

The Tensor Algebra Compiler

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

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



Tensors are everywhere

Data Analytics

NETFLIX

Movie ratings

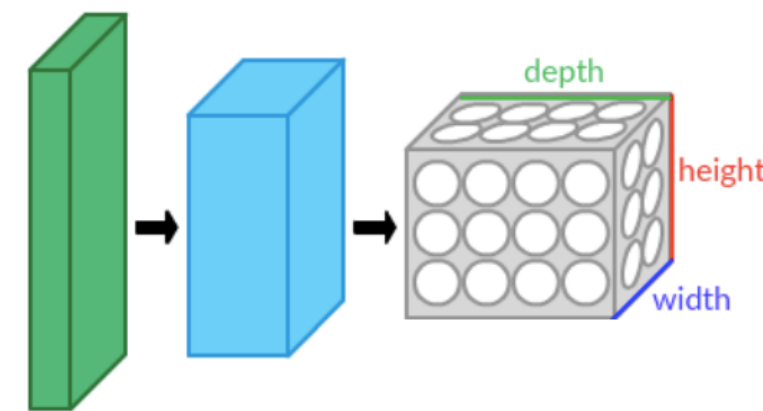
amazon

Product Reviews

facebook

Social interactions

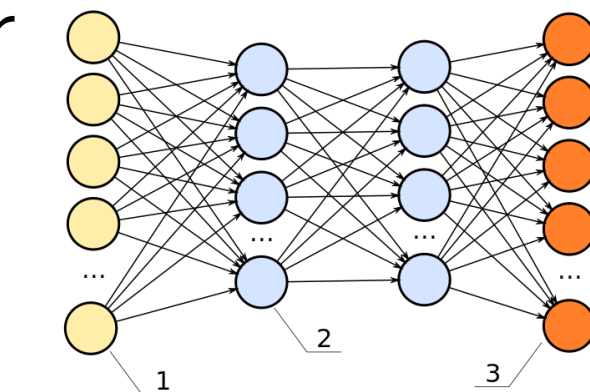
Machine Learning



Convolutional Layer

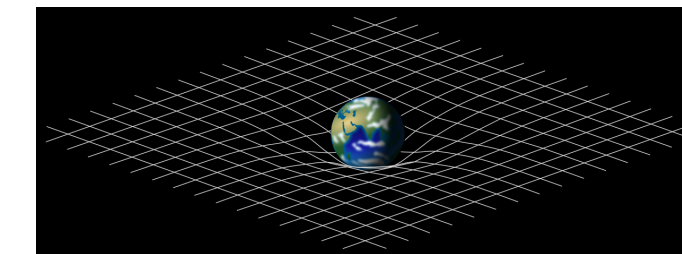
1 1 5 4 3
7 5 3 5 3
5 5 9 0 6
3 5 2 0 0

Images

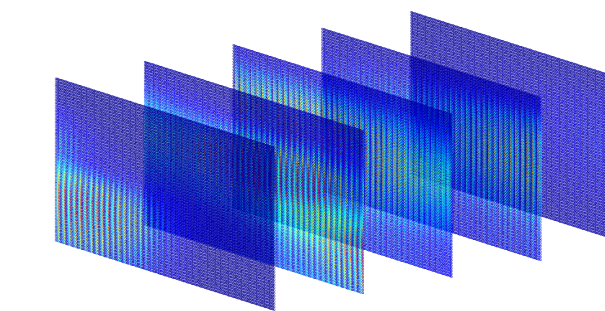


Connected Layer

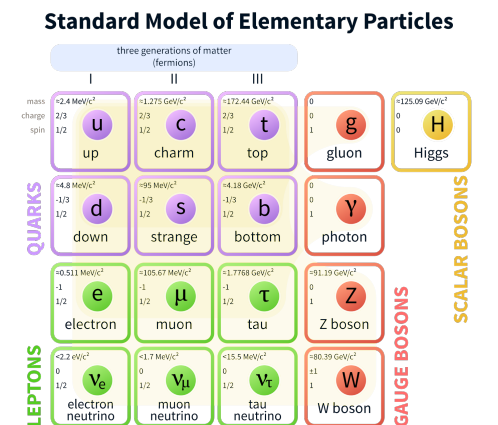
Science and Engineering



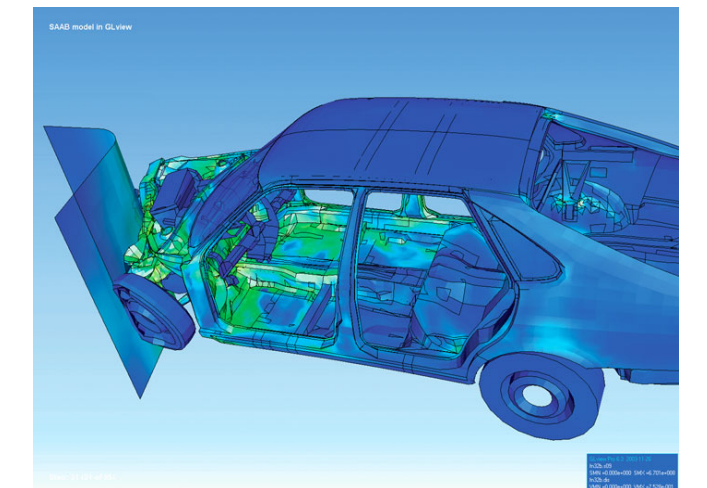
General Relativity



Fluorescence spectroscopy



QCD



Finite Element Method

Tensors are everywhere

Data Analytics

NETFLIX

Movie ratings

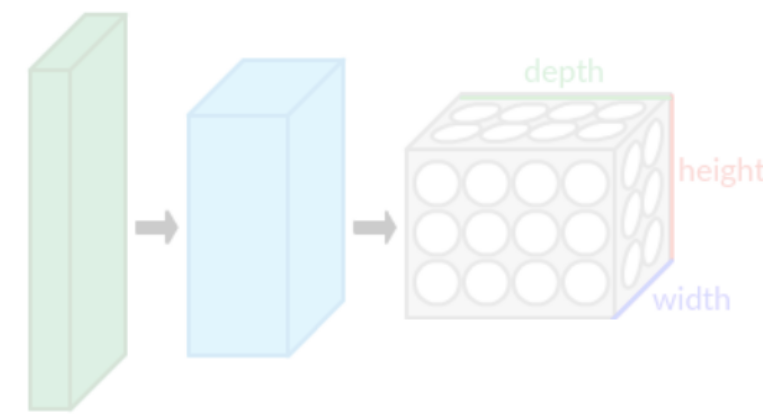
amazon

Product Reviews

facebook

Social interactions

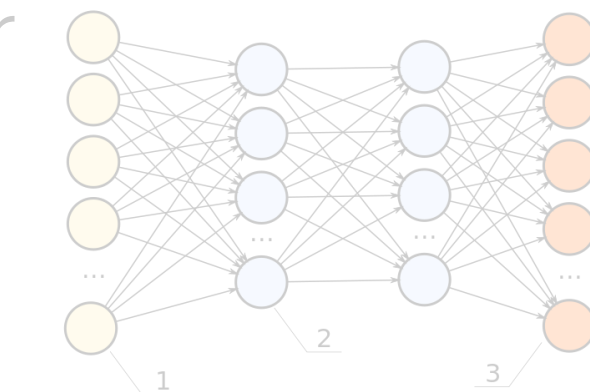
Machine Learning



Convolutional Layer

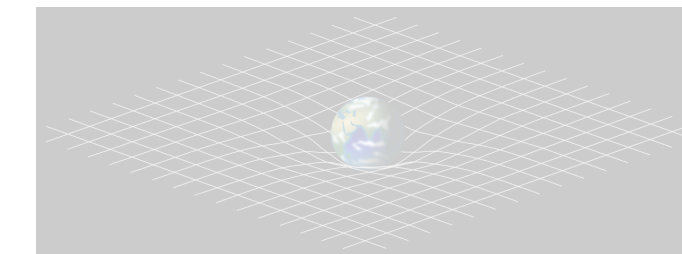
1 1 5 4 3
7 5 3 5 3
5 5 9 0 6
3 5 2 0 0

Images

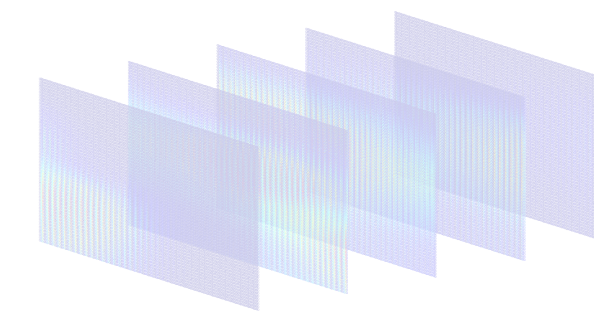


Connected Layer

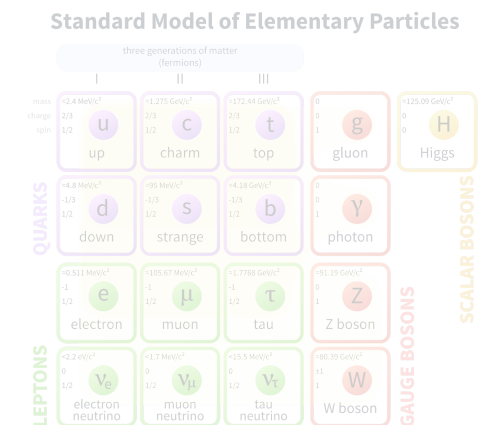
Science and Engineering



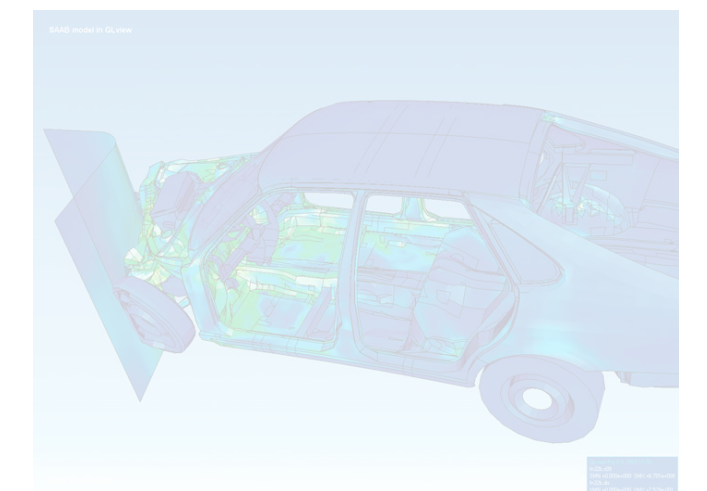
General Relativity



Fluorescence spectroscopy



QCD



Finite Element Method

Tensors are everywhere

Data Analytics

NETFLIX

Movie ratings

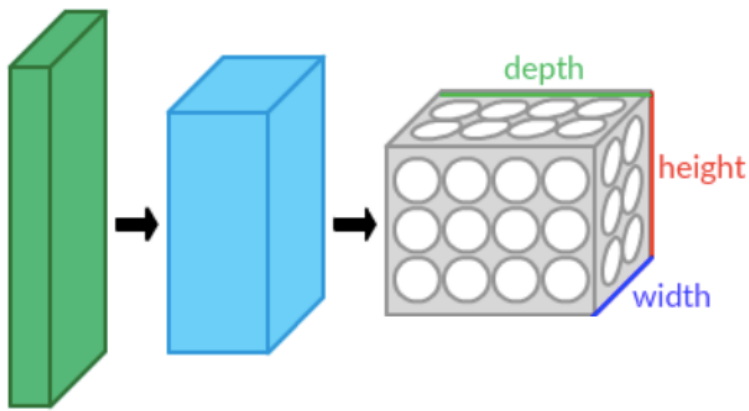
amazon

Product Reviews

facebook

Social interactions

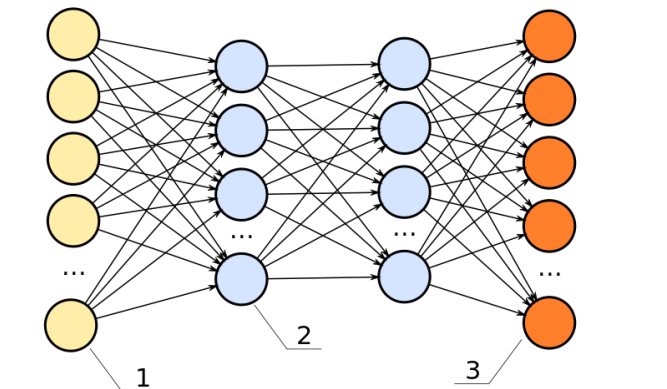
Machine Learning



Convolutional Layer

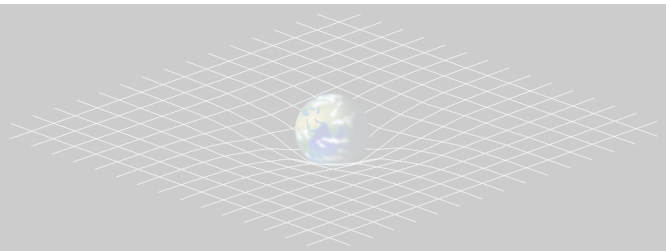
1 1 5 4 3
7 5 3 5 3
5 5 9 0 6
3 5 2 0 0

Images

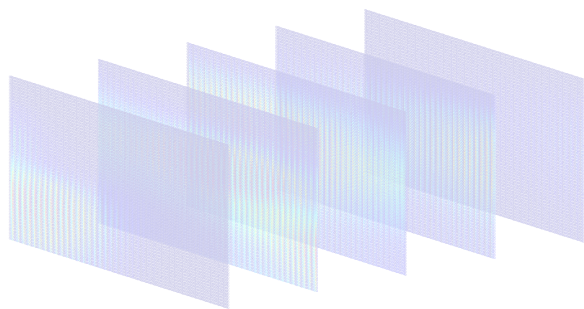


Connected Layer

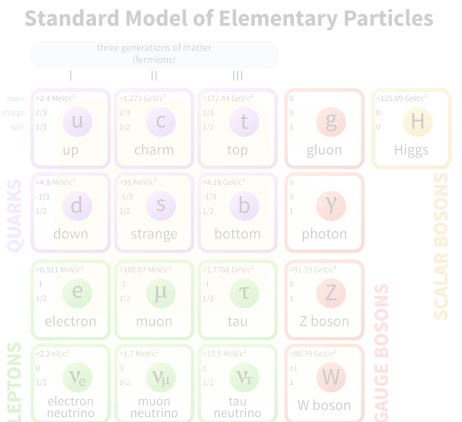
Science and Engineering



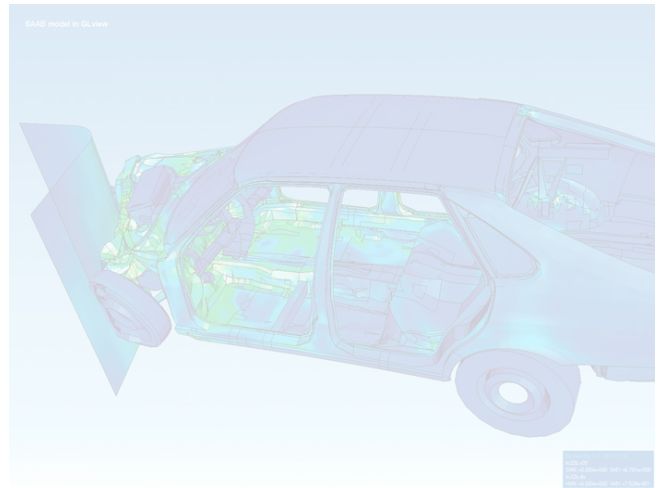
General Relativity



Fluorescence spectroscopy



QCD



Finite Element Method

Tensors are everywhere

Data Analytics

NETFLIX

Movie ratings

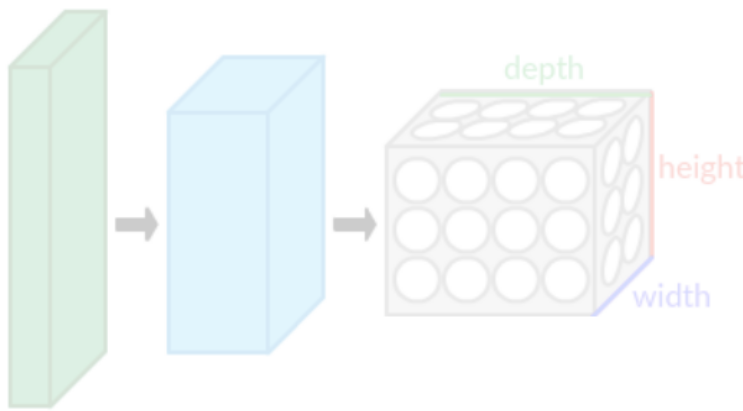
amazon

Product Reviews

facebook

Social interactions

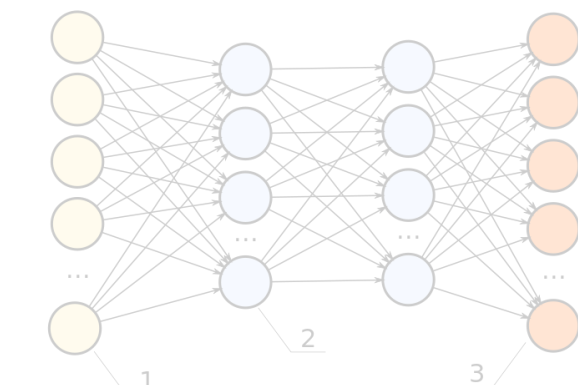
Machine Learning



Convolutional Layer

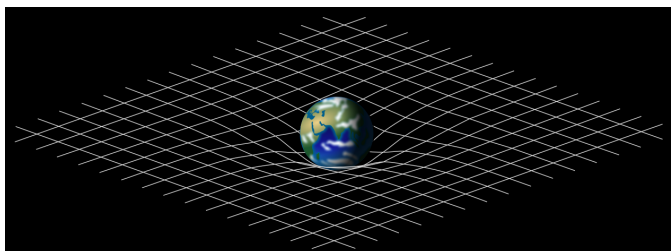


Images

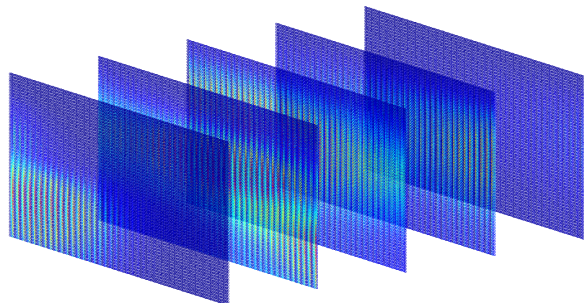


Connected Layer

Science and Engineering



General Relativity

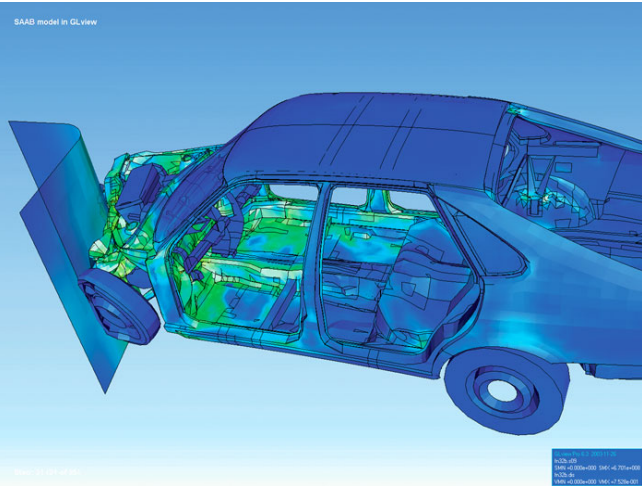


Fluorescence spectroscopy

Standard Model of Elementary Particles

three generations of matter (fermions)				
I	II	III		
u	c	t	s	H
up	charm	top	gluon	Higgs
d	s	b	photon	
down	strange	bottom		
e	μ	τ	Z boson	
electron	muon	tau		
ν_e	ν_μ	ν_τ	W boson	
electron neutrino	muon neutrino	tau neutrino		

