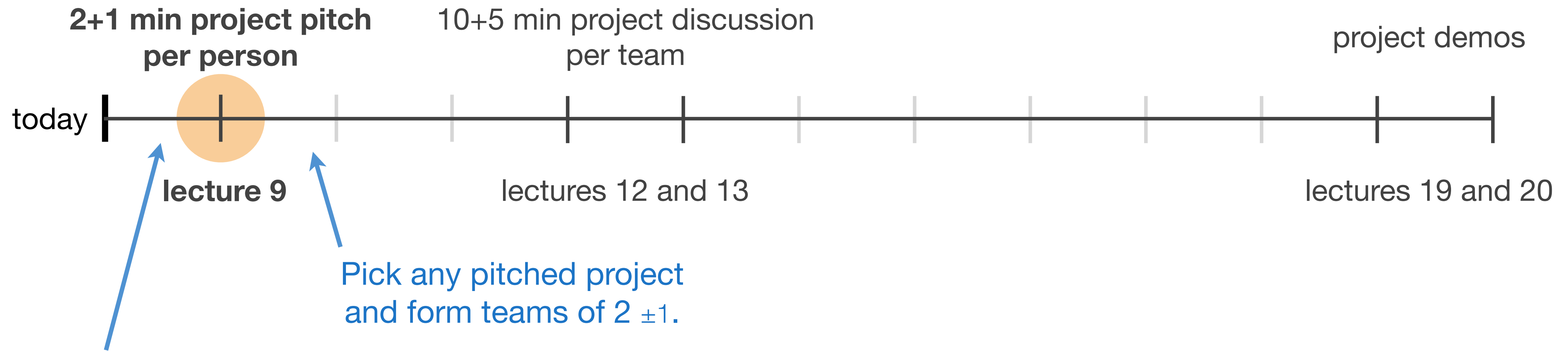


# Lecture 8 - Sparse Iteration Theory II

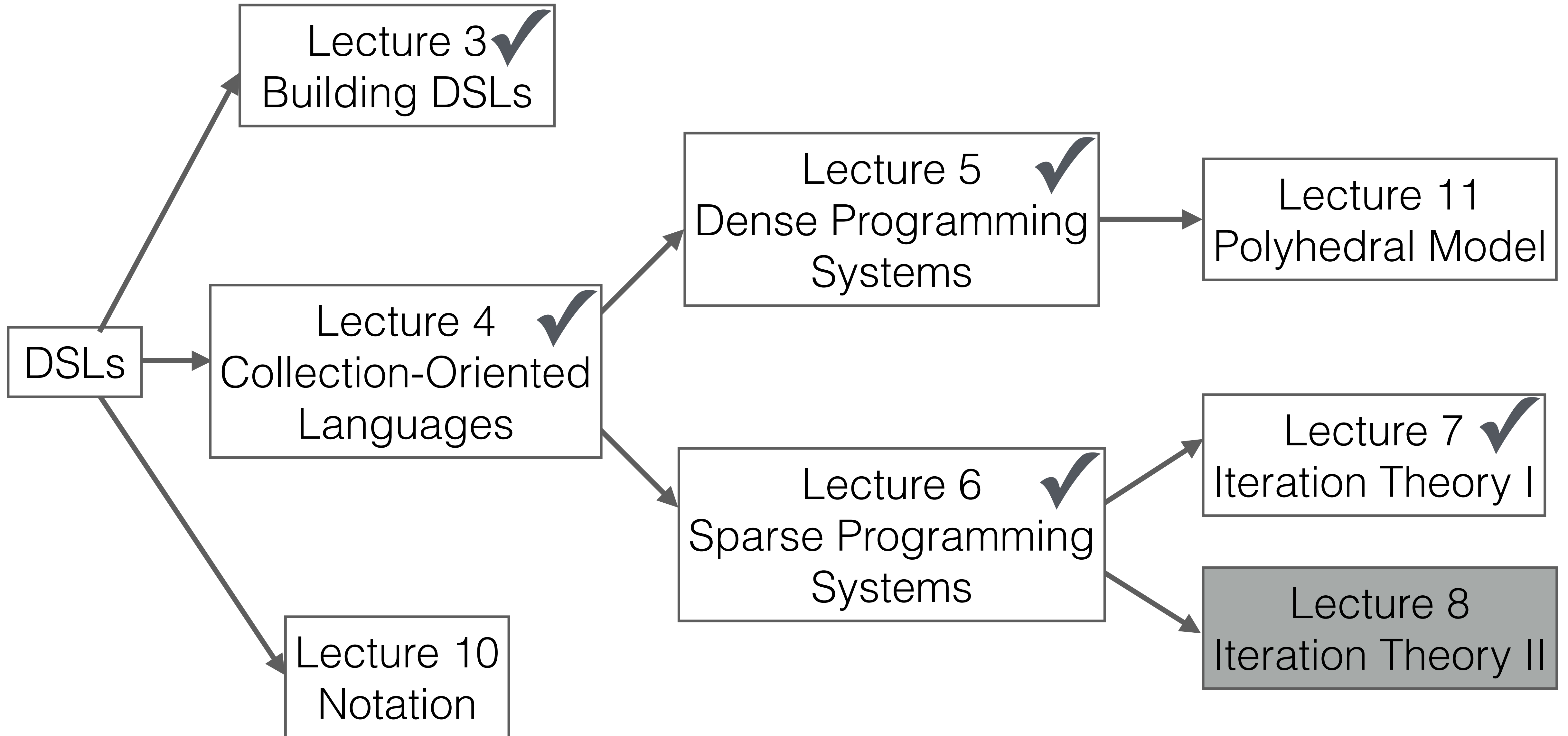
Stanford CS343D (Fall 2020)  
Fred Kjolstad and Pat Hanrahan

# Course Project



Each person contributes one pitch slide to a google slide deck. These pitches are not binding.

# Overview of lectures in the coming weeks



# Overview of topics

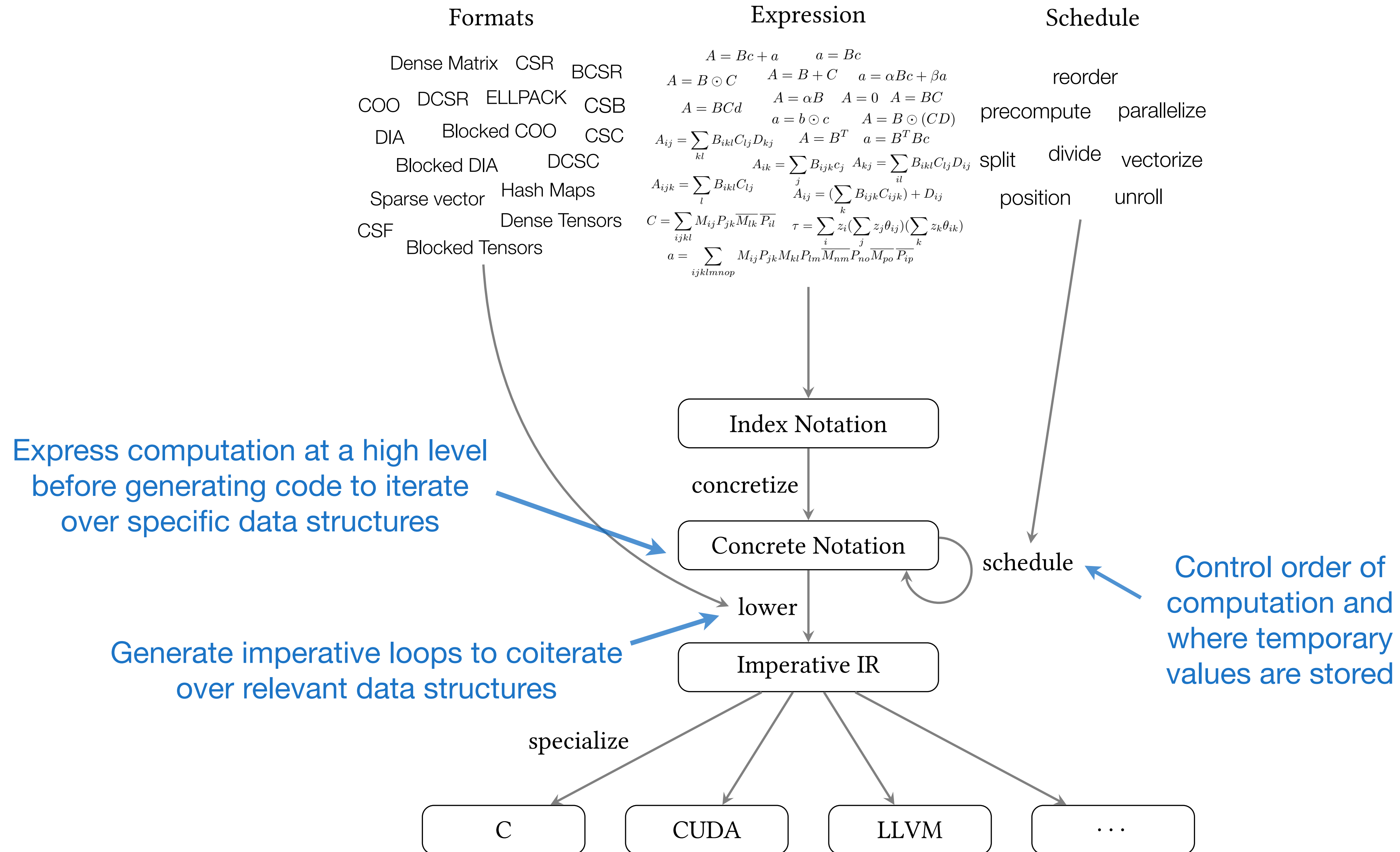
## Lecture 7

- Data representation
- Iteration spaces
- Iteration graph IR
- Iteration lattices to represent coiteration

## Lecture 8

- Concrete index notation IR
- Code generation algorithm
- Derived iteration spaces
- Optimizing transformations

# Overview of compilation stages



# Concrete index notation specifies order of computations and location of intermediate values

$$A_{ij} = B_{ij} + C_{ij}$$



$$\forall_i \forall_j A_{ij} = B_{ij} + C_{ij}$$

$$\alpha = \sum_i b_i c_i$$



$$\forall_i \alpha + = b_i c_i$$

$$a_i = \sum_j B_{ij} c_j$$



$$\forall_i \forall_j a_i = t \text{ where } t + = B_{ij} c_j$$

# Concrete index notation grammars

## Assignment statement

$A_{i\dots} = \text{expr}$

## Forall statement

$\forall_i \text{ stmt}$

## Where statement

$\text{stmt}_c \textbf{ where } \text{stmt}_p$

## Environment

index index  $\xrightarrow{\text{"collapse"}}$  index

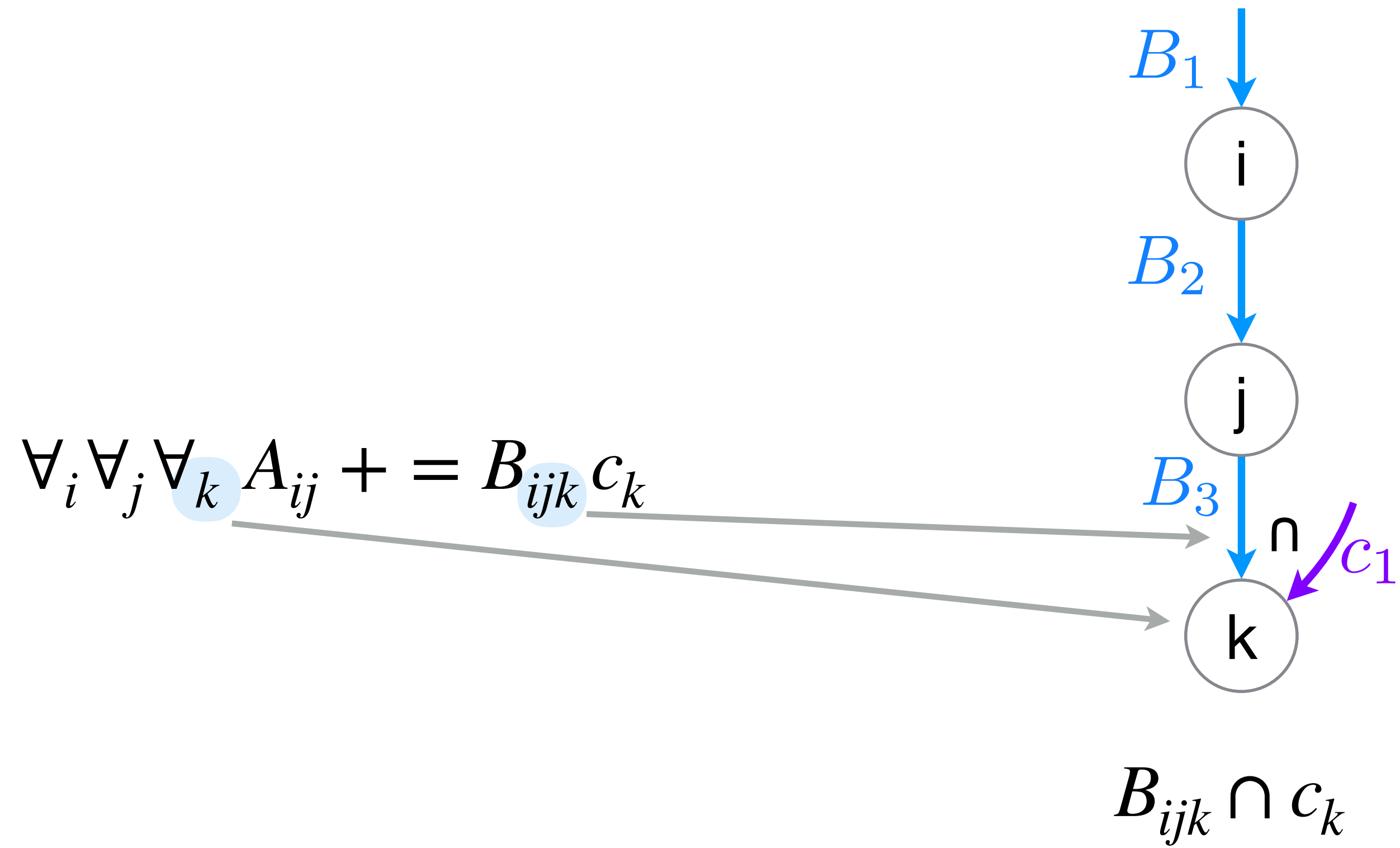
index  $\xrightarrow{\text{"split(" d "," s ")}}$  index index

