CHAIR FOR COMPILER CONSTRUCTION

cfaed CENTER FOR ADVANCING ELECTRONICS DRESDEN

Compilation for heterogeneous compute-in/near-memory systems

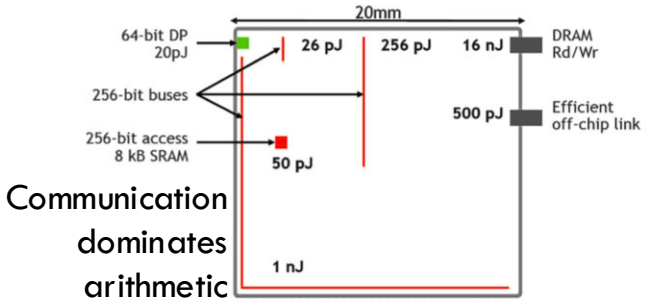
Hamid Farzaneh

TU Dresden, Germany

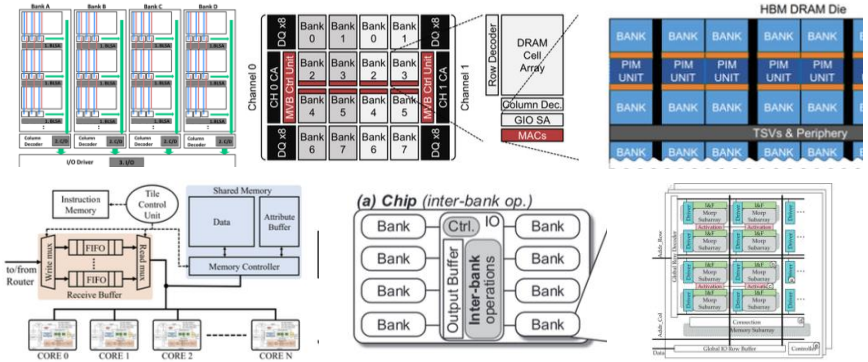
MCCSys Workshop
ASPLOS 2025

Emerging data-centric architectures

- Compute (almost) in-place, avoid data movement, transformations to match primitives
- Novel architectures for near-memory (CNM) and in-memory computing (CIM)



Bartolini, S., "Parallel Programming in Cyber-Physical Systems." Cyber-Physical Systems Security. Springer 2018



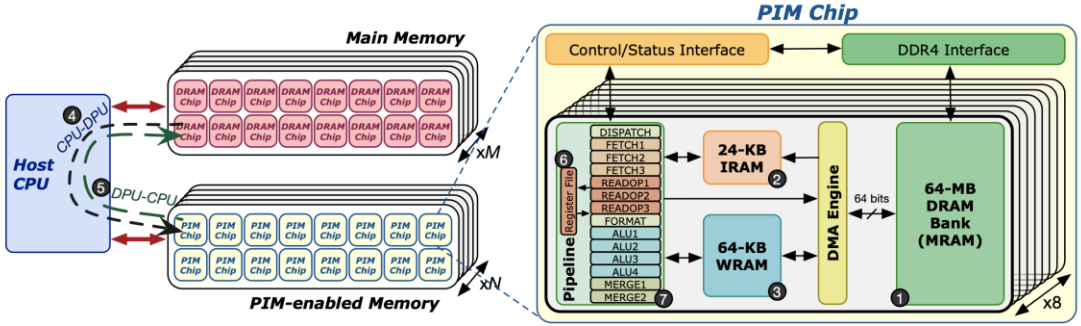
The Landscape of Compute-near-memory and Compute-in-memory: A Research and Commercial Overview

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

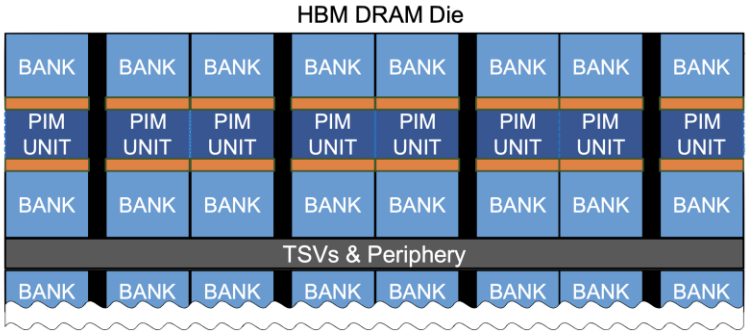
In today's data-centric world, where data fuels numerous application domains, with machine learning at the

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

Categorization: CNM and CIM Systems

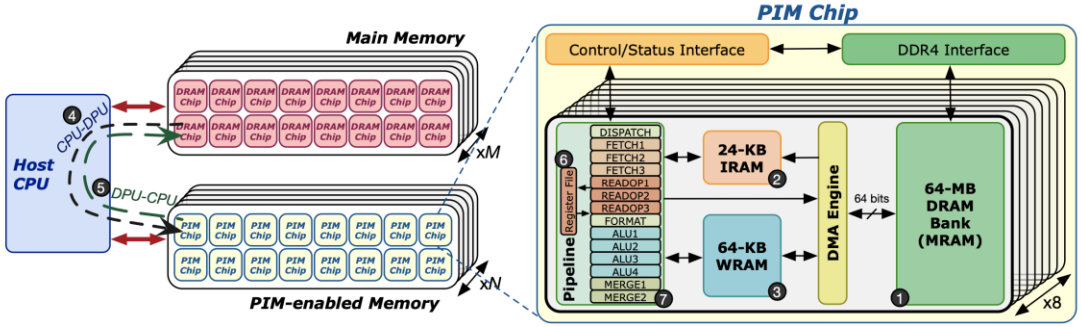


Gómez-Luna, Juan, et al. arXiv:2105.03814 (2021)

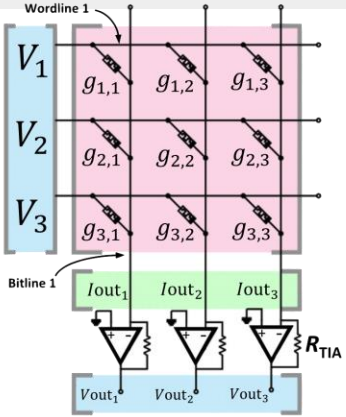


S. Lee et al. ISCA (2021)

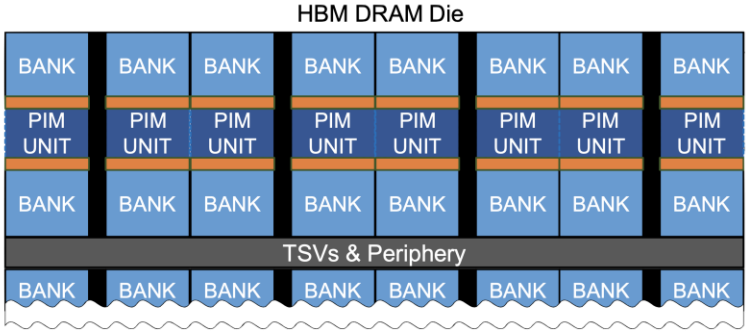
Categorization: CNM and CIM Systems



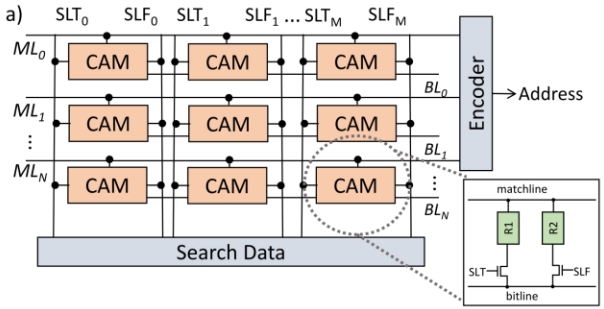
Gómez-Luna, Juan, et al. arXiv:2105.03814 (2021)



Aguirre, F. et al. Nature (2023)

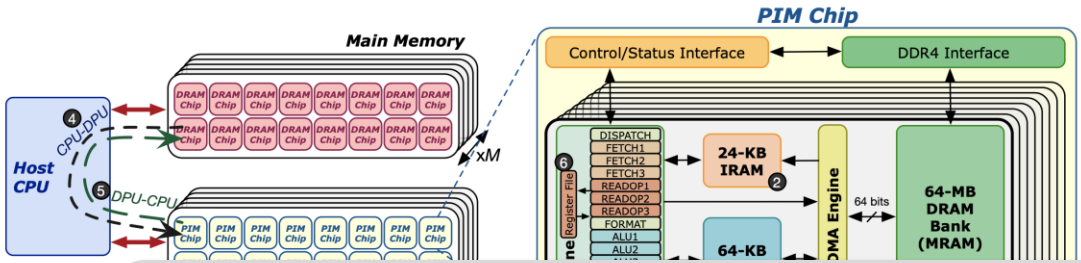


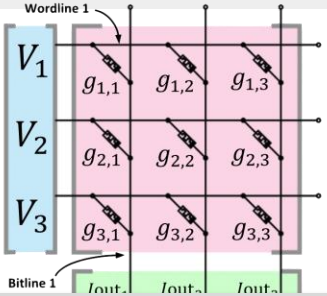
S. Lee et al. ISCA (2021)



