Preprint version

Parallel Processing Letters

AUTOMATIC DATA LAYOUT TRANSFORMATIONS IN THE EXASTENCILS CODE GENERATOR

STEFAN KRONAWITTER[†] and SEBASTIAN KUCKUK[‡] and
HARALD KÖSTLER[‡] and CHRISTIAN LENGAUER[†]

[†] *University of Passau, Passau, Germany*
[‡] *Friedrich-Alexander University of Erlangen-Nürnberg (FAU), Erlangen, Germany*

ABSTRACT

Performance optimizations should focus not only on the computations of an application, but also on the internal data layout. A well-known problem is whether a struct of arrays or an array of structs results in a higher performance for a particular application. Even though the switch from the one to the other is fairly simple to implement, testing both transformations can become laborious and error-prone. Additionally, there are more complex data layout transformations, such as a color splitting for multi-color kernels in the domain of stencil codes, that are manually difficult. As a remedy, we propose new flexible layout transformation statements for our domain-specific language ExaSlang that support arbitrary affine transformations. Since our code generator applies them automatically to the generated code, these statements enable the simple adaptation of the data layout without the need for any other modifications of the application code. This constitutes a big advance in the ease of testing and evaluating different memory layout schemes in order to identify the best.

Keywords: data layout transformation, color splitting, ExaStencils, stencil codes, domain-specific language, code generation

1. Introduction

Many scientific application areas, such as physics and chemistry simulations, require the solution of discretized partial differential equations (PDEs). Some of the most efficient solvers for this problem are from the class of multigrid methods [1, 2]. Tuning an implementation for a given hardware architecture and concrete problem specification is, however, notoriously challenging. Particularly the choice or adaptation of the underlying data structures and layout can be the deciding factor in attaining or preventing optimal performance. One prominent case in which an adaptation is necessary is that of multi-color kernels that are often favored for their parallelizability and numerical properties. E.g., the use of a simple Red-Black Gauss-Seidel (RBGS) smoother already results per se in an inefficient memory access pattern of only every other element. This reduces effective memory bandwidth and prevents suitable vectorization. As a remedy, a traditional optimization is to store the red and black points separately in memory. While easy to formulate on paper,

an efficient implementation is quite challenging, especially when done in the larger context of an existing solver [3, 4]. And, of course, it gets more complicated for more complex coloring schemes.

Most applications handle more than one physical quantity. Here, another well known layout transformation is to switch from a struct of arrays (SoA) to an array of structs (AoS) representation. Again, the most basic version applied to equally sized data fields can be implemented with reasonable effort. However, when staggered grids are used, such as in computational fluid dynamics (CFD), the number of data points per unknown and dimensions varies for each quantity, i.e., data field. Obviously, this complicates the implementation of such memory layout transformations severely.

With present-day high-performance computing (HPC) frameworks, the individual adaptation of the internal data layout to an application variant is usually infeasible. Moreover, choosing the suitable data layout a priori is virtually impossible since neither the target hardware nor the actual implementation of the compute kernels are known beforehand. The use of code generation techniques, as employed in ExaStencils, can be a viable solution since all data structures are compiled for a specific application. Moreover, an interface can lend users support in the adaptation of the underlying data structures and even in the exploration of new optimization techniques. We expound on the required functionality, i.e., the (automatic) memory layout transformations, in the code generator of project ExaStencils[a] [5]. Here, we go beyond traditional optimizations such as switching between SoA and AoS or reversing index dimensions. Instead, we implement an interface that facilitates the specification of arbitrary affine transformations with a concise and intuitive syntax. This allows not only color splitting with an arbitrary number of colors, but also researching new optimizations for applications that will be implemented in the future. We enhance our approach with the support of the concatenation of fields of non-matching sizes, which is necessary when operating on staggered grids for CFD simulations.

We make the following contributions:

- automatic data layout transformations implemented in the ExaStencils code generator,
- their exposition to users at a domain-specific language (DSL) level,
- a detailed description of their implementation,
- a demonstration of their efficiency for arbitrary affine transformations and
- a demonstration of their performance impact on CPUs and GPUs.

The rest of the paper is organized as follows. Section 2 briefly introduces our DSL ExaSlang, in which the layout transformations are integrated. The new language features are presented in Section 3.1, while Section 3.2 details how the layout transformations are implemented in our code generator. An evaluation of the performance impact and a presentation of how the layout transformation statements can be used is given in Section 4. Related work is discussed in Section 5 and Section 6 concludes.

[a] `http://www.exastencils.org/`

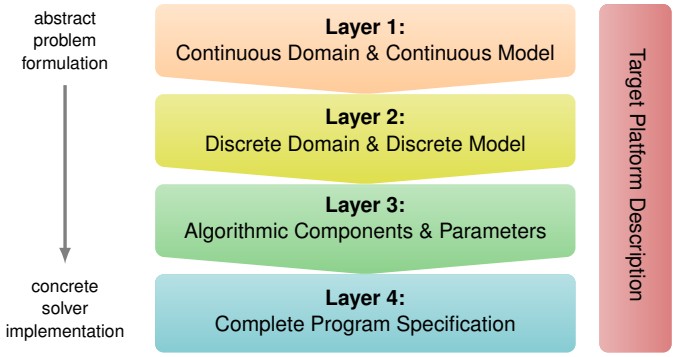

Fig. 1. ExaSlang is a DSL with four layers of abstraction. A separate target platform description language serves to specify properties of the target execution platform.