$\text{"bound(" index "," b ")}$

$\text{"parallelize(" index "," p "," r ")}$

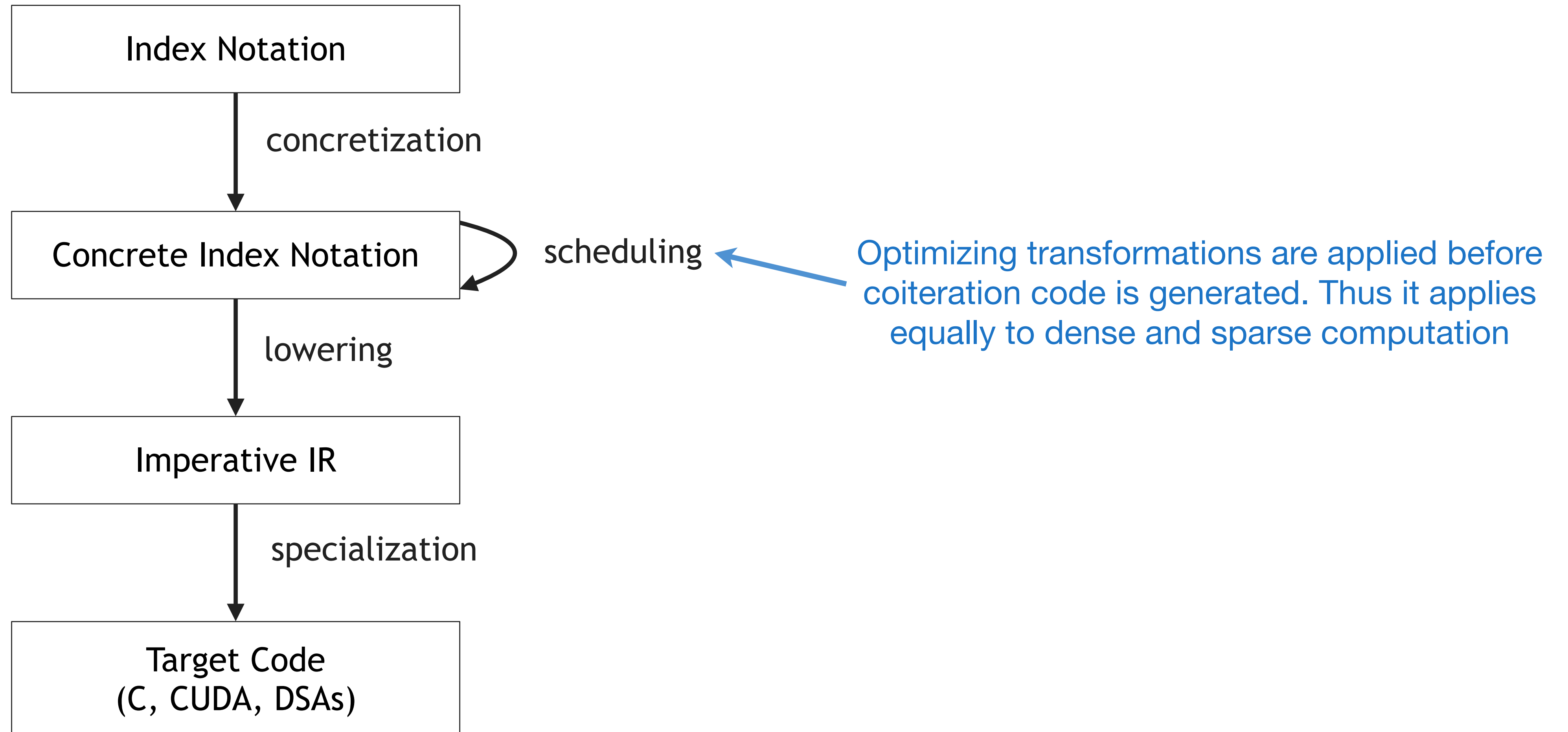
$\text{"unroll(" index "," u ")}$

# Concrete index notation contains iteration graphs

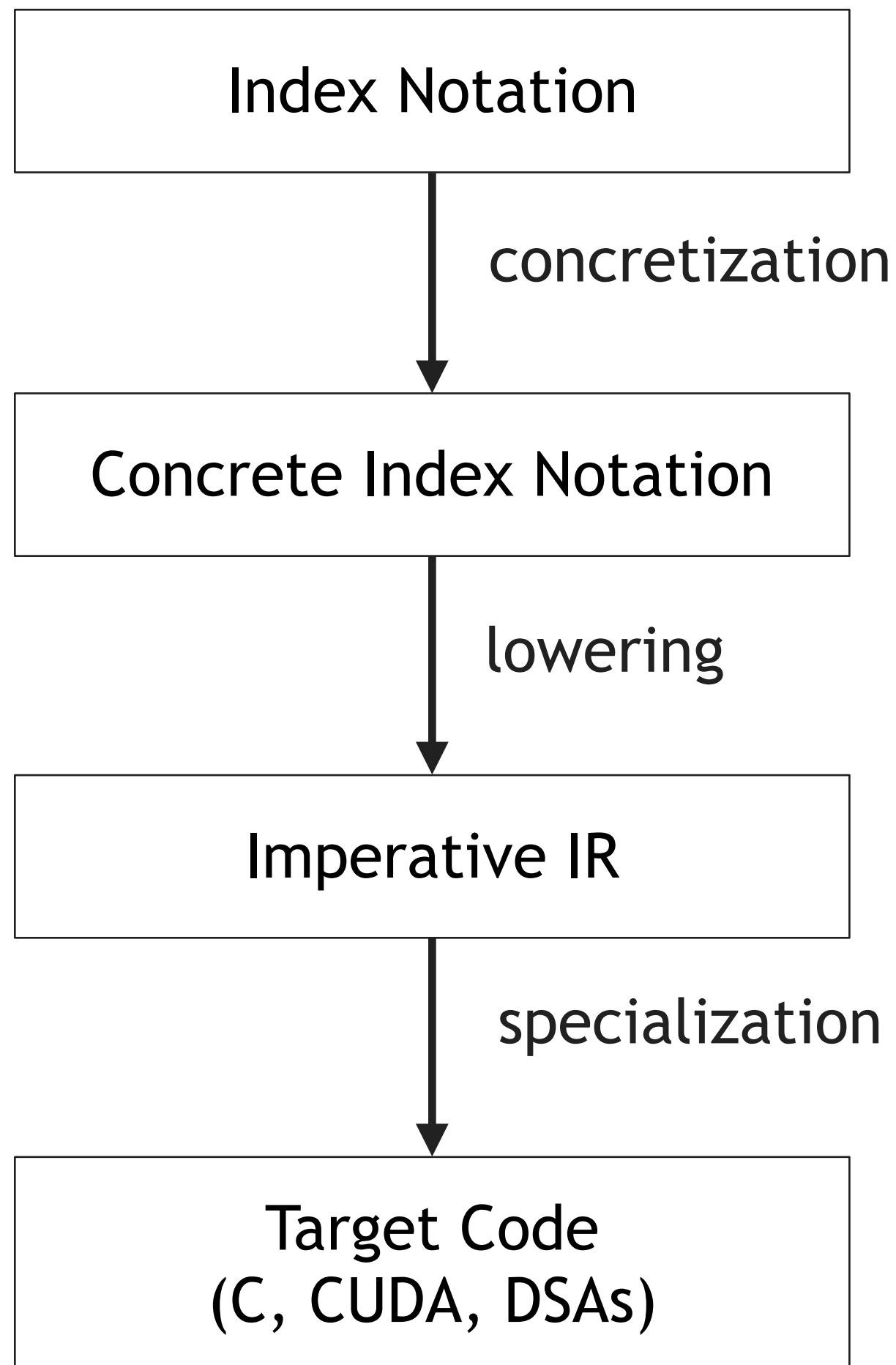




# Concrete index notation as an optimization IR



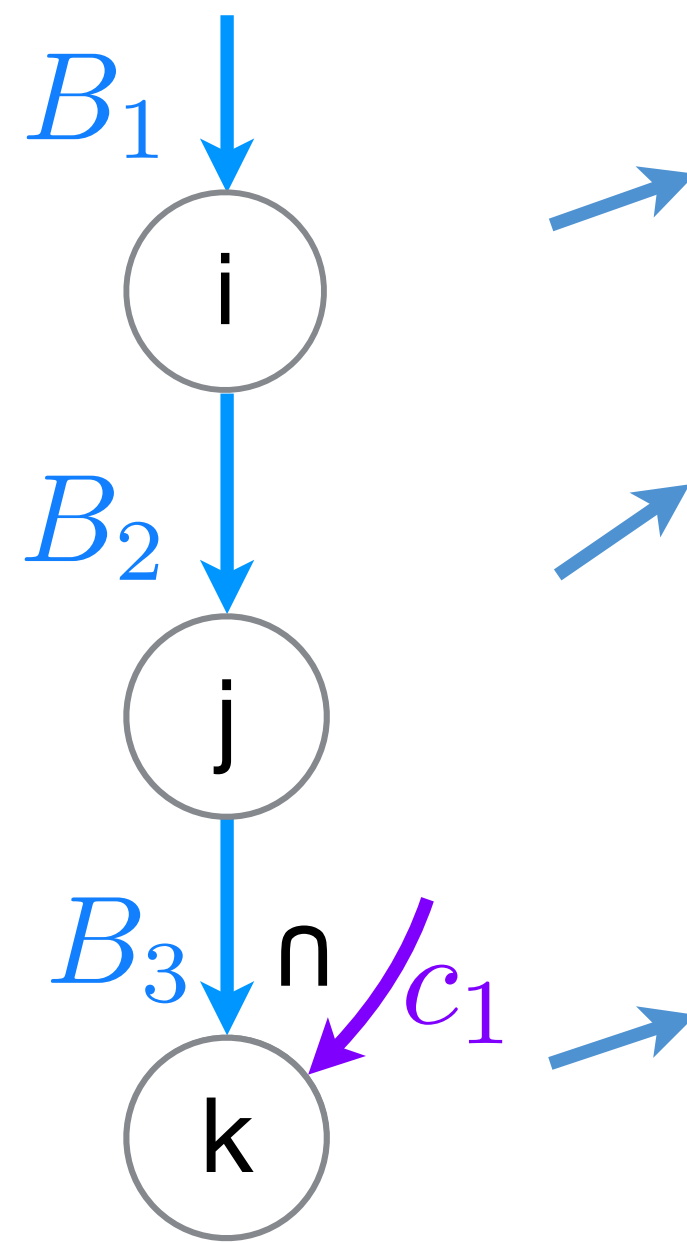
# Concrete index notation example



$$A_{ij} = \sum_k B_{ijk} c_k$$

Order of computation

$$\forall_i \forall_j \forall_k A_{ij} += B_{ijk} c_k$$



```

for (int i = 0; i < m; i++) {
  for (int pB2 = B2_pos[i]; pB2 < B2_pos[i+1]; pB2++) {
    int j = B2_crd[pB2];

    int pA2 = i*n + j;
    int pB3 = B3_pos[pB2];
    int pc1 = c1_pos[0];
    while (pB3 < B3_pos[pB2+1] && pc1 < c1_pos[1]) {
      int kB = B3_crd[pB3];
      int kc = c1_crd[pc1];
      int k = min(kB, kc);
      if (kB == k && kc == k) {
        A[pA2] += B[pB3] * c[pc1];
      }
      if (kB == k) pB3++;
      if (kc == k) pc1++;
    }
  }
}
  
```

# Concrete index notation example

Index Notation

concretization

Concrete Index Notation

optimization

lowering

Imperative IR

specialization

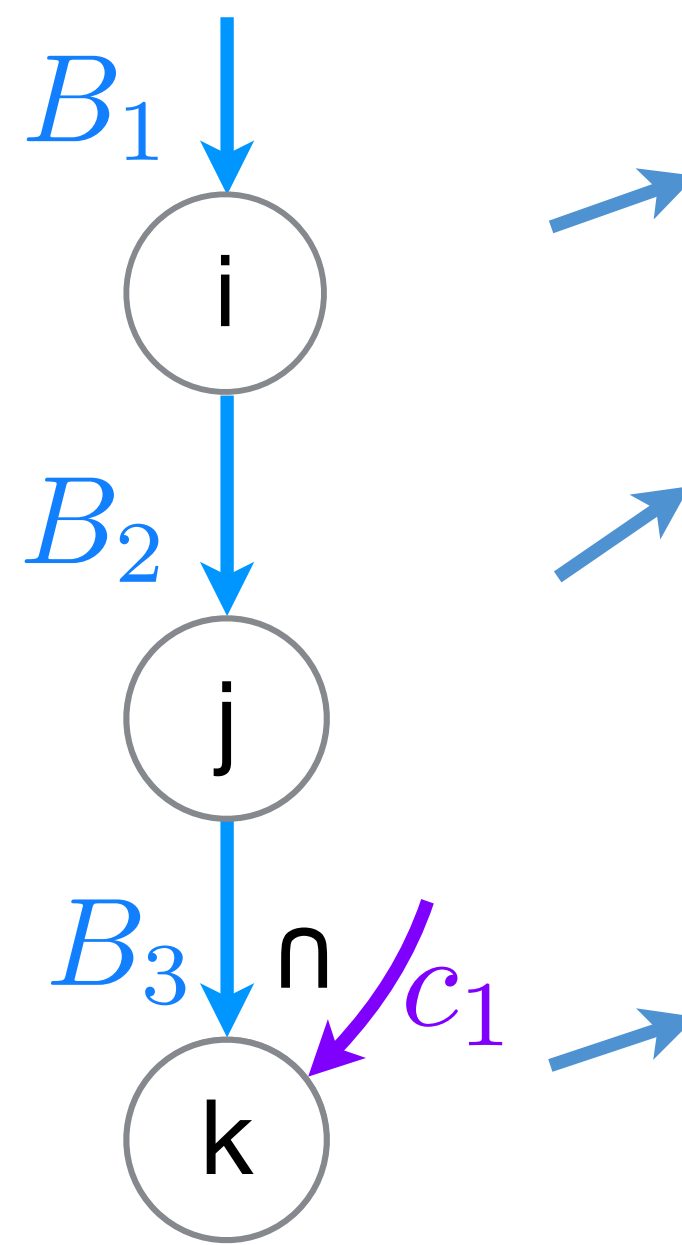
Target Code  
(C, CUDA, DSAs)

$$A_{ij} = \sum_k B_{ijk} c_k$$

Order of computation

Temporary storage

$$\forall_i \forall_j (A_{ij} = t) \text{ where } (\forall_k t += B_{ijk} c_k)$$



```

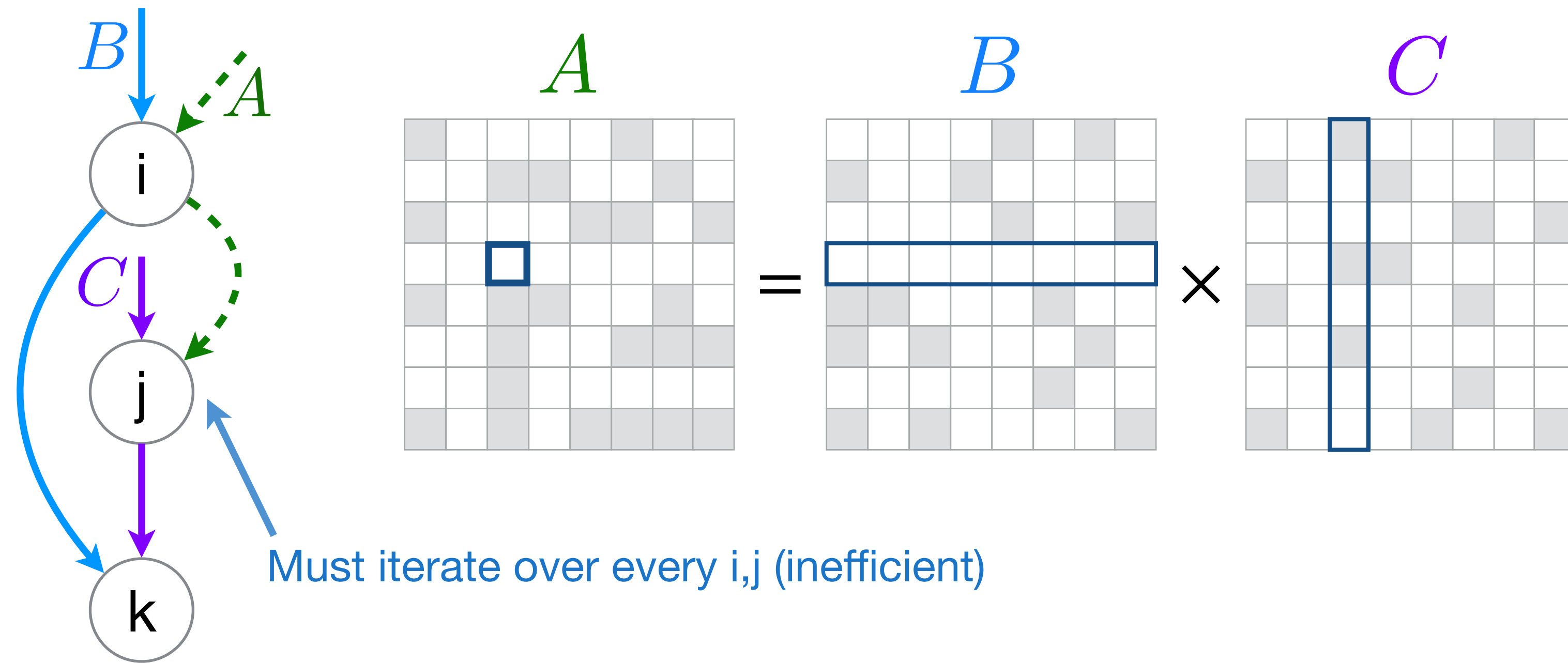
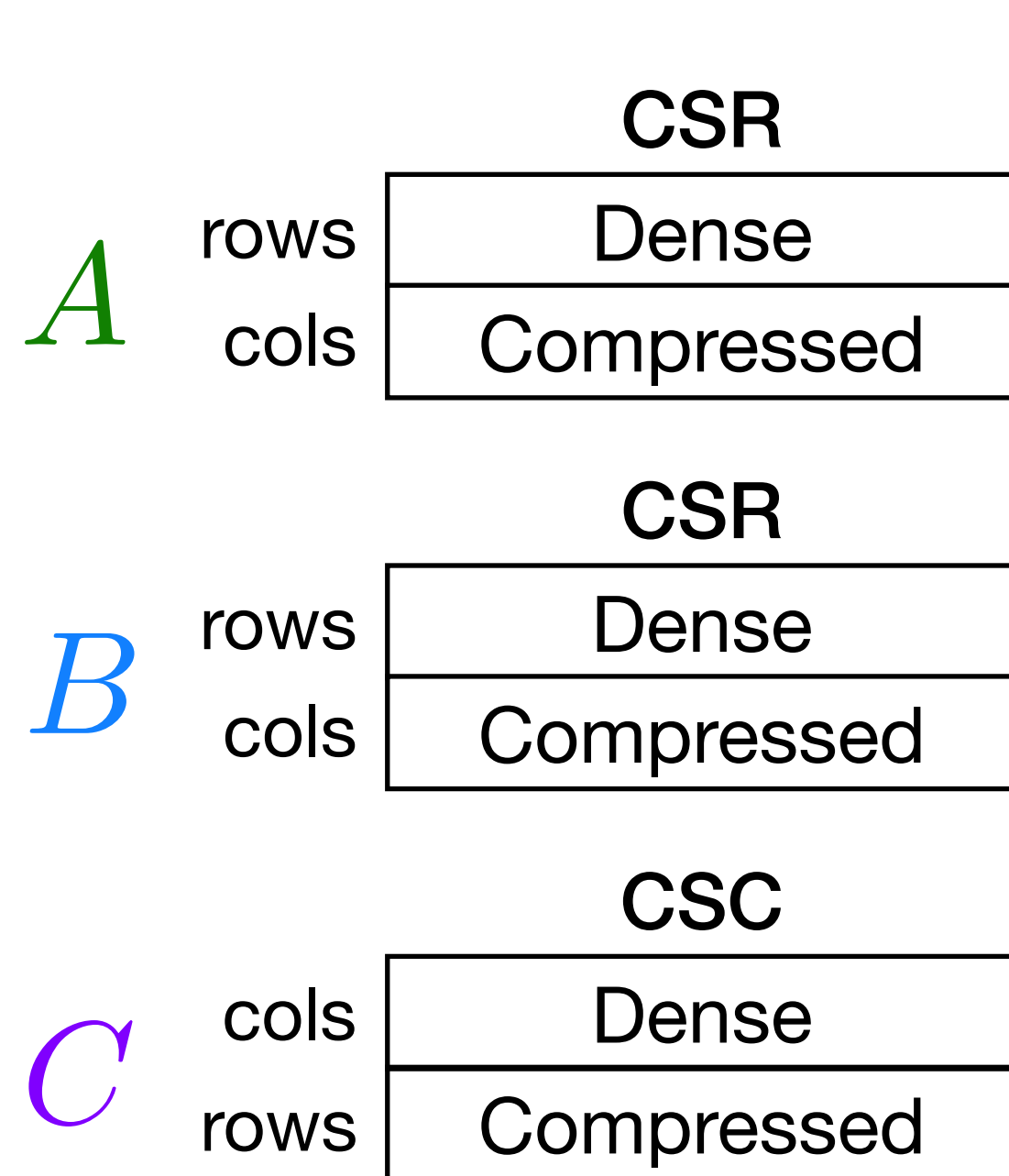
for (int i = 0; i < m; i++) {
  for (int pB2 = B2_pos[i]; pB2 < B2_pos[i+1]; pB2++) {
    int j = B2_crd[pB2];
    float t = 0.0;
    int pA2 = i*n + j;
    int pB3 = B3_pos[pB2];
    int pc1 = c1_pos[0];
    while (pB3 < B3_pos[pB2+1] && pc1 < c1_pos[1]) {
      int kB = B3_crd[pB3];
      int kc = c1_crd[pc1];
      int k = min(kB, kc);
      if (kB == k && kc == k) {
        t += B[pB3] * c[pc1];
      }
      if (kB == k) pB3++;
      if (kc == k) pc1++;
      A[pA2] = t;
    }
  }
}

```

# Workspace to scatter into results in sparse matrix multiplication

Inner Product  
Matrix Multiplication

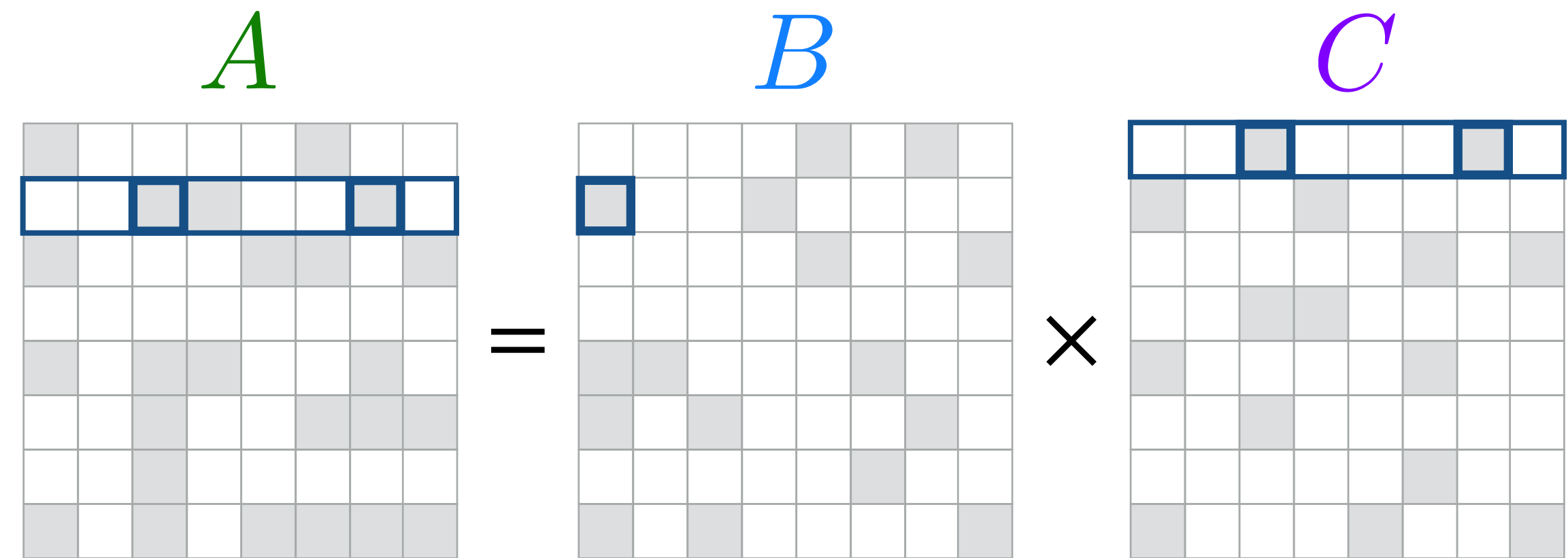
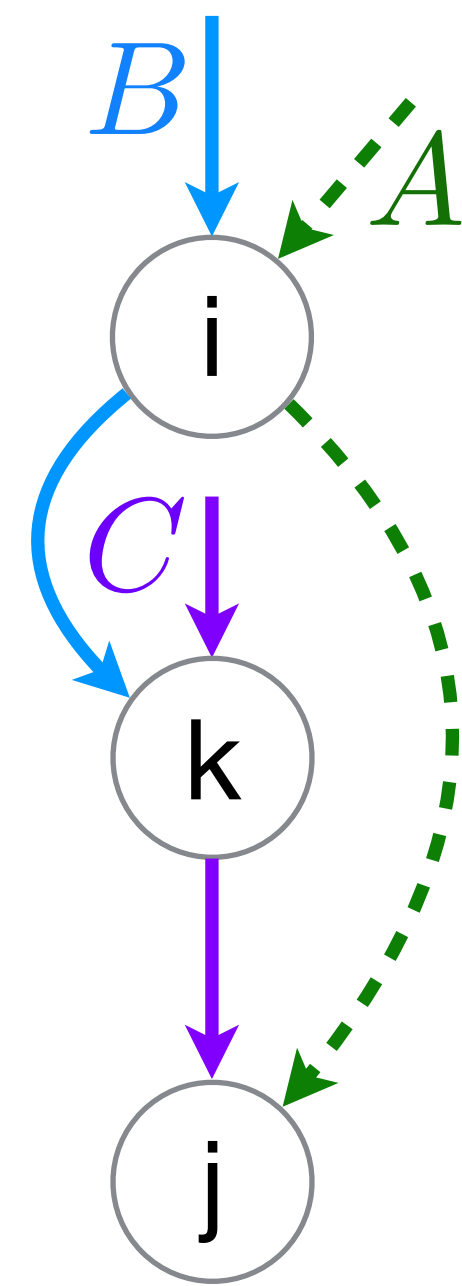
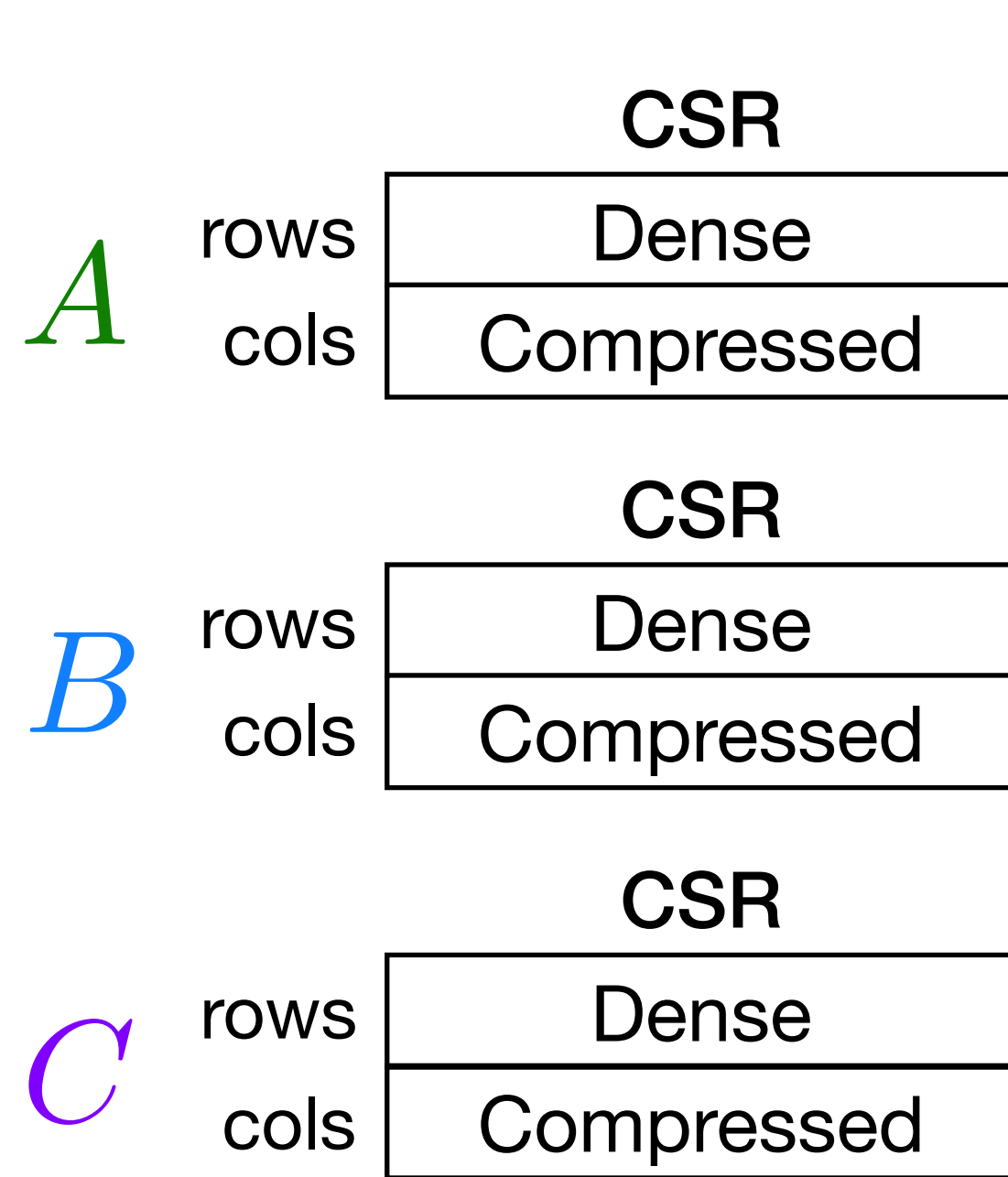
$$\forall_i \forall_j \forall_k \underbrace{A}_{\rightarrow} \underbrace{B}_{\rightarrow} \underbrace{C}_{\leftarrow} =$$



