Automatic Generation of Multi-Objective Polyhedral Compiler Transformations

Lorenzo Chelini[*]
Eindhoven University of Technology
l.chelini@tue.nl

Tobias Gysi
ETH Zurich
tobias.gysi@inf.ethz.ch

Tobias Grosser
ETH Zurich
tobias.grosser@inf.ethz.ch

Martin Kong
University of Oklahoma
mkong@ou.edu

Henk Corporaal
Eindhoven University of Technology
h.corporaal@tue.nl

ABSTRACT

To this day, polyhedral optimizing compilers use either extremely rigid (but accurate) cost models, one-size-fits-all general-purpose heuristics, or auto-tuning strategies to traverse and evaluate large optimization spaces.

In this paper, we introduce an adaptive and automatic scheduler that permits to generate novel loop transformation sequences (or recipes) capable of delivering strong performance for a variety of different architectures without relying on auto-tuning, nor on pre-determined transformation strategies.

We evaluate our approach using the Polybench/C benchmark suite against two modern state-of-the-art optimizers on three different architectures: An AMD ThreadRipper, an Intel Xeon Phi, and an Intel Xeon Platinum. Our results provide evidence that a set of high-level objectives backed up by an automatic adaptive scheduler (i.e., not hard-wired) is capable of achieving competitive performance, while only resorting to evaluating a handful of tuned variants.

CCS CONCEPTS

• **Theory of computation** → **Continuous optimization**; • **Software and its engineering** → **Compilers**; • **Information systems** → Information storage systems.

KEYWORDS

affine transformations; polyhedral model; loop optimization

ACM Reference Format:
Lorenzo Chelini, Tobias Gysi, Tobias Grosser, Martin Kong, and Henk Corporaal. 2020. Automatic Generation of Multi-Objective Polyhedral Compiler Transformations. In *Proceedings of the 2020 International Conference on Parallel Architectures and Compilation Techniques (PACT '20), October 3–7, 2020, Virtual Event, GA, USA.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3410463.3414635

[*]work done at ETH Zurich

1 INTRODUCTION

Modern polyhedral compilers and affine transformation frameworks [4, 7, 14, 17, 19, 26, 29, 31, 43] are, despite the vast progress made in the last 30 years, still hindered by rigid cost models and heuristics. To a large extent, their modern popularity is owed to the ability to compose transformations in a single pass, and to the success of general-purpose optimization criteria. The minimization of the maximal dependence distance [1, 7, 27]–a simple but effective heuristic that improves the program's locality by bringing computations that share data closer in time is one of such criteria [28]. Other efforts have also attempted to introduce additional performance objectives such as contiguity and inner-parallelism to maximize SIMD-vectorization opportunities [24, 40, 44]. As it turns out, architectural and hardware trends (i.e., an increasing number of cores, larger and deeper memory hierarchies, longer latencies, and widening vector units) together with ever more complex and demanding software applications [41] are rendering sub-optimal general-purpose (polyhedral) compiler techniques. This makes it imperative to design and develop novel and adaptive compiler optimizations that depart from single, general-purpose optimization criteria.

The *legal space formulation* [33, 39] and the notion of *performance lexicons* recently introduced in [23] give some unique opportunities to conduct guided search spaces operating with a higher notion of performance optimization. The latter uses a set of Integer Linear Program (ILP) cost functions that can be combined and used to create novel and customized transformation sequences. Examples of these ILP objectives are *Dependence Guided Fusion* (DGF) and *Stride Optimization* (SO). Indeed, steering the exploration of transformation variants with a performance lexicon is equivalent to pruning the candidate schedules that do not exhibit one/any of the desired traits. If we were to explicitly generate and test all combinations in a set with L objectives, or more precisely, permutations of ILP objectives, we would be required to traverse a space of size $\mathcal{O}(L!)$. In addition, the complexity drastically increases when the selection of transformations can be further customized to subsets of statements.

In this work, we develop mechanisms by which this extraordinarily large space can be characterized, queried, and navigated in an effective and scalable way. Two of our main contributions are: i) A method to estimate the impact of subsets and combinations of high-level performance objectives, where each objective is implemented as an ILP cost function, without the need for explicit compilation or exhaustive enumeration. ii) A novel technique to extrapolate the performance effects of larger combinations of ILP objectives in various scenarios (i.e., code patterns, statement partitions, or new architectures). These mechanisms enable us to navigate the legal space of transformations in a controlled fashion, testing only small combinations of objectives and quantifying their impact without actual runs, and form the basis for an adaptive scheduler—a form of greedy domain-specific compilation scheme, which is not bound by chains of prefixed transformation recipes.

Our work bridges the gap between general-purpose and domain-specific compilation while simultaneously reducing the need for auto-tuning. Moreover, the scheduling system described here is adaptive in nature, in the sense that the selection and sequence of transformations applied will vary depending upon patterns observed and the target architecture. Nonetheless, unlike [23], such patterns are not hardwired. In summary, our contributions are:

- A novel approach to generate new transformation recipes with an adaptive polyhedral scheduling engine; we rely on a pre-computed database to perform queries on the fly and decide the best sequence of ILP objectives to apply to subsets of program statements.
- A novel methodology to characterize, traverse and quantify large loop transformation spaces.
- Evidence that an enhanced and semantically rich ILP performance lexicon, backed up by an adaptive automatic scheduler is capable of delivering competitive, and in some cases strong, performance without resorting to large scale auto-tuning techniques.
- An extensive evaluation on three modern and architecturally diverse processors, achieving comparable or superior performance to current state-of-the-art optimizers.

The remaining of this paper is organized as follows: Sec.2 further motivates our research problem; Sec.3 presents the related background concepts, including recapping the legal space formulation and extensions to the performance lexicon; Sec.4 describes in detail the steps necessary to implement an automatic and adaptive polyhedral scheduling engine. Next, the effectiveness of our scheduling system is demonstrated in the experimental evaluation of Sec.5. We conclude by discussing related work in Sec.6 and the conclusion in Sec.7.

2 MOTIVATION

In intensive loop computations, compiler transformations must balance the complex interactions between parallelism and locality to achieve the best possible performance. On one end, maximizing parallelism often results in loop distribution (fission), loop permutations pushing dependence satisfaction into the innermost vectorizable loop level, and loss of program locality. On the other end, indiscriminate maximization of locality via loop fusion can reduce parallelism. Various polyhedral compilers try to address such complex interplay of objectives.

Acharya et al. [1, 2] enhanced Pluto's tiling-hyperplane algorithm, enabling the exploration of a richer transformation space. Their technique has proven extremely scalable but mostly applied to multi-core processors. Besides, they consider the selection of tile sizes as an orthogonal problem; Ultimately, affecting the program's parallelism and locality. Zinenko et al. [49] proposed a template for the construction of affine scheduling algorithms that accounts for multiple levels of parallelism and deep memory hierarchies. Their approach models multi-level parallelism and temporal/spatial locality by orchestrating a collection of parameterizable optimization problems with configurable constraints and objectives. Still, their approach hard-wires several design decisions, e.g., when to switch from carrying fewer spatial proximity relations to carrying as many as possible. For example, in the presence of both coincident (parallelism) and proximity (locality) relations being carried by the same schedule dimension, parallelism is always preferred. This design decision is equivalent to hard-wiring the order between two ILP objectives. Recently Kong and Pouchet [22, 23] introduced another successful solution, which consisted of an extensible transformation lexicon of "small" ILP objectives, each addressing a specific performance property. Their approach identified pre-determined patterns (i.e., shallow loop nests) found in the code and decided a priori the ILP objectives to apply depending on the pattern detected and the target machine. The lexicon was proven effective across several architectures. But it still comes with limitations: (i) The rigidness of how the subset of performance properties are selected and embedded into the ILP, both per pattern, and architecture, (ii) how the priority of the objectives is established. The second limitation heavily undermines the approach scalability, as the number of choices of the objective's priority order is $L!$ (L being the size of the lexicon). Thus, explicit enumeration and testing of all possible variants is only possible for a handful of cases, becoming quickly intractable with ILP lexicons of 8 or more objectives. If, in addition, we consider all possible statement clustering options (number of subsets in a set with S statements), the optimization space grows exponentially.

In a separate category of techniques, we have auto-tuning systems and directive-based approaches. The former conducts explicit searches of large candidate transformation spaces guided by particular criteria (i.e., minimizing time, optimizing for energy or memory footprint, or even searching for particular combinations of the prior) [3, 9, 12, 19, 38, 48]. The latter, offer control over the transformations applied by allowing the programmer to give precise constraints. But they increase the user's burden and make the code less portable.

In summary, the above strategies center more on the mechanisms to generate large exploration spaces rather than focusing on: (i) How to generate small but meaningful tuning spaces; (ii) cater end-users by providing high-level, composable and customizable policies that permit to navigate the space; and (iii) supply users with mechanisms to assess the characteristics and demographics of the constructed tuning spaces.

To further illustrate the complexity of this problem, we show in Figure 1 the performance distribution of more than 2000 tile variants generated with the Pluto compiler, for the `2mm` compute kernel (Listing 1). The generated variants are evaluated on two representative high-end systems—a 68-core Intel Xeon Phi and a 48-core Intel Xeon Platinum (formerly Skylake). Variants are grouped into bins by performance and loop fusion heuristic used: Maximal loop distribution (*nofuse*), maximal loop fusion (*maxfuse*) and, smart-fuse—a hybrid heuristic that attempts to balance locality and parallelism (*smartfuse*). Tile sizes are selected to produce tile volumes fitting between the L1- and L2-caches.

