User-Directed Loop-Transformations in Clang

Michael Kruse
Argonne Leadership Computing Facility
Argonne National Laboratory
Argonne, USA
mkruse@anl.gov

Hal Finkel
Argonne Leadership Computing Facility
Argonne National Laboratory
Argonne, USA
hfinkel@anl.gov

***Abstract*—Directives for the compiler such as pragmas can help programmers to separate an algorithm's semantics from its optimization. This keeps the code understandable and easier to optimize for different platforms. Simple transformations such as loop unrolling are already implemented in most mainstream compilers. We recently submitted a proposal to add generalized loop transformations to the OpenMP standard. We are also working on an implementation in LLVM/Clang/Polly to show its feasibility and usefulness. The current prototype allows applying patterns common to matrix-matrix multiplication optimizations.**

***Index Terms*—OpenMP, Pragma, Loop Transformation, C/C++, Clang, LLVM, Polly**

I. Motivation

Almost all processor time is spent in some kind of loop, and as a result, loops are a primary target for program optimization efforts. Changing the code directly comes with the disadvantage of making the code much less maintainable. That is, it makes reading and understanding the code more difficult, bugs occur easier and making semantic changes that would be simple in an unoptimized version might require large changes in the optimized variant. Porting to a new system architecture may require redoing the entire optimization effort and maintain several copies of the same code each optimized for a different target, even if the optimization does not get down to the assembly level. Understandably, this is usually only done for the most performance-critical functions of a code base, if at all.

For this reason, mainstream compilers implement pragma directives with the goal of separating the code's semantics from its optimization. That is, the code should compute the same result if the directives are not present. For instance, pragmas defined by the OpenMP standard [1] and supported by most of the mainstream compilers will parallelize code to run using multiple threads on the same machine. An alternative is to use platform-specific thread libraries such as pthreads. OpenMP also defines directives for vectorization and accelerator offloading. Besides OpenMP, there are a few more sets of pragma directives, such as OpenACC [2], OmpSs [3], OpenHMPP [4], OpenStream [5], etc., with limited compiler support.

Besides the standardized pragmas, most compilers implement pragmas specific to their implementation. Table I shows a selection of pragmas supported by popular compilers. By their nature, support and syntax varies heavily between compilers. Only `#pragma unroll` has broad support. `#pragma ivdep` made popular by icc and Cray to help vectorization is mimicked by other compilers as well, but with different interpretations of its meaning. However, no compiler allows applying multiple transformations on a single loop systematically.

In addition to straightforward trial-and-error execution time optimization, code transformation pragmas can be useful for machine-learning assisted autotuning. The OpenMP approach is to make the programmer responsible for the semantic correctness of the transformation. This unfortunately makes it hard for an autotuner which only measures the timing difference without understanding the code. Such an autotuner would therefore likely suggest transformations that make the program return wrong results or crash. Assistance by the compiler which understands the semantics and thus can either refuse to apply a transformation or insert fallback code (*code versioning*) that is executed instead if the transformation is unsafe enables loop autotuning. Even for programmer-controlled programs, warnings by the compiler about transformations that might change the program's result can be helpful.

In summary, pragma directives for code transformations are useful for assisting program optimization and are already widely used. However, outside of OpenMP, these cannot be used for portable code since compilers disagree on syntax, semantics, and only support a subset of transformations.

Our contributions for improving the usability of user-directed loop transformations are

- the idea of making user-directed loop transformations arbitrarily composable,
- an effort to standardize loop-transformation pragmas in OpenMP [19], and
- a prototype implementation using Clang and Polly that implements additional loop-transformation pragmas.

II. Pragma Directives in Clang

Clang in the current version (7.0) already supports the following pragma directives:

- Thread-parallelism: `#pragma omp parallel`, `#pragma omp task`, etc.
- Accelerator offloading: `#pragma omp target`
- `#pragma clang loop unroll` (or `#pragma unroll`)
- `#pragma unroll_and_jam`
- `#pragma clang loop distribute(enable)`
- `#pragma clang loop vectorize(enable)` (or `#pragma omp simd`)

Transformation	Syntax	Compiler Support
Threading	`#pragma omp parallel for`	OpenMP [1]
	`#pragma loop(hint_parallel(0))`	msvc [6]
	`#pragma parallel`	icc [7]
	`#pragma concur`	PGI [8]
Offloading	`#pragma omp target`	OpenMP [1]
	`#pragma acc kernels`	OpenACC [2]
	`#pragma offload`	icc [7]
Unrolling	`#pragma unroll`	icc [7], xlc [9], clang [10], Oracle [11], PGI [8], SGI [12], HP [13]
	`#pragma clang loop unroll(enable)`	clang [14]
	`#pragma GCC unroll` n	gcc [15]
	`#pragma _CRI unroll`	Cray [16]
Unroll-and-jam	`#pragma unroll_and_jam`	icc [7], clang [17]
	`#pragma unroll`	SGI [12]
	`#pragma unrollandfuse`	xlc [9]
	`#pragma stream_unroll`	xlc [9]
Loop fusion	`#pragma nofusion`	icc [7]
	`#pragma fuse`	SGI [12]
	`#pragma _CRI fusion`	Cray [16]
Loop distribution	`#pragma distribute_point`	icc [7]
	`#pragma clang loop distribute(enable)`	clang[14]
	`#pragma fission`	SGI [12]
	`#pragma _CRI nofission`	Cray [16]
Loop blocking	`#pragma block_loop`	xlc [9]
	`#pragma blocking size`	SGI [12]
	`#pragma _CRI blockingsize`	Cray [16]
Vectorization	`#pragma omp simd`	OpenMP [1]
	`#pragma simd`	icc [7]
	`#pragma vector`	icc [7], PGI [8]
	`#pragma loop(no_vector)`	msvc [6]
	`#pragma clang loop vectorize(enable)`	clang [14, 18]
Interleaving	`#pragma clang loop interleave(enable)`	clang [14, 18]
Software pipelining	`#pragma swp`	icc [7]
	`#pragma pipeloop`	Oracle [11]
Loop name	`#pragma loopid`	xlc [9]
Loop versioning	`#pragma altcode`	PGI [8]
Loop-invariant code motion	`#pragma noinvarif`	PGI [8]
Prefetching	`#pragma mem prefetch`	PGI [8]
	`#pragma prefetch`	SGI [12]
Interchange	`#pragma interchange`	SGI [12]
	`#pragma _CRI interchange`	Cray [16]
If-conversion	`#pragma IF_CONVERT`	HP [13]
Collapse loops	`#pragma _CRI collapse`	Cray [16]
Assume iteration independence	`#pragma ivdep`	icc [7], PGI [8], SGI [12],
	`#pragma GCC ivdep`	gcc [15]
	`#pragma loop(ivdep)`	msvc [6]
	`#pragma nomemorydepend`	Oracle [11]
	`#pragma nodepchk`	PGI [8], HP [13]
Iteration count estimation	`#pragma loop_count(`n`)`	icc [7]

TABLE I: Loop pragmas and the compilers which support them

- `#pragma clang loop interleave(enable)`

A. Front-End

Clang's current architecture (shown in Fig. 1) has two places where code transformations occur:

1) OpenMP (except `#pragma omp simd`) is implemented at the front-end level: The generated LLVM-IR contains calls to the OpenMP runtime.
2) Compiler-driven optimizations are implemented in the mid-end: A set of transformation passes that each consume LLVM-IR with loops and output transformed IR, but metadata attached to loops can influence the passes' decisions.

