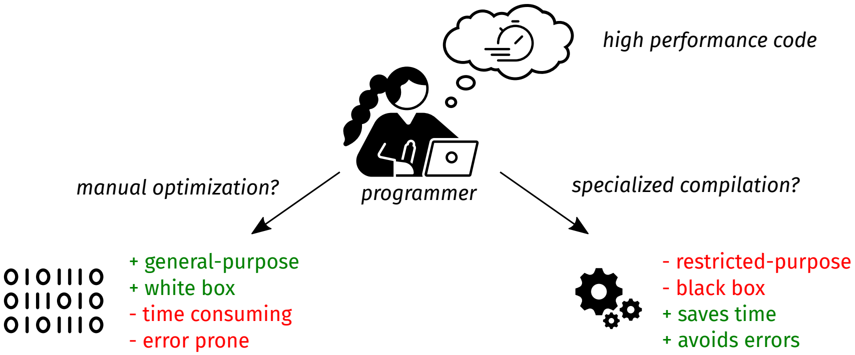
OptiTrust: an Interactive Optimization Framework

Thomas Kœhler   Arthur Charguéraud   Begatim Bytyqi
Damien Rouhling   Yann Barsamian

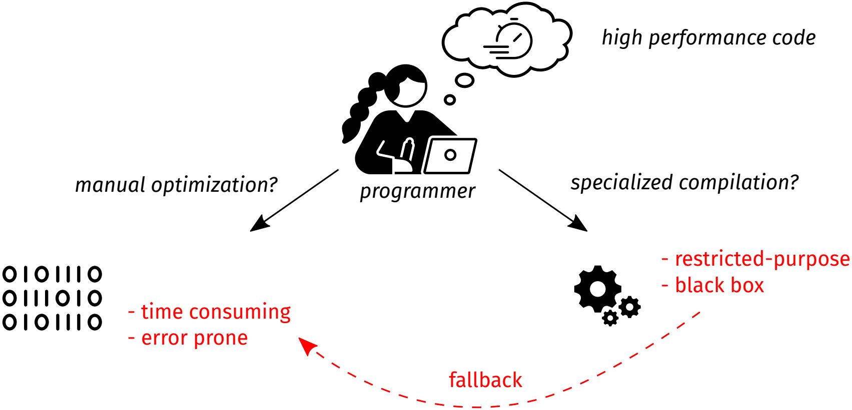
Inria

ARRAY Workshop — Orlando, June 2023

Why OptiTrust?



Why OptiTrust?



Why OptiTrust?

high performance code

programmer

+ general-purpose
+ white box

optimization framework

OptiTrust

+ saves time
+ avoids errors

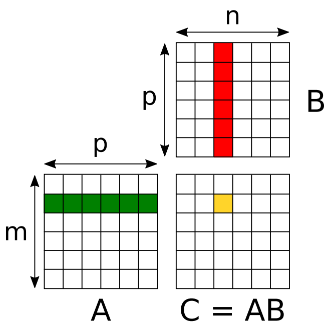
Example: Optimizing Matrix Multiplication

A standard benchmark to:

- showcase OptiTrust user experience
- compare to user-guided specialized compilers *(TVM)*

Unoptimized Matrix Multiplication

```
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;
  }
}
```



Optimization by Hand

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

- Standard optimizations:
  - improve data locality

    transform loops, change data layout
  - add parallelism

    vectorization, multi-threading
- 150× faster

  Intel i7-8665U, 4 cores, AVX2
- Time consuming
- Error prone
- 5× more lines of code

Optimization with TVM

TVM Algorithm = what to compute

```
k = tvm.reduce_axis((0, P))
A = tvm.placeholder((M, P))
B = tvm.placeholder((P, N))

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

Rewritten Algorithm

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

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

TVM Schedule = how to compute

