High-Level Loop Transformations and Polyhedral Compilation

Sven Verdoolaege

Polly Labs and KU Leuven (affiliated researcher)

May 30, 2017

Outline

1 Loop Transformations
   • Loop Distribution
   • Loop Fusion
   • Loop Tiling

2 Polyhedral Compilation
   • Introduction
   • Polyhedral Model
   • Schedules
   • Operations
   • Software

3 PPCG
   • Overview
   • Model Extraction
   • Dependence Analysis
   • Scheduling
   • Device Mapping

Loop Transformations

Loop Distribution

L: for (int i = 1; i < 100; ++i) {
   A[i] = f(i);
}

Can this loop be parallelized?

Requirement:
writes of iteration do not conflict with reads/writes of other iteration
L1:  \( W(A[1]) R(A[1]) R(A[0]) W(B[1]) \)

Loop distribution changes meaning!

L1: for (int i = 1; i < 100; ++i) {
   A[i] = f(i);
}

No conflicts between iterations of L1 ⇒ can be run in parallel
No conflicts between iterations of L2 ⇒ can be run in parallel

before distribution, L1 reads A[2] value written before code fragment
Loop Fusion

L1: for (int i = 0; i < 100; ++i)
    A[i] = f(i);
L2: for (int i = 0; i < 100; ++i)
    B[i] = g(A[i]);

Assume A does not fit in the cache
⇒ elements get evicted and reloaded for use in L2
Loop fusion (changes execution order ⇒ may not preserve meaning)
for (int i = 0; i < 100; ++i) {
    A[i] = f(i);
    B[i] = g(A[i]);
}
⇒ elements of A get reused immediately
⇒ better locality

If A not needed outside code fragment
⇒ array can be replaced by a scalar
⇒ memory compaction

Loop Tiling

L1: for (int i = 0; i < 8; ++i)
L2: for (int j = 0; j < 8; ++j)
    C[i][j] = A[i] * B[j];

Assume B does not fit in the cache
⇒ elements get (re)loaded and evicted in every iteration of L1

⇒ compute C in tiles, e.g., $4 \times 4$
Loop Tiling

L1: for (int i = 0; i < 8; ++i)
L2: for (int j = 0; j < 8; ++j)
\[ C[i][j] = A[i] \times B[j]; \]

Assume B does not fit in the cache
⇒ elements get (re)loaded and evicted in every iteration of L1
Loop tiling (changes execution order ⇒ may not preserve meaning)
for (int ti = 0; ti < 8; ti += 4)
for (int tj = 0; tj < 8; tj += 4)
for (int i = ti; i < ti + 4; ++i)
for (int j = tj; j < tj + 4; ++j)
\[ C[i][j] = A[i] \times B[j]; \]

Motivation

- Computer architectures are becoming more difficult to program efficiently
  - multiple levels of parallelism
  - non-uniform memory architectures

⇒ Advanced compiler optimizations are required
  - hierarchical partitioning and reordering of operations (e.g., parallelization, loop fusion, ...)
  - mapping to different processing units
  - memory transfers between processing units

⇒ Global view of individual operations is required
⇒ Polyhedral Model

Polyhedral Compilation

Polyhedral Model

Key features
- instance based
  ⇒ statement instances
  ⇒ array elements
- compact representation based on polyhedra or similar objects
  ⇒ Presburger sets and relations
  ⇒ ... 

Main constituents of program representation
- Instance Set
  ⇒ the set of all statement instances
- Access Relations
  ⇒ the array elements accessed by a statement instance
- Dependencies
  ⇒ the statement instances that depend on a statement instance
- Schedule
  ⇒ the relative execution order of statement instances
- Context
  ⇒ constraints on parameters

Polyhedral Compilation — Example

for (t = 0; t < T; t++)
for (i = 1; i < N - 1; i++)
\[ A[(t+1) \mod 2][i] = A[t \mod 2][i-1] + A[t \mod 2][i+1]; \]
Polyhedral Model — Example

for (i = 0; i < 3; ++i)
S: B[i] = f(A[i]);
for (i = 0; i < 3; ++i) >
T: C[i] = g(B[2 - i]);
input code

new code
for (c = 0; c < 3; ++c) {
B[c] = f(A[c]);
C[2 - c] = g(B[c]);
}

Polyhedral Model — Example

for (i = 0; i < 3; ++i)
S: B[i] = f(A[i]);
for (i = 0; i < 3; ++i) >
T: C[i] = g(B[2 - i]);
input code

new code
for (c = 0; c < 3; ++c) {
B[c] = f(A[c]);
C[2 - c] = g(B[c]);
}

Parametric Example: Matrix Multiplication

for (int i = 0; i < M; i++)
    for (int j = 0; j < N; j++) {
S1: C[i][j] = 0;
    for (int k = 0; k < K; k++)
S2: C[i][j] = C[i][j] + A[i][k] * B[k][j];
}

• Instance Set (set of statement instances)

    \{S1[i,j] : 0 \leq i < M \land 0 \leq j < N;\}
    \{S2[i,j,k] : 0 \leq i < M \land 0 \leq j < N \land 0 \leq k < K \}

• Access Relations (accessed array elements; W: write, R: read)

