PolyTOPS: Reconfigurable and Flexible Polyhedral Scheduler

Gianpietro Consolaro*‡, Zhen Zhang*, Harenome Razanajato*, Nelson Lossing*, Nassim Tchoulak*,
Adilla Susungi*, Artur Cesar Araujo Alves*, Renwei Zhang†, Denis Barthou*, Corinne Ancourt‡,
Cedric Bastoul*,
*Huawei Technologies France, Paris, France †Huawei Technologies Co., Ltd., Beijing, China
‡Mines-Paris PSL University

Abstract—**Polyhedral techniques have been widely used for automatic code optimization in low-level compilers and higher-level processes. Loop optimization is central to this technique, and several polyhedral schedulers like Feautrier, Pluto, isl and Tensor Scheduler have been proposed, each of them targeting a different architecture, parallelism model, or application scenario. The need for scenario-specific optimization is growing due to the heterogeneity of architectures. One of the most critical cases is represented by NPUs (Neural Processing Units) used for AI, which may require loop optimization with different objectives. Another factor to be considered is the framework or compiler in which polyhedral optimization takes place. Different scenarios, depending on the target architecture, compilation environment, and application domain, may require different kinds of optimization to best exploit the architecture feature set.**

We introduce a new configurable polyhedral scheduler, PolyTOPS, that can be adjusted to various scenarios with straightforward, high-level configurations. This scheduler allows the creation of diverse scheduling strategies that can be both scenario-specific (like state-of-the-art schedulers) and kernel-specific, breaking the concept of a one-size-fits-all scheduler approach. PolyTOPS has been used with isl and CLooG as code generators and has been integrated in MindSpore AKG deep learning compiler. Experimental results in different scenarios show good performance: a geomean speedup of 7.66x on MindSpore (for the NPU Ascend architecture) hybrid custom operators over isl scheduling, a geomean speedup up to 1.80x on PolyBench on different multicore architectures over Pluto scheduling. Finally, some comparisons with different state-of-the-art tools are presented in the PolyMage scenario.

Index Terms—**Polyhedral Optimization, Polyhedral Scheduling, Configurability, Flexibility**

I. INTRODUCTION

The polyhedral model has been widely used in modern optimizing compilers and frameworks for deep learning workloads, e.g., TC (Tensor Comprehension) [1] for PyTorch, AKG (Automatic Kernel Generator) [2] for MindSpore [3], nGraph [4] and Affine Dialect in MLIR [5]. Loop-based representations of computational kernels, combined with automatic mathematical (affine) transformations, enhance parallelism and data locality efficiency on the target hardware. This technique is characterised by its ability to systematically optimize execution time in most cases without (or with minimal) manual adjustment. It has been demonstrated to be successful on CPU, GPU and NPU (Neural Processing Unit), resulting in impressive performance improvements.

The core of the polyhedral optimization is the polyhedral scheduler, which applies affine transformations on the input for-loops according to its objective function. Some well-known polyhedral schedulers are the *Feautrier* scheduler [6], *Pluto* [7], *isl-scheduler* [8][9], and *Tensor* Scheduler [10]. They are characterized by the use of different cost functions, optimizing for different specific scenarios.

Polyhedral scheduler algorithms are based on mathematical optimization. Affine constraint systems and cost functions are constructed to maximize hardware efficiency by iteratively solving Integer Linear Programming (ILP) problems to find optimal transformations for each loop dimension. The lack of controllability and configurability makes it challenging to produce efficient transformations for new architectures or scenarios. For example, isl was used in the AKG project to target Huawei's NPU *Ascend architecture* [11]. It performs well in many cases but poorly in others. For instance, it cannot produce the transformation illustrated in Listing 1, which could easily be lowered on the vector unit of the NPU.

```
for ( i = 0;  i < 100;  i++){
  for ( j = 0;  j < 10;  j++){
0:  c[j][i] = a[j][i] * b;
1:  d[i][j] = e[i][j] * x;
  }
}
```

```
for ( j = 0;  j < 10;  j++)
  for ( i = 0;  i < 100;  i++)
0:  c[j][i] = a[j][i] * b;

for ( i = 0;  i < 100;  i++)
  for ( j = 0;  j < 10;  j++)
1:  d[i][j] = e[i][j] * x;
```

Listing 1: **Left**: original for-loop code. It is fully parallel and suitable for GPU architecture. The **Right** one is optimal for vectorization of the NPU (for both vectorized data loading and computing) thanks to the loop distribution and the interchange for the statement **0**. Pluto or isl may be able to find the loop distribution (specifying the correct fusion heuristic), but no interchange would be found.

Improving performance when state-of-the-art schedulers cannot find the optimal transformation can be attempted by adjust the initial scheduling results through additional passes, as can be seen, for example, in AutoPoly [12], the AKG module for Ascend backend. This method can be cumbersome, considering the complexity of finding optimal transformations that preserve the semantics. A recent idea, "Constraint Injection" [13], proposed to build an interface for the classical polyhedral scheduler that allows the injection of custom constraints or partial transformations. This approach can drive the scheduler to generate appropriate initial scheduling.

It shows execution time speed-ups of deep learning workloads on GPU. However, it is kernel-specific and only allows to optimize for the specific input case. Generalizing an expected optimization for any input kernel requires more engineering effort on pre-processing, e.g. dependency analysis, pattern matching on memory access, etc.

This paper proposes a novel design of a fully controllable iterative polyhedral scheduler: **PolyTOPS**. It allows the production of architecture-oriented optimizations (e.g. Listing 1) for any case from simple user configurations, and it can be easily adapted to new scenarios. PolyTOPS innovations are twofold:

- **Configurability**: PolyTOPS provides a rich expressivity on schedule strategies specification via an easy-to-use interface. All aspects of an iterative scheduling mechanism can be configured, e.g., parallelism control, vectorization, temporal and spatial locality, fine-grained controlling of statements loop fusion and fission, as well as partial schedule specifications. Moreover, the behaviour of PolyTOPS can be elegantly changed into a well-defined approach, e.g., Pluto-style, Feautrier-style, isl-style, Tensor-scheduler-style or extended to define novel strategies, e.g., scenario-specific, kernel-specific, extending the idea presented in [13].
- **Flexibility**: Instead of a single "one-size-fits-all" method, PolyTOPS exhibits a versatile design that can address scenario-specific optimizations. It is possible to start from a given generic strategy with little effort and then incrementally adjust this strategy for some particular loops of the kernel or for some particular architecture. PolyTOPS provides an extendable infrastructure for an iterative scheduler where constraints can be finely tuned – from predefined strategies down to dedicated transformation heuristics – for each statement and loop. We show how our approach can target multiple architectures (different types of CPU and Ascend NPU) and compare speedups with state-of-the-art schedulers.

We briefly introduce background notions for polyhedral schedulers in Section II. PolyTOPS design and implementation are detailed in Section III, and benchmark results on CPU and NPU are presented in Section IV.

II. BACKGROUND

Polyhedral optimization of kernels can roughly be decomposed into three stages. First, the input code and loop nests are represented as polyhedra. Then, algebraic transformations are applied to them, finally, a new optimized code that scans the transformed polyhedra is generated. In this section, we provide an overview of the techniques used in the first two stages and describe state-of-the-art methods.

A. Polyhedral Model

1) Iteration Domain: For each statement of the code, the iteration domain represents the range of values taken by the loop iterators surrounding this statement. The vector $\vec{it}$ composed by these iterators is called the *iteration vector*. Iteration domains

are assumed to be polyhedra of iteration vectors and can depend on parameters. The vector of parameters $\vec{N}$ is composed of variables that are constant during the execution of the code. The domain $\mathcal{D}$ of a statement S is defined as:

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

where M_S is a matrix defining the domain polyhedron.

2) Dependencies and Legality: A dependency $\delta_{S \to R}$ from statement S to statement R means that statement S needs to be executed before statement R to preserve the semantics of the program. This dependency is defined on a set of iteration vector values for S and R, with the following constraints: S and R access to the same memory location (either S or R is a write) and S is executed before R. These constraints, similarly to the domain of statements, define a polyhedron:

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

3) Scheduling Function: A scheduling function Θ maps each statement and iteration vector of its domain to a unique multi-dimensional date. Dates are totally ordered with the lexicographic order. Given a statement S, Θ_S is a multidimensional function defined dimension-wise by affine forms $\phi_{S,i}$. These affine forms depend on iterators $\vec{it}$ and parameters $\vec{N}$. Θ_S can be defined as follows:

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

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

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

where $T_{S,i}$ is the transformation vector.

For a statement S surrounded by k nested loops, at most $2k + 1$ dimensions [14] are necessary to express all possible scheduling transformations (strip mining is not expressed through the scheduler), but in practice, there is no upper limit on the number of dimensions.