Tile sizes, in addition to determining the program's locality, also control the number of parallel loop iterations to be distributed

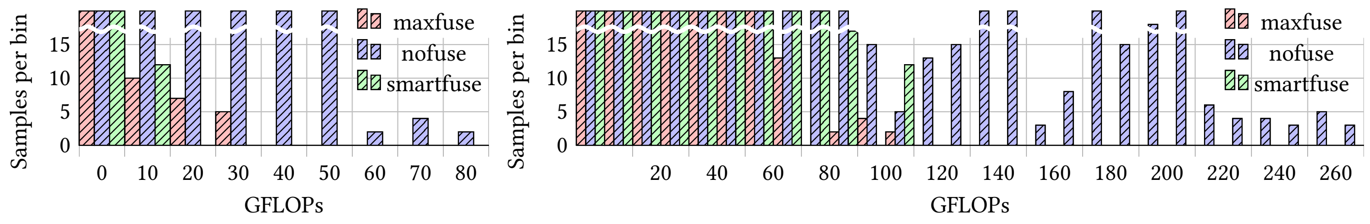

Figure 1: Pluto's performance distributions for Polybench/C 2mm kernel (Double Precision, Standard Dataset) on two Intel machines: a 68-core Intel Xeon Phi (left) and a 48-core Intel Xeon Platinum (right).

```
    for (int i = 0; i < N; ++i)
      for (int j = 0; j < N; ++j) {
S1:     tmp[i][j] = 0;
        for (int k = 0; k < N; ++k)
S2:       tmp[i][j] += alpha * A[i][k] * B[k][j];
    }
    for (int i = 0; i < N; ++i)
      for (int j = 0; j < N; ++j) {
S3:     D[i][j] *= beta;
        for (int k = 0; k < N; ++k)
S4:       D[i][j] += tmp[i][k] * C[k][j];
    }
```

Listing 1: 2mm kernel source code.

among the processing cores. In other words, the number of parallel iterations is inversely proportional to the tile size selected. The best performing variant is the *nofuse* heuristic, with 1x256x32 and 20x256x32 tile sizes, for the Xeon Platinum and the Xeon Phi, respectively. From the figure, we can observe that only a tiny fraction of the space (2/2000 variants) achieve approximately 80 GF/s on the Xeon Phi. This number increases to 3 points (over the same 2000 variants) on a 48-core Intel Xeon Platinum system, peaking at 260 GF/s. As a reference, Intel MKL's DGEMM achieves 1TF/s on the Xeon Phi and 0.75TF/s on the Xeon Platinum. In conclusion, extracting strong performance is a complicated task. Our work, however, achieves nearly identical performance by only generating and explicitly evaluating 5 different transformation variants, thus significantly reducing tuning time. In the following sections, we introduce and describe methods by which novel and customized sequences of affine transformations can be automatically generated with an adaptive scheduler, i.e., which does not resort to hard-wired transformation recipes (Sec.4 for a summary of the framework and approach).

3 BACKGROUND

The polyhedral model operates on Static Control Parts (SCoPs). Each SCoP consists of loops whose boundaries and access relations are functions of outer loop iterators and program (symbolic) parameters. Standard restrictions of the polyhedral model, such as having a single entry and exit point in each loop, use of non-affine expressions, presence of indirect accesses, and side-effect free functions can be relaxed at the cost of more conservative program analyses [6, 34, 36]. Next, we recap the basic internal abstractions used by polyhedral compilers: iteration domains, access functions, dependence polyhedra, and schedule.

Iteration domains capture the dynamic instances of each program statement S. An iteration domain $\mathcal{D}^S \in \mathbb{Z}^+$ is a polyhedron defined by a system $M\vec{x} \geq \vec{0}$ constructed from the upper and lower bounds of each loop iterator. A two-dimensional M-by-N loop nest with a single statement would be defined as $\{(i, j) : 0 \leq i \wedge 0 \leq j \wedge i \leq M \wedge j \leq N\}$. The relation between the iteration space $\mathcal{D}^S$ and the data-space of a multi-dimensional array A is represented with **access functions/relations**, which associate to each point $\vec{x} \in \mathcal{D}^S$ a data-element $F_S^A(\vec{x})$. For example, the array reference `C[i][j-1]` in a statement S can be modeled with the relation $F_S^A = \{S(i, j) \to C(a1, a2) : a1 = i \wedge a2 = j - 1\}$. Program transformations are modeled as **schedule** functions that map points in iteration domains to a multi-dimensional time-space. Each statement instance is assigned a timestamp representing its execution date. Schedules can be lexicographically compared. For instance, a statement instance with timestamp $\langle 0, 1, 0\rangle$ executes before any other statement starting with $\langle 1, *, *\rangle$ ("*" denotes any value). We use the *2d+1 schedule* representation which allows modeling the nesting structure explicitly of the program as well as the linear transformations occurring (i.e., loop permutation or skewing). The representation uses 2d+1 time dimensions and d+1 columns (or input dimensions), where d is the maximum loop depth. We assume the schedule has zero-offset, and refer to the even rows as *scalar dimensions* and the odd ones as *linear dimensions*. Finally, each program dependence is represented by one or more polyhedra. A **dependence polyhedron** $\mathcal{D}_{R,S}$ is constructed from the constraints of the iterations domains $\mathcal{D}^R$ and $\mathcal{D}^S$, the access functions F_R^A and F_S^A on the common array (as equalities), and the relative execution order of the statements involved. The latter must be preserved by the new schedule $(\Theta^R(x_R) \prec \Theta^S(x_S))$.

3.1 Building the Legal Space

Our work leverages the *convex set of semantic preserving affine schedules* [33, 39] which is a single convex space modeling the valid reordering transformations. The ILP system built integrates legal constraints for all dependence polyhedra of the original SCoP. Given a point $\langle x_R, y_S\rangle \in \mathcal{D}_{R,S}$, the semantics of the original program predicate that the statement instance x_R executes before y_S. Thus, the transformations Θ^R and Θ^S must preserve the ordering condition $\Theta^R(x_R) \prec \Theta^S(y_S)$, which means that the timestamps assigned to them must maintain their relative order. As we use multi-dimensional schedules of depth 2d+1, the difference between two schedules will be a lexicographic positive vector:

$\Theta^S(y_S) - \Theta^R(x_R) \succ (\delta_1, \delta_2, .., \delta_{2d+1})$; 2d+1 boolean (0/1) variables are used for each (convex) dependence polyhedron $\mathcal{D}_{R,S}$, one for each schedule dimension, and the dependence is satisfied when any of the δ_i variables become positive. The ability to decide at which level is a dependence met, is the biggest advantage of this class of scheduling frameworks. These constraints were formalized in [33, 39] as:

$$\forall \mathcal{D}_{R,S}, \forall p \in \{0..2d\}, \delta_p^{\mathcal{D}_{R,S}} : \sum_{p=0}^{2d} \delta_p^{\mathcal{D}_{R,S}} = 1$$
$$\forall \mathcal{D}_{R,S}, \forall p \in \{0..2d\}, \forall \langle \vec{x}_R, \vec{y}_S \rangle \in \mathcal{D}_{R,S} : \qquad (1)$$
$$\Theta_p^S(\vec{x}_S) - \Theta_p^R(\vec{y}_R) \geq -\sum_{c=0}^{p-1} \delta_c^{\mathcal{D}_{R,S}} \cdot (K \cdot \vec{n} + K) + \delta_p^{\mathcal{D}_{R,S}}$$

Eq. 1 states that all dependences must be satisfied exactly once, at some schedule dimension l. The summation expression on the right-hand-side of the inequality constitutes a nullification of the legality constraint for all schedule dimensions $p > l$; This is achieved by the term $K \cdot \vec{n} + K$ where K is a large enough positive integer, and $\vec{n}$ is the vector of parameter coefficients [33].

3.2 The ILP Performance Lexicon

Kong and Pouchet [23] complemented the *legal space* (Sec.3.1) with a **performance lexicon**, i.e., an extendable set of ILP objectives, each of which prunes the *legal space* to drive the lexicographic optimization process. The vocabulary covers multiple performance goals which operate at a much higher level than simple transformations such as “unroll loop-k by 4”. Their approach groups and classifies SCoP sub-components using a predefined set of metrics, such as the number of scheduling dimensions, properties of the dependence graph, or the number of self-dependencies. The predefined classes and the features of the target architecture (e.g., number of cores) are then used to tailor the transformations applied. A form of domain-specific compilation within a general-purpose flow, which delivers strong performance while producing one single optimized transformation variant.

3.3 Glossary

Throughout the remaining of this paper we use the following terminology:

- Legal Space: A single convex space modeling all legal affine transformations (Sec.3.1).
- (ILP) Performance Lexicon/Vocabulary: an extensible set of ILP cost functions, each maximizing a specific property in the output code (Sec.3.2 and Sec.3.4). We also refer to each ILP cost function/objective in the lexicon as a *performance idiom*.
- Nano-kernel (NK): A SCoP consisting of at most two statements, each of dimensionality $d \geq 1$, accessing at most four arrays, possibly inducing one or more program dependences. Dependences should only exist between a single pair of array references to effectively isolate the intended characterization from external noise (i.e. other dependences).
- Variant / Version: An input SCoP transformed by a subset of ILP objectives from the performance lexicon.
- Dependence Key: attributes of a nano-kernel used to find all its associated entries in the Nano-Kernel database. Also referred to as *Input Features*.
- Output Features: set of attributes extracted from the computed transformation and from the generated (output) code.
- Nano-kernel Database of Transformations (NK-database): A database that stores input and output features for each nano-kernel transformed by a set of ILP objectives.
- Statement Partition Policies: Equivalence relations produced by statement policies defined in Sec.4.6.
- Adaptive Scheduler: A greedy domain-specific compilation scheme, which is not bound by chains of prefixed transformation recipes.

Table 1: ILP Transformation Lexicon, abbreviation, and impacting performance property (Par:Parallelism, Loc:Locality).

