

# A Priori Loop Nest Normalization: Automatic Loop Scheduling in Complex Applications

Lukas Trümper  
Daisytuner  
Darmstadt, Germany  
lukas.truemper@daisytuner.com

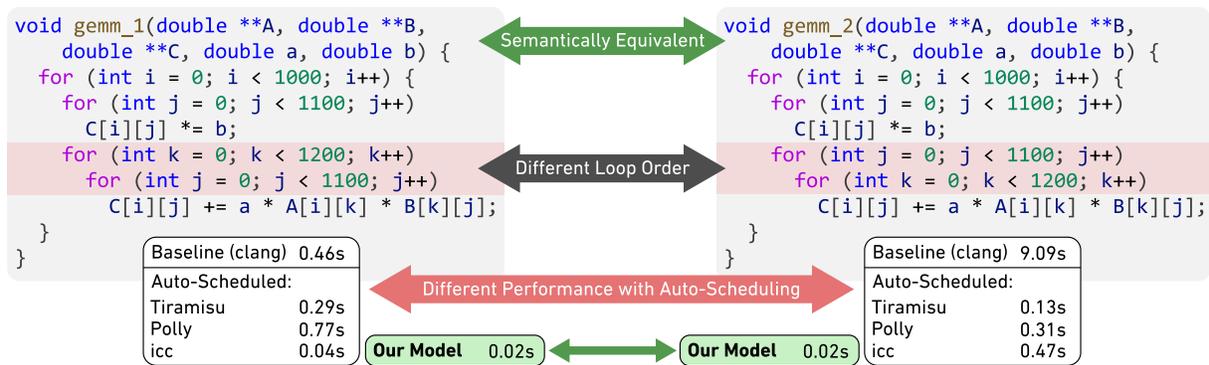
Philipp Schaad  
ETH Zurich  
Zurich, Switzerland  
philipp.schaad@inf.ethz.ch

Berke Ates  
ETH Zurich  
Zurich, Switzerland  
beates@student.ethz.ch

Alexandru Calotoiu  
ETH Zurich  
Zurich, Switzerland  
alexandru.calotoiu@inf.ethz.ch

Marcin Copik  
ETH Zurich  
Zurich, Switzerland  
marcin.copik@inf.ethz.ch

Torsten Hoefler  
ETH Zurich  
Zurich, Switzerland  
htor@inf.ethz.ch



**Figure 1.** Structurally different General Matrix-Matrix Multiply (GEMM) kernels yield significantly different performance.

## Abstract

The same computations are often expressed differently across software projects and programming languages. In particular, how computations involving loops are expressed varies due to the many possibilities to permute and compose loops. Since each variant may have unique performance properties, automatic approaches to loop scheduling must support many different optimization recipes. In this paper, we propose a priori loop nest normalization to align loop nests and reduce the variation before the optimization. Specifically, we define and apply normalization criteria, mapping loop nests with different memory access patterns to the same canonical form. Since the memory access pattern is susceptible to loop variations and critical for performance, this normalization allows many loop nests to be optimized by the same optimization recipe. To evaluate our approach, we apply the normalization

with optimizations designed for only the canonical form, improving the performance of many different loop nest variants. Across multiple implementations of 15 benchmarks using different languages, we outperform a baseline compiler in C on *average* by a factor of 21.13, state-of-the-art auto-schedulers such as *Polly* and the *Tiramisu auto-scheduler* by 2.31 and 2.89, as well as performance-oriented Python-based frameworks such as *NumPy*, *Numba*, and *DaCe* by 9.04, 3.92, and 1.47. Furthermore, we apply the concept to the *CLOUDSC* cloud microphysics scheme, an actively used component of the Integrated Forecasting System, achieving a 10% speedup over the highly-tuned Fortran code.

**CCS Concepts:** • Software and its engineering → Compilers.

**Keywords:** loop normalization, loop optimization, polyhedral analysis, compiler, code optimization

## ACM Reference Format:

Lukas Trümper, Philipp Schaad, Berke Ates, Alexandru Calotoiu, Marcin Copik, and Torsten Hoefler. 2025. A Priori Loop Nest Normalization: Automatic Loop Scheduling in Complex Applications. In *Proceedings of the 23rd ACM/IEEE International Symposium on Code Generation and Optimization (CGO '25)*, March 01–05, 2025, Las Vegas, NV, USA. ACM, New York, NY, USA, 13 pages. <https://doi.org/10.1145/3696443.3708951>



This work is licensed under a Creative Commons Attribution 4.0 International License.

CGO '25, March 01–05, 2025, Las Vegas, NV, USA

© 2025 Copyright held by the owner/author(s).

ACM ISBN 979-8-4007-1275-3/25/03

<https://doi.org/10.1145/3696443.3708951>

## 1 Introduction

Because of sophisticated code optimizations by compilers, developers of high-performance software applications can often conveniently express the same linear code differently while achieving high performance on modern processors. However, the performance of loop-based computations is highly susceptible to the specific permutations and composition of loops chosen by the developer.

Different frameworks implement automatic loop scheduling methods to enable automatic high-performance loop-based code. Since loop scheduling is a complex problem in and of itself [2, 32], these frameworks rely on approximate methods. For instance, *Polly* [18] finds a good schedule by optimization of an integer-linear program (ILP) [8]. However, Baghdadi et al. [4] show that this ILP only covers a certain fraction of the practically-relevant scheduling space [4]. Recent approaches [2, 3] leverage deep learning to search much larger scheduling spaces at the price of local optima.

Although such auto-schedulers can achieve significant speedups, their direct application to large, scientific applications is currently limited. The small example of GEMM presented in Figure 1 already shows that the results of auto-schedulers may vary by factors of  $3\times$ - $10\times$  depending on the chosen loop order. Hence, developers must manually align loop nests to look like the supported optimization recipes.

To enable robust, automatic loop scheduling in complex applications, this paper introduces a priori loop nest normalization for auto-scheduling. Specifically, we identify with *maximal loop fission* and *stride minimization* two normalization criteria, which canonicalize loop nests with different memory access patterns. This method allows us to more easily apply the same optimization recipes to various loop nests with different performance properties. We test the robustness of the optimization plus normalization with multiple semantically equivalent implementations of 15 benchmarks from *PolyBench* [29] across Python and C and with the highly optimized cloud microphysics scheme CLOUDSC written in Fortran. This results in our optimization pipeline outperforming not just baseline compiler results in C ( $\times 21$ ) and Fortran ( $\times 1.1$ ), but also non-normalizing state-of-the-art auto-schedulers such as *Polly* [18] ( $\times 2.3$ ), and the *Tiramisu auto-scheduler* [3] ( $\times 2.9$ ), as well as performance-oriented Python-based frameworks such as *NumPy*, *Numba*, and *DaCe* by 9.04, 3.92, and 1.47. In short, our contributions are

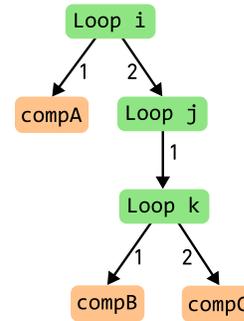
- Definition of a priori loop nest normalization to align loop nests with different performance properties.
- Implementation for LLVM IR and integration with a state-of-the-art loop scheduling algorithm.
- Evaluation of the robustness of optimization plus normalization on representative benchmarks across multiple programming languages as well as a case study optimizing a highly tuned cloud micro-physics simulation.

## 2 A-Priori Loop Nest Normalization for Auto-Scheduling

The cost of moving data through the memory hierarchy is the dominant factor for the performance of modern computing architectures [34]. Therefore, a priori loop nest normalization aims to provide simple memory access patterns as the starting point of the optimization. It should map loop nests with vastly different performance properties such as the *reuse distance* [6, 14, 30] or the *sustained memory bandwidth* to the same canonical form. After formally defining what we understand as loops and computations, we will introduce the two normalization criteria based on well-known compiler transformations.

```
for (int i ...) {
  compA();
  for (int j ...) {
    for (int k ...) {
      compB();
      compC();
    }
  }
}
```

(a) Loop nest pseudocode



(b) Loop nest tree representation

Figure 2. Characterization of loop nests.

**Computation.** We define a *computation* as a unit of work composed of one or more instructions, where exactly one of the instructions is a write of a scalar value to a data container.

**Loop.** A *loop* comprises an iterator with its initial values and update criterion, a termination condition, and a loop body composed of a sequence of computations.

**Loop nest.** A *loop nest* is a loop where the loop body can be composed by an ordered sequence of computations, loops, and loop nests. In the following, we use the notation  $comp[i, j, k]$  if a computation is nested inside the loops  $i$ ,  $j$ , and  $k$ , where  $i$  is the outermost and  $k$  the innermost loop. Similarly, we use the notation  $loop[i, j]$  to represent a loop nest or loop nested within the loops  $i$  and  $j$ . Figure 2 illustrates the tree representation of a loop nest.

