Schedule Trees

Sven Verdoolaege\textsuperscript{1}  Serge Guelton\textsuperscript{2}
Tobias Grosser\textsuperscript{3}  Albert Cohen\textsuperscript{3}

\textsuperscript{1}INRIA, École Normale Supérieure and KU Leuven
\textsuperscript{2}École Normale Supérieure and Télécom Bretagne
\textsuperscript{3}INRIA and École Normale Supérieure

January 20, 2014
Outline

1 Introduction
   - Example
   - Single Statement
   - Multiple Statements
   - Schedule Trees

2 Advantages
   - Useful in several contexts
   - More natural
   - More convenient
   - More expressive
   - Extensible

3 Conclusion
Outline

1 Introduction
   - Example
   - Single Statement
   - Multiple Statements
   - Schedule Trees

2 Advantages
   - Useful in several contexts
   - More natural
   - More convenient
   - More expressive
   - Extensible

3 Conclusion
Introductory Example

for (i = 0; i <= N; ++i)
S: a[i] = g(i);
for (i = 0; i <= N; ++i)
T: b[i] = f(a[N-i]);
Introductory Example

```c
for (i = 0; i <= N; ++i)
S:  a[i] = g(i);
for (i = 0; i <= N; ++i)
T:  b[i] = f(a[N-i]);
```

- **Iteration domain**
  \[
  \{ S[i] : 0 \leq i \leq N; T[i] : 0 \leq i \leq N \}
  \]

- **Dependences**
  \[
  \{ S[i] \rightarrow T[N - i] : 0 \leq i \leq N \}
  \]

- **Execution Order**
  - **Original Order**
    \[
    S[0], S[1], S[2], \ldots, S[N - 1], S[N], T[0], T[1], T[2], \ldots, T[N - 1], T[N]
    \]
Introductory Example

\begin{verbatim}
for (i = 0; i <= N; ++i) 
S:  a[i] = g(i);
for (i = 0; i <= N; ++i) 
T:  b[i] = f(a[N-i]);
\end{verbatim}

- Iteration domain

\[
\{ S[i] : 0 \leq i \leq N ; T[i] : 0 \leq i \leq N \} 
\]

- Dependences

\[
\{ S[i] \rightarrow T[N - i] : 0 \leq i \leq N \} 
\]

- Execution Order
  - Original Order
    \[ S[0], S[1], S[2], \ldots, S[N - 1], S[N], T[0], T[1], T[2], \ldots, T[N - 1], T[N] \]
  - Alternative Order
    \[ S[0], T[N], S[1], T[N - 1], S[2], T[N - 2], \ldots, S[N - 1], T[1], S[N], T[0] \]
Introductory Example

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

- **Iteration domain**

\[ \{ S[i] : 0 \leq i \leq N; T[i] : 0 \leq i \leq N \} \]

- **Dependences**

\[ \{ S[i] \rightarrow T[N - i] : 0 \leq i \leq N \} \]

- **Execution Order**
  - **Original Order**

\[
S[0], S[1], S[2], \ldots, S[N - 1], S[N], T[0], T[1], T[2], \ldots, T[N - 1], T[N]
\]
  - **Alternative Order**

\[
S[0], T[N], S[1], T[N - 1], S[2], T[N - 2], \ldots, S[N - 1], T[1], S[N], T[0]
\]
Expressing Transformations (Single Statement)

\[
\text{for } (i = 0; i <= N; ++i) \Rightarrow \text{for } (i = 0; i <= N; ++i) \\
b[i] = f(a[N-i]); \quad \Rightarrow \quad b[N-i] = f(a[i]);
\]
Expressing Transformations (Single Statement)

\[
\text{for } (i = 0; i <= N; ++i) \quad \Rightarrow \quad \text{for } (i = 0; i <= N; ++i)
\]
\[
b[i] = f(a[N-i]); \quad \Rightarrow \quad b[N-i] = f(a[i]);
\]

Two approaches

1. Modify Iteration Domain

   \[ T[i] \rightarrow T'[N - i] \]

   ▶ iteration domains have implicit execution order (lexicographic order)
   ▶ AST generator takes modified iteration domain as input
   ▶ access relations and dependence relations are adjusted accordingly
Expressing Transformations (Single Statement)

\[
\text{for } (i = 0; i \leq N; ++i) \Rightarrow \text{for } (i = 0; i \leq N; ++i)
\]
\[
b[i] = f(a[N-i]); \quad \Rightarrow \quad b[N-i] = f(a[i]);
\]

Two approaches

1. Modify Iteration Domain

   \[ T[i] \rightarrow T'[N - i] \]
   
