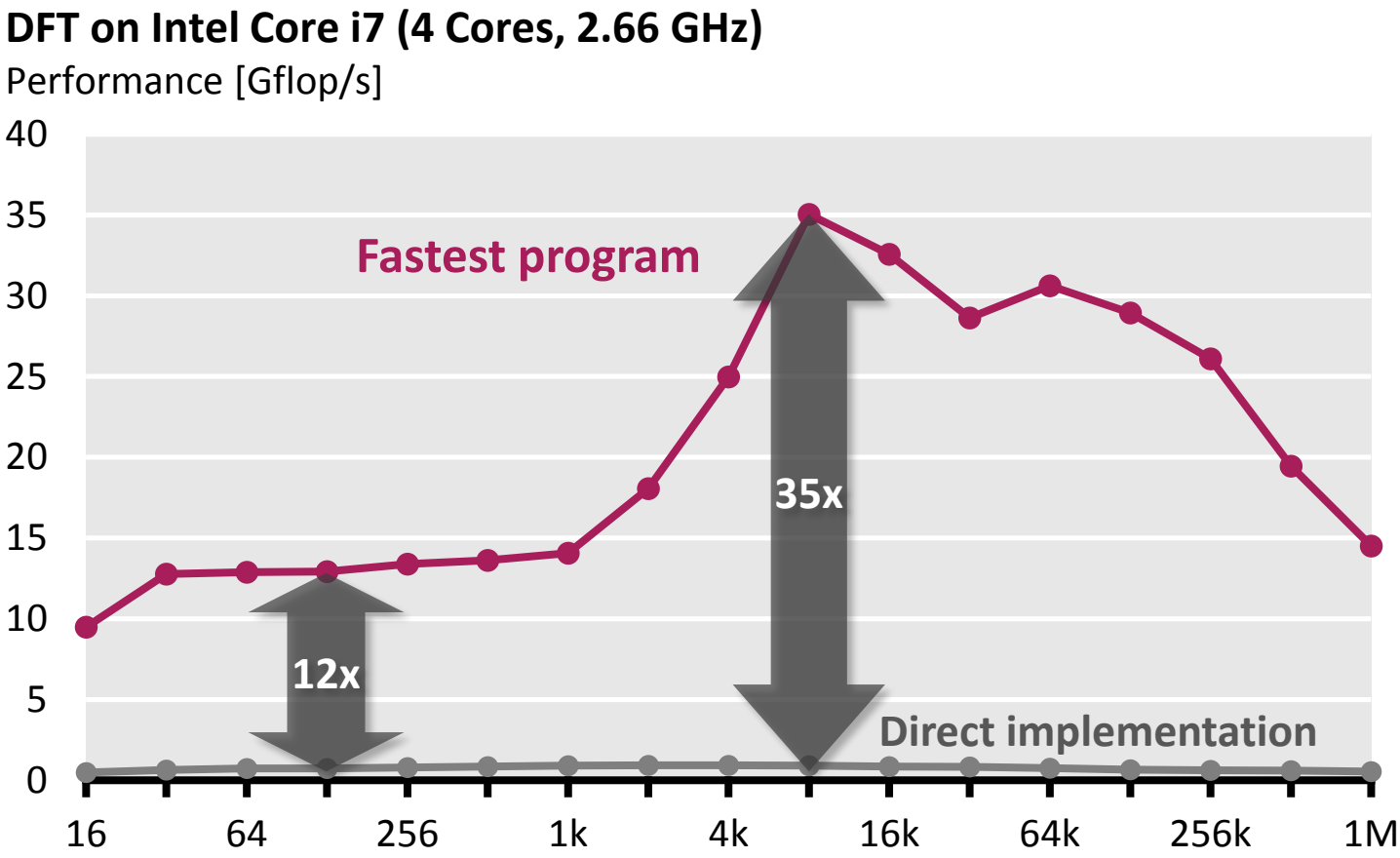
Building a Program Generator with LMS

Georg Ofenbeck
Alen Stojanov
Markus Püschel

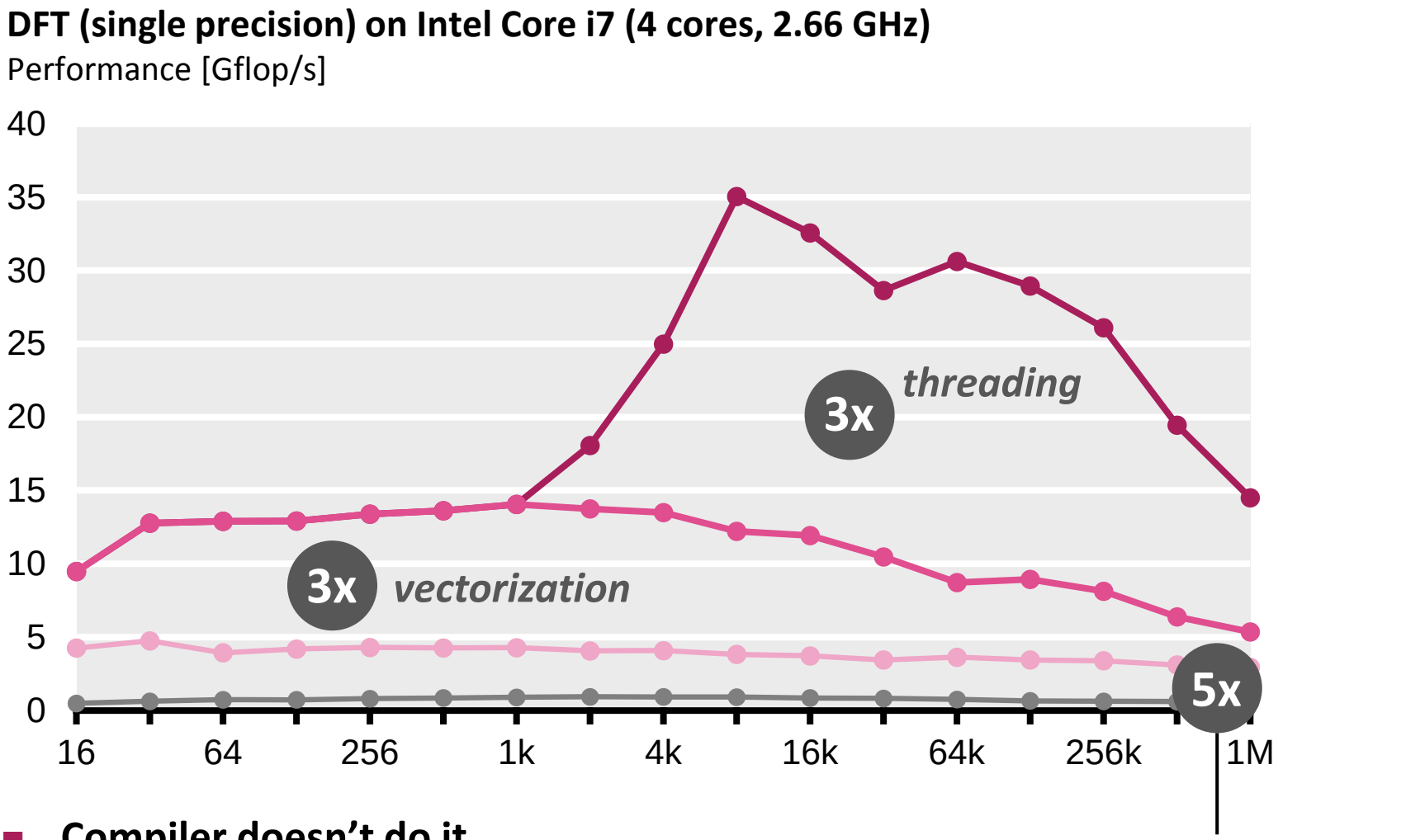
ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Why Program Generator? Example DFT



- Same number of operations
- Best compiler

DFT: Analysis



- **Compiler doesn't do it**
- **Doing by hand: Very tough**

Our Goal:

Computer writes high performance library code

Computer generated Functions for Intel IPP 6.0

3984 C functions

1M lines of code

Transforms: DFT (fwd+inv), RDFT (fwd+inv), DCT2, DCT3, DCT4, DHT, WHT

Sizes: 2–64 (DFT, RDFT, DHT); 2-powers (DCTs, WHT)

Precision: single, double

Data type: scalar, SSE, AVX (DFT, DCT), LRB (DFT)

Computer generated

Results: SpiralGen Inc.

So what is the problem?

Program language support for building program generators?

Requirements

- High level language support
  - Object / functional paradigm.
- Multiple levels of DSLs and DSL rewriting
- Performance transformations
  - Scalarization , Loop unrolling, Precomputation
- Abstraction over data representations
  - E.g. Complex: SplitComplex, InterleavedComplex, C99 etc.
- Rich environment
  - Databases, Libraries etc. etc.

Requirements

- Modern program language features
  - Object / functional paradigm.
- Multiple levels of DSLs and DSL rewriting
- Performance transformations
  - Scalarization , Loop unrolling, Precomputation
- Abstraction over data representations
  - E.g. Complex: SplitComplex, InterleavedComplex, C99 etc.
- Rich environment
  - Databases, Libraries etc. etc.