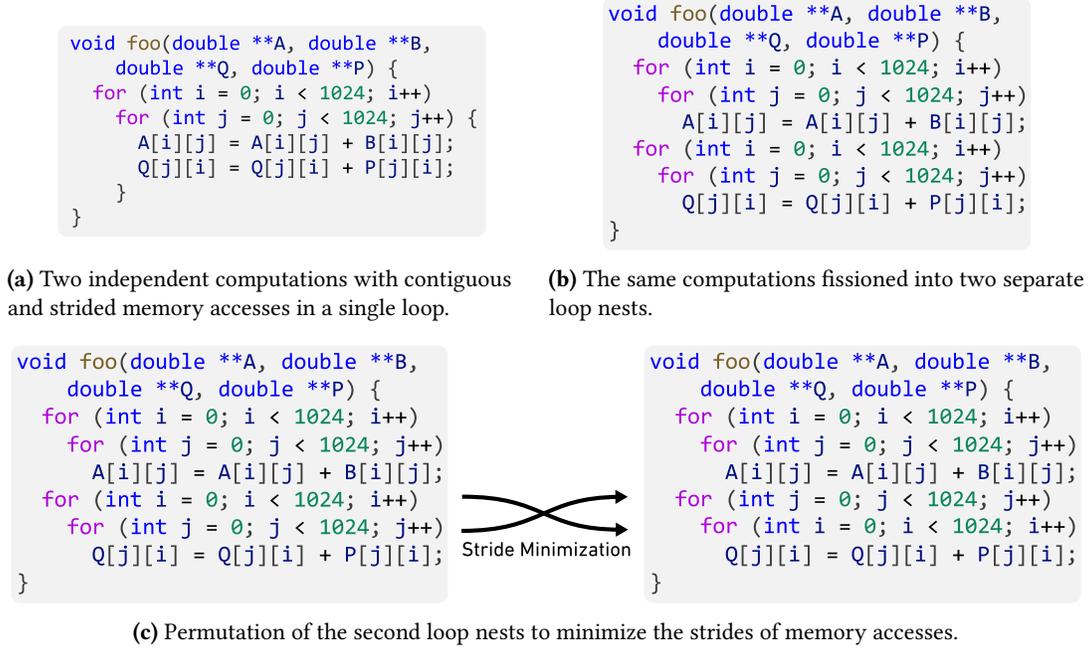


Figure 3. Loop nest code samples subject to normalization.

## 2.1 Maximal Loop Fission

Fusing computations into shared loops is a common technique to improve performance [25]. However, the combination of computations usually increases the complexity of memory accesses. An example, shown in Figure 3a, combines two computations with contiguous and strided memory accesses. Since the developer may apply such compositions manually when writing the code, we propose simplifying all loops to make them easier to analyze: we fission them as much as possible before the optimization

Let  $comp1[i_1, \dots, i_j, \dots, i_n]$  and  $comp2[i_1, \dots, i_j, \dots, i_n]$  be two computations within the same loop nest. If there are no data dependencies or loop-carried dependencies between  $comp1$  and  $comp2$ , we define a new loop nest with the same iterator, initial values, update criteria, and termination conditions  $i'_1 = i_1, \dots, i'_n = i_n$ . We then divide  $comp1$  and  $comp2$  across loop nests such that after fissioning we have  $comp1[i_1, \dots, i_j, \dots, i_n]$  and  $comp2[i'_1, \dots, i'_j, \dots, i'_n]$ . We repeat this for loops and computations nested at the same level of every loop nest until no such transformations are possible. The result is a sequence of "atomic" loop nests, as their loop bodies contain computations and loops that can not be separated due to data dependencies. An example of one instance of fissioning is illustrated in Figure 3a and Figure 3b, where the two computations are split into separate loop nests.

## 2.2 Stride Minimization

Depending on the order of loops within a loop nest, memory accesses have different strides, impacting the cache utilization. Since the optimal order depends on other optimizations

such as tiling or vectorization, loop permutation requires a non-trivial performance model and must be decided by an auto-scheduler [26]. To reduce the variations of loop nests, we propose stride minimization as a normalization criterion before optimization. We assume the stride minimization criterion is applied after the maximal loop fission criterion.

Let  $loop \rightarrow (i_1, \dots, i_j, i_k, \dots, i_n)$  be a loop nest with all iterators of nested loops ordered according to an in-order traversal. We define a generic optimization criterion,  $stride(loop)$ , which maps subsequent accesses to arrays within each computation of a loop nest  $loop$  to a real value. For instance, the sum of all distances between two subsequent accesses to all arrays over all computations is a suitable function.

Let  $\pi_1(loop) = (i_1, \dots, i_j, i_k, \dots, i_n)$  be a permutation and  $\pi_2(loop) = (i_1, \dots, i_k, i_j, \dots, i_n)$  be another legal permutation of the same loop nest. If  $stride(\pi_1) < stride(\pi_2)$ , then  $\pi_1$  is the permutation with the smaller stride. Generalizing, for each loop nest  $loop_{min}$ , we find and replace it with the legal permutation with a minimal stride  $\pi_{min}$ .

The complexity of finding  $loop_{min}$  depends on the definition of  $stride(loop)$ . Since our goal is to reduce the variation of loop nests for a downstream auto-scheduler, we argue that the minimum can simply be found by enumeration for many practically-relevant loop nests. For deep loop nests, we propose to sort groups of iterators as an approximation.

An example of stride minimization can be seen in Figure 3c, where the the sum of strides in minimized. If the dimensions are not statically known, other definitions of  $stride(loop)$  must be used, e.g., the number of out-of-order access w.r.t. the permutation of loop iterators and array dimensions.

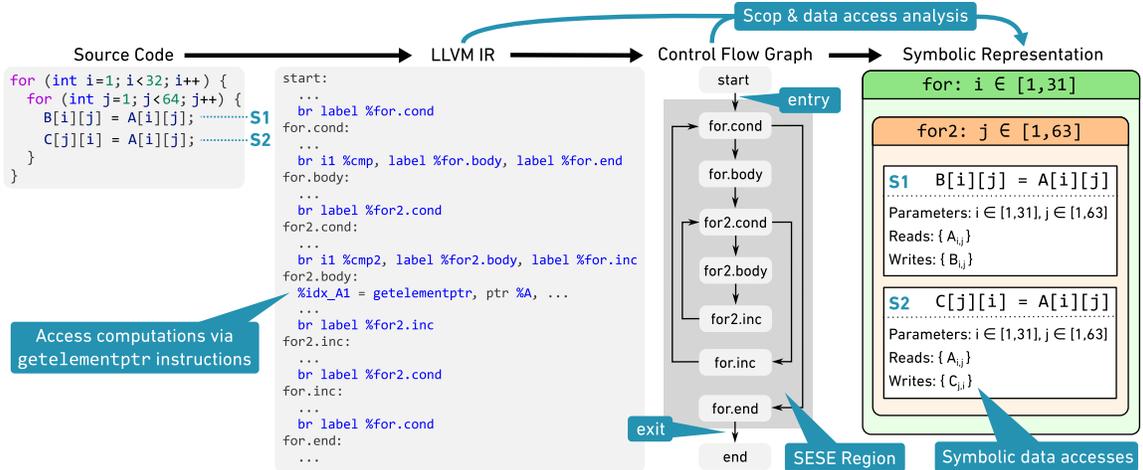


Figure 4. Lifting a symbolic representation of loop nests with high-level information from source code translated to LLVM IR.

### 3 Normalization on Intermediate Representations

We implement the normalization on LLVM IR to apply to as many codes as possible. In LLVM IR, loops and memory accesses are represented as instructions. Hence, all high-level information, such as array shapes, loop relations, and data dependencies, must be inferred through static analysis. For instance, to identify the strided memory access to array C in computation S2 of Figure 4, the accesses must first be derived from store and `getelementptr` and branch instructions. Therefore, we first lift a rich, symbolic representation of loop nests from LLVM IR to do the normalization. This lifting aims to represent loop nests in a hierarchy of loop and computation nodes, where loop iterators, domains, and data accesses are symbolic expressions. Figure 4 depicts the basic idea of this workflow.