    W = \{S1[i,j] \rightarrow C[i,j]; S2[i,j,k] \rightarrow C[i,j]\}
    R = \{S2[i,j,k] \rightarrow C[i,j]; S2[i,j,k] \rightarrow A[i,k]; S2[i,j,k] \rightarrow B[k,j]\}
Schedule Representation

Schedule $S$ keeps track of relative execution order of statement instances

$\Rightarrow$ for each pair of statement instances $i$ and $j$, schedule determines

- $i$ executed before $j$ ($i <_S j$),
- $i$ executed after $j$ ($j <_S i$), or
- $i$ and $j$ may be executed simultaneously

Schedule trees form a combined hierarchical schedule representation

- Main constructs:
  - affine schedule: instances are executed according to affine function
  - sequence: partitions instances through child filters executed in order

- Order of instances determined by outermost node that separates them

- Deriving schedule tree from AST
  - for loop $\Rightarrow$ affine schedule corresponding to loop iterator
  - compound statement $\Rightarrow$ sequence

Named Presburger Relation Schedules

Schedule tree with single (band) node

Flattening a schedule tree

- two nested band nodes
  $\Rightarrow$ replace by single band node with concatenated partial schedule
- sequence with as children either leaves or trees consisting of a single band node
  $\Rightarrow$ treat leaves as zero-dimensional band nodes
  $\Rightarrow$ pad lower-dimensional bands (e.g., with zero)
  $\Rightarrow$ construct one-dimensional band assigning increasing values to children
  $\Rightarrow$ combine one-dimensional band with children

Parametric Example: Matrix Multiplication

for (int $i = 0; i < M; i++$)
  for (int $j = 0; j < N; j++$) {
    $S1$: $C[i][j] = 0;$
    for (int $k = 0; k < K; k++$)
      $S2$: $C[i][j] = C[i][j] + A[i][k] \times B[k][j];$
  }

$S1[i,j] \rightarrow [i]; S2[i,j,k] \rightarrow [i]$
$S1[i,j] \rightarrow [j]; S2[i,j,k] \rightarrow [j]$
$S2[i,j,k] \rightarrow [k]$
Parametric Example: Matrix Multiplication

```java
for (int i = 0; i < M; i++)
    for (int j = 0; j < N; j++) {
        S1: C[i][j] = 0;
        for (int k = 0; k < K; k++)
            S2: C[i][j] = C[i][j] + A[i][k] * B[k][j];
    }
```

$S1[i,j] \rightarrow [i,j,0,0]; S2[i,j,k] \rightarrow [i,j,1,k]$

---

Loop Transformations and the Polyhedral Model

Loop transformations result in different execution order of statement instances

$\Rightarrow$ different schedule

Polyhedral model can be used to

- **evaluate** a schedule and/or
- **construct** a schedule

Polyhedral schedules can represent (combinations of)

- loop distribution
- loop fusion
- loop tiling
- ...

---

Schedule Properties

- **Validity**
  
  New schedule should preserve meaning

- **Parallelism**

  Can the iterations of a given loop be executed in parallel?

- **Locality**

  Statement instances scheduled closely to each other

- **Tilability**

  Can a given schedule band be tiled?

---

Schedule Validity

New schedule should preserve meaning

- **Internal restrictions**
  
  - No read of a value may be scheduled before the write of the value
  - No other write to same memory location may be scheduled in between

- **External restrictions (on non-temporaries)**
  
  - No write may be scheduled before initial read from a memory location
  - No write may be scheduled after last write to a memory location

- **Sufficient conditions:**

  - Every read of a memory location is scheduled after every preceding write to the same memory location
  - Every write to a memory location is scheduled after every preceding read or write to the same memory location
Dependences

Sufficient conditions for validity of schedule $S$:
- Every read of a memory location is scheduled after every preceding write to the same memory location
- Every write to a memory location is scheduled after every preceding read or write to the same memory location

Dependence relation $D$: pairs of statement instances
- accessing the same memory location
- of which at least one is a write
- with the first executed before the second in original code

Sufficient condition:
\[ \forall i \to j \in D : i \prec_S j \]

Dependence Analysis

Recall: sufficient conditions for validity of schedule $S$:
\[ \forall i \to j \in D : i \prec_S j \]

Dependence relation $D$: pairs of statement instances
- accessing the same memory location
- of which at least one is a write
- with the first executed before the second in original code

Computation:
\[ D = \left( (W^{-1} \circ R) \cup (W^{-1} \circ W) \cup (R^{-1} \circ W) \right) \cap (\prec_{S_0}) \]

$W$: write access relation
$R$: read access relation
$S_0$: original schedule

Local Validity

Schedule validity:
\[ \forall i \to j \in D : i \prec_S j \]

Consider subset of local dependences $L$

At outermost node: $L = D$

Current node
- band node with partial schedule $f$
  \[ \forall i \to j \in L : f(i) \leq_{\text{lex}} f(j) \]
  Carried dependences: $i \to j \in L : f(i) \not= f(j)$
  \[ \Rightarrow \text{no longer need to be considered in nested nodes} \]
  Remaining dependences: $L' = \{i \to j \in L : f(i) = f(j)\}$
- sequence node with child position $p$ and filters $F_k$
  \[ \forall i \to j \in L : p(i) \leq_{\text{lex}} p(j) \]
  Carried dependences: $i \to j \in L : p(i) \not= p(j)$
  Remaining dependences in child $c$: $L' = \{i \to j \in L : i, j \in F_c\}$
- leaf node: $L = \emptyset$

