==========
Summaries for the seventh meeting of the seminar "Automatic Parallelization in the Polyhedral Model"
Topic: "Code Generation in the Polyhedral Model Is Easier Than You Think"
==========
Summary 1 of 3

A polytope model is based on the linear algebra representation but the complexity of the code generation does not it to use for optimizing compilers. Moreover several algorithms require limitations on reordering function and even complexity of the real world programs hinders the integration. This paper presents how to make it possible in reasonable amount of time and in limited size.

Starting from the simple transformation, we define the scattering function for each of the loop nest which partition the iteration domain(separating into disjoint polyhedra).Then take the list of polydera scan to recursive generate each level of AST into scanning code i.e generate the scanning code for first dimension and resurse on the next dimension. Each recursion builds an AST node list describe by the polyhedra and list of its children. and project the resulting polyherda onto the outer dimensions. Generating code form AST is quite simple by constraints labeling and each node can be directly translated to loop bounds and surrounding conditional respectively, if the constraint concern the dimension corresponding node level or not.

Finally we separate these projections into different polyhedra and then remove the dead code. But separating polyhedral may isolate some points to ensure this problem and will not result in polyheron peeling. So it is always be good to compute separation before. Moreover removing of isolated points:- By first defining the point candidate to merge with node, then check, if a point directly follows the node in the ordering graph and cannot be merged, that implies that an input polyhedron is not convex. If not ,merge the point candidates with the node and remove the points from lists of polyhedral.

There are various complexity Issues like separation into disjoint polyhedra, e.g. for list on n polyhedra worst case complexity is $O(3^n)$.Also the memory consumption is too high. To remove these issues paper presents the solution that by the use of pattern matching to reduce the number of polyhedral computations .i.e at the given depth domains are often same or disjoint. And for memory use, instead of separating the projections onto different polyhedral , we can merge them when their intersections are not empty.

Thus algorithm shows practically that generating a code directly without redundant code is far more efficient than try to improve the naive one, although converting program from scattering function takes time.

Questions
1.)Limitation on reordering function (unimodular or invertible), how does these affect code?

2.)What is rational target, and how automatic selection of scattering function?

3.)For saving memory, instead of separating, we merge projections when their intersections are not empty. Show?

==========
Summary 2 of 3

This paper by C. Bastoul tackles the problem of code generation in the polyhedral model. While the abstractions provided by the polyhedral model are very useful to reason about program transformations, compilers have trouble generating good code out of an optimized representation. Oftentimes, the reordering function provided by a polyhedral optimizer does not fit the criteria of the compiler, hampering the opportunities for good optimizations to arise. Second, a reordering may not fit into naive loop building constructs, leading to inefficient code, sometimes rendering the whole optimization process meaningless.

To solve this problem, the author does not constrain the transformation (scattering) functions any further, since it is not concerned with a basis change of the polyhedra. Instead, the approach uses the old scattering function and polyhedron to define a new polyhedron. The points in the resulting polyhedron are ordered lexicographically until all dimensions of the source polyhedron are exhausted, after which new dimensions are added (in no particular order). The additional dimensions carry the transformation data. The author argues that this solves the problem of code generation with non-invertible transformations. Furthermore, the new constraint system will only have integer solutions, imposed by the lexicographic order. In cases where the transformation itself is rational, the author introduces auxiliary variables to rewrite the quotient into a integral form.

The final part of the approach is concerned with polyhedra scanning. Code generation for a polyhedron means

that we need to find a set of (nested) loops, which visit each point in the polyhedron exactly once in the order imposed by the polyhedron. The author adapts (or extends) an algorithm from Quilleré et al. – fixing critical parts in the process, such as high complexity and code explosion. The original algorithm recursively splits the set of polyhedra into disjoint subsets, where it generates the corresponding loop nests from the outer- to the innermost levels. The extension will first compute the relevant part of the polyhedra by intersecting them with the outer "context", i.e. the previously computed dimensions, or for dimension one, the bounds of the parameters. Then it will project the polyhedra onto the first dimension and then separate them into disjoint polyhedra, after which the scanning code for the first dimension can be generated. The algorithm proceeds recursively trough each dimension, repeating the process for each subset. Each polyhedron is intersected with the outer context, yielding the relevant subset, and then projected onto the context. Finally, these projections will be separated into disjoint polyhedra. After generating the code for the final dimension, the algorithm terminates.

However, even the proposed algorithm is still exponential in the worst case, since the separation takes up to 3^n polyhedral operations which are exponential themselves. By using pattern matching on the domains and separating trivial cases this can be reduced by a factor of two in practice. To avoid memory problems, the algorithm sometimes defaults to a more naive, but less memory intensive approach.

==========
Summary 3 of 3

Code generation is the last step of polyhedral code optimizations and was formally known as bottleneck not only regarding code quality (and therefore possible speedup) and allowed transformations but also compilation time.

Bastoul describes a new approach to tackle all those problems. Instead of restricting the applied transformations to be unimodular, invertible, integral or uniform, a general affine scheduling/transformation function is possible. He states that the quality of the produced code is high as long as memory con- sumption during the compilation is not an issue. If it is, the presented algorithm might produce more complex code but with a smaller memory footprint.

The main idea during the process is to lift the target polyhedron into a higher space each time former approaches used special characteristics of their restricted scheduling functions. The produced target polyhedron has more dimensions but it does not suffer from "holes" or rational coefficients. It also allows non-uniform transformations as e.g., conditionals with modulo expression guards.

The translation from target polyhedra to source code is done in a recursive fashion dimension by dimension. During this process the major computation and memory consumption is caused by splitting a list of polyhedra into disjoint polyhedra. At this point the algorithm makes a trade-off if the memory footprint becomes to large. It may add complex control flow (conditionals) instead of computing all disjoint polyhedra. During their evaluation this case almost never occurred. Bastoul also states that code bloat is handled better than in earlier code generation algorithms as singleton polyhedra (therefore conditionals) can often be removed in a post processing step.

The evaluation in the end shows a drastic decrease of compilation time, but the actual code quality is not evaluated further.

Questions and thoughts:

1. On modern machines parallelization of the actual code generation (or at least of the main com- putation kernel) could be quite interesting. Thus, it it possible to compute the disjoint polyhedra for one/all step in parallel?

2. When we look at Figure 6, is it possible that the code generation changes the scheduling or at least has so much freedom to choose different ones? This aims towards the possibly different stack distances (not taken into account for the sake of e.g., smaller code).

3. Is there some data on the impact of instruction cache pollution vs. control flow overhead? It seems both goals are contradicting and the produced code "should" optimize for execution time rather than code size if necessary.