**Karl F. A. Friebel**, Jeronimo Castrillon
Chair for Compiler Construction

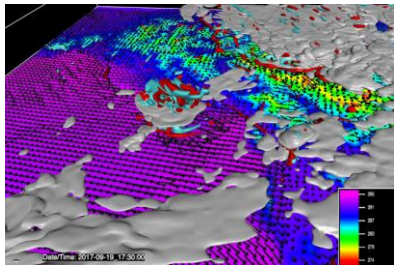
# MLIR stack for heterogeneous systems

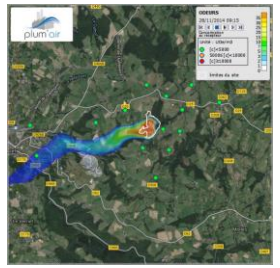
EVEREST+DAPHNE @ HiPEAC 2024, Munic// 2024-01-19

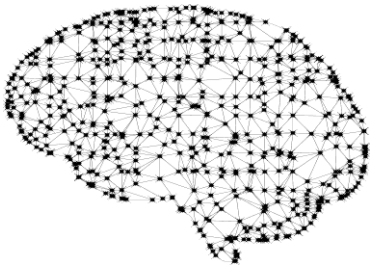
# Programming adaptable and heterogeneous systems

Golden age of computer architecture

- Massive parallel and heterogeneous systems
- Specialization for efficiency
- (Re-)Configurability for adaptability



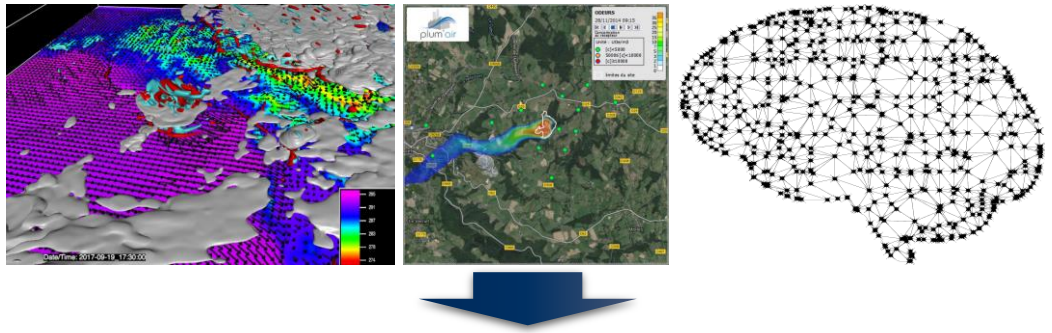




# Programming adaptable and heterogeneous systems

Golden age of computer architecture

- Massive parallel and heterogeneous systems
- Specialization for efficiency
- (Re-)Configurability for adaptability

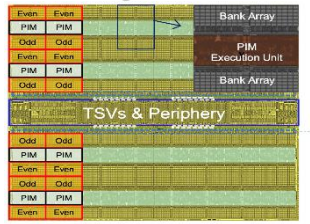


Heterogeneous HPC, BigData, ML systems

IT4I


HPC system

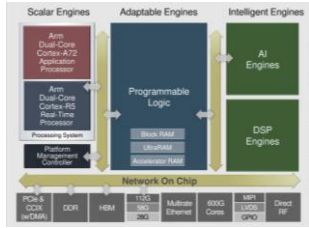
Samsung @ ISCA 2021




Emerging in/near memory computing

HBM-FPGA

AMD AECG

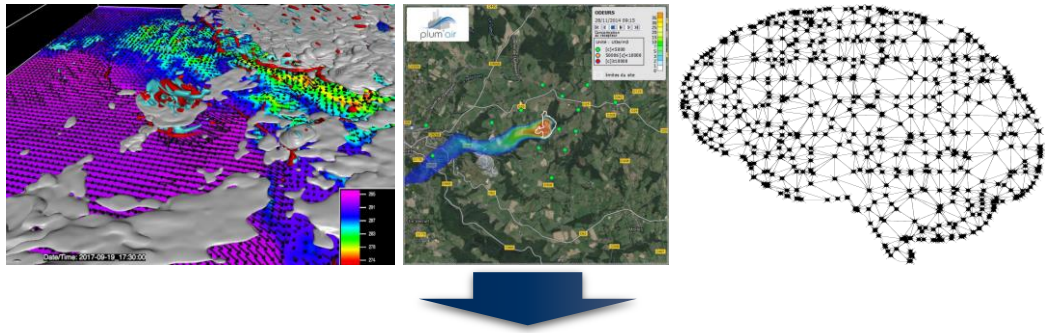



XDNA

# Programming adaptable and heterogeneous systems

Golden age of computer architecture

- Massive parallel and heterogeneous systems
- Specialization for efficiency
- (Re-)Configurability for adaptability

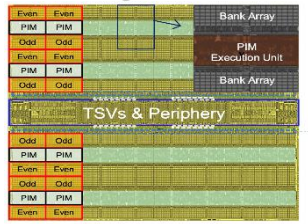


Heterogeneous HPC, BigData, ML systems

IT4I


HPC system

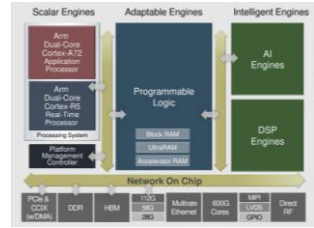
Samsung @ ISCA 2021




Emerging in/near memory computing

AMD AECG

HBM-FPGA




XDNA

Programmabilty challenges

- Huge effort to exploit underlying efficiency
- Interoperability of frameworks
- Performance portability

seconds — hours — days

Compile time

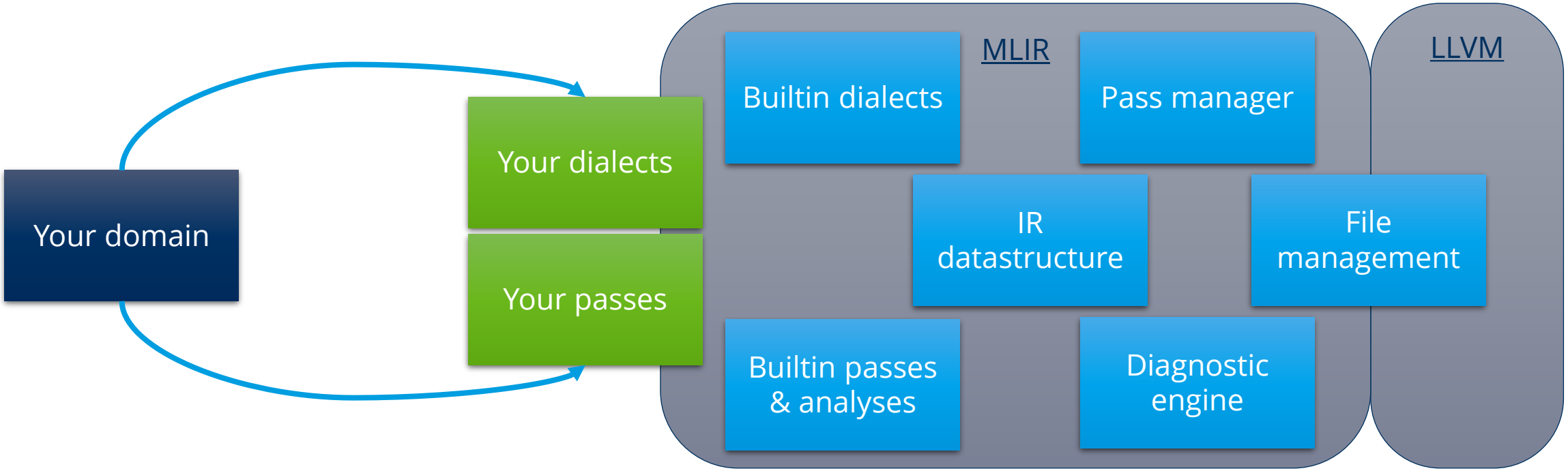
XILINX VITIS™

# Outline

- Introduction
- Heterogeneous MLIR
- EVEREST dialect stack
- Composable extensions

