A Pipeline Pattern Detection Technique in Polly

Delaram Talaashrafi [1], Johannes Doerfert [2], Marc Moreno Maza [1]
[1]Western University, [2]Argonne National Laboratory

Background and Overview (1/2)

The polyhedral model is effective for optimizing loop nests using different methods:

- loop tiling, loop parallelizing, ...

They all optimize for-loop nests on a **per-loop** basis.
This work is about exploiting **cross-loop** parallelization, through tasking.
It is done by detecting pipeline pattern between iteration blocks of different loop nests.

Polly LLVM-based framework, applies polyhedral transformations:

- analysis, transformation, scheduling, AST generation, code generation.

OpenMP supports **task parallelization** via:

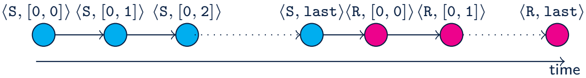
- `task` construct and `depend` clauses.

Background and Overview (2/2)

```
for(i=0; i<N-1; i++)
  for(j=0; j<N-1; j++)
  S:  A[i][j]=f(A[i][j], A[i][j+1], A[i+1][j+1]);

for(i=0; i<N/2-1; i++)
  for(j=0; j<N/2-1; j++)
  R:  B[i][j]=g(A[i][2*j], B[i][j+1], B[i+1][j+1], B[i][j]);
```

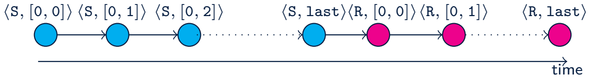
Background and Overview (2/2)

```
for(i=0; i<N-1; i++)
  for(j=0; j<N-1; j++)
  S:  A[i][j]=f(A[i][j], A[i][j+1], A[
      i+1][j+1]);

for(i=0; i<N/2-1; i++)
  for(j=0; j<N/2-1; j++)
  R:  B[i][j]=g(A[i][2*j], B[i][j+1],
      B[i+1][j+1], B[i][j]);
```



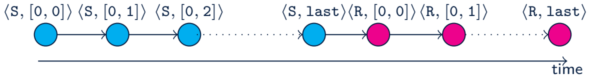
Background and Overview (2/2)

```
for(i=0; i<N-1; i++)
  for(j=0; j<N-1; j++)
  S:  A[i][j]=f(A[i][j], A[i][j+1], A[
      i+1][j+1]);

for(i=0; i<N/2-1; i++)
  for(j=0; j<N/2-1; j++)
  R:  B[i][j]=g(A[i][2*j], B[i][j+1],
      B[i+1][j+1], B[i][j]);
```



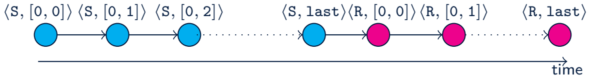
Background and Overview (2/2)

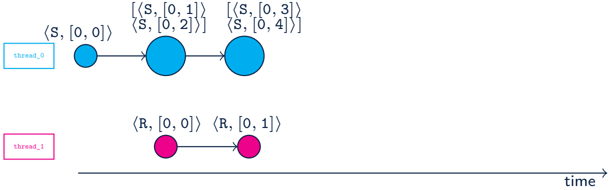
```
for(i=0; i<N-1; i++)
  for(j=0; j<N-1; j++)
  S:  A[i][j]=f(A[i][j], A[i][j+1], A[i+1][j+1]);

for(i=0; i<N/2-1; i++)
  for(j=0; j<N/2-1; j++)
  R:  B[i][j]=g(A[i][2*j], B[i][j+1], B[i+1][j+1], B[i][j]);
```



Background and Overview (2/2)

