

OptiTrust: an Interactive Framework for Source-to-Source Transformations

Arthur Charguéraud, Begatim Bytyqi, Damien Rouhling, Yann A Barsamian

▶ **To cite this version:**

Arthur Charguéraud, Begatim Bytyqi, Damien Rouhling, Yann A Barsamian. OptiTrust: an Interactive Framework for Source-to-Source Transformations. 2022. hal-03773485

HAL Id: hal-03773485
https://inria.hal.science/hal-03773485

Preprint submitted on 9 Sep 2022

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

OptiTrust: an Interactive Framework for Source-to-Source Transformations

Arthur Charguéraud
Inria
France

Begatim Bytyqi
Inria
France

Damien Rouhling
Inria
France

Yann Barsamian
École Européenne de Bruxelles
Belgium

Abstract

This paper presents an interactive framework for developing high-performance C code via series of source-to-source transformations. Optimization steps are described in transformation scripts, expressed as OCaml programs. The programmer can interactively visualize the textual differences associated with any step of the script. We demonstrate the effectiveness of OptiTrust by reproducing a manually optimized Particle-In-Cell numerical simulation, starting from a direct, unoptimized version of the algorithm. This case study covers many state-of-the-art optimization patterns that appear in numerical simulation codes. We argue that, compared with optimizing code by hand, deriving high performance code using a transformation script makes the code easier to review, easier to debug, and easier to maintain as the intended program or as the target hardware evolves.

1 Introduction

Turning a high-level, unoptimized algorithm into a high-performance code is a challenging process. Outside of niche domains where specialized compilers for domain-specific languages (DSL) can be leveraged, the transformations required for taking *full* advantage of the hardware capabilities are, by far, out of reach of automated compilers. Compilation hints provided by the programmer in the form of *pragmas* enable to describe a first set of transformations. However they are ill-suited to describe the second—or the twentieth—round of transformations.

In practice, it is commonplace for programmers to optimize code *manually*, i.e., to explicitly code the result of a combination of numerous transformations. Typically, to optimize the code that runs on a specific computation node, a programmer would focus on the critical computation kernels and then work for weeks, if not months, to optimize them. Doing so generally requires to reorganize the data layout in memory, to reorder loops, to set up parallel threads, to sparingly introduce concurrent instructions, etc. Common goals are to maximize the use of vector instructions (SIMD), to minimize the pressure on the memory bus, and to exploit all the cores available.

Manual code optimization is, however, quite unsatisfactory. **Cost**: Manually implementing code optimizations is very time consuming, in particular because optimizations typically make the code size grow substantially. **Performance**: Because each candidate implementation takes a lot of time to develop, the programmer often lacks time to explore the optimization space. **Maintainability**: When small changes are made to the high-level algorithm, corresponding changes need to be transposed to the optimized code. Doing so requires the programmers that manipulate the code to fully understand all the optimizations that have been baked into the code. **Correctness**: Manual code optimization is error prone. If a programmer combines 35 optimizations, and implements each one correctly with probability 98%, then the final code has over 50% chances of containing a bug!

An alternative route to producing a highly optimized code is to derive it from an unoptimized, high-level code, via a series of source-to-source transformations, guided by the programmer. In this paper, we present OptiTrust: an interactive framework for developing source-to-source transformation scripts. To argue for the expressiveness of the framework, we present a case study that reproduces a state-of-the-art, high-performance C implementation of a Particle-In-Cell (PIC) simulation, starting from a direct, unoptimized implementation of the simulation.

2 Related work

Optimizing compilers. General purpose compilers such as GCC or ICC apply advanced program optimizations. They are, however, inherently limited by three factors. First, they must respect the C standard, preventing them from, e.g., refactoring certain floating point expressions. Second, they may be unable to synthesize complex program invariants required to justify certain transformations. Third, they generally cannot apply transformations for which the risk of significantly degrading performances is too high. OptiTrust, in contrast, aims to leverage human expertise.

Compiler directives. To introduce human guidance, a common approach is to insert *pragmas* into the code. For example, Scout [1] is a pragma-based tool for guiding source-to-source transformations that introduce vector instructions. The main limitation of pragmas is that they are ill-suited for describing sequences of optimizations. Indeed, there is no

easy way to attach a pragma to a line of code that is generated by a first optimization. Kruse and Finkel [2] suggest the possibility to stack up pragmas, by providing labels as additional pragma arguments: a second pragma may refer to the labels introduced by the transformation described in a first pragma. This approach does not scale up well beyond a handful of successive transformations. OptiTrust, in contrast, aims to support chains of dozens of transformations.

Domain specific languages (DSLs). Another possible approach to overcome the limitations of general-purpose compilers is to leverage *domain specific languages* (DSL), such as Halide [3], TVM [4], or Boast [5]. Because DSLs are restricted languages, they can benefit from specialized compilers that explore a large space of possible code layout, using carefully tuned heuristics and ad-hoc performance diagnosis tools. Yet, the language restriction is also the Achilles' heel of DSLs: as soon as the user's application requires a single feature that falls outside of what the DSL can express, the programmer loses most if not all of the benefits of the DSL. In contrast to DSLs, OptiTrust sticks to a general-purpose language, supporting a large subset of the C language.

Code transformations via rewrite rules. A *rewrite rule* maps a code pattern to another code pattern. A number of tools exploit rewrite rules to perform source-to-source transformations. For example, TXL [6] is a multi-language rewrite system, whose patterns are expressed at the level of syntax, using grammars. Coccinelle [7] allows the programmer to describe *semantic patches* in C code. CodeBoost [8] applies the Stratego program transformation language [9] to C++ code. CodeBoost was used to turn high-level operations on matrices and vectors into typical high-performance source code. As we demonstrate through our case study, OptiTrust is also able to optimize vector-manipulating code. At the same time, OptiTrust provides a much more expressive language for describing transformations, going far beyond rewrite rules.

Source code manipulation frameworks. Frameworks that offer more expressiveness than rewrite rules generally give access to the abstract syntax tree (AST) of the source code. Traditional compilers employ an AST, but they are not designed for synthesizing pieces of AST. Moreover, traditional compilers operate on intermediate representations, and lose the structure of the original code. Other compilers, such as ROSE [10, 11] and Cetus [12], are designed to support code transformations (and code analyses) at the level of C code. Source-to-source transformations have also been employed to produce code targeting GPUs [13–15]. All these source-level tools focus only on implementing generic code transformations, whereas OptiTrust furthermore provides facilities for writing scripts to optimize specific programs.

Interestingly, Cetus [12] exploits an intermediate representation (IR), with fewer constructs than the C language, and yet has the possibility to return to readable C code. OptiTrust follows a similar route, but for interactive transformations.

Transformation scripts. Expressing a series of source-level transformations for a specific program can be done by means of a *transformation script*. Such scripts have appeared in particular in the context of polyhedral transformations [16, 17], for example in Loopy [18] and in work by Zinenko et al. [19]. CHiLL [20, 21] includes transformations that go beyond the polyhedral model. It has been applied to generate finely tuned CUDA code from high-level linear algebra kernels. Clava [22] is a C++ source-to-source analysis and transformation framework. Its transformation scripts are written in LARA, a JavaScript-based DSL with special constructs for code queries, analysis and transformations. POET [23, 24] is a scripting language for performing program transformations, for C/C++ as well as other languages. POET has been employed to generate optimized code for linear algebra kernels, including semi-automated exploration of a search space of possible optimizations. None of the aforementioned projects have demonstrated the ability to optimize realistic numerical simulation code like that of our case study. Moreover, none of them feature a system for targeting program points with the expressiveness and conciseness offered by OptiTrust targets.

