Automatic High-Performance Kernel Generation for DL Accelerators : Specialization to Generalization*

Prasanth Chatarasi

Research Staff Member @ AI Hardware Group,
IBM T.J. Watson Research Center,
YorkTown Heights, NY, USA
https://www.research.ibm.com/artificial-intelligence/hardware/
prasanth@ibm.com

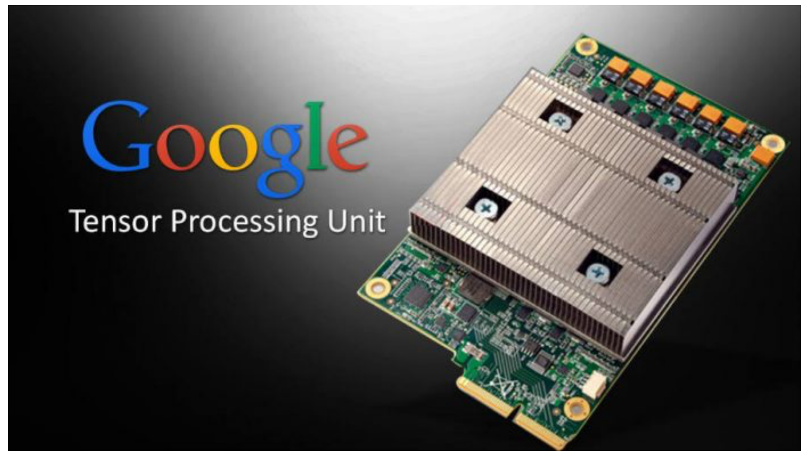
Google Brain, May 26th, 2021

Georgia Tech School of Computer Science

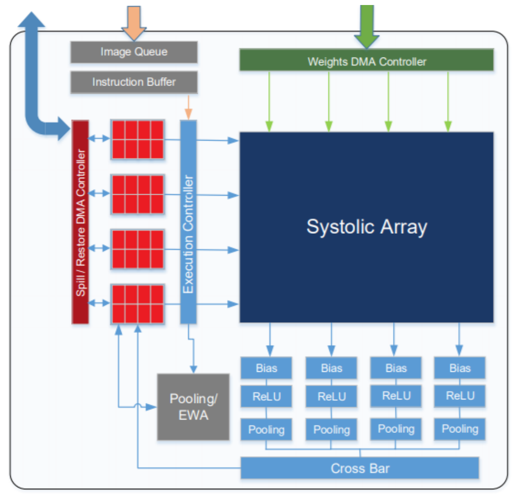
*Work done during PhD studies in Habanero Research Group
at Georgia Institute of Technology

Deep Learning (DL) Accelerators

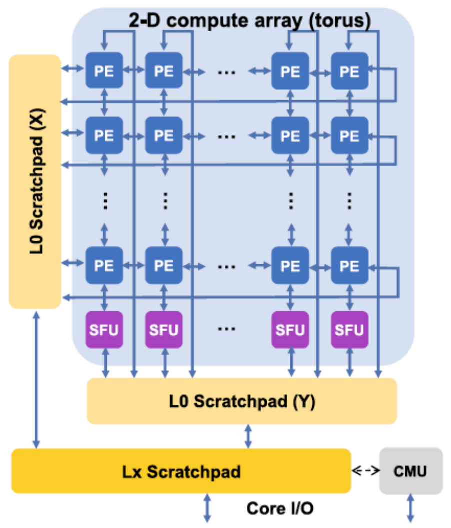
- **Emerged to address the demands of DL models training and inferences**
  - A large array of processing elements to provide high performance
  - Direct communication between PEs for energy efficiency
    - PE & PE requires ~3x less energy compared to PE & L2



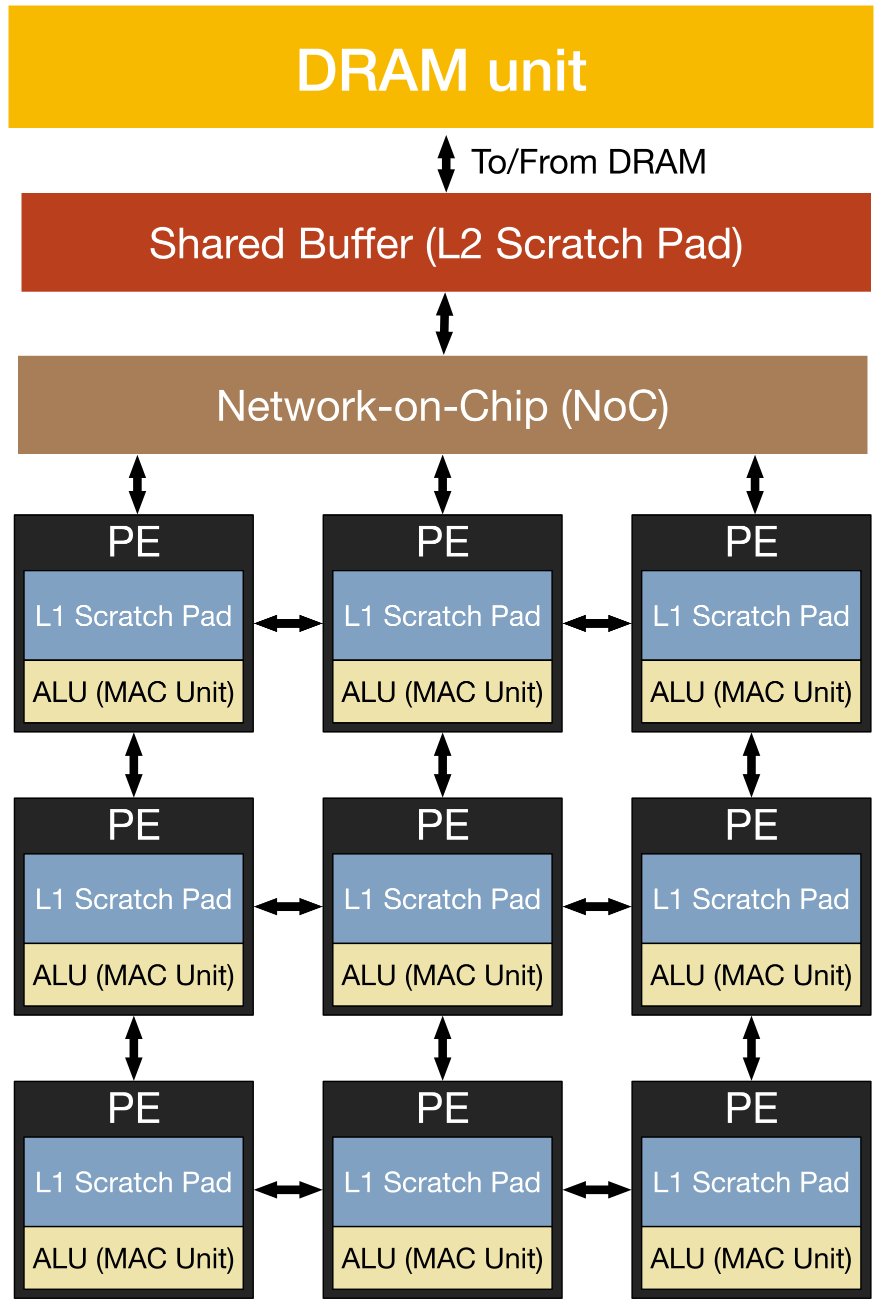
TPU, Google



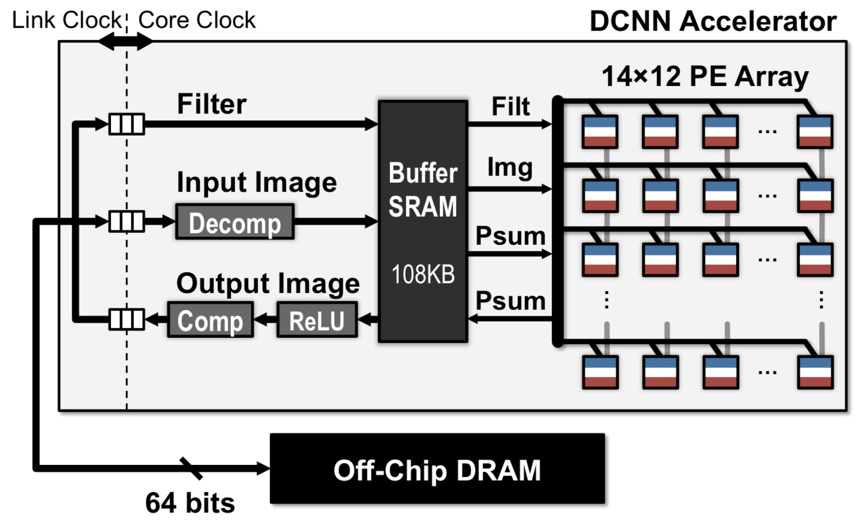
xDNN, Xilinx



RAPID, IBM

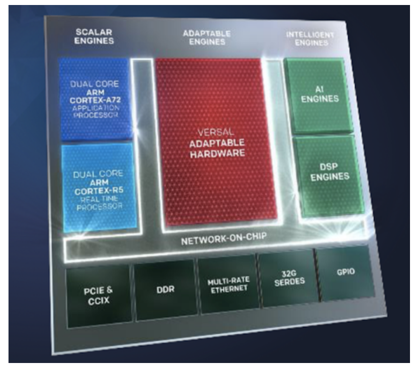


Abstract template



Eyeriss, MIT

DLA, NVIDIA



AI Engine, Xilinx (Versal)

Design Architecture of Deep Learning (DL) Compilers

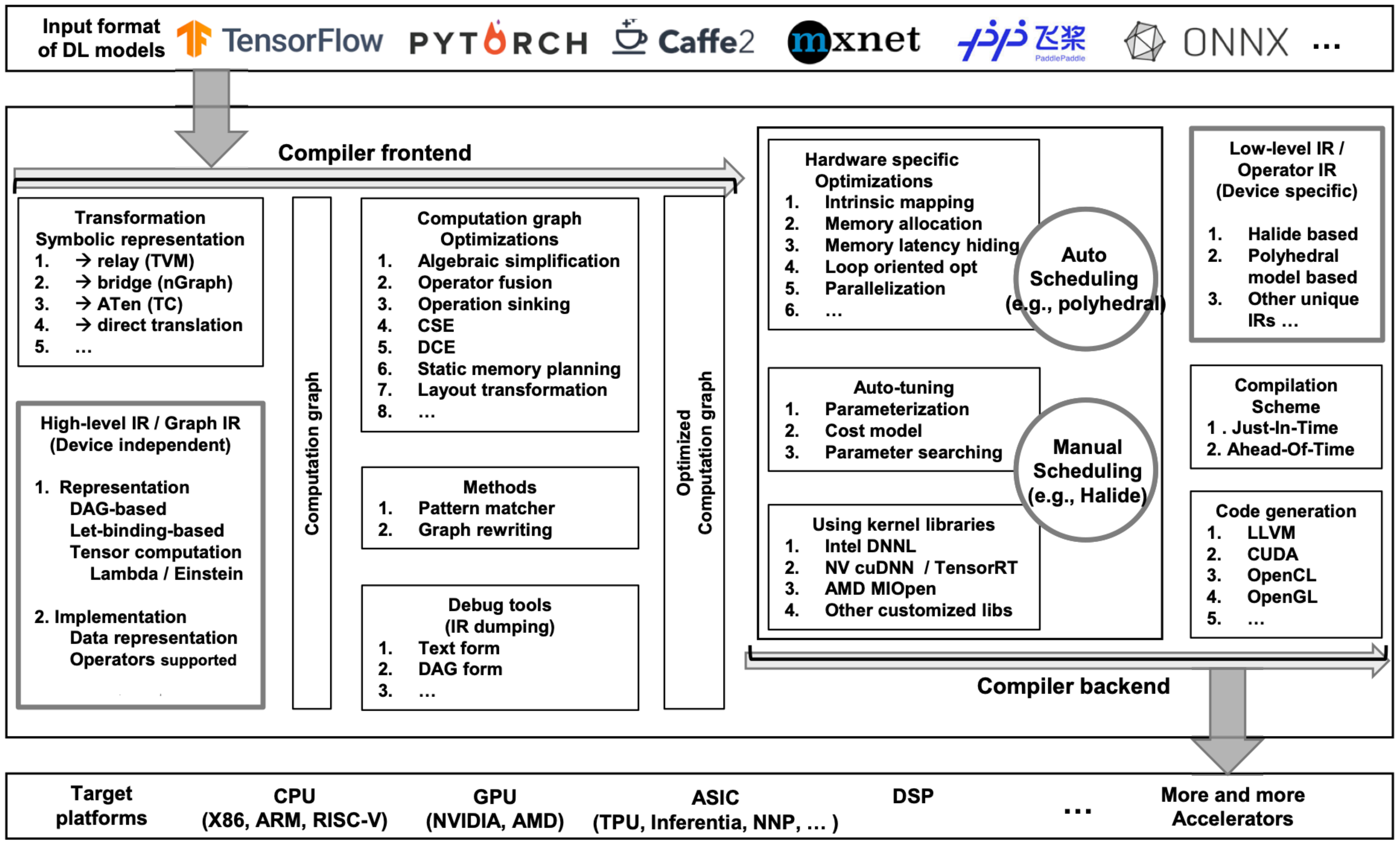


Fig. 2. The overview of commonly adopted design architecture of DL compilers.

Key steps in the flow:

- Graph compiler performing graph-level optimizations
  - e.g., Node/Layer fusion
- Kernel compiler performing kernel-level optimizations
  - e.g., loop scheduling

Li et. al., "*The Deep Learning Compiler: A Comprehensive Survey*", ArXived 2020

Kernel Compilers of DL Compiler stack

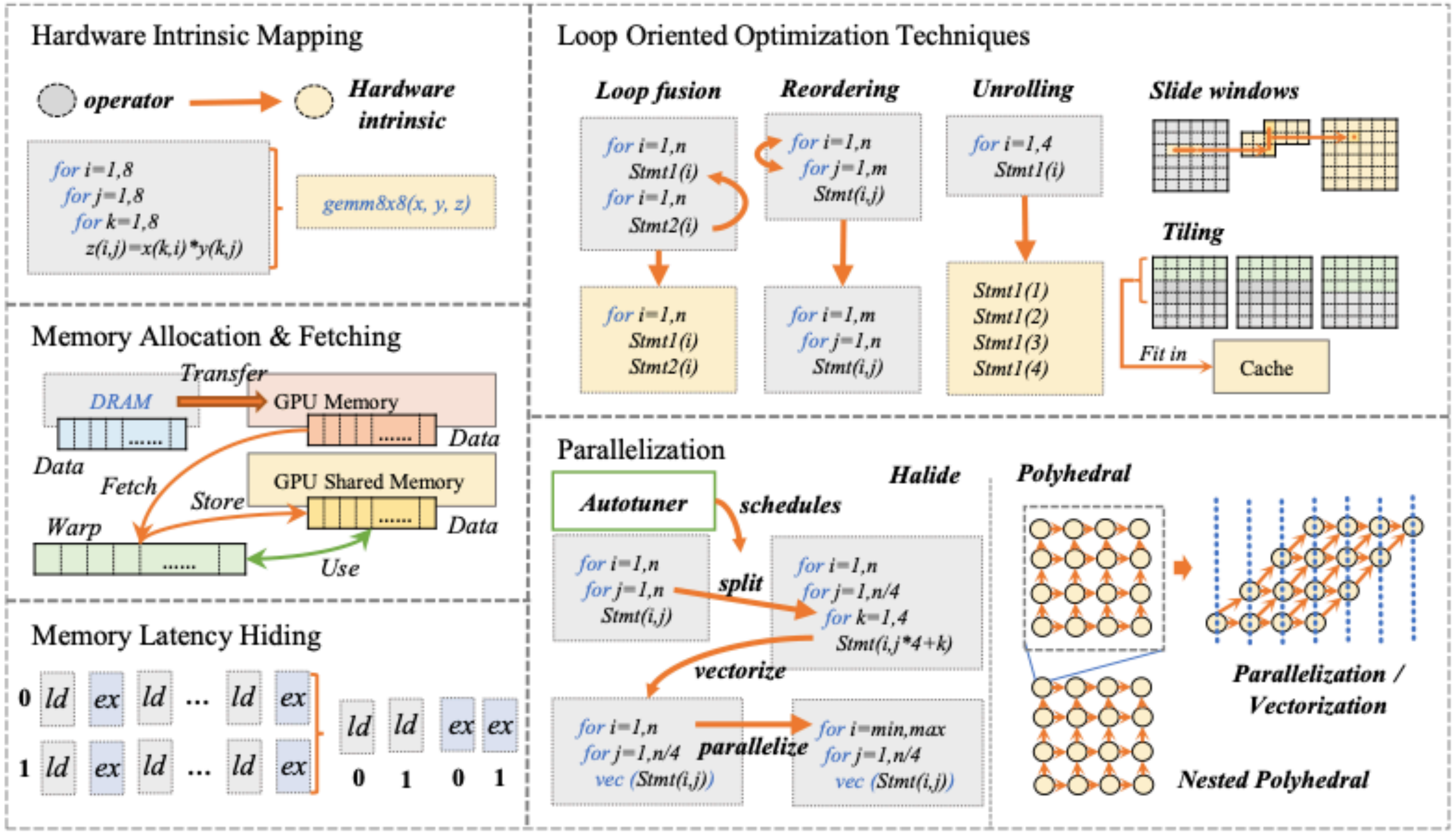


Fig. 4. Overview of hardware-specific optimizations applied in DL compilers.

Three major approaches for kernel compilers:

- Stitch manually-written kernel libraries for each node of the optimized graph
- Automatically generate entire kernels corresponding to each node of the graph
- Hybrid approach combining some parts with manually-written kernels and other parts with automatic generation

Li et. al., "*The Deep Learning Compiler: A Comprehensive Survey*", ArXived 2020

Manually-written vs Automatically-generated

1. **Manually-writing kernels is not a scalable approach, but a good temporary solution!**
    - Kernels are evolving rapidly, for, e.g., many variants of convolutions with different shapes/sizes
    - ML Accelerators also evolving so quickly, for, e.g., TPU V1, V2, and V3 with different capabilities
    - ***Need automatic kernel generation along with mappers and cost models that does fine-grained reasoning of both kernels and hardware to achieve peak performance!***

2. **Automatic kernel generation is a scalable solution, but several challenges.**
    - Need "GOOD" hardware abstractions to capture various accelerators
    - Need "GOOD" mapping abstractions to capture various mapping strategies of workloads
    - Need "GOOD" cost models to estimate performance and drive mappers/auto-tuners
    - Several variants:
        - Fixed hardware + Fixed kernel,
        - Fixed hardware + Allow different kernel possibilities
        - Allow various hardware choices + Allow different kernel possibilities

Overview of today's talk

1. Introduction & Background

2. Vyasa: A High-performance Vectorizing Compiler for Tensor Operations onto Xilinx AI Engine (2D SIMD unit)
    - Fixed hardware + Allow different kernel possibilities

3. PolyEDDO: A Polyhedral-based Compiler for Explicit De-coupled Data Orchestration (EDDO) architectures
    - Allow various hardware choices + Allow different kernel possibilities

4. Conclusions

Overview of today's talk

1. Introduction & Background

2. **Vyasa: A High-performance Vectorizing Compiler for Tensor Operations onto Xilinx AI Engine (2D SIMD unit)**
   - **Fix hardware + Allow different kernel possibilities**

3. PolyEDDO: A Polyhedral-based Compiler for Explicit De-coupled Data Orchestration (EDDO) architectures
   - Allow various hardware choices + Allow different kernel possibilities

4. Conclusions

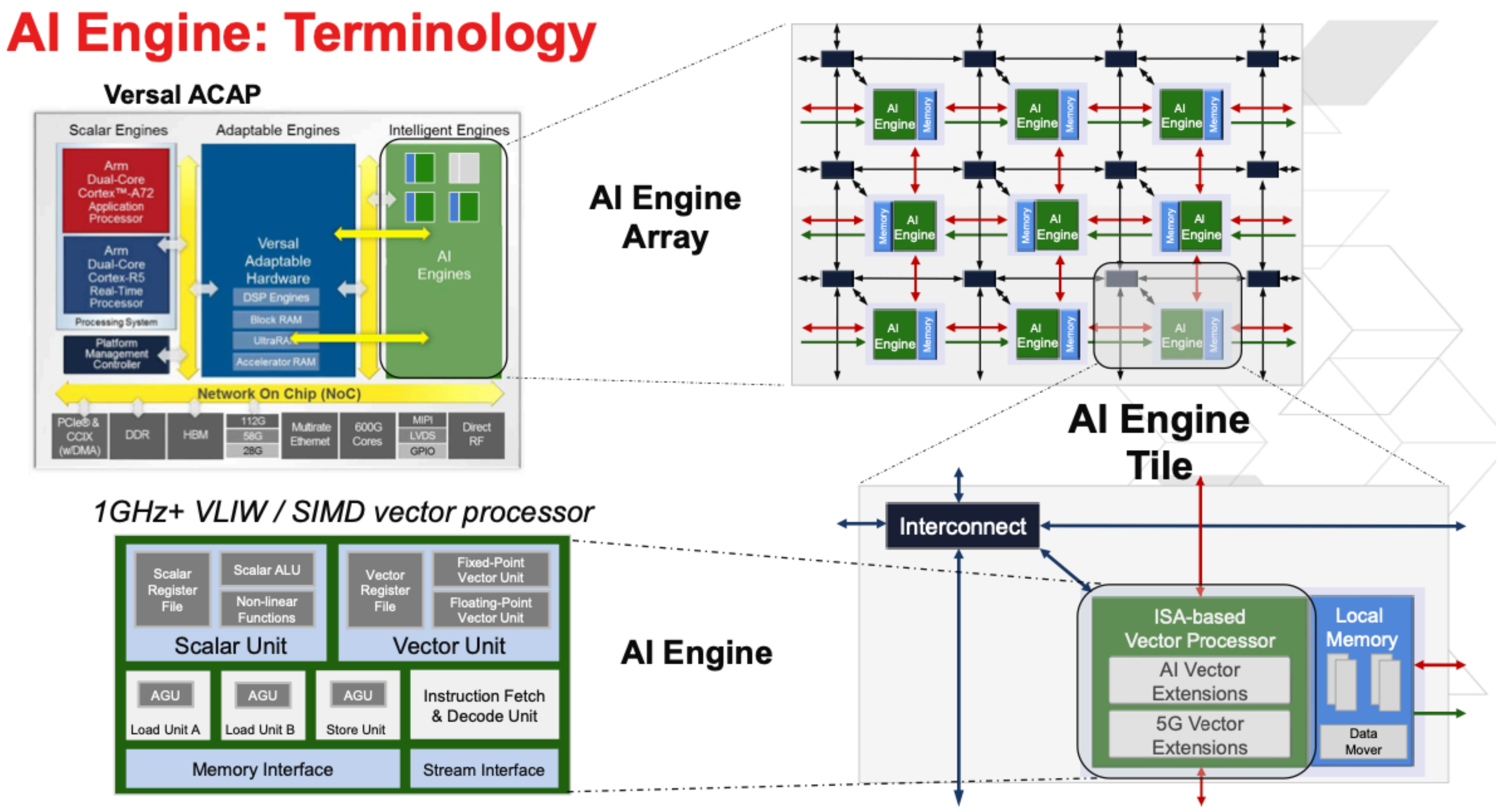
"Vyasa: A High-Performance Vectorizing Compiler for Tensor Convolutions on the Xilinx AI Engine"
Prasanth Chatarasi, Stephen Neuendorffer, Samuel Bayliss, Kees Vissers, and Vivek Sarkar
Proceedings of the 24th IEEE High Performance Extreme Computing Conference (HPEC'20)

