CHAIR FOR COMPILER CONSTRUCTION

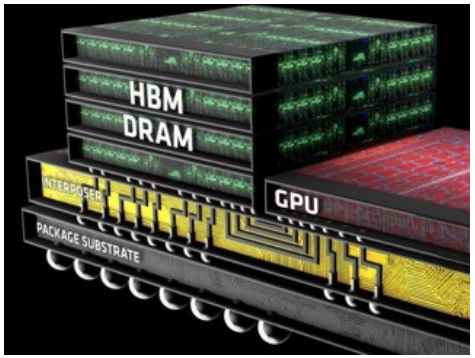
cfaed CENTER FOR ADVANCING ELECTRONICS DRESDEN

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

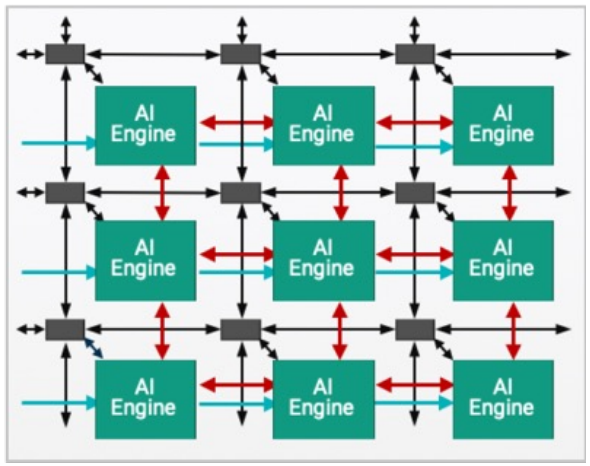


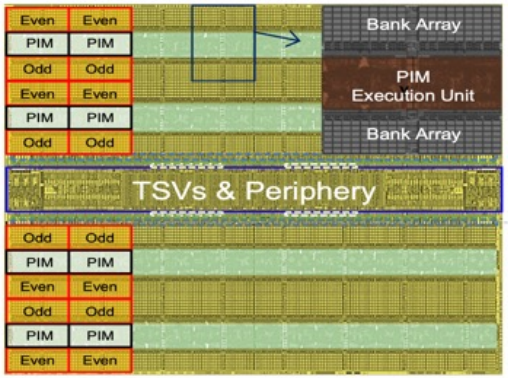
Source: AMD, AnandTech

High-bandwidth memory

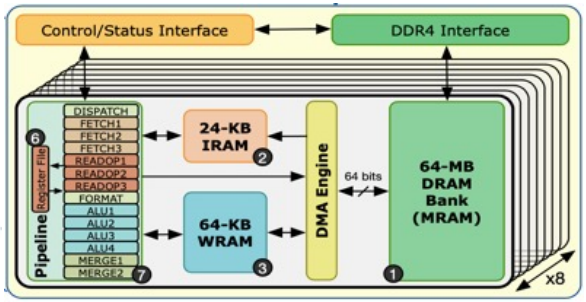
AI accelerators + Prog. logic

Source: AMD



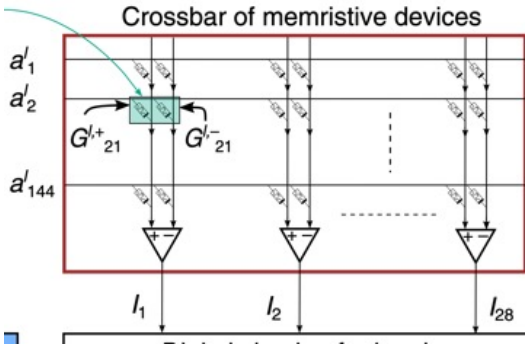


Source: Samsung

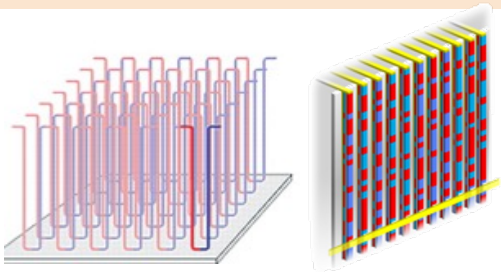


Source: UPMEM

Near-memory computing

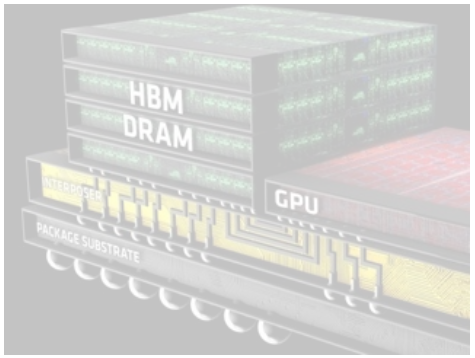


Source: IBM



Emerging memories + in-memory computing

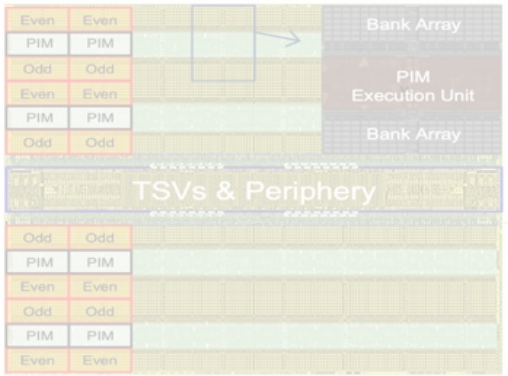
Emerging systems: Examples



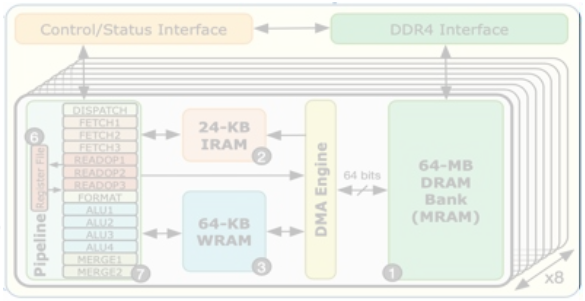
Source: AMD, AnandTech

High-bandwidth memory

AI accelerators + Prog. logic

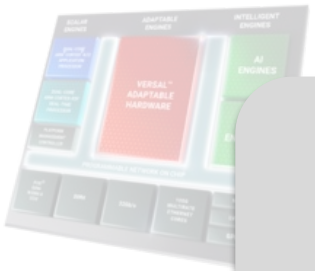


Source: Samsung

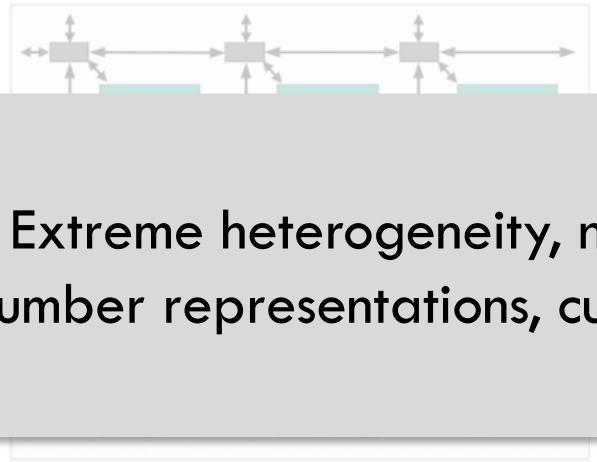


Source: UPMEM

Near-memory computing



Source: AMD



es +
uting

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

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

What we (naively) code

```
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 */
}
```

100X

What performance experts code

```