QCD



Finite Element Method

Tensors are everywhere

Data Analytics

NETFLIX

Movie ratings

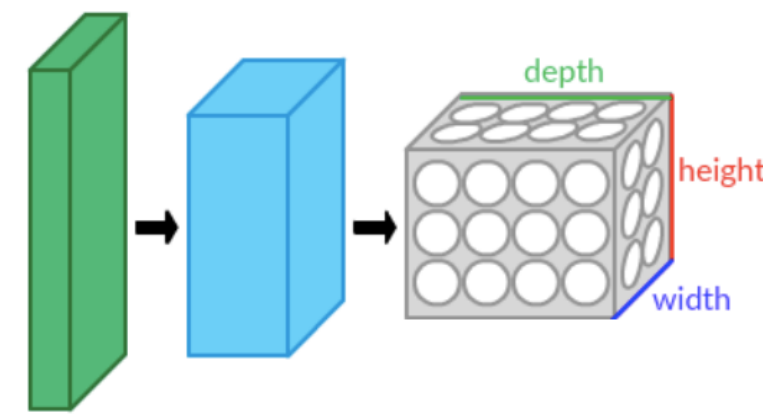
amazon

Product Reviews

facebook

Social interactions

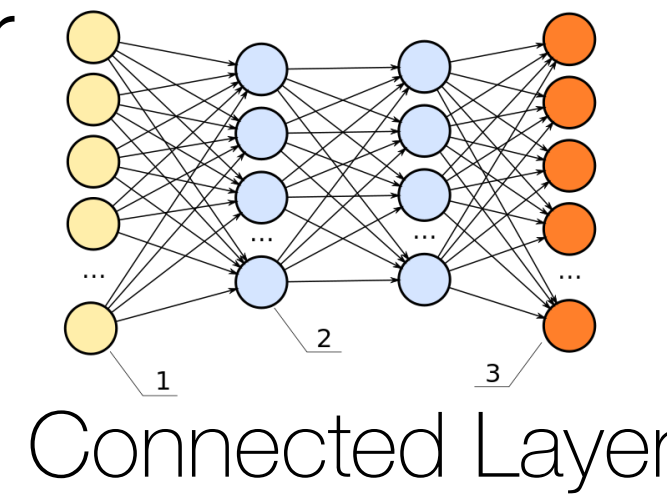
Machine Learning



Convolutional Layer

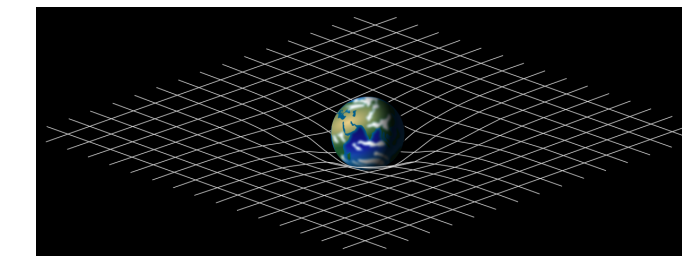
1 1 5 4 3
7 5 3 5 3
5 5 9 0 6
3 5 2 0 0

Images

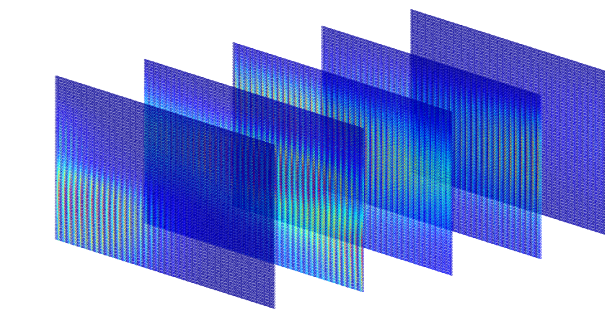


Connected Layer

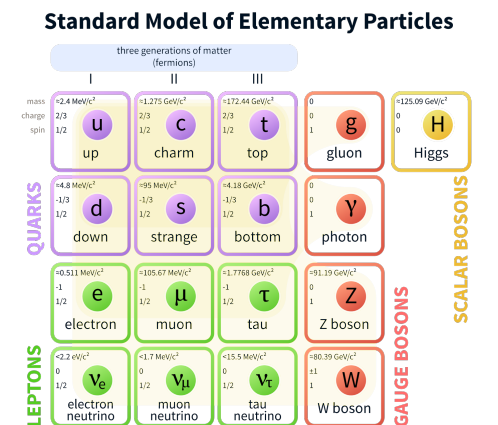
Science and Engineering



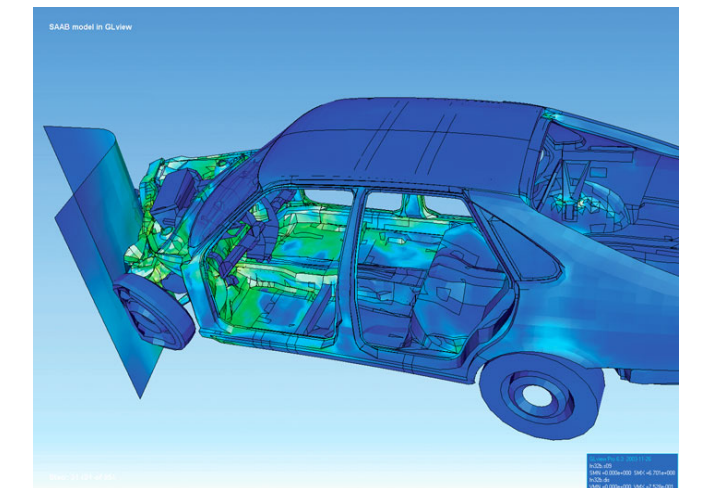
General Relativity



Fluorescence spectroscopy



QCD



Finite Element Method

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

Tensors are everywhere

Data Analytics

NETFLIX

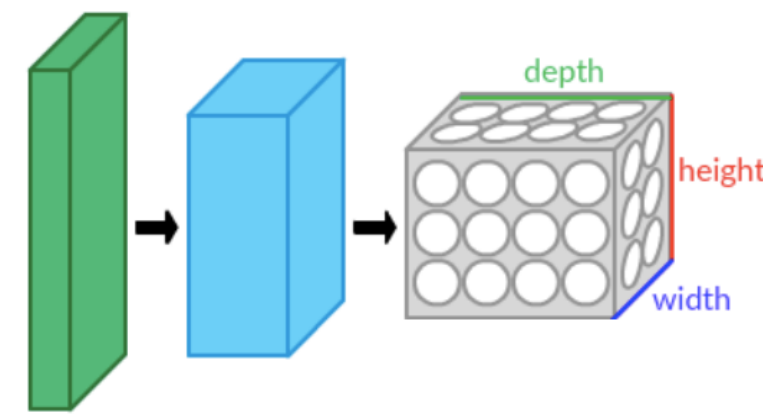
Movie ratings

facebook

Social interactions

amazon
Product Reviews

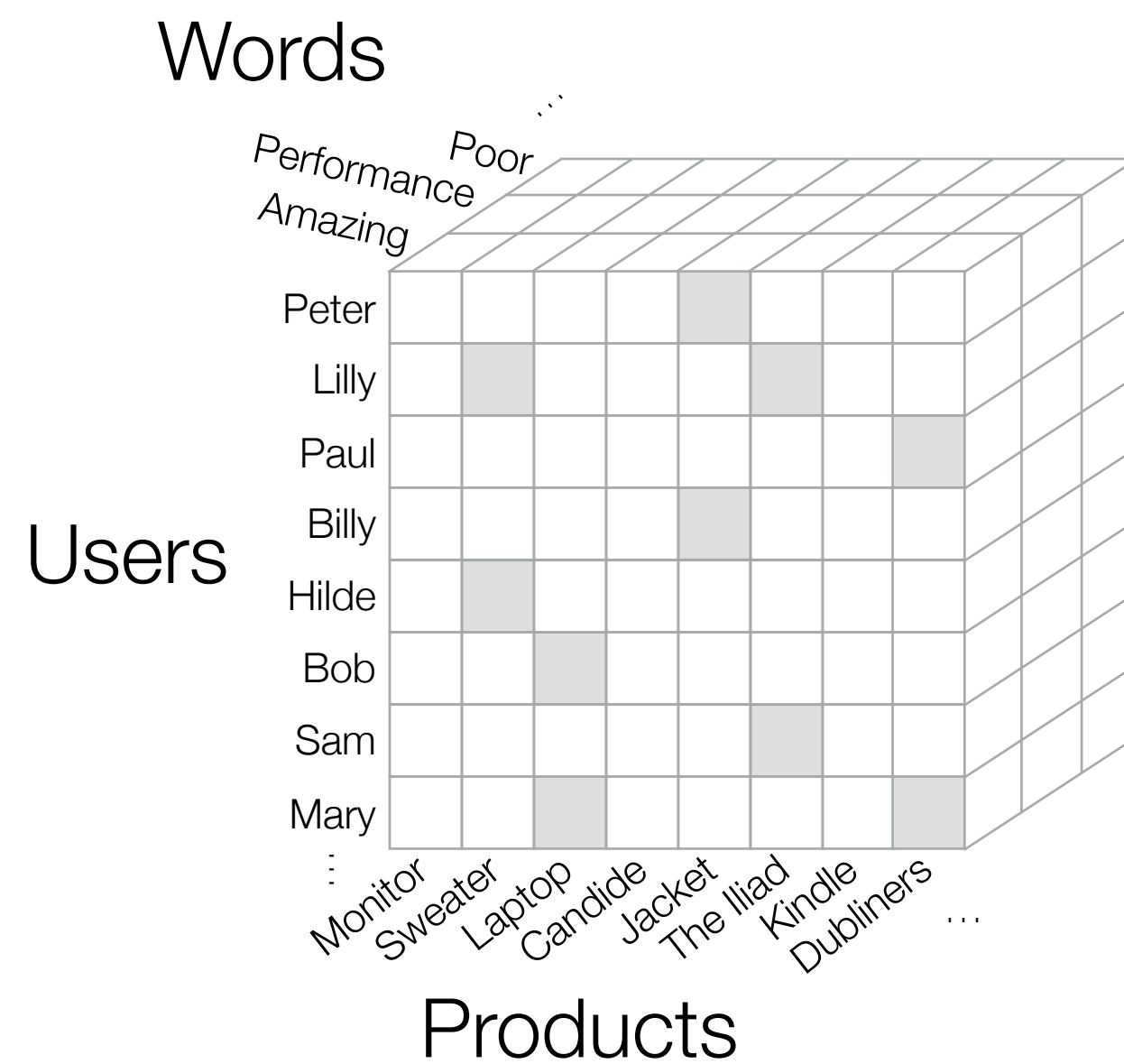
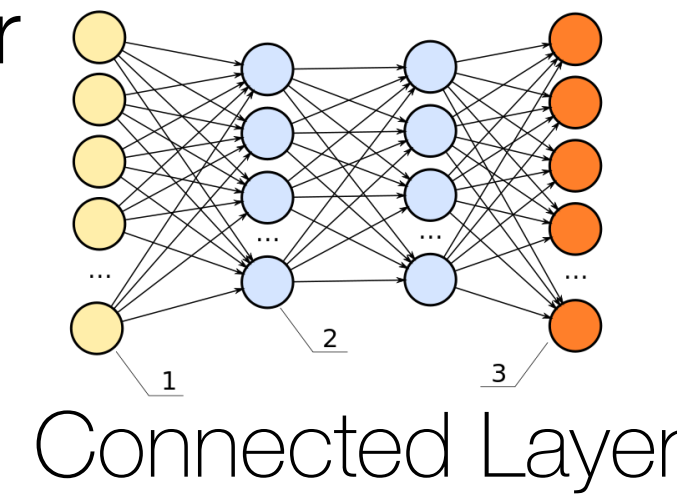
Machine Learning



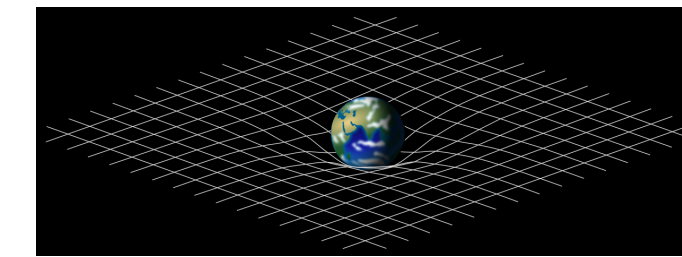
Convolutional Layer

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

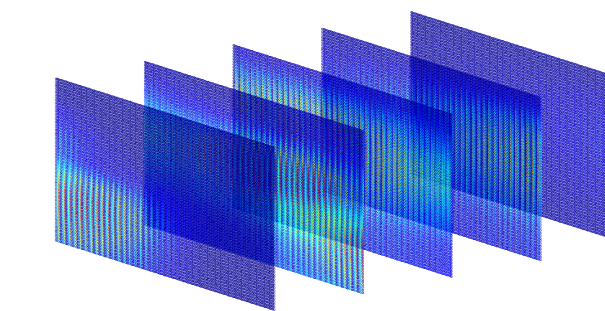
Images



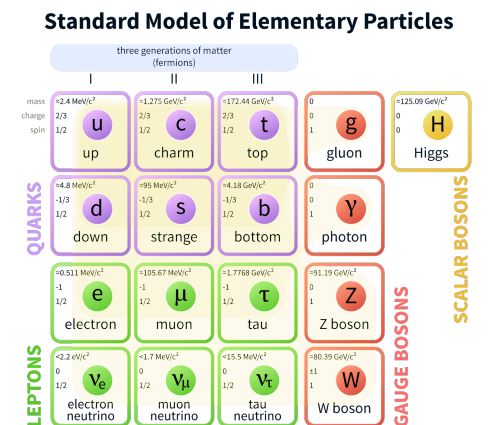
Science and Engineering



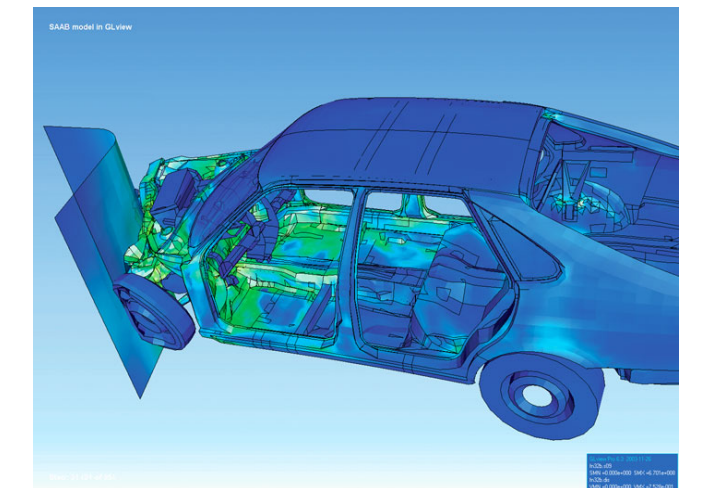
General Relativity



Fluorescence spectroscopy



QCD



Finite Element Method

Tensors are everywhere

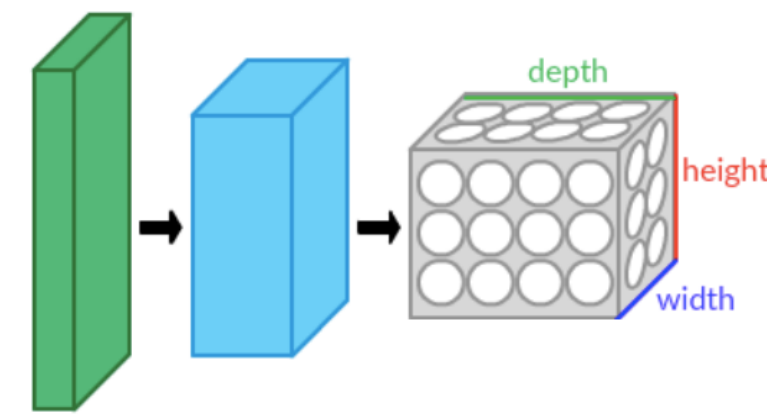
Data Analytics

NETFLIX
Movie ratings

facebook
Social interactions

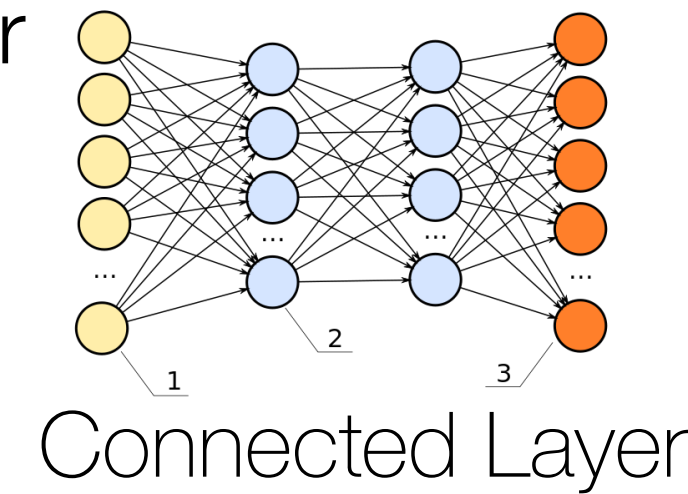
amazon
Product Reviews

Machine Learning

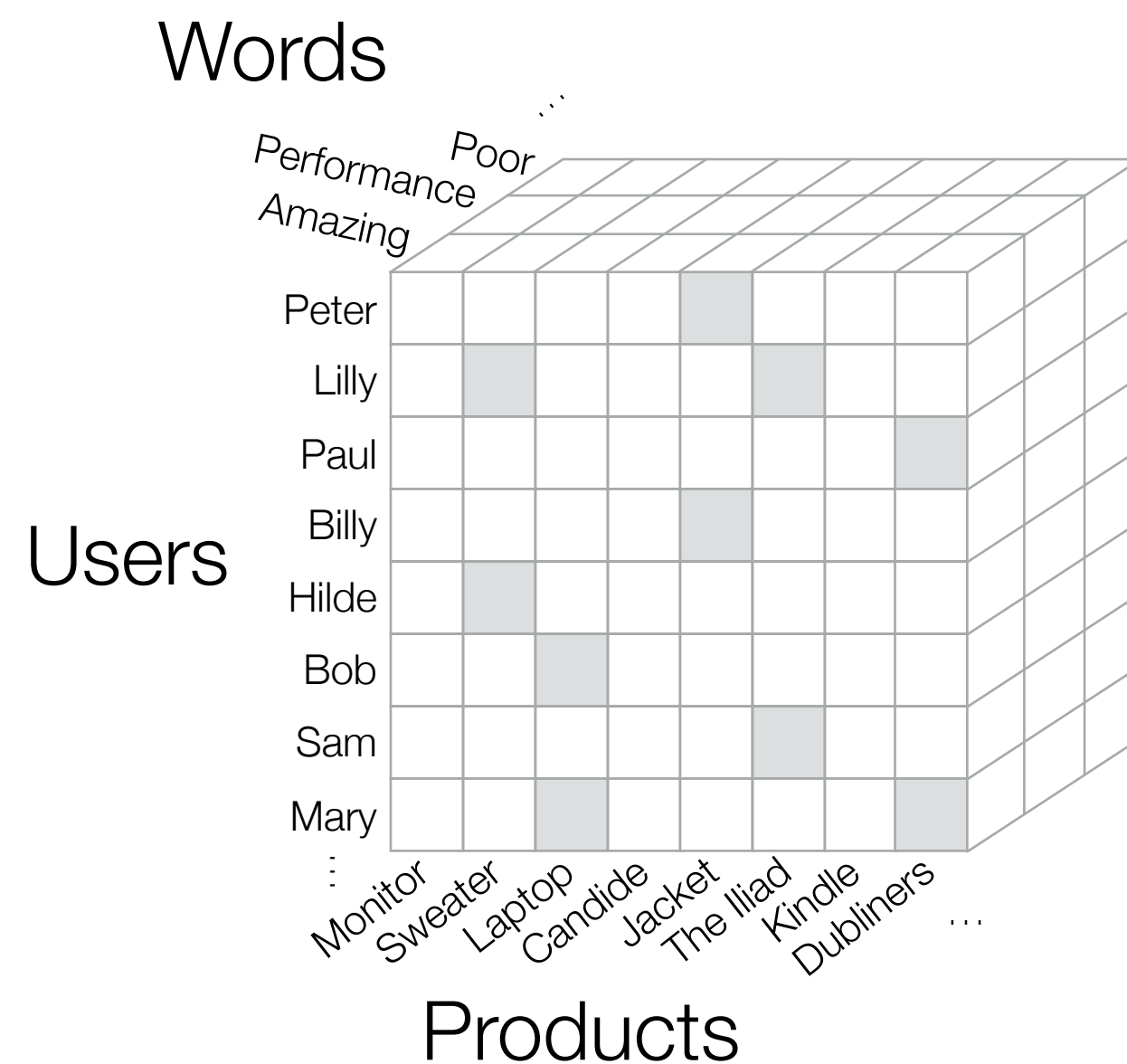


Convolutional Layer

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

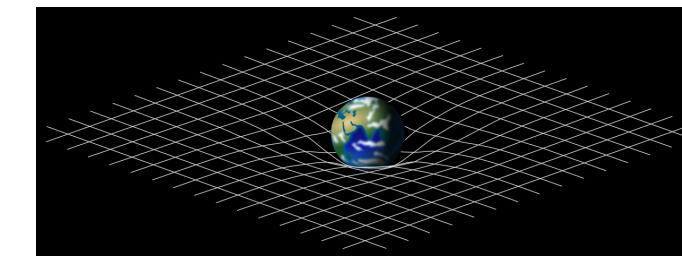


Connected Layer

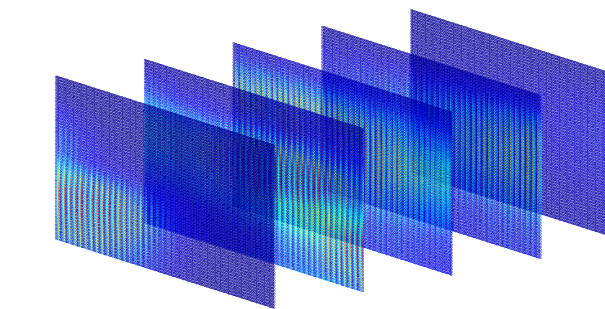


Extremely sparse
Dense storage: 107 Exabytes
Sparse storage: 13 Gigabytes

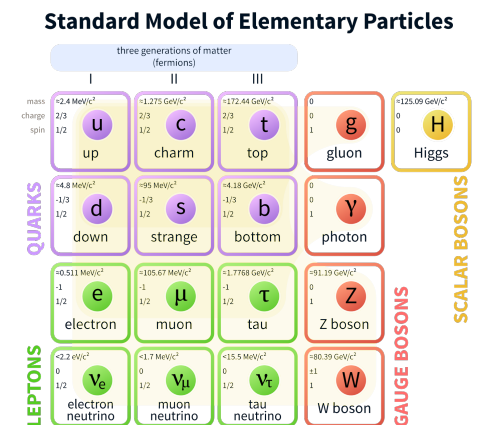
Science and Engineering



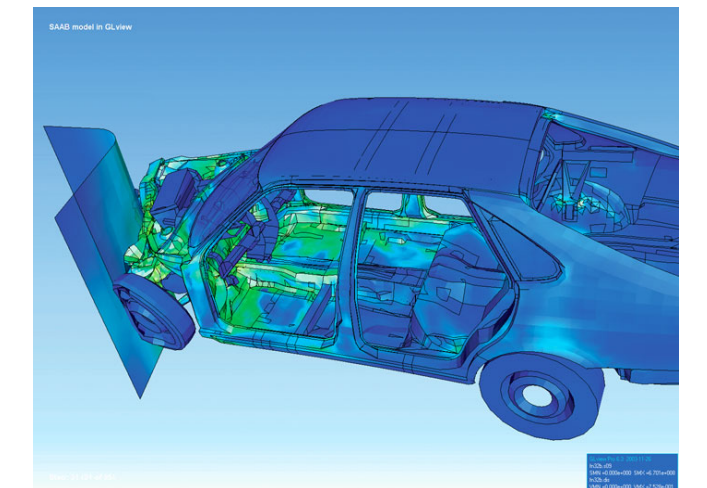
General Relativity



Fluorescence spectroscopy



QCD



Finite Element Method

We need a Tensor Algebra Compiler

We need a Tensor Algebra Compiler

$$a = Bc$$

We need a Tensor Algebra Compiler

$$a = Bc$$

Eigen

We need a Tensor Algebra Compiler

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

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

We need a Tensor Algebra Compiler

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

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

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

We need a Tensor Algebra Compiler

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

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

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

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

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

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

We need a Tensor Algebra Compiler

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

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

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

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

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

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



Binary kernels are not enough

We need a Tensor Algebra Compiler

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

Infinite number of possible tensor formats and expressions

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

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

$$\begin{array}{ll} A = B + C & a = b + c \\ A = BC & a = \alpha Bc + \beta a \\ A = \alpha B & A = B^T \end{array}$$

Advantages of a Tensor Algebra Compiler

Traditional Libraries

Choose a few expressions

Choose a few formats

Spend years hand optimizing kernels

$$\begin{array}{ll} A = B + C & a = b + c \\ A = BC & a = \alpha Bc + \beta a \\ A = \alpha B & A = B^T \end{array}$$

The Tensor Algebra Compiler (taco)

Any expression

Many formats

Compiler produces optimized code

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

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

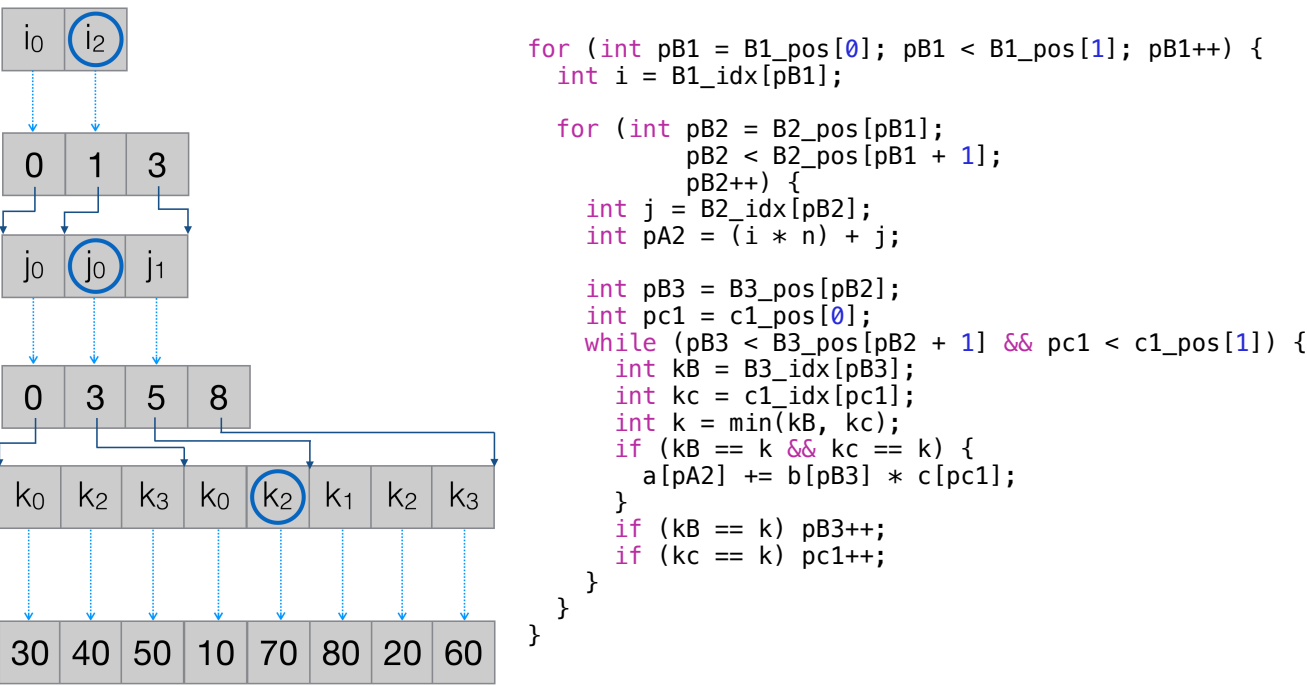
The Tensor Algebra Compiler (taco)

