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

Norman A. Rink
Jeronimo Castrillon

Chair for Compiler Construction
Technische Universität Dresden
Germany

Immo Huismann
Jörg Stiller
Jochen Fröhlich

Chair of Fluid Mechanics
Technische Universität Dresden
Germany

Adilla Susungi
Claude Tadonki

MINES ParisTech
PSL Research University
France

Outline

1. Background and motivation
2. The CFDlang domain-specific language
   1. Language definition
   2. Code generation
3. Evaluation of CFDlang-generated code performance
4. Summary and outlook

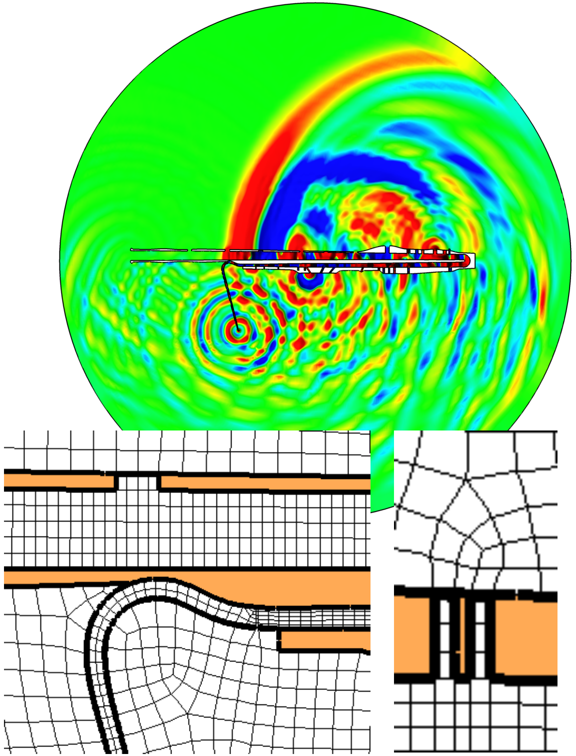
Outline

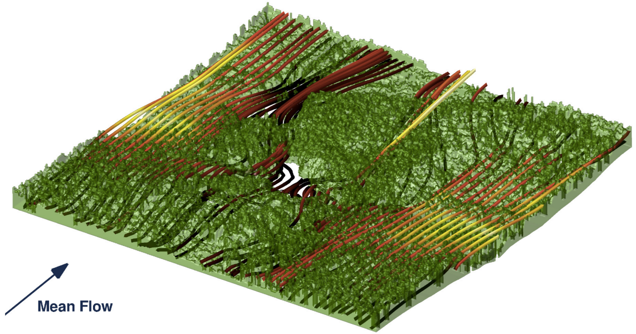
1. **Background and motivation**
2. The CFDlang domain-specific language
    1. Language definition
    2. Code generation
3. Evaluation of CFDlang-generated code performance
4. Summary and outlook

Why fluid dynamics?

- Design and engineering
  - Vehicles, aircraft etc.
  - Alternative to costly experiments, e.g. in wind channels.

Graphic omitted for copyright reasons.

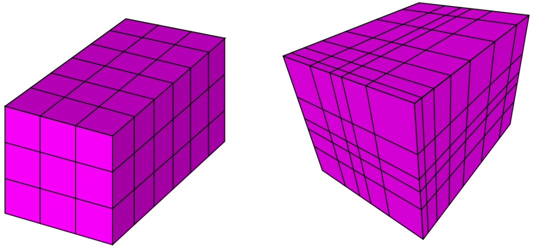




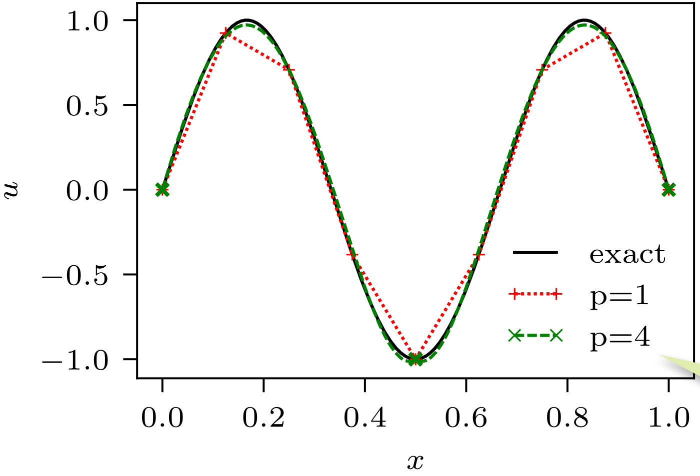
- Weather and climate simulation
  - Daily forecasts.
  - Natural disasters.

