ParameTrick: Coefficient Generalization for Faster Polyhedral Scheduling

Gianpietro Consolaro, Harenome Razanajato, Nelson Lossing, Denis Barthou, Zhen Zhang, Corinne Ancourt, Cedric Bastoul

ParameTrick: Coefficient Generalization for Faster Polyhedral Scheduling

Gianpietro Consolaro
Huawei Technologies France
Centre de Recherche en Informatique
Mines Paris - PSL University France

Harenome Razanajato
Huawei Technologies France

Nelson Lossing
Huawei Technologies France

Denis Barthou
Huawei Technologies France

Zhen Zhang
Huawei Technologies France

Corinne Ancourt
Centre de Recherche en Informatique
Mines Paris - PSL University France

Cedric Bastoul
Huawei Technologies France

Abstract

Polyhedral Schedulers have been widely used for loop optimization in general-purpose compilers and, more recently, deep-learning compilers. State-of-the-art scheduling algorithms define a vast space of loop transformations and find an optimal solution according to pre-designed cost functions. However, this Integer Linear Programming (ILP) approach can sometimes have complexity issues. The resolution time of ILP grows rapidly as the size of the problem increases. The complexity of the kernel to optimize, in terms of the number of statements, loops, and dependencies, has a significant impact on solving time. Additionally, the performance of ILP solvers can be severely impacted by big coefficients in the ILP, which mostly come from the loop bounds in the original kernel.

To tackle this issue, this paper introduces a technique called "ParameTrick". This technique is used during the scheduling pipeline to replace some large coefficients with parameters, which are new variables in the ILP model. The approach is analyzed to determine its impact on the solving time, as well as the transformations that are lost and how to limit the number of unfeasible transformations. Results show that ParameTrick leads to faster solving times while the space of dropped desirable solutions is limited.

Keywords: polyhedral scheduler, polyhedral optimization, integer linear programming

1 Introduction

The polyhedral model has been widely used in optimizing compilers. In recent years, it has become fundamental in deep learning frameworks such as Pytorch [7] (using Tensor Comprehension [11]) or MindSpore [4] (using AKG [13]). Polyhedral optimization can expose parallelism and improve data locality by exploiting different architectural features.

Polyhedral optimization relies on an algebraic representation, using polyhedrons (set of affine constraints) to express concepts such as domain (iteration space), dependencies (ordering relations between different iterations), and scheduling (complete execution order of the domain). This representation makes applying classic loop transformations (skewing, tiling, permutation, etc.) possible using linear algebra transformations.

The main challenge of this technique is to find a transformation that fully exploits architectural features to maximize performance. Polyhedral schedulers (such as Pluto [2][3], isl-scheduler [12], Feautrier [6]) use ILP (integer linear programming) to automatically find performant loop transformations that preserve semantics. The complexity of the scheduler consists of defining meaningful affine cost functions that guide the search for an optimal transformation. While this part of the research remains open, we tackle another aspect of the complexity of solving ILP problems in this paper. As polyhedral optimization is part of bigger complex compilation pipelines, it is required to minimize the time to find optimal transformations. ILP solving time can significantly increase with an increase in the number of constraint variables or the coefficient values. In polyhedral optimization, the number of constraints and variables grows with the number of statements and for-loops of the input program. Additionally, big loop-bound values and complex data accesses can result in larger ILP coefficients.

This paper presents *ParameTrick*, a simple preprocessing step that reduces complexity due to coefficients related to large loop-bounds. It may decrease the number of constraints in ILP problems and restrict the exploration space. Thanks to this trick, we will show how the compilation time can be

IMPACT 2024, January 17, 2024, Munich, Germany
2024.

consistently decreased while minimally limiting the transformation space. Our results first show the compilation speedup obtained in some of the PolyBench [9] dataset cases and some real cases from the MindSpore [4] pipeline. Furthermore, we show how, thanks to this technique, we can find solutions to problems that would normally become untreatable.

In Section 2, we will briefly describe polyhedral optimization. In Section 3, we describe *ParameTrick* technique and its limits. In Section 4, we show our result analysis on Polybench (Section 4.1) and some tests coming from MindSpore (Section 4.3). Finally, a summary and some future ideas are proposed in Section 5.

2 Background

Polyhedral optimization is a technique applied for loop transformations, and it can be divided into three main entities: polyhedral representation, polyhedral Scheduler, and polyhedral code generation.