```
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)
```

- Convenient *quickly try many schedules*
- Restricted *algorithm DSL + schedule API*
- Black box *implicit heuristics + unfamiliar IR*

Optimization with OptiTrust

OptiTrust Transformation Script

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

- transformation script = sequence of transformation steps (`!!`)
- transformation = function modifying the current program
- *target* = data structure describing where to apply transformations
- scripts are written in OCaml: **`let`**, `List.iter`, ..

Interacting with OptiTrust

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

```
 void mm(float* C, float* A, float* B, int m, int n, int p) {
+  for (int bi = 0; bi < exact_div(m, 32); bi++) {
+    for (int i = 0; i < 32; i++) {
+      for (int bj = 0; bj < exact_div(n, 32); bj++) {
+        for (int j = 0; j < 32; j++) {
           float sum = 0.f;
+          for (int bk = 0; bk < exact_div(p, 4); bk++) {
+            for (int k = 0; k < 4; k++) {
+              sum += A[bi * 32 + i][bk * 4 + k] * B[bk * 4 + k][b
+            }
+          }
+          C[bi * 32 + i][bj * 32 + j] = sum;
+        }
       }
     }
   }
 }
```

- pressing "F6" on a transformation step opens the corresponding diff. *first step above*
- pressing "Maj+F5" opens a trace of all transformations. *next slide*

SmallSteps: next 1 2 3 4 5 6 7 8 9

all ☑ details ☐ ast-before ☐ ast-after ☑ compact ☐ justification ☐ exectime ☐ io-steps ☐ target-steps ☐ noop-steps ☐ atomic-substeps ☐ basic-modules

```
!!! Loop.hoist_expr ~dest:[tBefore; cFor "bi"] "pB" ~indep:["bi"; "i"] [cArrayRead "B"];
```

- Variable.bind
- Loop.hoist_alloc
  - Variable.init_detach
  - Matrix.intro_malloc0
  - Loop.hoist
  - Loop.hoist
  - Loop.move_out
  - Instr.move
  - Loop.hoist
  - Loop.hoist
  - Loop.move_out
  - Instr.move
  - Variable.to_const
  - Variable.inline
  - Variable.to_const
  - Variable.inline
  - Variable.to_const
  - Variable.inline
  - Variable.to_const
  - Variable.inline
  - Variable.rename
- Loop.hoist_instr
  - Loop.fission
  - Loop.fission
  - Loop.move_out
  - Loop.fission
  - Instr.move
  - Loop.fission
  - Loop.move_out

```
#include "../../include/optitrust.h"
#include "matmul.h"
#include "omp.h"
// NOTE: using pretty matrix notation

void mm(float* C, float* A, float* B, int m, int n, int p) {
 for (int bi = 0; bi < exact_div(m, 32); bi++) {
  for (int bj = 0; bj < exact_div(n, 32); bj++) {
   float* sum = (float*)malloc(sizeof(float[32][32]));
   for (int i = 0; i < 32; i++) {
    for (int j = 0; j < 32; j++) {
     sum[i][j] = 0.f;
    }
   }
   for (int bk = 0; bk < exact_div(p, 4); bk++) {
    for (int i = 0; i < 32; i++) {
     for (int k = 0; k < 4; k++) {
      for (int j = 0; j < 32; j++) {
-       sum[i][j] +=
-         A[bi * 32 + i][bk * 4 + k] * B[bk * 4 + k][bj * 32 + j];
      }
     }
    }
   }
   for (int i = 0; i < 32; i++) {
    for (int j = 0; j < 32; j++) {
     C[bi * 32 + i][bj * 32 + j] = sum[i][j];
    }
   }
   free(sum);
```