#### 3.1 Lifting Symbolic Representations from LLVM IR

LLVM IR consists of instructions grouped into basic blocks. Basic blocks are connected through conditional and unconditional branches, typically represented in a *control-flow graph* (CFG). Polly implements several mechanisms to detect and lift a polyhedral representation of loop nests from LLVM IR. Since this simplifies large parts of the symbolic analysis, we use Polly as the basis of our lifting workflow. Based on Polly’s representation, we then generate an *Abstract Syntax Tree* (AST) of the loop nest using existing methods [17] implemented in the *integer set library* [35]. This AST consists of a tree of loops and computations nodes similar to our definition. To analyze strides and fissioning opportunities efficiently, we further augment the tree with dataflow information describing the subset of data produced and consumed by different nodes. We use the Stateful DataFlow multiGraph (SDFG) [5] and existing dataflow analysis [10].

To apply the changes to the original code, we use the property that the loop nests detected by Polly are maximal *single-entry-single-exit* (SESE) regions [21]. SESE regions are subgraphs of a CFG with unique incoming and outgoing edges. SESE regions can easily be removed from the original code and replaced by a function call to an external source code generated from an SDFG.

#### 3.2 Normalization Passes

We implement the two normalization criteria as two separate transformation passes in a pipeline based on the lifted representation of loop nests. An overview of the normalization pipeline is shown in Figure 5. In the first step, we fission the loop nests as maximally as possible. Since loop fissioning always splits loop nests into smaller loop nests of fewer computation nodes, we can apply transformations in a fixed-point pipeline until no more fissioning transformations apply. In the second step, we search for the loop permutation with minimal strides for each resulting loop nest of the first step. To find a minimal permutation, we enumerate all permutations and compute the strides from the symbolic expressions of memory accesses. It should be noted that although our representation is lifted from LLVM IR, the normalization can also be applied to SDFGs obtained from other sources.

## 4 The daisy scheduler

Ideally, normalization should make semantically equivalent implementations of an algorithm performance-equivalent. To test this hypothesis, we create a normalized auto-scheduler, *daisy*, and evaluate the impact of normalization on the auto-scheduler robustness. We create an A/B test comparing the original implementation of benchmarks (A variants) and alternative, semantically equivalent implementations (B variants).

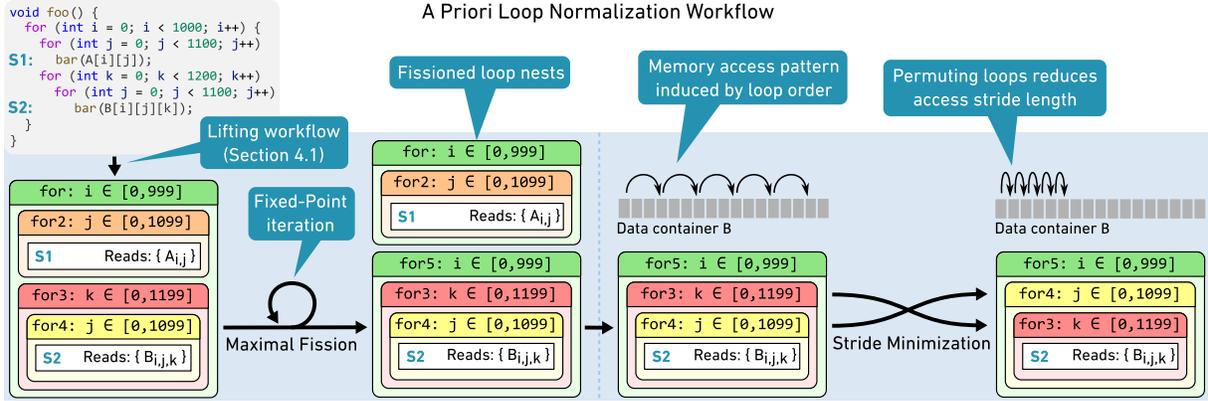


Figure 5. The normalization pipeline in two steps: Maximal loop fission and stride minimization.

**Optimization Algorithm.** We define a new auto scheduler, which applies our normalization passes and then queries optimizations from a database using *similarity-based transfer tuning* [33]. The stride minimization uses the sum of strides of all array accesses as the optimization criterion. The database consists of pairs of an embedding for the loop nest and transformation sequences including loop interchange, tiling, parallelization and vectorization. The database is seeded from normalized loop nests of the A variants and then applied to the normalized B variants. If a B loop nest is not reduced to an A loop nest, the transformation sequence cannot be applied.

**Seeding a Scheduling Database.** We collect all loop nests from the normalized A variants to define the auto-scheduler. For each loop nest corresponding to a BLAS-3 kernel, we add an optimization recipe to perform idiom detection, i.e., replacing the loop nest with the matching BLAS library call. The optimizations for other loop nests are found using an evolutionary search. In the first epoch of the search, the candidate optimizations for each loop nest are seeded using the Tiramisu auto-scheduler. This population is refined in three iterations through standard mutation and selection techniques, where the runtime determines the fitness. In the second and third epochs, the population is re-seeded using the current best optimization of the ten most similar loop nests and refined in three iterations again. The Euclidean distance of *performance embeddings* [33] determines the most similar loop nests.

**Benchmarks.** PolyBench [29] is a popular set of benchmarks for evaluating polyhedral compilers and auto-scheduling methods. Many of the implemented benchmarks offer several degrees of freedom, where the loop nests can be nested and permuted differently without changing the semantics of the algorithm. We have selected 15 parallelizable benchmarks where schedulers have a significant search space for optimization. To evaluate the auto-scheduler robustness, we

randomly generate an alternative B variant for each benchmark based on different permutations and compositions. In the following, we only consider the large input size.

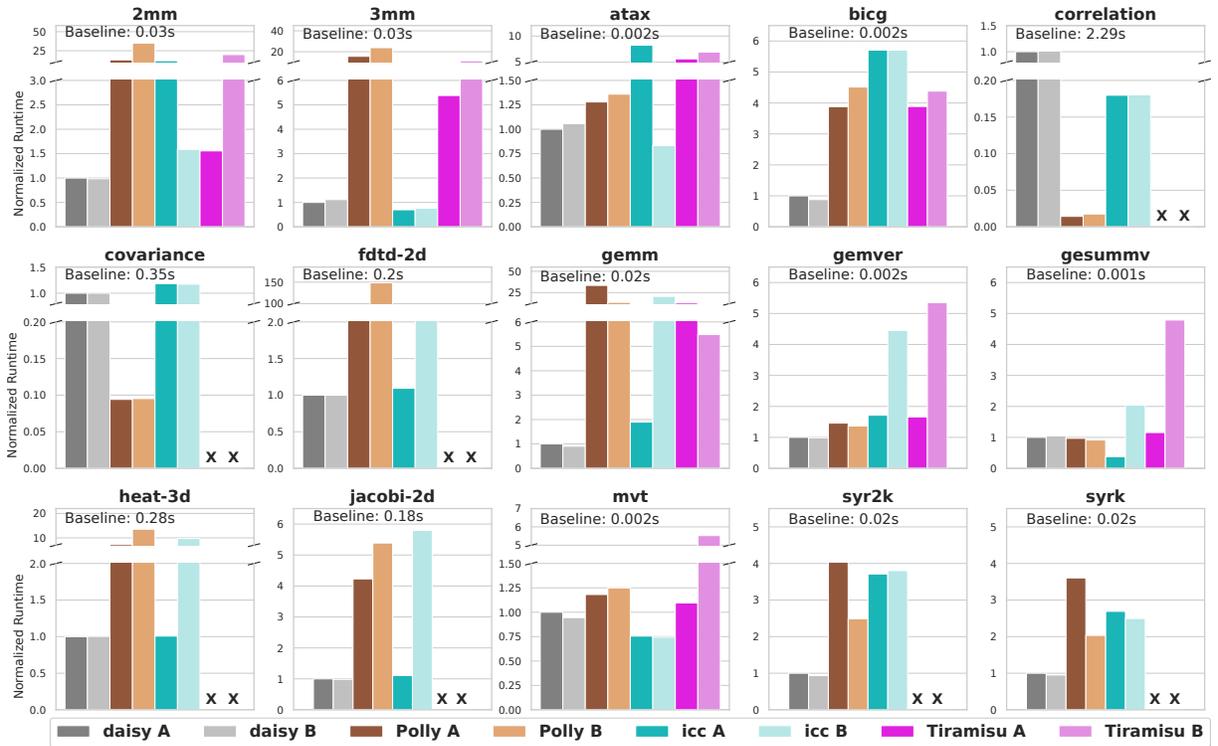
**Baselines.** We use Polly [18] based on LLVM 16.0.4 and with optimization flags `-O3 -polly -polly-parallel -polly-tiling -polly-vectorizer=stripmine -polly-2nd-level-tiling -gomp`. We configure the Tiramisu auto-scheduler [3] to run a Monte-Carlo Tree Search guided by the performance model. To account for the stochasticity of this search, we test the top three candidates and apply the best optimization among these. Building the original Tiramisu auto-scheduler for the Tiramisu DSL [4] using currently available software packages was unsuccessful. Therefore, we run the auto-scheduler as a standalone search and implement an adapter, which converts SDFGs to the JSON representation by the search. To simplify the conversion, we apply the maximal loop fission criterion as part of the adapter and restrict the conversion to perfectly nested parallel loops. Moreover, we compare the results to icc 2021.9.0, a general baseline with auto-parallelization `-parallel` and optimization `-O3` flags enabled.

