FFT and Solvers for Exascale: FFTX and SpectralPACK
a first look

F. Franchetti, D. G. Spampinato, A. Kulkarni, T. M. Low (Carnegie Mellon University), M. Franusich (SpiralGen, Inc.), D. T. Popovici, A. Canning, P. McCorquodale, B. Van Straalen, P. Colella (Lawrence Berkeley National Laboratory)

Our approach

Have you ever wondered about this?

No analogue to LAPACK for spectral methods

- **Medium-size 1D FFT (1k—10k data points) is most common library call**
  applications break down 3D problems themselves and then call the 1D FFT library
- **Higher-level FFT calls rarely used**
  FFTW *guru* interface is powerful but hard to use, leading to performance loss
- **Low arithmetic intensity and variation of FFT use make library approach hard**
  Algorithm-specific decompositions and FFT calls intertwined with non-FFT code

FFTW is de-facto standard interface for FFT

- **FFTW 3.X is the high-performance reference implementation:**
  supports multicore/SMP and MPI, and Cell processor
- **Vendor libraries support the FFTW 3.X interface:**
  Intel MKL, IBM ESSL, AMD ACML (end-of-life), Nvidia cuFFT, Cray LibSci/CRAFFT

Issue 1: 1D FFTW call is standard kernel for many applications

- **Parallel libraries and applications reduce to 1D FFTW call**
  P3DFFT, QBox, PS/DNS, CPMD, HACC,…
- **Supported by modern languages and environments**
  Python, Matlab,…

Issue 2: FFTW is slowly becoming obsolete

- **FFTW 2.1.5 (still in use, 1997), FFTW 3 (2004) minor updates since then** *Risk: loss of high-performance FFT standard library*
- **Development currently dormant, except for small bug fixes**
- **No native support for accelerators (GPUs, Xeon PHI, FPGAs) and SIMT**
- **Parallel/MPI version does not scale beyond 32 nodes**

FFTX: FFTW revamped for Exascale

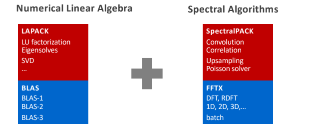
Modernized FFTW-style interface

- **Backwards compatible to FFTW 2.X and 3.X**
  old code runs unmodified and gains substantially but not fully
- **Small number of new features**
  futures/delayed execution, offloading, data placement, callback kernels

Code generation backend using SPIRAL

- **Library/application kernels are interpreted as specifications in DSL**
  extract semantics from source code and known library semantics
- **Compilation and advanced performance optimization**
  cross-call and cross library optimization, accelerator off-loading,…
- **Fine control over resource expenditure of optimization**
  compile-time, initialization-time, invocation time, optimization resources
- **Reference library implementation and bindings to vendor libraries**
  library-only reference implementation for ease of development

FFTX and SpectralPACK: long-term vision



Define the analogue to LAPACK for spectral algorithms

- **Define FFTX as the analogue to BLAS**
  provide user FFT functionality as well as algorithm building blocks
- **Define class of numerical algorithms to be supported by SpectralPACK**
  PDE solver classes (Green's function, sparse in normal/k space,…), signal processing,…
- **Define SpectralPACK functions**
  circular convolutions, NUFFT, Poisson solvers, free space convolution,…

Front end

Poisson's equation in free space

Partial differential equation (PDE)

$$\Delta(\Phi) = \rho$$
$$\rho : \mathbb{R}^3 \to \mathbb{R}$$
$$D = \mathrm{supp}(\rho) \subset \mathbb{R}^3$$

Poisson's equation. Δ is the Laplace operator

Solution characterization

$$\Phi : \mathbb{R}^3 \to \mathbb{R}$$
$$\Phi(\vec{x}) = \frac{Q}{4\pi||\vec{x}||} + o\left(\frac{1}{||\vec{x}||}\right) \text{ as } ||\vec{x}|| \to \infty$$
$$Q = \int_D \rho d\vec{x}$$

Approach: Green's function

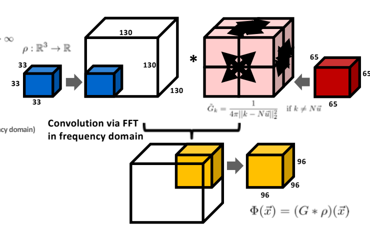
$$\Phi(\vec{x}) = \int_D G(\vec{x} - \vec{y})\rho(\vec{y})d\vec{y} \equiv (G * \rho)(\vec{x}), \quad G(\vec{x}) = \frac{1}{4\pi||\vec{x}||_2}$$

