

High-level programming abstractions and compilation for near and in-memory computing

Jeronimo Castrillon

Chair for Compiler Construction (CCC), TU Dresden,

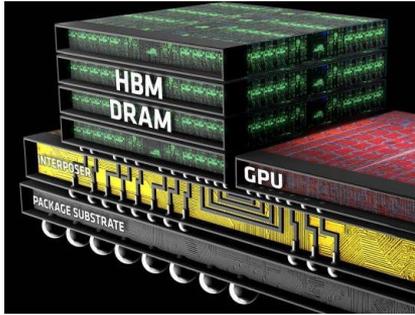
SCADS.AI Dresden/Leipzig & Center for Advancing Electronics (cfaed) Dresden

2nd Minisymposium on Applications and Benefits of UPMEM commercial Massively Parallel Processing-In-Memory Platform (ABUMPIMP 2024) @ Euro-Par 2024

August 26, 2024

Madrid, Spain

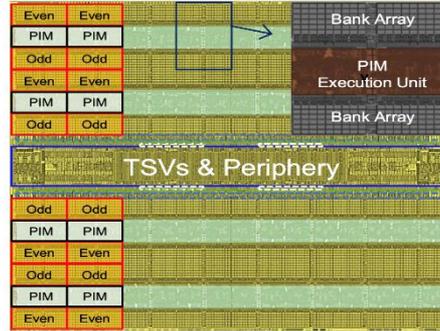
Emerging systems: Examples



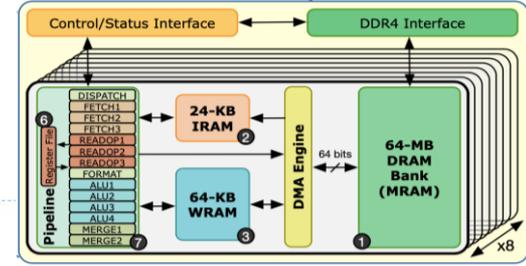
Source: AMD, AnandTech

High-bandwidth memory

AI accelerators + Prog. logic



Source: Samsung

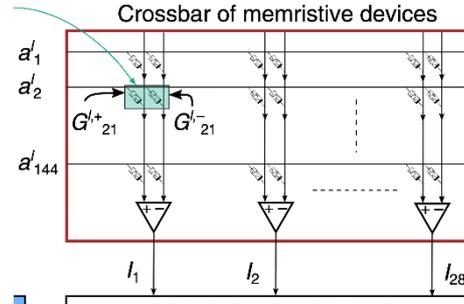
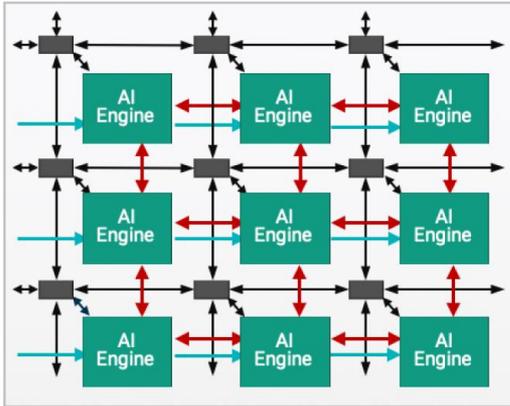


Source: UPMEM

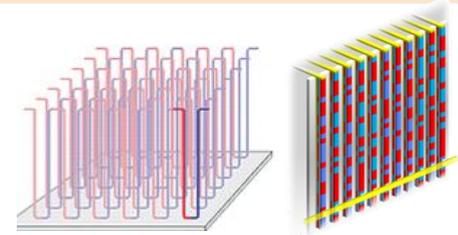
Near-memory computing



Source: AMD

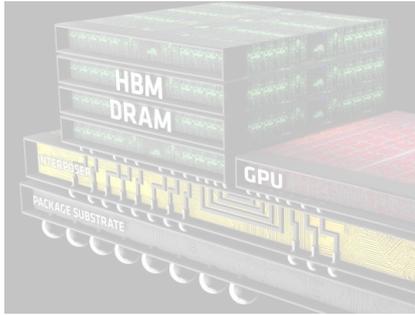


Source: IBM



Emerging memories + in-memory computing

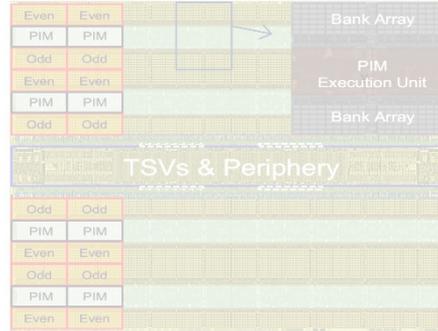
Emerging systems: Examples



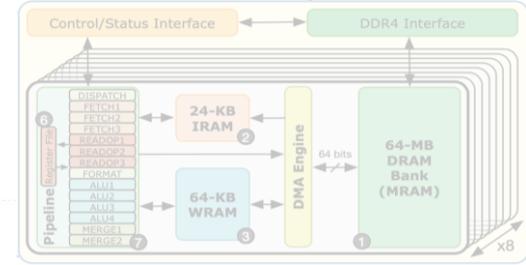
Source: AMD, AnandTech

High-bandwidth memory

AI accelerators + Prog. logic

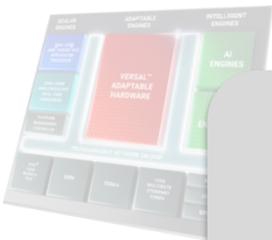


Source: Samsung



Source: UPMEM

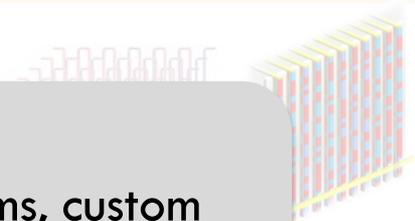
Near-memory computing



Source: AMD



Extreme heterogeneity, non Von Neumann paradigms, custom number representations, custom data mapping, complex APIs, ...

