Design and Use of Loop-Transformation Pragmas

Michael Kruse[(✉)] and Hal Finkel[(✉)]

Argonne Leadership Computing Facility, Argonne National Laboratory,
Lemont, IL 60439, USA
{mkruse,hfinkel}@anl.gov

Abstract. Adding a pragma directive into the source code is undoubtedly easier than rewriting it, for instance for loop unrolling. Moreover, if the application is maintained for multiple platforms, their difference in performance characteristics may require different code transformations. Code transformation directives allow replacing the directives depending on the platform, i.e. separation of code semantics and its performance optimization.

In this paper, we explore the design space (syntax and semantics) of adding such directive into a future OpenMP specification. Using a prototype implementation in Clang, we demonstrate the usefulness of such directives on a few benchmarks.

Keywords: OpenMP · Pragma · C/C++ · Clang · Polly

1 Introduction

In scientific computing, but also in most other kinds of applications, the majority of execution time is spent in loops. Consequently, when it comes to improving an application's performance, optimizing the hot loops and their bodies is the most obvious strategy.

While code should be written in a way that is the easiest to understand, it will likely not the variant the will execute the fastest. Platform details such as cache hierarchies, data temporal/spatial locality, prefetching, NUMA, SIMD, SIMT, occupancy, branch prediction, parallelism, work-groups, etc. will have a profound impact on application performance such that restructuring the loop is necessary. Since an application rarely runs on just a single platform, one may end up in multiple versions of the same code: One that is written without considering hardware details, and (at least) one for each supported platform, possibly even using different programming models.

OpenMP is intended to be a programming model for many architectures, and ideally would allow to share the same code all of them. It is comparatively low-effort to replace an OpenMP directive, for instance, using the C/C++ preprocessor and OpenMP 5.0 introduced direct support for this via the `metadirective`.

Currently, this can only change the parallelization, offloading and vectorization decisions, but not the structure of the code itself.

In our last year's contribution [5], we proposed additional directives in OpenMP for transforming loops, e.g. loop fusion/fission, interchange, unrolling etc. In this paper, we discuss choices of syntactic and semantics elements (Sect. 2) for such an addition, give and update on our prototype implementation (Sect. 3), and demonstrate how loop transformation can be used in applications and the performance improvements (Sect. 4).

2   Specification Design Considerations

In this section we explore some of the decisions to make for including loop transformation directives into a potential newer OpenMP standard. By its nature, this cannot be an exhaustive discussion, but a subjective selection of the most important features that came up in discussion with members of the OpenMP language committee members and others.

The first decision to make is whether to include such directive at all. Since the "MP" in OpenMP stands for "**M**ulti**P**rocessing", obviously the original targets of OpenMP were (symmetric) multi-core and -socket platforms and still today, most implementation are based on the pthreads API. Multiprocessing obviously does not include sequential loop transformations, but this is not per se a reason to exclude such transformations from OpenMP.

For one, there is a need of supporting functionality: The `collapse` clause has been added in OpenMP 3.0, although it is not directly related to multiprocessing. OpenACC [7] also supports a tile-clause. The `simd` construct has been added in OpenMP 4.0, which is exploits instruction-level parallelism, which also not included in the term multiprocessing.

Second, the scope of OpenMP has extended relative to its original goal. With target offloading also introduced in OpenMP 4.0, it also supports accelerators such as GPGPUs and FPGAs.

There are alternatives to not include code transformations into OpenMP:

– Continue with the current practice of compiler-specific extensions. Without standardization, this means these will be incompatible to each other.
– Include into a future version the host languages' specifications (C/C++/Fortran). This would compel OpenMP to add clarifications how its directives interact with the host language's directives. However, it is questionable whether e.g. the C++ standard committee will add specifications of pragma-directives. Even if all host languages add transformation directives, their semantics are unlikely to match, complicating OpenMP compatibility clarifications.
– Create a separate language specification using C/C++/Fortran with OpenMP as its host language. This new language would probably diverge from OpenMP over time as they might add features incompatible to each other. Comparisons can be drawn from OpenACC, which started as an initiative to add accelerator offloading to OpenMP.

For the directives themselves, we distinguish three aspects: Syntax, semantics and the available code transformations. The syntax describes which token streams are accepted by the compiler and the semantics define their meaning. Once these base rules have been defined, it should be straightforward to add transformations consistent with these rules.

