A Pipeline Pattern Detection Technique in Polly

Delaram Talaashrafi
Western University
London, Ontario, Canada
dtalaash@uwo.ca

Johannes Doerfert
Argonne National Laboratory
Lemont, IL, USA
jdoerfert@anl.gov

Marc Moreno Maza
Western University
London, Ontario, Canada
moreno@csd.uwo.ca

Abstract

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× on a quad-core processor while LLVM/Polly alone fails to exploit the parallelism.

Keywords: polyhedral model, pipelining, LLVM

ACM Reference Format:
Delaram Talaashrafi, Johannes Doerfert, and Marc Moreno Maza. 2022. A Pipeline Pattern Detection Technique in Polly. In *Proceedings of IMPACT 2022.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 Overview

The polyhedral model [15, 46] has proved to be very effective for optimizing loop nests by using different methods such as loop tiling, loop parallelizing, and software pipelining [1, 4, 7]. Almost all these methods optimize for-loop nests on a per-loop basis. However, another opportunity for optimization might exist in the program, which one can exploit by considering cross-loop parallelization; executing iteration blocks of different loop nests in parallel when it does not violate any dependence relations. There has been some efforts to consider this parallelization opportunity. The paper [40] generates pipelined multi-thread code by interleaving iterations of some loops. Paper [23] proposes an algorithm for detecting pipeline opportunities between iteration blocks of two for-loop nests. Also, [10] uses cross-loop data reuse for cache optimizations. However, we are not aware of any fully-automatic, LLVM-based method for detecting and exploiting parallelization opportunities between iterations of different for-loop nests through tasking.

The main 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; [1] we call it *cross-loop pipeline pattern*.

Detecting cross-loop pipelining provides a building block towards exploiting the natural data-flow parallelism. However, the existing loop optimization methods based on the polyhedral model have a limited ability to extract cross-loop pipeline patterns, as we explain in Section 2.

We assume the program consists of consecutive for-loop nests. We also assume that an iteration of a loop nest may depend on the previous iterations of the same loop nest, as well as iterations of the loop nests before it. Detecting cross-loop task parallelism is particularly important and effective for programs where (1) compute-intensive functions are called inside for-loop nests, or (2) no optimization opportunities for individual for-loop nests exist.

For instance, consider the program in Listing 1, where A and B are two initialized N × N matrices. Polly[20], LLVM-

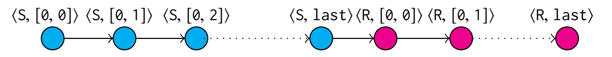
```
for(i=0; i<N-1; i++)
  for(j=0; j<N-1; j++)
  S:  A[i][j]=f(A[i][j], A[i][j+1], A[i+1][j+1]);

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]);
```

Figure 1. Example with cross-loop pipeline

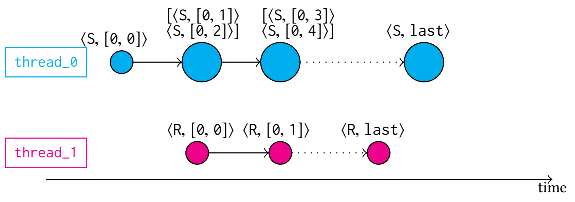
[1]Not be confused with software pipelining and DOACROSS loops, where a pipeline pattern exists between different iterations of the *same* loop nest.

based framework for applying polyhedral transformations, detects first level tiling, but it cannot detect parallelism for any of the for-loops in the program. However, there is a parallelization opportunity between iteration blocks of S and iteration blocks of R. For instance, consider the first two iterations of the second for-loop nest for computing B[0][0] and B[0][1]. The only element of the matrix A we need for the first iteration is A[0][0], and for the second iteration is A[0][2]. Therefore, when A[0][0] is computed (after finishing the first iteration of the first loop nest), we can compute B[0][0] and the following elements of the matrix A in parallel. With the same method, when A[0][2] is computed, we can compute B[0][1] and the following elements of the matrix A in parallel. Note that the iterations of each statement run in their sequential order.

Figure 2 illustrates this idea. The upper part, 2a, shows the sequential execution of the iterations of the statements S and R, where iterations of R begin after all iterations of S are finished. The lower part, 2b, shows the execution of the program after exploiting cross-loop task parallelization. In this case, `thread_0` runs the iteration blocks of S, and `thread_1` runs the iteration blocks of R. Thread_1 can start running an iteration of R right after `thread_0` finishes the iteration block of S that it depends on.



(a) Sequential execution. R starts after iterations of S are finished.



(b) Pipeline execution. Iterations of R are overlapped with iterations of S, and R is not part of the critical path anymore.

Figure 2. Visualization of pipeline execution

In this work, we detect and exploit the cross-loop pipeline pattern using the polyhedral model. Our method is implemented as part of LLVM/Polly[20] and operates at compile time on the LLVM-IR [25]. The main idea is to block iteration domains such that finishing each block provides the requirements for running not only the next block in the same iteration domain, but also blocks in other iteration domains. After computing dependence relations between newly generated blocks, we construct an OpenMP `task` for each block to exploit the detected cross-loop task parallelism.

We begin by motivating our work by explaining related works and comparing them to our research in Section 2. We provide background information in Section 3; after that, we explain our transformation algorithm in Section 4. Then, we go through the details of the scheduling algorithm and code generation in Section 5. We conclude with reporting on our experimental results in Section 6 and the future plans for continuing this project in Section 7.

2 Related works

Various lines of research have been investigated on automatically detecting pipeline-related patterns. In this section, we go through some of the papers that are related to our work.

Paper [35] discusses the exploitation of pipeline parallelism, including use-cases, and the critical challenge of managing dependencies between the source region and the target region. We try to address this challenge in the case of pipelining iterations of for-loop nests. Also, the software pipelining technique for pipelining iterations of a single loop is discussed in [24].

The paper [40] follows the same objective as our paper, that is, to use the polyhedral model and exploit pipeline parallelism opportunities between loop nests so as to optimize (part of) programs that conventional polyhedral optimizers cannot optimize. However, there are differences between the two approaches. Our prototype operates at the IR level of a non-optimized program, whereas the prototype in [40] operates at the source level of programs already optimized by Pluto and parallelized by the OpenMP API. Also, it can detect and exploit the pipeline pattern only if (1) the considered loop nests have identical iteration domains and chunk sizes, (2) are not associated with SIMD constructs, and (3) are in the same parallel region. Moreover, each iteration of the target loop nest should depend on the same or the previous iterations of its source loop nest. With these considerations, the prototype in [40] can use the clauses `ordered` and `nowait` of OpenMP to exploit the pipeline pattern. In our work, by using the general transformation algorithm described in Section 4, and by taking advantage of the OpenMP constructs `task` and `depend`, we can detect and exploit pipeline patterns in loop nests of sequential programs with arbitrary memory accesses. Also, our transformation algorithm for task detection is independent of the OpenMP tasking layer.

The method explained in [23] detects pipeline parallelism to make machine learning models executions more efficient on the so-called *computational memory accelerators* considered in that paper. We provide more details on the algorithm of this paper in Section 4.

