Parametric Tiling with Inter-Tile Data Reuse

Alexandre Isoard  Alain Darte

Compsys, LIP (Laboratoire de l’Informatique du Parallélisme), Lyon

IMPACT
4th International Workshop on
Polyhedral Compilation Techniques
January 20, 2014
Vienna, Austria
Outline

1 Motivation and challenges
   - Kernel offloading: rules of the game
   - Reminders: scheduling and tiling
   - Inter-tile data reuse: example

2 Parametric analysis
   - Tile index vs tile origin index
   - Exact inter-tile reuse
   - Approximated inter-tile reuse

3 Current implementation and results
   - Current status
   - Script with ISCC
   - Local memory allocation for PolyBench examples
Perform computations by blocks;
Exploit data reuse;
Use pipelining/prefetching;
Reduce and coalesce communications (burst).
Data reuse: on the full iteration domain

Rule 1: always use local data if already loaded or computed.
- Reduces communication volume, increases local memory.
- Enables full pipelining (load/compute/store sequence).
Rules and objectives

Data reuse: on the full iteration domain

Rule 1: always use local data if already loaded or computed.
- Reduces communication volume, increases local memory.
- Enables full pipelining (load/compute/store sequence).

Blocking: thanks to tiling

Rule 2: tiles executed in sequence (but a tile can be parallelized).
- Increases temporal reuse, reduces local memory.
- Increases spatial reuse, enables burst communications.
Rules and objectives

Data reuse: on the full iteration domain

Rule 1: always use local data if already loaded or computed.
- Reduces communication volume, increases local memory.
- Enables full pipelining (load/compute/store sequence).

Blocking: thanks to tiling

Rule 2: tiles executed in sequence (but a tile can be parallelized).
- Increases temporal reuse, reduces local memory.
- Increases spatial reuse, enables burst communications.

Variants for reuse domain, i.e., where data reuse is performed
- Iteration domain reduced thanks to hierarchical tiling.
- Data reuse in a $p$-dimensional stripe, or at bounded distance.
Rules and objectives

Data reuse: on the full iteration domain

Rule 1: always use local data if already loaded or computed.
- Reduces communication volume, increases local memory.
- Enables full pipelining (load/compute/store sequence).

Blocking: thanks to tiling

Rule 2: tiles executed in sequence (but a tile can be parallelized).
- Increases temporal reuse, reduces local memory.
- Increases spatial reuse, enables burst communications.

Variants for reuse domain, i.e., where data reuse is performed
- Iteration domain reduced thanks to hierarchical tiling.
- Data reuse in a $p$-dimensional stripe, or at bounded distance.

Then: scheduling/pipelining & memory allocation

Rule 3: reuse analysis independently on scheduling.

Rule 4: load as late as possible, store as soon as possible.
- Overlaps transfer and computation (multi-buffering).
- Reduces live-ranges, and possibly local memory size.
Rules and objectives

Parametric in terms of tile sizes?

Data reuse: on the full iteration domain

Rule 1: always use local data if already loaded or computed.

- Reduces communication volume, increases local memory.
- Enables full pipelining (load/compute/store sequence).

Blocking: thanks to tiling

Rule 2: tiles executed in sequence (but a tile can be parallelized).

- Increases temporal reuse, reduces local memory.
- Increases spatial reuse, enables burst communications.

Variants for reuse domain, i.e., where data reuse is performed

- Iteration domain reduced thanks to hierarchical tiling.
- Data reuse in a $p$-dimensional stripe, or at bounded distance.

Then: scheduling/pipelining & memory allocation

Rule 3: reuse analysis independently on scheduling.

Rule 4: load as late as possible, store as soon as possible.

- Overlaps transfer and computation (multi-buffering).
- Reduces live-ranges, and possibly local memory size.
Challenges and contributions

**General principle for Load sets**

Load a data indexed by $\vec{m}$ just before a tile indexed by $\vec{T}$ if:

- $\vec{m}$ is live-in for $\vec{T}$, i.e., read but not written earlier in $\vec{T}$.
- $\vec{m}$ has not been loaded in a previous tile.
- $\vec{m}$ has not been defined earlier.
Challenges and contributions

General principle for Load sets

Load a data indexed by $\vec{m}$ just before a tile indexed by $\vec{T}$ if:
- $\vec{m}$ is live-in for $\vec{T}$, i.e., read but not written earlier in $\vec{T}$.
- $\vec{m}$ has not been loaded in a previous tile.
- $\vec{m}$ has not been defined earlier.

Tiling defines a schedule on tile+iteration indices, thus “previous” and “earlier”. This schedule is not affine in terms of tile sizes.
Challenges and contributions

General principle for Load sets
Load a data indexed by $\vec{m}$ just before a tile indexed by $\vec{T}$ if:
- $\vec{m}$ is live-in for $\vec{T}$, i.e., read but not written earlier in $\vec{T}$.
- $\vec{m}$ has not been loaded in a previous tile.
- $\vec{m}$ has not been defined earlier.

Tiling defines a schedule on tile+iteration indices, thus “previous” and “earlier”. This schedule is not affine in terms of tile sizes.

