Infinity Stream

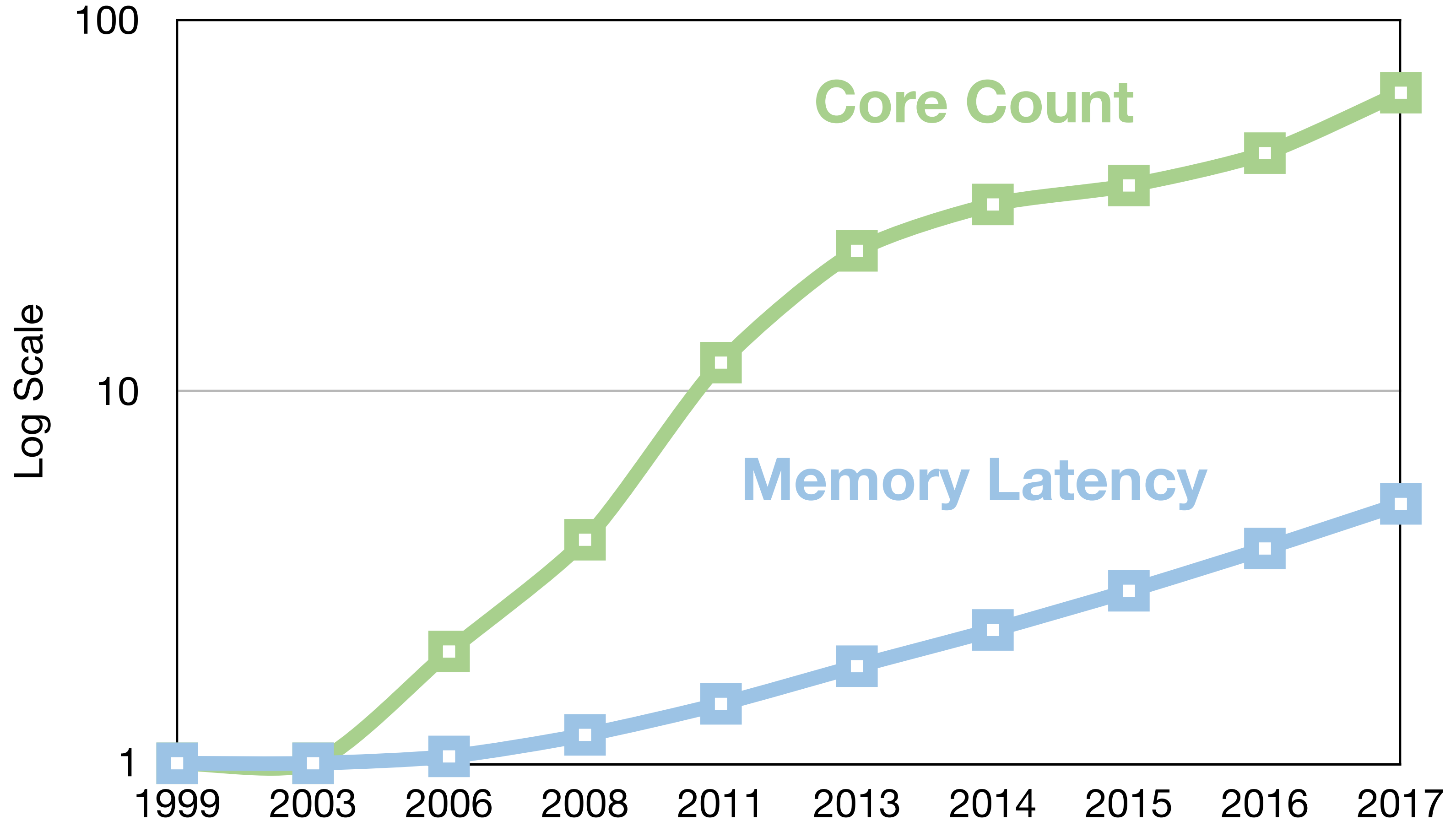
Portable and Programmer-Friendly In-/Near-Memory Fusion

Zhengrong Wang[1], Christopher Liu[1], Aman Arora[2], Lizy John[2], Tony Nowatzki[1]
[1]UCLA, [2]UT Austin

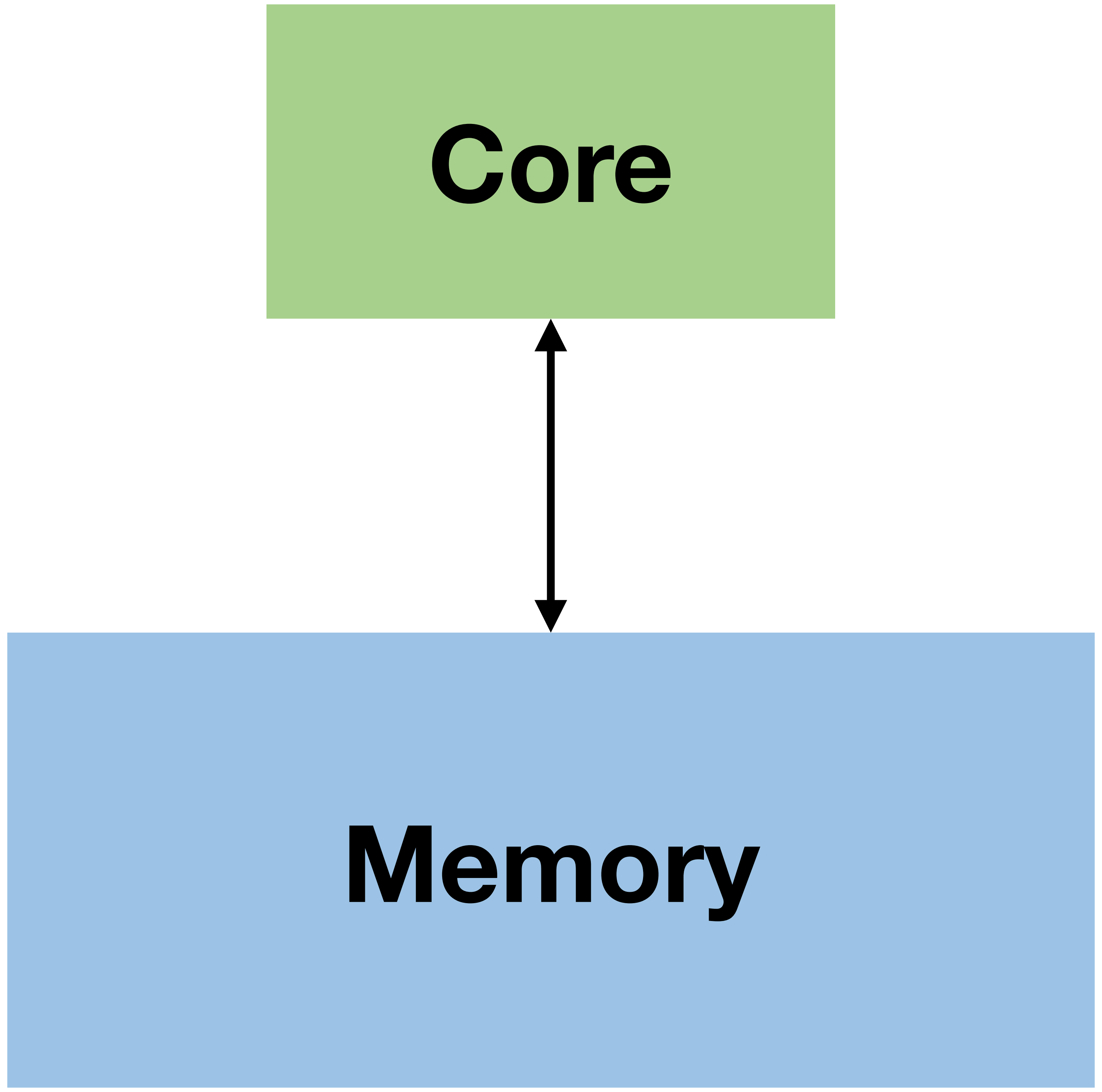
ASPLOS 2023
Vancouver, Canada

Von Neumann Bottleneck

Performance limited by memory bottleneck

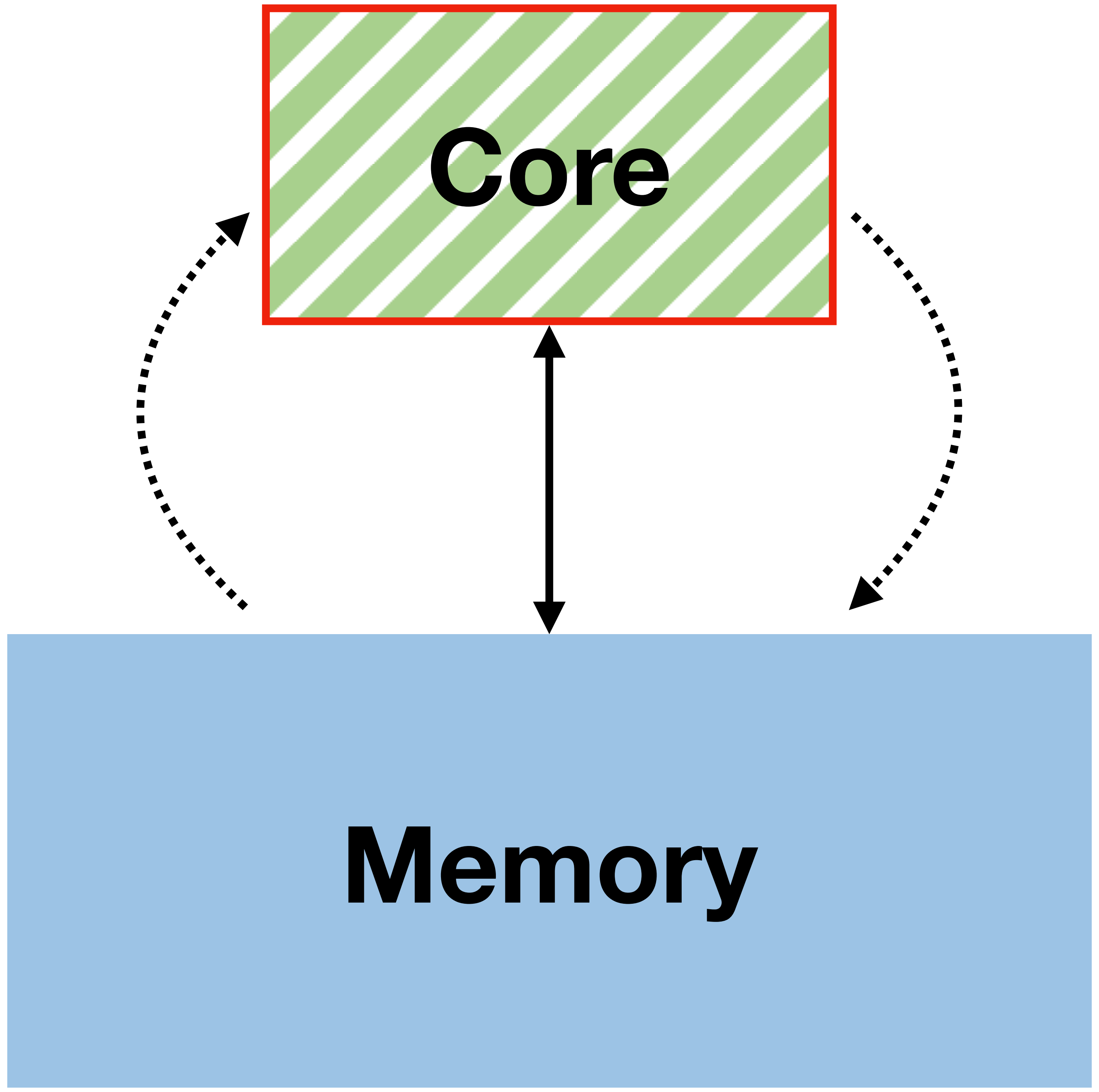


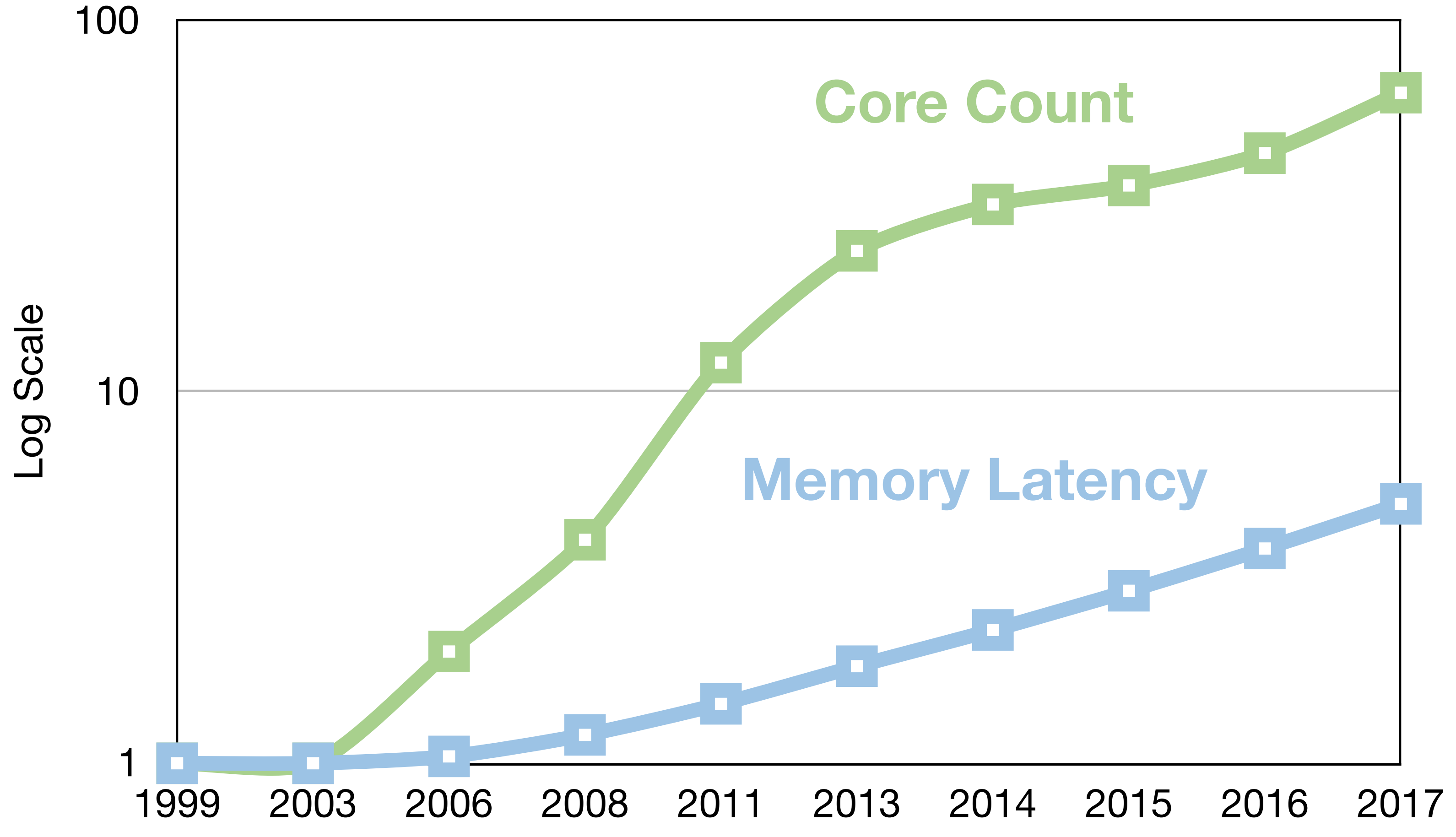
- *Understanding Latency Variation in Modern DRAM Chips: Experimental Characterization, Analysis, and Optimization*
- Karl Rupp's *48 Years of Microprocessor Trend*



Von Neumann Bottleneck

Performance limited by memory bottleneck





- *Understanding Latency Variation in Modern DRAM Chips: Experimental Characterization, Analysis, and Optimization*
- Karl Rupp's *48 Years of Microprocessor Trend*

Expensive data movement

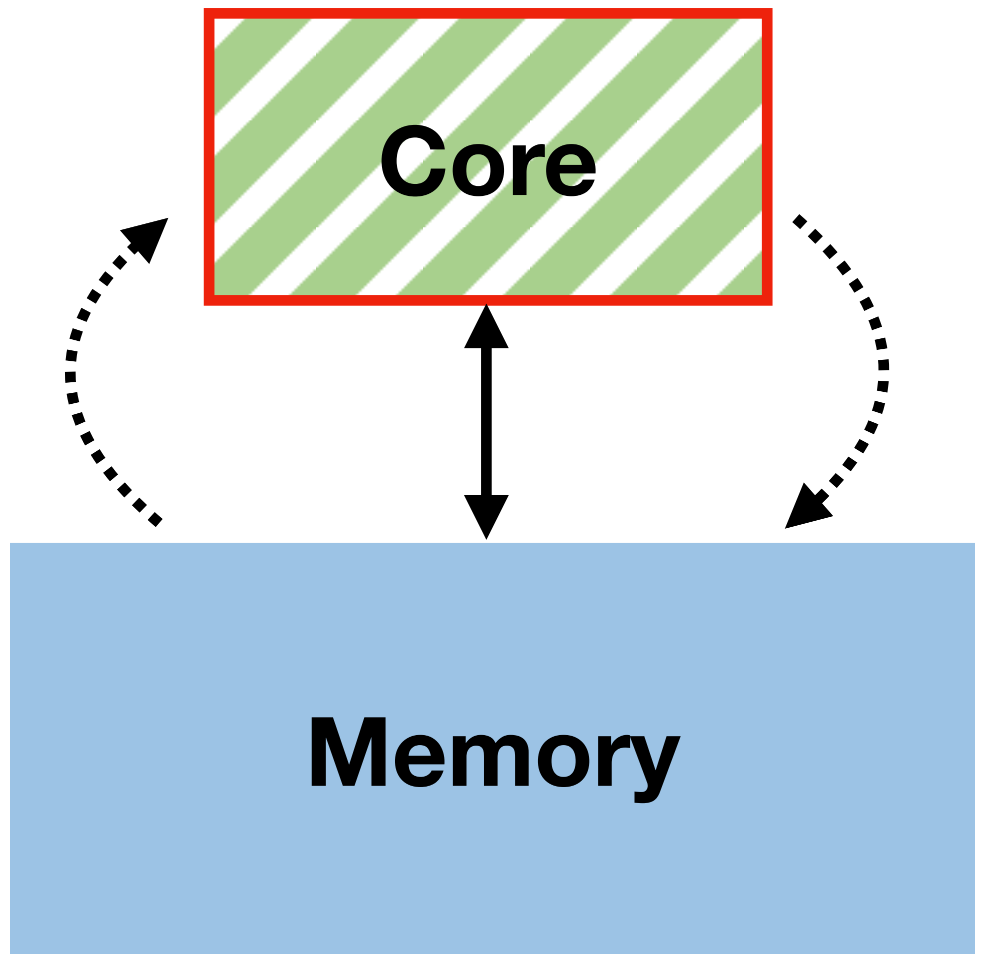
Compute Paradigms

More Expressive
Less Parallelism

Less Expressive
More Parallelism

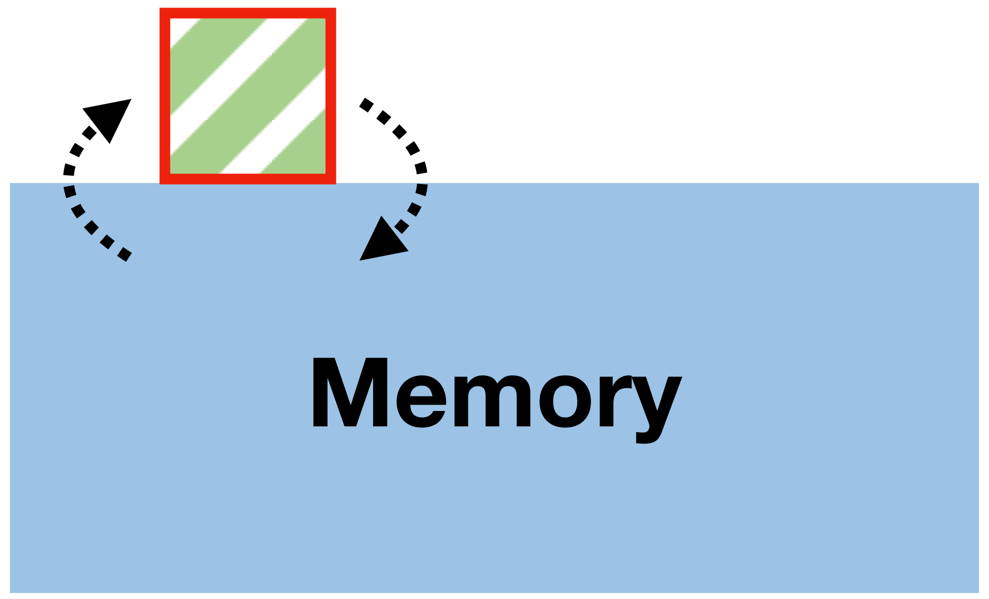
In-Core

Von Neumann Model



Near-Memory

Spatially Local Simple Cores



In-Memory

Massive Vector Processing

👍 **Supports complex control flow**

👎 **Von Neumann bottleneck**

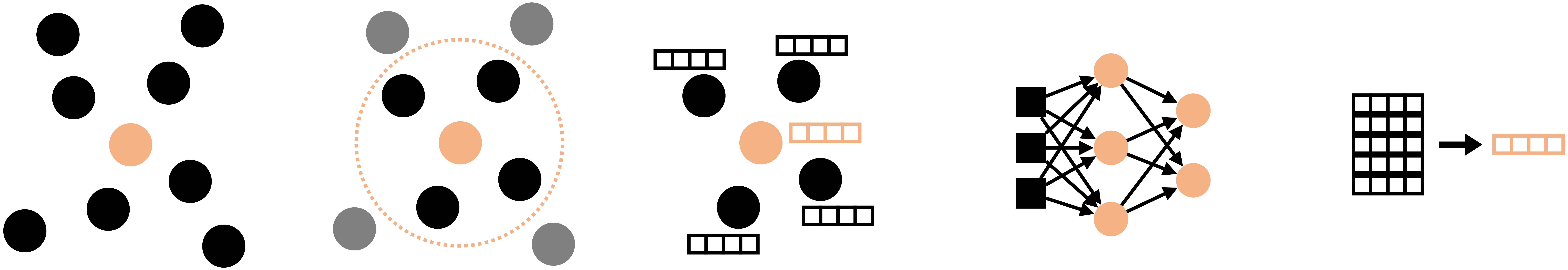
👍 **Programs similar to in-core without control flow**

👎 **Limited parallelism**

👍 **Massive amounts of parallelism**

👎 **Difficult to program due to many restrictions**

Example: PointNet++



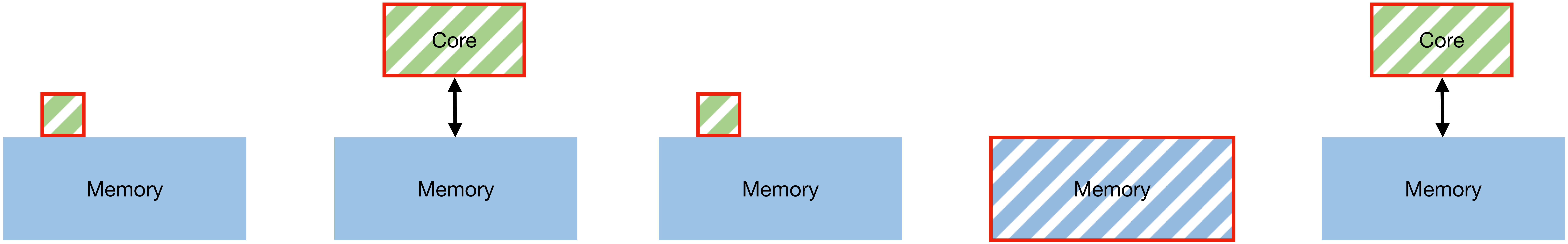
Furthest Sample — Irregular memory access

Ball Query — Complex control

Gather — Irregular memory access

Multi-layer Perceptron — Matrix computations

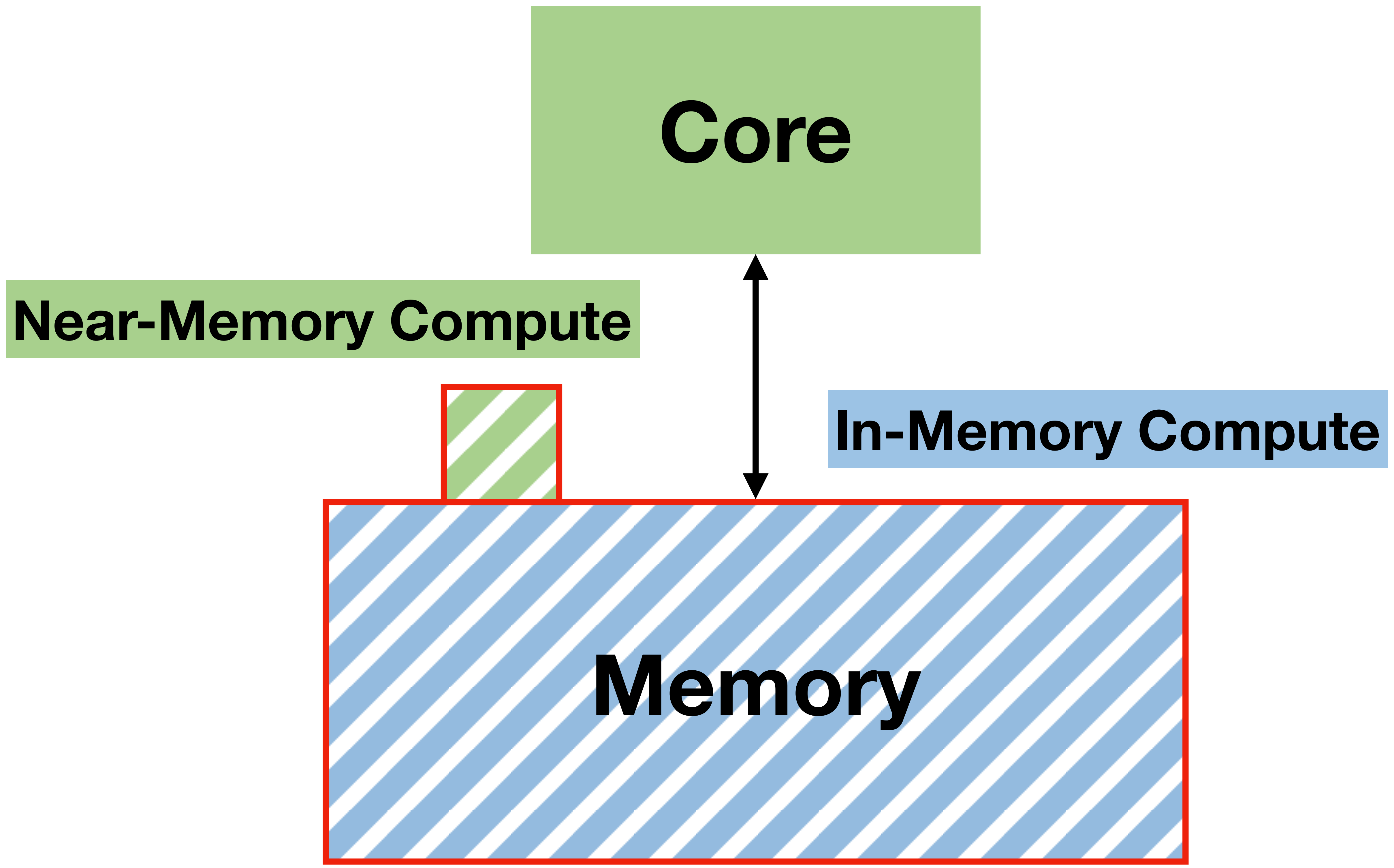
Aggregate — Element-wise *max*()



Near-Memory | **In-Core** | **Near-Memory** | **In-Memory** | **In-Core**

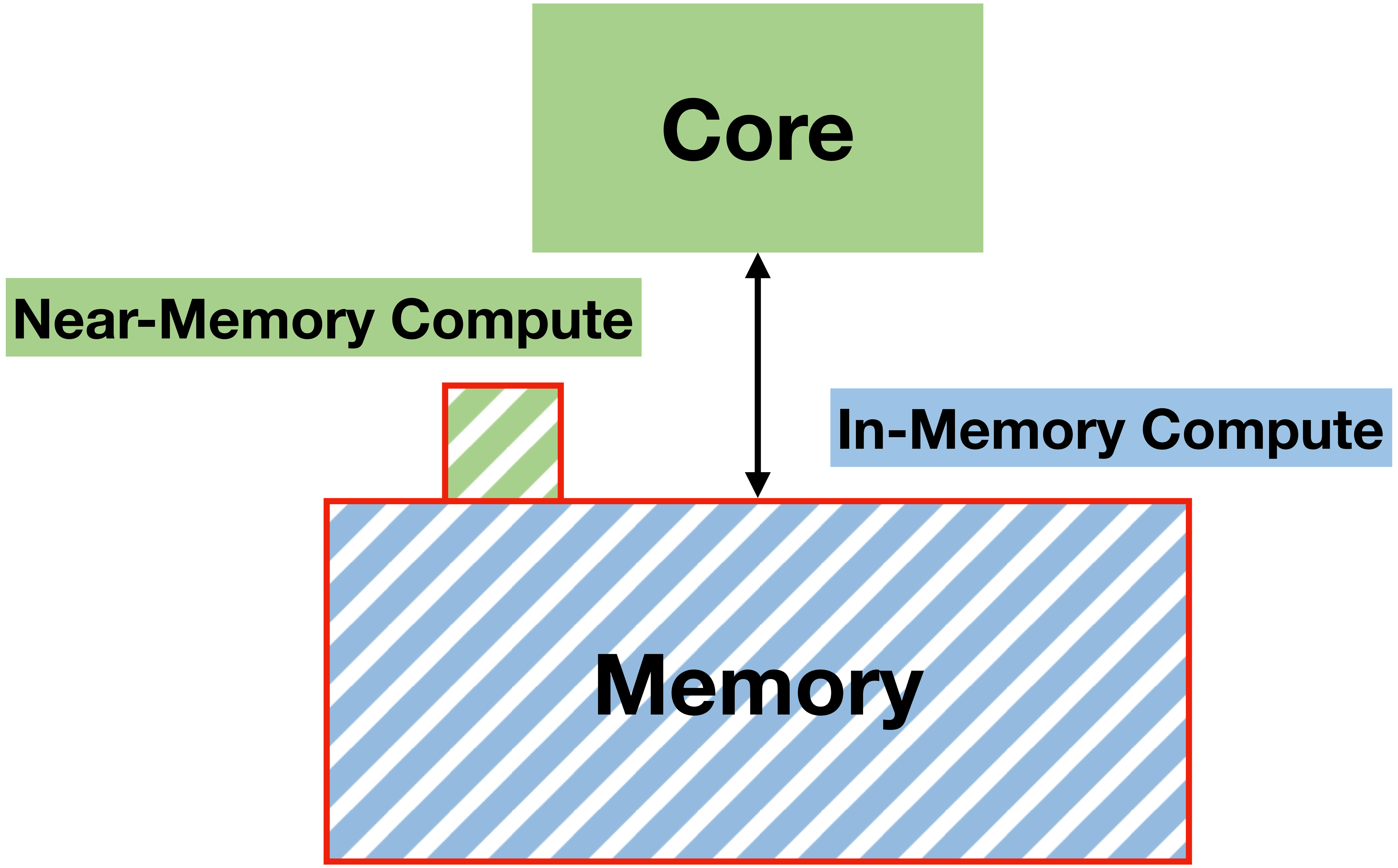
There is an *Adoption Problem*

Programming is *Difficult*




```
Mat matmul(Mat core_A, Mat core_B, Mat core_C):
  InMat in_A = inMemcpy(core_A, M ∗ N, fromCore)
  InMat in_B = inMemcpy(core_B, M ∗ N, fromCore)
  InMat in_C = inMalloc(M ∗ N)
  for m in [0, M):
    for n in [0, N):
      InVec in_V = in_A[m][:] ∗ in_B[:][c]
      in_V = inPartialReduce(+, in_V, rounds=3)
      NearVec near_V = nearMalloc(K)
      near_V = nearMemcpy(in_V, K / 2 / 2 / 2,
                          fromIn)
      NearScalar near_dotpdt = nearReduce(near_V)
      core_C[m][n] = coreMemcpy(near_dotpdt,
                                fromNear)
```

Hide Hardware Details




```
Mat matmul(Mat A, Mat B, Mat C):
  for m in [0, M):
    for n in [0, N):
      for k in [0, K):
        C[m][n] += A[m][k] * B[k][n]
```

Easier to program

Outline

Plain C Code

2
Unified Abstraction

Agnostically represents
1. In-core compute
2. In-memory compute
3. Near-memory compute

4
**Partitioning
Static vs Dynamic**

Portable binary without exposing µArch details to compiler

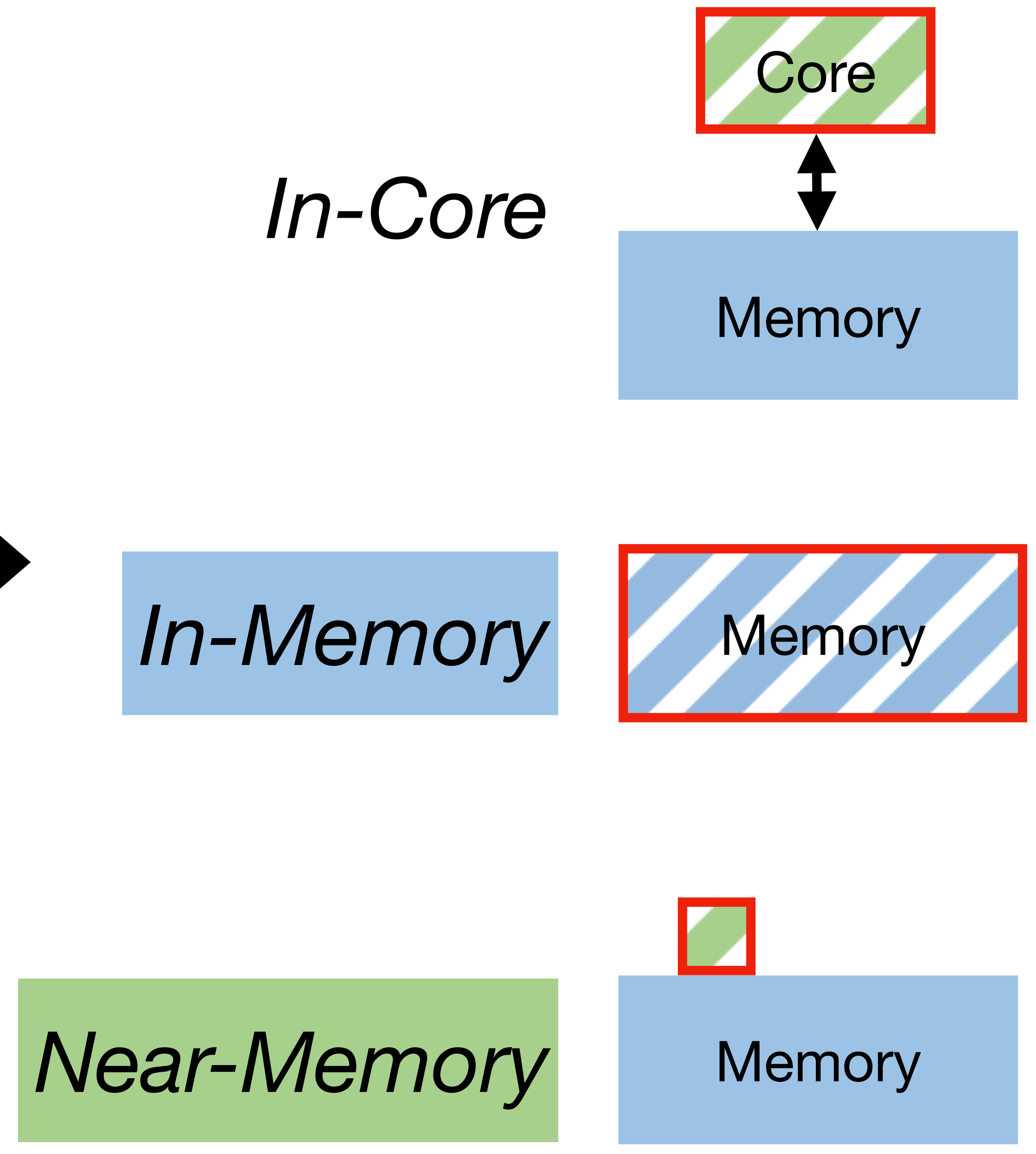
3
µArch-specific Optimizations

Specialize binary to take advantage of µArch details

µArch Extensions

Hardware support for compute paradigms

1
Compute Paradigms



Outline

Plain C Code

2

Unified Abstraction

Agnostically represents
1. In-core compute
2. In-memory compute
3. Near-memory compute

4

Partitioning Static vs Dynamic

Portable binary without exposing µArch details to compiler

3

µArch-specific Optimizations

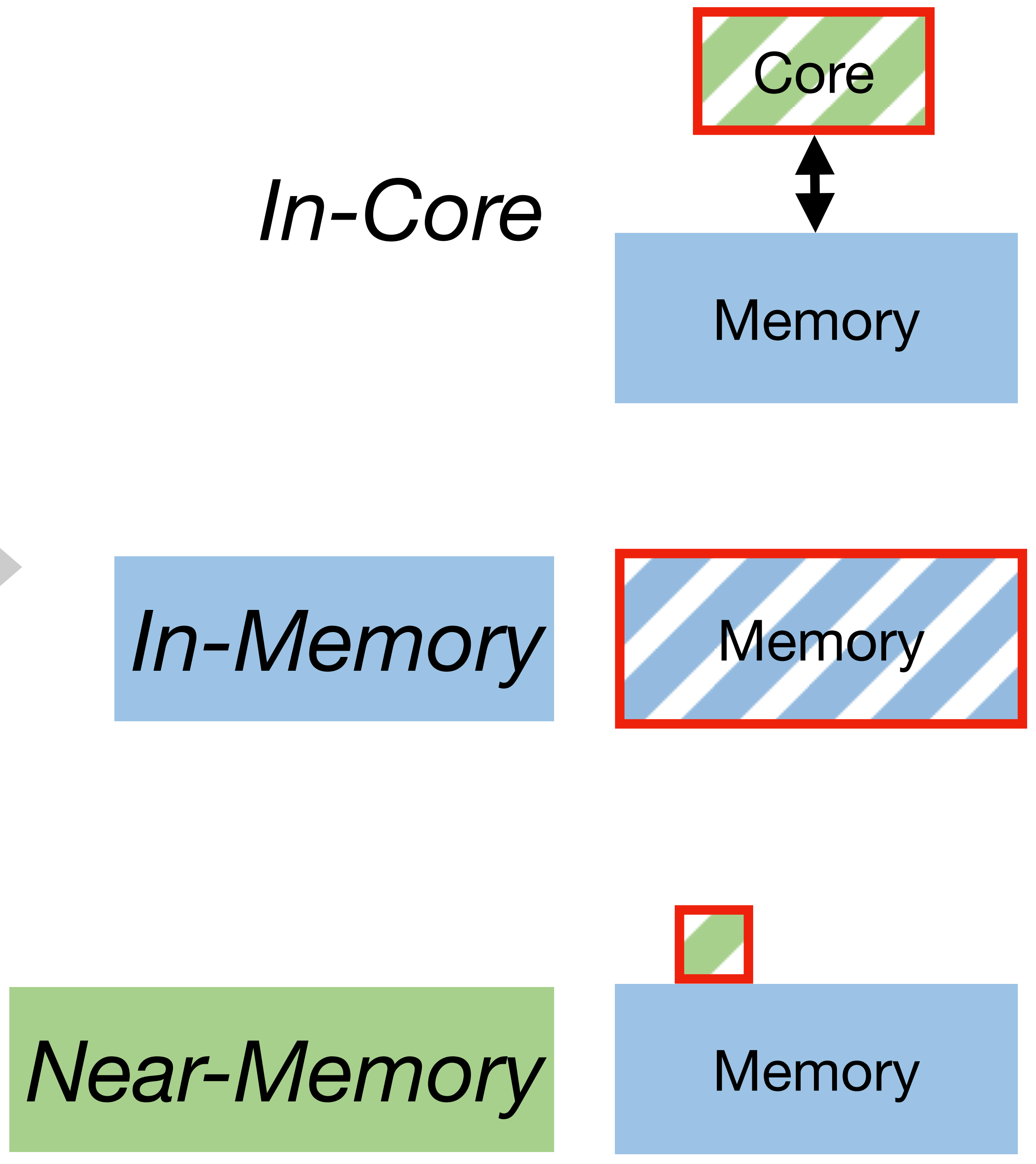
Specialize binary to take advantage of µArch details

