Parametric Tiling with Inter-Tile Data Reuse

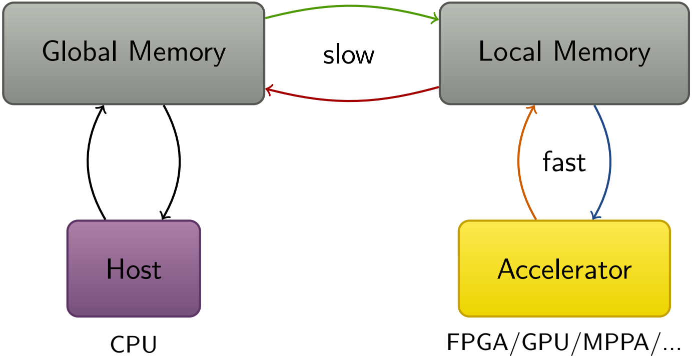
Alexandre Isoard **Alain Darte**

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

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

Outline

Kernel Offloading



- Perform computations by blocks;
- Exploit data reuse;
- Use pipelining/prefetching;
- Reduce and coalesce communications (burst).

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).

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

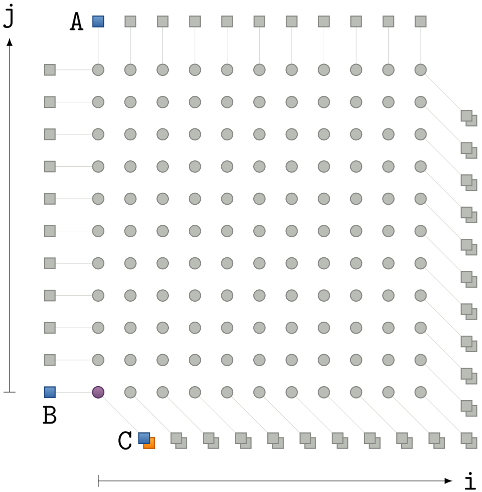
Reads/writes are functions of iteration points. Can we express the relation "happens before" among iterations in a quasi-affine way?

☛ 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.

Reads, writes, schedule

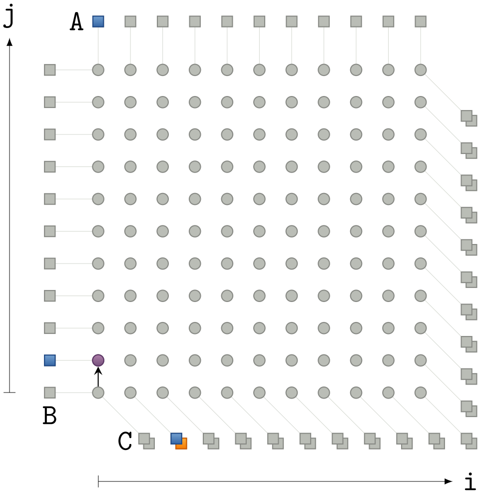


Product of two polynomials:

- arguments in A and B;
- result in 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

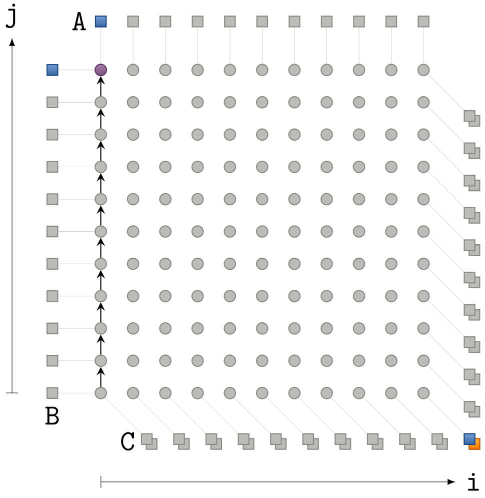


Product of two polynomials:

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

```
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

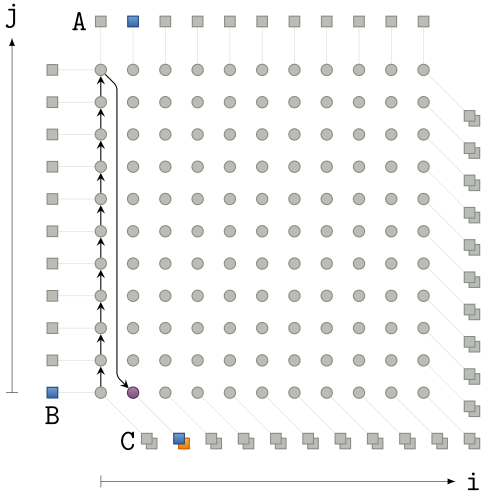


Product of two polynomials:

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

```
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

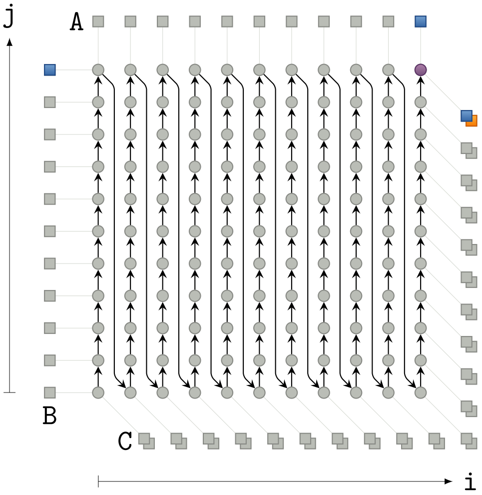


Product of two polynomials:

