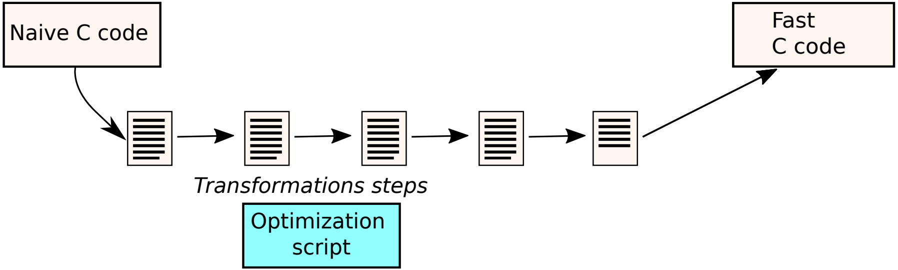
OptiTrust: Producing Trustworthy High-Performance Code via Source-to-Source Transformations

Thomas Kœhler   Arthur Charguéraud   Guillaume Bertholon

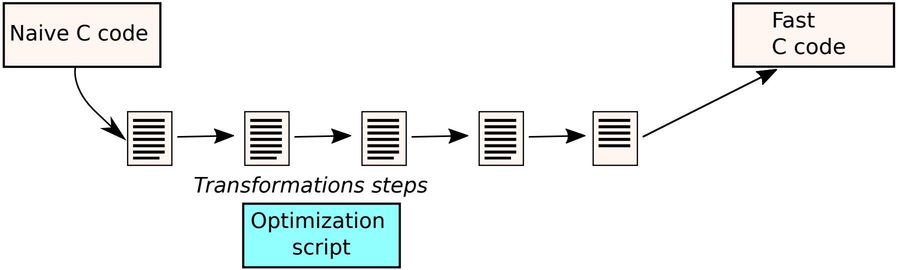
Saarland University, February 2025

The OptiTrust Tool



- optimizations derived by user-guided source-to-source transformations
- extensible and composable library of transformations
- all programs are typechecked using resources (shape-based Separation Logic)
- every transformation checks correctness criteria by leveraging resource usage
- intermediate C code can be visualized and diffed at every step
- bijective encoding into an internal imperative λ-calculus

The OptiTrust Tool



- built towards supporting general-purpose imperative code
- drive global optimizations by human expertise instead of fragile heuristics
- save time and avoid errors while reproducing hand optimized code

Case Studies

Validated Case Studies:

- Row-wise Image Blurring/Denoising: reproducing optimized OpenCV baseline

  baseline: OpenCV optimized library; multi-versioning, algorithmic sliding window optimization
- **Matrix Multiplication**: reproducing optimized TVM baseline

  loop and data layout transformations, OpenMP parallelism
- Particle-In-Cell Plasma Simulation (simplified)

  records, transforming values in memory

In The Pipeline:

- Harris Image Corner/Edge detection: reproducing Halide baseline

  uneven tiling, operator fusion, circular buffers
- Particle-In-Cell Plasma Simulation: reproducing hand optimized baseline

  AoS ↔ SoA, atomic bags

Disclaimer

- restricted language: only a subset of the C language
- restricted logic: only a subset of Separation Logic (shapes)
- limited set of transformations and incomplete correctness criteria

What we also want to explore

- transform verified code
  - full functional correctness specification
  - produce proof certificates
  - link to (verified) compilers
  - support transforming OCaml code
- adapt to the constant evolution of algorithms and hardware
  - extensible support for high-level DSLs
  - extensible support for hardware targets
- develop an interactive feedback loop
  - guided searches (e.g. guided by program sketches)
  - optimization suggestion and vizualization
- reason about numerical accuracy during optimization

DEMO: Matrix Multiplication

RowSum: unoptimized code

Source image `S`, with color components represented using type `T`.
Target image `D`, with color components represented using type `ST`.

$$\mathsf{D}[i] = \sum_{j=i}^{i+\mathsf{kn}} \mathsf{S}[j]$$

```
void rowSum(const int n, const int cn, const int w,
const T S[n+w-1][cn], ST D[n][cn]) {
  for (int i = 0; i < n; i++) { // for each target pixel
    for (int c = 0; c < cn; c++) { // for each channel (e.g., R,G,B)
      ST s = 0;
      for (int k = i; k < i+w; k++) { // for each source pixel
        s += (ST) S[k][c];
      }
      D[i][c] = s;
    }
  }
}
```

RowSum: optimized code

```
void rowSum(const int n, const int cn,
            const int w, const T* S, ST* D) {
  if (w == 3) {
    for (int ic = 0; ic < cn * n; ic++) {
      D[ic] = (ST) S[ic]
            + (ST) S[cn + ic]
            + (ST) S[2 * cn + ic];
    }
  } else if (w == 5) {
    for (int ic = 0; ic < cn * n; ic++) {
      D[ic] = (ST) S[ic]
            + (ST) S[cn + ic]
            + (ST) S[2 * cn + ic]
            + (ST) S[3 * cn + ic]
            + (ST) S[4 * cn + ic];
    }
  } else if (cn == 1) {
    ST s = (ST) 0;
    for (int i = 0; i < w; i++) {
      s += (ST) S[i];
    }
    D[0] = s;
    for (int i = 0; i < n - 1; i++) {
      s += (ST) S[i + w] - (ST) S[i];
      D[i + 1] = s;
    }
  } else if (cn == 3) {
    ST s0 = (ST) 0;
    ST s1 = (ST) 0;
    ST s2 = (ST) 0;
    for (int i = 0; i < 3 * w; i += 3) {
      s0 += (ST) S[i];
      s1 += (ST) S[i + 1];
      s2 += (ST) S[i + 2];
    }
    D[0] = s0;
    D[1] = s1;
    D[2] = s2;
    for (int i = 0; i < 3 * n - 3; i += 3) {
      s0 += (ST) S[3 * w + i] - (ST) S[i];
      s1 += (ST) S[3 * w + i + 1] - (ST) S[i + 1];
      s2 += (ST) S[3 * w + i + 2] - (ST) S[i + 2];
      D[i + 3] = s0;
      D[i + 4] = s1;
      D[i + 5] = s2;
    }
  } else if (cn == 4) {
    // [...] similar to cn == 3, with one more variable
  } else {
    for (int c = 0; c < cn; c++) {
      ST s = (ST) 0;
      for (int i = 0; i < cn * w; i += cn) {
        s += (ST) S[c + i];
      }
      D[c] = s;
      for (int i = c; i < cn * n - cn + c; i += cn) {
        s += (ST) S[cn * w + i] - (ST) S[i];
        D[cn + i] = s;
  } } } }
```