2. ExaSlang

The input for our code generator is a DSL called ExaSlang—short for ExaStencils language [6]. It consists of four different layers of abstraction, called ExaSlang 1 to 4, ranging from an abstract specification of the continuous PDE to be solved (ExaSlang 1) to a concrete definition of the implementation containing code for all components of the multigrid method (ExaSlang 4) as shown in Figure 1. The layout transformation that we discuss here resides in ExaSlang 4. To apply it, only few extensions to the language are required.

ExaSlang 4 is the most concrete form of ExaSlang, but it still contains language features that are specific to multigrid applications. From ExaSlang 4 code, the target code is generated—normally C++ plus MPI/OpenMP/CUDA or similar languages. Figure 2 shows ExaSlang 4 code for a RBGS smoother. ExaSlang 4 allows most objects, such as fields and functions, to be available at several levels of the multigrid hierarchy. Such level specifications and references are tied to the objects to which they belong via the `@` operator: function `Smoother` is available at all levels but the coarsest. The fields `Solution` and `RHS`, as well as the stencil `Laplace` in the function's body are also level-specific. If, as here, no level is specified explicitly, these identifiers reference the objects at the level at which the function is being applied. This can be highlighted by adding the optional label `@current`. Other than

```
Function Smoother@(all but coarsest) {
  color with {
    (i0+i1+i2) % 2,
    loop over Solution {
      Solution = Solution +
        1.0 / diag(Laplace) *
        (RHS - Laplace * Solution)
} } }
```

Fig. 2. ExaSlang 4 code for a RBGS smoother.

`current`, the keywords `coarser` and `finer` reference objects at the next coarser or finer level and the addition or subtraction of a constant offset, such as `current-2`, is also possible. Additionally, there are `coarsest`, `finest`, and `all` as well as the set difference `but`. A `color with` block starts with a non-empty sequence of modulo operations. Their numerators are expressions that may contain field iterators (`i0`, `i1`, ...) while their divisors must be natural numbers. These expressions specify a (possibly multi-dimensional) coloring that is applied to the `loop over` statements. In general, the `loop over` construct specifies a full iteration over a (multi-dimensional) field. However, the statements inside the `color with` block are executed once for each color and, hence, the `loop over` statements are restricted to the iterations across the current color. In the example, the body of the loop is executed first for all iterations for which the sum of the loop iterators `i0`, `i1`, and `i2` is even. In a second sweep, the remaining, odd iterations are executed.

3. Layout Transformations

Since the indices of field accesses are not stated in ExaSlang 4, the computation is abstracted from the actual memory layout, which specifies how field elements are organized in memory. The default memory layout is a direct mapping of the loop iteration vector to the position of the element accessed inside the memory. However, for a red-black coloring, this results in an update of only every other element, which reduces the effective memory bandwidth since data can only be transferred in chunks of 512 bits from and to main memory on current processor architectures. This becomes even more problematic if more than two colors are required. Our solution to this and similar problems is a language extension of ExaSlang 4 that provides a mechanism for the specification of arbitrary affine memory layout transformations. On the one hand, this provides users with the ability to experiment with different memory layouts. On the other hand, it represents a concise interface for the automatic tuning of the memory layouts of present fields in the future.

3.1. *New ExaSlang 4 Features*

A new top-level block, `LayoutTransformations`, gives ExaSlang 4 programmers or generators the opportunity to add three new kinds of directives. Figure 3 contains a simplified grammar of this extension.

3.1.1. *Renaming*

The first new statement, `rename`, is the simplest one. It takes the name of an existing field and a new, unused identifier. The original name is then replaced at all specified levels by the new one. The sole intent of this statement is to support the linking of generated code to external or legacy code. Note that `rename` is not intended to introduce aliasing and, thus, the new name introduced must not be used in the ExaSlang 4 code.

⟨*layoutTrafoBlock*⟩ ::= 'LayoutTransformations {' ⟨*layoutTrafoStmt-list*⟩ '}'

⟨*layoutTrafoStmt*⟩ ::= 'rename' ⟨*field*⟩ 'to' ⟨*field*⟩
| 'transform' ⟨*field-list*⟩ 'with [' ⟨*ident-list*⟩ '] => [' ⟨*expr-list*⟩ ']'
| 'concat' ⟨*levels?*⟩ ⟨*fieldName-list*⟩ 'into' ⟨*fieldName*⟩

⟨*field*⟩ ::= ⟨*fieldName*⟩⟨*levels?*⟩

Fig. 3. Simplified grammar of the new `LayoutTransformations` block. ⟨*X-list*⟩ specifies a repetition of the non-terminal ⟨*X*⟩, using a comma as a delimiter for all except the ⟨*layoutTrafoStmt-list*⟩, which does not require any special delimiter (other than spaces or newlines). The ⟨*field-list*⟩ and ⟨*fieldName-list*⟩ can be delimited alternatively with 'and'. ⟨*levels?*⟩ is an optional level specification prefixed with an `@` operator as introduced in Section 2. In the absence of a level specification, all levels are affected.

3.1.2. *Transformation*

An actual layout transformation can be specified with the `transform` statement. It is applied to all given fields individually and the desired transformation is specified by a linear mapping that assigns every element of the input field a new location.

For example, a color splitting of the fields `Solution` and `RHS` in Figure 2 is given in Figure 4. The first transformation adds a new dimension outermost with two possible values: one for the red and one for the black points. In ExaSlang 4, unlike in C, the elements of the leftmost dimension are stored consecutively in memory, i.e., in column-major order. This transformation results in the use of only every other element of the generated arrays. The second transformation then closes the gaps by scaling the innermost dimension down. Shrinking any other dimension would be possible as well but this would result in non-contiguous accesses. In contrast to two successive transformations for the color separation and the scaling, a single transformation specifying the composition of both is possible, too. Note that one need not specify how the extent of a dimension is affected by a `transform` statement or which extent a new dimensions has. This is incorporated automatically by the code generator as described in Section 3.2.2.