Exact case
Reads/writes are functions of iteration points. Can we express the relation “happens before” among iterations in a quasi-affine way?
- Yes. Parametric tiling with exact inter-tile reuse is feasible.
Challenges and contributions

General principle for Load sets
Load a data indexed by \( \vec{m} \) just before a tile indexed by \( \vec{T} \) if:
- \( \vec{m} \) is live-in for \( \vec{T} \), i.e., read but not written earlier in \( \vec{T} \).
- \( \vec{m} \) has not been loaded in a previous tile.
- \( \vec{m} \) has not been defined earlier.

Tiling defines a schedule on tile+iteration indices, thus “previous” and “earlier”. This schedule is not affine in terms of tile sizes.

Exact case
Reads/writes are functions of iteration points. Can we express the relation “happens before” among iterations in a quasi-affine way?
- Yes. Parametric tiling with exact inter-tile reuse is feasible.

Approximations
What if contributions of reads/writes are summarized at tile level? Approximated?
- No information loss if approximations are “pointwise”. More approximations needed otherwise.
Product of two polynomials:
- arguments in $A$ and $B$;
- result in $C$.

```c
for (int k = 0; k < 2*n - 1; k++) {
    C[k] = 0; // S0
}

for (int i = 0; i < n; i++) {
    for (int j = 0; j < n; j++) {
        C[i+j] += A[i]*B[j]; // S1
    }
}
```
Product of two polynomials:
- arguments in $A$ and $B$;
- result in $C$.

```c
for (int k = 0; k < 2*n - 1; k++) {
    C[k] = 0; // S0
}

for (int i = 0; i < n; i++) {
    for (int j = 0; j < n; j++) {
        C[i+j] += A[i] * B[j]; // S1
    }
}
```
Reads, writes, schedule

Product of two polynomials:
- arguments in $A$ and $B$;
- result in $C$.

```c
for (int k = 0; k < 2*n - 1; k++) {
    C[k] = 0; // S0
}
for (int i = 0; i < n; i++) {
    for (int j = 0; j < n; j++) {
        C[i+j] += A[i]*B[j]; // S1
    }
}
```
Motivation and challenges
Parametric analysis
Current implementation and results

Kernel offloading: rules of the game
Reminders: scheduling and tiling
Inter-tile data reuse: example

Reads, writes, schedule

Product of two polynomials:
- arguments in $A$ and $B$;
- result in $C$.

```c
for (int k = 0; k < 2*n - 1; k++) {
    C[k] = 0;  // S0
}
for (int i = 0; i < n; i++) {
    for (int j = 0; j < n; j++) {
        C[i+j] += A[i] * B[j];  // S1
    }
}
```

+ possibility of intra-tile parallelism.
Motivation and challenges
Parametric analysis
Current implementation and results

Kernel offloading: rules of the game
Reminders: scheduling and tiling
Inter-tile data reuse: example

Reads, writes, schedule

Product of two polynomials:
- arguments in A and B;
- result in C.

```c
for (int k = 0; k < 2*n - 1; k++) {
    C[k] = 0; // S0
}
for (int i = 0; i < n; i++) {
    for (int j = 0; j < n; j++) {
        C[i+j] += A[i] * B[j]; // S1
    }
}
```
Product of two polynomials:

- arguments in $A$ and $B$;
- result in $C$.

```c
for (int k = 0; k < 2*n - 1; k++) {
    C[k] = 0; // S0
}
for (int i = 0; i < n; i++) {
    for (int j = 0; j < n; j++) {
        C[i+j] += A[i]*B[j]; // S1
    }
}
```
Motivation and challenges
Parametric analysis
Current implementation and results

Dependences

Product of two polynomials:
- arguments in $A$ and $B$;
- result in $C$.

```c
for(int k=0; k<2*n-1; k++) {
    C[k] = 0; // S0
}
for(int i=0; i<n; i++) {
    for(int j=0; j<n; j++) {
        C[i+j] += A[i]*B[j]; // S1
    }
}
```
Scheduling alternatives: loop reversal + interchange

Product of two polynomials:
- arguments in \( A \) and \( B \);
- result in \( C \).

```c
for (int k = 0; k < 2*n - 1; k++) {
    C[k] = 0; // S0
}
for (int i = 0; i < n; i++) {
    for (int j = 0; j < n; j++) {
        C[i+j] += A[i]*B[j]; // S1
    }
}
```
Product of two polynomials:
- arguments in $A$ and $B$;
- result in $C$.

```c
for (int k = 0; k < 2*n - 1; k++) {
    C[k] = 0; // S0
}
for (int i = 0; i < n; i++) {
    for (int j = 0; j < n; j++) {
        C[i+j] += A[i]*B[j]; // S1
    }
}
```
Product of two polynomials:
- arguments in $A$ and $B$;
- result in $C$.

```c
for (int k = 0; k < 2*n - 1; k++) {
    C[k] = 0; // S0
}
for (int i = 0; i < n; i++) {
    for (int j = 0; j < n; j++) {
        C[i+j] += A[i]*B[j]; // S1
    }
}
```
Product of two polynomials:

- arguments in A and B;
- result in C.

```c
for(int k=0; k<2*n-1; k++)
  C[k] = 0; // S0

for(int i=0; i<n; i++)
  for(int j=0; j<n; j++)
    C[i+j] += A[i]*B[j]; // S1
```