3 Contribution

This paper introduces OptiTrust, an interactive transformation framework that allows refining an unoptimized code into a high performance code, through the development of a transformation script that applies a series of source-to-source optimizations. OptiTrust is meant to be used by an expert in program optimization who has some intuition of potentially interesting optimization strategies, and is interested in producing the corresponding optimized code without going through the tedious, error-prone and hard-to-maintainable process of writing optimized code by hand. A key aspect is that OptiTrust supports transformations whose correctness is nontrivial: such transformations are out of reach of automated optimization tools, and can only be applied by an expert with good understanding of the invariants of the program at hand. By providing interactive feedback in the form of human-readable code, *OptiTrust allows human expertise to be leveraged in the middle of a chain of optimizations.*

To demonstrate the expressiveness of OptiTrust, we present one major case study: the optimization of a numerical simulation, for which we leverage OptiTrust to produce a code that matches a pre-existing code [25] that had been carefully optimized by hand. As detailed in the rest of the paper, this one case study covers a broad scope of optimizations, including: (1) core data layout transformations [26] such as array-of-structure to structure-of-array, array tiling, structure fields inlining, interchange of dimensions of an array, scaling or shifting by a constant factor of all the values from an array; (2) instruction-level transformations [27] such as

variable and function inlining, uninlining, instruction reordering, constant propagation, dead code elimination, arithmetic simplifications, pattern-based rewriting; (3) common control flow transformations [28] such as loop fission, fusion, interchange, unswitching, unrolling, tiling, or loop-invariant code motion.

The key ingredients of the OptiTrust framework are:

1. An internal library for manipulating a simplified abstract syntax trees (AST). OptiTrust currently handles a large subset of C99; it also supports most of OpenMP pragmas, C11 alignment directives, and a few C++ features, e.g., references. The original source code is encoded into the OptiTrust AST, and can be decoded back to C code.
2. A user-level library of general-purpose transformations. A fair number of these transformations are illustrated throughout the paper. This library is extensible with custom, domain-specific optimizations that can be programmed either by composing existing transformations, or, if needed, by manipulating directly the AST.
3. An interactive interface to execute a specific line, or block of lines, from the transformation script, and to interactively visualize the corresponding *diff*. We propose keybindings for the VSCode editor, yet our *bash*-based tooling could be easily invoked from any code editor.
4. The design of a concise, expressive system for targeting program points, somewhat similar to XPath [29] for XML, but specialized for an AST. Concretely, a target is described via a list of constraints over nodes or edges. OptiTrust also supports matching on the string representation of expressions, possibly using regular expressions.
5. The use of OCaml for expressing transformation scripts, as well as for implementing the framework. The OCaml programming language features: a type-system, making scripts easier to maintain; native compilation, ensuring decent performance; higher-order functions, facilitating code factorization; and pattern matching, helpful for implementing custom AST manipulations.
6. The possibility for transformation scripts to depend on parameters. Optimization steps can be guarded by a if-statement that depends on a combination of parameters. Doing so enables sharing of pieces of script when producing not just one but several optimized code, to be benchmarked on one or several target hardware.

The major part of the paper consists of the presentation of the case study that illustrates many of OptiTrust features. In the last section of the paper, we give a few insights on the implementation of the framework.

4 Case Study: Optimization of a PIC Code

4.1 Objectives of the Case Study

The optimized code that we aim to reproduce is a *Particle-in-Cell* (PIC) plasma physics simulation, implementing Vlasov-Poisson equations. This code has been written by Yann Barsamian as part of his PhD thesis [30], which was coadvised by experts in applied mathematics and experts in code optimization. His work, published at Euro-Par'18 [25], integrates state-of-the-art vectorization techniques [31–33].

Barsamian et al.'s algorithm [25] is *strict-binning*: particles from a same cell are stored in a *bag* data structure. Bags are represented as linked lists of fixed-size arrays, called *chunks*, with 256 particles in each. This representation is a variant of AOSOA [34]. A key feature is that the bags support not just sequential, but also thread-safe push operations. The details of the algorithm may be found in [25, §2]. Understanding them is not required for reading the present paper.

Barsamian's implementation [25] was shown to scale up to the simulation of 256 billion particles using 128 Skylake sockets, for a total of 3072 cores and 12.3 TB of RAM. In the scope of this case study, we ignore the distributed aspects of the code (MPI), and focus on the thread-parallelization and vectorization of the code. We also leave aside the optional custom memory allocator, which maintains (per-thread) free lists of chunks. Indeed, the corresponding code is fairly complicated, moreover entangled with other initialization functions. The primary purpose of this custom allocator is to avoid the need for dynamic allocation during the steps of the simulation. We did not want to modify Barsamian's code; and we did not want implement an allocator from scratch for our program, as it would introduce a divergence unrelated to OptiTrust transformations. Instead, we relied on *jemalloc*, a drop-in replacement for *malloc*, for chunk allocation.

We started by writing a *naive* C implementation of the same algorithm, without any consideration for performance, instead with the aim of making our naive code as close as possible to the mathematical equations. Fig. 1 shows our naive code for the function that processes one time step.

The purpose of the case study is to demonstrate how OptiTrust can be used to transform the naive code into a code equivalent to the hand-written, carefully optimized code. By equivalent, we mean a code that: (1) employs the same data structures, (2) features the same loops, in particular the same parallel loops and the same vectorized loops, (3) produces the same output, up to tolerable floating-point rounding differences, (4) exhibits at least as good performance, and (5) reads as easily as manually optimized code.

In what follows, we present several highlights from our OptiTrust script. Each section is accompanied by a figure showing the transformation script and the corresponding *diff*—or at least a representative excerpt thereof. Note that many transformations apply not just to the main `step` function, but at the same time: (1) in the function `stepLeafFrog`, which applies the Leap-Frog method; (2) in the function `addParticle`, used when creating the initial particles; (3) in the function `reportParticlesState`, used for dumping particles at the end of the simulation. In our script, the variable `step` refers to the main `step` function, and `steps` refers to either `step` or `stepLeafFrog`.

```
bag* bagsCur, bagsNext; double* deposit; // Key data structures
void step() { // The code executed at each time step of the simulation
  for (int idCell = 0; idCell < nbCells; idCell++) { // For each cell
    // Read the electric field at the corners of the cell
    vect_nbCorners field_at_corners = getFieldAtCorners(idCell, field);
    // Enumerate the particles stored in the cell considered
    bag* b = &bagsCur[idCell];
    bag_iter bag_it;
    for (particle* p = bag_iter_destructive_begin(&bag_it, b);
         p != NULL; p = bag_iter_next(&bag_it)) {
      // Interpolate the field based on the position to the cell corners
      double_nbCorners coeffs = cornerInterpolationCoeff(p→pos);
      vect fieldAtPos = matrix_vect_mul(coeffs, field_at_corners);
      // Compute the acceleration using the formula a = q/m*E
      vect accel = vect_mul(particleCharge / particleMass, fieldAtPos);
      // Compute the updated speed and position for the particle
      vect speed2 = vect_add(p→speed, vect_mul(stepDuration, accel));
      vect pos2 = vect_add(p→pos, vect_mul(stepDuration, speed2));
      pos2 = wrapArea(pos2); // Periodic grid: apply wrap−around
      particle p2 = { pos2, speed2 };
      // Push the particle in the bag of the destination cell
      const int idCell2 = idCellOfPos(pos2);
      bag_push(&bagsNext[idCell2], p2);
      // Deposit the charge at the corners of the destination cell
      double_nbCorners contribs = cornerInterpolationCoeff(pos2);
      accumulateChargeAtCorners(deposit, idCell2, contribs);
    }
    bag_init_initial(b); // Reinitialized the bag that has been consumed
  }
  // Swap bags of current step with bags of next time step
  for (int idCell = 0; idCell < nbCells; idCell++) {
    bag_swap(&bagsCur[idCell], &bagsNext[idCell]);
  }
  // Compute the electric fiedl based on the charge distribution using a
  updateFieldUsingDepositAndResetDeposit(); // Poisson solver
}
```

Figure 1. Unoptimized, bag-based code for the PIC method.

