PolyEDDO
vs
Unified Buffer

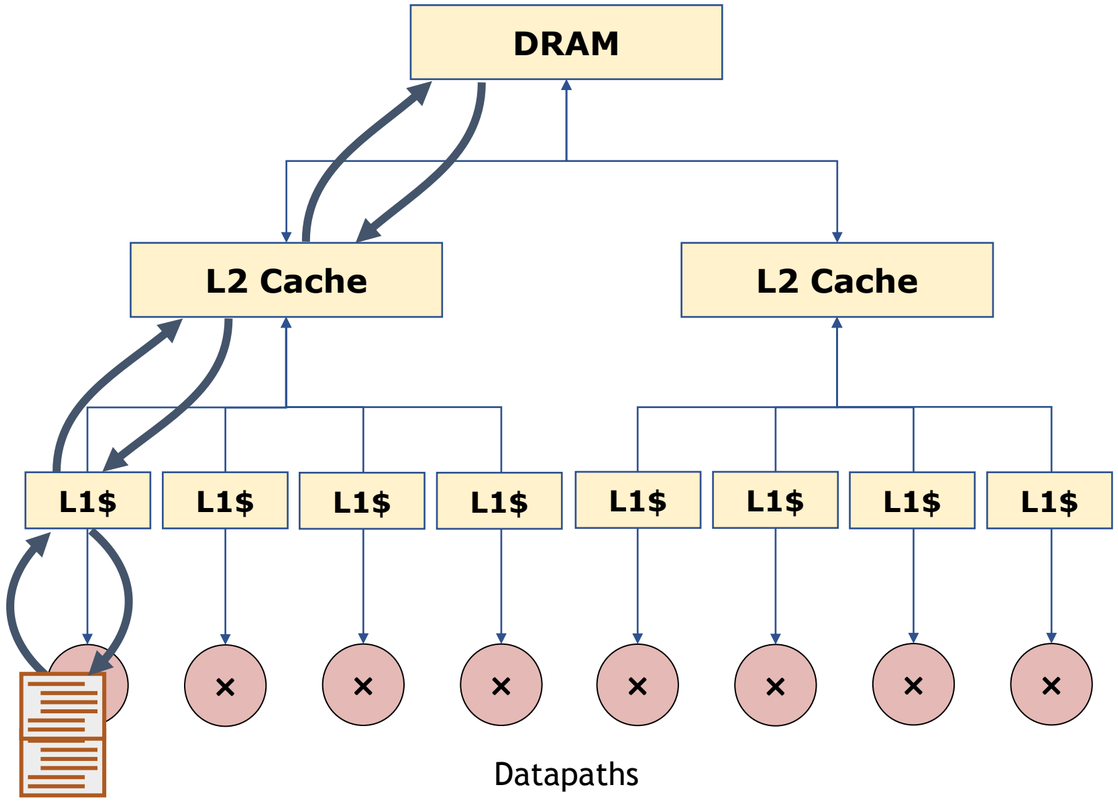
Application Compilation Comparison for
Push Memory Accelerator

Qiaoyi Liu

Pull Memory (CPU, GPU)

Pull Memory

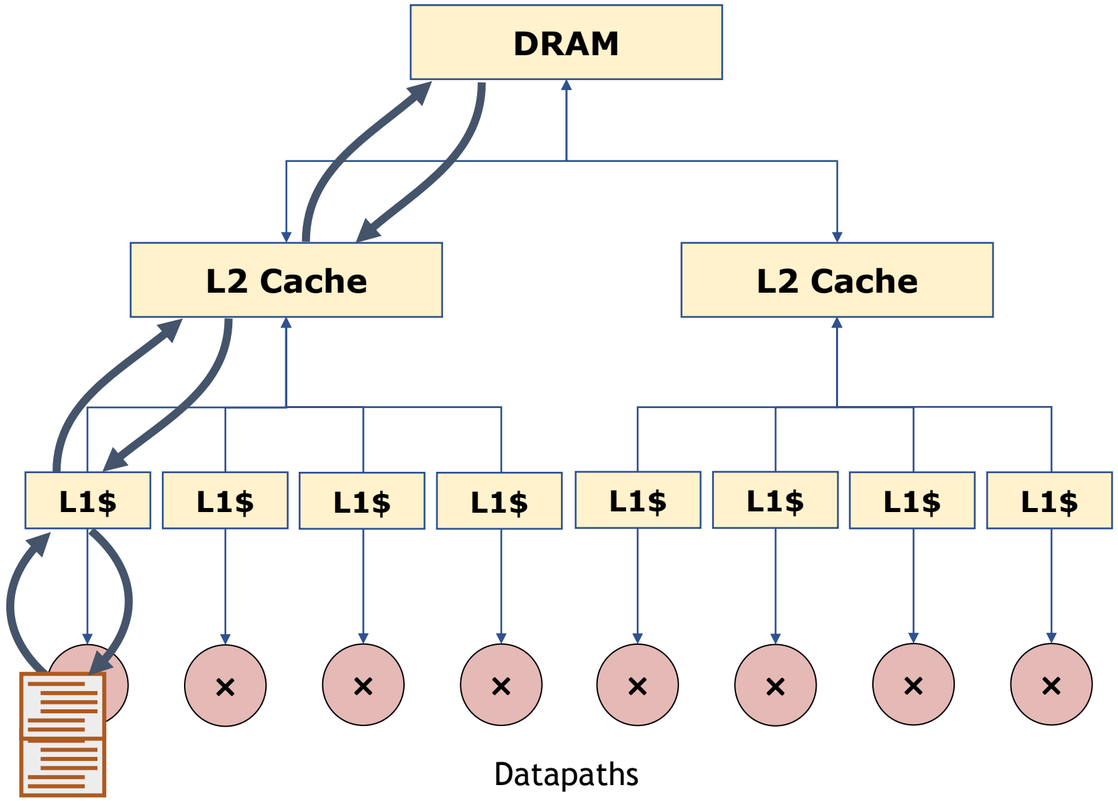
*Implicit Coupled Data Orchestration (**ICDO**)*



Push Memory Accelerators (Amber, Simba, …)

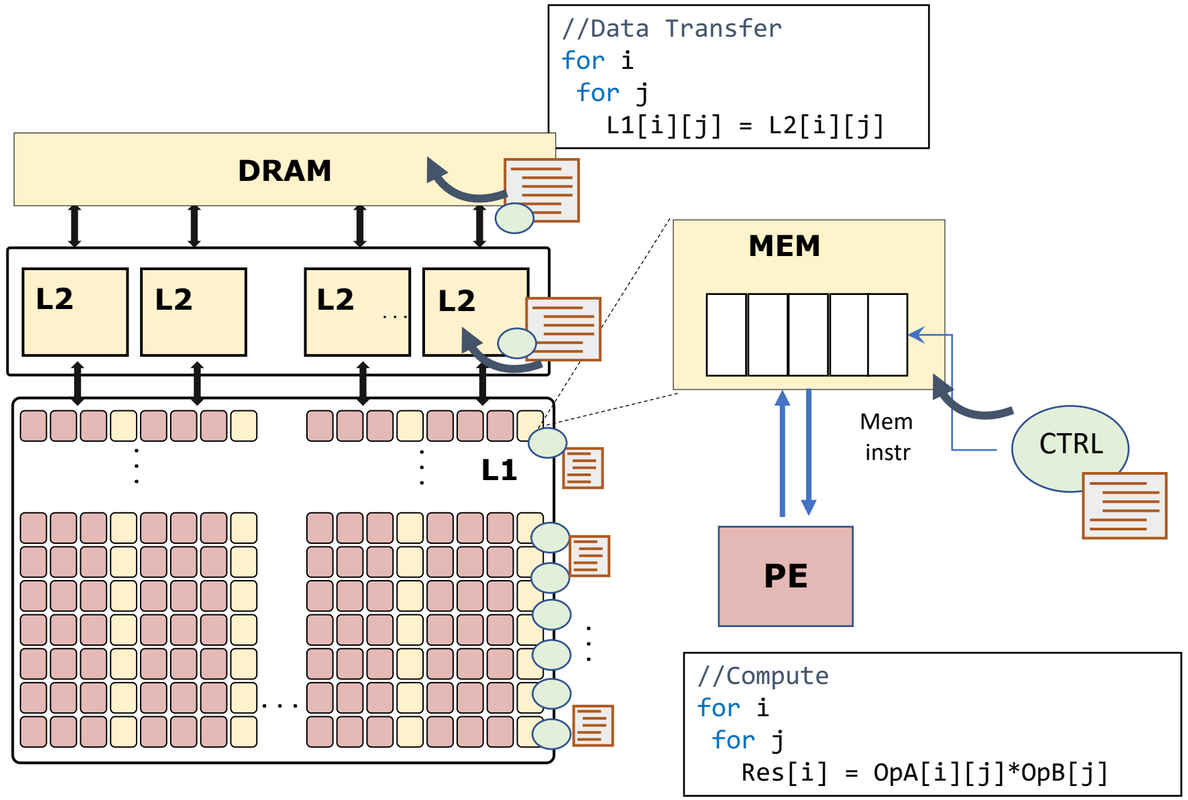
Pull Memory

*Implicit Coupled Data Orchestration (**ICDO**)*



Push Memory Model

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

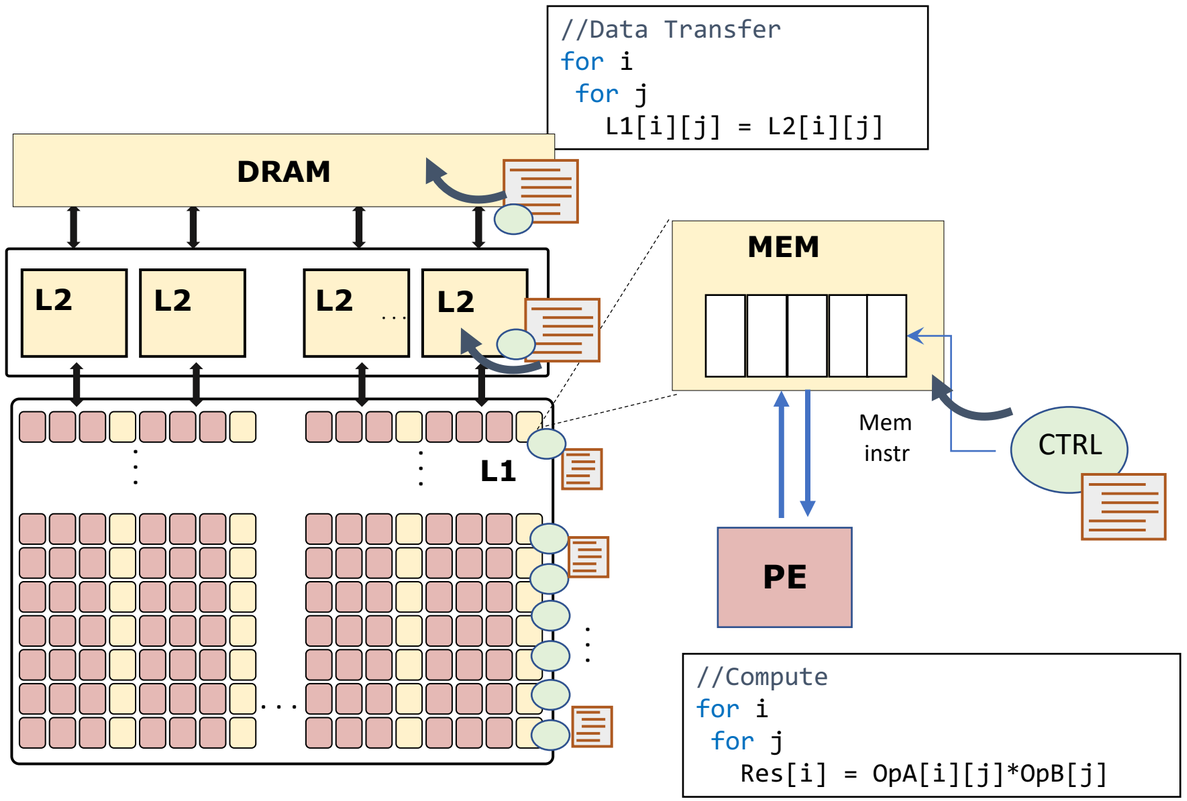


Benefits along with challenges: need a compiler

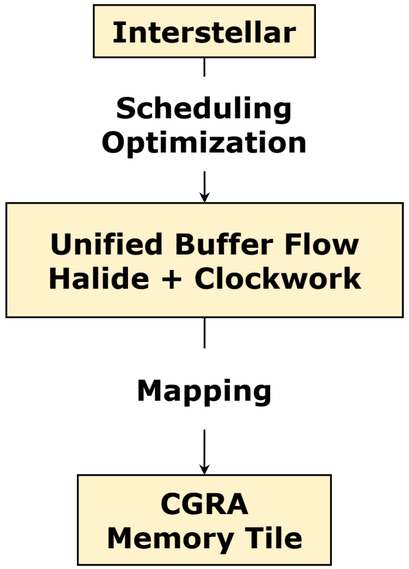
☺ Latency
☺ Energy efficiency
☹ A unique Programming challenge

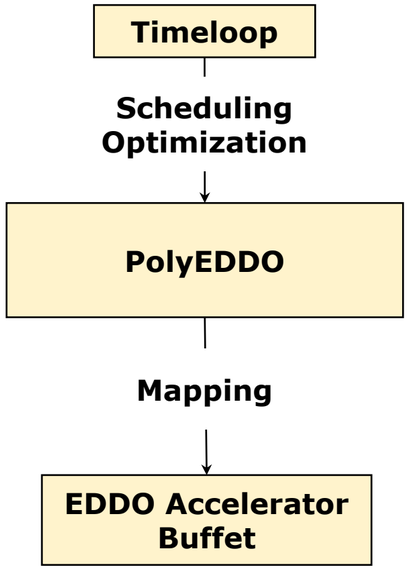
Push Memory Model

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



Application compilation tool chain





Application Compiler's Task

- For each stmt in the iteration space
  - **Scheduling**: find out the execution time
  - **Mapping**: map to the physical hardware that execute this operation

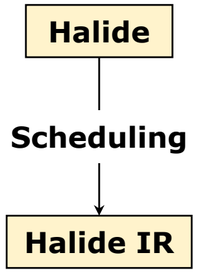
- There is a unique stmt: data transfer between memories