Micro-PIC: unoptimized code

```
for (int idStep = 0; idStep < nbSteps; idStep++) {
 for (int idPart = 0; idPart < nbParticles; idPart++) {
  // Each particle is updated at each time step
  const particle p = particles[idPart];
  // Interpolate the electric field at particle position
  double const (*coeffs)[8];
  cornerInterpolationCoeff(p.pos, coeffs);
  const vect fieldAtPos = matrix_vect_mul(coeffs, fieldAtCorners);
  // Acceleration, new speed and new position
  const vect accel = vect_mul(pCharge / pMass, fieldAtPos);
  const vect speed2 = vect_add(p.speed, vect_mul(deltaT, accel));
  const vect pos2 = vect_add(p.pos, vect_mul(deltaT, speed2));
  // Update the particle
  particles[idPart].speed = speed2;
  particles[idPart].pos = pos2;
} }
```

Micro-PIC: optimized code

```
const double fieldFactor = deltaT * deltaT * pCharge / pMass;
vect* const lFieldAtCorners = (vect*) malloc(nbCorners * sizeof(vect));
for (int i = 0; i < nbCorners; i++) {
  lFieldAtCorners[i].x = fieldAtCorners[i].x * fieldFactor;
  lFieldAtCorners[i].y = fieldAtCorners[i].y * fieldFactor;
  lFieldAtCorners[i].z = fieldAtCorners[i].z * fieldFactor;
}
for (int i = 0; i < nbPart; i++) {
  part[i].speed.x *= deltaT;
  part[i].speed.y *= deltaT;
  part[i].speed.z *= deltaT;
}
double* const coeffs = (double*) malloc(nbCorners * sizeof(double));
for (int idStep = 0; idStep < nbSteps; idStep++) {
  for (int idPart = 0; idPart < nbPart; idPart++) {
    const double rX = part[idPart].pos.x;
    const double rY = part[idPart].pos.y;
    const double rZ = part[idPart].pos.z;
    const double cX = 1. - rX;
    const double cY = 1. - rY;
    const double cZ = 1. - rZ;
    coeffs[0] = cX * cY * cZ;
    coeffs[1] = cX * cY * rZ;
    coeffs[2] = cX * rY * cZ;
    coeffs[3] = cX * rY * rZ;
    coeffs[4] = rX * cY * cZ;
    coeffs[5] = rX * cY * rZ;
    coeffs[6] = rX * rY * cZ;
    coeffs[7] = rX * rY * rZ;
```

```
    double fieldAtPos_x = 0.;
    double fieldAtPos_y = 0.;
    double fieldAtPos_z = 0.;
    for (int k = 0; k < nbCorners; k++) {
      fieldAtPos_x += coeffs[k] * lFieldAtCorners[k].x;
      fieldAtPos_y += coeffs[k] * lFieldAtCorners[k].y;
      fieldAtPos_z += coeffs[k] * lFieldAtCorners[k].z;
    }
    const double speed2_x = part[idPart].speed.x + fieldAtPos_x;
    const double speed2_y = part[idPart].speed.y + fieldAtPos_y;
    const double speed2_z = part[idPart].speed.z + fieldAtPos_z;
    part[idPart].pos.x += speed2_x;
    part[idPart].pos.y += speed2_y;
    part[idPart].pos.z += speed2_z;
    part[idPart].speed.x = speed2_x;
    part[idPart].speed.y = speed2_y;
    part[idPart].speed.z = speed2_z;
  }
}
free(coeffs);
for (int i = 0; i < nbPart; i++) {
  part[i].speed.x /= deltaT;
  part[i].speed.y /= deltaT;
  part[i].speed.z /= deltaT;
}
free(lFieldAtCorners);
```

Grammar of permissions

Syntax in C	Syntax in the theory	Description
$p \rightsquigarrow \texttt{Cell}$	$p \rightsquigarrow \mathsf{Cell}$	full permission
$p \rightsquigarrow \texttt{Matrix1}(n)$	$p \rightsquigarrow \mathsf{Matrix1}(n)$	array permission
$p \rightsquigarrow \texttt{Matrix2}(m,\ n)$	$p \rightsquigarrow \mathsf{Matrix2}(m, n)$	matrix permission
$\texttt{for}\ i\ \texttt{in}\ r\ \texttt{->}\ H(i)$	$\bigstar_{i \in r} H(i)$	group of permissions
$\texttt{\_RO}(\alpha,\ H)$	αH	read-only permission
$\texttt{\_Uninit}(H)$	$\mathsf{Uninit}(H)$	uninitialized permission

Grammar of permissions

Syntax in C	Syntax in the theory	Description
$p \rightsquigarrow \texttt{Cell}$	$p \rightsquigarrow \mathsf{Cell}$	full permission
$p \rightsquigarrow \texttt{Matrix1}(n)$	$p \rightsquigarrow \mathsf{Matrix1}(n)$	array permission
$p \rightsquigarrow \texttt{Matrix2}(m\texttt{,}\ n)$	$p \rightsquigarrow \mathsf{Matrix2}(m, n)$	matrix permission
$\texttt{for}\ i\ \texttt{in}\ r\ \texttt{->}\ H(i)$	$\bigstar_{i \in r} H(i)$	group of permissions
$\texttt{\_RO}(\alpha\texttt{,}\ H)$	αH	read-only permission
$\texttt{\_Uninit}(H)$	$\mathsf{Uninit}(H)$	uninitialized permission

Read-only permissions:

$$H = 1\,H \qquad \text{and} \qquad (\alpha + \beta)H = \alpha H \star \beta H$$

Uninitialized permissions:

$$\varnothing \overset{\text{alloc}}{\to} \mathsf{Uninit}(H) \overset{\text{write}}{\to} H \overset{\text{read}}{\to} H \qquad \text{and} \qquad H \overset{\text{weaken}}{\to} \mathsf{Uninit}(H)$$

Usage information

- typing triple $\{\Gamma\}\ t^{\Delta}\ \{\Gamma'\}$
- Γ and Γ' contain pure and linear resources of separation logic
- Δ binds keys of Γ and Γ' to *usage*.