This split unfortunately means that OpenMP-parallel loops are opaque to the LLVM passes further down the pipeline. Also, loops that are the result of other transformations (e.g. loop distribution) cannot be parallelized this way because the parallelization must have happened earlier in the front-end.

Multiple groups are working on improving the situation by adding parallel semantics to the IR specification [20, 21]. These and other approaches have been presented on LLVM's mailing list [22, 23] or its conferences [24, 25].

B. Mid-End

Any transformation pragma not handled in the front-end, including `#pragma omp simd`, is lowered to loop metadata. Metadata is a mechanism for annotating LLVM IR instructions with additional information, such as debug info. Passes can look up these annotations and change their behavior accordingly. An

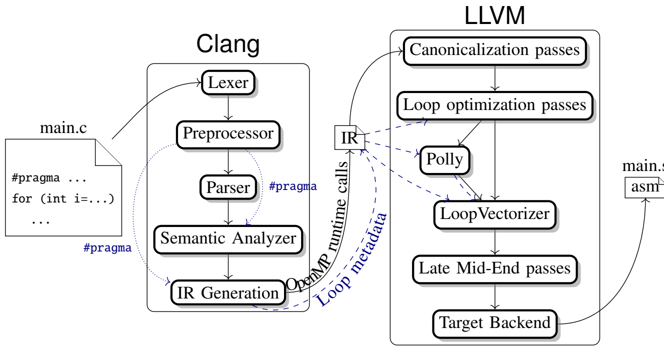


Fig. 1: Clang/LLVM compiler pipeline

LLVM Pass	Metadata
(Simple-)LoopUnswitch	*none*
LoopIdiom	*none*
LoopDeletion	*none*
LoopInterchange*	*none*
SimpleLoopUnroll	`llvm.loop.unroll.*`
LoopReroll*	*none*
LoopVersioningLICM+*	`llvm.loop.licm_versioning.disable`
LoopDistribute+	`llvm.loop.distribute.enable`
LoopVectorize+	`llvm.loop.vectorize.*` `llvm.loop.interleave.count` `llvm.loop.isvectorized`
LoopLoadElimination+	*none*
LoopUnrollAndJam*	`llvm.loop.unroll_and_jam.*`
LoopUnroll	`llvm.loop.unroll.*`
various	`llvm.mem.parallel_loop_access`

TABLE II: Loop transformation passes in the pipeline order and the metadata they process. Loop passes that canonicalize or only modify/move instructions without changing the loop structure are not included. Passes marked as * are not added to any default pipeline (such as `-O3`). Passes with a + marker can do code versioning.

overview of loop transformation passes and the metadata they look for is shown in Table II.

Some of the passes must be enabled explicitly, even when using the higher optimization level. For instance, the switch `-enable-unroll-and-jam` adds the LoopUnrollAndJam pass to the pipeline. When using clang, `-mllvm -enable-unroll-and-jam` has to be used. When not in the pass pipeline, any `#pragma unroll_and_jam` will be silently ignored.

For each function, the transformations are executed in the order of the passes in the pipeline. This implicitly defines order in case multiple pragmas are defined on the same loop. For instance,

```
#pragma clang loop vectorize(enable) distribute(enable)
```

will first try to distribute a loop, then vectorize the output loops. The order of the directives and/or pragmas is irrelevant. With this design it is not possible to apply transformations in any other order, similar to the transformation of OpenMP in the front-end. For instance, it is not possible to distribute a loop and then recognize a memcpy in one of the resulting loops using LoopIdiom. It is also not possible to apply a transformation multiple times, unless the responsible pass is scheduled multiple times as well.

Ideally, the pass structure of the mid-end should be an implementation detail, but as we have seen, it is visible through the execution order of the pragmas. Implementation details can change between versions, and even the optimization level (`-O0`, `-O1`, ...).

Loop passes can use LoopVersioning to get a clone of the instructions and control flow they want to transform. That is, each loop transformation makes its own copy of the code, with a potential code growth that is exponential in the number of passes since the versioned fallback code may still be optimized by further loop passes. The pass pipeline has up to 4 passes which potentially version, leading to up to 16 variants of the same code. In addition, many versioning conditions, such as checking for array aliasing, will be similar or equal for each versioning.

III. Extending Loop-Transformations Pragmas

We are working on improving support for user-directed code transformations in Clang, especially loop transformations. This includes addressing the shortcomings of the existing infrastructure described in the previous section and entirely new transformations.

If we only improve code transformations in Clang, the problem of the non-portability like for any of the pragmas in Table I remains. For this reason, we submitted a proposal for inclusion into the OpenMP standard [19]. We hope that this will improve the adoption of extended pragmas.

No discussion about an OpenMP standardization has started yet, hence in this paper we only describe our prototype implementation in Clang. In particular, the pragma syntax is different because the `#pragma omp` keyword is reserved for standardized OpenMP directives.

A. Transformation Order

The transformation order of the current pragmas is basically undefined. For the limited set of pragmas available, transformations are either commutative or a different order makes little sense. However, for additional transformations, the transformation order may become important.

The idea is that a pragma applies to the code that follows. That is, if the next line is a canonical for-loop, then that loop is transformed. If the next line is another pragma, then the output of that pragma is transformed. This means that the example

```
#pragma clang loop reverse
#pragma clang loop unroll factor(2)
for (int i = 0; i < 128; i+=1)
  Stmt(i);
```

will be partially unrolled by a factor of 2, then reversed. This will yield code comparable to

```
for (int i = 126; i >= 0; i-=2) {
  Stmt(i);
  Stmt(i+1);
}
```

instead of

```
for (int i = 127; i >= 0; i-=2) {
  Stmt(i);
  Stmt(i-1);
}
```

that would be the result if the transformations were applied in the reverse order.

B. Followup-Attributes

The metadata from Table II is used as a bag of attributes to a loop. By definition, their order is unimportant and cannot be used to pass ordered transformations.

