Interactive Source-to-Source Optimizations Validated using Static Resource Analysis

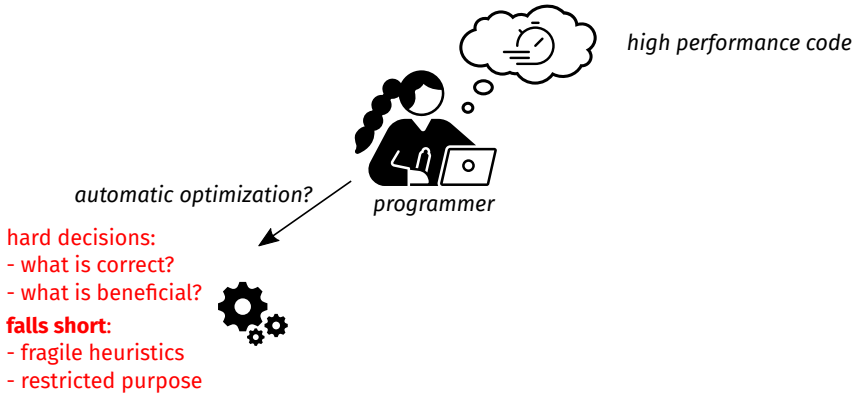
(OptiTrust)

Guillaume Bertholon Arthur Charguéraud Thomas Kœhler
Begatim Bytyqi Damien Rouhling

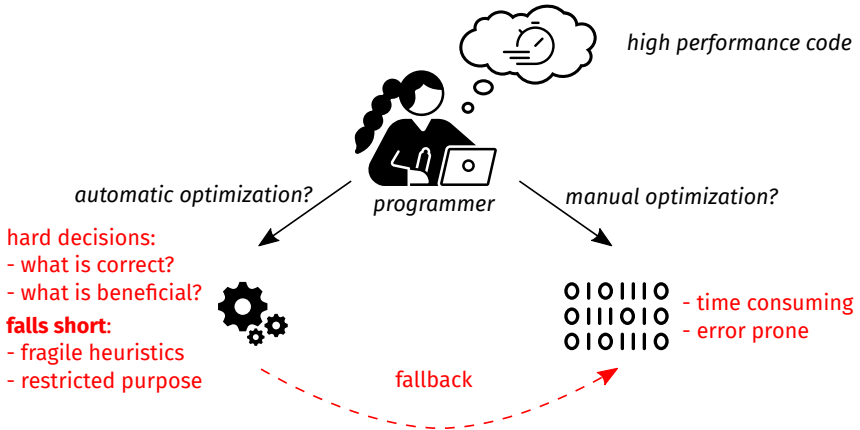
Inria

SOAP Workshop — Copenhagen, June 2024

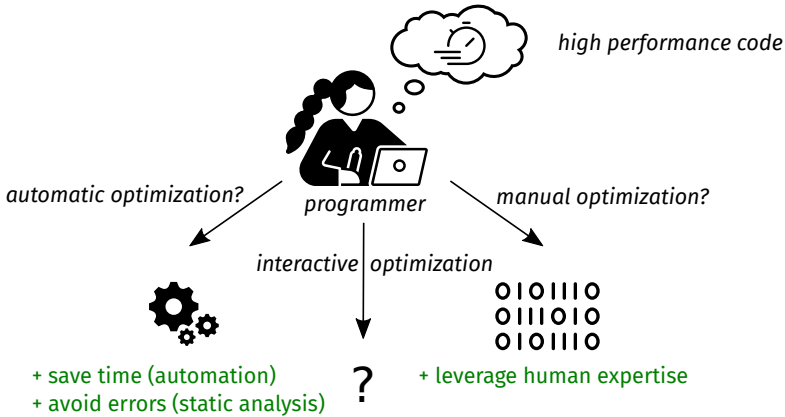
Producing High-Performance Code is Hard



Producing High-Performance Code is Hard



Producing High-Performance Code is Hard



The OptiTrust Interactive Optimization Framework

Users direct C code optimizations, OptiTrust checks the correctness of the optimizations

Case studies:

- **Matrix Multiplication reproducing TVM baseline** *linear algebra*
  loop and data layout transformations, OpenMP parallelism
- Box Filter (on rows) reproducing handwritten OpenCV code *image processing*
  multi-versioning, algorithmic sliding window optimization

Work in progress:

- Harris Corner Detection reproducing Halide baseline *image processing*
  uneven tiling, operator fusion, circular buffers, CSE
- Particle-In-Cell reproducing handwritten code *physics simulation*
  AoS ↔ SoA, transforming values in memory, atomic bags

Hand Optimizing Matrix Multiplication

Naive C code:

```
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;
  }
}
```

Optimized C code:

```
float* pB = (float*) malloc(sizeof(float[32][256][4][32]));
#pragma omp parallel for
for (int bj = 0; bj < 32; bj++)
 for (int bk = 0; bk < 256; bk++)
  for (int k = 0; k < 4; k++)
   for (int j = 0; j < 32; j++)
    pB[32768 * bj + 128 * bk + 32 * k + j] = B[1024 * (4 * bk + k) + 32 * bj + j];
#pragma omp parallel for
for (int bi = 0; bi < 32; bi++) {
 for (int bj = 0; bj < 32; bj++) {
 float* sum = (float*) malloc(sizeof(float[32][32]));
 for (int i = 0; i <
  for (int j = 0; j
   sum[32 * i + j] =
 for (int bk = 0; bk
  for (int i = 0; i <
   float s[32];
   memcpy(s, &sum[32
#pragma omp simd
   for (int j = 0; j
    s[j] += A[1024 *                                    2 * 0 + j];
#pragma omp simd
   for (int j = 0; j
    s[j] += A[1024 *                                    2 * 1 + j];
#pragma omp simd
   for (int j = 0; j < 32; j++)
    s[j] += A[1024 * (32 * bi + i) + 4 * bk + 2] * pB[32768 * bj + 128 * bk + 32 * 2 + j];
#pragma omp simd
   for (int j = 0; j < 32; j++)
    s[j] += A[1024 * (32 * bi + i) + 4 * bk + 3] * pB[32768 * bj + 128 * bk + 32 * 3 + j];
   memcpy(&sum[32 * i], s, sizeof(float[32]));
 } }
 for (int i = 0; i < 32; i++)
  for (int j = 0; j < 32; j++)
   C[1024 * (32*bi + i) + 32*bj + j] = sum[32*i + j];
 free(sum);
}
free(pB);
```

150× faster
7× more characters
time consuming
error prone

Hand Optimizing Matrix Multiplication

Naive C code:

```
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;
  }
}
```

Optimized C code:

```
float* pB = (float*) malloc(sizeof(float[32][256][4][32]));
#pragma omp parallel for
for (int bj = 0; bj < 32; bj++)
 for (int bk = 0; bk < 256; bk++)
  for (int k = 0; k < 4; k++)
   for (int j = 0; j < 32; j++)
    pB[32768 * bj + 128 * bk + 32 * k + j] = B[1024 * (4 * bk + k) + 32 * bj + j];
#pragma omp parallel for
for (int bi = 0
 for (int bj =
 float* sum = (
 for (int i = 0
  for (int j =
   sum[32 * i
 for (int bk =
  for (int i =
   float s[32]
   memcpy(s, 0
#pragma omp sim
   for (int j
    s[j] += A
#pragma omp sim
   for (int j
    s[j] += A
#pragma omp sim
   for (int j
    s[j] += A
#pragma omp sim
   for (int j = 0; j < 32; j++)
    s[j] += A[1024 * (32 * bi + i) + 4 * bk + 3] * pB[32768 * bj + 128 * bk + 32 * 3 + j];
   memcpy(&sum[32 * i], s, sizeof(float[32]));
 } }
 for (int i = 0; i < 32; i++)
  for (int j = 0; j < 32; j++)
   C[1024 * (32*bi + i) + 32*bj + j] = sum[32*i + j];
 free(sum);
}
free(pB);
```