Any expression

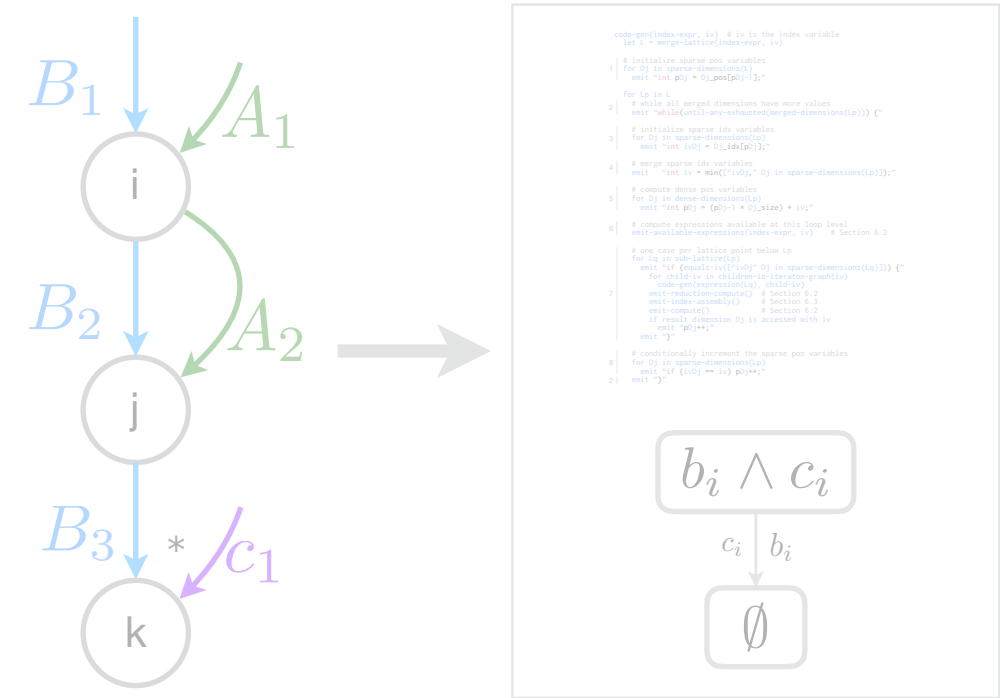
Many formats

Compiler produces optimized code

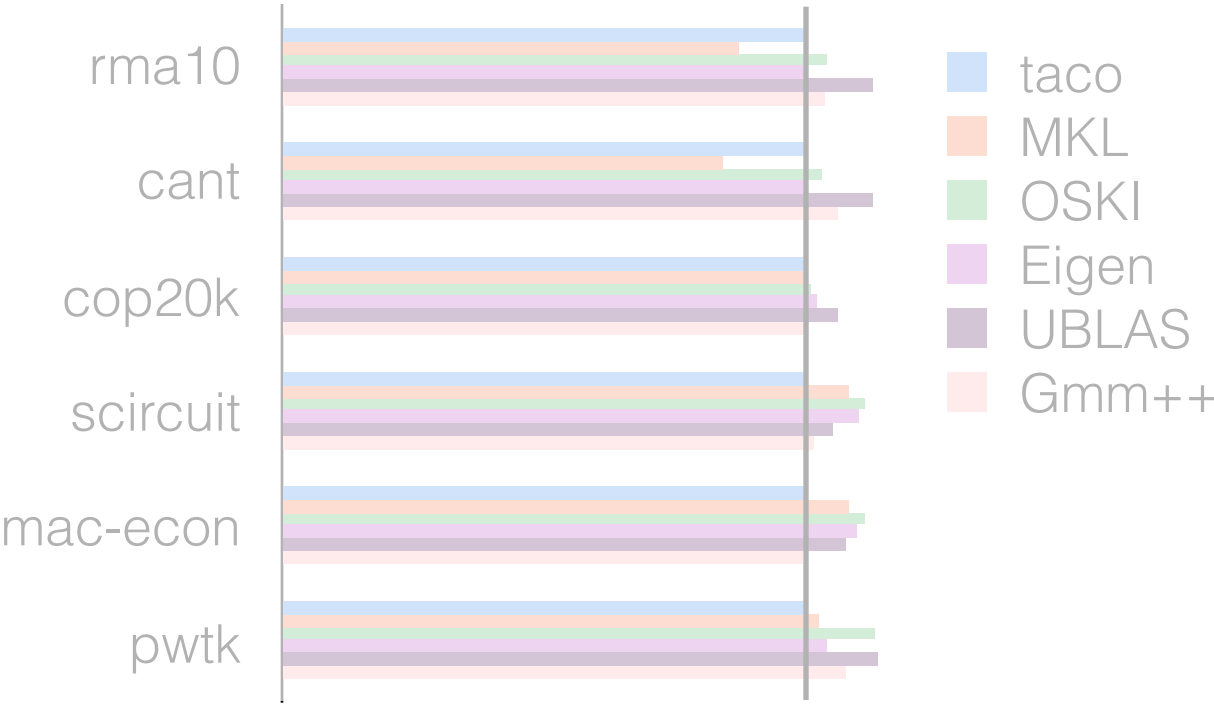
Sparse code and data



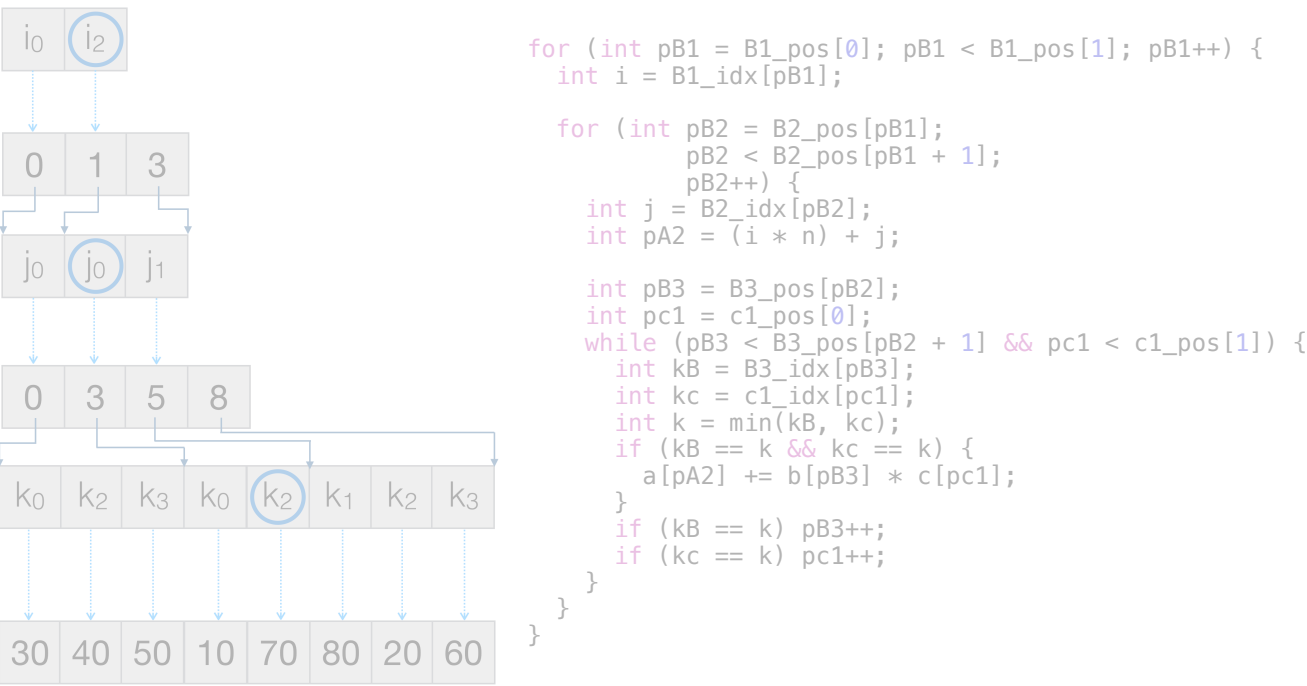
Intermediate Representation and Code Generation



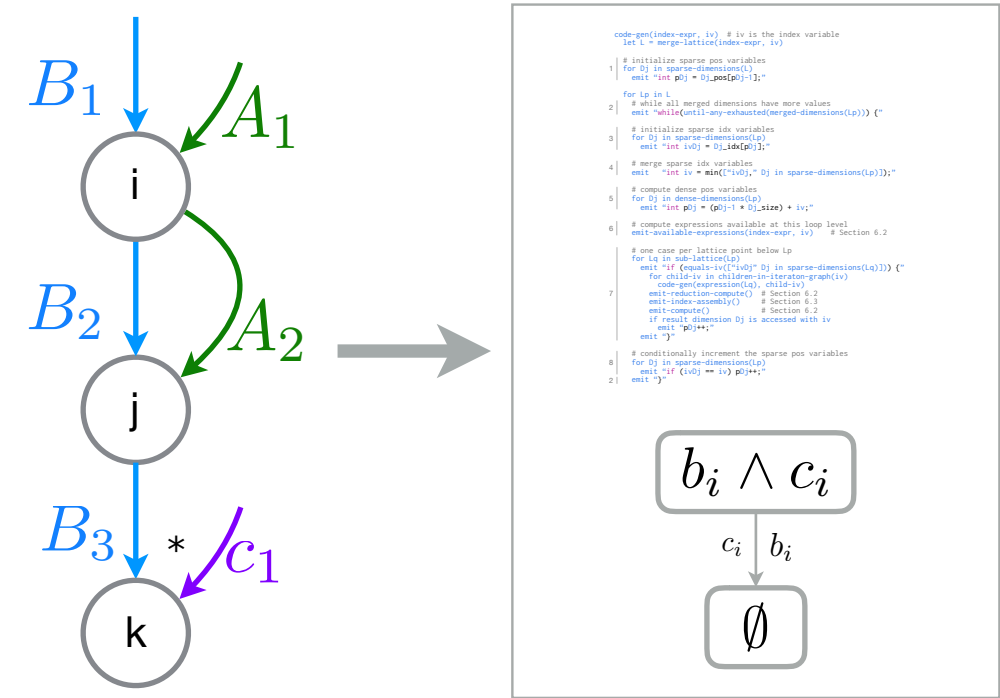
Evaluation



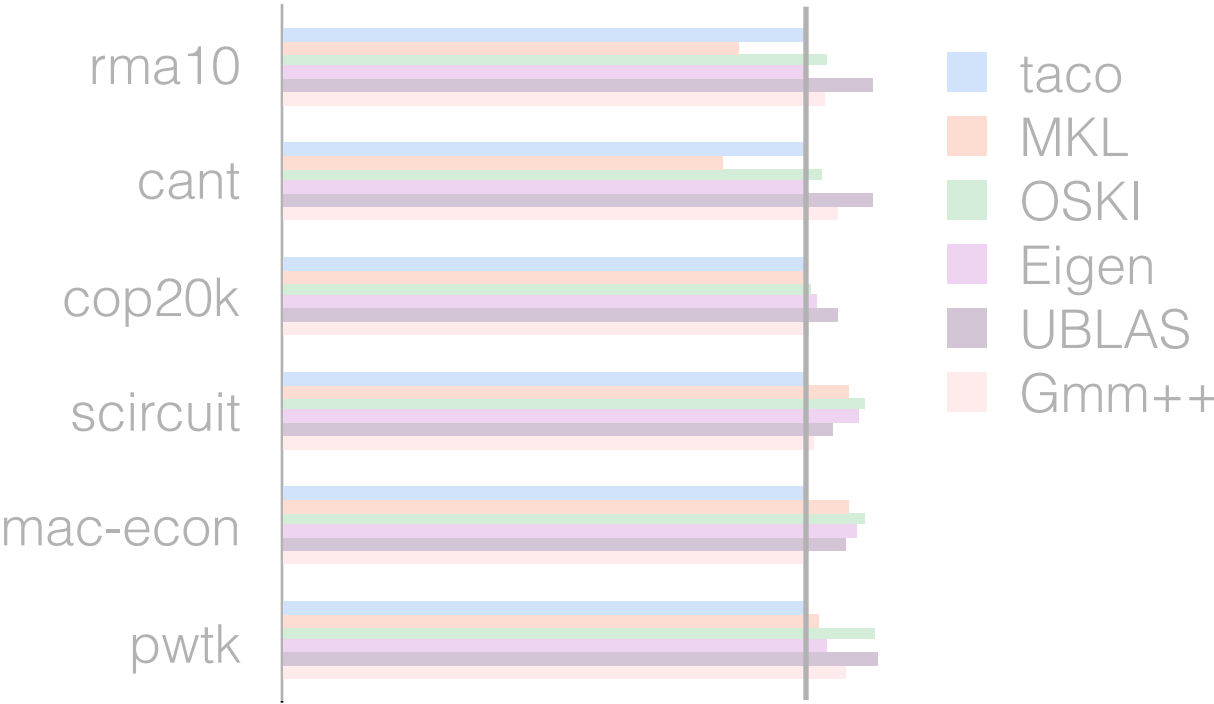
Sparse code and data



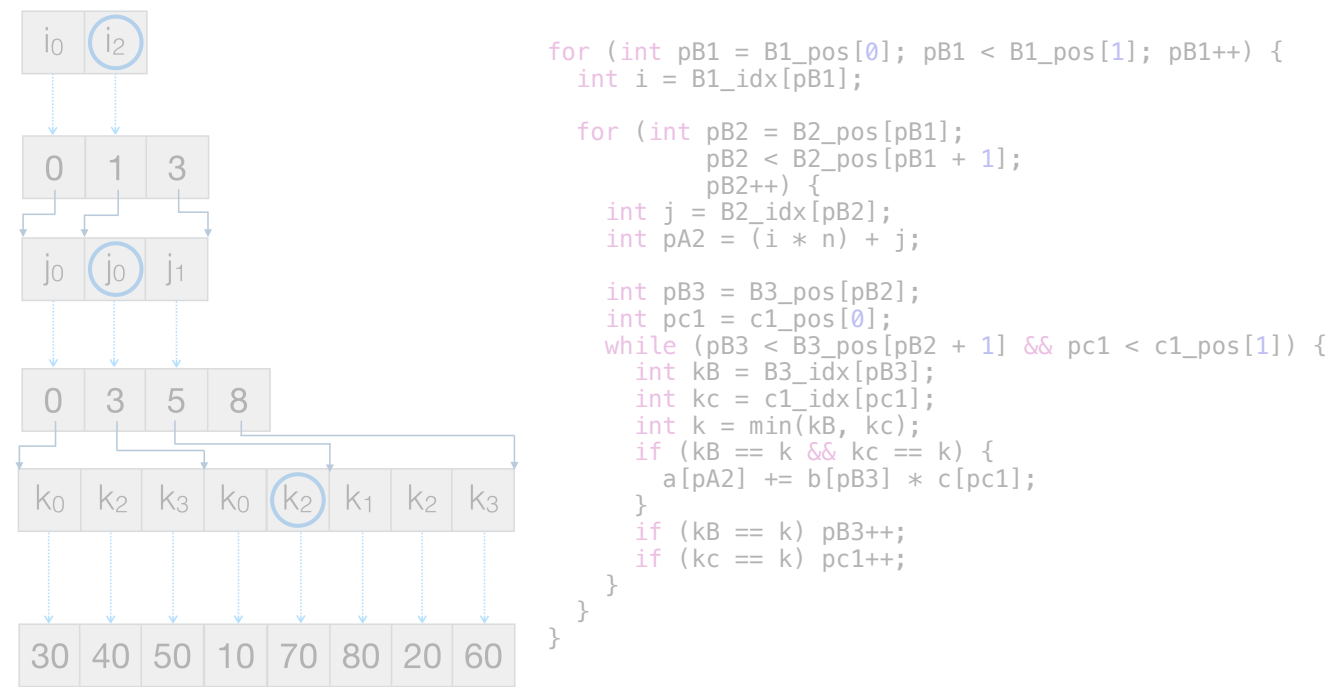
Intermediate Representation and Code Generation



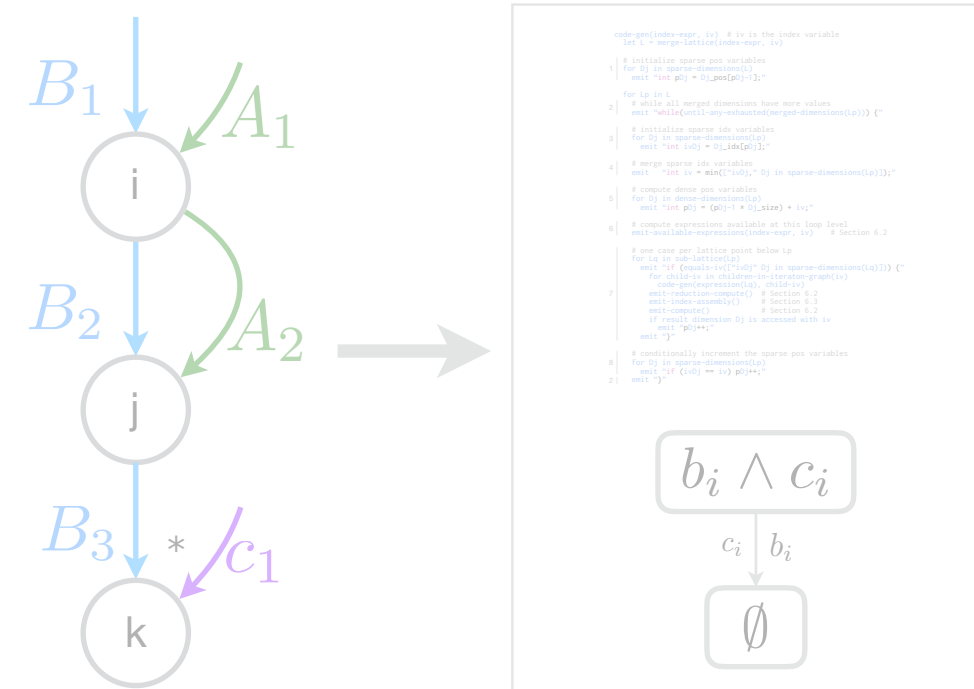
Evaluation



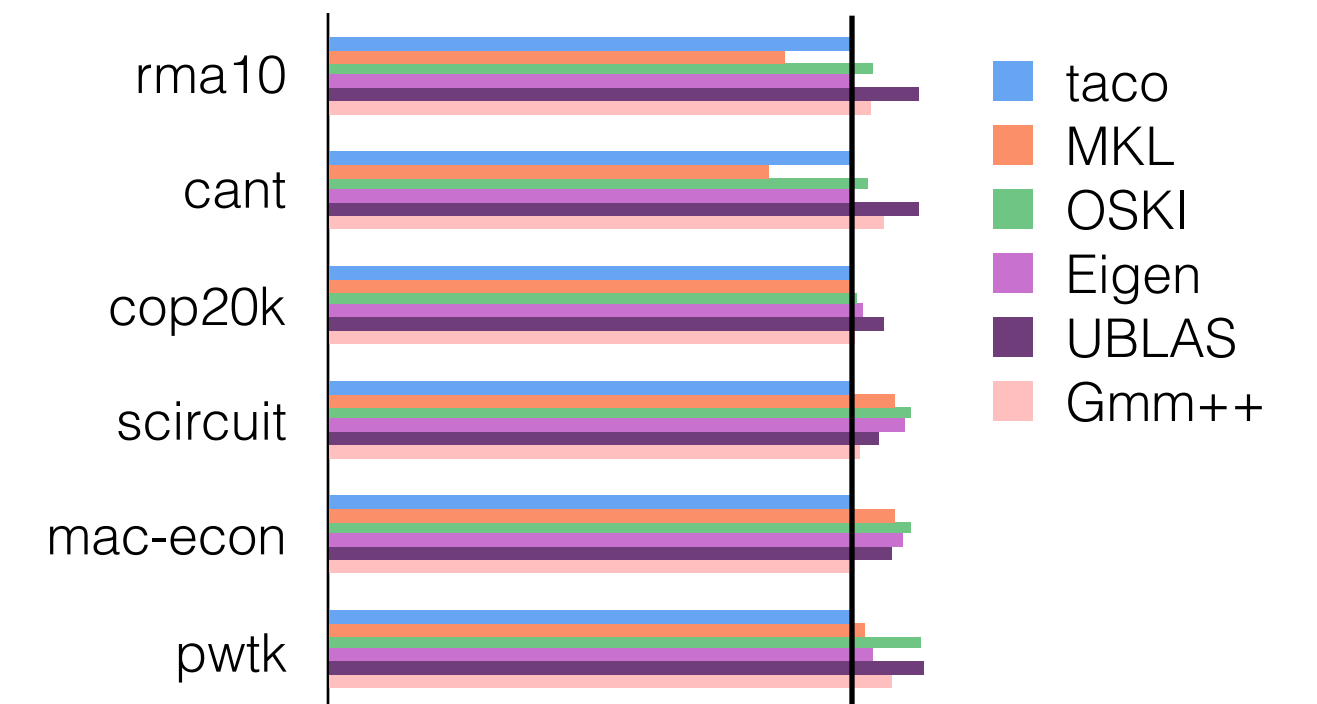
Sparse code and data



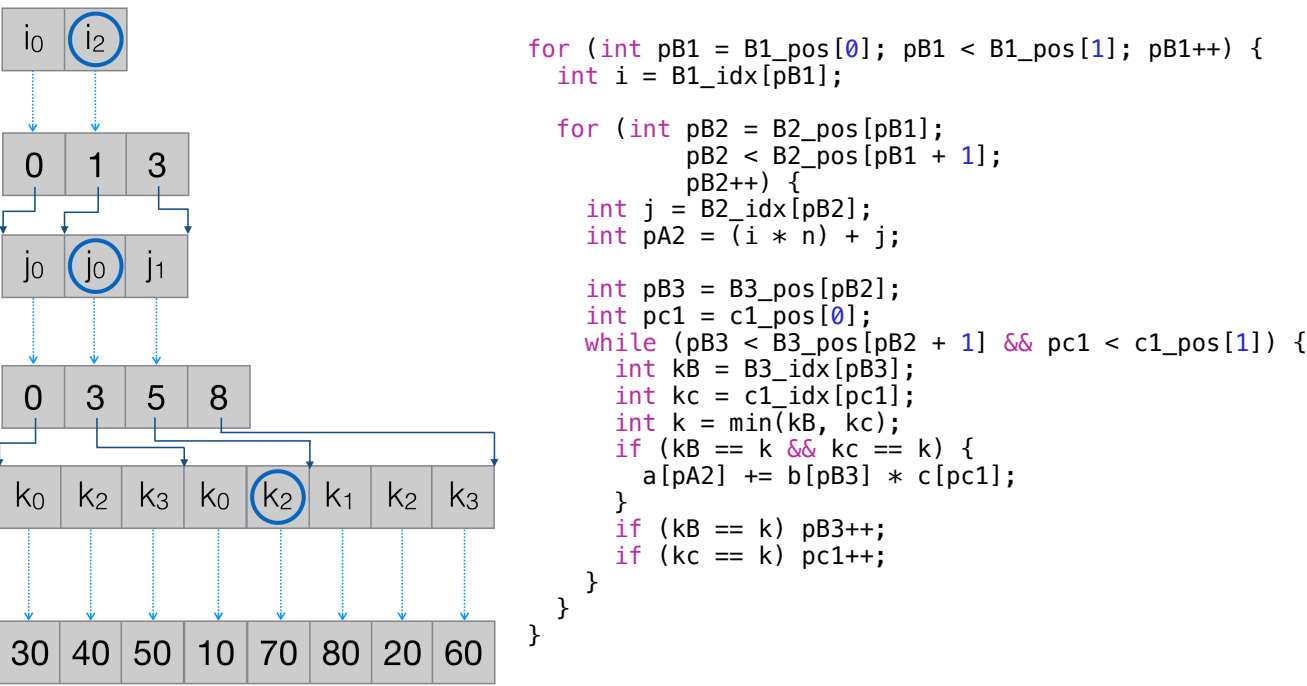
Intermediate Representation and Code Generation



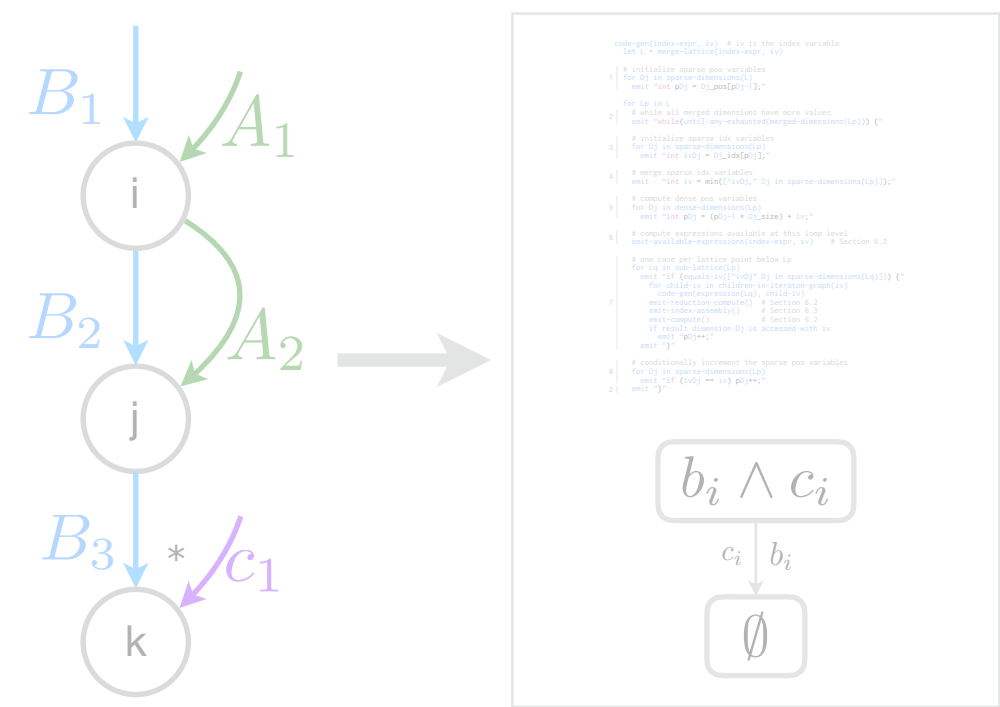
Evaluation



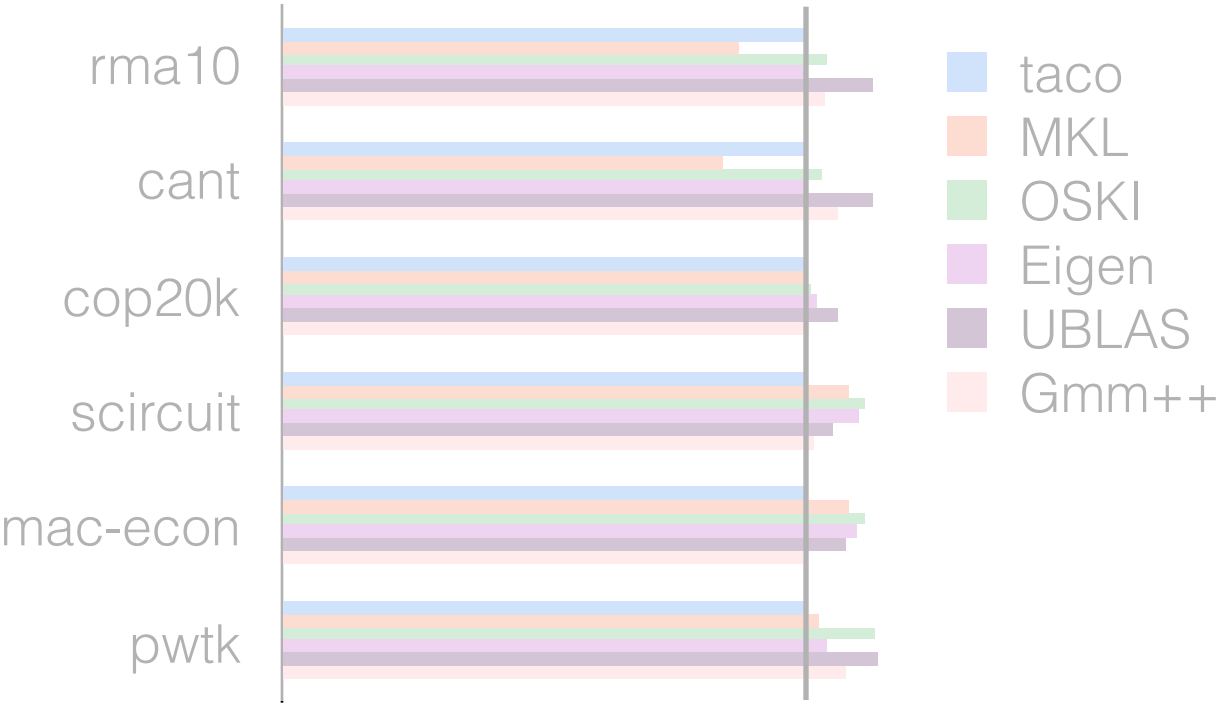
Sparse code and data



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

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

Tensor-vector multiplication example

Summation variable k

$$A_{ij} = \sum_{\underline{k}} B_{ij\underline{k}} * \underline{c_k}$$

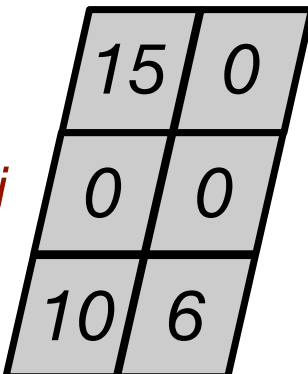
Tensor-vector multiplication example

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

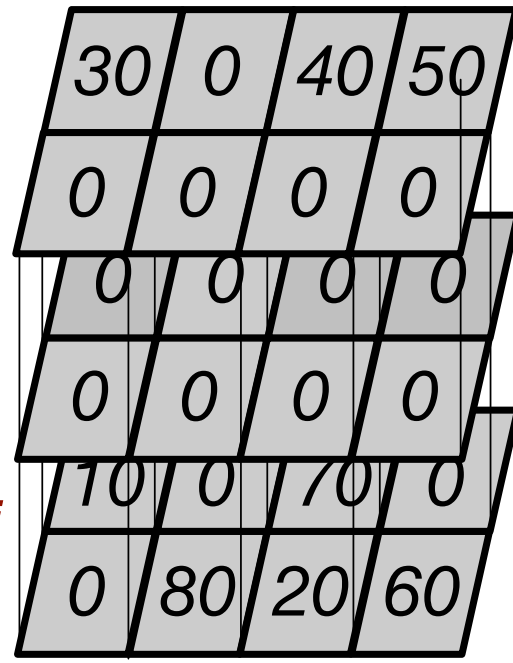
(dense, dense) (dense,dense,dense) (dense)

