A Polyhedral Compilation Framework for Loops with Dynamic Data-Dependent Bounds

Jie Zhao, Michael Kruse and Albert Cohen

INRIA & École Normale Supérieure
45 rue d'Ulm, 75005 Paris, France

27th International Conference on Compiler Construction (CC 2018)
Vienna, Austria

January 24, 2018

Outline

Dynamic counted loops

What are *dynamic counted loops*?

Dynamic counted loops

What are *dynamic counted loops*?

Definition

Counted loops with *dynamic data-dependent bounds* are counted loops (a.k.a. `do` loops in Fortran) with numerical constant strides, whose lower and/or upper bound may not be an affine function of enclosing loop counters and loop-invariant parameters

Dynamic counted loops

What are *dynamic counted loops*?

Definition

Counted loops with *dynamic data-dependent bounds* are counted loops (a.k.a. `do` loops in Fortran) with numerical constant strides, whose lower and/or upper bound may not be an affine function of enclosing loop counters and loop-invariant parameters

```
   for (i=0; i<N; i++) {
S0:  lb = idx[i];
S1:  ub = idx[i+1];
     for (j=lb; j<ub; j++)  // dynamically computed bounds
S2:    S(i, j);
   }
```

Dynamic counted loops

Why are we interested in the class of loop nest kernels involving `dynamic counted loops`?

Dynamic counted loops

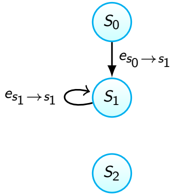
Why are we interested in the class of loop nest kernels involving `dynamic counted loops`?

- `dynamic counted loops` are less expressive than general `while` loops.
- Less expressive/general control flow enables more aggressive optimizations.
- Building on the state of the art polyhedral optimization of `while` loops by Benabderrahmane et al. [BPCB10], but the authors' efficient code generation algorithm is not completely described.
- [BPCB10] is constrained by inductive dependences on exit conditions which limit affine transformations and parallelization.

Comparison with general `while` loops

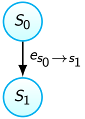
```
   for (i=0; i<N; i++) {
S0:   condition = ...;
      while (condition) {
S1:     condition = ...;
S2:     S;
      }
   }
```

A general while loop




```
   for (i=0; i<N; i++) {
S0:   m = condition;
      for (j=0; j<m; j++)
S1:     S;
   }
```

A dynamic counted loop



Real-life examples of dynamic counted loops

Dynamic counted loops play an important role in numerical solvers, media processing applications, data analytics, etc. They can be found in

- Dynamic programming
- Histogram of oriented gradients
- Finite element method
- Sparse matrix-vector/matrix-matrix multiplications
- ...

The polyhedral model

The polyhedral model represents a program and its semantics using iteration domains, access relations, dependences and schedules.

The polyhedral model

The polyhedral model represents a program and its semantics using iteration domains, access relations, dependences and schedules.

```
   for (i=0; i<N; i++) {
S0:  lb = idx[i];
S1:  ub = idx[i+1];
     for (j=lb; j<ub; j++)
S2:    S(i, j);
   }
```

The polyhedral model

The polyhedral model represents a program and its semantics using iteration domains, access relations, dependences and schedules.

```
    for (i=0; i<N; i++) {
S0:   lb = idx[i];
S1:   ub = idx[i+1];
      for (j=lb; j<ub; j++)
S2:     S(i, j);
    }
```

- Iteration domain:
  $\{S_0(i) : 0 \leq i < N; S_1(i) : 0 \leq i < N; S_2(i) : 0 \leq i < N\}$
- Access relation:
  - Write: $\{S_0(i) \to lb : 0 \leq i < N; S_1(i) \to ub : 0 \leq i < N\}$
  - Read: $\{\}$
- Dependence: $\{\}$[a]
- Schedule: $[S_0(i) \to (i, 0); S_1(i) \to (i, 1); S_2(i) \to (i, 2)]$

[a]Consider only true/flow dependences.

The polyhedral model

The polyhedral model is not able to classify the whole loop nest as a static control part (SCoP).

```
   for (i=0; i<N; i++) {
S0:  lb = idx[i];
S1:  ub = idx[i+1];
     for (j=lb; j<ub; j++)
S2:    S(i, j);
   }
```

- Iteration domain:
  $\{S_0(i) : 0 \leq i < N; S_1(i) : 0 \leq i < N; S_2(i) : 0 \leq i < N\}$
- Access relation:
  - Write: $\{S_0(i) \rightarrow lb : 0 \leq i < N; S_1(i) \rightarrow ub : 0 \leq i < N\}$
  - Read: $\{\}$
- Dependence: $\{\}$[a]
- Schedule: $[S_0(i) \rightarrow (i, 0); S_1(i) \rightarrow (i, 1); S_2(i) \rightarrow (i, 2)]$

[a] Consider only true/flow dependences.

The polyhedral model

The polyhedral model is not able to classify the whole loop nest as a static control part (SCoP).

```
   for (i=0; i<N; i++) {
S0:  lb = idx[i];
S1:  ub = idx[i+1];
     for (j=lb; j<ub; j++)
S2:    S(i, j);
   }
```

- Iteration domain:
  $\{S_0(i) : 0 \leq i < N; S_1(i) : 0 \leq i < N; S_2(i,j) : 0 \leq i < N \wedge lb \leq j < ub\}$
- Access relation:
  - Write: $\{S_0(i) \rightarrow lb : 0 \leq i < N; S_1(i) \rightarrow ub : 0 \leq i < N\}$
  - Read:
    $\{S_2(i,j) \rightarrow lb : 0 \leq i < N \wedge lb \leq j < ub; S_2(i,j) \rightarrow ub : 0 \leq i < N \wedge lb \leq j < ub\}$
- Dependence: $\{S_0[i]- > S_2[i' = i, j] : 0 \leq i < N \wedge lb \leq j < ub; S_1[i]- > S_2[i' = i, j] : 0 \leq i < N \wedge lb \leq j < ub\}$
- Schedule: $[S_0(i) \rightarrow (i, 0); S_1(i) \rightarrow (i, 1); S_2(i,j) \rightarrow (i,j)]$

The polyhedral model

The polyhedral model is not able to classify the whole loop nest as a static control part (SCoP).