B. Scheduler

The polyhedral scheduler is an algorithm that computes the scheduling function Θ. Two types of constraints govern the computation of this function. It has to preserve the semantics of the initial code and optimize some cost functions. Both types of constraints are integer affine constraints.

We now give an overview of the main components necessary to build the ILP problem, with the proximity cost function representing the most used cost function defined in the state-of-the-art:

1) Scheduling Problem Formalization: PolyTOPS is an *iterative* scheduler: The algorithm will find the full scheduling transformations Θ step by step, building an ILP problem to find each scheduling dimension $\phi_{S,i}$, starting from the outermost dimension until the innermost dimension. The algorithm makes sure to terminate when enough scheduling dimensions are found. The scheduler aims to find the optimal vector of coefficients $T_{S,i}$ for all S.

2) Validity/Legality Constraint: The validity constraint has been introduced by Feautrier [6]. This is the core of the polyhedral scheduler because it constrains the transformation vectors $T_{S,i}$ to have values that preserve the program semantic (ensuring the legality of the schedule). For each dependency $\delta_{S \to R}$, S has to be executed before R:

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

where the symbol $\succ$ stands for lexicographically greater.

Considering that for an iterative scheduler, each dimension $\phi_{S,i}$ is computed from the outermost to the innermost, the definition of validity becomes:

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

until the dependency $\delta_{S \to R}$ is satisfied. This implication can be linearized using the Farkas Lemma [15][6]: constraints are then expressed only on the space of variables composed by the vectors $T_{S,i}$ and $T_{R,i}$.

3) Progression Constraint: The progression constraint is added at each scheduling iteration to ensure the progression of the algorithm. Its role is to ensure that the schedule defines a complete order for the iteration space and to make sure that the trivial zero solution is avoided. The constraint definition forces the next scheduling solution to be linearly independent of previous solutions in the iteration space.

We define the matrix H_S as the concatenation (row by row) of previous scheduling dimension solutions $T_{S,i}$. We define the orthogonal complement $H_S^\perp$ as follows:

$$H_S^\perp = I - H_S^T (H_S H_S^T)^{-1} H_S$$

where I is the identity and H_S^T is the transposition of H_S.

The progression constraint, considering that we limit our scheduling search space in the positive orthant, is defined as the sum (row by row) of the orthogonal complement matrix as follows,

$$\forall i, H_{S,i}^\perp \cdot h_S^* \geq 0 \quad \wedge \quad \sum_i H_{S,i}^\perp \cdot h_S^* \geq 1 \quad (3)$$

with $H_{S,i}^\perp$ a row of $H_S^\perp$, and h_S^* the next solution to be computed.

4) Proximity Cost Function: The Proximity cost function was defined by Bondhugula *et al.* [7] in order to find among legal solutions the ones that optimize temporal locality.

The idea is to minimize the distance (in scheduling time) between multiple accesses to the same memory position. Data dependences describe multiple accesses to the same memory position, then the *Proximity* objective is to minimize the dependency distance. For a dependency $\delta_{S \to R}$, the constraint is defined by:

$$(\vec{it}, \vec{it}') \in \delta_{S \to R} \Rightarrow \phi_{i,R}(\vec{it}') - \phi_{i,S}(\vec{it}) \leq \vec{u}\vec{N} + w \quad (4)$$

where $\vec{u}$ and w are the cost functions to minimize.

Proximity accurately represents a useful transformation characteristic and, indirectly, favours the first dimensions to be parallel, with a dependency distance of 0.

C. State of the Art

In the polyhedral framework defined before, we briefly describe various state-of-the-art schedulers.

Feautrier's [6] scheduler is the first *iterative* polyhedral scheduler. The target is to optimize for single-core SIMD CPUs. The *Validity* constraint is combined with the *Feautrier* cost function. This cost function aims to find sequential outer dimensions that could carry as many dependencies as possible. This can lead to inner loop parallelism for SIMD vectorization exploitation.

Pluto [7] iterative scheduler introduces the *Proximity* cost function previously described. The aim is to exploit high parallelism in architectures like multi-core CPUs. A more recent version, *Pluto+* [16], extends features to support loop reversal and negative skewing and finds the solutions for some corner case problems that could not be solved by *Pluto*. *Pluto-lp-dfp* [17] is an extension resorting to linear programming instead of ILP. This relaxation decomposes the scheduling algorithm into a sequence of transformations, showing the potential benefits in terms of compilation time.

isl [8][9] iterative scheduler uses both the re-implementations of *Pluto* and *Feautrier* schedulers to maximize parallelism. If no external parallelism is found, the *Feautrier* cost function is applied to remove as many dependencies as possible and to find parallelism in subsequent dimensions.

Tensor [10] iterative scheduler is applied to tensor-based applications, such as AI, typically characterized by high parallelism and few dependencies. Their focus is the definition of the *Contiguity* cost function for cache spatial locality. It tries to find loop permutations that optimize memory access patterns. It achieves good results but is domain-specific, limiting scheduling transformations to loop interchanges only.

One-shot [14] is a scheduler that is not iterative and is computed by representing the whole multidimensional transformation $\Theta(S)$ as a single ILP problem. This formulation of the problem makes it easier to represent global constraints and cost functions over the full schedule Θ as opposed to iterative schedulers where constraints and cost functions are usually local to a single scheduling dimension ϕ_i. However, the large number of variables and constraints leads to scalability issues and extended compilation times. Extensions to the One-shot scheduler [18][19] propose addressing the complexity issue via a dictionary of cost functions and a cache mechanism of previously found optimal solutions.

Current research suggests that existing schedulers are tailored to a specific objective function, targeting some architecture.

Their behaviours are predetermined, and the available options do not provide the *flexibility* needed to achieve good results in different areas, for different architectures, and within different compilers. In AKG, for instance, the optimization target architectures are diverse, so any of the existing schedulers may be successful in some scenarios but not in others. The state-of-the-art still lacks configurability and flexibility.

Several recent works have explored more configurable approaches: PolyLingual [20] is a work-in-progress domain-specific language (DSL) for polyhedral schedulers. This DSL offers building block functions and types that ease the design of polyhedral schedulers. Expert knowledge is still required to fully design the scheduler logic, a new scheduler or any of the previously cited schedulers, but the set of scheduling strategies it offers should be very wide. However, the designer has to consider all potential edge cases and guarantee the algorithm's completeness.

A lower-level approach is to directly define the set of polyhedral transformations for a code, as proposed by Tiramisu [21]. In this case, the scheduler is replaced by an AI-guided search strategy among the combination of loop transformations given by the expert. In Clint [22] instead, a graphic interface allows the application of manual transformations directly to the polyhedron. Chlore [23] tackles the explainability problem and, given an input kernel and its transformed version, tries to recover the set of polyhedral transformations necessary to obtain the same transformed version. Finally "Constraint Injection" [13] proposes a way to inject simple constraints in the polyhedral scheduler, but it is essentially designed to target kernel-specific optimizations.

III. POLYTOPS

PolyTOPS is a configurable iterative polyhedral scheduler. The objective is to propose a flexible, easy-to-adjust tool to ensure that existing strategies, or a mix between existing strategies (generalizing what isl proposes), can be described with very little effort while still allowing the expert to guide the scheduler more precisely if needed be. To achieve this, PolyTOPS provides a general iterative scheduler scheme, where the strategy can be defined through a configuration file.

The workflow of PolyTOPS is described in Fig. 1. The main blocks are similar to schedulers such as isl [8] or Pluto [7]. Both the input and the output of PolyTOPS are polyhedral representations of the code. The main parts of the input are the initial schedule and the dependencies. They can be expressed as isl objects or in OpenScop format. The result of the core ILP-based scheduling algorithm may be further post-processed: this phase handles tiling, intra-tile optimization and skewing for wavefront parallelism (see [7, Section 5.3]). It is important to highlight that no tile-size decision is implemented in the core scheduler. Tile sizes must be externally provided for tiling to be applied. Finally, as PolyTOPS can output isl objects or an OpenScop representation, the code generation can then be done with tools or libraries such as isl or CLooG [24].