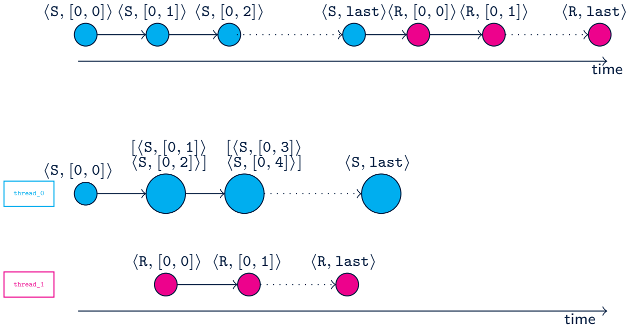
```
for(i=0; i<N-1; i++)
  for(j=0; j<N-1; j++)
  S:  A[i][j]=f(A[i][j], A[i][j+1], A[
      i+1][j+1]);

for(i=0; i<N/2-1; i++)
  for(j=0; j<N/2-1; j++)
  R:  B[i][j]=g(A[i][2*j], B[i][j+1],
      B[i+1][j+1], B[i][j]);
```





Background and Overview (2/2)

```
for(i=0; i<N-1; i++)
  for(j=0; j<N-1; j++)
  S:  A[i][j]=f(A[i][j], A[i][j+1], A[
      i+1][j+1]);

for(i=0; i<N/2-1; i++)
  for(j=0; j<N/2-1; j++)
  R:  B[i][j]=g(A[i][2*j], B[i][j+1],
      B[i+1][j+1], B[i][j]);
```



Transformation Algorithm (1/4)

Compute the **pipeline blocking map** of iteration domains such that:

- each block is an atomic task,
- we can establish a pipeline relation between all blocks of all statements,
- maximize the number of blocks of different loops that can execute in parallel.

Pipeline map

Consider two statements in a program:

- `S`: iteration domain $\mathcal{I}$, writes in memory location $\mathcal{M}$, $Wr(\mathcal{I} \to \mathcal{M})$
- `T`: iteration domain $\mathcal{J}$, reads from memory location $\mathcal{M}$, $Rd(\mathcal{J} \to \mathcal{M})$

The **pipeline map** between `S` and `T` is $\mathcal{T}_{\mathtt{S},\mathtt{T}}(\mathcal{I} \to \mathcal{J})$, where $(\vec{i}, \vec{j}) \in \mathcal{T}_{\mathtt{S},\mathtt{T}}$ if and only if:

1. after running all iterations of `S` up to $\vec{i}$, we can safely run all iterations of `T` up to $\vec{j}$,
2. $\vec{i}$ is the smallest vector and $\vec{j}$ is the largest vector with Property (1).

Transformation Algorithm (2/4)

Algorithm step I, computing pipeline map and source/target blocking map

1. Relate the iteration domains:
   $$[\mathcal{P}(\mathcal{J} \to \mathcal{I}), \mathcal{P} = Wr^{-1}(Rd)], \mathsf{Domain}(\mathcal{P}) = \mathcal{D}_{\mathcal{P}}$$
2. Map each member of $\mathcal{D}_{\mathcal{P}}$ to all members that are less than or equal to it:
   $$\mathcal{D}'_{\mathcal{P}}(\mathcal{J} \to \mathcal{J})$$
3. Map each $\vec{j} \in \mathcal{J}$ to the largest $\vec{i} \in \mathcal{I}$ that $\vec{j}$ and its previous iterations depend on:
   $$[\mathcal{H}(\mathcal{J} \to \mathcal{I}), \mathcal{H} = \mathsf{lexmax}(\mathcal{P}(\mathcal{D}'))]$$
4. The pipeline map is:
   $$\mathcal{T}_{\mathtt{S},\mathtt{T}} = \mathsf{lexmax}(\mathcal{H}^{-1})$$
5. Partition iteration domain of $\mathtt{S}$ ($\mathtt{T}$) with the domain (range) of $\mathcal{T}_{\mathtt{S},\mathtt{T}}$:
   $$\mathcal{B} = \mathsf{Dom}(\mathcal{T}_{\mathtt{S},\mathtt{T}}), \mathcal{B}' = \mathsf{lexleset}(\mathcal{I}, \mathcal{B}), (\mathcal{B} = \mathsf{Range}(\mathcal{T}_{\mathtt{S},\mathtt{T}})\ \mathcal{B}' = \mathsf{lexleset}(\mathcal{J}, \mathcal{B}))$$
6. Compute **source (target) blocking map**:
   $$[\mathcal{V}_{\mathtt{S}}(\mathcal{I} \to \mathcal{I}), \mathsf{lexmin}(\mathcal{B}')], ([\mathcal{Y}_{\mathtt{T}}(\mathcal{J} \to \mathcal{J}), \mathsf{lexmin}(\mathcal{B}')])$$