**Experimental Setup.** The experiments are performed on an Intel Xeon E5-2680v3 clocked at 2.50 GHz with 64 GB of main memory. We measure according to a standard framework [20], where measurements are taken until the variance drops below five percent, and the resulting median is reported as the runtime.

#### 4.1 Normalized Auto-Scheduling: Same Semantics, Same Performance

We evaluate the runtime of Polly, Tiramisu, and icc against daisy for the two implementations - A and B - of each of the 15 benchmarks. The results are summarized in Figure 6.

**Robustness.** The first observation is that since the A and B variants are semantically equivalent, a robust auto-scheduler should achieve a runtime ratio close to one. This is true **for daisy**, where the largest difference between



**Figure 6.** Comparison of our model with state-of-the-art auto-scheduling methods and the icc compiler. The runtime is expressed relative to the runtime of the A variant of the benchmarks using *daisy*. Hence, a lower value is better. The implementation of the Tiramisu scheduler could not be applied to some of the benchmarks successfully. We mark those with X.

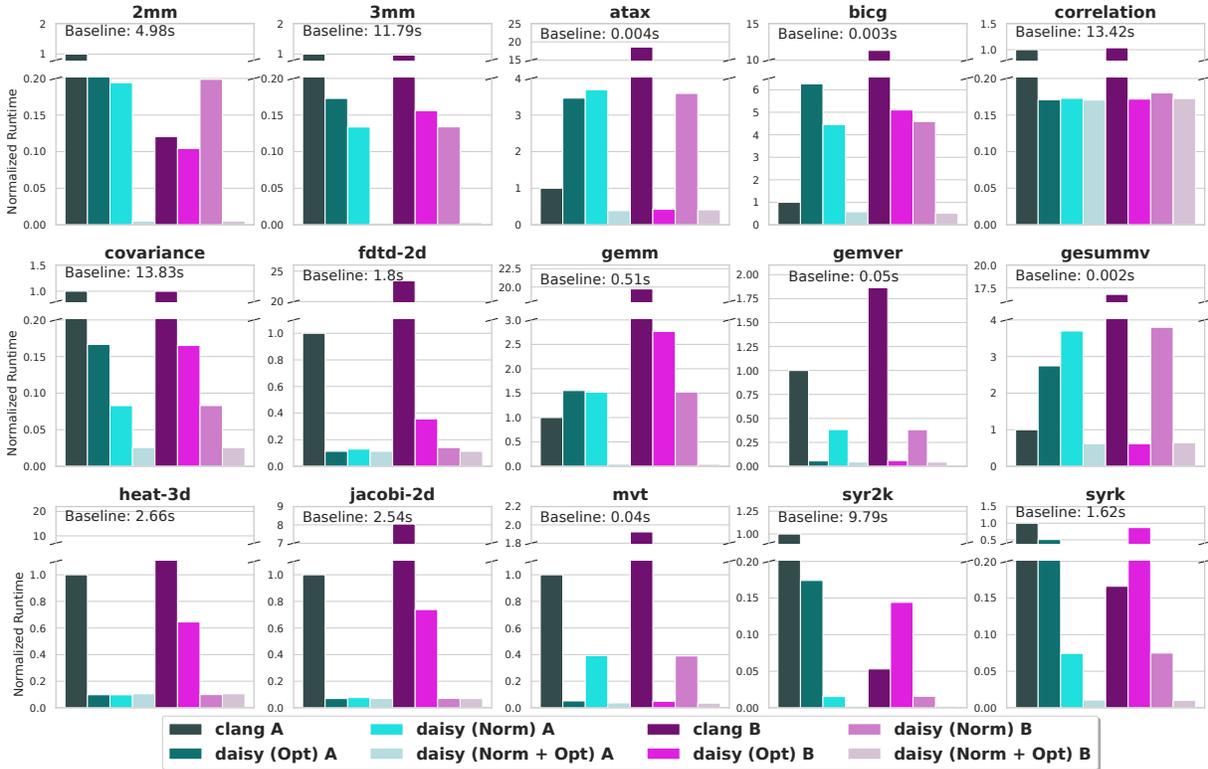
**the performance of the A and B implementations is 14% and the mean difference is just 5%.** However, all other approaches show significant variation between A and B implementations on several benchmarks, with differences of over an order of magnitude for applications such as *2mm* or *fdtd-2d*. For the latter, this can be explained by strided memory accesses in the B implementation that neither Polly nor icc can optimize well. Similarly, the performance of the Tiramisu auto-scheduler is susceptible to the specific structure of the loop nests of the A and B variants.

**Performance.** While achieving the same performance for the different implementations of the same benchmark is vital to prove the robustness of a scheduler, the performance must also be competitive with state-of-the-art approaches. Figure 6 shows the runtime of each benchmark and the baseline auto-schedulers relative to *daisy*. Our model achieves a geometric mean speedup of 2.31 over Polly, 2.89 over the Tiramisu auto-scheduler, and 1.58 over icc on the A variants. For the B variants, our auto-scheduler achieves a geometric mean speedup of 2.97 over Polly, 7.03 over the Tiramisu auto-scheduler, and 2.51 over icc. Our model underperforms compared to Polly on *correlation* and *covariance*. In those cases, our normalization passes fail to lift specific loop nests to the symbolic representations. As a result, the loop nest is

not optimized, and a reduction is executed in parallel, causing expensive atomic reductions in the C++ code. In general, however, *daisy* proves to be a sufficiently complex auto-scheduler. Most importantly, the runtime variations across A and B variants are in the order of measurement noise for our model. Hence, *daisy* optimizes the A and B variants of the benchmarks equally well using only optimizations derived from normalized A variants.

#### 4.2 Ablation Study: Same Optimizations, Different Performance

To analyze the impact of the normalization and the optimizations offered by transfer tuning in isolation, we now compare the results of compiling the benchmarks in the following scenarios: using only clang, using transfer tuning without normalization, using normalization without transfer tuning, and finally using the full pipeline in *daisy*. We do this for both the A and B versions of each benchmark. We note that the normal compiler optimizations -O3, etc. are applied in all configurations. Figure 7 summarizes the relative runtime of the benchmarks for optimization with and without prior normalization. The results show that **both normalization and optimization using the similarity-based transfer tuning algorithm are required to reach the best performance consistently.** Without the normalization step,



**Figure 7.** Comparison of clang and our model with and without normalization. The runtime is expressed relative to A variants of the benchmarks using clang. Hence, a lower value is better.

the database queried by the transfer tuning algorithm would need to explicitly enumerate all possible loop variations, which would not scale.

### 4.3 Auto-Scheduling beyond C: Different Language, Same Optimization

Applying auto-schedulers across programming languages is essential – and we wish to test daisy not just on different C implementations but also on implementations of the **same benchmarks in Python** – increasing the number of implementation variants considered. However, different programming languages have different syntactical features.

For instance, Figure 8 shows the implementations of the *symmetric rank-k update (SYRK)* kernel in PolyBench [29] and NPBench [36], a scientific benchmarking suite for high-performance NumPy [19]. For SYRK, the NPBench implementation uses ranges for indexing of NumPy arrays. When translating the Python benchmarks to our IR, such syntactical features may yield different representations for the same programs because of additional translation and optimization passes. To evaluate the effect of a priori loop nest normalization for auto-scheduling across programming languages, we apply the same database-based auto-scheduler from Section 4.1 to PolyBench benchmarks implemented in NPBench. In detail, we use the DaCe Python frontend [5] to obtain

an SDFG for the NPBench benchmark and then apply the normalization and auto-scheduler analogously. For reasons of comparability, we adapt the input sizes of the NPBench benchmarks to the large variants of PolyBench.

```
for (int i = 0; i < _PB_N; i++) {
    for (int j = 0; j <= i; j++)
        C[i][j] *= beta;
    for (int k = 0; k < _PB_M; k++)
        for (int j = 0; j < i; j++)
            C[i][j] += alpha * A[i][k] * B[j][k];
}
```