Tensor-vector multiplication example

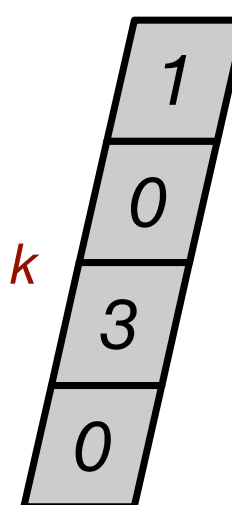
$$A_{ij} = \sum_k B_{ijk} * c_k$$



(dense, dense)



(dense,dense,dense)



(dense)

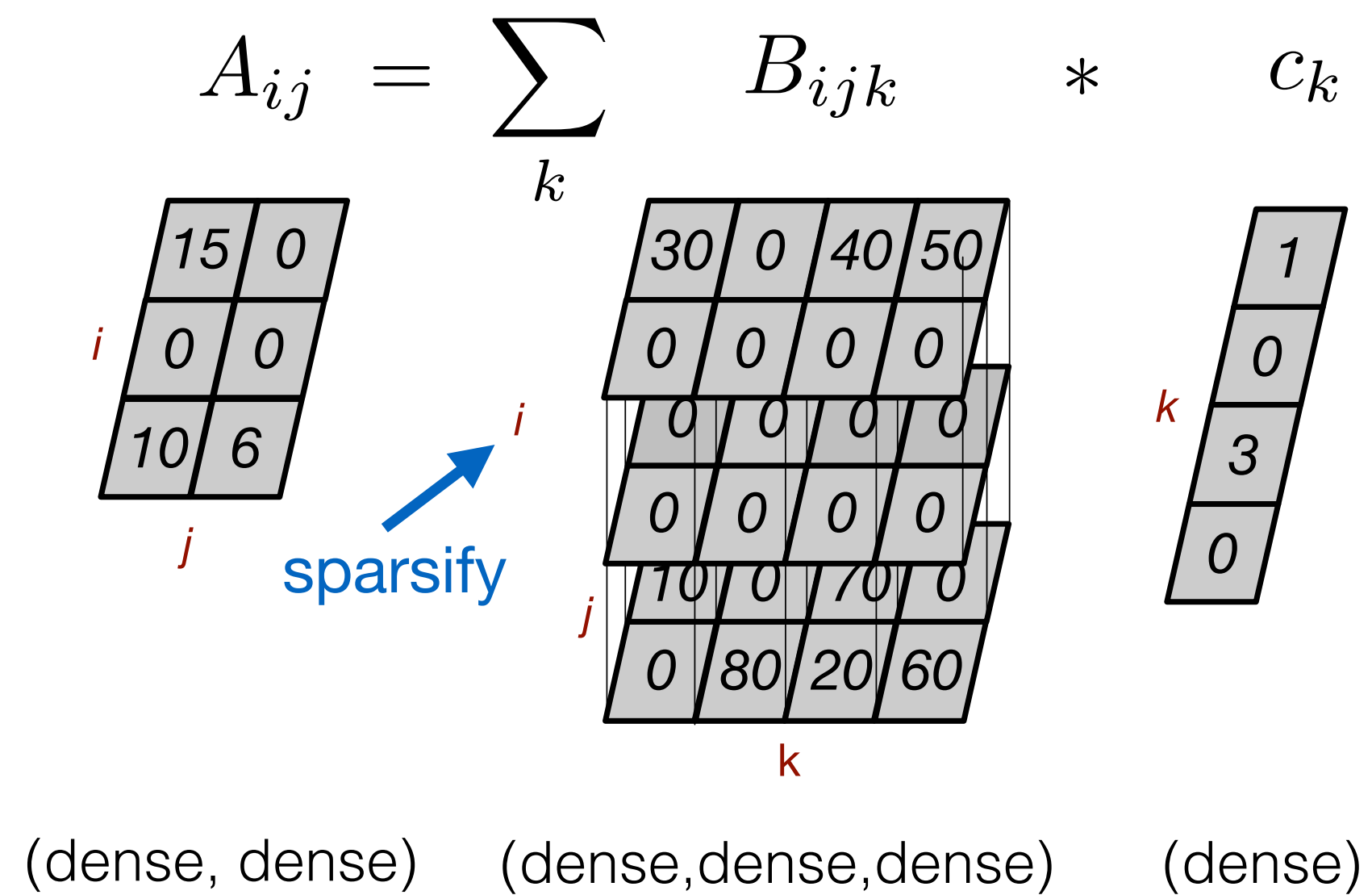
```

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

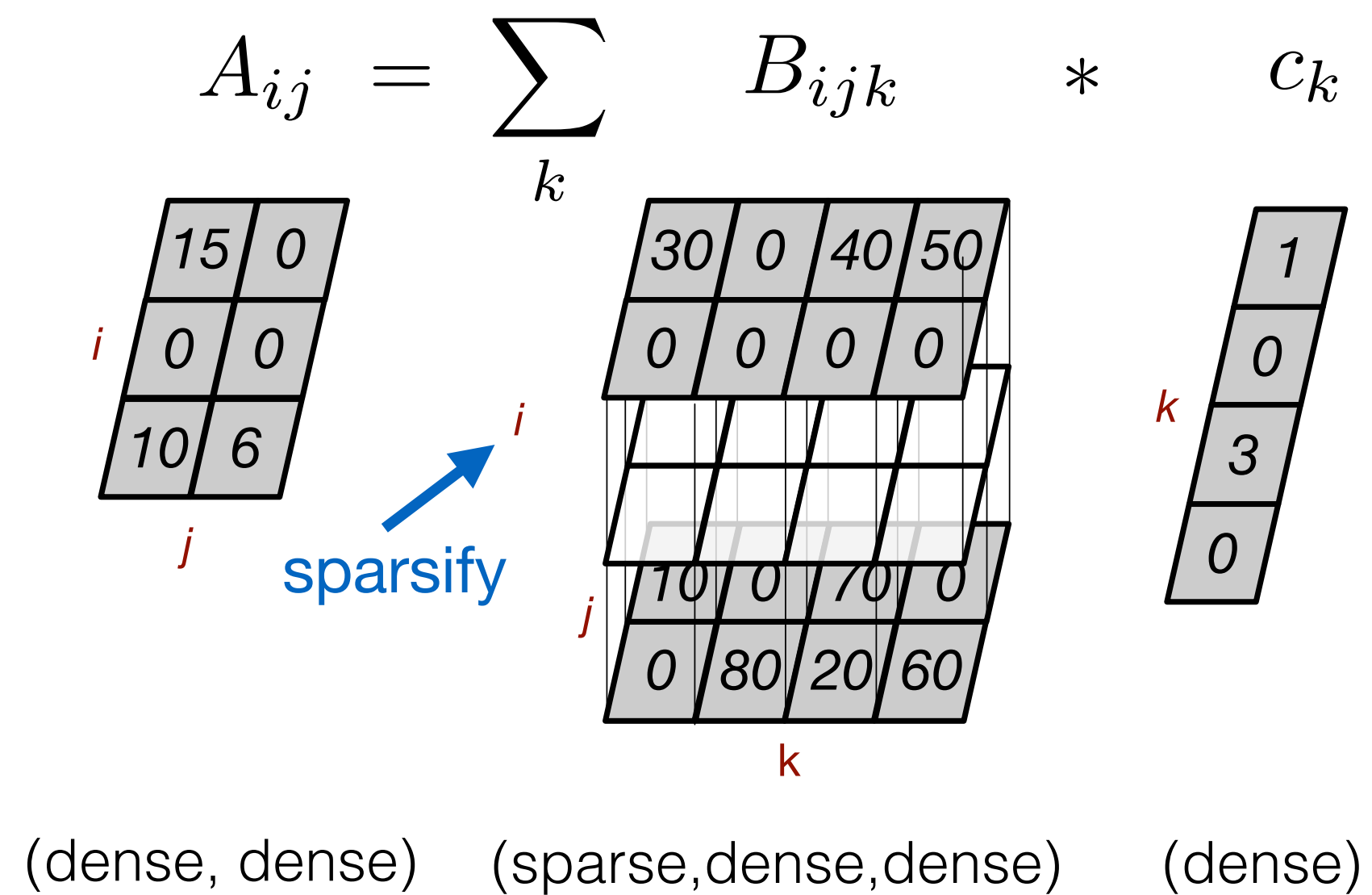


```
for (int i = 0; i < m; i++) {
    for (int j = 0; j < n; j++) {
        int pB2 = (i * n) + j;
        int pA2 = (i * n) + j;

        for (int k = 0; k < p; k++) {
            int pB3 = (pB2 * p) + k;

            a[pA2] += b[pB3] * c[k];
        }
    }
}
```

Tensor-vector multiplication example



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

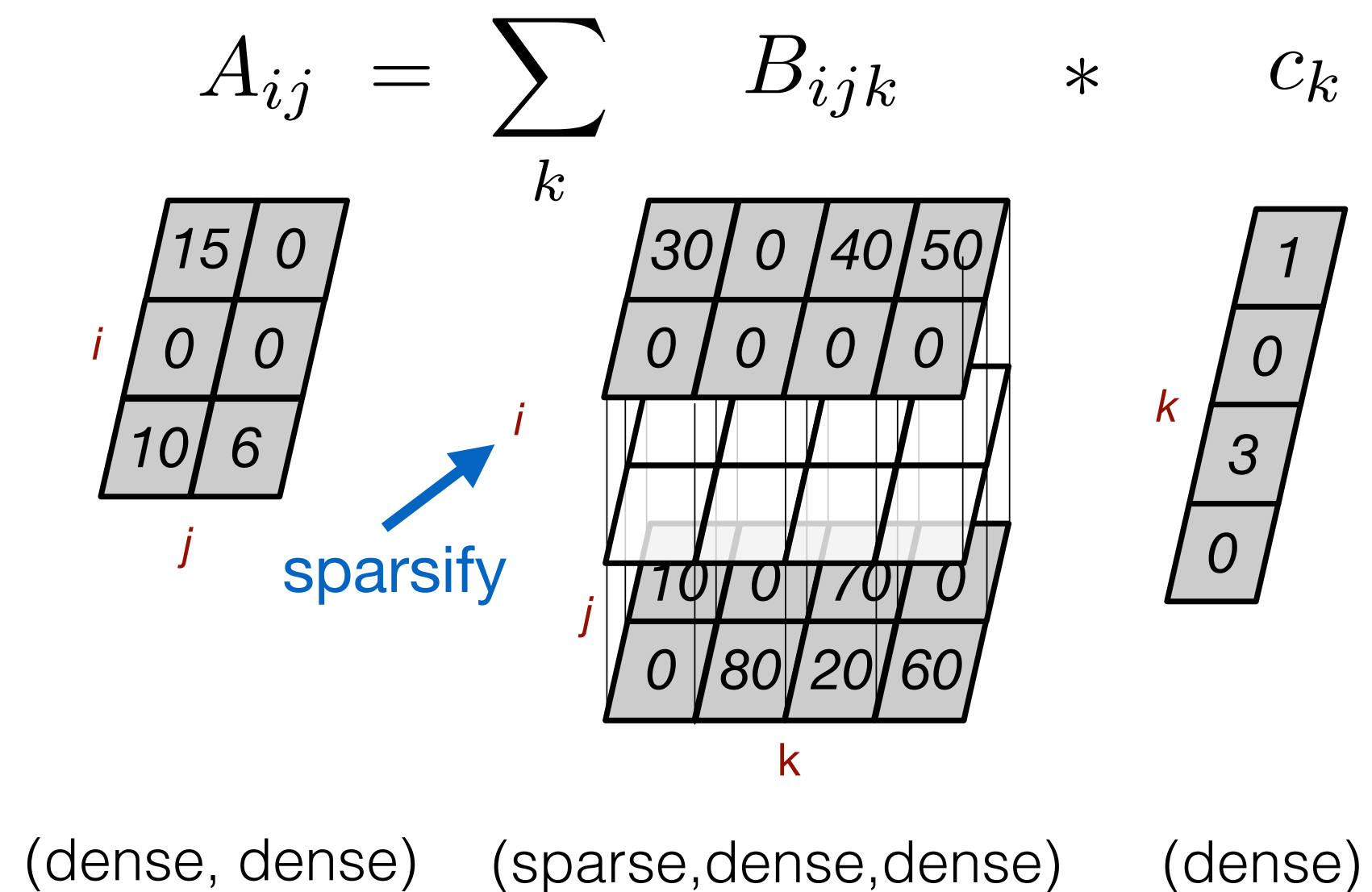
```
    for (int j = 0; j < n; j++) {
        int pB2 = (i * n) + j;
        int pA2 = (i * n) + j;
```

```
        for (int k = 0; k < p; k++) {
            int pB3 = (pB2 * p) + k;
```

```
            a[pA2] += b[pB3] * c[k];
```

```
        }
    }
}
```

Tensor-vector multiplication example



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

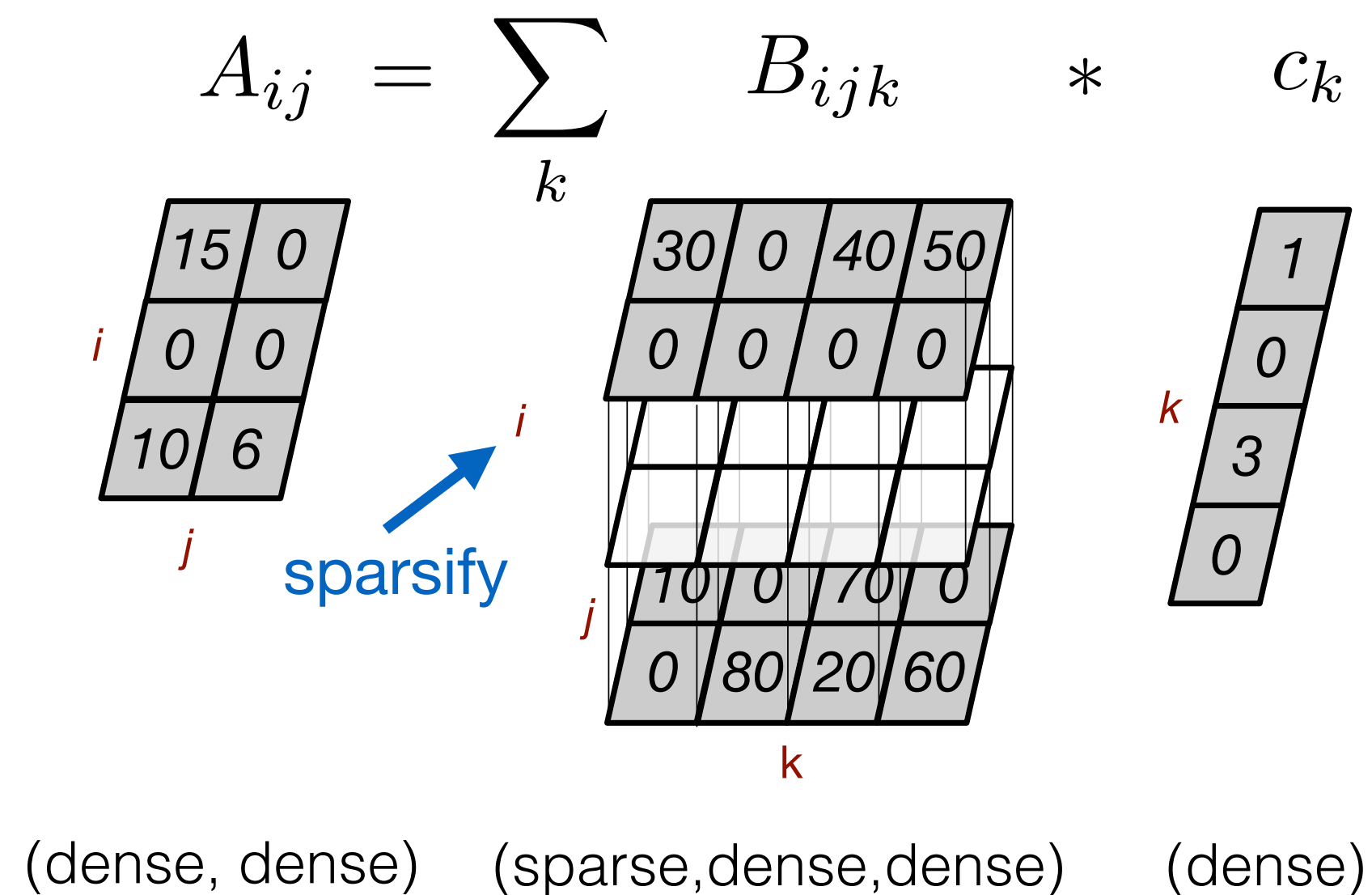
```
    for (int j = 0; j < n; j++) {
        int pB2 = (i * n) + j;
        int pA2 = (i * n) + j;
```

```
        for (int k = 0; k < p; k++) {
            int pB3 = (pB2 * p) + k;
```

```
            a[pA2] += b[pB3] * c[k];
```

```
        }
    }
}
```

Tensor-vector multiplication example



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

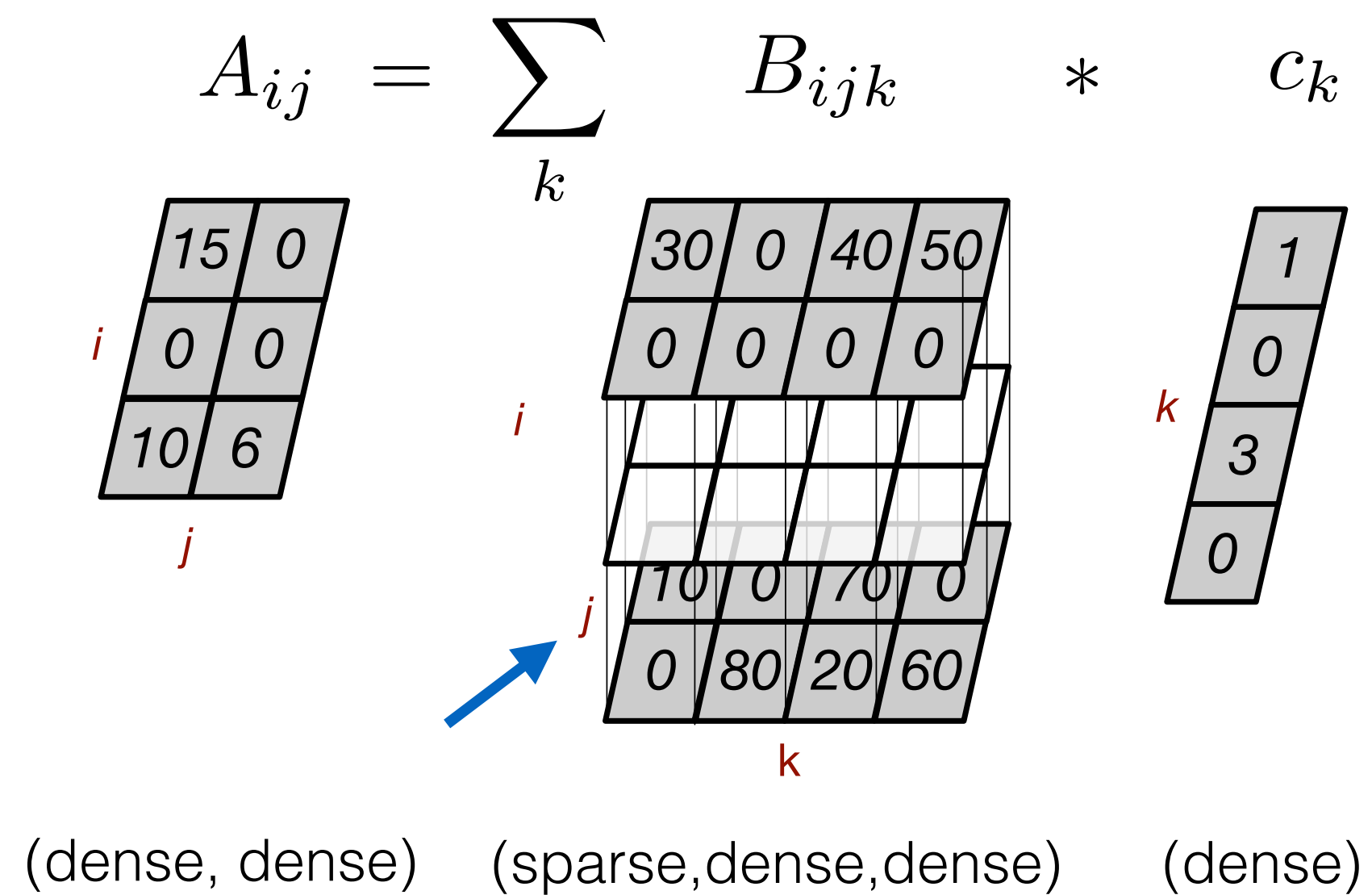
```
    for (int j = 0; j < n; j++) {
        int pB2 = (pB1 * n) + j;
        int pA2 = (i * n) + j;
```

```
        for (int k = 0; k < p; k++) {
            int pB3 = (pB2 * p) + k;
```

```
            a[pA2] += b[pB3] * c[k];
```

```
        }
    }
}
```

Tensor-vector multiplication example



```
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

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

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

```
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) (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

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

(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)

```
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)

