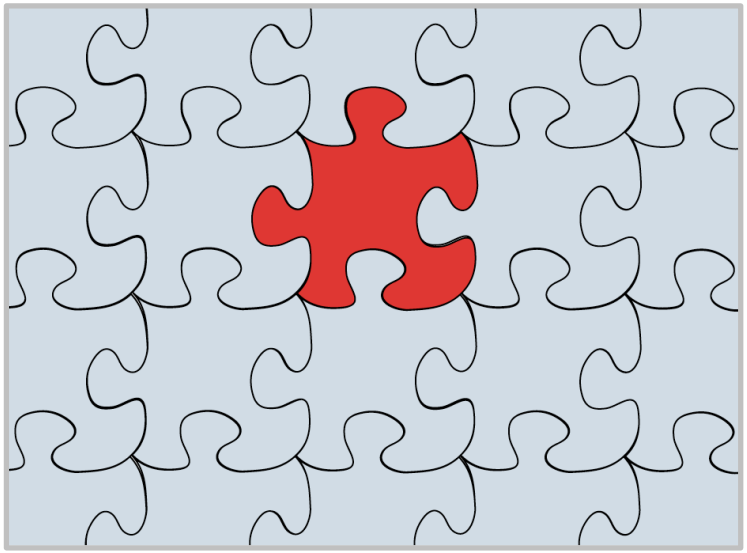
General-Size Library Generation With Spiral

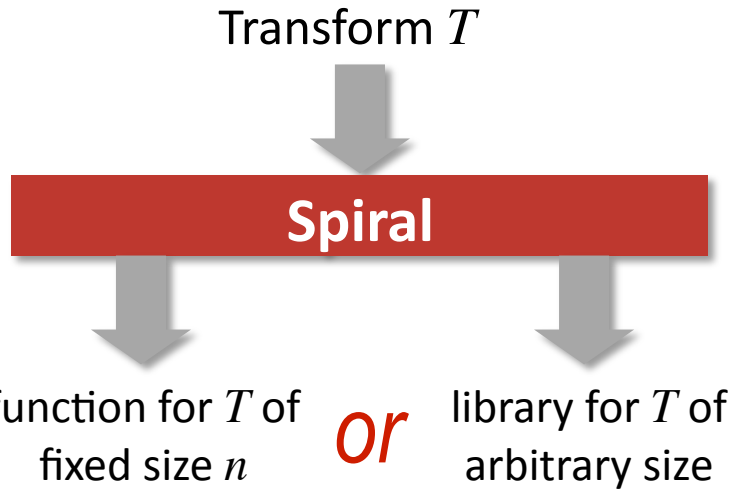
Yevgen Voronenko

Carnegie Mellon University

Overview

Spiral: Automating Library Development for Linear Transforms





Automated:

- Loop optimizations
- Vectorization
- Parallelization
- Derivation of general size libraries
- Algorithm restructuring and exploration
- Implementation

Fundamentally different problems

Generating Fixed Size Transform Code

Transform
user specified

$$\mathrm{DFT}_8$$

Fast algorithm in SPL
many choices

$$(\mathrm{DFT}_2 \otimes I_4)\, T_4^8\, (I_2 \otimes ((\mathrm{DFT}_2 \otimes I_2) \cdot T_2^4\, (I_2 \otimes \mathrm{DFT}_2)\, L_2^4))\, L_2^8$$

∑-SPL:

$$\sum \left(S_j\, \mathrm{DFT}_2\, G_j\right) \sum \left(\sum \left(S_{k,l}\, \mathrm{diag}\left(\mathrm{t}_{k,l}\right) \mathrm{DFT}_2\, G_l\right) \sum \left(S_m\, \mathrm{diag}\left(\mathrm{t}_m\right) \mathrm{DFT}_2\, G_{k,m}\right)\right)$$

C Code:

```
void sub(double *y, double *x) {
  double f0, f1, f2, f3, f4, f7, f8, f10, f11;
  f0 = x[0] - x[3];
  f1 = x[0] + x[3];
  f2 = x[1] - x[2];
  f3 = x[1] + x[2];
  f4 = f1 - f3;
  y[0] = f1 + f3;
  y[2] = 0.7071067811865476 * f4;
  f7 = 0.9238795325112867 * f0;
  f8 = 0.3826834323650898 * f2;
  y[1] = f7 + f8;
  f10 = 0.3826834323650898 * f0;
  f11 = (-0.9238795325112867) * f2;
  y[3] = f10 + f11;
}
```

Optimization at all abstraction levels

- parallelization
  vectorization
- loop
  optimizations
- constant folding
  scheduling
  ……

Generating General Size Transform Libraries

Input:

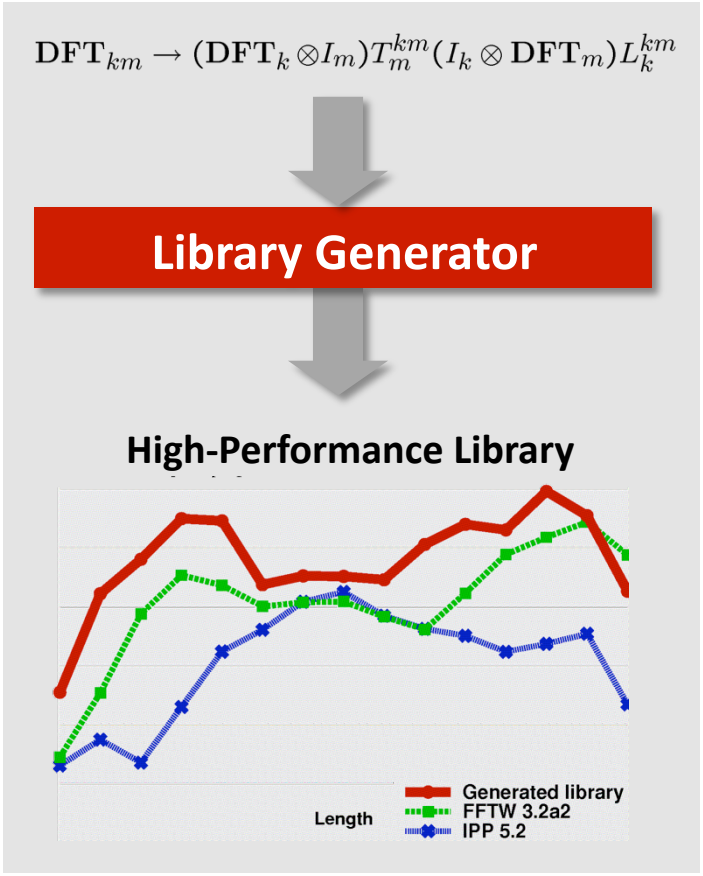
- **Transform**: DFT_n
- **Algorithms**:

$$\mathrm{DFT}_{km} \rightarrow (\mathrm{DFT}_k \otimes I_m) T^{km}_m (I_k \otimes \mathrm{DFT}_m) L^{km}_k$$
$$\mathrm{DFT}_2 \rightarrow \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix}$$

- **Vectorization**: 2-way SSE
- **Threading**: Yes

Output:

- Optimized library (10,000 lines of C++)
- For general input size
- Vectorized and multithreaded
- With runtime adaptation mechanism
- Performance competitive with hand -written code



Linear Transforms

- **Mathematically:** matrix-vector product $y = Tx$

Output vector — Transform matrix — Input vector

- **Examples:**

$$\mathbf{DFT}_n = \left[\omega_n^{k\ell}\right]_{0\le k,\ell<n}, \quad \omega_n = e^{-2\pi j/n}$$

$$\mathbf{WHT}_n = \begin{bmatrix} \mathbf{WHT}_{n/2} & \mathbf{WHT}_{n/2} \\ \mathbf{WHT}_{n/2} & -\mathbf{WHT}_{n/2} \end{bmatrix}$$

$$\mathbf{RDFT}_n = \begin{bmatrix} I''_n \\ \begin{bmatrix} \cos\frac{2\pi k\ell}{n} \\ -\sin\frac{2\pi k\ell}{n} \end{bmatrix} \end{bmatrix}$$

$$\mathbf{DHT}_n = \begin{bmatrix} I''_n \\ \begin{bmatrix} \mathrm{cas}\frac{2\pi k\ell}{n} \\ \mathrm{cms}\frac{2\pi k\ell}{n} \end{bmatrix} \end{bmatrix}$$

$$\mathbf{DCT\text{-}1}_n = \left[\cos\frac{k\ell\pi}{n-1}\right]$$

$$\mathbf{DST\text{-}1}_n = \left[\sin\frac{(k+1)(\ell+1)\pi}{n+1}\right]$$

$$\mathbf{DCT\text{-}2}_n = \left[\cos\frac{k(2\ell+1)\pi}{2n}\right]$$

$$\mathbf{DST\text{-}2}_n = \left[\sin\frac{(k+1)(2\ell+1)\pi}{2n}\right]$$

$$\mathbf{DCT\text{-}3}_n = \left[\cos\frac{(2k+1)\ell\pi}{2n}\right]$$

$$\mathbf{DST\text{-}3}_n = \left[\sin\frac{(2k+1)(\ell+1)\pi}{2n}\right]$$

$$\mathbf{DCT\text{-}4}_n = \left[\cos\frac{(2k+1)(2\ell+1)\pi}{4n}\right]$$

$$\mathbf{DST\text{-}4}_n = \left[\sin\frac{(2k+1)(2\ell+1)\pi}{4n}\right]$$

$$\mathbf{MDCT}_n = \left[\cos\frac{(2k+1)(2\ell+1+n)\pi}{4n}\right]$$

$$\mathbf{IMDCT}_n = \left[\cos\frac{(2\ell+1)(2k+1+n)\pi}{4n}\right]$$

$$\mathbf{Filt}_n(t) = \begin{bmatrix} t_0 & \dots & t_{k-1} & & \\ & \ddots & \dots & \ddots & \\ & & t_0 & \dots & t_{k-1} \end{bmatrix}$$

$$\downarrow 2\,\mathbf{Filt}_n(t) = \begin{bmatrix} t_0 & t_1 & \dots & t_{k-1} & & & \\ & & t_0 & t_1 & \dots & t_{k-1} & \\ & & & \ddots & \dots & \dots & \ddots \\ & & & & t_0 & \dots & t_{k-1} \end{bmatrix}$$