Since ExaSlang 4 supports also vectors and matrices as field element types [7], dimensions induced by them may also be specified in a transformation statement. Without such a directive, the new dimensions for vectors and matrices are added outermost, i.e., the default is an SoA representation. However, since the additional dimensions are not treated differently from the field dimensions, they can be in-

```
LayoutTransformations {
  transform Solution and RHS
    with [x,y,z] => [x,y,z,(x+y+z)%2]
  transform Solution and RHS
    with [x,y,z,c] => [x/2,y,z,c]
}
```

Fig. 4. ExaSlang 4 code for a layout transformation.

```
Function CopyField@all {
  loop over RHS_N {
    RHS_N = RHS
  }
}
```

(a) ExaSlang 4

```
void CopyField() {
  for (z = ...)
    for (y = ...)
      for (x = ...)
        RHS_N[z][y][x] =
          RHS[(x+y+z)%2][z][y][x/2];
}
```

(b) generated Pseudocode

Fig. 5. Explicit layout conversion for the transformation from Figure 4.

terchanged with the latter to achieve, e.g., an AoS layout. In the case of a 2D field and vector components, such a transformation is given by the expression `[x,y,v] => [v,x,y]`. As implied earlier, in case one only wants to transform the field dimensions, the additional dimensions of vectors or matrices need not be stated explicitly. Thus, the left-hand side of such a transformation may read `[x,y]`.

If a layout transformation is only advisable for a part of the application, an explicit conversion between different data layouts during the execution of the generated code is also possible. One simply declares separate fields for each data layout and targets them individually by appropriate layout transformations. Finally, an explicit conversion is then initiated by a simple copy loop as shown in Figure 5(a). Even though this loop looks like a naive one-to-one copy, existing data layout transformations for both fields are applied in the generated C++ code. E.g., for the layout transformation in Figure 4, the resulting C++ code is similar to the pseudocode shown in Figure 5(b).

3.1.3. *Concatenation*

The `concat` statement concatenates two or more fields to a new one. Only fields at the same level can be merged. Optionally, the level can be specified before the field names are listed. While the ExaSlang 4 code uses only the original, separated fields, their elements are placed in a single array in the generated C++ code. To separate the different input fields, a new dimension is added outermost whose possible values enumerate the original fields. Thus, this statement can be viewed as creating a SoA from the given fields. The extents of the inner dimensions are set to the maximum of the extents of all involved fields, which potentially introduces unused areas. The benefit of this statement is that the memory layout of the new, merged field can be adapted with the `transform` statement. For example, the new dimension can then be permuted to create an AoS. This may be useful in situations where elements of different fields are only accessed jointly. Furthermore, the original fields can also be transformed before concatenation.

3.2. *Implementation*

To apply such data layout transformations, we take advantage of the integer set library (isl) [8]. This C library offers functionality to represent and manipulate sets and relations of integer points bounded by affine inequalities. It was designed and implemented for polyhedral compilation [9], but also provides functionality that matches our requirements perfectly. For example, the application of a relation to a set, and the extrema computation for a set can be used to compute the new extents of a transformed data layout. And, since our code generator is capable of polyhedral optimization, the isl is already integrated and in use.

3.2.1. *Overview*

The overall structure of our layout transformation strategy is straight-forward:

(i) Collect all layout transformation statements.
(ii) Apply `transform` statements.
(iii) Perform all `rename` operations.
(iv) Create new fields for `concat` statements.
(v) If there is at least one `concat` statement, create and incorporate accesses to new fields and apply `transform` statements for them.

The first step (i) searches the entire abstract syntax tree (AST) for `LayoutTransformations` blocks and collects all their statements. In the second step (ii), all access expressions to fields are updated as specified by the `transform` statements. Excluded are only accesses to fields created with a `concat` statement, since they do not exist at this point but will be introduced later in step (v). Additionally, the field extents are adapted to ensure that the linearization is correct and a large enough chunk of memory will be allocated. Details of how a `transform` statement is applied are presented later in this section. Step (iii) performs all requested `rename` operations. Since every field is represented by a single object and these are not referenced by name in the AST, we can simply replace the `name` attribute of all targeted fields. The `concat` statements are processed in steps (iv) and (v). In the former, the new, concatenated fields are generated and some sanity checks are performed. For example, the data type of the field elements must be identical. The dimensionality of the new field is one larger than that of the original fields. The extent of this new dimension is the number of fields to be merged, while for the other dimensions the maximum of the extents from the old fields is set. Finally, in the last step (v), all accesses to any field that is to be concatenated with others are replaced by an access to the new field and a potential transformation is applied. Additionally, all other occurrences of an original field (in internal structures) are replaced by the new one.

Algorithm 1: Layout transformation for access expressions.

Input: The *access* expression to be transformed and a list of *enclosingConditions* for the current program location.

Output: The new, transformed field access expression.