In a first RFC on the LLVM mailing list [26], we suggested to add a function-wide (ordered) list of transformations of any loop in the function. While this works and our current prototype uses it, it has some problems. First, the function inliner needs to merge such lists. Second, it requires a reference to the loop to transform. Unfortunately, due to the nature of IR metadata, the "loop id" currently used by LLVM changes whenever the loop's attribute list is changed[1]. Third, the same "loop id" can be assigned to multiple loops. Naive cloning of code (like LoopVersioning does) will reuse the same "loop id" for a copied loop. There are even regression tests for multiple loops with the same "id". Fourth, transformation passes need to search the entire list for transformations they can apply and ensure that no other transformation is applied before it.

Our eventual approach [27] is more compatible with "loop ids" as bag of attributes by specifying new attributes. Every transformation gets *followup-attribute lists*. The list defines the bag of attributes a result loop will have. For instance, the `llvm.loop.unroll.followup_unrolled` attribute contains the metadata for the (partially) unrolled output loop. If not specified, the pass can automatically define the attributes; in case of LoopUnroll, it uses `llvm.loop.unroll.disable` to inhibit further unrolling.

Transformations can also have multiple output loops. In case of partial unrolling, there can be an epilogue for iterations that do not fill up the unroll factor. Its attributes can be defined using the `llvm.loop.unroll.followup_remainder` attribute. Typically, this loop is completely unrolled such there is no loop the attributes can be assigned to.

If a transformation cannot be applied for any reason, it is straightforward to also ignore the followup-attributes as they are meant for the transformed loop which does not exist. We decided against applying some followups even if a transformation failed (such as applying `followup_unrolled` even if the partial unrolling failed) or add an additional "backup" attribute list. This would significantly increase the systems complexity and the expected reaction is to fix the input code and/or transformation instead of specifying another chain of transformations.

Instead, if the transformation was explicitly requested by the programmer (which we call "forced"), the compiler should emit a warning that something did not apply as expected. In our proposal, this is done by a new WarnMissedTransformationsPass which is inserted into the pass manager after all transformation pass have run. Hence, in contrast to LLVM's current approach, it will even emit a warning if the responsible pass is not even in the pipeline. This unfortunately also means that the WarnMissedTransformations pass needs to understand all transformation metadata.

[1]For instance, LoopVersioningLICM adds `llvm.loop.licm_versioning.disable` to indicate that it does not have to run on a loop again.

The advantage to this approach is that is more compatible with the current implementation. If followup-attributes are not used, the behavior remains the same, even without mitigations such as IR conversions through AutoUpgrade.

C. Syntax

Unfortunately, the existing `#pragma clang loop` syntax already has a de-facto defined order, which is the order in Table II. To maintain compatibility, we have to introduce a distinguishable new syntax. When exclusively using the old syntax, clang has to emit the order that original pipeline would have. To avoid confusion, we disallow mixing old and new syntax.

The old syntax is

`#pragma clang loop` *transformation*`(`*option*`)` ...

and allows multiple *transformation*`(`*option*`)` on each line. Multiple options for the same transformation can be specified by using multiple variants of *transformation*. For instance, `vectorize(assume_safety) vectorize_width(4)` tells the compiler to skip semantic legality checks and use an SIMD width of 4. More complicated transformations such as tiling can have many options, which makes this syntax impractical. It is also ambiguous whether a transformation should be applied multiple times or whether only an option is set.

Since our goal is an inclusion into the OpenMP standard, we use its directive-clause syntax:

`#pragma clang loop` *transformation switch option*`(`*argument*`)` ...

The old and new syntax are distinguishable by the option in parenthesis immediately following the transformation keyword.

Interestingly, Clang's mechanisms for *`#pragma clang loop`* and OpenMP are quite different. The prototype implementations is oriented towards the *`#pragma clang loop`* route, because it is also used for the preexisting loop transformation pragmas and in contrast to the OpenMP path, is less invasive.

1) Preprocessor: The clang-loop pragma handler takes the tokens of each clause (*transformation*`(`*option*`)`), wraps each of them into a `annot_pragma_loop_hint` token and pushes that back into the token stream. For out new syntax we introduce a new token `annot_pragma_loop_transform` which contains the entire line of tokens of the pragma instead of individual clauses. The OpenMP pragma preprocessor also takes the entire line of tokens, encloses them between a `annot_pragma_openmp` and `annot_pragma_openmp_end` token, and pushes them all back to into the token stream.

This step is necessary because the tokens might be inside a `_Pragma("...")` of a macro. The preprocessor might therefore duplicate the tokens wherever the macro is expanded. The special tokens act as indicators for the parser that such a directive has been inserted.

2) Parser: The parser will handle the indicator tokens at expected positions. If an `annot_pragma_loop_hint` is encountered, the transformation and its option are parsed, and the result used to initialize an attribute of type LoopHintAttr. Since the data a LoopHintAttr can store is limited and unspecific, we introduce

a separate attribute for each of our new transformations when encountering a `annot_pragma_loop_transform`.

The OpenMP parser on the other hand does not create attributes, but uses the Sema object (calling its `ActOn...` methods), the semantic analyzer which also creates the AST. For parsing expressions according to the language rules, its tokens must be on the main token stack. For `annot_pragma_loop_*` this means that the wrapped tokens have to be pushed back to the stack before the expression parser is invoked. This is easier for the OpenMP pragma tokens which are already on the stack, terminated by a `annot_pragma_openmp_end` token that inhibits the expression parser from consuming tokens that do not belong to the pragma.

3) Semantic Analysis: The semantic analyzer is responsible for building the abstract syntax tree. For the LoopHintAttr and other transformation attributes, it creates an AttributedStmt as parent of the node that represents the loop that is annotated. It also checks the semantic correctness of the attributes, for instance, it emits warnings when the same `annot_pragma_loop_hint` clause is used more than once.

In the case of OpenMP, the pragmas are more invasive. Every OpenMP-annotated loop is nested into a CapturedStmt region which is handled like an outlined function, even for `#pragma omp simd`. Moreover, most OpenMP directives have their own type of AST node (in addition to CapturedStmt). That is, the AST will look differently compared to if OpenMP was disabled.

4) Code Generation: While Clang generates LLVM-IR for a loop, it also collects loop attributes. After the IR of the loop skeleton (loop header, latch, etc.) is complete, Clang sets the loop's metadata (see Table II). Of course, loops can be nested and hence there is a stack if loop attributes called LoopInfoStack.

Since OpenMP is handled in the front-end, there is much more to do. Depending on the directive, a CapturedStmt is either outlined in a separate function or into the same function. In the former case, a call to the OpenMP runtime (libomp) with a pointer to the outlined function. In other cases, the captured-but-expanded-inline loop body will be simplified again in the mid-end. For the `#pragma omp simd` directive, the loop will receive a meta annotation just as in the `#pragma clang loop vectorize` case.

