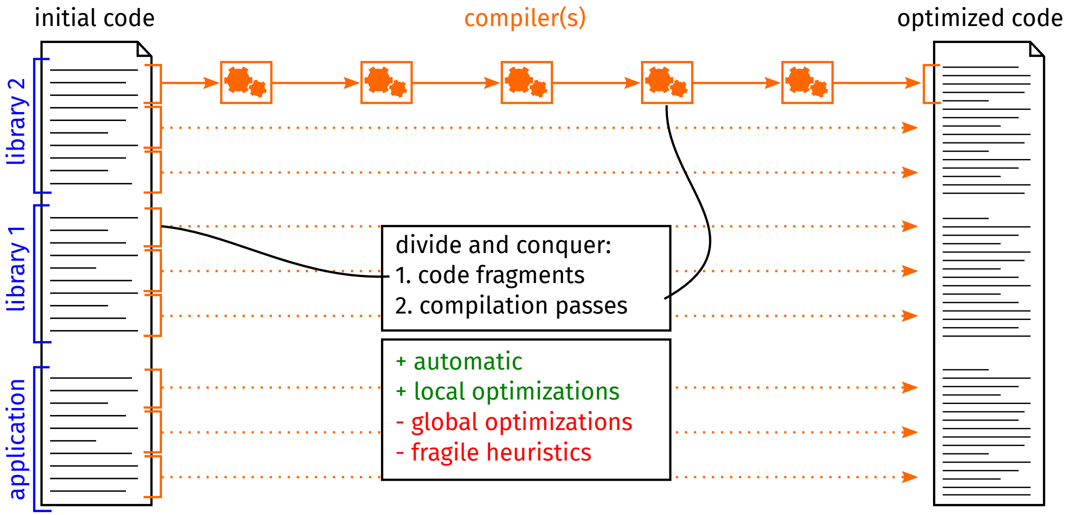
A Case For Interactive Optimization Assistants

Thomas Kœhler thok.eu

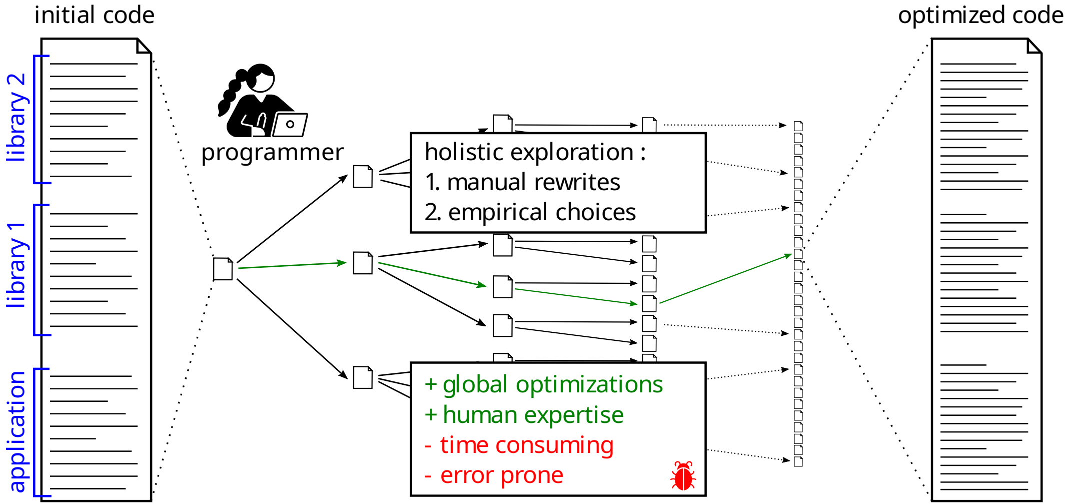
User-Schedulable Languages Workshop @ ASPLOS — March 2025, Rotterdam

How do we Optimize Programs?