µArch Extensions

Hardware support for compute paradigms

1

Compute Paradigms



Near-Memory Compute

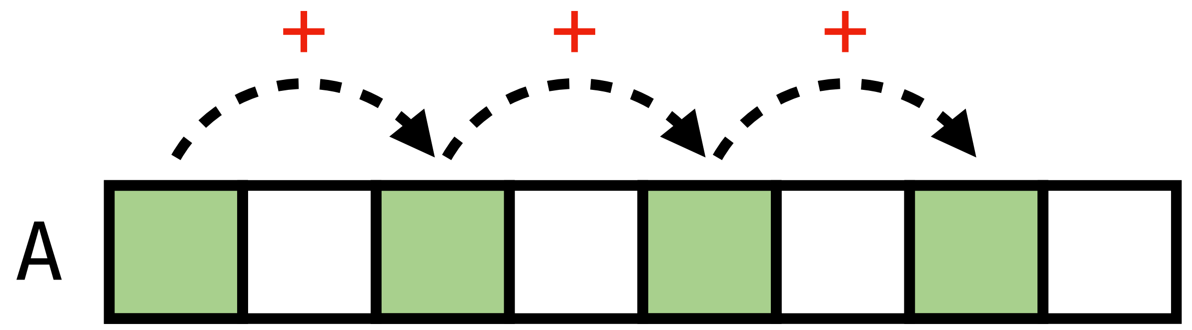
Offload computation closer to the data

`C[i] = A[i] & B[i]`

Supports complex **memory access patterns**

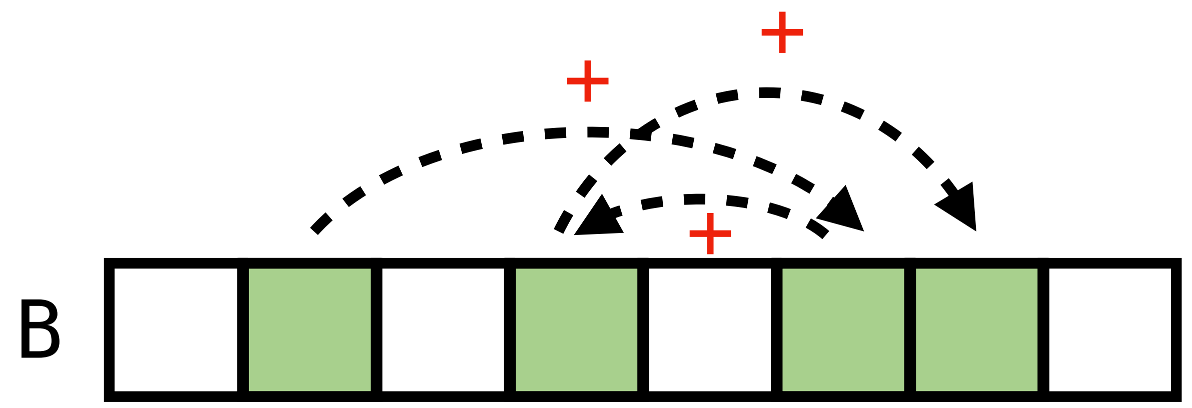
Affine Patterns

e.g., `A[2*i]`



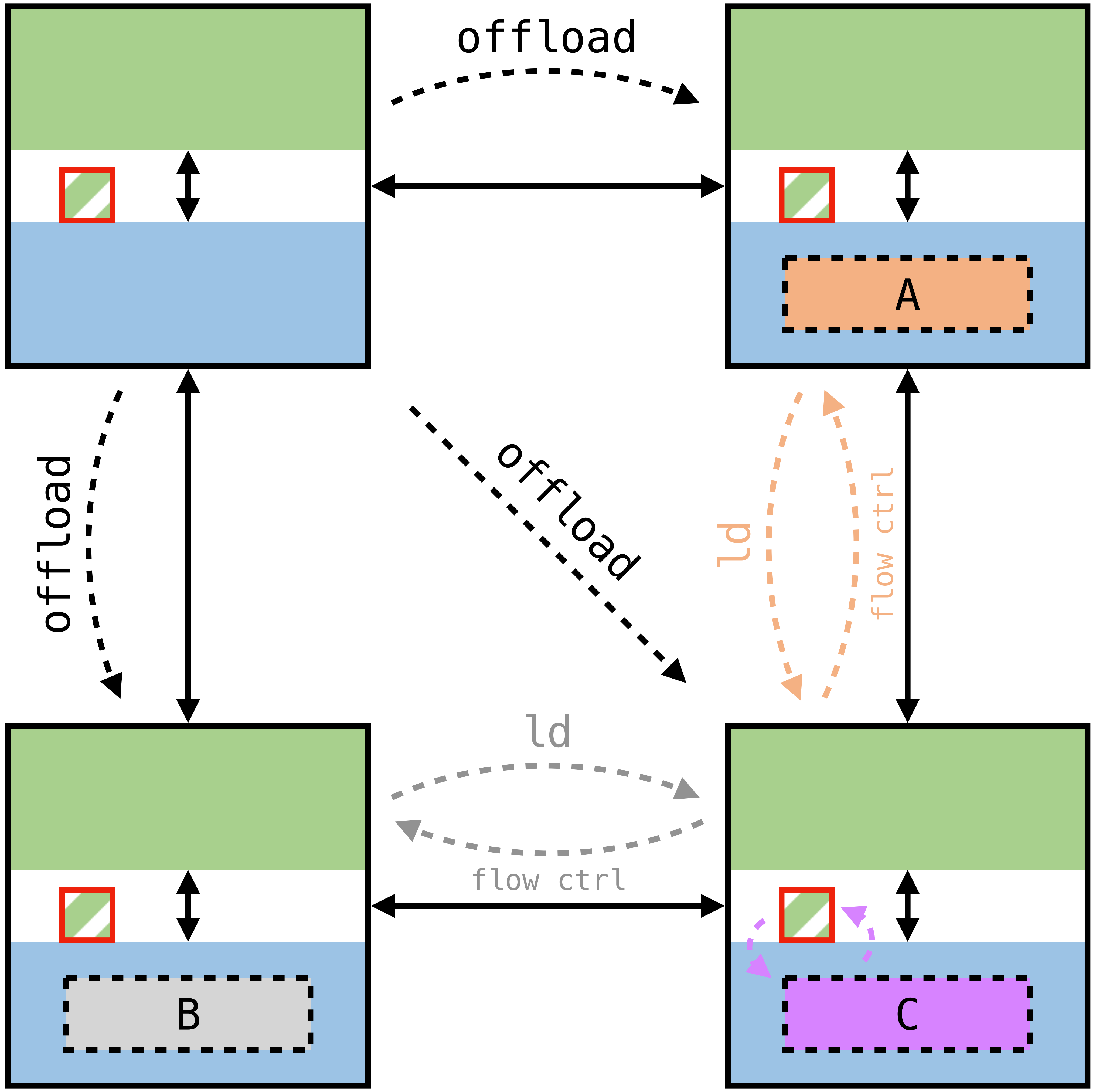
Indirect Patterns

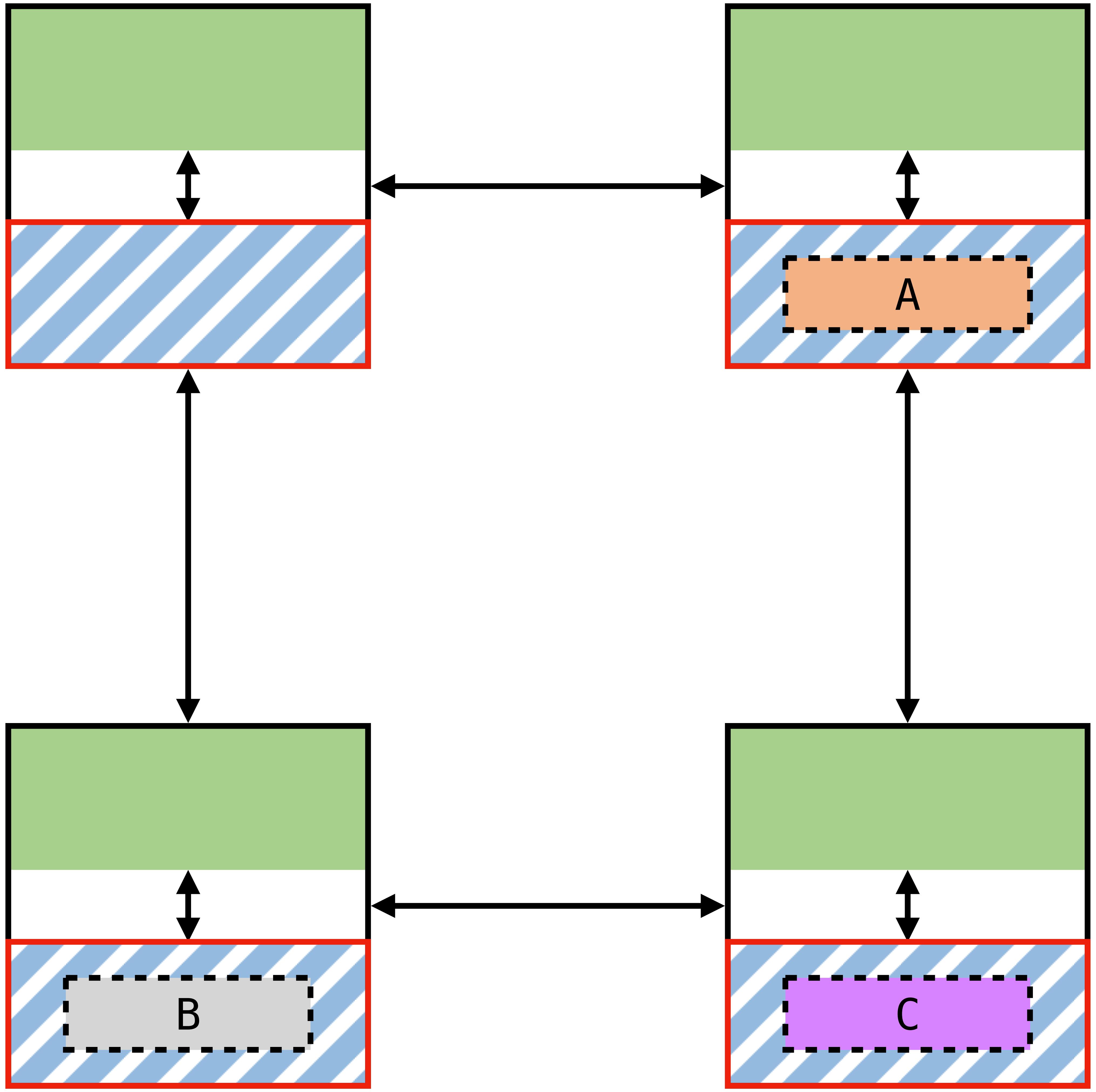
e.g., `B[A[i]]`



with **reduction** capabilities

Limitation: Lower compute width

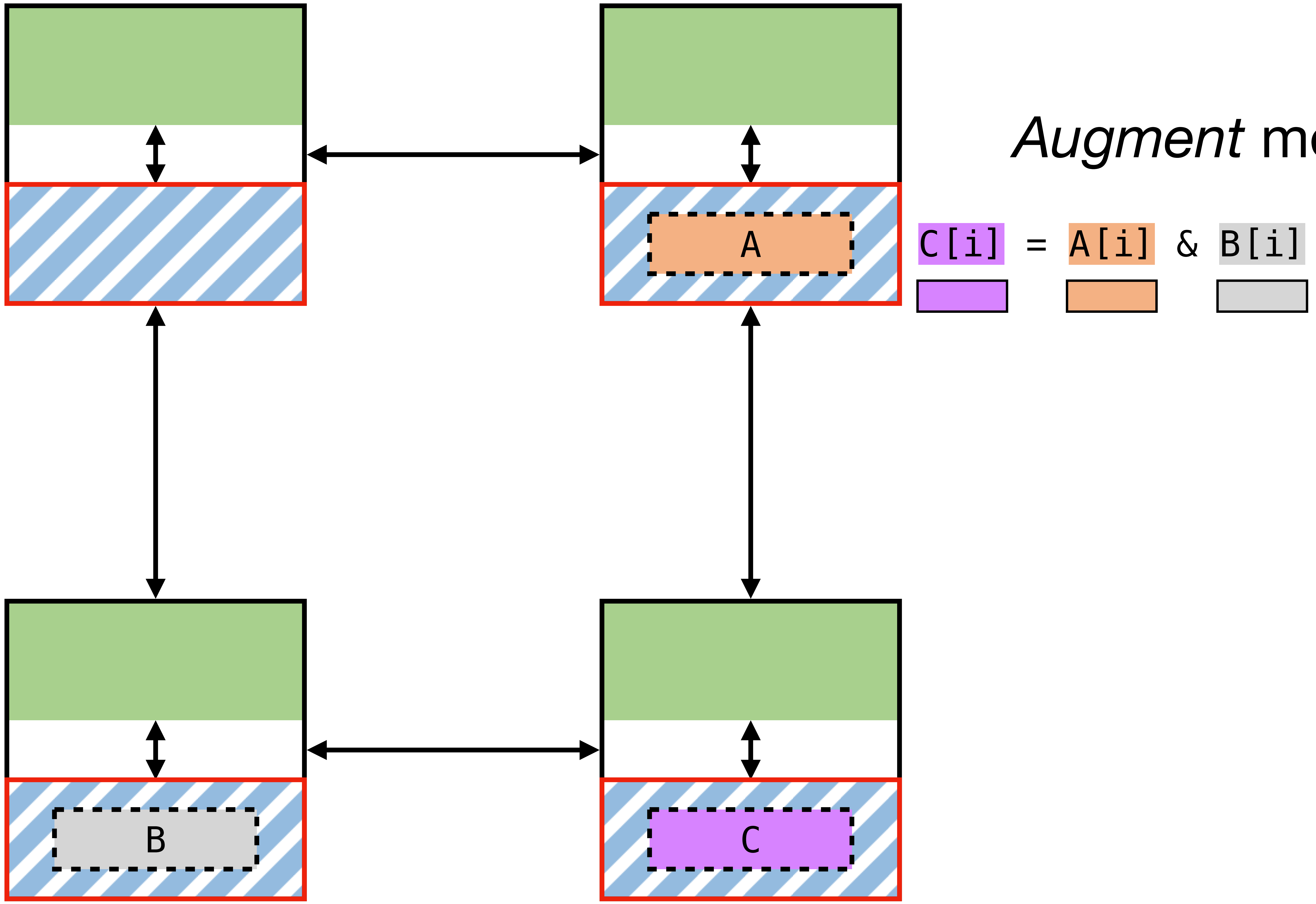




In-Memory Compute

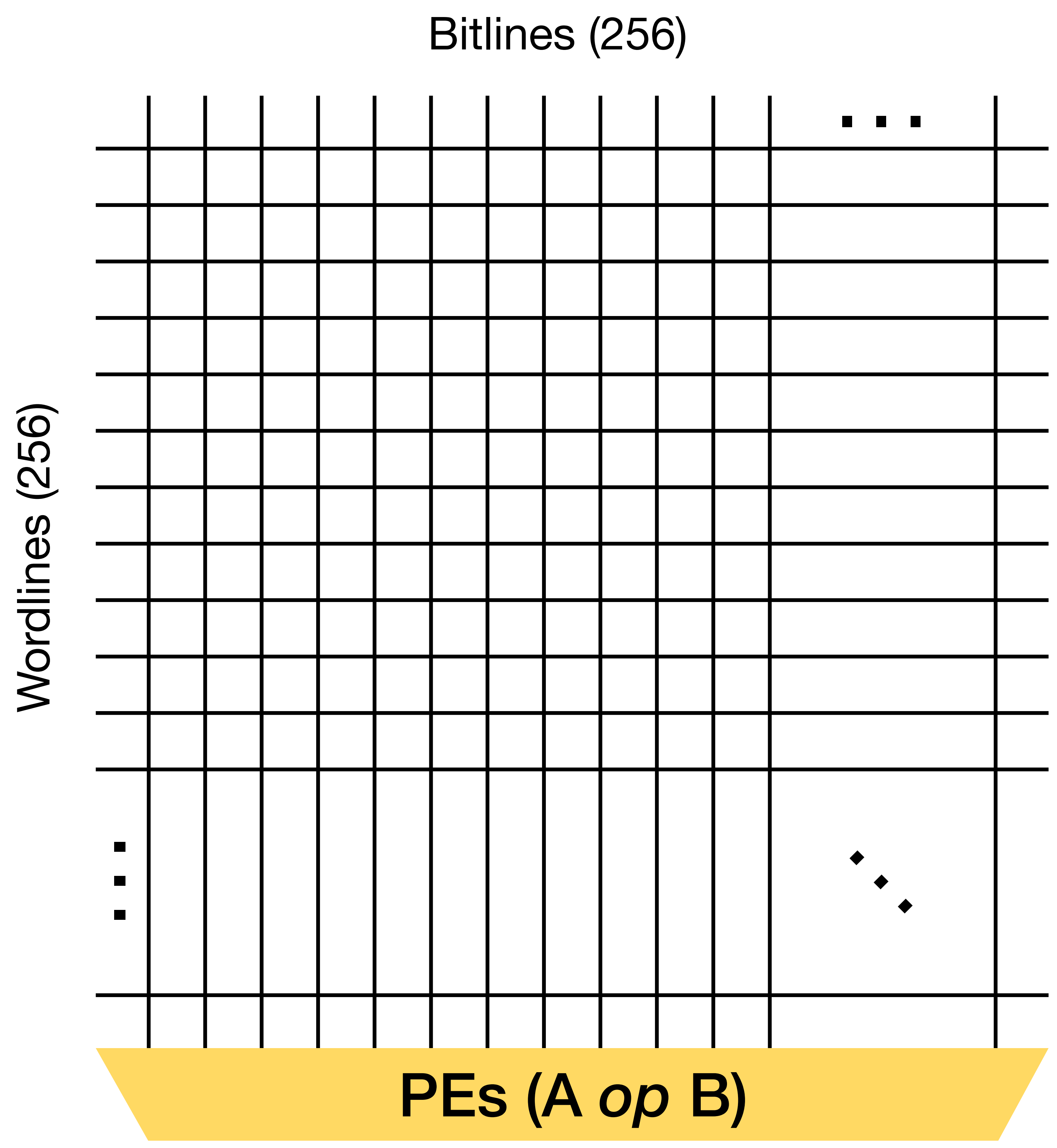
Augment memory technology with compute

`C[i] = A[i] & B[i]`

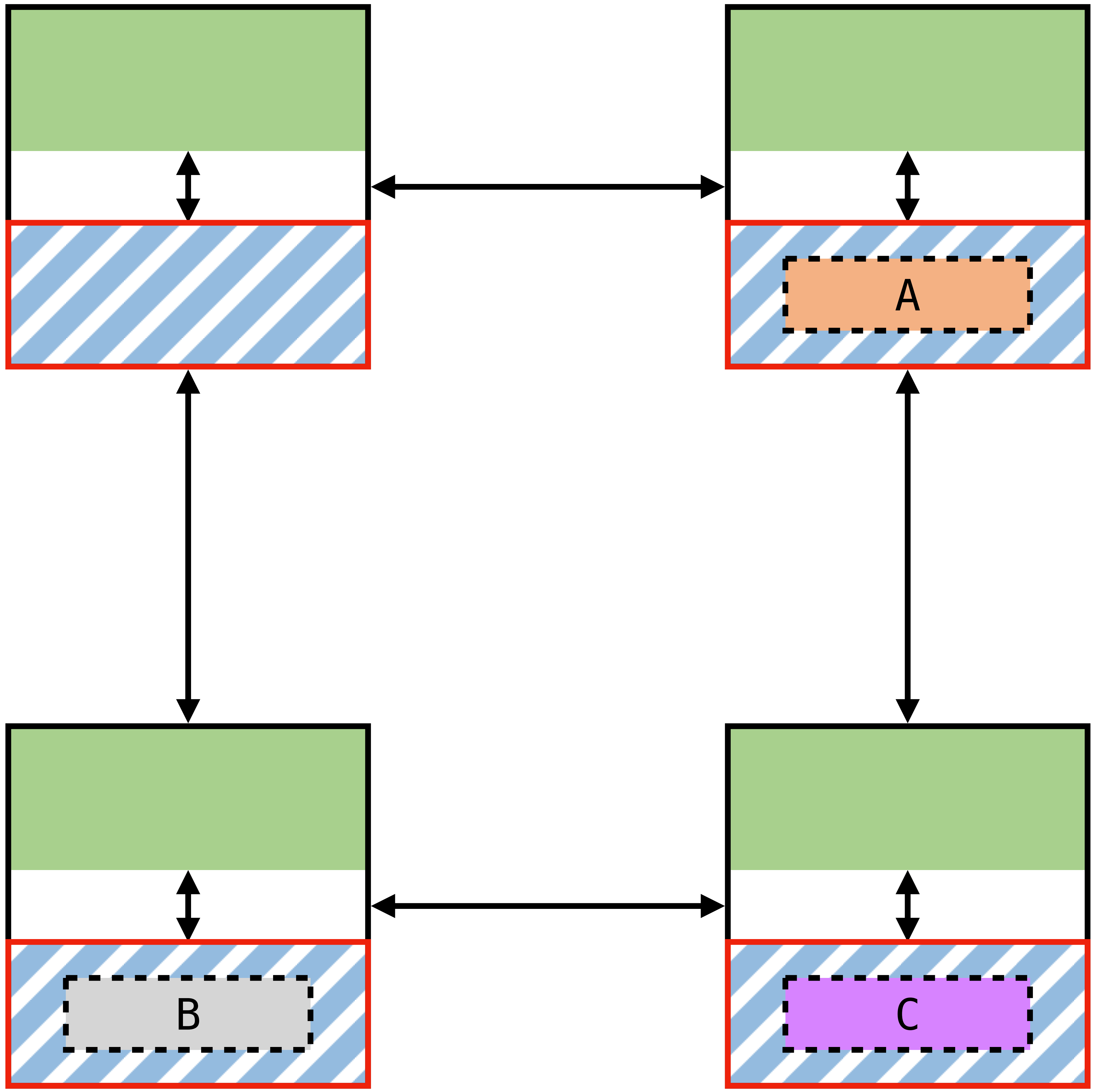


In-Memory Compute

Augment memory technology with compute



SRAM Array

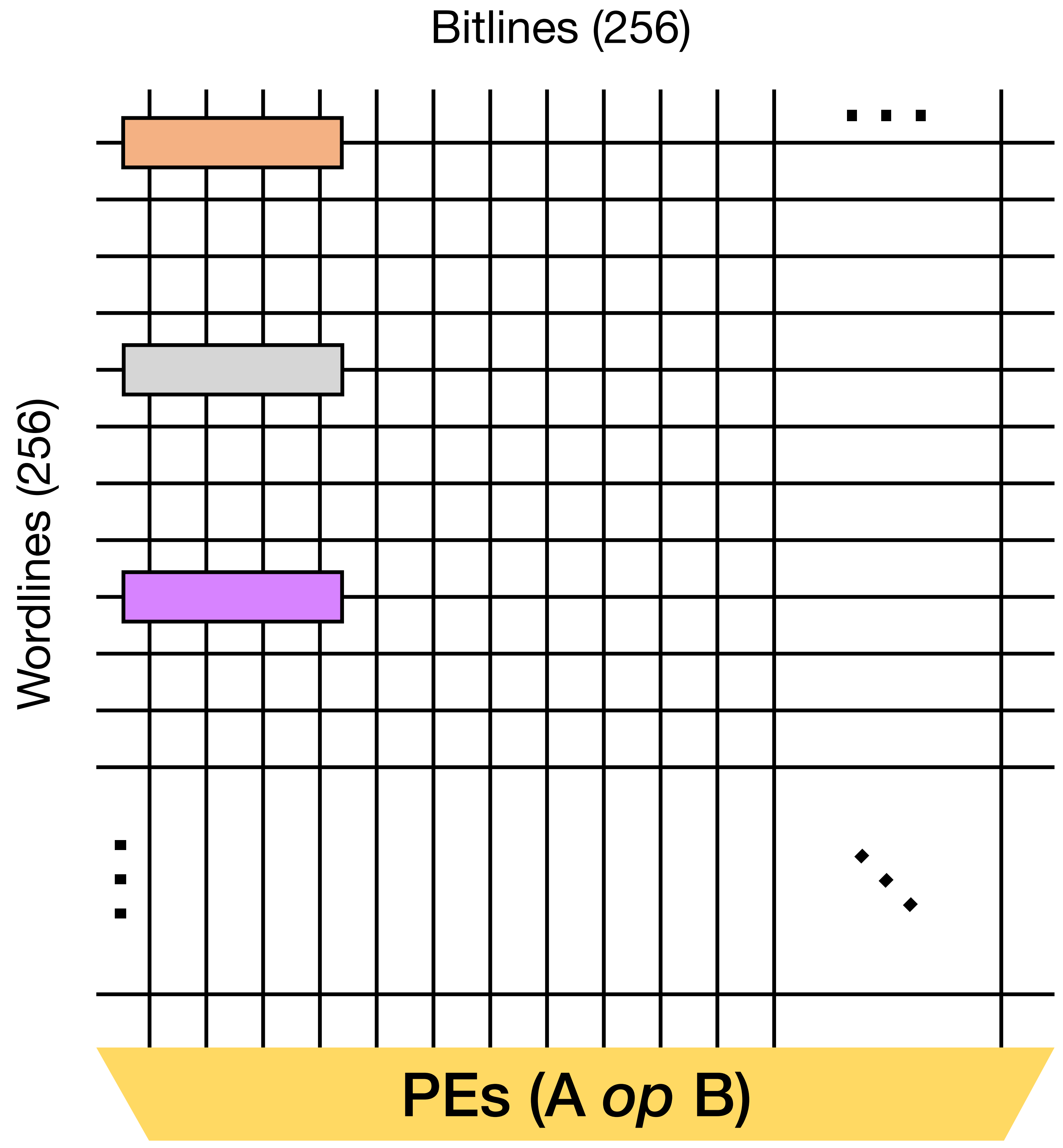


In-Memory Compute

Augment memory technology with compute

`C[i] = A[i] & B[i]`

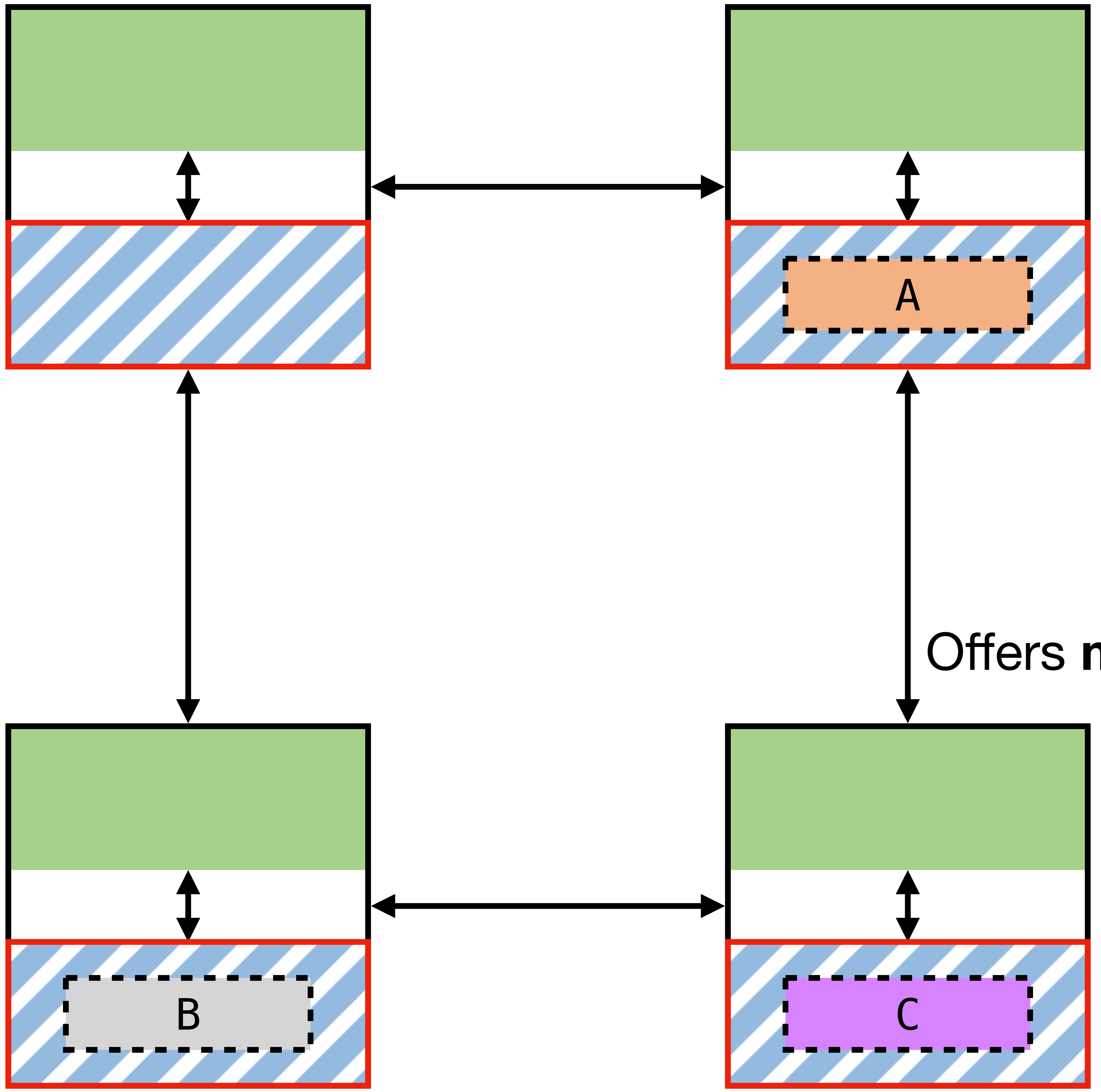
Standard Data Layout



SRAM Array

In-Memory Compute

Augment memory technology with compute



`C[i] = A[i] & B[i]`

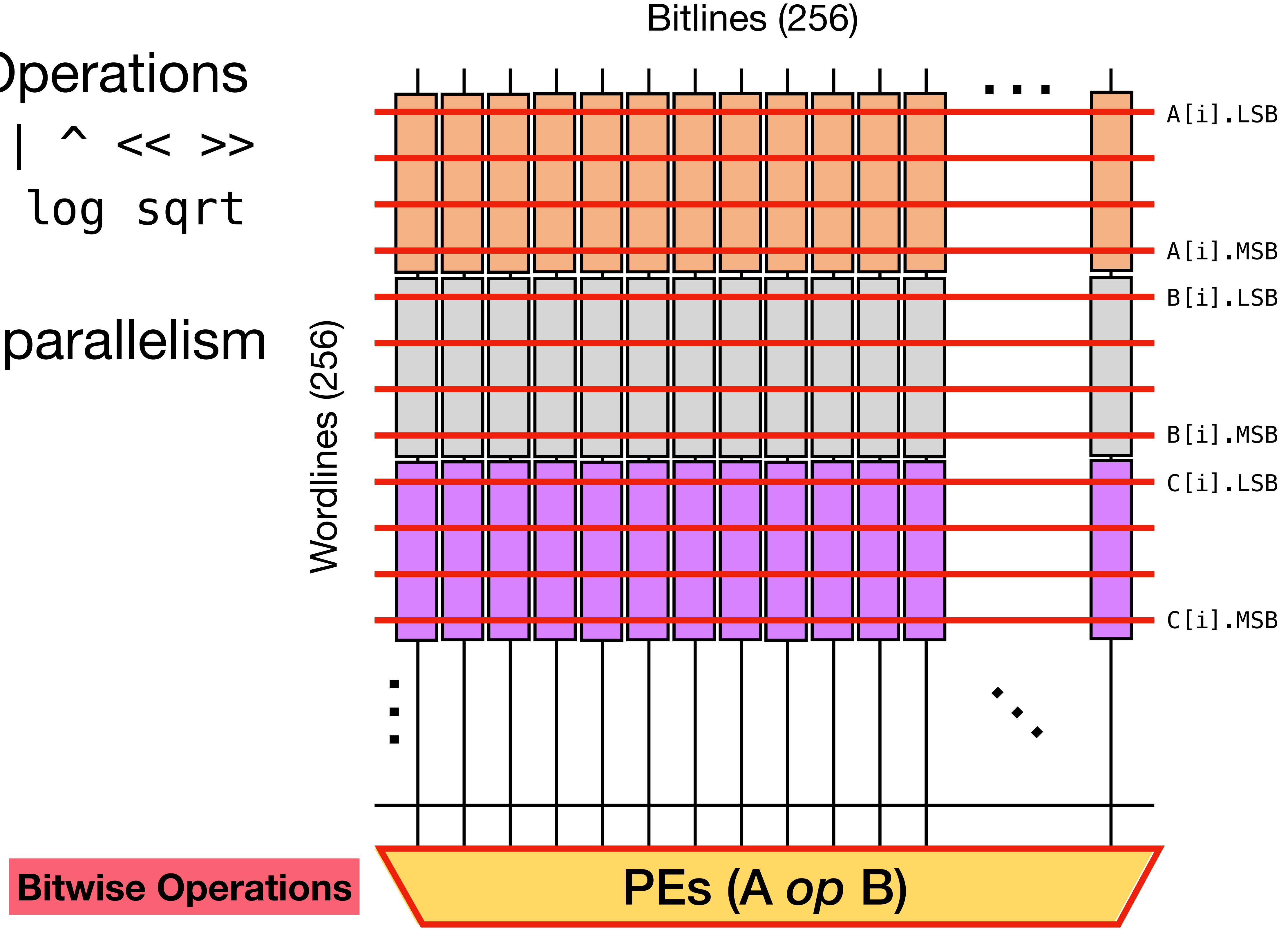
Supported Operations

`+ − * / % & | ^ << >>`

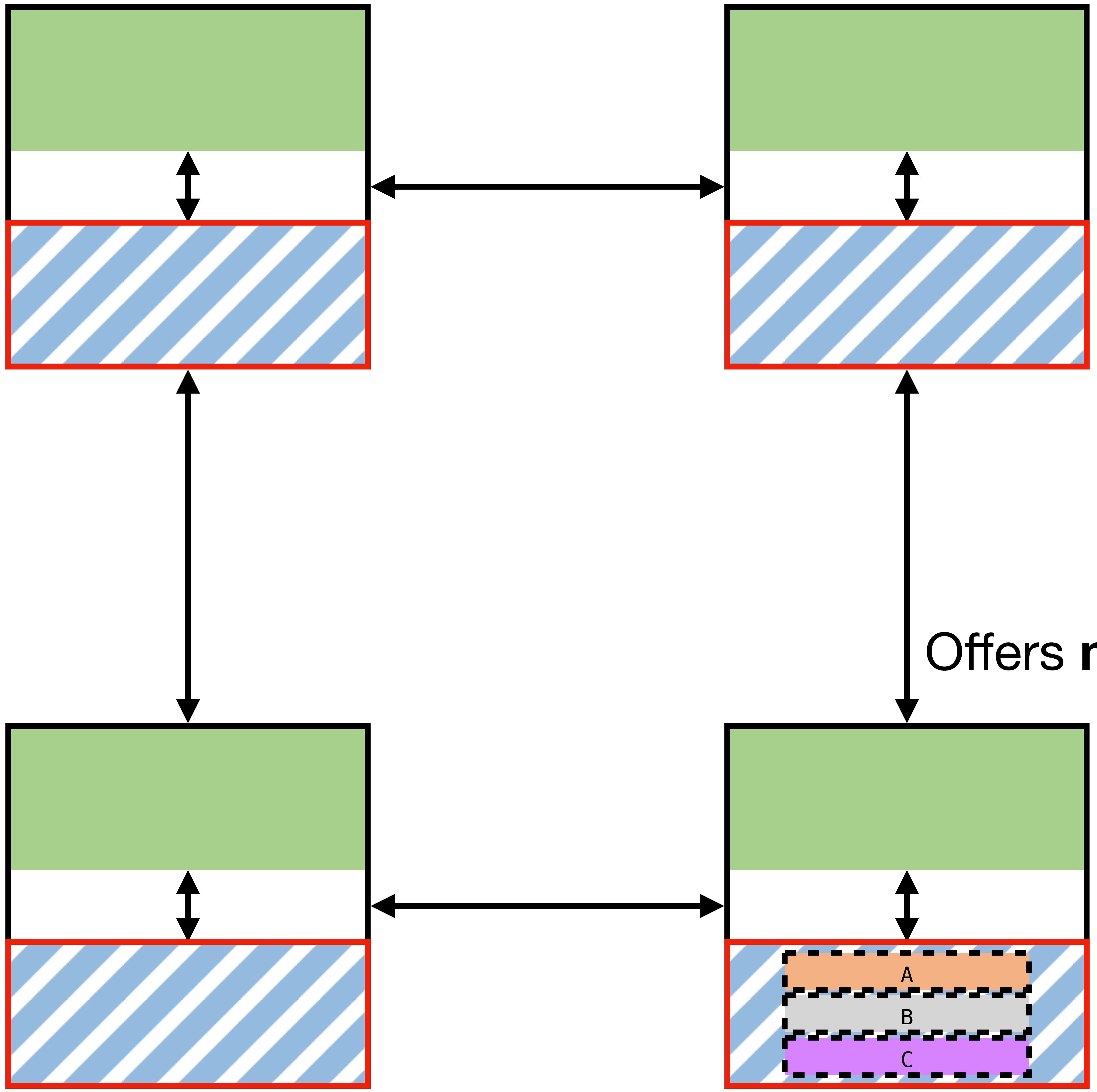
`sin cos exp log sqrt`

Offers **massive** vector parallelism

Bit-Serial Data Layout



SRAM Array



In-Memory Compute

Augment memory technology with compute

Supported Operations

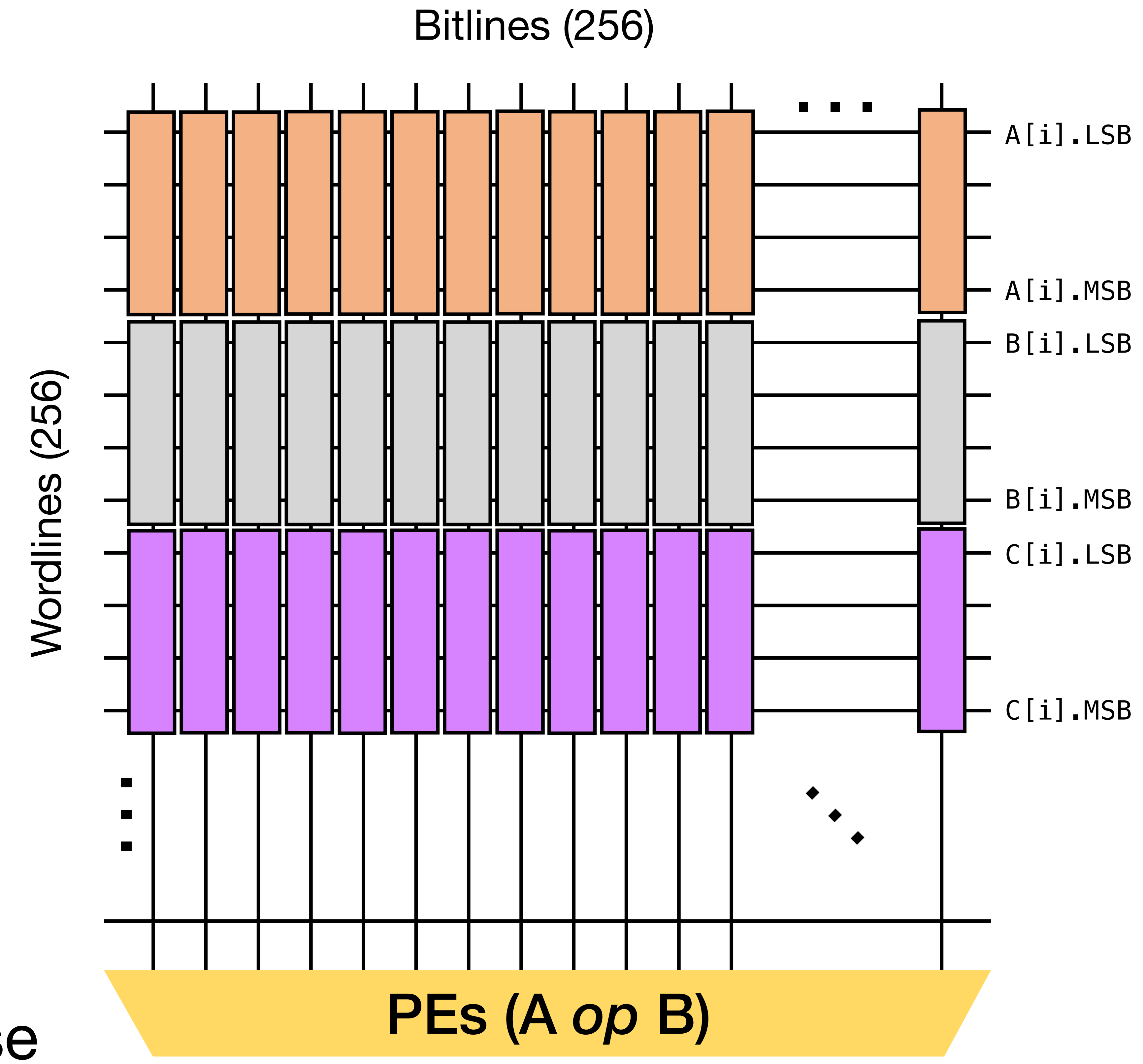
`+ - * / % & | ^ << >>`

`sin cos exp log sqrt`

Offers **massive** vector parallelism

`C[i] = A[i] & B[i]`

Bit-Serial Data Layout



SRAM Array

Challenge: Requires data alignment & transpose

Compute Paradigms

In-Memory Compute

Offers **massive** vector parallelism

Challenge: Requires data alignment & transpose

Near-Memory Compute

Supports complex **memory access patterns** with **reduction** capabilities

Limitation: Lower compute width

Programming Considerations

1. Orchestration

Meet the requirements for each compute paradigm

- Data alignment
- Data layout & tiling
- Bit-serial transpose
- Managing on-chip space

2. Fused In-/Near-Memory

Statically & dynamically take advantage of each compute paradigm

- **In-Memory:** Large input size & element-wise compute
- **Near-Memory:** Small input size or irregular memory patterns
- **Fusion** of in-/near-memory

3. Portability

Target a large variety of microarchitectures with a single binary

Programming Considerations

1. Orchestration

Meet the requirements for each compute paradigm

- Data alignment
- Data layout & tiling
- Bit-serial transpose
- Managing on-chip space

2. Fused In-/Near-Memory

Statically & dynamically take advantage of each compute paradigm

- **In-Memory:** Large input size & element-wise compute
- **Near-Memory:** Small input size or irregular memory patterns
- **Fusion** of in-/near-memory

3. Portability

Target a large variety of microarchitectures with a single binary

Transparency

minimizes programmer burden

Outline

Plain C Code

2

Unified Abstraction

Agnostically represents
1. In-core compute
2. In-memory compute
3. Near-memory compute