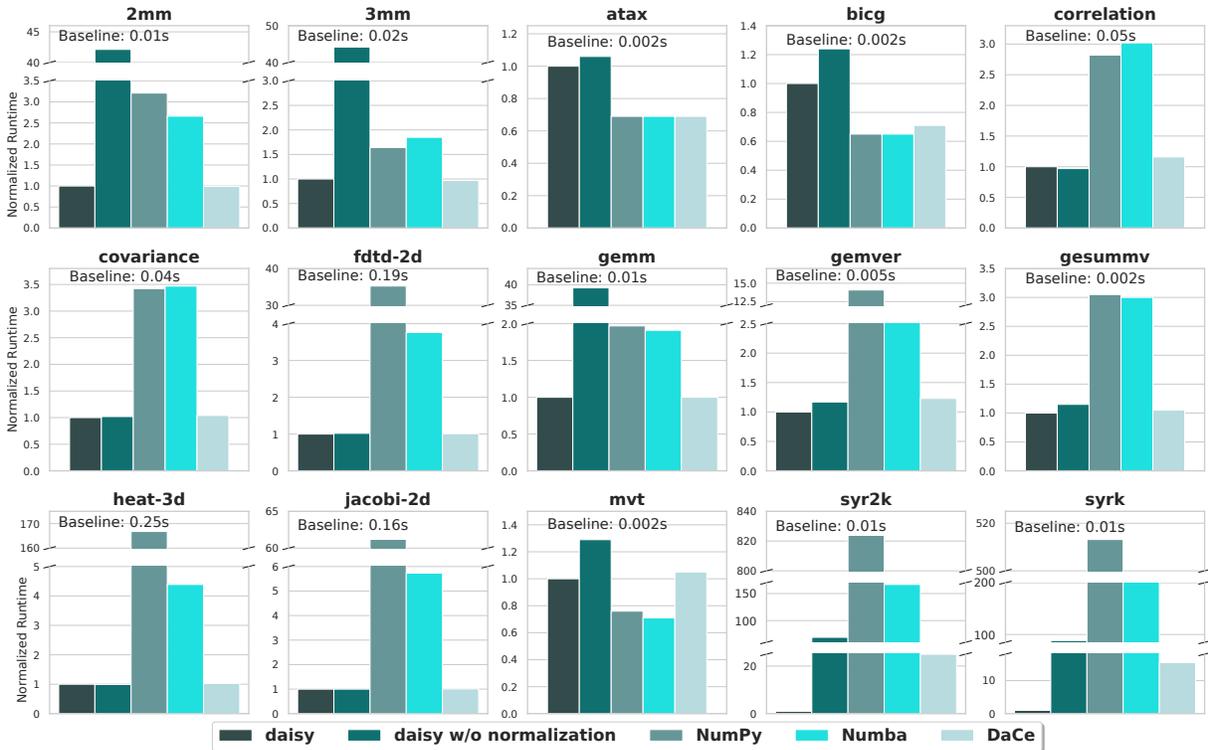
(a) PolyBench’s SYRK in C

```
for i in range(A.shape[0]):
    C[:,i+1] *= beta
    for k in range(A.shape[1]):
        C[:,i+1] += alpha * A[i,k] * A[:,i+1,k]
```

(b) NPBench’s SYRK using NumPy

**Figure 8.** The SYRK kernel implemented in C and NumPy.

**Baselines.** We consider three baselines: NumPy 1.25.2 [19], Numba 0.58.0 [23], and DaCe 0.14.2 [5]. All frameworks use custom operators to call optimized BLAS libraries for specific operations. Besides the operators, Numba and DaCe support additional optimizations. In detail, Numba is a just-in-time



**Figure 9.** Comparison of our model with NumPy-based frameworks implementing custom operators and optimizations for different applications. The runtime is expressed relative to the runtime of daisy . Hence, a lower value is better.

compiler that can automatically parallelize and vectorize loops if detected accordingly. DaCe generates SDFGs from a Python frontend, which can be optimized with various transformations - including, but not limited to the automatic parallelization and vectorization of loops.

**Results.** Figure 9 shows the runtime of the benchmarks for our approach and the improvements over the baseline frameworks. It also shows the results for daisy without prior normalization. Several benchmarks consist of BLAS kernels such as *gemm* and *gemv*, for which the different frameworks provide custom operators. Our model lifts BLAS-3 kernels to matching library calls and optimizes the remaining loop nests using optimizations found with the evolutionary search. The lifting of BLAS-3 kernels fails without normalization on several benchmarks, e.g., *2mm*, *3mm* and *gemm*. When applying normalization, our model mostly matches the performance of DaCe and outperforms NumPy and Numba. For the *syrk* and *syr2k* loop nests, our model outperforms all frameworks because the baseline frameworks do not provide custom operators here. Compared to unoptimized C code, the native Python code means a significant loss in performance in these cases. Furthermore, there are benchmarks for which auto-parallelization is necessary to achieve speedups, e.g., *heat-3d* and *jacobi-2d*. However, the structure of the loops, as implemented by the developer, does not comprise much

potential for further optimization. Note that the correlation and covariance benchmarks do not show the problems of Section 4.1 due to a different structure of the SDFGs from the Python frontend. The performance of our model is thus close to the performance of DaCe, outperforming DaCe in cases where no custom operators are available.

**In summary, a priori normalization enables the application of an auto-scheduler derived from specific variants of C loop nests to loop nests translated from Python programs.**

## 5 CLOUDSC: Case Study of Normalization and Optimization

We now expand our analysis to CLOUDSC, a parametrization scheme for simulating clouds and precipitation. The model is part of the Integrated Forecasting System (IFS) and is employed in production by ECMWF for weather forecasts and climate analysis. It is a nonlinear scheme that accounts for approximately 10% of the IFS forecast model. Initially written in Fortran, the code has been the focus of an effort to understand performance portability with versions now existing in C, as well as using OpenACC and CUDA. We use an SDFG generated by the DaCe framework and apply our normalization pipeline implemented in *daisy*.

```

! Function Definitions
REAL :: FOEWM, FOELDCPM
FOEWM ( PTARE ) = RZES * &
&(MIN(1.0, ((MAX(RTICE, MIN(RTWAT, PTARE)) - RTICE) * RTWAT_RTICE_R ** 2) &
& * EXP(R3LES * (PTARE - RTT) / (PTARE - R4IES)) &
& + (1.0 - MIN(1.0, ((MAX(RTICE, MIN(RTWAT, PTARE)) - RTICE) * RTWAT_RTICE_R ** 2) &
& * EXP(R3IES * (PTARE - RTT) / (PTARE - R4IES))))
FOELDCPM ( RTWAT_RTICE_R ) = MIN(1.0, ((MAX(RTICE, MIN(RTWAT, PTARE)) - &
& RTICE) * RTWAT_RTICE_R ** 2) * RALVDCP + &
&(1.0 - MIN(1.0, ((MAX(RTICE, MIN(RTWAT, PTARE)) - RTICE) * RTWAT_RTICE_R ** 2) &
& * RALSDCP

! Vertical loop
DO JK=NCLDTP, KLEV
...
! Loop Nest
DO JL=KIDIA, KFDIA
ZQP = 1.0/PAP(JL, JK)
ZQSAT = FOEWM(ZTP1(JL, JK)) * ZQP
ZQSAT = MIN(0.5, ZQSAT)
ZCOR = 1.0/(1.0-RETV * ZQSAT)
ZQSAT = ZQSAT*ZCOR
ZCOND = (ZQSMIX(JL, JK) - ZQSAT) / (1.0 + ZQSAT*ZCOR*FOEDEM(ZTP1(JL, JK)))
ZTP1(JL, JK) = ZTP1(JL, JK) + FOELDCPM(ZTP1(JL, JK)) * ZCOND
ZQSMIX(JL, JK) = ZQSMIX(JL, JK) - ZCOND
ZQSAT = FOEWM(ZTP1(JL, JK)) * ZQP
ZQSAT = MIN(0.5, ZQSAT)
ZCOR = 1.0/(1.0-RETV * ZQSAT)
ZQSAT = ZQSAT*ZCOR
ZCOND1 = (ZQSMIX(JL, JK) - ZQSAT) / (1.0 + ZQSAT*ZCOR*FOEDEM(ZTP1(JL, JK)))
ZTP1(JL, JK) = ZTP1(JL, JK) + FOELDCPM(ZTP1(JL, JK)) * ZCOND1
ZQSMIX(JL, JK) = ZQSMIX(JL, JK) - ZCOND1
...
ENDDO
...
ENDDO