Transform Algorithms: Example 4-point FFT

Cooley/Tukey fast Fourier transform (FFT):

$$\begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & j & -1 & -j \\ 1 & -1 & 1 & -1 \\ 1 & -j & -1 & j \end{bmatrix} = \begin{bmatrix} 1 & \cdot & 1 & \cdot \\ \cdot & 1 & \cdot & 1 \\ 1 & \cdot & -1 & \cdot \\ \cdot & 1 & \cdot & -1 \end{bmatrix} \begin{bmatrix} 1 & \cdot & \cdot & \cdot \\ \cdot & 1 & \cdot & \cdot \\ \cdot & \cdot & 1 & \cdot \\ \cdot & \cdot & \cdot & j \end{bmatrix} \begin{bmatrix} 1 & 1 & \cdot & \cdot \\ 1 & -1 & \cdot & \cdot \\ \cdot & \cdot & 1 & 1 \\ \cdot & \cdot & 1 & -1 \end{bmatrix} \begin{bmatrix} 1 & \cdot & \cdot & \cdot \\ \cdot & \cdot & 1 & \cdot \\ \cdot & 1 & \cdot & \cdot \\ \cdot & \cdot & \cdot & 1 \end{bmatrix}$$

Fourier transform — Diagonal matrix (twiddles)

$$\mathbf{DFT}_4 \rightarrow (\mathbf{DFT}_2 \otimes \mathrm{I}_2)\, \mathsf{T}^4_2 (\mathrm{I}_2 \otimes \mathbf{DFT}_2)\, \mathsf{L}^4_2$$

Kronecker product — Identity — Permutation

- Algorithms are divide-and-conquer: Breakdown rules
- Mathematical, declarative representation: SPL (signal processing language)
- SPL describes the structure of the dataflow

Fast Algorithms as "Breakdown Rules"

$$\mathbf{DFT}_n \longrightarrow (\mathbf{DFT}_k \otimes I_m) D_{k,m} (I_k \otimes \mathbf{DFT}_m) L^n_k \tag{2.1}$$

$$\mathbf{DFT}_n \longrightarrow V^{-1}_{m,k} (\mathbf{DFT}_k \otimes I_m)(I_k \otimes \mathbf{DFT}_m) V_{m,k} \tag{2.2}$$

$$\mathbf{DFT}_n \longrightarrow W_n^{-1} (I_1 \oplus \mathbf{DFT}_{n-1}) E_n (I_1 \oplus \mathbf{DFT}_{n-1}) W_n \tag{2.3}$$

$$\mathbf{DFT}_n \longrightarrow B^\top_{n,m} D_m \, \mathbf{DFT}_m \, D'_m \, \mathbf{DFT}_m \, D''_m B_{n,m}, \quad m \geq 2n-1 \tag{2.4}$$

$$\mathbf{DFT}_n \longrightarrow P^\top_{k/2,2m} \left(\mathbf{DFT}_{2m} \oplus \left(I_{k/2-1} \otimes_i C_{2m} \, \mathbf{rDFT}_{2m}((i+1)/k) \right) \right) \left(\mathbf{RDFT}'_k \otimes I_m \right) \tag{2.5}$$

$$\begin{vmatrix} \mathbf{RDFT}_n \\ \mathbf{RDFT}'_n \\ \mathbf{DHT}_n \\ \mathbf{DHT}'_n \end{vmatrix} \longrightarrow (P^\top_{k/2,m} \otimes I_2) \left(\begin{vmatrix} \mathbf{RDFT}_{2m} \\ \mathbf{RDFT}'_{2m} \\ \mathbf{DHT}_{2m} \\ \mathbf{DHT}'_{2m} \end{vmatrix} \oplus \left(I_{k/2-1} \otimes_i M_{2m} \begin{vmatrix} \mathbf{rDFT}_{2m}((i+1)/k) \\ \mathbf{rDFT}_{2m}((i+1)/k) \\ \mathbf{rDHT}_{2m}((i+1)/k) \\ \mathbf{rDHT}_{2m}((i+1)/k) \end{vmatrix} \right) \right)$$
$$\cdot \left(\begin{vmatrix} \mathbf{RDFT}'_k \\ \mathbf{RDFT}'_k \\ \mathbf{DHT}'_k \\ \mathbf{DHT}'_k \end{vmatrix} \otimes I_m \right) \tag{2.6}$$

$$\mathbf{RDFT}_n \longrightarrow D_n \cdot \mathbf{DCT\text{-}2}_n \cdot P_n, \quad n \text{ odd} \tag{2.7}$$

$$\mathbf{DCT\text{-}2}_n \longrightarrow P^\top_{k/2,2m} \left(\mathbf{DCT\text{-}2}_{2m} \, K^{2m}_2 \oplus \left(I_{k/2-1} \otimes N_{2m} \, \mathbf{RDFT\text{-}3}^\top_{2m} \right) \right) G_n (L^{n/2}_{k/2} \otimes I_2)$$
$$\cdot (I_m \otimes \mathbf{RDFT}'_k) Q_{m/2,k} \tag{2.8}$$

DFT Cooley-Tukey

DCT "Cooley-Tukey"

Capture complicated algorithms concisely

	Fixed Size	General Size
Adaptation	generation-time	runtime (offline)
Code size	100 – 10,000 LOC	1,000 – 10,000,000 LOC
Generated from	ruletree	breakdown rules
Contains	▪ 1 function ▪ loops ▪ arithmetic	▪ mutually recursive functions ▪ loops ▪ arithmetic ▪ initialization ▪ runtime adaptation mechanism
User code	`dft_1024(Y, X);`	`Env_1 dft(1024);` `dft.compute(Y, X);`

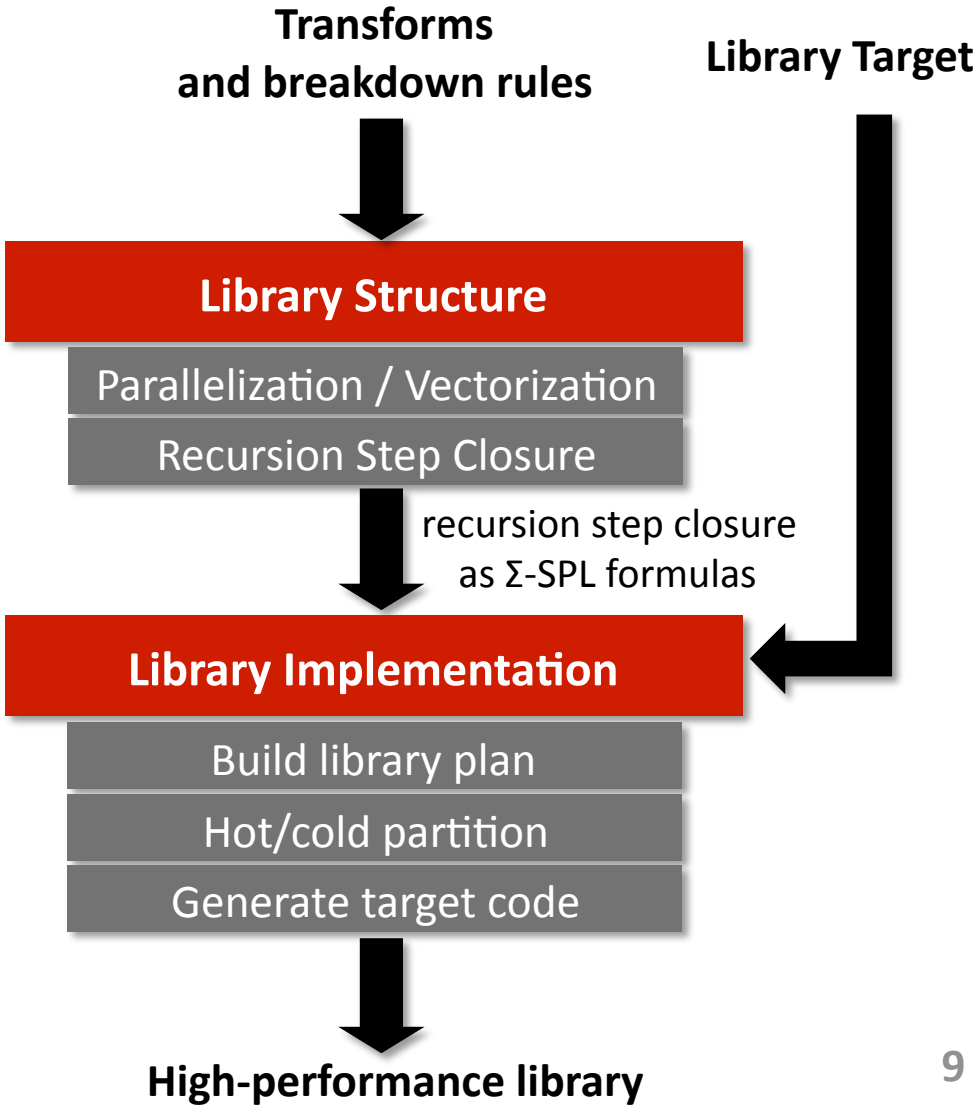
How Library Generation Works

Library Structure :
Target Independent

No generation-time search
Fully deterministic

Library Implementation :
Target dependent

Generates initialization code
Plugs into the target infrastructure



I. Background

II. **Library Generation: Structure**

Recursion Step Closure

Base Cases

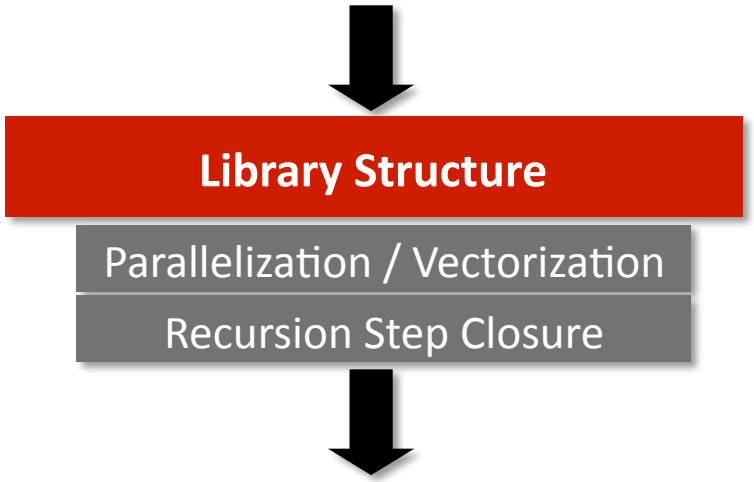
III. Library Generation: Implementation

IV. Experimental Results

V. Conclusions and Future Work

Library Structure

$$\mathsf{V}_{k,m}^{T}(\mathbf{DFT}_k \otimes \mathrm{I}_m)(\mathrm{I}_k \otimes \mathbf{DFT}_m)\,\mathsf{V}_{k,m}$$
$$(\mathbf{DFT}_k \otimes \mathrm{I}_m)\,\mathsf{T}_{m}^{km}(\mathrm{I}_k \otimes \mathbf{DFT}_m)\,\mathsf{L}_{k}^{km}$$



$$\mathbf{DFT}$$
$$\mathrm{S}_z\,\mathbf{DFT}\,\mathrm{G}_h$$
$$\mathrm{S}_h\,\mathbf{DFT}\,\mathrm{G}_z$$
$$\mathrm{S}_h\,\mathbf{DFT}\,\mathrm{diag}\left(\mathrm{Dat}\right)\mathrm{G}_h$$
$$\mathrm{S}_h\,\mathbf{DFT}\,\mathrm{G}_h$$
$$\mathrm{S}_h\,\mathbf{DFT}\,\mathrm{G}_{h\circ z}$$
$$\mathrm{S}_{h\circ z}\,\mathbf{DFT}\,\mathrm{G}_h$$
$$\mathrm{S}_h\,\mathbf{DFT}\,\mathrm{diag}\left(\mathrm{Dat}\right)\mathrm{G}_{h\circ z}$$
$$\mathrm{S}_z\,\mathbf{DFT}\,\mathrm{diag}\left(\mathrm{Dat}\right)\mathrm{G}_h$$
$$\mathrm{S}_{h\circ z}\,\mathbf{DFT}\,\mathrm{diag}\left(\mathrm{Dat}\right)\mathrm{G}_h$$

