Integer maps

Integer maps are binary relations between integer set.

The first set in the relation is called domain.
The second set is the range or the output.

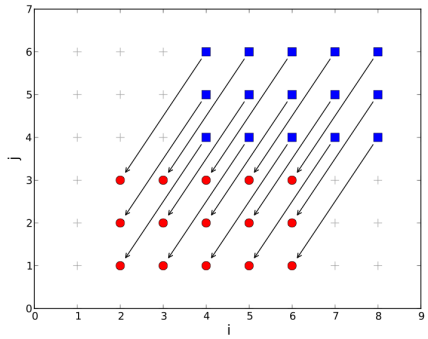


Figure 3: A two-dimensional integer map

Figure 3 illustrates the integer map $M = \{(i, j) \to (i-2, j-3)\}$ with the input values restricted to the elements contained in the blue rectangular of Figure 1. Each black arrow represents a relation between one input tuple and one output tuple. The input values (blue squares) shown are the very same values as illustrated in Figure 1. The output values (red circles) are the same values, but translated according to M.

The general form of an integer map is:

$$M = \{\vec{i} \to \vec{o} \in \mathbb{Z}^{d_1} \times \mathbb{Z}^{d_2} \mid f(\vec{i}, \vec{o}, \vec{p})\}$$

input tuples — output tuples — Presburger formula

Presburger formula that evaluates to true, iff $\vec{i}$ and $\vec{o}$ are related in M for given parameters $\vec{p}$. Integer maps can represent arbitrary relations and can, contrary to what the use of the "$\to$" notation suggests, relate multiple output values to a single input value. The Presburger formulas that describe integer maps follow the rules presented for

Named unions sets / named union maps

named integer sets are integer sets containing named tuples.
eg $\{S(i,j)\}$ an integer set with the name "S".

named integer maps: integer maps between two named spaces.

named union set, an integer set contains tuples from different spaces. eg $\{S_1(i,j); S_2(i)\}$

named union maps: binary relations between integer sets.

2.2 MODEL AND TRANSFORM IMPERATIVE PROGRAMS

Polyhedra or, in our case, integer sets and maps can be used to model "sufficiently regular" compute programs with the goal to reason about and precisely control higher-level properties without distraction from imperative or lower-level constructs. To do so the individual statement instances in a program (i.e., each dynamic execution of a statement inside a loop nest), their execution order as well as the individual array elements accessed are modeled, analyzed and transformed, whereas control flow constructs, loop induction variables, loop bounds or array subscripts are hidden and only regenerated when converting a transformed loop nest back to imperative constructs.

sidenote: A set can be approximated by computing various hulls (convex, affine, simple, polyhedral.)

Polyhedral representation

what kind of higher-level property?

1. Goal: reason about and precisely control higher-level property without distraction from imperative or low-level construct.

2. What is modeled?
   1. individual statement instances
   2. their execution order
   3. individual array element accessed

3. What is hidden?
   1. control-flow construct
   2. loop induction variables
   3. loop bounds
   4. array subscripts

Let's see the illustrative example

```
   for (i = 1; i <= n; i+=1)      ⟶ i-loop
     for (j = 1; j <= i; j+=1)    ⟶ j-loop
S:     A[i][j] = A[i-1][j] + A[i][j-1];
```

S: ↑ compute statement

To model this computation, 4 data structures are constructed

1. iteration space (I) ← integer set, describes the set of statement
2. schedule (S) ← integer map, assigns to each statement a possibly multi-dimensional time.
3. a relation of read-accesses Aread
4. a relation of must-write-access Awrite

→ defines memory locations

defines execution order

```
  for (i = 1; i <= n; i+=1)
    for (j = 1; j <= i; j+=1)
S:    A[i][j] = A[i-1][j] + A[i][j-1];
```

we get the following model

statements execution order

$$I = \{S(i,jj) \mid 1 \leqslant j \leqslant i \leqslant n\}$$ integer set

$$S = \{S(i,j) \to (i,j)\}$$ integer map assigned to statement instance.