Georgia Tech School of Computer Science

XILINX

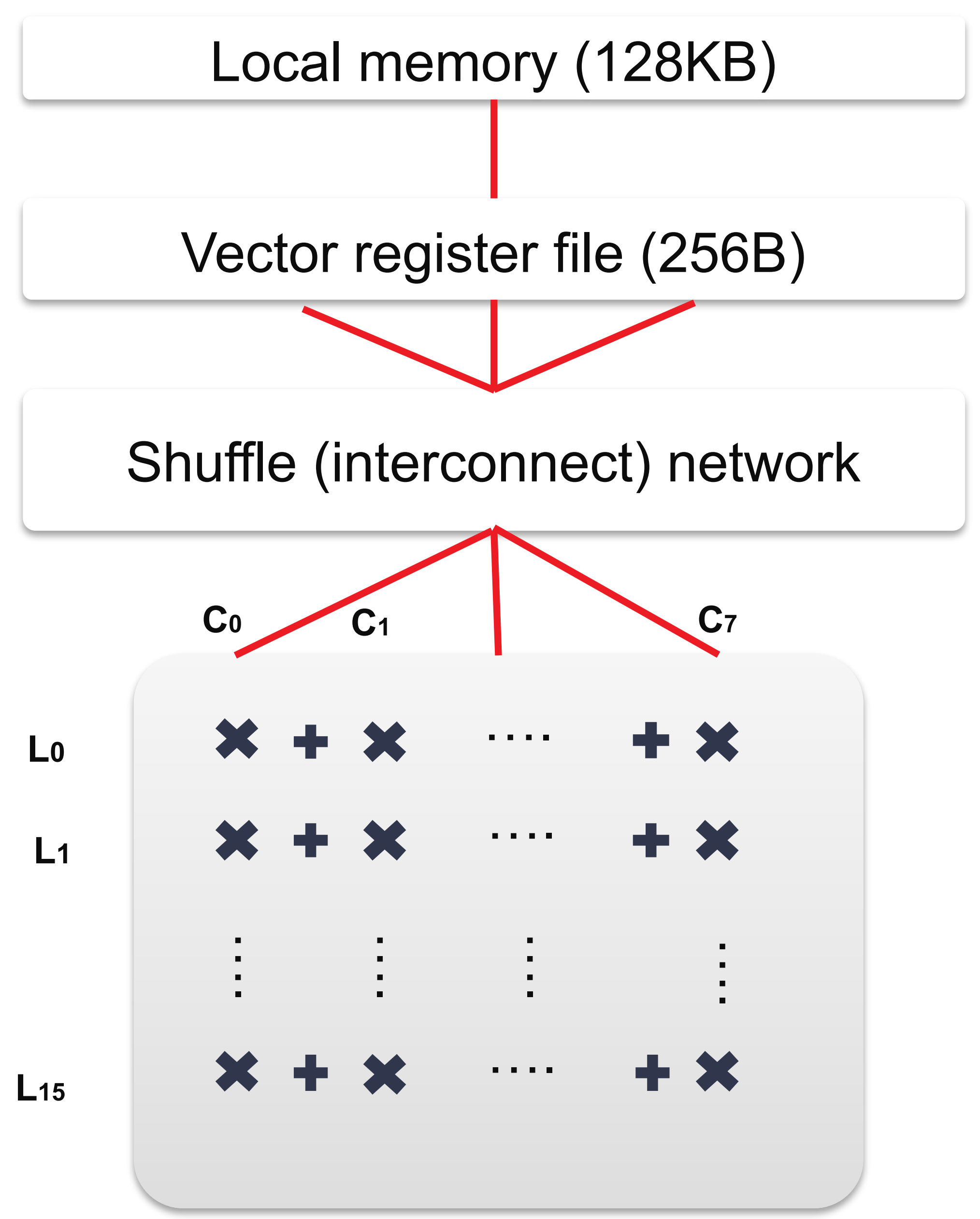
Xilinx Versal AI Engine

- **A High Performance & Power Efficient VLIW SIMD Core**
  - Part of Xilinx Versal Advanced Compute Acceleration Platform
    - Scalar Engines (CPUs), Adaptable Engines (Programmable logic), Intelligent Engines (AI Engines)



Key architectural features of AI Engine

Abstract view of AI Engine



Fixed Point SIMD Unit

1) **2D SIMD datapath for fixed point**
    - Reduction within a row/lane
    - #Columns depend on operand precision
        - 32-bit types: 8 rows x 1 col
        - 16-bit types: 8 rows x 4 col (or) 16 rows x 2 col
        - 8-bit types: 16 rows x 8 col

2) **Shuffle Interconnection network**
    - Between SIMD and vector register file
    - Supports arbitrary selection of elements from a vector register
        - Some constraints for 16-/8-bit types
    - Selection parameters are provided via vector intrinsics

Problem Statement & Challenges

Problem statement: How to implement high-performance primitives for tensor convolutions on AI Engine?

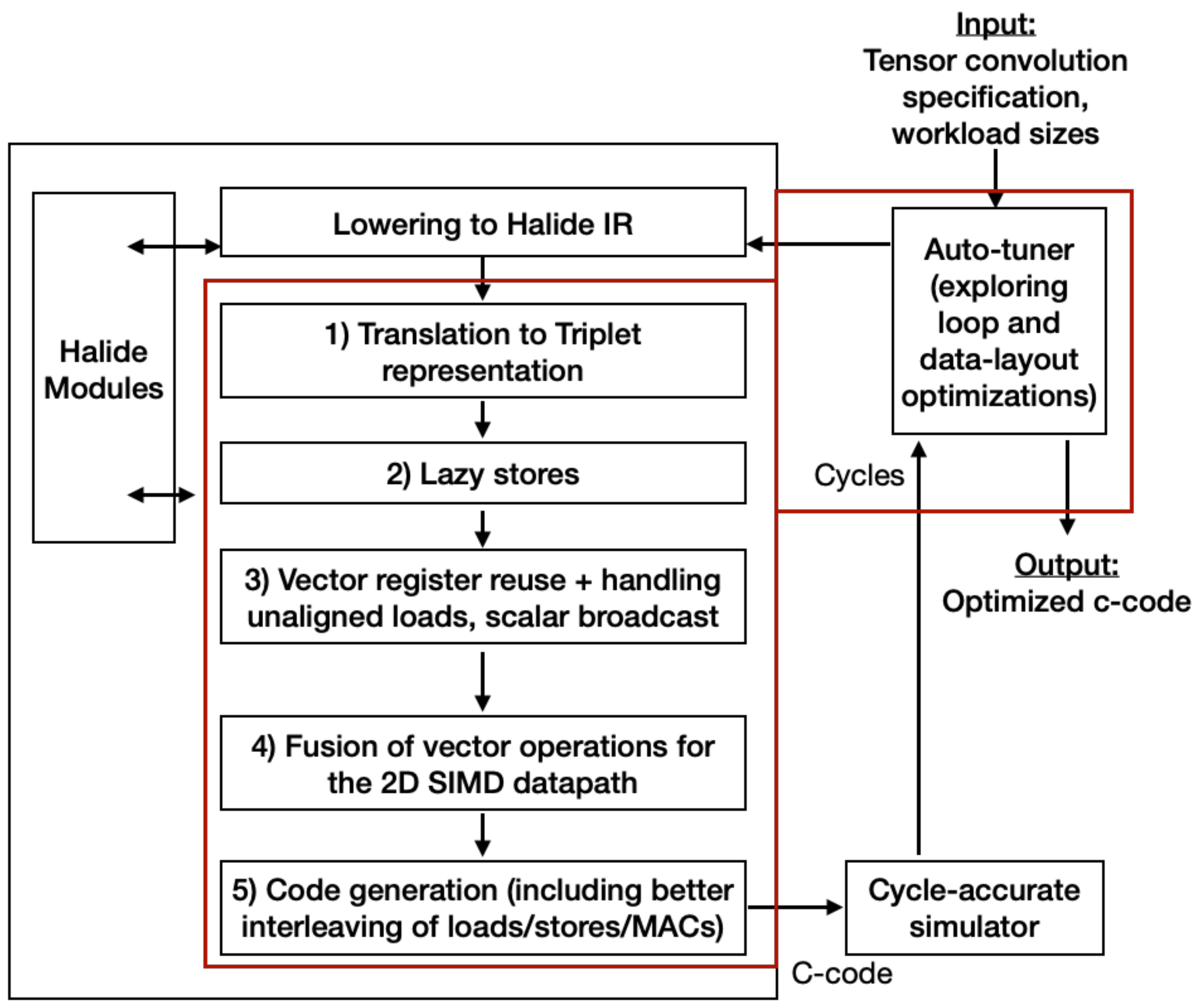
- Current practice: Programmers manually use vector intrinsics to program 2D SIMD unit and also explicitly specify shuffle network parameters for data selection

- Challenges: Error prone, written code may not be portable to a different schedule or data-layouts, daunting to explore all choices to find best implementation, tensor convolutions vary in sizes and types

Our approach: Vyasa, a domain-specific compiler to generate high performance primitives for tensor convolutions from a high-level specification

Vyasa means “compiler” in the Sanskrit language,
and also refers to the sage who first compiled the Mahabharata.

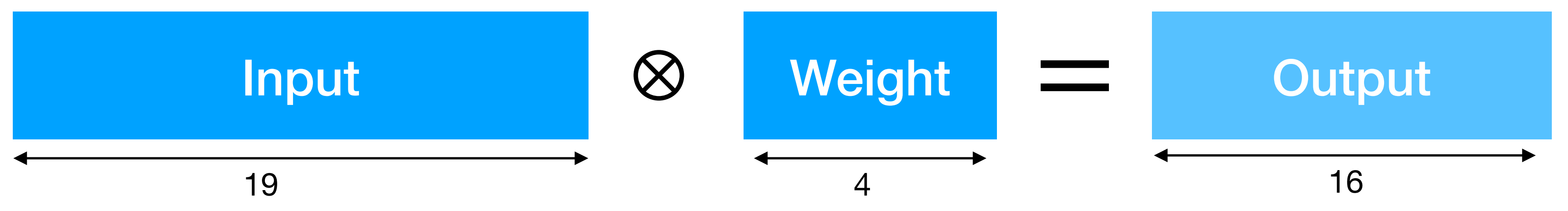
Our high-level approach (Vyasa)



Input:
Tensor convolution specification, workload sizes
Lowering to Halide IR
Halide Modules
1) Translation to Triplet representation
2) Lazy stores
3) Vector register reuse + handling unaligned loads, scalar broadcast
4) Fusion of vector operations for the 2D SIMD datapath
5) Code generation (including better interleaving of loads/stores/MACs)
Auto-tuner (exploring loop and data-layout optimizations)
Cycles
Output:
Optimized c-code
Cycle-accurate simulator
C-code

In this talk, I focus on Step-3 and Step-4 leveraging Shuffle Network and 2D SIMD datapath!

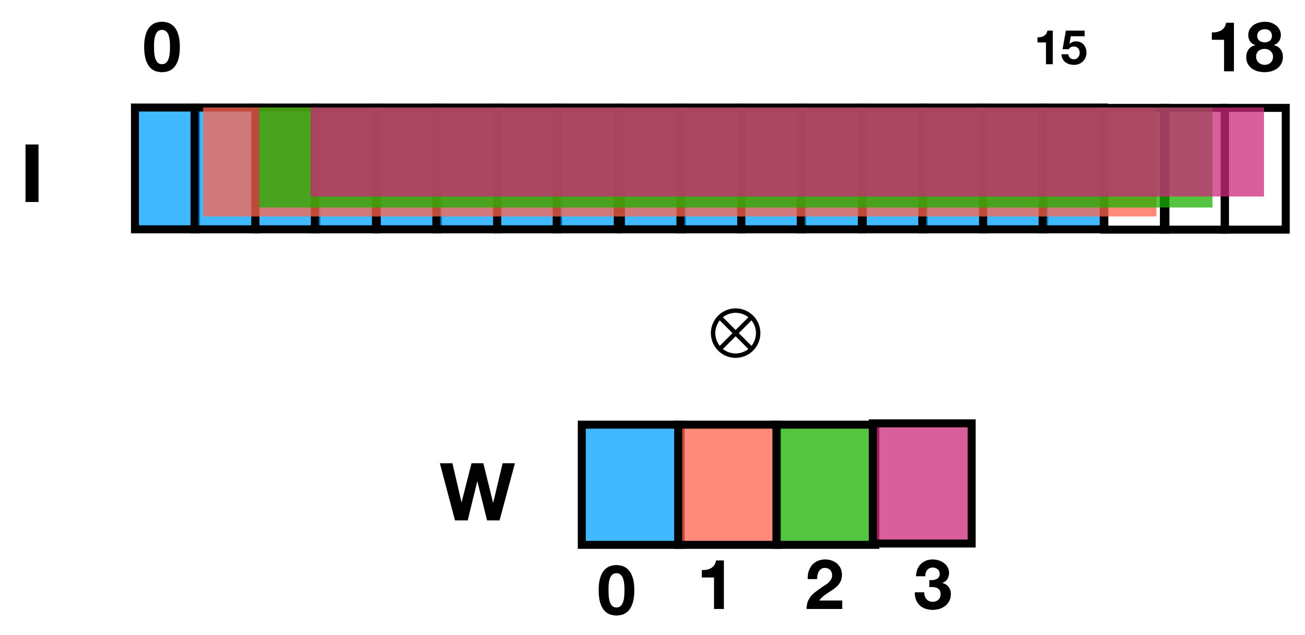
Running Example — CONV1D



```
for(x=0; x < 16; x++)
  for(w=0; w < 4; w++)
    O[x] += I[x+w]*W[w];
```

A sample schedule: Unroll w-loop and Vectorize x-loop (VLEN: 16)

*O(0:15) += W(0) * I(0:15)*
*O(0:15) += W(1) * I(1:16)*
*O(0:15) += W(2) * I(2:17)*
*O(0:15) += W(3) * I(3:18)*



Challenges

```
O(0:15) += W(0) * I(0:15)
O(0:15) += W(1) * I(1:16)
O(0:15) += W(2) * I(2:17)
O(0:15) += W(3) * I(3:18)
```

```
V1 = VLOAD(I, 0:15);
V2 = BROADCAST(W, 0);
V3 = VMAC(V1, V2);

V4 = VLOAD(I, 1:16);
V5 = BROADCAST(W, 1);
V3 = VMAC(V3, V4, V5);

V6 = VLOAD(I, 2:17);
V7 = BROADCAST(W, 2);
V3 = VMAC(V3, V6, V7);

V8 = VLOAD(I, 3:18);
V9 = BROADCAST(W, 3);
V3 = VMAC(V3, V8, V9);
VSTORE(O, 0:15, V3);
```

No support for unaligned loads

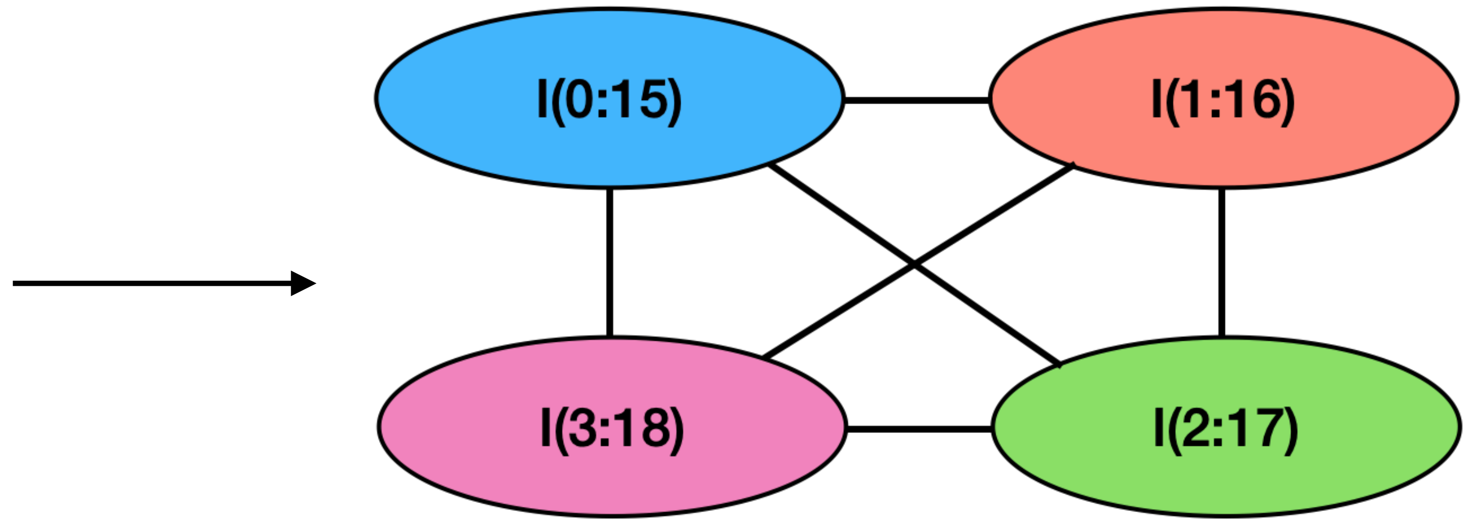
No support for broadcast operations

V6 and V8 have 15 elements in common. How to reuse them without loading again?

How to exploit multiple columns of 2D vector substrate?

1) Exploiting Vector Register Reuse

*O(0:15) += W(0) * I(0:15)*
*O(0:15) += W(1) * I(1:16)*
*O(0:15) += W(2) * I(2:17)*
*O(0:15) += W(3) * I(3:18)*



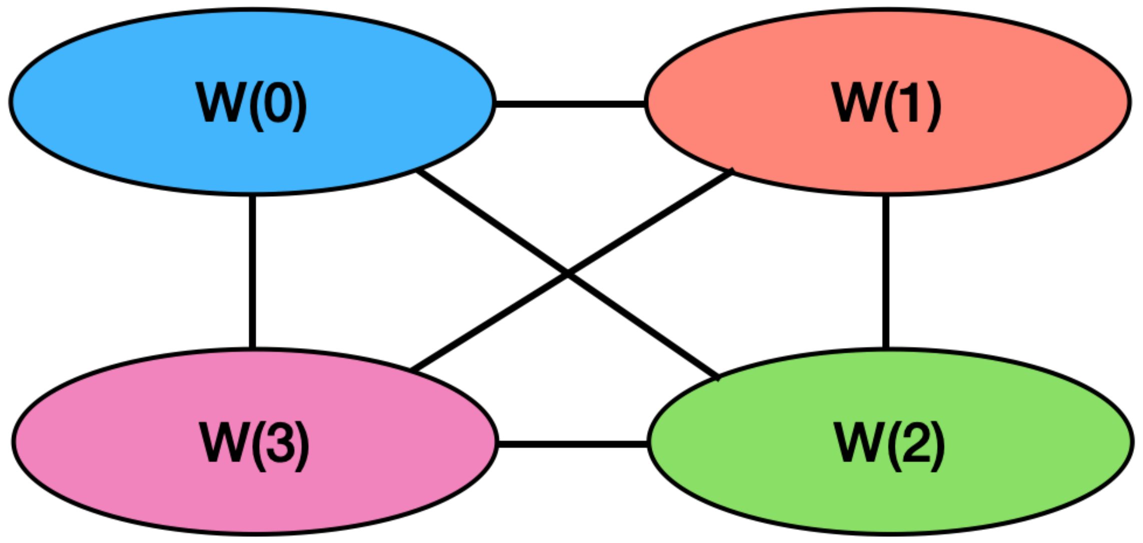
Connected component
V1 — I(0:31)

- Build "*temporal reuse graph*" with nodes being vector loads
  - Edge exists b/w nodes if there is at least one element in common

- <u>AI Engine allows to create logical vector registers of length up to 1024 bits</u>
  - Identify (aligned) connected components and assign each component to a vector register that can subsume the individual vector loads of the component.
  - Use shuffle interconnection network to select desired elements

2) Exploiting Spatial Locality

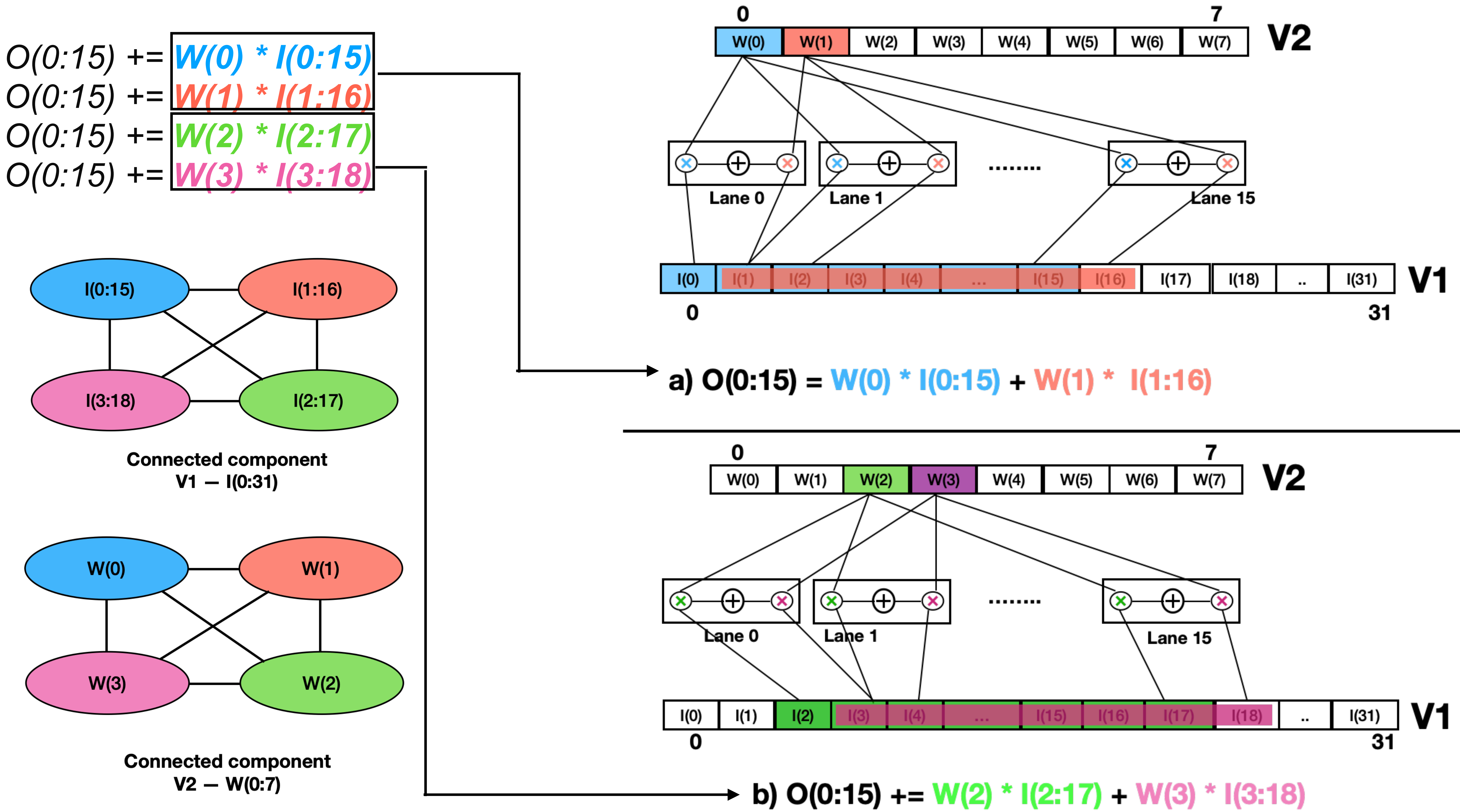
$$\begin{array}{r|c|l}
\cline{2-2}
O(0{:}15) \mathrel{+}= & W(0) & * \; I(0{:}15) \\
O(0{:}15) \mathrel{+}= & W(1) & * \; I(1{:}16) \\
O(0{:}15) \mathrel{+}= & W(2) & * \; I(2{:}17) \\
O(0{:}15) \mathrel{+}= & W(3) & * \; I(3{:}18) \\
\cline{2-2}
\end{array}$$