2.1   Syntax

In our first proposal [5], we suggested the following syntax

```
#pragma omp [loop(<loopname(s)>)] <transformation> <clauses...>
```

i.e. every transformation is a top-level directive. The `loop`-clause before the directive could be used to refer to a loop that is not on the following line or the result of another transformation on the next line. Since then, the OpenMP 5.0 standard was announced which includes a `loop`-directive. Even though a disambiguation is possible using the parenthesis following the clause, but not the directive, overloading the keyword might be ambiguous. Hence, we explore alternatives in this section.

Loop Directive. OpenMP 5.0 introduced the loop construct with the goal to give the compiler more freedom on optimization decisions. The first OpenMP specification was designed with symmetric multiprocessing in mind, but in the era of heterogeneous computing sensible defaults vary widely.

The idea of the loop-directive was to become the new default worksharing construct, since in most cases, or at least before performance-optimizing an application, the programmer does not care about how the body is executed in parallel, as long as the default choice is reasonable. In future OpenMP revisions, the loop-construct would gain features of the prescriptive worksharing-construct and preferred when adding new features. This maxim also applies to transformation-directives.

Clauses or (Sub-)Constructs. A transformation could be either expressed as a construct (as in [5]), or as a clause. Constructs usually indicate to the compiler to do something, whereas clauses pass options to the construct's doing. Therefore, a clause requires a construct to be added to.

Currently, OpenMP already uses both syntactic elements for what we might consider loop transformations. For instance, *#pragma omp simd* can be seen as a loop transformation that does vectorization. On the other side, the collapse clause (valid for multiple constructs such as loop, simd, etc.) is a transformation that occurs before the construct's effect.

When using the loop-construct, the transformation could either be clauses like the collapse-clause, or sub-constructs of the loop clause, similarly as every OpenMP construct is follows after an "omp" token. However, this would be a new syntactic element in OpenMP, since e.g. *#pragma omp for simd* is a combined construct, each of them can be used independently.

The order of any OpenMP 5.0 clauses is irrelevant, but transformations carried out in different orders generally result in different loop nests. This contradiction can be solved by either make such clauses order-dependent, require the compiler to ignore the order and instead apply an heuristic to determine the best order, or disallow multiple transformations on a single pragma.

If using the (sub-)construct as the primary syntax, clauses can still be allowed as syntactic sugar where it makes sense and does not cause ambiguity. Combined constructs could be allowed as well.

Loop Chains. Bertolacci et. al. [1] proposed a loopchain-construct with a schedule-clause. The loopchain encloses a loop nest to transform with the schedule clause that defines the transformations to apply on the loop nest, as illustrated in the example below (simplified from the paper).

```
#pragma omplc loopchain schedule(tile(10, parallel, serial))
{
  for (int i = lb ; i <= ub ; i += 1)
    A[i] = (B[i-1] + B[i] + B[i+1]);
  for (int i = lb ; i <= ub ; i += 1)
    A[i] = A[i] * (1.0 / 3.0);
}
```

Since the schedule applies the loop nest as a whole, the schedule must also specify an operation on parts that are not transformed. In the excerpt, the non-transformed part is indicated by the `serial` operator. If the loop chain is large with many transformations, the schedule clause can quickly become convoluted.

Referring to Other Loops. Some transformations such as tiling and loop fusion consume more than one loop on the next line and replace them with potentially more than one generated loop, which may be consumed by a follow-up transformation. For instance, the result of tiling two nested loops are four loops, we might want the parallelize the outermost, unroll-and-jam one of the middle loops and vectorize the innermost loop. Therefore, a syntax is needed to refer to loops that are not directly following the transformation directive.

This can either be done by assigning names to loops and referring to them, or with a path selector from the loop that is annotated. Loop names/identifiers have been described in [5], but also used by IBM xlc [4] and XLang [2].

Path selectors have been for node selection in trees, such as XPath [9] on XML. In some sense, the collapse clause, taking the number of perfectly nested loops as an argument, is such an selector. With more complex cases, such as "the third loop inside the following loop nest of two loops", maintainability becomes a problem: Adding or removing a loop before between the selector and the selected loop requires updating the selector.

2.2   Semantics

Prescriptive vs. Descriptive. Code transformations are inherently prescriptive: When used, the programmer is already working on performance optimization and cares about the executions order. The loop-construct is designed to be descriptive and, by default, applies the semantics of `order(concurrent)`, which allows the compiler to reorder the loop as it fits. Then changing the order using a loop transformation directive has no meaning: As the `order(concurrent)` clause allows an arbitrary permutation/thread-distribution, applying a user-defined permutation will have an undetermined result. It is also a worksharing-construct, meaning that it is meant to be executed in a `parallel` context. Non-worksharing, simple transformed loops would just run redundantly on every thread in the context.