Each individual step of the script is introduced by a "`!!`" symbol. This symbol is followed by the name of the transformation, possibly by arguments provided to the transformation, and then by the target describing where in the code the transformation should be applied. The tilde symbol introduces optional arguments in OCaml. Targets are expressed as list of constraints, of the form `[c1; c2; .. ; cn]`. The modifier `nbMulti` indicates that we expect the target to resolve to more than one node.

4.2 Parallelization of the Main Loop

Let us start by illustrating the ability of OptiTrust to manipulate loop nests, to achieve a specific, nontrivial iteration order. For each cell, the inner loop of Fig. 1 enumerates the particles stored in the bag associated with that cell. The optimized code processes the cells in parallel, but not all of them at the same time. The idea, by Kong et al. [32], is to execute 8 phases, one after the other. During each of these phases, threads process cubic blocks that are spaced away from each other in parallel. The interest is to avoid data races for particles that move no further than half-a-block away from their block. Fig. 3 illustrates the block phases: in 2D, there are not 8 but 4 phases. At the phase i, each of the 2x2 blocks whose cells are labeled with i can be handled in parallel.

0	0	1	1	0	0	1	1	0	0	1	1
0	0	1	1	0	0	1	1	0	0	1	1
2	2	3	3	2	2	3	3	2	2	3	3
2	2	3	3	2	2	3	3	2	2	3	3
0	0	1	1	0	0	1	1	0	0	1	1
0	0	1	1	0	0	1	1	0	0	1	1
2	2	3	3	2	2	3	3	2	2	3	3
2	2	3	3	2	2	3	3	2	2	3	3

Figure 3. Parallel processing

Fig. 2 shows how we realize the parallelization, starting from the unoptimized, sequential code. First, we replace `nbCells` by the product `gridX*gridY*gridZ`. We target the loop with index `idCells` that contains the variable `accel`. Then, we apply the transformation `grid_enumerate` to turn the loop over cell indices into 3 loops over the coordinates. Names for the loop indices are provided with help of the higher function `map_dims`, a shorthand for `List.map f ["X";"Y";"Z"]`, defined for the case study.

Second, we set up the iteration by block. Let us show the transformation involved for the x-dimension. Starting from:

```
for (int iX = 0; iX < gridX; iX++)
```

we apply a tiling transformation by the block size (the grid size is assumed to be a multiple of the block size):

```
for (int bX = 0; bX < gridX; bX += block)
  for (int iX = bX; iX < bX + block; iX++)
```

then we apply to the outer loop a *coloring* transformation, for separately processing even and odd iterations:

```
for (int cX = 0; cX < 2; cX ++)
  for (int bX = cX*block; bX < gridX; bX += 2*block)
```

Finally, we reorder all the loops to iterate first on phases, then on blocks of that phase, then on cells within each block.

We are then ready to make the code parallel: we mark the loops on `bX`, `bY`, and `bZ` as collapsed-parallel, and turn the bag-push and the charge deposit into atomic operations. The conditional for performing non-atomic bag push operations is introduced in a subsequent step (not shown). In summary, by carefully composing general-purpose transformations, we obtain exactly Kong et al.'s advanced iteration pattern [32].

4.3 Inlining of Vector and Matrix Operations

In order to enable data-layout and data-scaling transformations, we convert high-level vector operations into per-field operations. Fig. 4 shows the corresponding steps. The first, preliminary step, eliminates two local variables by replacing them with in-place write operations. This step avoids the need for an auxiliary array when subsequently splitting the loop. The second step performs two actions. First, it inlines the vector multiplication and addition operations. Second, it converts assignment of a structure into the corresponding set of assignments to the individual fields of the structure. These steps highlight in particular the ability of OptiTrust to maintain human-readable code through inlining.

Step: change the iteration over the cells to introduce coordinates

```
let tg = cTarget [step; cFor "idCell" ~body:[cVarDef "accel"]] in
!! Instr.read_last_write ~write:[cWriteVar "nbCells"] [tg; dForStop; cReadVar "nbCells"];
!! Loop.grid_enumerate ~indices:(map_dims (fun d -> "i"^d)) [tg];
```

```
- for (int idCell = 0; idCell < nbCells; idCell++) {
+   for (int iX = 0; iX < gridX; iX++) {
+     for (int iY = 0; iY < gridY; iY++) {
+       for (int iZ = 0; iZ < gridZ; iZ++) {
+         const int idCell = (iX * gridY + iY) * gridZ + iZ;
```

Step: parallel processing of the cells by phases with spaced-out blocks

```
!! iter_dims (fun d -> let index = "b"^d in
     Loop.tile (expr "block") ~bound:TileBoundDivides ~index [step; cFor ("i"^d)];
     Loop.color (lit "2") ~index:("c"^d) [step; cFor index] );
!! Loop.reorder ~order:((add_prefix "c" dims) @ (add_prefix "b" dims) @ idims) [step; cFor "cX"];
!! Expr.replace_fun "bag_push_concurrent" [step; cFun "bag_push"];
!! Instr.set_atomic [step; cLabel "charge"; cWrite ()];
!! Omp.parallel_for ~collapse:3 [step; cFor "bX"];
```

```
- for (int iX = 0; iX < gridX; iX++) {
-   for (int iY = 0; iY < gridY; iY++) {
-     for (int iZ = 0; iZ < gridZ; iZ++) {
+   for (int cX = 0; cX < 2; cX++) {
+     for (int cY = 0; cY < 2; cY++) {
+       for (int cZ = 0; cZ < 2; cZ++) {
+         #pragma omp parallel for collapse(3)
+         for (int bX = cX * block; bX < gridX; bX += 2 * block) {
+           for (int bY = cY * block; bY < gridY; bY += 2 * block) {
+             for (int bZ = cZ * block; bZ < gridZ; bZ += 2 * block) {
+               for (int iX = bX; iX < bX + block; iX++) {
+                 for (int iY = bY; iY < bY + block; iY++) {
+                   for (int iZ = bZ; iZ < bZ + block; iZ++) {
```

```
-     bag_push(&bagsNext[MINDEX1(nbCells, idCell2)], p2);
+                     bag_push_concurrent(
+                       &bagsNext[MINDEX1(nbCells, idCell2)], p2);
      double_nbCorners contribs;
    charge:
-     for (int k = 0; k < 8; k++) {
+       #pragma omp atomic
        depositCorners[MINDEX2(nbCells, 8, idCell2, k)] +=
```

Figure 2. Steps: parallelization of the main loop over the cells

Step: eliminate an intermediate storage by reusing an existing one

```
!! Variable.reuse (expr "p->speed") [step; cVarDef "speed2" ];
!! Variable.reuse (expr "p->pos") [step; cVarDef "pos2"];
```

```
- vect speed2 = vect_add(p->speed, vect_mul(stepDuration, accel));
- vect pos2 = vect_add(p->pos, vect_mul(stepDuration, speed2));
- pos2 = wrapArea(pos2);
- particle p2 = {pos2, speed2, p->id};
- const int idCell2 = idCellOfPos(pos2);
+ p->speed = vect_add(p->speed, vect_mul(stepDuration, accel));
+ p->pos = vect_add(p->pos, vect_mul(stepDuration, p->speed));
+ p->pos = wrapArea(p->pos);
+ particle p2 = {p->pos, p->speed, p->id};
+ const int idCell2 = idCellOfPos(p->pos);
```

