SPIRAL's Operator Language:
From Textbook Math to High Performance – With Correctness Guarantees

Franz Franchetti

Department of Electrical and Computer Engineering
Carnegie Mellon University
www.ece.cmu.edu/~franzf

Joint work with the SPIRAL team

This work was supported by DARPA, DOE, ONR, NSF, Intel, Mercury, and Nvidia

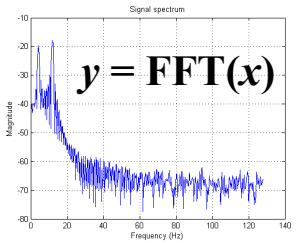
Idea: Go from Mathematics to Software

Given:

- **Mathematical problem specification**
  does not change
- **Computer platform**
  changes often

Wanted:

- **Very good implementation of specification on platform**
- **Proof of correctness**



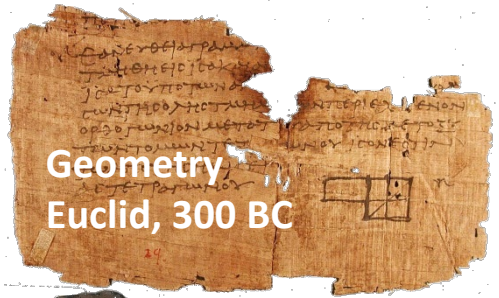
on

automatic

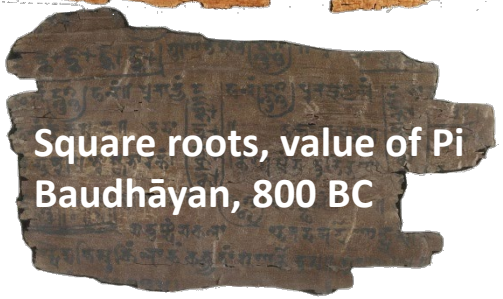
```
void fft64(double  *Y, double  *X) {
    ...
    s5674 = _mm256_permute2f128_pd(s5672, s5673, (0) | ((2) << 4));
    s5675 = _mm256_permute2f128_pd(s5672, s5673, (1) | ((3) << 4));
    s5676 = _mm256_unpacklo_pd(s5674, s5675);
    s5677 = _mm256_unpackhi_pd(s5674, s5675);
    s5678 = *((a3738 + 16));
    s5679 = *((a3738 + 17));
    s5680 = _mm256_permute2f128_pd(s5678, s5679, (0) | ((2) << 4));
    s5681 = _mm256_permute2f128_pd(s5678, s5679, (1) | ((3) << 4));
    s5682 = _mm256_unpacklo_pd(s5680, s5681);
    s5683 = _mm256_unpackhi_pd(s5680, s5681);
    t5735 = _mm256_add_pd(s5676, s5682);
    t5736 = _mm256_add_pd(s5677, s5683);
    t5737 = _mm256_add_pd(s5670, t5735);
    t5738 = _mm256_add_pd(s5671, t5736);
    t5739 = _mm256_sub_pd(s5670, _mm256_mul_pd(_mm_vbroadcast_sd(&(C22)), t5735));
    t5740 = _mm256_sub_pd(s5671, _mm256_mul_pd(_mm_vbroadcast_sd(&(C22)), t5736));
    t5741 = _mm256_mul_pd(_mm_vbroadcast_sd(&(C23)), _mm256_sub_pd(s5677, s5683));
    t5742 = _mm256_mul_pd(_mm_vbroadcast_sd(&(C23)), _mm256_sub_pd(s5676, s5682));
    ...
}
```

QED.

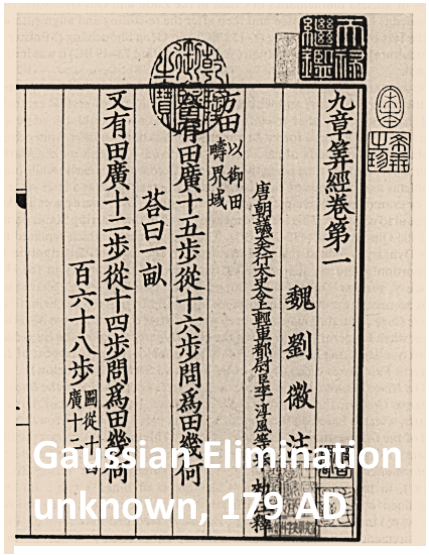
Algorithms and Mathematics: 2,500+ Years



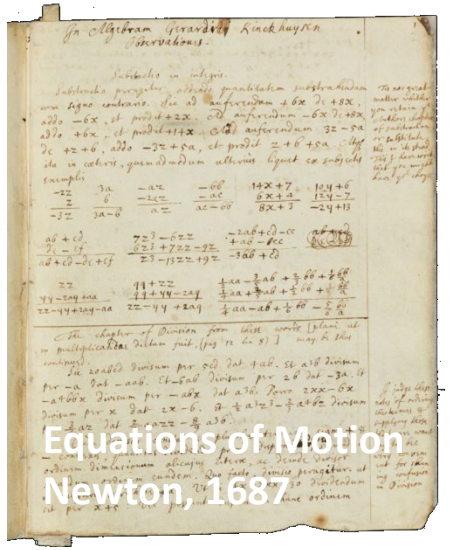
Geometry
Euclid, 300 BC



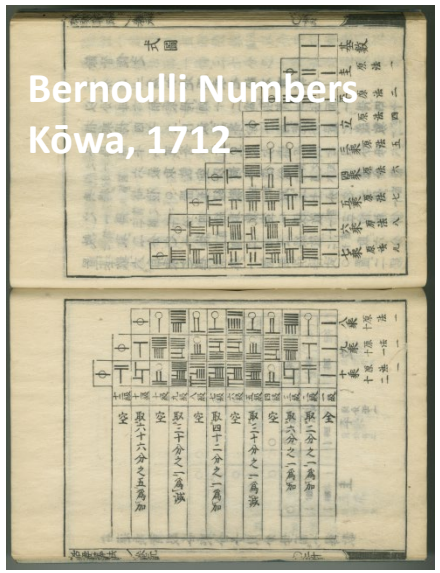
Square roots, value of Pi
Baudhāyan, 800 BC