One solution is to introduce new clauses that disable the default descriptive and worksharing behavior, such as `order(sequential)` and `noworksharing`. To avoid this boilerplate to be repeated with every loop construct, they might be implicit when a loop transformation is defined.

2.3   Level of Prescriptiveness

To avoid differences in performance when using different compilers, the specification should define the replacement code of a transformation. However, for code that is not performance-sensitive (such as edge cases, fallback code and pro- and epilogue), the compiler might retain some freedom. Taking the tile-construct as an example, the following decisions are not necessarily performance-relevant:

– Fallback code for rare cases where the transformation would be invalid, such as address range aliasing of two arrays that would cause a change in semantics.
– Where and how to execute partial tiles at the logical iteration space border: like a full tile but with additional border conditions or separately after/before all full tiles have been executed.
– If the iteration counter of the first iteration is not zero, divide tiles using the logical or physical iteration space?
– Assuming only the code inside a tile is performance-relevant, the outer iteration order over tiles does not need to be defined.
– If the specification allows tiling of non-perfectly nested loops, there is not obvious way to archive this.

A sensible approach could be to leave these decisions to the compiler, but consider adding clauses that fix this behavior.

OpenMP 5.0 already allows non-perfectly nested loops with the `collapse`-clause and only requires code between the loops to be executed at most as many times as it would be executed if moved inside the innermost loop, but at least as many times as in the original loops nest. Executing code more often than in the original code might be an unexpected side-effect of tiling. In the interest of user-friendless, the specification could disallow non-perfectly loop nests, but add a `nestify` transformation to make this behavior explicit in the code.

Transformation Order. The order in which multiple transformations are applied on the same loop can be either defined the programmer, the specification, or by the compiler. When defined by the programmer, the order is derived from the syntax. Otherwise, any order in the source is ignored and either the OpenMP specification has to specify the rule in which order transformations are applied, or it is implementation-defined such that the compiler can apply heuristics to determine the best ordering.

It might be straight-forward with transformations that consume one loop and replace it with another, but not all orderings are valid with other transformations. For instance, loop interchange requires at least two loops and cannot be applied if the previous transformation only returns a single loop. If the order is user-defined, the compiler can emit an error. Otherwise, either the OpenMP has to define which order to use, or the compiler developers.

However, performance optimization engineers will unlikely want to leave such decision up to the compiler or specification. This is because when using transformations, they will try to get a specific result that is optimal on the target platform and without transformation constructs, would write an alternative code path. A compiler "improving" its heuristic in later versions would also not helpful since it would regress the once-archived performance.

Compatibility with Legacy Directives. Several existing constructs and clauses in OpenMP can be interpreted as a loop transformation:

- The `for`, `loop` and `distribute`-constructs divide loop iterations between threads or teams.
- The `sections`-constructs distributes code regions between threads.
- The `simd` construct vectorizes a loop such that multiple input loop iterations are processed by one iteration of a generated loop, similarly to (partial) unrolling.

With this interpretation, applying other transformations to occur before and after the construct should be possible and make a syntax for new transformations that resemble existing transformations preferable.

Furthermore, existing combined constructs can be redefined as a sequence of transformations, instead of a textual definition. For instance,

```
#pragma omp for simd schedule(static) simdlen(4)
for (int i = 0; i < n; i+=1)
```

could be *defined* as

```
#pragma omp simd simdlen(4)
#pragma omp for schedule(static)
for (int i = 0; i < n; i+=1)
```

Note that this is different from

```
#pragma omp for schedule(static)
#pragma omp simd simdlen(4)
for (int i = 0; i < n; i+=1)
```

Table 1. Safety modes for transformation directives. Green is for safe transformations, red may have changed the code's semantics as does orange but only in corner cases.

	Heuristic	Default	Fallback	Force
always valid	*original or* transformed	transformed	transformed	transformed
valid with rtc	*original or* rtc	transformed	rtc	warning
invalid	*original*	transformed	warning	warning
impossible	*original*	warning	warning	warning

which might be more efficient if the number of iterations is not a multiple of the vector width. Using this transformation extension, it is possible to choose between the variants.

Semantic Safety. Generally, the OpenMP specification requires compilers to apply its directives without regard to whether it is semantically valid to do, i.e. the user guarantees that it is. This ensures that otherwise conservative compilers still honor the OpenMP directive, but defers the responsibility to the programmer.