- **Input:**
  - Breakdown rules
- **Output:**
  - Recursion step closure
  - Σ-SPL Implementation of each recursion step
- **Parallelization/Vectorization**
  - Adds additional breakdown rules
  - Orthogonal to the closure generation
- **Closure**
  - Descend
  - Parametrize
  - Repeat until no new recursion steps found

Recursion Steps

- **Cooley-Tukey FFT**

$$y = (\mathrm{DFT}_k \otimes I_m) T_m^{km} (I_k \otimes \mathrm{DFT}_m) L_k^{km} x$$

- **Implementation in FFTW 2.x**

```
DFT(int n, complex *Y, complex *X) {
  k = choose_factor(n); m = n/k;

  for i=0 to k-1
    DFT_strided(m, k, 1, Y + m*i, T + m*i)

  for i=0 to m-1
    DFT_scaled(k, m, precomputed_f, Y + i, Y + i)
}

DFT_strided(int n, int is, int os, complex *Y, complex *X) {…}

DFT_scaled(int n, int s, complex *F, complex *Y, complex *X) {…}
```

2 additional functions (recursion steps) needed …
… and they may spawn more

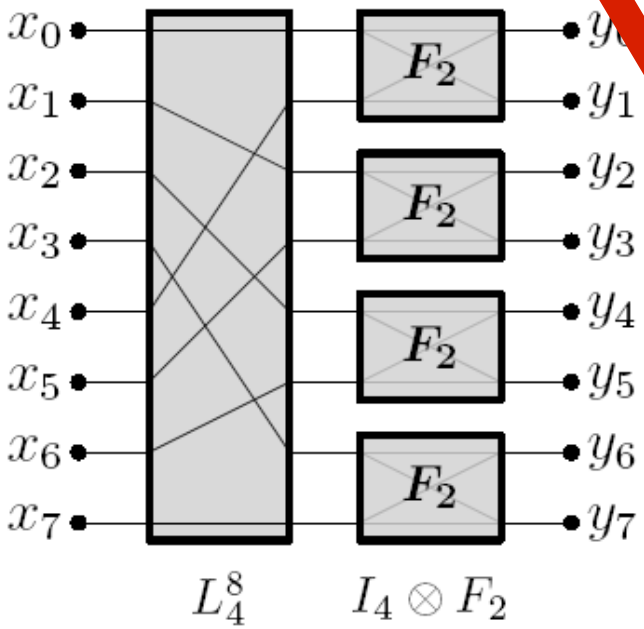
How to discover automatically?

Σ-SPL: Key to step combining

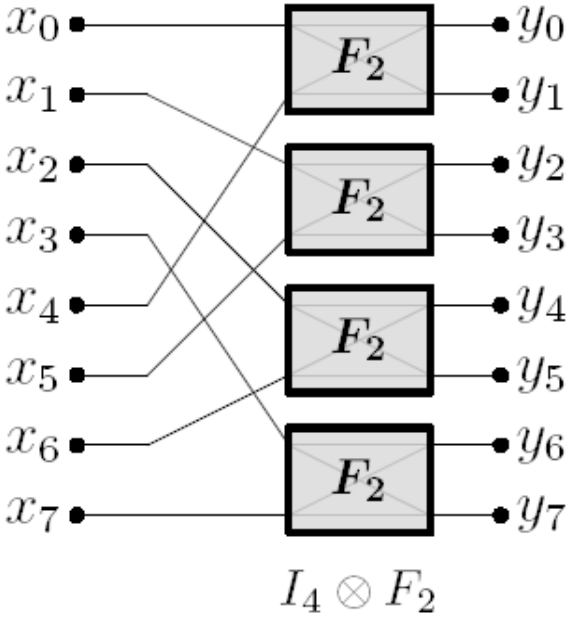
$$y = (I_4 \otimes F_2) L^8_4 x$$

our solution

$$y = \sum_{j=0}^{3} S_{h_{2j,1}} F_2 G_{h_{j,2}} x$$



original Spiral




```
for (int i=0; i<10; i++)
  t[i] = x[(i/5) + 2*(i % 5)];

for (int i=0; i<5; i++) {
  y[2*i] = t[2*i] + t[2*i+1];
  y[2*i+1] = t[2*i] - t[2*i+1];
}
```

impractical

```
for (int j=0;j<2;j++) {
  y[2*j] = x[2*j] + x[2*j+1];
  y[2*j+1] = x[2*j] - x[2*j+1];
}
```


Joint work with Franz Franchetti

Σ-SPL: Basic Idea

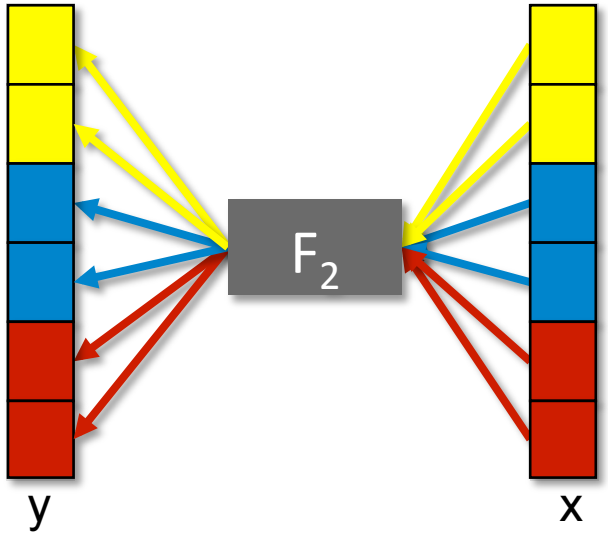
- **Four matrix constructs:**
  - **Σ** (sum) – explicit loop
  - $\mathbf{G}_f$ (gather) – load with index mapping f
  - $\mathbf{S}_f$ (scatter) – store with index mapping f
  - $\mathbf{Perm}_f$ – permute with the index mapping f
- **Predefined index-mapping functions**
  - **h** – stride
  - **z** – stride mod N (prime-factor algorithms)
  - **r** – reflecting stride (RDFT)
  - **½** – exponential

Example:

$$(I_3 \otimes F_2) \rightarrow \begin{bmatrix} F_2 & & \\ & F_2 & \\ & & F_2 \end{bmatrix} \rightarrow \begin{bmatrix} F_2 & & \\ & & \\ & & \end{bmatrix} + \begin{bmatrix} & & \\ & F_2 & \\ & & \end{bmatrix} + \begin{bmatrix} & & \\ & & \\ & & F_2 \end{bmatrix}$$

$$\rightarrow \mathsf{S}(f_0) F_2 \mathsf{G}(f_0) + \mathsf{S}(f_1) F_2 \mathsf{G}(f_1) + \mathsf{S}(f_2) F_2 \mathsf{G}(f_2)$$

$$\rightarrow \left(\sum_{j=0}^{2} \mathsf{S}_{f_j} F_2 \mathsf{G}_{f_j} \right)$$



$$\mathsf{G}_{f_j} = \begin{bmatrix} 1 & \\ & 1 \\ & \end{bmatrix} = \mathsf{S}_{f_j}^{\top}$$

Gather / Scatter

$$f_j = h_{2j,1}$$
$$i \mapsto 2j + i$$

Stride index-mapping

Recursion Step Closure

- **Input:** transform T and a breakdown rule
- **Output:** spawned recursion steps + Σ-SPL implementation
- **Algorithm:**

1. Apply the breakdown rule

$$\{\mathbf{DFT}_n\}$$

$$\downarrow$$

$$(\{\mathbf{DFT}_{n/k}\} \otimes I_k) T^n_k (I_{n/k} \otimes \{\mathbf{DFT}_k\}) L^n_{n/k}$$

2. Convert to Σ-SPL

$$\downarrow$$

$$\left(\sum_{i=0}^{k-1} \mathsf{S}_{h_{i,k}} \{\mathbf{DFT}_{n/k}\} \mathsf{G}_{h_{i,k}}\right) \mathrm{diag}(f) \left(\sum_{j=0}^{n/k-1} \mathsf{S}_{h_{jk,1}} \{\mathbf{DFT}_k\} \mathsf{G}_{h_{jk,1}}\right) \mathrm{perm}(\ell^n_{n/k})$$

3. Apply loop merging + index simplification rules.

$$\downarrow$$

$$\sum_{i=0}^{k-1} \mathsf{S}_{h_{i,k}} \{\mathbf{DFT}_{n/k}\} \mathrm{diag}(f \circ h_{i,k}) \mathsf{G}_{h_{i,k}} \sum_{j=0}^{n/k-1} \mathsf{S}_{h_{jk,1}} \{\mathbf{DFT}_k\} \mathsf{G}_{h_{j,n/k}}$$

4. Extract recursion steps

$$\downarrow$$

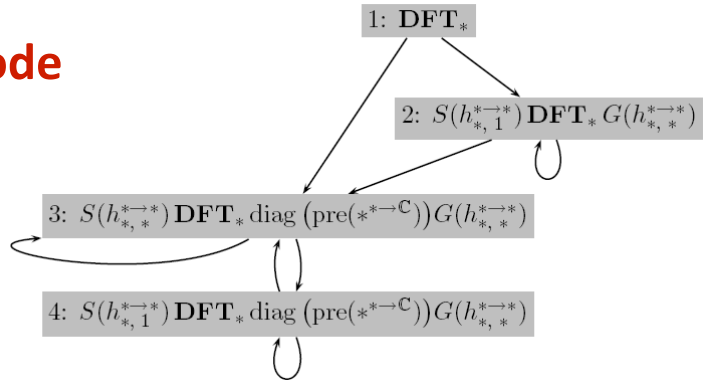
$$\sum_{i=0}^{k-1} \left\{ \mathsf{S}_{h_{i,k}} \mathbf{DFT}_{n/k} \mathrm{diag}(f \circ h_{i,k}) \mathsf{G}_{h_{i,k}} \right\} \sum_{j=0}^{n/k-1} \left\{ \mathsf{S}_{h_{jk,1}} \mathbf{DFT}_k \mathsf{G}_{h_{j,n/k}} \right\}$$