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

```
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

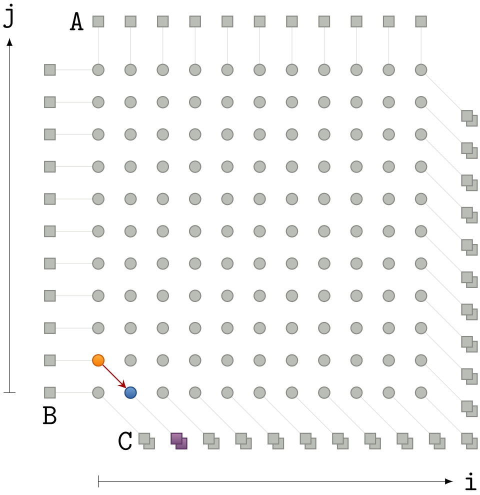


Product of two polynomials:

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

```
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
  }
}
```

Dependences

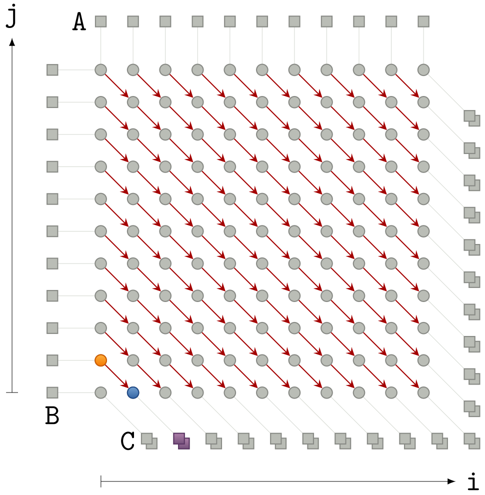


Product of two polynomials:

- arguments in A and B;
- result in 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
  }
}
```

Dependences

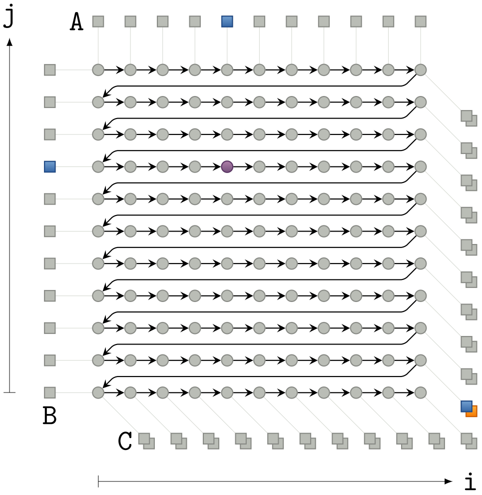


Product of two polynomials:

- arguments in A and B;
- result in 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

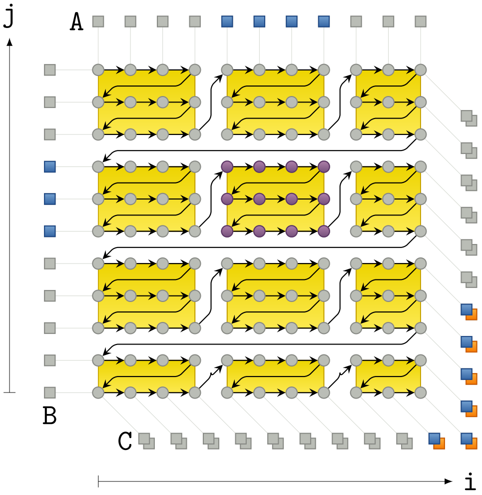


Product of two polynomials:

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

```
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+tiling

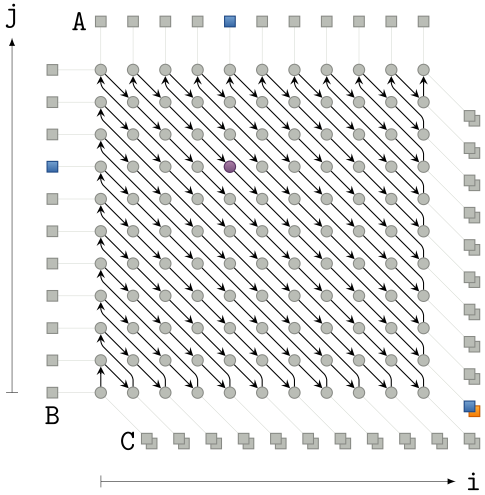


Product of two polynomials:

- arguments in A and B;
- result in 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 skewing

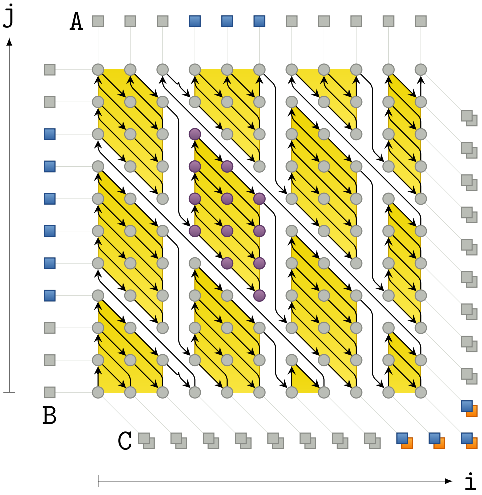


Product of two polynomials:

- arguments in A and B;
- result in 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 skewing+tiling

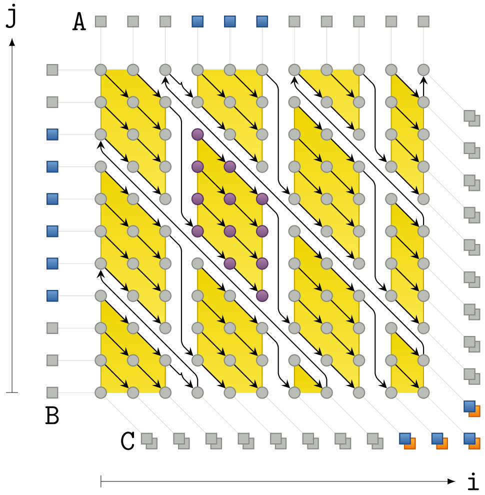


Product of two polynomials:

- arguments in A and B;
- result in 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 skewing+tiling



Product of two polynomials:

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

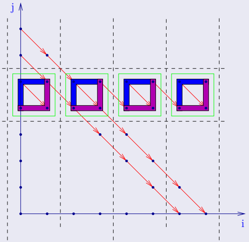
```
for(int k=0; k<2*n-1; k++) {
  C[k] = 0; // S0
}

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.

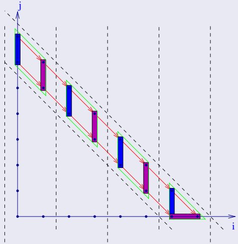
Inter-tile data reuse in a tile strip

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

$(i, j) \mapsto (n - j - 1, i)$



$(i, j) \mapsto (i + j, i)$

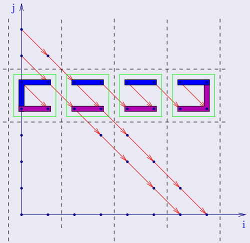


In a tile, **Load** $\simeq$ first read, **Store** $\simeq$ last write.

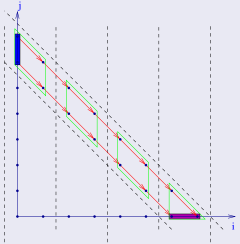
Inter-tile data reuse in a tile strip

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

$(i, j) \mapsto (n - j - 1, i)$



$(i, j) \mapsto (i + j, i)$

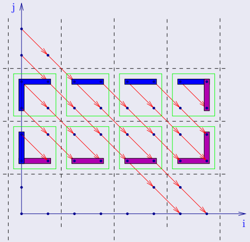


In a tile strip, **Load** $\simeq$ first read, **Store** $\simeq$ last write.

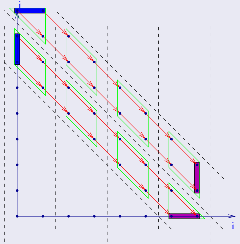
Inter-tile data reuse in a tile strip

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

$(i,j) \mapsto (n-j-1, i)$

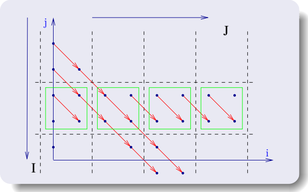


$(i,j) \mapsto (i+j, i)$



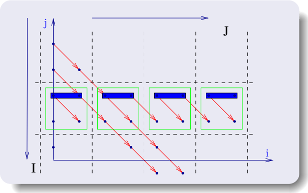
In a reuse domain, **Load** $\simeq$ first read, **Store** $\simeq$ last write.
Can actually be adapted to any parameterized reuse domain.

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 Load_A, Load_B, Load_C, Store_C.

Objective: data transfers

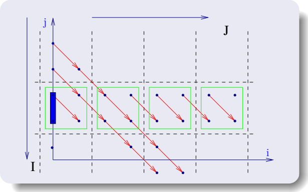


- Bound n, tiles of size $b \times b$.
- Tiling with $(i,j) \mapsto (i',j') = (n-j-1, i)$.
- Access functions $m = i + j = j' + n - i' - 1$.
- Tile origin (I, J).
- Transfers Load_A, Load_B, Load_C, Store_C.

Load sets.

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

Objective: data transfers



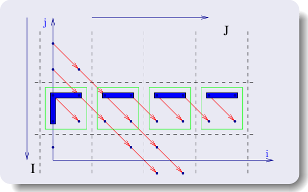
- Bound n, tiles of size $b \times b$.
- Tiling with $(i,j) \mapsto (i',j') = (n-j-1, i)$.
- Access functions $m = i + j = j' + n - i' - 1$.
- Tile origin (I, J).
- Transfers Load_A, Load_B, Load_C, Store_C.

