

The Tensor Algebra Compiler

$$\begin{array}{c} a = Bc \\ A = B \quad a = B^T c \\ a = Bc + a \quad a = Bc + b \quad a = b + c \\ A_{ij} = \sum_k B_{ijk} * c_k \quad a = B^T c + d \quad a = \alpha Bc + \beta a \\ a = BCd \quad A = B + C \quad A = B \odot C \quad A = BC \quad A = 0 \\ A = \alpha B \quad A = \alpha A + \beta B \quad A = \alpha B + A \quad A = A + \alpha I \quad A = B + C + D \\ A = (B + C)(d + e) \quad A_{ik} = \sum_j B_{ijk} * c_j \quad A_{ij} = \sum_k B_{ijk} * c_k \quad A_{ilk} = \sum_j B_{ijk} * C_{lj} \quad A_{il} = \sum_{j,k} B_{ijk} * C_{jl} * D_{kl} \\ A_{jl} = \sum_{i,k} B_{ijk} * C_{il} * D_{kl} \quad A_{jl} = \sum_{i,k} B_{ijk} * C_{il} * D_{kl} \quad A_{ik} = \sum_i B_{ijk} * c_j \quad A_{jl} = \sum_{i,k} B_{ijk} * C_{il} * D_{kl} \quad A_{ijl} = \sum_k B_{ijk} * C_{lk} \\ A_{ij} = B_{ij}(C_{ik}D_{kj}) \quad A_{ij} = B_{ij} \sum_k (C_{ik}D_{kj}) \quad A_{ij} = \sum_k B_{ijk}C_{ijk} + D_{ij} \quad A_{ij} = \sum_k C_{ijk}D_{ijk} \quad A_{ij} = \sum_{l,k} B_{lik} * C_{lj} * D_{kj} \quad A_{ij} = \sum_{k,l} B_{ikjl} * C_{kl} \end{array}$$

Fredrik Kjolstad, Shoaib Kamil, Stephen Chou, David Lugato, and Saman Amarasinghe



Tensors are everywhere

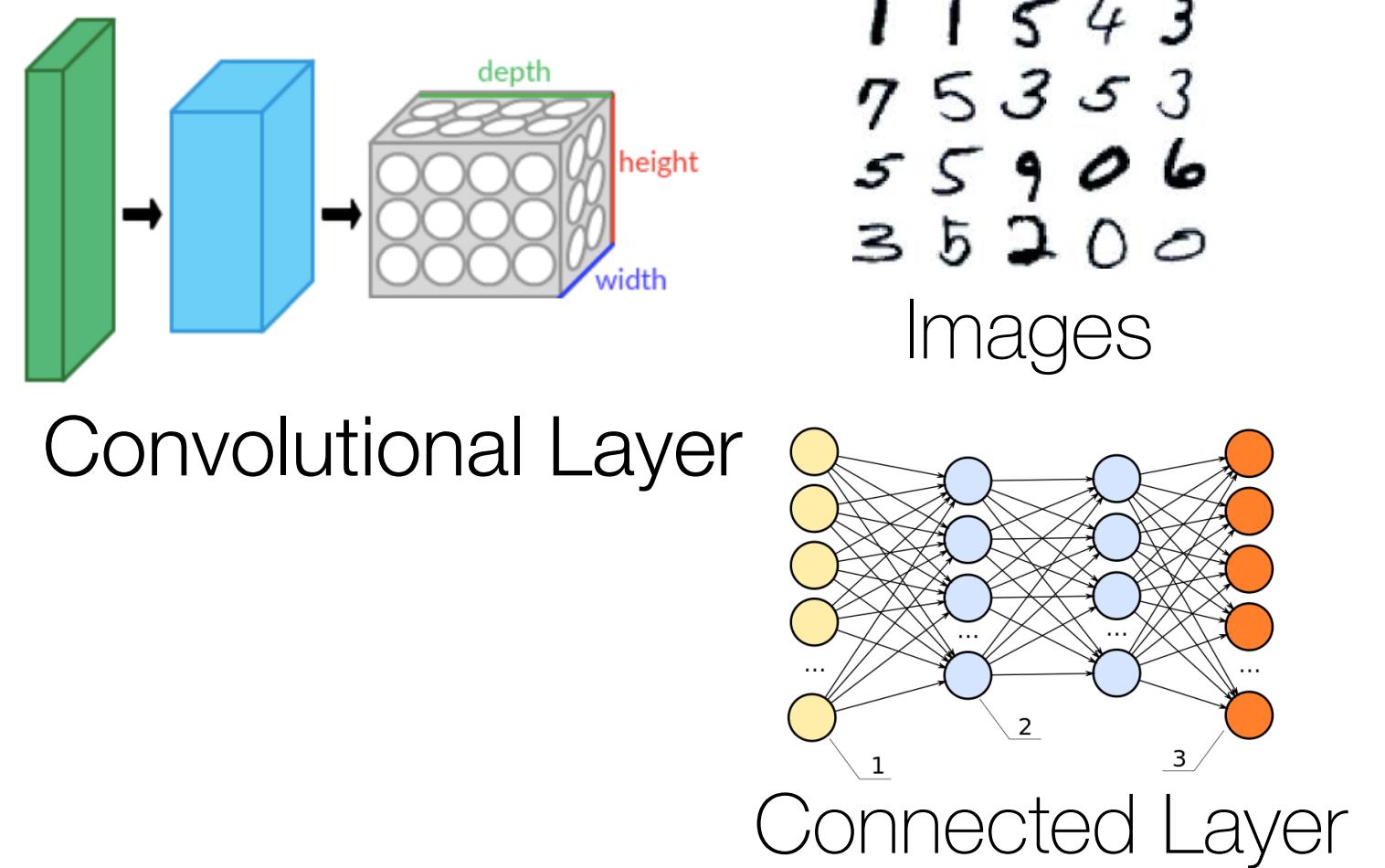
Data Analytics

NETFLIX
Movie ratings

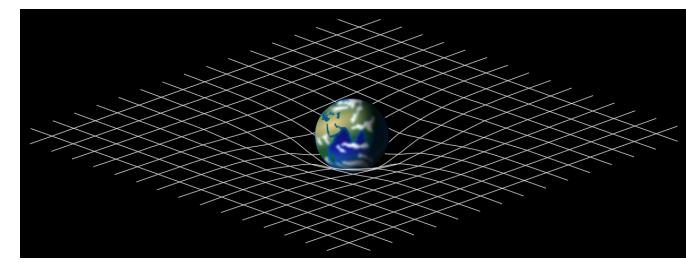
amazon
Product Reviews

facebook
Social interactions

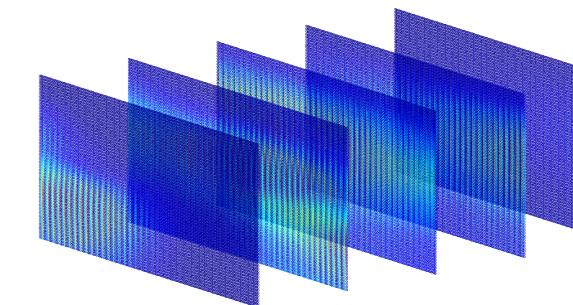
Machine Learning



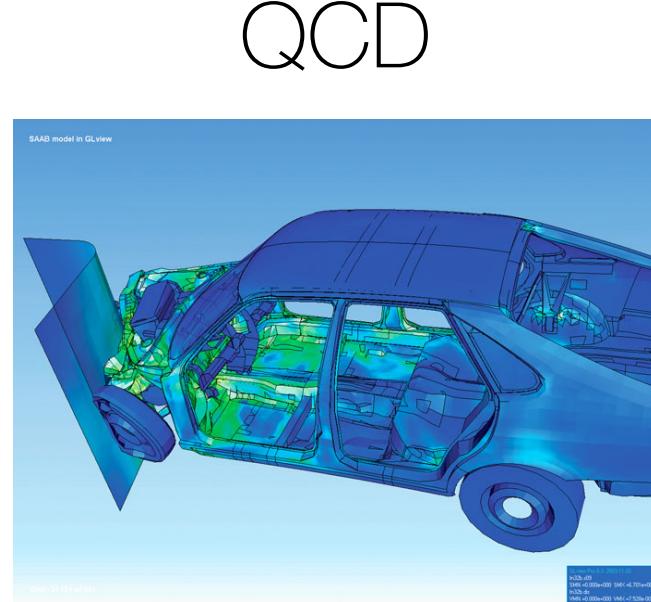
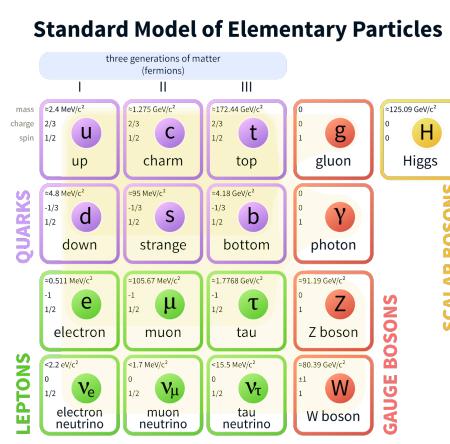
Science and Engineering



General Relativity



Fluorescence spectroscopy



Finite Element Method

Tensors are everywhere

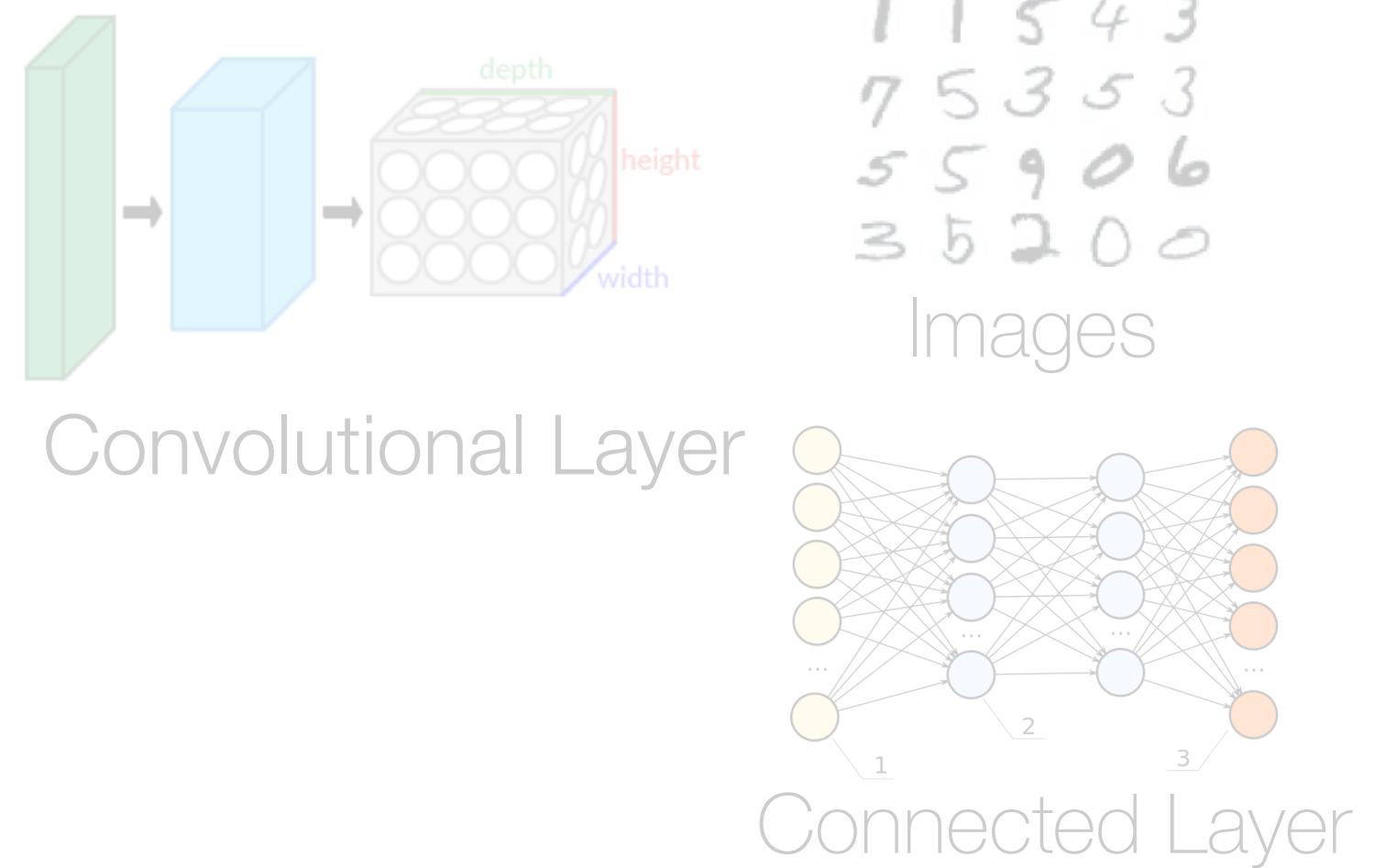
Data Analytics

NETFLIX
Movie ratings

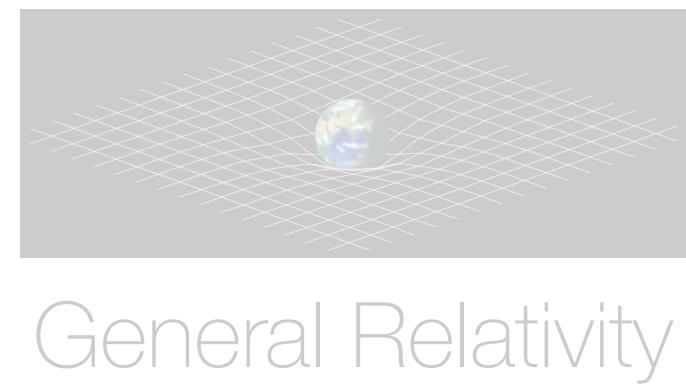
amazon
Product Reviews

facebook
Social interactions

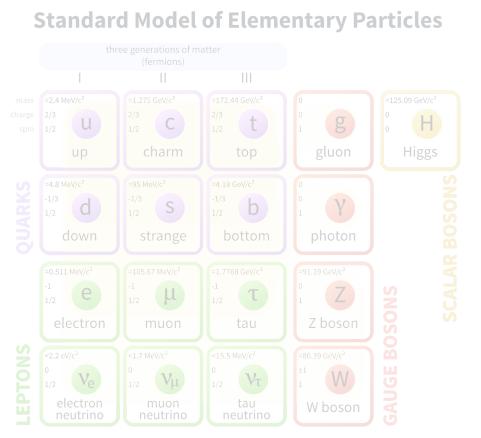
Machine Learning



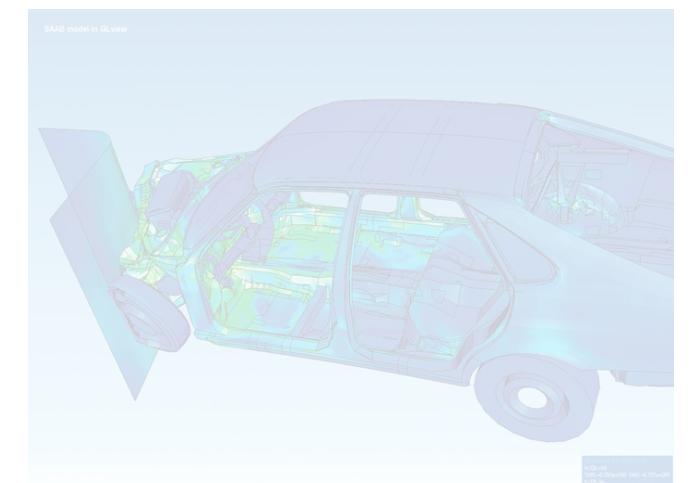
Science and Engineering



General Relativity



QCD



Finite Element Method

Tensors are everywhere

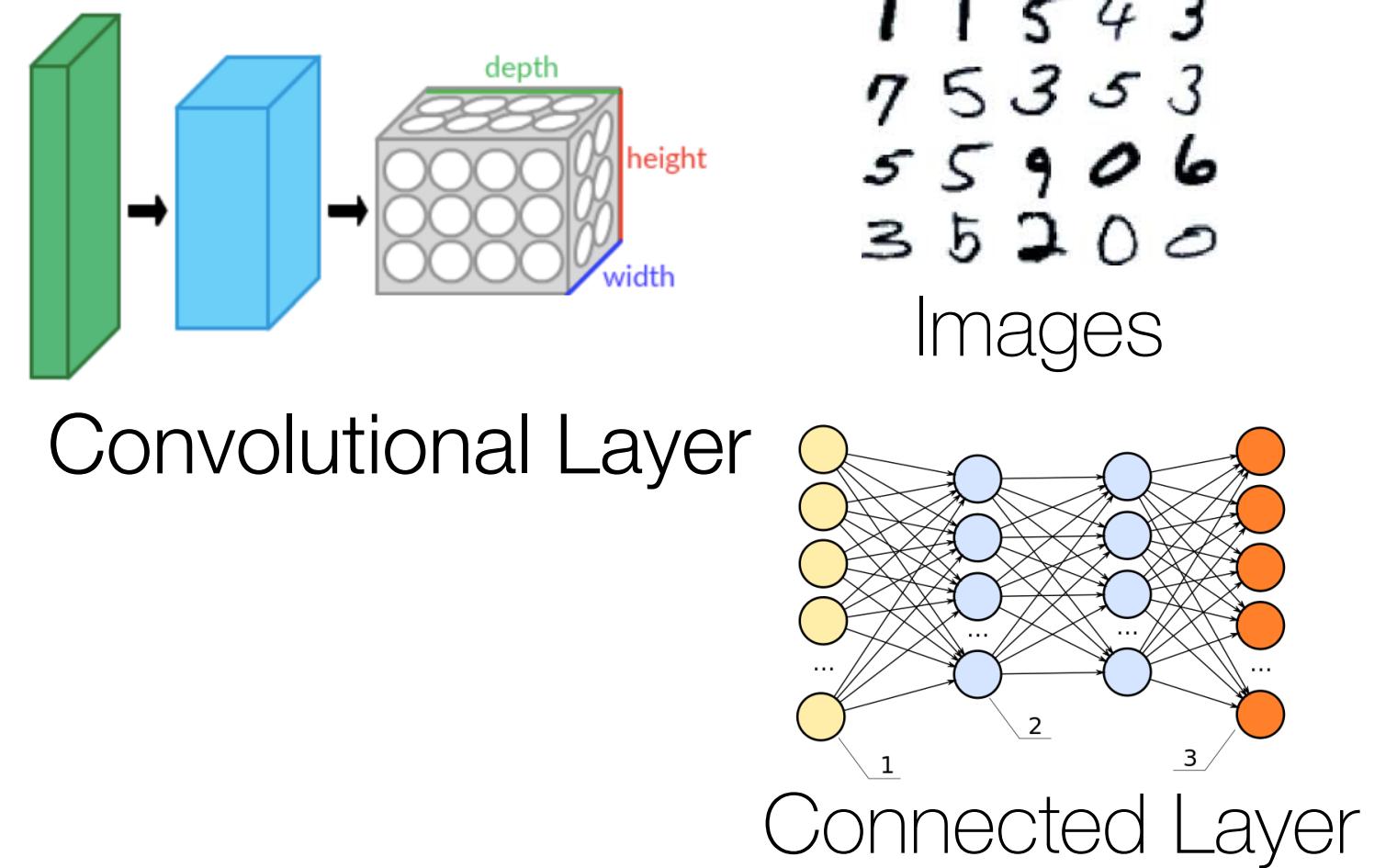
Data Analytics

NETFLIX
Movie ratings

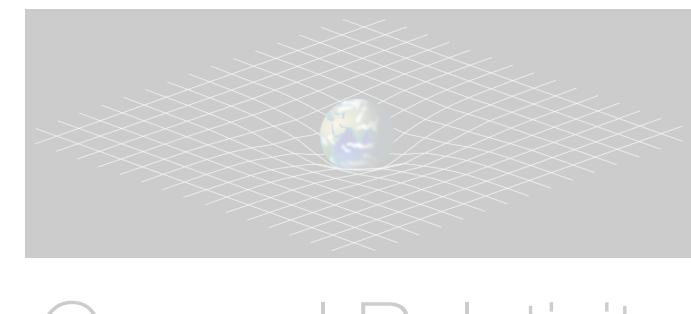
amazon
Product Reviews

facebook
Social interactions

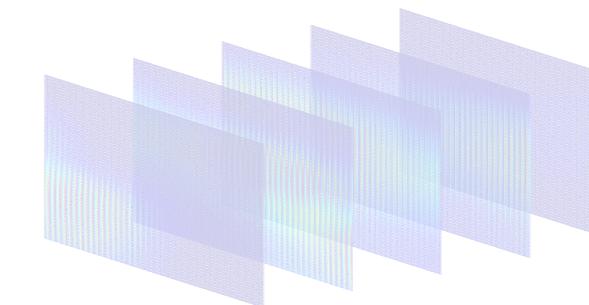
Machine Learning



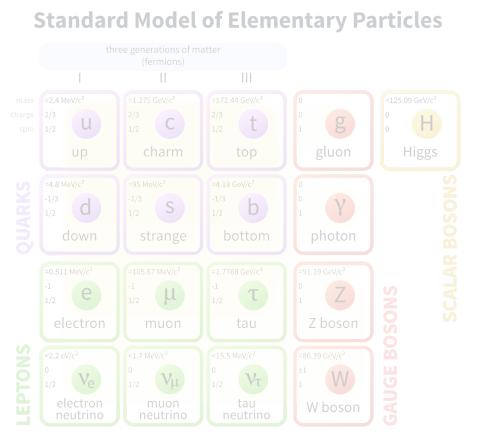
Science and Engineering



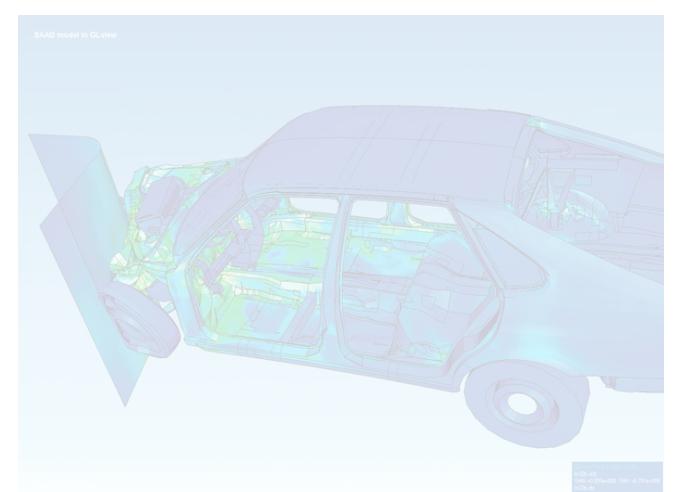
General Relativity



Fluorescence
spectroscopy



QCD



Finite Element Method

Tensors are everywhere

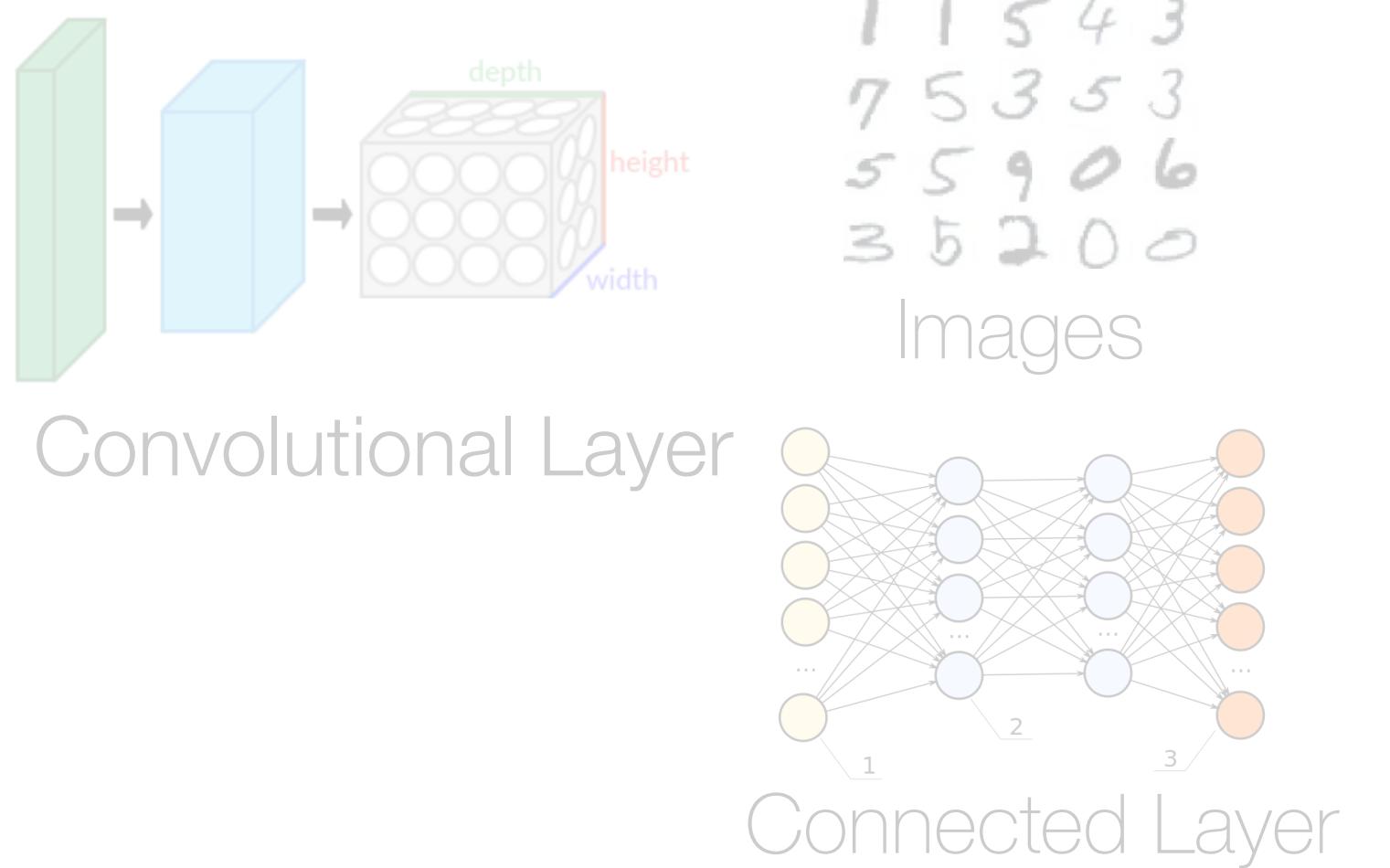
Data Analytics

NETFLIX
Movie ratings

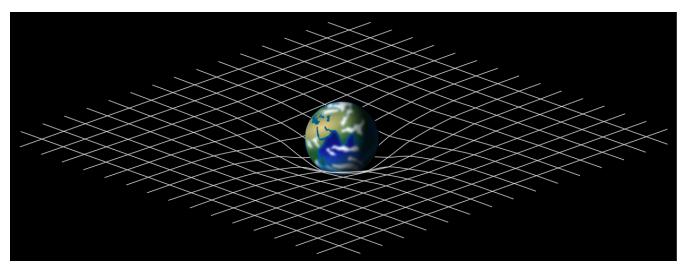
amazon
Product Reviews

facebook
Social interactions

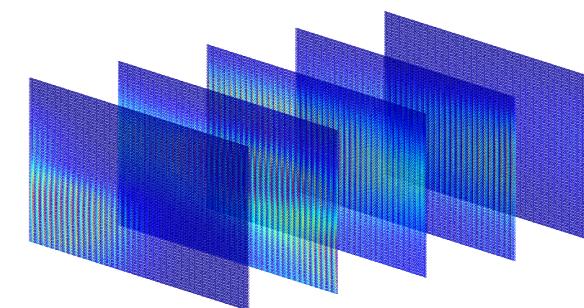
Machine Learning



Science and Engineering



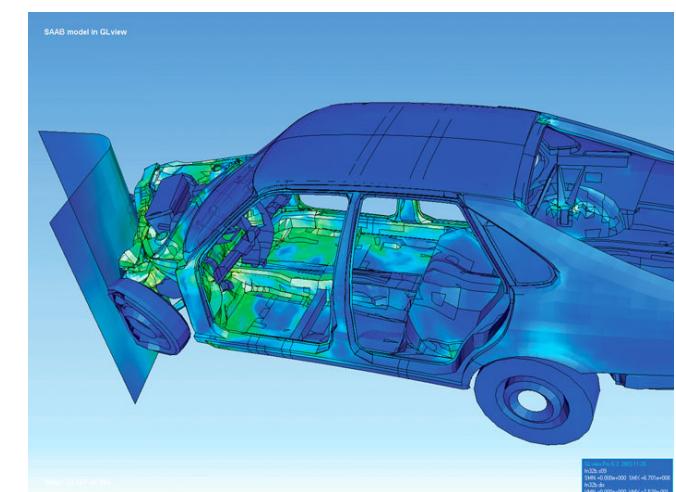
General Relativity



Fluorescence
spectroscopy

Standard Model of Elementary Particles		
three generations of matter (fermions)		
mass	2.4 MeV/c ²	127.0 GeV/c ²
charge	2/3	-1/3
up	u	c
down	d	s
strange	t	b
top	g	Y
bottom	H	Z boson
leptons	e	tau
electron	ν _e	ν _τ
muon	μ	τ
neutrino	ν _μ	ν _τ
GAUGE BOSONS	W	Z boson

QCD



Finite Element Method

Tensors are everywhere

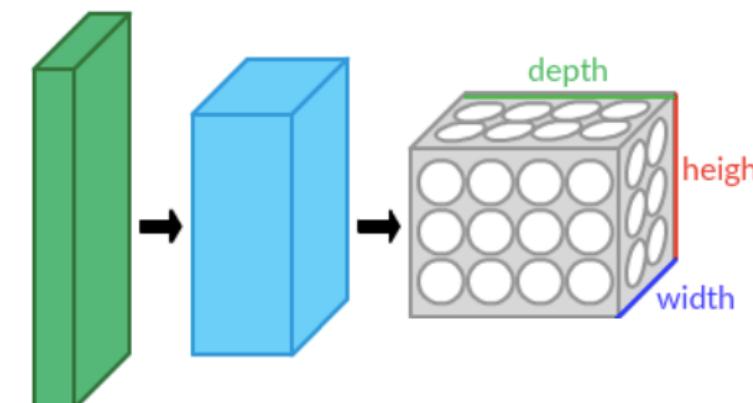
Data Analytics

NETFLIX
Movie ratings

amazon
Product Reviews

facebook
Social interactions

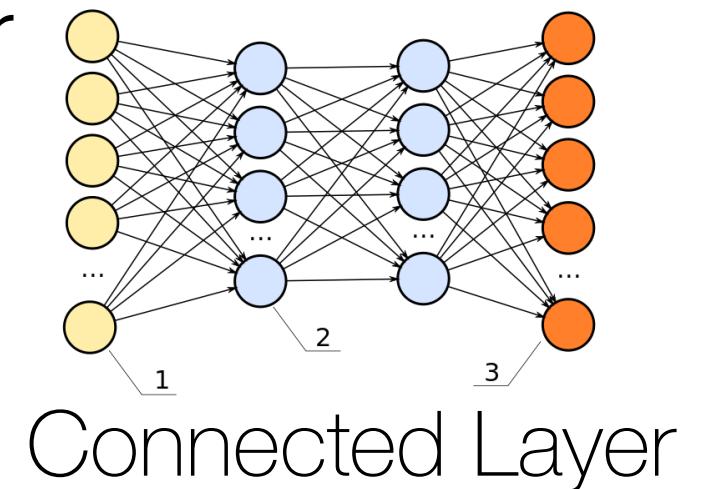
Machine Learning



1 1 5 4 3
7 5 3 5 3
5 5 9 0 6
3 5 2 0 0

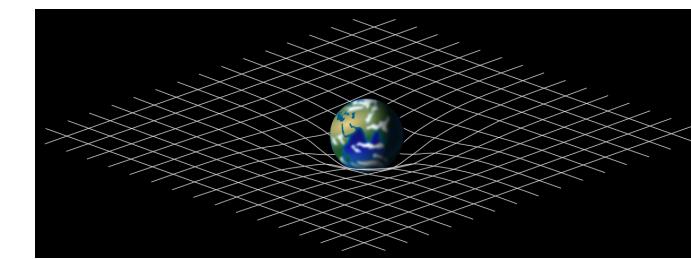
Images

Convolutional Layer

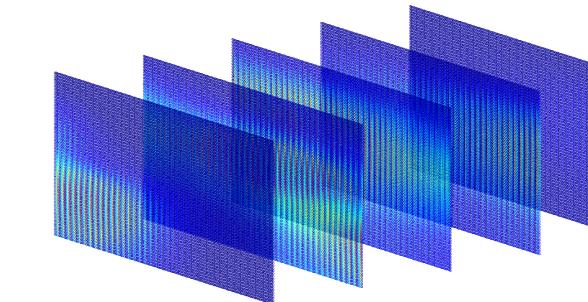


Connected Layer

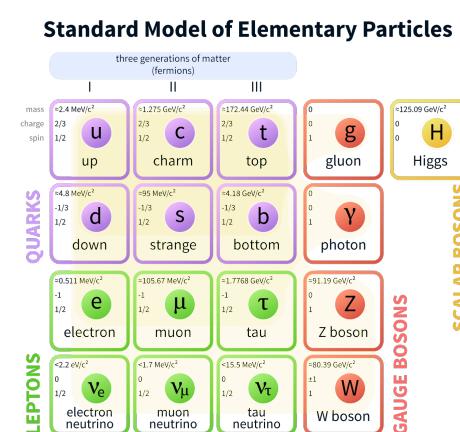
Science and Engineering



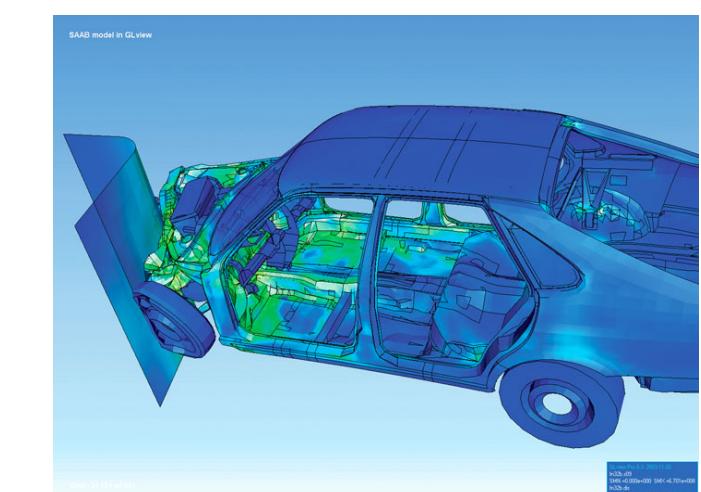
General Relativity



Fluorescence spectroscopy



QCD



Finite Element Method

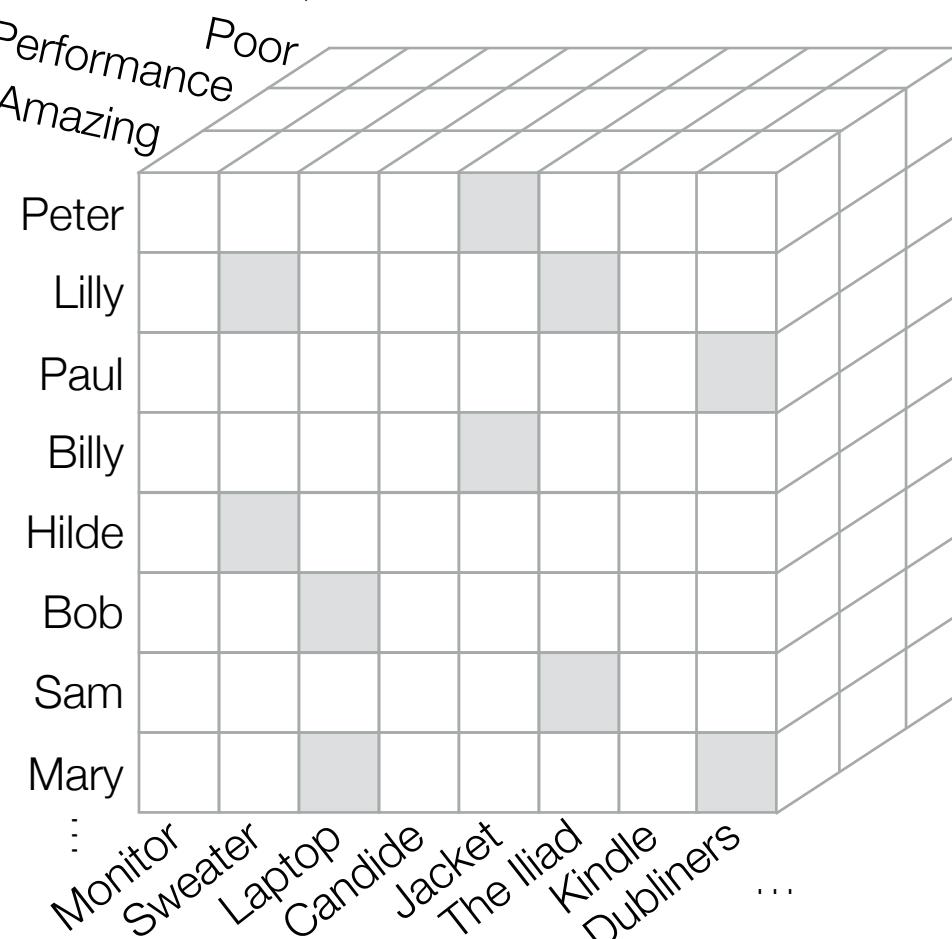
Scalars have 0 dimensions
Vectors have 1 dimension
Matrices have 2 dimensions
An n-tensor has n dimensions

Tensors are everywhere

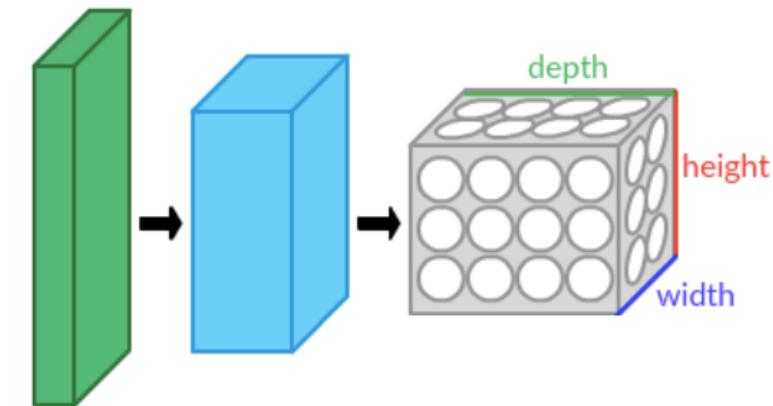
Data Analytics



Words
Users
Products



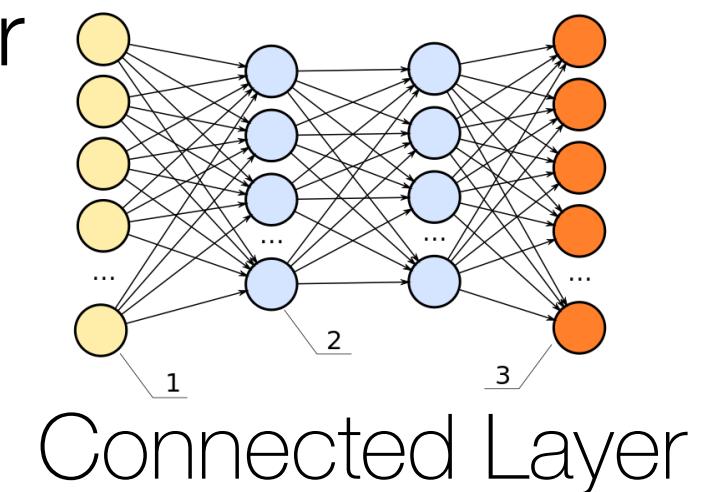
Machine Learning



Convolutional Layer

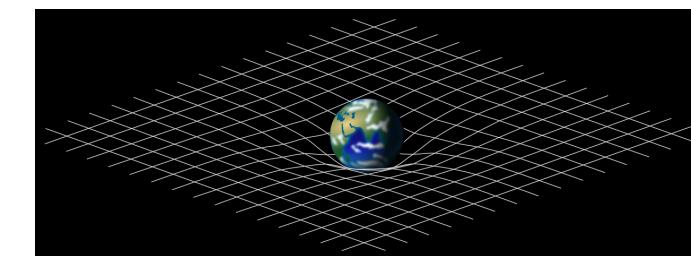
1 1 5 4 3
7 5 3 5 3
5 5 9 0 6
3 5 2 0 0

Images

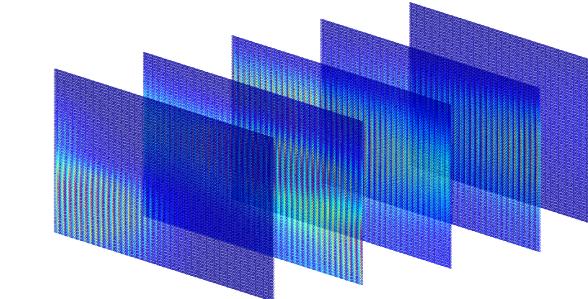


Connected Layer

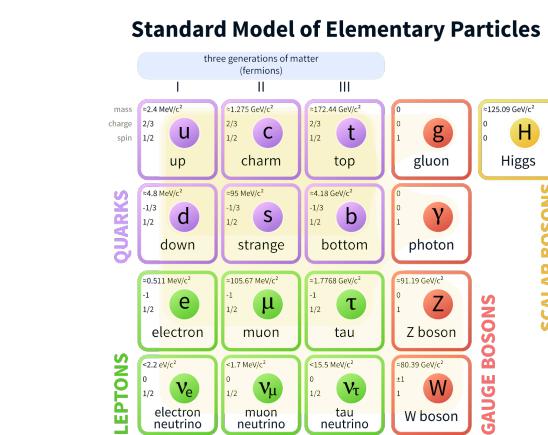
Science and Engineering



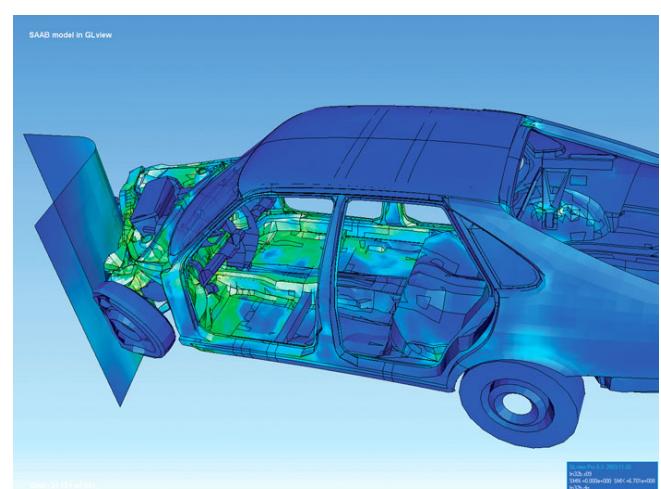
General Relativity



QCD
Fluorescence spectroscopy



Standard Model of Elementary Particles



Finite Element Method

Tensors are everywhere

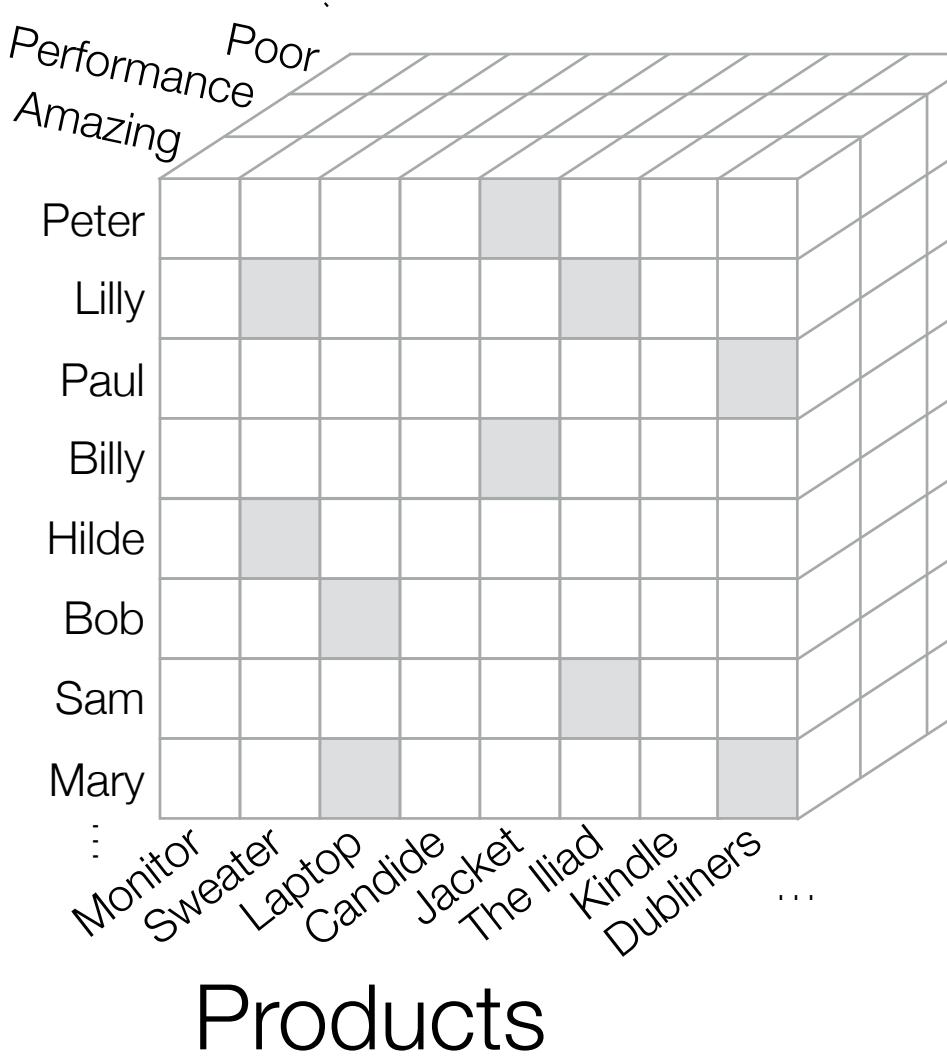
Data Analytics

NETFLIX
Movie ratings

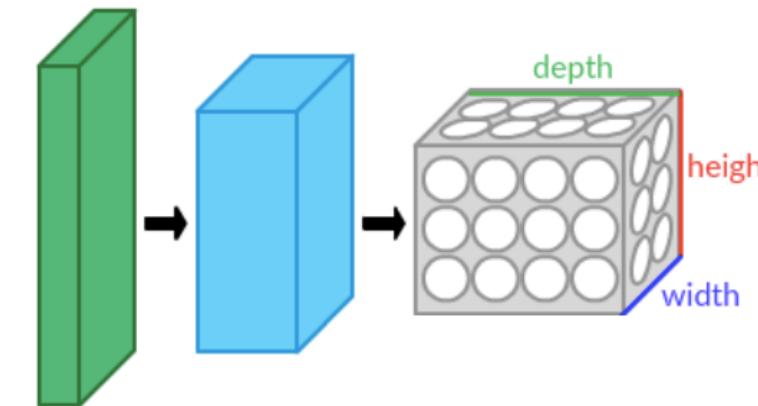
facebook
Social interactions

amazon
Product Reviews

Words
Users
Products



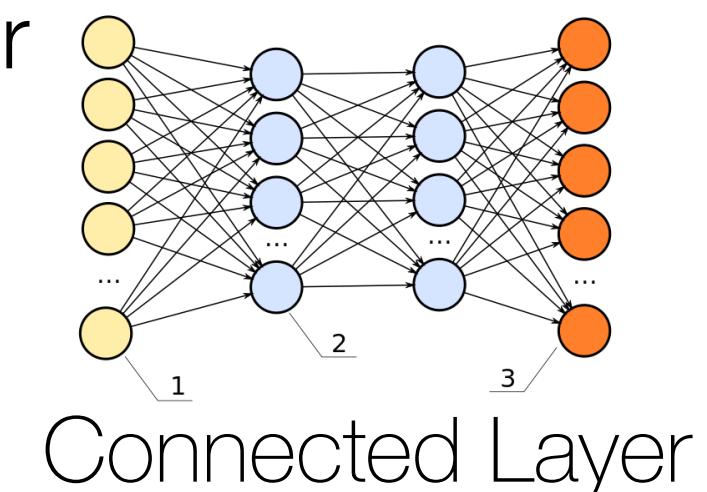
Machine Learning



Convolutional Layer

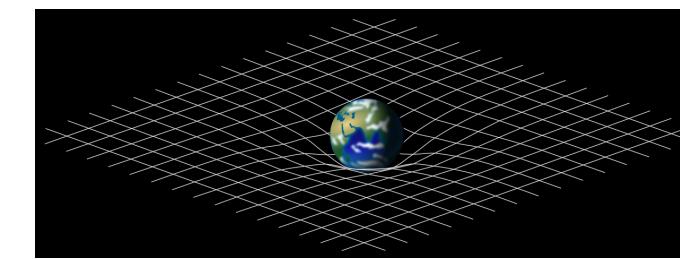
1 1 5 4 3
7 5 3 5 3
5 5 9 0 6
3 5 2 0 0

Images

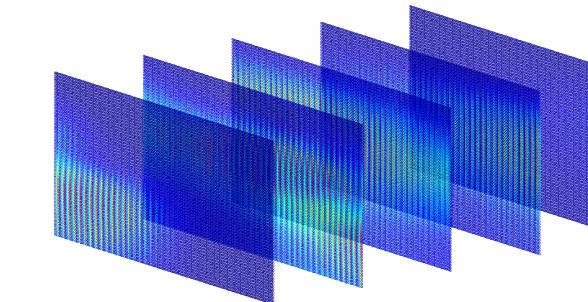


Connected Layer

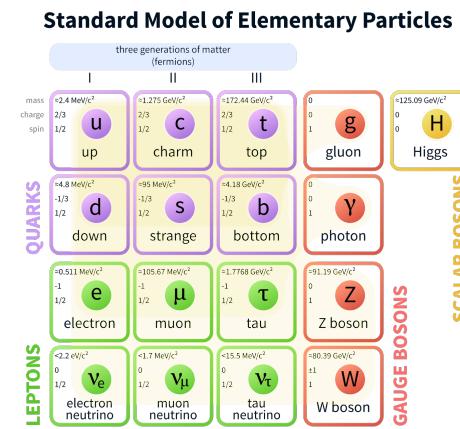
Science and Engineering



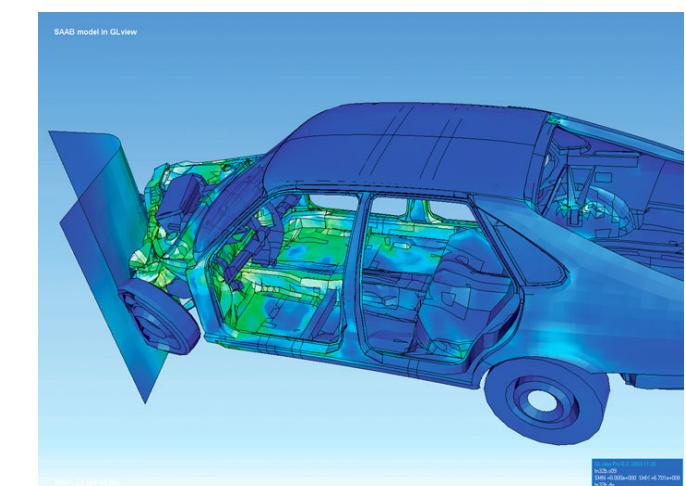
General Relativity



Fluorescence spectroscopy



QCD



Finite Element Method

Extremely sparse
Dense storage: 107 Exabytes
Sparse storage: 13 Gigabytes

We need a Tensor Algebra Compiler

Number of Variants:

We need a Tensor Algebra Compiler

$$a = Bc$$

We need a Tensor Algebra Compiler

$a = Bc$ Eigen

We need a Tensor Algebra Compiler

$a = Bc$ Eigen

$a = Bc + a$ CSparse

We need a Tensor Algebra Compiler

$a = Bc$ Eigen

$a = Bc + a$ CSparse

$a = \alpha Bc + \beta a$ OSKI

We need a Tensor Algebra Compiler

$a = Bc$ Eigen

$a = Bc + a$ CSparse

$a = \alpha Bc + \beta a$ OSKI

$a = Bc + b$ PETSc

$a = B^T c$ PETSc

$a = B^T c + d$ PETSc

We need a Tensor Algebra Compiler

$$a = Bc \quad \text{Eigen}$$

$$a = Bc + a \quad \text{CSparse}$$

$$a = \alpha Bc + \beta a \quad \text{OSKI}$$

$$a = Bc + b \quad \text{PETSc}$$

$$a = B^T c \quad \text{PETSc}$$

$$a = B^T c + d \quad \text{PETSc}$$



Binary kernels are not enough

We need a Tensor Algebra Compiler

$a = Bc$	Eigen	
$a = Bc + a$	CSparse	CSR
$a = \alpha Bc + \beta a$	OSKI	CSC
$a = Bc + b$	PETSc	COO
$a = B^T c$	PETSc	ELL
$a = B^T c + d$	PETSc	DIA

We need a Tensor Algebra Compiler

$a = Bc$	Eigen	
$a = Bc + a$	CSparse	(B) CSR
$a = \alpha Bc + \beta a$	OSKI	(B) CSC
$a = Bc + b$	PETSc	\times (B) COO
$a = B^T c$	PETSc	(B) ELL
$a = B^T c + d$	PETSc	(B) DIA

We need a Tensor Algebra Compiler

$a = Bc$	Eigen	
$a = Bc + a$	CSparse	(V)(B) CSR
$a = \alpha Bc + \beta a$	OSKI	(V)(B) CSC
$a = Bc + b$	PETSc	\times (V)(B) COO
$a = B^T c$	PETSc	(V)(B) ELL
$a = B^T c + d$	PETSc	(V)(B) DIA

We need a Tensor Algebra Compiler

$a = Bc$	Eigen	
$a = Bc + a$	CSparse	(D)(V)(B) CSR
$a = \alpha Bc + \beta a$	OSKI	(D)(V)(B) CSC
$a = Bc + b$	PETSc	\times (V)(B) COO
$a = B^T c$	PETSc	(V)(B) ELL
$a = B^T c + d$	PETSc	(V)(B) DIA

We need a Tensor Algebra Compiler

$a = Bc$	Eigen			
$a = Bc + a$	CSparse	(D)(V)(B) CSR		
$a = \alpha Bc + \beta a$	OSKI	(D)(V)(B) CSC		Dense Vectors
$a = Bc + b$	PETSc	(V)(B) COO	\times	Sparse Vectors
$a = B^T c$	PETSc	(V)(B) ELL		
$a = B^T c + d$	PETSc	(V)(B) DIA		

We need a Tensor Algebra Compiler

$a = Bc$	Eigen	(D)(V)(B) CSR	
$a = Bc + a$	CSparse	(D)(V)(B) CSC	
$a = \alpha Bc + \beta a$	OSKI	(V)(B) COO	Dense Vectors
$a = Bc + b$	PETSc	(V)(B) ELL	Sparse Vectors
$a = B^T c$	PETSc	(V)(B) DIA	
$a = B^T c + d$	PETSc		

Infinite number of possible tensor formats and expressions

$A = B$	$a = B^T c$		
$a = Bc + a$	$a = Bc + b$	$a = b + c$	$a = \alpha Bc + \beta a$
$A_{ij} = \sum_k B_{ijk} * c_k$	$a = B^T c + d$		
$a = BCd$	$A = B + C$	$A = B \odot C$	$A = BC$
			$A = 0$
			$A = B^T$
			$A = (B + C)d$
$A = \alpha B$	$A = \alpha A + \beta B$	$A = \alpha B + A$	$A = A + \alpha I$
$A = (B + C)(d + e)$	$A_{ik} = \sum_j B_{ijk} * c_j$	$A_{ij} = \sum_k B_{ijk} * c_k$	$A = B + C + D$
$A_{jl} = \sum_{i,k} B_{ijk} * C_{il} * D_{kl}$	$A_{jl} = \sum_{i,k} B_{ijk} * C_{il} * D_{kl}$	$A_{ik} = \sum_i B_{ijk} * c_j$	$A_{il} = \sum_{j,k} B_{ijk} * C_{jl} * D_{kl}$
$A_{ij} = B_{ij} \sum_k (C_{ik} D_{kj})$	$A_{ij} = B_{ij} (C_{ik} D_{kj})$	$A_{jl} = \sum_{i,k} B_{ijk} * C_{il} * D_{kl}$	$A_{ijl} = \sum_k B_{ijk} * C_{lk}$
		$A_{ij} = \sum_k B_{ijk} C_{ijk} + D_{ij}$	$A_{ij} = \sum_l B_{lik} * C_{lj} * D_{kj}$
		$A_{ij} = \sum_k C_{ijk} D_{ijk}$	$A_{ij} = \sum_{k,l} B_{ikl} * C_{kl}$
			$A_{ij} = \sum_{k,l} B_{ikl} * C_{kl}$

Number of Variants: ∞

Advantages of a Tensor Algebra Compiler

Traditional Libraries

Choose a few expressions

Choose a few formats

Spend years hand optimizing kernels

The Tensor Algebra Compiler (taco)

Any expression

Many formats

Compiler produces optimized code

Advantages of a Tensor Algebra Compiler

Traditional Libraries

Choose a few expressions

Choose a few formats

Spend years hand optimizing kernels

The Tensor Algebra Compiler (taco)

Any expression

Many formats

Compiler produces optimized code

$$A = B + C \quad a = b + c$$

$$A = BC \quad a = \alpha Bc + \beta a$$

$$A = \alpha B \quad A = B^T$$

Advantages of a Tensor Algebra Compiler

Traditional Libraries

Choose a few expressions

Choose a few formats

Spend years hand optimizing kernels

The Tensor Algebra Compiler (taco)

Any expression

Many formats

Compiler produces optimized code

$$A = B + C \quad a = b + c$$

$$A = BC \quad a = \alpha Bc + \beta a$$

$$A = \alpha B \quad A = B^T$$

$$\begin{array}{llll} a = Bc & A = B & a = Bc + a & a = Bc + b \\ A_{ij} = \sum_k B_{ijk} * c_k & a = b + c & a = B^T c & a = \alpha Bc + \beta a \\ A = B + C & A = B \odot C & a = B^T c + d & A = \alpha B + A \quad A = 0 \\ A = BC & A = \alpha B & A = B^T & A = \alpha A + \beta B \\ \end{array}$$

$$\begin{array}{llll} A_{ik} = \sum_j B_{ijk} * c_j & A = (B + C)d & A_{il} = \sum_{j,k} B_{ijk} * C_{jl} * D_{kl} \\ a = BCd & A = (B + C)(d + e) & A_{il} = \sum_{i,k} B_{ijk} * C_{il} * D_{kl} \\ A_{ijl} = \sum_k B_{ijk} * C_{lk} & A_{ilk} = \sum_j B_{ijk} * C_{lj} & \end{array}$$

Advantages of a Tensor Algebra Compiler

Traditional Libraries

Choose a few expressions

Choose a few formats

Spend years hand optimizing kernels

CSR

CSC

The Tensor Algebra Compiler (taco)

Any expression

Many formats

Compiler produces optimized code

Advantages of a Tensor Algebra Compiler

Traditional Libraries

Choose a few expressions

Choose a few formats

Spend years hand optimizing kernels

CSR
CSC

The Tensor Algebra Compiler (taco)

Any expression

Many formats

Compiler produces optimized code

DCSR DCSR
BCSC DCSC
CSR BCSR
CSB CSC Sparse Vector BCSC
Dense Tensor Sparse Tensors

Advantages of a Tensor Algebra Compiler

Traditional Libraries

Choose a few expressions

Choose a few formats

Spend years hand optimizing kernels

The Tensor Algebra Compiler (taco)

Any expression

Many formats

Compiler produces optimized code

Advantages of a Tensor Algebra Compiler

Traditional Libraries

Choose a few expressions

Choose a few formats

Spend years hand optimizing kernels

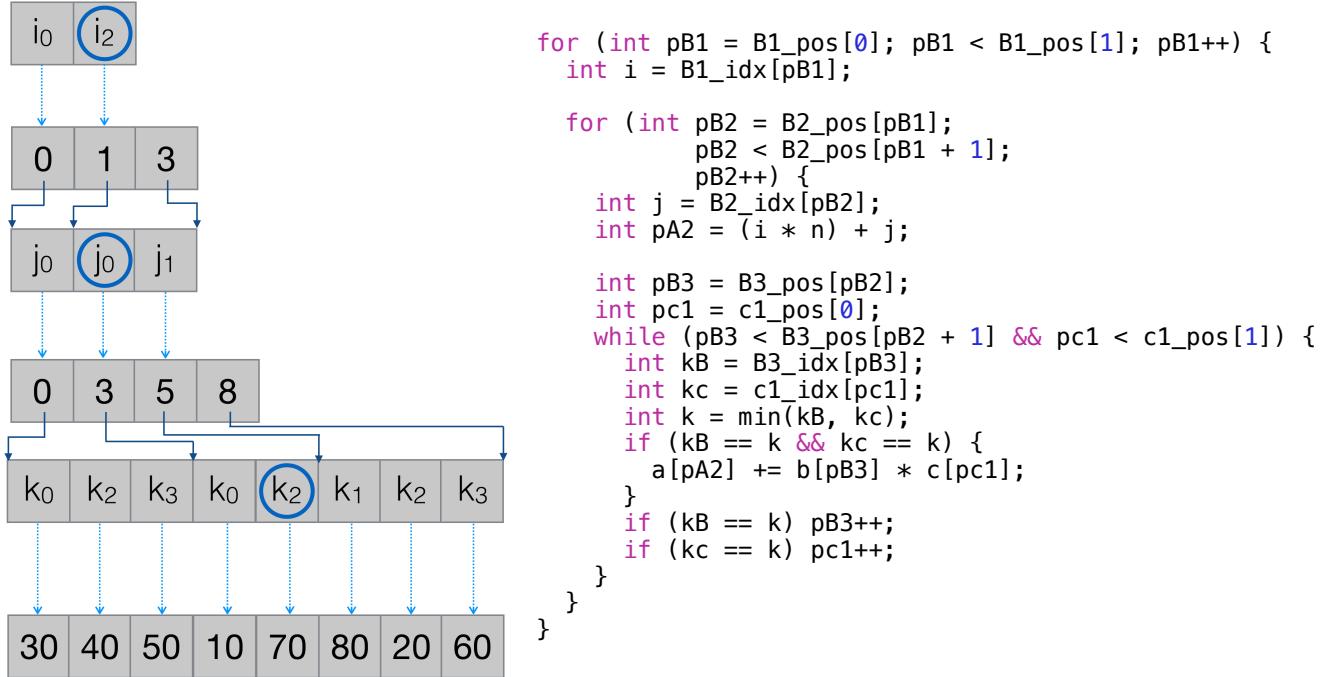
The Tensor Algebra Compiler (taco)

Any expression

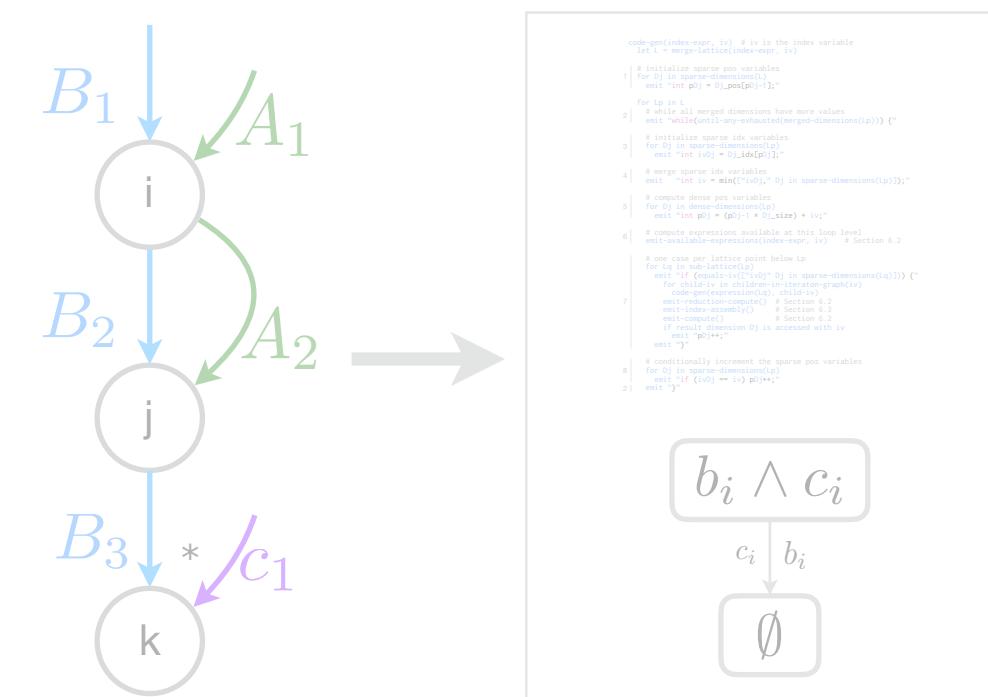
Many formats

Compiler produces optimized code

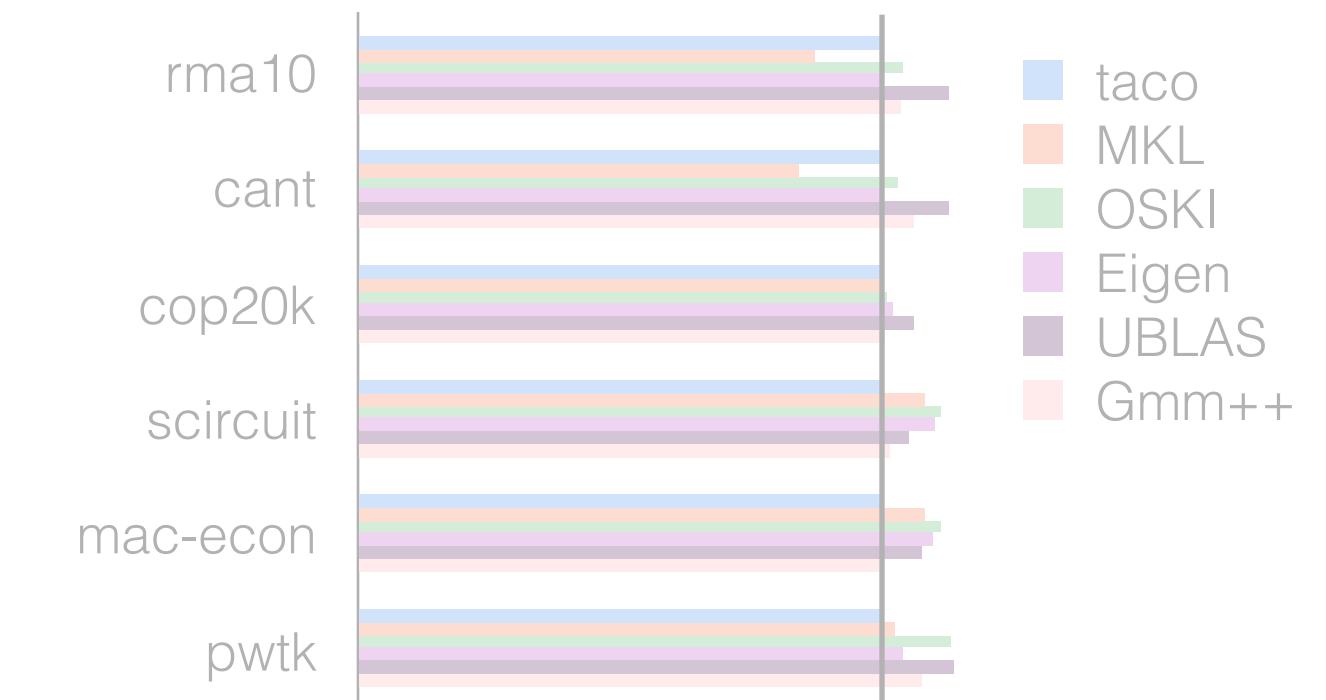
Sparse code and data



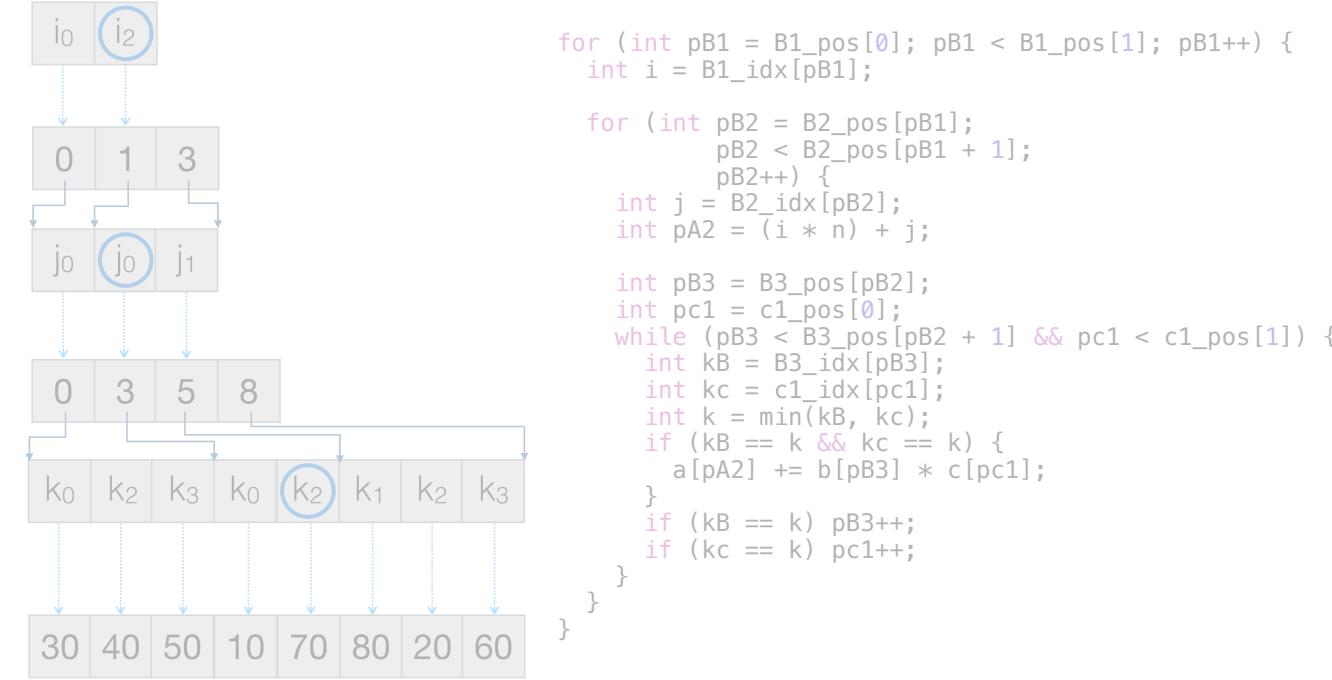
Intermediate Representation and Code Generation