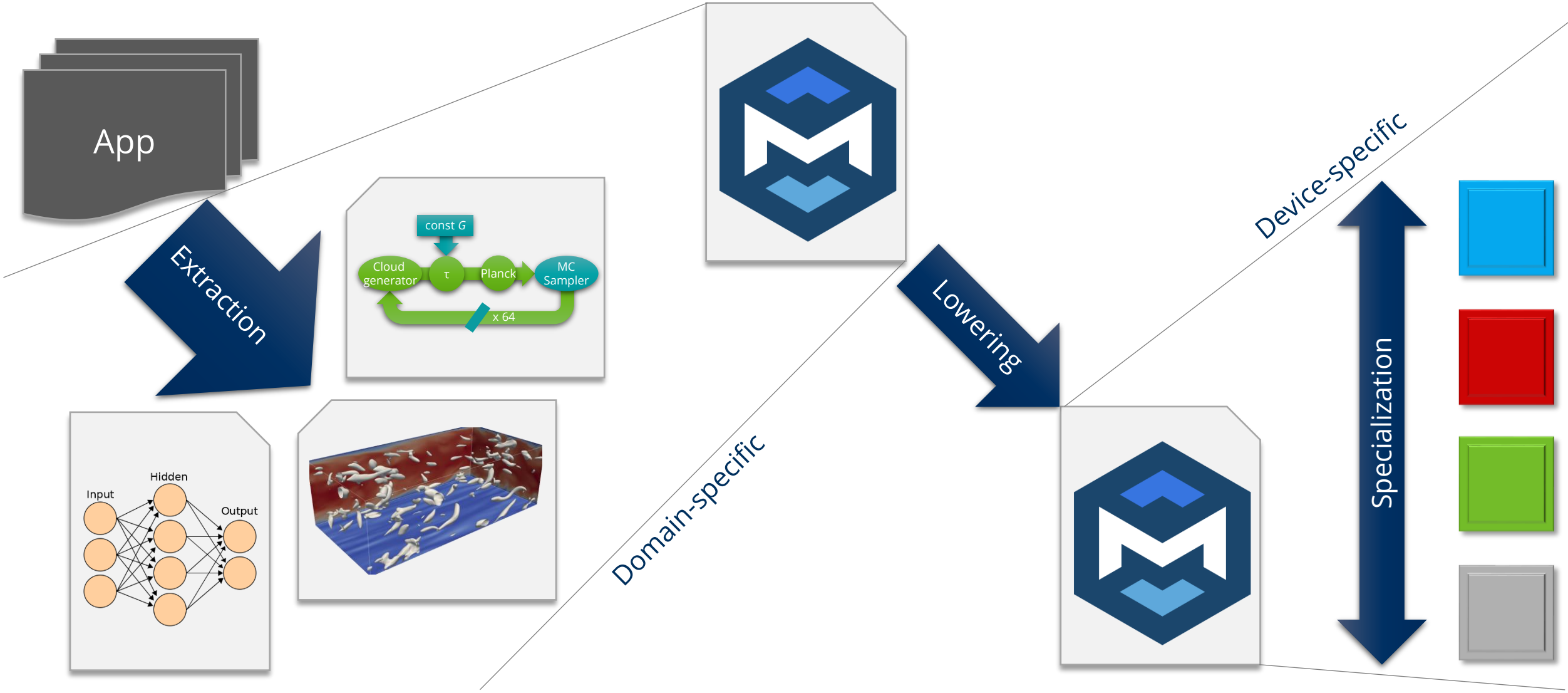
# Introduction to MLIR

MLIR is an LLVM framework for creating reusable compiler abstractions, centered around an extensible IR.




- MLIR provides the IR & pass management implementation

# MLIR-based compiler



# Progressive lowering

```
linalg.generic #traits
    ins(…) outs(…)
```

Abstract interface

```
scf.for %iv = %init to %ub step %cst1 {
    …
}
```

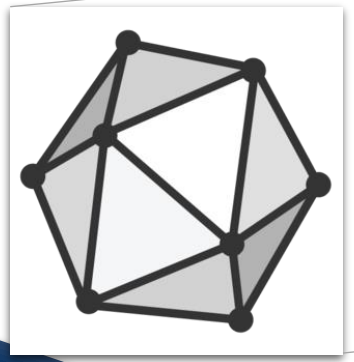
Device-archetype specific interface

```
^head(…):
    …
    cf.cond_br %cond, ^body(…), ^exit(…)
^body(…):
    …
```

Implementation backend

```
llvm.cond_br …
…
llvm.br …
```

# Targeting



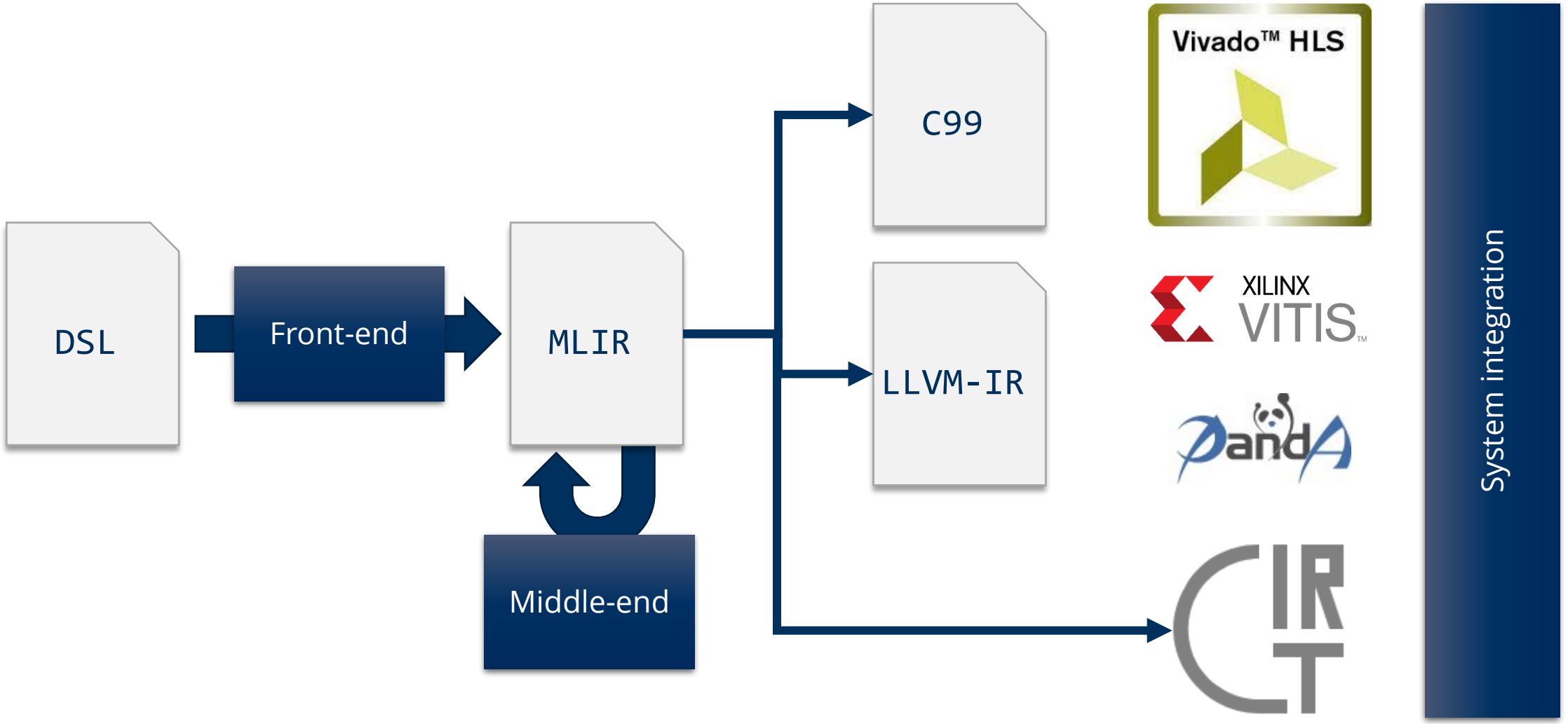
```
%ofm = "onnx.Conv"(%ifm, %wgt, %cst) {
    auto_pad = "NOTSET", dilations = [1, 1],
    group = 1 : si64, kernel_shape = [2, 2],
    pads = [0, 0, 0, 0], strides = [1, 1]}
    : (tensor<4x10x10x1xf32>, tensor<1x2x2x1xf32>, none)
    -> tensor<4x9x9x1xf32>
```