+ possibility of intra-tile parallelism.
Product of two polynomials:
- arguments in $A$ and $B$;
- result in $C$.

```c
for(int k=0; k<2*n-1; k++) {
    C[k] = 0; // S0
}
for(int i=0; i<n; i++) {
    for(int j=0; j<n; j++) {
        C[i+j] += A[i]*B[j]; // S1
    }
}
```

+ possibility of intra-tile parallelism.
Inter-tile data reuse in a tile strip

\[
\begin{align*}
\text{for}(i=0; \ i<n; \ i++) & \\
& \text{for}(j=0; \ j<n; \ j++) \\
& C[i+j] = C[i+j] + A[i]\times B[j];
\end{align*}
\]

\[
(i,j) \mapsto (n - j - 1, i)
\]

\[
(i,j) \mapsto (i + j, i)
\]

In a tile, \textbf{Load} \simeq \text{first read}, \textbf{Store} \simeq \text{last write}.
for(i=0; i<n; i++)
    for(j=0; j<n; j++)
        C[i+j] = C[i+j] + A[i]*B[j];

\[(i, j) \mapsto (n-j-1, i)\]

\[(i, j) \mapsto (i+j, i)\]

In a tile strip, \textbf{Load} \simeq \text{first read}, \textbf{Store} \simeq \text{last write}. 
Inter-tile data reuse in a tile strip

\[
\text{for}(i=0; \ i<n; \ i++) \\
\text{for}(j=0; \ j<n; \ j++) \\
C[i+j] = C[i+j] + A[i]*B[j];
\]

In a \textbf{reuse domain}, \textbf{Load} \thicksim \text{first read}, \textbf{Store} \thicksim \text{last write}.
Can actually be adapted to any parameterized reuse domain.
Objective: data transfers

- Bound $n$, tiles of size $b \times b$.
- Tiling with $(i, j) \mapsto (i', j') = (n - j - 1, i)$.
- Access functions $m = i + j = j' + n - i' - 1$.
- Tile origin $(I, J)$.
- Transfers $\text{Load}_A$, $\text{Load}_B$, $\text{Load}_C$, $\text{Store}_C$. 
Objective: data transfers

- Bound $n$, tiles of size $b \times b$.
- Tiling with $(i, j) \mapsto (i', j') = (n - j - 1, i)$.
- Access functions $m = i + j = j' + n - i' - 1$.
- Tile origin $(I, J)$.
- Transfers $\text{Load}_A$, $\text{Load}_B$, $\text{Load}_C$, $\text{Store}_C$.

Load sets.

$\text{Load}_A = \{m \mid 0 \leq m \leq n - 1, J \leq m \leq J + b - 1\}$
Objective: data transfers

- Bound $n$, tiles of size $b \times b$.
- Tiling with $(i, j) \mapsto (i', j') = (n - j - 1, i)$.
- Access functions $m = i + j = j' + n - i' - 1$.
- Tile origin $(I, J)$.
- Transfers $\text{Load}_A$, $\text{Load}_B$, $\text{Load}_C$, $\text{Store}_C$.

Load sets.

$\text{Load}_A = \{m \mid 0 \leq m \leq n - 1, J \leq m \leq J + b - 1\}$

$\text{Load}_B = \{m \mid J = 0, 0 \leq m \leq n - 1, n - l - b \leq m \leq n - l - 1\}$
Objective: data transfers

- Bound $n$, tiles of size $b \times b$.
- Tiling with $(i, j) \mapsto (i', j') = (n - j - 1, i)$.
- Access functions $m = i + j = j' + n - i' - 1$.
- Tile origin $(I, J)$.
- Transfers $\text{Load}_A, \text{Load}_B, \text{Load}_C, \text{Store}_C$.

Load sets.

$\text{Load}_A = \{m | 0 \leq m \leq n - 1, J \leq m \leq J + b - 1\}$

$\text{Load}_B = \{m | J = 0, 0 \leq m \leq n - 1, n - l - b \leq m \leq n - l - 1\}$

$\text{Load}_C = \{m | 0 \leq m, n - l - b \leq m \leq n - 1 - l, J = 0\}$

$\cup \{m | \max(1, J) \leq m + l - n + 1 \leq \min(n - 1, J + b - 1)\}$
Objective: data transfers and local memory sizes

- Bound \( n \), tiles of size \( b \times b \).
- Tiling with \((i, j) \mapsto (i', j') = (n - j - 1, i)\).
- Access functions \( m = i + j = j' + n - i' - 1 \).
- Tile origin \((I, J)\).
- Transfers \( \text{Load}_A, \text{Load}_B, \text{Load}_C, \text{Store}_C \).

