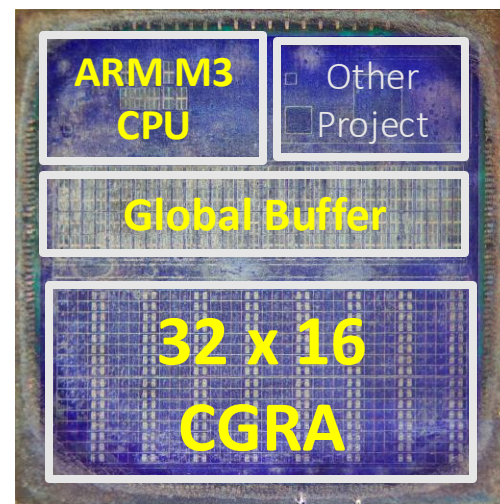


Custard and The Sparse Abstract Machine: Compiling Sparse Applications to Coarse- Grained Reconfigurable Arrays

Olivia Hsu

Dataflow hardware can accelerate sparse tensor algebra

$$\begin{array}{l}
 a = Bc + a \quad a = Bc \quad A = B + C \\
 a = B^T c + d \quad a = B^T c \quad A = \alpha B \quad a = Bc + b \\
 a = B^T c \quad a = b \odot c \quad a = B(c + d) \\
 A = B + C + D \quad A = BC \quad A = B \odot (CD) \\
 A = B \odot C \quad A = 0 \quad A = BCd \quad A = B^T Bc \\
 a = b + c \quad A = B \quad K = A^T c \quad \beta a \\
 A_{ij} = \sum_{kl} B_{ikl} C_{lj} D_{jk} \quad A_{ij} = \sum_{kl} B_{ikl} C_{lj} D_{jk} \\
 A_{ij} = \sum_{kl} B_{ikl} C_{lj} D_{jk} \quad A_{ij} = \sum_k B_{ijk} C_k \\
 A_{ij} = \sum_k B_{ijk} C_k \quad A_{ij} = \sum_k B_{ijk} C_k \\
 A_{ij} = \sum_k B_{ijk} C_k \quad A_{ijl} = \sum_k B_{ikl} C_{kj} \\
 A_{ij} = \sum_k B_{ijk} C_k \quad A_{ijl} = \sum_k B_{ikl} C_{kj} \\
 C = \sum_{ijkl} M_{ij} P_{jk} \overline{M}_{lk} \overline{P}_{il} \quad \tau = \sum_i z_i (\sum_j z_j \theta_{ij}) (\sum_k z_k \theta_{ik}) \\
 a = \sum_{ijklmnop} M_{ij} P_{jk} M_{kl} P_{lm} \overline{M}_{nm} \overline{P}_{no} \overline{P}_{po} \overline{P}_{ip}
 \end{array}$$



Onyx CGRA

[Koul et al. VLSI, HotChips 2024]

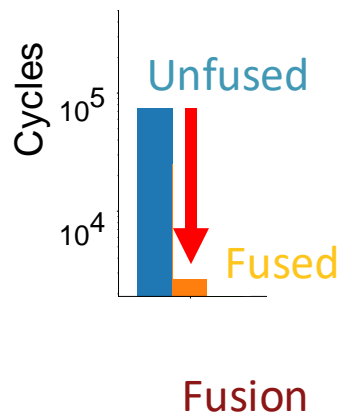
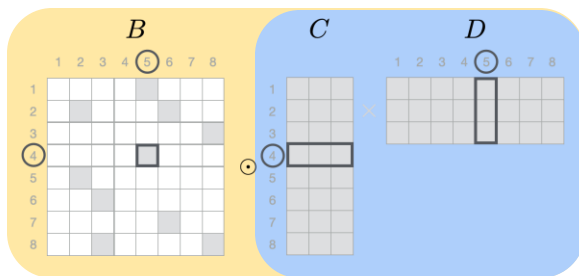
but really any sparse accelerator...

Need Generality to Handle This...

Performance requires generality in schedules

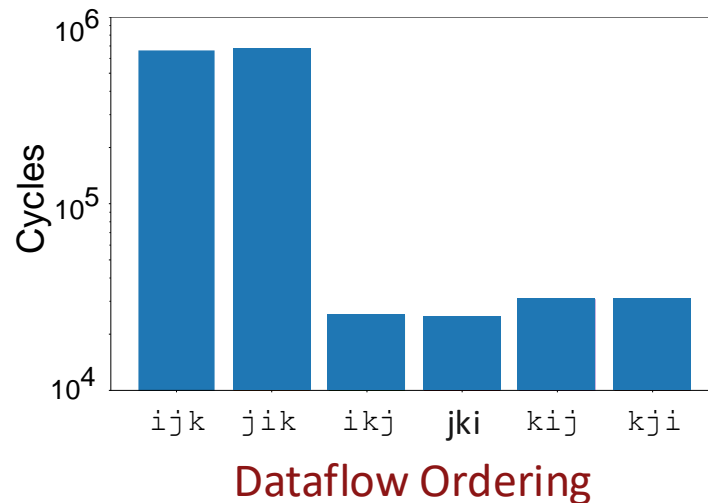
$$A = B \odot (CD)$$

SDDMM



$$X_{ij} = B_{ik} \cdot C_{kj}$$

SpMSpM



Efficient mapping requires a compiler

All of Sparse Tensor Algebra

$$\begin{aligned}
 &a = Bc + a \quad a = Bc \quad A = \dots \quad \text{Algebra} \\
 &a = B^T c + d \quad a = B^T c \quad A = \dots \quad a = B(c + d) \\
 &A = B + C \quad A = B \odot C \quad A = B \odot (CD) \\
 &A = B \odot C \quad A = BCD \quad A = B^T \quad a = B^T Bc \\
 &K = A^T C A \quad a = \alpha Bc + \beta a \\
 &A_{ij} = \sum_{kl} B_{ikl} C_{lj} D_{kj} \quad A_{kj} = \sum_{il} B_{ikl} C_{lj} D_{ij} \\
 &A_{lj} = \sum_{ik} B_{ikl} C_{ij} D_{kj} \quad A_{ij} = \sum_k B_{ijk} c_k \\
 &A_{ijk} = \sum_l B_{ikl} C_{lj} \quad A_{ik} = \sum_j B_{ijk} c_j \quad \text{Data analytics (tensor factorization)} \\
 &A_{jk} = \sum_i B_{ijk} c_i \quad A_{ijl} = \sum_k B_{ikl} C_{kj} \\
 &C = \sum_{ijkl} M_{ij} P_{jk} \overline{M_{lk}} \overline{P_{il}} \quad \tau = \sum_i z_i \left(\sum_j z_j \theta_{ij} \right) \left(\sum_k z_k \theta_{ik} \right) \\
 &a = \sum_{ijklmnop} M_{ij} P_{jk} M_{kl} P_{lm} \overline{M_{nm}} \overline{P_{no}} \overline{M_{po}} \overline{P_{ip}} \quad \text{Quantum Chromodynamics}
 \end{aligned}$$

Algorithm (expression)

×

reorder
precompute
parallelize split
map divide
vectorize unroll
position

Schedule



Lends itself to multiple backends

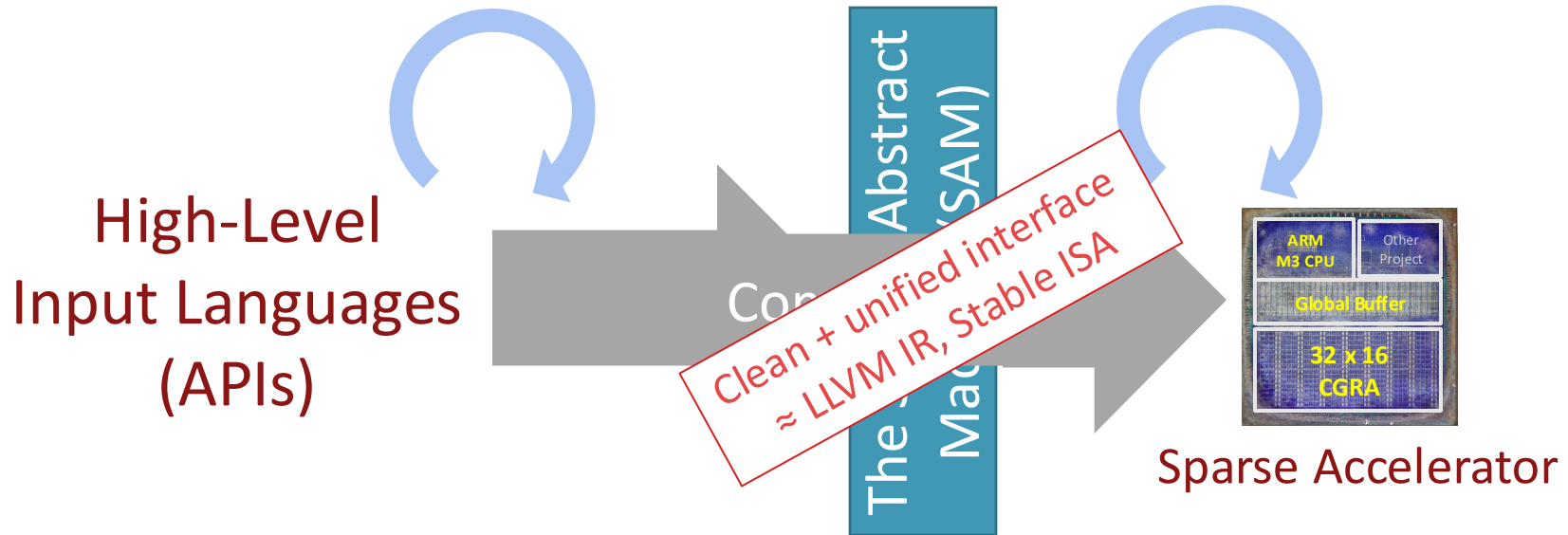
reorder
precompute
parallelize split
map divide
vectorize unroll
position

MatRaptor
Sigma
Onyx
Sparse TPU
Eyeriss V2
SCNN
SpaceA
UCNN
Spada
Fifer
Sparse CGRAs
SPU
ExTensor
Capstan
... and future