Solution: Φ(.) = convolution of RHS ρ(.) with Green's function G(.). Efficient through FFTs (frequency domain)

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

Green's function kernel in frequency domain

Hockney free-space convolution



Hockney: Convolution + problem-specific zero padding and output subset

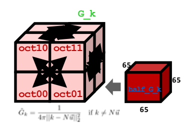
FFTX source code for Hockney

```
fftx_plan pruned_real_convolution_plan(fftx_real *in, fftx_real *out,
        fftx_complex *symbol, int n, int n_in, int n_out, int n_freq) {
    int rank = 3,
    batch_rank = 0,
    ...
    fftx_plan plans[5];
    fftx_plan p;

    tmp1 = fftx_create_zero_temp_real(rank, &padded_dims);

    plans[0] = fftx_plan_guru_copy_real(rank, &in_dimx, in, tmp1, MY_FFTX_MODE_SUB);

    tmp2 = fftx_create_temp_complex(rank, &freq_dims);
    plans[1] = fftx_plan_guru_dft_r2c(rank, &padded_dims, batch_rank,
        &batch_dims, tmp1, tmp2, MY_FFTX_MODE_SUB);

    tmp3 = fftx_create_temp_complex(rank, &freq_dims);
    plans[2] = fftx_plan_guru_pointwise_c2c(rank, &freq_dimx, batch_rank,
          &batch_dimx,   tmp2, tmp3, symbol, (fftx_callback)complex_scaling,
        MY_FFTX_MODE_SUB | FFTX_PW_POINTWISE);

    tmp4 = fftx_create_temp_real(rank, &padded_dims);
    plans[3] = fftx_plan_guru_dft_c2r(rank, &padded_dims, batch_rank,
        &batch_dims, tmp3, tmp4, MY_FFTX_MODE_SUB);

    plans[4] = fftx_plan_guru_copy_real(rank, &out_dimx, tmp4,
                                        out, MY_FFTX_MODE_SUB);

    p = fftx_plan_compose(numsubplans, plans, MY_FFTX_MODE_TOP);

    return p;
}
```

Looks like FFTW calls, but is a specification for SPIRAL.

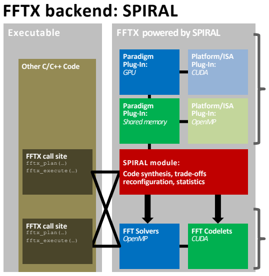
Describing the Green's function symmetry

```
// 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);
...
```

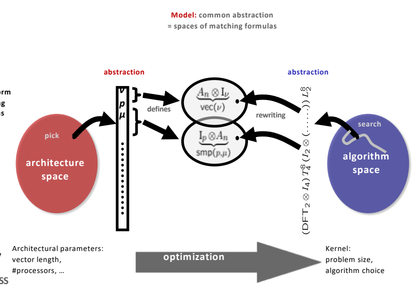


Back end

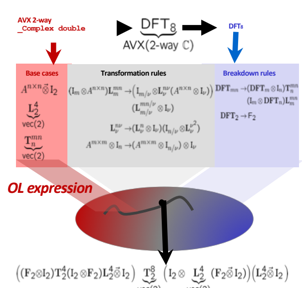
FFTX backend: SPIRAL



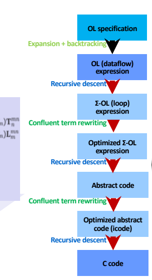
Platform-aware formal program synthesis



Autotuning in constraint solution space



Translating an OL expression into code



Constraint Solver Input:

DFT_8 AVX(2-way $\mathbb{C}$)

Output = Ruletree, expanded into

OL expression:

Σ-OL:

C code:

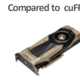
```
void dft8(_Complex double *Y, _Complex double *X) {
    __m256d s38, s39, s40, s41,...
    __m256d *a17, *a18;
    a17 = ((__m256d *) X);
    s38 = *(a17);
    s39 = *((a17 + 2));
    t38 = _mm256_add_pd(s38, s39);
    t39 = _mm256_sub_pd(s38, s39);
    ...
    s52 = _mm256_sub_pd(s45, s50);
    *((a18 + 3)) = s52;
}
```