Load sets. Local memory sizes with “double-buffering”.

\[
\text{Load}_A = \{ m \mid 0 \leq m \leq n - 1, \; J \leq m \leq J + b - 1 \}
\]
- size \( 2b \), when \( n \geq 2b + 1 \): at least 2 tiles available.
- size \( n \) when \( n \leq 2b \): less than 2 tiles.

\[
\text{Load}_B = \{ m \mid J = 0, \; 0 \leq m \leq n - 1, \; n - l - b \leq m \leq n - l - 1 \}
\]
- size \( b \) when \( n \geq b \): 1 full tile.
- size \( n \) when \( n \leq b - 1 \): 1 partial tile.

\[
\text{Load}_C = \{ m \mid 0 \leq m, \; n - l - b \leq m \leq n - 1 - l, \; J = 0 \}
\cup \{ m \mid \max(1, J) \leq m + l - n + 1 \leq \min(n - 1, J + b - 1) \}
\]
- size \( 3b - 1 = (2b - 1) + b \) si \( n \geq 2b + 1 \): 2 full tiles.
- size \( b + n - 1 = (2b - 1) + (n - b) \) si \( b \leq n \leq 2b \): 1 full tile, 1 partial tile.
- size \( 2n - 1 \) si \( n \leq b - 1 \): 1 partial tile.
Outline

1. Motivation and challenges

2. Parametric analysis
   - Tile index vs tile origin index
   - Exact inter-tile reuse
   - Approximated inter-tile reuse

3. Current implementation and results
Tiling, tiles, and schedules

With indices of tiles (tile sizes defined by $\hat{s} = (s_1, \ldots, s_n)$)

$$\vec{i} \in \text{Tile}(\vec{T}) \iff \begin{cases} s_1 T_1 \leq i_1 < s_1(T_1 + 1) \\ \vdots \\ s_n T_n \leq i_n < s_n(T_n + 1) \end{cases}$$

Schedule on iteration points: $\vec{i}^{\ddagger} < \vec{i} \iff (\vec{T}', \vec{i}^{\ddagger}) <_{\text{lex}} (\vec{T}, \vec{i})$. 
Tiling, tiles, and schedules

With indices of tiles (tile sizes defined by $\vec{s} = (s_1, \ldots, s_n)$)

$$\vec{i} \in \text{Tile}(\vec{T}) \iff \begin{cases} s_1 T_1 \leq i_1 < s_1 (T_1 + 1) \\ \\ \\ s_n T_n \leq i_n < s_n (T_n + 1) \end{cases}$$

◊ Schedule on iteration points: $\vec{i}' < \vec{i} \iff (\vec{T}', \vec{i}') <_{\text{lex}} (\vec{T}, \vec{i})$.

With indices of tile origins

$$\vec{i} \in \text{Tile}(\vec{I}) \iff \begin{cases} l_1 \leq i_1 < l_1 + s_1 \\ \\ \\ l_n \leq i_n < l_n + s_n \end{cases}$$

with $\vec{I}$, origin of $\text{Tile}(\vec{T})$, i.e., $\vec{I} = (s_1 T_1, \ldots, s_n T_n)$.

◊ Schedule on iteration points, for a tiling specified by a given tile:

$$\vec{i}' <_{\vec{I}} \vec{i} \iff \vec{i}' <_{\vec{I}} \vec{i} \iff (\vec{l}', \vec{i}') <_{\text{lex}} (\vec{l}, \vec{i})$$ and $\vec{l}' \equiv \vec{l}$.
Intuitive expression of Load/Store sets

For $\text{Tile}(\vec{i})$ with data reuse in ReuseDomain:

$$\text{Load}(\vec{i}) = \bigcup_{\vec{i} \in \text{Tile}(\vec{i})} \left( \text{read}(\vec{i}) \setminus \bigcup_{\vec{i}' < \vec{i}} \text{read}(\vec{i}') \cup \text{write}(\vec{i}') \right)$$

$$\text{Store}(\vec{i}) = \bigcup_{\vec{i} \in \text{Tile}(\vec{i})} \left( \text{write}(\vec{i}) \setminus \bigcup_{\vec{i}' > \vec{i}} \text{write}(\vec{i}') \right)$$

where $\vec{i}' < \vec{i}$ means that $i'$ is executed before $i$ in the tiled schedule.
Intuitive expression of Load/Store sets

For $\text{Tile}(\vec{i})$ with data reuse in $\text{ReuseDomain}$:

\[
\text{Load}(\vec{i}) = \bigcup_{\vec{i} \in \text{Tile}(\vec{i})} \left( \text{read}(\vec{i}) \setminus \bigcup_{\vec{i}' < \vec{i}} \text{read}(\vec{i}') \cup \text{write}(\vec{i}') \right)
\]

\[
\text{Store}(\vec{i}) = \bigcup_{\vec{i} \in \text{Tile}(\vec{i})} \left( \text{write}(\vec{i}) \setminus \bigcup_{\vec{i}' > \vec{i}} \text{write}(\vec{i}') \right)
\]

where $\vec{i}' < \vec{i}$ means that $i'$ is executed before $i$ in the tiled schedule.