Backend

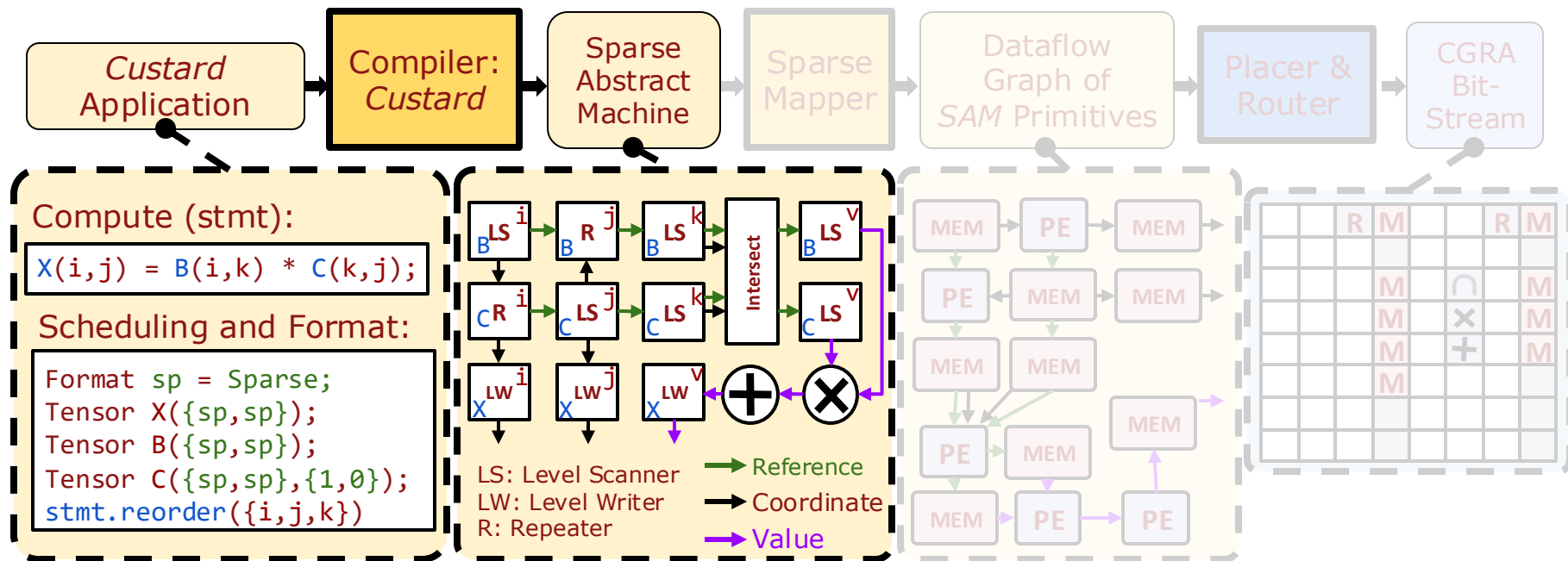
Programming dataflow requires an abstract machine that a compiler can target

[Hsu et al. ASPLOS 2023]



A clean interface decouples the compiler from specific hardware implementations

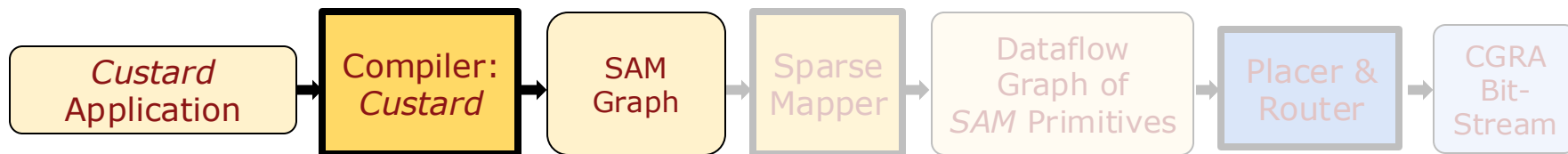
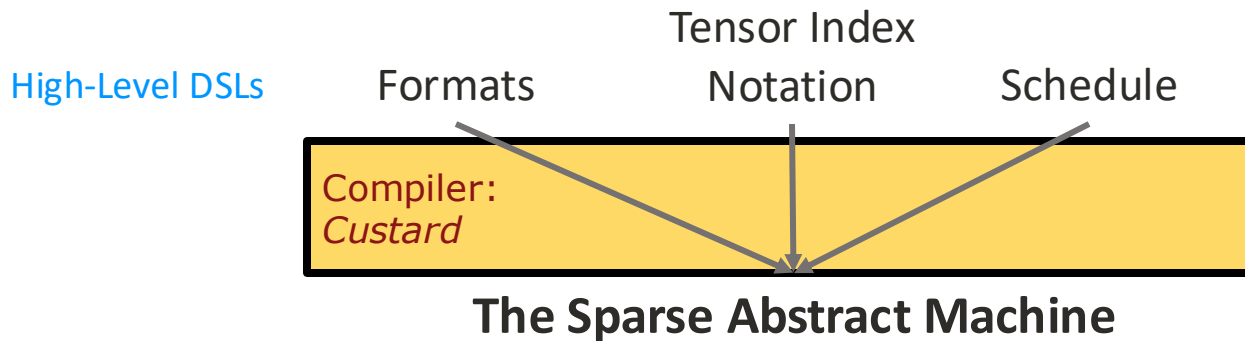
This work in the DSL-based CGRA flow



SAM leverages domain-specific languages and comes with the Custard compiler

Custard Application

```
Format sp = Sparse;  
Tensor X({sp,sp});  
Tensor B({sp,sp});  
Tensor C({sp,sp},{1,0});  
  
X(i,j) = B(i,k) * C(k,j); stmt.reorder({i,j,k});
```



Representing dataflow in SAM

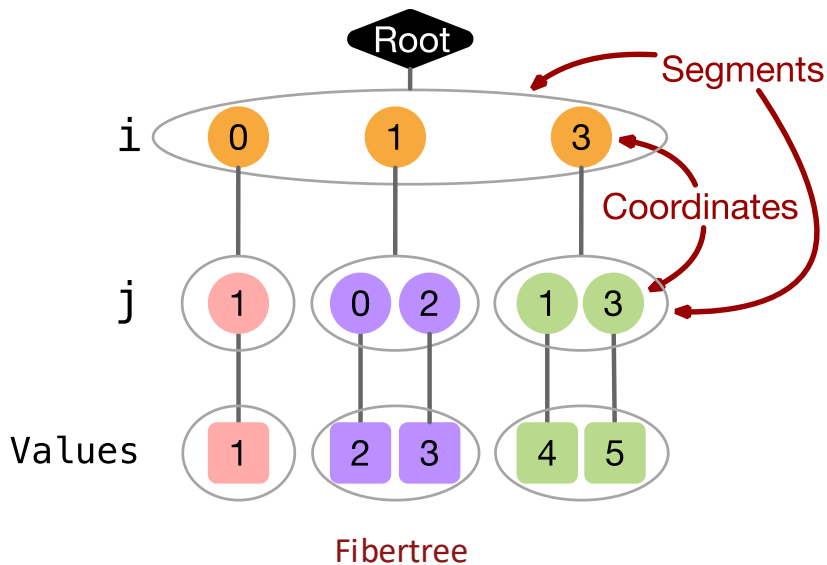
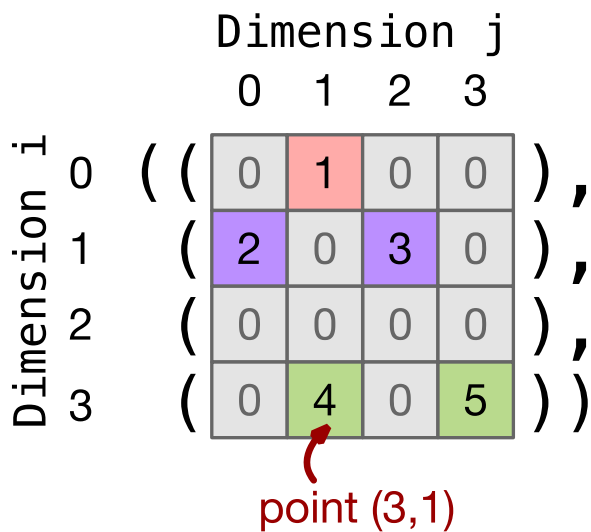
SAM represents:

1. Wires carrying data through streams
2. Modules that compute on the data through primitives



Representing tensors in SAM

SAM
Graph

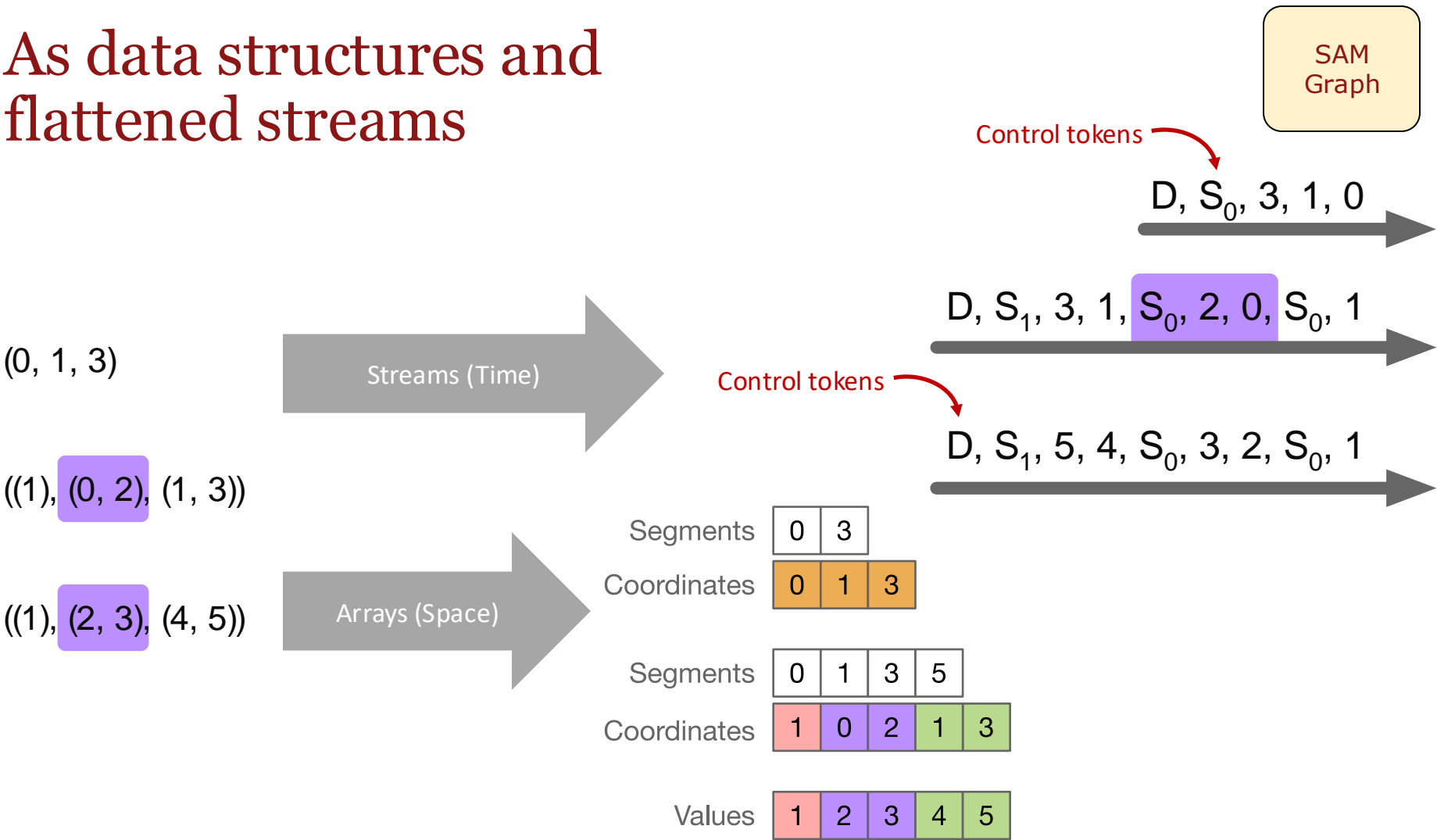


(0, 1, 3)