Objective Name	Abbreviation	Par.	Loc.
Skew Parallelism	SKEWPAR	✓	
Outer Parallelism - Inner Reuse	OPIR	✓	✓
Automatic Dimension Aligning (NEW)	ADA	✓	✓
Inner Parallelism	IP	✓	
Dependence Guided Fusion	DGF		✓
Separation of Independent Statements	SIS		✓
Fusion Preserving Parallelism (NEW)	FPP	✓	✓
Permutability (NEW)	PERM		✓
Stride Optimization	SO		✓
Stencil Vector Skewing	VSKW	✓	✓
Stencil Dependence Classification	SDC	✓	✓
Stencil Parallel Constraints	SPC	✓	✓

3.4 Extensions to the ILP Performance Lexicon

In this work we expand the vocabulary of ILP performance objectives introduced in [22, 23]. The complete list of ILP cost functions we use is reported in Table 1. We add three more cost functions: permutability (PERM), fusion preserving parallelism (FPP), and Automatic Dimensions Aligning (ADA). PERM is directly embedded into the construction of the legal space, as described in [24]:

$$\Theta_l^S(\vec{x}_S) - \Theta_l^R(\vec{y}_R) \geq -\sum_{p=0}^{l-1} (\delta_p^{\mathcal{D}_{R,S}} - \rho_p^{\mathcal{D}_{R,S}}) \cdot (K \cdot \vec{n} + K) + \delta_l^{\mathcal{D}_{R,S}}$$

The ρ boolean variables model permutability of loop dimensions, and are directly linked to the dependence satisfaction variables δ via the inequality $\delta \geq \rho$. This property is maximized with the sum of ρ variables per given dimension (loop level). But, unlike [24], here we restrict this property to a specific subset of statements, while also applying it to the scalar dimensions preceding the linear level. These constraints maximize the number of unrollable loop dimensions by satisfying dependencies as earlier as possible (like Feautrier’s [15] algorithm). This formulation requires relaxing the condition that the sum of δ variables for a given dependence to be equal to 1; instead, we accept this sum to be ≥ 1. The FPP constraints favor transformations that fuse loop dimensions that are both either sequential or parallel. Intuitively, we want to allow fusion when there is no loss of performance. This novel set of constraints applies to each inter-statement dependence polyhedron found between arbitrary statements R and S, and connects the parallelism satisfaction variables of the linear dimensions (the π binary variables) to the ϕ (fusion) variables of the immediate preceding scalar dimension:

$$\phi_l^{R,S} \le 1 - \pi_{l+1}^R + \pi_{l+1}^S \wedge \phi_l^{R,S} \le 1 + \pi_{l+1}^R - \pi_{l+1}^S, \ l \ even$$

$$\max \sum_{\mathcal{D}^{R,S}} W_l \times \phi_l^{\mathcal{D}^{R,S}}$$

We recall that each statement will have one π variable for each linear scheduling dimension (the odd dimensions of Θ^S in the $2d+1$ representation). Then, for every $\delta_c^{\mathcal{D}_{R,S}}$ we define $\pi_c^R \le 1 - \delta_c^{\mathcal{D}_{R,S}}$ and $\pi_c^S \le 1 - \delta_c^{\mathcal{D}_{R,S}}$. The overall property of parallelism is optimized level-wise by maximizing the sum of π_l variables at level l [23].

A $\phi_l^{R,S}$ variable becomes true (1), when both of the involved statements exhibit the same parallelism property, effectively cancelling each other and lifting the upper bound of $\phi_l^{R,S}$. The objective function of FPP uses weights $W_l = 2^{d - \lfloor l/2 \rfloor}$, where d is the maximum loop depth. An example of how this new objective would work is shown in the code snippets below. Without FPP, statements S1 and S2 could potentially be fused by some transformation, inhibiting the available parallelism. In contrast, leveraging the FPP objective will effectively interchange the loop order of either S1 or S2 to align their parallel/sequential property. The final choice of which loop order is produced will ultimately depend on additional objectives embedded in the ILP.

```
parfor (i=0; i<M; i++)            for (i=0; i<M; i++)
  for (j=0; j<N; j++)               parfor (j=0; j<N; j++)
S1: A[i] += 1.0;                  S2: A[j] += B[i]*C[j];
```

The third new ILP objective we introduce is the ADA cost function. Intuitively, the goal of this objective is to align the coefficients used in the linear dimensions of the schedule, to the affine functions of an array reference, from the outermost to the innermost dimension for the former, and from the *slowest* to the *fastest varying dimension* of the array. This objective is a simplified version of the OPIR performance idiom from [23], where the constraints relating the parallelism to the schedule coefficients are not embedded. The result of this objective on the rightmost loop nest of the previous example would essentially permute the loop order, making loop `j` the outermost loop. This happens because `j` is the only iterator in the slowest (and fastest too) varying array dimension for arrays `A` and `C`. The motivation of this cost function is to be able to couple it with FPP, making outer-coarse grained parallelism the driving property to extract, while avoiding loop fusion when it could render performance loss.

4 ADAPTIVE SCHEDULING

Our goal is to automate the selection and ordering of ILP objectives. This translates to generating new schedules that adapt to the computational patterns and hardware characteristics. We first build offline a nano-kernel database (NK-database) of transformations from a sufficiently large set of affine example programs (Sec.4.2). Each entry in the database represents a specific nano-kernel transformed by a small (2 or 3) set of ILP objectives from the performance lexicon. We note that, depending on the size of the lexicon, a single

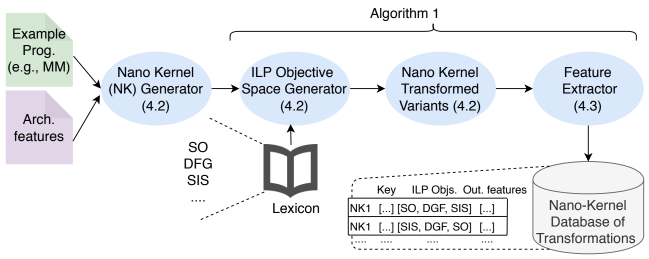
Algorithm 1
Example Prog. (e.g., MM)
Arch. features
Nano Kernel (NK) Generator (4.2)
ILP Objective Space Generator (4.2)
Nano Kernel Transformed Variants (4.2)
Feature Extractor (4.3)
SO
DFG
SIS
....
Lexicon
Key ILP Objs. Out. features
NK1 [...] [SO, DGF, SIS] [...]
NK1 [...] [SIS, DGF, SO] [...]
....
Nano-Kernel Database of Transformations

Figure 2: Offline construction of the Nano-Kernel Database.

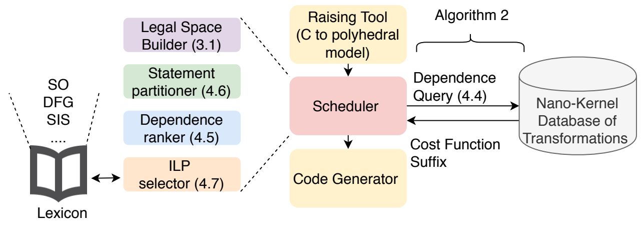

Figure 3: Online compilation phase.

nano-kernel can generate hundreds to thousands of entries in the database. All entries associated with a single nano-kernel are identified by the same *dependence key* (Sec.4.3). The difference among all the entries associated with a nano-kernel are the cost functions applied and the output features extracted post-transformation. Later, these output features are used to compute profitability scores that allow the scheduler to choose the most successful transformation matching a given dependence. Figures 2 and 3 show our framework and the modules used in both phases. The offline step constructs the NK-database. The blue ellipses show the steps to build a constrained search space of ILP performance objectives from the *Performance Lexicon*. Each point in this space represents a small combination of ILP cost functions, all of which are applied to the input nano-kernel, producing a space of transformations for the given input, and each being a variant of the input nano-kernel. We then extract output features from each transformation and store these in the database for future querying. During the online compilation phase, the scheduler (red box in Fig.3) decides the ILP objectives from the lexicon (Tab.1) to integrate into the system by performing queries to the NK-database with a subset of program dependences found in the compiled program (Sec.4.7 and 4.8).

Query results take the form of subsets of ILP objectives to be embedded in a determined order. Profitability scores drive the selection, and each score is computed from output-features stored in entries of the NK-database (Sec.4.4). This stage also sets the order of objectives in the optimization problem. As an illustration, consider the `2mm` kernel from Fig.1 with the dependences $\{D1 : S1 \rightarrow S2\}$, $\{D2 : S2 \rightarrow S2\}$, $\{D3 : S2 \rightarrow S4\}$, $\{D4 : S3 \rightarrow S4\}$ and $\{D5 : S4 \rightarrow S4\}$. The five dependences will be sorted by the *Dependence Ranker* component of Fig.3 (Sec.4.5), preferring the dependences $D2$ and $D5$ in the querying process. Then, the *ILP selector* will run a query on the NK-database of transformations to select up to three ILP cost functions to integrate into the ILP. The querying process is repeated until all dependences are exhausted or until a maximum number of objectives have been embedded. Note that preferring certain subset of dependences does not imply bias on ILP objectives.

During the online phase, we define five *statement partitioning policies* that allow limiting the impact of transformations on the statements belonging to each partition, a form of domain-specific compilation. The motivation is to group statements sharing similar behaviour so that they are transformed by the same set of ILP cost functions.

For example, the statements in the `2mm` kernel (Figure. 1) can be grouped into {S1,S2} and {S3,S4}, if we use as criteria that statements should write to the same array. Another criterion proposed groups statements by their dimensionality, which would produce partitions {S1,S3} and {S2,S4}. The principle followed is that statements in a partition behave the same, and so, should be subject to the same set of ILP objectives.

The partitioning policy affects the scheduler in that only dependences belonging in full to the partition will be considered for the query on the NK-database. This does not affect the legality of the transformation in any way since we are already operating in the convex legal space of semantic preserving transformations.

4.1 Leveraging the ILP Performance Lexicon

Our scheme exploits the combinatorial and modeling power of polyhedral compilers to generate a tractable and yet rich exploration space. The purpose of this space is not to find the best possible transformation for a given SCoP, but to produce a wide variety of candidate transformations such that the most relevant characteristics can be easily identified and mapped to concrete performance objectives. High-level objectives effectively serve the role of performance compasses that can guide the search for different and yet representative program variants. Nonetheless, we still have to deal with the potential combinatorial explosion arising from the number of statement subsets and the issue of choosing objectives for each statement partition. Evaluating all $N!$ variants (where N is the number of ILP cost functions/objectives to be considered), for all statement partitions, is not feasible. But, limiting the exploration to only a small number of permutations (up to three objectives) is possible. The systematic space exploration presented in the next section results in the NK-database which is then used to extrapolate the effect of larger combinations of ILP cost functions on complete kernels.

4.2 Building a Database of Nano-Kernels

We introduce the concept of **Nano-Kernel (NK)**, which has the following properties: 1) Consists of one or two program statements, each with a loop depth greater or equal to one. 2) Each statement accesses at most two multi-dimensional array references. 3) Statements may access the same array, in which case they induce a dependence.