Load sets.

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

$$\mathrm{Load}_B = \{m \mid J = 0,\ 0 \leq m \leq n-1,\ n - I - b \leq m \leq n - I - 1\}$$

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 Load_A, Load_B, Load_C, Store_C.

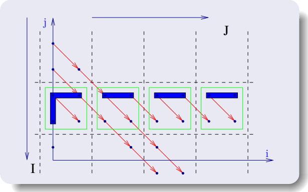
Load sets.

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

$$\mathrm{Load}_B = \{m \mid J = 0,\ 0 \le m \le n-1,\ n-I-b \le m \le n-I-1\}$$

$$\begin{aligned}\mathrm{Load}_C = & \{m \mid 0 \le m,\ n-I-b \le m \le n-1-I,\ J=0\} \\ & \cup \{m \mid \max(1,J) \le m+I-n+1 \le \min(n-1, J+b-1)\}\end{aligned}$$

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 Load_A, Load_B, Load_C, Store_C.

Load sets. Local memory sizes with "double-buffering".

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

- size $2b$, when $n \geq 2b+1$: at least 2 tiles available.
- size n when $n \leq 2b$: less than 2 tiles.

$$\mathrm{Load}_B = \{m \mid J = 0,\ 0 \leq m \leq n-1,\ n-I-b \leq m \leq n-I-1\}$$

- size b when $n \geq b$: 1 full tile.
- size n when $n \leq b-1$: 1 partial tile.

$$\mathrm{Load}_C = \{m \mid 0 \leq m,\ n-I-b \leq m \leq n-1-I,\ J = 0\}$$
$$\cup \{m \mid \max(1, J) \leq m + I - n + 1 \leq \min(n-1, J+b-1)\}$$

- 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

Tiling, tiles, and schedules

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

$$\vec{i} \in \mathrm{Tile}(\vec{T}) \Leftrightarrow \left\{ \begin{array}{c} s_1 T_1 \leq i_1 < s_1(T_1 + 1) \\ \vdots \\ s_n T_n \leq i_n < s_n(T_n + 1) \end{array} \right.$$

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

Tiling, tiles, and schedules

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

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

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

With indices of tile origins

$$\vec{i} \in \text{Tile}(\vec{I}) \Leftrightarrow \begin{cases} I_1 \leq i_1 < I_1 + s_1 \\ \vdots \\ I_n \leq i_n < I_n + s_n \end{cases}$$

with $\vec{I}$, origin of 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:

$$\vec{i'} <_{\vec{I}} \vec{i} \Leftrightarrow \vec{i'} <_{\vec{I'}} \vec{i} \Leftrightarrow (\vec{I'}, \vec{i'}) <_{lex} (\vec{I}, \vec{i}) \text{ and } \vec{I'} \overset{\vec{s}}{\equiv} \vec{I}$$

Intuitive expression of Load/Store sets

For $\mathrm{Tile}(\vec{I})$ with data reuse in ReuseDomain:

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

$$\mathrm{Store}(\vec{I}) = \bigcup_{\vec{i}\in\mathrm{Tile}(\vec{I})} \left(\mathrm{write}(\vec{i}) \setminus \bigcup_{\substack{\vec{i'}>\vec{i} \\ \vec{i'}\in\mathrm{ReuseDomain}}} \mathrm{write}(\vec{i'}) \right)$$

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

Intuitive expression of Load/Store sets

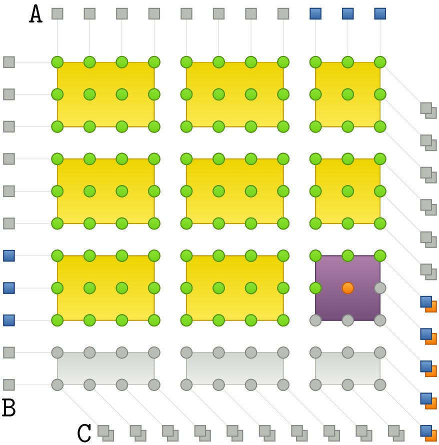
For Tile($\vec{I}$) with data reuse in ReuseDomain:

$$\mathsf{Load}(\vec{I}) = \bigcup_{\vec{i} \in \mathsf{Tile}(\vec{I})} \left(\mathrm{read}(\vec{i}) \setminus \bigcup_{\substack{\vec{i'} < \vec{i} \\ \vec{i'} \in \mathsf{ReuseDomain}}} \mathrm{read}(\vec{i'}) \cup \mathrm{write}(\vec{i'}) \right)$$

$$\mathsf{Store}(\vec{I}) = \bigcup_{\vec{i} \in \mathsf{Tile}(\vec{I})} \left(\mathrm{write}(\vec{i}) \setminus \bigcup_{\substack{\vec{i'} > \vec{i} \\ \vec{i'} \in \mathsf{ReuseDomain}}} \mathrm{write}(\vec{i'}) \right)$$

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?