Loop Distribution Validity

for (int $i = 1; i < 100; ++i$) {
    $S[i] \rightarrow [i]; T[i] \rightarrow [i]$}  
$S$: $A[i] = f(i);$  
}  
\{ $S[i] \}, \{ T[i] \}$

Dependences:
\{ $S[i] \rightarrow T[i] : 1 \leq i < 100; S[i] \rightarrow T[i + 1] : 1 \leq i, i + 1 < 100 \}
\{ $S[i] \rightarrow [i]; T[i] \rightarrow [i]$\}  
satisfied: $\{ S[i] \rightarrow T[i] : 1 \leq i < 100; S[i] \rightarrow T[i + 1] : 1 \leq i, i + 1 < 100 \}$
carried: $\{ S[i] \rightarrow T[i + 1] : 1 \leq i, i + 1 < 100 \}$
\{ $S[i] \}, \{ T[i] \}$
satisfied: $\{ S[i] \rightarrow T[i] : 1 \leq i < 100 \}$
carried: $\{ S[i] \rightarrow T[i] : 1 \leq i < 100 \}$
Polyhedral Compilation Schedules

for (int i = 1; i < 100; ++i) {
    {S[i] → [i], T[i] → [i]}
}

Schedules

Schedules

Schedules

Schedules

for (int i = 1; i < 100; ++i) {
    {S[i] → [i], T[i] → [i]}
}

Parallel Loops and Parallel Band Members

Recall:

Iterations of a given loop can be executed in parallel if
writes of iteration do not conflict with reads/writes of other iteration
(if there is no dependence between distinct iterations
(for any given iteration of the outer loops)

A band member with affine function $f$ is parallel if
\[ \forall i \rightarrow j \in L : f(i) = f(j) \]

with $L$ the local dependences

Loop Distribution and Parallelism

for (int i = 1; i < 100; ++i) {
    {S[i] → [i], T[i] → [i]}
}

for (int i = 1; i < 100; ++i) {
    {S[i] → [i], T[i] → [i]}
}

for (int i = 1; i < 100; ++i) {
    {S[i] → [i], T[i] → [i]}
}

for (int i = 1; i < 100; ++i) {
    {S[i] → [i], T[i] → [i]}
}

for (int i = 1; i < 100; ++i) {
    {S[i] → [i], T[i] → [i]}
}

}
Loop Distribution and Parallelism

for (int i = 1; i < 100; ++i) {
  \{ S[i] \rightarrow [i], T[i] \rightarrow [i] \}
}

S:  \quad A[i] = f(i);
T:  \quad B[i] = A[i] + A[i - 1];

\{ S[i] \}, \{ T[i] \}

Dependences:
\{ S[i] \rightarrow T[i] : 1 \leq i < 100; S[i] \rightarrow T[i + 1] : 1 \leq i, i + 1 < 100 \}

Loop distribution

\begin{align*}
\text{for (int i = 1; i < 100; ++i) } & \quad \{ S[i] \}, \{ T[i] \} \\
\text{A[i] } & = f(i) \\
\text{for (int i = 1; i < 100; ++i) } & \quad \{ S[i] \rightarrow [i], T[i] \rightarrow [i] \}
\end{align*}


\{ S[i] \rightarrow [i] \} \quad \{ T[i] \rightarrow [i] \}

local: \emptyset \quad \text{local: } \emptyset

\Rightarrow \text{parallel} \quad \Rightarrow \text{parallel}

Parallelism Example

for (int i = 1; i < 6; ++i)
for (int j = 0; j < 6; ++j)
S:  \quad A[i][j] = f(A[i - 1][j + 1]);
Dependences:
\{ S[i,j] \rightarrow S[i + 1,j - 1] : 1 \leq i, i + 1 < 6 \land 0 \leq j, j - 1 < 6 \}

\begin{align*}
\text{original schedule: } & \quad S[i,j] \rightarrow [i,j] \\
\text{new schedule: } & \quad S[i,j] \rightarrow [i + j,i] \\
(i + j)-\text{direction is outer parallel}
\end{align*}

Decomposition: loop skewing + loop interchange

\[ [i,j] \rightarrow [i,i+j] \rightarrow [i+j,i] \]

Array Dataflow Analysis

Given a read from an array element, what was the last write to the same array element before the read?

\begin{align*}
\text{for (i = 0; i < N; ++i) } & \quad F: \quad a[i+j] = f(a[i+j]) \\
\text{for (j = 0; j < N - i; ++j) } & \quad G: \quad g(a[i])
\end{align*}

Access relations:

\begin{align*}
A_1 & = \{ F[i,j] \rightarrow a[i+j] : 0 \leq i < N \land 0 \leq j < N - i \} \\
A_2 & = \{ G[i] \rightarrow a[i] : 0 \leq i < N \}
\end{align*}