# Workspace to scatter into results in sparse matrix multiplication

Linear Combination of Rows  
Matrix Multiplication

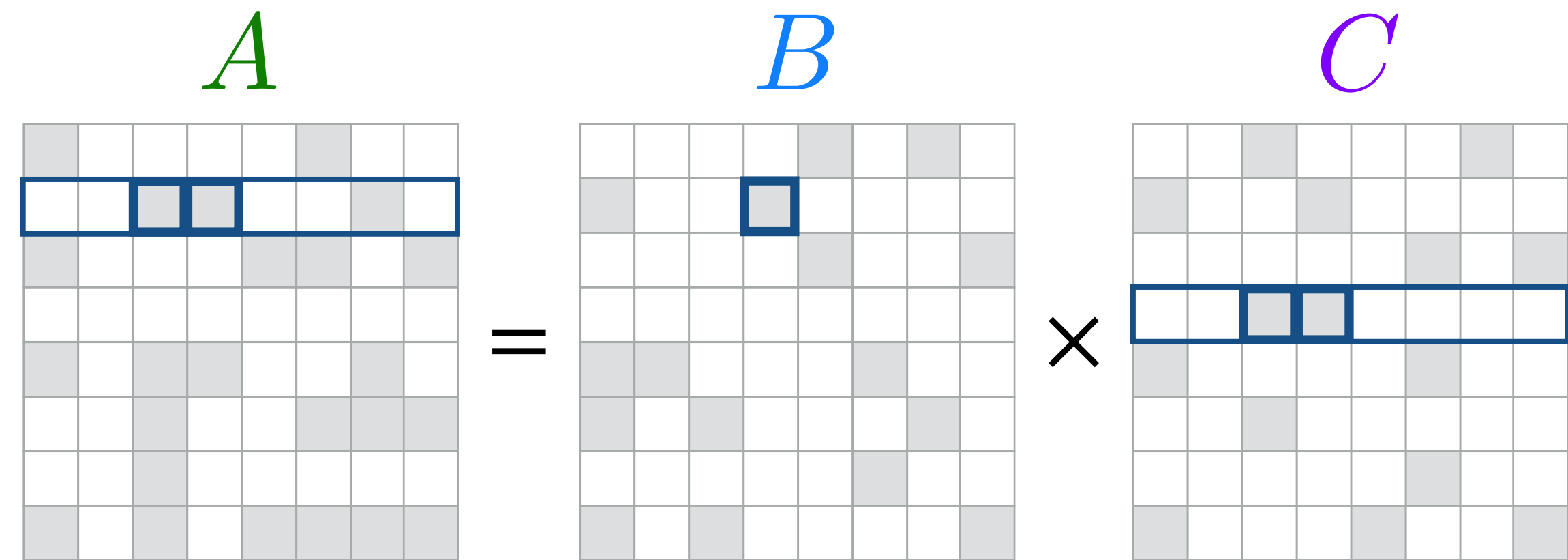
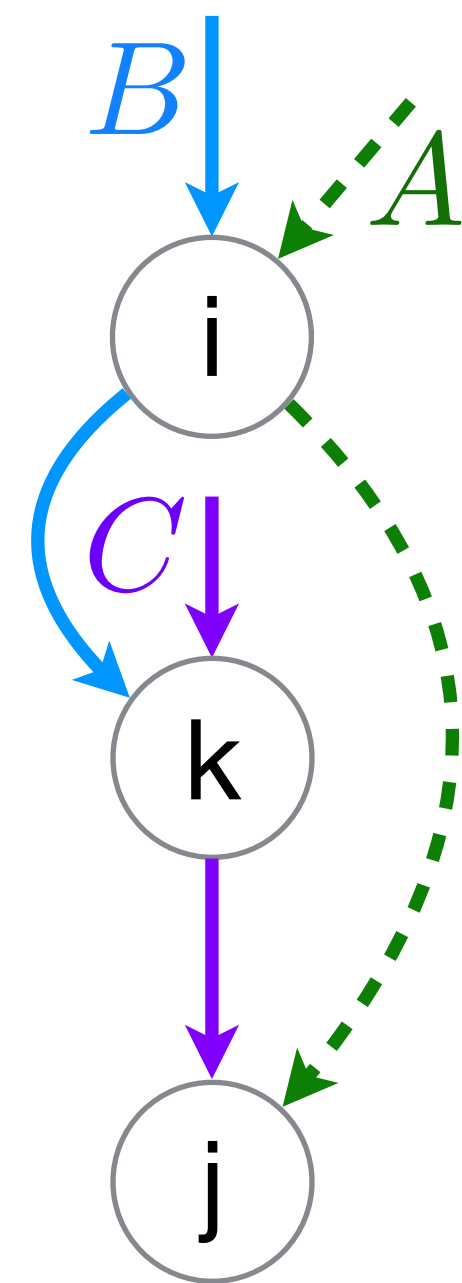
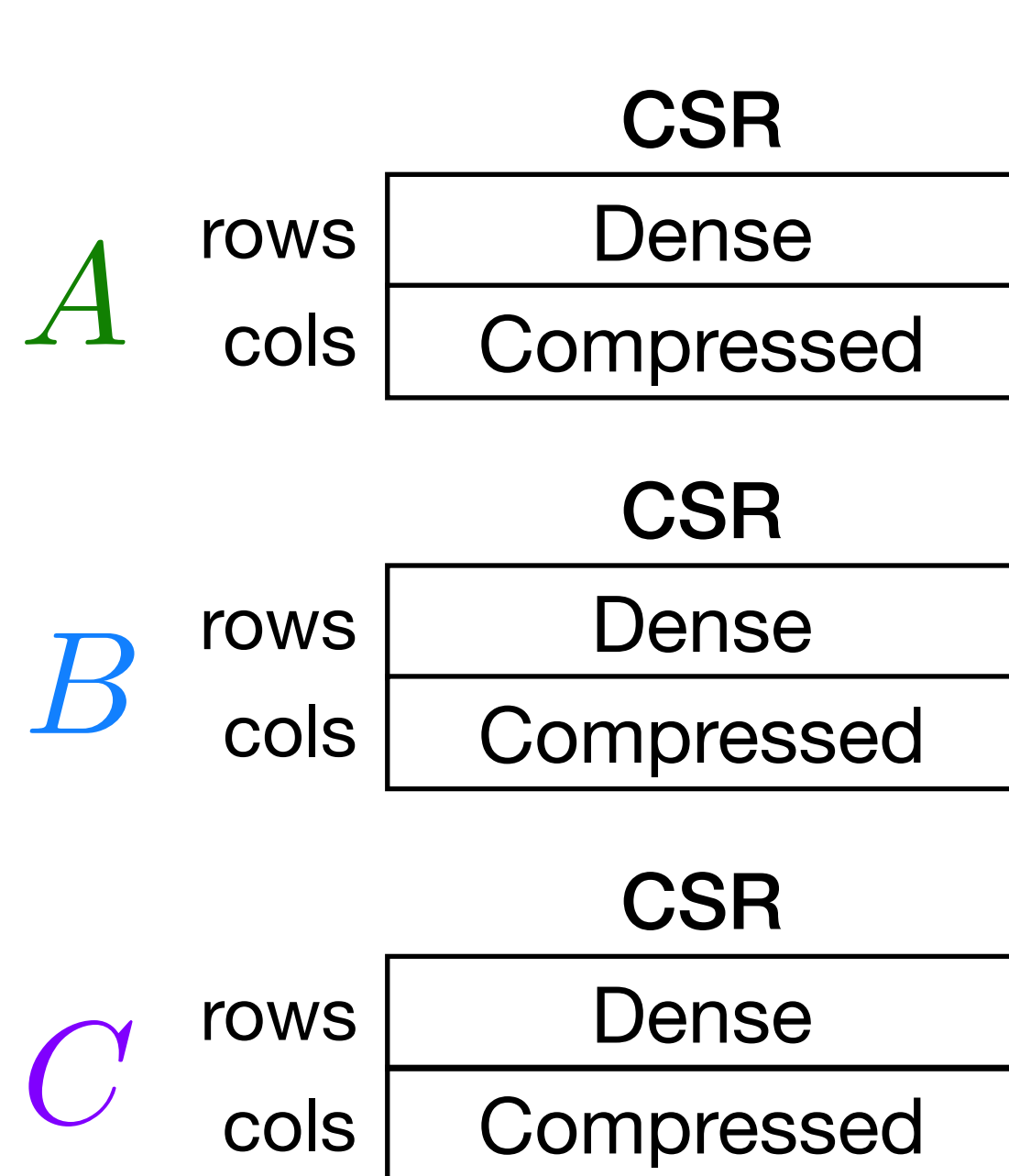
$$\forall_i \forall_k \forall_j \quad \underline{A}_{ij} \quad + = \quad \underline{B}_{ik} \underline{C}_{kj}$$



# Workspace to scatter into results in sparse matrix multiplication

Linear Combination of Rows  
Matrix Multiplication

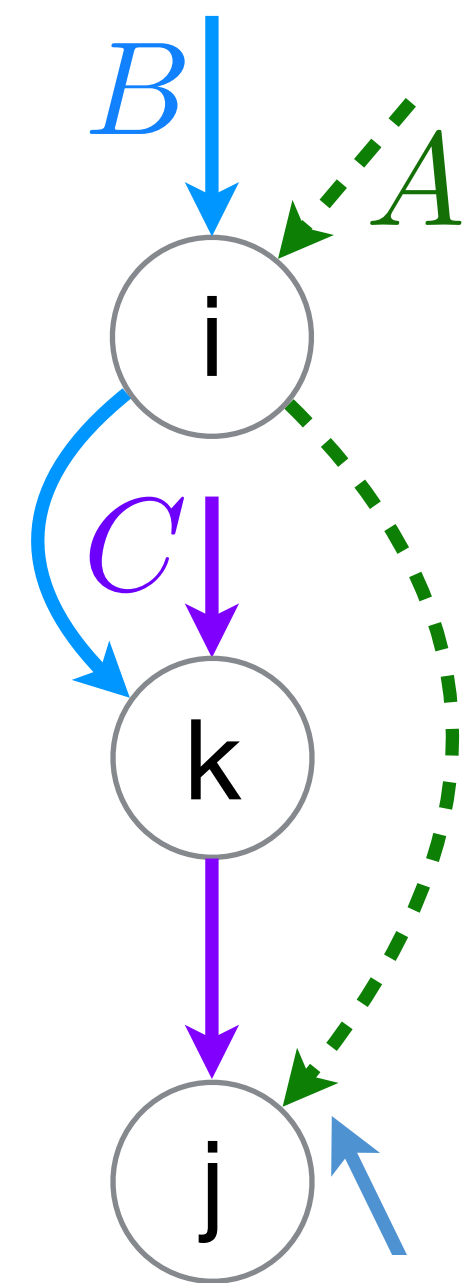
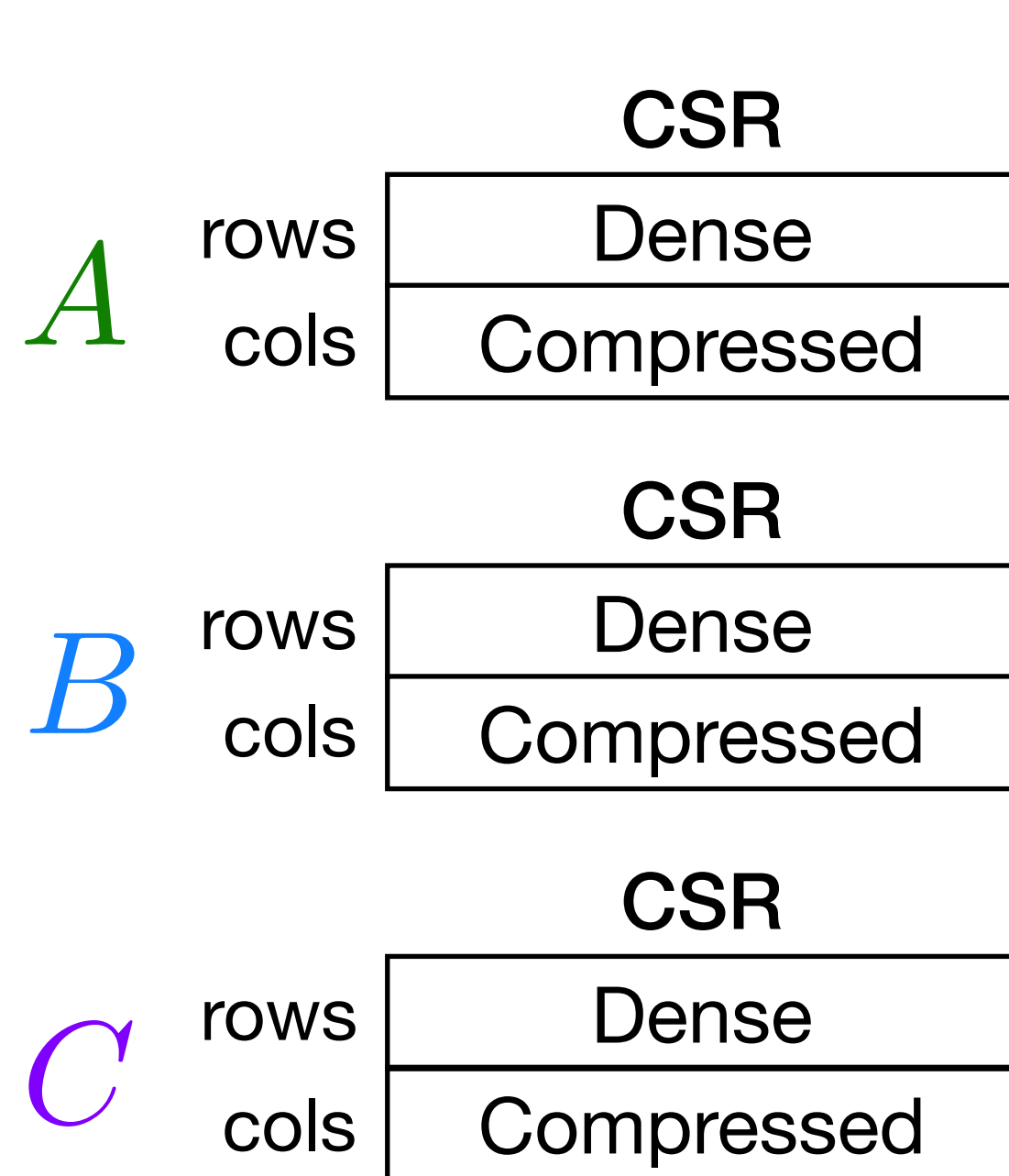
$$\forall_i \forall_k \forall_j \quad \underline{A}_{ij} \quad + = \quad \underline{B}_{ik} \underline{C}_{kj}$$



# Workspace to scatter into results in sparse matrix multiplication

Linear Combination of Rows  
Matrix Multiplication

$$\forall_i \forall_k \forall_j A_{ij} += B_{ik} C_{kj}$$



Scatter results into compressed A (inefficient)

```
for (int i = 0; i < m; i++) {
```

```
    for (int pB2 = B2_pos[i]; pB2 < B2_pos[i+1]; pB2++) {
        int k = B2_crd[pB2];
```

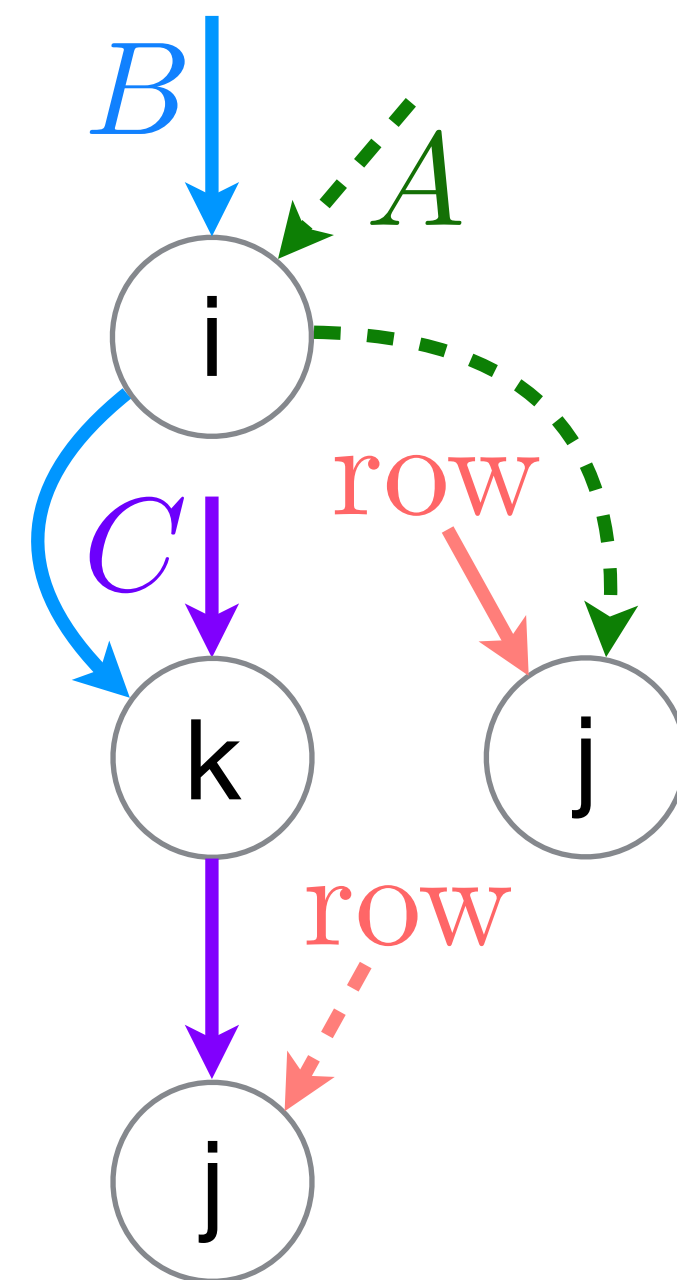
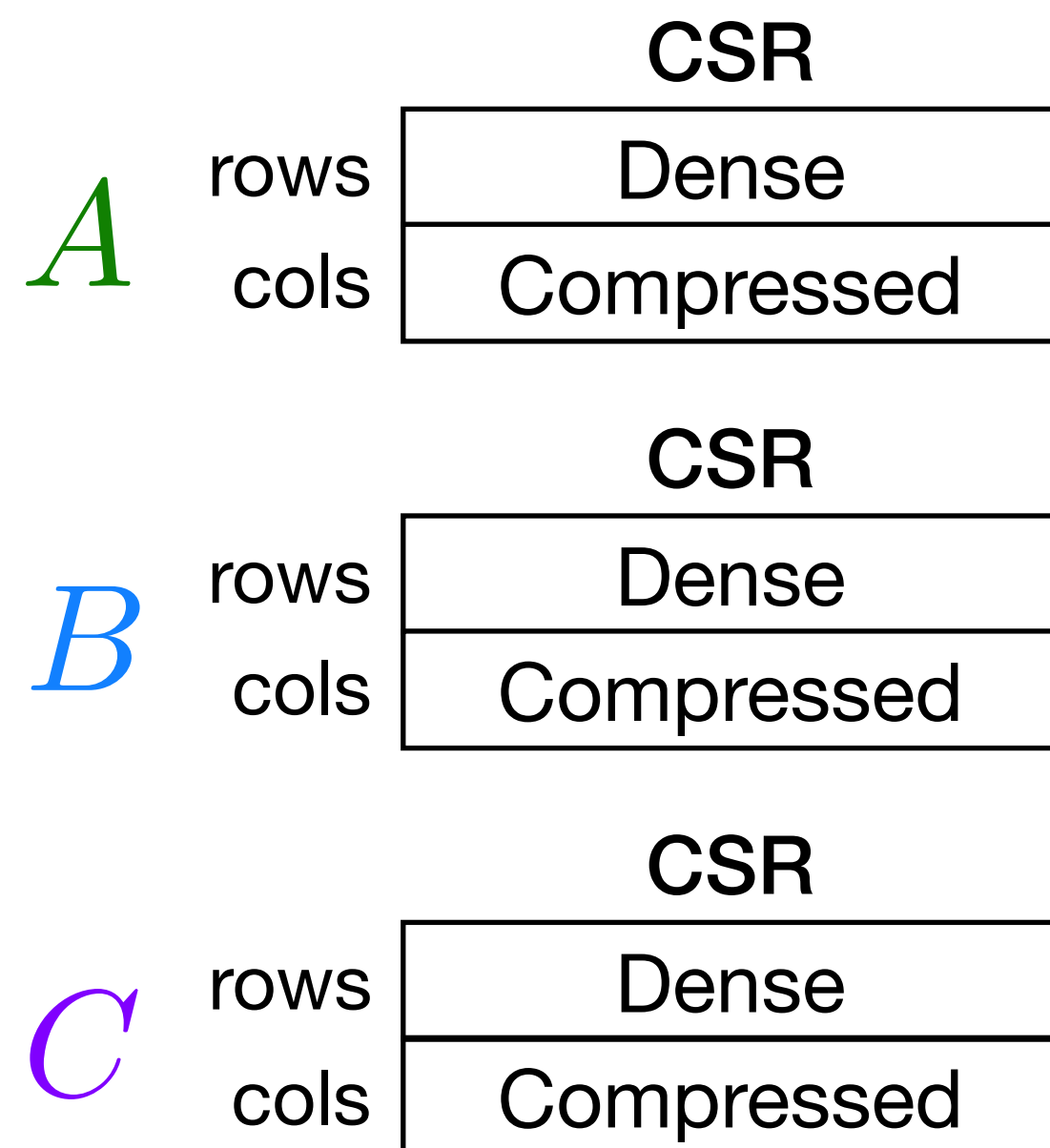
```
    }
```

```
}
```