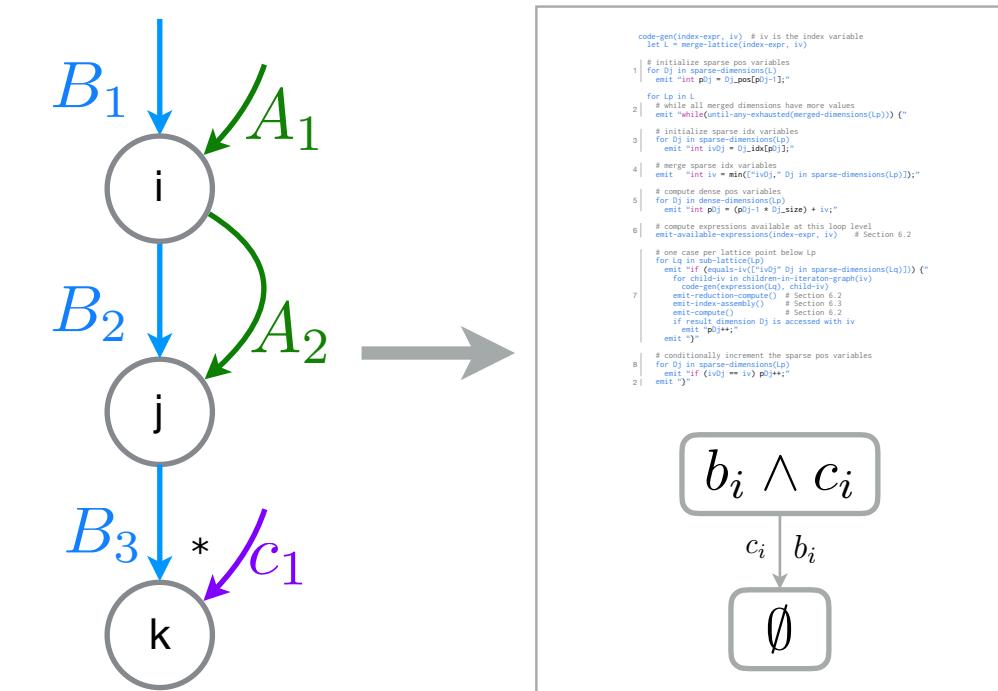
Evaluation



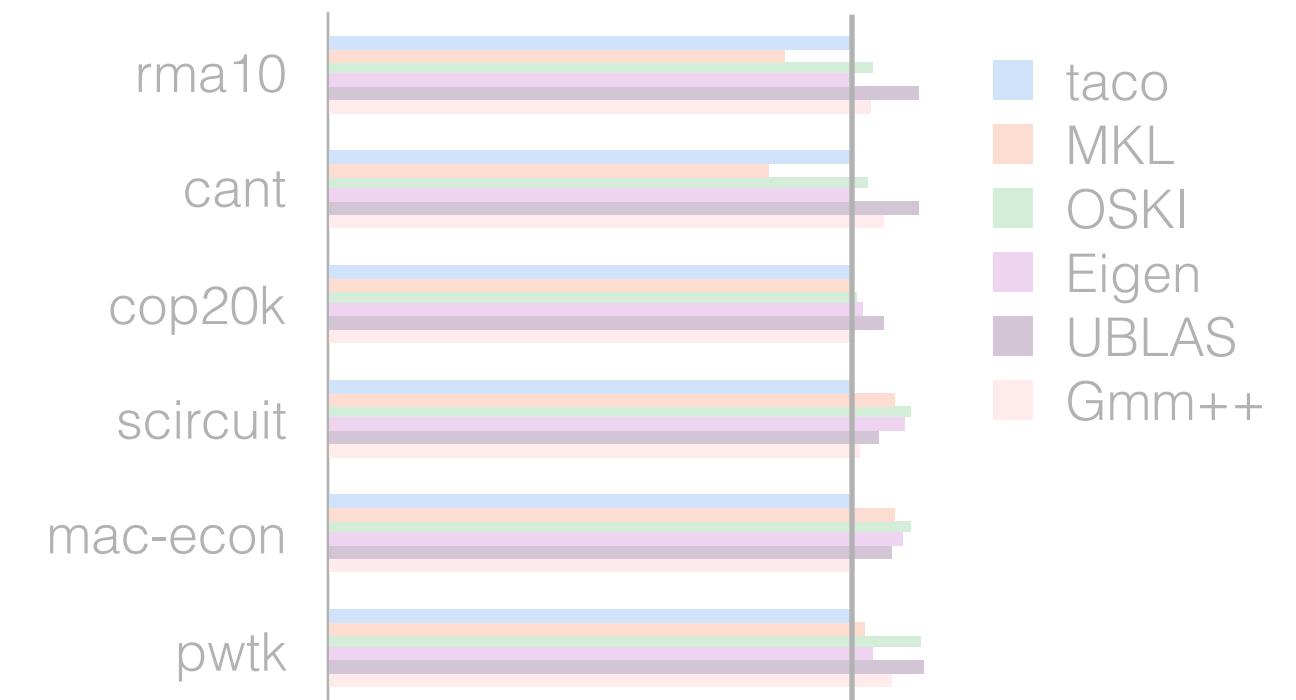
Sparse code and data



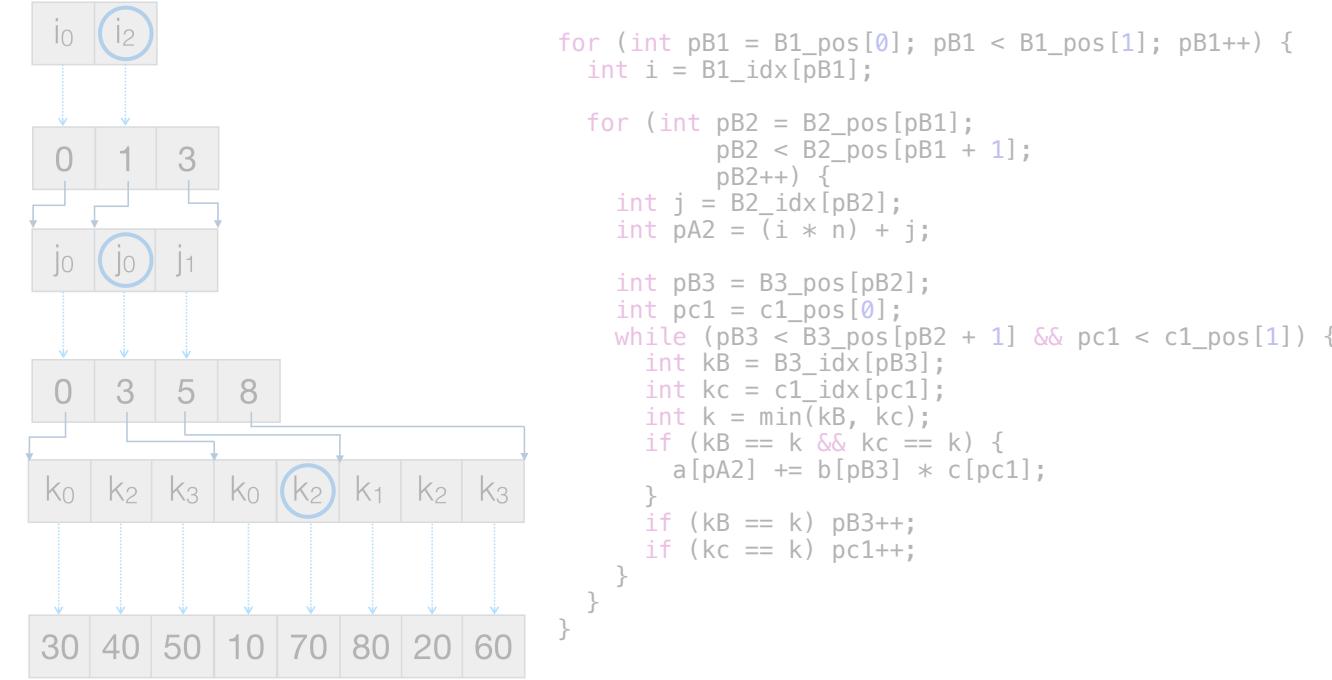
Intermediate Representation and Code Generation



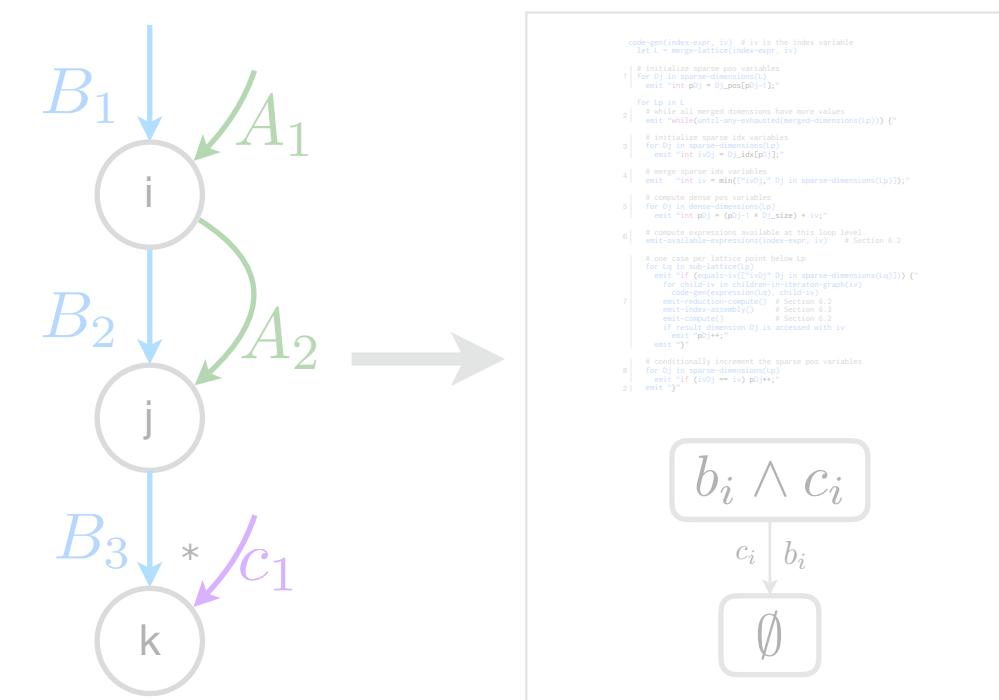
Evaluation



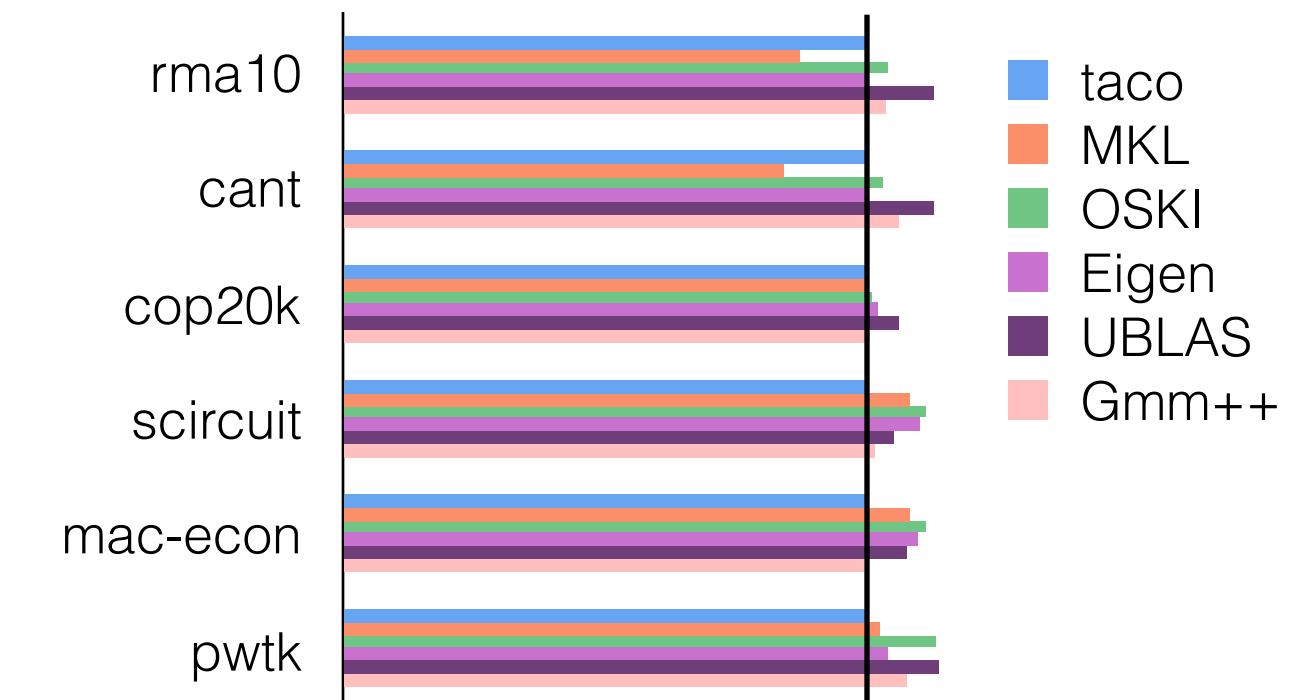
Sparse code and data



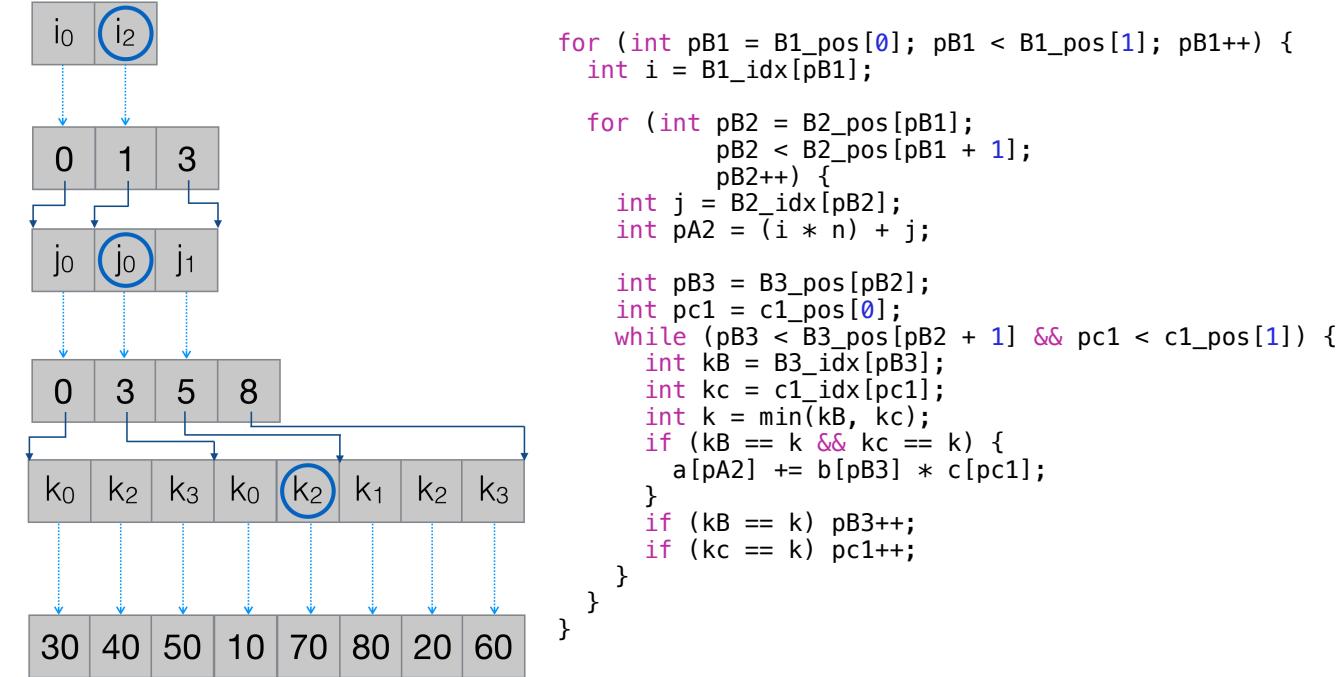
Intermediate Representation and Code Generation



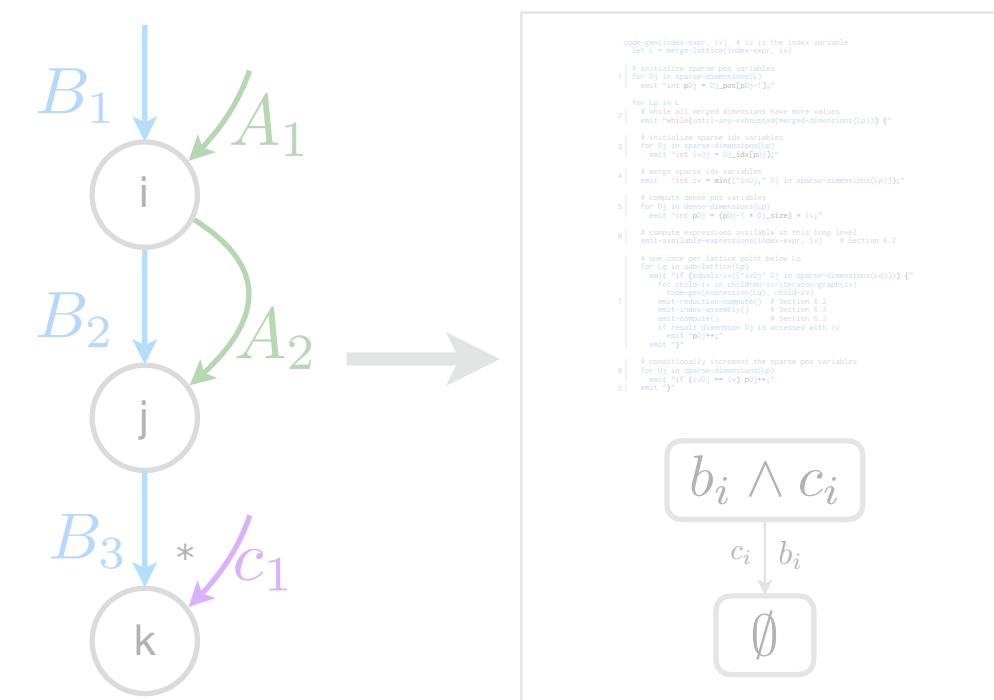
Evaluation



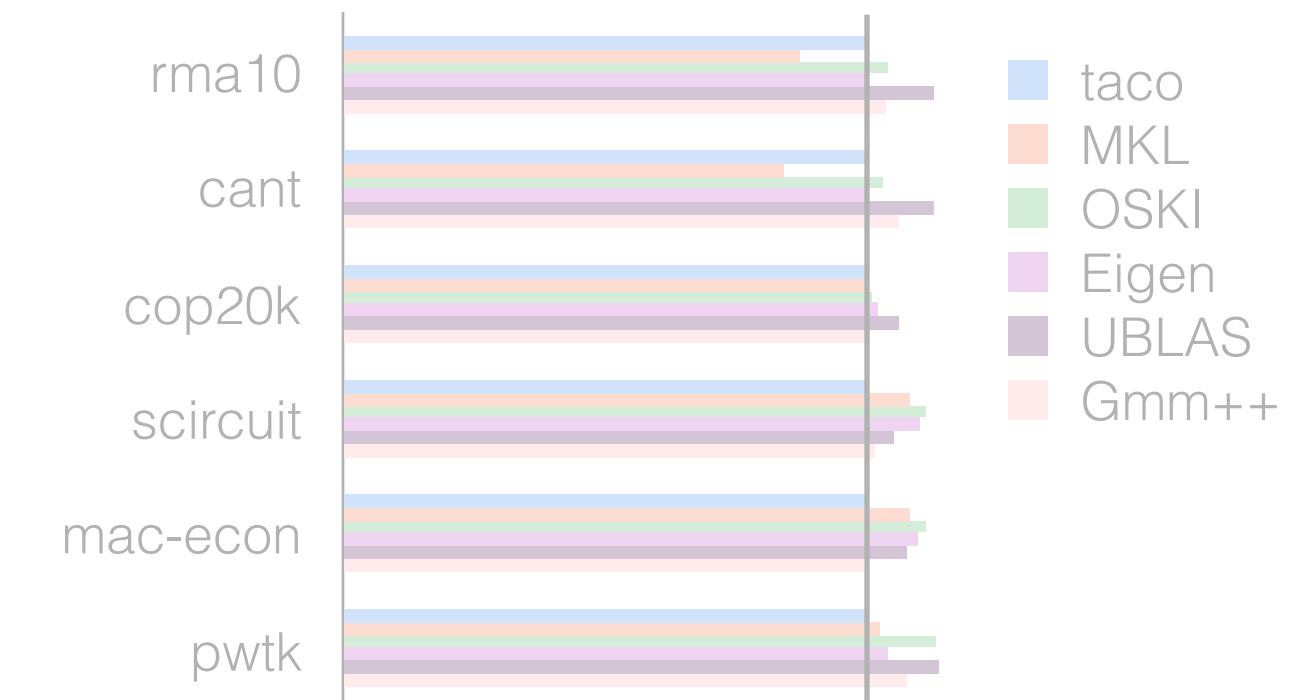
Sparse code and data



Intermediate Representation and Code Generation



Evaluation



Tensor-vector multiplication example

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

Tensor-vector multiplication example

Free variables i, j

$$A_{\underline{i} \underline{j}} = \sum_k B_{\underline{i} \underline{j} k} * c_k$$

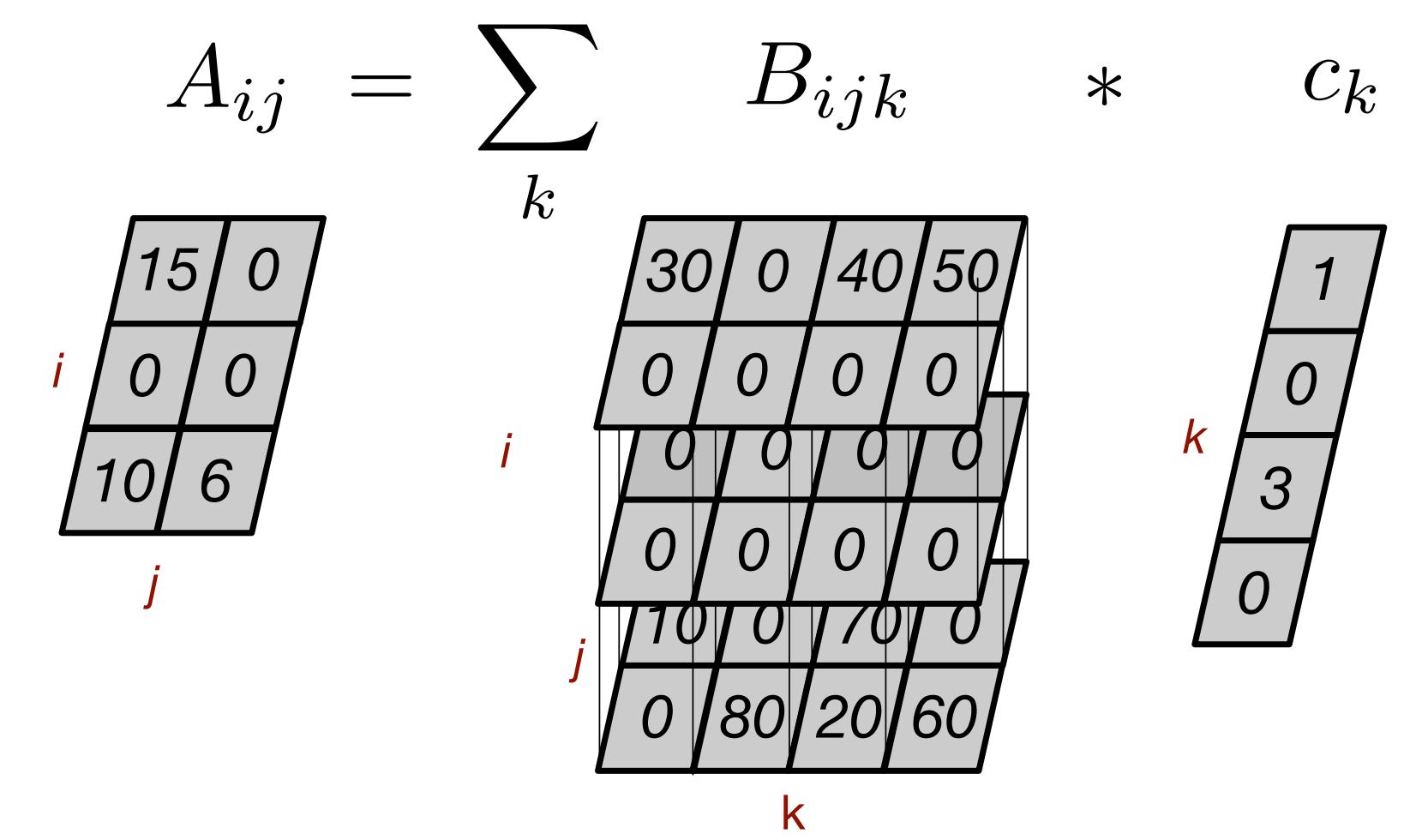
Tensor-vector multiplication example

Summation variable k

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

Tensor-vector multiplication example

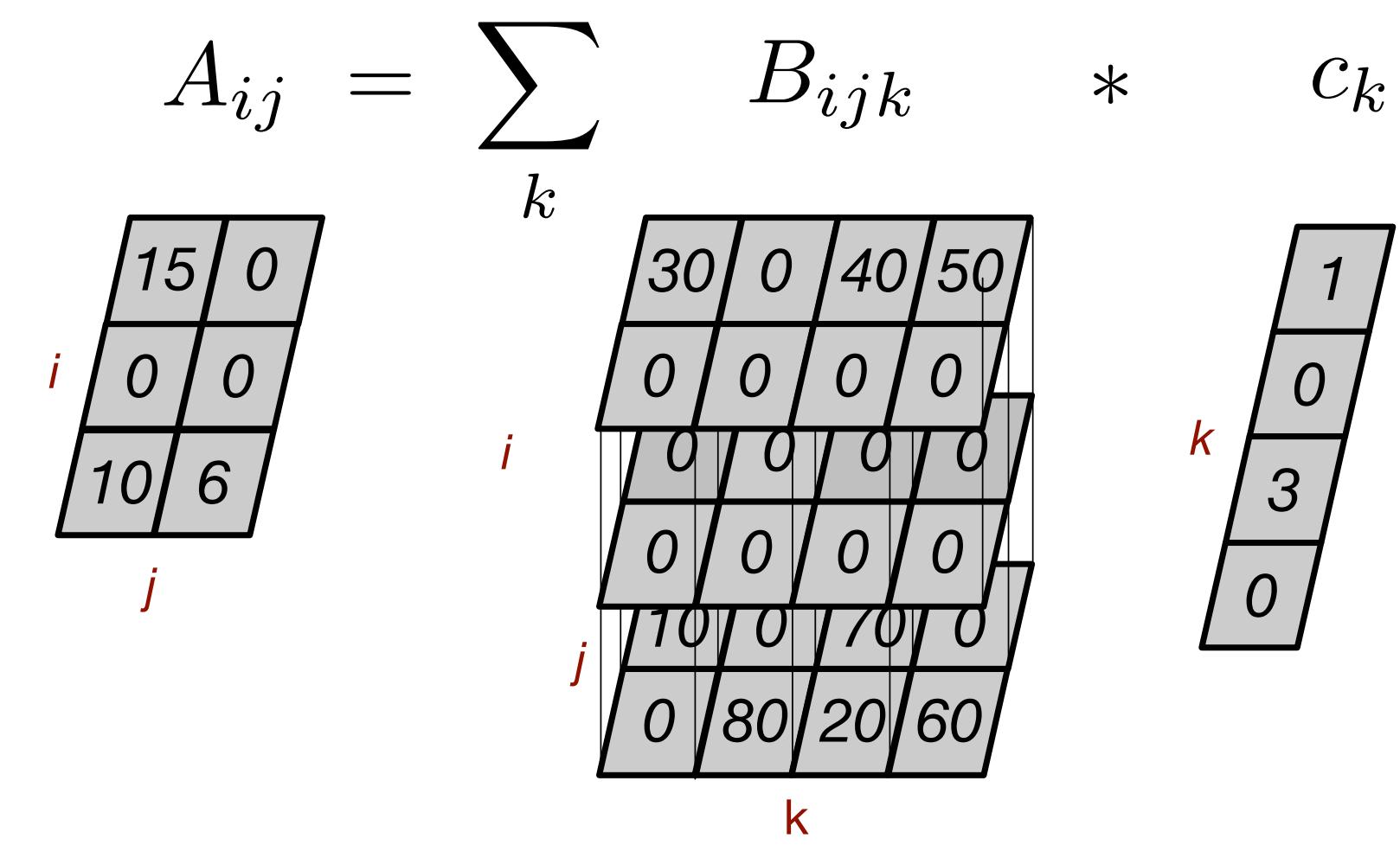
$$A_{ij} = \sum_k B_{ijk} * c_k$$


The diagram illustrates the computation of A_{ij} as a sum of products. It shows three components: a 2x2 matrix A_{ij} , a 4x4 matrix B_{ijk} , and a 4x1 vector c_k . The matrix A_{ij} has elements 15, 0, 0, 10 and 0, 6. The matrix B_{ijk} has elements 30, 0, 40, 50; 0, 0, 0, 0; 0, 0, 0, 0; and 10, 0, 70, 0. The vector c_k has elements 1, 0, 3, 0. Red indices i , j , and k are placed above the first row of A_{ij} , the first column of B_{ijk} , and the first element of c_k respectively. The multiplication is shown as $A_{ij} * B_{ijk} * c_k$.

(dense, dense) (dense,dense,dense) (dense)

Tensor-vector multiplication example

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

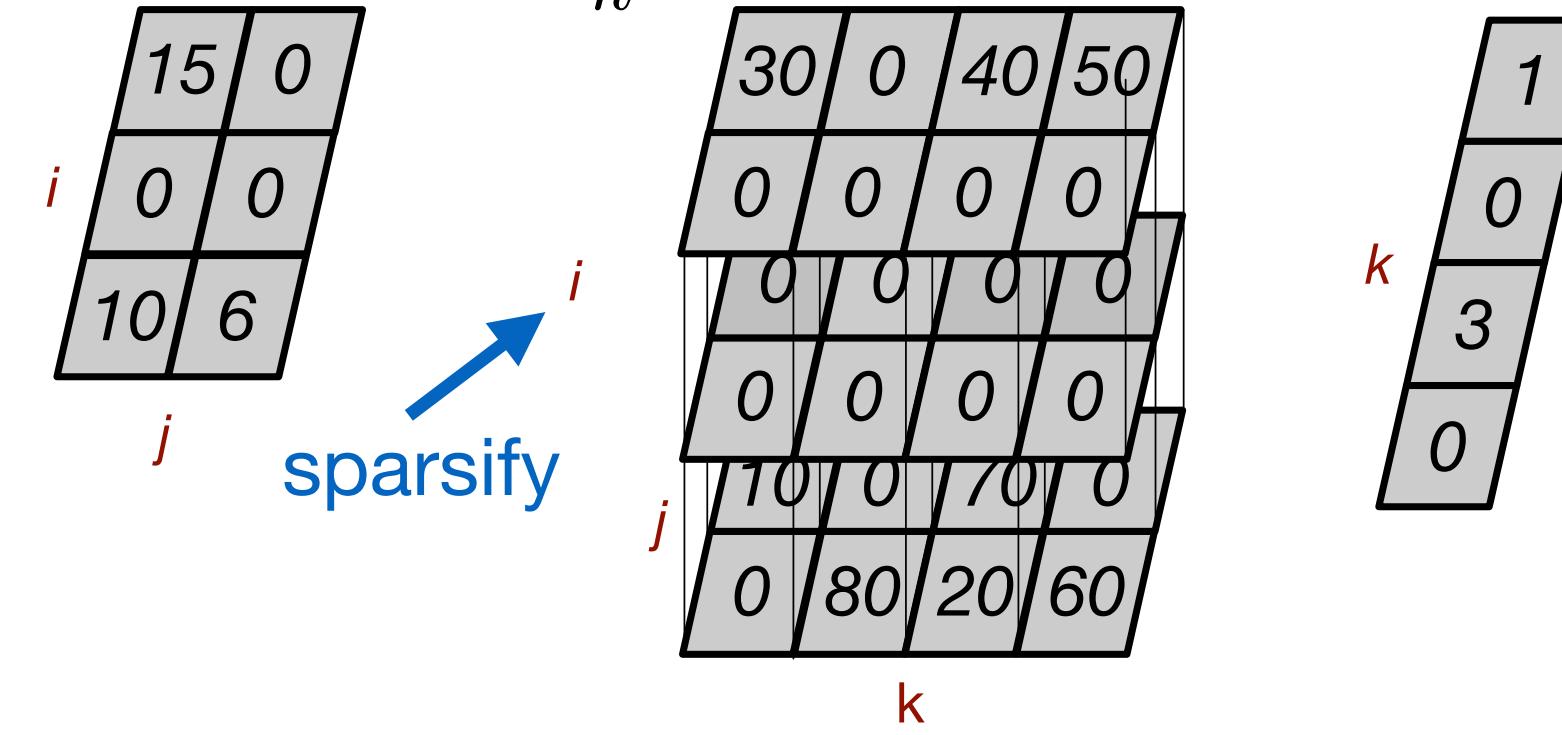

A 2×2 matrix A_{ij} with elements $\begin{matrix} 15 & 0 \\ 0 & 0 \\ 10 & 6 \end{matrix}$.
A 4×4 matrix B_{ijk} with elements $\begin{matrix} 30 & 0 & 40 & 50 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 10 & 0 & 70 & 0 \\ 0 & 80 & 20 & 60 \end{matrix}$.
A column vector c_k with elements $\begin{matrix} 1 \\ 0 \\ 3 \\ 0 \end{matrix}$.

(dense, dense) (dense,dense,dense) (dense)

```
for (int i = 0; i < m; i++) {  
  
    for (int j = 0; j < n; j++) {  
        int pB2 = (i * n) + j;  
        int pA2 = (i * n) + j;  
  
        for (int k = 0; k < p; k++) {  
            int pB3 = (pB2 * p) + k;  
  
            a[pA2] += b[pB3] * c[k];  
  
        }  
    }  
}
```

Tensor-vector multiplication example

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


(dense, dense) (dense,dense,dense) (dense)

```
for (int i = 0; i < m; i++) {  
    for (int j = 0; j < n; j++) {  
        int pB2 = (i * n) + j;  
        int pA2 = (i * n) + j;  
  
        for (int k = 0; k < p; k++) {  
            int pB3 = (pB2 * p) + k;  
  
            a[pA2] += b[pB3] * c[k];  
        }  
    }  
}
```

Tensor-vector multiplication example

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

(dense, dense) (sparse,dense,dense) (dense)

The diagram shows three tensors and a vector:

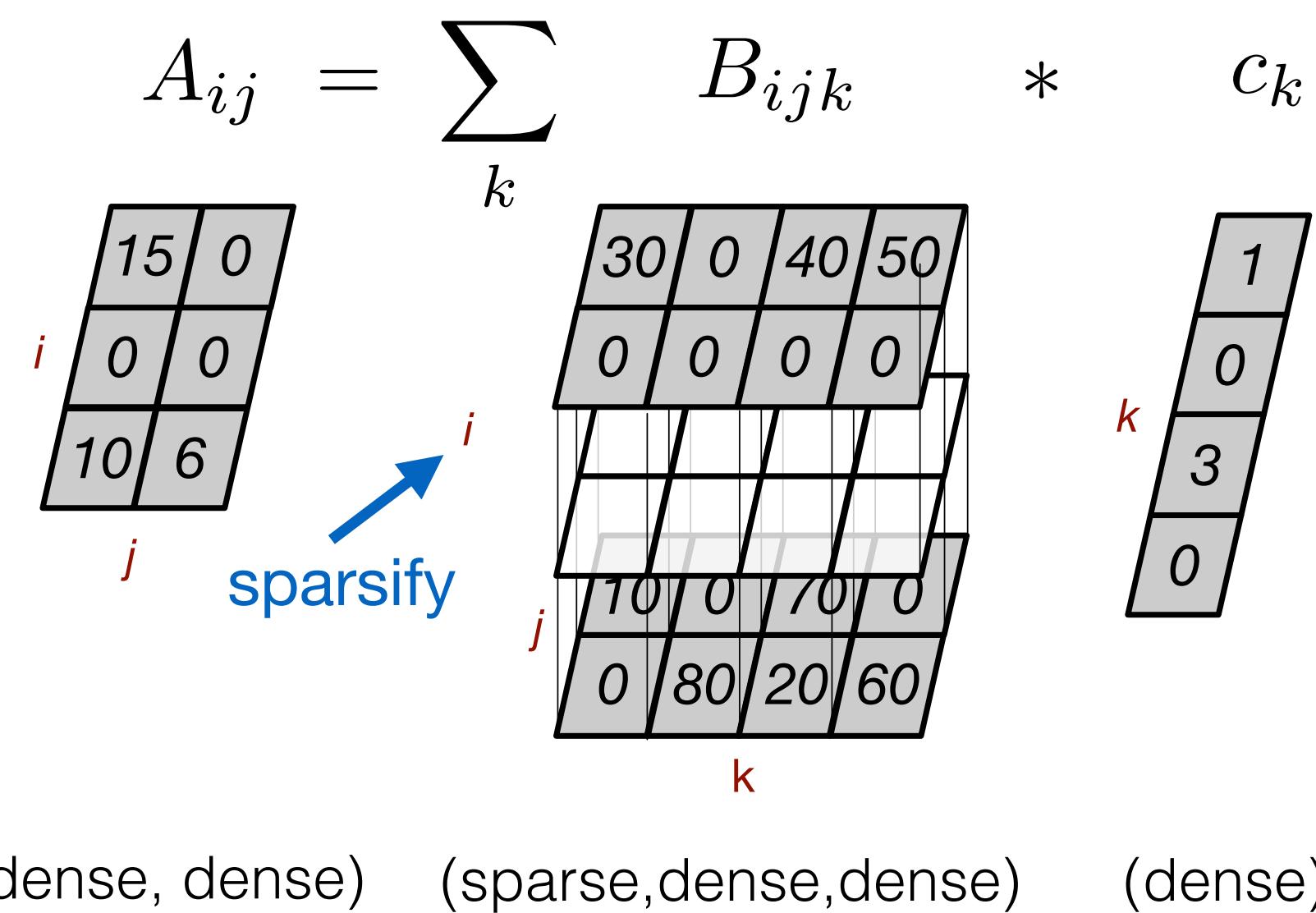
- A 2×2 matrix A with elements $15, 0, 0, 6$. It is labeled i (row) and j (column).
- A $4 \times 4 \times 4$ tensor B with elements $30, 0, 40, 50$ in the first row of the first slice. It is labeled i (row), j (column), and k (depth).
- A 4×1 vector c with elements $1, 0, 3, 0$.

A blue arrow labeled "sparsify" points from the matrix A to the tensor B , indicating that the matrix A is being converted into a sparse representation for the multiplication.

```
for (int i = 0; i < m; i++) {  
  
    for (int j = 0; j < n; j++) {  
        int pB2 = (i * n) + j;  
        int pA2 = (i * n) + j;  
  
        for (int k = 0; k < p; k++) {  
            int pB3 = (pB2 * p) + k;  
  
            a[pA2] += b[pB3] * c[k];  
  
        }  
    }  
}
```

Tensor-vector multiplication example

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

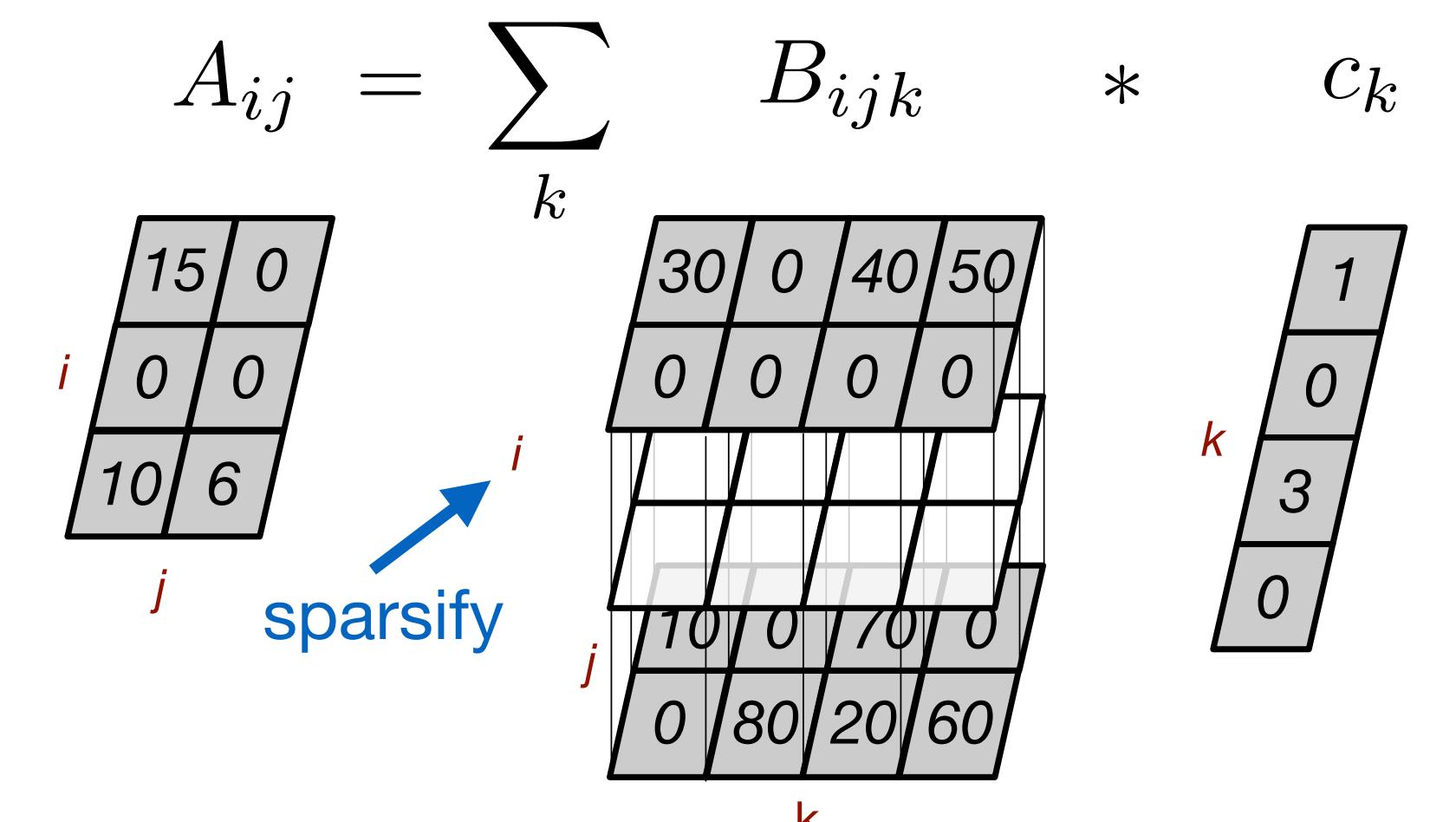

(dense, dense) (sparse,dense,dense) (dense)

i **j** **k**

```
for (int i = 0; i < m; i++) {  
    for (int j = 0; j < n; j++) {  
        int pB2 = (i * n) + j;  
        int pA2 = (i * n) + j;  
  
        for (int k = 0; k < p; k++) {  
            int pB3 = (pB2 * p) + k;  
  
            a[pA2] += b[pB3] * c[k];  
        }  
    }  
}
```

Tensor-vector multiplication example

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


(dense, dense) (sparse,dense,dense) (dense)

```
for (int pB1 = B1_pos[0]; pB1 < B1_pos[1]; pB1++) {  
    int i = B1_idx[pB1];  
  
    for (int j = 0; j < n; j++) {  
        int pB2 = (pB1 * n) + j;  
        int pA2 = (i * n) + j;  
  
        for (int k = 0; k < p; k++) {  
            int pB3 = (pB2 * p) + k;  
  
            a[pA2] += b[pB3] * c[k];  
        }  
    }  
}
```

Tensor-vector multiplication example

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

The diagram illustrates the computation of a matrix element A_{ij} as a weighted sum of elements from a sparse matrix B_{ijk} and a vector c_k . The matrices are labeled with dimensions i , j , and k in red. A blue arrow points from the matrix A_{ij} to the matrix B_{ijk} .

A_{ij}	B_{ijk}	c_k
$\begin{matrix} 15 & 0 \\ 0 & 0 \\ 10 & 6 \end{matrix}$	$\begin{matrix} 30 & 0 & 40 & 50 \\ 0 & 0 & 0 & 0 \\ 10 & 0 & 70 & 0 \\ 0 & 80 & 20 & 60 \end{matrix}$	$\begin{matrix} 1 \\ 0 \\ 3 \\ 0 \end{matrix}$

(dense, dense) (sparse,dense,dense) (dense)

```
for (int pB1 = B1_pos[0]; pB1 < B1_pos[1]; pB1++)  
    int i = B1_idx[pB1];  
  
    for (int j = 0; j < n; j++) {  
        int pB2 = (pB1 * n) + j;  
        int pA2 = (i * n) + j;  
  
        for (int k = 0; k < p; k++) {  
            int pB3 = (pB2 * p) + k;  
  
            a[pA2] += b[pB3] * c[k];  
        }  
    }  
}
```

Tensor-vector multiplication example

The diagram illustrates the computation of $A_{ij} = \sum_k B_{ijk} * c_k$ for three different memory access patterns:

- (dense, dense)**: The first pattern shows a 3x2 matrix A with values [15, 0; 0, 0; 10, 6] and a 4x4 matrix B with values [30, 0, 40, 50; 10, 0, 70, 0; 0, 80, 20, 60]. A blue arrow points from the bottom-left cell of A to the bottom-left cell of B , labeled j .
- (sparse, sparse, dense)**: The second pattern shows a 3x2 matrix A and a 4x4 matrix B . Matrix B has non-zero elements at indices (0,0), (0,1), (1,0), (1,1), (2,0), (2,1), (3,0), (3,1), (3,2), and (3,3). A red arrow points from the bottom-left cell of A to the bottom-left cell of B , labeled j .
- (dense)**: The third pattern shows a 3x2 matrix A and a vector c with values [1, 0, 3, 0]. A red arrow points from the bottom-left cell of A to the value 1 in c , labeled k .

```
for (int pB1 = B1_pos[0]; pB1 < B1_pos[1]; pB1++) {  
    int i = B1_idx[pB1];  
  
    for (int pB2 = B2_pos[pB1]; pB2 < B2_pos[pB1 + 1]; pB2++) {  
        int j = B2_idx[pB2];  
        int pA2 = (i * n) + j;  
  
        for (int k = 0; k < p; k++) {  
            int pB3 = (pB2 * p) + k;  
  
            a[pA2] += b[pB3] * c[k];  
        }  
    }  
}
```

Tensor-vector multiplication example

The diagram illustrates the multiplication of three tensors:

- (dense, dense) A_{ij}** : A 3x2 matrix with values [15, 0], [0, 0], and [10, 6].
- (sparse, sparse, dense) B_{ijk}** : A 3x4x4 tensor with slices showing non-zero elements at indices (i,j,k) = (0,0,0), (0,1,0), (0,2,0), (0,3,0), (1,0,0), (1,1,0), (1,2,0), (1,3,0), (2,0,0), (2,1,0), (2,2,0), (2,3,0).
- (dense) c_k** : A 4x1 vector with values [1, 0, 3, 0].

The result of the multiplication is a sparse tensor where only non-zero elements are shown. A blue arrow points to the index k in the third tensor, indicating the dimension being summed over.

```
for (int pB1 = B1_pos[0]; pB1 < B1_pos[1]; pB1++) {  
    int i = B1_idx[pB1];  
  
    for (int pB2 = B2_pos[pB1]; pB2 < B2_pos[pB1 + 1]; pB2++) {  
        int j = B2_idx[pB2];  
        int pA2 = (i * n) + j;  
  
        for (int k = 0; k < p; k++) {  
            int pB3 = (pB2 * p) + k;  
  
            a[pA2] += b[pB3] * c[k];  
        }  
    }  
}
```

Tensor-vector multiplication example

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

(dense, dense) (sparse, sparse, sparse) (dense)

The diagram shows three matrices: A (2x2), B (3x3), and c (4x1). Matrix A has values 15, 0, 0, 10 at indices (0,0), (0,1), (1,0), (1,1) respectively. Matrix B has values 30, 40, 50, 10, 70, 80, 20, 60 at indices (0,0), (0,1), (0,2), (1,0), (1,1), (1,2), (2,0), (2,1) respectively. Vector c has values 1, 0, 3, 0 at indices (0,0), (1,0), (2,0), (3,0) respectively. Red arrows point from the indices i and j of matrix A to the corresponding rows and columns of matrix B. A blue arrow points from the index k of vector c to the corresponding element in matrix B.

```
for (int pB1 = B1_pos[0]; pB1 < B1_pos[1]; pB1++) {  
    int i = B1_idx[pB1];  
  
    for (int pB2 = B2_pos[pB1]; pB2 < B2_pos[pB1 + 1]; pB2++) {  
        int j = B2_idx[pB2];  
        int pA2 = (i * n) + j;  
  
        for (int pB3 = B3_pos[pB2]; pB3 < B3_pos[pB2 + 1]; pB3++) {  
            int k = B3_idx[pB3];  
  
            a[pA2] += b[pB3] * c[k];  
        }  
    }  
}
```

Tensor-vector multiplication example

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

The diagram shows three components: a dense matrix A , a sparse matrix B , and a dense vector c .
Matrix A is a 2x2 matrix with elements 15, 0, 0, 6.
Matrix B is a 2x3 matrix with non-zero elements 30, 40, 50, 10, 70, 80, 20, 60. The zero elements are represented by gray boxes.
Vector c is a 3x1 column vector with elements 1, 0, 3.

(dense, dense) (sparse,sparse,sparse) (dense)

```
for (int pB1 = B1_pos[0]; pB1 < B1_pos[1]; pB1++) {  
    int i = B1_idx[pB1];  
  
    for (int pB2 = B2_pos[pB1]; pB2 < B2_pos[pB1 + 1]; pB2++) {  
        int j = B2_idx[pB2];  
        int pA2 = (i * n) + j;  
  
        for (int pB3 = B3_pos[pB2]; pB3 < B3_pos[pB2 + 1]; pB3++) {  
            int k = B3_idx[pB3];  
  
            a[pA2] += b[pB3] * c[k];  
        }  
    }  
}
```

Tensor-vector multiplication example

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

The diagram shows three components: a dense matrix A_{ij} (2x2), a sparse matrix B_{ijk} (3x3), and a sparse vector c_k (3). The matrix A_{ij} has values 15, 0, 0, 10, 6. The matrix B_{ijk} has non-zero values 30, 40, 50, 10, 70, 80, 20, 60. The vector c_k has values 1, 3. Red indices i , j , and k are shown above each component. Black lines connect the non-zero elements of B_{ijk} to the corresponding elements in A_{ij} . A blue arrow points from the value 1 in c_k to the value 1 in the vector.

(dense, dense) (sparse,sparse,sparse) (sparse)

```
for (int pB1 = B1_pos[0]; pB1 < B1_pos[1]; pB1++) {  
    int i = B1_idx[pB1];  
  
    for (int pB2 = B2_pos[pB1]; pB2 < B2_pos[pB1 + 1]; pB2++) {  
        int j = B2_idx[pB2];  
        int pA2 = (i * n) + j;  
  
        for (int pB3 = B3_pos[pB2]; pB3 < B3_pos[pB2 + 1]; pB3++) {  
            int k = B3_idx[pB3];  
  
            a[pA2] += b[pB3] * c[k];  
        }  
    }  
}
```

Tensor-vector multiplication example

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

(dense, dense) (sparse, sparse, sparse) (sparse)

The diagram shows three components: a 2x2 matrix A with values 15, 0, 0, 6; a 3x3x3 tensor B with values 30, 40, 50, 10, 70, 80, 20, 60; and a 3x3 matrix C with values 1, 3. Red indices i and j are shown for A and B , while red index k is shown for C . Black arrows indicate the summation of specific elements from B for each element in C .

```
for (int pB1 = B1_pos[0]; pB1 < B1_pos[1]; pB1++) {  
    int i = B1_idx[pB1];  
  
    for (int pB2 = B2_pos[pB1]; pB2 < B2_pos[pB1 + 1]; pB2++) {  
        int j = B2_idx[pB2];  
        int pA2 = (i * n) + j;  
  
        for (int pB3 = B3_pos[pB2]; pB3 < B3_pos[pB2 + 1]; pB3++) {  
            int k = B3_idx[pB3];  
  
            a[pA2] += b[pB3] * c[k];  
        }  
    }  
}
```

Tensor-vector multiplication example

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

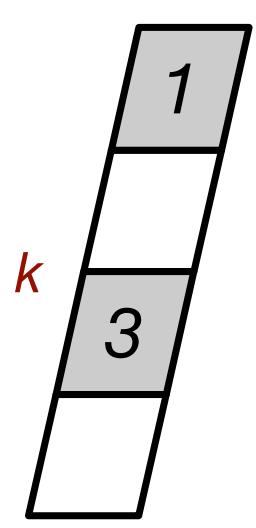
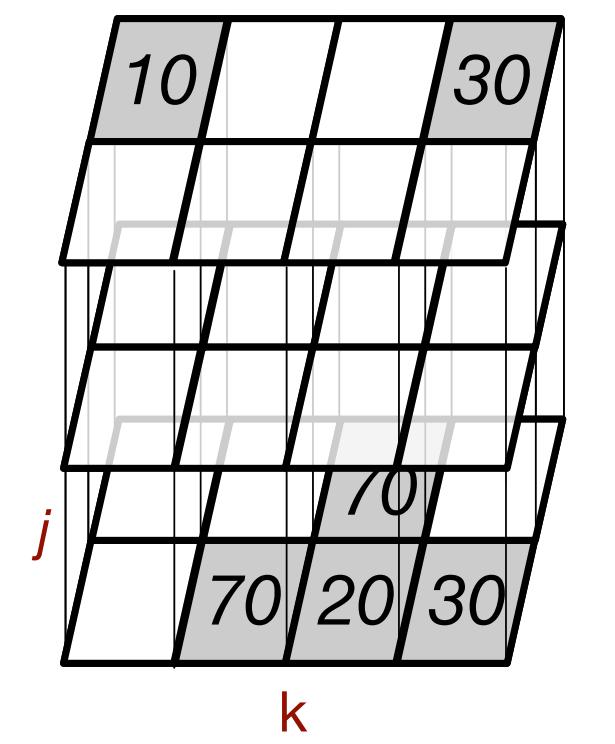
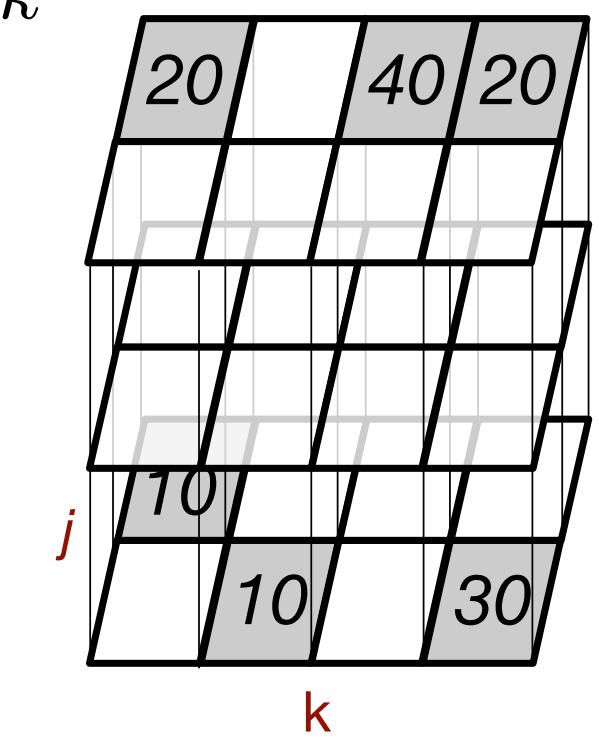
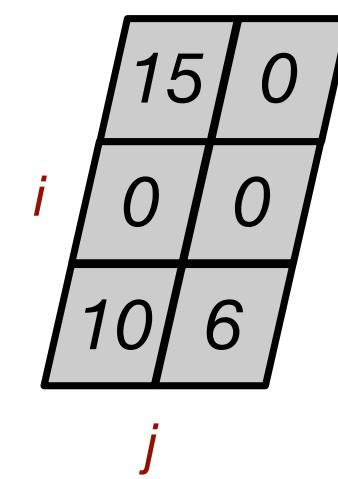
(dense, dense) (sparse,sparse,sparse) (sparse)

The diagram shows three components: a 2x2 matrix A with values 15, 0, 0, 6; a sparse matrix B with non-zero elements 30, 40, 50, 10, 70, 80, 20, 60; and a sparse vector c with values 1, 3. Red indices i and j are shown for A and B , while black index k is shown for B and c . A blue arrow points from the k index of c to the k index of B .

```
for (int pB1 = B1_pos[0]; pB1 < B1_pos[1]; pB1++) {  
    int i = B1_idx[pB1];  
  
    for (int pB2 = B2_pos[pB1]; pB2 < B2_pos[pB1 + 1]; pB2++) {  
        int j = B2_idx[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_idx[pB3];  
            int kc = c1_idx[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++;  
        }  
    }  
}
```

Tensor-vector multiplication example

$$A_{ij} = \sum_k (\underbrace{B_{ijk} + D_{ijk}}_k) * c_k$$



(dense, dense)

(sparse,sparse,sparse)

(sparse,sparse,sparse)

(sparse)

Tensor-vector multiplication example

$$A_{ij} = \sum_k (B_{ijk} + D_{ijk}) * c_k$$

(dense, dense)

(sparse, sparse, sparse)

(sparse, sparse, sparse)

```

int pB1 = B1_pos[0];
int pD1 = D1_pos[0];
while (pB1 < B1_pos[1] &&
       pD1 < D1_pos[1]) {
    int iB = B1_idx[pB1];
    int iD = D1_idx[pD1];
    int i = min(iB, iD);
    if (iB == i && iD == i) {
        int pB2 = B2_pos[pB1];
        int pD2 = D2_pos[pD1];
        while (pB2 < B2_pos[pB1 + 1] &&
               pD2 < D2_pos[pD1 + 1]) {
            int jB = B2_idx[pB2];
            int jD = D2_idx[pD2];
            int j = min(jB, jD);
            int pA2 = (i * A2_dimension) + j;
            if (jB == j && jD == j) {
                double tk = 0.0;
                int pB3 = B3_pos[pB2];
                int pD3 = D3_pos[pD2];
                int pc1 = c1_pos[0];
                while (pB3 < B3_pos[pB2 + 1] &&
                       pD3 < D3_pos[pD2 + 1] &&
                       pc1 < c1_pos[1]) {
                    int kB = B3_idx[pB3];
                    int kD = D3_idx[pD3];
                    int kc = c1_idx[pc1];
                    int k = min(kB, kB);
                    if (kB == k && kD == k && kc == k) {
                        tk += (b[pB3] + d[pD3]) * c[pc1];
                    } else if (kB == k && kc == k) {
                        tk += b[pB3] * c[pc1];
                    } else if (kD == k && kc == k) {
                        tk += d[pD3] * c[pc1];
                    }
                    if (kB == k) pB3++;
                    if (kD == k) pD3++;
                    if (kc == k) pc1++;
                }
                while (pB3 < B3_pos[pB2 + 1] &&
                       pc1 < c1_pos[1]) {
                    int kB0 = B3_idx[pB3];
                    int kc0 = c1_idx[pc1];
                    int k0 = min(kB0, kc0);
                    if (kB0 == k0 && kc0 == k0) {
                        tk += b[pB3] * c[pc1];
                    }
                    if (kB0 == k0) pB3++;
                    if (kc0 == k0) pc1++;
                }
                while (pD3 < D3_pos[pD2 + 1] &&
                       pc1 < c1_pos[1]) {
                    int kD0 = D3_idx[pD3];
                    int kc1 = c1_idx[pc1];
                    int k1 = min(kD0, kc1);
                    if (kD0 == k1 && kc1 == k1) {
                        tk += d[pD3] * c[pc1];
                    }
                    if (kD0 == k1) pD3++;
                    if (kc1 == k1) pc1++;
                }
                a[pA2] = tk;
            }
            else if (jB == j) {
                double tk0 = 0.0;
                int pB30 = B3_pos[pB2];
                int pc10 = c1_pos[0];
                while (pB30 < B3_pos[pB2 + 1] &&
                      pc10 < c1_pos[1]) {
                    int kB1 = B3_idx[pB30];
                    int kc2 = c1_idx[pc10];
                    int k2 = min(kB1, kc2);
                    if (kB1 == k2 && kc2 == k2) {
                        tk0 += b[pB30] * c[pc10];
                    }
                    if (kB1 == k2) pB30++;
                    if (kc2 == k2) pc10++;
                }
                a[pA2] = tk0;
            }
            else {
                double tk1 = 0.0;
                int pD30 = D3_pos[pD2];
                int pc11 = c1_pos[0];
                while (pD30 < D3_pos[pD2 + 1] &&
                      pc11 < c1_pos[1]) {
                    int kD1 = D3_idx[pD30];
                    int kc3 = c1_idx[pc11];
                    int k3 = min(kD1, kc3);
                    if (kD1 == k3 && kc3 == k3) {
                        tk1 += d[pD30] * c[pc11];
                    }
                    if (kD1 == k3) pD30++;
                    if (kc3 == k3) pc11++;
                }
                a[pA2] = tk1;
            }
            if (jB == j) pB2++;
            if (jD == j) pD2++;
        }
    }
}

while (pB1 < B1_pos[1]) {
    int iB0 = B1_idx[pB1];
    for (int pB2 = B2_pos[pB1];
         pB2 < B2_pos[pB1 + 1]; pB2++) {
        int jB2 = B2_idx[pB2];
        int pA24 = (i * A2_dimension) + jB0;
        double tk6 = 0.0;
        int pB33 = B3_pos[pB2];
        int pc16 = c1_pos[0];
        while (pB33 < B3_pos[pB2 + 1] &&
               pc16 < c1_pos[1]) {
            int kB4 = B3_idx[pB33];
            int kc8 = c1_idx[pc16];
            int k8 = min(kB4, kc8);
            if (kB4 == k8 && kc8 == k8) {
                tk6 += b[pB33] * c[pc16];
            }
            if (kc8 == k8) pc16++;
        }
        a[pA24] = tk6;
    }
}
while (pD1 < D1_pos[1]) {
    int iD0 = D1_idx[pD1];
    for (int pD2 = D2_pos[pD1];
         pD2 < D2_pos[pD1 + 1]; pD2++) {
        int jD2 = D2_idx[pD2];
        int pA25 = (i * A2_dimension) + jD0;
        double tk7 = 0.0;
        int pD33 = D3_pos[pD2];
        int pc17 = c1_pos[0];
        while (pD33 < D3_pos[pD2 + 1] &&
               pc17 < c1_pos[1]) {
            int kD4 = D3_idx[pD33];
            int kc9 = c1_idx[pc17];
            int k9 = min(kD4, kc9);
            if (kD4 == k9 && kc9 == k9) {
                tk7 += d[pD33] * c[pc17];
            }
            if (kc9 == k9) pc17++;
        }
        a[pA25] = tk7;
    }
}

```

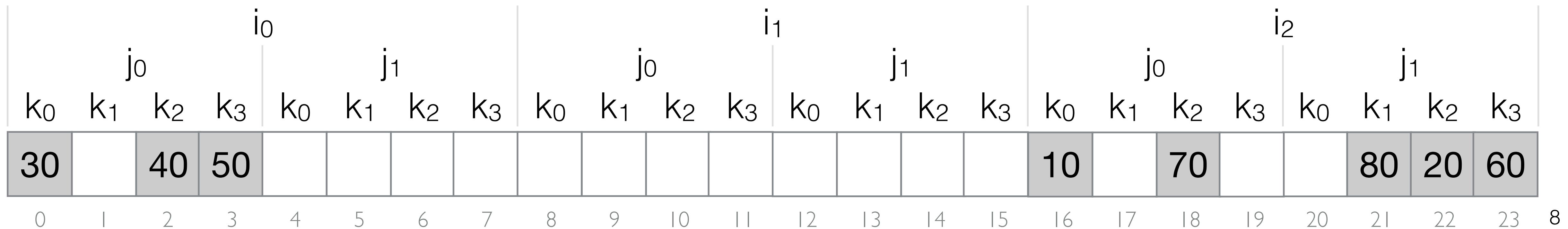
Dense storage formats

	k_0	k_1	k_2	k_3
i_0	30		40	50
j_0				
i_1				
j_1				

	k_0	k_1	k_2	k_3
i_1				
j_0				
i_2				
j_1				

	k_0	k_1	k_2	k_3
i_2	10		70	
j_0				
i_1				
j_1		80	20	60

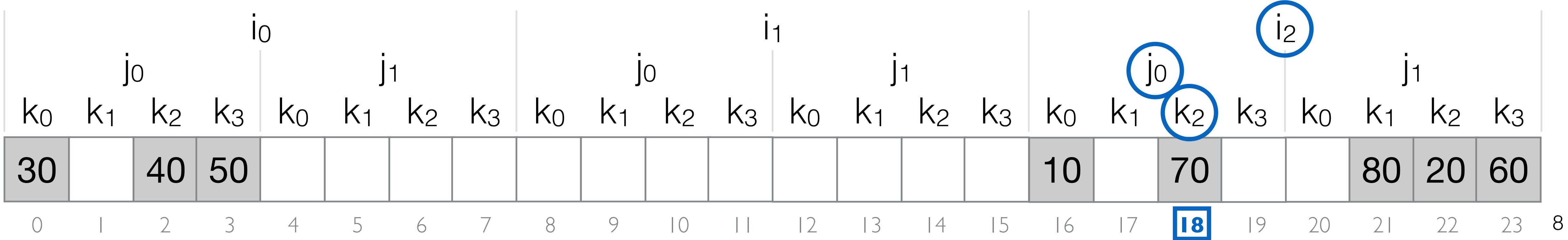
Dense storage formats



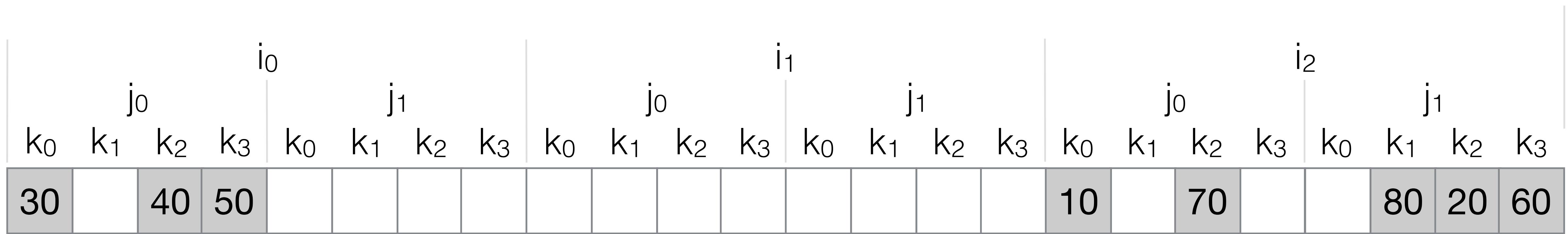
Dense storage formats

$$\text{locate } (2, 0, 2) \rightarrow i*2*4 + j*4 + k = 18$$

Random access: easy to locate element, but wasted storage



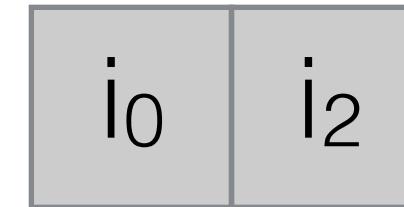
Compressed storage formats



Compressed storage formats

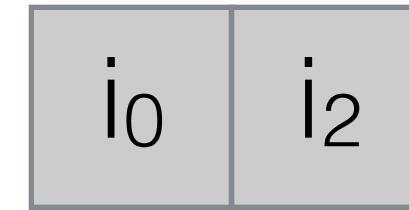
i_0	j_0	k_0	i_0	j_0	k_0	i_2	j_1	k_3
30	40	50	10	70	80	20	60	

Compressed storage formats



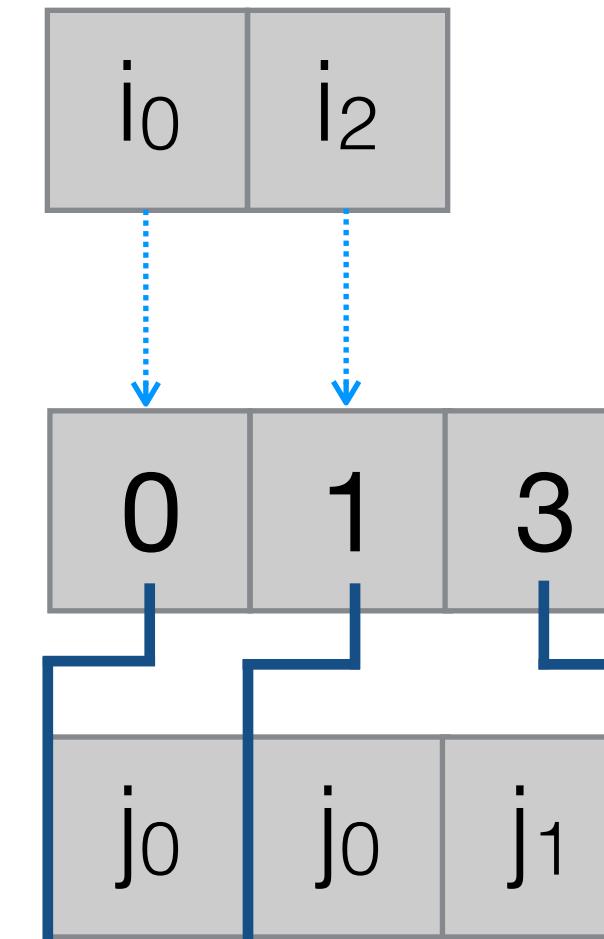
	j ₀			j ₀			j ₁		
	k ₀	k ₂	k ₃	k ₀	k ₂		k ₁	k ₂	k ₃
	30	40	50	10	70		80	20	60