3

µArch-specific Optimizations

Specialize binary to take advantage of µArch details

4

Partitioning Static vs Dynamic

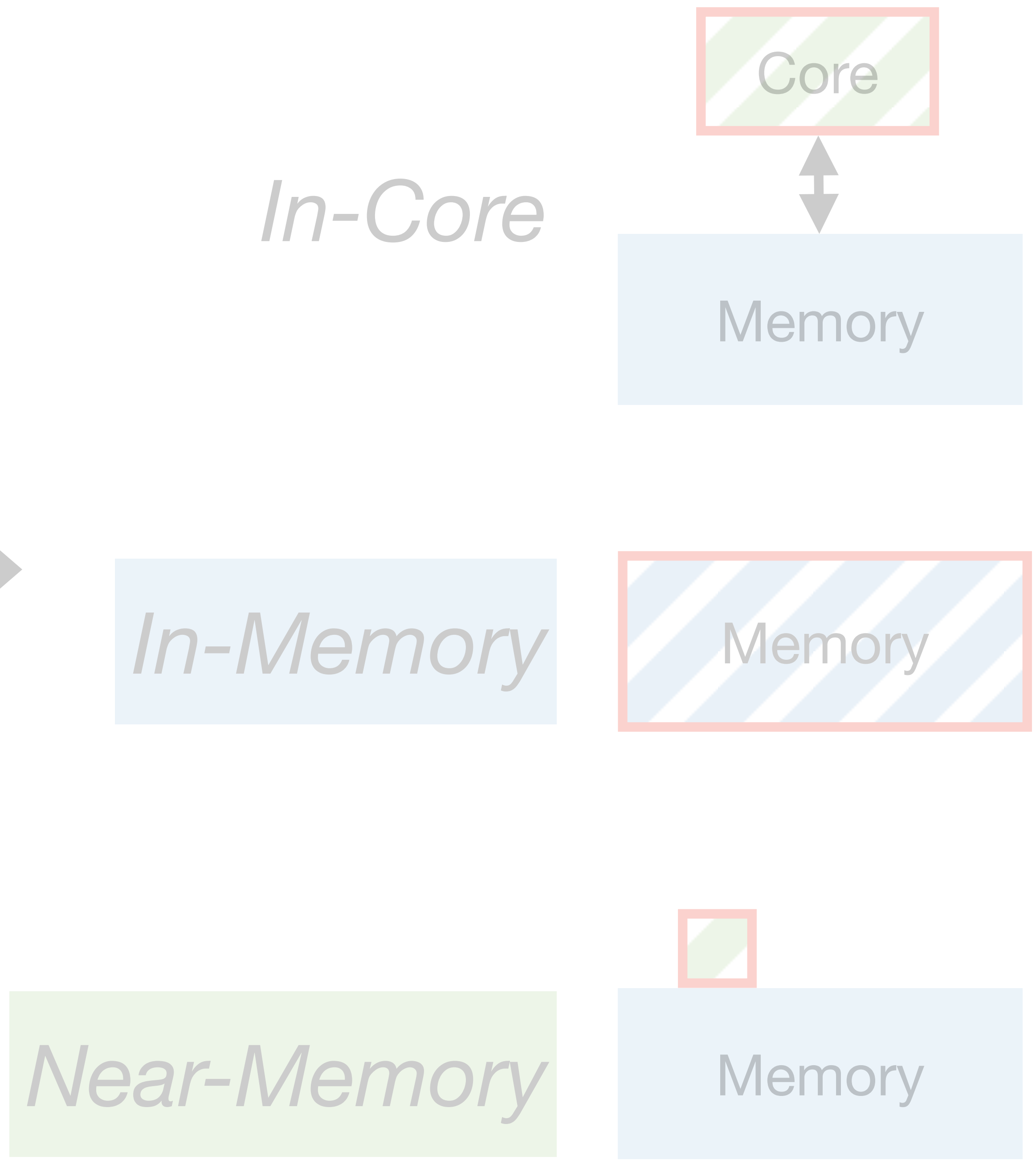
Portable binary without exposing µArch details to compiler

µArch Extensions

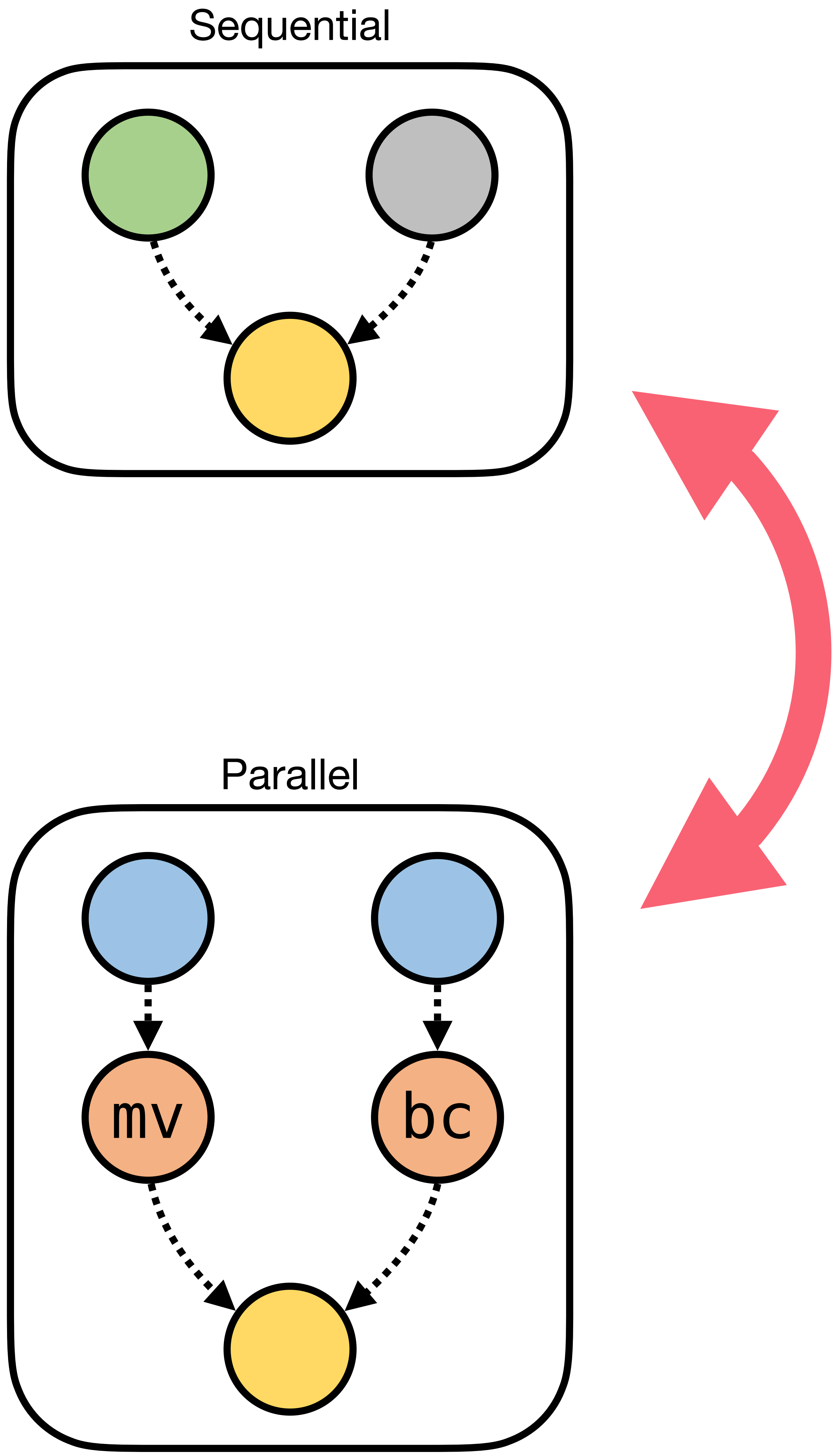
Hardware support for compute paradigms

1

Compute Paradigms



Dataflow Representation



Near-Memory Compute

- Memory access pattern
- Computation

In-Memory Compute

- N-D Tensors
- Data alignment
- Spatial reuse

Fused In-/Near-Memory

- Reduction operations
- Fusion

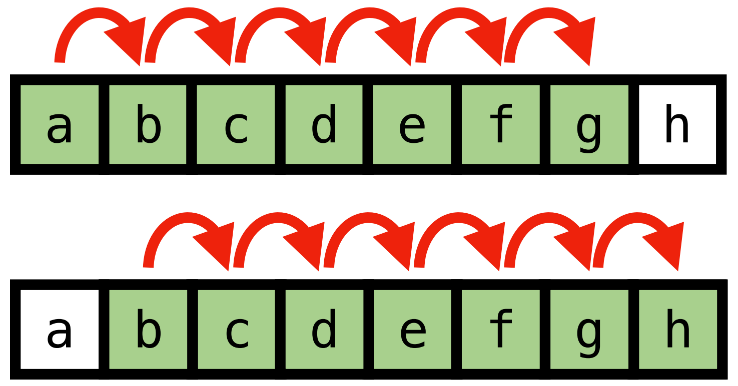
Stream Dataflow Graph

- Memory access pattern

1D Filter

```
for i in [0, N-1):
  B[i] = F[0] x A[i  ]
       + F[1] x A[i+1]
```

Memory Access Pattern



encoded by **Stream Node** A_0

encoded by **Stream Node** A_1

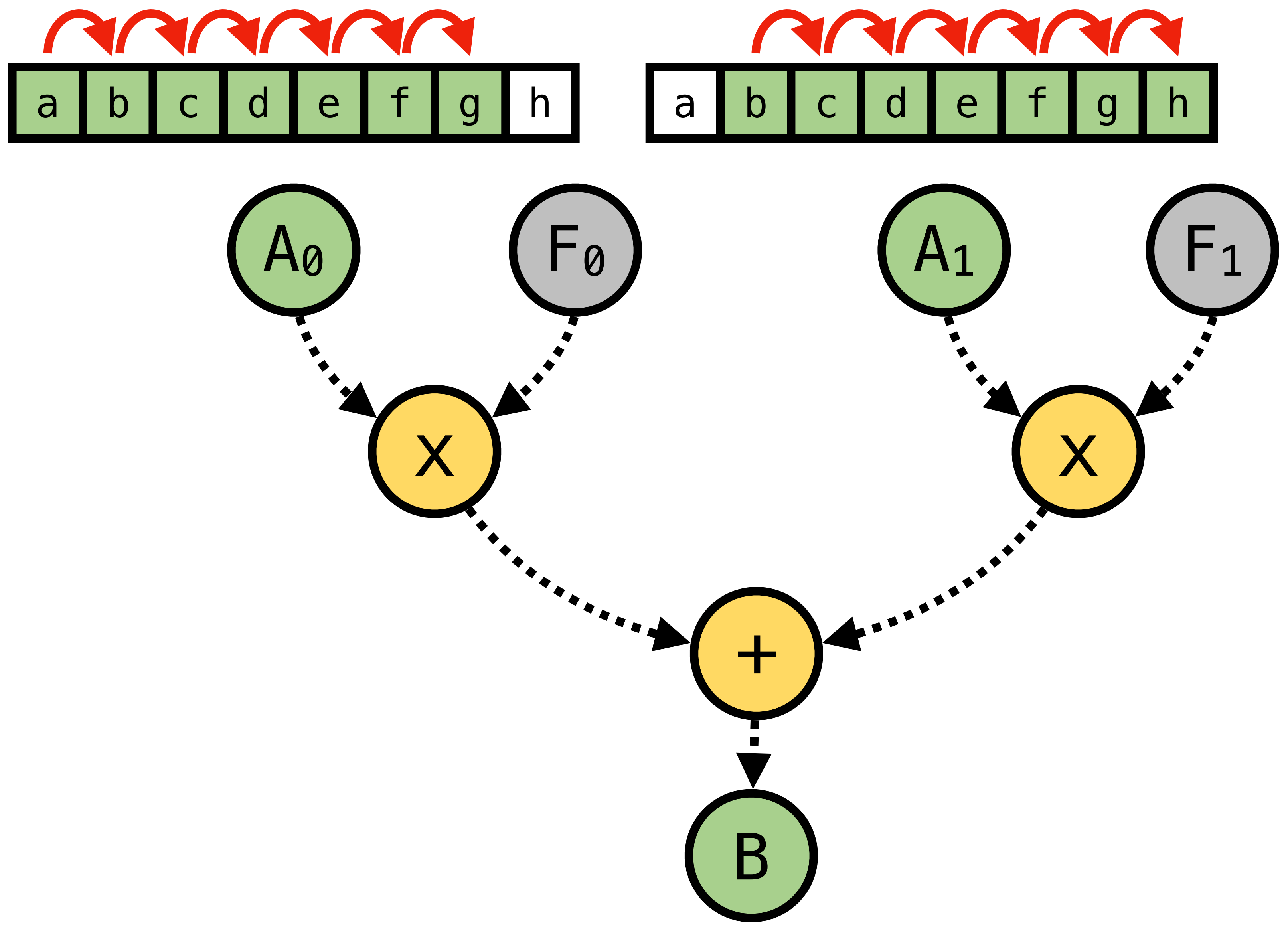
- Computation
- N-D Tensors
- Data alignment
- Reduction operations
- Spatial reuse
- Fusion

Stream Dataflow Graph

- Memory access pattern
- Computation

1D Filter

```
for i in [0, N−1):
  B[i] = F[0] x A[i  ]

       + F[1] x A[i+1]
```



- N-D Tensors
- Data alignment
- Reduction operations
- Spatial reuse
- Fusion

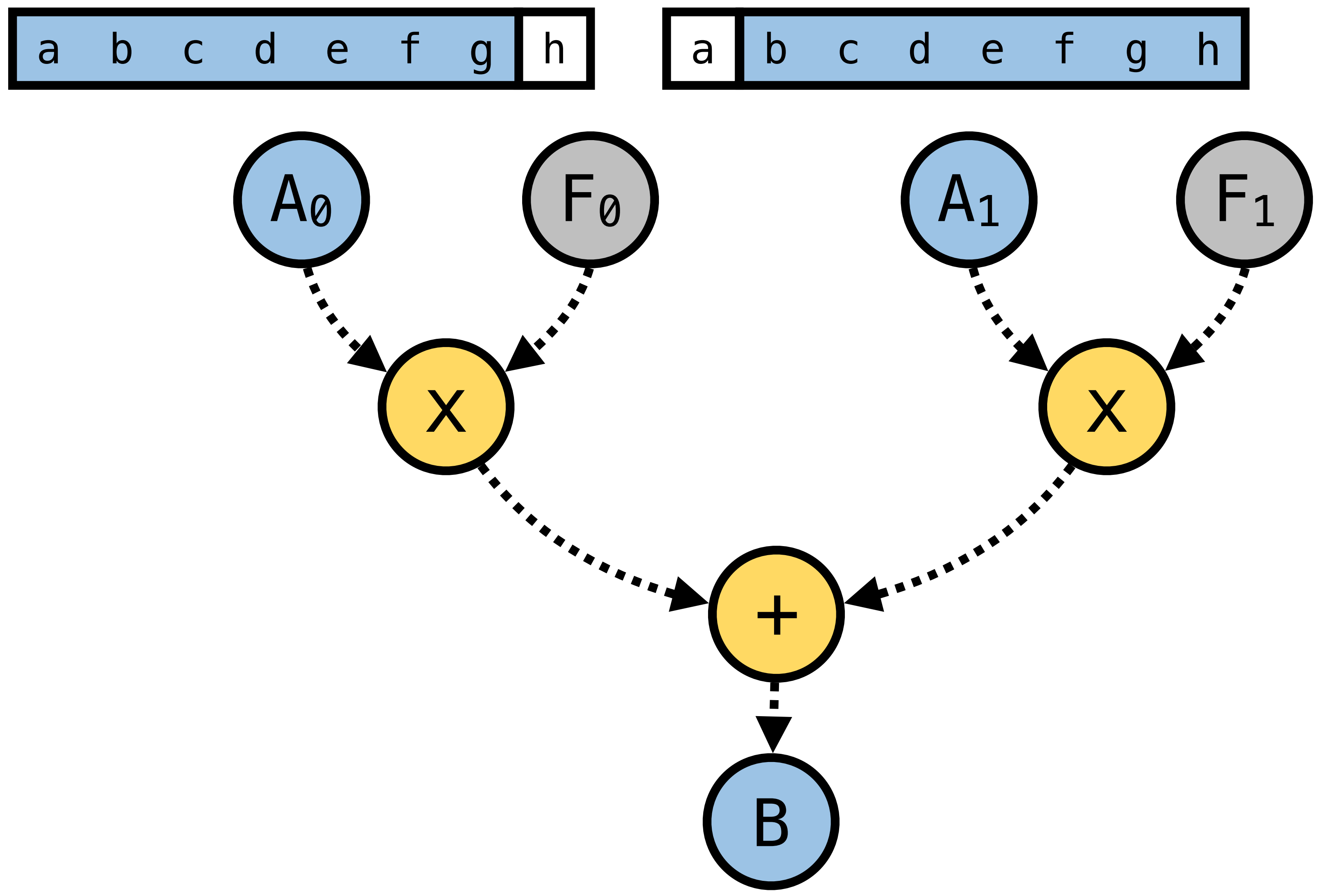
Tensor Dataflow Graph

- Memory access pattern
- Computation
- N-D Tensors
- Data alignment
- Reduction operations
- Spatial reuse
- Fusion

Observation: contiguous memory access may be tensorized

1D Filter

```
for i in [0, N-1):
  B[i] = F[0] x A[i  ]
       + F[1] x A[i+1]
```



But we need to map *data* to *hardware*

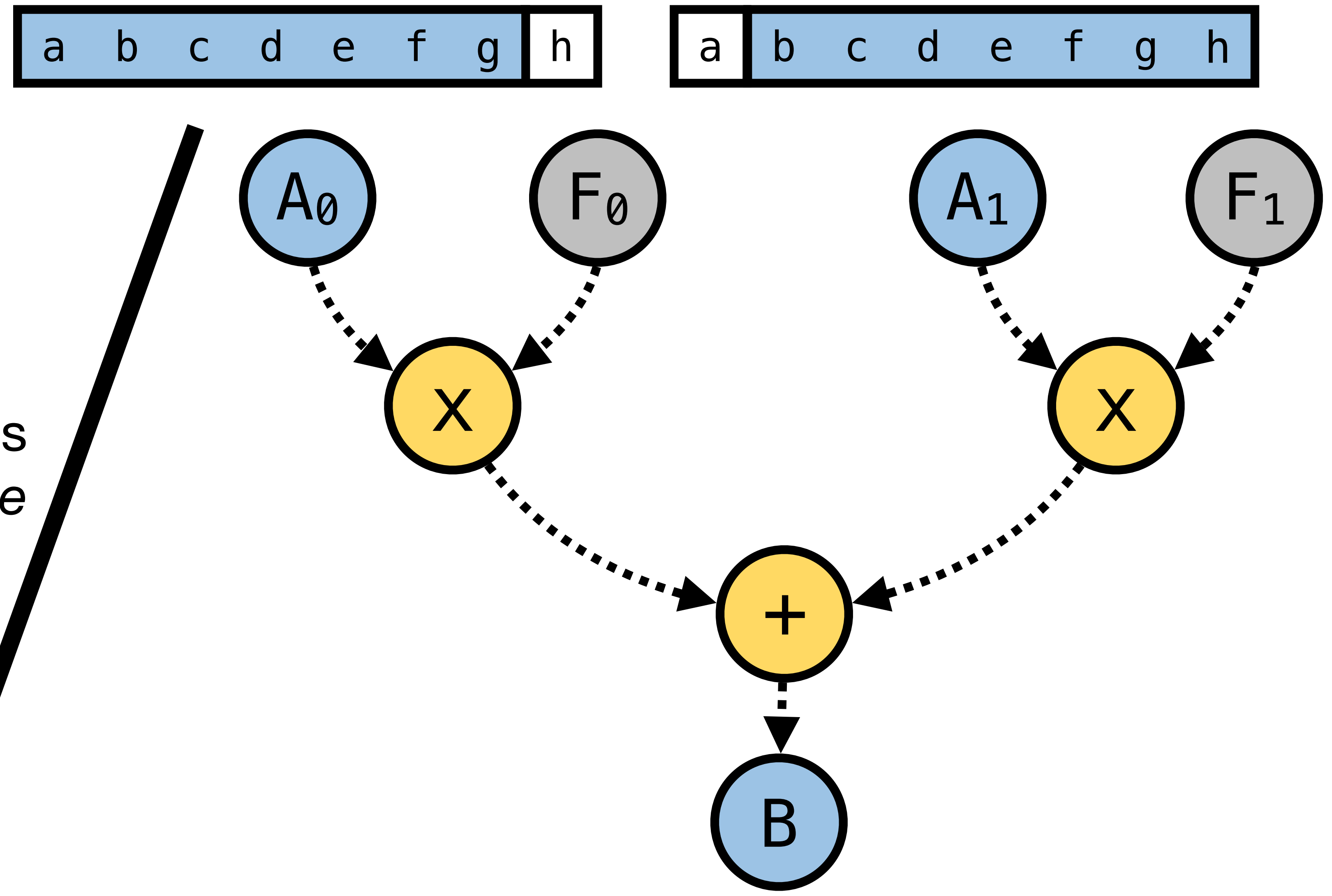
Tensor Dataflow Graph

- Memory access pattern
- Computation
- N-D Tensors
- Data alignment
- Reduction operations
- Spatial reuse
- Fusion

1D Filter

```
for i in [0, N-1):
  B[i] = F[0] x A[i  ]

       + F[1] x A[i+1]
```

Observation: contiguous memory access may be tensorized

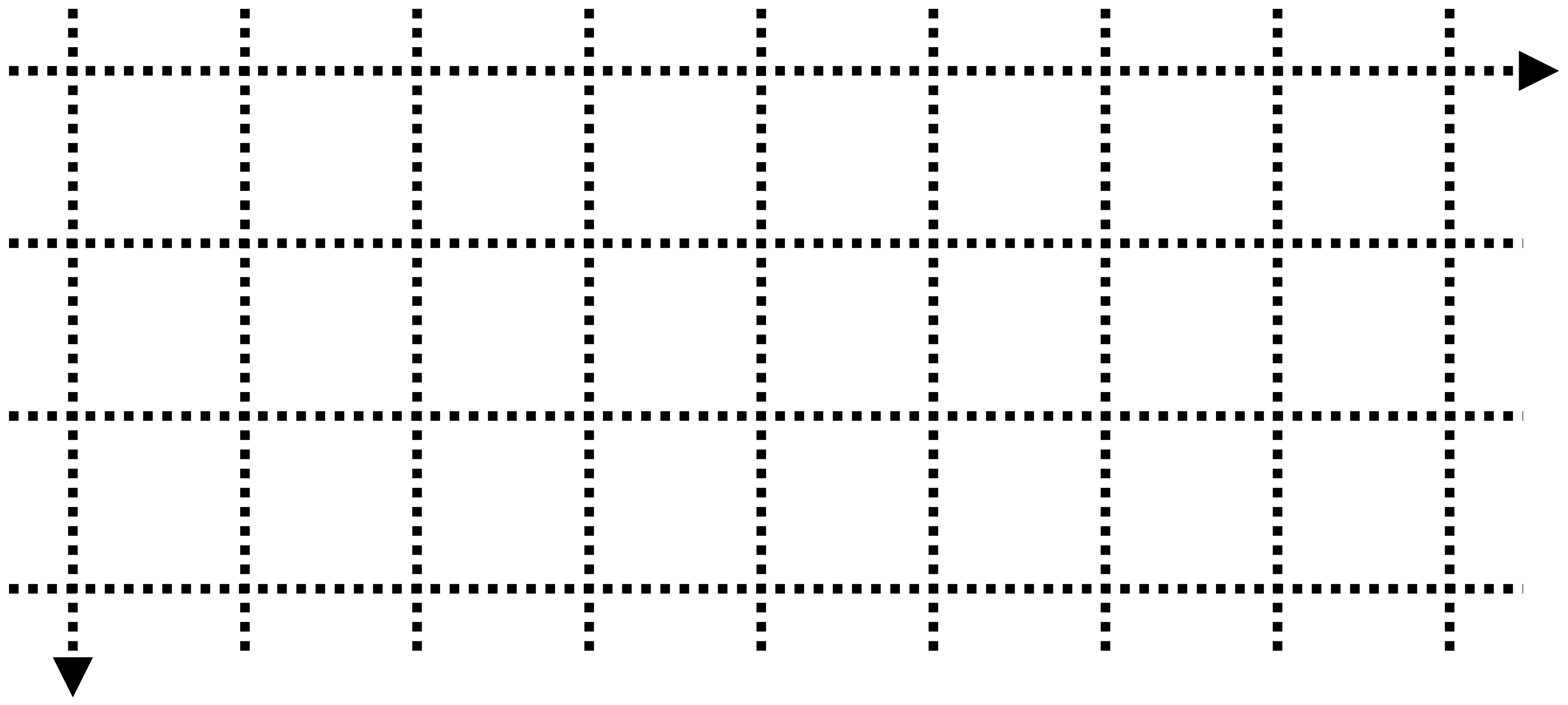


Maps
data → virtual vector lane

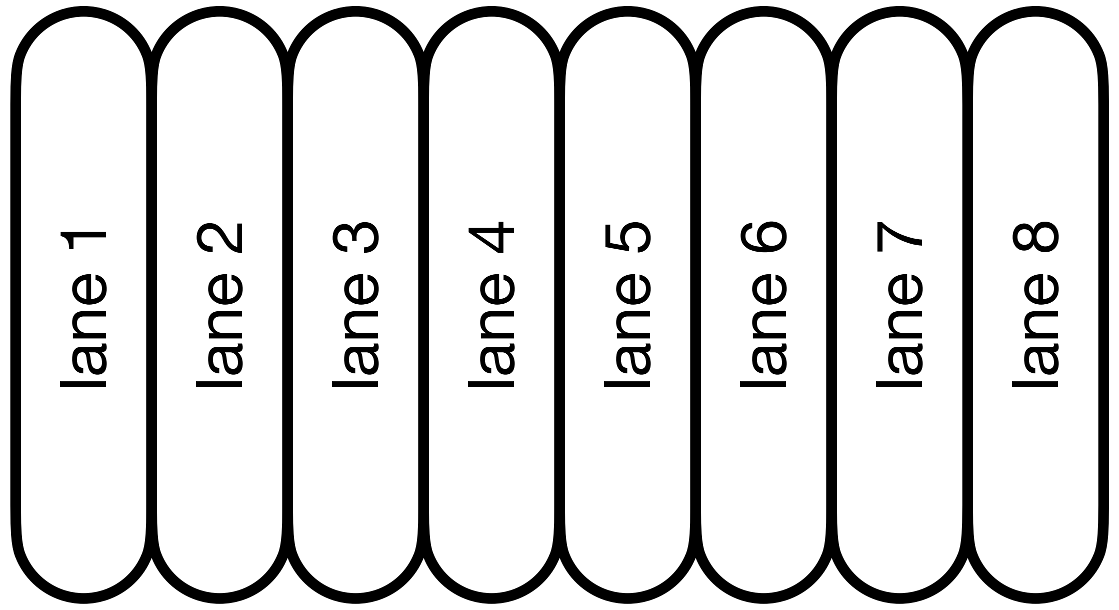
Global Lattice Space

Each cell represents a virtual vector lane

Describes N-dimensional alignment



Maps
virtual vector lane →
physical vector lane



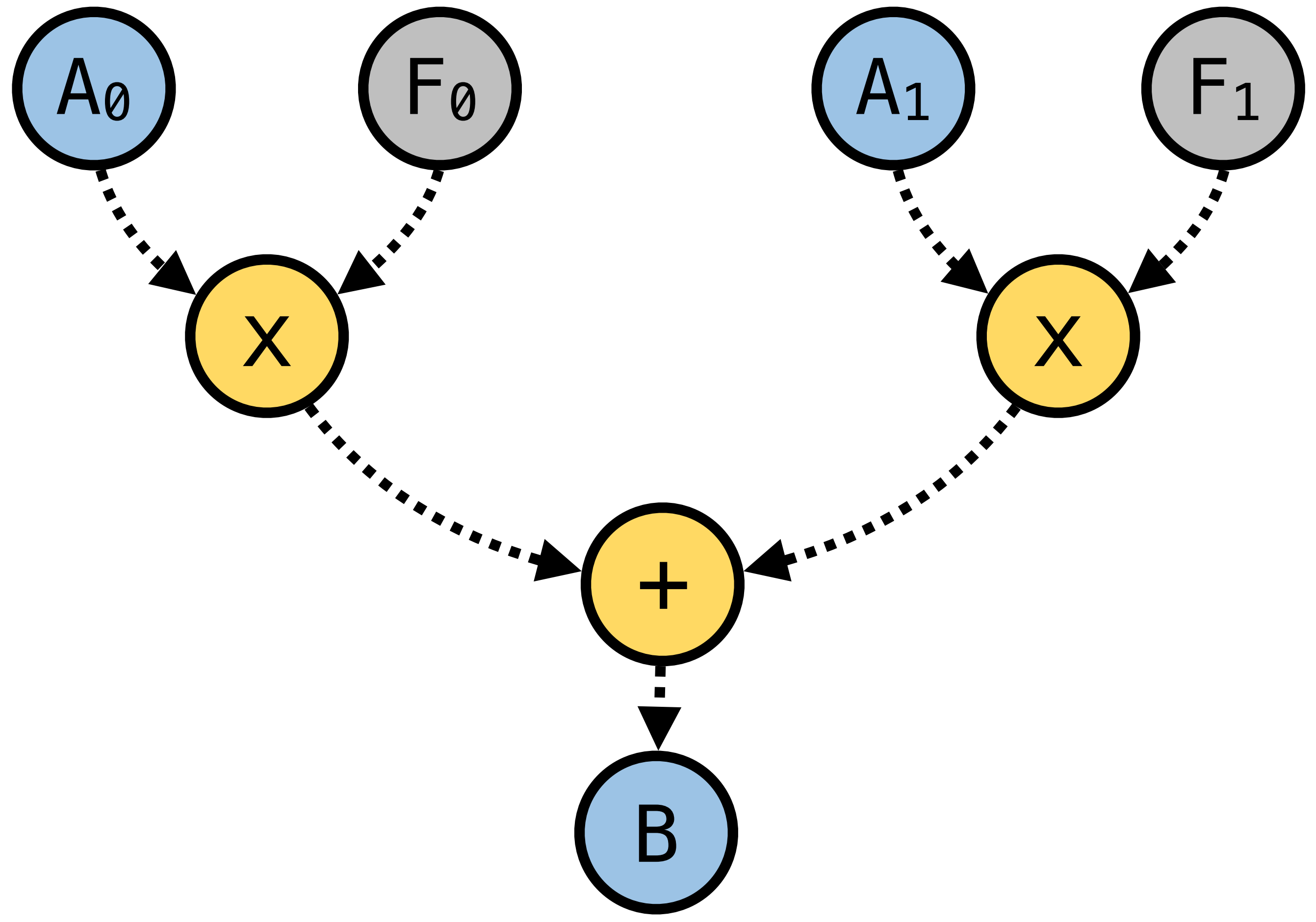
Hardware Vector Processor

Tensor Dataflow Graph

1D Filter

```
for i in [0, N-1):
  B[i] = F[0] x A[i  ]

       + F[1] x A[i+1]
```

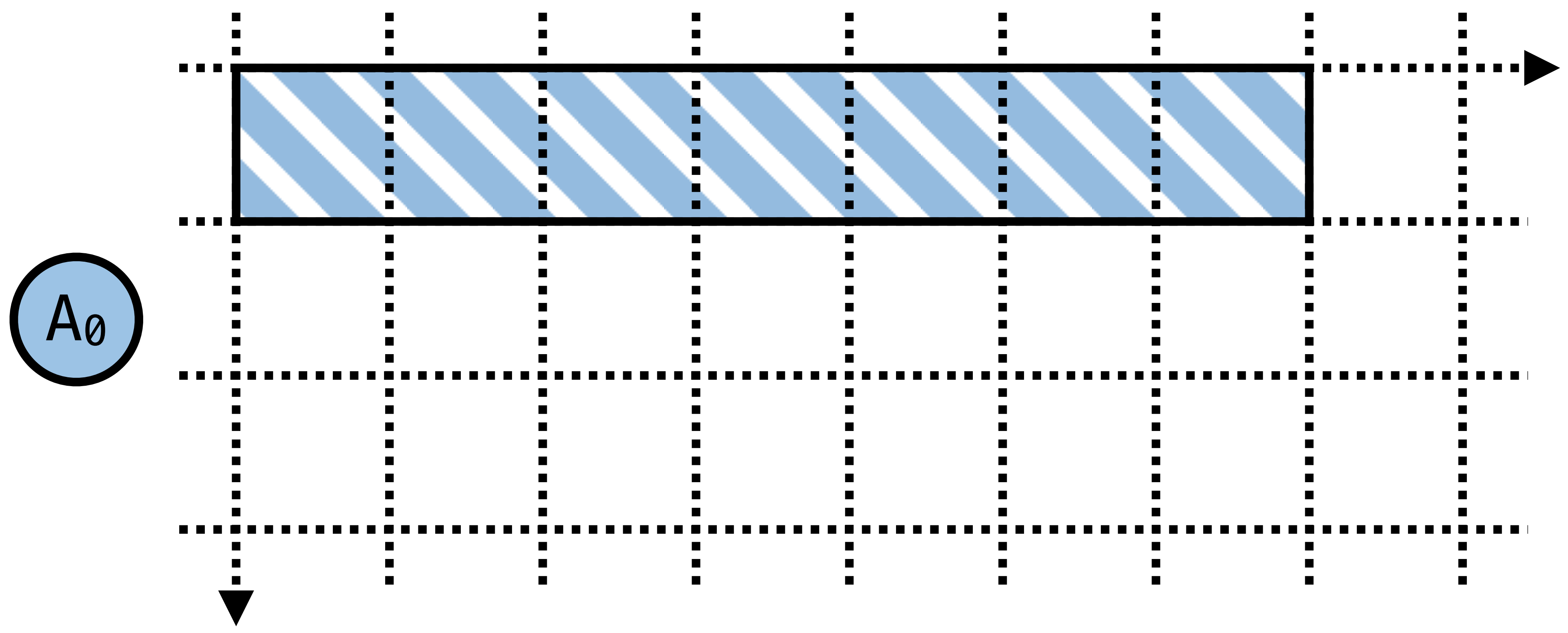
Memory access is represented as a hyperrectangle

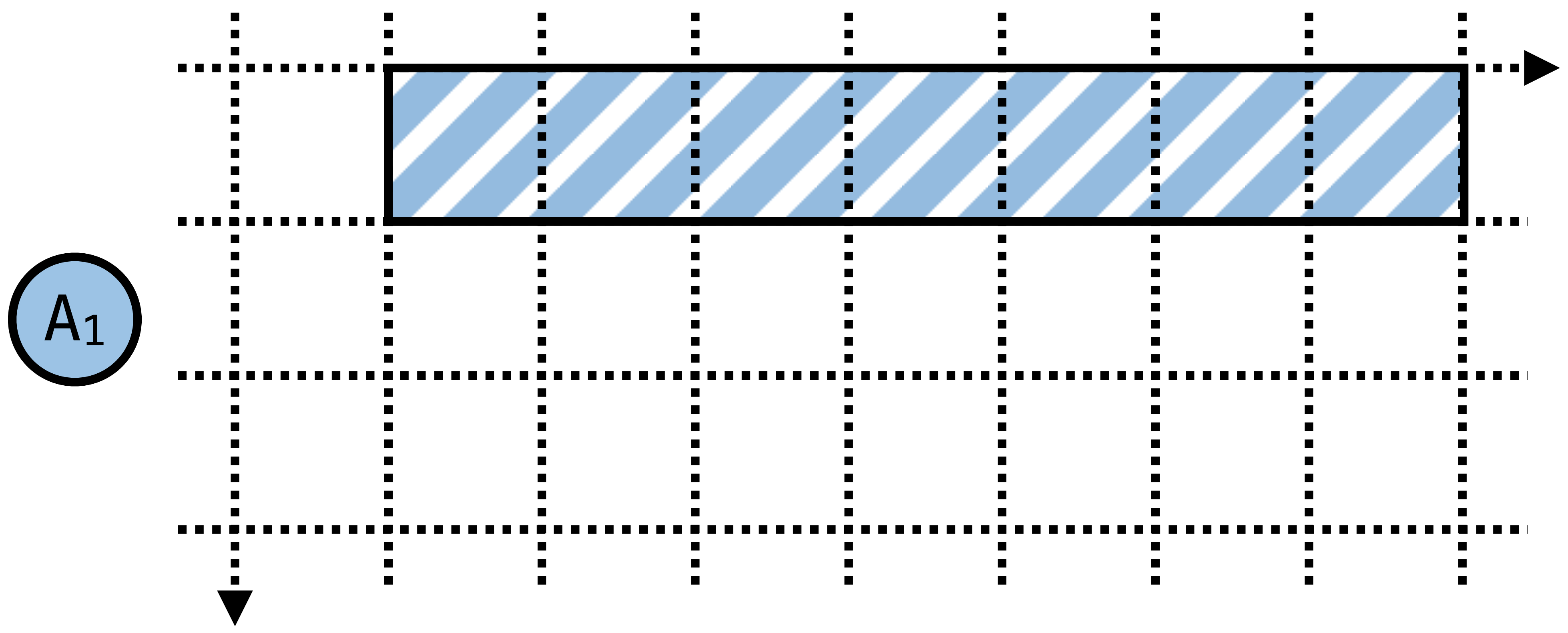


Global Lattice Space

Each cell represents a virtual vector lane

Describes N-dimensional alignment



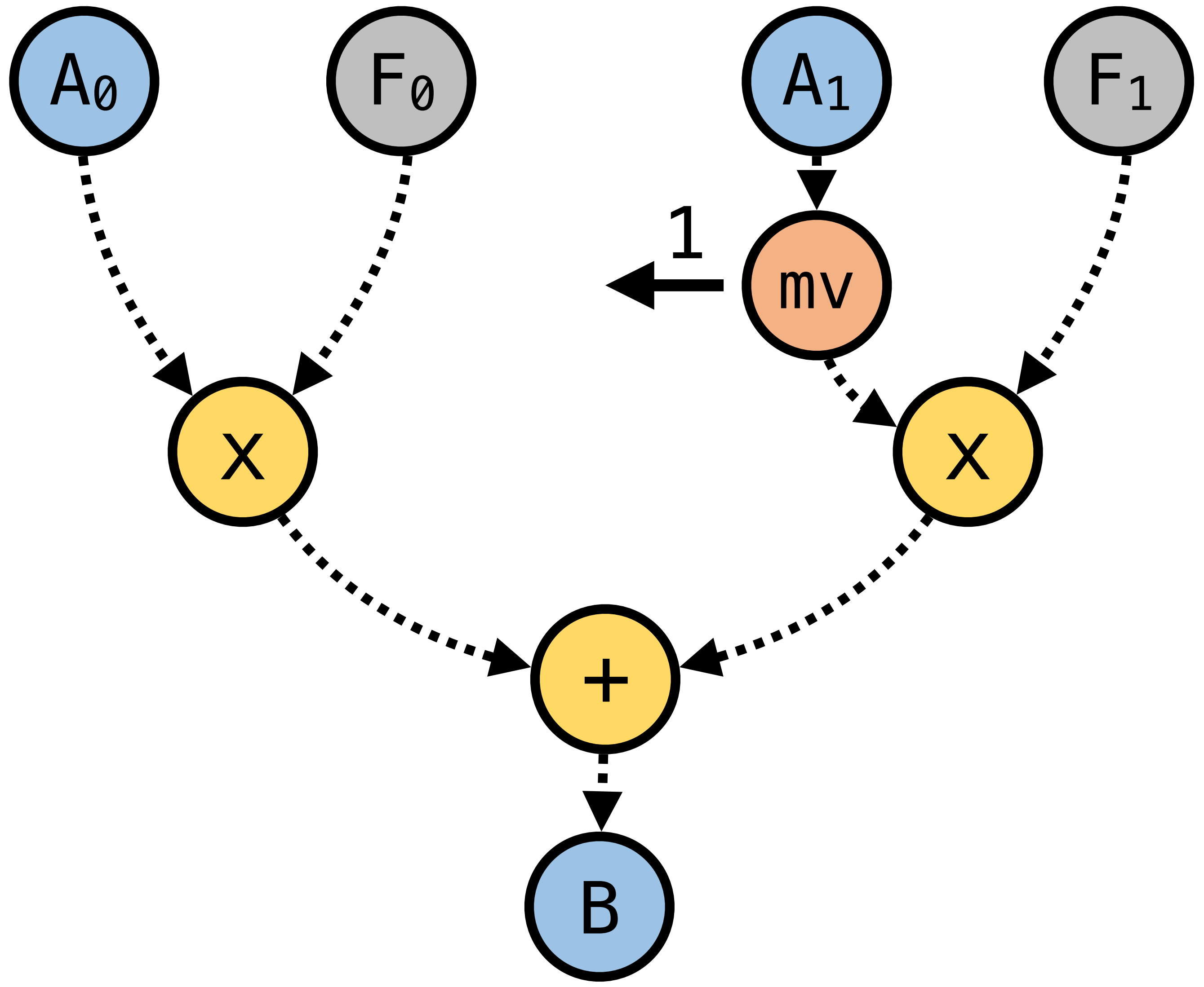


Tensor Dataflow Graph

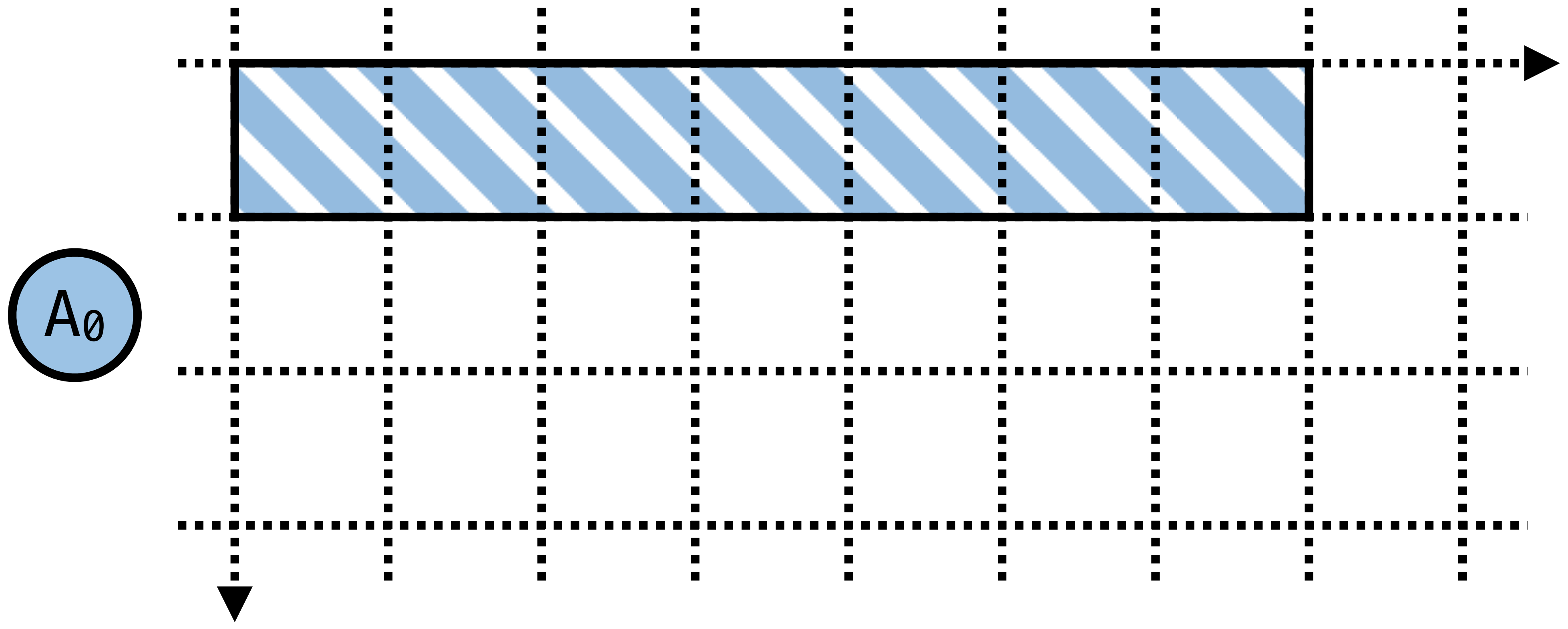
1D Filter

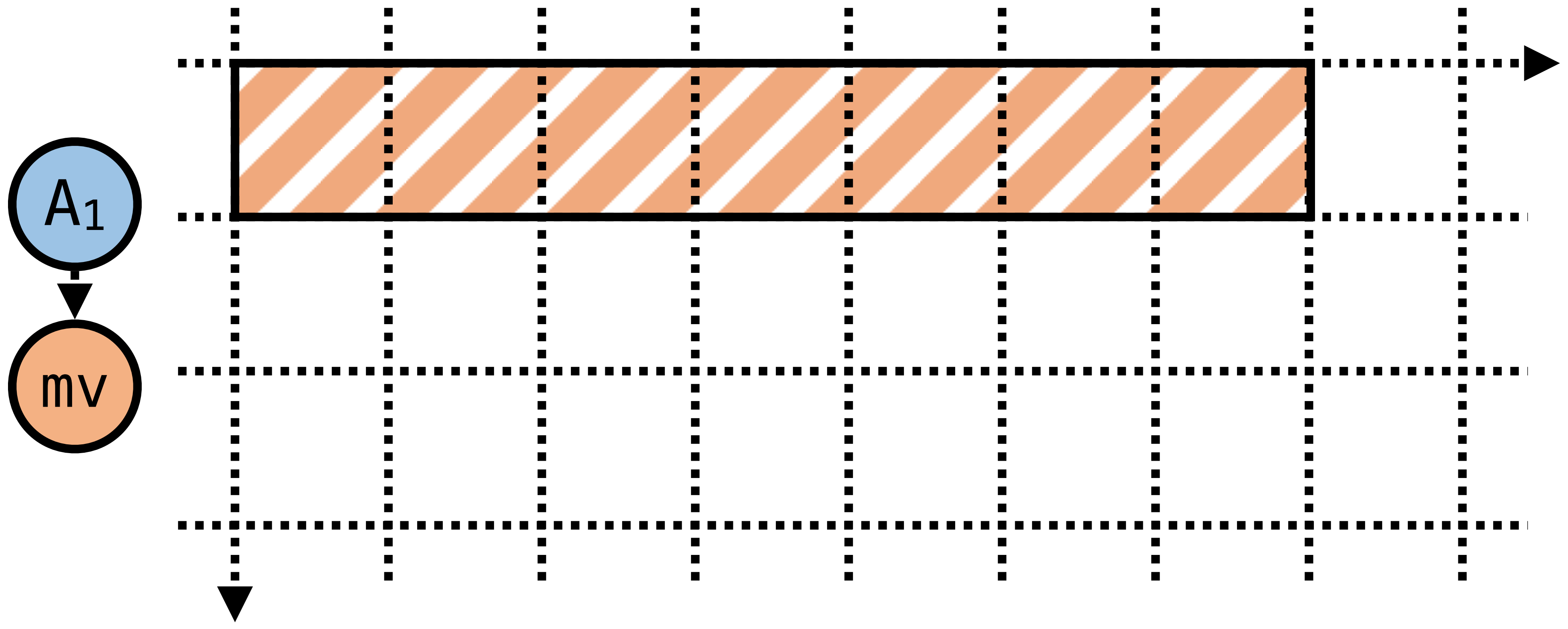
```
for i in [0, N-1):
  B[i] = F[0] x A[i  ]

       + F[1] x A[i+1]
```

