FFTX and SpectralPack: A First Look

Franz Franchetti, Daniele G. Spampinato, Anuva Kulkarni, Doru Thom Popovici, Tze Meng Low
Electrical and Computer Engineering Department
Carnegie Mellon University
Pittsburgh, PA, USA
{*franzf, spampinato*}*@cmu.edu, anuvak@andrew.cmu.edu,* {*dpopovic, lowt*}*@cmu.edu*

Michael Franusich
SpiralGen, Inc.
Pittsburgh, PA, USA
mike.franusich@spiralgen.com

Andrew Canning, Peter McCorquodale, Brian Van Straalen, Phillip Colella
Lawrence Berkeley National Laboratory
Berkeley, CA, USA
{*acanning, pwmccorquodale, bvstraalen, pcolella*}*@lbl.gov*

***Abstract*—We propose FFTX, a new framework for building high-performance FFT-based applications on exascale machines. Complex node architectures lead to multiple levels of parallelism and demand efficient ways of data communication. The current FFTW interface falls short in maximizing performance in such scenarios. FFTX is designed to enable application developers to leverage expert-level, automatic optimizations while navigating a familiar interface. FFTX is backwards compatible to FFTW and extends the FFTW Interface into an embedded Domain Specific Language (DSL) expressed as a library interface. By means of a SPIRAL-based back end, this enables build-time source-to-source translation and advanced performance optimizations, such as cross-library calls optimizations, targeting of accelerators through offloading, and inlining of user-provided kernels. We demonstrate the use of FFTX with the prototypical example of 1D and 3D pruned convolutions and discuss future extensions.**

***Keywords*-FFT; exasclale; code generator; high-performance;**

I. Introduction

The Discrete Fourier Transform (DFT) — and in particular its implementation using Fast Fourier Transform (FFT) [1], [2] algorithms — is a fundamental component for the design of scientific applications suitable for the emerging exascale computing ecosystem. Application domains include material science, chemistry, molecular dynamics, and cosmology. Some examples are FFT-based differential equation solvers to compute properties of materials such as stress and strain [3], simulation of incompressible flows [4], plane wave based electronic structure methods [5] [6] and particle-in-cell (PIC) codes [7].

Capturing application-specific properties of the FFTs is very important to enable high-performance execution on emerging platforms. For example, applications where a large subset of the inputs or outputs is either set to zero or not computed at all should exploit the zero structures to reduce data movements within the memory hierarchy on a node or across the network. The structure of multidimensional FFTs provide opportunities for parallelism at multiple levels — SIMD, threads, accelerators, and distributed systems. The best strategy for exploiting these opportunities strongly depends on the details of the use case and of the computing platform, as well as tradeoffs with other needs of the application in which the transforms are embedded.

Conventional FFT-based implementation approach and its limits. The implementation strategy for most of today's large science applications that depend on FFTs constists in transforming multidimensional problems into a sequence of 1D FFT calls, with the latter being performed by a library. Over the last decade or so the FFTW API became the de-facto standard FFT interface [8], [9], [10]. Vendors that provide FFT libraries like Intel, Cray, and IBM may still provide their own proprietary interface for backwards compatibility reasons, but all current vendor high-performance libraries, including Intel MKL [11], IBM ESSL [12], and Nvidia cuFFT [13], implement (at least a subset) of the FFTW interface. Thus, FFTW defines the standard FFT library interface and oftentimes is considered a key component of today's applications.

However, the approach of building up high-performance implementations out of calls to 1D FFTW kernels is breaking down on the current and emerging HPC platforms, for two reasons: First, node architectures have become more complex. Multiple cores and accelerators lead to multiple levels of parallelism, including threading and SIMD/SIMT. In addition, there are on-node complex memory hierarchies that are to varying extents user-controlled, and across which it is expensive to move data. This leads to more complex mappings of multidimensional FFT-based applications to the core 1D FFTs. Some of these are simply not expressible in the current FFTW interface; others can be expressed, but with significant programming effort, and often below the theoretically-predicted performance due to unexpected and opaque behavior of the FFT library software. Second, FFTW is no longer supported. The system comprises a high-level domain-specific language, a symbolic transformation/code

generation system, and an autotuning infrastructure that were essential in developing a general-purpose FFT high-performance library on single-processor architectures available between mid-1990s and mid-2000s. However, current support of FFTW is limited to volunteer contributions. As a result, the extensions to support new node architectures are more brittle and provide less coverage.

The FFTX interface. To overcome the limitations discussed above, we propose FFTX, a new framework for building FFT-based applications. In particular, FFTX provides: (1) A backwards-compatible approach that builds on the FFTW interface but extends it to enable extraction of high-performance on exascale machines; and (2) an evolutionary approach that enables applications to leverage higher level than 1D FFT building blocks and enables cross-block optimization.

SpectralPack. Eventually, the insights provided by opening up the design/tuning space between FFTX and full exascale applications will lead to new ways of designing them to obtain high-performance. Ultimately, our goal is to leverage such a co-design process releasing integrated FFT-based packages as a library, called SpectralPack. SpectralPack will provide a set of template computations covering a variety of spectral applications. In this paper we show the composition of a plan for computing a 3D convolution for the solution of constant-coefficient PDEs using the method of Green's functions as a prototypical example.

This paper is organized as follows. First, we discuss related work in Section II. Next, in Section III, we describe more in details the FFTX interface and in Section IV we provide early examples of applications. Finally, we conclude by describing the current and future work behind FFTX in Section V and draw our conclusions in Section VI.

II. Related Work

As discussed in Section I, the current status of FFT libraries and FFTW poses a significant risk to exascale application development. We now detail the alternatives to our proposed plan and the associated risks.

Rely on FFTW. To the best of our knowledge, the original development team of FFTW does not actively develop FFTW. Except for very minor fixes to the last minor revision (FFTW 3.3) in 2011, only a small number of fixes occurred since (with FFTW 3.3.7 the latest). FFTW continues to work decently on a number of (small core count) multicore CPUs with narrow SIMD vectors and small MPI node counts. Performance becomes an issue for some microarchitectures with wider SIMD vectors, large core counts, more than 32 MPI ranks, batch FFTs, and many important cases supported by the FFTW guru interface. The FFTW benchFFT webpage is outdated with a 2004/2006-era Pentium 4s and Xeons being the most modern profiled CPUs. FFTW does not support accelerators and GPUs. A major redesign and rewrite would be required to produce an FFT version that takes advantage of current and expected exascale machines. The trade-off of code size and complexity vs. performance makes it hard to add support for modern accelerators and advanced architectural features in a way that provides high-performance. This can be seen in the experimental Cell BE FFTW version and in FFTWs SIMD vectorization. Due to the complexity of the system only a small number of people who can understand it well enough could redesign and maintain it. Overall, one cannot expect that FFTW will be updated for the needs of exascale software, either by the original authors or the open source community.