Gaussian Elimination
unknown, 179 AD



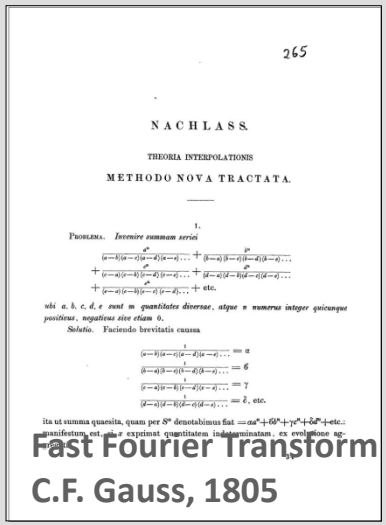
Equations of Motion
Newton, 1687



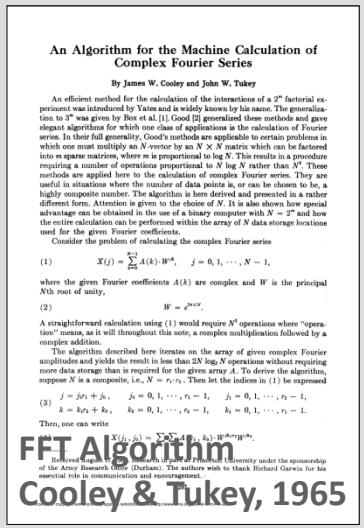
Bernoulli Numbers
Kōwa, 1712

Algebra
al-Khwārizmī, 830

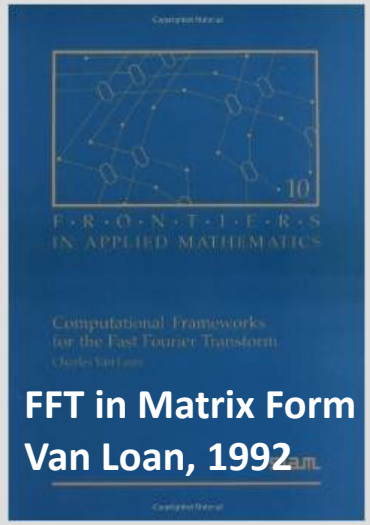
Fast Fourier Transform



Fast Fourier Transform
C.F. Gauss, 1805



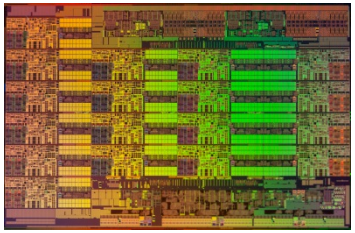
FFT Algorithm
Cooley & Tukey, 1965



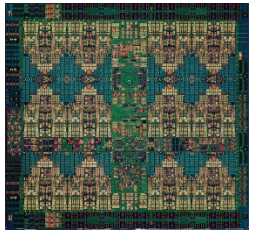
FFT in Matrix Form
Van Loan, 1992

Today's Computing Landscape

1 Gflop/s = one billion floating-point operations (additions or multiplications) per second

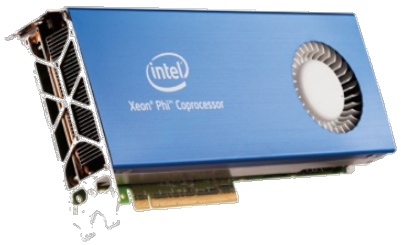


Intel Xeon 8180M
2.25 Tflop/s, 205 W
28 cores, 2.5—3.8 GHz
2-way—16-way AVX-512



IBM POWER9
768 Gflop/s, 300 W
24 cores, 4 GHz
4-way VSX-3

Nvidia Tesla V100
7.8 Tflop/s, 300 W
5120 cores, 1.2 GHz
32-way SIMT



Intel Xeon Phi 7290F
1.7 Tflop/s, 260 W
72 cores, 1.5 GHz
8-way/16-way LRBni

Snapdragon 835
15 Gflop/s, 2 W
8 cores, 2.3 GHz
A540 GPU, 682 DSP, NEON

Intel Atom C3858
32 Gflop/s, 25 W
16 cores, 2.0 GHz
2-way/4-way SSSE3

Dell PowerEdge R940
3.2 Tflop/s, 6 TB, 850 W
4x 24 cores, 2.1 GHz
4-way/8-way AVX

Summit
187.7 Pflop/s, 13 MW
9,216 x 22 cores POWER9
+ 27,648 V100 GPUs

Approach

- **Develop synthesis framework**
  operator language (OL), rewriting system

  Formal framework

- **Formalize algorithms**
  express algorithm as OL rewrite rules

  Algorithm formalization

- **Provide performance portability**
  optimization through rewriting

  Platform optimization

- **Correctness proof**
  rule chain = formal proof,
  symbolic evaluation and execution

  Formal proof

T. M. Low, F. Franchetti: **High Assurance Code Generation for Cyber-Physical Systems**
Proceedings of the 18th IEEE International Symposium on High Assurance Systems Engineering (HASE 2017), Jan 2017

Inspiration: Symbolic Integration

- **Rule based**
  basic functions, substitution
- **May not succeed**
  not all expressions can be symbolically integrated
- **Arbitrarily extensible**
  define new functions as integrals
  Γ(.), distributions, Lebesgue integral
- **Semantics preserving**
  rule chain = formal proof
- **Automation**
  Mathematica, Maple

Table of Integrals

BASIC FORMS

(1) $\int x^n dx = \frac{1}{n+1} x^{n+1}$

(2) $\int \frac{1}{x} dx = \ln x$

(3) $\int u dv = uv - \int v du$

(4) $\int u(x)v'(x)dx = u(x)v(x) - \int v(x)u'(x)dx$

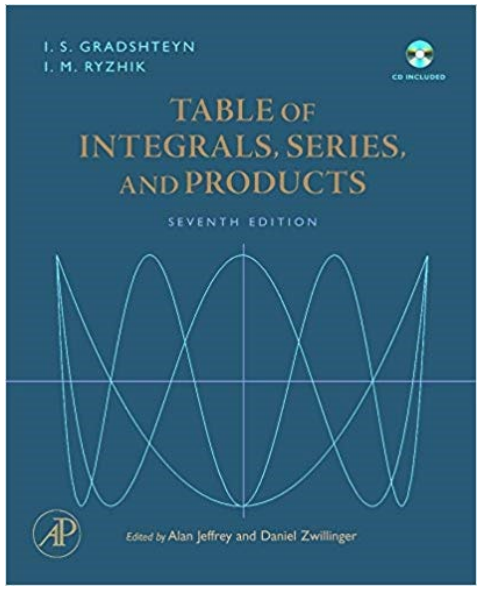
RATIONAL FUNCTIONS

(5) $\int \frac{1}{ax+b} dx = \frac{1}{a}\ln(ax+b)$

(6) $\int \frac{1}{(x+a)^2} dx = \frac{-1}{x+a}$

(7) $\int (x+a)^n dx = (x+a)^n \left(\frac{a}{1+n} + \frac{x}{1+n}\right), \; n \neq -1$

(8) $\int x(x+a)^n dx = \frac{(x+a)^{1+n}(nx+x-a)}{(n+2)(n+1)}$




```
In[31]:= Integrate[1/(a^2 Cos[t]^2 + b^2 Sin[t]^2), {t, 0, 2 Pi}]

Out[31]= 2 Sqrt[b^2/a^2] Pi / b^2
```

Wolfram Mathematica

```
In[33]:= Integrate[1/(a^2 ((E^(I t) + E^(-I t))/2)^2 + b^2 ((E^(I t) - E^(-I t))/(2 I))^2), t]

Out[33]= 0
```

OL Operators

Definition

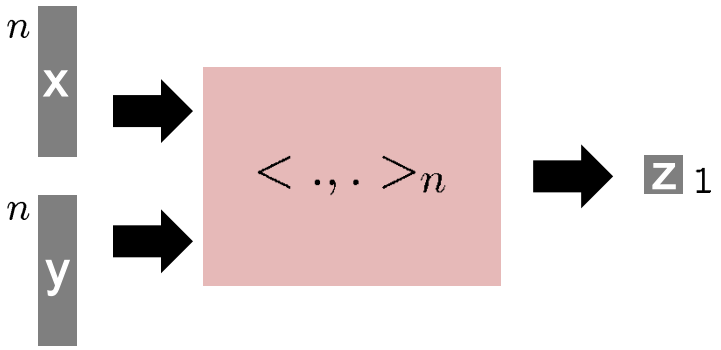
- **Operator: Multiple vectors ! Multiple vectors**
- **Stateless**
- **Higher-dimensional data is linearized**
- **Operators are potentially nonlinear**

$$\mathsf{M}: \begin{cases} \mathbb{C}^{n_0} \times \ldots \times \mathbb{C}^{n_{k-1}} \to \mathbb{C}^{N_0} \times \ldots \times \mathbb{C}^{N_{\ell-1}} \\ (\mathbf{x}_0, \mathbf{x}_1, \ldots, \mathbf{x}_{k-1}) \mapsto \mathsf{M}(\mathbf{x}_0, \mathbf{x}_1, \ldots, \mathbf{x}_{k-1}) \end{cases}$$

Example: Scalar product

$$< ., . >_n : \mathbb{R}^n \times \mathbb{R}^n \to \mathbb{R}$$

$$\left((x_i)_{i=0,\ldots,n-1}, (y_i)_{i=0,\ldots,n-1}\right) \mapsto \sum_{i=0}^{n-1} x_i y_i$$

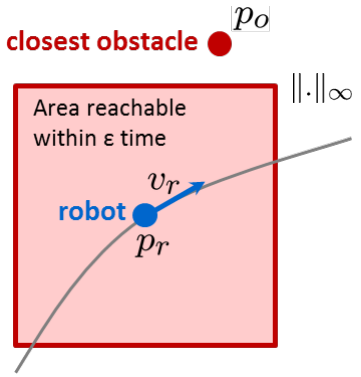


F. Franchetti, F. de Mesmay, Daniel McFarlin, and M. Püschel, **"Operator Language: A Program Generation Framework for Fast Kernels,"** Proceedings of IFIP Working Conference on Domain Specific Languages (DSL WC), 2009. ***Best Paper Award.***

Safety Distance as OL Operator

- **Passive Safety of Robots**

  p_o: Position of closest obstacle
  p_r: Position of robot
  v_r: Longitudinal velocity of robot
  A, b, V, ": constants



$$\|p_r - p_o\|_\infty > \frac{v_r^2}{2b} + V\frac{v_r}{b} + \left(\frac{A}{b} + 1\right)\left(\frac{A}{2}\varepsilon^2 + \varepsilon(v_r + V)\right)$$