In some scenarios the user might want the compiler to do a validity check. For instance, the programmer might be unsure themselves or the transformation is added by an autotuner trying out different loop transformations without understanding the code. For these cases, the directives may support options to instruct the compiler to verify semantic validity.

Table 1 shows how safety modes handle different situations for applying a code transformation. "Always valid" refers to code to which the transformation can be applied without changing its semantics. In the case of unrolling this is any loop since unrolling cannot change the code's effect (except execution time). "Valid with rtc" refers to code that can be transformed under conditions that can be checked dynamically. For instance, a transformation may require that two memory regions are not overlapping (alias), which can be checked at runtime if the compiler can deduce which addresses are accessed. "Invalid" means that the compiler cannot determine a reasonable runtime condition, i.e. must assume that the transformation will change the code's semantics. "Impossible" is code that the compiler can structurally impossible to transform, such as reversing a while-loop.

Note that these categories may depend on compiler capabilities; e.g. a compiler may have deduced the number of iterations of a while-loop. For the sake of a standardization, OpenMP should define minimum requirements for compilers to support with everything beyond being a quality-of-implementation.

Without OpenMP, the compiler would heuristically determine whether a transformation is profitable or not. Hence, it might apply it or not (indicated by

"original" in Table 1), but if it does, it has to ensure that the semantics do not change.

The default behavior of OpenMP directives[1] is to always apply even if it the code's semantics changes. It does not add a runtime check, meaning that the program result can also change in the "Valid with rtc" case. The compiler should emit a warning to the user if the transformation could not be applied at all.

With `fallback` semantics, the compiler must not emit semantically invalid code, but is allowed to generate fallback code in case a runtime condition fails. Still, it should warn if the transformation directive had no effect. In contrast to the heuristic approach, the compiler skips the profitability check and trusts the directive that the transformation is profitable.

Due to the possible fallback, it is still possible that the non-transformed code is executed without compiler warning and surprise the performance engineer. Instead `force` semantics can be used, which guarantees that either the transformed code is executed, or the compiler emits a warning. An additional `required` clause could change the warning to an hard error.

Another idea is a `hint` clause, which informs the compiler that the transformation is valid (i.e. skips the validity check), but still considers the profitability, possibly with a bump in favor of applying the transformation instead of the compiler's usual conservativeness.

2.4   Transformations

In addition to the general syntax and semantics, the available transformations have to be defined, including when they are applicable and what the result is. A convenient approach is to think of transformations as replacements: Remove the code it applies to and insert the result instead. Any follow-up transformation can apply on the transformed code as if the replacement was written in the source code. This should happen internally in the compiler, not textually.

In the remainder of the chapter, we try to define a selected set of transformations.

Loop Peeling. Some loop transformations work best when the loop is a multiple of a constant, such as (partial) unrolling, vectorization and tiling. If this is not a case, some iterations have to be extracted out of the main loop, which by itself is also a transformation. Unlike to relying on the implicit peeling, explicitly using a peeling transformation allows more options and naming the resulting prologue- and epilogue-loop to be referenced in follow-up transformations.

We can either the first k iterations into an prologue before the loop or the last k iterations into an epilogue after the loop. Peeling the first iterations is always possible, but for peeling the last iterations the number of iterations must be known in advance, which is the case of canonical loops as defined by OpenMP.

---

[1] Our previous paper [5] suggested to use safe semantics as the default, in conflict to the normal OpenMP behavior.
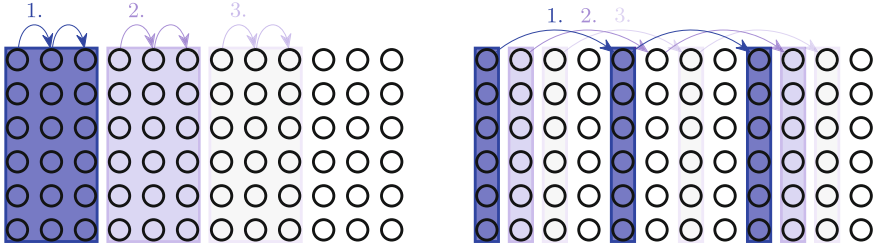
Fig. 1. (a) Strip-mining, (b) Stripe-mining

The number of iterations to peel can either be specified directly as the number k or indirectly as a goal to archive. A goal can be:

1. Make remaining main loop have a multiple of a constant number of iterations; useful for the aforementioned transformations.
2. Make the first access to an array aligned; useful for vectorized loads/stores and accesses that are faster when the compiler knows they are aligned.

