CFDlang: High-level code generation for high-order methods in fluid dynamics

Norman A. Rink
Immo Huismann
Technische Universität Dresden
Germany

Adilla Susungi
MINES ParisTech
PSL Research University
France

Jeronimo Castrillon
Technische Universität Dresden
Germany

Jörg Stiller
Jochen Fröhlich
Technische Universität Dresden
Germany

Claude Tadonki
MINES ParisTech
PSL Research University
France

Abstract

Numerical simulations continue to enable fast and enormous progress in science and engineering. Writing efficient numerical codes is a difficult challenge that encompasses a variety of tasks from designing the right algorithms to exploiting the full potential of a platform's architecture. Domain-specific languages (DSLs) can ease these tasks by offering the right abstractions for expressing numerical problems. With the aid of domain knowledge, efficient code can then be generated automatically from abstract expressions. In this work, we present the CFDlang DSL for expressing tensor operations that constitute the performance-critical code sections in a class of real numerical applications from fluid dynamics. We demonstrate that CFDlang can be used to generate code automatically that performs as well, if not better, than carefully hand-optimized code.

Keywords language (DSL) design, code generation and optimization, tensor expressions, numerical methods, computational fluid dynamics (CFD)

ACM Reference Format:
Norman A. Rink, Immo Huismann, Adilla Susungi, Jeronimo Castrillon, Jörg Stiller, Jochen Fröhlich, and Claude Tadonki. 2018. CFDlang: High-level code generation for high-order methods in fluid dynamics. In *RWDSL2018: Real World Domain Specific Languages Workshop 2018, February 24, 2018, Vienna, Austria.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3183895.3183900

1 Introduction

For many decades, numerous engineering and scientific disciplines have benefited from large-scale simulations of complex problems that are intractable without numerical methods. Due to large data sizes, run-times of several weeks or months are not unusual for numerical applications, even when executed on massive clusters of powerful modern computers. Therefore, developers and maintainers of numerical applications afford vast amounts of time and energy to code optimization: increasing the speed of an application by only a small fraction leads to large absolute savings in terms of run-time and energy consumption.

The process of optimizing numerical applications can be divided into two steps. (1) An expert in numerical methods selects algorithms and data structures that are best suited to the problem at hand. Some implementation aspects may already be fixed by the domain expert, with a view towards the platform that is targeted for executing the final application. (2) During implementation, appropriate tools, e.g. libraries and compilers, are picked that are optimized for the target platform. An optimization expert who is familiar with the target platform may assist in this step. Numerical codes often have long life-times of several decades, and step (2) must be repeated every time an application is migrated to a new platform. Domain-specific languages (DSLs) can decouple the two steps, and thus act as an interface between the expert in numerical methods and the optimization expert.

In this paper we present CFDlang, a DSL designed for expressing and optimizing performance-critical operations in fluid dynamics simulations. The applications of fluid dynamics are manifold, ranging from weather and climate simulation to the engineering of vehicles and aircraft. The CFDlang DSL specifically aims to assist in the development and optimization of numerical applications that rely on so-called high-order methods (Section 2), which are attractive since they can efficiently reduce numerical errors [10]. High-order methods introduce data structures that are higher-dimensional: instead of vectors and matrices, the use cases for CFDlang operate on tensors of generally more than two dimensions. The fundamental operation between tensors is *contraction*, which is the natural generalization of matrix multiplication. In CFDlang, tensor contraction can be expressed at a high level of abstraction, using a notation that does not unnecessarily clutter tensor expressions with loops and indices (Section 3). This significantly simplifies implementation and thus also assists numerical experts in carrying out step (1) by allowing them to focus more on the relevant algorithms. CFDlang generates efficient code for tensor contraction (also Section 3), thereby addressing step (2). Our evaluation shows that the generated code performs at least as well, and often better, than code that has been painstakingly optimized manually (Section 4).

In summary, the key features of CFDlang are as follows:

(i) The ability to express operations between tensors in an index-free fashion, at a high level of abstraction.
(ii) The capability to generate highly efficient codes, on or above par with hand-optimized solutions.
(iii) Localized and "drop-in" replacement for performance-critical code sections in existing numerical applications that are written in Fortran or C/C++.

The last point makes CFDlang particularly attractive to maintainers of large and complex existing applications. Rather than re-writing big parts of an application in a new DSL, developers can concentrate on those code sections that they know to be the key performance bottlenecks, which often consist of only a few loop nests. This differentiates CFDlang from other solutions that require that an entire numerical problem be re-formulated in a new DSL [4, 13, 24].

In recent years, the manipulation of high-dimensional data, often with tensorial structure, has featured prominently in application domains such as machine learning, image processing, and large-scale stochastic automata networks [22]. The goals of existing frameworks and DSLs [3, 6] designed for the machine learning domain are similar to those of CFDlang. While the design of the CFDlang DSL is driven by its key use cases, we believe that CFDlang may be more widely applicable to codes that manipulate high-dimensional data with tensorial structure.

2 Background and problem definition

The flow of fluids is governed by the Navier-Stokes equations (NSE), which are coupled partial differential equations in space and time [15]. No method is known for finding analytical solutions of the NSE, and computational fluid dynamics (CFD) is the discipline of studying the NSE with numerical methods. Calculating numerical solutions is extremely expensive. For example, the simulation of the flow in Figure 1 requires many thousands of CPU hours.

Numerical solution methods decompose the computational domain Ω into n_e small finite volume *elements* Ω_e of widths h_e. The numerical error ε decreases with h_e. Traditional methods typically approximate the solution on Ω_e with linear functions, i.e. polynomials of order $p = 1$, leading to the error bound $\|\varepsilon\| \lesssim \max_e h_e^2$. Current research in CFD focuses on so-called high-order methods, e.g. the spectral-element method (SEM) or discontinuous Galerkin methods, which can decrease the numerical error at lower costs than more traditional methods. In high-order methods, the solution of a problem is approximated with polynomials of order $p > 1$, resulting in the error bound $\|\varepsilon\| \lesssim \max_e h_e^q$, where either $q = p$ or $q = p + 1$ [10, 12].

