Memory Allocation and Array Contraction using the Polyhedral Model in GeCoS

Antoine Morvan, CAIRN Team, IRISA, antoine.morvan@irisa.fr

September 9, 2014

Note: This document assumes the reader is familiar with optimizing compilation using the polyhedral model. For newcomers, I recommend the OpenScop User Guide [1].

Contents

1 Introduction **1**

2 Problem Construction – Conflict and Difference Set **2**
2.1 Input Definition . . . 3
2.2 Dependencies and Domain Selection . . . 5
2.2.1 Selection for Memory Reallocation . . . 5
2.2.2 Selection for Array Contraction . . . 5
2.3 Conflict Set Construction . . . 6
2.3.1 Working in the Scheduled Space . . . 6
2.3.2 Deriving the Conflict Set . . . 6
2.4 Difference Set Construction . . . 7
2.5 Special Cases . . . 7
2.5.1 Parallel Schedules . . . 7
2.5.2 Multiple Writes per Statement . . . 7

3 Successive Modulo Technique **8**
3.1 Basic Principle . . . 8
3.2 Extensions . . . 8

4 Other Approches **8**

A Partial Unrolling and Distributed Memory **8**

---

1 Introduction

The polyhedral model allows complex loop transformations for optimizing different performance criteria, such as parallelism or memory access locality [1, 2, 3], on a subset of programs known as ACL (Aff Control

[1] http://icps.u-strasbg.fr/~bastoul/development/openscop/docs/openscop.pdf

Loops). Once a loop nest has been scheduled, it is also possible to minimize the memory footprint required to store the intermediate results computed during the scheduled loop nest execution.

Minimizing the memory footprint can be decomposed in three steps :

1. identifying which write accesses can be re-allocated;
2. constructing the problem inputs (i.e., the *conflict sets*, see 2.1);
3. solving the problems, resulting in the new access functions and array sizes.

The goal of this document is to synthesize the existing works [4, 5], and to provide an implementation oriented explanation of the different problematics. The representation of all objects in the polyhedral model is presented in the ISL [6] formalism.

Note : for the sake of simplicity, the copies of ISL objects are omitted.

Identifying Intermediate Results

Of course, because it would alter the program semantic, one cannot change the allocation of the input values, nor the allocation of the output values (i.e., the liveout values). It is simple to determine what are the input values (array dataflow analysis answers $\perp$). However, the ACL representation does not support "what happened before and what happens after" the ACL under analysis. It is therefore not possible to know what value will be *liveout* at the end of the ACL execution, and what value is only an intermediate result.

From a higher position, this information can be computed before extracting the ACLs from an AST-like representation of programs, even though it is not possible to compute it accurately in the general case. Therefore, in the remaining of the document, we assume that the values that can be reallocated are specified by annotations, either on the array that stores the values, or the statement that produces the values.

Intuition – Successive Modulo Summary

The basic idea behind memory reallocation is to first build the set of all the pairs of operations for which a conflict holds, namely the pairs of operations whose value is alive at the same time. This set can be built exactly and is called the *conflict set*.

Then, from this conflict set, one can build the set containing all the differences between each side for each pair. This is the *difference set* representing the lexicographic distance between each operation which conflicts. For instance, if iterations $(0,0)$ and $(1,0)$ produce a value that is alive at the same time, $((0,0),(1,0))$ and $((1,0),(0,0))$ are in the conflict set (it is symetric), and $(1,0)$ and $(-1,0)$ are in the difference set.

By computing the lexicographic maximum of the difference set (per dimension, see 3.1), we obtain the size of the array (each maximum on a dimension defining an *expansion degree*). The access function for the new array is the scheduled one, modulo the expansion degree.

---

2 Problem Construction – Conflict and Difference Set

The problem of minimizing the memory footprint after scheduling can be seen from two point of view, namely *array contraction* and *memory (re)allocation*. They result in different problem constructions, but the same solving technique can be used in both cases. We consider that there is only one write per statement (section 2.5.2 gives hint for cases where several writes occur per statement).

Array Contraction

One way of seing the problem is to consider that the transformation has to keep array names. In such a case, the schedule usually respects the memory dependencies, and the original memory allocation remains legal. Minimizing the memory footprint then consists in contracting existing arrays, and is called *array contraction*.