Data: *accessTemplates* is an initially empty mapping of a field to an access template whose content is preserved between different calls and *transformStmts* is a mapping of each field to a list of transformation statements for it.

```
   Function process_access(access, enclosingConditions):
1      field ← get_field(access)
2      if field ∉ accessTemplates.keys() then
3          stmts ← transformStmts(field)
4          trafo ← identity()
5          foreach stmt ∈ stmts do
6              aff ← create_isl_multi_aff(stmt)
7              trafo ← aff ∘ trafo
8          update_layout(field, trafo)
9          accTempl ← create_access_template(trafo)
10         accessTemplates(field) ← accTempl
11     accTempl ← accessTemplates(field)
12     newAccess ← specialize(accTempl, access)
13     newAccess ← simplifyWith(newAccess, enclosingConditions)
14     return newAccess
```

3.2.2. *Access Transformation*

The basic idea of the automatic data layout transformations is to modify all accesses to a field in a common way. Additionally, the layout information of the fields must be updated, i.e., the extent of each dimension must be adjusted. Algorithm 1 computes new field accesses for the specified transformations. Prior to the execution of this function, level specifications introduced in Section 2, are completely resolved: all objects, including the field accesses and the fields themselves, are specialized for each level. Thus, a field is identified not only by its name but by a combination of its name and level.

Initially, in line 1, the *field* that is referenced by the given *access* expression is extracted. The remainder of the function can then be divided into two parts:

(i) Lines 3 to 10 update the field layout and generate a template with which the new, transformed accesses to the same field can be generated, while

(ii) lines 11 to 14 retrieve a previously computed template, specialize it to the given access and simplify the result.

On the one hand, the field layout modification of part (i) must be executed exactly once per field to ensure correctness. On the other hand, the template generation need not be repeated since its result can be cached and reused later. The *accessTemplates* mapping performs such a caching. Therefore, part (i) is executed only if there is no template available yet for the field in question. It starts by retrieving a list

of `transform` statements for the current *field* from the *transformStmts* mapping (line 3). For each of these statements, an isl representation of the transformation expression is generated (line 6) and these individual transformations are composed to yield a single one with the same semantics (line 7). The isl provides functionality to perform such a composition. Updating the field layout for the transformation in line 8 also relies on methods provided by the isl. In detail, the affine expression representing the layout transformation is first transformed to a relation between two integer sets. Second, an integer set containing all valid indices for the old field layout is created and, third, the previously computed relation is applied to it which results in the set of all valid indices for the new, transformed layout. Finally, it is ensured that the minimal value of every individual dimension of the new integer set (after projecting all other dimensions out) is nonnegative and the new extent of that dimension is set to its maximum value plus 1. Line 9 recreates an AST, namely the template for the transformed access, from the isl transformation expression with predefined dummies for the input variables. This new template is then stored in the *accessTemplates* mapping (line 10) for a later use and to indicate that the field layout is already modified.

In contrast to part (i), part (ii) must be executed for each access expression that has to be transformed. The access template for *field* is retrieved in line 11. This template is specialized in line 12 by replacing the i-th dummy variable with the i-th expression of the old *access*. Line 13 performs an optimization that targets explicitly a color splitting for a colored loop nest. For example, the loop in Figure 2 is executed once for all loop iterations `[x,y,z]` for which the expression `(x+y+z)%2` equals to `0` and once for `1`. If additionally a color splitting, as presented in Figure 4, is applied, the modulo computation in the new field access expressions inside these loops can be specialized to the constant `0` or `1`, respectively. The information to which value the coloring expression is restricted in the current context is given via the *enclosingConditions* parameter. This simplification tackles the more complex memory address computations introduced by a color splitting and, thus, has the potential to increase the performance of the generated code even further. The transformed and simplified access is eventually returned in line 14.

4. Evaluation

This section addresses the performance influence of data layout transformations, both for a single smoother and for the complete multigrid code, and provides the corresponding layout transformation statements for all experiments.

4.1. *Experiment Setup*

All CPU experiments were executed on single-socket nodes equipped with Intel Xeon E5-2690 v2 processors. They consist of 10 Ivy Bridge EP cores, each running at 3.3 GHz when fully loaded and with Intel Turbo Boost enabled. Each node is equipped with an NVIDIA GeForce GTX TITAN Black GPU, which contains 2880

(a) 2D 1st-order (b) 3D 1st-order (c) 2D dense (d) 3D dense

Fig. 6. 2D and 3D stencil shapes.

CUDA cores running at 677 MHz in double-precision mode. While the CPU has access to 64 GB DDR3 memory, the TITAN Black contains 6 GB GDDR5 memory. Each experiment was implemented in ExaSlang 4 and compiled to C++/CUDA code for the given architecture. The C++ code was compiled with the GNU g++ compiler, using aggressive optimizations (-O3). The CPU-only experiments ran on the g++ version 6.3. The CUDA code was passed to the nvcc compiler of the CUDA Toolkit 8.0, which depends on the older g++ version 4.9. The code generator was configured to add OpenMP pragmas for parallelization and to emit vectorized code for double-precision computations using AVX intrinsics on the CPU. Other optimizations, such as address precalculation [10] and common subexpression elimination [11], were applied to all versions generated, while a rectangular, spacial tiling was only applied to the 3D versions. If not stated otherwise, we chose a problem size of 8192^2 in 2D and 512^3 in 3D. These are large enough to exceed the cache size, as in real-world applications.

4.2. *Colored Gauss-Seidel Smoother*

We start with a performance evaluation of our layout transformations for a colored Gauss-Seidel kernel. The kernel was run in isolation, not as part of a multigrid application. Each experiment was executed on a single node. We present the performance of the baseline experiments in million lattice updates per second (MLUP/s) and those of additional versions as speedups over the baseline. Both constant-coefficient (cc) versions and variable-coefficient (vc) versions were executed. Constant and variable refer to the multiplicative weights of every element loaded from the field: they are either compile-time constants, or they vary for each grid point and neighbor and are also stored in memory. The problem size of the vc version was reduced to 4096^2 and 256^3, respectively, to comply with the GPU's memory limit. For better comparability, the CPU versions applied the same, reduced problem size.