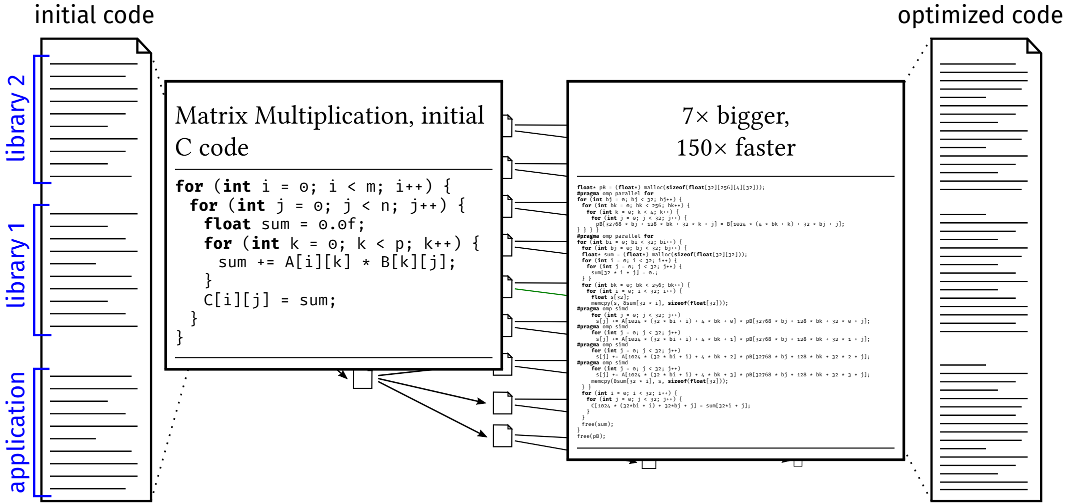
Automatic Compilation Passes



Manual Program Rewriting



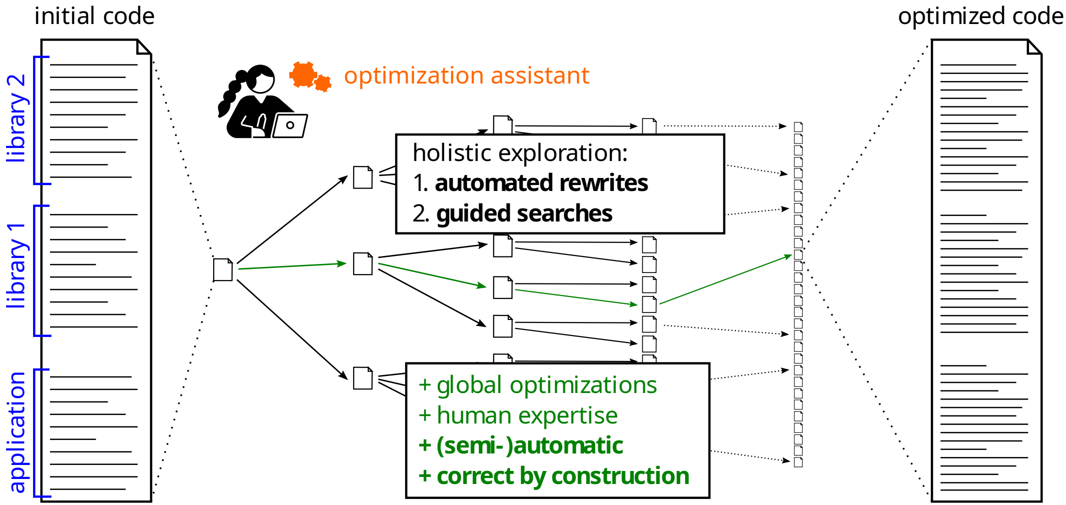
Manual Program Rewriting



We Need User-Scheduling

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

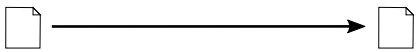
We Need Interactive Program Rewriting



My journey towards interactive optimization assistants:

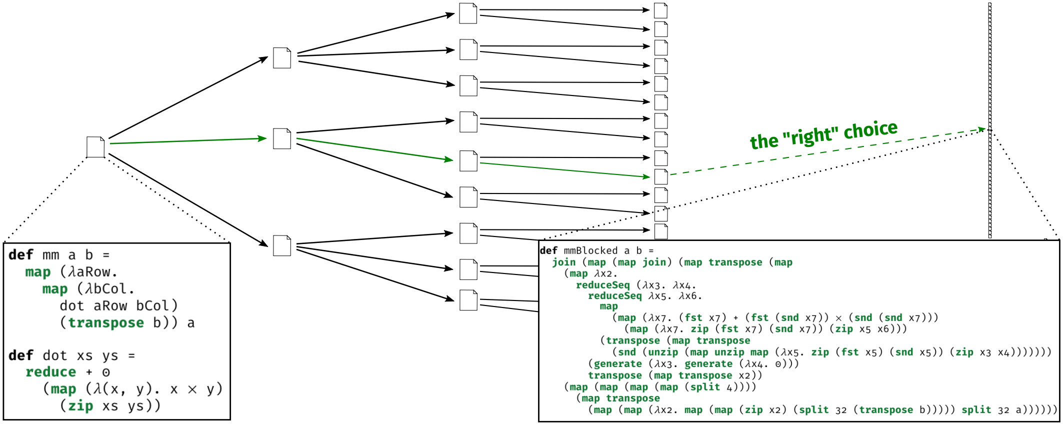
PhD: Functional Program Rewriting

rewrite rules



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

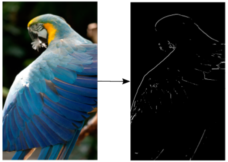
combinatorial rewriting space, correct and extensible



Achieving Expert Optimizations by Composition

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

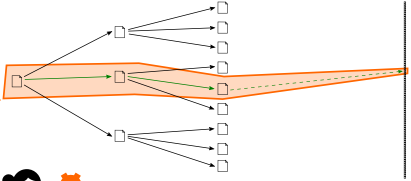
Harris: corner and edge detection



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

(4-core ARM Cortex A7)

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



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

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

Takeaway

Extensibility, composability and control matter.

- 6 expert optimizations = 51 generic + 19 backend + 4 specific rules
- can add rules without heavy compiler re-engineering
- can define custom optimization strategies through higher-order composition

Takeaway

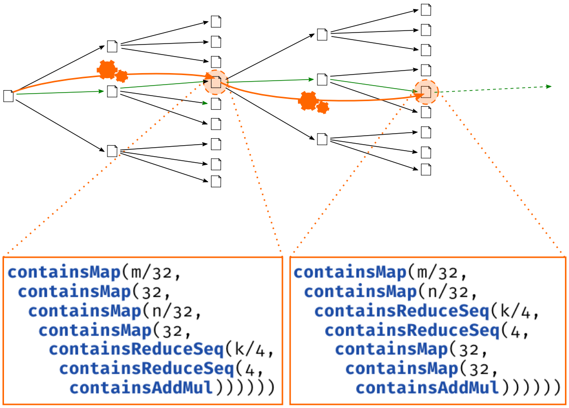
Premature control is the root of all evil.

- strategies took a lot of effort to write
  - Harris: 63 strategies, 600 lines of code
  - Matmul: 36 strategies, 200 lines of code
- user responsible for chaining and debugging thousands of rewrite steps
- strategies were often over-detailed and program-specific
- difficulties scaling to a more diverse and complex benchmark suite

Combining Automatic Search and Human Expertise

Guided Equality Saturation [POPL'24]
=
automatic rewrite search,
sharing equivalent subterms
+
specifying guides as program sketches

guided search :
582x faster, 116x less memory



Sketches Required for TVM-like Matrix Multiplication

1. split "loops"

```
containsMap(m/32,
 containsMap(32,
  containsMap(n/32,
   containsMap(32,
    containsReduceSeq(k/4,
     containsReduceSeq(4,
      containsAddMul))))))
```

2. reorder "loops"

```
containsMap(m/32,
 containsMap(n/32,
  containsReduceSeq(k/4,
   containsMap(32,
    containsReduceSeq(4,
     containsMap(32,
      containsAddMul))))))
```

3. introduce memory

```
containsMap(m / 32,
 containsMap(n / 32,
  containsReduceSeq(k / 4,
   containsMap(32,
    containsReduceSeq(4,
     containsMap(32,
      containsAddMul))))),
 containsToMem(n.k.f32,
  containsMap(n / 32,
   containsMap(k,
    containsMap(32.f32, ?)))))
```

4. thread, vectorize, unroll

```
containsMapPar(m / 32,
 containsMap(n / 32,
  containsReduceSeq(k / 4,
   containsMap(32,
    containsReduceSeqUnroll(4,
     containsMap(1,
      containsAddMulVec))))),
 containsToMem(n.k.f32,
  containsMapPar(n / 32,
   containsMap(k,
    containsMap(1.<32>f32, ?)))))
```

Takeaway

Semi-automation enables parsimonious control.