- The polyhedral representation is an algebraic representation of the kernel to optimize, composed of domain, dependencies, and scheduling functions.
- The polyhedral Scheduler is an algorithm that takes the polyhedral representation as input and tries to find a new execution order (transformation) of the loop iterations. This is an automatic algorithm based on ILP formulation
- The code generation (not discussed in this paper) focuses on generating the code that respects the transformation found by the polyhedral scheduler.

2.1 Polyhedral Representation

The polyhedral representation chooses to represent the for-loop-based computations using polyhedrons (expressed as a set of constraints).

In the representation, we can find three main components.

2.1.1 Domain The domain expresses exactly the set of iterations executed by the kernel. This is represented as a polyhedron, where each constraint is an affine combination of iterators and (if needed) parametric loop bounds.

The Domain for a given statement S can be mathematically represented as follows:

$$\mathcal{D}_S(\vec{N}) = \left\{ \vec{it} \;\middle|\; M_S \cdot \begin{pmatrix} \vec{it} \\ \vec{N} \\ 1 \end{pmatrix} \geq 0 \right\}$$

where $\vec{it}$ is the vector of iterators surrounding the statement, $\vec{N}$ is a vector of parametric constants (normally used for the loop bounds), and M_S is a matrix of integer coefficients defining the polyhedron.

2.1.2 Dependency Polyhedral data dependencies describe ordering relationships between iterations. When accessing multiple times the same memory location in different iterations, if at least one of the accesses is a write, we have a dependency: a dependency $\delta_{S\to R}$ specifies that the statement S must be executed before the statement R to preserve the semantic.

The dependency is defined on a set of iterations of the two statements. The set of these constraints composes a polyhedron similar to the Domain:

$$\delta_{S\to R} = \left\{ \left(\vec{it}_S, \vec{it}_R \right) \;\middle|\; M_{S\to R} \cdot \begin{pmatrix} \vec{it}_S \\ \vec{it}_R \\ \vec{N} \\ 1 \end{pmatrix} \geq 0 \right\}$$

where $\vec{N}$ is a vector of parametric constants, $\vec{it}_S \in D_S(\vec{N})$ (iteration of Statement S), $\vec{it}_R \in D_R(\vec{N})$ (iteration of Statement R).

2.1.3 Scheduling Functions The scheduling function maps each statement iteration to a specific multi-dimensional date. The scheduling defines a total order using a lexicographic representation. Given a statement S, the scheduling function Θ_S is defined as a combination of scheduling dimension $\phi_{S,i}$:

$$\Theta_S : \begin{array}{ll} \mathcal{D}_S(\vec{N}) & \to \mathbb{N}^m \\ \vec{it} & \mapsto (\phi_{S,0}(\vec{it}) \; ... \; \phi_{S,m-1}(\vec{it})) \end{array}$$

where m is the number of scheduling dimensions, and $\phi_{S,i}$ are defined by:

$$\phi_{S,i}(\vec{it}) = T_{S,i} \cdot \begin{pmatrix} \vec{it} \\ \vec{N} \\ 1 \end{pmatrix} \tag{1}$$

where $T_{S,i}$ is the transformation vector, with $\vec{it}$ being the vector of iterators surrounding the statement S and $\vec{N}$ the vector of parametric constants of the kernel.

2.2 Polyhedral Scheduler

The polyhedral scheduler objective is to find a loop transformation, a scheduling function transformation that reorders the iteration execution, preserving the semantics while optimizing a pre-designed cost function. The cost functions normally focus on performance-related objectives, such as data locality and parallelism.

The Scheduling problem is expressed using ILP representation. Most of the known for-loop transformations can be represented simply through linear transformations of the initial schedule. Different state-of-the-art schedulers, mainly differ because of the different approaches (*iterative* like Pluto [2][3], isl-scheduler [12], and [6], or *one-shot* such as [10]), and different cost functions

The polyhedral scheduler aims to determine the optimal Θ_S, specifically the $\phi_{S,i}$ $(0 \leq i \leq m-1)$ that compose it. The variables of the ILP problem are composed of the coefficients

of the transformation matrix $T_{S,i}$ shown in Equation 1, plus some cost functions if necessary.

Among the cost functions and constraints defined in state of the art, the *validity* constraint is the fundamental one because it guarantees the transformation legality (semantic preservation).