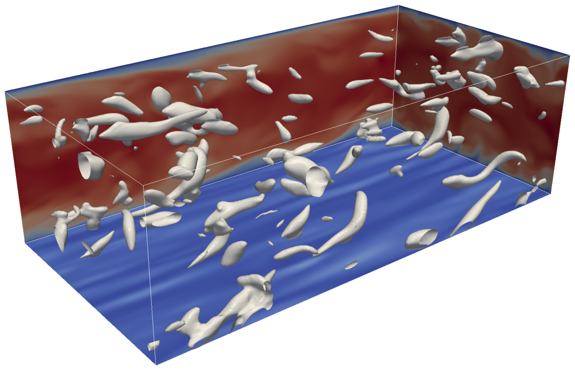


Figure 1. Turbulent flow in a plane channel, main flow direction from left to right. Gray surfaces enclose vortex cores, i.e. centers of large-scale turbulence. Colors on the bounding box encode the speed of the flow; red is fast, blue is slow.

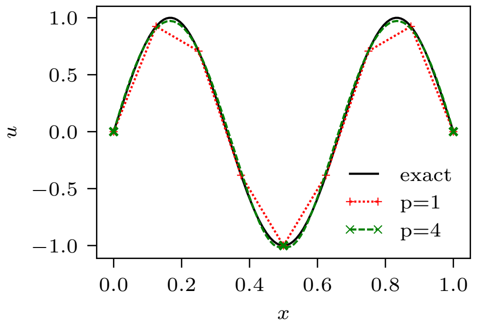


Figure 2. Function $u(x) = \sin(3\pi x)$ approximated with a linear method ($p = 1$) using $n_e = 8$ elements, and a high-order method ($p = 4$) using $n_e = 2$ elements.

For a 1-dimensional problem, the total number of degrees of freedom in a numerical method is in $O(n_e p)$ and determines the computational cost of the method. Figure 2 illustrates how a high-order method based on polynomials of order $p = 4$ achieves a lower approximation error than a linear method, with $p = 1$, while using the same total number of degrees of freedom.

Many of the operators that occur in CFD are linear and local, so that they can be evaluated on an element-by-element basis [8]. In three dimensions, the numerical solution inside an element Ω_e is determined by the coefficients $v_{ijk,e}$, where the indices $i, j, k = 0, \ldots, p$ correspond to the spatial dimensions, and p is the order of the polynomials used to approximate the exact solution. Thus, for a fixed element index e, the coefficients $v_{ijk,e}$ are stored in a 3-dimensional array, also referred to as a 3-dimensional *tensor.* The resulting number of degrees of freedom per element is $(p+1)^3$, and, hence, the total number of degrees of freedom is $n_e(p+1)^3$.

Operators relevant to computing numerical solutions often exhibit a structure that reflects the three spatial dimensions, i.e.

$$v_{ijk,e} = \sum_{i'=0}^{p} \sum_{j'=0}^{p} \sum_{k'=0}^{p} A_{kk'} B_{jj'} C_{ii'} u_{i'j'k',e} \,, \tag{1}$$

where the matrices A, B, and C each specify the action of the combined operator in one of the spatial dimensions. A direct implementation of Equation (1) consists of six nested loops, requiring $2 \cdot p^6$ arithmetic operations per element; and an implementation based on a single matrix multiplication carried out by an optimized DGEMM routine also has complexity $O(p^6)$. However, using an instance of *loop-invariant code motion*, as pointed out for example in [21], Equation (1) can be rearranged to

$$v_{ijk,e} = \sum_{k'=0}^{p} A_{kk'} \sum_{j'=0}^{p} B_{jj'} \sum_{i'=0}^{p} C_{ii'} u_{i'j'k',e} \,, \tag{2}$$

which can be evaluated with only $6 \cdot p^4$ operations. A more general treatment of the complexities of formulae like the ones in Equations (1) and (2) appears in [30]. Note that each summation over a pair of indices in Equations (1) and (2) is called a *tensor contraction.* Contractions generalize matrix multiplication to tensors of multiple dimensions, such as the 3-dimensional tensor $u_{i'j'k',e}$.

One of the key features of the CFDlang DSL is that it can automatically perform transformations that reduce the complexity of operators, analogous to going from Equation (1) to (2). To this end, the DSL expresses Equation (1) as

$$v = A \# B \# C \# u \,.\, [[1\ 8]\ [3\ 7]\ [5\ 6]] \,, \tag{3}$$

where $A \# B \# C \# u$ denotes the concatenation of the tensors A, B, C, and u, as it appears under the summations in Equation (1). The concatenation implicitly carries nine indices which are numbered from 0 to 8. The index pairs [1 8], [3 7], and [5 6] that appear after the period (.) in the assignment (3) are to be contracted, i.e. summed over. In generating code for the assignment (3), CFDlang is free to choose the most efficient evaluation order. Note that the syntax of CFDlang is motivated by the compact *tensor product* notation [18] that is commonly used in the CFD domain. In tensor product notation, the assignment (3) is written as

$$\mathbf{v}_e = (\mathbf{A} \otimes \mathbf{B} \otimes \mathbf{C})\ \mathbf{u}_e \,. \tag{4}$$

Two operators that are heavily utilized in CFD applications and that therefore constitute the key use cases for CFDlang are the *interpolation* operator and the *inverse Helmholtz* operator. The *interpolation* operator is defined by

$$\mathbf{v}_e = (\mathbf{A} \otimes \mathbf{A} \otimes \mathbf{A})\ \mathbf{u}_e \,, \tag{5}$$

the CFDlang notation of which is the same as statement (3) with $A = B = C$. The *inverse Helmholtz* operator is

$$\mathbf{v}_e = (\mathbf{S} \otimes \mathbf{S} \otimes \mathbf{S})\mathbf{D}_e^{-1}(\mathbf{S}^T \otimes \mathbf{S}^T \otimes \mathbf{S}^T)\ \mathbf{u}_e \,, \tag{6}$$

where $\mathbf{D}_e$ is a diagonal matrix, and can be expressed in CFDlang as a sequence of three assignments,