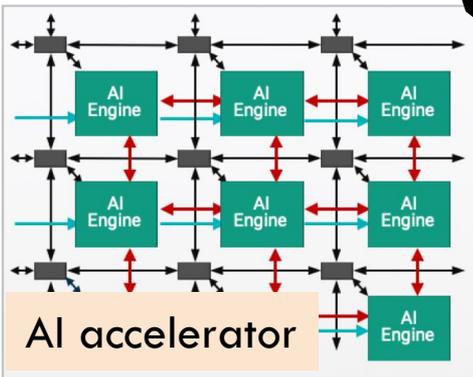


es +
 uting

Abstractions and compilation

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

What we want



```

1 void cfd_kernel(
2   double A[restrict 7][7],
3   double u[restrict 216][7][7][7],
4   double v[restrict 216][7][7][7])
5 {
6   /* element loop: */
7   for(int e = 0; e < 216; e++) {
8     for(int i0 = 0; i0 < 7; i0++) {
9       for(int j0 = 0; j0 < 7; j0++) {
10        for(int k0 = 0; k0 < 7; k0++) {
11          v[e][i0][j0][k0] = 0.0;
12          for(int i1 = 0; i1 < 7; i1++) {
13            for(int j1 = 0; j1 < 7; j1++) {
14              for(int k1 = 0; k1 < 7; k1++) {
15                v[e][i0][j0][k0] += A[i0][i1]
16                * A[j0][j1]

```

100X

???

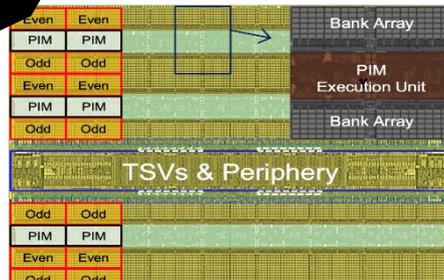
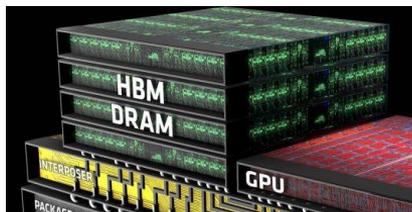
???

???

```

1 void cfd_kernel(
2   double A[restrict 7][7],
3   double u[restrict 216][7][7][7],
4   double v[restrict 216][7][7][7])
5 {
6   /* element loop: */
7   #pragma omp for
8   for (int e = 0; e < 216; e++) {
9     double t6[7][7][7];
10    /* 1st contraction: */
11    #pragma simd
12    for (int i0 = 0; i0 < 7; i0++) {
13      for (int i1 = 0; i1 < 7; i1++) {
14        /* #pragma simd */
15        for (int i2 = 0; i2 < 7; i2++) {
16          double t8 = 0.0;
17          for (int i3 = 0; i3 < 7; i3++)
18            t8 += A[i0][i3] * u[e][i1][i2][i3];
19          t6[i0][i1][i2] = t8;
20        } } /* end of 1st contraction */
21        double t7[7][7][7];
22        /* 2nd contraction: */
23        #pragma simd
24        for (int i4 = 0; i4 < 7; i4++) {
25          for (int i5 = 0; i5 < 7; i5++) {
26            /* #pragma simd */

```



Abstractions and compilation

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

What we want

```
1 void cfd_kernel(  
2     double A[restrict 7][7],  
3     double u[restrict 216][7][7][7],  
4     double v[restrict 216][7][7][7])  
5 {  
6     /* element loop: */  
7     for(int e = 0; e < 216; e++) {  
8         for(int i0 = 0; i0 < 7; i0++) {  
9             for(int j0 = 0; j0 < 7; j0++) {  
10                for(int k0 = 0; k0 < 7; k0++) {  
11                    v[e][i0][j0][k0] = 0.0;  
12                    for(int i1 = 0; i1 < 7; i1++) {  
13                        for(int j1 = 0; j1 < 7; j1++) {  
14                            for(int k1 = 0; k1 < 7; k1++) {  
15                                v[e][i0][j0][k0] += A[i0][i1]
```

100X

```
1 void cfd_kernel(  
2     double A[restrict 7][7],  
3     double u[restrict 216][7][7][7],  
4     double v[restrict 216][7][7][7])  
5 {  
6     /* element loop: */  
7     #pragma omp for  
8     for (int e = 0; e < 216; e++) {  
9         double t6[7][7][7];  
10        /* 1st contraction: */  
11        #pragma simd  
12        for (int i0 = 0; i0 < 7; i0++) {  
13            for (int i1 = 0; i1 < 7; i1++) {  
14                /* #pragma simd */  
15                for (int i2 = 0; i2 < 7; i2++) {  
16                    double t8 = 0.0;  
17                    for (int i3 = 0; i3 < 7; i3++)  
18                        t8 += A[i0][i3] * u[e][i1][i2][i3];  
19                    t6[i0][i1][i2] = t8;  
20                } } /* end of 1st contraction */  
21                double t7[7][7][7];  
22                /* 2nd contraction: */  
23                #pragma simd  
24                for (int i4 = 0; i4 < 7; i4++) {  
25                    for (int i5 = 0; i5 < 7; i5++) {  
26                        /* #pragma simd */
```

Need for higher-level programming abstractions and next-gen compilers as well as novel computational and costs models for emerging accelerators

AI acc

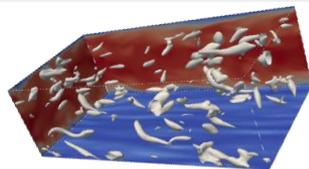
HBM+FPGA

Near-memory computing

Abstractions

Abstractions: Tensor expressions (Physics, ML)

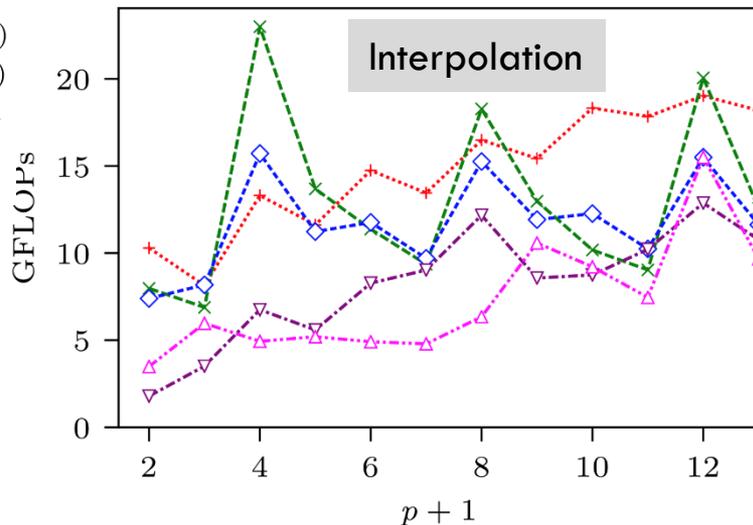
CFDlang



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

```
source = ...
var input A : matrix      &
var input u : tensorIN    &
var input output v : tensorOUT &
var input alpha : []      &
var input beta : []      &
v = alpha * (A # A # A # u .
  [[5 8] [3 7] [1 6]]) + beta * v
```