- **Definition as operator**

$$\text{SafeDist}_{V,A,b,\varepsilon}: \quad \mathbb{R} \times \mathbb{R}^2 \times \mathbb{R}^2 \to \mathbb{Z}_2$$
$$(v_r, p_r, p_o) \mapsto \Big(p(v_r) < d_\infty(p_r, p_o)\Big) \quad \text{with} \quad d_\infty(\vec{x}, \vec{y}) = \|\vec{x} - \vec{y}\|_\infty$$

$$p(x) = \alpha x^2 + \beta x + \gamma$$
$$\alpha = \frac{1}{2b}$$
$$\beta = \frac{V}{b} + \varepsilon\left(\frac{A}{b} + 1\right)$$
$$\gamma = \left(\frac{A}{b} + 1\right)\left(\frac{A}{2}\varepsilon^2 + \varepsilon V\right)$$

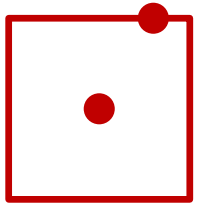
Formalizing Mathematical Objects in OL

- **Infinity norm**

$$\|.\|_\infty^n : \mathbb{R}^n \to \mathbb{R}$$
$$(x_i)_{i=0,\ldots,n-1} \mapsto \max_{i=0,\ldots,n-1} |x_i|$$

- **Chebyshev distance**

$$d_\infty^n(.,.) : \mathbb{R}^n \times \mathbb{R}^n \to \mathbb{R}$$
$$(x, y) \mapsto \|x - y\|_\infty^n$$



- **Vector subtraction**

$$(-)_n : \mathbb{R}^n \times \mathbb{R}^n \to \mathbb{R}^n$$
$$(x, y) \mapsto x - y$$

- **Pointwise comparison**

$$(<)_n : \mathbb{R}^n \times \mathbb{R}^n \to \mathbb{Z}_2^n$$
$$\left((x_i)_{i=0,\ldots,n-1}, (y_i)_{i=0,\ldots,n-1}\right) \mapsto (x_i < y_i)_{i=0,\ldots,n-1}$$

- **Scalar product**

$$< .,. >_n : \mathbb{R}^n \times \mathbb{R}^n \to \mathbb{R}$$
$$\left((x_i)_{i=0,\ldots,n-1}, (y_i)_{i=0,\ldots,n-1}\right) \mapsto \sum_{i=0}^{n-1} x_i y_i$$

- **Monomial enumerator**

$$(x^i)_n : \mathbb{R} \to \mathbb{R}^{n+1}$$
$$x \mapsto (x^i)_{i=0,\ldots,n}$$

- **Polynomial evaluation**

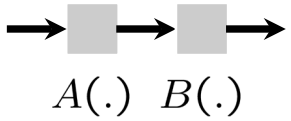
$$P[x, (a_0, \ldots, a_n)] : \mathbb{R} \to \mathbb{R}$$
$$x \mapsto a_0 x^n + a_1 x^{n-1} + \cdots + a_{n-1} x + a_n$$

Beyond the textbook: explicit vector length, infix operators as prefix operators

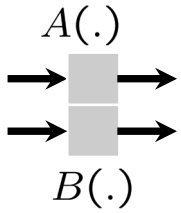
Operations and Operator Expressions

- **Operations (higher-order operators)**

$$\circ : (D \to S) \times (S \to R) \to (D \to R)$$
$$(A, B) \mapsto B \circ A$$

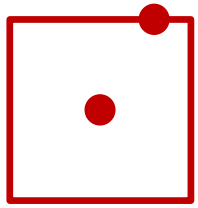


$$\times : (D \to R) \times (E \to S) \to (D \times E \to R \times S)$$
$$(A, B) \mapsto \Big((x, y) \mapsto \big(A(x), B(y)\big)\Big)$$



- **Operator expressions are operators**

$$\|.\|_\infty^n \circ (-)_n : \mathbb{R}^n \times \mathbb{R}^n \to \mathbb{R}$$
$$\big((x_i)_{i=0,\ldots,n-1}, (y_i)_{i=0,\ldots,n-1}\big) \mapsto \max_{i=0,\ldots,n-1} |x_i - y_i|$$

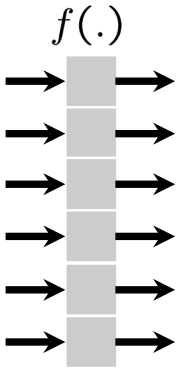


- **Short-hand notation: Infix notation**

$$A(.) - B(.) = \Big(x \mapsto A(x) - B(x)\Big)$$ can be expressed via $$(-)_n : \mathbb{R}^n \times \mathbb{R}^n \to \mathbb{R}^n$$
$$(x, y) \mapsto x - y$$

Basic OL Operators

- **Basic operators ≈ functional programming constructs**



map

$$\text{Pointwise}_{n,f_i} : \mathbb{R}^n \to \mathbb{R}^n$$
$$(x_i)_i \mapsto f_0(x_0) \oplus \cdots \oplus f_{n-1}(x_{n-1})$$

binop

$$\text{Atomic}_{f(.,.)} : \mathbb{R} \times \mathbb{R} \to \mathbb{R}$$
$$(x, y) \mapsto f(x, y)$$

map + zip

$$\text{Pointwise}_{n\times n,f_i} : \mathbb{R}^n \times \mathbb{R}^n \to \mathbb{R}^n$$
$$\big((x_i)_i, (y_i)_i\big) \mapsto f_0(x_0, y_0) \oplus \cdots \oplus f_{n-1}(x_{n-1}, y_{n-1})$$

fold

$$\text{Reduction}_{n,f_i} : \mathbb{R}^n \to \mathbb{R}$$
$$(x_i)_i \mapsto f_{n-1}(x_{n-1}, f_{n-2}(x_{n-2}, f_{n-3}(\dots f_0(x_0, \text{id}())\dots)$$

unfold

$$\text{Induction}_{n,f_i} : \mathbb{R} \to \mathbb{R}^{n+1}$$
$$x \mapsto (f_n(x, f_{n-1}(\dots)\dots), \dots, f_2(x, f_1(x, \text{id})), f_1(x, \text{id}), \text{id}())$$

- **Safety distance as (optimized) operator expression**

$$\text{SafeDist}_{V,A,b,\varepsilon} = \text{Atomic}_{(x,y)\mapsto x<y} \circ \Big(\big(\text{Reduction}_{3,(x,y)\mapsto x+y} \circ \text{Pointwise}_{3,x\mapsto a_i x} \circ \text{Induction}_{3,(a,b)\mapsto ab,1} \big) \times \big(\text{Reduction}_{2,(x,y)\mapsto \max(|x|,|y|)} \circ \text{Pointwise}_{2\times 2,(x,y)\mapsto x-y} \big) \Big)$$

Breaking Down Operators into Expressions

- **Application specific:** Safety Distance as Rewrite Rule

$$\text{SafeDist}_{V,A,b,\varepsilon}(.,.,.) \rightarrow \left(P[x,(a_0,a_1,a_2)](.) < d^2_\infty(.,.)\right)(.,.,.)$$

with $\quad a_0{=}\frac{1}{2b},\ a_1{=}\frac{V}{b}{+}\varepsilon\left(\frac{A}{b}{+}1\right),\ a_2{=}\left(\frac{A}{b}{+}1\right)\left(\frac{A}{2}\varepsilon^2{+}\varepsilon V\right)$

Problem specification: hand-developed or automatically produced

- **One-time effort:** mathematical library

$$d^n_\infty(.,.) \rightarrow \|.\|^n_\infty \circ (-)_n$$

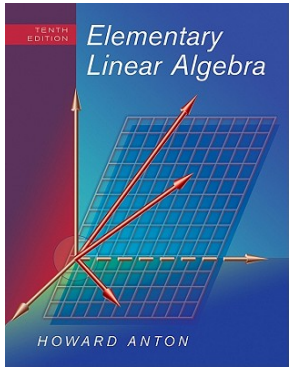
$$(\diamond)_n \rightarrow \text{Pointwise}_{n\times n,(a,b)\mapsto a\diamond b}, \quad \diamond \in \{+, -\cdot, \wedge, \vee, \dots\}$$