- specialized input sizes
  - $m = n = p = 1024$
- improved data locality
  - *transformed loops and data layout*
- parallel
  - *vectorization, multi-threading*

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 =
  l
  s
      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
i2,
(kax
bk,
s[CC
s[CC
s[CC
s[C].parallel(bi)
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

Optimizing MatMul 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"];
```

Optimizing MatMul 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"];
```

Optimizing MatMul 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
  for (int i = 0; i < m; i++
    __xmodifies("for j in 0.
    for (int j = 0; j < n; j+
      __xmodifies("&C[i][j] ~
      float sum = 0.0f;
      for (int k = 0; k < p;
        sum += A[i][k] * B[k][
      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
                    ("j", 32); ("k", 4)];

                     "i"; "k"; "j"] [cPlusEq ()];
                    fore; cFor "bi"] "pB"
                    yRead "B"];
                    n" ~copy_var:"s"
                    [cPlusEq ()] "k"];
                    sEq ()] "j"];
                     "mm1024"; cStrict; cFor ""];
                    PlusEq ()] "k"];
```

source-to-source transformations

validated using static resource analysis

based on separation logic

no black box code generation

readable intermediate steps

easier to compose and extend

Resource Analysis in MatMul

user-provided function contracts

```
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: $\star_{i \in 0..m} \star_{j \in 0..n} \left(\texttt{\&C[i][j]} \rightsquigarrow \texttt{Cell}\right)$

Resource Analysis in MatMul

user-provided partial loop contracts

```
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;
  }
}
```

Resource Analysis in MatMul

automatically inferred annotations

```
for (int i = 0; i < m; i++) {
  __xmodifies("for j in 0..n -> &C[i][j] ⇝ Cell");
  __sreads("A ⇝ Matrix2(m, p), B ⇝ Matrix2(p, n)");
  for (int j = 0; j < n; j++) {
    __xmodifies("&C[i][j] ⇝ Cell");
    __sreads("A ⇝ Matrix2(m, p), B ⇝ Matrix2(p, n)");
    float sum = 0.0f;
    for (int k = 0; k < p; k++) {
      __smodifies("&sum ⇝ Cell");
      __sreads("A ⇝ Matrix2(m, p), B ⇝ Matrix2(p, n)");
      __ghost(matrix2_ro_focus, "A, i, k");
      __ghost(matrix2_ro_focus, "B, k, j");
      sum += A[i][k] * B[k][j];
      __ghost(matrix2_ro_unfocus, "A");
      __ghost(matrix2_ro_unfocus, "B");
    }
    C[i][j] = sum;
  }
}
```

Resource Analysis in MatMul

automatically computed resources at every program point

$R_1 : \alpha_1(\texttt{A} \rightsquigarrow \texttt{Matrix2(m, p)}) \star R_2 : \alpha_2(\texttt{B} \rightsquigarrow \texttt{Matrix2(p, n)}) \star$
$R_3 : \star\, i \in 0..m \star j \in 0..n$ `&C[i][j]` $\rightsquigarrow$ `Cell`

```
for (int i = 0; i < m; i++) {
```
$R_4 : (\alpha_4/m)(\texttt{A} \rightsquigarrow \texttt{Matrix2(m, p)}) \star R_5 : (\alpha_5/m)(\texttt{B} \rightsquigarrow \texttt{Matrix2(p, n)}) \star$
$R_6 : \star\, j \in 0..n$ `&C[i][j]` $\rightsquigarrow$ `Cell`
```
  for (int j = 0; j < n; j++) {
```
$R_7 : \cdots \star R_8 : \cdots \star$
$R_9 : \texttt{\&C[i][j]} \rightsquigarrow \texttt{Cell}$
```
    float sum = 0.f;
```
$R_7 \star R_8 \star R_9 \star R_{10} : \texttt{\&sum} \rightsquigarrow \texttt{Cell}$
```
    for (int k = 0; k < p; k++) { /* [...] */ }
```
$R_7 \star R_8 \star R_9 \star R_{11} : \texttt{\&sum} \rightsquigarrow \texttt{Cell}$
```
    C[i][j] = sum;
```
$R_7 \star R_8 \star R_{11} \star R_{12} : \texttt{\&C[i][j]} \rightsquigarrow \texttt{Cell}$
```
  }
```
$R_4 \star R_5 \star R_{13} : \star\, j \in 0..n$ `&C[i][j]` $\rightsquigarrow$ `Cell`
```
}
```
$R_1 \star R_2 \star R_{14} : \star\, i \in 0..m \star j \in 0..n$ `&C[i][j]` $\rightsquigarrow$ `Cell`

