Towards Safe Interactive Optimization Across Layers

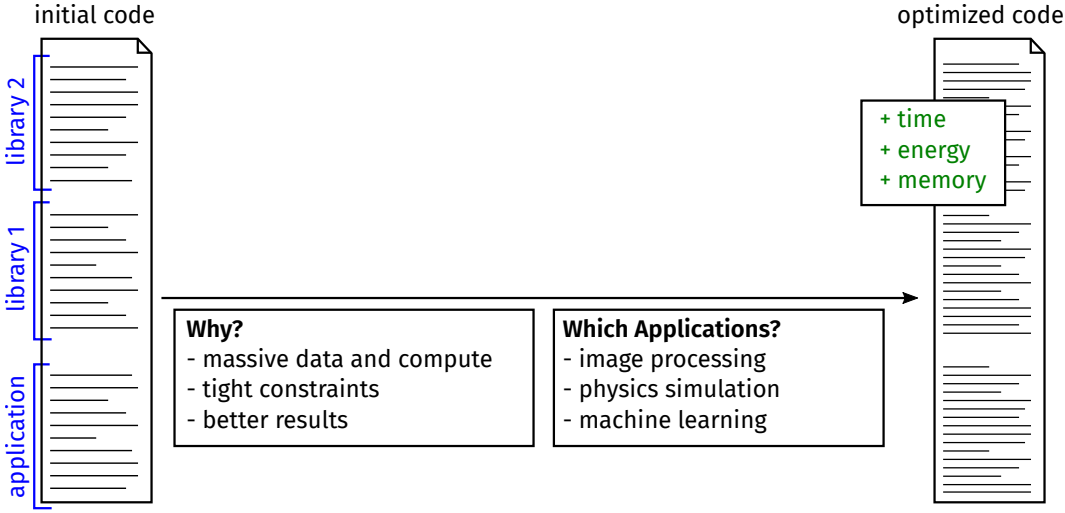
Thomas Kœhler thok.eu

Séminaire CASH — July 2024, Lyon

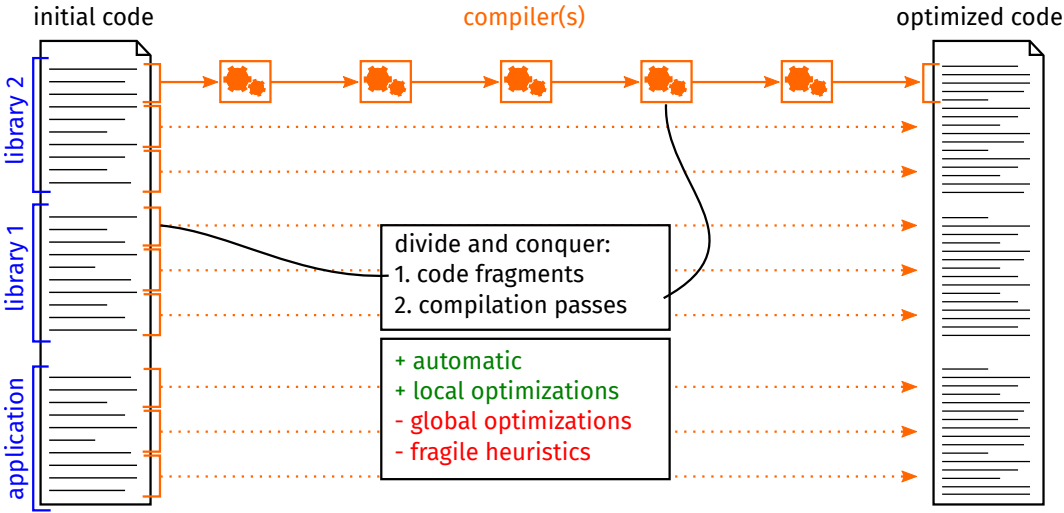
My Career


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
Inria CAMUS, Strasbourg
Inria
cnrs
CRCN Researcher