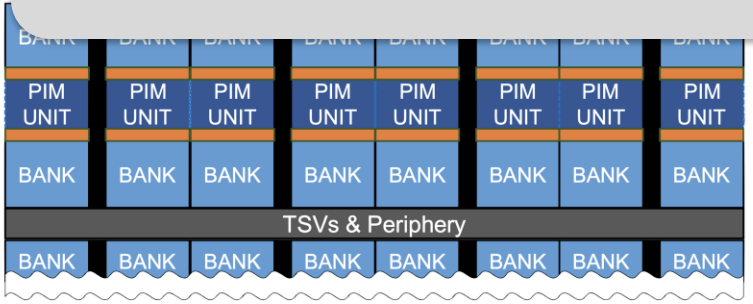
J. P. Cardoso de Lima, et al. FPL (2020)

Categorization: CNM and CIM Systems

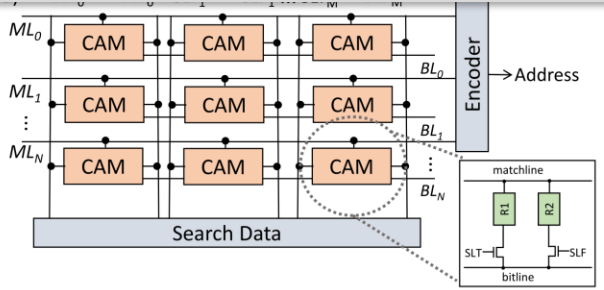




Different systems share a lot of commonalities.
There is no established programming methodology.



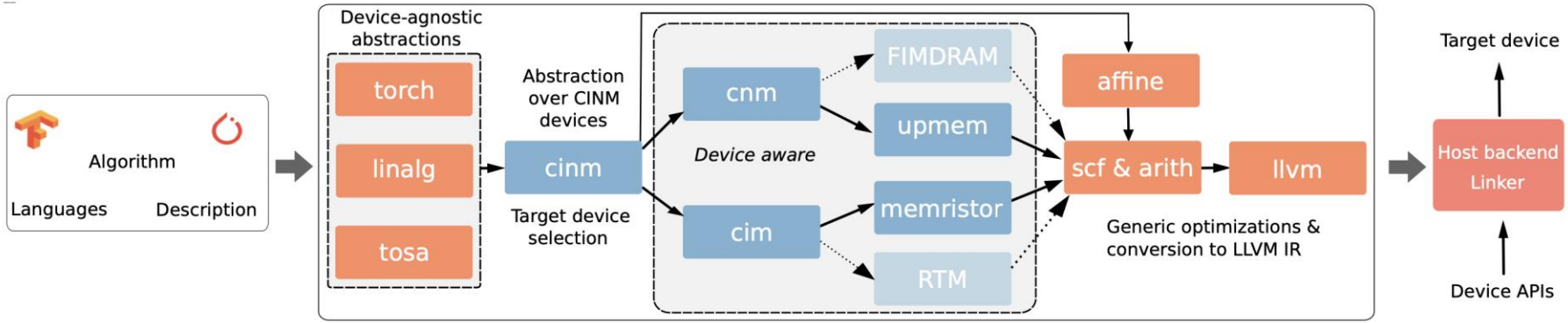
S. Lee et al. ISCA (2021)



J. P. Cardoso de Lima, et al. FPL (2020)

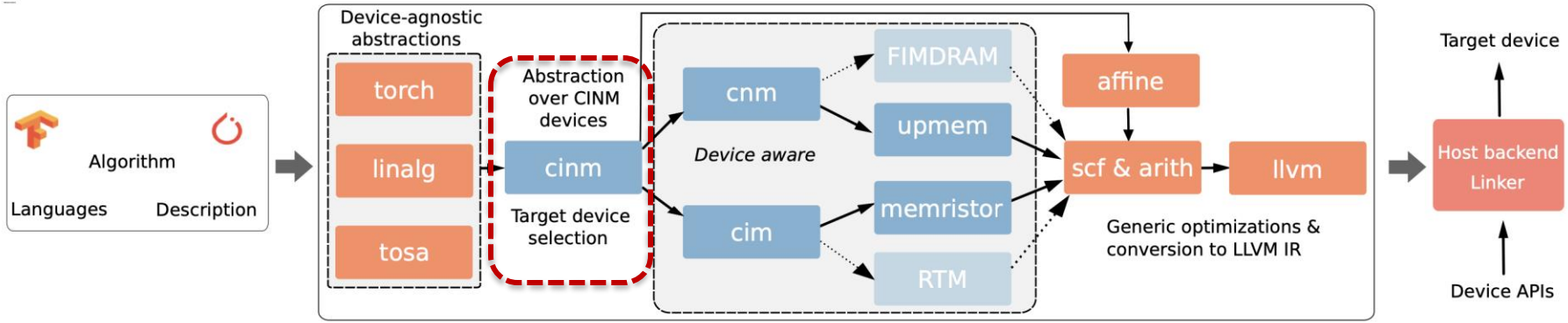
Cinnamon: A Compilation infrastructure for CINM

- Abstract set of operations for CINM systems
- Reusable blocks for different purposes
- Hierarchical flow: domain specific and target specific transformation
- Input: Common abstractions, such as linear algebra and beyond



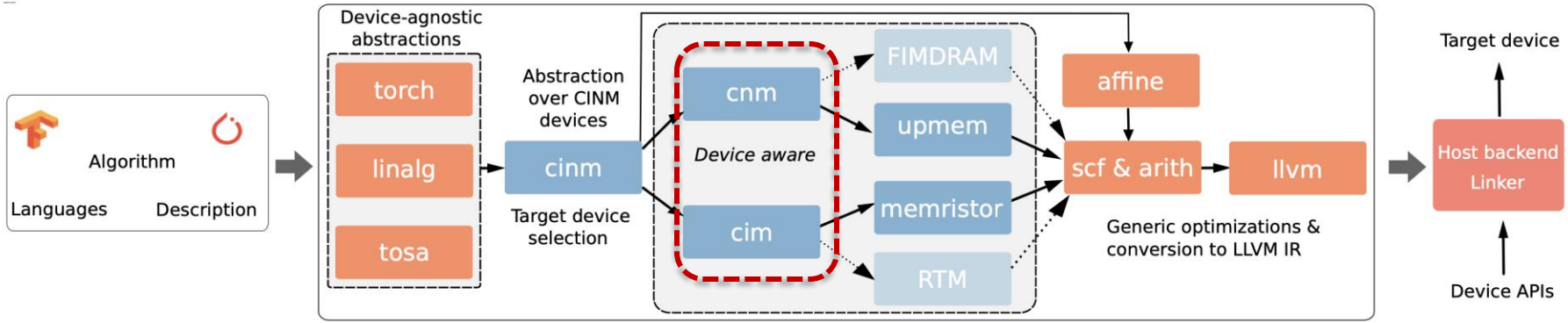
Cinnamon: A Compilation infrastructure for CINM

- Abstract set of operations for CINM systems
- Reusable blocks for different purposes
- Hierarchical flow: domain specific and target specific transformation
- Input: Common abstractions, such as linear algebra and beyond



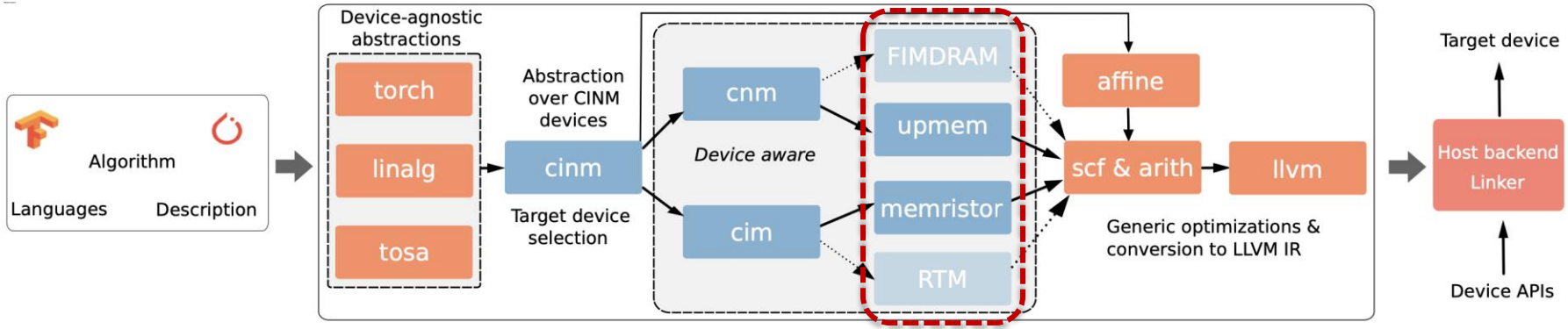
Cinnamon: A Compilation infrastructure for CINM