Grammar of usage kinds:

x : required	pure fact used by t
x : ensured	pure fact produced by t
y : full	linear resource consumed by t
y : produced	linear resource produced by t
y : uninit	resource being written to (before any read)
y : splittedFrac	linear resource used in read-only
y : joinedFrac	y was $(\alpha - \beta)H$ and was merged with βH
x or y not bound	resource not used (i.e., *framed*)

Correctness Criteria for instruction move

$$\boxed{\mathcal{E}\left[T_1^{\Delta_1}; T_2^{\Delta_2}\right]} \quad \longmapsto \quad \boxed{\mathcal{E}\left[T_2; T_1;\right]}$$

$$\text{correct if: } \begin{cases} \Delta_1.\mathsf{notRO} \mathbin{\dot{\cap}} \Delta_2 = \varnothing \\ \Delta_2.\mathsf{notRO} \mathbin{\dot{\cap}} \Delta_1 = \varnothing \end{cases}$$

→ Resources shared between T_1 and T_2 must be read-only.

Transformation: instruction delete

$$\boxed{\mathcal{E}\left[\Gamma\ T_1^{\Delta}\right]} \quad \longmapsto \quad \boxed{\mathcal{E}\,[\,]}$$

correct if:

$\mathcal{E}\,[\,\textbf{ghost}(\Gamma_m \longrightarrow \mathsf{IntoUninit}(\Gamma_m))]$ typechecks

where $\Gamma_m \equiv \Gamma \mathrel{|\!\cdot} \Delta.\mathsf{notRO}$.

→ All resources modified by T_1 can safely be turned into their uninitialized form.

Transformation: loop fission

$$\boxed{\begin{array}{l}\textbf{for}_{\ \chi}\ i \in r_i\ \{\\ \quad T_1^{\Delta_1}\\ \quad \Gamma\\ \quad T_2^{\Delta_2}\\ \}\end{array}} \longmapsto \boxed{\begin{array}{l}\textbf{for}_{\ \chi_1}\ i \in r_i\ \{\\ \quad T_1;\\ \}\\ \textbf{for}_{\ \chi_2}\ i \in r_i\ \{\\ \quad T_2;\\ \}\end{array}}$$

correct if:

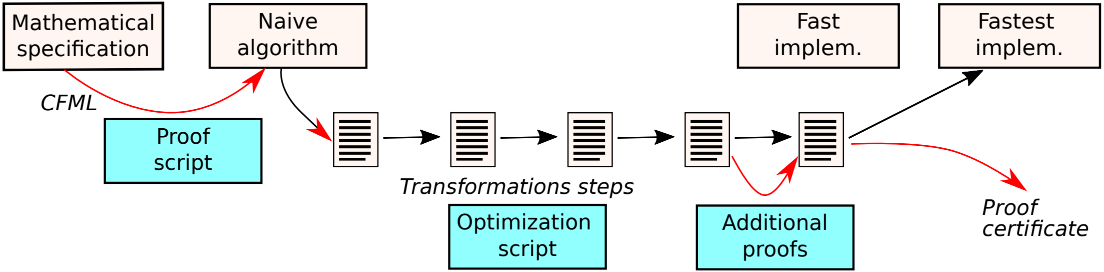
$$\begin{cases} i \text{ not free in } \chi.\mathsf{shrd}\\ \chi.\mathsf{shrd} \mathrel{|\!\cdot} (\Delta_1.\mathsf{notRO} \mathbin{\dot\cap} \Delta_2) = \varnothing\\ \chi.\mathsf{shrd} \mathrel{|\!\cdot} (\Delta_2.\mathsf{notRO} \mathbin{\dot\cap} \Delta_1) = \varnothing\\ \text{the output program typechecks}\end{cases}$$

with:

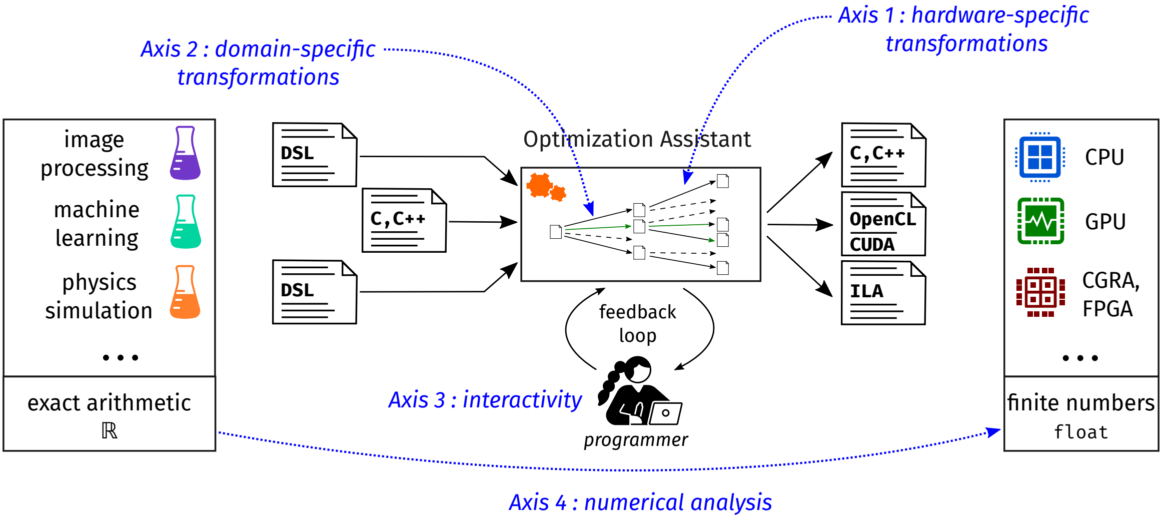
$$\_, F \equiv \Gamma \boxminus \chi.\mathsf{shrd.inv} \qquad \_, R \equiv F \boxminus \mathsf{StackAllocCells}(T_1)$$