- + CFDlang(outer)
- x CFDlang(inner)
- ◇ hand-optimized
- ▽ DGEMM
- △ specialized



N. A. Rink, et al. "CFDlang: High-level code generation for high-order fluid dynamics". RWDSL'18.

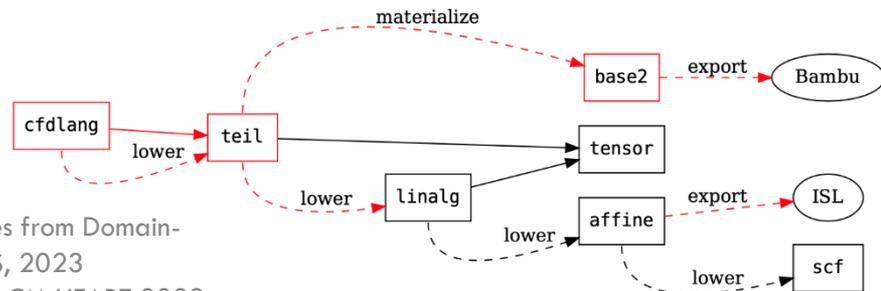
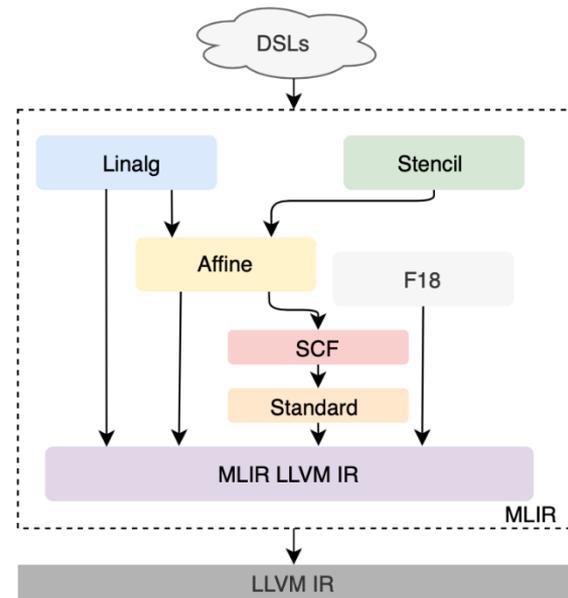
... meta-programming for cross-domain tensor ... E'18, 79-92.

... J. Castrillon. "Tell: a type-safe imperative ... Language", ARRAY'19, pp. 57-68

Plenty of other great DSL examples, e.g., Spiral, TACO, Halide, Lift, Firedrake, ML frameworks, ...

Tensor intermediate language (TeLL) in MLIR

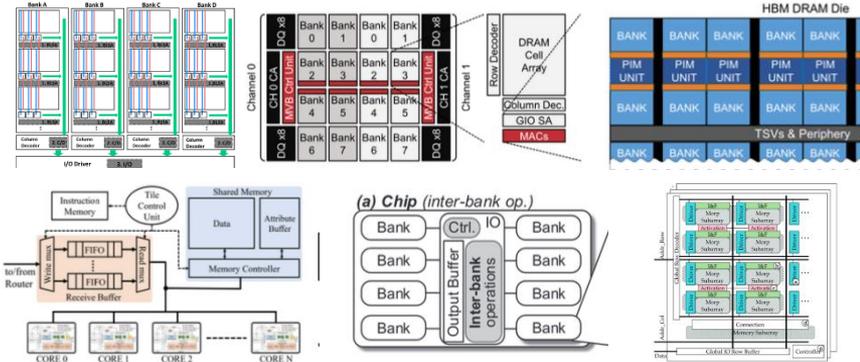
- MLIR: Multi-level intermediate reps.
 - Define own abstractions and transforms
 - Progressive lowering (and raising)
- TeLL: Primitive ops instead of index maps
 - Easier to express identities (big-O trfs)
 - Uses symbolic math, infinite precision
 - Lowers to SW and/or HW (with custom number representations)



S. Soldavini, et al. "Automatic Creation of High-Bandwidth Memory Architectures from Domain-Specific Languages: The Case of Computational Fluid Dynamics". In: ACM TRET, 2023
K. F. A. Friebel, J. Bi, J. Castrillon, "BASE2: An IR for Binary Numeral Types", In ACM HEART 2023

Compiling for near/in- memory

□ Lots in and near memory systems!



□ Commonalities

- Hierarchical HW
- Common high-level operations

A. Khan et al, "CINM (Cinnamon): A Compilation Infrastructure for Heterogeneous Compute In-Memory and Compute Near-Memory Paradigms", ASPLOS'25

The Landscape of Compute-near-memory and Compute-in-memory: A Research and Commercial Overview

ASIF ALI KHAN, TU Dresden, Germany
 JOÃO PAULO C. DE LIMA, TU Dresden and ScaDS.AI, Germany
 HAMID FARZANEH, TU Dresden, Germany
 JERONIMO CASTRILLON, TU Dresden and ScaDS.AI, Germany

In today's data-centric world, where data fuels numerous application domains, with machine learning at the

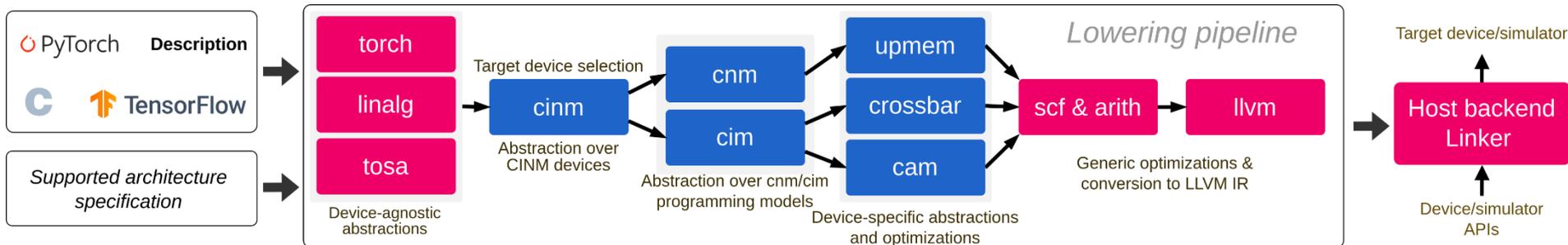
2024

A. Khan, et al "The Landscape of Compute-near-memory and Compute-in-memory: A Research and Commercial Overview." arXiv:2401.1442 (2024)