Connected component
V2 — W(0:7)

- Build "*spatial locality graph*" with nodes being scalar loads
  - Edge exists b/w nodes if they can be part of an aligned vector load
- Identify connected components
- AI Engine allows to create logical vector registers of length up to 1024 bits
  - Assign each connected component (aligned) to a logical vector register
  - Use shuffle interconnection network to select desired elements

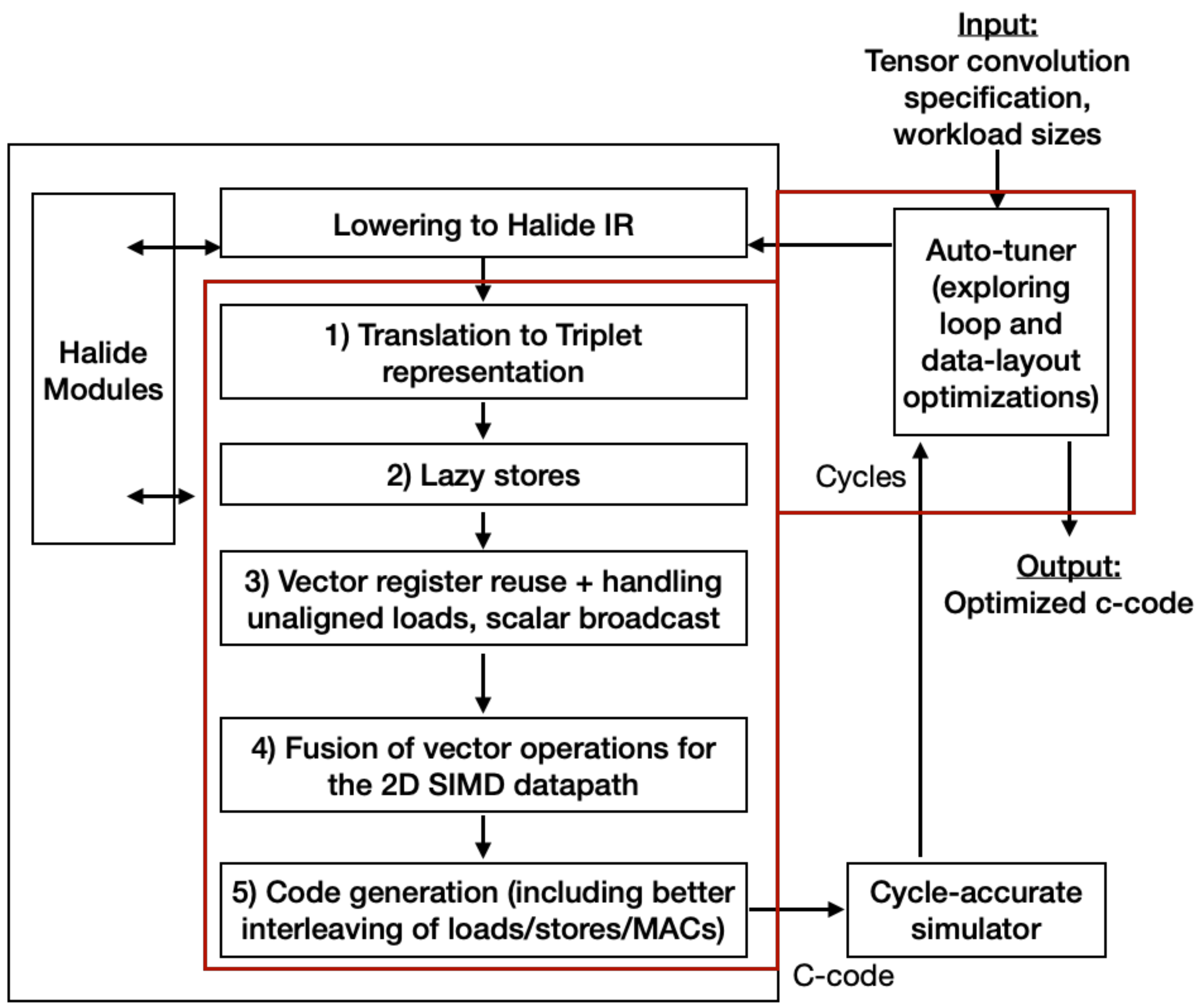
3) Grouping 1D Vector Operations



O(0:15) += W(0) * I(0:15)
O(0:15) += W(1) * I(1:16)
O(0:15) += W(2) * I(2:17)
O(0:15) += W(3) * I(3:18)
I(0:15)
I(1:16)
I(3:18)
I(2:17)
Connected component
V1 — I(0:31)
W(0)
W(1)
W(3)
W(2)
Connected component
V2 — W(0:7)
0
7
W(0) W(1) W(2) W(3) W(4) W(5) W(6) W(7)
V2
Lane 0
Lane 1
....
Lane 15
I(0) I(1) I(2) I(3) I(4) ... I(15) I(16) I(17) I(18) .. I(31)
V1
0
31
a) O(0:15) = W(0) * I(0:15) + W(1) * I(1:16)
b) O(0:15) += W(2) * I(2:17) + W(3) * I(3:18)

<u>All the 4 operations are performed with a single load of V1 and V2 (maximum reuse)</u>

Our high-level approach (Vyasa)


Input:
Tensor convolution specification, workload sizes
Lowering to Halide IR
Halide Modules
1) Translation to Triplet representation
2) Lazy stores
3) Vector register reuse + handling unaligned loads, scalar broadcast
4) Fusion of vector operations for the 2D SIMD datapath
5) Code generation (including better interleaving of loads/stores/MACs)
Auto-tuner (exploring loop and data-layout optimizations)
Cycles
Output:
Optimized c-code
Cycle-accurate simulator
C-code

Auto-tuner explores the space of schedules related to loop and data-layouts.

Loop transformations:

1. Choice of vectorization loop
2. Loop reordering
3. Loop unroll and jam

Data-layout choices:

1. Data permutation
2. Data tiling (blocking)

We assume that workload memory footprint fits into a AI Engine local scratchpad memory (128KB)

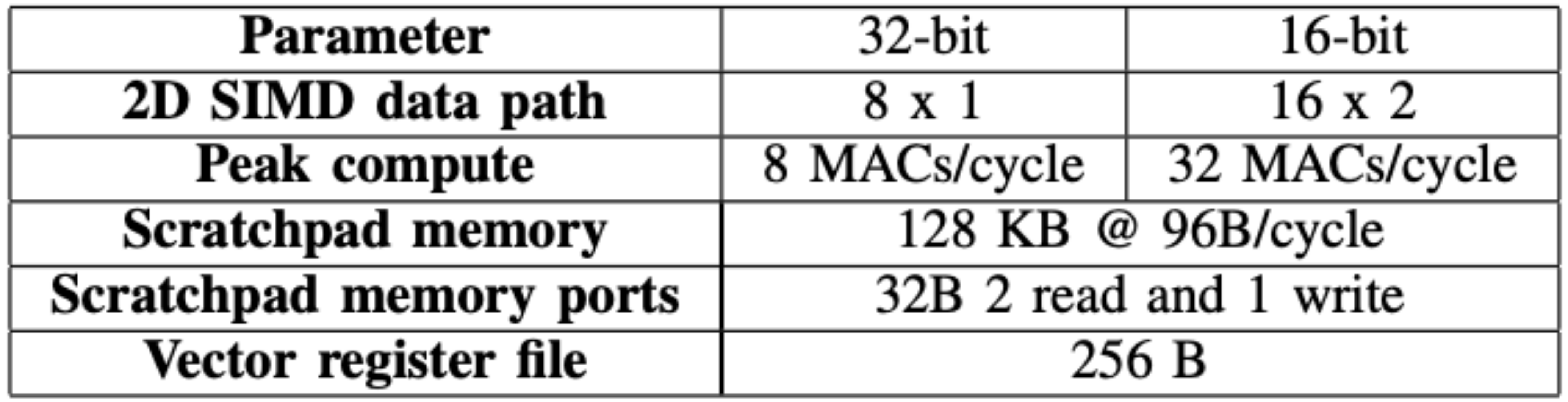
Evaluation

- **CONV2D workloads (only for this talk)**
  - CONV2D in Image processing pipelines/Computer Vision (CV)

    HALIDE CODE: O(x, y) += W(r, s) * I(x+r, y+s);
  - CONV2D in Convolutional Neural Networks (CNNs)

    *HALIDE CODE: O(x, y, k, n) += W(r, s, c, k) * I(x+r, y+s, c, n);*

AI Engine configurations

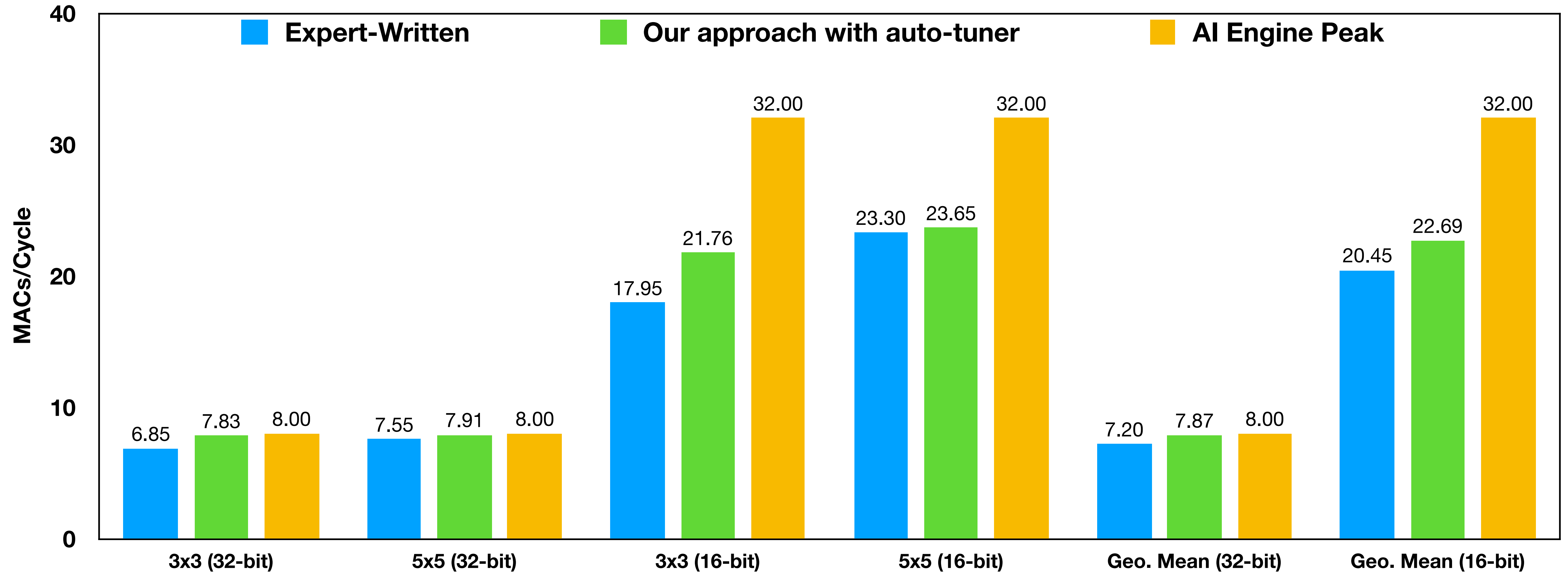


Parameter	32-bit	16-bit
2D SIMD data path	8 x 1	16 x 2
Peak compute	8 MACs/cycle	32 MACs/cycle
Scratchpad memory	128 KB @ 96B/cycle	
Scratchpad memory ports	32B 2 read and 1 write	
Vector register file	256 B	

- **Comparison variants**
  - Roofline peak
    - 32-bit types: 8 MACs/cycle
    - 16-bit types: 32 MACs/cycle
  - Expert-written and tuned kernels for Computer Vision

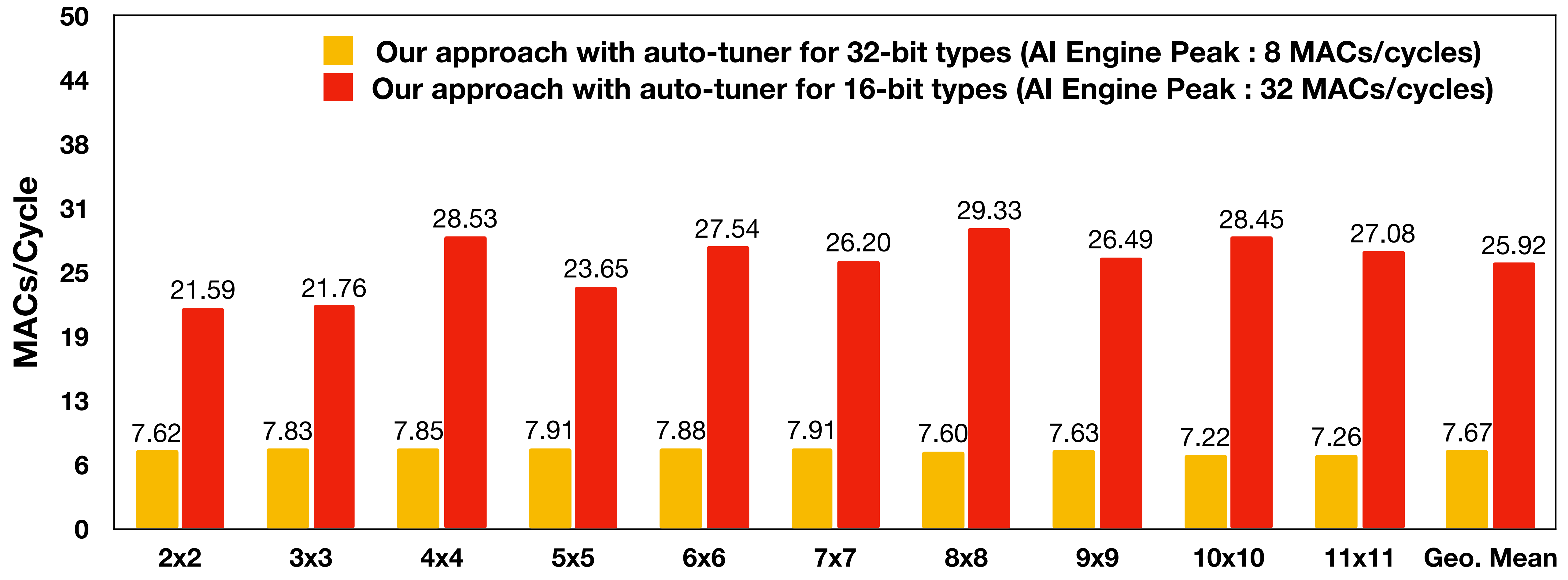
Evaluation: CONV2D's in CV (256x16)

HALIDE CODE: O(x, y) += W(r, s) * I(x+r, y+s);



- *Expert-written codes* are available only for 3x3 and 5x5 filters
  - Available as part of the Xilinx's AI Engine compiler infrastructure
- *Auto-tuner was able to find better schedules*
  - Especially non-trivial unroll and jam factors

Evaluation: CONV2D's in CV (256x16) with various Filters

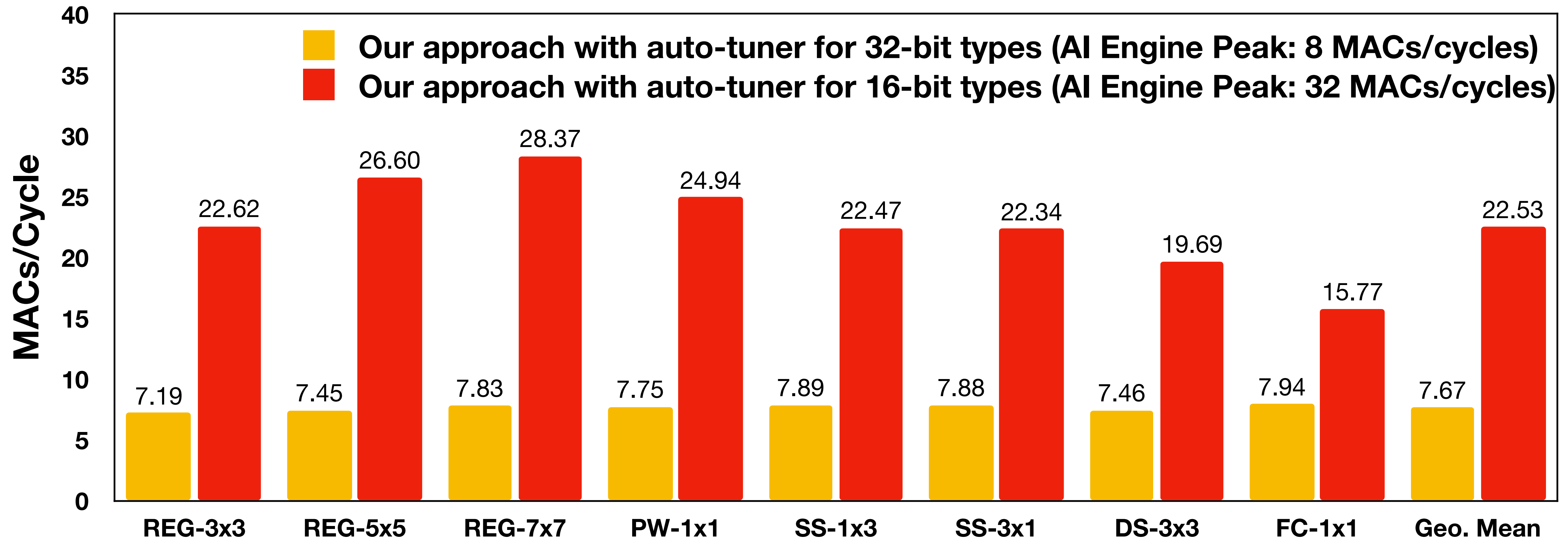


- *Even-sized filters (except 2x2), our approach achieved close to peak*
  - 87% for 16-bit and 95% for 32-bit

- *Odd-sized filters, our approach padded each row with an additional column*
  - For 16-bit type, number of reductions should be multiple of two (2 columns)

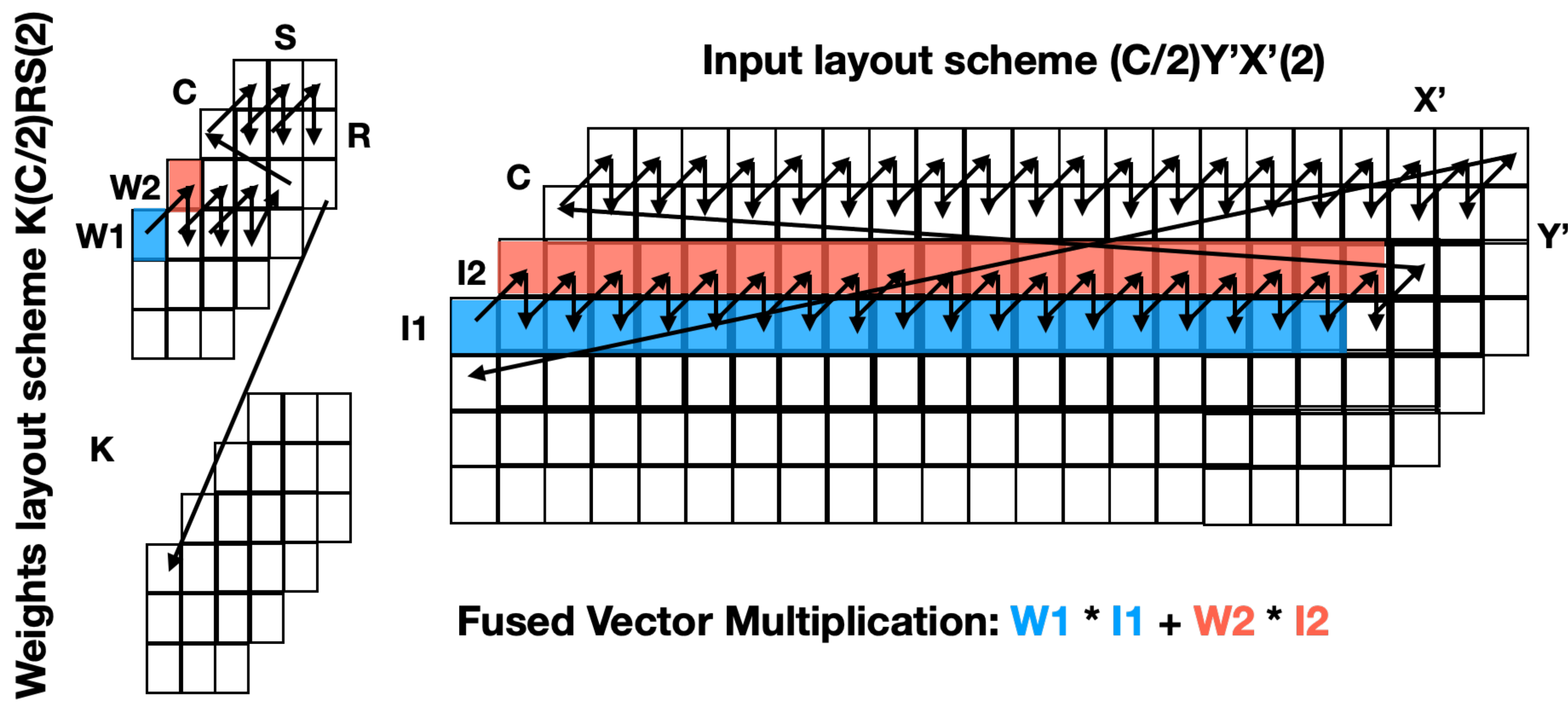
Evaluation: CONV2D's in CNN's (128x2x16)

<u>HALIDE CODE for REG CONV2D:</u> ***O(x, y, k, n) += W(r, s, c, k) * I(x+r, y+s, c, n);***



- *REG-CONV2D (3x3, 5x5, 7x7)*
  - Vectorization along Output width and Reduction along Filter channels
- *PW-CONV2D (1x1), SS-CONV2D (1x3, 3x1), FC-CONV2D (1x1)*
  - Vectorization along Output channels and Reduction along Filter channels
- *DS-CONV2D (3x3) — Padded each row*
  - Vectorization along Output width and Reduction along Filter width

Non-trivial data-layout choices


Weights layout scheme K(C/2)RS(2)
S
C
R
W2
W1
K
Input layout scheme (C/2)Y'X'(2)
X'
C
Y'
I2
I1
Fused Vector Multiplication: W1 * I1 + W2 * I2

- 16-bit *REG-CONV2D (3x3)*
  - Vectorization along Output width and Reduction along Filter channels
  - For the fused vector operation (W1xI1 + W2 x I2)
    - Data for (I1, I2) should be in a single vector register for the operation
    - I1(0) and I2(0) should be adjacent for shuffle network constraints
  - (C/2)Y'X'(2) refers to first laying out an input block of two channels followed by width, height, and remaining channels.