$$\chi_1 \equiv \begin{cases}\mathsf{vars} \equiv \chi.\mathsf{vars}\\ \mathsf{shrd} \equiv \chi.\mathsf{shrd} \mathrel{|\!\cdot} \Delta_1\\ \mathsf{excl.pre} \equiv \chi.\mathsf{excl.pre}\\ \mathsf{excl.post} \equiv R\end{cases} \qquad \chi_2 \equiv \begin{cases}\mathsf{vars} \equiv \chi.\mathsf{vars}\\ \mathsf{shrd} \equiv \chi.\mathsf{shrd} \mathrel{|\!\cdot} \Delta_2\\ \mathsf{excl.pre} \equiv R\\ \mathsf{excl.post} \equiv \chi.\mathsf{excl.post}\end{cases}$$

Towards Transforming Verified Code



Towards an Extensible Interactive Optimization Assistant



Axis 2 : domain-specific transformations

Axis 1 : hardware-specific transformations

image processing

machine learning

physics simulation

...

exact arithmetic

ℝ

DSL

C,C++

DSL

Optimization Assistant

C,C++

OpenCL

CUDA

ILA

CPU

GPU

CGRA, FPGA

...

finite numbers

`float`

feedback loop

Axis 3 : interactivity

programmer

Axis 4 : numerical analysis

Towards an Extensible Interactive Optimization Assistant

funding from AI project in 2025-2031 ?

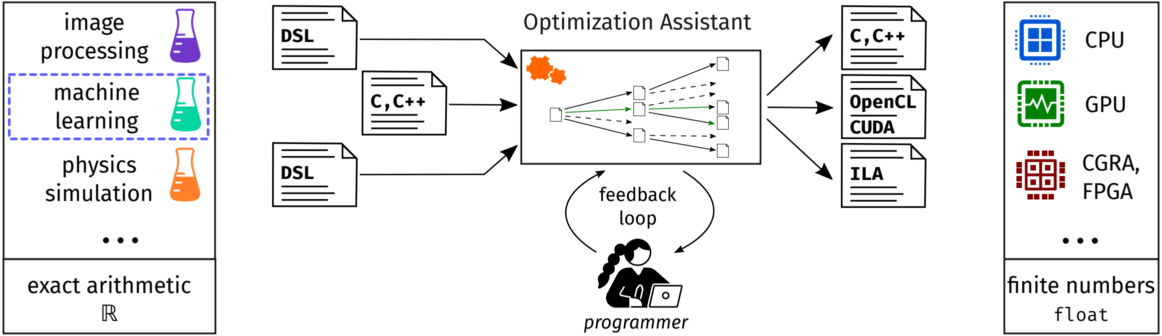


image processing

machine learning

physics simulation

...

exact arithmetic

ℝ

DSL

C,C++

DSL

Optimization Assistant

feedback loop

programmer

C,C++

OpenCL CUDA

ILA

CPU

GPU

CGRA, FPGA

...

finite numbers

`float`

Master Internships that are starting

Since January:

- optimization + numerical analysis, 2 interns co-supervised with Eva Darulova (Uppsala University)

In March:

- transformation of OCaml code, and refine from Coq to OCaml to C
- generalization to realistic numerical simulation code
- sketch-guided polyhedric optimization

In September:

- targeting GPUs, hardware-specific transformations (collab with Bastian Köpcke)

Maybe?

- Extensibility with DSLs as libraries

Discussion points

- How to enable defining domain-specific libraries? functions + transformations
  - How to support extensible constructs? (MimIR?)
  - How to faciliate user-defined transformations?
  - Domain-specific compiler = optimization assistant + libraries + heuristics?
- AI Applications? Can we leverage our work to improve the ML stack?
- How to leverage static analysis to infer contracts?
- Connection with Iris, possibly via RefinedC or a variant thereof?
- Collaboration: sharing software, benchmarks, PhD research visits

Discussion points

- How to enable defining domain-specific libraries? functions + transformations
  - How to support extensible constructs? (MimIR?)
  - How to faciliate user-defined transformations?
  - Domain-specific compiler = optimization assistant + libraries + heuristics?
- AI Applications? Can we leverage our work to improve the ML stack?
- How to leverage static analysis to infer contracts?
- Connection with Iris, possibly via RefinedC or a variant thereof?
- Collaboration: sharing software, benchmarks, PhD research visits

---

Traces, journal paper draft:

🌐 github.com/charguer/optitrust
🌐 chargueraud.org/softs/optitrust/

Appendix

Program Resources are Computed at Every Program Point

```
$R_1 : \alpha_1(\texttt{A} \rightsquigarrow \texttt{Matrix2(m, p)}) \star R_2 : \alpha_2(\texttt{B} \rightsquigarrow \texttt{Matrix2(p, n)}) \star$
$R_3 : \bigstar\, i \in 0..m \bigstar\, j \in 0..n$  $\texttt{\&C[i][j]} \rightsquigarrow \texttt{Cell}$
for (int i = 0; i < m; i++) {
  $R_4 : (\alpha_4/m)(\texttt{A} \rightsquigarrow \texttt{Matrix2(m, p)}) \star R_5 : (\alpha_5/m)(\texttt{B} \rightsquigarrow \texttt{Matrix2(p, n)}) \star$
  $R_6 : \bigstar\, j \in 0..n$  $\texttt{\&C[i][j]} \rightsquigarrow \texttt{Cell}$
  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;
    $R_7 \star R_8 \star R_{11} \star R_{12} : \texttt{\&C[i][j]} \rightsquigarrow \texttt{Cell}$
  }
  $R_4 \star R_5 \star R_{13} : \bigstar\, j \in 0..n$  $\texttt{\&C[i][j]} \rightsquigarrow \texttt{Cell}$
}
$R_1 \star R_2 \star R_{14} : \bigstar\, i \in 0..m \bigstar\, j \in 0..n$  $\texttt{\&C[i][j]} \rightsquigarrow \texttt{Cell}$
```