Operation	Type
<code>cinm.{add,sub,mul,div,min,max}(%lhs, %rhs)</code>	$T \times T \rightarrow T$
<code>cinm.{and,or,xor,not}(%lhs, %rhs)</code>	$T \times T \rightarrow T$
<code>cinm.gemv(%lhs, %rhs)</code>	$S^{m \times n} \times S^n \rightarrow S^m$
<code>cinm.gemm(%lhs, %rhs)</code>	$S^{m \times k} \times S^{k \times n} \rightarrow S^{m \times n}$
<code>cinm.transpose(%in, %perms)</code>	$S^n \times N^n \rightarrow S'$
<code>cinm.{histogram,majority}(%in)</code>	$S^n \rightarrow S^k$
<code>cinm.topk(%in, %k)</code>	$S^n \times N \rightarrow S^k \times N^k$
<code>cinm.simSearch #E, #k (%in1, %in2)</code>	$E \times N^k \times S^n \times S^n \times N \rightarrow S^k$
<code>cinm.mergePartial #op #dir (%lhs, %rhs)</code>	$E \times D \times T \times T \rightarrow T$
<code>cinm.popCount(%in)</code>	$T \rightarrow N$
<code>cinm.reduce #op (%in)</code>	$E \times S^n \rightarrow S$
<code>cinm.scan #op (%in)</code>	$E \times S^n \rightarrow S^n$

CINM: Generalized MLIR infrastructure

- ❑ From linear algebra abstractions (common to ML frameworks and beyond)
- ❑ Intermediate languages for **in and near memory computing**
- ❑ **Pattern recognition, target-specific models and optimizations**

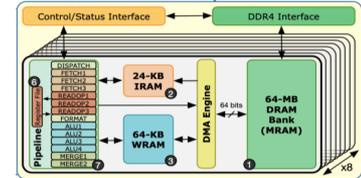


A. Khan et al, "CINM (Cinnamon): A Compilation Infrastructure for Heterogeneous Compute In-Memory and Compute Near-Memory Paradigms", ASPLOS'25
H. Farzaneh et al. "C4CAM: A Compiler for CAM-based In-memory Accelerators", ASPLOS'24



UPMEM example: Matmult

```
BARRIER_INIT(my_barrier, NR_TASKLETS);
int main() {
    ...
    barrier_wait(&my_barrier);
    int32_t point_per_tasklet = (ROWS*COLS)/NR_TASKLETS;
    uint32_t mram_base_addr_A = (uint32_t) (DPU_MRAM_HEAP_POINTER );
    uint32_t mram_base_addr_B = (uint32_t) (DPU_MRAM_HEAP_POINTER + ROWS * COLS *
↪ sizeof(T));
    uint32_t mram_base_addr_C = (uint32_t) (DPU_MRAM_HEAP_POINTER + 2 * ROWS * COLS
↪ * sizeof(T));
    for(int i = (tasklet_id * point_per_tasklet) ; i < (
↪ (tasklet_id+1)*point_per_tasklet ) ; i++) {
        if( new_row != row ){
            ...
            mram_read((__mram_ptr void const*) (mram_base_addr_A + mram_offset_A),
↪ cache_A, COLS * sizeof(T));
        }
        mram_read((__mram_ptr void const*) (mram_base_addr_B + mram_offset_B),
↪ cache_B, COLS * sizeof(T));
        dot_product(cache_C, cache_A, cache_B, number_of_dot_products);
        ...
    }
    ...
    mram_write( cache_C, (__mram_ptr void *) (mram_base_addr_C + mram_offset_C),
↪ point_per_tasklet * sizeof(T));
}
```



UPMEM example: Matmult

```
def mm(int32(64, 64) A, int32(64, 64) B) -> (int32(64, 64) C) {
  C(i,j) += A(i,k) * B(k,j)
  where i in 0:64, k in 0:64, j in 0:64
}
```

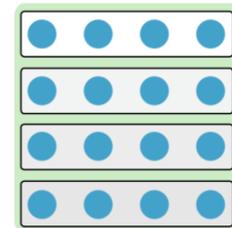
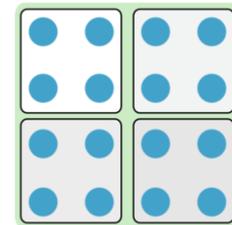
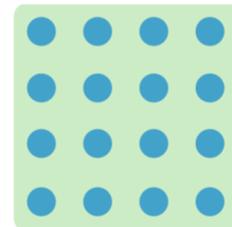


```
%3 = cinm.op.gemm %0, %1 : (tensor<64x64xi32>, tensor<64x64xi32>) -> tensor<64x64xi32>
```

Tiling to fit on hardware



```
%2 = affine.for %arg0 = 0 to 64 step 16 iter_args(%arg1 = %0) -> (tensor<64x64xi32>) {
  %3 = affine.for %arg2 = 0 to 64 iter_args(%arg3 = %arg1) -> (tensor<64x64xi32>) {
    %A_slice = tensor.extract_slice %A[%arg0, 0] [16, 64] [1, 1] : ...
    %B_slice = tensor.extract_slice %0[%arg4, %arg2] [64, 1] [1, 1] : ...
    %res = cinm.op.gemm %A_slice, %B_slice
      : (tensor<16x64xi32>, tensor<64x1xi32>) -> tensor<16x1xi32>
    %t = tensor.insert_slice %res into %C[%arg0, %arg2] [16, 1] [1, 1] : ...
    ...
  }
  affine.yield %3 : tensor<64x64xi32>
}
```



