Compilation for Heterogeneous Multicores with Distributed Memory

Uday Bondhugula
Indian Institute of Science

First Spring School on Polyhedral Code Analysis and Optimizations
St Germain au Mont d’Or
May 13–17, 2013
1 Introduction and Foundation

2 Distributed-memory code generation
   • General issues
   • Challenges and past efforts
   • Pluto distmem (Bondhugula et al.)
   • Compilation for heterogeneous multicore
1 Introduction and Foundation

2 Distributed-memory code generation
A polyhedral optimizer – various phases

1. Extracting a polyhedral representation
2. Dependence analysis
3. Transformation and parallelization
4. Storage optimization
5. Code generation (getting out of polyhedral extraction)
A polyhedral optimizer – various phases

1. Extracting a polyhedral representation
2. Dependence analysis
3. Transformation and parallelization
4. Memory optimization
5. Code generation (getting out of polyhedral extraction)
Heterogeneous architectures

A NODE WITHIN A CLUSTER

- North Bridge
- South Bridge
- DDR RAM
- CPU
- GPU1
- GPU2
- GPU N
Distributed-memory code generation – why again?

- Large amount of literature already exists through early 1990s
- Make a fresh attempt to solve this problem
Large amount of literature already exists through early 1990s

1. Past works: limited or no success
2. Nothing **really** exists today
3. We now have new code generators, new polyhedral libraries, tools, algorithms
4. Renewed interest because the same techniques are needed to compile for CPUs-GPU heterogeneous systems

Make a fresh attempt to solve this problem
Distributed-memory code generation – why again?

- Large amount of literature already exists through early 1990s

- Make a fresh attempt to solve this problem
Communication during parallelization is a result of data dependences

No data dependences \( \Rightarrow (\sim) \) No communication

Parallelism in loop implies no dependence satisfied, why then do we need communication?

Communication is due to dependences that are satisfied outside but have (non-zero) components on the parallel loop

A quick recap of different forms of parallelism
Why do we need communication?

- Communication during parallelization is a result of data dependences
- No data dependences $\Rightarrow$ (≈) No communication
- Parallelism in loop implies no dependence satisfied, why then do we need communication?
  - Communication is due to dependences that are satisfied outside but have (non-zero) components on the parallel loop
  - A quick recap of different forms of parallelism
Why do we need communication?

- Communication during parallelization is a result of data dependences.
- No data dependences $\Rightarrow (\sim)$ No communication.
- Parallelism in loop implies no dependence satisfied, why then do we need communication?
- Communication is due to dependences that are satisfied outside but have (non-zero) components on the parallel loop.
- A quick recap of different forms of parallelism.
**Outer parallelism (loops)**

- **Introduction and Foundation**

- **Outer parallelism (loops)**

  ![Diagram](image)

  **Figure:** Outer parallel loop \( i \): hyperplane \((1,0)\)

- Also called communication-free parallelism
- With distance vectors, \textit{rank of dependence matrix} < \textit{dimensionality of iteration space}
- For affine dependences, \( \phi(\vec{t}) - \phi(\vec{s}) = 0, \quad \forall \ \vec{s}, \vec{t} \) that are dependent
Inner parallelism (loops)

- A loop that does not satisfy / carry any dependences and which cannot be made the outermost loop
- Affine: $\phi(\vec{t}) - \phi(\vec{s}) = 0$, $\forall \vec{s}, \vec{t}$ that are not satisfied by outer loops, and $\phi$ is an inner loop

Figure: Inner parallel loop, $j$: hyperplane (0,1)
Pipelined parallelism (loops)

Figure: Pipelined parallel loop: $i$ or $j$, wavefront: $\phi = (1, 1)$

- Can be converted to inner parallelism
- Occurs when all dependences are in the forward direction in two or more dimensions
- Tiling is possible
Pipelined parallelism (loops)

- Can be converted to inner parallelism
- Occurs when all dependences are in the forward direction in two or more dimensions
- Tiling is possible

Figure: Pipelined parallel loop: $i$ or $j$, wavefront: $\phi = (1, 1)$
Tiling / Coarse-grained pipelined parallelism

Figure: Tiling an iteration space

- Reduces the frequency of communication / synchronization
- Improves locality
1 Introduction and Foundation

2 Distributed-memory code generation
   - General issues
   - Challenges and past efforts
   - Pluto distmem (Bondhugula et al.)
   - Compilation for heterogeneous multicores
1. What to send?
2. Whom to send to?
Distributed-memory code generation

- A very hard problem
- Requires solution to a number of sub-problems (GPU compilation, explicit data copying, etc.)
- Difficulties
  - For non-uniform dependences, not known how far dependences traverse
  - Number of iterations (or tiles) is not known at compile time
  - Number of processors may not be known at compile time (portability)
  - Virtual to physical processor approach: are two receiving virtual processors the same physical processor?
An example

for (t=1; t<=T-1; t++){
    for (j=1; j<=N-1; j++){
        u[t%2][j] = 0.333 * (u[(t-1)%2][j-1]
                         + u[(t-1)%2][j] + u[(t-1)%2][j+1]);
    }
}
An example

\[
\text{for } (t = 1; t <= T - 1; t++) \{
    \text{for } (j = 1; j <= N - 1; j++) \{
        u[t\%2][j] = 0.333 \times (u[(t-1)\%2][j-1]
                        + u[(t-1)\%2][j]
                        + u[(t-1)\%2][j+1]);
    \}
\}
\]
Sub-problems

1. Analyzing data accessed
2. Packing and unpacking data
3. Computing buffer sizes
4. Generating actual communication primitives (easy)
Computing data accessed – an example

- Performing distributed memory code generation on transformed code

```c
for (t=0; t<=T-1; t++) {
    for (i=1; i<=N-2; i++) {
        for (j=1; j<=N-2; j++) {
            a[i][j] = (a[i-1][j-1] + a[i-1][j] + a[i-1][j+1] + a[i][j-1] + a[i][j] + a[i][j+1] + a[i+1][j-1] + a[i+1][j] + a[i+1][j+1])/9.0;
        }
    }
}
```

- Distance vectors: (0,1,1), (0,1,0), (0,1,-1), (0,0,1), (0,1,-1), (1,-1,1), (1,0,-1), (1,-1,0), (1,-1,-1)

  1. \( T(t, i, j) = (t, t + i, 2t + i + j) \)
  2. Tile all dimensions
  3. Create a tile schedule, and identify loop to be parallelized
  4. Generate communication primitives on this code
Performing distributed memory code generation on transformed code

```c
for (t=0; t<=T-1; t++) {
    for (i=1; i<=N-2; i++) {
        for (j=1; j<=N-2; j++) {
            a[i][j] = (a[i-1][j-1] + a[i-1][j] + a[i-1][j+1] + a[i][j-1] +
                        a[i][j] + a[i][j+1] +
                        a[i+1][j-1] + a[i+1][j] + a[i+1][j+1])/9.0;
        }
    }
}
```

Distance vectors: (0,1,1), (0,1,0), (0,1,-1), (0,0,1), (0,1,-1), (1,-1,1), (1,0,-1), (1,-1,0), (1,-1,-1)

1. \( T(t, i, j) = (t, t + i, 2t + i + j) \)
2. Tile all dimensions
3. Create a tile schedule, and identify loop to be parallelized
4. Generate communication primitives on this code
Performing distributed memory code generation on transformed code

```c
for (t=0; t<=T-1; t++) {
    for (i=1; i<=N-2; i++) {
        for (j=1; j<=N-2; j++) {
            a[i][j] = (a[i-1][j-1] + a[i-1][j] + a[i-1][j+1] + a[i][j-1] + a[i][j] + a[i][j+1] + a[i+1][j-1] + a[i+1][j] + a[i+1][j+1])/9.0;
        }
    }
}
```

Distance vectors: (0,1,1), (0,1,0), (0,1,-1), (0,0,1), (0,1,-1), (1,-1,1), (1,0,-1), (1,-1,0), (1,-1,-1)

1. $T(t, i, j) = (t, t + i, 2t + i + j)$
2. Tile all dimensions
3. Create a tile schedule, and identify loop to be parallelized
4. Generate communication primitives on this code
Performing distributed memory code generation on transformed code

```
for (t=0; t<=T-1; t++) {
    for (i=1; i<=N-2; i++) {
        for (j=1; j<=N-2; j++) {
            a[i][j] = (a[i-1][j-1] + a[i-1][j] + a[i-1][j+1] + a[i][j-1] + a[i][j] + a[i][j+1] + a[i+1][j-1] + a[i+1][j] + a[i+1][j+1])/9.0;
        }
    }
}
```

Distance vectors: (0,1,1), (0,1,0), (0,1,-1), (0,0,1), (0,1,-1), (1,-1,1), (1,0,-1), (1,-1,0), (1,-1,-1)

1. \( T(t, i, j) = (t, t + i, 2t + i + j) \)
2. Tile all dimensions
3. Create a tile schedule, and identify loop to be parallelized
4. Generate communication primitives on this code
Computing data accessed (in transformed code)

```c
if ((N >= 3) && (T >= 1)) {
    for (t1=1;t1<=floord(8*N+9*T-25,8);t1++) {
        lbp=max(ceild(8*t1-T+1,8),ceild(8*t1+1,9));
        ubp=min(floord(8*t1+N+5,9),t1);
        #pragma omp parallel for private (lbv , ubv)
        for (t2=lbv;t2<=ubp;t2++) {
            for (t3=max(8*t1-7*t2+1,2*t2-N+3);t3<=min(min(2*t2+N-3,8*t1-7*t2+N+5),t2+N+T-3);t3++) {
                for (t4=max(max(8*t1-8*t2,t2-N+2),-t2+t3-N+2);t4<=min(min(min(T-1,t2-1),8*t1-8*t2+7),-t2+t3-N+2);t4++) {
                    a[t2-t4][t2+t3-t4] = (a[t2-t4-1][t2-t3-t4-1]+a[t2-t4-1][t2+t3-t4]+a[t2-t4-1][t2+t3-t4-1]+a[t2-t4-1][t2+t3-t4+1])
                }
            }
        }
    }
}
```