The objectives of the authors in [21] are similar to ours: they aim at exploiting parallelization between different loop nests. However, the method of [21] and the output are different. The authors discuss a method based on linear regression for detecting pipeline patterns in pairs of consecutive loop nests, using run-time information.

The Pluto [7] algorithm supports a method for detecting software pipelining in the form of DOACROSS loops applied on a program tiled by the Pluto algorithm. Paper [13] explains a polyhedral-model-based method for designing a compiler-runtime system to exploit task parallelism in distributed and shared memory architectures. It uses Pluto's tiling and parallelization for task detection, and the runtime system coordinates the dependencies between tasks.

Paper [41] uses the polyhedral model to exploit DOACROSS loops using OpenMP. Contrary to our approach, their input program is in data-flow graph language. The work explained in [9] optimizes programs using OpenMP tasks, where program annotations explicitly specify the tasks. Paper [37] introduces OpenStream as a data-flow extension of OpenMP. It can exploit pipeline and data-flow parallelism on an annotated program.

A well-studied subject closely related to our work is automatically extracting the data-flow graph from the program to run it on data-flow architectures. For example, paper [27] develops an algorithm for automatically extracting data-flow threads from programs. In another work, the paper [31] develops an LLVM-based prototype to find the data-flow graph between LLVM-IR instructions of a program.

In paper [32], the authors propose a fully-automatic and non-speculative compiler technique for improving the performance of DOACROSS loops via reducing the cost of communication. In the multiprocessor system on chip domain, paper [8] uses a not fully-automatic method based on machine learning and data mining algorithms to construct so-called coupled blocks, the smallest unit of a program considered as a task, and finds task parallelism relations between them. In the same domain, paper [11] extracts software pipelining and optimizes the granularity of the stages using linear programming.

Software pipelining is an important transformation and well-studied subject in high-level synthesis (HLS). For example, using the polyhedral model [30]improves the applicability and efficiency of nested loop pipelining, also known as nested software pipelining. In a similar approach, [28] uses the polyhedral model to improve software pipelining for HLS applications by extending the method to handle loops with uncertain and non-uniform dependencies. Moreover, paper[2] develops an algorithm based on the polyhedral model to optimize data transfer to offloading devices via different methods, including pipelining communications and computation.

3 Background

In this section, we briefly explain the background of our work, which includes some related concepts of the polyhedral model, pipeline parallelism, and OpenMP tasking.

3.1 Polyhedral Model

The polyhedral model [33] is a mathematical description for representing and manipulating static control parts (SCoPs) [15] of a program by using Presburger's arithmetic [39]. It is based on the notions of *iteration domain* of dynamic instances of a statement, *memory access relation* between those dynamic instances, and their *relative execution orders*, as well as the notion of a *schedule*. We refer to the articles [5, 16, 17] for more detailed explanation on the notions.

The polyhedral model optimizes programs via different loop transformations (e.g. fission, fusion, tilling). Each transformation is equivalent to changing the schedule of statements. Different approaches [1, 7] are proposed for loop transformation and code generation [3, 42] in the polyhedral model. Also, various libraries such as ISL[43–45], Polylib [29], Piplib [14], and Omegalib [22] implement the polyhedral model's underlying mathematical operations. Following Polly, we use the ISL library for Z-polyhedral computations.

The ISL library represents Z-polyhedra as *sets* of integer tuples. A *map* is a binary relation from one set to another. Different operations are defined on maps in the ISL library. The *inverse map* of a map M, denoted by M^{-1}, is the set of the pairs $(\vec{j}, \vec{i})$ such that $(\vec{i}, \vec{j}) \in M$. The domain (resp. range) of M denoted by $\mathsf{Dom}(M)$ (resp. $\mathsf{Range}(M)$) is the set of all first elements of members of M (resp. M^{-1}). We denote by $\mathsf{lexmax}(M)$ the subset of M consisting of all pairs $(\vec{i}, \vec{j})$ so that $\vec{i} \in \mathsf{Dom}(M)$ and $\vec{j}$ is the lexicographically largest $\vec{k} \in \mathsf{Range}(M)$ so that $(\vec{i}, \vec{k}) \in M$. The *composition* of two maps M_1 and M_2 is denoted by $M_1(M_2)$. It is the set of all pairs $(\vec{i}, \vec{j})$, such that there exists a vector $\vec{k}$, where $(\vec{i}, \vec{k}) \in M_2$ and $(\vec{k}, \vec{j}) \in M_1$

Schedule are represented in the form of a tree, called *schedule tree* in the ISL library. Nodes in a schedule tree have different types for representing different execution orders. The most important ones that we are using in this article are: domain node, band node, sequence node, mark node, and expansion node. More detailed information on the tree representation of the schedules can be found in [44, 45].

3.2 Pipeline Parallelism

Pipeline parallelism is a well-known technique [6, 18, 26, 36] for parallelizing different applications. It can be used when a sequence of data items has to go through a sequence of *stages*, and the input of each stage is the output of its previous stage. Concurrency happens when a stage i can start operating on a data item d after stage $i-1$ has finished processing d, but not the whole sequence of data items.

3.3 Tasking in OpenMP

Since version 3.0, OpenMP[12, 34] supports task parallelism, using the `omp task` pragma. To increase the applicability of task parallelism, OpenMP 4.0 introduces the depend clause. Let `M` be a shared memory location. Using the `depend(in:M)`, `depend(out:M)`, and `depend(inout:M)` clauses, one can specify whether the considered task reads, writes, or both reads and writes `M`. The runtime system uses this information to manage dependencies between tasks, and decide whether a task can execute, or should wait for other tasks to finish.

4 Transformation Algorithm

In this section, we explain our algorithm for detecting the cross-loop pipeline pattern in a program. We explain each step in detail and conclude the section with performance analysis of the algorithm.

The algorithm proposed in [23] provides the foundation for our transformation algorithm. The algorithm of [23] first considers two for-loop nests, called source and target, where iterations of the source loop nest write to a shared array, and iterations of the target loop nest read from that same shared array. Then, this algorithm finds a relation that maps the index of each write in the shared array to the maximum iteration of the target that can safely execute. Finally, using the specifications of the so-called *computational memory accelerator* studied in that paper, this map coordinates different pipeline stages between iteration blocks of the two considered for-loop nests.

The *pipeline map* computed by our algorithm (Section 4.1) considers iteration blocks of the source and the target for-loop nests for coordinating different stages of the pipeline. Moreover, on the contrary to the method in [23], our algorithm does not stop after finding the pipeline relations between pairs of for-loop nests. By computing *pipeline blocking maps of iteration domains* (Section 4.2), we extend the algorithm of [23] to detect the pipeline pattern between all dependent loop nests in the program. In addition, we compute *pipeline dependency maps* (Section 4.3) to determine dependence relations between tasks at compile time and make them suitable for generating task-parallel OpenMP program in the next phase.

4.1 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 $\mathcal{T}_{\mathsf{S,T}}(\mathcal{I} \to \mathcal{J})$, where $(\vec{i}, \vec{j}) \in \mathcal{T}_{\mathsf{S,T}}$ 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 smallest (lexicographically) vector and $\vec{j}$ is the largest (lexicographically) vector with Property (1). This map is called the pipeline map; because for every pair $(\vec{i}, \vec{j})$ in $\mathcal{T}_{\mathsf{S,T}}$, 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.