Our purpose is to extend the polyhedral model to handle dynamic counted loops and generate code for both general-purpose multicores and heterogeneous accelerators.

Program analysis

Program analysis

- Preprocessing
  - Subtract (dynamic) lower bounds.
  - Synthesize static upper bounds (static analysis or dynamic inspector).

Program analysis

- Preprocessing
  - Subtract (dynamic) lower bounds.
  - Synthesize static upper bounds (static analysis or dynamic inspector).

```
   for (i=0; i<N; i++) {
S0:  lb = idx[i];
S1:  ub = idx[i+1];
     for (j=lb; j<ub; j++)
S2:    S(i, j);
   }
```

Program analysis

- Preprocessing
  - Subtract (dynamic) lower bounds.
  - Synthesize static upper bounds (static analysis or dynamic inspector).

```
   for (i=0; i<N; i++) {
S0:  lb = idx[i];
S1:  ub = idx[i+1];
     for (j=lb; j<ub; j++)
S2:    S(i, j);
   }
```

```
   for (i=0; i<N; i++) {
S0:  m = idx[i+1] - idx[i];
     for (j=0; j<m; j++)
S1:    S(i, j+idx[i]);
   }
```

Program analysis

- Preprocessing
  - Subtract (dynamic) lower bounds.
  - Synthesize static upper bounds (static analysis or dynamic inspector).

```
   for (i=0; i<N; i++) {
S0:  lb = idx[i];
S1:  ub = idx[i+1];
     for (j=lb; j<ub; j++)
S2:    S(i, j);
   }
```

```
   for (i=0; i<N; i++) {
S0:  m = idx[i+1] - idx[i];
     for (j=0; j<m; j++)
S1:    S(i, j+idx[i]);
   }
```

- Modeling control dependences
  - Insert an exit predicate.
  - Delay the introduction of early exit.
  - Sink the dynamic conditions when targeting on GPUs.

Program analysis

- Preprocessing
  - Subtract (dynamic) lower bounds.
  - Synthesize static upper bounds (static analysis or dynamic inspector).

```
   for (i=0; i<N; i++) {
S0:  lb = idx[i];
S1:  ub = idx[i+1];
     for (j=lb; j<ub; j++)
S2:    S(i, j);
   }
```

```
   for (i=0; i<N; i++) {
S0:  m = idx[i+1] - idx[i];
     for (j=0; j<m; j++)
S1:    S(i, j+idx[i]);
   }
```

- Modeling control dependences
  - Insert an exit predicate.
  - Delay the introduction of early exit.
  - Sink the dynamic conditions when targeting on GPUs.

```
   for (i=0; i<N; i++) {
S0:  m = idx[i+1] - idx[i];
     for (j=0; j<u; j++)
S1:    if (j<m)
         S(i, j+idx[i]);
   }
```

Program analysis

- Preprocessing
  - Subtract (dynamic) lower bounds.
  - Synthesize static upper bounds (static analysis or dynamic inspector).

```
   for (i=0; i<N; i++) {
S0:  lb = idx[i];
S1:  ub = idx[i+1];
     for (j=lb; j<ub; j++)
S2:    S(i, j);
   }
```

```
   for (i=0; i<N; i++) {
S0:  m = idx[i+1] - idx[i];
     for (j=0; j<m; j++)
S1:    S(i, j+idx[i]);
   }
```

- Modeling control dependences
  - Insert an exit predicate.
  - Delay the introduction of early exit.
  - Sink the dynamic conditions when targeting on GPUs.

```
   for (i=0; i<N; i++) {
S0:  m = idx[i+1] - idx[i];
     for (j=0; j<u; j++)
S1:    if (j<m)
         S(i, j+idx[i]);
   }
```

```
   for (i=0; i<N; i++)
     for (j=0; j<u; j++) {
S0:    m = idx[i+1] - idx[i];
S1:    if (j<m)
         S(i, j+idx[i]);
     }
```

Program analysis

Program analysis

```
   for (i=0; i<N; i++) {
     for (j=idx[i]; j<idx[i+1]; j++)
S1:     S(i, j);
   }
```

Before preprocessing

```
   for (i=0; i<N; i++)
     for (j=0; j<u; j++) {
S0:     m = idx[i+1] - idx[i];
S1:     if (j<m)
          S(i, j+idx[i]);
```

After preprocessing

Program analysis

```
   for (i=0; i<N; i++) {
     for (j=idx[i]; j<idx[i+1]; j++)
S1:      S(i, j);
   }
```

Before preprocessing

```
   for (i=0; i<N; i++)
     for (j=0; j<u; j++) {
S0:      m = idx[i+1] - idx[i];
S1:      if (j<m)
           S(i, j+idx[i]);
```

After preprocessing

Polyhedral representation (schedule tree)

Program analysis

```
   for (i=0; i<N; i++) {
     for (j=idx[i]; j<idx[i+1]; j++)
S1:    S(i, j);
   }
```

Before preprocessing

```
   for (i=0; i<N; i++)
     for (j=0; j<u; j++) {
S0:    m = idx[i+1] - idx[i];
S1:    if (j<m)
         S(i, j+idx[i]);
```