2.2.1 Validity Constraint The validity constraint (introduced by Feautrier [6]) ensures that $T_{S,i}$ will preserve the semantics of the program. It is defined for each dependency $\delta_{S->R}$, and it constraints T to be scheduled after S.,

The constraint for the full scheduling function Θ_S and Θ_T can be represented as follows:

$$(\vec{it}, \vec{it}') \in \delta_{S\to R} \Rightarrow \Theta_R(\vec{it}') \succ \Theta_S(\vec{it})$$

The $\succ$ stands for lexicographically greater. Considering that each Θ is composed by a set of ϕ_i $(0 \leq i \leq m-1)$, the validity constraint expressed on the single dimensional component ϕ_i becomes:

$$(\vec{it}, \vec{it}') \in \delta_{S\to R} \Rightarrow \phi_{R,i}(\vec{it}') \geq \phi_{S,i}(\vec{it}) \tag{2}$$

that, thanks to the definition given in Equation 1, can be written as:

$$T_i^R * \begin{pmatrix} \vec{it}^R \\ \vec{N}^R \\ 1 \end{pmatrix} - T_i^S * \begin{pmatrix} \vec{it}^S \\ \vec{N}^S \\ 1 \end{pmatrix} \geq 0 \tag{3}$$

The two vectors T_i^S and T_i^R are the variables of our ILP problem, but as we may notice, the constraint is not linear. Farkas lemma must be applied to obtain linear constraints, as shown in [6].

3 ParameTrick

ParameTrick consists of a simple simplification applied during the Scheduling process. As shown in section 2, the polyhedral Scheduler solves ILP systems, composed of constraints such as the Validity one, coming from the dependencies. The complexity of the ILP depends on several factors:

- *number of dependencies*: the number of dependencies impacts the number of constraints. The more dependencies we have, the more constraints will compose the ILP.
- *number of statements and their dimensionality*: these two factors directly impact the number of variables of the ILP.
- *the numerical complexity of the constraints*: this depends on the numerical complexity of the dependencies. Big coefficients in the original dependencies may bring many complex constraints, especially considering that Farkas lemma and Fourier Motzkin Elimination are applied to the Validity constraint and other cost functions.

Kernel

```
for(i = 0; i <= 537; i++){
S: c[i] = b;
  for(j = 0; j <= 537; j++){
R:  d[i][j] = c[i];
  }
}
```

Domain

$$D_S = \begin{pmatrix} 1 & 0 \\ -1 & 537 \end{pmatrix} \begin{pmatrix} i^S \\ 1 \end{pmatrix} \geq 0$$

$$D_R = \begin{pmatrix} 1 & 0 & 0 \\ -1 & 0 & 537 \\ 0 & 1 & 0 \\ 0 & -1 & 537 \end{pmatrix} \begin{pmatrix} i^R \\ j^R \\ 1 \end{pmatrix} \geq 0$$

(a) Initial Kernel and Domains

Dependency $\delta_{S\to R}$

$$\begin{pmatrix} 1 & 0 & 0 & 0 \\ -1 & 0 & 0 & 537 \\ 0 & 1 & 0 & 0 \\ 0 & -1 & 0 & 537 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & -1 & 537 \\ 1 & -1 & 0 & 0 \\ -1 & 1 & 0 & 0 \end{pmatrix} \begin{pmatrix} i^S \\ i^R \\ j^R \\ 1 \end{pmatrix} \geq 0$$

Validity Constraint

$$\begin{pmatrix} 0 & 0 & 537 & -1 & 1 & 0 \\ 0 & 0 & 0 & -1 & 1 & 0 \\ -537 & 537 & 537 & -1 & 1 & 0 \\ -537 & 537 & 0 & -1 & 1 & 0 \end{pmatrix} \begin{pmatrix} t\_i^S \\ t\_i^R \\ t\_j^R \\ t\_1^S \\ t\_1^R \\ 1 \end{pmatrix} \geq 0$$

(b) Before ParameTrick

Dependency $\delta_{S\to R}$

$$\begin{pmatrix} 1 & 0 & 0 & 0 & 0 \\ -1 & 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & -1 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & -1 & 1 & 0 \\ 1 & -1 & 0 & 0 & 0 \\ -1 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \end{pmatrix} \begin{pmatrix} i^S \\ i^R \\ j^R \\ N \\ 1 \end{pmatrix} \geq 0$$