   - iteration domains have implicit execution order (lexicographic order)
   - AST generator takes modified iteration domain as input
   - access relations and dependence relations are adjusted accordingly

2. Explicit Schedule

   \[ T[i] \rightarrow [N - i] \]
   
   - iteration domains have no implicit execution order
   - execution order is determined by schedule space (lexicographic order)
   - AST generator takes iteration domain and schedule as input
   - schedule is typically a piecewise quasi-affine function
Expressing Transformations (Single Statement)

\[ \text{for } (i = 0; i \leq N; ++i) \quad \Rightarrow \quad \text{for } (i = 0; i \leq N; ++i) \]
\[ b[i] = f(a[N-i]); \quad \Rightarrow \quad b[N-i] = f(a[i]); \]

Two approaches

1. Modify Iteration Domain

\[ T[i] \rightarrow T'[N - i] \]

- iteration domains have implicit execution order (lexicographic order)
- AST generator takes modified iteration domain as input
- access relations and dependence relations are adjusted accordingly

2. Explicit Schedule

\[ T[i] \rightarrow [N - i] \]

- iteration domains have no implicit execution order
- execution order is determined by schedule space (lexicographic order)
- AST generator takes iteration domain and schedule as input
- schedule is typically a piecewise quasi-affine function
Expressing Transformations (Single Statement)

\[
\begin{align*}
\text{for} & \ (i = 0; i \leq N; ++i) \quad \Rightarrow \quad \text{for} \ (i = 0; i \leq N; ++i) \\
\quad b[i] = f(a[N-i]); & \quad \Rightarrow \quad b[N-i] = f(a[i]);
\end{align*}
\]

Two approaches

1. **Modify Iteration Domain**

   \[ T[i] \rightarrow T'[N - i] \]

   - iteration domains have implicit execution order (lexicographic order)
   - AST generator takes modified iteration domain as input
   - access relations and dependence relations are adjusted accordingly

2. **Explicit Schedule**

   \[ T[i] \rightarrow [N - i] \]

   - iteration domains have no implicit execution order
   - execution order is determined by schedule space (lexicographic order)
   - AST generator takes iteration domain and schedule as input
   - schedule is typically a piecewise quasi-affine function
Representing Schedules for Multiple Statements

\[
\begin{align*}
\text{for } (i = 0; i <= N; ++i) & \quad \text{for } (i = 0; i <= N; ++i) \{ \\
a[i] &= g(i) \\
b[i] &= f(a[N-i]) \\
\} \\
S[i] &\rightarrow [i] ; T[i] \rightarrow [N - i]
\end{align*}
\]

first \(S[i]\) then \(T[i]\)

\[
\begin{align*}
S[i] &\rightarrow [i] \\
T[i] &\rightarrow [i]
\end{align*}
\]

first \(S[i]\) then \(T[i]\)
Representing Schedules for Multiple Statements

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

\[
S[i] \rightarrow [i]; T[i] \rightarrow [N - i]
\]

first \(S[i]\) then \(T[i]\)

\[
\begin{align*}
S[i] & \rightarrow [i] \\
T[i] & \rightarrow [i]
\end{align*}
\]

first \(S[i]\) then \(T[i]\)

\[
\begin{align*}
S : \{ [i] \rightarrow [0, i] \} & \quad S : \{ [i] \rightarrow [i, 0] \} \\
T : \{ [i] \rightarrow [1, i] \} & \quad T : \{ [i] \rightarrow [N - i, 1] \}
\end{align*}
\]

\(\Rightarrow\) \textit{encode} statement ordering in affine function
Representing Schedules for Multiple Statements

for (i = 0; i <= N; ++i)  for (i = 0; i <= N; ++i) {
    a[i] = g(i); 
    a[i] = g(i); 
    b[i] = f(a[N-i]);   b[N-i] = f(a[i]);
} 

S[i] → [i]; T[i] → [N − i]

first S[i]    then T[i]

S[i] → [i]    T[i] → [i]  

first S[i]    then T[i]

S : { [i] → [0, i] }     S : { [i] → [i, 0] } 

Kelly 

T : { [i] → [1, i] }     T : { [i] → [N − i, 1] } 

union map 

{ S[i] → [0, i]; T[i] → [1, i] }   { S[i] → [i, 0]; T[i] → [N − i, 1] } 

⇒ encode statement ordering in affine function
### Representing Schedules for Multiple Statements

```c
for (i = 0; i <= N; ++i)
    a[i] = g(i);
for (i = 0; i <= N; ++i)
    b[i] = f(a[N-i]);
```