Compressed storage formats



k ₀	k ₂	k ₃	k ₀	k ₂	k ₁	k ₂	k ₃
30	40	50	10	70	80	20	60

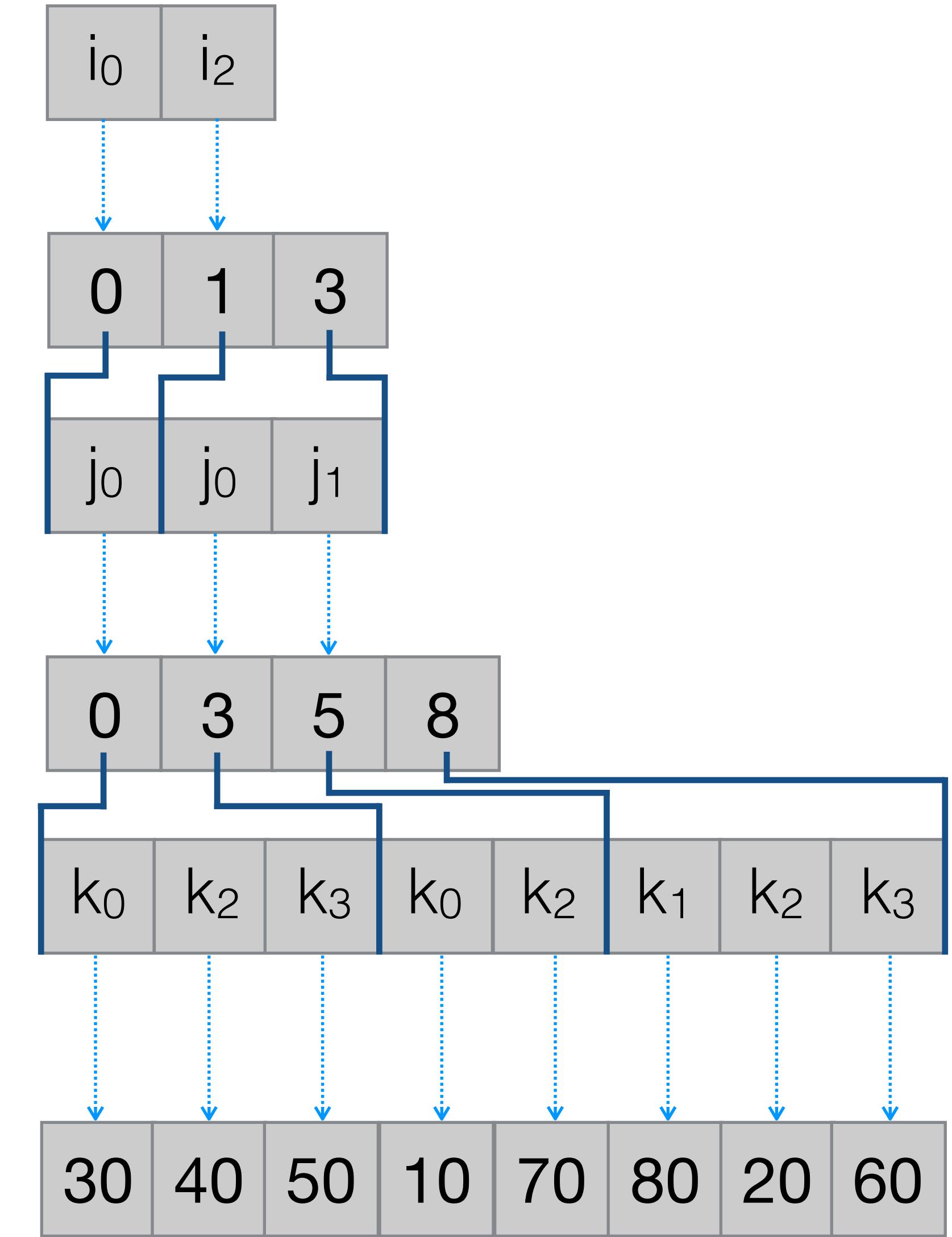
Compressed storage formats



k_0	k_2	k_3	k_0	k_2	k_1	k_2	k_3
30	40	50	10	70	80	20	60

Compressed storage formats

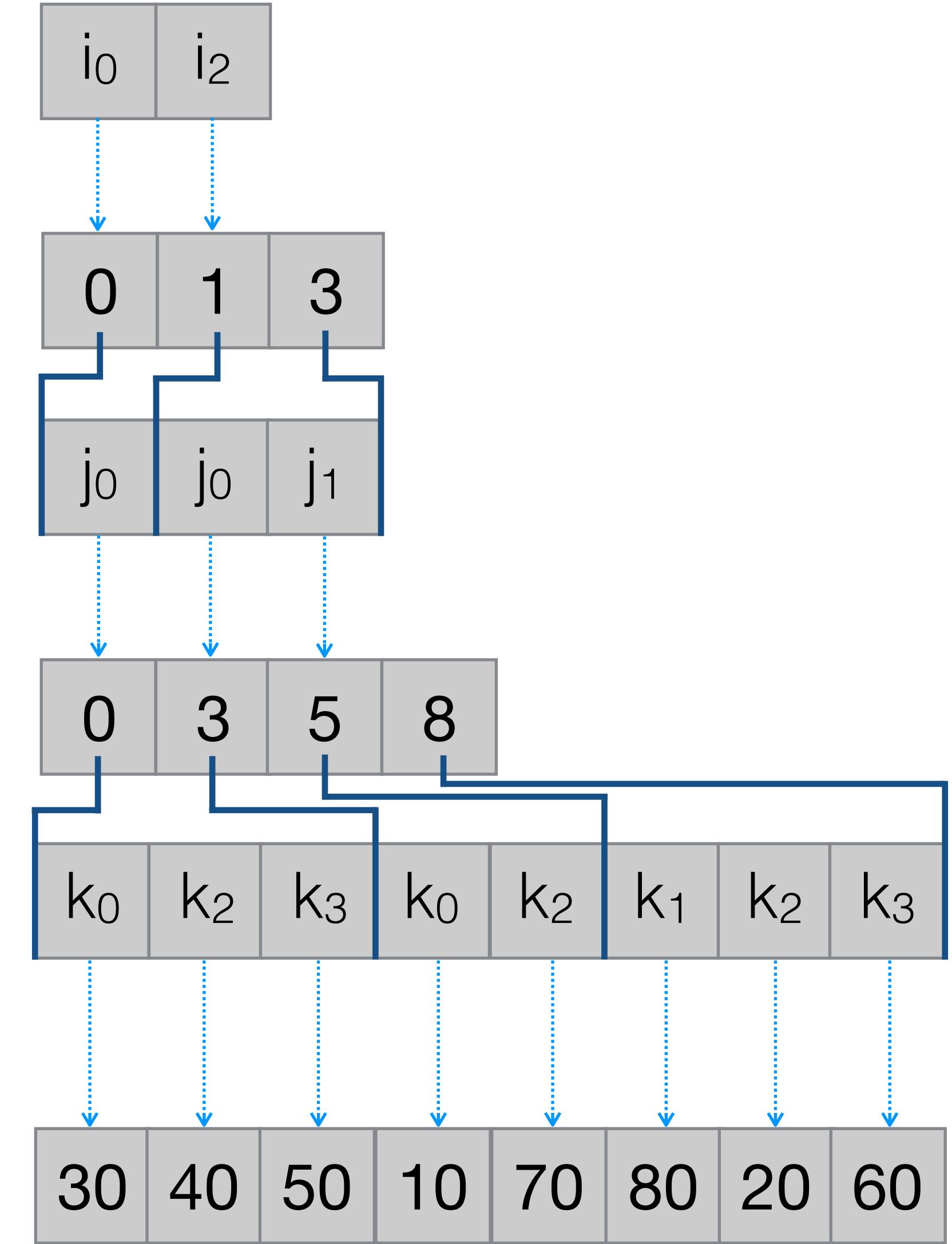
No random access: must traverse level by level to locate coordinate



Compressed storage formats

No random access: must traverse level by level to locate coordinate

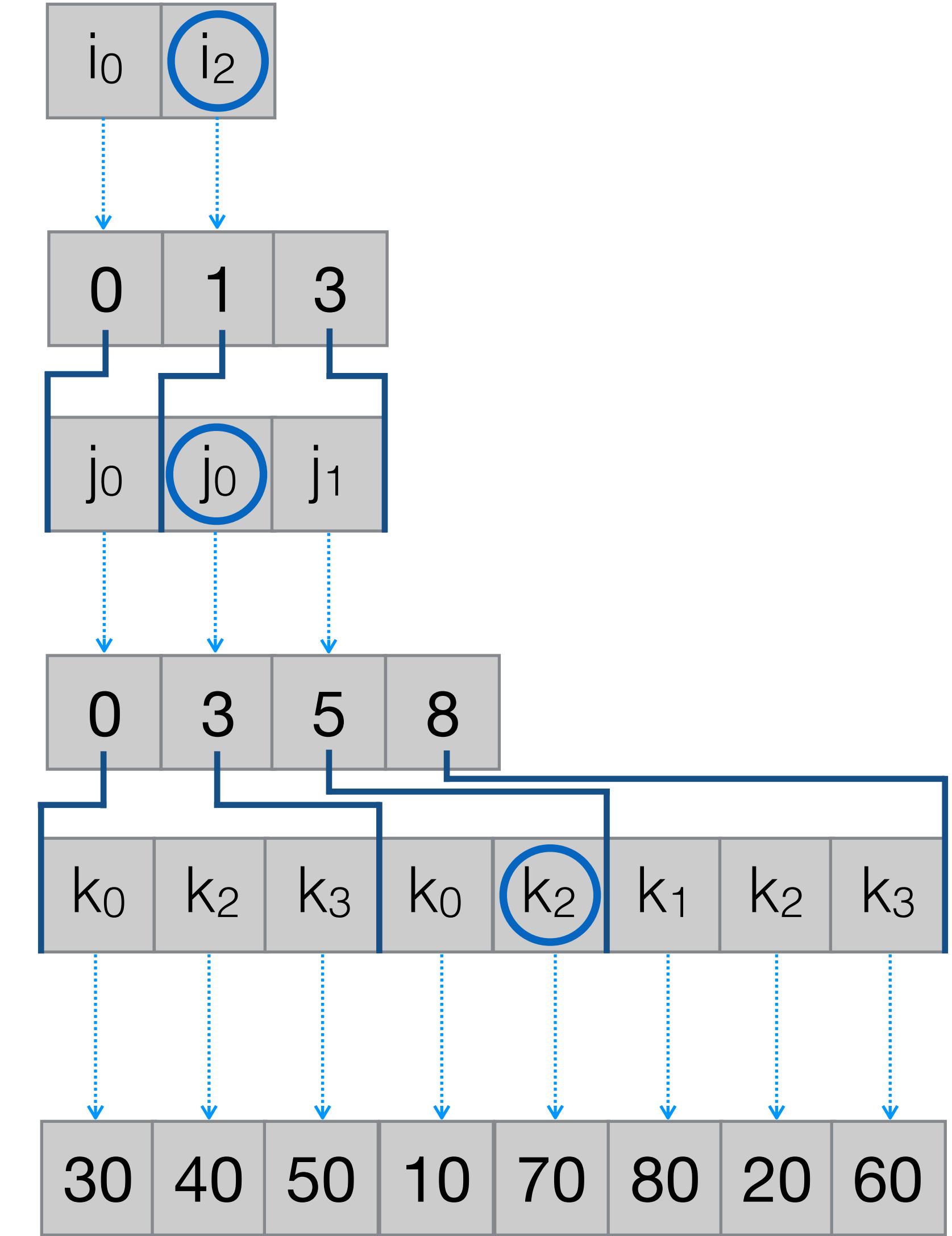
locate (i_2, j_0, k_2)



Compressed storage formats

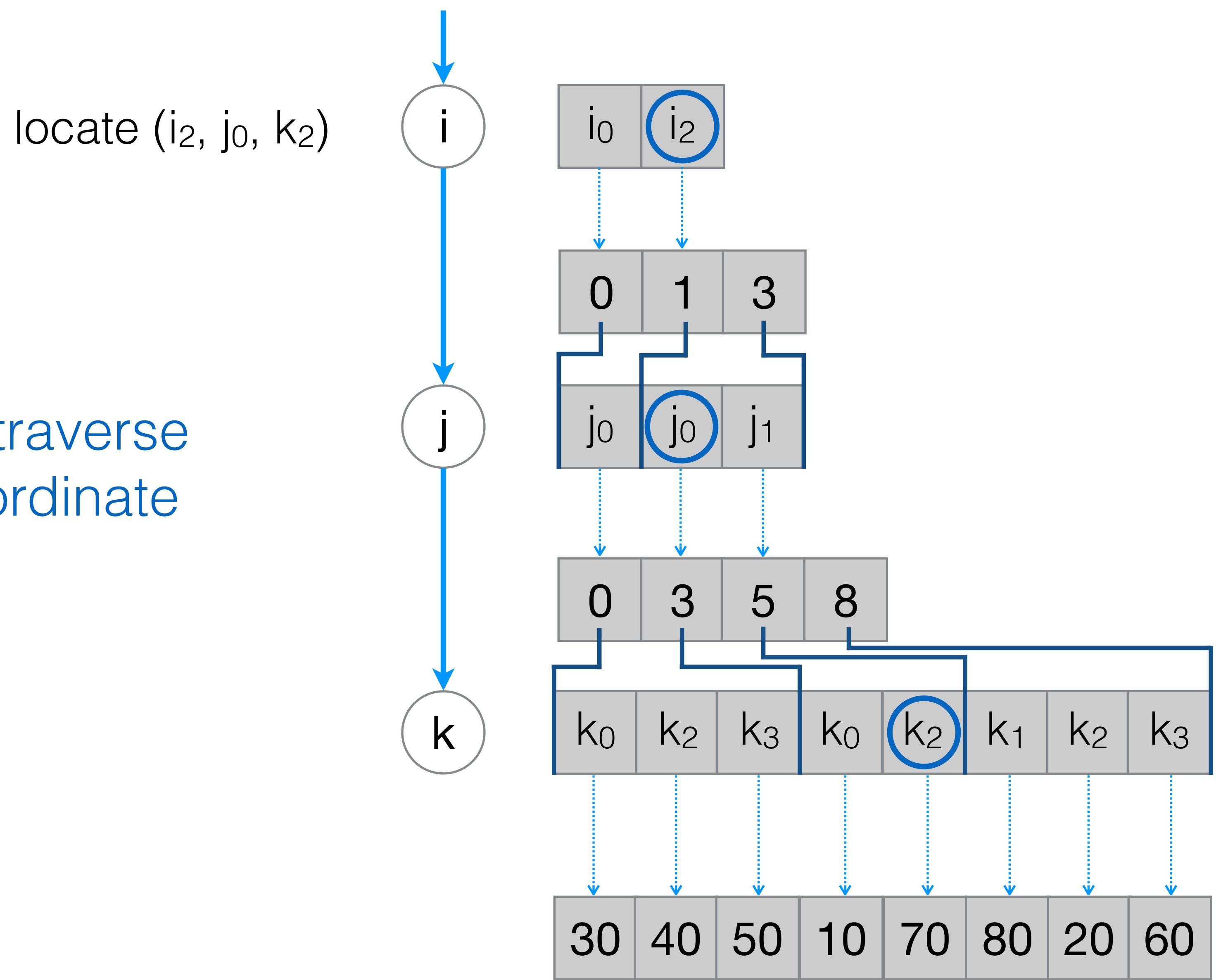
No random access: must traverse level by level to locate coordinate

locate (i_2, j_0, k_2)

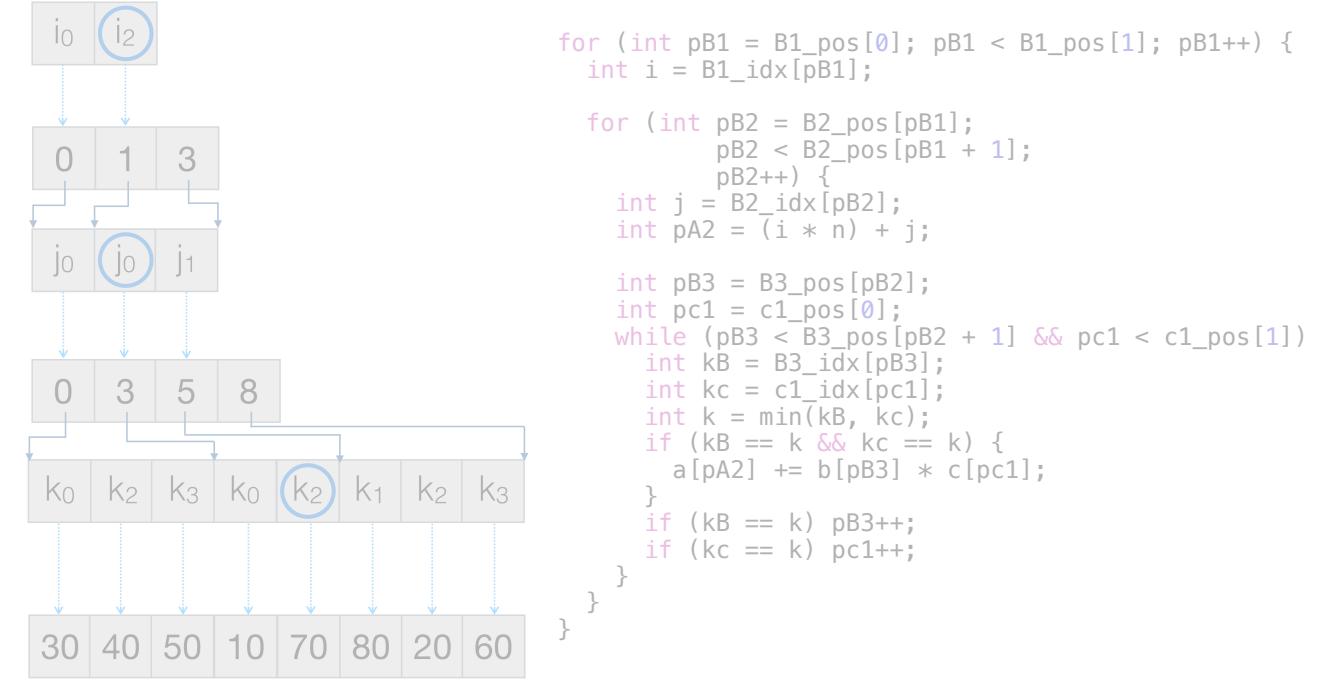


Compressed storage formats

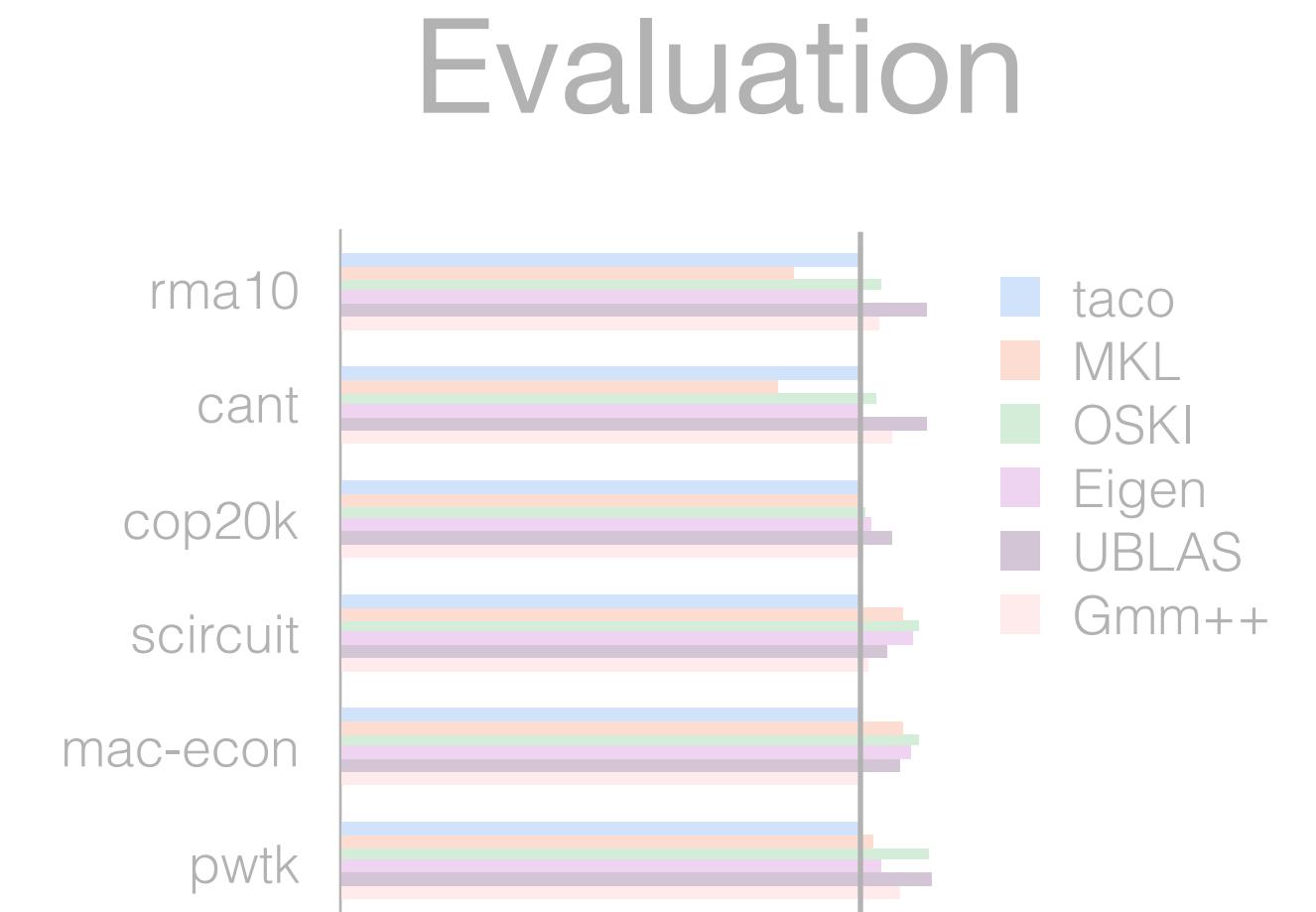
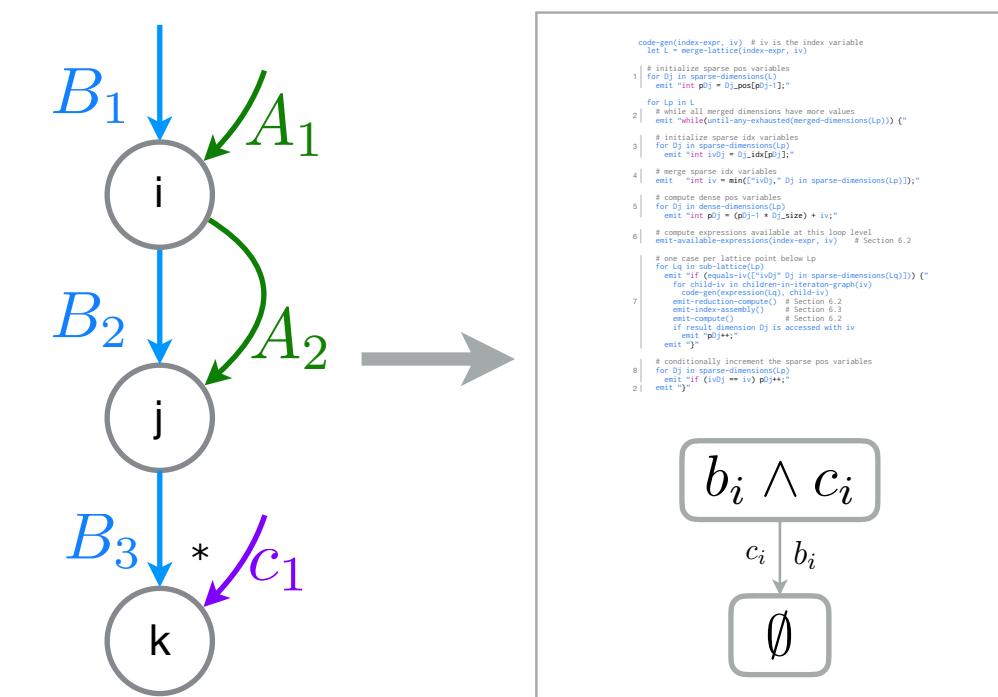
No random access: must traverse level by level to locate coordinate



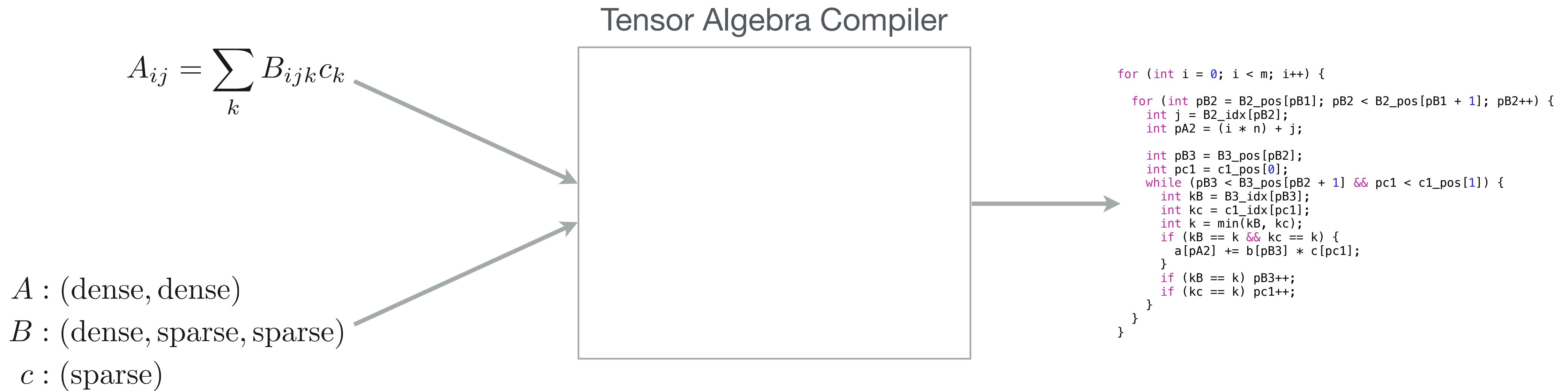
Sparse code and data



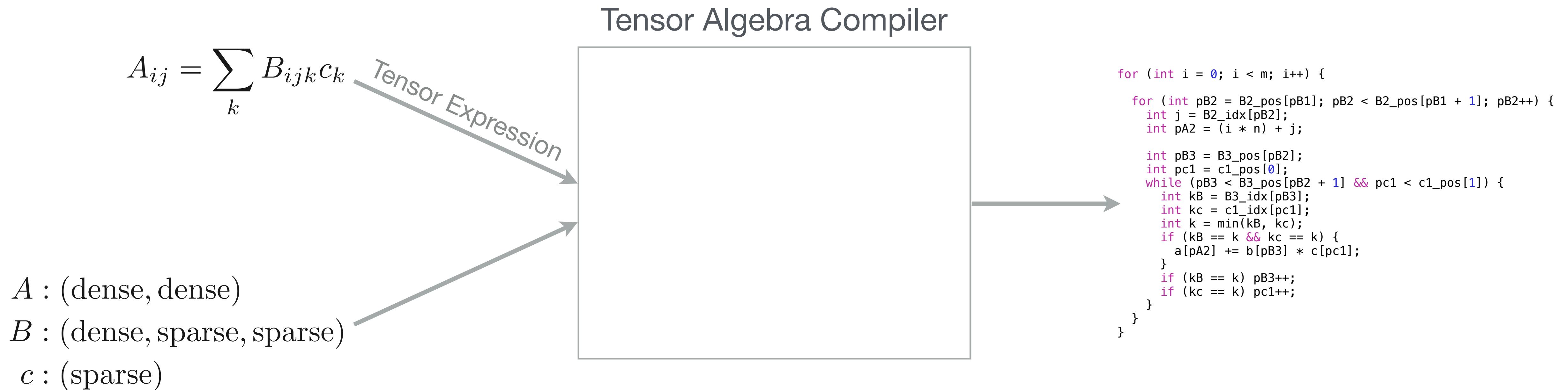
Intermediate Representation and Code Generation



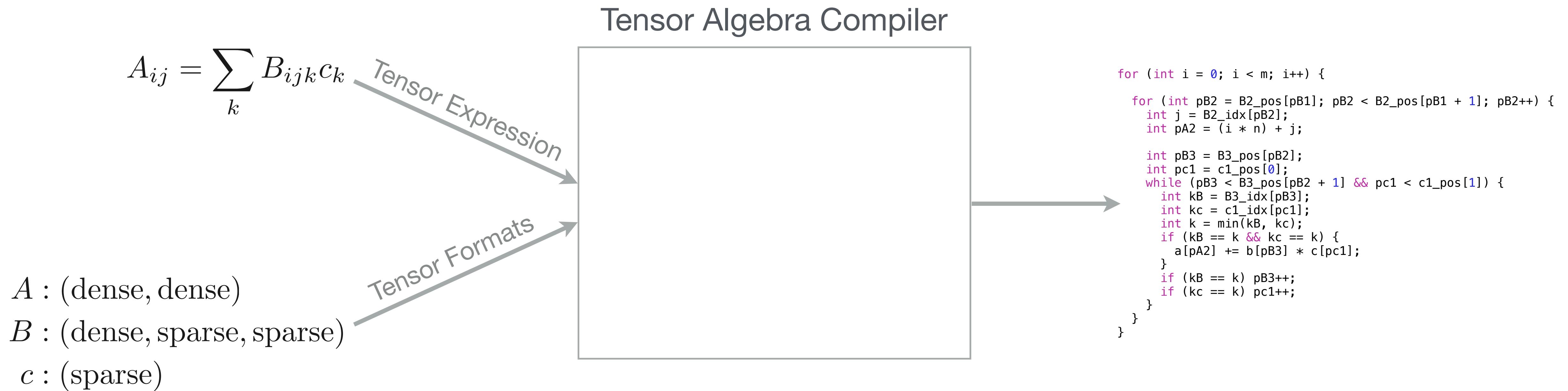
The Tensor Algebra Compiler (taco)



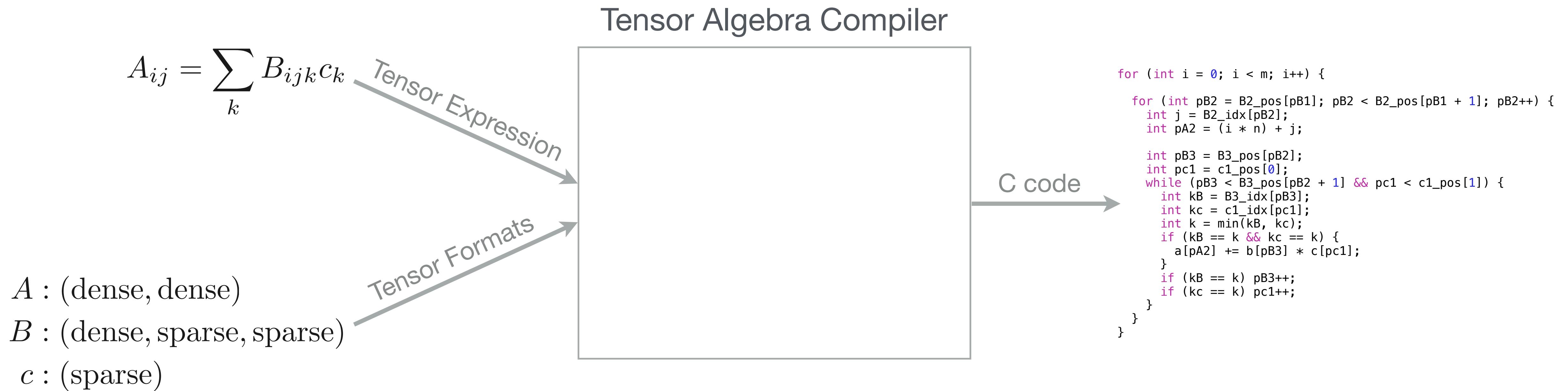
The Tensor Algebra Compiler (taco)



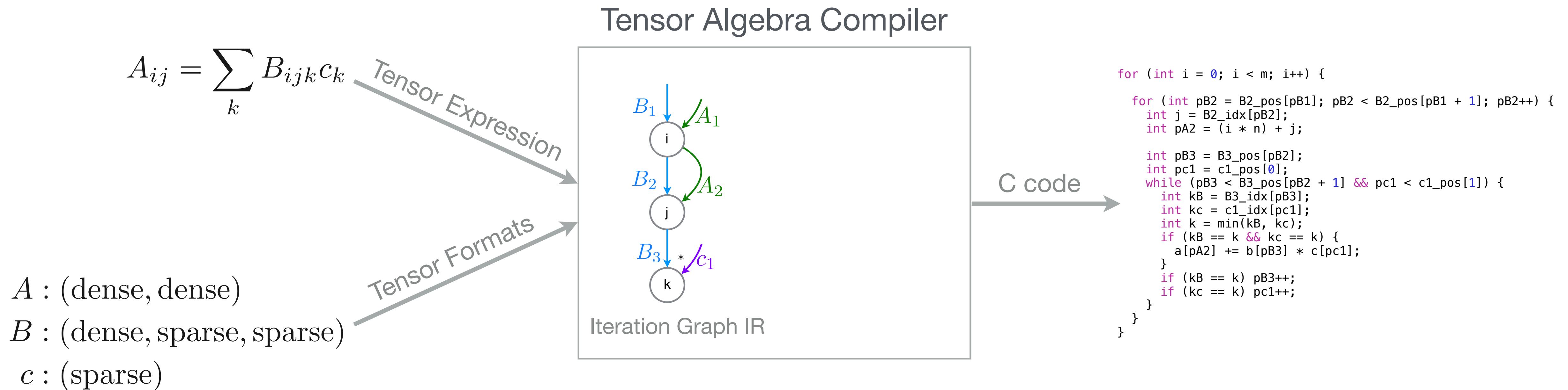
The Tensor Algebra Compiler (taco)



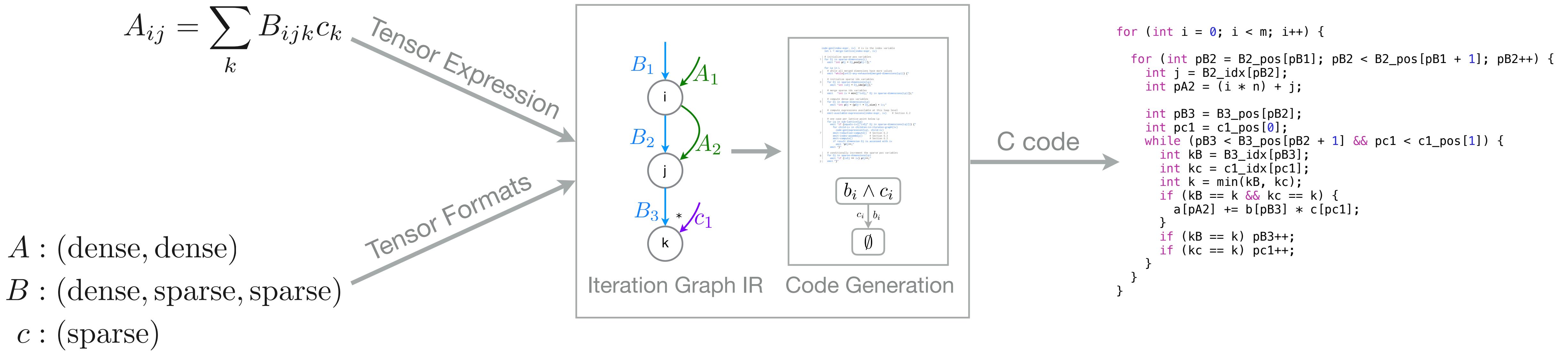
The Tensor Algebra Compiler (taco)



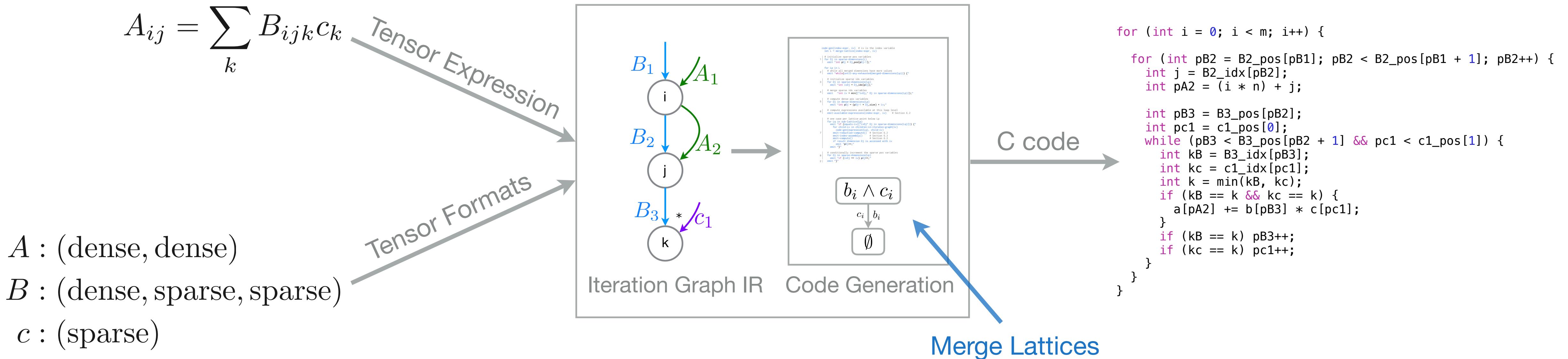
The Tensor Algebra Compiler (taco)



The Tensor Algebra Compiler (taco)



The Tensor Algebra Compiler (taco)



Sparse iteration spaces and Iteration Graphs

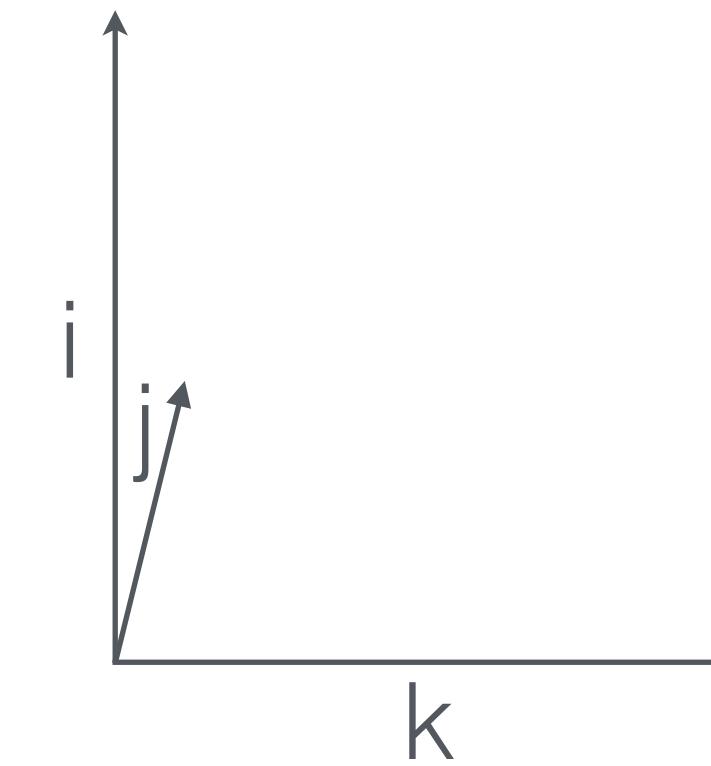
$$A_{ij} = \sum_k B_{ijk} * c_k$$

Sparse iteration spaces and Iteration Graphs

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

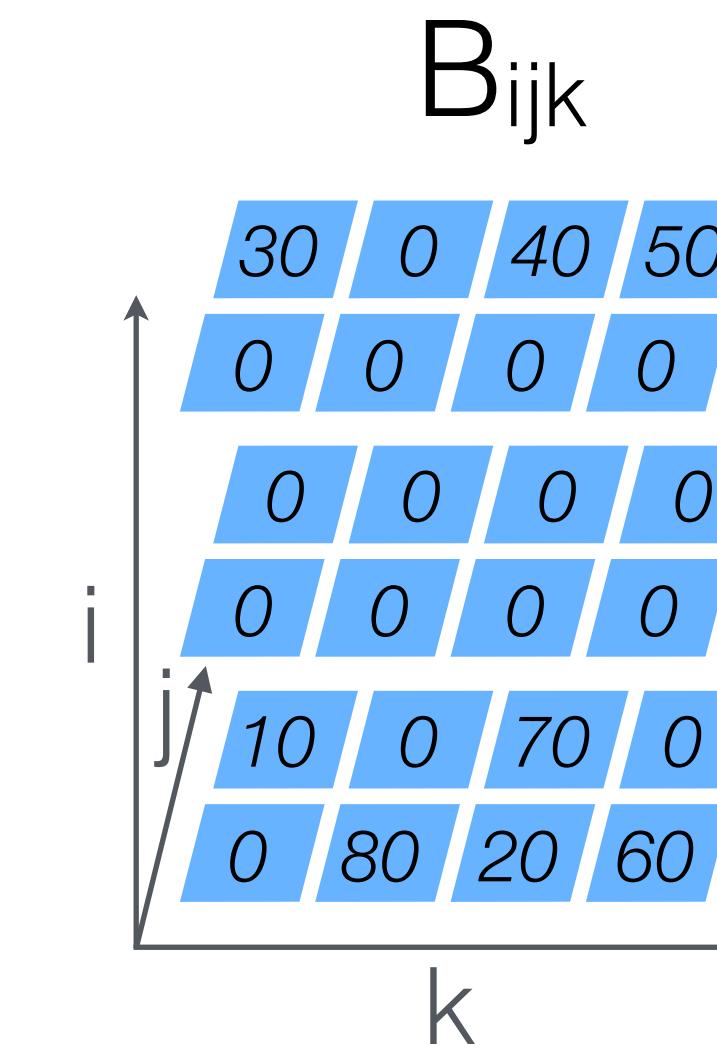
Sparse iteration spaces and Iteration Graphs

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



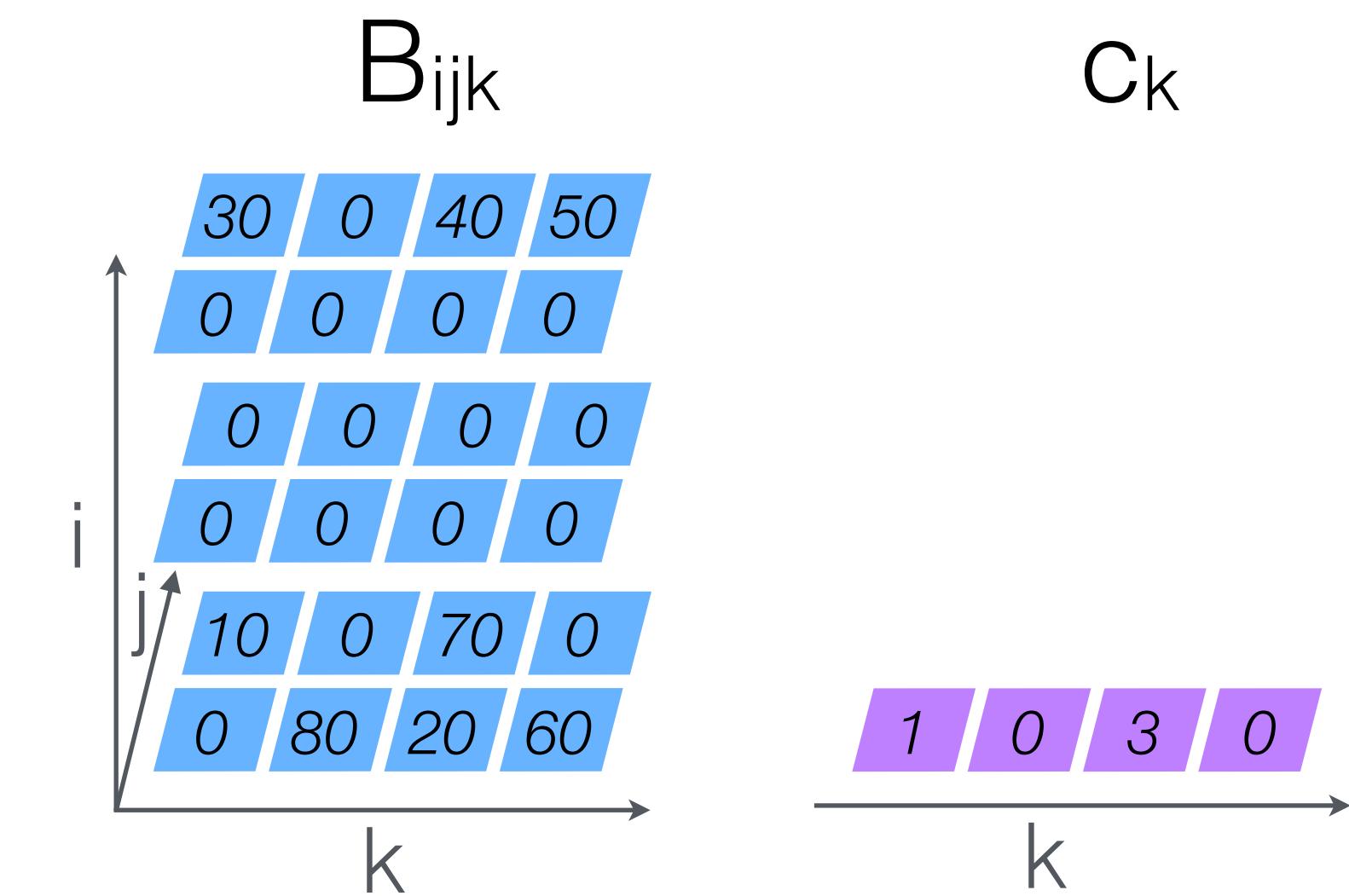
Sparse iteration spaces and Iteration Graphs

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



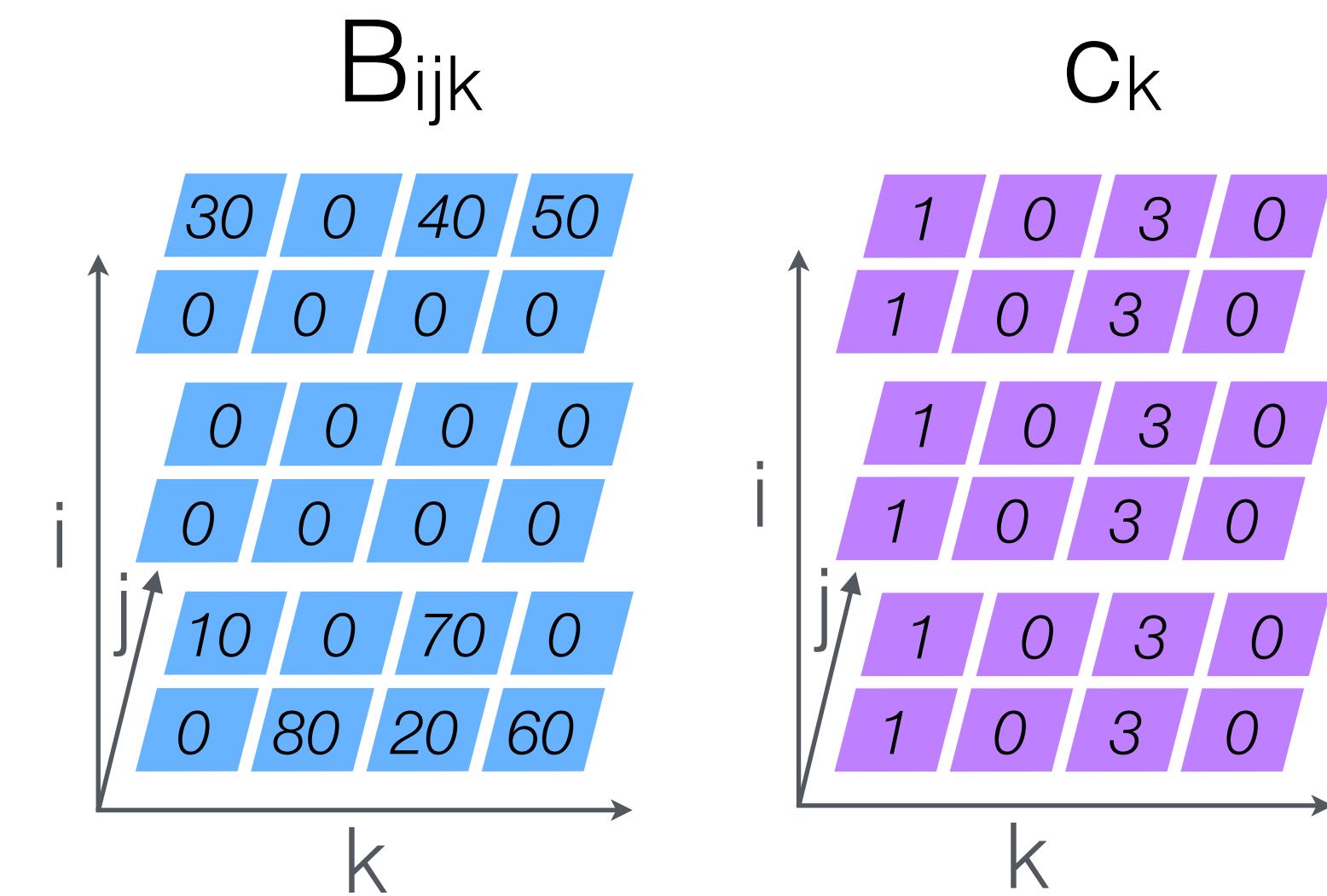
Sparse iteration spaces and Iteration Graphs

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



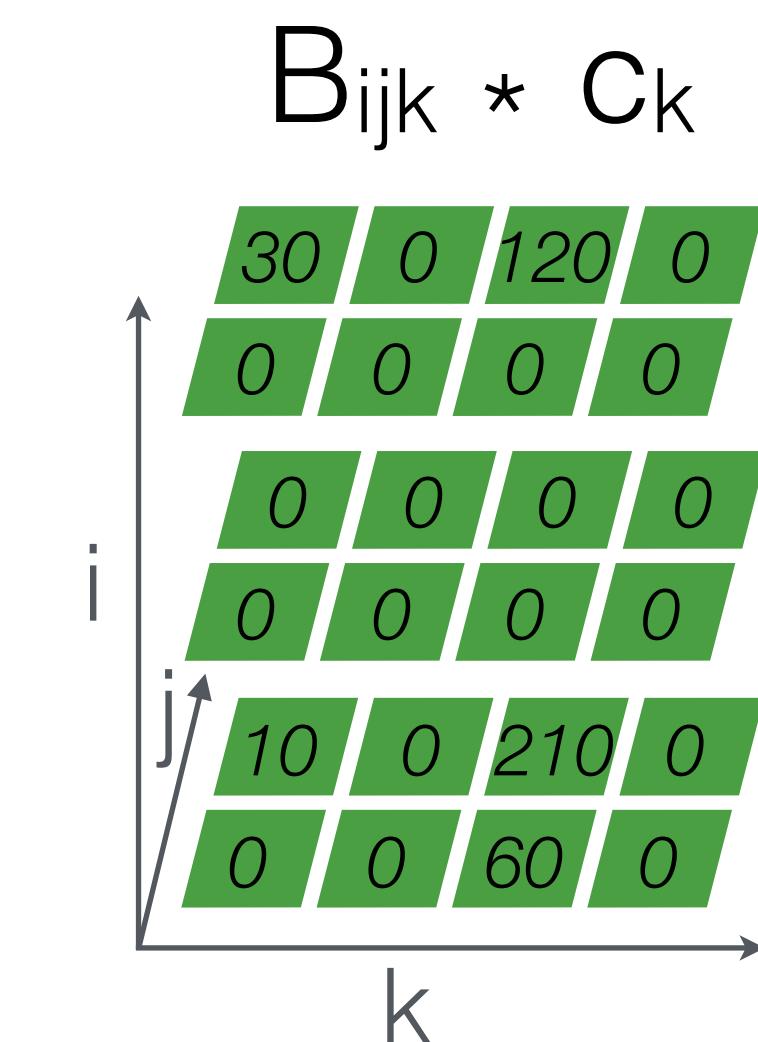
Sparse iteration spaces and Iteration Graphs

$$A_{ij} = \sum_k B_{ijk} * C_k$$



Sparse iteration spaces and Iteration Graphs

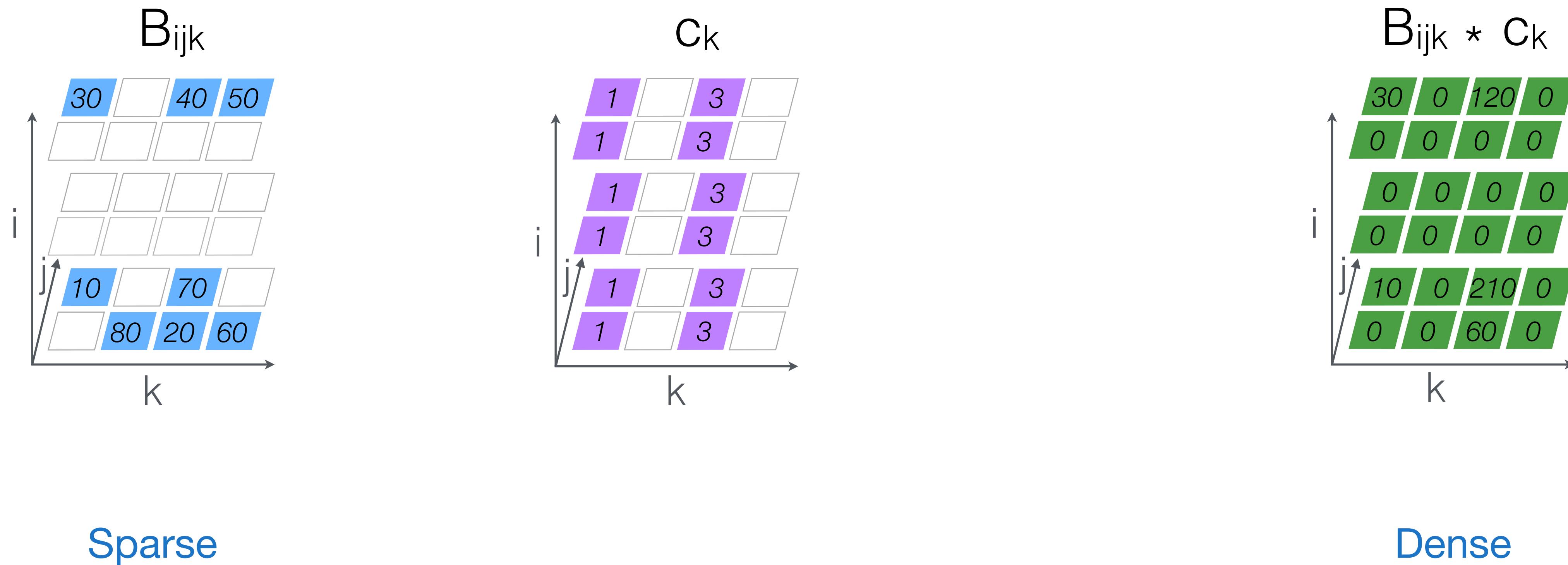
$$A_{ij} = \sum_k B_{ijk} * C_k$$



Dense

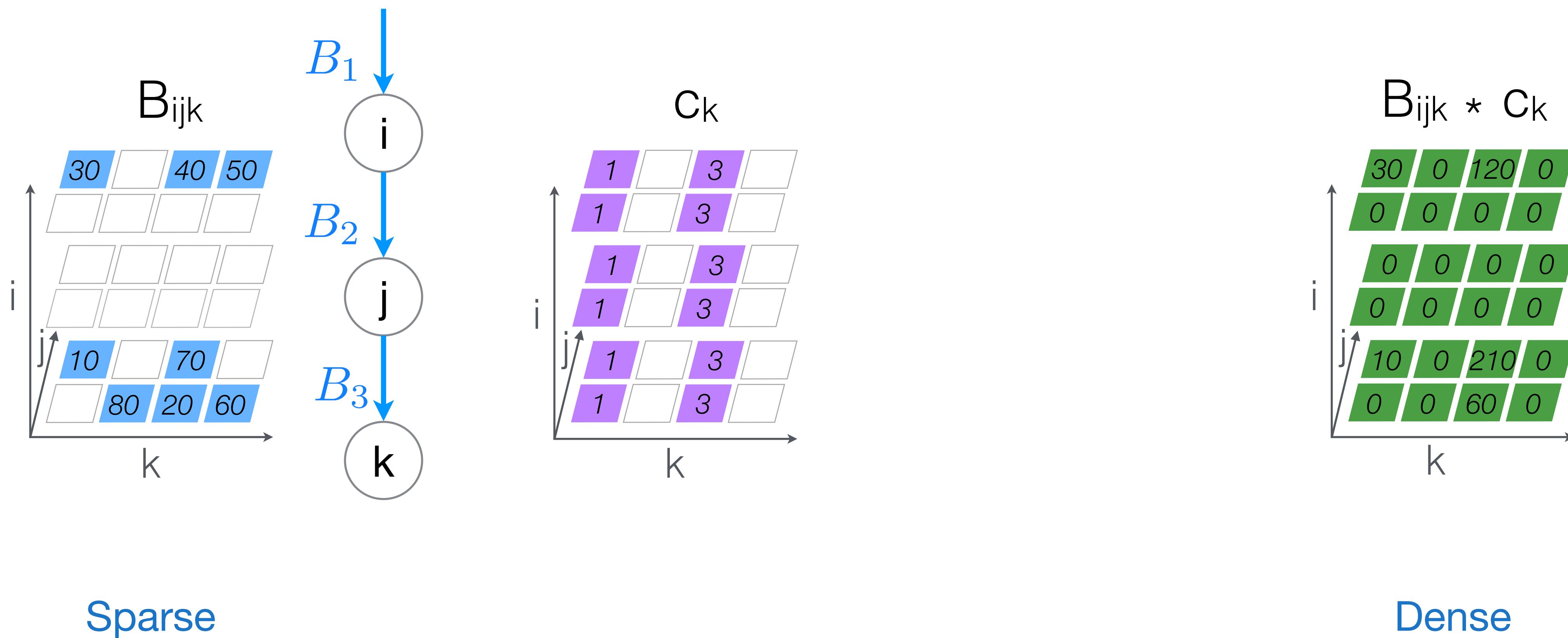
Sparse iteration spaces and Iteration Graphs

$$A_{ij} = \sum_k B_{ijk} * C_k$$



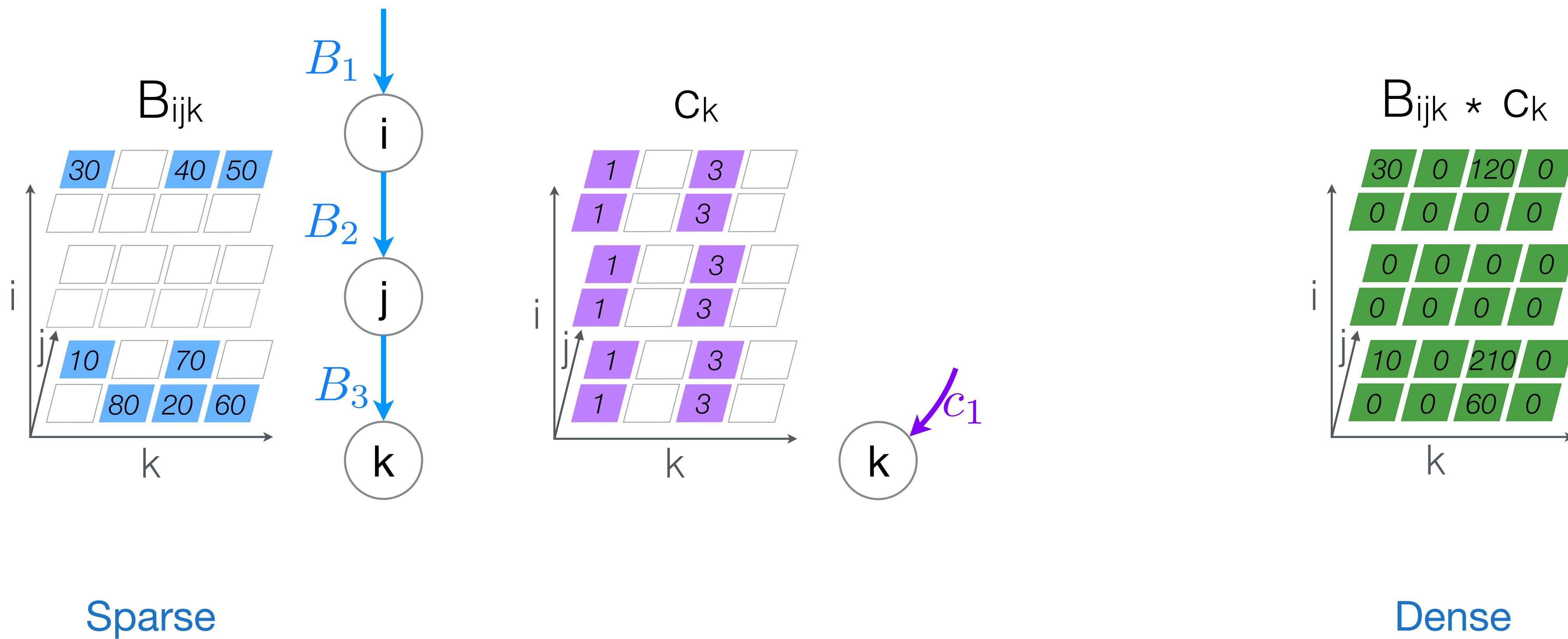
Sparse iteration spaces and Iteration Graphs

$$A_{ij} = \sum_k B_{ijk} * C_k$$



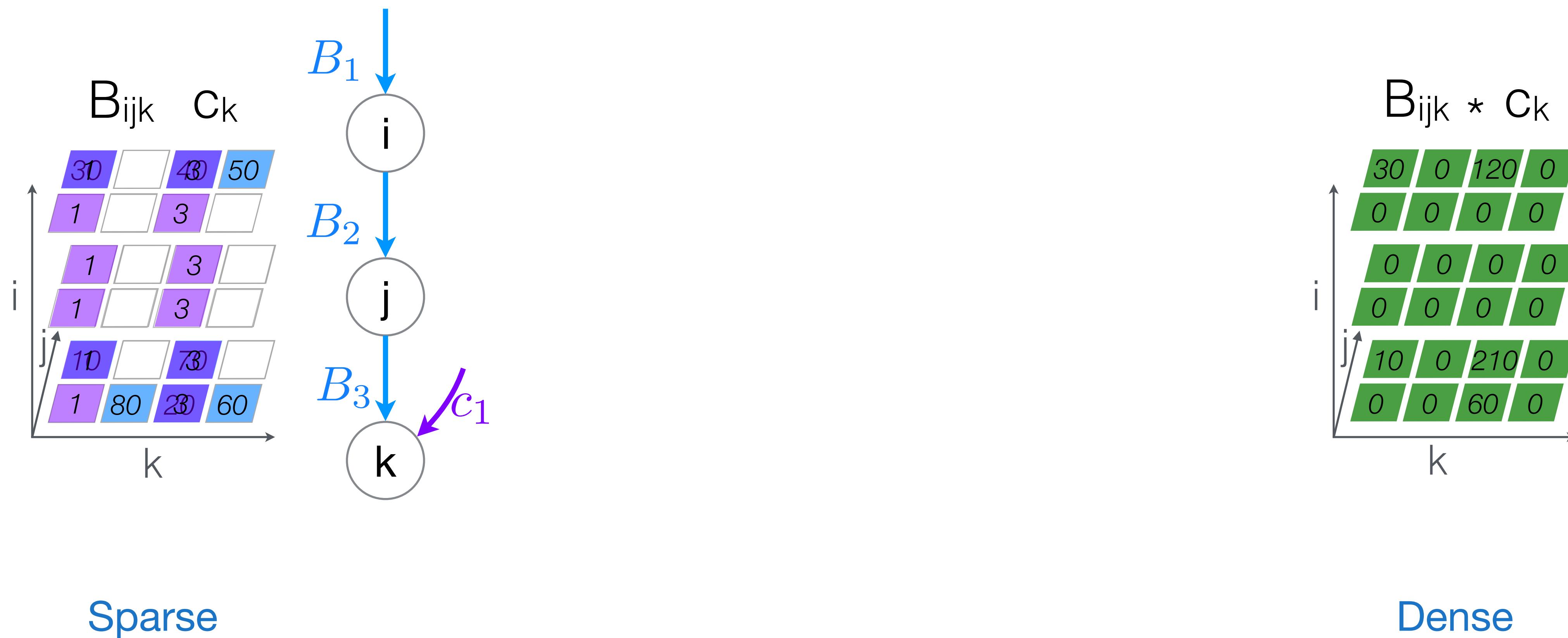
Sparse iteration spaces and Iteration Graphs

$$A_{ij} = \sum_k B_{ijk} * C_k$$



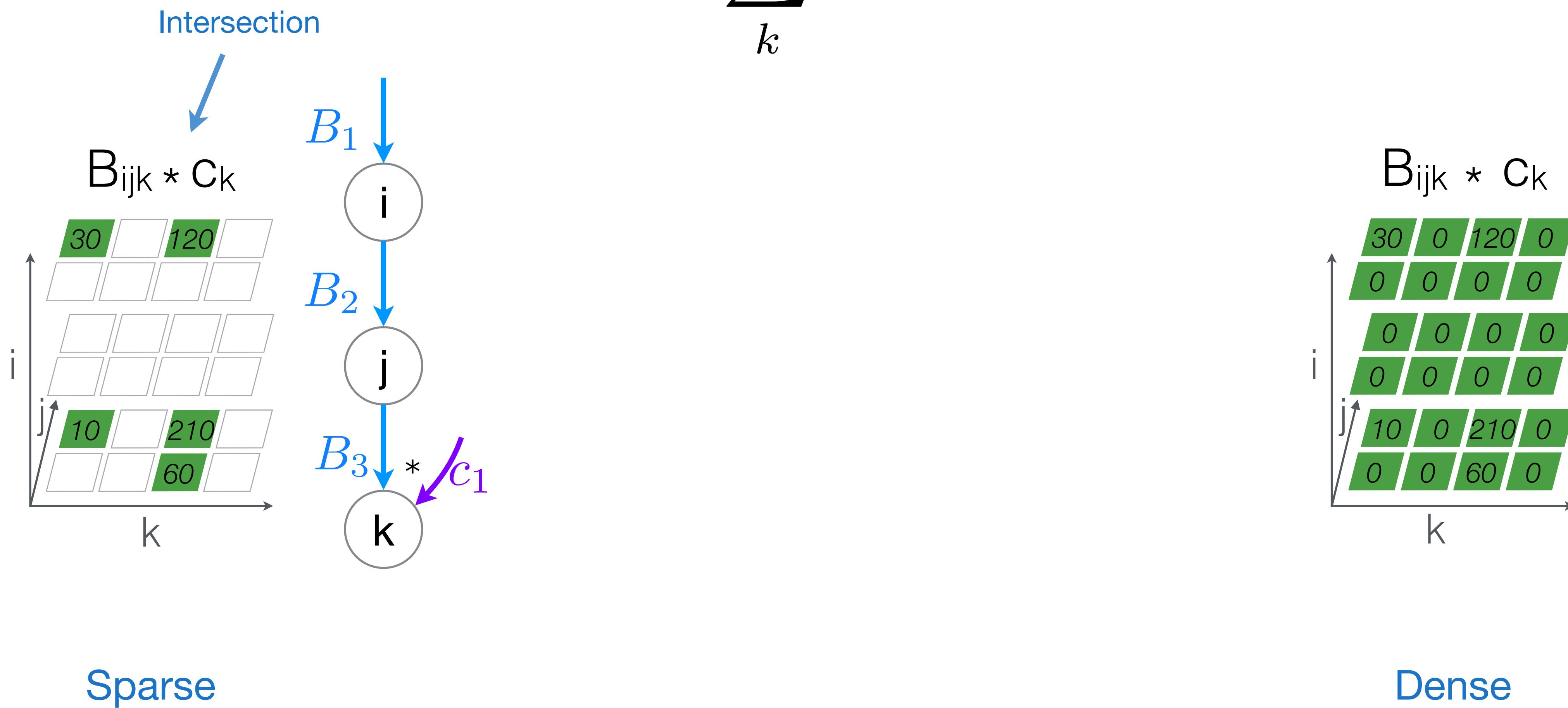
Sparse iteration spaces and Iteration Graphs

$$A_{ij} = \sum_k B_{ijk} * C_k$$



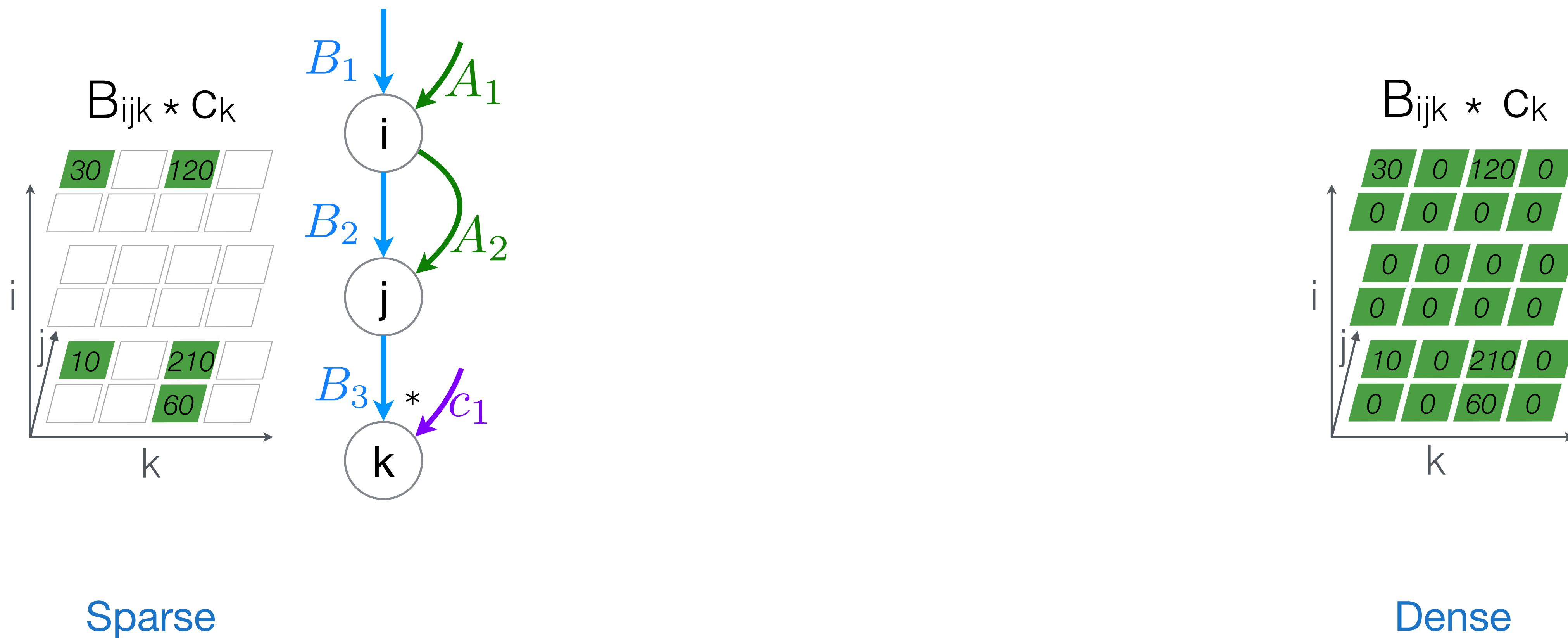
Sparse iteration spaces and Iteration Graphs

$$A_{ij} = \sum_k B_{ijk} * C_k$$



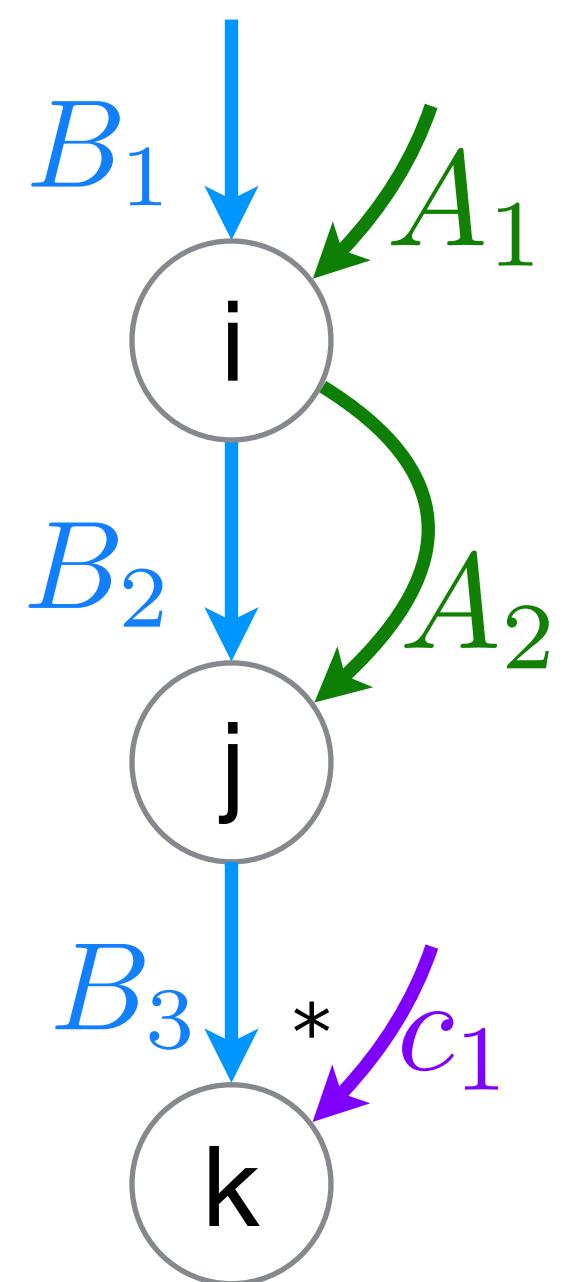
Sparse iteration spaces and Iteration Graphs

$$A_{ij} = \sum_k B_{ijk} * C_k$$



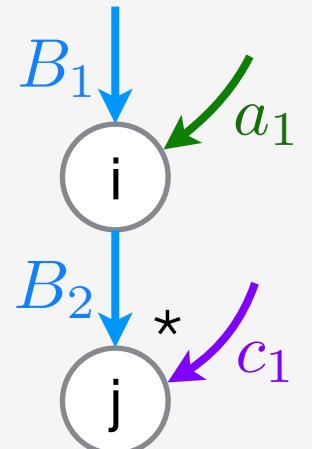
Examples of Iteration Graph

$$A_{ij} = \sum_k B_{ijk} * C_k$$

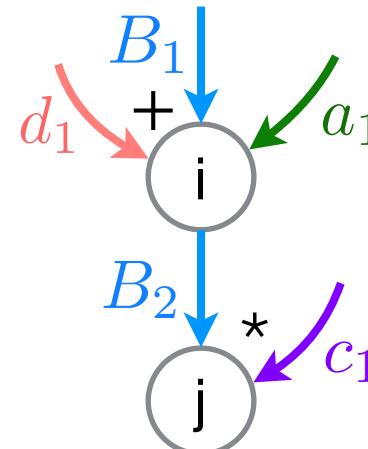


Examples of Iteration Graph

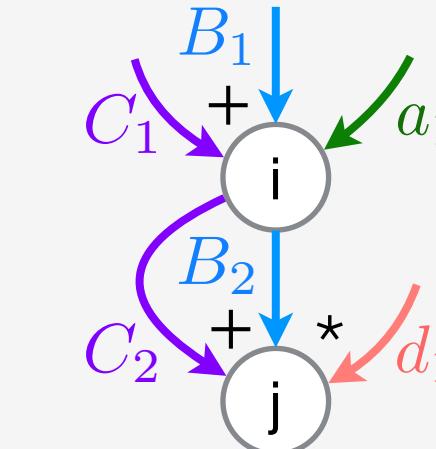
$$a_i = \sum_j B_{ij} c_j$$



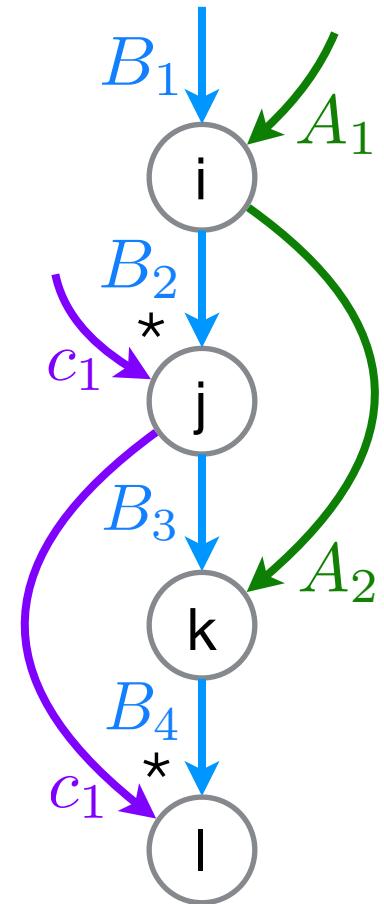
$$a_i = \sum_j \alpha B_{ij} c_j + \beta d_i$$