Validity Constraint

$$\begin{pmatrix} 0 & 0 & 1 & -1 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & -1 & 1 & 0 & 0 & 0 \\ -1 & 1 & 1 & -1 & 1 & 0 & 0 & 0 \\ -1 & 1 & 0 & -1 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & -1 & 1 & 0 \end{pmatrix} \begin{pmatrix} t\_i^S \\ t\_i^R \\ t\_j^R \\ t\_N^S \\ t\_N^R \\ t\_1^S \\ t\_1^R \\ 1 \end{pmatrix} \geq 0$$

(c) After ParameTrick (N = 537)

Figure 1. Simple kernel and respective domain representations **(a)**, the dependency and the corresponding validity constraints after Farkas-lemma and Fourier Motzkin Elimination **(b)**. Finally, the dependency and the validity constraints after applying ParameTrick **(c)**

ParameTrick addresses the third factor, reducing the numerical complexity of the ILP problem. It replaces all the dependency polyhedron's large constant coefficients (loop bounds) with new parameters (that will be added as new columns). This reduces the presence of large coefficients in the dependencies but increases the number of variables in the ILP.

If the transformation contains a parametric shifting (where the parameter has been introduced by *ParameTrick*), we finally substitute it with the original coefficient.

The pre-processing step significantly impacts the solving time of the ILP because it simplifies the constraint system. Additionally, this trick ensures that only actual dependencies are present because we use the real bounds during the dependency analysis, allowing for precise dependency detection. If we substituted the loop bounds before the dependency analysis, we could detect some extra dependencies.

This trick simplifies the ILP solver work but can lead to the exclusion of some transformations. This strongly depends on the constraints that we add for each parameter introduced. The naive method would be to add no information about the parameters. However, adding more constraints (for example, specifying that a parameter is greater than 0 or the relation between different parameters) allows us to find more solutions in general but increases the compilation time.

In Fig. 1, we show a basic kernel with the corresponding domain (a), dependency, and Validity constraint deriving from

Input Kernel

```
for(i = 0; i < 15; i++){
  for(j = 0; j < 10; j++){
S:  A[i][j] = 0;
  }
}
```

Collapsed Kernel

```
for(i = 0; i < 150; i++){
S:  A[i/10][i%10] = 0;
}
```

Initial Schedule

$\Theta_S = [i, j]$

Collapsed Scheduling

$\Theta_S = [10i + j]$

Figure 2. *Loop Collapsing example:* in this figure, we can see the loop collapsing transformation (top) and the corresponding schedule transformation (bottom)

dependency (b). As visible, even from a trivial dependency as the one in the example, we can see that the corresponding validity constraint is numerically complex. In Fig. 1(c), we show the corresponding dependency after applying *ParameTrick* (substituting the loop bound 537 with a parameter N) and the derived Validity constraint. The numerical complexity of the matrices (both the dependency one and the constraint one) decreased. On the other side, the number of variables increased. We need to add the parameters introduced by our technique for the dependency. For the validity constraint, we need to add, for each statement, a variable for each parameter introduced. These variables are used to obtain parametric shifting.

Even though the transformation looks trivial, it can strongly impact the total Scheduling compilation time. The majority of the scheduler time is spent inside the ILP solver. Simplifications as ParameTrick can tremendously decrease the solving time, especially when the input kernel contains many dependencies that could become untreatable when building the ILP problem.

3.1 Limitations

ParameTrick objective is to decrease the complexity of the ILP solver, but in reality, this can also reduce the space of possible transformations. In other words, *ParameTrick* narrows down the set of scheduling transformations that the scheduler can find, similar to how input kernels with only parametric loop bounds would limit the scheduler search space.

The main transformation that is dropped is loop collapse. Focusing on Fig. 2 example, we can see that the collapse scheduling transformation consists of a skewing where the skew coefficient is the initial loop bound.

If we apply *ParameTrick*, using, for instance, $N = 15$ and $M = 10$, we cannot find internally the same collapse. This should correspond to:

$$\Theta_S = [M \cdot i + j]$$

but $M \cdot i$ is not an affine transformation in the form described in Equation 1.

When we substitute the loop bounds with parameters, we lose several transformations because of the loss of precise information. To preserve some of these transformations, we can introduce simple domain constraints, for instance, $N > 0$, and relational constraints like $N > M$. However, even with these constraints, it may not always be enough, especially with more complex dependencies. In some cases, it may be necessary to know the exact difference between the parameters or other relations, but this would increase the complexity of the problem, thereby losing some of the benefits of our approach.