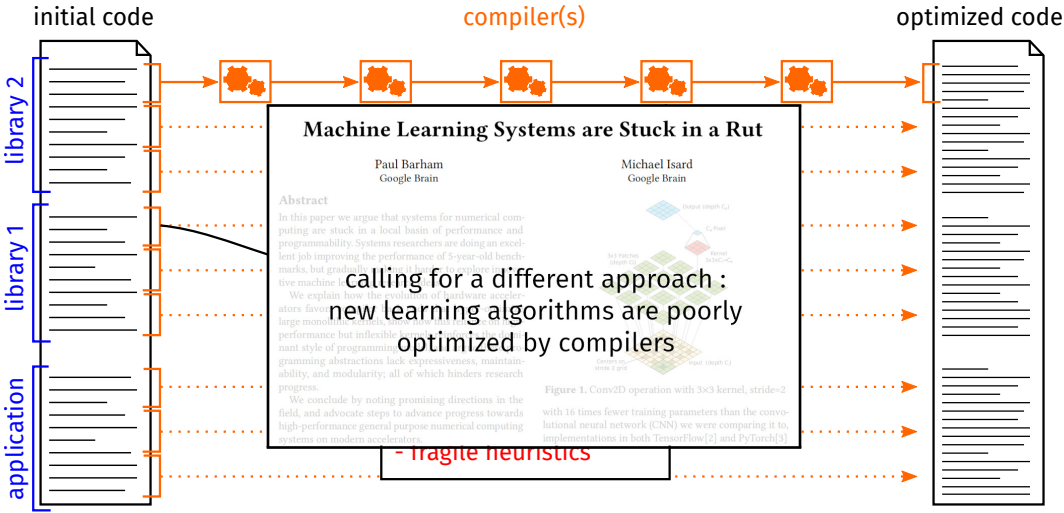
Optimizing Programs



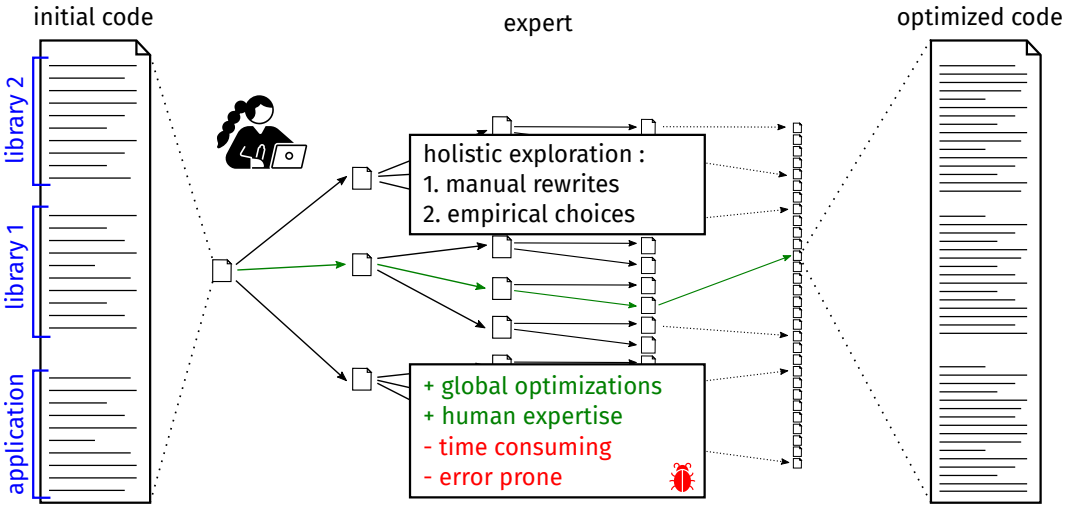
Automatic Program Optimization



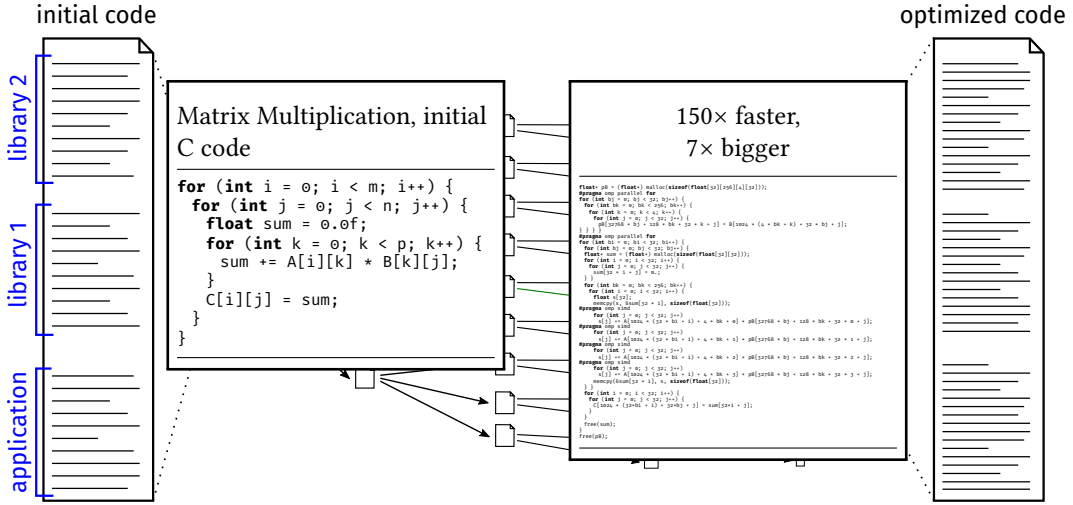
Automatic Program Optimization



Manual Program Optimization



Manual Program Optimization

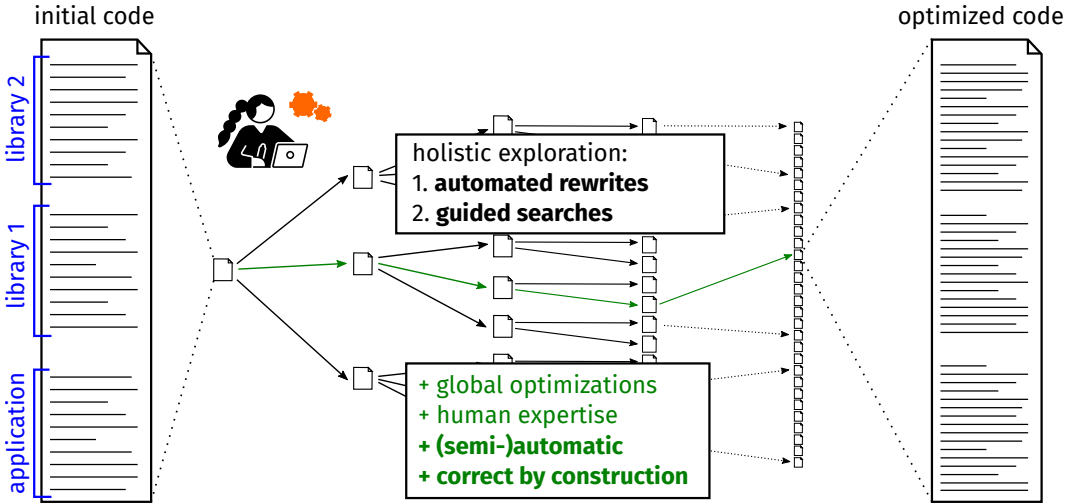


What Future for Program Optimization?