$$t = (S \# S \# S \# u \,.\, [[1\ 6]\ [3\ 7]\ [5\ 8]]) \tag{7a}$$
$$r = D\_inv * t \tag{7b}$$
$$v = (S \# S \# S \# r \,.\, [[0\ 6]\ [2\ 7]\ [4\ 8]]) \,. \tag{7c}$$

```
⟨program⟩ ::= ⟨decl⟩* ⟨elem⟩? ⟨stmt⟩*
⟨decl⟩ ::= var ⟨io⟩? ⟨id⟩ : [⟨ilist⟩]
⟨io⟩ ::= input | output
⟨elem⟩ ::= elem [⟨idlist⟩] ⟨int⟩
⟨stmt⟩ ::= ⟨id⟩ = ⟨expr⟩
⟨expr⟩ ::= ⟨term⟩ | ⟨term⟩ (+|-) ⟨expr⟩
⟨term⟩ ::= ⟨factor⟩ | ⟨factor⟩ (*|/) ⟨expr⟩ | ⟨factor⟩ . [⟨plist⟩]
⟨factor⟩ ::= ⟨atom⟩ | ⟨atom⟩ # ⟨factor⟩
⟨atom⟩ ::= ⟨id⟩ | ( ⟨expr⟩ )
⟨ilist⟩ ::= ϵ | ⟨int⟩ ⟨ilist⟩
⟨int⟩ ::= [0-9][0-9]*
⟨plist⟩ ::= ⟨pair⟩ | ⟨pair⟩ ⟨plist⟩
⟨pair⟩ ::= [⟨int⟩ ⟨int⟩]
⟨idlist⟩ ::= ⟨id⟩ | ⟨id⟩ ⟨idlist⟩
⟨id⟩ ::= [a-zA-Z][a-zA-Z0-9]*
```

Figure 3. The CFDlang grammar.

The *interpolation* and *inverse Helmholtz* operators can easily consume up to 90% of the run-time of a CFD application.

Although rearranging the evaluation order of tensor operations can reduce run-time complexity, some difficulties remain. Namely, CFDlang cannot directly benefit from optimized DGEMM implementations since this would require costly transpositions of tenors [7, 26]. Moreover, DGEMM implementations are optimized for multiplication of large matrices, while the tensors that occur in our use cases typically have dimensions determined by $p \in \{1, \ldots, 12\}$.

3 DSL design and implementation

In this section we introduce the CFDlang DSL. After introducing the DSL itself, we explain how executable code generated from the DSL is integrated into a numerical application. We then discuss the high-level code generator of CFDlang.

3.1 The CFD language (CFDlang)

The DSL is intended to replace performance-critical tensor operations in larger numerical applications. Therefore, its design ensures that the relevant tensor operations can be expressed efficiently, in a few lines of DSL code. This is achieved by (a) an index-free notation of tensors and (b) a compact notation for products and contractions.

Figure 3 specifies the grammar of CFDlang. Non-terminals appear in angle brackets and terminal symbols are in bold face. A CFDlang program consists of a list of declarations (*decl*) followed by a list of statements (*stmt*). In between those lists, there may be an optional *element* directive. Statements are assignments of expressions (*expr*) to identifiers (*id*). No control flow constructs are required to express the relevant tensor operations, which simplifies notation considerably and gives the code generator the freedom to organize control flow in adequate and efficient ways. Since CFDlang programs are intended to be short and typically operate on only a few tensors, variables do not need to be introduced at different scopes. Hence, all declared variables are visible globally within a CFDlang program.

3.1.1 Declarations

Declarations start with the keyword **var**, followed by an optional I/O qualifier and the identifier to be declared. An I/O qualifier indicates that the declared variable is used by the ambient numerical application for communicating data into and out of the CFDlang program. Hence, no memory needs to be allocated for this variable inside the CFDlang program; instead, the memory already allocated by the ambient application is used. CFDlang assumes that different input and output variables do not alias. This assumption, also made by Fortran, helps in generating more efficient code.

Variables declared without an I/O qualifier are local to the CFDlang program. Generally, memory must be allocated for such variables, but they may also be optimized away or coalesced with other variables.

Declarations end with the type of the declared tensor. A tensor is a multi-dimensional array, and hence its type is a list of dimensions, which appears inside brackets, after the colon. An empty list inside the brackets declares a scalar.

Since CFDlang is designed for numerical applications, the elements of a tensor are assumed to be double-precision floating-point numbers. However, this standard setting can be changed, even to integer data types, when invoking the code generator. A straightforward extension of the given grammar would allow that names be introduced for types that are frequently used in declarations.

3.1.2 The element directive

A CFDlang program specifies the operations that are to be performed per element Ω_e. Typically, the same program must be executed for every element. The element directive, which starts with the keyword **elem**, indicates that this is indeed the desired behavior. The directive specifies a list of variables, in brackets, that are instantiated for each element, followed by the total number of elements. CFDlang assumes that the operations performed for different elements are fully independent of each other.

3.1.3 Tensor expressions

The usual arithmetic operations (+, −, ∗, /) operate entry-wise on tensors. Multiplication and division are also allowed between a scalar and a tensor of arbitrary dimension.

The key to expressing tensor operations efficiently lies in the interplay between the the hash operator (in the last alternative for *factor*) and the period operator (in the last alternative for *term*). The hash operator concatenates tensors (cf. Section 2), thus forming a bigger tensor from its operands. The tensor resulting from the hash operator has as many dimensions as both operands combined. The period operator applies contractions to its left argument, the indices of which are numbered implicitly from left to right, starting at zero. The second argument to the period operator is a list of pairs, designating the index pairs to be contracted over. The last line of the CFDlang program in Figure 4 demonstrates how one expresses the contractions in the *interpolation* operator, as defined in Equation (5), for $p + 1 = 7$ and $n_e = 216$.

```
var input  A : [7 7]
var input  u : [7 7 7]
var output v : [7 7 7]

elem [u v] 216

v = A # A # A # u . [[1 6] [3 7] [5 8]]
```

Figure 4. DSL code for the *interpolation* operator.

3.2 Integration with the ambient application

CFDlang programs can be compiled into executable code and linked against the ambient numerical application when the application is built. Alternatively, programs can be built dynamically at application run-time. This approach is usually preferred since the dimensions of tensors are not necessarily known until application run-time, and compilers can generate better code once these dimensions have been fixed.