High-order methods and tensors (1/2)

- Numerical methods
  - Used to study problems described by (otherwise) **intractable partial differential equations.**
  - Compute approximate solutions or simulations.



https://en.wikipedia.org/wiki/Regular_grid



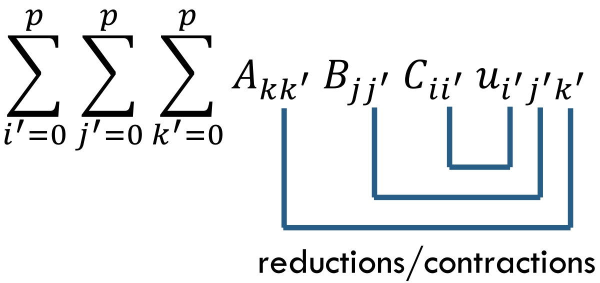
- Fluid dynamics and high-order methods
  - Fluid flows governed by the **Navier-Stokes equations.**
  - Subdivide volume of interest into volume **elements $\mathbf{\Omega_e}$.**
  - Approximate solutions with polynomials of degree p:

$$u(x) = u_p \cdot x^p + u_{p-1} \cdot x^{p-1} + \cdots + u_1 \cdot x + u_0$$

High-order method: higher accuracy at the same computational complexity.

High-order methods and tensors (2/2)

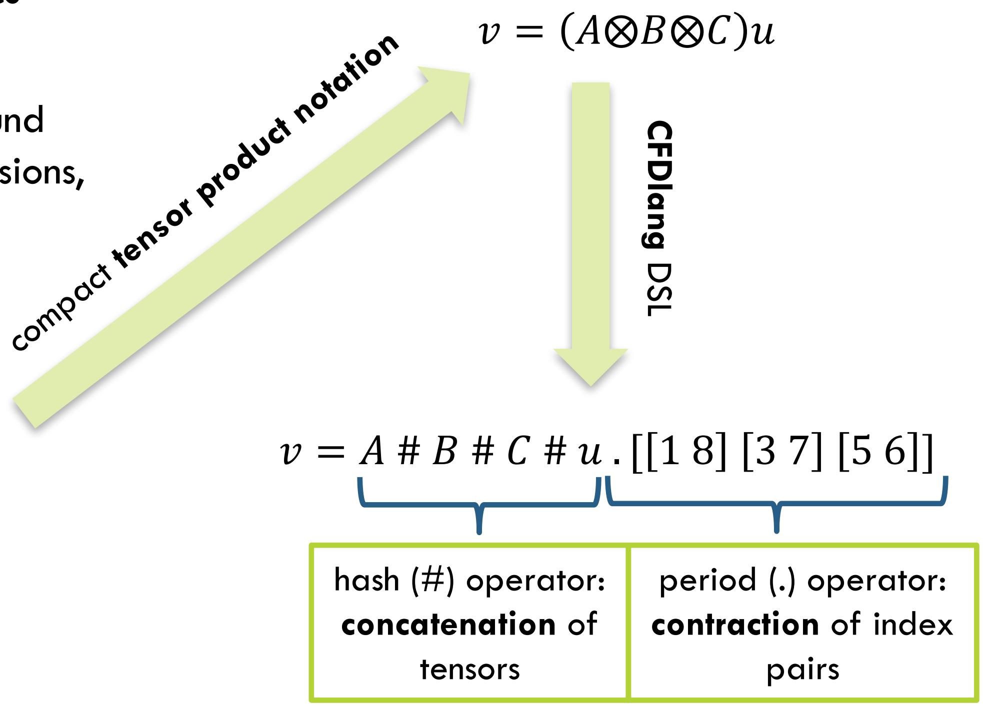
- 3-dimensional problems in fluid dynamics
  - Coefficients u_{ijk}.
  - Structure of **operators** (i.e. compute-bound **kernels**) reflects the three spatial dimensions, e.g.:



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

reductions/contractions

  - Matrices A, B, C.
  - 3-dimensional **tensors** (i.e. **arrays**) u, v.