```
#include "../../include/optitrust.h"
#include "matmul.h"
#include "omp.h"
// NOTE: using pretty matrix notation

void mm(float* C, float* A, float* B, int m, int n, int p) {
+ float* pB =
+   (float*)malloc(sizeof(float[exact_div(n, 32)][exact_div(p, 4)][4][32]));
+ for (int bj = 0; bj < exact_div(n, 32); bj++) {
+  for (int bk = 0; bk < exact_div(p, 4); bk++) {
+   for (int k = 0; k < 4; k++) {
+    for (int j = 0; j < 32; j++) {
+     pB[bj][bk][k][j] = B[bk * 4 + k][bj * 32 + j];
+    }
+   }
+  }
+ }
 for (int bi = 0; bi < exact_div(m, 32); bi++) {
  for (int bj = 0; bj < exact_div(n, 32); bj++) {
   float* sum = (float*)malloc(sizeof(float[32][32]));
   for (int i = 0; i < 32; i++) {
    for (int j = 0; j < 32; j++) {
     sum[i][j] = 0.f;
    }
   }
   for (int bk = 0; bk < exact_div(p, 4); bk++) {
    for (int i = 0; i < 32; i++) {
     for (int k = 0; k < 4; k++) {
      for (int j = 0; j < 32; j++) {
+       sum[i][j] += A[bi * 32 + i][bk * 4 + k] * pB[bj][bk][k][j];
      }
     }
    }
   }
   for (int i = 0; i < 32; i++) {
    for (int j = 0; j < 32; j++) {
     C[bi * 32 + i][bj * 32 + j] = sum[i][j];
    }
   }
   free(sum);
```

SmallSteps: next 1 2 3 4 5 6 7 8 9

all ☑ details ☐ ast-before ☐ ast-after ☑ compact ☐ justification ☐ exectime ☐ io-steps ☐ target-steps ☐ noop-steps ☐ atomic-substeps ☐ basic-modules

```
!!! Loop.hoist_expr ~dest:[tBefore; cFor "bi"] "pB" ~indep:["bi"; "i"] [cArrayRead "B"];
```

- Variable.bind
- Loop.hoist_alloc
  - Variable.init_detach
  - Matrix.intro_malloc0
  - Loop.hoist
  - Loop.hoist
  - Loop.move_out
  - Instr.move
  - Loop.hoist
  - Loop.hoist
  - Loop.move_out
  - Instr.move
  - Variable.to_const
  - Variable.inline
  - Variable.to_const
  - Variable.inline
  - Variable.to_const
  - Variable.inline
  - Variable.to_const
  - Variable.inline
  - Variable.rename
- Loop.hoist_instr
  - Loop.fission
  - Loop.fission
  - Loop.move_out
  - Loop.fission
  - Instr.move
  - Loop.fission
  - Loop.move_out

```
    float* sum = (float*)malloc(sizeof(float[32][32]));
    for (int i = 0; i < 32; i++) {
     for (int j = 0; j < 32; j++) {
      sum[i][j] = 0.f;
     }
    }
    for (int bk = 0; bk < exact_div(p, 4); bk++) {
     for (int i = 0; i < 32; i++) {
      for (int k = 0; k < 4; k++) {
       for (int j = 0; j < 32; j++) {
-       sum[i][j] +=
-         A[bi * 32 + i][bk * 4 + k] * B[bk * 4 + k][bj * 32 + j];
       }
      }
     }
    }
    for (int i = 0; i < 32; i++) {
     for (int j = 0; j < 32; j++) {
      C[bi * 32 + i][bj * 32 + j] = sum[i][j];
     }
    }
    free(sum);
```

```
    float* sum = (float*)malloc(sizeof(float[32][32]));
    for (int i = 0; i < 32; i++) {
     for (int j = 0; j < 32; j++) {
      sum[i][j] = 0.f;
     }
    }
    for (int bk = 0; bk < exact_div(p, 4); bk++) {
     for (int i = 0; i < 32; i++) {
      for (int k = 0; k < 4; k++) {
       for (int j = 0; j < 32; j++) {
+        float pB = B[bk * 4 + k][bj * 32 + j];
+        sum[i][j] += A[bi * 32 + i][bk * 4 + k] * pB;
       }
      }
     }
    }
    for (int i = 0; i < 32; i++) {
     for (int j = 0; j < 32; j++) {
      C[bi * 32 + i][bj * 32 + j] = sum[i][j];
     }
    }
    free(sum);
```

SmallSteps: next 1 2 3 4 5 6 7 8 9

all ☑ details ☐ ast-before ☐ ast-after ☑ compact ☐ justification ☐ exectime ☐ io-steps ☐ target-steps ☐ noop-steps ☐ atomic-substeps ☐ basic-modules