- simple sketches instead of complex strategies
- sketches 10× smaller than complete program, focus on key optimization insights
- sufficient for guided search to infer the rewrites and missing program details

Takeaway

Parsimonious control is not enough for productivity.

- users are not supported in iteratively developing their sketch sequences
- little feedback to help decision-making
- need to learn an unfamiliar language to write sketches
- *also relevant to developing rewriting strategies*

PostDoc: Assisting Interactive Optimization

OptiTrust Optimization Assistant

What to compute: C code

```c
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: OCaml script

```ocaml
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"];
```

PostDoc: Assisting Interactive Optimization

OptiTrust Optimization Assistant

What to compute: C 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: OCaml 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"];
```

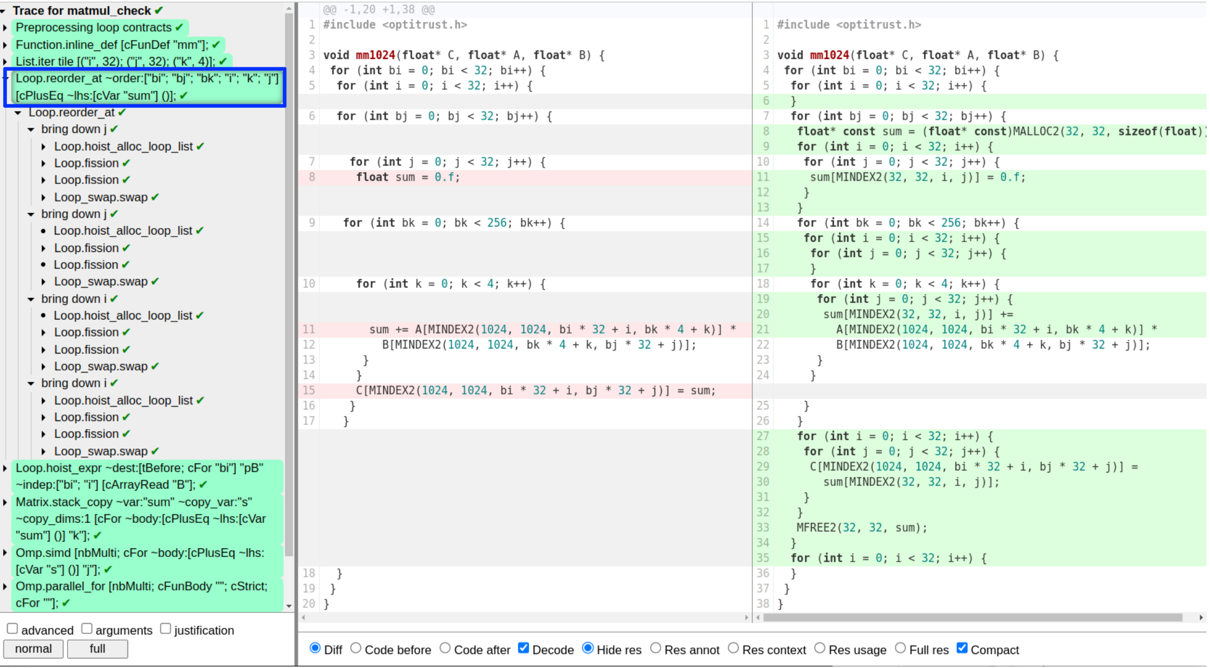
- validated through static resource analysis based on separation logic

Visualizing the Effect of Transformations

Pressing “F6” on a transformation step opens the corresponding diff:

```
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;
    }
  }
```



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

Takeaway

Accessibility, interactivity and feedback matter.

