Towards Interactive Program Optimization with Guaranteed Numerical Accuracy

Thomas Kœhler thok.eu

October 2025, IRMIA++ Seminar, Strasbourg

My Career

2018

2020

2022

2024

Oct

PhD
with Michel Steuwer and Phil Trinder — *Glasgow* — University of Glasgow

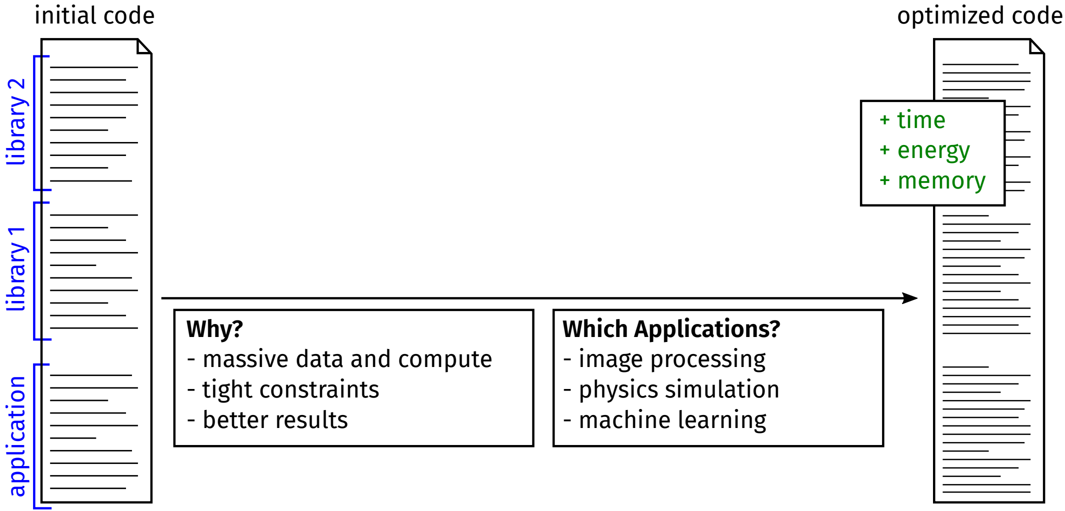
R&D Internship — *Edinburgh* — HUAWEI

PostDoc
with Arthur Charguéraud

CRCN Researcher

Inria CAMUS, Strasbourg — Inria, cnrs

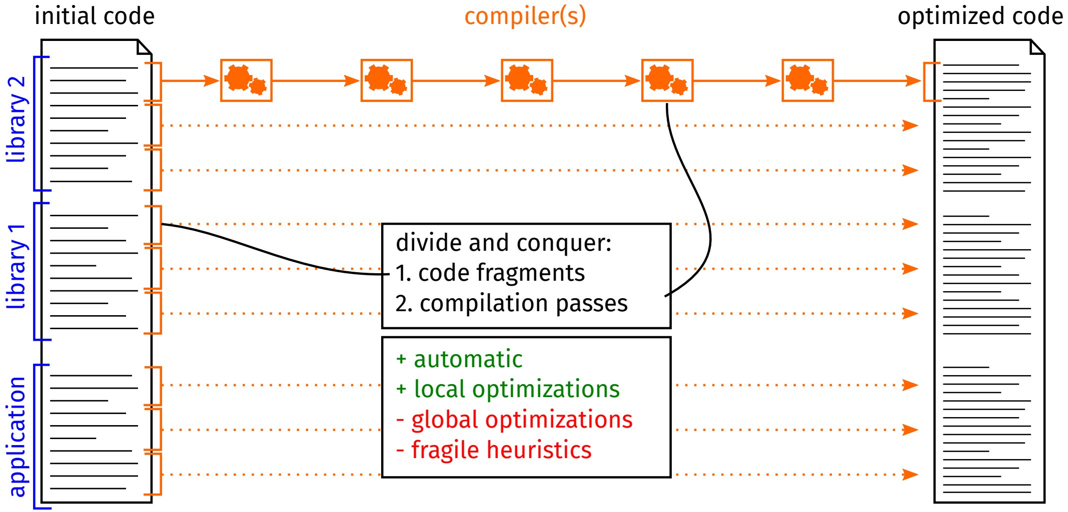
Optimizing Programs



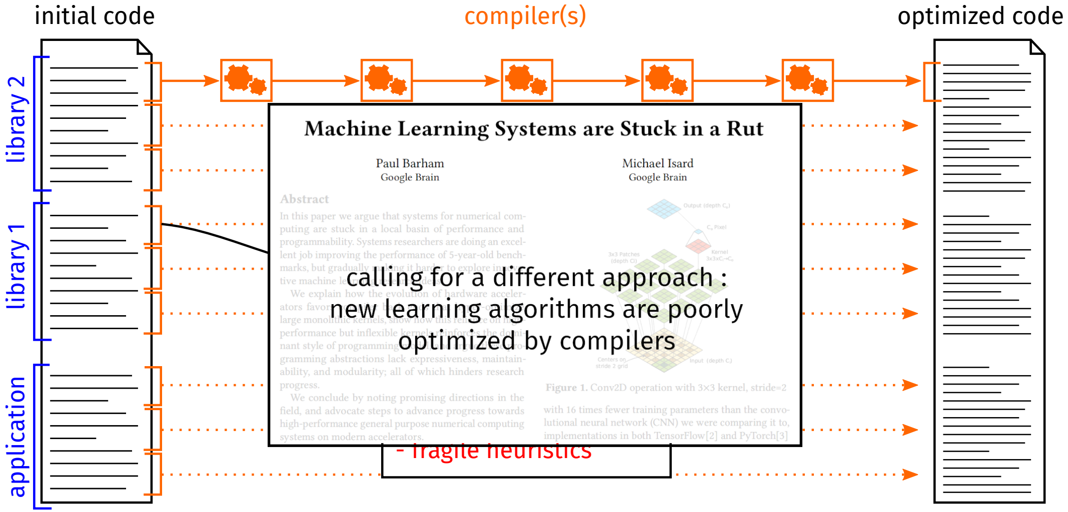
Why?
- massive data and compute
- tight constraints
- better results