The significant contribution of PolyTOPS is in the configuration block, which supports two kinds of interfaces, JSON
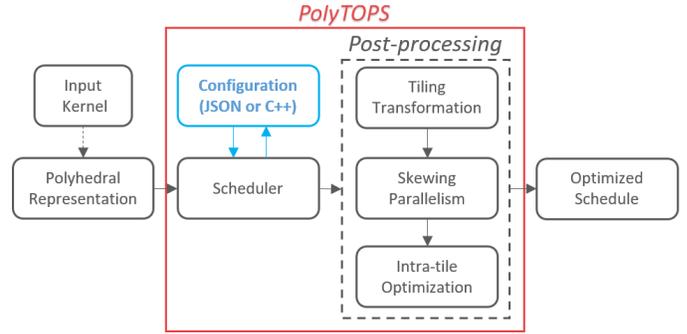

Fig. 1: PolyTOPS workflow representation, showing the major blocks, including the post-processing and the **configuration**. We support both Openscop and isl-representation as Polyhedral Representation

and C++. This feature allows the specification of high-level optimization strategies. Predefined or new strategies can be composed or extended through simple keywords. Further customization, down to kernel-specific strategies such as statement fusion or partial schedule specification, is also possible.

The configuration of PolyTOPS controls the scheduler for each specific scheduling dimension. For example, we could add cost functions for a specific dimension, distribute some statements in another dimension or add constraints to another one. The configurable features can be divided into two main types:

- *Local configurations*: They directly control the ILP creation. Predefined *cost functions* can be selected, and their priority order can be specified. New variables, ILP *constraints* or cost functions can be defined. Last but not least, it is possible to define the statement *distribution/fusion* for each dimension. It will be translated internally into specific constraints that will force the distribution specification.
- *Global Configurations*: These are higher-level features that do not only impact the definition of the ILP for a specific scheduling dimension. These features require several logic steps in the scheduling algorithm to be satisfied. For example, the *directives* are suggestions to the scheduler to attempt to vectorize or parallelize a specific loop. Another example is *AutoVectorization*, used to automatically detect (based on the memory stride and access pattern) what loops should be scheduled innermost for possible vectorization.

Let us now describe all possible configurations, starting from the locals and going on with the global ones.

A. Local Configurations

1) Cost functions control: A specific combination of predefined or new cost functions and their priorities (given by the textual order, from leftmost to rightmost) can be defined or omitted for each scheduling dimension (Listing 2 line 7). The objective function is the vector of variables. The order of the variables is important because they are minimized in lexicographic order.

```
1{
2  "scheduling_strategy" : {
3    "new_variables" : ["x"],
4    "ILP_construction" : [
5        {
6          "scheduling_dimension" : "default",
7          "cost_functions" : ["contiguity", "proximity", "x"]
8        }
9    ],
10   "custom_constraints" : [
11       {
12         "scheduling_dimension" : "default",
13         "constraints" : ["x - Si_it_i >= 0"]
14       }
15   ],
16   "fusion" : [
17       {
18         "scheduling_dimension" : 0,
19         "total_distribution" : false,
20         "stmts_fusion" : [["0", "1"], ["2"]]
21       }
22   ],
23   "directives" : [
24       {
25         "type" : "vectorize",
26         "stmts" : "0",
27         "iterator" : "1"
28       }
29   ]
30 }
31}
```

Listing 2: JSON example showing most of the configurable features of PolyTOPS, including cost function control and definition, constraints definition, fusion control and directives

New variables can be introduced, as shown in Listing 2 line 3, used in *custom constraint* definitions and as cost functions.

The predefined cost functions are `proximity` from Pluto [7], `feautrier` [6], `contiguity` inspired from the cost function defined in Tensor scheduler [10], and a new simple one named `bigLoopsFirst` that tries to schedule first (outermost) the loops with the largest iteration space.

The `contiguity` cost function is designed to schedule the iterators in the order that offers a better spatial locality. For each statement S, given the set of iterators $\vec{it}_S$, we define a cost function named `contiguity` as follows:

$$\texttt{contiguity}(S) = \sum_{i=0}^{|T_S^{it}|} T_{S,i}^{\vec{it}} \times c_{S,i} \qquad (5)$$

$$\texttt{contiguity}(S) \geq 0$$

where $c_{S,i}$ is a support coefficient that describes a priority order optimizing the memory access pattern. For instance, focusing on the kernel in Listing 1(left), the two vectors of support coefficient $\vec{c}_{S0}$ and $\vec{c}_{S1}$ would be respectively:

$$\vec{c}_{S0} = (10 \quad 1) \quad \text{with} \quad \vec{it}_{S0} = \vec{it}_{S1} = \begin{pmatrix} i \\ j \end{pmatrix}$$
$$\vec{c}_{S1} = (1 \quad 10)$$

to force the scheduler to select the outermost loops with the smallest contiguity coefficients.

The `bigLoopsFirst` (BLF) cost function is designed to schedule the loops with the largest domains outermost. The design is the same as the `contiguity` cost function, but the coefficients c_i used in Formula 5 are based on prioritizing the dimensions with the highest bounds first. BLF can be useful in scenarios where kernels have a lot of parallelism, in which only one level or a few levels of outer parallelism are exploitable by the architecture. We try then to maximize the number of parallel iterations. For instance, focusing on the kernel in Listing 1, the two vectors $\vec{c}_{S0}$ and $\vec{c}_{S1}$ would be respectively:

$$\vec{c}_{S0} = (1 \quad 10) \quad \text{with} \quad \vec{it}_{S0} = \vec{it}_{S1} = \begin{pmatrix} i \\ j \end{pmatrix}.$$
$$\vec{c}_{S1} = (1 \quad 10)$$

2) Custom Constraints: Using a simple interface, custom constraints are affine inequalities or equations. They can constraint the scheduling functions $\phi_{S,i}(\vec{it})$ defined by Eq. (1), for any statement S and dimension i through their vector of coefficients, $T_{S,i}$. We can separate this vector into subvectors $(T_{S,i}^{\vec{it}} \ T_{S,i}^{\vec{N}} \ T_{S,i}^1)$. A constraint can involve any of the coefficient of $T_{S,i}$ using the notation:

$$S[stmt]\_[var\_type]\_[idx\_var],$$

where i is implicitly defined as the current dimension considered (iterative scheduler) and:

- *stmt* is a statement number from 0 to $M - 1$ (where M is the number of statements), following the initial textual order. It identifies a unique statement S.
- *var_type* can be one of the following keywords: *it* refers to the subvector $T_{S,i}^{\vec{it}}$, *par* refers to the subvector $T_{S,i}^{\vec{N}}$ and *cst* refers to the constant term $T_{S,i}^1$.
- *idx_var* is the index of the variable. The outermost iterator of a statement is considered iterator 0. The order of the parameters is the textual order in the input program.

Additionally, any user-defined variable can be used in the constraints. Notice that replacing *stmt* or *idx_var* with the keyword 'i' represents the sum of all the variables of that type.

For example, a constraint that disables skewing for Statement 3 would be expressed by:

$$S3\_it\_i \leq 1.$$

This is equivalent to:

$$\sum_k T_{S3,k}^{\vec{it}} \leq 1.$$

Custom constraints can be defined either for specific scheduling dimensions or for all of them using the `fusion` keyword (see the example in Listing 2 lines 10-15).

The constraints that are accepted must be *affine*. This means that given the vector of variables $\vec{V}$ just described, it is possible to define all the constraints in the form:

$$constraint = A\vec{V} + c \quad \begin{matrix} \geq \\ = \end{matrix} \quad 0$$

where A is a matrix of integer coefficients, and c is an integer.

```
StrategyInfo strategy(last_solution, stmts, old_strategy){
    StrategyInfo new_strategy;
    if(!last_solution.parallel &&
       !old_strategy.recompute_last_solution) {
        new_strategy.recompute_last_solution = true;
        new_strategy.cost_functions = {"feautrier"}
    } else {
        new_strategy.cost_functions = {"proximity"};
    }
    return new_strategy;
}
```

Listing 3: An isl style configuration described using the C++ configuration. Feautrier is used as the fallback case when the Proximity fails to extract parallelism

3) Fusion/Distribution control: Custom loop-fusion decisions can be specified for a specific kernel. We give the ability to control the fusion, selecting which statements to fuse and which ones to distribute for each level. Listing 2, lines 16-22, shows a configuration example specifying that statements 0 and 1 are to be fused and statement 2 distributed at the scheduling dimension 0.

B. Global Configurations

1) Directives: Directives, Listing 2 lines 23-29, specify that certain loops should be `parallel`, or `vectorized` (scheduled innermost and not fused) or `sequential`. This can be used to suggest partial code transformations while the remaining scheduling transformation decisions are left to the scheduler. The scheduler will try to satisfy the directives unless scheduling legality can not be guaranteed. Directives that prevent legality preservation are discarded.

2) Auto Vectorization: This option instructs the scheduler to use a simple heuristic to detect dimensions that could be vectorized for each statement. The heuristic looks for dimensions that move contiguously in memory. The scheduler then computes a scheduling transformation where: (1) the vectorizable dimensions are scheduled as innermost and (2) the corresponding statements are unfused for this scheduling dimension. For architectures such as CPUs or NPUs, vectorization is critical for performance.