```

(a) A part of the simulation of the erosion of clouds. Functions definitions are provided, and the outer vertical loop is shown.

```

...
! Vertical loop
DO JK=NCLDTP, KLEV
...
! Loop Nest
DO JL=KIDIA, KFDIA
ZQP_0(JL) = 1.0/PAP(JL, JK)
ENDDO
DO JL=KIDIA, KFDIA
ZQSAT = FOEWM(ZTP1(JL, JK)) * ZQP_0(JL)
ZQSAT = MIN(0.5, ZQSAT)
ZCOR = 1.0/(1.0-RETV * ZQSAT)
ZQSAT = ZQSAT*ZCOR
ZCOND_0(JL) = (ZQSMIX(JL, JK) - ZQSAT) / (1.0 + ZQSAT*ZCOR*FOEDEM(ZTP1(JL, JK)))
ENDDO
DO JL=KIDIA, KFDIA
ZTP1(JL, JK) = ZTP1(JL, JK) + FOELDCPM(ZTP1(JL, JK)) * ZCOND_0(JL)
ENDDO
DO JL=KIDIA, KFDIA
ZQSMIX(JL, JK) = ZQSMIX(JL, JK) - ZCOND_0(JL)
ENDDO
DO JL=KIDIA, KFDIA
ZQSAT = FOEWM(ZTP1(JL, JK)) * ZQP_0(JL)
ZQSAT = MIN(0.5, ZQSAT)
ZCOR = 1.0/(1.0-RETV * ZQSAT)
ZQSAT = ZQSAT_4(JL) * ZCOR
ZCOND1_0(JL) = (ZQSMIX(JL, JK) - ZQSAT) / (1.0 + ZQSAT*ZCOR*FOEDEM(ZTP1(JL, JK)))
ENDDO
DO JL=KIDIA, KFDIA
ZTP1(JL, JK) = ZTP1(JL, JK) + FOELDCPM(ZTP1(JL, JK)) * ZCOND1_0(JL)
ENDDO
DO JL=KIDIA, KFDIA
ZQSMIX(JL, JK) = ZQSMIX(JL, JK) - ZCOND1_0(JL)
...
ENDDO

```

(b) A part of the simulation of the erosion of clouds fissioned into individual computations and fused by one-to-one producer-consumer loop nest relations.

**Figure 10.** A loop nest taken from from the vertical loop of CLOUDSC before and after normalization and fusion.

An important characteristic of the code is the way it accesses data. The simulated volume is divided into vertical columns, each computed independently. When iterating vertically through a column, the physical properties stored in multiple large arrays are updated. The update during each step of the vertical loop comprises several nested loops, each implementing distinct physical equations. These innermost loops iterate over a "tiling" parameter — NPROMA, simultaneously updating multiple independent columns. Hence, this innermost tiling parameter divides the total number of columns between the outermost loop NBLOCKS and the innermost loops NPROMA, i.e.,  $\text{num\_columns} = \text{NBLOCKS} * \text{NPROMA}$ . Since both loops are fully data parallel, users can divide the total problem size into NPROMA and NBLOCKS to find the optimum balance between parallelism and data locality for their hardware system.

### 5.1 Discovering Performance Optimizations in the Erosion of Clouds

We first analyze a single physical update step inside the vertical loop. Figure 10a depicts a part of the simulation of cloud erosion. Since the computation is fully data-oblivious, we can investigate its performance independent of other computations inside the model.

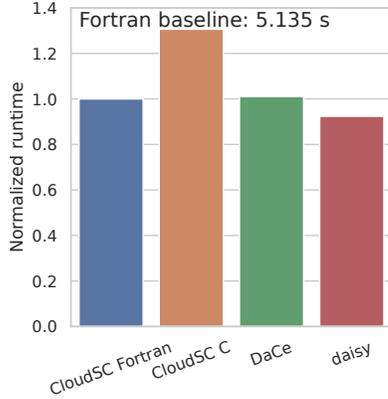
The chosen loop nest is representative of many innermost loops within the vertical loop: It updates two arrays, ZTP1 and ZQSMIX, over the NPROMA dimension with an iterator JL in a fully data-parallel manner. For this, it computes several intermediate scalars, which can be assumed to be

live only for the loop's scope. By default, CLOUDSC is compiled with loop unrolling and function inlining to maximize instruction-level parallelism. Therefore, the computations of the sub-routines FOEWM and FOELDCPM are inlined, and the loop body is unrolled. Hence, the loop body is significantly larger than the source code suggests, potentially hindering crucial compiler optimizations such as register allocations.

We apply maximal loop fission to divide individual computations into smaller loops. This reverts the original decision by the developers to group the computations according to the physical equation. This enables the application of a typical optimization recipe, which iteratively fuses all one-to-one producer-consumer relations between loop nests. Hence, intermediate results are computed for the whole dimension of JL and stored in the local arrays ZQP\_0 and ZCOND\_0. Each loop nest only contains scalars, which are used within a short distance of instructions. The resulting loop nests are shown in Figure 10b.

	Original	Optimized
Single Iteration[ms]	0.040	0.006
KLEV Iterations[ms]	5.468	0.882
L1 Loads	2632	1281
L1 Evicts	963	178

**Table 1.** The table shows the runtime for a single iteration and KLEV iterations of the loop nests. It further shows the absolute number of loads and evicts on the L1 cache.

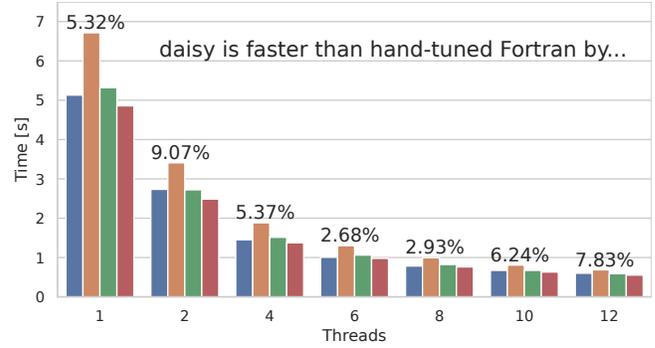


**Figure 11.** CLOUDSC runtime for sequential execution of the Fortran, C, DaCe, and daisy versions from left to right. The runtime is normalized by the Fortran version. Hence, a lower value is better.

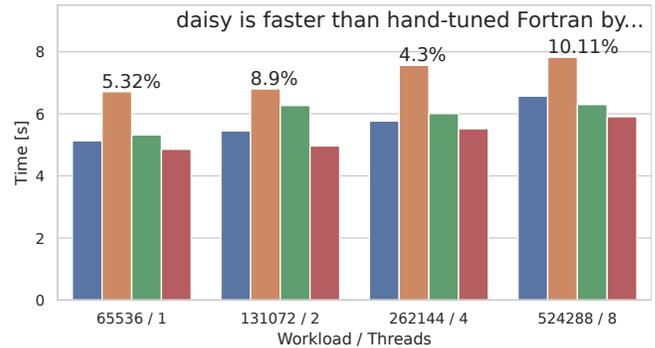
**Results.** We measure the performance of the two versions for a single iteration and KLEV iterations, corresponding to the size of the vertical loop. We set  $NPROMA=128$ , as this value offers the best results on our hardware. The experimental setup is analogous to Chapter 4. Furthermore, we measure load and evicts to and from the L1 cache to analyze the improvements of our optimization. Table 1 summarizes both runtime and memory behavior, showing that our optimization yields a speedup of 4 $\times$ . Furthermore, the optimization reduces the pressure on the L1 cache, as there are significantly fewer transfers between L1 and L2. Hence, the normalization allows us to discover new applications of well-known performance optimizations as it transforms the physical equation into a canonical form.

## 5.2 Evaluation

We now apply the same pipeline to the full model and compare the runtime of daisy against the Fortran, C, and DaCe versions. Figure 11 shows the measurements for sequential execution. We keep  $NPROMA = 128$  and set  $NBLOCKS = 512$ . The optimizations applied by daisy yield a speedup of 1.08 $\times$  compared to the second-best version, which is the highly-tuned Fortran code. As shown previously, optimizing the loop nest in the erosion of clouds saves approximately four milliseconds per execution of the vertical loop, accumulating to about 200 milliseconds for the full model. Hence, this optimization primarily explains the performance improvements. Furthermore, we compare the different versions' strong and weak scaling behavior in Figure 12a and Figure 12b. With an optimization of the application's critical path, the performance improvements also translate to the parallel execution. We also measure the FLOP/s for Fortran and daisy and compare it to the peak FLOP/s of the machine. The peak FLOP/s of the machine is measured with 52522.83 MFLOP/s based on a benchmark optimized for



**(a)** Strong scaling behavior of CLOUDSC for the Fortran, C, DaCe, and daisy versions from left to right and grouped by the number of threads.



**(b)** Weak scaling behavior of CLOUDSC for the Fortran, C, DaCe, and daisy versions.

**Figure 12.** Experimental results for the CLOUDSC case study.

Fused-multiply-add (FMA) and AVX instructions. The Fortran version yields 13634.03 MFLOP/s, and the daisy version yields 14792.81 MFLOP/s, which is 25.96% and 28.16% of peak performance, respectively. This underscores that CLOUDSC is already a highly tuned application, so finding optimization opportunities is difficult. Hence, daisy improved the performance of a critical loop nest in a highly-tuned application.