# Workspace to scatter into results in sparse matrix multiplication

Linear Combination of Rows  
Matrix Multiplication

$$\forall_i \forall_j (A_{ij} = \text{row}_j) \text{ where } (\forall_k \forall_j \text{row}_j += B_{ik} C_{kj})$$



[Gustavson 1978]

```

for (int i = 0; i < m; i++) {
    for (int pB2 = B2_pos[i]; pB2 < B2_pos[i+1]; pB2++) {
        int k = B2_crd[pB2];

        for (int pC2 = C2_pos[k]; pC2 < C2_pos[k+1]; pC2++) {
            int j = C2_crd[pC2];
            row[j] += B[pB2] * C[pC2];
        }
    }

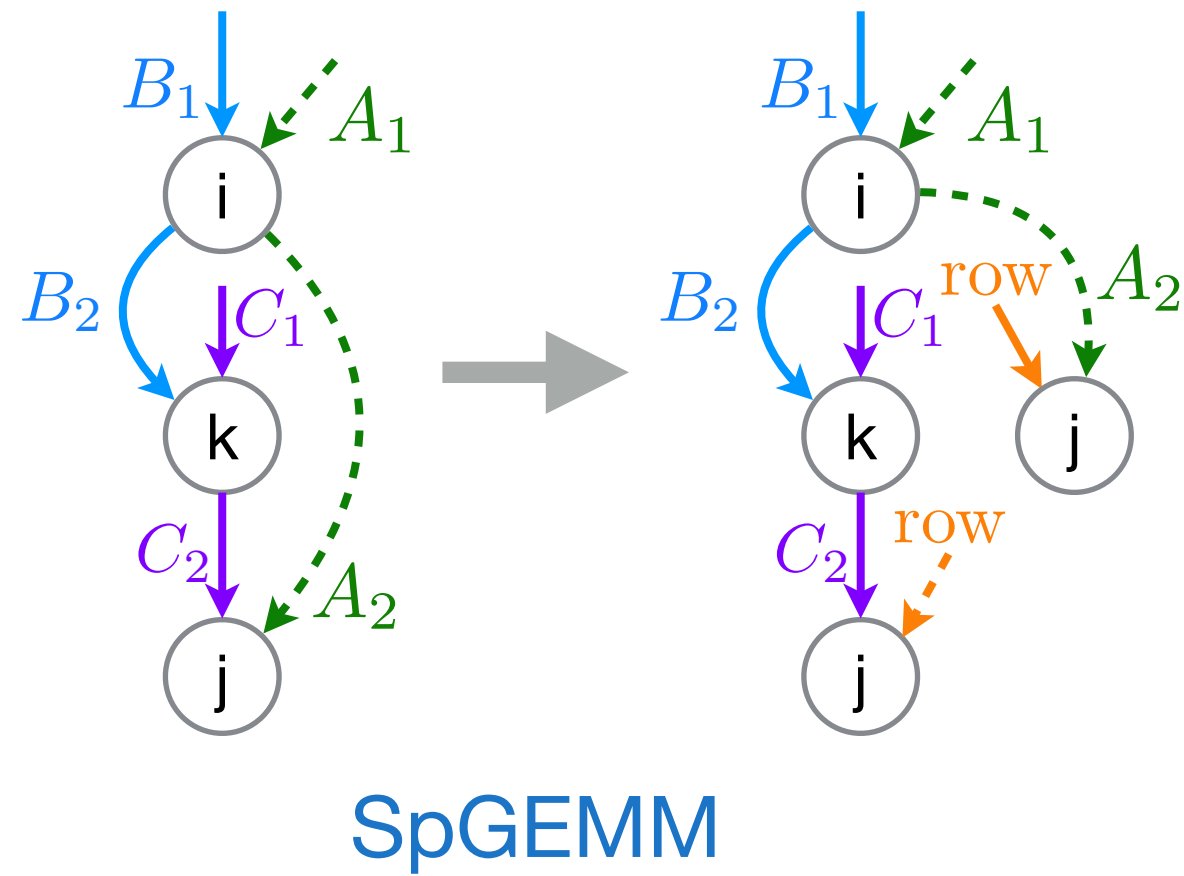
    for (int pA2 = A2_pos[i]; pA2 < A2_pos[i+1]; pA2++) {
        int j = A2_crd[pA2];
        A[pA2] = row[j];
        row[j] = 0.0;
    }
}

```

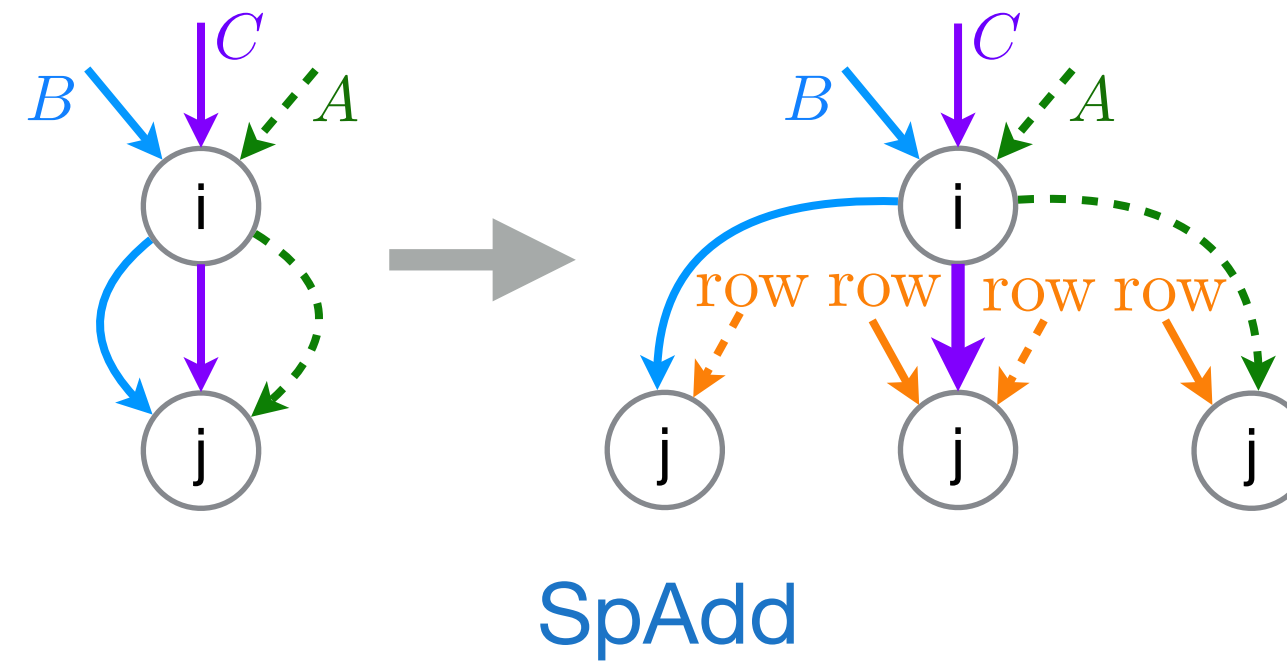


# Other uses of where clauses

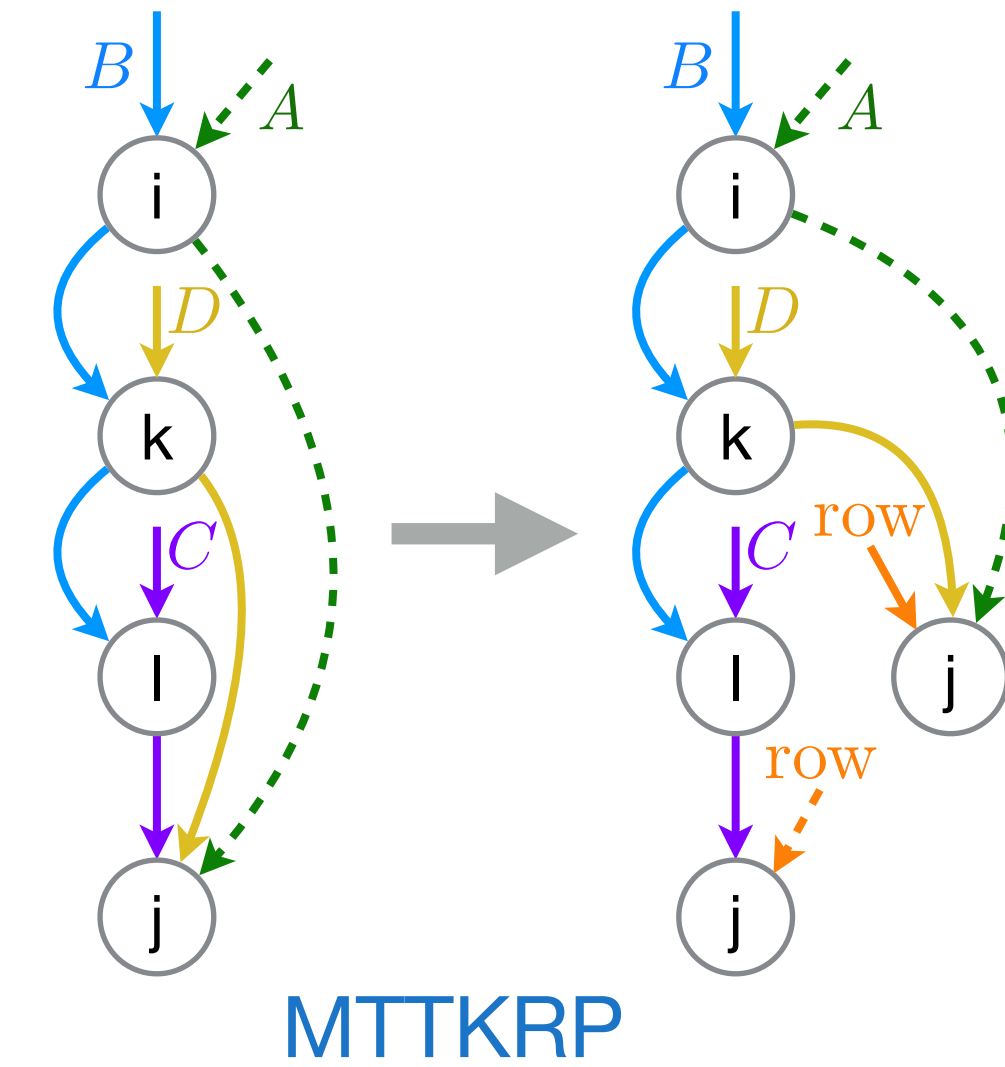
## Scatter into results



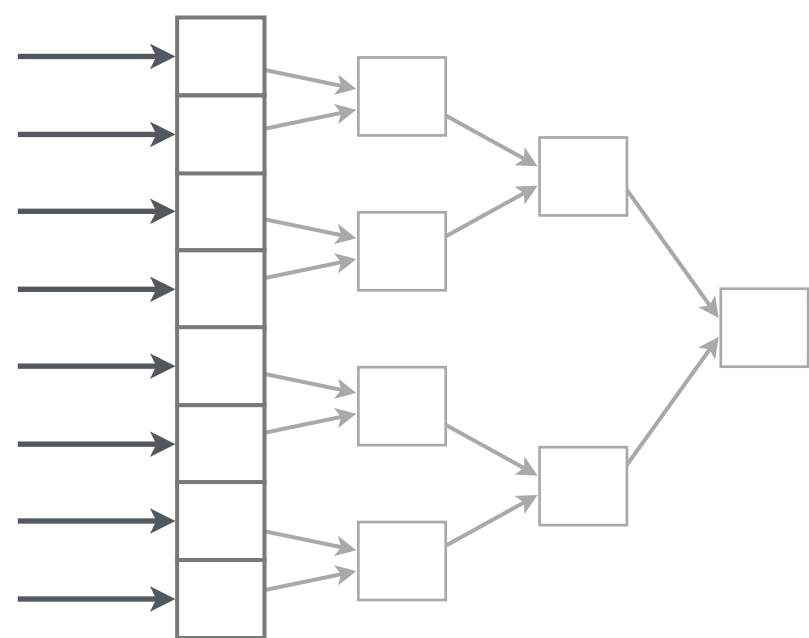
## Simplify merge code



## Loop-invariant code motion



## Prepare reductions



## GPU shared memories



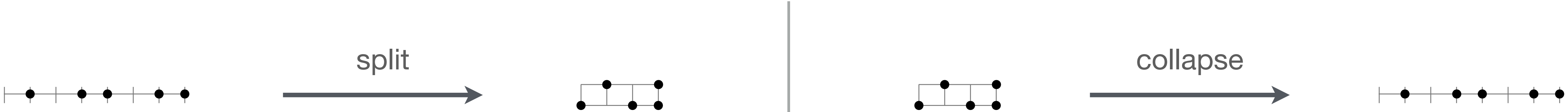
## Mixed precision

```

for (int i = 0; i < m; i++) {
    double tj = 0.0;
    for (int pB2 = B2_pos[i];
         pB2 < B2_pos[i+1];
         pB2++) {
        int j = B2_crd[pB2];
        tj += B[pB2] * c[j];
    }
    a[i] = tj;
}
    
```

Single-precision floating point

# A derived iteration space is one where dimensions have been split or collapsed



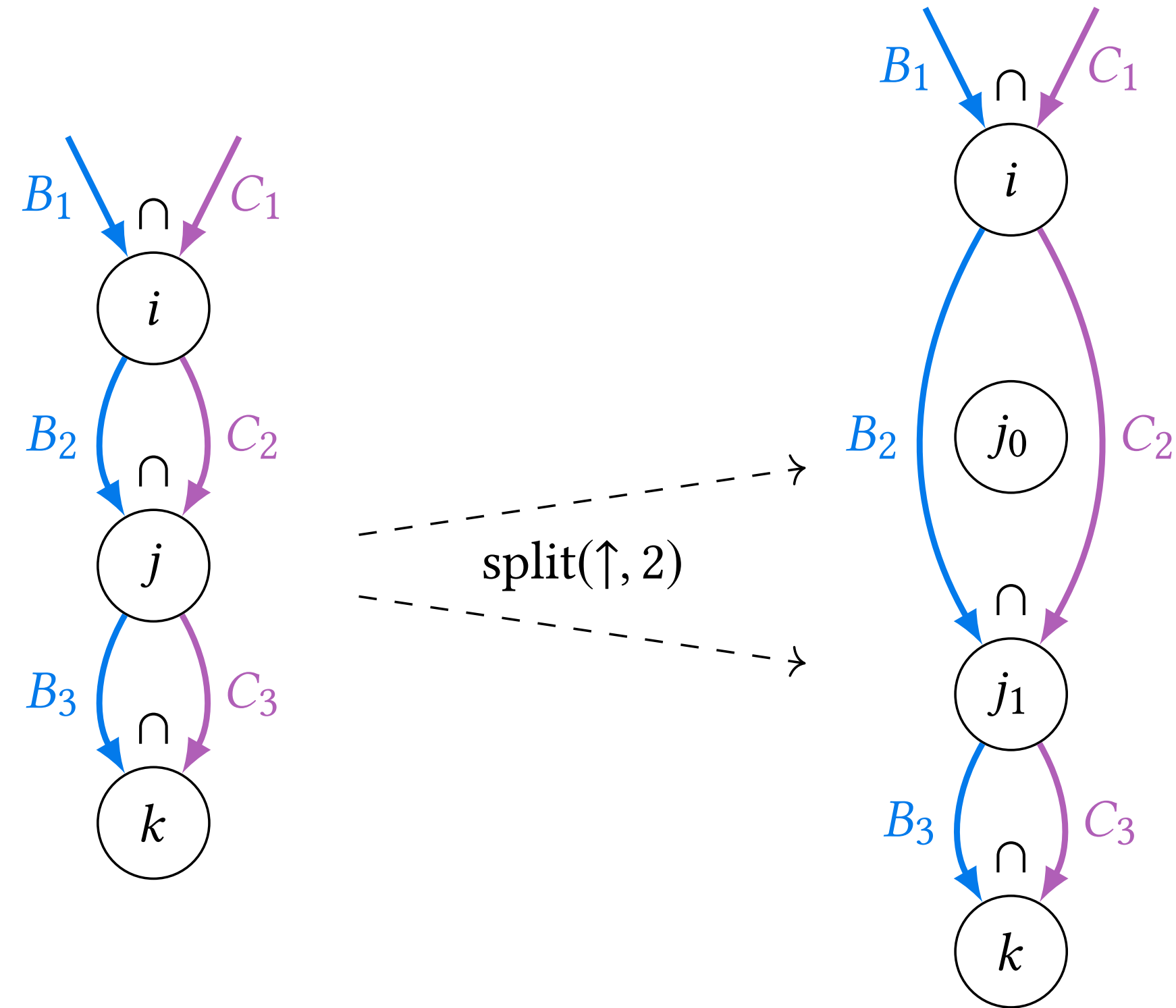
$$\forall_i a_i = b_i + c_i \xrightarrow{\text{split}} \forall_{i_0} \forall_{i_1} a_i = b_i + c_i : i \xrightarrow{\text{split}(\uparrow, 4)} i_0 i_1$$

↑
↑
↑

derived index variables
original index variable  $i$

Cannot simply rewrite access expressions as with dense, so the mapping functions must be stored in the IR, so we can later emit code to remap coordinates.

# The split iteration space function



$$\frac{\forall i \forall j \forall k B_{ijk} \cap C_{ijk}}{}$$

$$i \in B_1 \cap C_1$$

$$j \in B_2 \cap C_2$$

$$k \in B_3 \cap C_3$$

$$\frac{\forall i \forall j_0 \forall j_1 \forall k B_{ijk} \cap C_{ijk} : j}{\text{split}(\uparrow, 2)} \rightarrow j_0 j_1$$

$$i \in B_1 \cap C_1$$

$$j_0 \in [0, 4)$$

$$j_1 \in B_2 \cap C_2$$

$$k \in B_3 \cap C_3$$

# The split iteration space function comes in several variants

`split(direction, stride, [tensor operand])`

The function signature is shown with blue underlines under 'direction', 'stride', and '[tensor operand]'. Two blue arrows point from the explanatory text below to these underlined parts: one from the left text to 'direction' and one from the right text to '[tensor operand]'.

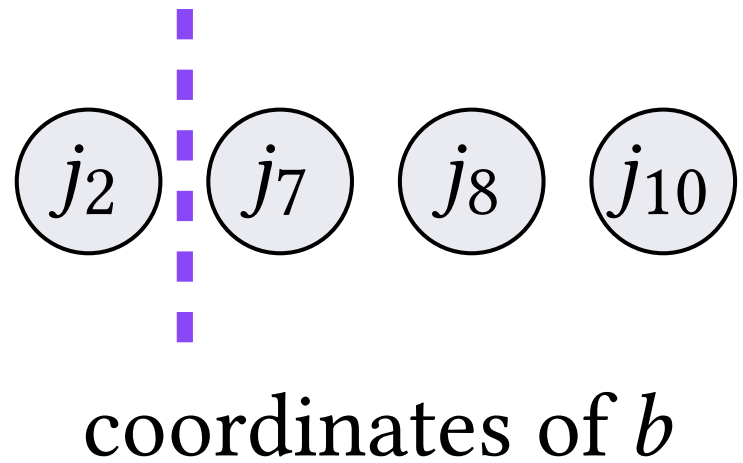
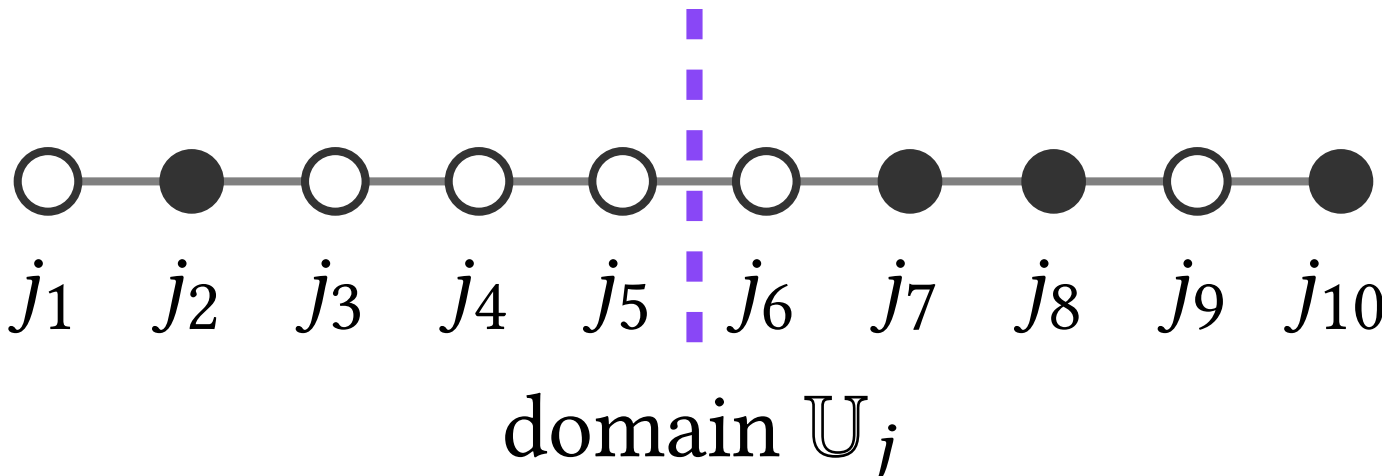
Split off a loop with stride iterations as the:

- outer loop (direction =  $\uparrow$ ), or
- inner loop (direction =  $\downarrow$ )

Optional tensor operand whose nonzero coordinates the split applies to. If not given, the split applies to the universe of coordinates of the split index variable.