Table I
FFTW library interface supported by commercial vendor libraries. Missing functionalities are denoted as "No" or "Partial". Routines classified as "Mixed" are supported but not all sizes are optimized to the same extend. We denote higher dimensional FFT with nD FFT.

FFTW Features	Intel MKL	IBM ESSL	Nvidia cuFFT
Complex DFTs - Including Real-to-Complex and Complex-to-Real			
1D, 2D, 3D	Mixed	Mixed	Mixed
nD FFT, $n > 3$	Mixed	No	No
Batched FFT	Mixed	Partial	Mixed
Guru Interface	No	No	Partial
Real DFTs			
1D, 2D, 3D	Partial	Partial	No
nD FFT, $n > 3$	No	No	No
Batched FFT	No	No	No
Guru Interface	No	No	No
DCTs and DSTs	Mixed	Partial	No

Rely on vendors. FFTW has become the de-facto standard FFT interface in HPC. Intel (with its Math Kernel Library, MKL [11], and the Cluster MKL), IBM (with ESSL [12] and PESSL), and Nvidia (with cuFFT [13]) maintain HPC FFT libraries. AMD and Cray use FFTW as their FFT library. Both MKL and ESSL provide their own native interface and implement a mapping to parts of the FFTW interface. Neither (Cluster) MKL nor (P)ESSL nor cuFFT implement the full FFTW interface, and implementation/optimization quality varies. There exist some highly optimized routines (e.g., complex 2-power FFTs) while other kernels use some translation routine and can incur (very) high overheads. Performance can vary by 2x to 10x or more. This means a number of routines run at 10% of the efficiency of 2-power FFTs. Table I below provides a detailed assessment of the interface of vendor FFT libraries. Significant portions of the FFTW interface are not supported by commercial vendor libraries. These missing functionalities are denoted as "No" or "Partial" in the table above. Routines that are supported by commercial vendor libraries are classified as "Mixed" when not all sizes are optimized to the same extent.

This lack of deployment of a fully-featured FFT software stack on the part of the hardware vendors is unlikely to change. FFT libraries do not carry the same weight as BLAS libraries for vendors since FFTs do not play any role for

the Top500 ranking, and the HPC Challenge ranking that included FFTs did not have the hoped-for impact. Further, the optimization space for FFTs is much larger than for numerical linear algebra as the algorithm strongly depends on the problem size and properties, requiring a specialized software and staffing infrastructure that the hardware vendors are unlikely to support.

III. FFTX Overview

FFTX provides two key components that we describe more in details in this section: A library interface and a code generator backend.

A. FFTW as DSL

To enable higher-level optimizations, a faithful embedded DSL representation of high-level mathematical description of algorithm/implementation space is necessary. FFTX extends the FFTW Interface into an embedded DSL, expressed as a library interface. The FFTX interface is backwards compatible to FFTW 2.X and 3.X so that legacy code using FFTW runs unmodified and gains substantially on hardware to which FFTX has been ported. The gains can be substantial but not reach full potential when only using legacy mode. To express those additional opportunities for performance, FFTX provides a small number of new features beyond the FFTW interface to express algorithmic features such as futures/delayed execution, offloading, data placement, callback kernels, and sparsity of inputs or outputs. Such changes have the potential to extract much higher performance than standard FFTW calls since higher level operations and new hardware features can be addressed. This interface is designed as an embedded DSL, for which we provide a standard C/C++ reference library implementation that enables rapid assessment of the interface by applications developers.

B. A code generation backend

FFT-based application kernels implemented using the extended FFTW interface described above are treated as specifications. This enables build-time source-to-source translation and advanced performance optimizations, such as cross-library call and cross library optimization, targeting of accelerators through off-loading, and inlining of user-provided kernels. Our approach allows for fine control over resource expenditure during the optimization. Users can control compile-time, initialization-time, invocation time optimization resources if they need to.

The core code generation, symbolic analysis, and autotuning software for this project will be based on SPIRAL [14], [15], [16]. SPIRAL automatically maps computational kernels across a wide range of computing platforms to highly efficient code, and proves the correctness of the synthesized code [17]. This addresses two fundamental problems that software developers are faced with: performance portability across the ever-changing parallel platforms, and verifiable correctness of sophisticated floating-point code. The problem is attacked as follows: A formal framework captures computational algorithms, computing platforms, and program transformations of interest, using a unifying mathematical formalism called operator language (OL) [18]. Then the problem is cast as synthesizing highly optimized computational kernels for a given machine as a strongly constrained optimization problem solved by a multi-stage rewriting system. Since all rewrite steps are semantics-preserving operations, SPIRAL's approach allows to formally prove the equivalence between the kernel specification and the synthesized program. The SPIRAL approach has many of the same structural components as FFTW a high-level DSL, symbolic analysis, code generation, and autotuning. However, the approach used by SPIRAL generates new source code for both the FFT calls and the glue code (e.g. pointwise operations and data motion) in an FFT-based application.

In the next section we will show how the FFTX interface can be used to describe a simple example of a pruned convolution and preliminary code generated by the SPIRAL-based backend.

IV. Example Applications

Constant-coefficient PDEs are often solved using the method of Green's functions, which involves convolution of the input signal with the Green's function. This convolution is performed in the Fourier domain to reduce complexity.

Large parallel FFTs are memory-bound and hence become a communication bottleneck on shared memory systems/GPUs. Depending on the problem, if a large number of inputs or outputs is zero or not computed, pruning can be used to exploit these properties of the data and have a large reduction in the communication overhead.

We illustrate the use of FFTX for a simple 1D pruned convolution example. We begin by introducing how a program is structured when using FFTX.

A. Structure of an FFTX-based application

Figure 1 shows the structure of an application that make use of the FFTX interface. The *fftx.h* header file contains the definition of all types, macros, and functions necessary to write up an FFTX application. The Appendix lists the mathematical definition of the functions used in this example.

The calls to the *fftx_init* and *fftx_shutdown* functions identify an FFTX region in the program. They set up the environment with appropriate options, such as declaring that FFTX should operate in high-performance mode (i.e., enabling symbolic analysis, code generation, and autotuning in the backend) as shown in Fig. 1. Next, comes the definition of the computation. Similarly to FFTW, this is achieved by first building a plan, i.e., a sequence of computational and data movement steps that, when executed,

```
1  #include <fftx.h>
2
3  fftx_plan pruned_real_convolution_plan(...) {
4    // produces an fftx_plan (see Fig. 2)
5  }
6
7  int main() {
8    // declare input, output, and G[k]
9    fftx_real *in, *out;
10   fftx_complex *G_k;
11   fftx_plan p;
12
13   // initialize FFTX
14   fftx_init(FFTX_HIGH_PERFORMANCE);
15   // alloc/init input, G[k], and output
16   ...
17   // build FFTX plan
18   p = pruned_real_convolution_plan(in, out, G_k,
19       n, n_in, n_out, n_freq);
20   // execute once the pruned convolution
21   fftx_execute(p);
22   // cleanup
23   ...
24   // shut down FFTX
25   fftx_shutdown();
26 }
```

Figure 1. Structure of an FFTX application.

applies the computation to the application input to obtain its output. For example, Fig. 1, lines 22-23, show the use of function *pruned_real_convolution_plan* to build the plan for computing a pruned convolution. We will soon discuss its content. Here, we only need to stress two apsects: 1) Writing such functions is the only point where the application expert has to apply her knowledge, 2) The resulting plan is used as a specification by FFTX and eventually transformed in highly optimized code that computes it. A plan once built can be executed once or many times by simply passing the plan to the *fftx_execute* function. Next, we focus on how the FFTX interface can be used to build a computational plan for 1D pruned convolution.