- 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 Safe Interactive Optimization Across Layers

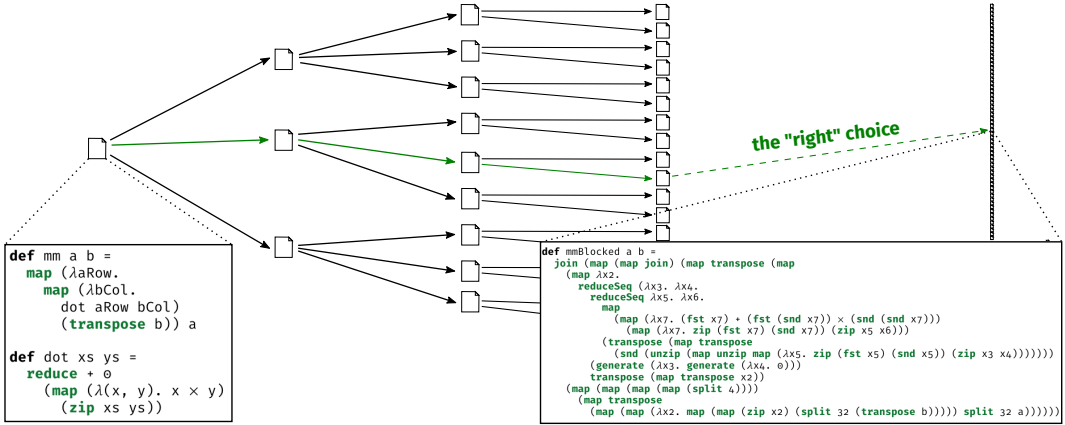


PhD: Functional Program Rewriting

rewrite rules

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

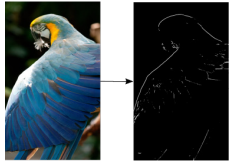
combinatorial rewriting space, correct and extensible



Achieving Expert Optimizations by Composition

6 expert optimizations
⟶ decomposed into **74 rules**

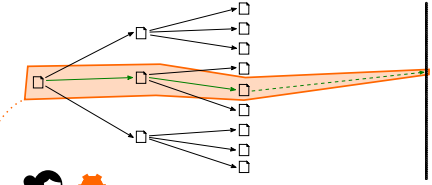
Harris: corner and edge detection



16x faster than OpenCV,
1.4x faster than Halide! **[CGO'21** *A***]**

specialized compiler of industrial strength

enormous rewrite space,
10x bigger programs than before,
thousands of rewrite steps



rewriting strategies:
[ICFP'20 *A**, **CACM'23** 🏆 **]**

```
baseline ;
tile(32,32)      @ outermost(mapNest(2))   ;;
fissionReduceMap @ outermost(appliedReduce);;
split(4)         @ innermost(appliedReduce);;
reorder(List(1,2,5,6,3,4))
```

Combining Automatic Search and Human Expertise

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

=

automatic rewrite search,
sharing equivalent subterms

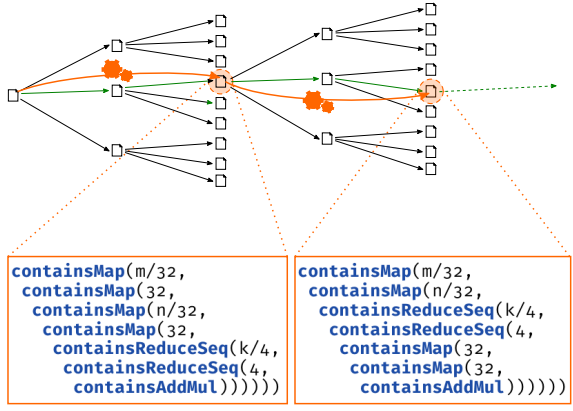
+

specifying guides as program sketches

MIT PL Review: 🏆
"significant potential to shape the future direction of PL research and/or industry practice"

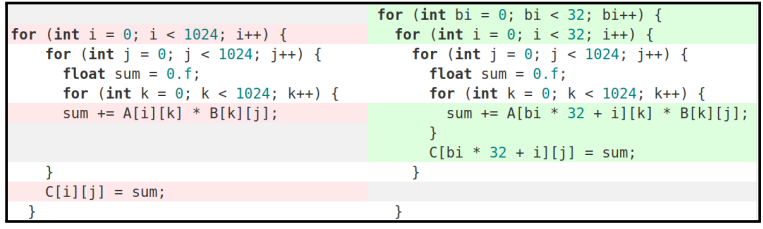
theorem proving collaboration:
Andrés Goens, University of Amsterdam

guided search :
582x faster, 116x less memory



PostDoc: Transforming Imperative Programs

- add **imperative choices** to the optimization space *reusing memory*
- **general-purpose language** more expressive than functional array language *C, C++*



```
for (int i = 0; i < 1024; i++) {
    for (int j = 0; j < 1024; j++) {
      float sum = 0.f;
      for (int k = 0; k < 1024; k++) {
      sum += A[i][k] * B[k][j];
    }
    C[i][j] = sum;
  }
```

```
for (int bi = 0; bi < 32; bi++) {
  for (int i = 0; i < 32; i++) {
    for (int j = 0; j < 1024; j++) {
      float sum = 0.f;
      for (int k = 0; k < 1024; k++) {
        sum += A[bi * 32 + i][k] * B[k][j];
      }
      C[bi * 32 + i][j] = sum;
    }
  }
```