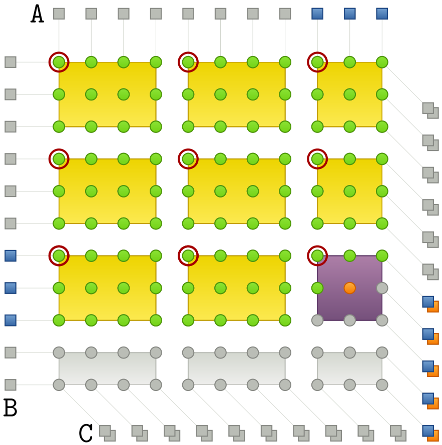
Tiling, relation “happens before” and unaligned tiles



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

- $\vec{i} \in \text{Tile}(\vec{T})$ and $\vec{i'} \in \text{Tile}(\vec{T'})$
- $(\vec{T'}, \vec{i'}) <_{lex} (\vec{T}, \vec{i})$

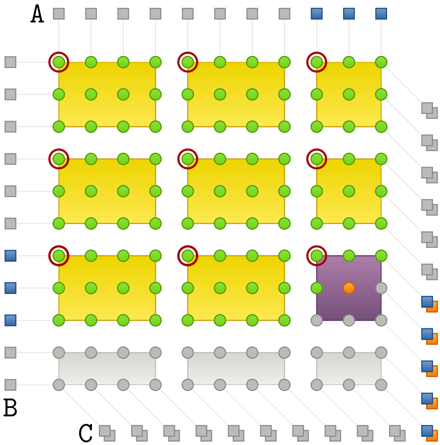
Tiling, relation "happens before" and unaligned tiles



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

- $\vec{i} \in \mathrm{Tile}(\vec{I})$ and $\vec{i'} \in \mathrm{Tile}(\vec{I'})$
- $(\vec{I'}, \vec{i'}) <_{lex} (\vec{I}, \vec{i})$ and $\vec{I'} \stackrel{\vec{s}}{\equiv} \vec{I}$

Tiling, relation "happens before" and unaligned tiles



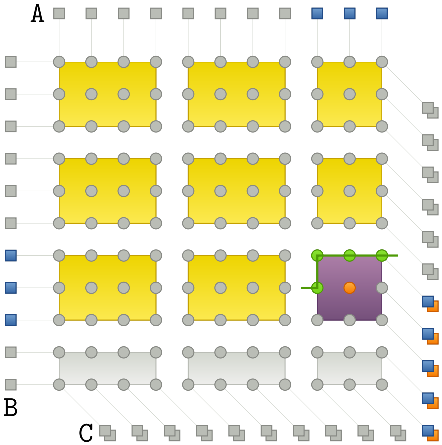
$\vec{i'} <_{\vec{I}} \vec{i}$ iff

- $\vec{i} \in \mathrm{Tile}(\vec{I})$ and $\vec{i'} \in \mathrm{Tile}(\vec{I'})$
- $\vec{I'} = \vec{I} \wedge \vec{i'} <_{lex} \vec{i}$

  or

  $\vec{I'} <_{lex} \vec{I} \wedge \vec{I'} \stackrel{\vec{s}}{\equiv} \vec{I} \Leftrightarrow \vec{I} \sqsubset_{\vec{s}} \vec{I'}$

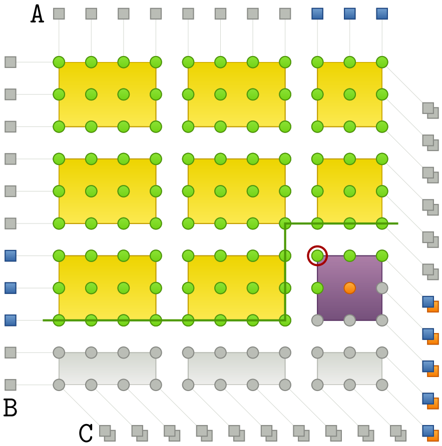
Tiling, relation “happens before” and unaligned tiles



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

- $\vec{i} \in \mathrm{Tile}(\vec{I})$ and $\vec{i'} \in \mathrm{Tile}(\vec{I'})$
- $\vec{I'} = \vec{I} \wedge \vec{i'} <_{lex} \vec{i}$
  or
  $\vec{I'} <_{lex} \vec{I} \wedge \vec{I'} \stackrel{\vec{s}}{\equiv} \vec{I} \Leftrightarrow \vec{I} \sqsubset_{\vec{s}} \vec{I'}$