Technology + Results

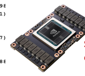
Generated code For Hockney convolution

```
void ioprunedconv_130_0_62_72_130(double  *Y, double  *X, double * S) {
    static double D84[260] = {65.5, 0.0, (-0.50000000000001132), (-
20.686114762237267),
      (-0.50000000000000081), (-10.337014680426078), (-0.50000000000000455),
    ...
for(int i18899 = 0; i18899 <= 1; i18899++) {
    for(int i18912 = 0; i18912 <= 4; i18912++) {
        a9807 = ((2*i18899) + (4*i18912));
        a9808 = (a9807 + 1);
        a9809 = (a9807 + 52);
        a9810 = (a9807 + 53);
        a9811 = (a9807 + 104);
        a9812 = (a9807 + 105);
        s3295 = (*((X + a9807)) + *((X + a9809))
            + *((X + a9811)));
        s3296 = (*((X + a9808)) + *((X + a9810))
            + *((X + a9812)));
        s3297 = (((0.30901699437494745*((X + a98
            - (0.80901699437494745*((X + a9811))
            + *((X + a9807)));
        ...
        *((104 + Y + a12569)) = ((s3983 - s3987)
            + (0.80901699437494745*t6537)
            + (0.58778525229247314*t6538));
        *((105 + Y + a12569)) = (((s3984 - s3988
            + (0.80901699437494745*t6538))
            - (0.58778525229247314*t6537));
    }
}
```

FFTX/SPIRAL with OpenACC backend
Compared to cuFFT expert interface



15% faster on TITAN V

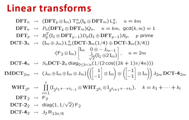


Same speed on Tesla V100

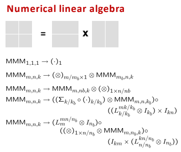
1,000s of lines of code, cross-call optimization, etc., transparently used

Algorithms: rules in domain-specific language

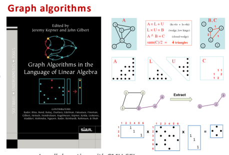
Linear transforms



Numerical linear algebra

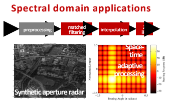


Graph algorithms



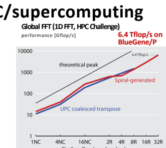
In collaboration with CMU-SEI

Spectral domain applications



SPIRAL: success in HPC/supercomputing

- **NCSA Blue Waters**
  PAID Program, FFTs for Blue Waters
- **RIKEN K computer**
  FFTs for the HPC-ACE ISA
- **LANL RoadRunner**
  FFTs for the Cell processor
- **PSC/XSEDE Bridges**
  Large size FFTs
- **LLNL BlueGene/L and P**
  FFTW for BlueGene/L's Double FPU
- **ANL BlueGene/Q Mira**
  Early Science Program, FFTW for BGQ QPX



BlueGene/P at Argonne National Laboratory
128k cores (quad-core CPUs) at 850 MHz

2006 Gordon Bell Prize (Peak Performance Award) with LLNL and IBM
2010 HPC Challenge Class II Award (Most Productive System) with ANL and IBM

SPIRAL 8.0: available under open source

- **Open-source SPIRAL available**
  - non-viral license (BSD)
  - Initial version, effort ongoing to open source whole system
  - Commercial support via SpiralGen, Inc.
- **Developed over 20 years**
  - Funding: DARPA (OPAL, DESA, HACMS, PERFECT, BRASS), NSF, ONR, DoD HPC, JPL, DOE, CMU SEI, Intel, Nvidia, Mercury
- **Open sourced under DARPA PERFECT**

F. Franchetti, T. M. Low, D. T. Popovici, R. M. Veras, D. G. Spampinato, J. R. Johnson, M. Püschel, J. C. Hoe, J. M. F. Moura: SPIRAL: Extreme Performance Portability, Proceedings of the IEEE, Vol. 106, No. 11, 2018. Special Issue on From High Level Specification to High Performance Code

www.spiral.net

Carnegie Mellon — SpiralGen, Accelerating Innovation in Computing — Berkeley Lab, Lawrence Berkeley National Laboratory — ECP Exascale Computing Project