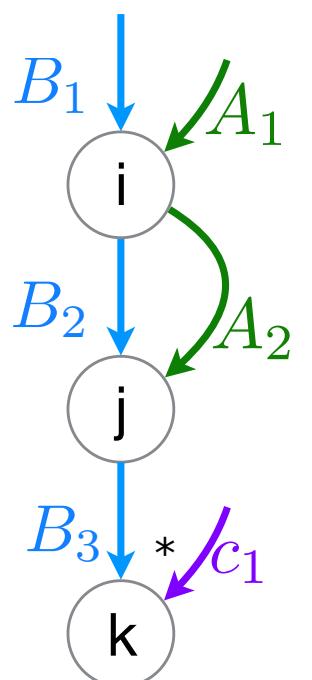
$$a_i = \sum_j (B_{ij} + C_{ij}) d_j$$



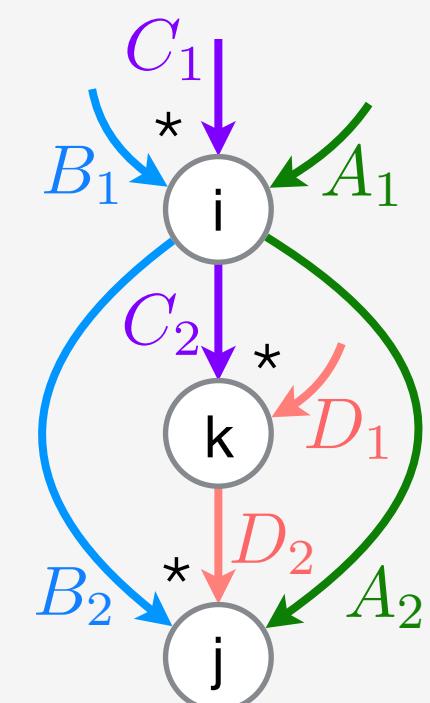
$$a_{ik} = \sum_j \sum_l B_{ijkl} c_{jl}$$



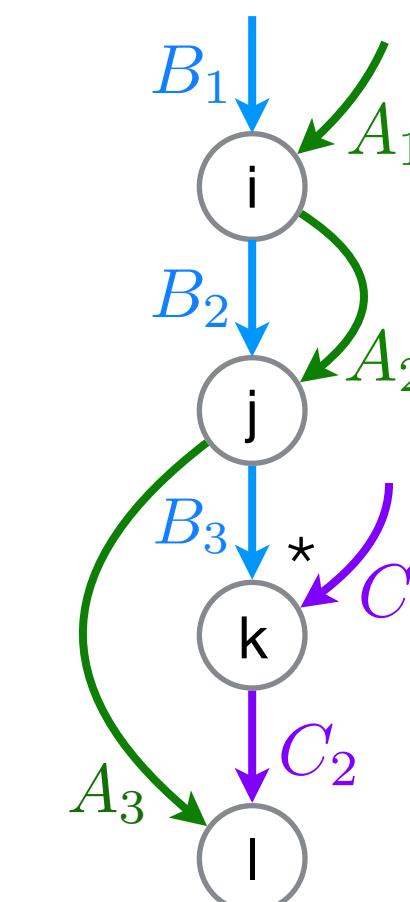
$$A_{ij} = \sum_k B_{ijk} * c_k$$



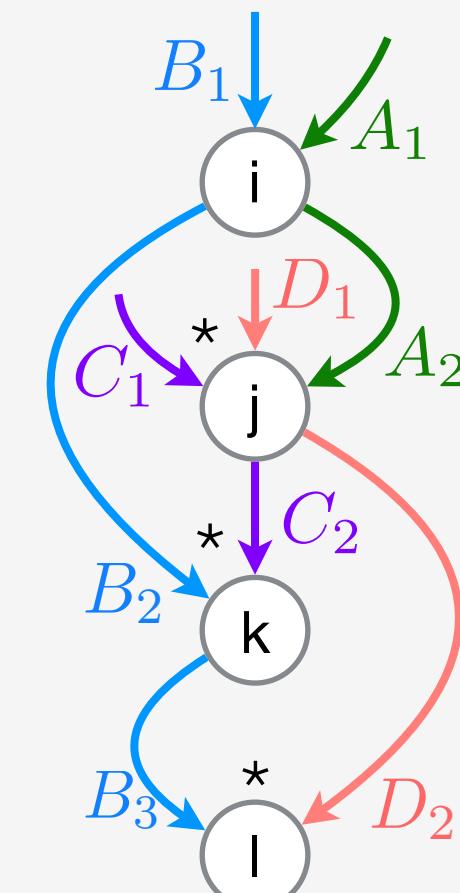
$$A_{ij} = \sum_k B_{ij} (C_{ik} D_{kj})$$



$$A_{ijl} = \sum_k B_{ijk} C_{lk}$$

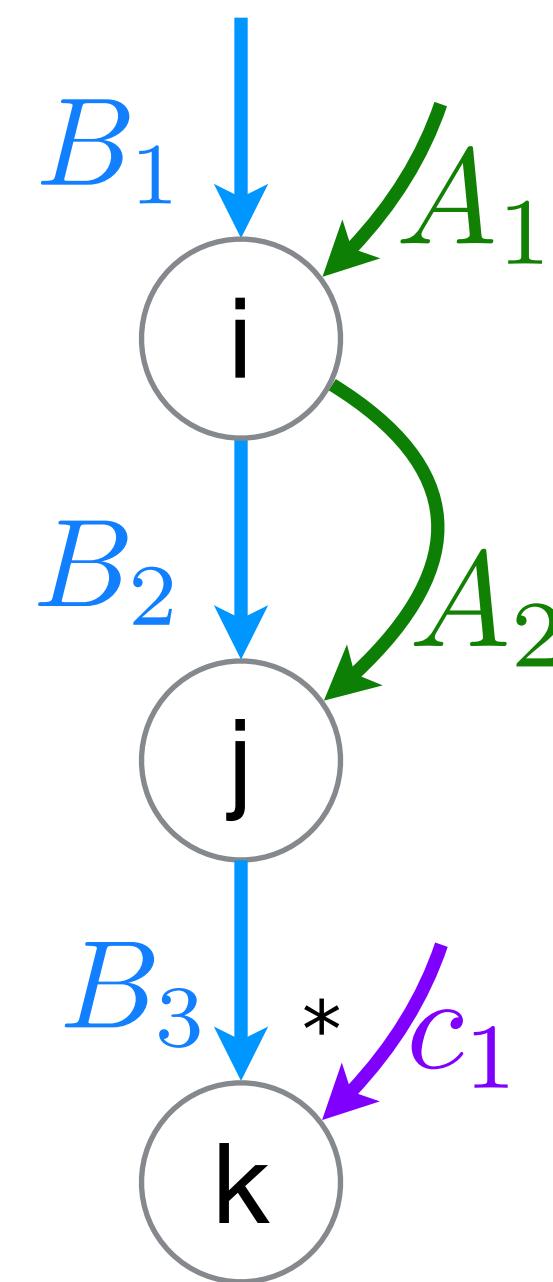


$$A_{ij} = \sum_k \sum_l B_{ikl} C_{kji} D_{lj}$$



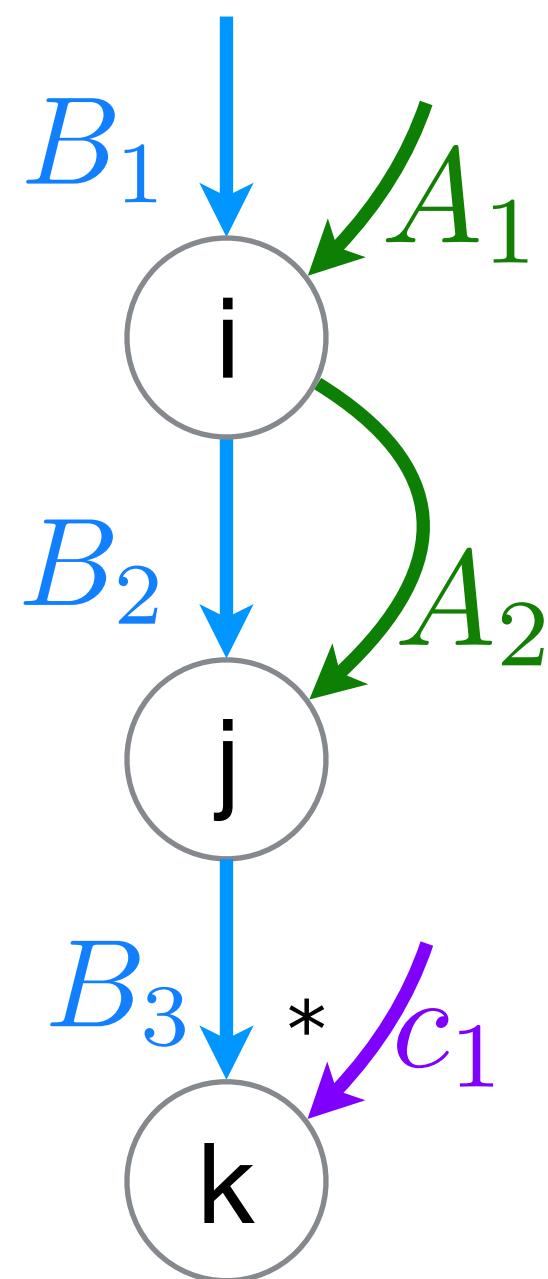
Examples of Iteration Graph

$$A_{ij} = \sum_k B_{ijk} * C_k$$



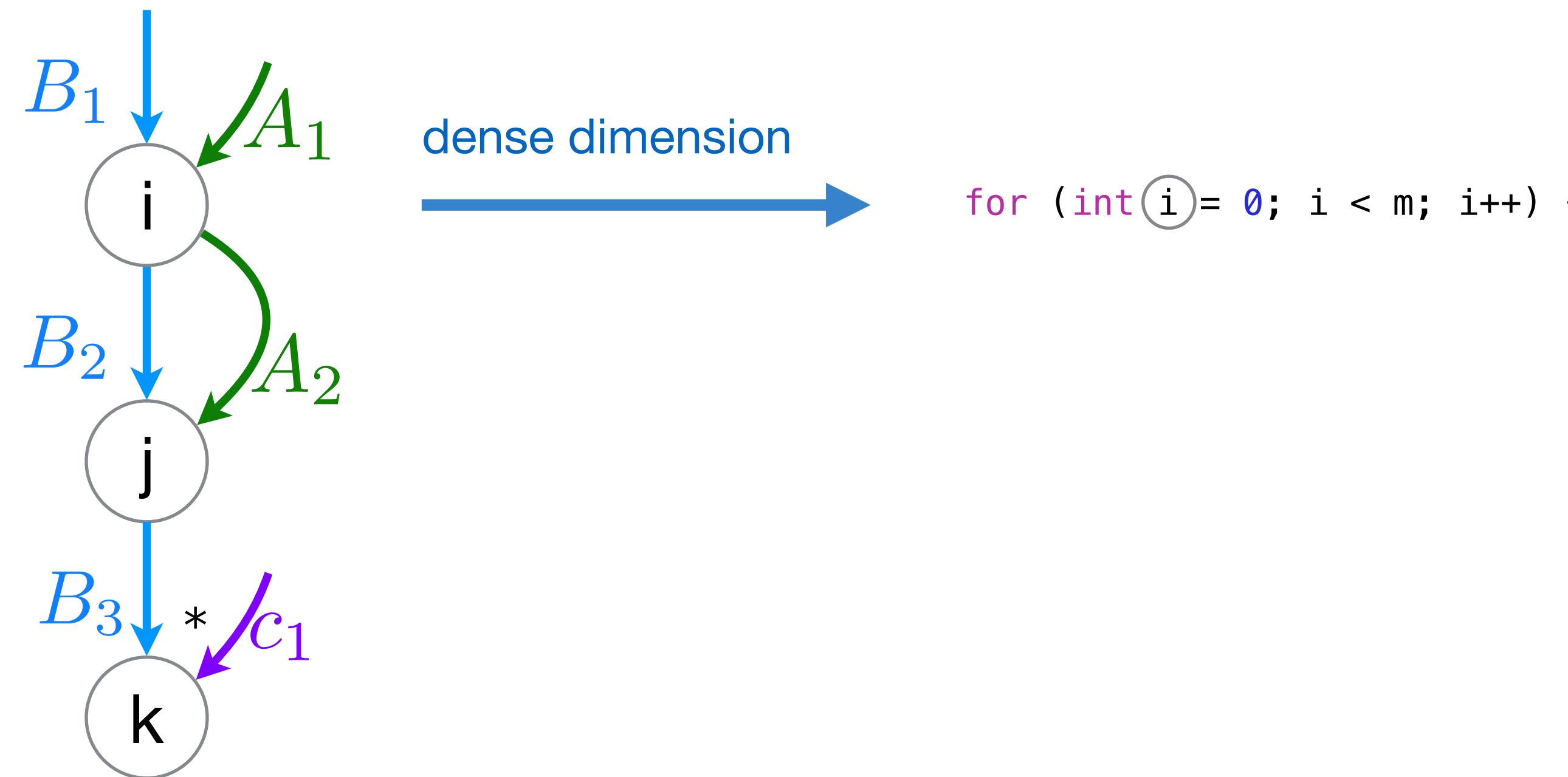
Code Generation from Iteration Graphs

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



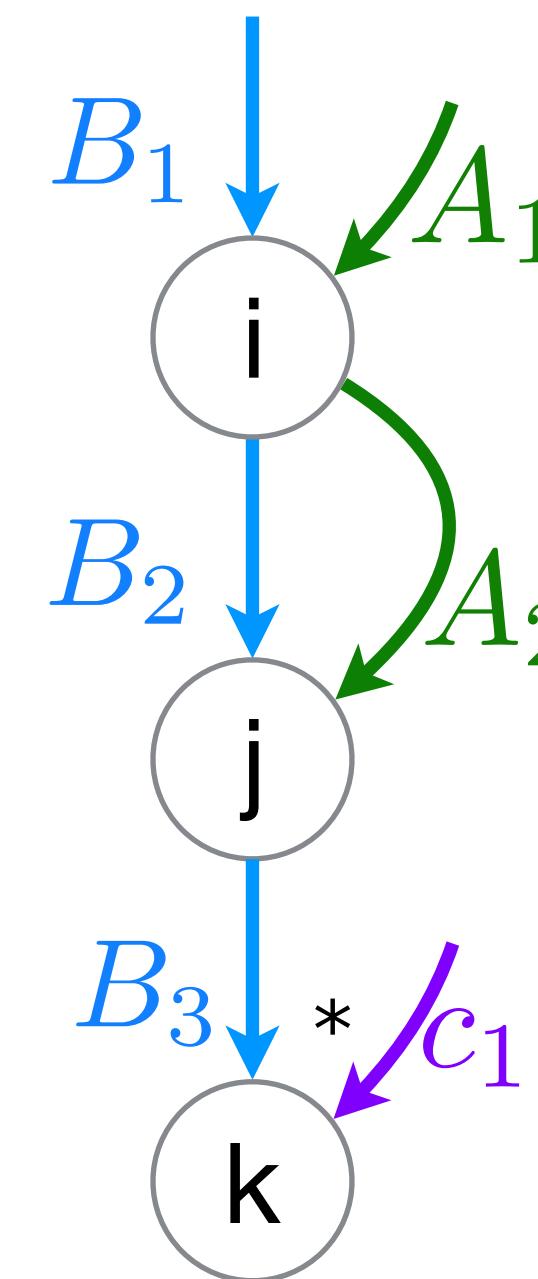
Code Generation from Iteration Graphs

$$A_{ij} = \sum_k B_{ijk} * C_k$$



Code Generation from Iteration Graphs

$$A_{ij} = \sum_k B_{ijk} * C_k$$



dense dimension

sparse dimension

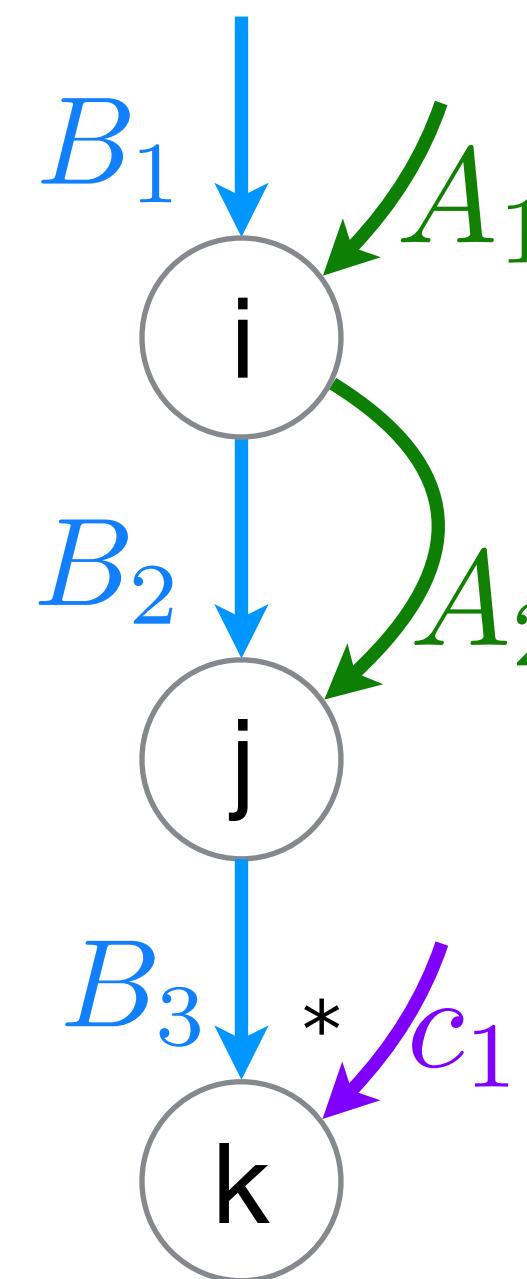
```
for (int i = 0; i < m; i++) {
```

```
    for (int pB2 = B2_pos[pB1]; pB2 < B2_pos[pB1 + 1]; pB2++) {  
        int j = B2_idx[pB2];  
        int pA2 = (i * n) + j;
```

```
}
```

Code Generation from Iteration Graphs

$$A_{ij} = \sum_k B_{ijk} * C_k$$



dense dimension

sparse dimension

merged dimensions

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

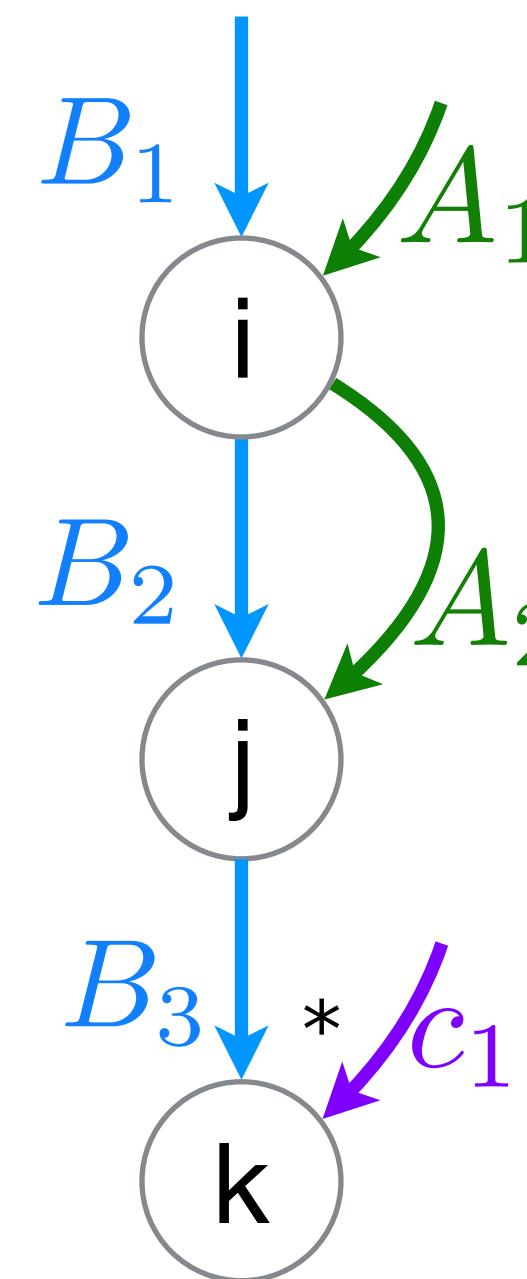
```
    for (int pB2 = B2_pos[pB1]; pB2 < B2_pos[pB1 + 1]; pB2++) {  
        int j = B2_idx[pB2];  
        int pA2 = (i * n) + j;
```

```
        int pB3 = B3_pos[pB2];  
        int pc1 = c1_pos[0];  
        while (pB3 < B3_pos[pB2 + 1] && pc1 < c1_pos[1]) {  
            int kB = B3_idx[pB3];  
            int kc = c1_idx[pc1];  
            int k = min(kB, kc);  
            if (kB == k && kc == k) {
```

```
                }  
                if (kB == k) pB3++;  
                if (kc == k) pc1++;  
            }  
        }
```

Code Generation from Iteration Graphs

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



dense dimension

sparse dimension

merged dimensions

compute statement

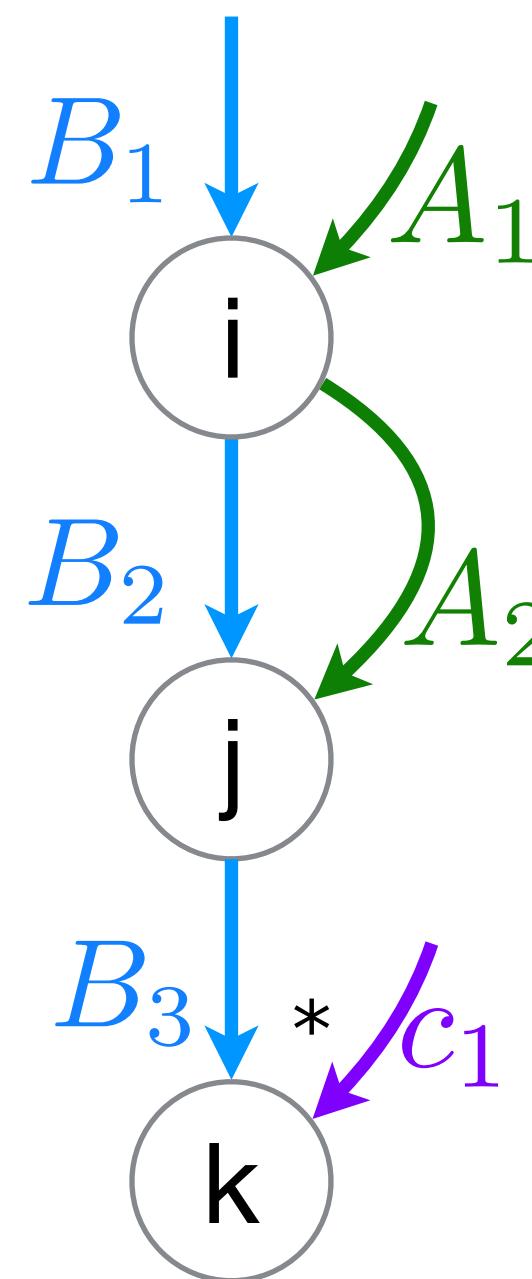
```
for (int i = 0; i < m; i++) {
```

```
    for (int pB2 = B2_pos[pB1]; pB2 < B2_pos[pB1 + 1]; pB2++) {  
        int j = B2_idx[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_idx[pB3];  
            int kc = c1_idx[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++;  
        }  
    }  
}
```

Code Generation from Iteration Graphs

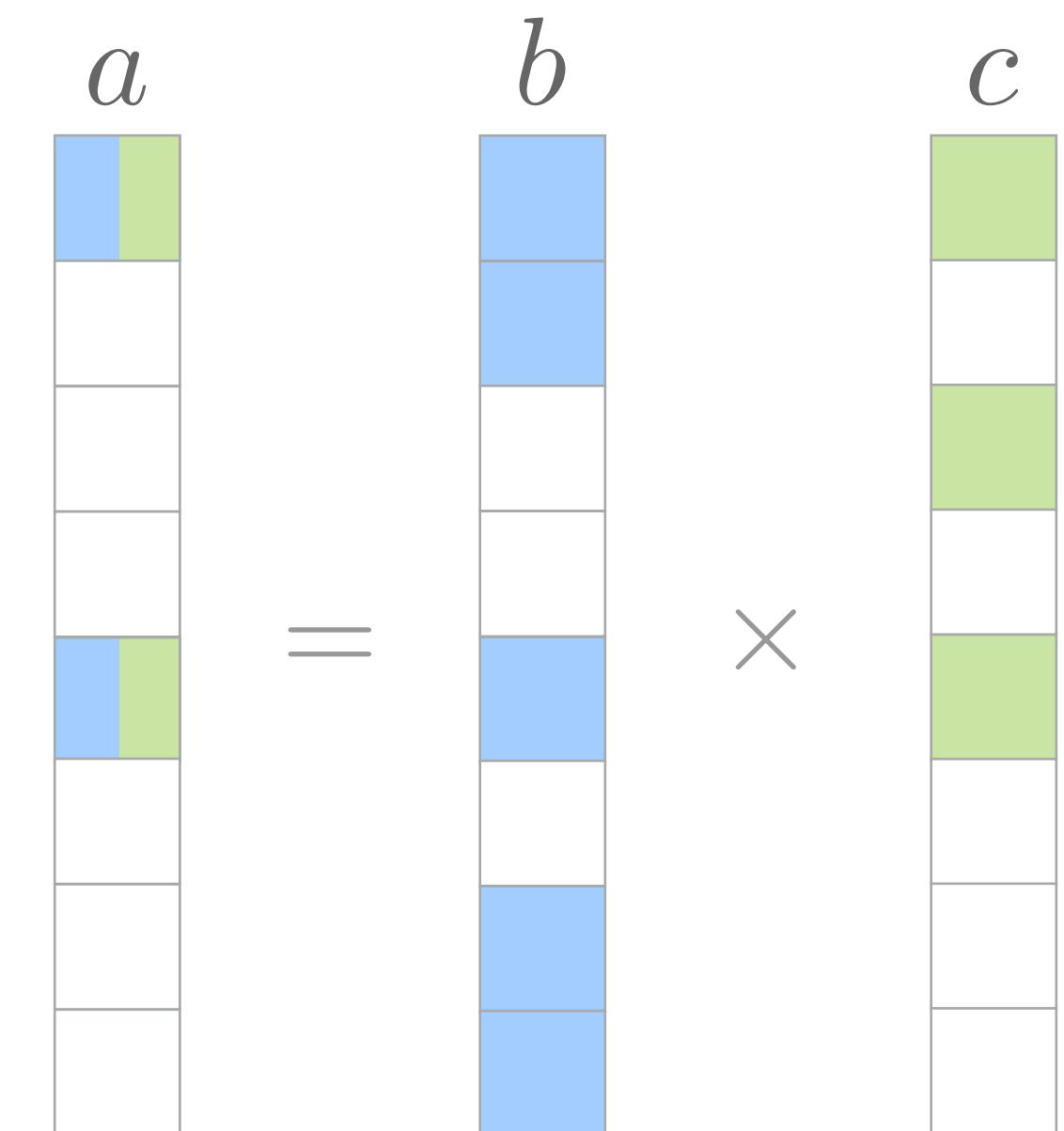
$$A_{ij} = \sum_k B_{ijk} * c_k$$



```
for (int i= 0; i < m; i++) {  
    for (int pB2 = B2_pos[pB1]; pB2 < B2_pos[pB1 + 1]; pB2++) {  
        int j= B2_idx[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_idx[pB3];  
            int kc = c1_idx[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++;  
        }  
    }  
}
```

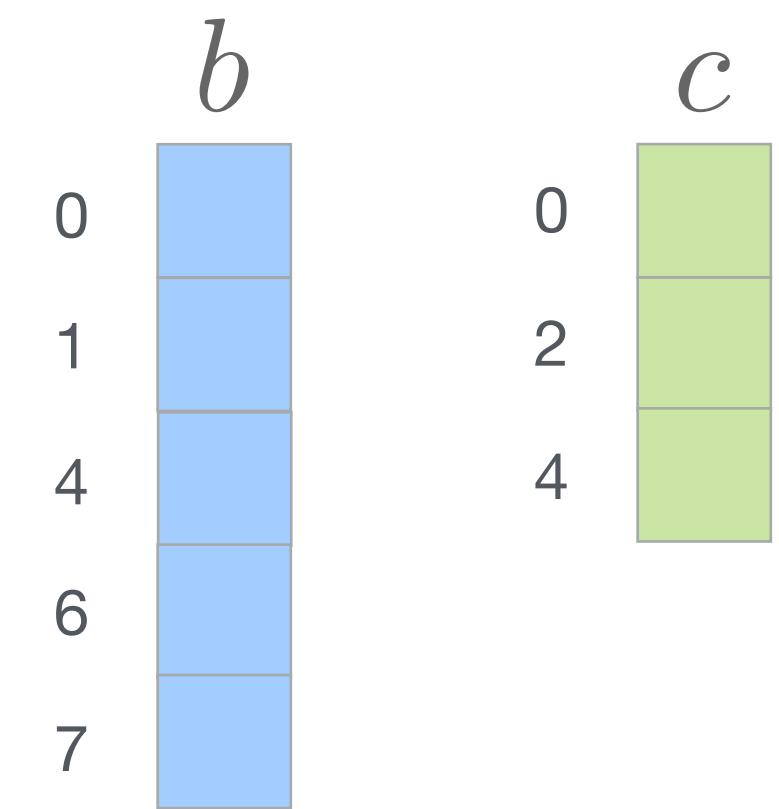
Merge Lattice for a Conjunction

$$a_i = b_i c_i$$



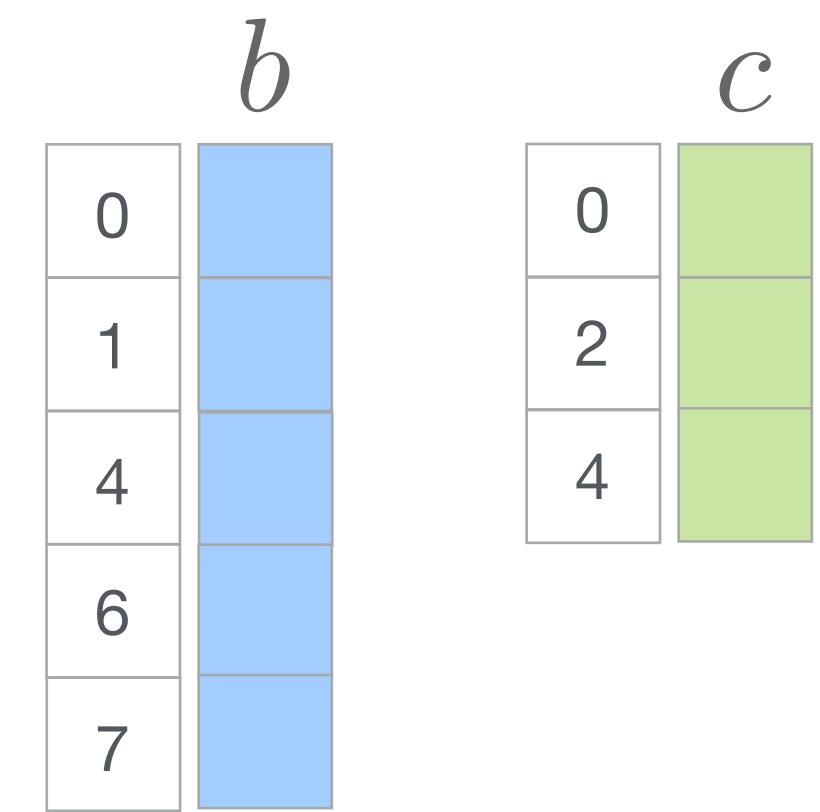
Merge Lattice for a Conjunction

$$a_i = b_i c_i$$



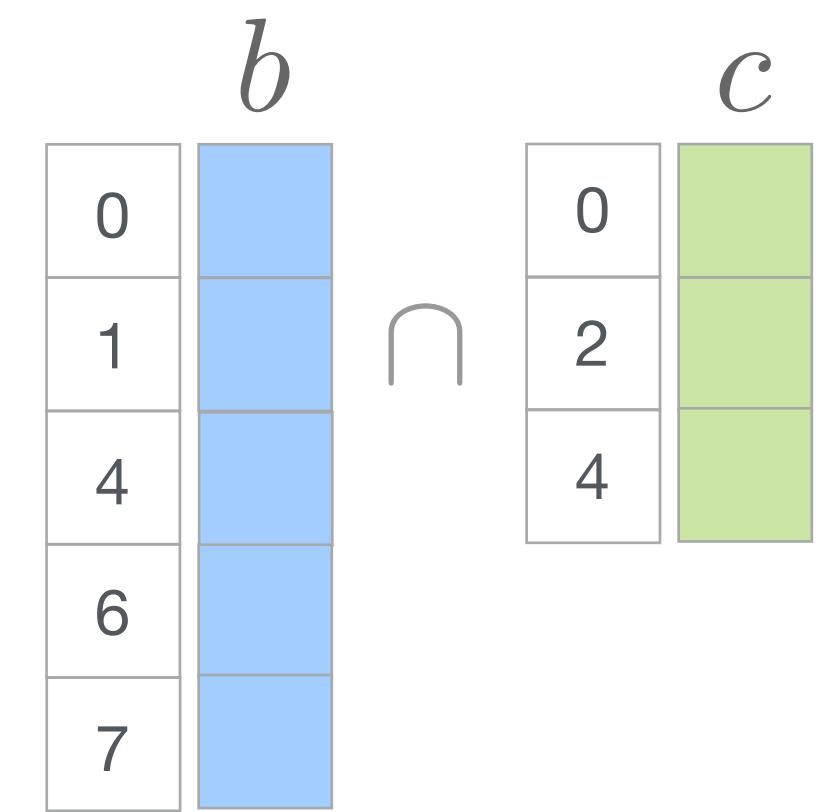
Merge Lattice for a Conjunction

$$a_i = b_i c_i$$



Merge Lattice for a Conjunction

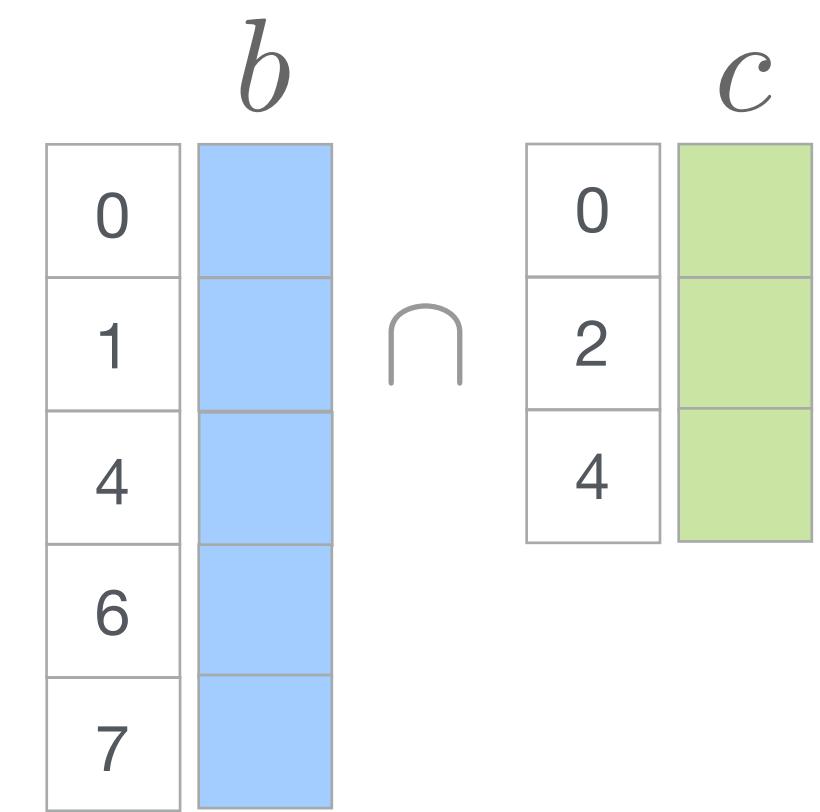
$$a_i = b_i c_i$$



Merge Lattice for a Conjunction

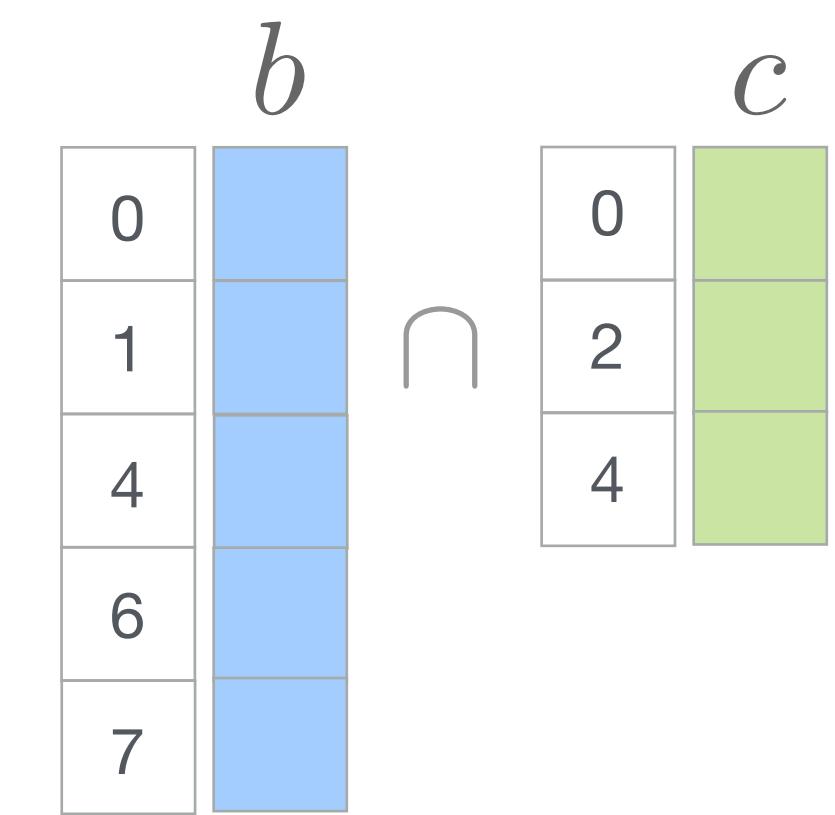
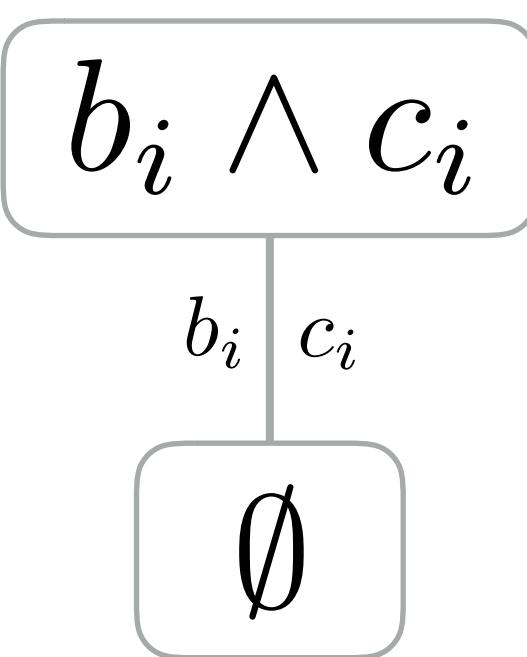
$$a_i = b_i c_i$$

$b_i \wedge c_i$



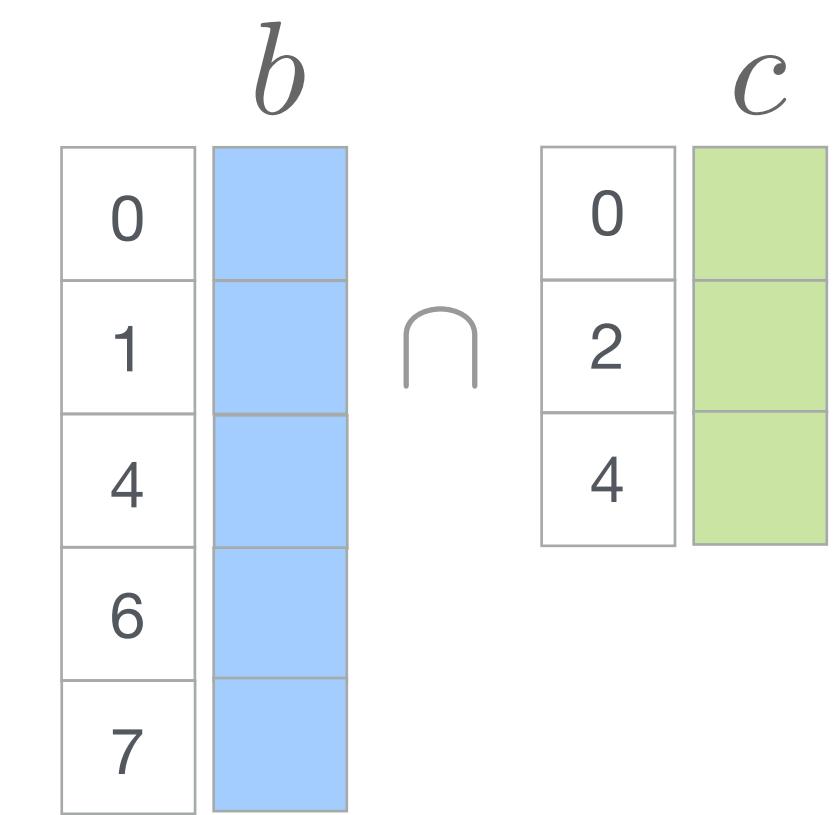
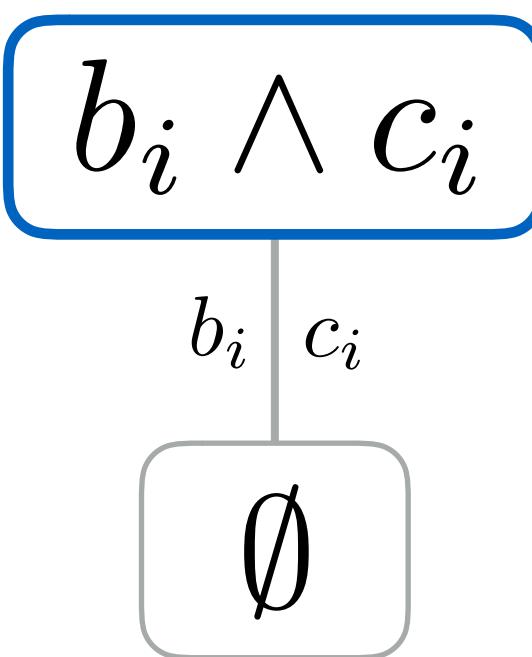
Merge Lattice for a Conjunction

$$a_i = b_i c_i$$



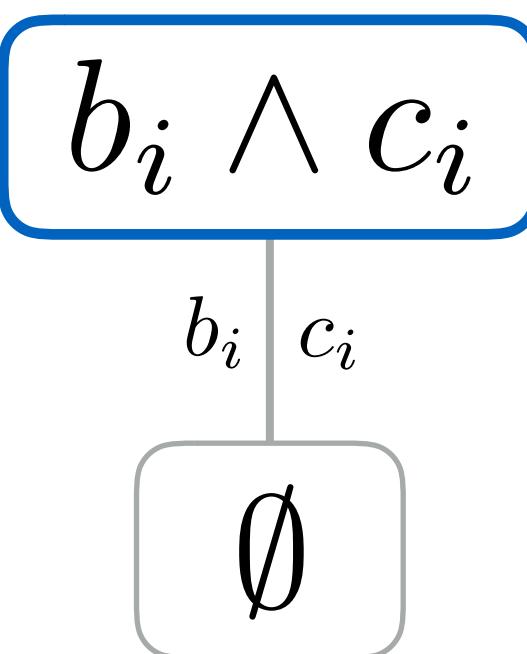
Merge Lattice for a Conjunction

$$a_i = b_i c_i$$



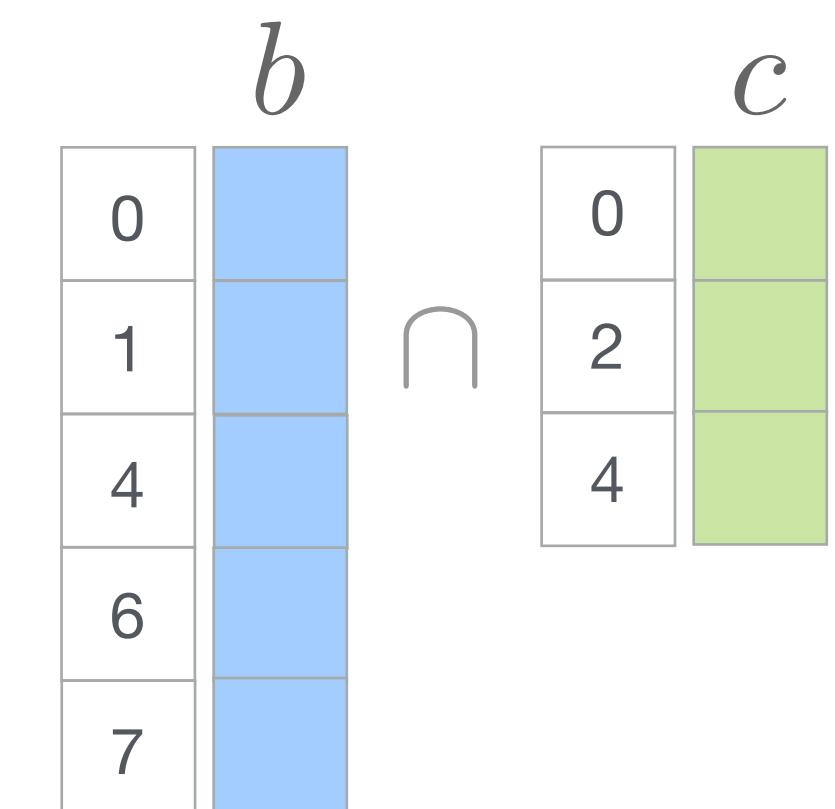
Merge Lattice for a Conjunction

$$a_i = b_i c_i$$



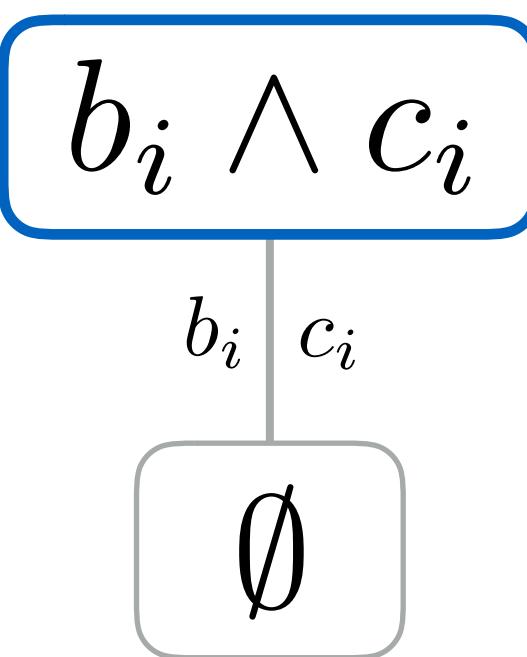
```
int pb1 = b1_pos[0];
int pc1 = c1_pos[0];
while (pb1 < b1_pos[1] && pc1 < c1_pos[1]) {
    int ib = b1_idx[pb1];
    int ic = c1_idx[pc1];
    int i = min(ib, ic);

    if (ib == i) pb1++;
    if (ic == i) pc1++;
}
```

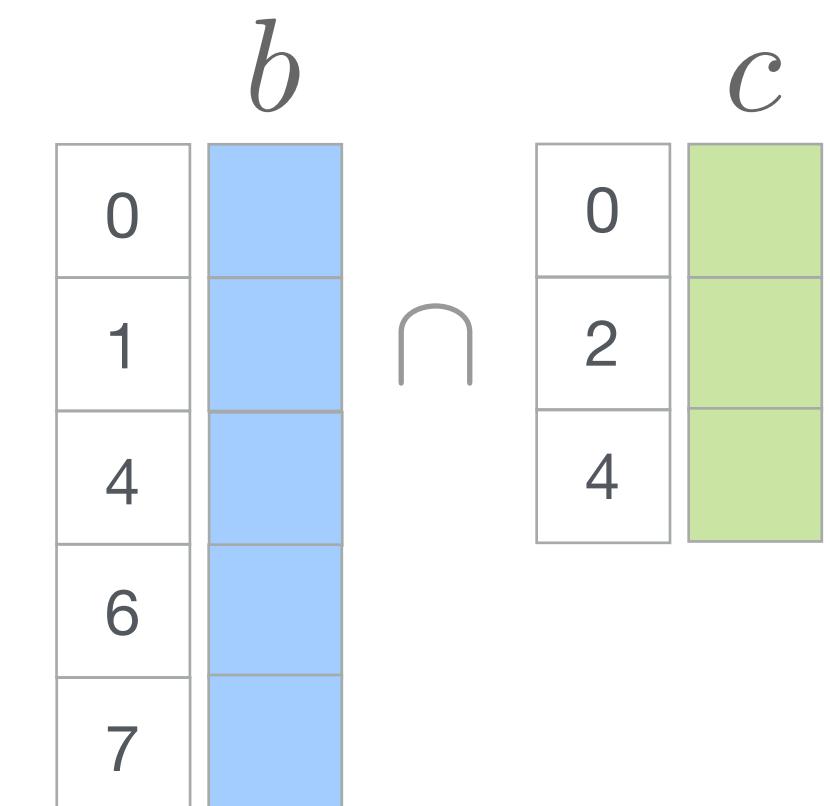


Merge Lattice for a Conjunction

$$a_i = b_i c_i$$

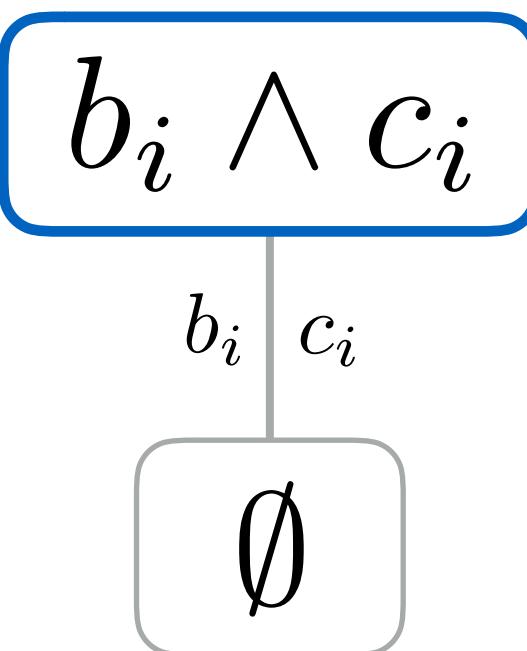


```
int pb1 = b1_pos[0];
int pc1 = c1_pos[0];
while (pb1 < b1_pos[1] && pc1 < c1_pos[1]) {
    int ib = b1_idx[pb1];
    int ic = c1_idx[pc1];
    int i = min(ib, ic);
    if (ib == i && ic == i) {
        a[i] = b[pb1] * c[pc1];
    }
    if (ib == i) pb1++;
    if (ic == i) pc1++;
}
```

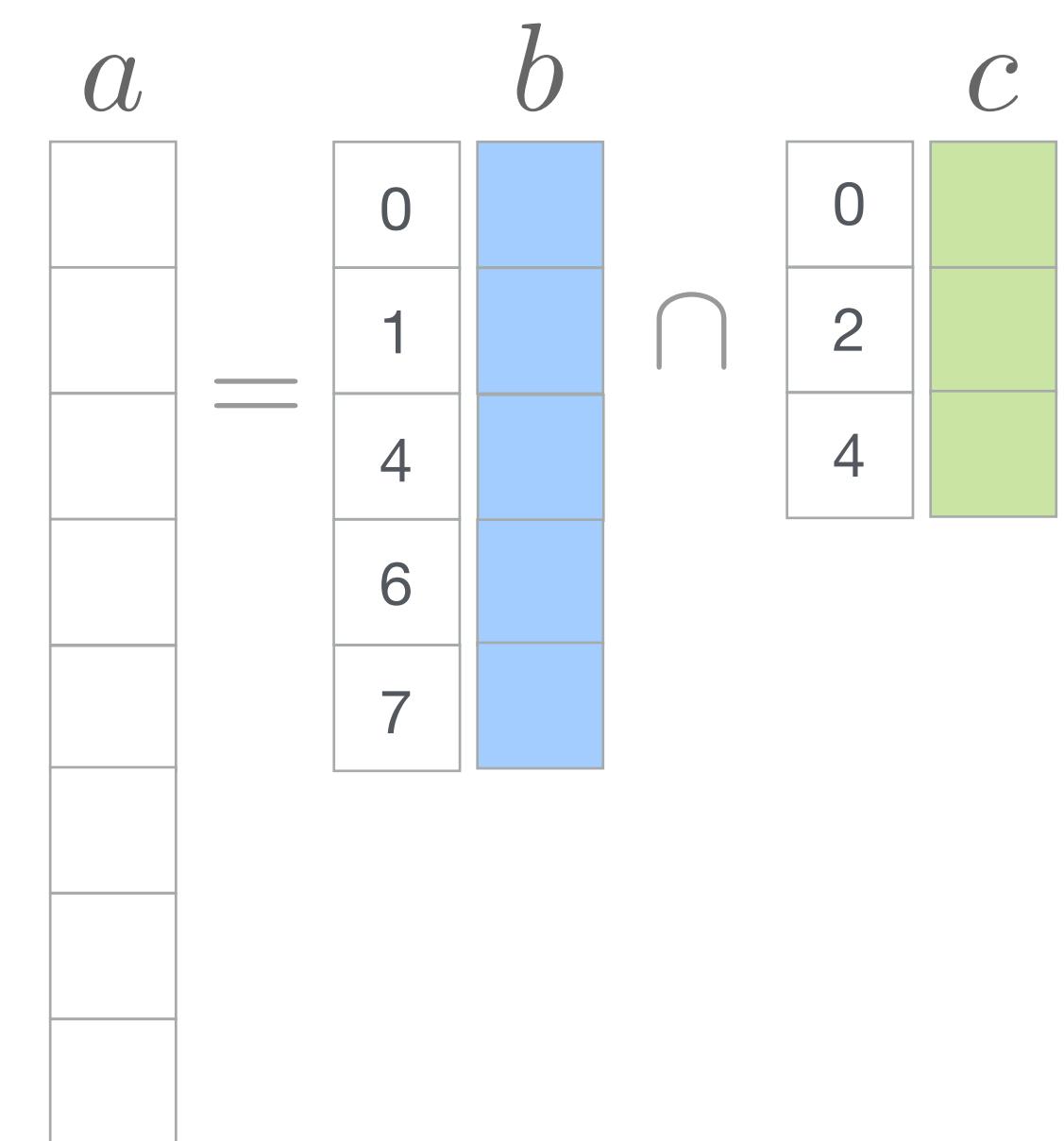


Merge Lattice for a Conjunction

$$a_i = b_i c_i$$



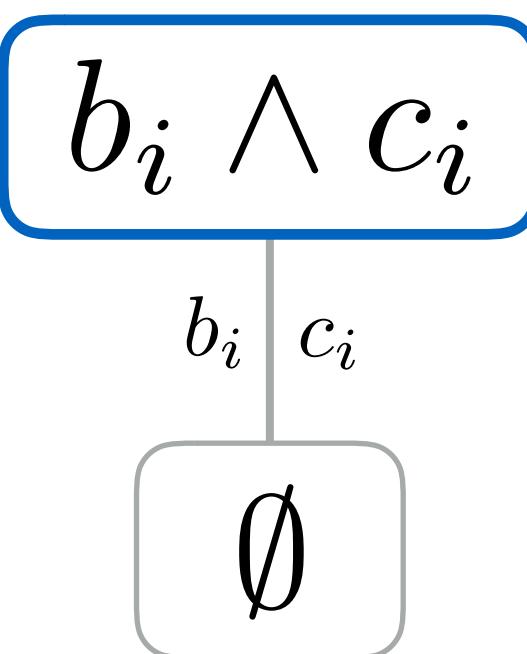
```
int pb1 = b1_pos[0];
int pc1 = c1_pos[0];
while (pb1 < b1_pos[1] && pc1 < c1_pos[1]) {
    int ib = b1_idx[pb1];
    int ic = c1_idx[pc1];
    int i = min(ib, ic);
    if (ib == i && ic == i) {
        a[i] = b[pb1] * c[pc1];
    }
    if (ib == i) pb1++;
    if (ic == i) pc1++;
}
```



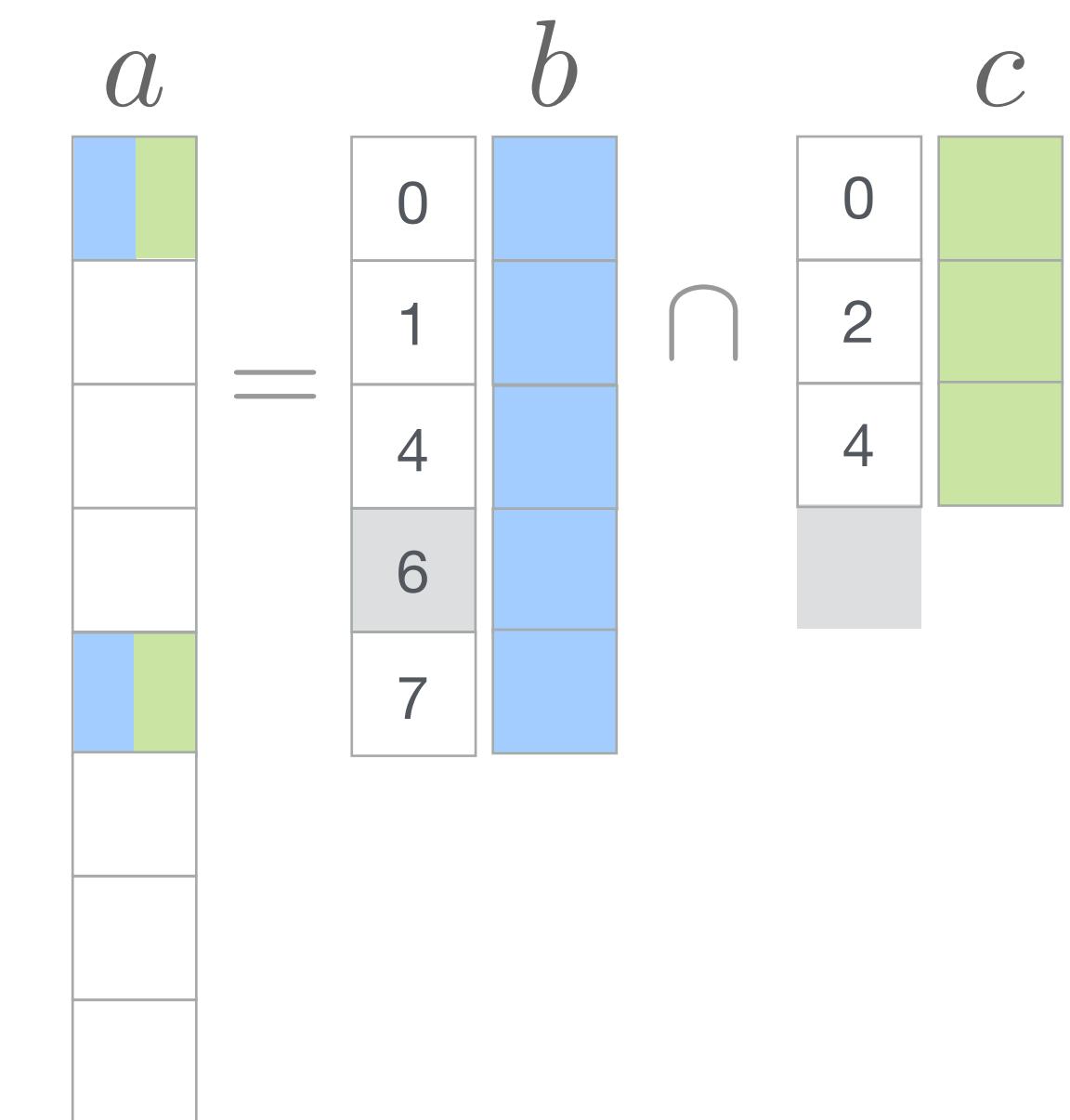
Merge Lattice for a Conjunction

$$a_i = b_i c_i$$

$i = 4$



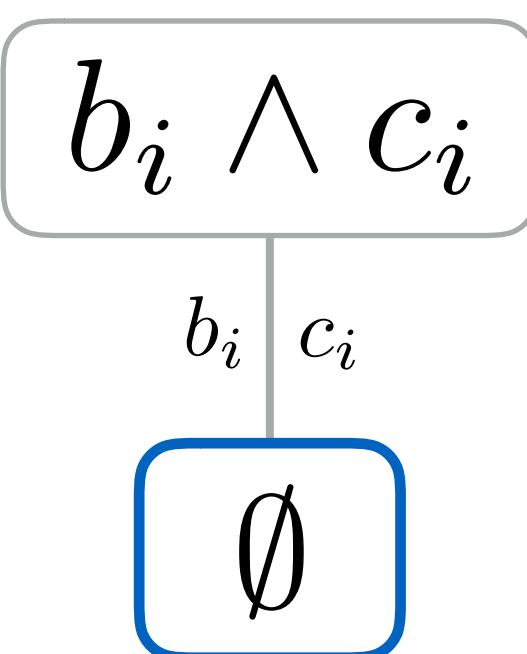
```
int pb1 = b1_pos[0];
int pc1 = c1_pos[0];
while (pb1 < b1_pos[1] && pc1 < c1_pos[1]) {
    int ib = b1_idx[pb1];
    int ic = c1_idx[pc1];
    int i = min(ib, ic);
    if (ib == i && ic == i) {
        a[i] = b[pb1] * c[pc1];
    }
    if (ib == i) pb1++;
    if (ic == i) pc1++;
}
```