4.2.1. *Two Colors*

Our first experiments address the performance of a cc and vc RBGS kernel using a 1-st order stencil, as depicted in Figure 6. The ExaSlang 4 code for the kernel in 3D is shown in Figure 2. The variable `Laplace` is a discretized operator and, as such, either a stencil in the cc version or a stencil field in the vc version. An appropriate layout transformation for the former is given in Figure 4 while, for

the latter, the stencil field is transformed in the same way as the other fields. The 2D experiments are analogous. In addition to the performance of a single kernel execution, we evaluated a temporally blocked version of five and four subsequent steps in 2D cc and 3D cc and three, respectively two steps, subsequently in the vc cases. The consequence of a temporal blocking is that subsequent invocations of the loop nest are combined to increase the data locality and reuse data from previous executions, while they are still in the processor's on-chip cache. The basis of this optimization is the polyhedral model [9]. The currently implemented temporal blocking is only useful for CPUs. Custom techniques [12] necessary for GPUs remain future work.

Table 1. Achieved Speedup for a 2D and 3D Red-Black Gauss-Seidel Kernel.

		2D cc	3D cc	2D vc	3D vc
CPU	base [MLUP/s]	921 (91%)	902 (89%)	376 (99%)	293 (97%)
	color splitting	1.38× (84%)	1.42× (85%)	1.61× (90%)	1.80× (96%)
	temporal blocking	3.73×	2.93×	2.35×	2.24×
	both	4.08×	4.98×	2.39×	2.56×
GPU	base [MLUP/s]	4654	3593	1739	1286
	color splitting	1.86×	1.80×	1.75×	1.79×

Table 1 summarizes the results of all RBGS kernel experiments. The values in parentheses for the base and its color-splitting version represent the fraction of the roofline performance, which is based on the measured memory bandwidth of 48.5 GB/s using a streaming benchmark. For the base cc versions in both 2D and 3D, the computation of one new value, i.e., one lattice update (LUP), requires six double-precision values to be transferred between the CPU and main memory. Three are required by the computation: one element is loaded from both the `Solution` and the `RHS` fields (all others are still in the processor's cache) and one updated value is written to the memory. The remaining three are due to the inefficient memory layout: their direct neighbors have to be transferred, too. Therefore, the roofline is:

$$\frac{48493 \text{ MB/s}}{(3 \cdot 2) \cdot 8 \text{ B/LUP}} = 1010 \text{ MLUP/s}$$

A color splitting reduces the number of transferred elements to one read from both fields and one store, as well as a write-allocate for the store, which results in 1515 MLUP/s. The write-allocate is required, since the hardware has to load a cache line from the memory before it can be modified. Without color splitting, this additional load has already been performed since the direct neighbors are required to compute the new value and data is loaded in chunks of 8 double precision values. A temporal blocking was able to reduce the bandwidth requirements even further, which leads directly to a higher performance. However, the resulting code becomes very complex and we are no longer able to provide a meaningful roofline.

```
Function Smoother {
 color with {
  i0 % 2,
  i1 % 2,
  loop over Solution {
   Solution = Solution +
    1.0/diag(LaplaceDense) *
    (RHS-LaplaceDense*Solution)
} } }
```

(a) Smoother Code

```
LayoutTransformations {
 transform Solution and RHS
  with [x,y] => [x/2,y/2,x%2,y%2]
}
```

(b) Transformation for a single time step

```
LayoutTransformations {
 transform Solution and RHS
  with [x,y] => [x/2,y,x%2]
}
```

(c) Transformation for temporal blocking

Fig. 7. ExaSlang 4 code for a colored 9-point Gauss-Seidel Kernel.

A Gauss-Seidel kernel with variable coefficients instead of constant coefficients has an even higher requirement of memory bandwidth. Not only the field elements but also the coefficient of every neighbor must be loaded from main memory, which requires 5 and 7 additional values from main memory in 2D and 3D. Without a layout transformation, only every other element is needed and, thus, twice as much data has to be loaded. The performance results confirm that a color splitting can be more effective for variable coefficients than for constant coefficients.

4.2.2. *Multiple Colors*

Besides the 1st-order stencils, we also evaluated dense constant-coefficient versions. These stencils access the neighbors not only along the axes, but also along the diagonals. This results in a 9-point stencil in 2D and a 27-point stencil in 3D as presented in Figure 6. Figure 7 shows the ExaSlang 4 code for the 2D smoother and the layout transformations that employ a colored 9-point stencil. Two different layout transformations are given, since the one in Figure 7(c) performs better with temporal blocking. Evaluating both requires the adaptation of the presented transformation statement only; no additional modifications are necessary. Four different colors are required here for an easy parallelization. A 3D version of a 27-point stencil with eight colors is straight-forward. Note that the actual `loop over Solution` statement is taken over from the RBGS code. Only the coloring scheme and the stencil are different. The definition of the latter (namely `LaplaceDense`) is not shown here; it is a simple list of nine mappings of the neighboring indices to the corresponding constant coefficients.