- familiar C code rather than specialized language / IR
- interactive visualization of intermediate steps (diffs, traces)
  - reversible encoding from C to internal imperative λ-calculus
- no black box code generation: "what you see is what you get"

- reasonably concise scripts thanks to composability
  - Matmul: 8 script steps result in 55 basic transformations (+61 "ghost transformations").
  - names are very useful: eases composition as well as targeting and marking subterms.
  - easier to define abstract loop transformations on C

Takeaway

Much remains to be done.

- only subset of C supported
- interactivity and feedback remains basic
- no search automation
- ...

A Case for Interactive Optimization Assistants

1. **Extensibility, composability and control matter.**
   - yet, Premature control is the root of all evil.
2. **Semi-automation enables parsimonious control.**
   - yet, Parsimonious control is not enough for productivity.
3. **Accessibility, interactivity and feedback matter.**
   - yet, Much remains to be done.

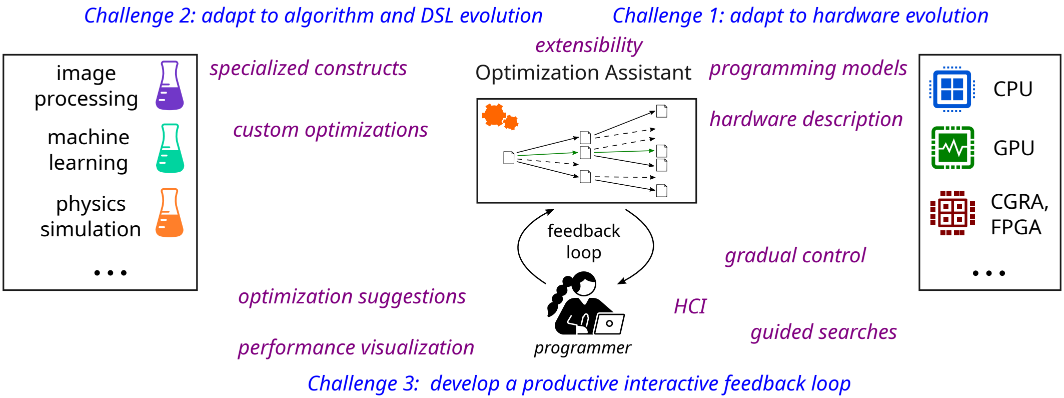
Optimization Assistants: Towards Mainstream User-Scheduling

Mainstream Development Tools:

- Debuggers
- Profilers
- AI Assistants (GitHub Copilot)
- **Optimization Assistants?** *(aka. User-Scheduling++)*
- *Proof Assistants?*

Improve the traditional (profile; manual rewrite; debug) cycle.

A Few Technical Challenges



A Few Community Challenges

The user-scheduling community could benefit from sharing:

1. benchmarks
2. evaluation methodologies
3. software
4. terminology definitions?

A Few Community Challenges

The user-scheduling community could benefit from sharing:

1. benchmarks
2. evaluation methodologies
3. software
4. terminology definitions?

Thanks! I am curious to see where this workshop leads us. thok.eu

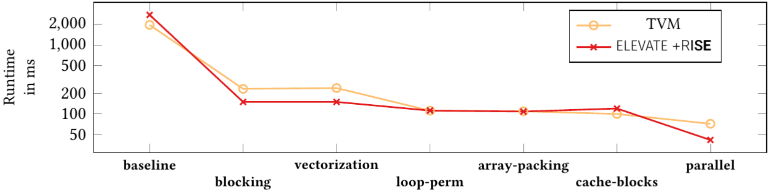
Backup Slides

1st Benchmark: Matrix Multiplication on Intel CPU

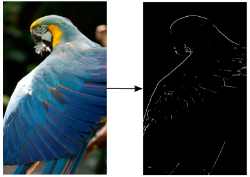
- 6 optimizations
  - transform loops *blocking, permutation, unrolling*
  - change data layout *array packing*
  - add parallelism *vectorization, multi-threading*

- performance is on par with reference schedules from TVM.

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



2^{nd} 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

OptiTrust 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