$$A_{read} = \{S(i,j) \to A(i-1,j); S(i,j) \to A(i,j-1)\}$$

$$A_{write} = \{S(i,j) \to A(i,j)\}$$

integer map assigned to statement the memory locations they accesses

memory location

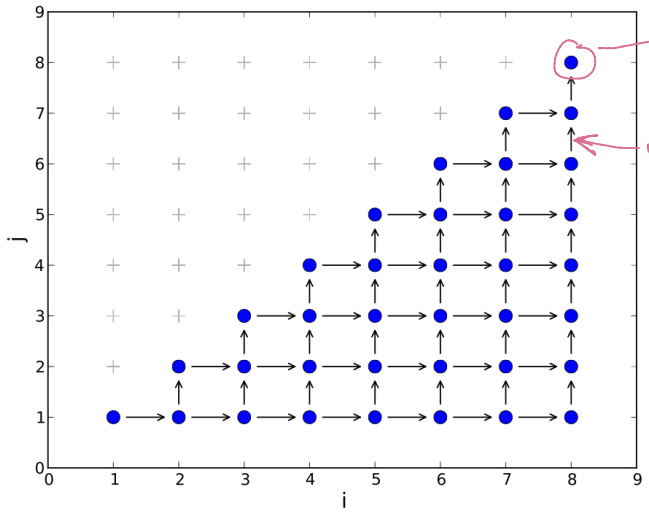


Figure 4: Iteration Space – Unmodified

The illustration of I in Figure 4 represents each statement instance with a single dot. The arrows between such statement instances illustrate data flow dependences modeled by an integer map $D = \{S(i,j) \to S(i,j+1); S(i,j) \to S(i+1,j)\}$. Those dependences relate statement instances with the statement instances they depend on. Computing precise data-flow dependences [54, 104, 92] is one analysis that is significantly facilitated by the use of an integer set based representation.

* Compute precise data-flow dependences is one analysis that is significantly facilitated by the use of an integer set based representation.

improve data-locality based on polyhedral representation,

ensure that statement instances that operate on the same data elements are executed in close time proximity.

2. each statement instance is always mapped to a point in time that is later than the execution time of all statement instances it depends on.

within these constraints we are free to modify the schedule.

tiling is in this case both legal and effective. To implement loop tiling we define a new schedule

$$S' = \{S(i)(j) \to (\lfloor i/3 \rfloor)(\lfloor j/3 \rfloor)(i)(j)\}$$

which defines an execution order where the statement instances are always executed in blocks of size 3×3. The new execution order is illustrated in Figure 5 and is shown in two levels. At the higher level, the blue blocks are executed in lexicographic order. At the lower level, within the individual blue blocks, the statement instances are again executed in lexicographic order. As statement instances that are close by are placed in the same block, they are also executed close by in time.

one advantage of integer set representation

compute precise data-flow dependencies.

1. 2.

static control part / program

A SCoP is a program (region) that consists of a set of *statements* possibly enclosed by (not necessarily perfectly) *nested loops* and *conditional branches*. Within this region read-only scalar values are called *parameters*. The statements in the SCoP are side effect free, besides explicit *reads* and *writes* to multi-dimensional arrays or scalar values. Loops are regular loop bounds with a single lower and a single upper bound and increments by fixed, positive integers (`i+=10`). Both loop bounds and array accesses are (piecewise-quasi) affine expressions in terms of parameters and induction variables of outer loops.

imperative program that can be translated into polyhedral representation

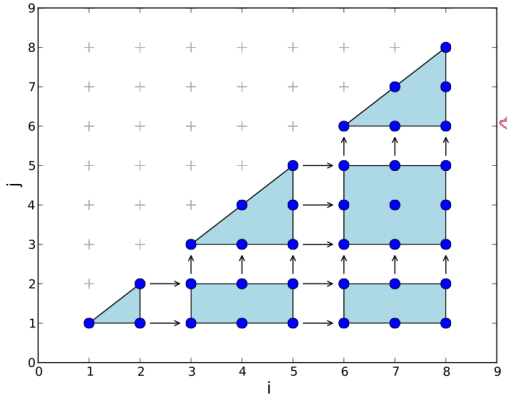
perfectly fit into RNN computation

I: iteration space is an integer set

S: schedule → assign each statements a multi-dimensional time

How to understand locality? → statement instances operating on the same data elements are executed in close time proximity.