((1), (0, 2), (1, 3))

((1), (2, 3), (4, 5))

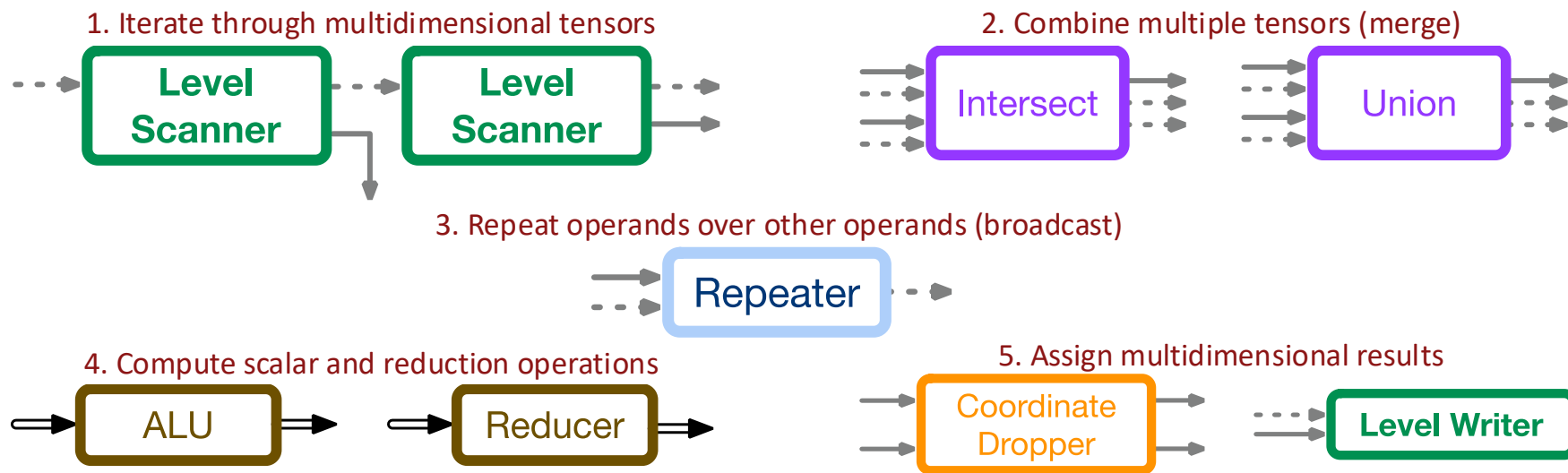
As data structures and flattened streams



SAM supports all of tensor algebra

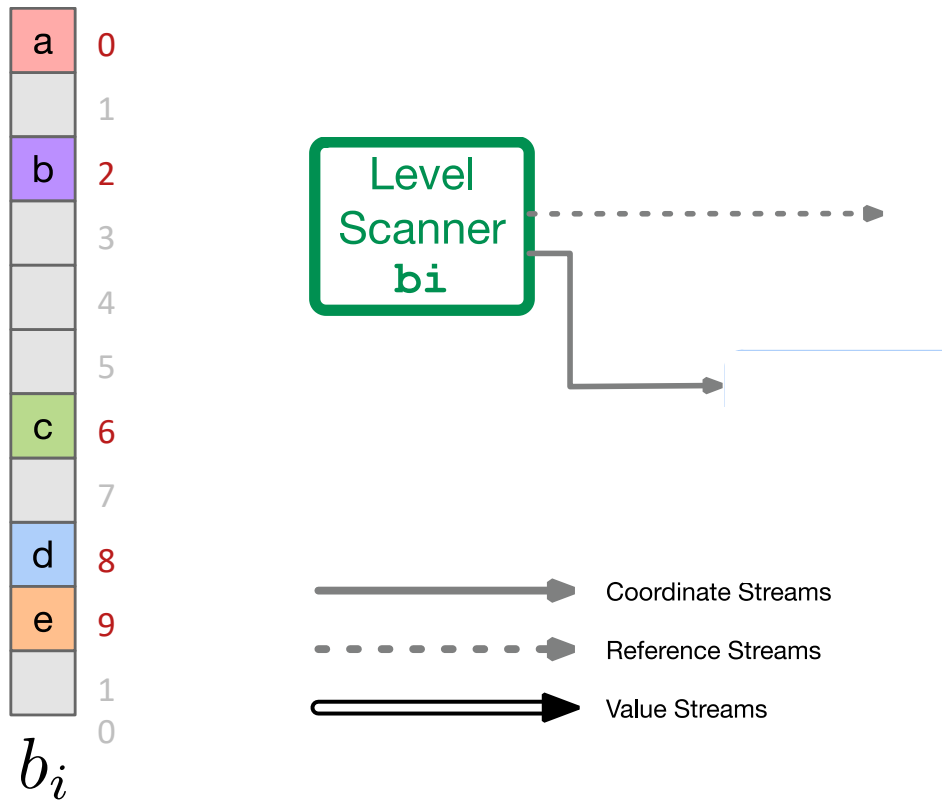
SAM
Graph

The sparse abstract machine has clean interfaces defined for each feature of sparse tensor algebra, called primitives