4 Results

4.1 PolyBench

In this section, we present the results of our experiments on PolyBench [9] benchmark, where we applied *ParameTrick* to improve the compilation time. We used PolyTOPS [5] scheduler, specifying the Pluto-like configuration (based on Pluto [2][3]), using the ILP solver provided by isl-0.25 for the experiments.

PolyBench is a benchmark containing 30 kernels from different domains, such as data mining, linear algebra, stencil applications, etc. This benchmark has been extensively used for comparisons in the state-of-the-art schedulers because it contains the most common characteristics of different domains. Notice that, in our benchmarks, we excluded *adi, nussinov, deriche,* and *ludcmp* because they contain a reversed loop. Our scheduler, similarly to Pluto, cannot schedule such a case, and it would simply fall back to the initial schedule.

All the PolyBench cases are, by default, parametric. To show the benefits of *ParameTrick* in our experiment, we applied a preprocessing that substitutes the parametric loop bounds with the corresponding constants (using the values provided by the *Large dataset* defined in Polybench). This means that the whole pipeline, including the final code generation, is applied to the non-parametric version of the programs. Moreover, ParameTrick introduces parameters only internally to the schedule, but we substitute them with the corresponding constant in case of parametric shifting.

In the results, we decided to show three different cases:

- *Original*: for this case, we did not apply *ParameTrick*.
- *p-trick*: in this case, we applied *ParameTrick*, adding the inequalities specifying that our parameters are positive.
- *p-trick-extra*: for this evaluation, we use *ParameTrick*, adding inequalities on the positivity of the parameters and constraints specifying the relationship (greater or smaller) between the different parameters.

For the experimental evaluation, we used a workstation with Intel Xeon E5-2683 CPU (x86 64), with 2 sockets with 16 cores each (2 threads for each core). 80 MiB of L3 cache. The compiler is gcc-10.5

4.1.1 Compilation Time Table. 1 shows the compilation time speedup applying *ParameTrick* on PolyBench test cases. The most relevant speedups are coming from cases with complex dependencies in general or a high number of statements

Case	Compilation			Execution	
	Original **Time** (ms)	**Speedup** (original / p-trick)	**Speedup** (original / p-trick-extra)	**Speedup** (original / p-trick)	**Speedup** (original / p-trick-extra)
3mm	>600000	**>3636.32**	>176.67	n.a.	n.a.
correlation	2201.4	**2.31**	0.09	**25.12**	1
cholesky	9032.07	**151.43**	147.72	2.05	**2.23**
lu	14774.03	263.04	**291.1**	1.61	1.61
floyd-warshall	11820.16	**391.19**	386.67	1.27	1.27
bicg	19.04	**0.91**	0.73	1	1
covariance	887.05	**5.63**	2.41	1	1
fdtd-2d	650.69	**6.86**	5.47	1	1
gemm	22.49	**1.01**	0.59	1	1
gemver	3282.38	**107.28**	106.12	1	1
gesummv	29.96	0.93	0.93	1	1
heat-3d	751.84	**4.41**	4.2	1	1
jacobi-1d	30.97	**1.23**	1.22	1	1
jacobi-2d	351.02	**5.33**	4.82	1	1
mvt	11.35	0.91	**0.94**	1	1
seidel-2d	51.89	**0.97**	0.9	1	1
syr2k	22.25	**0.99**	0.79	1	1
syrk	20.93	**0.92**	0.73	1	1
trisolv	23.15	**1.09**	1.08	1	1
trmm	31.14	**1.02**	0.61	1	1
2mm	2523.95	**35.93**	5.53	0.8	**1**
gramschmidt	274.84	**1.53**	0.32	0.79	0.79
symm	322.79	**4.55**	3.5	0.78	0.78
atax	59.94	**2.12**	1.28	0.76	**1**
doitgen	3984.24	**34.59**	7.6	0.08	0.08
durbin	195.04	**2.79**	2.67	0.00002	0.00002

Table 1. PolyBench original *compilation time* (**left**) and *execution time* (**right**) and speedup using two different versions of *ParameTrick*. **p-trick** when enabled with basic constraints. **p-trick-extra** when enabled with additional constraints between parameters. The highest speedups among the two methods are in bold. The timing is in milliseconds (ms). For these results, isl-solver has been used to solve the ILP problem.

or dimensionality. We can notice tremendous speedups in cases like *3mm*, *cholesky*, *doitgen*, *floyd-warshall*, *gemver*, *lu*.