To compute the pipeline map, we take a similar approach as the one used in [23]. Let $Wr(\mathcal{I} \to \mathcal{M})$ be the *write relation* of S, that is, the set of the pairs $(\vec{i}, m) \in \mathcal{I} \times \mathcal{M}$ so that location m is written at iteration $\vec{i}$. Similarly, let $Rd(\mathcal{J} \to \mathcal{M})$ be the *read relation* of T, that is, the set of the pairs $(\vec{j}, m) \in \mathcal{J} \times \mathcal{M}$ so that location m is read at iteration $\vec{j}$. Also, assume that there is no over-write, that is, Wr is injective.

Using Wr and Rd, we define $\mathcal{P}(\mathcal{J} \to \mathcal{I})$, as the composition of Wr^{-1} by Rd, that is, $\mathcal{P} = Wr^{-1}(Rd)$. Relation $\mathcal{P}$ relates the two iteration domains.

Then, we find the domain of $\mathcal{P}$, $\mathcal{D}_{\mathcal{P}}$. By mapping each member of $\mathcal{D}_{\mathcal{P}}$ to all other members that are lexicographically less than or equal to it, we get the map $\mathcal{D}'(\mathcal{J} \to \mathcal{J})$.

After that, we find the relation $\mathcal{H}(\mathcal{J} \to \mathcal{I})$ defined by $\mathcal{H} = \mathsf{lexmax}(\mathcal{P}(\mathcal{D}'))$. This relation maps each read iteration $\vec{j}$ of the target statement to the lexicographically largest write iteration $\vec{i}$ of the source statement that $\vec{j}$ and its previous iterations depend on.

The final step to get the pipeline map is to find $\mathcal{H}^{-1}(\mathcal{I} \to \mathcal{J})$, and deduce the pipeline map $\mathcal{T}_{\mathsf{S,T}}$ as:

$$\mathcal{T}_{\mathsf{S,T}} = \mathsf{lexmax}(\mathcal{H}^{-1}). \tag{1}$$

Because one iteration of the source statement may be mapped to multiple iterations of the target statement, we use the operation lexmax to get the maximum one.

As an example, consider Listing 1 with `N=20`. The pipeline map between statements S and R is:

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

In the next step, 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 first 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 lexicographically largest member of that part, we get a *source blocking map* $\mathcal{V}_{\mathsf{S}}(\mathcal{I} \to \mathcal{I})$ (resp. a *target blocking map* $\mathcal{Y}_{\mathsf{S}}(\mathcal{I} \to \mathcal{I})$). To compute these maps, let $\mathcal{B} = \mathsf{Dom}(\mathcal{T})$ if S is the source in the pipeline map $\mathcal{T}$ (resp. $\mathcal{B} = \mathsf{Range}(\mathcal{T})$ if S is the target in the pipeline map $\mathcal{T}$). We compute $\mathcal{B}'$ as:

$$\mathcal{B}' = \mathsf{lexleset}(\mathcal{I}, \mathcal{B})$$

Then, the source blocking map, $\mathcal{V}_{\mathsf{S}}(\mathcal{I} \to \mathcal{I})$ (resp. the target blocking map $\mathcal{Y}_{\mathsf{S}}(\mathcal{I} \to \mathcal{I})$) is as:

$$\mathsf{lexmin}(\mathcal{B}'_{\mathcal{T}}). \tag{2}$$

If there are no iterations of T depending on the final iterations of S, then those last iterations of S do not appear in $\mathcal{T}_{\mathsf{S,T}}$; therefore, they do not appear in the source blocking map. To handle this case, we add a block consisting of all remaining iterations by mapping them to the lexicographically maximum iteration of the iteration domain.

Continuing with the example of the Listing 1, 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:

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

This means that iterations $[1,1]$ and $[1,2]$ are in one block, and $[1,3]$ and $[1,4]$ are in another block.

4.2 Pipeline Blocking Maps of Iteration Domains

It is important to note that for each statement, there are *several* pipeline maps. As a result, there are several source and target blocking maps. For instance, consider Listing 3, which adds a for-loop nest to Listing 1. There are two source

```
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]);

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

Figure 3. Example with 3 loop nests

blocking maps for the statement S; one for the pipeline map between S and R, and one for the pipeline map between S and U. For the statement R, there is one target blocking map for the pipeline map between S and R, and one source blocking map for th pipeline map between R and U. For the statement U, there are two target blocking maps; for the pipeline maps between S and U, and between R and U.

However, we need to have a single *pipeline blocking map* of iteration domain per statement, where each pipeline block can be considered an *atomic task*. Therefore, for each statement, we should integrate all its source and target blocking maps such that we can establish a pipeline relation between all blocks of all statements. We also need to choose these blocks to maximize the number of blocks of different loops that can execute in parallel to get the best possible performance at the end. To satisfy both conditions, we minimize the size of the blocks as much as possible and construct the *optimal blocks* from all blocking maps associated with each statement. In fact, for each statement, we compute the lexmin of the union of all source and target pipeline blocking maps:

$$\mathcal{E}_{\mathsf{S}} = \mathrm{lexmin}((\bigcup_j(\mathcal{V}^j_{\mathsf{S}}) \cup (\bigcup_i(\mathcal{Y}^i_{\mathsf{S}}))). \tag{3}$$

In this equation, $\mathcal{Y}^i_{\mathsf{S}}$ (resp. $\mathcal{V}^j_{\mathsf{S}}$) goes over the target (resp. source) blocking maps of S with respect to the pipeline maps between S and other statements that S depends on (resp. statements that depend on S).

From this point on, for two vectors $\vec{i}$ and $\vec{j}$ in the iteration domain of S, if $\mathcal{E}_{\mathsf{S}}(\vec{i}) = \mathcal{E}_{\mathsf{S}}(\vec{j}) = \vec{\ell}$, we say that $\vec{i}$ and $\vec{j}$ are in the same block, and we call this block $\vec{\ell}$. With this definition, we can say that equation 3 assigns to each iteration the smallest block that it belongs to, among all source and target blocking maps.

To illustrate the effectiveness of choosing optimal blocks for correctness and efficiency, consider figure 4. In this example, statements S_1 and S_2 are sources of the statement S_3, and S_3 is the source for statement S_4. We want to find the first pipeline block of S_3 after $\vec{j_0}$. In other words, we are looking for the lexicographical maximum vector of the first block after $\vec{j_0}$.

After finishing the execution of S1 up to iteration $\vec{i_1}$, the dependencies to S1 are satisfied for iterations of S_3 up to $\vec{j_1}$. The same holds for iterations of S2 up to $\vec{i_2}$ and iterations of S_3 up to $\vec{j_2}$. On the other hand, after finishing iterations of S_3 up to $\vec{j_3}$, we can run iterations of S_4 up to $\vec{i_3}$. As a result, after finishing iterations of S1 up to $\vec{i_1}$, and iterations of S2 up to $\vec{i_2}$, we can safely run S_3 up to iteration $\vec{j_2}$. Therefore, considering any vector between $\vec{j_0}$ and $\vec{j_2}$ maintains the correct execution of S_3. However, by we choose $\vec{j_3}$, the optimal block computed by equation 3, we maximize the number of blocks of different statements that can run in parallel; because S_4 can also start running right after $\vec{j_3}$ is finished.

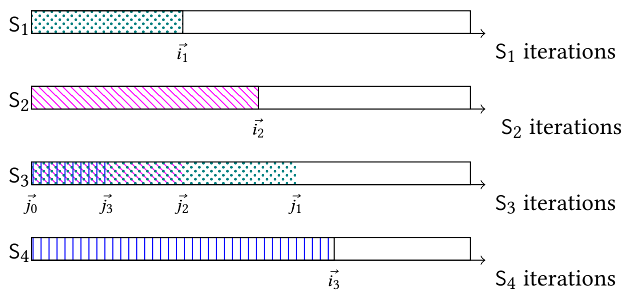


Figure 4. Choosing $\vec{j_3}$ as the first pipeline block after $\vec{j_0}$ maintains correctness and maximizes the number of blocks of different statements that can run in parallel.

4.3 Pipeline Dependency Relations

Up to this point, we have found the pipeline blocking maps of the iteration domain of each statement. These blocks of iterations are considered as the tasks (pipeline stages). However, to have a correct task-parallel program, we also need to compute the dependence relations between different tasks so that they can be used to coordinate OpenMP tasks. Therefore, after computing pipeline blocking maps of all statements, in the next step, we find the requirements of each block. By requirements of a block, we mean the blocks of its source statements it needs to execute safely. This part explains how to find *pipeline dependency relations*, which are maps between each block and its requirements. These maps will be used as in-dependencies (`depend(in:)`) of the OpenMP tasks we create in the next phase.

To find pipeline dependency relations of a statement S, consider a specific pipeline map $\mathcal{T}_i$ and its corresponding target blocking map, $\mathcal{Y}_i$. First, for every block of S, that is, for each element of Range($\mathcal{E}_S$), we compute the block of $\mathcal{Y}_i$ that it belongs to. Then, we can get the last required block using $\mathcal{T}_i$. Considering all target blocking maps of S, we get an array of maps showing the requirements of the blocks of S. We show this array with Q_S, and each index of it with Q^i_S. Relation 4 shows the computation of each map Q^i_S.

$$Q^i_S = \mathcal{T}_i^{-1}(\mathcal{Y}_i(\mathrm{Range}(\mathcal{E}_S))). \tag{4}$$

In equation 4, $\mathcal{T}_i$ goes over all pipeline maps that S is considered as their target statement, and $\mathcal{Y}_i$ is the target blocking map corresponds to $\mathcal{T}_i$.

Furthermore, running each block of a statement S provides the requirements for some blocks of the statements that are dependent on S. This only depends on the last executed iteration of S. We can get this relation from the identity map of Range($\mathcal{E}_S$), and we call it Q'_S. This maps will be used as the out-dependency (`depend(out:)`) of the OpenMP tasks we create in the next phase.

The final algorithm for finding the cross-loop pipeline relation of a SCoP is summarized in Algorithm 1.

Algorithm 1: Pipeline detection algorithm