Can we express $\vec{i}' < \vec{i}$ ("happens before") in a parametric way?
Tiling, relation “happens before” and unaligned tiles

$\vec{i}' < \vec{i}$ iff

1. $\vec{i} \in \text{Tile}(\vec{T})$ and $\vec{i}' \in \text{Tile}(\vec{T}')$
2. $(\vec{T}', \vec{i}') <_{\text{lex}} (\vec{T}, \vec{i})$
Tiling, relation “happens before” and unaligned tiles

\[ \vec{i'} \prec \vec{i} \text{ iff } \]

- \( \vec{i} \in \text{Tile}(\vec{I}) \) and \( \vec{i'} \in \text{Tile}(\vec{I'}) \)
- \( (\vec{I'}, \vec{i'}) \prec_{\text{lex}} (\vec{I}, \vec{i}) \) and \( \vec{I'} \equiv_{\text{s}} \vec{I} \)
Tiling, relation “happens before” and unaligned tiles

\[ \vec{i}' < \vec{i} \text{ iff } \]

- \( \vec{i} \in \text{Tile}(\vec{i}) \) and \( \vec{i}' \in \text{Tile}(\vec{i}') \)
- \( \vec{i}' = \vec{i} \land \vec{i}' <_{\text{lex}} \vec{i} \)

or

\( \vec{i}' <_{\text{lex}} \vec{i} \land \vec{i}' \equiv \overline{\vec{i}} \Leftrightarrow \vec{i} \sqsubseteq \overline{\vec{i}} \vec{i}' \)
Tiling, relation “happens before” and unaligned tiles

$\vec{i}' \prec \vec{i}$ iff

- $\vec{i} \in \text{Tile}(\vec{i})$ and $\vec{i}' \in \text{Tile}(\vec{i}')$
- $\vec{i}' = \vec{i} \land \vec{i}' \prec_{\text{lex}} \vec{i}$
  or
  $\vec{i}' \prec_{\text{lex}} \vec{i} \land \vec{i}' \equiv \vec{i} \Leftrightarrow \vec{i} \sqsubset_{\vec{s}} \vec{i}'$
Tiling, relation “happens before” and unaligned tiles

\[ \vec{i}’ < \vec{i} \text{ iff } \]

- \( \vec{i} \in \text{Tile}(\vec{i}) \) and \( \vec{i}’ \in \text{Tile}(\vec{i}’) \)
- \( \vec{i}’ = \vec{i} \land \vec{i}’ <_{lex} \vec{i} \)

or

\( (i_1’ < I_1) \lor (i_1’ < I_1 + s_1 \land i_2’ < I_2) \)
Tiling, relation “happens before” and unaligned tiles

\[ \vec{i}' < \vec{i} \text{ iff } \]

- \( \vec{i} \in \text{Tile}(\vec{i}) \) and \( \vec{i}' \in \text{Tile}(\vec{i}') \)
- \( \vec{i}' = \vec{i} \land \vec{i}' <_{\text{lex}} \vec{i} \)

or

\( (I_1' \leq I_1 - s_1) \lor (I_1' \leq I_1 \land I_2' \leq I_2 - s_2) \)
Tiling, relation “happens before” and unaligned tiles

\[
\vec{i}' < \vec{i} \text{ iff }
\]

- \(\vec{i} \in \text{Tile}(\vec{i})\) and \(\vec{i}' \in \text{Tile}(\vec{i}')\)
- \(\vec{i}' = \vec{i} \land \vec{i}' <_{\text{lex}} \vec{i}\)

or

\(\vec{i}' <_{\prec} \vec{i}\): partial order on tiles
(aligned and unaligned tiles)
Load/Store computations with In/Out sets

Contribution of reads/writes summarized at tile level:

\[
\begin{align*}
\text{In}(\vec{i}) &= \bigcup_{\vec{i} \in \text{Tile}(\vec{i})} \left( \text{read}(\vec{i}) \setminus \bigcup_{\vec{i}' \in \text{Tile}(\vec{i}), \vec{i}' <_{\text{lex}} \vec{i}} \text{write}(\vec{i}') \right) \\
\text{Out}(\vec{i}) &= \bigcup_{\vec{i} \in \text{Tile}(\vec{i})} \text{write}(\vec{i})
\end{align*}
\]
Load/Store computations with In/Out sets

Contribution of reads/writes summarized at tile level:

\[
\begin{align*}
\text{In}(\vec{i}) &= \bigcup_{\vec{i} \in \text{Tile}(\vec{i})} \left( \text{read}(\vec{i}) \setminus \bigcup_{\vec{i}' \in \text{Tile}(\vec{i}), \vec{i}' <_{\text{lex}} \vec{i}} \text{write}(\vec{i}') \right) \\
\text{Out}(\vec{i}) &= \bigcup_{\vec{i} \in \text{Tile}(\vec{i})} \text{write}(\vec{i}) \\
\text{Load}(\vec{i}) &= \bigcup_{\vec{i} \in \text{Tile}(\vec{i})} \left( \text{read}(\vec{i}) \setminus \bigcup_{\vec{i}' < \vec{i}} \text{read}(\vec{i}') \cup \text{write}(\vec{i}') \right) \cup \text{In}(\vec{i}) \setminus \bigcup_{\vec{i}' \prec_s \vec{i}} \text{In}(\vec{i}') \cup \text{Out}(\vec{i}')
\end{align*}
\]
Approximations: why?

Some operations *may* execute

- if conditions that are not analyzable.

Some data *may* be accessed

- access functions that are not fully analyzable.

Approximated In/Out sets for tiles  

- due to the analysis (e.g., array regions);
- by choice to represent simpler sets (e.g., hyper-rectangles);
- to simplify the analysis (e.g., Fourier-Motzkin).