Peeling might be necessary spanning multiple loops in a loop nests, since transformations like tiling and unroll-and-jam also apply on multiple nested loops.

Collapse. This combines multiple nested loops into a single logical loop that can be referred to by other transformations. It should not change the execution order of the inner body. OpenMP added a clause with similar semantics in version 3.0 and even assigns logical iteration numbers to loop body executions. A collapse loop-transformation would allow using this functionality independently of other constructs.

Strip- and Stripe-Mining. Strip-mining can be seen as one-dimensional tiling. In contrast to tiling in general, the execution order is not changed, i.e. like unrolling never changes the program's result. Unlike unrolling, it increases the control-flow complexity and therefore is only intended to be used in combination with other transformations. For instance, partial unrolling can be implemented by strip-mining followed by a full unroll of the inner loop. The name is inspired by the term from open-pit mining: The pit is deepened by one strip at a time, as visualized in Fig. 1a.

In contrast, stripe-mining does change the execution order: Each inner loop processes a constant number of iterations that are equidistantally distributed over the iteration space. As shown in Fig. 1b, each form a set of stripes, lending to the transformation's name.

3   Prototype Implementation

We created an implementation of some transformation directives in Clang and Polly, which we already described in [6]. Because such transformations are not part of OpenMP yet, use a hybrid of Clang's native syntax for loop transformation extensions and OpenMP construct/clauses syntax. The general syntax is:

```
#pragma clang loop(<loopname>) <transforamtion> <clauses...>
```

Our code is available on Github[2] Currently, it should be considered as prototype quality and is not intended for use in production. For instance, it may crash on syntax errors instead of diagnostic output.

In addition to the transformations mentioned in [6], we implemented unrolling, unroll-and-jam, thread-parallelization and peeling for tiled loops. The parallelization transformation, in contrast to OpenMP's worksharing constructs, can be combined with other transformations. It should become unnecessary once the interaction between OpenMP's parallelization constructs and loop transformations have been specified. We unfortunately did not implement loop distributions such that it had to be replicated manually for the evaluation.

4   Evaluation

In this section, we explore how transformation directives can be useful to improve the performance of a selection of kernels. Please keep in mind that we do not intend to discover new techniques how to improve these kernels over typically hand-optimized kernels in specialized libraries or in literature. Instead, we want to illustrate how these directives help exploring common optimization techniques. This is most relevant if no hand-optimized library for the kernel in question is available for a platform.

Unless mentioned otherwise, the execution time was measured on an Intel Core i7 7700HQ (Kaby Lake architecture), 2.8 Ghz with Turbo Boost off and compiled using the `-ffast-math` switch. When using parallelism, we use all 8 hardware threads (4 physical cores).

4.1   heat-3d

---

2 https://github.com/SOLLVE/clang/tree/pragma and https://github.com/SOLLVE/polly/tree/pragma.
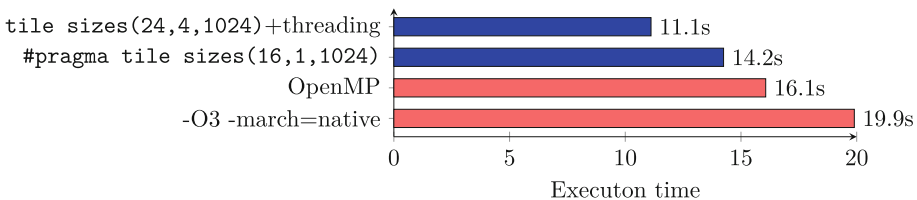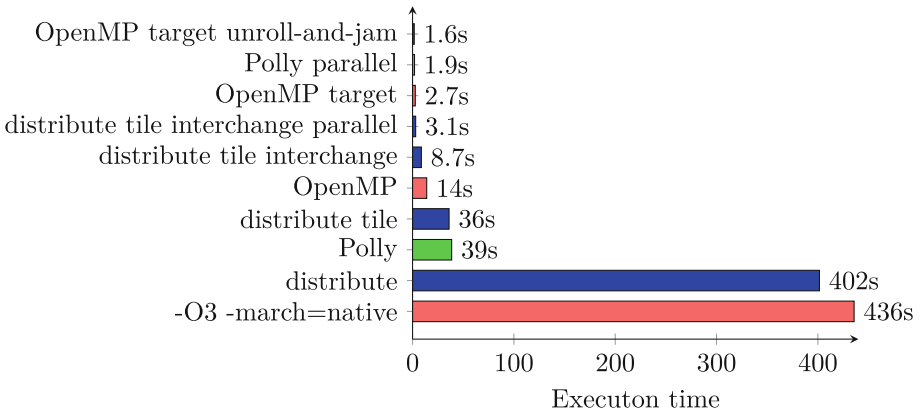
The benchmark "heat-3d" from Polybench [8] is 3-dimensional 10-point stencil. We are using a volume of 800^3 and 10 time-steps. Typical for repeated stencil codes, it alternatingly switches input- and output arrays. Its 3rd dimension makes it more difficult for the hardware prefetcher.