```
Input  : Scop in its polyhedral representation
Output : Scop with pipeline information
1  for all statement pairs S and T of the Scop do
2      if T depends on S then
3          $\mathcal{T}_{S,T}$ = pipeline map(S, T);
4          $\mathcal{V}_{S,T}$ = source blocking map(S, $\mathcal{T}_{S,T}$);
5          $\mathcal{Y}_{T,S}$ = target blocking map(T, $\mathcal{T}_{S,T}$);
6          $\mathcal{E}_S = \mathcal{E}_S \cup \mathcal{V}_{S,T}$;
7          $\mathcal{E}_T = \mathcal{E}_T \cup \mathcal{Y}_{T,S}$;
8  for all statements S of the Scop do
9      $\mathcal{E}_S$ = lexmin($\mathcal{E}_S$);
10     $Q'_S$ = identity map(Range($\mathcal{E}_S$));
11 for all pipeline maps $\mathcal{T}_{S,T}$ do
12     $Q_T$ = append($\mathcal{T}_{S,T}^{-1}(\mathcal{Y}_{T,S}(\mathrm{Range}(\mathcal{E}_T)), Q_T)$;
13 for all statements S of the Scop do
14     add $\mathcal{E}_S$, $Q_S$, $Q'_S$ to the Scop;
15 return Scop;
```

4.4 Algorithm Efficiency

In this part, we show that in the general case of the algorithm, the best performance we can get from cross-loop pipelining is constraint by the most time-consuming loop nest. We also explain that with our transformation algorithm, we can automatically get the ideal speedup (ignoring the overhead related to task creation).

Assume that the input program consists of N for-loop nests $L_1, \cdots, L_N$. We want to compare the total running time of the pipelined execution and the sequential execution.

We have to run all iterations of all loops, and since we do not consider any other form of parallelism in the general case, we do not reduce the running time of individual for-loop nests. The performance improvement 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}*, and we have the following formula for the running time of the pipelined program:

$$\mathrm{time}(L_{max}) \le \mathrm{time(pipeline)} \le \mathrm{time(sequential)} \tag{5}$$

The lower bound happens when the first loop nest has the maximum running time and the execution of all other for-loop nests can be covered by that. The average case is when the i^{th} loop nest has the maximum running time. Also, we usually cannot assume that the running time of all loop nests after the maximum loop can be covered. For instance, consider Figure 5.

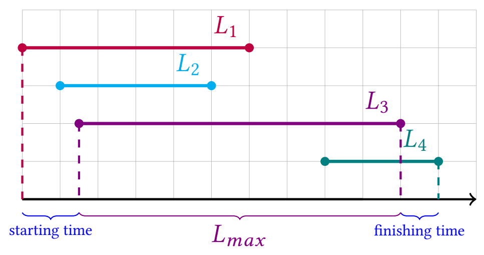


Figure 5. Average case performance of pipelined program, where the third loop has the largest running time.

As a result, we can compute the total running time of the pipelined program with equation 6.

$$\mathrm{time(pipeline)} = \text{starting time} + \mathrm{time}(L_{max}) + \text{finishing time} \tag{6}$$

In equation 6, starting time is the duration between the beginning of the program and beginning of the L_{max}, and finishing time is the duration between the termination of L_{max} and the termination of the program.

As explained in 4.2, we use optimal blocks for blocking iteration domains. In fact, with the assumption that we have enough hardware resources, the program starts running an iteration block as soon as its requirements are satisfied. As a result, we minimize the starting and finishing times, and we get the maximum possible overlap between the iteration blocks of different for-loop nests.

5 Implementation

We implement the algorithm explained in Section 4 as a part of Polly [20] and use the ISL library [43] 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.

5.1 Analysis

In the analysis passes, we extend the definition of the SCoP to include information needed for pipelining. We use the iteration domains and memory access relations and compute the maps $\mathcal{E}_\mathsf{S}$, Q_S, and Q'_S for every statement in the SCoP, by using Algorithm 1.

5.2 Transformation and Scheduling

In this step, we use the pipeline information of the SCoP to find the new schedule tree. For each statement S, we transform its schedule to separate the loops iterating over the blocks determined by $\mathcal{E}_\mathsf{S}$, from the ones iterating inside each block. The reason is that each block is an atomic task, with its dependencies computed in Q_S array of maps.

We define the *pipeline loop* to be the inner-most loop that iterates over blocks. The critical property of a pipeline loop is that its body iterates inside the blocks. Therefore, each of its iterations is a single task. To summarize, our goal is to construct a new schedule tree that:

1- blocks iteration domains,
2- finds pipeline loops, and
3- attaches dependency information to each task.

To begin with, we want the pipeline dependency relations to be defined as functions of the induction variables of the loops iterating over blocks. Therefore, for each statement S, we construct a `pw_multi_aff_list` from the maps in the array Q_S and a `pw_multi_aff` from the map Q'_S. After that, we create a mark node from them to add to the schedule tree.

To construct the final schedule, we begin by creating two separate schedule trees: one for iterating over blocks and one for iterating inside each block. Then, we *expand* the first schedule tree with the second one.

Let $\mathcal{D}_{\mathcal{E}_\mathsf{S}}$ and $\mathcal{R}_{\mathcal{E}_\mathsf{S}}$ be the domain and the range of $\mathcal{E}_\mathsf{S}$, respectively. We first create a schedule domain node from $\mathcal{R}_{\mathcal{E}_\mathsf{S}}$. Then, we get the partial schedule of the identity map of $R_{\mathcal{E}_\mathsf{S}}$ and add the corresponding band node to the created domain node. This schedule tree iterates over the blocks in lexicographical order. The next step is to construct the expansion schedule tree for iterating inside the blocks. This time, we repeat the same process as above, and we use $D_{\mathcal{E}_\mathsf{S}}$ (instead of $R_{\mathcal{E}_\mathsf{S}}$) for creating the domain node and the partial schedule. At this step, we add the mark node containing pipeline dependency information. Note that this mark node is located before the band node iterating inside the block, and it can be used for finding the pipeline loop. To complete the expansion process, we need to provide the *contraction function* for mapping domain elements of the original schedule and domain elements of the expansion schedule. For this purpose, we use the map $\mathcal{E}_\mathsf{S}$, as it defines this relation by definition.

To summarize, Algorithm 2 is our final scheduling method.

Algorithm 2: Schedule tree computations