In a typical use case, therefore, the ambient numerical application first assembles the CFDlang program code as a string, essentially by filling in constants for the tensor dimensions. This string is then passed to the CFDlang run-time library in a *build* call, cf. Figure 5, which causes the library to invoke the high-level code generator to turn the CFDlang program into efficient C code. The generated C code is then compiled into executable machine code by running the host system's C compiler. In the CFD domain, the Intel compiler suite is the de-facto standard for applications written

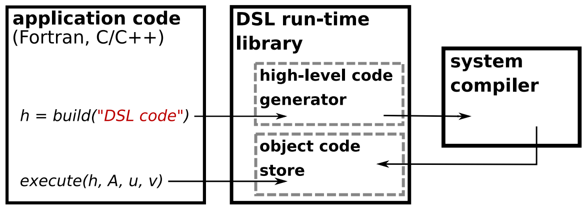


Figure 5. Dynamic code generation for DSL programs.

```
var input  x : [3 4 5]
var input  y : [3 4 5]
var output z : [3 4 5]

z = x * y
```

Figure 6. Entry-wise multiplication of tensors.

in Fortran or C/C++ and targeted at platforms based on Intel hardware. The CFDlang run-time library stores the machine code produced by the system compiler and returns a *handle* to the ambient application. To execute the compiled CFDlang program, the *handle* is used in an *execute* call to the run-time library. In this call, arrays must be supplied as additional arguments, corresponding to the variables declared with I/O qualifiers in the CFDlang program; see again Figure 5 and compare with the variable declarations in Figure 4.

Generating executable code for CFDlang programs at application run-time incurs a time penalty of typically a few seconds. Since it is not unusual that applications consume several thousand CPU hours, a penalty of a few seconds is easily amortized if the generated code reduces the run-time of the performance-critical tensor operations by even just a few percent. That this is indeed the case will be verified in the evaluation of CFDlang in Section 4.

Depending on whether the ambient application is written in Fortran or C/C++, the CFDlang code generator can switch between column-major and row-major array layout.

3.3 Code generation

A straightforward strategy for lowering CFDlang programs to C code would be as follows. For every assignment x = *expr* in the CFDlang program, create a perfect loop nest that iterates over all dimensions of the tensor x. In the body of the loop nest, assign one entry of *expr* to the corresponding entry of the variable x. The CFDlang program in Figure 6 and the corresponding C code in Figure 7 illustrate this. The function `cfd_kernel` implements the entire CFDlang program, and the declarations in the program in Figure 6 are used to construct the signature of `cfd_kernel`. The `restrict` keyword in the function parameters in Figure 7 implements CFDlang's convention that different variables do not alias.

The outlined code generation strategy can be improved to generate C code that, in turn, compiles to more efficient

```c
void cfd_kernel(
  double x[restrict 3][4][5],
  double y[restrict 3][4][5],
  double z[restrict 3][4][5])
{
  for (int i0 = 0; i0 < 3; i0++) {
    for (int i1 = 0; i1 < 4; i1++) {
      for (int i2 = 0; i2 < 5; i2++) {
        z[i0][i1][i2] = x[i0][i1][i2]
                      * y[i0][i1][i2];
} } } }
```

Figure 7. Generated C code for the program in Figure 6.

```c
void cfd_kernel(
  double A[restrict 7][7],
  double u[restrict 216][7][7][7],
  double v[restrict 216][7][7][7])
{
  /* element loop: */
  for(int e = 0; e < 216; e++) {
    for(int i0 = 0; i0 < 7; i0++) {
    for(int j0 = 0; j0 < 7; j0++) {
    for(int k0 = 0; k0 < 7; k0++) {
      v[e][i0][j0][k0] = 0.0;
      for(int i1 = 0; i1 < 7; i1++) {
      for(int j1 = 0; j1 < 7; j1++) {
      for(int k1 = 0; k1 < 7; k1++) {
        v[e][i0][j0][k0] += A[i0][i1]
                          * A[j0][j1]
                          * A[k0][k1]
                          * u[e][i1][j1][k1];
    } } } } } }
  } /* end of element loop */
}
```

Figure 8. Naive C code for the *interpolation* operator.

executable machine code. The following sections describe a number of improvements.

3.3.1 Algorithmic transformations

High-order methods in fluid dynamics crucially rely on the tensor contraction operation. For example, the last line of the program in Figure 4 uses multiple contractions to express the *interpolation* operator. A naive translation of this CFDlang program produces the C code in Figure 8. The C code also illustrates how an *element loop* is generated from the element directive in the program in Figure 4. In this case, the element loop iterates over 216 elements.

The implementation of the `cfd_kernel` function in Figure 8 corresponds to the evaluation order in Equation (1). Thus, the run-time complexity per element is $O(p^6)$, with $p + 1 = 7$. As explained in the discussion in Section 2, the complexity can be reduced significantly by transforming multiple contractions into a sequence of single contractions.

The multiple contractions in the last line line of Figure 4 can be represented as an un-directed *contraction* graph as follows. For each tensor that occurs in the contraction expression, place a node in the contraction graph. The node has as many outgoing legs as the tensor has indices. If a tensor appears in an expression multiple times, place separate nodes in the graph, one for each occurrence of the tensor. Outgoing legs from nodes are joined to form edges if the legs correspond to index pairs that are contracted. Outgoing legs that correspond to indices that are not contracted are left dangling. Figure 9 shows the contraction graph resulting from the last line of Figure 4. C code is generated from the contraction graph by emitting a contraction for one edge at a time. For the *interpolation* operator, this leads to the sequence of contractions shown in Figure 10. Since this corresponds to the evaluation order in Equation (2), the complexity of the program in Figure 10 is now $O(p^4)$.

By transforming multiple contractions into a sequence of single contractions, as described, the CFDlang code generator turns the *interpolation* operator from Figure 4 into the C code that appears between lines 10 and 43 in Figure 11.

The order in which edges of contraction graphs are treated determines the order in which contractions appear in the generated C code, and this order affects the total number of executed arithmetic instructions. In general, the problem of finding the order that minimizes the number of instructions is NP-complete [16]. In the use cases from CFD, the order of contractions does not affect the total number of executed operations. This is because the loop bounds are the same in all loops (other than the element loop), cf. Figures 8 and 11, which in turn is a consequence of choosing the same basis polynomials of order p for all spatial directions.

