Western

Pipeline Pattern Detection in the Polyhedral Model

Delaram Talaashrafi [1] Johannes Doerfert [2] Marc Moreno Maza [1]

[1]Western University [2]Argonne National Laboratory

ORCCA
Ontario Research Centre for Computer Algebra

Introduction

The polyhedral model has repeatedly shown how it facilitates various loop transformations, including loop parallelization, loop tiling, and software pipelining. However, parallelism is almost exclusively exploited on a per-loop basis without much work on detecting cross-loop parallelization opportunities. While many problems can be scheduled such that loop dimensions are dependence-free, the resulting loop parallelism does not necessarily maximize concurrent execution, especially not for unbalanced problems. In this work, we introduce a polyhedral-model-based analysis and scheduling algorithm that exposes and utilizes cross-loop parallelization through tasking. This work exploits pipeline patterns between iterations in different loop nests, and it is well suited to handle imbalanced iterations. Our LLVM/Polly-based prototype performs schedule modifications and code generation targeting a minimal, language agnostic tasking layer. We present results using an implementation of this API with the OpenMP `task` construct. For different computation patterns, we achieved speed-ups of up to $3.5\times$ on a quad-core processor while LLVM/Polly alone fails to exploit the parallelism. The objective of this paper is to detect the cross-loop task parallelism in a program. We exploit this opportunity by detecting pipeline pattern between iteration blocks of different for-loop nests; we call it **cross-loop pipeline pattern**. Detecting this pattern provides a building block towards exploiting the natural data-flow parallelism. There has been some efforts to consider this parallelization opportunity. The paper [3] generates pipelined multi-thread code by interleaving iterations of some loops. Paper [2] proposes an algorithm for detecting pipeline opportunities between iteration blocks of two loop nests.

Motivating example

Consider the program below, where `A` and `B` are two $\mathtt{N} \times \mathtt{N}$ matrices, and loops are sequential.

```
1  for(i=0; i<N-1; i++)
2    for(j=0; j<N-1; j++)
3    S:  A[i][j]=f(A[i][j], A[i][j+1], A[i+1][j+1]);
4
5  for(i=0; i<N/2-1; i++)
6    for(j=0; j<N/2-1; j++)
7    R:  B[i][j]=g(A[i][2*j], B[i][j+1], B[i+1][j+1],
8                  B[i][j]);
```

Figure 1. Example with cross-loop pipeline

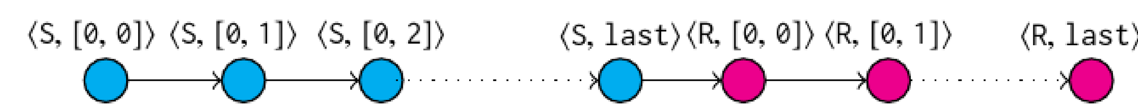

Figure 2. Sequential execution. `R` starts after iterations of `S` are finished.

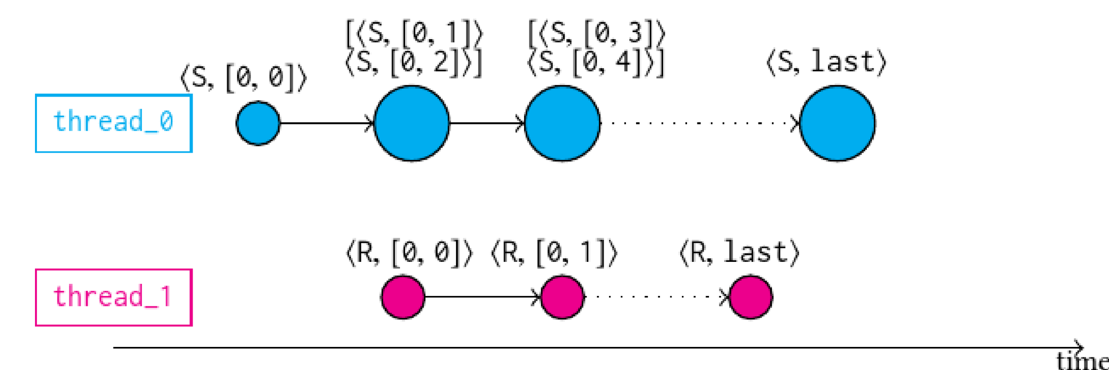

Figure 3. Pipeline execution. Iterations of `R` are overlapped with iterations of `S`.