5. Repeat until closure is reached

Parametrization (not shown) derives the independent parameter set for each recursion step

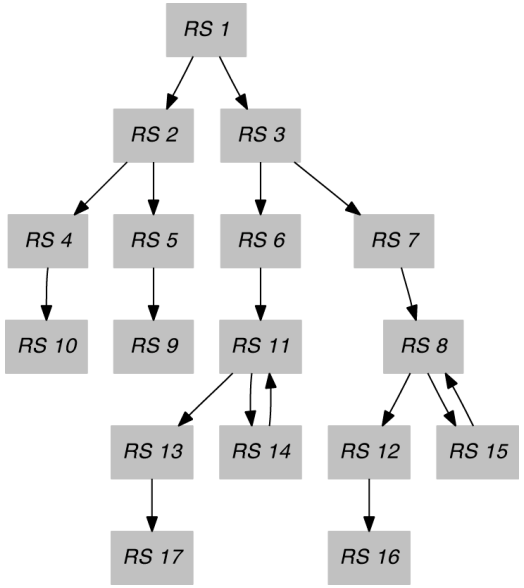
Repeat

DFT: scalar code



Library = 4 mutually recursive functions

DCT4: vector code



1 : $\mathrm{Vec}_2(\mathbf{DCT\text{-}4}_{u_1})$

2 : $\mathrm{Vec}_2(\mathrm{GT}(\mathrm{diag}\,(N_{2u_8})\,\mathbf{RDFT\text{-}3}^{\top}_{2u_8}\,\mathrm{rcdiag}(\mathrm{pre}(u_4{}^{\mathbb{Z}\times 2u_8\to\mathbb{R}})),\ h^{2u_8\to u_6}_{0,\,1,u_7}\circ\ell^{2u_8}_{u_8},\ r^{2u_8\to u_9}_{0,\,u_{11},1,\,u_{12}},\ \{u_{13}\}))$

3 : $\mathrm{Vec}_2(\mathrm{GT}(\mathbf{RDFT\text{-}3}_{u_1}\,\mathrm{diag}\,(N_{u_1}),\ r^{u_1\to u_3}_{0,\,u_5,1,\,u_6},\ h^{u_1\to u_8}_{0,\,u_9,1},\ \{u_{10}\}))$

4 : $\mathrm{VJam}_2(\mathrm{GT}(\mathrm{diag}\,(N_{2u_9})\,\mathbf{RDFT\text{-}3}^{\top}_{2u_9}\,\mathrm{rcdiag}(\mathrm{pre}(u_4{}^{\mathbb{Z}\times\mathbb{Z}\times 2u_9\to\mathbb{R}})),\ h^{2u_9\to u_6}_{0,\,1,u_7,u_8}\circ\ell^{2u_9}_{u_9},\ r^{2u_9\to u_{10}}_{0,\,u_{12},1,2,\,u_{13}},\ \{2,u_{14}\}))$

5 : $\mathrm{GT}(\mathrm{diag}\,(N_{2u_9})\,\mathbf{RDFT\text{-}3}^{\top}_{2u_9}\,\mathrm{rcdiag}(\mathrm{pre}(u_4{}^{\mathbb{Z}\times 2u_9\to\mathbb{R}})),\ h^{2u_9\to u_6}_{u_7,\,1,u_8}\circ\ell^{2u_9}_{u_9},\ r^{2u_9\to u_{10}}_{u_{12},\,u_{13},1,\,u_{14}},\ \{u_{15}\})$

6 : $\mathrm{VJam}_2(\mathrm{GT}(\mathbf{RDFT\text{-}3}_{u_1}\,\mathrm{diag}\,(N_{u_1}),\ r^{u_1\to u_3}_{0,\,u_5,1,2,\,u_6},\ h^{u_1\to u_8}_{0,\,u_9,1,2},\ \{2,u_{10}\}))$

7 : $\mathrm{GT}(\mathbf{RDFT\text{-}3}_{u_1}\,\mathrm{diag}\,(N_{u_1}),\ r^{u_1\to u_3}_{u_5,\,u_6,1,\,u_7},\ h^{u_1\to u_9}_{u_{10},\,u_{11},1},\ \{u_{12}\})$

8 : $\mathrm{S}(h^{u_1\to u_2}_{u_3,\,u_4})\,\mathbf{RDFT\text{-}3}_{u_1}\,\mathrm{diag}\,(N_{u_1})\mathrm{G}(r^{u_1\to u_7}_{u_9,\,u_{10},\,u_{11}})$

9 : $\mathrm{S}(r^{2u_{13}\to u_1}_{u_3,\,u_4,\,u_5})\mathrm{diag}\,(N_{2u_{13}})\,\mathbf{RDFT\text{-}3}^{\top}_{2u_{13}}\,\mathrm{rcdiag}(\mathrm{pre}(u_9{}^{2u_{13}\to\mathbb{R}}))\mathrm{G}(h^{2u_{13}\to u_{11}}_{u_{12},\,1}\circ\ell^{2u_{13}}_{u_{13}})$

10 : $\mathrm{VJam}_2(\mathrm{GT}(\mathrm{diag}\,(N_{2u_9})\,\mathbf{RDFT\text{-}3}^{\top}_{2u_9}\,\mathrm{rcdiag}(\mathrm{pre}(u_4{}^{\mathbb{Z}\times 2u_9\to\mathbb{R}})),\ h^{2u_9\to u_6}_{u_7,\,1,u_8}\circ\ell^{2u_9}_{u_9},\ r^{2u_9\to u_{10}}_{u_{12},\,u_{13},1,\,u_{14}},\ \{2\}))$

11 : $\mathrm{VJam}_2(\mathrm{GT}(\mathbf{RDFT\text{-}3}_{u_1}\,\mathrm{diag}\,(N_{u_1}),\ r^{u_1\to u_3}_{u_5,\,u_6,1,\,u_7},\ h^{u_1\to u_9}_{u_{10},\,u_{11},1},\ \{2\}))$

12 : $\mathrm{GT}(\mathrm{diag}\,(C_{u_1})\,\mathbf{rDFT}_{2u_1}(\lambda\text{-wrap}(\lambda_1{}^{\mathbb{Z}\times\mathbb{Z}\to\mathbb{R}})),\ h^{2u_1\to u_4}_{0,\,1,u_5},\ h^{2u_{10}\to u_7}_{u_8,\,u_9}\circ(r^{u_1\to u_{10}}_{0,\,u_{12},1,\,u_{13}}\otimes\imath_2),\ \{u_{14}\})$

13 : $\mathrm{VJam}_2(\mathrm{GT}(\mathrm{diag}\,(C_{u_1})\,\mathbf{rDFT}_{2u_1}(\lambda\text{-wrap}(\lambda_1{}^{\mathbb{Z}\times\mathbb{Z}\times\mathbb{Z}\to\mathbb{R}})),\ h^{2u_1\to u_4}_{u_5,\,u_6,1,u_7},\ h^{2u_{12}\to u_9}_{u_{10},\,u_{11},1}\circ(r^{u_1\to u_{12}}_{0,\,u_{14},0,1,\,u_{15}}\otimes\imath_2),\ \{2,u_{16}\}))$

14 : $\mathrm{VJam}_2(\mathrm{GT}(\mathbf{RDFT\text{-}3}_{u_1}\,\mathrm{diag}\,(N_{u_1}),\ r^{u_1\to u_3}_{u_5,\,u_6,1,u_7,\,u_8},\ h^{u_1\to u_{10}}_{u_{11},\,u_{12},1,u_{13}},\ \{2,u_{14}\}))$

15 : $\mathrm{GT}(\mathbf{RDFT\text{-}3}_{u_1}\,\mathrm{diag}\,(N_{u_1}),\ r^{u_1\to u_3}_{u_5,\,u_6,u_7,\,u_8},\ h^{u_1\to u_{10}}_{0,\,u_{11},1},\ \{u_{12}\})$

16 : $\mathrm{S}(h^{2u_5\to u_2}_{u_3,\,u_4}\circ(r^{u_6\to u_5}_{u_7,\,u_8,\,u_9}\otimes\imath_2))\mathrm{diag}\,(C_{u_6})\,\mathbf{rDFT}_{2u_6}(\lambda\text{-wrap}(\lambda_1{}^{\mathbb{Z}\to\mathbb{R}}))\mathrm{G}(h^{2u_6\to u_{13}}_{u_{14},\,1})$

17 : $\mathrm{VJam}_2(\mathrm{GT}(\mathrm{diag}\,(C_{u_1})\,\mathbf{rDFT}_{2u_1}(\lambda\text{-wrap}(\lambda_1{}^{\mathbb{Z}\times\mathbb{Z}\to\mathbb{R}})),\ h^{2u_1\to u_4}_{u_5,\,u_6,1},\ h^{2u_{11}\to u_8}_{u_9,\,u_{10},1}\circ(r^{u_1\to u_{11}}_{u_{13},\,u_{14},\,u_{15}}\otimes\imath_2),\ \{2\}))$

Base Cases

- **Base cases are called "codelets" in FFTW, UHFFT**
- **Why needed:**
  - Recursion step closure is translated into mutually recursive functions
  - Recursion must be terminated
  - The larger the base case – the less recursion overhead
- **How many:**
  - **In FFTW 3.2:** 183 codelets for complex DFT (21 types)

    147 codelets for real DFT (18 types)
  - **In the generator:** # codelet types · # recursion steps
- **Obtained by specializing recursion steps to fixed sizes and using standard Spiral to generate code**

$$\left\{ \mathrm{S}(h^{2\to u_2}_{u_3,1})\,\mathbf{DFT}_2\,\mathrm{G}(h^{2\to u_6}_{u_7,u_8}) \right\}$$
$$\left\{ \mathrm{S}(h^{3\to u_2}_{u_3,1})\,\mathbf{DFT}_3\,\mathrm{G}(h^{3\to u_6}_{u_7,u_8}) \right\}$$
$$\ldots$$