Which Applications?
- image processing
- physics simulation
- machine learning

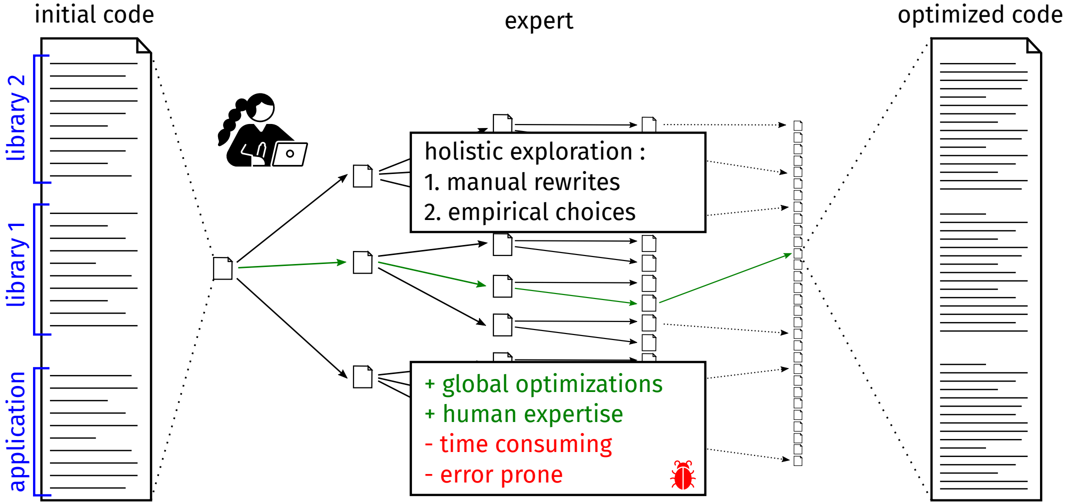
Automatic Program Optimization (GCC/Clang)



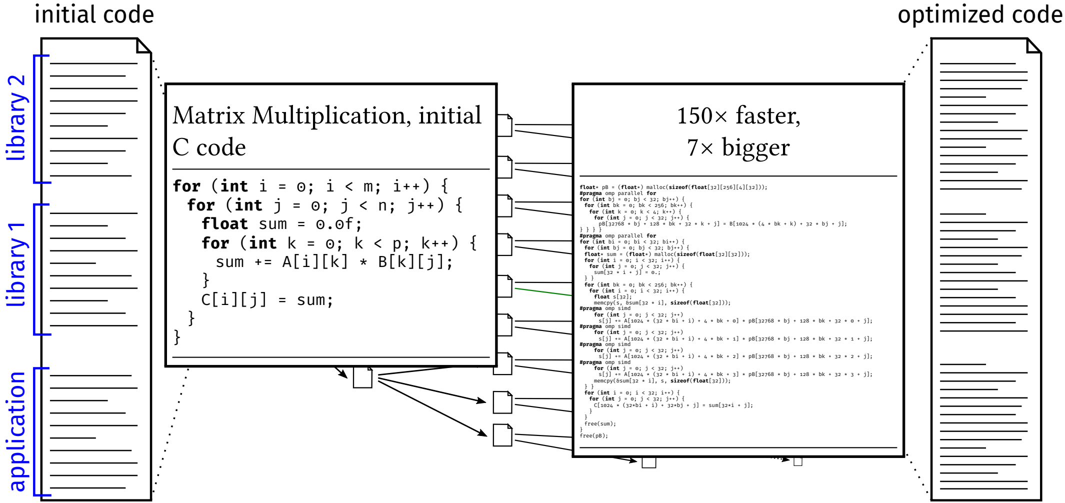
Automatic Program Optimization (GCC/Clang)



Manual Program Optimization (BLAS/MKL)



Manual Program Optimization (BLAS/MKL)

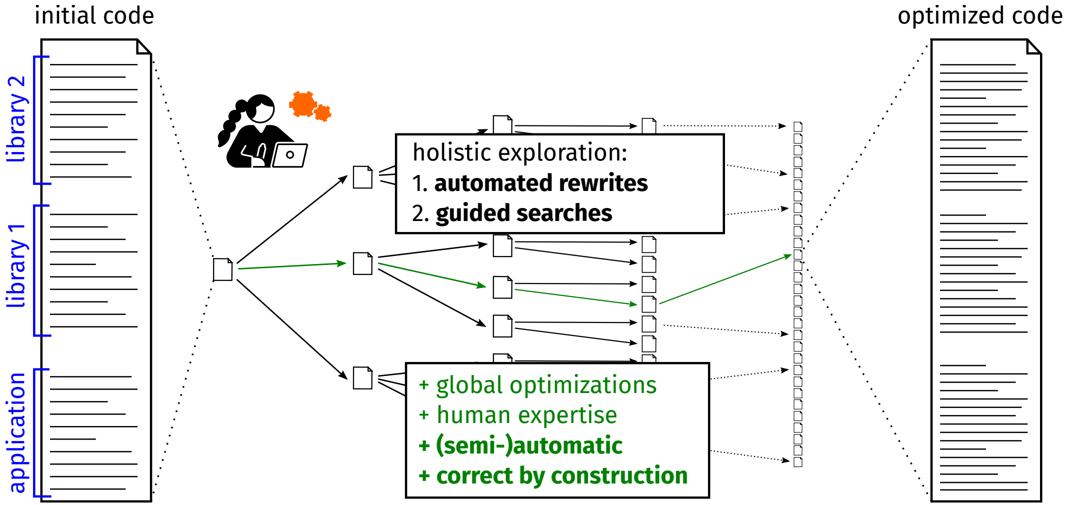


What Future for Program Optimization?