- **sequence**
  - `S[i]` → `[i]`
  - `T[i]` → `[i]`

- **Schedule tree**
  - `S[i]` → `[i]`
  - `T[i]` → `[i]`

- **Kelly**
  - `S` : `{ [i] → [0, i] }`
  - `T` : `{ [i] → [1, i] }`

- **Union map**
  - `{ S[i] → [0, i]; T[i] → [1, i] }`

- **Other representations**
  - "2d+1": special case of Kelly’s abstraction
  - "band forest": precursor to schedule trees
Representing Schedules for Multiple Statements

\[
\begin{align*}
\text{for } (i = 0; i \leq N; ++i) & \quad \text{for } (i = 0; i \leq N; ++i) \\
a[i] &= g(i); & a[i] &= g(i); \\
\text{for } (i = 0; i \leq N; ++i) & \quad b[i] = f(a[N-i]); \\
b[i] &= f(a[N-i]); &
\end{align*}
\]

sequence

\[
\begin{align*}
S[i] &\rightarrow [i]; T[i] \rightarrow [N - i] \\
S[i] \rightarrow [i] &\quad T[i] \rightarrow [i] \\
S[i] \rightarrow [0, i] &\quad T[i] \rightarrow [1, i] \\
S : \{ [i] \rightarrow [0, i] \} &\quad S : \{ [i] \rightarrow [i, 0] \} \\
T : \{ [i] \rightarrow [1, i] \} &\quad T : \{ [i] \rightarrow [N - i, 1] \} \\
\{ S[i] \rightarrow [0, i]; T[i] \rightarrow [1, i] \} &\quad \{ S[i] \rightarrow [i, 0]; T[i] \rightarrow [N - i, 1] \}
\end{align*}
\]

Other representations:
- "2d + 1": special case of Kelly’s abstraction
- band forest: precursor to schedule trees
Schedule Trees

\[
\begin{align*}
\text{sequence} \\
\{ S[i] \} & \quad \{ T[i] \} \\
\{ S[i] \rightarrow [i] \} & \quad \{ T[i] \rightarrow [i] \}
\end{align*}
\]

- Core node types
  - Band: multi-dimensional piecewise quasi-affine partial schedule
  - Filter: selects statement instances that are executed by descendants
  - Sequence: children executed in given order
  - Set: children executed in 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
  - Leaf: for easy navigation
Schedule Trees

Core node types
- **Band**: multi-dimensional piecewise quasi-affine partial schedule
- **Filter**: selects statement instances that are executed by descendants
- **Sequence**: children executed in given order
- **Set**: children executed in arbitrary order
Schedule Trees

\[
\begin{align*}
\text{sequence} & \quad \{ S[i] \} & \quad \{ T[i] \} \\
\{ S[i] \rightarrow [i] \} & \quad \{ T[i] \rightarrow [i] \}
\end{align*}
\]

- **Core node types**
  - **Band**: multi-dimensional piecewise quasi-affine partial schedule
  - **Filter**: selects statement instances that are executed by descendants
  - **Sequence**: children executed in given order
  - **Set**: children executed in arbitrary order
Schedule Trees

- **Core node types**
  - **Band**: multi-dimensional piecewise quasi-affine partial schedule
  - **Filter**: selects statement instances that are executed by descendants
  - **Sequence**: children executed in given order
  - **Set**: children executed in arbitrary order

- **Sequence** node

- **Domain** node

- **Context** node

- **Mark** node

- **Leaf** node
Schedule Trees

Core node types
- **Band**: multi-dimensional piecewise quasi-affine partial schedule
- **Filter**: selects statement instances that are executed by descendants
- **Sequence**: children executed in given order
- **Set**: children executed in arbitrary order

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

\[
\{ S[i] : 0 \leq i \leq N; T[i] : 0 \leq i \leq N \}
\]

sequence

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

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

- **Core node types**
  - **Band**: multi-dimensional piecewise quasi-affine partial schedule
  - **Filter**: selects statement instances that are executed by descendants
  - **Sequence**: children executed in given order
  - **Set**: children executed in arbitrary order

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

\[
\{ N \mod 256 = 0 \}
\]

\[
\{ S[i] : 0 \leq i \leq N ; T[i] : 0 \leq i \leq N \}
\]

sequence

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

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

- **Core node types**
  - **Band**: multi-dimensional piecewise quasi-affine partial schedule
  - **Filter**: selects statement instances that are executed by descendants
  - **Sequence**: children executed in given order
  - **Set**: children executed in arbitrary order