The performance results for the multi-color Gauss-Seidel experiments are shown in Table 2. As for the two color versions, the performance of a single smoothing step both with and without color splitting is very close to the expected performance based on the memory bandwidth. However, in 2D, color splitting reduces the performance of temporal blocking. The deterioration could be due to missing optimizations or code simplifications, that are of no consequence for only two colors. But, due to the

Table 2. Achieved Speedup for a 2D Four-Color and a 3D Eight-Color Gauss-Seidel Kernel.

		2D cc	3D cc
CPU	base [MLUP/s]	722 (95%)	457 (90%)
	color splitting	1.36× (97%)	1.29× (97%)
	temporal blocking	2.07×	1.60×
	both	1.89×	2.25×
GPU	base [MLUP/s]	2801	1043
	color splitting	1.76×	1.69×

simple and fast evaluation of various layout transformations, one can switch easily back to the faster version without losing valuable development time.

4.3. *Multigrid Solvers*

The previous experiments evaluated the performance impact of color splitting for the smoother component of a multigrid application. However, there are other multigrid components that are also affected by a layout transformation. To demonstrate the applicability and benefit of color splitting for a complete multigrid application, we conducted experiments for two different applications. The first is the well-known model problem given by Poisson's equations. The second is a multigrid version of an optical flow detection method [7]. For both applications, a shared-memory OpenMP or CUDA version running on a single CPU or GPU and a hybrid OpenMP/MPI or CUDA/MPI parallel version for 12 nodes were generated. Note that the GeForce TITAN Black does not support GPUDirect and, thus, the MPI communication has to be performed by the CPU. I.e., both sender and receiver must transfer the communicated data between the host and the device memory, which imposes an additional communication overhead. In contrast to the previous Gauss-Seidel experiments, a temporal blocking was not applied here since it currently cannot be used in conjunction with colored kernels and MPI communication, independently of any layout transformation. Except for the layout transformations block, the ExaSlang 4 codes of all versions are identical, no matter whether OpenMP or CUDA code with or without MPI is generated. Thus, there is no copy kernel to change the memory layout dynamically and the kernels for the restriction or interpolation are also not modified. Consequently, these kernels access elements of both colors, regardless of where they are stored.

In the case of the baseline experiments, we give the performance of the multigrid solvers as the total time required by the complete solver. Performance improvements of the layout transformations are again stated as speedups over the baseline. Thus, the inverse of the speedup is the fraction of the execution time over the base.

4.3.1. *Poisson*

Different versions of 2D and 3D multigrid solvers for the Poisson equation were automatically generated, based on a finite-difference discretization of the equation

on $(0,1)^d$. The V-cycles had three pre- and three post-smoothing RBGS steps with constant coefficients. Color splitting, as presented in Figure 4, was only applied at the three finest levels. At coarser levels, color splitting is not recommended since the fields fit entirely into cache. This means its benefit vanishes while the drawback of more complicated address computations still persists. Termination occurs when the L_2 norm of the initial residual has been reduced by a factor of 10^5. In all 2D experiments, three V-cycles are executed while, in 3D, four are required.

Table 3. Achieved Speedup of a 2D and 3D Solver for Poisson's Equation.

		2D		3D	
		1 Node	12 Nodes	1 Node	12 Nodes
CPU	base [ms]	2198	2407	5247	7787
	color splitting	1.35×	1.30×	1.35×	1.18×
GPU	base [ms]	552	910	1567	5153
	color splitting	1.32×	1.18×	1.30×	1.08×

Performance results of these experiments are presented in Table 3. In all experiments, the storage of the red and black colored elements in distinct memory locations increased the performance. The improvement of the MPI-parallel version in 3D is slightly lower than of the others. This can be explained by the much higher surface-to-volume ratio of a 3D cube compared to a 2D square and, therefore, the higher communication volume in 3D. The improvements of the CUDA/MPI versions are also less, since the communication is more costly and the computation is faster than on the CPU.

4.3.2. *Optical Flow Detection*

A second, more complex multigrid solver is for an approximation of the optical flow [13]. Here, an approximate motion within an image sequence is computed based on gray value resp. color intensity changes. Concerning the problem size, roughly 17 million pixel per image were chosen and the termination criterion for the solver was the same as in the previous application. This results in 4 V-cycles for the 3D version, running on a single node, and 5 V-cycles for the others. Details can be found in the literature, on this application and an earlier version of the implementation [7] as well as on multigrid methods for optical flow [14, 15].

Since the motion, or flow, of a single pixel consists of multiple components, namely one per dimension, the element type of the corresponding field is a vector, not a scalar. This increases the dimensionality of the entire field but, as explained in Section 3.1, a layout transformation statement may ignore these additional dimensions if they should not be modified.

The layout transformations evaluated for the 3D optical flow computation are shown in Figure 8. In 3D, it pays to concatenate some of the helper fields to enable

```
LayoutTransformations {
  concat @finest Ix, Iy, Iz, It into I
  concat IxIx, IxIy, IxIz, IyIy, IyIz, IzIz into II
  transform I@finest, II@((finest-1) to finest),
            rhs@((finest-1) to finest), flow@((finest-1) to finest)
    with [x,y,z] => [x/2,y,z,(x+y+z)%2]
  transform residual, cgTmp0, cgTmp1, II@(0 to (finest-2)),
            rhs@(0 to (finest-2)), flow@(0 to (finest-2))
    with [x,y,z,v] => [v,x,y,z]
}
```

Fig. 8. Layout transformations for the optical flow computation.

an SoA-to-AoS transformation, as shown in the second `concat` statement. The first `concat` does not affect the performance of the generated code, but it allows a more compact transformation statement. Its individual parts must not be listed separately to perform a color splitting. In general, the RBGS smoother allows for two mutually exclusive field layout schemes to increase the performance:

(i) split by color and place the different components of the higher-order elements in distinct memory locations,

(ii) do not split by color but store the higher-order elements together in memory.