```
!!! Loop.hoist_expr ~dest:[tBefore; cFor "bi"] "pB" ~indep:["bi"; "i"] [cArrayRead "B"];
```

- Variable.bind
- Loop.hoist_alloc
  - Variable.init_detach
  - Matrix.intro_malloc0
  - Loop.hoist
  - Loop.hoist
  - Loop.move_out
  - Instr.move
  - Loop.hoist
  - Loop.hoist
  - Loop.move_out
  - Instr.move
  - Variable.to_const
  - Variable.inline
  - Variable.to_const
  - Variable.inline
  - Variable.to_const
  - Variable.inline
  - Variable.to_const
  - Variable.inline
  - Variable.rename
- Loop.hoist_instr
  - Loop.fission
  - Loop.fission
  - Loop.move_out
  - Loop.fission
  - Instr.move
  - Loop.fission
  - Loop.move_out

```
#include "../../include/optitrust.h"
#include "matmul.h"
#include "omp.h"
// NOTE: using pretty matrix notation

void mm(float* C, float* A, float* B, int m, int n, int p) {
 for (int bi = 0; bi < exact_div(m, 32); bi++) {
  for (int bj = 0; bj < exact_div(n, 32); bj++) {
   float* sum = (float*)malloc(sizeof(float[32][32]));
   for (int i = 0; i < 32; i++) {
    for (int j = 0; j < 32; j++) {
     sum[i][j] = 0.f;
    }
   }
   for (int bk = 0; bk < exact_div(p, 4); bk++) {
    for (int i = 0; i < 32; i++) {
     for (int k = 0; k < 4; k++) {
      for (int j = 0; j < 32; j++) {
-       float pB = B[bk * 4 + k][bj * 32 + j];
-       sum[i][j] += A[bi * 32 + i][bk * 4 + k] * pB;
      }
     }
    }
   }
   for (int i = 0; i < 32; i++) {
    for (int j = 0; j < 32; j++) {
     C[bi * 32 + i][bj * 32 + j] = sum[i][j];
    }
   }
   free(sum);
  }
 }
}

void mm1024(float* C, float* A, float* B) { mm(C, A, B, 1024, 1024, 1024); }
```

```
#include "../../include/optitrust.h"
#include "matmul.h"
#include "omp.h"
// NOTE: using pretty matrix notation

void mm(float* C, float* A, float* B, int m, int n, int p) {
+ float* pB =
+   (float*)malloc(sizeof(float[exact_div(n, 32)][exact_div(p, 4)][4][32]));
 for (int bi = 0; bi < exact_div(m, 32); bi++) {
  for (int bj = 0; bj < exact_div(n, 32); bj++) {
   float* sum = (float*)malloc(sizeof(float[32][32]));
   for (int i = 0; i < 32; i++) {
    for (int j = 0; j < 32; j++) {
     sum[i][j] = 0.f;
    }
   }
   for (int bk = 0; bk < exact_div(p, 4); bk++) {
    for (int i = 0; i < 32; i++) {
     for (int k = 0; k < 4; k++) {
      for (int j = 0; j < 32; j++) {
+       pB[bj][bk][k][j] = B[bk * 4 + k][bj * 32 + j];
+       sum[i][j] += A[bi * 32 + i][bk * 4 + k] * pB[bj][bk][k][j];
      }
     }
    }
   }
   for (int i = 0; i < 32; i++) {
    for (int j = 0; j < 32; j++) {
     C[bi * 32 + i][bj * 32 + j] = sum[i][j];
    }
   }
   free(sum);
  }
 }
+ free(pB);
}

void mm1024(float* C, float* A, float* B) { mm(C, A, B, 1024, 1024, 1024); }
```

SmallSteps: next 1 2 3 4 5 6 7 8 9

all ☑ details ☐ ast-before ☐ ast-after ☑ compact ☐ justification ☐ exectime ☐ io-steps ☐ target-steps ☐ noop-steps ☐ atomic-substeps ☐ basic-modules