UPMEM example: Matmult

```
%2 = affine.for %arg0 = 0 to 64 step 16 iter_args(%arg1 = %0) -> (tensor<64x64xi32>) {  
  %3 = affine.for %arg2 = 0 to 64 iter_args(%arg3 = %arg1) -> (tensor<64x64xi32>) {  
    ... // extract slices  
    %5 = cinm.op.gemm %A_slice, %B_slice  
      : (tensor<16x64xi32>, tensor<64x1xi32>) -> tensor<16x1xi32>  
    ... // insert slice back  
  }  
  affine.yield %3 : tensor<64x64xi32>  
}
```



cinm-to-cnm

```
%wg = cnm.workgroup : !cnm.workgroup<1x16x1>  
... // input preparation  
%A_buf = cnm.alloc() for %wg : !cnm.buffer<64xi32 on 1x16x1>  
%B_buf = cnm.alloc() for %wg : !cnm.buffer<64xi32 on 1x16x1>  
%C_buf = cnm.alloc() for %wg : !cnm.buffer<i32 on 1x16x1>  
cnm.scatter %A_slice into %A_buf[#map] of %wg : ... // filling buffers on devices  
cnm.scatter %B_slice into %B_buf[#map1] of %wg : ... // filling buffers on devices  
cnm.launch %wg in(%A_buf, %B_buf : ...) out(%C_buf : ...) {  
  ^bb0(%row: memref<64xi32>, %col: memref<64xi32>, %res: memref<i32>):  
    linalg.reduce ... // %res += %row * %col  
}  
cnm.gather %C_buf[#map2] of %wg into %C_slice : ...  
cnm.free_workgroup %wg : ...
```

Host code

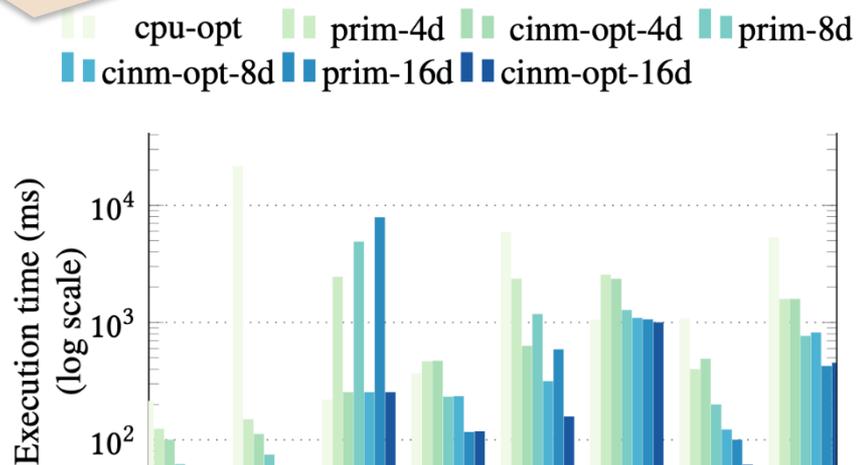
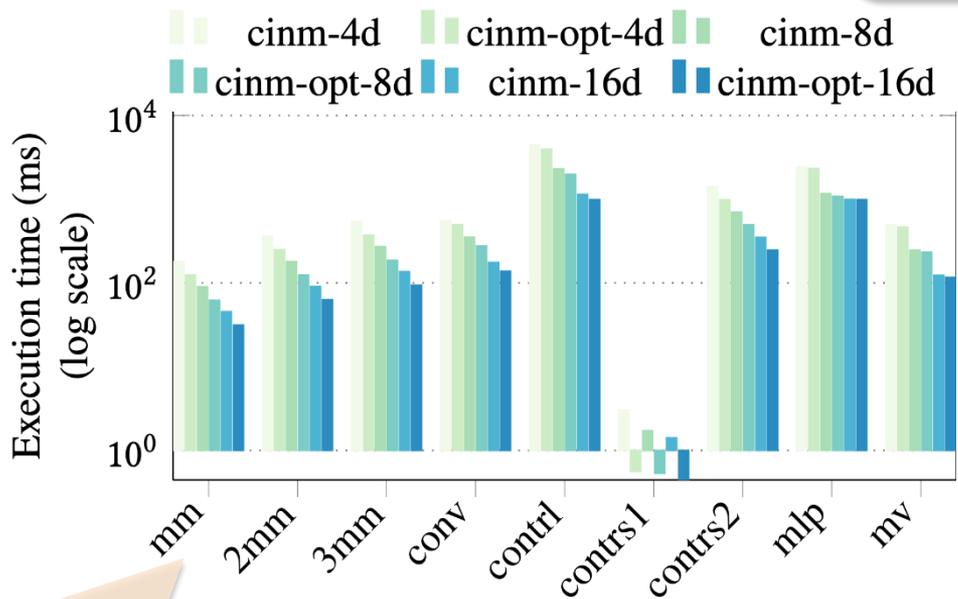
```
func.func @main() {  
  ...  
  %1 = upmem.alloc_dpus : !upmem.hierarchy<1x16x1>  
  scf.for %arg0 = %c0 to %c64 step %c16 {  
    scf.for %arg1 = %c0 to %c64 step %c1 {  
      upmem.scatter %subview[264, 64, #map] onto %1 : ...  
      upmem.scatter %alloc_0[8, 64, #map1] onto %1 : ...  
      upmem.scatter %0[0, 1, #map2] onto %1 : ...  
      upmem.launch_func @dpu_kernels::@main %1 : ...  
      upmem.gather %alloc_1[0, 1, #map2] from %1 : ...  
      ...  
    }  
  }  
  upmem.free_dpus %1 : !upmem.hierarchy<1x16x1>  
  return  
}
```

Device code

```
upmem.module @dpu_kernels {  
  upmem.func @main()  
    attributes {num_tasklets = 1 : i64} {  
    ...  
    upmem.memcpy mram_to_wram ...  
    upmem.memcpy mram_to_wram ...  
    scf.for %arg0 = 0 to 64 {  
      %6 = memref.load %1[%arg0] : memref<64xi32>  
      %7 = memref.load %3[%arg0] : memref<64xi32>  
      %8 = memref.load %5[] : memref<i32>  
      %9 = arith.muli %6, %7 : i32  
      %10 = arith.addi %9, %8 : i32  
      memref.store %10, %5[] : memref<i32>  
    }  
    upmem.memcpy wram_to_mram ...  
    upmem.return  
  }  
}
```

UPMEM example: Results

Manual designs 2-5x faster than CPU. **CINM 1.5-2x faster than hand-optimized code**



Optimizations achieve **40%-50% speedup** (geomean)

A. A. Khan, et al. "CINM (Cinnamon): A Compilation Infrastructure for Heterogeneous Compute In-Memory and Compute Near-Memory Paradigms", ASPLOS'25