However, the given schedule may not respect the memory dependencies (write after read and/or write after write), but only the dataflow dependencies (read after write). Therefore, the original memory allocation may not be legal for the schedule, and the minimum required memory size can be larger than the original memory allocation. Finding this minimum size in such a case is called *partial array expansion*. It is simply a variant of array contraction where the result may be larger than the original memory allocation.

Memory Allocation

The other point of view is to consider that the memory will be completely reallocated. The starting point is a *fully expanded program* in single assignment, namely the result of array dataflow analysis [1]. From there, original array names are completely ignored, and the problem simply consists in minimizing the memory footprint.

2.1 Input Definition

The following subsections explain how to build the conflict set for both array contraction and memory allocation, starting from the PRDG given by the array dataflow analysis, and various information comming from the AST:

- The PRDG: a graph where nodes represent statements, with their associated iteration domains, and edges represent dependencies between statements, with their associated relation and validity domain. This PRDG can be represented as a `isl_union_map` and a `isl_union_set` in ISL (cf. example).
- The writes: a relation that associates, for each statement, what are the written arrays, and for what index function. It can be represented using a single `isl_union_map`.
- The reads: idem, but for reads.
- The schedule: new logical dates for the ACL operations. Represented as a single `isl_union_map`, where the image is a single `isl_set` (all the ranges belong to the same name space).

Example 1 : Gaussian filter

```
void gauss(int M, int N, int in[N][M], int out[N][M]) {
  int i, j;
  int tmp[N][M];
  for (i = 1; i < N - 1; i++)
    for (j = 0; j < M; j++)
S0:   tmp[i][j] = 3 * in[i][j] - in[i - 1][j] - in[i + 1][j];
  for (i = 1; i < N - 1; i++)
    for (j = 1; j < M - 1; j++)
S1:   out[i][j] = 3 * tmp[i][j] - tmp[i][j - 1] - tmp[i][j + 1];
}
```

Figure 1: Example ACL, representing a 3×3 Gaussian filter

The ACL from Figure 1 leads to the following objects in the polyhedral model (ISL notation, details of construction skipped):

- Domains: $[N, M] \rightarrow \{$
  $S0[0, i, j] : 0 \leq j \leq M - 1 \ \& \ 1 \leq i \leq N - 2;$
  $S1[1, i, j] : 1 \leq j \leq M - 2 \ \& \ 1 \leq i \leq N - 2\}$
- PRDG: $[N, M] \rightarrow \{$
  $S1[1, i, j] \leftrightarrow S0[0, i, j'] : j - 1 \leq j' \leq j + 1\}$
  Note : it is a relation.
- writes: $[N, M] \rightarrow \{$
  $S0[0, i, j] \rightarrow tmp[i, j];$
  $S1[1, i, j] \rightarrow out[i, j]\}$
- reads: $[N, M] \rightarrow \{$
  $S0[0, i, j] \rightarrow in[i - 1, j]; S0[0, i, j] \rightarrow in[i, j]; S0[0, i, j] \rightarrow in[i + 1, j];$
  $S1[1, i, j] \rightarrow tmp[i, j - 1]; S1[1, i, j] \rightarrow tmp[i, j]; S1[1, i, j] \rightarrow tmp[i, j + 1]\}$

On top, we use the following schedule (partial loop fusion) :

- schedule: $[N, M] \rightarrow \{$
  $S0[0, i, j] \rightarrow [i, j];$
  $S1[1, i, j] \rightarrow [i, j + 2]\}$

Example 2 : Forward Substitution

```
void forward_subst(int N, int b[N], int L[N][N], int y[N]) {
  int i, j;
  for(i = 0; i < N; i++) {
S0: y[i] = b[i];
    for(j = 0; j < i; j++)
S1:   y[i] = y[i] - L[i][j] * y[j];
S2: y[i] = y[i] / L[i][i];
  }
}
```

Figure 2: Example ACL, representing a forward substitution on square system matrix

The ACL from Figure 2 leads to the following objects in the polyhedral model (ISL notation, details of construction skipped):