NKs are extremely small SCoPs which allow us to capture structural patterns relative to statements and potential dependences between them. Their simplicity also permits us to determine the impact of applied transformations easily, and to test large numbers of ILP combinations, albeit in isolation. Algorithm 1 is used to update the NK-database with any input affine program by using every pair of array references. Given two statements R and S within it, and a predefined performance lexicon L, the algorithm builds NKs from every pair of references, inducing dependences in some cases. The corresponding polyhedral SCoP is then built, and several SCoP features are extracted from it (Sec.4.3). To minimize the potential influence of spurious dependences and other disturbances arising from static features (i.e., a high-stride of an array `A[j][i]` with loop order `ij`), if necessary, we create mock array references fully expanded to the iteration domain dimensionality. If both statements require a mock-array, then different names are used as the objective is not to introduce new dependences nor additional complexities that affect the measurement of the transformation and property to be detected. The SCoP built is transformed against all 2- and 3-permutations of performance idioms produced from the lexicon L, and stored in the $transfos$ set. The particular set of objectives (a tuple of ILP cost functions), together with their order, is stored in the variable *objectives*. The associated legal space (Sec.3.1) is constructed, and the objectives embedded into the ILP. The system is then solved, and an optimized version of the program is generated in *opt*. After this, several output features are computed and composed with the initial `entry_key` and tuple of objectives. Lastly, the entry is then added to the database.

Algorithm 1 Build Nano-Kernel Database

```
Input: L (ILP Performance Lexicon); R,S (Program Statements with
  iteration domains $\mathcal{D}^R$ and $\mathcal{D}^S$, access functions $F_R^A$ and $F_S^B$, and
  schedules $\Theta^R$ and $\Theta^S$); arch (Architecture Features); DB (current
  NK-database);
Output: DB (updated NK-database);
  /* L : lexicon defined in Table 1 */
  transfos ⟵ Permutations (L,2)
  transfos ⟵ transfos $\bigcup$ Permutations (L,3)
  for all $F_R^A \in R$ do
    for all $F_S^B \in S$ do
      SCoP ⟵ create_SCoP($\mathcal{D}^R$,$\mathcal{D}^S$,$F_R^A$,$F_S^B$,$\Theta^R$,$\Theta^S$)
      /* Dependence key features as per Section 4.3 */
      entry_key ⟵ extract_SCoP_features (SCoP)
      for all objectives ∈ transfos do
        ilp ⟵ build_legal_space (SCoP)
        ilp ⟵ embed_objectives (ilp, objectives, arch)
        opt ⟵ apply_transformation (SCoP, ilp)
        features_out ⟵ extract_code_features (opt)
        entry ⟵ {entry_key, objectives, features_out}
        DB.insert (entry)
      end for
    end for
  end for
```

4.3 Input SCoP Features and Dependence Keys

To identify the multiple database entries associated to a single Nano-Kernel we extract a **dependence key**, which is a simple vector of attributes. The attributes we consider are the following:

Table 2: Features used in profitability score calculation.

Variable	Metric Description	Type (Range)
fusionF	Fusion Score	Integer (≥1)
strideF	Stride Score	Integer (≥1)
locF	Lines of Code	Integer (≥ depth)
unrollable	NK can be unrolled	Boolean
noOutPar	No. of Outer Parallel Loops	Integer (≥ 0)
noSimdPar	No. of SIMD Parallel Loops	Integer (≥ 0)
noCores	No. of Processor Cores	Integer
noInner	No. of Inner Loops	Integer (≥1)
vecLen	SIMD Vector Length	DP/SP (4/8)
localityScore	Aggregate Locality Score	
parallelScore	Aggregate Parallelism Score	
locParScore	Unified Locality & Parallelism Score	

a) A boolean variable (0 or 1) to determine whether it's an inter-statement dependence or a self-dependence; b) The number of loop dimensions of the source and target statement; c) The number of dimensions of the source and target array reference; d) The dependence type: none, flow, anti, read or write; e) The cardinality of a dependence: 1-to-1, 1-to-N, N-to-1 or N-to-N; f) Boolean flags to indicate whether the source or target statement are triangular iteration domains; g) One boolean flag to record whether the iteration domain of each statement is skewed; h) An integer storing the number of array dimensions which have a constant shift (i.e., for A[i+1][j][k-2] would be 2); i) Finally, a boolean that states if the array reference has two or more iterators in any dimension. The above attributes are computed for each NK and suffice to capture, identify, and differentiate a large fraction of dependences that can be found in affine programs. As a dependence polyhedron is an affine relation between two iteration domains [42], we compute the cardinality of this relation as a property: Bijective (or 1-to-1) iff each source statement instance is associated with a single target statement instance; 1-to-N, iff a single statement instance is the dependence source of multiple statement instances; N-to-1, is the reciprocal case. Finally, the N-to-N case happens when a dependence instance has multiple source statement instances and multiple target statement instances (i.e., two back-to-back reductions on the same scalar variable).

$$\begin{aligned} &localityScore\,(fusionF, strideF, locF, unrollable) = \\ &\qquad fusionF \times strideF \times locF / unrollS(unrollable) \\ &unrollS(unrollable) = \begin{cases} vecLen & \text{if } unrollable = true \\ 1.0 : & \text{otherwise} \end{cases} \end{aligned} \tag{2}$$

4.4 Architecture Independent Metrics, Architecture Dependent Choices: Assessing the ILP Cost Functions Impact

At compile-time, the output features of the database entries serve as the input to the profitability score function used in **run_query** of Algorithm 2. Output features capture three classes of profitability: parallelism only, locality only, and combined locality and parallelism. Features that capture parallelism involve the number of outer parallel loops, inner parallel loops, number of cores, and the SIMD-vector length. The count of parallel loops can be zero (no parallelism), one (fused-parallel), two (distributed-parallel), or higher (parallel but likely with some light code explosion). Table 2 describes the input and output variables used in Eq.2-4.

In a similar spirit, fusion scores (Eq.2) can assume the value of 1 (perfect fusion across all loop levels) or higher. We assign weights of the form 2^l to each loop level, from the innermost dimension ($l = 0$) and increasing outward. The four branch-conditions in Eq.3 are used to handle potential zero-values. The intuition behind this score is that, ideally, the work will be performed by the outer-parallel loops and distributed (divided) among the parallel hardware units (cores, SIMD-units, and the number of inner parallel loops). This case is covered by the first branch. If only outer-parallel loops are found, then the score only benefits from coarse-grained parallelism (second branch). Contrary, if only inner-parallel loops are found, the parallel profitability will be the number of inner-parallel loops divided by the vector length (third branch). The division is justified in that lower, non-zero scores will be preferred. We also note that, with today's hardware trends, it is increasingly common for the number of hardware-cores in a processor to exceed the SIMD-vector length. Thus, the second branch will normally produce a smaller (preferred) value than the third branch. Finally, if no parallelism is exhibited, then the parallel score results in a constant loss.

$$\begin{aligned} &parallelScore\,(noOutPar, noSimdPar, noCores) = \\ &\begin{cases} noOutPar/(noCores \times noSimdPar \times vecLen) : \\ \qquad \text{if } noOutPar > 0 \wedge noSimdPar > 0 \\ noOutPar/(noCores) : \\ \qquad \text{if } noOutPar > 0 \wedge noSimdPar = 0 \\ noSimdPar/vecLen : \\ \qquad \text{if } noOutPar = 0 \wedge noSimdPar > 0 \\ noCores \times vecLen : \text{otherwise} \end{cases} \end{aligned} \tag{3}$$

$$\begin{aligned} &locParScore(noCores, noInner) = \\ &\begin{cases} noCores \times (1 + noInner) : \text{is multi-core} \\ noCores/(1 + noInner) : \text{is many-core} \end{cases} \\ &nanoKernelScore = localityScore \times \\ &\qquad parallelScore \times locParScore \end{aligned} \tag{4}$$

Lastly, we assume that the balance between locality and parallelism (Eq.4) is mainly captured by the number of innermost loops, where higher values translate to lower locality. This assumption is based on the polyhedral scanning process and the common observation that compact code normally performs better. For instance, the number of innermost (parallel or sequential) loops can drastically increase by multi-dimensional loop shifting, or by fusing loop nests that have been skewed in different ways. Both can also impact the number of lines of code produced (variable *locF* in Eq.2). As such, we assume that parallelism has likely been maximized when the number of innermost loops is 1, otherwise loop distribution would happen at some loop level, increasing the number of loops lexically observable in the transformed code. Similarly to Eq.2 and Eq.3, lower parallelism+locality scores (Eq.4) are still always preferred, but for different reasons. We penalize finding more inner-loops in the multi-core case (locality loss), while rewarding in the many-core case (increase in parallelism). We note that both the parallelism score and the locality score already prefer fewer inner loops.

The scoring function described permits us to extrapolate the impact of applying arbitrary (i.e., non-hard-wired) set of objective functions (and their order) on any affine code. The NK-database is the main building block for designing and implementing our scheme. The metrics stored in the database are mostly architecture-independent—they represent properties exposed and enforced in the output/transformed code—making it possible to search and select objectives that produce specific properties for specific architectures or for ones that are similar enough. Equations 2-4 result in a single number, with lower non-zero values being better. Our strategy offers the advantage that scores can be adapted to architectural features. For a many-core processor, we could prefer properties that expose outer coarse-grained parallelism, whereas, for a 10-core processor, we could favor transformations that trade parallelism for a higher locality.

4.5 Ranking Dependences

Before selecting the ILP objectives that best optimize a given SCoP, we proceed to order the dependences *local to the current statement partition*. The motivation is to use the most influential dependences to decide the overall transformation strategy. Sorting dependences also allows us to cover as many of them as possible with a single subset of objectives, since a particular dependence structure could exist between several pairs of statements. We first prioritize dependences that involve statements with maximum loop depth. The second criteria, is to favor dependences accessing the larger arrays. Third, we prefer to use intra-statement dependences over inter-statement ones. This last choice serves to prioritize parallelism. As an illustration, a dependence involving two 3-dimensional statements takes higher precedence over any dependence relating two 1-dimensional statements.