I. Background

II. Library Generation: Structure

Recursion Step Closure

Base Cases

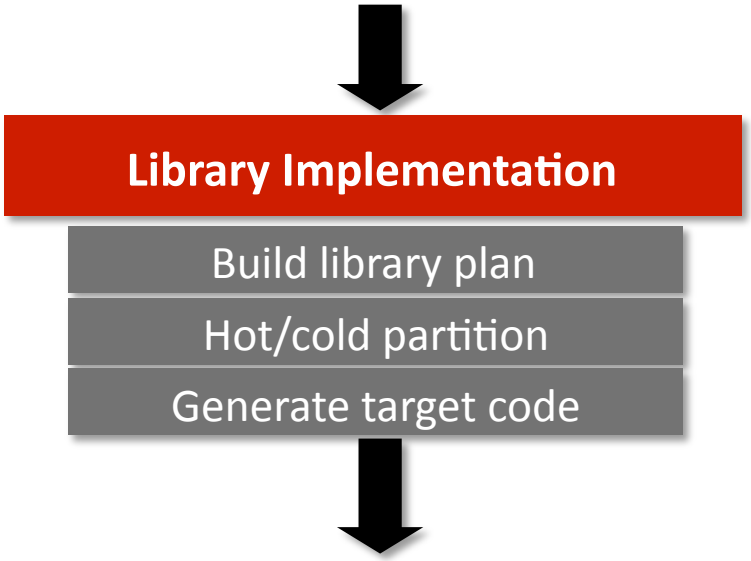
III. Library Generation: Implementation

IV. Experimental Results

V. Conclusions and Future Work

Library Implementation

$$\mathbf{DFT}$$
$$\mathrm{S}_z\,\mathbf{DFT}\,\mathrm{G}_h$$
$$\mathrm{S}_h\,\mathbf{DFT}\,\mathrm{G}_z$$
$$\mathrm{S}_h\,\mathbf{DFT}\,\mathrm{diag}\left(\mathrm{Dat}\right)\mathrm{G}_h$$
$$\mathrm{S}_h\,\mathbf{DFT}\,\mathrm{G}_h$$
$$\mathrm{S}_h\,\mathbf{DFT}\,\mathrm{G}_{h\circ z}$$
$$\mathrm{S}_{h\circ z}\,\mathbf{DFT}\,\mathrm{G}_h$$
$$\mathrm{S}_h\,\mathbf{DFT}\,\mathrm{diag}\left(\mathrm{Dat}\right)\mathrm{G}_{h\circ z}$$
$$\mathrm{S}_z\,\mathbf{DFT}\,\mathrm{diag}\left(\mathrm{Dat}\right)\mathrm{G}_h$$
$$\mathrm{S}_{h\circ z}\,\mathbf{DFT}\,\mathrm{diag}\left(\mathrm{Dat}\right)\mathrm{G}_h$$



High-performance library

- **Input:**
  - Recursion step closure
  - Σ-SPL implementation of each recursion step (base cases + recursions)
- **Output:**
  - High-performance library
  - Target language: C++, Java, etc.
- **Process:**
  - Build library plan
  - Perform hot/cold partitioning
  - Generate target language code

What Kind of Code to Generate: Functions?

- **Example: computing DFT_{32} with Intel IPP**

```
IppsDFTSpec_C_64fc *f;
ippsDFTInitAlloc_C_64fc (f, 32, ...);
ippsDFTFwd_CToC_64fc(X, Y, f, NULL );
```

→ **initialize**

→ **compute**

- **In fact, DFT is a higher order function (HOF):**

$$N \longrightarrow (X \longrightarrow Y)$$

- **Descriptors = standard way to implement HOFs, also called "closures"**
  - "Closures" also equivalent to objects
  - We will implement recursion steps as C++ objects

- **Outer parameters = cold**
- **Inner parameters = hot**

Need to determine hot/cold parameters in recursion steps

Step 1: Build Library Plan

RS 1

- **Formula:** DFT_{u_1}
- **Parameters:** u_1
- **Children:** RS 2, RS 3
- **Implementation 1:** DFT-LibBase
  - *Applicable:* $u_1 = 2$
  - *Formula:* $\begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix}$
- **Implementation 2:** DFT-CTA
  - *Applicable:* isPrime(u_1) = 0
  - *Freedoms:* f ∈ divisors(u_1)
  - *Formula:* <…omitted…>

RS 2

- **Formula:** $\mathrm{S}_{\ldots} \mathrm{DFT}_{u_1} \mathrm{G}_{h^{u_1 \to u_6}_{u_7,u_8}}$
- **Parameters:** u_1, u_2, u_3, u_6, u_7, u_8
- **Children:** RS 2, RS 3
- **Implementation 1:** DFT-LibBase
  - *Applicable*: $u_1 = 2$
  - *Formula*: $\mathrm{S}_{h^{2\to u_2}_{u_3,1}} \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix} \mathrm{G}_{h^{2\to u_6}_{u_7,u_8}}$
- **Implementation 2:** DFT-CTA
  - *Applicable*: isPrime(u_1) = 0
  - *Freedoms*: f ∈ divisors(u_1)
  - *Formula*: <…omitted..>

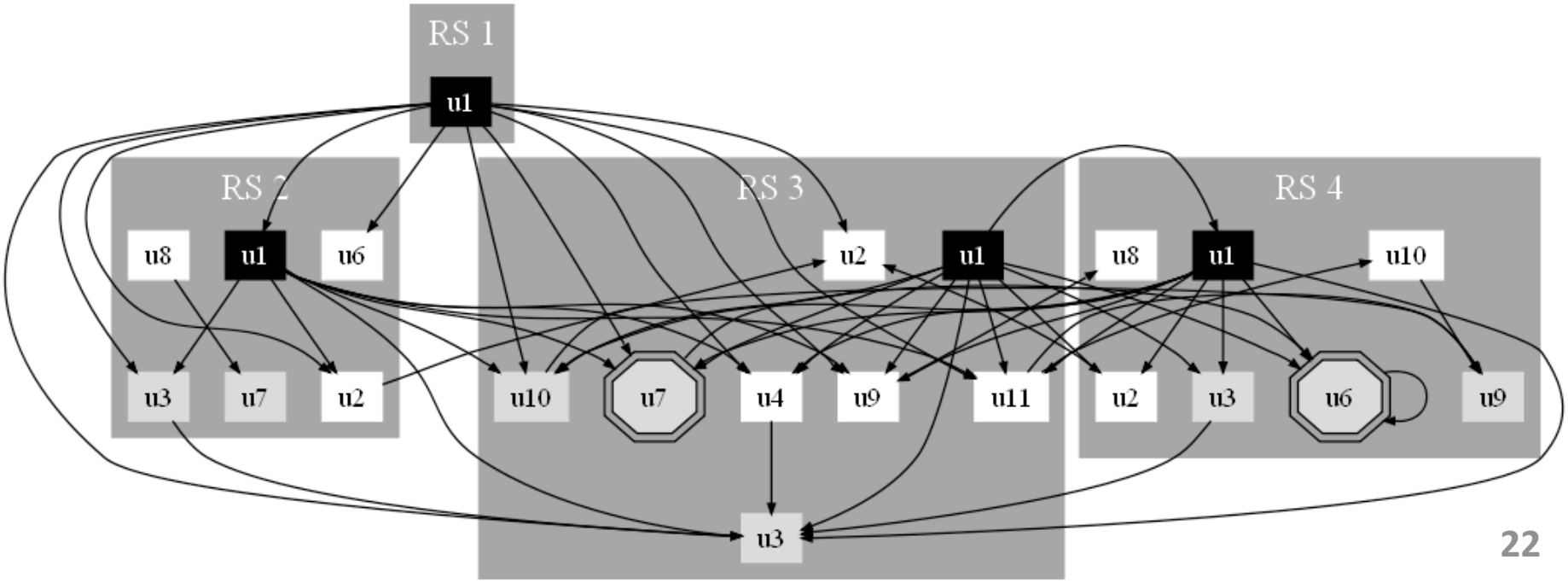
$$\sum_{i=0}^{u_1/f_1-1} \mathrm{RS3}\left((u_2, u_1, u_3, u_4, u_7^{\mathbb{Z}\to\mathbb{C}}, u_9, u_{10}, u_{11}) \leftarrow (u_1, f_1, i, u_1/f_1, \Omega^1_{u_1} \circ dt \circ h^{f_1\to u_1}_{i,\, u_1/f_1}, u_1, i, u_1/f_1)\right)$$

Parameters have dependencies

$$\sum_{j=0}^{f_1-1} \mathrm{RS2}\left((u_2, u_1, u_3, u_6, u_7, u_8) \leftarrow (u_1, u_1/f_1, u_1 j/f_1, u_1, j, f_1)\right)$$

Step 2: Hot/Cold Partition

- **Build the "parameter flow graph"**
- **Phase 1 (Backward IDA):**
  - Mark mandatory *cold* parameters
  - Propagate coldness
  - none ! cold ➔ cold ! cold
- **Phase 2 (Forward IDA):**
  - Mark mandatory *hot* parameters
  - Propagate hotness
  - hot ! none ➔ hot ! hot



Step 3: Generate Target Language Code

Generated Code (fragment)

```
class Env_1 : public Env {
  int _rule, f_1, u1;
  char *_dat;
  Env *child1, *child2;
  Env_1(int u1);
  void compute(double *Y, double *X);
};

class Env_2 : public Env {
  int _rule, u8;
  Func_Int_Int_Real *u4;
  char *_dat;
  Env *child1;
  Env_2(int u8, Func_Int_Int_Real *u4);
  void compute(double *Y, double *X, int
    u6, int u7, int u9, int u11, int u12,
    int u13);
};
```

User Code

```
Env_1 dft(1024);
dft.compute(Y, X);
```

I. Background

II. Library Generation: Structure

Recursion Step Closure

Base Cases

III. Library Generation: Implementation

IV. Experimental Results

V. Conclusions and Future Work

Experimental Results : Setup

- **Generated libraries are in C++, compiled with Intel C++ Compiler 10.1**
- **Use SSE intrinsics for vectorization**
  - 2-way for double precision
  - 4-way for single precision
- **Compared against**
  - FFTW 3.2 alpha 2
  - Intel IPP 5.3
- **Platform**
  - Intel Xeon 5160 3 GHz (2x2 cores, 8 MB L2$)

