==
Summaries for the seventh meeting of the seminar "Automatic Parallelization in the Polyhedral Model"
Topic: "A Practical Automatic Polyhedral Parallelizer and Locality Optimizer"
==
Summary 1 of 3

The authors of the paper introduce PluTo, a fully automated source-to-source program optimizer from C to OpenMP parallel code. In particular, it seeks to optimize for locality and parallelism for arbitrarily nested loops using the polyhedral model. The focus of this approach lies in finding good ways to perform tiling, i.e. in separating the iteration spaces into (concurrently executable) parts which have good data locality features, in an affine transformation framework.

The framework itself uses LooPo to determine the dependence polyhedra and domains, which are the input into the transformation framework for parallelization and locality optimization described in this paper. The output, a set of statement-wise affine transformations, is fed into ClooG for code generation.

The theoretical framework itself is based on the legality of tiling multiple domains with affine dependences. In particular, tiling of a polyhedral is said to be legal as long as each tile can be executed atomically and there is no cyclic dependency between tiles (or rather, if there exists a total execution order). In order to argue about optimality of data locality, a cost function is needed. For any dependence, the cost functions will measure the number of hyperplanes traversed along the normal. This measures the communication needed between tiles, or in a sequential loop, the reuse distance. Minimizing the cost function will lead to nonlinear programs if the input was not uniform or if there was a self-reference for some statement. Using a bounding functions and the Farkas' Lemma makes it possible to get an ILP formulation. Solving the LP results in a solution for each coefficient used by the optimal mappings of each statement. Since the authors want at least as many linearly independent solutions as the dimension of its domain, they augment the ILP by additional constraints which enforce linear independence between successive solutions.

The algorithm for tiling itself specifies a higher-dimensional domain and transformations for the tile loops. This is accomplished by creating so-called supernodes for all iterators that appear in each hyperplane and adding corresponding constraints. These supernodes correspond to the loops over the tiles spaces. Furthermore, the authors want to allow tiling multiple times, to account for different levels (sizes) of caches and for parallelism. For each such level, the scattering functions are duplicated and a supernode is added – preserving the legality constraints of the transformation system. The generated structure can be directly used to give a parallel schedule, as all dependences are forward and non-negative. Computing the tiles in a wavefront schedule therefore satisfies all dependency constraints. In order to facilitate vectorization, tiles can be reordered such that the parallel loops are executed at the innermost position in a nest without hurting the legality constraints.

==
Summary 2 of 3

The paper presents a tool "PLuTo", that is an end-to-end fully automatic code optimizer. It is based on integer linear optimization framework, that tiles the code for parallelism and locality using affine transformations. Tiling is grouping of points in iteration space into smaller blocks, helps to reuse the code when the blocks fit in the faster memory.

The first thing is to define the legality condition for the tiling, i.e. the tiling of a statement's iteration space by a set of hyperplanes is said to be legal if each tile can be executed automatically and a valid total reordering of the tiles can be constructed, that means no two tiles dependent on each other. However, if it is imposed only on dependences that have not been satisfied up to a certain depth.

Another point is cost function which gives number of hyperplanes the dependence traverses along the hyperplane normal to the affine transformation and minimization of the function produce an objective non-linear in loop variable and hyperplane coefficients. We need at least as many independent solutions as the dimension of its domain so linearly independent hyperplanes are found iteratively till all the dependencies are satisfied.

For design considerations input dependences must be bounded , also while developing a linearly independent sub-space for each statement , there are number of choices so to solve this problem we construction of linearly independent sub-space for each statement transformation.

In this approach of optimizind code that models tiling in te transformation framework by finding the affine transformation that make rectangular tiling in the transformation space legal and for transformation that may convert to imperfectly nested code, polyhedral tiling is method to get the tiled code form the code generator in one pass guaranteeing legality nest since we find tiling hyperplanes , there may not be a single loop that satisfies all dependences.

for the process of transformation the sequences of nested loop as input to the LoopPO scanner + dependence tester that produce dependence polyhedra. Next parallelization and locality optimization is applied and then statement wise affine transformation is given to polyhedral tile specified. The output of this step is applied to CLooG, that can scan a union of polyhedra an optionally new global lexicographic specified as through scattering function. And then we provide this output to the syntactic transformer that produces OpenMP parallelized code . Moreover as the part of a post process, we move the parallel loop within a tile innermost and make use of ignore dependence pragmas to explicitly force vectorization.

Hence this tool is fully automatic polyhedral source-to-source program optimizer, and it is not just for C/Fortran but also for the language from which polyhedral can be extracted and analyzed.

Question:

what is dependence upto certain depth?

===
Summary 3 of 3

By combining several research tools and integrating an own framework to find affine transformations Bondhugula et al. created a source-to-source automatic polyhedral parallelizer and locality optimizer.

To find a good schedule they propose a cost function which unifies communication and locality optimizations, thus each dimension from outer to inner will have the lowest possible communica- tion/synchronization cost which also minimizes the reuse distance for this dimension. When comparing their (complete) framework (pluto) to other approaches in the field, they state that for automatic par- allelization the cost function is a key piece and may/should be extended in the future.

Their scheduling approach is heavily based on tiling, supposed to be fully automatic and general applicability is also desired, therefore it is not surprising they integrated the actual tiling directly into their scheduling framework instead of a post pass on the AST. The modifications on old loop bounds and the new loops over the tiles are are encoded in the iteration domains of the target space. Legality is guaranteed by the model and the correctness of the code generator.

Since their primary goal is to find tiling hyperplanes parallelization might become a problem without considering pipeline parallelism. For this matter they present an algorithm to perform unimodular transformations on their additional loops (supernodes). It will produce pipeline parallel dimensions with only forward dependencies by summarizing the schedules for this loops. At the same time this does not change the tile shapes and introduces only little code complexity.

After some implementation details were given to reduce the search space and therefore counter possible exponential growth with additional statements, the comparison with other tools begins and is concluded with evaluation on various computation intensive kernels. In this part the uniqueness of their automatic approach is stated again together with the need of a accurate cost function even if somehow provided by an “expert”. It is also mentioned that comparison of different tools is hard due to the lack of access and restrictions on the input.

Questions and notes?

• Why is the evaluation given in GFLoPs and how are they to be interpreted?
• The evaluation for parallel execution always uses extremely large problem sizes.