After preprocessing

Polyhedral representation (schedule tree)

domain

$\{S_1(i)\}$

$S_1(i) \to (i)$

Program analysis

```
   for (i=0; i<N; i++) {
     for (j=idx[i]; j<idx[i+1]; j++)
S1:    S(i, j);
   }
```

Before preprocessing

```
   for (i=0; i<N; i++)
     for (j=0; j<u; j++) {
S0:    m = idx[i+1] - idx[i];
S1:    if (j<m)
         S(i, j+idx[i]);
```

After preprocessing

Polyhedral representation (schedule tree)

domain

$\{S_1(i)\}$

$S_1(i) \to (i)$

domain

$\{S_0(i,j); S_1(i,j)\}$

$S_0(i,j) \to (i); S_1(i,j) \to (i); S_0(i,j) \to (j); S_1(i,j) \to (j)$

sequence

$S_0(i,j)$ $\quad$ $S_1(i,j)$

Program analysis

```
   for (i=0; i<N; i++) {
     for (j=idx[i]; j<idx[i+1]; j++)
S1:    S(i, j);
   }
```

Before preprocessing

```
   for (i=0; i<N; i++)
     for (j=0; j<u; j++) {
S0:    m = idx[i+1] - idx[i];
S1:    if (j<m)
         S(i, j+idx[i]);
```

After preprocessing

Polyhedral representation (schedule tree)

domain

$\{S_1(i)\}$

$S_1(i) \to (i)$

domain

$\{S_0(i,j); S_1(i,j)\}$

$S_0(i,j) \to (i); S_1(i,j) \to (i); S_0(i,j) \to (j); S_1(i,j) \to (j)$

sequence

$S_0(i,j)$ $\quad$ $S_1(i,j)$

Loop transformations like tiling would be impossible for the original code.

Schedule tree

domain

$\{S_0(i,j); S_1(i,j)\}$

$S_0(i,j) \to (i); S_1(i,j) \to (i); S_0(i,j) \to (j); S_1(i,j) \to (j)$

sequence

$S_0(i,j)$ $S_1(i,j)$

Schedule tree

domain

$\{S_0(i,j); S_1(i,j)\}$

$S_0(i,j) \to (i); S_1(i,j) \to (i); S_0(i,j) \to (j); S_1(i,j) \to (j)$

sequence

$S_0(i,j)$ $\qquad$ $S_1(i,j)$

- Core node types
  - Domain: set of statement instances to be scheduled
  - Band: multi-dimensional piecewise quasi-affine partial schedule
  - Filter: selects statement instances that are executed by descendants
  - Sequence/Set: children executed in given/arbitrary order

Schedule tree

domain

$\{S_0(i,j); S_1(i,j)\}$

$S_0(i,j) \to (i); S_1(i,j) \to (i); S_0(i,j) \to (j); S_1(i,j) \to (j)$

sequence

$S_0(i,j)$ $S_1(i,j)$

- Core node types
  - Domain: set of statement instances to be scheduled
  - Band: multi-dimensional piecewise quasi-affine partial schedule
  - Filter: selects statement instances that are executed by descendants
  - Sequence/Set: children executed in given/arbitrary order
- Convenience node types
  - Mark: attach additional information to subtrees
  - Extension: add additional domain elements to facilitate non-polyhedral semantics

Schedule transformation

- Schedule generation
  - Apply any affine transformation, e.g., a variant of the Pluto algorithm.
  - Insert a mark node below each band node associated with a dynamically counted loop.

Schedule transformation

- Schedule generation
  - Apply any affine transformation, e.g., a variant of the Pluto algorithm.
  - Insert a mark node below each band node associated with a dynamically counted loop.

```
   for (i=0; i<N; i++)
     for (j=0; j<u; j++) {
S0:    m = idx[i+1] - idx[i];
S1:    if (j<m)
         S(i, j+idx[i]);
     }
```

Schedule transformation

- Schedule generation
  - Apply any affine transformation, e.g., a variant of the Pluto algorithm.
  - Insert a mark node below each band node associated with a dynamically counted loop.

```
    for (i=0; i<N; i++)
      for (j=0; j<u; j++) {
S0:     m = idx[i+1] - idx[i];
S1:     if (j<m)
          S(i, j+idx[i]);
      }
```

domain

$\{S_0(i,j); S_1(i,j)\}$

$S_0(i,j) \to (i); S_1(i,j) \to (i); S_0(i,j) \to (j); S_1(i,j) \to (j)$

sequence

$S_0(i,j)$ $S_1(i,j)$

Schedule transformation

- Schedule generation
  - Apply any affine transformation, e.g., a variant of the Pluto algorithm.
  - Insert a mark node below each band node associated with a dynamically counted loop.

```
   for (i=0; i<N; i++)
     for (j=0; j<u; j++) {
S0:    m = idx[i+1] - idx[i];
S1:    if (j<m)
         S(i, j+idx[i]);
     }
```

domain

$\{S_0(i,j); S_1(i,j)\}$

$S_0(i,j) \to (i); S_1(i,j) \to (i); S_0(i,j) \to (j); S_1(i,j) \to (j)$

sequence

$S_0(i,j)$ $S_1(i,j)$

domain

$\{S_0(i,j); S_1(i,j)\}$

$S_0(i,j) \to (i); S_1(i,j) \to (i); S_0(i,j) \to (j); S_1(i,j) \to (j)$

mark: "*dynamic_counted_loop*"

sequence

$S_0(i,j)$ $S_1(i,j)$

Schedule transformation

- Schedule generation
  - Apply any affine transformation, e.g., a variant of the Pluto algorithm.
  - Insert a mark node below each band node associated with a dynamically counted loop.

```
    for (i=0; i<N; i++)
      for (j=0; j<u; j++) {
S0:     m = idx[i+1] - idx[i];
S1:     if (j<m)
          S(i, j+idx[i]);
      }
```

domain

$\{S_0(i,j); S_1(i,j)\}$

$S_0(i,j) \to (i); S_1(i,j) \to (i); S_0(i,j) \to (j); S_1(i,j) \to (j)$

sequence

$S_0(i,j)$ $S_1(i,j)$

domain

$\{S_0(i,j); S_1(i,j)\}$

$S_0(i,j) \to (i); S_1(i,j) \to (i); S_0(i,j) \to (j); S_1(i,j) \to (j)$

mark: "*dynamic_counted_loop*"

sequence

$S_0(i,j)$ $S_1(i,j)$

Schedule transformation

Perform any loop transformations, e.g., tiling.