- Domains: $[N] \rightarrow \{$
  $S0[i, 0, 0] : i >= 0 \ \& \ i <= -1 + N;$
  $S1[i, 1, j] : j >= 0 \ \& \ j <= -1 + i \ \& \ i >= 0 \ \& \ i <= -1 + N;$
  $S2[i, 2, 0] : i >= 0 \ \& \ i <= -1 + N\}$
- PRDG: $[N] \rightarrow \{$
  $S1[i, 1, 0] \rightarrow S0[i, 0, 0];$
  $S1[i, 1, j] \rightarrow S1[i, 1, j - 1] : j >= 1;$
  $S1[i, 1, j] \rightarrow S2[j, 2, 0];$
  $S2[0, 2, 0] \rightarrow S0[0, 0, 0];$
  $S2[i, 2, 0] \rightarrow S1[i, 1, i - 1] : i >= 1\}$
- writes: $[N] \rightarrow \{$
  $S0[i, 0, 0] \rightarrow y[i];$
  $S1[i, 1, j] \rightarrow y[i];$
  $S2[i, 2, 0] \rightarrow y[i]\}$

- reads: $[N] \rightarrow \{$
  $S0[i,0,0] \rightarrow b[i];$
  $S1[i,1,j] \rightarrow L[i,j]; S1[i,1,j] \rightarrow y[i]; S1[i,1,j] \rightarrow y[j];$
  $S2[i,2,0] \rightarrow L[i,i]; S2[i,2,0] \rightarrow y[i]\}$

On top, we use the following schedule (identity) :

- schedule: $[N] \rightarrow \{$
  $S0[i,0,0] \rightarrow S0[i,0,0];$
  $S1[i,1,j] \rightarrow S1[i,1,j];$
  $S2[i,2,0] \rightarrow S2[i,2,0]\}$

2.2 Dependencies and Domain Selection

The first step consists in selecting dependencies and domains related to the statement/array that needs to be reallocated. Basically it sums up to selecting nodes and edges in the PRDG.

Note : special attention is required in the case where several writes can occur in a single statement, see 2.5.2.

2.2.1 Selection for Memory Reallocation

For the memory allocation, the analysis is computed per statement.

TODO

The resulting objects are:

- $PRDG_S$ is a subset of the PRDG, containing only the dependencies where the statement S is the producer;
- D_S is the iteration domain of S.

TODO: example

2.2.2 Selection for Array Contraction

For the array contraction, the analysis is computed per array. For contracting an array `t`, the analysis has to focus on the dependencies involving `t` only. Selecting the dependencies involving `t` only can be done by filtering the PRDG :

1. build the list of the statements reading `t` : $lreads$ (test the tuple name)
2. build the list of the statements writing `t` : $lwrites$ (test the tuple name)
3. $PRDG_t = PRDG \cap lreads \times lwrites$
4. $D_t = lwrites$

The resulting objects are:

- $PRDG_t$ is a subset of the PRDG, containing only the dependencies involving the array `t`;
- D_t is the iteration domains of all statements writing in `t`.

TODO: example

2.3 Conflict Set Construction

Once the dependencies have been filtered for the particular statement/array, it is possible to compute the *conflict set*, namely the set of all pairs of operations in the scheduled ACL for which there is a *conflict*. A conflict holds iff, given two existing operations $\vec{w_1}$ and $\vec{w_2}$:

$$CS = \vec{w_1} \bowtie \vec{w_2} : \{[\vec{w_1}] \to [\vec{w_2}] : \vec{w_1} \preceq lastuse(\vec{w_2}) \ \& \ \vec{w_2} \preceq lastuse(\vec{w_1})\}$$

that is if the last use of the value produced by $\vec{w_2}$ is after $\vec{w_1}$, and the last use of the value produced by $\vec{w_1}$ is after $\vec{w_2}$.

2.3.1 Working in the Scheduled Space

Starting from the previously filtered PRDG (either $PRDG_S$ or $PRDG_t$, noted $PRDG_f$) and domain (either D_S or D_t, noted D_f) the first step consists in representing $PRDG_f$ and D_f in the scheduled space. This will make the analysis valid given the schedule:

1. $scheduledPRDG_f = schedule \circ PRDG_f$
   `isl_union_map_apply_range(isl_union_map_apply_domain(`$PRDG_f$`,schedule),schedule)`
2. $scheduledD_f = schedule[D_f]$
   `isl_union_set_apply(`D_f`,schedule)`

The scheduled space unifies all the statements. Therefore, the scheduled object above represent `map / set` instead of `union_map / union_set`[2]. Also, let n be the number of dimensions in the scheduled space.