TOSA

```
%ofm = "tosa.conv2d"(%ifm, %wgt, %bias) {
    pad = array<i64: 0, 0, 0, 0>,
    stride = array<i64: 1, 1>,
    dilation = array<i64: 1, 1>}
    : (tensor<4x10x10x1xf32>, tensor<1x2x2x1xf32>, tensor<1xf32>)
    -> tensor<4x9x9x1xf32>
```

DVC

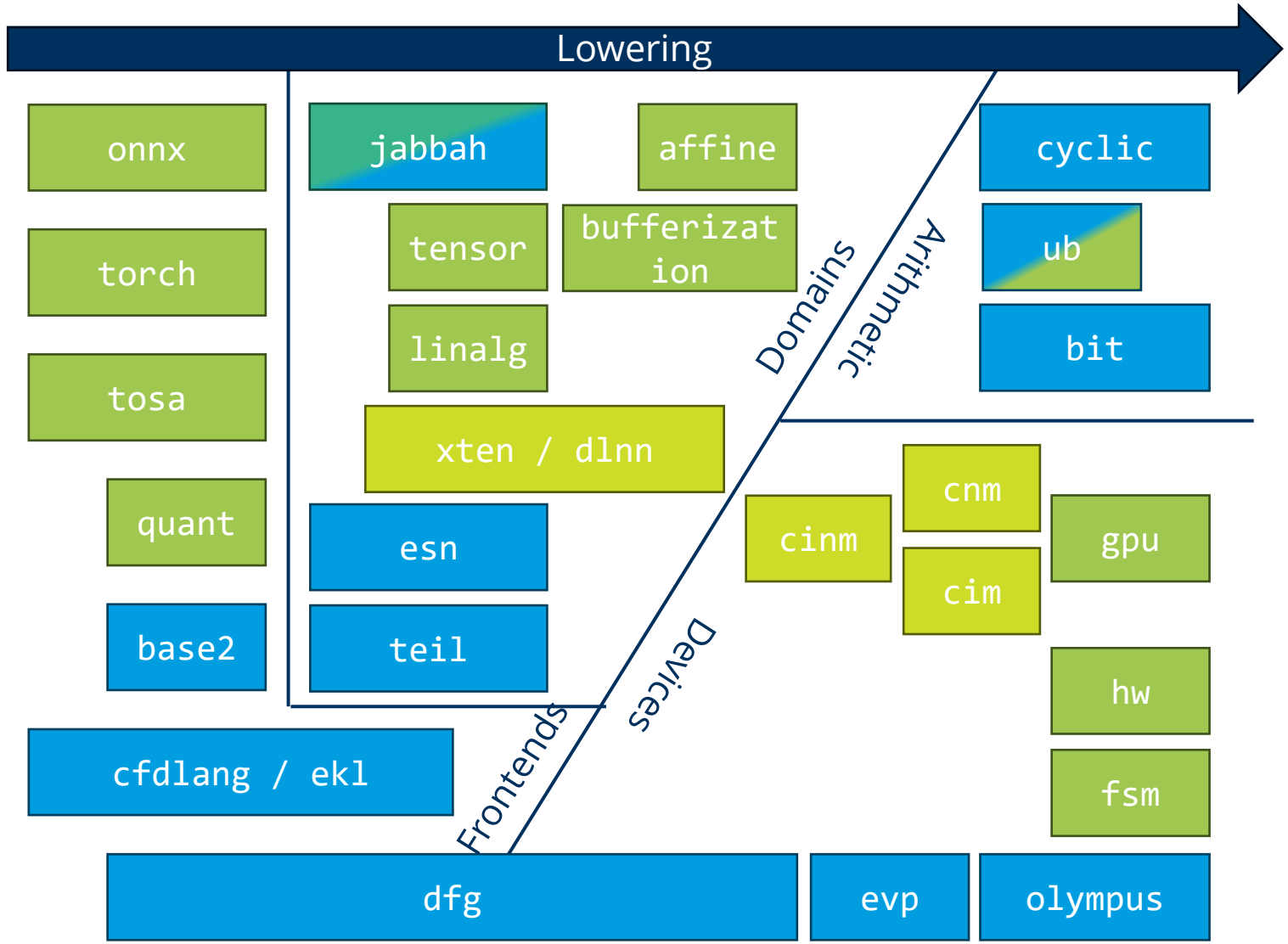
# MLIR end-to-end flow



# Outline

- Introduction
- Heterogeneous MLIR
- EVEREST dialect stack
- Programming models

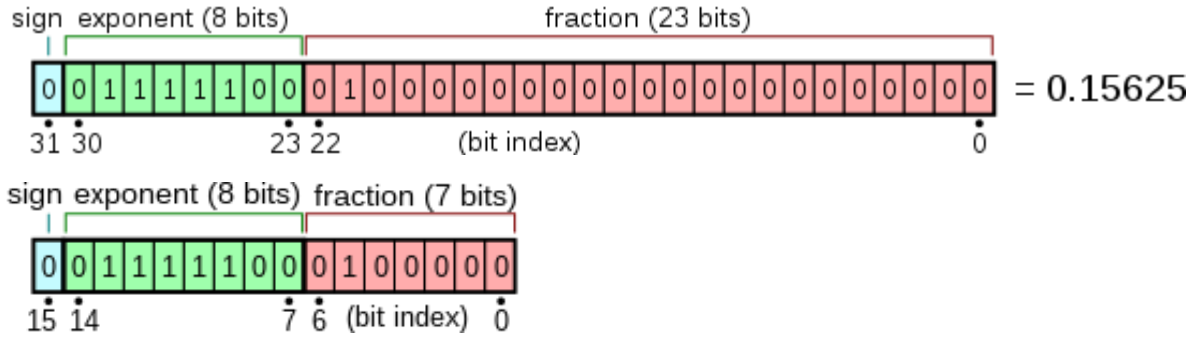
# EVEREST dialect stack



# Binary number types in `base2-mlir`

- Binary number types lack representation and specification in core
  - Operational signedness
  - Underspecified behavior
- Adaptable platforms thrive on customizable data types
  - FPGAs & fixed-point