Algorithms: Example FFT, n = 4

Fast Fourier transform (FFT)

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

Representation using matrix algebra

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

- ***SPL (Signal processing language):*** **Mathematical, declarative, point-free**
- **Divide-and-conquer algorithms = breakdown rules in SPL**

Degrees of Freedom

radix

$$\mathbf{DFT}_{km} = (\mathbf{DFT}_k \otimes \mathrm{I}_m) T^{km}_m (\mathrm{I}_k \otimes \mathbf{DFT}_m) L^{km}_k$$

SPL to Code

SPL S	Pseudo code for $y = Sx$
$A_n B_n$	`<code for: t = Bx>` `<code for: y = At>`

$I_m \otimes A_n$	`for (i=0; i<m; i++)`
	`  <code for:`
	`  y[i*n:1:i*n+n-1] = A(x[i*n:1:i*n+n-1])>`

$A_m \otimes I_n$	`for (i=0; i<n; i++)`
	`  <code for:`
	`  y[i:n:i+m*n-n] = A(x[i:n:i+m*n-n])>`

D_n	`for (i=0; i<n; i++)` `  y[i] = D[i]*x[i];`

L^{km}_k	`for (i=0; i<k; i++)`
	`  for (j=0; j<m; j++)`
	`  y[i*m+j] = x[j*k+i];`

F_2	`y[0] = x[0] + x[1];` `y[1] = x[0] - x[1];`

$$\mathrm{I}_m \otimes A_n = \begin{bmatrix} A_n & & \\ & \ddots & \\ & & A_n \end{bmatrix}$$

SPL to Code

$$F2 = \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix} \qquad y = \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix} x$$

```
def F2( in: Array[Complex]): Array[Complex] =
{
   val out = new Array[Complex](in.size)
   val t1 = in(0);          val t2 = in(1)
   val r1 = t1 + t2;        val r2 = t1 - t2
   out.update(0,r1);    out.update(1,r2)
   out
}
```

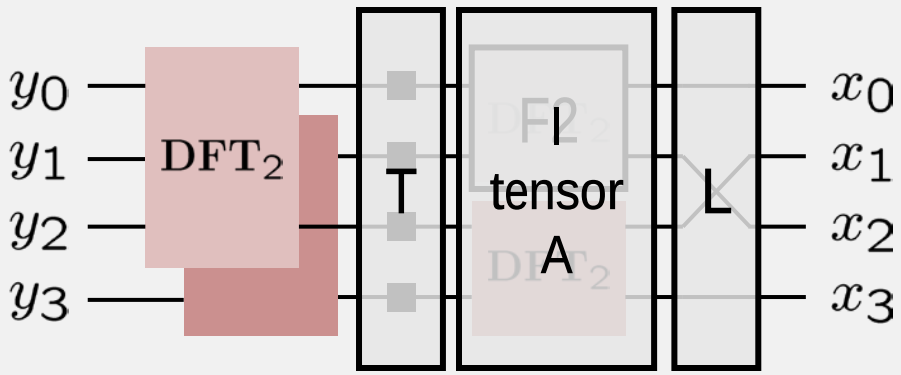
SPL to Code

$$I(2) \otimes F2 = \begin{bmatrix} 1 & 1 & . & . \\ 1 & -1 & . & . \\ . & . & 1 & 1 \\ . & . & 1 & -1 \end{bmatrix} \qquad y = \begin{bmatrix} 1 & 1 & . & . \\ 1 & -1 & . & . \\ . & . & 1 & 1 \\ . & . & 1 & -1 \end{bmatrix} x$$

```
def I_tensor_A (
                    in: Array[Complex],
                    in_size: Int, n: Int,
                    A: ((Array[Complex]) =>    Array[Complex])
                ): Array[Complex] =
          in.grouped(in_size/n).toArray flatMap (part => A(part))
```