domain

$\{S_0(i,j); S_1(i,j)\}$

$S_0(i,j) \to (i); S_1(i,j) \to (i); S_0(i,j) \to (j); S_1(i,j) \to (j)$

mark: "*dynamic_counted_loop*"

sequence

$S_0(i,j)$ $\quad$ $S_1(i,j)$

Schedule transformation

Perform any loop transformations, e.g., tiling.

domain

$\{S_0(i,j); S_1(i,j)\}$

$S_0(i,j) \to (i); S_1(i,j) \to (i); S_0(i,j) \to (j); S_1(i,j) \to (j)$

mark: "*dynamic_counted_loop*"

sequence

$S_0(i,j)$ $\quad$ $S_1(i,j)$

domain

$\{S_0(i,j); S_1(i,j)\}$

$S_0(i,j) \to (i/4); S_1(i,j) \to (i/4); S_0(i,j) \to (j/8); S_1(i,j) \to (j/8)$

mark: "*dynamic_counted_loop*"

$S_0(i,j) \to (i); S_1(i,j) \to (i); S_0(i,j) \to (j); S_1(i,j) \to (j)$

mark: "*dynamic_counted_loop*"

sequence

$S_0(i,j)$ $\quad$ $S_1(i,j)$

Schedule transformation

Perform any loop transformations, e.g., tiling.

domain

$\{S_0(i,j); S_1(i,j)\}$

$S_0(i,j) \to (i/4); S_1(i,j) \to (i/4); S_0(i,j) \to (j/8); S_1(i,j) \to (j/8)$

mark: "*dynamic_counted_loop*"

$S_0(i,j) \to (i); S_1(i,j) \to (i); S_0(i,j) \to (j); S_1(i,j) \to (j)$

mark: "*dynamic_counted_loop*"

sequence

$S_0(i,j)$ $S_1(i,j)$

Schedule transformation

Perform any loop transformations, e.g., tiling.

- domain
  - $\{S_0(i,j); S_1(i,j)\}$
    - $S_0(i,j) \to (i/4); S_1(i,j) \to (i/4); S_0(i,j) \to (j/8); S_1(i,j) \to (j/8)$
      - mark: "*dynamic_counted_loop*"
        - $S_0(i,j) \to (i); S_1(i,j) \to (i); S_0(i,j) \to (j); S_1(i,j) \to (j)$
          - mark: "*dynamic_counted_loop*"
            - sequence
              - $S_0(i,j)$
              - $S_1(i,j)$

- domain
  - $\{S_0(i,j); S_1(i,j)\}$
    - $S_0(i,j) \to (i/4); S_1(i,j) \to (i/4)$
      - mark: "*dynamic_counted_loop*"
        - $S_0(i,j) \to (j/8); S_1(i,j) \to (j/8)$
          - mark: "*dynamic_counted_loop*"
            - $S_0(i,j) \to (i); S_1(i,j) \to (i)$
              - mark: "*dynamic_counted_loop*"
                - $S_0(i,j) \to (j); S_1(i,j) \to (j)$
                  - mark: "*dynamic_counted_loop*"
                    - sequence
                      - $S_0(i,j)$
                      - $S_1(i,j)$

Schedule transformation

Replace each occurrence of mark nodes with an extension node.

domain

$\{S_0(i,j); S_1(i,j)\}$

$S_0(i,j) \to (i/4); S_1(i,j) \to (i/4); S_0(i,j) \to (j/8); S_1(i,j) \to (j/8)$

mark: "*dynamic_counted_loop*"

$S_0(i,j) \to (i); S_1(i,j) \to (i); S_0(i,j) \to (j); S_1(i,j) \to (j)$

mark: "*dynamic_counted_loop*"

sequence

$S_0(i,j)$ $S_1(i,j)$

domain

$\{S_0(i,j); S_1(i,j)\}$

$S_0(i,j) \to (i/4); S_1(i,j) \to (i/4)$

extension: "$[i0] \to exit()$"

$S_0(i,j) \to (j/8); S_1(i,j) \to (j/8)$

extension: "$[i0, i1] \to exit()$"

$S_0(i,j) \to (i); S_1(i,j) \to (i)$

extension: "$[i0, i1, i2] \to exit()$"

$S_0(i,j) \to (j); S_1(i,j) \to (j)$

extension: "$[i0, i1, i2, i3] \to exit()$"

sequence

$S_0(i,j)$ $S_1(i,j)$

Schedule transformation

Replace each occurrence of mark nodes with an extension node.

domain

$\{S_0(i,j); S_1(i,j)\}$

$S_0(i,j) \to (i/4); S_1(i,j) \to (i/4); S_0(i,j) \to (j/8); S_1(i,j) \to (j/8)$

mark: "*dynamic_counted_loop*"

$S_0(i,j) \to (i); S_1(i,j) \to (i); S_0(i,j) \to (j); S_1(i,j) \to (j)$

mark: "*dynamic_counted_loop*"

sequence

$S_0(i,j)$ $S_1(i,j)$

domain

$\{S_0(i,j); S_1(i,j)\}$

$S_0(i,j) \to (i/4); S_1(i,j) \to (i/4)$

extension: "$[i0] \to exit()$"

$S_0(i,j) \to (j/8); S_1(i,j) \to (j/8)$

extension: "$[i0, i1] \to exit()$"

$S_0(i,j) \to (i); S_1(i,j) \to (i)$

extension: "$[i0, i1, i2] \to exit()$"

$S_0(i,j) \to (j); S_1(i,j) \to (j)$

extension: "$[i0, i1, i2, i3] \to exit()$"

sequence

$S_0(i,j)$ $S_1(i,j)$

Schedule transformation

Replace each occurrence of mark nodes with an extension node.