4.6 Statement Partitioning

Next we describe a few control knobs that permit the scheduler to *adapt* the selection of ILP objectives to specific dependences patterns and which leads to a simple form of domain-specific compilation. The scheduling mechanism being described so far is greedy in nature. It chooses a small subset of dependences to determine the overall behavior of the SCoP. While the interplay among statements stemming from dependences and data reuse is vital to achieve strong performance, an overarching goal of this work is to deliver a flexible adaptive scheduler that produces customized and robust transformations, i.e., that depart from general “one size fits all” approaches and hard-wired cost models. Flexibility in our scheduler is attained by defining five *partition policies* that lead to equivalence classes among statements. Statements in the partition are subjected to the same set of transformations. The overall principle is that statements in an equivalence class should have similar behavior, if not identical. Effects of a partition policy are enforced by the scheduler when choosing dependences, since it only selects dependences *local to the partition*, i.e., where both the source and target of the dependence reside in the same partition. The five equivalence classes are: 1) Dim: Statements in a partition should have identical number of surrounding loops. 2) Max: Consider each statement its own partition. In practice, this amounts to focusing on single statement properties (i.e., parallelism, stride locality). This policy ignores potential inter-statement locality, but which is regained by minimizing the coefficients on scalar dimensions. 3) Selfdep: Statements belonging to a partition should have identical number of self-dependences. 4) Single: A single partition with all dependences. 5) Write: Statements in the partition should write to the same array. In most cases, this results on using inter- and intra- statement, flow and write dependences to drive the scheduling decisions. We note that the above partition policies are unaware of the program dependence graph and its Strongly Connected Components (SCCs). This means that some policy could consider statements in different SCCs for loop fusion or, vice-versa, statements in the same SCC for loop distribution. Thus, to avoid unnecessary objectives, which would not result in useful transformations, we identify these scenarios and skip their insertion in the ILP system.

4.7 Selecting ILP Objectives

Next, we use the *nano-kernel* space previously built to implement and customize the selection of ILP cost functions. The motivation for this lays on a critical premise: **that only a few dependences and structural properties are necessary to decide the overall behavior of a partition of statements**. The steps to build the final ILP system, while selecting the set of objectives together with their priority is shown in Algorithm 2.

Algorithm 2 Select ILP Cost Functions

```
Input: $P_i$ (statement partition), L (ILP Performance Lexicon), NK-
    database (Nano-Kernel Database), prefix (Previously selected ILP
    objectives)
Output: suffix (Sequence of ILP objectives for $P_i$)
  /* Select dependences from the current partition */
  deps ← select_dependences_from_partition ($P_i$);
  /* Sort list of dependences as per Sec.4.5 */
  deps.sort ();
  suffix ← [];
  lexicon ← L;
  while deps $\neq \varnothing$ and lexicon $\neq \varnothing$ do
    current ← get_next_dependence (deps);
    /* Build dependence key as per Sec. 4.3 */
    key ← compute_dependence_key (current);
    /* Determine the cost functions to append. Priority*/
    /* of cost function defined by its position */
    /* in the objectives vector. See also Section 4.8*/
    objectives ← run_query (NK-database, key, prefix);
    if objectives $\neq \varnothing$ then
      suffix.append (objectives);
      lexicon ← lexicon - objectives;
    end if
  end while
  return suffix;
```

Algorithm 2 leverages several properties of convex optimization. It is applied to each statement partition produced by a given policy (Sec.4.6). The final result is a vector $s = \{O_{p_1}, O_{p_2}, ..., O_{p_k}\}$, where the sub-indices $p_1, p_2, ..., p_k$ indicate the priority of the objective. In particular, if $i < j$ for some O_{p_i} and O_{p_j}, then objective O_{p_i} has a higher priority than O_{p_j}. In a nutshell, this algorithm computes and appends suffixes $O_{p_l}..O_{p_{l+d}}$ (with $d = 2 \vee d = 1$) to a growing vector `suffix`. Note that, O_{p_l} (the leading objective of the new suffix), and the last objective in `prefix` (the previously selected objectives), must match. This constraint guarantees that the specific order of objectives is: a) maintained, and b) the transformation effects are compounded and aggregated. The first iteration of the above algorithm can choose, for example, to maximize outer and inner parallelism, while the second iteration could maximize inner parallelism and fusibility of loops. Ultimately, this yields the effect of maximizing all three properties in a single transformation, but restricted to the set of statements belonging to the current partition. Equations 2-4 are used in **run_query** to evaluate and compare all possible transformation candidates (small permutations of ILP objectives) that meet the current selection criteria.

It is possible to not add new ILP objectives for every dependence within the current statement partition. This happens when the number of dependence polyhedra within the partition is larger than the number of ILP objectives comprising the lexicon, at which case we run out of objectives before dependences, or on the contrary, we exhaust the dependences before the objectives (as in the `2mm case`). So the selection loop terminates with either one of these two cases. Besides, we highlight that it is possible to have several structurally similar dependence polyhedra whose patterns are covered by previously added ILP objectives. Continuing with our `2mm` example, if statements S2 and S4 are grouped into the same partition, and the reduction dependence S2⟶S2 is selected to run the query, then this dependence pattern, which repeats for S4⟶S4, is also covered.

4.8 Avoiding Conflicting Objectives

To dynamically compute transformation recipes, the selection of ILP cost functions from the NK-database performed by **run_query** enforces several other filters. We thus pass the vector of previously selected ILP objectives in the `prefix` of Algorithm 2. First, we avoid selecting two or more ILP cost functions that attempt to extract the same property in different manners. That is, we never select two objectives from the set {OPIR, ADA, SKEWPAR, SDC} nor from {IP, VSKW}. The second restriction is to limit the number of ILP objectives per statement partition to at most six (out of the 12 cost functions considered). This design restriction is because extracting outer coarse-grained parallelism, SIMD-parallelism, good stride access, fusion, and permutability (for the legality of loop unrolling) already accounts for five objectives. Third, the selection must also enforce that the new subset of objectives must not intersect with previously chosen ones (except for one). Fourth, the selected vector of objectives for any query must achieve the best profitability score (lower is better). Fifth, the transformation candidate must, obviously, match the dependence key provided. Lastly, we leverage a *consistency criteria* to ensure that the new subset of objectives does not contradict any of the previously selected ones. Specifically, we require the leading objective to match the objective with the lowest current priority already embedded in the ILP system, i.e., the last objective currently in the vector. For example, if the scheduler has already selected objectives ⟨ SO, DGF, SIS ⟩, then the second query must start with the SIS objective. This is the key to compound and aggregate performance benefits with ILP objectives.

5 EXPERIMENTAL EVALUATION

We run our experiments on two multi-core systems equipped with a 12-core AMD Threadripper 2920X (with 1 thread per core) and a 48-core Intel Xeon Platinum 8160 processor (dual-socket, 24 cores per socket, 2 threads per core). As a reference, a DGEMM achieves 173.85 GFLOP/s in the former while 0.75 TFLOP/s for the latter. Besides, we run our benchmarks on a many-core system equipped with an Intel Xeon Phi 7250 processor (68 cores, 2 threads per core), with a peak performance of 1.5 TFLOP/s. The many-core system requires more parallelism to obtain peak performance than the traditional multi-core machines. We include the many-core system in our evaluation to show that our scheduler achieves high-performance across a broad range of target systems. We refer to these systems as AMDTR (Threadripper), SKL (Xeon Platinum), and KNL (Xeon Phi) for brevity.

We implemented the scheduling strategy and nano-kernel building approach in the PoCC compiler [31], which is, to the best of our knowledge, the only open-source polyhedral compiler implementing the legal space single-shot scheduling engine.

We use the PolyBench/C-3.2 benchmark suite [32], and compare our scheduler against 3 baselines: the best performance in a tiled and fusion auto-tuning space generated with the Pluto compiler (PLT-AT), version 0.11.4. The default performance obtained with Pluto (PLT-DF), which uses tile sizes of 32^d and the smartfuse heuristic, and MDT, the model-driven transformation approach described in [23]. The PLT-AT space considers three fusion heuristics implement in Pluto: Nofuse that amounts to maximal loop distribution, maximum possible parallelism, but lowest locality. Maxfuse that maximizes loop fusion, likely diminishing parallelism. Smartfuse, roughly a middle ground that balances parallelism and locality. Tile sizes are chosen, such as the kernel's footprint fits between the L1 and L2 cache sizes. We select tile values traversing powers-of-2–between 1 and, 512 or 1024, depending on the benchmark–and non-powers-of 2 as done in [23]. Our criteria produces 2188 different source variants for `gemm`, over 7200 variants for `doitgen`, and 768 variants for `gemver`. We omit results for the `ludcmp` kernel since all loop transformations are disallowed by multiple scalar dependences, rendering its performance distribution as flat as `cholesky`'s. Lastly, several `symm` variants are removed due to code explosion and AOCC failing to compile them for the AMDTR platform. We filter out variants exceeding the 512KB of source code size.

We generated for all benchmarks the 5 transformed versions corresponding to the partition policies described in Sec.4.6. We recall that our scheduler produces "new transformation recipes" by deciding, for each statement partition, the subset of ILP objectives to apply together with their priority (the position in which the optimization variable is placed in the overall ILP system). A NK-database of transformations is generated for each benchmark, consisting of $O(|A|^2 \times {}^LP_3)$ entries, where $|A|$ is the number of array references in the benchmark, L is the size of the performance lexicon, and ${}^LP_3$ is the number of 3-permutations from L. *Although in our evaluation each benchmark is used to build its own NK-database, we note that this need not be the case, as a single database can be constructed in a fully-offline one shot fashion, thereby substantially reducing the end-to-end time for possible auto-tuning*. As an example,