SPL to Code

Data flow graph



Call graph

f(x) =>

x

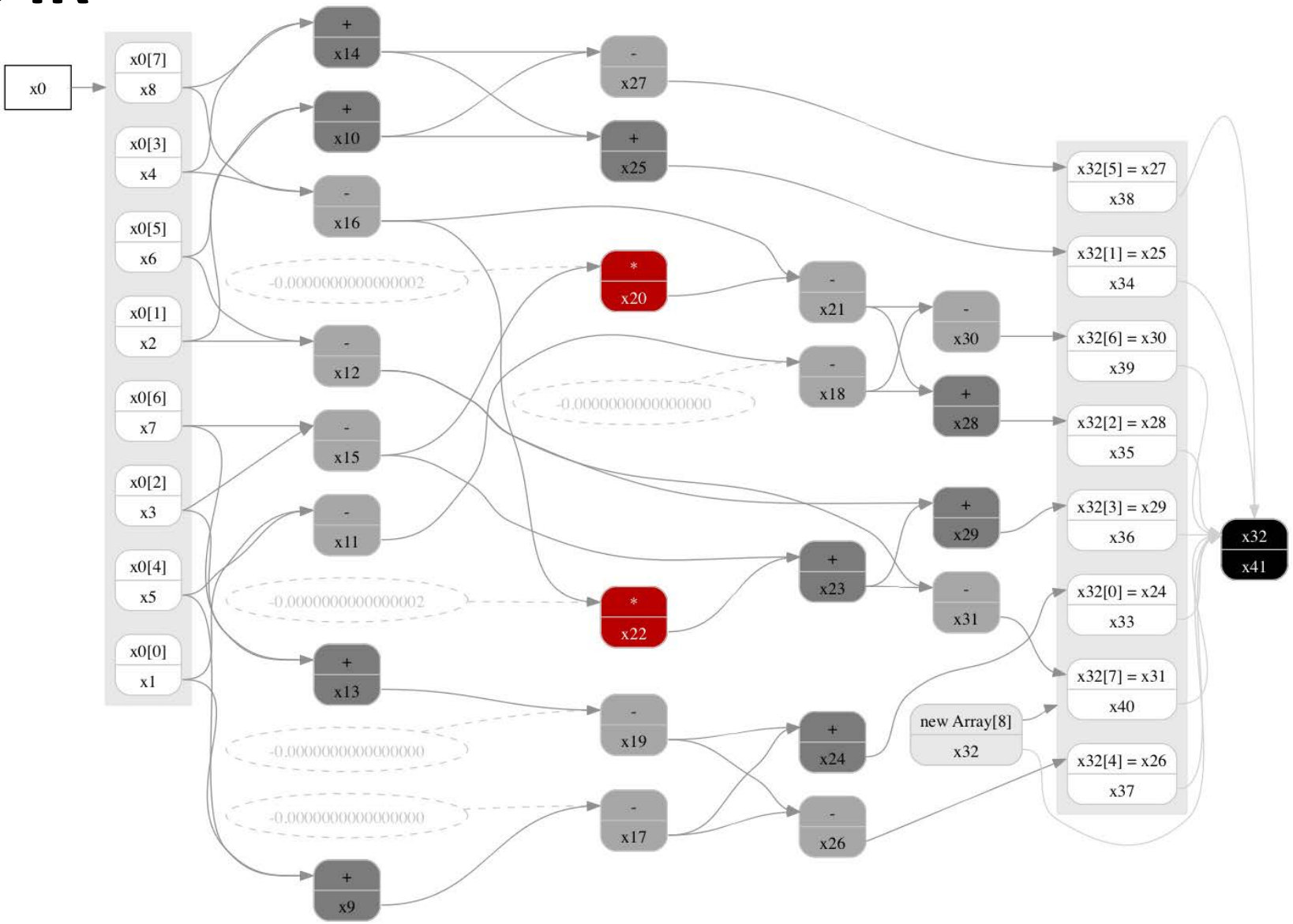
SPL DSL to CIR

```
case class Complex(_re:  Double , _im:  Double )
```

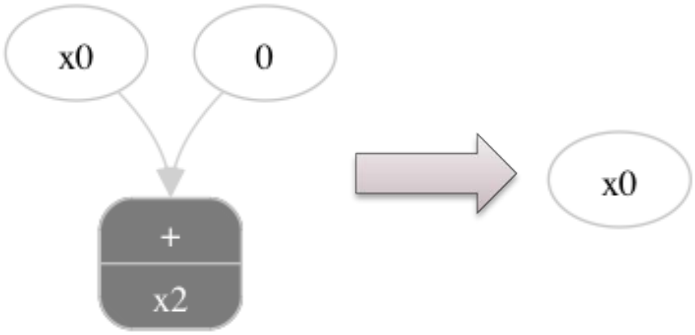
SPL DSL to CIR

```
case class Complex(_re: Rep[Double], _im: Rep[Double])
```