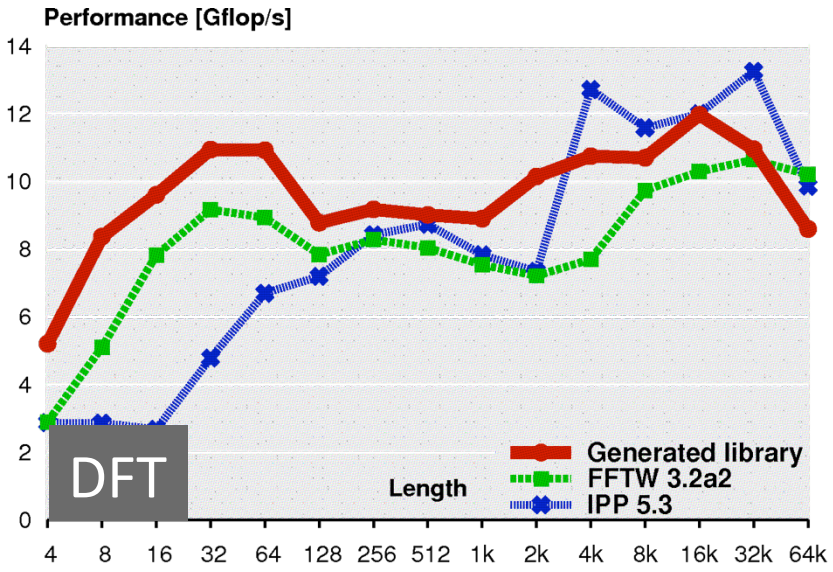
Code Size Snapshot

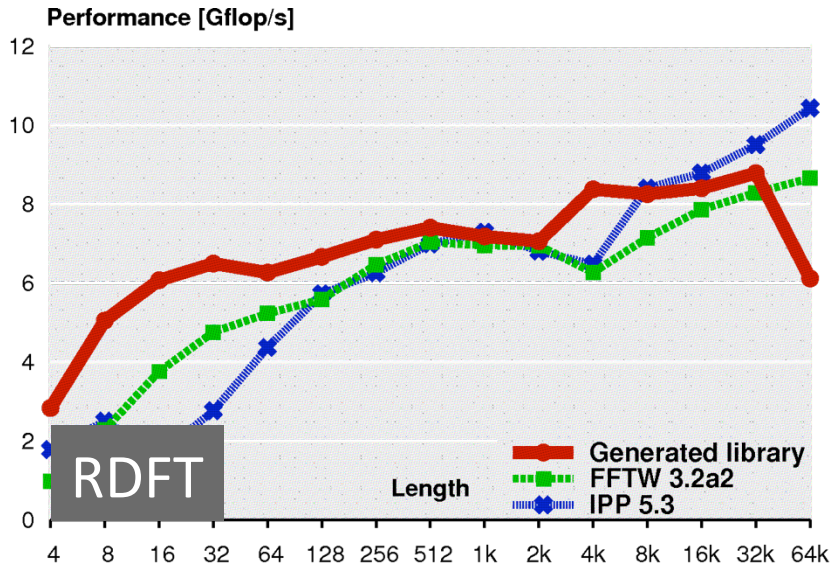
Transform	Code size	
	non-parallelized	parallelized
no vectorization		
DFT	13.1 KLOC / 0.59 MB	10.3 KLOC / 0.45 MB
RDFT	8.5 KLOC / 0.36 MB	8.8 KLOC / 0.39 MB
DHT	9.1 KLOC / 0.40 MB	9.4 KLOC / 0.39 MB
DCT-2	12.0 KLOC / 0.55 MB	12.4 KLOC / 0.57 MB
DCT-3	12.0 KLOC / 0.56 MB	12.3 KLOC / 0.59 MB
DCT-4	6.8 KLOC / 0.33 MB	7.1 KLOC / 0.35 MB
WHT	5.6 KLOC / 0.21 MB	—
4-way vectorization		
DFT	17.9 KLOC / 1.09 MB	18.2 KLOC / 1.11 MB
RDFT	16.2 KLOC / 0.86 MB	16.5 KLOC / 0.91 MB
scaled RDFT	16.5 KLOC / 0.88 MB	—
DHT	17.9 KLOC / 1.02 MB	18.3 KLOC / 1.04 MB
DCT-2	23.3 KLOC / 1.50 MB	23.6 KLOC / 1.53 MB
DCT-3	32.0 KLOC / 2.17 MB	32.3 KLOC / 2.20 MB
DCT-4	8.3 KLOC / 0.63 MB	8.6 KLOC / 0.66 MB
WHT	8.5 KLOC / 0.53 MB	6.9 KLOC / 0.4 MB
2D DFT	20.6 KLOC / 1.32 MB	20.8 KLOC / 1.33 MB
2D DCT-2	27.0 KLOC / 2.1 MB	27.2 KLOC / 2.11 MB
FIR Filter	109 KLOC / 5.69 MB	74 KLOC / 3.44 MB
Downsampled FIR Filter	151 KLOC / 7.7 MB	92 KLOC / 4.61 MB

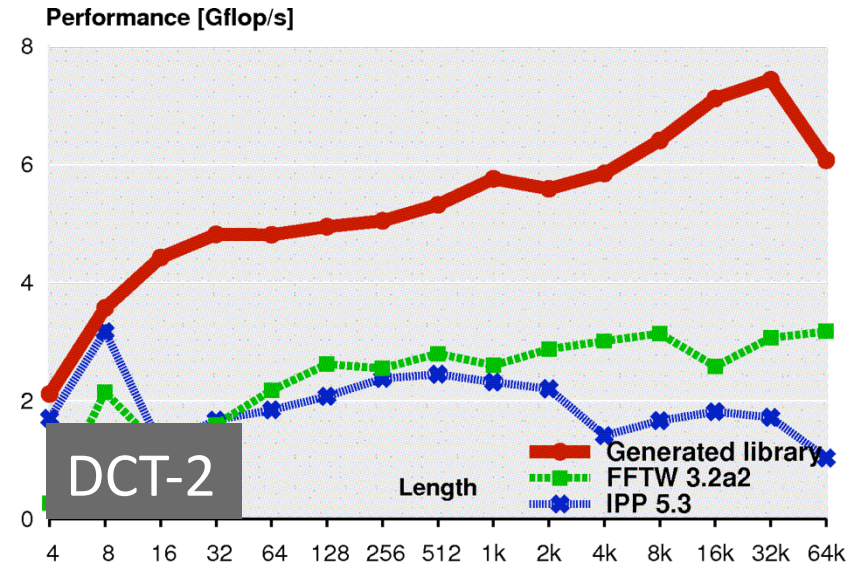
Quick turnaround time. **Example:** native algorithm for DCT-2

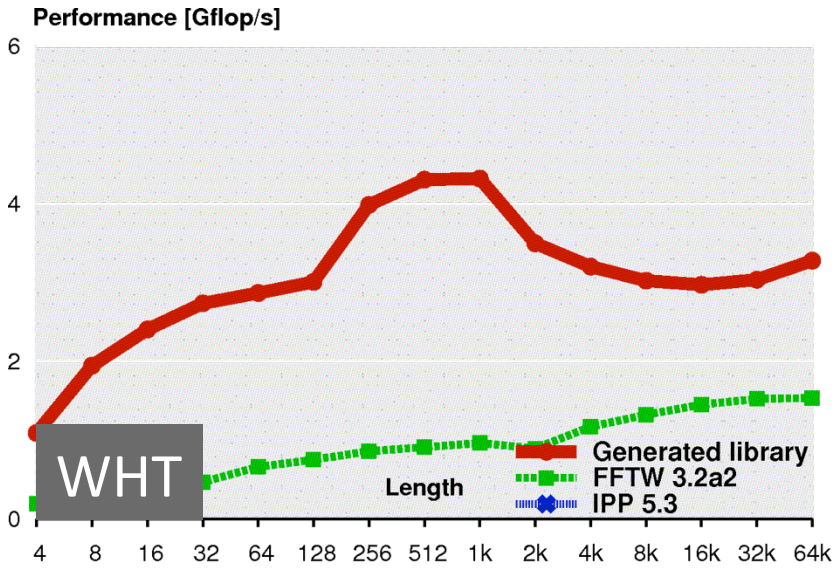
Double Precision Performance: Intel Xeon 5160