Program Resources are Computed at Every Program Point

```
$R_1 : \alpha_1$(A $\rightsquigarrow$ Matrix2(m, p)) $\star$ $R_2 : \alpha_2$(B $\rightsquigarrow$ Matrix2(p, n)) $\star$
$R_3 : \bigstar\, i \in 0..m \bigstar\, j \in 0..n$ &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)) $\star$
  $R_6 : \bigstar\, j \in 0..n$ &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
  }
  $R_4 \star R_5 \star R_{13} : \bigstar\, j \in 0..n$ &C[i][j] $\rightsquigarrow$ Cell
}
$R_1 \star R_2 \star R_{14} : \bigstar\, i \in 0..m \bigstar\, j \in 0..n$ &C[i][j] $\rightsquigarrow$ Cell
```

Matmul: unoptimized code, multi-dimensional

```
void mm(const int m, const int n, const int p,
 const float A[m][p], const float B[p][n], float C[m][n]) {
  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;
    }
  }
}

void mm1024(
 const float A[1024][1024],
 const float B[1024][1024],
 float C[1024][1024]) {
  mm(1024, 1024, 1024, A, B, C);
}
```

Matrix-multiply, optimized code, 150x on 4 cores

```
void mm1024(const float* A, const float* B, float* C) {
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++) {
          s[j] += A[1024 * (32 * bi + i) + 4 * bk] * pB[32768 * bj + 128 * bk + j]; }
        #pragma omp simd
        for (int j = 0; j < 32; j++) {
          s[j] += A[1 + 1024 * (32 * bi + i) + 4 * bk] * pB[32 + 32768 * bj + 128 * bk + j]; }
        #pragma omp simd
        for (int j = 0; j < 32; j++) {
          s[j] += A[2 + 1024 * (32 * bi + i) + 4 * bk] * pB[64 + 32768 * bj + 128 * bk + j]; }
        #pragma omp simd
        for (int j = 0; j < 32; j++) {
          s[j] += A[3 + 1024 * (32 * bi + i) + 4 * bk] * pB[96 + 32768 * bj + 128 * bk + 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); }
```

Matmul: optimizations at play

- An auxiliary matrix stores the transpose of `B`
  → it greatly improves memory access patterns
- Loops are tiled and reordered to operate on blocks of size 32
  → blocking improves locality, and enables parallelization
- Results accumulated into an intermediate array of size 32
  → this array is mapped onto a 256-bit register
- Results accumulated into another array of size 32x32
  → features a different memory layout; memcopy operations is used
- One loop with 4 iterations is manually unrolled
  → appears necessary to help the compiler vectorize (use SIMD)

Matmul: annotated code

```
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)");
  for (int i = 0; i < m; i++) {
    __xmodifies("for j in 0..n -> &C[MINDEX2(m, n, i, j)] ⤳ Cell");
    for (int j = 0; j < n; j++) {
      __xmodifies("&C[MINDEX2(m, n, i, j)] ⤳ Cell");
      float sum = 0.0f;
      for (int k = 0; k < p; k++) {
        __ghost_begin(focusA, matrix2_ro_focus, "A, i, k");
        __ghost_begin(focusB, matrix2_ro_focus, "B, k, j");
        sum += A[MINDEX2(m, p, i, k)] * B[MINDEX2(p, n, k, j)];
        __ghost_end(focusA);
        __ghost_end(focusB);
      }
      C[MINDEX2(m, n, i, j)] = sum;
    }
  }
}
```

Matmul: annotated code

```
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)");
  for (int i = 0; i < m; i++) {
    __xmodifies("for j in 0..n -> &C[MINDEX2(m, n, i, j)] ⇝ Cell");
    for (int j = 0; j < n; j++) {
      __xmodifies("&C[MINDEX2(m, n, i, j)] ⇝ Cell");
      float sum = 0.0f;
      for (int k = 0; k < p; k++) {
        __ghost_begin(focusA, matrix2_ro_focus, "A, i, k");
        __ghost_begin(focusB, matrix2_ro_focus, "B, k, j");
        sum += A[MINDEX2(m, p, i, k)] * B[MINDEX2(p, n, k, j)];
        __ghost_end(focusA);
        __ghost_end(focusB);
      }
      C[MINDEX2(m, n, i, j)] = sum;
    }
  }
}
```

Goal: reproduce TVM output, with achieves 150× on a 4-core Intel i7.

Future work: match Intel MKL, which achieves 204×.

Matmul: transformation script

```
!! Function.inline_def [cFunDef "mm"];
let tile (loop_id, tile_size) = Loop.tile (int tile_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 ~lhs:[cVar "sum"] ()];
!! 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 ""];
!! Loop.unroll ~simpl:Arith.do_nothing [cFor ~body:[cPlusEq ~lhs:[cVar "s"] ()] "k"];
!! Cleanup.std ();
```

Matmul: the TVM approach

Input code

featuring the transpose of B

```
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))

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))
```

Transformation hints

a.k.a. *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)
```

OpenCV case study

OpenCV’s box-blur: row-sum function

OpenCV is a reference, open source library for computer vision.

Box-blur: each pixel is replaced with an average of the KxK nearby pixels.

Implementation:

1. compute sums of K nearby pixels on every row
2. compute sums of K nearby pixels on every column
3. postpone the division to a final processing step

RowSum: transformation script

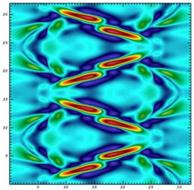
```
!! Specialize.variable_multi ~mark_then:fst ~mark_else:"nokn"
    ["kn", int 3; "kn", int 5] [cFunBody "rowSum"; cFor "i"];
!! Reduce.elim ~inline:true [nbMulti; cMark "kn"; cFun "reduce_spe1"];
!! Loop.collapse [nbMulti; cMark "kn"; cFor "i"];