Summary and Questions

- ***Summary***
  - Manually writing vector code for high-performant tensor convolutions achieving peak performance is extremely challenging!
  - **Automatic kernel generation can be the key!**
    - Proposed a convolution-specific IR for easier analysis and transformations
    - Our approach (Vyasa) can work for any convolution variant regardless of its variations and shapes/sizes.
    - Achieved close to the peak performance for a variety of tensor convolutions
- ***Questions***
  - How about beyond tensor-style operations?
    - E.g., fused convolutions (depth-wise + point-wise), non-rectilinear iteration spaces (Symmetric GEMM)
  - How about beyond the AI Engine?
    - E.g., other accelerators like IBM Rapid, MIT Eyeriss, NVDLA…

Overview of today's talk

1. Introduction & Background

2. Vyasa: A High-performance Vectorizing Compiler for Tensor Operations onto Xilinx AI Engine (2D SIMD unit)
   - Fixed hardware + Allow different kernel possibilities

3. **PolyEDDO: A Polyhedral-based Compiler for Explicit De-coupled Data Orchestration (EDDO) architectures**
   - **Allow various hardware choices + Allow different kernel possibilities**

4. Conclusions

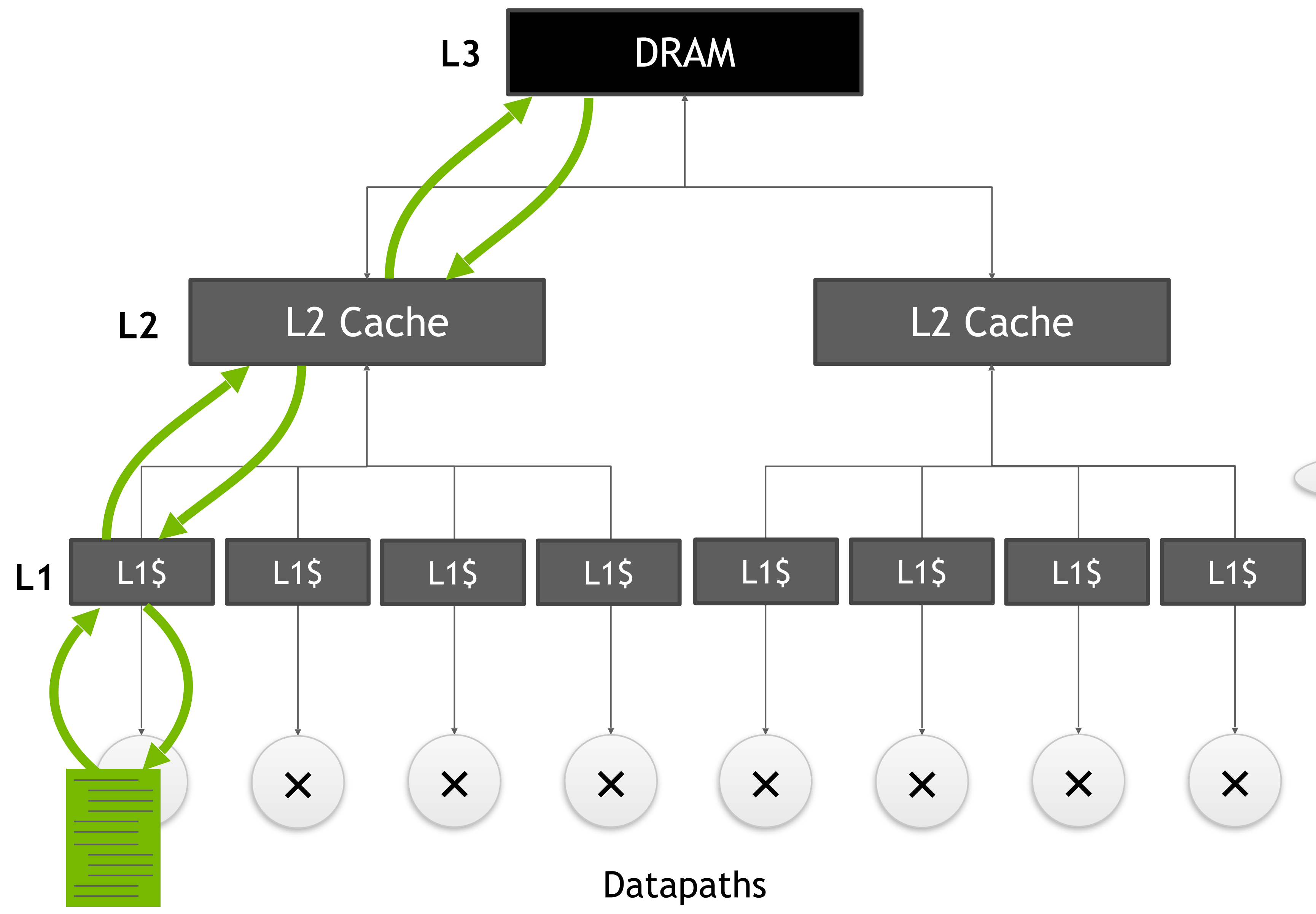
"Hardware Abstractions for targeting EDDO Architectures with the Polyhedral Model"
Angshuman Parashar, **Prasanth Chatarasi**, and Po-An Tsai,
11th International Workshop on Polyhedral Compilation Techniques (IMPACT'21)

Georgia Tech School of Computer Science

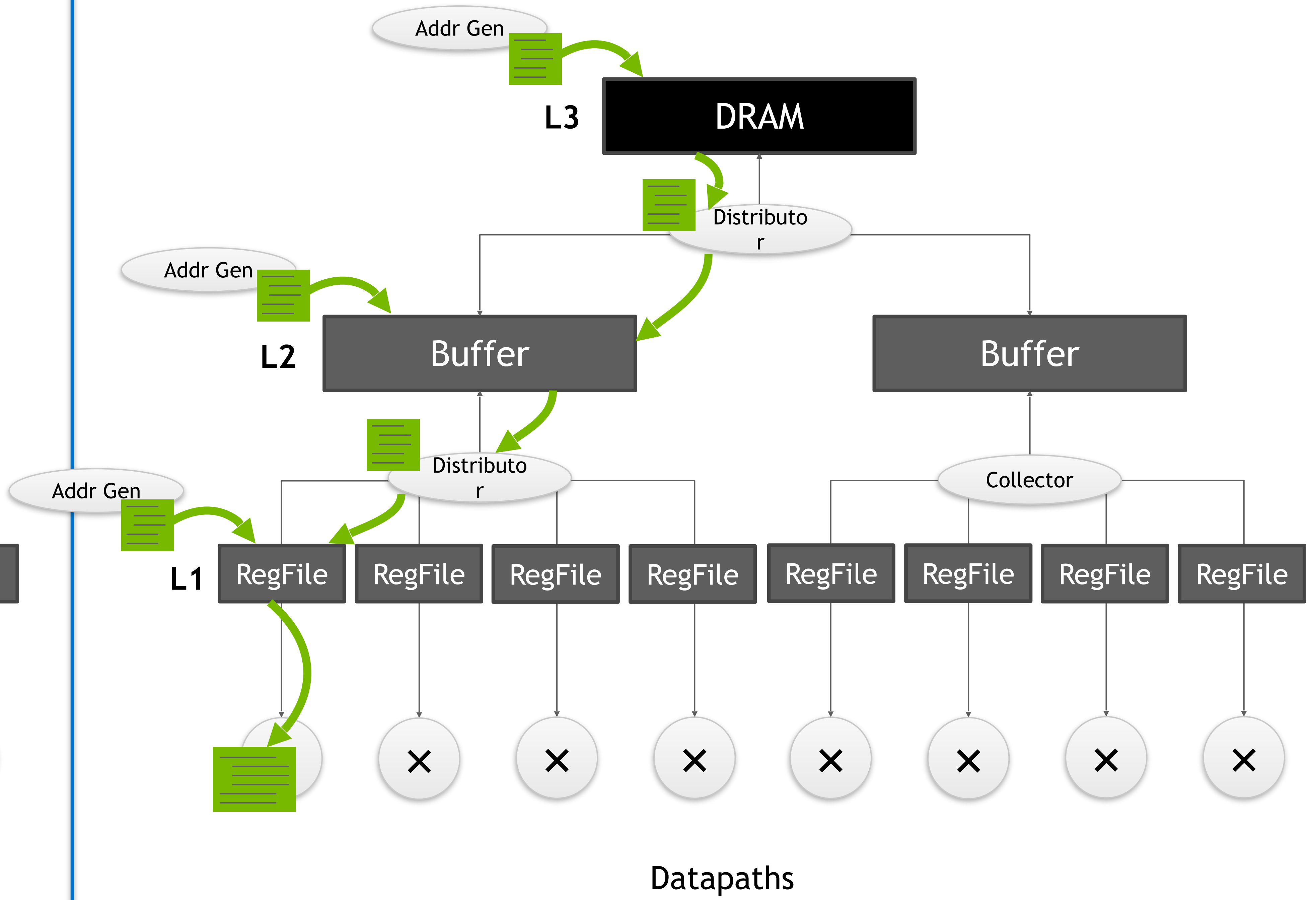
NVIDIA

ICDO vs EDDO Architectures

*Implicit Coupled Data Orchestration (**ICDO**)*
e.g., CPUs, GPUs



*Explicit Decoupled Data Orchestration (**EDDO**)*
e.g., IBM Rapid AI, NVIDIA Simba, NVDLA, Eyeriss, etc.



EDDO architectures attempt to minimize data movement costs

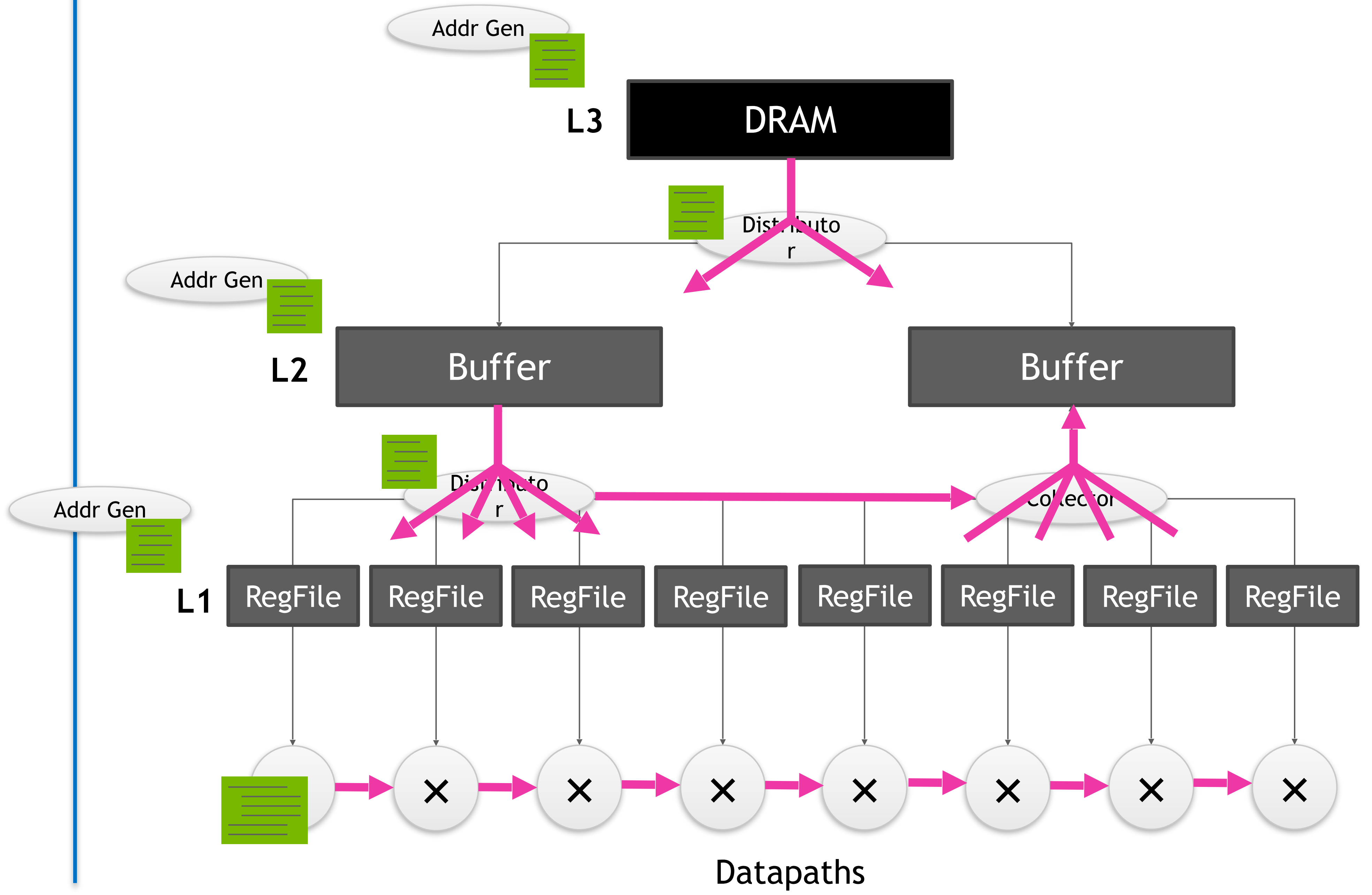
EDDO Architectures

Benefits

- Dedicated (and often statically programmed) state machines more efficient than general cores
- Perfect "prefetching"
- **Buffet** storage idiom provides fine-grain synchronization and efficient storage, or scratchpads + Send/Recv synchronization
- Hardware mechanisms for reuse

*Explicit Decoupled Data Orchestration (**EDDO**)*

e.g., IBM Rapid AI, NVIDIA Simba, NVDLA, Eyeriss, etc.

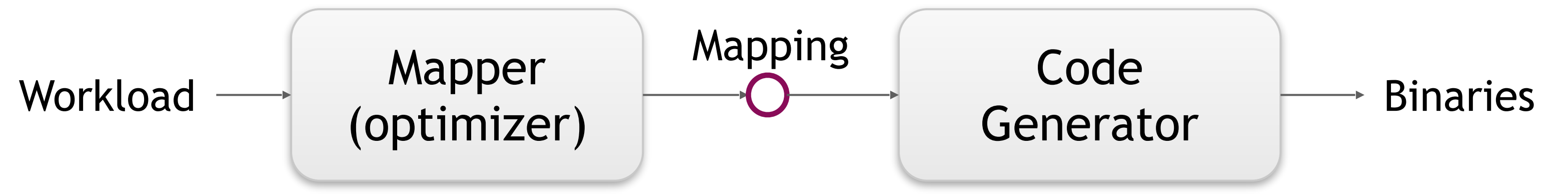


Pellauer et. al., *"Buffets: An Efficient and Composable Storage Idiom for Explicit Decoupled Data Orchestration"*, ASPLOS 2019

EDDO Architectures

Challenges

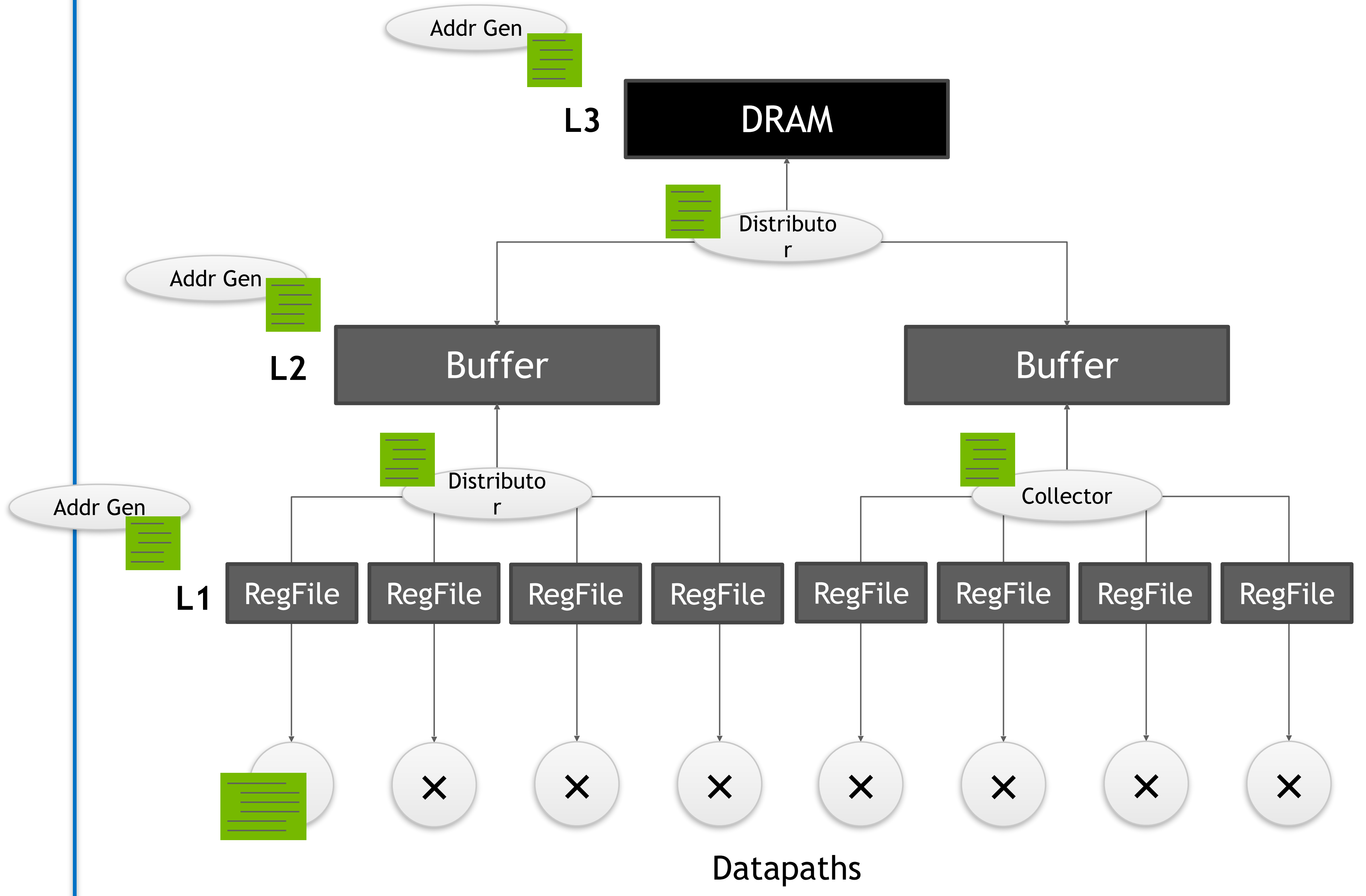
1. **No single binary:** Collection of distinct binaries that program distributed state machines working together to execute algorithm
   - E.g., CNN layer on EDDO arch → ~250 distinct state machines.
2. **Reuse optimization** is critical for efficiency
   - E.g., CNN layer on EDDO arch → 480,000 mappings, 11x spread in energy efficiency, 1 optimal mapping
   - Need an optimizer or *mapper*



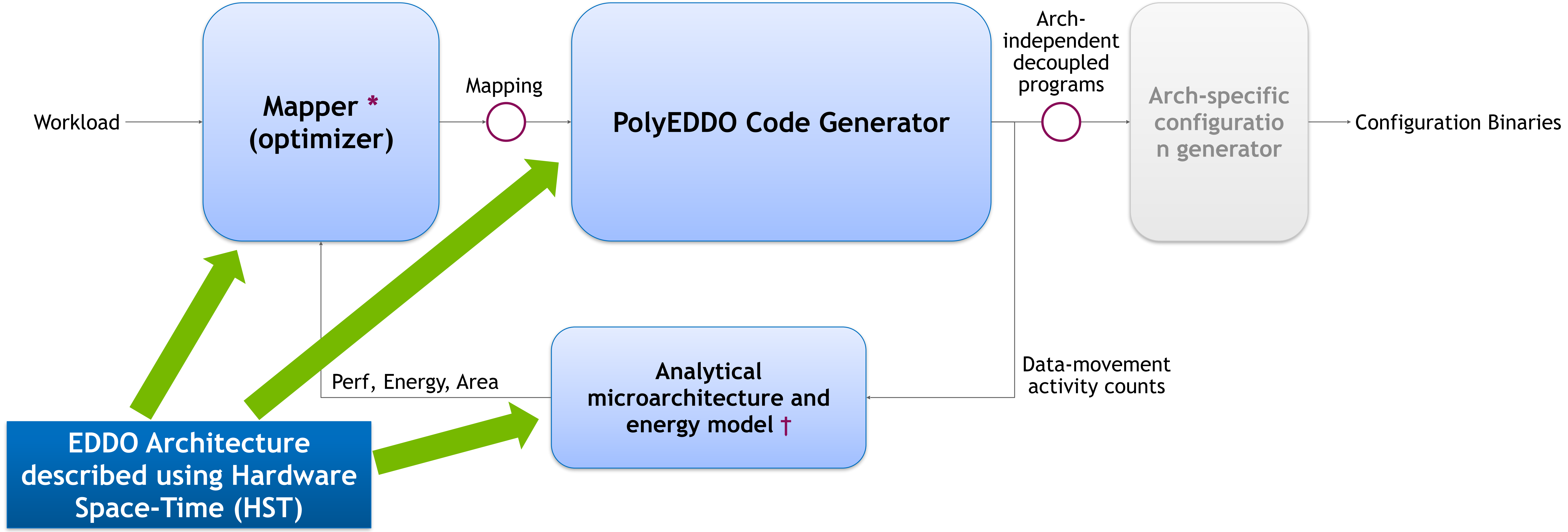
3. Variety of EDDO architectures, constantly evolving
   - Need an abstraction that Mapper and Code Generator will target

*Explicit Decoupled Data Orchestration (**EDDO**)*

e.g., IBM Rapid AI, NVIDIA Simba, NVDLA, Eyeriss, etc.