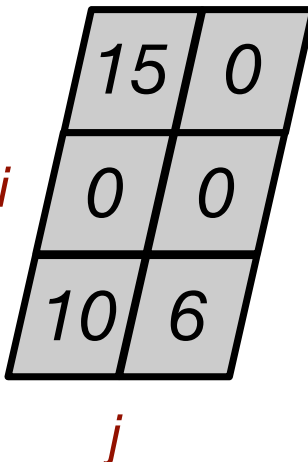
```
for (int pB1 = B1_pos[0]; pB1 < B1_pos[1]; pB1++) {
    int i = B1_idx[pB1];

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

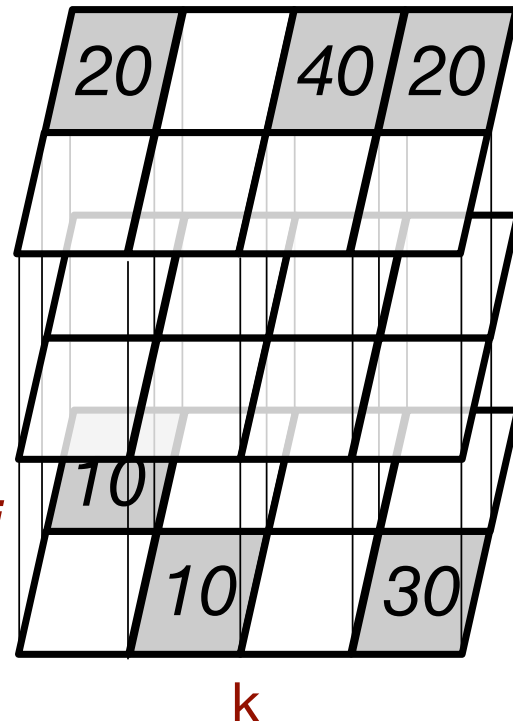
        int pB3 = B3_pos[pB2];
        int pc1 = c1_pos[0];
        while (pB3 < B3_pos[pB2 + 1] && pc1 < c1_pos[1]) {
            int kB = B3_idx[pB3];
            int kc = c1_idx[pc1];
            int k = min(kB, kc);
            if (kB == k && kc == k) {
                a[pA2] += b[pB3] * c[pc1];
       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

Tensor-vector multiplication example

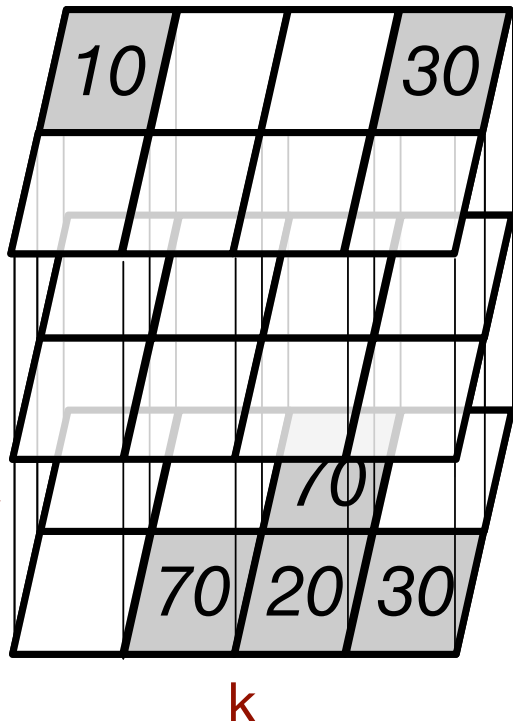
$$A_{ij} = \sum_k \left(\underline{B_{ijk}} + \underline{D_{ijk}} \right) * c_k$$



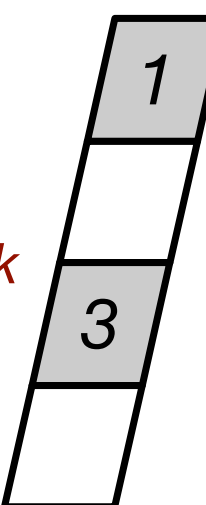
(dense, dense)



(sparse, sparse, sparse)



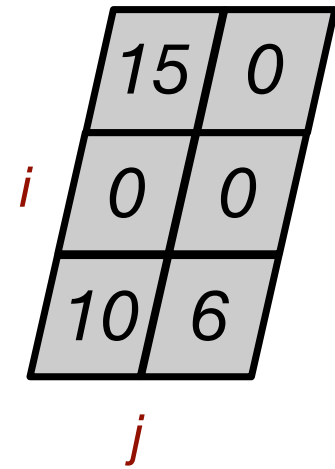
(sparse, sparse, sparse)



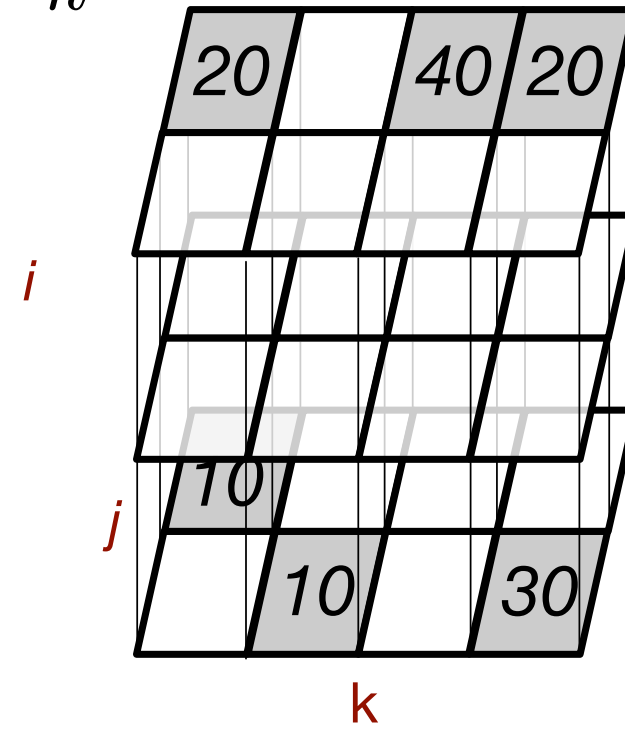
(sparse)

Tensor-vector multiplication example

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

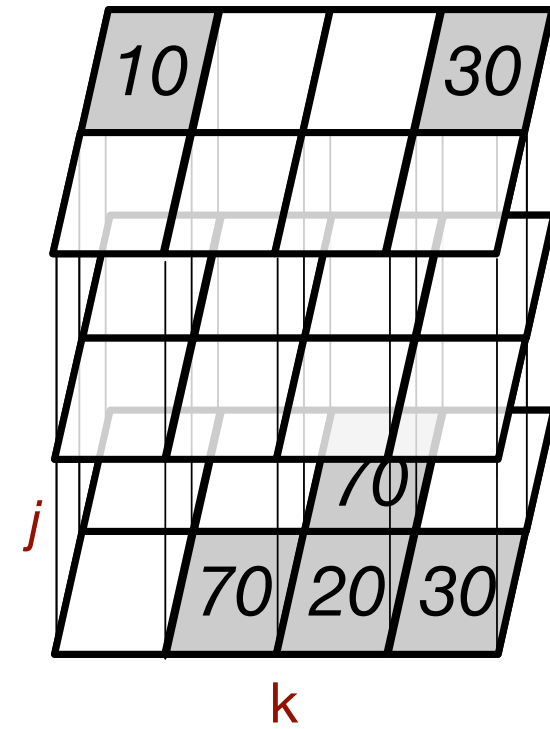


(dense, dense)

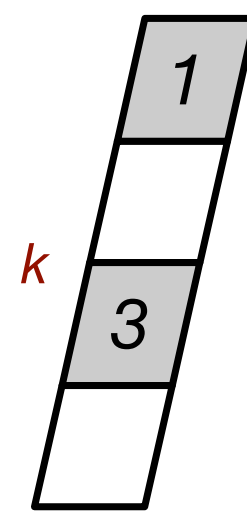


(sparse,sparse,sparse)

(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, kD, kc);
                    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 (pB2 < B2_pos[pB1 + 1]) {
    int jB0 = B2_idx[pB2];
    int pA20 = (i * A2_dimension) + jB0;
    double tk2 = 0.0;
    int pB31 = B3_pos[pB2];
    int pc12 = c1_pos[0];
    while (pB31 < B3_pos[pB2 + 1] &&
          pc12 < c1_pos[1]) {
        int kB2 = B3_idx[pB31];
        int kc4 = c1_idx[pc12];
        int k4 = min(kB2, kc4);
        if (kB2 == k4 && kc4 == k4) {
            tk2 += b[pB31] * c[pc12];
        }
        if (kB2 == k4) pB31++;
        if (kc4 == k4) pc12++;
    }
    a[pA20] = tk2;
    pB2++;
}
while (pD2 < D2_pos[pD1 + 1]) {
    int jD0 = D2_idx[pD2];
    int pA21 = (i * A2_dimension) + jD0;
    double tk3 = 0.0;
    int pD31 = D3_pos[pD2];
    int pc13 = c1_pos[0];
    while (pD31 < D3_pos[pD2 + 1] &&
          pc13 < c1_pos[1]) {
        int kD2 = D3_idx[pD31];
        int kc5 = c1_idx[pc13];
        int k5 = min(kD2, kc5);
        if (kD2 == k5 && kc5 == k5) {
            tk3 += d[pD31] * c[pc13];
        }
        if (kD2 == k5) pD31++;
        if (kc5 == k5) pc13++;
    }
    a[pA21] = tk3;
    pD2++;
}
else if (iB == i) {
    for (int pB2 = B2_pos[pB1];
         pB2 < B2_pos[pB1 + 1];
         pB2++) {
        int jB1 = B2_idx[pB2];
        int pA22 = (i * A2_dimension) + jB1;
        double tk4 = 0.0;
        int pB32 = B3_pos[pB2];
        int pc14 = c1_pos[0];
        while (pB32 < B3_pos[pB2 + 1] &&
              pc14 < c1_pos[1]) {
            int kB3 = B3_idx[pB32];
            int kc6 = c1_idx[pc14];
            int k6 = min(kB3, kc6);
            if (kB3 == k6 && kc6 == k6) {
                tk4 += b[pB32] * c[pc14];
            }
            if (kB3 == k6) pB32++;
            if (kc6 == k6) pc14++;
        }
        a[pA22] = tk4;
    }
}
else {
    for (int pD2 = D2_pos[pD1];
         pD2 < D2_pos[pD1 + 1];
         pD2++) {
        int jD1 = D2_idx[pD2];
        int pA23 = (i * A2_dimension) + jD1;
        double tk5 = 0;
        int pD32 = D3_pos[pD2];
        int pc15 = c1_pos[0];
        while (pD32 < D3_pos[pD2 + 1] &&
              pc15 < c1_pos[1]) {
            int kD3 = D3_idx[pD32];
            int kc7 = c1_idx[pc15];
            int k7 = min(kD3, kc7);
            if (kD3 == k7 && kc7 == k7) {
                tk5 += d[pD32] * c[pc15];
            }
            if (kD3 == k7) pD32++;
            if (kc7 == k7) pc15++;
        }
        a[pA23] = tk5;
    }
}
if (iB == i) pB1++;
if (iD == i) pD1++;
}
```

```
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 = (iB0 * A2_dimension) + jB2;
        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 (kB4 == k8) pB33++;
            if (kc8 == k8) pc16++;
        }
        a[pA24] = tk6;
    }
    pB1++;
}
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 = (iD0 * A2_dimension) + jD2;
        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 (kD4 == k9) pD33++;
            if (kc9 == k9) pc17++;
        }
        a[pA25] = tk7;
    }
    pD1++;
}
```

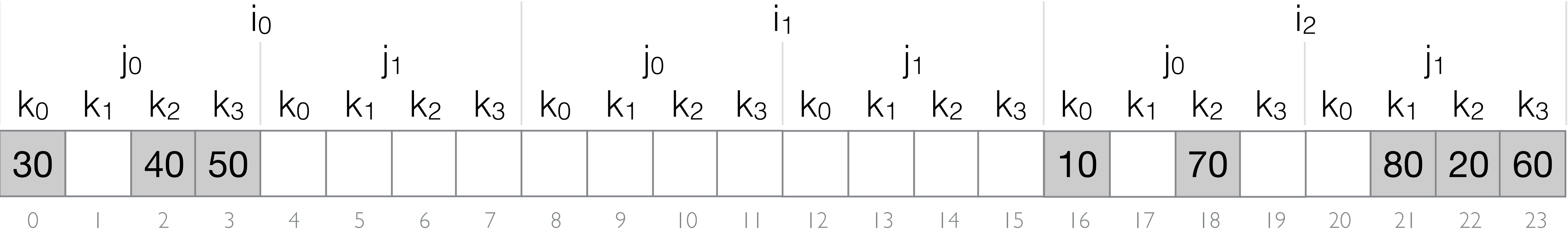
Dense storage formats

		k_0	k_1	k_2	k_3
i_0	j_0	30		40	50
	j_1				

		k_0	k_1	k_2	k_3
i_1	j_0				
	j_1				

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

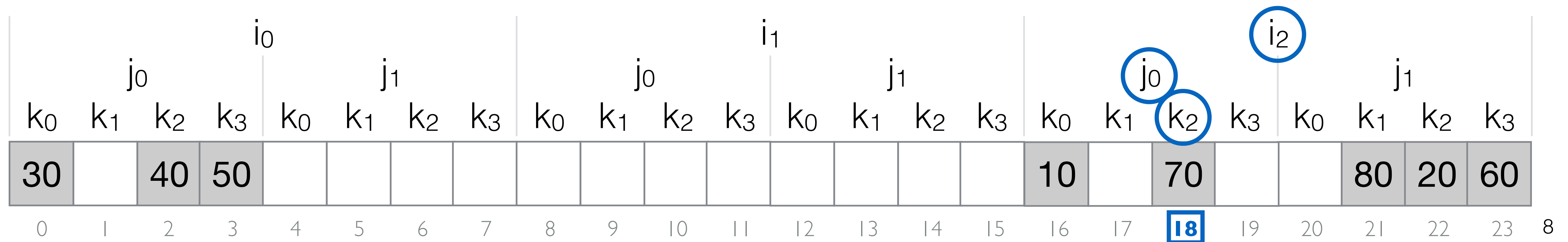
Dense storage formats



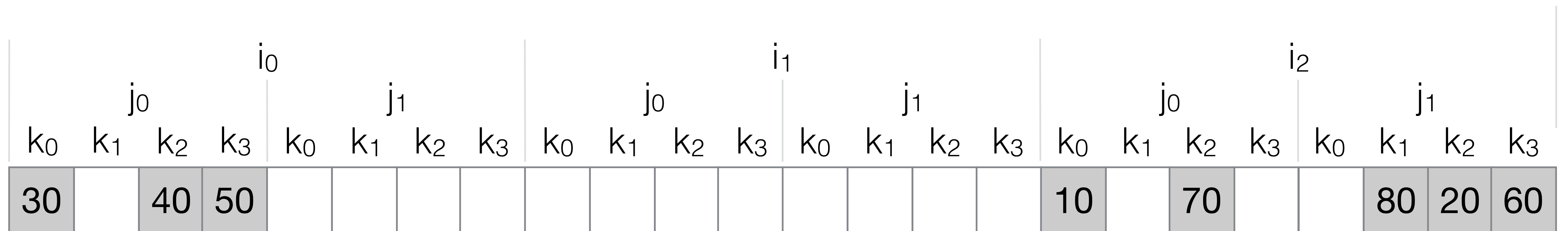
Dense storage formats

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

Random access: easy to locate element, but wasted storage



Compressed storage formats



Compressed storage formats

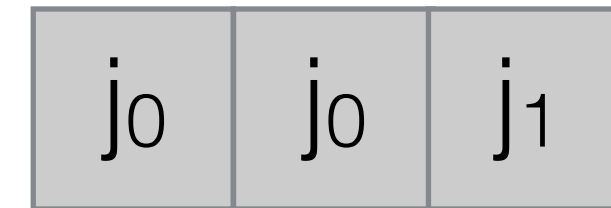
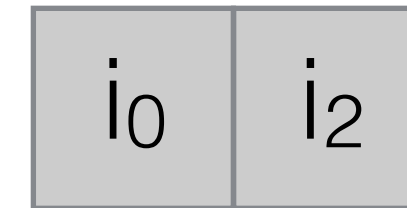
i_0			j_0			i_2	
j_0			j_0			j_1	
k_0	k_2	k_3	k_0	k_2	k_1	k_2	k_3
30	40	50	10	70	80	20	60

Compressed storage formats

i_0	i_2
-------	-------

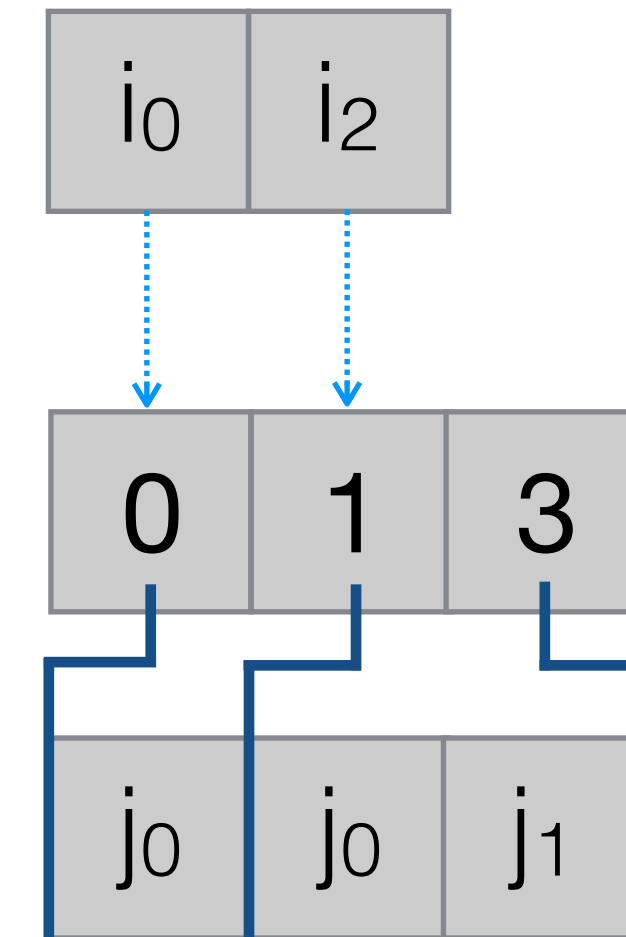
j_0			j_0		j_1		
k_0	k_2	k_3	k_0	k_2	k_1	k_2	k_3
30	40	50	10	70	80	20	60

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

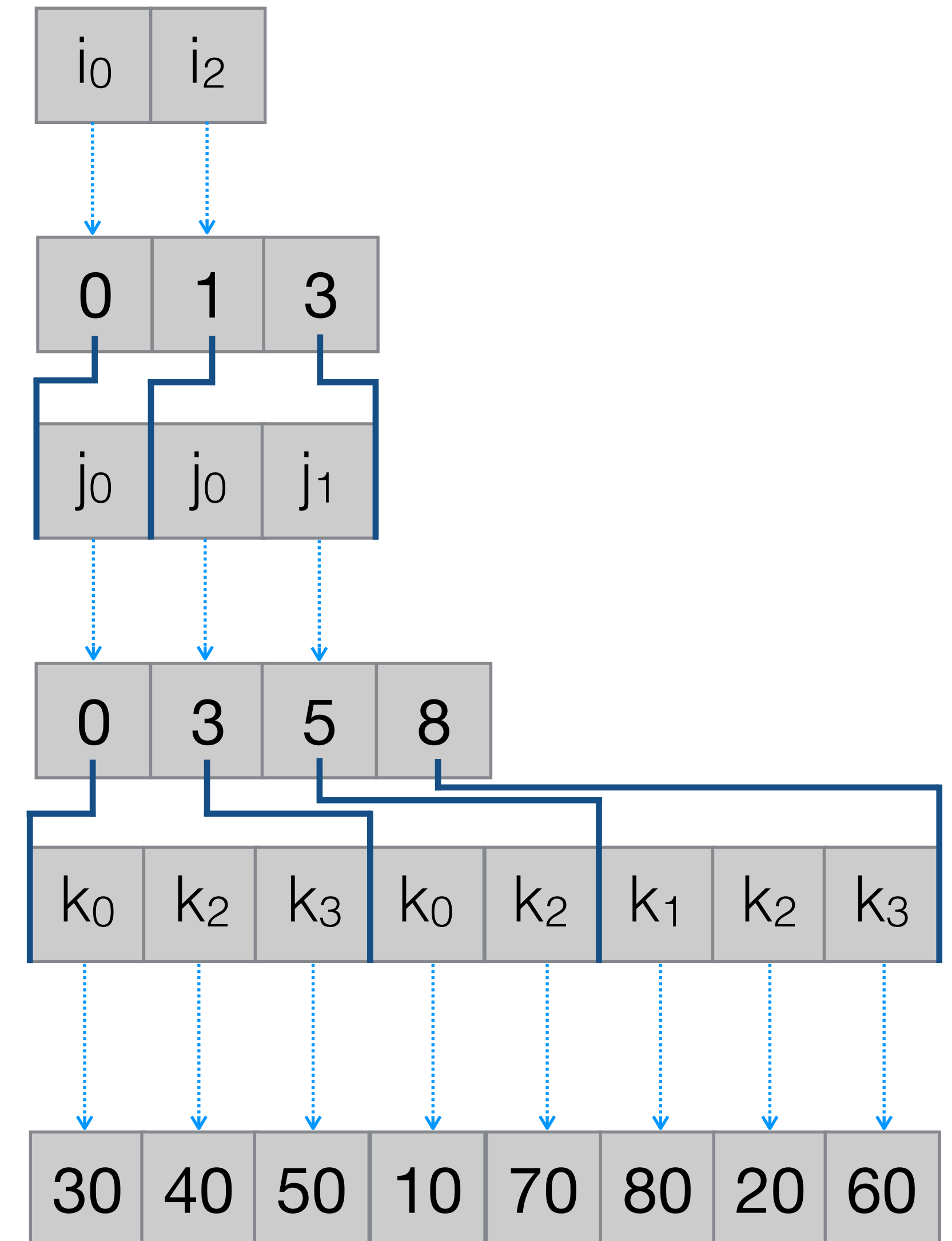
Compressed storage formats



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

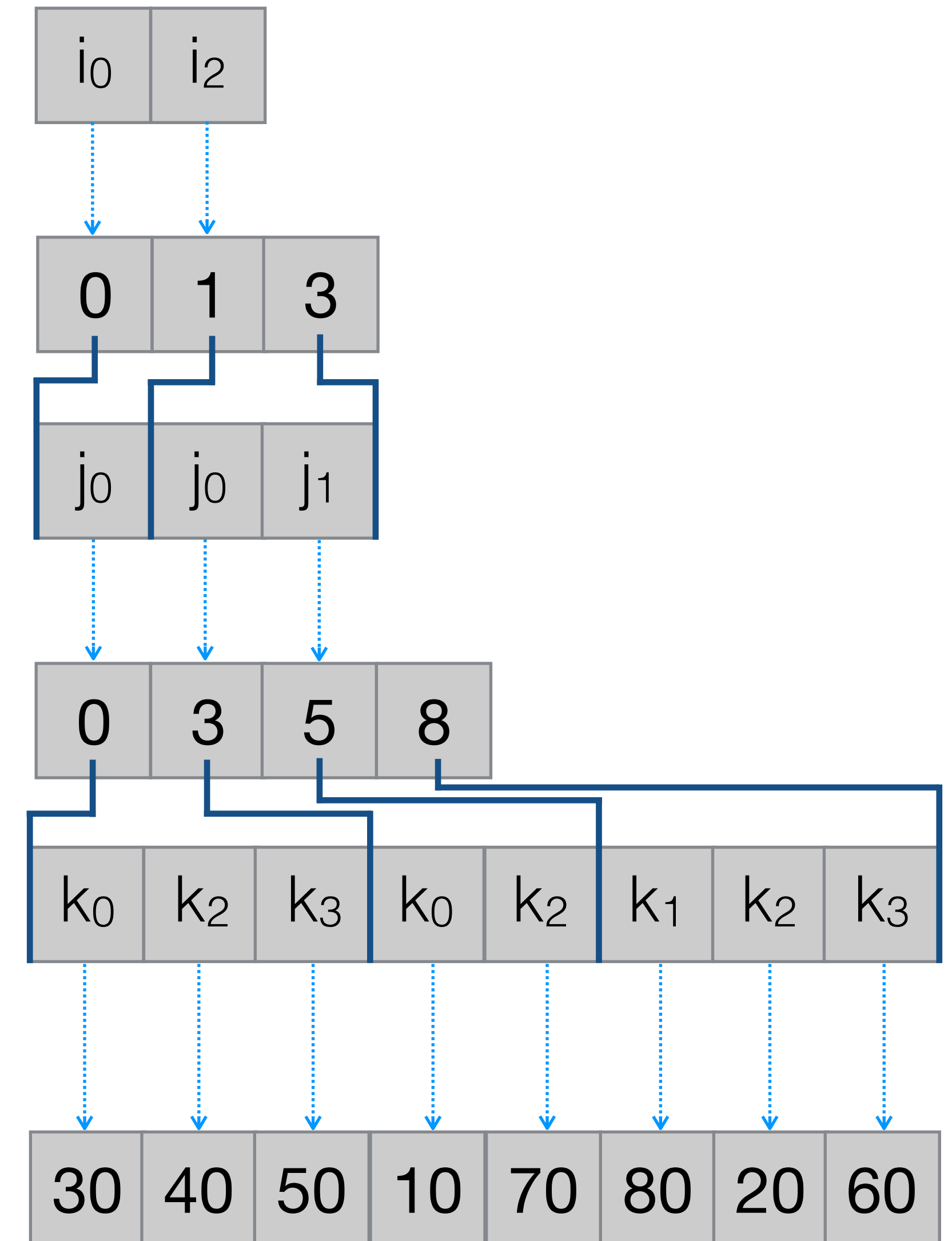
No random access: must traverse level by level to locate coordinate



Compressed storage formats

locate (i_2, j_0, k_2)