- compilers answer neither current nor future optimization needs
- algorithms and hardware architectures evolve faster than compilers
- falling back to manual optimization **slows down progress**

PLDI 2024 Keynote: The Future of Fast Code: Giving Hardware What It Wants

Towards Interactive Optimization



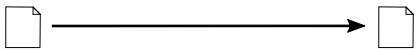
User-Guided Program Optimization (Halide)

```
Func blur_3x3(Func input) {
  Func blur_x, blur_y;
  Var x, y, xi, yi;

  // The algorithm - no storage or order
  blur_x(x, y) = (input(x-1, y) + input(x, y) + input(x+1, y))/3;
  blur_y(x, y) = (blur_x(x, y-1) + blur_x(x, y) + blur_x(x, y+1))/3;

  // The schedule - defines order, locality; implies storage
  blur_y.tile(x, y, xi, yi, 256, 32)
        .vectorize(xi, 8).parallel(y);
  blur_x.compute_at(blur_y, x).vectorize(x, 8);

  return blur_y;
}
```

- Industry adoption with, e.g. code optimized in Adobe Photoshop
- Academic community with first "User-Schedulable Languages" workshop in 2025

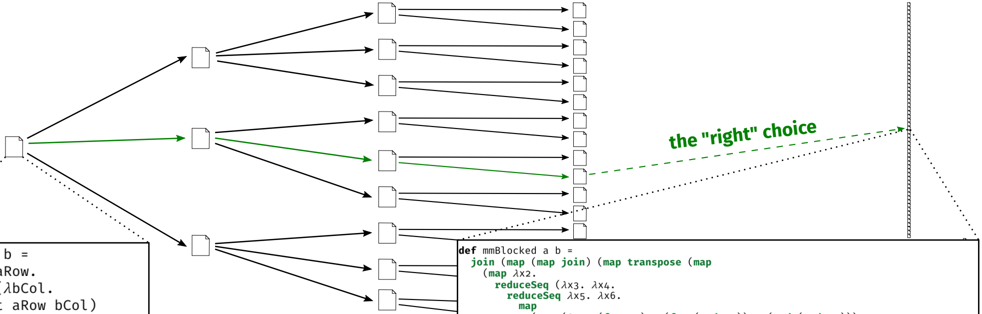
PhD: Rewriting Functional Programs (Rise)

rewrite rules



```
(map f) . (map g) = map (f . g)
```

combinatorial rewriting space, correct and extensible




```
def mm a b =
  map (λaRow.
    map (λbCol.
      dot aRow bCol)
      (transpose b)) a

def dot xs ys =
  reduce + 0
    (map (λ(x, y). x × y)
      (zip xs ys))
```

```
def mmBlocked a b =
  join (map (map join) (map transpose (map
    (map λx2.
      reduceSeq (λx3. λx4.
        reduceSeq λx5. λx6.
          map
            (map (λx7. (fst x7) + (fst (snd x7)) × (snd (snd x7)))
              (map (λx7. zip (fst x7) (snd x7)) (zip x5 x6)))
          (transpose (map transpose
            (snd (unzip (map unzip map (λx5. zip (fst x5) (snd x5)) (zip x3 x4)))))))
        (generate (λx3. generate (λx4. 0)))
        transpose (map transpose x2))
    (map (map (map (map (split 4))))
      (map transpose
        (map (map (λx2. map (map (zip x2) (split 32 (transpose b))))) split 32 a))))))
```

PostDoc: Transforming Imperative Programs (OptiTrust)

```
void mm(float* c, float* a, float* b, int m, int n, int p) {
  __requires("A: int * int → float, B: int * int → float");
  __reads("a ⇝ Matrix2(m, p, A), b ⇝ Matrix2(p, n, B)");
  __writes("c ⇝ Matrix2(m, n, matmul(A, B, p))");

  for (int i = 0; i < m; i++) { // [...]
    for (int j = 0; j < n; j++) {
      __xwrites("&c[i][j]  ⤳ matmul(A, B, p)(i, j)");

      float s = 0.f; // [...]
      for (int k = 0; k < p; k++) {
        __spreserves("&s  ⤳ sum(k, fun k0 → A(i, k0) *. B(k0, j))");
        s += a[i][k] * b[k][j];
        // [...]
      }

      c[i][j] = s;
} } }
```

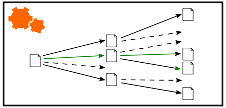
PostDoc: Transforming Imperative Programs (OptiTrust)

```
Function.inline_def [cFunDef "mm"];
let tile (id, tile_size) = Loop.tile (int tile_size)
  ~index:("b" ^ id) ~bound:TileDivides [cFor id] in
List.iter tile [("i", 32); ("j", 32); ("k", 4)];
Loop.reorder_at
  ~order:["bi"; "bj"; "bk"; "i"; "k"; "j"] [cPlusEq ()];
Loop.hoist_expr ~dest:[tBefore; cFor "bi"] "pB"
  ~indep:["bi"; "i"] [cArrayRead "B"];
Matrix.stack_copy ~var:"sum" ~copy_var:"s"
  ~copy_dims:1 [cFor ~body:[cPlusEq ()] "k"];
Omp.simd [cFor ~body:[cPlusEq ()] "j"];
Omp.parallel_for [cFunBody "mm1024"; cStrict; cFor ""];
Loop.unroll [cFor ~body:[cPlusEq ()] "k"];
```

Vision: Interactive Optimization Across the Stack

image processing
machine learning
physics simulation
...