B. 1D pruned convolution

This example considers an N-point pruned real convolution in 1D using FFTX. For a discrete input signal $\vec{\rho}$, we assume only first N_s elements are non-zero. For the output, only the last N_d elements are requested.

We select the identity function as a simple convolution kernel $\vec{G}$.

The convolution operation can be written as

$$(\vec{\rho} * \vec{G})_x = \sum \rho_{x-x'} G_{x'} \tag{1}$$

The Fourier transform of the convolution kernel is real-valued and an even function. Using $\tilde{.}$ to denote the Fourier transform and k to denote the wavenumber,

$$\tilde{G}_k = \frac{1}{N}, k \in [0, \cdots, N-1] \tag{2}$$

A real to complex FFT is performed on $\vec{\rho}$ to get the conjugate even signal $\tilde{\rho}$. Because of the conjugate even symmetry, we only need to store half of this signal. Similarly, $\tilde{G}$ is symmetric so once more, only half of the signal is stored. Hence, for both signals we store $[0 \cdots \frac{N}{2}]$ points where N is even. After a pointwise multiplication of $\tilde{\rho}$ and $\tilde{G}$, inverse FFT transform is computed to get $N_k = \frac{N}{2} + 1$ complex elements in frequency domain.

Plan composition using FFTX. Implementing the pruned 1D convolution in FFTX requires the the construction of the plan shown in Fig. 2.

The overall plan is composed of a sequence of subplans each performing one of the five steps described above. In particular, a zero vector of length N is created (Fig. 2, line 37) and used to created the zero padding of the input signal of length N_s (Fig. 2, l. 40-41). Next, the 1D real DFT is computed (Fig. 2, l. 45-46) followed by a pointwise multiplication with $\hat{G}$ (Fig. 2, l. 50-53). Finally, the inverse DFT is taken (Fig. 2, l. 57-58) and the last N_d elements from its output signal are stored in the output vector (Fig. 2, l. 61). Note, how the different subplans depend in general on a number of parameters required to describe the data involved in a given computation. These descriptors include information about the dimensionality of the data (i.e, its rank *r_k*) and an *fftx_iodim* or *fftx_iodimx* object with information about strides and offsets of the input and output data. More specifically, an M-dimensional cube is described by an array of M such objects. For example, object *p_d*, used in Fig. 2, l. 45, is of type *fftx_iodim*. This is the analogous of FFTW's *fftw_iodim* and describes that the DFT is computed on N points with unit input and output stride (see its definition in Fig. 2, l. 20). The object *f_dx* used in Fig. 2, l. 51, is of type *fftx_iodimx*. This type extends *fftx_iodim* with additional information such as the offset of the data and also includes the description of third multidimensional cube that in the call to *fftx_plan_guru_pointwise_c2c* is used to describe access to the Green's function. The definition of *fftx_iodimx* is given in Fig. 3.

We mentioned that the data associated to the Green's function is described by object *f_dx* along with the input signal $\hat{\rho}$. The actual pointwise scaling operation between the two complex vectors is described by the *cplx_scaling* parameter passed in Fig. 2, l. 52. This is a function composed by the FFTX pointwise helper macros shown in Fig. 4.

These and others provided by the FFTX interface define a simple two-operand language similar to x86 ASM that can be used by the Spiral backend to optimize the computation across subplans. A preliminary example of code generated for the entire application using the SPIRAL-based back end is provided in Fig. 5.

C. 3D pruned convolution

Now we extend the above 1D example to a 3D case. For a 3D input signal ρ, we assume only first N_s elements are

```
#define FFTX_MODE (FFTX_ESTIMATE | FFTX_OBSERVE)
#define FFTX_MODE_SUB (FFTX_MODE | FFTX_SUBPLAN)

fftx_temp_real tmp1, tmp4;
fftx_temp_complex tmp2, tmp3;

fftx_plan pruned_real_convolution_plan(
            fftx_real *in, fftx_real *out,
            fftx_complex *G_k,
            int N, int N_s, int N_d, int N_k) {

  int rk = 1, b_rk = 0, // rank 1 + no batch
    n_subp = 5;  // requires 5 FFTX subplans

  // intermediate sub-plans and top-level plan
  fftx_plan plans[5], p;

  // FFTX iodim definitions for 1D + pruning
  // zero-padding to N real elements at stride 1
  fftx_iodim p_d = { N, 1, 1 },
    // N/2+1 complex elements in frequency domain
    f_d = { N_k, 1, 1 },
    // no batching
    b_d = { 1, 1, 1 };

  // input: N_s real elements at unit stride
  fftx_iodimx i_dx = { N_s, 0, 0, 0, 1, 1, 1 },
    // output: N_d real elements at unit stride
    o_dx = { N_d, N - N_d, 0, 0, 1, 1, 1 },
    // n/2+1 complex elements in frequency domain
    f_dx = { N_k, 0, 0, 0, 1, 1, 1 },
    // no batching
    b_dx = { 1, 0, 0, 0, 1, 1, 1 };

  // create zero-initialized rank-dimensional
  // temporary for zero-padding the input
  tmp1 = fftx_create_zero_temp_real(rk, &p_d);

  // copy rank-D data into zeroed temporary
  plans[0] = fftx_plan_guru_copy_real(rk,
        &i_dx, in, tmp1, FFTX_MODE_SUB);

  // RDFT on the padded data
  tmp2 = fftx_create_temp_complex(rk, &f_d);
  plans[1] = fftx_plan_guru_dft_r2c(rk, &p_d,
    b_rk, &b_d, tmp1, tmp2, FFTX_MODE_SUB);

  // pointwise operation
  tmp3 = fftx_create_temp_complex(rk, &f_d);
  plans[2] = fftx_plan_guru_pointwise_c2c(rk,
        &f_dx, b_rk, &b_dx, tmp2, tmp3, G_k,
        (fftx_callback)cplx_scaling, // See Fig. Z
    FFTX_MODE_SUB | FFTX_PW_POINTWISE);

  // iRDFT on the scaled data
  tmp4 = fftx_create_temp_real(rk, &p_d);
  plans[3] = fftx_plan_guru_dft_c2r(rk, &p_d,
        b_rk, &b_d, tmp3, tmp4, FFTX_MODE_SUB);

  // copy out the rank-D data of interest
  plans[4] = fftx_plan_guru_copy_real(rk, &o_dx,
        tmp4, out, FFTX_MODE_SUB);

  // create the top level plan.
  p = fftx_plan_compose(n_subp, plans,
        FFTX_MODE);

  // plan used with fftx_execute() - See Fig. 1
  return p;
}
```