- Image of \((t2 - t4, t3 - t2 + t4)\) over an integer set
- Straightforward to accomplish via polyhedral libraries
  - ISL: just create an isl map
  - Polylib: use polylib image function or projections
if ((N >= 3) && (T >= 1)) {
    for (t1=1; t1 <= floord(8*N+9*T−25,8); t1++) {
        lbp=max(ceild(8*t1−T+1,8),ceild(8*t1+1,9));
        ubp=min(floord(8*t1+N+5,9),t1);
        #pragma omp parallel for private (lbv, ubv)
        for (t2=lbv; t2 <= ubp; t2++) {
            for (t3=max(8*t1−7*t2+1,2*t2−N+3); t3 <= min(min(2*t2+N−3,8*t1−7*t2+N+5),t2+N+T−3); t3++) {
                for (t4=max(max(8*t1−8*t2,t2−N+2),−t2+t3−N+2); t4 <= min(min(T−1,t2−1),8*t1−8*t2+7),−t2+t3; t4++) {
                    a[t2−t4][−t2+t3−t4−1] = (a[t2−t4−1][−t2+t3−t4−1]+a[t2−t4−1][−t2+t3−t4]+a[t2−t4−1][−t2+t3−t4+1])
                }
            }
        }
    }
}

- Image of \((t2−t4, t3−t2+t4)\) over an integer set
- Straightforward to accomplish via polyhedral libraries
  - ISL: just create an isl map
  - Polylib: use polylib image function or projections
Computing data accessed – parametric

- But what we are interested in: data accessed for a given $t_1$, $t_2$ for example
- Parametric in $t_1$, $t_2$, $N$
  - Similar approach: don’t eliminate $t_1$, $t_2$ from the system
  - Yields data written to or being read in a given iteration

For previous code, given $t_1$, $t_2$, $N$, we get:

1. $1 \leq d_2 \leq N - 2$
2. $\max(1, 32t_2 - 31) \leq d_1 \leq \min(T - 2, 32t_2 + 31)$
3. $64t_2 - 32t_1 - 31 \leq d_1 \leq 64t_2 - 32t_1 + 31$
4. $-31 \leq 32t_1 - 32t_2 \leq N - 1$

- $d_1$ can be bounded
Computing data accessed – parametric

- But what we are interested in: data accessed for a given \( t_1, t_2 \) for example
- Parametric in \( t_1, t_2, N \)
- Similar approach: don’t eliminate \( t_1, t_2 \) from the system
- Yields data written to or being read in a given iteration

For previous code, given \( t_1, t_2, N \), we get:

\[
1 \leq d_2 \leq N - 2 \\
\max(1, 32t_2 - 31) \leq d_1 \leq \min(T - 2, 32t_2 + 31) \\
64t_2 - 32t_1 - 31 \leq d_1 \leq 64t_2 - 32t_1 + 31 \\
-31 \leq 32t_1 - 32t_2 \leq N - 1
\]

- \( d_1 \) can be bounded
Various approaches from the literature

1. Access function based
2. Dependence based

- Dependence information is already available
- Natural
- May not be the right granularity
- May have to redo the kind of analyses already done for dependences (last write)
- May not be very natural
Various approaches from the literature

1. Access function based
2. Dependence based

1. Amarasinghe and Lam (dep based, PLDI 1993)
2. PIPS / Par4All (?–?)
4. Griebl and Classen (dep based, IPDPS 2006)
5. Pluto distmem – Bondhugula et al. (dep based, 2011)
Amarasinghe and Lam

- First comprehensive approach
- Uses last writer tree (Maydan et al. POPL 1993)

Limitations
- Not a polyhedral approach (restricted to perfect nests)
- Duplicate communication due to simple virtual to physical processor model
Amarasinghe and Lam

- First comprehensive approach
- Uses last writer tree (Maydan et al. POPL 1993)
- Limitations
  - Not a polyhedral approach (restricted to perfect nests)
  - Duplicate communication due to simple virtual to physical processor model
dHPF

- Access function based: looks at writes and remote read accesses to determine communication data
- Coupled with owner computes rule
- Used Omega

- Not fully automatic
- Limitations
  - Seem to have forgotten about last writer
  - Leads to redundant communication for one-to-many style dependences (with parametric problem sizes or number of processors)
  - Lot of redundant communication when receivers for a communication set change across outer sequential loop iterations
program jacobi
  integer N, m
  PARAMETER (N=10000)
  double precision a(N, N), b(N, N)

!HPF$ processors p(32,1)
!HPF$ template t(N,N)
!HPF$ align a(i,j) with t(i,j)
!HPF$ align b(i,j) with t(i,j)
!HPF$ distribute t(block,block) onto p

C      --  Initializations  --

C      --  Jacobi  --
  do time = 1, 1000
    !HPF$ INDEPENDENT
    do j = 2, N - 1
      !HPF$ INDEPENDENT
      do i = 2, N - 1
        a(i, j) = 0.25 * (b(i - 1, j) + b(i + 1, j) + b(i, j - 1) +
          b(i, j + 1))
      enddo
    enddo
  do j = 2, N - 1
    do i = 2, N - 1
      b(i, j) = a(i, j)
    enddo
  enddo
Construct communication polytopes per dependence [IPDPS 2006]

Use a straightforward virtual to physical processor mapping

Strengths and Limitations

- First dependence-based polyhedral approach for code generation
- Duplication of data across dependences
- Preliminary and conceptual
Other non-polyhedral approaches

- Too many in the literature to name!
- Quite brittle in the presence of transformations applied prior to code generation
- In handling imperfect loop nests or non-uniform dependences
Flow dependences lead to communication (anti and output dependences do not)

The **flow-out** set of a tile is the set of all values that are written to inside the tile, and then next read from outside the tile.

The **write-out** set of a tile is the set of all those data elements to which the last write access across the entire iteration space is performed in the tile.

Construct flow-out sets using flow dependences.
Flow dependences lead to communication (anti and output dependences do not)

The **flow-out** set of a tile is the set of all values that are written to inside the tile, and then next read from outside the tile

The **write-out** set of a tile is the set of all those data elements to which the last write access across the entire iteration space is performed in the tile

Construct flow-out sets using flow dependences
Flow dependences lead to communication (anti and output dependences do not)

The **flow-out** set of a tile is the set of all values that are written to inside the tile, and then next read from outside the tile.

The **write-out** set of a tile is the set of all those data elements to which the last write access across the entire iteration space is performed in the tile.

Construct flow-out sets using flow dependences.
Flow-out set

\[
\text{for (}t=1; t<=T-1; t++\text{)} \\
\text{for (}j=1; j<=N-1; j++\text{)} \\
u[t\%2][j] = 0.333 \times (u[(t-1)%2][j-1] + u[(t-1)%2][j] + u[(t-1)%2][j+1]);
\]
for (t=1; t<=T-1; t++)
    for (j=1; j<=N-1; j++)
        u[t%2][j] = 0.333*(u[(t-1)%2][j-1] + u[(t-1)%2][j] + u[(t-1)%2][j+1]);
Computing flow-out set for variable $x$

**Input** Depth of parallel loop: $l$; set $S_w$ of $\langle$write access, statement$\rangle$ pairs for variable $x$

1: $F_{out}^x = \emptyset$

2: for each $\langle M_w, S_i \rangle \in S_w$ do

3: for each dependence $e(S_i \rightarrow S_j) \in E$ do

4: if $e$ is of type RAW and source access of $e$ is $M_w$ then

5: $E_l = \left\{ t_1^i = t_1^j \land t_2^i = t_2^j \land \ldots \land t_l^i = t_l^j \right\}$

6: $C_e^t = D_e^T \cap E_l$

7: $I_{e}^t = \text{project\_out} \left( C_e^t, m_{S_i} + 1, m_{S_j} \right)$

8: $O_{e}^t = \text{project\_out} \left( D_e^T, m_{S_i} + 1, m_{S_j} \right) \setminus I_{e}^t$

9: $F_{out}^x = F_{out}^x \cup I_p(M_w^{S_i}, O_{e}^t, l)$

10: end if

11: end for

12: end for

**Output** $F_{out}^x$
Can use a compiler-assisted dynamic scheme

Define two functions as part of the output code for each data variable, \( x \), that can be a multidimensional array or a scalar. If \( t_1, \ldots, t_l \) is the set of sequential dimensions surrounding parallel dimension \( t_p \):

1. \( \pi(t_1, t_2, \ldots, t_l, t_p) \): rank of processor that executes \((t_1, t_2, \ldots, t_l, t_p)\)

2. \( \sigma_x(t_1, t_2, \ldots, t_l, t_p) \): set of processors that need the flow-out set for data variable \( x \) from the processor calling this function
Determining communication partners

1. Can use a compiler-assisted dynamic scheme
   Define two functions as part of the output code for each data variable, $x$, that can be a multidimensional array or a scalar. If $t_1, \ldots, t_l$ is the set of sequential dimensions surrounding parallel dimension $t_p$:

   - $\pi(t_1, t_2, \ldots, t_l, t_p)$: rank of processor that executes $(t_1, t_2, \ldots, t_l, t_p)$

   - $\sigma_x(t_1, t_2, \ldots, t_l, t_p)$: set of processors that need the flow-out set for data variable $x$ from the processor calling this function
Determining Communication Partners Precisely

\[ \sigma_x(t_1, t_2, \ldots, t_l, t_p) = \{ \pi(t'_1, t'_2, \ldots, t'_l, t'_p) \mid \exists e \in E \text{ on } x, \]
\[ D^T_e(t_1, \ldots, t_p, \ldots, t'_1, \ldots, t'_p, \ldots, \vec{p}, 1) \} \]

\( D^T_e \) is the dependence polyhedron corresponding to \( e \)
Constructing $\sigma$

Constructing $\sigma$ for each variable $x$:

1. For each relevant RAW dependence polyhedron in the transformed space, eliminate all dimensions inner to $t_p$
2. Scan dependence polyhedron to generate loops for target iterators while treating source iterators as parameters
3. Running the generated loop nest at runtime will enumerate all dependent tiles, $(t'_1, t'_2, ..., t'_l, t'_p)$, given coordinates of the source tile
Constructing $\sigma$ for each variable $x$:

1. For each relevant RAW dependence polyhedron in the transformed space, eliminate all dimensions inner to $t_p$

2. Scan dependence polyhedron to generate loops for target iterators while treating source iterators as parameters

3. Running the generated loop nest at runtime will enumerate all dependent tiles, $(t'_1, t'_2, ..., t'_l, t'_p)$, given coordinates of the source tile
Constructing $\sigma$ for each variable $x$:

1. For each relevant RAW dependence polyhedron in the transformed space, eliminate all dimensions inner to $t_p$
2. Scan dependence polyhedron to generate loops for target iterators while treating source iterators as parameters
3. Running the generated loop nest at runtime will enumerate all dependent tiles, $(t'_1, t'_2, ..., t'_i, t'_p)$, given coordinates of the source tile
Strengths and Limitations

- Good for broadcast or multicast style communication
- A processor will never receive the same data twice
- Not good for disjoint point-to-point communication
- A processor could be sent data that it does not need

\[ \text{FO}(\text{ST}) \text{ is sent to } \{\pi(\text{RT}_1) \cup \pi(\text{RT}_2) \cup \pi(\text{RT}_3)\} \]
Strengths and Limitations

- Good for broadcast or multicast style communication
- A processor will never receive the same data twice
- Not good for disjoint point-to-point communication
- A processor could be sent data that it does not need

\[ \text{FO(ST)} \text{ is sent to } \{\pi(RT_1) \cup \pi(RT_2) \cup \pi(RT_3)\} \]
Flow-out intersection flow-in

- Analogous to flow-out set, define flow-in set
- Intersect flow-out set with flow-in set to determine data to be sent from tile to another
  - Could lead to duplication if receiving tiles requiring same data are mapped to the same processor
  - Not good if there are too many tiles but few processors

\[
F_1 = \text{FO}(\text{ST}) \cap \text{FI}(\text{RT}_1) \text{ is sent to } \pi(\text{RT}_1),
\]
\[
F_2 = \text{FO}(\text{ST}) \cap \text{FI}(\text{RT}_2) \text{ is sent to } \pi(\text{RT}_2),
\]
\[
F_3 = \text{FO}(\text{ST}) \cap \text{FI}(\text{RT}_3) \text{ is sent to } \pi(\text{RT}_3)
\]
Flow-out intersection flow-in

- Analogous to flow-out set, define flow-in set
- Intersect flow-out set with flow-in set to determine data to be sent from tile to another
  - Could lead to duplication if receiving tiles requiring same data are mapped to the same processor
  - Not good if there are too many tiles but few processors

\[
F_1 = FO(ST) \cap FI(RT_1) \text{ is sent to } \pi(RT_1), \\
F_2 = FO(ST) \cap FI(RT_2) \text{ is sent to } \pi(RT_2), \\
F_3 = FO(ST) \cap FI(RT_3) \text{ is sent to } \pi(RT_3)
\]
Flow-out set partitioning

A better approach: partition flow-out set and combine benefits of flow-out and flow-out-intersect-flow-in approach

- Check “one-to-one ness” of dependences
- Check if flow-in set is independent of parallel dimension
- Can also decide at runtime
- Real problem arises when two receiving tiles require common data and they are mapped in an unknown way
Flow-out set partitioning

A better approach: partition flow-out set and combine benefits of flow-out and flow-out-intersect-flow-in approach

- Check “one-to-one ness” of dependences
- Check if flow-in set is independent of parallel dimension
- Can also decide at runtime
- Real problem arises when two receiving tiles require common data and they are mapped in an unknown way
Flow-out set partitioning

A better approach: partition flow-out set and combine benefits of flow-out and flow-out-intersect-flow-in approach

- Check “one-to-one ness” of dependences
- Check if flow-in set is independent of parallel dimension
- Can also decide at runtime

- Real problem arises when two receiving tiles require common data and they are mapped in an unknown way
Flow-out set partitioning

A better approach: partition flow-out set and combine benefits of flow-out and flow-out-intersect-flow-in approach

- Check “one-to-one ness” of dependences
- Check if flow-in set is independent of parallel dimension
- Can also decide at runtime
- Real problem arises when two receiving tiles require common data and they are mapped in an unknown way
Packing and unpacking data

- Similar to GPU scratchpad management (Baskaran et al., Armin et al.) but no one needs to index data to be sent out
- Use a linearized counted buffer

```plaintext
for (d0=max(max(1,32*t1−32*t3),32*t3−N+32);
    d0<=min(T−2,32*t1−32*t3+30);d0++) for
    d1=max(1,32*t3−d0+30);d1<=min(N−2,32*t3−d0+31);d1++) {
        send_buf_u[send_count_u++] = u[d0][d1];

        if (t1 <= min(floord(32*t3+T−33,32),2*t3−1)) {
            for (d1=−32*t1+64*t3−31;d1<=min(N−1,−32*t1+64*t3);d1++)
                send_buf_u[send_count_u++] = u[32*t1−32*t3+31][d1];
        }
    }
```

- Unpacking – just reverse the assignment
Packing and unpacking data

- Similar to GPU scratchpad management (Baskaran et al., Armin et al.) but no one needs to index data to be sent out

- Use a linearized counted buffer

```c
for (d0=max(max(1,32*t1−32*t3),32*t3−N+32);
    d0<=min(T−2,32*t1−32*t3+30);d0++) for
    d1=max(1,32*t3−d0+30);d1<=min(N−2,32*t3−d0+31);d1++) {
    send_buf_u[send_count_u++] = u[d0][d1];

    if (t1 <= min(floord(32*t3+T−33,32),2*t3−1)) {
        for (d1=−32*t1+64*t3−31;d1<=min(N−1,−32*t1+64*t3);d1++)
            send_buf_u[send_count_u++] = u[32*t1−32*t3+31][d1];
    }
}
```

- Unpacking – just reverse the assignment
Packing and unpacking data

- Similar to GPU scratchpad management (Baskaran et al., Armin et al.) but no one needs to index data to be sent out
- Use a linearized counted buffer

```c
for (d0=max(max(1,32*t1−32*t3),32*t3−N+32);
    d0<=min(T−2,32*t1−32*t3+30);d0++) for
    d1=max(1,32*t3−d0+30);d1<=min(N−2,32*t3−d0+31);d1++) {
    send_buf_u[send_count_u++]= u[d0][d1];
    if (t1 <= min(floord(32*t3+T−33,32),2*t3−1)) {
        for (d1=−32*t1+64*t3−31;d1<=min(N−1,−32*t1+64*t3);d1++)
            send_buf_u[send_count_u++]= u[32*t1−32*t3+31][d1];
    }
}
```

- Unpacking – just reverse the assignment
Packing and unpacking data

- **unicast-pack**: each receiving iteration belongs to a different compute device (use with foifi)
- **multicast-pack**: all receiving iterations belong to the same compute device (use with flow-out)
- If you know receiving iterations belong to different processors, use unicast pack with foifi
- If you know they are the same processor, use multicast pack with flow-out
- Choice can also be made at runtime
Packing and unpacking data

- **unicast-pack**: each receiving iteration belongs to a different compute device (use with foifi)
- **multicast-pack**: all receiving iterations belong to the same compute device (use with flow-out)
- If you know receiving iterations belong to different processors, use unicast pack with *foifi*
- If you know they are the same processor, use multicast pack with *flow-out*
- Choice can also be made at runtime
Floyd-Warshall example

Use to compute all-pairs shortest-paths in a directed graph

```c
for (k=0; k < N; k++) {
    for (y=0; y < N; y++) {
        for (x=0; x < N; x++) {
            pathDistanceMatrix[y][x] = min(pathDistanceMatrix[y][k] + pathDistanceMatrix[k][x], pathDistanceMatrix[y][x]);
        }
    }
}
```

**Figure:** Floyd-warshall algorithm
Question: Does the pivot row and column itself get updated?

- Not if the edge weights are non-negative
- So, the inner two loops are parallel
Question: Does the pivot row and column itself get updated?

- Not if the edge weights are non-negative
- So, the inner two loops are parallel
Question: Does the pivot row and column itself get updated?