!! Loop.swap [nbMulti; cMark "nokn"; cFor "i"];
!! Reduce.slide ~mark_alloc:"acc" [nbMulti; cMark "nokn"; cArrayWrite "D"];
!! Reduce.elim [nbMulti; cMark "acc"; cFun "reduce_spe1"];
!! Variable.elim_reuse [nbMulti; cMark "acc"];
!! Reduce.elim ~inline:true [nbMulti; cMark "nokn"; cFor "i"; cFun "reduce_spe1"];

!! Specialize.variable_multi ~mark_then:fst
    ["cn", int 1; "cn", int 3; "cn", int 4] [cMark "nokn"; cFor "c"];
!! Loop.unroll [nbMulti; cMark "cn"; cFor "c"];
!! Target.foreach [nbMulti; cMark "cn"] (fun c ->
    Loop.fusion_targets ~into:FuseIntoLast [nbMulti; c; cFor "i" ~body:[cArrayWrite "D"]];
    Instr.gather_targets [c; cStrict; cArrayWrite "D"];
    Loop.fusion_targets ~into:FuseIntoLast [nbMulti; c; cFor ~stop:[cVar "kn"] "i"];
    Instr.gather_targets [c; cFor "i"; cArrayWrite "D"];
);
!! Cleanup.std ();
```

PIC case study

Plasma simulation, Particle-in-Cell method (PIC)



In our EuroPar'18 article:

- 256 billion particles
- 3072 cores on 128 sockets
- 12.3 TB RAM

```
Foreach time step
    Foreach particle $i$ with position $\mathbf{x_i}$ and velocity $\mathbf{v_i}$
        Interpolate the electric grid $\mathbf{E}$ at the position $\mathbf{x_i}$
        Update the velocity of the particle
        Update the position of the particle
        Accumulate the particle's charge on the nearest grid points
    Recompute $\mathbf{E}$ from the charge stored in the grid
```

PIC: numerical scaling for uniform particles

Acceleration for charged particles:

$$\mathbf{F} = m \cdot \mathbf{a} \qquad \text{and} \qquad \mathbf{F} = q \cdot \mathbf{E} \qquad \text{gives} \qquad \mathbf{a} = \frac{q}{m}\mathbf{E}$$

Update speed and position of every particle at every time step:

$$\mathbf{v} \mathrel{+}= \mathbf{a} \cdot \Delta_t \qquad \text{and} \qquad \mathbf{x} \mathrel{+}= \mathbf{v} \cdot \Delta_t$$

PIC: numerical scaling for uniform particles

Acceleration for charged particles:

$$\mathbf{F} = m \cdot \mathbf{a} \qquad \text{and} \qquad \mathbf{F} = q \cdot \mathbf{E} \qquad \text{gives} \qquad \mathbf{a} = \frac{q}{m}\mathbf{E}$$

Update speed and position of every particle at every time step:

$$\mathbf{v} \mathrel{+}= \mathbf{a} \cdot \Delta_t \qquad \text{and} \qquad \mathbf{x} \mathrel{+}= \mathbf{v} \cdot \Delta_t$$

Scaling values:

$$\mathbf{E}' = \frac{q\Delta_t^2}{m} \cdot \mathbf{E} \qquad \text{and} \qquad \mathbf{v}' = \Delta_t \cdot \mathbf{v}$$

Update to speed and position, revisited:

$$\mathbf{v}' \mathrel{+}= \mathbf{E}' \qquad \text{and} \qquad \mathbf{x}' \mathrel{+}= \mathbf{v}'$$

Micro-PIC optimized code: scaling of the speed

```
void simulate(double stepDuration,
  particle* particles, int nbParticles,
  vect* fieldAtCorners, int nbSteps,
  double pCharge, double pMass) {
  ...
  for (int i = 0; i < nbParticles; i++) {
    particles[i].speed.x *= deltaT;
    particles[i].speed.y *= deltaT;
    particles[i].speed.z *= deltaT;
  }
  for (int idStep = 0; idStep < nbSteps; idStep++) {
    for (int idPart = 0; idPart < nbParticles; idPart++) {
      ... // body of the core loops, with particles[i].speed scaled
    }
  }
  for (int i = 0; i < nbParticles; i++) {
    particles[i].speed.x /= deltaT;
    particles[i].speed.y /= deltaT;
    particles[i].speed.z /= deltaT;
  }
  ...
}
```

→ General-purpose compilers cannot apply such a transformation.

Micro-PIC optimized code: core loops

```
for (int idStep = 0; idStep < nbSteps; idStep++) {
  for (int idPart = 0; idPart < nbParticles; idPart++) {
    cornerInterpolationCoeff(particles[idPart].pos, coeffs);
    double fieldAtPosX = 0.;
    double fieldAtPosY = 0.;
    double fieldAtPosZ = 0.;
    for (int k = 0; k < nbCorners; k++) {
      fieldAtPosX += coeffs[k] * lFieldAtCorners[k].x;
      fieldAtPosY += coeffs[k] * lFieldAtCorners[k].y;
      fieldAtPosZ += coeffs[k] * lFieldAtCorners[k].z;
    }
    const double speed2X = particles[idPart].speed.x + fieldAtPosX;
    const double speed2Y = particles[idPart].speed.y + fieldAtPosY;
    const double speed2Z = particles[idPart].speed.z + fieldAtPosZ;
    particles[idPart].pos.x += speed2X;
    particles[idPart].pos.y += speed2Y;
    particles[idPart].pos.z += speed2Z;
    particles[idPart].speed.x = speed2X;
    particles[idPart].speed.y = speed2Y;
    particles[idPart].speed.z = speed2Z;
  }
}
```

→ Looks similar to handwritten code.

Full Particle-in-Cell simulation

Many more optimizations performed:

- array-of-structure-to-structure-of-arrays
- scaling of numerical values
- parallel processing of the grid cells
- use of dynamically-sized bags
- use of concurrent bags
- fission of the main loop in 3 parts
- vectorization of the deposit of the charges

Full Particle-in-Cell simulation

Many more optimizations performed:

- array-of-structure-to-structure-of-arrays
- scaling of numerical values
- parallel processing of the grid cells
- use of dynamically-sized bags
- use of concurrent bags
- fission of the main loop in 3 parts
- vectorization of the deposit of the charges

So far: reproduced the desired code using 140 transformation steps.

Work remaining: implement the numerous missing correctness criteria.

Inside OptiTrust

OptiTrust's internal language

Internally, the OptiTrust abstract syntax tree (AST) consists of an imperative λ-calculus.

1. Parse C/C++ code using Clang
2. Read Clang output in OCaml using the ClangML package
3. Translate to OptiTrust AST: eliminate mutable vars and l-values
4. Typecheck the code
5. Repeat: apply a transformation, then retypecheck
6. Translate back to C syntax and print code

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;
  }
}
```