  - AI accelerators & bfloat (& friends)

```
add [nuw, nsw] <ty> <op1>, <op2>
sub [nuw, nsw] <ty> <op1>, <op2>
mul [nuw, nsw] <ty> <op1>, <op2>

udiv [exact] <ty> <op1>, <op2>
urem <ty> <op1>, <op2>
sdiv [exact] <ty> <op1>, <op2>
srem <ty> <op1>, <op2>

icmp <pred> <ty> <op1>, <op2>
```

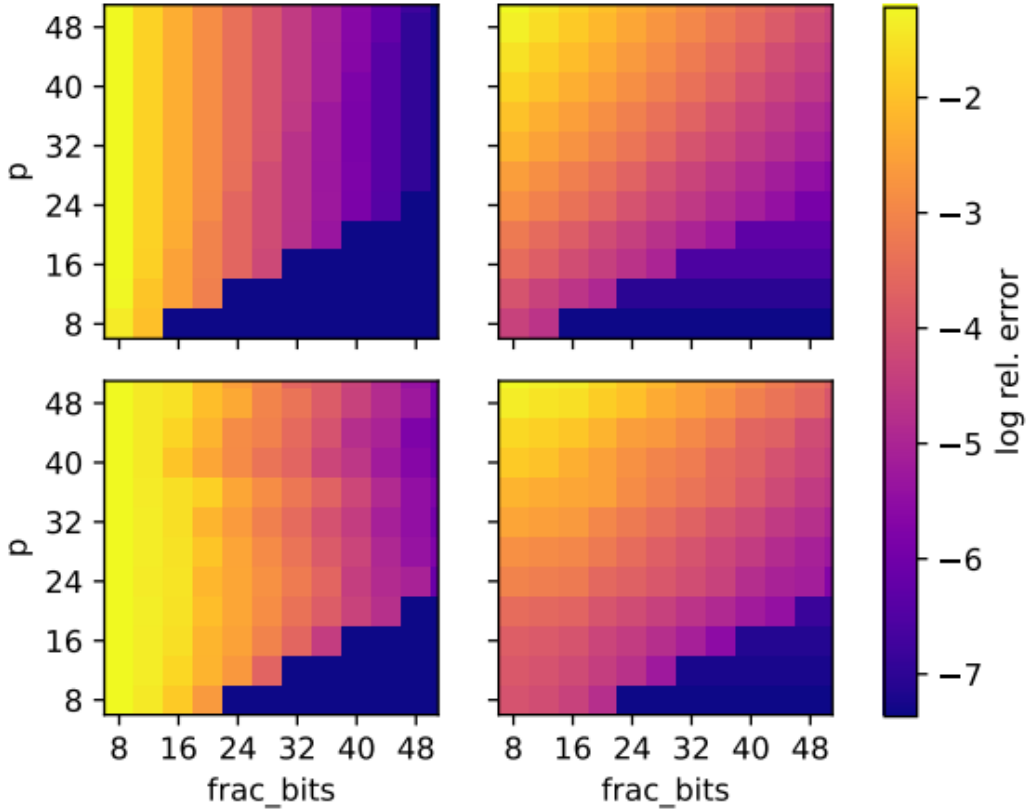


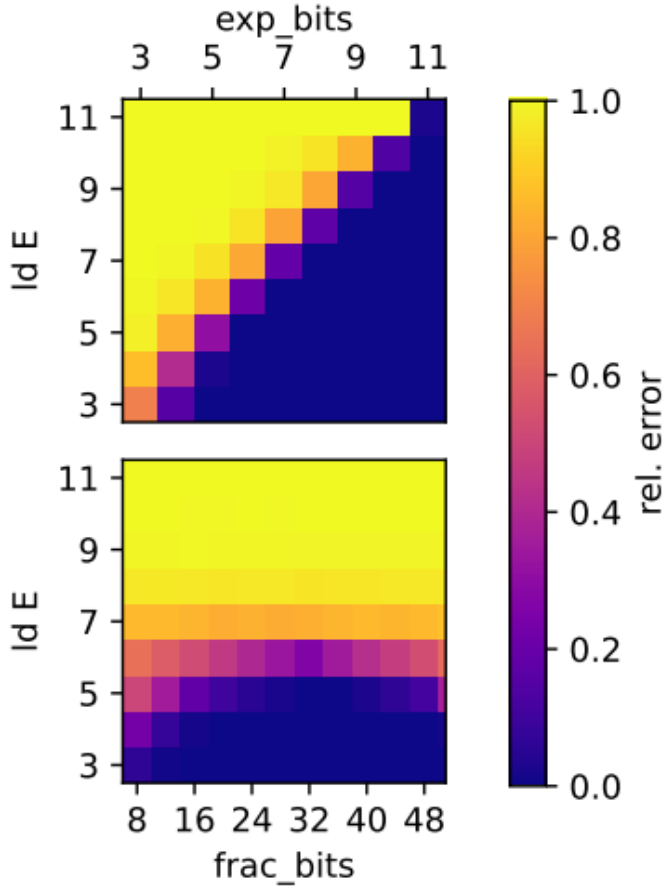
# Litmus test

Floating-point version:
`!ieee754<frac_bits, exp_bits>`

---

Fixed-point version:
`!fixed<signed 64, -frac_bits>`

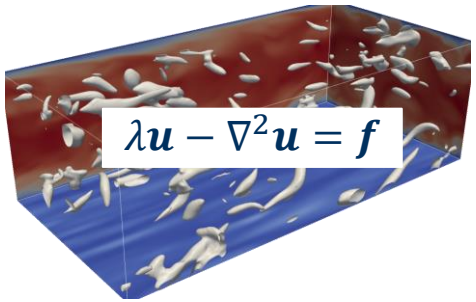


# Domain-specific languages (DSLs)

- Reduce input problem space
- Improve signal-to-noise ratio in input
- Leverage domain-specific optimization knowledge