```
Input  : Pipeline information of statements in a scop
Output : Updated schedule tree
1  for all statements S in scop do
2      $\mathcal{D}_{\mathcal{E}_S}$ = Domain($\mathcal{E}_S$), $\mathcal{R}_{\mathcal{E}_S}$ = Range($\mathcal{E}_S$);
3      $ps_1$ = partial schedule(identity map($\mathcal{R}_{\mathcal{E}_S}$));
4      $ps_2$ = partial schedule(identity map($\mathcal{D}_{\mathcal{E}_S}$));
5      $m$ = mark node( $Q_S, Q'_S$ );
6      $node_1$ = domain node($\mathcal{R}_{\mathcal{E}_S}$);
7      $sch_1$ = insert partial schedule($ps_1, node_1$);
8      $node_2$ = domain node($\mathcal{D}_{\mathcal{E}_S}$);
9      $sch_2$ = insert partial schedule($ps_2, node_2$);
10     $sch_2$ = insert mark node($m, node_2$);
11     $contraction$ = union_pw_multi_aff($\mathcal{E}_S$));
12     $sch_S$ = expand($sch_1, sch_2, contraction$);
13 $sch$ = sequence($sch_{\forall S \in scop}$);
14 return $sch$
```

5.3 AST

In the AST generation phase, we use the schedule tree to create the AST. Specifically, we use the mark nodes in the schedule tree to annotate the AST. Listing 6 shows parts of the AST of the transformed program of Listing 3. In Listing 6, there exists a for-loop nest corresponding to each loop nest in the original program. The comments in lines 3, 11, and 16 are representatives of the AST annotations. They show that the for loops in lines 2, 10, and 15 are the pipeline loop in their loop nest. They also contain the pipeline dependency information for the block that follows them.

5.4 Code Generation

The main idea for the code generation phase is to extract the tasks, which are the bodies of pipeline loops, to function calls. Then by passing the extracted function along with its dependency information to a high-level function implemented in a framework capable of task parallelism, we can utilize the detected parallelism. In this prototype, we can generate code for programs with for-loop nests of depth at most two, with

```
for(c0=0; c0<N; c0+=1)
  for(c1=0; c1<N; c1+=1) {
    // task
    ...
    S(c0,c1)
    ...
}
if (N>=2) {
  for (c0=0; c0<N/2; c0+=1)
    for (c1=0; c1<N/2; c1+=1) {
      // task
      R(c0, c1);
    }
  for (c0=0; c0<N/2; c0+=1)
    for (c1=0; c1<N/2; c1+=1) {
      // task
      U(c0, c1);
      }
}
```

Figure 6. Example of the AST of pipelined program

only one task annotation per loop nest. However, considering loops in the general case is feasible, and it is a matter of further developing our code generation function.

To get the pipeline dependency information, we use the annotations of the AST and convert them to the format needed by the framework we are using. In this work, we use OpenMP `task` constructs with `depend` clauses. Remember that the annotation assigns a `pw_multi_aff_list` and a `pw_multi_aff` to each task. Using OpenMP terms, each member of the `pw_multi_aff_list` is an in-dependency of the task, and the `pw_multi_aff` is its out-dependency. To find pipeline dependency information of each task, we compute unique integer values from each in-dependencies and the out-dependency. Each piece is a vector that we convert to an integer. We multiply each dimension to a large enough integer and add them all to get a single integer. To distinguish between each pw_multi_affs, we pair an index with the integer we got in the previous step.

In the final step, we extract all loop nests that we want to pipeline in another function. This function is called in `omp parallel` and `omp single` pragmas to launch and initialize the tasks.

5.5 OpenMP Tasks

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 5.4. We also need the size of the input argument and the total number of statements that a task depends on them. Listing 7 shows the signature of this high-level function.

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

Figure 7. Signature of the function for creating tasks

To coordinate between tasks, we define a global integer pointer `dependArr` and initialize it with `NULL`. We treat the pointer `dependArr` as a linearized two-dimensional array, where each column corresponds for each statement, and each row is for a specific iteration block of that statement. We also define a variable, `writeNum` to keep the number of loop-nests in the program that are sources of other loop-nests. With these assumptions, each task writes in the location `[writeNumber*outDepend+outIdx]` and reads from the locations `[writeNumber*inDepend[i]+inIdx[i]]` of `dependArr`. Also, based on equation 1 and the definition of pipeline maps, for maintaining the correctness of the program, blocks of the same for-loop nest should run in order. To add this in-dependency, we use the fact that all tasks created from iterations of the same for-loop nest have the same function pointer. Therefore, we keep track of the number of tasks created from each loop nest in a global array, `funcCount`, and use the function pointer of each task to coordinate different blocks of that task. The code in Listing 8 illustrates the creation of a task in the general case.