Quick Overview of Unified Buffer

```
brighten(x, y) = input(x, y) * 2;
blur(x, y) = (brighten(x, y  ) + brighten(x+1, y  ) +
              brighten(x, y+1) + brighten(x+1, y+1))/4;
blur.tile(x, y, xo, yo, xi, yi, 63, 63)
    .hw_accelerate(xi, xo);
brighten.store_at(blur, xo)
        .compute_at(blur, xo);
input.stream_to_accelerator();
```

```
for (y, 0, 64)
 for (x, 0, 64)
  brighten(x, y) = input(x, y) * 2;

for (y, 0, 63)
 for (x, 0, 63)
  blur(x, y) = (brighten(x, y) + brighten(x+1, y) +
                brighten(x, y+1) + brighten(x+1, y+1))/4;
```



Halide schedule:

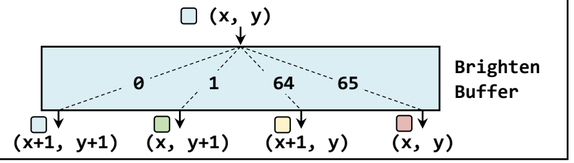
- Loop order, tiling, Interleaving granularity
- Create parallelization
- Create memory hierarchy

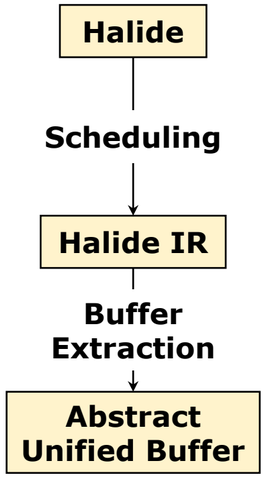
Quick Overview of Unified Buffer

```
brighten(x, y) = input(x, y) * 2;
blur(x, y) = (brighten(x, y  ) + brighten(x+1, y  ) +
              brighten(x, y+1) + brighten(x+1, y+1))/4;
blur.tile(x, y, xo, yo, xi, yi, 63, 63)
    .hw_accelerate(xi, xo);
brighten.store_at(blur, xo)
        .compute_at(blur, xo);
input.stream_to_accelerator();
```

```
for (y, 0, 64)
 for (x, 0, 64)
  brighten(x, y) = input(x, y) * 2;

for (y, 0, 63)
 for (x, 0, 63)
  blur(x, y) = (brighten(x, y) + brighten(x+1, y) +
                brighten(x, y+1) + brighten(x+1, y+1))/4;
```





Halide schedule:

- Loop order, tiling, Interleaving granularity
- Create parallelization
- Create memory hierarchy

Buffer Extraction:

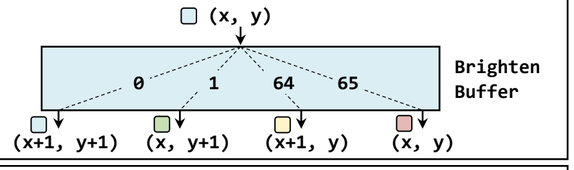
- Extract access pattern from compute centric loop to memory centric Ubuffer
- Analyze dependencies ,polyhedral scheduling create pipeline parallelism

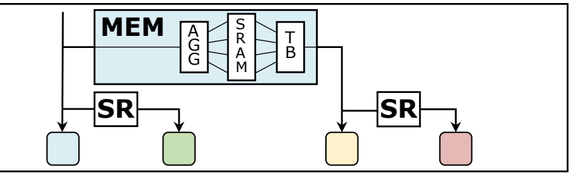
Quick Overview of Unified Buffer

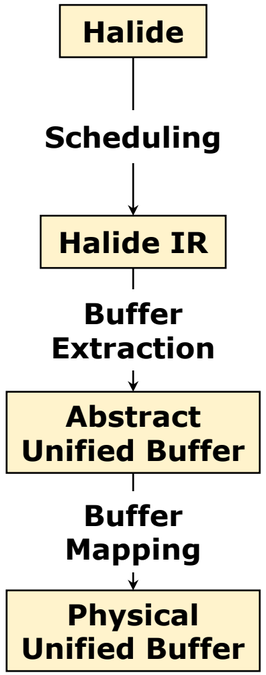
```
brighten(x, y) = input(x, y) * 2;
blur(x, y) = (brighten(x, y  ) + brighten(x+1, y  ) +
              brighten(x, y+1) + brighten(x+1, y+1))/4;
blur.tile(x, y, xo, yo, xi, yi, 63, 63)
    .hw_accelerate(xi, xo);
brighten.store_at(blur, xo)
        .compute_at(blur, xo);
input.stream_to_accelerator();
```

```
for (y, 0, 64)
 for (x, 0, 64)
  brighten(x, y) = input(x, y) * 2;

for (y, 0, 63)
 for (x, 0, 63)
  blur(x, y) = (brighten(x, y) + brighten(x+1, y) +
                brighten(x, y+1) + brighten(x+1, y+1))/4;
```







Halide schedule:

- Loop order, tiling, Interleaving granularity
- Create parallelization
- Create memory hierarchy

Buffer Extraction:

- Extract access pattern from compute centric loop to memory centric Ubuffer
- Analyze dependencies ,polyhedral scheduling create pipeline parallelism

Buffer mapping (Satisfy the schedule created above)**:**

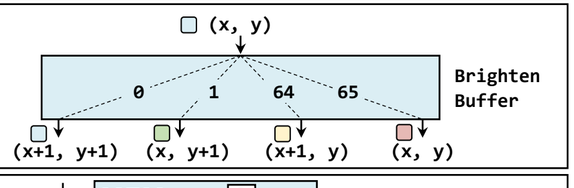
- Mapping **UBuffer** to **physical hardware** with constraint (capacity, port…)
- Reuse analysis to reduce resource consumption

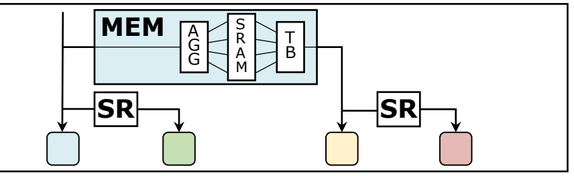
Problem 1:
Scheduling and mapping interact with each other and it's an iterative approach

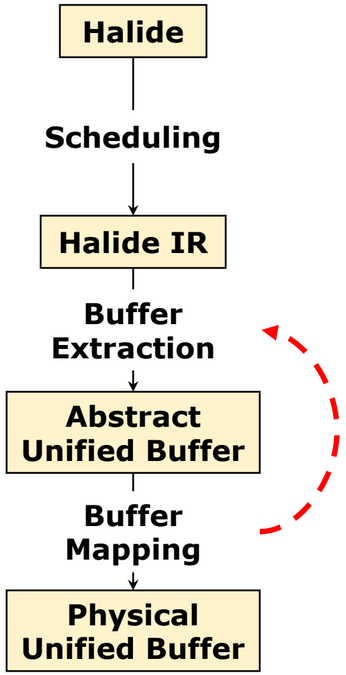
```
brighten(x, y) = input(x, y) * 2;
blur(x, y) = (brighten(x, y  ) + brighten(x+1, y  ) +
              brighten(x, y+1) + brighten(x+1, y+1))/4;
blur.tile(x, y, xo, yo, xi, yi, 63, 63)
    .hw_accelerate(xi, xo);
brighten.store_at(blur, xo)
        .compute_at(blur, xo);
input.stream_to_accelerator();
```

```
for (y, 0, 64)
 for (x, 0, 64)
  brighten(x, y) = input(x, y) * 2;

for (y, 0, 63)
 for (x, 0, 63)
  blur(x, y) = (brighten(x, y) + brighten(x+1, y) +
                brighten(x, y+1) + brighten(x+1, y+1))/4;
```







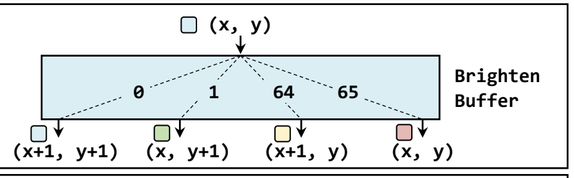
Hardware constraints affect scheduling,
and we did not know the hardware until mapping is finished

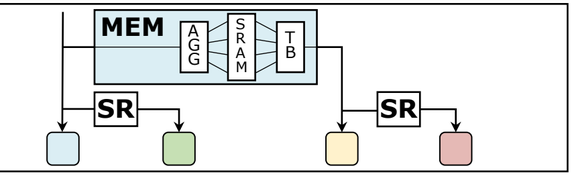
Problem 2:
Do we need [*global*] [*cycle accurate*] schedule?

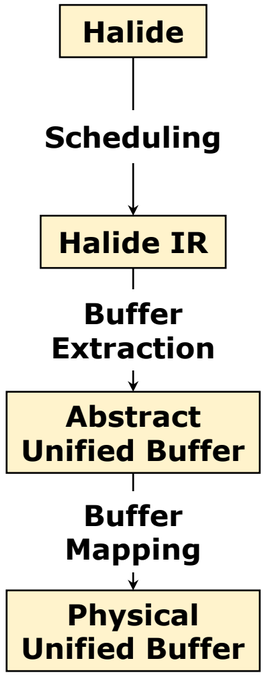
```
brighten(x, y) = input(x, y) * 2;
blur(x, y) = (brighten(x, y  ) + brighten(x+1, y  ) +
              brighten(x, y+1) + brighten(x+1, y+1))/4;
blur.tile(x, y, xo, yo, xi, yi, 63, 63)
    .hw_accelerate(xi, xo);
brighten.store_at(blur, xo)
        .compute_at(blur, xo);
input.stream_to_accelerator();
```

```
for (y, 0, 64)
 for (x, 0, 64)
  brighten(x, y) = input(x, y) * 2;

for (y, 0, 63)
 for (x, 0, 63)
  blur(x, y) = (brighten(x, y) + brighten(x+1, y) +
                brighten(x, y+1) + brighten(x+1, y+1))/4;
```







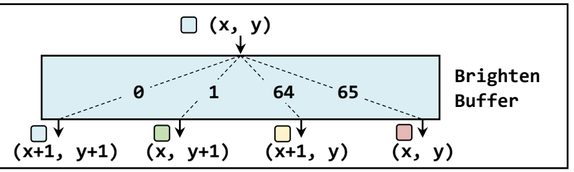
In order to create pipeline parallelism,
we create a global cycle accurate scheduler
and build a hardware controller based on this abstraction
efficient ?

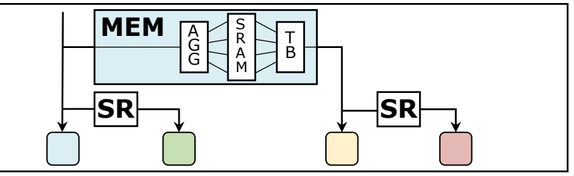
Problem 3:
Lowering to loop nest, lifting to pipeline parallelism

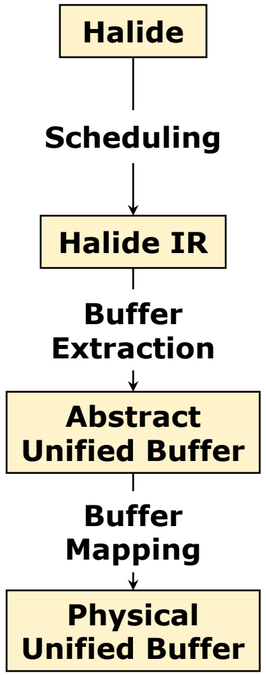
```
brighten(x, y) = input(x, y) * 2;
blur(x, y) = (brighten(x, y  ) + brighten(x+1, y  ) +
              brighten(x, y+1) + brighten(x+1, y+1))/4;
blur.tile(x, y, xo, yo, xi, yi, 63, 63)
    .hw_accelerate(xi, xo);
brighten.store_at(blur, xo)
        .compute_at(blur, xo);
input.stream_to_accelerator();
```

```
for (y, 0, 64)
 for (x, 0, 64)
  brighten(x, y) = input(x, y) * 2;

for (y, 0, 63)
 for (x, 0, 63)
  blur(x, y) = (brighten(x, y) + brighten(x+1, y) +
                brighten(x, y+1) + brighten(x+1, y+1))/4;
```







Lowering

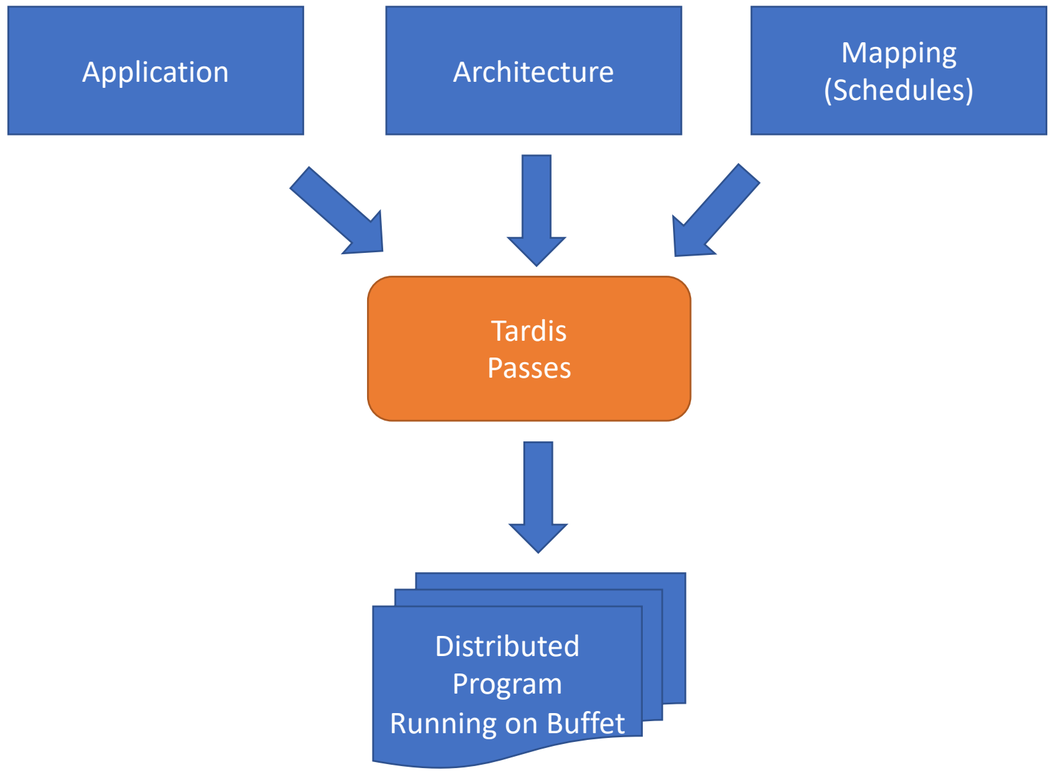
Lifting

Imperative loopnest is more constrained than the dataflow hardware execution
Hard to express pipeline parallelism