Transforming multiple contractions into a sequence of single contractions also introduces temporary variables that store intermediate results (e.g. `t6`, `t7` in Figures 10 and 11). Allocating large amounts of memory for storing intermediate results may adversely affect the performance of the memory system, possibly even exhausting local disk space [4]. For general tensor contractions, this has to be taken into account when making predictions about the performance of the generated code. However, in the CFD use cases, $p \in \{1, \dots, 12\}$ and hence the memory required by temporary variables is typically small enough to fit into the lowest-level cache. In fact, all tensors, including temporary ones, that are required for evaluating expressions in one element Ω_e typically fit into the lowest-level cache.

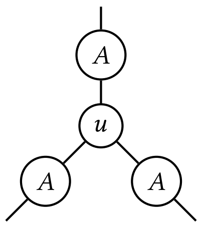


Figure 9. Contraction graph for the *interpolation* operator.

```
t6   = A # u   .  [[1  4]]
t7   = A # t6  .  [[1  4]]
v    = A # t7  .  [[1  4]]
```

Figure 10. Sequence of single contractions.

3.3.2 Parallelization

The CFDlang code generator takes advantage of thread-level parallelism and SIMD parallelism.

Instances of a CFDlang program that are executed for different elements Ω_e are fully independent of each other, cf. Section 3.1.2. Therefore, program instances for different elements can be executed in parallel threads, without any need for synchronization. Thread-parallel code can be generated easily by adding an OpenMP pragma to the element loop, resulting in line 7 in Figure 11. It is assumed that the ambient application is enabled for the use of threads.

The multiple loop nests that occur in the C code generated from CFDlang programs provide opportunities for vectorization with SIMD instructions. By placing SIMD pragmas in front of loops, the CFDlang code generator assists the system C compiler in vectorizing the generated code.[1] In the placement of pragmas, CFDlang can follow two strategies.

The first strategy places a pragma in front of the innermost loop in each 3-level loop nest. This is indicated by the comments in lines 14, 26, and 37 in Figure 11. Vectorizing the loops immediately after these comments should be straightforward. Hence, one would expect good performance from this strategy. However, if the loop bounds in lines 15, 27, and 38 are not multiples of the processor's SIMD width, the compiler will introduce additional instructions immediately after these loops to handle any remaining indices between the last multiple of the SIMD width and the loop bound. Since these additional instructions are nested inside two loops (and the element loop), they are executed many times. Therefore, one would expect this strategy to yield best performance only if the loop bounds are multiples of the SIMD width. By contrast, placing pragmas one level deeper, in front of the contraction loops in lines 17, 29 and 40, would result in under-utilization of the processor's SIMD unit.

The second strategy places a pragma in front of the outermost loop in each 3-level loop nest, as in lines 11, 23, and 34 in Figure 11. This will cause the compiler to unroll nested loops and vectorize the remaining outermost loop. Unrolling replicates instructions in the loop body, potentially giving the compiler more and better opportunities for vectorization. However, loop unrolling can also have negative effects such as increasing register pressure.

[1] In this work, we rely on compiler-specific SIMD pragmas since the Intel compiler suite is the de-facto standard for our use cases. To be more vendor-independent, one could use OpenMP SIMD pragmas instead.

3.3.3 Parameter specialization

Compilers generally produce more efficient machine code for programs whose control flow is statically known. In the context of CFD applications, this means that the dimensions of tensors, which translate into loop bounds, must be known at compile-time. Therefore, the requirement that all tensor dimensions must be specified as explicit constants has been built into CFDlang.

4 Evaluation

In this section we compare the performance of CFDlang-generated code with manually optimized code. The evaluation platform is a 24-core Intel Xeon E5-2680 v3 CPU (Haswell), running at 2.50Ghz. The 24 cores are split evenly between two NUMA nodes, each of which has access to 32GB of main memory. The size of the L1 data cache is 32KB per core, and a single core can reach a maximum performance of 40 GFLOPs. This figure takes into account that (i) there are two execution units per core, (ii) the SIMD width is 4, and (iii) a fused multiply-add instruction executes two operations in one cycle, which is relevant in our context since tensor contraction is a sequence of multiplications and additions. Executable code is generated with the Intel compiler suite (either *icc* or *ifort* as appropriate). Specifically, the Intel Compiler and MKL version 2017.2.174 is used.

4.1 Code variants

The baseline for the performance measurements is a code variant where all loop bounds have been *specialized* to constant values at compile-time. Although implemented in Fortran, the *spezialized* variant is equivalent to the code in Figure 11 without the pragmas.

For comparison we also evaluate a code variant that relies on the DGEMM routine from the Intel MKL library. DGEMM implements a flexible version of matrix multiplication, is considered fast, and is therefore commonly used in numerical applications. However, using DGEMM for the contraction of higher-dimensional tensors incurs a performance penalty due to data transposition, as explained at the end of Section 2.

To obtain *hand-optimized* implementations of the *interpolation* and *inverse Helmholtz* operators, a domain expert has spent several man months profiling and improving the respective Fortran codes. Data sizes are small enough so that all tensors that are processed in a single element fit into the L1 cache. For example, the per-element memory required for any of the 3-dimensional tensors in Figures 4, 10, and 11 is $7^3 \cdot \texttt{sizeof(double)} = 2744$ bytes. Therefore, the latency of memory accesses does not make a dominant contribution to the total latency of the *interpolation* and *inverse Helmholtz* operators. Hence, the number of executed instructions is a good proxy for the total latency. Manual optimization that aims at reducing the number of instructions crucially relies on loop unrolling. By increasing the number of instructions

```c
void cfd_kernel(
  double A[restrict 7][7],
  double u[restrict 216][7][7][7],
  double v[restrict 216][7][7][7])
{
  /* element loop: */
  #pragma omp for
  for (int e = 0; e < 216; e++) {
    double t6[7][7][7];
    /* 1st contraction: */
    #pragma simd
    for (int i0 = 0; i0 < 7; i0++) {
    for (int i1 = 0; i1 < 7; i1++) {
    /* #pragma simd */
    for (int i2 = 0; i2 < 7; i2++) {
      double t8 = 0.0;
      for (int i3 = 0; i3 < 7; i3++)
        t8 += A[i0][i3] * u[e][i1][i2][i3];
      t6[i0][i1][i2] = t8;
    } } } /* end of 1st contraction */
    double t7[7][7][7];
    /* 2nd contraction: */
    #pragma simd
    for (int i4 = 0; i4 < 7; i4++) {
    for (int i5 = 0; i5 < 7; i5++) {
    /* #pragma simd */
    for (int i6 = 0; i6 < 7; i6++) {
      double t9 = 0.0;
      for (int i7 = 0; i7 < 7; i7++)
        t9 += A[i4][i7] * t6[i5][i6][i7];
      t7[i4][i5][i6] = t9;
    } } } /* end of 2nd contraction */
    /* 3rd contraction: */
    #pragma simd
    for (int i8 = 0; i8 < 7; i8++) {
    for (int i9 = 0; i9 < 7; i9++) {
    /* #pragma simd */
    for (int i10 = 0; i10 < 7; i10++) {
      double t10 = 0.0;
      for (int i11 = 0; i11 < 7; i11++)
        t10 += A[i8][i11] * t7[i9][i10][i11];
      v[e][i8][i9][i10] = t10;
    } } } /* end of third contraction */
  } /* end of element loop */
}
```