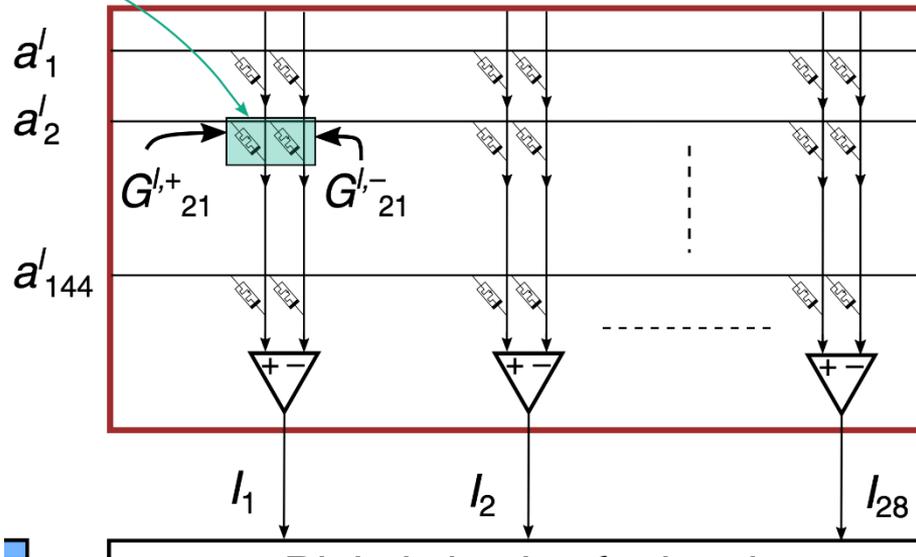
10x-40x less lines of code



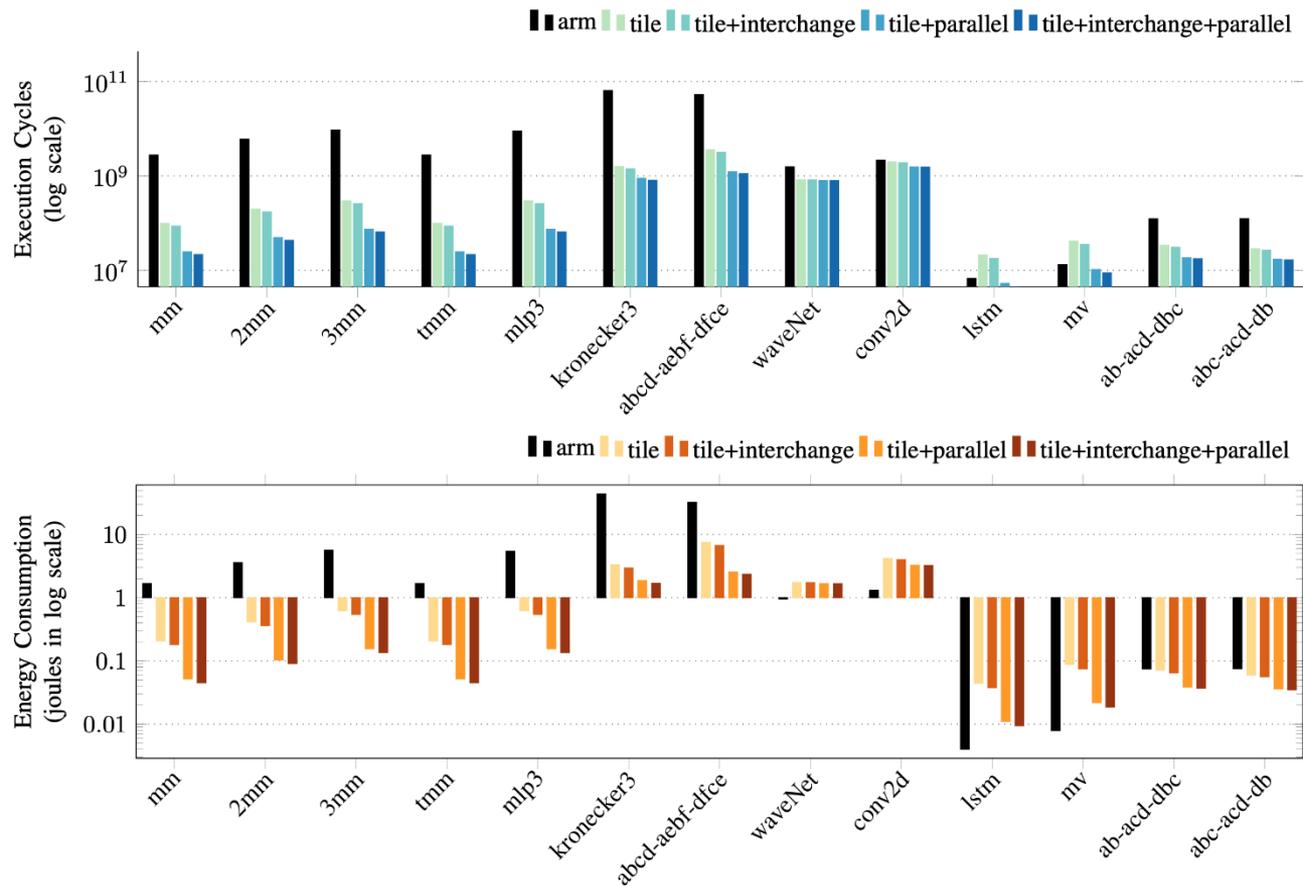
CIM: Lowering examples

```
def contr(int16(K,L,M) A, int16(L,K,N) B)
  -> (int16(M,N) C)
{
  C(m,n) += A(k,l,m) * B(l,k,n)
}
```