Combined Transformations on MatMul

Combined transformations:

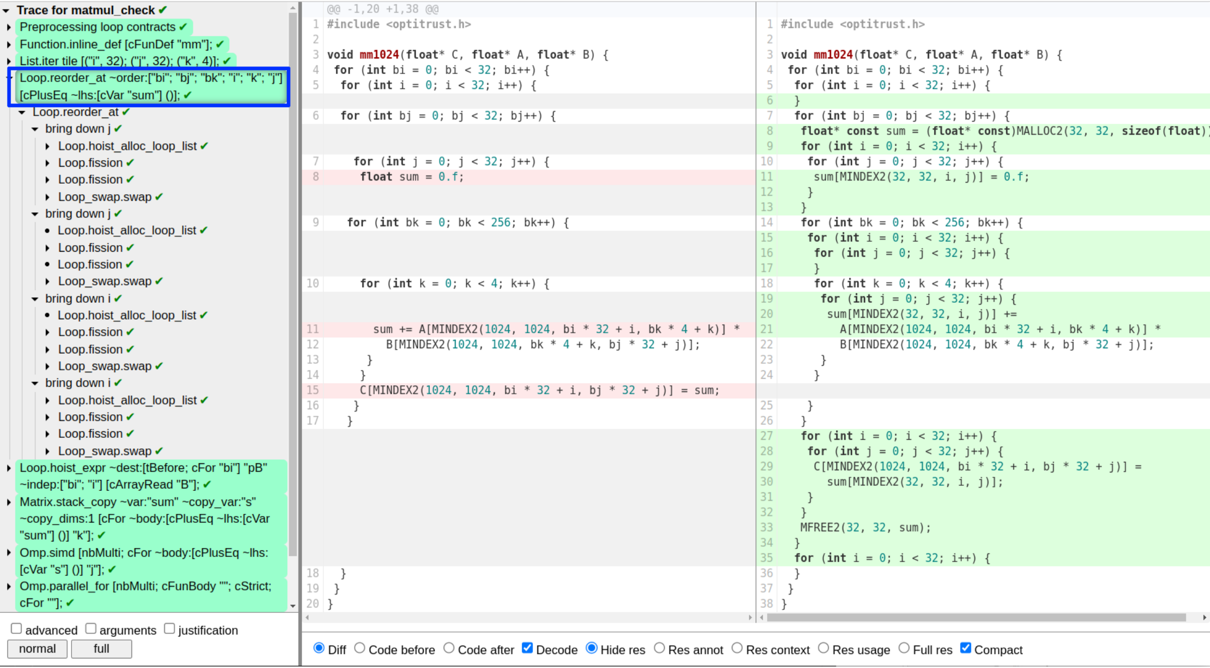
- compose basic transformations
- are valid by composition

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 ""]; ✓

☐ 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

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`
- ...

Typing algorithm

Compute resources at every program point.

$$\Gamma \quad := \quad \langle x_0 : \tau_0, \ldots, x_n : \tau_n \mid y_0 : H_0, \ldots, y_n : H_n \rangle$$

- τ denotes either a program type or a mathematical type
- H denotes a linear resource of separation logic

Typing judgment:

$$\{\Gamma\}\, t^{\Delta}\, \{\Gamma'\}$$

where Δ denotes the usage information, detailed further.

Grammar of contracts

Function contract γ in a function definition: $\textbf{fun}(a_1, ..., a_n)_\gamma \mapsto t$
aims to satisfy the triple:

$$\{\gamma.\mathsf{pre}\}\ f(a_1, ..., a_n)\ \{\gamma.\mathsf{post}\}$$

Grammar of contracts

Function contract γ in a function definition: $\textbf{fun}(a_1, ..., a_n)_\gamma \mapsto t$
aims to satisfy the triple:

$$\{\gamma.\mathsf{pre}\}\ f(a_1, ..., a_n)\ \{\gamma.\mathsf{post}\}$$

Loop contract χ in a loop: $\textbf{for}\ (i \in r)_\chi\{t\}$
aims to satisfy the triple:

$$\begin{array}{l}\{\chi.\mathsf{vars} \star \chi.\mathsf{shrd.reads} \star \chi.\mathsf{shrd.inv}(i) \star \chi.\mathsf{excl.pre}(i)\} \\ t \\ \{\chi.\mathsf{shrd.reads} \star \chi.\mathsf{shrd.inv}(i+1) \star \chi.\mathsf{excl.post}(i)\}\end{array}$$

About OptiTrust

Open-source project, open to external contributions.

Disclaimer: not yet ready for production use.

Entry point: our 60-page TOPLAS submission.