Figure 11. Generated C code for the *interpolation* operator.

in loop bodies, loop unrolling can positively affect the total number of executed instructions in two ways. First, it improves the ratio of body instructions to instructions that are merely required to manage the loop (e.g. branch instructions). Second, it may present the compiler with additional opportunities for vectorizing loops that it would otherwise miss.

Vectorization can reduce the number of instructions since a single SIMD instruction processes multiple data items.

Manual optimization is a long and tedious process the results of which are very sensitive to the dimensions of the processed data. For our operators, this means that different code variants will lead to best performance for different polynomial orders. In the following, for each polynomial order p we only report the performance of the *hand-optimized* code variant that performed the best for this p.

4.2 Performance results

Figures 12 and 13 show the performance of different code variants for varying polynomial order p. The horizontal axes denote $p + 1$ (rather than p) since for polynomial order p, tensor dimensions take the value $p+1$. In Figures 12a and 13a, performance is measured by the number of updates to entries of the tensor $\mathbf{v}_e$ in Equations (5) and (6) respectively. Updating each entry requires a fixed number of tensor contractions, each of which has complexity $O(p)$. Hence, the graphs in Figures 12a and 13a roughly evolve as $1/p$.

CFDlang-generated code with the innermost placement of SIMD pragmas, i.e. *CFDlang(inner)*, generally performs about as well as the *hand-optimized* code. However, for the *inverse Helmholtz* operator and $p + 1 \geq 6$, the performance of *CFDlang(inner)* stays behind the *hand-optimized* code by an almost constant offset. The CFDlang-generated code with the outermost placement of SIMD pragmas, i.e. *CFDlang(outer)*, generally outperforms all other code variants. One notable exception is when $p + 1 = 4$, i.e. the SIMD width, in which case *CFDlang(inner)* performs the best of all variants by a wide margin. Note also that for the *interpolation* operator, *CFDlang(inner)* indeed performs best when $p+1$ is a multiple of the SIMD width, as anticipated in in Section 3.3.2.

In summary, Figures 12 and 13 show that code generated from the CFDlang DSL can perform at least as well as carefully hand-tuned code, resulting from a time consuming optimization process. It is clear from the different behaviors of the *CFDlang(outer)* and *CFDlang(inner)* code variants that neither currently exploits the full potential of vectorization. A detailed analysis of vectorization opportunities in use cases from the CFD domain is beyond the scope of this work, but this should be looked into in the future.

Finally, note that Figures 12 and 13 correspond to code execution in a single thread on one core. As explained in Section 3.3.2, evaluation of different elements in separate OpenMP threads is straightforward. Performance should scale well with the number of OpenMP threads provided that (i) there are more elements than threads, and that (ii) at any point in time no more than one thread is executing per core. When multiple threads execute on the same core, they share the core's cache and hence cache size will at some point become a limiting factor. A study of the behavior of multiple threads and caches is left for future work.

5 Related work

CFDlang has been designed specifically for CFD applications that employ high-order spectral element methods. Many frameworks and languages exist that targeted different use cases and that vary in their degree of generality. The NumPy library [1], for example, offers completely general functions for manipulating high-dimensional arrays. However, when using library functions to implement individual tensor operations, one cannot utilize the algorithmic transformations employed by CDFlang. Similar statements hold for the Theano [6] and TensorFlow [3] packages. The XLA compiler aims to overcome this but is still experimental [2]. In [29] an intermediate language for tensor operations is proposed, together with meta-programming capabilities for specifying code transformations.

The Tensor Contraction Engine (TCE) [4] generates efficient code for quantum-chemistry applications. Since the tensors involved are large, TCE considers trade-offs between code transformations and the amount of memory allocated for intermediate results. Our CFD use cases rely on comparably small data structures. Hence, the trade-offs considered by TCE are not immediately relevant.

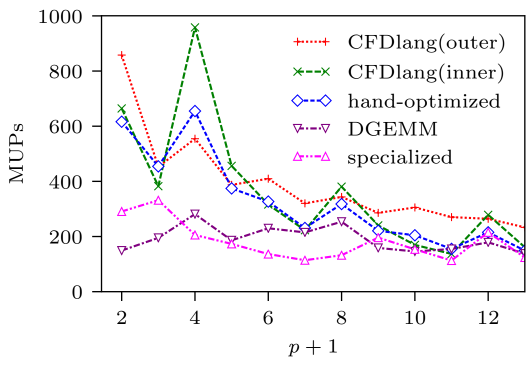
The Tensor Transposition Compiler (TTC) [27] focuses on generating efficient tensor transpositions. Suitably transposing tensors can turn a general contraction into a matrix multiplication, for which one can rely on fast implementations of GEMM-like routines [26], as implemented for CFD in [7, 19]. To some extent, TTC relies on blocking to generate efficient transpositions. This, again, is more important when data structures are larger than in our use cases.

The Tensor Algebra Compiler (TACO) [14] generates efficient code for operations on tensors for which each dimension can have a different data layout, either dense or sparse. This flexibility is not required in the present work since all tensors in the studied CFD use cases are dense.