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 */
```

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

```
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]
```

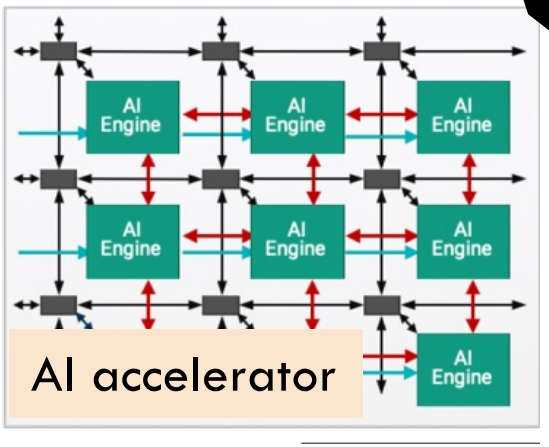
???

???

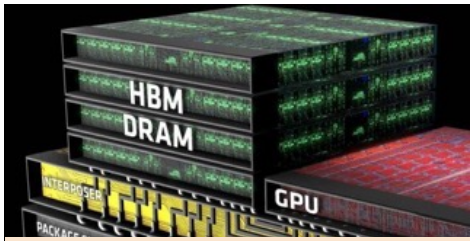
100X

:3:3

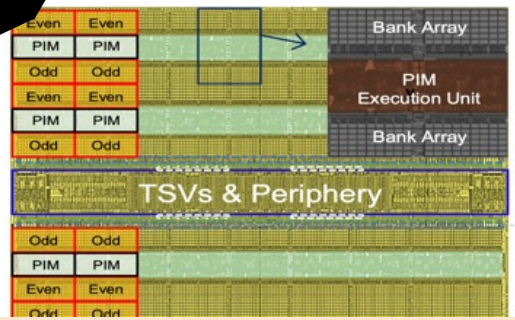
```
void cfd_kernel(
  double A[restrict 7][7],
  double u[restrict 216][7][7][7],
  double v[restrict 216][7][7][7])
{
  /* element loop: */
  #pragma omp for
  for (int e = 0; e < 216; e++) {
    double t6[7][7][7];
    /* 1st contraction: */
    #pragma simd
    for (int i0 = 0; i0 < 7; i0++) {
    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 */
```



AI accelerator



HBM+FPGA



Near-memory computing

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

```
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]
```

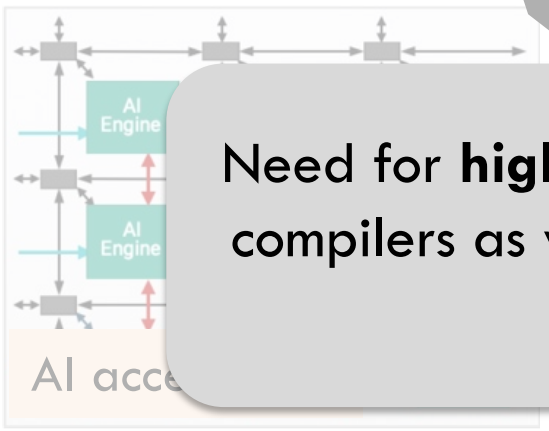
100X

???

???

???

```
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 */
```



AI acce

HBM+FPGA

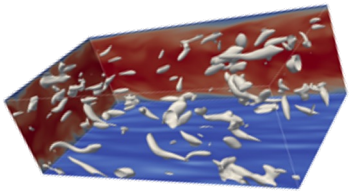
Near-memory computing

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

Abstractions

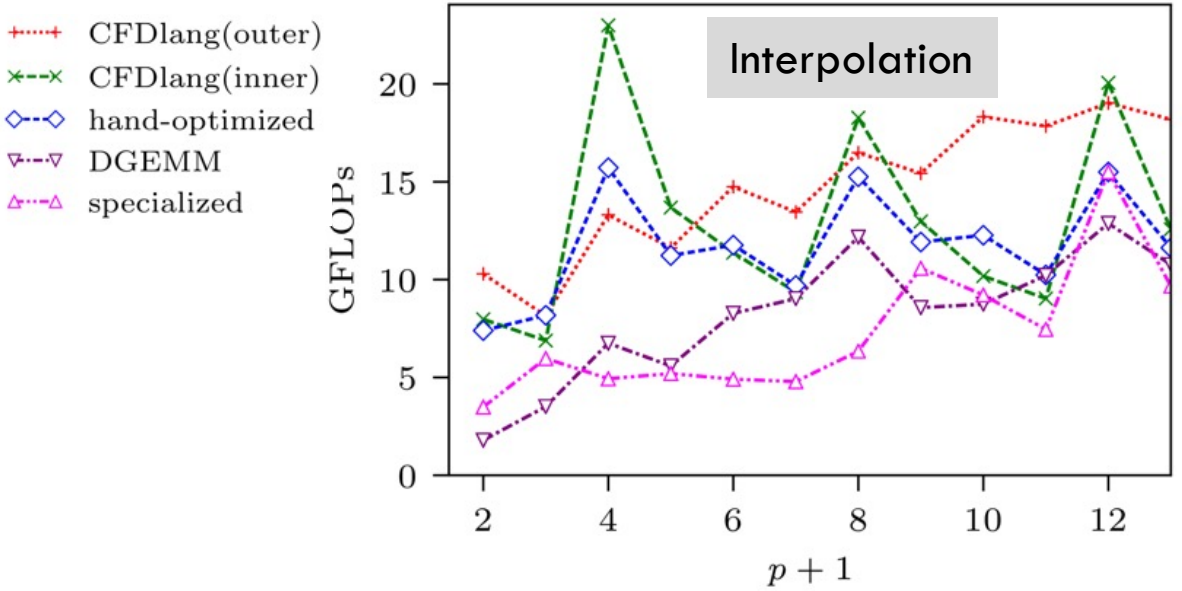
Abstractions: Tensor expressions (Physics, ML)