- Abstract set of operations for CINM systems
- Reusable blocks for different purposes
- Hierarchical flow: domain specific and target specific transformation
- Input: Common abstractions, such as linear algebra and beyond



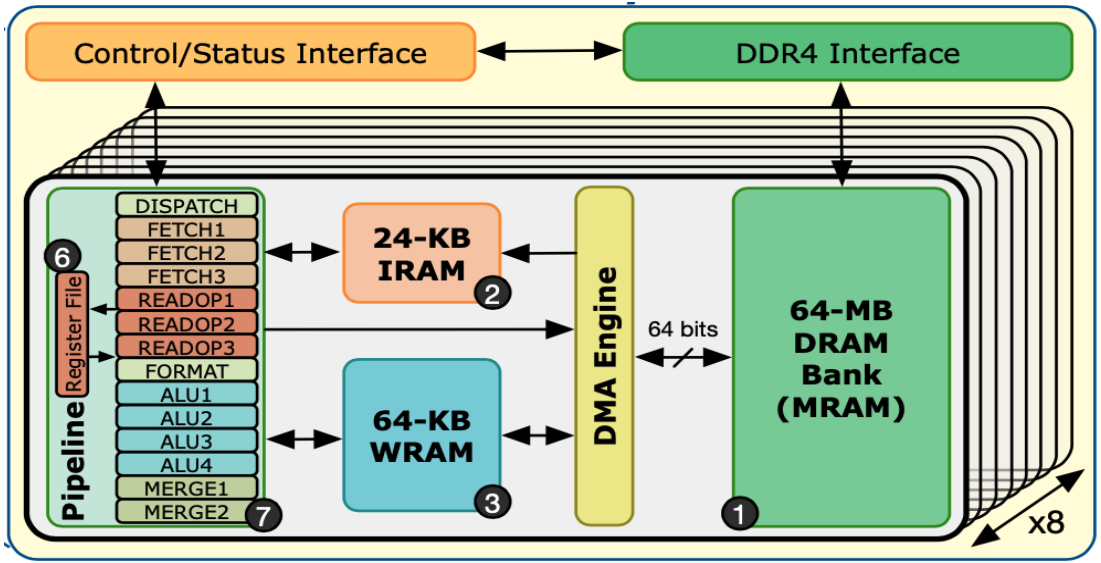
Cinnamon: A Compilation infrastructure for CINM

- Abstract set of operations for CINM systems
- Reusable blocks for different purposes
- Hierarchical flow: domain specific and target specific transformation
- Input: Common abstractions, such as linear algebra and beyond



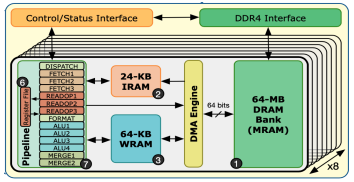
UPMEM example: Matmult

```
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
}
```



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
}
```

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>
```

UPMEM example: Matmult

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


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 = N : 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
   }
}
```

CIM-Crossbar example: Matmult

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

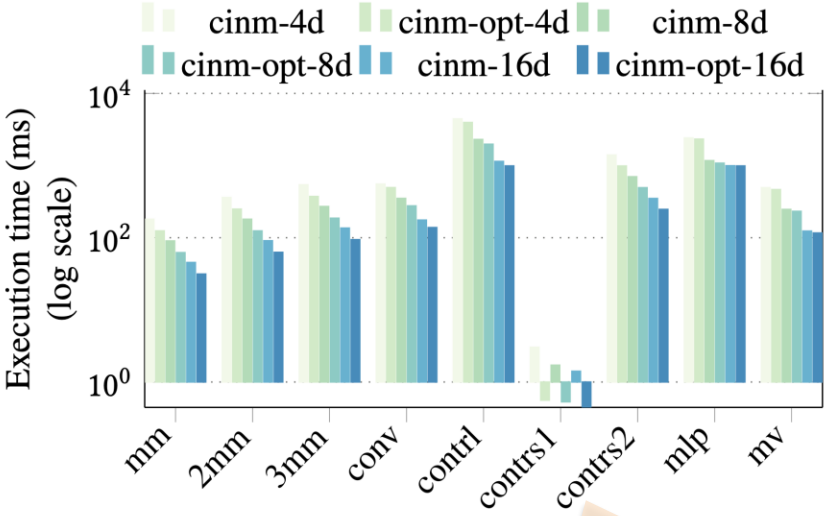


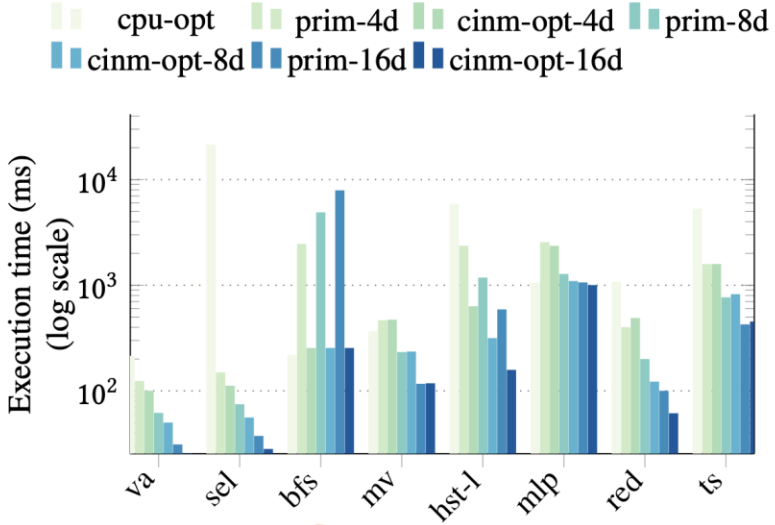
```
%dev = cim.acquire_device -> !cim.deviceId
%xbar = cim.acquire_crossbar %dev : ... -> !cim.crossbarId
%fut0 = cim.op.gemm %xbar, %barg0, %btw0 : ... -> !cim.future<…>
%bm0 = cim.barrier %fut0 : !cim.future<…> -> tensor<…>
cim.release_crossbar %xbar : !cim.crossbarId
cim.release_device %dev : !cim.deviceId
```



```
…
memristor.write_to_crossbar %c0_i32, %rhsb : i32, memref<…>
memristor.gemm %xbar, %lhsb, %resb : i32, memref<…>, memref<…>
memristor.barrier %xbar : i32
…
```

UPMEM CNM system: results





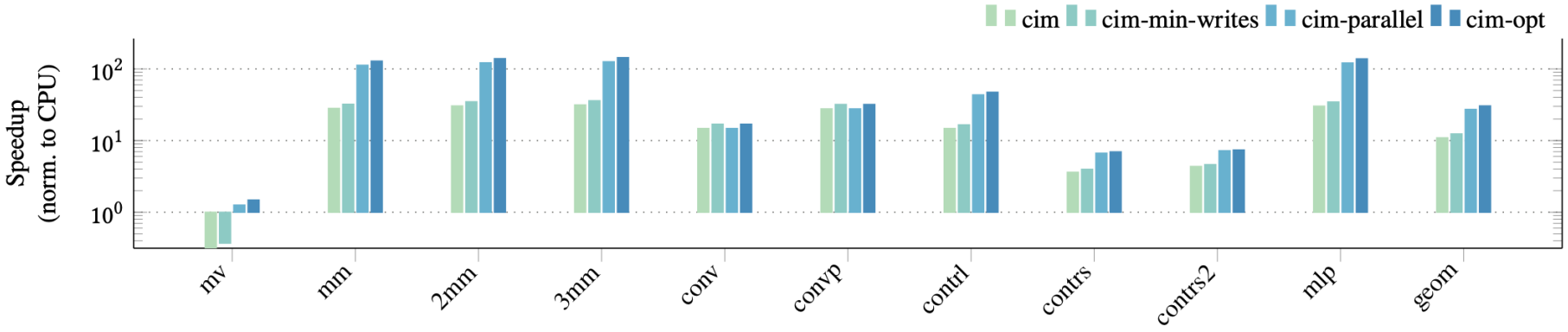
1-DIMM 128 DPUs
4-DIMMs 512 DPUs
8-DIMMs 1024 DPUs
16-DIMMs 2048 DPUs

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

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

CIM-Crossbar: results

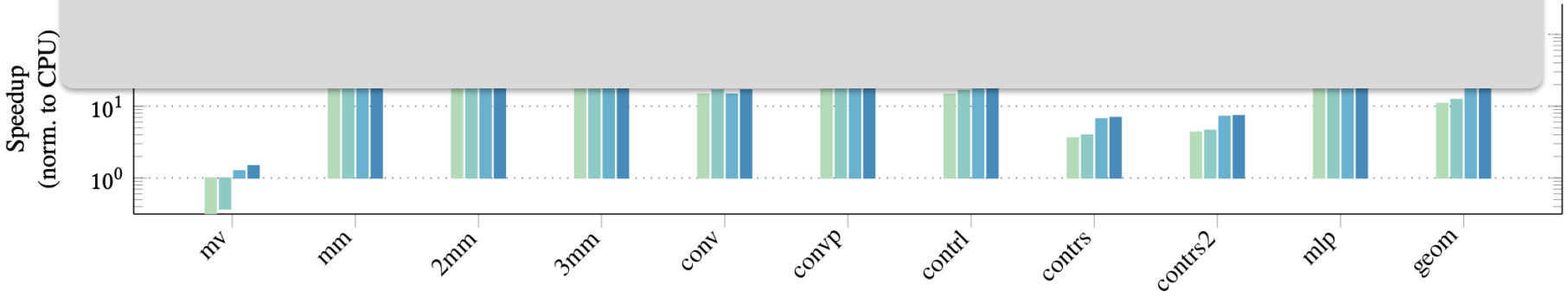
- Crossbar target: validate results of SOA
- Reuse the domain-specific transformations
- Enable device-specific optimizations



CIM-Crossbar: results

- Crossbar target: validate results of SOA
- Reuse the domain-specific transformations
- [illegible]

Will be presented on Thursday



Domain-target implementation challenges

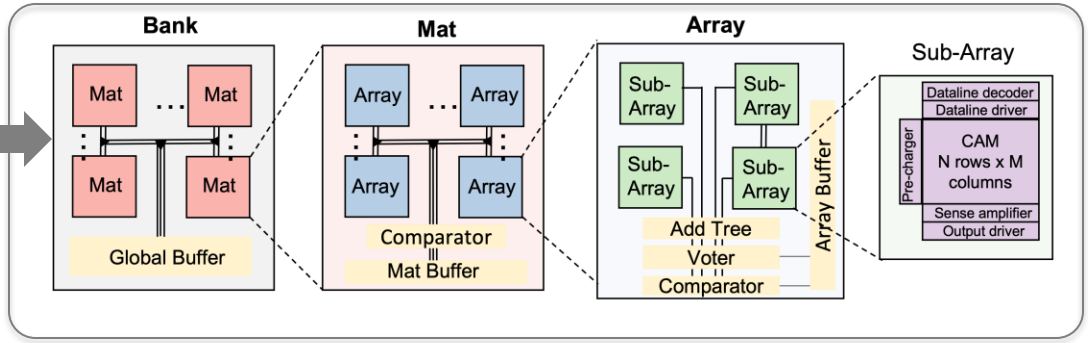
What the user writes