In some cases, we can notice a slowdown, which can be explained by the fact that the kernels are really simple (and the compilation time is paltry). In these cases, the numerical simplification is unimportant, while adding variables (by adding parameters) has a negative impact.

Furthermore, we can highlight how *p-trick* performs generally better than *p-trick-extra*. The last method introduces some extra information about the relation between different parameters, describing the dependency polyhedron more precisely. This allows us to find some solutions that cannot be found without this extra information, but it increases the complexity of the problem.

4.1.2 Execution Time In the previous subsection, we showed the strong impact of *ParameTrick* on the compilation time. For a complete analysis, we need to analyze the impact of the scheduling transformation that is found to understand what kind of opportunities are lost.

For these results, we simply compared the execution time of the different transformations found in the three different modalities (original, p-trick, and p-trick-extra). Our pluto-like scheduler uses the Proximity cost function and Validity constraint. Similarly to Pluto, it applies some post-processing, such as tiling (using a default size of 32), intra-tile optimization, and wavefront skewing when necessary. The final code is generated using Cloog [1] using OpenMP to generate parallel loops.

For cases like *doitgen, durbin, floyd-warshall, gramschmidt, symm, cholesky* the transformation found is different because, without *ParameTrick*, the scheduler would apply a loop collapse (or, more in general, a skewing depending on the loop bound). For *durbin* in particular, we can notice a relevant slowdown. In this case, without *ParameTrick* we

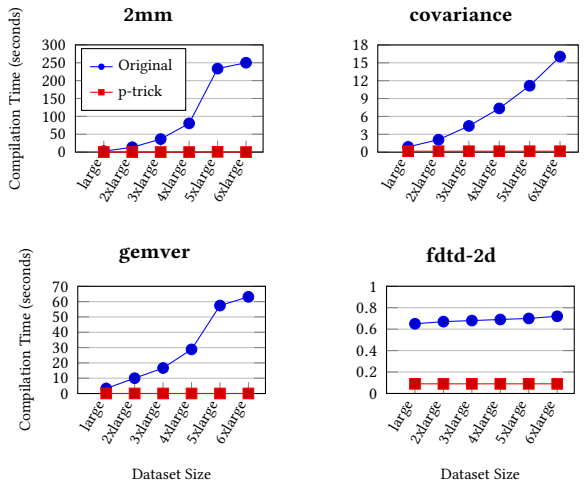

Figure 3. Compilation time (seconds), with (in red) and without (blue) *ParameTrick*, changing the dataset size (loop bounds). For the experiments, isl ILP solver has been used.

Case	Original Time (ms)	Time (ms) (p-trick)	Speedup (p-trick)
batch_norm	>600000	9764	**>61.45**
two2fractal_v1	96519	78	**1237.42**
two2fractal_v2	105	51	**2.05**
two2fractal_v3	344	28	**12.29**
maxpool_grad_v1	5583	2333	**2.39**
force_grad	381	180	**2.12**
max_pool_grad_v2	>600000	529	**>1134**
hpl_cholesky	9291	121	**76.78**
hpl_lu	29396	97	**303.05**

Table 2. Compilation time for some AKG [13] input cases. Original compilation time, compilation time using *ParameTrick* (**p-trick**), and the speedup. FPL-16.0.6 [8] has been used as ILP-solver. *The transformation found is the same when enabling/disabling ParameTrick for all the cases, so no performance analysis has been provided.*

can apply tiling and find parallelism, which is not possible when using *ParameTrick*. For *doitgen*, similarly to *durbin*, without *ParameTrick*, we can find external loop parallelism and tile more dimensions (obtaining bigger loop bands). For *correlation*, on the contrary, we obtain a significant speedup from some interchanges and different shifting choices. For *floyd-warshall* and *cholesky* instead, we obtain good speedup, even if the solution would be suboptimal if we do not apply *ParameTrick*.