$v = (A \otimes B \otimes C) u$

compact **tensor product** notation

CFDlang DSL

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

hash (#) operator: **concatenation** of tensors

period (.) operator: **contraction** of index pairs

Outline

1. Background and motivation
2. The CFDlang domain-specific language
   1. Language definition
   2. Code generation
3. Evaluation of CFDlang-generated code performance
4. Summary and outlook

The CFDlang DSL (1/2)

```
⟨program⟩   ::=  ⟨decl⟩* ⟨elem⟩? ⟨stmt⟩*
⟨decl⟩      ::=  var ⟨io⟩? ⟨id⟩ : [⟨int-list⟩]
⟨io⟩        ::=  input | output
⟨elem⟩      ::=  elem [⟨id-list⟩] ⟨int⟩
⟨stmt⟩      ::=  ⟨id⟩ = ⟨expr⟩
⟨expr⟩      ::=  ⟨term⟩ | ⟨term⟩ (+|-) ⟨expr⟩
⟨term⟩      ::=  ⟨factor⟩ | ⟨factor⟩ (*|/) ⟨expr⟩
                | ⟨factor⟩ . [⟨pair-list⟩]
⟨factor⟩    ::=  ⟨atom⟩ | ⟨atom⟩ # ⟨factor⟩
⟨atom⟩      ::=  ⟨id⟩ | ( ⟨expr⟩ )
```

- CDFlang program structure
  - **Declarations** (of tensors) and **statements.**
  - Statements assign **expressions** to (tensor) **variables.**
- **Input/output** qualifiers
  - Declare variables for communication between the kernel and the ambient numerical application.
- **Element directive**
  - Informs the DSL about which tensors are to be instantiated once per volume element Ω_e.

Embarrassing parallelism between kernel executions for different volume elements.

The CFDlang DSL (2/2)

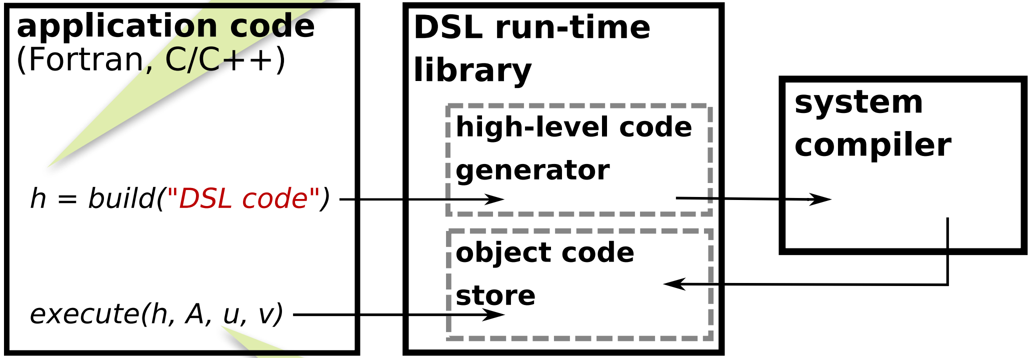
```
var input  x : [3 4 5]
var input  y : [3 4 5]
var output z : [3 4 5]

z  =  x  *  y
```

```
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];
      }
    }
  }
}
```

- ✓ Expressions and assignments.
- ✓ Loop nests.
- ✓ Kernel signatures/interface.
- ✓ Aliasing.

Integration of DSL programs

Kernel handle:
pointer to generated object code
(for low-overhead kernel calls).



application code
(Fortran, C/C++)

h = build("DSL code")

execute(h, A, u, v)

DSL run-time library

high-level code generator

object code store

system compiler

Input/output tensors:
passed as arguments in kernel call.