Step: inline vector operations, and convert struct assignments to per-field assignments

```
!! Function.inline [steps; cFuns ["vect_mul"; "vect_add"]];
let tg = cOr [[steps]; [cTopFunDefReg "bag_push_.*"]] in
!! List.iter (fun typ -> Struct.set_explicit [nbMulti; tg; cWrite ~typ ()]) ["particle"; "vect"];
```

```
- vect accel = vect_mul(particleCharge / particleMass, fieldAtPos);
+ vect accel = {particleCharge / particleMass * fieldAtPos.x,
+               particleCharge / particleMass * fieldAtPos.y,
+               particleCharge / particleMass * fieldAtPos.z};
- p->speed = vect_add(p->speed, vect_mul(stepDuration, accel));
+ p->speed.x = p->speed.x + stepDuration * accel.x;
+ p->speed.y = p->speed.y + stepDuration * accel.y;
+ p->speed.z = p->speed.z + stepDuration * accel.z;
- p->pos = vect_add(p->pos, vect_mul(stepDuration, p->speed));
+ p->pos.x = p->pos.x + stepDuration * p->speed.x;
+ p->pos.y = p->pos.y + stepDuration * p->speed.y;
+ p->pos.z = p->pos.z + stepDuration * p->speed.z;
```

```
-       c->items[index] = p;
+       c->items[index].pos.x = p.pos.x;
+       c->items[index].pos.y = p.pos.y;
+       c->items[index].pos.z = p.pos.z;
+       c->items[index].speed.x = p.speed.x;
+       c->items[index].speed.y = p.speed.y;
+       c->items[index].speed.z = p.speed.z;
```

Figure 4. Steps: inlining of vector operations, and expliciting of struct assignments

4.4 Array-of-Structures to Structure-of-Arrays

In the unoptimized code, a bag is represented as a linked list of *chunks*, where each chunk consists of a fixed-sized arrays of particles, and each particle consists of two vectors (`pos` and `speed`). In the optimized code, however, each chunk stores 6 fixed-sized arrays: x-positions, y-positions, z-positions, x-speeds, y-speeds, and z-speeds. These arrays can be efficiently processed by vector instructions (like with AOSOA [34]). Fig. 5 describes the steps involved in this conversion of the data representation from an array-of-structures to a structure-of-arrays (AoS-to-SoA). This transformation is a performance-critical data layout optimization that compilers respecting the C standard cannot apply.

To begin with, we need to reveal all the relevant array accesses. The first step of Fig. 5 replaces the loop that iterates over the bag cells using an iterator, with a pair of nested loops that directly access the internal representation of the chunks. For this transformation, we extended the bag library with two higher-order functions, named `bag_iter_ho_basic` and `bag_iter_ho_chunk`, describing the high-level and the low-level iteration patterns, respectively. The transformation employs pattern matching to recognize an instance of the body of `bag_iter_ho_basic`, then replaces it with the corresponding instance of the body of `bag_iter_ho_chunk`.

In the second step, we eliminate a local variable named `p`, which denotes the particle being processed. Doing so reveals accesses with the pattern, e.g., `c→items[i].speed.x`. In the third step, we convert this pattern into `c→itemsSpeedX[i]`. To that end, we reveal, in the type definition of `particle`, the components of the type `vect` in the fields `speed` and `pos`; then, we reveal, in the type `chunk`, the components of the type `particle` in the field `items`, which is a fixed-size array.

4.5 Scaling and shifting of values

We next explain how OptiTrust allows to apply transformations to numerical data stored in memory, for the purpose of simplifying computations. For simplicity, we consider a single dimension in the formulae, and we ignore the wraparound (i.e., the periodic conditions). Consider a particle at a given time step. Its speed, v, and its position, x, are updated. For the updated position, we need to compute the index of the destination cell, i, as well as the normalized position, r, relative to the corner of the cell i (with $0 \leq r < 1$). The value of r is used for interpolations. The formulae are as follows:

$$v \mathrel{+}= \frac{q}{m} E \Delta t \qquad x \mathrel{+}= v \Delta t \qquad i = \left\lfloor \frac{x}{\Delta x} \right\rfloor \qquad r = \frac{x - i\Delta x}{\Delta x}$$

where E denotes the electric field, Δt denotes the duration of a time step, Δx the width of a cell, and q and m are constants.

The idea of *scaling* is to store and manipulate not the values of E, v and x, but instead the values E', v' and x' defined as: $E' = \frac{q}{m} \frac{\Delta t^2}{\Delta x} E$ and $v' = \frac{\Delta t}{\Delta x} v$ and $x' = \frac{1}{\Delta x} x$. Using these entities, the previous formulae simplify as follows.

$$v' \mathrel{+}= E' \qquad x' \mathrel{+}= v' \qquad i = \lfloor x' \rfloor \qquad r = x' - i$$

Fig. 6 first shows the step that performs the scaling of positions. We provide as targets to the transformation the read operations (whose results are multiplied) and write operations (whose arguments are divided) on array cells and variables that store positions. Compared with applying scaling by hand, targeting reads and writes in a systematic way using a tool makes it much less likely to miss out instructions.

Fig. 6 next shows the step that performs the simplification of arithmetic expressions. OptiTrust features a module for performing basic simplification of arithmetic expressions (expand products, aggregate sums and products, including cancellation of divisions and subtractions of equal terms). Rather than specifying precisely the instructions to simplify, we found it easier to request simplification throughout the steps function, using the wrapper function `with_nosimpl` to exclude the one loop that contains operations to ignore.

After *scaling*, we apply *shifting*. The idea is to store coordinates not relative to the side of the grid, but instead relative to the side of the cell that contains the particle. Bowers et al. [31, Section III.E.] used this representation, storing, for each particle, the identifier of its cell plus the relative coordinates. Barsamian et al.'s algorithm [25], however, features one distinct bag per cell (*strict binning*), thus there is no need to store cell identifiers. Concretely, shifting amounts to introducing $x'' = x' - i$. The value of r is then immediately available, as x''. The absolute coordinate x' can be recovered as $x'' + i$. When moving a particle, we need to shift its coordinate by an amount that depends on the array cell in which the particle is about to be stored. We provided the relevant shifting values explicitly in the script, as this information would be quite challenging for a tool to automatically synthesize from the code in a robust manner.

4.6 Adding Dimensions to Matrices (Delocalize)

We next present another kind of data layout transformation, which introduces distributed and/or redundant memory storage. We call *delocalize* this transformation that, on a given scope, replaces accumulation operations on a matrix `a[i][j]` with the corresponding operations on a fresh, auxiliary matrix `d[i][j][k]`. This transformation takes as argument the scope considered, the name of the existing matrix `a`, a fresh name `d`, and a size K. The idea is that: `a[i][j]` = $\Sigma_{k \in [0,K)}$ `d[i][j][k]`. When entering the scope, `d[i][j][k]` is zero-initialized, except `d[i][j][0]` which is set to `a[i][j]`. When leaving the scope `a[i][j]` is set to the sum of the `d[i][j][k]` values. Within the scope, expressions of the form `a[i][j] += v` are replaced with `d[i][j][ANY(K)] += v`, where `ANY(K)` indicates that the programmer is there free to choose any valid index in the range $[0, K)$.