Goal:
build the first
Optimization Assistant



that adapts to domain and hardware evolution, and avoids falling back to manual optimization

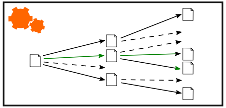
CPU
GPU
CGRA, FPGA
...

Vision: Interactive Optimization Across the Stack

image processing

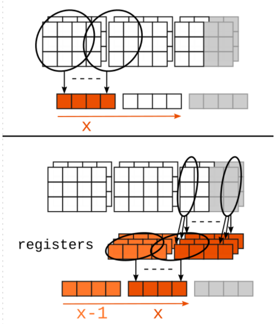
Harris: corner and edge detection

$$\begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix} = \begin{bmatrix} 1 \\ 2 \\ 1 \end{bmatrix} \begin{bmatrix} -1 & 0 & 1 \end{bmatrix}$$



6 expert optimizations
⟶ decomposed into **74 rules**
16x faster than OpenCV,
1.4x faster than Halide!
[CGO'21 *A*]

CPU



Vision: Interactive Optimization Across the Stack

GPU Gems 3

GPU Gems 3 is now available for free online!

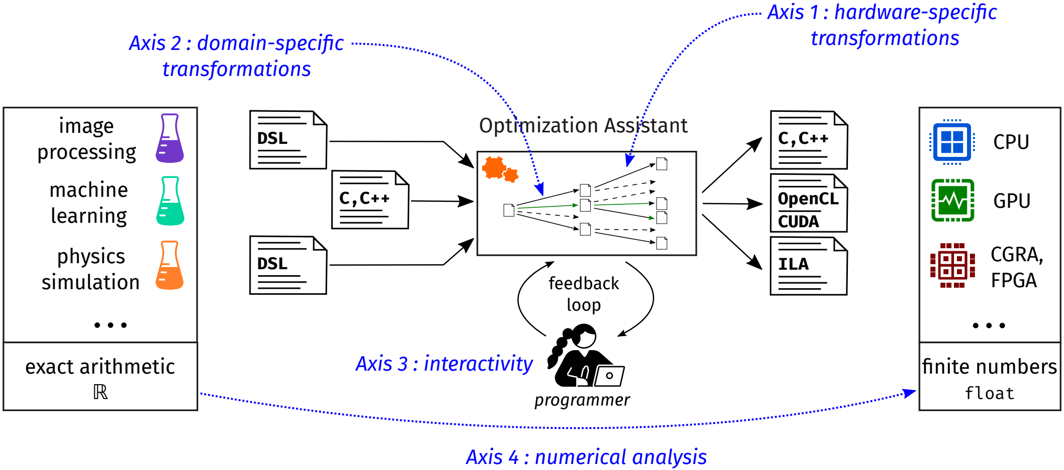
Chapter 40. Incremental Computation of the Gaussian

40.6 Conclusion

We have presented a very simple technique to evaluate the Gaussian at regularly spaced points. It is not an approximation, and it has the simplicity of polynomial forward differencing, so that only one vector instruction is needed for a coefficient update at each point on a GPU.

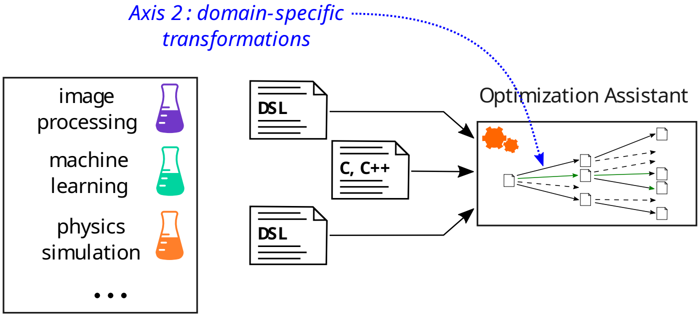
The method of *forward quotients* can accelerate the computation of any function that is the exponential of a polynomial. Modern computers can perform multiplications as fast as additions, so the technique is as powerful as forward differencing. This method opens up incremental computations to a new class of functions.