Tiling, relation “happens before” and unaligned tiles



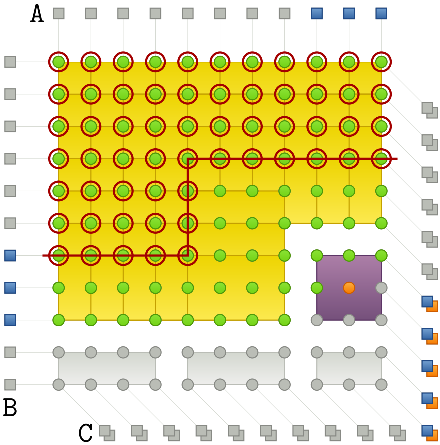
$\vec{i'} <_{\vec{I}} \vec{i}$ iff

- $\vec{i} \in \mathrm{Tile}(\vec{I})$ and $\vec{i'} \in \mathrm{Tile}(\vec{I'})$
- $\vec{I'} = \vec{I} \wedge \vec{i'} <_{lex} \vec{i}$

  or

$(i'_1 < I_1) \vee (i'_1 < I_1 + s_1 \wedge i'_2 < I_2)$

Tiling, relation “happens before” and unaligned tiles



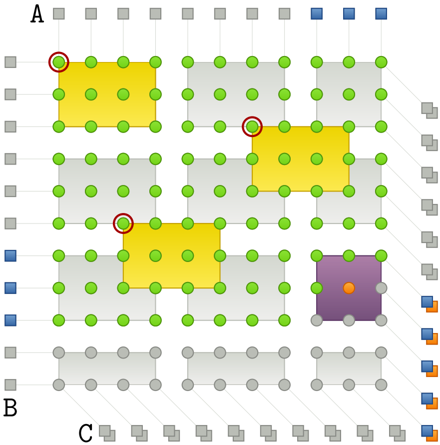
$\vec{i'} <_{\vec{I}} \vec{i}$ iff

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

or

$(I'_1 \le I_1 - s_1) \vee (I'_1 \le I_1 \wedge I'_2 \le I_2 - s_2)$

Tiling, relation “happens before” and unaligned tiles



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

- $\vec{i} \in \mathrm{Tile}(\vec{I})$ and $\vec{i'} \in \mathrm{Tile}(\vec{I'})$
- $\vec{I'} = \vec{I} \wedge \vec{i'} <_{lex} \vec{i}$

  or

  $\vec{I'} \prec_{\vec{s}} \vec{I}$: partial order on tiles (aligned and unaligned tiles)

Load/Store computations with In/Out sets

Contribution of reads/writes summarized at tile level:

$$\begin{cases} \mathrm{In}(\vec{I}) = \bigcup_{\vec{i}\in\mathsf{Tile}(\vec{I})} \left(\mathrm{read}(\vec{i}) \setminus \bigcup_{\vec{i'}\in\mathsf{Tile}(\vec{I}),\, \vec{i'} <_{lex} \vec{i}} \mathrm{write}(\vec{i'}) \right) \\ \mathrm{Out}(\vec{I}) = \bigcup_{\vec{i}\in\mathsf{Tile}(\vec{I})} \mathrm{write}(\vec{i}) \end{cases}$$

Load/Store computations with In/Out sets

Contribution of reads/writes summarized at tile level:

$$\begin{cases} \mathrm{In}(\vec{I}) = \bigcup\limits_{\vec{i} \in \mathsf{Tile}(\vec{I})} \left(\mathrm{read}(\vec{i}) \setminus \bigcup\limits_{\vec{i'} \in \mathsf{Tile}(\vec{I}),\, \vec{i'} <_{lex} \vec{i}} \mathrm{write}(\vec{i'}) \right) \\ \mathrm{Out}(\vec{I}) = \bigcup\limits_{\vec{i} \in \mathsf{Tile}(\vec{I})} \mathrm{write}(\vec{i}) \end{cases}$$

$$\mathrm{Load}(\vec{I}) = \bigcup_{\vec{i} \in \mathsf{Tile}(\vec{I})} \left(\mathrm{read}(\vec{i}) \setminus \bigcup_{\substack{\vec{i'} < \vec{i} \\ \vec{i'} \in \mathsf{ReuseDomain}}} \mathrm{read}(\vec{i'}) \cup \mathrm{write}(\vec{i'}) \right)$$

☛ $$\mathrm{Load}(\vec{I}) = \mathrm{In}(\vec{I}) \setminus \left(\bigcup_{\vec{I'} \prec_{\vec{s}} \vec{I}} \mathrm{In}(\vec{I'}) \cup \mathrm{Out}(\vec{I'}) \right)$$

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 ☛ $\overline{\mathrm{In}}$, $\overline{\mathrm{Out}}$, $\underline{\mathrm{Out}}$.

- 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 ☛ $\overline{\mathrm{Store}}$, $\overline{\mathrm{Load}}$.

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

Equality of unions

"Exact approximated" load formula

$$\mathrm{Load}(\vec{I}) = \overline{\mathrm{Ra}}_{\vec{I}} \cap ((\overline{\mathrm{In}}' \cup \overline{\mathrm{Out}})(\vec{I}) \setminus (\overline{\mathrm{In}}' \cup \overline{\mathrm{Out}})(\vec{I'} \sqsubset_{\vec{s}} \vec{I}))$$

Equality of unions

"Exact approximated" load formula

$$\mathrm{Load}(\vec{I}) = \overline{\mathrm{Ra}}_{\vec{I}} \cap ((\overline{\mathrm{In}}' \cup \overline{\mathrm{Out}})(\vec{I}) \setminus (\overline{\mathrm{In}}' \cup \overline{\mathrm{Out}})(\vec{I'} \sqsubset_{\vec{s}} \vec{I}))$$