D. Compposibility

Our goal is that the loops resulting from a transformation can again be transformed using a pragma. For clang, there is the problem that thread-parallelism is handled in the front-end and LLVM-IR has no semantics for parallelism. That is, its pragmas cannot apply on loops that are only created in the mid-end, nor can a `#pragma omp for` loop be processed in the mid-end. As a temporarily solution, we think about adding non-OpenMP pragmas for thread-parallelism that are handled in the mid-end, such as *`#pragma clang loop thread_parallel`*.

Moreover, transformations can have more than one input loop (such as OpenMP's `collapse` clause), and more than one output loop (such as strip-mining). To be able to refer to specific loops, we allow to assign identifiers to loop. A loop identifier must be unique within a function. For instance,

```
#pragma clang loop id(i)
for (int i = 0; i < n; i += 1)
```

assigns the identifier `i` to the loop. In case a loop is a canonical for-loop, its induction variable name might be used as a loop identifier unless it is ambiguous.

The loop identifiers can be used by transformations to refer to loops that are not on the next line. For instance, unroll-and-jam requires a loop to be unrolled and one to be jammed. In the following example, the loop `i` would be unrolled-and-jammed into the `k`-loop (instead the `j`-loop which would yield two `k`-loops inside it).

```
#pragma clang loop(i,k) unroll_and_jam factor(2)
for (int i = 0; i < n; i += 1)
  for (int j = 0; j < n; j += 1)
    for (int k = 0; k < n; k += 1)
```

In case there are multiple output loops, the pragma can define the identifiers of the those:

```
#pragma clang loop stripmine size(4) pit_id(i1) strip_id(i2)
for (int i = 0; i < 128; i += 1)
```

Its result is equivalent to as if the programmer had written the following.

```
#pragma clang loop id(i1)
for (int i1 = 0; i1 < 128; i1 += 4)
  #pragma clang loop id(i2)
  for (int i2 = i1; i2 < i1+4; i2 += 1)
```

Some transformations can be understood as syntactic sugar for other transformations. For instance, the aforementioned unroll-and-jam can also be expressed as an unroll followed be (one or multiple) loop fusions. Standard loop tiling is nothing else than strip-mining of each loop, then permute the loop nest order such that the strip loops become the interior loops.

E. Additional Transformations

Some new transformations have already been mentioned in the examples for illustration purposes. Many more are possible, see Table I and [19] for ideas.

In our prototype, we started implementing a limited set that are of immediate interest for us, mainly optimizing a matrix-matrix multiplication as shown in Section IV. In addition to `#pragma clang loop id`, we implemented the following pragmas.

1) Loop Reversal: Invert the iteration order of a loop. I.e.

```
#pragma clang loop reverse
for (int i = 0; i < n; i+=1)
```

is transformed into

```
for (int i = n-1; i >= 0; i-=1)
```

This was the first transformation we implemented because it is one of the simplest: Exactly one input and output loop.

2) Loop Interchange: Permute the order of perfectly nested loops. For instance, the result of

```
#pragma clang loop(i,j) interchange permutation(j,i)
for (int i = 0; i < n; i+=1)
  for (int j = 0; j < m; j+=1)
```

is

```
for (int j = 0; j < m; j+=1)
  for (int i = 0; i < n; i+=1)
```

The order of more than two loops can be altered by explicitly specifying the permutation.

3) Tiling: Tiling is a technique to improve temporal locality of accesses, especially of stencils. The example

```
#pragma clang loop(i,j) tile sizes(4,8)
for (int i = 0; i < n; i+=1)
  for (int j = 0; j < m; j+=1)
```

should be transformed into the following:

```
for (int i1 = 0; i1 < n; i1+=4)
  for (int j1 = 0; j1 < m; j1+=8)
    for (int i2 = i1; i2 < n && i2 < i1+4; i2+=1)
      for (int j2 = j2; j2 < m && j2 < j1+8; j2+=1)
```

Any number of loops can be tiled. Tiling just a single loop is the same as strip-mining.

4) Array Packing: Temporarily copies the data of a loop's working set into a new buffer. This may improve access locality because the extracted working set fits into a cache level and/or can be prefetched. For example, the loop nest

```
for (int i = 0; i < n; i+=1)
  #pragma clang loop pack array(A)
  for (int j = 0; j < 32; j+=1)
    f(A[j][i], i);
```

is transformed to something approximately equivalent of:

```
auto Packed_A[32];
for (int i = 0; i < n; i+=1) {
  for (int _ = 0; _ < 32; _+=1)
    Packed_A[_] = A[_][i]; // Copy-in
  for (int j = 0; j < 32; j+=1)
    f(Packed_A[j], i);
}
```

By default, the packed array is allocated on the stack, but with the clause `allocate(malloc)`, the memory will be allocated (and free'd) on the heap.

F. Polly as Loop-Transformer

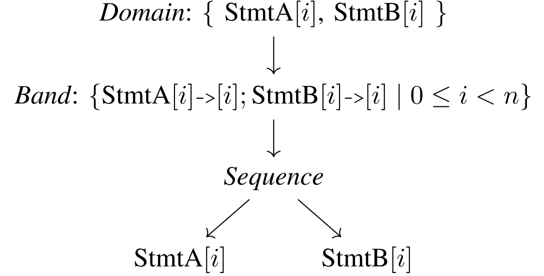
Writing a new loop transformation pass in LLVM is a significant amount of work since it works on the low-level IR. Some components such LoopVersioning can be reused, but even the dependency analysis will probably have to be written from scratch. This is despite LLVM has multiple dependency analyses (AliasAnalysis, DependenceInfo, LoopAccessInfo, PolyhedralInfo), but which have all been written with specific applications in mind.

Additionally, the pass manager architecture (neither the new nor "legacy") does not allow dynamically repeated application of transformation passes. This would also amplify the code blowup due to repeated code versioning.

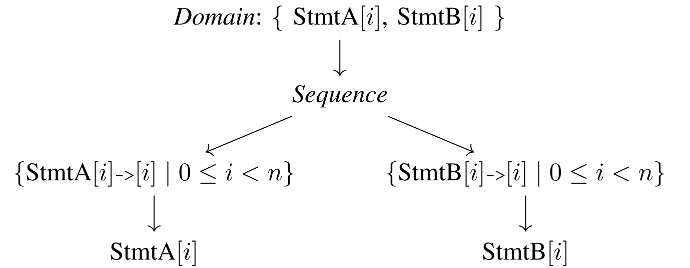
For our prototype implementation, we are using Polly [28] to implement the additional transformations. Polly takes LLVM-IR code and 'lifts' is into another representation –*schedule trees* [29] – in which loop transformations are easier to express. To transform loops, only the schedule tree needs to be changed and Polly takes care for everything else.

Using Polly, we can implement most transformations as follows. First, let Polly create a schedule tree for a loop nest, then iteratively apply each transformation in the metadata to the schedule tree. For every transformation we can check whether it violates any dependencies and if violations are found, act according to a chosen policy. When done, Polly generates LLVM-IR from the schedule tree including code versioning.