Vision: Interactive Optimization Across the Stack



Axis 2: Libraries of Composable Domain-Specific Transformations

Goal: create an optimization space that evolves with domains and their specialized languages *(DSLs)*



Plan:

- AI project: novel ML stack
- long term: mixing DSLs, libraries = functions + transformations + heuristics

Difficulties:

- breaking abstraction and stack boundaries
- facilitate defining custom transformations

Axis 2: Libraries of Composable Domain-Specific Transformations

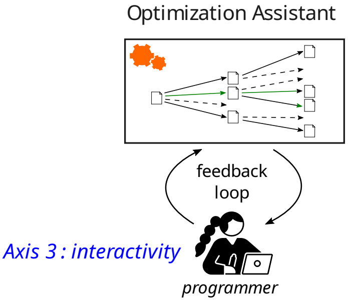
```
// library of (unoptimized) functions

decl dot (k: nat) (A B: k.real): real

// library of transformations

rule dot_comm (k: nat) (A B: k.real):
  dot A B = dot B A

rule dot_to_imperative (k: nat) (A B: k.real):
  imp (dot A B) = { reads a ~> A, b ~> B, writes c ~> dot A B
    var s = 0
    for i in range(k) {
      s += a[i] * b[i]
    }
    c = s
  }

// ... use optimization assistant to derive optimized code ...
```

Axis 3: Interactivity between Programmer and Assistant

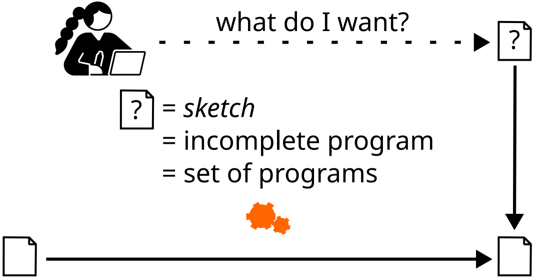
Goal: develop an interactive feedback loop allowing
to productively explore the optimization space

Plan:
- now: sketch-guided optimization of imperative code
  hybrid polyhedric algorithm
- later: optimization suggestions and vizualisations
  bottlenecks, hot code

Difficulty:
- exploring a complex and evolving optimization space



Sketch-Guided Program Optimization



Sketch-Guided Equality Saturation

Guided Equality Saturation [POPL'24 *A]**

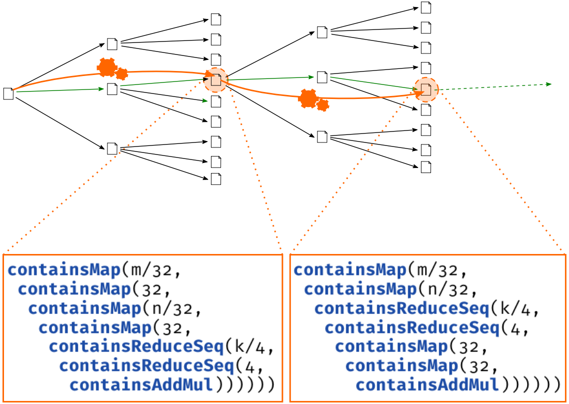
=

automatic rewrite search,
sharing equivalent subterms

+

specifying guides as program sketches

guided search :
582x faster, 116x less memory



Sketch-Guided Polyhedral Compilation

- internship and PhD of Valéran Maytié
- supervised together with Cédric Bastoul and Christophe Alias

Specification

```
    A[i], B[i] : { 1 ⩽ i ⩽ N }
    C[i] : { 0 ⩽ i < N }
    D[i] : { 1 ⩽ i ⩽ N } := A[i] + B[i]
out E[i] : { 0 ⩽ i < N } := C[i] + D[i+1]
```

Sketch

```
for i {
  D[i];
  E[i];
}
```

Output

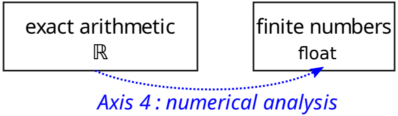
```
evar [ce, cd] { ce = cd - 1 }
for i in { 1 ⩽ i+cd ⩽ N } {
  D[i+cd] = A[i+cd] + B[i+cd];
  E[i+ce] = C[i+ce] + D[i+ce+1];
}
```

Half Time

Axis 4: Guaranteeing Numerical Accuracy during Optimization

Goal: guarantee numerical accuracy during optimization,
avoid errors and allow optimizations

error-tolerant trigonometric function = 4x faster
fixed point numbers for FPGA



Difficulty:

- most compilers do not perform numerical analyses
  'gcc': no optimisations over floats
  'gcc -ffast-math': treats floats as reals

Plan:

- now: functional language on arrays
- later: imperative language

International Collaboration:

- Eva Darulova, Uppsala University [TOPLAS'17, SAS'23]

Floating Point Analysis and Optimization Tools

FPTalks Community Tools

Numerics research tools and systems

Events   Education   Tools   Mailing List   About

The floating-point research community has developed many tools that authors of numerical software can use to test, analyze, or improve their code. They are listed here in alphabetical order and categorized into three general types: 1) dynamic tools, 2) static tools, 3) commercial tools, and 4) miscellaneous. You can find tutorials for some of these tools at fpanalysistools.org, and many of the reduced-precision tools were compared in a 2020 survey by authors from the Polytechnic University of Milan.

Floating Point Analysis and Optimization Tools

- tools to tune the precision of variables and operations (*mixed-precision tuning*)
- tools that rewrite arithmetic expressions
- **no tool that applies advanced compiler optimizations over loops and arrays**

Floating Point Analysis and Optimization Tools

Scaling up Roundoff Analysis of Functional Data Structure Programs

Anastasia Isychev[1] (✉) and Eva Darulova[2]

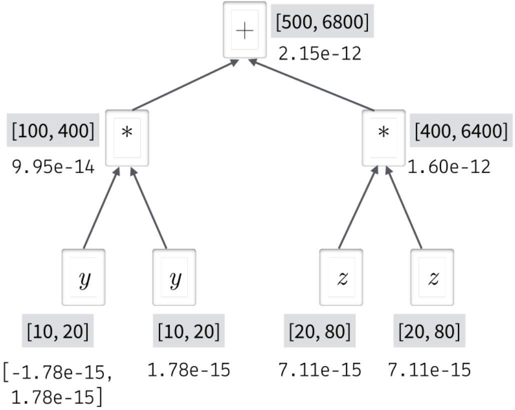
[SAS'23]