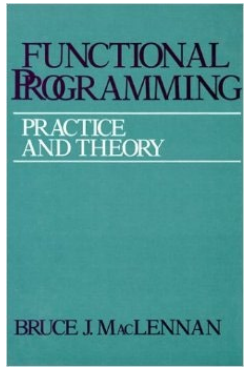
$$\|.\|^n_\infty \rightarrow \text{Reduction}_{n,(a,b)\mapsto \max(|a|,|b|)}$$

$$< .,. >_n \rightarrow \text{Reduction}_{n,(a,b)\mapsto a+b} \circ \text{Pointwise}_{n\times n,(a,b)\mapsto ab}$$

$$P[x,(a_0,\dots,a_n)] \rightarrow < (a_0,\dots,a_n),. > \circ (x^i)_n$$

$$(x^i)_n \rightarrow \text{Induction}_{n,(a,b)\mapsto ab,1}$$





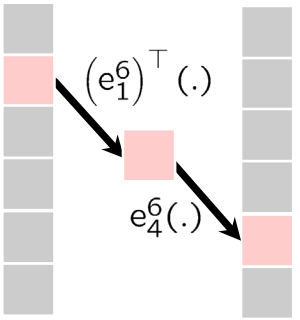
Library of well-known identities expressed in OL

Σ–OL: Low-Level Operator Language

- **Selection and embedding operator:** *gather and scatter*

$$(\mathsf{e}_i^n)^\top(.) : \mathbb{R}^n \to \mathbb{R}^1$$
$$(x_i)_{i=0,\ldots,n-1} \mapsto x_i$$

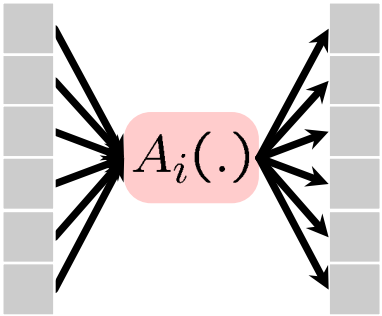
$$\mathsf{e}_i^n(.) : \mathbb{R}^1 \to \mathbb{R}^n$$
$$(x) \mapsto (0, \ldots, 0, \underbrace{x}_{i^{\text{th}}}, 0, \ldots, 0)$$



- **Iterative operations:** *loop*

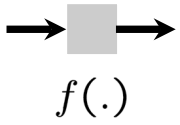
$$\bigsqcup_{i=0}^{n-1} : (D \to R)^n \to (D \to R)$$
$$A_i \mapsto (x \mapsto A_0(x) \sqcup \cdots \sqcup A_{n-1}(x))$$

with $\sqcup \in \{\sum, \vee, \wedge, \prod, \min, \max, \ldots\}$



- **Atomic operators:** *nonlinear scalar functions*

$$\text{Atomic}_f : \mathbb{R}^1 \to \mathbb{R}^1$$
$$(x) \mapsto (f(x))$$



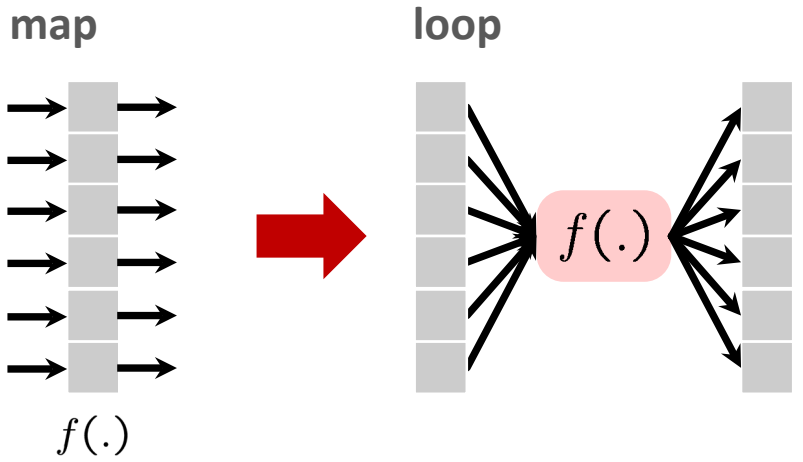
Σ-OL operator expressions = array-based programs with for loops

Rule-Based Translation and Optimization

- **Translating Basic OL into Σ-OL**

$$\text{Pointwise}_{n,f_i} \rightarrow \sum_{i=0}^{n-1} \left(\text{e}_i^n \circ \text{Atomic}_{f_i} \circ (\text{e}_i^n)^\top \right)$$

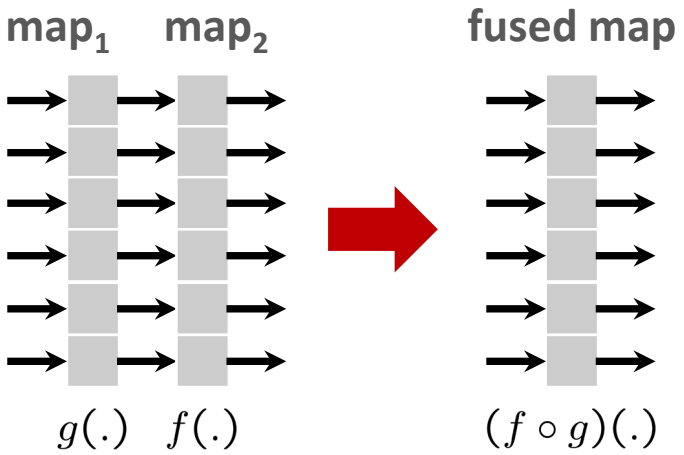
$$\text{Reduction}_{n,(a,b)\mapsto a+b} \rightarrow \sum_{i=0}^{n-1} (\text{e}_i^n)^\top$$



- **Optimizing Basic OL/Σ-OL**

$$\text{Pointwise}_{n,f_i} \circ \text{Pointwise}_{n,g_i} \rightarrow \text{Pointwise}_{n,f_i \circ g_i}$$

$$\text{Pointwise}_{n,f_i} \circ \text{e}_n^j \rightarrow \text{e}_n^j \circ \text{Pointwise}_{1,f_j}$$



Captures program optimizations that are traditionally hard to do

Last Step: Abstract Code

Code objects

- Values and types
- Arithmetic operations
- Logic operations
- Constants, arrays and scalar variables
- Assignments and control flow

Properties: at the same time

- Program = (abstract syntax) tree
- Represents program in restricted C
- OL operator over real numbers and machine numbers (floating-point)
- Pure functional interpretation
- Represents lambda expression

```
# Dynamic Window Monitor

let(
  i3 := var("i3", TInt), i5 := var("i5", TInt),
  w2 := var("w2", TBool), w1 := var("w1", T_Real(64)),
  s8 := var("s8", T_Real(64)), s7 := var("s7", T_Real(64)),
  s6 := var("s6", T_Real(64)), s5 := var("s5", T_Real(64)),
  s4 := var("s4", T_Real(64)), s1 := var("s1", T_Real(64)),
  q4 := var("q4", T_Real(64)), q3 := var("q3", T_Real(64)),
  D := var("D", TPtr(T_Real(64)).aligned([16, 0])),
  X := var("X", TPtr(T_Real(64)).aligned([16, 0])),

  func(TInt, "dwmonitor", [ X, D ],
     decl([q3, q4, s1, s4, s5, s6, s7, s8, w1, w2],
        chain(
           assign(s5, V(0.0)),
           assign(s8, nth(X, V(0))),
           assign(s7, V(1.0)),
           loop(i5, [0..2],
              chain(
                 assign(s4, mul(s7, nth(D, i5))),
                 assign(s5, add(s5, s4)),
                 assign(s7, mul(s7, s8))
              )
           ),
           assign(s1, V(0.0)),
           loop(i3, [0..1],
              chain(
                 assign(q3, nth(X, add(i3, V(1)))),
                 assign(q4, nth(X, add(V(3), i3))),
                 assign(w1, sub(q3, q4)),
                 assign(s6, cond(geq(w1, V(0)), w1, neg(w1))),
                 assign(s1, cond(geq(s1, s6), s1, s6))
              )
           ),
           assign(w2, geq(s1, s5)),
           creturn(w2)
        )
     )
  )
)
```