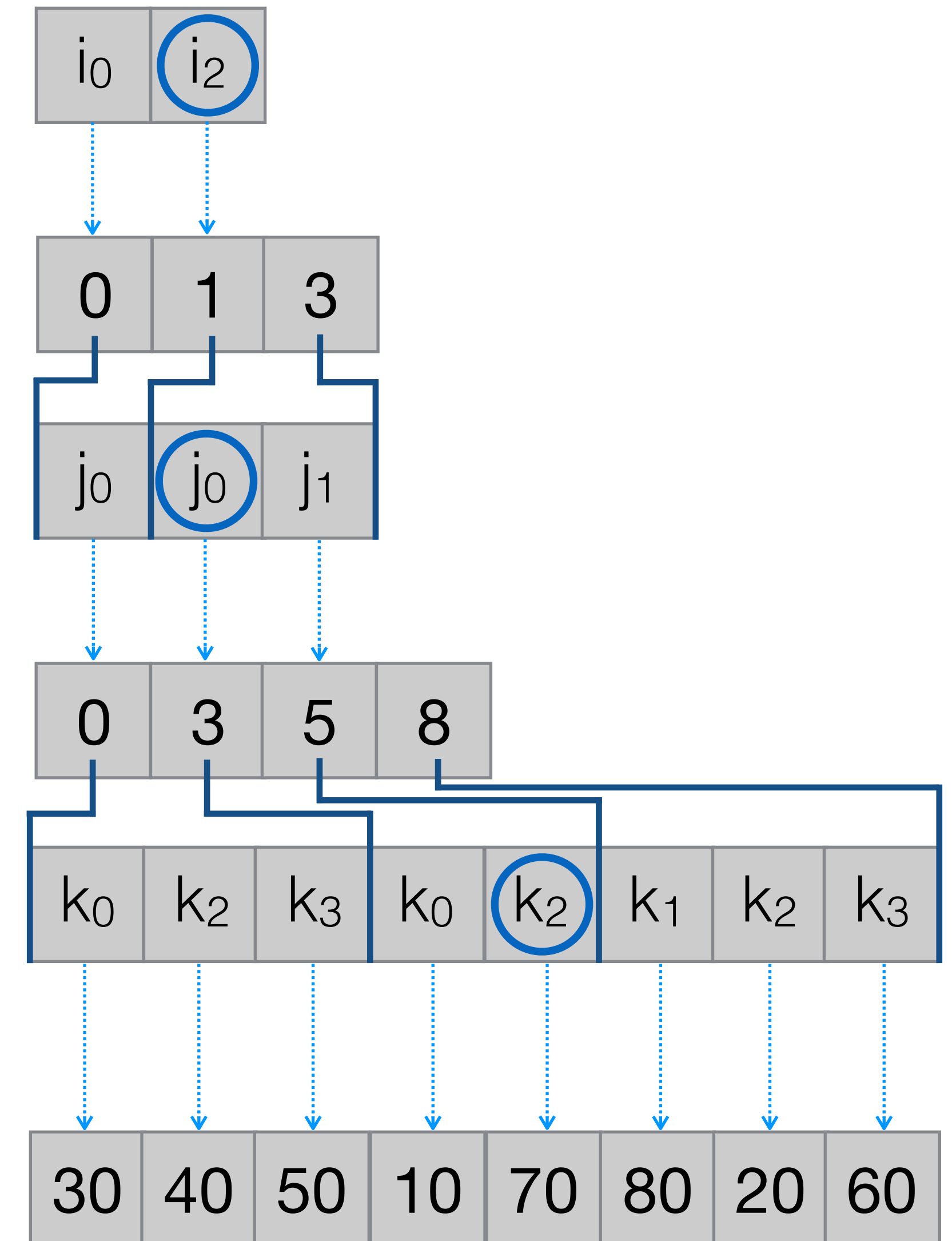
No random access: must traverse level by level to locate coordinate



Compressed storage formats

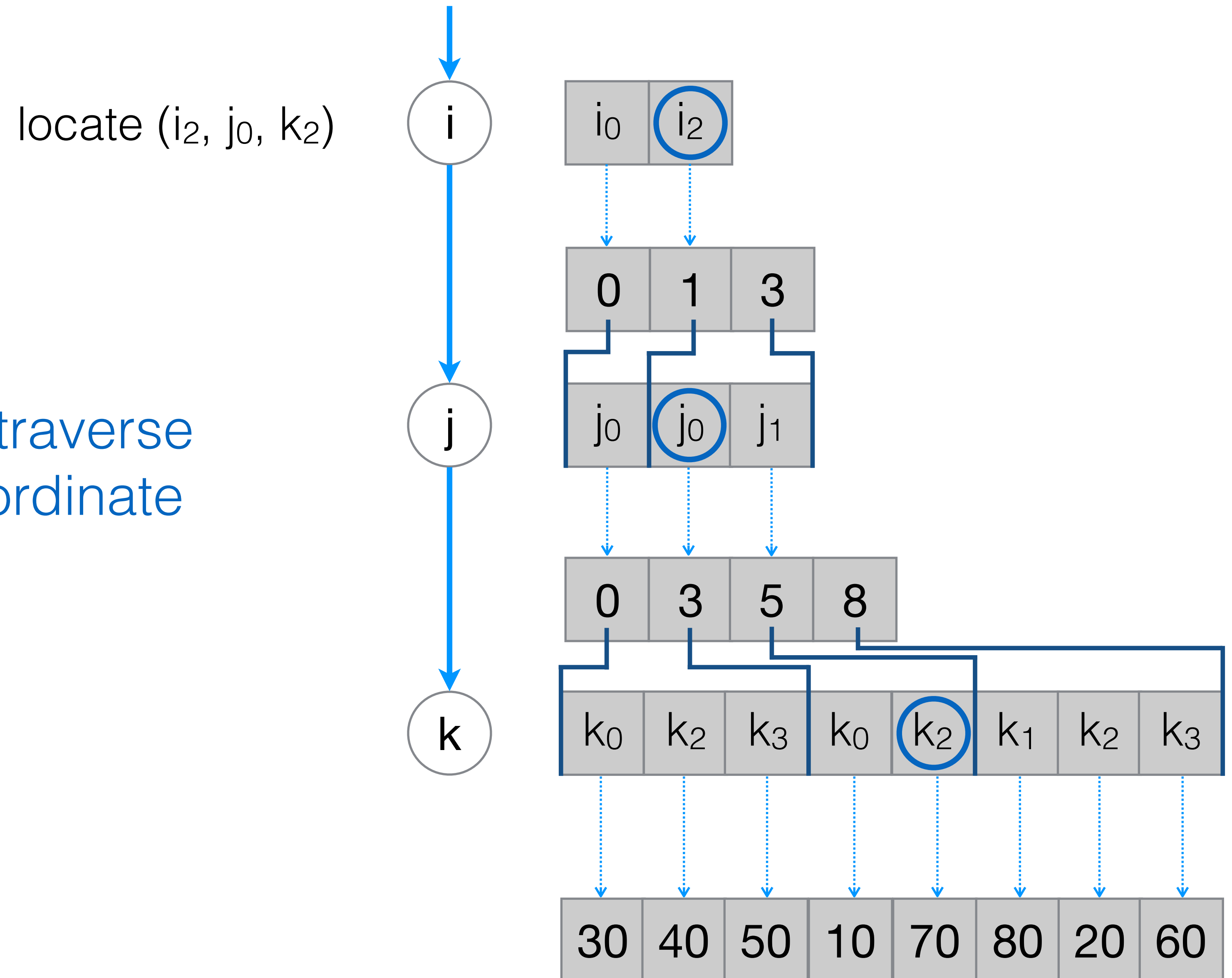
locate (i_2, j_0, k_2)

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

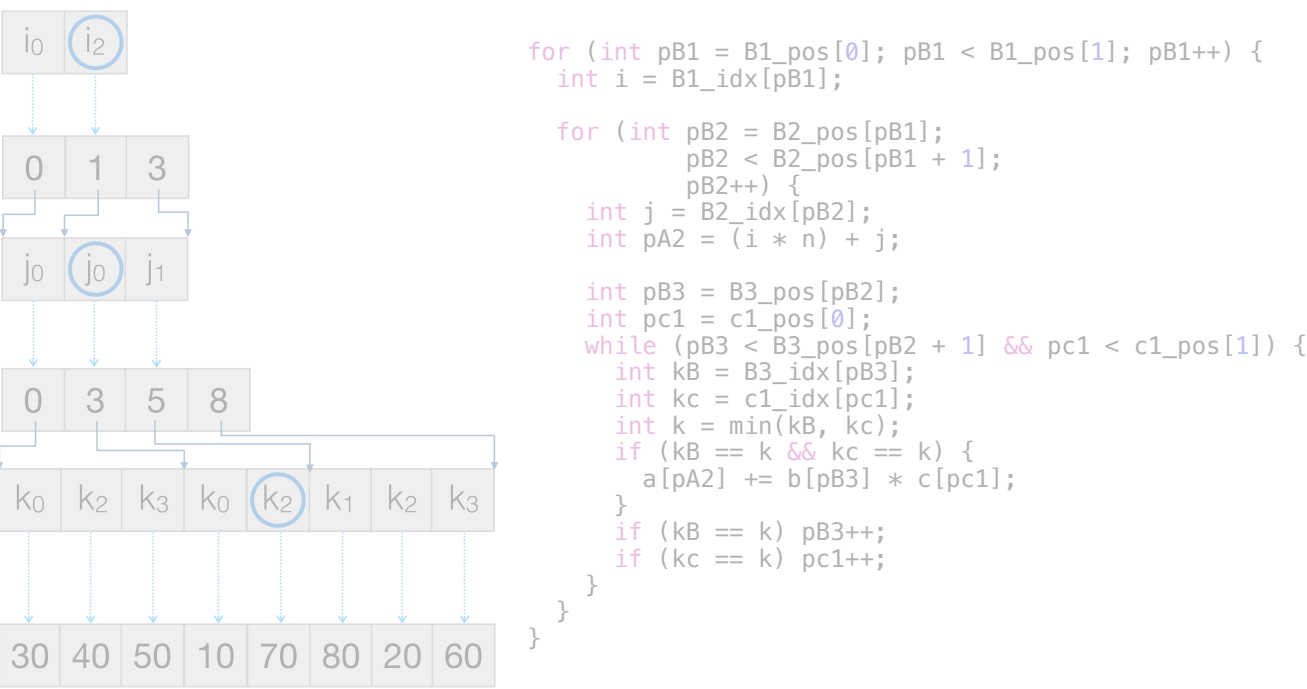


Compressed storage formats

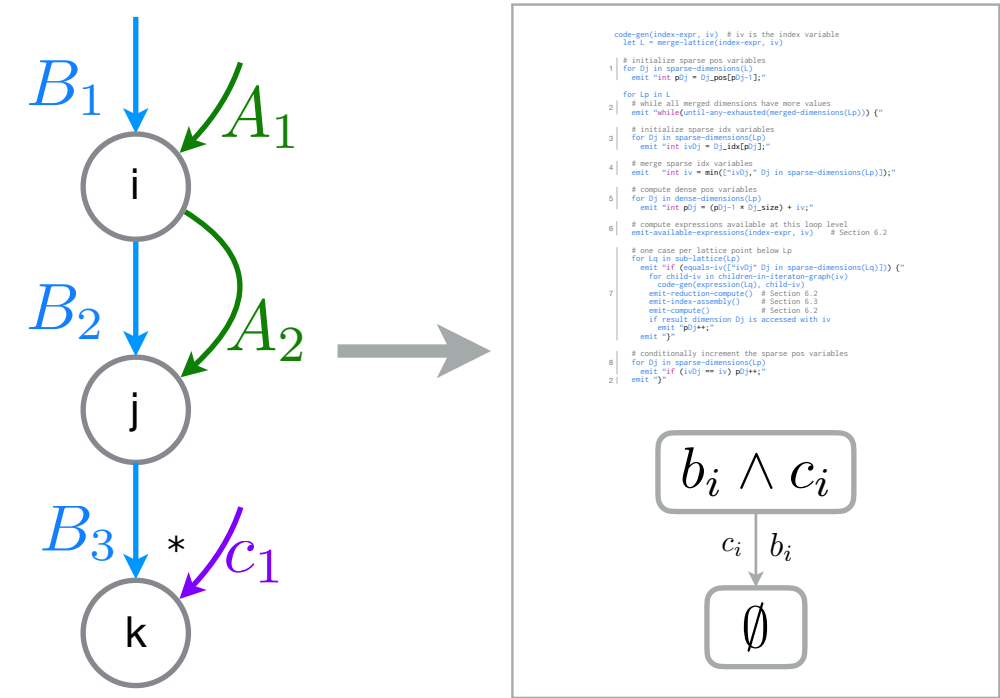
No random access: must traverse level by level to locate coordinate



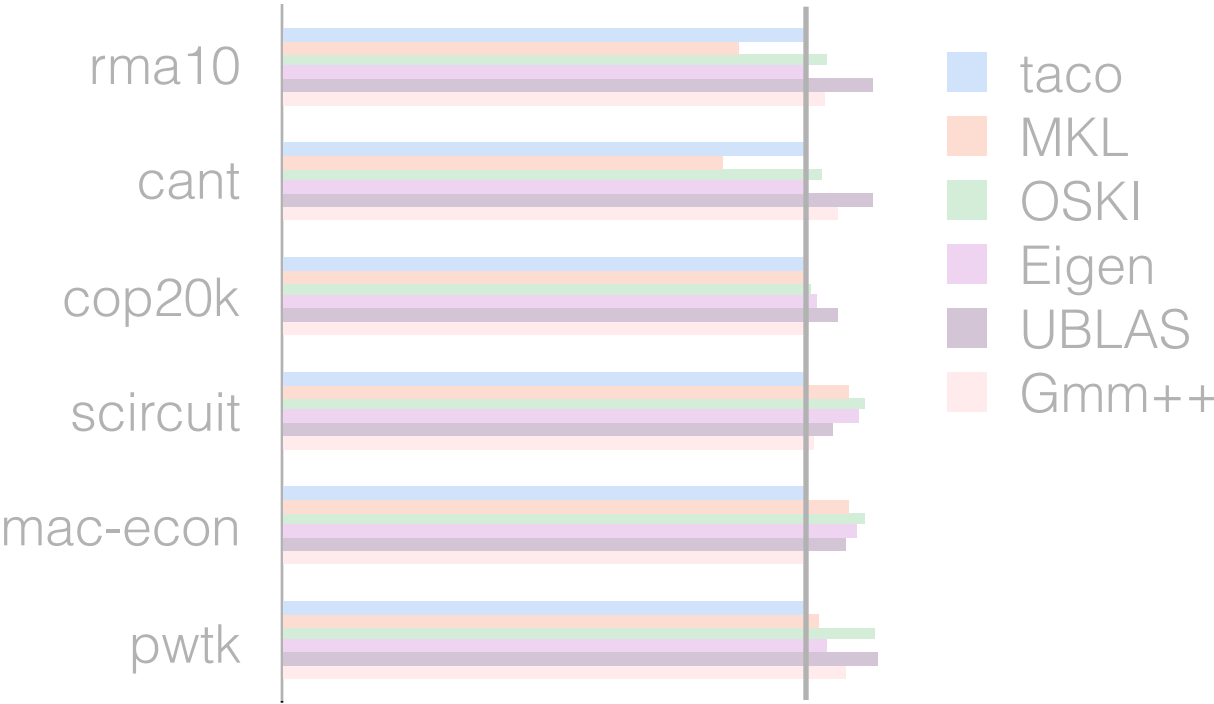
Sparse code and data



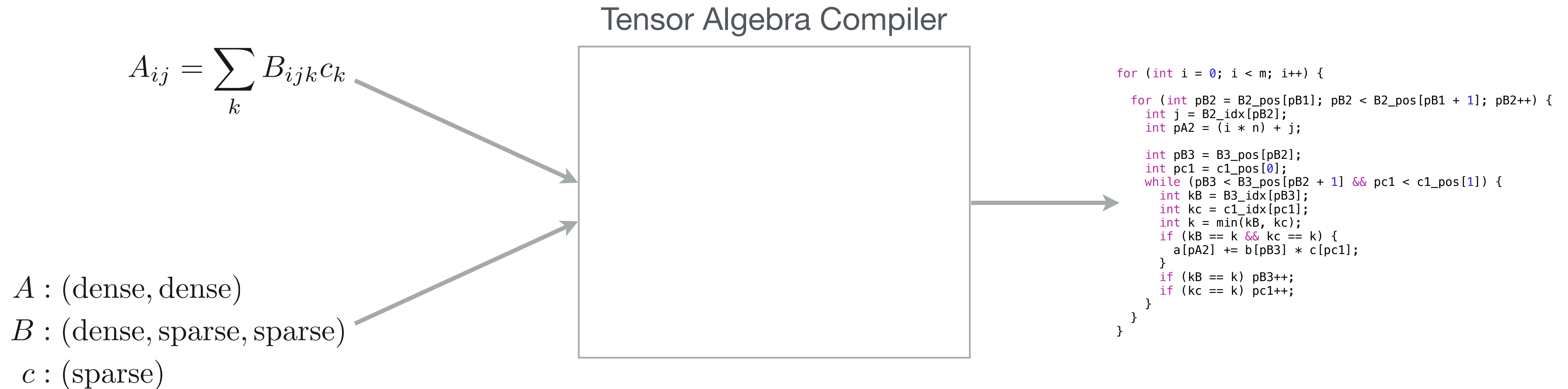
Intermediate Representation and Code Generation



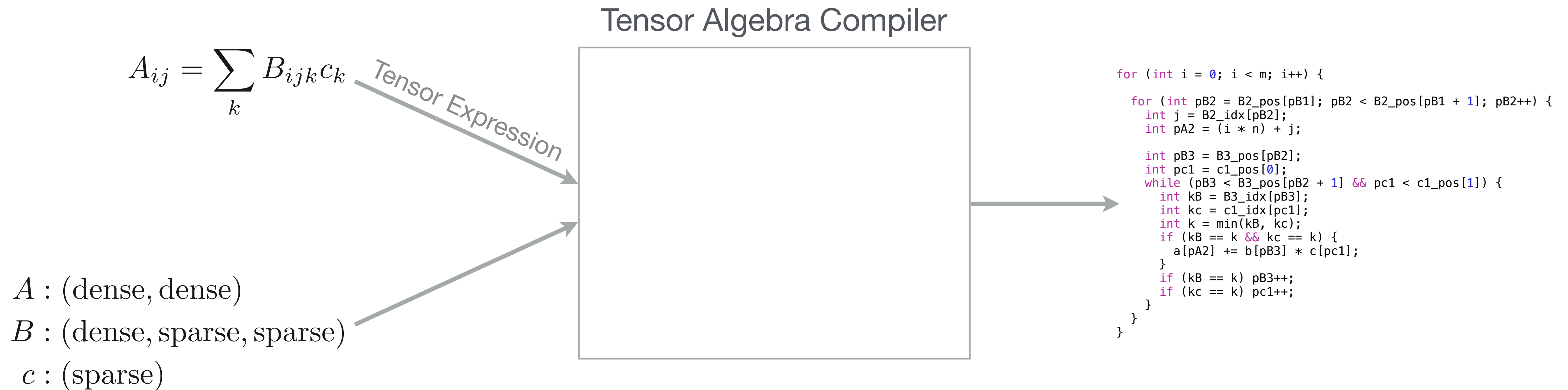
Evaluation



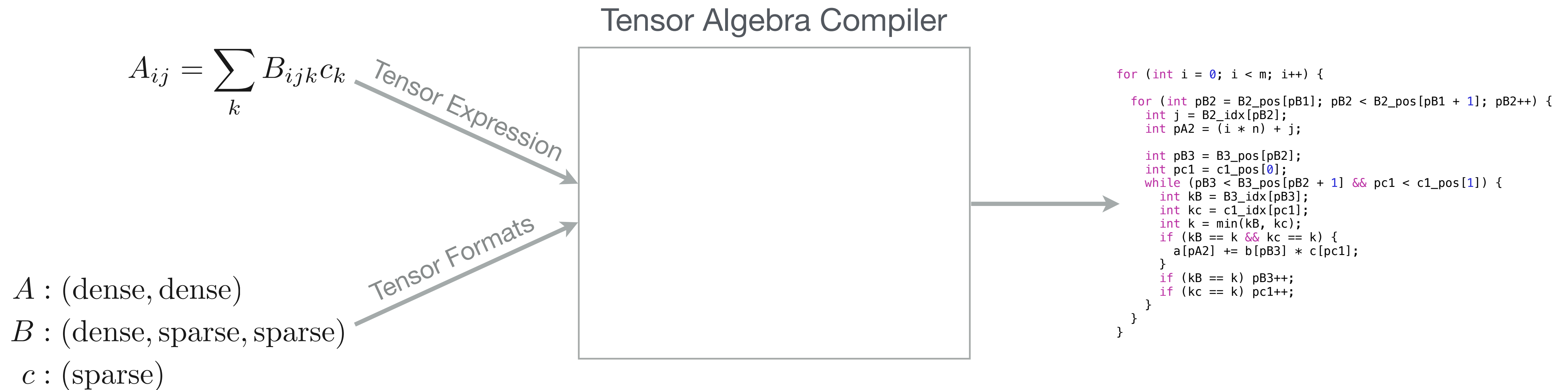
The Tensor Algebra Compiler (taco)



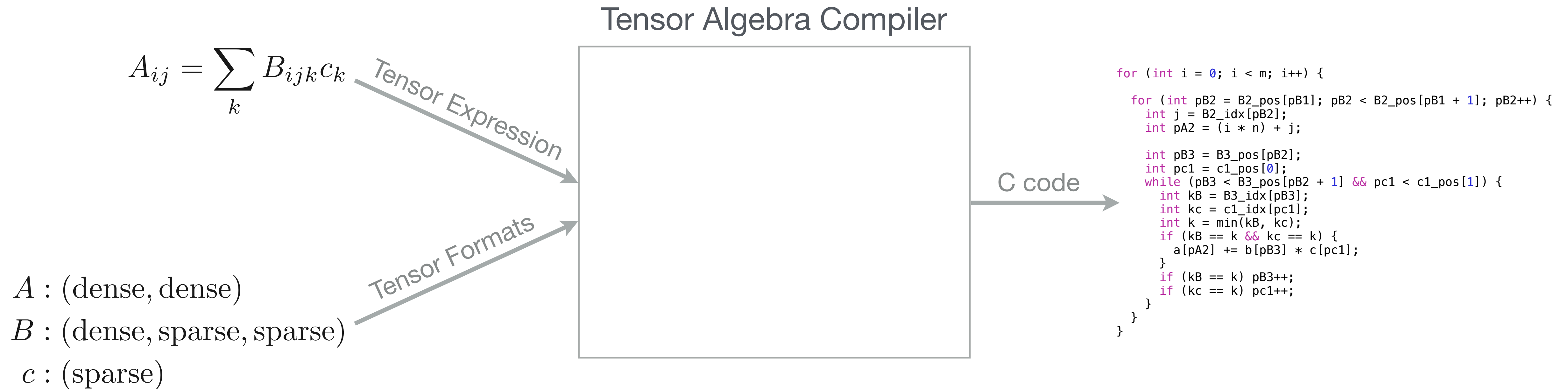
The Tensor Algebra Compiler (taco)



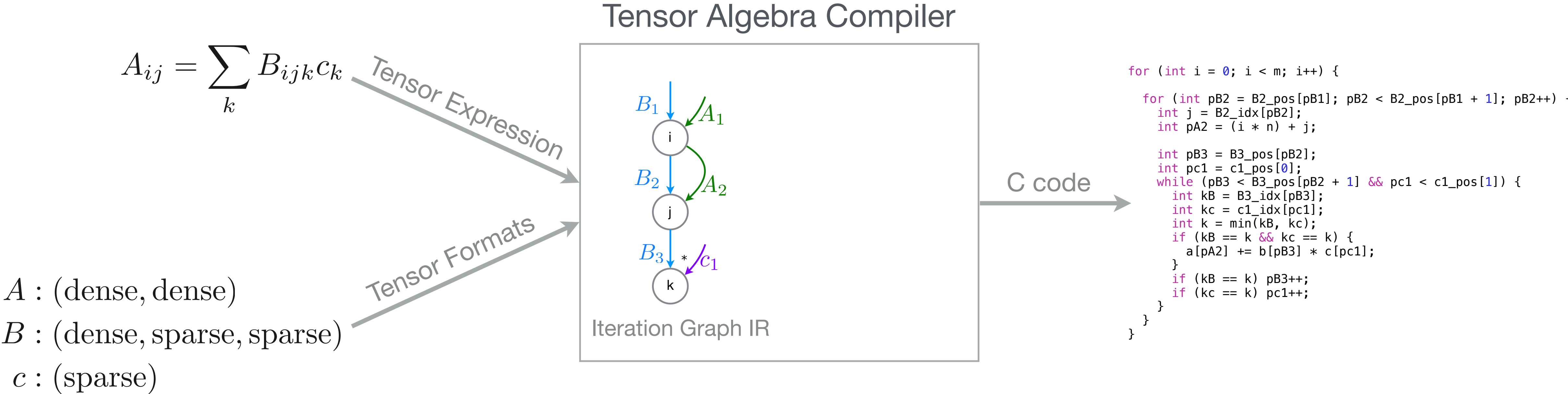
The Tensor Algebra Compiler (taco)



The Tensor Algebra Compiler (taco)



The Tensor Algebra Compiler (taco)



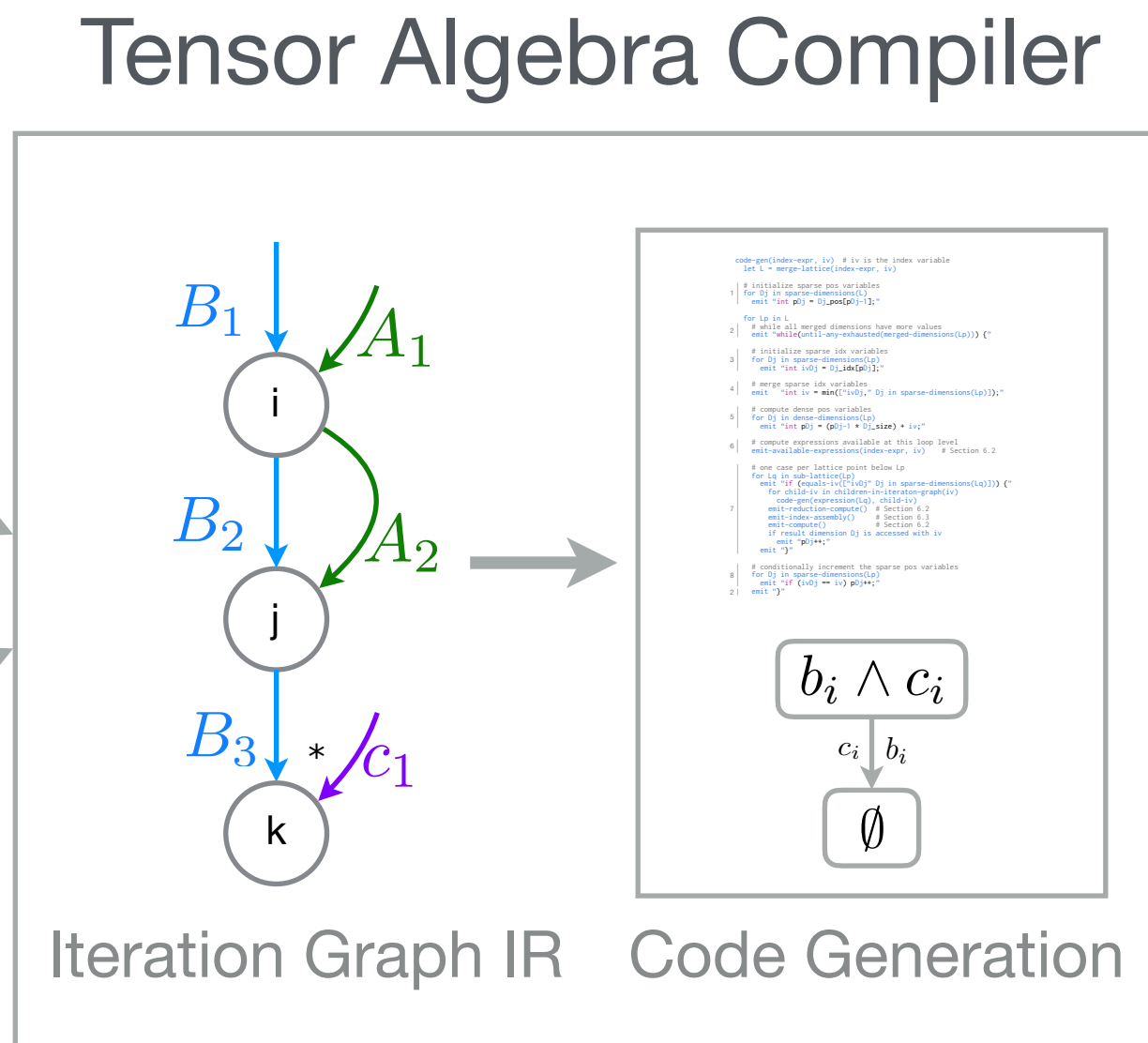
The Tensor Algebra Compiler (taco)