Transformation Algorithm (3/4)

Algorithm step II, computing pipeline blocking maps

There are several source and target blocking maps associated with each statement.

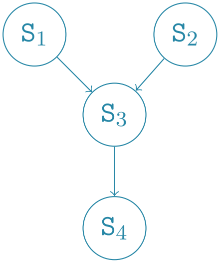
- Minimize the size of the blocks and construct the **optimal blocks**.
- get the lexmin of the union of all source and target blocking maps:
  $\mathcal{E}_{\mathsf{S}} = \mathsf{lexmin}((\bigcup_j (\mathcal{V}_{\mathsf{S}}^j) \cup (\bigcup_i (\mathcal{Y}_{\mathsf{S}}^i)))$

Algorithm step III, computing pipeline dependency relations

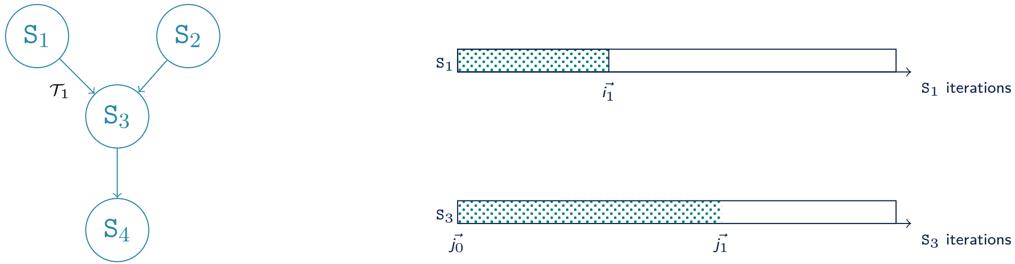
In a task-parallel program, there are dependency relations between different tasks.

- **Pipeline dependency relations** map each block to the blocks it needs to run correctly.
- For a statement S and a pipeline map $\mathcal{T}_i$, where S is the target:
  $\mathcal{Q}_{\mathsf{S}}^i = \mathcal{T}_i^{-1}(\mathcal{Y}_i(\mathsf{Range}(\mathcal{E}_{\mathsf{S}})))$

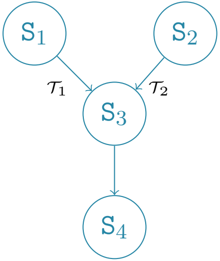
Transformation Algorithm (4/4)

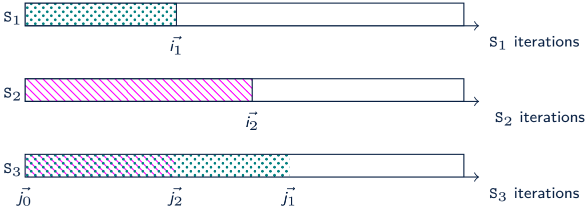


Transformation Algorithm (4/4)