The former increases data locality for the smoother code and allows it to be vectorized. Other, not colored parts suffer from a more complex address computation. The latter provides a simpler memory layout and an increased data locality for those not colored parts. For the finer levels, the performance of the smoother is the dominant factor and, hence, layout scheme (i) is advisable. At coarser levels, larger parts of the fields fit into cache and the effort of the address computation becomes more relevant: layout scheme (ii) is preferable.

The first transformation statement applies a two-color split to field `I` and to the two finest levels of fields `II`, `rhs`, and `flow`. Note that these fields refer to 4D data structures: the first three dimensions (`x`, `y`, and `z`) correspond to the problem dimensions and the fourth (not named) to the vector elements. Since the dimensions added by higher-order data types, such as vectors or matrices, are appended rightmost/outermost, this corresponds to layout scheme (i). The second transformation statement permutes the vector dimension innermost for some other fields and coarser levels of `II`, `rhs`, and `flow`, i.e., an SoA-to-AoS transformation is applied and layout scheme (ii) takes hold.

Table 4 presents the speedup achieved by the layout transformations of the different versions. The results are similar to those of the Poisson experiments and, thus, as expected. The layout transformations in this experiment may be more complex and not obvious but, in contrast to a manual integration, evaluating several different versions is as simple as writing very few lines that specify the transformation and execute our code generator.

Table 4. Achieved Speedup for a 2D and 3D Solver for an Optical Flow Simulation.

		2D		3D	
		1 Node	12 Nodes	1 Node	12 Nodes
CPU	base [ms]	2688	3054	3435	6432
	layout transformations	1.38×	1.26×	1.33×	1.20×
GPU	base [ms]	694	1242	1027	3636
	layout transformations	1.27×	1.15×	1.30×	1.08×

5. Related Work

The use of data layout transformations for performance optimization is not new. O'Boyle and Knijnenburg [16] present a very detailed low-level application of layout transformations. These are represented by nonsingular matrices and, thus, enable only dimensionality-preserving transformations. This excludes the kinds of color splitting that we present and use. Clauss et. al. [17] try to optimize the spacial locality by providing a new array reference function to the compiler. Array elements are ordered in the order in which they are accessed during the execution of the loop nest. This approach works perfectly if every element is accessed only once. In case of a later reuse, the address computation can become very complex.

Layout transformations similar to the ones in our examples are part of existing libraries and frameworks, such as Kokkos [18] or YASK [19]. Kokkos, for example, is a C++ performance portability programming ecosystem. It relies heavily on C++ template programming to generate optimized kernels for different hardware architectures, including GPUs. Like other code generation approaches, it supports the choice of an appropriate layout for a given hardware. However, if the required layout is not yet available, a custom-implementation is necessary. In contrast, ExaStencils facilitates such extensions with the support of generic layout transformations expressed via a single linear transformation. Moreover, Kokkos is focused on parallelization within one MPI rank and, in consequence, the synchronization of data between ranks is still the user's responsibility. Thus, when modifying the memory layout, other parts of the code might require an according adaptation, e.g., the incorporation of MPI data types. ExaStencils does not have this problem since it generates code for data exchange via MPI as well which, thus, can be adapted to the optimized layout automatically.

There are also several approaches to the automatic computation of a suitable data layout transformation, either directly [20, 17, 21, 22, 16] or in combination with a loop transformation [23, 24, 25]. We have not done so, but it would be possible to add one or more of these techniques to our code generator.

6. Conclusions

We offer a new feature of our DSL ExaSlang and the corresponding code generator: a set of layout transformation statements for a fast and easy modification of data layouts. With the help of these statements, the laborious and error-prone modification

of every access to a data field, including its initialization, is done automatically by the ExaStencils code generator. One use case is, e.g., a simple RBGS smoother, which is very profitable because of its parallelizability and numerical properties. Without any modifications, it accesses only every other data element, which complicates the use of vector units provided by modern processor architectures and also wastes memory bandwidth, which tends to be the most critical resource. We demonstrated that these problems can be solved by adding only one single transformation statement. And even for larger applications only few additional statements are required, which enables the testing of several different layout schemes with little effort.

To ease the implementation of such layout transformation statements in other tools and code generators, we offered a fairly detailed insight into how these new statements are applied by the ExaStencils code generator. Since some of the concepts required by automatic layout transformations and for polyhedral compilation are identical or at least similar, the isl, a library actually developed to support polyhedral compilation, is of great help.

7. Acknowledgments

This work has been part of the DFG Priority Programme 1648 "Software for Exascale Computing" in project ExaStencils under contracts RU 422/15 and LE 912/15.