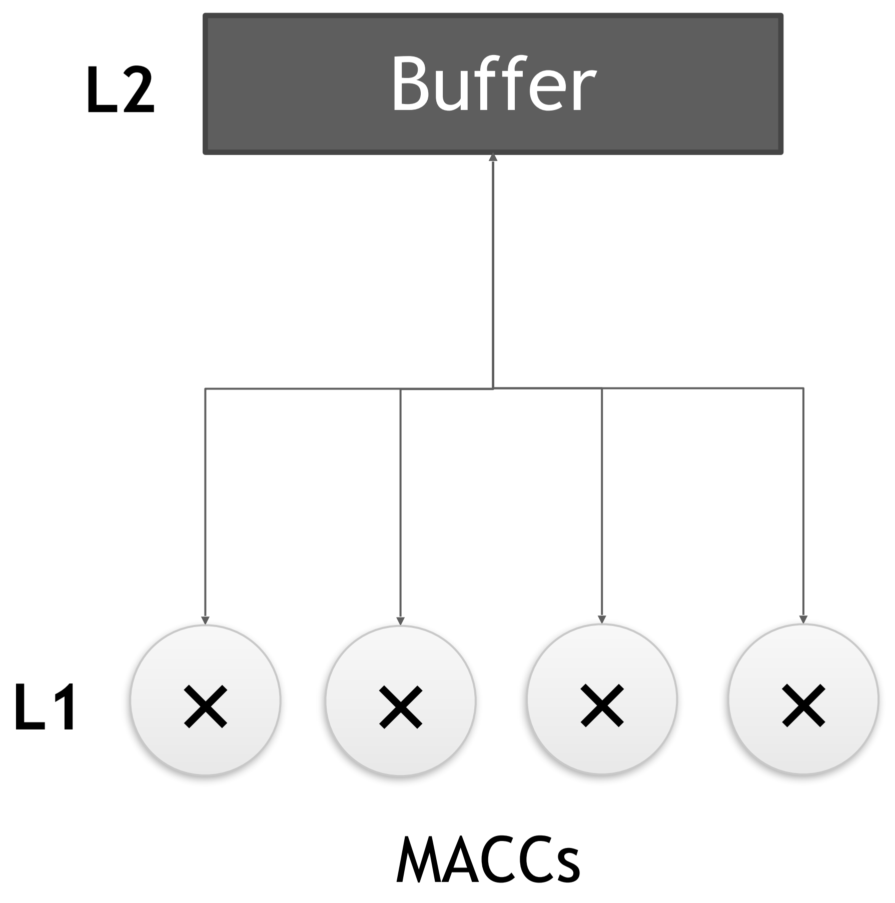


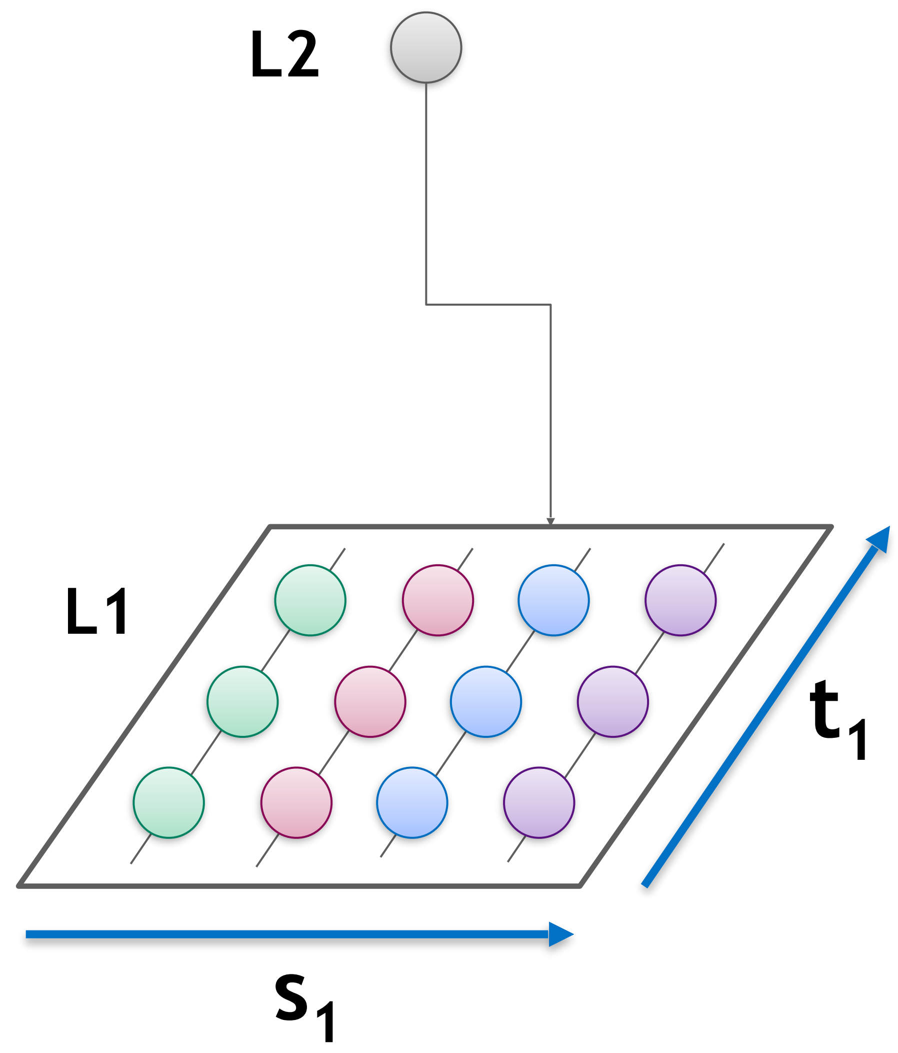
Overall Compilation Flow


Workload
Mapper *
(optimizer)
Mapping
PolyEDDO Code Generator
Arch-independent decoupled programs
Arch-specific configuratio n generator
Configuration Binaries
Perf, Energy, Area
Analytical microarchitecture and energy model †
Data-movement activity counts
EDDO Architecture described using Hardware Space-Time (HST)

*† Parashar et. al., “*Timeloop: Timeloop: A Systematic Approach to DNN Accelerator Evaluation*”, ISPASS 2019
† Wu et. al., “*Accelergy: An Architecture-Level Energy Estimation Methodology for Accelerator Designs*”, ICCAD 2019

Example1 — Symbolic Hardware Space-Time (SHST)





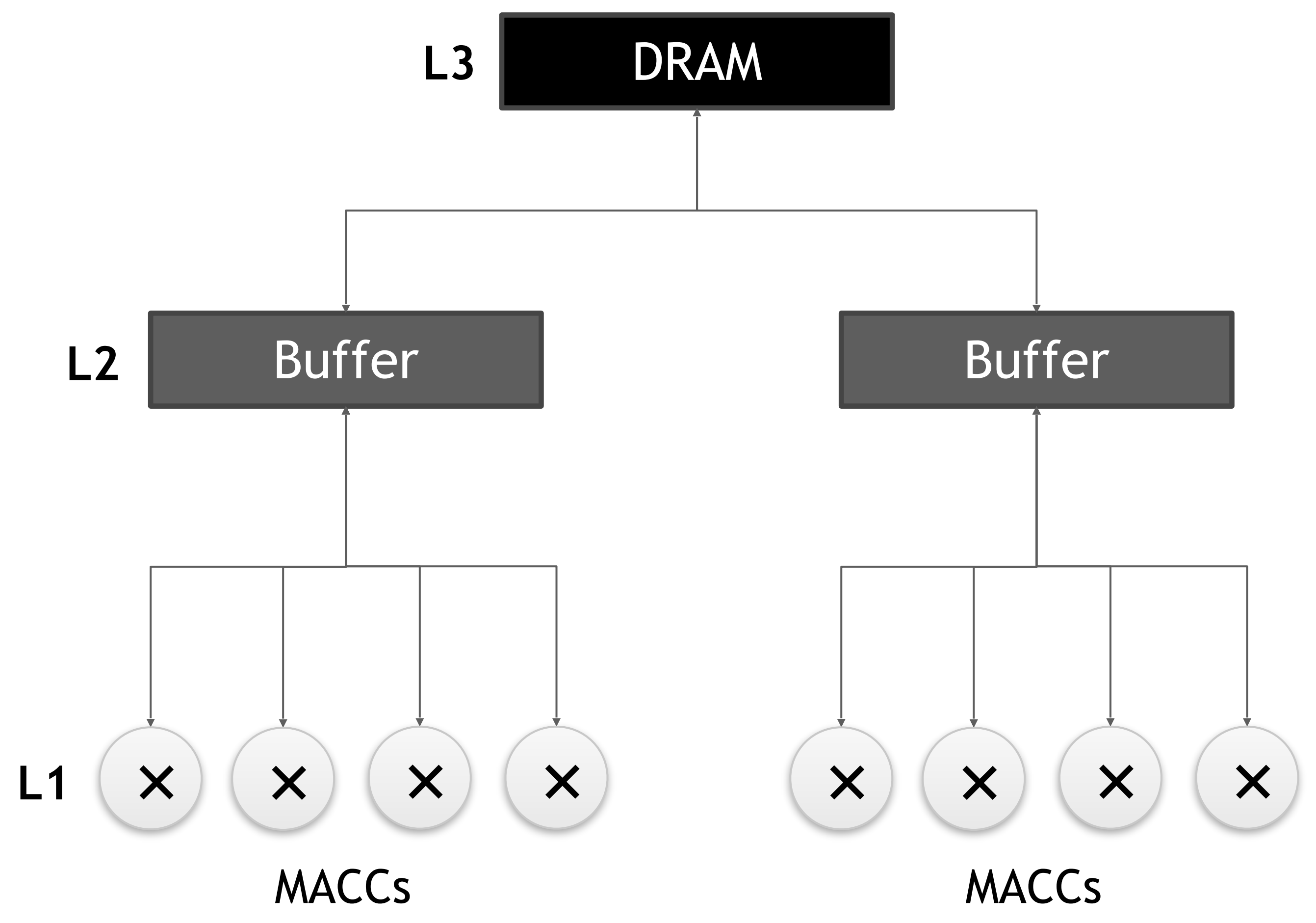
$$SpaceTime_2\,[s_2,\, t_2] \rightarrow SpaceTime_1\,[s_1,\, t_1] :$$

$$s_2 = 0 \;\&\; t_2 = 0 \;\&$$
$$0 \le s_1 < 4 \;\&\; 0 \le t_1 < 3$$

Single L2, 4 L1s, 3 time-steps

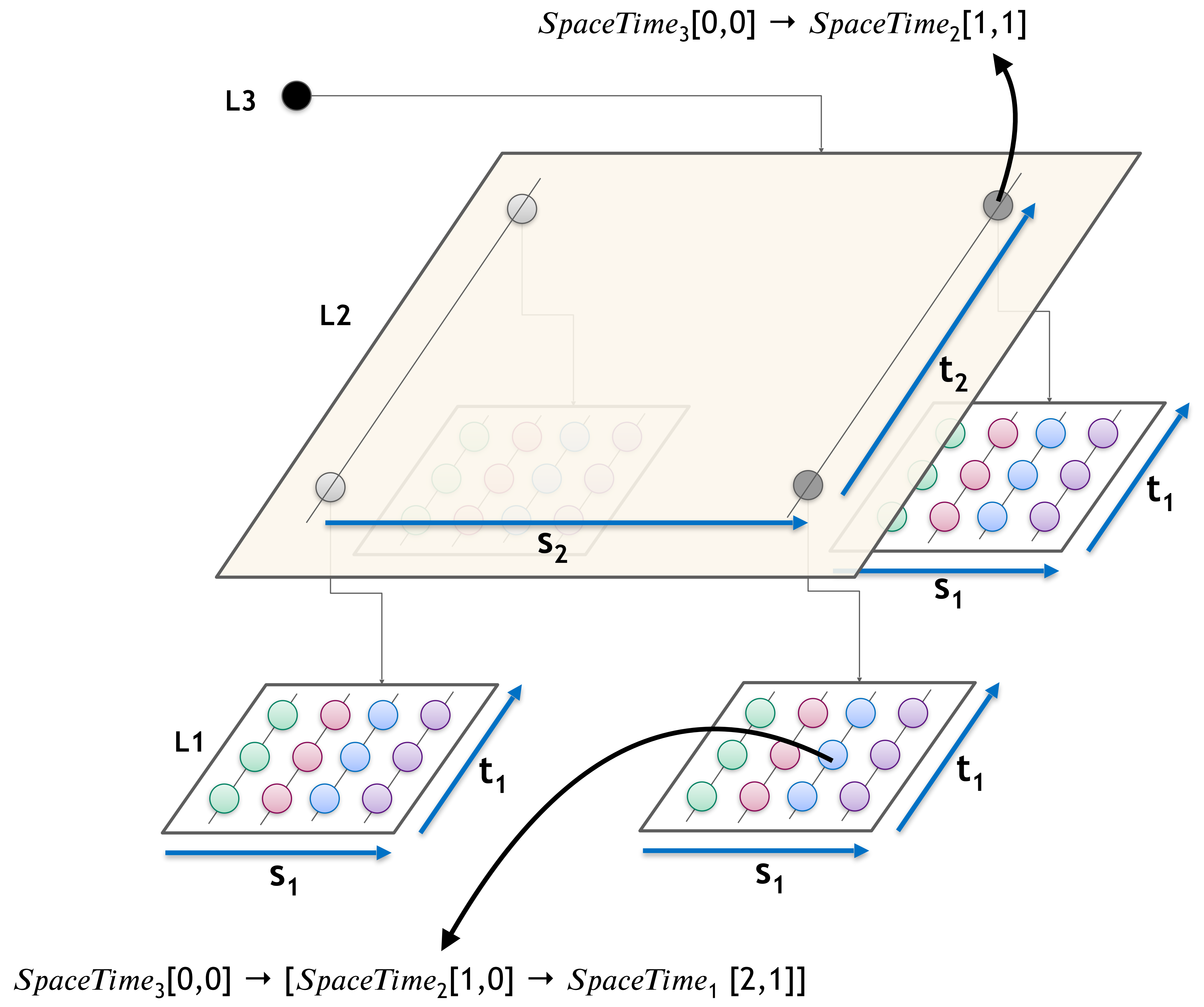
- In each step, the L2 delivers a tile of data to each L1
- Across all these L1 time steps, the resident tile in L2 does not change. In effect, **time is stagnant for L2**

Example2 — Symbolic Hardware Space-Time (SHST)

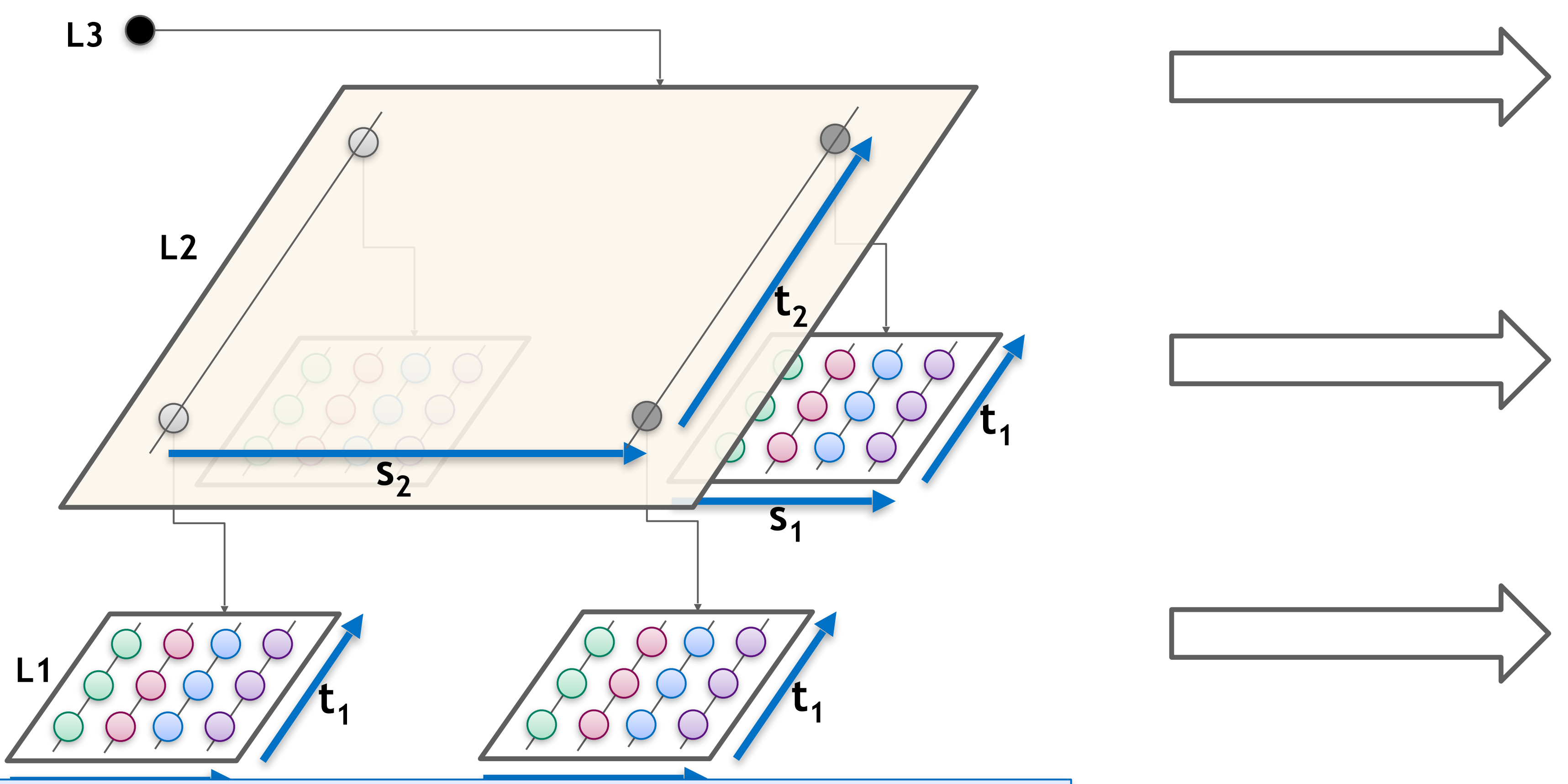


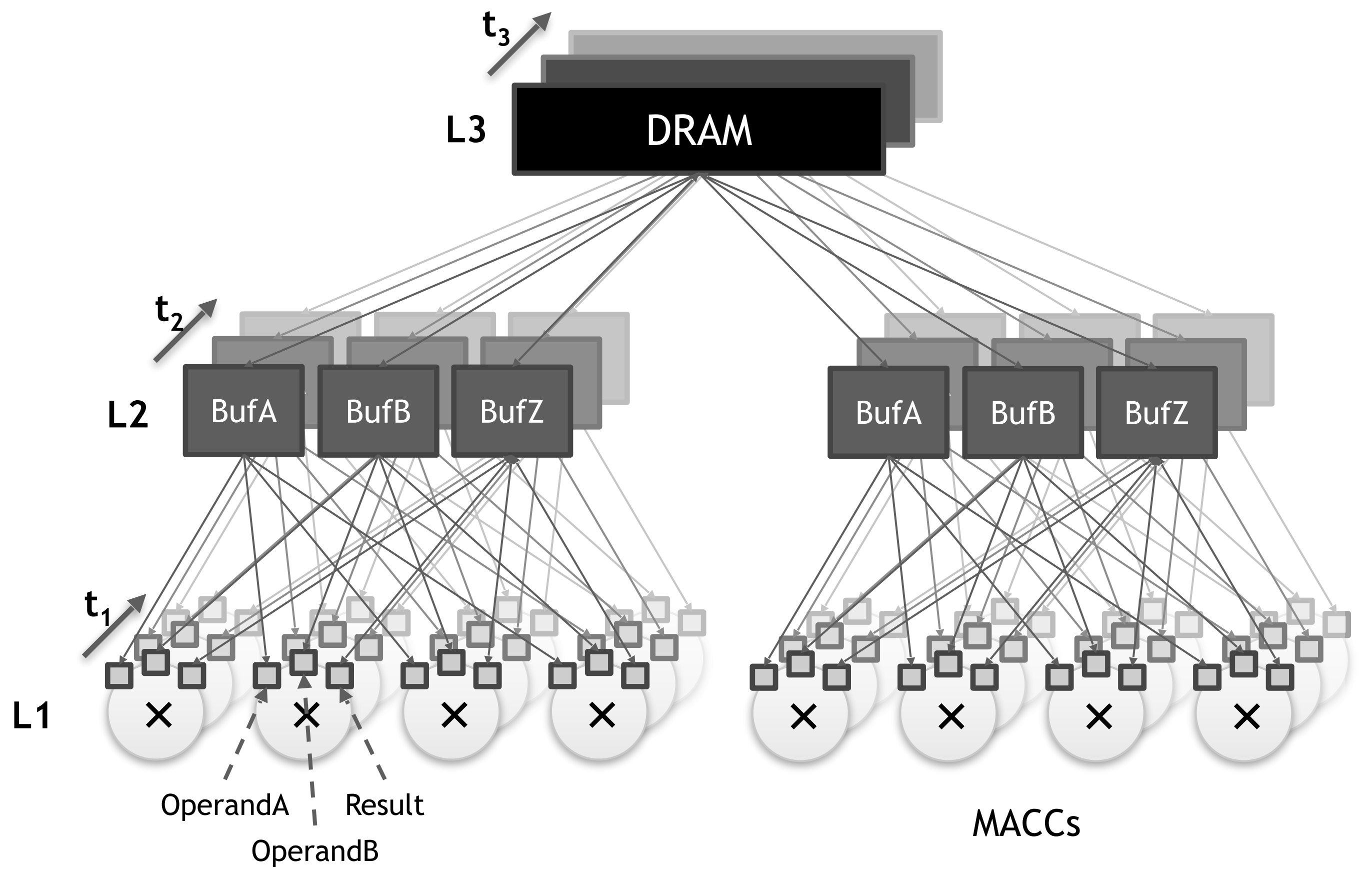
$SpaceTime_3\,[s_3, t_3] \rightarrow [SpaceTime_2\,[s_2, t_2] \rightarrow SpaceTime_1\,[s_1, t_1]]$:

$s_3 = 0$	$t_3 = 0$
$0 \le s_2 < 2$	$0 \le t_2 < 2$
$0 \le s_1 < 4$	$0 \le t_1 < 3$


$SpaceTime_3[0,0] \rightarrow SpaceTime_2[1,1]$
L3
L2
t_2
s_2
t_1
s_1
L1
t_1
s_1
t_1
s_1
$SpaceTime_3[0,0] \rightarrow [SpaceTime_2[1,0] \rightarrow SpaceTime_1\,[2,1]]$

Example3 — Partitioned Buffers





Workload mappings target SHST

SHST $SpaceTime_3\ [s_3, t_3] \rightarrow [SpaceTime_2\ [s_2, t_2] \rightarrow SpaceTime_1\ [s_1, t_1]]$

PHST

HST

Θ^{HST}(DRAM)	=	$SpaceTime_3\ [0, 0]$	$\rightarrow$	DRAM $[s_3, t_3]$
Θ^{HST}(BufA)	=	$SpaceTime_3\ [0, 0] \rightarrow SpaceTime_2\ [s_2, t_2]$	$\rightarrow$	BufA $[s_2, t_2]$
Θ^{HST}(BufB)	=	$SpaceTime_3\ [0, 0] \rightarrow SpaceTime_2\ [s_2, t_2]$	$\rightarrow$	BufB $[s_2, t_2]$
Θ^{HST}(BufZ)	=	$SpaceTime_3\ [0, 0] \rightarrow SpaceTime_2\ [s_2, t_2]$	$\rightarrow$	BufZ $[s_2, t_2]$
Θ^{HST}(OperandA)	=	$SpaceTime_3\ [0, 0] \rightarrow [SpaceTime_2\ [s_2, t_2] \rightarrow SpaceTime_1\ [s_1, t_1]]$	$\rightarrow$	OperandA $[2s_2 + s_1, t_2, t_1]$
Θ^{HST}(OperandB)	=	$SpaceTime_3\ [0, 0] \rightarrow [SpaceTime_2\ [s_2, t_2] \rightarrow SpaceTime_1\ [s_1, t_1]]$	$\rightarrow$	OperandB $[2s_2 + s_1, t_2, t_1]$
Θ^{HST}(Result)	=	$SpaceTime_3\ [0, 0] \rightarrow [SpaceTime_2\ [s_2, t_2] \rightarrow SpaceTime_1\ [s_1, t_1]]$	$\rightarrow$	Result $[2s_2 + s_1, t_2, t_1]$