2.3.2 Deriving the Conflict Set

In the scheduled space, the conflict set CS (built as a relation here) is derived as follows:

1. $lastUse = lexMax(scheduledPRDG_f^{-1})$
   `isl_map_lexmax(isl_map_reverse(`$scheduledPRDG_f$`))`
   The function $lastUse$ associates, to a write, its last use (read) in the scheduled PRDG.
2. $lexGE = \{\vec{x} \leftrightarrow \vec{y} \mid \vec{x} \succeq \vec{y}\}$ with $lexGE \subseteq \mathbb{Z}^n \times \mathbb{Z}^n$
   `isl_map_lex_ge(isl_set_get_space(`D_f`))`
   This relation represents the lexicographic predecessors.
3. $predWrites = lexGE \cap \mathbb{Z}^n \times ran(scheduledPRDG_f)$
   `isl_map_intersect_range(`$lexGE$`,isl_map_range(`$scheduledPRDG_f$`))`
   This relation associates, to a read, all the writes preceding it.
4. $CS_{part1} = predWrites \circ lastUse$
   `isl_map_apply_range(`$lastUse$`,`$predWrites$`)`
   This relation associates, to a write, all the writes executed before its last use.
5. $CS_{part2} = CS_{part1} \cap CS_{part1}^{-1}$
   `isl_map_intersect(`CS_{part1}`,isl_map_reverse(`CS_{part1}`))`
6. Since CS is symetric, any write conflicts with itself:
   $CS = CS_{part2} \cup (\{\vec{x} \leftrightarrow \vec{y} \mid \vec{x} = \vec{y}\} \cap D_f \times D_f)$
   `tmp1 = isl_map_lex_eq(isl_set_get_space(`D_f`))`
   `tmp2 = isl_map_intersect_domain(isl_map_intersect_range(tmp,`D_f`),`D_f`)`
   `isl_map_union(`CS_{part2}`,tmp2)`

In order to build a set ($\subseteq \mathbb{Z}^{2n}$) out of this relation:
`isl_map_get_range(isl_map_move_dims(`CS`,isl_dim_out,0,isl_dim_in,0,`n`))`

[2]Use functions isl_union_set_get_set_at($scheduledD_f$,0) / isl_union_map_get_map_at($scheduledPRDG_f$,0)

2.4 Difference Set Construction

The difference set represents the set of differences of elements in the conflict set:

$$DS = \{\vec{u} \mid \vec{u} = \vec{x} - \vec{y} \wedge (\vec{x}, \vec{y}) \in CS\}$$

It is built as follows:

1. $DS_{r1} = \{CS \leftrightarrow \mathbb{Z}^n\}$
   `tmp = isl_map_move_dims(`CS`, isl_dim_in, 0, isl_dim_out, 0, `n`)`
   `isl_map_insert_dims(tmp, isl_dim_out, 0, `n`)`
2. $sub = \{(\vec{x}, \vec{y}) \rightarrow \vec{x} - \vec{y}\} \subseteq (\mathbb{Z}^n \times \mathbb{Z}^n) \times \mathbb{Z}^n$
   `sub = isl_basic_map_universe(isl_map_get_space(`DS_r1`))`
   `lspace = isl_local_space_from_space(isl_map_get_space(`DS_r1`))`
   `for (`$i : 0 \rightarrow n$`)`
   `.  cst = isl_equality_alloc(lspace)`
   `.  cst = isl_constraint_set_constant_si(cst,0);`
   `.  cst = isl_constraint_set_coefficient_si(cst,isl_dim_in, i, 1);`
   `.  cst = isl_constraint_set_coefficient_si(cst,isl_dim_in, i+`n`, -1);`
   `.  cst = isl_constraint_set_coefficient_si(cst,isl_dim_out, i, -1);`
   `.  sub = isl_basic_map_add_constraint(sub, cst);`
3. $DS_{r2} = DS_{r1} \cap sub$
   `isl_map_intersect(`DS_{r1}`, sub)`
4. $DS = ran(DS_{r2})$
   `isl_map_range(`DS_{r2}`)`

2.5 Special Cases

2.5.1 Parallel Schedules

In the step 6 of the conflict set derivation (cf sub section 2.3.2), instead of building the lexicographic equiality function, apply equality only for non parallel dimensions. Apply no constraint on parallel dimensions.