C. Configurations Strategy

The configurations can be specified using two different interfaces, each of them more suitable to different configuration scenarios:

1) JSON interface: The JSON interface, as seen in Listing 2, allows to tailor strategies for the input kernel. Local configurations are statically defined and mapped to scheduling dimensions. The configurations specify cost functions, extra constraints and possible loop distributions. However, this interface does not offer the freedom to define complex strategies that take outermost partial schedules into account.

2) C++ interface: In this configuration, the strategy is defined in a dynamic library that is loaded by PolyTOPS and called before each scheduling iteration. This enables a dynamic specification of each scheduling strategy, generalizing

isl [8] strategy, which calls a Pluto-style scheduler as default and a Feautrier-style scheduler as fallback. This example is shown in Listing 3. Furthermore, the strategy definition has access to many details concerning the statements and the partial schedule computed until the present iteration. This gives the opportunity to create more complex strategies.

The configuration is expressive enough to allow switching between different strategies like Pluto-style, Feautrier-style, isl-style, and TensorScheduler-style and define new ones. The only limit is that the configuration can only influence the core "Scheduler" block of Fig. 1. For instance, the main Pluto ILP strategy can be easily replicated using the configuration, but the post-processing and internal fusion heuristics cannot.

D. Common Algorithmic Structure

PolyTOPS relies on an algorithmic structure shown in Algorithm 1 that is common to the iterative schedulers, such as Feautrier's [6], Pluto [7], isl-scheduler [8] and Tensor Scheduler [10]). This is a generalization of Pluto algorithm, using the configuration strategy to drive the scheduler.

The termination criteria of the algorithm are to check if the iteration space is completely covered and if all the dependencies are fulfilled (*line 42*). The algorithm iterates to find a new scheduling dimension until the termination criteria are met (from *line 4 to line 42*). To compute the next dimension ϕ, the scheduler firstly verifies if the fusion heuristic (or the interface for PolyTOPS) imposes a loop distribution for this scheduling dimension (*lines 8-14*). If not, the algorithm continues with the standard step (*lines 16-21*), constructing the ILP system composed of the cost functions and constraints defined for the dependencies that are not yet completely satisfied. If no solution is found, the algorithm attempts to remove the dependencies satisfied by the previous scheduling dimension, and it continues building the ILP problem and trying to find a solution (*lines 23-30*). If all preceding steps fail, loop distribution is enforced by analyzing the strongly connected components (SCC) of the dependency graph and distributing the loop of different SCC (*lines 32-36*). Once the solution is found, it updates the progression constraint, ensuring that the next computed dimension of ϕ will be linearly independent from the previous ones and that the schedule is a bijective transformation. The algorithm computes $Bands$ and $ParallelDimension$. $Bands$ are used in post-processing tiling to determine which dimensions can be tiled. $ParallelDimension$ indicate which scheduling dimensions are parallel.

This algorithmic scheme covers all iterative schedulers of the literature just by defining the appropriate configurations. PolyTOPS extends them with the ability to select and define the cost functions and constraints (*lines 16, 26*). The JSON interface is expressed statically, so it is parsed once at the beginning of the scheduling algorithm, while the C++ interface allows for a logic-based decision using the information from the schedule Θ found so far (*line 6*), so it is updated for each scheduling iteration. For both cases, the configuration impacts loop fusion/distribution decision (*line 9*).

Algorithm 1: PolyTOPS Scheduler

Data: Input Dependencies *deps*, Statements *S*, Scheduling Configuration *config*

Result: Scheduling Θ $\forall S$, Tilability: **Bands**, Parallelism info for each level: **ParallelDimension**

```
 1  constraints ← CreateConstraints(config, deps);
 2  dimension ← 0;
 3  band ← 0;
 4  repeat
 5  │  if config.type = C++ then
 6  │  │  config ← UpdateConfiguration(Θ);
 7  │  end
 8  │  if config.Distribute(dimension) then
 9  │  │  φ ← Distribute(dimension, config);
10  │  │  Θ.Append(φ);
11  │  │  Bands.Append(band);
12  │  │  RemoveSatisfiedDependencies(deps);
13  │  │  /* Ends the current band        */
14  │  │  band ← band + 1;
15  │  else
16  │  │  ILP ← constraints(dimension);
17  │  │  φ ← Solve(ILP);
18  │  │  if φ ≠ ∅ then
19  │  │  │  Θ.Append(φ);
20  │  │  │  /* Same band as before        */
21  │  │  │  Bands.Append(band);
22  │  │  else
23  │  │  │  /* Change band and retry!      */
24  │  │  │  RemoveSatisfiedDependencies(deps);
25  │  │  │  band ← band + 1;
26  │  │  │  ILP ← constraints(dimension);
27  │  │  │  φ ← Solve(ILP);
28  │  │  │  if φ ≠ ∅ then
29  │  │  │  │  Θ.Append(φ);
30  │  │  │  │  Bands.Append(band);
31  │  │  │  else
32  │  │  │  │  φ ← UnfuseSCCs(deps);
33  │  │  │  │  Θ.Append(φ);
34  │  │  │  │  Bands.Append(band);
35  │  │  │  │  RemoveSatisfiedDependencies(deps);
36  │  │  │  │  band ← band + 1;
37  │  │  │  end
38  │  │  end
39  │  end
40  │  ParallelDimension.Append(φ.isParallel());
41  │  P ← ProgressionConstraint(Θ);
42  until P = ∅ && deps = ∅;
43  return Θ;
```

Legality constraints (Eq. (2)) and progression constraints (Eq. (3)) are always included when computing a solution, whatever the configuration provided. This implies that the scheduler always terminates (similar proof as the one for Pluto [7]). Moreover, the scheduler is guaranteed to find a valid schedule if no custom constraints and no fusion/distribution control are defined in the configuration. Indeed, strategies do not prevent finding a legal schedule, and directives are ignored when they conflict with the legality. Only the custom constraints and fusion/distribution may lead to an empty solution. This is different from approaches such as Tiramisu [21] or other approaches that do not use a scheduler since, in this case, each scheduling function obtained by composing transformations

TABLE I: **(Ascend 910 NPU)** Custom Operator results, showing the number of cycles for each case and the speedup obtained by PolyTOPS results over the isl ones

Case	Input/Output	isl (cycles)	PolyTOPS (cycles)	Speedup
LU decomp	16x16	27943	18333	1.52
trsmL off diag	16x16x16	15375	704	21.84
	16x16x32	31126	1122	27.74
	16x16x48	45172	1518	29.76
	16x16x64	62414	1938	32.21
	16x16x80	75611	2324	32.53
	16x16x96	93387	2724	34.28
	16x16x112	108384	3223	33.63
trsmU transpose	16x16x16	55370	22100	2.51
	16x32x16	107159	44298	2.42
	16x48x16	160547	64281	2.50
	16x64x16	212907	87914	2.42
	16x80x16	267627	106479	2.51
	16x96x16	317589	130221	2.44
	16x112x16	370941	151204	2.45

has to be proved valid.

IV. EXPERIMENTAL RESULTS

Our experiments focus on demonstrating the *flexibility* of PolyTOPS, capable of adapting to the different scenarios shown in the following session, and the expressiveness of the *configurability*, capable of changing the behaviour of the scheduler.

A. MindSpore Hybrid Custom Operators

In the first scenario, PolyTOPS is used in the context of MindSpore [3] for the generation of custom operators on NPU hybrid custom operators [25] for AI applications. The experiments are run on an Atlas 800 (model 9010) server featuring 8 Ascend 910 NPU accelerators [26][27]. The goal is to define custom operators differently from the default AI operators that are already predefined. When creating these operators, it is possible to express some *directives* passed through the AKG [2] compiler to PolyTOPS as part of the internal configuration. PolyTOPS schedules the operator and tries to comply with the provided directives. Table I shows the speedups obtained with such directives compared to isl (the default scheduler previously used in AKG [2]), both cases implemented in MindSpore and isl is used for code generation. The speedups are significant for all the 3 operators with all the different sizes, with a geomean speedup of 7.66x.

These results come from a manual specification, mostly focusing on vectorization directives that will end up applying interchanges and vectorizing innermost. We can see an example of one of the operators in Fig. 2a. In this case, the directives hint to vectorize the loop k. The result obtained is shown in Fig. 2b. The speedup obtained is because isl would detect k as parallel and schedules it as the outermost loop, thus losing the vectorization opportunity.