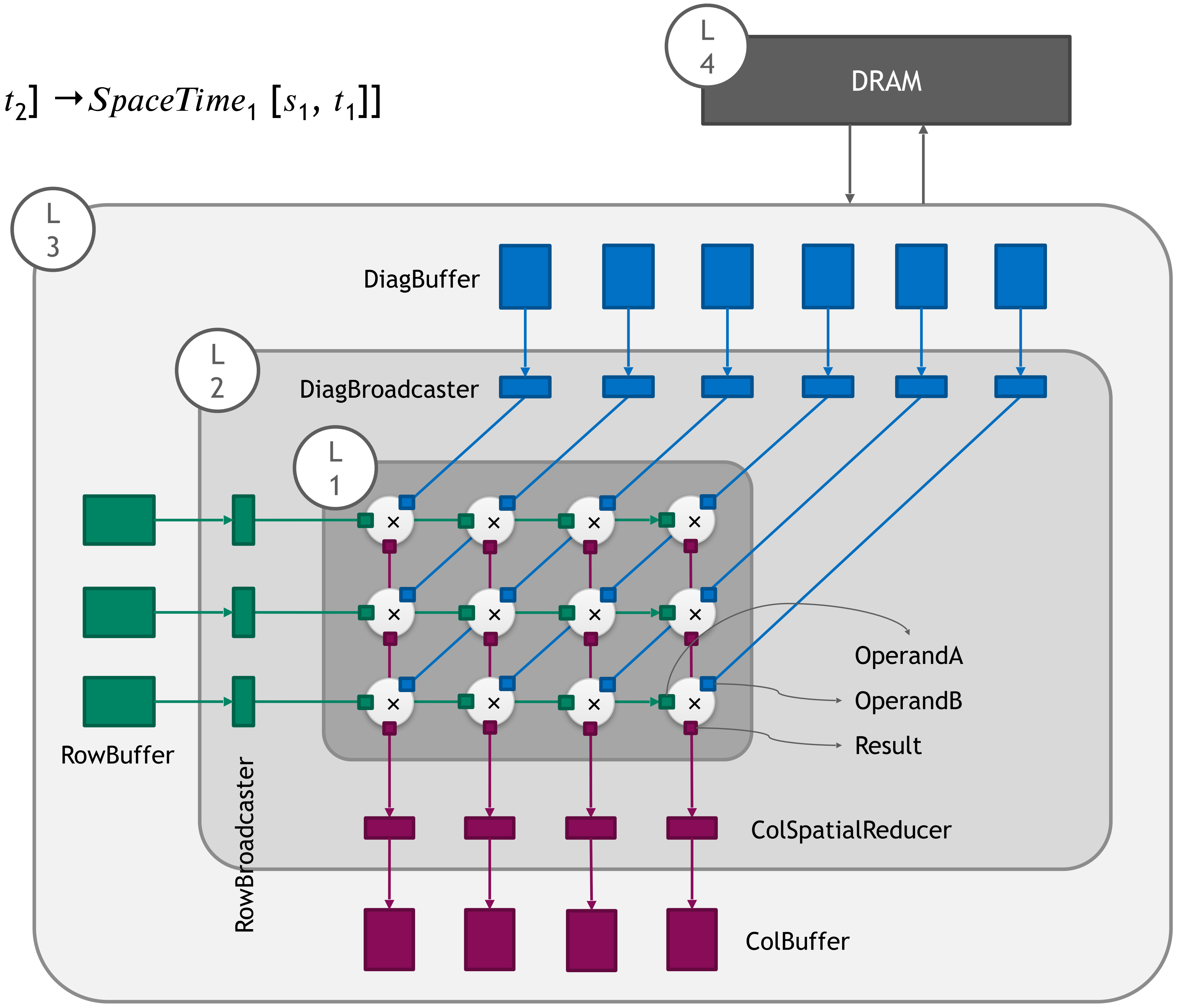
Example4 — Eyeriss-like accelerator

SHST: $SpaceTime_4\,[s_4,\,t_4] \rightarrow SpaceTime_3\,[s_3,\,t_3] \rightarrow [SpaceTime_2\,[s_2,\,t_2] \rightarrow SpaceTime_1\,[s_1,\,t_1]]$

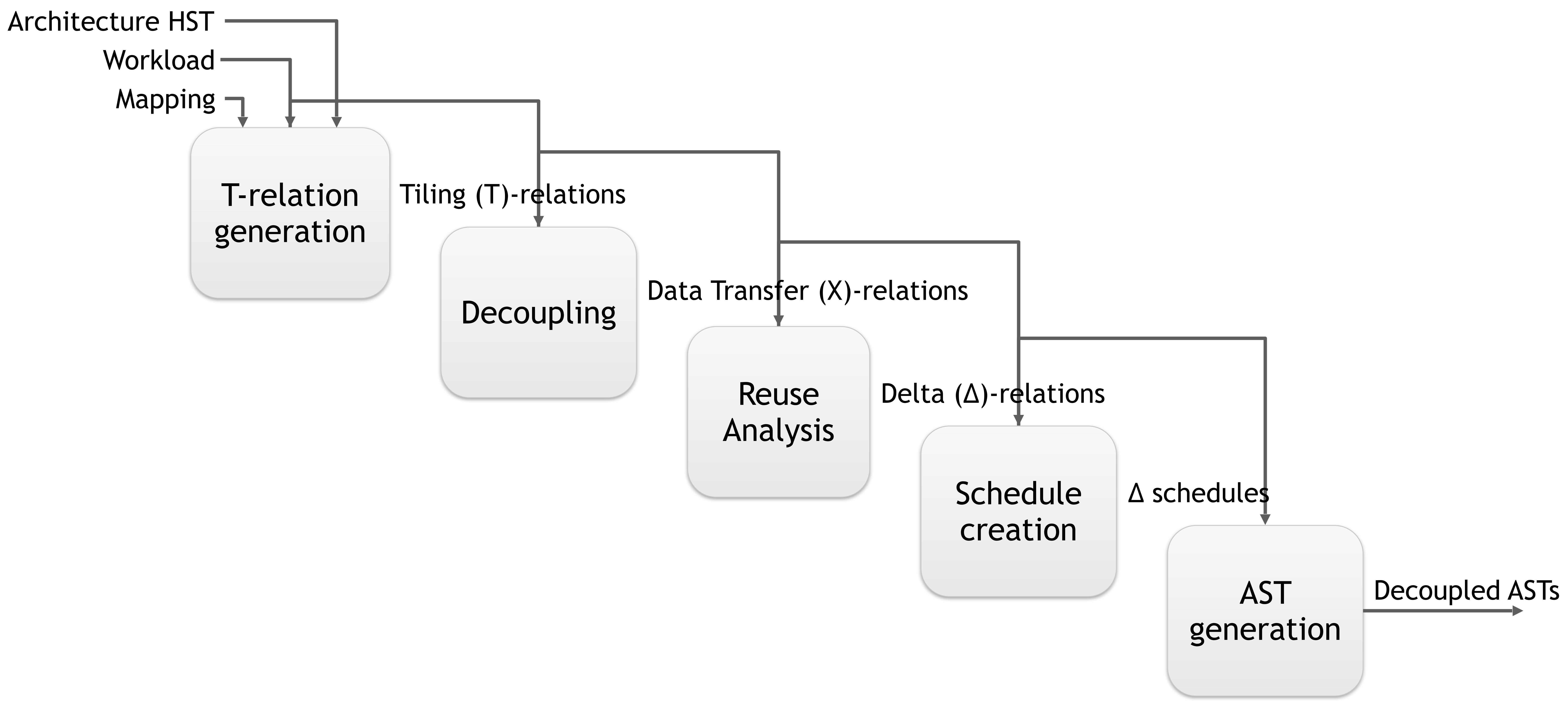
See paper for full HST

Observe how different the architecture is from CPUs and GPUs

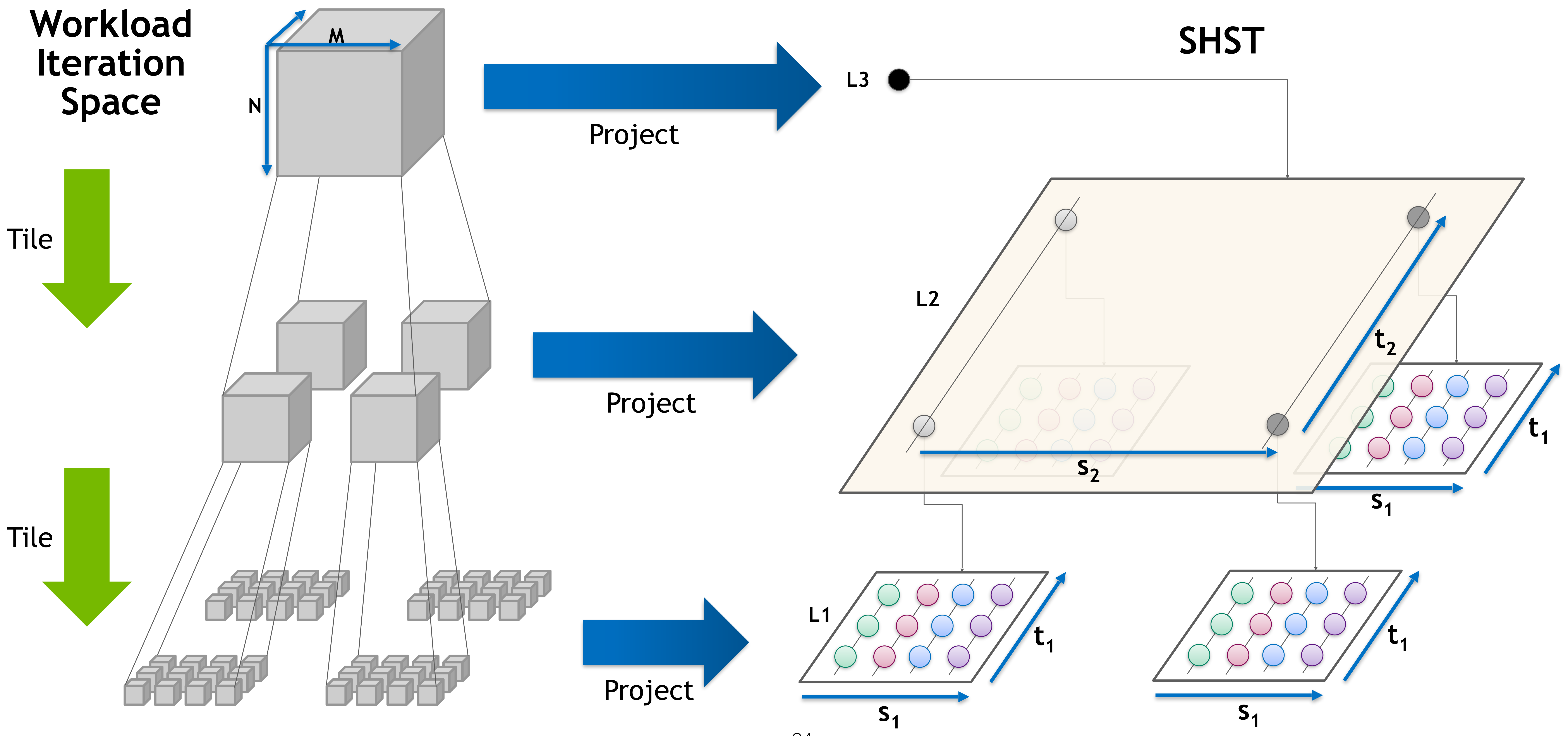
Workload mappings target SHST



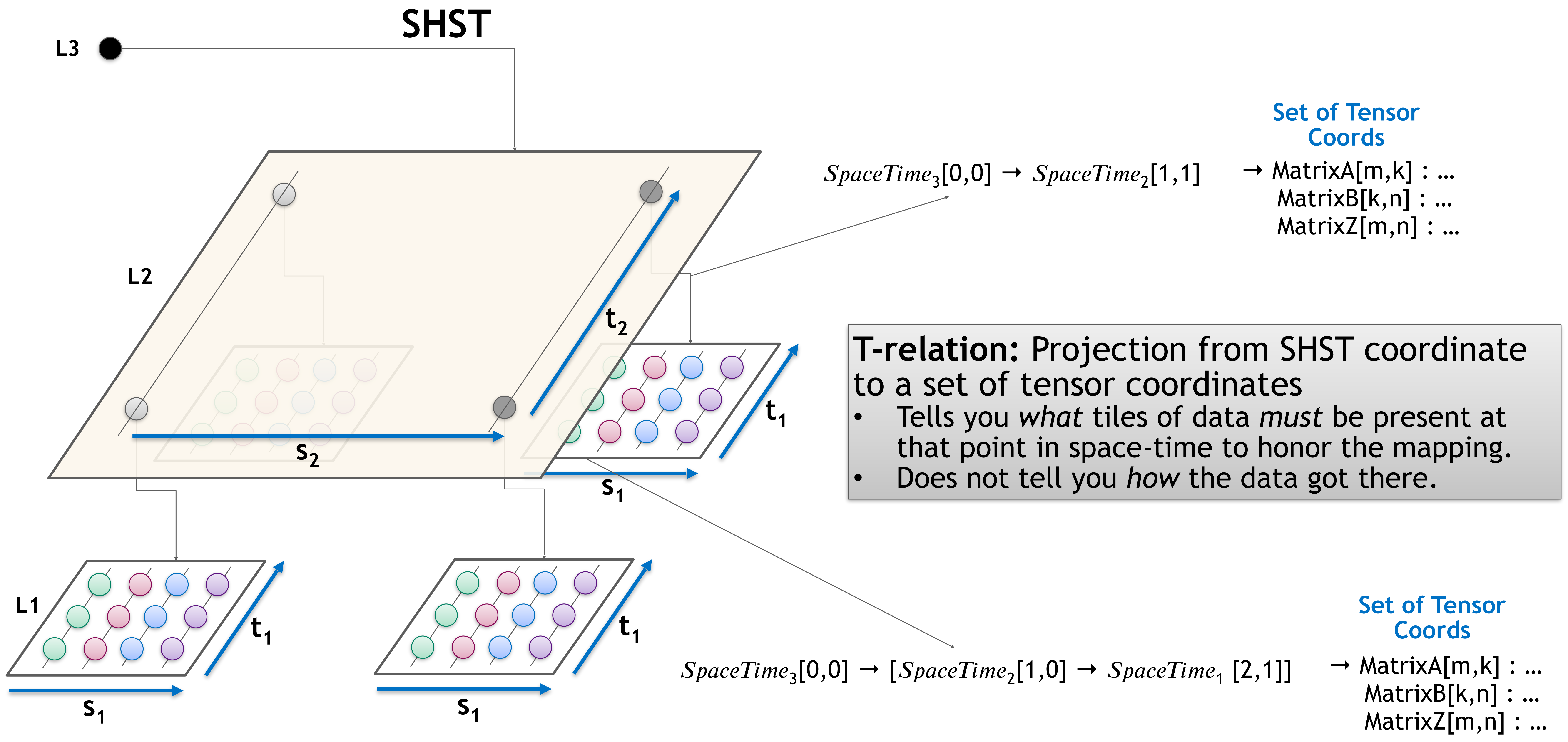
PolyEDDO Code Generator



Mapping workloads (Tensor operations)



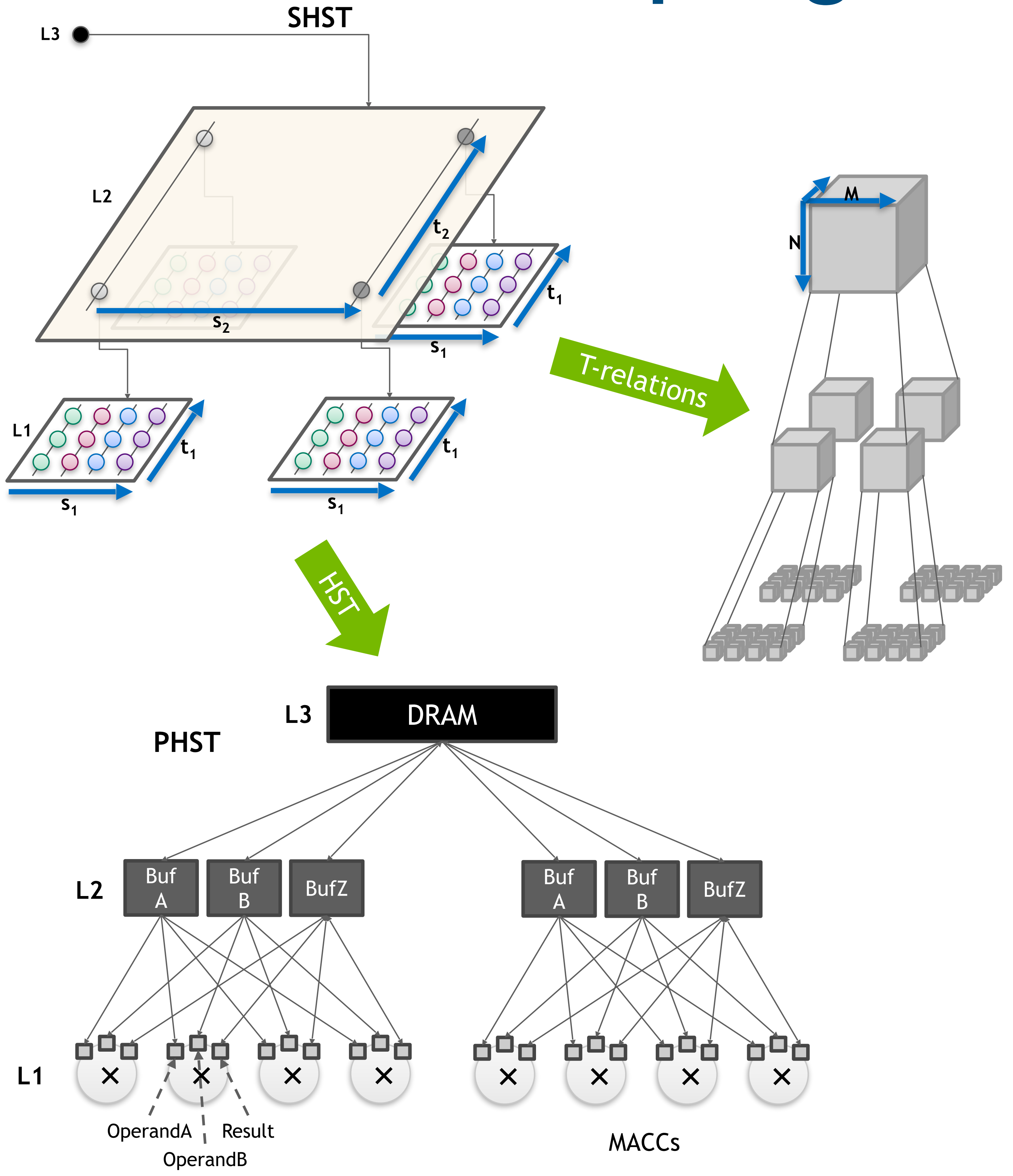
Mapping workloads (The Tiling-relation, T-relation)

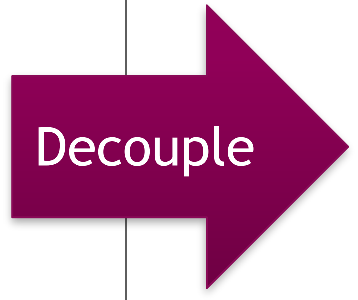


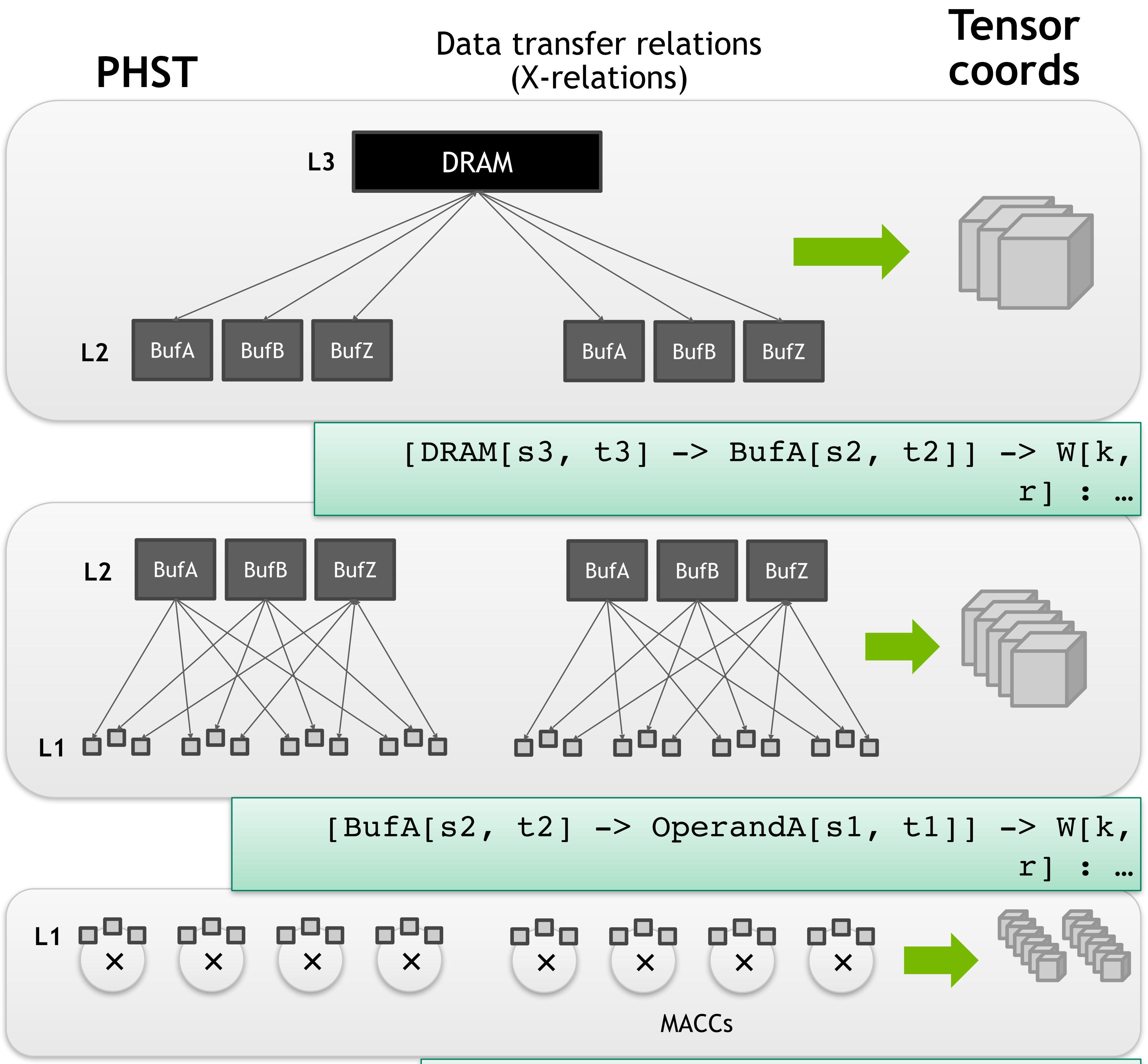
T-relation: Projection from SHST coordinate to a set of tensor coordinates

- Tells you *what* tiles of data *must* be present at that point in space-time to honor the mapping.
- Does not tell you *how* the data got there.