The OptiTrust Project:

- user-guided source-to-source transformations
- validated through static resource analysis *based on separation logic*
- internally: imperative λ-calculus

Expert Optimizations over General-Purpose Imperative Code

Realized Case Studies:

with Arthur Charguéraud and Guillaume Bertholon

- **Matrix Multiplication**
  linear algebra, baseline: TVM specialized compiler — loop and data layout transformations, OpenMP parallelism
- Box Filter (row-based): blurring and denoising
  image processing, baseline: OpenCV optimized library — multi-versioning, algorithmic sliding window optimization

In Progress:

- Harris: corner and edge detection
  image processing, baseline: specialized compiler Halide — uneven tiling, operator fusion, circular buffers, CSE
- Particle-In-Cell: plasma simulation
  physics simulation, baseline: manual expert optimizations — AoS ↔ SoA, transforming values in memory, atomic bags

Optimizing MatMul with Domain-Specific Compilers

What to compute: TVM Algorithm

```
pB = tvm.compute((N / 32, P, 32),
  lambda bj, k, j:
  B[k, bj * 32 + j])

C = tvm.compute((M, N),
  lambda i, j:
  sum(A[i, k] *
      pB[j // 32, k, j % 32],
      axis=k))
```

How to compute: TVM Schedule

```
CC = s.cache_write(C, "global")
bi, bj, i, j = s[C].tile(
  C.op.axis[0], C.op.axis[1], 32, 32)
s[CC].compute_at(s[C], bj)
i2, j2 = s[CC].op.axis
(kaxis,) = s[CC].op.reduce_axis
bk, k = s[CC].split(kaxis, factor=4)
s[CC].reorder(bk, i2, k, j2)
s[CC].vectorize(j2)
s[CC].unroll(k)
s[C].parallel(bi)
bj3, _, j3 = s[pB].op.axis
s[pB].vectorize(j3)
s[pB].parallel(bj3)
```

Optimizing MatMul with Domain-Specific Compilers

What to compute: TVM Algorithm

```
pB = tvm.compute((N / 32, P, 32),
  lambda bj, k, j:
  B[
C =
  la
  su
      pB[j // 32, k, j % 32],
      axis=k))
```

high-level DSL
restricted domain
rewritten for schedule

How to compute: TVM Schedule

```
CC = s.cache_write(C, "global")
bi, bj, i, j = s[C].tile(
  C.op.axis[0], C.op.axis[1], 32, 32)
s[CC].c
i2, j2
(kaxis,
bk, k =
s[CC].r
s[CC].v
s[CC].u
s[C].pa
bj3, _, j3 = s[pB].op.axis
s[pB].vectorize(j3)
s[pB].parallel(bj3)
```

concise
safe
built-in optimizations
black box code generation
implicit heuristics + unfamiliar IR

MatMul Optimization with OptiTrust

What to compute: general-purpose code

```
void mm(float* C, float* A, float* B,
        int m, int n, int p) {
  for (int i = 0; i < m; i++) {
    for (int j = 0; j < n; j++) {
      float sum = 0.0f;
      for (int k = 0; k < p; k++)
        sum += A[i][k] * B[k][j];
      C[i][j] = sum;
    }
  }
}
```

How to optimize: transformation script

```
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"];
```

MatMul Optimization with OptiTrust

What to compute: general-purpose code

```
void mm(float* C, float* A, float* B,
        int m, int n, int p) {
  __reads("A ~> Matrix2(m, p)");
  __reads("B ~> Matrix2(p, n)");
  __modifies("C ~> Matrix2(m, n)");
  for (int i = 0; i < m; i++) {
    __xmodifies("for j in 0..n→  &C[i][j]~> Cell")
        ;
    for (int j = 0; j < n; j++) {
      __xmodifies("&C[i][j] ~> Cell");
      float sum = 0.0f;
      for (int k = 0; k < p; k++)
        sum += A[i][k] * B[k][j];
      C[i][j] = sum;
    }
  }
}
```