Although these results are achieved through manual directive specifications, we discovered that the same configuration file, enabling auto-vectorization and using the proximity cost

```
def trsmL_off_diag(a, b):
  inverse_0 = allocate(b.shape, b.dtype)
  row = b.shape[0]
  col = b.shape[1]
  for i in range(row):
    for j in range(i):
      for l in parallel(col // 16):
        for k in vectorize(16):
          inverse_0[i, l*16+k] = a[i, j] * b[j, l*16+k]
          b[i, l*16+k] = b[i, l*16+k] - inverse_0[i, l*16+k]
  return b
```

(a) Input code. Directives are displayed in red.

```
def trsmL_off_diag(a, b):
  inverse_0 = allocate(b.shape, b.dtype)
  row = b.shape[0]
  col = b.shape[1]
  for l in parallel(col // 16):
    for i in range(row):
      for j in range(i):
        for k in vectorize(16):
          inverse_0[i, l*16+k] = a[i, j] * b[j, l*16+k]
        for k in vectorize(16):
          b[i, l*16+k] = b[i, l*16+k] - inverse_0[i, l*16+k]
  return b
```

(b) Optimized code (before tiling) thanks to PolyTOPS

Listing 4: Custom Operator example.

```
{
  "scheduling_strategy" : {
    "ILP_construction" : [
      {
        "scheduling_dimension":"default",
        "cost_functions":["proximity"]
      }
    ],
  }
}
```

```
{
  "scheduling_strategy" : {
    "ILP_construction" : [
      {
        "scheduling_dimension":"default",
        "cost_functions":["contiguity",
                          "proximity"],
        "constraints":["no-skewing"]
      }
    ],
  }
}
```

Listing 5: JSON configurations showing pluto-style (on the left) and tensor-scheduler-style (on the right)

used in the *tensor-scheduler-style* strategy (with proximity as secondary). The no-skewing constraint is also applied (Listing 5 right). The isl-style strategy (Listing 3) defaults to proximity, and if no parallelism is found, it recomputes the scheduling dimension, resorting to Feautrier's strategy. These strategies are the same as their state-of-the-art counterparts regarding ILP construction and the primary objective. Our heuristic for the fusion strategy distributes statements with a different loop dimensionality (number of surrounding loops), similar to Pluto's smartfuse heuristic.

Last but not least, we show the result obtained using a *kernel-specific* configuration for each kernel. These configurations are obtained by playing with the cost functions, fusion decisions and vectorization directives, and they can change between different architectures and kernels.

Out of clarity, in Fig. 2, we removed the kernels nussinov, adi, deriche, ludcmp and floyd-warshall where the results are identical between Pluto and PolyTOPS. For the first 4 cases, both Pluto and PolyTOPS fall back to the initial schedule. Performance can be improved but it requires support for the negative scheduling coefficients. Floyd-warshall is too simple to obtain speedups applying loop transformations.

1) Results Analysis: Focusing on the charts in Fig. 2, we can see those heuristics like pluto-style, tensor-scheduler-style, and isl-style perform differently depending on the kernel. For example, isl-style performs well for stencil applications with complex dependencies like jacobi-2d, jacobi-1d, and heat-3d, where the Feautrier's fallback is crucial for parallelism. However, in other cases like correlation, covariance, durbin, lu, and trmm, isl-style performs poorly. Pluto-style and tensor-scheduler-style differ mainly in the no-skewing constraint. Pluto-style finds a complex skewing that enables parallelism in jacobi-1d, but the generated code is complex, degrading the overall performance compared to the tensor-scheduler-style solution. On the other hand, pluto-style outperforms tensor-scheduler-style in fdtd-2d because parallelism (that requires skewing) is crucial for performance improvements in this case. In some cases, tensor-scheduler-style performs better because of contiguity interchange.

As expected, the kernel-specific configuration outperforms or at least obtains the same speedup as the other strategies, obtaining an overall *geomean speedup* of **1.82** for *AMD*, **1.71** for *Intel1* and **1.76** for *Intel2*.

For *gramschmidt* (Intel1 and Intel2), in the kernel-specific configuration, thanks to a fusion decision based on maximizing the data reuse, we can find a better speedup compared to Pluto.

B. Comparing scheduling strategies on Polybench

The second part of our experiments is focused on the Polybench [28] benchmark. In this experimental section, we chose to compare PolyTOPS results against Pluto. This section uses CLooG [24] for code generation for all schedulers.

We repeated the tests in three different system configurations:

- **AMD**: AMD EPYC 7452, with 32 cores (2 threads for each core), 2 sockets. 256 MiB of L3 cache. The compiler version is gcc-11.3.
- **Intel1**: 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.
- **Intel2**: Intel Xeon Silver 4215 CPU (x86_64), with 2 sockets with 8 cores each (2 threads for each core). 22 MiB of L3 cache. The compiler is gcc-10.5.

Polybench contains heterogeneous kernels coming from different domains, such as linear algebra, data mining and stencil computation, and it represents a reference for the polyhedral optimization benchmarks. In our experiments, the performance obtained by PolyTOPS (using different configurations) is compared to Pluto, using the last development version (commit eddc385). For Pluto, the options `--parallel --tile --nounrolljam --no-diamond-tiling` are used. The last two are disabled because their post-processing is not available in PolyTOPS so far.

Our study of PolyTOPS showcases three general strategies for each kernel: The proximity cost function is used in the *pluto-style* strategy (Listing 5 left), while contiguity cost function is

function, could systematically be used for all kernels for all sizes. This suggests that the ability, through configurations, has the potential not only to obtain kernel-specific optimizations but also to generate effective heuristics for groups of use cases or specific scenarios.
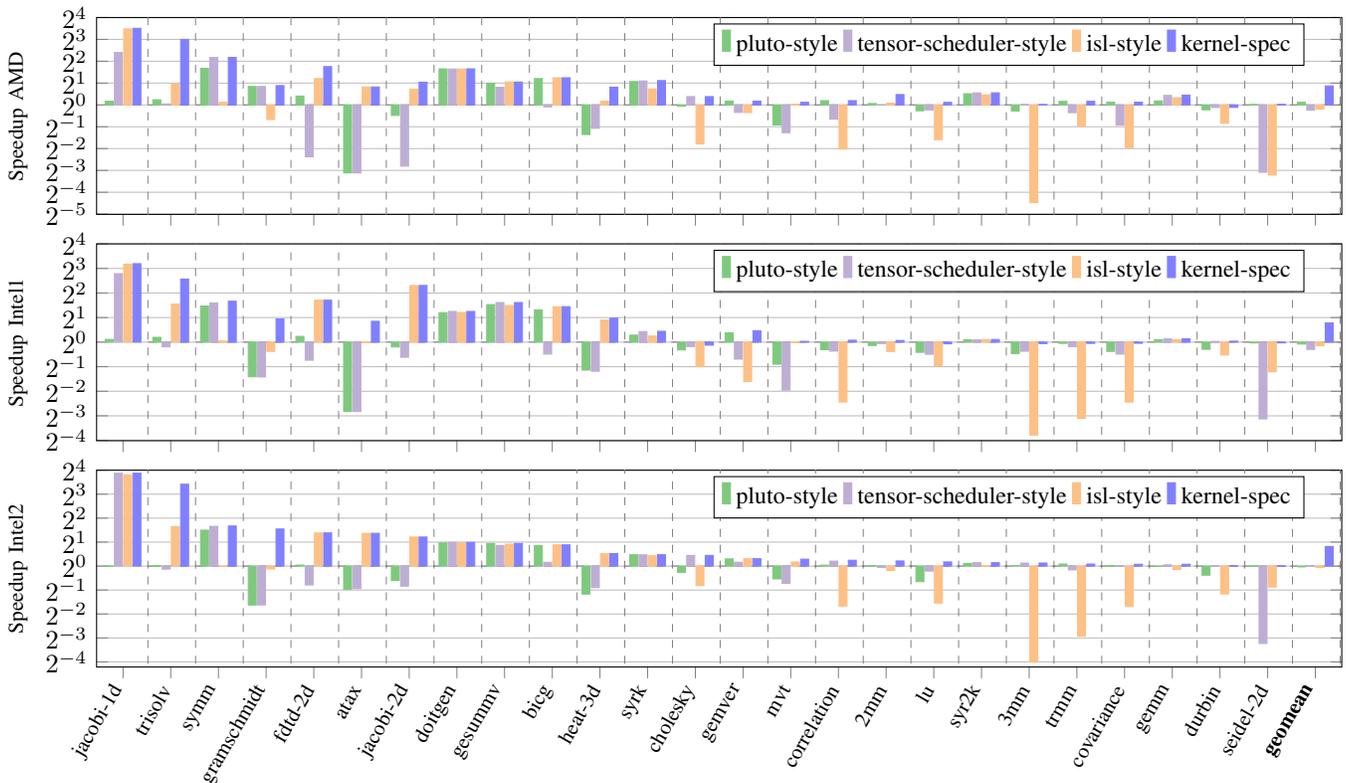
Fig. 2: Speedups (in log scale) of PolyTOPS (using 4 different configurations, *pluto-style*, *tensor-scheduler-style*, *isl-style* and *kernel-specific*) compared to Pluto. The *kernel-specific* configuration is at least as good as the three previous ones. The results are sorted by decreasing *kernel-specific* speedups in Intel2 machine. Tests done on AMD (top), Intel1(middle) and Intel2(bottom)

A hardware counter analysis shows indeed a smaller number of L3-cache-misses (around 5 times less) for our configuration compared to Pluto's.