Map to all writes: \( R'' = A_1^{-1} \circ A_2 \) = \{ G[i] \rightarrow F[i',i - i'] : 0 \leq i' \leq i < N \}

Map to all preceding writes:

\begin{align*}
R' & = R'' \cap (<s)^{-1} = \{ G[i] \rightarrow F[i',i - i'] : 0 \leq i' \leq i < N \} \\
\text{Last preceding write: } & \quad R = \max_{<s} R' = \{ G[i] \rightarrow F[i,0] : 0 \leq i < N \}
\end{align*}

Locality

Statement instances i and j that reuse memory
\Rightarrow \text{scheduled closely to each other: } f(j) - f(i) \text{ small}

Types of locality:

- temporal locality
  \Rightarrow \text{instances that access the same memory element}

- spatial locality
  \Rightarrow \text{instances that access adjacent memory elements}

Sometimes further distinction made:

- self locality
  \Rightarrow \text{pair of instances from same statement}

- group locality
  \Rightarrow \text{any pair of statement instances}

Temporal locality often restricted to
pairs of writes and reads that refer to the same value
\Rightarrow \text{dataflow}
Tiling a Band

Input:
- band of affine schedule functions
  \( f_1, f_2, \ldots, f_n \)
- tile sizes
  \( T_1, T_2, \ldots, T_n \)

Steps (conceptually)
1. divide each direction into chunks of size \( T_i \) (strip-mining)
   
   \[ [f_1/T_1, f_2/T_2, \ldots, f_n/T_n], f_n \]
   
   does not change execution order \( \Rightarrow \) always valid
2. combine the chunking (interchange)
   
   \[ [f_1/T_1, f_2/T_2, \ldots, f_n/T_n], f_1, f_2, \ldots, f_n \]

sufficient condition for interchange:
- all members are valid for local dependences at (top of) band
- \( \Rightarrow \) permutable band

Loop Tiling Example

for (int \( i = 0; \ i < 8; \ ++i \))
  for (int \( j = 0; \ j < 8; \ ++j \))
  \[
  S: \quad C[i][j] = A[i] * B[j];
  \]

\( \blacklozenge \) strip-mine

\[
S[i,j] \rightarrow 4[i/4]
\]

\[
S[i,j] \rightarrow i
\]

for (int \( ti = 0; \ ti < 8; \ ti += 4 \))
  for (int \( i = ti; \ i < ti + 4; \ ++i \))
  for (int \( tj = 0; \ tj < 8; \ tj += 4 \))
  \[
  S[i,j] \rightarrow 4[j/4]
  \]
  \[
  S[i,j] \rightarrow j
  \]

for (int \( j = tj; \ j < tj + 4; \ ++j \))
  \[
  C[i][j] = A[i] * B[j];
  \]

Operations on Polyhedral Model

- **Model Extraction**
  - Input: AST
  - Output: instance set, access relations, original schedule

- **Dependence analysis**
  - Input: instance set, access relations, original schedule
  - Output: dependence relations

- **Scheduling**
  - Input: instance set, dependence relations
  - Output: schedule

- **AST generation (polyhedral scanning, code generation)**
  - Input: instance set, schedule
  - Output: AST

- **Data layout transformations**
  - Input: access relations, dependence relations
  - Output: transformed access relations
Polyhedral Model Requirements

Requirements for basic polyhedral model: “regular” code

- Static control
  - control does not depend on input data
- Affine
  - all relevant expressions are (quasi-)affine
- No Aliasing
  - essentially no pointer manipulations

Note:
- polyhedral model may be \textit{approximation} of input that does not strictly satisfy all requirements
- many extensions are available

Aliasing

Some possible ways of handling aliasing:

- use an input language that does not permit aliasing
- pretend the problem does not exist
- require user to ensure absence of aliasing
  - e.g., use restrict keyword
- handle as may-write
  - may lead to too many dependences
- check aliasing at run-time
  - use original code in case of aliasing

Polyhedral Scheduling

Polyhedral model can be used to

- \textit{evaluate} a schedule and/or
- \textit{construct} a schedule

Some popular polyhedral schedulers:

- Feautrier
  - maximal inner parallelism
    - carry as many dependences as possible at outer bands
- Pluto
  - tilable bands
  - locality: $f(j) - f(i)$ small
    - parallelism as extreme case: $f(j) - f(i) = 0$

Many other scheduling algorithms have been proposed

Data layout transformations

- Memory compaction
  - Reuse memory locations to store different data
    - apply non-injective mapping to array elements
    - reduce memory requirements
    - extreme case: replace array by scalar

  \begin{verbatim}
  for (int i = 0; i < 100; ++i) {
    A[i] = f(i);
    B[i] = g(A[i]);
  }
  \end{verbatim}

- Expansion
  - Use different memory locations to store different data
    - map different accesses to memory element to distinct locations
    - increase scheduling freedom (e.g., more parallelism)
False Dependences

```
for (int i = 0; i < n; ++i) {
    S: t = f1(A[i]);
    T: B[i] = f2(t);
}
```

Dependences

- read-after-write ("true"): \{ S[i] → T[i'] : i' ≥ i \}
- write-after-read ("anti"): \{ T[i] → S[i'] : i' > i \}
- write-after-write ("output"): \{ S[i] → S[i'] : i' > i \}