In the pipeline execution, `thread_0` runs the iteration blocks of Statement `S`, and `thread_1` runs the iteration blocks of Statement `R`. `Thread_1` can start running an iteration of `R` right after `thread_0` finishes the iteration block of `S` that it depends on.

Finding pipeline map

Consider two statements `S` and `T` with respective iteration domains $\mathcal{I}$ and $\mathcal{J}$. Also, assume that the iterations of `S` write in a set of memory locations $\mathcal{M}$, and that the iterations of `T` read from $\mathcal{M}$. We define the **pipeline map** between `S` and `T` to be the relation $\mathcal{T}_{\mathtt{S},\mathtt{T}}(\mathcal{I} \rightarrow \mathcal{J})$, 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 $\vec{j}$, and (2) $\vec{i}$ is the lexicographically smallest vector and $\vec{j}$ is the lexicographically largest vector with Property (1). This map is called the pipeline map, because for every pair $(\vec{i}, \vec{j})$ in $\mathcal{T}_{\mathtt{S},\mathtt{T}}$, we can run iterations of `T` up to $\vec{j}$ and iterations of `S` after $\vec{i}$, in parallel. Repeating this pattern creates a pipeline among iteration blocks of the loop nests. We use the pipeline maps to partition the iteration domain of each statement to get the iteration blocks that are in pipeline relation. For a statement `S` and a pipeline map $\mathcal{T}$, if `S` is the source (resp. target) statement, we partition its iteration domain, $\mathcal{I}$, such that each element of $\mathsf{Dom}(\mathcal{T})$ (resp. $\mathsf{Range}(\mathcal{T})$) is the lexicographically largest member of its part. Then, by mapping each member of each part to the largest member of that part, we obtain the **source blocking map** $\mathcal{V}_{\mathtt{S}}(\mathcal{I} \rightarrow \mathcal{I})$ (resp. a **target blocking map** $\mathcal{Y}_{\mathtt{S}}(\mathcal{I} \rightarrow \mathcal{I})$).

Example of pipeline map

Consider the previous example with `N=20`. The pipeline map between statements `S` and `R` is:

$$\{\mathtt{S}[i_0, i_1] \rightarrow \mathtt{R}[o_0, o_1] : \exists (e_0 = \lfloor (i1)/2 \rfloor : o_0 = i_0 \wedge 2e_0 = i_1 \wedge 2o_1 \geq i_1 \wedge 2o_1 \leq 1 + i_1 \wedge i_0 \geq 0 \wedge i_0 \leq 8 \wedge i_1 \geq 0 \wedge i_1 \leq 16)\}.$$

One part of the source blocking map is:

$$\exists (e_0 = \lfloor (o_1)/2 \rfloor : o_0 = i_0 \wedge 2e_0 = o_1 \wedge i_0 \geq 0 \wedge i_0 \leq 8 \wedge i_1 \geq 0 \wedge i_1 \leq 16 \wedge o_1 \geq i_1 \wedge o_1 \leq 1 + i_1).$$

Therefore, some elements of the map are:

$$\{\mathtt{S}[1,1] \rightarrow \mathtt{S}[1,2], \mathtt{S}[1,2] \rightarrow S[1,2], \mathtt{S}[1,3] \rightarrow \mathtt{S}[1,4], \mathtt{S}[1,4] \rightarrow \mathtt{S}[1,4]\}.$$

Iterations $[1,1]$ and $[1,2]$ are in one block, and $[1,3]$ and $[1,4]$ are in another block.