Figure 2. Plan for computing the pruned 1D FFT using FFTX.

```
typedef struct {
  int n,       // Length along dimension

      iofs,    // Input, output and data stride
      oofs,  // Allow shift or reversal
      dofs,  //

      is,      // Input stride
      os,      // Output stride
      ds;     // Independent data stride,
             // used to capture symmetries
} fftx_iodimx;
```

Figure 3. The *fftx_iodimx* descriptor.

```
// pointwise scaling function:
// complex multiply by symbol
void cplx_scaling(fftx_complex *in,
     fftx_complex *out,
     fftx_complex *data) {

  FFTX_COMPLEX_TEMP(t);
  FFTX_COMPLEX_MOV(t, in);
  FFTX_COMPLEX_MULT(t, data);
  FFTX_COMPLEX_MOV(out, t);

}
```

Figure 4. Pointwise scaling function composed by FFTX pointwise helper macros.

non-zero in each dimension. For the output, only the last N_d elements are requested in each dimension.

Plan composition using FFTX. The FFTX implementation for the 3D pruned convolution requires exactly the same sequence of plans used for the 1D case and shown in Fig. 2. The only difference compared to the 1D case is the definition of *fftx_iodim* and *fftx_iodimx* objects for 3D cubes as shown in Fig. 6.

Operations this time require accessing the data along three different dimensions. These accesses are described by 3D arrays of *fftx_iodim* and *fftx_iodimx* objects, one for each dimension. For example, the array *o_dx* defined in Fig. 6, l. 27-31, is used to read a corner of size N_d^3 elements out of a cube of size N^3 starting from offset $(N - N_d, N - N_d, N - N_d)$.

D. Poisson solver with Method of Local Correction

Consider the problem of solving Poisson's equation in free space. This solution is required in many scientific computing applications such as cosmological simulations, molecular dynamics and methods for solving viscous incompressible flows [19], [20], [4].

$$\Delta(\Phi) = \rho,$$

where $\rho : \mathbb{R}^3 \to \mathbb{R}$ is the charge distribution with support on a compact set $D = \text{supp}(\rho) \subset \mathbb{R}^3$ and Δ is the Laplace operator. We seek a solution $\Phi : \mathbb{R}^3 \to \mathbb{R}$ such that far-field

```
void prconv8_3_5(double *Y, double *X,
    double S2[10]) {
    double a149, a150, a151, a152, s205,
    s206, s207, s208, s209, s210, s211,
    s212, s213, s214, s215, s216, s217,
    s218, s219, s220, s221, s222, t77,
    t78, t79, t80;
    s205 = (*(X) + *((X + 2)));
    s206 = (*(X) - *((X + 2)));
    s207 = (*((X + 1)) + *((X + 3)));
    s208 = (*((X + 1)) - *((X + 3)));
    s209 = (*(S2)*(s205 + s207));
    s210 = (*((S2 + 8))*(s205 - s207));
    a149 = (0.70710678118654757**((X + 1)));
    a150 = (0.70710678118654757**((X + 3)));
    s211 = (a149 - a150);
    s212 = (a149 + a150);
    t77 = (*(X) + s211);
    t78 = (*((X + 2)) + s212);
    t79 = (*(X) - s211);
    t80 = (s212 - *((X + 2)));
    s213 = ((*((S2 + 2))*t77)
            + (*((S2 + 3))*t78));
    s214 = ((*((S2 + 3))*t77)
            - (*((S2 + 2))*t78));
    s215 = ((*((S2 + 6))*t79)
            + (*((S2 + 7))*t80));
    s216 = ((*((S2 + 7))*t79)
            - (*((S2 + 6))*t80));
    s217 = (s209 + s210);
    s218 = (s209 - s210);
    a151 = (0.70710678118654757*(s213 - s215));
    a152 = (0.70710678118654757*(s214 + s216));
    s219 = (2.0*((*((S2 + 4))*s206)
            + (*((S2 + 5))*s208)));
    s220 = (2.0*((*((S2 + 5))*s206)
            - (*((S2 + 4))*s208)));
    s221 = (s218 + s220);
    s222 = (2.0*(a151 + a152));
    *(Y) = (s221 - s222);
    *((Y + 1)) = ((s217 + s219)
            - (2.0*(s213 + s215)));
    *((Y + 2)) = ((s218 - s220)
            - (2.0*(a151 - a152)));
    *((Y + 3)) = ((s217 - s219)
            + (2.0*(s214 - s216)));
    *((Y + 4)) = (s221 + s222);
}
```

Figure 5. Code generated by the FFTX back end for the plan in Fig. 2 setting $N = 8$, $N_s = 3$, and $N_d = 5$.

```
fftx_plan pruned_real_convolution_plan(...) {
  int rk = 3,           // 3D = rank 3
    b_rk = 0,           // no batch
    n_subp = 5;         // need 5 FFTX subplans
  fftx_plan plans[5];
// intermediate sub-plans
  fftx_plan p;          // top-level plan

  // FFTX iodim definitions for 3D + pruning
  fftx_iodim p_d[] = {
      { N, 1, 1 },
      { N, N, N },
      { N, N*N, N*N }
    },
    f_d[] = {
      { N, 1, 1 },
      { N, N, N },
      { N_k, N*N, N*N }
    },
    // no batching
    b_d = { 1, 1, 1 };

  fftx_iodimx i_dx[] = {
      { N_s, 0, 0, 0, 1, 1, 1 },
      { N_s, 0, 0, 0, N_s, N, 1 },
      { N_s, 0, 0, 0, N_s*N_s, N*N, 1 }
    },
    o_dx[] = {
      { N_d, N-N_d, 0, 0, 1, 1, 1 },
      { N_d, N-N_d, 0, 0, N, N_d, 1 },
      { N_d, N-N_d, 0, 0,
        N*N, N_d*N_d, 1 }
    },
    f_dx[] = {
      { N, 0, 0, 0, 1, 1, 1 },
      { N, 0, 0, 0, N, N, N },
      { N_k, 0, 0, 0, N*N, N*N, N*N }
    },
    // no batching
    b_dx = { 1, 0, 0, 0, 1, 1, 1, };

  ... // As in Fig. 2

  p = fftx_plan_compose(n_subp, plans,
        FFTX_MODE);

  // plan to be used with fftx_execute()
  return p;
}
```

Figure 6. Data access descriptors for computing the pruned 3D FFT using FFTX. The plan used is the same reported in Fig. 2.

behavior is

$$\Phi(\vec{x}) = \frac{Q}{4\pi||\vec{x}||} + o(\frac{1}{||\vec{x}||}) \text{ as } ||\vec{x}|| \to \infty$$
$$Q = \int_D \rho d\vec{x},$$

which can be expressed as convolution with the Green's function

$$\Phi(\vec{x}) = \int_D G(\vec{x} - \vec{y})\rho(\vec{y})d\vec{y} \equiv (G * \rho)(\vec{x}), \qquad (3)$$

where

$$G(\vec{x}) = \frac{1}{4\pi||\vec{x}||_2}. \qquad (4)$$

Several algorithms have been developed for fast solutions of Poisson's equations. A solution for manycore architectures is an algorithm [21] based on the Method of Local Corrections (MLC) [22] and designed for a computationally efficient parallel implementation for 3D gridded data.