Loop Tiling to increase locality



← scheduling leads to new execution order

Figure 5: Iteration Space – Tiled

tiling is in this case both legal and effective. To implement loop tiling we define a new schedule

$$S' = \{S(i)(j) \rightarrow (\lfloor i/3 \rfloor)(\lfloor j/3 \rfloor)(i)(j)\}$$

materialize the transformations back to imperative codes

↑

polyhedral AST generator

The polyhedral representation

The representation we use to model SCoPs consists of the following components:

1. ITERATION SPACE/DOMAIN The set of statements instances that are part of a SCoP. It is modeled as a named union set, where each named component of the union set describes a statement, with individual instances of a statement being described by the elements contained in the corresponding named set.

2. ACCESS RELATIONS A set of read, write and may-write access relations relate statement instances to the data-locations they access. These access relations are modeled as named union maps.

3. DEPENDENCES A relation between statement instances that defines restrictions on the execution order, due to producer-consumer relationships or the shared usage of certain data locations. Data dependences are modeled as named union maps.

4. SCHEDULE An execution order which assigns each statement instance a multi-dimensional execution time. One statement instance is executed before another statement instance, if its execution time is lexicographically smaller.

It is important to note that there is a strong separation between the statement instances themselves and the order in which they are executed. Program optimizations that do not change the set of statements that are executed, but only change the order they are executed in, consequently only affect the schedule.

Note for schedule:

1. program optimizations only change the order that statements are executed.
2. a schedule is only valid if it is of a form such that all data-dependences go forward in time.
   the time at the source of the dependence must be lexico-graphically strictly smaller than the time at the target.

3. 1. given a set of dependence, we can compute a schedule.
   2. given an ① iteration space ② access relations, ③ a schedule, the corresponding data dependences can be compute.

Transformations

We can group loop transformations according to what program properties are modified:

regularly modeled with integer set

- The order in which computations are performed
- The loop structure, but not the order of computation
- The data layout and the data locations accessed
- The computation (algorithmic changes)

The first three kinds of transformations are regularly modeled with integer sets, whereas modeling algorithmic changes is more difficult, but possible in some cases [140]. Integer sets are also used for accelerator programming and vectorization, but such transformations are mostly a combination of execution reordering and data layout transformations supplemented with the generation of specialized target instructions or library calls. We will use such GPU code generation

Classical transformations that reorder computations are

1. fusion
2. fission
3. reversal
4. interchange
5. strip-mining
6. skewing

→ can be modeled by schedule transformations.

combinations of such elementary transformations yield tiling or unroll and jam.

1. Fusion

Fusion

$$\mathcal{I} = \{S1(i) : 1 \leqslant i < n;\ S2(i) : 1 \leqslant i < m\}$$

iteration space (named union set)

original schedule

$$\mathcal{S} = \{S1(i) \to (0,i);\ S2(i) \to (1,i)\}$$

named union map

how to calculate this?