Vector scaling example with repeaters

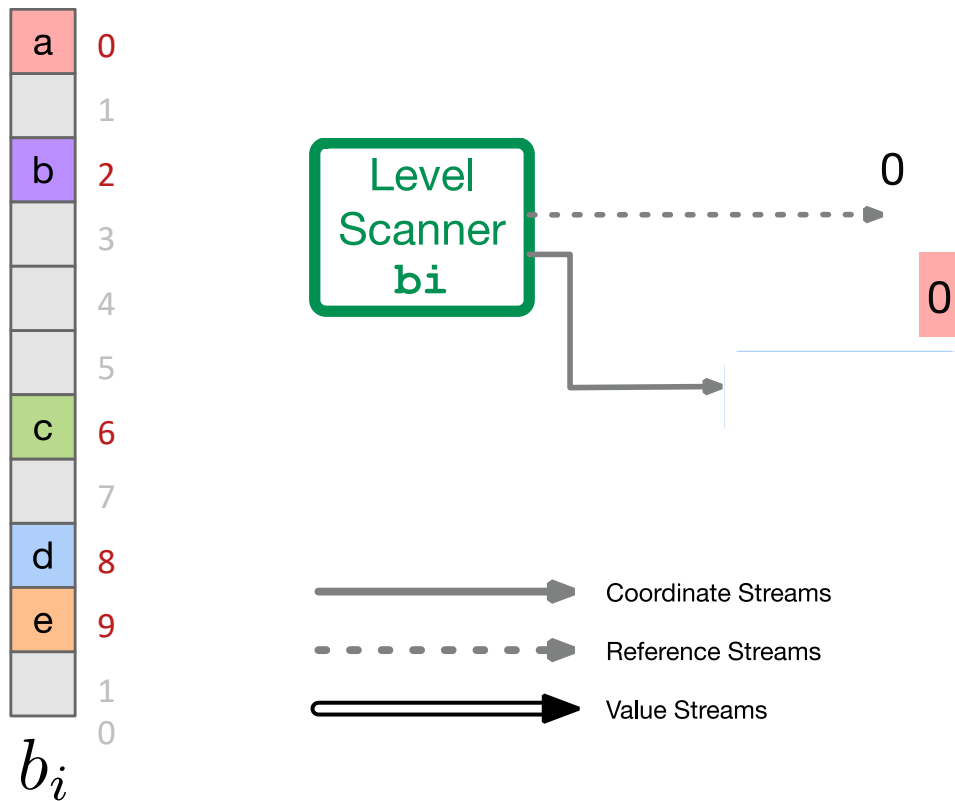
$$x_i = \underline{b_i} \cdot c$$



Vector scaling example with repeaters

SAM
Graph

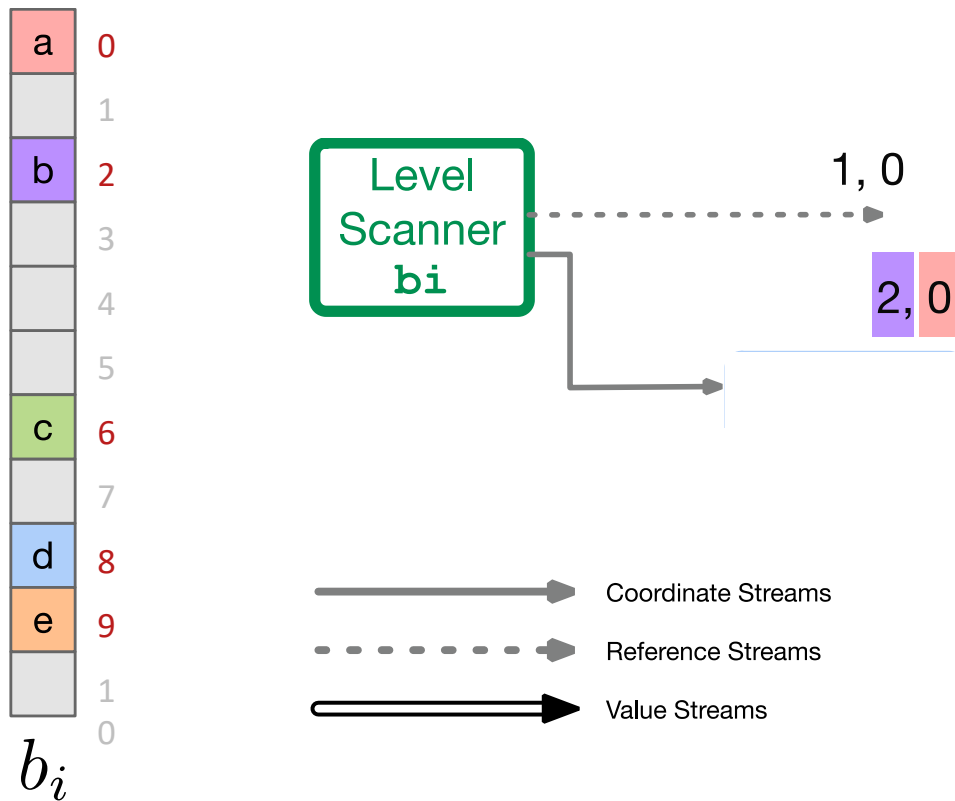
$$x_i = \underline{b_i} \cdot c$$



Vector scaling example with repeaters

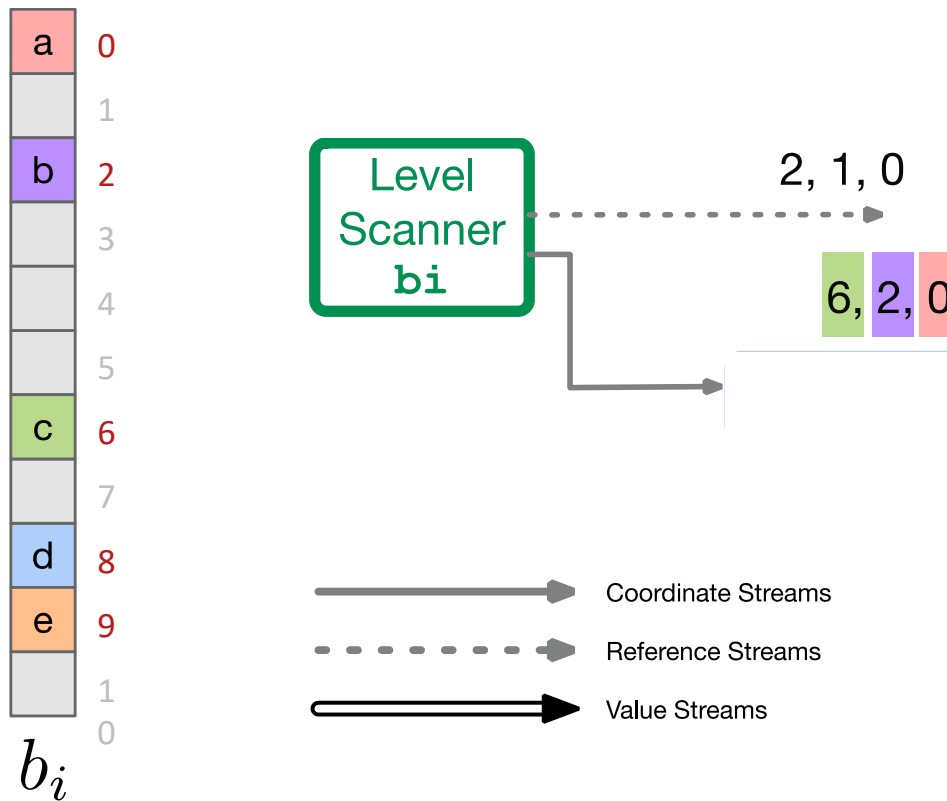
SAM
Graph

$$x_i = \underline{b_i} \cdot c$$



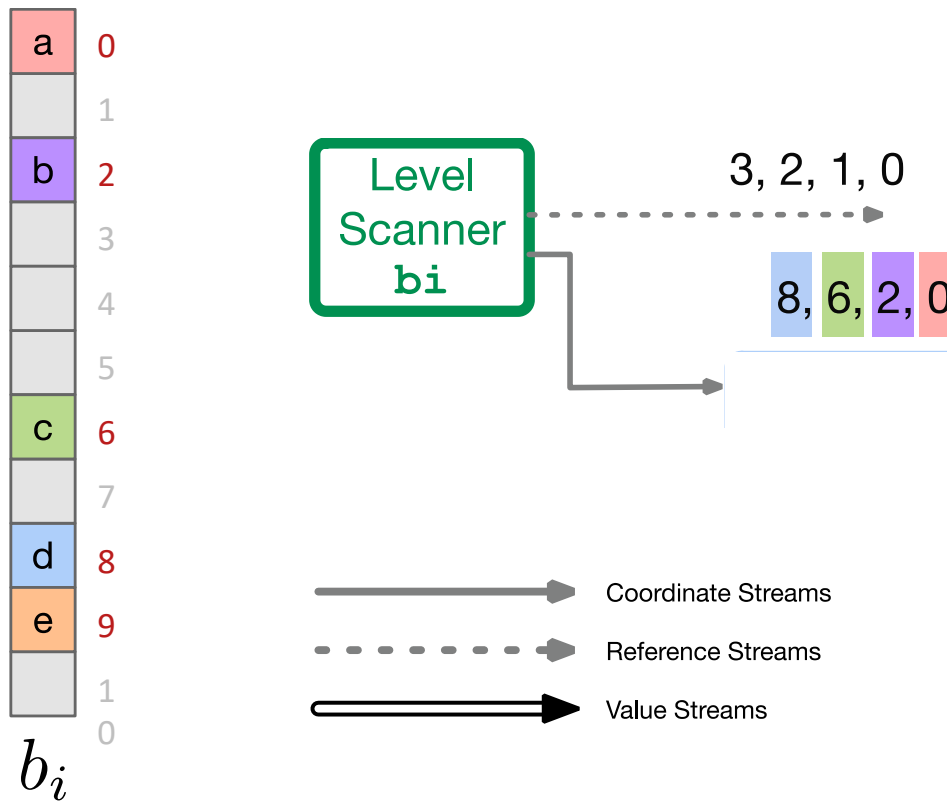
Vector scaling example with repeaters