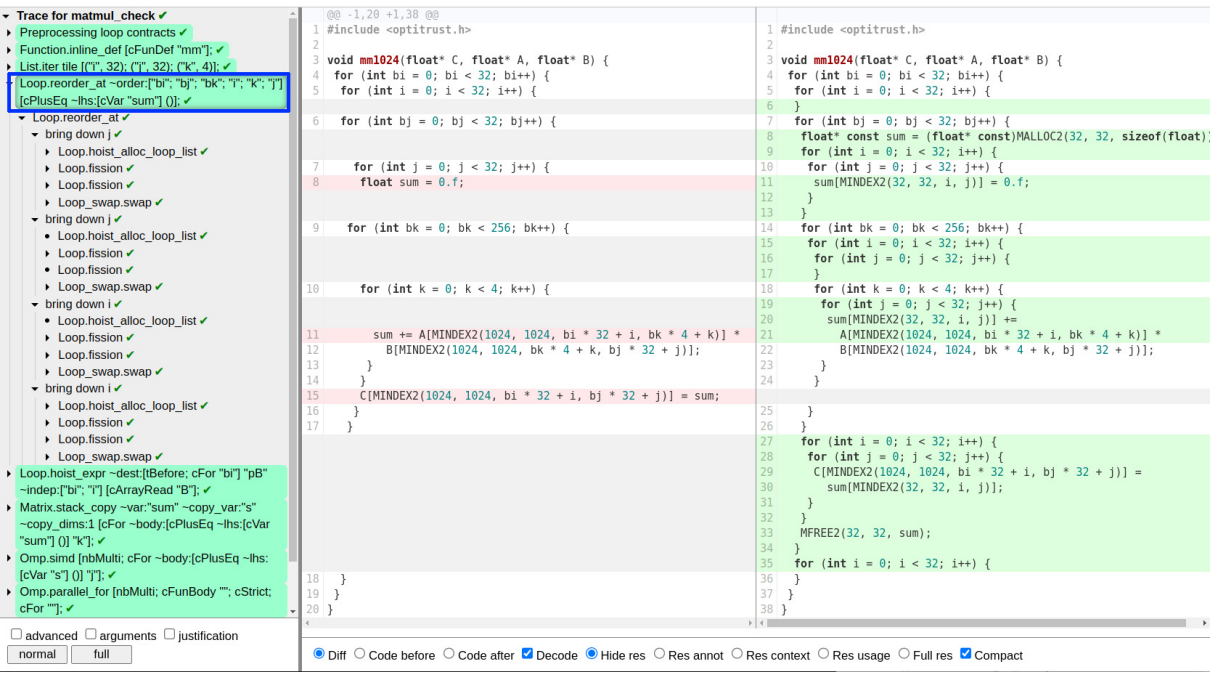
How to optimize: transformation script

```
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"];
```

MatMul Resource Contract for OptiTrust

```
void mm(float* C, float* A, float* B, int m, int n, int p) {
  __reads("A ⇝ Matrix2(m, p), B ⇝ Matrix2(p, n)");
  __modifies("C ⇝ Matrix2(m, n)");
  // [...]
}
```

- clauses describe ownership of resources (permissions)
- a matrix is a conjunction of cells: $\star_{i \in 0..m} \star_{j \in 0..n}$ (`&C[i][j] ⇝ Cell`)


Trace for matmul_check ✔
Preprocessing loop contracts ✔
Function.inline_def [cFunDef "mm"]; ✔
List.iter tile [("i", 32); ("j", 32); ("k", 4)]; ✔
Loop.reorder_at ~order:["bi"; "bj"; "bk"; "i"; "k"; "j"] [cPlusEq ~lhs:[cVar "sum"] ()]; ✔
Loop.reorder_at ✔
bring down j ✔
Loop.hoist_alloc_loop_list ✔
Loop.fission ✔
Loop.fission ✔
Loop_swap.swap ✔
bring down j ✔
Loop.hoist_alloc_loop_list ✔
Loop.fission ✔
Loop.fission ✔
Loop_swap.swap ✔
bring down i ✔
Loop.hoist_alloc_loop_list ✔
Loop.fission ✔
Loop.fission ✔
Loop_swap.swap ✔
bring down i ✔
Loop.hoist_alloc_loop_list ✔
Loop.fission ✔
Loop.fission ✔
Loop_swap.swap ✔
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 ~lhs:[cVar "sum"] ()] "k"]; ✔
Omp.simd [nbMulti; cFor ~body:[cPlusEq ~lhs:[cVar "s"] ()] "j"]; ✔
Omp.parallel_for [nbMulti; cFunBody ""; cStrict; cFor ""]; ✔
advanced
arguments
justification
normal
full
@@ -1,20 +1,38 @@
#include <optitrust.h>
void mm1024(float* C, float* A, float* B) {
for (int bi = 0; bi < 32; bi++) {
for (int i = 0; i < 32; i++) {
for (int bj = 0; bj < 32; bj++) {
for (int j = 0; j < 32; j++) {
float sum = 0.f;
for (int bk = 0; bk < 256; bk++) {
for (int k = 0; k < 4; k++) {
sum += A[MINDEX2(1024, 1024, bi * 32 + i, bk * 4 + k)] *
B[MINDEX2(1024, 1024, bk * 4 + k, bj * 32 + j)];
}
}
C[MINDEX2(1024, 1024, bi * 32 + i, bj * 32 + j)] = sum;
}
}
}
}
#include <optitrust.h>
void mm1024(float* C, float* A, float* B) {
for (int bi = 0; bi < 32; bi++) {
for (int i = 0; i < 32; i++) {
}
for (int bj = 0; bj < 32; bj++) {
float* const sum = (float* const)MALLOC2(32, 32, sizeof(float));
for (int i = 0; i < 32; i++) {
for (int j = 0; j < 32; j++) {
sum[MINDEX2(32, 32, i, j)] = 0.f;
}
}
for (int bk = 0; bk < 256; bk++) {
for (int i = 0; i < 32; i++) {
for (int j = 0; j < 32; j++) {
}
for (int k = 0; k < 4; k++) {
for (int j = 0; j < 32; j++) {
sum[MINDEX2(32, 32, i, j)] +=
A[MINDEX2(1024, 1024, bi * 32 + i, bk * 4 + k)] *
B[MINDEX2(1024, 1024, bk * 4 + k, bj * 32 + j)];
}
}
}
}
for (int i = 0; i < 32; i++) {
for (int j = 0; j < 32; j++) {
C[MINDEX2(1024, 1024, bi * 32 + i, bj * 32 + j)] =
sum[MINDEX2(32, 32, i, j)];
}
}
MFREE2(32, 32, sum);
}
for (int i = 0; i < 32; i++) {
}
}
}
Diff
Code before
Code after
Decode
Hide res
Res annot
Res context
Res usage
Full res
Compact