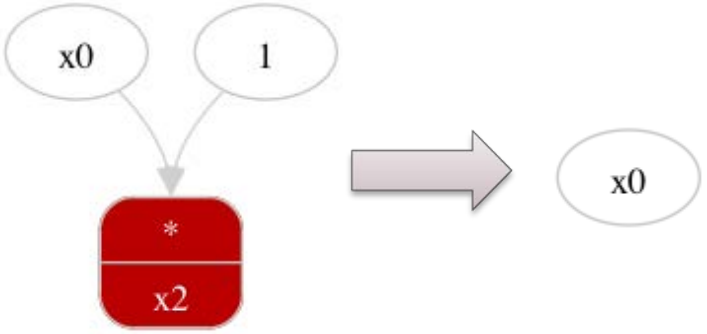
C-IR


x0
x0[7]
x8
x0[3]
x4
x0[5]
x6
x0[1]
x2
x0[6]
x7
x0[2]
x3
x0[4]
x5
x0[0]
x1
+ x14
+ x10
- x16
- x12
- x15
- x11
+ x13
+ x9
-0.00000000000000002
-0.00000000000000002
-0.00000000000000000
-0.00000000000000000
- x27
+ x25
* x20
* x22
- x21
- x18
+ x23
- x19
- x17
- x30
+ x28
+ x29
- x31
+ x24
- x26
new Array[8]
x32
x32[5] = x27
x38
x32[1] = x25
x34
x32[6] = x30
x39
x32[2] = x28
x35
x32[3] = x29
x36
x32[0] = x24
x33
x32[7] = x31
x40
x32[4] = x26
x37
x32
x41