$$x_i = \underline{b_i} \cdot c$$



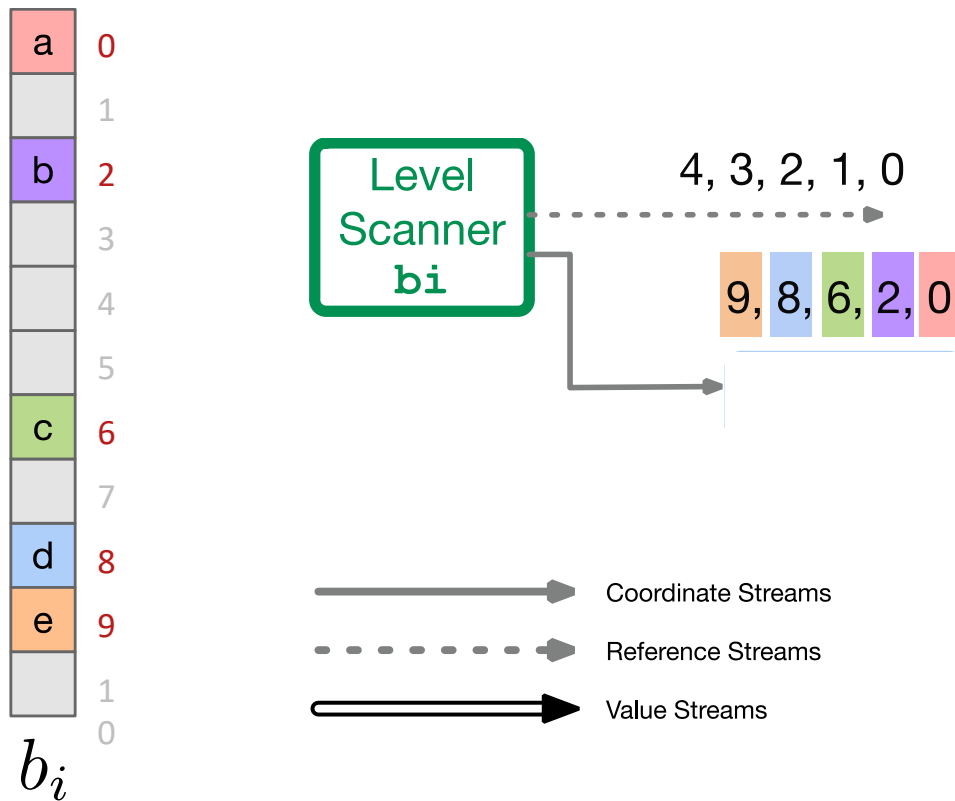
Vector scaling example with repeaters

$$x_i = \underline{b_i} \cdot c$$



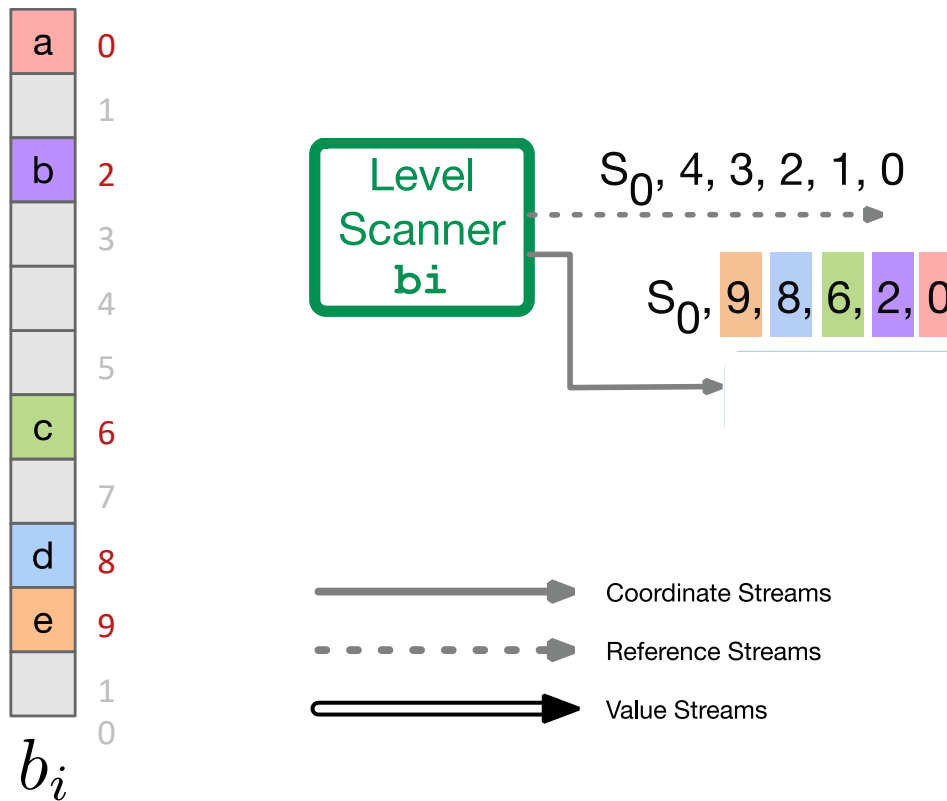
Vector scaling example with repeaters

$$x_i = \underline{b_i} \cdot c$$



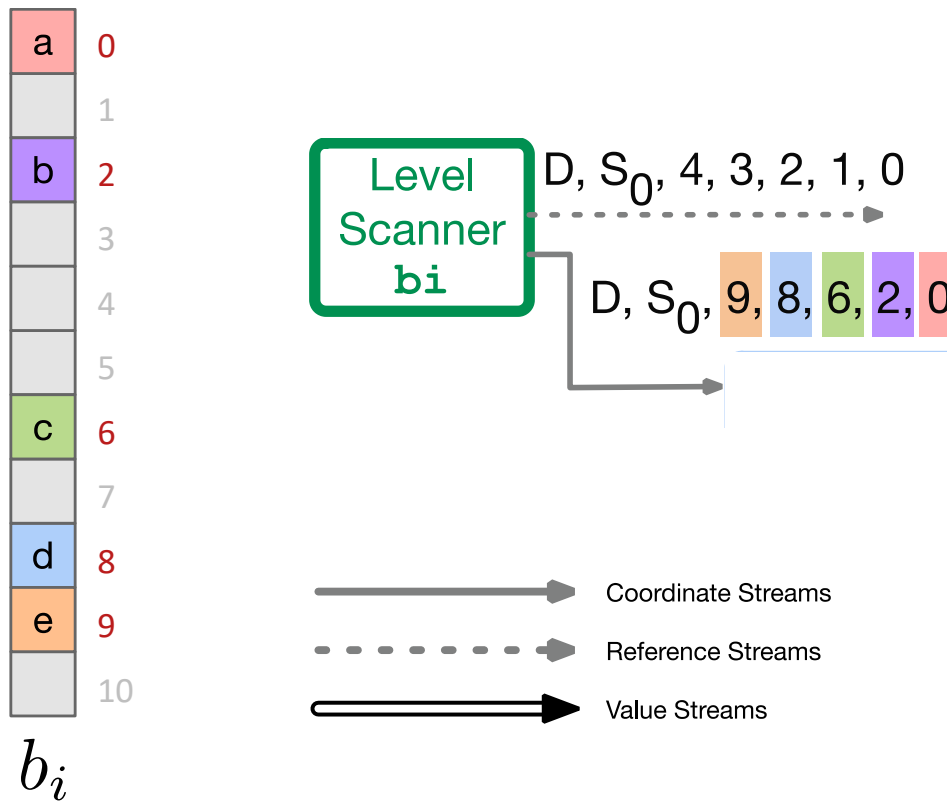
Vector scaling example with repeaters

$$x_i = \underline{b_i} \cdot c$$



Vector scaling example with repeaters

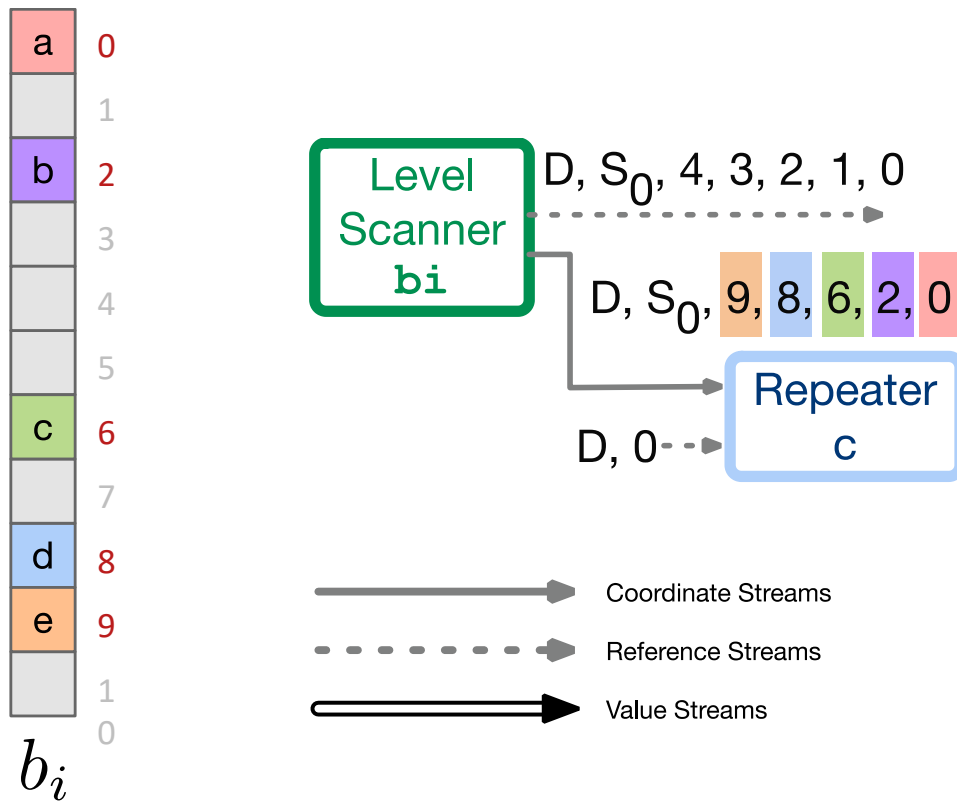