→ MLIR dialects are closely related to DSLs



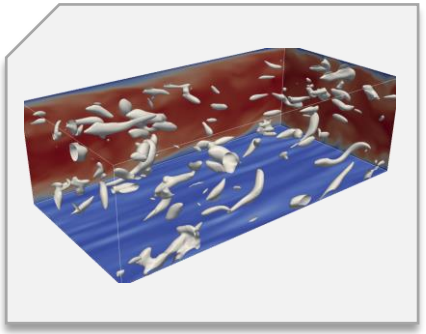

$$t = (S \otimes S \otimes S \otimes u)_{axbycz}{}^{xyz}$$

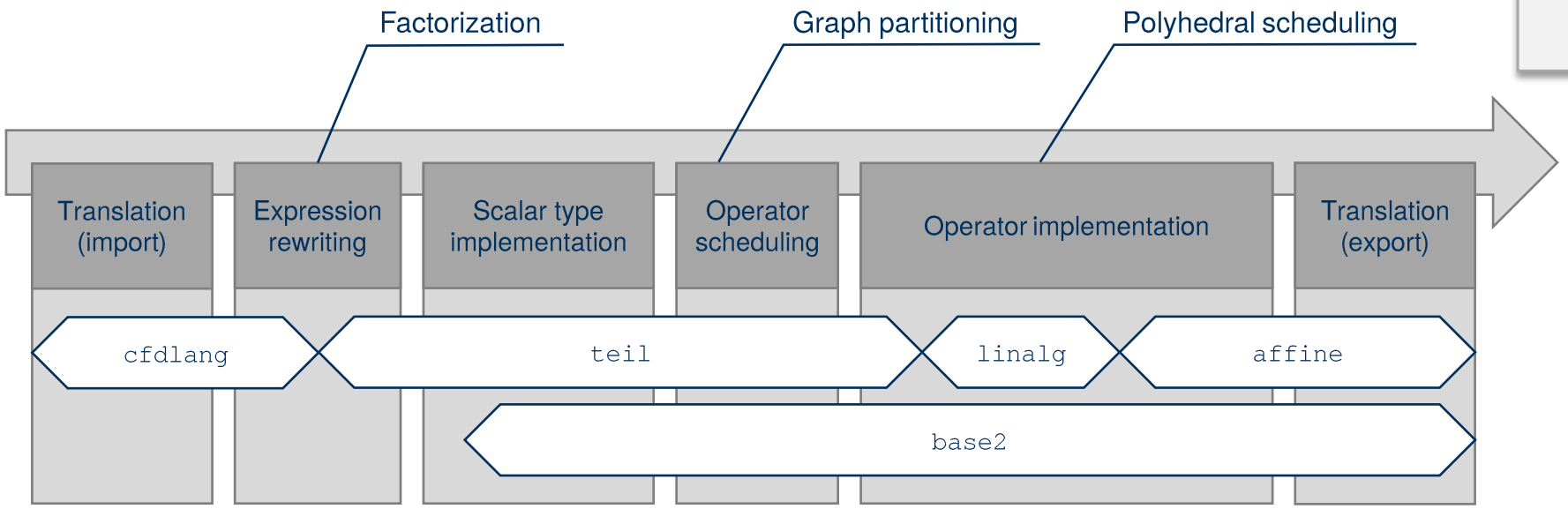
$$r = D \cdot t$$

$$v = (S \otimes S \otimes S \otimes r)_{xaybzc}{}^{xyz}$$

```
1 var input S    : [11 11]
2 var input D    : [11 11 11]
3 var input u    : [11 11 11]
4 var output v   : [11 11 11]
5 var t          : [11 11 11]
6 var r          : [11 11 11]
7 t = S#S#S#u . [[1 6][3 7][5 8]]
8 r = D * t
9 v = S#S#S#r . [[0 6][2 7][4 8]]
```

# Example 1 : Fluid dynamics





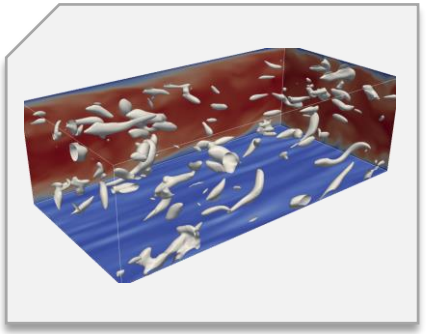

```
`base2.fixed_point` `<` $integerBits `,` $fractionalBits `>`
```

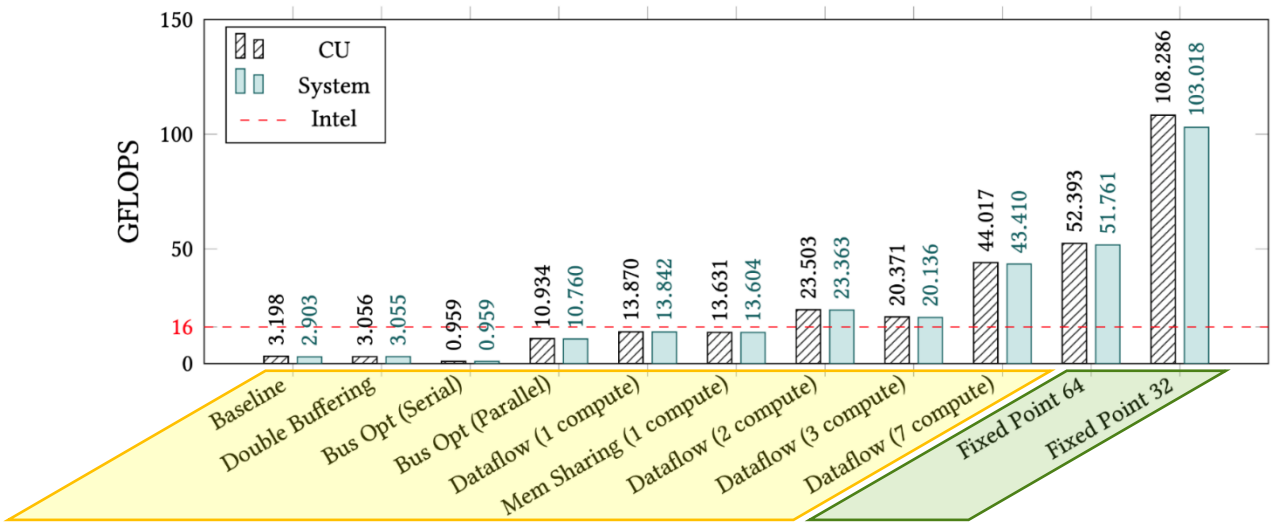
```
`base2.ieee754` `<` $precision `,` $expBits ( `,` $bias ) `>`
```

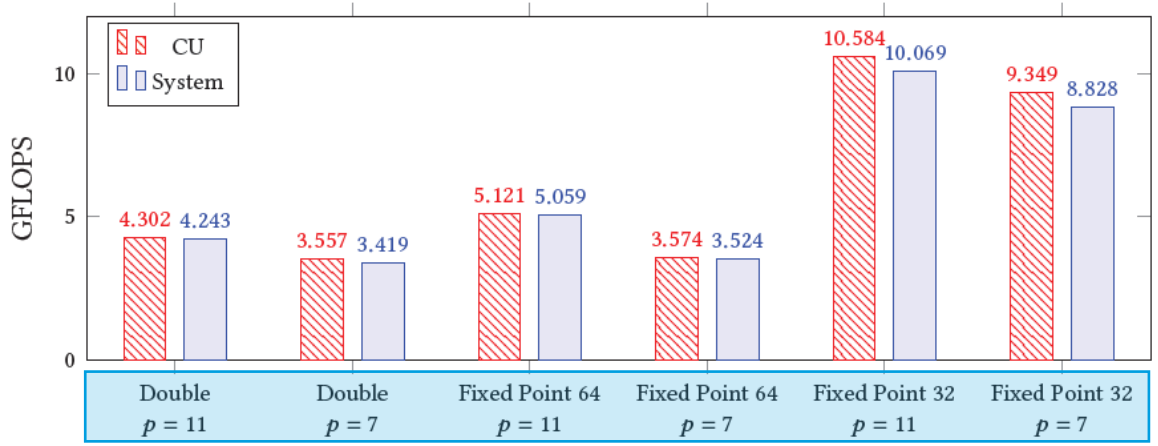
```
`base2.add` $roundingMode $lhs $rhs `:` type(result)
```