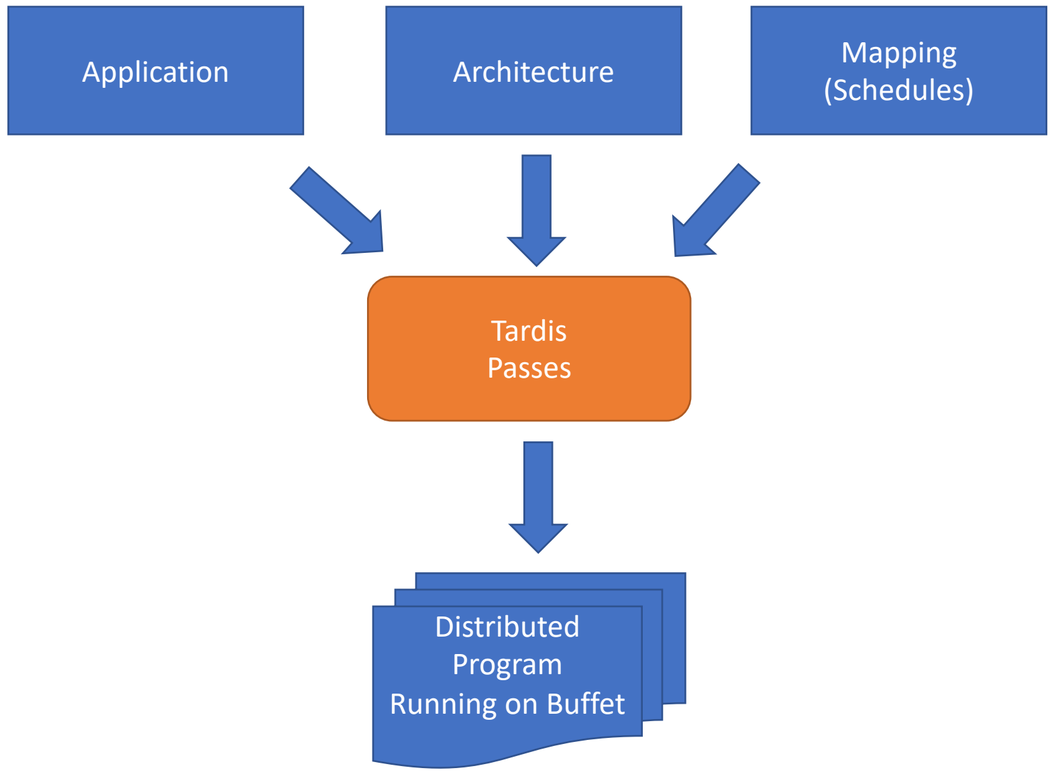
Problems

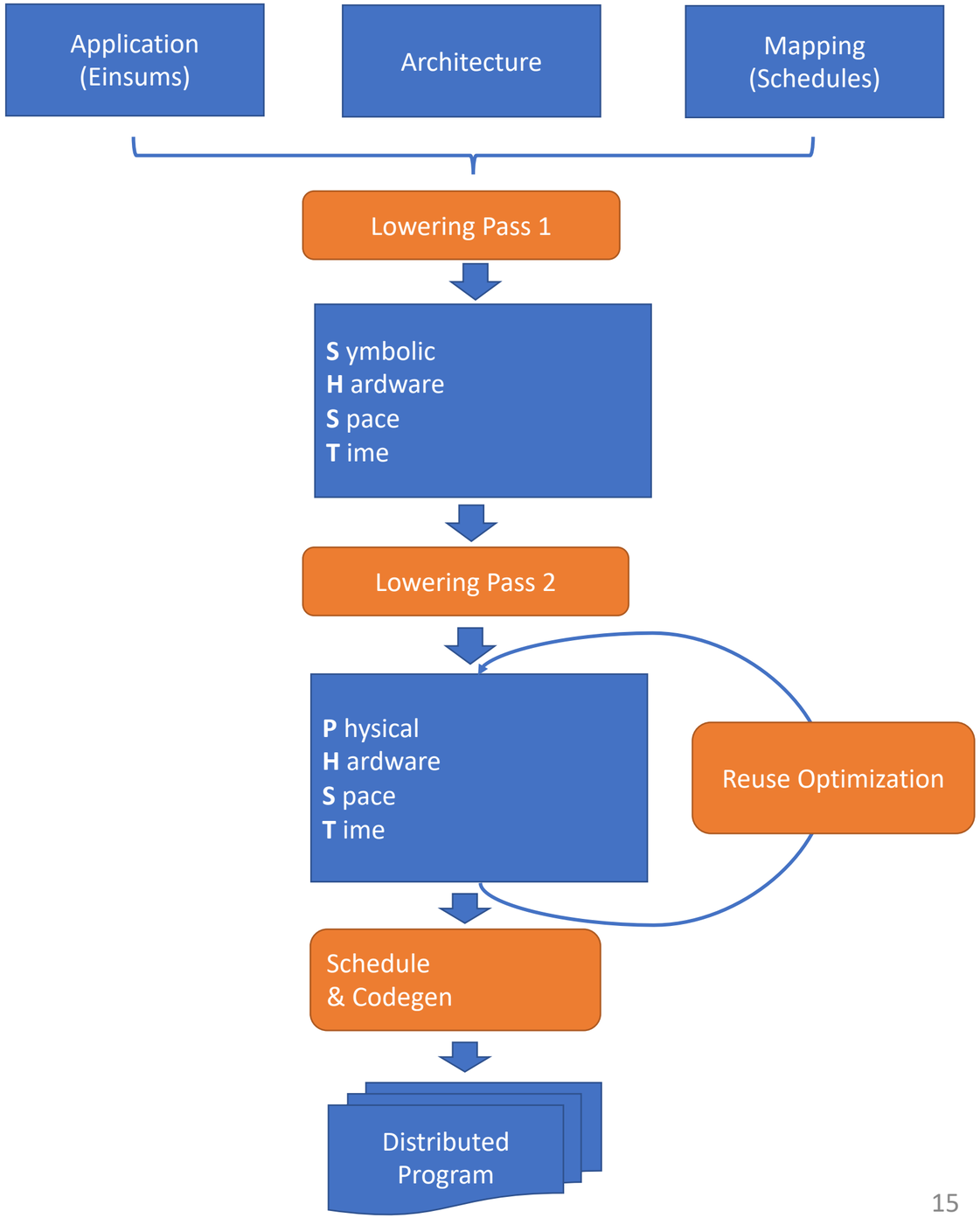
- Scheduling and mapping is related. (Iterative approach)
- Could we optimize cycle accurate schedule?
- Halide lR(Loopnest), any better representation?

PolyEDDO Flow



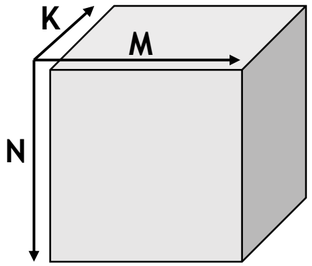
Compiler passes

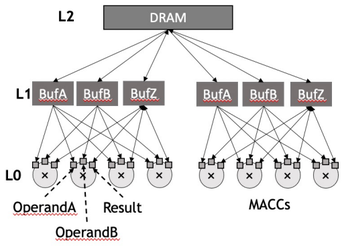




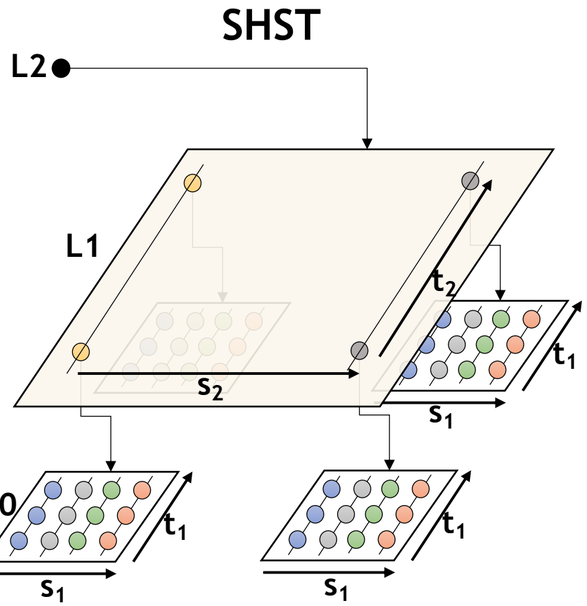
Lowering Pass

$$Z_{mn} = A_{mk}B_{kn}$$

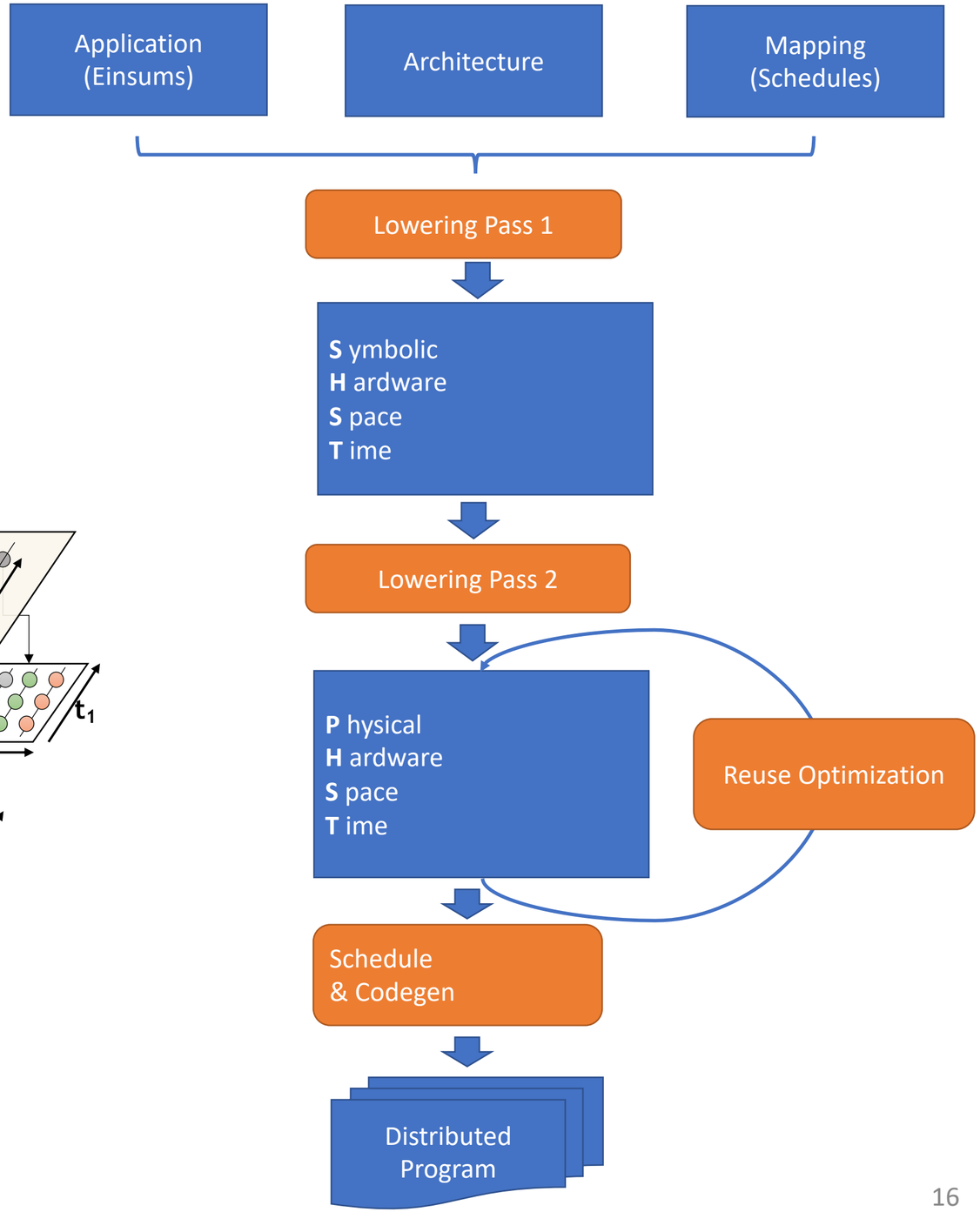




Lowering Pass 1

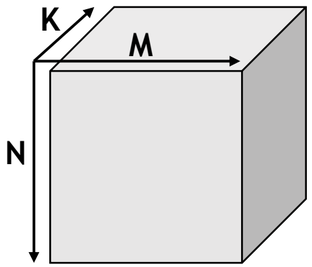


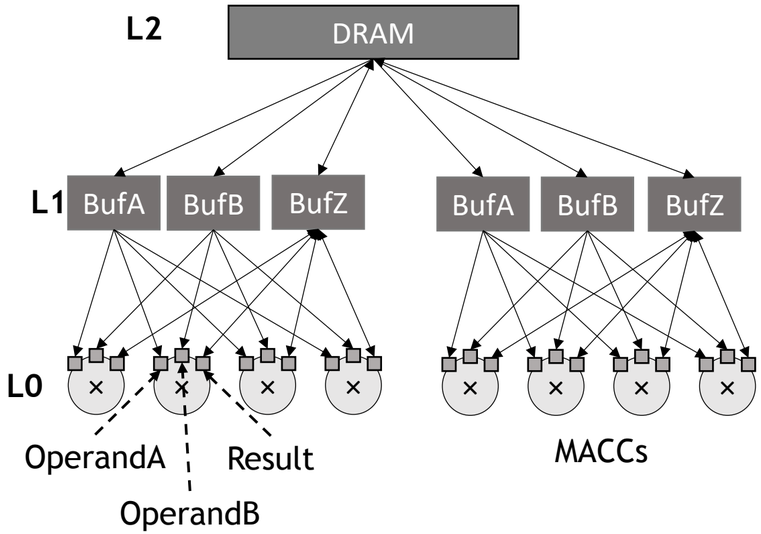
Loop Tiling, loop ordering,
memory hierarchy, parallelization