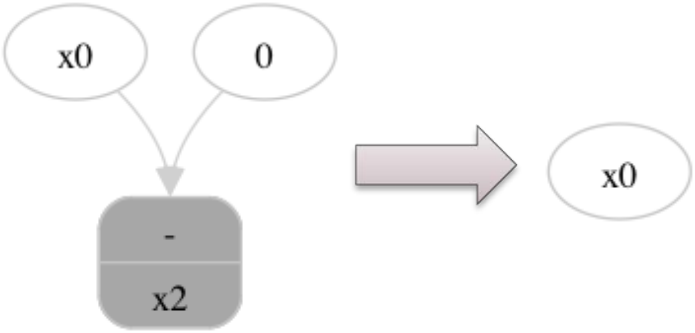
C-IR Optimizations



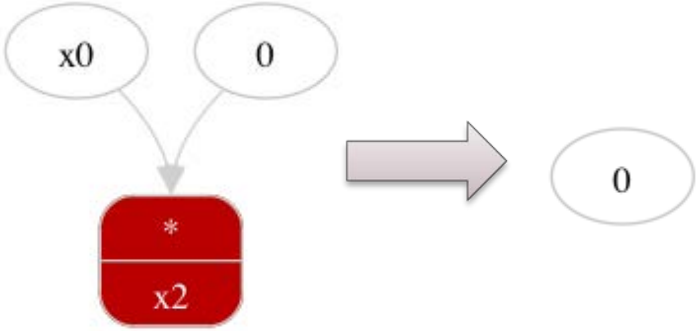
Addition with zero



Multiplication with one

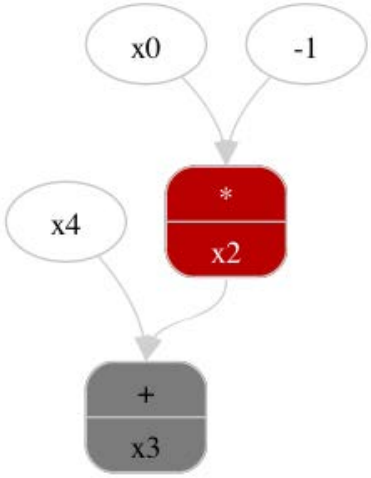


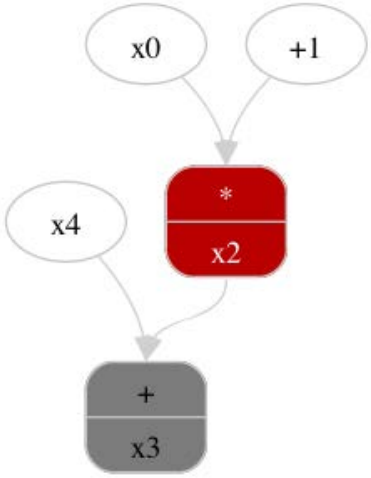
Subtraction with zero



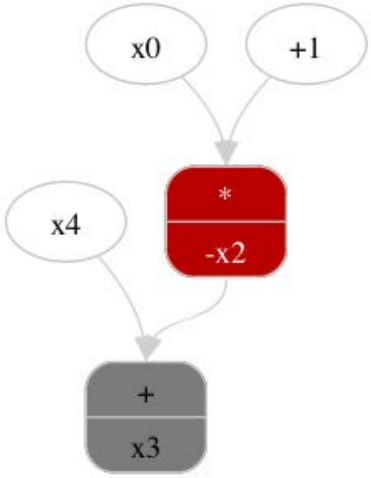
Multiplication with zero

C-IR Making all constants positive

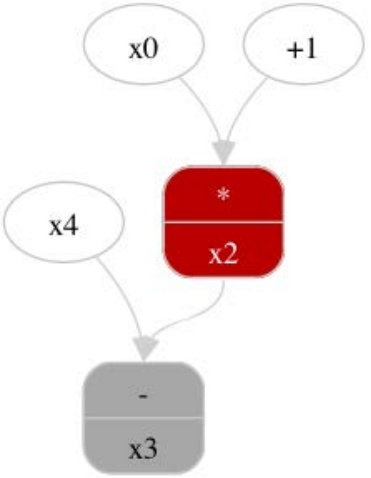




1. Make constant positive



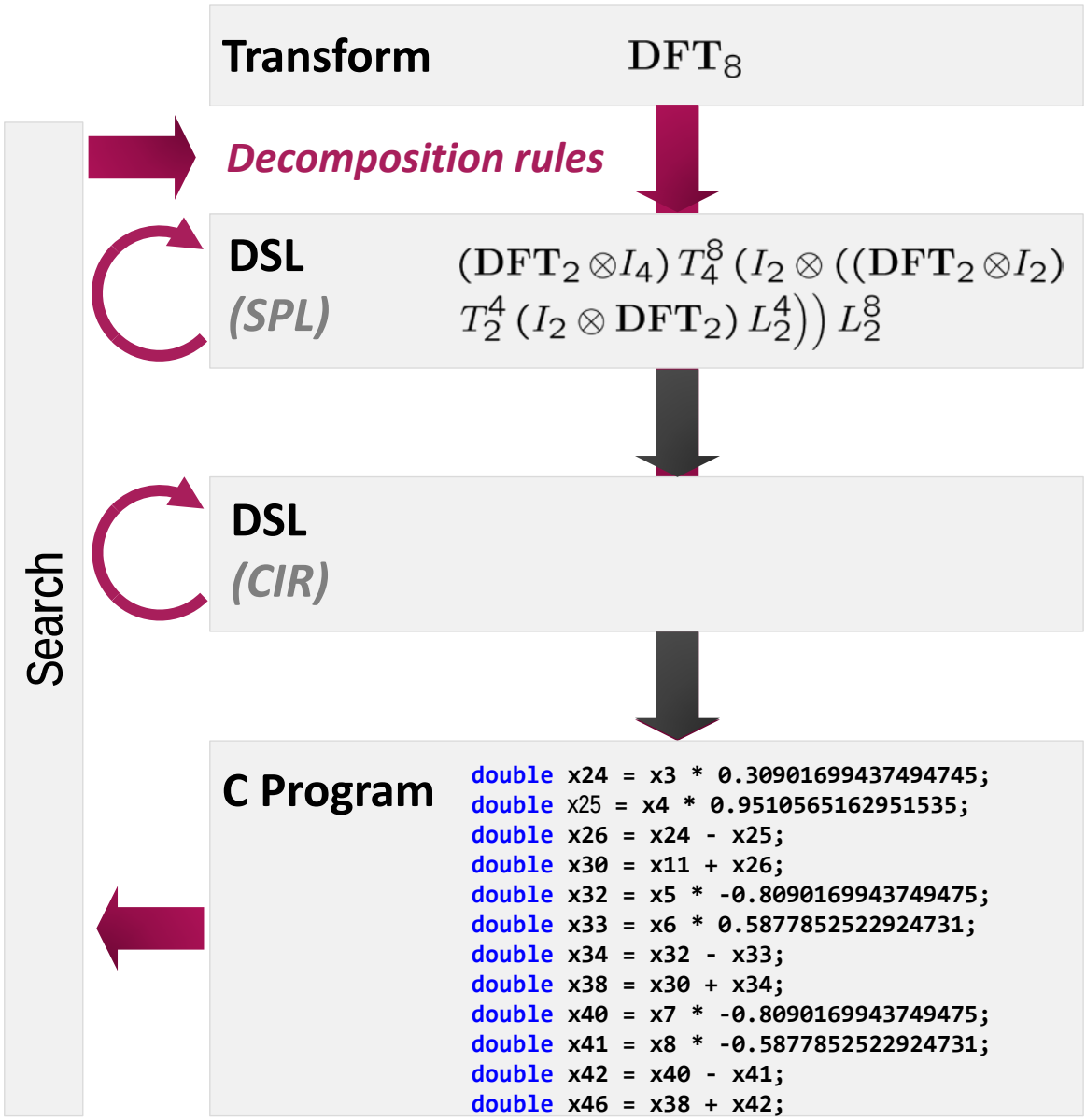
2. Annotate the symbol as negative (propagate)