Another case where fusion is important is showcased in *symm*: Our fusion heuristic decides to distribute one statement from the beginning, enabling parallelism. The result produced by Pluto is, instead, fully sequential (a complete fusion is applied).

Another factor to highlight is the fact that, for a few cases, we need to change the kernel-specific configuration between different architectures. This can be explained by several factors, such as different cache sizes, different numbers of cores and threads and different environments (compiler, architecture, operating system). Among these cases, we can find *jacobi-2d*, *heat-3d* and *fdtd-2d*, where for the Intel machines isl-style is the most performant configuration, while for AMD a simple loop distribution performs better.

In some cases, our pluto-style strategy can outperform the Pluto scheduler, and in some other cases, the reverse situation applies. This is mostly given by different fusion heuristics implemented in the two schedulers. This gives an idea of the impact of the fusion heuristic on the optimization problem and the limits of the existing heuristics.

These experiments show that kernel-specific configurations can be really useful to explore transformations with minimal effort. However, the results also show that the generic strategies
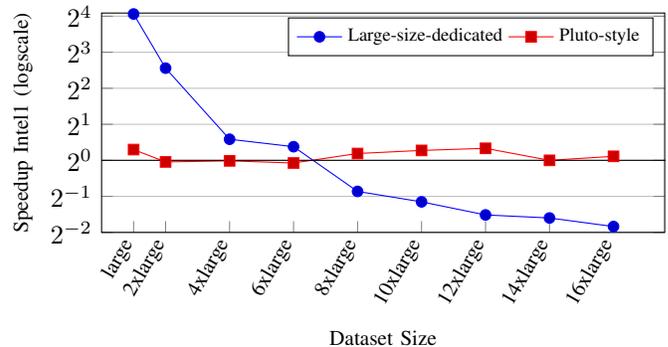

Fig. 3: Speedups of PolyTOPS compared to Pluto for Jacobi-1d using two different configurations and multiple data set sizes. The blue one is (best) dedicated configuration considered for large size. The red one is the configuration Pluto-style.

defined so far have room for improvement because they ignore many scenario factors (architecture, use case characteristics). PolyTOPS can help to design generic configurations that can work better than the state-of-the-art.

2) Dataset Size Analysis: In the context of kernel size and scheduling choices, *Jacobi-1d* (similarly to *trisolv*) is an example that highlights the impact of kernel size on performance. The charts in Fig. 2 show that our solution outperforms Pluto for all machines. Upon closer inspection, our solution generates a simple and fully sequential code, while
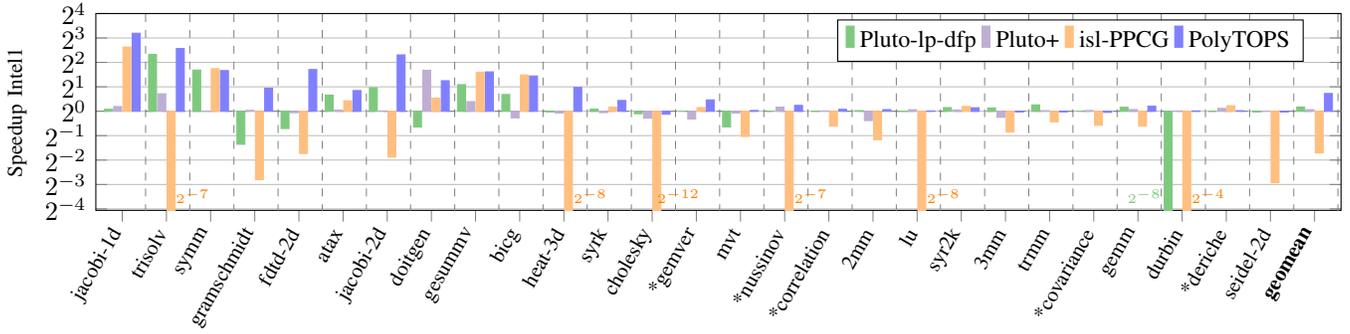
Fig. 4: Speedups (log-scale) of PolyTOPS using the kernel-specific configuration, Pluto-lp-dfp (best fusion heuristic) and Pluto+ compared to Pluto (last dev version). For the cases marked with the symbol *, no solution was found by any of the fusion heuristics of Pluto-lp presented in [29].

Pluto generates a more complex code with several conditions and complex data accesses, enabling parallelism for the inner loop through initial skewing.

The graph in Fig. 3 demonstrates how the speedup of large-size dedicated configuration (in blue) changes with different Polybench dataset sizes. We also present the results of our pluto-style configuration (in red), which is more consistent when the size changes. It is noticeable that the parallelism achieved by Pluto (and our pluto-style version) has a greater impact when the size increases. This indicates that the size of the kernel is an important factor that the scheduler should take into account, which points to a possible direction for future research. Furthermore, we want to highlight the flexibility of PolyTOPS, which can be reconfigured for all sizes, ensuring it is at least as good as Pluto, and demonstrating the power of our reconfigurability.

C. Comparing scheduling tools on Polybench

We compare PolyTOPS with Pluto+ [16], Pluto-lp-dfp [17] and isl-PPCG [9]. The first two works extend Pluto in several ways, while isl-PPCG is a specific version of isl-scheduler used in PPCG project. The differences relevant to the scope of our experiments are that Pluto+ allows negative coefficients and that several fusion heuristics are available in Pluto-lp-dfp. isl-PPCG instead uses a combination of Pluto and Feautrier scheduling algorithms, and it also uses different fusion heuristics. Speedups over Pluto are reported in Fig. 4. For Pluto-lp-dfp, only the highest speedup obtained from the three fusion heuristics for each code is shown [29] (except for some cases where some of the fusion heuristics did not produce a result). The speedup shown for PolyTOPS corresponds to kernel-specific configurations. As for PolyBench, CLooG is the code generator used for all schedulers.

In the case of *doitgen*, Pluto+ outperforms PolyTOPS by enabling parametric shifting. This is a transformation by default in Pluto+, with no option to disable it. The same solution can be obtained with PolyTOPS by enabling parametric shifting. However, we consider it unfair to compare with Pluto, which does not allow it.

Negative coefficient support is required to find transformations for *nussinov* and *deriche*. It is currently only supported in Pluto+. Pluto, Pluto-lp-dfp and PolyTOPS only post-process

the initial schedule, whereas for *deriche* Pluto+ can compute a slightly better schedule.

Pluto-lp-dfp achieves a slight speedup over PolyTOPS for *trmm* and *3mm* due to different intra-tile optimization (post-processing).

In all other cases, PolyTOPS performs better (or similarly) than all the other versions for two reasons. Firstly, allowing negative coefficients in Pluto+ is not beneficial for most of the Polybench cases. Secondly, fusion heuristics Pluto-lp-dfp focus on generic high-level fusion heuristics that cannot compete (except for some cases like *trisolv* and *symm*) with the kernel-specific fusion decisions from our configurations.

Regarding isl-PPCG results, cases like *trisolv, gramschmidt, jacobi-2d, heat-3d, cholesky, nussinov, lu,* and *seidel-2d* show big slowdowns, mainly because of fusion choices, complex skewing (caused by Feautrier cost function) that generate a complex final code. A part from that, we can see that in some other cases like *jacobi-1d, symm, gesummv, and bicg* isl is capable of finding transformations that are really close (or equal) to the ones found with the best configuration of PolyTOPS.

D. Comparing scheduling tools on PolyMage

We finally compare PolyTOPS performances on the Poly-MAGE [30] benchmark suite on *Intel1*. It contains 7 use cases coming from image processing. Loop-based computations and stencils characterize these codes, making them interesting scenarios for polyhedral optimizations. For our experiments, we started from the naive version of the codes provided in the benchmark and adapted them to our pipeline with several pre-processing steps. Clan [31] was used to transform the C++ codes into OpenScop format, and it has been adapted to support the division operation in the array indices.

From the results in Table II, notice that many results are not available: For *camera-pipe, interpolate and pyramid-blending* Pluto (in all the different versions) does not support local variables in the polyhedral representation. These are necessary to represent if statements using modulo and division operations, and they are also necessary for some complex accesses. isl-PPCG can handle all the cases except *pyramid-blending*, where the transformation generated is empty due to some internal error.