```
void *taskInput = malloc(inputSize);
memcpy(taskInput, input, inputSize);
int *self = (int *) f;
#pragma omp task
depend(out:dependArr[writeNum*outDepend+outIdx])
depend(iterator(i=0:dependNum),in:dependArr
                [writeNum*inDepend[i]+inIdx[i]])
depend(in:self[funcCount[outIdx]-1])
depend(out:self[funcCount[outIdx]])
{
    f(taskInput);
    free(taskInput);
}
```

Figure 8. Function for creating task in OpenMP

6 Evaluation

This section shows the evaluation of our algorithm and prototype using two benchmark sets, where programs are compiled using the `Clang` compiler with the `O3` option, and all tests run on an x86_64 Intel quad-core processor with two threads per core, clocks at 2900.000 MHz.

In the first benchmark set, we want to show the improvements that cross-loop pipelining can make to the programs it is designed for; programs consist of a sequence of compute-intensive serial for-loop nests. For this benchmark, we simulate compute-intensive kernels by using the `next_prime` function of the GMP library [19]. The basic data structure, `gmp_data`, is an array of `mpz` (data structure for multi-precision integer in the GMP library), and it has `SIZE` elements. All loop nests have depth two, and the i^{th} loop nest of the program updates elements of the matrix $\mathtt{A}_i$ by calling a function that adds its input arguments element-wise and finds the $\mathtt{num}_i^{th}$ prime number after that (with `next_prime` function). The $\mathtt{A}_i$s are two dimensional $\mathtt{N} \times \mathtt{N}$ matrices of `gmp_data`. Kernels are designed such that Polly cannot parallelize the loops (loops are sequential) and the running time of the version optimized with Polly is comparable with the sequential version. Table 9 shows the properties of our experimental data. The *Specification* column shows the number of loop nests and the values of num_is. The *Memory access* column shows the read access of each statement from the arrays written in the previous loop nests (lower and upper bounds of the loops are set accordingly). In this column, `Si` is the statement in the i^{th} loop nest.

Recall that blocks of one for-loop nest should run sequentially. Therefore, for a program with n loop nests, there can be at most n tasks running in parallel. Figure 10 shows the pipelined program's speed-up compared with the sequential program for different values of `N` and `SIZE`. From Table 9 and Figure 10, we can see that cross-loop pipelining always gains speed-up; however the amount of it depends on the loops access patterns.

In the second benchmark set, we use different variants of a sequence of matrix multiplication, the `2mm` and `3mm` benchmarks of Polybench [38] followed by the similar kernel `4mm`. To be able to generate code and also to make it a more suitable application of our framework, we consider matrix multiplication as consecutive vector-matrix multiplications. Our goal in this benchmark is to illustrate the advantages and disadvantages of cross-loop pipelining compared to Polly (Pluto's scheduling algorithm).

For `n=2,3,4`, the `nmm` and `nmmt` kernels are `n` consecutive matrix multiplications; in `nmmt` kernels, the second matrix is transposed beforehand. Similarly, the `ngmm` and `ngmmt` kernels are generalized matrix multiplication. Wherein the loop nest, each element of the result matrix (e.g. `C[i][j]`) is multiplied by the addition of the element of the result matrix (`C`) in the same column of the next row (`C[i+1][j]`) and the element in the same row of the previous column (`C[i][j-1]`). Figure 11 shows the *logarithm* of speed-ups of the programs generated by applying cross-loop pipelining (`pipeline`), Polly running with all available threads (`polly_8`), and Polly running with `n` threads (`n` is the number of loop nests) (`polly`), with respect to the sequential version.

Name	Specifications	Memory access
P1	2 for-loop $\mathtt{num}_{1,2} = 1$	$\mathtt{S2} \leftarrow \mathtt{A}_1\mathtt{[i][j]}$
P2	2 for-loop $\mathtt{num}_1 = 2$ $\mathtt{num}_2 = 6$	$\mathtt{S2} \leftarrow \mathtt{A}_1\mathtt{[2*i][2*j]}$
P3	3 for-loop $\mathtt{num}_{1,2,3} = 1$	$\mathtt{S2, S3} \leftarrow \mathtt{A}_1\mathtt{[i][j]}$ $\mathtt{S3} \leftarrow \mathtt{A}_2\mathtt{[i][j]}$
P4	3 for-loop $\mathtt{num}_{1,2} = 2$ $\mathtt{num}_3 = 8$	$\mathtt{S2} \leftarrow \mathtt{A}_1\mathtt{[i+j][j]}$ $\mathtt{S3} \leftarrow \mathtt{A}_1\mathtt{[2*i+j][2*j]}$ $\mathtt{S3} \leftarrow \mathtt{A}_2\mathtt{[2*i][2*j]}$
P5	4 for-loop $\mathtt{num}_{1,2,3,4} = 1$	$\mathtt{S2, S3, S4} \leftarrow \mathtt{A}_1\mathtt{[i][j]}$ $\mathtt{S3, S4} \leftarrow \mathtt{A}_2\mathtt{[i][j]}$ $\mathtt{S4} \leftarrow \mathtt{A}_3\mathtt{[i][j]}$
P6	4 for-loop $\mathtt{num}_1 = 1$ $\mathtt{num}_2 = 8$ $\mathtt{num}_{3,4} = 32$	$\mathtt{S2, S3, S4} \leftarrow \mathtt{A}_1\mathtt{[i+j][j]}$ $\mathtt{S3, S4} \leftarrow \mathtt{A}_2\mathtt{[i][j]}$ $\mathtt{S4} \leftarrow \mathtt{A}_3\mathtt{[i][j]}$
P7	4 for-loop $\mathtt{num}_1 = 1$ $\mathtt{num}_{2,3,4} = 8$	$\mathtt{S2, S3} \leftarrow \mathtt{A}_1\mathtt{[2*i][2*j]}$ $\mathtt{S3} \leftarrow \mathtt{A}_2\mathtt{[2*i][2*j]}$ $\mathtt{S4} \leftarrow \mathtt{A}_1\mathtt{[i][j]}$ $\mathtt{S4} \leftarrow \mathtt{A}_2\mathtt{[i][j]}$
P8	4 for-loop $\mathtt{num}_{1,2,3,4} = 1$	$\mathtt{S2, S3} \leftarrow \mathtt{A}_1\mathtt{[i][j]}$ $\mathtt{S4} \leftarrow \mathtt{A}_2\mathtt{[i][j]}$
P9	4 for-loop $\mathtt{num}_{1,2,3,4} = 1$	$\mathtt{S2, S4} \leftarrow \mathtt{A}_1\mathtt{[i][2*j]}$ $\mathtt{S3} \leftarrow \mathtt{A}_1\mathtt{[i][j]}$ $\mathtt{S3} \leftarrow \mathtt{A}_2\mathtt{[i][2*j]}$ $\mathtt{S4} \leftarrow \mathtt{A}_2\mathtt{[i][j]}$
P10	4 for-loop $\mathtt{num}_1 = 1$ $\mathtt{num}_{2,3,4} = 2$	$\mathtt{S2} \leftarrow \mathtt{A}_1\mathtt{[i+j][j]}$ $\mathtt{S3} \leftarrow \mathtt{A}_1\mathtt{[i][j]}$ $\mathtt{S4} \leftarrow \mathtt{A}_2\mathtt{[i][j]}$

Figure 9. Properties of the experimental data. The *Specification* column shows the number of loop nests and values of num_is. The *Memory access* column shows the read accesses of each statement.