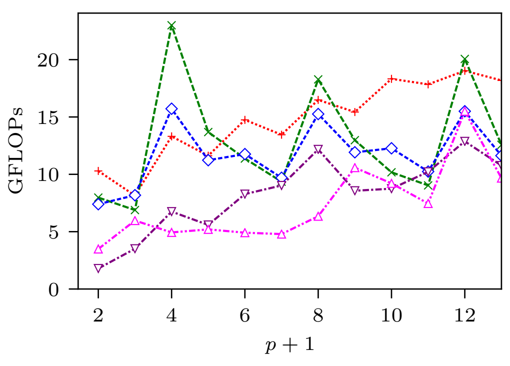
The variational forms compiler from [13] and the Firedrake framework [24] offer complete solutions for implementing classes of CFD codes. This is different from our work, which focuses on providing localized drop-in replacements for the performance-critical loop nests in an application. Thus, CFDlang seems closer in spirit to the COFFEE compiler [17], used internally by the Firedrake framework.

In the OP2 framework one can write hardware-agnostic kernels for CFD applications as well as entire applications using domain decomposition [20]. Optimizations employed by OP2 include loop fusion and fission, which we have not yet explored for CFDlang. Thus far, the OP2 framework has not been applied in the context of high-order methods.

In the Memory Access Pattern Specification (MAPS) framework [5] kernels are annotated with memory access patterns. Optimizations utilize these annotations to achieve better performance than other frameworks. However, MAPS currently

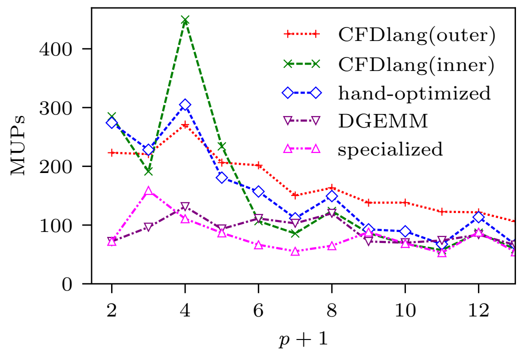


a. Millions of updates per second (MUPs).

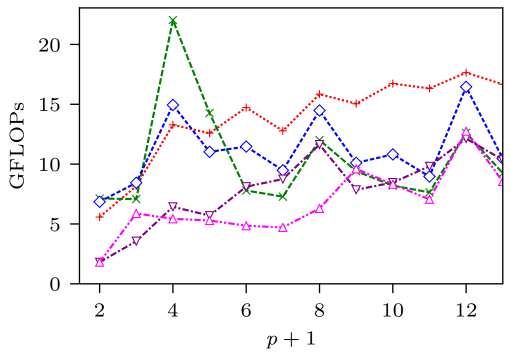


b. GFLOPs (max. 40 GFLOPs).

Figure 12. *Interpolation* operator, single thread performance, $n_e = 6^3 = 216$ elements.



a. Millions of updates per second (MUPs).



b. GFLOPs (max. 40 GFLOPs).

Figure 13. *Inverse Helmholtz* operator, single thread performance, $n_e = 6^3 = 216$ elements.

targets GPUs only, while the present work has focused on improving performance for individual CPU cores.

The SPIRAL project, e.g. [23], has produced a number of DSLs and code generation strategies. Among these, LGen focuses on linear algebra computations and it exploits the structure of matrices to generate efficient code [25].

6 Summary and outlook

We have presented CFDlang, a new DSL for expressing tensor operations that appear in a class of applications from computational fluid dynamics (CFD). From a CFDlang program, executable code is generated that performs as well or better than manually optimized code. This has been demonstrated for two numerical operators that are the key building blocks of real numerical simulations.

To reduce the run-time complexity of generated code, CFDlang transforms multiple tensor contractions into a sequence of single contractions. While finding an optimal contraction sequence is generally NP-complete [16], the situation is simpler for the considered use cases and data sizes. To make CFDlang more widely applicable, future work should study heuristics for handling more general tensor contractions.

The second ingredient in obtaining efficient code from CFDlang is vectorization. By using preprocessor pragmas, the system C compiler has been guided towards two different vectorization strategies. Although both strategies lead to good performance, no one strategy alone can consistently outperform hand-optimized codes. Relying on pragmas for vectorization makes it difficult to understand why the final machine code behaves as observed. This could be resolved by enhancing CFDlang's code generator with its own vectorizer that unrolls loops as necessary and emits architecture-specific SIMD intrinsics into the generated C code. Further motivation for focusing on vectorization in the future can be derived from initial experiments that suggest that considerably better performance can be achieved for polynomial orders $p + 1 = 4, 8, 12$, i.e. multiples of the SIMD width.

Because of the independence of volume elements in the studied use cases, numerical operators can be evaluated for different elements in parallel. The presented code generator for CFDlang exploits this parallelism by emitting OpenMP pragmas to execute code for different elements in parallel threads. A detailed study of the properties and limitations of this thread-level parallelism is left to future work.

The development of CFDlang has been driven by its key use cases from the CFD domain. Further use cases, potentially also from different application domains, could be explored in the future. Recent work suggests that certain problems in machine learning [28] and quantum physics [9, 11] may call for efficient evaluation of tensor contraction. It would be interesting to see if the performance-critical code sections of these problems can also be easily expressed with CFDlang.

Acknowledgments

This work was partially funded by the German Research Council (DFG) through the Cluster of Excellence 'Center for Advancing Electronics Dresden' (cfaed). The authors thank Albert Cohen and P. Sadayappan for useful discussions.

References

[1] 2017. NumPy, package for scientific computing with Python. http://www.numpy.org/. (2017).

[2] 2017. XLA: Accelerated Linear Algebra. https://www.tensorflow.org/performance/xla/. (2017).

[3] M Abadi, P Barham, J Chen, Z Chen, A, J Dean, M Devin, S Ghemawat, G Irving, M Isard, M Kudlur, J Levenberg, R Monga, S Moore, D Gordon Murray, B Steiner, PA Tucker, V Vasudevan, P Warden, M Wicke, Y Yu, and X Zhang. 2016. TensorFlow: A system for large-scale machine learning. *CoRR* abs/1605.08695 (2016).