Stephanie Soldavini, et al. 2022. Automatic Creation of High-Bandwidth Memory Architectures from Domain-Specific Languages: The Case of Computational Fluid Dynamics. ACM Trans. Reconfigurable Technol. Syst. Just Accepted (September 2022). https://doi.org/10.1145/3563553

# Example 1 : Fluid dynamics




Algorithm

\+ Data types

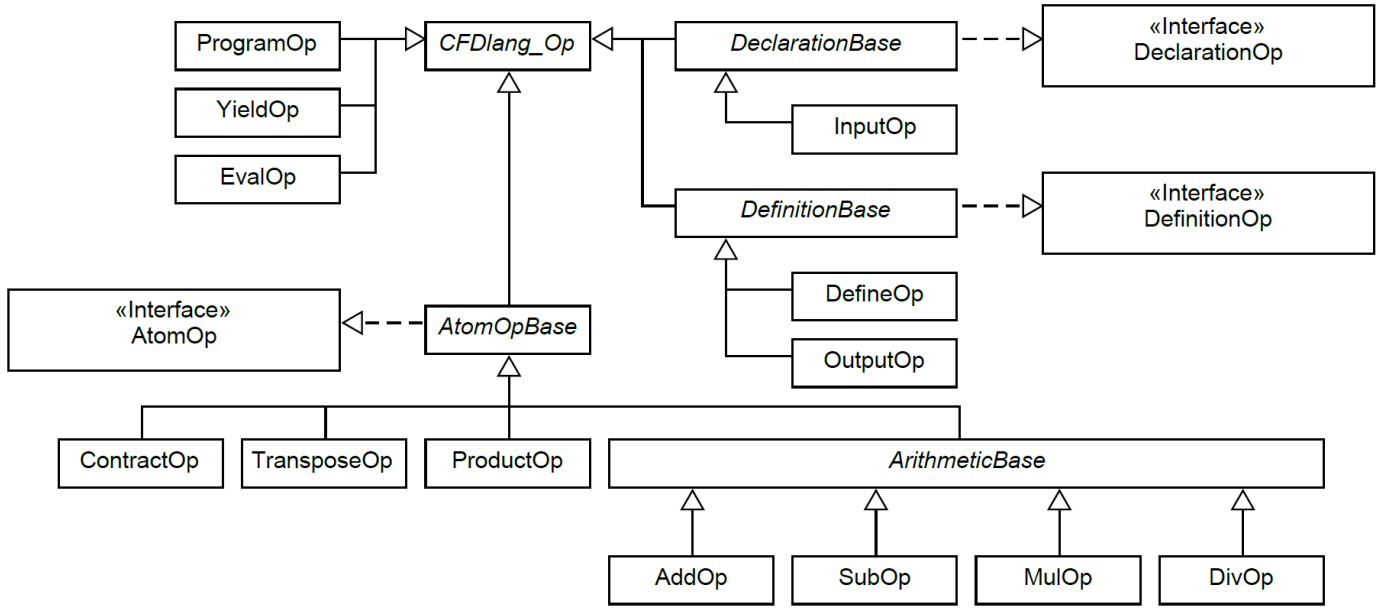
\+ Data movement

= Performance

Stephanie Soldavini, et al. 2022. Automatic Creation of High-Bandwidth Memory Architectures from Domain-Specific Languages: The Case of Computational Fluid Dynamics. ACM Trans. Reconfigurable Technol. Syst. Just Accepted (September 2022). https://doi.org/10.1145/3563553

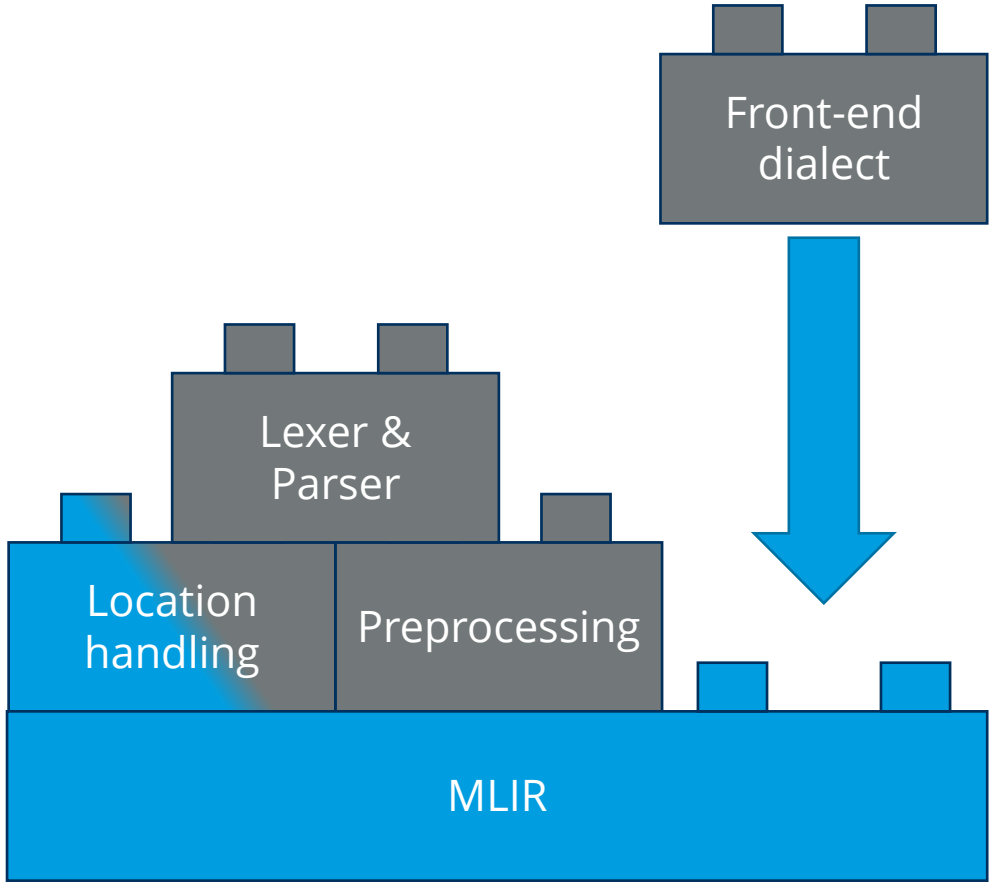
# Domain-specific languages (DSLs)

Hiding: Maybe too detailed




## Diagnostic messages

```
friebel@lua:~/repos/evp-tools$ build/bin/evp-translate --import-cfdlang test/Target/CFDlang/diag.cfd
test/Target/CFDlang/diag.cfd:7:5: error: type mismatch: value of type [11] cannot bind to declaration of type [11 11 11]
t = S # S # S # u . [[1 6] [3 7] [5 8] [2 4]]
    ^~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
test/Target/CFDlang/diag.cfd:5:5: note: see declaration here
var t         : [11 11 11]
    ^
friebel@lua:~/repos/evp-tools$
```