False dependences not from dataflow, but from reuse of memory location t

Possible solution: expansion/privatization

```
for (int i = 0; i < n; ++i) {
    S: t[i] = f1(A[i]);
    T: B[i] = f2(t[i]);
}
```

```
dataflow (subset of "true" dependences): \{ S[i] → T[i] \}
```

⇒ only remaining dependences are dataflow induced

Expansion

Assume:

- instance sets and access relations are static and exact
  ⇒ each read has exactly one corresponding write
- single read and write per statement
  ⇒ expanded array indexed by statement instance of write

```
for (int i = 0; i < n; ++i) {
    S: t = f1(A[i]);
    T: B[i] = f2(t);
}
```

```
dataflow: \{ S[i] → T[i] \}
```

Maximal Static Expansion

```
for (int i = 0; i < n; ++i) {
    S1: t = f1(i);
    S2: A[i] = t;
    S3: t = f2(i);
    S4: if (f3(i))
        t = f4(i);
    S5: B[i] = t;
    S6: B[i] = t;
}
```

```
dataflow cannot be determined independently of run-time information
⇒ approximate dataflow
\{ S1[i] → S2[i]; S3[i] → S6[i]; S5[i] → S6[i] \}
⇒ a read may be associated to more than one write
⇒ corresponding equivalence classes should not be expanded apart
```

Approximate Dataflow Analysis

How to compute dataflow in presence of data dependent control?

Two approaches

- **Direct computation**
  - distinguish between may- and must-writes
- **Derived from exact run-time dependent dataflow**
  - compute exact dataflow in terms of run-time information
  - exploit properties of run-time information
  - project out run-time information
May Writes

Keep track of whether write is possible or definite

- **Must-writes**
  - Array elements are *definitely* written by statement instance

- **May-writes**
  - Array elements are *possibly* written by statement instance
    - statement instance not necessarily executed
      - for (i = 0; i < n; ++i)
        - if (A[i] > 0)
        - S: B[i] = A[i];
      - May-write: \{S[i] → B[i]\}
    - array element not necessarily accessed
      - int A[N];
      - /* ... */
      - T: A[B[0]] = 5;
      - May-write: \{T[0] → A[a] : 0 ≤ a < N\}

Must-write access relation is subset of may-write access relation

---

Approximate Dataflow — Direct Computation

- **Read-after-write dependences**
  - write and read access same memory location
  - write executed before the read

  ⇒ Approximate dataflow analysis with no must-writes

- **Dataflow dependences**
  - write and read access same memory location
  - write executed before the read
  - no intermediate write to same memory location
    ⇒ intermediate write kills dependence

- **Approximate dataflow dependences**
  - may-write and read access same memory location
  - may-write executed before the read
  - no intermediate must-write to same memory location
    ⇒ intermediate must-write kills dependence

---

Approximate Dataflow Analysis

How to compute dataflow in presence of data dependent control?

Two approaches

- **Direct computation**
  - distinguish between may- and must-writes

- **Derived from exact run-time dependent dataflow**
  - compute exact dataflow in terms of run-time information
  - exploit properties of run-time information
  - project out run-time information

---

Run-time Dependent Dataflow Analysis

Approaches

- “fuzzy array dataflow analysis”
- “on-demand-parametric array dataflow analysis”

```c
for (int i = 0; i < n; ++i) {
    t = f1(i);
    S2: A[i] = t;
    S3: t = f2(i);
    S4: if (f3(i))
        S5: t = f4(i);
    S6: B[i] = t;
}
```

- **Run-time dependent dataflow**

\[
\{ S1[i] → S2[i]; S3[i] → S6[i] : \beta^S_{S6} = 0; S5[i] → S6[i] : \beta^S_{S6} = 1 \}
\]

- **\(\beta^P_C\):** any potential source instance \(P\) is executed for sink \(C\)
- **\(\lambda^P_C\):** last potential source instance \(P\) executed for sink \(C\)

- **Approximate dataflow (project out \(\beta\) and \(\lambda\))**

\[
\{ S1[i] → S2[i]; S3[i] → S6[i]; S5[i] → S6[i] \}
\]
Representing Dynamic Conditions

N1: \( n = f(); \)
\[
\text{for (int } k = 0; k < 100; ++k) { \\
\text{\hspace{1em} m = g();} \\
\text{\hspace{1em} for (int } i = 0; i < m; ++i) \\
\text{\hspace{2em} for (int } j = 0; j < n; ++j) \\
\text{\hspace{3em} a[j][i] = g();} \\
\text{\hspace{1em}} } \\
\}
\]

What is instance set (restricted to \( \text{A statement} \))?
\( \{ A[k,i,j] : 0 \leq k < 100 \land 0 \leq i < m \land 0 \leq j < n \} \)?
\( \Rightarrow \text{no, } m \text{ and } n \text{ cannot be treated as symbolic constants} \)
\( \text{they are modified inside k-loop} \)
\( \{ A[k,i,j] : 0 \leq k < 100 \land 0 \leq i < \text{valueOf}_\text{m}(k) \land 0 \leq j < \text{valueOf}_\text{n}(k) \} \)
\( \Rightarrow \text{requires uninterpreted functions (of arity 0)} \)
Alternative: use overapproximation of instance set and keep track of which elements are executed