[4] G Baumgartner, A Auer, DE Bernholdt, A Bibireata, V Choppella, D Cociorva, X Gao, RJ Harrison, S Hirata, S Krishnamoorthy, S Krishnan, C-C Lam, Q Lu, M Nooijen, RM Pitzer, J Ramanujam, P Sadayappan, and A Sibiryakov. 2005. Synthesis of High-Performance Parallel Programs for a Class of ab Initio Quantum Chemistry Models. *Proc. IEEE* 93, 2 (2005), 276–292.

[5] T Ben-Nun, E Levy, A Barak, and E Rubin. 2015. Memory Access Patterns: The Missing Piece of the multi-GPU Puzzle. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC '15)*. 19:1–19:12.

[6] J Bergstra, O Breuleux, F Bastien, P Lamblin, R Pascanu, G Desjardins, J Turian, D Warde-Farley, and Y Bengio. 2010. Theano: a CPU and GPU Math Expression Compiler. In *Proceedings of the 9th Python in Science Conference*. 3–10.

[7] PE Buis and WR Dyksen. 1996. Efficient vector and parallel manipulation of tensor products. *ACM Transactions on Mathematical Software (TOMS)* 22, 1 (1996), 18–23.

[8] CD Cantwell, SJ Sherwin, RM Kirby, and PHJ Kelly. 2011. From h to p efficiently: Strategy selection for operator evaluation on hexahedral and tetrahedral elements. *Computers & Fluids* 43, 1 (2011), 23–28.

[9] G Carleo and M Troyer. 2017. Solving the quantum many-body problem with artificial neural networks. *Science* 355, 6325 (2017), 602–606.

[10] MO Deville, PF Fischer, and EH Mund. 2002. *High-Order Methods for Incompressible Fluid Flow.* Cambridge University Press.

[11] MR Hush. 2017. Machine learning for quantum physics. *Science* 355, 6325 (2017), 580–580.

[12] GE Karniadakis and SJ Sherwin. 2013. *Spectral/hp Element Methods for Computational Fluid Dynamics.* Oxford University Press.

[13] RC Kirby and A Logg. 2006. A Compiler for Variational Forms. *ACM Transactions on Mathematical Software (TOMS)* 32, 3 (2006), 417–444.

[14] F Kjolstad, S Kamil, S Chou, D Lugato, and S Amarasinghe. 2017. The Tensor Algebra Compiler. *Proceedings of the ACM on Programming Languages* 1, OOPSLA (2017), 77:1–77:29.

[15] E Krause. 2005. *Fluid mechanics.* Springer.

[16] C-C Lam, P Sadayappan, and R Wenger. 1997. On Optimizing A Class Of Multi-Dimensional Loops With Reductions For Parallel Execution. *Parallel Processing Letters* 7 (1997), 157–168.

[17] F Luporini, AL Varbanescu, F Rathgeber, G-T Bercea, J Ramanujam, DA Ham, and PHJ Kelly. 2015. Cross-Loop Optimization of Arithmetic Intensity for Finite Element Local Assembly. *ACM Transactions on Architecture and Code Optimization (TACO).* 11, 4 (2015), 57:1–57:25.

[18] RE Lynch, JR Rice, and DH Thomas. 1964. Direct solution of partial difference equations by tensor product methods. *Numer. Math.* 6, 1 (1964), 185–199.

[19] D Moxey, CD Cantwell, RM Kirby, and SJ Sherwin. 2016. Optimising the performance of the spectral/hp element method with collective linear algebra operations. *Computer Methods in Applied Mechanics and Engineering* 310 (2016), 628–645.

[20] GR Mudalige, IZ Reguly, and MB Giles. 2016. Auto-vectorizing a large-scale production unstructured-mesh CFD application. In *Proceedings of the 3rd Workshop on Programming Models for SIMD/Vector Processing (WPMVP '16)*. 5:1–5:8.

[21] KB Ølgaard and GN Wells. 2010. Optimizations for Quadrature Representations of Finite Element Tensors Through Automated Code Generation. *ACM Trans. Math. Software* 37, 1 (2010), 8:1–8:23.

[22] B Plateau and WJ Stewart. 1997. Stochastic Automata Networks. In *Computational Probability.* Kluwer Academic Press, 113–152.

[23] M Püschel, JMF Moura, JR Johnson, D Padua, MM Veloso, BW Singer, J Xiong, F Franchetti, A Gacic, Y Voronenko, K Chen, RW Johnson, and N Rizzolo. 2005. SPIRAL: Code Generation for DSP Transforms. *Proc. IEEE* 93, 2 (2005), 232–275.

[24] F Rathgeber, DA Ham, L Mitchell, M Lange, F Luporini, ATT McRae, G-T Bercea, GR Markall, and PHJ Kelly. 2016. Firedrake: Automating the Finite Element Method by Composing Abstractions. *ACM Trans. Math. Software* 43, 3 (2016), 24:1–24:27.

[25] DG Spampinato and M Püschel. 2016. A Basic Linear Algebra Compiler for Structured Matrices. In *Proceedings of the 2016 International Symposium on Code Generation and Optimization (CGO '16)*. 117–127.

[26] P Springer and P Bientinesi. 2016. Design of a high-performance GEMM-like Tensor-Tensor Multiplication. *CoRR* abs/1607.00145 (2016).

[27] P Springer, JR Hammond, and P Bientinesi. 2017. TTC: A High-Performance Compiler for Tensor Transpositions. *ACM Transactions on Mathematical Software (TOMS)* 44, 2 (2017), 15:1–15:21.

[28] E Stoudenmire and DJ Schwab. 2016. Supervised Learning with Tensor Networks. In *Advances in Neural Information Processing Systems 29*, DD Lee, M Sugiyama, UV Luxburg, I Guyon, and R Garnett (Eds.). 4799–4807.

[29] A Susungi, NA Rink, J Castrillón, I Huismann, A Cohen, C Tadonki, J Stiller, and J Fröhlich. 2017. Towards Compositional and Generative Tensor Optimizations. In *Proceedings of the 16th ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences (GPCE 2017)*. 169–175.

[30] C Tadonki and B Philippe. 2001. Parallel Multiplication of a Vector by a Kronecker Product of Matrices. In *Parallel Numerical Linear Algebra*, J Dongarra and EJ Kontoghiorghes (Eds.). 71–89.