- Not if the edge weights are non-negative
- So, the inner two loops are parallel
Floyd-Warshall communication pattern

Figure: Communication for Floyd-Warshall: at outer loop iteration $k - 1$, processor(s) updating the $k^{th}$ row and $k^{th}$ column broadcast them to processors along their column and row respectively.
Floyd-Warshall results on distributed memory

- FOP
- FOIFI
- FO
- hand-opt UPC / MPI

Graph showing speedup vs. number of nodes.
Heterogeneous architectures

A NODE WITHIN A CLUSTER

- North Bridge
- South Bridge
- DDR RAM
Communication problem manifests as data movement problem

- Have to use multiple libraries to express data movement communication (CUDA-direct, OpenCL, MPI)
## Scaling on multiple GPUs

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Device combination</th>
<th>Total execution time</th>
<th>Total communication volume</th>
<th>Speedup</th>
<th>FOP</th>
<th>FO</th>
<th>Reduction</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>FOP</td>
<td>FO</td>
<td>FOP</td>
<td>FO</td>
<td>Reduction</td>
<td></td>
</tr>
<tr>
<td><strong>floyd</strong></td>
<td>1 CPU (12 cores)</td>
<td>890s</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td></td>
</tr>
<tr>
<td></td>
<td>1 GPU</td>
<td>113s</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td></td>
</tr>
<tr>
<td></td>
<td>1 CPU + 1 GPU</td>
<td>–</td>
<td>148s</td>
<td>180s</td>
<td>1.22</td>
<td>0.8 GB</td>
<td>25.0 GB</td>
</tr>
<tr>
<td></td>
<td>2 GPUs</td>
<td>–</td>
<td>65s</td>
<td>104s</td>
<td>1.60</td>
<td>1.6 GB</td>
<td>51.0 GB</td>
</tr>
<tr>
<td></td>
<td>4 GPUs</td>
<td>–</td>
<td>43s</td>
<td>107s</td>
<td>2.49</td>
<td>3.1 GB</td>
<td>102.0 GB</td>
</tr>
<tr>
<td><strong>heat-2d</strong></td>
<td>1 CPU (12 cores)</td>
<td>1112s</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td></td>
</tr>
<tr>
<td></td>
<td>1 GPU</td>
<td>266s</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td></td>
</tr>
<tr>
<td></td>
<td>1 CPU + 1 GPU</td>
<td>–</td>
<td>242s</td>
<td>255s</td>
<td>1.05</td>
<td>0.6 GB</td>
<td>21.0 GB</td>
</tr>
<tr>
<td></td>
<td>2 GPUs</td>
<td>–</td>
<td>138s</td>
<td>157s</td>
<td>1.14</td>
<td>0.6 GB</td>
<td>21.0 GB</td>
</tr>
<tr>
<td></td>
<td>4 GPUs</td>
<td>–</td>
<td>80s</td>
<td>124s</td>
<td>1.55</td>
<td>1.9 GB</td>
<td>62.0 GB</td>
</tr>
</tbody>
</table>

**Table:** Results on a heterogeneous system
Compiler-assisted scheduling

- Using hyperplanes for scheduling is a limitation
- Compiler-assisted dynamic scheduling can give very good improvement on multicores [Baskaran et al. PPoPP 2009]
Compiler-assisted scheduling

- Using hyperplanes for scheduling is a limitation
- Compiler-assisted dynamic scheduling can give very good improvement on multicores [Baskaran et al. PPoPP 2009]
Compiler-assisted scheduling

- Dependences between scheduling units are available from compiler
- Number of scheduling units may only be known dynamically
- Execution time of scheduling units may only be known dynamically

![Diagram showing scheduling units and their dependencies](image)
Compiler-assisted scheduling

- Dependences between scheduling units are available from compiler
- Number of scheduling units may only be known dynamically
- Execution time of scheduling units may only be known dynamically
Compiler-assisted scheduling

- Dependences between scheduling units are available from compiler
- Number of scheduling units may only be known dynamically
- Execution time of scheduling units may only be known dynamically
Open problems

- Load balancing, dynamic scheduling, and point-to-point synchronization are important
- Runtime support is crucial
- Building polyhedral runtimes – lot of new runtimes being developed (StarPU, CnC, OCR)
Open problems

- Load balancing, dynamic scheduling, and point-to-point synchronization are important.
- Runtime support is crucial.
- Building polyhedral runtimes – lot of new runtimes being developed (StarPU, CnC, OCR).
Tool demo
Memory Optimization

Alias, Bondhugula, Darte

First Spring School on Polyhedral Code Analysis and Optimizations
St Germain au Mont d’Or
May 13–17, 2013
1 Introduction and Motivation
1 Introduction and Motivation
Memory optimization in a polyhedral optimizer

1. Extracting a polyhedral representation
2. Dependence analysis
3. Transformation and parallelization
4. Memory optimization
5. Code generation (out of polyhedral extraction)
Memory optimization in a polyhedral optimizer

1. Extracting a polyhedral representation
2. Dependence analysis
3. Transformation and parallelization
4. Memory optimization
5. Code generation (out of polyhedral extraction)

- It is schedule dependent!
Motivation

- Smaller memory footprint ⇒
  - better cache performance
  - More applications can fit in memory

- Critical for embedded systems (limited memory)
  - Language may not specify memory semantics
    - Domain-specific languages
    - Data flow languages
  - A naive translation to imperative code may lead to too much storage
Introduction and Motivation

**Motivation**

- Smaller memory footprint $\Rightarrow$
  - better cache performance
  - More applications can fit in memory
- Critical for embedded systems (limited memory)
- Language may not specify memory semantics
  - Domain-specific languages
  - Data flow languages
- A naive translation to imperative code may lead to too much storage
Common transformations that can affect memory optimization

- Loop fusion, permutation, tiling, parallelization

Does performing memory optimization add new dependences?
- Yes, but only WAR and WAW dependences
- Does not affect “data flow”
- Each read still reads from the same write
Transformations and memory optimization

- **Common transformations that can affect memory optimization**
  - Loop fusion, permutation, tiling, parallelization

- **Does performing memory optimization add new dependences?**
  - Yes, but only WAR and WAW dependences
  - Does not affect “data flow”
  - Each read still reads from the same write
Common transformations that can affect memory optimization
- Loop fusion, permutation, tiling, parallelization

Does performing memory optimization add new dependences?
- Yes, but only WAR and WAW dependences
- Does not affect “data flow”
- Each read still reads from the same write
Transformations and memory optimization

- Common transformations that can affect memory optimization
  - Loop fusion, permutation, tiling, parallelization
- Does performing memory optimization add new dependences?
  - Yes, but only WAR and WAW dependences
  - Does not affect “data flow”
  - Each read still reads from the same write
Introduction and Motivation

An example

for (t=0; t<=T−1; t++)
for (i=1; i<=N−2; i++)
a[t][i] = a[t][i−1] + a[t−1][i] + a[t−1][i+1];

Life times of a[t][i],
a[t+1][i] do not overlap

for (t=0; t<=T−1; t++)
for (i=1; i<=N−2; i++)
a[t][i] = a[t−1][i] + a[t−1][i−1] + a[t−1][i+1];

Life times of a[t][i],
a[t+1][i] overlap, but a[t][i],
a[t+2][i] do not

for (t=0; t<=T−1; t++)
for (i=1; i<=N−2; i++)
a[t%2][i] = a[(t−1) % 2][i] + a[(t−1) % 2][i−1] + a[(t−1) % 2][i+1];

Memory reduction: $T \times N$
to $N$

Reduction: $T \times N$ to $2N$
An example

for (t=0; t<=T-1; t++)
    for (i=1; i<=N-2; i++)
        a[t][i] = a[t][i-1] + a[t-1][i] + a[t-1][i+1];

Life times of a[t][i], a[t+1][i] do not overlap

for (t=0; t<=T-1; t++)
    for (i=1; i<=N-2; i++)
        a[t][i] = a[t-1][i] + a[t-1][i-1] + a[t-1][i+1];

Life times of a[t][i], a[t+1][i+1] do not overlap

- Can allocate along (1,-1): a[t][i] → buf[t - i + N]
- Reduction: T * N to N + T

Memory reduction: T * N to N
Introduction and Motivation

A LabVIEW program
for (t1 = 0; t1 < N; ++t1)
    for (t2 = 0; t2 < N; ++t2)
        c1[t1][t2] = c[t1][t2];

for (int i = 0; i < N; ++i) {
    for (t1 = 0; t1 < N; ++t1)
        c2[t1][t2] = c1[t1][t2];
    for (int j = 0; j < N; ++j) {
        // copy c2 to c3 as data flows through shift register
        for (t1 = 0; t1 < N; ++t1)
            c3[t1][t2] = c2[t1][t2];
        for (int k = 0; k < N; ++k) {
            float r1 = a[i][k]; // array read node
            float r2 = b[k][j]; // array read node
            float r3 = c3[i][j]; // array read node
            float r4 = r1 * r2; // multiply primitive
            float r5 = r3 + r4; // add primitive
        }
        // copy c3 to c4
        for (t1 = 0; t1 < N; ++t1) {
            for (t2 = 0; t2 < N; ++t2) {
                c4[t1][t2] = c3[t1][t2];
            }
        }
        c4[i][j] = r5; // array write node
    }
}

// copy back c4 to c3 as data flows into right shift register
for (t1 = 0; t1 < N; ++t1)
    for (t2 = 0; t2 < N; ++t2)
        c3[t1][t2] = c4[t1][t2];

// j loop
for (t1 = 0; t1 < N; ++t1)
    for (t2 = 0; t2 < N; ++t2)
        c2[t1][t2] = c3[t1][t2];

// i loop
// copy c1 to c_out
for (t1 = 0; t1 < N; ++t1)
    for (t2 = 0; t2 < N; ++t2)
        c_out[t1][t2] = c1[t1][t2];
for (i = 0; i < N; ++i){
    for (j = 0; j < N; ++j){
        for (k = 0; k < N; ++k){
            c[i][j] += a[i][k] * b[k][j];
        }
    }
}
Array Contraction with Lattice-Based Memory Allocation

Alain Darte

CNRS, Compsys
Laboratoire de l’Informatique du Parallélisme
École normale supérieure de Lyon

May 13-17, 2013, St Germain au Mont d’Or
Outline

1 Introduction
   - References
   - Example of usages

2 Main algorithms
   - De Greef, Catthoor, De Man
   - Lefebvre and Feautrier
   - Quilleré and Rajopadhye

3 Lattice-based memory allocation
   - Modulo allocations
   - A critical lattice problem
   - Known results on critical (integer) lattices
Introduction
Main algorithms
Lattice-based memory allocation
References
Example of usages

Some references on static memory optimization schemes

Early work on parallel memories


Sliding windows

- Bouchebaba and Coelho (2002). Tiling and memory reuse of sequences of nested loops.