Detecting pipeline relations

For each statement S, there are potentially several pipeline maps, for which S is either a source or a target. As a result, there are potentially several source and target blocking maps attached to S. However, for the statement S, we must have a single pipeline blocking map of the iteration domain of S, so that each pipeline block can be considered as a *task* (actually a pipeline stage). Therefore, for each statement S, we integrate all its source and target blocking maps together. Our goal is to establish a pipeline relation between all blocks of all statements. Moreover, we choose these blocks so as to maximize the number of blocks of different loops that can execute in parallel. In order to generate a correct task-parallel program we compute the dependence relations between all tasks. As for the pipeline maps and blocking maps, the construction of this dependence graph is achieved with algebraic operations on binary relations. The performance improvement of the pipelined program comes from the places that we can **overlap** the execution of iteration blocks of *different* for-loop nests. Therefore, *the performance of the pipelined program is limited to the loop nest with the maximum running time,* L_{max}. Figure 4 shows this idea.

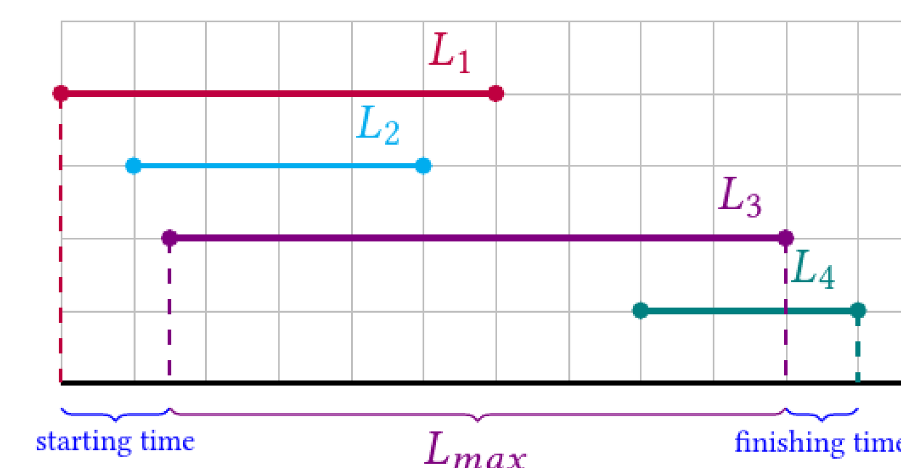

Figure 4. Average case performance of pipelined program, where the third loop has the largest running time.

Code generation for the pipelined program

We implement the pipeline detection algorithm as a part of Polly [1] and use the ISL library [4] for polyhedral computation. We modify Polly passes in the analysis, transformation, and code generation phases to add support for the pipeline pattern detection and code generation. For exploiting the detected parallelism, we use OpenMP `task` constructs. First, we extend the definition of the SCoP to include information needed for pipelining. Then, we used the information to create a schedule tree and the AST that includes tasks and their dependencies. In the final step, we design a high-level OpenMP function for exploiting the detected task parallelism. Each task is defined as a function pointer with its input arguments integrated into a structure. We use the in-dependencies and out-dependencies of the tasks as computed in the previous steps. We also need the size of the input and the total number of statements that a task depends on.

```
1  void CreateTask(void (*f) (void *), void *input,
2                  int outDepend, int outIdx,
3                  int *inDepend, int *inIdx,
4                  int inputSize, int dependNum)
```

Figure 5. Signature of the function for creating tasks.

Evaluation

We simulate compute-intensive kernels by consecutively calling the `next_prime` function of the GMP library on the elements of `mpz` matrices.