- 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
```



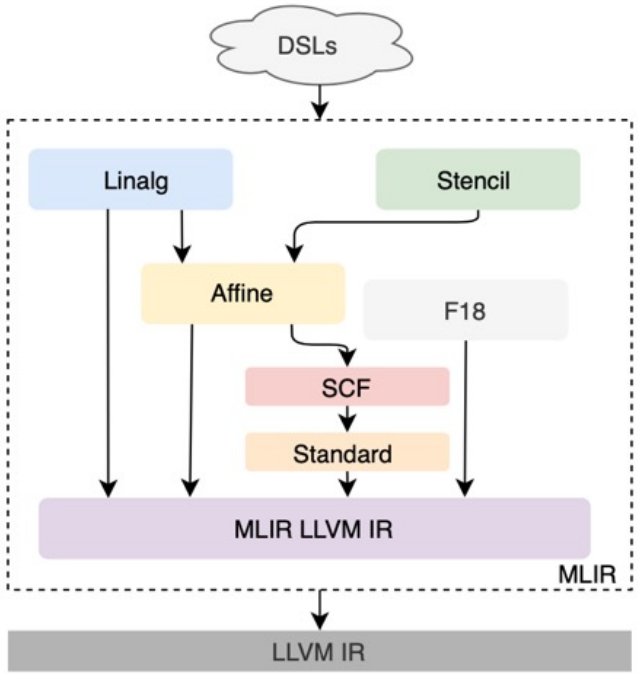
N. A. Rink, et al. "CFDlang: High-level code generation for high-
luid dynamics". RWDSL'18.
eta-programming for cross-domain tensor
E'18, 79-92.
J. Castrillon. "TeIL: a type-safe imperative
Language", ARRAY'19, pp. 57-68

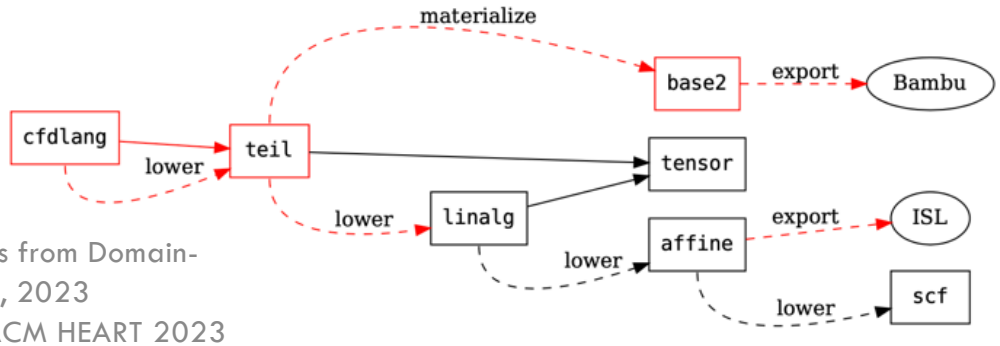
Plenty of other great DSL examples, e.g., Spiral, TACO, Halide, Lift, Firedrake, ML frameworks, …

Tensor intermediate language (TeIL) in MLIR

- MLIR: Multi-level intermediate reps.
  - Define own abstractions and transforms
  - Progressive lowering (and raising)

- TeIL: 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)



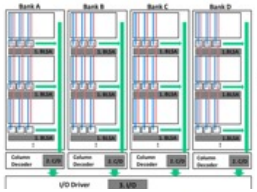
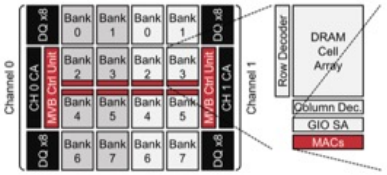


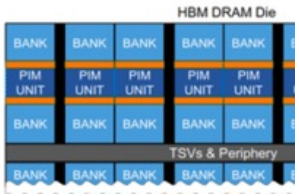
S. Soldavini, et al. “Automatic Creation of High-Bandwidth Memory Architectures from Domain-Specific Languages: The Case of Computational Fluid Dynamics”. In: ACM TRETS, 2023
K. F. A. Friebel, J. Bi, J. Castrillon, "BASE2: An IR for Binary Numeral Types", In ACM HEART 2023

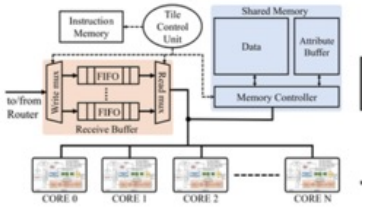
Compiling for near/in-memory

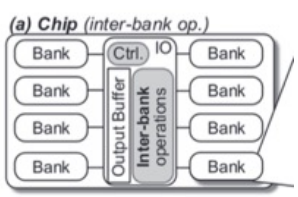
Colorful landscape

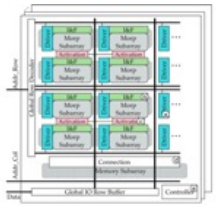
- Lots **in and near memory** systems!