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

\[
\begin{align*}
\{ : N \mod 256 = 0 \} \\
\{ S[i] : 0 \leq i \leq N; T[i] : 0 \leq i \leq N \} \\
\text{sequence} \\
\{ S[i] \} & \quad \{ T[i] \} \\
\{ S[i] \rightarrow [i] \} & \quad \{ T[i] \rightarrow [i] \}
\end{align*}
\]

- **Core node types**
  - **Band**: multi-dimensional piecewise quasi-affine partial schedule
  - **Filter**: selects statement instances that are executed by descendants
  - **Sequence**: children executed in given order
  - **Set**: children executed in 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
  - **Leaf**: for easy navigation
Comparison

\[ T_1 : \{ i \} \rightarrow [0, i] \]
\[ T_2 : \{ i, j \} \rightarrow [1, j, 0, i] \]
\[ T_3 : \{ i \} \rightarrow [1, i - 1, 1] \]

\{ S_1[i] \rightarrow [0, i, 0, 0]; S_2[i, j] \rightarrow [1, j, 0, i]; S_3[i] \rightarrow [1, i - 1, 1, 0] \}

- Kelly’s abstraction
  - schedule spread over statements
  - relaxed lexicographic order

- union maps
  - single object
  - strict lexicographic order
  - schedule transformations can be composed

- schedule trees
  - single object
  - relaxed lexicographic order
Outline

1 Introduction
   - Example
   - Single Statement
   - Multiple Statements
   - Schedule Trees

2 Advantages
   - Useful in several contexts
   - More natural
   - More convenient
   - More expressive
   - Extensible

3 Conclusion
Schedule Uses

- Representing the original execution order
  - Input to dependence analysis (in isl)
  - Basis for manual/incremental transformations
- Scheduling
  - Construction based on dependences
  - Schedule modifications
- AST generation
  - Generate AST from schedule
Schedule Uses

- Representing the original execution order
  - Input to dependence analysis (in isl)
  - Basis for manual/incremental transformations

- Scheduling
  - Construction based on dependences
  - Schedule modifications

- AST generation
  - Generate AST from schedule
Schedule Trees Everywhere

Old PPCG:

C code → parse → internal tree → encode → union map

dependences ← dependence analysis ← internal tree ← decode

scheduler → band forest → tile → band forest → encode

AST ← AST generator ← internal tree ← decode ← union map
Schedule Trees Everywhere

Old PPCG:

C code → parse → internal tree → encode → union map

dependencies ← dependence analysis ← internal tree ← decode

scheduler → band forest → tile → band forest → encode

AST ← AST generator ← internal tree ← decode ← union map

New PPCG:

C code → parse → schedule tree → dependence analysis

tile ← schedule tree ← scheduler ← dependences

schedule tree → AST generator → AST
Schedule Construction Example

\begin{verbatim}
for (i = 0; i <= N; ++i)
S:  a[i] = g(i);
for (i = 0; i <= N; ++i)
T:  b[i] = f(a[N-i]);
U:  c = 0;
\end{verbatim}

- **Iteration domain**

\{ S[i] \mid 0 \leq i \leq N; T[i] \mid 0 \leq i \leq N; U[] \}

- **Dependences**

\{ S[i] \rightarrow T[N - i] \mid 0 \leq i \leq N \}
Schedule Construction Example

\begin{align*}
\textbf{for} \quad (i = 0; \ i \leq N; \ ++i) \\
S: \quad a[i] &= g(i) \\
\textbf{for} \quad (i = 0; \ i \leq N; \ ++i) \\
T: \quad b[i] &= f(a[N-i]) \\
U: \quad c &= 0;
\end{align*}

- **Iteration domain**

\begin{align*}
\{ S[i] : 0 \leq i \leq N; \ T[i] : 0 \leq i \leq N; \ U[] \}
\end{align*}

- **Dependences**

\begin{align*}
\{ S[i] \rightarrow T[N-i] : 0 \leq i \leq N \}
\end{align*}
Schedule Construction Example

```plaintext
for (i = 0; i <= N; ++i)
S:   a[i] = g(i);
for (i = 0; i <= N; ++i)
T:   b[i] = f(a[N-i]);
U:   c = 0;

• Iteration domain

{ S[i] : 0 ≤ i ≤ N; T[i] : 0 ≤ i ≤ N; U[] }

• Dependences
```
Schedule Construction Example

```plaintext
for (i = 0; i <= N; ++i)
S:    a[i] = g(i);
for (i = 0; i <= N; ++i)
T:    b[i] = f(a[N-i]);
U:    c = 0;
```