This means that, on a parallel dimension, all writes conflict with each other.

2.5.2 Multiple Writes per Statement

The polyhedral model allows the representation of multiple writes per statement. This case occurs, for instance, when using pure functions[3] in statements. It is a particular case where the above techniques does not apply, because the isolated dependencies may be wrong. Let us examine the example loop nest of Figure 3. For such a loop nest, the dependencies per array are:

- on array `x` : $d_1 : [N] \rightarrow \{S[i,j] \rightarrow S[i-1,j]\}$
- on array `y` : $d_2 : [N] \rightarrow \{S[i,j] \rightarrow S[i-2,j]\}$

The PRDG for this example would have only one edge with an associated relation instead of a function:

$$d : [N] \rightarrow \{S[i,j] \leftrightarrow S[i',j] : i - 2 \leq i' \leq i - 1\}$$

Indeed, using this relation for the `y` array would lead to the correct conflict set, because the lexicographic maximum of d is d_2. However, it is obvious that this same maximum, when considering array `x`, results in an overestimated distance, although conservative.

[3]Pure functions do not have any side effect.

```
//Pure function : without side effect
void pureFunction(int* out1, int* out2, int in1, int in2) {
   *out1 = foo(in1, in2); //other pure
   *out2 = bar(in1, in2); //functions
}
void main() {
  ...
  for (i = 1 .. N)
    for (j = 0 .. N)
   /* the semantic we use assume the
    * derefenced locations are written */
S:    pureFunction(&x[i][j],&y[i][j],x[i-1][j],y[i-2][j]);
   /* another notation would be */
S:    (x[i][j],y[i][j]) = pureFunction(x[i-1][j],y[i-2][j]);
  ...
}
```

Figure 3: Example of an ACL involving pure functions where multiple writes occurs in one statement.

Computing accurate distances when statements may issue several writes requires a more precise representation of the dependencies, where edges are associated per read/write access instead of per statement (GeCoS provides such information through the `org.polymodel.dataflow` meta-model).

---

3 Successive Modulo Technique

3.1 Basic Principle

3.2 Extensions

Let us consider the memory allocation for the fully expanded program, and assigning one array per write. If a single read access in the original program uses a value that can be produced by differents writes, the new memory allocation implies guarded accesses. Example ... TODO

---

4 Other Approches

See *Lattice Based Memory Allocation* [5].

A Partial Unrolling and Distributed Memory

References

[1] P. Feautrier. Dataflow Analysis of Array and Scalar References. *International Journal of Parallel Programming*, 20(1):23–53, 1991.

[2] M. W. Benabderrahmane, L.-N. Pouchet, A. Cohen, and C. Bastoul. The Polyhedral Model is More Widely Applicable Than You Think. In *Proceedings of Int. Conf. on Compiler Construction*, pages 283–303. Springer, 2010.

[3] U. Bondhugula, A. Hartono, J. Ramanujam, and P. Sadayappan. PLuTo: A Practical and Fully Automatic Polyhedral Program Optimization System. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, Tucson, AZ, June 2008. ACM.

[4] V. Lefebvre and P. Feautrier. Optimizing Storage Size for Static Control Programs in Automatic Parallelizers. In Christian Lengauer, Martin Griebl, and Sergei Gorlatch, editors, *Euro-Par'97 Parallel Processing*, volume 1300 of *Lecture Notes in Computer Science*, pages 356–363. Springer Berlin Heidelberg, 1997.

[5] C. Alias, F. Baray, and A. Darte. Bee+Cl@k: an Implementation of Lattice-Based Array Contraction in the Source-to-Source Translator ROSE. In *LCTES*, pages 73–82, 2007.

[6] S. Verdoolaege. ISL: An Integer Set Library for the Polyhedral Model. In *ICMS*, pages 299–302, 2010.