- 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

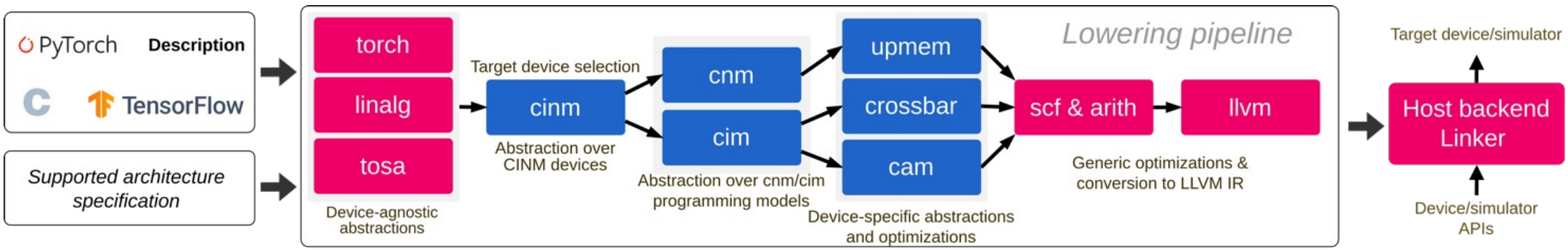
In today's data-centric world, where data fuels numerous application domains, with machine learning at the

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

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**

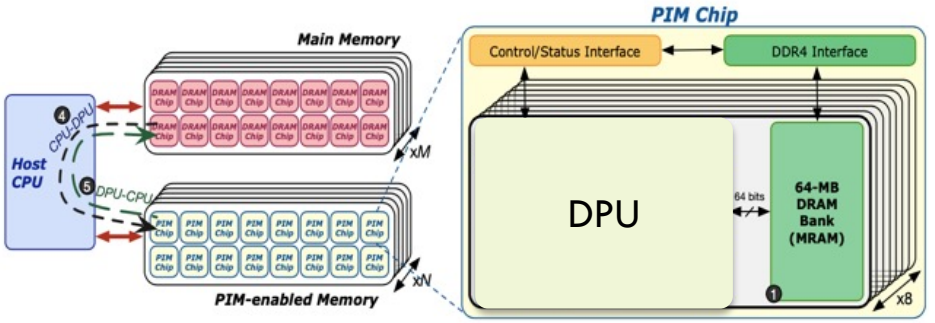


A. Khan et al, "CINM (Cinnamon): A Compilation Infrastructure for Heterogeneous Compute In-Memory and Compute Near-Memory Paradigms", ASPLOS'25
H. Farzaneh et al. "C4CAM: A Compiler for CAM-based In-memory Accelerators", ASPLOS'24

DISRUPTIVE MEMORY TECHNOLOGIES
DFG PRIORITY PROGRAM 2377

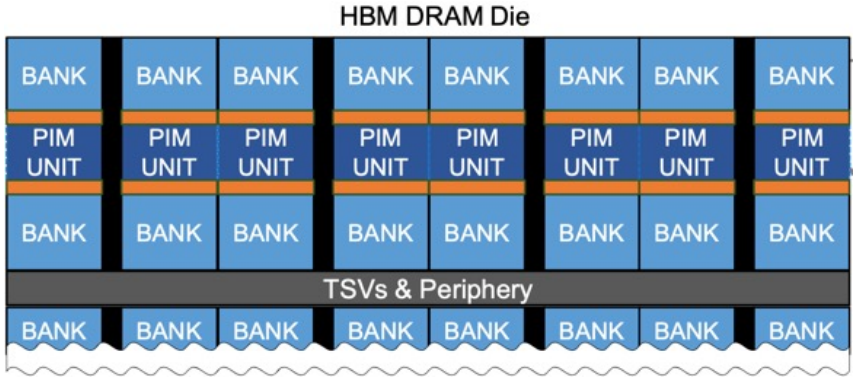
CNM architectures

- Commonalities
  - Many PIM units, fixed-/multi-function
  - Multiple address spaces
  - Can be synchronous/asynchronous



Gómez-Luna, Juan, et al.
arXiv:2105.03814 (2021)

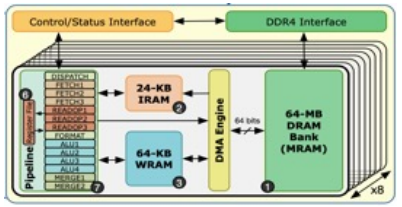
- Abstract programming model similar to GPUs: scatter, execute and gather



Lee et al., FIMDRAM, ISCA 2021

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

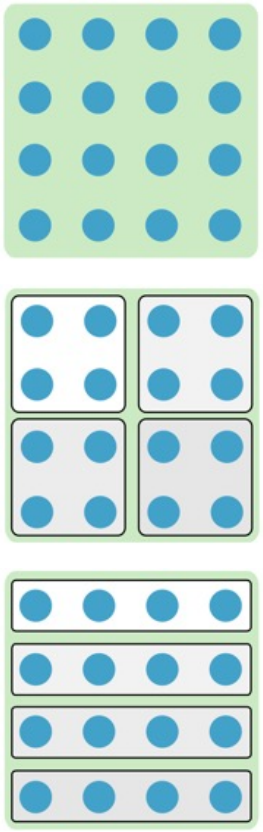
```
def mm(int32(64, 64) A, int32(64, 64) B) -> (int32(64, 64) C) {
  C(i,j) += A(i,k) * B(k,j)
            where i in 0:64, k in 0:64, j in 0:64
}
```