- **Iteration domain**

  \{ S[i] : 0 \leq i \leq N; T[i] : 0 \leq i \leq N; U[] \}

- **Dependences**
Schedule Construction Example

```
for (i = 0; i <= N; ++i)
S: a[i] = g(i);
for (i = 0; i <= N; ++i)
T: b[i] = f(a[N-i]);
U: c = 0;
```

- **Iteration domain**
  \[
  \{ S[i] : 0 \leq i \leq N; T[i] : 0 \leq i \leq N; U[] \} 
  \]

- **Dependences**
  \[
  \{ S[i] \rightarrow T[N - i] : 0 \leq i \leq N \} 
  \]
**Schedule Construction Example**

\[
\text{for } (i = 0; i \leq N; ++i) \\
\text{S: } a[i] = g(i); \\
\text{for } (i = 0; i \leq N; ++i) \\
\text{T: } b[i] = f(a[N-i]); \\
\text{U: } c = 0;
\]

- **Iteration domain**
  \[
  \{ S[i] : 0 \leq i \leq N; T[i] : 0 \leq i \leq N; U[] \}
  \]

- **Dependences**
  \[
  \{ S[i] \rightarrow T[N - i] : 0 \leq i \leq N \}
  \]
Schedule Construction Example

```c
for (i = 0; i <= N; ++i)
S:   a[i] = g(i);
for (i = 0; i <= N; ++i)
T:   b[i] = f(a[N-i]);
U:c = 0;
```

- **Iteration domain**

  \[
  \{ S[i] : 0 \leq i \leq N ; T[i] : 0 \leq i \leq N ; U[] \}
  \]

- **Dependences**

  \[
  \{ S[i] \rightarrow T[N - i] : 0 \leq i \leq N \}
  \]
Schedule Construction Example

\[\text{for } (i = 0; i \leq N; ++i)\]
\[S: \quad a[i] = g(i);\]
\[\text{for } (i = 0; i \leq N; ++i)\]
\[T: \quad b[i] = f(a[N-i]);\]
\[U: c = 0;\]

- Iteration domain

\[\{ S[i] : 0 \leq i \leq N; T[i] : 0 \leq i \leq N; U[] \}\]

- Dependences
Schedule Construction Example

```plaintext
for (i = 0; i <= N; ++i)
S:  a[i] = g(i);
for (i = 0; i <= N; ++i)
T:  b[i] = f(a[N-i]);
U: c = 0;
```

- **Iteration domain**

  \[\{ S[i] : 0 \leq i \leq N; T[i] : 0 \leq i \leq N; U[] \}\]

- **Dependences**
Schedule Construction Example

```c
for (i = 0; i <= N; ++i)
S: a[i] = g(i);
for (i = 0; i <= N; ++i)
T: b[i] = f(a[N-i]);
U:c = 0;
```

- Iteration domain

\{ S[i] : 0 \leq i \leq N ; T[i] : 0 \leq i \leq N ; U[] \}

- Dependences
Schedule Construction Example

\[
\text{for } (i = 0; i <= N; ++i) \\
S: \quad a[i] = g(i); \\
T: \quad b[i] = f(a[N-i]); \\
U: c = 0;
\]

- Iteration domain

\{ S[i] : 0 \leq i \leq N; T[i] : 0 \leq i \leq N; U[] \}

- Dependences

\[ S[i] \rightarrow [i]; T[i] \rightarrow [N - i] \]
Schedule Construction Example

\textbf{for} (i = 0; i \leq N; ++i)
\textbf{S:} \ a[i] = g(i);
\textbf{for} (i = 0; i \leq N; ++i)
\textbf{T:} \ b[i] = f(a[N-i]);
\textbf{U:} \ c = 0;

- Iteration domain

\{ \ S[i]: 0 \leq i \leq N; T[i]: 0 \leq i \leq N; U[] \}

- Dependences

⇒ natural representation of constructed schedule
Local Transformations

Typical scenario:

1. Construct tilable bands (e.g., using Pluto algorithm)
2. Individually tile (some) tilable bands
   - Given a band $D(i) \rightarrow f(i)$, insert a band $D(i) \rightarrow \lfloor f(i)/S \rfloor$
   - First iterate over blocks of size $S$ and then iterate within each block
Local Transformations

Typical scenario:

1. Construct tilable bands (e.g., using Pluto algorithm)
2. Individually tile (some) tilable bands
   - Given a band $D(i) \to f(i)$, insert a band $D(i) \to \lfloor f(i)/S \rfloor$
   - First iterate over blocks of size $S$ and then iterate within each block

Tiled individually:
- bands of different dimensionality
- different tile sizes $S$ per band
Local Transformations

Typical scenario:

1. Construct tilable bands (e.g., using Pluto algorithm)
2. Individually tile (some) tilable bands
   - Given a band $D(i) \rightarrow f(i)$, insert a band $D(i) \rightarrow \lfloor f(i)/S \rfloor$
   - First iterate over blocks of size $S$ and then iterate within each block

Tiled individually:
- bands of different dimensionality
- different tile sizes $S$ per band
Local Transformations

Typical scenario:

1. Construct tilable bands (e.g., using Pluto algorithm)
2. Individually tile (some) tilable bands
   - Given a band $D(i) \rightarrow f(i)$, insert a band $D(i) \rightarrow \lfloor f(i)/S \rfloor$
   - First iterate over blocks of size $S$ and then iterate within each block

Tiled individually:

- bands of different dimensionality
- different tile sizes $S$ per band
Local Transformations

Typical scenario:

1. Construct tilable bands (e.g., using Pluto algorithm)
2. Individually tile (some) tilable bands
   - Given a band $D(i) \rightarrow f(i)$, insert a band $D(i) \rightarrow \lfloor f(i)/S \rfloor$
   - First iterate over blocks of size $S$ and then iterate within each block

Tiled individually:
- bands of different dimensionality
- different tile sizes $S$ per band

```latex
def S_2[i, j, k]
\begin{align*}
S_2[i, j, k] & \rightarrow (k, j) \\
S_2[i, j, k] & \rightarrow (i)
\end{align*}
```

```
set
```

```
def S_1[i, j]; S_3[i, j, k]
\begin{align*}
S_1[i, j] & \rightarrow ([i/s_0], [j/s_1], 0); \\
S_3[i, j, k] & \rightarrow ([i/s_0], [j/s_1], [k/s_2])
\end{align*}
```

```
def S_1[i, j] \rightarrow (i, j, 0);
S_3[i, j, k] \rightarrow (i, j, k)
```
Local Transformations

Schedule Tree:

- $S_2[i, j, k] \rightarrow (k, j)$
- $S_2[i, j, k] \rightarrow (i)$
- $S_1[i, j]; S_3[i, j, k] \rightarrow (i, j, k)$

Kelly's abstraction:

- $T_1: \{ [i, j] \rightarrow [1, i, j, 0] \}$
- $T_2: \{ [i, j, k] \rightarrow [0, k, j, i] \}$
- $T_3: \{ [i, j, k] \rightarrow [1, i, j, k] \}$
Local Transformations

Schedule Tree:

```
S_2[i, j, k] → (k, j)
```

```
S_2[i, j, k] → (i)
```

```
S_1[i, j]; S_3[i, j, k]
```

```
S_1[i, j] → (i, j, 0); S_3[i, j, k] → (i, j, k)
```

Kelly’s abstraction:

```
T_1 : { [i, j] → [1, i, j, 0] }
```

```
T_2 : { [i, j, k] → [0, k, j, i] }
```

```
T_3 : { [i, j, k] → [1, i, j, k] }
```

How to identify node that needs to be tiled?

- interval of dimensions
- list of statements or values for set/sequence encodings
Local Transformations

Schedule Tree:

\[ S_2[i, j, k] \rightarrow (k, j) \]

\[ S_2[i, j, k] \rightarrow (i) \]

\[ S_1[i, j]; S_3[i, j, k] \]

Kelly’s abstraction:

\[ T_1 : \{ [i, j] \rightarrow [1, i, j, 0] \} \]

\[ T_2 : \{ [i, j, k] \rightarrow [0, k, j, i] \} \]

\[ T_3 : \{ [i, j, k] \rightarrow [1, i, j, k] \} \]

How to identify node that needs to be tiled?

- interval of dimensions
- list of statements or values for set/sequence encodings

Union map representation additionally requires alignment of single schedule space
CARP Project

Design tools and techniques to aid
Correct and Efficient Accelerator Programming

CARP Approach

Domain Specific Languages
DSL -> PENCIL compilers
Performance metadata

PENCIL – Platform Neutral Compute Intermediate Language

Polyhedral compilation

OpenCL

Direct OpenCL programming

NVIDIA GPUs

AMD GPUs

ARM GPUs

…

Other accelerators

Formal verification

Widely supported industry standard

Auto tuning
Advanced Use: CUDA/OpenCL Code Generation

- Schedule tree logically split into two parts
  - Outer part mapped to host code
  - Subtrees mapped to device code
- Device part has additional symbolic constants
  - block and thread identifiers
  - internal context nodes
- Each thread executes only part of iteration domain
  - selected using filter nodes
Advanced Use: CUDA/OpenCL Code Generation