Crossbar of memristive devices

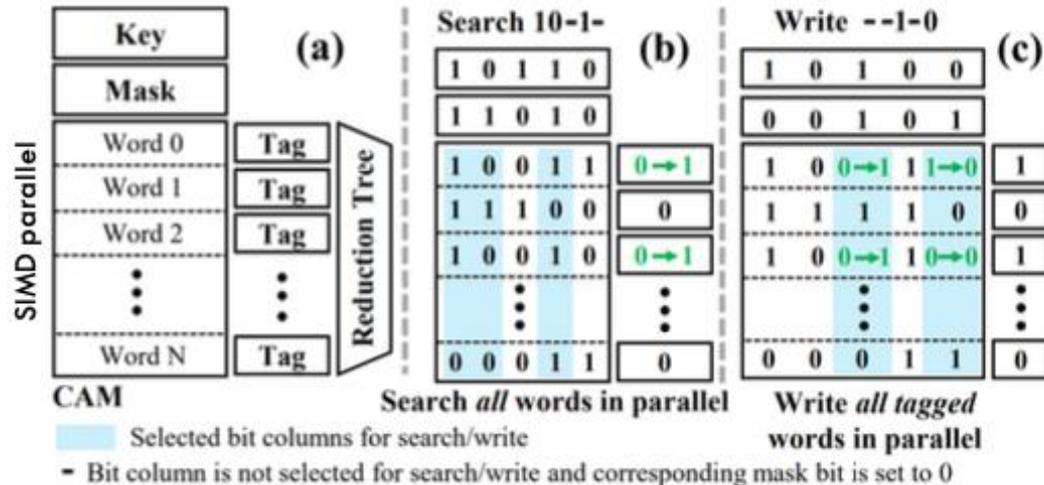
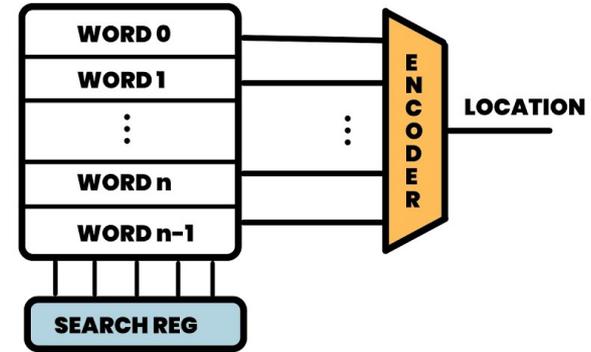


Optimization results: Crossbars beyond matmult



Content-addressable memories (CAMs) and Processors

- Emerging Content-addressable memories (CAMs) and associative processors (APs)
- APs: Different programming paradigm based on **search** and **write**



ISCA'20

CINM: Lowering for different CAM-based acc.



```
...
dist = torch.frobenius_norm(...)
s, v = torch.topk(...)
...
```

```
...
%v:2 = cim.execute(...) ({
  %t1 = cim.norm (...)
  %t2, %t3 = cim.topk (...)
  cim.yield (...)
})
...
```

```
...
// %arg1: bank, %arg2: mat
// %arg3: array, %arg4: sub-array
scf.parallel (%arg1) = ... {
  ...
  scf.parallel (%arg4) = ... {
    ...
    cam.search best eucl (...)
    %t3:2 = cam.read best (...)
    %t4 = cam.merge_partial_subarray
    ...
  } ...
} ...
```

MC

```
...
%t1= cim.matmul (...)
%t2 = cim.sign (...)
scf.parallel (%arg1) = ... {
  ...
  scf.parallel (%arg4) = ... {
    ...
    cam.search best eucl (...)
    %t3:2 = cam.read best (...)
    %t4 = cam.merge_partial_subarray (...)
    ...
  } ...
} ...
```

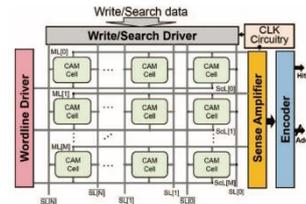
TCAM

```
...
%t1= cim.matmul (...)
%t2 = cim.sign (...)
scf.parallel (%arg1) = ... {
  ...
  scf.parallel (%arg4) = ... {
    ...
    scf.for (%arg5) = ... { //selective search
      ...
      cam.search best eucl (...)
      %t3:2 = cam.read best (...)
      %t4 = cam.merge_partial_subarray (...)
    } ...
  } ...
} ...
```

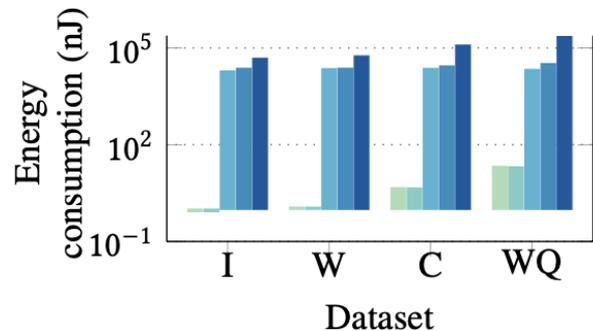
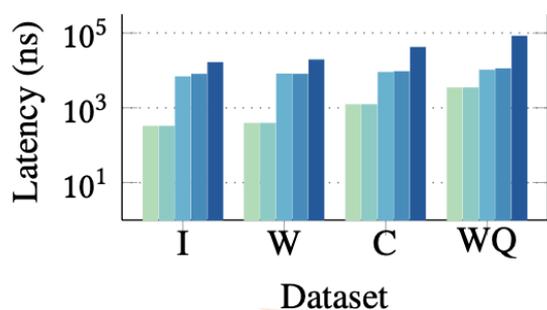
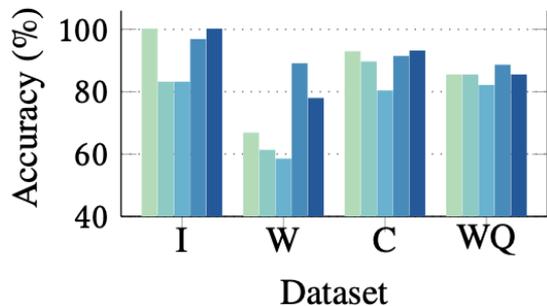
Selective search

Content addressable memories (CAMs)

- ❑ NVM-based CAMs: Great for **KNNs**, One-shot learning, ...
- ❑ CINM support for **similarity** and **CAM arch exploration**
- ❑ Automatic flow from TorchScript **matches manual designs**