Parallel languages & scratch-pad memory optimizations

Some references on static memory optimization schemes

Schedule-independent mapping


Polyhedral in-place array reuse

- Alias, Baray, Darte (2007). Bee+Cl@k: an implementation of lattice-based array contraction in the source-to-source translator Rose.
Array contraction in sequential programs

Simpler situation: no reuse, just a storage problem

```plaintext
for(I, I in P) {
    /* P is a multidimensional polytope */
    A[I] = ...
}
for(I, I in P) {
    ... = A[I]
}
```

How to map A in memory to store only a portion of it?

1D array cells reuse: similar to rotating registers

```plaintext
for(i=2, i<N, i++) {
    a[i] = a[i-1] + a[i-2];
}
for(i=2, i<N, i++) {
    a[i mod 2] += a[i-1 mod 2];
}
```
Intermediate buffer for pipelined DCT-like example

Two synchronized, pipelined (ASAP) processes, communicating through a shared buffer $A$.

```c
for(b_r=0; b_r<64; b_r++) {
    for(b_c=0; b_c<64; b_c++) {
        for(r=0; r<8; r++) {
            A[b_r][b_c][r][*] = ...;
        }
    }
}
```

```c
for(b_r=0; b_r<64; b_r++) {
    for(b_c=0; b_c<64; b_c++) {
        for(c=0; c<8; c++) {
            ... = A[b_r][b_c][*][c];
        }
    }
}
```
Intermediate buffer for pipelined DCT-like example

Two synchronized, pipelined (ASAP) processes, communicating through a shared buffer $A$.

```c
for(b_r=0; b_r<64; b_r++) {
    for(b_c=0; b_c<64; b_c++) {
        for(r=0; r<8; r++) {
            A[b_r][b_c][r][*] = ...;
        }
    }
}
```

```c
for(b_r=0; b_r<64; b_r++) {
    for(b_c=0; b_c<64; b_c++) {
        for(c=0; c<8; c++) {
            ... = A[b_r][b_c][*][c];
        }
    }
}
```

Full array (no reuse) $64 \times 64 \times 8 \times 8 = 2^{18} = 256K$.

Intuitive solution write in $A(b_r \mod 2, b_c \mod 2, r, c)$ (4 blocks)

“Best” linear allocation 112 with $\sigma = \begin{cases} 
  r \mod 4 \\
  16(b_r + b_c) + 2r + c \mod 28 
\end{cases}$
Double-buffering for pipelined tiles with inter-tile reuse

\[
\begin{align*}
\text{for}(i=0; \ i<n; \ i++) & \\
\text{for}(j=0; \ j<n; \ j++) & \\
\quad c[i+j] &= c[i+j] + p[i]*q[j];
\end{align*}
\]

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

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

\textbf{Load} \; \simeq \; \text{first reads} \; \cap \; \text{tile domain.} \; \textbf{Store} \; \simeq \; \text{last writes} \; \cap \; \text{tile domain.}
Double-buffering for pipelined tiles with inter-tile reuse

for (i = 0; i < n; i++)
    for (j = 0; j < n; j++)
        \[ c[i+j] = c[i+j] + p[i]*q[j]; \]

Load \( \simeq \) first reads \( \cap \) tile domain. Store \( \simeq \) last writes \( \cap \) tile domain.
Outline

1 Introduction
   - References
   - Example of usages

2 Main algorithms
   - De Greef, Catthoor, De Man
   - Lefebvre and Feautrier
   - Quilleré and Rajopadhye

3 Lattice-based memory allocation
   - Modulo allocations
   - A critical lattice problem
   - Known results on critical (integer) lattices
• Linearize the time. Ex: $t(i, j) \sim t = Mi + j$. 
De Greef, Catthoor, De Man (1997)

- Linearize the time. Ex: \( t(i,j) \leadsto t = Mi + j \).
- Linearize the addresses. Ex. \( a(i,j) \leadsto a = i - Nj \).
De Greef, Catthoor, De Man (1997)

- Linearize the time. Ex: \( t(i,j) \leadsto t = Mi + j \).
- Linearize the addresses. Ex. \( a(i,j) \leadsto a = i - Nj \).
- Compute the BOATD (\textit{binary occupied address-time domain}) for any pair \( W/R \) of references:

\[
BOATD = \{(a, t) | \text{address } a \text{ is live at time } t\}
\]
Linearize the time. Ex: $t(i,j) \rightsquigarrow t = Mi + j$.
Linearize the addresses. Ex. $a(i,j) \rightsquigarrow a = i - Nj$.
Compute the BOATD (*binary occupied address-time domain*) for any pair W/R of references:

$$BOATD = \{(a, t) \mid \text{address } a \text{ is live at time } t\}$$

Compute the maximal address difference at a given time:

$$b = \max\{|a_1 - a_2| \mid (a_1, t) \in BOATD_1 \text{ and } BOATD_2\}$$
De Gref, Catthoor, De Man (1997)

- Linearize the time. Ex: \( t(i, j) \sim t = Mi + j \).
- Linearize the addresses. Ex. \( a(i, j) \sim a = i - Nj \).
- Compute the BOATD (binary occupied address-time domain) for any pair W/R of references:

\[
BOATD = \{(a, t) \mid \text{address } a \text{ is live at time } t\}
\]

- Compute the maximal address difference at a given time:

\[
b = \max\{|a_1 - a_2| \mid (a_1, t) \in BOATD_1 \text{ and } BOATD_2\}
\]

- Map the array with the function \( A[i, j] \sim a(i, j) \mod (b + 1) \).
De Greet, Catthoor, De Man (1997)

- Linearize the time. Ex: \( t(i, j) \Rightarrow t = Mi + j \).
- Linearize the addresses. Ex. \( a(i, j) \Rightarrow a = i - Nj \).
- Compute the BOATD (binary occupied address-time domain) for any pair W/R of references:

\[
BOATD = \{(a, t) \mid \text{address } a \text{ is live at time } t\}
\]

- Compute the maximal address difference at a given time:

\[
b = \max\{|a_1 - a_2| \mid (a_1, t) \in BOATD_1 \text{ and } BOATD_2\}
\]

- Map the array with the function \( A[i, j] \Rightarrow a(i, j) \mod (b + 1) \).

Not parametric, based on linearizations, final 1D computation.
Multi-dimensional extension: Tronçon et al. (2002).
Simple pipeline example

Sequential execution of each process.
The second one is run one “clock cycle” later.
Reads occur before writes.

```c
for(i=0; i<N; i++) {
    for(j=0; j<N; j++) {
        /* Statement S */
        A[i][j] = ...;
    }
}
```

Schedule $\theta(S, i,j) = Ni + j$.

```c
for(i=0; i<N; i++) {
    for(j=0; j<N; j++) {
        /* Statement T */
        ... = A[i][j];
    }
}
```

Schedule $\theta(T, i,j) = Ni + j + 1$.

💡 Design intermediate buffer for $A$ with memory reuse?
Simultaneously live values.

Warning: the “conflict” is between array cells, or between two iterations of $S$, not between an iteration of $S$ and an iteration of $T$.

- $Ni + j \mod 2$ optimal.
- $i + Nj \mod N(N - 1)$ awful.
- should consider also $Ni - j$ and $i - Nj$.
- Tronçon et al: $2 \times N$. 
Lefebvre and Feautrier (1999)

Special framework.

- Forget about original arrays: \((S, \vec{i})\) will write in \(A_S[\vec{i} \mod \vec{b}]\).
- Exact dependence analysis is used to rewrite the program.
Special framework.

- Forget about original arrays: \((S, \vec{i})\) will write in \(A_S[\vec{i} \mod \vec{b}]\).
- Exact dependence analysis is used to rewrite the program.
- Compute \(D(S)\) lexicomaximum between the write at \((S, \vec{i})\) and its last read w.r.t. a multi-dimensional schedule \(\sigma\).
Special framework.

- Forget about original arrays: \((S, \vec{i})\) will write in \(A_S[\vec{i} \mod \vec{b}]\).
- Exact dependence analysis is used to rewrite the program.
- Compute \(D(S)\) lexico-maximum between the write at \((S, \vec{i})\) and its last read w.r.t. a multi-dimensional schedule \(\sigma\).
- Compute \(C_p\), set of “conflicting” iterations \((\vec{i}, \vec{j})\) at depth \(p\):
  - \([\theta(S, \vec{i}), \theta(S, \vec{i}) + D(S)] \cap [\theta(S, \vec{j}), \theta(S, \vec{j}) + D(S)] \neq \emptyset\).
  - Occur at depth \(p\): \(\vec{i} \prec_p \vec{j}\).
Lefebvre and Feautrier (1999)

Special framework.