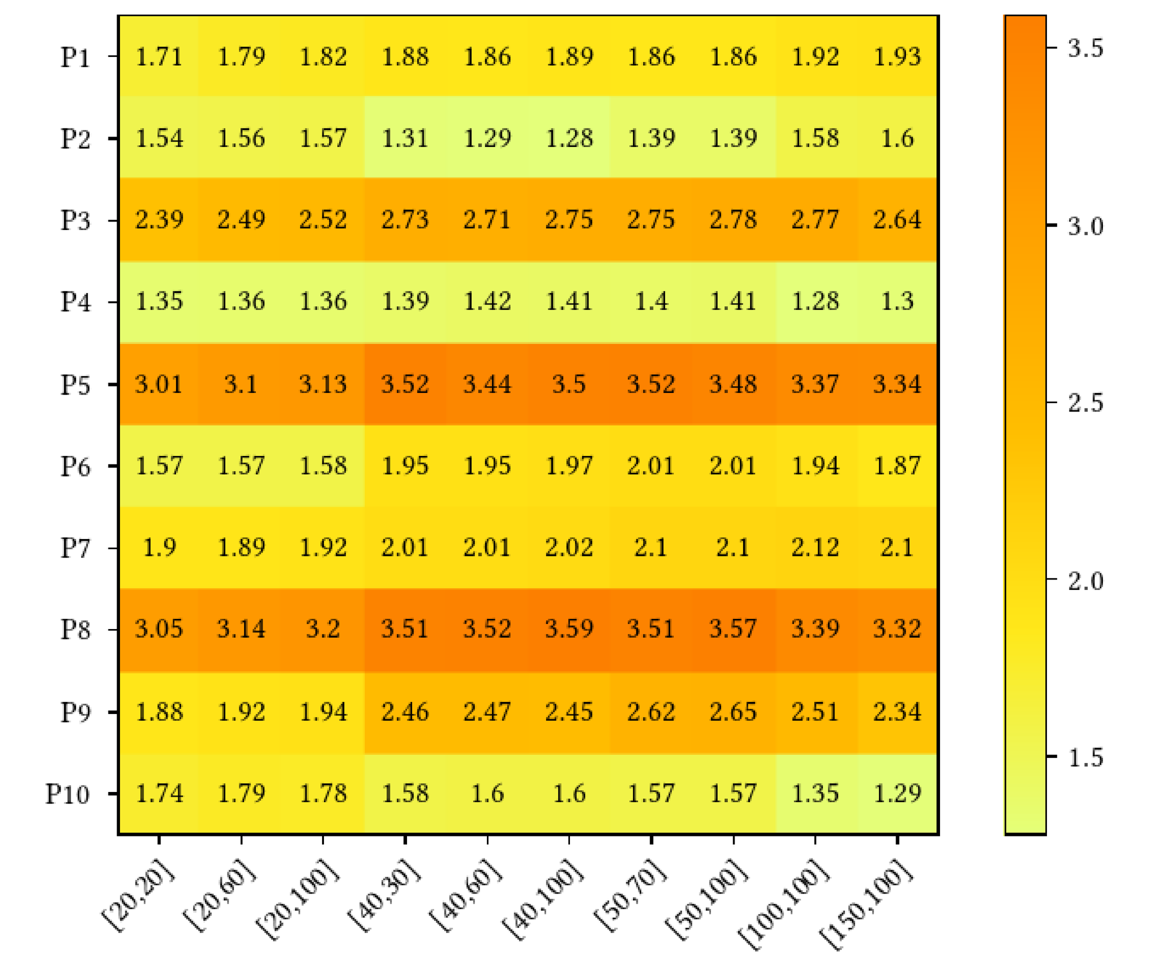

Figure 6. Speed-up of the tests, considering different values for N and SIZE.

References

[1] Tobias Grosser, Hongbin Zheng, Raghesh Aloor, Andreas Simbürger, Armin Größlinger, and Louis-Noël Pouchet. Polly-polyhedral optimization in llvm. In *Proceedings of the First International Workshop on Polyhedral Compilation Techniques (IMPACT)*, volume 2011, page 1, 2011.

[2] Kornilios Kourtis, Martino Dazzi, Nikolas Ioannou, Tobias Grosser, Abu Sebastian, and Evangelos Eleftheriou. Compiling neural networks for a computational memory accelerator. 2020.

[3] Harenome Razanajato, Cédric Bastoul, and Vincent Loechner. Pipelined multithreading generation in a polyhedral compiler. In *IMPACT 2020, in conjunction with HiPEAC 2020*, 2020.

[4] Sven Verdoolaege. isl: An integer set library for the polyhedral model. In *International Congress on Mathematical Software*, pages 299–302. Springer, 2010.