$$x_i = \underline{b_i} \cdot c$$



Vector scaling example with repeaters

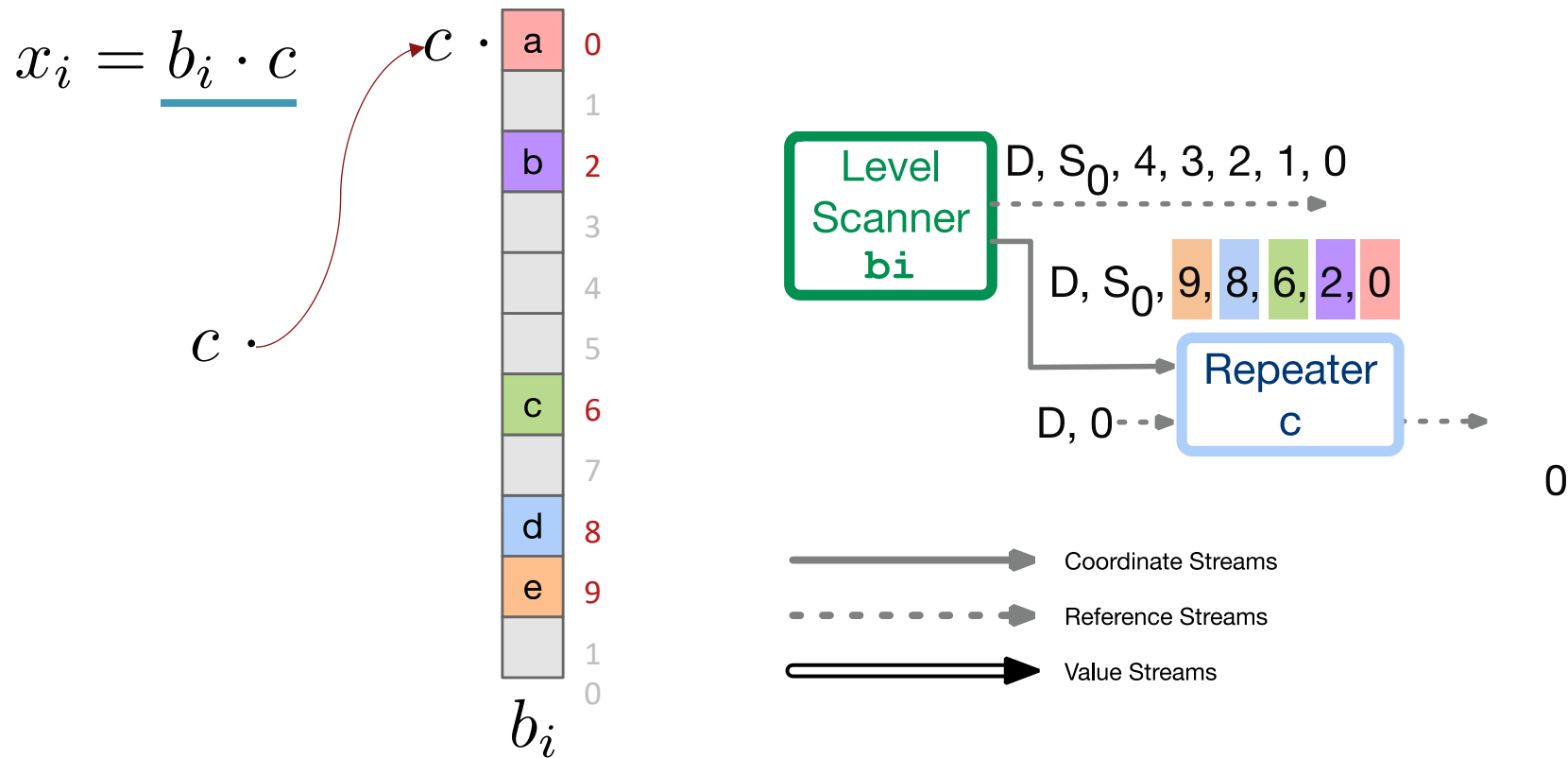
$$x_i = \underline{b_i \cdot c}$$

$c \cdot$



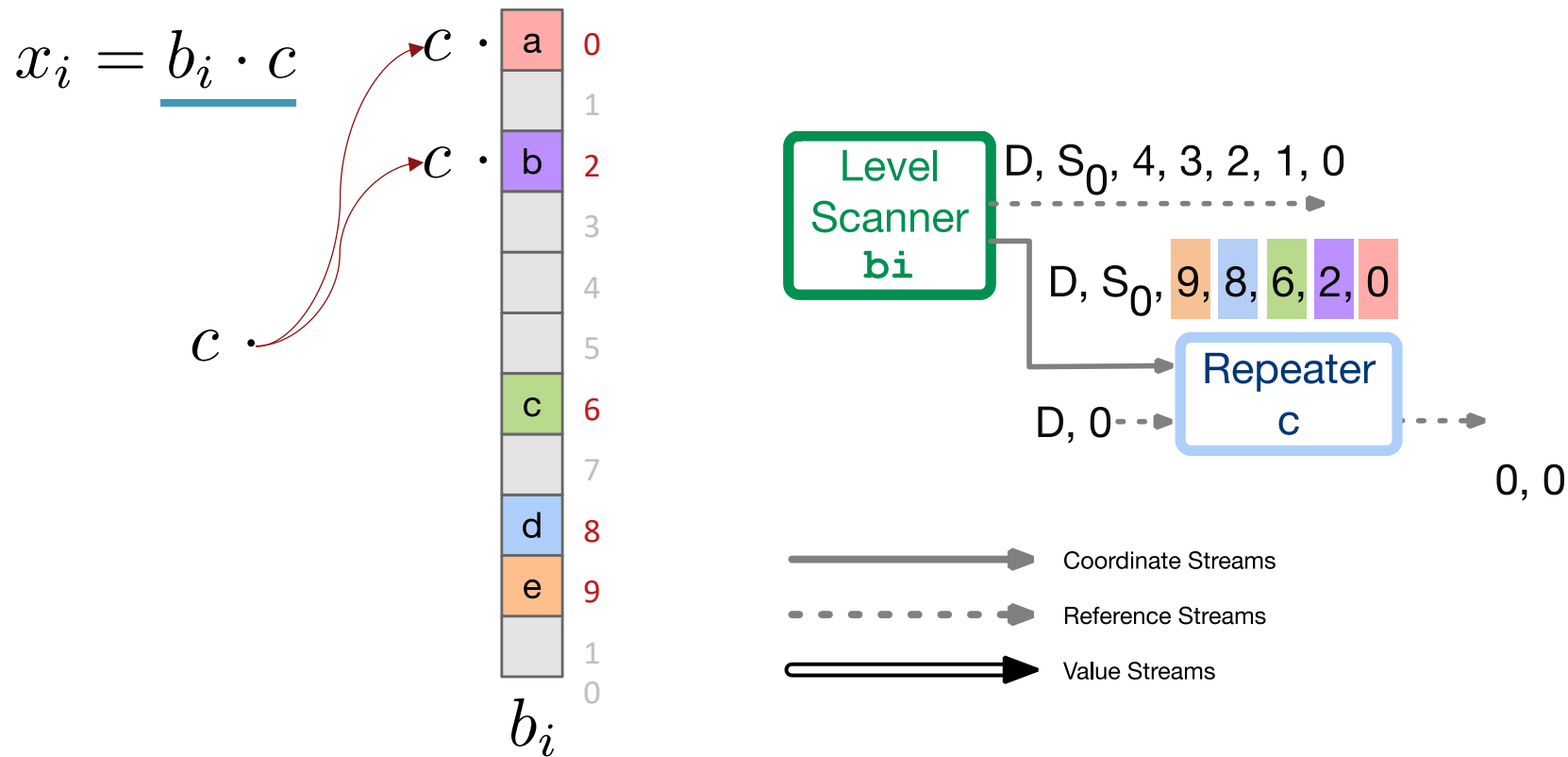
Vector scaling example with repeaters

SAM
Graph



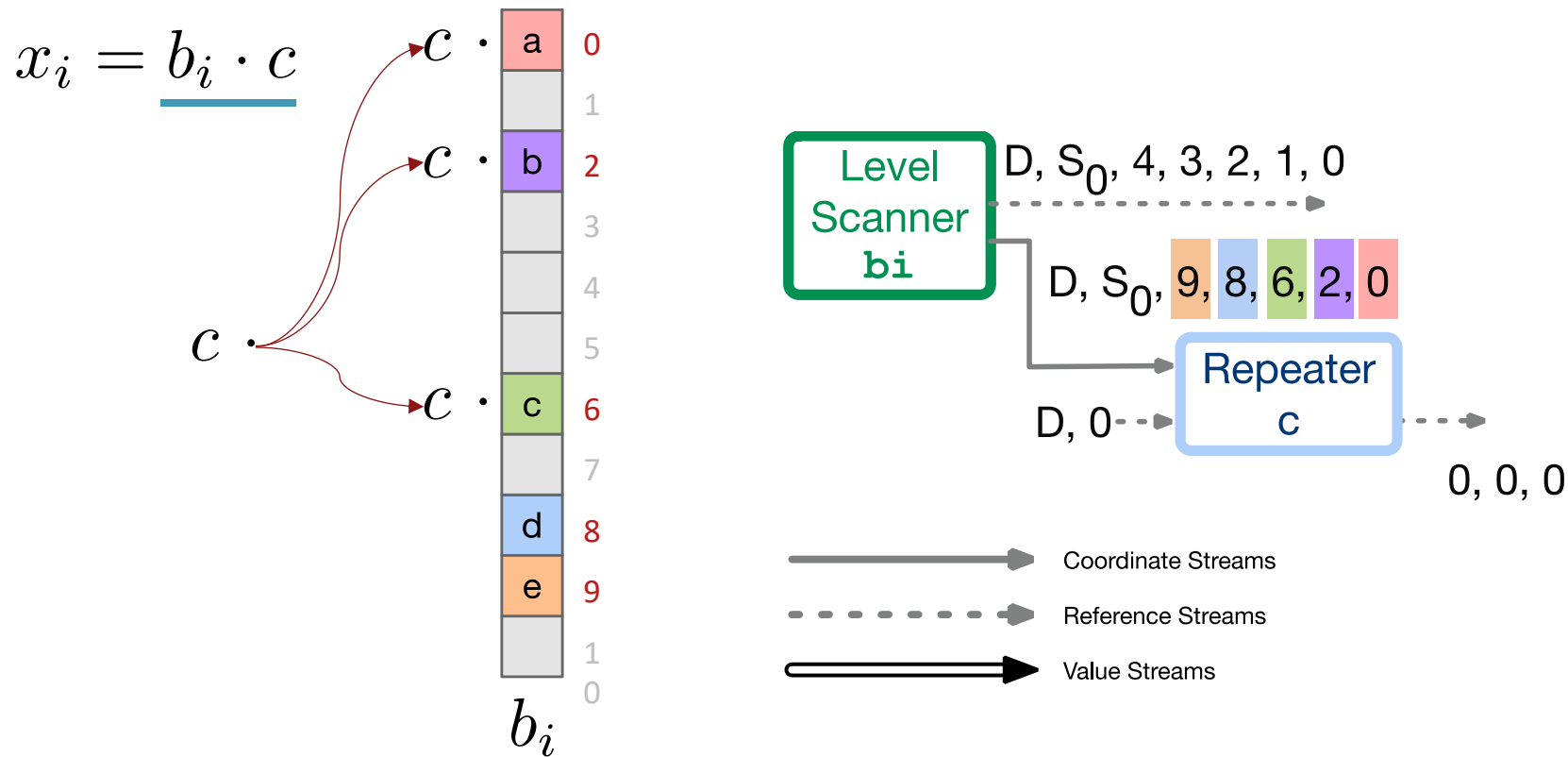
Vector scaling example with repeaters

SAM
Graph



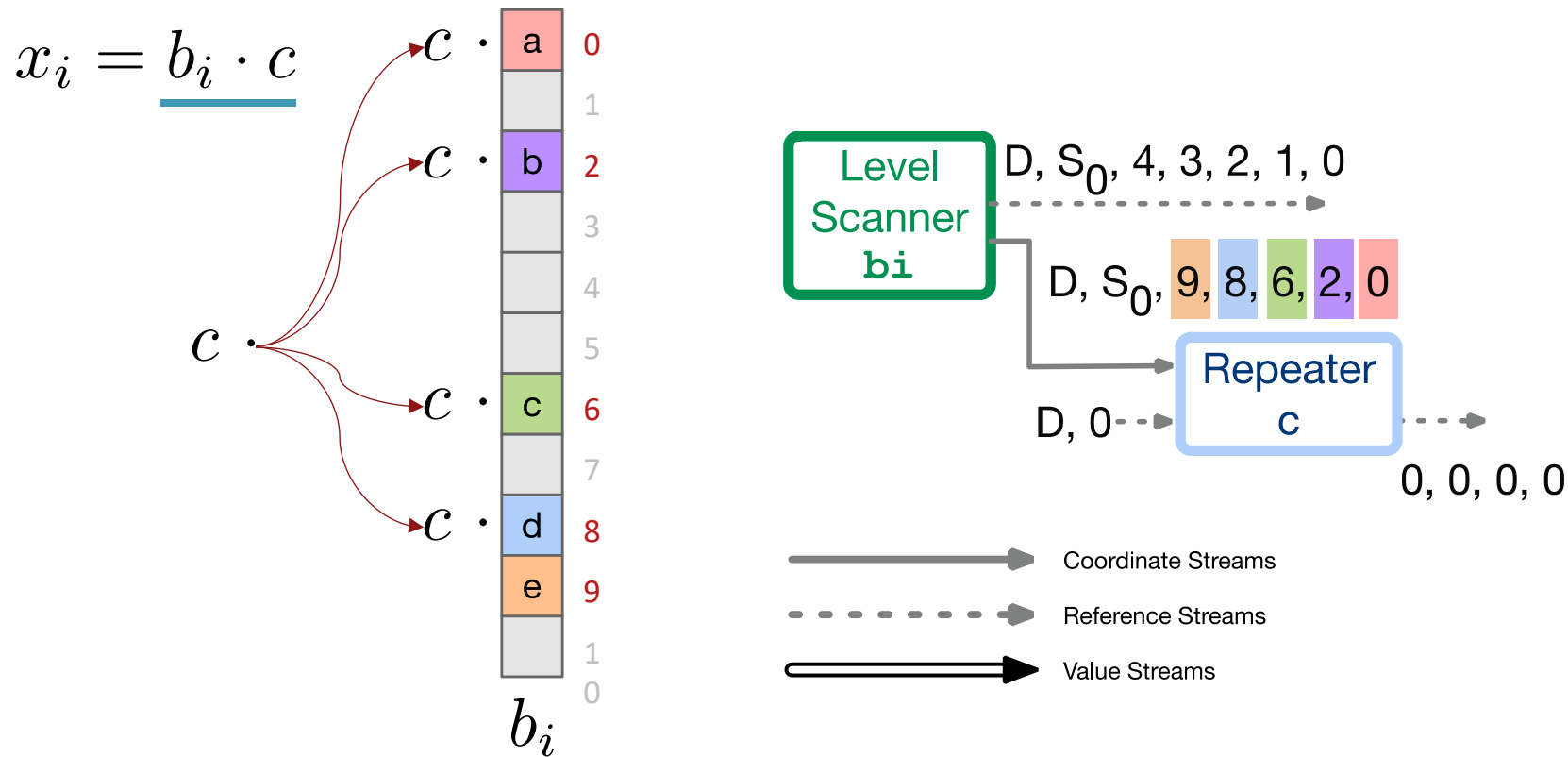
Vector scaling example with repeaters

SAM
Graph