transformation map

$$\mathcal{T} = \{(0,i) \to (i,0);\ (1,i) \to (i,1)\}$$

named union map

$$\mathcal{S}_T = \{S1(i) \to (i,0);\ S2(i) \to (i,1)\}$$

transformed schedule

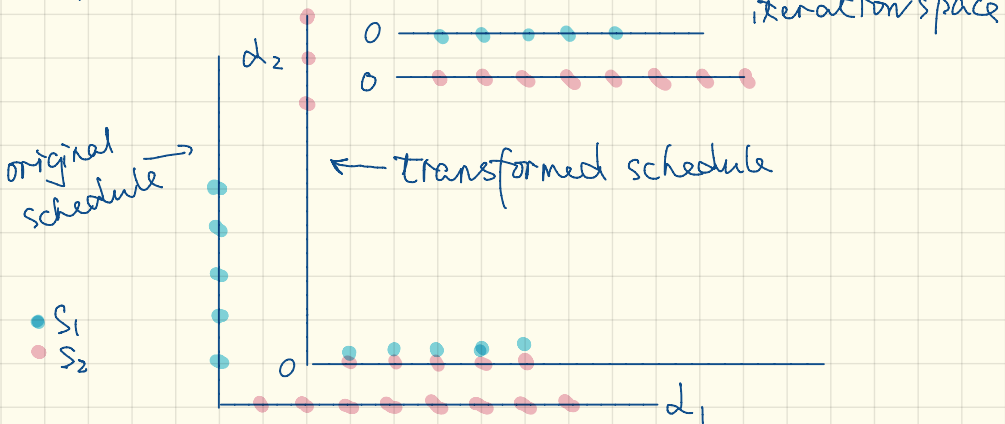
```
// Original loops
for (i=1; i<n; i+=1)
  S1(i);

for (i=1; i<m; i+=1)
  S2(i);

// Fused loops
for (i=1; i < min(n,m); i+=1) {
  S1(i);
  S2(i);
}
for (i=max(1,m); i<n; i+=1)
  S1(i);
for (i=max(1,n); i<m; i+=1)
  S2(i);
```

Notes for the above example.

1. There are two spaces.
2. schedule assigns each statement instance a multi-dimensional execution time, here 2-d is used.



2. Fission

Fission

$$\mathcal{I} = \{S1(i) : 1 \leqslant i < n;$$
$$S2(i) : 1 \leqslant i < n\}$$
$$\mathcal{S} = \{S1(i) \to (i, 0);$$
$$S2(i) \to (i, 1)\}$$
$$\mathcal{T} = \{(i, 0) \to (0, i);$$
$$(i, 1) \to (1, i)\}$$
$$\mathcal{S}_T = \{S1(i) \to (0, i);$$
$$S2(i) \to (1, i)\}$$

```
// Original loop
for (i=1; i<n; i+=1)
  S1(i);
  S2(i);

// Separated loops
for (i=1; i<n; i+=1)
  S1(i);

for (i=1; i<n; i+=1)
  S2(i);
```

3. Reversal

Reversal

$$\mathcal{I} = \{S1(i) : 1 \leqslant i < n\}$$

$$\mathcal{S} = \{S1(i) \to (i)\}$$

$$\mathcal{T} = \{(i) \to (-i)\}$$

$$\mathcal{S}_T = \{S1(i) \to (-i)\}$$

```
// Original loop
for (i=1; i<n; i+=1)
  S1(i);

// Reversed loop
for (i=1-n; i<0; i+=1)
  S1(-i);
```

4. Interchange

Interchange

$$\mathcal{I} = \{S1(i,j) : 1 \leqslant i < n; \\ \wedge 1 \leqslant j < m\}$$

$$\mathcal{S} = \{S1(i,j) \to (i,j)\}$$

$$\mathcal{T} = \{(i,j) \to (j,i)\}$$

$$\mathcal{S}_T = \{S1(i,j) \to (i,j)\}$$

```
// Original loops
for (i=1; i<n; i+=1)
  for (j=1; j<m; j+=1)
    S1(i,j);

// Interchanged loops
for (i=1; i<m; i+=1)
  for (j=1; j<n; j+=1)
    S1(j,i);
```