Resource Analysis in MatMul

automatically computed resources at every program point

$R_1 : \alpha_1$(A $\rightsquigarrow$ Matrix2(m, p)) $\star$ $R_2 : \alpha_2$(B $\rightsquigarrow$ Matrix2(p, n)) $\star$
$R_3 : \star\, i \in 0..m \star j \in 0..n$ &C[i][j] $\rightsquigarrow$ Cell

```
for (int i = 0; i < m; i++) {
```
$R_4 : (\alpha_4/m)$(A $\rightsquigarrow$ Matrix2(m, p)) $\star$ $R_5 : (\alpha_5/m)$(B $\rightsquigarrow$ Matrix2(p, n)) $\star$
$R_6 : \star\, j \in 0..n$ &C[i][j] $\rightsquigarrow$ Cell
```
  for (int j = 0; j < n; j++) {
```
$R_7 : \cdots \star R_8 : \cdots \star$
R_9 : &C[i][j] $\rightsquigarrow$ Cell
```
    float sum = 0.f;
```
$R_7 \star R_8 \star R_9 \star R_{10}$: &sum $\rightsquigarrow$ Cell
```
    for (int k = 0; k < p; k++) { /* [...] */ }
```
$R_7 \star R_8 \star R_9 \star R_{11}$: &sum $\rightsquigarrow$ Cell
```
    C[i][j] = sum;
```
$R_7 \star R_8 \star R_{11} \star R_{12}$: &C[i][j] $\rightsquigarrow$ Cell
```
  }
```
$R_4 \star R_5 \star R_{13} : \star\, j \in 0..n$ &C[i][j] $\rightsquigarrow$ Cell
```
}
```
$R_1 \star R_2 \star R_{14} : \star\, i \in 0..m \star j \in 0..n$ &C[i][j] $\rightsquigarrow$ Cell

Combined Transformations on MatMul

Combined transformations:

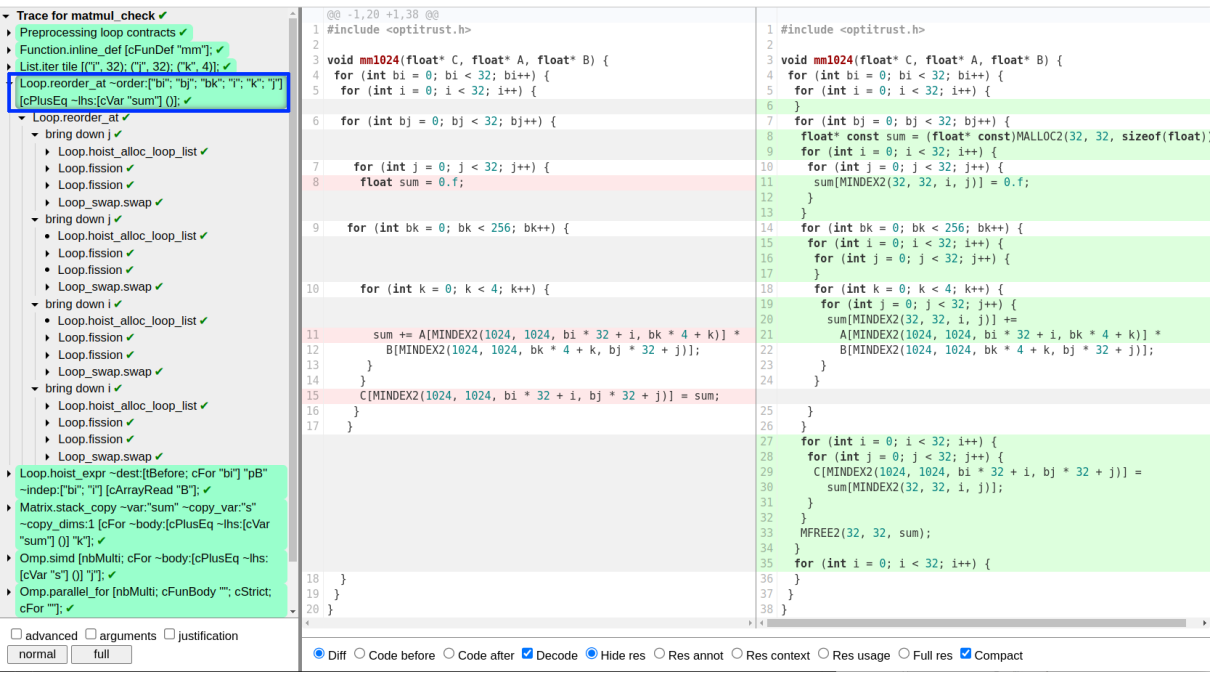
- compose basic transformations
- are valid by composition