Let's consider an example on how schedule trees are transformed. Below we see the schedule tree of a single loop with two statements: StmtA and StmtB. Both statements execute in the same loop. The execution order of the loop is defined by the lexicographic ordering of the band's schedule function. Hence, the effective execution order of all statements is: StmtA[0], StmtB[0], StmtA[1], StmtB[1], ...



Interchanging the band and sequence node in the schedule tree is isomorphic to a loop distribution of the source code it represents, as shown below. Here, the sequence ensures that all instances of StmtA are executed before any instance of StmtB, but the band ordering between the statement instances is not changed. Therefore, this tree represents the execution order StmtA[0], StmtA[1], ..., StmtB[0], StmtB[1], ...



Polly's infrastructure then converts the schedule tree into an AST and then back to LLVM-IR, but with the transformation applied. Only a single fallback copy for arbitrarily many transformations is generated, if needed at all. Once the runtime check confirms that the preconditions making the transformation valid (e.g. no overlapping memory regions), no additional checking is required. Another advantage of a single dedicated loop transformation pass is that the analyses, including dependency analysis, needs to happen once only, instead repeatedly for every transformation.

If desired, Polly can also apply its loop nest optimizer which utilizes a linear program solver before IR generation. We add artificial *transformational dependencies* to ensure that user-defined transformations are not overridden, but we did not implement this is the prototype yet.

As an exception, `#pragma clang loop pack` cannot implemented using this technique as it is mainly a data layout transformation. Only the copy-in and copy-out to the local memory allocation modify the schedule tree; these are new statements that are inserted before, respectively after the code that uses them. Parts of the code already existed in Polly as part of its matrix-matrix multiplication optimization [30], but had to be generalized to

```
#if __kabylake__
  #pragma clang loop(j2) pack array(A)
  #pragma clang loop(i1) pack array(B)
  #pragma clang loop(i1,j1,k1,i2,j2,k2) interchange \
          permutation(j1,k1,i1,j2,i2,k2)
  #pragma clang loop(i,j,k) tile sizes(96,2048,256) \
          pit_ids(i1,j1,k1) tile_ids(i2,j2,k2)
#elif __haswell__
  [...]
#endif

#pragma clang loop id(i)
for (int i = 0; i < M; i+=1)
  #pragma clang loop id(j)
  for (int j = 0; j < N; j+=1)
    #pragma clang loop id(k)
    for (int k = 0; k < K; k+=1)
      C[i][j] += A[i][k] * B[k][j];
```

Listing 1: Optimization of matrix-matrix multiplication using our proposed pragmas. The tile sizes were derived using the analytical model in [33] for Intel's Kaby Lake architecture.

arbitrary loops. To define an index function and size of the packed array, we use the bounding box technique from [31]. The possibility to allocate such arrays on the heap instead on the stack (`allocate(malloc)`) has been added in a Google Summer of Code project [32].

IV. Matrix-Matrix Multiplication

We chose this matrix-matrix multiplication to illustrate the power of user-directed transformations with relatively few lines and separation of semantics and optimization through pragmas. Matrix-matrix multiplication is one of the best studied problems for performance-optimization with many BLAS library implementations that we can assume to be close to the best attainable performance. Comparing to them allows estimating the gap between hand-optimized implementations and compiler-produced code. In this paper we are not searching for the best transformation chains for arbitrary algorithms[2], but to show how pragmas can make implementing such algorithms easier, potentially even in those specialized libraries.

Fortunately, the paper [33] describes the common techniques for optimizing matrix-matrix multiplication such that we do not need to find the optimal transformations ourselves. Our version is shown in Listing 1, which also illustrates how different transformations can be applied for different compilation targets. It only contains the most performance-sensitive loop. Most of the time, matrix-matrix multiplication is written including a statement `C[i][j] = 0` in the loop nest to clear the content the array C might have had before. To also optimize this formulation, we would have to loop-distribute the set-zero statement and inner reduction into two different loop nests. Unfortunately, our prototype does not support loop distribution yet and LLVM's LoopDistribution pass is not able to handle deeply nested loops. With loop distribution, the set-zero loop nest could be transformed into a single `memset` call. Again, LLVM's LoopIdiom pass only supports innermost loops such that it would result in a loop of memsets.

[2]We could only compare the pragma implementation with our manual replication of the same – which should behave identically.

Polly's main output is LLVM's intermediate representation, but it can also dump the AST representation from which the IR is generated, shown in Listing 2. The AST representation is one of isl's data structures.

The runtime check verifies the assumptions that must hold to make this transformation valid. For instance, it ensures that that arrays `A` and `B` do not overlap.

It is important that the array `Packed_B` is transposed. Otherwise, the accesses to are strided (not consecutive) in the innermost loop `c5` such that the processor's prefetcher will not work as efficient and the L1 cache lines are not fully used. In our experiments, without transpose the kernel was ten times slower.

The technique in [33] suggests that the dimensions `j2` and `i2` should be vectorized. In contrast to the loop `k2`, these do not carry the reduction, hence have less data-flow dependencies. Unfortunately, LLVM's LoopVectorize currently only supports innermost loops, hence the `k2`-loop is the only one we can directly vectorize, but its heuristics decide that it is profitable without us having to add another pragma. Extending LoopVectorize to more than innermost loops is work in progress [34]. Another project addressing the issue is the Unified Region Vectorizer [35], which is not part of LLVM. Polly's matrix-matrix multiplication optimization [30] has a workaround that unroll-and-jams (it calls it register tiling) the loops to be vectorized and then relies on the SLPVectorizer to combine the unrolled instructions to vector instructions. The same unroll-and-jam is also applied on the packed arrays. We could also try to use Polly's vector code generator (`-mllvm -polly-vectorizer`) which unfortunately also prioritizes innermost loops.

A. Effectiveness

We tested the execution speed of Listings 1 and 2, and compared it compared it with other implementations such as various BLAS libraries. All execution times were taken for a single-thread double-precision matrix-multiplication kernel using the parameters $M = 2000$, $N = 2300$, $K = 2600$. The results are shown in Fig. 2.

The naïve version (Listing 1 without pragmas) compiled with Clang 7.0 executes in 75 seconds (gcc's results are similar). With the pragma transformations manually applied (like Listing 2 but without `floord` calls) the execution time shrinks to 3.9 seconds. If Polly applies these transformations as directed by the pragmas in Listing 2, the runtime shrinks even more to 2.2 seconds. This is possible because Polly applies additional metadata that indicate that, for instance, the arrays do not alias. Polly's matrix-matrix multiplication recognition [30] optimizes the kernel such that it runs in 1.14s, which is 42% of the processor's theoretical floating-point limit. This speed should eventually also be reachable using pragmas after we improved the vectorizer situation.