```
%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 : ...
```

UPMEM example: Matmult

Host code

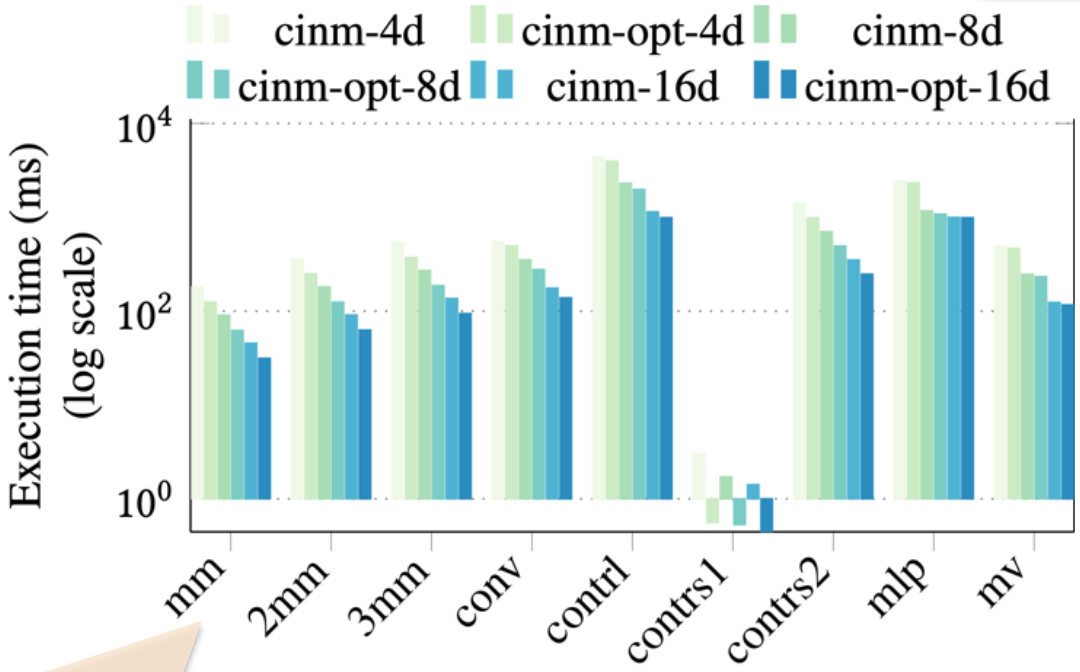
```
func.func @main() {
  ...
  %1 = upmem.alloc_dpus : !upmem.hierarchy<1x16x1>
  scf.for %arg0 = %c0 to %c64 step %c16 {
    scf.for %arg1 = %c0 to %c64 step %c1 {
      upmem.scatter %subview[264, 64, #map] onto %1 : ...
      upmem.scatter %alloc_0[8, 64, #map1] onto %1 : ...
      upmem.scatter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

