A polyhedral compilation framework for loops with dynamic data-dependent bounds

Jie Zhao and Albert Cohen

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

7th International Workshop on Polyhedral Compilation Techniques
(IMPACT 2017)
Stockholm, Sweden

January 23, 2017
Outline

1. Introduction
   - Motivation
   - Examples

2. Polyhedral compilation of dynamic counted loops
   - Schedule tree
   - Program analysis
   - Code generation

3. Experimental results
   - HOG descriptor
   - SpMV computations
   - Inspector-executor codes

4. Conclusion
1 Introduction
   • Motivation
   • Examples

2 Polyhedral compilation of dynamic counted loops
   • Schedule tree
   • Program analysis
   • Code generation

3 Experimental results
   • HOG descriptor
   • SpMV computations
   • Inspector-executor codes

4 Conclusion
Motivation

What are dynamic counted loops?
What are dynamic counted loops?

Definition

Counted loops with dynamic data-dependent bounds are regular counted loops with numerical constant strides, whose lower and/or upper bound may not be an affine function of enclosing loop counters and loop-invariant parameters.
Motivation

What are dynamic counted loops?

Definition

Counted loops with dynamic data-dependent bounds are regular counted loops 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++) {
    m = condition; // dynamically computed upper bound
    for (j=0; j<m; j++)
        S;
}
```
Motivation

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

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.
An illustrative example

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

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

A general while loop
A dynamic counted loop
An illustrative example

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

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

A general while loop

A dynamic counted loop
An illustrative example

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

A general while loop

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

A dynamic counted loop

Introduction  Examples  January 23, 2017  6 / 26
More examples

```java
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++)
            S2: S(i, j, k);
}
```

Histogram of Oriented Gradients (HOG)

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

Sparse matrix-vector CSR

```java
for (k=0; k<2*M-1; k++) {
    S0: m = f(k);
    for (i=0; i<m; i++) {
        S1: n = g(i);
        for (j=0; j<n; j++)
            S2: S(k, i, j);
    }
}
```

Sparse matrix-vector DIA

```java
for (i=0; i<N; i++) {
    S0: m = f(i, K);
    for (jj=0; jj<m; jj++)
        S1: n = g(i, jj, K);
    for (j=0; j<n; j++)
        S2: S(i, jj, j);
}
```

Sparse matrix-vector ELL

CSR: compressed sparse row

DIA: diagonal

ELL: ELLPack.
More examples

```c
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++)
            S2: S(i, j, k);
}
```

**Histogram of Oriented Gradients (HOG)**

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

**Sparse matrix-vector CSR**

```c
for (k=0; k<2*M-1; k++) {
    S0: m = f(k);
    for (i=0; i<m; i++)
        S1: n = g(i);
        for (j=0; j<n; j++)
            S2: S(k, i, j);
}
```

**Sparse matrix-vector DIA**

```c
for (i=0; i<N; i++) {
    S0: m = f(i, K);
    for (jj=0; jj<m; jj++)
        S1: n = g(i, jj, K);
        for (j=0; j<n; j++)
            S2: S(i, jj, j);
}
```

**Sparse matrix-vector ELL**

```c
for (i=0; i<N; i++) {
    S0: m = f(i, K);
    for (jj=0; jj<m; jj++)
        S1: n = g(i, jj, K);
        for (j=0; j<n; j++)
            S2: S(i, jj, j);
}
```

**CSR: compressed sparse row**

**DIA: diagonal**

**ELL: ELLPack.**
Outline

1 Introduction
   - Motivation
   - Examples

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

3 Experimental results
   - HOG descriptor
   - SpMV computations
   - Inspector-executor codes

4 Conclusion
Schedule tree (in isl)

```
domain
|
context: [u]
|
{S_0(i); S_1(i,j)}
|
S_0(i) → (i); S_1(i,j) → (i)
|
sequence

S_0(i)  {S_1(i,j) : j < u}

S_1(i,j) → (j)
```
Core node types

- **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 (in isl)

- **Core node types**
  - 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

- **“External” node types**
  - Domain: set of statement instances to be scheduled
  - Context: external constraints on symbolic constants
Schedule tree (in isl)

domain
\[\text{context: } [u]\]
\{S_0(i); S_1(i, j)\}
\[S_0(i) \rightarrow (i); S_1(i, j) \rightarrow (i)\]
sequence
\[S_0(i) \rightarrow \{S_1(i, j) : j < u\}\]

- **Core node types**
  - **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

- **“External” node types**
  - **Domain**: set of statement instances to be scheduled
  - **Context**: external constraints on symbolic constants

- **Convenience node types**
  - **Mark**: attach additional information to subtrees
Program analysis

Preprocessing

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

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

```c
for (i = 0; i < N; i++) {
    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 definition and check at the beginning of each iteration of a dynamically counted loop.
- Delay the introduction of break instructions until code generation to keep the control flow in a manageable form for a polyhedral compiler.

```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]);
}
```
Preprocessing

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

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

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

Modeling control dependences

- Insert an exit predicate definition and check at the beginning of each iteration of a dynamically counted loop.
- Delay the introduction of break instructions until code generation to keep the control flow in a manageable form for a polyhedral compiler.