- Forget about original arrays: $(S, \vec{i})$ will write in $A_S[\vec{i} \mod \vec{b}]$.
- Exact dependence analysis is used to rewrite the program.
- Compute $D(S)$ lexico-maximum between the write at $(S, \vec{i})$ and its last read w.r.t. a multi-dimensional schedule $\sigma$.
- Compute $C_p$, set of “conflicting” iterations $(\vec{i}, \vec{j})$ at depth $p$:
  - $[\theta(S, \vec{i}), \theta(S, \vec{i}) + D(S)] \cap [\theta(S, \vec{j}), \theta(S, \vec{j}) + D(S)] \neq \emptyset$.
  - Occur at depth $p$: $\vec{i} <_p \vec{j}$.
  - Compute $\vec{b}_p$ as $1 + \text{the maximum of } j_p - i_p$ for $(\vec{i}, \vec{j}) \in C_p$. 

Validity: simple arguments on the number of dimensions.
+ Very robust multi-dimensional view. Parametric.
  - One statement, single assignment, fixed basis, ane schedule.
Special framework.

- Forget about original arrays: \((S, \vec{i})\) will write in \(A_S[\vec{i} \mod \vec{b}]\).
- Exact dependence analysis is used to rewrite the program.
- Compute \(D(S)\) lexicographic maximum between the write at \((S, \vec{i})\) and its last read w.r.t. a multi-dimensional schedule \(\sigma\).
- Compute \(C_p\), set of “conflicting” iterations \((\vec{i}, \vec{j})\) at depth \(p\):
  - \([\theta(S, \vec{i}), \theta(S, \vec{i}) + D(S)] \cap [\theta(S, \vec{j}), \theta(S, \vec{j}) + D(S)] \neq \emptyset\).
  - Occur at depth \(p\): \(\vec{i} \prec_p \vec{j}\).
  - Compute \(\vec{b}_p\) as \(1 + \text{the maximum of } j_p - i_p\) for \((\vec{i}, \vec{j}) \in C_p\).
- Validity: simple arguments on the number of dimensions.
Lefebvre and Feautrier (1999)

Special framework.

- Forget about original arrays: \((S, \vec{i})\) will write in \(A_S[\vec{i} \mod \vec{b}]\).
- Exact dependence analysis is used to rewrite the program.
- Compute \(D(S)\) lexicographic-maximum between the write at \((S, \vec{i})\) and its last read w.r.t. a multi-dimensional schedule \(\sigma\).
- Compute \(C_p\), set of “conflicting” iterations \((\vec{i}, \vec{j})\) at depth \(p\):
  - \([\theta(S, \vec{i}), \theta(S, \vec{i}) + D(S)] \cap [\theta(S, \vec{j}), \theta(S, \vec{j}) + D(S)] \neq \emptyset\).
  - Occur at depth \(p\): \(\vec{i} \prec_p \vec{j}\).
  - Compute \(\vec{b}_p\) as \(1 + \max \{j_p - i_p\} for (i, j) \in C_p\).
- Validity: simple arguments on the number of dimensions.

- One statement, single assignment, fixed basis, affine schedule.
Reason with conflicting differences.

With exact set:

\[
\begin{align*}
\text{then} & : 2 \Rightarrow 4. \\
\text{then} & : N \Rightarrow N.
\end{align*}
\]
Reason with conflicting differences.

With set based on $D(S)$:

- $i$ then $j$: $2$ then $N \leadsto 2N$.
- $j$ then $i$: $N$ then $1 \leadsto N$. 

Note: with skewed schedule, can find $N^2$. 
Reason with conflicting differences.

With exact set:
- $i$ then $j$: 2 then 2 $\rightsquigarrow$ 4.
- $j$ then $i$: $N$ then 1 $\rightsquigarrow$ $N$. 
Reason with conflicting differences.

With exact set:
- \( i \text{ then } j: \ 2 \text{ then } 2 \mapsto 4. \)
- \( j \text{ then } i: \ N \text{ then } 1 \mapsto N. \)

Note: with “skewed schedule”, can find \( N^2 \).
Basically, look at the dimension of $D(S)$ and use projections for missing dimensions.

Main difference/contribution w.r.t. Lefebvre-Feautrier: think in the basis defined by the schedule $\theta$ and not in the initial basis.

 النوبي، سانجا!
Outline

1 Introduction
   - References
   - Example of usages

2 Main algorithms
   - De Greef, Catthoor, De Man
   - Lefebvre and Feautrier
   - Quilleré and Rajopadhye

3 Lattice-based memory allocation
   - Modulo allocations
   - A critical lattice problem
   - Known results on critical (integer) lattices
Memory reuse for scheduled programs

**Given**

- An array $A$ with multiple reads and writes.
- Scheduled program or communicating processes, thanks to $\theta$.

**Goal**

- Reduction of the allocation size (size of buffer).
- Simplicity of the addressing functions.

**Solutions**

- Optimal size with Ehrhart counting \(\approx\) approximations?
- Approximation of maximal number of live values \(\approx\) mapping?
- Modulo mapping \(i \mapsto A_i \mod b\) \(\approx\) simple and quite efficient.
Definitions for memory allocations

**Definition**

A linear allocation of size $m$ is a homomorphism $\sigma : \mathbb{Z}^n \rightarrow G$, where $G \subset \mathbb{Z}^p$ is a finite abelian group of $m$ elements.

Allocations are equivalent if they have the same kernel (renaming).
Definitions for memory allocations

**Definition**

A **linear allocation** of size $m$ is a homomorphism $\sigma : \mathbb{Z}^n \rightarrow G$, where $G \subset \mathbb{Z}^p$ is a finite abelian group of $m$ elements.

Allocations are **equivalent** if they have the same kernel (renaming).

**Definition**

A **modulo allocation** $(M, \vec{b})$, defined by $M \in \mathcal{M}_{p,n}(\mathbb{Z})$ and $\vec{b} \in \mathbb{N}^p$, maps the index $\vec{i}$ to $\sigma(\vec{i}) = M\vec{i} \mod \vec{b}$ (the modulo is applied componentwise) in a $p$-dimensional array of shape $\vec{b}$. 
Definitions for memory allocations

**Definition**

A **linear allocation** of size $m$ is a homomorphism $\sigma : \mathbb{Z}^n \rightarrow G$, where $G \subset \mathbb{Z}^p$ is a finite abelian group of $m$ elements.

Allocations are **equivalent** if they have the same kernel (renaming).

**Definition**

A **modulo allocation** $(M, \vec{b})$, defined by $M \in \mathcal{M}_{p,n}(\mathbb{Z})$ and $\vec{b} \in \mathbb{N}^p$, maps the index $\vec{i}$ to $\sigma(\vec{i}) = M\vec{i} \mod \vec{b}$ (the modulo is applied componentwise) in a $p$-dimensional array of shape $\vec{b}$.

**Property**

Any linear allocation is **equivalent** to a modulo allocation $(U, \vec{s})$ with $U$ unimodular. The equivalent modulo mapping of smallest depth can be found thanks to Smith form.
Definitions for conflict and difference sets CS and DS

**Definition**

Two indices $\vec{i}$ and $\vec{j}$ of $\mathbb{Z}^n$ are **conflicting** ($\vec{i} \mathbin{\triangleleft} \vec{j}$) if they correspond to two simultaneously live values in the schedule $\theta$. 
Definitions for conflict and difference sets CS and DS

**Definition**

Two indices $\vec{i}$ and $\vec{j}$ of $\mathbb{Z}^n$ are **conflicting** ($\vec{i} \triangleleft \vec{j}$) if they correspond to two simultaneously live values in the schedule $\theta$.

**Definition**

A memory allocation is **valid** iff $\vec{i} \triangleleft \vec{j}, \vec{i} \neq \vec{j} \Rightarrow \sigma(\vec{i}) \neq \sigma(\vec{j})$.

🏠 A linear allocation $\sigma$ is valid iff $\text{DS} \cap \ker \sigma = \{\vec{0}\}$.
Definitions for conflict and difference sets CS and DS

**Definition**

Two indices \( \vec{i} \) and \( \vec{j} \) of \( \mathbb{Z}^n \) are **conflicting** \((\vec{i} \bowtie \vec{j})\) if they correspond to two simultaneously live values in the schedule \( \theta \).

**Definition**

A memory allocation is **valid** iff \( \vec{i} \bowtie \vec{j}, \vec{i} \neq \vec{j} \Rightarrow \sigma(\vec{i}) \neq \sigma(\vec{j}). \)

\[\blacktriangleright \text{ A linear allocation } \sigma \text{ is valid iff } DS \cap \ker \sigma = \{0\}.\]

**Property**

An allocation is optimal iff its kernel is a **strictly admissible** (integer) lattice for DS, of minimal determinant, i.e., if it is a **critical** lattice.