Representing Dynamic Conditions

N1: \( n = f(); \)
\[
\text{for (int } k = 0; k < 100; ++k) { \\
\text{\hspace{1em} m = g();} \\
\text{\hspace{1em} for (int } i = 0; i < m; ++i) \\
\text{\hspace{2em} for (int } j = 0; j < n; ++j) \\
\text{\hspace{3em} a[j][i] = g();} \\
\text{\hspace{1em}} } \\
\}
\]

Polyhedral Process Networks

- Main purpose: extract task level parallelism from dataflow graph
  
  \text{statement} \rightarrow \text{process}
  
  \text{flow dependence} \rightarrow \text{communication channel}

\( \Rightarrow \text{requires dataflow analysis} \)

Processes are mapped to parallel hardware (e.g., FPGA)

Example:

for (int \( i = 0; i < n; ++i \)) {
    write(fifo, f1(A[i]));
}
for (int \( i = 0; i < n; ++i \)) {
    \text{B[i] = f2(read(fifo));}
}
Process Networks with Dynamic Control

```c
for (int i = 0; i < n; ++i) {
    S1: t = f1(i);
    S2: A[i] = t;
    S3: t = f2(i);
    S4: if (f3(i))
        S5: t = f4(i);
    S6: B[i] = t;
}
```

Run-time dependent dataflow:

- `S1[i] → S2[i], S3[i] → S6[i] : β^SS_{56} = 0`
- `S5[i] → S6[i] : β^SS_{56} = 1; S4[i] → S5[i]`

---

CARP Project (2011–2015)

Design tools and techniques to aid Correct and Efficient Accelerator Programming

**Domain Specific Languages**

- pencil
- Platform-Neutral
- Compute Intermediate Language
- Direct pencil programming (hand written pencil code)
- Optimizing, auto-parallelizing pencil → OpenCL compiler

**Direct OpenCL programming**

- GPUs
- CPUs
- FPGAs
- Other accelerators

---

PPCG Overview

PPCG:

-_detect/expose parallelism
- map parts of the code to an accelerator
- copy data to/from device
- introduce local copies of data

pencil:

- C99 with restrictions and some extra builtins and pragmas

---

Polyhedral Software

http://polyhedral.info/software.html

- Core set manipulation libraries
  - integer sets: isl, omega(+), ...
  - rational sets: PolyLib, PPL, ...
- Model extraction
  - clan, pet, ...
- Dependence analysis
  - petit, candl, isl, FADA, ...
- Scheduler libraries
  - LetSee, isl, ...
- AST generation
  - omega(+), CLooG, isl, ...
- Source-to-source polyhedral compilers
  - Pluto, PoCC, PPCG, ...
- Compilers using polyhedral compilation
  - gcc/graphite, LLVM/Polly, ...

---

Polyhedral Compilation Operations May 30, 2017 55 / 82
Process Networks with Dynamic Control

for (int i = 0; i < n; ++i) {
    S1: t = f1(i);
    S2: A[i] = t;
    S3: t = f2(i);
    S4: if (f3(i))
        S5: t = f4(i);
    S6: B[i] = t;
}

---

Polyhedral Software

[4, 7, 8, 9, 10, 11, 16, 18, 19, 20, 21, 22, 23, 29, 31, 34]
PPCG Internal Structure

Instance Set
Region that needs to be extracted may be

- marked by
  
  ```c
  #pragma scop
  #pragma endscop
  ```
- autodetected (--pet-autodetect)

Internal structured dynamic control is encapsulated

```c
for (int x = 0; x < n; ++x) {
  A: s = f();
  B: while (P(x, s))
      s = g(s);
  C: h(s);
}
```

Instance set: \{ A[x] : 0 \leq x < n; B[x] : 0 \leq x < n; C[x] : 0 \leq x < n \}

Note: currently, internal order of accesses is lost

⇒ possible loss of accuracy in dependence analysis

Connection with other Libraries and Tools

- **LLVM**
- **imath**
- **GMP**
- **clang**
- **isl**
- **NTL**
- **PolyLib**
- **Polly**
- **pet**
- **barvinok**
- **isa**
- **iscc**

**Licenses:**
- BSD/MIT
- LGPL
- GPL

Isl: manipulates parametric affine sets and relations
Pet: extracts polyhedral model from clang AST
PCCG: Polyhedral Parallel Code Generator
Pencilcc: pencil compiler

Inlining
Enabled through C99 `inline` keyword on function definition

```c
inline void set_diagonal(int n, float A[const restrict static n][n], float v)
{
  for (int i = 0; i < n; ++i)
    U: A[i][i] = v;
}
```

```c
void f(int n, float A[const restrict static n][n])
{
  #pragma scop
  S: set_diagonal(n, A, 0.f);
  for (int i = 0; i < n; ++i)
    T: for (int j = i + 1; j < n; ++j)
  #pragma endscop
}
```