The algorithm [21] is a domain-decomposition method based on computing local convolutions with the freespace Green's function on overlapping rectangular domains. The smooth global coupling among the domains is computed using a much coarser (and less computationally expensive) discretization. This is similar to multigrid approaches [23] [24] [25]. Unlike multigrid, though, the method is non-

iterative. After suitable discretization, a key part of the MLC method is Hockney's algorithm, which computes free-space convolution (infinite domain boundary conditions) quickly using the FFT. A solution to the free space convolution is obtained using discrete convolution between ρ and G on a domain that is at double the size of the support of ρ. Hence, the input to the convolution has zero-padded entries. We are only interested in a subset of the output of the convolution due to coarsening grids in MLC. Thus, both input and output have a zero-structure which can be pruned to reduce load/store operations. Additionally, the convolution has real inputs and outputs, hence a real to complex (R2C) forward transform and a complex to real (C2R) inverse transform are used.

The Fourier domain analytic representation of the discrete Green's function corresponding to the Laplacian operator is

$$\tilde{G}_k = \frac{1}{4\pi||k - N\vec{u}||_2^2} \quad \text{if } k \neq N\vec{u} \tag{5}$$

The numerically computed Green's function has the same properties, i.e., it is real and symmetric about the origin in both space and Fourier domains. Therefore, is enough to precompute and store only $1/8^{th}$ (one octant) of the 3D data. The FFTX API can be used to convolve the input and the Green's function using a plan very similar to the one shown in Fig. 6 with the four additional copies of the Green's function octant shown in Fig. 7. The data access descriptors exposed by these copies make use of both negative and positive strides to describe the symmetry of the discrete Green's function in (5).

```
// FFTX data access descriptors.
// Access is to four octants of a symmetric cube.
// Cube size is N^3 and M = N/2.
fftx_iodimx oct00[] = {
  { M+1, 0, 0, 0, 1, 1, 1 },
  { M+1, 0, 0, 0, M+1, 2*M, 1 },
  { M+1, 0, 0, 0, (M+1)*(M+1), 4*M*M, 1} },
  oct01[] = {
  { M-1, M-1, M+1, 0, -1, 1, 1 },
  { M+1, 0, 0, 0, M+1, 2*M, 1 },
  { M+1, 0, 0, 0, (M+1)*(M+1), 4*M*M, 1} },
  oct10[] = {
  { M+1, 0, 0, 0, 1, 1, 1 },
  { M-1, M-1, M+1, 0, -(M+1), 2*M, 1 },
  { M+1, 0, 0, 0, (M+1)*(M+1), 4*M*M, 1} },
 oct11[] = {
  { M-1, M-1, M+1, 0, -1, 1, 1 },
  { M-1, M-1, M+1, 0, -(M+1), 2*M, 1 },
  { M+1, 0, 0, 0, (M+1)*(M+1), 4*M*M, 1} };

...
fftx_temp_complex half_G_k =
  fftx_create_zero_temp_complex(rk, f_d);
plans[2] = fftx_plan_guru_copy_complex(rk, oct00,
  G_k, half_G_k, FFTX_MODE_SUB);
plans[3] = fftx_plan_guru_copy_complex(rk, oct01,
  G_k, half_G_k, MY_FFTX_MODE_SUB);
plans[4] = fftx_plan_guru_copy_complex(rk, oct10,
  G_k, half_G_k, MY_FFTX_MODE_SUB);
plans[5] = fftx_plan_guru_copy_complex(rk, oct11,
  G_k, half_G_k, MY_FFTX_MODE_SUB);
...
```

Figure 7. The four descriptors used to copy half of the symmetric discrete Green's function in (5) when only one of its octants is provided.

V. Current and Future Work

In this section, we discuss added features for optimization for FFTX that could enable the solution of the Maxwell's Equations using pseudo-spectral methods.

Consider the Pseudo-Spectral Analytical Time-Domain (PSATD) method with domain decomposition proposed by [7], which solves Maxwell's equations using pseudo-spectral methods. Domain decomposition is standard for finite-difference solvers but not for spectral solvers. However, this combined new paradigm allows favorable parallel scaling of electromagnetic solvers [26].

In PSATD, Maxwell's equations in Fourier domain from step n to step $n+1$ on staggered grids are:

$$\begin{aligned}\tilde{\mathbf{E}}^{n+1} = C\tilde{\mathbf{E}}^n + iS\hat{k} \times \tilde{\mathbf{B}}^n - \frac{S}{ck}\tilde{\mathbf{J}}^{n+1/2} + \\ i\frac{\hat{k}}{k}\left[\left(\frac{S}{ck\Delta t} - 1\right)\tilde{\rho}^{n+1} + \left(C - \frac{S}{ck\Delta t}\right)\tilde{\rho}^n\right]\end{aligned} \tag{6}$$

$$\tilde{\mathbf{B}}^{n+1} = C\tilde{\mathbf{B}}^n - iS\hat{k} \times \tilde{\mathbf{E}}^n + i\frac{1-C}{ck}\hat{k} \times \tilde{\mathbf{J}}^{n+1/2} \tag{7}$$

where $\tilde{a}$ is the Fourier transform of the quantity a, $\mathbf{E}$ is electric field, $\mathbf{B}$ is magnetic field, $\mathbf{J}$ is current density and ρ is charge density. $\vec{k}$ is the wave vector of length $k = \sqrt{k_x^2 + k_y^2 + k_z^2}$, and $\hat{k} = \vec{k}/k$. c is speed of light and Δt is the time step. S = sin($kc\Delta t$) and C = cos($kc\Delta t$). Δt and is the time step and n is the time index. This system of equations is written as a linear system as follows, where M_s is referred to as a transformation matrix.

$$\begin{bmatrix} \tilde{\mathbf{E}}^{n+1} \\ \tilde{\mathbf{B}}^{n+1} \end{bmatrix} = M^s \times \begin{bmatrix} \tilde{\mathbf{E}}^n \\ \tilde{\mathbf{B}}^n \\ \tilde{\mathbf{J}}^{n+1/2} \\ \tilde{\rho}^n \\ \tilde{\rho}^{n+1} \end{bmatrix} \tag{8}$$

Since FFTs do not parallelize well, local FFTs of electrical and magnetic fields are computed instead of a large 3D global FFT in each domain. The domains include guard cells, which are exchanged between neighboring domains in a communication step in order to reconstruct the full solution. The simulation evolves Maxwell's equations in each time step indexed by n, and update of $\mathbf{E}$ and $\mathbf{B}$ consists of more operations than just pointwise multiplication. Each update involves multiplication of the Fourier transforms with a transformation matrix depending on $\vec{k}$. Staggered grids require complex conjugates of $\vec{k}$ in the transformation matrix. Additionally, a communication step of guard cells

between domains is necessary to reconstruct the global field.