■ C4CAM-3b
 ■ C4CAM-2b
 ■ C4CAM-1b+LSH
 ■ Cosine-GPU
 ■ Euclidean-GPU

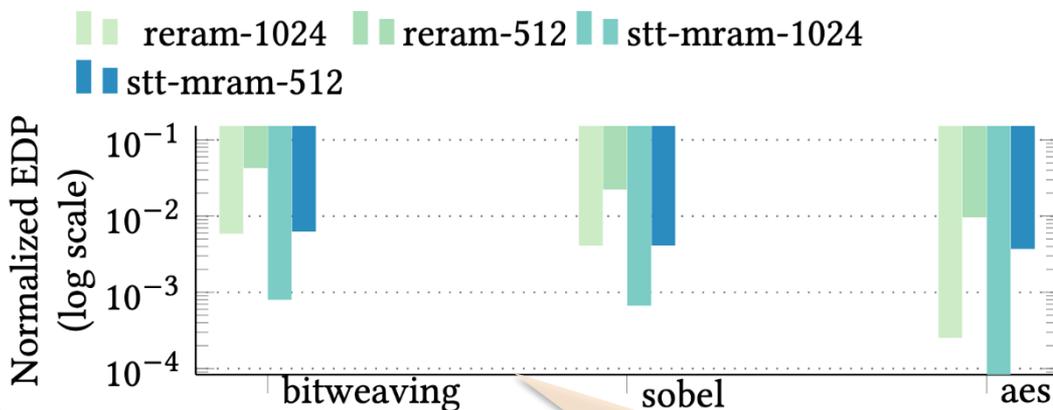
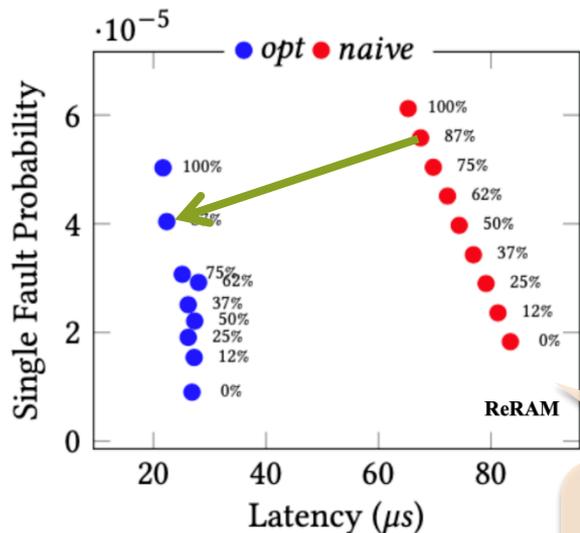


H. Farzaneh, et al. "C4CAM: A Compiler for CAM-based In-memory Accelerators", ASPLOS, 2024

KNN results (128x128 CAM): **14x faster and ~10⁴ less energy** compared to GPU

Logic-in-memory in NVMs

- ❑ Massively parallel multi-operand bit-wise operations in-memory
- ❑ Complex mapping of operands, operations and temporaries to columns



Optimized mapping: **Less latency (3x), better reliability (~1.4x)**

Orders of magnitude better EDP vs CPU baseline

H. Farzaneh, et al. "SHERLOCK: Scheduling Efficient and Reliable Bulk Bitwise Operations in NVMs", DAC 2024

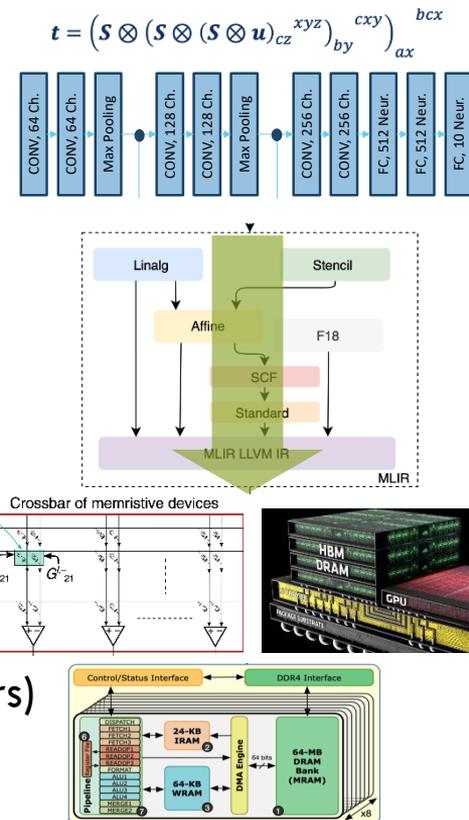
Closing

Automatic optimization for heterogeneous in-memory computing

- Domain-specific abstractions
- Abstract primitives, compute model, trade-offs
- Safe and correct optimization with high-level compilers

Challenges

- Still work on (formal) modeling primitives
- Simulators, prototypes in interdisciplinary research efforts
- From **algorithmic** abstractions to **geometry** (device parameters)



Thanks! & Acknowledgements



..., and previous members of the group (Norman Rink, Sven Karol, Sebastian Ertel, Andres Goens), and collaborators (J. Fröhlich, I. Sbalzarini, A. Cohen, T. Grosser, T. Hoefler, H. Härtig, H. Corporaal, C. Pilato, S. Parkin, P. Jääskeläinen, J-J. Chen, A. Jones, X.S. Hu, M. Niemier, M. Tahoori)

Funded by
DFG



CO4RTM (450944241)
HetCIM (502388442)



BMBF (01IS18026A-D)



STAATSMINISTERIUM
FÜR WISSENSCHAFT
KULTUR UND TOURISMUS



[**RWDSL'18**] N. A. Rink, et al. "CFDlang: High-level code generation for high-order methods in fluid dynamics". RWDSL'18.

[**GPCE'18**] A. Susungi, et al. "Meta-programming for cross-domain tensor optimizations" GPCE'18, 79-92.

[**Array'19**] N.A. Rink, N. A. and J. Castrillon. "Tell: a type-safe imperative Tensor Intermediate Language", ARRAY'19, pp. 57-68.

[**TRETS'23**] S. Soldavini, et al. "Automatic Creation of High-Bandwidth Memory Architectures from Domain-Specific Languages: The Case of Computational Fluid Dynamics". In: ACM TRETS, Sept. 2022.

[**HEART'23**] K. F. A. Friebe, J. Bi, J. Castrillon, "BASE2: An IR for Binary Numeral Types", In ACM HEART 2023

[**CINM'25**] A. Khan et al, "CINM (Cinnamon): A Compilation Infrastructure for Heterogeneous Compute In-

Memory and Compute Near-Memory Paradigms", ASPLOS'25.

[**TCAD'21**] A. Siemieniuk, et al. "OCC: An Automated End-to-End Machine Learning Optimizing Compiler for Computing-In-Memory", IEEE TCAD, 2021.

[**LANDSCAPE'24**] A. Khan, et al "The Landscape of Compute-near-memory and Compute-in-memory: A Research and Commercial Overview." arXiv:2401.1442 (2024).

[**C4CAM'24**] H. Farzaneh, et al. "C4CAM: A Compiler for CAM-based In-memory Accelerators", ASPLOS 2024.

[**DAC'24**] H. Farzaneh, et al. "'SHERLOCK: Scheduling Efficient and Reliable Bulk Bitwise Operations in NVMs", DAC 2024.