In the case study, we exploit delocalize 3 times. First, we use it to introduce redundant representation for the `deposit` array. We explain the details further on. Second, we use it to introduce a per-thread representation for the matrix `depositCorners`, leading to the introduction of the matrix

Step: changing the iteration pattern from an iterator to low-level loops

```
!! Function.inline [steps; cFuns ["bag_iter_begin"; "bag_iter_destructive_begin"]];
!! Loop.change_iter ~src:"bag_iter_ho_basic" ~dest:"bag_iter_ho_chunk" [steps; cVarDef "bag_it"];
!! Instr.delete [nbMulti; cTopFunDefAndDeclReg "bag_iter.*"];
```

```
bag_iter bag_it;
for (particle *p = bag_iter_destructive_begin(&bag_it, b); p != NULL;
     p = bag_iter_next(&bag_it)) {
```

```
+    for (chunk *c = b->front; c != NULL; c = chunk_next(c, true)) {
+      const int nb = c->size;
+      for (int i = 0; i < nb; i++) {
+        particle *p = &c->items[i];
```

Step: inlining of a variable that corresponds to a pointer on an array cell

```
!! Instr.inline_last_write [nbMulti; steps; cReadVar "p"];
```

```
particle *p = &c->items[i];
const int iX0 = int_of_double(p->pos.x / cellX);
const double rX0 = (p->pos.x - iX0 * cellX) / cellX;
```

```
+       const int iX0 = int_of_double(c->items[i].pos.x / cellX);
+       const double rX0 = (c->items[i].pos.x - iX0 * cellX) / cellX;
```

Step: array-of-structures to structure-of-arrays change in data layout

```
!! Struct.reveal_fields ["speed"; "pos"] [cTypDef "particle"];
!! Struct.reveal_field "items" [cTypDef "chunk"];
```

```
typedef struct chunk {
  struct chunk *next;
  int size;
  particle items[CHUNK_SIZE];
} chunk;
```

```
typedef struct chunk {
  struct chunk *next;
  int size;
+   double itemsPosX[CHUNK_SIZE];
+   double itemsPosY[CHUNK_SIZE];
+   double itemsPosZ[CHUNK_SIZE];
+   double itemsSpeedX[CHUNK_SIZE];
+   double itemsSpeedY[CHUNK_SIZE];
+   double itemsSpeedZ[CHUNK_SIZE];
} chunk;
```

```
c->items[i].speed.x = c->items[i].speed.x + stepDuration * accel.x;
c->items[i].speed.y = c->items[i].speed.y + stepDuration * accel.y;
c->items[i].speed.z = c->items[i].speed.z + stepDuration * accel.z;
```

```
c->itemsSpeedX[i] = c->itemsSpeedX[i] + stepDuration * accel.x;
c->itemsSpeedY[i] = c->itemsSpeedY[i] + stepDuration * accel.y;
c->itemsSpeedZ[i] = c->itemsSpeedZ[i] + stepDuration * accel.z;
```

Figure 5. Steps: array-of-structures to structure-of-arrays

Step: scaling of positions by a constant factor, amounting to normalizing grid cells

```
!! iter_dims (fun d -> let factor = var_mut ("cell"^d) in
    Accesses.scale ~factor [cTopFunDef "addParticle"; cFieldRead ~field:(lowercase_ascii d) ~base:[cVar "pos"] ()];
    Accesses.scale ~inv:true ~factor [nbMulti; steps; cOr [
      [sExprRegexp ~substr:true ("c->itemsPos"^d^"\\[i\\]")];
      [cFieldWrite ~field:("pos"^d) ()] ]]);
```

```
const int iX0 = int_of_double(c->itemsPosX[i] / cellX);
const double rX0 = (c->itemsPosX[i] - iX0 * cellX) / cellX;
```

```
+       const int iX0 = int_of_double(c->itemsPosX[i] * cellX / cellX);
+       const double rX0 = (c->itemsPosX[i] * cellX - iX0 * cellX) / cellX;
```

```
p2.posX = c->itemsPosX[i];
p2.posY = c->itemsPosY[i];
p2.posZ = c->itemsPosZ[i];
```

```
+       p2.posX = c->itemsPosX[i] * cellX / cellX;
+       p2.posY = c->itemsPosY[i] * cellY / cellY;
+       p2.posZ = c->itemsPosZ[i] * cellZ / cellZ;
```

Step: simplification of arithmetic expressions after scaling of speeds and positions

```
!! Variable.inline [steps; cVarDef "accel"];
!! Arith.with_nosimpl [nbMulti; steps; cFor "idCorner"] (fun () ->
    Arith.(simpl_rec expand) [nbMulti; steps]);
```

```
vect accel = {fieldAtPosX / (stepDuration * stepDuration) * cellX,
              fieldAtPosY / (stepDuration * stepDuration) * cellY,
              fieldAtPosZ / (stepDuration * stepDuration) * cellZ};
c->itemsSpeedX[i] = (c->itemsSpeedX[i] / (stepDuration / cellX) +
                     stepDuration * accel.x) *
                    (stepDuration / cellX);
```

```
+ c->itemsSpeedX[i] += fieldAtPosX;
```

```
c->itemsPosX[i] =
    (c->itemsPosX[i] * cellX +
     stepDuration * (c->itemsSpeedX[i] / (stepDuration / cellX))) /
    cellX;
```

```
+         c->itemsPosX[i] += c->itemsSpeedX[i];
```

Figure 6. Steps: excerpt from the scaling transformations

Step: introduction of OptiTrust matrix indexing operators (simplified presentation)

```
!! Matrix.intro_mops [nbMulti; cVarDefs ["deposit"; "bagsNext"]];
```

```
double *deposit = (double *)malloc(nbCells * sizeof(int));
for (int idCell = 0; idCell < nbCells; idCell++) {
  deposit[idCell] += 1.;
```

```
+   double *deposit = (double *)MMALLOC1(nbCells, sizeof(int));
    for (int idCell = 0; idCell < nbCells; idCell++) {
+     deposit[MINDEX1(nbCells, idCell)] += 1.;
```

Step: delocalize for the deposit array (simplified presentation)

```
!! Matrix.delocalize "deposit" ~into:"depositCorners" ~acc:"sum" ~ops:(Ast.Local_arith (Lit_double 0., Binop_add))
  ~dim:(var "nbCorners") ~index:"idCorner" ~indices:["idCell"] ~alloc_instr:[cVarDef "deposit"] [cFor "idCell"];
```

```
double* deposit = (double*)MALLOC1(nbCells, sizeof(int));
for (int idStep = 0; idStep < nbSteps; idStep++) {
```

```
double* deposit = (double*)MALLOC1(nbCells, sizeof(int));
for (int idStep = 0; idStep < nbSteps; idStep++) {
+   double* depositCorners = (double*)MALLOC2(nbCorners, nbCells, sizeof(int));
+   for (int idCell = 0; idCell < nbCells; idCell++) {
+     depositCorners[MINDEX2(nbCorners, nbCells, 0, idCell)] =
+         deposit[MINDEX1(nbCells, idCell)];
+     for (int idCorner = 1; idCorner < nbCorners; idCorner++)
+       depositCorners[MINDEX2(nbCorners, nbCells, idCorner, idCell)] = 0.;
+   }
```

```
  for (int idCell = 0; idCell < nbCells; idCell++) {
    const int nb = nbParticlesInCell(idCell);
    for (int i = 0; i < nb; i++) {
      const int idCell2 = computeParticleDestination(idCell, i);
      for (int k = 0; k < 8; k++) {

        deposit[MINDEX1(nbCells, indicesOfCorners(idCell2).v[k])] += 1.;

    }
  }
```

```
  for (int idCell = 0; idCell < nbCells; idCell++) {
    const int nb = nbParticlesInCell(idCell);
    for (int i = 0; i < nb; i++) {
      const int idCell2 = computeParticleDestination(idCell, i);
      for (int k = 0; k < 8; k++) {
+       depositCorners[MINDEX2(nbCorners, nbCells, ANY(nbCorners),
+                              indicesOfCorners(idCell2).v[k])] += 1.;
      }
    }
  }
+ for (int idCell = 0; idCell < nbCells; idCell++) {
+   double sum = 0.;
+   for (int idCorner = 0; idCorner < nbCorners; idCorner++) {
+     sum += depositCorners[MINDEX2(nbCorners, nbCells, idCorner, idCell)];
+   }
+   deposit[MINDEX1(nbCells, idCell)] = sum;
  }
+ MFREE(depositCorners);
```