Ingredient 1: Data Flow Analysis

Dataflow-based

Fluctuat

with interval-like abstract domains



Ingredient 2: Abstraction of Floating Point Operations

Arithmetic operations: computed as if with real arithmetic and then **rounded**

- Different rounding modes: **to nearest** (default), to 0, to +/- infinity
- **Abstraction** for arithmetic operations and rounding to nearest:

float　　　　machine epsilon

$$\tilde{op} = op(1 + e) + d \quad \text{where } |e| \leq \epsilon, |d| \leq \delta$$

real　　　　subnormal numbers

- (special values (NaN, infinity) - ignore for this talk)

Novel Ingredient: Grouped Analysis of Array Elements

- functional DSL over real-valued **vectors** and **matrices**
- data structure (DS) **abstraction**

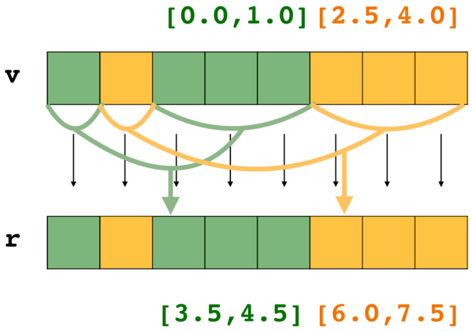
    group DS elements by range
- transfer functions

    analyze **once** per unique range

    reduce to **straight-line dataflow** analysis

    with **interval** arithmetic

```
val r = v.map(x => x + 3.5)
```

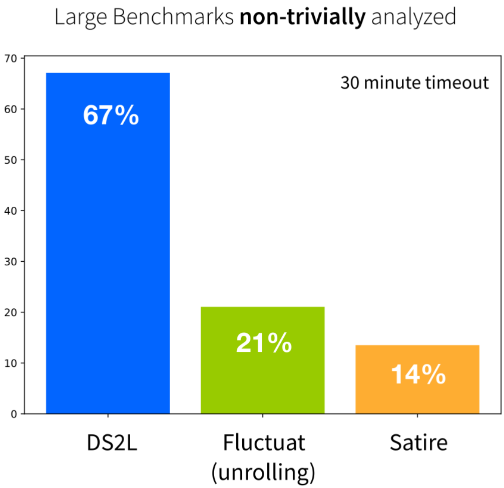


hiding a lot of details

Benchmarks

• statistical computations	`avg, stdDeviation, variance`
• digital filters	`roux1, goubault, harmonic, nonlin{1-3}`
• differential equations	`lorenz, pendulum`
• signal processing	`alphaBlending, fftvector, fftmatrix`
• stencil computations	`convolve2d_size3, sobel3, heat1d`
• neural networks	`lyapunov, controllerTora`

Scalability Results



- benchmark sizes: 10k - 708k iterations
- arithm. ops/iteration: 1 - 1k

- on average on large benchmarks:

  DS2L is **>20x** resp. **>250x faster**

- DS2L's errors ~2x larger than Fluctuat's
- DS2L's errors comparable to Satire

Combining Analysis and Optimization

2 Master Theses supervised with Eva Darulova:

Numerical Analysis and Optimisation of Functional Array Programs

Simon Björklund

Exploring Accuracy and Performance Trade-offs in Functional Array Programs

Filip von Knorring

Simon's Work

Daisy

- Bounds rounding errors using static analysis
- Implements several different methods
- Can now analyse functional array programs
- Programs are given in a language called **DS^{2}L** (DS · DSL)

Shine

- Compiles a language called **Rise** into low-level languages like **C**
- Uses the high-level formulation to perform optimisations
- The optimisations can be customised for different applications
- For us: only those that preserve floating-point behaviour

Simon’s Work

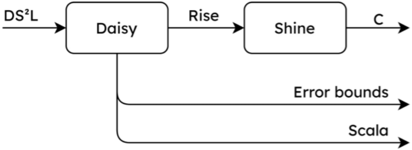
```
def dot(a: Vector, b: Vector): Real = {
    require(
        a ⩾ 0.0 && a ⩽ 1.0 && a.size(100) &&
        b ⩾ 0.0 && b ⩽ 1.0 && b.size(100)
    )
    (a * b).sum()
}
```

```
depFun((n: Nat) ⇒ fun(n.f64)(a ⇒ fun(n.f64)(b ⇒
    zip(a)(b)
        ▷ map(fun(x ⇒ fst(x) * snd(x)))
        ▷ reduce(add)(lf64(0.0))
)))
```

```
void dot(double* output, int n, double* a, double* b) {
    for (int i = 0; i < n; i++) {
      output += (a[i] * b[i]);
    }
}
```

Simon's Work

Results!