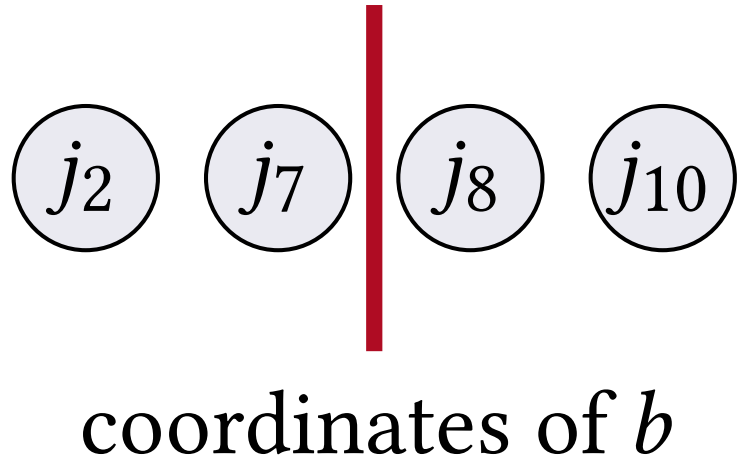
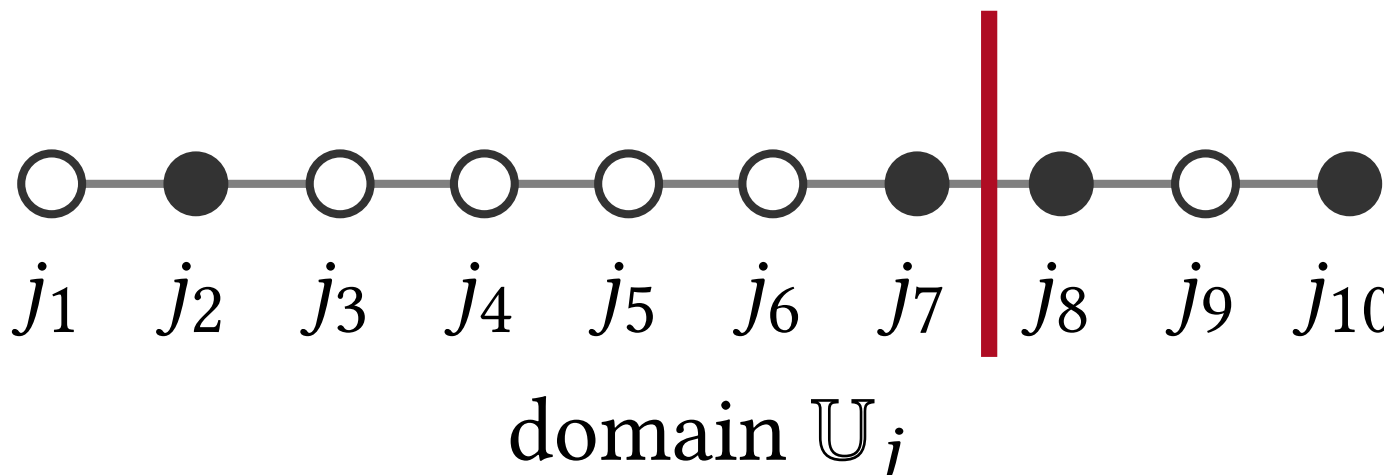
# The split iteration space function can split with respect to the universe of coordinates or the coordinates of one operand

Universe split



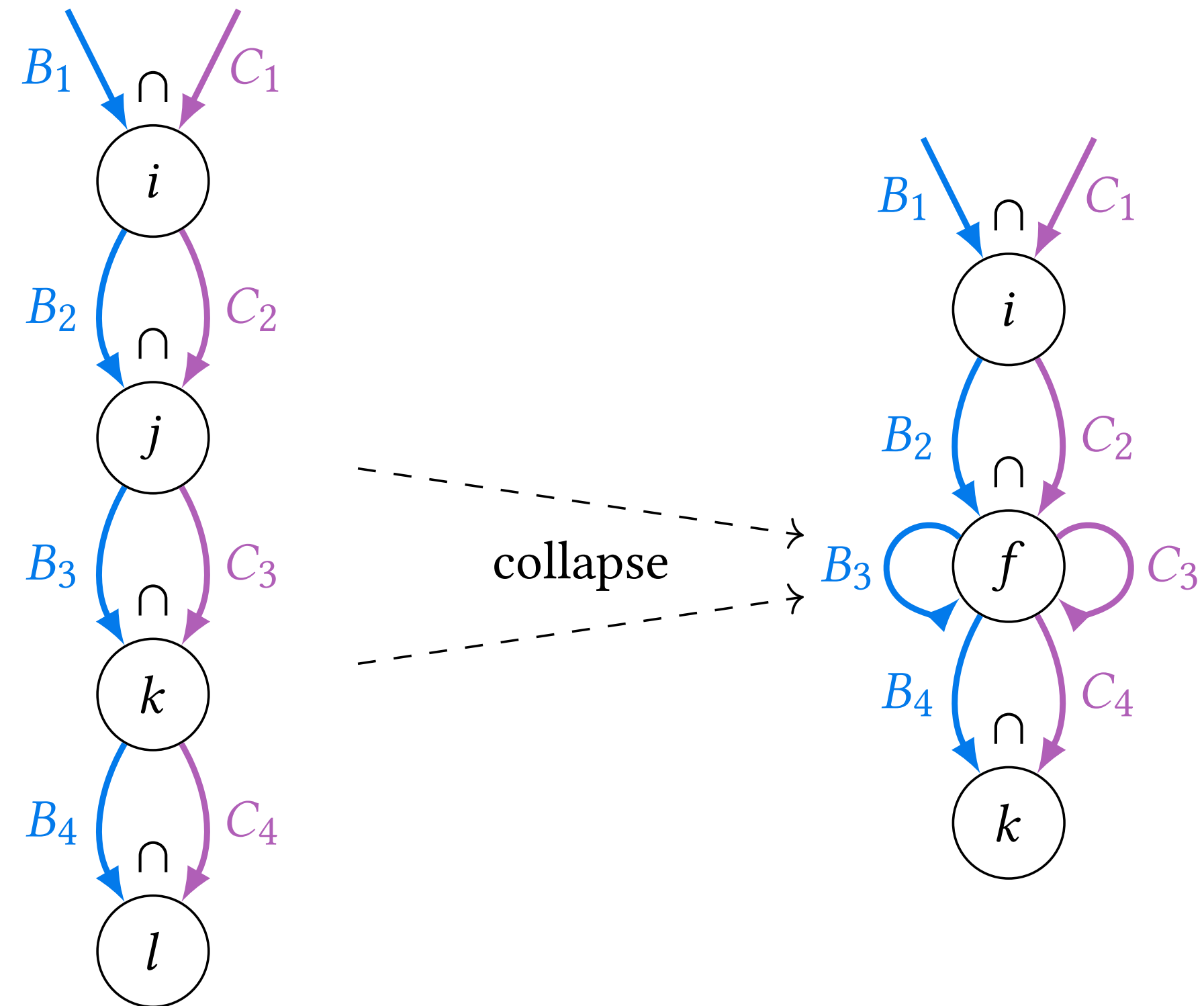
even split in coordinate universe, but uneven split in  $b$ 's nonzero coordinates

Coordinate tree split



uneven split in coordinate universe, but even split in  $b$ 's nonzero coordinates

# The collapse iteration space function



$$\frac{\forall i \forall j \forall k \forall l B_{ijkl} \cap C_{ijkl}}{}$$

- $i \in B_1 \cap C_1$
- $j \in B_2 \cap C_2$
- $k \in B_3 \cap C_3$
- $l \in B_4 \cap C_4$

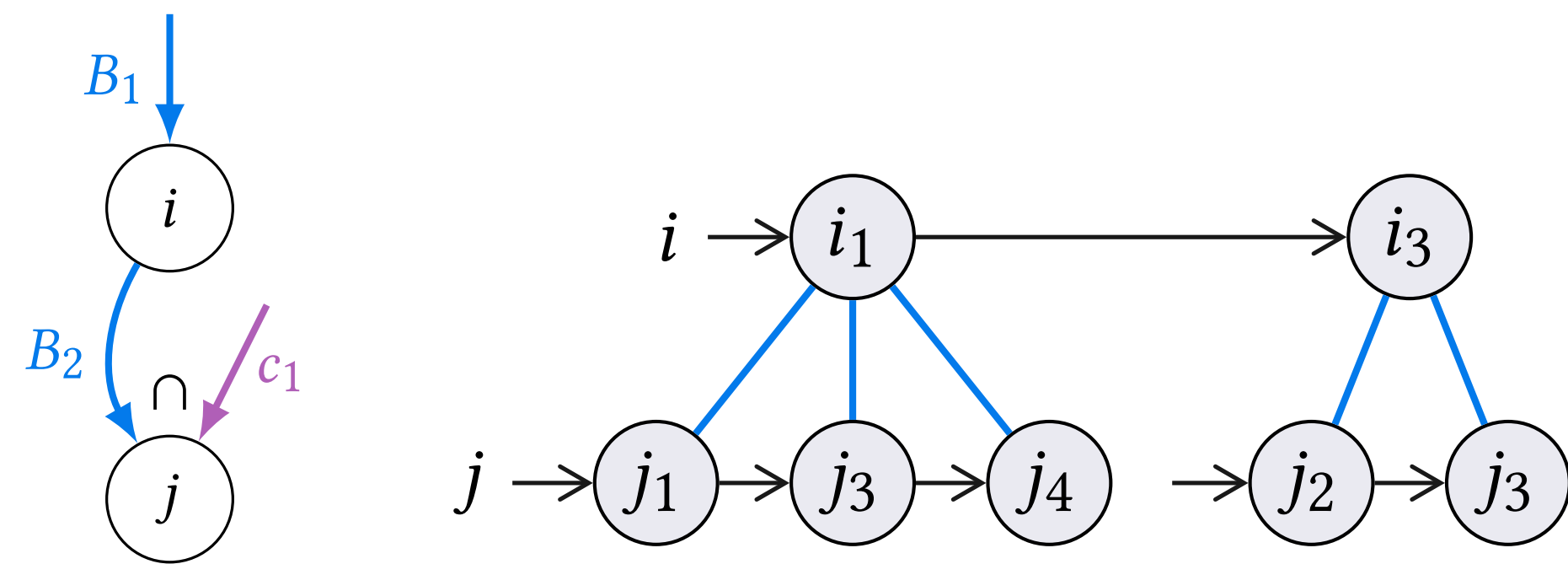
$$\frac{\forall i \forall f \forall l B_{ijkl} \cap C_{ijkl} : jk \xrightarrow{\text{collapse}} f}{}$$

- $i \in B_1 \cap C_1$
- $f \in (B_2 \times B_3) \cap (C_2 \times C_3)$
- $k \in B_4 \cap C_4$

Iterate over Cartesian combination of coordinates in  $i$  and  $j$ .

# The collapse function leads to bottom-up iteration

## Pre-split top-down iteration

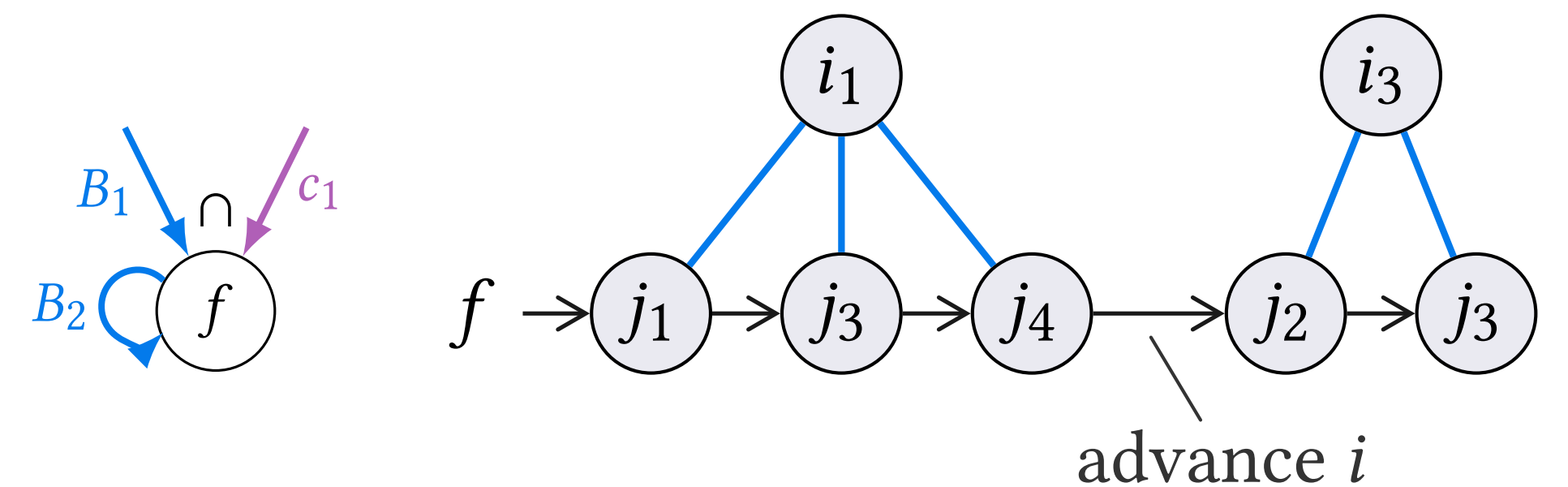


```

for (int i = 0; i < m; i++) {
  for (int jB = B2_pos[i];
       jB < B2_pos[i+1]; jB++) {
    int j = B2_crd[jB];
    a[i] += B[jB] * c[j];
  }
}

```

## Post-split bottom-up iteration



```

for (int f = 0, i = 0;
     f < B2_pos[m]; f++) {
  if (f >= B2_pos[m]) break;

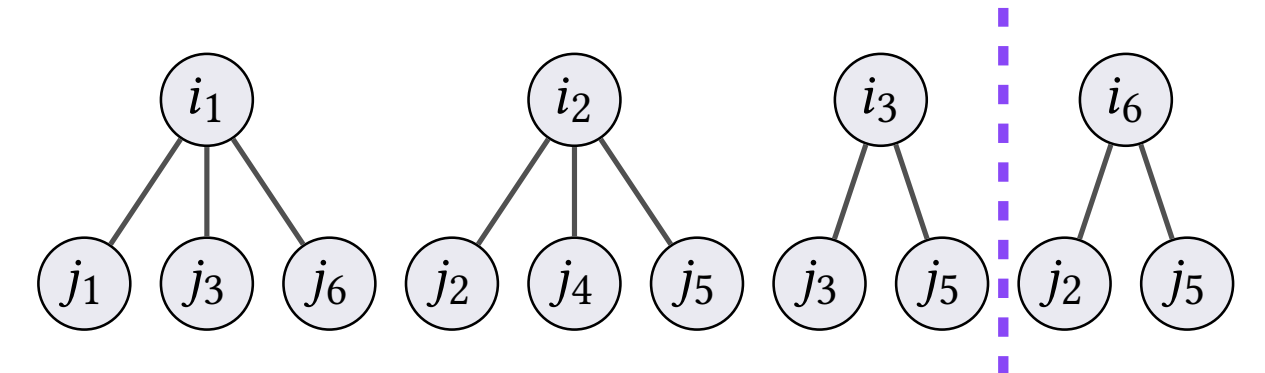
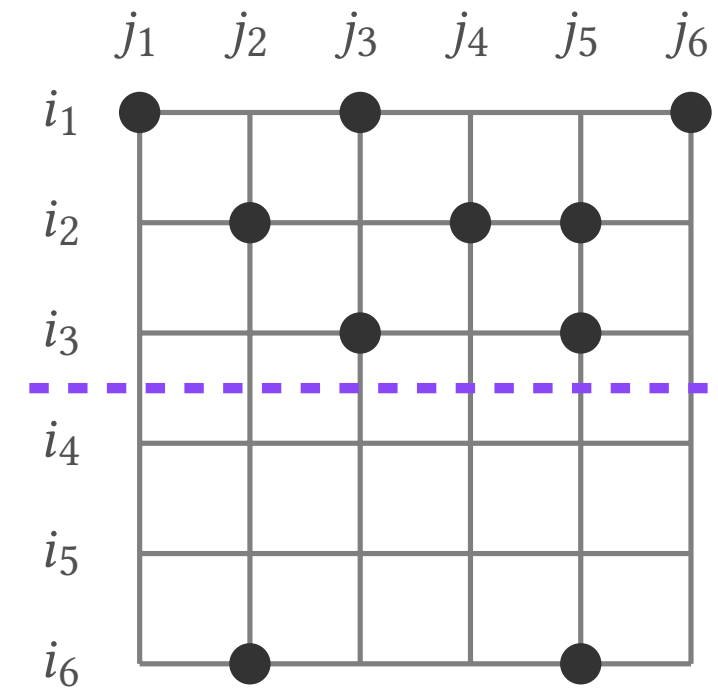
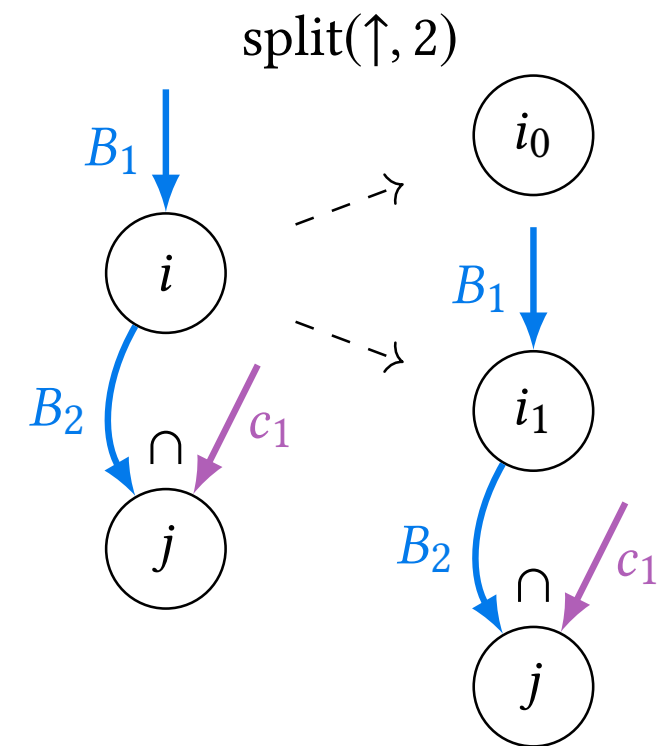
  int j = B2_crd[f];
  while (f == B2_pos[i+1]) i++;

  a[i] += B[f] * c[j];
}

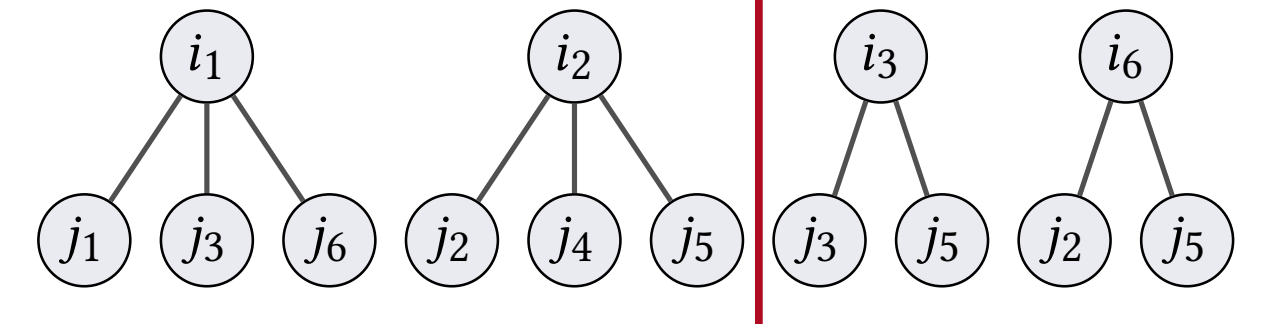
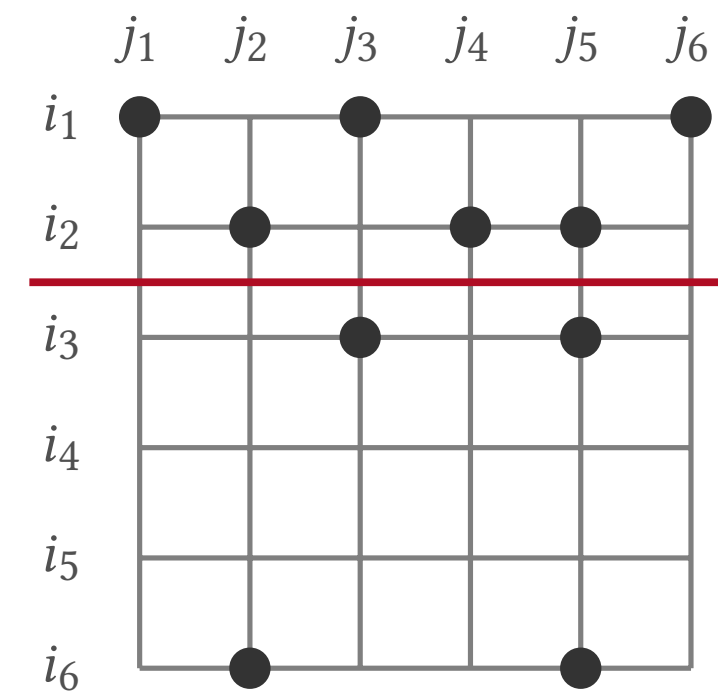
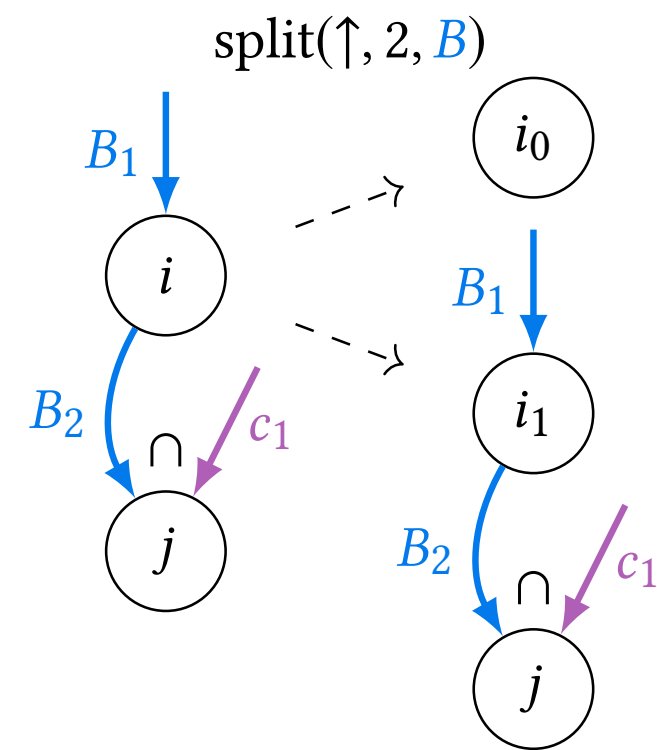
```