2-way vectorization, up to 2 threads



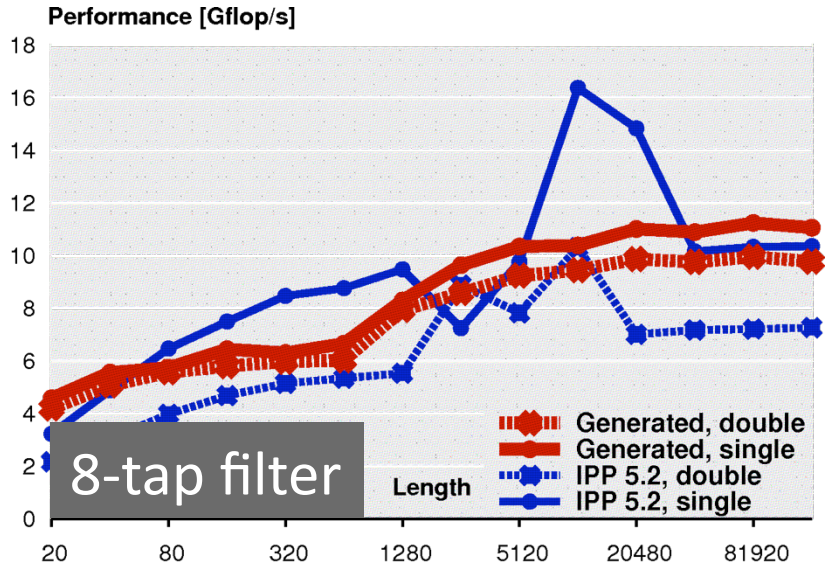


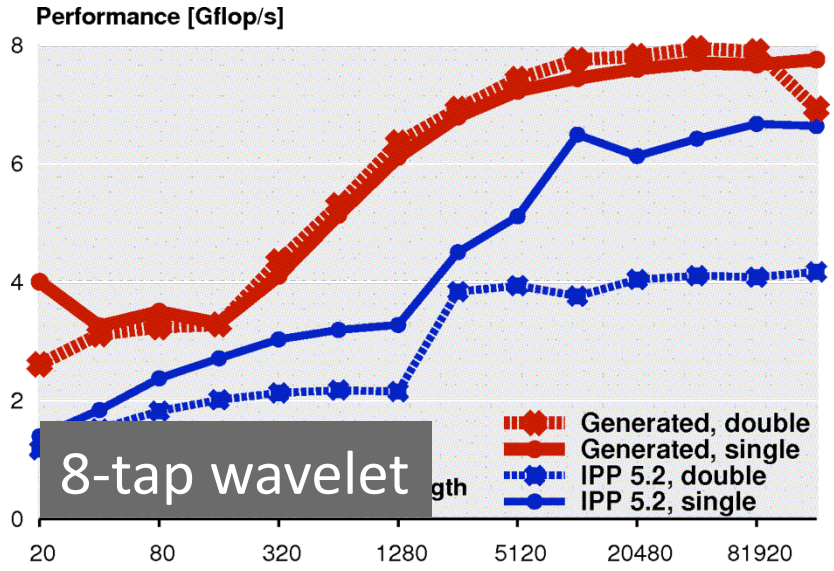


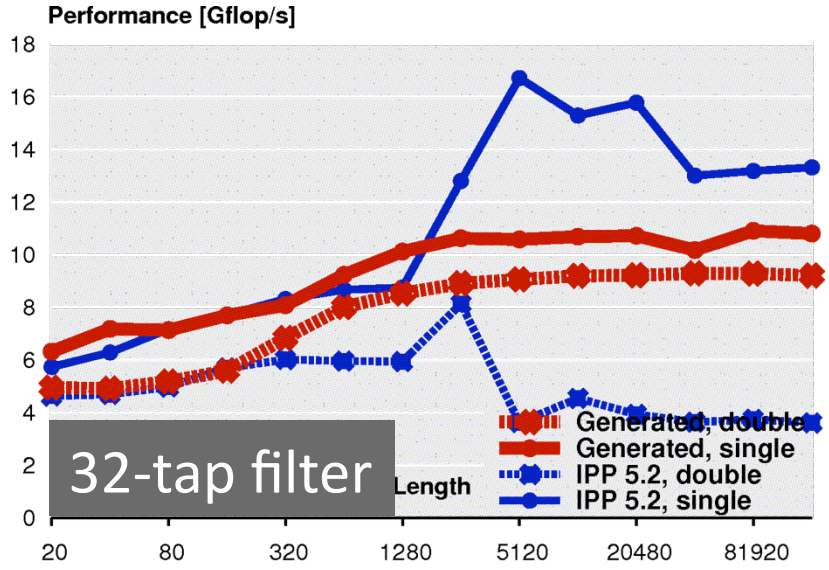


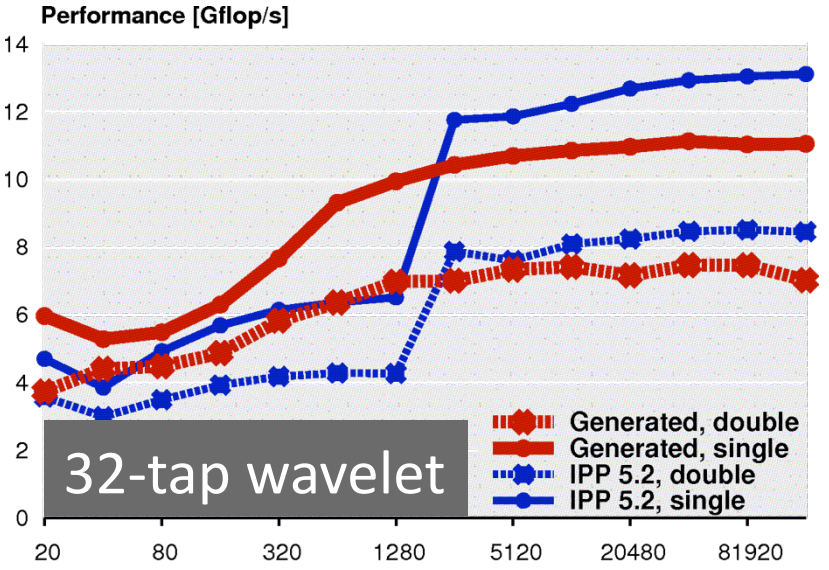
FIR Filter Performance

2- and 4-way vectorization, up to 2 threads



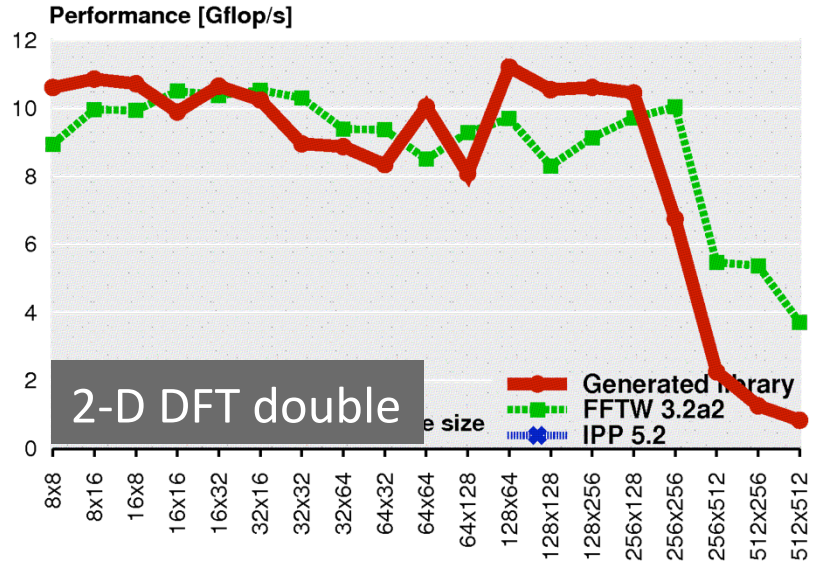


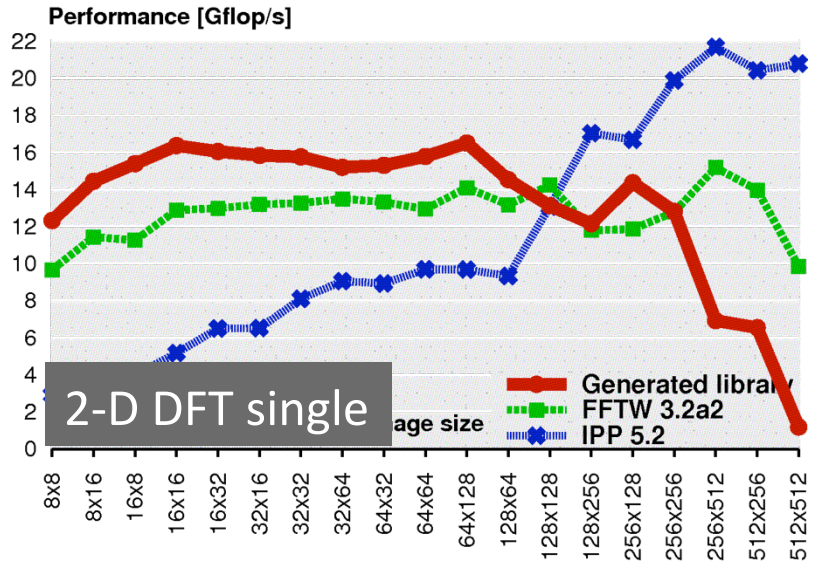


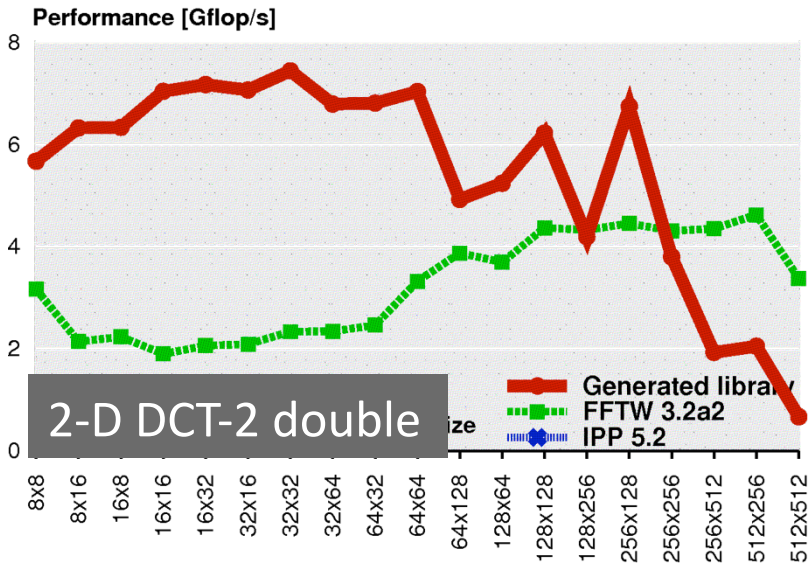


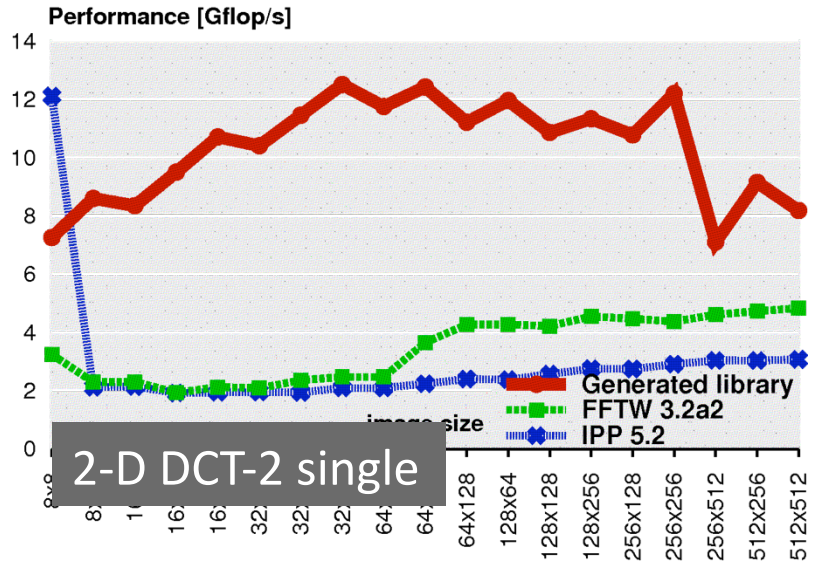
2-D Transforms Performance

2- or 4-way vectorization, up to 2 threads

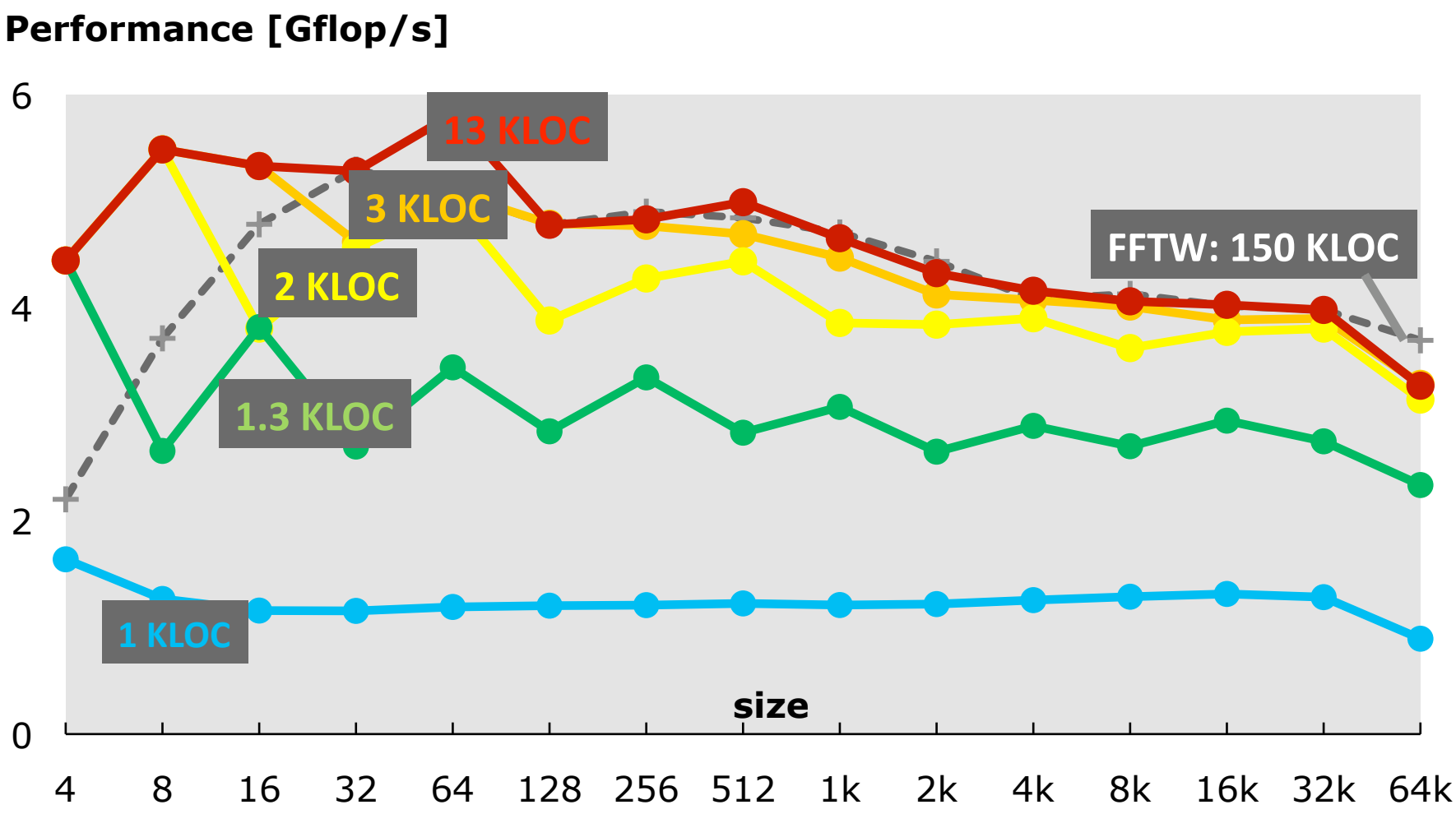








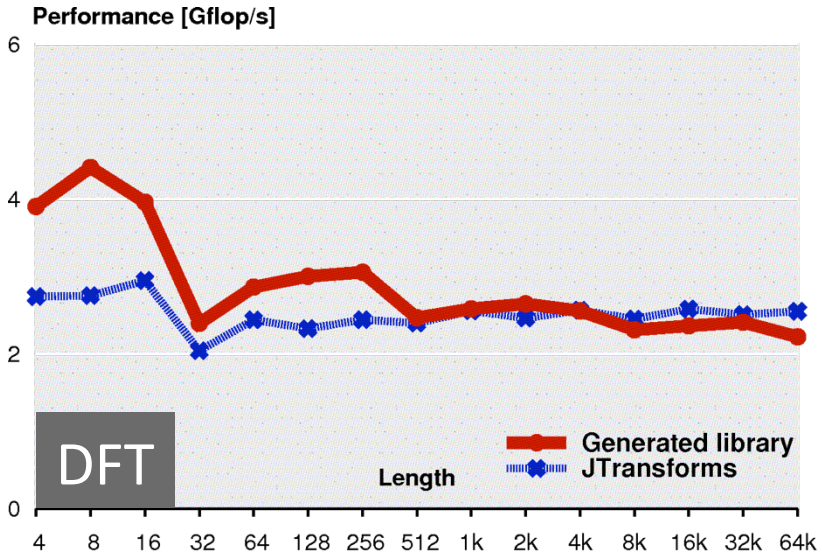
Qualitative Customization: Code Size

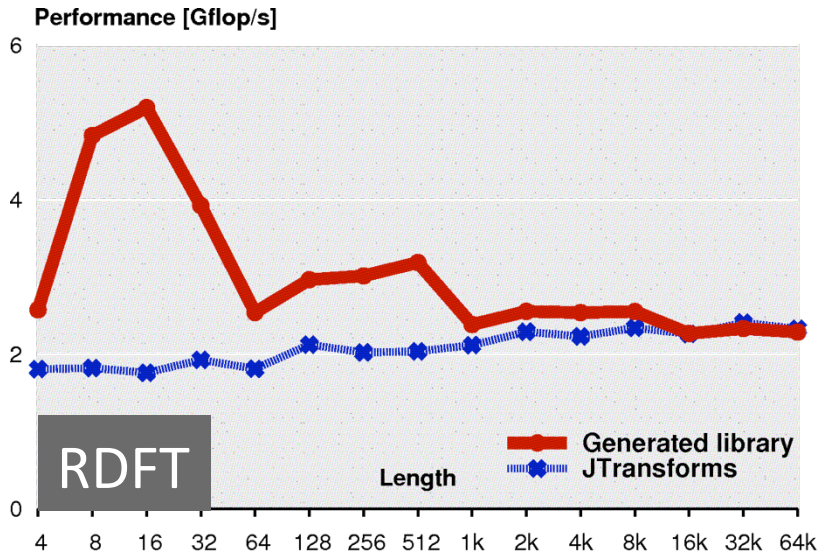


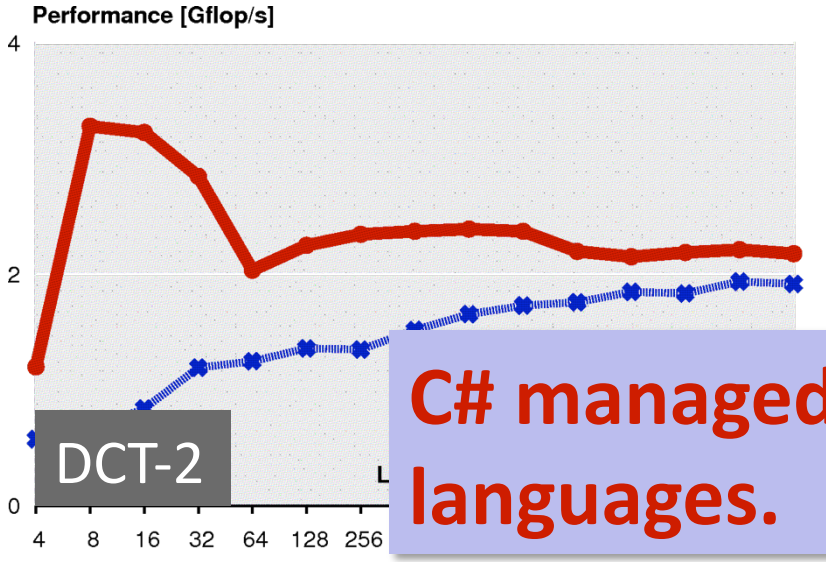
Code size controlled by changing # of base cases...
... but other possibilities are also available

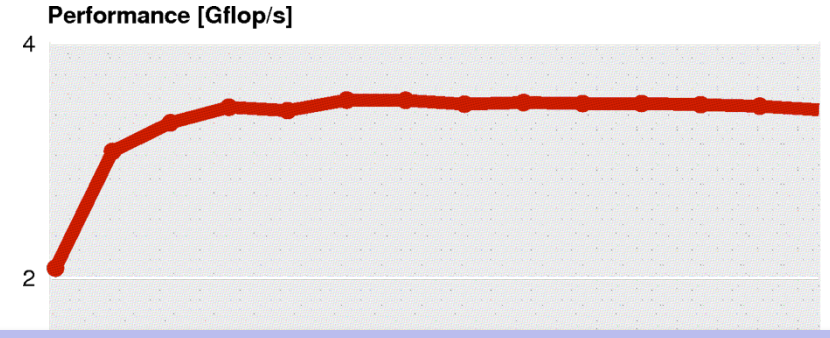
Backend Customization: Java

Joint work with Frédéric de Mesmay





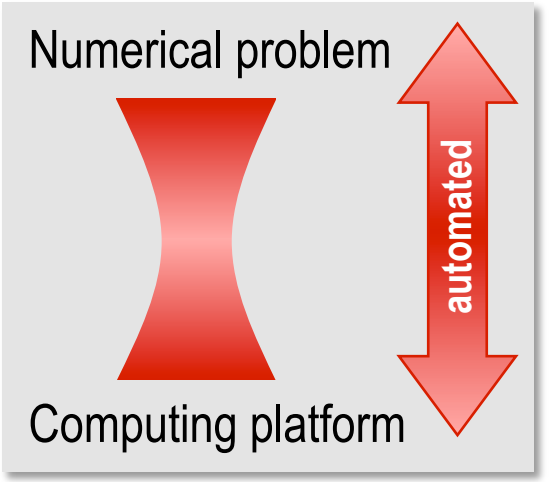




C# managed code possible, and other languages.

Future Work and Conclusions

- **Milestone:** Computer generation of high-performance libraries for the entire domain of linear transforms
- **Vertical integration (algorithm ! "FFTW")**
- **Important advantage: customization**
- **Ongoing work: Industry collaboration**
  - Robustness
  - Extra features
  - Completeness (full cross-product)



- **Future: new optimizations for last bits of performance**
  - Software pipelining (Cell, PowerPC)
  - Scheduling and register allocation (target = assembly code)
  - Advanced index mapping optimizations (e.g., sliding pointers)

Question: What can we do to build common tool bases for compiler-based autotuning and for construction of self -tuning or autotuning libraries?

- **Layers of DSLs very powerful idea**
- **Use rewriting to move between levels**
- **Existing generic tools:** Stratego, Maude, (Spiral?), others.
- **In Spiral:**
  - Transform
  - SPL
  - Sigma-SPL
  - Typed Sigma-SPL
  - Code IR 1
  - Code IR 2 -- binary ops, registers