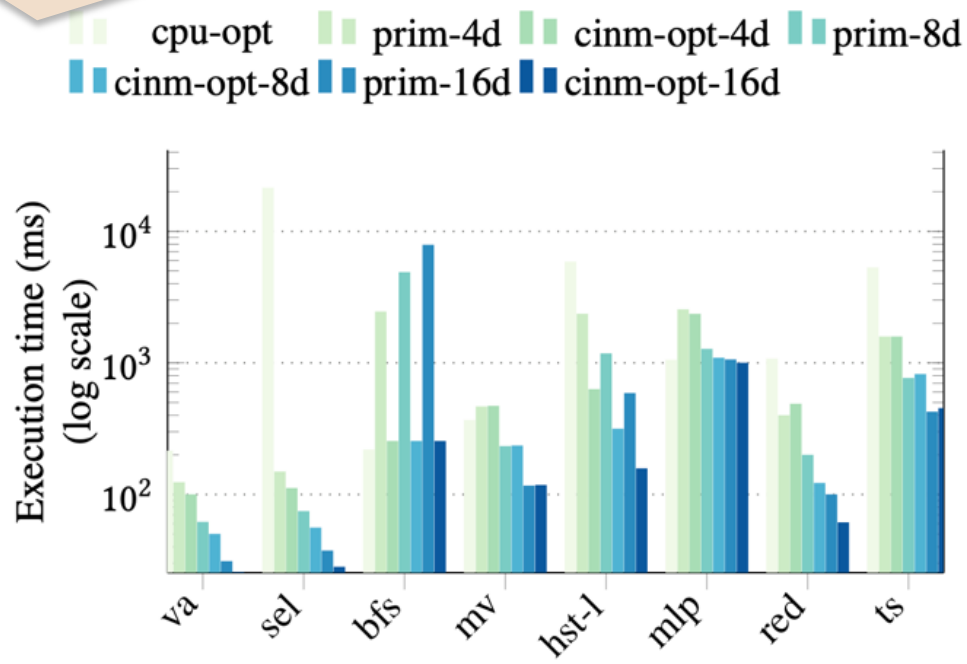
Device code

```
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**





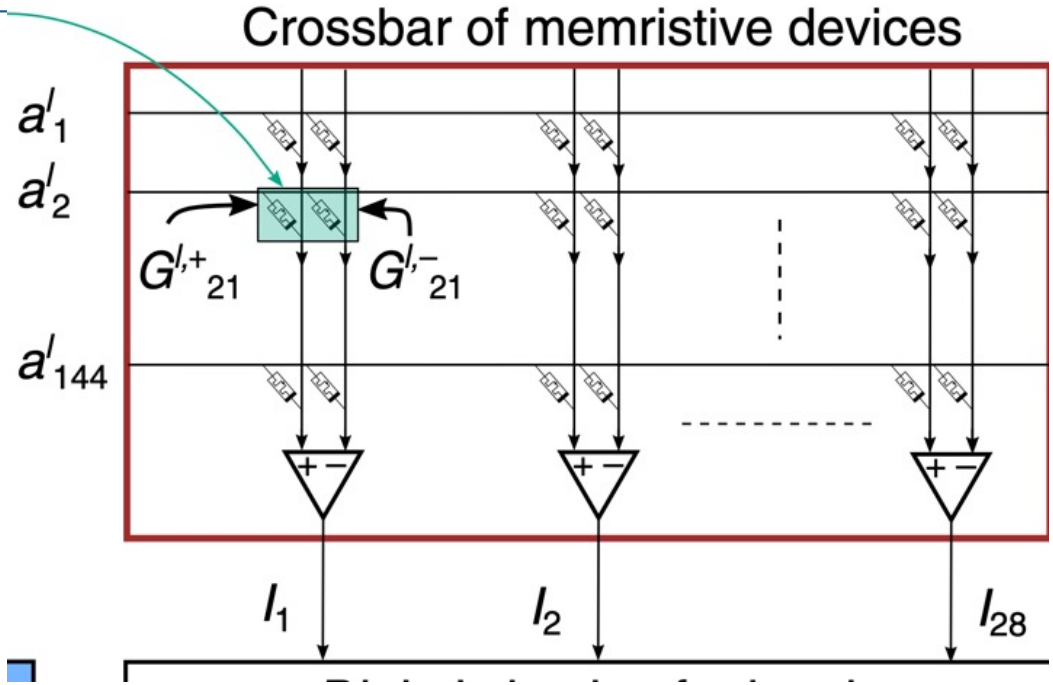
Optimizations achieve **40%-50% speedup** (geomean)

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

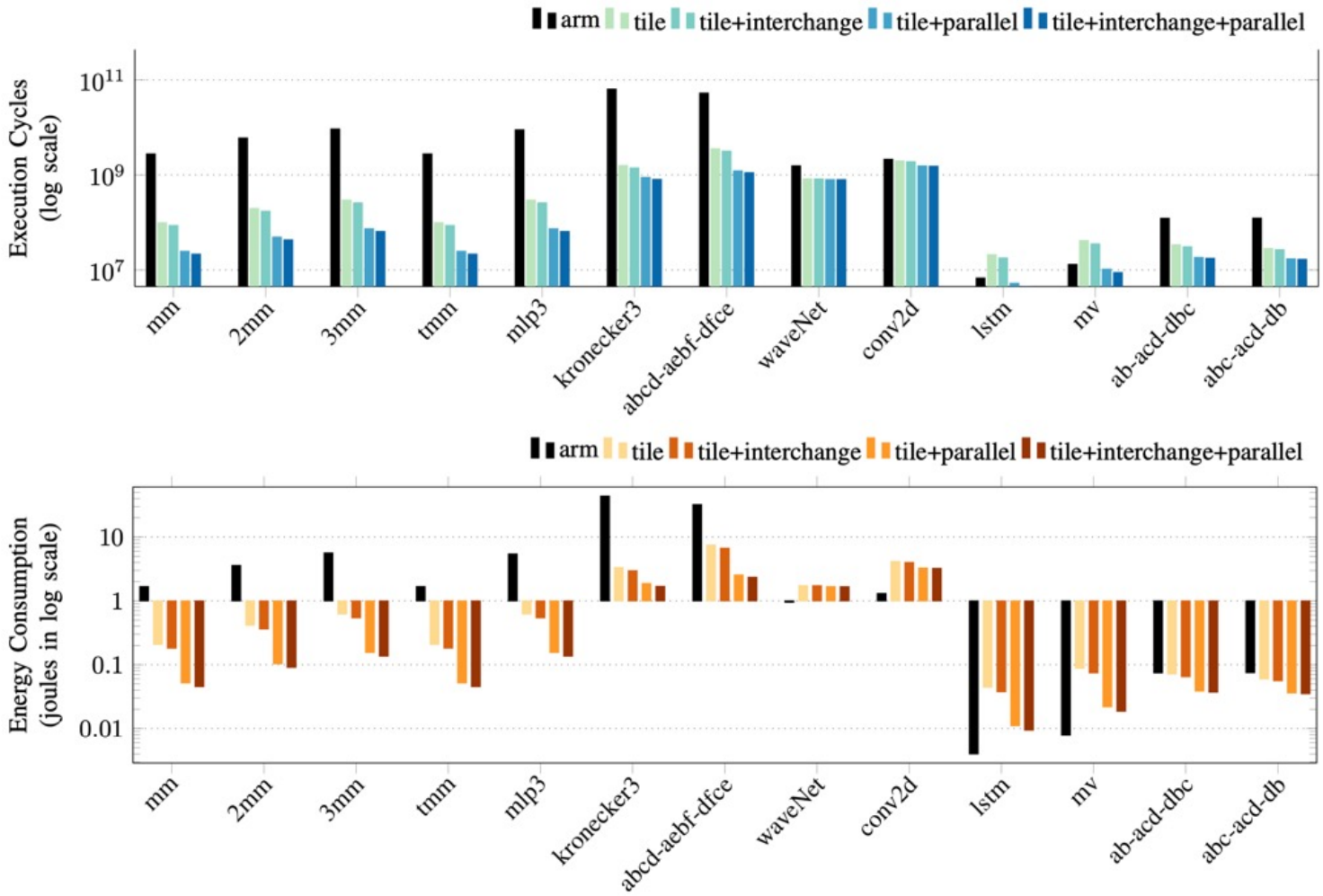
10x-40x less lines of code

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)
}
```



