CFDlang:
a (small) DSL for linear and tensor algebra

Norman A. Rink
Technische Universität Dresden
norman.rink@tu-dresden.de

3 September 2018
Outline

1. Background and motivation
2. The CFDlang domain-specific language
3. Evaluation of CFDlang-generated code performance
4. Formal language specification
5. Summary and outlook
Outline

1. **Background and motivation**
2. The CFDlang domain-specific language
3. Evaluation of CFDlang-generated code performance
4. Formal language specification
5. Summary and outlook
HPC is applied linear algebra

- Numerical simulation
  - Vehicles and aircraft design
  - Weather and climate simulation/forecasting

- Geometry reconstruction
  - Face recognition
  - Medical imaging

- Machine learning
  - Image processing
  - Data analytics
High-order methods and tensors

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

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

\[v = (A \otimes B \otimes C) u\]

hash (#) operator: concatenation of tensors
period (.) operator: contraction of index pairs

No explicit loops.
No explicit indices.
→ Close to math. notation but also good for the compiler/optimizer.
Outline

1. Background and motivation
2. The CFDlang domain-specific language
3. Evaluation of CFDlang-generated code performance
4. Formal language specification
5. Summary and outlook
The CFDlang DSL (1/2)

- **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 $\Omega_e$.

---

Embarrassing parallelism between kernel executions for different volume elements.
The CFDlang DSL (2/2)

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

- Expressions and assignments.
- Loop nests.
- Kernel signatures/interface.
- Aliasing.
Integration of DSL programs

- **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.

---

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)
```

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

```
DSL run-time library

high-level code generator

object code store

system compiler
```

---
Multiple contractions (interpolation operator):

```c
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]]
```

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