```
def forward(self, input: Tensor, dot: bool = False)
    ↪ -> Tensor:
    others = self.weight.transpose(-2, -1)
    matmul = torch.matmul(input, (others))
    values, indices = torch.ops.aten.topk(matmul, 1,
        ↪ largest=False)
    return indices
```

What the device expects

```
…
cam_config = load_config()
cam = CAMASim(cam_config)
cam.write(CAM_Data)
CAM_pred_ids, _, _ = cam.query(CAM_Query)
return CAM_pred_ids
…
```

Only manual translation and mapping

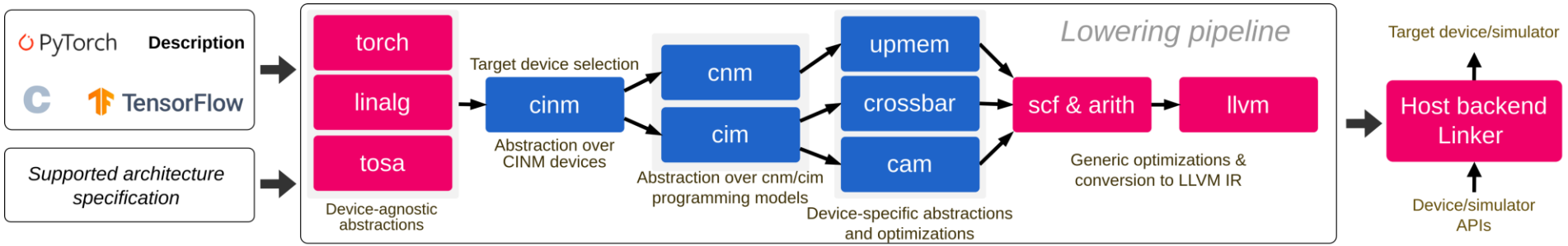


Challenges with manual designs

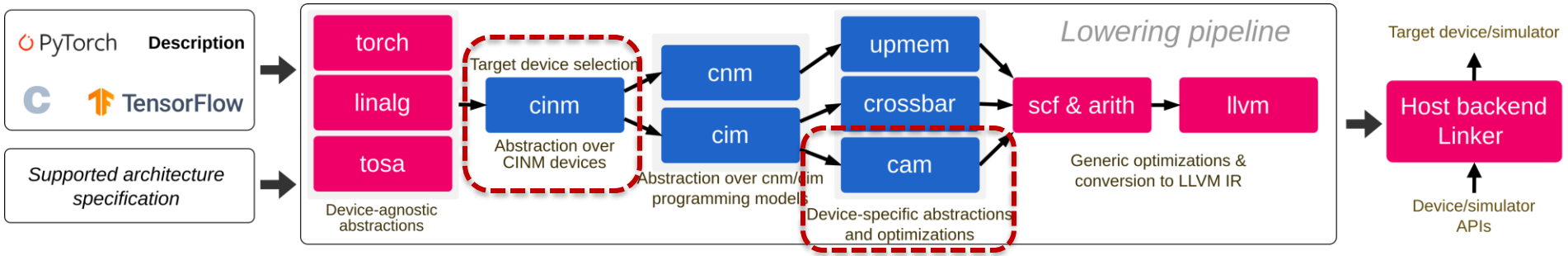
- Key variations in current CAM-based designs
  - CAM types (TCAM, MCAM, ACAM) with perf/accuracy trade-offs
  - Distance metrics (e.g., Hamming, Euclidean, cosine similarity)
- Language variations
  - Varied merging strategies for handling partial results

- Presently, all of this is handled manually with low-level APIs, restricting CAMs to device experts

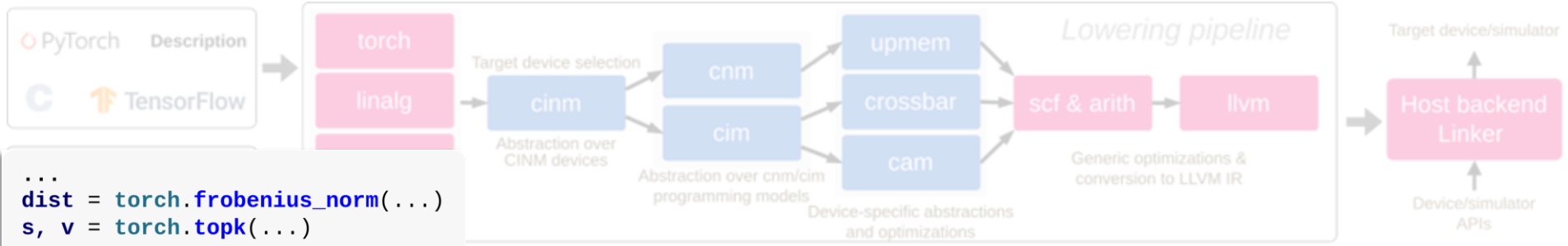
C4CAM: Lowering for different CAM-based acc.



C4CAM: Lowering for different CAM-based acc.

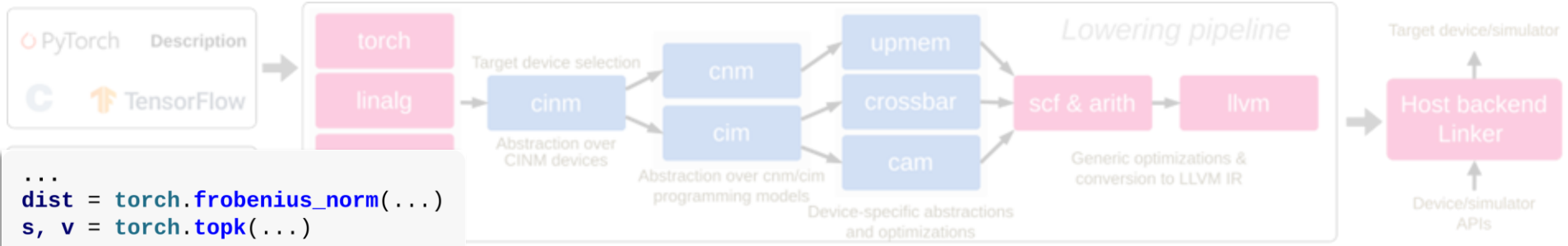


C4CAM: Lowering for different CAM-based acc.




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

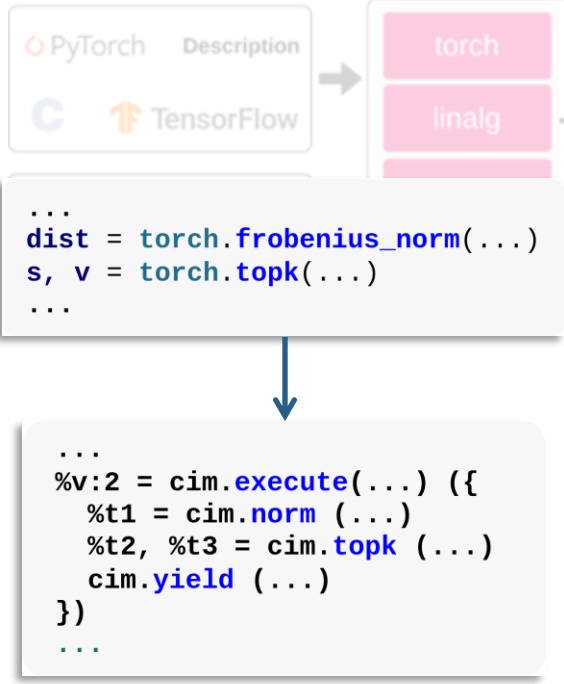
C4CAM: Lowering for different CAM-based acc.




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

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

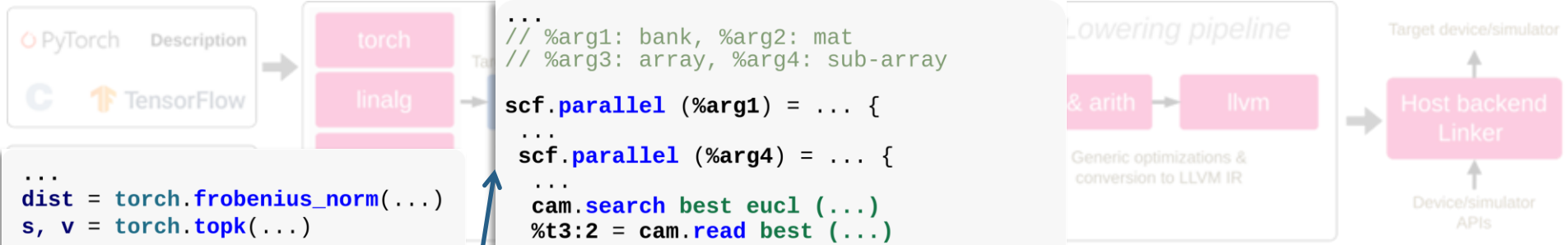
C4CAM: Lowering for different CAM-based acc.



Algorithm 1: Similarity search detection.

