better.

where $\boldsymbol{u} = (u_x, u_y)$ is the velocity, $\omega = \hat{\boldsymbol{z}} \cdot \boldsymbol{\nabla} \times \boldsymbol{u}$ is the z component of the vorticity, and ν is the kinematic viscosity. The Fourier transform of equation (13) is

$$\frac{\partial \omega_{\boldsymbol{k}}}{\partial t} + \nu k^2 \omega_{\boldsymbol{k}} = \sum_{\boldsymbol{p},\boldsymbol{q}} \frac{\epsilon_{\boldsymbol{k},\boldsymbol{p},\boldsymbol{q}}}{q^2} \omega^*_{\boldsymbol{p}} \omega^*_{\boldsymbol{q}} \tag{14}$$

where $\boldsymbol{k}$, $\boldsymbol{p}$, and $\boldsymbol{q}$ are two-dimensional wave-vectors, ω_k is the Fourier transform of ω, and

$$\epsilon_{\boldsymbol{k},\boldsymbol{p},\boldsymbol{q}} = \hat{\boldsymbol{z}} \cdot (\boldsymbol{p} \times \boldsymbol{q}) \delta(\boldsymbol{k} + \boldsymbol{p} + \boldsymbol{q}). \tag{15}$$

The right-hand side of equation (14) is a convolution, and it is most efficiently computed via an FFT-based convolution: the individual terms are computed in Fourier space and then transformed back to physical space, multiplied, and then the result is transformed back into Fourier space. The diffusive term $\nu k^2 \omega_{\boldsymbol{k}}$ is directly computed and the system is advanced in time using one's favourite time-stepping method.

Since ω is real valued, $\omega_{\boldsymbol{k}}$ is complex with Hermitian symmetry. Setting the Fourier origin in the middle of the array and keeping only $\omega_{\boldsymbol{k}}$ with $\boldsymbol{k} = (k_x, k_y)$ having $k_y > 0$, one can use "2/3" padding to dealias the convolution. For the two-dimensional convolution used here with $2n_x \times n_y$ Fourier modes, implicit dealiasing requires an extra $n_x \times n_y$ complex values as work memory, whereas conventional zero-padding dealiasing requires 2.5 times that amount.

A speed comparison for various problem sizes $N \times N$ is shown in Figure 14. Both the implicit and explicit simulations performed 1000 predictor-corrector time-steps solving the 2D Navier–Stokes equations, and were single-threaded. The problem sizes were chosen to be optimal for each method. As can be seen in the figure, the implicit method is about twice as fast as the explicit method. The implicit method also uses less memory. In Figure 15, we perform the same comparison, this time using four threads for both the implicit and explicitly dealiased cases.

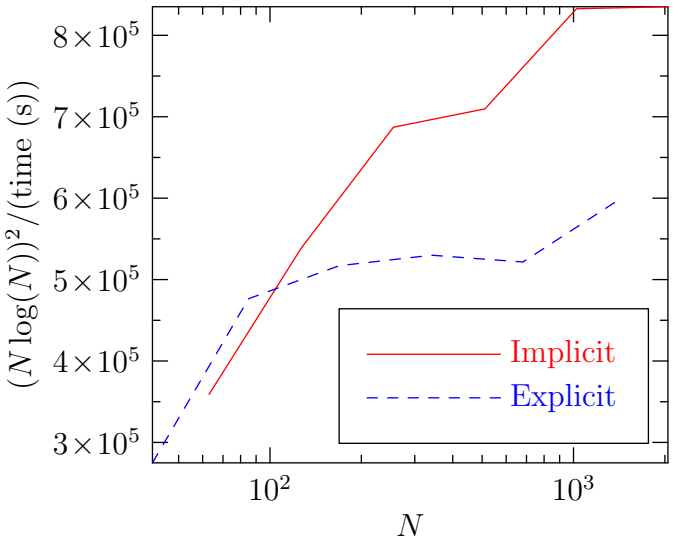

Fig. 14: Timing results for explicit and implicit padding for pseudospectral simulations. The problem size N, which was chosen to be the optimal problem size for each method, is shown on the horizontal axis. Both implementations are from `FFTW++` and use one thread.

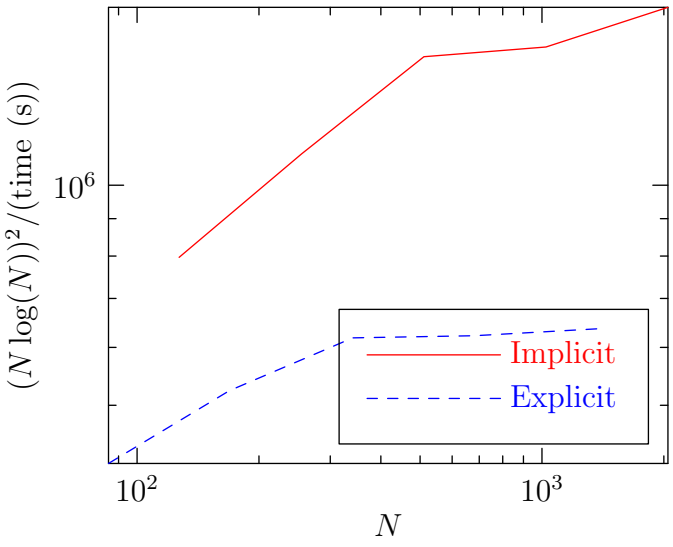

Fig. 15: Timing results for explicit and implicit padding for pseudospectral simulations. The problem size N, which was chosen to be the optimal problem size for each method, is shown on the horizontal axis. Both implementations are from `FFTW++` and use one four threads.