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

$$A : (\text{dense}, \text{dense})$$
$$B : (\text{dense}, \text{sparse}, \text{sparse})$$
$$c : (\text{sparse})$$

Tensor Expression

Tensor Formats



C code

```
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++;  
        }  
    }  
}
```

The Tensor Algebra Compiler (taco)

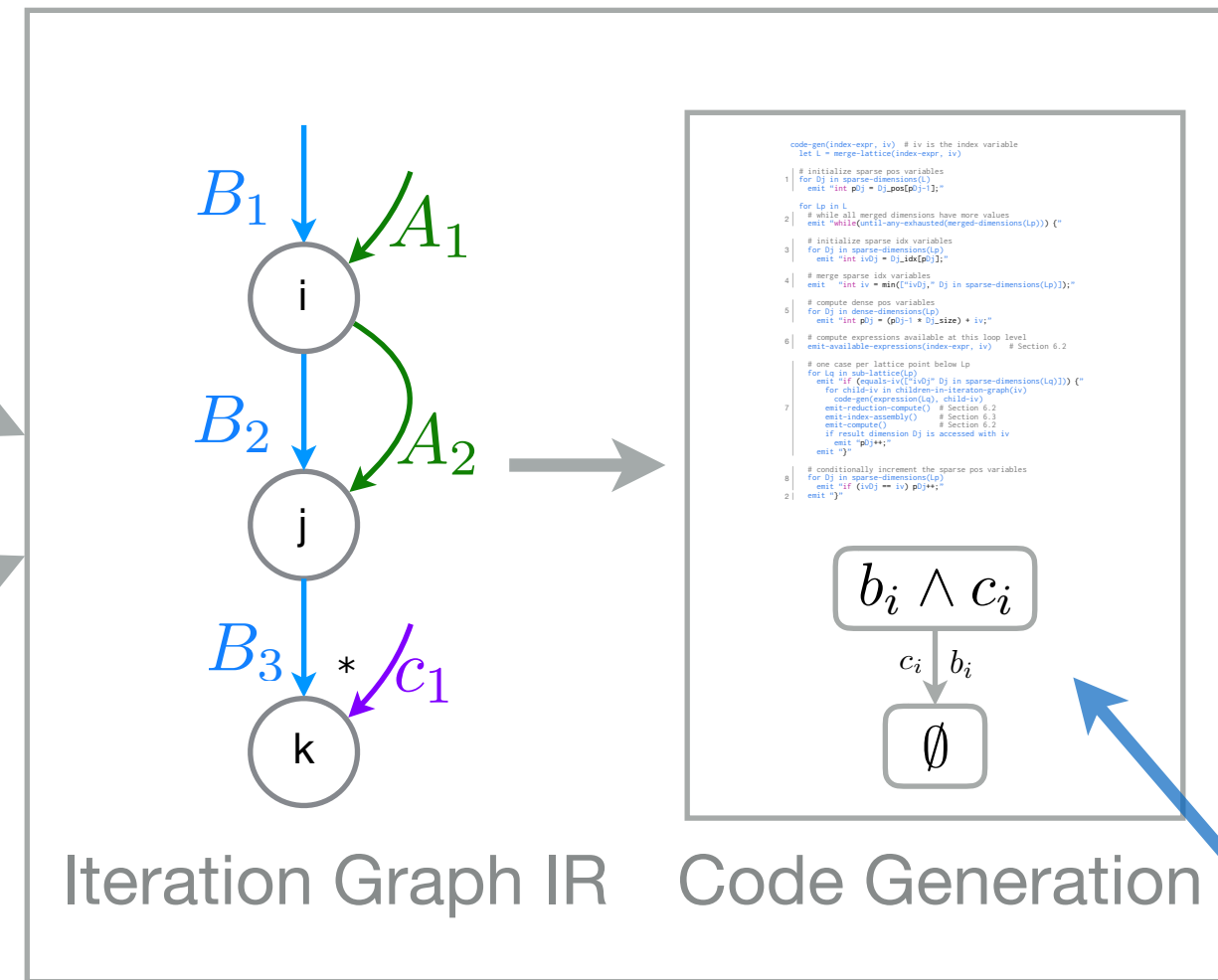
Tensor Algebra Compiler

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

Tensor Expression

$$A : (\text{dense}, \text{dense})$$
$$B : (\text{dense}, \text{sparse}, \text{sparse})$$
$$c : (\text{sparse})$$

Tensor Formats



C code

Merge Lattices

```
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++;  
        }  
    }  
}
```

Sparse iteration spaces and Iteration Graphs

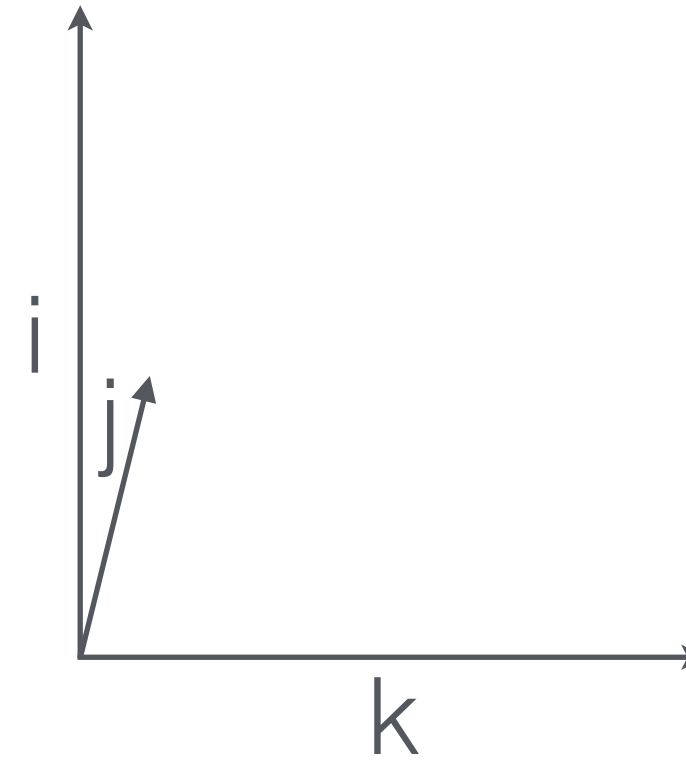
$$A_{ij} = \sum_k B_{ijk} * c_k$$

Sparse iteration spaces and Iteration Graphs

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

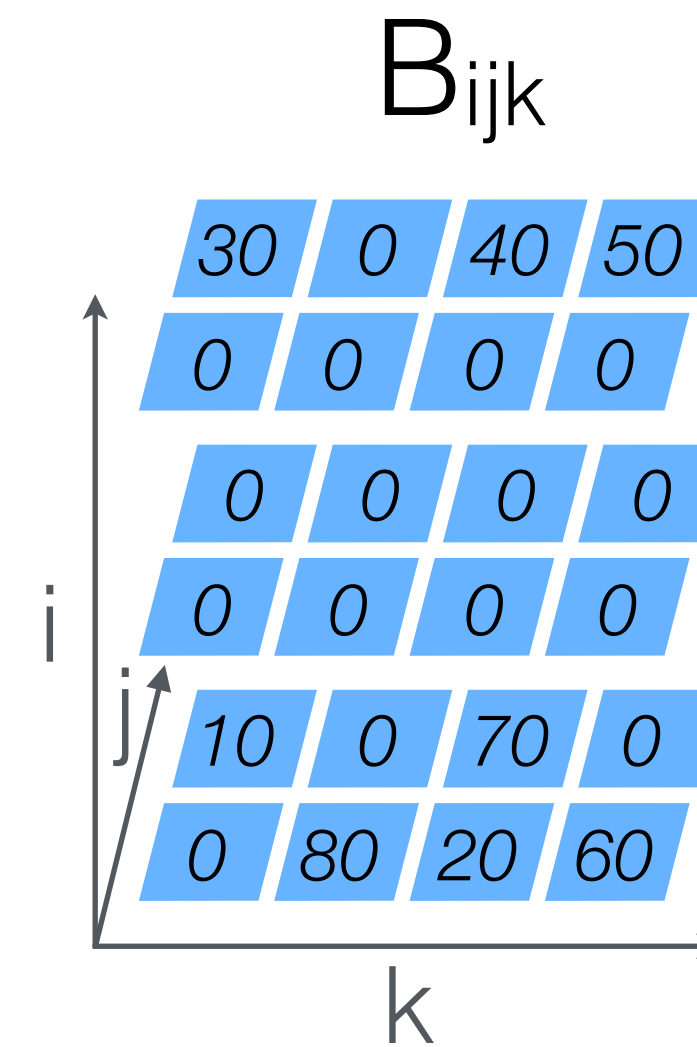

Sparse iteration spaces and Iteration Graphs

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



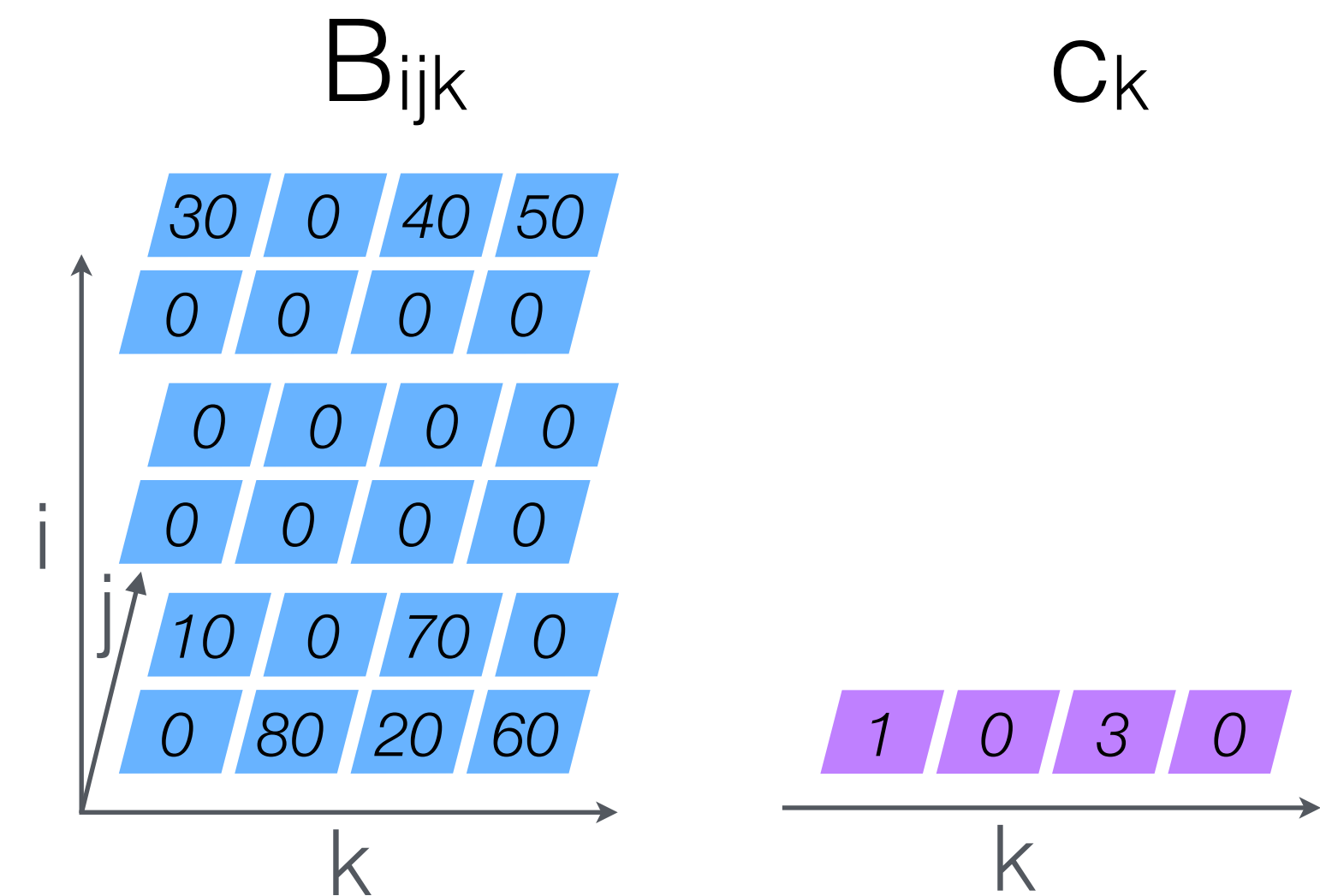
Sparse iteration spaces and Iteration Graphs

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



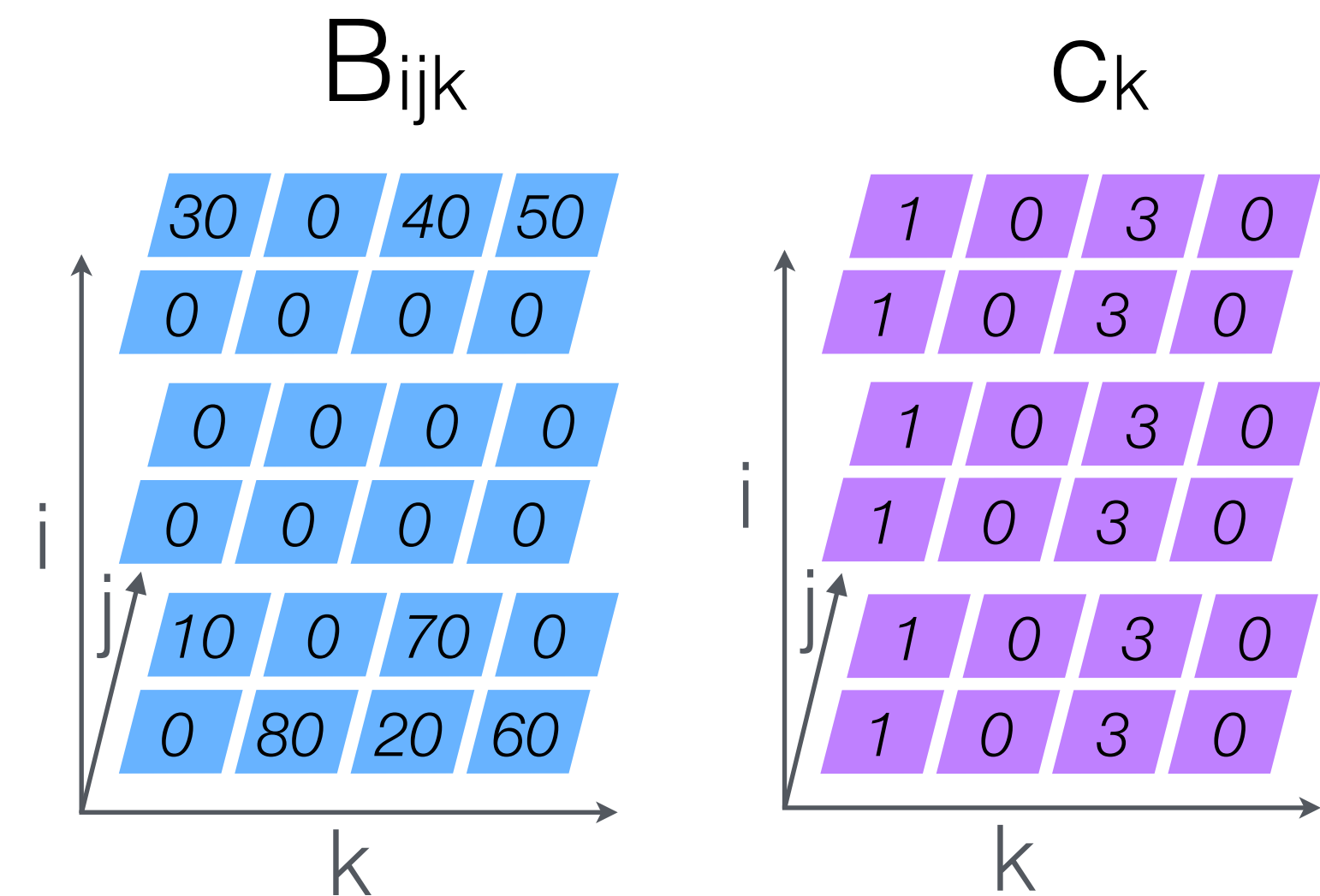
Sparse iteration spaces and Iteration Graphs

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



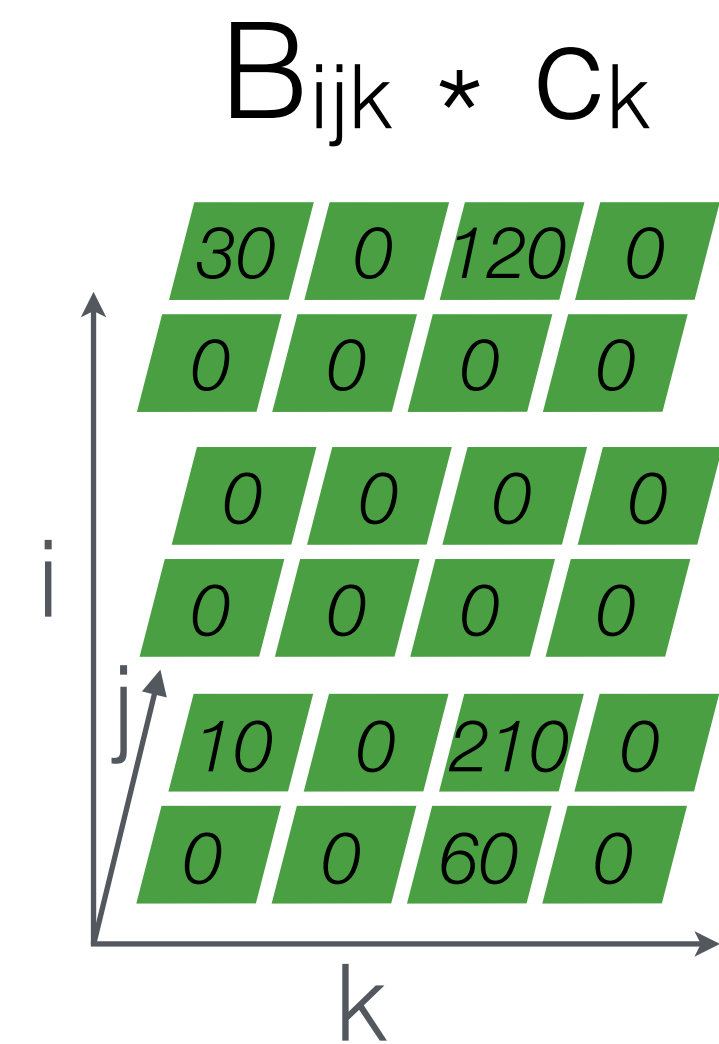
Sparse iteration spaces and Iteration Graphs

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



Sparse iteration spaces and Iteration Graphs

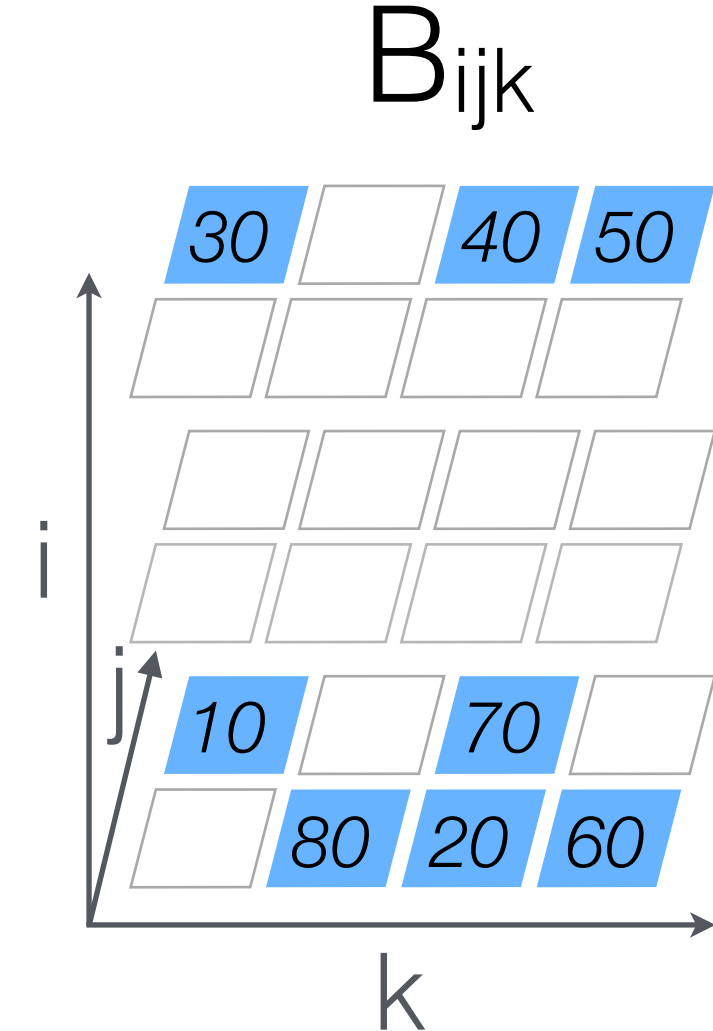
$$A_{ij} = \sum_k B_{ijk} * c_k$$



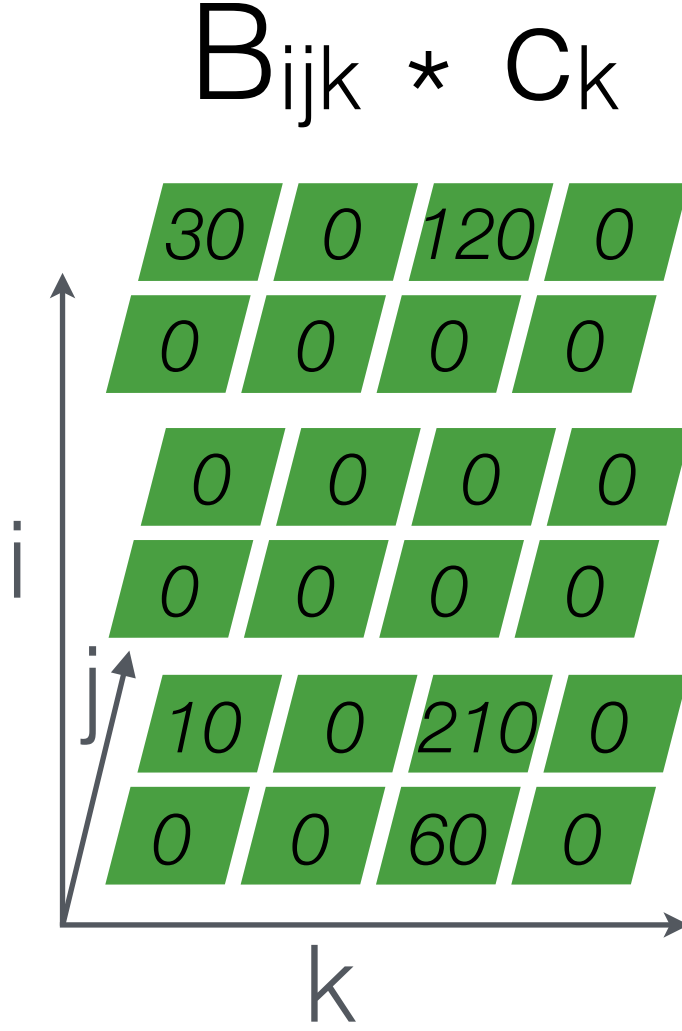
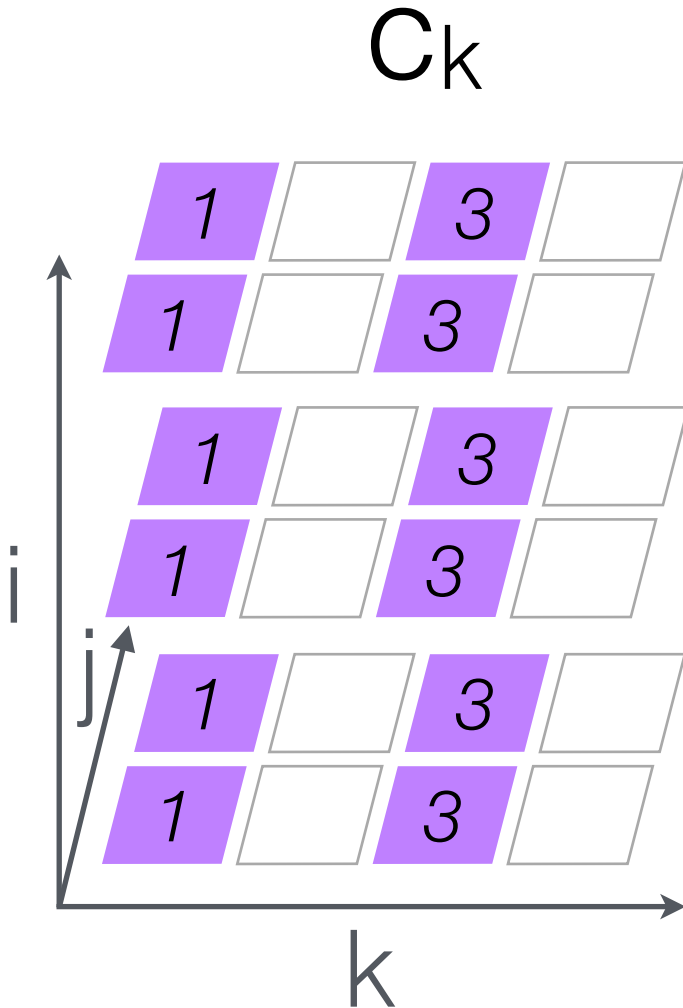
Dense