```c
/* 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 */
```
Evaluation order of contractions affects overall run-time complexity:

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

Minimizing number of arithmetic operations is generally NP-complete.

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

SIMD parallelism and vectorization
- Many (nested) loops.
- Unclear which loops are best to be vectorized.
- Not the reduction loops!
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
3. Evaluation of CFDlang-generated code performance
4. Formal language specification
5. Summary and outlook
Performance evaluation (1/2)

Interpolation operator:

\[
\text{var input } A : [7 \ 7] \\
\text{var input } u : [7 \ 7 \ 7] \\
\text{var output } v : [7 \ 7 \ 7] \\
\]

\[
\text{elem } [u \ v] \ 216 \\
\]

\[
v = A \# A \# A \# u \cdot [[1 \ 6][3 \ 7][5 \ 8]]
\]

Code variants:

- CFDlang-generated
- hand-optimized
- DGEMM (Intel MKL)
- specialized (baseline)

Inverse Helmholtz operator:

\[
\text{var input } S : [7 \ 7] \\
\text{var input } D : [7 \ 7 \ 7] \\
\text{var input } u : [7 \ 7 \ 7] \\
\]

\[
\text{var output } v : [7 \ 7 \ 7] \\
\]

\[
\text{elem } [D \ u \ v] \ 216 \\
\]

\[
v = S \# S \# S \# u \cdot [[1 \ 6][3 \ 7][5 \ 8]] \\
v = D \ast v \\
v = S \# S \# S \# v \cdot [[0 \ 6][2 \ 7][4 \ 8]]
\]
Performance evaluation (2/2)

Interpolation operator:

Inverse Helmholtz operator:
1. Background and motivation
2. The CFDlang domain-specific language
3. Evaluation of CFDlang-generated code performance
4. Formal language specification
5. Summary and outlook
Linear algebra and CFDlang

- Linear algebra ...
  - ... is well-understood (1st year undergraduate level).
  - ... has a powerful formalism.
    - Evaluation of components of vectors/matrices specified by arithmetic formulae.
  - ... comes with clear rules that say what operations are allowed.
    - E.g. multiplication of matrices requires that their dimensions match suitably.

- Goal:
  - Formally specify the behavior of CFDlang programs in terms of linear algebra.
    - Evaluation of programs \(\rightarrow\) specify in terms of arithmetic formulae.
    - Correctness of programs \(\rightarrow\) rely on the rules for linear algebra operations.
Matrix-vector multiplication: \( u = (v \# w) \). [2 3]

in arithmetic: \( u_i = \sum_{j=1}^{n} v_{ij} w_j \)

Typing rules:

Derivation tree taken from: "Detection and exploitation of data-parallelism in assignments of multi-dimensional tensors", Jasper Ullrich, Bachelor thesis, 2018

http://nbn-resolving.de/urn:nbn:de:bsz:14-qucosa2-319722
Program evaluation

Evaluation of assignments:

\[ x \in \text{dom}(\Gamma) \quad t = \Gamma(x) \]

\[ \forall i \leq t \ x_i \in \text{dom}(\mu) \land r_i = \text{eval}_{\Gamma,\mu}(e_i) \]

\[ \langle \mu, x = e \rangle \rightarrow_{\Gamma} \langle \mu \{ \forall i \leq t \ x_i \mapsto r_i \}, \emptyset \rangle \]

\text{ev-stmt}

Translation to arithmetic:

\[
\text{eval}_{\Gamma,\mu}(x_{i_1 \ldots i_k}) = \mu(x_{i_1 \ldots i_k}) \tag{7}
\]

\[
\text{eval}_{\Gamma,\mu}((e)_{i_1 \ldots i_k}) = \text{eval}_{\Gamma,\mu}(e_{i_1 \ldots i_k}) \tag{8}
\]

\[
\text{eval}_{\Gamma,\mu}(e_0 \#_i e_{i_{01} \ldots i_{0k} i_{11} \ldots i_{1l}}) = \text{eval}_{\Gamma,\mu}(e_{0 i_{01} \ldots i_{0k}}) \cdot \text{eval}_{\Gamma,\mu}(e_{i_{11} \ldots i_{1l}}) \tag{9}
\]

\[
\text{eval}_{\Gamma,\mu}(e^\cdot [m n]_{i_{11} \ldots i_m \ldots i_{n1} \ldots i_k}) = \text{eval}_{\Gamma,\mu}(e_{i_1 \ldots i_n \ldots i_m \ldots i_k}) \tag{10}
\]

\[
\text{eval}_{\Gamma,\mu}(e.[m n]_{i_{11} \ldots i_m \ldots i_{n1} \ldots i_k}) = \sum_{l=1}^{\pi_m(t)} \text{eval}_{\Gamma,\mu}(e_{i_1 \ldots i_{m-1} l_{i_{m+1} \ldots i_{n-1} l_{i_{n+1} \ldots i_k}}}), \quad \text{where } \Gamma \vdash e : t \tag{11}
\]

Assignments update the memory \( \mu \).
Program evaluation – example

\[ u = (v \# w).[2 3] \]

\[ \begin{align*}
\text{eval}_{\mu}(x_{i_1 \cdots i_k}) &= \mu(x_{i_1 \cdots i_k}) \\
\text{eval}_{\mu}(e_{i_1 \cdots i_k}) &= \text{eval}_{\mu}(e_{i_1 \cdots i_k}) \\
\text{eval}_{\mu}(e \# e_{i_{01} \cdots i_{0b} i_{11} \cdots i_{1l}}) &= \text{eval}_{\mu}(e_{i_{01} \cdots i_{0b}}) \cdot \text{eval}_{\mu}(e_{i_{11} \cdots i_{1l}}) \\
\text{eval}_{\mu}(e \cdot [m \cdot n]_{i_1 \cdots i_m \cdots i_k}) &= \text{eval}_{\mu}(e_{i_1 \cdots i_m \cdots i_k}) \\
\text{eval}_{\mu}(e \cdot [m \cdot n]_{i_1 \cdots i_m \cdots i_k}) &= \sum_{t=1}^{\pi_m(t)} \text{eval}_{\mu}(e_{i_1 \cdots i_m \cdots i_k}) \\
\text{eval}(v \# w).[2 3]_i &= \sum_{j=1}^{400} \mu(v_{ij}) \cdot \mu(w_j)
\end{align*} \]

for all \(1 \leq i \leq 300\),

\[ \mu(u_i) = \sum_{j=1}^{400} \mu(v_{ij}) \cdot \mu(w_j) \]
Can now prove mathematically that programs have certain properties.

"Standard" properties:

- **Type safety**
  - A well-typed CFDlang program does not access memory out-of-bounds.
- **Termination**
  - A well-typed CFDlang program terminates. (This is trivial though.)

More interesting properties:

- CFDlang programs compute correct results when operating on padded memory/tensors.
  - This can improve the performance of vectorized code.
- A large class of assignments can be evaluated without using additional temporary memory.

Outline

1. Background and motivation
2. The CFDlang domain-specific language
3. Evaluation of CFDlang-generated code performance
4. Formal language specification
5. Summary and outlook
Summary: what we have got

- CFDlang DSL
  - Abstractions for tensor operations, esp. contractions.
  - Mathematical notation: no explicit loops or indices.
  - Formal specification of the (domain-specific) language.

- 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.
Outlook: what we (might) want

- **Performance**
  - Smarter and more widely-applicable vectorization.
  - Schedule more than one (OpenMP) thread per core.

- **Language design**
  - Implement further numerical kernels.
  - Bring notation closer to **mathematical and abstract tensor product notation**.
    - Perhaps look at MATLAB/Octave for inspiration.

- **Formal work and optimization**
  - Any code transformations that is valid as an equation between arithmetic formulae is correct.
  - Use the formal memory $\mu$ to analyze memory usage (with polyhedral methods?).
CFDlang: a (small) DSL for linear and tensor algebra

Norman A. Rink
Technische Universität Dresden
norman.rink@tu-dresden.de

3 September 2018

Thank you.