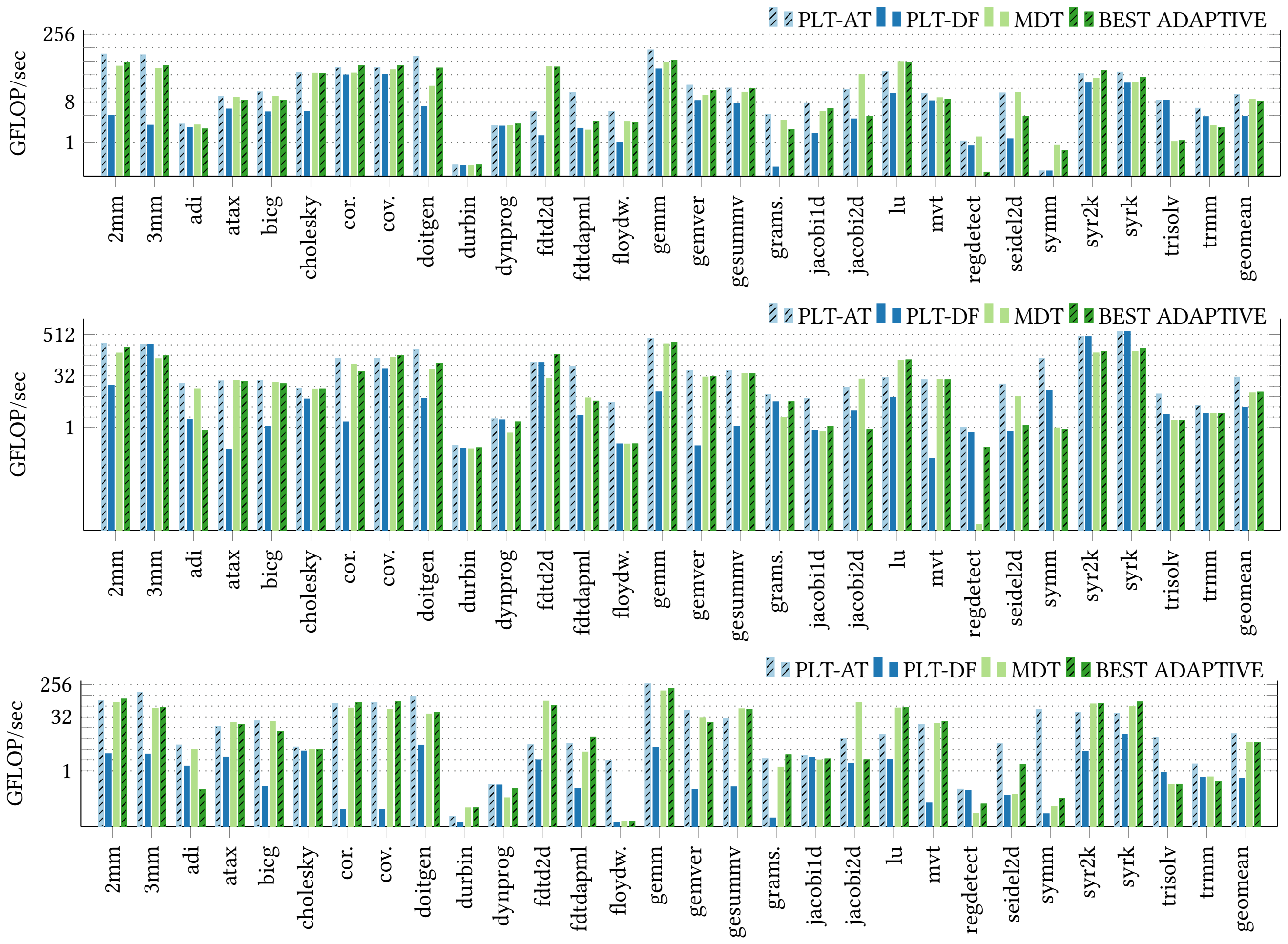

Figure 4: Polybench performance summary in log scale (GFLOP/s): 12-core AMDTR (top), 48-core SKL (middle), and 68-core KNL (bottom) systems for the different schedulers (Double Precision, Standard Dataset).

a lexicon consisting of 6 objectives, and a benchmark with 6 array references (e.g., dgemm), takes approximately 1.2 hours. We limit L to six objectives (OPIR, ADA, SIS, SO, DGF, IP) for all linear algebra kernels (non-stencils) and use the full lexicon (all 12 ILP objectives) for stencils and stencil-like benchmarks. The motivation for this design is that most linear algebra kernels already exhibit sufficiently large optimization spaces, which benefit more from fusion exploration, and exploiting parallelism and intra-statement locality via loop permutation. In contrast, stencil benchmarks require more pattern-specific objectives (i.e., SDC, SPC, VSKW) to find strong performing transformations.

After the polyhedral optimization, we run vendor-specific compilers to lower the generated code. In particular, we run AOCC on the AMD system and ICC 2018 on the Intel machines. We set both compilers to the `-O3` optimization level. We use OpenMP to generate multi-threaded code and pass additional flags to AOCC (`-unroll-full-max-count, -finline-aggressive, -flto`) and ICC (`-funroll-all-loops`) targetting its AVX512 vector ISA. All measurements are collected using double precision, standard dataset, and report the average value of 5 runs removing the best and worse time.

Summary: Our evaluation aims to demonstrate that polyhedral *adaptive scheduling (non-hardwired)*, leveraging a performance lexicon, can produce (novel) transformation recipes that translate into strong performance competitive with auto-tuning spaces. We choose the Polybench suite since it is already designed to cover most of the "polyhedral dwarfs" scenarios. It contains a breadth of kernels showcasing loop nests of different dimensionality and shape, various affine array access patterns and dependences with different structures. Moreover, while our evaluation is Polybench-specific, both the offline and online stages of our scheduling framework can be used on other benchmarks suites (e.g., Rodinia) with suitable SCoPs. As the creation of the NK-database of transformations is meant as an offline phase, one-time cost, our evaluation incurs on a minimal online overhead due to the need of evaluating only five transformed variants for each input. Our scheduling strategy preserves the benefits of model-driven transformations while

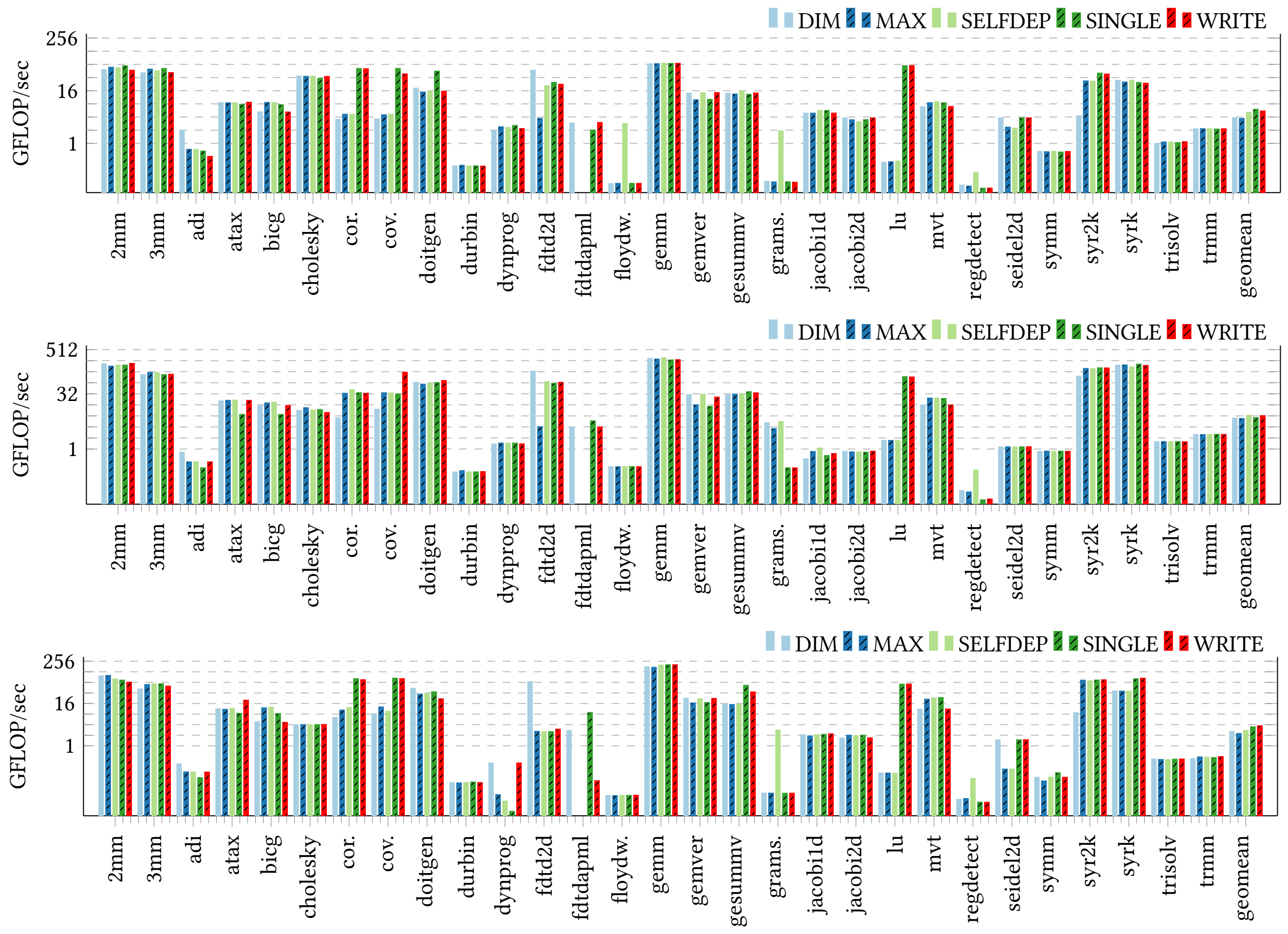

Figure 5: Impact of Statement Partitioning Policy on the AMD Threadripper (top), Intel Xeon Platinum (middle) and Intel Xeon Phi (bottom)

increasing the scheduler's flexibility and transformation robustness. For instance, the KNL and SKL MDT variants are precisely those used in [23], but their work lacked a customized recipe for the Threadripper architecture. We reuse the SKL MDT mode for AMDTR (given that they are both multi-core systems with similar memory hierarchy). In that sense, the MDT transformation for AMDTR, while in average obtain as good performance as the *Best Adaptive*, they are not fully customized to the architecture. Steering our attention onto the PLT-AT space, we observe from Figure.4 that, even though in most cases, we do not match the best performance in the PLT-AT space, we are in average approximately 1.5× and 2.5× slower than its best variant. Nonetheless, the *Best (of 5) Adaptive* variants represents a significant reduction in online tuning time, as we only resort to explicitly generating and evaluating 5 benchmark versions. Lastly, the *Best Adaptive* is at least 2× faster than PLT-DF (Pluto default), across all architectures, and peaking at 8× better in the KNL.