The baseline can be improved only slightly using OpenMP parallelism (*#pragma omp parallel for collapse(2)* and *#pragma omp simd* for the innermost loop). Tiling improves the performance even more on just a single thread, but can further improved with threading.

The tile sizes were determined using trial-and-error, a task which could also be done by an autotuner. More advanced time-tiling techniques such as diamond-overlap and tiling and could also result in an improvement.

4.2   syr2k

Polybench's "syr2k" is a rank-2k matrix-matrix update; we are benchmarking matrices of size 4000^2 and $2000 * 5200$. We run this benchmark on a 2-socket Intel Xeon Gold 6152 CPU (22 cores each, 88 threads in total) with an NVidia Tesla V100-SXM2 GPU.

Configuration	Execution time
OpenMP target unroll-and-jam	1.6s
Polly parallel	1.9s
OpenMP target	2.7s
distribute tile interchange parallel	3.1s
distribute tile interchange	8.7s
OpenMP	14s
distribute tile	36s
Polly	39s
distribute	402s
-O3 -march=native	436s

Executon time

We use the default DATASET_EXTRALARGE for Polybench's "syr2k". In contrast to the stencils, we can gain very high speed-ups.

While loop distribution does not gain a lot by itself, tiling (by $256 \times 96 \times 16$) improves the performance by a factor more than 11, followed by a speed-up of another 4x with a loop interchange. With parallelization on all 44 cores (88 threads), the execution time has improved by a factor of 140 over the original loop.

Interestingly, while single-threaded performance of the Polly-optimized version (using a tile size of 32 in all dimensions and not interchange) is worse, with parallelization it is even better with a speed-up factor of 330. Evidently, the shared memory bandwidth of the shared caches changes the bottleneck, such that the tile size optimized for single-thread performance is worse. Replication of Polly's optimized loop nest using pragmas replicates the same performance. We might be able to further improve the performance by searching for a tile size

that minimized the traffic higher-level caches. Using `#pragma omp parallel for` alone utilizing 88 OpenMP threads yields an improvement of the factor 31.

The performance characteristics changes when offloading to the GPU. With a straightforward `#pragma omp target teams distribute collapse(2)` of the outer loops and `#pragma omp parallel for reduction` of the inner loops, the kernel computes in 2.7 s, which is slower than the best CPU performance. Only with an additional unroll-and-jam did we beat the two CPUs. Tiling did not show any improvement.

4.3   covariance
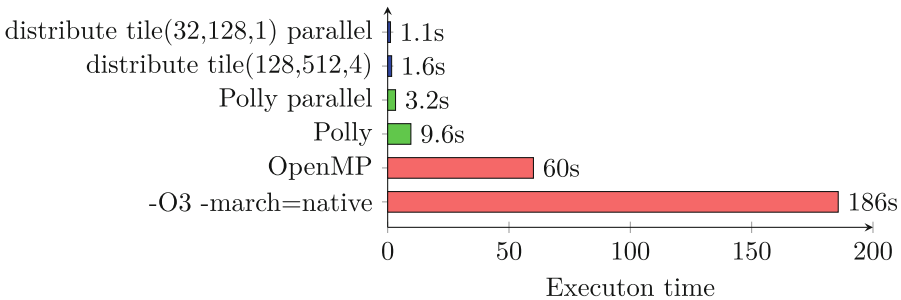
The main issue with the covariance benchmarks from Polybench is that the fastest iterator moves the outer data array dimensions leading to strided accesses which cause most of cache lines unused. If we just transpose the data array (manually), execution time already shrinks to 15 s. The problem can be lessened with tiling. Unlike the non-tiled version, parallelism improves the execution time only marginally.