By comparison, Netlib's reference BLAS implemented requires 33.5 seconds to do the multiplication. ATLAS from

```
double Packed_B[256][2048];
double Packed_A[96][256];
if (`\emph{runtime check}`) {
  if (M >= 1)
    for (int c0 = 0; c0 <= floord(N - 1, 2048); c0 += 1)     // Loop j1
      for (int c1 = 0; c1 <= floord(K - 1, 256); c1 += 1) { // Loop k1

        // Copy-in: B -> Packed_B
        for (int c4 = 0; c4 <= min(2047, N - 2048 * c0 - 1); c4 += 1)
          for (int c5 = 0; c5 <= min(255, K - 256 * c1 - 1); c5 += 1)
            Packed_B[c4][c5] = B[256 * c1 + c5][2048 * c0 + c4];

        for (int c2 = 0; c2 <= floord(M - 1, 96); c2 += 1) { // Loop i1

          // Copy-in: A -> Packed_A
          for (int c6 = 0; c6 <= min(95, M - 96 * c2 - 1); c6 += 1)
            for (int c7 = 0; c7 <= min(255, K - 256 * c1 - 1); c7 += 1)
              Packed_A[c6][c7] = A[96 * c2 + c6][256 * c1 + c7];

          for (int c3 = 0; c3 <= min(2047, N - 2048 * c0 - 1); c3 += 1)   // Loop j2
            for (int c4 = 0; c4 <= min(95, M - 96 * c2 - 1); c4 += 1)     // Loop i2
              for (int c5 = 0; c5 <= min(255, K - 256 * c1 - 1); c5 += 1) // Loop k2
                C[96 * c2 + c4][2048 * c0 + c3] += Packed_A[c4][c5] * Packed_B[c3][c5];
        }
      }
} else {  /* original code */ }
```

Listing 2: Transformed loop nest of Listing 1; this is the AST as emitted by Polly (`-mllvm -debug-only=polly-ast`) modified for readability.

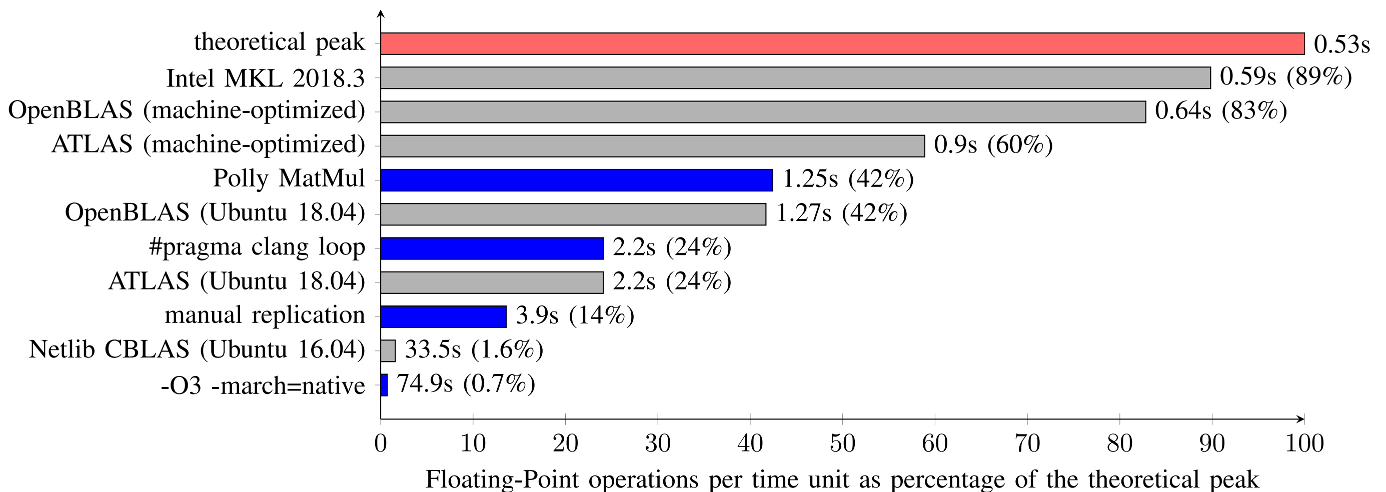


Fig. 2: Comparison of double-precision matrix-multiplication performance on an Intel Core i7 7700HQ (Kaby Lake architecture), 2.8 Ghz, Turbo Boost off

the Ubuntu 18.04 (Xenial Xerus) software repository needs 2.2s for the same work, but when optimized for the target machine, only needs 0.9 seconds. OpenBLAS, also from the Ubuntu software repository, needs 1.3 seconds, and only 0.6 seconds when compiled for the target machine. Intel's MKL library runs in 0.59 seconds, which is impressive 89% of the theoretical flop-limited peak performance. ATLAS, OpenBLAS and MKL use hand-written vector kernels for the innermost tiles.

V. Related Work

As we have seen in Table I, other compilers also implement pragmas. Most of them only allow controlling their existing loop optimization passes in the pipeline in the same manner as LLVM does. For instance, gcc version 8.1 [15] adds support for `#pragma unroll`, but does not support any other transformation pragma although gcc has more loop transformations.

We used Polly to implement most loop transformations, but by default it tries to automatically determine which transformations are profitable using linear programming. Apart from the equation system solver, Polly also applies tiling and a matrix-matrix multiplication optimization [30] whenever possible but can only be controlled via command-line flags, not in-source annotations.

IBM's xlf compiler has a dedicated loop transformation component, called ASTI [36]. It's data structure is the Loop Structure Graph (LSG) which shares some similarities to isl's schedule trees and might be able to carry-out xlf's supported

pragmas.

Silicon Graphics also developed a compiler with a dedicated loop transformation phase called Loop Nest Optimization (LNO) [12] which was used to implements its numerous (compared to other compilers) loop transformations. Today, the compiler lives on in its derivatives. One of them is the open source Open64 compiler [37] which also contains the LNO component.

Multiple research groups already explored the composition of loop transformations, many of them based on the polyhedral model. The Unifying Reordering Framework [38] describes loop transformations mathematically, including semantic legality and code generations. The Clint [39] tool is able to visualize multiple loop transformations.

Many source-to-source compilers can apply the loop transformations themselves and generate a new source file with the transformation baked-in. The instructions of which transformations to apply can be in the source file itself like in a comment of the input language (Clay [40], Goofi [41], Orio [42]) or like our proposal as a pragma (X-Language [43], HMPP [44]). Goofi also comes with a graphical tool with a preview of the loop transformations. The other possibility is to have the transformations in a separate file, as done by URUK [45] and CHiLL [46]. POET [47] uses an XML-like description file that only contains the loop body code in the target language.

Halide [48] and Tensor Comprehensions [49] are both libraries that include a compiler. In Halide, a syntax tree is created from C++ expression templates. In Tensor Comprehensions, the source is passed as a string which is parsed by the library. Both libraries have objects representing the code and calling its methods transform the represented code.