- domain
  - $\{S_0(i,j); S_1(i,j)\}$
    - $S_0(i,j) \to (i/4); S_1(i,j) \to (i/4); S_0(i,j) \to (j/8); S_1(i,j) \to (j/8)$
      - mark: "*dynamic_counted_loop*"
        - $S_0(i,j) \to (i); S_1(i,j) \to (i); S_0(i,j) \to (j); S_1(i,j) \to (j)$
          - mark: "*dynamic_counted_loop*"
            - sequence
              - $S_0(i,j)$
              - $S_1(i,j)$

- domain
  - $\{S_0(i,j); S_1(i,j)\}$
    - $S_0(i,j) \to (i/4); S_1(i,j) \to (i/4)$
      - extension: "$[i0] \to exit()$"
        - $S_0(i,j) \to (j/8); S_1(i,j) \to (j/8)$
          - extension: "$[i0, i1] \to exit()$"
            - $S_0(i,j) \to (i); S_1(i,j) \to (i)$
              - extension: "$[i0, i1, i2] \to exit()$"
                - $S_0(i,j) \to (j); S_1(i,j) \to (j)$
                  - extension: "$[i0, i1, i2, i3] \to exit()$"
                    - sequence
                      - $S_0(i,j)$
                      - $S_1(i,j)$

Extension nodes are inserted everywhere an early exit statement may be needed, associated with the loop depth.

Code generation

domain

$\{S_0(i,j); S_1(i,j)\}$

$S_0(i,j) \rightarrow (i/4); S_1(i,j) \rightarrow (i/4)$

extension: "[$i0$] $\rightarrow$ $exit()$"

$S_0(i,j) \rightarrow (j/8); S_1(i,j) \rightarrow (j/8)$

extension: "[$i0$, $i1$] $\rightarrow$ $exit()$"

$S_0(i,j) \rightarrow (i); S_1(i,j) \rightarrow (i)$

extension: "[$i0$, $i1$, $i2$] $\rightarrow$ $exit()$"

$S_0(i,j) \rightarrow (j); S_1(i,j) \rightarrow (j)$

extension: "[$i0$, $i1$, $i2$, $i3$] $\rightarrow$ $exit()$"

sequence

$S_0(i,j)$ $S_1(i,j)$

Code generation

domain

$\{S_0(i,j); S_1(i,j)\}$

$S_0(i,j) \to (i/4); S_1(i,j) \to (i/4)$

extension: "$[i0] \to exit()$"

$S_0(i,j) \to (j/8); S_1(i,j) \to (j/8)$

extension: "$[i0, i1] \to exit()$"

$S_0(i,j) \to (i); S_1(i,j) \to (i)$

extension: "$[i0, i1, i2] \to exit()$"

$S_0(i,j) \to (j); S_1(i,j) \to (j)$

extension: "$[i0, i1, i2, i3] \to exit()$"

sequence

$S_0(i,j)$ $S_1(i,j)$

- loop depths (telling where to insert an early exit), loop iterators and associated predicate list (constructing the conditions) are known.

Code generation

domain

$\{S_0(i,j); S_1(i,j)\}$

$S_0(i,j) \rightarrow (i/4); S_1(i,j) \rightarrow (i/4)$

extension: "$[i0] \rightarrow exit()$"

$S_0(i,j) \rightarrow (j/8); S_1(i,j) \rightarrow (j/8)$

extension: "$[i0, i1] \rightarrow exit()$"

$S_0(i,j) \rightarrow (i); S_1(i,j) \rightarrow (i)$

extension: "$[i0, i1, i2] \rightarrow exit()$"

$S_0(i,j) \rightarrow (j); S_1(i,j) \rightarrow (j)$

extension: "$[i0, i1, i2, i3] \rightarrow exit()$"

sequence

$S_0(i,j)$ $S_1(i,j)$

- loop depths (telling where to insert an early exit), loop iterators and associated predicate list (constructing the conditions) are known.
- Whether a loop is dynamic counted can be determined.

Code generation

domain

$\{S_0(i,j); S_1(i,j)\}$

$S_0(i,j) \to (i/4); S_1(i,j) \to (i/4)$

extension: "$[i0] \to exit()$"

$S_0(i,j) \to (j/8); S_1(i,j) \to (j/8)$

extension: "$[i0, i1] \to exit()$"

$S_0(i,j) \to (i); S_1(i,j) \to (i)$

extension: "$[i0, i1, i2] \to exit()$"

$S_0(i,j) \to (j); S_1(i,j) \to (j)$

extension: "$[i0, i1, i2, i3] \to exit()$"

sequence

$S_0(i,j)$ $S_1(i,j)$

- loop depths (telling where to insert an early exit), loop iterators and associated predicate list (constructing the conditions) are known.
- Whether a loop is dynamic counted can be determined.
- Generate goto (with a label counter) for GPUs or change back to dynamic bounds for CPUs.