Decomposition of MatMul Steps by OptiTrust

Combined Transformations:

- compose basic transformations
- validated by transitivity

Basic Transformations:

- direcly modify AST

  imperative λ-calculus
- check sufficient correctness conditions

8 MatMul Steps:

- 55 basic transformations
- 61 “ghost” transformations

- `Instr`: `move`, `delete`, `insert`, `duplicate`
- `Function`: `inline`
- `Variable`: `inline`, `bind`, `to_const`, `init_detach`, `local_name`
- `Loop`: `fusion`, `fission`, `swap`, `hoist`, `move_out`, `tile`, `collapse`, `unroll`
- `Matrix`: `intro_malloc0`,
- `Omp`: `parallel_for`, `simd`
- `Arith`: `simpl`
- …

OptiTrust's Sufficient Condition for `Instr.move`

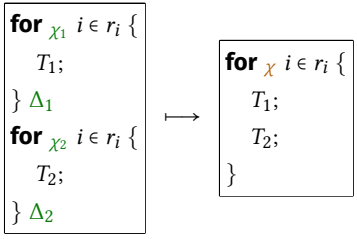
$$E\left[\begin{array}{l} T_1; \Delta_1 \\ T_2; \Delta_2 \end{array}\right] \longmapsto E\left[\begin{array}{l} T_2; \\ T_1; \end{array}\right]$$

correct if:

$$\begin{cases} \Delta_1.\mathsf{notRO} \cap \Delta_2 = \varnothing \\ \Delta_2.\mathsf{notRO} \cap \Delta_1 = \varnothing \end{cases}$$

- leverages computed resource information (Δ_1)
- correct if resources are exclusively shared in read-only

OptiTrust's Sufficient Condition for `Loop.fusion`



correct if:

$$\begin{cases} i \text{ not free in } \chi_1.\mathsf{shrd} \text{ and } \chi_2.\mathsf{shrd} \\ \chi_1.\mathsf{shrd} \vdash (\Delta_1.\mathsf{notRO} \cap \Delta_2) = \varnothing \\ \chi_2.\mathsf{shrd} \vdash (\Delta_2.\mathsf{notRO} \cap \Delta_1) = \varnothing \end{cases}$$

with:

$$\_, \_, Q_1, P_2 \equiv \mathsf{PartialSub}(\chi_1.\mathsf{excl.post}, \chi_2.\mathsf{excl.pre}) \qquad \chi \equiv \begin{cases} \mathsf{vars} \equiv \chi_1.\mathsf{vars}, \chi_2.\mathsf{vars} \\ \mathsf{shrd} \equiv \chi_1.\mathsf{shrd} \star \chi_2.\mathsf{shrd} \\ \mathsf{excl.pre} \equiv \chi_1.\mathsf{excl.pre} \star P_2 \\ \mathsf{excl.post} \equiv \chi_2.\mathsf{excl.post} \star Q_1 \end{cases}$$

Future: Safe Interactive Optimization Across Layers

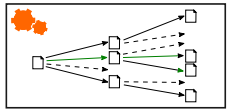
image processing

machine learning

physics simulation

• • •

Goal:
build the first
Optimization Assistant



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

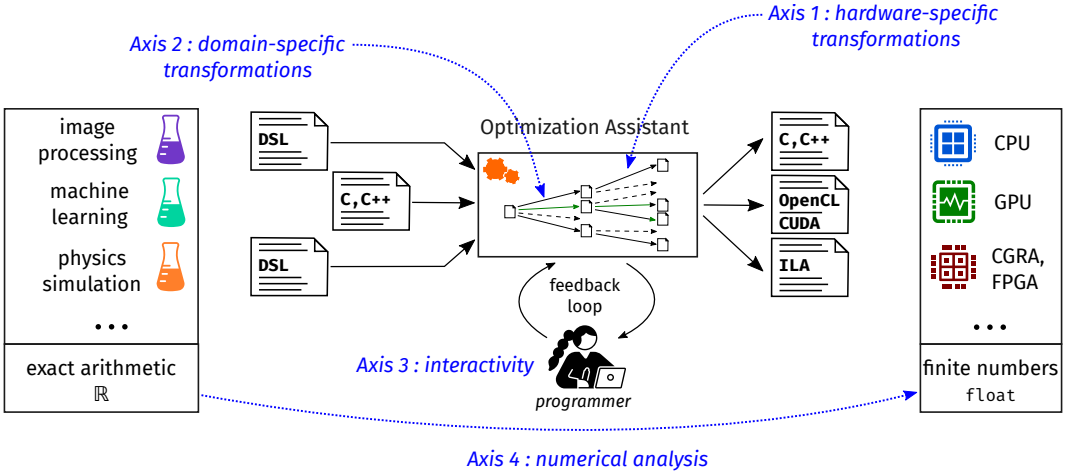
CPU

GPU

CGRA, FPGA

• • •

Future: Safe Interactive Optimization Across Layers



Future: Safe Interactive Optimization Across Layers

- Looking for Master students (interns)
- Preparing for PhD students (ANR JCJC)
- Starting fruitful collaborations
  - Combine different expertise
  - Leverage overlapping interests
  - Exchange benchmarks and applications
- Creating a French, European, and International network (workshops, ERC, ..)