Decoupling — Breaking the hierarchy





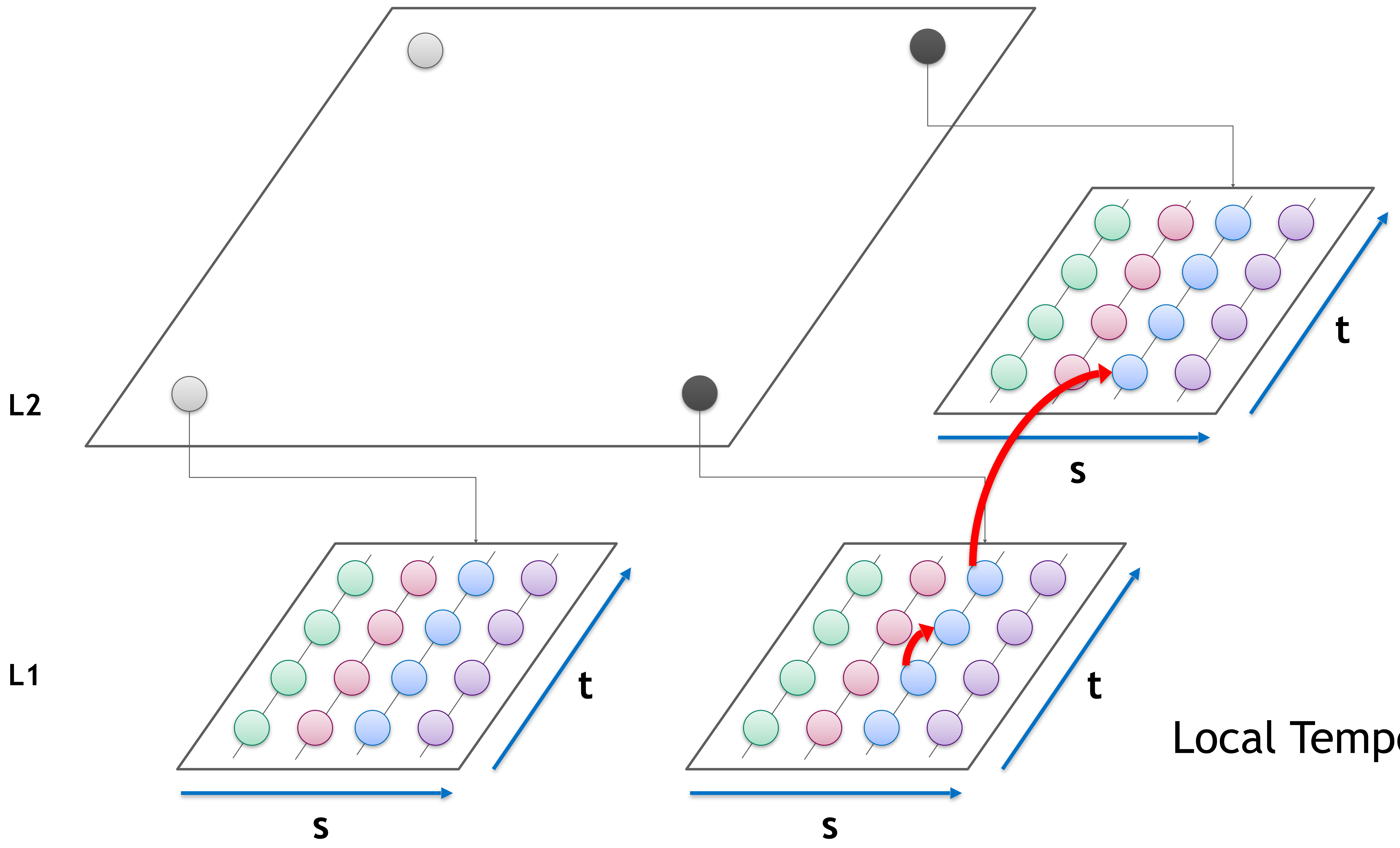


```
[DRAM[s3, t3] -> BufA[s2, t2]] -> W[k,
                                    r] : …
```

```
[BufA[s2, t2] -> OperandA[s1, t1]] -> W[k,
                                       r] : …
```

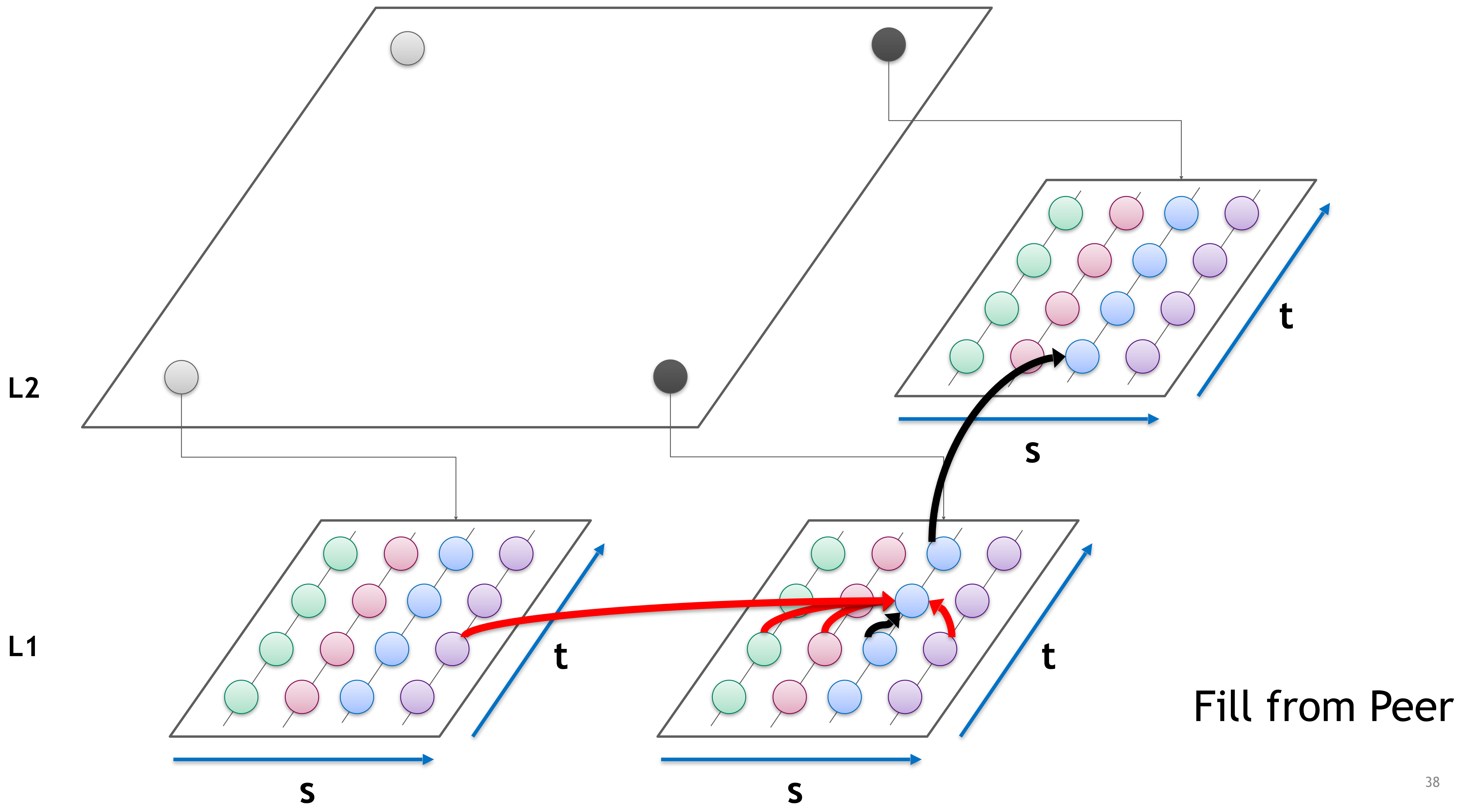
```
[MACC[s1, t1]] -> MulAcc[k, p, r] :
                                    …
```

REUSE ANALYSIS



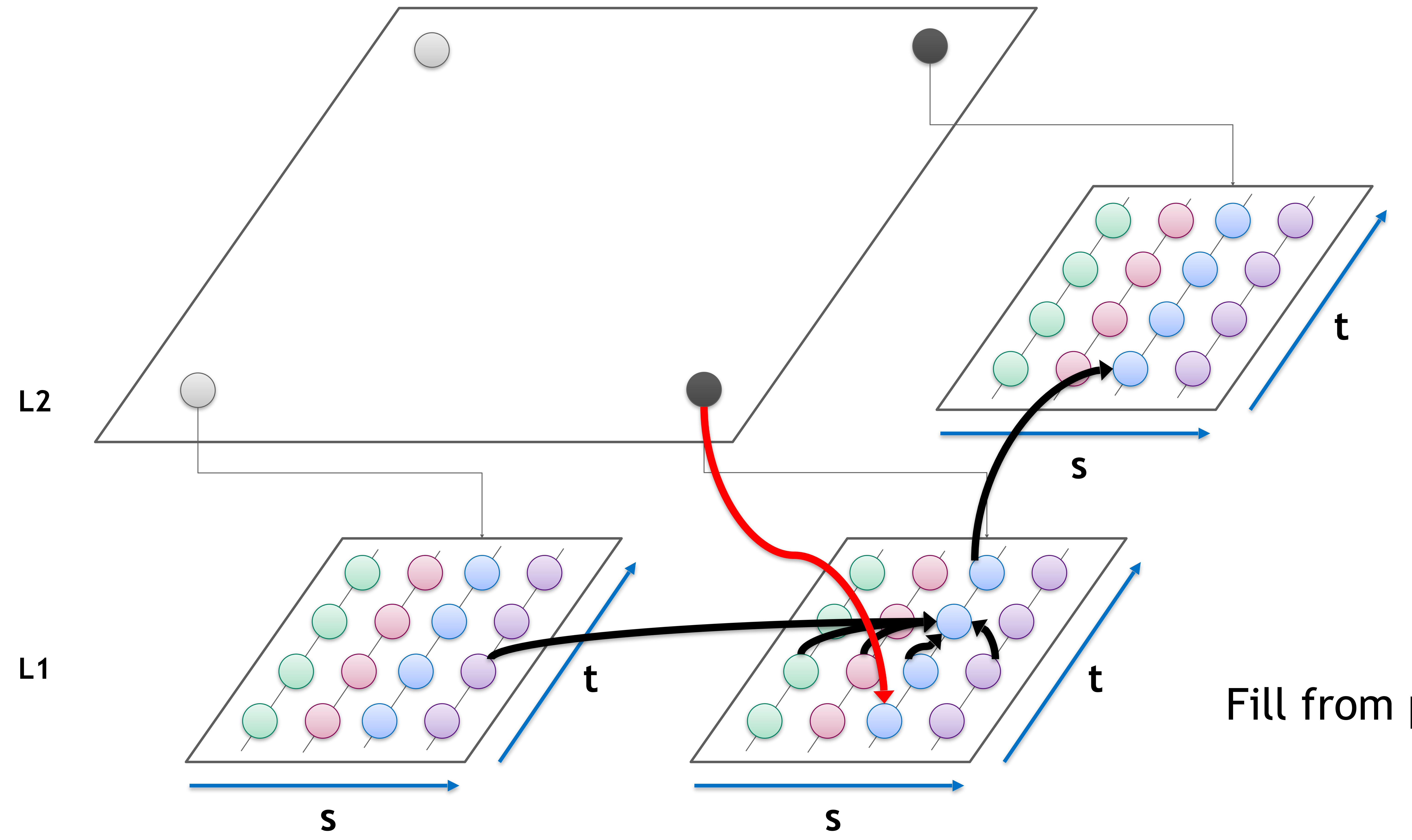
Local Temporal Reuse

REUSE ANALYSIS



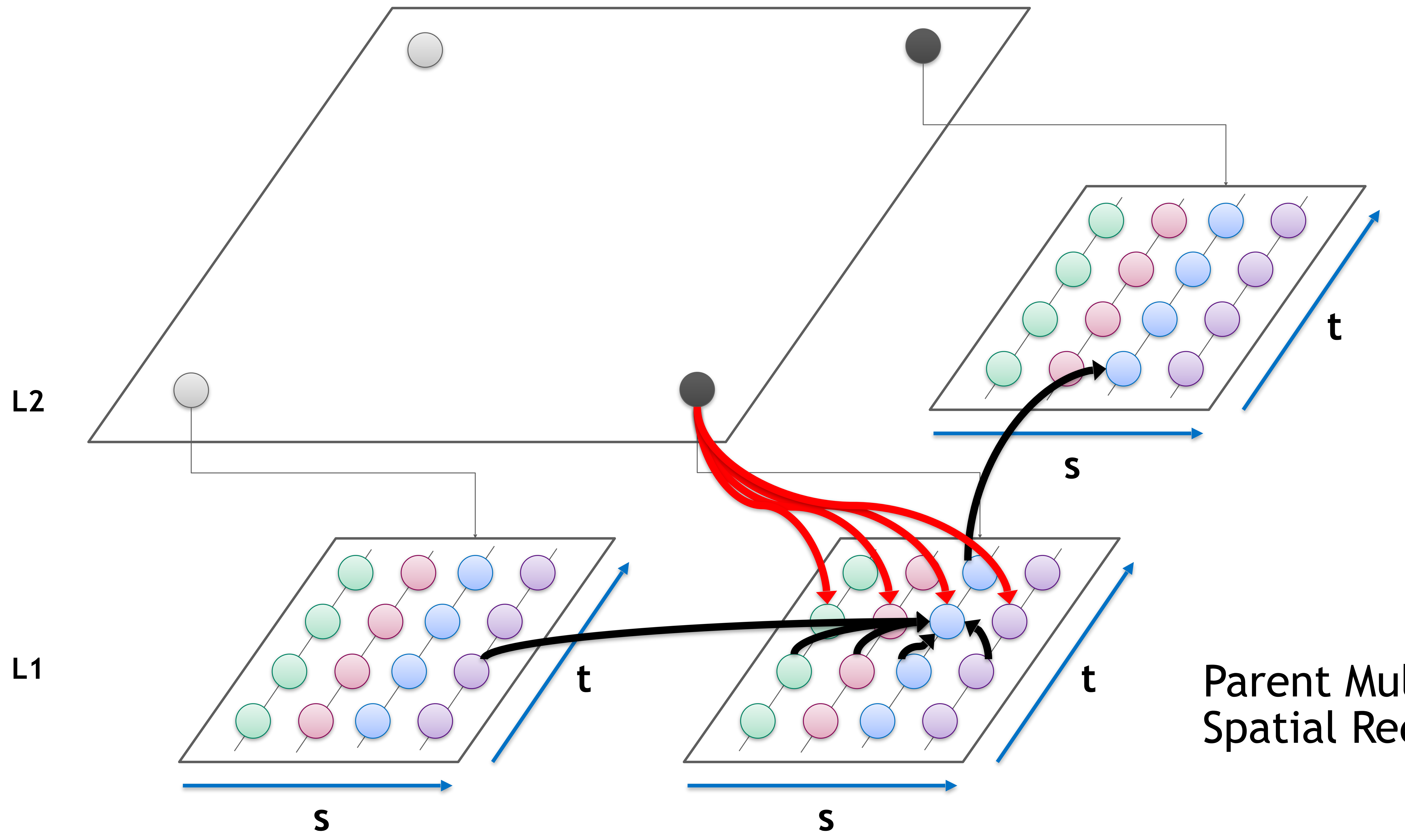
Fill from Peer

REUSE ANALYSIS



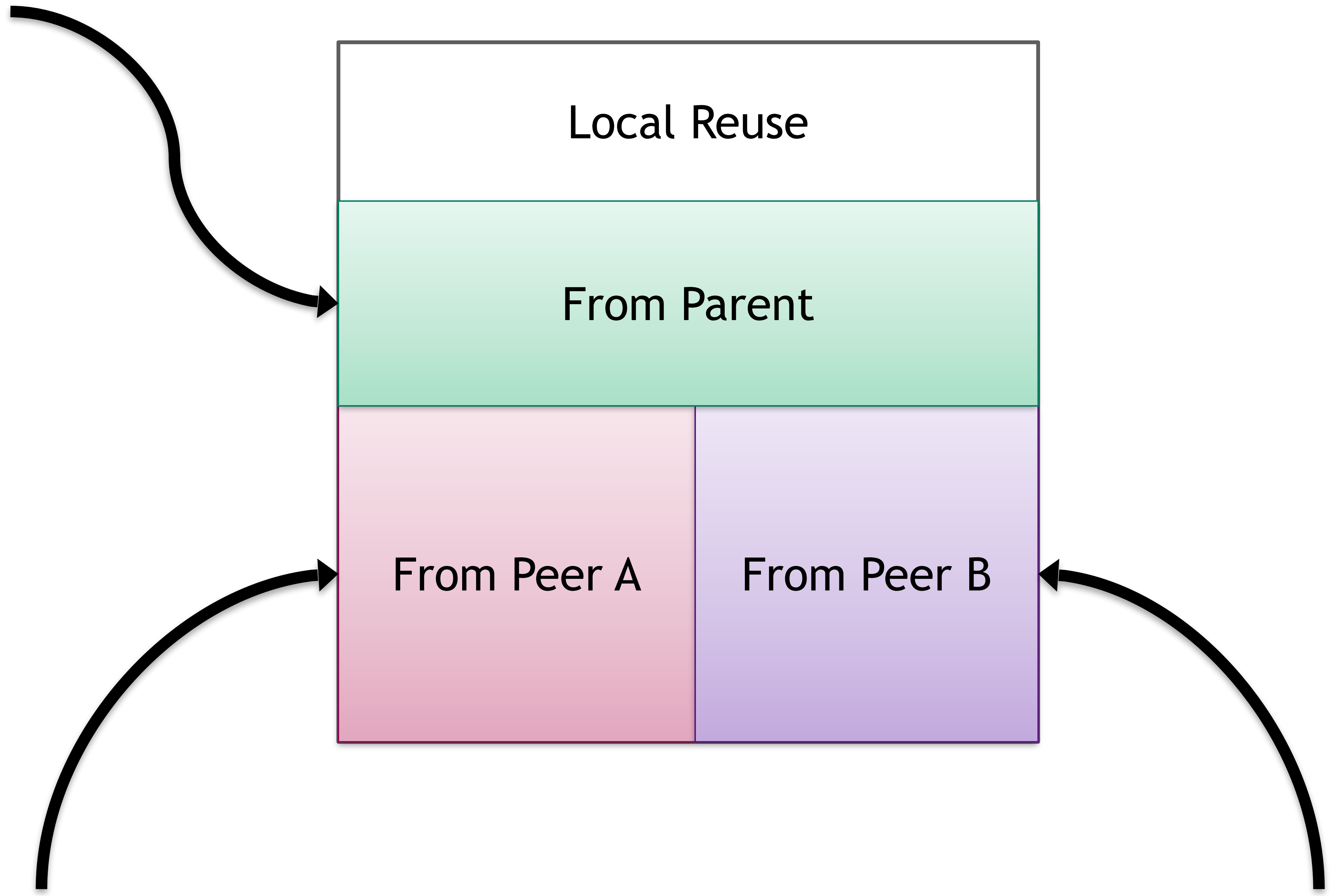
Fill from parent

REUSE ANALYSIS



Parent Multicast/
Spatial Reduction

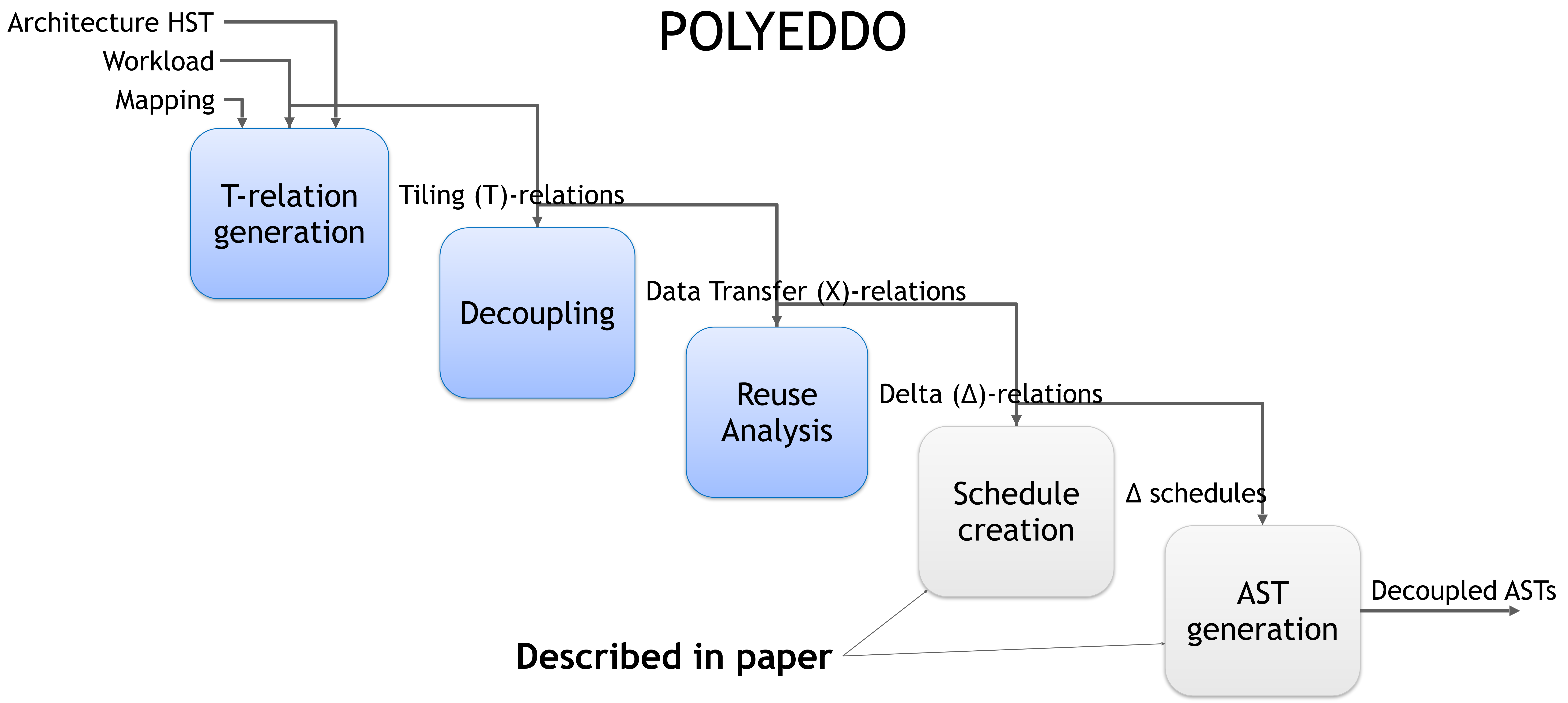
OPTIMIZATION PROBLEM (FOR A SINGLE MAPPING!)