Step: apply a bijection to the indexing pattern of the depositCorner array

```
!! Function.insert "int bij(int nbCells, int nbCorners, int idCell, int idCorner) { ... (* 14 lines not shown *) ...
      return MINDEX2(nbCells, nbCorners, bijection[idCorner], idCorner);" [tBefore; step];
!! Matrix.biject "bij" [step; cVarDef "depositCorners"];
!! Expr.replace (expr "MINDEX2(nbCells, nbCorners, idCell2, idCorner)")
         [step; sExpr "bij(nbCells, nbCorners, indicesOfCorners(idCell2).v[idCorner], idCorner)"];
```

```
for (int idCorner = 0; idCorner < nbCorners; idCorner++) {
  deposit[MINDEX1(nbCells,
                  indicesOfCorners(idCell2).v[idCorner])] +=
```

```
  for (int idCorner = 0; idCorner < nbCorners; idCorner++) {
+   depositCorners[MINDEX2(nbCells, nbCorners,
+                          indicesOfCorners(idCell2).v[idCorner],
+                          idCorner)] +=
```

```
for (int idCorner = 0; idCorner < nbCorners; idCorner++) {
  sum += depositCorners[MINDEX2(nbCells, nbCorners, idCell, idCorner)];
```

```
  for (int idCorner = 0; idCorner < nbCorners; idCorner++) {
+   sum += depositCorners[bij(nbCells, nbCorners, idCell, idCorner)];
```

Figure 7. Steps: excerpt from the delocalize transformations

`depositCornersThreads`. In this second case, delocalize is similar to the `reduction` pragma of OpenMP. Exploiting that OpenMP feature directly is not possible, because we subsequently need to fuse the aggregation loops introduced by two different delocalize operations. Third, we use delocalize to replace each destination bag with a pair of bags: one *private* bag for *non-atomically* receiving particles that come from nearby cells (at most half a block away from the currently processed block of cells [32]) and one *shared* bag for *atomically* receiving particles that come from further away [25, §2]. In that case, the aggregation is performed with respect to the function that initializes empty bags, and the (destructive) bag union function.

To ease array manipulations, OptiTrust provides macros for allocating and for accessing n-dimensional arrays represented as flat C arrays. `MALLOC2(X,Y,sizeof(double))` allocates a matrix of size `X*Y`, and `MINDEX2(X,Y,x,y)` refers to the coordinates `(x,y)` of a matrix of size `X*Y`. It is defined as `x*Y+y`. The first step from Fig. 7 shows the introduction of OptiTrust macros. The second step illustrates the application of a delocalize transformation that introduces redundant cells for storing the charge accumulated at every corner. Observe the expression `ANY(nbCorners)`, which we refine to `idCorner` in a subsequent step (not shown). For clarity, the script and *diff* shown in step 2 of Fig. 7 are associated with a unit test that exhibits a similar structure as the real code.

The motivation for introducing redundant cells for corners is that, by applying an appropriate shuffling of array cells, it is possible to deposit the charge contributions associated with one particle on the 8 corners of its destination cell via 8 write operations performed in *adjacent* memory locations [33]. The shuffling is implemented by the bijection function introduced at the third step of Fig. 7. The bijection is applied to all accesses to the array. By design, the write operation in the performance-critical loop simplifies to a simple expression

Step: vectorization of the interpolation function

```
let ctx = cTarget [nbMulti; cTopFunDef "cornerInterpolationCoeff"] in
!! Rewrite.equiv_at "double a; ==> 1. - a == (1. + (-1.) * a)" [nbMulti; ctx; cVarDefReg "c."; cInit()];
!! Rewrite.equiv_at "double a; ==> a == (0. + 1. * a)" [nbMulti; cWrite(); cVarReg "r[X-Z]"];
!! Variable.inline [nbMulti; ctx; cVarDefReg "c."];
!! Variable.intro_pattern_array ~const:true ~pattern_aux_vars:"double rX, rY, rZ"
   ~pattern_vars:"double coefX, signX, coefY, signY, coefZ, signZ"
   ~pattern:"(coefX + signX * rX) * (coefY + signY * rY) * (coefZ + signZ * rZ)" [nbMulti; ctx; cWrite(); dRHS];
!! Instr.move_out ~dest:[tBefore; ctx] [nbMulti; ctx; cVarDefReg "\\(coef\\|sign\\)."];
!! Loop.fold_instrs ~index:"idCorner" [ctx; cWrite()];
```

```
const double cX = 1. - rX;
const double cY = 1. - rY;
const double cZ = 1. - rZ;
double_nbCorners r;
r.v[0] = cX * cY * cZ;
r.v[1] = cX * cY * rZ;
r.v[2] = cX * rY * cZ;
r.v[3] = cX * rY * rZ;
r.v[4] = rX * cY * cZ;
r.v[5] = rX * cY * rZ;
r.v[6] = rX * rY * cZ;
r.v[7] = rX * rY * rZ;
```

```
double_nbCorners r;
+ for (int idCorner = 0; idCorner < nbCorners; idCorner++) {
+   r.v[idCorner] = (coefX[idCorner] + signX[idCorner] * rX) *
+                   (coefY[idCorner] + signY[idCorner] * rY) *
+                   (coefZ[idCorner] + signZ[idCorner] * rZ);
+ }
```

with 6 arrays of constant values being defined outside of the function, for example:

```
+ const double coefZ[8] = {1., 0., 1., 0., 1., 0., 1., 0.};
```

Figure 8. Steps: several instruction-level code optimizations

`depositCorners[MINDEX2(nbCells,8,idCell2,k)]`, which can be vectorized.

We are not aware of any previously existing tool that would be able to express or synthesize the combination of the *delocalize* and *bijection* transformations exploited here to achieve vectorization of the charge deposit process.

4.7 Instruction-Level Code Optimization

Fig. 8 illustrates a instruction-level optimization, performed by means of the pattern rewriting feature of OptiTrust. The optimization considered corresponds to the last ingredient of Vincenti et al.'s vectorization technique [33]. It consists of computing the interpolation coefficients for the 8 corners by introducing arrays of constants, and rewriting arithmetic expressions into a uniform pattern in such a way that the 8 computations can be factorized into a loop. This loop, unlike the original code, is sucessfully vectorized by GCC and ICC.