# Two-dimensional tiling examples

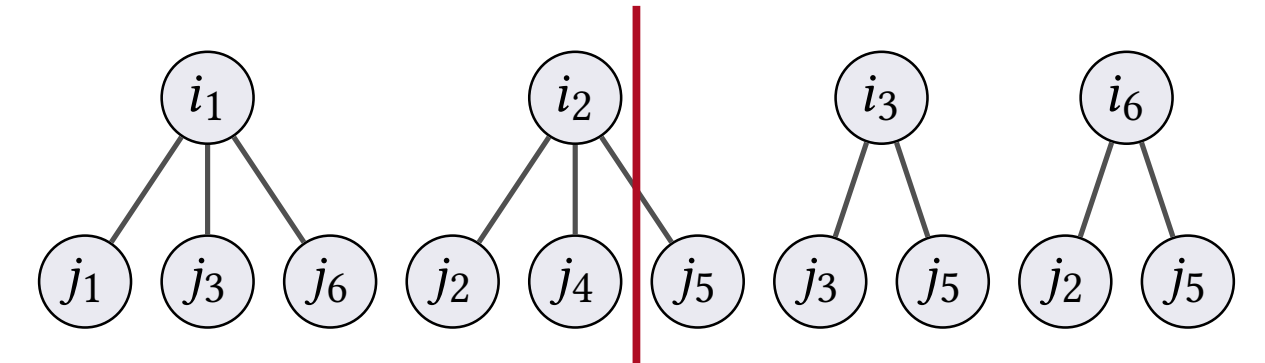
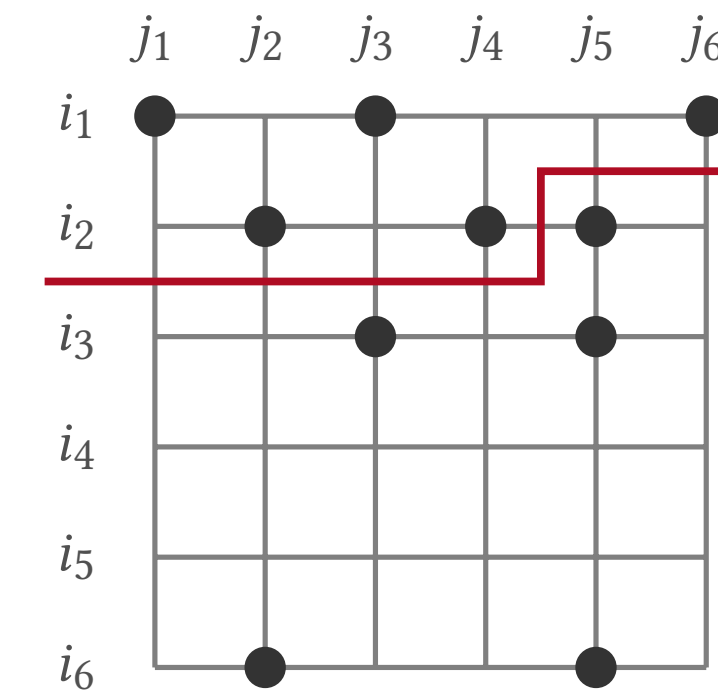
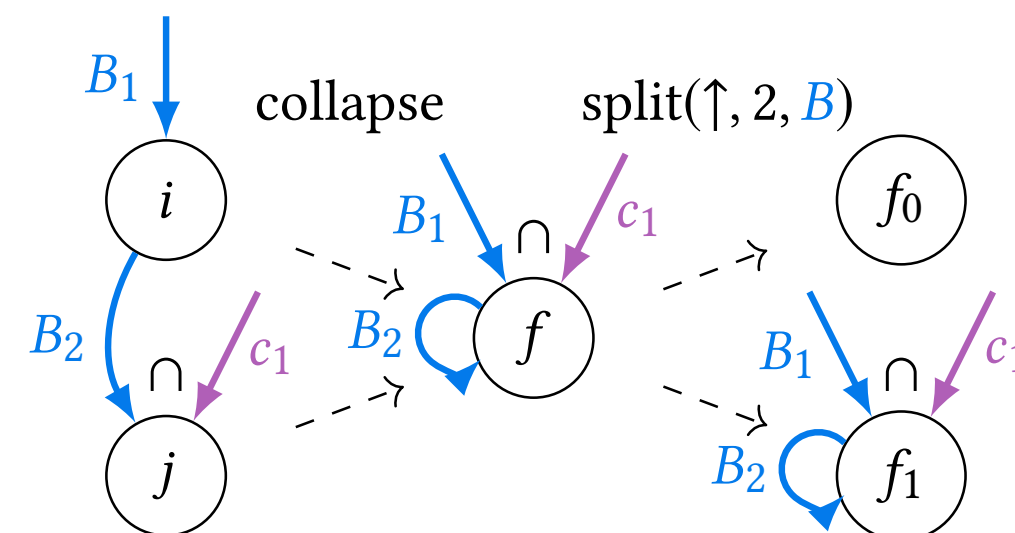
Universe split



Coordinate tree split

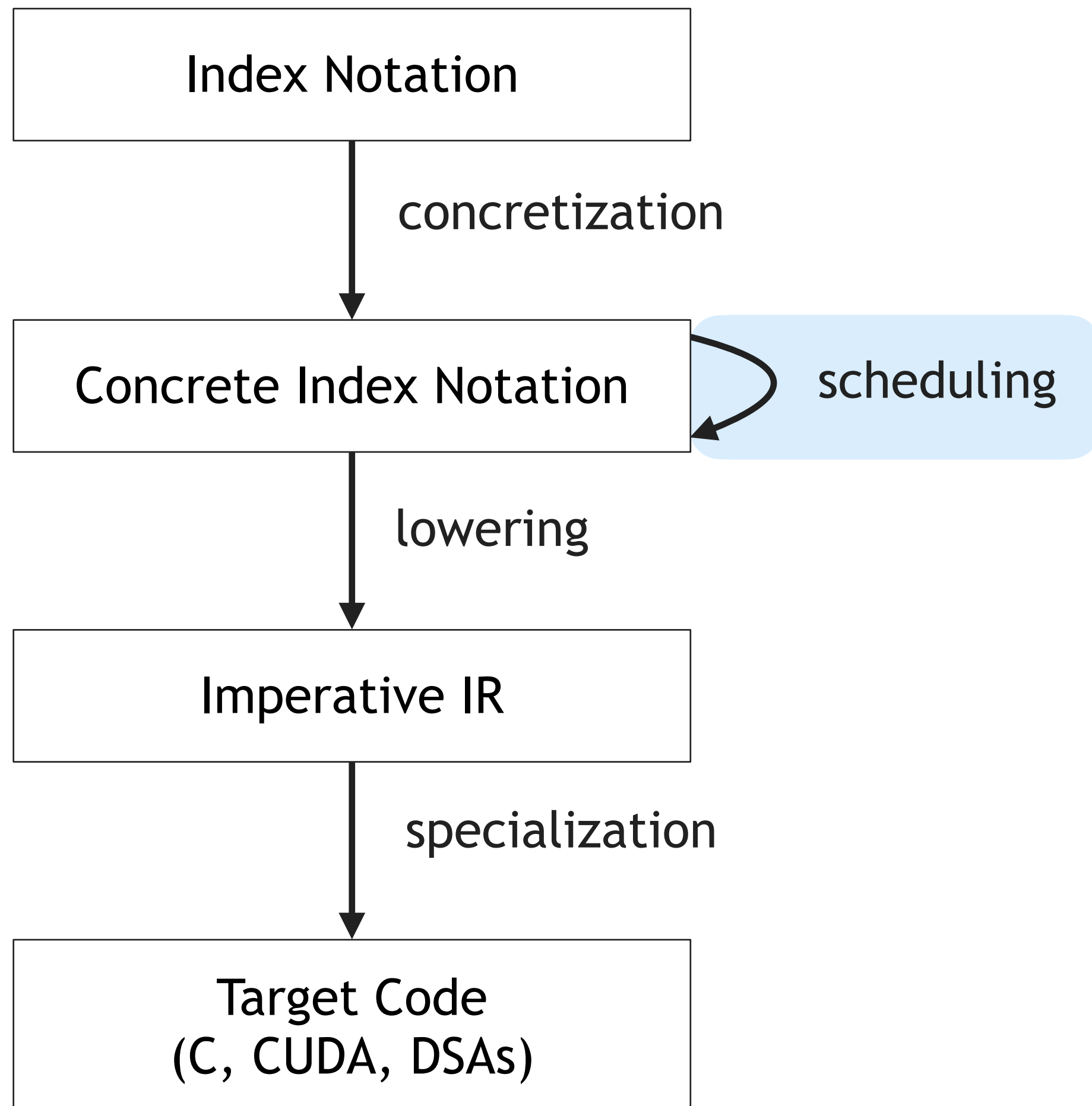


Collapse+coordinate tree split





# A sparse (tensor algebra) scheduling language



- **reorder(i, j)** interchanges loops i and j
- **split(i, i<sub>1</sub>, i<sub>2</sub>, d, s, t)** strip-mines i into two loops i<sub>1</sub> and i<sub>2</sub>, where i<sub>1</sub> or i<sub>2</sub> is of size s depending on the direction d. The tensor t is optional and, if given, means the loop is strip-mined w.r.t. its nonzeros.
- **collapse(i, j, f)** collapses loops i and j into a new loop f, which iterates over their Cartesian combination.
- **precompute(S, e, t, I)** precomputes expression e in index statement S before the loops I and stores the results in tensor t.
- **unroll, parallelize, vectorize, ...**

# Lowering algorithm for code generation

```
function LOWER(assignment statement  $S_{\text{assignment}}$ )  
    Emit compute code  
end function
```

```
function LOWER(while statement  $S_{\text{while}}$ )  
    LOWER(producer statement of  $S_{\text{while}}$ )  
    LOWER(consumer statement of  $S_{\text{while}}$ )  
end function
```

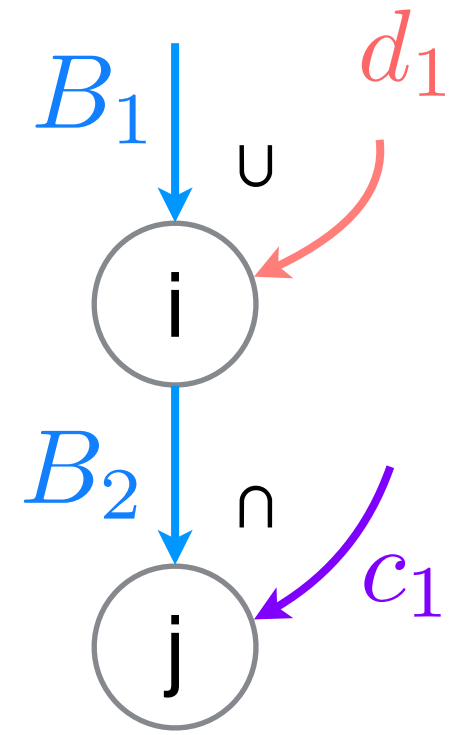
```
function LOWER(sequence statement  $S_{\text{sequence}}$ )  
    LOWER(definition statement of  $S_{\text{sequence}}$ )  
    LOWER(mutation statement of  $S_{\text{sequence}}$ )  
end function
```

```
function LOWER(multi statement  $S_{\text{multi}}$ )  
    LOWER(left statement of  $S_{\text{multi}}$ )  
    LOWER(right statement of  $S_{\text{multi}}$ )  
end function
```

```
function LOWER(forall statement  $S_{\text{forall}}$  of index variable  $i$ )  
    let  $\mathcal{L}$  be an iteration lattice constructed from  $S_{\text{forall}}$   
    Emit initialize iterators  
    for each lattice point  $\mathcal{L}_p$  in  $\mathcal{L}$  do  
        Emit loop header  
        Emit access iterators  
        Emit map candidate coordinates to the original space  
        Emit resolve the coordinate of  $i$   
        Emit map resolved coordinate to each derived space  
        Emit locate from locators  
        for each lattice point  $\mathcal{L}_q < \mathcal{L}_p$  in  $\mathcal{L}$  do  
            Emit conditional header  
            let  $S_{\text{simplified}}$  be a statement constructed from  
                the body of  $S_{\text{forall}}$  by removing operands  
                that have run out of values in  $\mathcal{L}_q$   
            LOWER( $S_{\text{simplified}}$ )  
            Emit assembly code  
            Emit conditional footer  
        end for  
        Emit advance iterators  
        Emit loop footer  
    end for  
end function
```

# Recursive lowering of concrete index notation

$$a_i = B_{ij} c_j + d_i$$

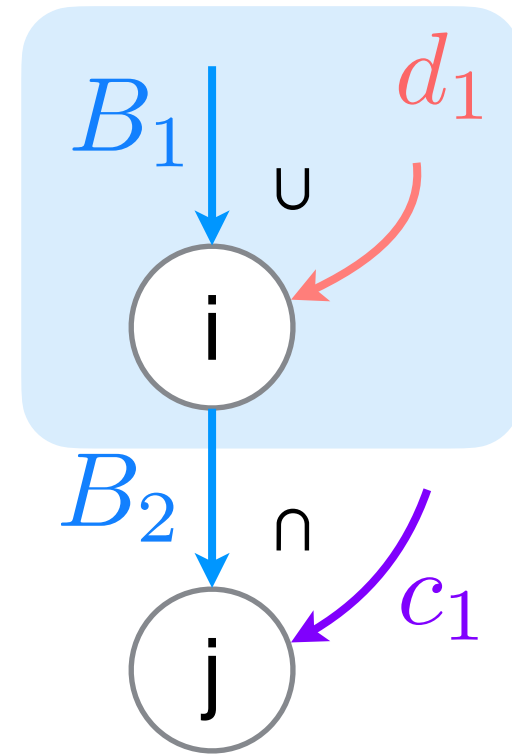


---

```
function LOWER(forall statement  $S_{\text{forall}}$  of index variable  $i$ )  
  let  $\mathcal{L}$  be an iteration lattice constructed from  $S_{\text{forall}}$   
  Emit initialize iterators  
  for each lattice point  $\mathcal{L}_p$  in  $\mathcal{L}$  do  
    Emit loop header  
    Emit access iterators  
    Emit resolve the coordinate of  $i$   
    Emit locate from locators  
    for each lattice point  $\mathcal{L}_q < \mathcal{L}_p$  in  $\mathcal{L}$  do  
      Emit conditional header  
      let  $S_{\text{simplified}}$  be a statement constructed from  
        the body of  $S_{\text{forall}}$  by removing operands  
        that have run out of values in  $\mathcal{L}_q$   
      LOWER( $S_{\text{simplified}}$ )  
      Emit conditional footer  
    end for  
    Emit advance iterators  
    Emit loop footer  
  end for  
end function
```

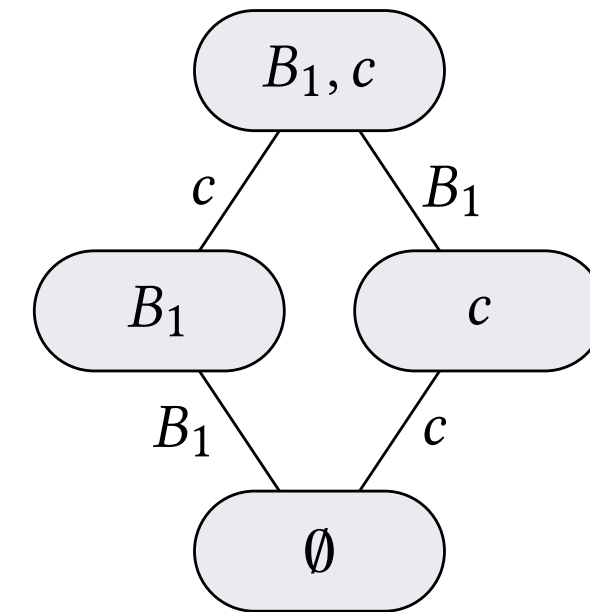
# Recursive lowering of concrete index notation

$$a_i = B_{ij} c_j + d_i$$



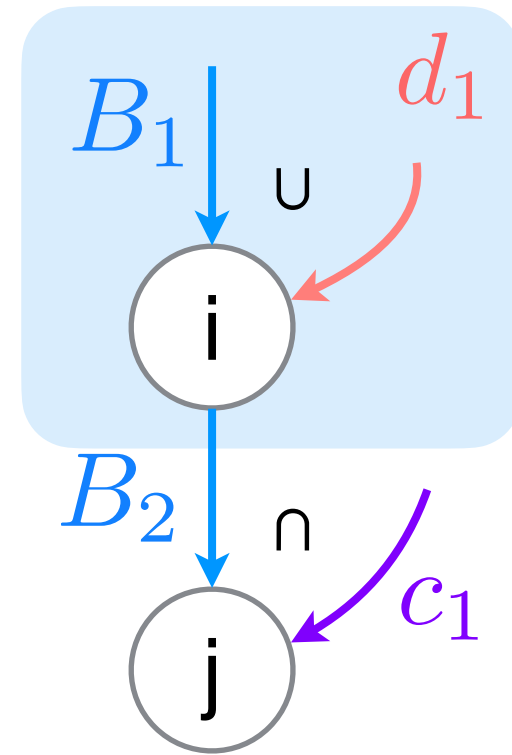

---

**function** LOWER(forall statement  $S_{\text{forall}}$  of index variable  $i$ )  
**let**  $\mathcal{L}$  be an iteration lattice constructed from  $S_{\text{forall}}$   
**Emit** initialize iterators  
**for** each lattice point  $\mathcal{L}_p$  in  $\mathcal{L}$  **do**  
    **Emit** loop header  
    **Emit** access iterators  
    **Emit** resolve the coordinate of  $i$   
    **Emit** locate from locators  
    **for** each lattice point  $\mathcal{L}_q < \mathcal{L}_p$  in  $\mathcal{L}$  **do**  
        **Emit** conditional header  
        **let**  $S_{\text{simplified}}$  be a statement constructed from  
            the body of  $S_{\text{forall}}$  by removing operands  
            that have run out of values in  $\mathcal{L}_q$   
        LOWER( $S_{\text{simplified}}$ )  
        **Emit** conditional footer  
    **end for**  
    **Emit** advance iterators  
    **Emit** loop footer  
**end for**  
**end function**

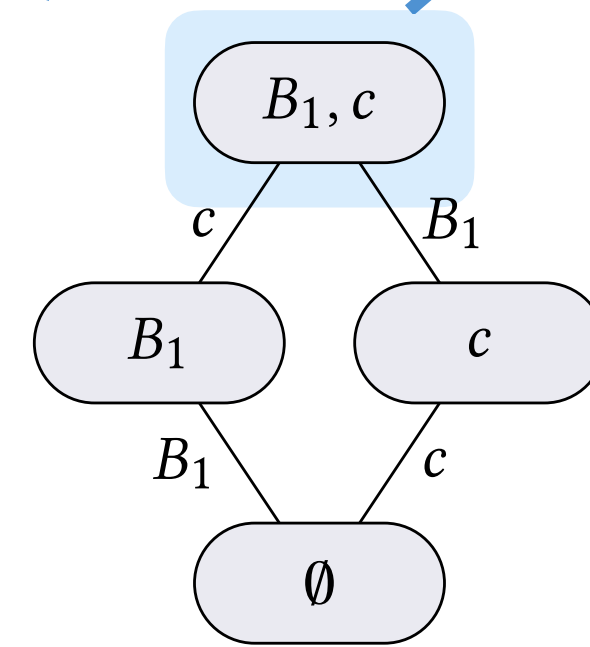


# Recursive lowering of concrete index notation

$$a_i = B_{ij} c_j + d_i$$



```
int iB = B1_pos[0];
int id = d1_pos[0];
while (iB < B1_pos[1] && id < d1_pos) {
    int iB = B1_crd[iB];
    int id = d1_crd[id];
    int i = min(iB, id);
}
```