Approximated Load/Store sets

- to simplify code generation;
- to perform communications by blocks;
- to simplify memory allocation;
- ...
Equality of unions

"Exact approximated" load formula

\[
\text{Load}(\vec{i}) = \overline{\text{Ra}_\vec{i}} \cap ((\overline{\text{In}'} \cup \text{Out})(\vec{i}) \setminus (\overline{\text{In}'} \cup \text{Out})(\vec{i}' \sqsubseteq \vec{s} \vec{i}))
\]
Equality of unions

“Exact approximated” load formula

$$\text{Load}(\vec{i}) = \overline{\text{Ra}_I} \cap ((\overline{\text{In}}' \cup \overline{\text{Out}})(\vec{i}) \setminus (\overline{\text{In}}' \cup \overline{\text{Out}})(\vec{i}' \sqsubseteq \vec{s} \vec{i}))$$

Simplified “exact” load formula, with aligned tiles

$$\text{Load}(\vec{i}) = (\overline{\text{In}} \cup \overline{\text{Out}})(\vec{i}) \setminus (\overline{\text{In}} \cup \overline{\text{Out}})(\vec{i}' \sqsubseteq \vec{s} \vec{i})$$
Equality of unions

“Exact approximated” load formula

\[
\text{Load}(\vec{I}) = \text{Ra}_{\vec{I}} \cap (\text{In}' \cup \text{Out})(\vec{I}) \setminus (\text{In}' \cup \text{Out})(\vec{I}' \sqsubseteq \vec{s} \vec{I})
\]

Simplified “exact” load formula, with aligned tiles

\[
\text{Load}(\vec{I}) = (\text{In} \cup \text{Out})(\vec{I}) \setminus \bigcup\limits_{\vec{I}' \sqsubseteq \vec{s} \vec{I}} (\text{In} \cup \text{Out})(\vec{I}')
\]
Equality of unions

“Exact approximated” load formula

\[ \text{Load}(\vec{i}) = \overline{\text{Ra}_\vec{i}} \cap ((\overline{\text{In}'} \cup \text{Out})(\vec{i}) \setminus (\overline{\text{In}'} \cup \text{Out})(\vec{i}' \sqsubseteq \vec{s} \vec{i})) \]

Simplified “exact” load formula, with aligned tiles

\[ \text{Load}(\vec{i}) = F(\vec{i}) \setminus \bigcup_{\vec{i}' \sqsubseteq \vec{s} \vec{i}} F(\vec{i}') \]
Equality of unions

“Exact approximated” load formula

\[
\text{Load}(\vec{i}) = Ra_{\vec{i}} \cap ((\text{In}' \cup \text{Out})(\vec{i}) \setminus (\text{In}' \cup \text{Out})(\vec{i}' \sqsupset \vec{s} \vec{i}))
\]

Simplified “exact” load formula, with aligned tiles or all tiles?

\[
\text{Load}(\vec{i}) = F(\vec{i}) \setminus \bigcup_{\vec{i}' \sqsubseteq \vec{s} \vec{i}} F(\vec{i}') \neq F(\vec{i}) \setminus \bigcup_{\vec{i}' \prec \vec{s} \vec{i}} F(\vec{i}')
\]
Equality of unions

“Exact approximated” load formula

\[
\text{Load}(\vec{I}) = \bar{\text{Ra}}_I \cap ((\text{In}' \cup \text{Out}')(\vec{I}) \setminus (\text{In}' \cup \text{Out})(\vec{I}' \sqsubseteq_\vec{s} \vec{I}))
\]

Simplified “exact” load formula, with aligned tiles or all tiles?

\[
\text{Load}(\vec{I}) = F(\vec{I}) \setminus \bigcup_{\vec{I}' \sqsubseteq_\vec{s} \vec{I}} F(\vec{I}') \overset{?}{=} F(\vec{I}) \setminus \bigcup_{\vec{I}' \prec_\vec{s} \vec{I}} F(\vec{I}')
\]

Definition (Function stable for unions)

\(F : \mathcal{C} \subseteq \mathcal{P}(\mathcal{A}) \rightarrow \mathcal{P}(\mathcal{B})\) is stable for unions iff \(\forall \mathcal{C}', \mathcal{C}'' \subseteq \mathcal{C}, \bigcup_{X \in \mathcal{C}'} X = \bigcup_{X \in \mathcal{C}''} X \Rightarrow \bigcup_{X \in \mathcal{C}'} F(X) = \bigcup_{X \in \mathcal{C}''} F(X)\).

\[
\bigcup_{\vec{I}' \sqsubseteq_\vec{s} \vec{I}} \text{Tile}(\vec{I}') = \bigcup_{\vec{I}' \prec_\vec{s} \vec{I}} \text{Tile}(\vec{I}') \overset{?}{=} \bigcup_{\vec{I}' \sqsubseteq_\vec{s} \vec{I}} F(\vec{I}') = \bigcup_{\vec{I}' \prec_\vec{s} \vec{I}} F(\vec{I}')
\]
Pointwise functions