Memory access is represented as a hyperrectangle



Move Node mv realigns hyperrectangle



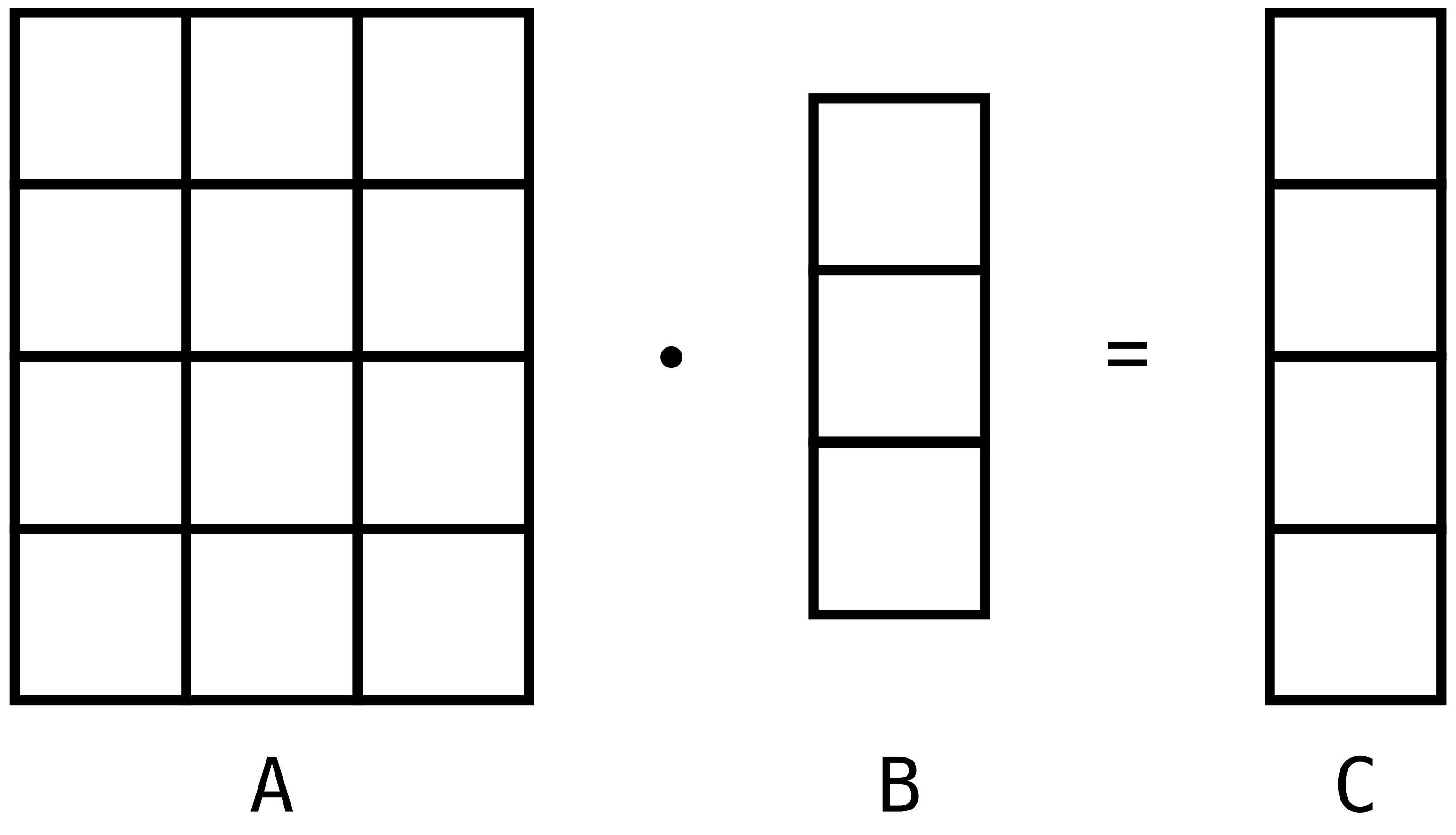


Matrix-Vector Multiplication

- Memory access pattern
- Computation
- N-D Tensors
- Data alignment
- Reduction operations
- Spatial reuse
- Fusion

```
for m in [0, M):
  for k in [0, K):
    C[m] += A[m][k] * B[k]
```

Example



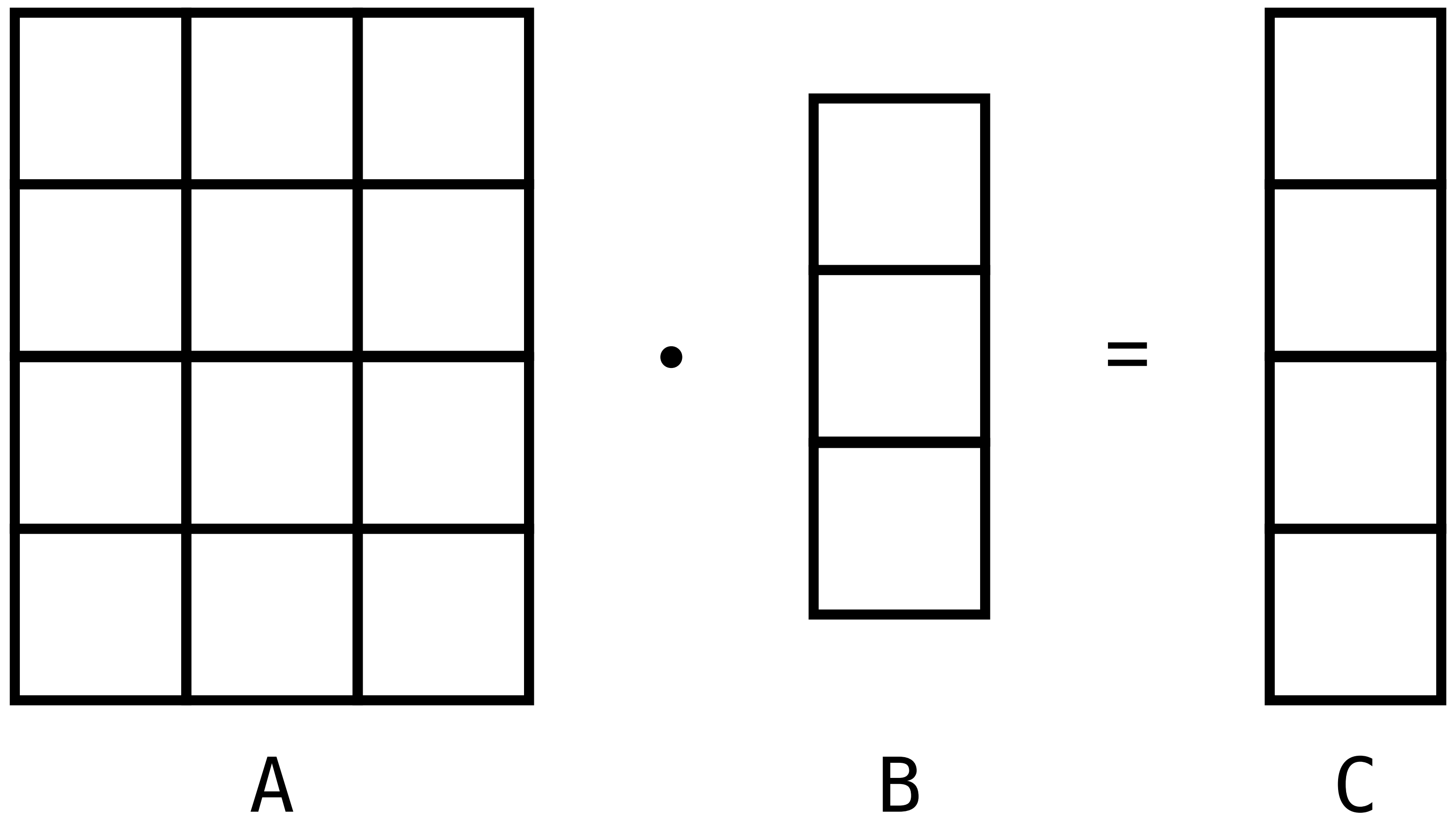
Matrix-Vector Multiplication

- Memory access pattern
- Computation
- N-D Tensors
- Data alignment

```
for m in [0, M):
  C[m] = sum(A[m][:] * B[k])
```

Vectorize

Example



- Reduction operations

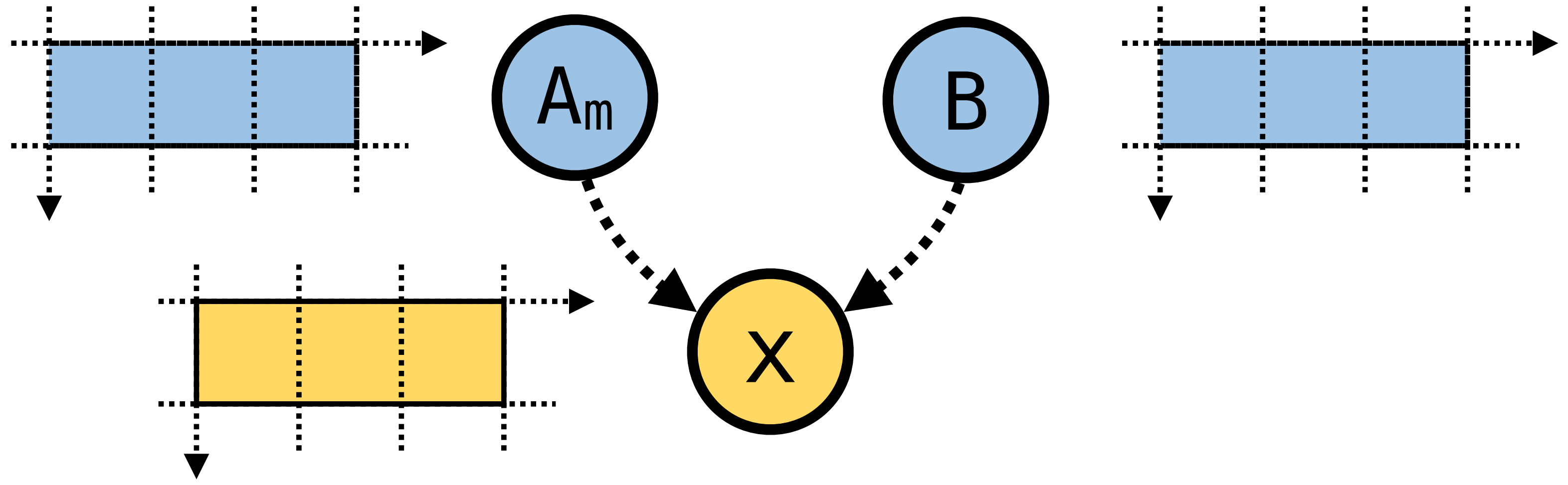
- Spatial reuse
- Fusion

Matrix-Vector Multiplication

- Memory access pattern
- Computation
- N-D Tensors
- Data alignment

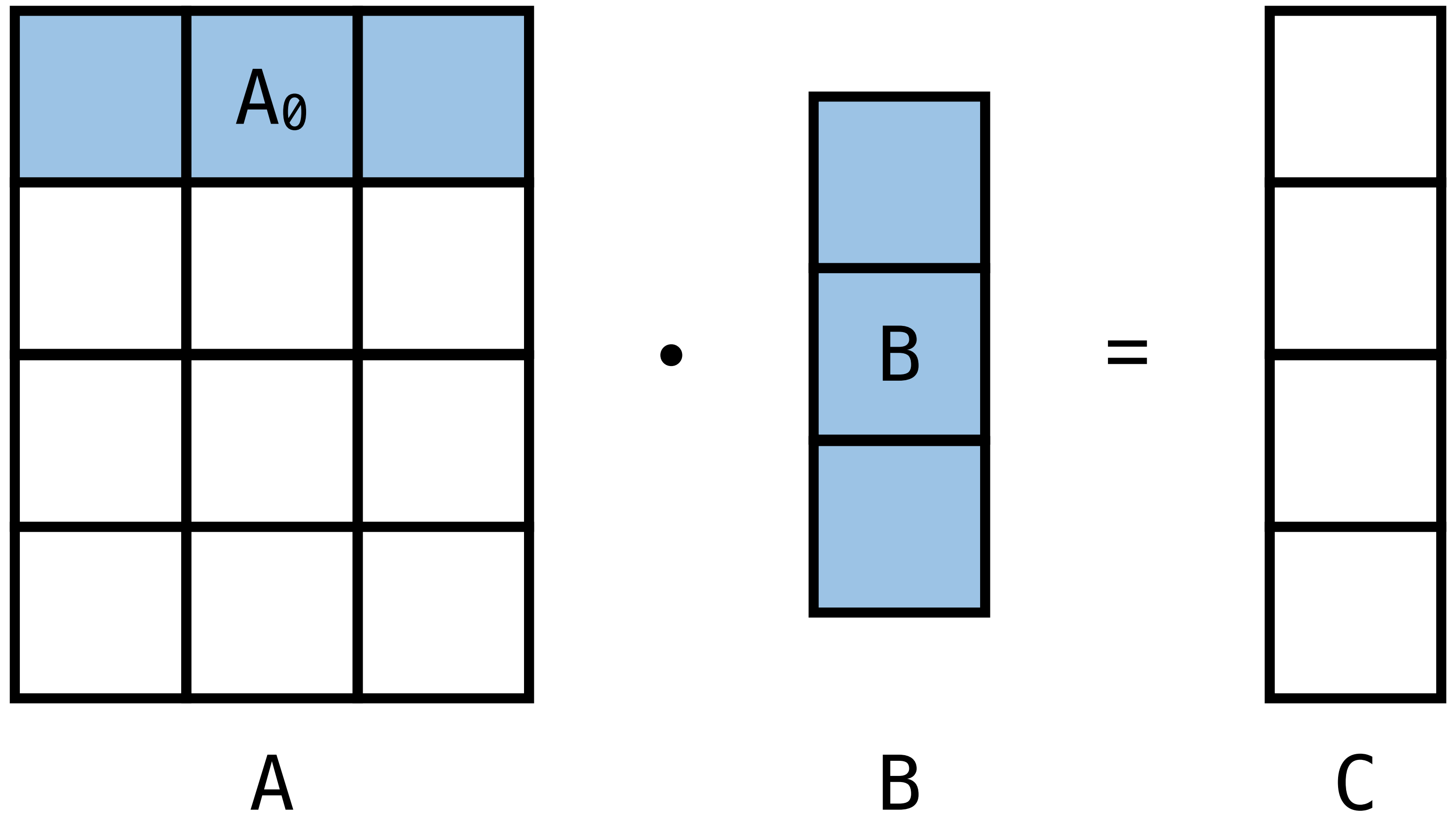
```
for m in [0, M):
  C[m] = sum(A[m][:] * B[k])
```

Spatial



- Reduction operations

Example



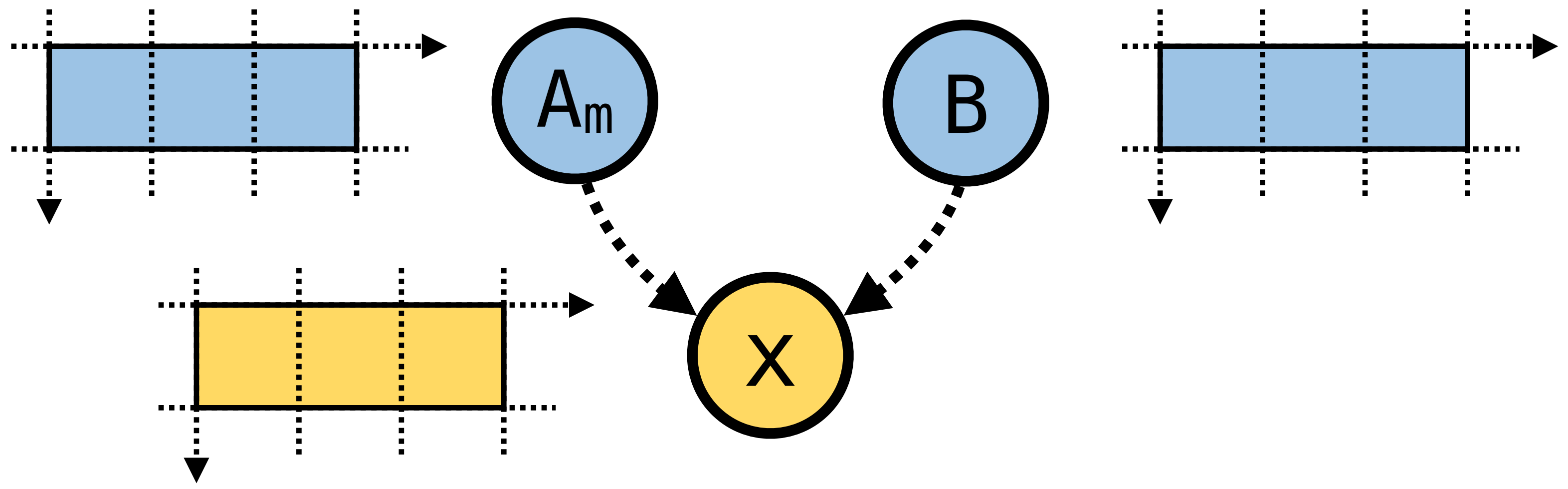
- Spatial reuse
- Fusion

Matrix-Vector Multiplication

- Memory access pattern
- Computation
- N-D Tensors
- Data alignment

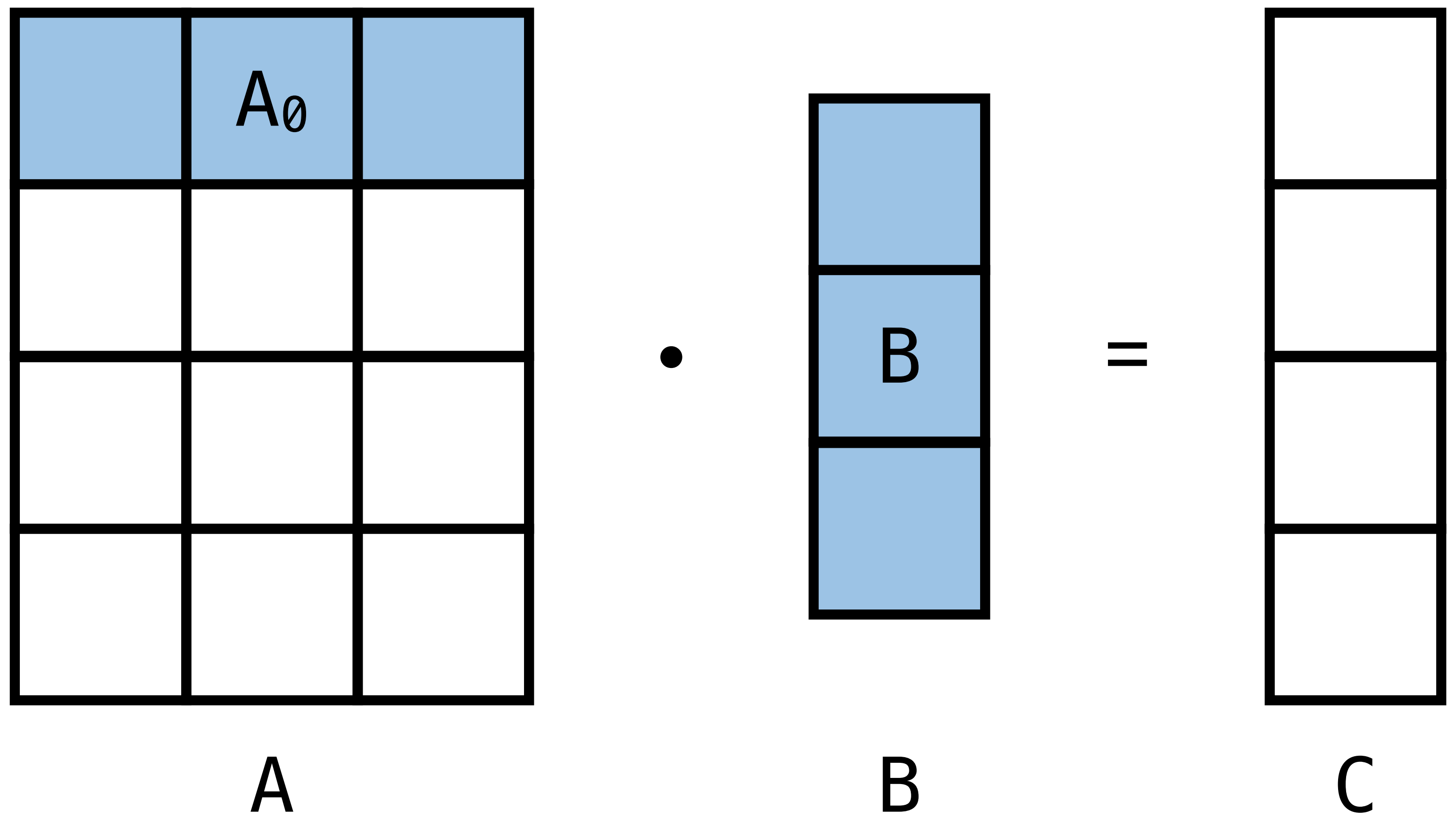
```
for m in [0, M):
  C[m] = sum(A[m][:] * B[k])
```

Spatial



- Reduction operations

Example



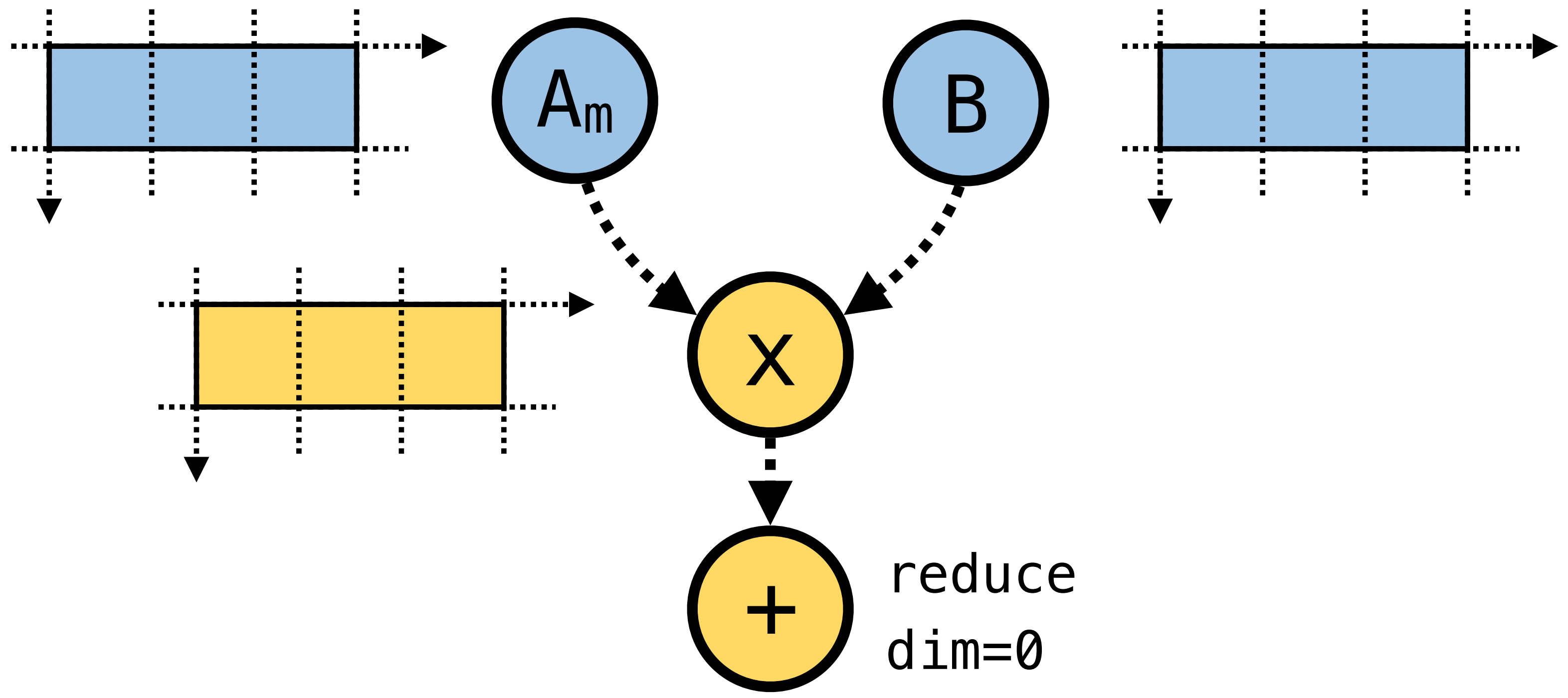
- Spatial reuse
- Fusion

Matrix-Vector Multiplication

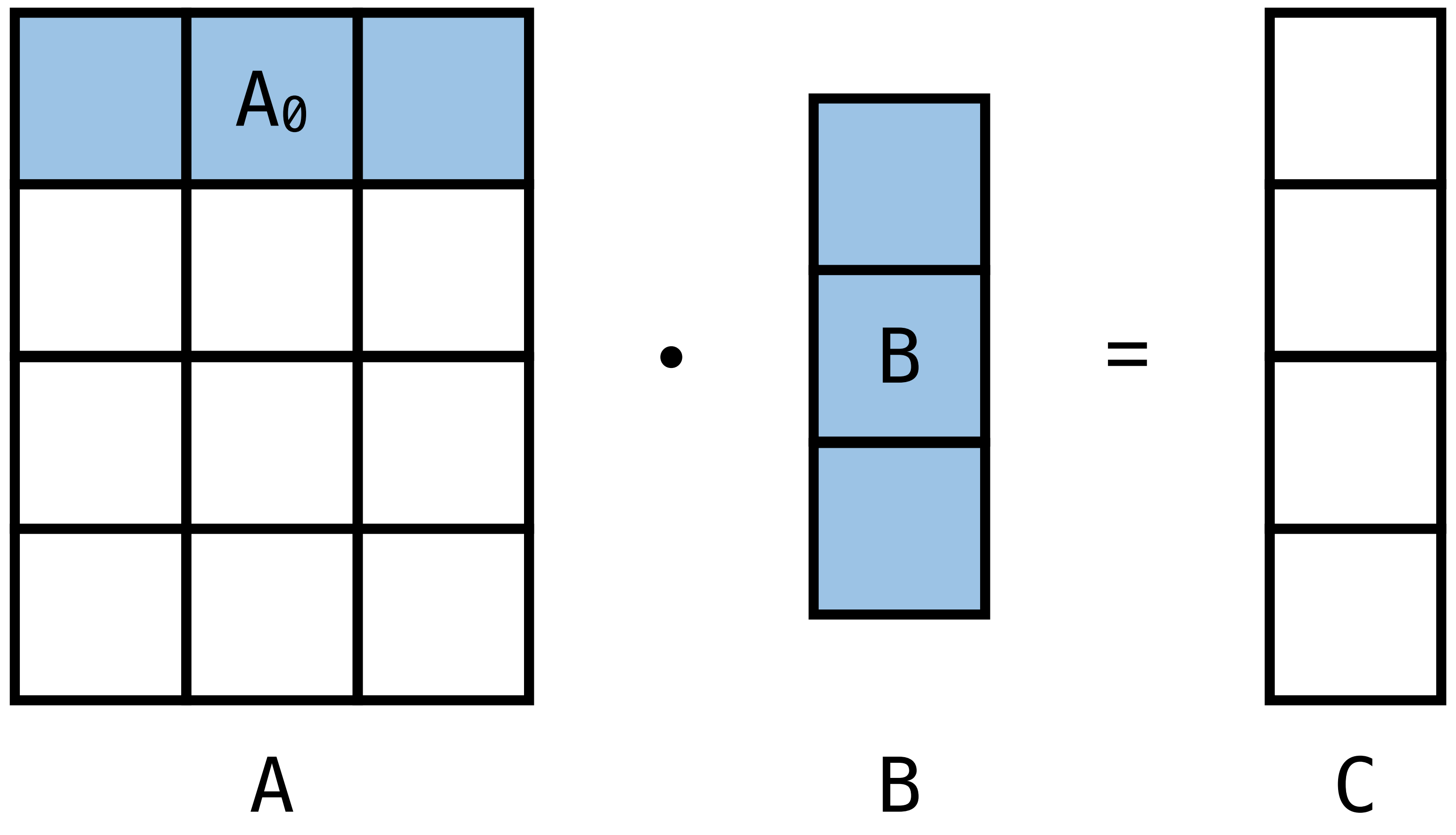
- Memory access pattern
- Computation
- N-D Tensors
- Data alignment

```
for m in [0, M):
    C[m] = sum(A[m][:] * B[k])
```

Spatial



Example



- Reduction operations

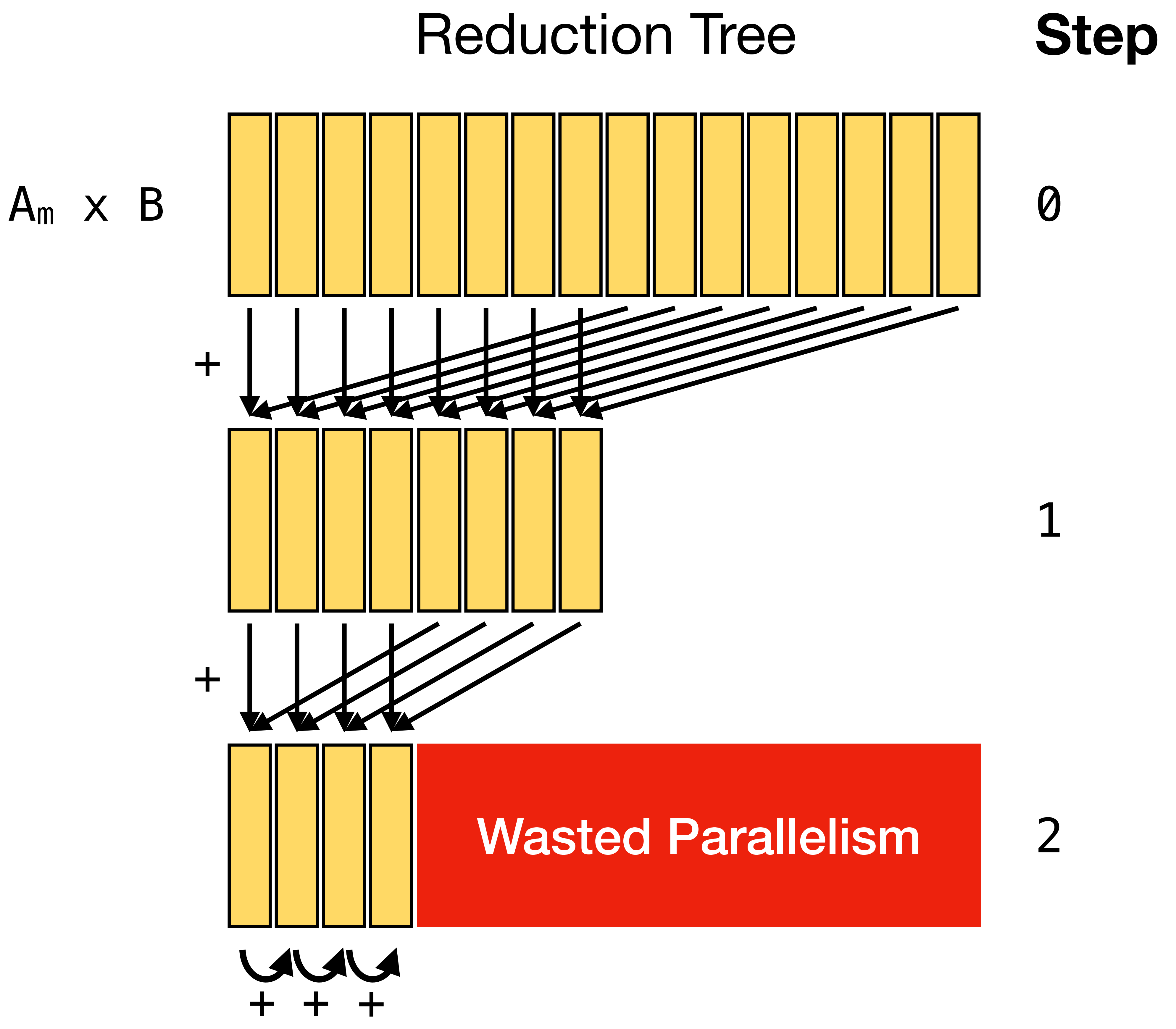
- Spatial reuse
- Fusion

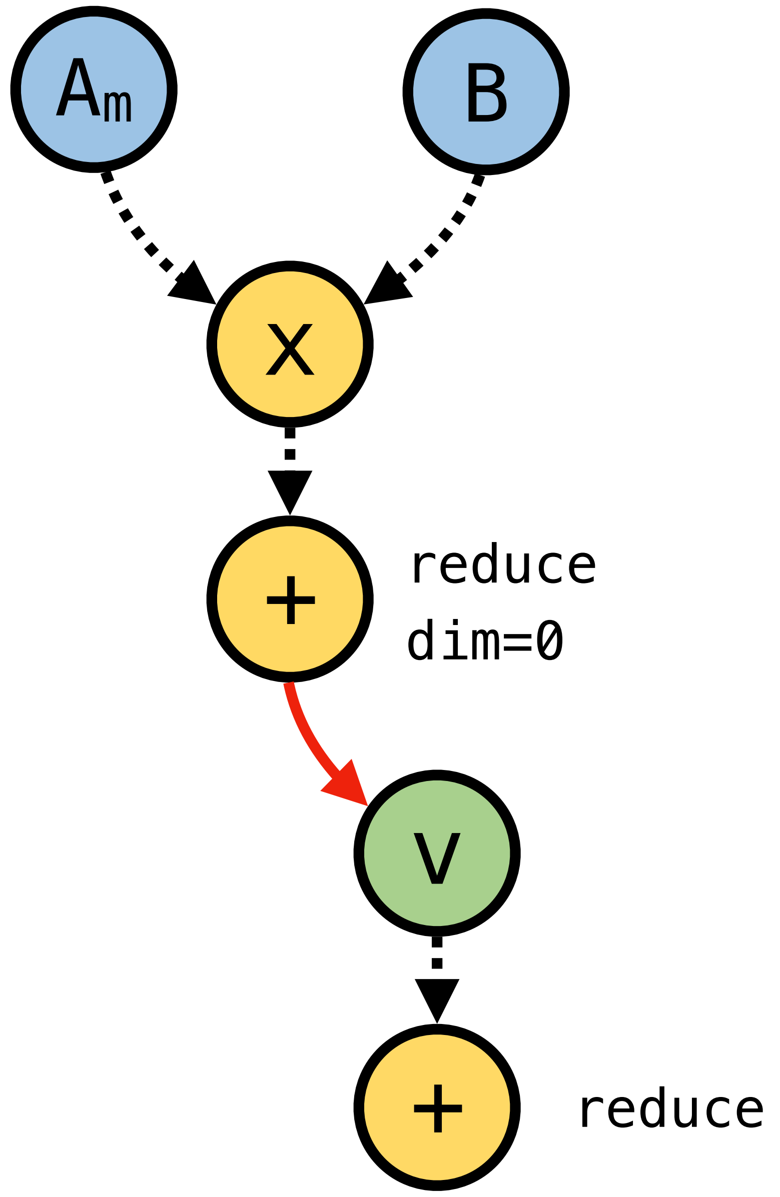
Matrix-Vector Multiplication

- Memory access pattern
- Computation
- N-D Tensors
- Data alignment
- Reduction operations
- Spatial reuse
- Fusion

