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

1 Introduction
   - Dynamic counted loops
   - Examples
   - The polyhedral model

2 Polyhedral compilation of dynamic counted loops
   - Program analysis
   - Schedule tree
   - Schedule transformation
   - Code generation
   - General applicability

3 Experimental results
   - Setup and methodology
   - Evaluation on GPUs
   - Evaluation on CPUs

4 Conclusion
Dynamic counted loops

What are *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.

```plaintext
for (i=0; i<N; i++) {
  lb = idx[i];
  ub = idx[i+1];
  for (j=lb; j<ub; j++) // dynamically computed bounds
    S(i, j);
}
```
Dynamic counted loops

Why are we interested in the class of loop nest kernels involving 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
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
- ...

Real-life examples of dynamic counted loops
The polyhedral model

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

```c
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.

```plaintext
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)]

---

^aConsider 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).

```c
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:** \{\} \footnote{Consider only true/flow dependences.}

- **Schedule:** [S_0(i) \rightarrow (i, 0); S_1(i) \rightarrow (i, 1); S_2(i) \rightarrow (i, 2)]
The polyhedral model

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

```c
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 \land 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 \land lb \leq j < ub; S_2(i, j) \rightarrow ub : 0 \leq i < N \land lb \leq j < ub\}

- **Dependence:**
  \{S_0[i] \rightarrow S_2[i' = i, j] : 0 \leq i < N \land lb \leq j < ub; S_1[i] \rightarrow S_2[i' = i, j] : 0 \leq i < N \land 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

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);
}

Polyhedral compilation of dynamic counted loops

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 < m; j++)
    S1: 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

January 24, 2018 8 / 21
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).

```c
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);
}
```

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

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

- Polyhedral compilation of dynamic counted loops
Preprocessing

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

```c
for (i = 0; i < N; i++) {
    lb = idx[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++) {
    m = idx[i+1] - idx[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).

```c
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);
}
```

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

```c
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]);
}
```
Preprocessing

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

```c
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);
}
```

Modeling control dependences

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

```c
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).

```c
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);
}
```

```c
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.

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

```c
for (i=0; i<N; i++)
    for (j=0; j<u; j++)
        if (j<m)
            S(i, j+idx[i]);
```
Before preprocessing

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

After preprocessing

Polyhedral representation (schedule tree)

domain \{ \text{S1}(i) \}

\text{S1}(i) \rightarrow (i)

domain \{ \text{S0}(i, j), \text{S1}(i, j) \}

\text{S0}(i, j) \rightarrow (i);
\text{S1}(i, j) \rightarrow (i);
\text{S0}(i, j) \rightarrow (j);
\text{S1}(i, j) \rightarrow (j);

sequence \text{S0}(i, j) \text{S1}(i, j)
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)

\[
S_1(i) ightarrow (i)
\]

\[
S_0(i, j) ightarrow (i); \\
S_1(i, j) ightarrow (i); \\
S_0(i, j) ightarrow (j); \\
S_1(i, j) ightarrow (j)
\]

sequence

Polyhedral compilation of dynamic counted loops
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

Before preprocessing

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

After 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]);
}
```

Polyhedral representation (schedule tree)

```
domain
   \{S_1(i)\}
S_1(i) \rightarrow (i)
```
Program analysis

Before preprocessing

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

After preprocessing

```plaintext
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]);
    }
```

Polyhedral representation (schedule tree)

```
domain

{S1(i)}

S1(i) \rightarrow (i)

domain

{S0(i,j); S1(i,j)}

S0(i,j) \rightarrow (i); S1(i,j) \rightarrow (i); S0(i,j) \rightarrow (j); S1(i,j) \rightarrow (j)

sequence

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

S0(i,j) → (i); S1(i,j) → (j); S0(i,j) → (j); S1(i,j) → (j)

Loop transformations like tiling would be impossible for the original code.
domain
\{S_0(i,j); S_1(i,j)\}

\begin{align*}
S_0(i,j) \rightarrow (i); & \quad S_1(i,j) \rightarrow (i); & \quad S_0(i,j) \rightarrow (j); & \quad S_1(i,j) \rightarrow (j)
\end{align*}

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
Schedule tree

- 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.