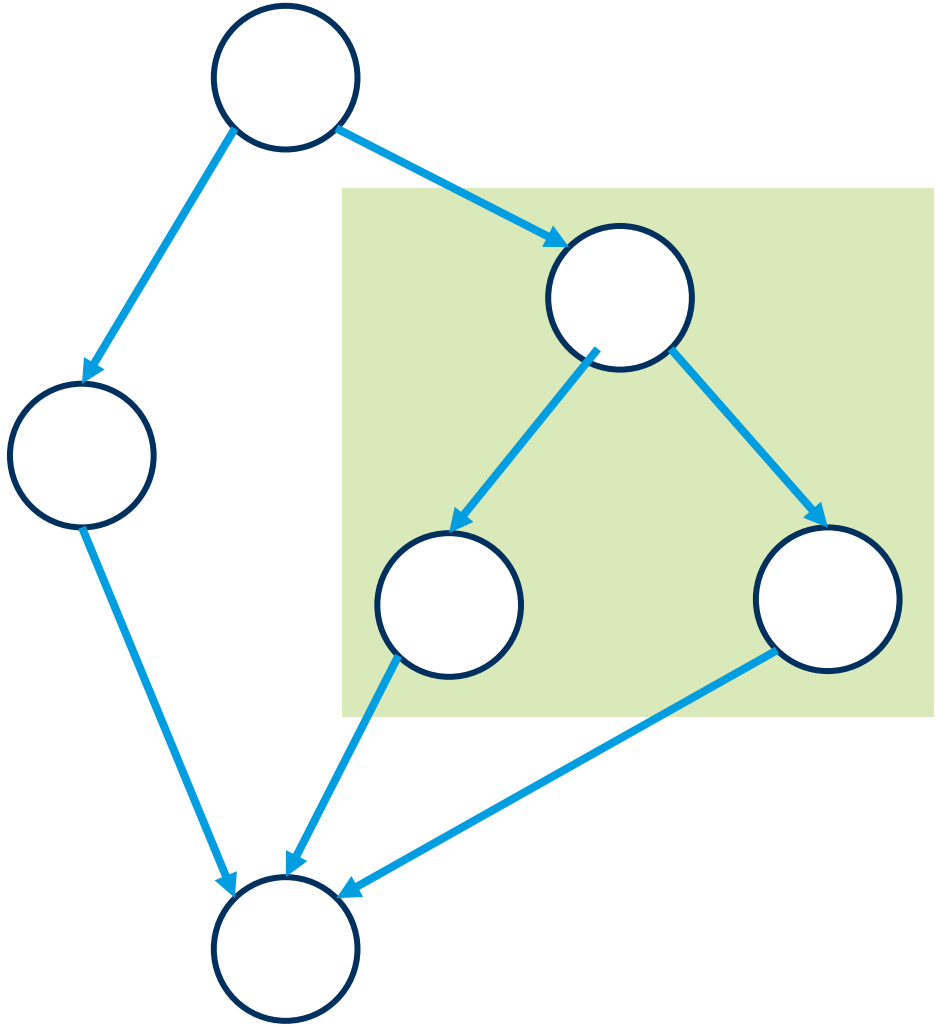
# Outline

- Introduction
- Heterogeneous MLIR
- EVEREST dialect stack
- Programming models

# KPNs for offloading

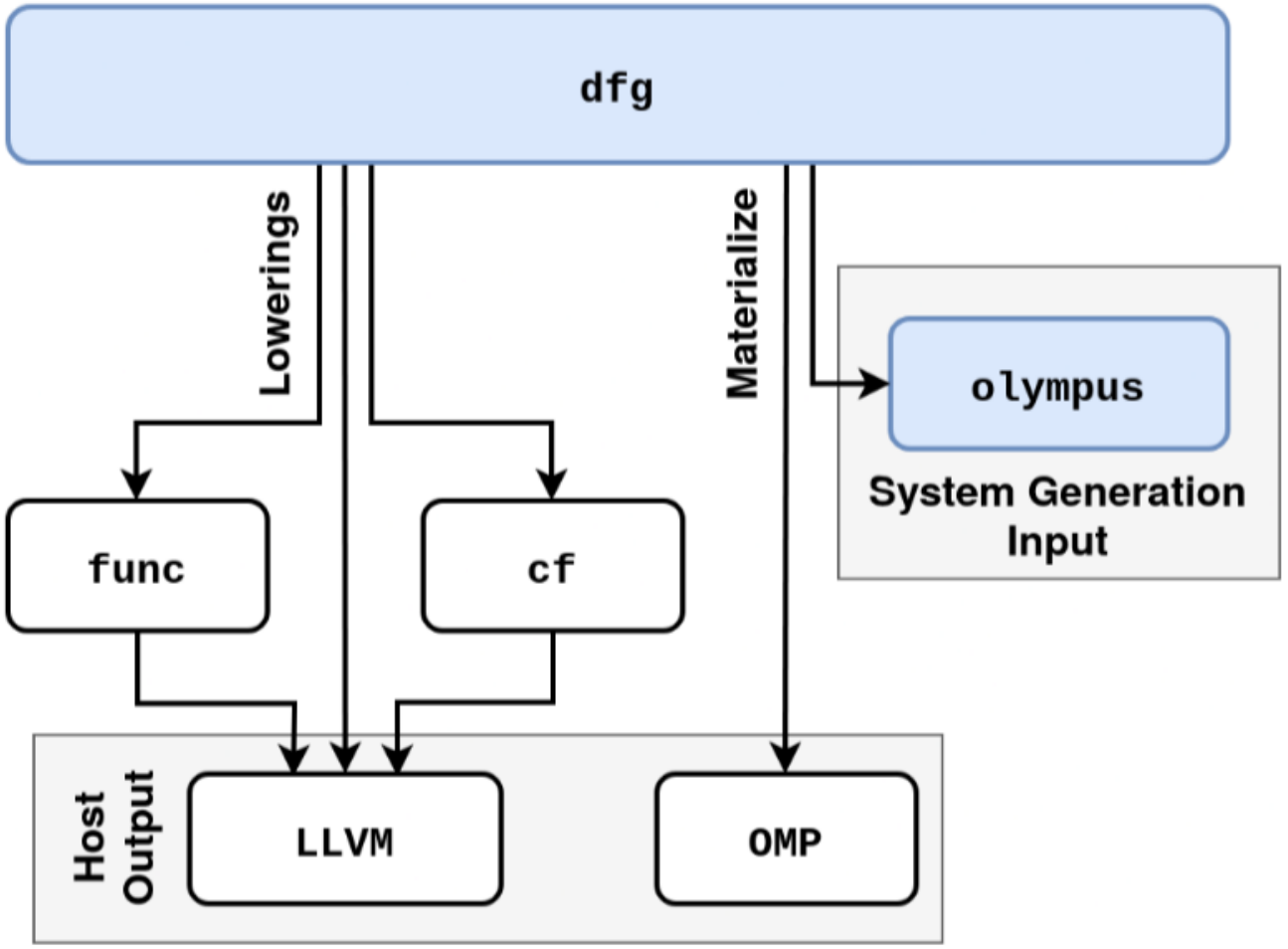
- KPNs model dataflow applications
- Offloaded regions bisect channels
- Graph transforms

```
dfg.operator @mul inputs (%a_in: i64, %b_in :
↪  i64)
                  outputs (%c_out: i64)
{
    dfg.loop inputs (%a_in: i64, %b_in : i64)
             outputs (%c_out: i64) {
        %a = dfg.pull %a_in : i64
        %b = dfg.pull %b_in : i64

        %c = arith.muli %a, %b : i64

        dfg.push(%c) %c_out : i64
    }
}
```