Spatial

```
for m in [0, M):
    C[m] = sum(A[m][:] * B[k])
```





Insufficient parallelism

Use near-memory to perform reduction

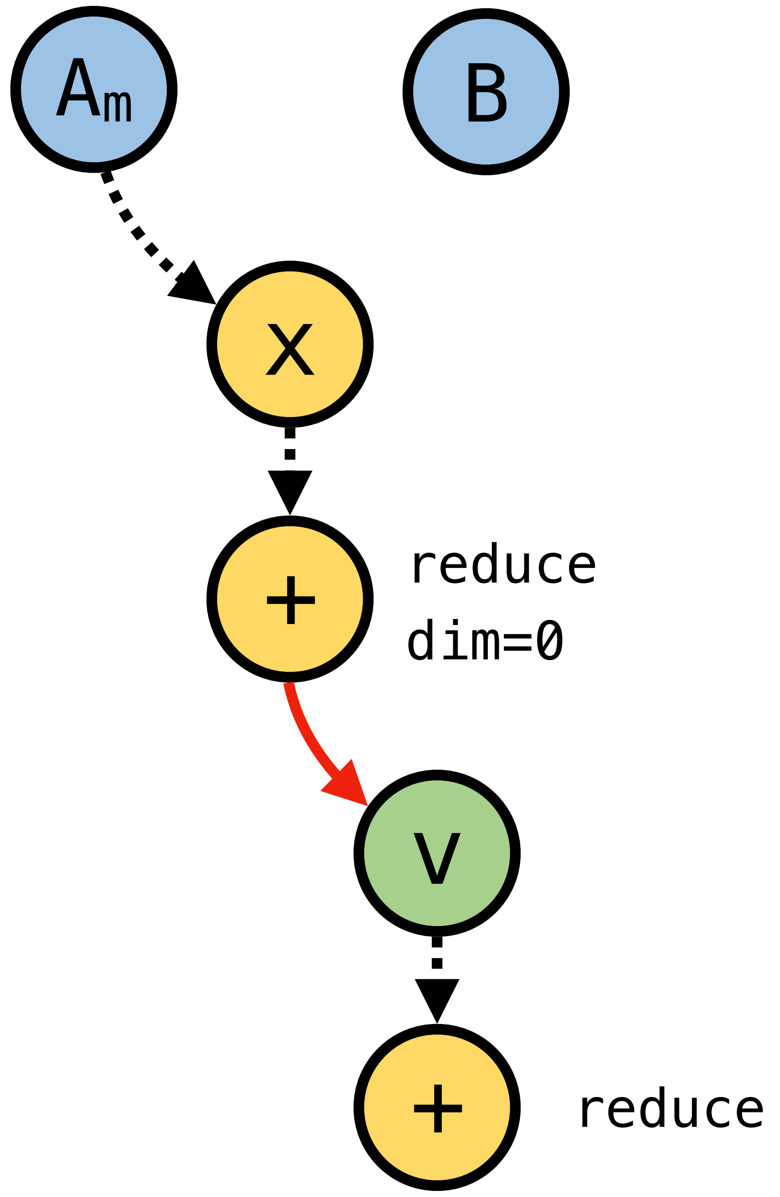
Matrix-Vector Multiplication

- Memory access pattern
- Computation
- N-D Tensors
- Data alignment
- Reduction operations
- Spatial reuse
- Fusion

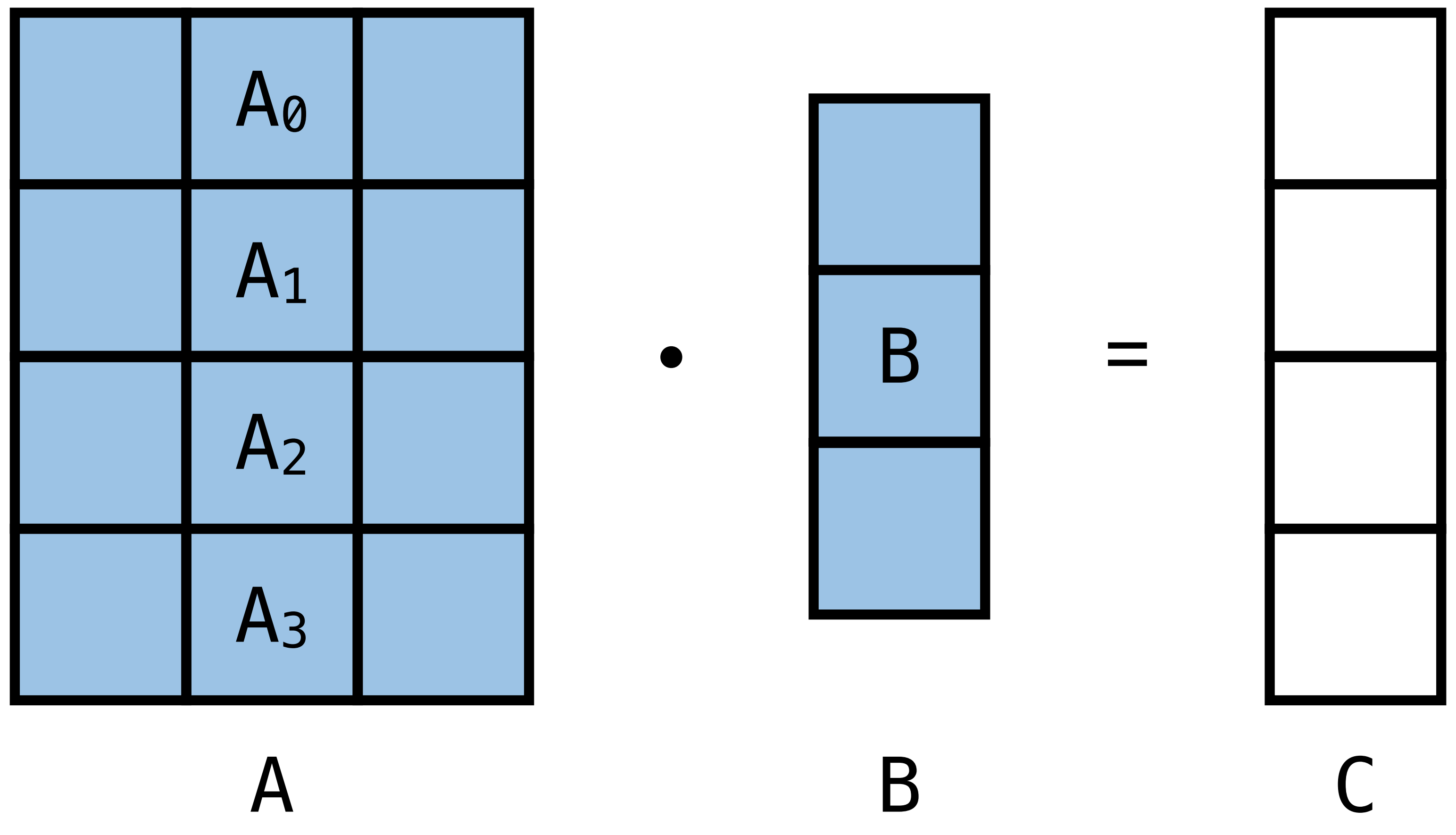
Spatial

```
for m in [0, M):
  C[m] = sum(A[m][:] * B[k])
```

Idea: Expose more parallelism through spatial reuse



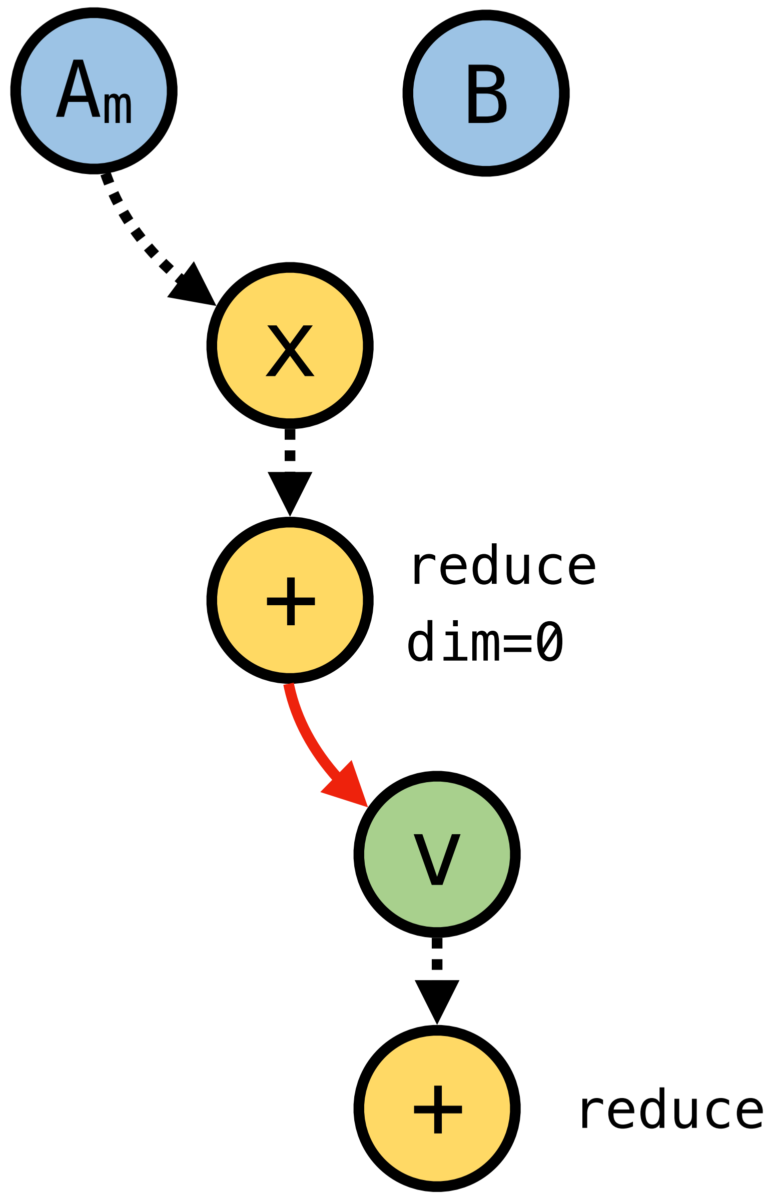
Example



Matrix-Vector Multiplication

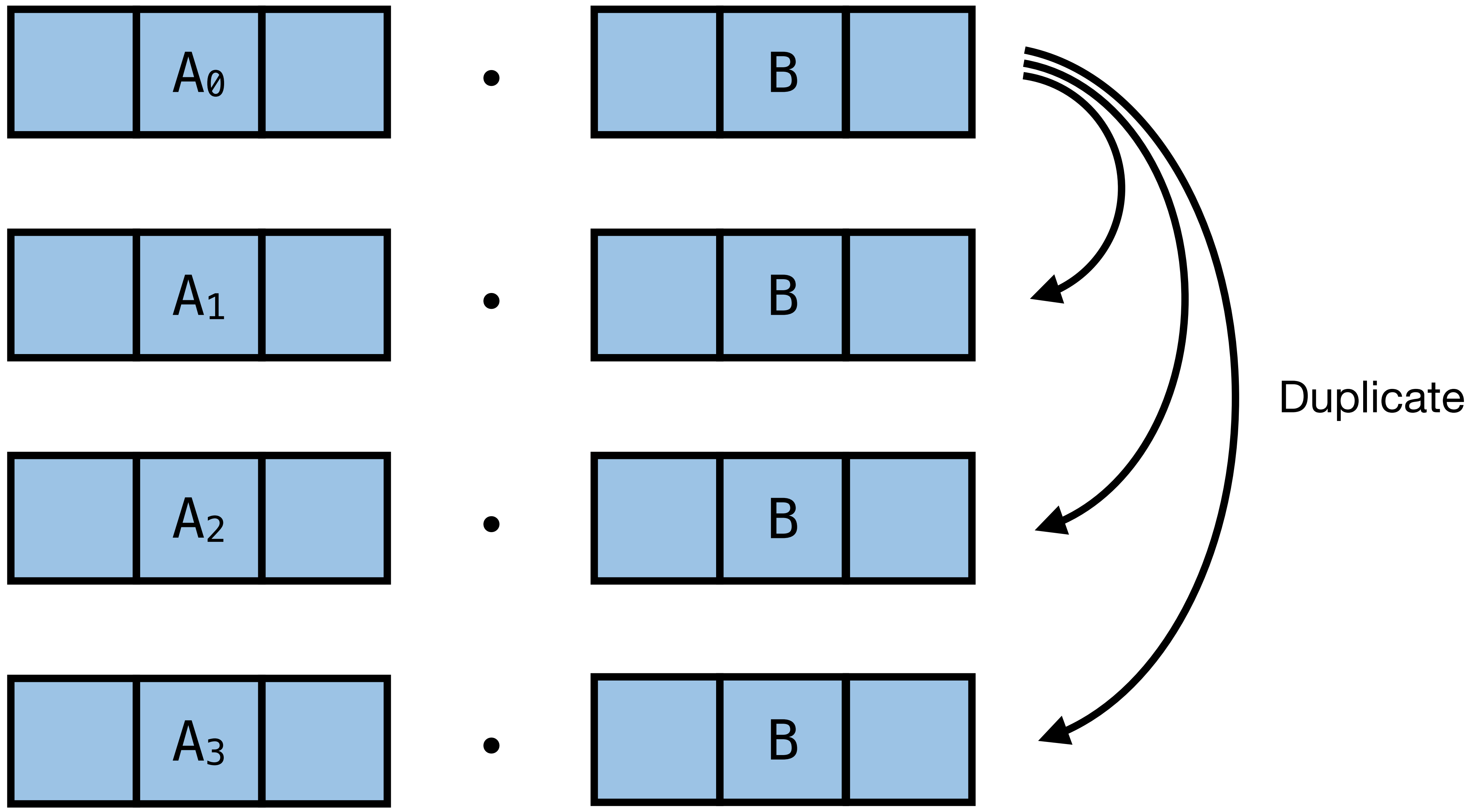
Spatial

```
for m in [0, M):
  C[m] = sum(A[m][:] * B[k])
```



Idea: Expose more parallelism through spatial reuse

Example



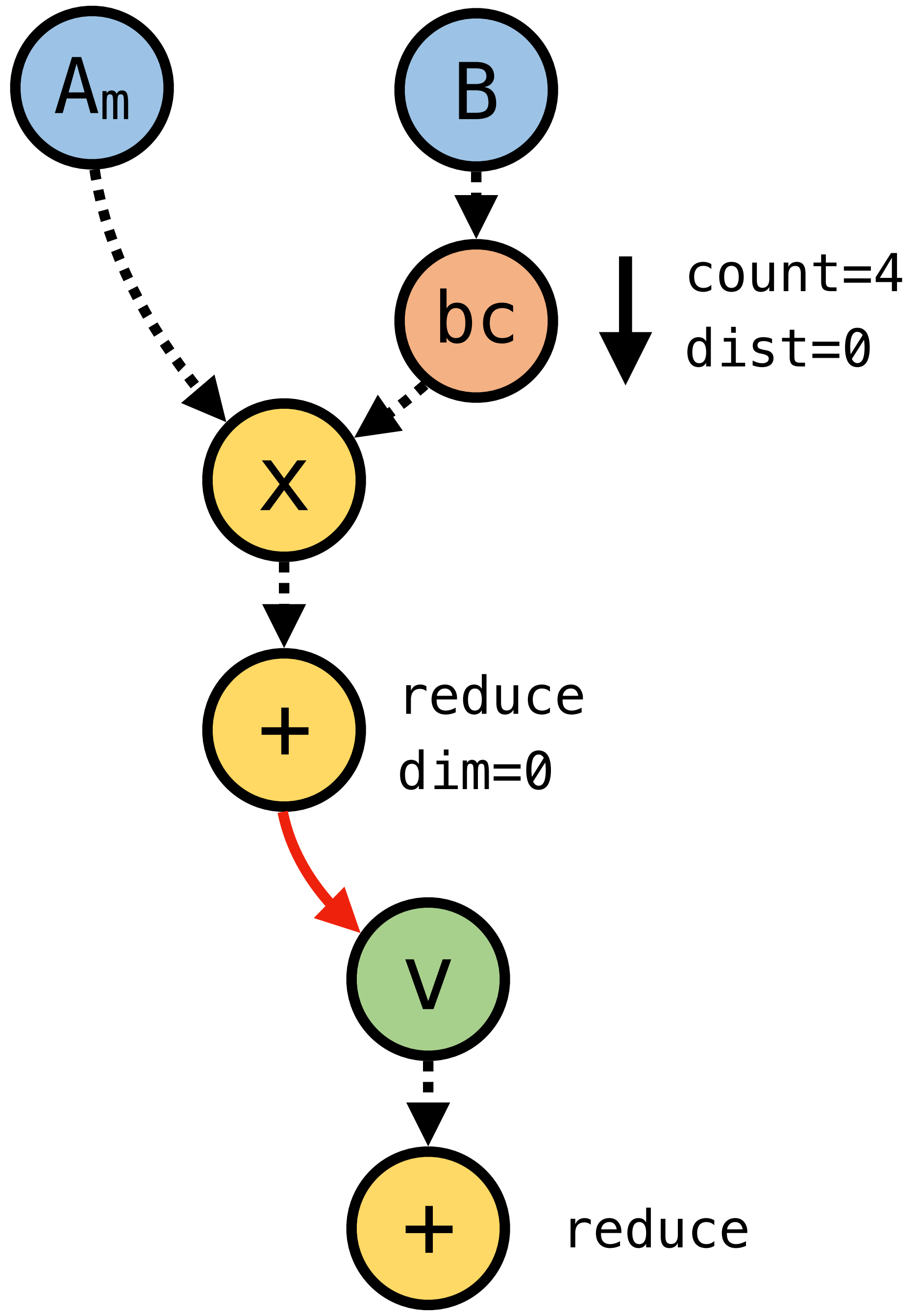
Matrix-Vector Multiplication

- Memory access pattern
- Computation
- N-D Tensors
- Data alignment
- Reduction operations
- Spatial reuse
- Fusion

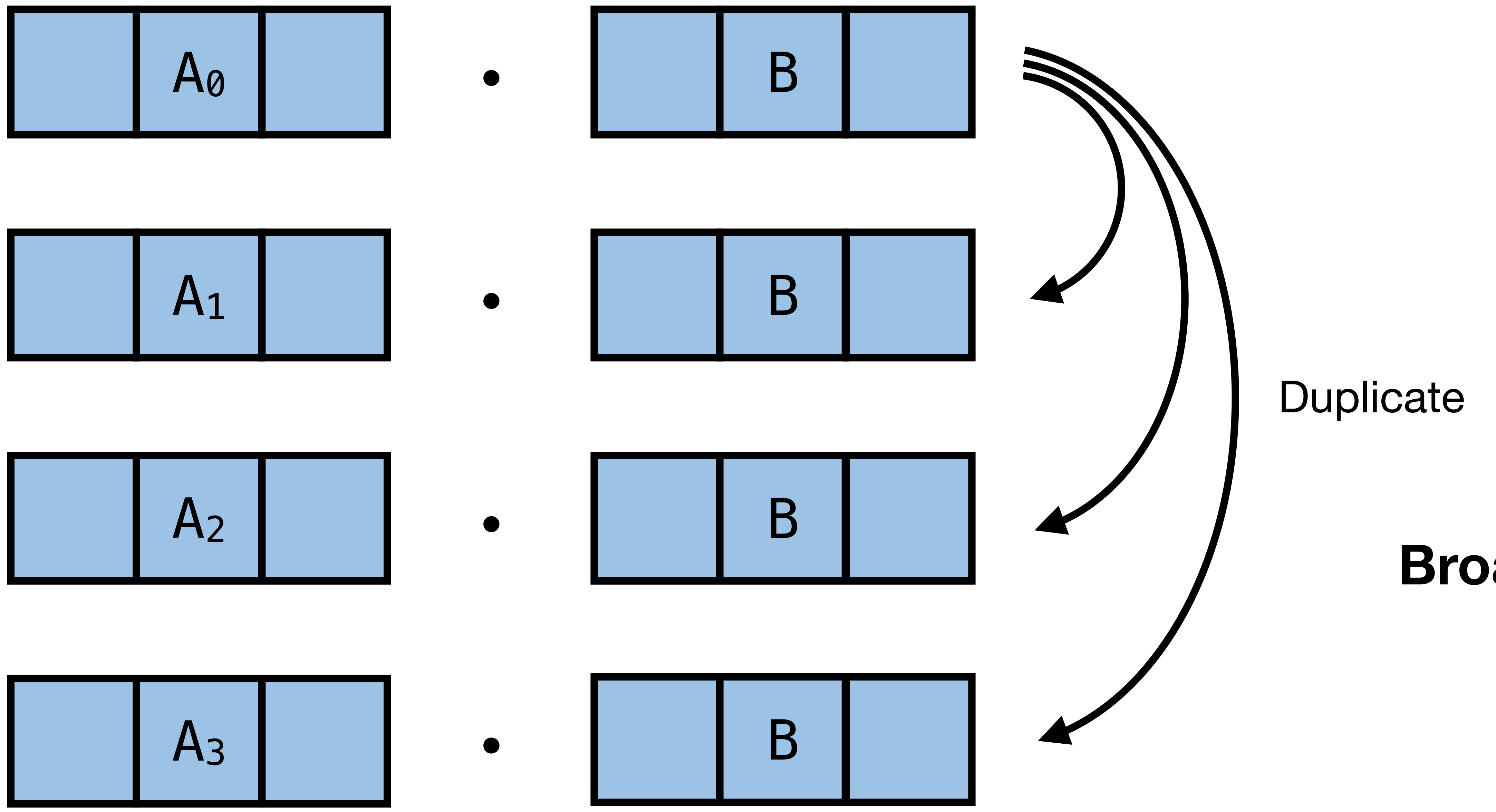
Spatial

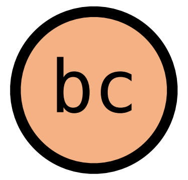
```
for m in [0, M):
  C[m] = sum(A[m][:] * B[k])
```

Idea: Expose more parallelism through spatial reuse



Example



Broadcast Node 

captures spatial reuse

Matrix-Vector Multiplication

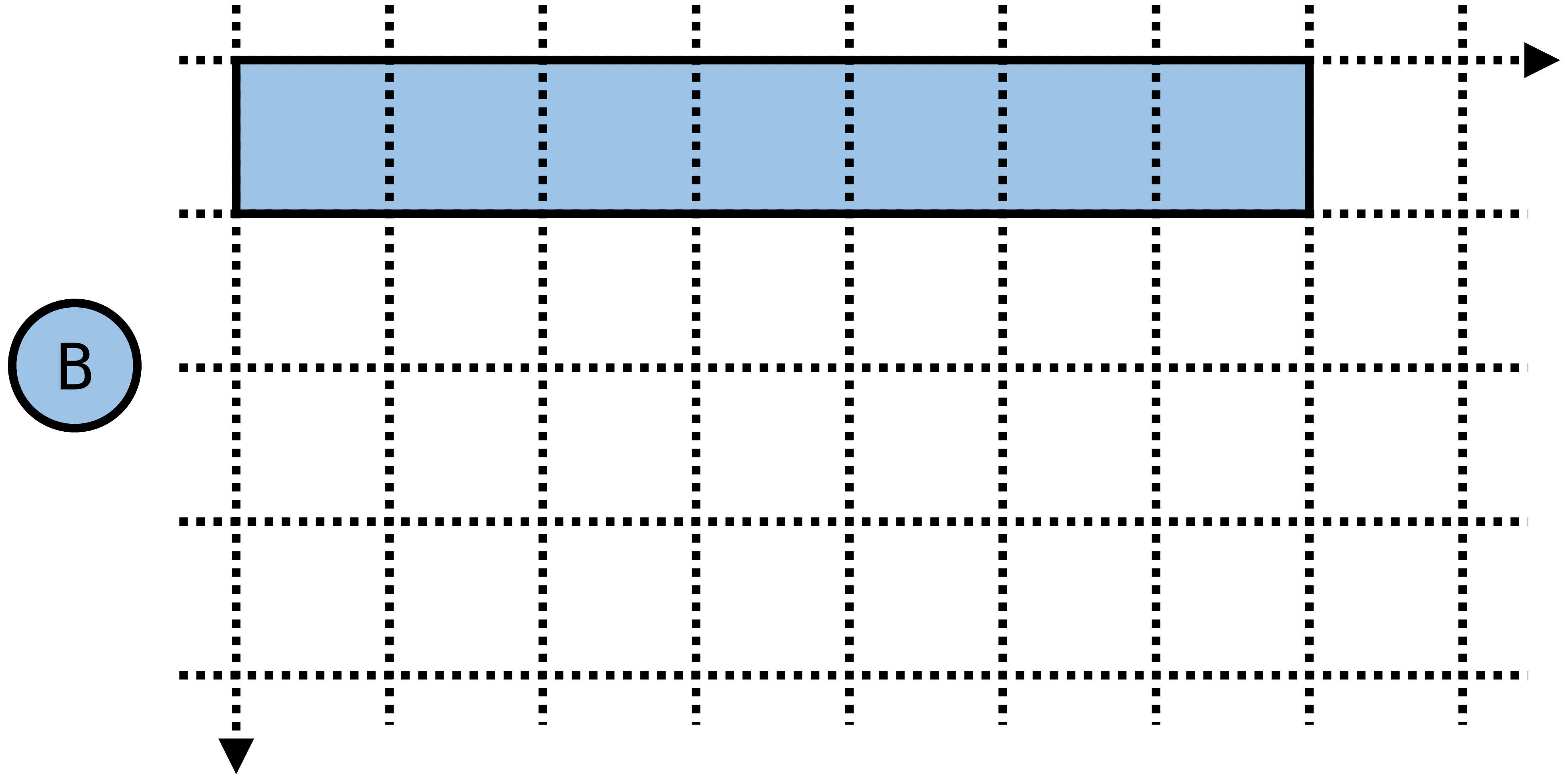
- Memory access pattern
- Computation
- N-D Tensors
- Data alignment
- Reduction operations

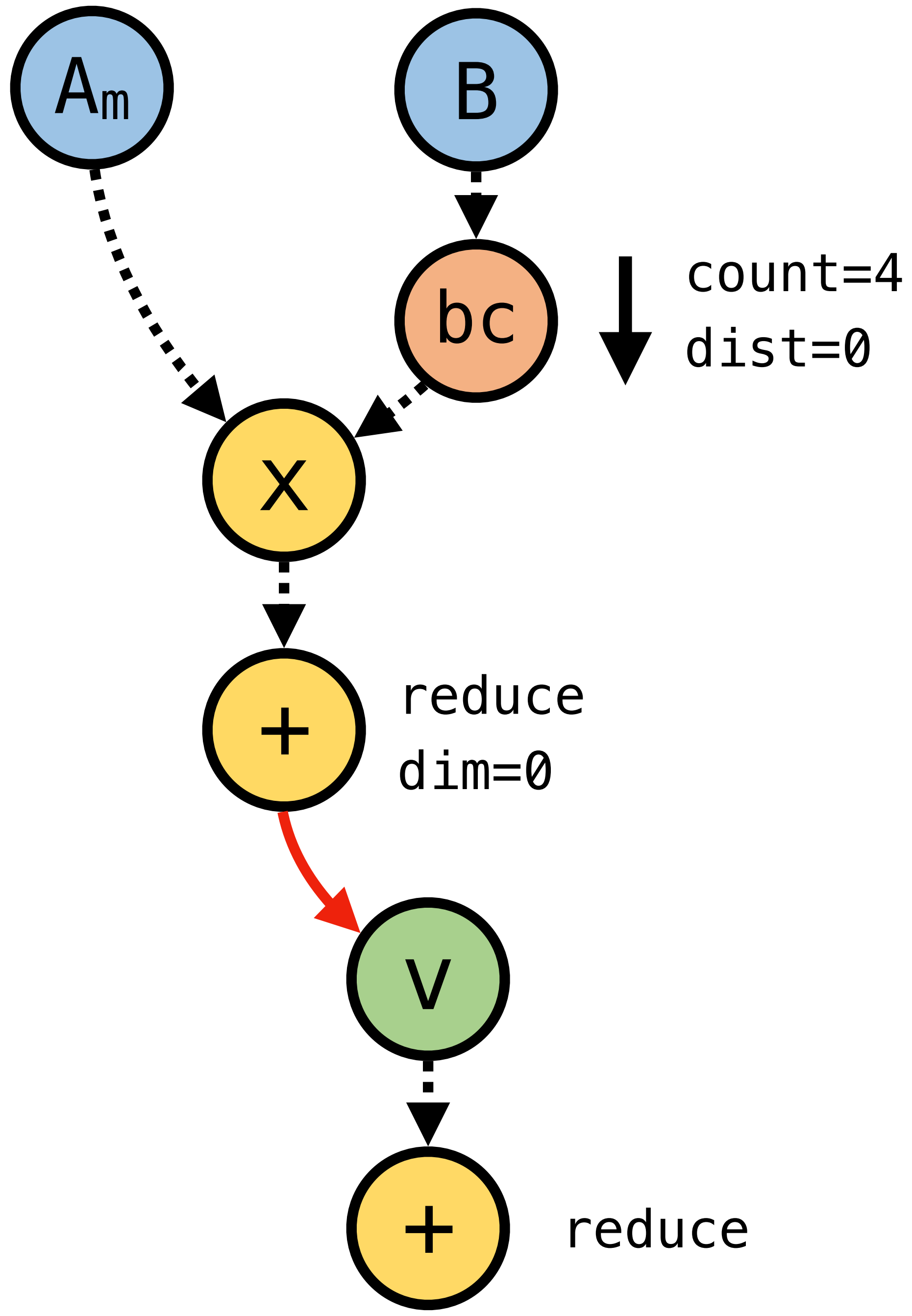
Spatial

```
for m in [0, M):
  C[m] = sum(A[m][:] * B[k])
```

Idea: Expose more parallelism through spatial reuse

- Spatial reuse





Broadcast Node bc captures spatial reuse

- Fusion

Matrix-Vector Multiplication

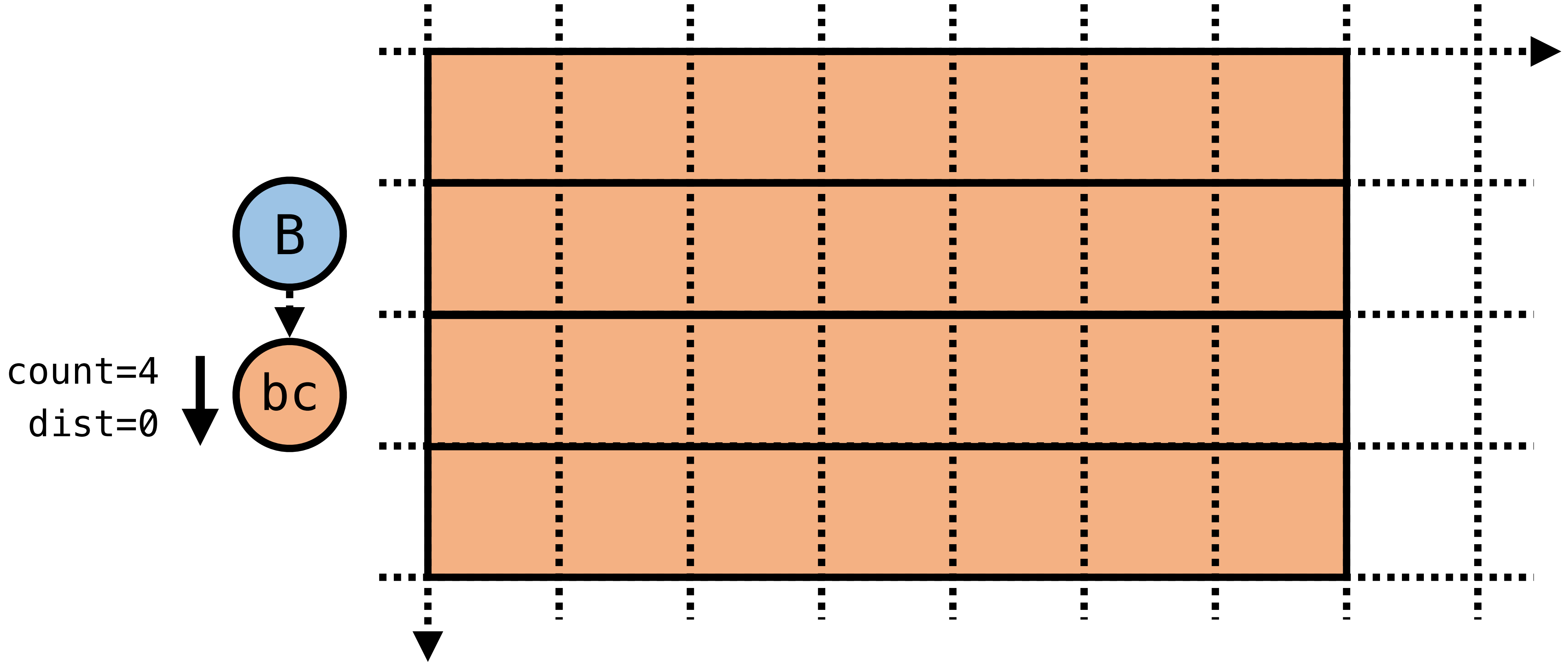
- Memory access pattern
- Computation
- N-D Tensors
- Data alignment
- Reduction operations

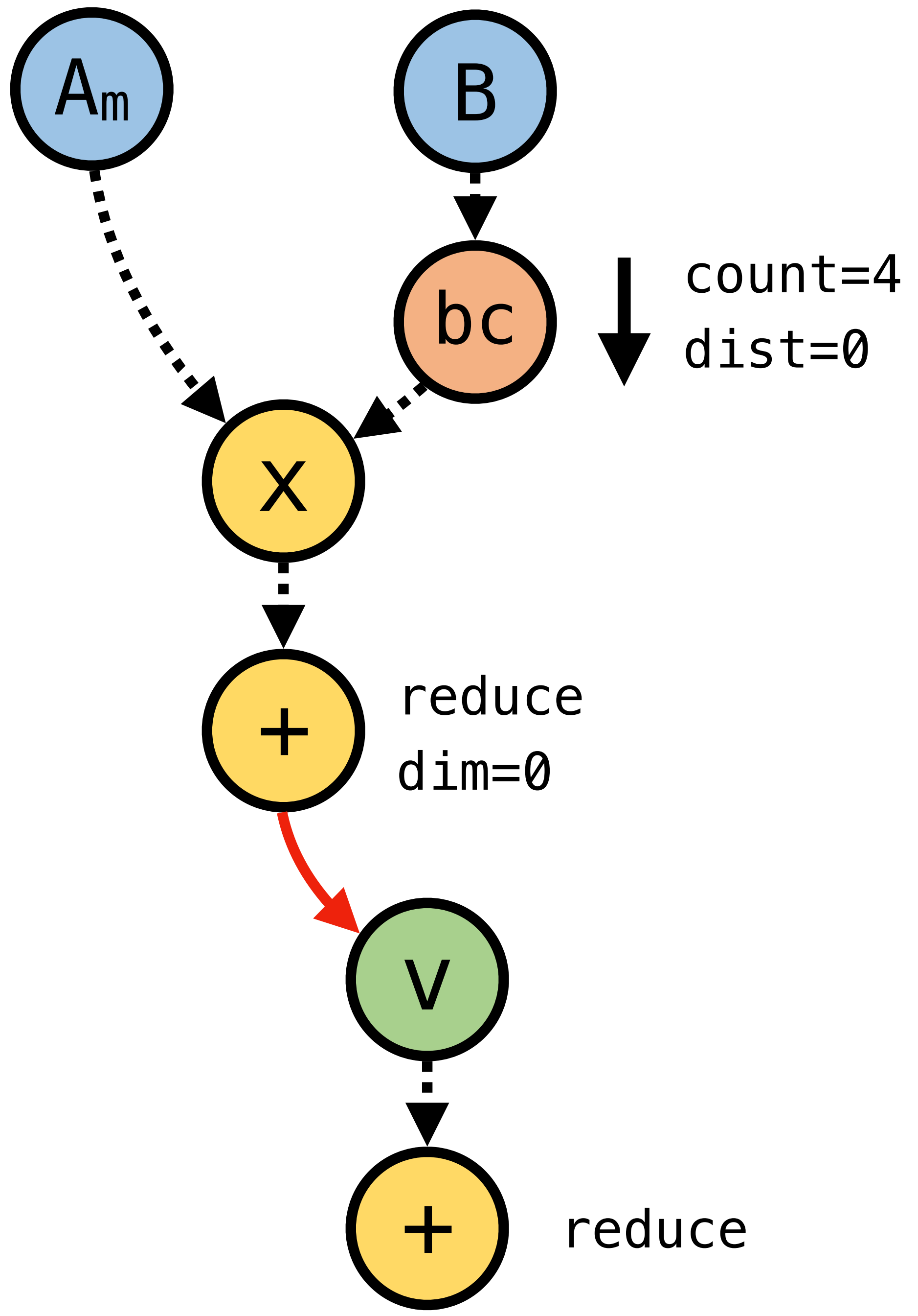
Spatial

```
for m in [0, M):
  C[m] = sum(A[m][:] * B[k])
```

Idea: Expose more parallelism through spatial reuse

- Spatial reuse





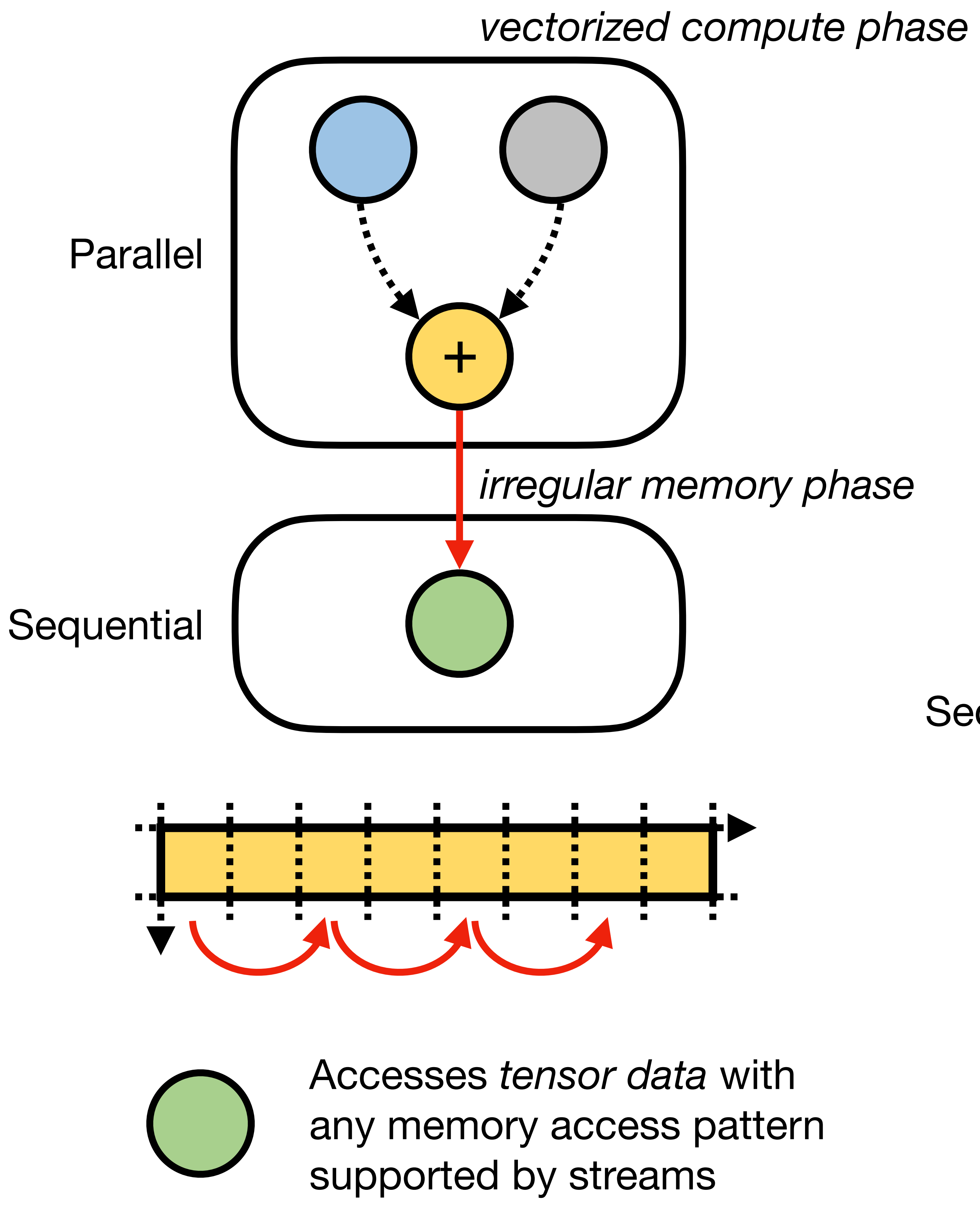
Broadcast Node bc captures spatial reuse

- Fusion

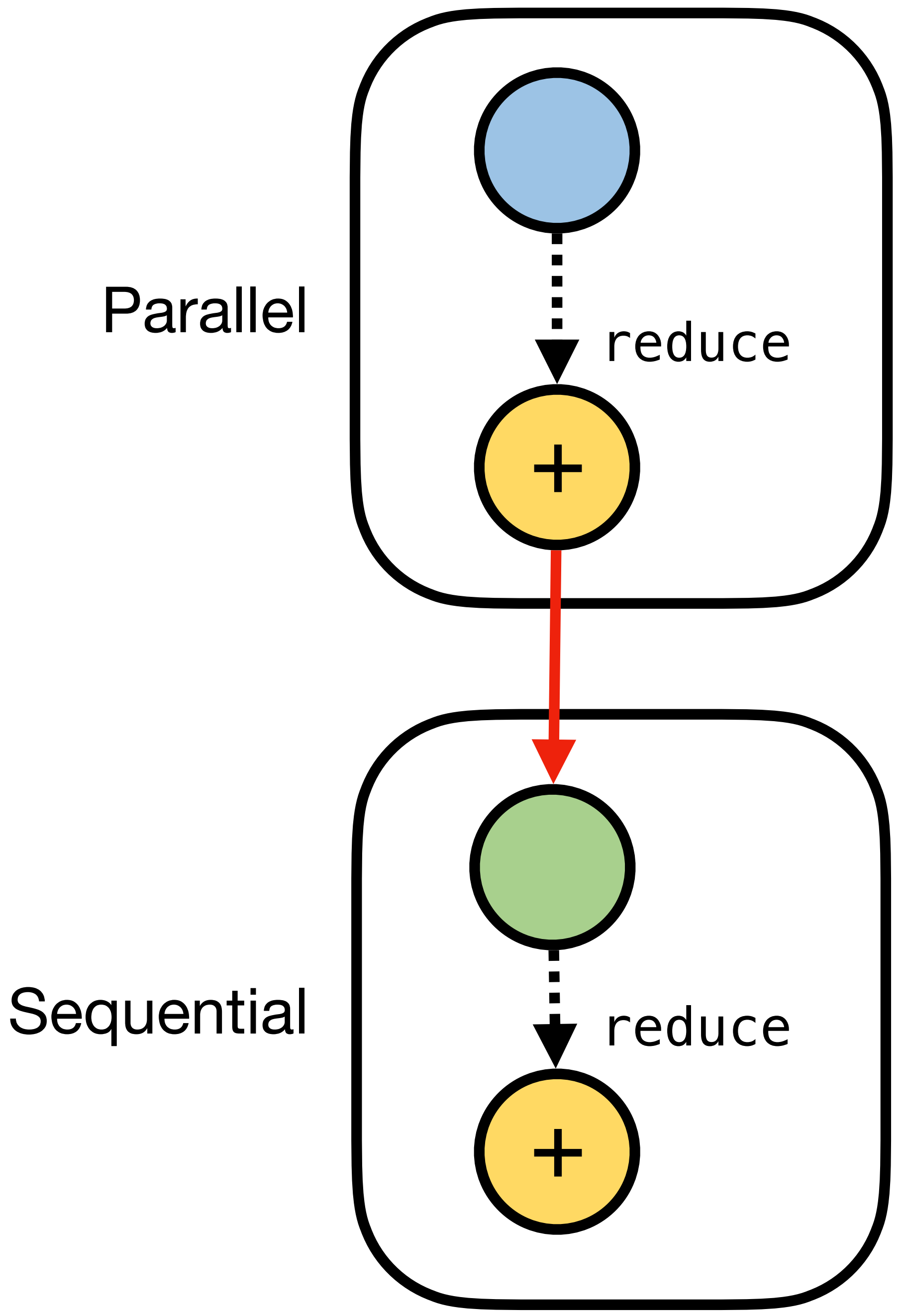
Stream ↔ Tensor

- Memory access pattern
- Computation
- N-D Tensors
- Data alignment
- Reduction operations
- Spatial reuse
- Fusion