Simplified "exact" load formula, with aligned tiles

$$\mathrm{Load}(\vec{I}) = (\overline{\mathrm{In}} \cup \overline{\mathrm{Out}})(\vec{I}) \setminus (\overline{\mathrm{In}} \cup \overline{\mathrm{Out}})(\vec{I'} \sqsubset_{\vec{s}} \vec{I})$$

Equality of unions

"Exact approximated" load formula

$$\mathrm{Load}(\vec{I}) = \overline{\mathrm{Ra}}_{\vec{I}} \cap ((\overline{\mathrm{In}}' \cup \overline{\mathrm{Out}})(\vec{I}) \setminus (\overline{\mathrm{In}}' \cup \overline{\mathrm{Out}})(\vec{I}' \sqsubset_{\vec{s}} \vec{I}))$$

Simplified "exact" load formula, with aligned tiles

$$\mathrm{Load}(\vec{I}) = (\overline{\mathrm{In}} \cup \overline{\mathrm{Out}})(\vec{I}) \setminus \bigcup_{\vec{I}' \sqsubset_{\vec{s}} \vec{I}} (\overline{\mathrm{In}} \cup \overline{\mathrm{Out}})(\vec{I}')$$

Equality of unions

"Exact approximated" load formula

$$\mathrm{Load}(\vec{I}) = \overline{\mathrm{Ra}}_{\vec{I}} \cap ((\overline{\mathrm{In}}' \cup \overline{\mathrm{Out}})(\vec{I}) \setminus (\overline{\mathrm{In}}' \cup \overline{\mathrm{Out}})(\vec{I'} \sqsubset_{\vec{s}} \vec{I}))$$

Simplified "exact" load formula, with aligned tiles

$$\mathrm{Load}(\vec{I}) = F(\vec{I}) \setminus \bigcup_{\vec{I'} \sqsubset_{\vec{s}} \vec{I}} F(\vec{I'})$$

Equality of unions

"Exact approximated" load formula

$$\mathrm{Load}(\vec{I}) = \overline{\mathrm{Ra}}_{\vec{I}} \cap ((\overline{\mathrm{In}}' \cup \overline{\mathrm{Out}})(\vec{I}) \setminus (\overline{\mathrm{In}}' \cup \overline{\mathrm{Out}})(\vec{I}' \sqsubset_{\vec{s}} \vec{I}))$$

Simplified "exact" load formula, with aligned tiles or all tiles?

$$\mathrm{Load}(\vec{I}) = F(\vec{I}) \setminus \bigcup_{\vec{I}' \sqsubset_{\vec{s}} \vec{I}} F(\vec{I}') \stackrel{?}{=} F(\vec{I}) \setminus \bigcup_{\vec{I}' \prec_{\vec{s}} \vec{I}} F(\vec{I}')$$

Equality of unions

"Exact approximated" load formula

$$\mathrm{Load}(\vec{I}) = \overline{\mathrm{Ra}}_{\vec{I}} \cap ((\overline{\mathrm{In}}' \cup \overline{\mathrm{Out}})(\vec{I}) \setminus (\overline{\mathrm{In}}' \cup \overline{\mathrm{Out}})(\vec{I}' \sqsubset_{\vec{s}} \vec{I}))$$

Simplified "exact" load formula, with aligned tiles or all tiles?

$$\mathrm{Load}(\vec{I}) = F(\vec{I}) \setminus \bigcup_{\vec{I}' \sqsubset_{\vec{s}} \vec{I}} F(\vec{I}') \stackrel{?}{=} F(\vec{I}) \setminus \bigcup_{\vec{I}' \prec_{\vec{s}} \vec{I}} F(\vec{I}')$$

Definition (Function stable for unions)

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

$$\bigcup_{\vec{I}' \sqsubset_{\vec{s}} \vec{I}} \mathrm{Tile}(\vec{I}') = \bigcup_{\vec{I}' \prec_{\vec{s}} \vec{I}} \mathrm{Tile}(\vec{I}') \stackrel{?}{\Rightarrow} \bigcup_{\vec{I}' \sqsubset_{\vec{s}} \vec{I}} F(\vec{I}') = \bigcup_{\vec{I}' \prec_{\vec{s}} \vec{I}} F(\vec{I}')$$

Pointwise functions

Definition (Function stable for unions)

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

equivalent to

Definition (Pointwise function)

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

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

Pointwise functions

Definition (Function stable for unions)

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

equivalent to

Definition (Pointwise function)

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

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

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

Current implementation and future work

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?