As Figure 11 shows, the speed-up we gain by using Polly is more than that by cross-loop pipelining in the `nmm` and `nmmt`. For these kernels, Polly can optimize locality by tilling, and it also parallelizes all loop nests. However, in the `gnmm` and `gnmmt` kernels, Polly cannot detect any optimization, but by using cross-loop pipelining, we can gain speed-up.

7 Conclusion and Future Works

In this work, we developed a polyhedral model-based algorithm for detecting cross-loop task parallelism. We implemented our algorithm as part of LLVM/Polly. With this prototype, we can detect parallelization opportunities that conventional polyhedral optimizers cannot detect. We exploit the detected parallelism using OpenMP task constructs. We tested our prototype on kernels with compute-intensive

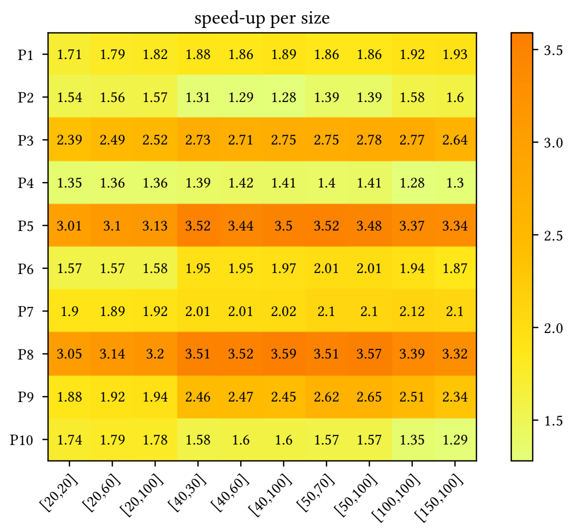


Figure 10. Speed-up of the tests from Table 9, considering different values for `N` and `SIZE`, comparing sequential version and pipelined version

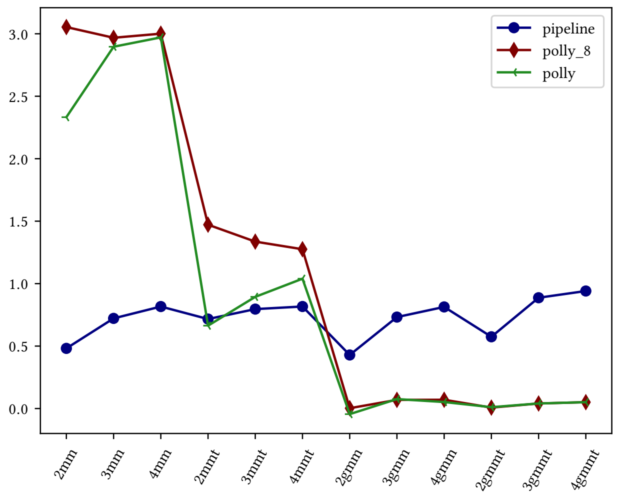


Figure 11. Comparing logarithm of speed-up gains of Polly running by all available threads, Polly running by `n` threads (`n` is the number of loop nests), and cross-loop pipelining for variants of generalized matrix multiplication.

function calls inside for-loop nests and reported the speed-ups considering different sizes and different memory access patterns. We also considered kernels with variants of a sequence of generalized matrix multiplications and compared the speed-ups of cross-loop pipelining and Polly.

We plan to generalize our code generation phase to generate code for loops with arbitrary depth and the number of tasks per loop. After this generalization, we can experiment with more complicated algorithms.

Also, as mentioned in Section 4, we assume that the write functions are injective; we want to study the possibilities of extending the transformation algorithm to relax this assumption. Moreover, we would like to extend both the transformation algorithm and the prototype to work correctly with the algorithms that detect DOACROSS parallelism in loops.

An essential factor in the performance of the final program is the granularity of the tasks. An interesting idea would be to develop an algorithm to choose a good task granularity when there are multiple choices.

The system's design is so that the tasking layer is independent of creating and scheduling the task. Therefore, we expect to be able to change the tasking layer from the OpenMP `task` to other platforms with minimal changes. For future works, we would like to experiment with this idea and have results in both performance improvements of different tasking platforms and how easy it is to use our method and make it compatible with other platforms.

In the current version of this work, when using the cross-loop tasking, we do not take advantage of other parallelization opportunities. We would like to know the effect of the cross-loop pipelining on the other patterns and study the results of possible combinations of this method with other optimization techniques on the performance improvements.

As explained in Section 2, an important application of detecting task relations in a program is mapping data to data-flow architectures. They may need to have the relation between LLVM level instructions of a program. It is an interesting idea to consider the transformation algorithm of this paper to lower-level instructions.

References

[1] Aravind Acharya and Uday Bondhugula. 2015. Pluto+: Near-complete modeling of affine transformations for parallelism and locality. *ACM SIGPLAN Notices* 50, 8 (2015), 54–64.

[2] Christophe Alias, Alain Darte, and Alexandru Plesco. 2013. Optimizing remote accesses for offloaded kernels: Application to high-level synthesis for FPGA. In *2013 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 575–580.

[3] Cédric Bastoul. 2004. Code generation in the polyhedral model is easier than you think. In *Proceedings. 13th International Conference on Parallel Architecture and Compilation Techniques, 2004. PACT 2004.* IEEE, 7–16.

[4] Cédric Bastoul, Albert Cohen, Sylvain Girbal, Saurabh Sharma, and Olivier Temam. 2003. Putting polyhedral loop transformations to work. In *International Workshop on Languages and Compilers for Parallel Computing*. Springer, 209–225.

[5] Mohamed-Walid Benabderrahmane, Louis-Noël Pouchet, Albert Cohen, and Cédric Bastoul. 2010. The polyhedral model is more widely applicable than you think. In *International Conference on Compiler Construction*. Springer, 283–303.

[6] Guy E Blelloch and Margaret Reid-Miller. 1999. Pipelining with futures. *Theory of Computing Systems* 32, 3 (1999), 213–239.

[7] Uday Bondhugula, A Hartono, J Ramanujam, and P Sadayappan. 2008. Pluto: A practical and fully automatic polyhedral program optimization system. In *Proceedings of the ACM SIGPLAN 2008 Conference on Programming Language Design and Implementation (PLDI 08), Tucson, AZ (June 2008)*. Citeseer.

[8] Jianjiang Ceng, Jerónimo Castrillón, Weihua Sheng, Hanno Scharwächter, Rainer Leupers, Gerd Ascheid, Heinrich Meyr, Tsuyoshi Isshiki, and Hiroaki Kunieda. 2008. MAPS: an integrated framework for MPSoC application parallelization. In *Proceedings of the 45th annual Design Automation Conference*. 754–759.

[9] Prasanth Chatarasi, Jun Shirako, and Vivek Sarkar. 2015. Polyhedral optimizations of explicitly parallel programs. In *2015 International Conference on Parallel Architecture and Compilation (PACT)*. IEEE, 213–226.

[10] Keith Cooper, Ken Kennedy, and Nathaniel McIntosh. 1996. Cross-loop reuse analysis and its application to cache optimizations. In *International Workshop on Languages and Compilers for Parallel Computing*. Springer, 1–19.

[11] Daniel Cordes, Andreas Heinig, Peter Marwedel, and Arindam Mallik. 2011. Automatic extraction of pipeline parallelism for embedded software using linear programming. In *2011 IEEE 17th International Conference on Parallel and Distributed Systems*. IEEE, 699–706.

[12] Leonardo Dagum and Ramesh Menon. 1998. OpenMP: an industry standard API for shared-memory programming. *IEEE computational science and engineering* 5, 1 (1998), 46–55.