Load as Stream

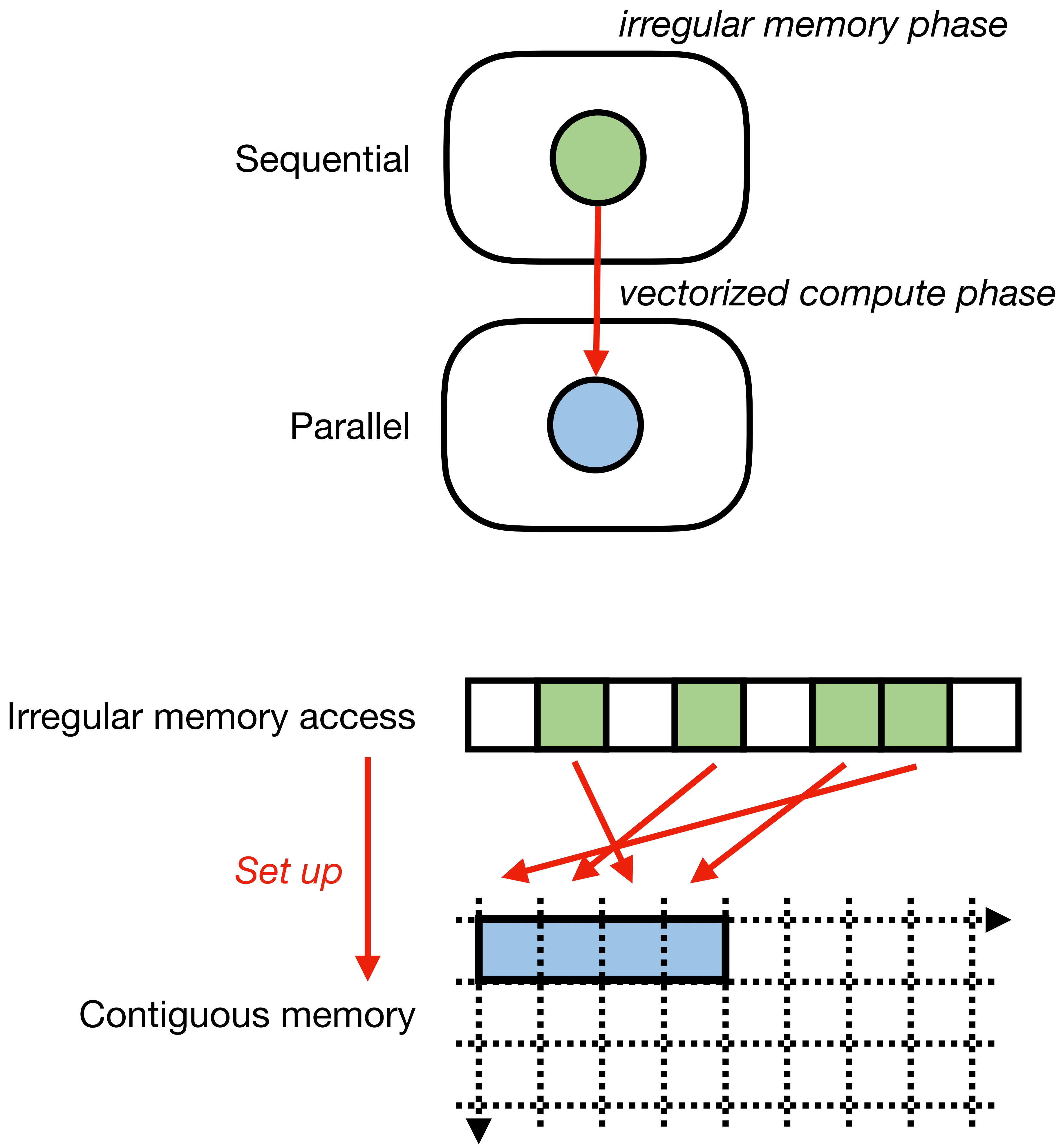


Reduction Stream



Special case of
Load as Stream

Store as Tensor



Outline

Plain C Code

2
Unified Abstraction

Agnostically represents
1. In-core compute
2. In-memory compute
3. Near-memory compute

4
Partitioning Static vs Dynamic

Portable binary without exposing µArch details to compiler

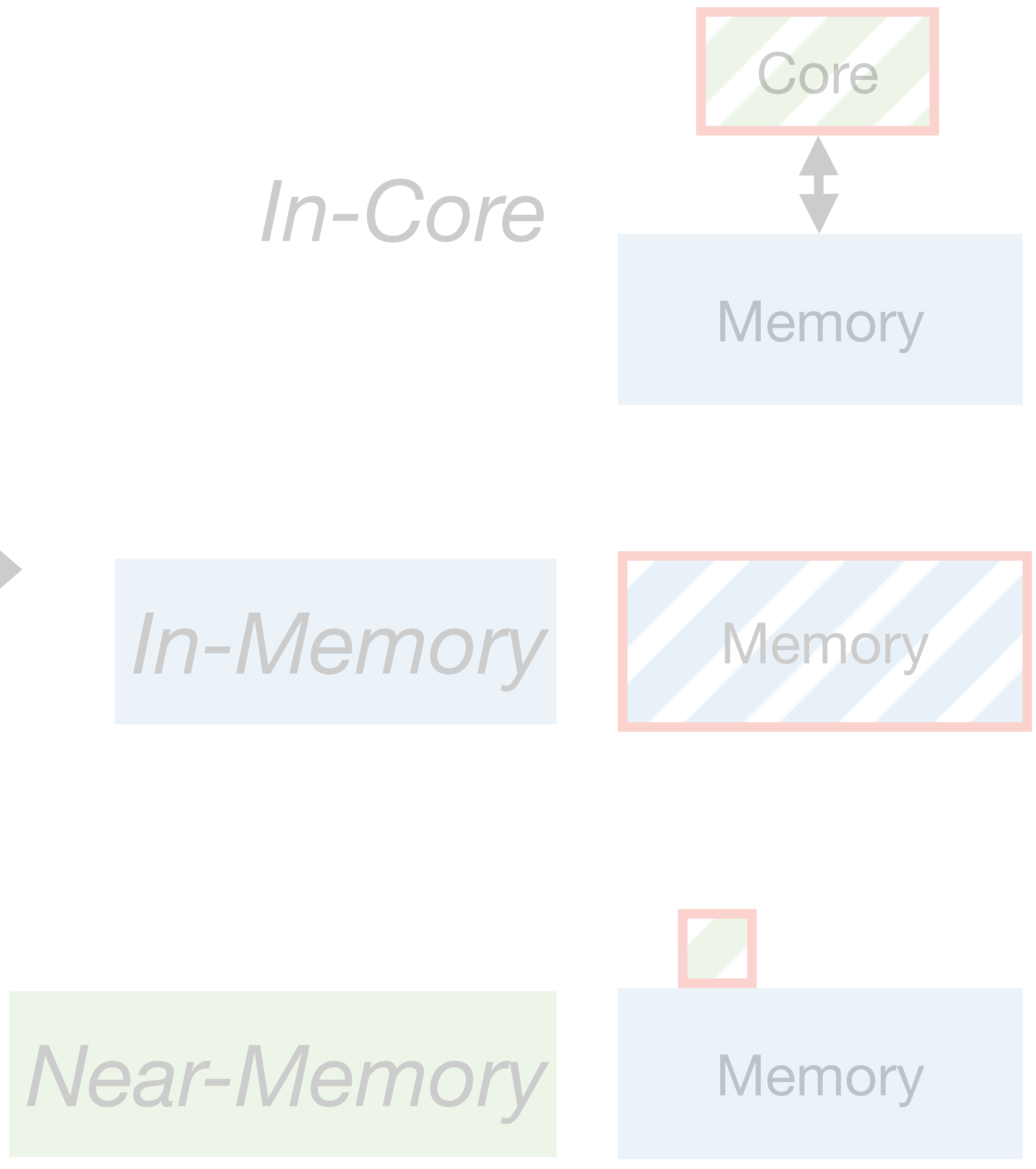
3
µArch-specific Optimizations

Specialize binary to take advantage of µArch details

µArch Extensions

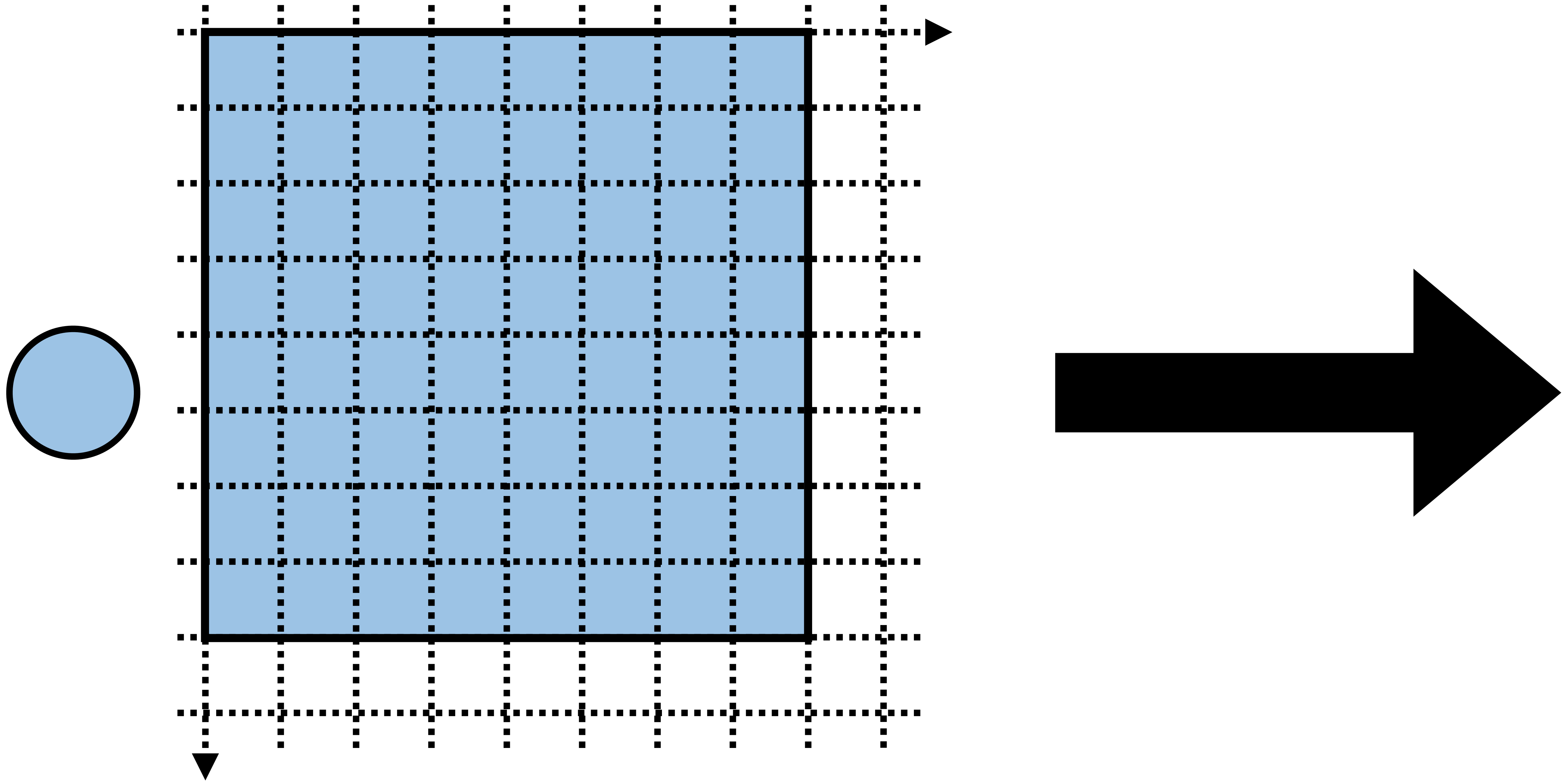
Hardware support for compute paradigms

1
Compute Paradigms

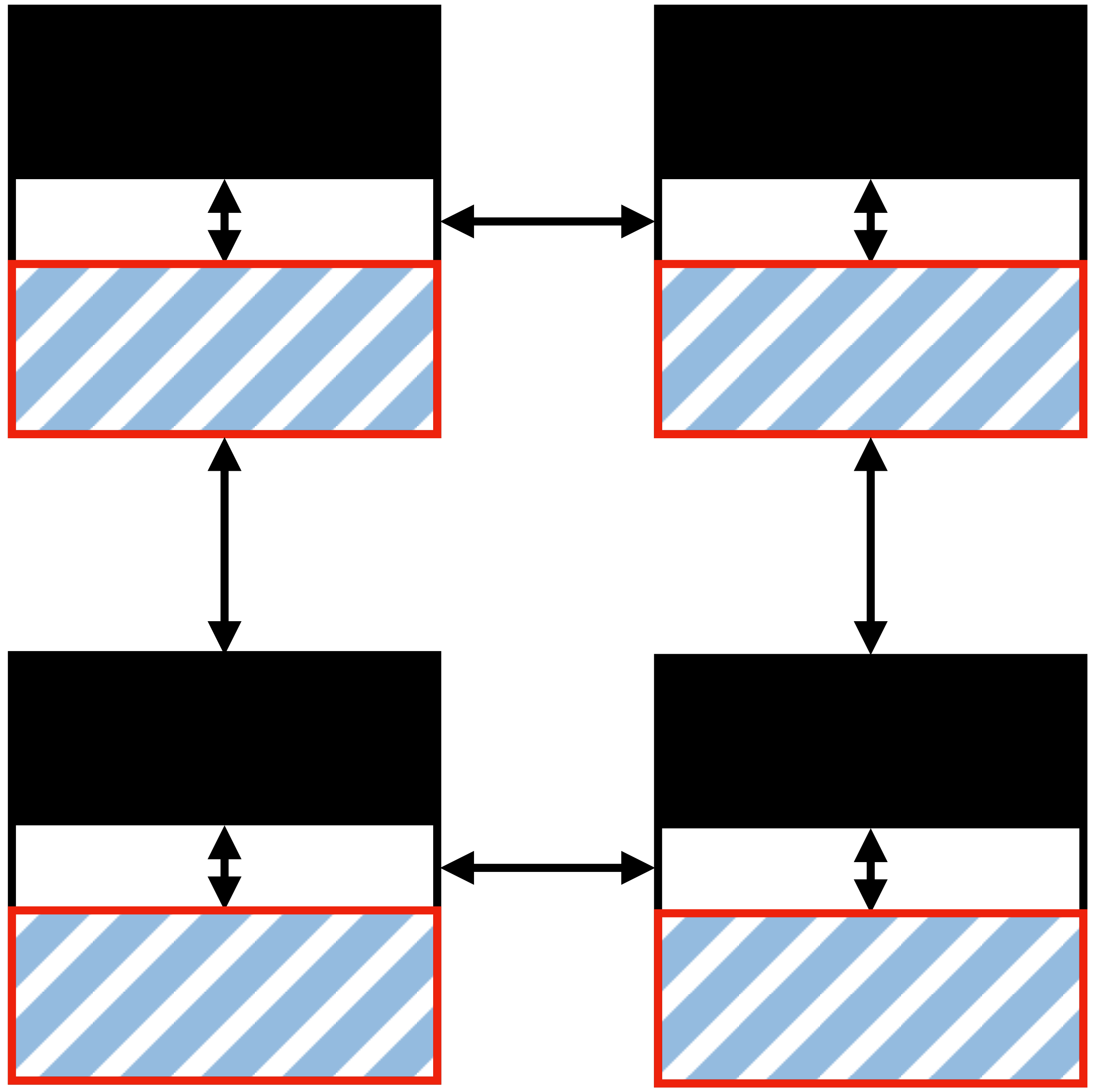


Mapping to Hardware

Global Lattice

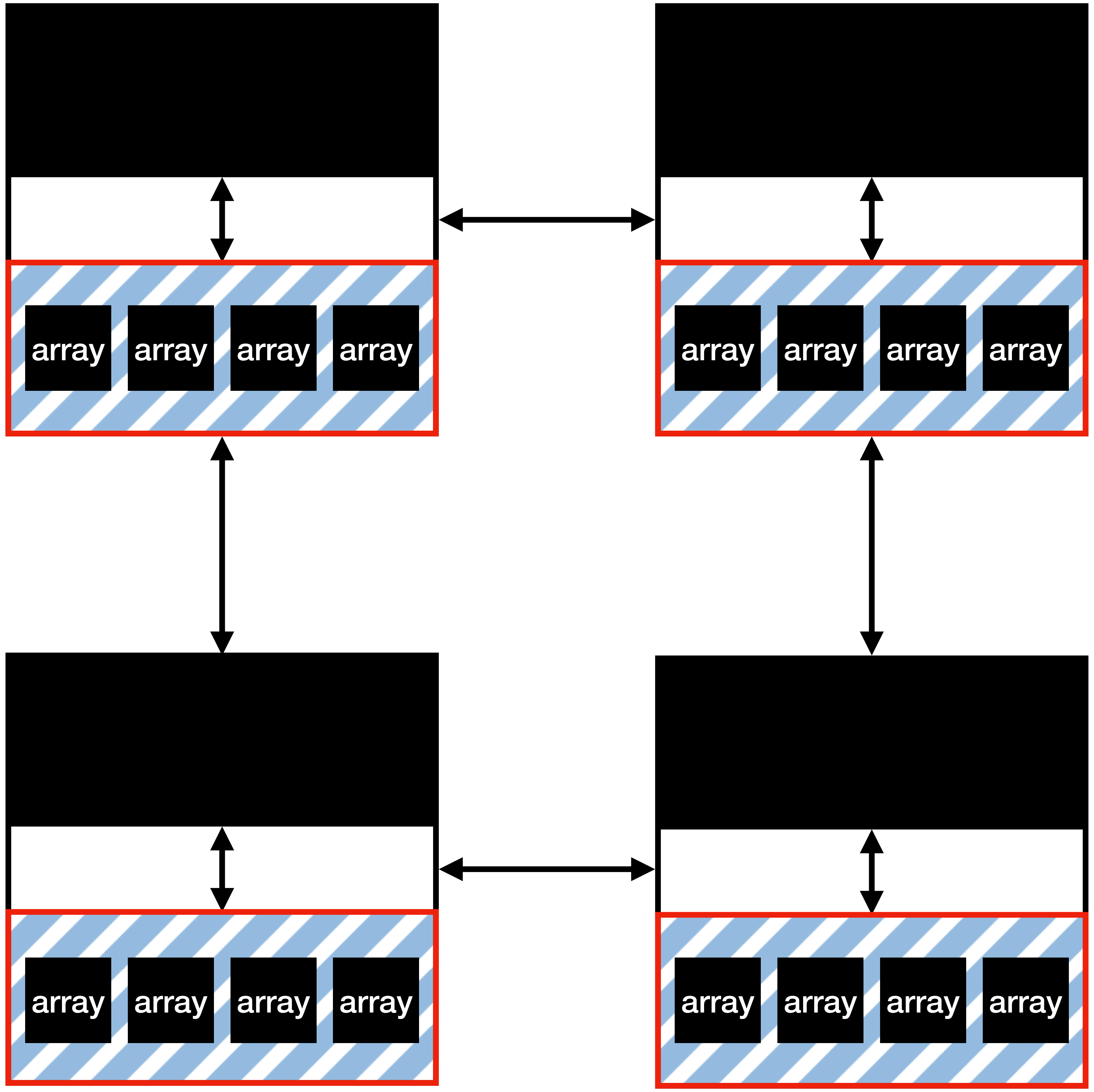


SRAM Banks

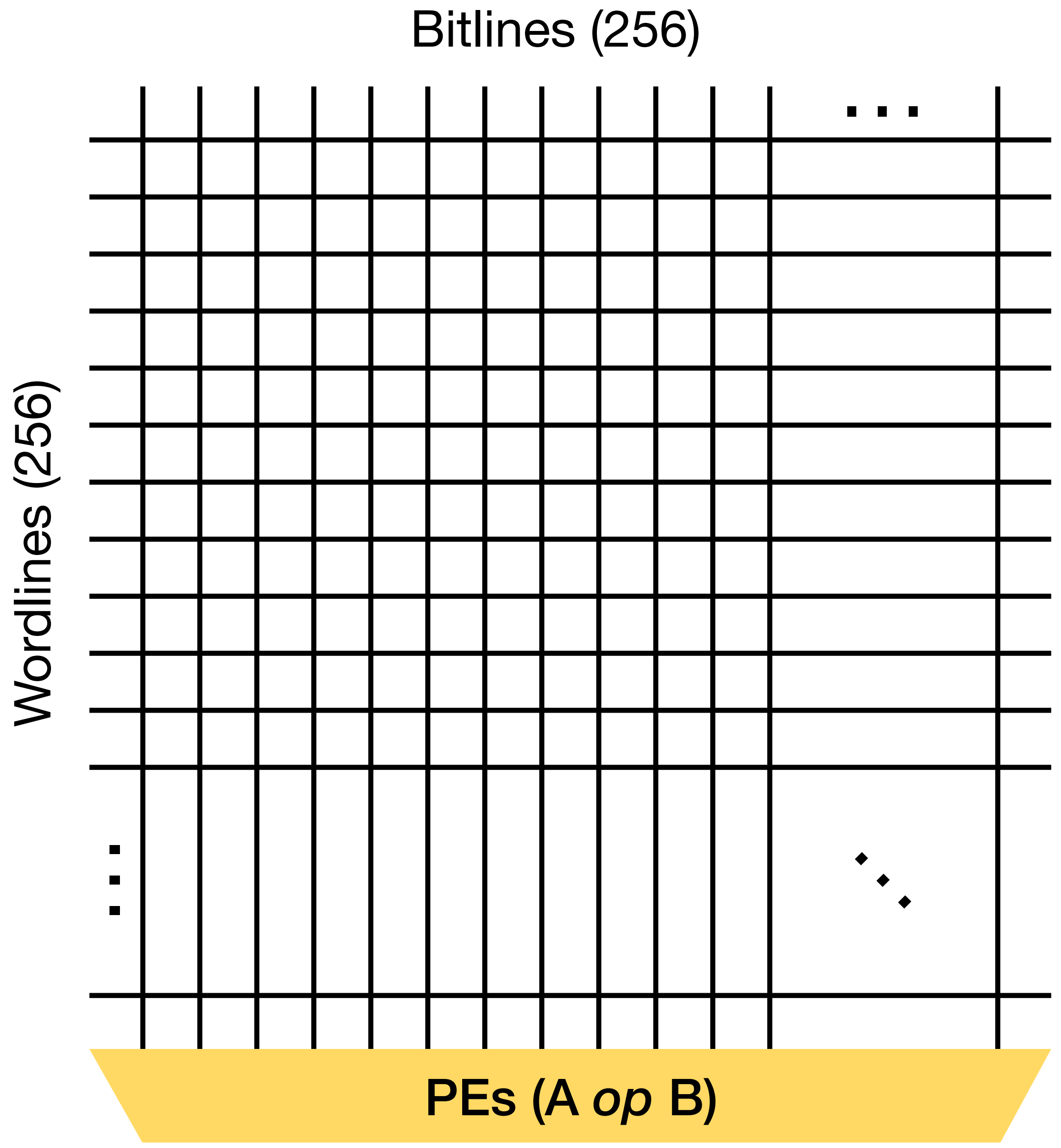


In-Memory Organization

Question
What are the effects of tiling on
on inter-bank traffic?

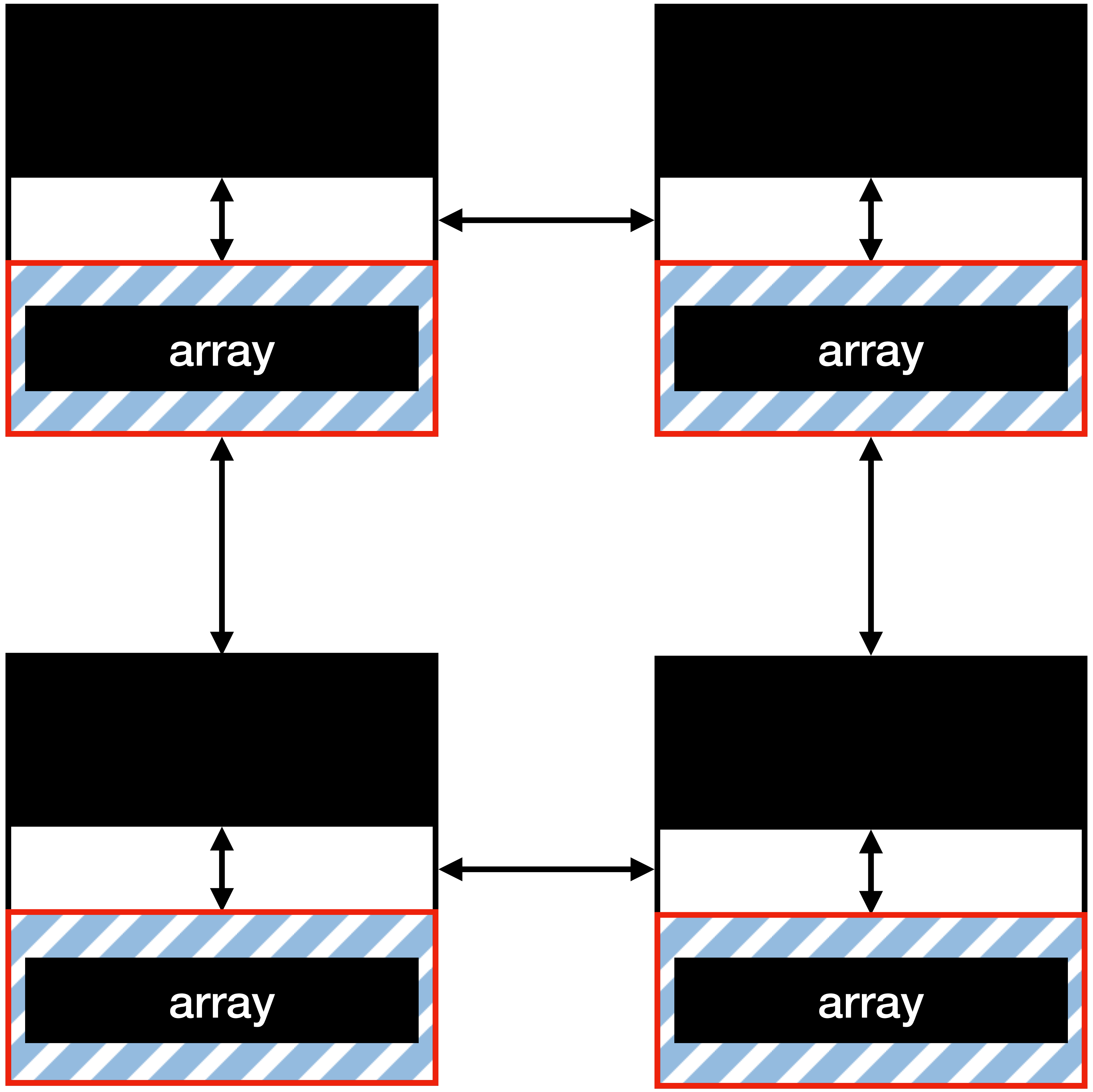


Each last-level cache bank consists of many in-memory SRAM arrays

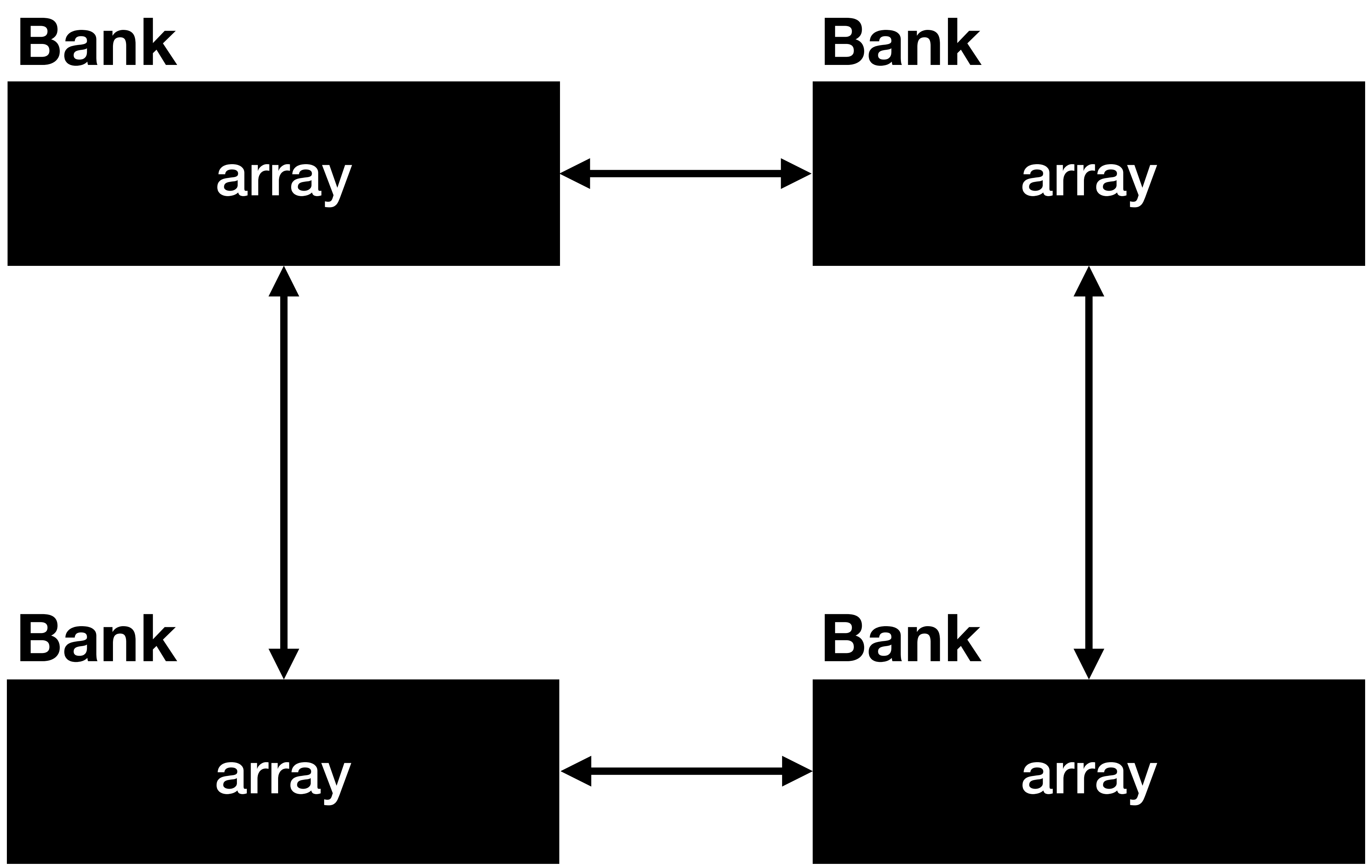


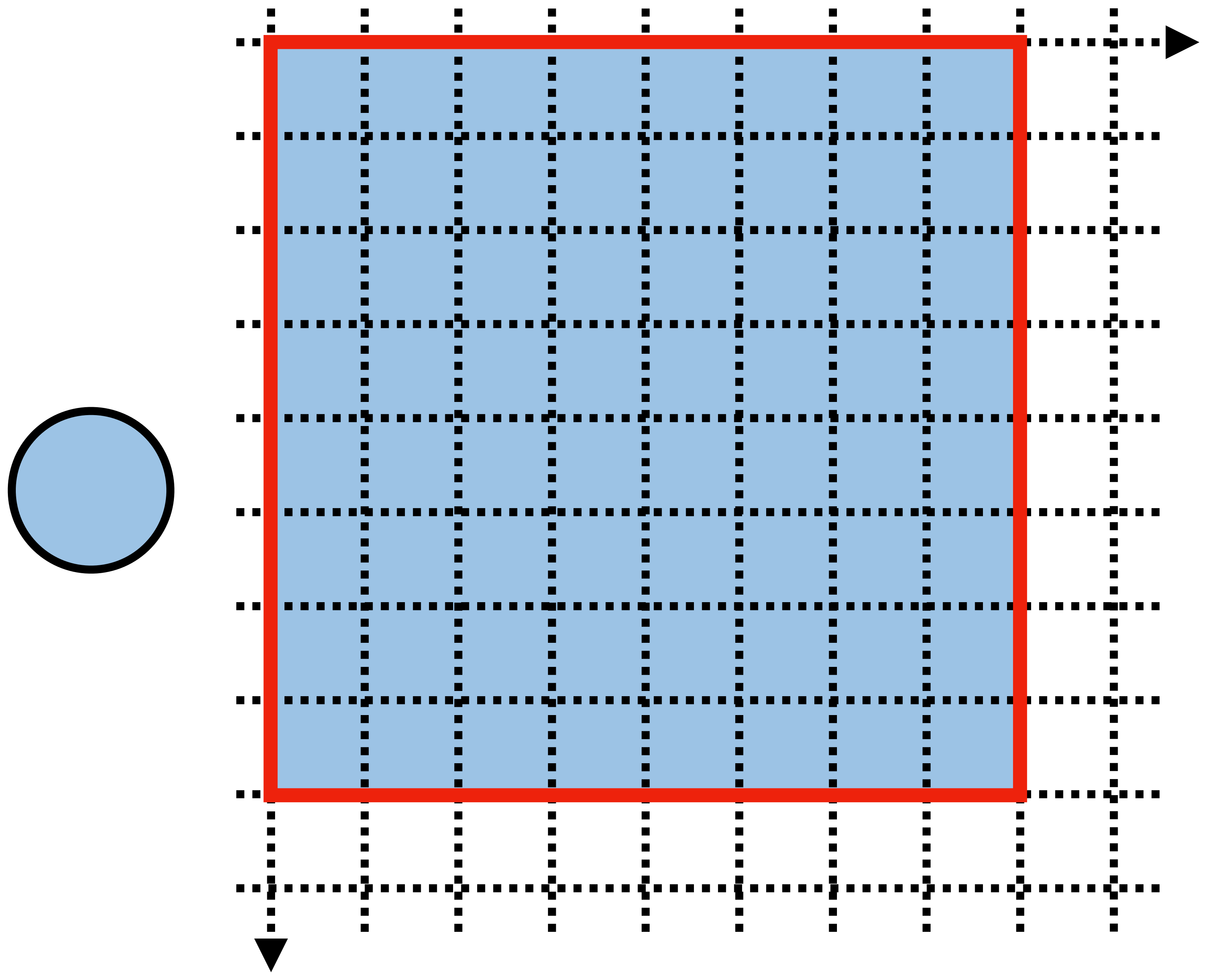
In-Memory Organization

Question
What are the effects of tiling on
on inter-bank traffic?