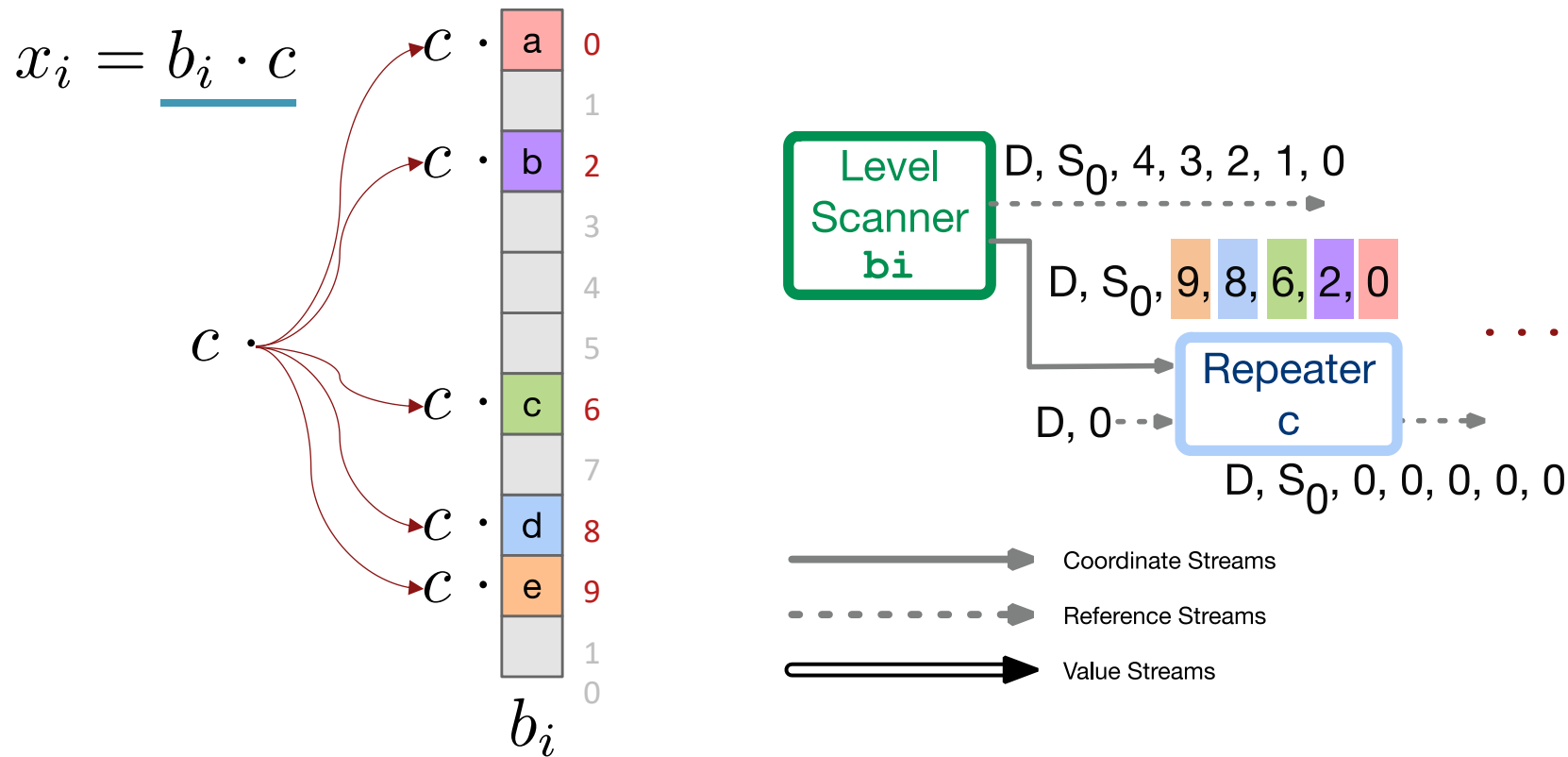
Vector scaling example with repeaters

SAM
Graph

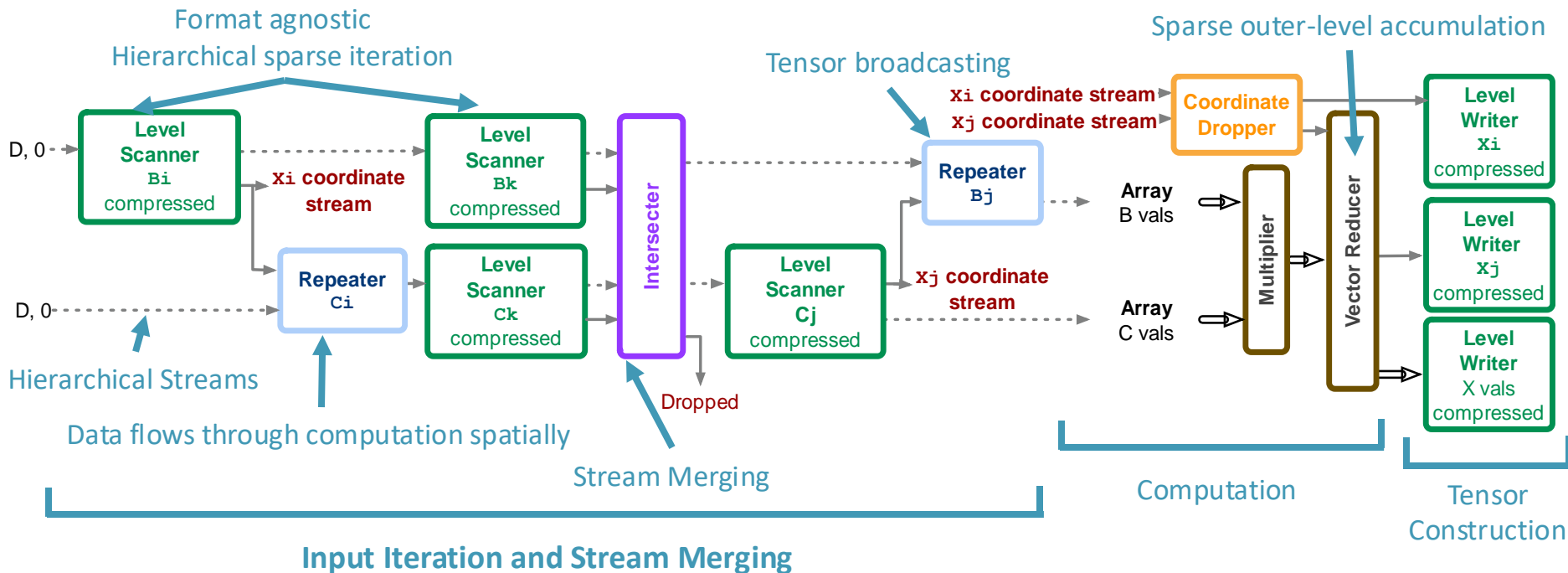


Vector scaling example with repeaters

SAM
Graph

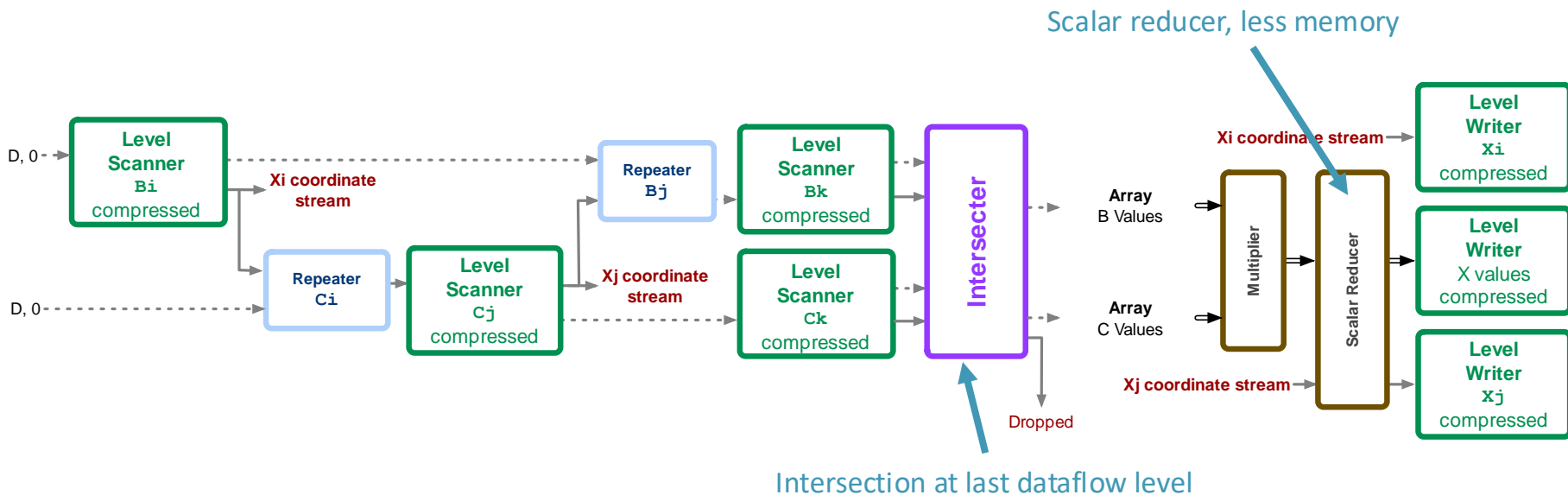


Primitives compose to compute expressions: SpM*SpM



$$\forall_i \forall_k \forall_j X_{ij} = B_{ik} \cdot C_{kj}$$

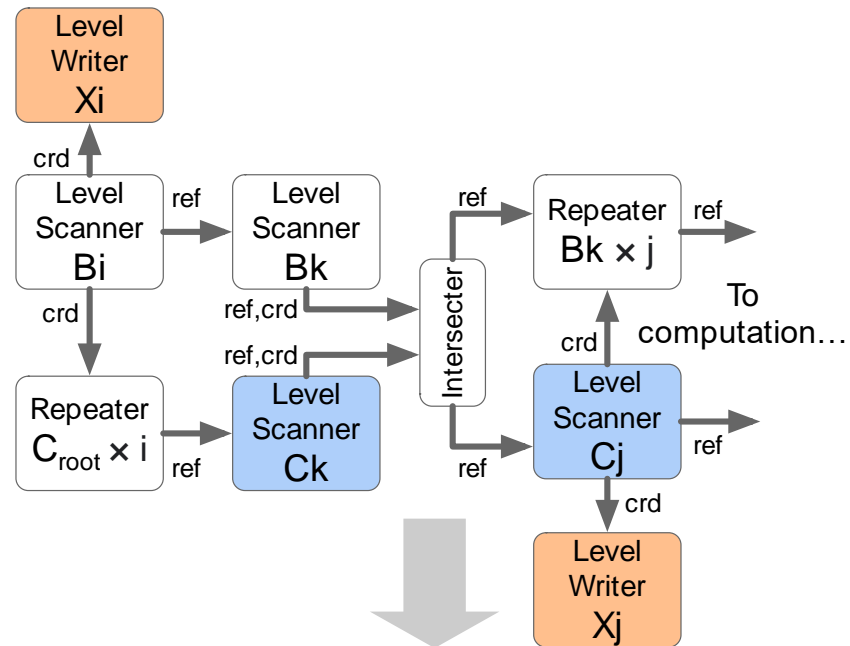
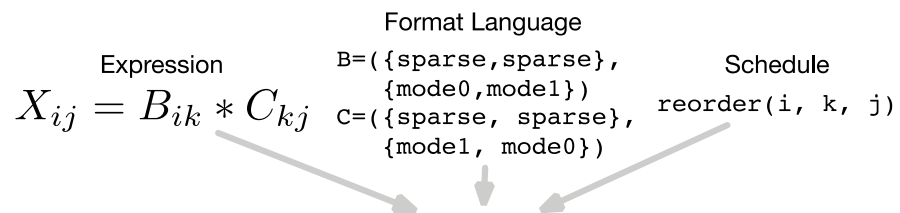
Inner-product algorithm in SAM