For such problems involving more complex operations on the Fourier transformed signal, we see the need to develop the FFTX interface as a more general concept than just for pointwise operations. Introducing the ability to handle more general scenarios such as tensor contractions would be of significant value for optimizing parallel FFTs in exascale simulations that use PSATD, such as WarpX, an exascale computing platform for beam plasma simulations [27].

VI. Conclusion

FFTs play a prominent role in the design of applications for emerging exascale systems. These systems exspose unprecedented high-performance opportunities if applications can take advantage of their complex multiple levels of parallelism. The goal of FFTX is to provide a framework for users to maximize performance and productivity when developing FFT-based applications. In this paper, we have introduced the FFTX interface and discussed its use for building a prototypical pruned convolution as an example. We have also shown how the SPIRAL-based backend of FFTX can generate code for the whole application by treating the FFTX plan composition as an input specification. Finally, we addressed current and future extensions to the FFTX interface which are primarily driven by requirements of other important applications of exascale interest, such as Poisson solvers and pseudo-spectral analytical time-domain in the context of beam plasma simulations. Ultimately the goal is the development of SpectralPack, a comprehensive set of integrated FFT-based appplication template plans unified in a single high-performance framework.

Appendix
Mathematical Definition of FFTX Functions

In this appendix we provide the mathematical semantics of the functions used in the application examples presented in Section IV. Next we fix the notation used in our formulations.

Standard basis We define the standard basis in $\mathbb{R}^n$,

$$\mathrm{e}_i^{n\times 1} = (\delta_{ij})_{j=0,\ldots,n-1} \in \mathbb{R}^n, \quad \delta_{ij} = \begin{cases} 0, & i \neq j \\ 1, & i = j \end{cases}. \tag{9}$$

In (9), δ_{ij} is the Kronecker delta, while $\mathrm{e}_i^{n\times 1} \in \mathbb{R}^{n\times 1}$ is the ith column standard basis vector. Similarly, we denote with $\mathrm{e}_i^{1\times n} \in \mathbb{R}^{1\times n}$ the respective row vector.

FFTX functions In general, FFTX operations functions are set up via planner descriptors which are eventually executed. Formally, for the application of an operator A to a source vector $\vec{x}$ that yields a destination vector $\vec{y}$ we write

$$\vec{y} \doteq A\vec{x}.$$

The original data in $\vec{y}$ is untouched if the zero to be assigned to y_i is an entry of a standard basis vector describing the operation, and not a zero due to input data and computation. For scalars a and b

$$(a \doteq b) := \begin{cases} a, & \text{if } b = \delta_{ik}, i \neq k \\ b, & \text{otherwise} \end{cases}$$

and vector assignment generalizes this idea. Further definitions used in this appendix are listed in Table II.

Table II
Preliminary definitions. $(<\text{obj}>, <\text{ofs}>, <\text{s}>) \in \{(\text{in}, \text{iofs}, \text{is}), (\text{out}, \text{oofs}, \text{os}), (\text{data}, \text{dofs}, \text{ds})\}$.

$$\begin{aligned}
r &= \text{rank} \\
h &= \text{howmany_rank} \\
\vec{n} = (n_0, \ldots, n_{r-1}) &= \text{Map}(\text{dims}, \text{i} \mapsto \text{i.n}) \\
\vec{m} = (m_0, \ldots, m_{h-1}) &= \text{Map}(\text{howmany_dims}, \ \text{i} \mapsto \text{i.n}) \\
\vec{b}_{<\text{obj}>} &= \text{Map}(\text{dims}, \text{i} \mapsto \text{i.} < \text{ofs} >) \\
\vec{s}_{<\text{obj}>} &= \text{Map}(\text{dims}, \text{i} \mapsto \text{i.} < \text{s} >) \\
\vec{b}_{h,<\text{obj}>} &= \text{Map}(\text{howmany_dims}, \text{i} \mapsto \text{i.} < \text{ofs} >) \\
\vec{s}_{h,<\text{obj}>} &= \text{Map}(\text{howmany_dims}, \text{i} \mapsto \text{i.} < \text{s} >) \\
N &= \prod_{i=0}^{r-1} n_i \\
M_{<\text{obj}>} &= \sum_{i=0}^{h-1} m_i s^{(i)}_{h,<\text{obj}>} \\
\vec{s} &= \left(\prod_{i=0}^{r-2} n_i, \ldots, n_0 n_1, n_0, 1 \right) \\
\vec{x} \in \mathbb{R}^N \text{ (or } \mathbb{C}^N) &= \text{in} \\
\vec{y} \in \mathbb{R}^N \text{ (or } \mathbb{C}^N) &= \text{out} \\
\vec{G} \in \mathbb{R}^N \text{ (or } \mathbb{C}^N) &= \text{data}
\end{aligned}$$

A. Copy Function

The copy function copies a data cube or sub-cube from source to destination. Parameters include the input and output data access descriptors, and the source and target memory locations.

```
fftx_plan fftx_plan_guru_copy_real(int rank,
  fftx_iodimx *dimsx, fftx_real *in,
  fftx_real *out, unsigned flags);
```

An similar function is available for complex data. The semantics of the copy operation is given as follows. Let $\sigma = \langle \vec{b}_{\text{in}}, \vec{s}_{\text{in}}, \vec{b}_{\text{out}}, \vec{s}_{\text{out}} \rangle$, then, the copy operator returned by the planner is defined as

$$\text{Copy}_\sigma^{r,\vec{n}} = \sum_{\vec{j} \in \bigtimes_{i=1}^{r-1} \mathbb{I}_{n_i}} \mathrm{e}^{N\times 1}_{(\vec{b}_{\text{out}}+\vec{j})\cdot\vec{s}_{\text{out}}} \cdot \mathrm{e}^{1\times N}_{(\vec{b}_{\text{in}}+\vec{j})\cdot\vec{s}_{\text{in}}},$$

and invoking the executor performs the operation

$$\vec{y} \doteq \text{Copy}_\sigma^{r,\vec{n}}\, \vec{x}.$$

B. Complex DFT

The complex DFT plan is set up as follows.

```
fftx_plan_t fftx_plan_guru_dft(
  int rank, fftx_iodim *dims,
  int howmany_rank, fftx_iodim *howmany_dims,
  fftx_complex *in, fftx_complex *out,
  int sign, unsigned flags);
```