```
1  /* Pattern matching for dot product similarity */
2  Replace op<topk> (op<matmul> (arg2, op<transpose> (arg1)), arg3)
3    with op <similarity> (dot, arg1, arg2, arg3);
4  /* Pattern matching for Euclidean distance */
5  Replace op <topk>(op<norm> (op<sub> (arg1, arg2)), arg3)
6    with op <similarity> (euc, arg1, arg2, arg3);
7  /* Pattern matching for cosine similarity */
8  Replace op <smulmat>(op <div> (cons1, op<mul>(op<norm>(arg1),
     op<norm> (arg2))), op<matmul>(arg1, op<transpose>(arg2)))
9    with op <similarity> (cos, arg1, arg2);
10 /* Pattern matching for Hamming distance */
11 Replace op <nonzero>(op <cmp>(lt, op <popcount>(op <xor>(arg1,
     arg2), arg3)
12   with op <similarity> (ham, arg1, arg2, arg3);
```

C4CAM: 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 (...)
})
...
```

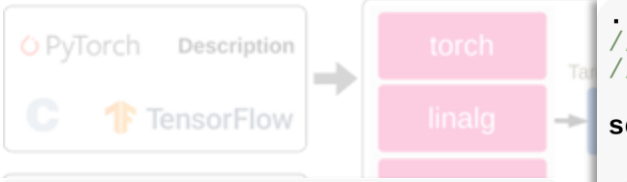
```
...
// %arg1: bank, %arg2: mat
// %arg3: array, %arg4: sub-array
scf.parallel (%arg1) = ... {
 ...
 scf.parallel (%arg4) = ... {
  ...
  cam.search best eucl (...)
  %t3:2 = cam.read best (...)
  %t4 = cam.merge_partial_subarray (...)
  ...
 } ...
} ...
```

MCAM

```
...
%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

C4CAM: 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
  ...
 } ...
} ...
```

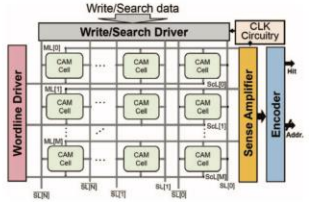
TCAM

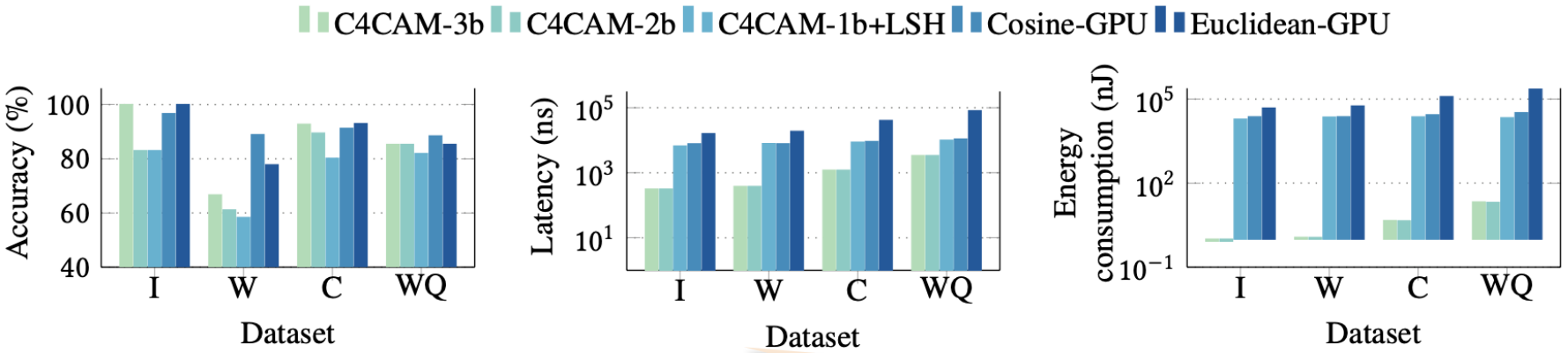
Selective search

```
...
%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 (...)
  ...
 } ...
} ...
```

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**



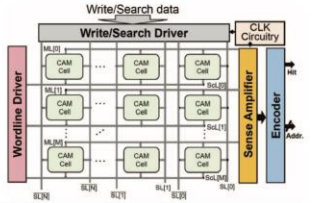


H. Farzaneh, et al. "C4CAM: A Compiler for CAM-based In-memory Accelerators", ASPLOS, 2024

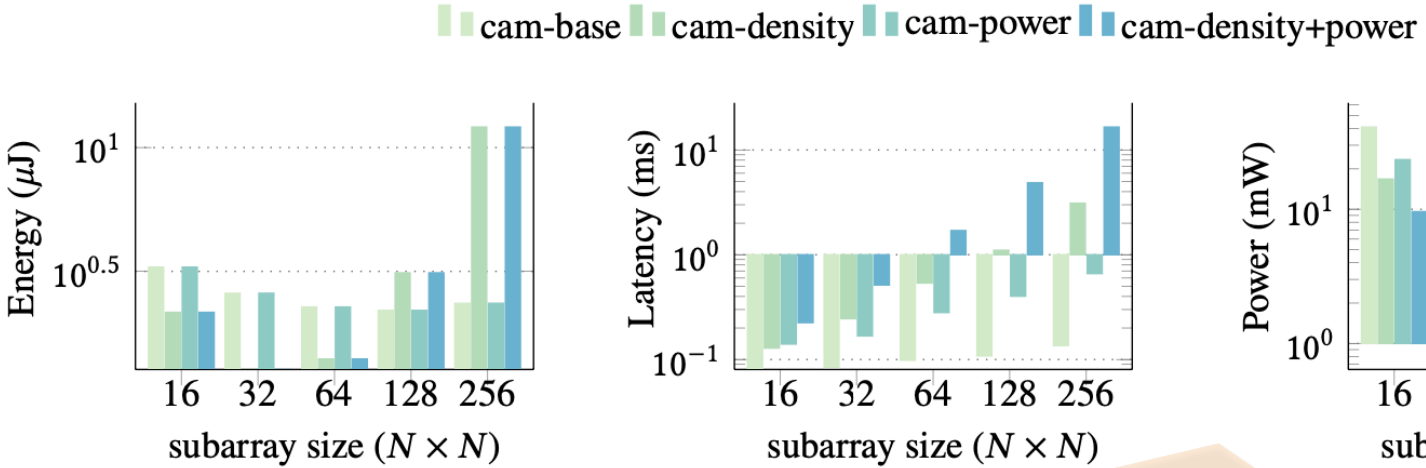
KNN results (128x128 CAM): **14x faster and ~10^4 less energy** compared to GPU