## 6 Discussion

The following sections discuss the implications and limitations of our work.

**Normalization Criteria.** The idea of our normalization is the simplification of memory accesses. This is motivated by the observation that optimization recipes usually apply to a particular memory access pattern, but large applications group computations in loop nests according to formulas. We derived the normalization criteria from two common ways to change the memory accesses of a loop nest: by permutation and composition. Our findings open a research avenue in exploring normalization criteria and understanding their impact on optimization pipeline performance.

**Complexity of Auto-scheduling.** Auto-scheduling is typically formulated as a search over a humongous space of possible combinations of transformations [2, 3, 32]. This unconstrained formulation requires careful design and training of performance models and search methods. A priori loop nest normalization separates the search and input space and maps semantically equivalent loop nests to a single problem instance. This reduces the necessary model complexity without a loss of generality of the optimization for the unconstrained search space.

## 7 Related Work

Loop transformations and normalization have been studied for decades. In particular, Chelini et al. [11] evaluate stride minimization as an optimization criterion for loop scheduling, and Callahan [9] discusses loop distribution as a technique to simplify the analysis of parallel loops. This paper, in turn, demonstrates the application of such techniques for the design of robust auto-schedulers. The following briefly overviews the related work on auto-scheduling and loop normalization.

**Optimization Criteria.** Auto-scheduling approaches define loop scheduling as an optimization problem over different types of objective functions. In the domain of polyhedral optimization, the minimization of the *maximal dependence distance* [1, 7, 24] is a popular objective function, which is also implemented in *Pluto* [8] and *Polly* [18]. While these approaches usually guarantee global optima for the solutions of the scheduling problems, the modeled schedule space and objective function may fail to capture the complex features of modern hardware [4]. Recent works [11, 22, 27] therefore combine multiple specialized criteria such as *Stride Optimization* and *Dependence Guided Fusion* into a multi-objective optimization. Another research direction learns the objective function from data using deep learning [2, 3, 13, 31, 32]. All these approaches seek to optimize more complex objective functions in larger schedule spaces at the cost of a global optimum. This paper proposes normalization as a pre-processing step based on criteria that resemble manually derived objective functions. Normalization is primarily applicable to models based on local optimization to reduce the variation of states.

**Idiom Detection.** Idiom detection seeks to detect and replace specific idioms with optimized implementations. Since the detection is prone to variations, loop normalization techniques have been analyzed in this context early on. Pinter and Pinter [28] define a loop normal-form for auto-parallelization based on idiom detection, which requires that each loop-carried dependency is between two loop iterations. The authors propose *loop unrolling* to ensure this property. Callahan [9] analyzes complex bounded recurrences to recognize parallelism in loops. The author discusses loop distribution as a

technique to simplify this analysis. Several approaches seek to detect idioms on LLVM IR [12, 15, 16]. *Declarative Loop Tactics* [12] uses Polly to compute the Scops of the code and detects idioms based on the polyhedral representation. This approach enforces a description of memory accesses by affine function, which is more of a constraint than a normalization. *LiLAC* [16] and *KernelFaRer* [15] require the standard compiler optimizations to be performed before detection. In contrast, we propose performing data-centric normalizations beyond the standard optimizations of the compiler, which may significantly change the memory access scheme.

## 8 Conclusion

In this paper, we present a priori loop nest normalization, simplifying the auto-scheduling of loop nests in complex applications. The approach maps loop nests with different memory access patterns to the same canonical form, significantly reducing the variety of loop nests to be optimized.

We demonstrate the approach in different case studies, highlighting the improved robustness and higher applicability of optimizations. The approach outperforms state-of-the-art compilers, auto-schedulers, and performance-oriented frameworks in C, Python, and Fortran by significant factors. In particular, we apply the approach to a highly-tuned scientific simulation, where it identifies additional optimizations resulting in a 10% speedup.

The approach enables the application of state-of-the-art auto-schedulers to large scientific code bases, bringing mathematical formulas into a form more amenable to optimization.

## Acknowledgments

This project has received funding from the European Research Council (ERC) under the European Union’s Horizon 2020 program (grant agreement PSAP, No. 101002047), from the European High-Performance Computing Joint Undertaking (JU) under grant agreement EUPilot No 101034126 and by the ETH Future Computing Laboratory (EFCL), financed by a donation from Huawei Technologies. L.T. wishes to thank D.K. for many valuable discussions.

## References

- [1] Aravind Acharya and Uday Bondhugula. 2015. PLUTO+: Near-Complete Modeling of Affine Transformations for Parallelism and Locality. *SIGPLAN Not.* 50, 8 (jan 2015), 54–64. <https://doi.org/10.1145/2858788.2688512>
- [2] Andrew Adams, Karima Ma, Luke Anderson, Riyadh Baghdadi, Tzu-Mao Li, Michaël Gharbi, Benoit Steiner, Steven Johnson, Kayvon Fatahalian, Frédo Durand, and Jonathan Ragan-Kelley. 2019. Learning to Optimize Halide with Tree Search and Random Programs. *ACM Trans. Graph.* 38, 4, Article 121 (jul 2019), 12 pages. <https://doi.org/10.1145/3306346.3322967>
- [3] Riyadh Baghdadi, Massinissa Merouani, Mohamed-Hicham Leghettas, Kamel Abdous, Taha Arbaoui, Karima Benatchba, and Saman P. Amarasinghe. 2021. A Deep Learning Based Cost Model for Automatic