```
!!! Loop.hoist_expr ~dest:[tBefore; cFor "bi"] "pB" ~indep:["bi"; "i"] [cArrayRead "B"];
```

- Variable.bind
- Loop.hoist_alloc
  - Variable.init_detach
  - Matrix.intro_malloc0
  - Loop.hoist
  - Loop.hoist
  - Loop.move_out
  - Instr.move
  - Loop.hoist
  - Loop.hoist
  - Loop.move_out
  - Instr.move
  - Variable.to_const
  - Variable.inline
  - Variable.to_const
  - Variable.inline
  - Variable.to_const
  - Variable.inline
  - Variable.to_const
  - Variable.inline
  - Variable.rename
- Loop.hoist_instr
  - Loop.fission
  - Loop.fission
  - Loop.move_out
  - Loop.fission
  - Instr.move
  - Loop.fission
  - Loop.move_out

```
#include "../../include/optitrust.h"
#include "matmul.h"
#include "omp.h"
// NOTE: using pretty matrix notation

void mm(float* C, float* A, float* B, int m, int n, int p) {
 float* pB =
   (float*)malloc(sizeof(float[exact_div(n, 32)][exact_div(p, 4)][4][32]));
 for (int bi = 0; bi < exact_div(m, 32); bi++) {
  for (int bj = 0; bj < exact_div(n, 32); bj++) {
   float* sum = (float*)malloc(sizeof(float[32][32]));
   for (int i = 0; i < 32; i++) {
    for (int j = 0; j < 32; j++) {
     sum[i][j] = 0.f;
    }
   }
   for (int bk = 0; bk < exact_div(p, 4); bk++) {
    for (int i = 0; i < 32; i++) {
     for (int k = 0; k < 4; k++) {
      for (int j = 0; j < 32; j++) {
-      pB[bj][bk][k][j] = B[bk * 4 + k][bj * 32 + j];
       sum[i][j] += A[bi * 32 + i][bk * 4 + k] * pB[bj][bk][k][j];
      }
     }
    }
   }
   for (int i = 0; i < 32; i++) {
    for (int j = 0; j < 32; j++) {
     C[bi * 32 + i][bj * 32 + j] = sum[i][j];
    }
   }
```

```
#include "../../include/optitrust.h"
#include "matmul.h"
#include "omp.h"
// NOTE: using pretty matrix notation