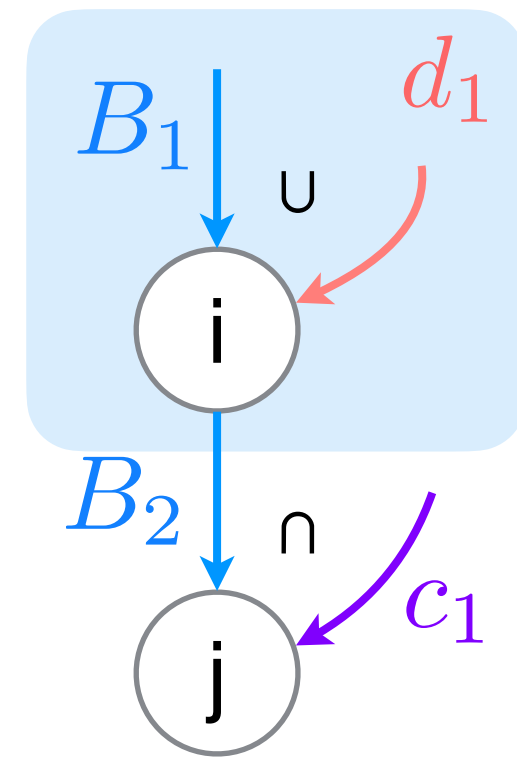


```
if (iB == i) iB++;
if (id == i) id++;
}
```

```
function LOWER(forall statement  $S_{forall}$  of index variable  $i$ )
  let  $\mathcal{L}$  be an iteration lattice constructed from  $S_{forall}$ 
  Emit initialize iterators
  for each lattice point  $\mathcal{L}_p$  in  $\mathcal{L}$  do
    Emit loop header
    Emit access iterators
    Emit resolve the coordinate of  $i$ 
    Emit locate from locators
    for each lattice point  $\mathcal{L}_q < \mathcal{L}_p$  in  $\mathcal{L}$  do
      Emit conditional header
      let  $S_{simplified}$  be a statement constructed from
        the body of  $S_{forall}$  by removing operands
        that have run out of values in  $\mathcal{L}_q$ 
      LOWER( $S_{simplified}$ )
      Emit conditional footer
    end for
    Emit advance iterators
    Emit loop footer
  end for
end function
```

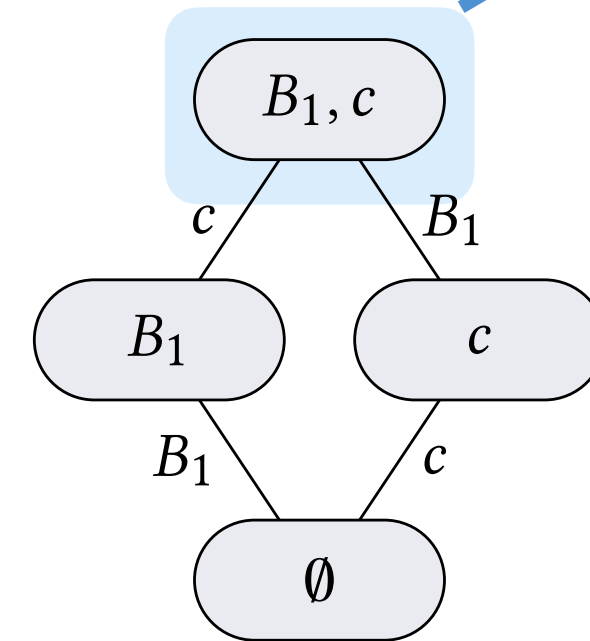
# Recursive lowering of concrete index notation

$$a_i = B_{ij} c_j + d_i$$



```

function LOWER(forall statement  $S_{forall}$  of index variable  $i$ )
  let  $\mathcal{L}$  be an iteration lattice constructed from  $S_{forall}$ 
  Emit initialize iterators
  for each lattice point  $\mathcal{L}_p$  in  $\mathcal{L}$  do
    Emit loop header
    Emit access iterators
    Emit resolve the coordinate of  $i$ 
    Emit locate from locators
    for each lattice point  $\mathcal{L}_q < \mathcal{L}_p$  in  $\mathcal{L}$  do
      Emit conditional header
      let  $S_{simplified}$  be a statement constructed from
        the body of  $S_{forall}$  by removing operands
        that have run out of values in  $\mathcal{L}_q$ 
      LOWER( $S_{simplified}$ )
      Emit conditional footer
    end for
    Emit advance iterators
    Emit loop footer
  end for
end function
  
```



```

int iB = B1_pos[0];
int id = d1_pos[0];
while (iB < B1_pos[1] && id < d1_pos) {
  int iB = B1_crd[iB];
  int id = d1_crd[id];
  int i = min(iB, id);
  if (iB == i && id == i) {
  
```

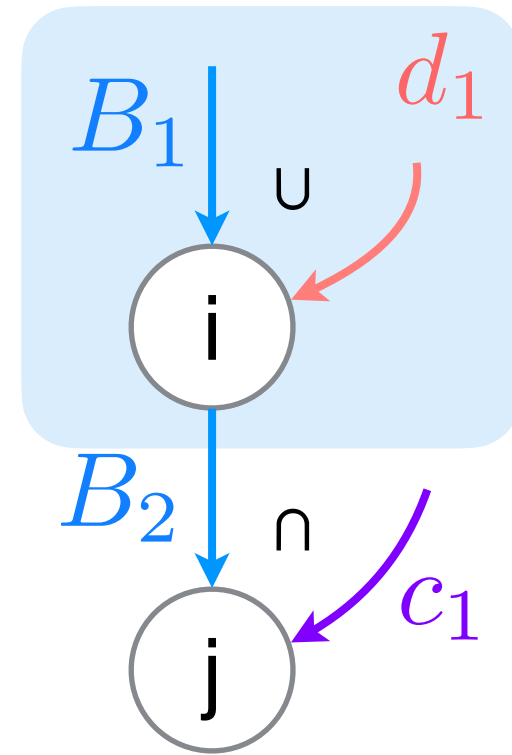
}

```

if (iB == i) iB++;
if (id == i) id++;
}
  
```

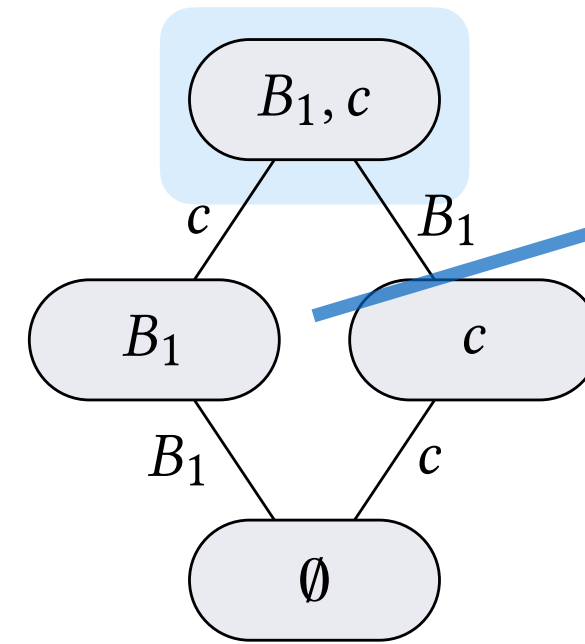
# Recursive lowering of concrete index notation

$$a_i = B_{ij} c_j + d_i$$



```

function LOWER(forall statement  $S_{forall}$  of index variable  $i$ )
  let  $\mathcal{L}$  be an iteration lattice constructed from  $S_{forall}$ 
  Emit initialize iterators
  for each lattice point  $\mathcal{L}_p$  in  $\mathcal{L}$  do
    Emit loop header
    Emit access iterators
    Emit resolve the coordinate of  $i$ 
    Emit locate from locators
    for each lattice point  $\mathcal{L}_q < \mathcal{L}_p$  in  $\mathcal{L}$  do
      Emit conditional header
      let  $S_{simplified}$  be a statement constructed from
        the body of  $S_{forall}$  by removing operands
        that have run out of values in  $\mathcal{L}_q$ 
      LOWER( $S_{simplified}$ )
      Emit conditional footer
    end for
    Emit advance iterators
    Emit loop footer
  end for
end function
  
```

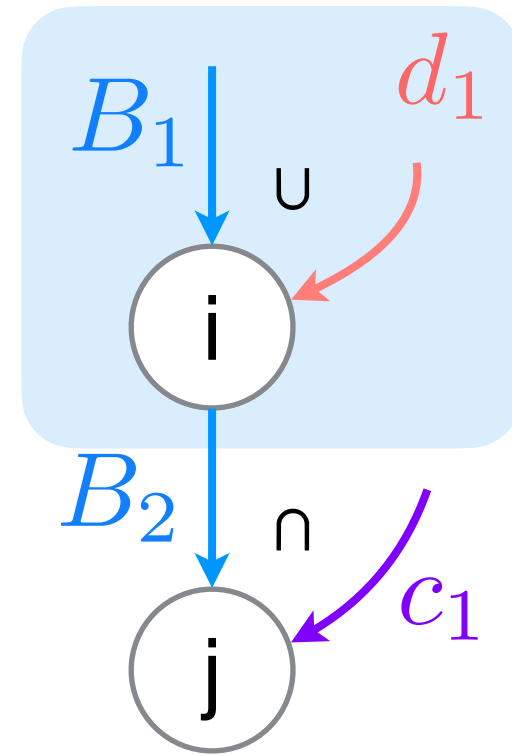


```

int iB = B1_pos[0];
int id = d1_pos[0];
while (iB < B1_pos[1] && id < d1_pos) {
  int iB = B1_crd[iB];
  int id = d1_crd[id];
  int i = min(iB, id);
  if (iB == i && id == i) {
    }
  else if (iB == i) {
    if (iB == i) iB++;
    if (id == i) id++;
  }
}
  
```

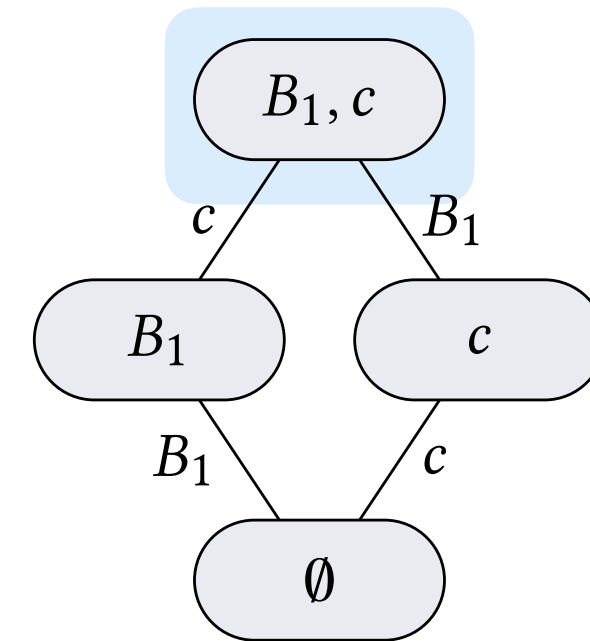
# Recursive lowering of concrete index notation

$$a_i = B_{ij} c_j + d_i$$



```

function LOWER(forall statement  $S_{forall}$  of index variable  $i$ )
  let  $\mathcal{L}$  be an iteration lattice constructed from  $S_{forall}$ 
  Emit initialize iterators
  for each lattice point  $\mathcal{L}_p$  in  $\mathcal{L}$  do
    Emit loop header
    Emit access iterators
    Emit resolve the coordinate of  $i$ 
    Emit locate from locators
    for each lattice point  $\mathcal{L}_q < \mathcal{L}_p$  in  $\mathcal{L}$  do
      Emit conditional header
      let  $S_{simplified}$  be a statement constructed from
        the body of  $S_{forall}$  by removing operands
        that have run out of values in  $\mathcal{L}_q$ 
      LOWER( $S_{simplified}$ )
      Emit conditional footer
    end for
    Emit advance iterators
    Emit loop footer
  end for
end function
  
```



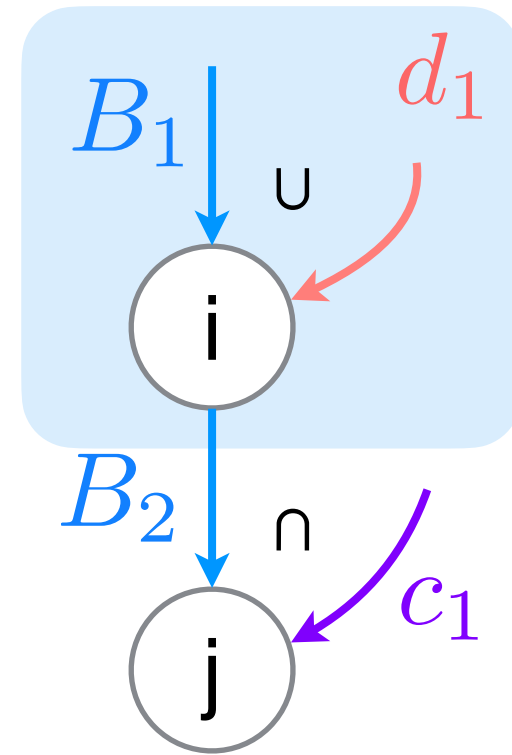
```

int iB = B1_pos[0];
int id = d1_pos[0];
while (iB < B1_pos[1] && id < d1_pos) {
  int iB = B1_crd[iB];
  int id = d1_crd[id];
  int i = min(iB, id);
  if (iB == i && id == i) {
    }
  else if (iB == i) {
    }
  else {
    a[i] = d[id];
  }
  if (iB == i) iB++;
  if (id == i) id++;
}
  
```



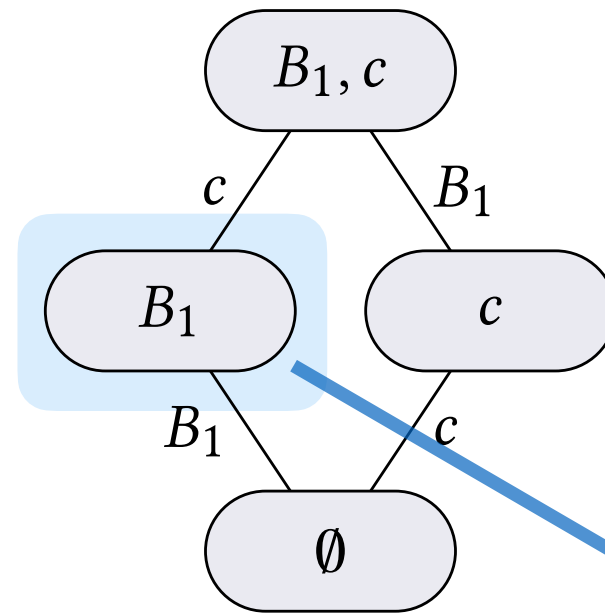
# Recursive lowering of concrete index notation

$$a_i = B_{ij} c_j + d_i$$



```

function LOWER(forall statement  $S_{forall}$  of index variable  $i$ )
  let  $\mathcal{L}$  be an iteration lattice constructed from  $S_{forall}$ 
  Emit initialize iterators
  for each lattice point  $\mathcal{L}_p$  in  $\mathcal{L}$  do
    Emit loop header
    Emit access iterators
    Emit resolve the coordinate of  $i$ 
    Emit locate from locators
    for each lattice point  $\mathcal{L}_q < \mathcal{L}_p$  in  $\mathcal{L}$  do
      Emit conditional header
      let  $S_{simplified}$  be a statement constructed from
        the body of  $S_{forall}$  by removing operands
        that have run out of values in  $\mathcal{L}_q$ 
      LOWER( $S_{simplified}$ )
      Emit conditional footer
    end for
    Emit advance iterators
    Emit loop footer
  end for
end function
  
```



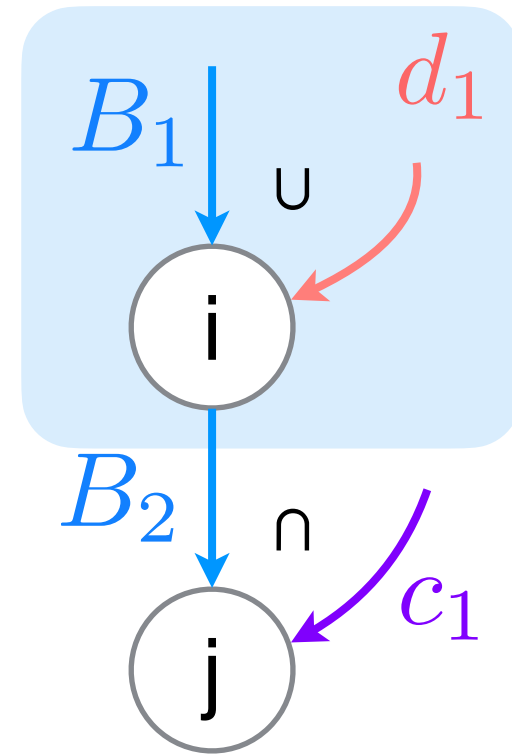
```

int iB = B1_pos[0];
int id = d1_pos[0];
while (iB < B1_pos[1] && id < d1_pos) {
  int iB = B1_crd[iB];
  int id = d1_crd[id];
  int i = min(iB, id);
  if (iB == i && id == i) {
  }
  else if (iB == i) {
  }
  else {
    a[i] = d[id];
  }
  if (iB == i) iB++;
  if (id == i) id++;
}

while (iB < B1_pos[1]) {
  int i = B1_crd[iB];
  for (int jB = B2_pos[iB]; jB < B2_pos[iB+1]; jB++) {
    int j = B2_crd[jB];
    a[i] += B[jB] * c[j];
  }
  iB++;
}
  
```

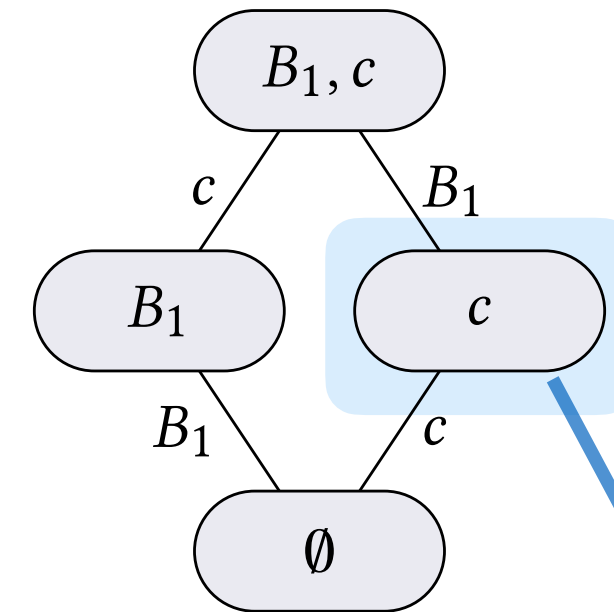
# Recursive lowering of concrete index notation

$$a_i = B_{ij} c_j + d_i$$



```

function LOWER(forall statement  $S_{forall}$  of index variable  $i$ )
  let  $\mathcal{L}$  be an iteration lattice constructed from  $S_{forall}$ 
  Emit initialize iterators
  for each lattice point  $\mathcal{L}_p$  in  $\mathcal{L}$  do
    Emit loop header
    Emit access iterators
    Emit resolve the coordinate of  $i$ 
    Emit locate from locators
    for each lattice point  $\mathcal{L}_q < \mathcal{L}_p$  in  $\mathcal{L}$  do
      Emit conditional header
      let  $S_{simplified}$  be a statement constructed from
        the body of  $S_{forall}$  by removing operands
        that have run out of values in  $\mathcal{L}_q$ 
      LOWER( $S_{simplified}$ )
      Emit conditional footer
    end for
    Emit advance iterators
    Emit loop footer
  end for
end function
  
```



```

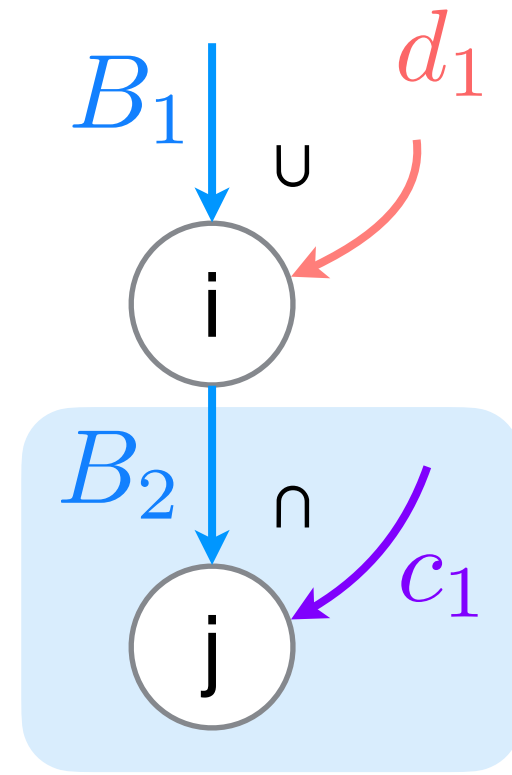
int iB = B1_pos[0];
int id = d1_pos[0];
while (iB < B1_pos[1] && id < d1_pos) {
  int iB = B1_crd[iB];
  int id = d1_crd[id];
  int i = min(iB, id);
  if (iB == i && id == i) {
  }
  else if (iB == i) {
  }
  else {
    a[i] = d[id];
  }
  if (iB == i) iB++;
  if (id == i) id++;
}

while (iB < B1_pos[1]) {
  int i = B1_crd[iB];
  for (int jB = B2_pos[iB]; jB < B2_pos[iB+1]; jB++) {
    int j = B2_crd[jB];
    a[i] += B[jB] * c[j];
  }
  iB++;
}

while (id < d1_pos) {
  int i = d1_crd[id];
  a[i] = d[id];
  id++;
}
  