- **High-level code generator:**
  - CDFlang programs are lowered to C code.
- System C compiler:
  - *icc* (Intel compiler suite).
  - Kernel object code is loaded into the application's memory at **application run-time.**
  - Tensor dimensions are not known until run-time.

Code generation and optimization (1/4)

Multiple contractions (*interpolation* operator):

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

elem [u v] 216   /* 6^3 = 216 */

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

```
/* 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 */
```

What is the complexity of this?
(in terms of $p + 1 = 7$)

Code generation and optimization (2/4)

- Evaluation order of contractions affects overall run-time complexity:

$$v_{ijk} = \sum_{i'=0}^{p} \sum_{j'=0}^{p} \sum_{k'=0}^{p} A_{kk'} A_{jj'} A_{ii'} u_{i'j'k'}$$

$\mathcal{O}(p^6)$

$$v_{ijk} = \sum_{k'=0}^{p} A_{kk'} \sum_{j'=0}^{p} A_{jj'} \sum_{i'=0}^{p} A_{ii'} u_{i'j'k'}$$

$\mathcal{O}(p^4)$

- Minimizing number of arithmetic operations is generally NP-complete.
  - C-C Lam, P Sadayappan, and R Wenger. *On Optimizing A Class Of Multi-Dimensional Loops With Reductions For Parallel Execution*. 1997.
  - CFD use cases have simpler combinatorics.

- Trade-off: doing reductions in sequence introduces **temporary variables.**
  - Acceptable for CFD use cases due to small data size.

Code generation and optimization (3/4)

- Thread-level parallelism
  - Kernels executed for different elements are fully independent.
  - Those kernels can be run in parallel threads.

```
/* element loop: */
#pragma omp for
for (int e = 0; e < 216; e++) {
 ...
} /* end of element loop */
```

- SIMD parallelism and vectorization
  - Many (nested) loops.
  - Unclear which loops are best to be vectorized.
  - **Not** the reduction loops!

```
/* single reduction: */
#pragma simd
for (int i0 = 0; i0 < 7; i0++) {
 for (int j0 = 0; j0 < 7; j0++) {
  #pragma simd
  for (int k0 = 0; k0 < 7; k0++) {
   v[e][i0][j0][k0] = 0.0;
   for (int k1 = 0; k1 < 7; k1++) {
    v[e][i0][j0][k0] += A[k0][k1]
             * u[e][i0][j0][k1];
   }
} } }
```

Code generation and optimization (4/4)

- Code generation summary
  - **Transform nested reduction** loops into sequences of reduction loops.
  - Guide the system compiler's vectorizer by inserting **SIMD pragmas** in suitable places.
  - Computations on different (volume) elements are embarrassingly parallel.
    - Run kernels in parallel threads.
    - Usually only one thread per core.
    - (Detailed study not part of this work.)

Outline

1. Background and Motivation
2. The CFDlang domain-specific language
    1. Language definition
    2. Code generation
3. **Evaluation of CFDlang-generated code performance**
4. Summary and outlook

Performance evaluation (1/2)

Interpolation operator:

```
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]]
```

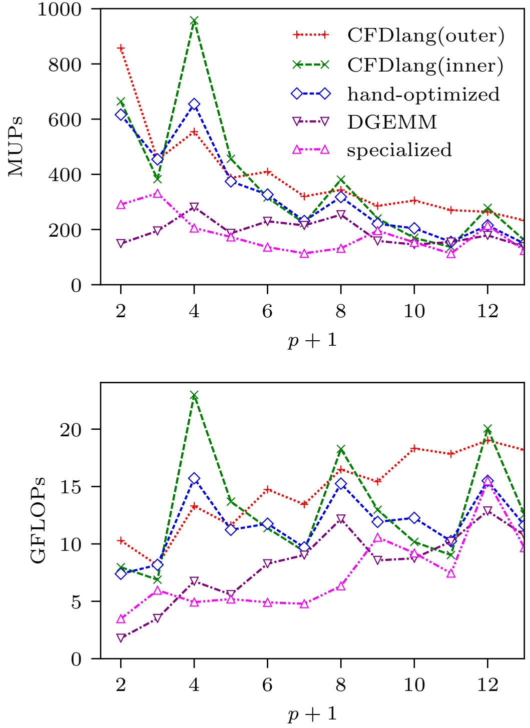
Inverse Helmholtz operator:

```
var input  S : [7 7]
var input  D : [7 7 7]
var input  u : [7 7 7]

var output v : [7 7 7]

elem [D u v] 216

v = S # S # S # u . [[1 6][3 7][5 8]]
v = D * v
v = S # S # S # v . [[0 6][2 7][4 8]]
```