CAMs: Design space exploration

- Retargetable compiler for CAM exploration: Sizes and features
- Compiler flags for optimization target



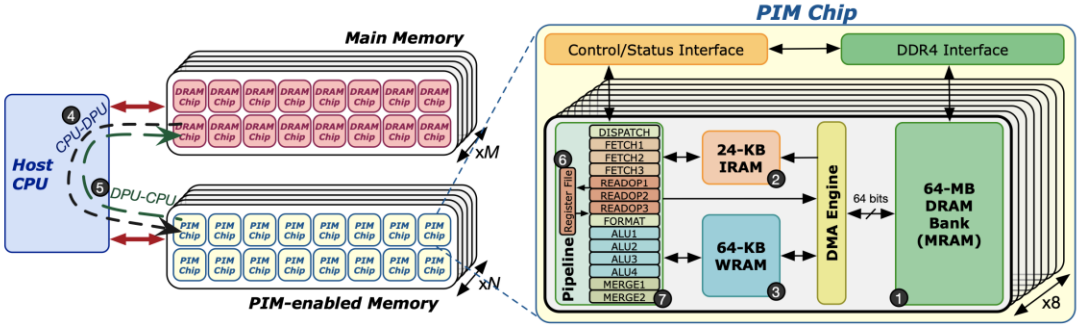
Asplos 2024



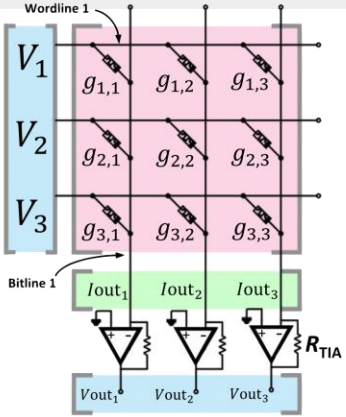
H. Farzaneh, et al. "C4CAM: A Compiler for CAM-based In-memory Accelerators", ASPLOS, 2024

Different flags expose trade-offs w/o manual re-coding. Next: include **device-level parameters**

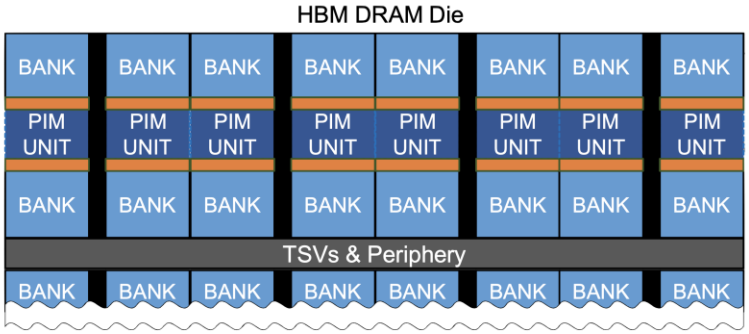
Cinnamon – C4CAM



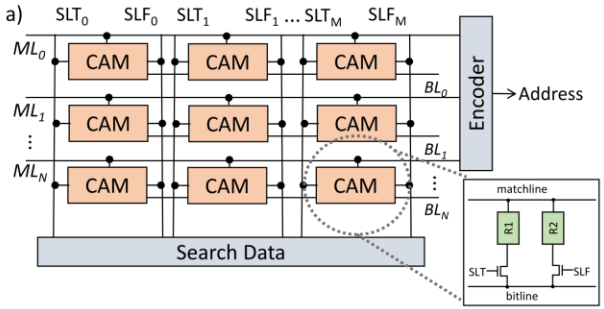
Gómez-Luna, Juan, et al. arXiv:2105.03814 (2021)



Aguirre, F. et al. Nature (2023)



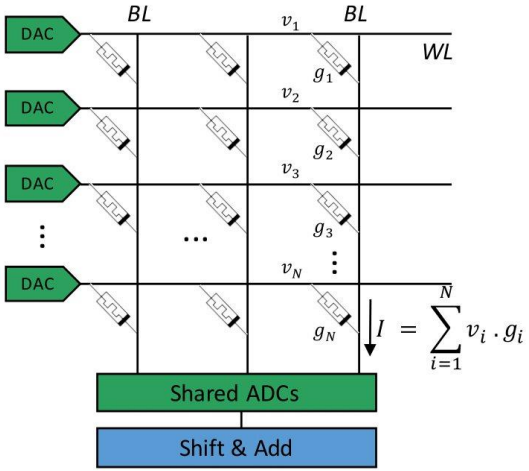
S. Lee et al. ISCA (2021)

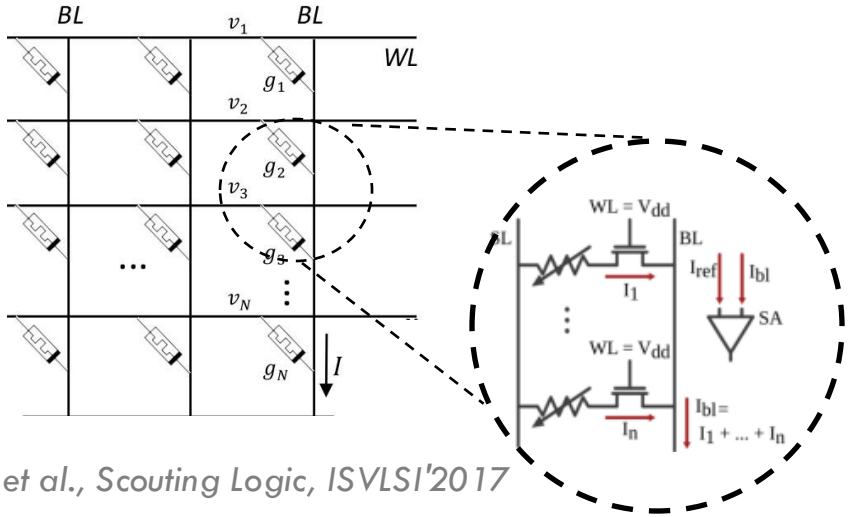


J. P. Cardoso de Lima, et al. FPL (2020)

Memristor-based CIM

- Memristor-crossbars enable:
  - Analog dot-product using DA/AD converters
  - Bitwise operations (OR, AND, and NOT) using custom SA





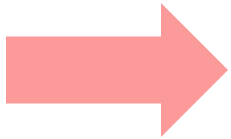
L. Xie et al., Scouting Logic, ISVLSI'2017

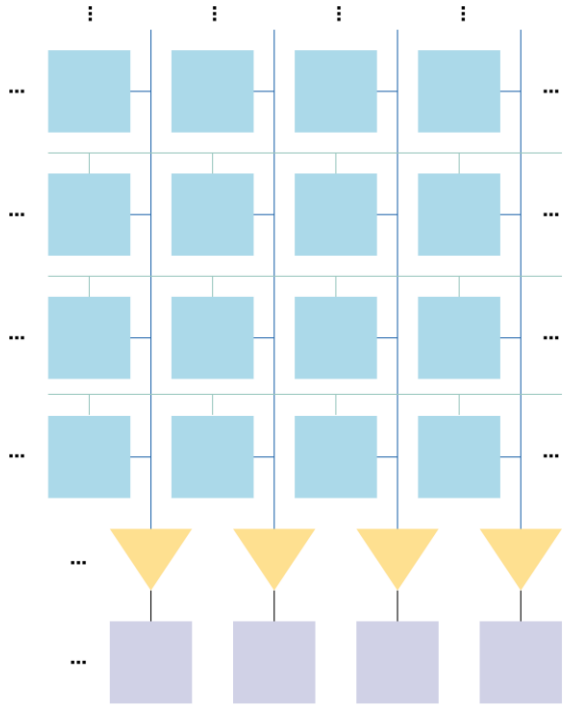
Sherlock: Logic-in-memory in NVMs

```
for (i = 0 to K):
  t1 = NOT(C1[i]);
  T2 = AND(T1, data[i]);
  T3 = AND(T2, meq1);
  m_gt = OR(T3, mgt);
  T4 = NOT(data[i]);
  T5 = AND(T4, C2[i]);
  T6 = AND(T5, m_eq2);
  m_lt= OR(T6, mlt);
  T7 = XOR(data[i], C1[i]);
  T8 = NOT(T7);
  meq1 = AND(meq1, T8);
  T9 = XOR(data[i], C2[i]);
  T10 = NOT(T9);
  meq2 = AND(meq2, T10);
```

Sherlock: Logic-in-memory in NVMs

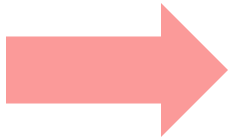
```
for (i = 0 to K):
  t1 = NOT(C1[i]);
  T2 = AND(T1, data[i]);
  T3 = AND(T2, meq1);
  m_gt = OR(T3, mgt);
  T4 = NOT(data[i]);
  T5 = AND(T4, C2[i]);
  T6 = AND(T5, m_eq2);
  m_lt= OR(T6, mlt);
  T7 = XOR(data[i], C1[i]);
  T8 = NOT(T7);
  meq1 = AND(meq1, T8);
  T9 = XOR(data[i], C2[i]);
  T10 = NOT(T9);
  meq2 = AND(meq2, T10);
```