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

```c
for (i = 0; i < N; i++) {
    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 definition and check at the beginning of each iteration of a dynamically counted loop.
- Delay the introduction of break instructions until code generation to keep the control flow in a manageable form for a polyhedral compiler.

```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]);
}
```
Program analysis

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

```c
for (i = 0; i < N; i++) {
    for (j = idx[i]; j < idx[i + 1]; j++)
        S1: 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 definition and check at the beginning of each iteration of a dynamically counted loop.
  - Delay the introduction of break instructions until code generation to keep the control flow in a manageable form for a polyhedral compiler.
Preprocessing
- Subtract (dynamic) lower bounds.
- Synthesize static upper bounds (static analysis or dynamic inspector).

Modeling control dependences
- Insert an exit predicate definition and check at the beginning of each iteration of a dynamically counted loop.
- Delay the introduction of break instructions until code generation to keep the control flow in a manageable form for a polyhedral compiler.
Program analysis

- 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 counter loop.
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 counter loop.

```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]);
}
```

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

mark: "dynamic counted loop (j, j<u)"
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 counter loop.

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

```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

context: [u]

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

S0(i) → (i); S1(i, j) → (i)

sequence

S0(i) {S1(i, j) : j < u}

S1(i, j) → (j)
```

Polyhedral compilation of dynamic counted loops
Program analysis

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 counter loop.

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

domain
context: [u]

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

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

sequence

S_0(i) \quad \{S_1(i, j) : j < u\}

S_1(i, j) \rightarrow (j)

Polyhedral compilation of dynamic counted loops

Program analysis

January 23, 2017 11 / 26
Program analysis

**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 counter loop.

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

**domain**
- context: [u]
- \{S_0(i); S_1(i, j)\}  
  \(S_0(i) \rightarrow (i); S_1(i, j) \rightarrow (i)\)
  \(S_0(i) \rightarrow (i); S_1(i, j) \rightarrow (j)\)
  \(\{S_1(i, j) : j < u\}\)

**mark:** "dynamic_counted_loop([j], j < u)"

**sequence**
- \(S_0(i, j) \rightarrow (j)\)
- \(S_1(i, j) \rightarrow (j)\)
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 counter loop.

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

domain context: [u]

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

S0(i) → (i); S1(i,j) → (i)

sequence

S0(i) → (i); S1(i,j) → (j)

mark: "dynamic_counted_loop([j], j < u)"
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 counter loop.

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

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

Domain context: \([u]\)

- \(S_0(i); S_1(i,j)\)
- \(S_0(i) \rightarrow (i); S_1(i,j) \rightarrow (i)\)
- \(S_0(i) \rightarrow (i); \{S_1(i,j) : j < u\}\)
- \(S_1(i,j) \rightarrow (j)\)

Mark: "dynamic_counted_loop([j], j < u)"
About the general applicability of affine transformations.
About the general applicability of affine transformations

- by default, resort to unoptimized exit-predicated execution with static upper bounds
- simple yet optimized code generation template for tiling, strip mining, skewing, interchange.
- the template does not apply to fusion and reversal.
About the general applicability of affine transformations

- by default, resort to unoptimized exit-predicated execution with static upper bounds
- simple yet optimized code generation template for tiling, strip mining, skewing, interchange.
- the template does not apply to fusion and reversal.
About the general applicability of affine transformations

- by default, resort to unoptimized exit-predicated execution with static upper bounds
- simple yet optimized code generation template for tiling, strip mining, skewing, interchange.
- the template does not apply to fusion and reversal.