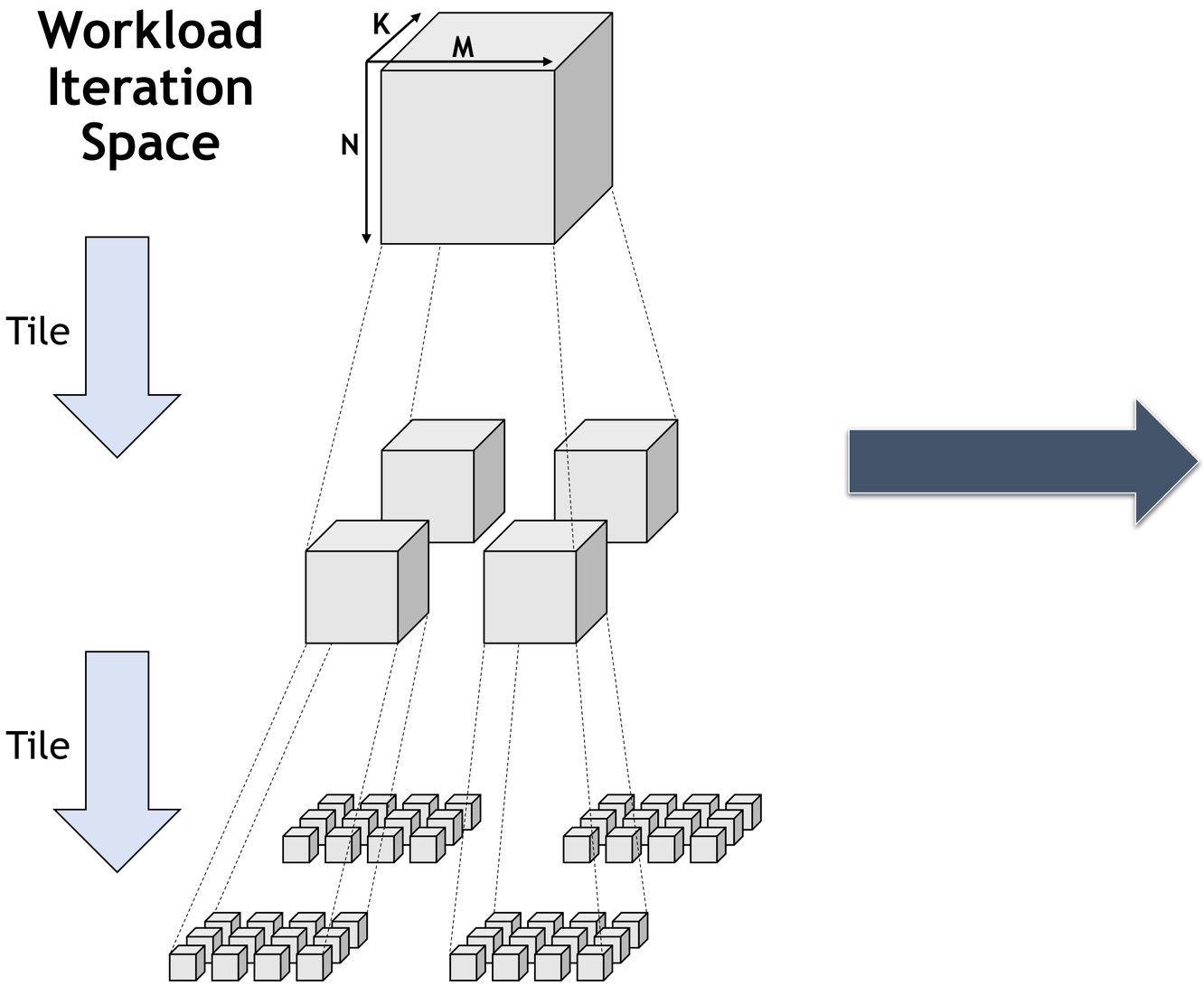
Example: Map GEMM to 3-level Arch

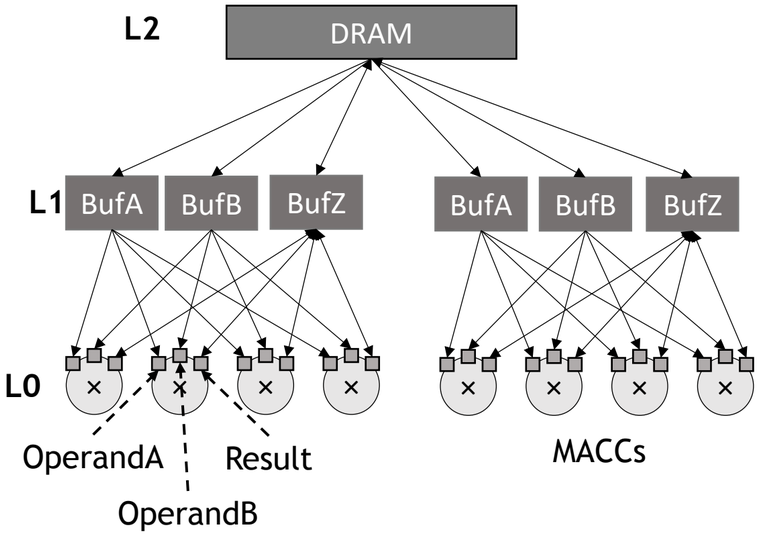
$$Z_{mn} = A_{mk}B_{kn}$$





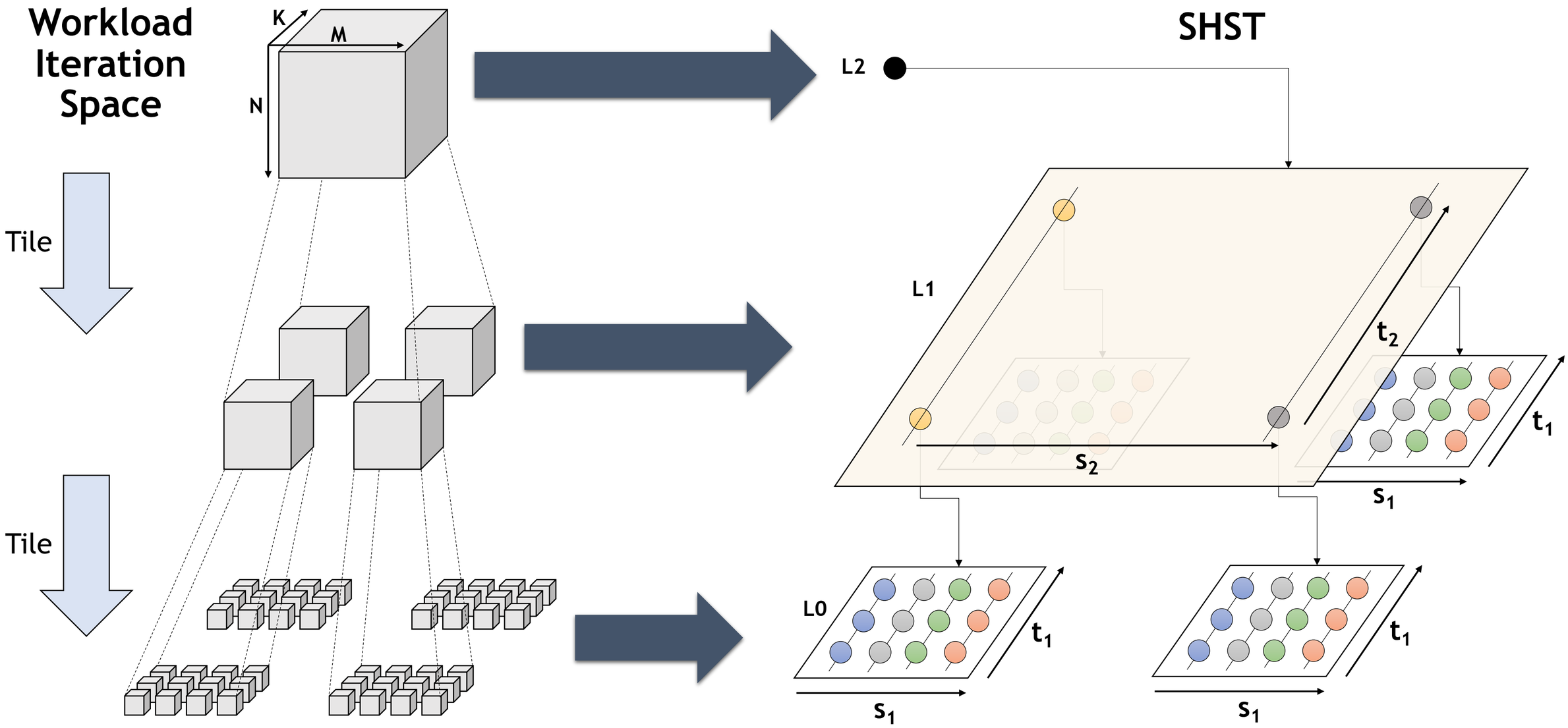
Tiling the Iteration Space



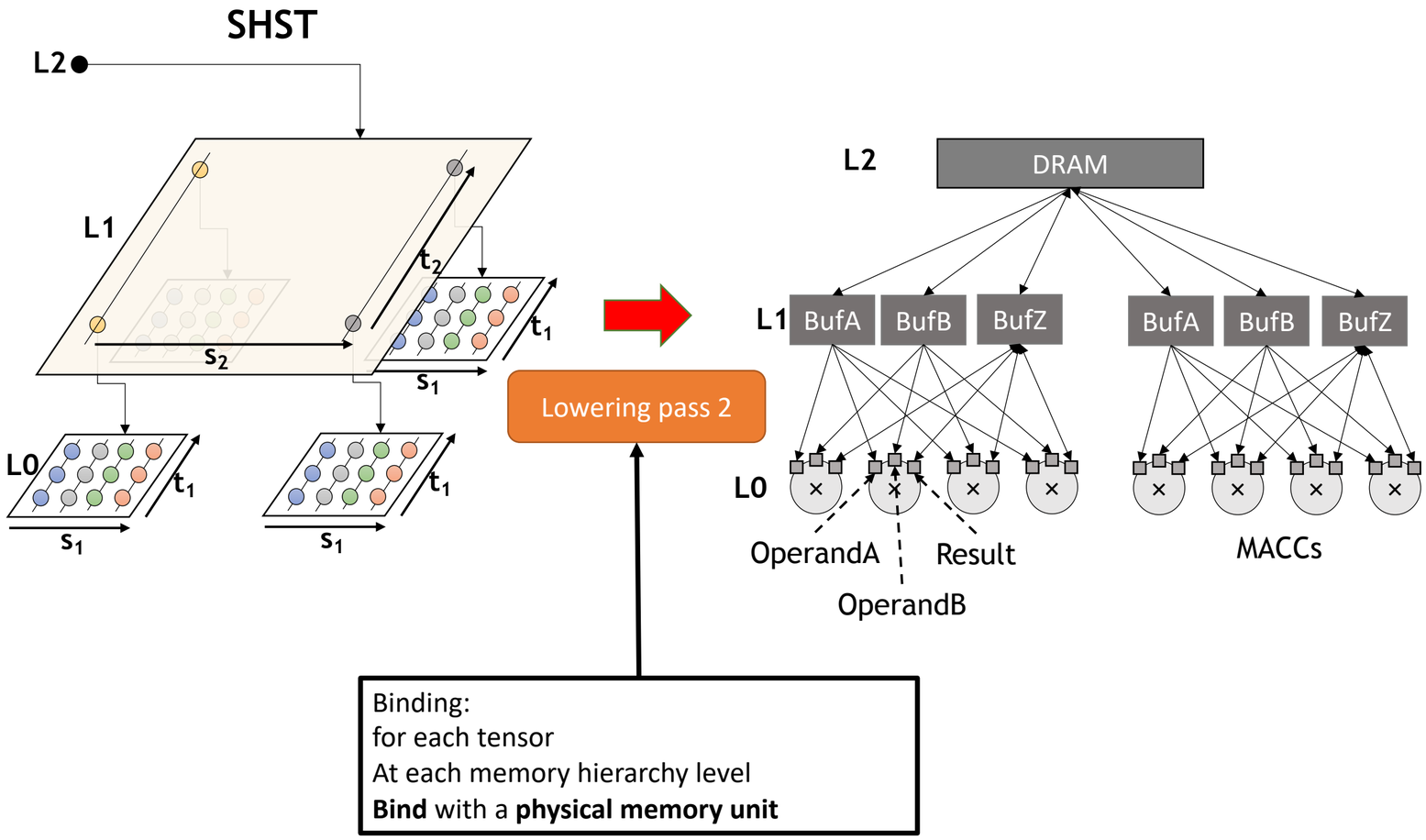


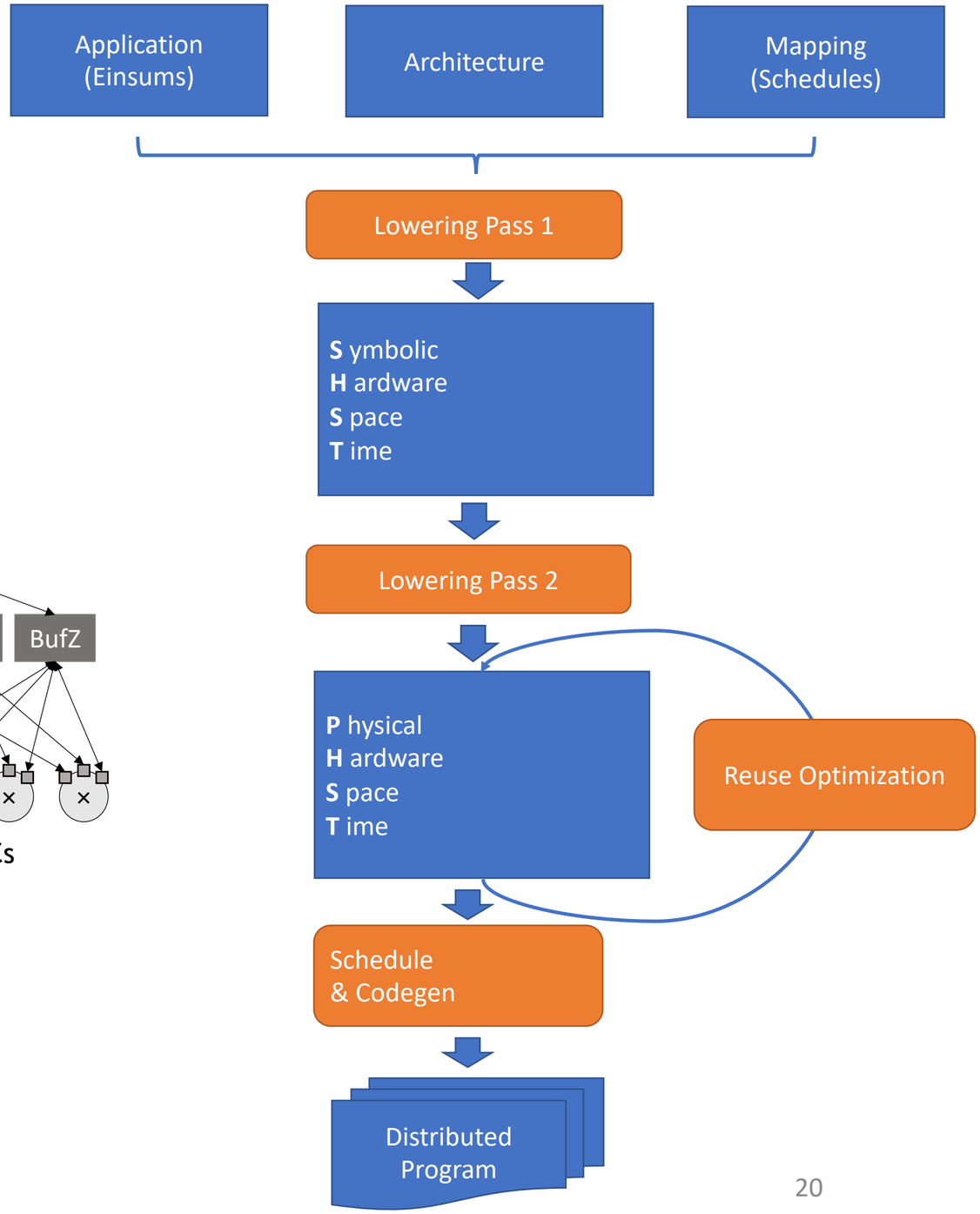
Symbolic Hardware Space-Time (SHST)