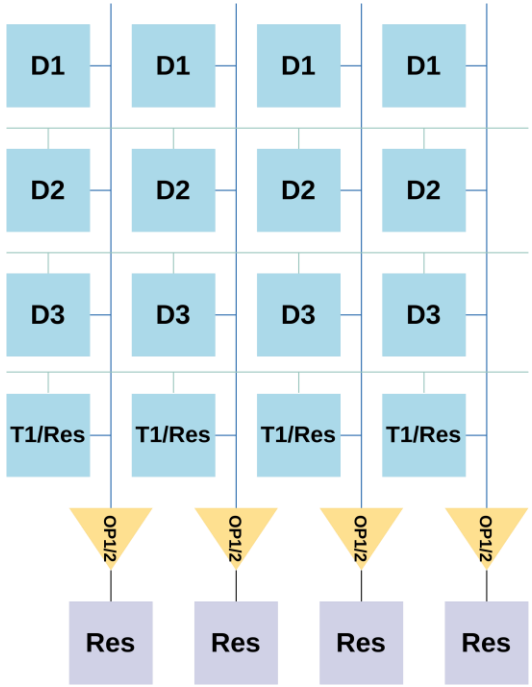




Sherlock: Logic-in-memory in NVMs

```
for (i = 0 to K):
  t1 = NOT(C1[i]);
  T2 = AND(T1, data[i]);
  T3 = AND(T2, meq1);
  m_gt = OR(T3, mgt);
  T4 = NOT(data[i]);
  T5 = AND(T4, C2[i]);
  T6 = AND(T5, m_eq2);
  m_lt= OR(T6, mlt);
  T7 = XOR(data[i], C1[i]);
  T8 = NOT(T7);
  meq1 = AND(meq1, T8);
  T9 = XOR(data[i], C2[i]);
  T10 = NOT(T9);
  meq2 = AND(meq2, T10);
```

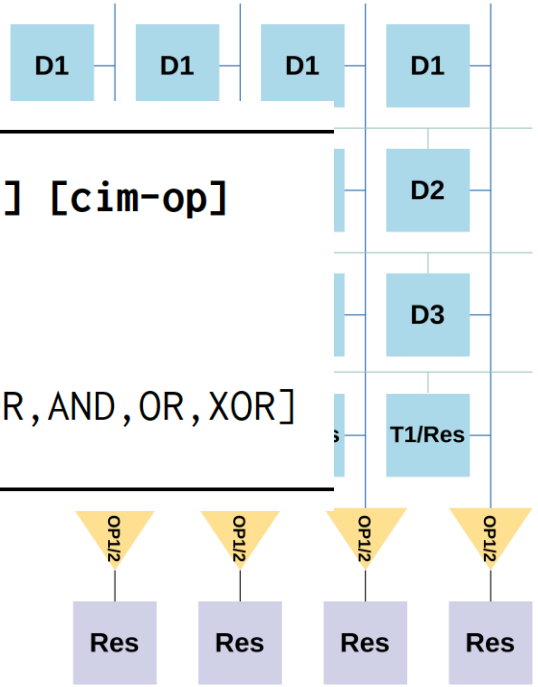




Sherlock: Logic-in-memory in NVMs

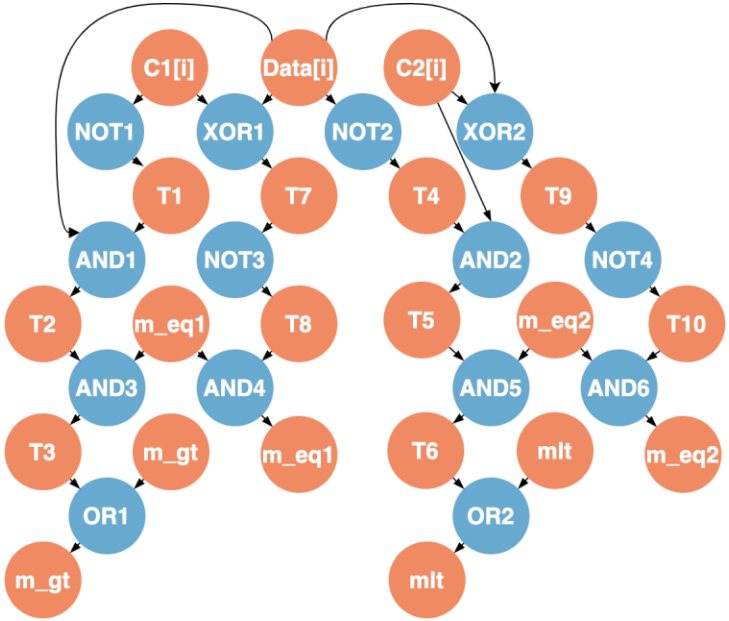
```
for (i = 0 to K):
  t1 = NOT(C1[i]);
  T2 = AND(T
  T3 = AND(T
  m_gt = OR(
  T4 = NOT(d
  T5 = AND(T
  T6 = AND(T
  m_lt= OR(T
  T7 = XOR(d
  T8 = NOT(T
  meq1 = AND(meq1, T8);
  T9 = XOR(data[i], C2[i]);
  T10 = NOT(T9);
  meq2 = AND(meq2, T10);
```

```
...
op address[arrayID][columns][rows] [cim-op]
write [0][4,8,12,16][932]
Read  [0][1,5,9, 13][5]
Shift [0] R[3]
Read  [0][4,8,12,16][933,934] [XOR,AND,OR,XOR]
...
```



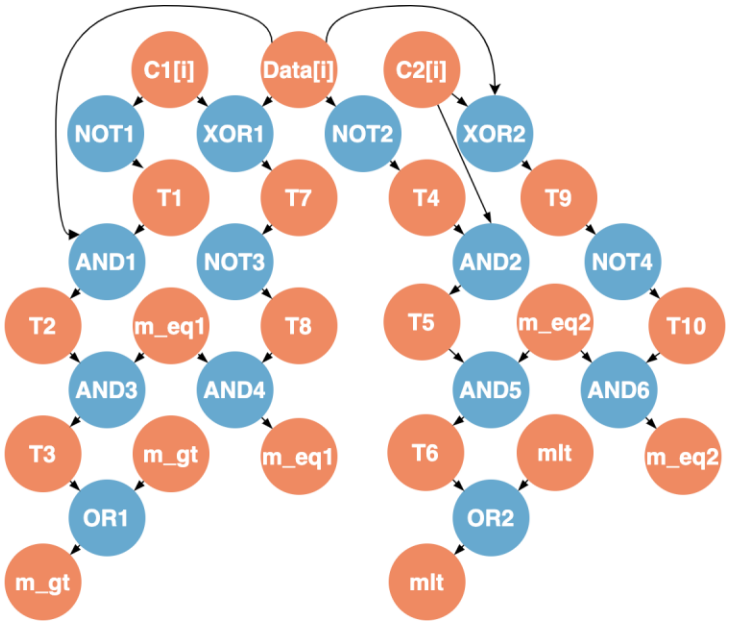
Sherlock mapping

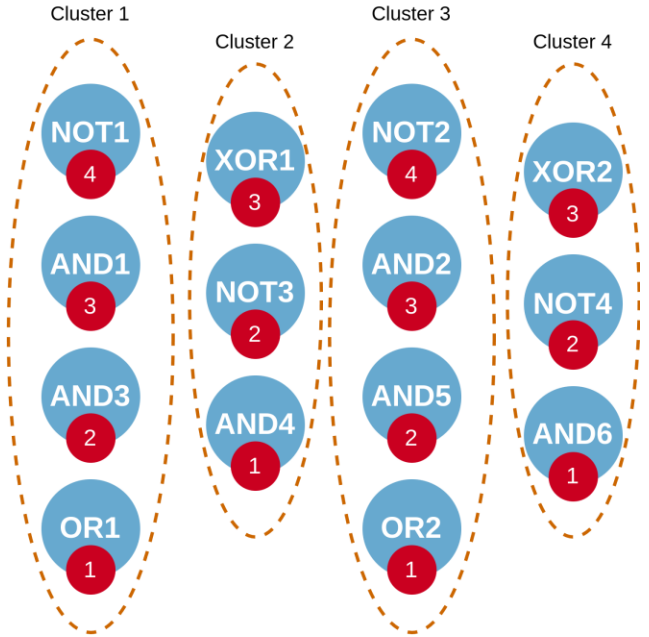
- Goal: find clusters of operations that match the device model