Overall, we observe in Figure 4 that the best adaptive variant achieves strong performance in the BLAS-2 and BLAS-3 kernels, as well as benchmarks `correlation` and `covariance`, on all three architectures, being 10× better than PLT-DF in numerous instances. Some notable improvements over the MDT variants are visible on `doitgen` and `fdtdapml` with the AMDTR and KNL machines. For the former, on the AMD system performance improves 2.5×. This is due to our adaptive scheduling system yielding the transformation recipe ⟨ SO, DGF, SIS, IP, OPIR ⟩ while the MDT variant produces ⟨ SO, IP, OPIR, SIS ⟩. In other words, adaptive scheduling chooses first to fuse, separate independent statements and then exploit outer parallelism (`doitgen` having multiple levels of it), while MDT resorts to maximize data reuse by serializing the outermost loop, and parallelizing the second outermost dimension and maintaining all 3 statements distributed. In `fdtdapml`'s case, the best adaptive variant fuses all 16 statements while successfully parallelizing the outermost loop, satisfying its 86 dependences via a combination of loop skewing of its inner space dimensions, time-shifting, and loop distribution at the second scalar dimension. Finally, we can also appreciate some remarkable performance for `cholesky`, relative to `gemm`, which is owed to truly aggressive loop unrolling performed by the AOCC compiler. Next, we explain the performance impact of the five partitioning policies (Sec.4.6). Figure 5 shows the performance achieved for all policies across all benchmarks and on all architectures. We observe that most BLAS-3 and BLAS-2 linear algebra benchmarks exhibit minimal variation among policies. For BLAS-3 kernels, the scheduler favors transformations that maximize first inner parallelism (IP), stride access optimization (SO), and outer coarse-grained parallelism (OPIR/ADA). For instance, in `3mm` with *write* policy, our scheduler produces 3 statement partitions and prioritizes the self-dependence on the main compute loop for each Matmul. Datamining benchmarks such as `correlation` and `covariance`, although similar to BLAS-3 kernels, benefit more from considering larger statement partitions, as this enables the scheduler to avoid fusion inhibiting transformations. As an illustration, the *single* policy for `correlation` produces the transformation recipe ⟨ DGF, SIS, OPIR, IP, SO, ADA ⟩ (DGF with the highest priority), whereas the *write* policy produces two distinct sets of transformations: ⟨ IP, SO, ADA, DGF, OPIR ⟩ for partitions #1–#3, and ⟨ DGF, SIS, OPIR, IP, SO ⟩ for statement partition #4, which is the performance dominating one. In practice, almost the same set of objectives. On the other hand, the *selfdep* policy produces only two partitions, but with a significant loss in performance due to fusion of the most compute-intensive statement with a lower-dimensional statement (resulting from having DGF with the highest priority). In turn, this induces an outermost serial loop. A similar phenomenon is observed for `lu` across all architectures. The previous two partitioning policies avoid the phenomenon. Lastly, the underlying reasons for competitive performance on stencil kernels vary. For instance, `fdtd2d` largely benefits from the *dim* policy, using the dependences among the three full-dimensional statements with ⟨ SDC, SO, VSKW, ADA ⟩.

On the Optimality of the Scheduling Approach

Our adaptive ILP selection is greedy in nature, and does not guarantee an optimal global solution. However, as each objective embedded maximizes some performance property, in practice our scheduler consistently delivers strong performance, as shown in Figures 4 and 5. Besides, we rely on the compounded effect of using multiple ILP cost functions designed in a convex space to achieve good performance at the cost of minor space exploration. In most cases, all five partitioning policies deliver nearly identical performance, surpassing that obtained with the Pluto compiler and its default options (smart-fusion with 32^d tile sizes). Nonetheless, we can still get unlucky at times and observe some significant performance variation. For instance, Polybench's `correlation` and `covariance` benchmarks achieve their best performance when using the *single* and *write* partitioning policies because they give a better view of the SCoP. This caveat is not different from exposing many fusion heuristics without complete certainty of which one performs best. Finally, ranking and prioritizing dependences also contributes to selecting ILP objectives that address first the most compute intensive statements (e.g. S2 and S4 in `2mm`).

6 RELATED WORK

Fully-automatic polyhedral compilers: Decades of research in fully-automatic compilation lead to sophisticated general-purpose optimizers such as Polly [18] and Pluto [8]. Although general-purpose compilers boost productivity, they do not always obtain good performance due to the low level at which they need to operate and their one-size-fits-all optimization strategy [4]. Our scheduler, on the other hand, carefully crafts the transformation objectives based on the architectural feature of the target machine, while maintaining the high productivity of general-purpose compilers.
User-driven transformers: Multiple works expose a scheduling language on top of the polyhedral model through directives or pragmas. UTF been arguably the very first of them [21]. AlphaZ expresses program transformations as a set of equations based on the Alpha language [47]. The Xlanguage allows users to generate multi-version programs by specifying the type of transformation to apply as well as the transformation parameters [13]. Along the same line, URUK, Loopy, Orio, LOCUS and Chill expose the user loop transformations to exploit the deep memory hierarchy and parallelism of modern machines [10, 16, 20, 30, 37]. Bagnères et al. with Chlore and Clay opened the polyhedral black-box allowing more close interaction between compiler and users [5]. Their work enables users to examine, refine, and freeze a sequence of loop transformation directives. Recently, Kruse et al. proposed an extension to the OpenMP pragmas to broaden the range of transformation in the Clang compiler and allows the composition of them [25]. Baghdadi et al. introduced TIRAMISU, a framework that features a scheduling language to target multiple platforms from multi-core to distributed machines [4]. Halide [35] uses interval analysis to represent loop bounds; therefore, it cannot naturally represent non-rectangular iteration spaces. The same applies to another non-polyhedral compiler: TVM [11]. Yi et al. proposed POET, which allows the user to apply a set of transformations to arbitrary programming languages via XML-based transformation scripts [46]. Vocke et al, extended Halide to target DSPs and add explicit scheduling commands to move data to/from scratchpad memories [45]. Our compiler, on the other hand, bridges the gap between productivity (of general-purpose compiler) and performance (of directive-based transformers) by using high-level objectives. The high-level objectives are lowered to low-level optimization by an automatic scheduler.

7 CONCLUSION AND FUTURE WORK

We have presented a scheduler capable of generating transformation recipes in an automatic fashion. The scheduler queries a database of nano-kernels, prioritizes dependences and exploits common properties among statements to produce partitions that behave similarly. We deliver performance on multi- and many-core systems competitive and, in some cases, superior to two state-of-the-art polyhedral optimizers.

ACKNOWLEDGMENT

This work was partially supported by the European Commission Horizon 2020 programme through the NeMeCo grant agreement, id. 780215 as well as the Swiss National Science Foundation under the Ambizione program (grant PZ00P2168016), ARM Holdings plc and Xilinx Inc in the context of Polly Labs, and resources provided by the Extreme Science and Engineering Discovery Environment (XSEDE), which is supported by National Science Foundation grant number ACI-1548562.

REFERENCES

[1] Aravind Acharya and Uday Bondhugula. 2015. PLUTO+: Near-complete Modeling of Affine Transformations for Parallelism and Locality. In Proceedings of the 20th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP 2015). ACM, New York, NY, USA, 54–64. https://doi.org/10.1145/2688500.2688512

[2] Aravind Acharya, Uday Bondhugula, and Albert Cohen. 2018. Polyhedral Auto-transformation with No Integer Linear Programming. In Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2018). ACM, New York, NY, USA, 529–542. https://doi.org/10.1145/3192366.3192401

[3] Jason Ansel, Shoaib Kamil, Kalyan Veeramachaneni, Jonathan Ragan-Kelley, Jeffrey Bosboom, Una-May O'Reilly, and Saman Amarasinghe. 2014. OpenTuner: An Extensible Framework for Program Autotuning. In Proceedings of the 23rd International Conference on Parallel Architectures and Compilation (PACT '14). ACM, New York, NY, USA, 303–316. https://doi.org/10.1145/2628071.2628092

[4] Riyadh Baghdadi, Jessica Ray, Malek Ben Romdhane, Emanuele Del Sozzo, Abdurrahman Akkas, Yunming Zhang, Patricia Suriana, Shoaib Kamil, and Saman Amarasinghe. 2019. Tiramisu: A polyhedral compiler for expressing fast and portable code. In Proceedings of the 2019 IEEE/ACM International Symposium on Code Generation and Optimization. IEEE Press, 193–205.

[5] Lénaïc Bagnères, Oleksandr Zinenko, Stéphane Huot, and Cédric Bastoul. 2016. Opening Polyhedral Compiler's Black Box. In Proceedings of the 2016 International Symposium on Code Generation and Optimization (CGO'16). ACM, New York, NY, USA, 128–138. https://doi.org/10.1145/2854038.2854048

[6] Mohamed-Walid Benabderrahmane, Louis-Noël Pouchet, Albert Cohen, and Cédric Bastoul. 2010. The Polyhedral Model is More Widely Applicable Than You Think. In Proceedings of the 19th Joint European Conference on Theory and Practice of Software, International Conference on Compiler Construction (CC'10/ETAPS'10). Springer-Verlag, Berlin, Heidelberg, 283–303. https://doi.org/10.1007/978-3-642-11970-5_16

[7] Uday Bondhugula, Albert Hartono, J. Ramanujam, and P. Sadayappan. 2008. A Practical Automatic Polyhedral Parallelizer and Locality Optimizer. In Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '08). ACM, New York, NY, USA, 101–113. https://doi.org/10.1145/1375581.1375595

[8] Uday Bondhugula, Albert Hartono, Jagannathan Ramanujam, and Ponnuswamy Sadayappan. 2008. A practical automatic polyhedral parallelizer and locality optimizer. In Acm Sigplan Notices, Vol. 43. ACM, 101–113.

[9] Chun Chen, Jacqueline Chame, and Mary Hall. 2005. Combining Models and Guided Empirical Search to Optimize for Multiple Levels of the Memory Hierarchy. In Proceedings of the International Symposium on Code Generation and Optimization (CGO '05). IEEE Computer Society, Washington, DC, USA, 111–122. https://doi.org/10.1109/CGO.2005.10

[10] Chun Chen, Jacqueline Chame, and Mary Hall. 2008. CHiLL: A framework for composing high-level loop transformations. Technical Report. USC Computer Science.

[11] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Yan, Haichen Shen, Meghan Cowan, Leyuan Wang, Yuwei Hu, Luis Ceze, et al. 2018. {TVM}: An automated end-to-end optimizing compiler for deep learning. In 13th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 18). 578–594.