A projection from tiled iteration space to logical memory hierarchy(s, t)



Data centric lowering

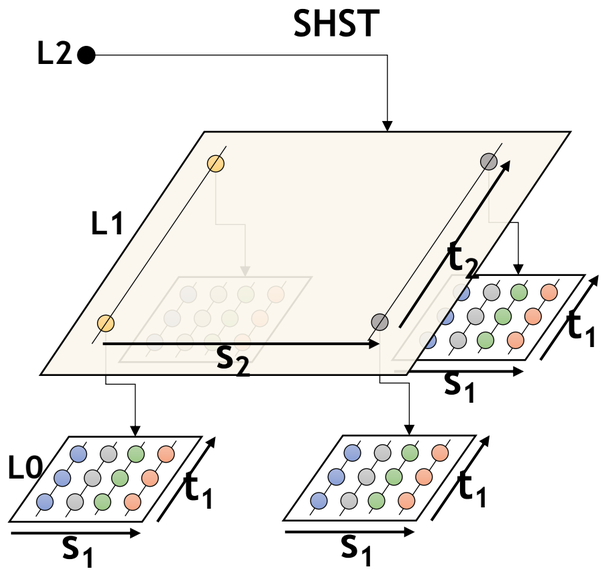




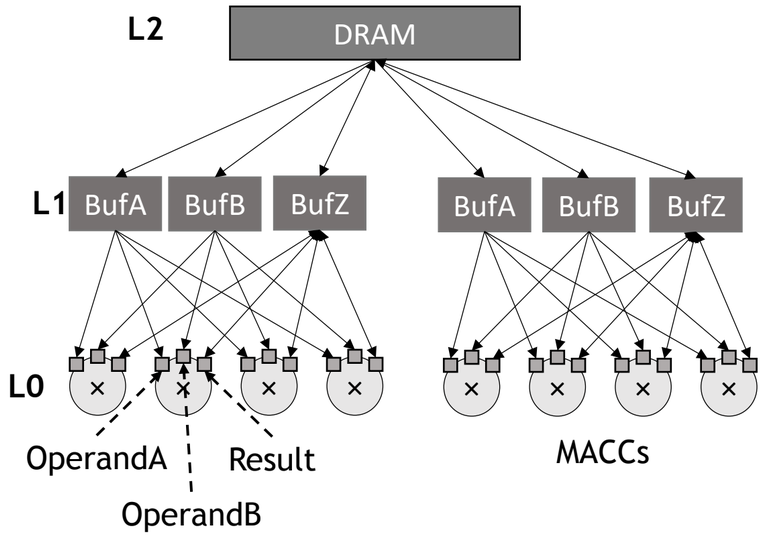
Binding Tensor with Memory

$$Z_{mn} = A_{mk}B_{kn}$$

	L2	L1	L0
Data Space:			
Z_{mn}	DRAM	Buf_Z	Result
A_{mk}	DRAM	Buf_A	Op_A
B_{kn}	DRAM	Buf_B	Op_B

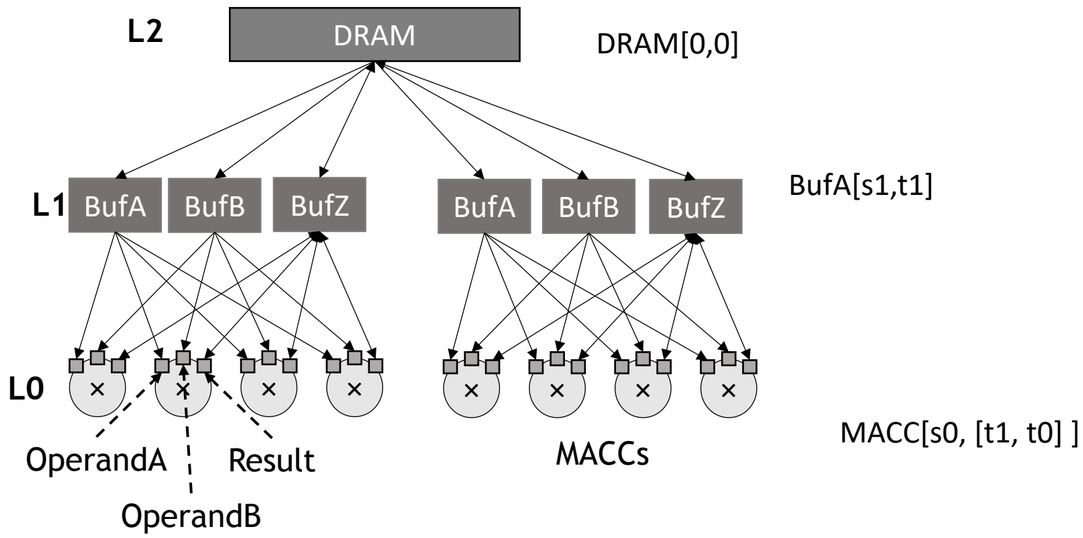


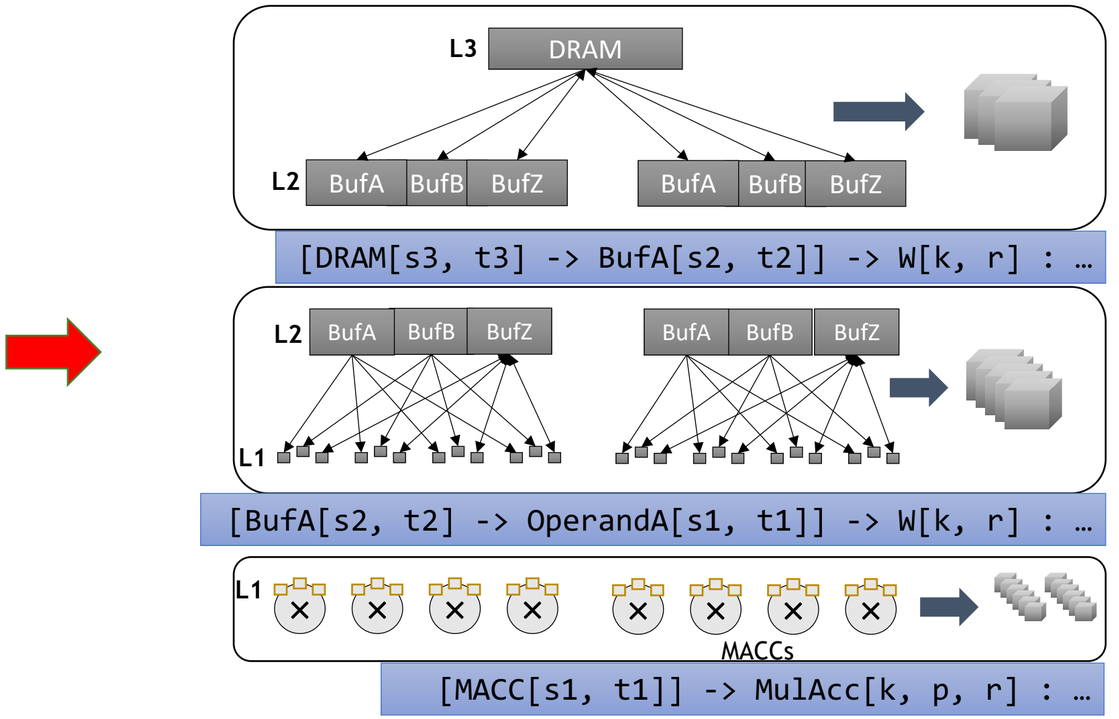
Binding
(Data Centric)



PHST

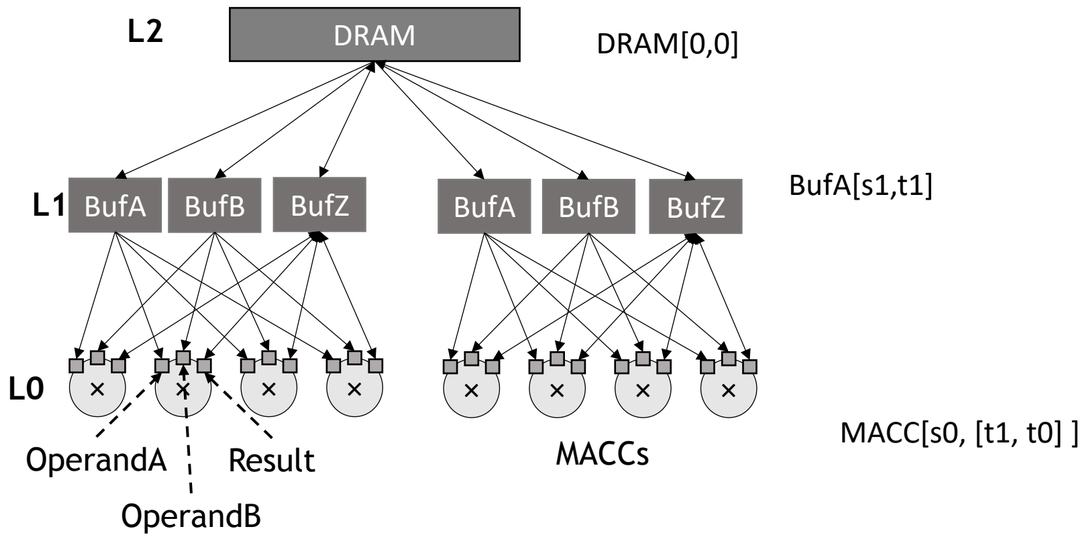
- The connection between PHST
  - Data transfer actions
  - Compute actions
  - Further map to the program running on controller/MACCs