- Schedule tree logically split into two parts
  - Outer part mapped to host code
  - Subtrees mapped to device code
- Device part has additional symbolic constants
  - block and thread identifiers
  - internal context nodes
- Each thread executes only part of iteration domain
  - selected using filter nodes

Old PPCG used nested AST generation
- difficult to understand and debug
Advanced Use: CUDA/OpenCL Code Generation

```c
for (t = 0; t < T; t++) {
    for (i = 1; i < N - 1; i++)
    for (j = 1; j < N - 1; j++)
        A[j] = B[j];
}
```

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

\[
S[t, i] \rightarrow [0]; t[t, j] \rightarrow [1]
\]

\[
\text{set}
\]

- **T[t, j]**
  - **mark:** kernel
  - \(0 \leq b < 32768 \land 0 \leq t < 32\)
  - **T[t, j] :** \(b = \lfloor j/32 \rfloor \mod 32768\)
  - **T[t, j] → \lfloor j/32 \rfloor\)
  - **T[t, j] : t = j \mod 32\)
  - **T[t, j] → j \mod 32\)

- **S[t, i]**
  - **mark:** kernel
  - \(0 \leq b < 32768 \land 0 \leq t < 32\)
  - **S[t, i] :** \(b = \lfloor i/32 \rfloor \mod 32768\)
  - **S[t, i] → \lfloor i/32 \rfloor\)
  - **S[t, i] : t = i \mod 32\)
  - **S[t, i] → i \mod 32\)
Advanced Use: CUDA/OpenCL Code Generation

```c
for (t = 0; t < T; t++) {
    for (i = 1; i < N - 1; i++)
    for (j = 1; j < N - 1; j++)
        A[j] = B[j];
}
```

```
S[t, i] → [t]; t[t, j] → [t]
S[t, i] → [0]; t[t, j] → [1]
```

subtree mapped to device

```
T[t, j] : b = ⌊j/32⌋ mod 32768
T[t, j] → ⌊j/32⌋
T[t, j] : t = j mod 32
T[t, j] → j mod 32
```

```
mark: kernel
```

```
0 ≤ b < 32768 ∧ 0 ≤ t < 32
```

```
S[t, i] : b = ⌊i/32⌋ mod 32768
S[t, i] → ⌊i/32⌋
S[t, i] : t = i mod 32
S[t, i] → i mod 32
```

```
mark: kernel
```

```
0 ≤ b < 32768 ∧ 0 ≤ t < 32
```
Advanced Use: CUDA/OpenCL Code Generation

```c
for (t = 0; t < T; t++) {
    for (i = 1; i < N - 1; i++)
    for (j = 1; j < N - 1; j++)
        A[j] = B[j];
}
```

introduce identifiers

- `S[t, i]` → `[t]`
- `t[t, j]` → `[t]`
- `S[t, i]` → `[0]`
- `t[t, j]` → `[1]`

set

```
mark: kernel
```

```
0 ≤ b < 32768 ∧ 0 ≤ t < 32
```

```
T[t, j] : b = [j/32] mod 32768
```

```
T[t, j] : t = j mod 32
```

```
S[t, i] : b = [i/32] mod 32768
```

```
S[t, i] : t = i mod 32
```

```
S[t, i] → [i/32]
```

```
S[t, i] → i mod 32
```
```c
for (t = 0; t < T; t++) {
    for (i = 1; i < N - 1; i++)
    for (j = 1; j < N - 1; j++)
        A[j] = B[j];
}
```

S[t, i] → [t]; t[t, j] → [t]
S[t, i] → [0]; t[t, j] → [1]
set

filter on identifiers

```
```
Extension

In final stages of scheduling, additional statements may need to be added

- Copy code
- Synchronization
- ...

These additional statements depend on ancestors

- the statements should only be executed in a given part of the schedule tree
- iteration domains depend on outer schedule (e.g., data to be copied)

⇒ new “extension” node type
⇒ maps outer schedule dimensions to extra iteration domain
Extension

\[ 0 \leq b_0, b_1 < 128 \land 0 \leq t_0 < 32 \land 0 \leq t_1 < 16 \]

\[ S_0[i, j] : b_0 = \lfloor i/32 \rfloor \mod 128 \land b_1 = \lfloor j/32 \rfloor \mod 128; \]

\[ S_1[i, j, k] : b_0 = \lfloor i/32 \rfloor \mod 128 \land b_1 = \lfloor j/32 \rfloor \mod 128 \]

\[ \text{sequence} \]

\[ \text{write}_C[u, v] \]

\[ S_0[i, j]; S_1[i, j, k] \]

\[ S_0[i, j] \rightarrow \lfloor [i/32], [j/32] \rfloor; \]