- Code Optimization. In *Proceedings of Machine Learning and Systems 2021, MLSys 2021, virtual, April 5-9, 2021*, Alex Smola, Alex Dimakis, and Ion Stoica (Eds.). mlsys.org. <https://proceedings.mlsys.org/paper/2021/hash/3def184ad8f4755ff269862ea77393dd-Abstract.html>
- [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* (Washington, DC, USA) (CGO 2019). IEEE Press, 193–205.
  - [5] Tal Ben-Nun, Johannes de Fine Licht, Alexandros N. Ziogas, Timo Schneider, and Torsten Hoefler. 2019. Stateful Dataflow Multigraphs: A Data-Centric Model for Performance Portability on Heterogeneous Architectures. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis* (Denver, Colorado) (SC '19). Association for Computing Machinery, New York, NY, USA, Article 81, 14 pages. <https://doi.org/10.1145/3295500.3356173>
  - [6] Kristof Beyls and Erik H. D'Hollander. 2001. Reuse Distance as a Metric for Cache Behavior. In *In Proceedings of the IASTED Conference on Parallel and Distributed Computing and Systems*. 617–662.
  - [7] Uday Bondhugula, Muthu Baskaran, Sriram Krishnamoorthy, J. Ramanujam, Atanas Rountev, and P. Sadayappan. 2008. Automatic Transformations for Communication-Minimized Parallelization and Locality Optimization in the Polyhedral Model. In *Compiler Construction*, Laurie Hendren (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 132–146.
  - [8] Uday Bondhugula, Albert Hartono, J. Ramanujam, and P. Sadayappan. 2008. A Practical Automatic Polyhedral Parallelizer and Locality Optimizer. *SIGPLAN Not.* 43, 6 (jun 2008), 101–113. <https://doi.org/10.1145/1379022.1375595>
  - [9] D. Callahan. 1992. Recognizing and parallelizing bounded recurrences. In *Languages and Compilers for Parallel Computing*, Utpal Banerjee, David Gelernter, Alex Nicolau, and David Padua (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 169–185.
  - [10] Alexandru Calotoiu, Tal Ben-Nun, Grzegorz Kwasniewski, Johannes de Fine Licht, Timo Schneider, Philipp Schaad, and Torsten Hoefler. 2022. Lifting C Semantics for Dataflow Optimization. In *Proceedings of the 36th ACM International Conference on Supercomputing* (Virtual Event) (ICS '22). Association for Computing Machinery, New York, NY, USA, Article 17, 13 pages. <https://doi.org/10.1145/3524059.3532389>
  - [11] Lorenzo Chelini, Tobias Gysi, Tobias Grosser, Martin Kong, and Henk Corporaal. 2020. Automatic Generation of Multi-Objective Polyhedral Compiler Transformations. In *Proceedings of the ACM International Conference on Parallel Architectures and Compilation Techniques* (Virtual Event, GA, USA) (PACT '20). Association for Computing Machinery, New York, NY, USA, 83–96. <https://doi.org/10.1145/3410463.3414635>
  - [12] Lorenzo Chelini, Oleksandr Zinenko, Tobias Grosser, and Henk Corporaal. 2019. Declarative Loop Tactics for Domain-Specific Optimization. *ACM Trans. Archit. Code Optim.* 16, 4, Article 55 (dec 2019), 25 pages. <https://doi.org/10.1145/3372266>
  - [13] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Yan, Meghan Cowan, Haichen Shen, Leyuan Wang, Yuwei Hu, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. 2018. TVM: An Automated End-to-End Optimizing Compiler for Deep Learning. In *Proceedings of the 13th USENIX Conference on Operating Systems Design and Implementation* (Carlsbad, CA, USA) (OSDI'18). USENIX Association, USA, 579–594.
  - [14] Edward Grady Coffman and Peter J Denning. 1973. *Operating systems theory*. Vol. 973. prentice-Hall Englewood Cliffs, NJ.
  - [15] João P. L. De Carvalho, Braedy Kuzma, Ivan Korostelev, José Nelson Amaral, Christopher Barton, José Moreira, and Guido Araujo. 2021. KernelFaRer: Replacing Native-Code Idioms with High-Performance Library Calls. *ACM Trans. Archit. Code Optim.* 18, 3, Article 38 (jun 2021), 22 pages. <https://doi.org/10.1145/3459010>
  - [16] Philip Ginsbach, Bruce Collie, and Michael F. P. O'Boyle. 2020. Automatically Harnessing Sparse Acceleration. In *Proceedings of the 29th International Conference on Compiler Construction* (San Diego, CA, USA) (CC 2020). Association for Computing Machinery, New York, NY, USA, 179–190. <https://doi.org/10.1145/3377555.3377893>
  - [17] Tobias Grosser, Sven Verdoolaege, and Albert Cohen. 2015. Polyhedral AST Generation Is More Than Scanning Polyhedra. *ACM Trans. Program. Lang. Syst.* 37, 4, Article 12 (jul 2015), 50 pages. <https://doi.org/10.1145/2743016>
  - [18] Tobias Grosser, Hongbin Zheng, Raghesh Aloor, Andreas Simbürger, Armin Größlinger, and Louis-Noël Pouchet. 2011. Polly - Polyhedral optimization in LLVM. <http://impact2011.inrialpes.fr/en/index.html> First International Workshop on Polyhedral Compilation Techniques, IPACT 2011 ; Conference date: 03-04-2011 Through 03-04-2011.
  - [19] Charles R. Harris, K. Jarrod Millman, St'efan J. van der Walt, Raf Gommers, Pauli Virtanen, David Cournapeau, Eric Wieser, Julian Taylor, Sebastian Berg, Nathaniel J. Smith, Robert Kern, Matti Picus, Stephan Hoyer, Marten H. van Kerkwijk, Matthew Brett, Allan Haldane, Jaime Fernández del Río, Mark Wiebe, Pearu Peterson, Pierre G'erard-Marchant, Kevin Sheppard, Tyler Reddy, Warren Weckesser, Hameer Abbasi, Christoph Gohlke, and Travis E. Oliphant. 2020. Array programming with NumPy. *Nature* 585, 7825 (Sept. 2020), 357–362. <https://doi.org/10.1038/s41586-020-2649-2>
  - [20] Torsten Hoefler and Roberto Belli. 2015. Scientific Benchmarking of Parallel Computing Systems: Twelve Ways to Tell the Masses When Reporting Performance Results. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis* (Austin, Texas) (SC '15). Association for Computing Machinery, New York, NY, USA, Article 73, 12 pages. <https://doi.org/10.1145/2807591.2807644>
  - [21] Richard Johnson, David Pearson, and Keshav Pingali. 1994. The Program Structure Tree: Computing Control Regions in Linear Time. *SIGPLAN Not.* 29, 6 (jun 1994), 171–185. <https://doi.org/10.1145/773473.178258>
  - [22] 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* (Phoenix, AZ, USA) (PLDI 2019). Association for Computing Machinery, New York, NY, USA, 469–484. <https://doi.org/10.1145/3314221.3314653>
  - [23] Siu Kwan Lam, Antoine Pitrou, and Stanley Seibert. 2015. Numba: A LLVM-Based Python JIT Compiler. In *Proceedings of the Second Workshop on the LLVM Compiler Infrastructure in HPC* (Austin, Texas) (LLVM '15). Association for Computing Machinery, New York, NY, USA, Article 7, 6 pages. <https://doi.org/10.1145/2833157.2833162>
  - [24] 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* (Rhodes, Greece) (ICS '99). Association for Computing Machinery, New York, NY, USA, 228–237. <https://doi.org/10.1145/305138.305197>
  - [25] 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) (PPoPP '14). <http://www-users.cs.umn.edu/~sanyam/publications/p233-mehta.pdf>
  - [26] Lina Mezdour, Khadidja Kadem, Massinissa Merouani, Amina Selma Haichour, Saman Amarasinghe, and Riyadh Baghdadi. 2023. A Deep Learning Model for Loop Interchange. In *Proceedings of the 32nd ACM SIGPLAN International Conference on Compiler Construction* (Montréal, QC, Canada) (CC 2023). Association for Computing Machinery, New York, NY, USA, 50–60. <https://doi.org/10.1145/3578360.3580257>

- [27] Julian Miller, Lukas Trümper, Christian Terboven, and Matthias S. Müller. 2021. A Theoretical Model for Global Optimization of Parallel Algorithms. *Mathematics* 9, 14 (2021). <https://doi.org/10.3390/math9141685>
- [28] Shlomit S. Pinter and Ron Y. Pinter. 1994. Program Optimization and Parallelization Using Idioms. *ACM Trans. Program. Lang. Syst.* 16, 3 (may 1994), 305–327. <https://doi.org/10.1145/177492.177494>
- [29] Louis-Noël Pouchet and Tomofumi Yuki. 2017. PolyBench: The polyhedral benchmark suite (version 4.2).
- [30] Philipp Schaad, Tal Ben-Nun, and Torsten Hoefler. 2022. Boosting Performance Optimization with Interactive Data Movement Visualization. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis (Dallas, Texas) (SC '22)*. IEEE Press, Article 64, 16 pages.
- [31] Shikhar Singh, James Hegarty, Hugh Leather, and Benoit Steiner. 2022. A Graph Neural Network-Based Performance Model for Deep Learning Applications. In *Proceedings of the 6th ACM SIGPLAN International Symposium on Machine Programming (San Diego, CA, USA) (MAPS 2022)*. Association for Computing Machinery, New York, NY, USA, 11–20. <https://doi.org/10.1145/3520312.3534863>
- [32] Benoit Steiner, Chris Cummins, Horace He, and Hugh Leather. 2021. Value Learning for Throughput Optimization of Deep Learning Workloads. In *Proceedings of Machine Learning and Systems*, A. Smola, A. Dimakis, and I. Stoica (Eds.), Vol. 3. 323–334. <https://proceedings.mlsys.org/paper/2021/file/73278a4a86960eeb576a8fd4c9ec6997-Paper.pdf>
- [33] Lukas Trümper, Tal Ben-Nun, Philipp Schaad, Alexandru Calotoiu, and Torsten Hoefler. 2023. Performance Embeddings: A Similarity-Based Transfer Tuning Approach to Performance Optimization. In *Proceedings of the 37th International Conference on Supercomputing (Orlando, FL, USA) (ICS '23)*. Association for Computing Machinery, New York, NY, USA, 50–62. <https://doi.org/10.1145/3577193.3593714>
- [34] Didem Unat, Anshu Dubey, Torsten Hoefler, John Shalf, Mark Abraham, Mauro Bianco, Bradford L. Chamberlain, Romain Cledat, H. Carter Edwards, Hal Finkel, Karl Fuerlinger, Frank Hannig, Emmanuel Jeannot, Amir Kamil, Jeff Keasler, Paul H J Kelly, Vitus Leung, Hatem Ltaief, Naoya Maruyama, Chris J. Newburn, and Miquel Pericás. 2017. Trends in Data Locality Abstractions for HPC Systems. *IEEE Transactions on Parallel and Distributed Systems* 28, 10 (2017), 3007–3020. <https://doi.org/10.1109/TPDS.2017.2703149>
- [35] Sven Verdoolaege. 2010. isl: An Integer Set Library for the Polyhedral Model. In *Mathematical Software – ICMS 2010*, Komei Fukuda, Joris van der Hoeven, Michael Joswig, and Nobuki Takayama (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 299–302.
- [36] Alexandros Nikolaos Ziogas, Tal Ben-Nun, Timo Schneider, and Torsten Hoefler. 2021. NPbench: A Benchmarking Suite for High-Performance NumPy. In *Proceedings of the ACM International Conference on Supercomputing (Virtual Event, USA) (ICS '21)*. Association for Computing Machinery, New York, NY, USA, 63–74. <https://doi.org/10.1145/3447818.3460360>

Received 2024-09-11; accepted 2024-11-04