Code generation

domain

context: \([u]\)

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

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

mark: "dynamic_counted_loop([j], j < u)"

sequence

\(S_0(i, j)\) \hspace{1cm} \(S_1(i, j)\)
The code generation template

```c
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)
        break;
    }
    if (j >= m)
      break;
  }
...
The code generation template

```c
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)
                break;
        }
    if (j>=m)
        break;
} 
```
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]];
        if (j >=m)
            break;
    }
```

```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]];
                if (jj+j >=m)
                    break;
            }
        if (jj >=m)
            break;
    }
```
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]];
            if (j >= m)
                break;
    }
```

```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]];
            if (j >= m)
                break;
    }
```
SpMV CSR code

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

```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) {
                m = idx[ii+i+1] - idx[ii+i];
                if (jj+j < m)
                    y[ii+i] += A[jj+j] * x[col[jj+j]];
                if (jj+j >= m)
                    break;
            }
        if (jj >= m)
            break;
    }
```
Outline

1 Introduction
   • Motivation
   • Examples

2 Polyhedral compilation of dynamic counted loops
   • Schedule tree
   • Program analysis
   • Code generation

3 Experimental results
   • HOG descriptor
   • SpMV computations
   • Inspector-executor codes

4 Conclusion
Performance of the HOG descriptor including data transfer time\textsuperscript{1}

![Graph showing performance of HOG descriptor with block sizes and speedup values.]

\textsuperscript{1}Running on NVIDIA Quadro K4000 GPU with Intel Xeon E5-2630 CPU.
Our work enables the effective parallelization and optimization of HOG on the target platform, outperforming sequential execution and PPCG’s inner parallelism version (total execution time including data transfers).

---

1 Running on NVIDIA Quadro K4000 GPU with Intel Xeon E5-2630 CPU.
Performance of the HOG descriptor without data transfer time\(^2\)

The performance is the speedup w.r.t. the sequential execution time.
CUDA vs. sequential execution time of CSR SpMV

![CUDA vs. sequential execution time of CSR SpMV](image)

- Based on the University of Florida sparse matrix collection [DH11]
Experimental results

SpMV computations

CUDA vs. sequential execution time of DIA SpMV

- cant
- consph
- cop20_A
- mac_econ_fwd500
- mc2depi
- pdb1HYS
- Press_Poisson
- pwtk

Execution time/s

Sequential vs. CUDA
SpMV computations

CUDA vs. sequential execution time of ELL SpMV

- Experimental results
- January 23, 2017

The diagram compares the execution time of SpMV computations using CUDA and sequential execution across various datasets. The x-axis represents different datasets such as 'cant', 'consph', 'cop20_A', 'mac_econ_fwd500', 'mc2depi', 'pdb1HYS', 'Press_Poisson', 'pwtk', 'rma10', and 'tomographic1'. The y-axis represents the execution time in milliseconds (ms). The datasets are color-coded differently for CUDA (red) and sequential (blue) execution, showing the comparison in performance.
An inspector-executor implementation of CSR SpMV

```c
for (i=0; i<M; i++) {
    for (k=0; k<N; k++) {
        marked = false;
        for (j=idx[i]; j<idx[i+1]; j++)
            if (k == col[j])
                if (!marked) {
                    marked = true;
                    exp_idx[count] = k;
                    count++;
                }
    }
    f_idx[i+1] = count;
}
```

inspector

```c
for (i=0; i<N; i++) {
    m = f_idx[i+1] - f_idx[i];
    for (j=0; j<m; j++)
        if (j<m)
            y[i] += val[j+f_idx[i]] * x[exp_idx[j+f_idx[i]]];
}
```

executor

Experimental results
CUDA vs. sequential execution time of the CSR SpMV executor

![Graph showing execution time comparison between Sequential and CUDA for various codes.](image)
Outline

1. Introduction
   - Motivation
   - Examples

2. Polyhedral compilation of dynamic counted loops
   - Schedule tree
   - Program analysis
   - Code generation

3. Experimental results
   - HOG descriptor
   - SpMV computations
   - Inspector-executor codes

4. Conclusion
Our work

Future work
Conclusion

- **Our work**
  - models control dependences on data-dependent predicates by revisiting the work of Benabderrahmane et al. [BPCB10].
  - does not resort to more expressive first-order logic with non-interpreted functions/predicates, like [SCF03, SLC16].
  - provides code generation templates for multiple scenarios, including the inspector-executor scheme [VHS15].

- **Future work**
Our work

- models control dependences on data-dependent predicates by revisiting the work of Benabderrahmane et al. [BPCB10].
- does not resort to more expressive first-order logic with non-interpreted functions/predicates, like [SCF03, SLC'16].
- provides code generation templates for multiple scenarios, including the inspector-executor scheme [VHS15].

Future work

- fully automate and implement the framework in PPCG [VCJC'13].
- conduct further experiments on CPU and GPU platforms, comparing the performance with the CUSP library.
References