The semantics of the complex DFT operation are given as follows. Let $\sigma = \langle \vec{s}_{\text{in}}, \vec{s}_{\text{out}}, \vec{s}_{h,\text{in}}, \vec{s}_{h,\text{out}} \rangle$. The operator returned by the planner is defined as

$$\begin{aligned}&\text{DFT}_\sigma^{r,h,\vec{n},\vec{m}} = \\ &\sum_{\vec{k} \in \bigtimes_{i=1}^{h-1} \mathbb{I}_{m_i}} \left(\left(\sum_{\vec{j} \in \bigtimes_{i=1}^{r-1} \mathbb{I}_{n_i}} \text{e}_{(\vec{k} \oplus \vec{j}) \cdot (\vec{s}_{h,\text{out}} \oplus \vec{s}_{\text{out}})}^{M_{\text{out}} \times 1} \cdot \text{e}_{\vec{j} \cdot \vec{s}}^{1 \times N} \right) \right. \\ &\left. \text{DFT}_{n_0 \times \ldots n_{r-1}} \left(\sum_{\vec{j} \in \bigtimes_{i=1}^{r-1} \mathbb{I}_{n_i}} \text{e}_{\vec{j} \cdot \vec{s}}^{N \times 1} \cdot \text{e}_{(\vec{k} \oplus \vec{j}) \cdot (\vec{s}_{h,\text{in}} \oplus \vec{s}_{\text{in}})}^{1 \times M_{\text{in}}} \right) \right)\end{aligned}$$

The multidimensional complex DFT $\text{DFT}_{m \times \cdots \times k \times n}$ can be defined recursively as

$$\begin{aligned}\text{DFT}_{m \times \cdots \times k \times n} &\rightarrow \text{DFT}_{m \times \cdots \times k} \otimes \text{DFT}_n \qquad (10) \\ \text{DFT}_n &= \left[\omega_n^{ij} \right]_{ij},\end{aligned}$$

where ω_n is a primitive nth root of unity. Invoking the operator will perform the operation

$$\vec{y} \doteq \text{DFT}_\sigma^{r,h,\vec{n},\vec{m}}\, \vec{x}.$$

C. Real-to-complex DFT

The real-to-complex DFT plan is set up as follows:

```
fftx_plan_t fftx_plan_guru_dft_r2c(
  fftx_context_t context,
  int rank, const fftx_iodim *dims,
  int howmany_rank, fftx_iodim *howmany_dims,
  fftx_real *in, fftx_complex *out,
  unsigned flags);
```

Let $\sigma = \langle \vec{s}_{\text{in}}, \vec{s}_{\text{out}}, \vec{s}_{h,\text{in}}, \vec{s}_{h,\text{out}} \rangle$. The operator returned by the planner is defined as

$$\begin{aligned}&\text{RDFT}_\sigma^{r,h,\vec{n},\vec{m}} = \\ &\sum_{\vec{k} \in \bigtimes_{i=1}^{h-1} \mathbb{I}_{m_i}} \left(\left(\sum_{\vec{j} \in \bigtimes_{i=1}^{r-1} \mathbb{I}_{n_i}} \text{e}_{(\vec{k} \oplus \vec{j}) \cdot (\vec{s}_{h,\text{out}} \oplus \vec{s}_{\text{out}})}^{M_{\text{out}} \times 1} \cdot \text{e}_{\vec{j} \cdot \vec{s}}^{1 \times N} \right) \right. \\ &\left. \text{RDFT}_{n_0 \times \ldots n_{r-1}} \left(\sum_{\vec{j} \in \bigtimes_{i=1}^{r-1} \mathbb{I}_{n_i}} \text{e}_{\vec{j} \cdot \vec{s}}^{N \times 1} \cdot \text{e}_{(\vec{k} \oplus \vec{j}) \cdot (\vec{s}_{h,\text{in}} \oplus \vec{s}_{\text{in}})}^{1 \times M_{\text{in}}} \right) \right)\end{aligned}$$

Invoking this operator will perform the operation

$$\vec{y} \doteq \text{RDFT}_\sigma^{r,h,\vec{n},\vec{m}}\, \vec{x}.$$

The multidimensional real DFT $\text{RDFT}_{m \times \cdots \times k \times n}$ can be defined recursively by decomposing it into a Kronecker product between the multidimensional DFT in (10) and a 1D RDFT as follows

$$\text{RDFT}_{m \times \cdots \times k \times n} \rightarrow \text{DFT}_{m \times \cdots \times k} \otimes \text{RDFT}_n. \qquad (11)$$

The n-point 1D real DFT in (11) is defined as

$$\begin{aligned}\text{RDFT}_n &= \left(\sum_{\vec{j} \in \mathbb{I}_{n+2}} \text{e}_{\vec{j}}^{(n+2) \times 1} \cdot \left(\text{e}_{\vec{j}}^{1 \times (n+2)} \oplus 0^{(n-2) \times 1} \right) \right) \\ &\quad \overline{\text{DFT}_n} \left(\sum_{\vec{j} \in \mathbb{I}_n} (\text{e}_{\vec{j}}^{n \times 1} \otimes \text{e}_0^{2 \times 1}) \cdot \text{e}_{\vec{j}}^{1 \times n} \right) \\ \text{DFT}_n &= \left[\omega_n^{ij} \right]_{ij},\end{aligned}$$

where ω_n is a primitive nth root of unity and the operator $\overline{(\cdot)}$ maps a complex $r \times c$ matrix M to a real $2r \times 2c$ matrix $\overline{M}$ replacing every entry $u + iv$ of M with $\left[\begin{smallmatrix} u & -v \\ v & u \end{smallmatrix} \right]$.

D. Pointwise Operation

The pointwise operation is set up as shown.

```
fftx_plan fftx_plan_guru_pointwise_c2c(
  int rank, fftx_iodimx *dims,
  int howmany_rank, fftx_iodimx *howmany_dims,
  fftx_complex *in, fftx_complex *out,
  fftw_complex *data, fftx_callback func,
  unsigned flags);
```

The semantics of the pointwise operation are given as follows. Let

$$\begin{aligned}&\text{func} = p : \mathbb{C} \times \mathbb{C} \rightarrow \mathbb{C} \\ &\sigma = \langle \vec{s}_{\text{in}}, \vec{s}_{\text{out}}, \vec{s}_{\text{data}}, \vec{s}_{h,\text{in}}, \vec{s}_{h,\text{out}}, \vec{s}_{h,\text{data}} \rangle.\end{aligned}$$