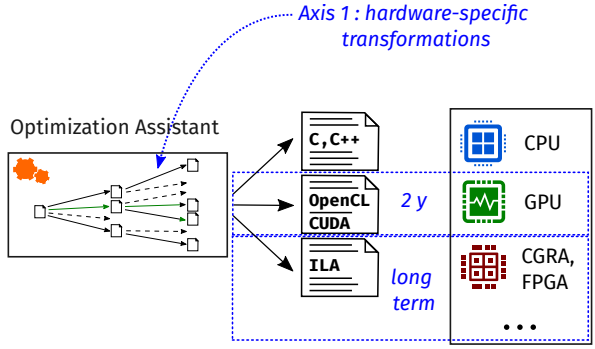
Or, .. maximize fun and impact by not working alone

Axis 1: Correct and Composable Hardware-Specific Transformations

Goal: create an optimization space
that evolves with hardware
by composition of transformations

Approach:
- decompose expert optimizations
  into simpler transformations

Difficulty:
- identify and guarantee the correctness
  of transformations, taking programming
  models into account

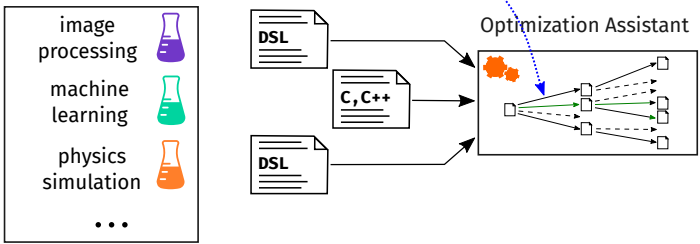


International Collaborations:
- Bastian Köpcke, Universität Münster [**arXiv'22**, GPGPU'22, PLDI'24]
- Jackson Woodruff, University of Edinburgh [PLDI'22, **arXiv'23**, CC'23]

Axis 2: Libraries of Composable Domain-Specific Transformations

Axis 2 : domain-specific transformations

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



Plan:
- 2 y: Halide-like DSL
- 4 y: combine with Axis 1 (GPUs)
- long term: mixing DSLs, libraries = functions + transformations + heuristics

Overlapping Interests?
- Gabriel Radanne
- AnyDSL team (Roland Leissa, Sebastian Hack)
- Inria EMERAUDE

Difficulties:
- synthesizing code annotations
- facilitate defining custom transformations

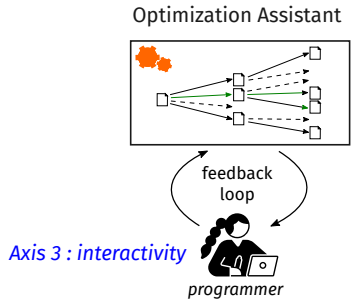
Axis 3: Interactivity between Programmer and Assistant

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

Plan:
- 3 years: interactive equality saturation on imperative code
  polyhedric and/or beam search
- long term: optimization suggestions and vizualisations
  bottlenecks, hot code

Difficulty:
- exploring a complex and evolving optimization space

Interdisciplinary Collaboration:
- Géry Casiez (Inria LOKI, Human-Computer Interaction)



Axis 4: Guaranteeing Numerical Accuracy during Optimization

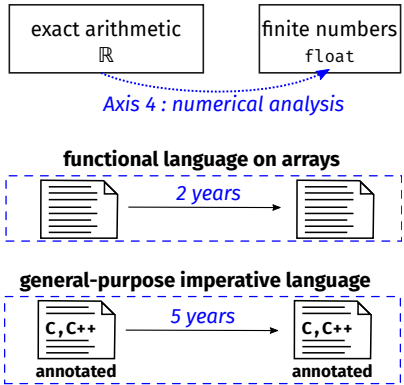
Goal: guarantee numerical accuracy during optimization, avoid errors and allow optimizations

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

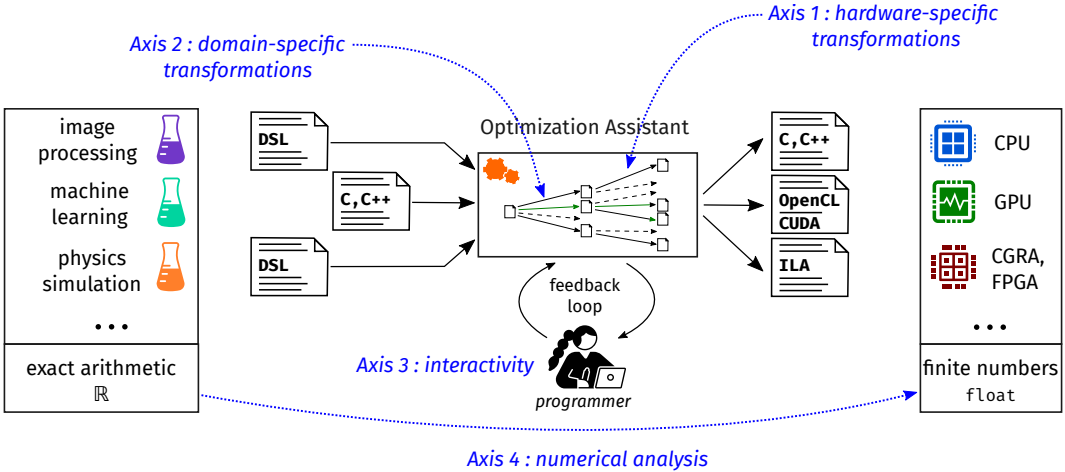
Approach:
- interval analysis on values and errors
  error is below `e(x o y) + d`