Code generation

- On GPUs, dynamic counted loops are enforced by `goto` statements, skipping empty iterations.
- On CPUs, dynamic conditions are taken back.
- Why different code generation templates are needed?
  - On GPUs, threads, thread blocks, need fix bounds.
  - On CPUs, early exits like `goto` are not allowed.

Code generation

```
   for (i=0; i<N; i++) {
     for (j=0; j<u1; j++) {
       for (k=0; k<u2; k++) {
         for (...) {
S0:        m = f(i);
S1:        n = g(i);
            ...
Sn:        if (j<m&&k<n&&...)
             S(i, j, k, ...);
           ...
         }
         ...
         if (k>=n)
           goto label_u_2;
       }
       label_u_2: ;
       if (j>=m)
         goto label_u_1;
     }
     label_u_1: ;
   }
```

code generation template for GPUs

```
   #pragma omp parallel for
   for (i=0; i<N; i++) {
S0:        m = f(i);
S1:        n = g(i);
            ...
     for (j=0; j<m; j++) {
       for (k=0; k<n; k++) {
         for (...) {
Sn:        S(i, j, k, ...);
         }
       }
     }
   }
```

code generation template for CPUs

Code generation

SpMV CSR code

```
 for (i=0; i<N; i++)
   for (j=0; j<u; j++) {
S0:  m = idx[i+1] - idx[i];
S1:  if (j<m)
       y[i] += A[j]*x[col[j]];
   }
```

Code generation

SpMV CSR code

```
 for (i=0; i<N; i++)
   for (j=0; j<u; j++) {
S0:   m = idx[i+1] - idx[i];
S1:   if (j<m)
        y[i] += A[j]*x[col[j]];
   }
```

```
 for (ii=0; ii<N/4; ii+=4){
S0:m=idx[ii+1]-idx[ii];
   for (jj=0; jj<m/8; jj+=8)
     for (i=0; i<=min(3,N-ii); i++)
       for (j=0; j<=min(7,m-jj); j
    ++)
S1:          y[ii+i] += A[jj+j]*x[col[jj
    +j]];
   }
```

Code generation

SpMV CSR code

```
 for (i=0; i<N; i++)
   for (j=0; j<u; j++) {
S0:  m = idx[i+1] - idx[i];
S1:  if (j<m)
       y[i] += A[j]*x[col[j]];
   }
```

```
 for (ii=0; ii<N/4; ii+=4){
S0:m=idx[ii+1]-idx[ii];
   for (jj=0; jj<m/8; jj+=8)
     for (i=0; i<=min(3,N-ii); i++)
       for (j=0; j<=min(7,m-jj); j
    ++)
S1:          y[ii+i] += A[jj+j]*x[col[jj
    +j]];
   }
```

```
   for (ii=32*b0; ii<N; ii+=8192) {
     for (jj=32*b1; jj<u; jj+=8192) {
       for (i=t0; i<=min(31,N-ii); i+=32)
         for(j=t1; i<=min(31,u-jj); i+=32) {
S0:        m = idx[ii+i+1] - idx[ii+i];
S1:        if (jj+j<m)
             y[ii+i] += A[jj+j]*x[col[jj+j]];
           if (jj+j>=m)
             goto label0;
         }label0: ;
       if (jj>=m)
         goto label1;
     }label1: ;
   }
```

General applicability

General applicability

- Affine transformations: loop tiling, skewing, shifting, interchange, etc.

General applicability

- Affine transformations: loop tiling, skewing, shifting, interchange, etc.
- Special cases have to be taken to handle loop fusion.

General applicability

- Affine transformations: loop tiling, skewing, shifting, interchange, etc.
- Special cases have to be taken to handle loop fusion.

```
   for (i=0; i<N; i++) {
     for(j=0; j<u1; j++) {
S0:    m=f(i);
       if(j<m);
S1:      S1(i,j);
     }
   }
   for (i=0; i<N; i++) {
     for(j=0; j<u2; j++) {
S2:    n=g(i);
       if(j<n);
S3:      S3(i,j);
     }
   }
```

Before fusion

```
   for (i=0; i<N; i++) {
     for(j=0; j<max(u1,u2); j++) {
S0:    m=f(i);
S2:    n=g(i);
       if(j<m);
S1:      S1(i,j);
       if(j<n);
S3:      S3(i,j);
       if(j>=m && j>=n)
         goto label0;
     }label0: ;
   }
```

After fusion

General applicability

- Affine transformations: loop tiling, skewing, shifting, interchange, etc.
- Special cases have to be taken to handle loop fusion.

```
    for (i=0; i<N; i++) {
      for(j=0; j<u1; j++) {
S0:     m=f(i);
        if(j<m);
S1:       S1(i,j);
      }
    }
    for (i=0; i<N; i++) {
      for(j=0; j<u2; j++) {
S2:     n=g(i);
        if(j<n);
S3:       S3(i,j);
      }
    }
```

Before fusion

```
    for (i=0; i<N; i++) {
      for(j=0; j<max(u1,u2); j++) {
S0:     m=f(i);
S2:     n=g(i);
        if(j<m);
S1:       S1(i,j);
        if(j<n);
S3:       S3(i,j);
        if(j>=m && j>=n)
          goto label0;
      }label0: ;
    }
```

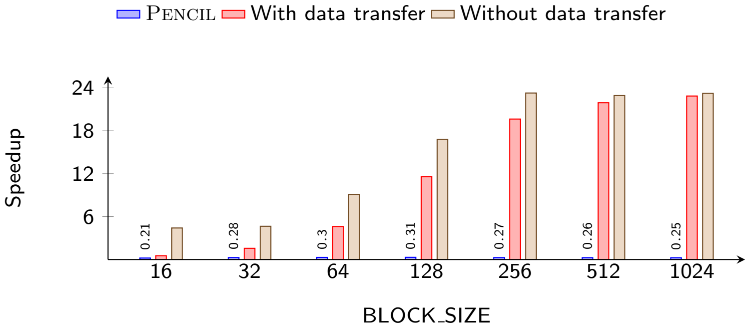
After fusion

- A normal loop can be treated as a specific case of dynamic counted loop by reasoning on its static upper bound as a predicate.