Options:

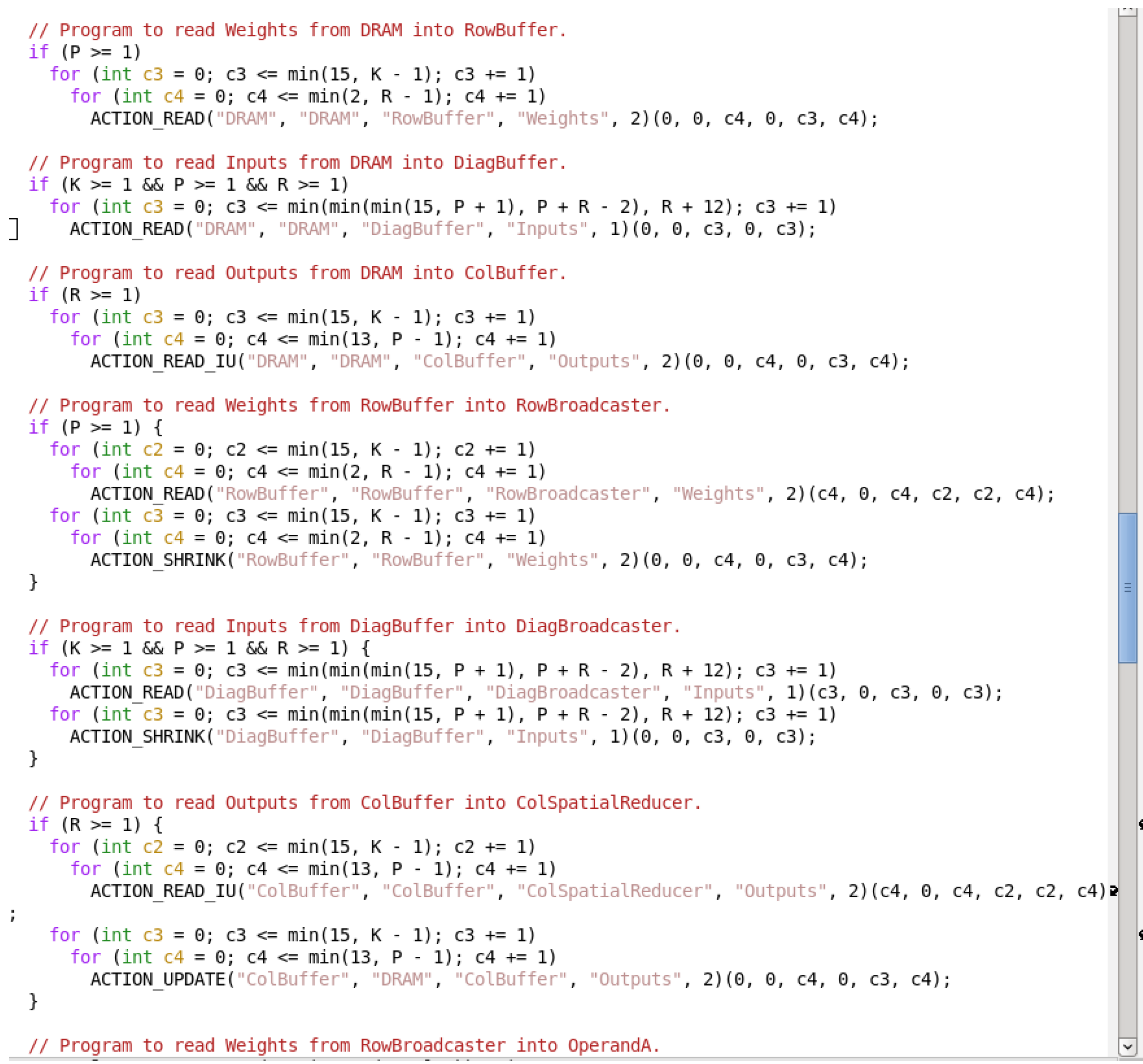
1. Enumerate all possibilities and find optimum solution
2. Use a heuristic
3. Expose choices to mapping (and thereby the mapspace)

POLYEDDO

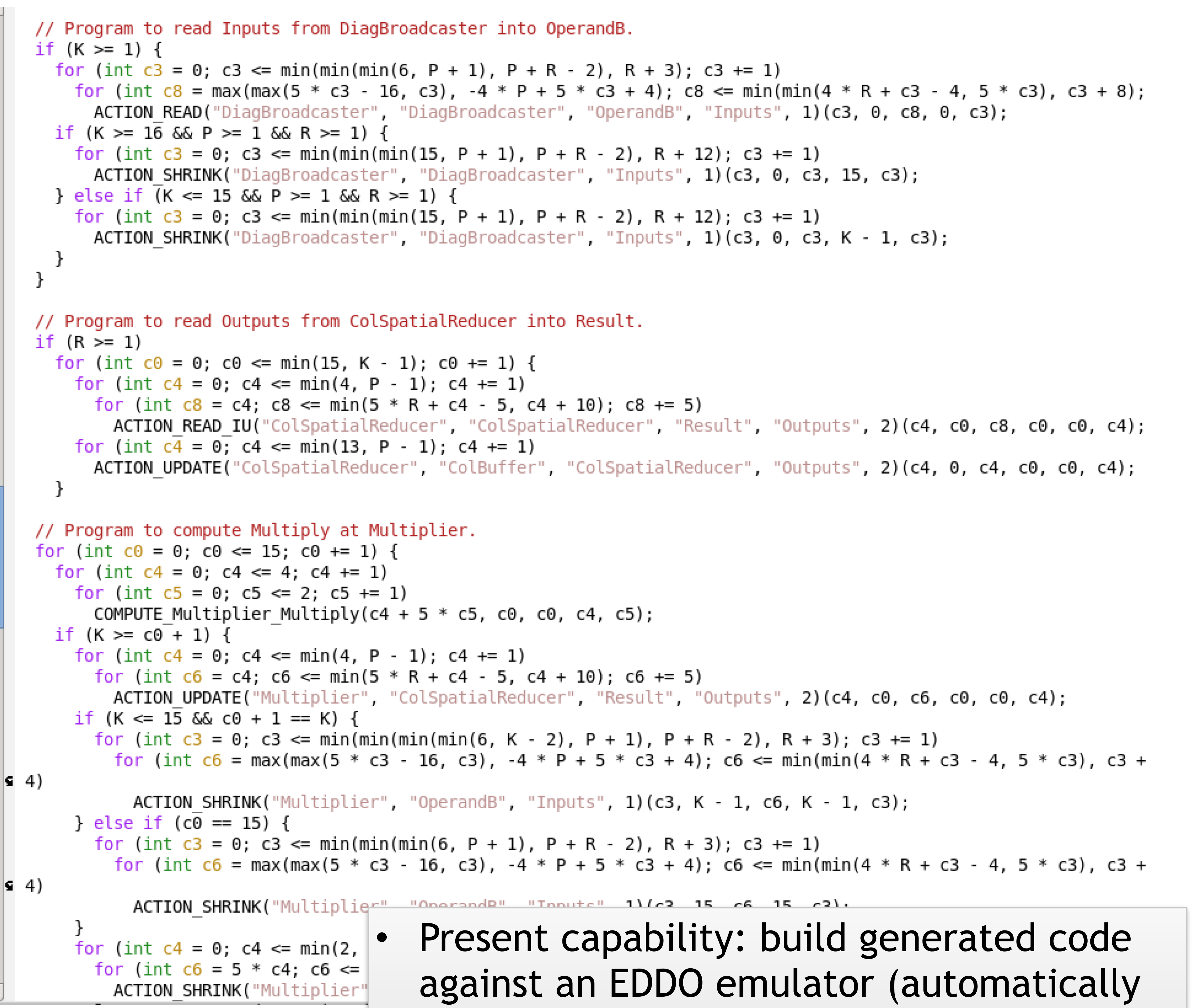


Architecture HST
Workload
Mapping
T-relation generation
Tiling (T)-relations
Decoupling
Data Transfer (X)-relations
Reuse Analysis
Delta (Δ)-relations
Schedule creation
Δ schedules
AST generation
Decoupled ASTs
Described in paper

EXAMPLE OUTPUT



```
// Program to read Weights from DRAM into RowBuffer.
if (P >= 1)
  for (int c3 = 0; c3 <= min(15, K - 1); c3 += 1)
    for (int c4 = 0; c4 <= min(2, R - 1); c4 += 1)
      ACTION_READ("DRAM", "DRAM", "RowBuffer", "Weights", 2)(0, 0, c4, 0, c3, c4);

// Program to read Inputs from DRAM into DiagBuffer.
if (K >= 1 && P >= 1 && R >= 1)
  for (int c3 = 0; c3 <= min(min(min(15, P + 1), P + R - 2), R + 12); c3 += 1)
    ACTION_READ("DRAM", "DRAM", "DiagBuffer", "Inputs", 1)(0, 0, c3, 0, c3);

// Program to read Outputs from DRAM into ColBuffer.
if (R >= 1)
  for (int c3 = 0; c3 <= min(15, K - 1); c3 += 1)
    for (int c4 = 0; c4 <= min(13, P - 1); c4 += 1)
      ACTION_READ_IU("DRAM", "DRAM", "ColBuffer", "Outputs", 2)(0, 0, c4, 0, c3, c4);

// Program to read Weights from RowBuffer into RowBroadcaster.
if (P >= 1) {
  for (int c2 = 0; c2 <= min(15, K - 1); c2 += 1)
    for (int c4 = 0; c4 <= min(2, R - 1); c4 += 1)
      ACTION_READ("RowBuffer", "RowBuffer", "RowBroadcaster", "Weights", 2)(c4, 0, c4, c2, c2, c4);
  for (int c3 = 0; c3 <= min(15, K - 1); c3 += 1)
    for (int c4 = 0; c4 <= min(2, R - 1); c4 += 1)
      ACTION_SHRINK("RowBuffer", "RowBuffer", "Weights", 2)(0, 0, c4, 0, c3, c4);
}

// Program to read Inputs from DiagBuffer into DiagBroadcaster.
if (K >= 1 && P >= 1 && R >= 1) {
  for (int c3 = 0; c3 <= min(min(min(15, P + 1), P + R - 2), R + 12); c3 += 1)
    ACTION_READ("DiagBuffer", "DiagBuffer", "DiagBroadcaster", "Inputs", 1)(c3, 0, c3, 0, c3);
  for (int c3 = 0; c3 <= min(min(min(15, P + 1), P + R - 2), R + 12); c3 += 1)
    ACTION_SHRINK("DiagBuffer", "DiagBuffer", "Inputs", 1)(0, 0, c3, 0, c3);
}

// Program to read Outputs from ColBuffer into ColSpatialReducer.
if (R >= 1) {
  for (int c2 = 0; c2 <= min(15, K - 1); c2 += 1)
    for (int c4 = 0; c4 <= min(13, P - 1); c4 += 1)
      ACTION_READ_IU("ColBuffer", "ColBuffer", "ColSpatialReducer", "Outputs", 2)(c4, 0, c4, c2, c2, c4);
  for (int c3 = 0; c3 <= min(15, K - 1); c3 += 1)
    for (int c4 = 0; c4 <= min(13, P - 1); c4 += 1)
      ACTION_UPDATE("ColBuffer", "DRAM", "ColBuffer", "Outputs", 2)(0, 0, c4, 0, c3, c4);
}

// Program to read Weights from RowBroadcaster into OperandA.
```



```
// Program to read Inputs from DiagBroadcaster into OperandB.
if (K >= 1) {
  for (int c3 = 0; c3 <= min(min(min(6, P + 1), P + R - 2), R + 3); c3 += 1)
    for (int c8 = max(max(5 * c3 - 16, c3), -4 * P + 5 * c3 + 4); c8 <= min(min(4 * R + c3 - 4, 5 * c3), c3 + 8);
      ACTION_READ("DiagBroadcaster", "DiagBroadcaster", "OperandB", "Inputs", 1)(c3, 0, c8, 0, c3);
  if (K >= 16 && P >= 1 && R >= 1) {
    for (int c3 = 0; c3 <= min(min(min(15, P + 1), P + R - 2), R + 12); c3 += 1)
      ACTION_SHRINK("DiagBroadcaster", "DiagBroadcaster", "Inputs", 1)(c3, 0, c3, 15, c3);
  } else if (K <= 15 && P >= 1 && R >= 1) {
    for (int c3 = 0; c3 <= min(min(min(15, P + 1), P + R - 2), R + 12); c3 += 1)
      ACTION_SHRINK("DiagBroadcaster", "DiagBroadcaster", "Inputs", 1)(c3, 0, c3, K - 1, c3);
  }
}

// Program to read Outputs from ColSpatialReducer into Result.
if (R >= 1)
  for (int c0 = 0; c0 <= min(15, K - 1); c0 += 1) {
    for (int c4 = 0; c4 <= min(4, P - 1); c4 += 1)
      for (int c8 = c4; c8 <= min(5 * R + c4 - 5, c4 + 10); c8 += 5)
        ACTION_READ_IU("ColSpatialReducer", "ColSpatialReducer", "Result", "Outputs", 2)(c4, c0, c8, c0, c0, c4);
    for (int c4 = 0; c4 <= min(13, P - 1); c4 += 1)
      ACTION_UPDATE("ColSpatialReducer", "ColBuffer", "ColSpatialReducer", "Outputs", 2)(c4, 0, c4, c0, c0, c4);
  }

// Program to compute Multiply at Multiplier.
for (int c0 = 0; c0 <= 15; c0 += 1) {
  for (int c4 = 0; c4 <= 4; c4 += 1)
    for (int c5 = 0; c5 <= 2; c5 += 1)
      COMPUTE_Multiplier_Multiply(c4 + 5 * c5, c0, c0, c4, c5);
  if (K >= c0 + 1) {
    for (int c4 = 0; c4 <= min(4, P - 1); c4 += 1)
      for (int c6 = c4; c6 <= min(5 * R + c4 - 5, c4 + 10); c6 += 5)
        ACTION_UPDATE("Multiplier", "ColSpatialReducer", "Result", "Outputs", 2)(c4, c0, c6, c0, c0, c4);
    if (K <= 15 && c0 + 1 == K) {
      for (int c3 = 0; c3 <= min(min(min(min(6, K - 2), P + 1), P + R - 2), R + 3); c3 += 1)
        for (int c6 = max(max(5 * c3 - 16, c3), -4 * P + 5 * c3 + 4); c6 <= min(min(4 * R + c3 - 4, 5 * c3), c3 + 4)
          ACTION_SHRINK("Multiplier", "OperandB", "Inputs", 1)(c3, K - 1, c6, K - 1, c3);
    } else if (c0 == 15) {
      for (int c3 = 0; c3 <= min(min(min(6, P + 1), P + R - 2), R + 3); c3 += 1)
        for (int c6 = max(max(5 * c3 - 16, c3), -4 * P + 5 * c3 + 4); c6 <= min(min(4 * R + c3 - 4, 5 * c3), c3 + 4)
          ACTION_SHRINK("Multiplier", "OperandB", "Inputs", 1)(c3, 15, c6, 15, c3);
    }
    for (int c4 = 0; c4 <= min(2,
      for (int c6 = 5 * c4; c6 <=
        ACTION_SHRINK("Multiplier"
```

- Present capability: build generated code against an EDDO emulator (automatically configured from the PHST)

Summary and Questions?

- ***Summary***
  - HST (Hardware Space-Time) – an abstraction for EDDO architectures represented using the Polyhedral Model
  - PolyEDDO (WIP) – an analysis and code-generation flow based on HST
- ***Research questions***
  - How do we think about mapping imperfectly-nested loops to generic EDDO architectures?
  - How do we capture sparsity extensions of the accelerators?

Acknowledgments

- **Ph.D. Advisors:**
  - Vivek Sarkar (advisor), and Jun Shirako (co-advisor)

- **Collaborators**
  - Albert Cohen, Martin Kong, Tushar Krishna, Hyoukjun Kwon, John Mellor-Crummey, Karthik Murthy, Stephen Neuendorffer, Angshuman Parashar, Micheal Pellauer, Kees Vissers, and others

- **Other mentors**
  - Kesav Nori, Uday Bondhugula, Milind Chabbi, Shams Imam, Deepak Majeti, Rishi Surendran, and others

- **IBM Research, Habanero & Synergy Research Group Members**

Backup

Why do we need accelerators?

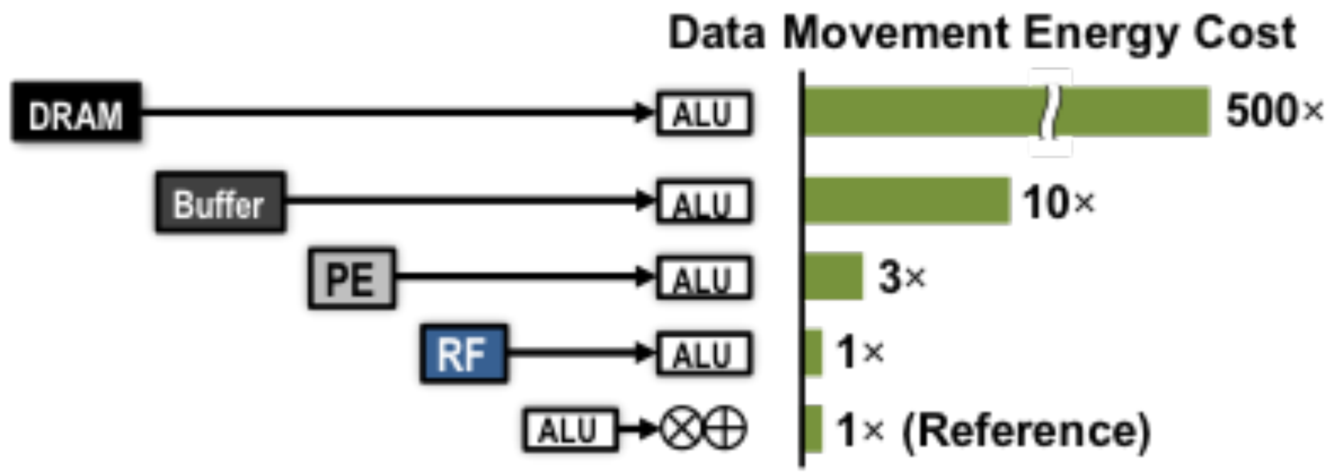
1) DNN models have tight constraints on latency, throughput, and energy consumption, esp. on edge devices

2) DNN models have trillions of computations

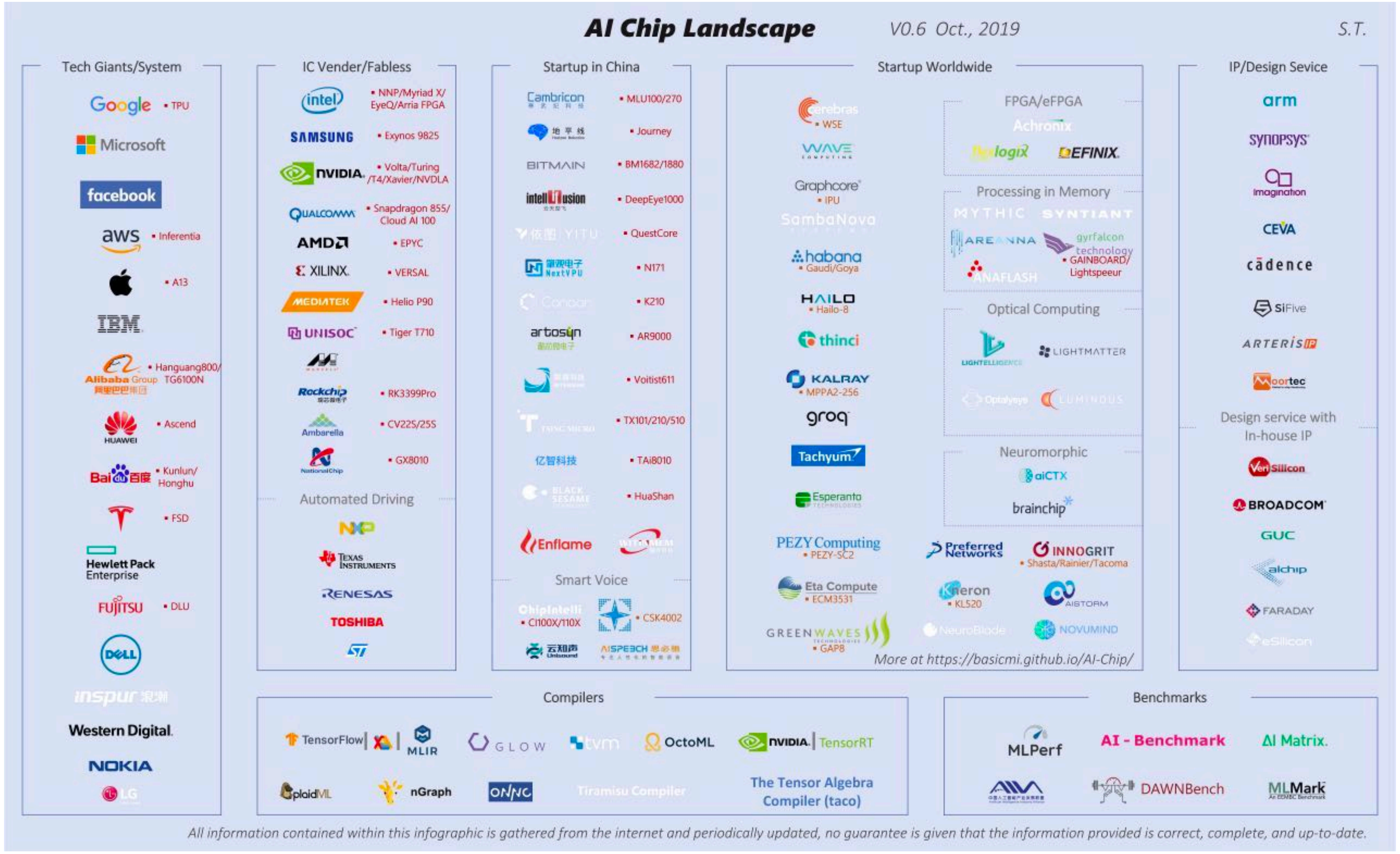
Need high throughput — Makes CPUs inefficient

3) DNN models involve heavy data movement

Need to reduce energy — Makes GPUs inefficient



Landscape of DNN Accelerators


AI Chip Landscape
V0.6 Oct., 2019
S.T.
Tech Giants/System
Google • TPU
Microsoft
facebook
aws • Inferentia
• A13
IBM
Alibaba Group • Hanguang800/ TG6100N
HUAWEI • Ascend
Baidu • Kunlun/ Honghu
• FSD
Hewlett Packard Enterprise
FUJITSU • DLU
DELL
inspur
Western Digital
NOKIA
LG
IC Vender/Fabless
intel • NNP/Myriad X/ EyeQ/Arria FPGA
SAMSUNG • Exynos 9825
NVIDIA • Volta/Turing /T4/Xavier/NVDLA
QUALCOMM • Snapdragon 855/ Cloud AI 100
AMD • EPYC
XILINX • VERSAL
MEDIATEK • Helio P90
UNISOC • Tiger T710
Rockchip • RK3399Pro
Ambarella • CV22S/25S
NationalChip • GX8010
Automated Driving
NXP
Texas Instruments
RENESAS
TOSHIBA
ST
Startup in China
Cambricon • MLU100/270
• Journey
BITMAIN • BM1682/1880
intellifusion • DeepEye1000
YITU • QuestCore
NextVPU • N171
Canaan • K210
artosyn • AR9000
• Voitist611
• TX101/210/510
亿智科技 • TAi8010
BLACK SESAME • HuaShan
Enflame
WITINMEM
Smart Voice
Chipintelli • CI100X/110X
• CSK4002
Unisound
AISPEECH
Startup Worldwide
cerebras • WSE
WAVE COMPUTING
Graphcore • IPU
SambaNova
habana • Gaudi/Goya
HAILO • Hailo-8
thinci
KALRAY • MPPA2-256
groq
Tachyum
Esperanto
PEZY Computing • PEZY-SC2
Eta Compute • ECM3531
GREENWAVES • GAP8
FPGA/eFPGA
Achronix
Flexlogix
EFINIX
Processing in Memory
MYTHIC
SYNTIANT
ARETANNA
gyrfalcon technology • GAINBOARD/ Lightspeeur
ANAFLASH
Optical Computing
LIGHTELLIGENCE
LIGHTMATTER
Optalysys
LUMINOUS
Neuromorphic
aiCTX
brainchip
Preferred Networks
INNOGRIT • Shasta/Rainier/Tacoma
Kneron • KL520
AISTORM
NeuroBlade
NOVUMIND
More at https://basicmi.github.io/AI-Chip/
IP/Design Sevice
arm
SYNOPSYS
imagination
CEVA
cādence
SiFive
ARTERIS IP
moortec
Design service with In-house IP
VeriSilicon
BROADCOM
GUC
alchip
FARADAY
eSilicon
Compilers
TensorFlow
MLIR
GLOW
tvm
OctoML
NVIDIA TensorRT
PlaidML
nGraph
ONNC
Tiramisu Compiler
The Tensor Algebra Compiler (taco)
Benchmarks
MLPerf
AI - Benchmark
AI Matrix
AIIA
DAWNBench
MLMark
All information contained within this infographic is gathered from the internet and periodically updated, no guarantee is given that the information provided is correct, complete, and up-to-date.