3. Change the numeric expression (or propagate)

Differences to Delite

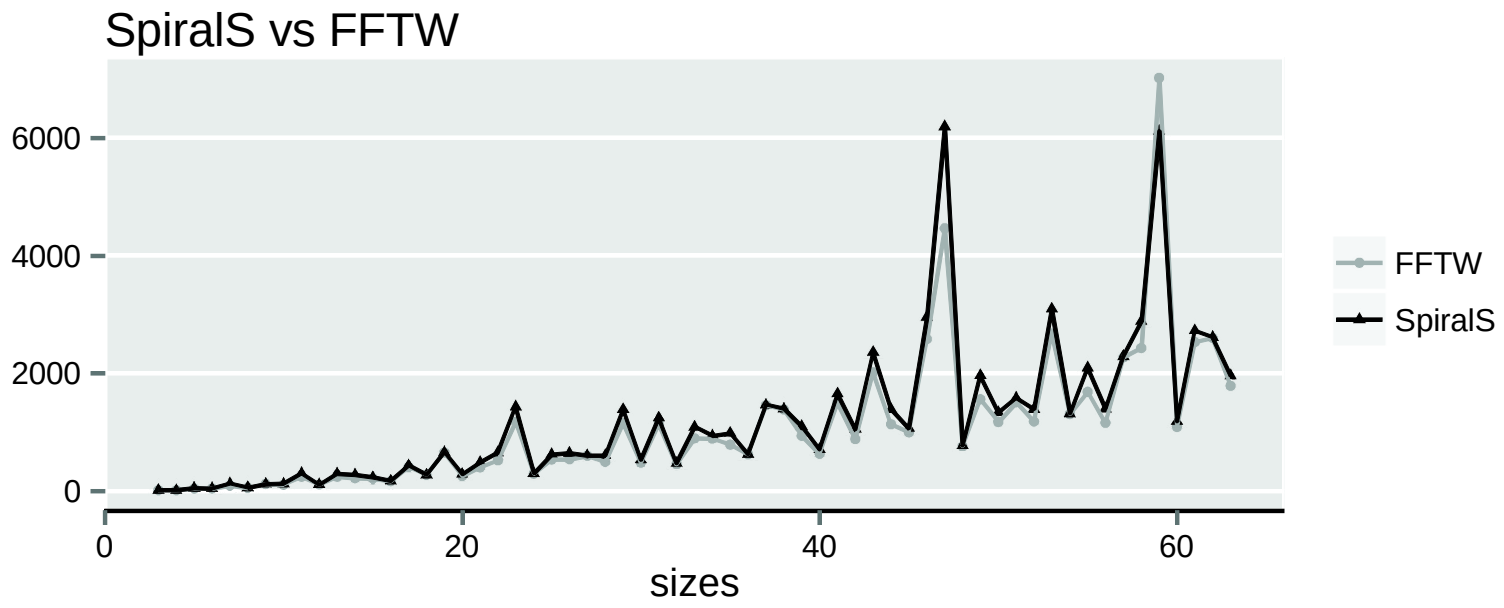
- Input a transform
- DSLs used internally
- Multiple levels of DSLs
- Heavy Rewrite per DLS
- Search



Result of the Tutorial Code (SpiralS Mini)

- **Can produce unrolled scalar replaced code**

- **Adding four more rules and 2 C IR rewrites yields scalar code equivalent to FFTW codelet generator**



What next?

Teaser: Abstracting over Staging

```
def F2(x: Array[Complex]): Array[Complex] =  ….

case class Complex (val re: Rep[Double], val im: Rep[Double]) {
    def +(that: Complex): Complex =
        new Complex(this.re + that.re,this.im + that.im)
        ……
}
```

```
...
double x24 = x3 * 0.30901699437494745;
double x25 = x4 * 0.9510565162951535;
double x26 = x24 - x25;
double x30 = x11 + x26;
double x32 = x5 * -0.8090169943749475;
double x33 = x6 * 0.5877852522924731;
double x34 = x32 - x33;
double x38 = x30 + x34;
double x40 = x7 * -0.8090169943749475;
double x41 = x8 * -0.5877852522924731;
double x42 = x40 - x41;
double x46 = x38 + x42;
double x48 = x9 * 0.30901699437494745;
double x49 = x10 * -0.9510565162951535;
double x50 = x48 - x49;
double x54 = x46 + x50;
...
```

Teaser: Abstracting over Staging

```
def F2(x: Array[Complex[T]): Array[Complex[T]] = {


case class Complex (val re: T, val im: T) {
      def +(that: Complex[T]): Complex[T] =
            new Complex[T](this.re + that.re,this.im + that.im)
            ……
}
```