TABLE II: PolyMage benchmark: Timing (milliseconds) and relative speedups among PolyTOPS and the state-of-the-art schedulers (isl-PPCG, Pluto, Pluto-lp-dfp, Pluto+). For some cases and schedulers, the results are unavailable (n.a.) because of technical limitations.

Benchmark	PolyTOPS (ms)	isl-PPCG (ms)	Pluto (ms)	Pluto-lp-dfp (ms)	Pluto+ (ms)	Speedup (isl-PPCG)	Speedup (Pluto-dev)	Speedup (Pluto-lp-dfp)	Speedup (Pluto+)
harris	47	108	57	47	57	2.28	1.19	1	1.19
unsharp-mask	120	120	134	120	132	1	1.10	1	1.10
camera-pipe	88	177	n.a	n.a.	n.a	2.01	n.a	n.a	n.a
interpolate	89	71	n.a	n.a	n.a	0.79	n.a	n.a	n.a
pyramid-blending	74	n.a	n.a	n.a	n.a	n.a	n.a	n.a	n.a

The available results show that PolyTOPS outperforms or is on par with state-of-the-art schedulers. The codes *camera-pipe*, *interpolate*, and *pyramid-blending* contain many statements while having a low loop dimensionality. Thus the major difficulty when optimizing them is selecting a good fusion heuristic that may enable better parallelism while remaining cache-friendly. For *interpolate*, the performance obtained is lower than isl-PPCG because PPCG uses a more precise code generation. In our case, Cloog [24] often does not take into account all directives specifying parallel dimensions for code generation, losing several parallelization opportunities. For *pyramid-blending*, no code is generated by isl.

V. CONCLUSION

PolyTOPS is a novel polyhedral scheduler tool that improves upon the state-of-the-art black-box polyhedral schedulers by offering an easy way to configure and tune polyhedral scheduling. It can adapt to various application scenarios where polyhedral optimization was previously dismissed due to the poor results of black-box schedulers. Inputs and outputs can be expressed either as isl objects or in OpenScop format. Thus, the output of PolyTOPS can be fed into code generation tools such as isl [8] or CLooG [24]. PolyTOPS has been integrated into MindSpore AKG compiler [2]. The performance of PolyTOPS has been evaluated on one application scenario and on two benchmark suites. On the application scenario of *hybrid custom operators* [25] for an Ascend NPU, a feature of MindSpore [3], the configurability and flexibility of PolyTOPS led to better performance than with the isl scheduler (up to x34 speedup). On Polybench [28] benchmark suite, we showed that simple configurations can mimic the behaviours of state-of-the-art scheduling strategies (isl, Tensor Scheduler, Pluto) and that completely new general configurations can be created with little effort, outperforming Pluto scheduler [7] (x1.8 geomean speedup) using kernel-specific configurations in different CPUs. On the Polymage benchmark suite, we have shown that PolyTOPS outperforms or is on par with other schedulers.

This work paves the way for further research. The design of fusion heuristics is crucial for high performance and could be an extension for PolyTOPS configurations. Extending the currently proposed rules for fusion and defining pattern-guided fusion heuristics would be a way to enrich the existing scheduling heuristics. Finally, more software and hardware-specific configuration extensions could prove useful: Internal heuristics for fusion, tiling adapted to the input hardware

configuration and scheduling decisions based on the kernel size.

APPENDIX A
ARTIFACT APPENDIX

A. Abstract

This artifact provides a docker image that contains programs and scripts to generate results for Fig. 2, Fig. 3, Fig. 4 and Table II. Results may differ depending on the target architecture or system.

The image contains PolyTOPS, Pluto, Pluto+, Pluto-lp, and PPCG. Additional software such as clan, Candl, Cloog, isl and FPL are also available.

In this artifact, you will be able to replicate the results shown in the paper and test PolyTOPS and its functionalities.

B. Artifact check-list (meta-information)

- **Goal**: Reproduce results for Fig. 2, Fig. 3, Fig. 4 and Table II
- **Compilation**: private
- **Hardware**: see Section IV-B.
- **Metrics**: Time (Average time over the number of repetitions) in ms for PolyMage and cycles for PolyBench tests.
- **Output**: CSV(comma separated values) files.
- **How much disk space is required (approximately)?**: 6GB.
- **How much time is needed to prepare workflow (approximately)?**: 1 min.
- **How much time is needed to complete experiments (approximately)?**: more than 12 hours for *Intel1* (full experiments are required, but the timing can be tremendously reduced if excluding *isl-PPCG* results as described in Section A-E3), while around 5 hours for *AMD* and *Intel2* (only Section A-E1 needs to be tested for these 2 machines.)
- **Archived?**: the artifact can be found in Zenodo https://doi.org/10.5281/zenodo.10203989.

C. Description

1) Delivery: a docker image can be found on Zenodo [32] (https://doi.org/10.5281/zenodo.10203989)

2) Hardware dependencies: see Section IV-B, respectively *Intel1, Intel2, AMD.*

3) Software dependencies: Docker v24

4) Data Sets: Polybench, PolyMage

D. Installation

The `docker` is published on Zenodo [32] and can be loaded from file `polytops.tar` as follows:

```
$ docker load -i polytops.tar
```

Upon success, image `polytops:cgo-2024` will be available.

E. Experiment workflow

A new `polytops:cgo-2024` container can be run using the following command:

```
$ docker run -it --cap-add=SYS_NICE polytops:cgo-2024
```

The image is set up so the default command is '`/bin/bash --login`'.
Note that the internal configuration (see '`/etc/profile.d`') requires a login shell for additional software to be found and executed.

Once inside the container, you can run the following commands:

```
$ cd $HOME/test
$ bash ./run_complete_artifact.sh
```

To replicate our results, we **strongly suggest** to the users to wrap test executions in the following command:

```
$ sudo --login nice -n -20 bash -c "{ cd $(pwd); <test>; }"
```

Password is `polytops`. This command allows us to prioritize the execution of our experiments. For instance, the previous command would become:

```
$ sudo --login nice -n -20 bash -c "{ cd $(pwd);
    bash ./run_complete_artifact.sh;}"
```

For readability, we will not rewrite it for all the following commands.

The script "run_complete_artifact.sh" will run all the scripts and generate the output timings, representing the results shown in:

- *PolyTOPS-results* (Fig. 2):
  $HOME/test/test_fig2_and_4/fig_2.csv
- *Data-Size* (Fig. 3):
  $HOME/test/test_fig3/fig_3.csv
- SOTA (Fig. 4):
  $HOME/test/test_fig2_and_4/fig_4.csv
- PolyMage (Table II):
  $HOME/test/test_polymage/times_polymage.csv

Notice that the complete script is configured for *Intel1*, while for *Intel2* and *AMD* you can refer to Section A-E1 that explains how to run only the specific tests in Fig. 2.

The results can also be computed singularly for each test case.

1) PolyTOPS-results: To obtain the results described in Fig. 2, users can run these commands:

```
$ cd $HOME/test/test_fig2_and_4/
$ bash test_fig2.sh -c $HOME/test/paper_best_configs_INTEL1
    -n 10
```

where the option *-c* specifies the root path of the PolyTOPS configuration files that we used for PolyBench cases, and *-n* specifies the number of executions we want to run for each final transformation. For the *-c* option, you can select any of the following paths, depending on which experiment of Fig. 2 you want to reproduce:

- For Fig. 2 *Intel1* machine you can use:
  $HOME/test/paper_best_configs_INTEL1/
- For Fig. 2 *Intel2* machine you can use:
  $HOME/test/paper_best_configs_INTEL2/
- For Fig. 2 *AMD* machine you can use:
  $HOME/test/paper_best_configs_AMD/

2) Data-Size: To replicate these experiments, it is just necessary to run the following commands:

```
$ cd $HOME/test/test_fig3/
$ bash test_fig3.sh -n 10
```

where *-n* is an option specifying the number of executions for each program version. The output is generated automatically in `$HOME/test/test_fig3/fig_3.csv`.

3) SOTA: To replicate Fig. 4, you can run the following commands:

```
$ cd $HOME/test/test_fig2_and_4/
$ bash test_fig4.sh -n 10
```

where *-n* specifies the number of executions for each program version. **Notice** that in this part of the experiments, a big portion of time is taken by isl-PPCG results (see the tremendous slowdowns in Fig. 4). If the user wants to exclude isl from the experiment, the `$HOME/test/test_fig2_and_4/test_fig4.sh` script can be edited at line 32, removing the *isl* keyword.

4) PolyMage: To replicate the PolyMage experiments in Table II, you can run the following command:

```
$ cd $HOME/test/test_polymage
$ bash test_polymage.sh
```

The *output* will be available in the file `$HOME/test/test_polymage/times_polymage.csv`.