4.8 Other Steps from the PIC Transformation Script

In addition to the transformations discussed so far, the script of our case study includes the following important steps:

- The fusion of the two loops that iterate over the corners of the target cell, when accumulating charges.
- The optimization of the wrap-around computations. First, we replace the floating point modulo with an integer modulo—after scaling is applied to normalize the dimensions of grid cells. Second, in case the grid size is statically known to be a power of two, we replace the integer modulo with a bitwise-and operation.
- The introduction of a conditional in the code to execute either a non-atomic or an atomic bag push operation, depending on whether the destination of the particle is more than half a block away from its current block.
- An optimization of the aggregation of the private and shared bags at the end of the time step, to merge them directly into the bag used by the next iteration.
- The splitting of the main loop into three parts: one to update speeds, one to update positions, and one to push the particle in its destination bag and deposit its charge. Like in Barsamian et al.'s code [25], we introduce an auxiliary array of size `CHUNK_SIZE` to save identifiers of the destination cells across split loops.
- An transformation for representing relative positions in single precision instead of double precision, as suggested by Bowers et al. [31, III.E.]. This transformation is optional, and is controlled by a parameter.

Our full script consists of 150 steps marked with `!!`, organized in 34 *big steps*, each corresponding to one high-level optimization. For each small or big step, the *diff* can be displayed and reviewed. Currently, processing the full script takes less than 10 seconds. OptiTrust features a checkpoint mechanism, so when working on a specific big step, the *diff* of a given step gets displayed in less than 1 second.

Using a checker program, we could verify that both our unoptimized code and our optimized code behave essentially like Barsamian's code: the relative error is less than $2 \cdot 10^{-13}$ when simulating 1 million particles for 1000 steps. (We reproduced the experiment using 50 different seeds). Such a minor divergence is expected due to rounding errors.

4.9 Performance Evaluation

The primary purpose of our case study is not to report on performance improvement. What matters is the fact that

Socket description			Simulation		Throughput (million part./sec/core)		
	RAM	Cores	Part.	Grid	Orig.	Ours	Diff
#1	96GB	18	2000m	64^3	23.2	27.6	+19.1%
#2	16GB	10	200m	32^3	10.3	11.3	+9%
#3	32GB	4	200m	64^3	18.2	21.2	+16.5%

Figure 9. Benchmark results. Higher throughput is better.

OptiTrust allows to produce a code equivalent to one that had been manually optimized: (1) the output code is human readable; in particular, lines are short, and do not contain meaningless variable names; (2) the data structures are the same as in the original code: same arrays, same bags with fixed-sized arrays; (3) the loop structures are the same as in the original code, up to minor difference in the description of ranges; (4) the `omp` pragmas are the same as in the original code, and they appear at the corresponding places in the code; (5) loops from the `step` function are vectorized in the same way, according to GCC and ICC vectorization reports.

Our refinement from unoptimized code to optimized code lead to code whose memory load patterns appeared, in two places, to be beneficial. First, in the loop updating the speeds, our code reads the electric field associated with the cell only once outside the loop. Second, in the most critical loop (60% to 70% of the exec. time), our code loads particle data ahead of the conditional that tests whether the particle moves more than half a block away, whereas Barsamian's code loads it inside the branches, thereby delaying the memory loads. We did not want to modify Barsamian's code, nor to make our code less efficient, thus we kept these two differences.

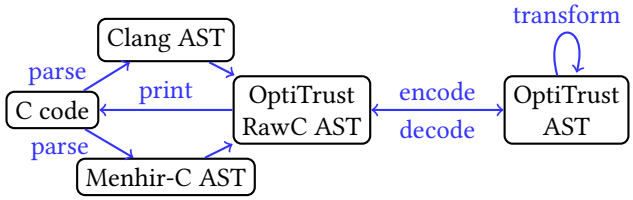
Barsamian et al. [25] report figures for a 24-core Xeon Platinum 8160 socket, with 96 GB of RAM. Their simulation involves 2 billion particles on a 64^3 grid, and runs for 500 steps. We checked that running for a few dozen steps suffices for performance measures; to be on the safe side, we used 100 steps in our benchmarks. We observed that execution times, even when using different seeds, vary by no more than 1% between runs on a server (up to 5% on a laptop). For each machine, we averaged execution times over 8 runs.

Our benchmark uses 3 different machines. Machine #1 hosts a 18-core Xeon Gold 6240 socket. Machine #2 hosts a 10-core E5-2650 socket; this machine has only 15 GB of RAM, so we used a 32^3 grid, and fewer particles. Machine #3 is a 4-core laptop (i7-8650U). Like Barsamian et al. [25], we compiled the code using Intel C compiler (ICC). They report a throughput of 30.8 million particles per seconds per core.

Results appear in Figure 9, which compares the unmodified, original code [25] (column Orig.), against the code produced by our transformation script (column Ours). We conclude that our output code delivers at least as good performance as the original code that we aimed to reproduce.

5 OptiTrust Infrastructure

The diagram that follows gives a high-level overview of OptiTrust's infrastructure. OptiTrust integrates two parsers: the C/C++ Clang parser [35]; and a faster, OCaml-based, C parser called Menhir-C [36].



The OptiTrust AST is simpler than a standard C AST. We convert the AST obtained from the selected parser into the OptiTrust AST by performing two encodings. First, we replace stack-allocated variables by heap allocated variables. Second, we eliminate the notion of modifiable left-value, replacing assignments with calls to a `set` functions. Together, these encodings lead, essentially, to an imperative λ-calculus. This language features much simpler semantics than C—it is used in many research papers on programming languages.

To get back from OptiTrust AST to C syntax, we leverage a small number of annotations introduced during the encoding phase, and systematically format the output code using clang-format. This way, the round-trip from C and back to C is the identity when no transformation is applied. This round-trip property ensures that *diff*s are minimal. Moreover, it gives confidence in the correctness of our translations.

An OptiTrust transformation is implemented in three stages. First, we resolve the provided target into a list of AST paths. Second, we mark the corresponding AST nodes (unless a single node is targeted, in which case we do not need marks). In general we are required to introduce marks because applying a transformation at the first target could move the positions of the other targets. Third, we apply the desired transformation at each of the marks. All transformation view the OptiTrust AST as an immutable data structure: they are implemented in a purely applicative way. Preventing in-place modifications avoids numerous pitfalls.

For resolving targets, OptiTrust features a backtracking algorithm. It is the user's responsibility to provide targets with constraints that avoid exponential searches. For constraints expressed in terms of the string representation of expressions, we are careful to generate string representations only when they are required, and only within the scopes that might contain the desired targets.

Overall, the OptiTrust implementation involves about 25k lines of OCaml code. The regression suite contains 170 unit tests, featuring 880 individual steps. Besides, we provide a generator that takes a transformation script and produces an interactive HTML page for navigating through the *diff*s associated with all the small and big steps involved in the script. This stand-alone page may be convenient for third-party reviewing of the optimization process.

References

[1] O. Krzikalla, K. Feldhoff, R. Müller-Pfefferkorn, and W. E. Nagel, "Scout: A source-to-source transformator for SIMD-optimizations," in *Euro-Par Workshops (2)*, ser. LNCS, vol. 7156. Springer, 2011.

[2] M. Kruse and H. Finkel, "A proposal for loop-transformation pragmas," *CoRR*, vol. abs/1805.03374, 2018. [Online]. Available: http://arxiv.org/abs/1805.03374

[3] J. Ragan-Kelley, C. Barnes, A. Adams, S. Paris, F. Durand, and S. Amarasinghe, "Halide: A language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines," in *Conference on Programming Language Design and Implementation*, 2013. [Online]. Available: https://doi.org/10.1145/2491956.2462176

[4] T. Chen, T. Moreau, Z. Jiang, L. Zheng, E. Yan, H. Shen, M. Cowan, L. Wang, Y. Hu, L. Ceze, C. Guestrin, and A. Krishnamurthy, "TVM: An automated end-to-end optimizing compiler for deep learning," in *OSDI*. USENIX Association, 2018. [Online]. Available: https://www.usenix.org/system/files/osdi18-chen.pdf

[5] B. Videau, K. Pouget, L. Genovese, T. Deutsch, D. Komatitsch, F. Desprez, and J.-F. Méhaut, "BOAST: A metaprogramming framework to produce portable and efficient computing kernels for HPC applications," *International Journal of High Performance Computing Applications*, vol. 32, no. 1, pp. 28–44, Jan. 2018. [Online]. Available: https://hal.archives-ouvertes.fr/hal-01620778

[6] J. R. Cordy, "The TXL source transformation language," *Science of Computer Programming*, vol. 61, no. 3, pp. 190–210, 2006.