void mm(float* C, float* A, float* B, int m, int n, int p) {
 float* pB =
   (float*)malloc(sizeof(float[exact_div(n, 32)][exact_div(p, 4)][4][32]));
+ for (int bj = 0; bj < exact_div(n, 32); bj++) {
+  for (int bk = 0; bk < exact_div(p, 4); bk++) {
+   for (int k = 0; k < 4; k++) {
+    for (int j = 0; j < 32; j++) {
+     pB[bj][bk][k][j] = B[bk * 4 + k][bj * 32 + j];
+    }
+   }
+  }
+ }
 for (int bi = 0; bi < exact_div(m, 32); bi++) {
  for (int bj = 0; bj < exact_div(n, 32); bj++) {
   float* sum = (float*)malloc(sizeof(float[32][32]));
   for (int i = 0; i < 32; i++) {
    for (int j = 0; j < 32; j++) {
     sum[i][j] = 0.f;
    }
   }
   for (int bk = 0; bk < exact_div(p, 4); bk++) {
    for (int i = 0; i < 32; i++) {
     for (int k = 0; k < 4; k++) {
      for (int j = 0; j < 32; j++) {
       sum[i][j] += A[bi * 32 + i][bk * 4 + k] * pB[bj][bk][k][j];
      }
     }
    }
   }
   for (int i = 0; i < 32; i++) {
    for (int j = 0; j < 32; j++) {
     C[bi * 32 + i][bj * 32 + j] = sum[i][j];
    }
   }
```

Transformations are Composed into Abstractions

9 script steps result in >60 basic steps of >20 basic transformations:

- `Arith`: `simpl`
- `Sequence`: `delete`, `elim`
- `Function`: `inline`
- `Variable`: `inline`, `bind`, `init_detach`
- `Instr`: `move`
- `Array`: `inline_constant`
- `Matrix`: `intro_malloc0`, `copy_to_stack`, `elim_mindex`
- `Loop`: `fission`, `swap`, `hoist`, `move_out`, `tile`, `unroll`
- `Omp`: `parallel_for`, `simd`
- ...

Example Recap

OptiTrust Transformation Script

```
let tile (loop_id, size) = Loop.tile (* ... *) [cFor loop_id] in
!! List.iter tile [("i", 32); ("j", 32); ("k", 4)];
!! Loop.reorder_at ~order:["bi"; "bj"; "bk"; "i"; "k"; "j"]
  [cPlusEq [cVar "sum"]];
!! Loop.hoist_expr ~dest:[tBefore; cFor "bi"] "pB"
  ~indep:["bi"; "i"] [cArrayRead "B"];
!! Function.inline_def [cFunDef "mm"];
!! Matrix.stack_copy ~var:"sum" ~copy_var:"s" ~copy_dims:1
  [cFor ~body:[cPlusEq [cVar "sum"]] "k"];
!! Matrix.elim_mops [];
!! Loop.unroll [cFor ~body:[cPlusEq [cVar "s"]] "k"];
!! Omp.simd [nbMulti; cFor ~body:[cPlusEq [cVar "s"]] "j"];
!! Omp.parallel_for [nbMulti; cFunBody ""; cStrict; cFor ""];
```

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)
```

- achieves the same performance as the TVM schedule
- transforms C code rather than specialized languages / IRs — *general-purpose*
- is interactive, providing diffs and traces of familiar C code — *white box*
- is reasonably concise compared to a specialized API — *still saves time*

Ongoing Work: Justifying Transformation Correctness

- Memory resources are annotated in a subset of Separation Logic:

Initial Matrix Multiplication Annotations

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

- From few annotations, resources are computed at every code location:

Computed Resources for Accumulation Instruction

```
__reads("A => Matrix(m, p); B => Matrix(p, n)");
__modifies("sum => Cell");
sum += A[i][k] * B[k][j];
```

Ongoing Work: Justifying Transformation Correctness

Transformations can:

- Leverage resources to justify their correctness: *avoids errors*

`Omp.parallel_for` is correct when each iteration has separate writes.

```
#pragma omp parallel for
for (int bi = 0; bi < exact_div(m, 32); bi++) {
  __only_this_iteration_modifies("C => MatrixTile ((bi, 32), (0, n))");
```

- Transform annotations to change view on resources:

`Omp.parallel_for ~modifies:"C => MatrixTile((bi, 32), (0, n))"`

```
__ghost("Matrix.tile_dim(C, (0, 32), (0, n))");
// ^ consumes 'C => Matrix(m, n)', produces C tiles
for (int bi = 0; bi < exact_div(m, 32); bi++) {
  __only_this_iteration_modifies("C => MatrixTile((bi, 32), (0, n))");
```

Conclusion

- OptiTrust is an interactive framework for optimizing general-purpose C code
- 3 existing case studies:
  - Matrix Multiplication : same performance as TVM in 9 script steps
  - Harris Corner Detection : same performance as Halide in 11 script steps (`Stencil.`)
  - Particle-In-Cell : beyond specialized compilers in 140 script steps
- Questions for the ARRAY community:
  - Gradual transition between high-level array programming, and interactive optimization?
  - Implement array compilation techniques as transformations that can be reused and applied to hybrid code?