Setup and methodology

- Input: C programs with PENCIL extensions
- Code generator: PPCG (`ppcg-0.05-197-ge774645-pencilcc`)
- Output:
  - CUDA code for GPUs
  - OpenMP code for CPUs
- Architectures:
  - GPUs: NVIDIA Quadro K4000
  - CPUs: 12-core Intel Xeon(R) E5-2630 v2 @2.60GHz
- Compilation:
  - CUDA code: nvcc7.5.15 (*-O3*)
  - OpenMP code: icc17.0.0 (*-Ofast -fstrict-aliasing -qopenmp*)
- Methodology: Run each benchmark 9 times and retain the median value.

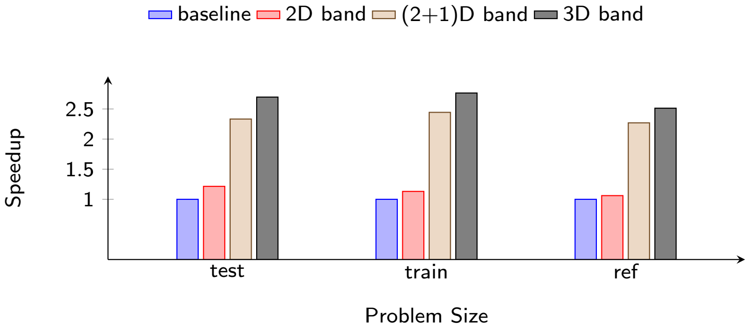
Evaluation on GPUs



Performance of the HOG descriptor on GPU

- *BLOCK_SIZE*: defines the size of an image block.
- Our technique can obtain a speedup ranging from 4.4× to 23.3× while PPCG suffers from a degradation by about 75%, illustrating the importance of parallelizing and optimizing dynamic counted loops.

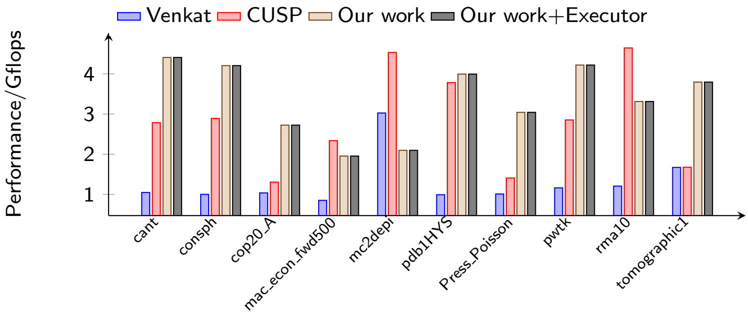
Evaluation on GPUs



Performance of equake on GPU

- $2D$: a 2-dimensional permutable band on the dynamic counted loop, enabling unrolling.
- $(2+1)D$: a 2-dimensional outer band and an inner band (dynamic counted loop), enabling interchange.
- $3D$: a 3-dimensional permutable band on the dynamic counted loop, enabling fusion.
- Handling dynamic counted loops enables more loop transformations, leading to performance improvements in each case.

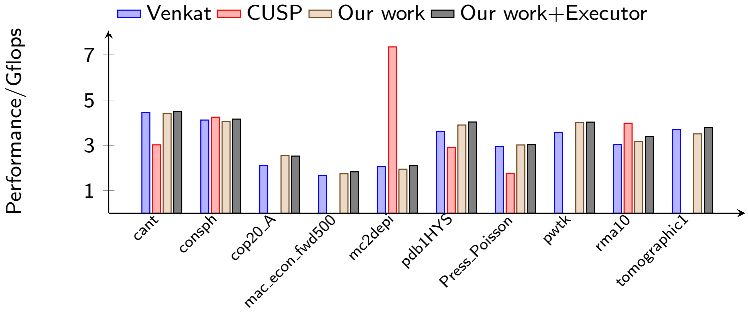
Evaluation on GPUs



Performance of the CSR SpMV on GPU

- PENCIL extension is used to deal with indirect accesses (subscripts of subscripts).
- Our technique enables tiling automatically, neither resorting to transformations like make-dense, compact-and-pad, etc, nor assuming the tiling sizes are divisible by loop iteration times like Venkat et al.'s work [VHS15].
- Our technique can also apply to the executor of Venkat et al.'s work [VHS15] as a complementary optimization.

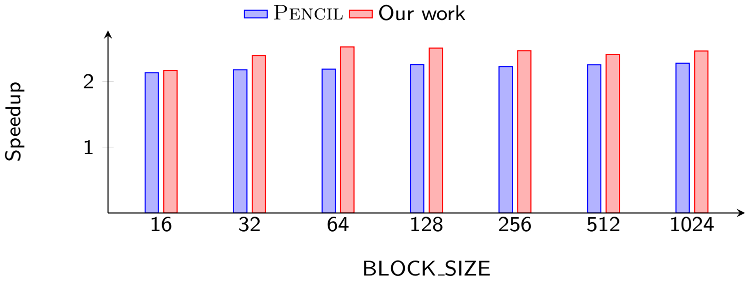
Evaluation on GPUs



Performance of the ELL SpMV on GPU

- Venkat et al. [VHS15] derived ELL from CSR by tiling the dynamic counted loop with the maximum number of nonzero entries in a row. No early exit statements exist in their code.
- Our technique emits early exit statements when there are fewer non-zeros in a row, minimizing the number of iterations of the dynamic counted loop.
- The CUSP library [BG09] encounters a *format_conversion* with some input matrices, while our technique remains applicable on all formats.