Sherlock mapping

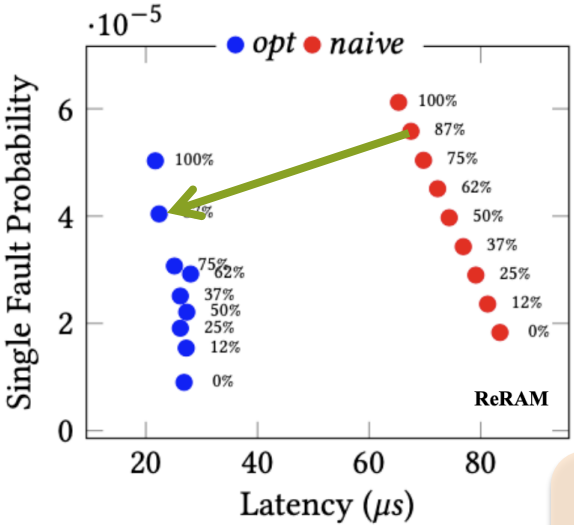
- Goal: find clusters of operations that match the device model



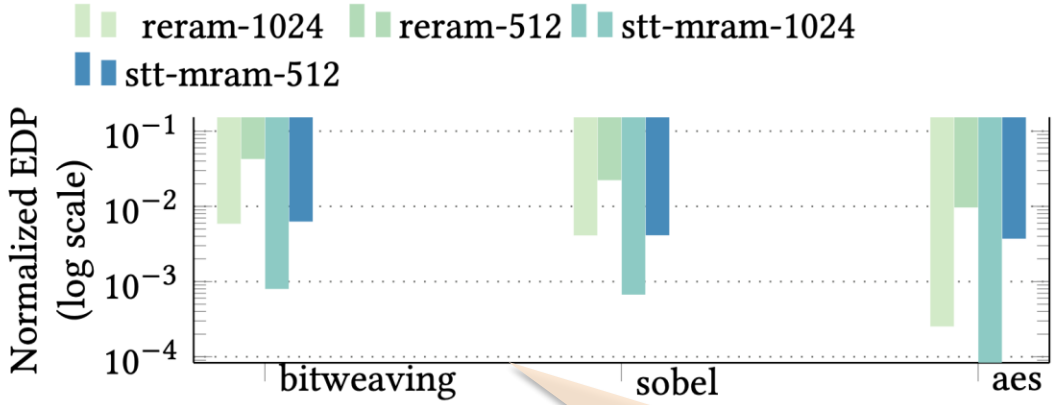


Sherlock: Logic-in-memory in NVMs

- 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 (10x), better reliability (~4.6x)**

Orders of magnitude better EDP vs CPU baseline

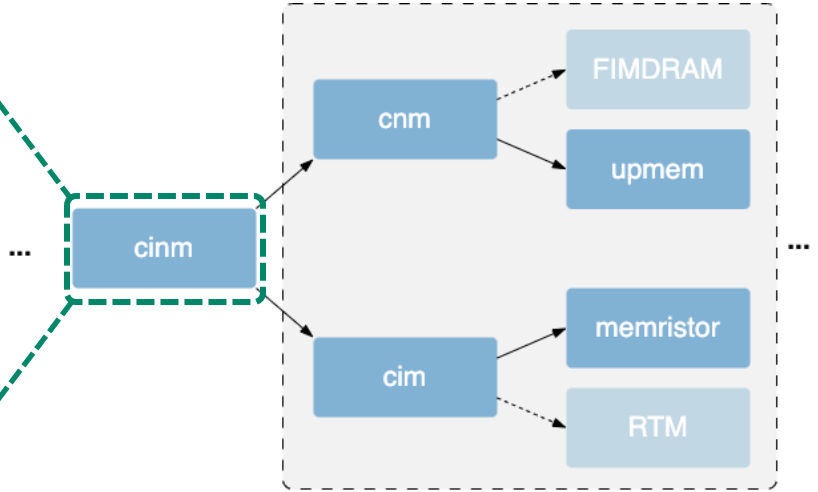
Conclusions

- Most recent innovations in computing systems center around memory
- Data movement dominates the execution time and energy consumption
- Near- and in-memory computing achieve orders of magnitude energy savings
- For these novel architectures to go mainstream, programmability/accessibility is the key, and requires more attention
- Cinnamon is a step in that direction, targeting heterogeneous CINM systems

Thanks!

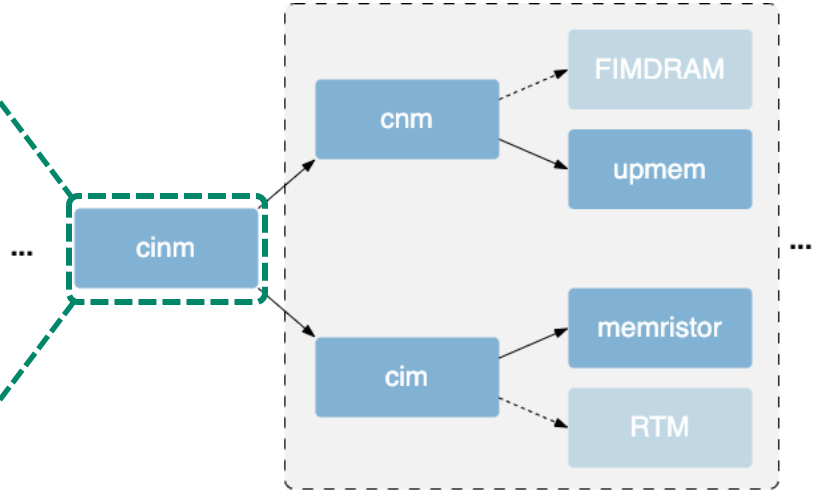
CINM: Common operations and transformations

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



CINM: Common operations and transformations

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

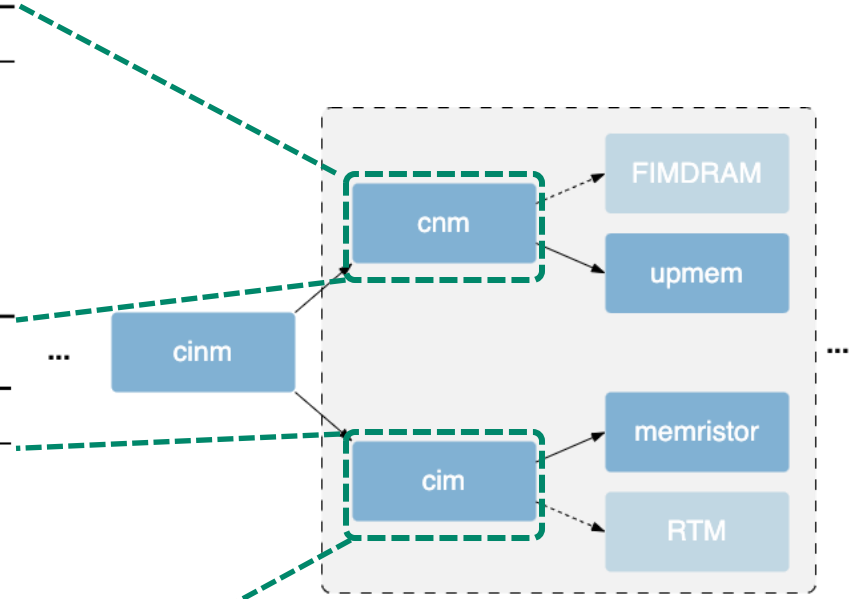


+ Common transformations: Tiling, loop reorder, operation fusion, operation rewriting , …

CNM/CIM: Abstraction programming models

Operation	Description
cnm.allocate(%arg, %arg2)	Allocate workgroup on the specified CNM device.
cnm.launch(%arg, %arg)	Launch the workgroup execution.
cnm.scatter(%arg, %arg2)	Move specified elements' indices of the input tensor to the destination tensor.
cnm.gather(%arg, %arg2)	Symmeterical to scatter, copy back.
cnm.wait(%arg, %arg2)	Wait to synchronize.

Operation	Description
cim.acquire()	Acquire a CIM device, returns ID.
cim.write(%arg, %arg2)	Write specified input tensor to the acquired CIM device.
cim.execute(%arg, %arg)	Launch the execution on the acquired CIM device.
cim.read(%arg)	Read data from the acquired CIM device.
cim.barrier(%arg, %arg2)	Wait to synchronize or finish executing.
cim.release(%arg)	Release the device.



CNM/CIM: Abstraction programming models

Operation	Description
cnm.allocate(%arg, %arg2)	Allocate workgroup on the specified CNM device.
cnm.launch(%arg, %arg)	Launch the workgroup execution.
cnm.scatter(%arg, %arg2)	Move specified elements' indices of the input tensor to the destination tensor.
cnm.gather(%arg, %arg2)	Symmeterical to scatter, copy back.
cnm.wait(%arg, %arg2)	Wait to synchronize.

Operation	Description
cim.acquire()	Acquire a CIM device, returns ID.
cim.write(%arg, %arg2)	Write specified input tensor to the acquired CIM device.
cim.execute(%arg, %arg)	Launch the execution on the acquired CIM device.
cim.read(%arg)	Read data from the acquired CIM device.
cim.barrier(%arg, %arg2)	Wait to synchronize or finish executing.
cim.release(%arg)	Release the device.