[7] J. Lawall and G. Muller, "Coccinelle: 10 years of automated evolution in the Linux kernel," in *USENIX Conference on Usenix Annual Technical Conference*, ser. USENIX ATC '18. USENIX Association, 2018.

[8] O. Bagge, K. Kalleberg, M. Haveraaen, and E. Visser, "Design of the codeboost transformation system for domain-specific optimisation of C++ programs," in *Proceedings Third IEEE International Workshop on Source Code Analysis and Manipulation*, 2003, pp. 65–74.

[9] M. Bravenboer, K. T. Kalleberg, R. Vermaas, and E. Visser, "Stratego/XT 0.17. a language and toolset for program transformation," *Sci. Comput. Program.*, vol. 72, no. 1–2, p. 52–70, jun 2008. [Online]. Available: https://doi.org/10.1016/j.scico.2007.11.003

[10] D. Quinlan, "ROSE: Compiler support for object-oriented frameworks," *Parallel processing letters*, vol. 10, no. 02n03, pp. 215–226, 2000. [Online]. Available: https://digital.library.unt.edu/ark:/67531/metadc741175/m2/1/high_res_d/793936.pdf

[11] D. Quinlan and C. Liao, "The ROSE source-to-source compiler infrastructure," in *Cetus users and compiler infrastructure workshop, in conjunction with PACT*, vol. 2011, 2011, p. 1.

[12] C. Dave, H. Bae, S.-J. Min, S. Lee, R. Eigenmann, and S. Midkiff, "Cetus: A source-to-source compiler infrastructure for multicores," *Computer*, vol. 42, no. 12, pp. 36–42, 2009.

[13] M. Amini, "Source-to-source automatic program transformations for GPU-like hardware accelerators," Ph.D. dissertation, Ecole Nationale Supérieure des Mines de Paris, 2012.

[14] A. Konstantinidis, "Source-to-source compilation of loop programs for manycore processors," Ph.D. dissertation, Imperial College London, 2013.

[15] Y. Lebras, "Code optimization based on source to source transformations using profile guided metrics," Ph.D. dissertation, Université Paris-Saclay (ComUE), 2019. [Online]. Available: https://www.theses.fr/2019SACLV037.pdf

[16] U. Bondhugula, A. Hartono, J. Ramanujam, and P. Sadayappan, "A practical automatic polyhedral parallelizer and locality optimizer," in *PLDI'08 ACM Conf. on Programming language design and implementation*, 2008.

[17] L. Bagnères, O. Zinenko, S. Huot, and C. Bastoul, "Opening Polyhedral Compiler's Black Box," in *IEEE/ACM International Symposium on Code Generation and Optimization*, Mar. 2016.

[18] K. S. Namjoshi and N. Singhania, "Loopy: Programmable and formally verified loop transformations," in *Static Analysis - 23rd International Symposium, SAS*, ser. LNCS, vol. 9837. Springer, 2016.

[19] O. Zinenko, L. Chelini, and T. Grosser, "Declarative Transformations in the Polyhedral Model," Research Report RR-9243, Dec. 2018. [Online]. Available: https://hal.inria.fr/hal-01965599

[20] C. Chen, J. Chame, and M. W. Hall, "CHiLL: A framework for composing high-level loop transformations," University of Southern California, Technical Report 08-897, Jun 2008.

[21] G. Rudy, M. M. Khan, M. Hall, C. Chen, and J. Chame, "A programming language interface to describe transformations and code generation," in *Languages and Compilers for Parallel Computing*. Springer Berlin Heidelberg, 2011.

[22] J. Bispo and J. M. Cardoso, "Clava: C/C++ source-to-source compilation using LARA," *SoftwareX*, vol. 12, p. 100565, 2020. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S2352711019302122/pdf

[23] Q. Yi and A. Qasem, "Exploring the optimization space of dense linear algebra kernels," in *LCPC*, 2008.

[24] Q. Yi, Q. Wang, and H. Cui, "Specializing compiler optimizations through programmable composition for dense matrix computations," in *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO-47. USA: IEEE Computer Society, 2014, p. 596–608. [Online]. Available: https://doi.org/10.1109/MICRO.2014.14

[25] Y. Barsamian, A. Charguéraud, S. A. Hirstoaga, and M. Mehrenberger, "Efficient strict-binning particle-in-cell algorithm for multi-core SIMD processors," in *24th International Conference on Parallel and Distributed Computing (Euro-Par)*, ser. Lecture Notes in Computer Science, vol. 11014. Springer, Cham, 2018, pp. 749–763.

[26] I.-J. Sung, J. A. Stratton, and W.-M. W. Hwu, "Data layout transformation exploiting memory-level parallelism in structured grid many-core applications," ser. PACT '10, 2010, pp. 513–522.

[27] J. Allen and K. Kennedy, *Optimizing Compilers for Modern Architectures*. Morgan Kaufmann Publishers, 2002.

[28] M. Wolfe, *High performance compilers for parallel computing*. Addison-Wesley Publishing Company, 1995.

[29] J. Clark, S. DeRose *et al.*, "XML path language (xpath)," 1999. [Online]. Available: https://www.w3.org/TR/1999/REC-xpath-19991116/

[30] Y. Barsamian, "Pic-vert: A particle-in-cell implementation for multi-core architectures," Ph.D. dissertation, Université de Strasbourg, 2018. [Online]. Available: https://hal.archives-ouvertes.fr/tel-01940700

[31] K. J. Bowers, B. J. Albright, L. Yin, B. Bergen, and T. J. T. Kwan, "Ultrahigh performance three-dimensional electromagnetic relativistic kinetic plasma simulation," *Physics of Plasmas*, vol. 15, no. 5, p. 055703, 2008.

[32] X. Kong, M. C. Huang, C. Ren, and V. K. Decyk, "Particle-in-cell simulations with charge-conserving current deposition on graphic processing units," *Journal of Computational Physics*, vol. 230, no. 4, pp. 1676–1685, 2011.

[33] H. Vincenti, M. Lobet, R. Lehe, R. Sasanka, and J.-L. Vay, "An efficient and portable SIMD algorithm for charge/current deposition in particle-in-cell codes," *Computer Physics Communications*, vol. 210, pp. 145–154, 2016.

[34] Intel and A. K. Sharp, "Memory layout transformations," Nov. 2013, compiler Methodology for Intel MIC Architecture https://www.intel.com/content/www/us/en/developer/articles/technical/memory-layout-transformations.html.

[35] C. Lattner, "LLVM and Clang: Next generation compiler technology," in *The BSD conference*, vol. 5, 2008.

[36] J.-H. Jourdan and F. Pottier, "A simple, possibly correct LR parser for C11," *ACM Transactions on Programming Languages and Systems*, vol. 39, no. 4, pp. 14:1–14:36, Aug. 2017. [Online]. Available: http://cambium.inria.fr/~fpottier/publis/jourdan-fpottier-2016.pdf