- Translated a set of 14 benchmarks from **DS²L** to **Rise**
- Neither can be ran, so we compile them further
- **DS²L → Scala**, **Rise → C**
- Run both programs with 500 random inputs
- Compare performance and output values of both

Simon's Work

Benchmark	Iterations	Median Scala Time	Median C Time	Speedup
`alphaBlending`	**500k**	2046ns ± 10ns	18ns ± 0ns	114x
`avg`	**500k**	591ns ± 4ns	37ns ± 0ns	16x
`controllerTora`	29k	6585ns ± 35ns	3178ns ± 5ns	2x
`goubault`	**500k**	533ns ± 3ns	93ns ± 0ns	6x
`harmonic`	**500k**	637ns ± 4ns	143ns ± 1ns	4x
`lorentz`	**500k**	263ns ± 1ns	74ns ± 0ns	4x
`lyapunov`	**500k**	1909ns ± 11ns	31ns ± 0ns	62x
`nonlin1`	178k	940ns ± 6ns	453ns ± 0ns	2x
`nonlin2`	303k	524ns ± 2ns	249ns ± 0ns	2x
`nonlin3`	208k	897ns ± 6ns	382ns ± 0ns	2x
`pendulum`	100k	6159ns ± 33ns	797ns ± 9ns	8x
`roux1`	**500k**	533ns ± 3ns	93ns ± 0ns	6x
`stdDeviation`	**500k**	1058ns ± 6ns	112ns ± 0ns	9x
`variance`	**500k**	1057ns ± 7ns	111ns ± 0ns	10x

Simon's Work

```
def lorentz(m: Matrix): Vector = {
    require(m ⩾ 1.0 && m ⩽ 2.0 && m.size(21,3))

    val init: Vector = m.row(0)
    m.fold(init)((acc, v) ⇒ {
        val x:Real = acc.at(0)
        val y:Real = acc.at(1)
        val z:Real = acc.at(2)
        val tmpx:Real = x + 10.0*(y - x)*0.005
        val tmpy:Real = y + (28.0*x - y - x*z)*0.005
        val tmpz:Real = z + (x*y - 2.666667*z)*0.005
        Vector(List(tmpx,tmpy,tmpz))
    })
}
```

Error is 4.3e-14

Simon’s Work

```
depFun((n: Nat) ⇒
  fun(n.3.f64)(m ⇒
    fun(init ⇒
      m ▷ reduceSeq(fun(acc ⇒ fun(v ⇒
        fun(x ⇒
          fun(y ⇒
            fun(z ⇒
              fun(tmpx ⇒
                fun(tmpy ⇒
                  fun(tmpz ⇒
                    makeArray(3)(tmpx)(tmpy)(tmpz) ▷ mapSeqUnroll(fun(x ⇒ x)) ▷ toMem
                  )
                  ((z + (((x * y) - (lf64(2.666667) * z)) * lf64(0.005))))
                )
                ((y + ((((lf64(28.0) * x) - y) - (x * z)) * lf64(0.005))))
              )
              ((x + ((lf64(10.0) * (y - x)) * lf64(0.005))))
            )
            (acc ▷ element(2))
          )
          (acc ▷ element(1))
        )
        (acc ▷ element(0))
      )))(init)
    )
    (m ▷ element(0))
  )
)
```

Simon’s Work

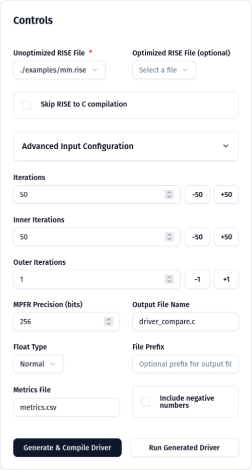
```
void lorentz(double* output, int n, double* m) {
    // set fold init as first row of m
    for (int i = 0; i < 3; i++) {
        output[i] = m[i];
    }

    // perform fold, storing the accumulator in output
    for (int i = 0; i < n; i++) {
        output[0] += ((10.0 * (output[1] - output[0])) * 0.005);
        output[1] += ((((28.0 * output[0]) - output[1]) - (output[0] * output[2])) * 0.005);
        output[2] += (((output[0] * output[1]) - (2.666667 * output[2])) * 0.005);
    }
}
```

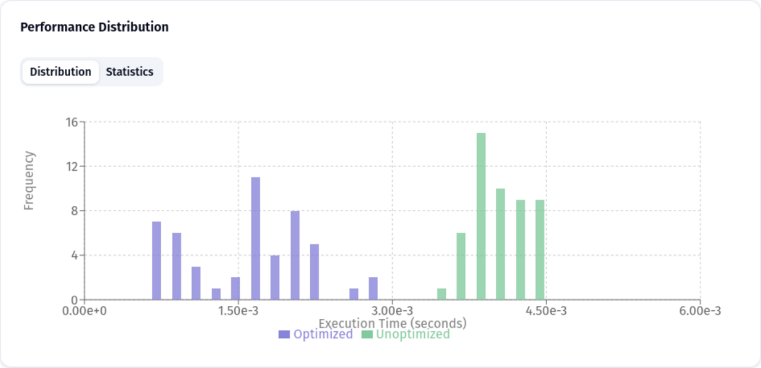
Error is still
4.3e-14

Filip's Work

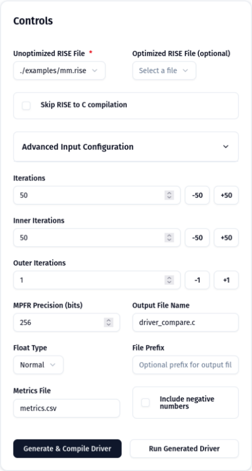
- curated a set of benchmark Rise programs
- each with an unoptimized reference and optimized variants
- generated executable code for both standard and high-precision evaluation (MPFR)
  shadow execution technique
- visualized the trade-offs between accuracy and performance
- tested different input distributions varying subnormal values and large dynamic ranges