Definition (Function stable for unions)

\[ F : \mathcal{C} \subseteq \mathcal{P}(\mathcal{A}) \rightarrow \mathcal{P}(\mathcal{B}) \text{ is stable for unions iff } \forall C', C'' \subseteq \mathcal{C}, \ \cup_{X \in C'} X = \cup_{X \in C''} X \Rightarrow \cup_{X \in C'} F(X) = \cup_{X \in C''} F(X). \]

equivalent to

Definition (Pointwise function)

\( \mathcal{A}, \mathcal{B} \text{ two sets, } \mathcal{C} \subseteq \mathcal{P}(\mathcal{A}). F : \mathcal{C} \rightarrow \mathcal{P}(\mathcal{B}) \text{ is pointwise iff there exists } f : \mathcal{A} \rightarrow \mathcal{P}(\mathcal{B}) \text{ such that } \forall X \in \mathcal{C}, F(X) = \bigcup_{x \in X} f(x). \)

Ex:

\[ F(\vec{i}) = (\overline{\text{In}} \cup \overline{\text{Out}})(\vec{i}) = \bigcup_{\vec{i} \in T(\vec{i})} (\overline{\text{read}} \cup \overline{\text{write}})(\vec{i}). \]
Pointwise functions

Definition (Function stable for unions)

\[ F : \mathcal{C} \subseteq \mathcal{P}(A) \rightarrow \mathcal{P}(B) \] is stable for unions iff \( \forall \mathcal{C}', \mathcal{C}'', \subseteq \mathcal{C}, \)
\[ \bigcup_{X \in \mathcal{C}'} X = \bigcup_{X \in \mathcal{C}''} X \Rightarrow \bigcup_{X \in \mathcal{C}'} F(X) = \bigcup_{X \in \mathcal{C}''} F(X). \]

Definition (Pointwise function)

\( \mathcal{A}, \mathcal{B} \) two sets, \( \mathcal{C} \subseteq \mathcal{P}(A) \). \( F : \mathcal{C} \rightarrow \mathcal{P}(B) \) is pointwise iff there exists \( f : \mathcal{A} \rightarrow \mathcal{P}(B) \) such that \( \forall X \in \mathcal{C}, F(X) = \bigcup_{x \in X} f(x). \)

Ex: \( F(\vec{i}) = (\overline{\text{In}} \cup \overline{\text{Out}})(\vec{i}) = \bigcup_{\vec{i} \in T(\vec{i})} (\overline{\text{read}} \cup \overline{\text{write}})(\vec{i}). \)

Point-wise approximations

- Largest pointwise under-approximation: \( \underline{f}(x) = \bigcap_{Y \in \mathcal{C}, x \in Y} F(Y). \)
- Pointwise over-approximations schemes are possible.
Outline

1 Motivation and challenges

2 Parametric analysis

3 Current implementation and results
   - Current status
   - Script with ISCC
   - Local memory allocation for PolyBench examples
In progress: development of an automated tool

- **ISCC script** (see demo) $\Rightarrow$ complete tool based on ISL.
- Implement approximation schemes: due to code and/or by choice (**complexity issues**). Integrate with PIPS?
- Improve memory size computation: complexity issues, schedules (parallelism), piecewise lattice-based allocation.

**To do:** experiments with blocking (see also DATE’13)

- FPGA? Workstation? GPU? Kalray MPPA?
- Cost model for hierarchical tiling.
- Other schemes of reuse (partial storage).

**Pointwise functions**

- Useful for other approximations?
# Inputs

Params := [N, s_1, s_2] -> { : s_1 >= 0 and s_2 >= 0 }; 
Domain := [N] -> { 
    S_1[k] : 0 <= k < 2N-1; 
    S_2[i, j] : 0 <= i, j < N; 
} * Params; 

Read := [N] -> { 
    # Read access functions 
    S_2[i, j] -> A[i]; 
    S_2[i, j] -> B[j]; 
    S_2[i, j] -> C[i+j]; 
} * Domain; 

Write := [N] -> { 
    # Write access functions 
    S_1[k] -> C[k]; 
    S_2[i, j] -> C[i+j]; 
} * Domain; 

Theta := [N] -> { 
    # Preliminary mapping 
    S_1[k] -> [k, 0, 0]; 
    S_2[i, j] -> [i+j, i, 1]; 
};
# Tools for set manipulations

Tiling := \([s_1, s_2] \to \{ \# \text{ Two dimensional tiling} \)
\[
\begin{align*}
&\left[ [I_1, I_2] \to [i_1, i_2, k] \right] \to [i_1, i_2, k] : \\
&I_1 \leq i_1 < I_1 + s_1 \text{ and } I_2 \leq i_2 < I_2 + s_2 
\end{align*}
\]

Coalesce := \{ \left[ [I_1, I_2] \to [i_1, i_2, k] \right] \};

Strip := \{ \left[ [I_1, I_2] \to [I_1', I_2'] \right] \};

Prev := \{ \# \text{ Lexicographic order} \)
\[
\begin{align*}
&\left[ [I_1, I_2] \to [i_1, i_2, k] \right] \to \left[ [I_1, I_2] \to [i_1', i_2', k'] \right] : \\
i_1' \leq i_1 - 1 \text{ or } (i_1' \leq i_1 \text{ and } i_2' \leq i_2 - 1) \\
or (i_1' \leq i_1 \text{ and } i_2' \leq i_2 \text{ and } k' \leq k - 1)
\end{align*}
\]