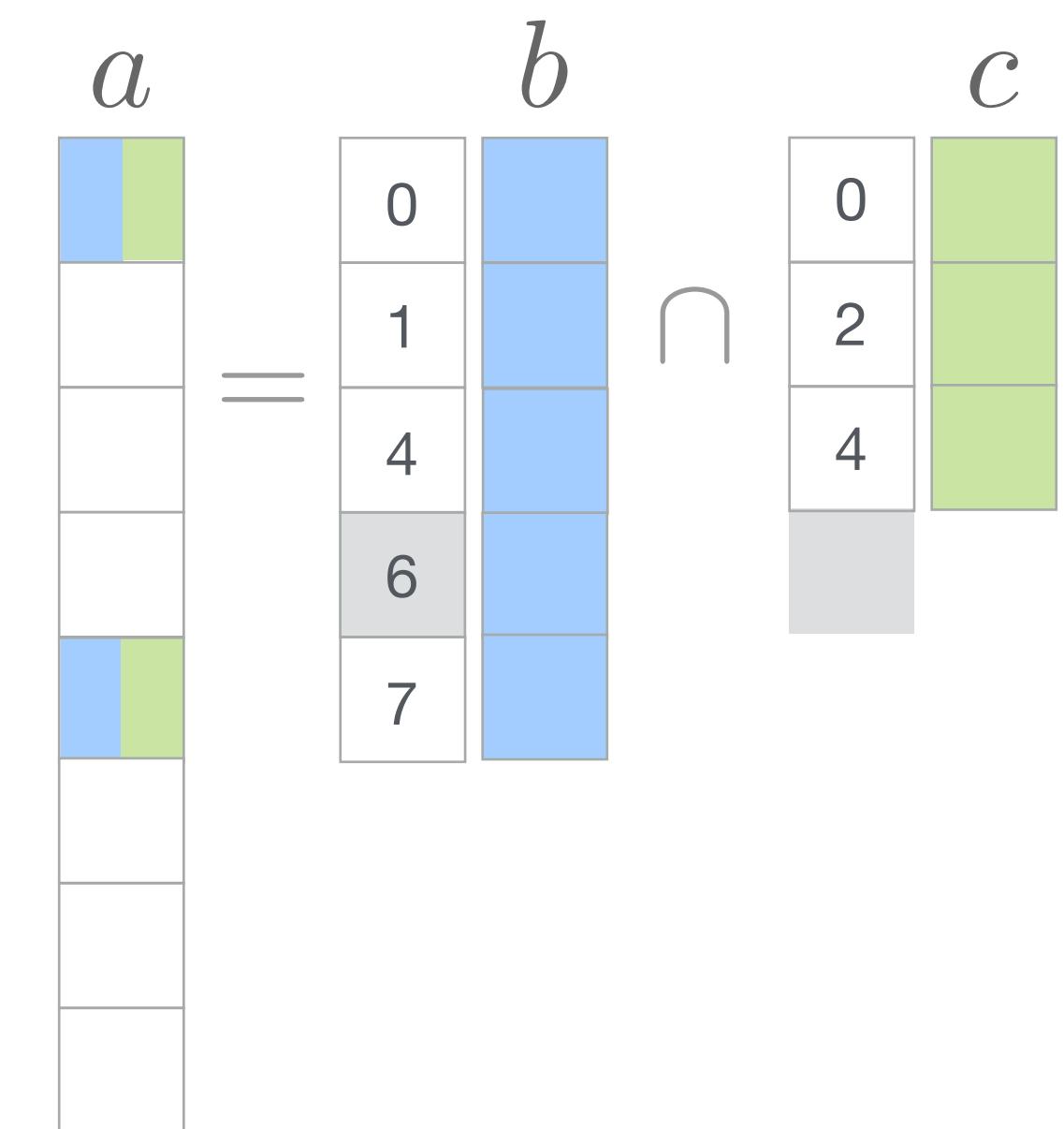
Merge Lattice for a Conjunction

$$a_i = b_i c_i$$

$i = 4$

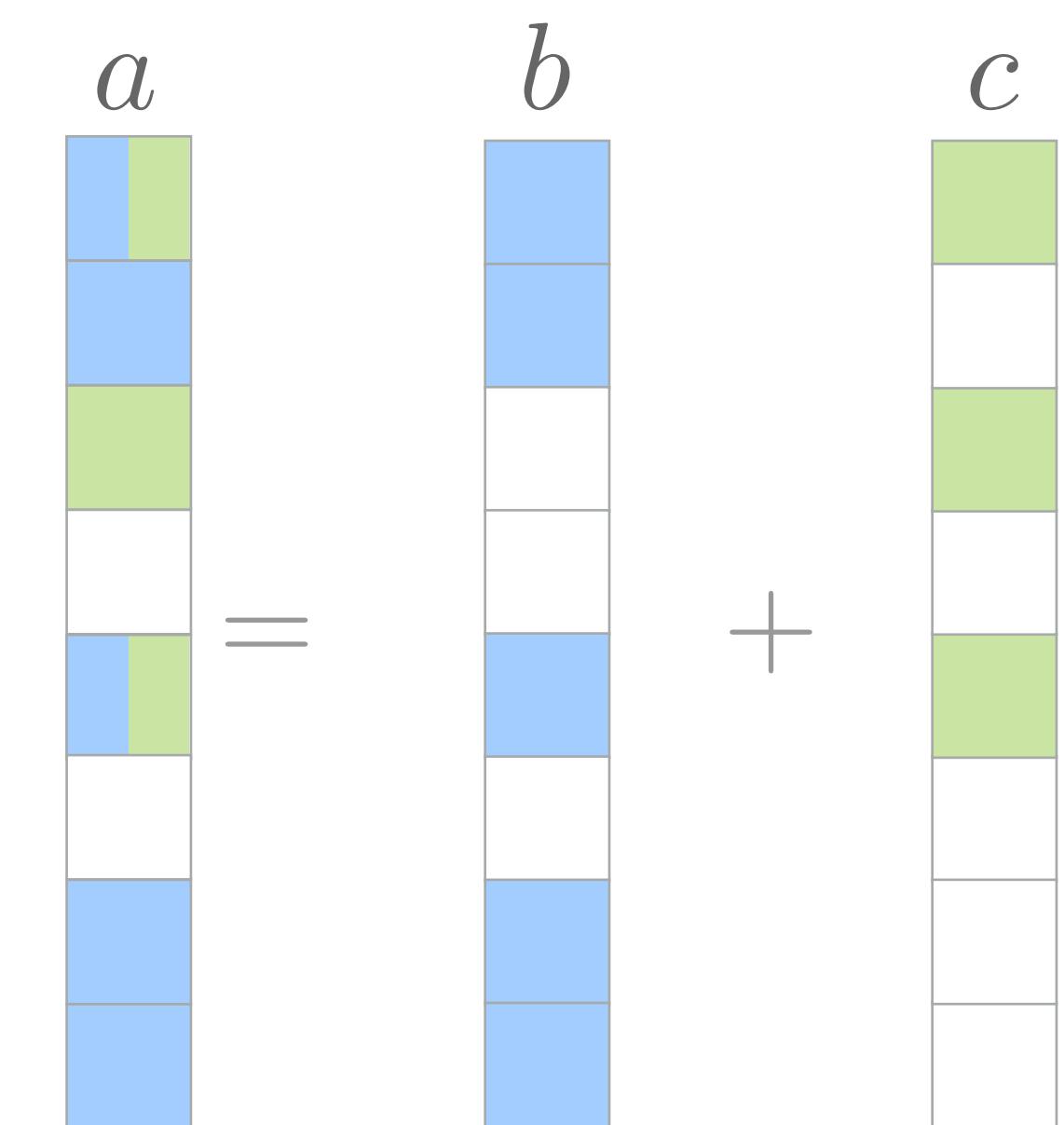


```
int pb1 = b1_pos[0];
int pc1 = c1_pos[0];
while (pb1 < b1_pos[1] && pc1 < c1_pos[1]) {
    int ib = b1_idx[pb1];
    int ic = c1_idx[pc1];
    int i = min(ib, ic);
    if (ib == i && ic == i) {
        a[i] = b[pb1] * c[pc1];
    }
    if (ib == i) pb1++;
    if (ic == i) pc1++;
}
```



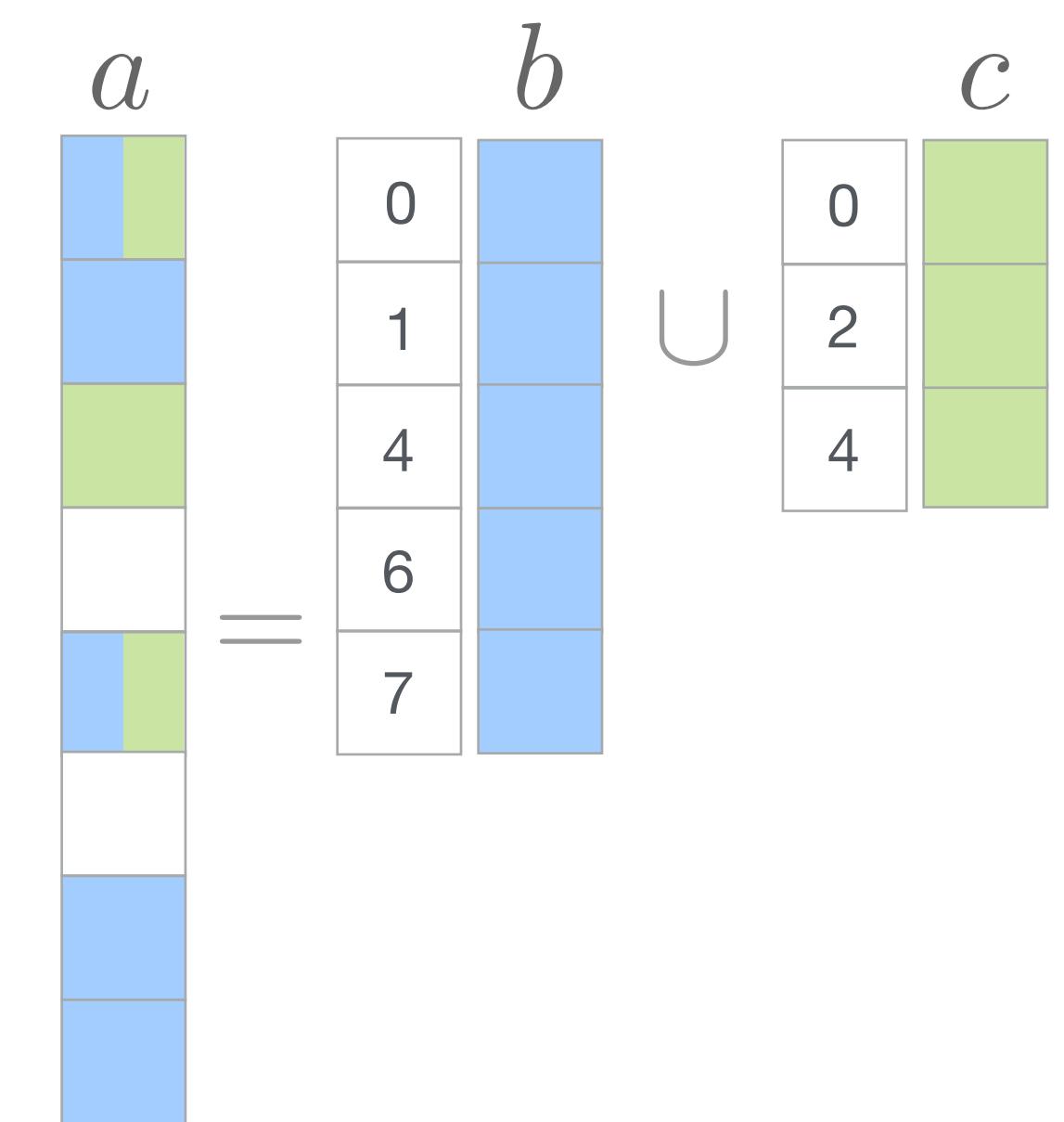
Merge Lattice for a Disjunction

$$a_i = b_i + c_i$$



Merge Lattice for a Disjunction

$$a_i = b_i + c_i$$



Merge Lattice for a Disjunction

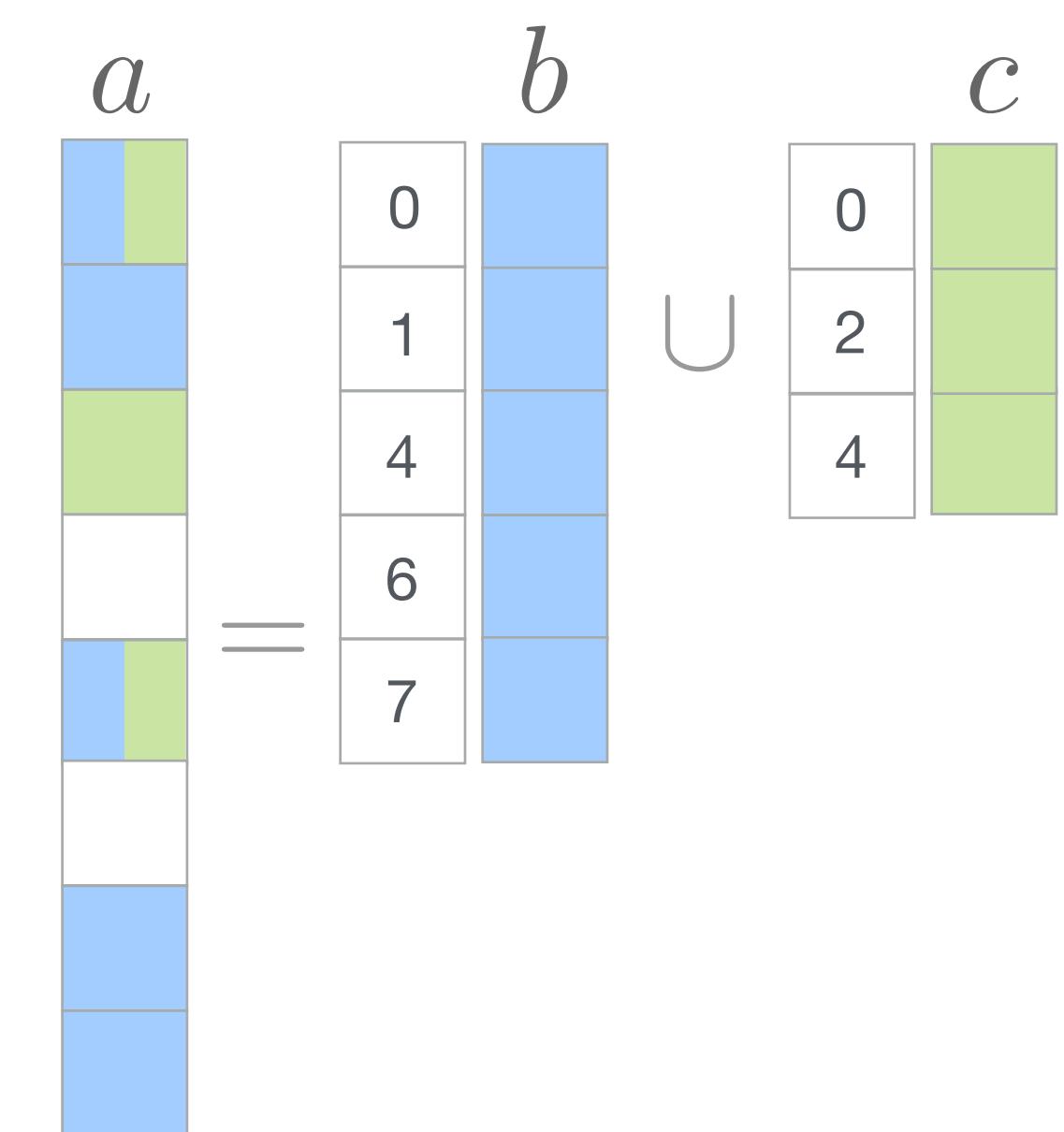
$$a_i = b_i + c_i$$



Merge Lattice for a Disjunction

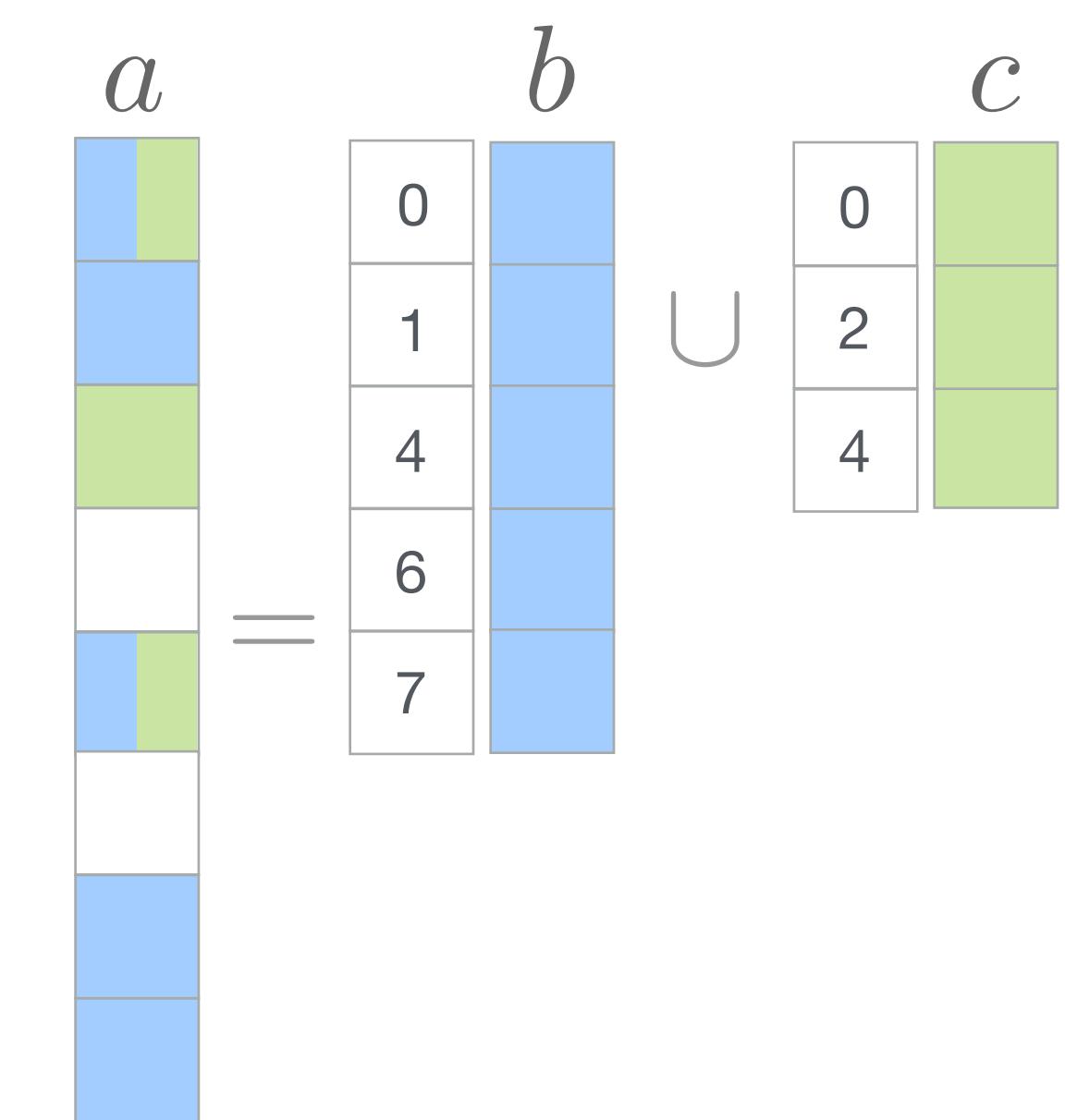
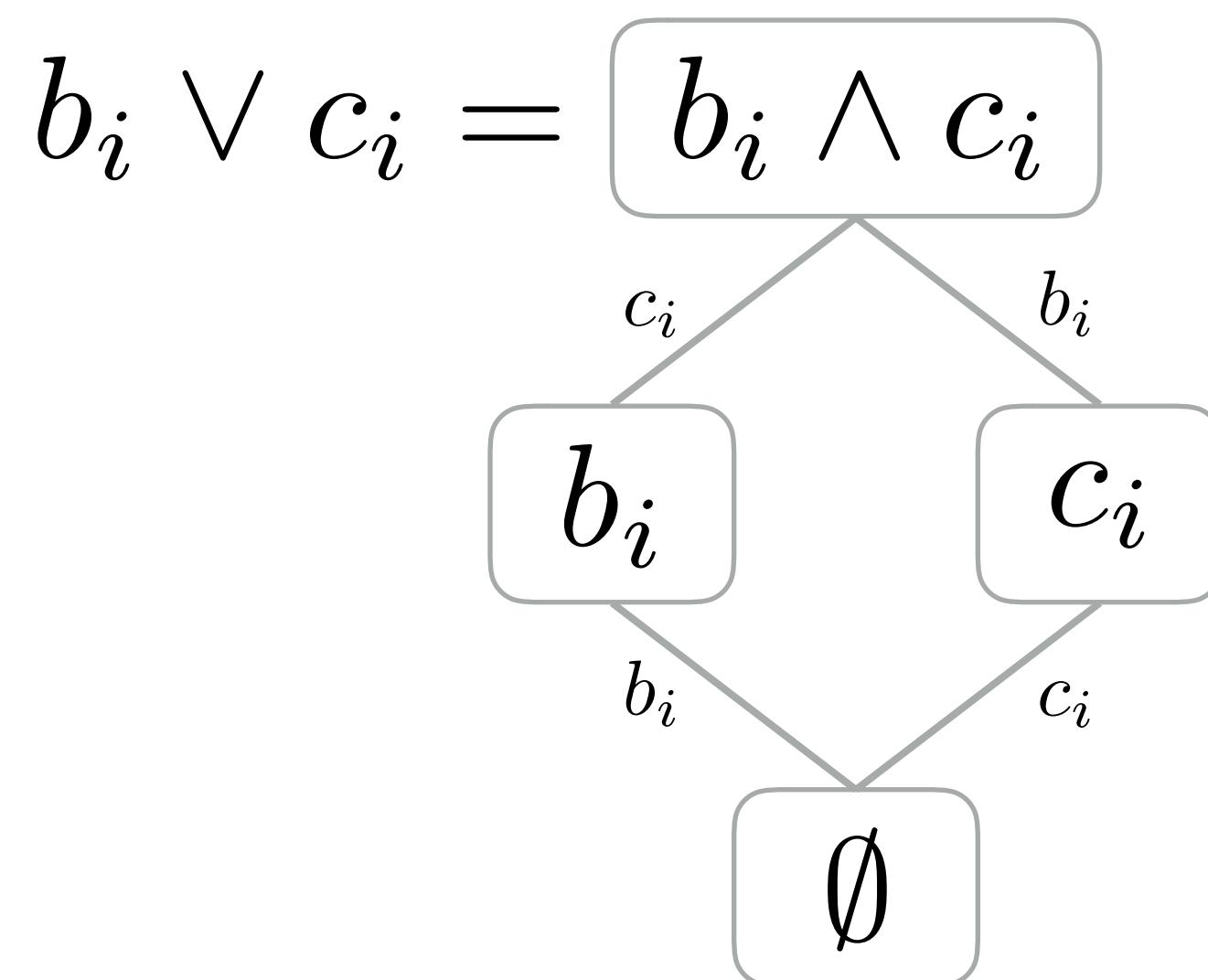
$$a_i = b_i + c_i$$

$$b_i \vee c_i = \boxed{b_i \wedge c_i} \vee b_i \vee c_i$$



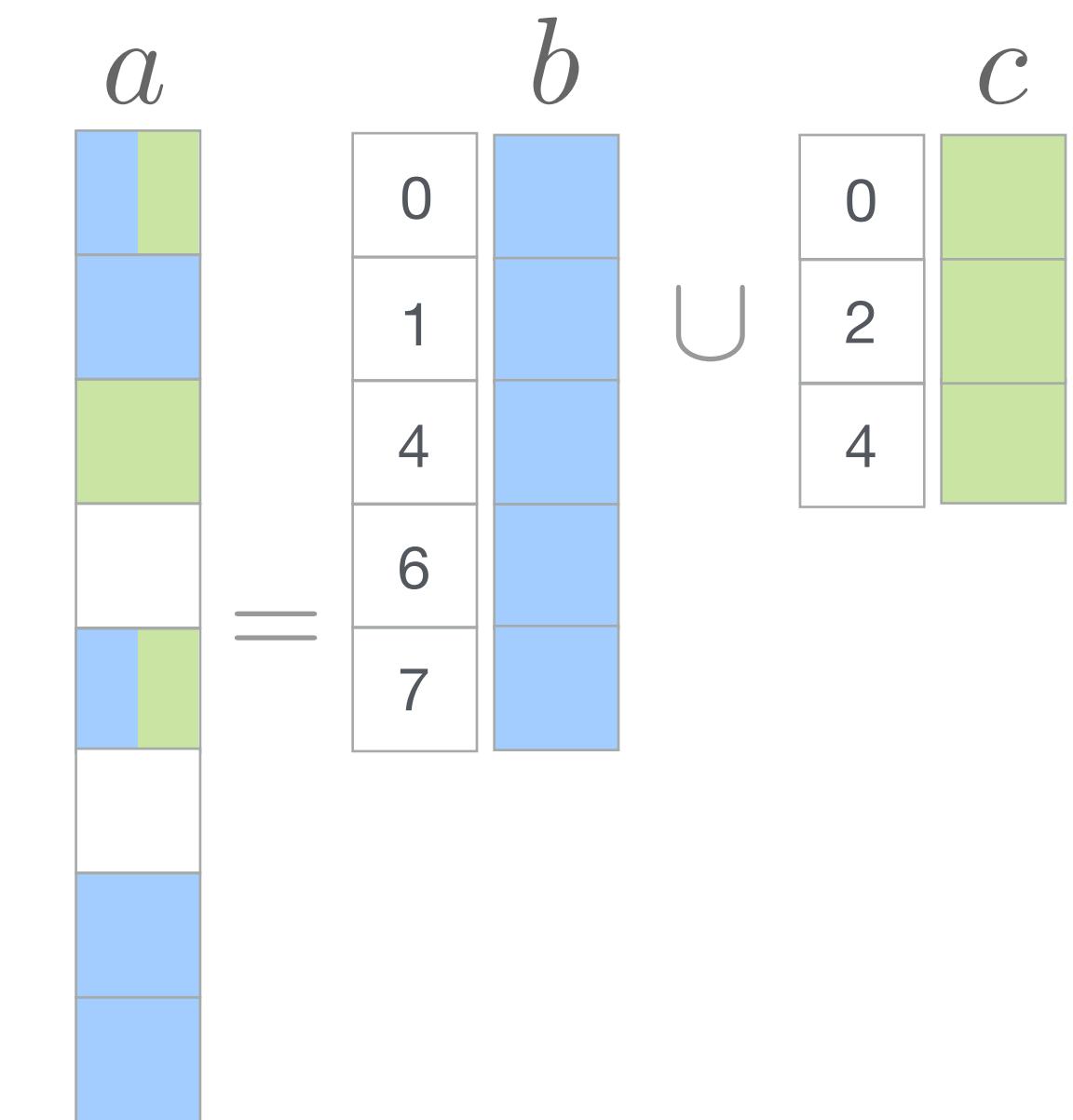
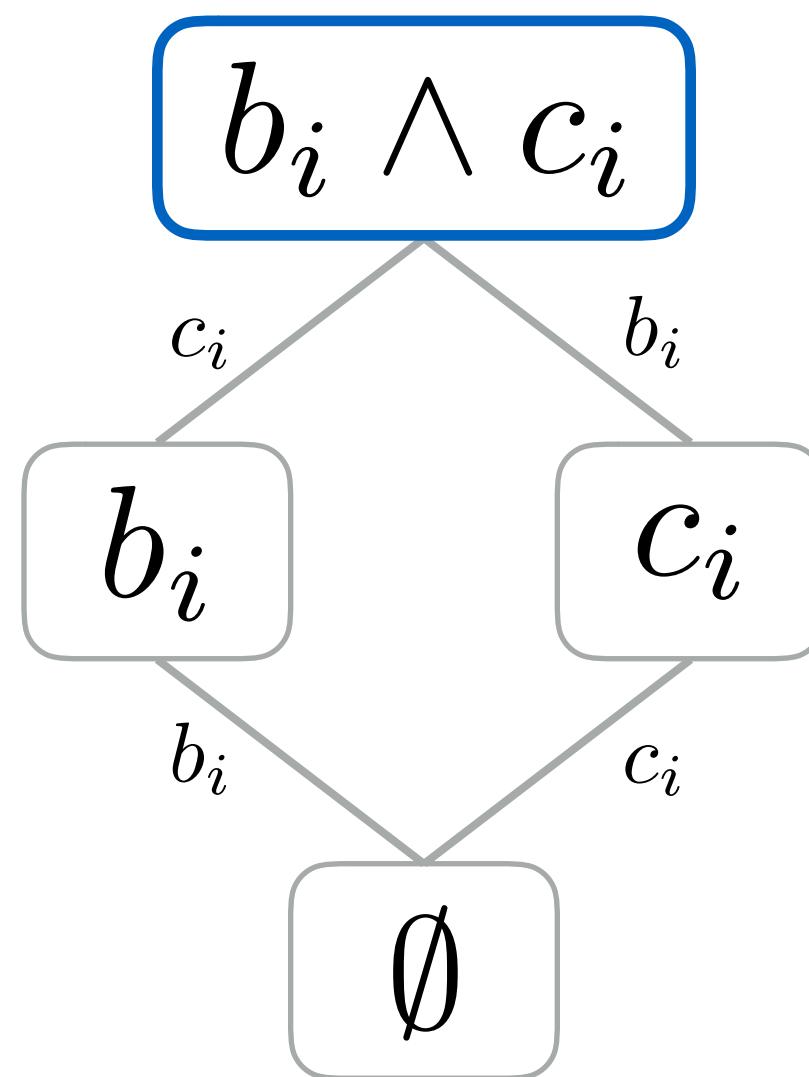
Merge Lattice for a Disjunction

$$a_i = b_i + c_i$$



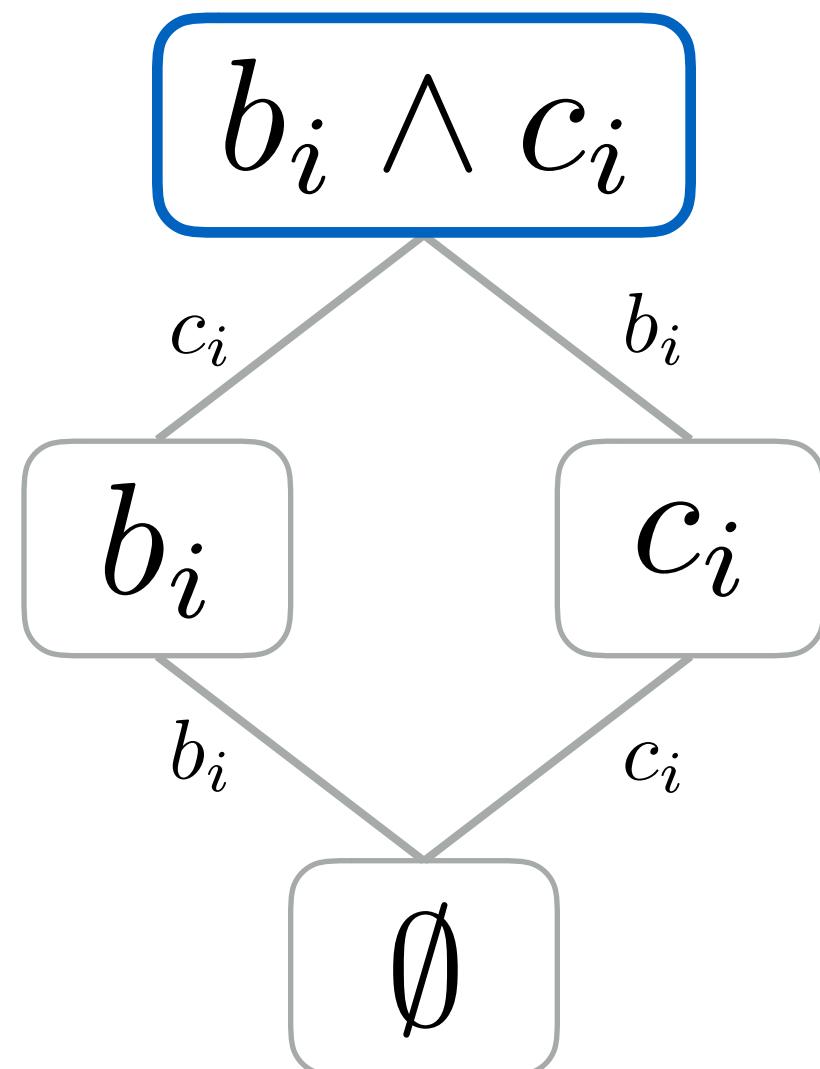
Merge Lattice for a Disjunction

$$a_i = b_i + c_i$$



Merge Lattice for a Disjunction

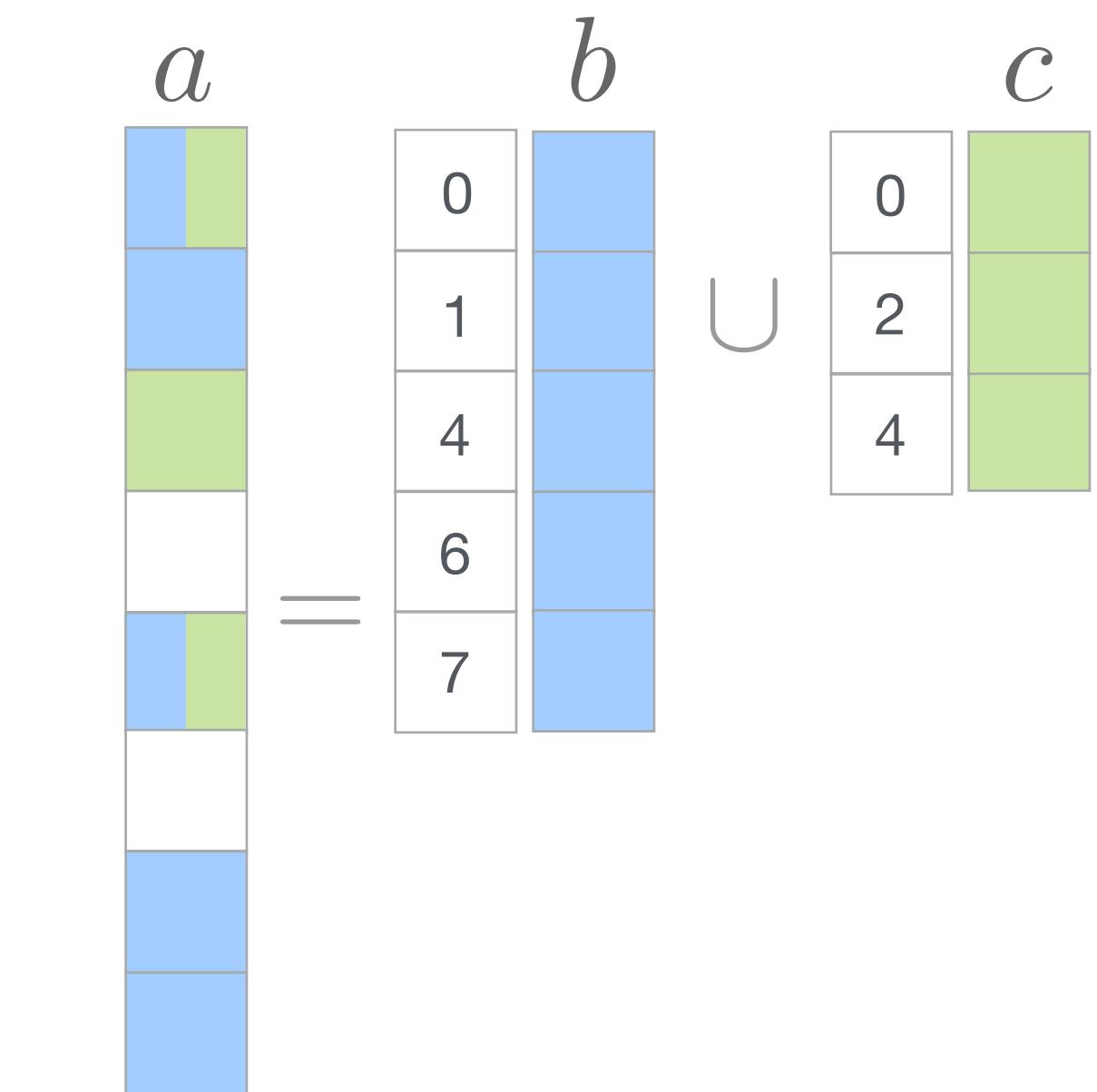
$$a_i = b_i + c_i$$



```
int pb1 = b1_pos[0];
int pc1 = c1_pos[0];
while (pb1 < b1_pos[1] && pc1 < c1_pos[1]) {
    int ib = b1_idx[pb1];
    int ic = c1_idx[pc1];
    int i = min(ib, ic);

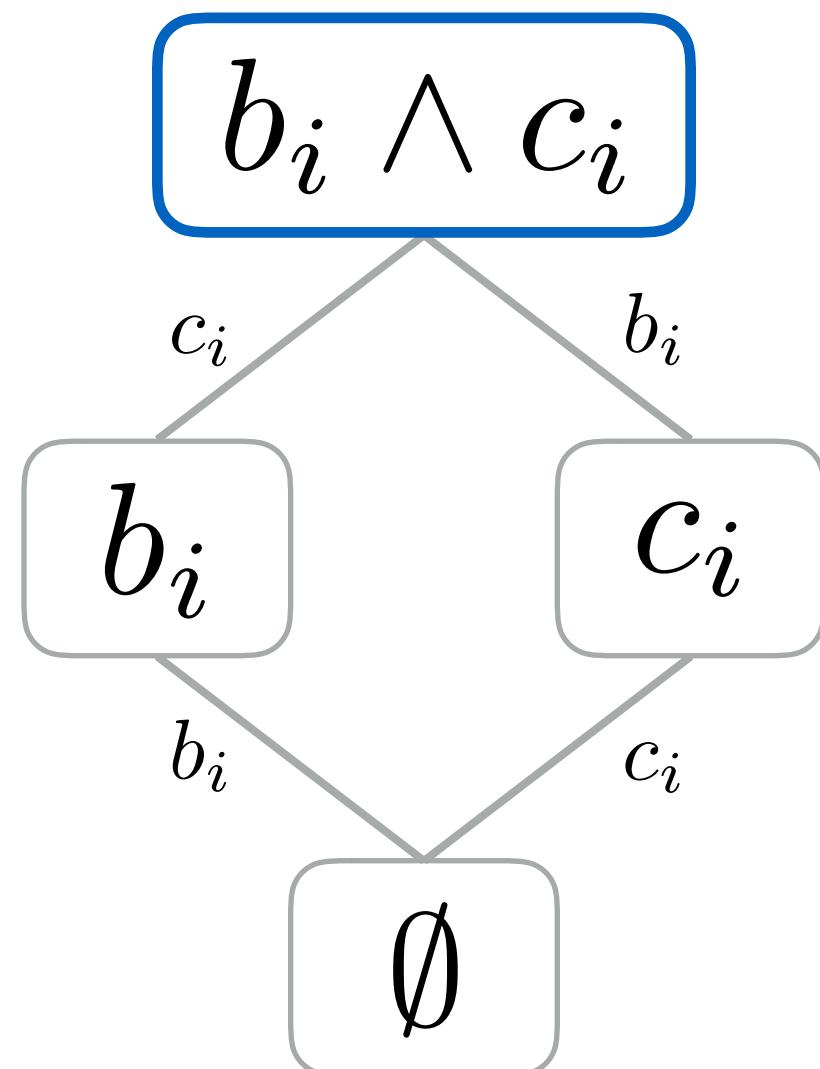
    if (ib == i) pb1++;
    if (ic == i) pc1++;

}
```



Merge Lattice for a Disjunction

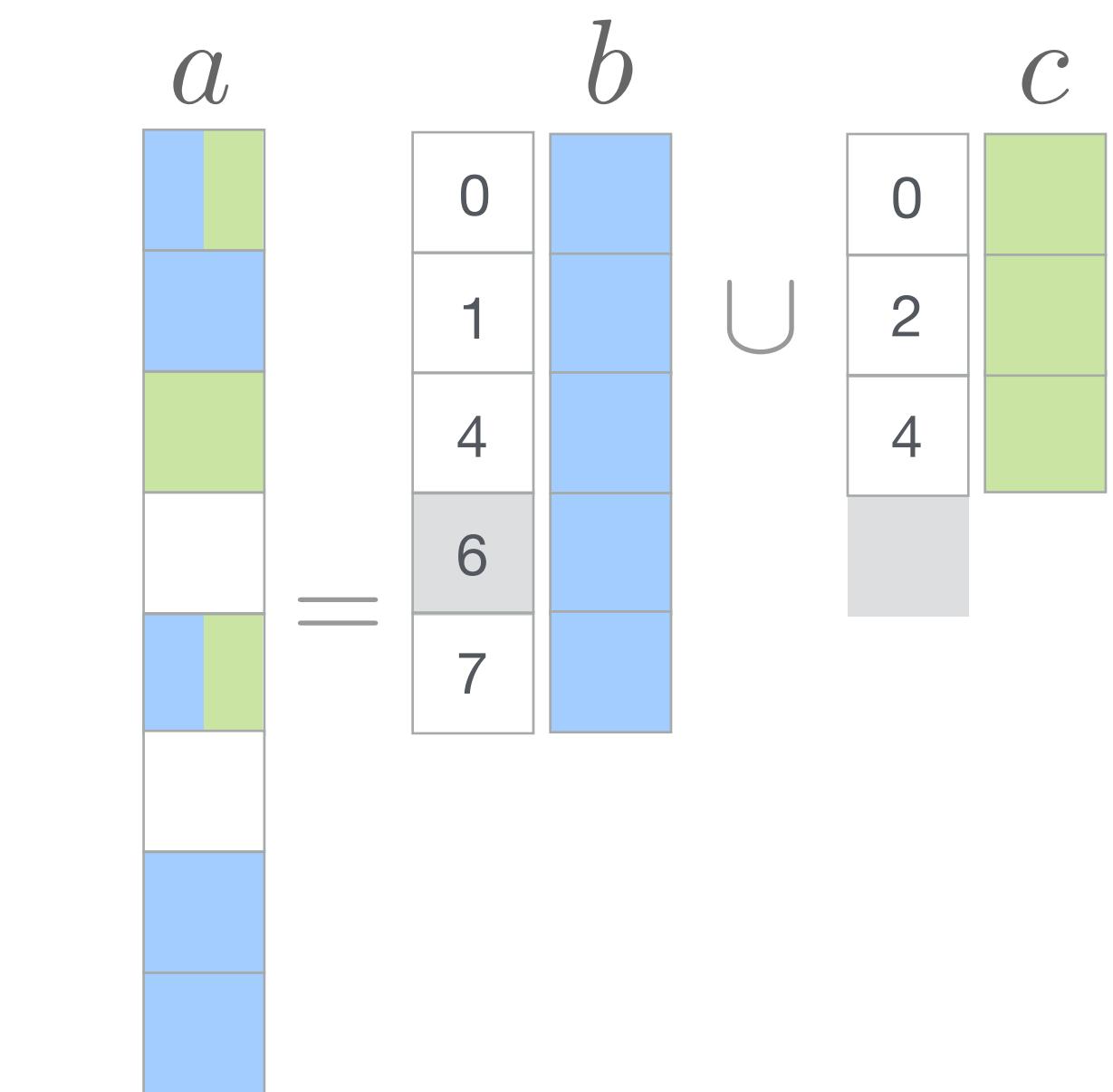
$$a_i = b_i + c_i$$



```
int pb1 = b1_pos[0];
int pc1 = c1_pos[0];
while (pb1 < b1_pos[1] && pc1 < c1_pos[1]) {
    int ib = b1_idx[pb1];
    int ic = c1_idx[pc1];
    int i = min(ib, ic);

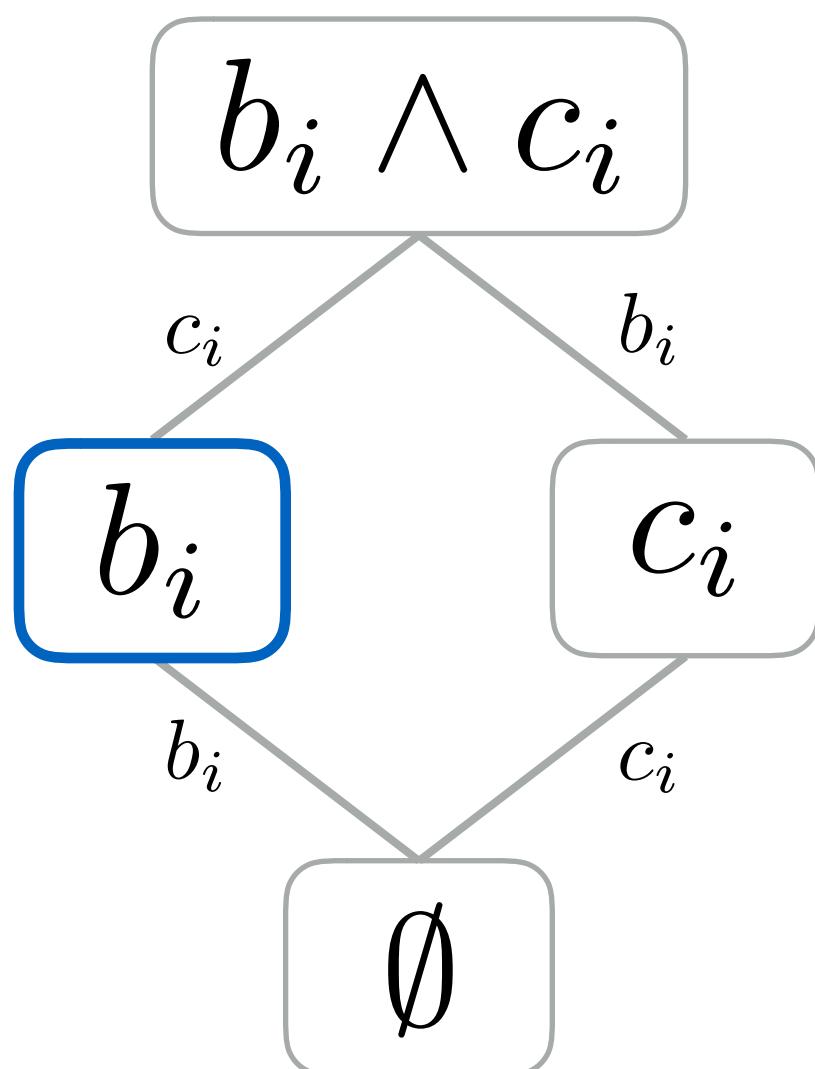
    if (ib == i) pb1++;
    if (ic == i) pc1++;

}
```



Merge Lattice for a Disjunction

$$a_i = b_i + c_i$$

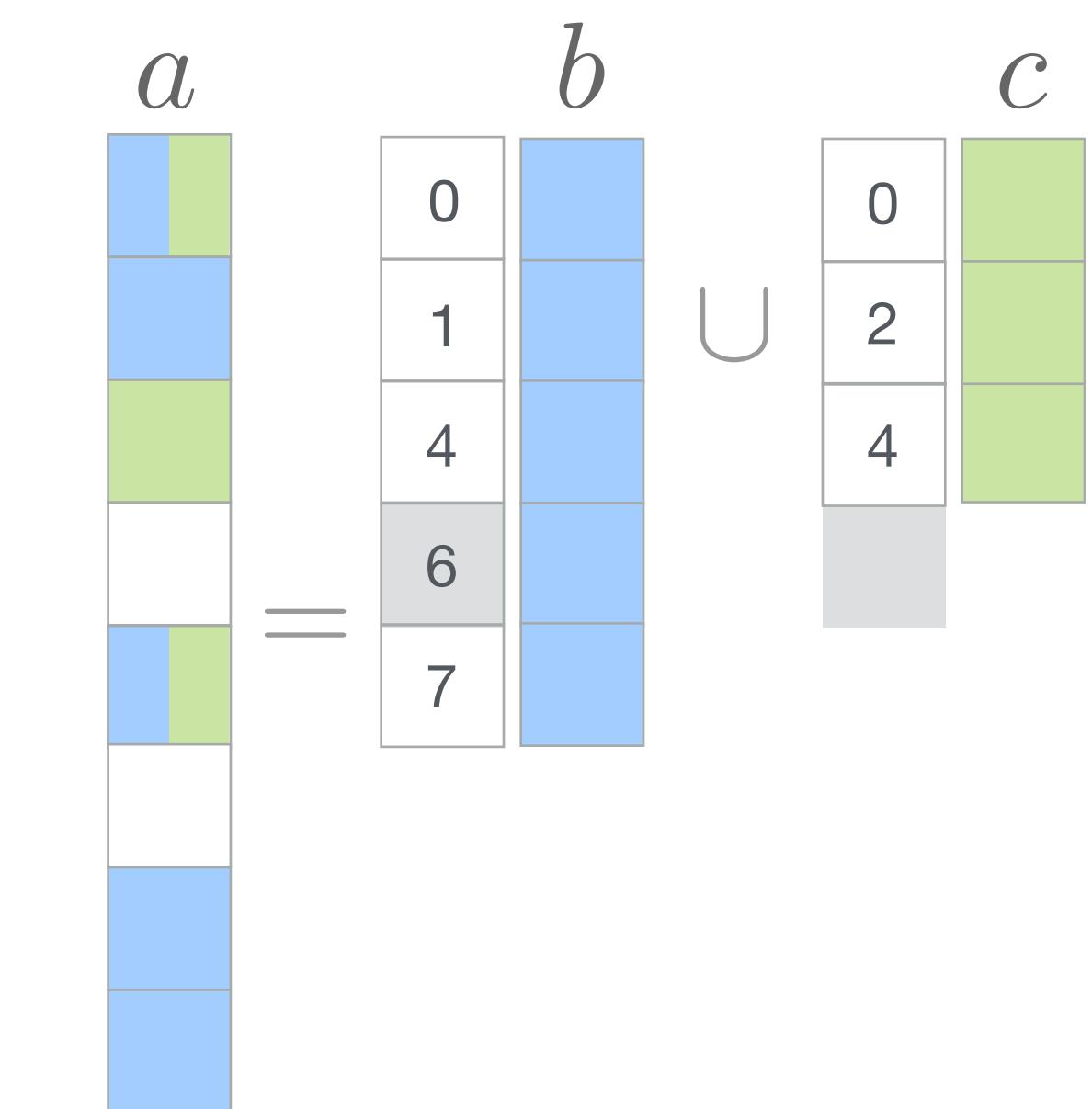


```

int pb1 = b1_pos[0];
int pc1 = c1_pos[0];
while (pb1 < b1_pos[1] && pc1 < c1_pos[1]) {
    int ib = b1_idx[pb1];
    int ic = c1_idx[pc1];
    int i = min(ib, ic);

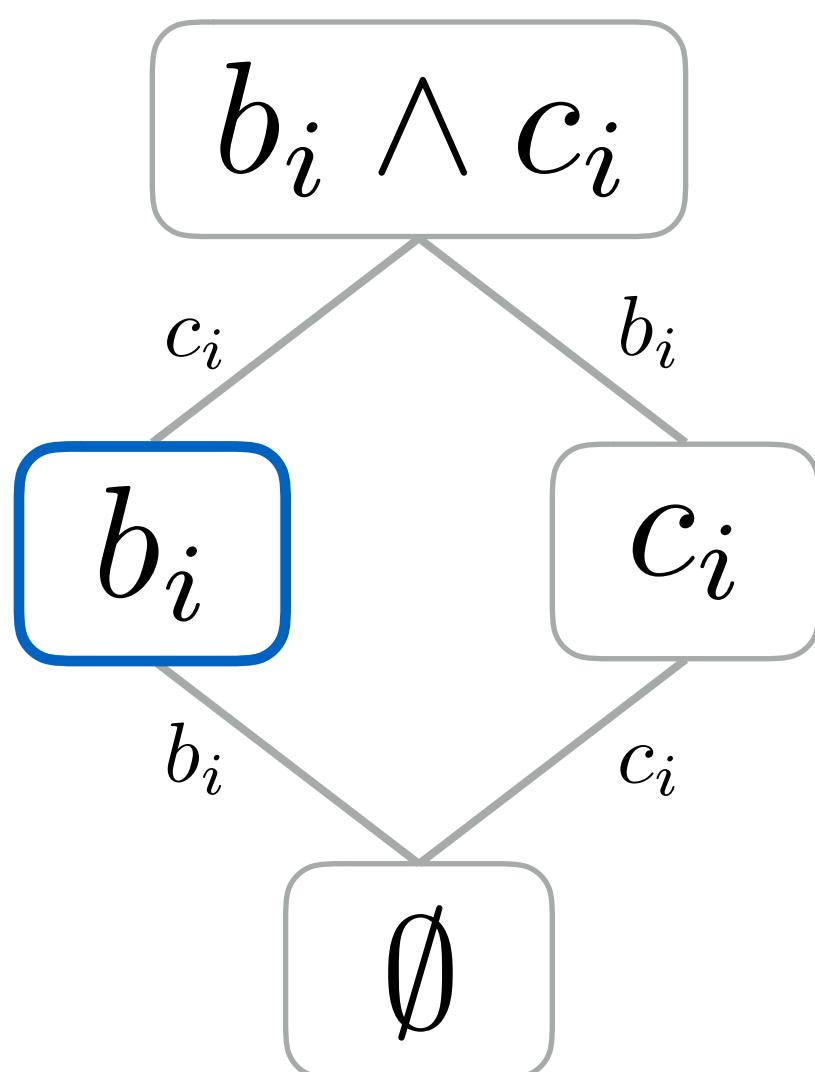
    if (ib == i) pb1++;
    if (ic == i) pc1++;

}
  
```



Merge Lattice for a Disjunction

$$a_i = b_i + c_i$$

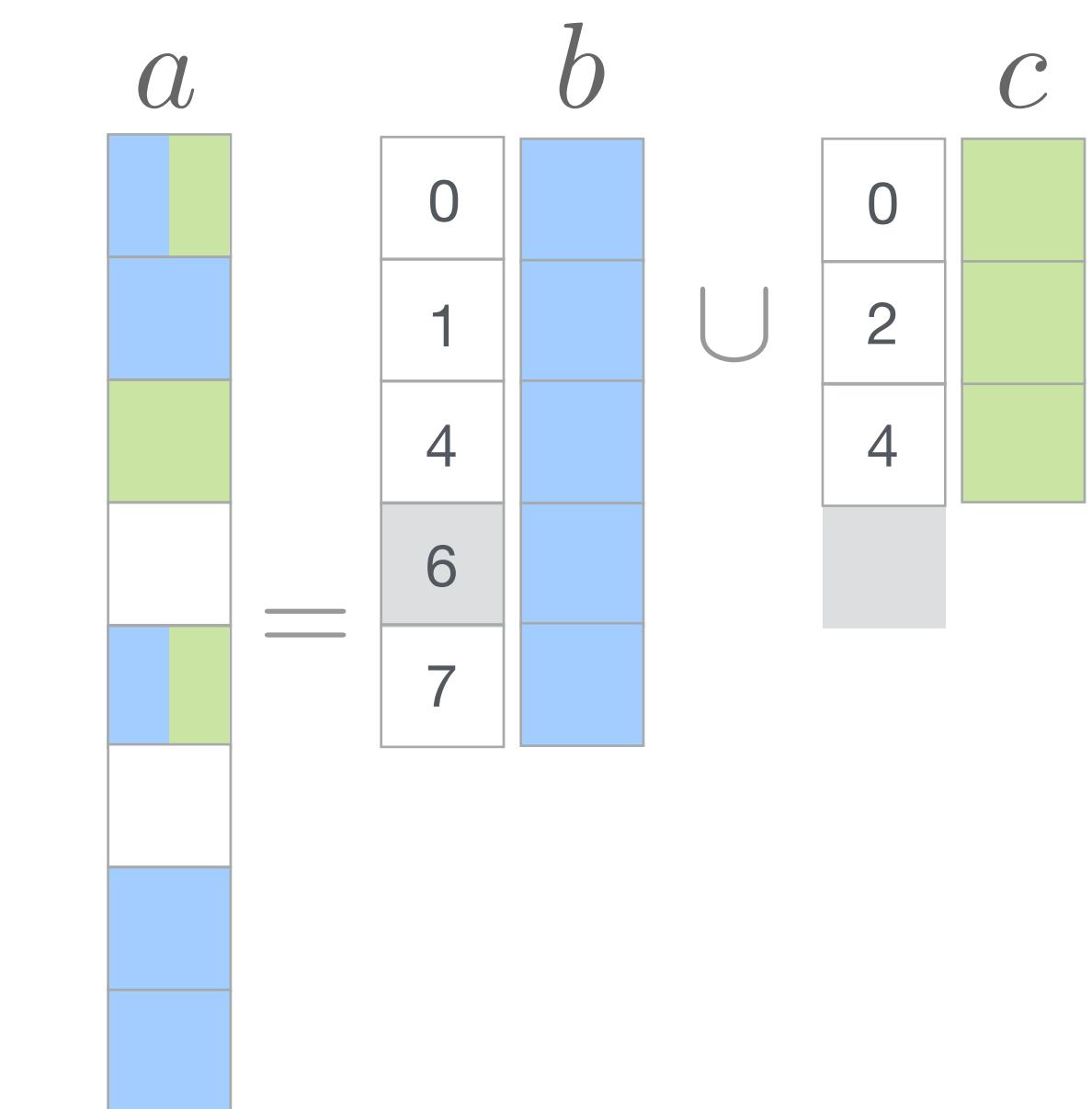


```

int pb1 = b1_pos[0];
int pc1 = c1_pos[0];
while (pb1 < b1_pos[1] && pc1 < c1_pos[1]) {
    int ib = b1_idx[pb1];
    int ic = c1_idx[pc1];
    int i = min(ib, ic);

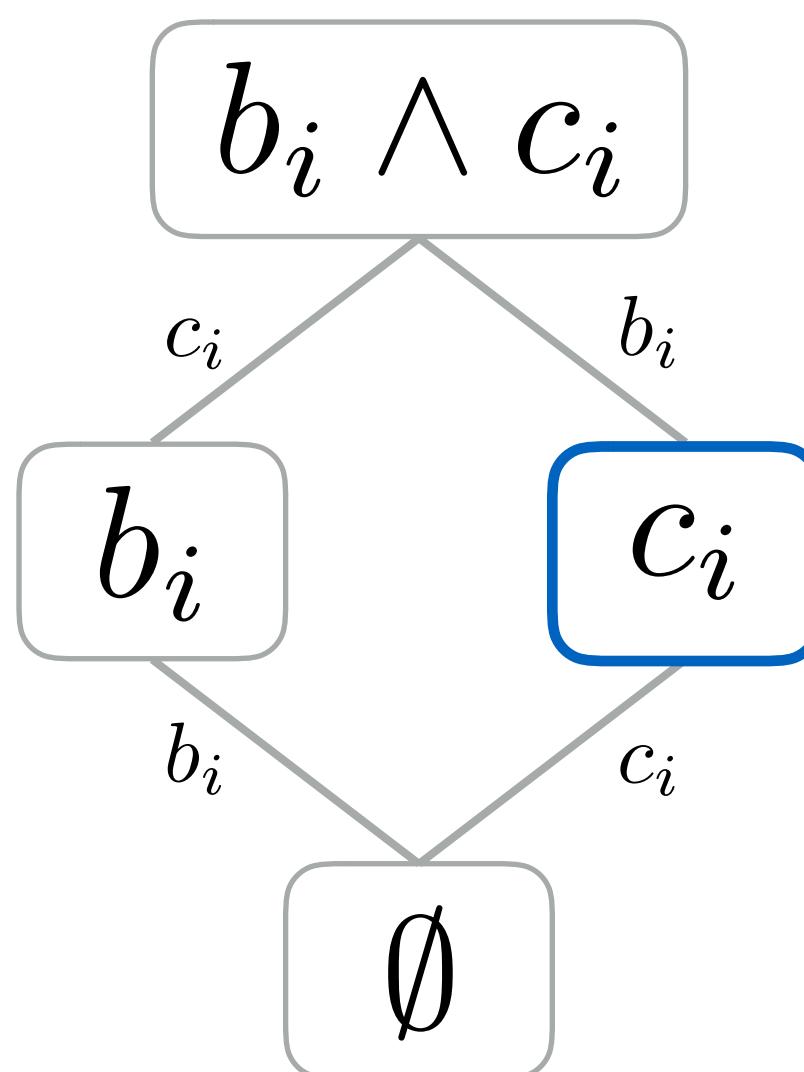
    if (ib == i) pb1++;
    if (ic == i) pc1++;

    while (pb1 < b1_pos[1]) {
        int i = b1_idx[pb1];
        a[i] = b[pb1++];
    }
}
  
```



Merge Lattice for a Disjunction

$$a_i = b_i + c_i$$



```

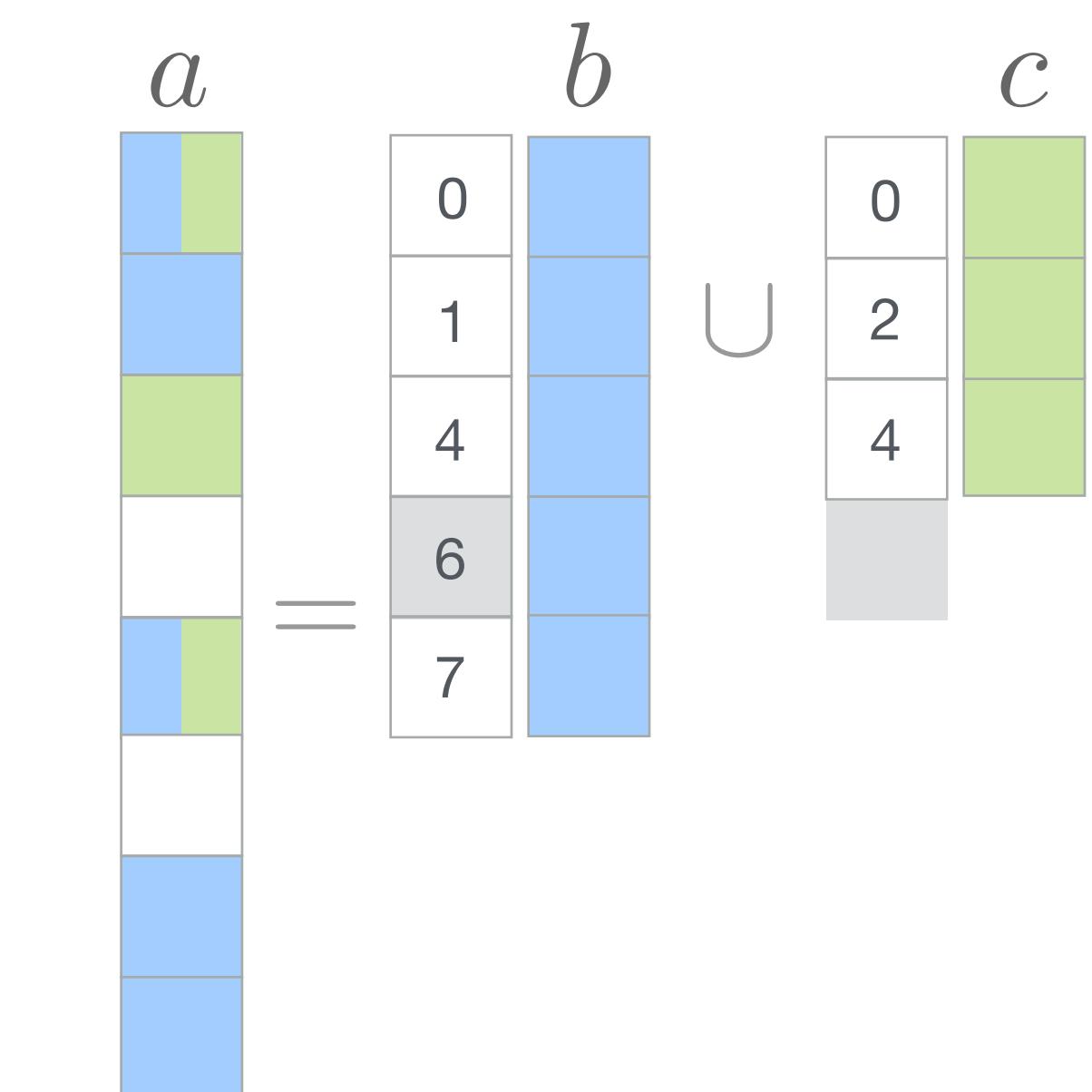
int pb1 = b1_pos[0];
int pc1 = c1_pos[0];
while (pb1 < b1_pos[1] && pc1 < c1_pos[1]) {
    int ib = b1_idx[pb1];
    int ic = c1_idx[pc1];
    int i = min(ib, ic);

    if (ib == i) pb1++;
    if (ic == i) pc1++;

}

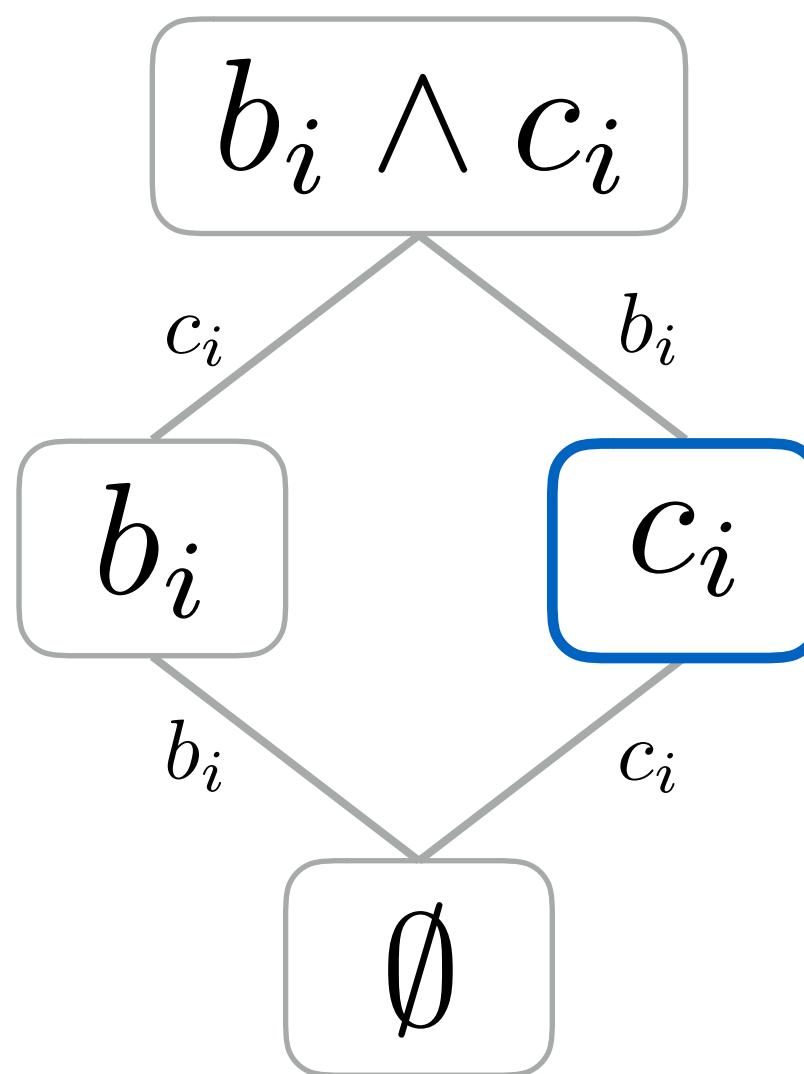
while (pb1 < b1_pos[1]) {
    int i = b1_idx[pb1];
    a[i] = b[pb1++];
}

while (pc1 < c1_pos[1]) {
    int i = c1_idx[pc1];
    a[i] = c[pc1++];
}
  
```



Merge Lattice for a Disjunction

$$a_i = b_i + c_i$$



```

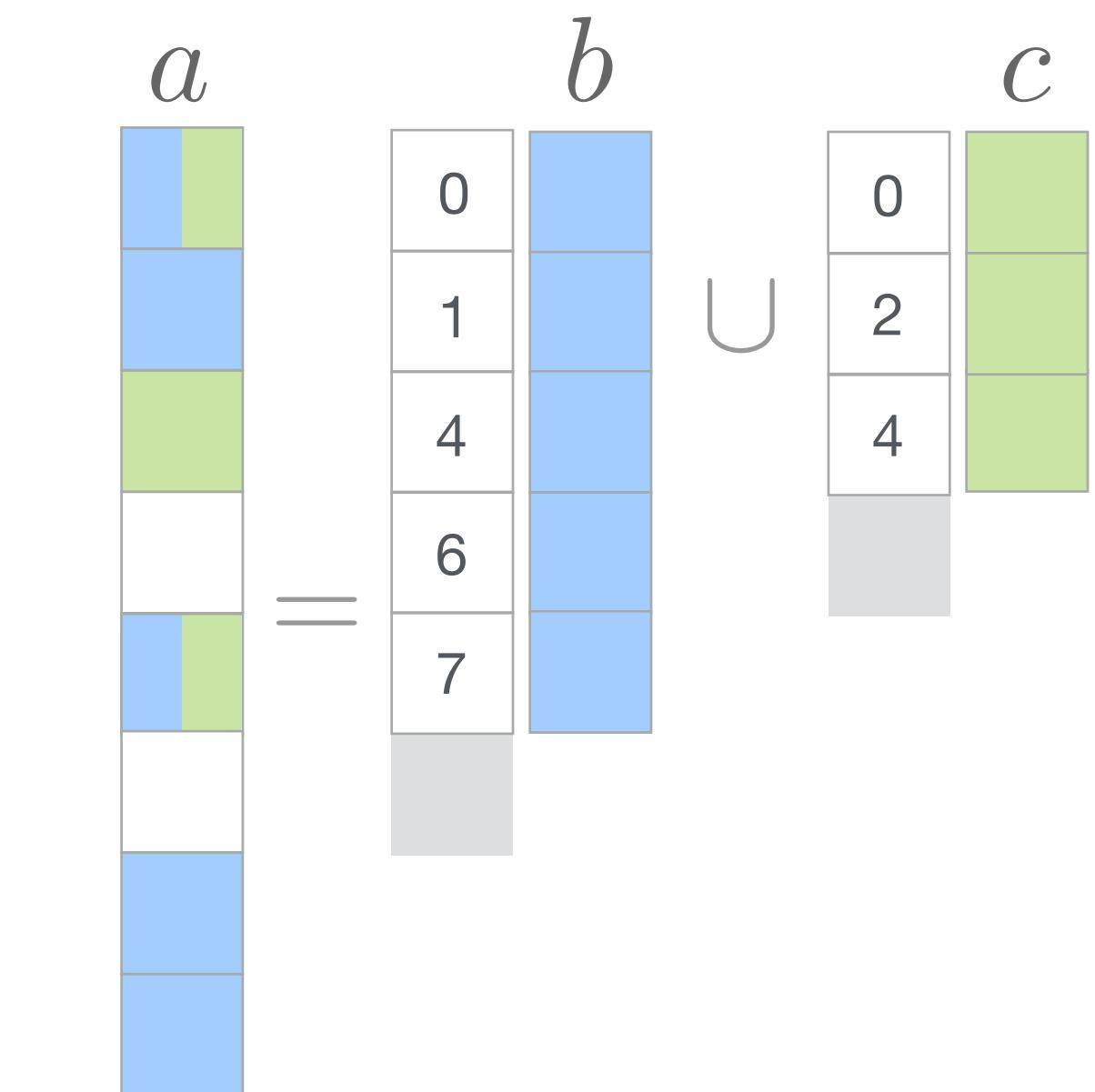
int pb1 = b1_pos[0];
int pc1 = c1_pos[0];
while (pb1 < b1_pos[1] && pc1 < c1_pos[1]) {
    int ib = b1_idx[pb1];
    int ic = c1_idx[pc1];
    int i = min(ib, ic);

    if (ib == i) pb1++;
    if (ic == i) pc1++;

}

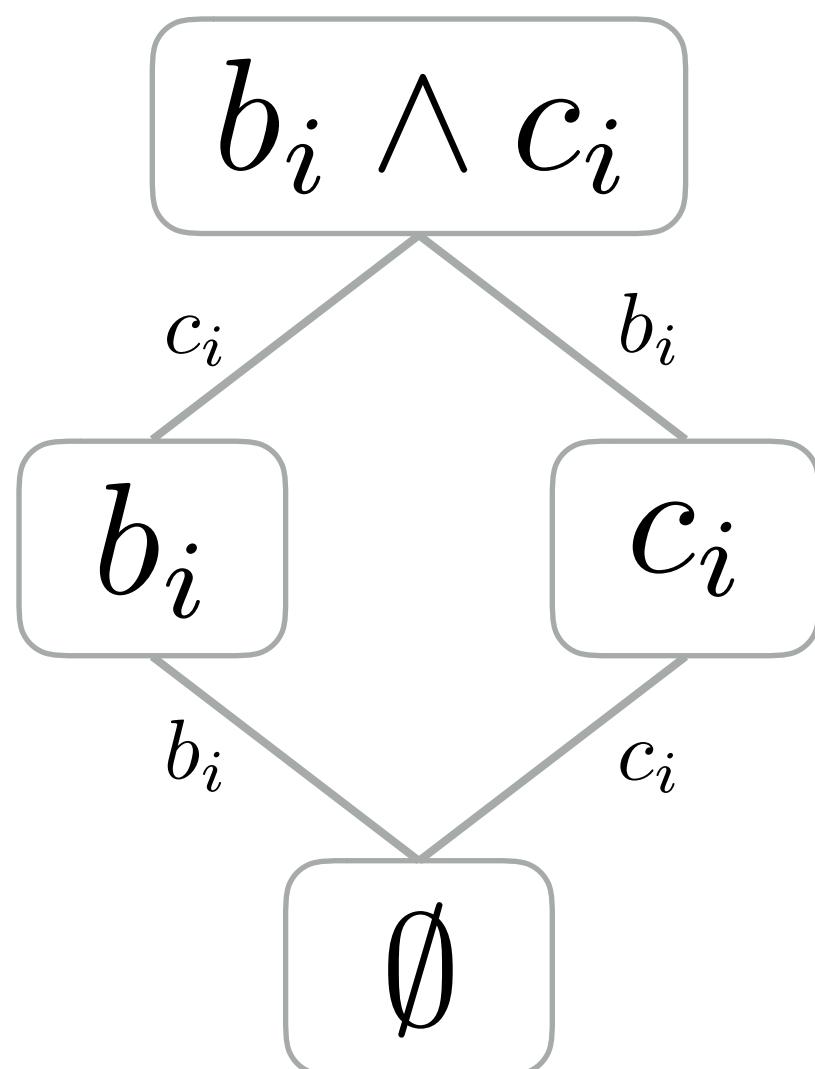
while (pb1 < b1_pos[1]) {
    int i = b1_idx[pb1];
    a[i] = b[pb1++];
}

while (pc1 < c1_pos[1]) {
    int i = c1_idx[pc1];
    a[i] = c[pc1++];
}
  