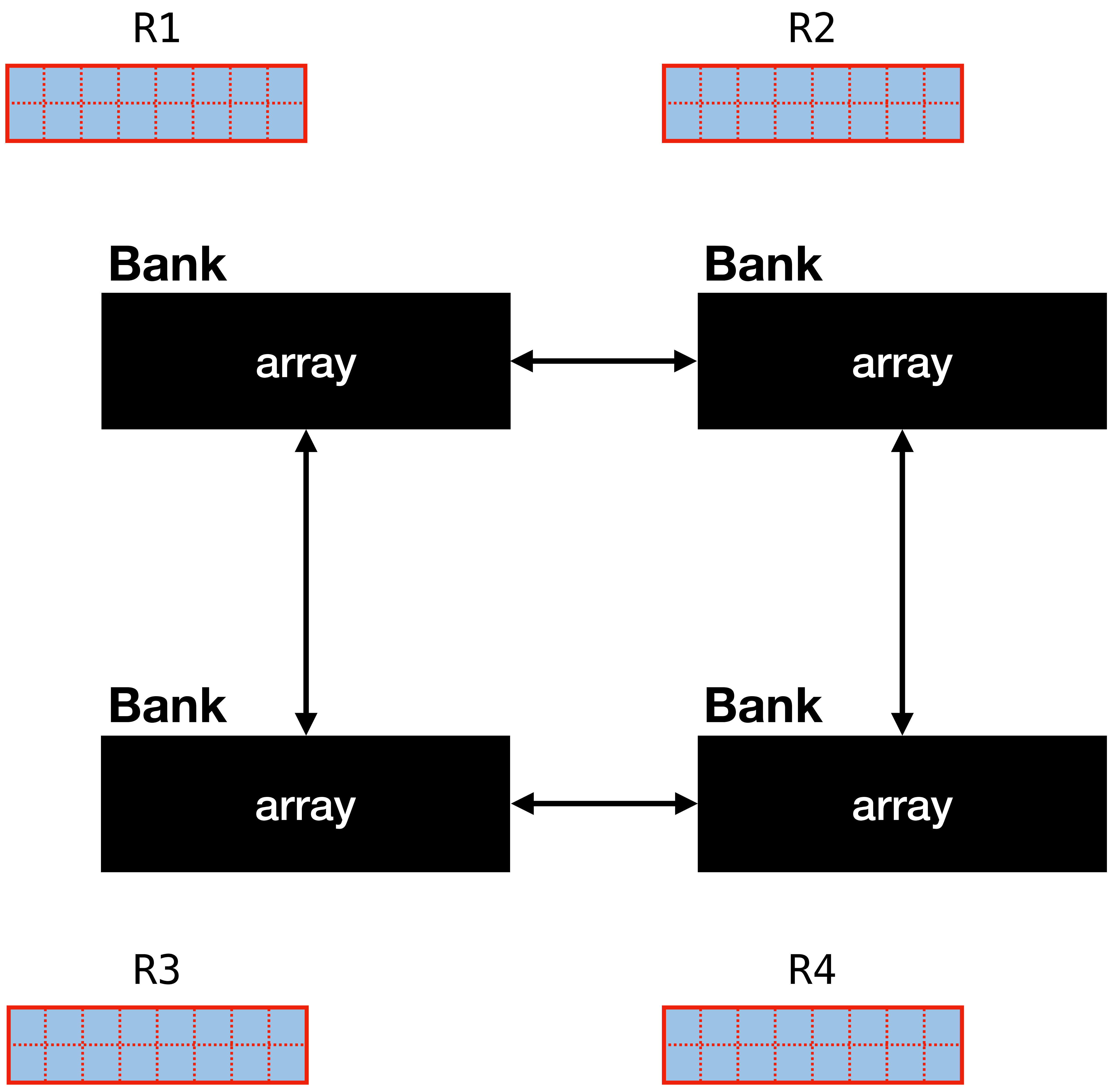


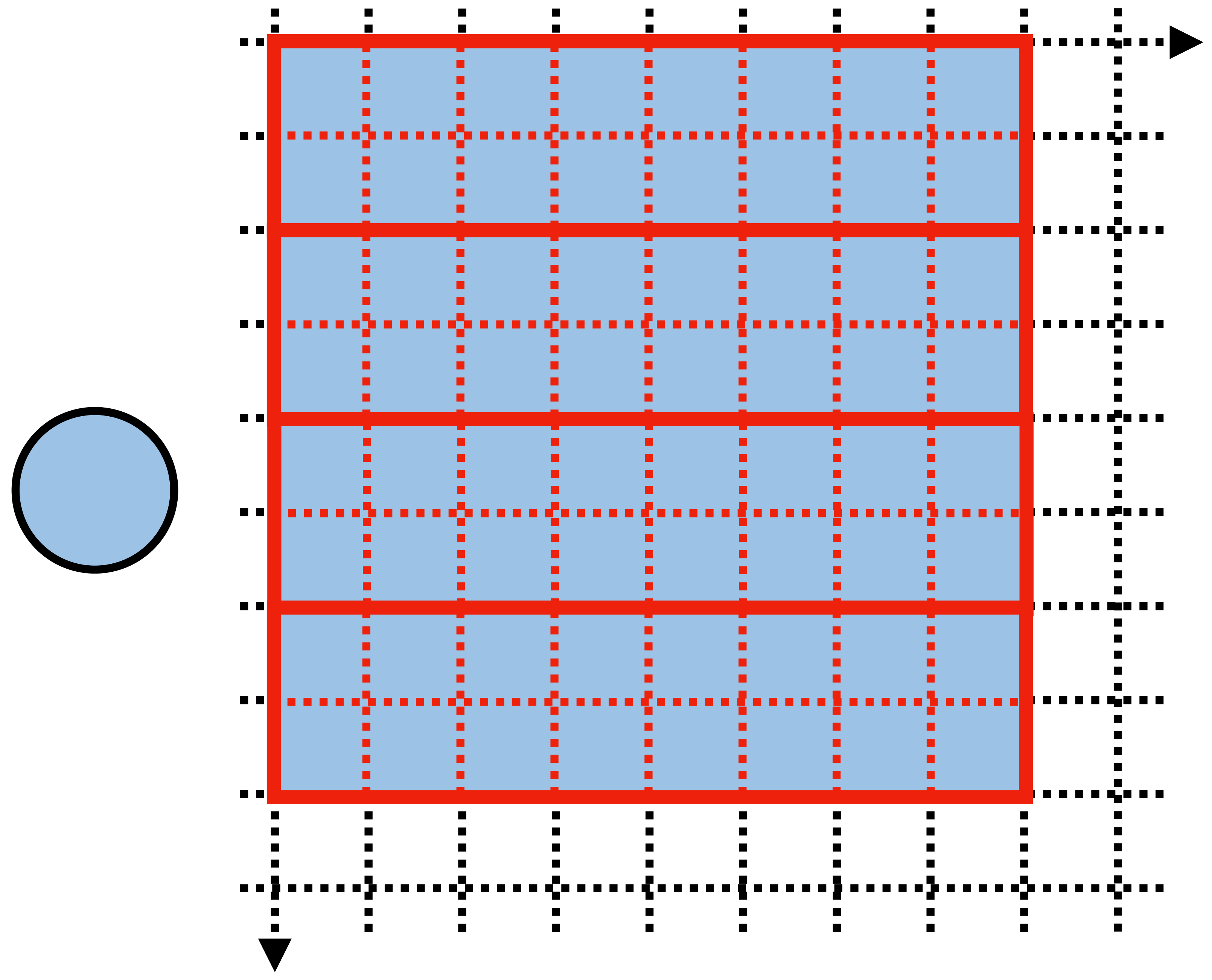
Abstract many SRAM arrays
into a single large SRAM array

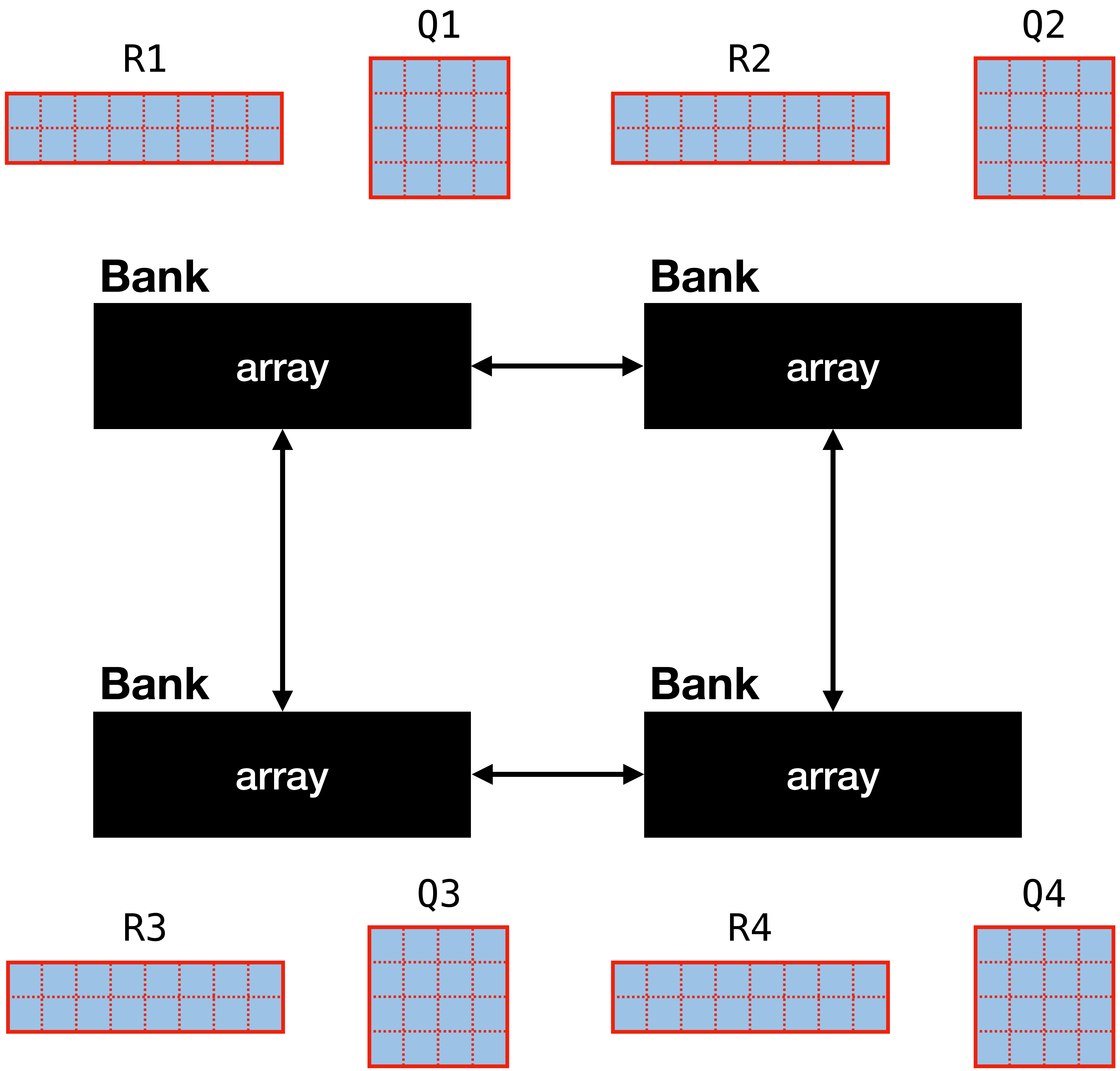


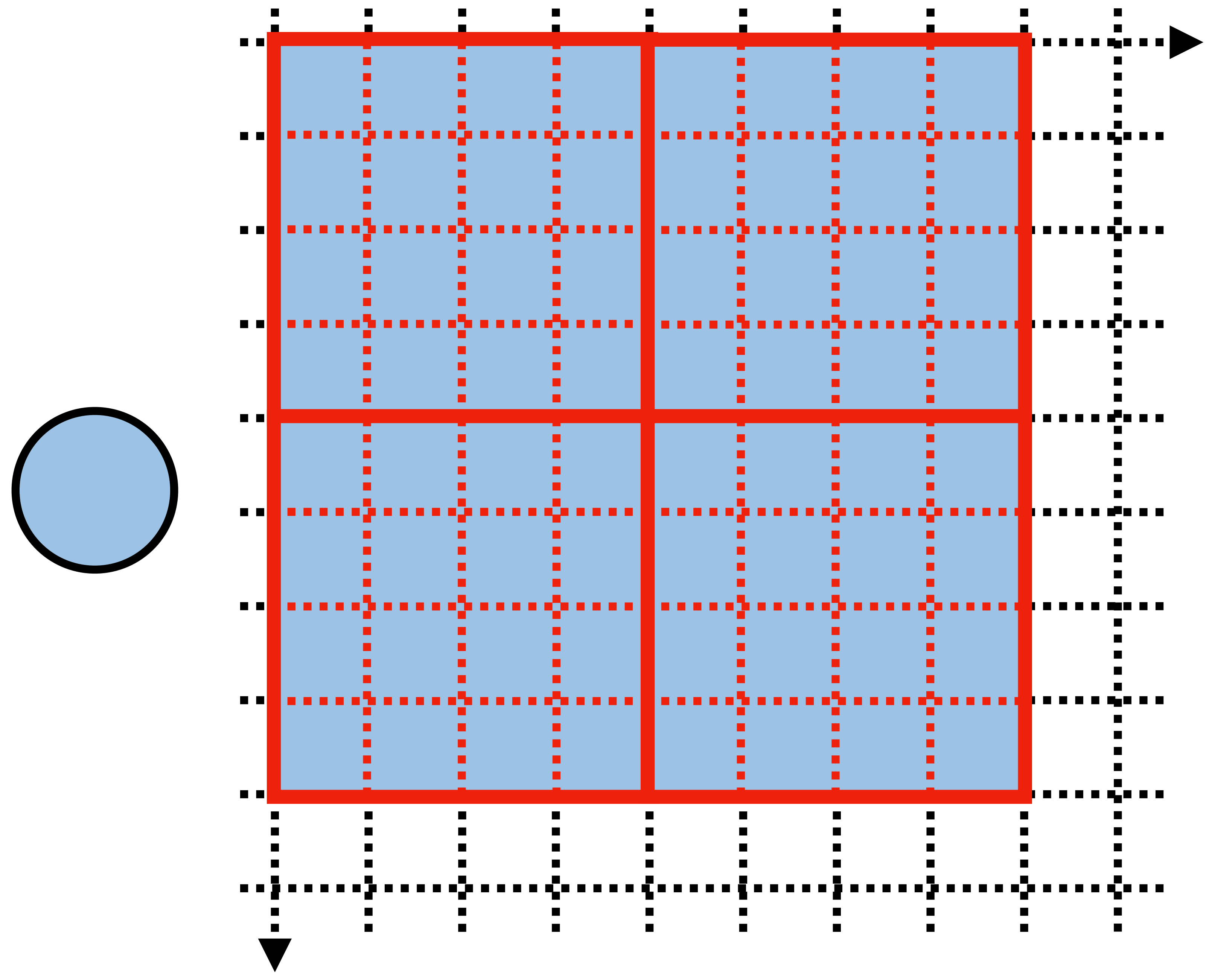


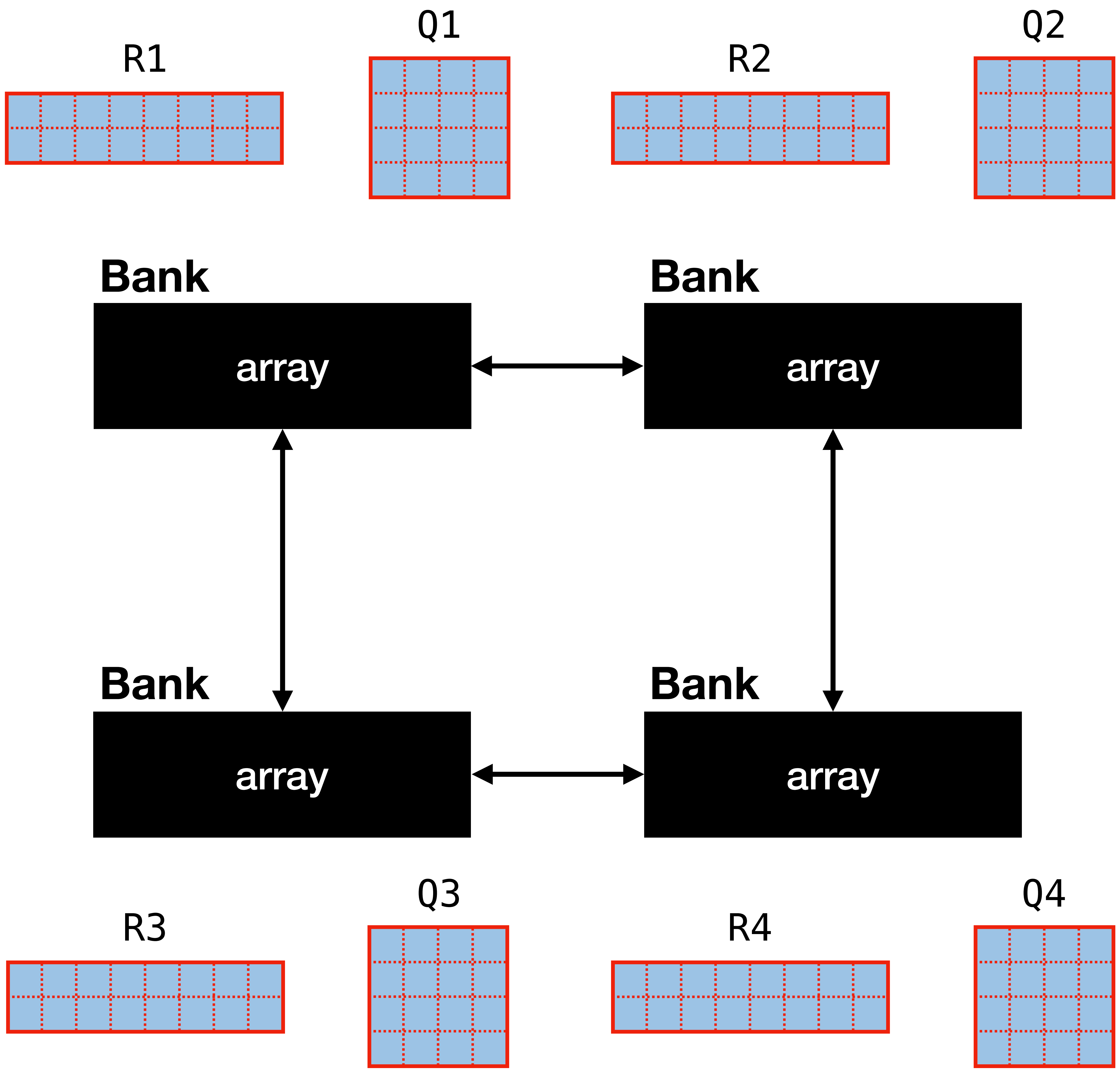
Lattice cells are a sub-matrix of some larger matrix


R1
R2
Bank
array
Bank
array
Bank
array
Bank
array
R3
R4

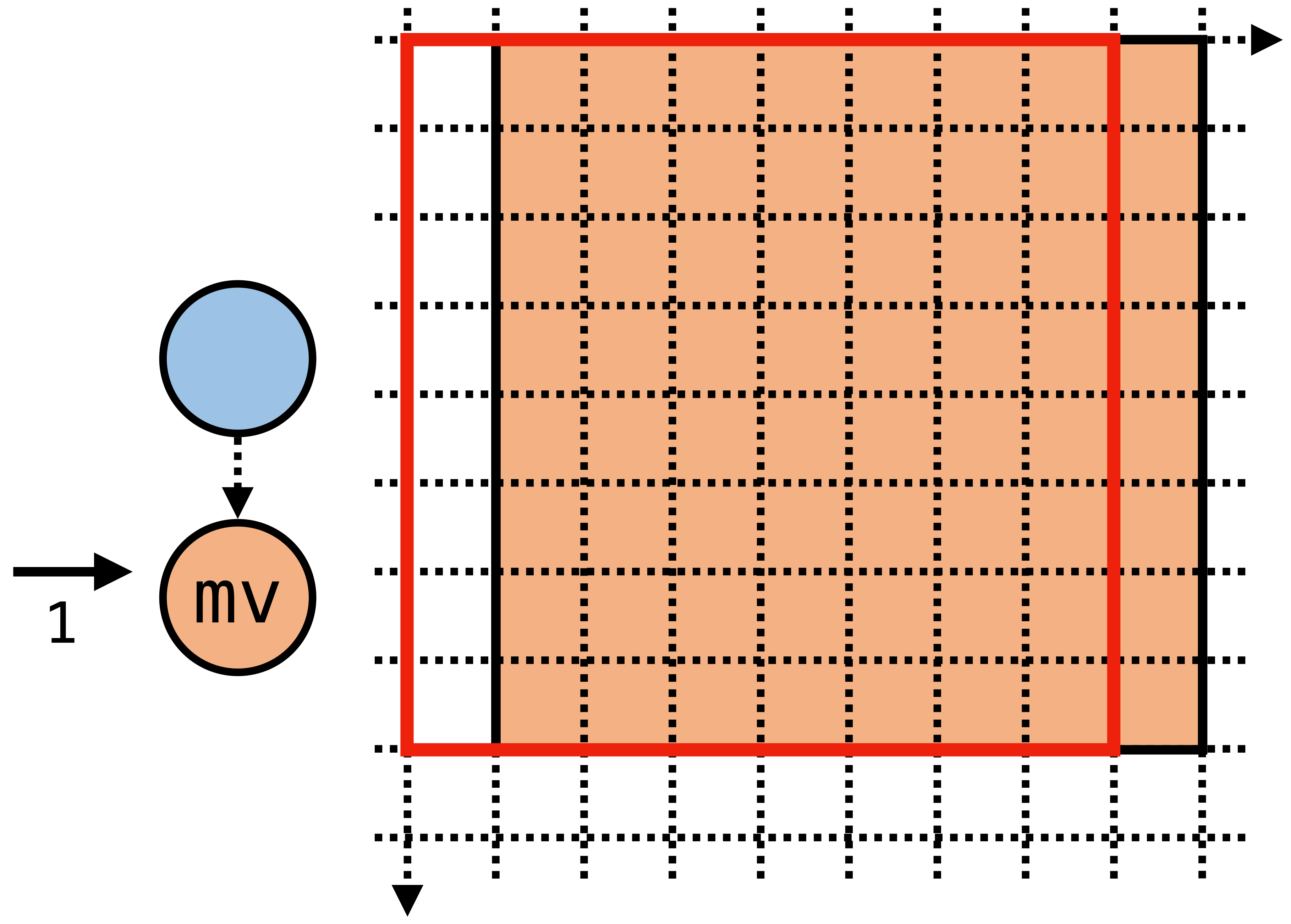



R1
Q1
R2
Q2
Bank
array
Bank
array
Bank
array
Bank
array
R3
Q3
R4
Q4

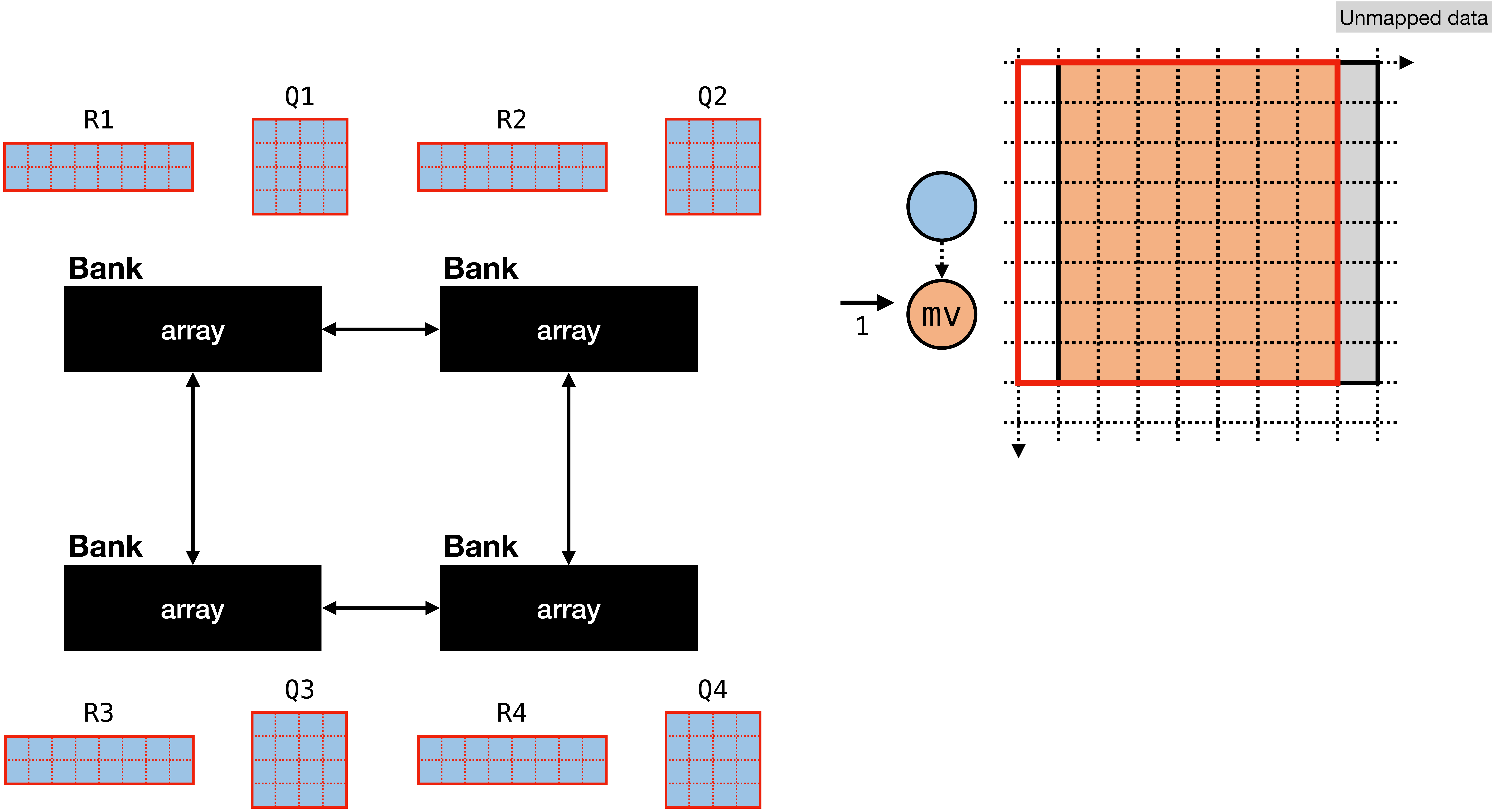


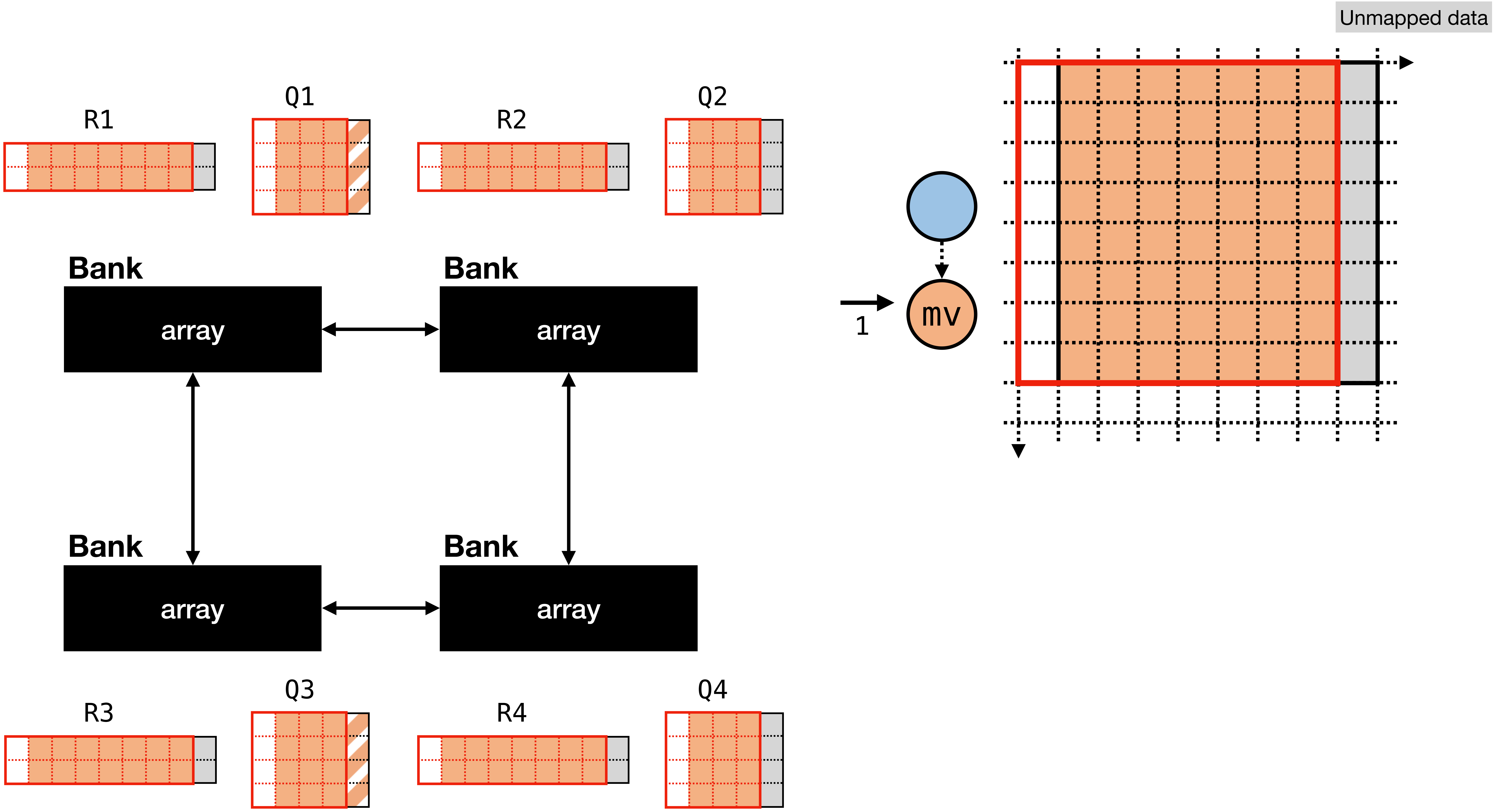


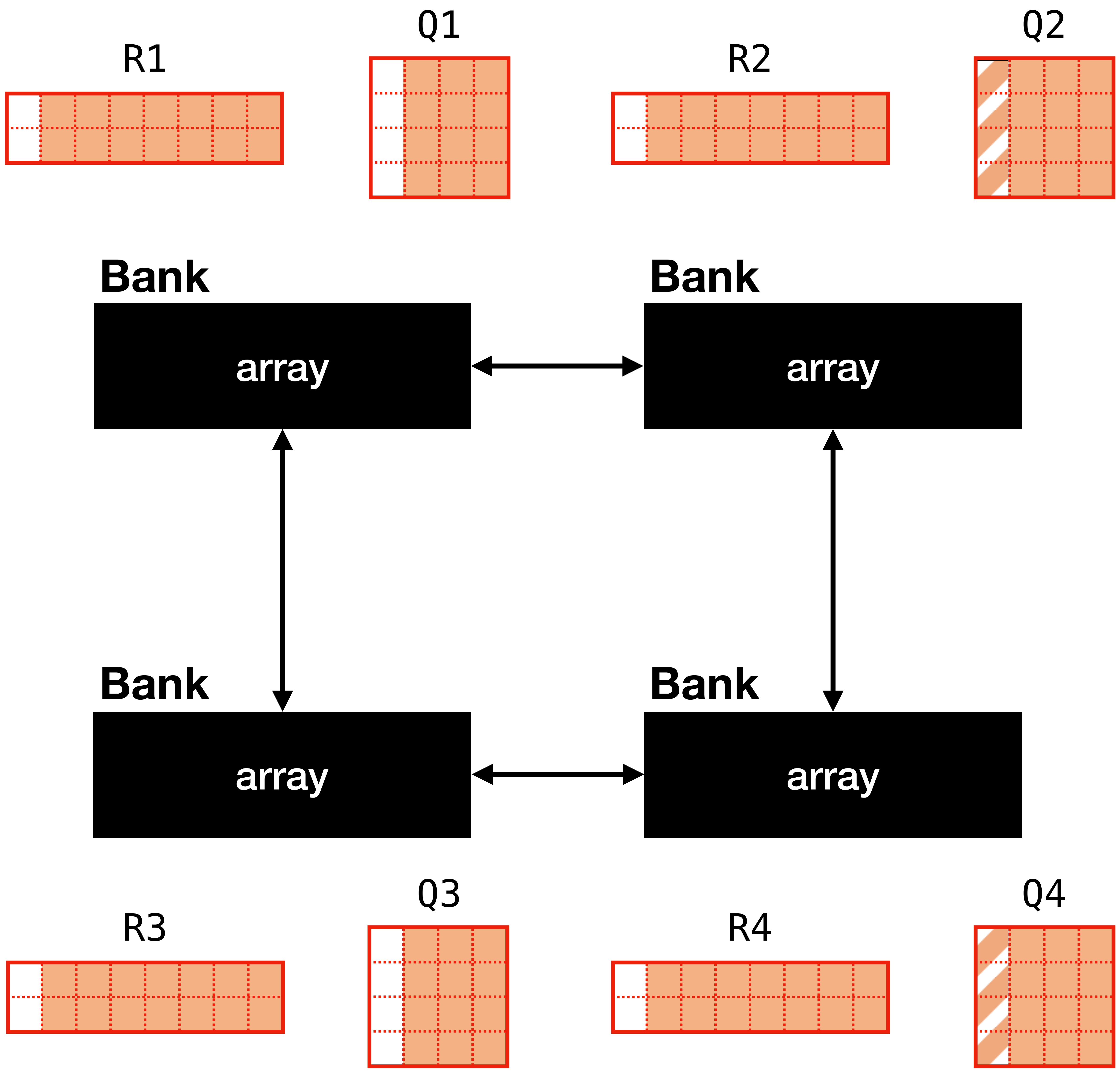
R1
Q1
R2
Q2
Bank
array
Bank
array
Bank
array
Bank
array
R3
Q3
R4
Q4

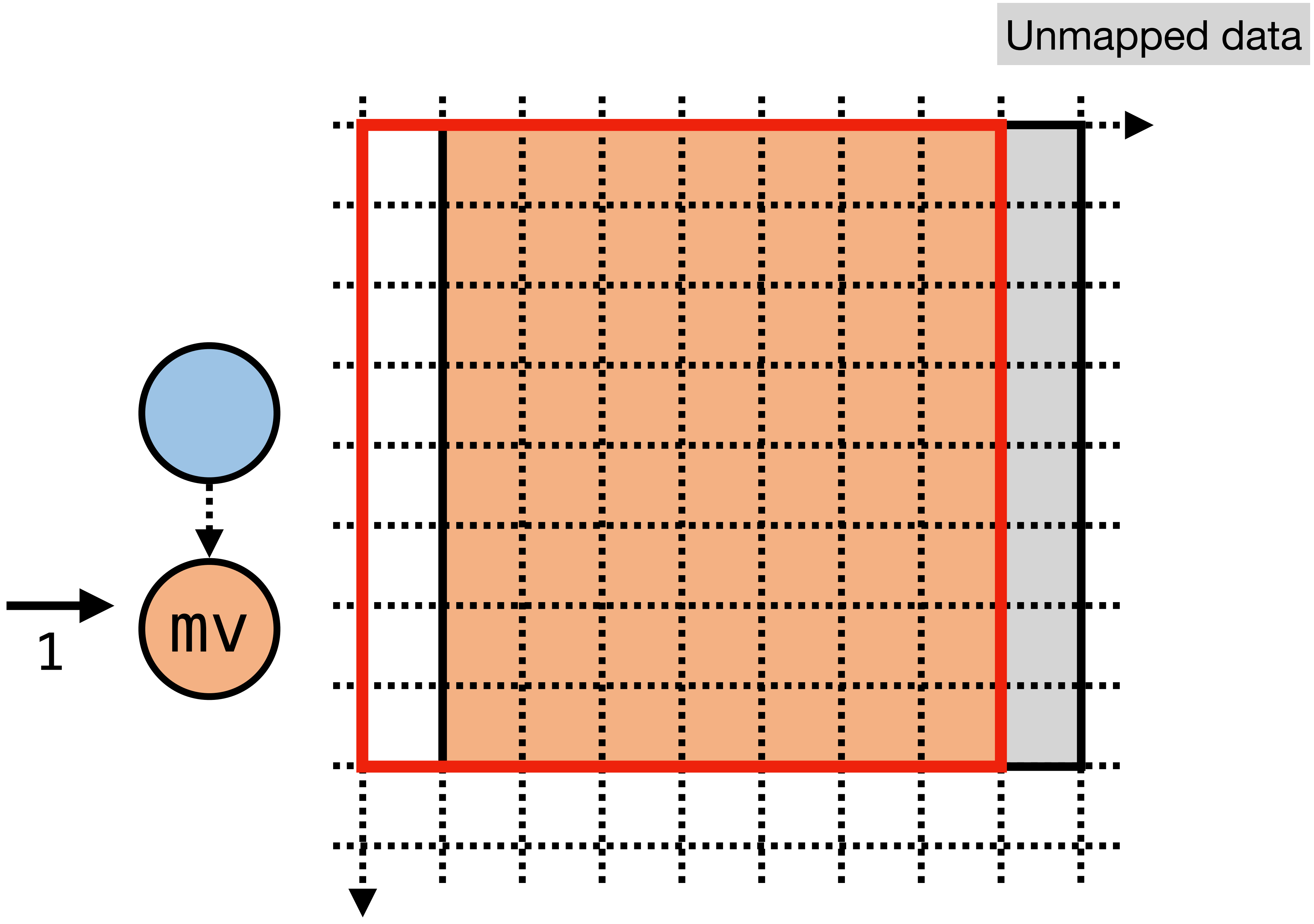


1
mv

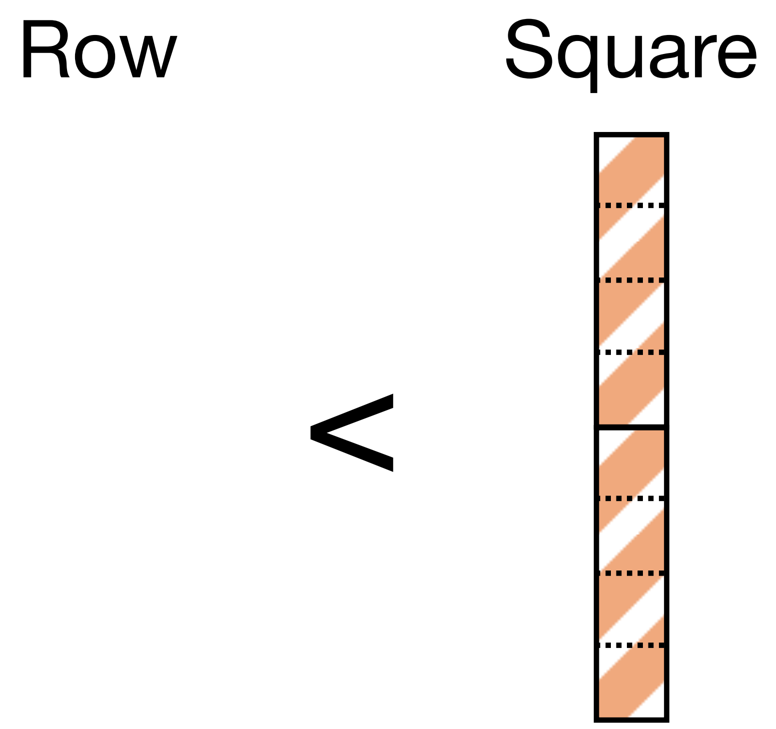

Unmapped data
R1
Q1
R2
Q2
Bank
array
Bank
array
Bank
array
Bank
array
R3
Q3
R4
Q4
1
mv

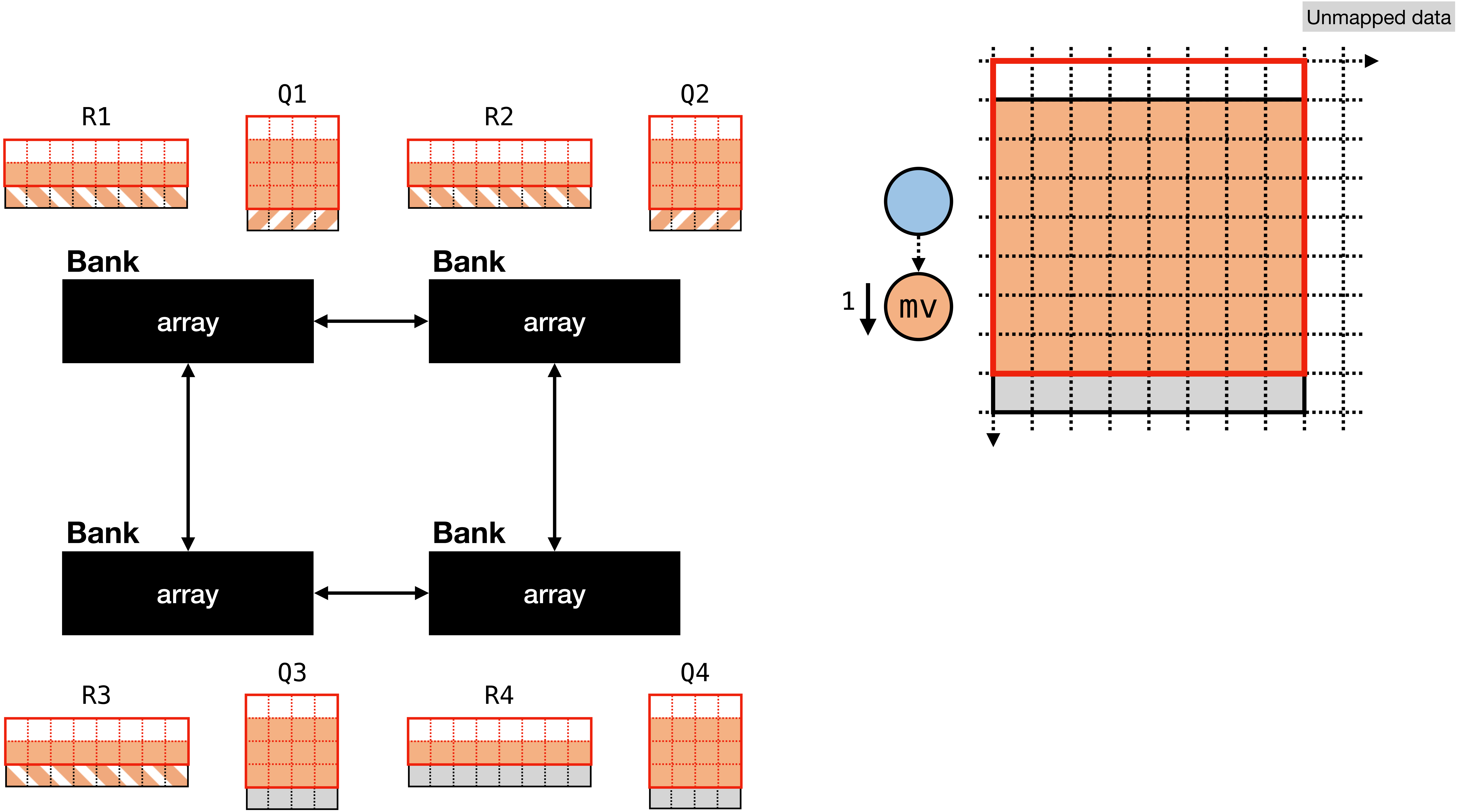

Unmapped data
R1
Q1
R2
Q2
Bank
array
Bank
array
Bank
array
Bank
array
R3
Q3
R4
Q4
1
mv