```
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.

```c
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.

```c
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

\[
\{S_0(i,j); S_1(i,j)\}
\]

Sequence

\[
S_0(i,j) \rightarrow (i); S_1(i,j) \rightarrow (i); S_0(i,j) \rightarrow (j); S_1(i,j) \rightarrow (j)
\]
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.

```latex
\begin{align*}
\text{for } (i=0; i<N; i++) \\
\quad \text{for } (j=0; j<u; j++) \\
\quad \text{S0: } & m = \text{idx}[i+1] - \text{idx}[i] \\
\quad \text{S1: } & \text{if } (j<m) \\
\quad & \text{S}(i, j+\text{idx}[i]); \\
\end{align*}
```

\[S_0(i, j) \rightarrow (i); S_1(i, j) \rightarrow (i); S_0(i, j) \rightarrow (j); S_1(i, j) \rightarrow (j)\]

\[\text{domain } \{S_0(i, j); S_1(i, j)\}\]

\[\text{mark: } \text{"dynamic\_counted\_loop"}\]

\[\text{sequence } \text{S}_0(i, j) \quad \text{S}_1(i, j)\]
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.

```plaintext
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]);
  }
```

```
```

```
```

```
```

```plaintext
mark: "dynamic_counted_loop"
```
Schedule transformation

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

\[ \text{domain} \]
\[ \{ S_0(i, j); S_1(i, j) \} \]
\[ S_0(i, j) \rightarrow (i); S_1(i, j) \rightarrow (i); S_0(i, j) \rightarrow (j); S_1(i, j) \rightarrow (j) \]
\[ \text{mark: "dynamic\_counted\_loop"} \]
\[ \text{sequence} \]
\[ S_0(i, j) \quad S_1(i, j) \]
Schedule transformation

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

```
domain
{S_0(i, j); S_1(i, j)}
S_0(i, j) → (i); S_1(i, j) → (i); S_0(i, j) → (j); S_1(i, j) → (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) → (i/4); S_1(i, j) → (i/4); S_0(i, j) → (j/8); S_1(i, j) → (j/8)
mark: "dynamic_counted_loop"
sequence
S_0(i, j)       S_1(i, j)
```
Schedule transformation

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

![Diagram showing loop transformations]

- **domain**
  \[ \{ S_0(i, j); S_1(i, j) \} \]
  \[ S_0(i, j) \rightarrow (i/4); S_1(i, j) \rightarrow (i/4); S_0(i, j) \rightarrow (j/8); S_1(i, j) \rightarrow (j/8) \]
  mark: "dynamic counted loop"

- **sequence**
  \[ S_0(i, j) \rightarrow (i); S_1(i, j) \rightarrow (i); S_0(i, j) \rightarrow (j); S_1(i, j) \rightarrow (j) \]
  mark: "dynamic counted loop"
Perform any loop transformations, e.g., tiling.
Replace each occurrence of mark nodes with an extension node.

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

S_0(i, j) → (i/4); S_1(i, j) → (i/4); S_0(i, j) → (j/8); S_1(i, j) → (j/8)

mark: "dynamic_counted_loop"

S_0(i, j) → (i); S_1(i, j) → (i); S_0(i, j) → (j); S_1(i, j) → (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) → (i/4); S_1(i, j) → (i/4)

extension: "[i0] → exit()"

S_0(i, j) → (j/8); S_1(i, j) → (j/8)

extension: "[i0, i1] → exit()"

S_0(i, j) → (i); S_1(i, j) → (i)

extension: "[i0, i1, i2] → exit()"

S_0(i, j) → (j); S_1(i, j) → (j)

extension: "[i0, i1, i2, i3] → exit()"

sequence

S_0(i, j)           S_1(i, j)
Replace each occurrence of mark nodes with an extension node.