T = Rep[Double] or Double

staging on / off

```
def F2(x: List[Rep[Complex[T]]): List[Rep[Complex[T]]
```

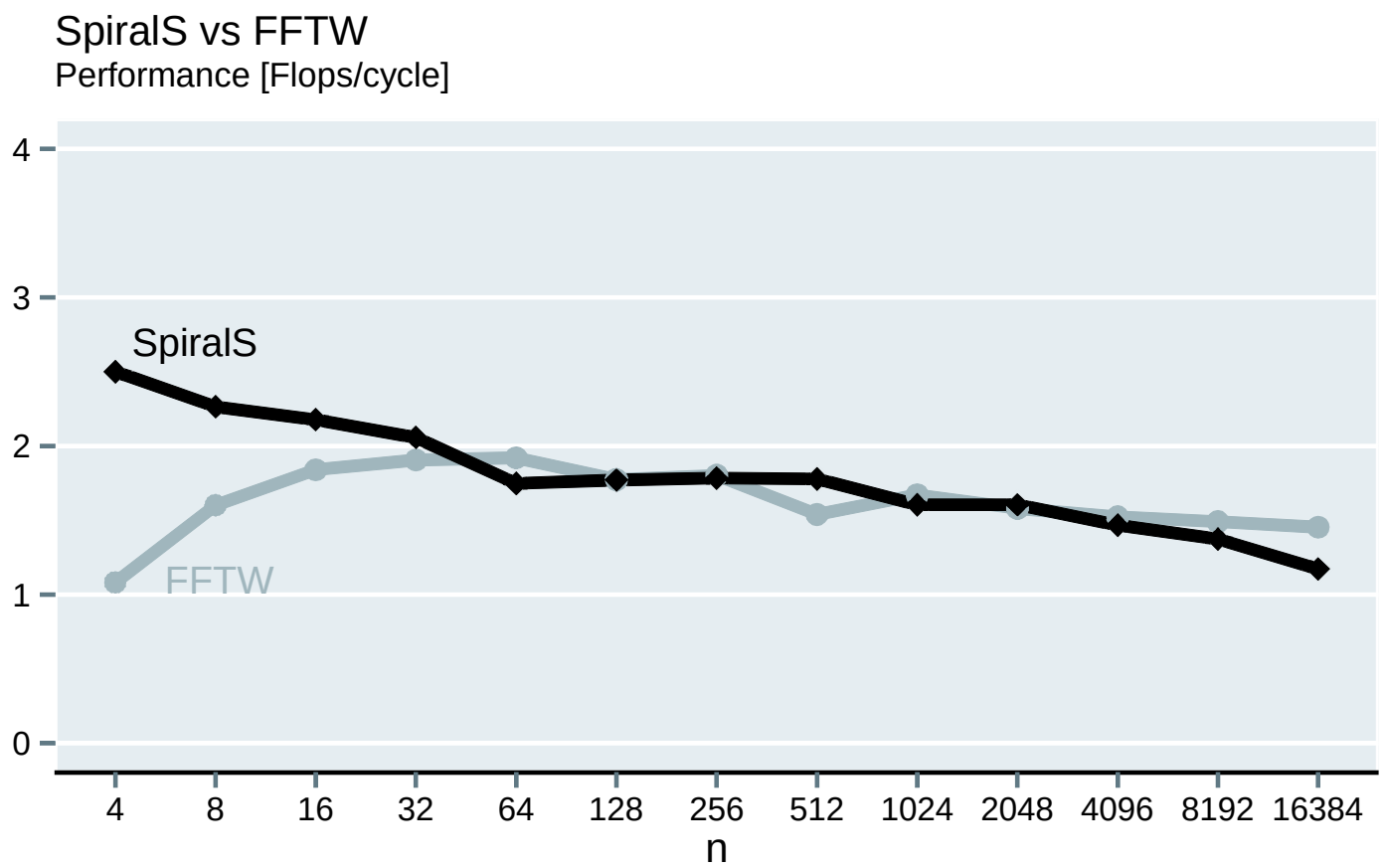
C99 Complex

```
def F2(x: Rep[Array [Complex[T]]]): Rep[Array[Rep[Complex[T]]]
```

non scalarized code

Teaser: Abstracting over Staging

- **Can abstract over**
  - Data Layout (Interleaved Complex, SplitComplex, C99 etc.)
  - Unrolling
  - Scalarization
  - Pre-computation



Spiral Reference

Markus Püschel, Franz Franchetti and Yevgen Voronenko
Spiral
in Encyclopedia of Parallel Computing, Eds. David Padua, Springer 2011