Translating Σ–OL to Abstract Code

Compilation rules: recursive descent

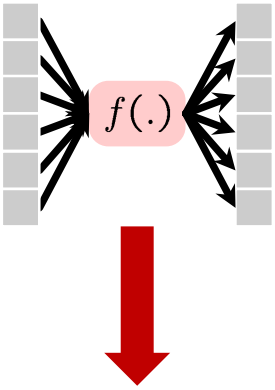
$$\text{Code}\left(y = (A \circ B)(x)\right) \rightarrow \left\{\texttt{decl}(t), \text{Code}\left(t = B(x)\right), \text{Code}\left(y = A(t)\right)\right\}$$

$$\text{Code}\left(y = \left(\sum_{i=0}^{n-1} A_i\right)(x)\right) \rightarrow \left\{y := \vec{0}, \texttt{for}(i = 0..n-1)\ \text{Code}\left(y{+}= A_i(x)\right)\right\}$$

$$\text{Code}\left(y = (\mathsf{e}_i^n)^\top(x)\right) \rightarrow y[0] := x[i]$$

$$\text{Code}\left(y = \mathsf{e}_i^n(x)\right) \rightarrow \left\{y = \vec{0}, y[i] := x[0]\right\}$$

$$\text{Code}\left(y = \text{Atomic}_f(x)\right) \rightarrow y[0] := f(x[i])$$




```
chain(
  assign(Y, V(0.0),
  loop(i1, [0..5],
    assign(nth(y, i1),
      f(nth(X, i1))
  )
)
```

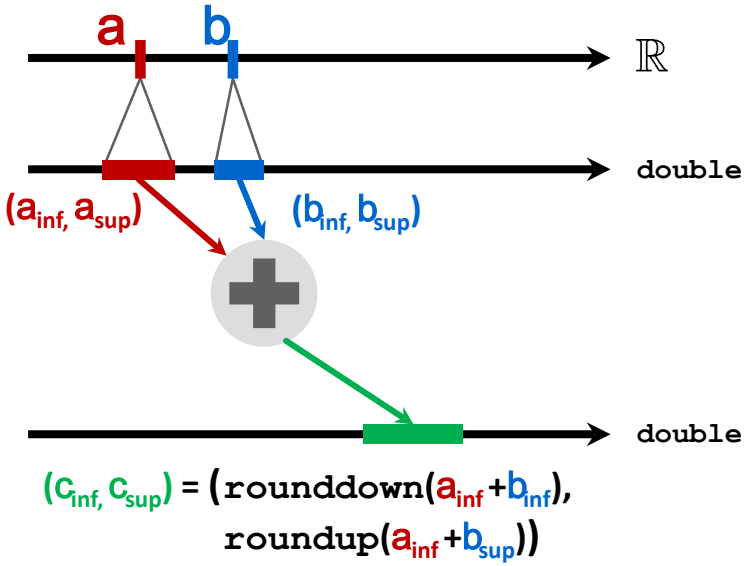
Cleanup rules: term rewriting

```
chain(a, chain(b))    → chain([a, b])
decl(D, decl(E, c))   → decl([D, E], c)
loop(i, decl(D, c))   → decl(D, loop(i, c))
chain(a, decl(D, b)) → decl(D, chain([a, b]))
```

Rule-based code generation and backend compilation

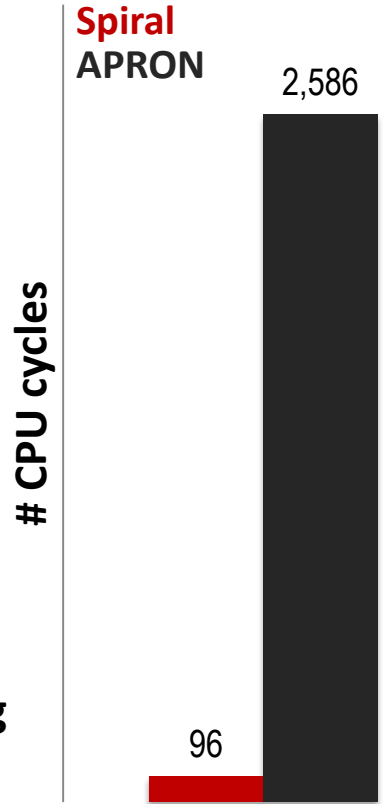
Sound Floating-Point Arithmetic

Interval Arithmetic

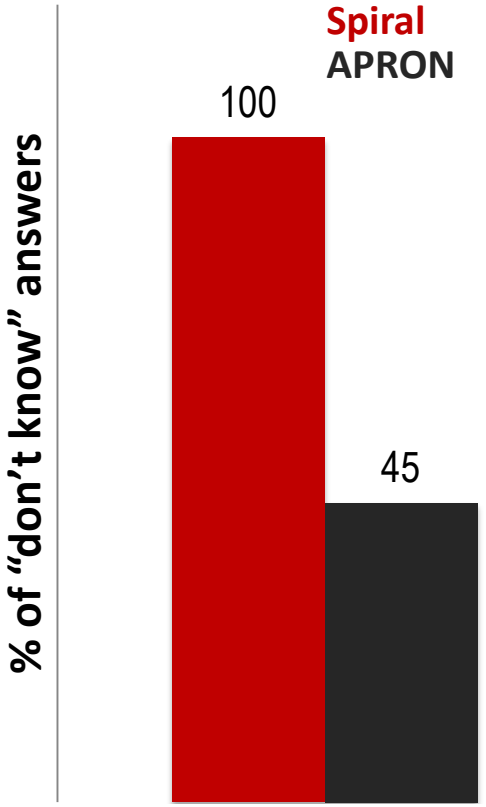


- **Standard implementation very slow**
  rounding mode change, IEEE754 denormals
- **Efficient implementation is challenging**
  lots of effort: 10x overhead over scalar
- **Application has headroom**
  only `float` required
- **Numerically ok for application**
  no iterations, ill-conditioning etc.

Performance



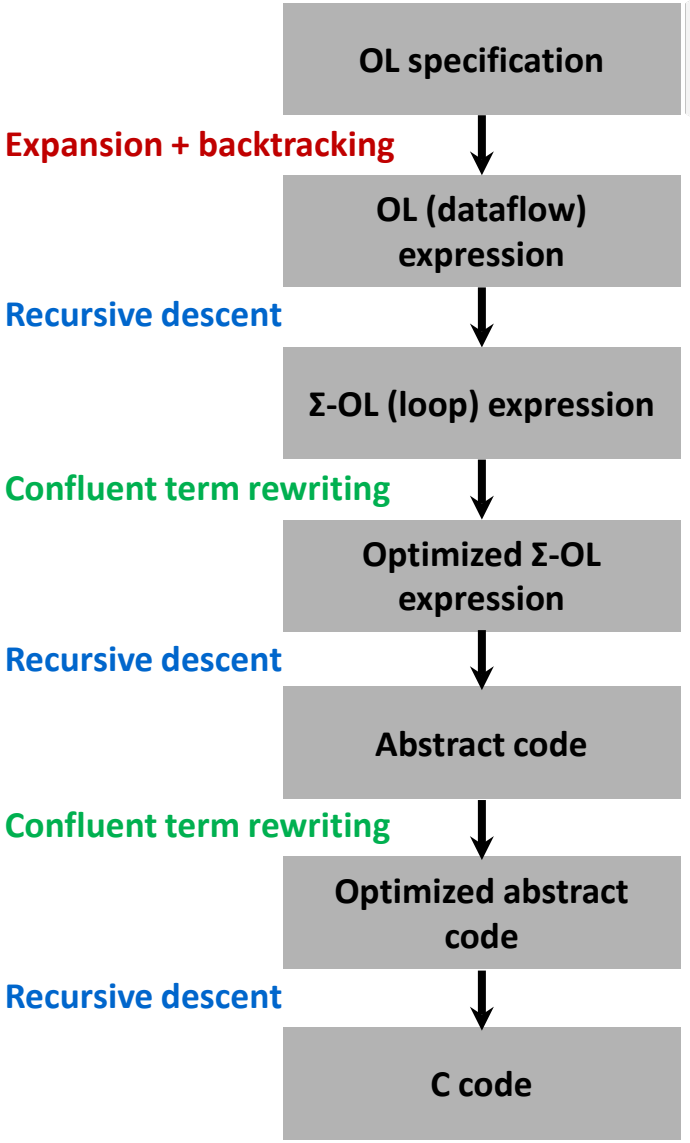
Precision at boundary



a < b ? for
$a - b \approx 10^{-15}$

SandyBridge CPU, Intel C Compiler,
SPIRAL vs. APRON Interval Arithmetic Library

Putting it Together: One Big Rule System



Mathematical specification

$$\text{SafeDist}_{V,A,b,\varepsilon}(.,.,.) \rightarrow \Big(P[x,(a_0,a_1,a_2)](.) < d^2_\infty(.,.)\Big)(.,.,.)$$

with $a_0 = \frac{1}{2b},\ a_1 = \frac{V}{b} + \varepsilon\left(\frac{A}{b}+1\right),\ a_2 = \left(\frac{A}{b}+1\right)\left(\frac{A}{2}\varepsilon^2 + \varepsilon V\right)$