Filip’s Work

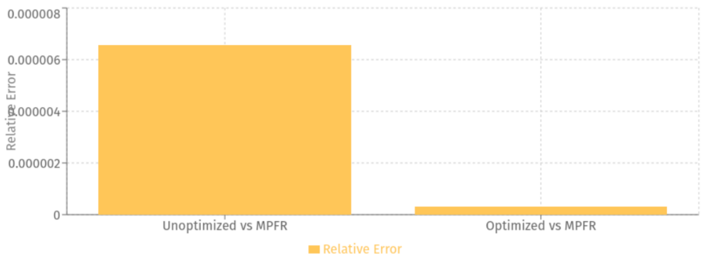

Controls
Unoptimized RISE File *
./examples/mm.rise
Optimized RISE File (optional)
Select a file
Skip RISE to C compilation
Advanced Input Configuration
Iterations
50
-50
+50
Inner Iterations
50
-50
+50
Outer Iterations
1
-1
+1
MPFR Precision (bits)
256
Output File Name
driver_compare.c
Float Type
Normal
File Prefix
Optional prefix for output fil
Metrics File
metrics.csv
Include negative numbers
Generate & Compile Driver
Run Generated Driver


Performance Distribution
Distribution
Statistics
Frequency
16
12
8
4
0
0.00e+0
1.50e-3
3.00e-3
4.50e-3
6.00e-3
Execution Time (seconds)
Optimized
Unoptimized

Filip’s Work



Accuracy Comparison



Metric	Unoptimized	Optimized
Absolute Error	3.4451782894320786	0.1639282894320786
Relative Error	0.000006571188512796013	0.00000031266994098466437
ULPs	110.246249056	5.245731137

Filip’s Work

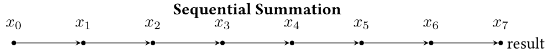


Figure 5.1: Sequential summation of x_0 through x_7, evaluated left to right.

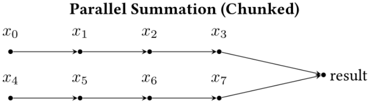


Figure 5.2: Chunked parallel reduction: pairs of local reductions followed by a sequential aggregation.

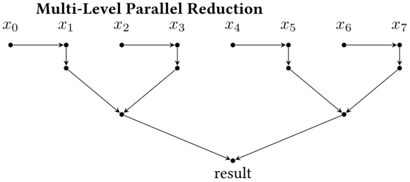


Figure 5.3: Multi-level parallel reduction with four chunked rows merged into two intermediaries, then a final result.

Filip’s Work

Table 5.1: Summation Benchmark: Performance vs Accuracy

Variant	Runtime (avg)	Speedup	Abs. Err.	ULPs	Runtime Stats (min/med/max/stddev)
Sequential	14.27ms	1×	58.47	467.80	12.55ms/13.63ms/28.94ms/197.90µs
Parallel	4.43ms	3.22×	1.77	14.18	3.17ms/4.28ms/7.26ms/155.61µs
Multi-Stage Parallel	4.84ms	3.19×	0.97	7.81	2.70ms/4.74ms/8.11ms/0s
Vectorized Parallel	5.85ms	2.54×	0.02	0.18	3.40ms/5.82ms/13.57ms/91.89µs

ULP = Unit in the Last Place ≃ spacing between adjacent representable numbers

Filip’s Work

Table 5.2: Average Benchmark: Performance vs Accuracy

Variant	Runtime (avg)	Speedup	Abs. Err.	ULPs	Runtime Stats (min/med/max/stddev)
Sequential	5.78ms	1×	7.56e-9	0.25	3.35ms/5.56ms/ 11.89ms/85.83µs
Parallel	1.62ms	3.56×	7.56e-9	0.25	750.83µs/1.39ms/ 4.49ms/130.24µs

Filip's Work

Table 5.4: Matrix Multiplication Benchmark

Variant	Runtime (avg)	Speedup	Abs. Err.	ULPs	Runtime Stats (min/med/max/stddev)
Sequential	577.27ms	1×	2.25e-5	2.81	464.59ms/562.19ms/ 2.31s/49.91ms
Parallel	265.11ms	2.17×	2.25e-5	2.81	199.70ms/262.77ms/ 613.50ms/19.28ms
Float Unsafe Parallel	293.79ms	1.77×	7.27e-6	0.90	227.03ms/301.46ms/ 440.32ms/700.35µs

Filip's Work

Table 5.5: Parallel Sum Reduction with Varying Input Distributions

Benchmark	Runtime (avg)	Speedup	Abs. Error	ULPs	Rel. Error
Normal	4.42ms	3.22×	1.77	14.18	8.45e-7
Negative	4.50ms	3.18×	0.01	15.42	9.19e-7
Subnormal	4.39ms	3.26×	1.64e-41	131.59	7.84e-6
Mixed	4.78ms	3.05×	3.73	59.74	3.56e-6
Magnitude	4.74ms	3.09×	1.57e+19	11.72	6.98e-7

Future Work

- Porting Daisy analysis directly over Rise programs
- Arbitrary dimensions, mixed precision, and mixing rewrites preserving either real semantics or float semantics
- More advanced design-space exploration and visualizing pareto fronts
- Support imperative code in the long term, for example building upon OptiTrust annotations to keep analyzing functional specifications

Thanks!

Discussion points:

- Do you write optimized code or libraries by hand?
- Would you use an optimization assistant?
- What do you need from it?
- How do you want to interact with it? With scripts? With sketches? With cost functions? What kind of feedback?
- Would you create your own libraries of domain-specific abstractions and optimizations?
- What kind of numerical accuracy guarantees or capabilities do you need? Would you write accuracy specifications and what would they look like?
- Do you have benchmarks of progressive difficulty for us to look into?

The End