The operator returned by the planner is defined as

$$\begin{aligned}\text{P}_\sigma^{r,h,\vec{n},\vec{m}} = \sum_{\vec{k} \in \bigtimes_{i=1}^{h-1} \mathbb{I}_{m_i}} &\left(\sum_{\vec{j} \in \bigtimes_{i=1}^{r-1} \mathbb{I}_{n_i}} \left(\text{e}_{(\vec{k} \oplus (\vec{j} + b_{\text{out}})) \cdot (\vec{s}_{h,\text{out}} \oplus \vec{s}_{\text{out}})}^{1 \times M_{\text{out}}} \right) \right. \\ &\circ p \circ \left(\text{e}_{(\vec{k} \oplus (\vec{j} + b_{\text{in}})) \cdot (\vec{s}_{h,\text{in}} \oplus \vec{s}_{\text{in}})}^{1 \times M_{\text{in}}} \right. \\ &\left. \left. \times \text{e}_{(\vec{k} \oplus (\vec{j} + b_{\text{data}})) \cdot (\vec{s}_{h,\text{data}} \oplus \vec{s}_{\text{data}})}^{1 \times M_{\text{data}}} \right) \right)\end{aligned}$$

Invoking this operator will perform the operation

$$\vec{y} \doteq \text{P}_\sigma^{r,h,\vec{n},\vec{m}}\, \vec{x}.$$

ACKNOWLEDGMENT

This research was supported at the Lawrence Berkeley National Laboratory by the Office of Advanced Scientfc Computng Research of the U.S. Department of Energy under Contract No. DE-AC02-05CH11231.

REFERENCES

[1] J. W. Cooley and J. W. Tukey, "An algorithm for the machine calculation of complex Fourier series," *Math. of Computation*, vol. 19, pp. 297–301, 1965.

[2] F. Franchetti and M. Püschel, *Encyclopedia of Parallel Computing*. Springer, 2011, ch. Fast Fourier Transform.

[3] H. Moulinec and P. Suquet, "A FFT-based numerical method for computing the mechanical properties of composites from images of their microstructures," in *IUTAM Symposium on Microstructure-Property Interactions in Composite Materials*, R. Pyrz, Ed. Springer Netherlands, 1995, pp. 235–246.

[4] D. F. Martin and P. Colella, "A cell-centered adaptive projection method for the incompressible Euler equations," *Journal of Computational Physics*, vol. 163, no. 2, pp. 271–312, 2000.

[5] A. Canning and D. Raczkowski, "Scaling first-principles plane-wave codes to thousands of processors," *Computer Physics Communications*, vol. 169, no. 1, pp. 449–453, 2005.

[6] A. Canning, "Scalable parallel 3d ffts for electronic structure codes," in *High Performance Computing for Computational Science - VECPAR 2008*, J. M. L. M. Palma, P. R. Amestoy, M. Daydé, M. Mattoso, and J. C. Lopes, Eds., 2008, pp. 280–286.

[7] J.-L. Vay, I. Haber, and B. B. Godfrey, "A domain decomposition method for pseudo-spectral electromagnetic simulations of plasmas," *Journal of Computational Physics*, vol. 243, pp. 260–268, 2013.

[8] M. Frigo and S. G. Johnson, "FFTW: An adaptive software architecture for the FFT," in *Proc. IEEE Int'l Conf. Acoustics, Speech, and Signal Processing (ICASSP)*, vol. 3, 1998, pp. 1381–1384, www.fftw.org.

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

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

[11] Intel, "Math kernel library," software.intel.com/mkl.

[12] IBM, "Engineering and scientific subroutine library," www.ibm.com/support/knowledgecenter/en/SSFHY8/essl_welcome.html.

[13] NVIDIA, "cuFFT," developer.nvidia.com/cufft.

[14] F. Franchetti, T.-M. Low, T. Popovici, R. Veras, D. G. Spampinato, J. Johnson, M. Püschel, J. C. Hoe, and J. M. F. Moura, "SPIRAL: Extreme performance portability," *Proceedings of the IEEE, special issue on "From High Level Specification to High Performance Code"*, vol. 106, no. 11, 2018.

[15] M. Püschel, F. Franchetti, and Y. Voronenko, *Encyclopedia of Parallel Computing*. Springer, 2011, ch. Spiral.

[16] 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, and N. Rizzolo, "SPIRAL: Code generation for DSP transforms," *Proceedings of the IEEE, special issue on "Program Generation, Optimization, and Adaptation"*, vol. 93, no. 2, pp. 232– 275, 2005.

[17] T.-M. Low and F. Franchetti, "High assurance code generation for cyber-physical systems," in *IEEE International Symposium on High Assurance Systems Engineering (HASE)*, 2017.

[18] F. Franchetti, F. de Mesmay, D. McFarlin, and M. Püschel, "Operator language: A program generation framework for fast kernels," in *IFIP Working Conference on Domain Specific Languages (DSL WC)*, 2009.

[19] A. J. Chorin, "A numerical method for solving incompressible viscous flow problems," *Journal of Computational Physics*, vol. 135, no. 2, pp. 118 – 125, 1997. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0021999197957168

[20] R. Hockney and J. Eastwood, *Computer Simulation Using Particles*, ser. Advanced book program: Addison-Wesley. McGraw-Hill, 1981.

[21] P. McCorquodale, P. Colella, G. T. Balls, and S. B. Baden, "A local corrections algorithm for solving Poisson's equation in three dimensions," vol. 2, 10 2006.

[22] C. R. Anderson, "A method of local corrections for computing the velocity field due to a distribution of vortex blobs," *Journal of Computational Physics*, vol. 62, no. 1, pp. 111–123, 1986.

[23] A. Brandt, "Multi-level adaptive solutions to boundary-value problems," *Mathematics of Computation*, vol. 31, no. 138, pp. 333–390, 1977.

[24] P. Basu, A. Venkat, M. Hall, S. Williams, B. Van Straalen, and L. Oliker, "Compiler generation and autotuning of communication-avoiding operators for geometric multigrid," in *20th Annual International Conference on High Performance Computing, HiPC 2013*. IEEE Computer Society, 2013, pp. 452–461.

[25] P. Colella, D. T. Graves, J. N. Johnson, H. S. Johansen, N. D. Keen, T. J. Ligocki, D. F. Martin, P. W. Mccorquodale, D. Modiano, P. O. Schwartz, T. D. Sternberg, and B. V. Straalen, "Chombo software package for AMR applications design document," Tech. Rep., 2003.

[26] H. Vincenti and J. Vay, "Ultrahigh-order Maxwell solver with extreme scalability for electromagnetic PIC simulations of plasmas," *Computer Physics Communications*, vol. 228, pp. 22–29, 2018.

[27] J.-L. Vay, A. Almgren, J. Bell, L. Ge, D. Grote, M. Hogan, O. Kononenko, R. Lehe, A. Myers, C. Ng, J. Park, R. Ryne, O. Shapoval, M. Thevenet, and W. Zhang, "Warp-X: A new exascale computing platform for beam-plasma simulations," *Nuclear Instruments and Methods in Physics Research Section A: Accelerators, Spectrometers, Detectors and Associated Equipment*, 2018.