Matrix Multiplication Performance

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

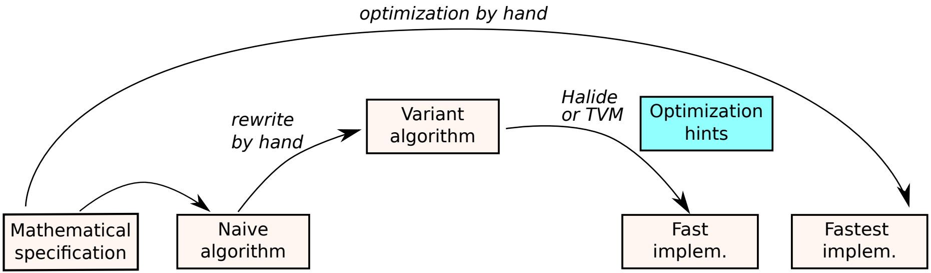
version	single-thread	multi-thread
unoptimized	1×	1×
optimized	46×	150×
TVM	46×	150×
numpy (Intel MKL)[1]	71×	183×

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

High-performance programming

Increasing complexity of the hardware poses new challenges.



Optimization by hand

- ✓ Generally applicable
- ✓ Fastest output code
- ✗ Very costly
- ✗ Error-prone

Using specialized compilers

- ✗ Limited applicability
- ✗ Fast but not fastest
- ✓ Much less costly
- ✓ Much safer

Trusted code base

Compilers are incredibly hard to get 100% correct.

OptiTrust is designed to minimize the TCB, with its 2-layer approach:

- **Basic transformations**
  - minimal implementation
  - carefully studied correctness criteria
- **Combined transformations**
  - implement high-level procedures
  - correct by composition, thus out of the TCB

In the future, for code verified w.r.t. full functional correctness, the production of proof certificates would make OptiTrust out of the TCB.

Future work

- More transformations
- More language features
- More separation logic
  - full functional correctness
  - proof certificates
  - link to verified compilers
- More performance (incremental typechecker, ...)
- More case studies
- More documentation, more tutorials
- More users, more contributors

Contributors to OptiTrust

Development:

Damien Rouhling	Postdoc 1 year
Begatim Bytyqi	Engineer 1.5 year
Thomas Koehler	Postdoc 2 years, now CNRS researcher
Guillaume Bertholon	PhD 2 years, now 3rd year

With advices from:

Jens Gustedt, Alain Ketterlin	Inria Strasbourg
François Pottier, Xavier Leroy	Inria Paris
Yannick Zakowski	Inria Lyon

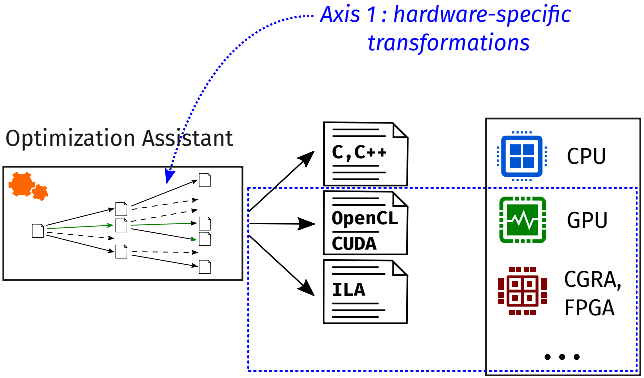
Funded by Inria and ANR.

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



Axis 1 : hardware-specific transformations

Optimization Assistant

C,C++

OpenCL
CUDA

ILA

CPU

GPU

CGRA,
FPGA

• • •

Master Intern from September 2025 ✓
supervised with Arthur Charguéraud

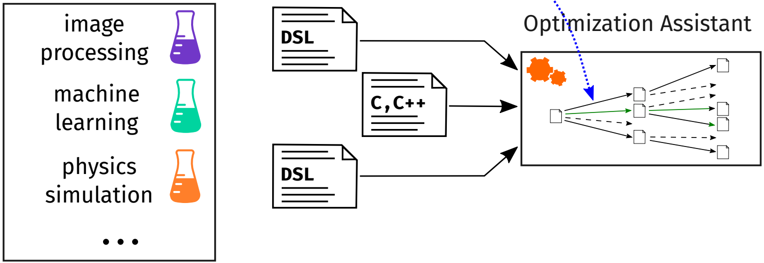
PhD Researcher from 2026/2027 ?

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

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

Axis 2 : domain-specific transformations



Plan:

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

Master Intern / PhD / PostDoc ?
with Christophe Alias,
Eric Leclercq, Annabelle Gillet

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

Difficulties:
- breaking abstraction and stack boundaries
- 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:

- now: sketch-guided optimization of imperative code
  polyhedric and/or beam search
- later: optimization suggestions and vizualisations
  bottlenecks, hot code

Difficulty:

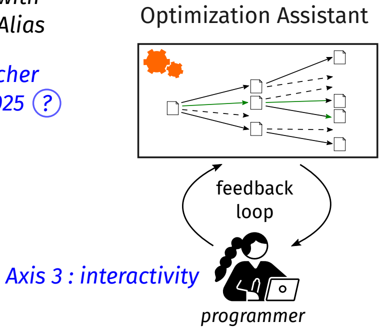
- exploring a complex and evolving optimization space

Interdisciplinary Collaboration:

- Géry Casiez (Inria LOKI, Human-Computer Interaction)

Master Intern
from March 2025 ✓
supervised with
Christophe Alias

PhD Researcher
from late 2025 ?



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:

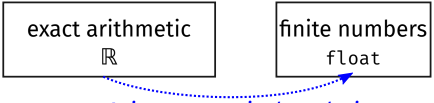
- most compilers do not perform numerical analyses

  'gcc': no optimisations over floats
  'gcc -ffast-math': treats floats as reals

International Collaboration:

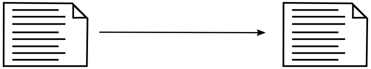
- Eva Darulova, Uppsala University [TOPLAS'17, SAS'23]

  1-2 Master Interns from Spring 2025 ✓
  hiring PhDs/PostDocs under ERC Starting grant HORNET



Axis 4 : numerical analysis

functional language on arrays



- interval analysis on values and errors

  error is below `e(x o y) + d`

- empirical pareto fronts

  shadow execution