Polly's sub-optimal choice of a tile size of 32 for each dimensions also leads to lower performance, for both, the parallel- and single-threaded cases.

4.4   dgemm
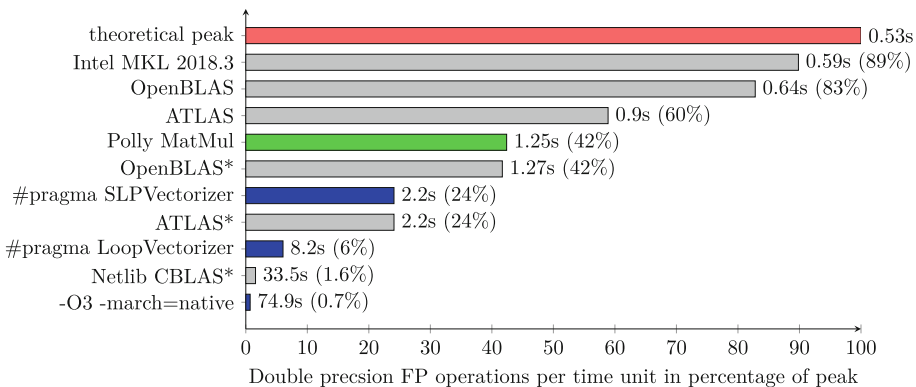
In [5], we already optimized Polybench's "gemm" kernel, but because of lack of support by LLVM's loop vectorizer, we could only vectorize the innermost loop.

This is sub-optimal because this means that the register dependency is also carried by the innermost loop, restricting the CPU's ability to reorder instructions.

```
#pragma clang loop(i1) pack array(B) \
  isl_redirect("{ [c,j,k]    -> [B[x,y] -> PackedB[floord(y,8) mod 256,x mod 256,y mod 8]] }")
#pragma clang loop(j2) pack array(A) \
  isl_redirect("{ [c,j,k,l] -> [A[x,y] -> PackedA[floord(x,4) mod 16 ,y mod 256,x mod 4]] }")
#pragma clang loop(i2) unrollingandjam factor(4)
#pragma clang loop(j2) unrollingandjam factor(8)
#pragma clang loop(i1,j1,k1,i2,j2) interchange permutation(j1,k1,i1,j2,i2)
#pragma clang loop(i,j,k) tile sizes(64,2048,256) \
                           floor_ids(i1,j1,k1) tile_ids(i2,j2,k2) peel(rectangular)
  for (int i = 0; i < M; i += 1)
        for (int j = 0; j < N; j += 1)
              for (int k = 0; k < K; k += 1)
                    C[i][j] += A[i][k] * B[k][j];
```

Fig. 2. Replication of Polly's matrix-multiplication optimization using directives; Libraries marked with (*) were precompiled from the Ubuntu software repository, hence not optimized for the evaluation system

To avoid this problem, Polly's matrix-multiplication optimization [3] unroll-and-jams non-inner loops and relies on LLVM's SLP vectorizer to combine the unrolled iterations into vector instructions. We replicate this behavior in Fig. 2. The `isl_redirect`-clause ensures that the packed arrays' data layout follow the changed access pattern. For production implementations of the array packing, this should be derived automatically by the compiler.

Unfortunately, the performance is even worse than with the innermost-loop vectorization because, unlike with Polly's output, the SLP vectorizer does vectorize the jammed loops. We are working on identifying and fixing the issue in the prototype version.

4.5   456.hmmer
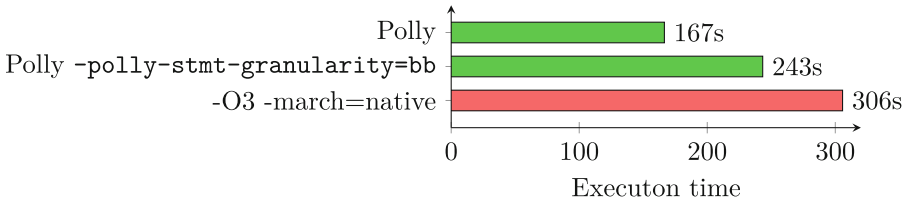
The most performance-critical code of "456.hmmer" from SPEC CPU 2006 is shown in Fig. 3. Even though it is just one loop, it does 3 logical computations, of which 2 have no loop-carried dependencies. Separating the sequential computation allows the parallelization and/or vectorization of the two other parts.