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

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



Towards Safe Interactive Optimization Across Layers

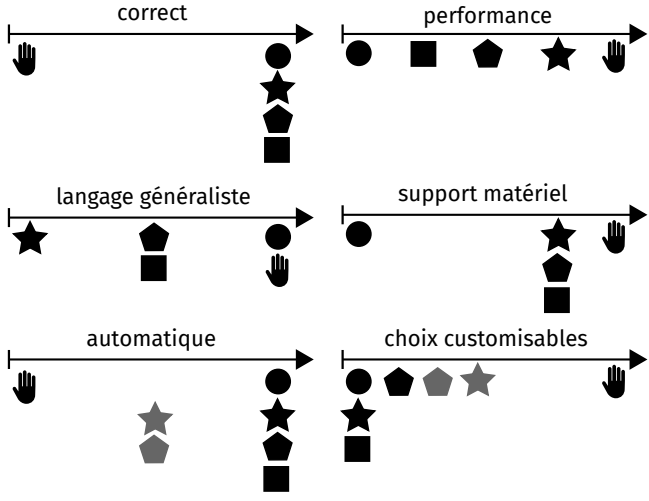


Thomas Kœhler thok.eu

Backup Slides

Les Compromis

- compilateurs généralistes
  gcc, clang
- optimisation manuelle
- compilateurs spécialisés
  Halide, TVM
- compilateurs polyédriques
  PLUTO, Polly
- compilateurs tableau
  Accelerate, SaC, Futhark



Les Compromis

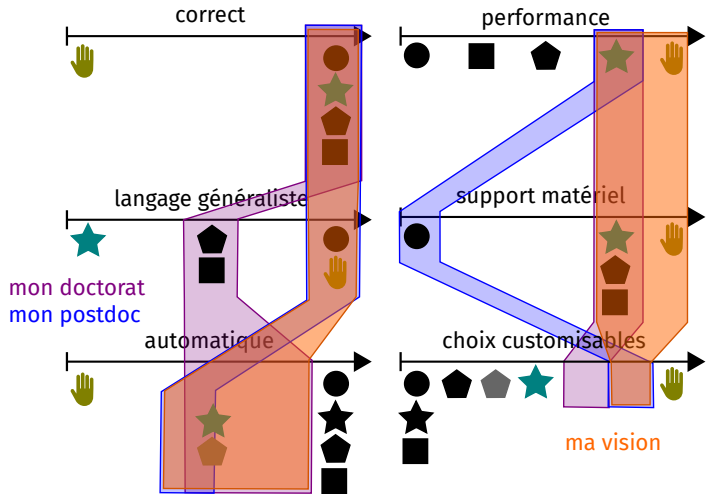
compilateurs généralistes
gcc, clang

optimisation manuelle

compilateurs spécialisés
Halide, TVM

compilateurs polyédriques
PLUTO, Polly

compilateurs tableau
Accelerate, SaC, Futhark

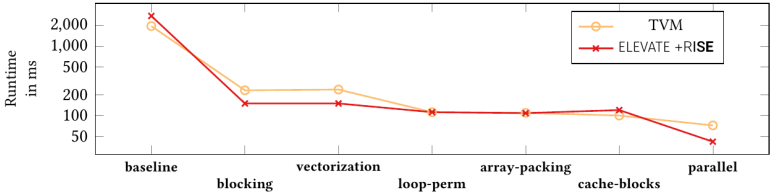


1st Benchmark: Matrix Multiplication on Intel CPU

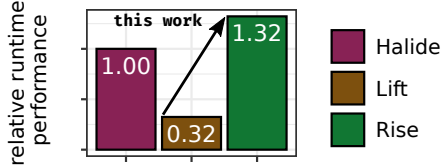
- 6 optimizations
  - transform loops *blocking, permutation, unrolling*
  - change data layout *array packing*
  - add parallelism *vectorization, multi-threading*

- performance is on par with reference schedules from TVM.

  https://tvm.apache.org/docs/how_to/optimize_operators/opt_gemm.html



2nd Benchmark: Corner Detection on ARM CPUs



- standard corner detection pipeline
- 6 well-known optimizations

  circular buffering, operator fusion, multi-threading, vectorization, convolution separation, register rotation

- extensibility + control

  $\Longrightarrow$

  faster code than Halide, with 2 additional optimizations

Matrix Multiplication Performance

- Intel(R) Core(TM) i7-8665U CPU, AVX2 (8 floats), 4 cores (8 hyperthreads)
- Relative speedup on 1024^3 input:

version	single-thread	multi-thread
unoptimized	1×	1×
optimized	46×	150×
TVM	46×	150×
numpy (Intel MKL)[1]	71×	183×

Both codes have 90th percentile runtime of 9.4ms over 200 benchmark runs, corresponding to a speedup of 150× compared to the 90th percentile of the naive code.

[1] uses assembly code, explicit vectorization, custom thread library