\[ S_1[i, j, k] \rightarrow \lfloor [i/32], [j/32] \rfloor \]

\[ S_0[i, j] \rightarrow [0]; S_1[i, j, k] \rightarrow \lfloor [k/32] \rfloor \]

\[ [i_0, i_1, i_2] \rightarrow \text{sync}[]; \]

\[ [i_0, i_1, i_2] \rightarrow \text{read}_A[u, v] : \]

\[ 0 \leq u, v \leq 4095 \land b_0 = \lfloor u/32 \rfloor \land i_2 = \lfloor v/32 \rfloor; \]

\[ [i_0, i_1, i_2] \rightarrow \text{read}_B[u, v] : \ldots \]
Extension

\[ 0 \leq b_0, b_1 < 128 \land 0 \leq t_0 < 32 \land 0 \leq t_1 < 16 \]

\[ S_0[i, j] : b_0 = \lfloor i/32 \rfloor \mod 128 \land b_1 = \lfloor j/32 \rfloor \mod 128; \]
\[ S_1[i, j, k] : b_0 = \lfloor i/32 \rfloor \mod 128 \land b_1 = \lfloor j/32 \rfloor \mod 128 \]

\[ [] \rightarrow \text{write}_C[u, v] : 0 \leq u, v \leq 4095 \land b_0 = \lfloor u/32 \rfloor \land b_1 = \lfloor v/32 \rfloor \]

\[ S_0[i, j]; S_1[i, j, k] \]
\[ S_0[i, j] \rightarrow [\lfloor i/32 \rfloor, \lfloor j/32 \rfloor]; \]
\[ S_1[i, j, k] \rightarrow [\lfloor i/32 \rfloor, \lfloor j/32 \rfloor] \]
\[ S_0[i, j] \rightarrow [0]; S_1[i, j, k] \rightarrow [\lfloor k/32 \rfloor] \]

\[ [i_0, i_1, i_2] \rightarrow \text{sync}[]; \]
\[ [i_0, i_1, i_2] \rightarrow \text{read}_A[u, v] : \]
\[ 0 \leq u, v \leq 4095 \land b_0 = \lfloor u/32 \rfloor \land i_2 = \lfloor v/32 \rfloor; \]
\[ [i_0, i_1, i_2] \rightarrow \text{read}_B[u, v] : \ldots \]
Extension

\[
0 \leq b_0, b_1 < 128 \land 0 \leq t_0 < 32 \land 0 \leq t_1 < 16
\]

\[
S_0[i, j]: b_0 = \lfloor i/32 \rfloor \mod 128 \land b_1 = \lfloor j/32 \rfloor \mod 128;
\]
\[
S_1[i, j, k]: b_0 = \lfloor i/32 \rfloor \mod 128 \land b_1 = \lfloor j/32 \rfloor \mod 128
\]

\[
[] \to \text{write}_C[u, v]: 0 \leq u, v \leq 4095 \land b_0 = \lfloor u/32 \rfloor \land b_1 = \lfloor v/32 \rfloor
\]

sequence

\[
S_0[i, j]; S_1[i, j, k]
\]
\[
S_0[i, j] \to [\lfloor i/32 \rfloor, \lfloor j/32 \rfloor];
\]
\[
S_1[i, j, k] \to [\lfloor i/32 \rfloor, \lfloor j/32 \rfloor]
\]
\[
S_0[i, j] \to [0]; S_1[i, j, k] \to [\lfloor k/32 \rfloor]
\]
\[
[i_0, i_1, i_2] \to \text{sync}[];
\]
\[
[i_0, i_1, i_2] \to \text{read}_A[u, v]:
\]
\[
0 \leq u, v \leq 4095 \land b_0 = \lfloor u/32 \rfloor \land i_2 = \lfloor v/32 \rfloor;
\]
\[
[i_0, i_1, i_2] \to \text{read}_B[u, v]: \ldots
\]
Outline

1 Introduction
   • Example
   • Single Statement
   • Multiple Statements
   • Schedule Trees

2 Advantages
   • Useful in several contexts
   • More natural
   • More convenient
   • More expressive
   • Extensible

3 Conclusion
Conclusion

Conclusion:

Exploit the tree nature of a schedule rather than encoding it in a flat representation

Schedule trees are

- useful in several contexts
- more natural
- more convenient
- more expressive
- extensible
Conclusion:

Exploit the tree nature of a schedule rather than encoding it in a flat representation

Schedule trees are
- useful in several contexts
- more natural
- more convenient
- more expressive
- extensible

Future work
- apply separation on schedule tree
- additional node types
  - parametric tiling
  - clustering
  - ...