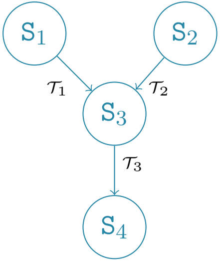


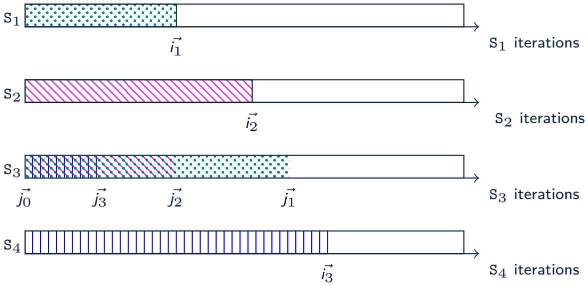
Transformation Algorithm (4/4)



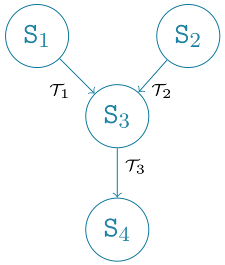


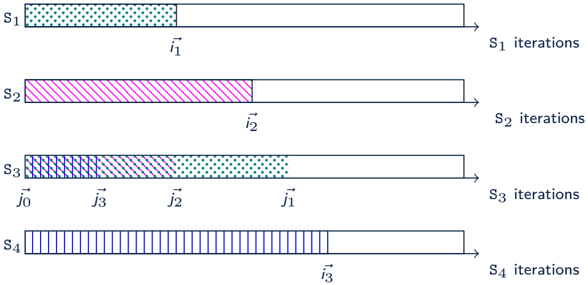
Transformation Algorithm (4/4)





Transformation Algorithm (4/4)





Optimal block of S_3: $\langle S_3, \vec{j_3} \rangle$
Pipeline dependencies: $\langle S_1, \vec{i_1} \rangle$, $\langle S_2, \vec{i_2} \rangle$

Implementation (1/2)

Analysis passes of Polly

Extend analysis passes of Polly to compute pipeline information for the iteration domains.

Scheduling

1. Create a schedule tree to iterate **over** blocks,
2. Create a schedule tree to iterate **inside** each blocks,
3. **Expand** the first tree with the second tree.
4. Create `pw_multi_aff_list` objects from pipeline dependency relations,
5. Add the `pw_multi_aff_list` objects as mark nodes to the schedule tree.

Implementation (2/2)

Abstract syntax tree

Generate AST from the new schedule tree.
The mark nodes in the schedule tree **annotates** the AST.

Code generation

1. Outline tasks to function calls,
2. Compute unique integer numbers from `pw_multi_aff_list` objects
   - this can be used in OpenMP `depend` clauses.
3. Replace the tasks part in the code with call to the `CreateTask` function that:
   - gets tasks and dependencies, creates OpenMP `tasks` with proper `depend` clauses,
   - handles the order between tasks created from the same loop nest.

Evaluation

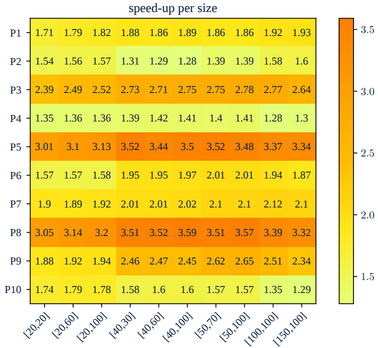


Figure: Speed-up of the tests with different access functions, considering different sizes, comparing sequential version and pipelined version.

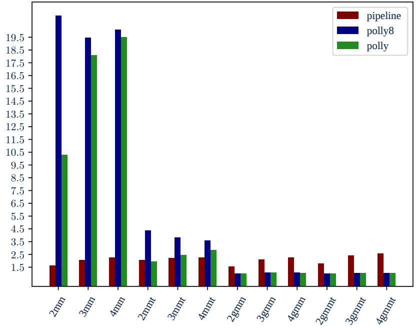


Figure: Comparing logarithm of speed-up gains of Polly running by all available threads, Polly running by `n` threads (`n` is the number of loop nests), and cross-loop pipelining for variants of generalized matrix multiplication.

Thank You!