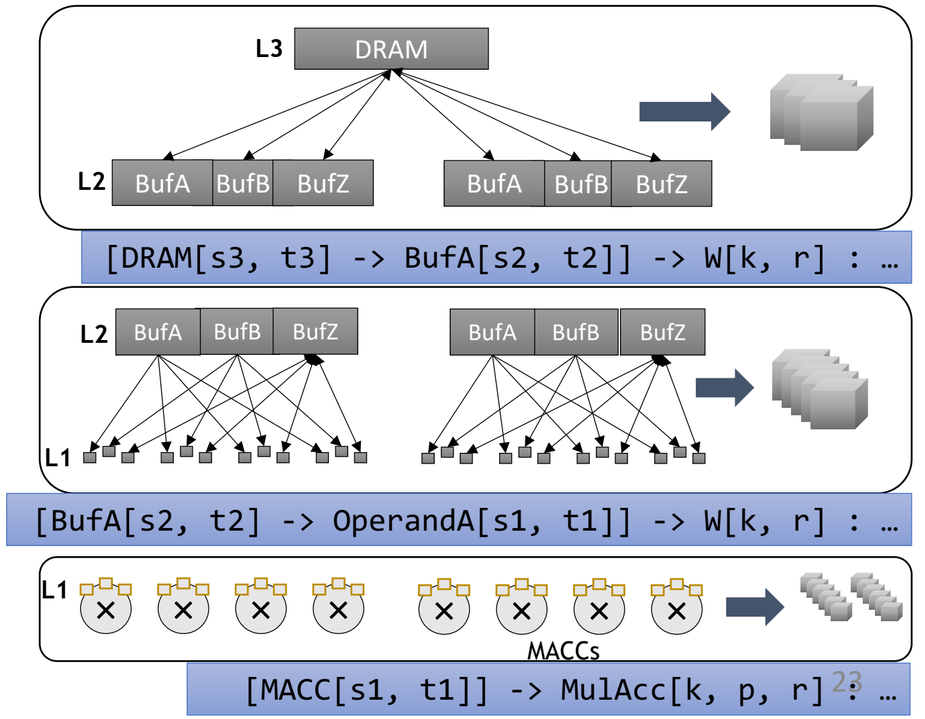




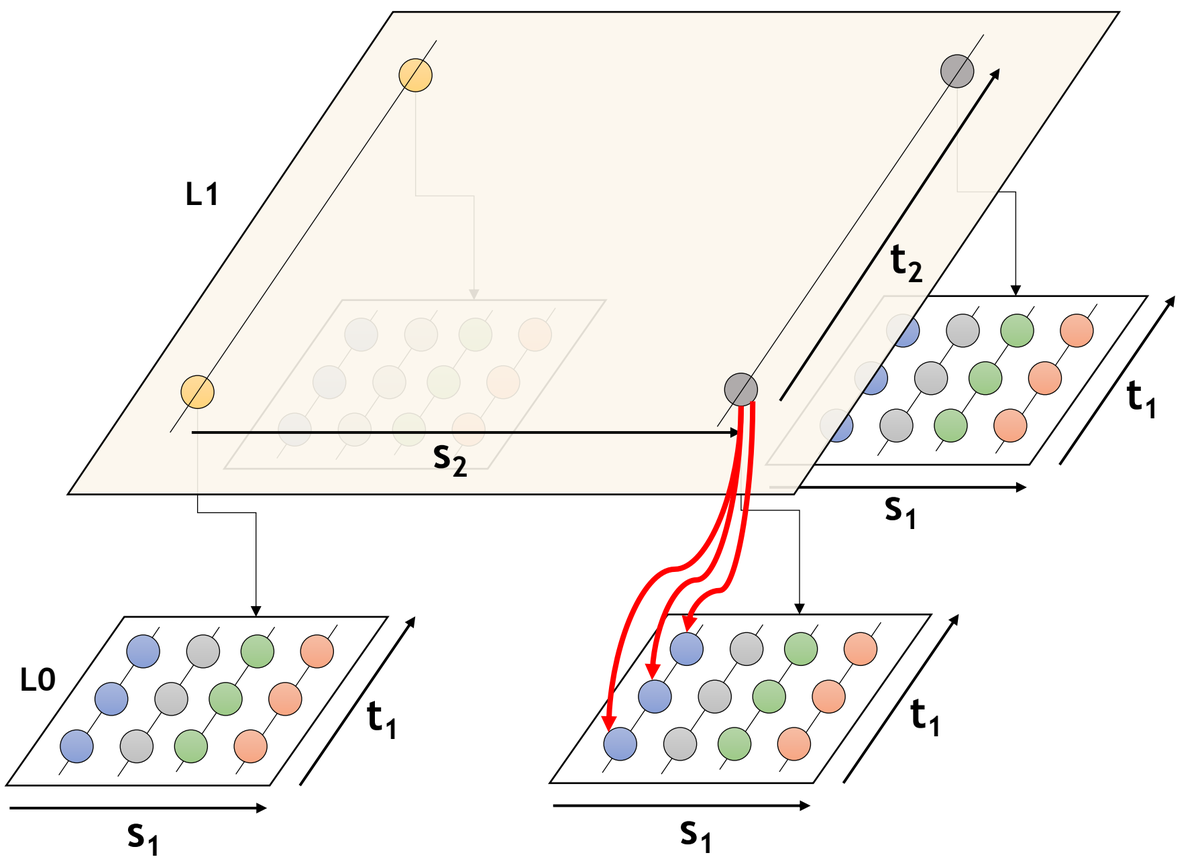
Reuse optimization on PHST

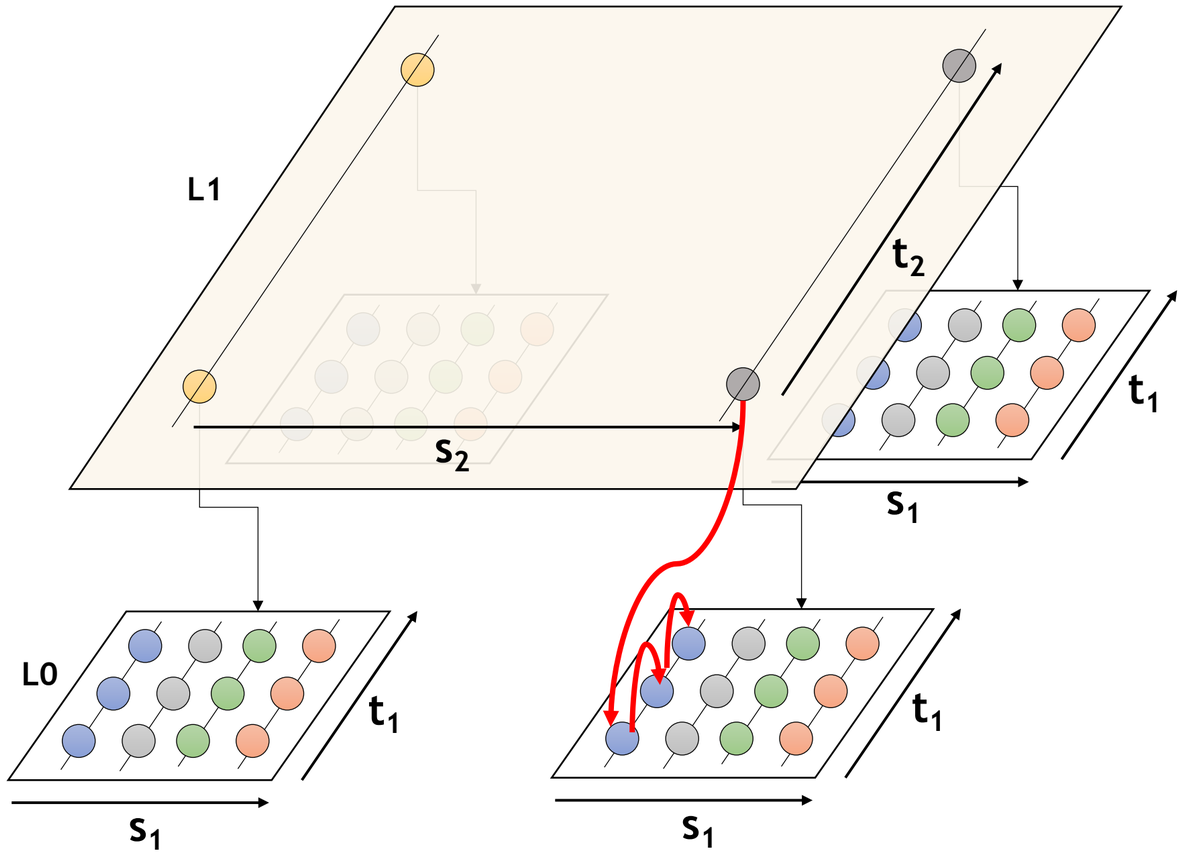
- Each PHST Node also has space-time coordinate
  - Reuse analysis:
    - Analyze the access map for each PHST along space/time coordinate
    - Eliminate the re-access



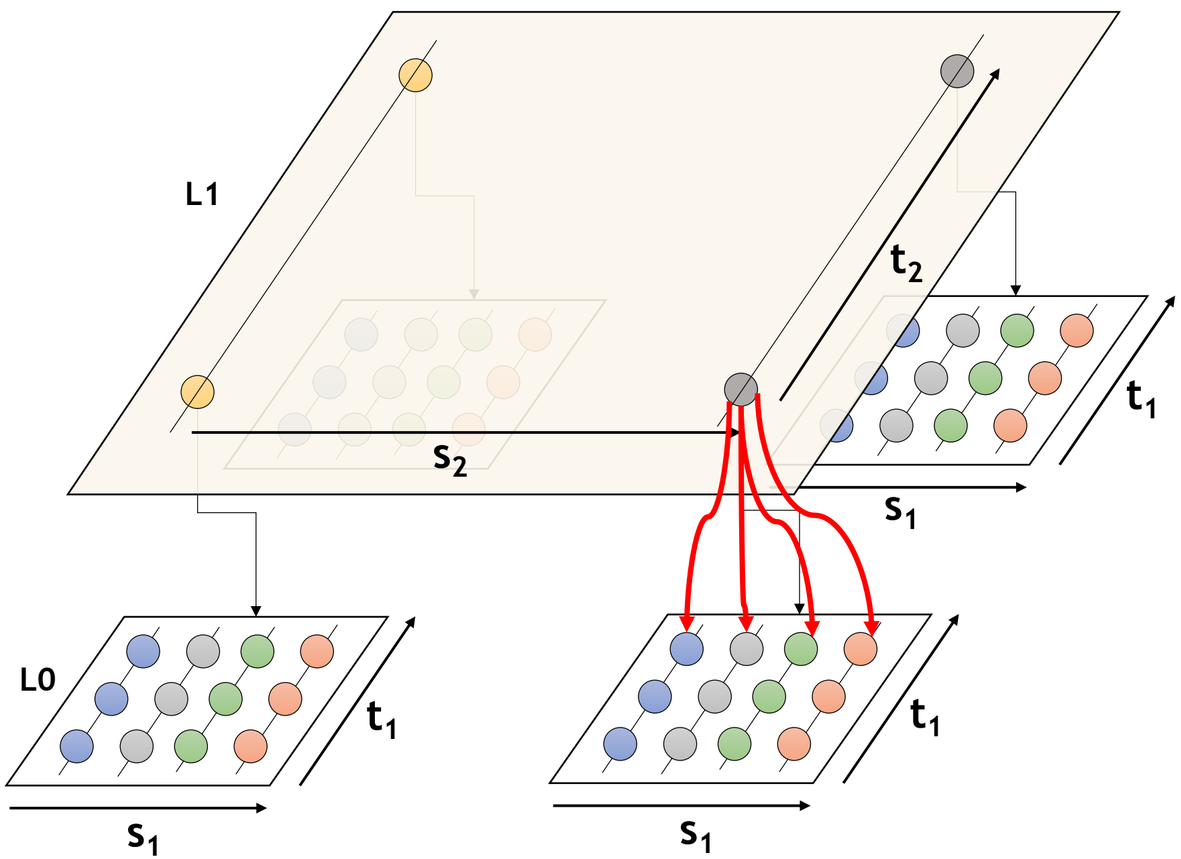


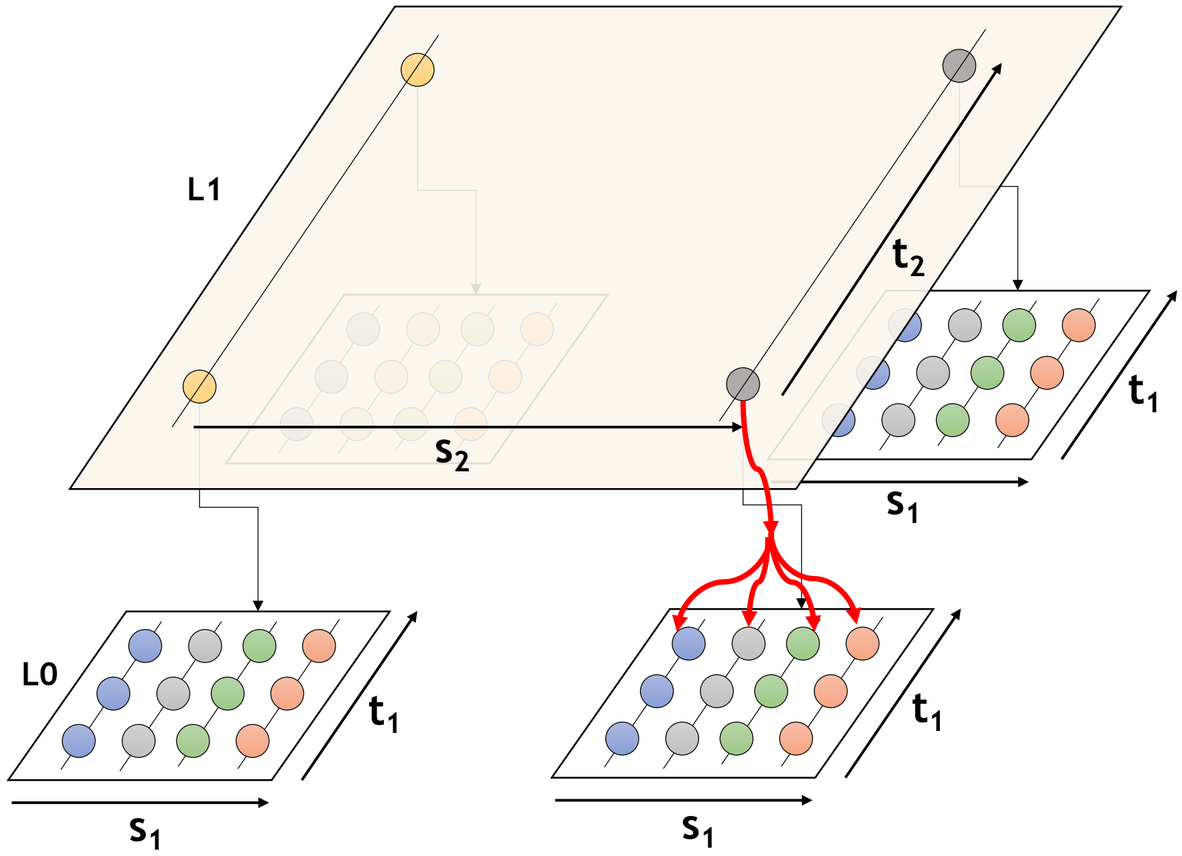
Reuse optimization along time dimension





Reuse optimization along space dimension: Multicast



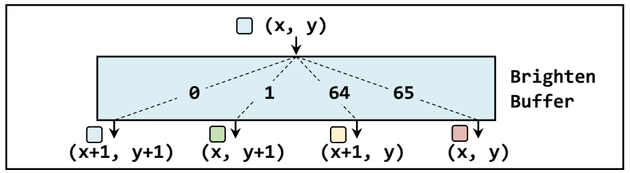


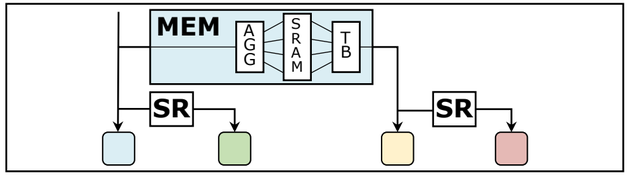
Compare with Ubuffer Flow

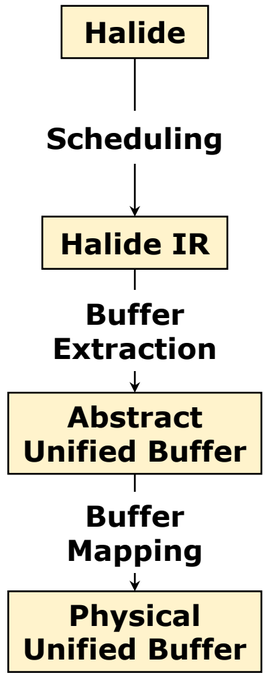
```
brighten(x, y) = input(x, y) * 2;
blur(x, y) = (brighten(x, y  ) + brighten(x+1, y  ) +
              brighten(x, y+1) + brighten(x+1, y+1))/4;
blur.tile(x, y, xo, yo, xi, yi, 63, 63)
    .hw_accelerate(xi, xo);
brighten.store_at(blur, xo)
        .compute_at(blur, xo);
input.stream_to_accelerator();
```

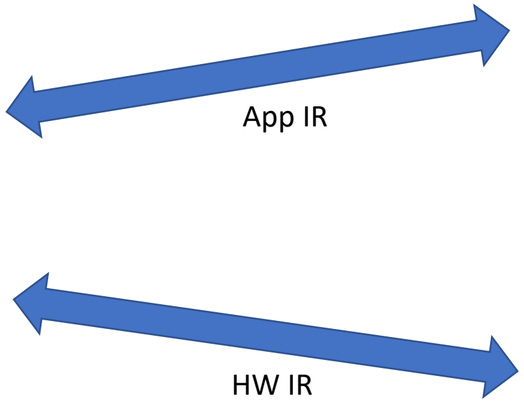
```
for (y, 0, 64)
 for (x, 0, 64)
  brighten(x, y) = input(x, y) * 2;

for (y, 0, 63)
 for (x, 0, 63)
  blur(x, y) = (brighten(x, y) + brighten(x+1, y) +
                brighten(x, y+1) + brighten(x+1, y+1))/4;
```