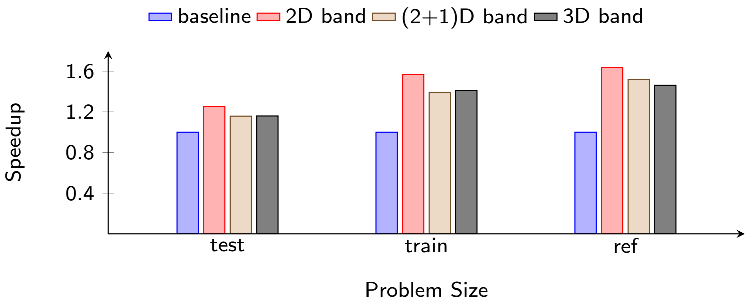
Evaluation on CPUs



Performance of the HOG descriptor on CPU

- The original dynamic condition can be taken back when generating OpenMP code on CPU architectures, avoiding the combination of nested bands and the refactoring of the control flow.
- Our technique enables aggressive loop transformations including tiling, interchange, etc., leading to a better performance when these optimizations are turned on.

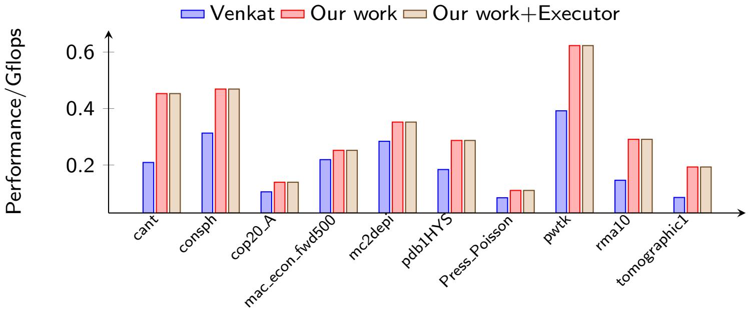
Evaluation on CPUs



Performance of `equake` on CPU

- The original dynamic condition can be taken back when generating OpenMP code on CPU architectures, avoiding the combination of nested bands and the refactoring of the control flow.
- Our technique enables aggressive loop transformations including tiling, interchange, etc., leading to a better performance when these optimizations are turned on.

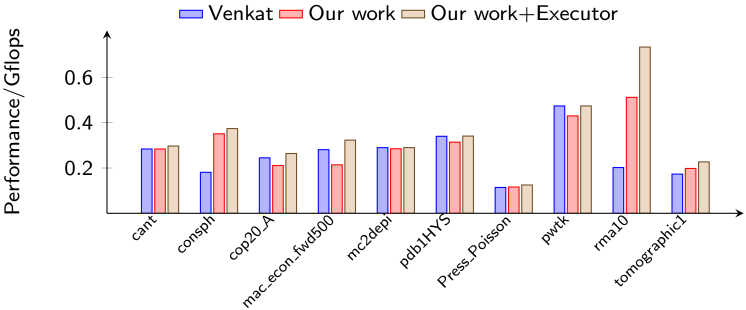
Evaluation on CPUs



Performance of the CSR SpMV on CPU

- The original dynamic condition can be taken back when generating OpenMP code on CPU architectures, avoiding the combination of nested bands and the refactoring of the control flow.
- Our technique enables aggressive loop transformations including tiling, interchange, etc., leading to a better performance when these optimizations are turned on.

Evaluation on CPUs



Performance of the ELL SpMV on CPU

- The original dynamic condition can be taken back when generating OpenMP code on CPU architectures, avoiding the combination of nested bands and the refactoring of the control flow.
- Our technique enables aggressive loop transformations including tiling, interchange, etc., leading to a better performance when these optimizations are turned on.

Conclusion

- We model control dependences on data-dependent predicates by revisiting the work of Benabderrahmane et al. [BPCB10].
- Our technique does not resort to more expressive first-order logic with non-interpreted functions/predicates, like [SCF03, SLC^{+}16].
- We implement a schedule-tree-based algorithm to fully automate the framework.
- Our work provides code generation templates for multiple scenarios, including the inspector-executor scheme [VHS15].
- We show an in-depth performance comparison with the state of the art, with both CPU and GPU platforms being taken into consideration.

References

- Nathan Bell and Michael Garland.
  Implementing sparse matrix-vector multiplication on throughput-oriented processors.
  In *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, page No. 18. ACM, 2009.
- Mohamed-Walid Benabderrahmane, Louis-Noël Pouchet, Albert Cohen, and Cédric Bastoul.
  The polyhedral model is more widely applicable than you think.
  In *Proceedings of 19th International Conference on Compiler Construction*, pages 283–303. Springer, 2010.
- Michelle Mills Strout, Larry Carter, and Jeanne Ferrante.
  Compile-time composition of run-time data and iteration reorderings.
  In *Proceedings of the ACM SIGPLAN 2003 conference on Programming Language Design and Implementation*, pages 91–102. ACM, 2003.
- Michelle Mills Strout, Alan LaMielle, Larry Carter, Jeanne Ferrante, Barbara Kreaseck, and Catherine Olschanowsky.
  An approach for code generation in the sparse polyhedral framework.
  Parallel Computing, 53:32–57, 2016.
- Anand Venkat, Mary Hall, and Michelle Strout.
  Loop and data transformations for sparse matrix code.
  In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 521–532, 2015.