```



Merge Lattice for a Disjunction

$$a_i = b_i + c_i$$



```

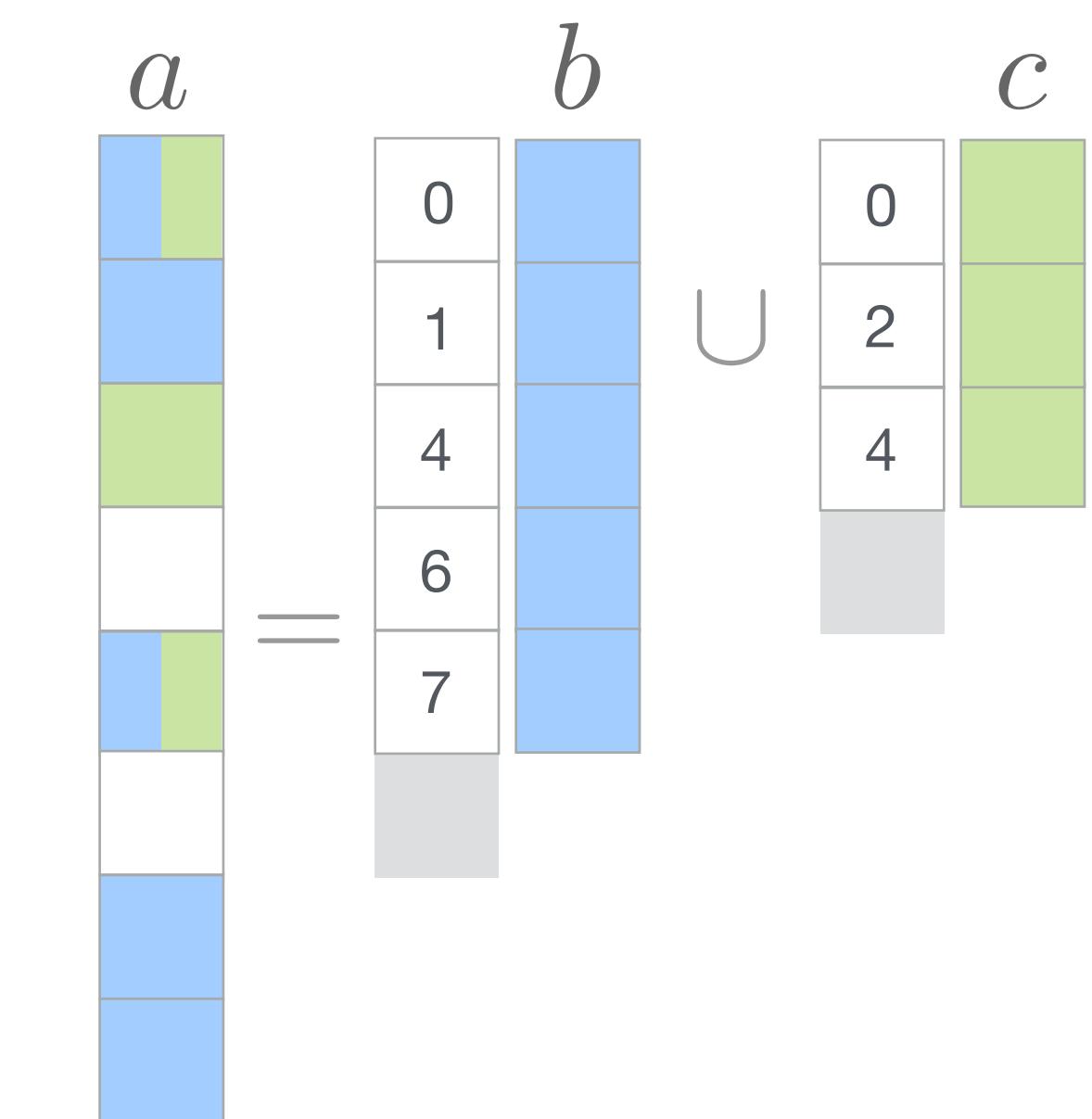
int pb1 = b1_pos[0];
int pc1 = c1_pos[0];
while (pb1 < b1_pos[1] && pc1 < c1_pos[1]) {
    int ib = b1_idx[pb1];
    int ic = c1_idx[pc1];
    int i = min(ib, ic);

    if (ib == i) pb1++;
    if (ic == i) pc1++;

}

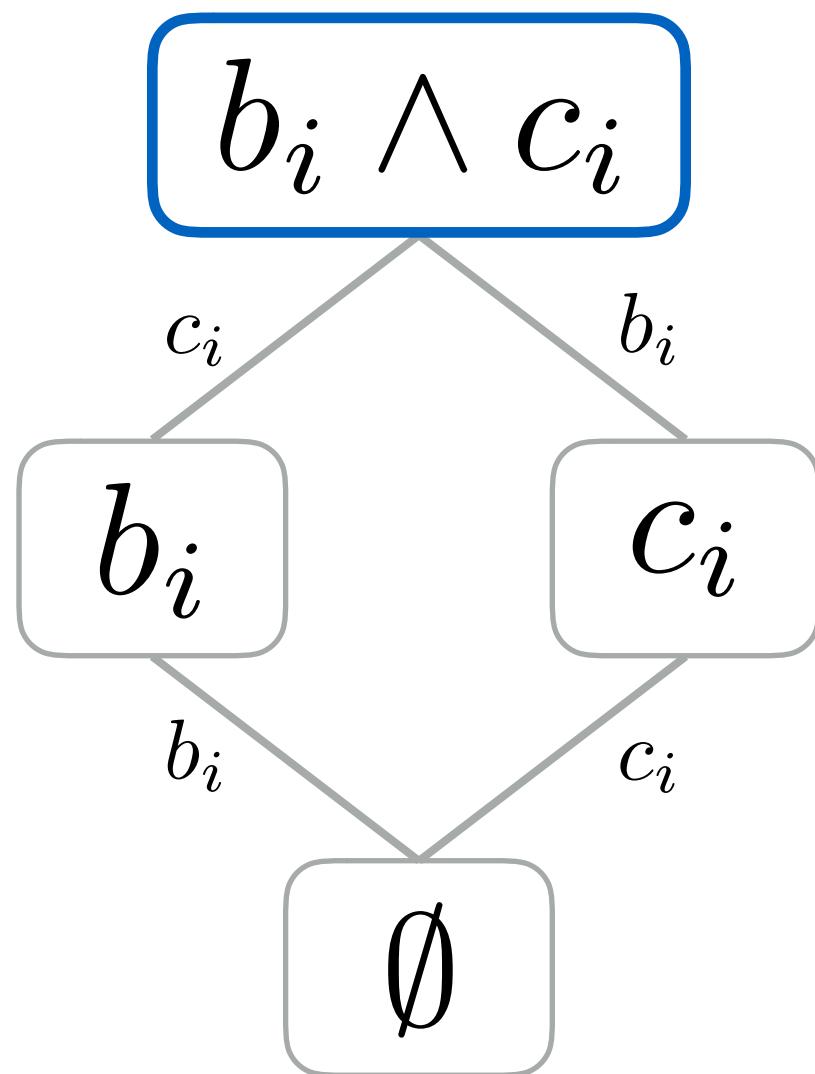
while (pb1 < b1_pos[1]) {
    int i = b1_idx[pb1];
    a[i] = b[pb1++];
}

while (pc1 < c1_pos[1]) {
    int i = c1_idx[pc1];
    a[i] = c[pc1++];
}
  
```



Merge Lattice for a Disjunction

$$a_i = b_i + c_i$$

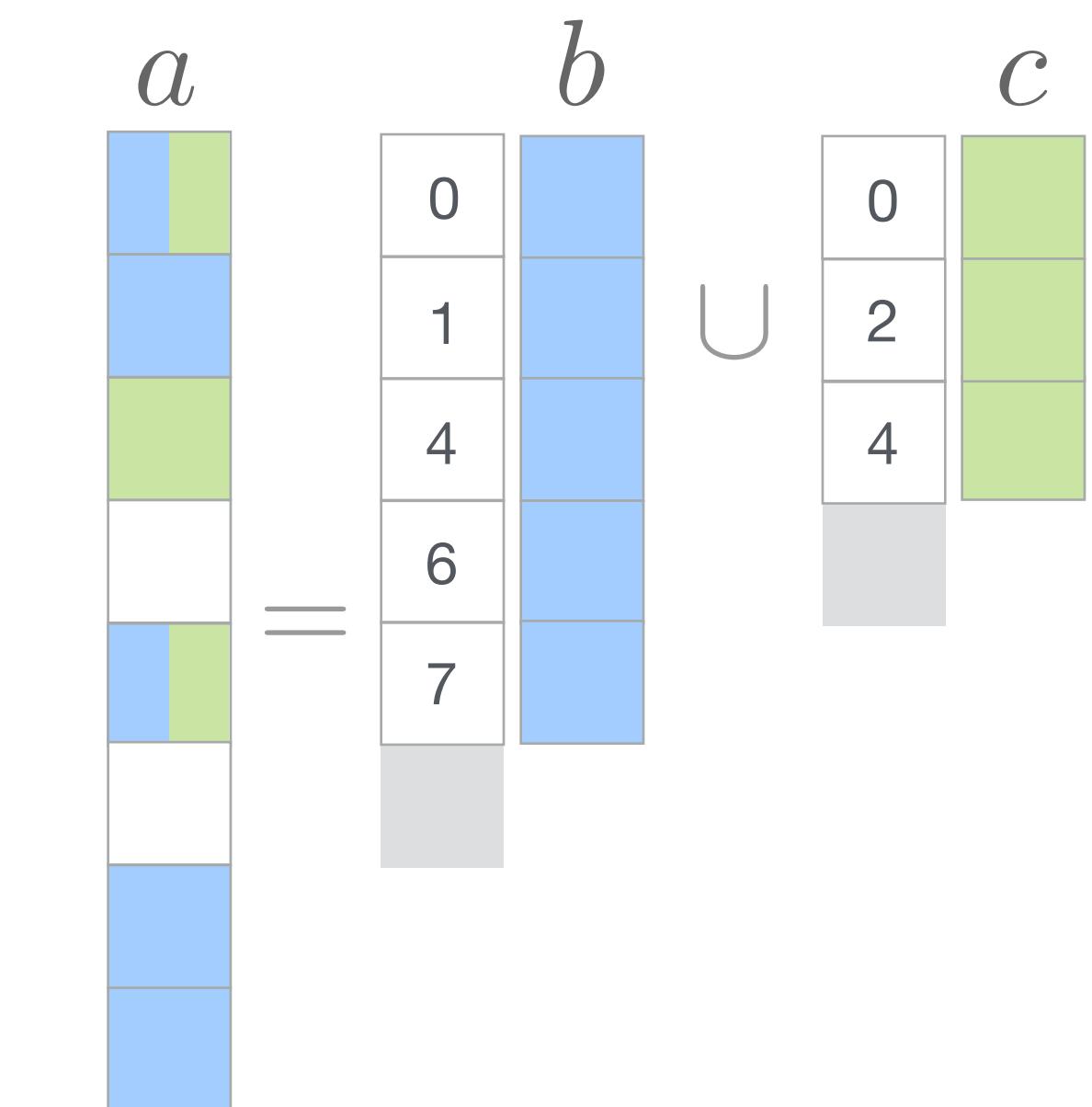


```

int pb1 = b1_pos[0];
int pc1 = c1_pos[0];
while (pb1 < b1_pos[1] && pc1 < c1_pos[1]) {
    int ib = b1_idx[pb1];
    int ic = c1_idx[pc1];
    int i = min(ib, ic);
    if (ib == i) pb1++;
    if (ic == i) pc1++;
}

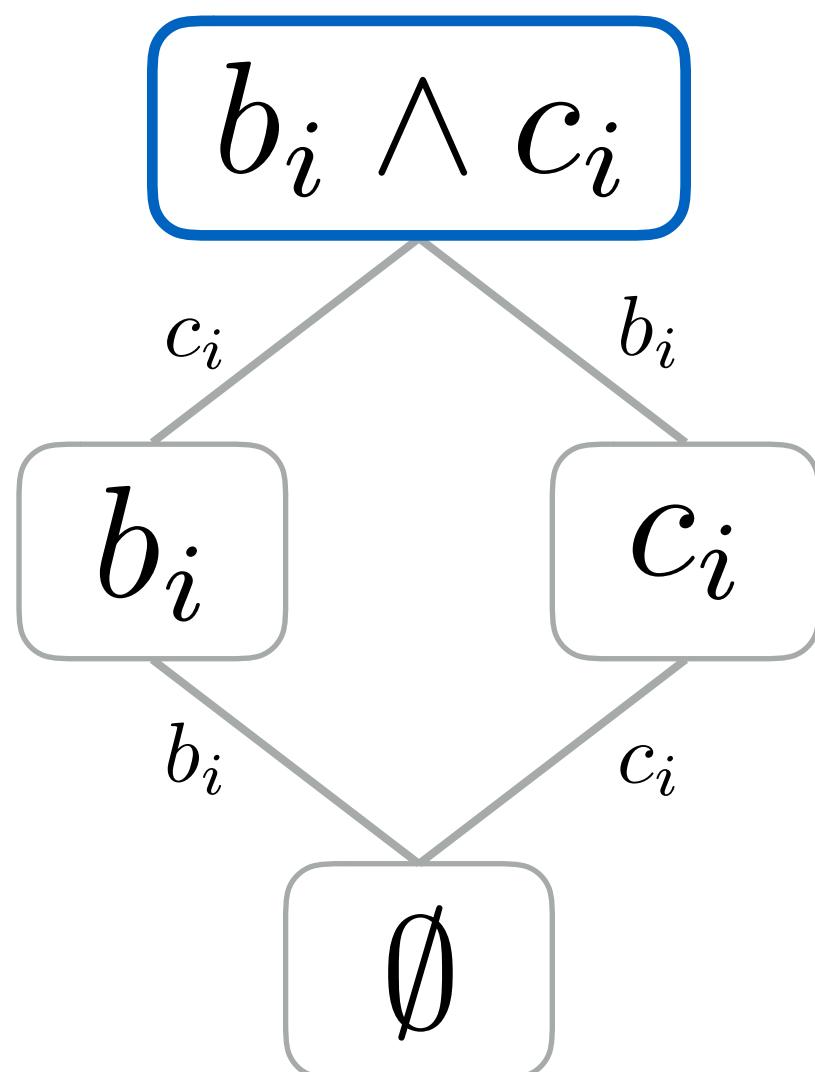
while (pb1 < b1_pos[1]) {
    int i = b1_idx[pb1];
    a[i] = b[pb1++];
}

while (pc1 < c1_pos[1]) {
    int i = c1_idx[pc1];
    a[i] = c[pc1++];
}
  
```



Merge Lattice for a Disjunction

$$a_i = b_i + c_i$$

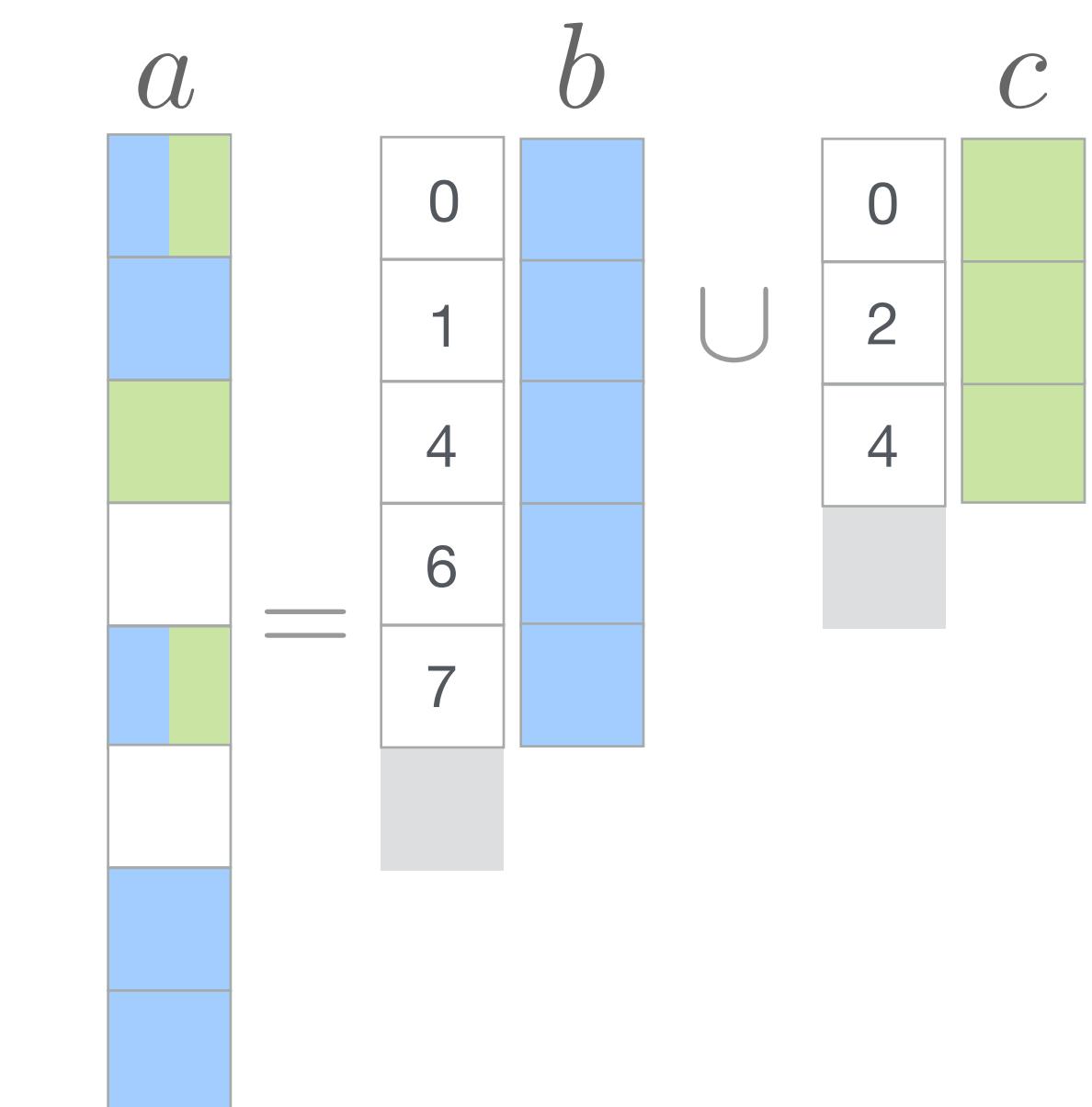


```

int pb1 = b1_pos[0];
int pc1 = c1_pos[0];
while (pb1 < b1_pos[1] && pc1 < c1_pos[1]) {
    int ib = b1_idx[pb1];
    int ic = c1_idx[pc1];
    int i = min(ib, ic);
    if (ib == i && ic == i) {
        a[i] = b[pb1] + c[pc1];
    }
    if (ib == i) pb1++;
    if (ic == i) pc1++;
}

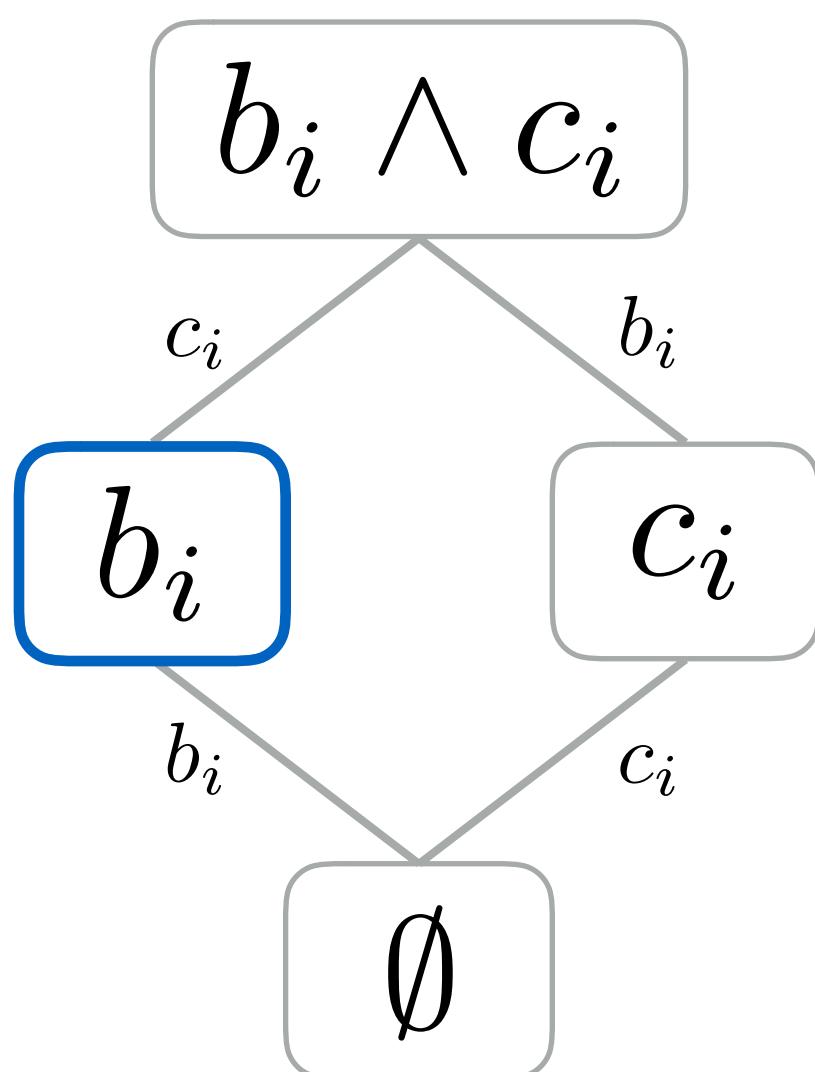
while (pb1 < b1_pos[1]) {
    int i = b1_idx[pb1];
    a[i] = b[pb1++];
}

while (pc1 < c1_pos[1]) {
    int i = c1_idx[pc1];
    a[i] = c[pc1++];
}
  
```



Merge Lattice for a Disjunction

$$a_i = b_i + c_i$$

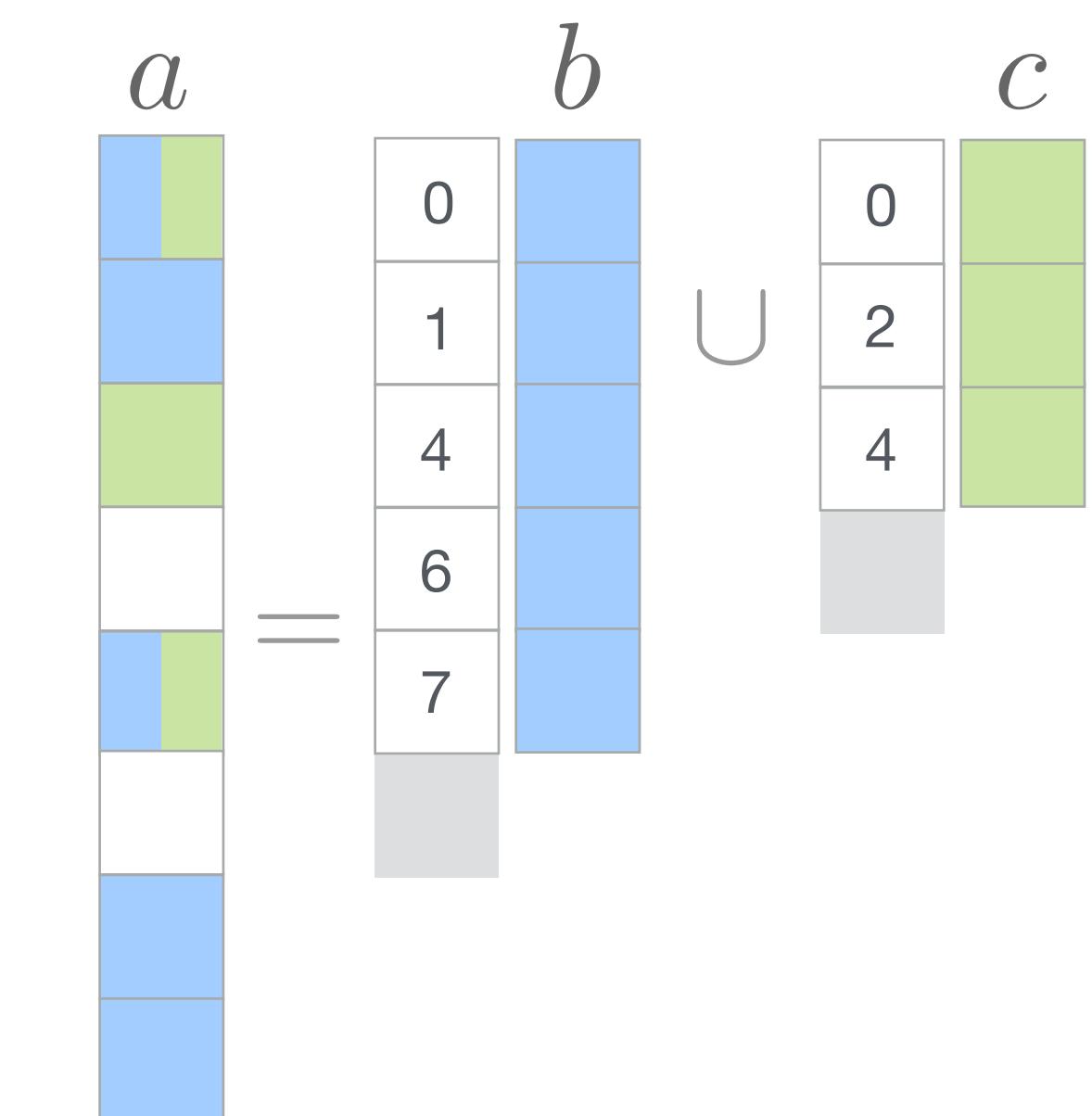


```

int pb1 = b1_pos[0];
int pc1 = c1_pos[0];
while (pb1 < b1_pos[1] && pc1 < c1_pos[1]) {
    int ib = b1_idx[pb1];
    int ic = c1_idx[pc1];
    int i = min(ib, ic);
    if (ib == i && ic == i) {
        a[i] = b[pb1] + c[pc1];
    }
    else if (ib == i) {
        a[i] = b[pb1];
    }
    if (ib == i) pb1++;
    if (ic == i) pc1++;
}

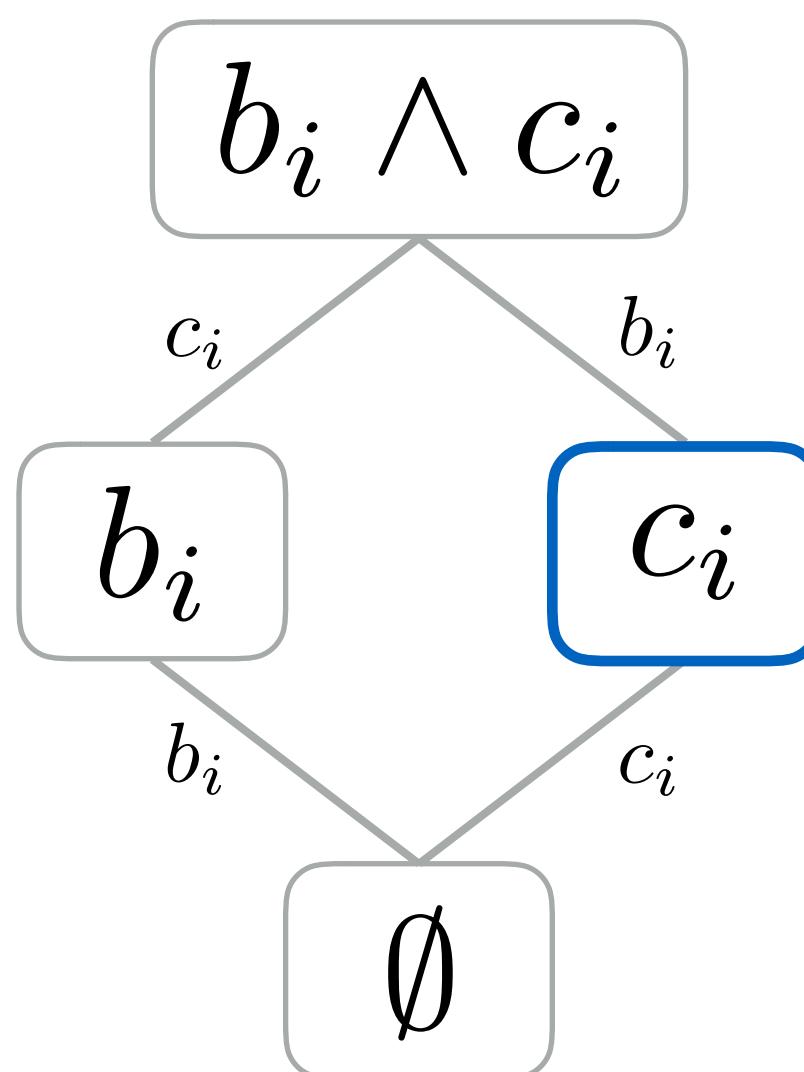
while (pb1 < b1_pos[1]) {
    int i = b1_idx[pb1];
    a[i] = b[pb1++];
}

while (pc1 < c1_pos[1]) {
    int i = c1_idx[pc1];
    a[i] = c[pc1++];
}
  
```



Merge Lattice for a Disjunction

$$a_i = b_i + c_i$$

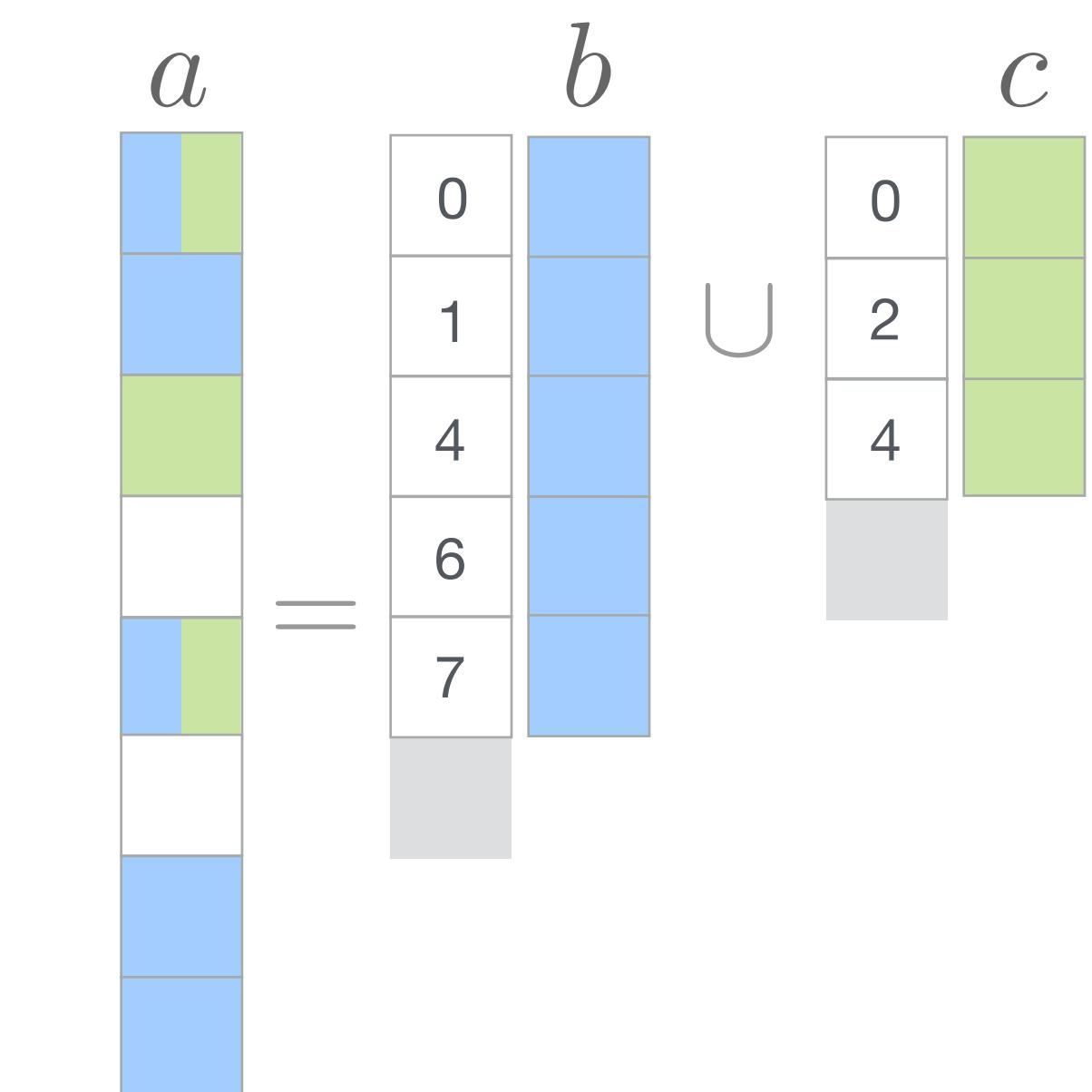


```

int pb1 = b1_pos[0];
int pc1 = c1_pos[0];
while (pb1 < b1_pos[1] && pc1 < c1_pos[1]) {
    int ib = b1_idx[pb1];
    int ic = c1_idx[pc1];
    int i = min(ib, ic);
    if (ib == i && ic == i) {
        a[i] = b[pb1] + c[pc1];
    } else if (ib == i) {
        a[i] = b[pb1];
    } else {
        a[i] = c[pc1];
    }
    if (ib == i) pb1++;
    if (ic == i) pc1++;
}

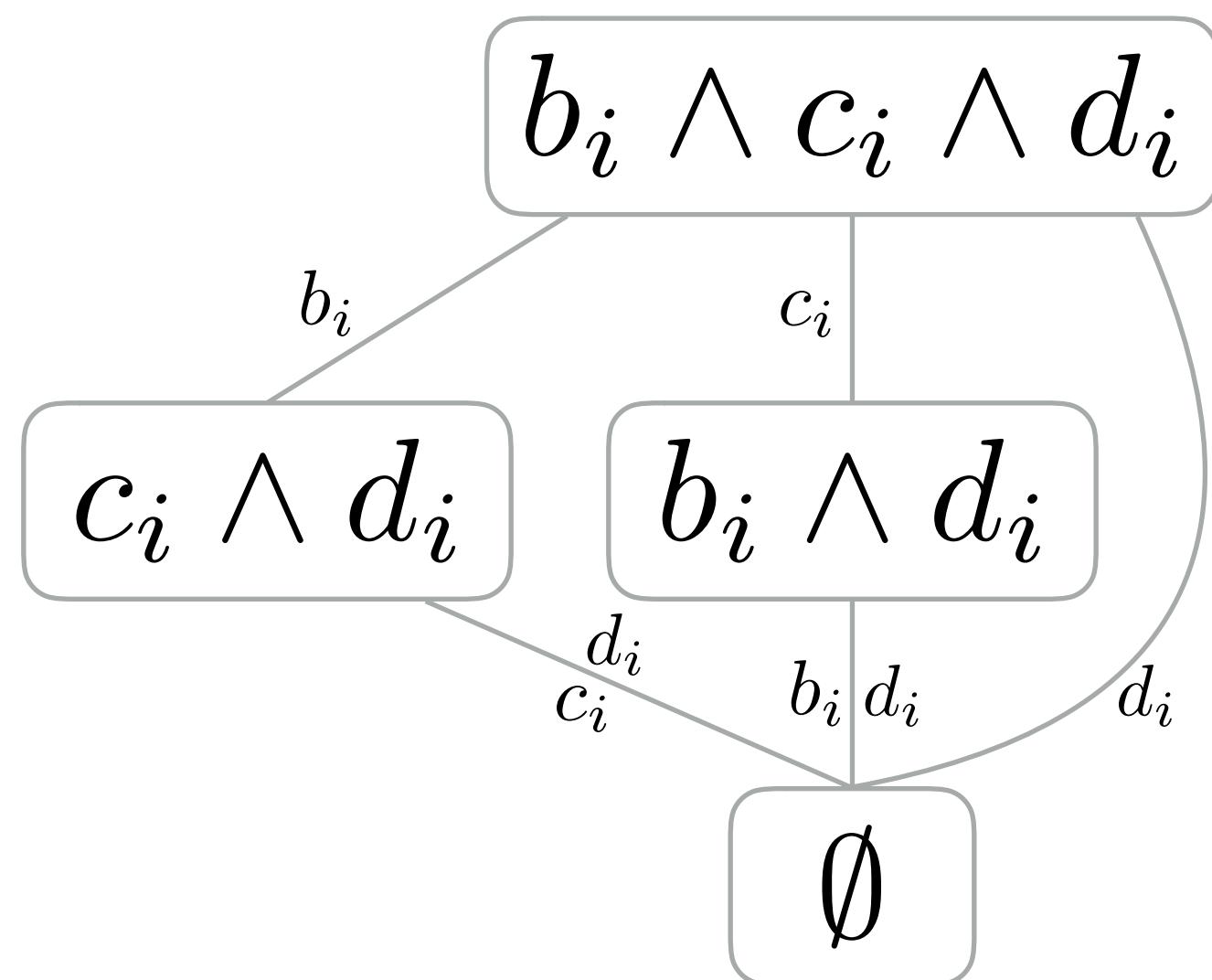
while (pb1 < b1_pos[1]) {
    int i = b1_idx[pb1];
    a[i] = b[pb1++];
}

while (pc1 < c1_pos[1]) {
    int i = c1_idx[pc1];
    a[i] = c[pc1++];
}
  
```



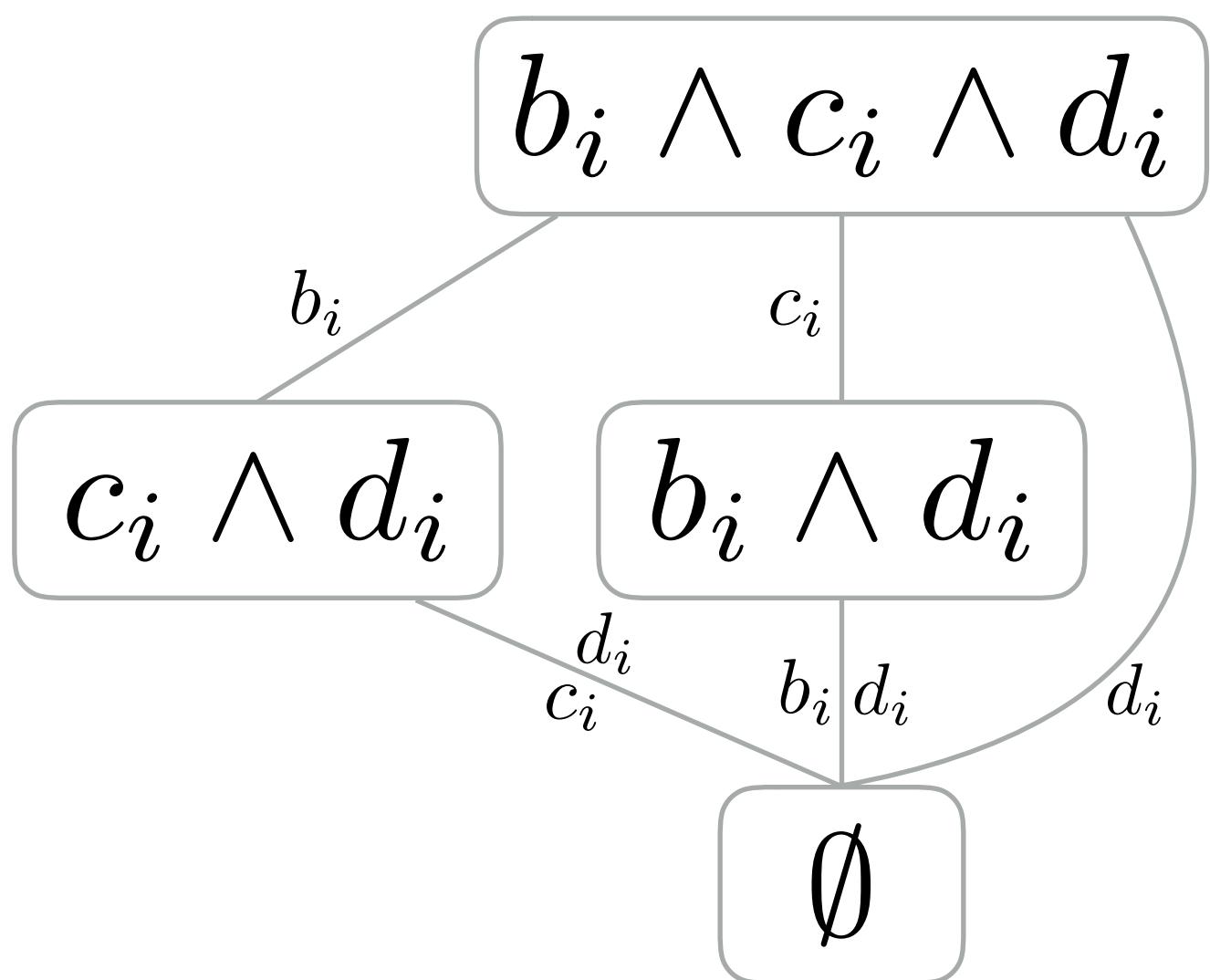
Merge Lattice for a Compound Expression

$$a_i = (b_i + c_i)d_i$$



Merge Lattice for a Compound Expression

$$a_i = (b_i + c_i)d_i$$



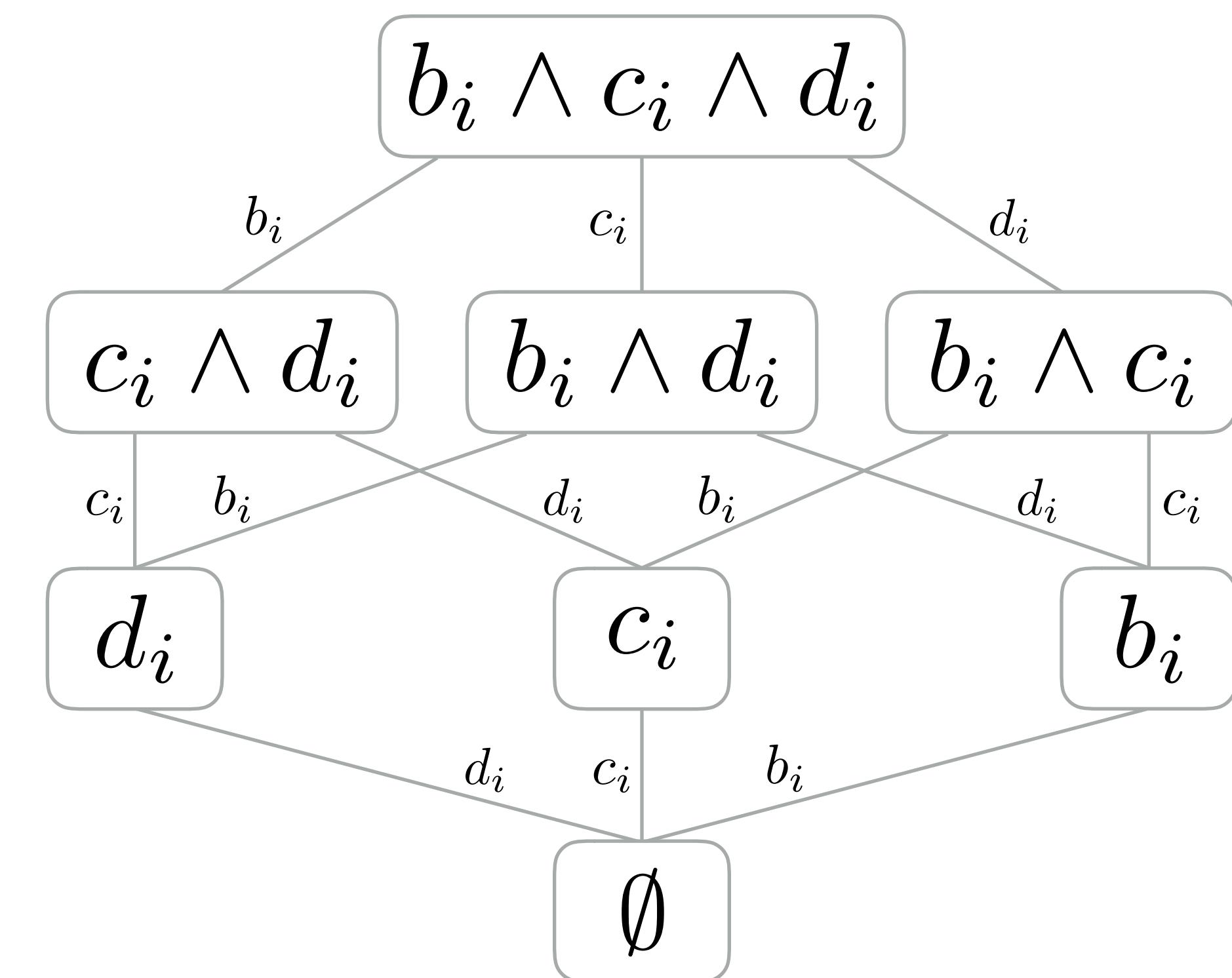
```
int pb1 = b1_pos[0];
int pc1 = c1_pos[0];
int pd1 = d1_pos[0];
while (pb1 < b1_pos[1] && pc1 < c1_pos[1] && pd1 < d1_pos[1]) {
    int ib = b1_idx[pb1];
    int ic = c1_idx[pc1];
    int id = d1_idx[pd1];
    int i = min(ib, ic, id);
    if (ib == i && ic == i && id == i) {
        a[i] = (b[pb1] + c[pc1]) * d[pd1];
    } else if (ib == i && id == i) {
        a[i] = b[pb1] * d[pd1];
    } else if (ic == i && id == i) {
        a[i] = c[pc1] * d[pd1];
    }
    if (ib == i) pb1++;
    if (ic == i) pc1++;
    if (id == i) pd1++;
}

while (pc1 < c1_pos[1] && pd1 < d1_pos[1]) {
    int ic0 = c1_idx[pc1];
    int id1 = d1_idx[pd1];
    int il = min(ic0, id1);
    if (ic0 == il && id1 == il) {
        a[il] = c[pc1] * d[pd1];
    }
    if (ic0 == il) pc1++;
    if (id1 == il) pd1++;
}

while (pb1 < b1_pos[1] && pd1 < d1_pos[1]) {
    int ib0 = b1_idx[pb1];
    int id0 = d1_idx[pd1];
    int i0 = min(ib0, id0);
    if (ib0 == i0 && id0 == i0) {
        a[i0] = b[pb1] * d[pd1];
    }
    if (ib0 == i0) pb1++;
    if (id0 == i0) pd1++;
}
```

Merge Lattice for a Compound Expression

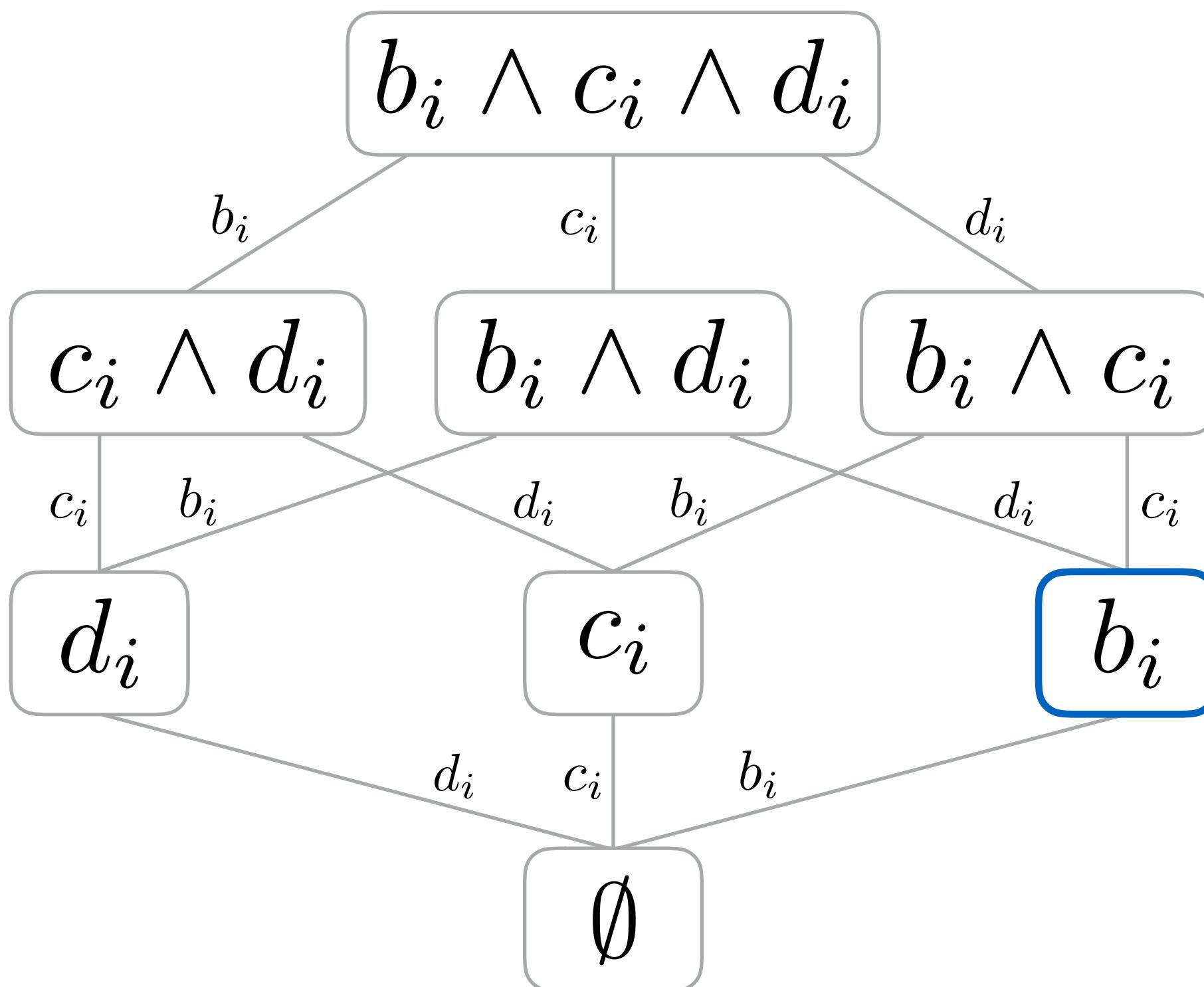
$$a_i = b_i + c_i + d_i$$



Merge Lattice for a Compound Expression

$$a_i = b_i + c_i + d_i$$

↑
dense?



```

int pb1 = b1_pos[0];
int pc1 = c1_pos[0];
int pd1 = d1_pos[0];
while (pb1 < b1_pos[1] && pc1 < c1_pos[1] && pd1 < d1_pos[1]) {
    int ib = b1_idx[pb1];
    int ic = c1_idx[pc1];
    int id = d1_idx[pd1];
    int i = min(ib, ic, id);
    if (ib == i && ic == i && id == i) {
        a[i] = b[pb1] + c[pc1] + d[pd1];
    } else if (ib == i && id == i) {
        a[i] = b[pb1] + d[pd1];
    } else if (ic == i && id == i) {
        a[i] = c[pc1] + d[pd1];
    } else if (ib == i && ic == i) {
        a[i] = b[pb1] + c[pc1];
    } else if (ib == i) {
        a[i] = b[pb1];
    } else if (ic == i) {
        a[i] = c[pc1];
    } else {
        a[i] = d[pd1];
    }
    if (ib == i) pb1++;
    if (ic == i) pc1++;
    if (id == i) pd1++;
}

while (pb1 < b1_pos[1] && pd1 < d1_pos[1]) {
    int ib0 = b1_idx[pb1];
    int id0 = d1_idx[pd1];
    int i0 = min(ib0, id0);
    if (ib0 == i0 && id0 == i0) {
        a[i0] = b[pb1] + d[pd1];
    } else if (ib0 == i0) {
        a[i0] = b[pb1];
    } else {
        a[i0] = d[pd1];
    }
    if (ib0 == i0) pb1++;
    if (id0 == i0) pd1++;
}

while (pb1 < b1_pos[1] && pc1 < c1_pos[1]) {
    int ib1 = b1_idx[pb1];
    int ic1 = c1_idx[pc1];
    int i2 = min(ib1, ic1);
    if (ib1 == i2 && ic1 == i2) {
        a[i2] = b[pb1] + c[pc1];
    } else if (ib1 == i2) {
        a[i2] = b[pb1];
    } else {
        a[i2] = c[pc1];
    }
    if (ib1 == i2) pb1++;
    if (ic1 == i2) pc1++;
}

while (pd1 < d1_pos[1]) {
    int id2 = d1_idx[pd1];
    a[id2] = d[pd1];
    pd1++;
}

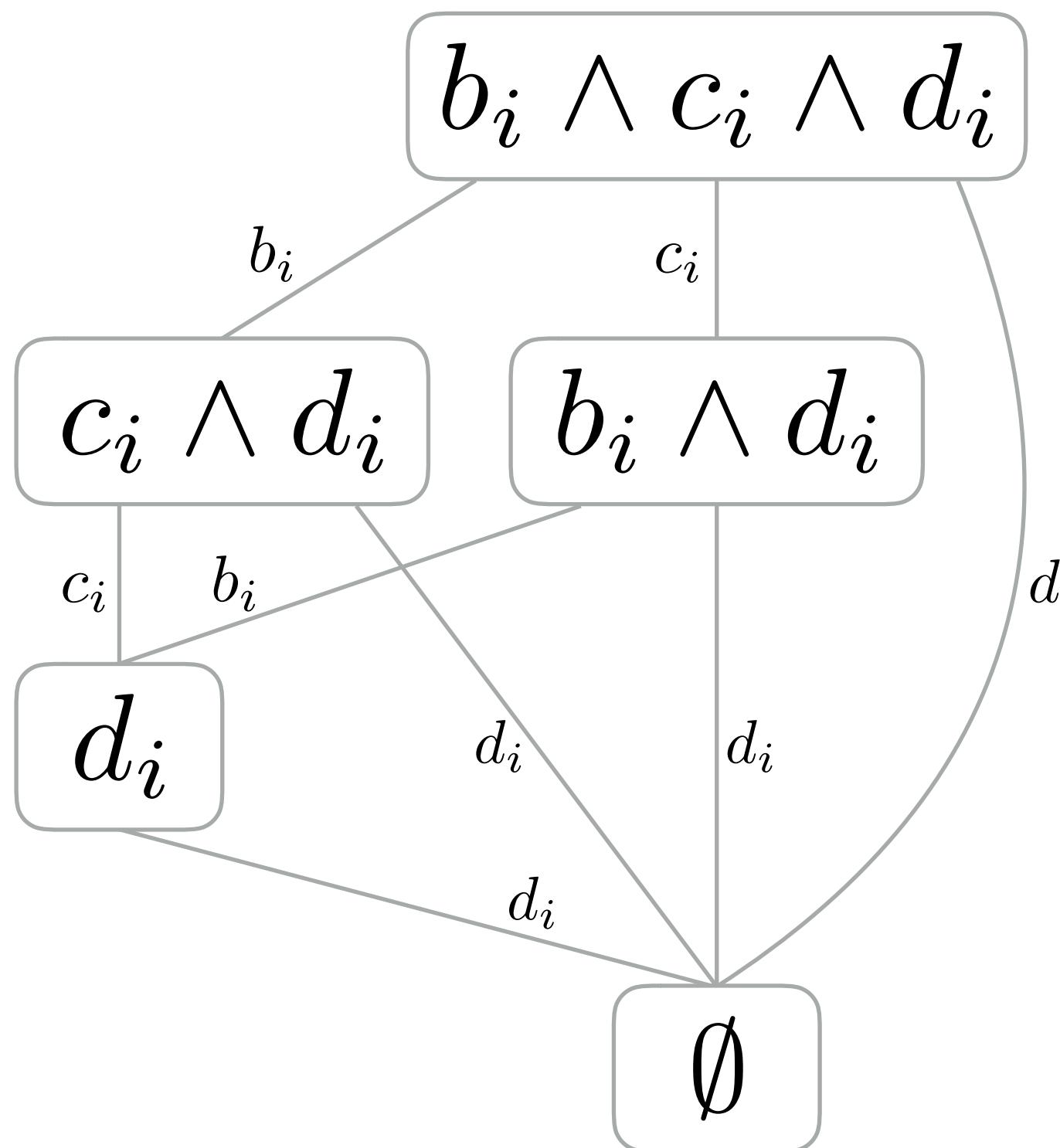
while (pc1 < c1_pos[1]) {
    int ic2 = c1_idx[pc1];
    a[ic2] = c[pc1];
    pc1++;
}

while (pb1 < b1_pos[1]) {
    int ib2 = b1_idx[pb1];
    a[ib2] = b[pb1];
    pb1++;
}
  
```

Merge Lattice for a Compound Expression

$$a_i = b_i + c_i + d_i$$

↑
dense



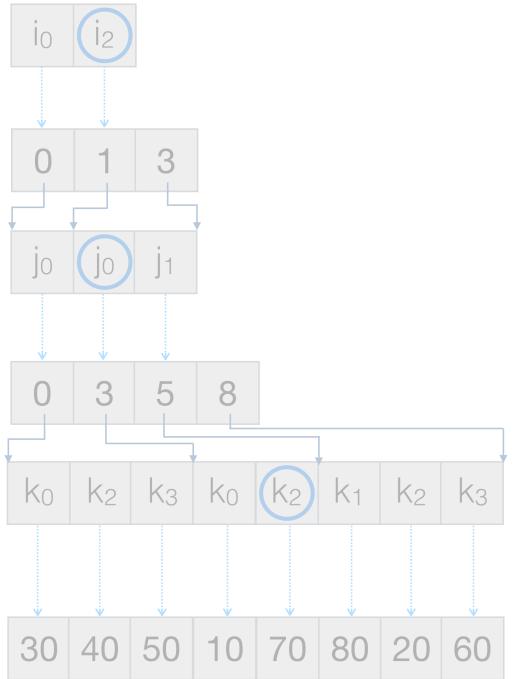
```

int pb1 = b1_pos[0];
int pc1 = c1_pos[0];
int id = 0;
while (pb1 < b1_pos[1] && pc1 < c1_pos[1]) {
    int ib = b1_idx[pb1];
    int ic = c1_idx[pc1];
    int pd1 = id;
    int pa1 = id;
    if (ib == id && ic == id) {
        a[pa1] = b[pb1] + c[pc1] + d[pd1];
    }
    else if (ib == id) {
        a[pa1] = b[pb1] + d[pd1];
    }
    else if (ic == id) {
        a[pa1] = c[pc1] + d[pd1];
    }
    else {
        a[pa1] = d[pd1];
    }
    if (ib == id) pb1++;
    if (ic == id) pc1++;
    id++;
}

while (pb1 < b1_pos[1]) {
    int ib0 = b1_idx[pb1];
    int pd10 = id;
    int pa10 = id;
    if (ib0 == id) {
        a[pa10] = b[pb1] + d[pd10];
    }
    else {
        a[pa10] = d[pd10];
    }
    if (ib0 == id) pb1++;
    id++;
}

while (id < d1_dimension) {
    int pd12 = id;
    int pa12 = id;
    a[pa12] = d[pd12];
    id++;
}
  
```

Sparse code and data



```

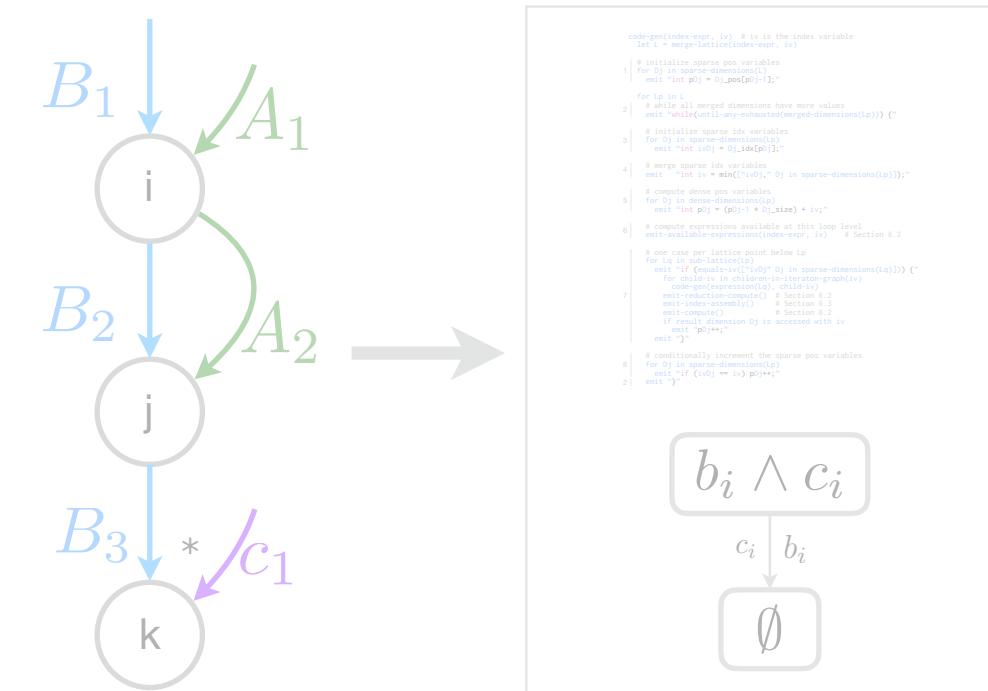
for (int pB1 = B1_pos[0]; pB1 < B1_pos[1]; pB1++) {
    int i = B1_idx[pB1];

    for (int pB2 = B2_pos[pB1];
        pB2 < B2_pos[pB1 + 1];
        pB2++) {
        int j = B2_idx[pB2];
        int pA2 = (i * n) + j;

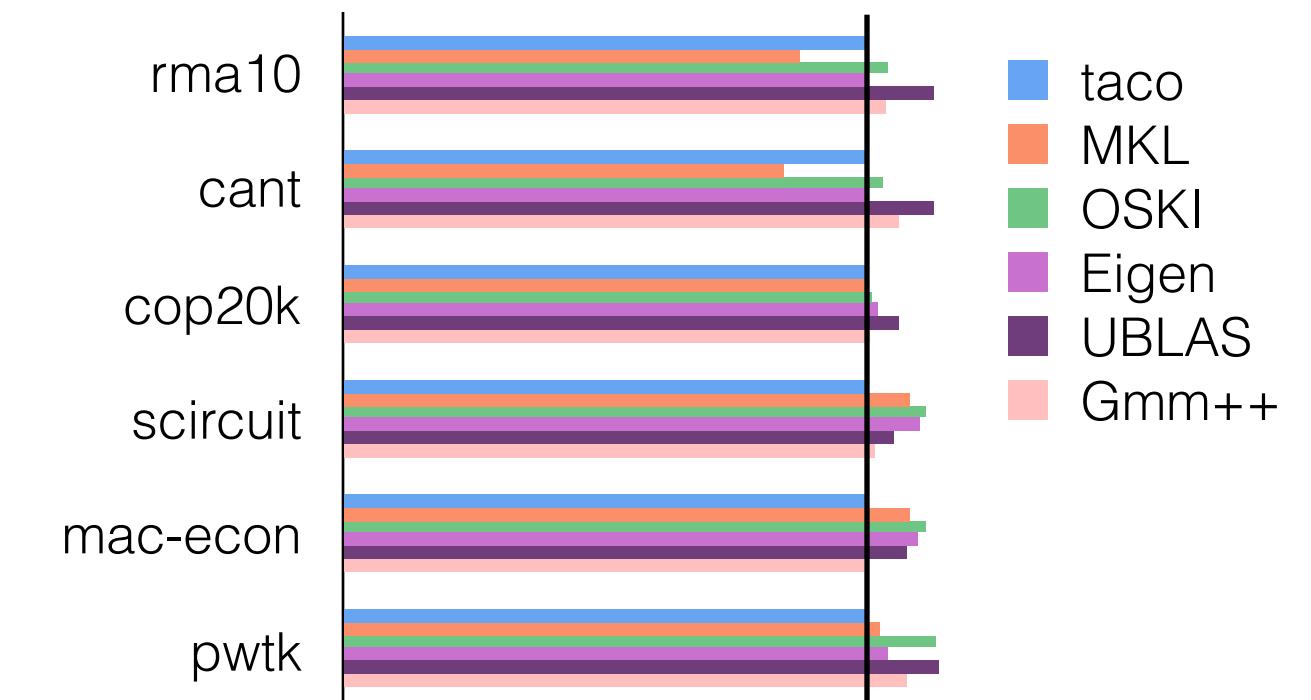
        int pB3 = B3_pos[pB2];
        int pc1 = c1_pos[0];
        while (pB3 < B3_pos[pB2 + 1] && pc1 < c1_pos[1]) {
            int kB = B3_idx[pB3];
            int kc = c1_idx[pc1];
            int k = min(kB, kc);
            if (kB == k && kc == k) {
                a[pA2] += b[pB3] * c[pc1];
            }
            if (kB == k) pB3++;
            if (kc == k) pc1++;
        }
    }
}

```

Intermediate Representation and Code Generation



Evaluation



The Tensor Algebra Compiler is fast and general

$$y = \alpha A^T x + \beta z \quad a = Bc$$

$$a = B^T c \quad A = B + C + D \quad y = b - Ax$$

$$a = \frac{A - B}{Bc + a} \quad A_{ijk} = \sum_l B_{ijl} C_{kl} \quad A = B \circ (CD)$$

$$A = B \odot C \quad A_{ijk} = B_{ijk} + C_{ijk} \quad A_{ij} \sum_k B_{ijk} c_k$$

$$A = B^T \quad \alpha = \sum_{i,j,k} B_{ijk} C_{ijk} \quad A_{ij} = \sum_{k,l} B_{ikl} C_{kj} D_{lj}$$

$$A = B + C \quad a = B^T c + d \quad a = b + c \quad A = \alpha A + \beta B$$

$$A_{ik} = \sum_j B_{ijk} * c_j \quad A = 0 \quad A = B + C + D \quad A = (B + C)(d + e)$$

$$A = \alpha B \quad A = (B + C)d \quad A = BC \quad A_{ilk} = \sum_j B_{ijk} * C_{lj}$$

$$A_{ijl} = \sum_k B_{ijk} * C_{lk} \quad a = BCd \quad a = Bc + b \quad A = \sum_j A_{jl} + \alpha I$$

$$A = \alpha B + A \quad A_{jl} = \sum_{i,k} B_{ijk} * C_{il} * D_{kl} \quad a = \alpha Bc + \beta a$$

The Tensor Algebra Compiler is fast and general

$$y = \alpha A^T x + \beta z \quad \boxed{a = Bc}$$

Sparse Matrix-Vector Multiplication (SpMV)

$$a = B^T c \quad A = B + C + D \quad y = b - Ax$$

$$a = \frac{A - B}{Bc + a} \quad A_{ijk} = \sum_l B_{ijl} C_{kl} \quad A = B \circ (CD)$$

$$A = B \odot C \quad A_{ijk} = B_{ijk} + C_{ijk} \quad A_{ij} \sum_k B_{ijk} c_k$$

$$A = B^T \quad \alpha = \sum_{i,j,k} B_{ijk} C_{ijk} \quad A_{ij} = \sum_{k,l} B_{ikl} C_{kj} D_{lj}$$

$$A = B + C \quad a = B^T c + d \quad a = b + c \quad A = \alpha A + \beta B$$

$$A_{ik} = \sum_j B_{ijk} * c_j \quad A = 0 \quad A = B + C + D \quad A = (B + C)(d + e)$$

$$A = \alpha B \quad A = (B + C)d \quad A = BC \quad A_{ilk} = \sum_j B_{ijk} * C_{lj}$$

$$A_{ijl} = \sum_k B_{ijk} * C_{lk} \quad a = BCd \quad a = Bc + b \quad A = \sum_j A + \alpha I$$

$$A = \alpha B + A \quad A_{jl} = \sum_{i,k} B_{ijk} * C_{il} * D_{kl} \quad a = \alpha Bc + \beta a$$