Instance set: \{ U[i] : 0 \leq i < n; T[i,j] : 0 \leq i < j < n \}
Access Relations and Function Calls

```c
void set_diagonal ( int n,
float A[ const restrict static n ][ n], float v)
{
    for ( int i = 0; i < n; ++i)
        A[i][i] = v;
}

void f( int n, float A[ const restrict static n ][ n])
{
    #pragma scop
    S: set_diagonal (n, A, 0.f);
    for ( int i = 0; i < n; ++i)
        for ( int j = i + 1; j < n; ++j)
    #pragma endscop
}
```

May-write: \{ S[] \rightarrow A[i,i]: 0 \leq i < n; T[i,j] \rightarrow A[i,j]: 0 \leq i < j < n \}
Must-write: \{ S[] \rightarrow A[i,i]: 0 \leq i < n; T[i,j] \rightarrow A[i,j]: 0 \leq i < j < n \}

Access Relations and Structures

```c
struct s {
    int a;
    int b;
};
```

```c
int f()
{
    struct s a, b[10];
    S: a.b = 57;
    T: a.a = 42;
    for ( int i = 0; i < 10; ++i)
        U: b[i] = a;
}
```

May-write: \{ S[] \rightarrow a.b[a] \rightarrow b[]; T[] \rightarrow a.a[a] \rightarrow a[]; U[i] \rightarrow b_a[b][i] \rightarrow a[]; U[i] \rightarrow b_b[b][i] \rightarrow b[] \}
Must-write: \{ S[] \rightarrow a_b[a] \rightarrow b[]; T[] \rightarrow a_a[a] \rightarrow a[]; U[i] \rightarrow b_a[b][i] \rightarrow a[]; U[i] \rightarrow b_b[b][i] \rightarrow b[] \}

Summary Functions

Analysis of accesses in called function may be inaccurate or even infeasible
- missing body (library function without source)
- unstructured control
- aliasing
- pattern inside dynamic control is ignored
- additional information not explicitly expressed in code

⇒ explicitly specify accesses in summary function

Summary Function Example

```c
int f(int i); int maybe(); struct s { int a; }
void set_odd_summary(int n, struct s A[static n]) {
    for ( int i = 1; i < n; i += 2)
        if ( maybe() )
            A[i].a = 0;
}
```

```c
__attribute__ (( pencil_access ( set_odd_summary )))
void set_odd(int n, struct s A[static n])
{
    for ( int i = 0; i < n; ++i)
        A[2 * f(i) + 1].a = i;
}
```

```c
void foo(int n, struct s B[static 2 * n])
{
    #pragma scop
    S: set_odd(2 * n, B);
    #pragma endscop
}
```

May-write: \{ S[] \rightarrow B_a[B][i] \rightarrow a[]: 0 \leq i < 2n \land i \mod 2 = 1 \}
Context

The context collects constraints on the symbolic constants derived by pet
- exclude values that result in undefined behavior
  * negative array sizes
  * out-of-bounds accesses
  * signed integer overflow
- __builtin_assume or __pencil_assume
  ⇒ any constraint can be specified
  ⇒ only quasi-affine constraints on symbolic constants are exploited
- specified on PPCG command line
  - --ctx
  - --assume-non-negative-parameters

Main purpose: simplify generated AST

Dependence analysis in isl

isl contains generic dependence analysis engine
⇒ determines dependence relations between “sources” and “sinks”

Input:
- Sink $K : I \rightarrow D$
- May-source $Y : I \rightarrow D$
- Kill $L : I \rightarrow D$
- Schedule $S$ on $I$ ⇒ defines “before” and “intermediate”

Output:
- May-dependence relation: triples $(i, k, a)$
  - $i$ has a may-source to $a$
  - $k$ has a sink to $a$
  - $i$ is scheduled before $k$
  - there is no intermediate kill to $a$
- May-no-source: sinks $k \rightarrow a$ with no kill to $a$ before $k$

Dependence analysis in PPCG

isl:
- May-dependence relation: triples $(i, k, a)$
  - $i$ has a may-source to $a$
  - $k$ has a sink to $a$
  - $i$ is scheduled before $k$
  - there is no intermediate kill to $a$
- May-no-source: sinks $k \rightarrow a$ with no kill to $a$ before $k$

PPCG (without live-range reordering):
- flow dependences (without $a$) and live-in (may-no-source)
  - sink: may-read
  - may-source: may-write
  - kill: must-write
- false dependences (without $a$)
  - sink: may-write
  - may-source: may-read or may-write
  - kill: must-write
- killed writes (without $k$) (⇒ removed from may-write to get live-out)
  - sink: must-write
  - may-source: may-write

Live-Range Reordering

Live-range reordering
- allows such live-ranges to be reordered
- using somewhat different classification of dependences
- computed using different calls to the same dependence analysis engine
Pure Kills

Basic idea:
- Must-writes kill dependences to earlier writes
- Pure kills can also be useful
- Used only as kills during dependence analysis, not as source

Kills can be inserted
- automatically by pet
  - Variable declared within SCoP
    ⇒ kill at declaration
    ⇒ kill at end of enclosing block (if within SCoP)
  - Variable declared in scope that contains SCoP, only used inside
    ⇒ kill at end of SCoP
- manually by the user
  - __pencil_kill