Final code

```
int dwmonitor(float  *X, double  *D) {
    __m128d u1, u2, u3, u4, u5, u6, u7, u8 , x1, x10, x13, x14, x17
    int w1;
    unsigned _xm = _mm_getcsr();
    _mm_setcsr(_xm & 0xffff0000 | 0x0000dfc0);
    u5 = _mm_set1_pd(0.0);
    u2 = _mm_cvtps_pd(_mm_addsub_ps(_mm_set1_ps(FLT_MIN), _mm_set1_
    u1 = _mm_set_pd(1.0, (-1.0));
    for(int i5 = 0; i5 <= 2; i5++) {
        x6 = _mm_addsub_pd(_mm_set1_pd((DBL_MIN + DBL_MIN)), _mm_lo
        x1 = _mm_addsub_pd(_mm_set1_pd(0.0), u1);
        x2 = _mm_mul_pd(x1, x6);
        x3 = _mm_mul_pd(_mm_shuffle_pd(x1, x1, _MM_SHUFFLE2(0, 1)),
        x4 = _mm_sub_pd(_mm_set1_pd(0.0), _mm_min_pd(x3, x2));
        u3 = _mm_add_pd(_mm_max_pd(_mm_shuffle_pd(x4, x4, _MM_SHUFF
```

Final Synthesized C Code

```
int dwmonitor(float  *X, double  *D) {
    __m128d u1, u2, u3, u4, u5, u6, u7, u8 , x1, x10, x13, x14, x17, x18, x19, x2, x3, x4, x6, x7, x8, x9;
    int w1;
    unsigned _xm = _mm_getcsr();
    _mm_setcsr(_xm & 0xffff0000 | 0x0000dfc0);
    u5 = _mm_set1_pd(0.0);
    u2 = _mm_cvtps_pd(_mm_addsub_ps(_mm_set1_ps(FLT_MIN), _mm_set1_ps(X[0])));
    u1 = _mm_set_pd(1.0, (-1.0));
    for(int i5 = 0; i5 <= 2; i5++) {
        x6 = _mm_addsub_pd(_mm_set1_pd((DBL_MIN + DBL_MIN)), _mm_loaddup_pd(&(D[i5])));
        x1 = _mm_addsub_pd(_mm_set1_pd(0.0), u1);
        x2 = _mm_mul_pd(x1, x6);
        x3 = _mm_mul_pd(_mm_shuffle_pd(x1, x1, _MM_SHUFFLE2(0, 1)), x6);
    }
    u6
    for
        u8 = _mm_cvtps_pd(_mm_addsub_ps(_mm_set1_ps(FLT_MIN), _mm_set1_ps(X[(i3 + 1)])));
        u7 = _mm_cvtps_pd(_mm_addsub_ps(_mm_set1_ps(FLT_MIN), _mm_set1_ps(X[(3 + i3)])));
        x14 = _mm_add_pd(u8, _mm_shuffle_pd(u7, u7, _MM_SHUFFLE2(0, 1)));
        x13 = _mm_shuffle_pd(x14, x14, _MM_SHUFFLE2(0, 1));
        u4 = _mm_shuffle_pd(_mm_min_pd(x14, x13), _mm_max_pd(x14, x13), _MM_SHUFFLE2(1, 0));
        u6 = _mm_shuffle_pd(_mm_min_pd(u6, u4), _mm_max_pd(u6, u4), _MM_SHUFFLE2(1, 0));
    }
    x17 = _mm_addsub_pd(_mm_set1_pd(0.0), u6);
    x18 = _mm_addsub_pd(_mm_set1_pd(0.0), u5);
    x19 = _mm_cmpge_pd(x17, _mm_shuffle_pd(x18, x18, _MM_SHUFFLE2(0, 1)));
    w1 = (_mm_testc_si128(_mm_castpd_si128(x19), _mm_set_epi32(0xffffffff, 0xffffffff, 0xffffffff, 0xffffffff)) -
          (_mm_testnzc_si128(_mm_castpd_si128(x19), _mm_set_epi32(0xffffffff, 0xffffffff, 0xffffffff, 0xffffffff))));
    __asm nop;
    if (_mm_getcsr() & 0x0d) {
        _mm_setcsr(_xm);
        return -1;
    }
    _mm_setcsr(_xm);
    return w1;
}
```

$$\mathsf{SafeDist}_{V,A,b,\varepsilon} = \mathsf{Atomic}_{(x,y)\mapsto x<y} \circ \Big(\big(\mathsf{Reduction}_{3,(x,y)\mapsto x+y} \circ \mathsf{Pointwise}_{3,x\mapsto a_i x} \circ \mathsf{Induction}_{3,(a,b)\mapsto ab,1} \big) \times \big(\mathsf{Reduction}_{2,(x,y)\mapsto \max(|x|,|y|)} \circ \mathsf{Pointwise}_{2\times 2,(x,y)\mapsto x-y} \big) \Big)$$

Formal Proof: Specification = Code

Mathematical equivalency

- **Code over *reals* = specification**
  code as operator =
  specification as operator
- **Rules are mathematical identities**
  all objects have mathematical semantics
- **Synthesis is semantics preserving**
  chain of semantics-preserving rules

Floating-point: interval arithmetic

- **Numerical results are sound**
  true answer guaranteed in result interval
- **Logical answers are sound**
  conservative: true/false/unknown
- **Floating-point uncertainty**
  operators over random variables

```
<800 lines>
-------------------------------------------------------------------
RULE: eT_Pointwise
-------------------------------------------------------------------
old expression
-------------------------------------------------------------------
eT(3, i17) o
PointWise(3, Lambda([ r16, i14 ], mul(r16, nth(D, i14)))) o
Induction(3, Lambda([ r9, r10 ], mul(r9, r10)), V(1.0)) o
eT(5, 0)
-------------------------------------------------------------------
new expression
-------------------------------------------------------------------
PointWise(1, Lambda([ r16, i19 ], mul(r16, nth(D, i17)))) o
eT(3, i17) o
Induction(3, Lambda([ r9, r10 ], mul(r9, r10)), V(1.0)) o
eT(5, 0)
-------------------------------------------------------------------


-------------------------------------------------------------------
RULE: eT_Induction
-------------------------------------------------------------------
old expression
-------------------------------------------------------------------
PointWise(1, Lambda([ r16, i19 ], mul(r16, nth(D, i17)))) o
eT(3, i17) o
Induction(3, Lambda([ r9, r10 ], mul(r9, r10)), V(1.0)) o
eT(5, 0)
-------------------------------------------------------------------
new expression
-------------------------------------------------------------------
PointWise(1, Lambda([ r16, i19 ], mul(r16, nth(D, i17)))) o
Inductor(3, i17, Lambda([ r9, r10 ], mul(r9, r10)), V(1.0)) o
eT(5, 0)
-------------------------------------------------------------------
<10500 lines>
```

Rewrite rule:

$$(e_n^j)^\top \circ \text{Pointwise}_{n,f_i} \rightarrow \text{Pointwise}_{1,f_j} \circ (e_n^j)^\top$$

Formal Approach for all Types of Parallelism

- **Multithreading** (Multicore)
- **Vector SIMD** (SSE, VMX/Altivec,…)
- **Message Passing** (Clusters, MPP)
- **Streaming/multibuffering** (Cell)
- **Graphics Processors** (GPUs)
- **Gate-level parallelism** (FPGA)
- **HW/SW partitioning** (CPU + FPGA)

$$\mathrm{I}_p \otimes_{\|} A_{\mu n}, \quad \mathbf{L}^{mn}_m \bar{\otimes} \mathrm{I}_\mu$$

$$A \hat{\otimes} \mathrm{I}_\nu \quad \underbrace{\mathbf{L}^{2\nu}_2}_{\text{isa}}, \quad \underbrace{\mathbf{L}^{2\nu}_\nu}_{\text{isa}}, \quad \underbrace{\mathbf{L}^{\nu^2}_\nu}_{\text{isa}}$$

$$\mathrm{I}_p \otimes_{\|} A_n, \quad \underbrace{\mathbf{L}^{p^2}_p \bar{\otimes} \mathrm{I}_{n/p^2}}_{\text{all-to-all}}$$