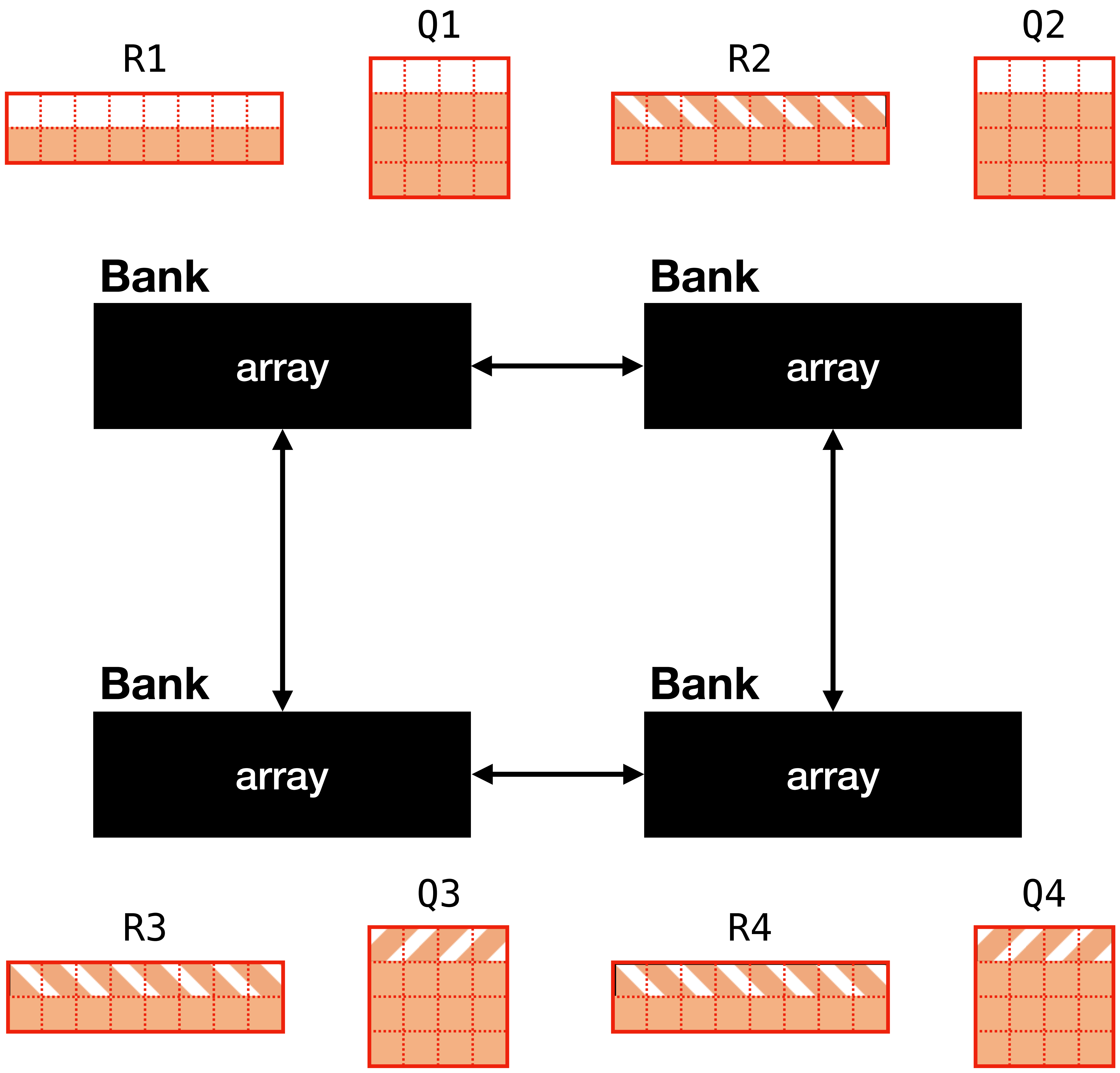


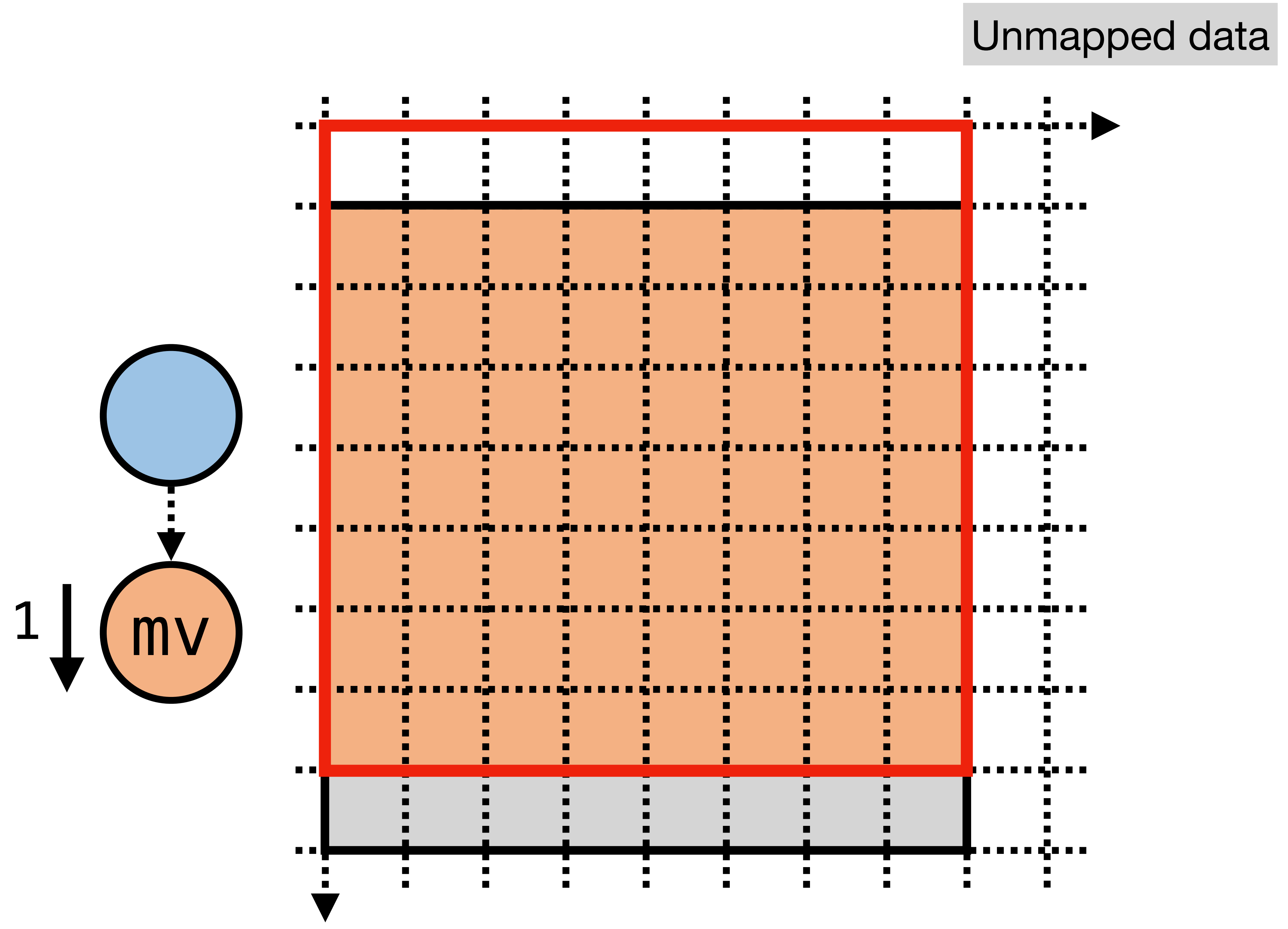


Inter-Bank Data Movement

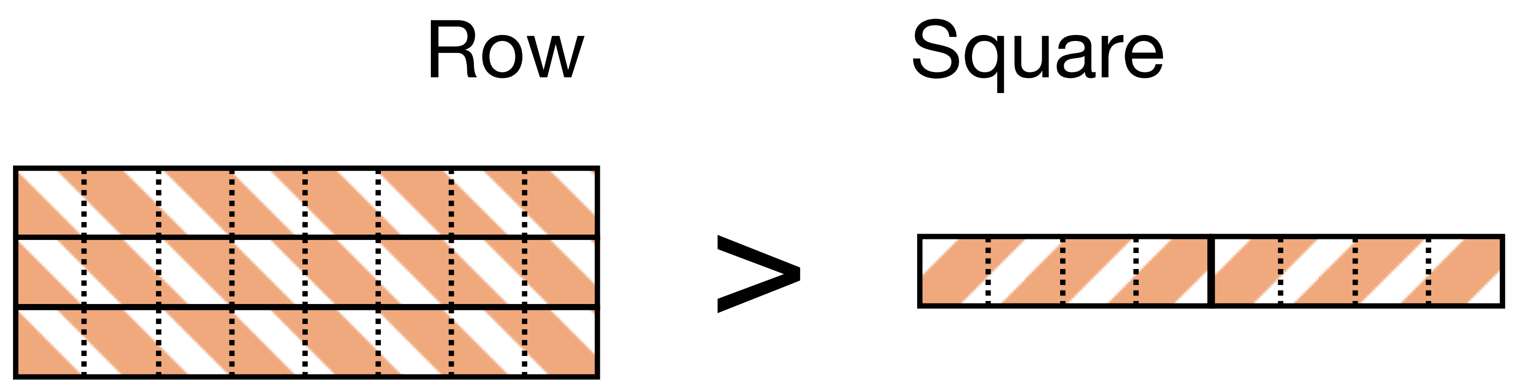



Unmapped data
R1
Q1
R2
Q2
Bank
array
Bank
array
Bank
array
Bank
array
R3
Q3
R4
Q4
1
mv





Inter-Bank Data Movement



Inter-bank Data Movement

depends on

Tiling Size + Move Direction

Inter-bank Data Movement
depends on
Tiling Size + Move Direction

Static

Compiler

Dynamic

Runtime &
JIT Compiler

Layout Hints

1. Inter-Bank Traffic:

Best tiling size that minimizes inter-bank data movement

Tensor T is moved along dimensions d_1, d_2, d_3, …

2. Data Alignment:

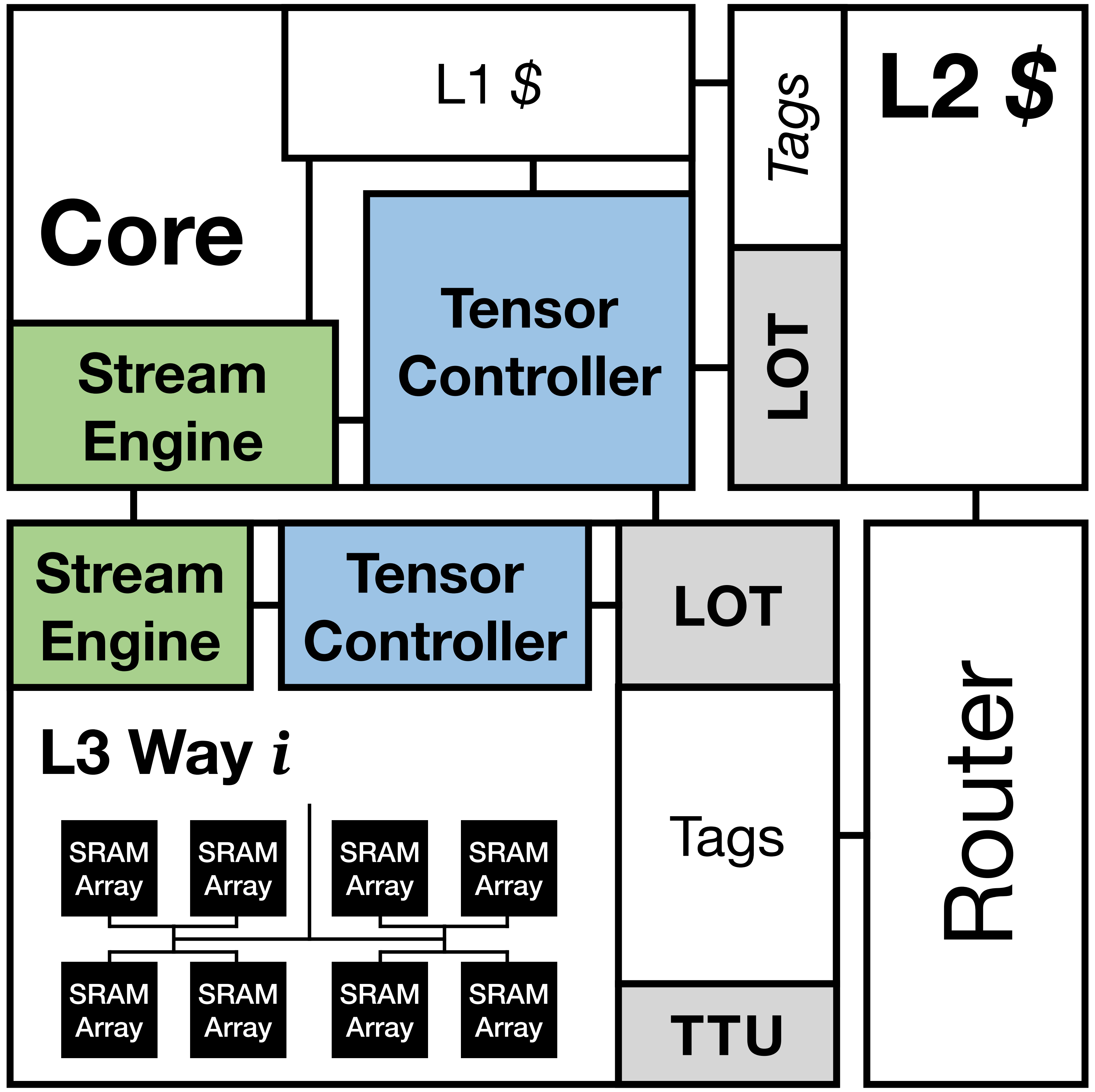
Tensors A & B must have matching tiling size

Tensors A & B are used together

Microarchitecture Extensions

Tensor Transpose Unit

Transposes array elements to and from bit-serial format



Layout Override Table

Tracks which arrays are currently transposed

More details in paper

Outline

Plain C Code

2

Unified Abstraction

Agnostically represents
1. In-core compute
2. In-memory compute
3. Near-memory compute

4

Partitioning
Static vs Dynamic

Portable binary without exposing µArch details to compiler

3

µArch-specific Optimizations

Specialize binary to take advantage of µArch details

µArch Extensions

Hardware support for compute paradigms

1

Compute Paradigms

In-Core

Core

Memory

In-Memory

Memory

Near-Memory

Memory

Outline

Plain C Code

Static

Unified Abstraction

Agnostically represents
1. In-core compute
2. In-memory compute
3. Near-memory compute

Dynamic

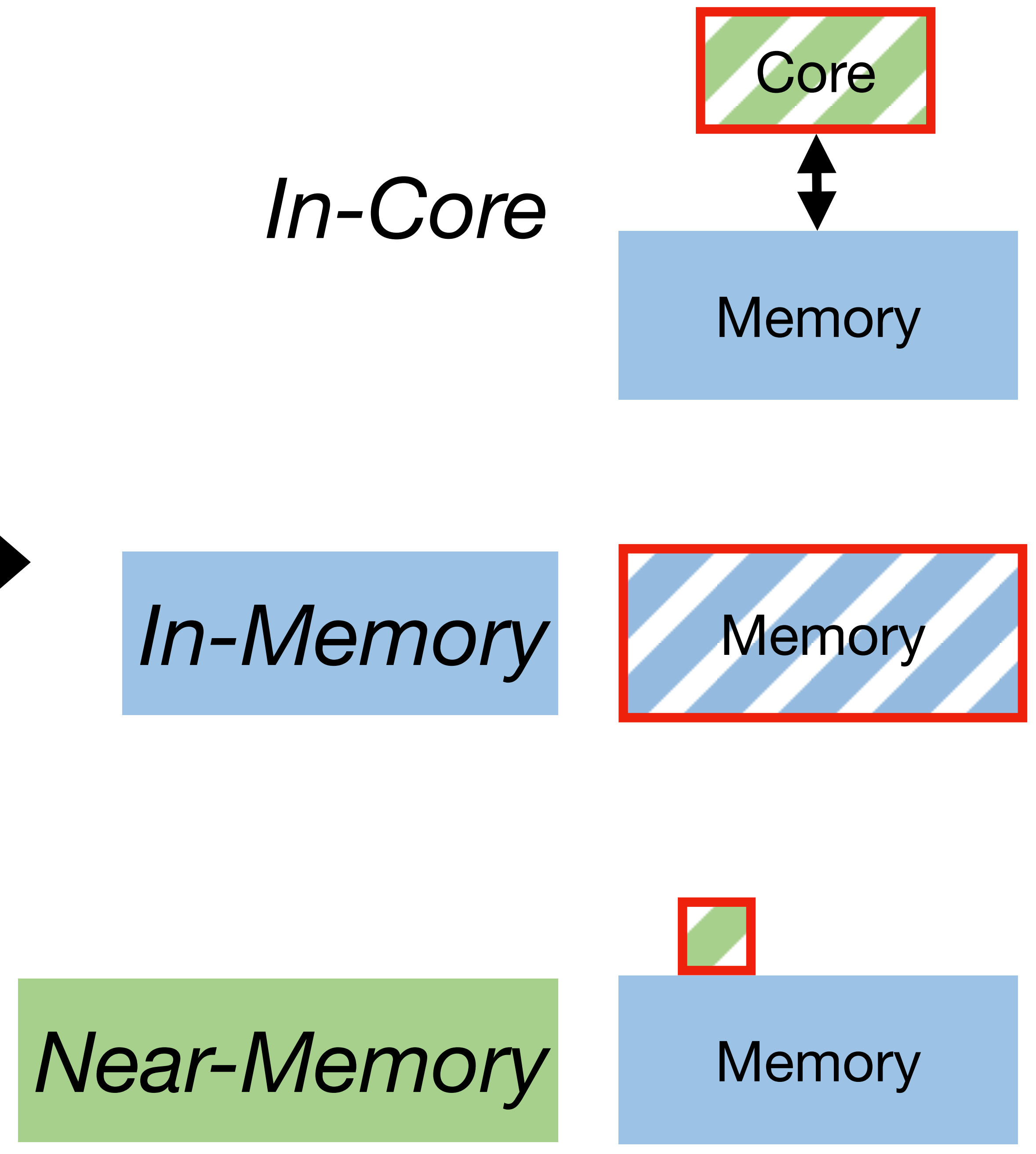
µArch-specific Optimizations

Specialize binary to take advantage of µArch details

µArch Support

Hardware support for compute paradigms

Compute Paradigms



Key Challenges

Plain C Code

Static

Unified Abstraction

Agnostically represents
1. In-core compute
2. In-memory compute
3. Near-memory compute

Dynamic

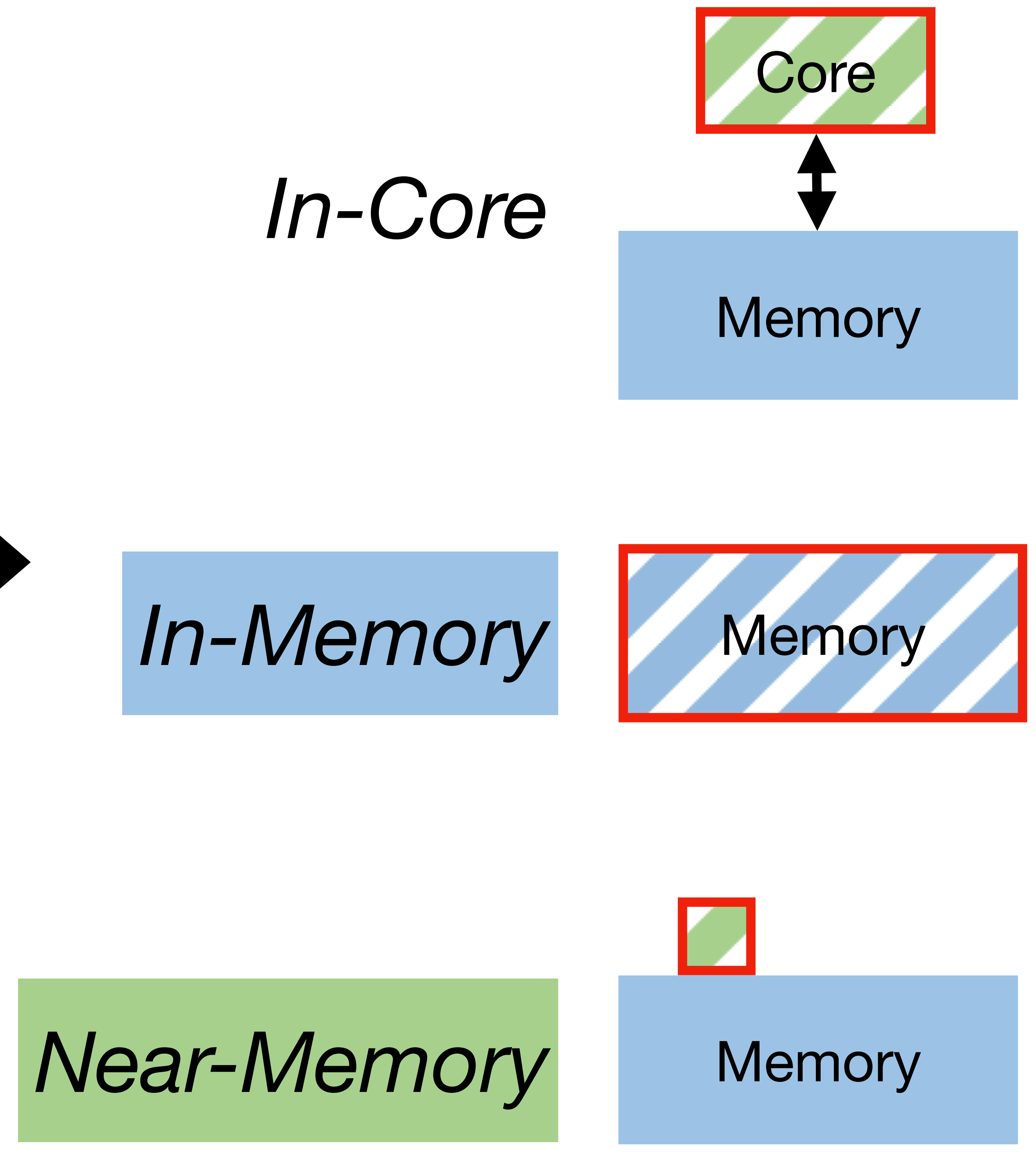
µArch-specific Optimizations

Specialize binary to take advantage of µArch details

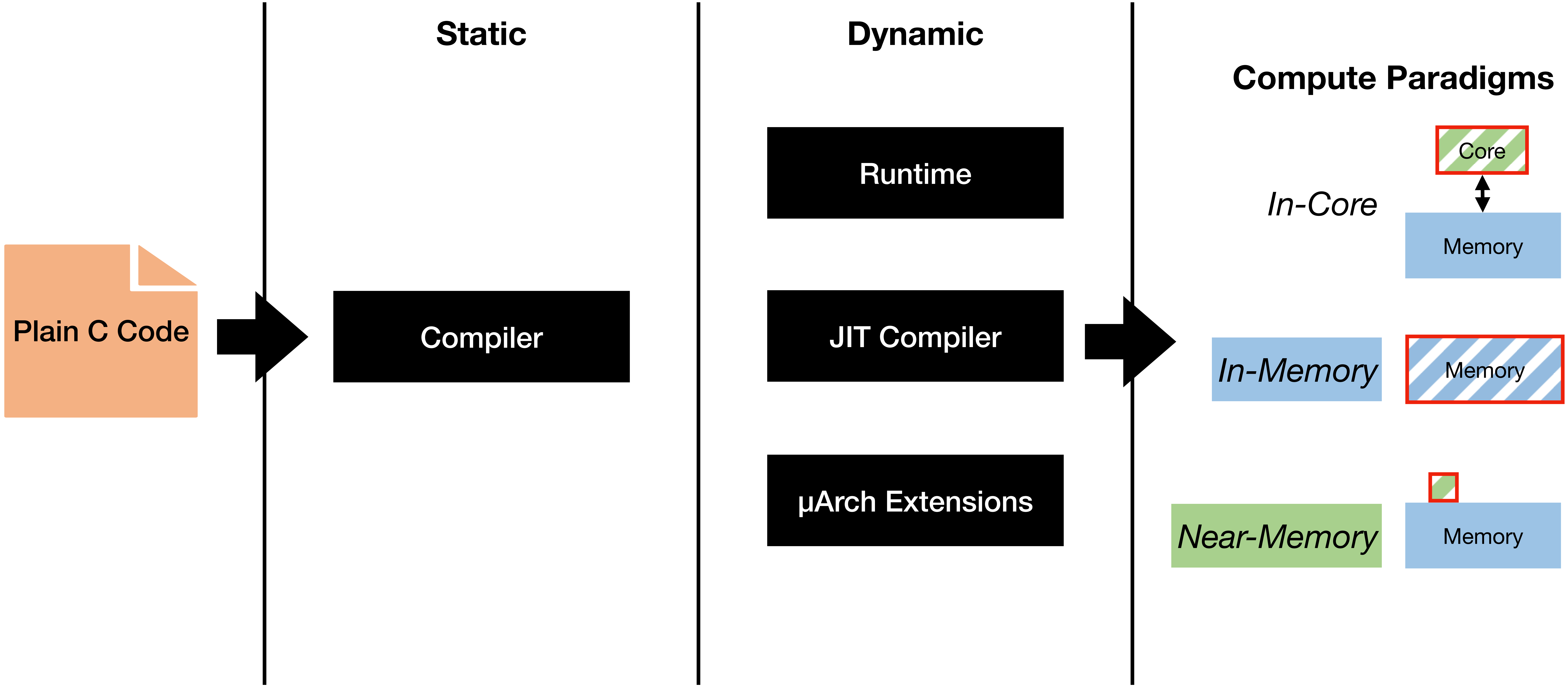
µArch Extensions

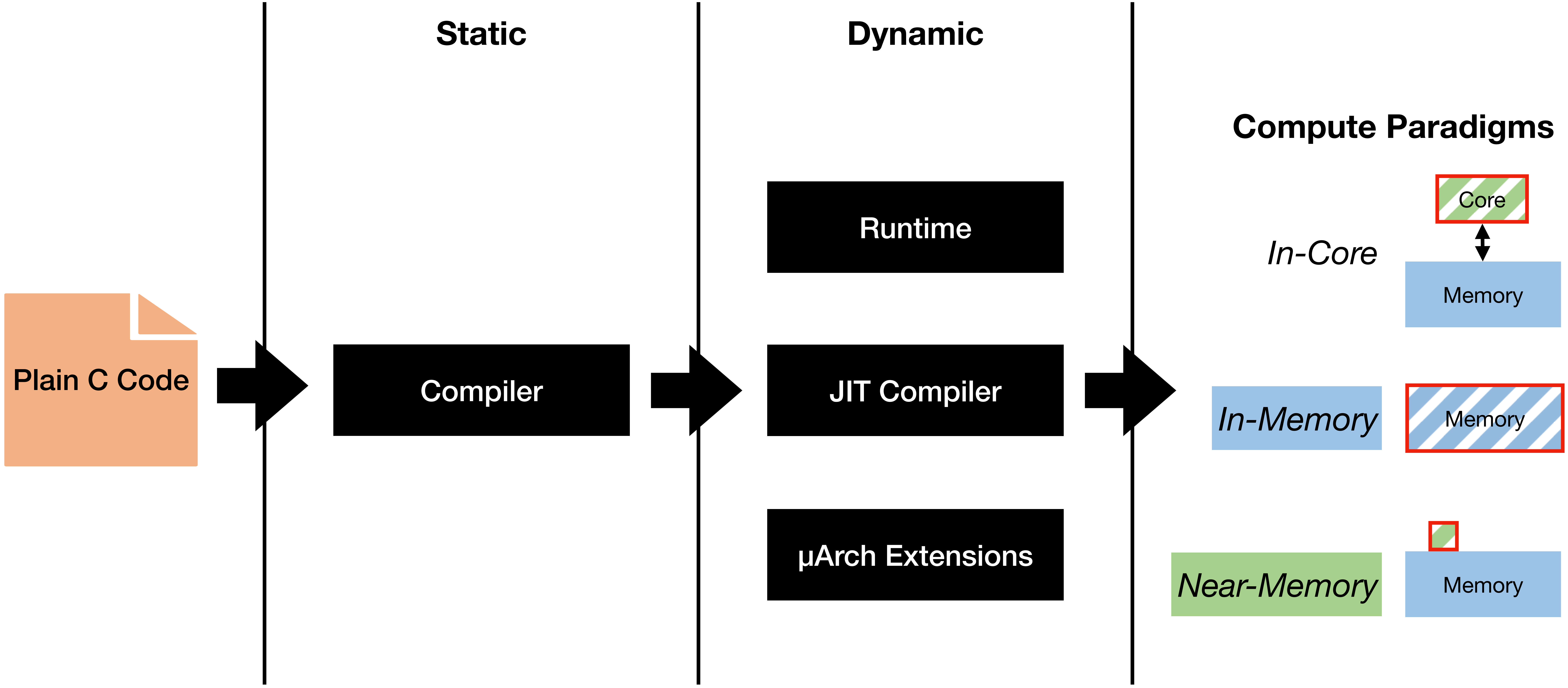
Hardware support for compute paradigms

Compute Paradigms



Outline

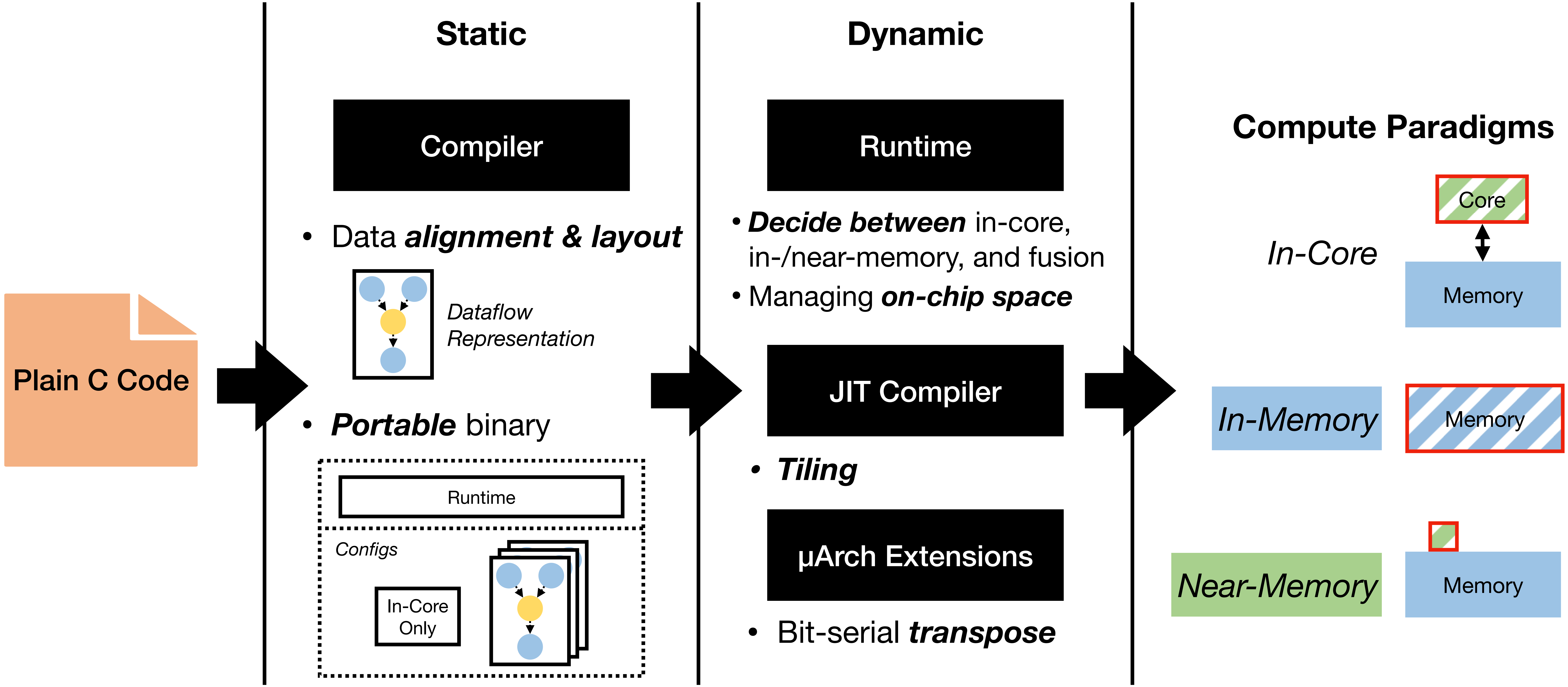

Static
Dynamic
Compute Paradigms
Plain C Code
Compiler
Runtime
JIT Compiler
µArch Extensions
In-Core
Core
Memory
In-Memory
Memory
Near-Memory
Memory



Requirements

- Data ***alignment & layout***
- ***Portable*** binary
- ***Decide between*** in-core, in-/near-memory, and fusion
- Managing ***on-chip space***
- ***Tiling***
- Bit-serial ***transpose***

The Infinity Stream Approach



Simulation Methodology

Simulator

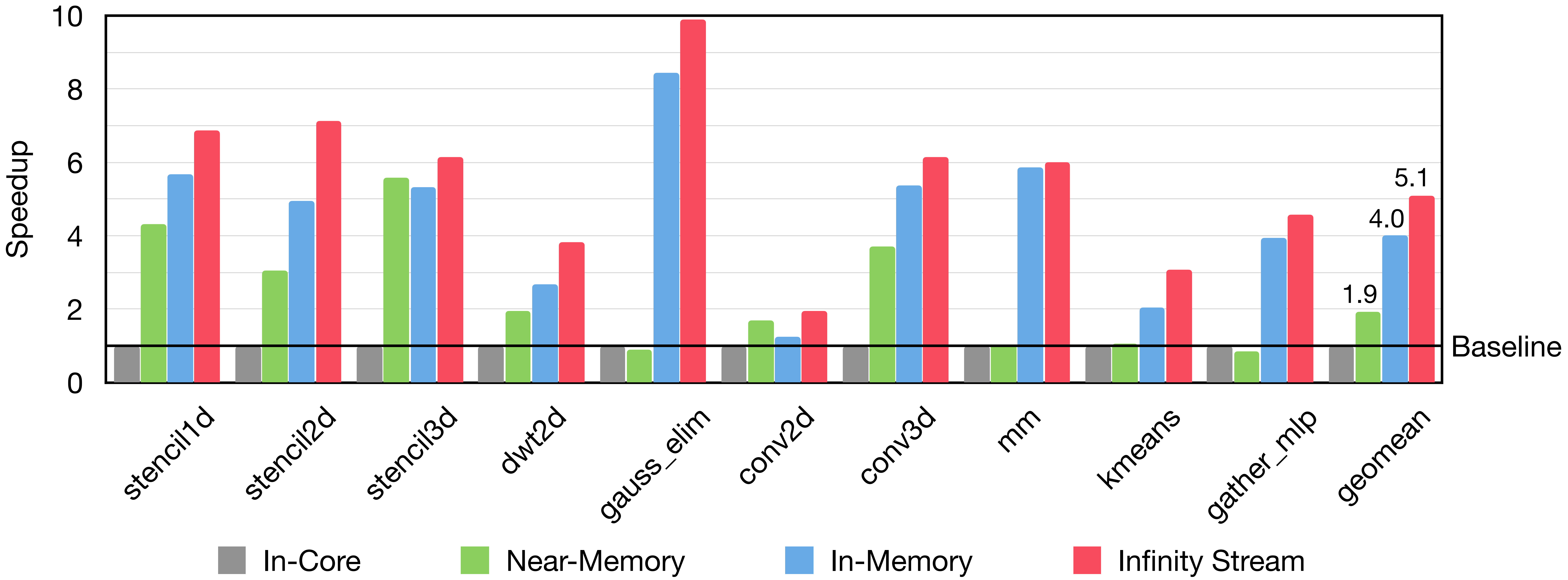
gem5 with partial AVX-512 support

Hardware Configuration

- 64 Cores (8x8)
- 18 ways (16 SRAM arrays/way)
- 8KB SRAM arrays (256x256)

L3 Total: 144MB

Fused Execution Speedup



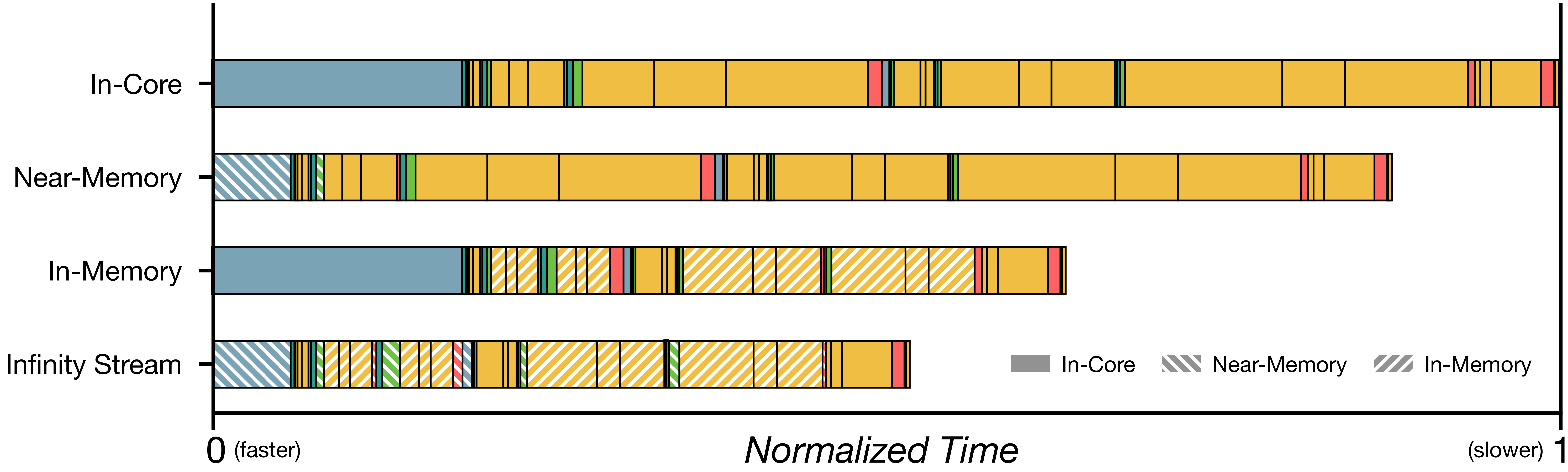
- 64 Cores (8x8)
- 18 ways (16 SRAM arrays/way)
- 8KB SRAM arrays (256x256)

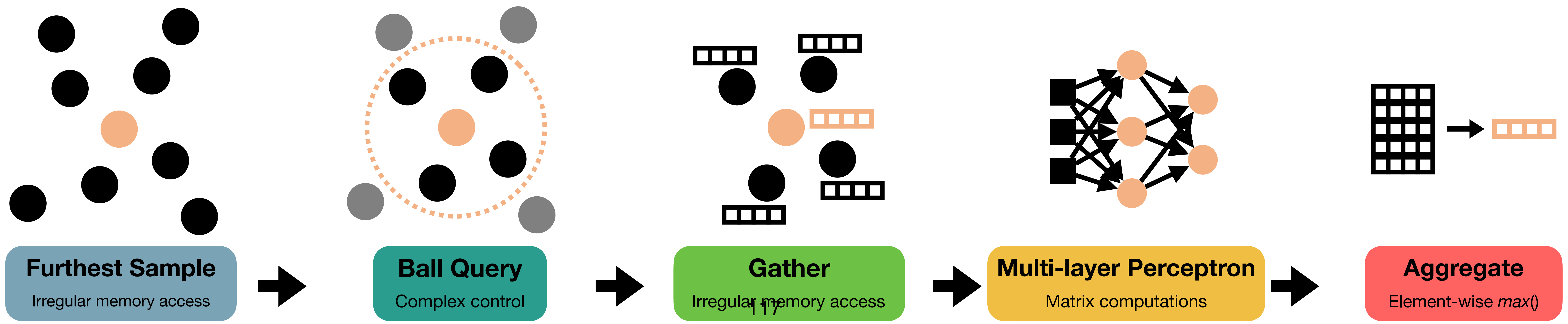
L3 Total: 144MB

Area 6.52% overhead

Energy Efficiency
- 5.6x over In-Core
- 2.4x over Near-Memory
- 1.6x over In-Memory

Case Study: PointNet++





Stream S
Memory Access Pattern

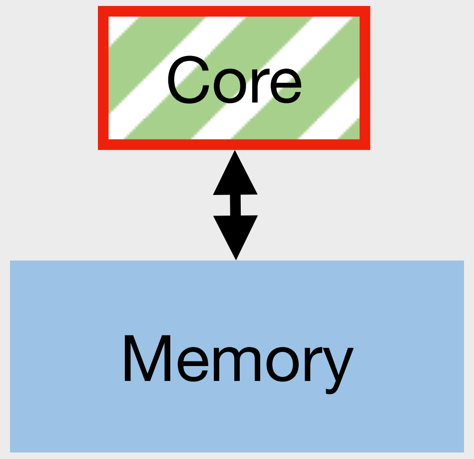
Tensor T
Vectorized Memory Access

Move mv
Virtual Vector Alignment

Broadcast bc
Spatial Reuse

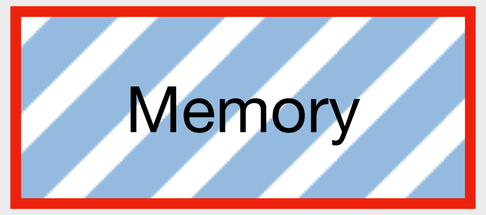
Global Lattice Space
Virtual Vector Lanes

In-Core
👍 Complex Control Flow



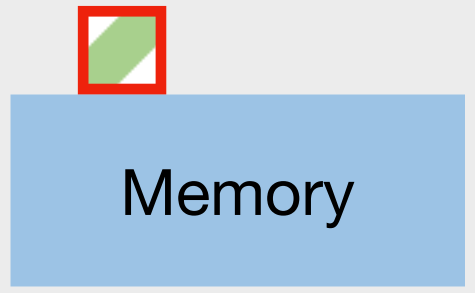
👎 Von Neumann Bottleneck

In-Memory
👍 Massive Vector Processing



👎 Strict Alignment Req.

Near-Memory
👍 Complex Memory Patterns



👎 Low Compute Width

Area
6.52%
overhead

Energy Efficiency
5.62x

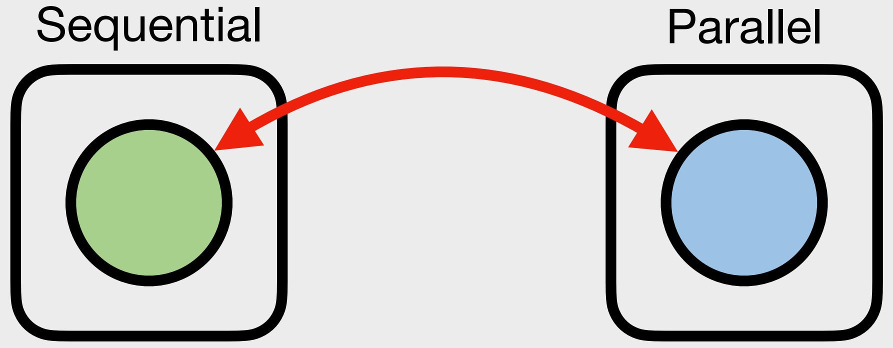
Infinity Stream

Tensor Transpose Unit

Layout Override Table

uArch Extensions

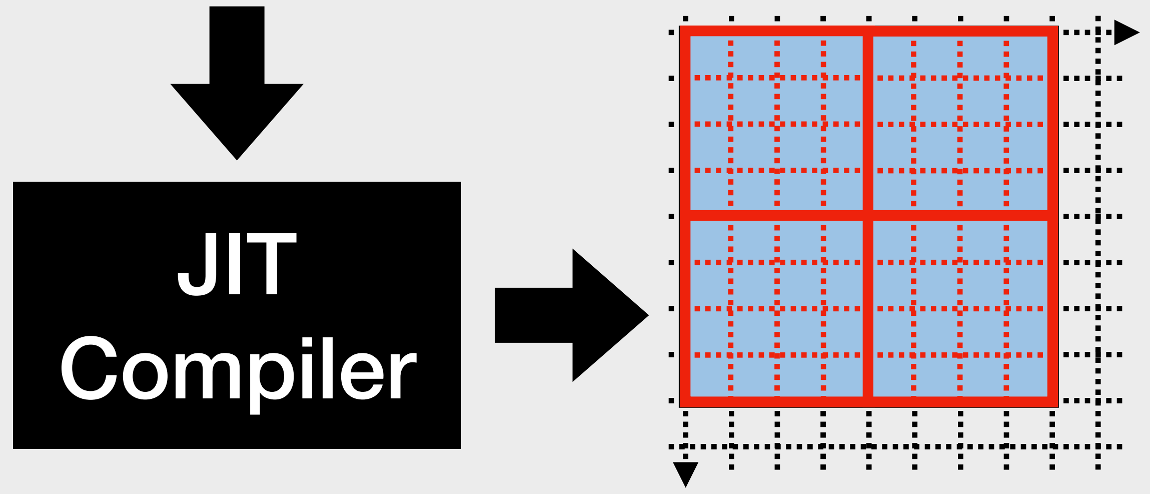
In-/Near-Memory Fusion



Supports memory irregularity

Extremely high compute width

💡 Hints



Runtime Tiling
Minimizes Network Traffic

Fusion Speedup