For cases as *correlation, 2mm, and atax,* p-trick-extra can find the same original solution, but p-trick cannot because there is a shifting that cannot be applied without knowing the relation between the different parameters. Even though the different solution is sub-optimal in general, it is interesting to notice that in *correlation*, the sub-optimal solution (considering the Proximity cost function [2]) is in practice much faster in execution time. For *3mm*, verifying if the transformation found is the same was impossible because the original version takes an indefinite time. The scheduling transformation remains the same for all the other cases (where we obtain a speedup of 1).

4.2 Dataset Size Analysis

This section analyses the relationship between dataset size (i.e., loop bounds sizes) and compilation time (excluding *ParameTrick*). To conduct the tests, we begin with the *large* dataset sizes as defined in polybench and then increase the loop bounds by a factor indicated on the axis (for instance, *2xlarge* is equivalent to large, but with each loop bound multiplied by 2).

In Fig. 3, we can see that for *2mm, covariance, and gemver*, the compilation time is greatly impacted by the loop-bounds size. *ParameTrick* has the advantage of being *completely invariant* to this factor, remaining constant for all the possible loop bounds values. On the other hand, we can see that cases like *fdtd-2d* are not really impacted by the loop bounds variation.

The size analysis requires further analysis to be explained in detail because, as we can see, the compilation time growth changes depending on the input, suggesting that some dependencies (and some particular polyhedrons) are more variant to the increase of the loop bounds. Furthermore, we think GMP internal representation can play a fundamental role in these slowdowns, but we leave this analysis for the future since the GMP overhead is given by isl. To obtain a precise analysis, this will require a deep look into isl implementation, gathering information about GMP usage for each use case.

We want to remark that we used isl instead of FPL for this analysis because, in this case, isl outperforms FPL. We noticed that while FPL scales better than isl when the ILP number of variables increases, isl can handle the increase of numerical complexity better than FPL.

4.3 MindSpore AKG

The *ParameTrick* is not limited to simple benchmarks such as PolyBench and can be applied in real applications such as AKG [13] in MindSpore [4]. MindSpore is a deep-learning framework that can be used, similarly to Pytorch and Tensorflow, to define deep-learning models.

AKG is the deep-learning compiler used to lower and optimize the operators of the model. Its optimization relies on polyhedral optimization. Minimizing the compilation time without losing essential optimization opportunities is a key objective in deep learning compilers. A method such as *ParameTrick* has a lot of value in such an environment because it can tremendously reduce the full compilation time of deep learning models. The tradeoff is minimal, losing almost no important transformations, especially considering the

simplicity of deep learning operators, which rarely require complex transformations.

Table 2 shows the benefits obtained using *ParameTrick* to some realistic use cases, showing noticeable speedups. In these experiments, we used FPL-16.0.6 [8] as ILP-solver to show that the impact of *ParameTrick* is preserved even on a more recent solver. The cases shown come from the composition of real deep learning operators from some of the most used deep learning models, such as Resnet, Bert, and Transformers. For some cases, the compilation time is a few hundred milliseconds, but considering that a deep learning model can be composed by thousands of these operators, speedups such as the one showed in 2 are quite impressive. *In this scenario, the final scheduling transformation is the same*, enabling or disabling *ParameTrick*, so no further performance analysis is presented.

5 Conclusion

In this paper, we discussed a simple technique, *ParameTrick*, that can be applied to mitigate the compilation time spent during polyhedral scheduling. Our method temporarily substitutes numerical loop bounds with parametric ones during the scheduling phase. If the final scheduling transformation contains some parametric solutions, we substitute the parameters with the original constant. This simplifies the numerical complexity of the ILP formulated, achieving great speedup in the overall compilation time for several input cases from PolyBench and, similarly, for some cases from the AKG pipeline. The drawback of this technique is the reduction of the space of possible transformations that may lead to suboptimal solutions. We showed that in PolyBench, our technique can lose some good optimization opportunities, but in most cases, it does not affect the execution performance while drastically reducing the compilation time. Furthermore, thanks to our technique, we can deal with input problems that would be untreatable otherwise, as shown for *3mm* from PolyBench and in two cases coming from the AKG scenario. Our work shows the benefits of *ParameTrick* and indirectly highlights the different optimization opportunities lost when using such a technique or when the input kernel contains parametric bounds.

For future work, we strongly believe that a deep analysis of the impact of different loop bounds on the different ILP solvers' compilation time would be extremely interesting to understand better the benefits of *ParameTrick*. Moreover, some of the PolyBench examples highlighted how, in some cases, we can obtain better solutions (in execution time) that are suboptimal considering the Proximity cost function. In the future, we would like to investigate these cases to understand the limitations of Pluto's cost function and how to overcome them.