**domain**

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

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

mark: "dynamic_counted_loop"

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

mark: "dynamic_counted_loop"

**sequence**

\(S_0(i, j) \rightarrow S_1(i, j)\)
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) \rightarrow (i/4); S_1(i, j) \rightarrow (j/8);
mark: "dynamic_counted_loop"

S_0(i, j) \rightarrow (i); S_1(i, j) \rightarrow (j);
mark: "dynamic_counted_loop"

sequence
S_0(i, j) \quad S_1(i, j)
```

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

```
Domain
\{S_0(i, j); S_1(i, j)\}

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

extension: "[0] \rightarrow exit()"

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

extension: "[i_0, i_1] \rightarrow exit()"

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

extension: "[i_0, i_1, i_2] \rightarrow exit()"

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

extension: "[i_0, i_1, i_2, i_3] \rightarrow exit()"

sequence
S_0(i, j) \quad S_1(i, j)
```
Code generation

\[
\begin{align*}
\text{domain} & \quad \{S_0(i, j); S_1(i, j)\} \\
S_0(i, j) & \rightarrow (i/4); S_1(i, j) \rightarrow (i/4) \\
\text{extension: } & \quad "[i0] \rightarrow \text{exit()}" \\
S_0(i, j) & \rightarrow (j/8); S_1(i, j) \rightarrow (j/8) \\
\text{extension: } & \quad "[i0, i1] \rightarrow \text{exit()}" \\
S_0(i, j) & \rightarrow (i); S_1(i, j) \rightarrow (i) \\
\text{extension: } & \quad "[i0, i1, i2] \rightarrow \text{exit()}" \\
S_0(i, j) & \rightarrow (j); S_1(i, j) \rightarrow (j) \\
\text{extension: } & \quad "[i0, i1, i2, i3] \rightarrow \text{exit()}" \\
\text{sequence} & \\
S_0(i, j) & \quad \quad S_1(i, j)
\end{align*}
\]
- Loop depths (telling where to insert an early exit), loop iterators and associated predicate list (constructing the conditions) are known.
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.
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.
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.
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, ...);

            }
        }
    }
}

#pragma omp parallel for
for (i=0; i<N; i++) {
    S0: m = f(i);
    S1: n = g(i);
    ...
    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
code generation template for CPUs
SpMV CSR code

```c
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]];
    }
```

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

```c
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]];
            }
    }
}
```

```
```
SpMV CSR code

```c
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) {
  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]];
        else goto label0 ;
      label0 : ;
    if (jj >=m) goto label1 ;
  label1 : ;
}
```

Polyhedral compilation of dynamic counted loops
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)
            \( y[i] += A[j] \times x[col[j]] \);
    }

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: \( y[ii+i] += A[jj+j] \times x[col[jj+j]] \);
} label0:

if (jj >=m) goto label1;
if (jj >=m)
    goto label1;
}

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

Polyhedral compilation of dynamic counted loops
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 ;
    }
}
```

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

```c
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

```c
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
Affine transformations: loop tiling, skewing, shifting, interchange, etc.

Special cases have to be taken to handle loop fusion.

Before fusion

\[
\text{for } (i=0; i<N; i++) \{
    \text{for } (j=0; j<u1; j++) \{
        \text{S0: } m=f(i);
        \text{if } (j < m);
        \text{S1: } S1(i,j);
    \}
\}
\]

\[
\text{for } (i=0; i<N; i++) \{
    \text{for } (j=0; j<u2; j++) \{
        \text{S2: } n=g(i);
        \text{if } (j < n);
        \text{S3: } S3(i,j);
    \}
\}
\]

After fusion

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

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.
**BLOCK_SIZE**: defines the size of an image block.

- Our technique can obtain a speedup ranging from $4.4 \times$ to $23.3 \times$ while PPCG suffers from a degradation by about 75%, illustrating the importance of parallelizing and optimizing dynamic counted loops.
Handling dynamic counted loops enables more loop transformations, leading to performance improvements in each case.

- **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.
**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.
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 \textit{format\_conversion} with some input matrices, while our technique remains applicable on all formats.
Experimental results

Evaluation on CPUs

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.
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.
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.
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.
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