TiledPrev := \([s_1, s_2] \to \{ \# \text{ Special \textquote{lexicographic} order} \)
\[
\begin{align*}
&\left[ [I_1, I_2] \to [I_1', I_2'] \right] : I_1' \leq I_1 - s_1 \text{ or} \\
&(I_1' \leq I_1 \text{ and } I_2' \leq I_2 - s_2) \right) \}
\]

TiledNext := TiledPrev^-1;

TiledRead := Tiling.(Theta^-1).Read;

TiledWrite := Tiling.(Theta^-1).Write;
# Set/relation computations

In := Coalesce.(TiledRead - (Prev.TiledWrite));
Out := Coalesce.TiledWrite;
Load := In - ((TiledPrev.In) + (TiledPrev.Out));
Store := Out - (TiledNext.Out);
print coalesce (Load % Params);
print coalesce (Store % Params);
Pipelined schedule

\[
\begin{align*}
\text{Load}(0) & \rightarrow \text{Compute}(0) & \rightarrow \text{Store}(0) \\
\text{Load}(1) & \rightarrow \text{Compute}(1) & \rightarrow \text{Store}(1) \\
\text{Load}(2) & \rightarrow \text{Compute}(2) & \rightarrow \text{Store}(2) \\
\text{Load}(3) & \rightarrow \text{Compute}(3) & \rightarrow \text{Store}(3)
\end{align*}
\]
## Sizes of arrays in local memory

<table>
<thead>
<tr>
<th>Transformation for tiling</th>
<th>Sequential memory size</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>jacobi-1d-imper</strong></td>
<td></td>
</tr>
<tr>
<td>$S_0(t, i) \mapsto (t, 2t + i, 0)$</td>
<td>A[$2s_1 + s_2$]</td>
</tr>
<tr>
<td>$S_1(t, j) \mapsto (t, 2t + j + 1, 1)$</td>
<td>B[$2s_1 + s_2 - 1$]</td>
</tr>
<tr>
<td><strong>jacobi-2d-imper</strong></td>
<td></td>
</tr>
<tr>
<td>$S_0(t, i, j) \mapsto (t, 2t + i, 2t + i + j, 0)$</td>
<td>A[$2s_1 + s_2, \min(2s_1, s_2 + 1) + s_3$]</td>
</tr>
<tr>
<td>$S_1(t, i, j) \mapsto (t, 2t + i + 1, 2t + i + j + 1, 1)$</td>
<td>B[$2s_1 + s_2 - 1, \min(2s_1, s_2) + s_3 - 1$]</td>
</tr>
<tr>
<td><strong>seidel-2d</strong></td>
<td></td>
</tr>
<tr>
<td>$S_0(t, i, j) \mapsto (t, t + i, 2t + i + j)$</td>
<td>A[$s_1 + s_2 + 1$, $\min(2s_1 + 2, s_1 + s_2, 2s_2 + 2) + s_3$]</td>
</tr>
<tr>
<td><strong>floyd-warshall</strong></td>
<td></td>
</tr>
<tr>
<td>$S_0(k, i, j) \mapsto (k, i, j)$</td>
<td>path $[\max(k + 1, n - k), \max(k + 1, n - k)]$</td>
</tr>
</tbody>
</table>
## Sizes of arrays in local memory

<table>
<thead>
<tr>
<th>Transformation for tiling</th>
<th>Pipelined memory size</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>jacobi-1d-imper</strong></td>
<td></td>
</tr>
<tr>
<td>(S_0(t, i) \mapsto (t, 2t + i, 0))</td>
<td>(A[2s_1 + 2s_2])</td>
</tr>
<tr>
<td>(S_1(t, j) \mapsto (t, 2t + j + 1, 1))</td>
<td>(B[2s_1 + 2s_2 - 2])</td>
</tr>
<tr>
<td><strong>jacobi-2d-imper</strong></td>
<td></td>
</tr>
<tr>
<td>(S_0(t, i, j) \mapsto (t, 2t + i, 2t + i + j, 0))</td>
<td>(A[2s_1 + s_2, \min(2s_1, s_2 + 1) + 2s_3])</td>
</tr>
<tr>
<td>(S_1(t, i, j) \mapsto (t, 2t + i + 1, 2t + i + j + 1, 1))</td>
<td>(B[2s_1 + s_2 - 1, \min(2s_1, s_2 + 1) + 2s_3 - 2])</td>
</tr>
<tr>
<td><strong>seidel-2d</strong></td>
<td></td>
</tr>
<tr>
<td>(S_0(t, i, j) \mapsto (t, t + i, 2t + i + j))</td>
<td>(A[\min(2s_1 + 2, s_1 + s_2, 2s_2 + 2) + 2s_3, s_1 + s_2 + 1])</td>
</tr>
<tr>
<td><strong>floyd-warshall</strong></td>
<td>path</td>
</tr>
<tr>
<td>(S_0(k, i, j) \mapsto (k, i, j))</td>
<td>(\max(k + 1, n - k), \max(k + 1, n - k, 2s_2))</td>
</tr>
</tbody>
</table>
Questions ?