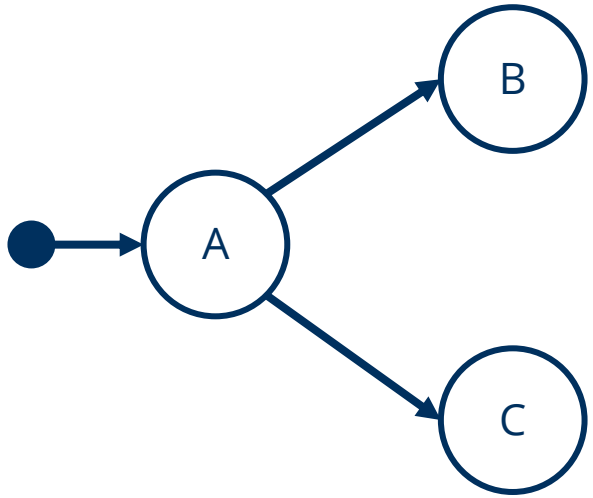
# DFG

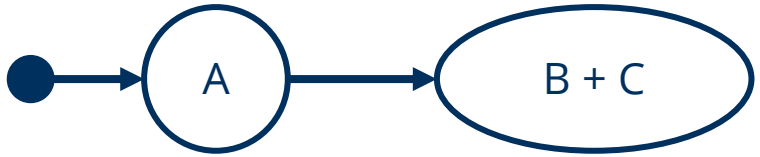
- Task programming model
- Non-invasive representation
- Transactional and asynchronous offloading possible
- Supports nested topologies



# KPNs for hardware

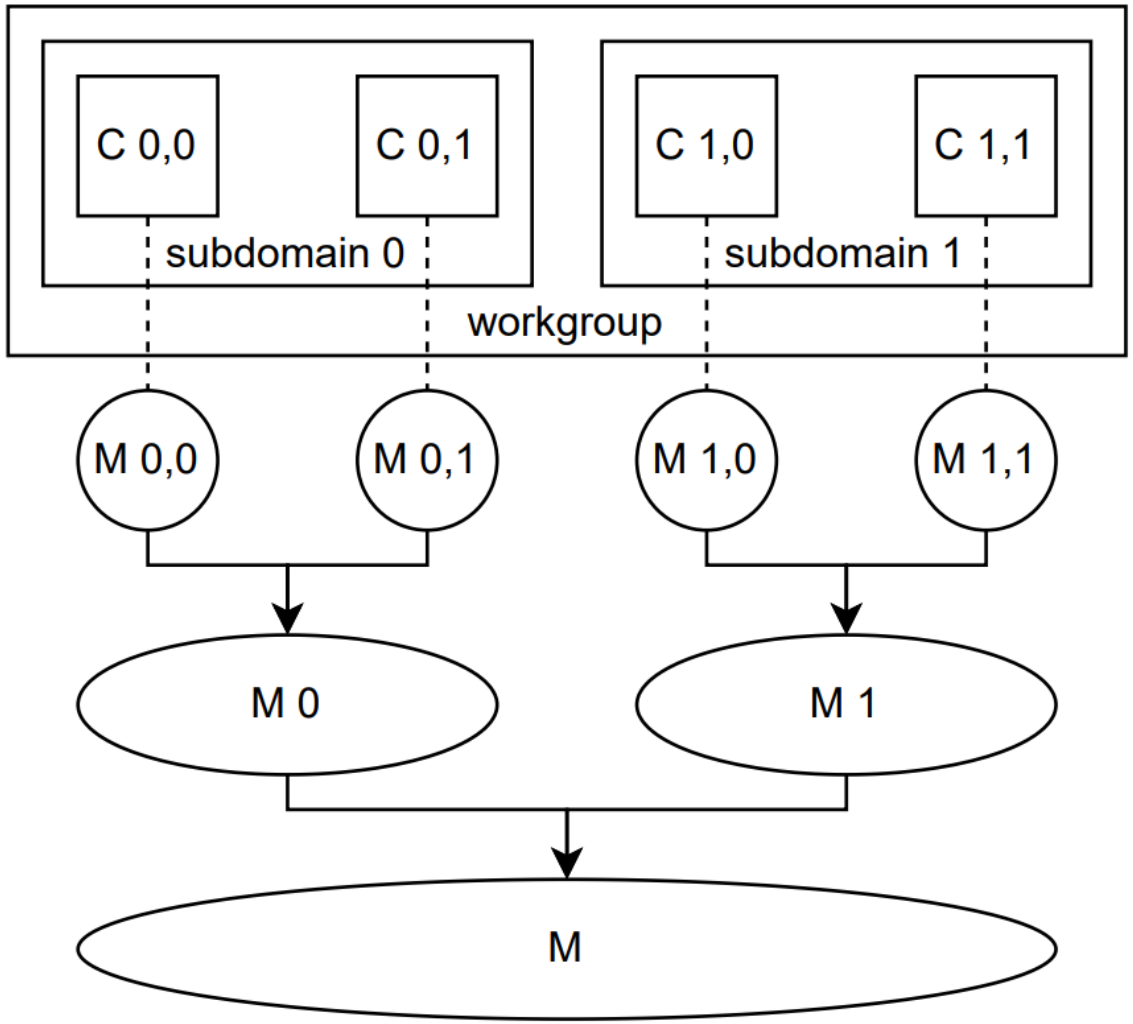
- Pipelining, also with CIRCT pipeline dialect
- Communication state machine optimization
- Better control over QoR
- Automatic device resource sharing



# CNM

- “Workgroup” compute model
- Logical memory hierarchy
- Explicit data residency and movement
- Natural concurrency



# CNM

- Restricted buffer usage
- Explicit outlining / “SSA enclaves”
- Device-specific per-CU code
- Workgroup-relative logical addressing

```
%wg = cnm.workgroup [8 2] { cnm.physical_dims = ["dpu", "
    ↪ thread"] }
%A_buf = cnm.alloc() for %wg { cnm.physical_space = "global" }
    : !cnm.buffer<16x16xi16, level 0> for !cnm.workgroup<8x2>
...
%sc_a_token = cnm.scatter %A_tile into %A_buf[#scatter_map] of
    ↪ %wg
    : tensor<128x32xi16> into !cnm.buffer<16x16xi16, level 0>
        ↪ of !cnm.workgroup<8x2>
...
%e_token = cnm.launch %wg (%A_buf, %B_buf, %C_buf: !cnm.buffer
    ↪ <16x16xi16, level 0>, ...) {
  ^bb0(%A_space: memref<16x16xi16>, %B_space: ...):
      scf.for %arg2 = 0 to %cst16 {
          %0 = memref.load %A_space[%arg0, %arg1] : memref
              ↪ <16x16xi16>
          %1 = memref.load %B_space[%arg1, %arg2] : memref
              ↪ <16x16xi16>
          ...
      }
} : !cnm.workgroup<8x2>
%C_tile, %g_token = cnm.gather %C_buf[#scatter_map] of %wg
    : !cnm.workgroup<8x2> into tensor<128x32xi16>
```

TECHNISCHE UNIVERSITÄT DRESDEN

CHAIR FOR COMPILER CONSTRUCTION

DRESDEN concept

cfaed

CENTER FOR ADVANCING ELECTRONICS DRESDEN