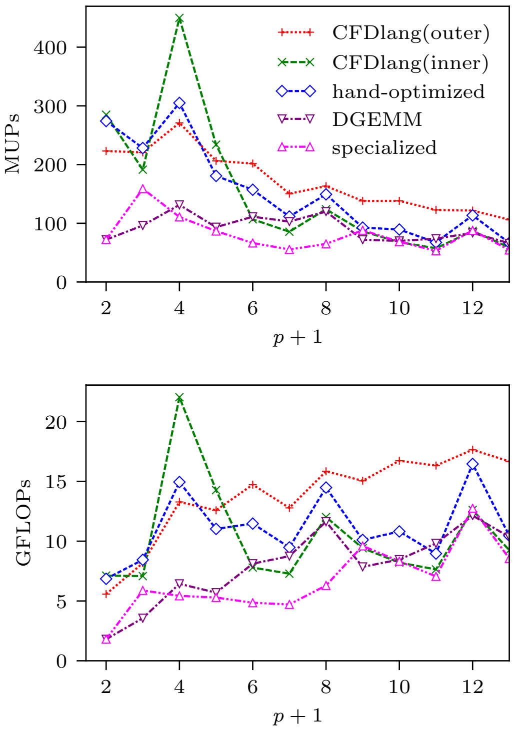
- Code variants:
  - CFDlang-generated
  - hand-optimized
  - DGEMM (Intel MKL)
  - *specialized* (baseline)

Performance evaluation (2/2)

Interpolation operator:



Inverse Helmholtz operator:



Outline

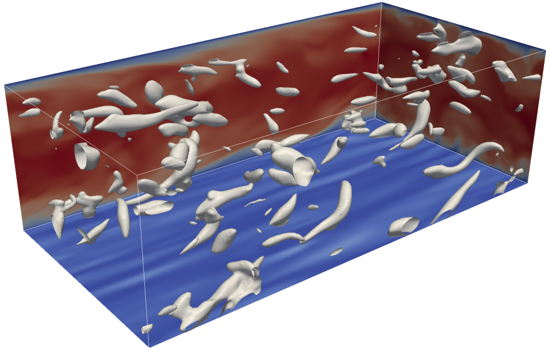
1. Background and motivation

2. The CFDlang domain-specific language
    1. Language definition
    2. Code generation

3. Evaluation of CFDlang-generated code performance

4. **Summary and outlook**

Summary and outlook



- CFDlang DSL
  - Abstractions for **tensor operations**, esp. **contractions**.
  - Mathematical notation: no explicit loops or indices.
- Code generation and performance
  - Automatic re-ordering of nested contractions.
  - Automatic **parallelization** (with OpenMP thread) and **vectorization** (with SIMD pragmas).
  - On par or better than best manually optimized codes.
- Language design
  - Implement further numerical kernels.
  - Derive requirements for extensions of the current CFDlang DSL.
  - Bring notation closer to **mathematical and abstract tensor product notation**.

CHAIR FOR
COMPILER
CONSTRUCTION

cfaed CENTER FOR ADVANCING ELECTRONICS DRESDEN

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

Norman A. Rink
Jeronimo Castrillon

Immo Huismann
Jörg Stiller
Jochen Fröhlich

Adilla Susungi
Claude Tadonki

Work supported by the German Research Foundation (DFG) within the Cluster of Excellence 'Center for Advancing Electronics Dresden' (cfaed).

Thank you.