[12] Yufei Ding, Jason Ansel, Kalyan Veeramachaneni, Xipeng Shen, Una-May O'Reilly, and Saman Amarasinghe. 2015. Autotuning Algorithmic Choice for Input Sensitivity. In Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '15). ACM, New York, NY, USA, 379–390. https://doi.org/10.1145/2737924.2737969

[13] Sebastien Donadio, James Brodman, Thomas Roeder, Kamen Yotov, Denis Barthou, Albert Cohen, María Jesús Garzarán, David Padua, and Keshav Pingali. 2006. A Language for the Compact Representation of Multiple Program Versions. In Languages and Compilers for Parallel Computing, Eduard Ayguadé, Gerald Baumgartner, J. Ramanujam, and P. Sadayappan (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 136–151.

[14] Paul Feautrier. 1991. Dataflow analysis of array and scalar references. International Journal of Parallel Programming 20, 1 (01 Feb 1991), 23–53. https://doi.org/10.1007/BF01407931

[15] Paul Feautrier. 1992. Some efficient solutions to the affine scheduling problem. I. One-dimensional time. International Journal of Parallel Programming 21, 5 (01 Oct 1992), 313–347. https://doi.org/10.1007/BF01407835

[16] Sylvain Girbal, Nicolas Vasilache, Cédric Bastoul, Albert Cohen, David Parello, Marc Sigler, and Olivier Temam. 2006. Semi-automatic Composition of Loop Transformations for Deep Parallelism and Memory Hierarchies. Int. J. Parallel Program. 34, 3 (June 2006), 261–317. https://doi.org/10.1007/s10766-006-0012-3

[17] Tobias Grosser, Armin Groesslinger, and Christian Lengauer. 2012. Polly—performing polyhedral optimizations on a low-level intermediate representation. Parallel Processing Letters 22, 04 (2012), 1250010.

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

[19] Mary Hall, Jacqueline Chame, Chun Chen, Jaewook Shin, Gabe Rudy, and Malik Murtaza Khan. 2010. Loop Transformation Recipes for Code Generation and Auto-tuning. In Proceedings of the 22Nd International Conference on Languages and Compilers for Parallel Computing (LCPC'09). Springer-Verlag, Berlin, Heidelberg, 50–64. https://doi.org/10.1007/978-3-642-13374-9_4

[20] A. Hartono, B. Norris, and P. Sadayappan. 2009. Annotation-based empirical performance tuning using Orio. In 2009 IEEE International Symposium on Parallel Distributed Processing. 1–11. https://doi.org/10.1109/IPDPS.2009.5161004

[21] Wayne Kelly and William Pugh. 1992. A framework for unifying reordering transformations. Technical Report. University of Maryland.

[22] Martin Kong and Louis-Noël Pouchet. 2018. A Performance Vocabulary for Affine Loop Transformations. CoRR abs/1811.06043 (2018). arXiv:1811.06043 http://arxiv.org/abs/1811.06043

[23] Martin Kong and Louis-Noël Pouchet. 2019. 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). ACM, New York, NY, USA, 469–484. https://doi.org/10.1145/3314221.3314653

[24] Martin Kong, Richard Veras, Kevin Stock, Franz Franchetti, Louis-Noël Pouchet, and P. Sadayappan. 2013. When Polyhedral Transformations Meet SIMD Code Generation. In Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '13). ACM, New York, NY, USA, 127–138. https://doi.org/10.1145/2491956.2462187

[25] Michael Kruse and Hal Finkel. 2018. A Proposal for Loop-Transformation Pragmas. In Evolving OpenMP for Evolving Architectures, Bronis R. de Supinski, Pedro Valero-Lara, Xavier Martorell, Sergi Mateo Bellido, and Jesus Labarta (Eds.). Springer International Publishing, Cham, 37–52.

[26] Leslie Lamport. 1974. The Parallel Execution of DO Loops. Commun. ACM 17, 2 (Feb. 1974), 83–93. https://doi.org/10.1145/360827.360844

[27] Amy W. Lim, Gerald I. Cheong, and Monica S. Lam. 1999. An affine partitioning algorithm to maximize parallelism and minimize communication. In Proceedings of the 13th international conference on Supercomputing, ICS 1999, Rhodes, Greece, June 20-25, 1999. 228–237. https://doi.org/10.1145/305138.305197

[28] Sanyam Mehta, Pei-Hung Lin, and Pen-Chung Yew. 2014. Revisiting Loop Fusion in the Polyhedral Framework. In Proceedings of the 19th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP '14). ACM, New York, NY, USA, 233–246. https://doi.org/10.1145/2555243.2555250

[29] Benoit Meister, Nicolas Vasilache, David Wohlford, Muthu Manikandan Baskaran, Allen Leung, and Richard Lethin. 2011. R-stream compiler. In Encyclopedia of Parallel Computing. Springer, 1756–1765.

[30] Kedar S Namjoshi and Nimit Singhania. 2016. Loopy: Programmable and formally verified loop transformations. In International Static Analysis Symposium. Springer, 383–402.

[31] PoCC. 2018. The Polyhedral Compiler Collection. (2018). https://sourceforge.net/projects/pocc/ Online; accessed on September 2018.

[32] Louis-Noël Pouchet. 2012. Polybench: The polyhedral benchmark suite. URL: http://polybench.sf.net (2012).

[33] Louis-Noël Pouchet, Uday Bondhugula, Cédric Bastoul, Albert Cohen, J. Ramanujam, P. Sadayappan, and Nicolas Vasilache. 2011. Loop Transformations: Convexity, Pruning and Optimization. In Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '11). ACM, New York, NY, USA, 549–562. https://doi.org/10.1145/1926385.1926449

[34] W. Pugh. 1991. The Omega test: A fast and practical integer programming algorithm for dependence analysis. In Supercomputing '91:Proceedings of the 1991 ACM/IEEE Conference on Supercomputing. 4–13. https://doi.org/10.1145/125826.125848

[35] Jonathan Ragan-Kelley, Connelly Barnes, Andrew Adams, Sylvain Paris, Frédo Durand, and Saman Amarasinghe. 2013. Halide: a language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines. In Acm Sigplan Notices, Vol. 48. ACM, 519–530.

[36] Michelle Mills Strout, Larry Carter, and Jeanne Ferrante. 2003. Compile-time Composition of Run-time Data and Iteration Reorderings. In Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation (PLDI '03). ACM, New York, NY, USA, 91–102. https://doi.org/10.1145/781131.781142

[37] Thiago S. F. X. Teixeira, Corinne Ancourt, David Padua, and William Gropp. 2019. Locus: A System and a Language for Program Optimization. In Proceedings of the 2019 IEEE/ACM International Symposium on Code Generation and Optimization (CGO 2019). IEEE Press, Piscataway, NJ, USA, 217–228. http://dl.acm.org/citation.cfm?id=3314872.3314898

[38] Ananta Tiwari, Chun Chen, Jacqueline Chame, Mary Hall, and Jeffrey K. Hollingsworth. 2009. A Scalable Auto-tuning Framework for Compiler Optimization. In Proceedings of the 2009 IEEE International Symposium on

Parallel&Distributed Processing (IPDPS '09). IEEE Computer Society, Washington, DC, USA, 1–12. https://doi.org/10.1109/IPDPS.2009.5161054

[39] Nicolas Vasilache. 2007. Scalable Program Optimization Techniques in the Polyhedral Model. Ph.D. Dissertation. University of Paris-Sud 11.

[40] Nicolas Vasilache, Benoit Meister, Muthu Baskaran, and Richard Lethin. 2012. Joint scheduling and layout optimization to enable multi-level vectorization. (2012).

[41] Nicolas Vasilache, Oleksandr Zinenko, Theodoros Theodoridis, Priya Goyal, Zachary Devito, William S. Moses, Sven Verdoolaege, Andrew Adams, and Albert Cohen. 2019. The Next 700 Accelerated Layers: From Mathematical Expressions of Network Computation Graphs to Accelerated GPU Kernels, Automatically. ACM Trans. Archit. Code Optim. 16, 4, Article 38 (Oct. 2019), 26 pages. https://doi.org/10.1145/3355606

[42] Sven Verdoolaege. 2010. isl: An Integer Set Library for the Polyhedral Model. In ICMS, Vol. 6327. Springer, 299–302.

[43] Sven Verdoolaege, Juan Carlos Juega, Albert Cohen, José Ignacio Gómez, Christian Tenllado, and Francky Catthoor. 2013. Polyhedral Parallel Code Generation for CUDA. ACM Trans. Archit. Code Optim. 9, 4, Article 54 (Jan. 2013), 23 pages. https://doi.org/10.1145/2400682.2400713

[44] Sven Verdoolaege and Alexandre Isoard. 2017. Consecutivity in the isl Polyhedral Scheduler. (2017).

[45] Sander Vocke, Henk Corporaal, Roel Jordans, Rosilde Corvino, and Rick Nas. 2017. Extending Halide to Improve Software Development for Imaging DSPs. ACM Trans. Archit. Code Optim. 14, 3, Article 21 (Aug. 2017), 25 pages. https://doi.org/10.1145/3106343

[46] Qing Yi. 2012. POET: A Scripting Language for Applying Parameterized Source-to-source Program Transformations. Softw. Pract. Exper. 42, 6 (June 2012), 675–706. https://doi.org/10.1002/spe.1089

[47] Tomofumi Yuki, Gautam Gupta, DaeGon Kim, Tanveer Pathan, and Sanjay Rajopadhye. 2012. Alphaz: A system for design space exploration in the polyhedral model. In International Workshop on Languages and Compilers for Parallel Computing. Springer, 17–31.

[48] Hans Zima, Mary Hall, Chun Chen, and Jaqueline Chame. 2009. Model-guided Autotuning of High-productivity Languages for Petascale Computing. In Proceedings of the 18th ACM International Symposium on High Performance Distributed Computing (HPDC '09). ACM, New York, NY, USA, 151–166. https://doi.org/10.1145/1551609.1551611

[49] Oleksandr Zinenko, Sven Verdoolaege, Chandan Reddy, Jun Shirako, Tobias Grosser, Vivek Sarkar, and Albert Cohen. 2018. Modeling the Conflicting Demands of Parallelism and Temporal/Spatial Locality in Affine Scheduling. In Proceedings of the 27th International Conference on Compiler Construction (CC 2018). Association for Computing Machinery, New York, NY, USA, 3–13. https://doi.org/10.1145/3178372.3179507