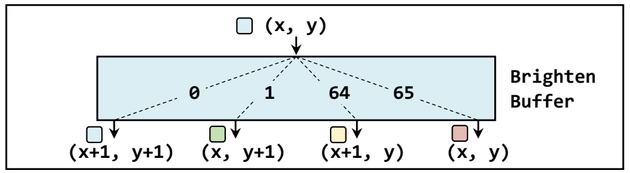


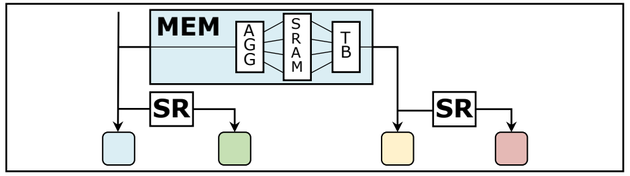
Compare with Ubuffer Flow

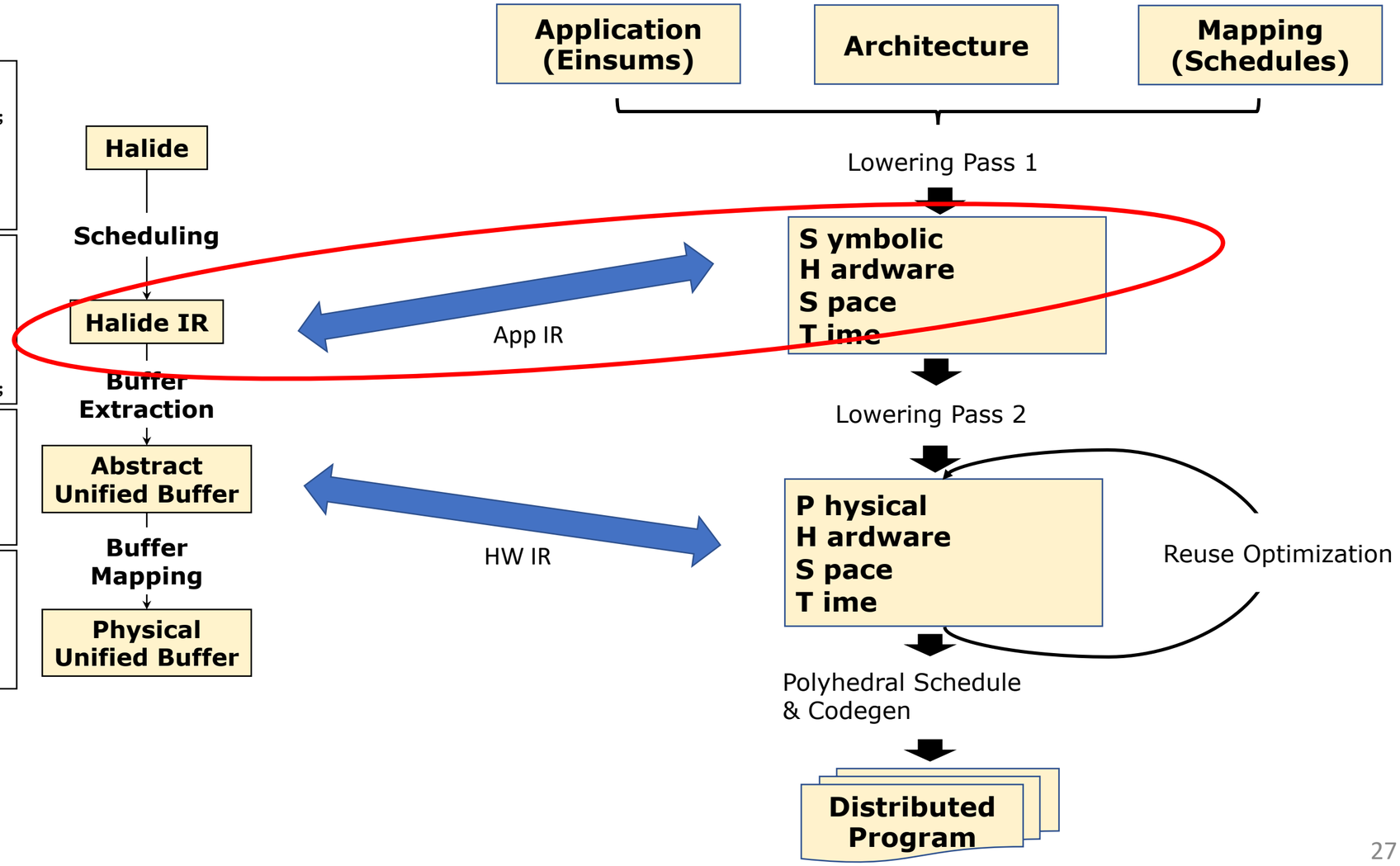
```
brighten(x, y) = input(x, y) * 2;
blur(x, y) = (brighten(x, y  ) + brighten(x+1, y  ) +
              brighten(x, y+1) + brighten(x+1, y+1))/4;
blur.tile(x, y, xo, yo, xi, yi, 63, 63)
    .hw_accelerate(xi, xo);
brighten.store_at(blur, xo)
        .compute_at(blur, xo);
input.stream_to_accelerator();
```

```
for (y, 0, 64)
 for (x, 0, 64)
  brighten(x, y) = input(x, y) * 2;

for (y, 0, 63)
 for (x, 0, 63)
  blur(x, y) = (brighten(x, y) + brighten(x+1, y) +
                brighten(x, y+1) + brighten(x+1, y+1))/4;
```







Halide IR Loopnest vs SHST

- SHST logic memory hierarchical tree with 2D rank
  - Two characteristics:
    - Parallelization
    - Memory hierarchy

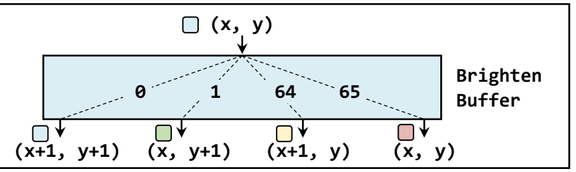
- Halide IR: loop nest
  - Unrolled loop & Rolled loop
    - Reinterpret loop nest semantic, relax the consistency model
    - Not as formal as the other
  - Data transfer is explicitly represented by copy stmt
    - Loop nest is more flexible:
      - bypass a hierarchy level
      - Within one level (pipeline)
    - Root from the specific domain and architecture
      - Stencil Pipeline
      - DNN kernel

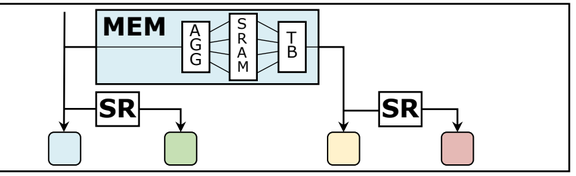
Compare with Ubuffer Flow

```
brighten(x, y) = input(x, y) * 2;
blur(x, y) = (brighten(x, y  ) + brighten(x+1, y  ) +
              brighten(x, y+1) + brighten(x+1, y+1))/4;
blur.tile(x, y, xo, yo, xi, yi, 63, 63)
    .hw_accelerate(xi, xo);
brighten.store_at(blur, xo)
        .compute_at(blur, xo);
input.stream_to_accelerator();
```

```
for (y, 0, 64)
 for (x, 0, 64)
  brighten(x, y) = input(x, y) * 2;

for (y, 0, 63)
 for (x, 0, 63)
  blur(x, y) = (brighten(x, y) + brighten(x+1, y) +
                brighten(x, y+1) + brighten(x+1, y+1))/4;
```





Halide

Scheduling

Halide IR

Buffer Extraction

Abstract Unified Buffer

Buffer Mapping

Physical Unified Buffer

App IR

HW IR

Application (Einsums)

Architecture

Mapping (Schedules)

Lowering Pass 1

S ymbolic H ardware S pace T ime

Lowering Pass 2

P hysical H ardware S pace T ime

Reuse Optimization

Polyhedral Schedule & Codegen

Distributed Program

Ubuffer vs PHST: Creation (explicit binding)

- Ubuffer: assign dedicated resource to minimum latency
  - Map to dedicated buffer and compute in pipeline
  - Bank the memory to max bandwidth
  - Problem:
    - mapping stage cannot always satisfy the schedule
    - relax the II in schedule (longer latency)
- PHST: Take the hardware binding information
  - Tensor->physical memory
  - Multi-tensor to one-memory: memory sharing
  - One-tensor to multi-memory: memory banking

Different HW Binding

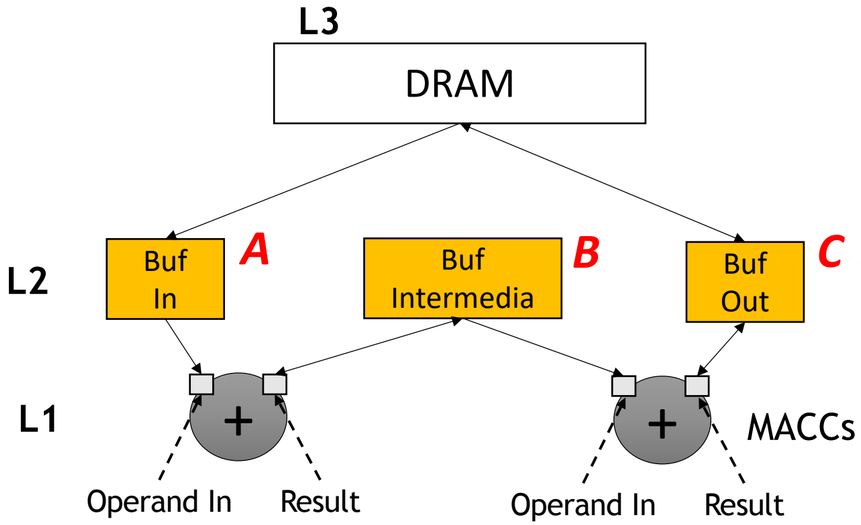
Einsum0: $B_i = A_{i+r}$

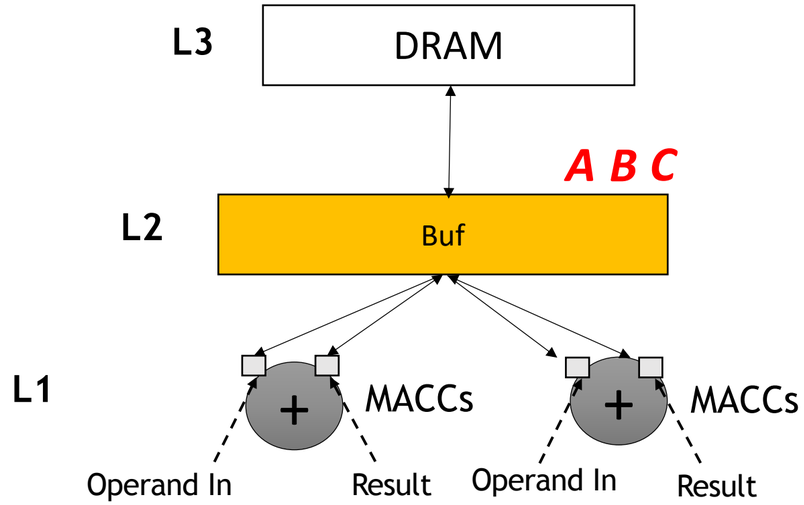
Einsum1: $C_i = B_{i+r}$

Back to back conv1D with all weight to be 1

```
for I in 0:10
 for r in 0:3
   B[i] += A[i+r]

for I in 0:8
 for r in 0:3
   C[i] += B[i+r]
```





Ubuffer vs PHST: HW binding in scheduling

- HW binding is a hint for mapping and make scheduling easier
  - **Compute** operation compete for PEs
  - **Memory** operation compete for memory ports

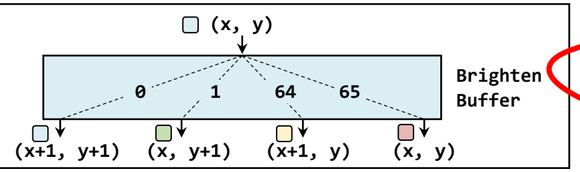
- Hardware resource aware scheduling
  - If someone can tell me this tensor is bind with which physical memory or even lake or pond, that’s helpful

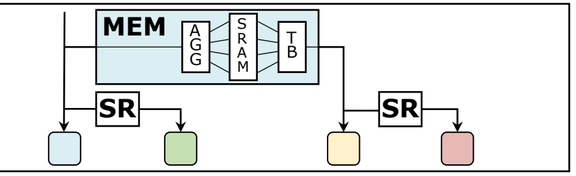
Compare with Ubuffer Flow

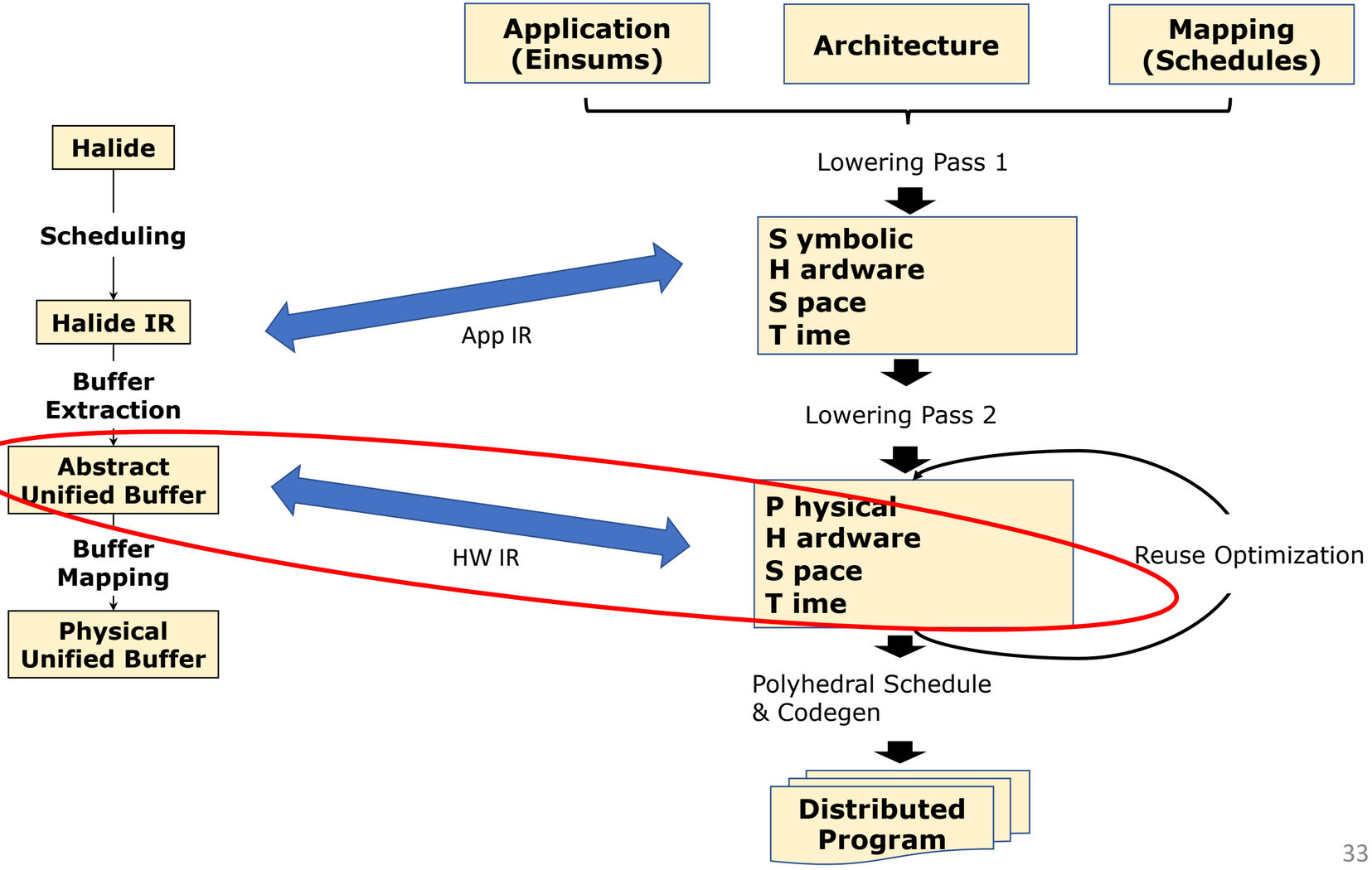
```
brighten(x, y) = input(x, y) * 2;
blur(x, y) = (brighten(x, y  ) + brighten(x+1, y  ) +
              brighten(x, y+1) + brighten(x+1, y+1))/4;
blur.tile(x, y, xo, yo, xi, yi, 63, 63)
    .hw_accelerate(xi, xo);
brighten.store_at(blur, xo)
        .compute_at(blur, xo);
input.stream_to_accelerator();
```

```
for (y, 0, 64)
 for (x, 0, 64)
  brighten(x, y) = input(x, y) * 2;

for (y, 0, 63)
 for (x, 0, 63)
  blur(x, y) = (brighten(x, y) + brighten(x+1, y) +
                brighten(x, y+1) + brighten(x+1, y+1))/4;
```