F. Evaluation and expected result

The results produced (*PolyTOPS-results*, *Data-Size*, *SOTA*, and *PolyMage*) are CSV files containing the average execution time of the different versions of the test cases and the standard deviation of the timing.

The results in the paper are equivalent for *PolyMage* (Table II), while for the other charts, we calculated a speedup (compared to Pluto results) in log scale (base 2) using the following formula:

$$speedup = pluto\_time/variant\_time$$

where *variant_time* represents any of the variants of PolyTOPS or any other scheduler in the charts in Fig. 2, Fig. 4, and Fig. 3.

G. Experiment customization

If you want to use our tool for custom cases, you can use the polytops command directly. PolyTOPS supports OpenScop as input (produced by Clan) and the final code generation is done by Cloog. Given an input C file *input.c* (that must contain the proper PRAGMA), a simple pipeline to generate an optimized version *out.c* is:

```
$ clan input.c | polytops --input-format=openscop
  --tiling=true --compute-dependencies=true
  --output-format=openscop | cloog stdin -openscop -o
  ./out.c
```

The option *–help* provides a list of all the available options.

Moreover, we also provide another script in

`$HOME/test/scripts/single_case.sh`

that can be used to run a similar pipeline but with some extra PolyTOPS options. This script contains several extra functionalities that can be displayed with the *–help* option.

REFERENCES

[1] Nicolas Vasilache et al. *Tensor Comprehensions: Framework-Agnostic High-Performance Machine Learning Abstractions*. 2018. arXiv: 1802.04730.

[2] Jie Zhao et al. "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*. PLDI 2021. Virtual, Canada: Association for Computing Machinery, 2021, pp. 1233–1248. DOI: 10.1145/3453483.3454106.

[3] Lei Chen. *Deep Learning and Practice with MindSpore*. Springer, 2021. ISBN: 978-981-16-2232-8. DOI: 10.1007/978-981-16-2233-5.

[4] Scott Cyphers et al. "Intel nGraph: An Intermediate Representation, Compiler, and Executor for Deep Learning". In: *CoRR* abs/1801.08058 (2018). arXiv: 1801.08058.

[5] Chris Lattner et al. "MLIR: A Compiler Infrastructure for the End of Moore's Law". In: *CoRR* abs/2002.11054 (2020). arXiv: 2002.11054.

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

[7] Uday Bondhugula et al. "A Practical Automatic Polyhedral Parallelizer and Locality Optimizer". In: *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI'08 (Tucson, AZ, USA, June 7–13, 2008), pp. 101–113. DOI: 10.1145/1375581.1375595.

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

[9] Sven Verdoolaege et al. "Polyhedral Parallel Code Generation for CUDA". In: *ACM Trans. Archit. Code Optim.* 9.4 (Jan. 2013). ISSN: 1544-3566. DOI: 10.1145/2400682.2400713.

[10] Benoît Meister, Eric Papenhausen Akai Kaeru, and Benoît Pradelle Silexica. "Polyhedral Tensor Schedulers". In: *2019 International Conference on High Performance Computing Simulation (HPCS)*. 2019, pp. 504–512. DOI: 10.1109/HPCS48598.2019.9188233.

[11] Heng Liao et al. "Ascend: a Scalable and Unified Architecture for Ubiquitous Deep Neural Network Computing : Industry Track Paper". In: *2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. 2021, pp. 789–801. DOI: 10.1109/HPCA51647.2021.00071.

[12] Cedric Bastoul. "Keynote: Automatic operator generation for deep learning frameworks in the all-scenario context: MindSpore/AKG architecture, features and challenges." In: 12th International Workshop on Polyhedral Compilation Techniques (IMPACT 22), 2022. URL: https://impact-workshop.org/impact2022/slides/keynote.pdf.

[13] Cedric Bastoul et al. "Optimizing GPU Deep Learning Operators with Polyhedral Scheduling Constraint Injection". In: *2022 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. 2022, pp. 313–324. DOI: 10.1109/CGO53902.2022.9741260.

[14] Louis-Noël Pouchet et al. "Loop Transformations: Convexity, Pruning and Optimization". In: *ACM SIGPLAN Notices* 46 (May 2011), pp. 549–562. DOI: 10.1145/1925844.1926449.

[15] A. Schrijver. *Theory of Linear and Integer programming*. Wiley-Interscience, 1986. ISBN: 978-0-471-98232-6.

[16] Uday Bondhugula, Aravind Acharya, and Albert Cohen. "The pluto+ algorithm: A practical approach for parallelization and locality optimization of affine loop nests". In: *ACM Transactions on Programming Languages and Systems (TOPLAS)* 38.3 (2016), pp. 1–32. DOI: 10.1145/2896389.

[17] Aravind Acharya, Uday Bondhugula, and Albert Cohen. "Polyhedral Auto-Transformation with No Integer Linear Programming". In: *SIGPLAN Not.* 53.4 (June 2018), pp. 529–542. ISSN: 0362-1340. DOI: 10.1145/3296979.3192401. URL: https://doi.org/10.1145/3296979.3192401.

[18] Martin Kong and Louis-Noël Pouchet. "Model-Driven Transformations for Multi- and Many-Core CPUs". In: *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI 2019. Phoenix, AZ, USA: Association for Computing Machinery, 2019, pp. 469–484. ISBN: 9781450367127. URL: https://doi.org/10.1145/3314221.3314653.

[19] Lorenzo Chelini et al. "Automatic Generation of Multi-Objective Polyhedral Compiler Transformations". In: *Proceedings of the ACM International Conference on Parallel Architectures and Compilation Techniques*. PACT '20. Virtual Event, GA, USA: Association for Computing Machinery, 2020, pp. 83–96. DOI: 10.1145/3410463.3414635.

[20] Tom Hammer and Vincent Loechner. "PolyLingual: a Programmable Polyhedral Scheduler". In: (2023). URL: https://impact-workshop.org/impact2023/papers/paper6.pdf.

[21] Riyadh Baghdadi et al. "A Deep Learning Based Cost Model for Automatic Code Optimization". In: *CoRR* abs/2104.04955 (2021). arXiv: 2104.04955.

[22] Oleksandr Zinenko, Stéphane Huot, and Cédric Bastoul. "Clint: A direct manipulation tool for parallelizing compute-intensive program parts". In: *2014 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*. 2014, pp. 109–112. DOI: 10.1109/VLHCC.2014.6883031.

[23] Lénaïc Bagnères et al. "Opening Polyhedral Compiler's Black Box". In: *Proceedings of the 2016 International Symposium on Code Generation and Optimization. CGO '16*. Barcelona, Spain: Association for Computing Machinery, 2016, pp. 128–138. DOI: 10.1145/2854038.2854048.

[24] Cedric Bastoul. "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. USA: IEEE Computer Society, 2004, pp. 7–16. DOI: 10.1109/PACT.2004.1342537.

[25] *4.1 Unified Cross-Platform MindSpore Hybrid DSL Expression*. 2022. URL: https://mindspore.cn/news/newschildren/en?id=1985.

[26] Xiaoyao Liang. "Chapter 3 - Hardware architecture". In: *Ascend AI Processor Architecture and Programming*. Ed. by Xiaoyao Liang. Elsevier, 2020, pp. 75–100. ISBN: 9780128234891.

[27] Heng Liao et al. "Ascend: a Scalable and Unified Architecture for Ubiquitous Deep Neural Network Computing : Industry Track Paper". In: *2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. 2021, pp. 789–801. DOI: 10.1109/HPCA51647.2021.00071.

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

[29] Aravind Acharya, Uday Bondhugula, and Albert Cohen. "Effective Loop Fusion in Polyhedral Compilation Using Fusion Conflict Graphs". In: *ACM Trans. Archit. Code Optim.* 17.4 (Sept. 2020). DOI: 10.1145/3416510.

[30] Ravi Teja Mullapudi, Vinay Vasista, and Uday Bondhugula. "PolyMage: Automatic Optimization for Image Processing Pipelines". In: *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems*. ASPLOS '15. Istanbul, Turkey: Association for Computing Machinery, 2015, pp. 429–443. ISBN: 9781450328357. DOI: 10.1145/2694344.2694364. URL: https://doi.org/10.1145/2694344.2694364.

[31] Cédric Bastoul et al. "Putting Polyhedral Loop Transformations to Work". In: *LCPC'16 International Workshop on Languages and Compilers for Parallel Computers, LNCS 2958*. College Station, Texas, Oct. 2003, pp. 209–225.

[32] Gianpietro Consolaro, Harenome Razanajato, and Nelson Lossing. *PolyTOPS artifact*. Nov. 2023. DOI: 10.5281/zenodo.10203989. URL: https://doi.org/10.5281/zenodo.10203989.