5. Strip-mining → it looks similar to loop tiling but on 1-D.

Strip-Mining

$\mathcal{I} = \{S1(i) : 1 \leqslant i < n\}$

$\mathcal{S} = \{S1(i) \to (i)\}$

$\mathcal{T} = \{(i) \to (4 * \lfloor i/4 \rfloor, i)\}$

$\mathcal{S}_T = \{S1(i) \to (4 * \lfloor i/4 \rfloor, i)\}$

```
// Original loop
for (i=1; i<n; i+=1)
  S1(i);

// Strip mined loop
for (ti=0; ti<n; ti+=4)
  for (i = max(1,ti);      ← boundary condition
       i <= min(n-1,ti+3);
       i += 1)
    S1(i);
```

Loop transformations might have different complexity:

1. specific boundary condition handling
2. schedule-only transformation ← this is preferred to simplify both reasoning about as well as their implementations.
3. transformations that require modifications of the iteration space.
4. different schedules are assigned to different instances of the same statement. → piecewise schedule
   ↳ example: index set splitting

In previous works index set splitting was often modeled by introducing (virtual) statement copies. We advocate for the usage of partial schedules as it eliminates the need for statement copies as another concept.

There is also a set of classical loop transformations that change the loop structure, but not the order in which statement instances are executed. Such transformations are loop unrolling, loop peeling or loop unswitching. All of these do not require any schedule transformations, but are different ways of specializing code to reduce control overhead. In Chapter 10 we discuss our work on translating a sched-

6. Skewing

Skewing

$$\mathcal{I} = \{S1(i,j) : 1 \leqslant i < n \wedge 1 \leqslant i < m\}$$

$$\mathcal{S} = \{S1(i,j) \to (i,j)\}$$

$$\mathcal{T} = \{(i,j) \to (i,i+j)\}$$

$$\mathcal{S}_T = \{S1(i) \to (i,i+j)\}$$

```
// Original loops
for (i=1; i<n; i+=1)
  for (j=1; j<n; j+=1)
    S1(i,j);

// Skewed loops
for (i=1; i<n; i+=1)
  for (j=i+1; j<m+i; j+=1)
    S1(i, j-i);
```

2. Tiling

Tiling

$$\mathcal{I} = \{S1(i,j) : 0 \leqslant i < 1024 \wedge 0 \leqslant i < 1024\}$$

$$\mathcal{S} = \{S1(i,j) \to (i,j)\}$$

$$\mathcal{T} = \{(i,j) \to (4 * \lfloor i/4 \rfloor, 4 * \lfloor j/4 \rfloor, i, j)\}$$

$$\mathcal{S}_T = \{S1(i) \to (4 * \lfloor i/4 \rfloor, 4 * \lfloor j/4 \rfloor, i, j)\}$$

```
// Original loops
for (i=0; i<1024; i+=1)
  for (j=0; j<1024; j+=1)
    S1(i,j);

// Tiled loops
for (ti=0; ti<1024; ti+=4)
  for (tj=0; tj<1024; tj+=4)
    for (i=tj; i<ti+4; i+=1)
      for (j=tj; j<tj+4; j+=1)
    S1(i, j-i);
```

8 unroll and Jam

Unroll and Jam

$$\mathcal{I} = \{S1(i,j) : 1 \leqslant i < 1024 \wedge 1 \leqslant i < 1024\}$$

$$\mathcal{S} = \{S1(i,j) \rightarrow (i,j)\}$$

$$\mathcal{T} = \{(i,j) \rightarrow (4 * \lfloor i/4 \rfloor, j, i)\}$$

$$\mathcal{S}_T = \{S1(i) \rightarrow (4 * \lfloor i/4 \rfloor, j, i)\}$$

```
// Original loops
for (i=0; i<1024; i+=1)
  for (j=0; j<1024; j+=1)
    S1(i,j);

// Unroll and jammed loops
for (ti=0; ti<1024; ti+=4)
  for (j=0; j<1024; j+=1)
    for (i=tj; i<ti+4; i+=1)
      S1(i, j-i);
```