From now on, we assume \( DS \subseteq \hat{K}, \text{ where } \hat{K} \) is the set of integer points in a 0-symmetric polytope \( K, \text{ dim Vect}(\hat{K}) = n. \)
Critical and admissible lattices
Critical and admissible lattices

0−Symmetric Polytope: vertices (8,1), (−8,−1), (−1,5), and (1,−5)
Critical and admissible lattices

Lattice: Basis (9,0), (0,6)  Determinant: 54  (i mod 9, j mod 6)
Critical and admissible lattices

Lattice: Basis (9,0), (0,5)  Determinant: 45  (i mod 9, j mod 5)
Critical and admissible lattices

Lattice: Basis (8,0), (6,6)  Determinant: 48  (i–j mod 8, j mod 6)
Critical and admissible lattices

Lattice: Basis (8,0), (4,4)
Determinant: 32
(i–j mod 8, j mod 4)
Critical and admissible lattices

Lattice: Basis (8,0), (3,4)  
Determinant: 32  
4i–3j mod 32
Critical and admissible lattices

Lattice: Basis (7,0), (4,4)  Determinant: 28  (i−j mod 7, j mod 4)

\[
\text{Lattice: Basis (7,0), (4,4) (i−j mod 7, j mod 4)}
\]

\[
\text{Determinant: 28}
\]
### Critical and admissible lattices

<table>
<thead>
<tr>
<th>Critical Lattice: Basis (4,3), (8,0)</th>
<th>Determinant: 24</th>
<th>3i–4j mod 24</th>
</tr>
</thead>
</table>

![Diagram of a critical lattice with basis (4,3), (8,0) and determinant 24, showing points and a vector mod 24.](image-url)
Lattice-based memory allocation: process

1. **Lifetime analysis** of the array elements of $A$, w.r.t. schedule $\theta$.
2. **Relation $\triangleleft$**: Build the polytope of conflicting differences.
3. **Admissible lattice**: Build an admissible $\Lambda$ of small determinant.
4. **Modulo function**: Compute $\sigma = (M, \vec{b})$ such that $\ker \sigma = \Lambda$.
5. **Code generation**: Replace $A(\vec{i})$ with $A(M\vec{i} \mod \vec{b})$.

Some steps can be combined, depending on the heuristic.

Not a perfect scheme, does not reach minimal size, but: robust, expressed in terms of $\theta$, conservative.
Lattice-based memory allocation: process

1. **Lifetime analysis** of the array elements of $A$, w.r.t. schedule $\theta$.
2. **Relation $\triangleleft$**: Build the polytope of conflicting differences.
3. **Admissible lattice**: Build an admissible $\Lambda$ of small determinant.
4. **Modulo function**: Compute $\sigma = (M, \vec{b})$ such that $\ker \sigma = \Lambda$.
5. **Code generation**: Replace $A(i)$ with $A(Mi \mod \vec{b})$.

Some steps can be combined, depending on the heuristic.

✍ Not a perfect scheme, does not reach minimal size, but: robust, expressed in terms of $\theta$, conservative.

**Theoretical questions:**

- Optimal? Lower and upper bounds? Approximations?
- Restriction to subclasses of allocations (e.g., 1D)?
- Rounding of rational lattices? Parameterized solutions?
Optimal solutions

- A lattice is uniquely determined by the Hermite form $H$ of a basis $A$: $A = HQ$ with $Q$ unimodular, $H$ lower triangular, with $0 \leq H_{i,j} < H_{i,i}$.
- Generate all such triangular matrices with given determinant $d$.
- For each, check with integer linear programming that $H\vec{x} \in K$, $\vec{x} \in \mathbb{Z}^n$, has no solution other than $\vec{0}$.

Increase $d$ until you get a solution.
Optimal solutions

- A lattice is uniquely determined by the **Hermite form** $H$ of a basis $A$: $A = HQ$ with $Q$ unimodular, $H$ lower triangular, with $0 \leq H_{i,j} < H_{i,i}$.

- Generate all such triangular matrices with given determinant $d$.

- For each, check with **integer linear programming** that $H\vec{x} \in K$, $\vec{x} \in \mathbb{Z}^n$, has no solution other than $\vec{0}$.

Increase $d$ until you get a solution.

- Number of lattices $H_n(d)$ is well known.

- For the DCT example from $d = 1$ to $d = 112$, there are 86,416,644 lattices to check in 4D (two days), and 941,901 lattices in 3D (30 minutes).

- Is there a faster method?
Critical determinant $\Delta(K)$

$\Delta(K) = \inf\{\det(\Lambda) \mid \Lambda \text{ lattice strictly admissible for } K\}$.

Minkowski’s first theorem + Minkowski-Hlawka theorem:

$\frac{\text{Vol}(K)}{2^n} \leq \Delta(K) \leq \text{Vol}(K)$
Critical determinant \( \Delta(K) \)

\[ \Delta(K) = \inf \{ \det(\Lambda) \mid \Lambda \text{ lattice strictly admissible for } K \} . \]

Minkowski’s first theorem + Minkowski-Hlawka theorem:

\[
\frac{\text{Vol}(K)}{2^n} \leq \Delta(K) \leq \text{Vol}(K)
\]

Constructive upper bound (Rogers):

\[ \Delta(K) \left( \prod_{i=1}^{n} \lambda_i(K) \right) \leq 2^{\frac{n-1}{2}} \Rightarrow \Delta(K) \leq \left( \frac{n!}{2^{\frac{n+1}{2}}} \right) \text{Vol}(K) \]

thanks to Minkowski’s second theorem:

\[ \text{Vol}(K) \left( \prod_{i=1}^{n} \lambda_i(K) \right) \geq 2^n / n! \]

where \( \lambda_i(K) \) is the \( i \)-th successive minimum of \( K \):

\[ \lambda_i(K) = \inf \{ \lambda > 0 \mid \dim(\text{Vect}(\lambda K \cap \mathbb{Z}^n)) \geq i \} \]
Integer critical determinant $\Delta_{\mathbb{Z}}(K)$

$\Delta_{\mathbb{Z}}(K) = \inf \{ \det(\Lambda) \mid \Lambda \text{ integer lattice strictly admissible for } K \}$. 

- $\Delta(\lambda K) = \lambda^n \Delta(K)$ but $\Delta_{\mathbb{Z}}(\lambda K) \leq \lceil \lambda \rceil^n \Delta_{\mathbb{Z}}(K)$. 
Integer critical determinant $\Delta_{\mathbb{Z}}(K)$

\[ \Delta_{\mathbb{Z}}(K) = \inf \{ \det(\Lambda) \mid \Lambda \text{ integer lattice strictly admissible for } K \} . \]

- $\Delta(\lambda K) = \lambda^n \Delta(K)$ but $\Delta_{\mathbb{Z}}(\lambda K) \leq \lceil \lambda \rceil^n \Delta_{\mathbb{Z}}(K)$.

- Rogers’ mechanism adapted to integer lattices:
  \[ \Delta_{\mathbb{Z}}(K) \leq c_n \text{Vol}(K) \text{ with } c_n = n! \text{ instead of } n!/2^{(n+1)/2}. \]
Integer critical determinant $\Delta_{\mathbb{Z}}(K)$

$\Delta_{\mathbb{Z}}(K) = \inf \{ \det(\Lambda) \mid \Lambda \text{ integer lattice strictly admissible for } K \}$.

- $\Delta(\lambda K) = \lambda^n \Delta(K)$ but $\Delta_{\mathbb{Z}}(\lambda K) \leq \lceil \lambda \rceil^n \Delta_{\mathbb{Z}}(K)$.
- Rogers’ mechanism adapted to integer lattices:
  \[ \Delta_{\mathbb{Z}}(K) \leq c_n \text{Vol}(K) \text{ with } c_n = n! \text{ instead of } n!/2^{(n+1)/2}. \]
- Heuristic based on LLL: $c_n = 2^{n(n+3)/4}(n+1)^{n/2}$. 
Integer critical determinant $\Delta_{\mathbb{Z}}(K)$

$\Delta_{\mathbb{Z}}(K) = \inf \{ \det(\Lambda) \mid \Lambda \text{ integer lattice strictly admissible for } K \}$.

- $\Delta(\lambda K) = \lambda^n \Delta(K)$ but $\Delta_{\mathbb{Z}}(\lambda K) \leq \lceil \lambda \rceil^n \Delta_{\mathbb{Z}}(K)$.
- Rogers’ mechanism adapted to integer lattices:
  $\Delta_{\mathbb{Z}}(K) \leq c_n \text{Vol}(K)$ with $c_n = n!$ instead of $n!/2^{(n+1)/2}$.
- Heuristic based on LLL: $c_n = 2^{n(n+3)/4} (n+1)n/2$.
- Heuristics based on gauge functions $F_i$ and $F_i^*$ for special basis:
  
  - $F_i$ and successive minima $c_n = (n!)^2$
  - $F_i$ and generalized basis reduction $c_n = 2^{n^2} n!$
  - $F_i^*$ and successive minima $c_n = (n!)^2$
  - $F_i^*$ and Korkine-Zolotarev $c_n = (n!)^2$

Very large upper bounds in the worse cases.
Gauge functions and projections (Lovász and Scarf, 1992):

- \( F(\vec{x}) = \inf\{ \lambda > 0 \mid \vec{x} \in \lambda K \} \) defines a distance function such that \( F(\alpha \vec{x}) = |\alpha| F(\vec{x}) \).

- For a basis \((\vec{a}_i)_{1 \leq i \leq n}\) of \( \mathbb{Z}^n \), define the gauge function \( F_i(\vec{x}) = \inf\{ F(\vec{y}) \mid \vec{y} \in \vec{x} + \text{Vect}(\vec{a}_1, \ldots, \vec{a}_{i-1}) \} \).