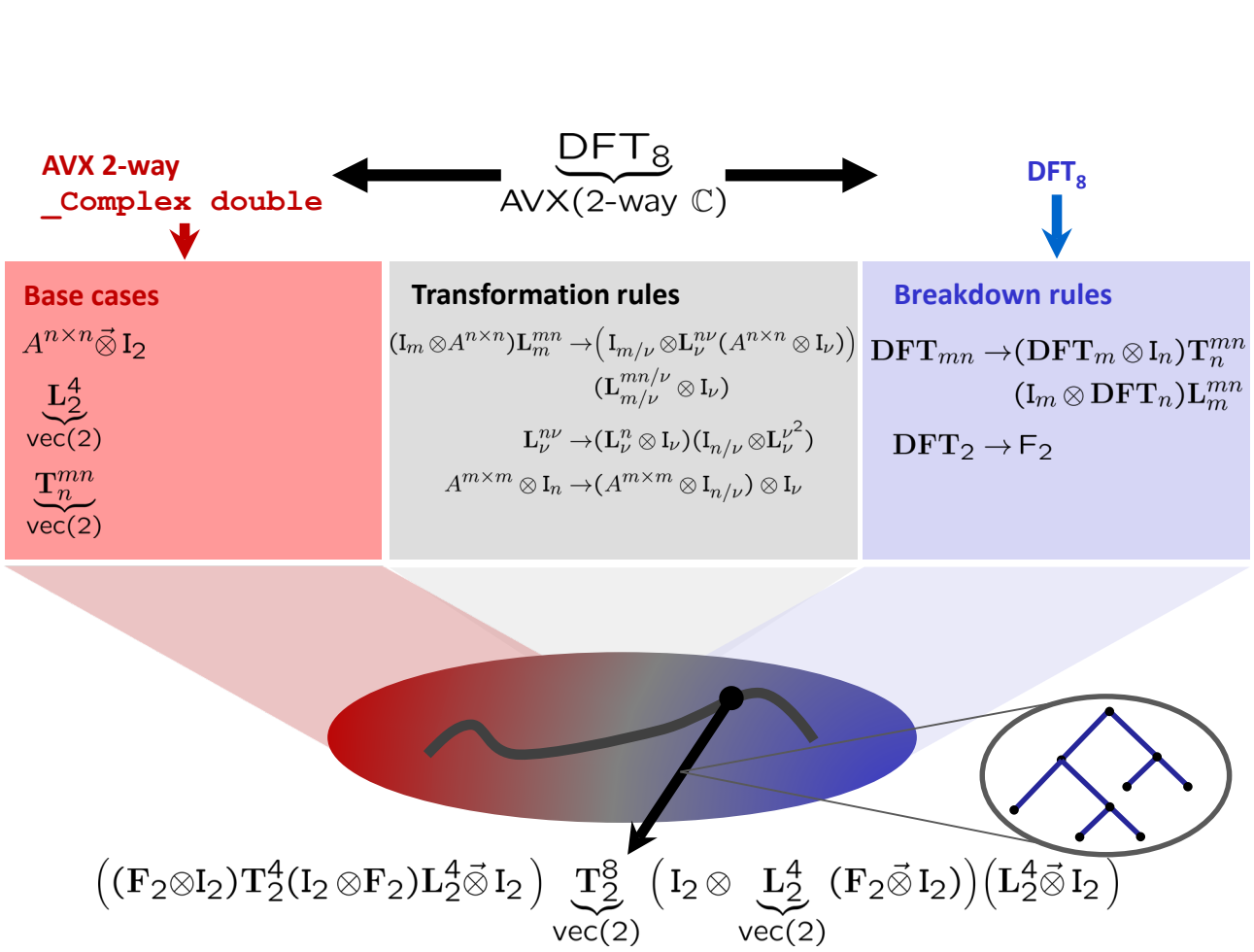
$$\mathrm{I}_n \otimes_2 A_{\mu n}, \quad \mathbf{L}^{mn}_m \bar{\otimes} \mathrm{I}_\mu$$

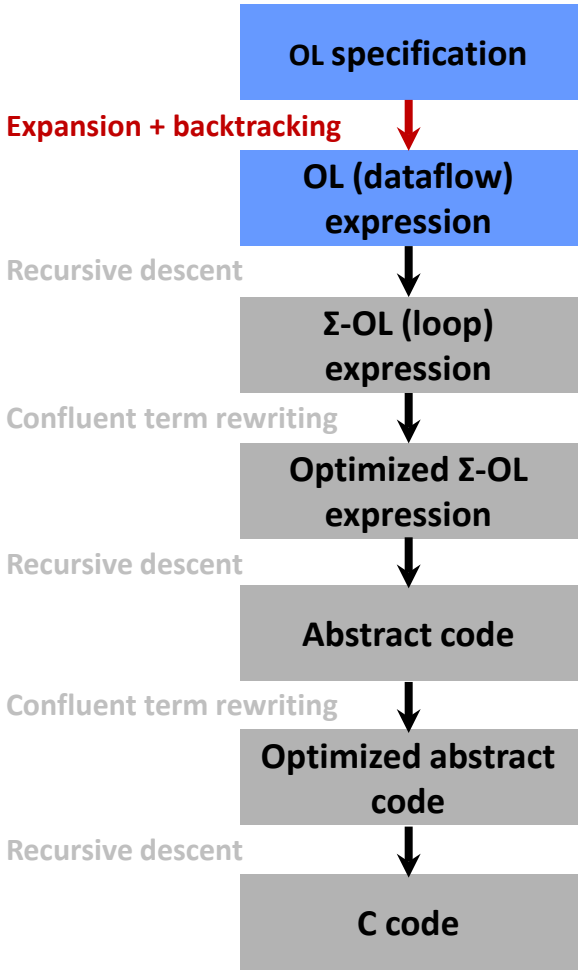
$$\prod_{i=0}^{n-1} A_i, \quad A_n \hat{\otimes} \mathrm{I}_w, \quad P_n \otimes Q_w$$

$$\prod_{i=0}^{n-1^{\mathsf{ir}}} A, \quad \mathrm{I}_s \tilde{\otimes} A, \quad \underbrace{\mathbf{L}^m_n}_{\text{bram}}$$

$$\underbrace{A_1}_{\text{fpga}}, \quad \underbrace{A_2}_{\text{fpga}}, \quad \underbrace{A_3}_{\text{fpga}}, \quad \underbrace{A_4}_{\text{fpga}}$$

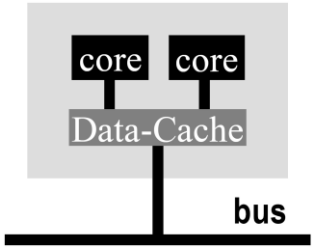
Autotuning in Constraint Solution Space





Modeling Hardware: Base Cases

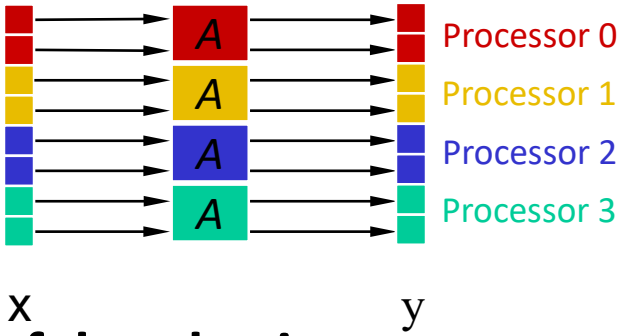
- **Hardware abstraction: shared cache with cache lines**



$$\underbrace{A_{k\times m\to n}}_{\mathrm{smp}(p,\mu)}$$

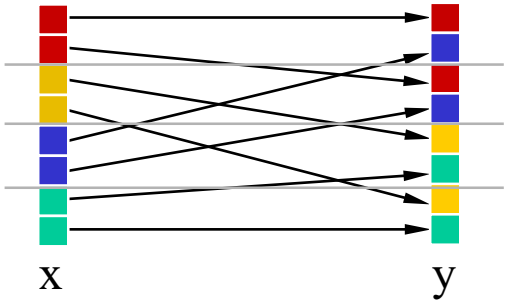
- **Tensor product: embarrassingly parallel operator**