The vectorization speed-up (of the entire 456.hmmer run on an Intel Xeon E5-2667 v3 (Haswell architecture) running at 3.20 GHz) is shown in the graph. Earlier versions of Polly only separated one of the computations (using the `-polly-stmt-granularity=bb` option). However, the current version separates all 3 computations using its automatic optimizer. The same would be possible using a loop distribute directive.

```
for (k = 1; k <= M; k++) {
  mc[k] = mpp[k-1]    + tpmm[k-1];
  if ((sc = ip[k-1]  + tpim[k-1]) > mc[k])  mc[k] = sc;
  if ((sc = dpp[k-1] + tpdm[k-1]) > mc[k])  mc[k] = sc;
  if ((sc = xmb  + bp[k])         > mc[k])  mc[k] = sc;    Compute mc[k] (vectorizable)
  mc[k] += ms[k];
  if (mc[k] < -INFTY) mc[k] = -INFTY;
  dc[k] = dc[k-1] + tpdd[k-1];
  if ((sc = mc[k-1] + tpmd[k-1]) > dc[k]) dc[k] = sc;    Compute dc[k] (not vectorizable)
  if (dc[k] < -INFTY) dc[k] = -INFTY;
  if (k < M) {
    ic[k] = mpp[k] + tpmi[k];
    if ((sc = ip[k] + tpii[k]) > ic[k]) ic[k] = sc;    Compute ic[k] (vectorizable)
    ic[k] += is[k];
    if (ic[k] < -INFTY) ic[k] = -INFTY;
  }
}
```

Fig. 3. 456.hmmer hotspot code

5   Conclusion

Loop – and more generally, code – -transformation directives can be a useful tool to improve a hot code's performance without going too low-level. Completely automatic optimizers such as Polly rely on heuristics which are necessarily approximation they do not know the code's dynamic properties (such as number of loop iterations) and have an incomplete performance model of the target machine. They are also conservative, i.e. rather do nothing than to regress performance.

Transformation directives take the burden of profitability analysis off the compile and to the programmer who either knows which transformations are beneficial or can try out multiple approaches, possibly assisted by an autotuner.

We seek to add such transformation directives into a future OpenMP specification, to replace the current compiler-specific pragmas and ensure composability with OpenMP's directives. We discussed some design choices for syntax and semantics that have to be made with various (dis-)advantages in terms of compatibility, consistency, complexity of implementation and ease of understanding.

References

1. Bertolacci, I., Strout, M.M., de Supinski, B.R., Scogland, T.R.W., Davis, E.C., Olschanowsky, C.: Extending OpenMP to Facilitate loop optimization. In: de Supinski, B.R., Valero-Lara, P., Martorell, X., Mateo Bellido, S., Labarta, J. (eds.) IWOMP 2018. LNCS, vol. 11128, pp. 53–65. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-98521-3_4
2. Donadio, S., et al.: A language for the compact representation of multiple program versions. In: Ayguadé, E., Baumgartner, G., Ramanujam, J., Sadayappan, P. (eds.) LCPC 2005. LNCS, vol. 4339, pp. 136–151. Springer, Heidelberg (2006). https://doi.org/10.1007/978-3-540-69330-7_10
3. Gareev, R., Grosser, T., Kruse, M.: High-performance generalized tensor operations: a compiler-oriented approach. ACM Trans. Archit. Code Optim. **15**(3), 34:1–34:27 (2018). https://doi.org/10.1145/3235029
4. IBM: Product documentation for XL C/C++ for AIX, V13.1.3 (2015)
5. Kruse, M., Finkel, H.: A proposal for loop-transformation pragmas. In: de Supinski, B.R., Valero-Lara, P., Martorell, X., Mateo Bellido, S., Labarta, J. (eds.) IWOMP 2018. LNCS, vol. 11128, pp. 37–52. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-98521-3_3
6. Kruse, M., Finkel, H.: User-directed loop-transformations in Clang. In: 2018 IEEE/ACM 5th Workshop on the LLVM Compiler Infrastructure in HPC (LLVM-HPC), pp. 49–58, November 2018. https://doi.org/10.1109/LLVM-HPC.2018.8639402
7. OpenACC-Standard.org: The OpenACC Application Programming Interface Version 4.0, November 2017
8. Pouchet, L.N., Yuki, T.: Polybench 4.2.1 beta. https://sourceforge.net/projects/polybench
9. Spiegel, J., Robie, J., Dyck, M.: XML Path Language (XPath) 3.1. W3C recommendation, W3C, March 2017. https://www.w3.org/TR/2017/REC-xpath-31-20170321/