The Tensor Algebra Compiler is fast and general

$$y = \alpha A^T x + \beta z \quad \boxed{a = Bc}$$

$$a = B^T c \quad A = B + C + D \quad y = b - Ax$$

$$a = \frac{A - B}{Bc + a} \quad A_{ijk} = \sum_l B_{ijl} C_{kl} \quad A = B \circ (CD)$$

$$A = B \odot C \quad A_{ijk} = B_{ijk} + C_{ijk} \quad A_{ij} \sum_k B_{ijk} c_k$$

$$A = B^T \quad \alpha = \sum_{i,j,k} B_{ijk} C_{ijk} \quad A_{ij} = \sum_{k,l} B_{ikl} C_{kj} D_{lj}$$

$$A = B + C \quad a = B^T c + d \quad a = b + c \quad A = \alpha A + \beta B$$

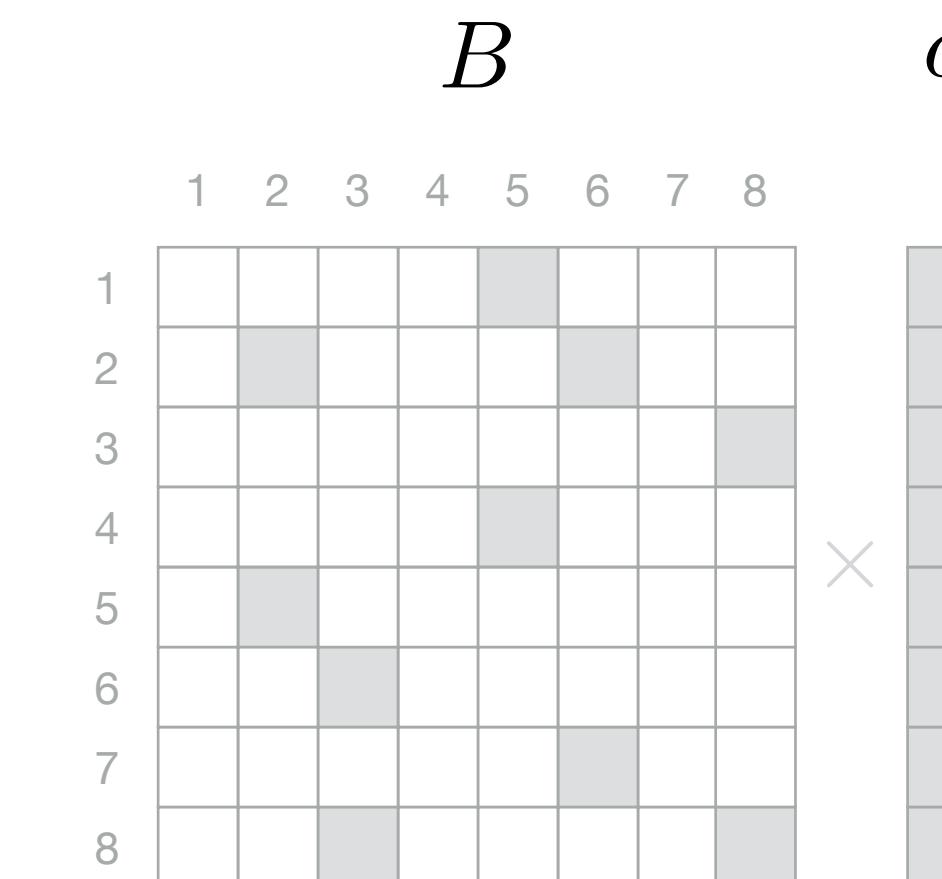
$$A_{ik} = \sum_j B_{ijk} * c_j \quad A = 0 \quad A = B + C + D \quad A = (B + C)(d + e)$$

$$A = \alpha B \quad A = (B + C)d \quad A = BC \quad A_{ilk} = \sum_j B_{ijk} * C_{lj}$$

$$A_{ijl} = \sum_k B_{ijk} * C_{lk} \quad a = BCd \quad a = Bc + b \quad A = \sum_j A + \alpha I$$

$$A = \alpha B + A \quad A_{jl} = \sum_{i,k} B_{ijk} * C_{il} * D_{kl} \quad a = \alpha Bc + \beta a$$

Sparse Matrix-Vector Multiplication (SpMV)



The Tensor Algebra Compiler is fast and general

$$y = \alpha A^T x + \beta z \quad a = Bc$$

$$a = B^T c \quad A = B + C + D \quad y = b - Ax$$

$$a = \frac{A - B}{Bc + a} \quad A_{ijk} = \sum_l B_{ijl} C_{kl} \quad A = B \circ (CD)$$

$$A = B \odot C \quad A_{ijk} = B_{ijk} + C_{ijk} \quad A_{ij} \sum_k B_{ijk} c_k$$

$$A = B^T \quad \alpha = \sum_{i,j,k} B_{ijk} C_{ijk} \quad A_{ij} = \sum_{k,l} B_{ikl} C_{kj} D_{lj}$$

$$A = B + C \quad a = B^T c + d \quad a = b + c \quad A = \alpha A + \beta B$$

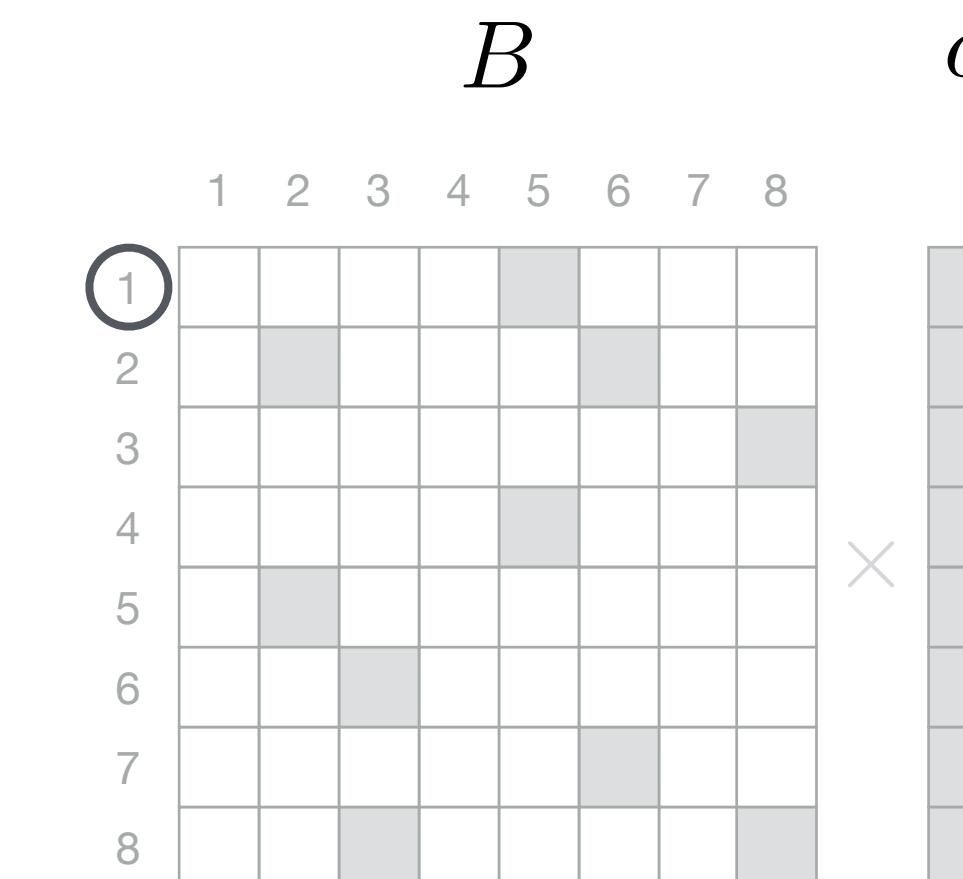
$$A_{ik} = \sum_j B_{ijk} * c_j \quad A = 0 \quad A = B + C + D \quad A = (B + C)(d + e)$$

$$A = \alpha B \quad A = (B + C)d \quad A = BC \quad A_{ilk} = \sum_j B_{ijk} * C_{lj}$$

$$A_{ijl} = \sum_k B_{ijk} * C_{lk} \quad a = BCd \quad a = Bc + b \quad A = \sum_j A + \alpha I$$

$$A = \alpha B + A \quad A_{jl} = \sum_{i,k} B_{ijk} * C_{il} * D_{kl} \quad a = \alpha Bc + \beta a$$

Sparse Matrix-Vector Multiplication (SpMV)



The Tensor Algebra Compiler is fast and general

$$y = \alpha A^T x + \beta z \quad a = Bc$$

$$a = B^T c \quad A = B + C + D \quad y = b - Ax$$

$$a = \frac{A - B}{Bc + a} \quad A_{ijk} = \sum_l B_{ijl} C_{kl} \quad A = B \circ (CD)$$

$$A = B \odot C \quad A_{ijk} = B_{ijk} + C_{ijk} \quad A_{ij} \sum_k B_{ijk} c_k$$

$$A = B^T \quad \alpha = \sum_{i,j,k} B_{ijk} C_{ijk} \quad A_{ij} = \sum_{k,l} B_{ikl} C_{kj} D_{lj}$$

$$A = B + C \quad a = B^T c + d \quad a = b + c \quad A = \alpha A + \beta B$$

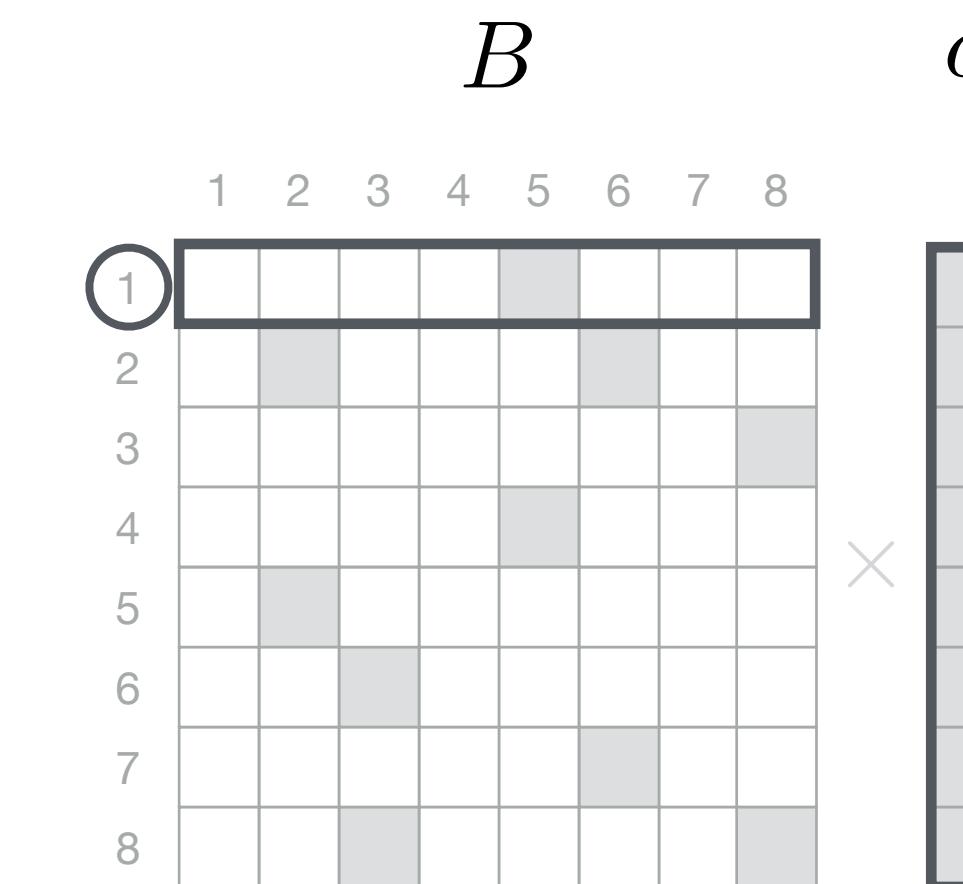
$$A_{ik} = \sum_j B_{ijk} * c_j \quad A = 0 \quad A = B + C + D \quad A = (B + C)(d + e)$$

$$A = \alpha B \quad A = (B + C)d \quad A = BC \quad A_{ilk} = \sum_j B_{ijk} * C_{lj}$$

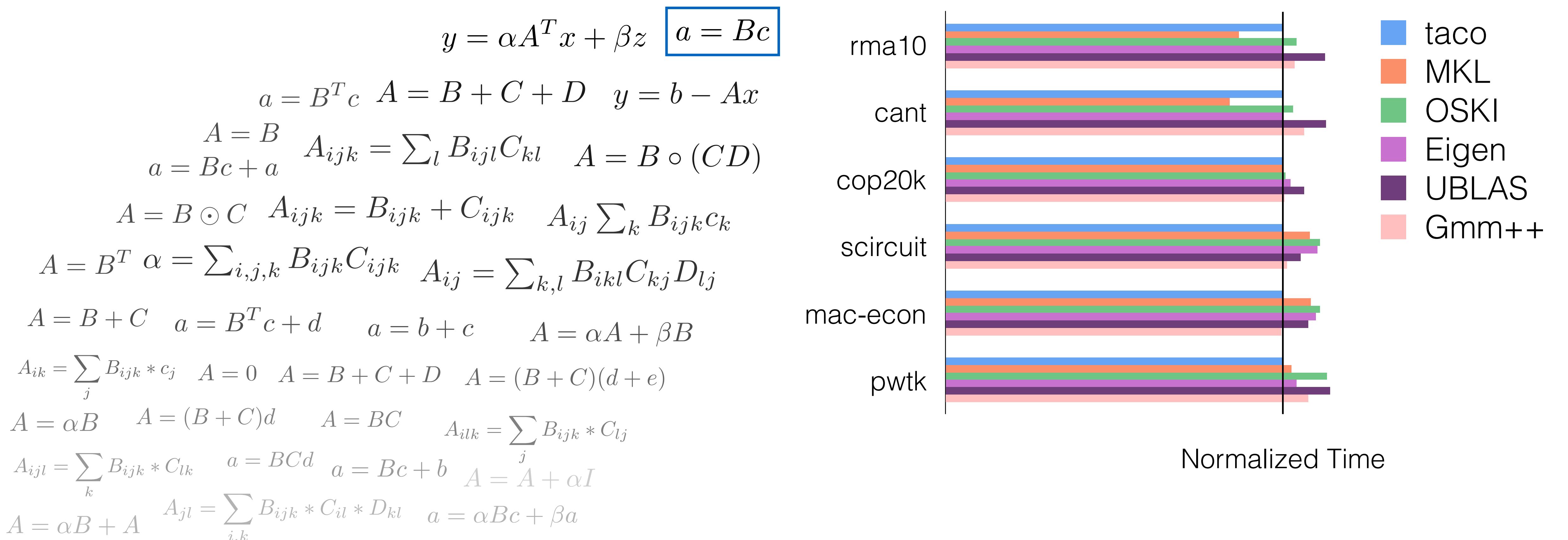
$$A_{ijl} = \sum_k B_{ijk} * C_{lk} \quad a = BCd \quad a = Bc + b \quad A = \sum_j A_{ilj} + \alpha I$$

$$A = \alpha B + A \quad A_{jl} = \sum_{i,k} B_{ijk} * C_{il} * D_{kl} \quad a = \alpha Bc + \beta a$$

Sparse Matrix-Vector Multiplication (SpMV)

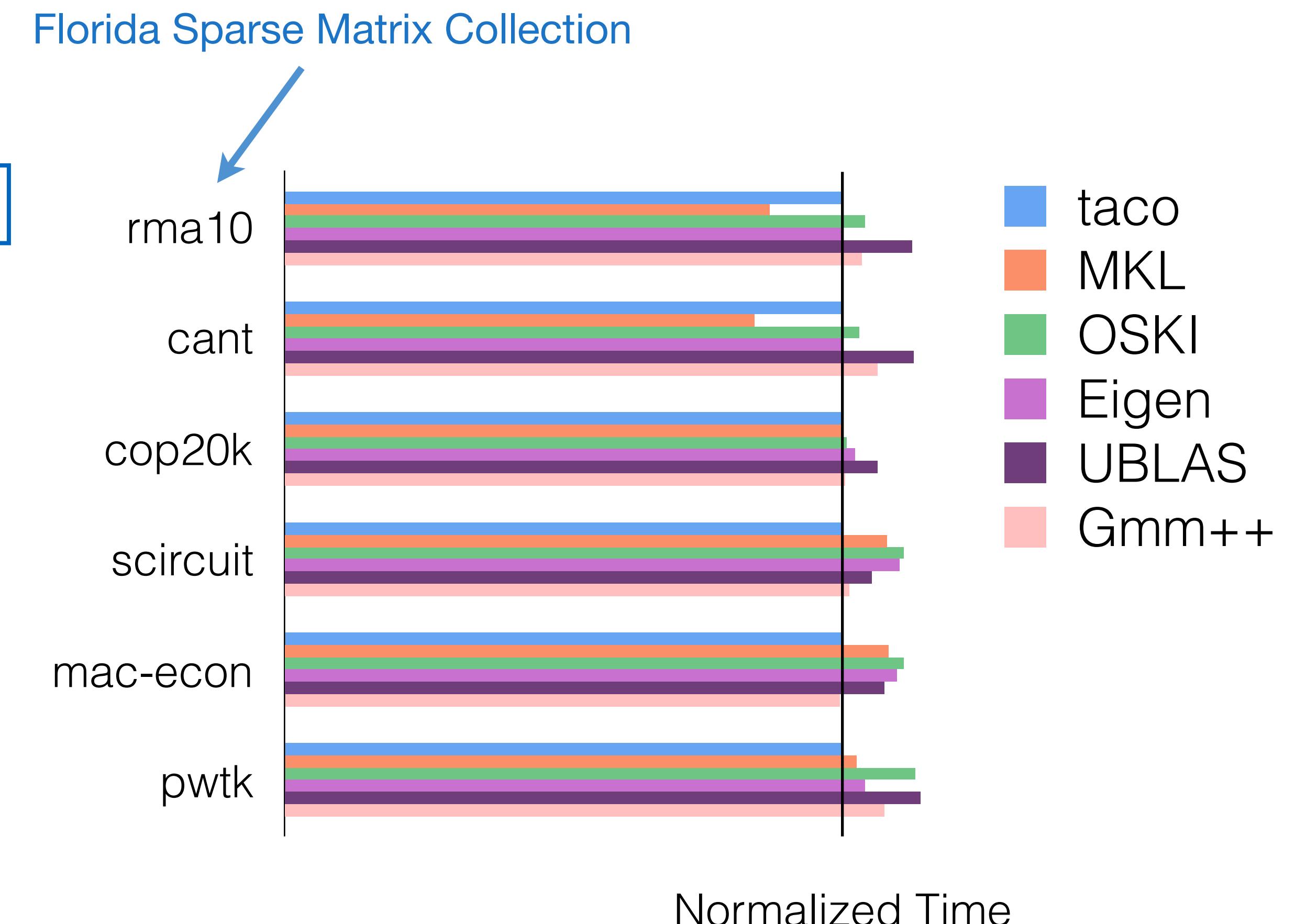


SpMV is competitive with hand-optimized implementations



SpMV is competitive with hand-optimized implementations

$$\begin{aligned}
 & y = \alpha A^T x + \beta z \quad \boxed{a = Bc} \\
 & a = B^T c \quad A = B + C + D \quad y = b - Ax \\
 & a = \frac{A}{B} \quad A_{ijk} = \sum_l B_{ijl} C_{kl} \quad A = B \circ (CD) \\
 & a = B \odot C \quad A_{ijk} = B_{ijk} + C_{ijk} \quad A_{ij} \sum_k B_{ijk} c_k \\
 & A = B^T \quad \alpha = \sum_{i,j,k} B_{ijk} C_{ijk} \quad A_{ij} = \sum_{k,l} B_{ikl} C_{kj} D_{lj} \\
 & A = B + C \quad a = B^T c + d \quad a = b + c \quad A = \alpha A + \beta B \\
 & A_{ik} = \sum_j B_{ijk} * c_j \quad A = 0 \quad A = B + C + D \quad A = (B + C)(d + e) \\
 & A = \alpha B \quad A = (B + C)d \quad A = BC \quad A_{ilk} = \sum_j B_{ijk} * C_{lj} \\
 & A_{ijl} = \sum_k B_{ijk} * C_{lk} \quad a = BCd \quad a = Bc + b \quad A = \sum_j A + \alpha I \\
 & A = \alpha B + A \quad A_{jl} = \sum_{i,k} B_{ijk} * C_{il} * D_{kl} \quad a = \alpha Bc + \beta a
 \end{aligned}$$



SpMV is competitive with hand-optimized implementations

$$y = \alpha A^T x + \beta z \quad \boxed{a = Bc}$$

$$a = B^T c \quad A = B + C + D \quad y = b - Ax$$

$$a = \begin{matrix} A \\ B \end{matrix} \quad A_{ijk} = \sum_l B_{ijl} C_{kl} \quad A = B \circ (CD)$$

$$a = Bc + a \quad A_{ijk} = B_{ijk} + C_{ijk} \quad A_{ij} \sum_k B_{ijk} c_k$$

$$A = B^T \quad \alpha = \sum_{i,j,k} B_{ijk} C_{ijk} \quad A_{ij} = \sum_{k,l} B_{ikl} C_{kj} D_{lj}$$

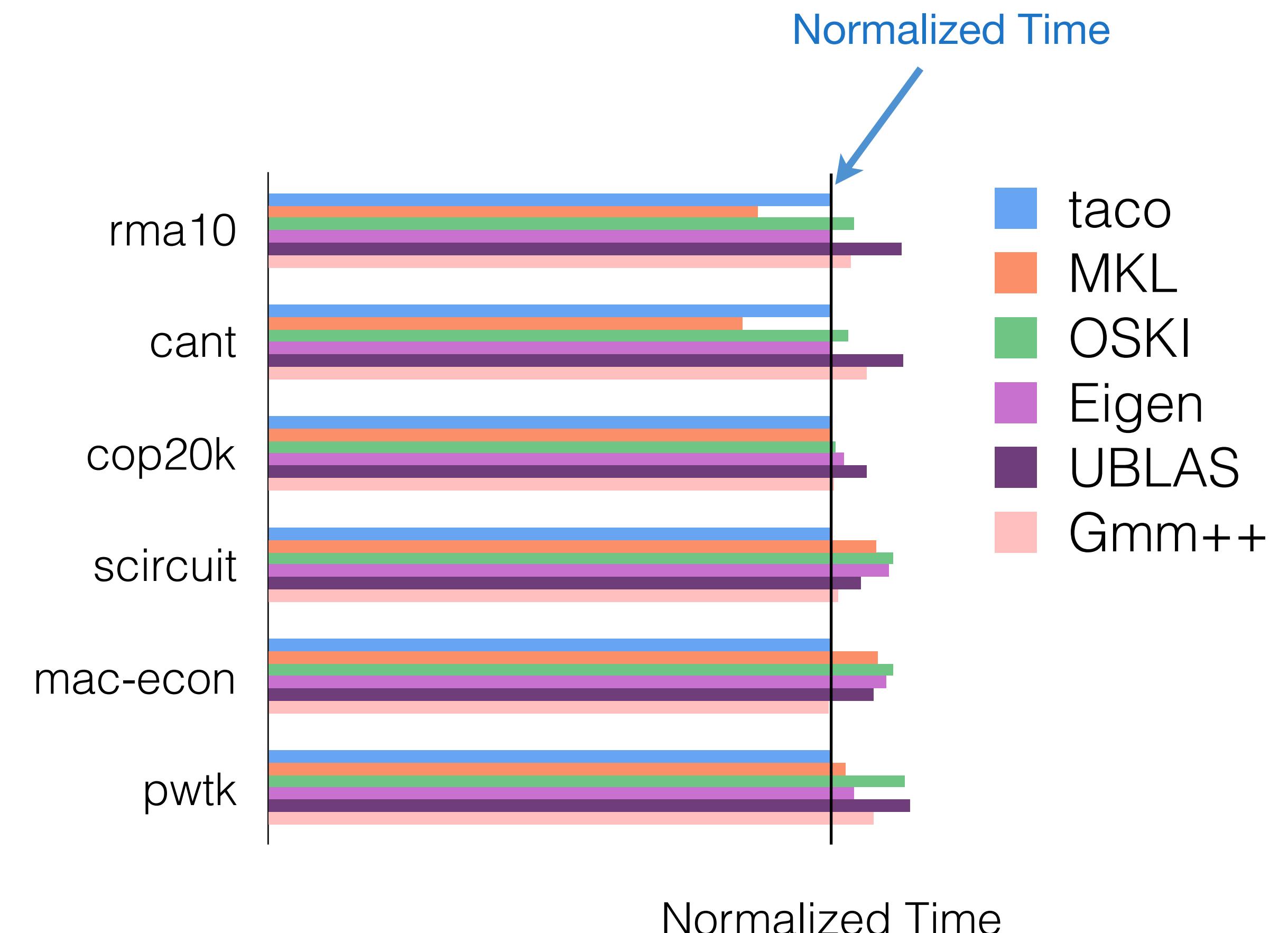
$$A = B + C \quad a = B^T c + d \quad a = b + c \quad A = \alpha A + \beta B$$

$$A_{ik} = \sum_j B_{ijk} * c_j \quad A = 0 \quad A = B + C + D \quad A = (B + C)(d + e)$$

$$A = \alpha B \quad A = (B + C)d \quad A = BC \quad A_{ilk} = \sum_j B_{ijk} * C_{lj}$$

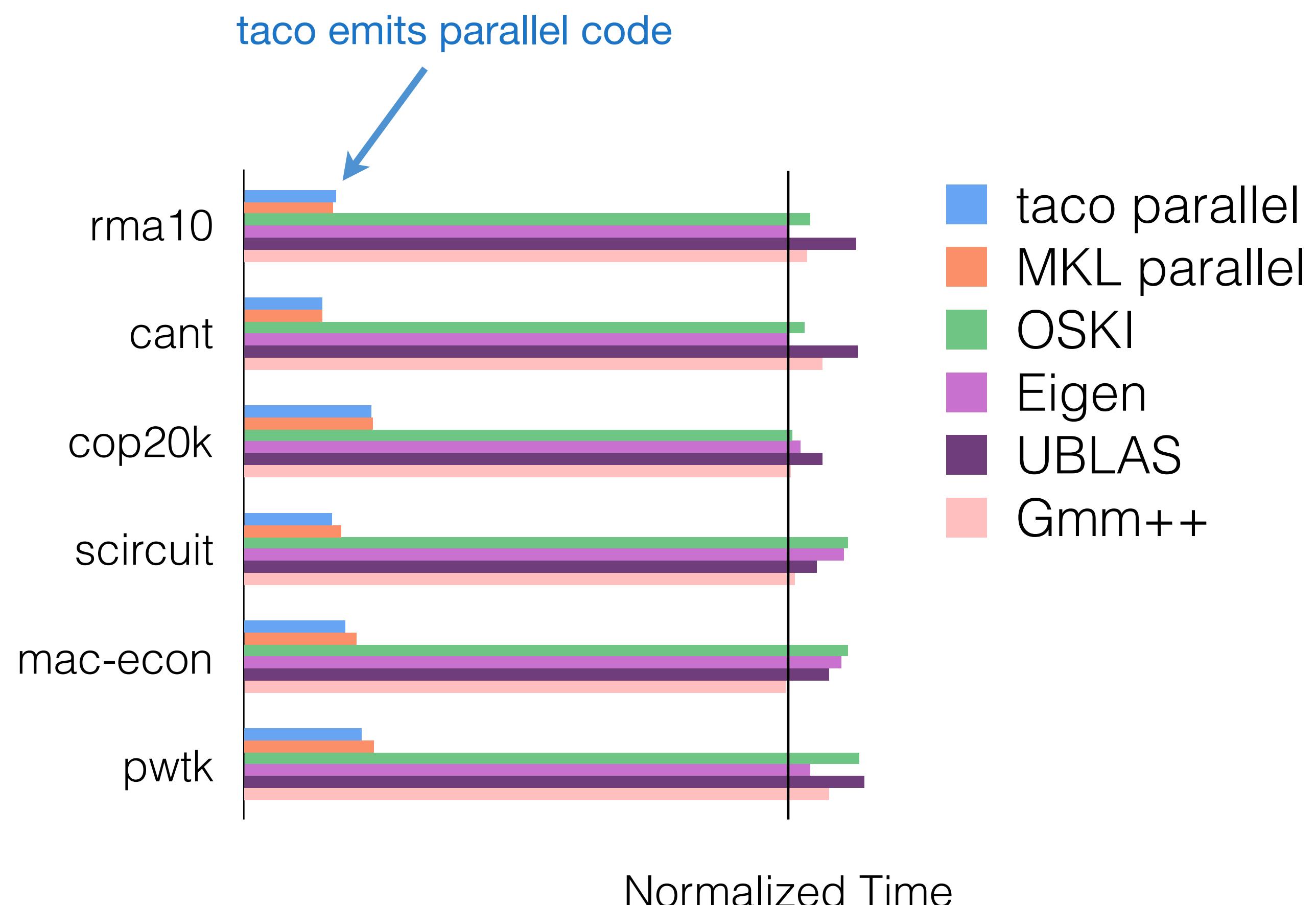
$$A_{ijl} = \sum_k B_{ijk} * C_{lk} \quad a = BCd \quad a = Bc + b \quad A = \sum_j A + \alpha I$$

$$A = \alpha B + A \quad A_{jl} = \sum_{i,k} B_{ijk} * C_{il} * D_{kl} \quad a = \alpha Bc + \beta a$$



SpMV is parallel

$$\begin{aligned}
& y = \alpha A^T x + \beta z \quad \boxed{a = Bc} \\
& a = B^T c \quad A = B + C + D \quad y = b - Ax \\
& a = \begin{matrix} A \\ B \end{matrix} \quad A_{ijk} = \sum_l B_{ijl} C_{kl} \quad A = B \circ (CD) \\
& a = Bc + a \quad A_{ijk} = B_{ijk} + C_{ijk} \quad A_{ij} \sum_k B_{ijk} c_k \\
& A = B \odot C \quad A_{ijk} = B_{ijk} + C_{ijk} \quad A_{ij} \sum_k B_{ijk} c_k \\
& A = B^T \quad \alpha = \sum_{i,j,k} B_{ijk} C_{ijk} \quad A_{ij} = \sum_{k,l} B_{ikl} C_{kj} D_{lj} \\
& A = B + C \quad a = B^T c + d \quad a = b + c \quad A = \alpha A + \beta B \\
& A_{ik} = \sum_j B_{ijk} * c_j \quad A = 0 \quad A = B + C + D \quad A = (B + C)(d + e) \\
& A = \alpha B \quad A = (B + C)d \quad A = BC \quad A_{ilk} = \sum_j B_{ijk} * C_{lj} \\
& A_{ijl} = \sum_k B_{ijk} * C_{lk} \quad a = BCd \quad a = Bc + b \quad A = \sum_j A + \alpha I \\
& A = \alpha B + A \quad A_{jl} = \sum_{i,k} B_{ijk} * C_{il} * D_{kl} \quad a = \alpha Bc + \beta a
\end{aligned}$$



SDDMM must be computed in a single kernel

$$y = \alpha A^T x + \beta z \quad a = Bc$$

Sampled Dense-Dense Matrix Multiplication (SDDMM)

$$a = B^T c \quad A = B + C + D \quad y = b - Ax$$

$$a = \frac{A - B}{Bc + a} \quad A_{ijk} = \sum_l B_{ijl} C_{kl} \quad \boxed{A = B \circ (CD)}$$

$$A = B \odot C \quad A_{ijk} = B_{ijk} + C_{ijk} \quad A_{ij} \sum_k B_{ijk} c_k$$

$$A = B^T \quad \alpha = \sum_{i,j,k} B_{ijk} C_{ijk} \quad A_{ij} = \sum_{k,l} B_{ikl} C_{kj} D_{lj}$$

$$A = B + C \quad a = B^T c + d \quad a = b + c \quad A = \alpha A + \beta B$$

$$A_{ik} = \sum_j B_{ijk} * c_j \quad A = 0 \quad A = B + C + D \quad A = (B + C)(d + e)$$

$$A = \alpha B \quad A = (B + C)d \quad A = BC \quad A_{ilk} = \sum_j B_{ijk} * C_{lj}$$

$$A_{ijl} = \sum_k B_{ijk} * C_{lk} \quad a = BCd \quad a = Bc + b \quad A = \overset{j}{\underset{k}{\overset{l}{A}}} + \alpha I$$

$$A = \alpha B + A \quad A_{jl} = \sum_{i,k} B_{ijk} * C_{il} * D_{kl} \quad a = \alpha Bc + \beta a$$

SDDMM must be computed in a single kernel

$$y = \alpha A^T x + \beta z \quad a = Bc$$

$$a = B^T c \quad A = B + C + D \quad y = b - Ax$$

$$a = \frac{A = B}{Bc + a} \quad A_{ijk} = \sum_l B_{ijl} C_{kl} \quad \boxed{A = B \circ (CD)}$$

$$A = B \odot C \quad A_{ijk} = B_{ijk} + C_{ijk} \quad A_{ij} \sum_k B_{ijk} c_k$$

$$A = B^T \quad \alpha = \sum_{i,j,k} B_{ijk} C_{ijk} \quad A_{ij} = \sum_{k,l} B_{ikl} C_{kj} D_{lj}$$

$$A = B + C \quad a = B^T c + d \quad a = b + c \quad A = \alpha A + \beta B$$

$$A_{ik} = \sum_j B_{ijk} * c_j \quad A = 0 \quad A = B + C + D \quad A = (B + C)(d + e)$$

$$A = \alpha B \quad A = (B + C)d \quad A = BC \quad A_{ilk} = \sum_j B_{ijk} * C_{lj}$$

$$A_{ijl} = \sum_k B_{ijk} * C_{lk} \quad a = BCd \quad a = Bc + b \quad A = \overset{j}{\underset{i}{\overset{l}{A}}} + \alpha I$$

$$A = \alpha B + A \quad A_{jl} = \sum_{i,k} B_{ijk} * C_{il} * D_{kl} \quad a = \alpha Bc + \beta a$$

Sampled Dense-Dense Matrix Multiplication (SDDMM)

C



SDDMM must be computed in a single kernel

$$y = \alpha A^T x + \beta z \quad a = Bc$$

$$a = B^T c \quad A = B + C + D \quad y = b - Ax$$

$$a = \frac{A = B}{Bc + a} \quad A_{ijk} = \sum_l B_{ijl} C_{kl} \quad \boxed{A = B \circ (CD)}$$

$$A = B \odot C \quad A_{ijk} = B_{ijk} + C_{ijk} \quad A_{ij} \sum_k B_{ijk} c_k$$

$$A = B^T \quad \alpha = \sum_{i,j,k} B_{ijk} C_{ijk} \quad A_{ij} = \sum_{k,l} B_{ikl} C_{kj} D_{lj}$$

$$A = B + C \quad a = B^T c + d \quad a = b + c \quad A = \alpha A + \beta B$$

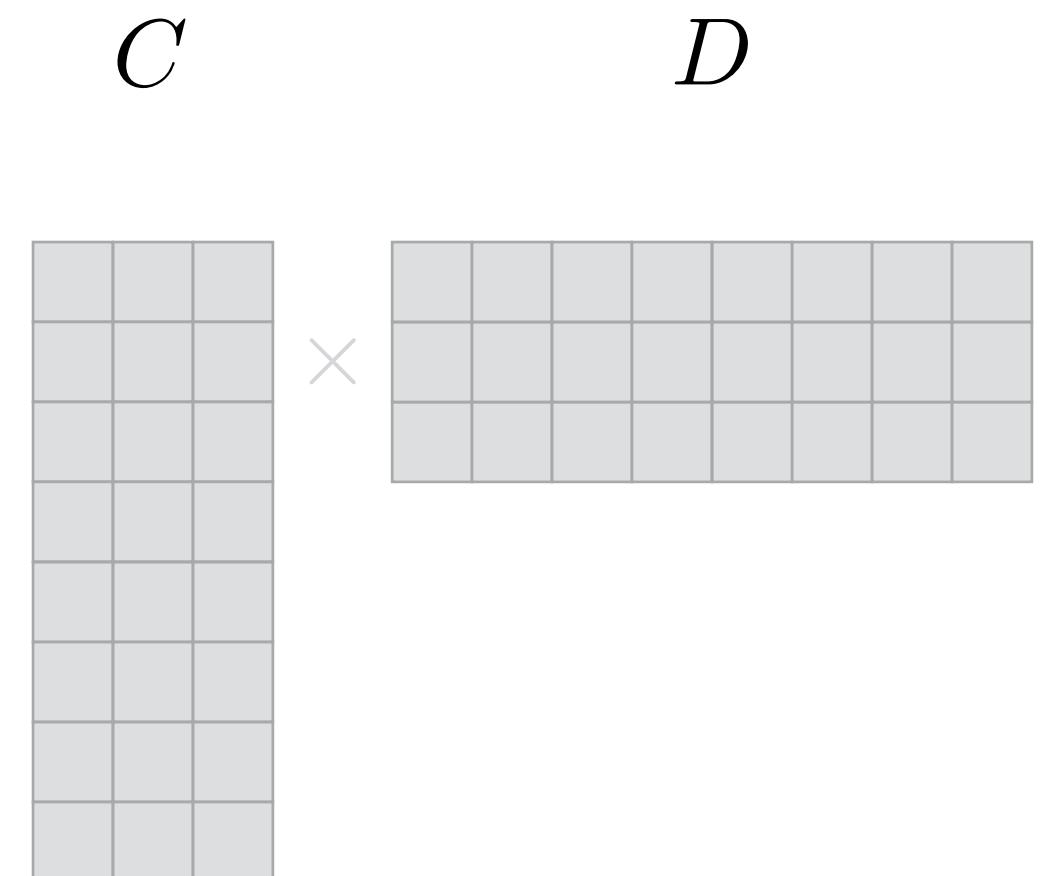
$$A_{ik} = \sum_j B_{ijk} * c_j \quad A = 0 \quad A = B + C + D \quad A = (B + C)(d + e)$$

$$A = \alpha B \quad A = (B + C)d \quad A = BC \quad A_{ilk} = \sum_j B_{ijk} * C_{lj}$$

$$A_{ijl} = \sum_k B_{ijk} * C_{lk} \quad a = BCd \quad a = Bc + b \quad A = \sum_j A + \alpha I$$

$$A = \alpha B + A \quad A_{jl} = \sum_{i,k} B_{ijk} * C_{il} * D_{kl} \quad a = \alpha Bc + \beta a$$

Sampled Dense-Dense Matrix Multiplication (SDDMM)



SDDMM must be computed in a single kernel

$$y = \alpha A^T x + \beta z \quad a = Bc$$

$$a = B^T c \quad A = B + C + D \quad y = b - Ax$$

$$a = \begin{matrix} A = B \\ Bc + a \end{matrix} \quad A_{ijk} = \sum_l B_{ijl} C_{kl} \quad \boxed{A = B \circ (CD)}$$

$$A = B \odot C \quad A_{ijk} = B_{ijk} + C_{ijk} \quad A_{ij} \sum_k B_{ijk} c_k$$

$$A = B^T \quad \alpha = \sum_{i,j,k} B_{ijk} C_{ijk} \quad A_{ij} = \sum_{k,l} B_{ikl} C_{kj} D_{lj}$$

$$A = B + C \quad a = B^T c + d \quad a = b + c \quad A = \alpha A + \beta B$$

$$A_{ik} = \sum_j B_{ijk} * c_j \quad A = 0 \quad A = B + C + D \quad A = (B + C)(d + e)$$

$$A = \alpha B \quad A = (B + C)d \quad A = BC \quad A_{ilk} = \sum_j B_{ijk} * C_{lj}$$

$$A_{ijl} = \sum_k B_{ijk} * C_{lk} \quad a = BCd \quad a = Bc + b \quad A = \sum_j A + \alpha I$$

$$A = \alpha B + A \quad A_{jl} = \sum_{i,k} B_{ijk} * C_{il} * D_{kl} \quad a = \alpha Bc + \beta a$$

Sampled Dense-Dense Matrix Multiplication (SDDMM)

$$\begin{array}{c} C \\ \times \\ D \end{array}$$

1	2	3	4	5	6	7	8
1							
2							
3							
4							
5							
6							
7							
8							

64 inner product

SDDMM must be computed in a single kernel

$$y = \alpha A^T x + \beta z \quad a = Bc$$

$$a = B^T c \quad A = B + C + D \quad y = b - Ax$$

$$a = \begin{matrix} A = B \\ Bc + a \end{matrix} \quad A_{ijk} = \sum_l B_{ijl} C_{kl} \quad \boxed{A = B \circ (CD)}$$

$$A = B \odot C \quad A_{ijk} = B_{ijk} + C_{ijk} \quad A_{ij} \sum_k B_{ijk} c_k$$

$$A = B^T \quad \alpha = \sum_{i,j,k} B_{ijk} C_{ijk} \quad A_{ij} = \sum_{k,l} B_{ikl} C_{kj} D_{lj}$$

$$A = B + C \quad a = B^T c + d \quad a = b + c \quad A = \alpha A + \beta B$$

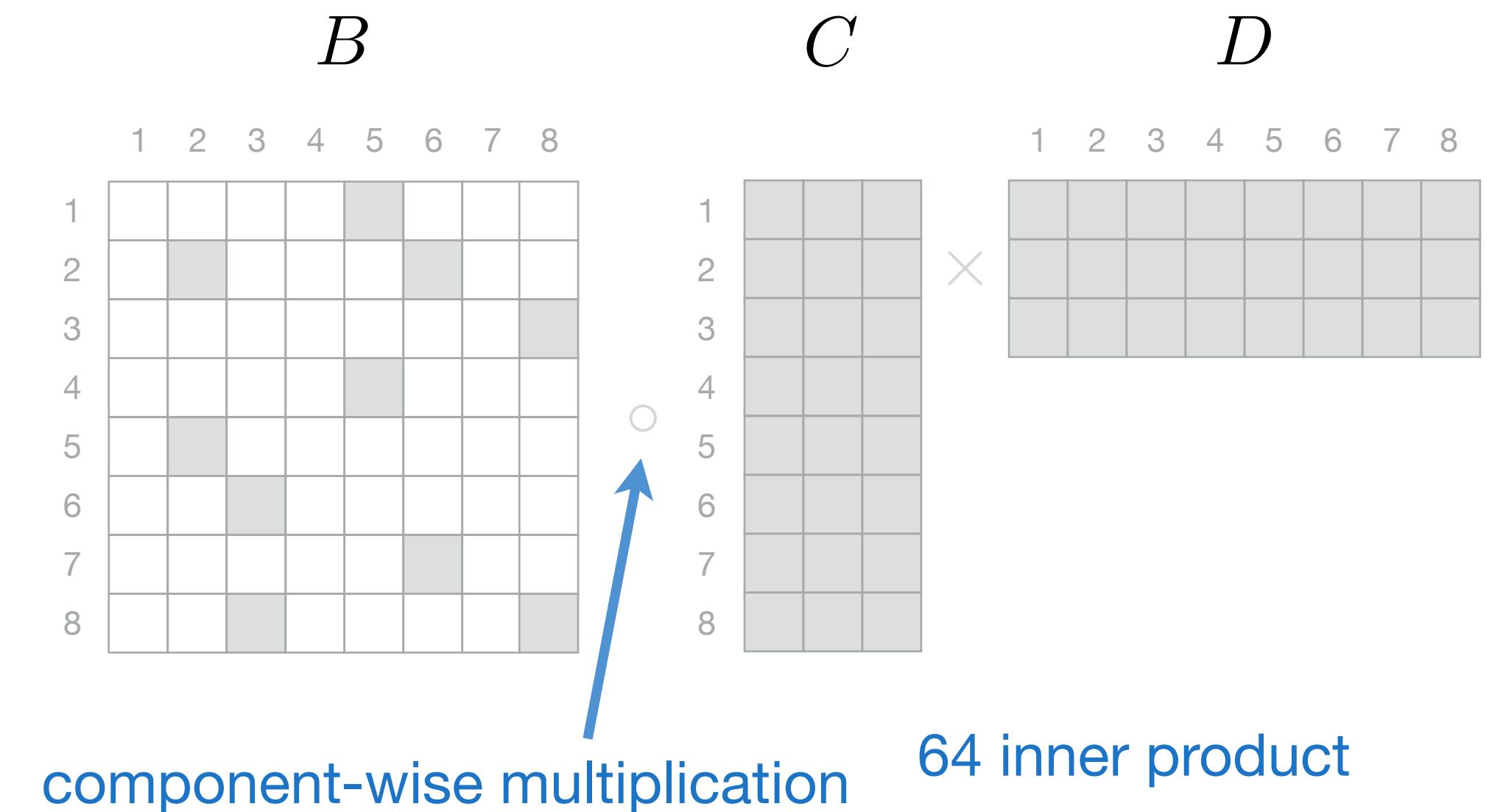
$$A_{ik} = \sum_j B_{ijk} * c_j \quad A = 0 \quad A = B + C + D \quad A = (B + C)(d + e)$$

$$A = \alpha B \quad A = (B + C)d \quad A = BC \quad A_{ilk} = \sum_j B_{ijk} * C_{lj}$$

$$A_{ijl} = \sum_k B_{ijk} * C_{lk} \quad a = BCd \quad a = Bc + b \quad A = \sum_j A + \alpha I$$

$$A = \alpha B + A \quad A_{jl} = \sum_{i,k} B_{ijk} * C_{il} * D_{kl} \quad a = \alpha Bc + \beta a$$

Sampled Dense-Dense Matrix Multiplication (SDDMM)



SDDMM must be computed in a single kernel

$$y = \alpha A^T x + \beta z \quad a = Bc$$

$$a = B^T c \quad A = B + C + D \quad y = b - Ax$$

$$a = \begin{matrix} A = B \\ Bc + a \end{matrix} \quad A_{ijk} = \sum_l B_{ijl} C_{kl} \quad \boxed{A = B \circ (CD)}$$

$$A = B \odot C \quad A_{ijk} = B_{ijk} + C_{ijk} \quad A_{ij} \sum_k B_{ijk} c_k$$

$$A = B^T \quad \alpha = \sum_{i,j,k} B_{ijk} C_{ijk} \quad A_{ij} = \sum_{k,l} B_{ikl} C_{kj} D_{lj}$$

$$A = B + C \quad a = B^T c + d \quad a = b + c \quad A = \alpha A + \beta B$$

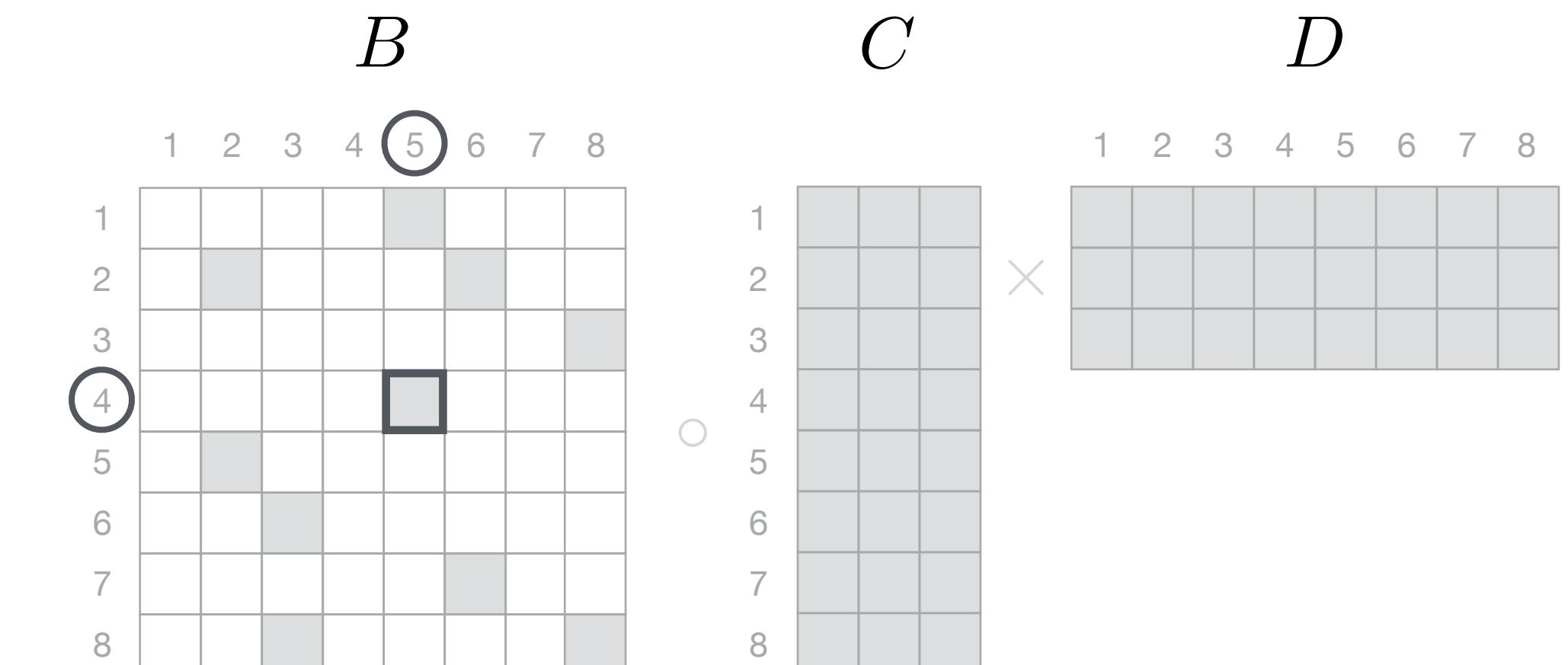
$$A_{ik} = \sum_j B_{ijk} * c_j \quad A = 0 \quad A = B + C + D \quad A = (B + C)(d + e)$$

$$A = \alpha B \quad A = (B + C)d \quad A = BC \quad A_{ilk} = \sum_j B_{ijk} * C_{lj}$$

$$A_{ijl} = \sum_k B_{ijk} * C_{lk} \quad a = BCd \quad a = Bc + b \quad A = \sum_j A + \alpha I$$

$$A = \alpha B + A \quad A_{jl} = \sum_{i,k} B_{ijk} * C_{il} * D_{kl} \quad a = \alpha Bc + \beta a$$

Sampled Dense-Dense Matrix Multiplication (SDDMM)



SDDMM must be computed in a single kernel

$$y = \alpha A^T x + \beta z \quad a = Bc$$

$$a = B^T c \quad A = B + C + D \quad y = b - Ax$$

$$a = \begin{matrix} A = B \\ Bc + a \end{matrix} \quad A_{ijk} = \sum_l B_{ijl} C_{kl} \quad \boxed{A = B \circ (CD)}$$

$$A = B \odot C \quad A_{ijk} = B_{ijk} + C_{ijk} \quad A_{ij} \sum_k B_{ijk} c_k$$

$$A = B^T \quad \alpha = \sum_{i,j,k} B_{ijk} C_{ijk} \quad A_{ij} = \sum_{k,l} B_{ikl} C_{kj} D_{lj}$$

$$A = B + C \quad a = B^T c + d \quad a = b + c \quad A = \alpha A + \beta B$$

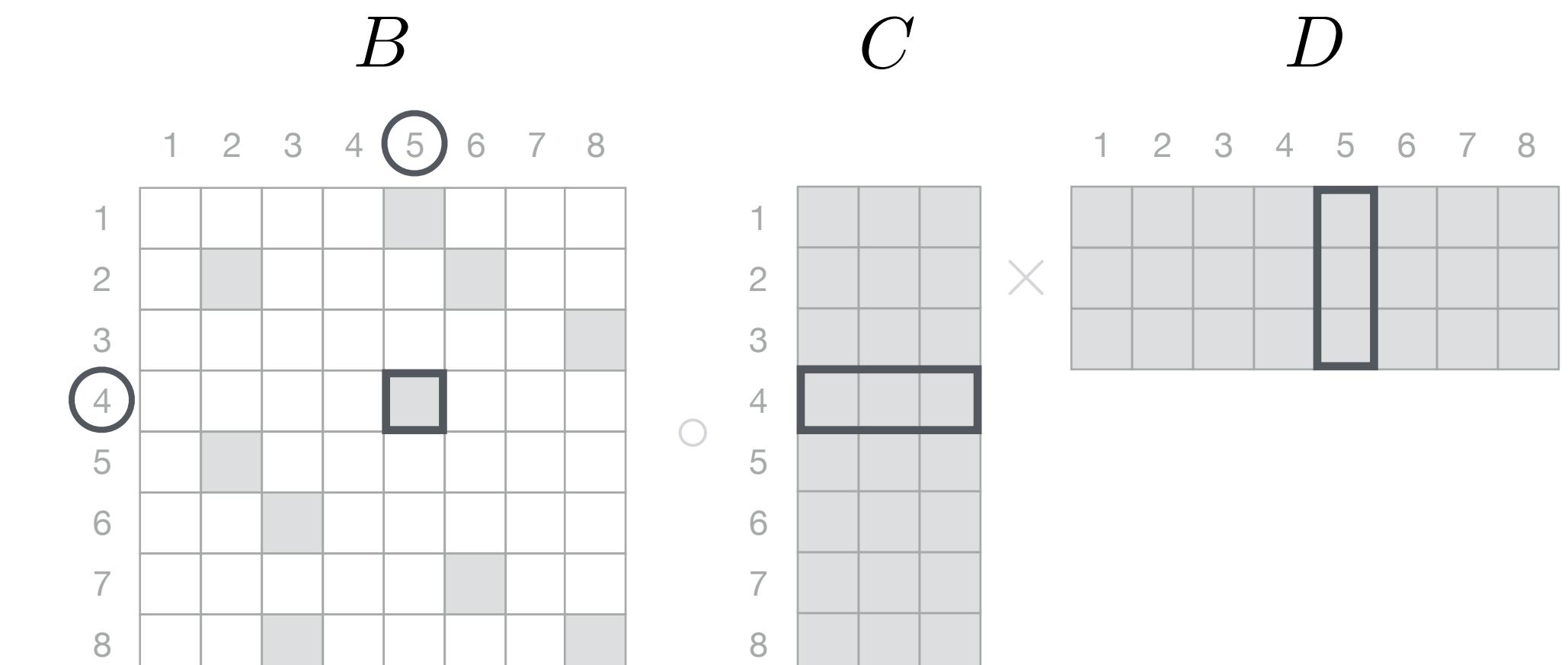
$$A_{ik} = \sum_j B_{ijk} * c_j \quad A = 0 \quad A = B + C + D \quad A = (B + C)(d + e)$$

$$A = \alpha B \quad A = (B + C)d \quad A = BC \quad A_{ilk} = \sum_j B_{ijk} * C_{lj}$$

$$A_{ijl} = \sum_k B_{ijk} * C_{lk} \quad a = BCd \quad a = Bc + b \quad A = \sum_j A + \alpha I$$

$$A = \alpha B + A \quad A_{jl} = \sum_{i,k} B_{ijk} * C_{il} * D_{kl} \quad a = \alpha Bc + \beta a$$

Sampled Dense-Dense Matrix Multiplication (SDDMM)



64 inner product

SDDMM must be computed in a single kernel

$$y = \alpha A^T x + \beta z \quad a = Bc$$

$$a = B^T c \quad A = B + C + D \quad y = b - Ax$$

$$a = \begin{matrix} A = B \\ Bc + a \end{matrix} \quad A_{ijk} = \sum_l B_{ijl} C_{kl} \quad \boxed{A = B \circ (CD)}$$

$$A = B \odot C \quad A_{ijk} = B_{ijk} + C_{ijk} \quad A_{ij} \sum_k B_{ijk} c_k$$

$$A = B^T \quad \alpha = \sum_{i,j,k} B_{ijk} C_{ijk} \quad A_{ij} = \sum_{k,l} B_{ikl} C_{kj} D_{lj}$$

$$A = B + C \quad a = B^T c + d \quad a = b + c \quad A = \alpha A + \beta B$$

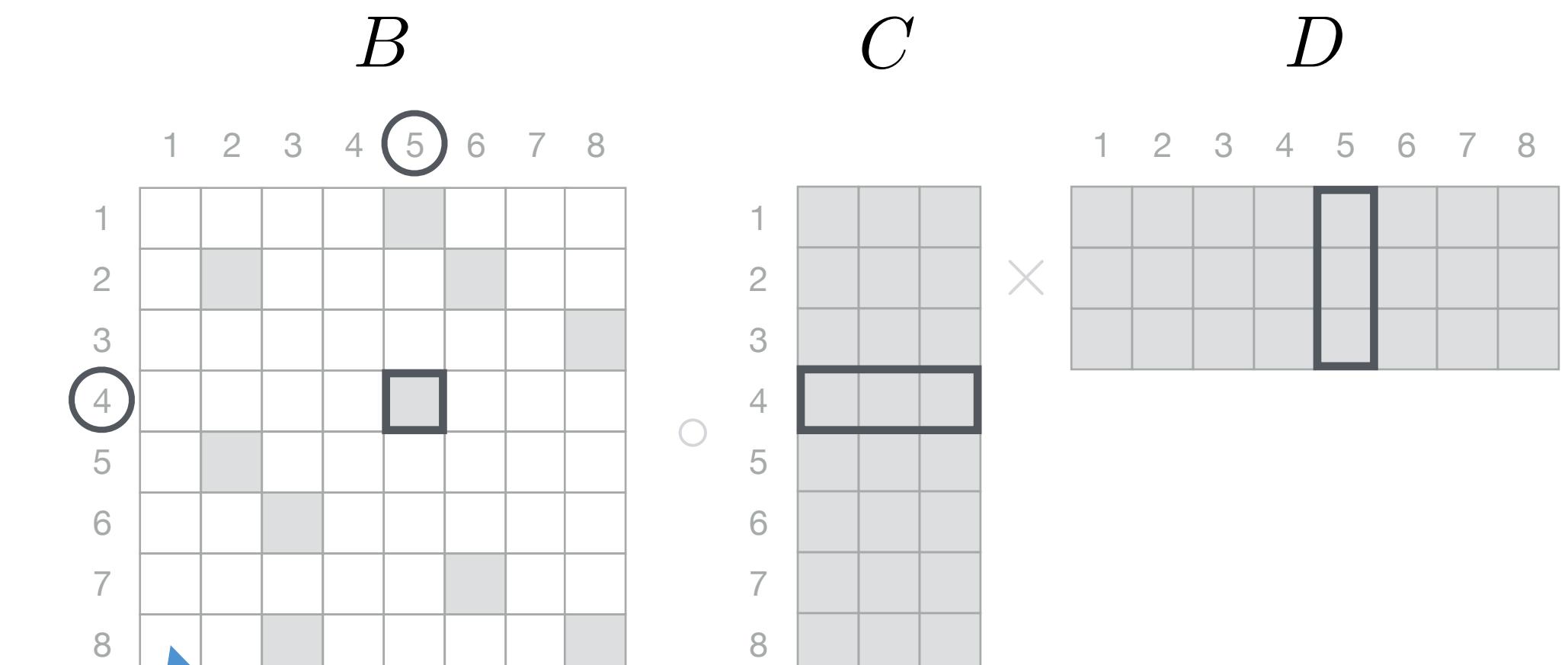
$$A_{ik} = \sum_j B_{ijk} * c_j \quad A = 0 \quad A = B + C + D \quad A = (B + C)(d + e)$$

$$A = \alpha B \quad A = (B + C)d \quad A = BC \quad A_{ilk} = \sum_j B_{ijk} * C_{lj}$$

$$A_{ijl} = \sum_k B_{ijk} * C_{lk} \quad a = BCd \quad a = Bc + b \quad A = \sum_j A + \alpha I$$

$$A = \alpha B + A \quad A_{jl} = \sum_{i,k} B_{ijk} * C_{il} * D_{kl} \quad a = \alpha Bc + \beta a$$

Sampled Dense-Dense Matrix Multiplication (SDDMM)



64 inner product

this dot product need not be computed

SDDMM must be computed in a single kernel

$$y = \alpha A^T x + \beta z \quad a = Bc$$

$$a = B^T c \quad A = B + C + D \quad y = b - Ax$$

$$a = \begin{matrix} A = B \\ Bc + a \end{matrix} \quad A_{ijk} = \sum_l B_{ijl} C_{kl} \quad \boxed{A = B \circ (CD)}$$

$$A = B \odot C \quad A_{ijk} = B_{ijk} + C_{ijk} \quad A_{ij} \sum_k B_{ijk} c_k$$

$$A = B^T \quad \alpha = \sum_{i,j,k} B_{ijk} C_{ijk} \quad A_{ij} = \sum_{k,l} B_{ikl} C_{kj} D_{lj}$$

$$A = B + C \quad a = B^T c + d \quad a = b + c \quad A = \alpha A + \beta B$$

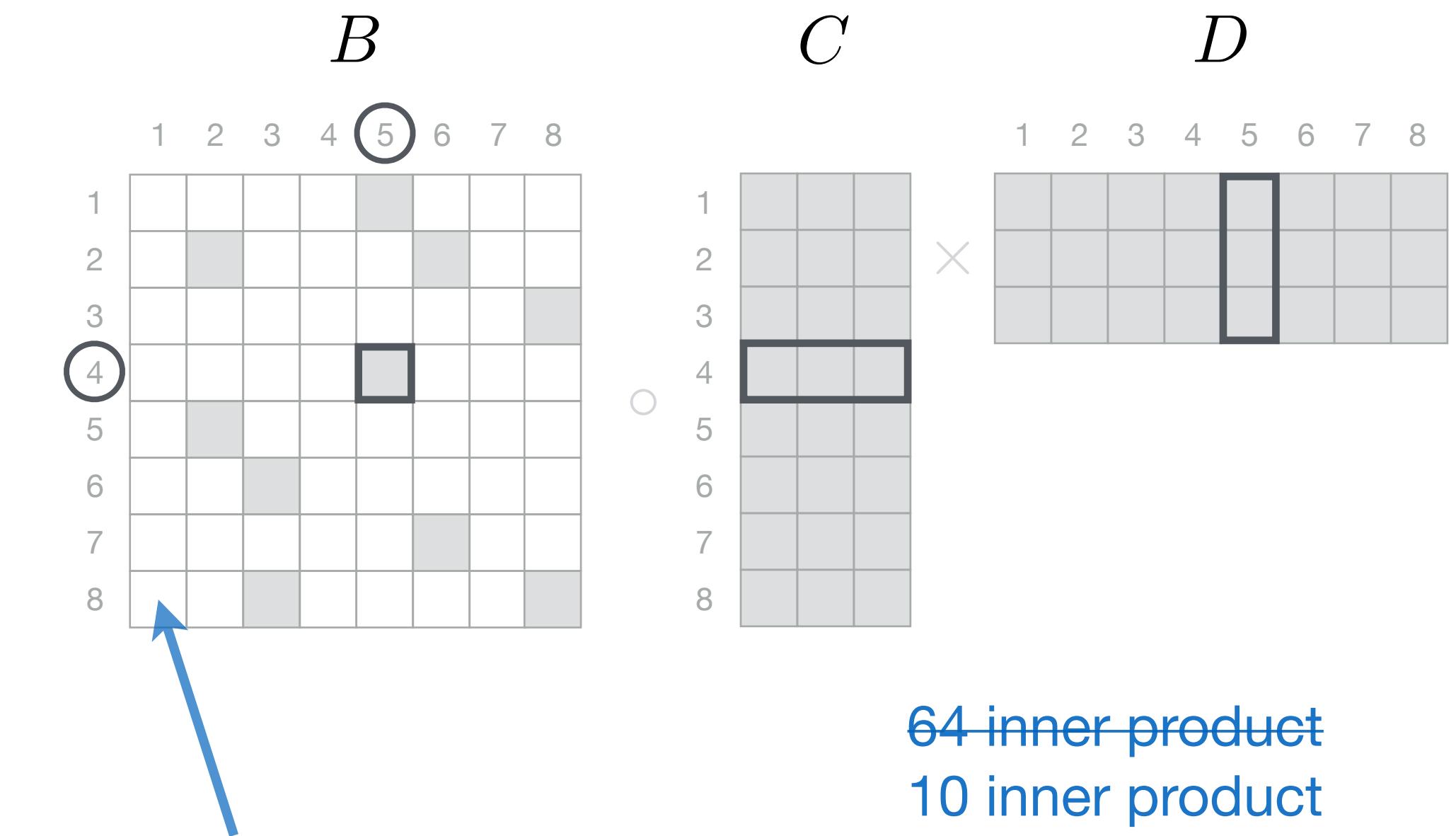
$$A_{ik} = \sum_j B_{ijk} * c_j \quad A = 0 \quad A = B + C + D \quad A = (B + C)(d + e)$$

$$A = \alpha B \quad A = (B + C)d \quad A = BC \quad A_{ilk} = \sum_j B_{ijk} * C_{lj}$$

$$A_{ijl} = \sum_k B_{ijk} * C_{lk} \quad a = BCd \quad a = Bc + b \quad A = \sum_j A + \alpha I$$

$$A = \alpha B + A \quad A_{jl} = \sum_{i,k} B_{ijk} * C_{il} * D_{kl} \quad a = \alpha Bc + \beta a$$

Sampled Dense-Dense Matrix Multiplication (SDDMM)



SDDMM must be computed in a single kernel

$$y = \alpha A^T x + \beta z \quad a = Bc$$

$$a = B^T c \quad A = B + C + D \quad y = b - Ax$$

$$a = Bc + a \quad A_{ijk} = \sum_l B_{ijl} C_{kl} \quad \boxed{A = B \circ (CD)}$$

$$A = B \odot C \quad A_{ijk} = B_{ijk} + C_{ijk} \quad A_{ij} \sum_k B_{ijk} c_k$$

$$A = B^T \quad \alpha = \sum_{i,j,k} B_{ijk} C_{ijk} \quad A_{ij} = \sum_{k,l} B_{ikl} C_{kj} D_{lj}$$

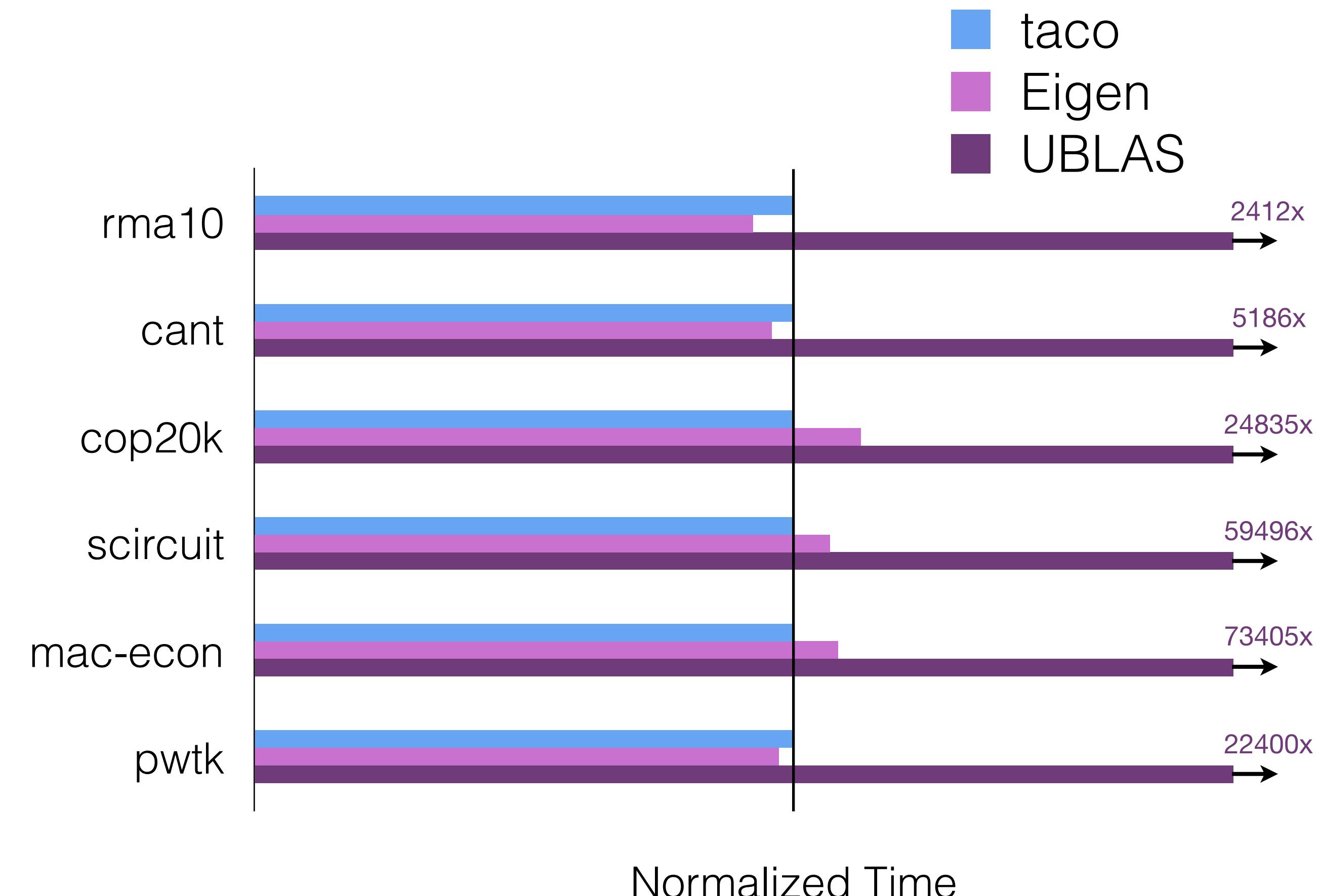
$$A = B + C \quad a = B^T c + d \quad a = b + c \quad A = \alpha A + \beta B$$

$$A_{ik} = \sum_j B_{ijk} * c_j \quad A = 0 \quad A = B + C + D \quad A = (B + C)(d + e)$$

$$A = \alpha B \quad A = (B + C)d \quad A = BC \quad A_{ilk} = \sum_j B_{ijk} * C_{lj}$$

$$A_{ijl} = \sum_k B_{ijk} * C_{lk} \quad a = BCd \quad a = Bc + b \quad A = \sum_j A + \alpha I$$

$$A = \alpha B + A \quad A_{jl} = \sum_{i,k} B_{ijk} * C_{il} * D_{kl} \quad a = \alpha Bc + \beta a$$



Normalized Time

MTTKRP must be computed in a single kernel

$$y = \alpha A^T x + \beta z \quad a = Bc$$

$$a = B^T c \quad A = B + C + D \quad y = b - Ax$$

$$a = \frac{A - B}{Bc + a} \quad A_{ijk} = \sum_l B_{ijl} C_{kl} \quad A = B \circ (CD)$$

$$A = B \odot C \quad A_{ijk} = B_{ijk} + C_{ijk} \quad A_{ij} \sum_k B_{ijk} c_k$$

$$A = B^T \quad \alpha = \sum_{i,j,k} B_{ijk} C_{ijk} \quad \boxed{A_{ij} = \sum_{k,l} B_{ikl} C_{kj} D_{lj}}$$