Dependence analysis in PPCG

isl:
- May-dependence relation: triples (i, k, a)
  - i has a may-source to a
  - k has a sink to a
  - i is scheduled before k
  - there is no intermediate kill to a
- May-no-source: sinks k → a with no kill to a before k

PPCG (without live-range reordering):
- flow dependences (without a) and live-in (may-no-source)
  - sink: may-read
  - may-source: may-write
  - kill: must-write or pure kill
- false dependences (without a)
  - sink: may-write
  - may-source: may-read or may-write
  - kill: must-write
- killed writes (without k) (⇒ removed from may-write to get live-out)
  - sink: must-write or pure kill
  - may-source: may-write

Absence of Loop Carried Dependences

void foo(int n, int A[restrict static n][n], int B[restrict static n][n])
{
    for (int i = 0; i < n; ++i)
    #pragma pencil independent
        for (int j = 0; j < n; ++j)
            B[i][A[i][j]] = i + j;
}

Assume each row of A has distinct elements
⇒ no loop-carried dependences, but PPCG cannot tell
⇒ add #pragma pencil independent

Note: not handled very efficiently in current version of PPCG
⇒ only add when needed
Optimization Criteria for PPCG

- Two levels of parallelism
  - blocks and threads (work groups and work items)
  - parallelism
- In PPCG, second level obtained through tiling
  - tilability
- Reduced working set for some arrays
  - mapping to shared memory or registers
  - obtained through tiling
  - tilability
- Reduced data movement
  - locality
- Simple schedules
  - schedule used in several subsequent steps, including AST generation
  - simplicity

Scheduling Constraints

- Validity $a \rightarrow b$
  - statement instance $b$ needs to be executed after $a$
  - $f(b) \geq f(a)$
- Proximity $a \rightarrow b$
  - statement instance $b$ preferably executed close to $a$
  - $f(b) - f(a)$ as small as possible
- Coincidence $a \rightarrow b$
  - statement instance $b$ preferably executed together with $a$
  - $f(b) = f(a)$
  - band member only considered “coincident” if it coschedules all pairs
- Conditional validity (live-range reordering)
  - condition $b \rightarrow c$
  - conditioned validity $a \rightarrow b, c \rightarrow d$

Schedule constraints only relevant if coscheduled by outer nodes
Other schedule constraints are said to be carried by some outer node

Dependences and Schedule Constraints

Traditional dependences
- flow dependences
  - validity constraints
  - proximity constraints
  - coincidence constraints (when parallelism is important)
- false dependences
  - validity constraints
  - coincidence constraints (when parallelism is important)
  - proximity constraints (optional for memory reuse)
- pairs of reads with shared write (“input dependences”)
  - proximity constraints (optional)

Live-range reordering
- somewhat different classification of dependences
- slightly different mapping to schedule constraints

Current PPCG
- adds false dependences to proximity constraints for historical reasons
- does not consider input dependences
- uses live-range reordering by default

Forced Outer Coincidence Scheduler

Recall:
- Feautrier
  - maximal inner parallelism
  - carry as many dependences as possible at outer bands
- Pluto
  - tilable bands
  - locality: $f(j) - f(i)$ small
  - parallelism as extreme case: $f(j) - f(i) = 0$

PPCG uses variant of Pluto-algorithm with Feautrier fallback
- force outer coincidence in each band
- locally fall back to Feautrier if infeasible (single step)

Members in bands constructed by Pluto-algorithm are permutable
- if outer member cannot be coincident, then no member can be
Each step in Feautrier algorithm carries as many dependences as possible
- subsequent application of Pluto more likely to find coincident member
Device Mapping

Input: schedule tree

If schedule tree contains no coincident band member
⇒ generate pure CPU code

Otherwise:

- select subtree for mapping to the device
  selected subtree is entire schedule tree, except
  - coincidence-free children of outer set node
  - coincidence-free initial children of outer sequence node
- within selected subtree, generate kernels for
  - outermost bands with coincident members
  - maximal coincidence-free subtrees
    ⇒ insert zero-dimensional band node
- add data copying to/from device around selected subtree
- add device initialization and clean-up around entire schedule tree

Data Copying to/from Device

Copy-out:
- take may-writes
- remove writes only needed for dataflow inside selected subtree
- approximate to entire array

May-persist:
- elements that may need to be preserved by selected subtree
- consists of
  - elements that may need to be preserved by entire SCOp
    ⇒ elements not definitely written and not definitely killed
  - elements in potential dataflow across selected subtree

May-not-written: \( \text{copy-out} \cap \text{ran may-persist} \) \( \supseteq \) must-write

Copy-in: live-in \( \cup \) may-not-written

Note: if array elements are structures, then entire structures are copied

Data Copying Example

```c
__pencil_kill (A);
for (int i = 0; i < n; i++)
  if (B[i] > 0)
    A[i] = B[i];
```

A may be written
⇒ A in copy-out

A may also not be written (completely), but no data can flow across kill
⇒ parts of A may (be expected to) survive
⇒ A also needs to be in copy-in

References I


References II


References III


References IV


http://icps.u-strasbg.fr/~bastoul/development/candl/.


References V


References VI


References VII


References VIII


References IX


References