- Define \( F_i^* \) for polar reciprocal \( K^* = \{ \vec{y} \mid \vec{x}.\vec{y} \leq 1, \forall \vec{x} \in K \} \).
Gauge functions and projections (Lovász and Scarf, 1992):

- \( F(\vec{x}) = \inf \{ \lambda > 0 \mid \vec{x} \in \lambda K \} \) defines a distance function such that \( F(\alpha \vec{x}) = |\alpha| F(\vec{x}) \).
- For a basis \((\vec{a}_i)_{1 \leq i \leq n}\) of \( \mathbb{Z}^n \), define the gauge function \( F_i(\vec{x}) = \inf \{ F(\vec{y}) \mid \vec{y} \in \vec{x} + \text{Vect}(\vec{a}_1, \ldots, \vec{a}_{i-1}) \} \).
- Define \( F_i^* \) for polar reciprocal \( K^* = \{ \vec{y} \mid \vec{x} \cdot \vec{y} \leq 1, \forall \vec{x} \in K \} \).

Properties:

- \( F_i \) and \( F_i^* \) can be computed by linear programming.
- \( \prod_{1 \leq i \leq n} F_i(\vec{a}_i) \text{Vol}(K) \leq 2^n \) for any basis \((\vec{a}_i)_{1 \leq i \leq n}\).
- If \((\vec{c}_{n-i+1})_{1 \leq i \leq n}\) is the dual basis of \((\vec{a}_i)_{1 \leq i \leq n}\) then \( F_i(\vec{a}_i) F_{n-i+1}^*(\vec{c}_{n-i+1}) = 1 \) the two visions are linked.
Rogers’ mechanism

- Choose $\rho_i \in \mathbb{N}$, $1 \leq i \leq n$, $\rho_i$ multiple of $\rho_{i+1}$ for $i < n$, and $\dim(L_i) \leq i - 1$, where $L_i = \text{Vect}(K / \rho_i \cap \mathbb{Z}^n)$ (thus $\rho_i \lambda_i(K) > 1$).
- Choose a basis $(\vec{a}_i)_{1 \leq i \leq n}$ of $\mathbb{Z}^n$ s.t. $L_i \subseteq \text{Vect}(\vec{a}_1, \ldots, \vec{a}_{i-1})$.
- Define $\Lambda$ as the lattice generated by the vectors $(\rho_i \vec{a}_i)_{1 \leq i \leq n}$.

Implementation involves computing the successive minima of $K$. 
Heuristic based on $F_i$

1. Choose a basis $(\tilde{a}_i)_{1 \leq i \leq n}$ of $\mathbb{Z}^n$.
2. Compute $F_i(\tilde{a}_i) = \inf \{ F(\tilde{y}) \mid \tilde{y} \in \tilde{a}_i + \text{Vect}(\tilde{a}_1, \ldots, \tilde{a}_{i-1}) \}$.
3. Choose $n$ integers $\rho_i$ such that $\rho_i F_i(\tilde{a}_i) > 1$.
4. Define $\Lambda$ as the lattice generated by the vectors $(\rho_i \tilde{a}_i)_{1 \leq i \leq n}$.

Implementation involves solving linear programming problems.
Heuristic based on $F_i^*$: Lefebvre-Feautrier!

- Choose a basis $(\vec{c}_i)_{1 \leq i \leq n}$ of $\mathbb{Z}^n$.
- Get $F_i^*(\vec{c}_i) = \sup\{\vec{c}_i.\vec{z} \mid \vec{z} \in K, \vec{c}_1.\vec{z} = \ldots = \vec{c}_{i-1}.\vec{z} = 0\}$.
- Choose $n$ integers $\rho_i$ such that $\rho_i > F_i^*(\vec{c}_i)$.
- Let $M$ be the matrix with row vectors $(\vec{c}_i)_{1 \leq i \leq n}$ and $\vec{b}$ the vector such that $b_i = \rho_i$.

**Generalization of Lefebvre-Feautrier:** successive widths of $K$.

- Builds a corresponding lattice.
- Can be modified to get a 1D schedule with same determinant.
- In practice: look for “interesting” basis.
Size of local buffers, with “double-buffering” execution

- Loop bound $n$, tile size $b$ (in ex. $b = 10$)
- Tiling with $(i, j) \mapsto (i', j') = (n - j - 1, i)$
- Access $m = i + j = j' + n - i' - 1$
- First index in tile $(bl, bJ)$.
- Transfers Load$_p$, Load$_q$, Load$_c$, Store$_c$.

\[
\text{Load}_p(J) = \{m \mid 0 \leq m \leq n - 1, \ bJ \leq m \leq bJ + b - 1\}
\]
- size $2b$, if $n \geq 2b + 1$: 2 overlapping tiles.
- size $n$ if $n \leq 2b$: less than 2 tiles.
Size of local buffers, with “double-buffering” execution

- Loop bound $n$, tile size $b$ (in ex. $b = 10$)
- Tiling with $(i, j) \mapsto (i', j') = (n - j - 1, i)$
- Access $m = i + j = j' + n - i' - 1$.
- First index in tile $(bl, bJ)$.
- Transfers $\text{Load}_p$, $\text{Load}_q$, $\text{Load}_c$, $\text{Store}_c$.

$$\text{Load}_p(J) = \{ m \mid 0 \leq m \leq n - 1, \ bJ \leq m \leq bJ + b - 1 \}$$
- size $2b$, if $n \geq 2b + 1$: 2 overlapping tiles.
- size $n$ if $n \leq 2b$: less than 2 tiles.

$$\text{Load}_q(J) = \{ m \mid J = 0, \ 0 \leq m \leq n - 1, \ n - bl - b \leq m \leq n - bl - 1 \}$$
- size $b$ if $n \geq b$: 1 full tile.
- size $n$ if $n \leq b - 1$: 1 incomplete tile.
Size of local buffers, with “double-buffering” execution

- Loop bound \( n \), tile size \( b \) (in ex. \( b = 10 \))
- Tiling with \((i,j) \mapsto (i',j') = (n - j - 1, i)\)
- Access \( m = i + j = j' + n - i' - 1 \).
- First index in tile \((bl, bJ)\).
- Transfers \(\text{Load}_p, \text{Load}_q, \text{Load}_c, \text{Store}_c\).

\[
\text{Load}_p(J) = \{m \mid 0 \leq m \leq n - 1, bJ \leq m \leq bJ + b - 1\}
\]
- size \( 2b \), if \( n \geq 2b + 1 \): 2 overlapping tiles.
- size \( n \) if \( n \leq 2b \): less than 2 tiles.

\[
\text{Load}_q(J) = \{m \mid J = 0, 0 \leq m \leq n - 1, n - bl - b \leq m \leq n - bl - 1\}
\]
- size \( b \) if \( n \geq b \): 1 full tile.
- size \( n \) if \( n \leq b - 1 \): 1 incomplete tile.

\[
\text{Load}_c(J) = \{m \mid 0 \leq m, n - bl - b \leq m \leq n - 1 - bl, J = 0\}
\cup \{m \mid \max(1, bJ) \leq m + bl - n + 1 \leq \min(n - 1, bJ + b - 1)\}
\]
- size \( (2b - 1) + b = 3b - 1 \) if \( n \geq 2b + 1 \): 2 full overlapping tiles.
- size \( (2b - 1) + (n - b) = b + n - 1 \) if \( b \leq n \leq 2b \): 1 full, one incomplete
- size \( 2n - 1 \) if \( n \leq b - 1 \): only one tile.

\[
\text{Load}_p(J) = \{m \mid 0 \leq m \leq n - 1, bJ \leq m \leq bJ + b - 1\}
\]
- size \( 2b \), if \( n \geq 2b + 1 \): 2 overlapping tiles.
- size \( n \) if \( n \leq 2b \): less than 2 tiles.
Bee: Buffer Sizing and Allocation under Scheduling Constraints

Tool Demo

Christophe Alias
http://perso.ens-lyon.fr/christophe.alias

INRIA, ENS Lyon
Laboratoire de l’Informatique du Parallélisme

First Spring School on Polyhedral Code Analysis and Optimizations, May 2013
Collapse array cells that do not conflict.

Linear allocation functions:

\[ \sigma(i) = A_i \mod \vec{b} \]

Here: \( f[i] \rightarrow f_{\text{compacted}}[i\%2] \).
Step 1. Interference Analysis

- Array liveness analysis
- Interference relation $\bowtie_a : DS = \{ \vec{i} - \vec{j}, \ a[\vec{i}] \bowtie_a a[\vec{j}] \}$

Step 2. Mapping Derivation

- Critical lattice method (Cl@k).
- LF method

http://compsys-tools.ens-lyon.fr/bee
```c
int fib(int N)
{
    #pragma schedule[0]
    f[0] = 0;
    #pragma schedule[1]
    f[1] = 1;
    for(i=0; i<N; i++)
    {
        #pragma schedule[2][i]
        t1[i] = f[i-1];
        #pragma schedule[2][i+1]
        t2[i] = f[i-2];
        #pragma schedule[2][i+2]
        f[i] = t1[i] + t2[i];
    }
    #pragma schedule[3]
    return f[N-1];
}
```

- **Schedule** specified with `#pragma`.
- Here, software pipelining with $// = 1$.
  - $t1[]$ requires 2 cells
  - $t2[]$ requires 1 cell
Process 1 sends \( x[\cdot] \) to process 2.

Process 2 convoloves \( x[\cdot] \) \textit{on-the-fly}:

\[
y[i] = \sum_{k=0}^{3} x[i - k] \ast w[k]
\]

- Minimal size for \( x\_buffer \)? Allocation?