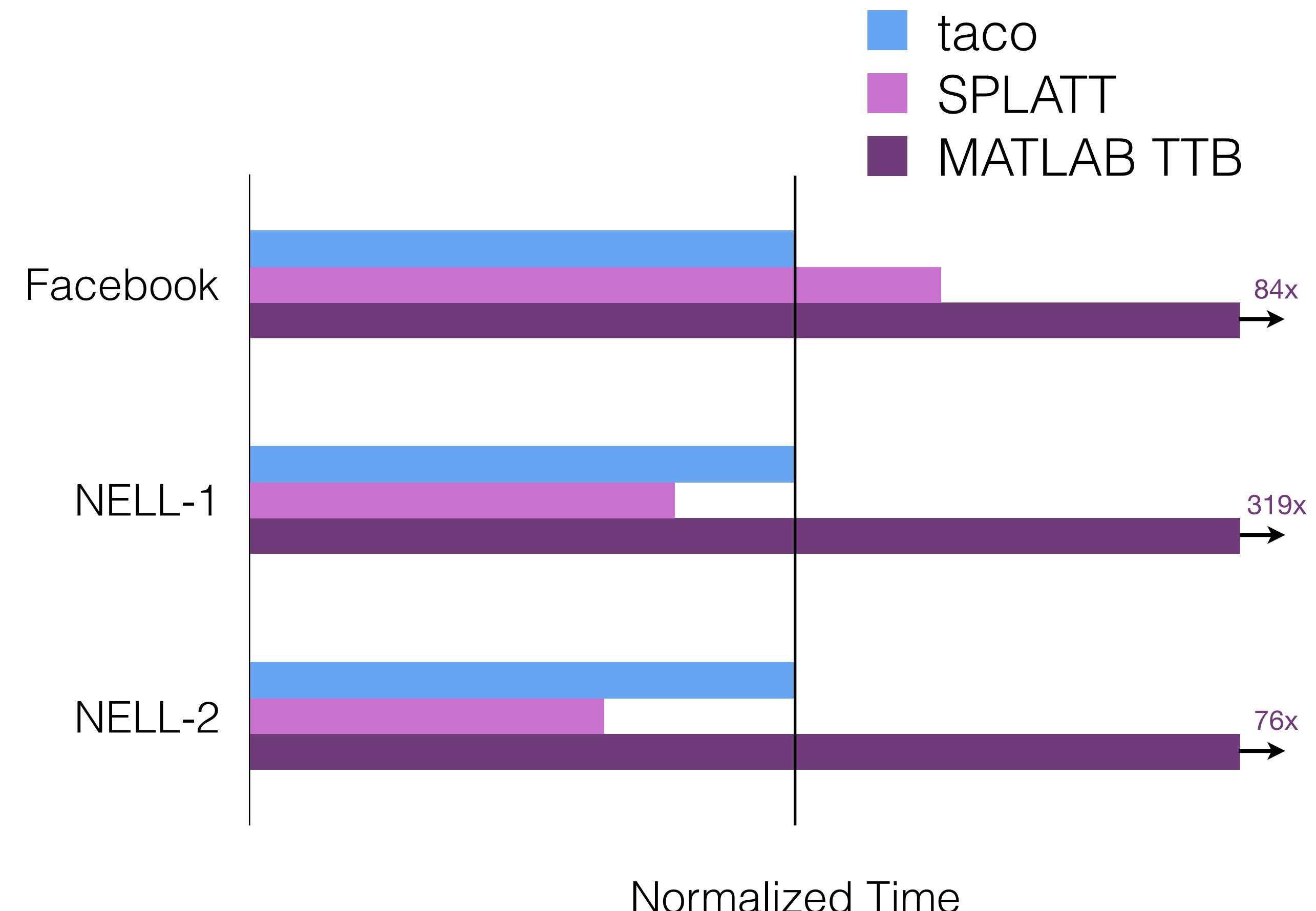
$$A = B + C \quad a = B^T c + d \quad a = b + c \quad A = \alpha A + \beta B$$

$$A_{ik} = \sum_j B_{ijk} * c_j \quad A = 0 \quad A = B + C + D \quad A = (B + C)(d + e)$$

$$A = \alpha B \quad A = (B + C)d \quad A = BC \quad A_{ilk} = \sum_j B_{ijk} * C_{lj}$$

$$A_{ijl} = \sum_k B_{ijk} * C_{lk} \quad a = BCd \quad a = Bc + b \quad A = \sum_j A + \alpha I$$

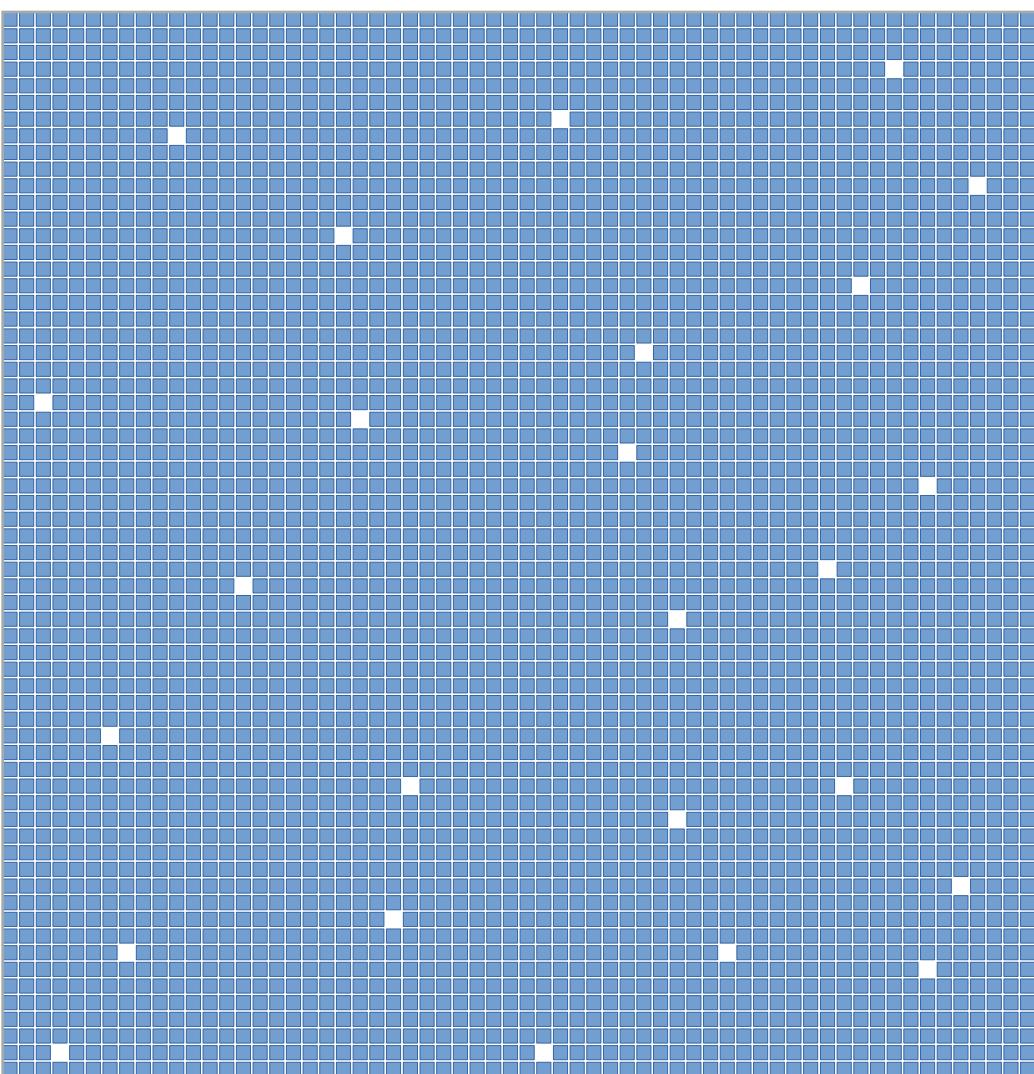
$$A = \alpha B + A \quad A_{jl} = \sum_{i,k} B_{ijk} * C_{il} * D_{kl} \quad a = \alpha Bc + \beta a$$



Normalized Time

One size does not fit all - different matrices need different formats

Dense Matrix



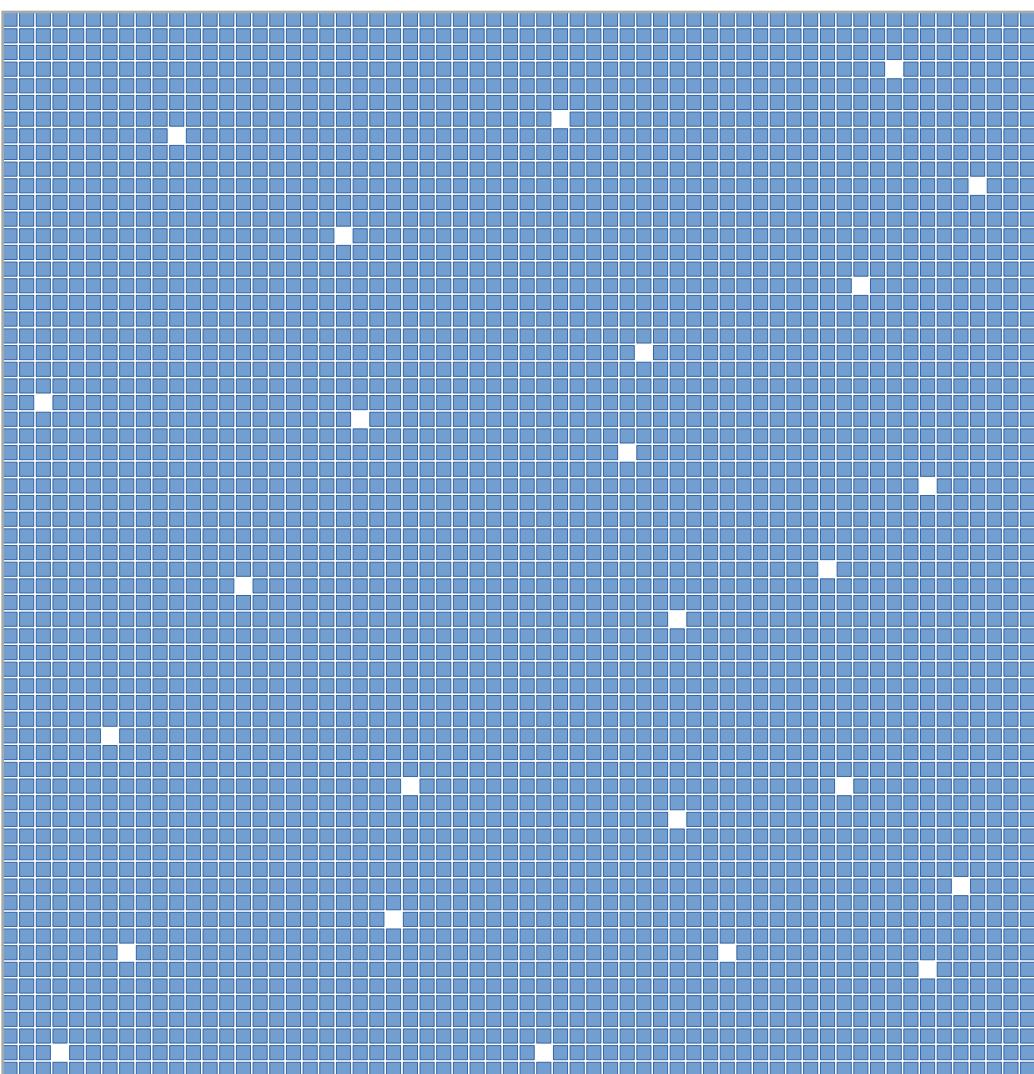
Normalized Time

dense - dense
sparse - dense
dense - sparse
sparse - sparse



One size does not fit all - different matrices need different formats

Dense Matrix



Normalized Time

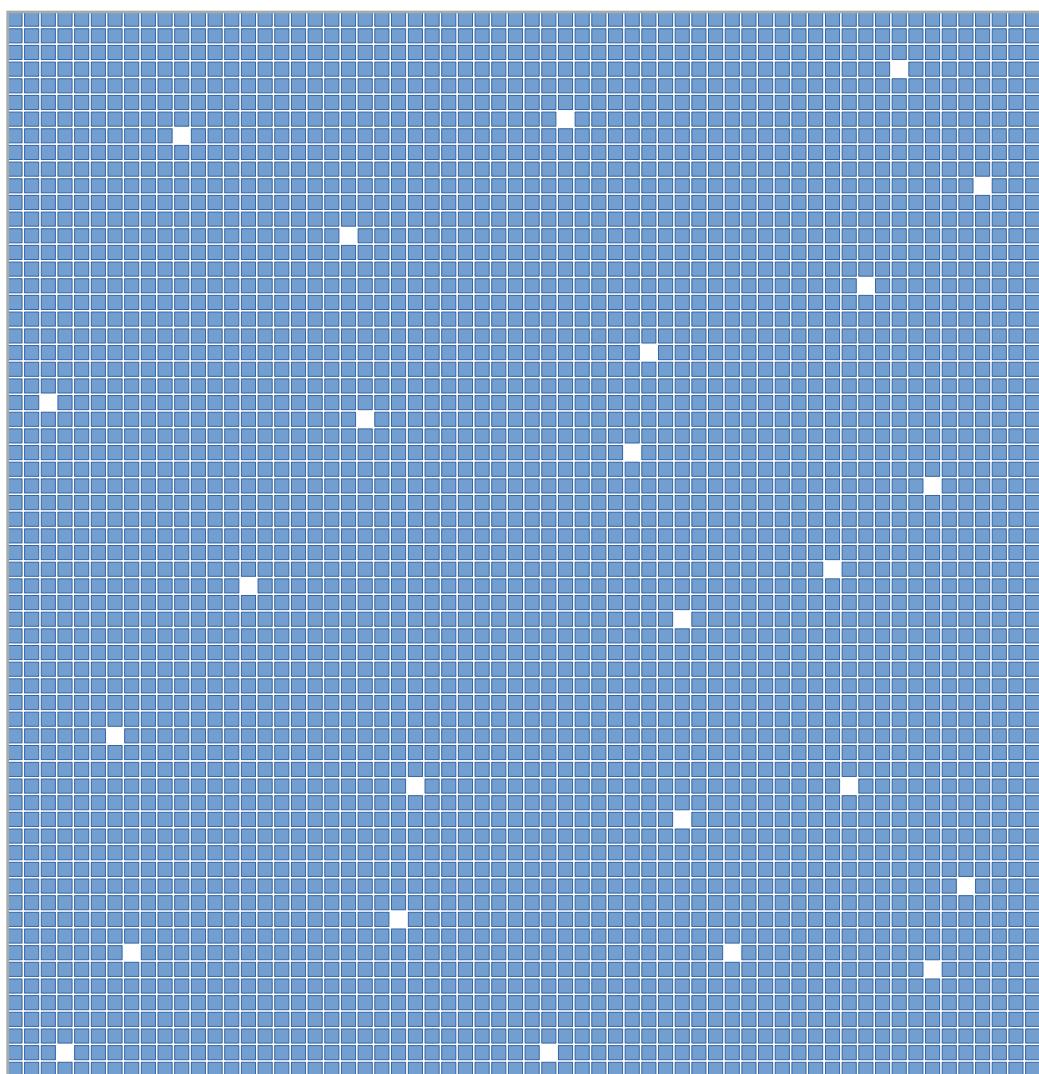


dense - dense
sparse - dense
dense - sparse
sparse - sparse

Formats (row-major)

One size does not fit all - different matrices need different formats

Dense Matrix



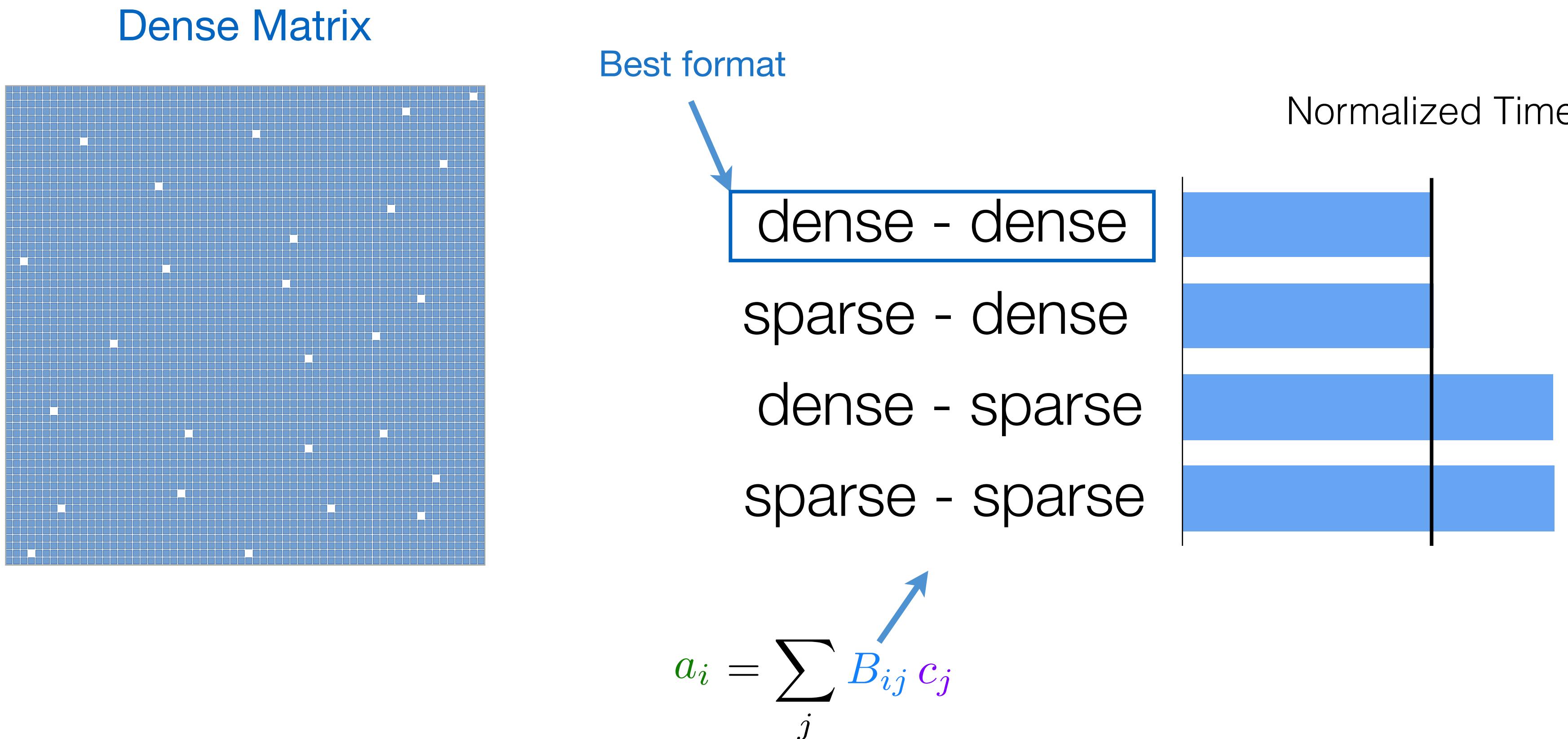
Normalized Time



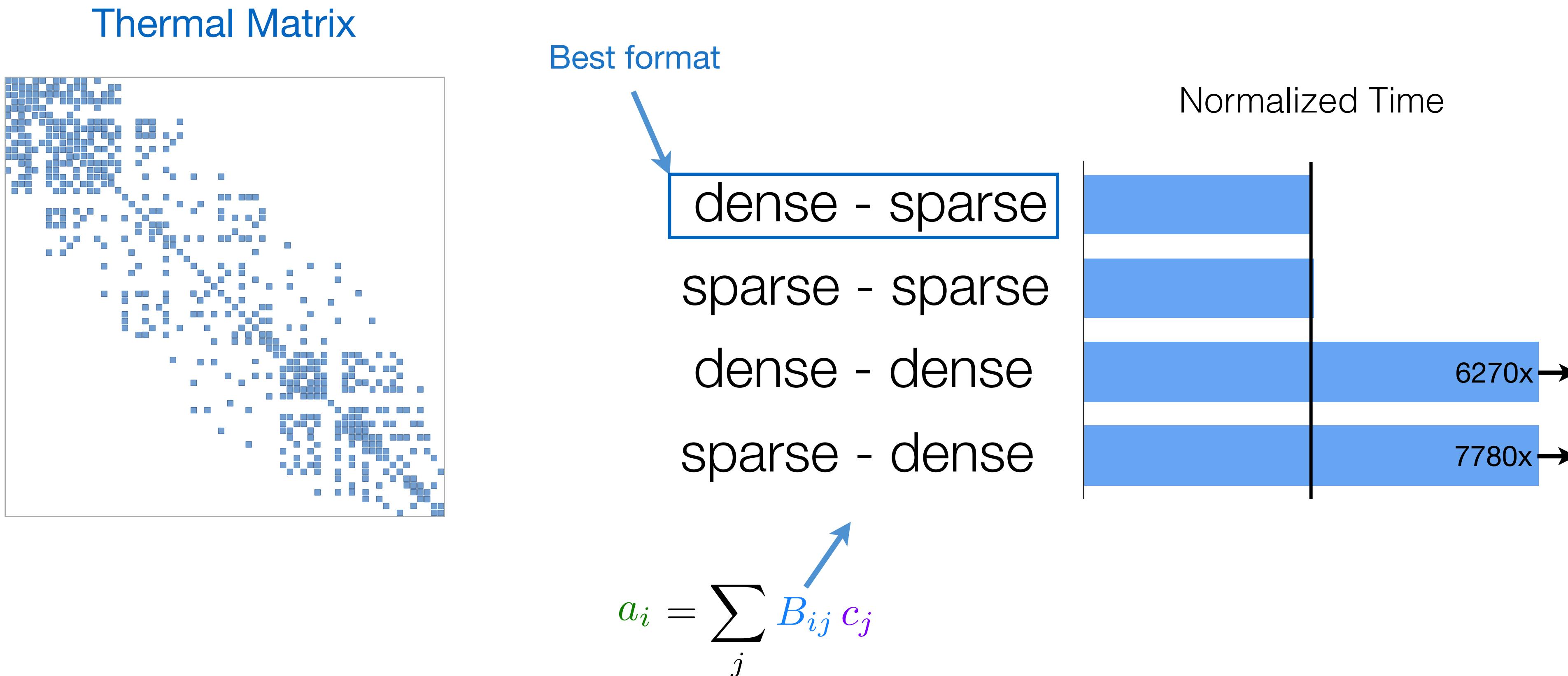
dense - dense
sparse - dense
dense - sparse
sparse - sparse

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

One size does not fit all - different matrices need different formats

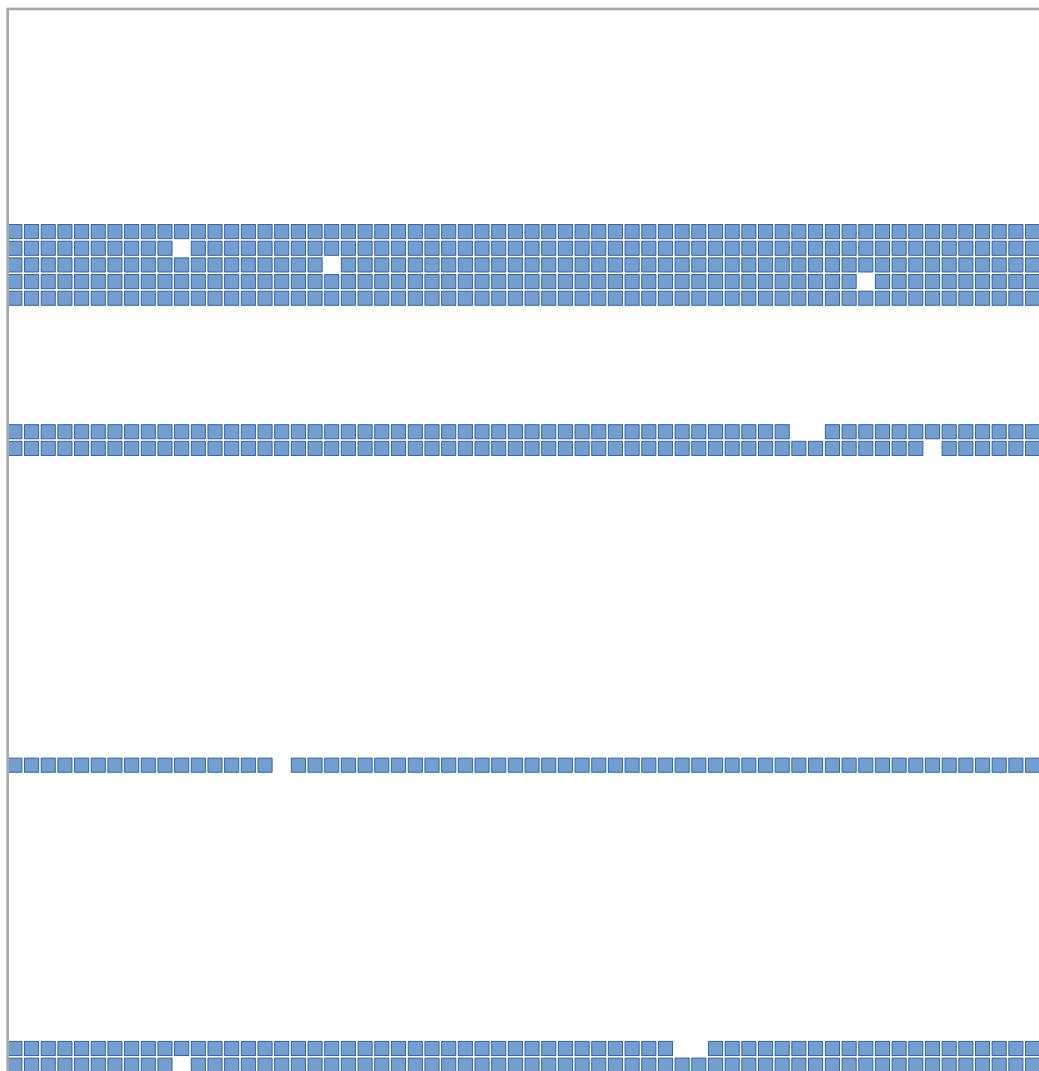


One size does not fit all - different matrices need different formats



One size does not fit all - different matrices need different formats

Row-sliced Matrix



Best format

sparse - dense

sparse - sparse

dense - sparse

dense - dense

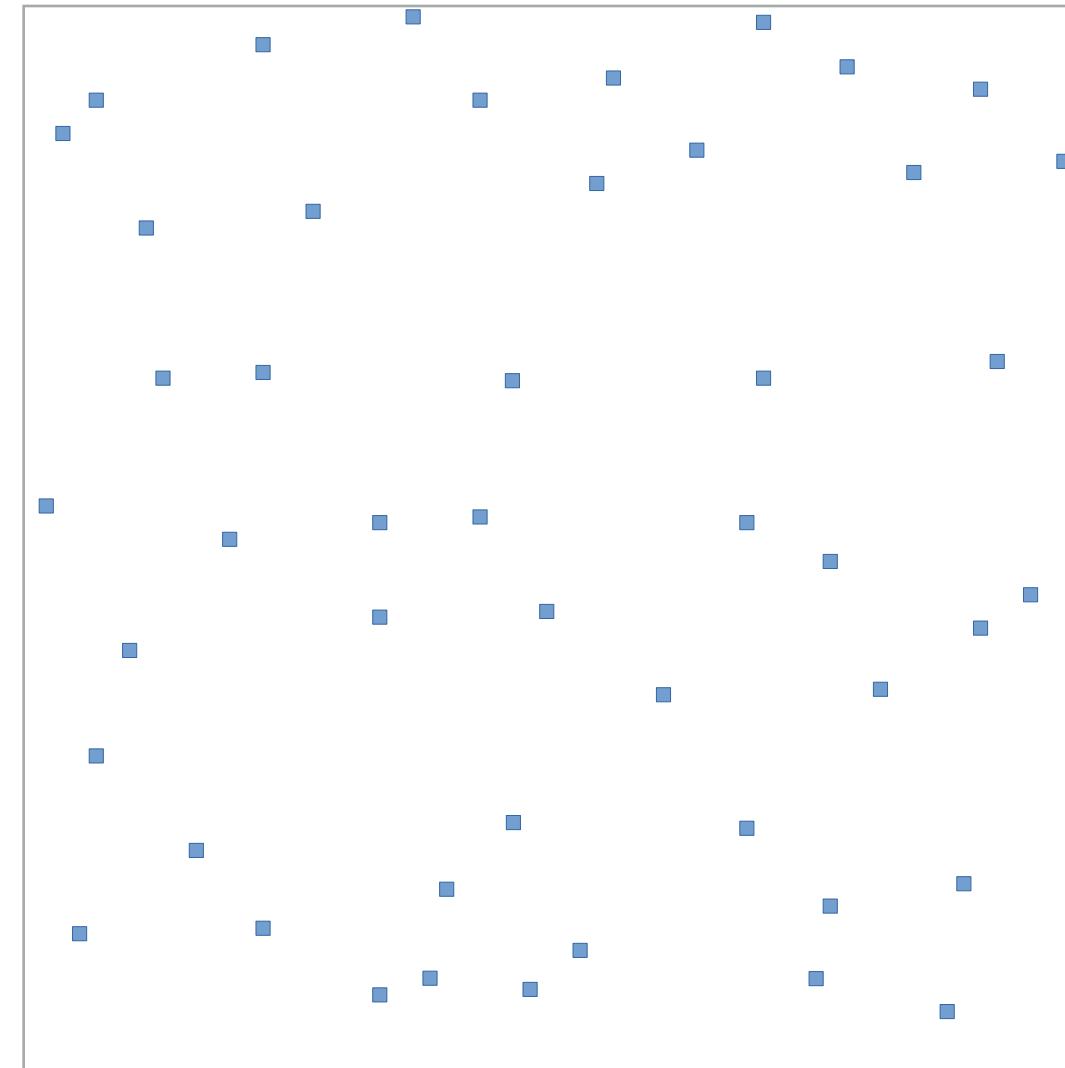
Normalized Time



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

One size does not fit all - different matrices need different formats

Hypersparse Matrix



Best format

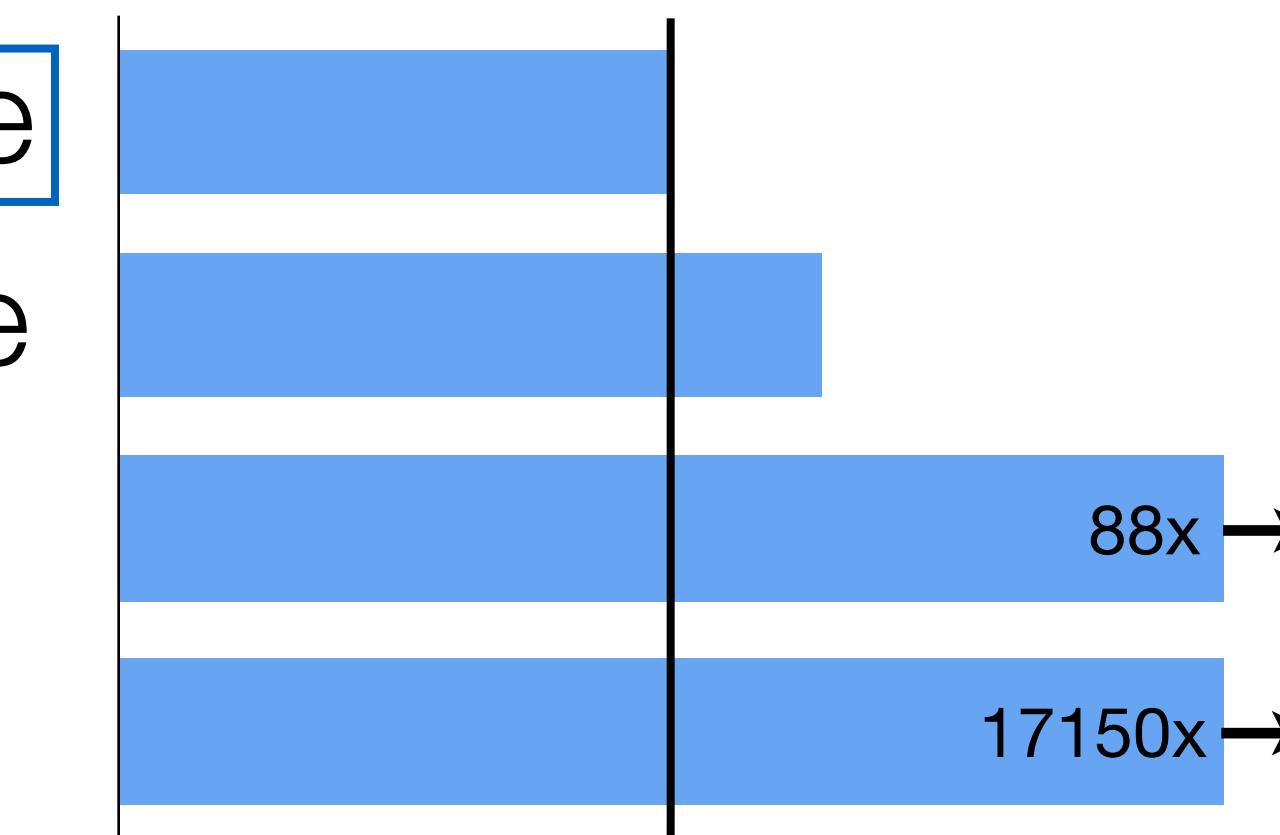
sparse - sparse

dense - sparse

sparse - dense

dense - dense

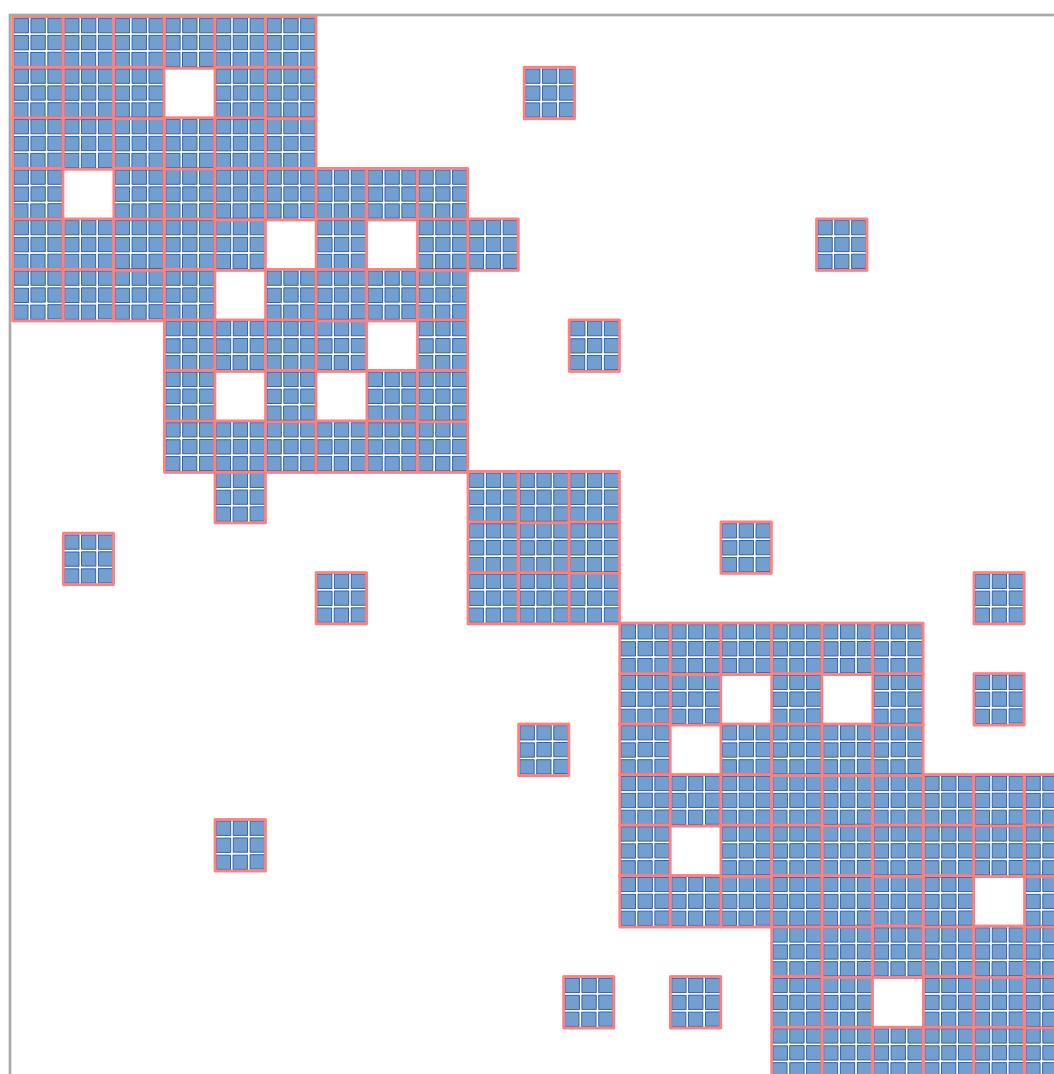
Normalized Time



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

Blocked matrices are 4-tensors

Blocked FEM Matrix



3x3 blocks

dense - sparse - dense - dense
sparse - sparse - dense - dense
dense - sparse
sparse - sparse

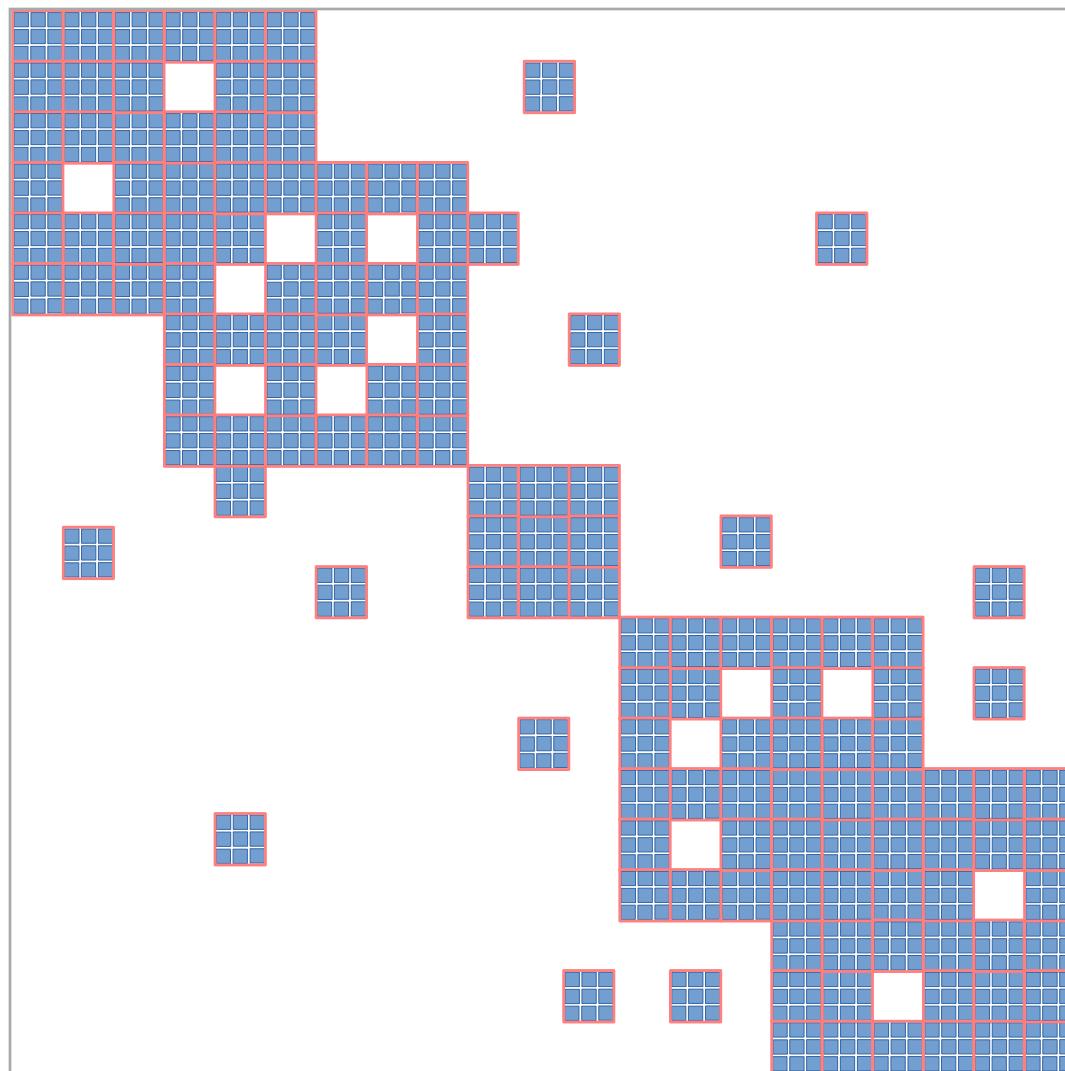
Normalized Time



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

Blocked matrices are 4-tensors

Blocked FEM Matrix



4-tensor

dense - sparse - dense - dense
sparse - sparse - dense - dense
dense - sparse
sparse - sparse

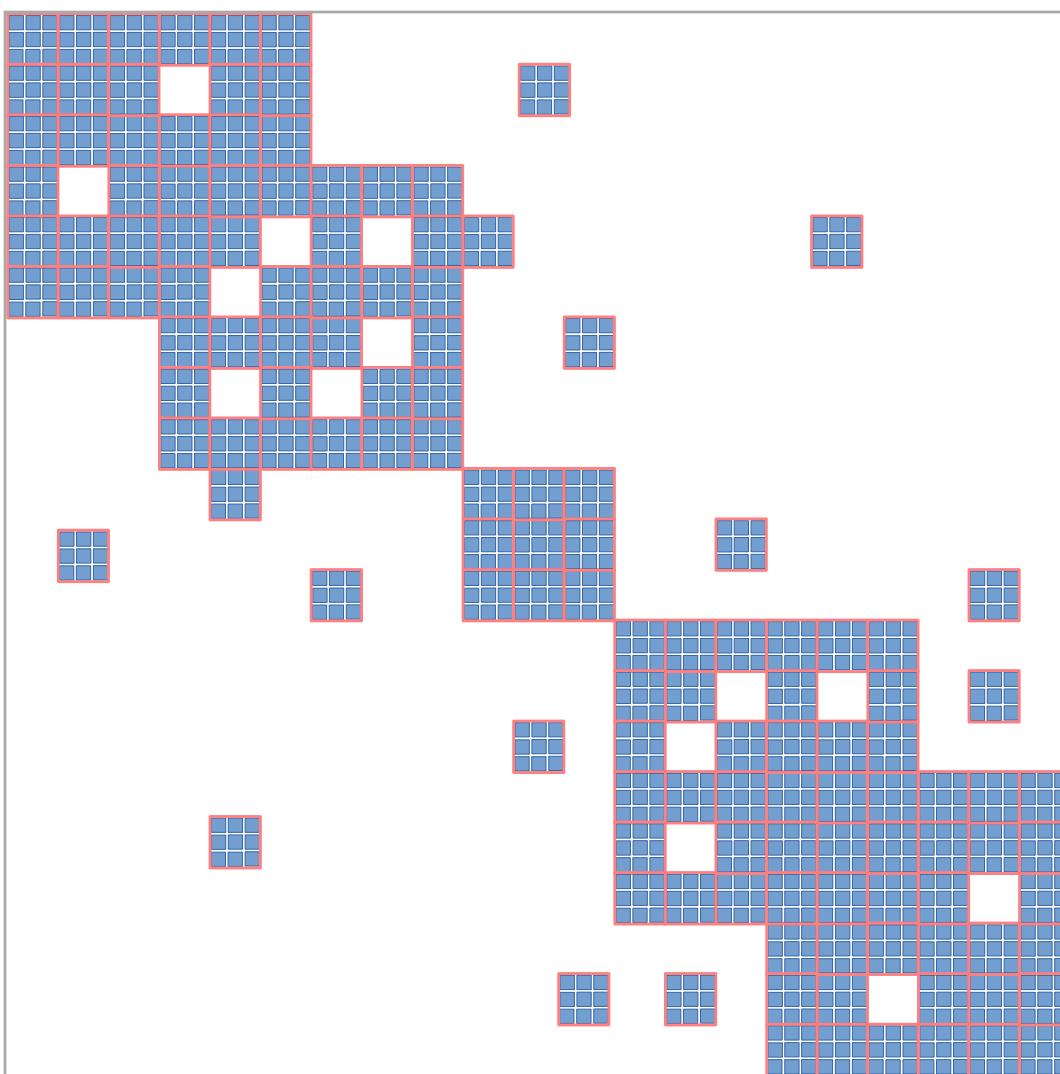
Normalized Time



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

Blocked matrices are 4-tensors

Blocked FEM Matrix



$$a_{ik} = \sum_j \sum_l B_{ijkl} c_{jl}$$

dense - sparse - dense - dense
sparse - sparse - dense - dense
dense - sparse
sparse - sparse

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

Normalized Time



Future Work (some of it)

Portability/Acceleration



Formats

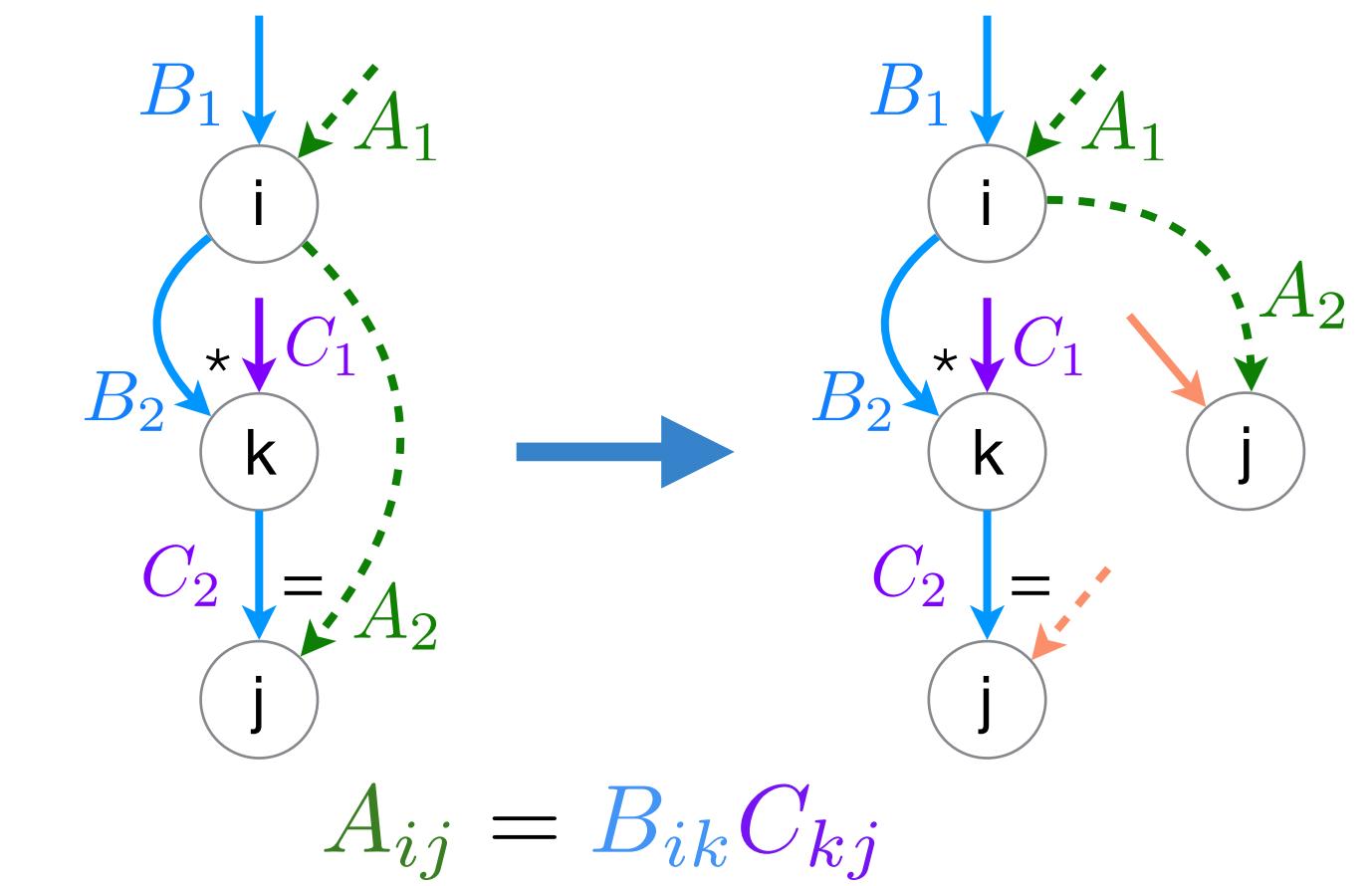
COO

ELL

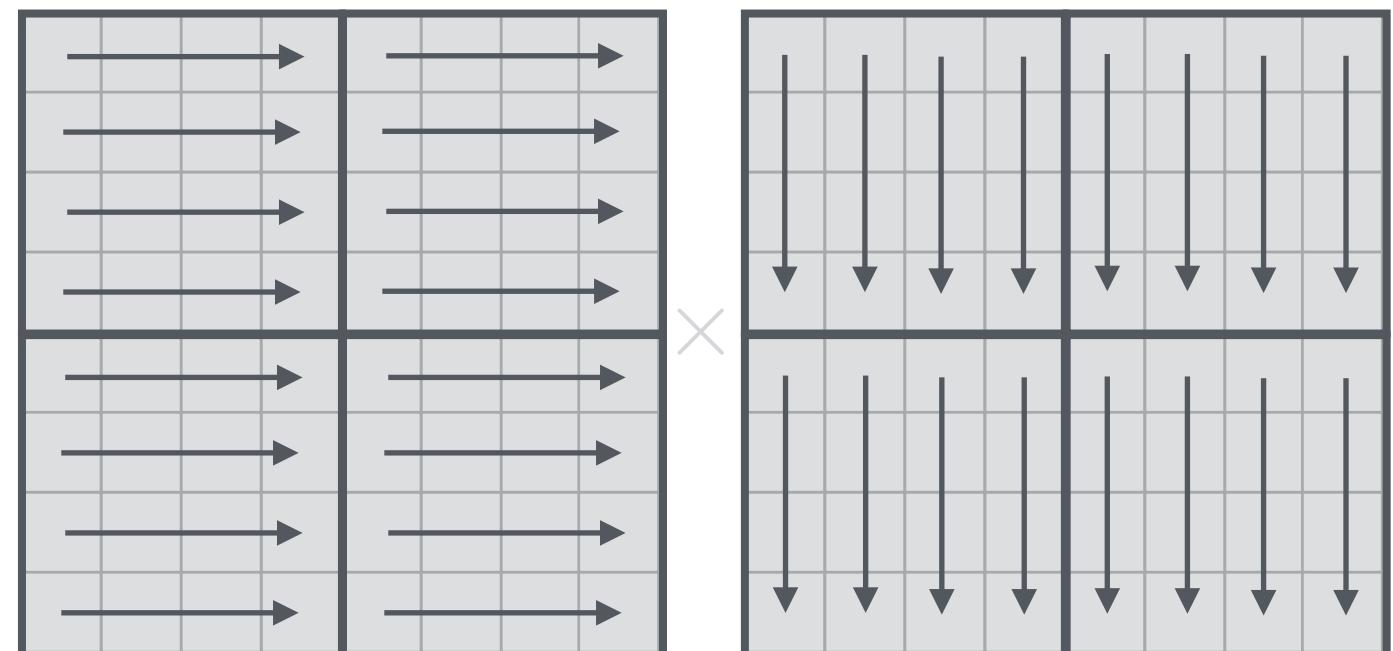
DIA

Hash maps

Workspaces



Scheduling Language



Semirings

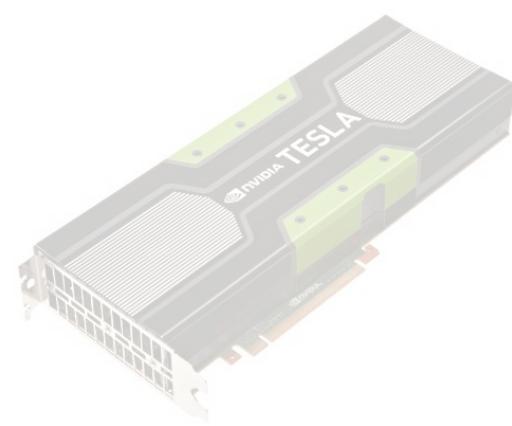
$(+, *)$	float
$(\max, +)$	double
(\min, \max)	complex
(\wedge, \vee)	int
$(f(), g())$	bool

Policy

Inspector-Executor	
ML	Sampling
Heuristics	
	Autotuning

Future Work (some of it)

Portability/Acceleration



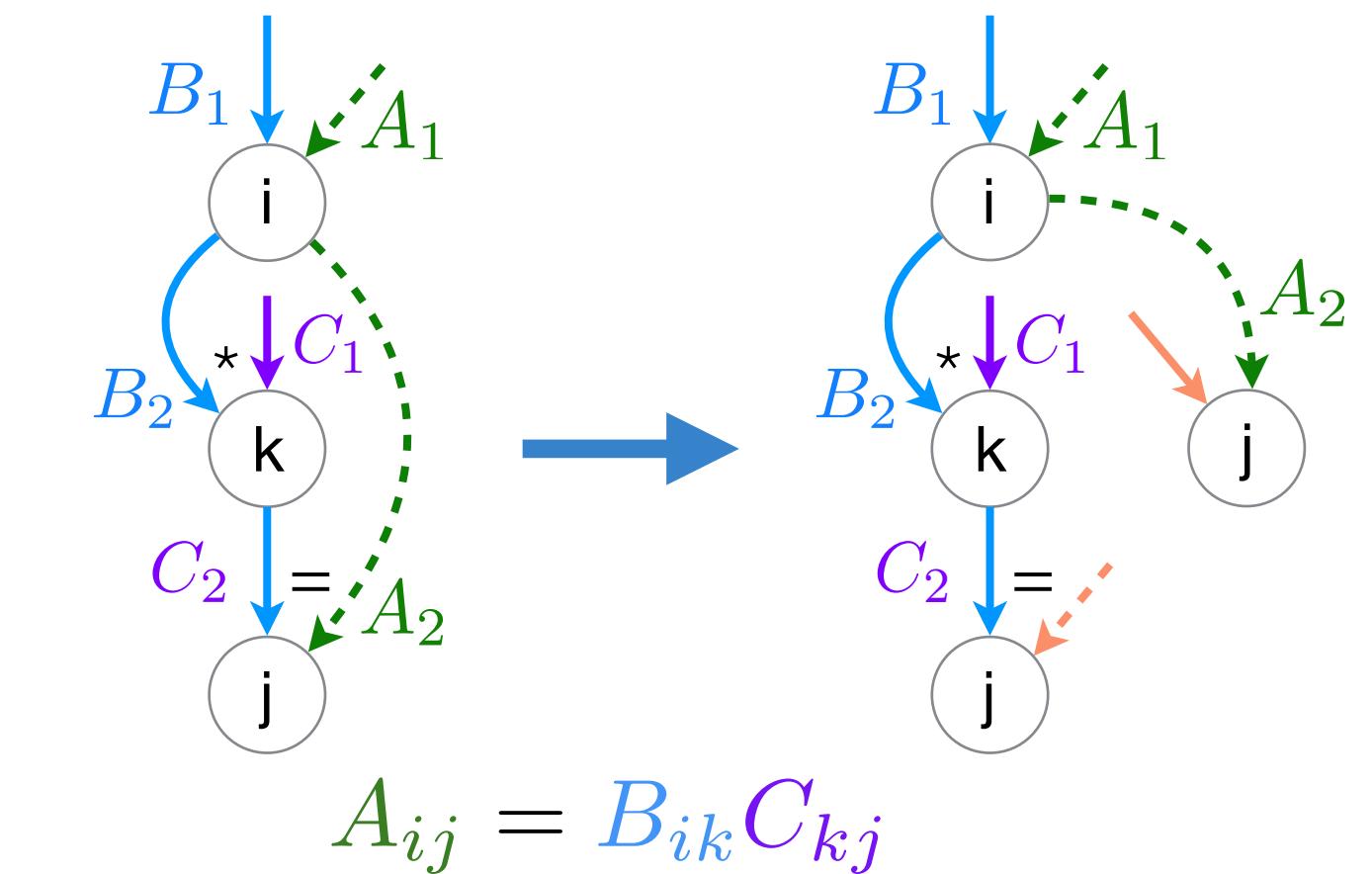
Formats

COO

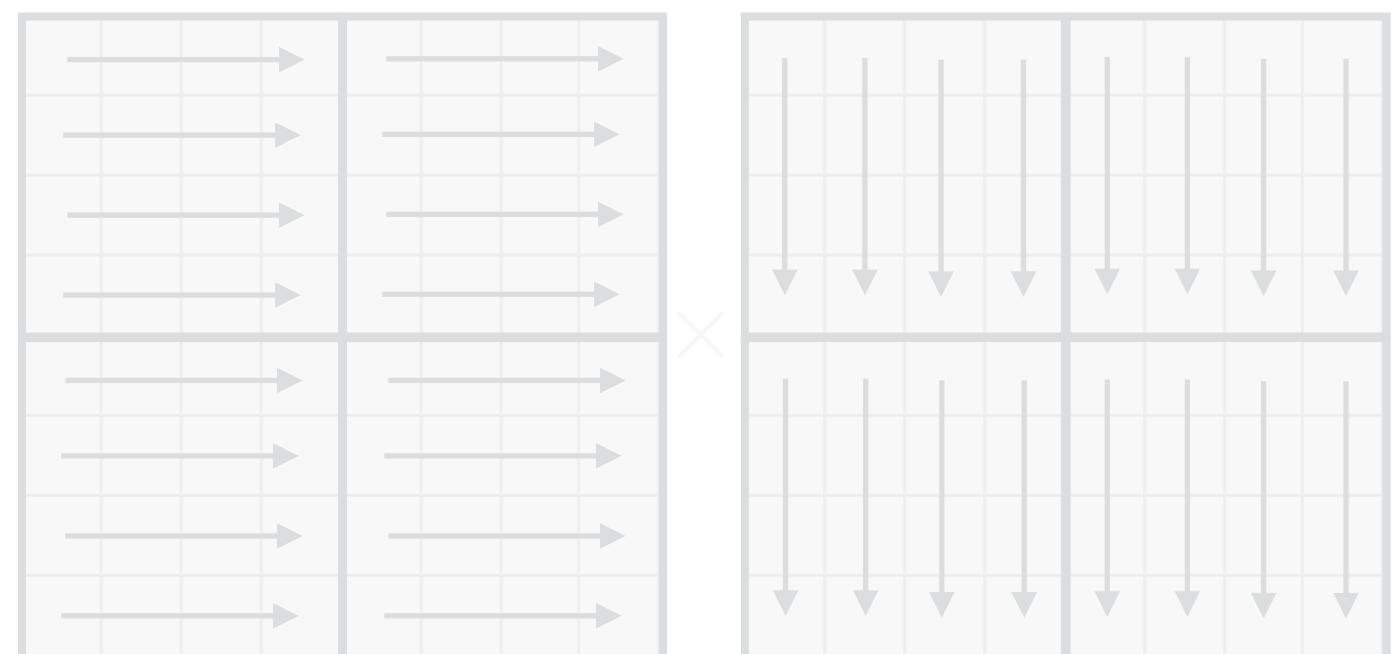
ELL

DIA
Hash maps

Workspaces



Scheduling Language



Semirings

$(+, *)$	float
$(\max, +)$	double
(\min, \max)	complex
(\wedge, \vee)	int
$(f(), g())$	bool

Policy

Inspector-Executor
ML
Heuristics
Sampling
Autotuning

Tensor Algebra Compiler Library and Tools

C++ Tensor Library

```
Format CSR({Dense,Sparse});  
Format CSF({Sparse,Sparse,Sparse});  
Format SVEC({Sparse});  
  
Tensor<double> A({1024,1024}, CSR);  
Tensor<double> B = read("B.tns", CSF);  
Tensor<double> c = read("c.tns", SVEC);  
  
Var i, j, k;  
A(i,j) = B(i,j,k) * c(k);  
  
A.evaluate(); // Compiles, assembles, and computes  
// the expression
```

tensor-compiler.org

github.com/tensor-compiler/taco



(Vanilla) MIT License

Online Code Generator

A screenshot of a web browser window titled "taco: the tensor algebra compiler". The URL bar shows "tensor-compiler.org". The page has a blue header with links for "Docs", "Publications", "Demo", and "GitHub". Below the header, a message states: "This is an alpha implementation of the tensor algebra compiler theory and contains known bugs, which are documented [here](#). If you find additional issues, please consider submitting a bug report." A main input field contains the tensor algebra expression "y(i) = A(i,j) * x(j)". To the right of the input field is a "GENERATE KERNEL" button with a dropdown menu icon. Below the input field, there are three rows for defining tensors: "y" (Dimension 1: Dense), "A" (Dimension 1: Dense, Dimension 2: Sparse), and "x" (Dimension 1: Dense). At the bottom, tabs for "COMPUTE LOOPS" (selected), "ASSEMBLY LOOPS", and "COMPLETE CODE" are visible, along with a note: "/* The generated compute loops will appear here */". A footer at the bottom of the page says: "The generated code is provided "as is" without warranty of any kind. To help us improve taco, we keep a record of all tensor algebra expressions submitted to the taco online server." A copyright notice at the bottom right reads: "Icons made by [Freepik](#) from [www.flaticon.com](#) is licensed by CC 3.0 BY".

tensor-compiler.org/codegen

Tensor Algebra Compiler Library and Tools

C++ Tensor Library

```
Format CSR({Dense,Sparse});  
Format CSF({Sparse,Sparse,Sparse});  
Format SVEC({Sparse});  
  
Tensor<double> A({1024,1024}, CSR);  
Tensor<double> B = read("B.tns", CSF);  
Tensor<double> c = read("c.tns", SVEC);  
  
Var i, j, k;  
A(i,j) = B(i,j,k) * c(k);  
  
A.evaluate(); // Compiles, assemble, and compute  
// the expression
```

tensor-compiler.org

github.com/tensor-compiler/taco



(Vanilla) MIT License

Online Code Generator

A screenshot of a web browser window titled "taco: the tensor algebra compiler". The URL bar shows "tensor-compiler.org". The page has a blue header with links for "Docs", "Publications", "Demo", and "GitHub". Below the header, a message states: "This is an alpha implementation of the tensor algebra compiler theory and contains known bugs, which are documented [here](#). If you find additional issues, please consider submitting a bug report." A text input field contains the expression "y(i) = A(i,j) * x(j)". To the right of the input field is a "GENERATE KERNEL" button with a dropdown menu icon. Below the input field, there are three rows for defining tensors: "y" (Dimension 1: Dense), "A" (Dimension 1: Dense, Dimension 2: Sparse), and "x" (Dimension 1: Dense). At the bottom, there are tabs for "COMPUTE LOOPS" (which is selected), "ASSEMBLY LOOPS", and "COMPLETE CODE". A note below the tabs says: "/* The generated compute loops will appear here */". A footer at the bottom of the page states: "The generated code is provided "as is" without warranty of any kind. To help us improve taco, we keep a record of all tensor algebra expressions submitted to the taco online server." A copyright notice at the bottom right says: "Icons made by [Freepik](#) from [www.flaticon.com](#) is licensed by CC 3.0 BY".

tensor-compiler.org/codegen

Tensor Algebra Compiler Library and Tools

C++ Tensor Library

```
Format CSR({Dense,Sparse});  
Format CSF({Sparse,Sparse,Sparse});  
Format SVEC({Sparse});  
  
Tensor<double> A({1024,1024}, CSR);  
Tensor<double> B = read("B.tns", CSF);  
Tensor<double> c = read("c.tns", SVEC);  
  
Var i, j, k;  
A(i,j) = B(i,j,k) * c(k);  
  
A.evaluate(); // Compiles, assemble, and compute  
// the expression
```

tensor-compiler.org

github.com/tensor-compiler/taco



(Vanilla) MIT License

Online Code Generator

The screenshot shows the taco online code generator interface. At the top, it says "taco: the tensor algebra compiler" and has links for "Docs", "Publications", "Demo", and "GitHub". Below that, a message states: "This is an alpha implementation of the tensor algebra compiler theory and contains known bugs, which are documented [here](#). If you find additional issues, please consider submitting a bug report." A text input field contains the expression "y(i) = A(i,j) * x(j)". To the right of the input field is a "GENERATE KERNEL" button with a dropdown menu icon. Below the input field, there are three dropdown menus for tensors y, A, and x, each labeled "Tensor" and "Format (reorder dimensions by dragging the drop-down menus)". The dropdown for y is set to "Dimension 1 Dense". The dropdown for A is set to "Dimension 1 Dense" and "Dimension 2 Sparse". The dropdown for x is set to "Dimension 1 Dense". At the bottom, there are tabs for "COMPUTE LOOPS" (which is selected), "ASSEMBLY LOOPS", and "COMPLETE CODE". A note below the tabs says: "/* The generated compute loops will appear here */". Another note at the bottom says: "The generated code is provided "as is" without warranty of any kind. To help us improve taco, we keep a record of all tensor algebra expressions submitted to the taco online server." A footer at the very bottom says: "Icons made by [Freepik](#) from [www.flaticon.com](#) is licensed by CC 3.0 BY".

tensor-compiler.org/codegen

Tensor Algebra Compiler Library and Tools

C++ Tensor Library

```
Format CSR({Dense,Sparse});  
Format CSF({Sparse,Sparse,Sparse});  
Format SVEC({Sparse});  
  
Tensor<double> A({1024,1024}, CSR);  
Tensor<double> B = read("B.tns", CSF);  
Tensor<double> c = read("c.tns", SVEC);  
  
Var i, j, k;  
A(i,j) = B(i,j,k) * c(k);  
  
A.evaluate(); // Compiles, assembles, and computes  
// the expression
```

tensor-compiler.org

github.com/tensor-compiler/taco



(Vanilla) MIT License

Online Code Generator

The screenshot shows a web browser window for "tensor-compiler.org". The title bar says "taco: the tensor algebra compiler". The main content area has a message: "This is an alpha implementation of the tensor algebra compiler theory and contains known bugs, which are documented [here](#). If you find additional issues, please consider submitting a bug report." Below this is a text input field containing "y(i) = A(i,j) * x(j)". To the right of the input field is a "GENERATE KERNEL" button with a dropdown menu icon. Below the input field, there are three sections for tensors "y", "A", and "x", each with a "Tensor" dropdown and a "Format" dropdown. The "y" section has "Dimension 1 Dense". The "A" section has "Dimension 1 Dense" and "Dimension 2 Sparse". The "x" section has "Dimension 1 Dense". At the bottom, there are tabs for "COMPUTE LOOPS" (which is selected), "ASSEMBLY LOOPS", and "COMPLETE CODE". A note below the tabs says "/* The generated compute loops will appear here */". A footer at the bottom of the page states: "The generated code is provided "as is" without warranty of any kind. To help us improve taco, we keep a record of all tensor algebra expressions submitted to the taco online server." A copyright notice at the bottom right says "Icons made by Freepik from www.flaticon.com is licensed by CC 3.0 BY".

tensor-compiler.org/codegen

Tensor Algebra Compiler Library and Tools

C++ Tensor Library

```
Format CSR({Dense,Sparse});  
Format CSF({Sparse,Sparse,Sparse});  
Format SVEC({Sparse});  
  
Tensor<double> A({1024,1024}, CSR);  
Tensor<double> B = read("B.tns", CSF);  
Tensor<double> c = read("c.tns", SVEC);  
  
Var i, j, k;  
A(i,j) = B(i,j,k) * c(k);  
  
A.evaluate(); // Compiles, assembles, and computes  
// the expression
```

tensor-compiler.org

github.com/tensor-compiler/taco



(Vanilla) MIT License

Online Code Generator

A screenshot of a web browser window titled "taco: the tensor algebra compiler". The URL bar shows "tensor-compiler.org". The page has a blue header with links for "Docs", "Publications", "Demo", and "GitHub". Below the header, a message states: "This is an alpha implementation of the tensor algebra compiler theory and contains known bugs, which are documented [here](#). If you find additional issues, please consider submitting a bug report." A text input field contains the expression "y(i) = A(i,j) * x(j)". To the right of the input field is a "GENERATE KERNEL" button with a dropdown arrow. Below the input field, there are three rows for defining tensors: "y" (Dimension 1: Dense), "A" (Dimension 1: Dense, Dimension 2: Sparse), and "x" (Dimension 1: Dense). At the bottom of the page, tabs for "COMPUTE LOOPS", "ASSEMBLY LOOPS", and "COMPLETE CODE" are visible, along with a note: "/* The generated compute loops will appear here */". A footer at the bottom right says: "The generated code is provided "as is" without warranty of any kind. To help us improve taco, we keep a record of all tensor algebra expressions submitted to the taco online server." Another footer at the very bottom right credits "Icons made by Freepik from www.flaticon.com is licensed by CC 3.0 BY".

tensor-compiler.org/codegen

Tensor Algebra Compiler Library and Tools

C++ Tensor Library

```
Format CSR({Dense,Sparse});  
Format CSF({Sparse,Sparse,Sparse});  
Format SVEC({Sparse});  
  
Tensor<double> A({1024,1024}, CSR);  
Tensor<double> B = read("B.tns", CSF);  
Tensor<double> c = read("c.tns", SVEC);  
  
Var i, j, k;  
A(i,j) = B(i,j,k) * c(k);  
  
A.evaluate(); // Compiles, assembles, and computes  
// the expression
```

tensor-compiler.org

github.com/tensor-compiler/taco



(Vanilla) MIT License

Online Code Generator

A screenshot of a web browser window titled "taco: the tensor algebra compiler". The URL bar shows "tensor-compiler.org". The page has a blue header with links for "Docs", "Publications", "Demo", and "GitHub". Below the header, a message states: "This is an alpha implementation of the tensor algebra compiler theory and contains known bugs, which are documented [here](#). If you find additional issues, please consider submitting a bug report." A main input field contains the tensor algebra expression "y(i) = A(i,j) * x(j)". To the right of the input field is a "GENERATE KERNEL" button with a dropdown menu icon. Below the input field, there are three rows for defining tensors: "y" (Dimension 1: Dense), "A" (Dimension 1: Dense, Dimension 2: Sparse), and "x" (Dimension 1: Dense). At the bottom, tabs for "COMPUTE LOOPS" (selected), "ASSEMBLY LOOPS", and "COMPLETE CODE" are visible, along with a note: "/* The generated compute loops will appear here */". A footer at the bottom of the page says: "The generated code is provided "as is" without warranty of any kind. To help us improve taco, we keep a record of all tensor algebra expressions submitted to the taco online server." A copyright notice at the bottom right reads: "Icons made by [Freepik](#) from [www.flaticon.com](#) is licensed by CC 3.0 BY".

tensor-compiler.org/codegen

Tensor Algebra Compiler Library and Tools

C++ Tensor Library

```
Format CSR({Dense,Sparse});  
Format CSF({Sparse,Sparse,Sparse});  
Format SVEC({Sparse});  
  
Tensor<double> A({1024,1024}, CSR);  
Tensor<double> B = read("B.tns", CSF);  
Tensor<double> c = read("c.tns", SVEC);  
  
Var i, j, k;  
A(i,j) = B(i,j,k) * c(k);  
  
A.evaluate(); // Compiles, assembles, and computes  
// the expression
```

tensor-compiler.org

github.com/tensor-compiler/taco



(Vanilla) MIT License

Online Code Generator

A screenshot of a web browser window titled "taco: the tensor algebra compiler". The URL bar shows "tensor-compiler.org". The page has a blue header with links for "Docs", "Publications", "Demo", and "GitHub". Below the header, a message states: "This is an alpha implementation of the tensor algebra compiler theory and contains known bugs, which are documented [here](#). If you find additional issues, please consider submitting a bug report." A main input field contains the tensor algebra expression "y(i) = A(i,j) * x(j)". To the right of the input field is a "GENERATE KERNEL" button with a dropdown menu icon. Below the input field, there are three rows for defining tensors: "y" (Dimension 1: Dense), "A" (Dimension 1: Dense, Dimension 2: Sparse), and "x" (Dimension 1: Dense). At the bottom, tabs for "COMPUTE LOOPS" (selected), "ASSEMBLY LOOPS", and "COMPLETE CODE" are visible. A note below the tabs says: "/* The generated compute loops will appear here */". A footer at the bottom of the page states: "The generated code is provided "as is" without warranty of any kind. To help us improve taco, we keep a record of all tensor algebra expressions submitted to the taco online server." A copyright notice at the bottom right says: "Icons made by [Freepik](#) from [www.flaticon.com](#) is licensed by CC 3.0 BY".

tensor-compiler.org/codegen