Optimization results: Crossbars beyond matmult

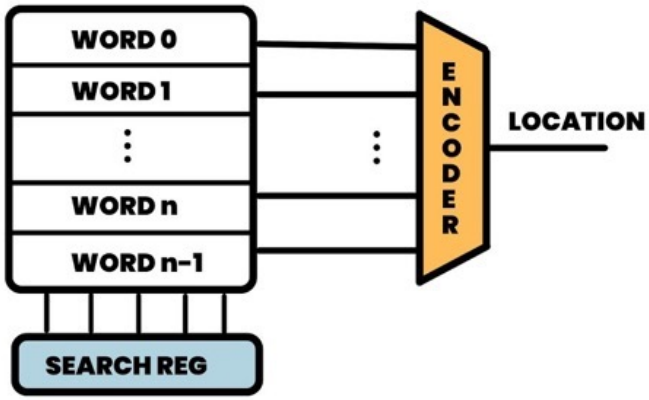


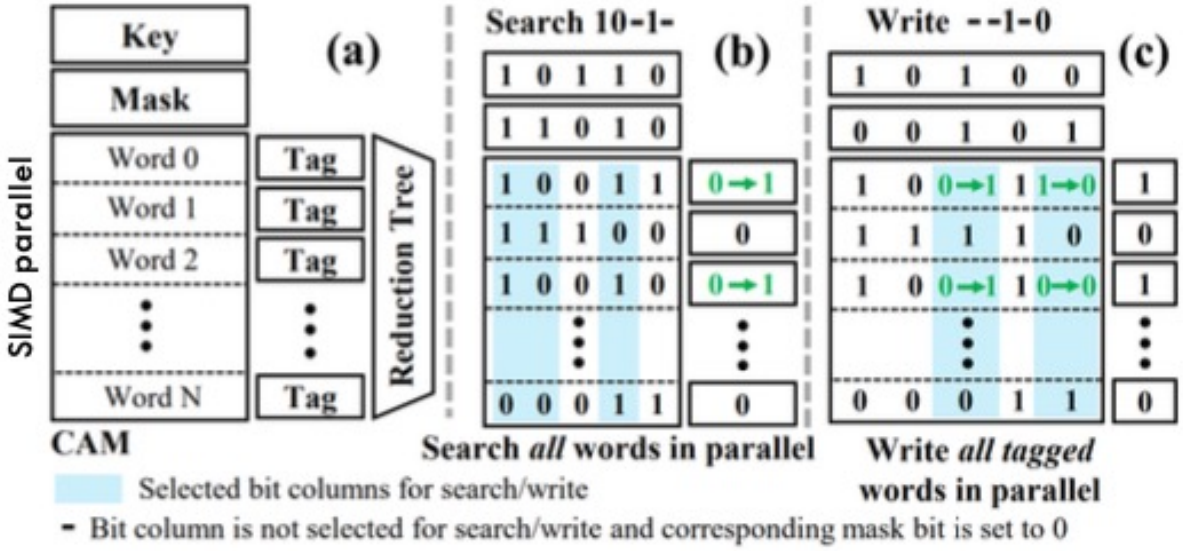
A. Siemieniuk, L
Learning Optim

Machine

Content-addressable memories (CAMs) and Processors

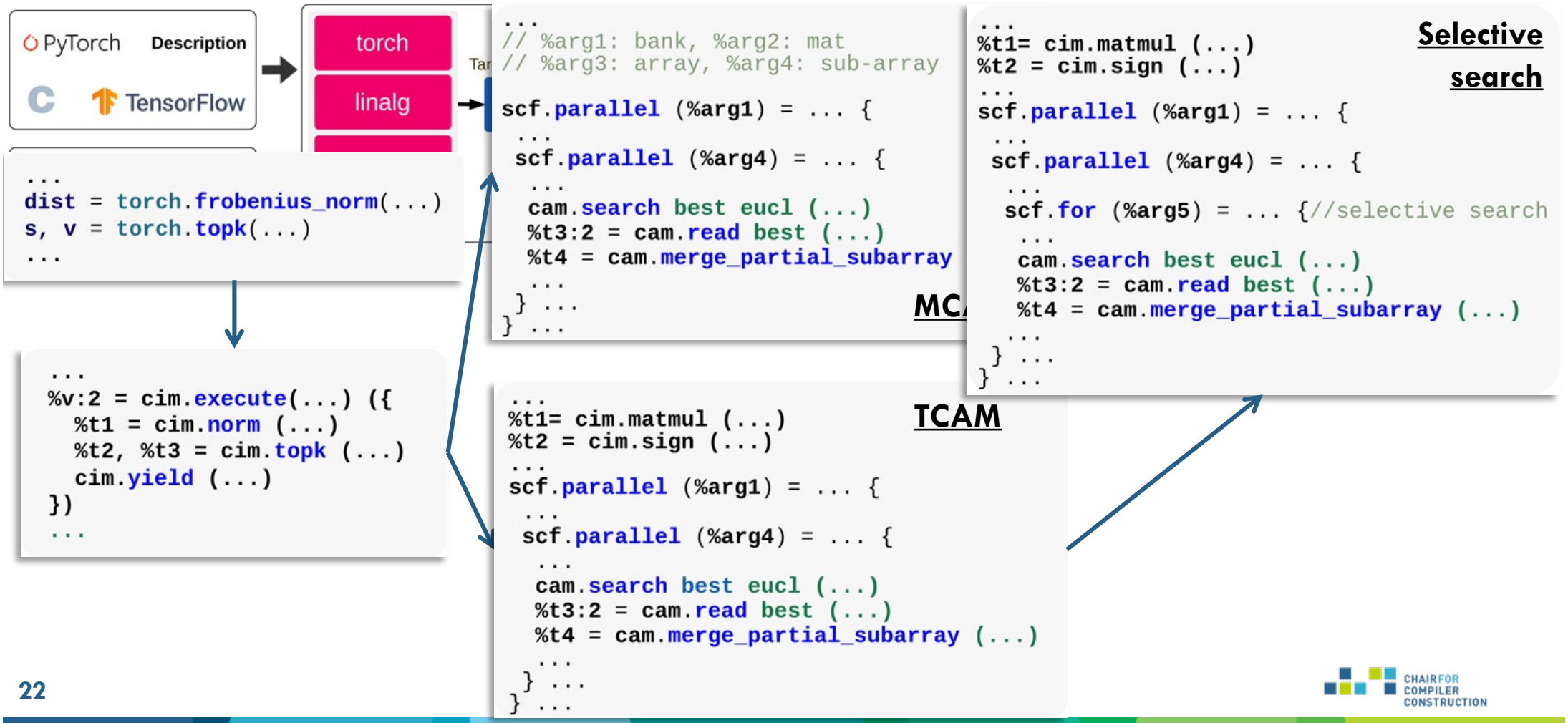
- Emerging Content-addressable memories (CAMs) and associative processors (APs)
- APs: Different programming paradigm based on **search** and **write**





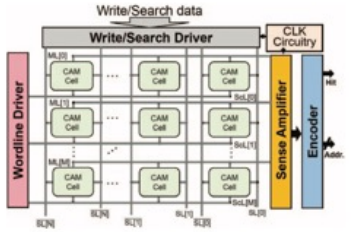
Zha, Yue, and Jing Li. "Hyper-AP: Enhancing associative processing through a full-stack optimization." 2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA). IEEE, 2020.

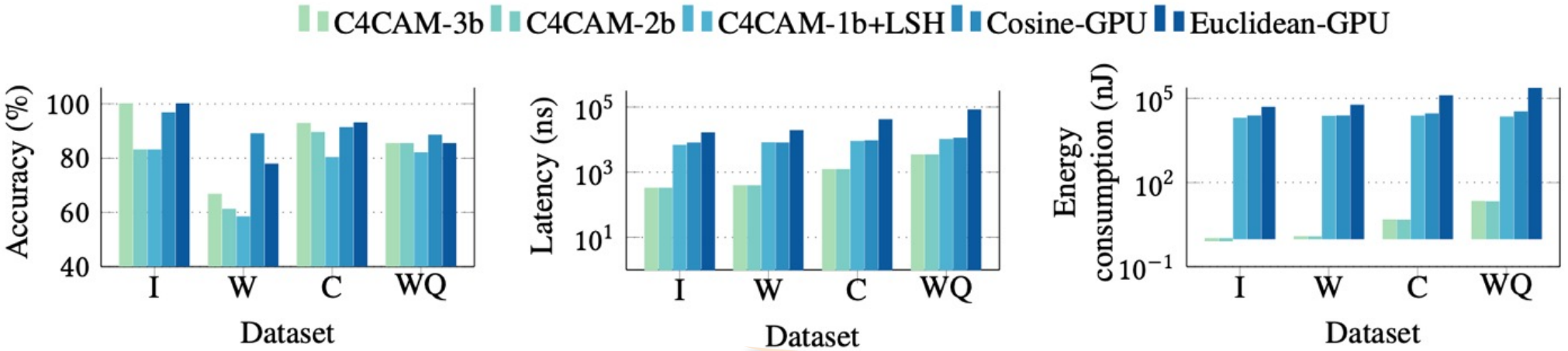
CINM: Lowering for different CAM-based acc.