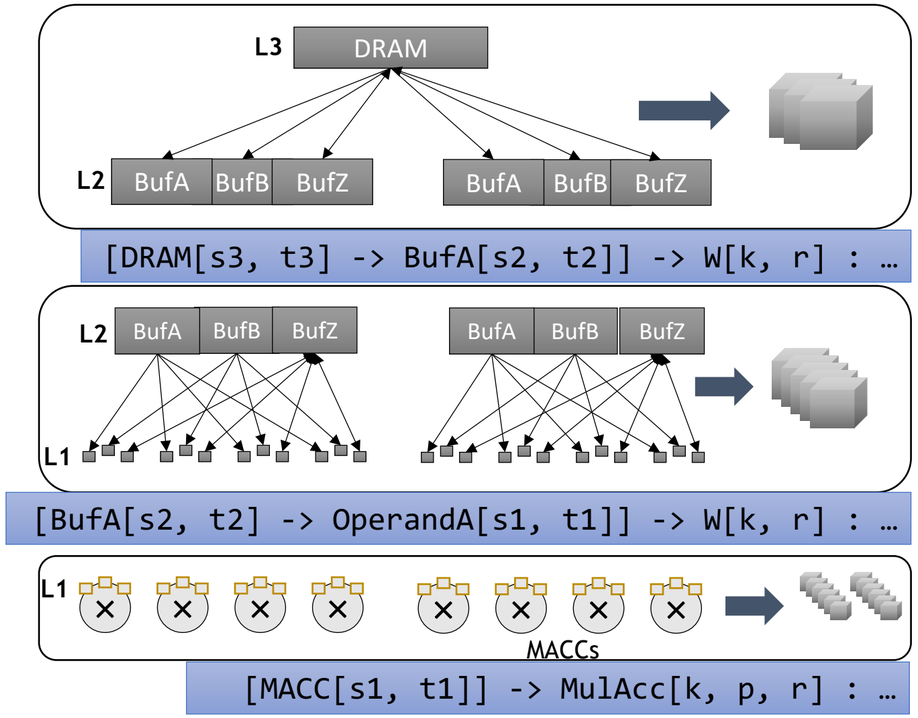


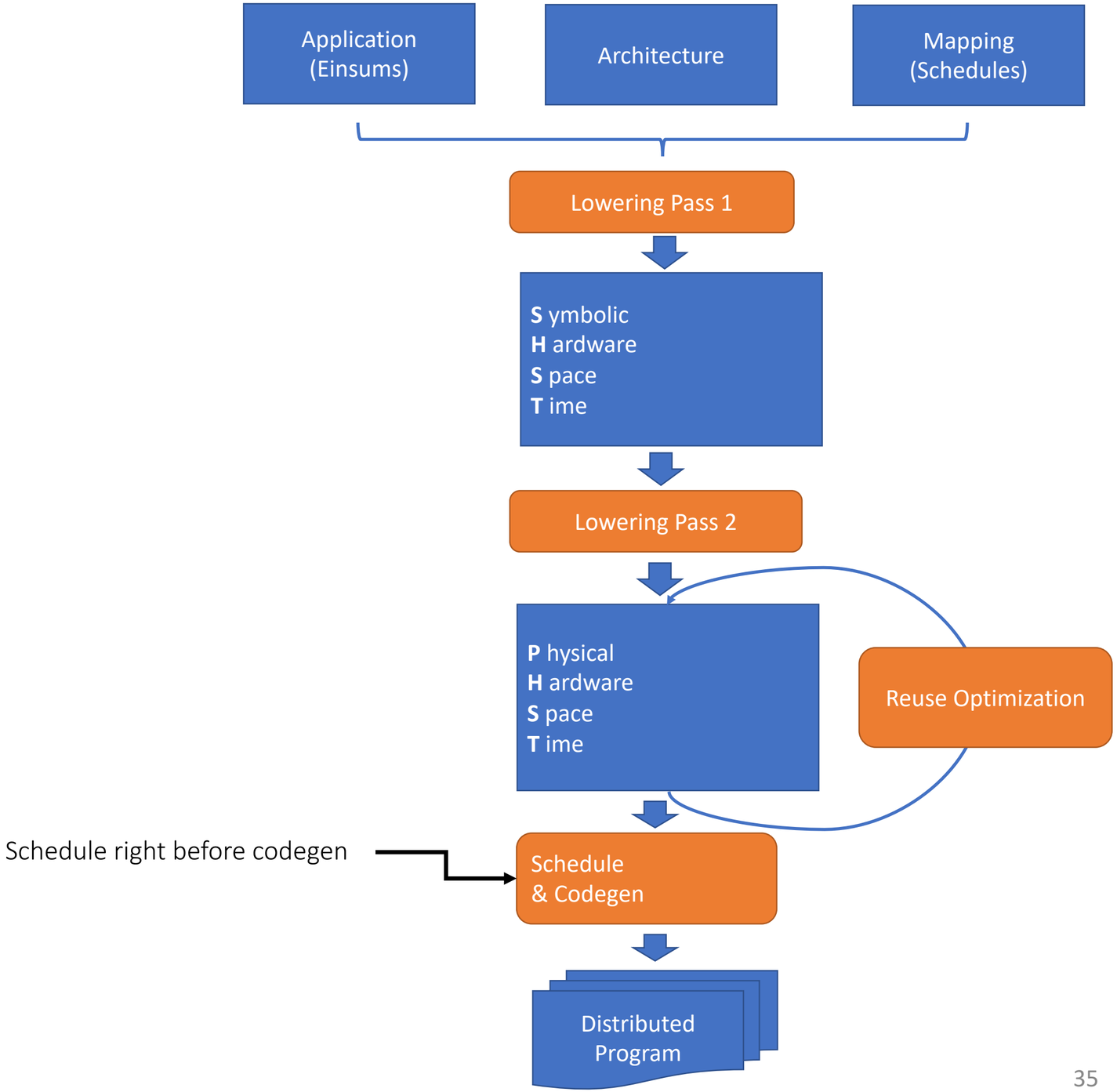




Look into the PHST, it’s pretty similar to UB

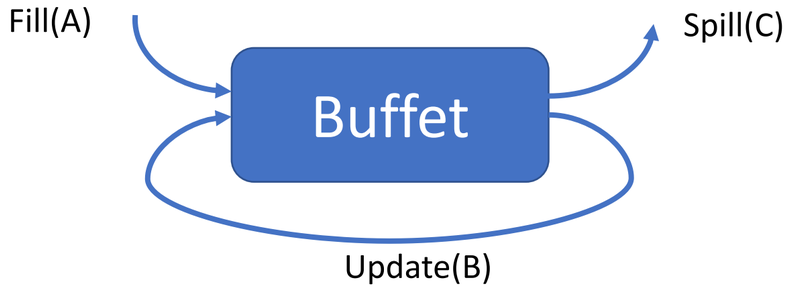
- UB(Memory Tile Instruction)
  - OP:
    - Write
    - Read
  - Access map:
    - IterDom(i0,i1,…)->Tensor Coordinate(Address)
  - Schedule:
    - IterDom(i0, i1…) -> Time
- PHST(Buffet Instruction)
  - OP:
    - Fill
    - Read_with_update / Read_with_drain
  - Access map:
    - IterDom(s,t)->Tensor Coordinate(Address)




Application
(Einsums)
Architecture
Mapping
(Schedules)
Lowering Pass 1
Symbolic
Hardware
Space
Time
Lowering Pass 2
Physical
Hardware
Space
Time
Reuse Optimization
Schedule right before codegen
Schedule
& Codegen
Distributed
Program

Ubuffer vs PHST: synchronization

- PHST: Buffet Operation
  - A: Fill
  - B: Read_with_update
  - C: Read_with_ drain(Spill)



- Buffet synchronization
  - RAW dependency checking
  - A->B A->C
- UBuffer: Pure sw control synchronization

- Local Schedule is enough due to HW synchronization
  - Schedule is created between all stmts within one physical hardware
  - Schedule is created after mapping is done

Scheduler Discussion

- Global schedule is useful
  - More capable compiler can target more general hardware
    - UB->Buffet: √
    - PolyEDDO->CGRA Mem: ? non-trivial
  - Map to ready-valid hardware just take the partial schedule
  - Dillon's thesis: help eval the FIFO size prevent deadlock
- Fine-grained cycle accurate schedule for all stmts may be overkilled?
  - For coarse grained data transfer, a time stamp for the whole block
    - Token per op/token per tile
    - Loop order is undefined at app IR level
    - Reorder data transfer loop

Conclusion

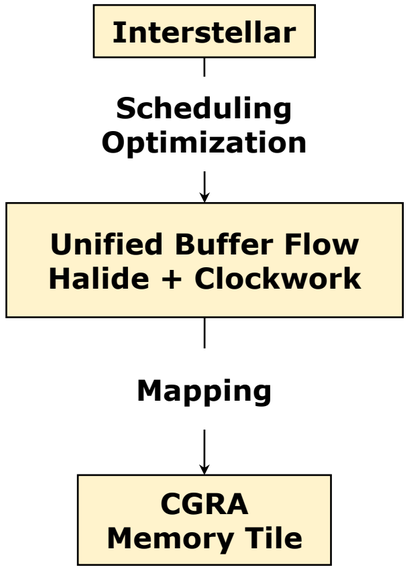
- The compiler IR is tailored for
  - Target application behavior
  - Memory hardware architecture
- SHST/Halide IR loopnest has equivalent capability
  - SHST is helpful to formalize parallelization reuse analysis
  - Halide IR is more flexible to represent data transfer
- HW Binding is a hint for scheduling
  - Simplify the mapping problem
- Scheduling:
  - Global schedule is useful to target a wider variety of applications & memory
  - Fine-grained schedule could be optimized

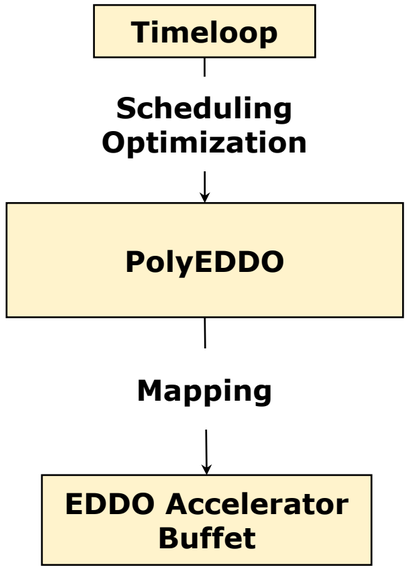
Backup

Further Discussion

- Lazy evaluation:
  - Local schedule vs. global schedule
    - Schedule is created before map stmts to hardware
    - Schedule is created between all stmts related to one physical hardware
  - Loop order:
    - Do we need to have a fine-grained cycle accurate schedule for each stmt under a loopnest or we just need a coarse grained timestamp
  - Architecture as an input
    - CGRA has more flexible interconnect
    - UB is agnostic about it and we offload that job to the PnR tool

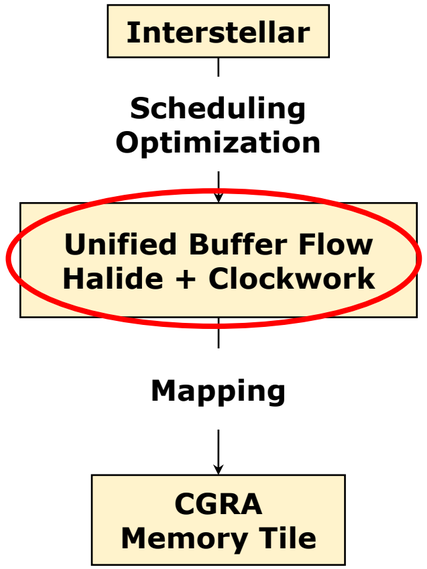
Application compilation tool chain

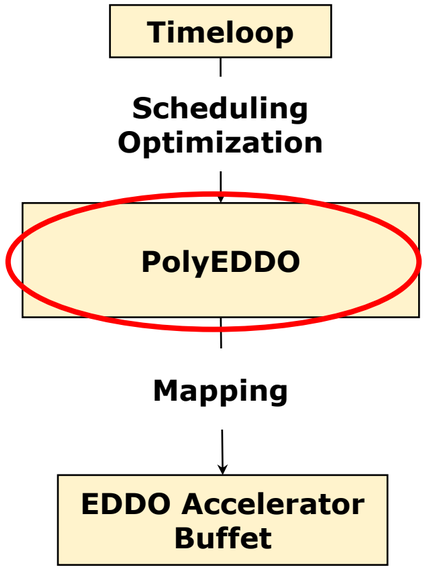




Focus on the mapping stage

Taking schedule generated from optimizer and compiling to program/configuration running on HW





How memory transfer is different

- IterDom Tile size -> data tile size fetched from outer memory
  - Restricted the data must be fetched from memory that is one level above.
  - Analogy: always create a cpy stmt between two neared memory.
- Similar logic can be applied to different memory level
  - Reuse is calculated between different space/time coordinate which result in temporal or spatial sharing
- Explicitly define the memory transfer as a copy stmt
  - Automatically explore reuse or banking for enough bandwidth