Similar to the parallel extensions in the C++17 [50] standard library, Intel's Threading Building Blocks [51], RAJA [52] and Kokkos [53] are template libraries. The payload code is written using lambdas and an *execution policy* specifies how it should be called.

Our intended use case – autotuning loop transformations – has also been explored by POET [47] and Orio [42]. The atJIT [54] project is even able to use our extended transformation in Polly. It tries out different transformations of a function within as single processes using a Just-In-Time compiler.

VI. Conclusion

Our goal is to make more loop transformations via pragma directives available to the programmer. Such pragmas would make applying common loop optimization technique much easier and allow better separation of a code's semantics and its optimization.

We are working on two fronts to make it happen: First, we try to add such pragmas to the OpenMP standard [19]. This would encourage any compiler that claims OpenMP-compatibility to implement them.

The second approach is to implement such pragmas in LLVM. In this paper we presented the details of our prototype implementation using Clang to parse the pragmas and Polly to carry-out the transformations. Experiments on matrix-matrix multiplication code show that kernels optimized using our pragmas can – performance-wise – be in the realm of hand-optimized BLAS libraries.

VII. Acknowledgments

This research was supported by the Exascale Computing Project (17-SC-20-SC), a collaborative effort of two U.S. Department of Energy organizations (Office of Science and the National Nuclear Security Administration) responsible for the planning and preparation of a capable exascale ecosystem, including software, applications, hardware, advanced system engineering, and early testbed platforms, in support of the nations exascale computing imperative.

This research used resources of the Argonne Leadership Computing Facility, which is a DOE Office of Science User Facility supported under Contract DE-AC02-06CH11357.

References

[1] *OpenMP Application Program Interface Version 4.0*, OpenMP Architecture Review Board, Jul. 2017.

[2] *The OpenACC Application Programming Interface Version 4.0*, OpenACC-Standard.org, Nov. 2017.

[3] A. Duran, E. Ayguadé, R. M. Badia, J. Labarta, L. Martinell, X. Martorell, and J. Planas, "Ompss: A Proposal for Programming Heterogeneous Multi-Core Architectures," *Parallel Processing Letters*, vol. 21, no. 02, pp. 173–193, 2011.

[4] J. M. Andión, M. Arenaz, F. Bodin, G. Rodríguez, and J. Touriño, "Locality-Aware Automatic Parallelization for GPGPU with OpenHMPP Directives," *International Journal of Parallel Programming*, vol. 44, no. 3, pp. 620–643, Jun. 2016.

[5] A. Pop and A. Cohen, "OpenStream: Expressiveness and Data-flow Compilation of OpenMP Streaming Programs," *ACM Trans. Archit. Code Optim.*, vol. 9, no. 4, pp. 53:1–53:25, Jan. 2013. [Online]. Available: http://doi.acm.org/10.1145/2400682.2400712

[6] *C/C++ Preprocessor Reference*, Microsoft. [Online]. Available: http://docs.microsoft.com/en-us/cpp/preprocessor/loop

[7] *Intel C++ Compiler 18.0 Developer Guide and Reference*, Intel, May 2018.

[8] *PGI version 18.7 Documentation for x86 and NVIDIA Processors*, PGI. [Online]. Available: https://www.pgroup.com/resources/docs/18.7/x86/pgi-ref-guide/index.htm#directive-pragma-ref

[9] *Product documentation for XL C/C++ for AIX, V13.1.3*, IBM.

[10] *Attributes in Clang*. [Online]. Available: http://clang.llvm.org/docs/AttributeReference.html

[11] *Sun Studio 12: C User's Guide, 2.8 Pragmas*, Oracle Corporation. [Online]. Available: https://docs.oracle.com/cd/E19205-01/819-5265/bjaby/index.html

[12] *MIPSpro N32/64 Compiling and Performance Tuning Guide*, SGI.

[13] *HP aC++/HP C A.06.29 Programmer's Guide*, Hewlett-Packard.

[14] *Clang Language Extensions*. [Online]. Available: http://clang.llvm.org/docs/LanguageExtensions.html

[15] *Loop-Specific Pragmas*, Free Software Foundation. [Online]. Available: http://gcc.gnu.org/onlinedocs/gcc/Loop-Specific-Pragmas.html

[16] *Cray C and C++ Reference Manual (8.7)*, Cray. [Online]. Available: https://pubs.cray.com/content/S-2179/8.7/cray-c-and-c++-reference-manual/scalar-optimization-directives

[17] D. Green, *Add unroll_and_jam pragma handling*. [Online]. Available: https://reviews.llvm.org/rL338566

[18] *Auto-Vectorization in LLVM*. [Online]. Available: http://llvm.org/docs/Vectorizers.html

[19] M. Kruse and H. Finkel, "A Proposal for Loop-Transformation Pragmas," *CoRR*, vol. abs/1805.03374, 2018. [Online]. Available: http://arxiv.org/abs/1805.03374

[20] X. Tian, H. Saito, E. Su, J. Lin, S. Guggilla, D. Caballero, M. Masten, A. Savonichev, M. Rice, E. Demikhovsky, A. Zaks, G. Rapaport, A. Gaba, V. Porpodas, and E. Garcia, "LLVM Compiler Implementation for Explicit Parallelization and SIMD Vectorization," in *Proceedings of the Fourth*

Workshop on the LLVM Compiler Infrastructure in HPC, ser. LLVM-HPC'17. New York, NY, USA: ACM, 2017, pp. 4:1–4:11.

[21] T. B. Schardl, W. S. Moses, and C. E. Leiserson, "Tapir: Embedding Fork-Join Parallelism into LLVM's Intermediate Representation," in *Proceedings of the 22nd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP'17)*. ACM, 2017, pp. 249–265.

[22] H. Finkel and X. Tian, "[RFC] IR-level Region Annotations," llvm-dev mailing list post, Jan. 2017. [Online]. Available: http://lists.llvm.org/pipermail/llvm-dev/2017-January/108906.html

[23] J. Doerfert, "[rfc] abstract parallel ir optimizations," llvm-dev mailing list post, Jun. 2018. [Online]. Available: http://lists.llvm.org/pipermail/llvm-dev/2018-June/123841.html

[24] H. Saito, "Extending LoopVectorizer towards supporting OpenMP4.5 SIMD and outer loop auto-vectorization," EuroLLVM 2018 presentation, 2016. [Online]. Available: http://llvm.org/devmtg/2016-11/#talk7

[25] H. Finkel, J. Doerfert, X. Tian, and G. Stelle, "A Parallel IR in Real Life: Optimizing OpenMP," EuroLLVM 2018 presentation, 2018. [Online]. Available: http://llvm.org/devmtg/2018-04/talks.html#Talk_1

[26] M. Kruse, "RFC: Extending loop metadata," llvm-dev mailing list post, May 2018. [Online]. Available: https://lists.llvm.org/pipermail/llvm-dev/2018-May/123690.html

[27] ——, "[Unroll/UnrollAndJam/Vectorizer/Distribute] Add followup loop attributes," LLVM code review, Jul. 2018. [Online]. Available: https://reviews.llvm.org/D49281

[28] T. Grosser, H. Zheng, R. Aloor, A. Simbürger, A. Größlinger, and L.-N. Pouchet, "Polly – Polyhedral Optimization in LLVM," in *First International Workshop on Polyhedral Compilation Techniques (IMPACT'11)*, 2011.