$$y = \left(\mathrm{I}_p \otimes A\right)(x)$$



- **Permutation: problematic; may produce false sharing**

$$y = \mathrm{L}^{8}_{4}(x)$$



Example Program Transformation Rule Set

$$\underbrace{AB}_{\mathsf{smp}(p,\mu)} \rightarrow \underbrace{A}_{\mathsf{smp}(p,\mu)}\ \underbrace{B}_{\mathsf{smp}(p,\mu)}$$

$$\underbrace{A_m \otimes \mathrm{I}_n}_{\mathsf{smp}(p,\mu)} \rightarrow \underbrace{\left(\mathrm{L}^{mp}_{m} \otimes \mathrm{I}_{n/p}\right)\left(\mathrm{I}_p \otimes (A_m \otimes \mathrm{I}_{n/p})\right)\left(\mathrm{L}^{mp}_{p} \otimes \mathrm{I}_{n/p}\right)}_{\mathsf{smp}(p,\mu)}$$

$$\underbrace{\mathrm{L}^{mn}_{m}}_{\mathsf{smp}(p,\mu)} \rightarrow \begin{cases} \underbrace{\left(\mathrm{I}_p \otimes \mathrm{L}^{mn/p}_{m/p}\right)}_{\mathsf{smp}(p,\mu)}\underbrace{\left(\mathrm{L}^{pn}_{p} \otimes \mathrm{I}_{m/p}\right)}_{\mathsf{smp}(p,\mu)} \\ \underbrace{\left(\mathrm{L}^{pm}_{m} \otimes \mathrm{I}_{n/p}\right)}_{\mathsf{smp}(p,\mu)}\underbrace{\left(\mathrm{I}_p \otimes \mathrm{L}^{mn/p}_{m}\right)}_{\mathsf{smp}(p,\mu)} \end{cases}$$

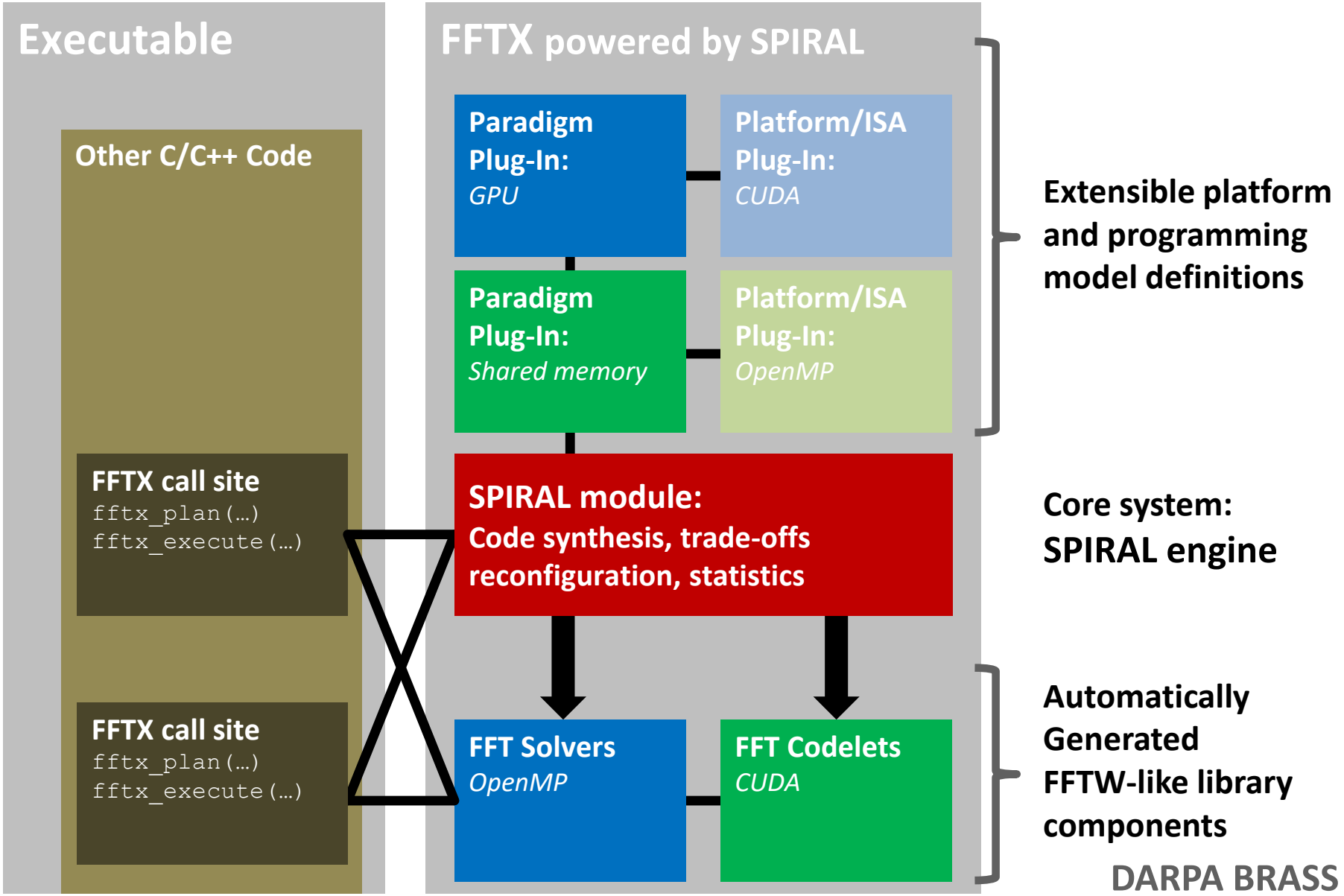
Recursive rules

$$\underbrace{\mathrm{I}_m \otimes A_n}_{\mathsf{smp}(p,\mu)} \rightarrow \mathrm{I}_p \otimes_{\|} \left(\mathrm{I}_{m/p} \otimes A_n\right)$$

$$\underbrace{(P \otimes \mathrm{I}_n)}_{\mathsf{smp}(p,\mu)} \rightarrow \left(P \otimes \mathrm{I}_{n/\mu}\right) \overline{\otimes}\, \mathrm{I}_\mu$$

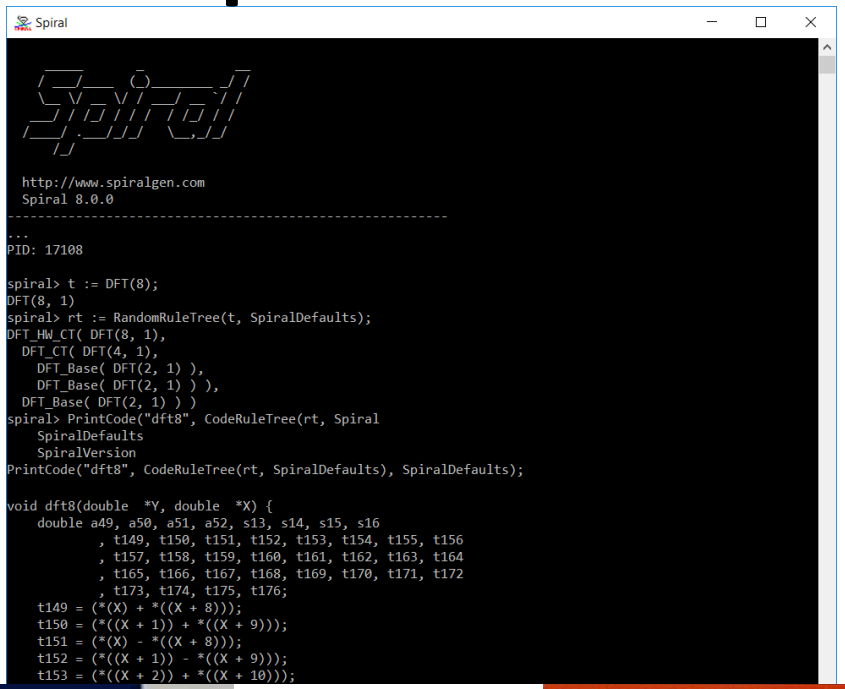
Base case rules

FFTX Backend: SPIRAL



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**



www.spiral.net

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*