Two-Phase Global Loop Transformations

Sven Verdoolaege[*] Francky Catthoor
Maurice Bruynooghe Gerda Janssens

1 Introduction

As the memory subsystem typically accounts for over 50% of the power consumption, optimizing the global memory accesses of an application is crucial for achieving low power realizations. This is especially true for multi-media systems such as medical image processing and video compression algorithms, which typically manipulate large multi-dimensional arrays resulting in a very large amount of data storage and transfers. Improving the global memory accesses generally also has a positive influence on the performance because it reduces the (external) bus traffic and it improves the cache hit rates.

The Data Transfer and Storage Exploration (DTSE) methodology aims to solve this global optimization problem. The methodology is split into several substeps combined in two groups: platform independent and platform dependent steps. The platform independent steps transform the program independently of the parameters of the memory (data storage) target platform, which is, in effect, chosen or constructed based on the results of these steps and subsequently used to further optimize the program in the platform dependent steps. The global loop transformation step is one of the platform independent steps that aims to optimize global data transfer and storage by increasing the access regularity and locality of the program. This step uses a polyhedral model and is itself composed of two substeps: linear transformation and translation.

2 Program model

The programs we consider consist of a number of statements, each enclosed in a (possibly zero-dimensional) loop nest and possibly guarded by conditionals (`if`-statements), with all the loop bounds and conditions affine expressions of outer loop iterators and structural parameters. The *iteration domains*, i.e., the sets of iterators for which each statement is executed, can then be represented by (unions of) polytopes (convex sets bounded by hyperplanes). In this representation, each dimension corresponds to a loop iterator.

To optimize the memory usage of a program, we need to investigate the relation between the accesses to memory in the statements of the program. We say that iteration $\vec{j}$ of statement B depends on iteration $\vec{\imath}$ of statement A if they both access the same array element and if the former is executed after the latter. We distinguish four kinds of dependences: input (read, read), flow

[*] `sven@cs.kuleuven.ac.be`

or true (write, read), anti (write, read) and output (write, write). We also say that a pair of dependent statements exhibit reuse [5], more specifically, temporal reuse. It is called self reuse if the two statements are the same, i.e., the statement depends on a previous iteration of the same statement, and group reuse otherwise. Spatial reuse (either self or group) exists when two statements access array elements that are close to each other, i.e., elements that are likely to be in the same cache line or memory page.

The *dependence relation* δ of a pair of statements combines all individual dependences between iterations of the two statements. If the index expressions, mapping iterator values to array indices, are affine in the enclosing loop iterators, then the dependence relation can also be represented by a polytope. During the translation step, we use a simpler abstraction known as the *dependence polytope* D. It is the convex hull of the *dependence (distance) vector*s between two statements, where a dependence vector is the difference in loop iterators of two depending iterations. I.e., $D = \operatorname{conv}\{\vec{\jmath} - \vec{\imath} \mid (\vec{\imath}, \vec{\jmath}) \in \delta\}$. Next to statements that directly depend on each other, we also consider statements that depend on each other through a dependence chain, which results in an *indirect dependence.*

3 Our approach

We perform loop transformations by transforming each statement iteration polytope by an affine function in two steps. In the first step, we determine the linear part of the affine transformation and in the second step the offset.

The second step is currently mainly focused on group locality [3]. By decreasing the distance over a dependence, we increase the chance of finding the array element in a memory hierarchy level closer to the processor during the second access. Decreasing the distance between the first and the final reference to an array element also reduces the lifetime of the element which may result in a reduction in final memory requirement for the array. Other optimization criteria during the second step are related to the estimates of the final physical memory hierarchy mapping, namely the memory requirements for multiple simultaneously alive arrays and data reuse considerations. To ensure legality of the transformation, the final distance vectors need to be lexicographically positive. In practice, this means we only need to consider the lexicographically minimal element of the dependence polytope, rather than the whole set.

Since the linear transformation precedes the translation step, it needs to ensure a valid choice still exists. A sufficient condition is that the (possibly indirect) distance vectors over a circuit are lexicographically positive after the linear transformation. This in turn can be translated into conditions on the dependence relations, which are changed as a result of the transformation. Here too, we need not consider the dependence relations themselves as the information available in their affine hulls is sufficient. As to optimality, we concentrate on the locality of self dependences and the regularity of group dependences [4]. The locality of self dependences can be optimized by selecting a linear transformation for the statement involved that places the self reuse in the inner loop(s). Regularity is a property that increases with the decrease in variance between the individual distance vectors and can be measured by the dimension of the dependence polytope. Regularity can be optimized by selecting a pair of linear transformations for the statements involved in the dependence that optimizes this dimension. If not all irregularity can be removed, we attempt to place it in the innermost loop(s). By optimizing regularity, we increase the opportunity of the next step to optimize group locality. In the future, other criteria related to the estimated mapping cost on the final physical memory hierarchy (such as data reuse info) may have to be introduced here too. Experiments will have to verify this.

The translation step has been implemented as a SUIF1 compiler pass. We are currently refining the linear transformation step and we plan to implement it as well. This will allow us to further investigate the relation between the optimization criteria in order to improve our algorithms such that they do not produce a single solution, but rather expose the trade-offs involved among the factors, e.g., memory size and number of accesses, that contribute to the overall power consumption. These trade-offs can then be used in subsequent steps to select or construct the target platform.

4 Related work

Many researchers have used some form of affine-by-statement scheduling, including Darte and Robert, Feautrier and Lim and Lam. They mainly focused on parallelism, although recently locality has also been considered [2]. They do not investigate trade-offs between the different optimization criteria discussed above, and typically perform the affine transformation in a single step. By splitting into two steps we hope to reduce the complexity of the transformation. In this sense, our approach is similar to that of Kelly and Pugh [1], who also provide a search tree of legal transformation in which to search for an optimal solution. The main difference with their approach is that they construct the affine transformation row by row. Our second step is a combination of loop fusion and loop shifting, which has also been investigated by Manjikian and Abdelrahman, Fraboulet et al., Song et al. and Darte and Huard.

References

[1] Wayne Kelly and William Pugh. Finding legal reordering transformations using mappings. In *Languages and Compilers for Parallel Computing*, pages 107–124, 1994.

[2] Amy W. Lim, Shih-Wei Liao, and Monica S. Lam. Blocking and array contraction across arbitrarily nested loops using affine partitioning. *ACM SIGPLAN Notices*, 36(7):103–112, 2001.

[3] Sven Verdoolaege, Maurice Bruynooghe, Gerda Janssens, and Francky Catthoor. Multi-dimensional incremental loop fusion for data locality. In Danielle Martin, editor, *IEEE 14th International Conference on Application-specific Systems, Architectures and Processors*, The Hague, The Netherlands, June 2003.

[4] Sven Verdoolaege, Koen Danckaert, Francky Catthoor, Maurice Bruynooghe, and Gerda Janssens. An access regularity criterion and regularity improvement heuristics for data transfer optimization by global loop transformations. In *1st Workshop on Optimization for DSP and Embedded Systems, ODES*, March 2003.

[5] M. E. Wolf and M. S. Lam. A Data Locality Optimizing Algorithm. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'91)*, pages 30–44, June 1991.