Script ISCC 1/3

```
# Inputs
Params := [N, s_1, s_2] -> { : s_1 >= 0 and s_2 >= 0 };
Domain := [N] -> { # Iteration domains
  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]; };
```

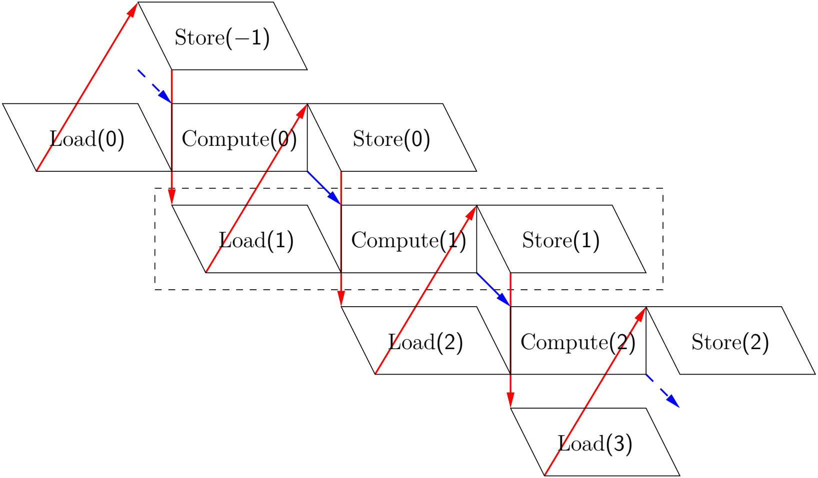
Script ISCC 2/3

```
# Tools for set manipulations
Tiling := [s_1, s_2] -> { # Two dimensional tiling
  [[I_1, I_2] -> [i_1, i_2, k]] -> [i_1, i_2, k] :
      I_1 <= i_1 < I_1 + s_1 and I_2 <= i_2 < I_2 + s_2 };
Coalesce := { [I_1, I_2] -> [[I_1, I_2] -> [i_1, i_2, k]] };
Strip := { [I_1, I_2] -> [I_1, I_2'] };
Prev := { # Lexicographic order
  [[I_1, I_2] -> [i_1, i_2, k]] -> [[I_1, I_2] -> [i_1', i_2', k']] :
      i_1' <= i_1 - 1 or (i_1' <= i_1 and i_2' <= i_2 - 1)
      or (i_1' <= i_1 and i_2' <= i_2 and k' <= k - 1) };
TiledPrev := [s_1, s_2] -> { # Special ``lexicographic'' order
  [I_1, I_2] -> [I_1', I_2'] : I_1' <= I_1 - s_1 or
      (I_1' <= I_1 and I_2' <= I_2 - s_2) } * Strip;
TiledNext := TiledPrev^-1;
TiledRead  := Tiling.(Theta^-1).Read;
TiledWrite := Tiling.(Theta^-1).Write;
```

Script ISCC 3/3

```
# 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



Sizes of arrays in local memory

Transformation for tiling	Sequential memory size
jacobi-1d-imper	
$S_0(t,i) \mapsto (\underline{t, 2t+i}, 0)$ $S_1(t,j) \mapsto (\underline{t, 2t+j+1}, 1)$	$\mathtt{A}[2s_1 + s_2]$ $\mathtt{B}[2s_1 + s_2 - 1]$
jacobi-2d-imper	
$S_0(t,i,j) \mapsto (\underline{t, 2t+i, 2t+i+j}, 0)$ $S_1(t,i,j) \mapsto (\underline{t, 2t+i+1, 2t+i+j+1}, 1)$	$\mathtt{A}[2s_1 + s_2, \min(2s_1, s_2+1) + s_3]$ $\mathtt{B}[2s_1 + s_2 - 1, \min(2s_1, s_2) + s_3 - 1]$
seidel-2d	
$S_0(t,i,j) \mapsto (\underline{t, t+i, 2t+i+j})$	$\mathtt{A}\begin{bmatrix} s_1 + s_2 + 1, \\ \min(2s_1+2, s_1+s_2, 2s_2+2) + s_3 \end{bmatrix}$
floyd-warshall	
$S_0(k,i,j) \mapsto (k, \underline{i,j})$	$\mathtt{path}\begin{bmatrix} \max(k+1, n-k), \\ \max(k+1, n-k) \end{bmatrix}$

Sizes of arrays in local memory

Transformation for tiling	Pipelined memory size
jacobi-1d-imper	
$S_0(t,i)\mapsto(\underline{t,2t+i},0)$ $S_1(t,j)\mapsto(\underline{t,2t+j+1},1)$	$\mathtt{A}[2s_1+2s_2]$ $\mathtt{B}[2s_1+2s_2-2]$
jacobi-2d-imper	
$S_0(t,i,j)\mapsto(\underline{t,2t+i,2t+i+j},0)$ $S_1(t,i,j)\mapsto(\underline{t,2t+i+1,2t+i+j+1},1)$	$\mathtt{A}[2s_1+s_2,\min(2s_1,s_2+1)+2s_3]$ $\mathtt{B}[2s_1+s_2-1,\min(2s_1,s_2+1)+2s_3-2]$
seidel-2d	
$S_0(t,i,j)\mapsto(\underline{t,t+i,2t+i+j})$	$\mathtt{A}\begin{bmatrix} s_1+s_2+1, \\ \min(2s_1+2,s_1+s_2,2s_2+2)+2s_3 \end{bmatrix}$
floyd-warshall	
$S_0(k,i,j)\mapsto(k,\underline{i,j})$	$\mathtt{path}\begin{bmatrix} \max(k+1,n-k), \\ \max(k+1,n-k,2s_2) \end{bmatrix}$

Merci

Questions ?