[13] Roshan Dathathri, Ravi Teja Mullapudi, and Uday Bondhugula. 2016. Compiling affine loop nests for a dynamic scheduling runtime on shared and distributed memory. *ACM Transactions on Parallel Computing (TOPC)* 3, 2 (2016), 1–28.

[14] Paul Feautrier. 1988. Parametric integer programming. *RAIRO-Operations Research* 22, 3 (1988), 243–268.

[15] Paul Feautrier. 1991. Dataflow analysis of array and scalar references. *International Journal of Parallel Programming* 20, 1 (1991), 23–53.

[16] Paul Feautrier. 1992. Some efficient solutions to the affine scheduling problem. I. One-dimensional time. *International journal of parallel programming* 21, 5 (1992), 313–347.

[17] Paul Feautrier. 1992. Some efficient solutions to the affine scheduling problem. Part II. Multidimensional time. *International journal of parallel programming* 21, 6 (1992), 389–420.

[18] Michael I Gordon, William Thies, and Saman Amarasinghe. 2006. Exploiting coarse-grained task, data, and pipeline parallelism in stream programs. *ACM SIGPLAN Notices* 41, 11 (2006), 151–162.

[19] Torbjörn Granlund and the GMP development team. 2012. *GNU MP: The GNU Multiple Precision Arithmetic Library* (5.0.5 ed.). http://gmplib.org/.

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

[21] Zia Ul Huda, Rohit Atre, Ali Jannesari, and Felix Wolf. 2016. Automatic parallel pattern detection in the algorithm structure design space. In *2016 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 43–52.

[22] Wayne Kelly, Vadim Maslov, William Pugh, Evan Rosser, Tatiana Shpeisman, and David Wonnacott. 1995. The omega library interface guide.

[23] Kornilios Kourtis, Martino Dazzi, Nikolas Ioannou, Tobias Grosser, Abu Sebastian, and Evangelos Eleftheriou. 2020. Compiling Neural Networks for a Computational Memory Accelerator. (2020).

[24] Monica Lam. 1988. Software pipelining: An effective scheduling technique for VLIW machines. In *Proceedings of the ACM SIGPLAN 1988 conference on Programming Language design and Implementation*. 318–328.

[25] Chris Arthur Lattner. 2002. *LLVM: An infrastructure for multi-stage optimization*. Ph. D. Dissertation. University of Illinois at Urbana-Champaign.

[26] I-Ting Angelina Lee, Charles E Leiserson, Tao B Schardl, Zhunping Zhang, and Jim Sukha. 2015. On-the-fly pipeline parallelism. *ACM Transactions on Parallel Computing (TOPC)* 2, 3 (2015), 1–42.

[27] Feng Li, Antoniu Pop, and Albert Cohen. 2012. Automatic extraction of coarse-grained data-flow threads from imperative programs. *IEEE Micro* 32, 4 (2012), 19–31.

[28] Junyi Liu, John Wickerson, Samuel Bayliss, and George A Constantinides. 2017. Polyhedral-based dynamic loop pipelining for high-level synthesis. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 37, 9 (2017), 1802–1815.

[29] Vincent Loechner. 1999. PolyLib: A library for manipulating parameterized polyhedra.

[30] Antoine Morvan, Steven Derrien, and Patrice Quinton. 2013. Polyhedral bubble insertion: A method to improve nested loop pipelining for high-level synthesis. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 32, 3 (2013), 339–352.

[31] Manideepa Mukherjee, Alexander Fell, and Apala Guha. 2017. DFGenTool: A dataflow graph generation tool for coarse grain reconfigurable architectures. In *2017 30th International Conference on VLSI Design and 2017 16th International Conference on Embedded Systems (VLSID)*. IEEE, 67–72.

[32] Guilherme Ottoni, Ram Rangan, Adam Stoler, and David I August. 2005. Automatic thread extraction with decoupled software pipelining. In *38th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'05)*. IEEE, 12–pp.

[33] David Padua. 2011. *Encyclopedia of parallel computing*. Springer Science & Business Media.

[34] Josep M Perez, Vicenç Beltran, Jesus Labarta, and Eduard Ayguadé. 2017. Improving the integration of task nesting and dependencies in OpenMP. In *2017 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 809–818.

[35] Keshav Pingali, Donald Nguyen, Milind Kulkarni, Martin Burtscher, M Amber Hassaan, Rashid Kaleem, Tsung-Hsien Lee, Andrew Lenharth, Roman Manevich, Mario Méndez-Lojo, et al. 2011. The tao of parallelism in algorithms. In *Proceedings of the 32nd ACM SIGPLAN conference on Programming language design and implementation*. 12–25.

[36] Antoniu Pop and Albert Cohen. 2011. A stream-computing extension to OpenMP. In *Proceedings of the 6th International Conference on High Performance and Embedded Architectures and Compilers*. 5–14.

[37] Antoniu Pop and Albert Cohen. 2013. Openstream: Expressiveness and data-flow compilation of openmp streaming programs. *ACM Transactions on Architecture and Code Optimization (TACO)* 9, 4 (2013), 1–25.

[38] Louis-Noël Pouchet et al. 2012. Polybench: The polyhedral benchmark suite. *URL: http://www. cs. ucla. edu/pouchet/software/polybench* 437 (2012), 1–1.

[39] William Pugh and David Wonnacott. 1994. Static analysis of upper and lower bounds on dependences and parallelism. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 16, 4 (1994), 1248–1278.

[40] Harenome Razanajato, Cédric Bastoul, and Vincent Loechner. 2020. Pipelined Multithreading Generation in a Polyhedral Compiler. In *IMPACT 2020, in conjunction with HiPEAC 2020*.

[41] Alina Sbîrlea, Jun Shirako, Louis-Noël Pouchet, and Vivek Sarkar. 2015. Polyhedral optimizations for a data-flow graph language. In *Languages and Compilers for Parallel Computing*. Springer, 57–72.

[42] Nicolas Vasilache, Cédric Bastoul, and Albert Cohen. 2006. Polyhedral code generation in the real world. In *International Conference on Compiler Construction*. Springer, 185–201.

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

[44] Sven Verdoolaege. 2016. Integer set library: Manual. *Tech. Rep.* (2016).

[45] Sven Verdoolaege, Serge Guelton, Tobias Grosser, and Albert Cohen. 2014. Schedule trees. In *International Workshop on Polyhedral Compilation Techniques, Date: 2014/01/20-2014/01/20, Location: Vienna, Austria*.

[46] Tomofumi Yuki, Paul Feautrier, Sanjay Rajopadhye, and Vijay Saraswat. 2013. Array dataflow analysis for polyhedral X10 programs. In *Proceedings of the 18th ACM SIGPLAN symposium on Principles and practice of parallel programming*. 23–34.