VII. Conclusion and Future Work

A. Conclusion

In this article we reviewed the technique of implicit dealiasing FFT-based convolutions and compared it to conventional zero-padding techniques for a variety of types of convolutions. We looked at applications of convolutions to signal processing, data and image analysis, and pseudospectral simulations, and compared the performance of the implicitly dealiased convolutions in `FFTW++` with `Python`-based and `FFTW`-based implementations of conventional zero-padded dealiasing.

Implicitly dealiased convolutions are currently only implemented for complex data, with and without Hermitian symmetry, and some of the applications that we considered use real-valued data. In these cases, the `FFTW++` implementation has a disadvantage: the computational complexity and memory requirements are double what they could be if real/complex transforms were used. Despite this handicap, the complex convolutions of `FFTW++` were approximately as fast as the real convolutions in other implementations, and `FFTW++` was

faster when using multiple threads. When comparing complex convolutions to complex convolutions, FFTW++ was approximately twice as fast when using one thread, a result which was further improved when using multiple threads.

The FFTW++ library also allows various performance advantages by allowing the user to pass a pointer to a function which reduces the computational cost and memory requirements in cases such as autoconvolution or autocorrelation. Libraries such as scipy compute binary convolutions, and monary operations such as autoconvolution or autocorrelation are computed by calling the binary convolution routine, which is inefficient; the use of function points circumvents this problem in FFTW++.

The FFTW++ library also allows one to compute $2/3$ padded convolutions on complex Hermitian-symmetric data with general post-transform multiplication operators, which is useful for pseudospectral simulations of nonlinear partial differential equations. The performance of implicitly dealiased convolutions on complex, Hermitian-symmetric data was compared to conventional zero-padding techniques in the context of a pseudo-spectral simulation of the two-dimensional Navier–Stokes equations, and it was observed that the implicitly dealiased version had a significant performance benefit over conventional padding in terms of both computation time and memory requirements.

B. Future Work

Implicitly dealiased convolutions are useful tools which offer performance advantages over other similar methods. In this article, we explored the performance on shared-memory architectures. For very large problems, a distributed memory architecture must be considered. For example, simulations of magnetohydrodynamic turbulence can have 1024^3 data points, which simply does not fit in the memory of most shared memory architectures. In addition, a distributed-memory implementation would benefit from reduced communication costs stemming from the reduced memory footprint of implicitly dealiased convolutions. In addition to shared-memory architectures, GPU or MIC based implementation could be very useful, ideally using the OpenCL programming language to take advantage of as many computing platforms as possible. These implementations will require a re-working of the basic algorithm and significant implementation costs.

This article also highlights the need to develop implicitly dealiased convolution on real-valued data. The multi-dimensional convolutions on complex data available currently in FFTW++ have performance similar to the conventional convolutions on real data. By using complex-to-real and real-to-complex FFTs, implicitly dealiased convolutions on real data will theoretically be twice as fast as alternative implementations while reducing memory requirements.

In addition to increased performance, the implementation in FFTW++ allows the user to compute a larger variety of convolution types, from monary to binary to general n-ary convolutions, and on more general data types. We expect that implicitly dealiased convolutions will become the standard technique for computing FFT-based convolutions.

REFERENCES

[1] C. F. Gauss, "Nachlass: Theoria interpolationis methodo nova tractata," in *Carl Friedrich Gauss Werke*. Göttingen: Königliche Gesellschaft der Wissenschaften, 1866, vol. 3, pp. 265–327.

[2] J. W. Cooley and J. W. Tukey, "An algorithm for the machine calculation of complex Fourier series," *Mathematics of Computation*, vol. 19, no. 90, pp. 297–301, April 1965.

[3] J. C. Bowman and M. Roberts, "Efficient dealiased convolutions without padding," *SIAM J. Sci. Comput.*, vol. 33, no. 1, pp. 386–406, 2011.

[4] ——, "FFTW++: A fast Fourier transform C^{++} header class for the FFTW3 library," http://fftwpp.sourceforge.net, 2010.

[5] S. A. Orszag, "Elimination of aliasing in finite-difference schemes by filtering high-wavenumber components," *Journal of the Atmospheric Sciences*, vol. 28, p. 1074, 1971.