[29] S. Verdoolaege, S. Guelton, T. Grosser, and A. Cohen, "Schedule Trees," in *Fourth International Workshop on Polyhedral Compilation Techniques (IMPACT'14)*, 2014.

[30] R. Gareev, T. Grosser, and M. Kruse, "High-Performance Generalized Tensor Operations: A Compiler-Oriented Approach," *ACM Trans. Archit. Code Optim.*, vol. 15, no. 3, pp. 34:1–34:27, Sep. 2018.

[31] M. Kruse, "Lattice QCD Optimization and Polytopic Representations of Distributed Memory," Theses, Université Paris Sud - Paris XI, Sep. 2014. [Online]. Available: https://hal.inria.fr/tel-01078440

[32] N. Bonfante, "Maximal static expansion for efficient loop parallelization on gpu," Google Summer of Code project, 2017. [Online]. Available: http://pollylabs.org/gsoc2017/Maximal-static-expansion-for-efficient-loop-parallelization-on-GPU.html

[33] T. M. Low, F. D. Igual, T. M. Smith, and E. S. Quintana-Orti, "Analytical Modeling Is Enough for High-Performance BLIS," *Transactions on Mathematical Software (TOMS)*, vol. 43, no. 2, pp. 12:1–12:18, Aug. 2016.

[34] A. Zaks and G. Rapaport, "Vectorizing Loops with VPlan - Current State and Next Step," LLVM Developer's Meeting presentation, Oct. 2017. [Online]. Available: https://llvm.org/devmtg/2017-10/#talk17

[35] S. Moll, "The Region Vectorizer." [Online]. Available: https://github.com/cdl-saarland/rv

[36] V. Sarkar, "Automatic selection of high-order transformations in the ibm xl fortran compilers," *IBM Journal of Research and Development*, vol. 41, no. 3, pp. 233–264, May 1997.

[37] "Open64 Compiler and Tools." [Online]. Available: https://sourceforge.net/projects/open64/

[38] W. Kelly and W. Pugh, "A Framework for Unifying Reordering

[42] A. Hartono, B. Norris, and P. Sadayappan, "Annotation-Based Empirical Performance Tuning Using Orio," in *Proceedings of the 23rd IEEE International Parallel And Distributed Computing Symposium (IPDPS'09)*.

Transformations," University of Maryland, Technical Report UMIACS-TR-93-134/CS-TR-3193, 1992.

[39] O. Zinenko, S. Huot, and C. Bastoul, "Clint: A Direct Manipulation Tool for Parallelizing Compute-Intensive Program Parts," in *2014 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*. IEEE, 2014.

[40] L. Bagnères, O. Zinenko, S. Huot, and C. Bastoul, "Opening Polyhedral Compiler's Black Box," in *14th Annual IEEE/ACM International Symposium on Code Generation and Optimization (CGO'16)*. IEEE, 2016.

[41] R. Müller-Pfefferkorn, W. E. Nagel, and B. Trenkler, "Optimizing Cache Access: A Tool for Source-to-Source Transformations and Real-Life Compiler Tests," in *Proceedings of the 10th International Euro-Par Conference (Euro-Par'04)*. Springer, 2004.
IEEE, 2009.

[43] S. Donadio, J. Brodman, T. Roeder, K. Yotov, D. Barthou, A. Cohen, M. J. Garzarán, D. Padua, and K. Pingali, "A Language for the Compact Representation of Multiple Program Versions," in *Proceedings of the 18th International Workshop on Languages and Compilers for Parallel Computing (LCPC'05)*. Springer, 2006, pp. 136–151.

[44] R. Dolbeau, S. Bihan, and F. Bodin, "HMPP: A Hybrid Multi-core Parallel Programming Environment," in *First Workshop on General Purpose Processing on Graphics Processing Units (GPGPU'07)*, 2007.

[45] S. Girbal, N. Vasilache, C. Bastoul, A. Cohen, D. Parello, M. Sigler, and O. Temam, "Semi-Automatic Composition of Loop Transformations for Deep Parallelism and Memory Hierarchies," *International Journal of Parallel Programming*, vol. 34, no. 3, pp. 261–317, Jun. 2006.

[46] A. Tiwari, C. Chen, J. Chame, M. Hall, and J. K. Hollingsworth, "A Scalable Auto-tuning Framework for Compiler Optimization," in *Proceedings of the 23rd IEEE International Parallel And Distributed Computing Symposium (IPDPS'09)*. IEEE, 2009.

[47] Q. Yi, K. Seymour, H. You, R. Vuduc, and D. Quinlan, "POET: Parameterized Optimizations for Empirical Tuning," in *Proceedings of the 21st IEEE International Parallel And Distributed Computing Symposium (IPDPS'07)*. IEEE, 2007.

[48] J. Ragan-Kelley, C. Barnes, A. Adams, S. Paris, F. Durand, and S. Amarasinghe, "Halide: A Language and Compiler for Optimizing Parallelism, Locality, and Recomputation in Image Processing Pipelines," in *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'13)*. ACM, 2013, pp. 519–530.

[49] N. Vasilache, O. Zinenko, T. Theodoridis, P. Goyal, Z. DeVito, W. S. Moses, S. Verdoolaege, A. Adams, and A. Cohen, "Tensor Comprehensions: Framework-Agnostic High-Performance Machine Learning Abstractions," *CoRR*, vol. abs/1802.04730, 2018.

[50] *ISO/IEC 14882:2017*, International Organization for Standardization, Dec. 2017.

[51] *Threading Building Blocks*, Intel. [Online]. Available: https://www.threadingbuildingblocks.org

[52] R. D. Hornung and J. A. Keasler, "The RAJA Portability Layer: Overview and Status," Lawrence Livermore National Lab, Technical Report LLNL-TR-661403, 2014.

[53] H. C. Edwards, C. R. Trott, and D. Sunderland, "Kokkos: Enabling manycore performance portability through polymorphic memory access patterns," *Journal of Parallel and Distributed Computing*, vol. 74, no. 12, pp. 3202–3216, 2014.

[54] K. Farvardin, "atJIT: A just-in-time autotuning compiler for C++." [Online]. Available: https://github.com/kavon/atJIT