$$\forall_i \forall_j \forall_k X_{ij} = B_{ik} * C_{kj}$$

Custard's compiler algorithm to SAM

Compiler:
Custard

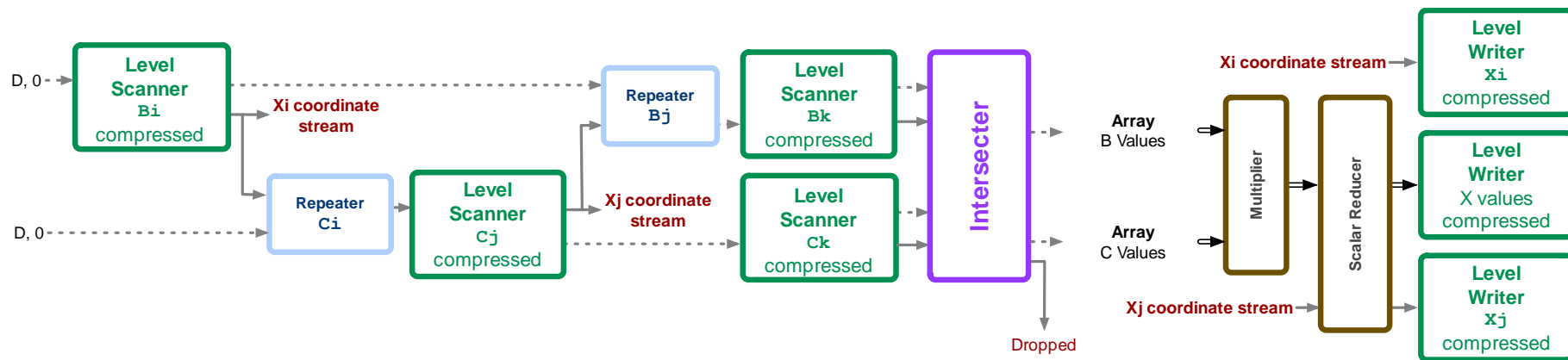


Demo: Generating SAM graphs with Custard

```
> ./sparse_demo.sh compile
```

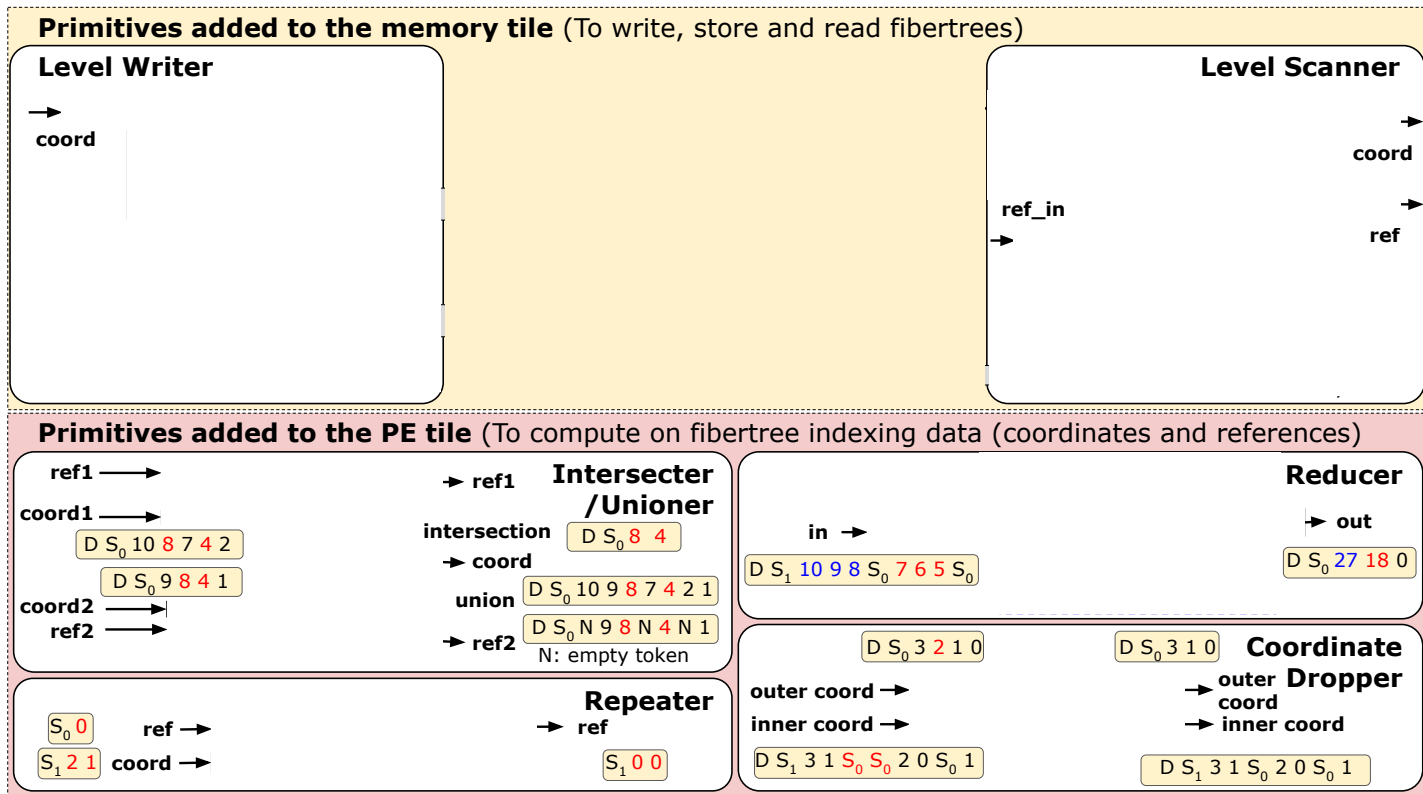
- This runs the applications in `./sam/compiler/sam-kernels.sh` through the Custard compiler
- All SAM graphs generated in `./sam/compiler/sam-outputs/`
- View the SpMSpM kernel `matmul_ijk` in `./sam/compiler/sam-outputs/png/matmul_ijk.png` in VSCode or using `docker cp`
- We will also view a smaller kernel, `mat_elemadd` in `./sam/compiler/sam-outputs/png/mat_elemadd.png`, which we will be using for the rest of the demo

Inner-product algorithm in SAM



$$\forall_i \forall_j \forall_k X_{ij} = B_{ik} * C_{kj}$$

SAM as the architectural specification of our sparse CGRA fabric



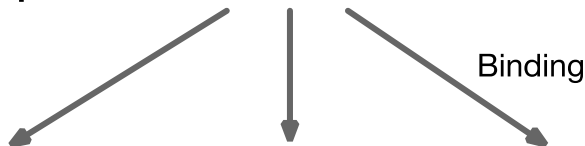
Demo: Mapping to CGRA Microarchitecture

```
> ./sparse_demo.sh lower
```

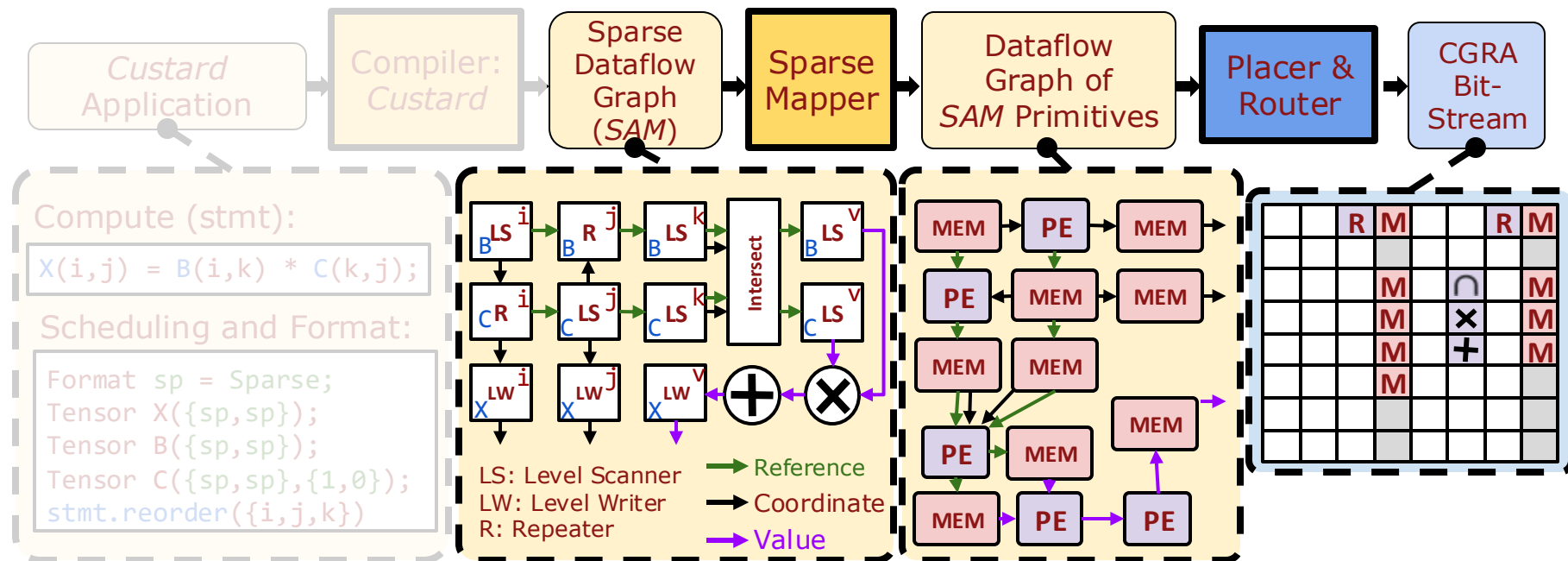
- This runs the SAM graph through the lowering process to produce a hardware-aware sparse dataflow graph
- We can visualize that graph in
`/aha/sam/hw_aware_mat_elemadd.png`

Although abstract, SAM binds to real hardware

Sparse Abstract Machine



Tool flow that maps SAM to a CGRA



Demo: Generating CGRA Bitstream for sparse applications

Run the following command:

```
> ./sparse_demo.sh gen
```

- This generates a CGRA bitstream from the hardware-aware graph using tools introduced later
- It also generates a testbench that runs an example matrix through, checking it with gold (written in Numpy)
- Explore output files generated in
`/aha/garnet/SPARSE_TESTS/mat_elemadd_0/`

- Dataflow hardware, like CGRAs, can speed up sparse computation
- Presented ideas from the Sparse Abstract Machine and Onyx
 - SAM is an abstract IR that represents sparse tensor algebra as dataflow graphs
 - SAM comes with a decoupled frontend compiler Custard
- Introduced the AHA flow for sparse applications

Conclusion