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**



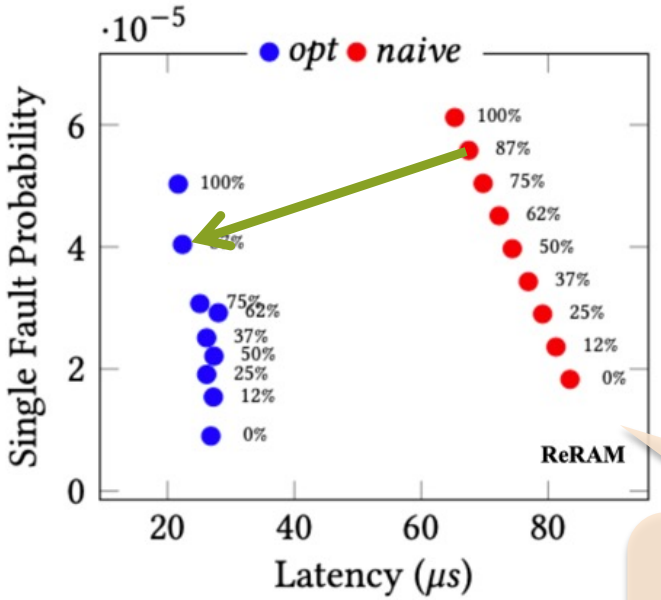


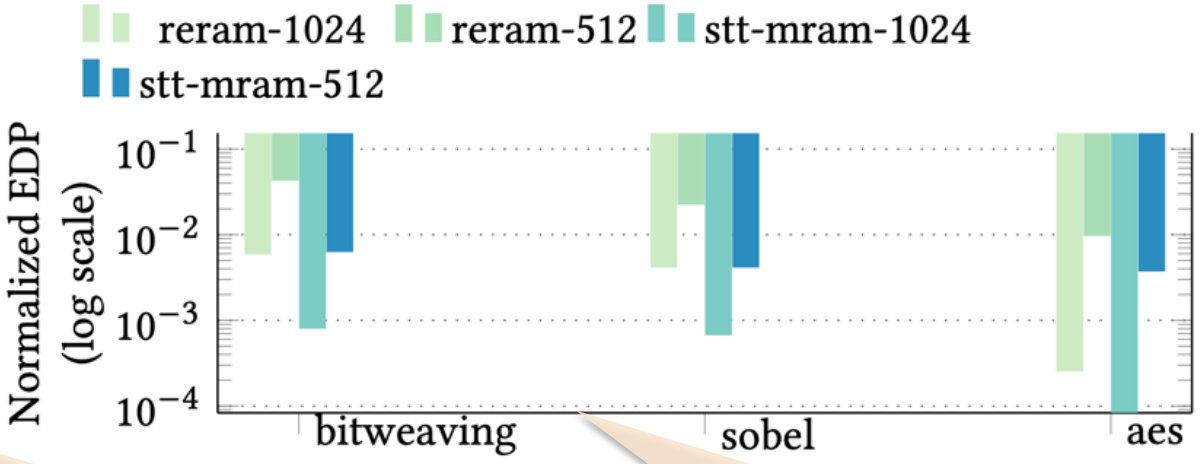
H. Farzaneh, et al. "C4CAM: A Compiler for CAM-based In-memory Accelerators", ASPLOS, 2024

KNN results (128x128 CAM): **14x faster and ~10^4 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





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

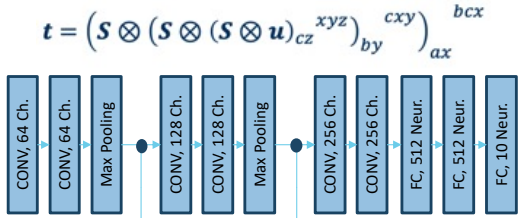
Optimized mapping: **Less latency (3x), better reliability (~1.4x)**

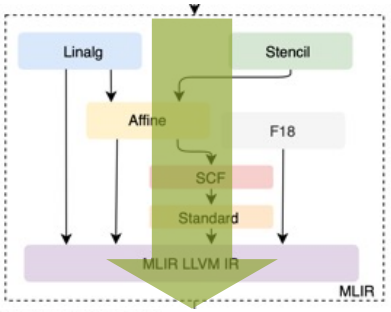
Orders of magnitude better EDP vs CPU baseline

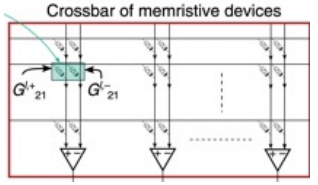
Closing

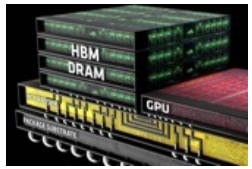
Summary

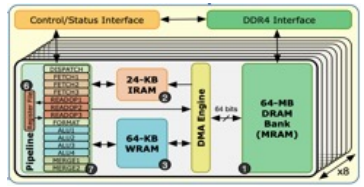
- Automatic optimization for heterogeneous in-memory computing
  - Domain-specific abstractions
  - Abstract primitives, compute model, trade-offs
  - Safe and correct optimization with high-level compilers
- Challenges
  - Still work on (formal) modeling primitives
  - Simulators, prototypes in interdisciplinary research efforts
  - From **algorithmic** abstractions to **geometry** (device parameters)











Thanks! & Acknowledgements

Funded by DFG

DISRUPTIVE MEMORY TECHNOLOGIES
DFG PRIORITY PROGRAM 2377

CO4RTM (450944241)
HetCIM (502388442)

https://everest-h2020.eu

GA: 957269

ScaDS.AI DRESDEN LEIPZIG

BMBF (01IS18026A-D)

AMD

STAATSMINISTERIUM FÜR WISSENSCHAFT KULTUR UND TOURISMUS

Freistaat SACHSEN

..., 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)

References

[**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. Friebel, 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.