MatMul: 8 script steps result in 55 basic transformations (+61 ghost transformations).

Zoom on the 3rd transformation step:

```
!! Loop.reorder_at ~order:["bi"; "bj"; "bk"; "i"; "k"; "j"] [cPlusEq ()];
```

- 18 basic transformations
  - 4 `Loop.swap`
  - 6 `Loop.fission`
  - 2 `Loop.hoist`
  - .. 6 more
- 32 ghost transformations


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 ""]; ✔
@@ -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++) {
}
}
}
advanced
arguments
justification
normal
full
Diff
Code before
Code after
Decode
Hide res
Res annot
Res context
Res usage
Full res
Compact

Basic Transformations

Basic transformations:

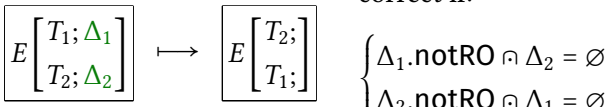
- transform the AST directly

  internally, we transform an imperative lambda calculus AST with bijective encoding from C

- check for sufficient correctness conditions

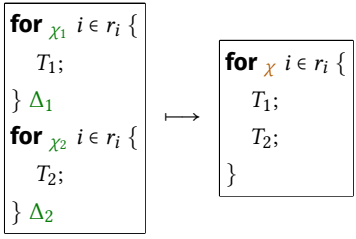
- `Instr`: `move`, `delete`, `insert`, `duplicate`
- `Function`: `inline`
- `Variable`: `inline`, `bind`, `to_const`, `init_detach`, `local_name`
- `Loop`: `fusion`, `fission`, `swap`, `hoist`, `move_out`, `tile`, `collapse`, `unroll`
- `Matrix`: `intro_malloc0`,
- `Omp`: `parallel_for`, `simd`
- `Arith`: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

Correctness Criteria for `Instr.move`



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

Correctness Criteria for `Loop.fusion`



correct if:

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

with:

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

Conclusion

- OptiTrust is an interactive framework for optimizing general-purpose C code
- Transformations are validated using a static resource analysis
- 2 completed case studies, we are working on more: suggest your own!
- Future Work:
  - support more C features and other languages

    function pointers, union types, switch, variable length arrays, abrupt termination and jumps
  - increase expressiveness towards full separation logic
  - make it easy for users to implement new transformations
  - extensions to guide users towards beneficial transformations

Conclusion

- OptiTrust is an interactive framework for optimizing general-purpose C code
- Transformations are validated using a static resource analysis
- 2 completed case studies, we are working on more: suggest your own!
- Future Work:
  - support more C features and other languages
    function pointers, union types, switch, variable length arrays, abrupt termination and jumps
  - increase expressiveness towards full separation logic
  - make it easy for users to implement new transformations
  - extensions to guide users towards beneficial transformations

---

Thanks,
we are open to collaboration!

github.com/charguer/optitrust
charglueraud.org/softs/optitrust/traces/

Backup Slides

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