```

# Recursive lowering of concrete index notation

$$a_i = B_{ij} c_j + d_i$$



**function** LOWER(forall statement  $S_{\text{forall}}$  of index variable  $i$ )

**let**  $\mathcal{L}$  be an iteration lattice constructed from  $S_{\text{forall}}$

**Emit** initialize iterators

**for** each lattice point  $\mathcal{L}_p$  in  $\mathcal{L}$  **do**

**Emit** loop header

**Emit** access iterators

**Emit** resolve the coordinate of  $i$

**Emit** locate from locators

**for** each lattice point  $\mathcal{L}_q < \mathcal{L}_p$  in  $\mathcal{L}$  **do**

**Emit** conditional header

**let**  $S_{\text{simplified}}$  be a statement constructed from the body of  $S_{\text{forall}}$  by removing operands that have run out of values in  $\mathcal{L}_q$

LOWER( $S_{\text{simplified}}$ )

**Emit** conditional footer

**end for**

**Emit** advance iterators

**Emit** loop footer

**end for**

**end function**

```

int iB = B1_pos[0];
int id = d1_pos[0];
while (iB < B1_pos[1] && id < d1_pos) {
    int iB = B1_crd[iB];
    int id = d1_crd[id];
    int i = min(iB, id);
    if (iB == i && id == i) {

    }
    else if (iB == i) {

    }
    else {
        a[i] = d[id];
    }
    if (iB == i) iB++;
    if (id == i) id++;
}

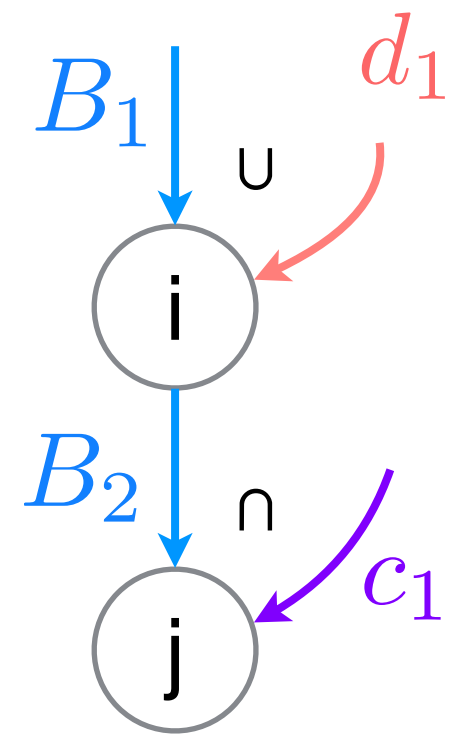
while (iB < B1_pos[1]) {
    int i = B1_crd[iB];
    for (int jB = B2_pos[iB]; jB < B2_pos[iB+1]; jB++) {
        int j = B2_crd[jB];
        a[i] += B[jB] * c[j];
    }
    iB++;
}

while (id < d1_pos) {
    int i = d1_crd[id];
    a[i] = d[id];
    id++;
}

```

# Recursive lowering of concrete index notation

$$a_i = B_{ij} c_j + d_i$$



$$a_i = B_{ij} c_j + d_i$$

```

int iB = B1_pos[0];
int id = d1_pos[0];
while (iB < B1_pos[1] && id < d1_pos) {
    int iB = B1_crd[iB];
    int id = d1_crd[id];
    int i = min(iB, id);
    if (iB == i && id == i) {
        double tj = 0.0;
        for (int jB = B2_pos[iB]; jB < B2_pos[iB+1]; jB++) {
            int j = B2_crd[jB];
            tj += B[jB] * c[j];
        }
        a[i] = tj + d[id];
    }
    else if (iB == i) {
        double tj = 0.0;
        for (int jB = B2_pos[iB]; jB < B2_pos[iB+1]; jB++) {
            int j = B2_crd[jB];
            tj += B[jB] * c[j];
        }
        a[i] = tj + d[id];
    }
    else {
        a[i] = d[id];
    }
    if (iB == i) iB++;
    if (id == i) id++;
}

while (iB < B1_pos[1]) {
    int i = B1_crd[iB];
    for (int jB = B2_pos[iB]; jB < B2_pos[iB+1]; jB++) {
        int j = B2_crd[jB];
        a[i] += B[jB] * c[j];
    }
    iB++;
}

while (id < d1_pos) {
    int i = d1_crd[id];
    a[i] = d[id];
    id++;
}

```

**function** LOWER(forall statement  $S_{\text{forall}}$  of index variable  $i$ )

**let**  $\mathcal{L}$  be an iteration lattice constructed from  $S_{\text{forall}}$

**Emit** initialize iterators

**for** each lattice point  $\mathcal{L}_p$  in  $\mathcal{L}$  **do**

**Emit** loop header

**Emit** access iterators

**Emit** resolve the coordinate of  $i$

**Emit** locate from locators

**for** each lattice point  $\mathcal{L}_q < \mathcal{L}_p$  in  $\mathcal{L}$  **do**

**Emit** conditional header

**let**  $S_{\text{simplified}}$  be a statement constructed from the body of  $S_{\text{forall}}$  by removing operands that have run out of values in  $\mathcal{L}_q$

LOWER( $S_{\text{simplified}}$ )

**Emit** conditional footer

**end for**

**Emit** advance iterators

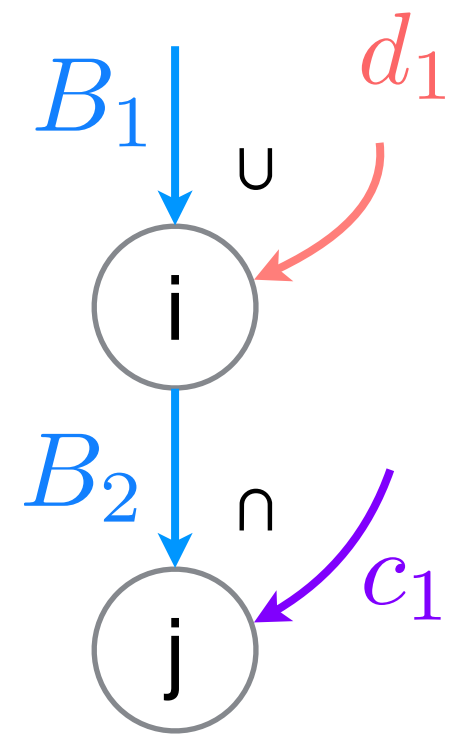
**Emit** loop footer

**end for**

**end function**

# Recursive lowering of concrete index notation

$$a_i = B_{ij} c_j + d_i$$



$$a_i = B_{ij} c_j + d_i$$

$$a_i = B_{ij} c_j$$

```

function LOWER(forall statement  $S_{forall}$  of index variable  $i$ )
  let  $\mathcal{L}$  be an iteration lattice constructed from  $S_{forall}$ 
  Emit initialize iterators
  for each lattice point  $\mathcal{L}_p$  in  $\mathcal{L}$  do
    Emit loop header
    Emit access iterators
    Emit resolve the coordinate of  $i$ 
    Emit locate from locators
    for each lattice point  $\mathcal{L}_q < \mathcal{L}_p$  in  $\mathcal{L}$  do
      Emit conditional header
      let  $S_{simplified}$  be a statement constructed from
        the body of  $S_{forall}$  by removing operands
        that have run out of values in  $\mathcal{L}_q$ 
      LOWER( $S_{simplified}$ )
      Emit conditional footer
    end for
    Emit advance iterators
    Emit loop footer
  end for
end function
  
```

```

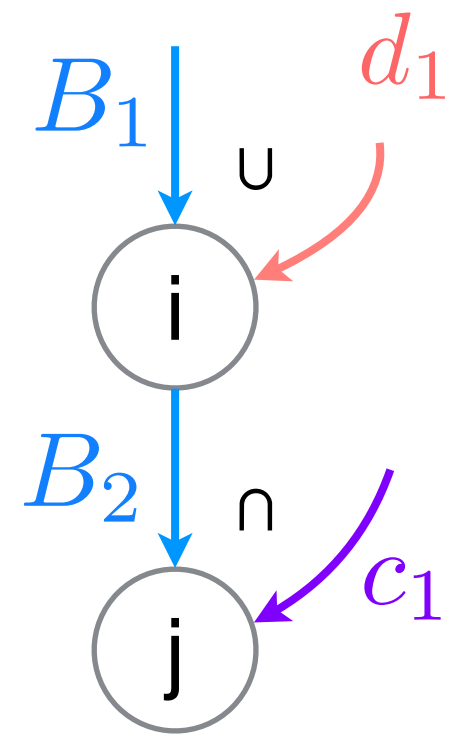
int iB = B1_pos[0];
int id = d1_pos[0];
while (iB < B1_pos[1] && id < d1_pos) {
  int iB = B1_crd[iB];
  int id = d1_crd[id];
  int i = min(iB, id);
  if (iB == i && id == i) {
    double tj = 0.0;
    for (int jB = B2_pos[iB]; jB < B2_pos[iB+1]; jB++) {
      int j = B2_crd[jB];
      tj += B[jB] * c[j];
    }
    a[i] = tj + d[id];
  }
  else if (iB == i) {
    for (int jB = B2_pos[iB]; jB < B2_pos[iB+1]; jB++) {
      int j = B2_crd[jB];
      a[i] += B[jB] * c[j];
    }
  }
  else {
    a[i] = d[id];
  }
  if (iB == i) iB++;
  if (id == i) id++;
}

while (iB < B1_pos[1]) {
  int i = B1_crd[iB];
  for (int jB = B2_pos[iB]; jB < B2_pos[iB+1]; jB++) {
    int j = B2_crd[jB];
    a[i] += B[jB] * c[j];
  }
  iB++;
}

while (id < d1_pos) {
  int i = d1_crd[id];
  a[i] = d[id];
  id++;
}
  
```

# Recursive lowering of concrete index notation

$$a_i = B_{ij} c_j + d_i$$



$$a_i = B_{ij} c_j + d_i$$

$$a_i = B_{ij} c_j$$

$$a_i = d_i$$

```

int iB = B1_pos[0];
int id = d1_pos[0];
while (iB < B1_pos[1] && id < d1_pos) {
    int iB = B1_crd[iB];
    int id = d1_crd[id];
    int i = min(iB, id);
    if (iB == i && id == i) {
        double tj = 0.0;
        for (int jB = B2_pos[iB]; jB < B2_pos[iB+1]; jB++) {
            int j = B2_crd[jB];
            tj += B[jB] * c[j];
        }
        a[i] = tj + d[id];
    }
    else if (iB == i) {
        for (int jB = B2_pos[iB]; jB < B2_pos[iB+1]; jB++) {
            int j = B2_crd[jB];
            a[i] += B[jB] * c[j];
        }
    }
    else {
        a[i] = d[id];
    }
    if (iB == i) iB++;
    if (id == i) id++;
}

while (iB < B1_pos[1]) {
    int i = B1_crd[iB];
    for (int jB = B2_pos[iB]; jB < B2_pos[iB+1]; jB++) {
        int j = B2_crd[jB];
        a[i] += B[jB] * c[j];
    }
    iB++;
}

while (id < d1_pos) {
    int i = d1_crd[id];
    a[i] = d[id];
    id++;
}

```

**function** LOWER(forall statement  $S_{\text{forall}}$  of index variable  $i$ )

**let**  $\mathcal{L}$  be an iteration lattice constructed from  $S_{\text{forall}}$

**Emit** initialize iterators

**for** each lattice point  $\mathcal{L}_p$  in  $\mathcal{L}$  **do**

**Emit** loop header

**Emit** access iterators

**Emit** resolve the coordinate of  $i$

**Emit** locate from locators

**for** each lattice point  $\mathcal{L}_q < \mathcal{L}_p$  in  $\mathcal{L}$  **do**

**Emit** conditional header

**let**  $S_{\text{simplified}}$  be a statement constructed from the body of  $S_{\text{forall}}$  by removing operands that have run out of values in  $\mathcal{L}_q$

LOWER( $S_{\text{simplified}}$ )

**Emit** conditional footer

**end for**

**Emit** advance iterators

**Emit** loop footer

**end for**

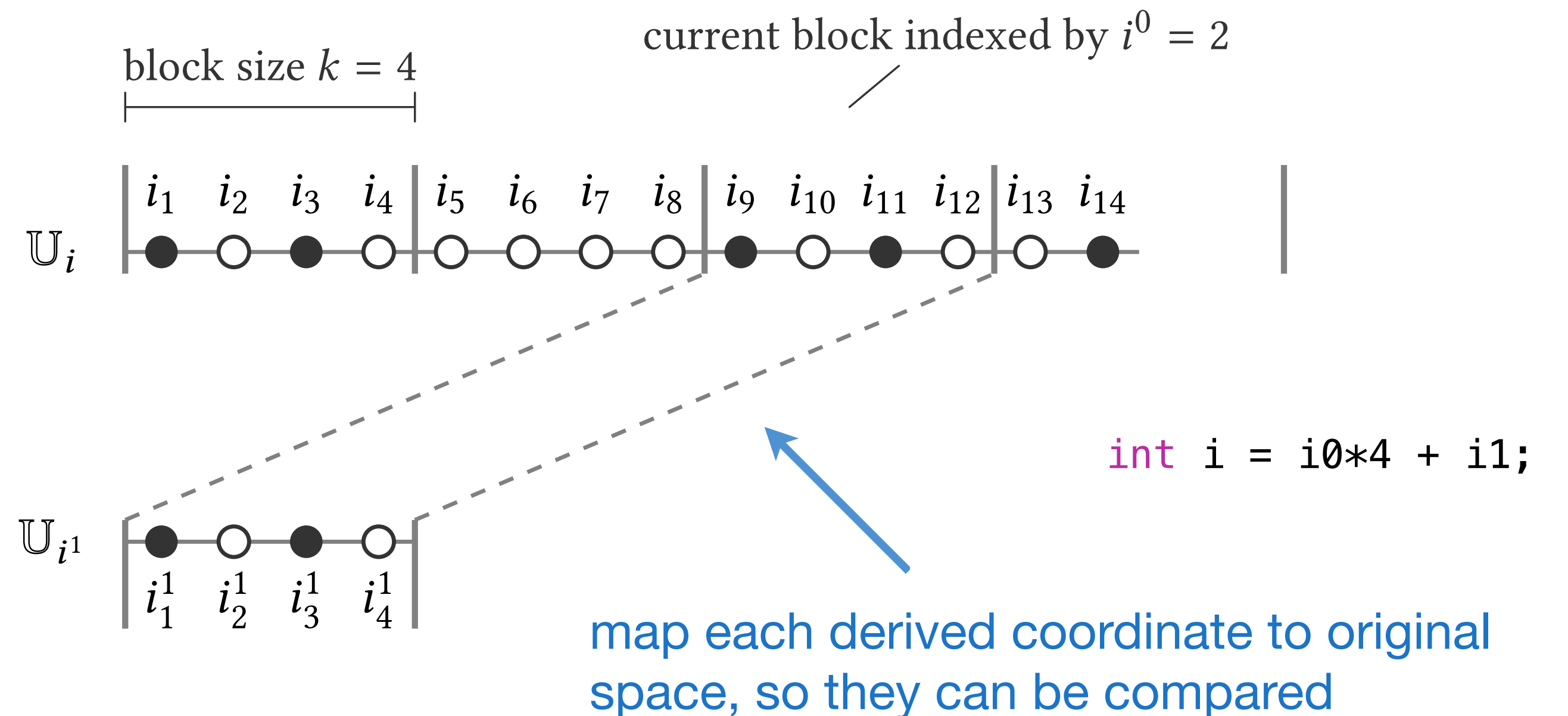
**end function**

# Coordinate remapping

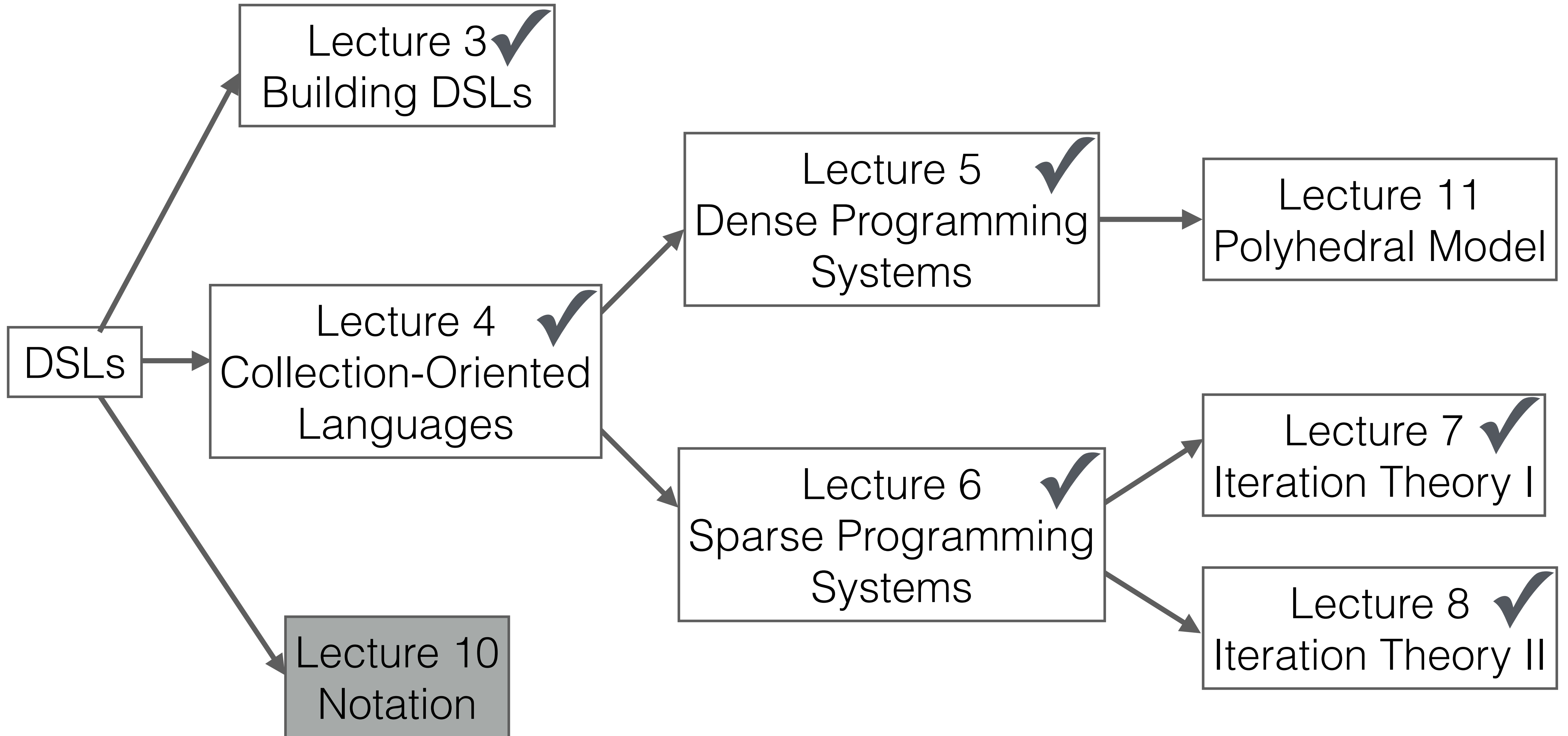
```
int ib = b_crd[ib];
int ic = c_crd[ic];
int i = min(ib, ic);
```

```
function LOWER(forall statement  $S_{forall}$  of index variable  $i$ )
  let  $\mathcal{L}$  be an iteration lattice constructed from  $S_{forall}$ 
  Emit initialize iterators
  for each lattice point  $\mathcal{L}_p$  in  $\mathcal{L}$  do
    Emit loop header
    Emit access iterators
    Emit map candidate coordinates to the original space
    Emit resolve the coordinate of  $i$ 
    Emit map resolved coordinate to each derived space
    Emit locate from locators
    for each lattice point  $\mathcal{L}_q < \mathcal{L}_p$  in  $\mathcal{L}$  do
      Emit conditional header
      let  $S_{simplified}$  be a statement constructed from
        the body of  $S_{forall}$  by removing operands
        that have run out of values in  $\mathcal{L}_q$ 
      LOWER( $S_{simplified}$ )
      Emit conditional footer
    end for
    Emit advance iterators
    Emit loop footer
  end for
end function
```

$$\forall_{i_0} \forall_{i_1} b_i \cap c_i : i \xrightarrow{\text{split}(\downarrow, 4)} i^0 i^1$$

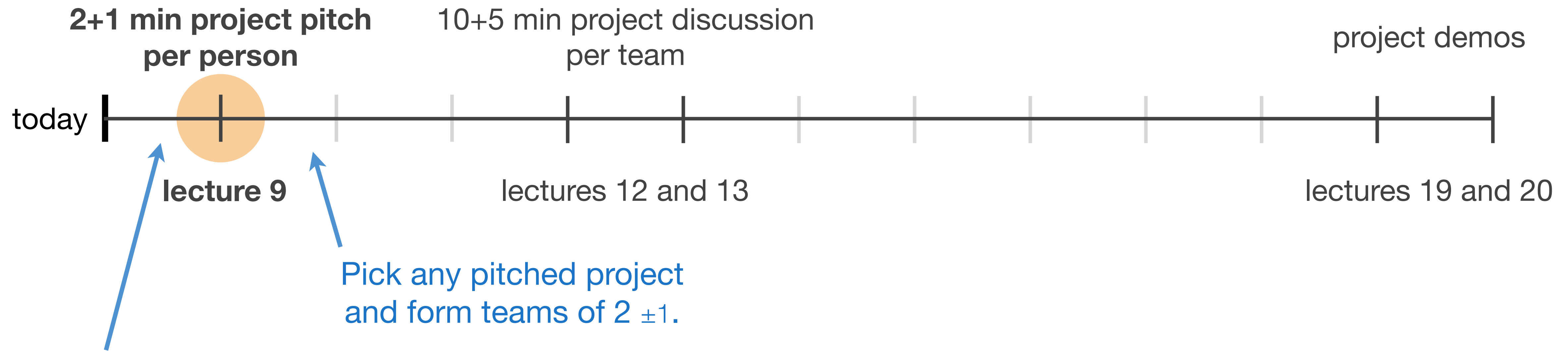


# Overview of lectures in the coming weeks





# Course Project



Each person contributes one pitch slide to a google slide deck. These pitches are not binding.