Conclusion

- OptiTrust is an interactive framework for optimizing general-purpose C code
- 3 existing case studies:
  - Matrix Multiplication : same performance as TVM in 9 script steps
  - Harris Corner Detection : same performance as Halide in 11 script steps (`Stencil.`)
  - Particle-In-Cell : beyond specialized compilers in 140 script steps
- Questions for the ARRAY community:
  - Gradual transition between high-level array programming, and interactive optimization?
  - Implement array compilation techniques as transformations that can be reused and applied to hybrid code?

---

thok.eu

Thanks!

optitrust.inria.fr
» /traces/matmul.html
» /traces/harris.html

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

Harris Script

OptiTrust transformation script for Harris corner detection

```
let fuse (ops, overlaps, outputs) =
 Stencil.fusion_targets ~nest_of:2 ~outputs ~overlaps [ctx; any cFun ops] in
let overlaps_2x2 vars = List.map (fun i -> i, [int 2; int 2]) vars in
!! List.iter fuse
 [["grayscale"], [], ["gray"];
  ["sobelX"; "sobelY"], [], ["ix"; "iy"];
  ["mul"; "sum3x3"; "coarsity"], overlaps_2x2 ["ixx"; "ixy"; "iyy"], ["out"]];
!! Stencil.fusion_targets_tile [int 32] ~outputs:["out"]
  ~overlaps:["gray", [int 4]; "ix", [int 2]]
  [ctx; nbMulti; cFor "y"];
!! simpl_mins [ctx];
!! Matrix.storage_folding ~dim:0 ~size:(int 4) [ctx; multi cVarDef ["gray"; "ix"; "iy"]];
!! Matrix.elim [ctx; multi cVarDef ["ixx"; "ixy"; "iyy"]];
let inline v = Matrix.inline_constant ~simpl ~decl:[cVarDef v] [ctx; nbMulti; cArrayRead v] in
!! List.iter inline ["weights_sobelX"; "weights_sobelY"; "weights_sum3x3"];
let bind_gradient name =
 Variable.bind_syntactic ~dest:[ctx; tBefore; cVarDef "acc_sxx"] ~fresh_name:(name ^ "${occ}") [ctx;
      cArrayRead name] in
!! List.iter bind_gradient ["ix"; "iy"];
!! Matrix.elim_mops [ctx];
!! Omp.parallel_for [ctx; cFor "y"];
!! Omp.simd ~clause:[Simdlen 8] [ctx; nbMulti; cFor "x"];
```

Expression Hoisting Transformation

`Loop.hoist_expr` is defined by combining simpler transformations.

```
let%transfo hoist_expr (* ... *) (tg : target) : unit =
 (* .. calls hoist_expr_loop_list .. *)
let%transfo hoist_expr_loop_list (* ... *) (tg : target) : unit =
 Target.iter (fun t p ->
  let instr_path = find_surrounding_instr p t in
  Variable.bind name (target_of_path p);
  hoist_decl_loop_list loops (target_of_path instr_path)) tg

let%transfo hoist_decl_loop_list (* ... *) (tg : target) : unit =
 (* .. calls hoist_alloc_loop_list .. *)
 (* .. calls hoist_instr_loop_list .. *)

let%transfo hoist_alloc_loop_list (* ... *) (tg : target) : unit =
 (* .. calls Variable.init_detach, Matrix.intro_malloc0, Instr.move,
     Loop.move_out, Loop.hoist .. *)

let%transfo hoist_instr_loop_list (* ... *) (tg : target) : unit =
 (* .. calls Instr.move, Loop.move_out, Loop.fission .. *)
```