References

[1] Cedric Bastoul. 2004. Code Generation in the Polyhedral Model Is Easier Than You Think. In *Proceedings of the 13th International Conference on Parallel Architectures and Compilation Techniques (PACT '04)*. IEEE Computer Society, USA, 7–16. https://doi.org/10.1109/PACT.2004.1342537

[2] Uday Bondhugula, Albert Hartono, Jagannathan Ramanujam, and Ponnuswamy Sadayappan. 2008. A Practical Automatic Polyhedral Parallelizer and Locality Optimizer. In *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation*. ACM, Tucson, AZ, USA, 101–113. https://doi.org/10.1145/1375581.1375595

[3] Uday Bondhugula, J. Ramanujam, and P. Sadayappan. 2007. *PLuTo: A Practical and Fully Automatic Polyhedral Parallelizer and Locality Optimizer*. Technical Report OSU-CISRC-10/07-TR70. The Ohio State University.

[4] Lei Chen. 2021. *Deep Learning and Practice with MindSpore*. Springer Nature, Singapore. https://doi.org/10.1007/978-981-16-2233-5

[5] Gianpietro Consolaro, Zhen Zhang, Harenome Razanajato, Nelson Lossing, Nassim Tchoulak, Adilla Susungi, Artur Cesar Araujo Alves, Renwei Zhang, Denis Barthou, Corinne Ancourt, and Cedric Bastoul. 2024. PolyTOPS: Reconfigurable and Flexible Polyhedral Scheduler. arXiv:2401.06665

[6] Paul Feautrier. 1992. Some efficient solutions to the affine scheduling problem. I. One-dimensional time. *International Journal of Parallel Programming* 21 (1992), 313–347. https://doi.org/10.1007/BF01407835

[7] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. 2019. PyTorch: An Imperative Style, High-Performance Deep Learning Library. In *Advances in Neural Information Processing Systems*, H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett (Eds.), Vol. 32. Curran Associates, Inc., Vancouver, BC, Canada. https://proceedings.neurips.cc/paper/2019/file/bdbca288fee7f92f2bfa9f7012727740-Paper.pdf

[8] Arjun Pitchanathan, Christian Ulmann, Michel Weber, Torsten Hoefler, and Tobias Grosser. 2021. FPL: Fast Presburger Arithmetic through Transprecision. *Proc. ACM Program. Lang.* 5, OOPSLA, Article 162 (oct 2021), 26 pages. https://doi.org/10.1145/3485539

[9] Louis-Noël Pouchet et al. 2012. Polybench: The polyhedral benchmark suite. https://web.cse.ohio-state.edu/~pouchet.2/software/polybench/

[10] Louis-Noël Pouchet, Uday Bondhugula, Cédric Bastoul, Albert Cohen, J. Ramanujam, Ponnuswamy Sadayappan, and Nicolas Vasilache. 2011. Loop Transformations: Convexity, Pruning and Optimization. *ACM SIGPLAN Notices* 46 (05 2011), 549–562. https://doi.org/10.1145/1925844.1926449

[11] Nicolas Vasilache, Oleksandr Zinenko, Theodoros Theodoridis, Priya Goyal, Zachary DeVito, William S. Moses, Sven Verdoolaege, Andrew Adams, and Albert Cohen. 2018. Tensor Comprehensions: Framework-Agnostic High-Performance Machine Learning Abstractions. arXiv:1802.04730

[12] Sven Verdoolaege. 2010. Isl: An Integer Set Library for the Polyhedral Model. In *Proceedings of the Third International Congress Conference on Mathematical Software (ICMS'10)*. Springer-Verlag, Berlin, Heidelberg, 299–302. https://doi.org/10.1007/978-3-642-15582-6_49

[13] Jie Zhao, Bojie Li, Wang Nie, Zhen Geng, Renwei Zhang, Xiong Gao, Bin Cheng, Chen Wu, Yun Cheng, Zheng Li, Peng Di, Kun Zhang, and Xuefeng Jin. 2021. AKG: Automatic Kernel Generation for Neural Processing Units Using Polyhedral Transformations. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation* (Virtual, Canada) *(PLDI 2021)*. Association for Computing Machinery, New York, NY, USA, 1233–1248. https://doi.org/10.1145/3453483.3454106