References

[1] Wolfgang Hackbusch. *Multi-Grid Methods and Applications*. Springer-Verlag, 1985.
[2] Ulrich Trottenberg, Cornelius W. Osterlee, and Anton Schüller. *Multigrid*. Academic Press, 2000.
[3] Harald Köstler, Markus Stürmer, and Thomas Pohl. Performance engineering to achieve real-time high dynamic range imaging. *J. Real-Time Image Processing*, 11(1):127–139, 2016.
[4] Markus Kowarschik, Ulrich Rüde, Christian Weiss, and Wolfgang Karl. Cache-aware multigrid methods for solving Poisson's equation in two dimensions. *Computing*, 64(4):381–399, 2000.
[5] Christian Lengauer et al. ExaStencils: Advanced stencil-code engineering. In Luis Lopes et al., editors, *Euro-Par 2014: Parallel Processing Workshops, Part II*, volume 8806 of *Lecture Notes in Computer Science*, pages 553–564. Springer, 2014.
[6] Christian Schmitt, Sebastian Kuckuk, Frank Hannig, Harald Köstler, and Jürgen Teich. ExaSlang: A domain-specific language for highly scalable multigrid solvers. In *Proc. 4th Int. Workshop on Domain-Specific Languages and High-Level Frameworks for High Performance Computing (WOLFHPC)*, pages 42–51. ACM, November 2014.
[7] Christian Schmitt, Sebastian Kuckuk, Frank Hannig, Jürgen Teich, Harald Köstler, Ulrich Rüde, and Christian Lengauer. Systems of partial differential equations in exaslang. In Hans-Joachim Bungartz, Philipp Neumann, and Wolfgang E. Nagel, editors, *Software for Exascale Computing – SPPEXA 2013-2015*, volume 113 of *Lecture Notes in Computational Science and Engineering*, pages 47–67. Springer, 2016.
[8] Sven Verdoolaege. *isl*: An integer set library for the polyhedral model. In Komei Fukuda, Joris van der Hoeven, Michael Joswig, and Mobuki Takayama, editors, *Mathematical Software (ICMS 2010)*, LNCS 6327, pages 299–302. Springer, 2010.

[9] P. Feautrier and C. Lengauer. Polyhedron model. In D. Padua et al., editors, *Encyclopedia of Parallel Computing*, pages 1581–1592. Springer, September 2011.

[10] Stefan Kronawitter and Christian Lengauer. Optimizations applied by the ExaStencils code generator. Technical Report MIP-1502, Faculty of Computer Science and Mathematics, University of Passau, January 2015. 10 pages.

[11] Stefan Kronawitter, Sebastian Kuckuk, and Christian Lengauer. Redundancy elimination in the ExaStencils code generator. In Jesus Carretero et al., editors, *Algorithms and Architectures for Parallel Processing (ICA3PP), Collocated Workshops*, volume 10049 of *Lecture Notes in Computer Science*, pages 159–173. Springer, 2016. First Int. Workshop on Data Locality in Modern Computing Systems (DLMCS).

[12] Tobias Grosser, Albert Cohen, Justin Holewinski, P. Sadayappan, and Sven Verdoolaege. Hybrid hexagonal / classical tiling for GPUs. In *Proc. 12th Int. Symp. on Code Generation and Optimization (CGO)*. ACM, 2014.

[13] Berthold KP Horn and Brian G Schunck. Determining optical flow. *Artificial Intelligence*, 17(1–3):185–203, 1981.

[14] Andrés Bruhn, Joachim Weickert, Christian Feddern, Timo Kohlberger, and Christoph Schnorr. Variational optical flow computation in real time. *IEEE Trans. on Image Processing (TIP)*, 14(5):608–615, 2005.

[15] El Mostafa Kalmoun, Harald Köstler, and Ulrich Rüde. 3D optical flow computation using a parallel variational multigrid scheme with application to cardiac c-arm CT motion. *Image and Vision Computing*, 25(9):1482–1494, 2007.

[16] Michael F. P. O'Boyle and Peter M. W. Knijnenburg. Nonsingular data transformations: Definition, validity, and applications. *Int. J. Parallel Programming*, 27(3):131–159, June 1999.

[17] Philippe Clauss and Benoit Meister. Automatic memory layout transformations to optimize spatial locality in parameterized loop nests. *SIGARCH Computer Architecture News*, 28:11–19, March 2000.

[18] H. Carter Edwards, Christian R. Trott, and Daniel Sunderland. Kokkos: Enabling manycore performance portability through polymorphic memory access patterns. *J. Parallel and Distributed Computing*, 74(12):3202–3216, 2014.

[19] Charles Yount, Josh Tobin, Alexander Breuer, and Alejandro Duran. YASK - yet another stencil kernel: A framework for HPC stencil code-generation and tuning. In *Proc. 6th Int. Workshop on Domain-Specific Languages and High-Level Frameworks for High Performance Computing (WOLFHPC)*, pages 30–39. IEEE Press, 2016.

[20] G. Chen, M. Kandemir, and M. Karakoy. A constraint network-based approach to memory layout optimization. In *Proc. Conf. on Design, Automation and Test in Europe (DATE)*, volume 2, pages 1156–1161. IEEE Computer Society, 2005.

[21] Shun-tak Albert Leung. *Array Restructuring for Cache Locality*. PhD thesis, University of Washington, Department of Computer Science & Engineering, 1996.

[22] Qingda Lu, Xiaoyang Gao, Sriram Krishnamoorthy, Gerald Baumgartner, J. Ramanujam, and P. Sadayappan. Empirical performance model-driven data layout optimization and library call selection for tensor contraction expressions. *J. Parallel and Distributed Computing (JPDC)*, 72(3):338 – 352, 2012.

[23] Philippe Clauss, Vincent Loechner, and Benoit Meister. Minimizing strides in loops with affine array references. In *Proc. Compilers for Parallel Computers (CPC)*, June 2001. 12 pp.

[24] Vincent Loechner, Benoit Meister, and Philippe Clauss. Precise data locality optimization of nested loops. *J. Supercomputing*, 21:37–76, January 2002.

[25] Ozcan Ozturk. Data locality and parallelism optimization using a constraint-based approach. *J. Parallel and Distributed Computing*, 71(2):280–287, 2011.