Sparse iteration spaces and Iteration Graphs

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



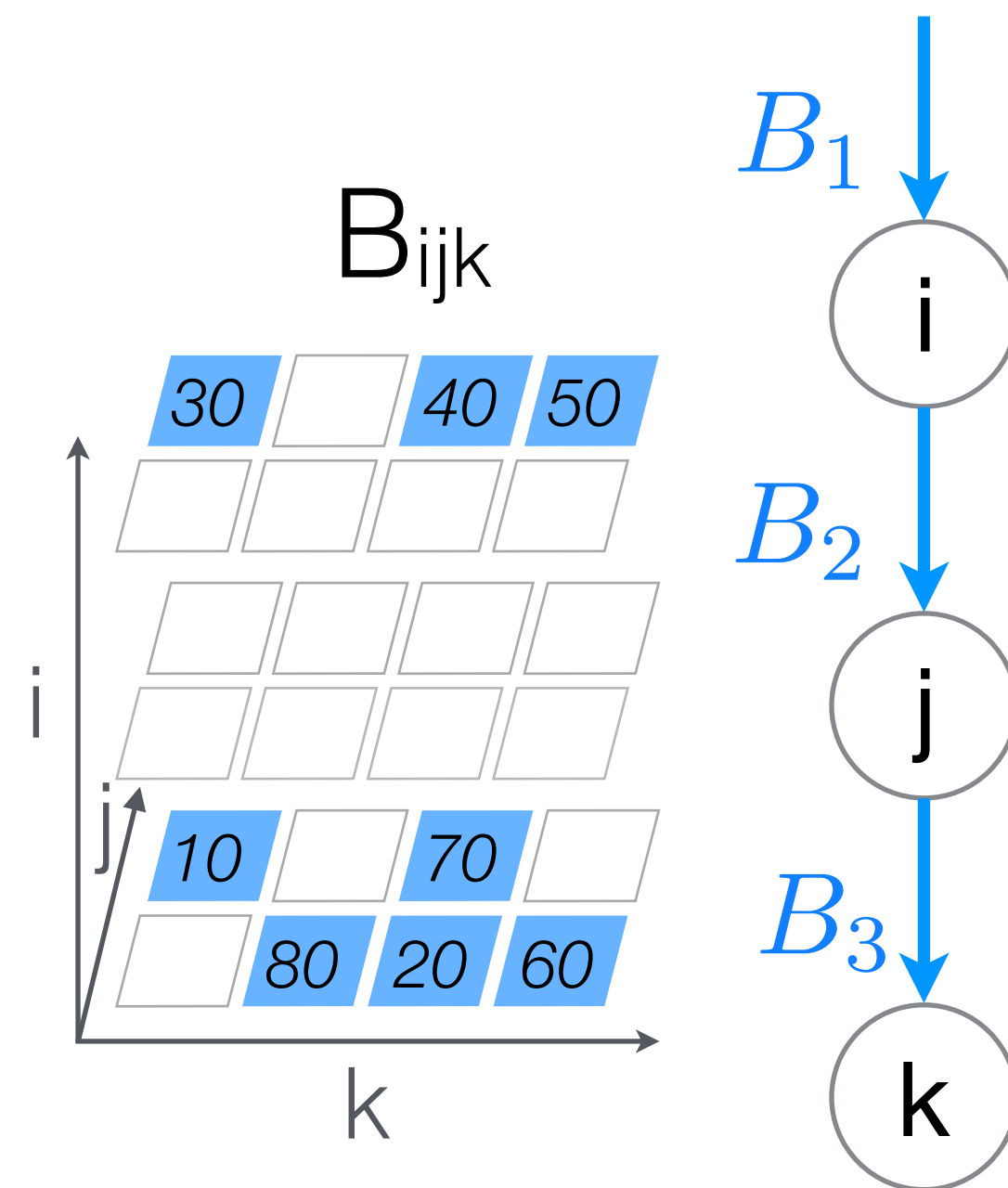
Sparse



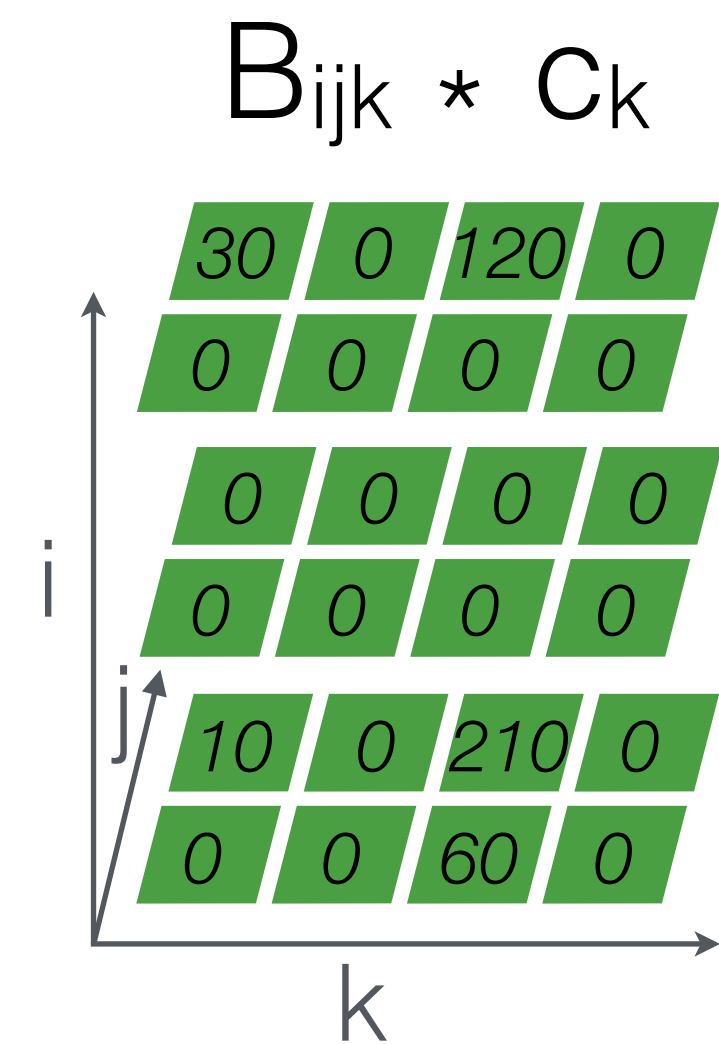
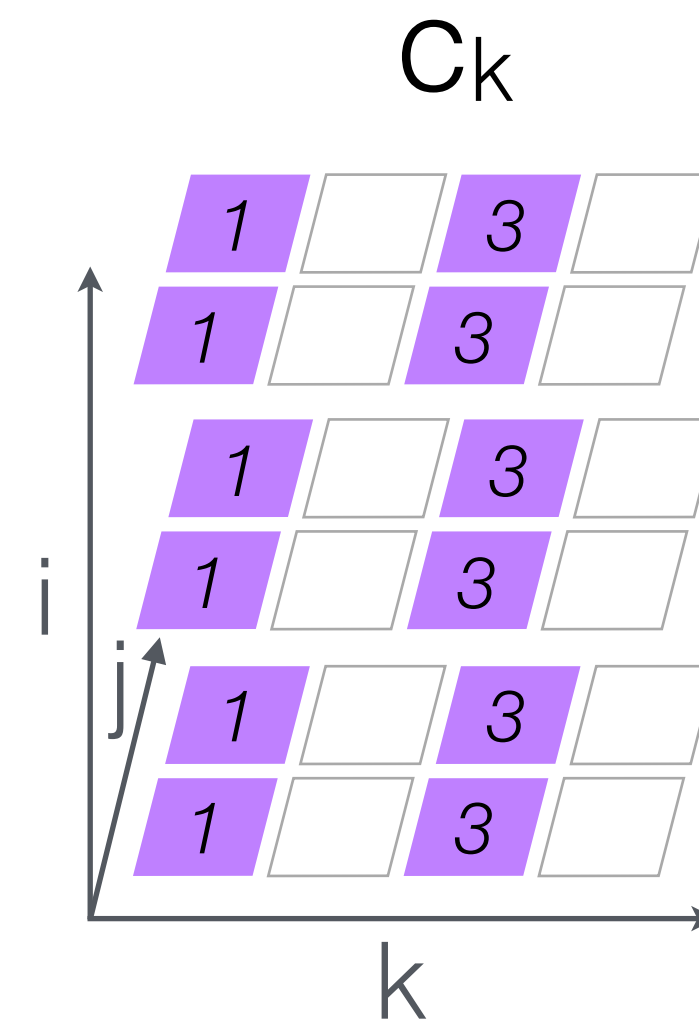
Dense

Sparse iteration spaces and Iteration Graphs

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



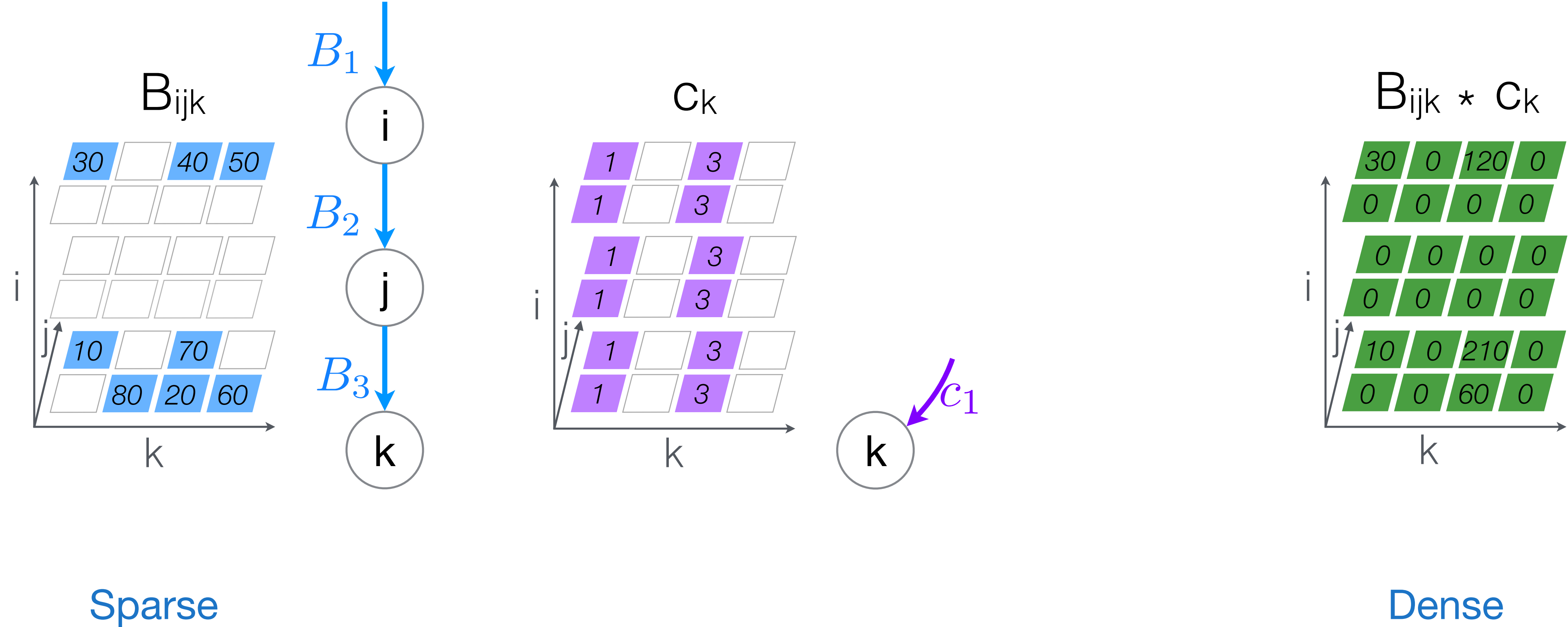
Sparse



Dense

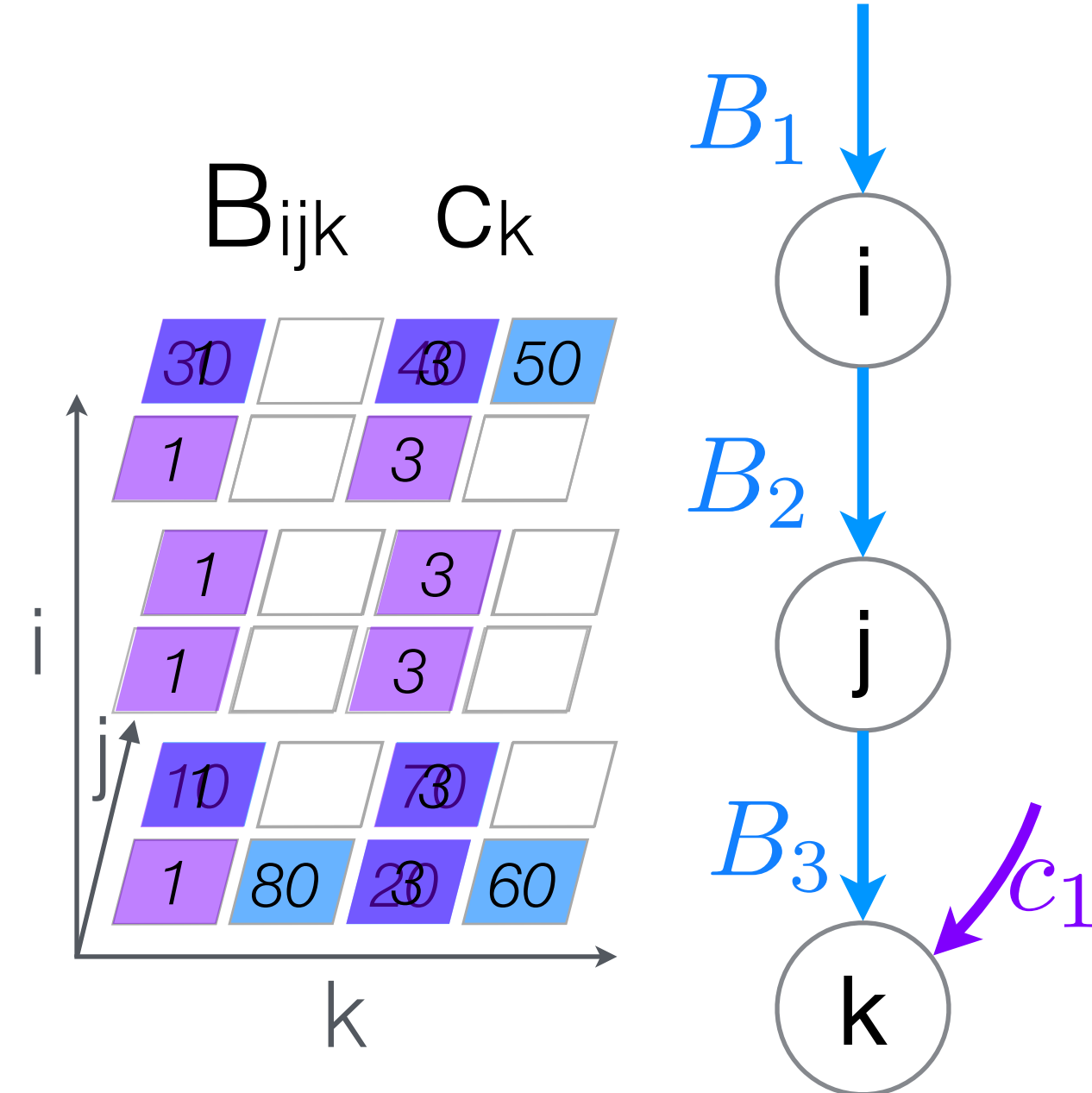
Sparse iteration spaces and Iteration Graphs

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

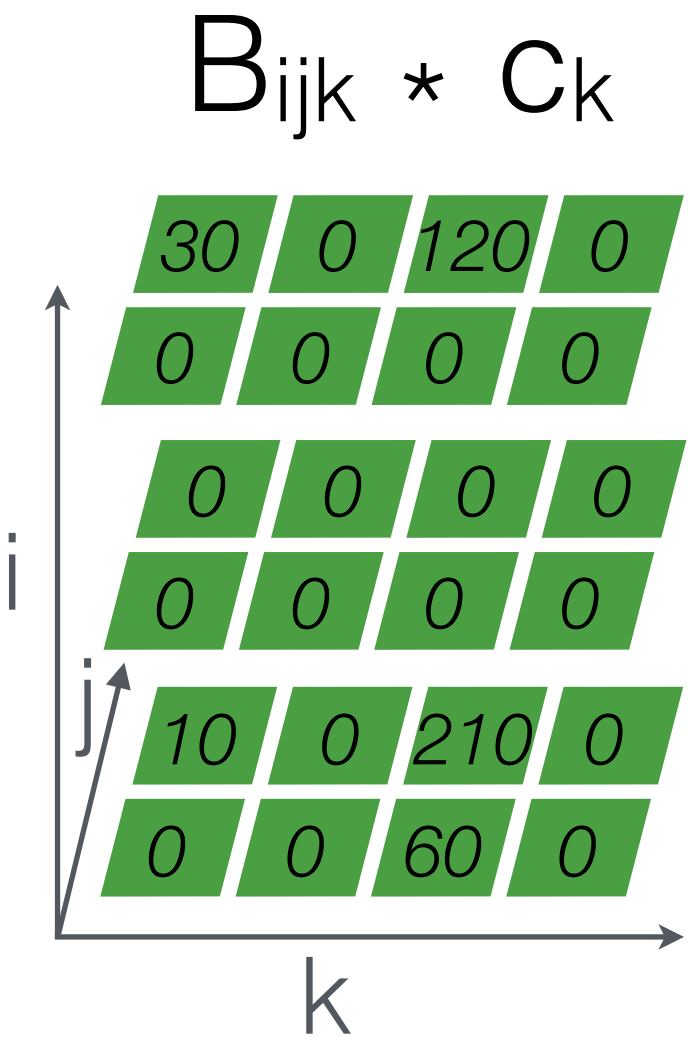


Sparse iteration spaces and Iteration Graphs

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



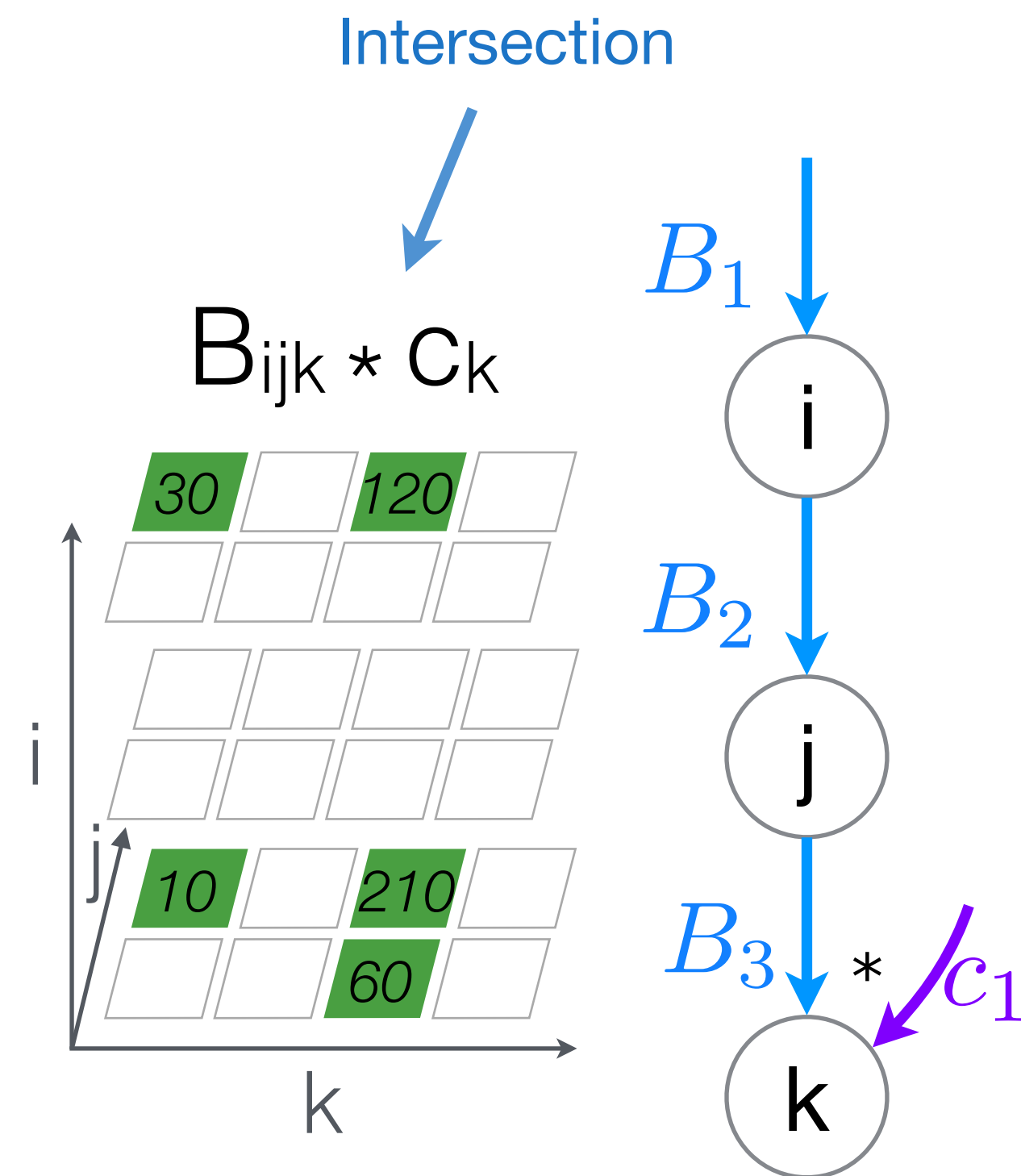
Sparse



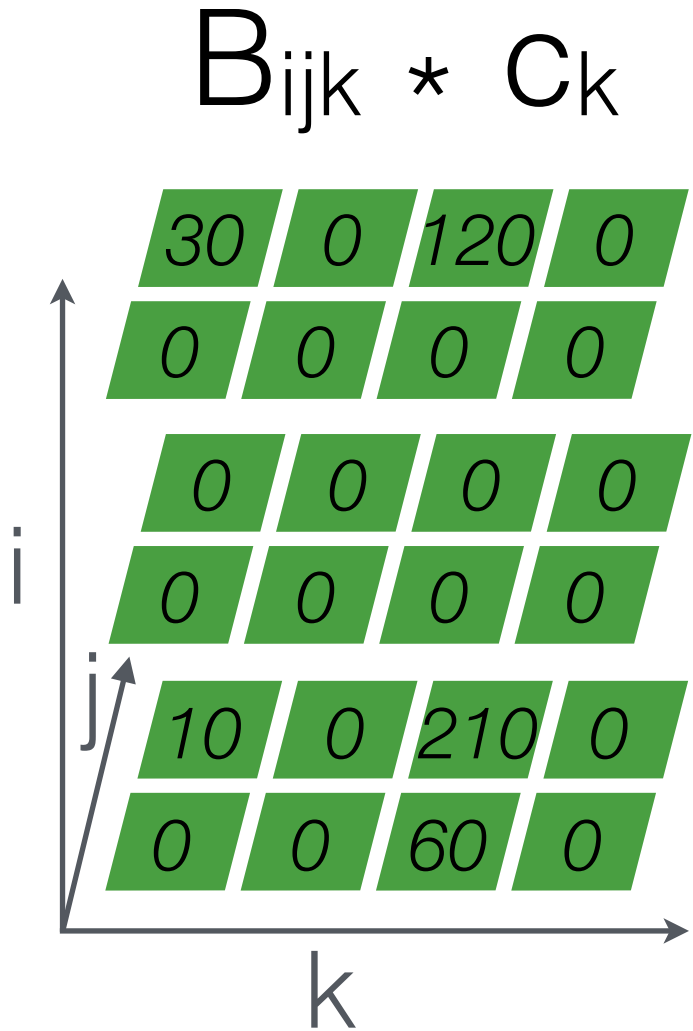
Dense

Sparse iteration spaces and Iteration Graphs

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



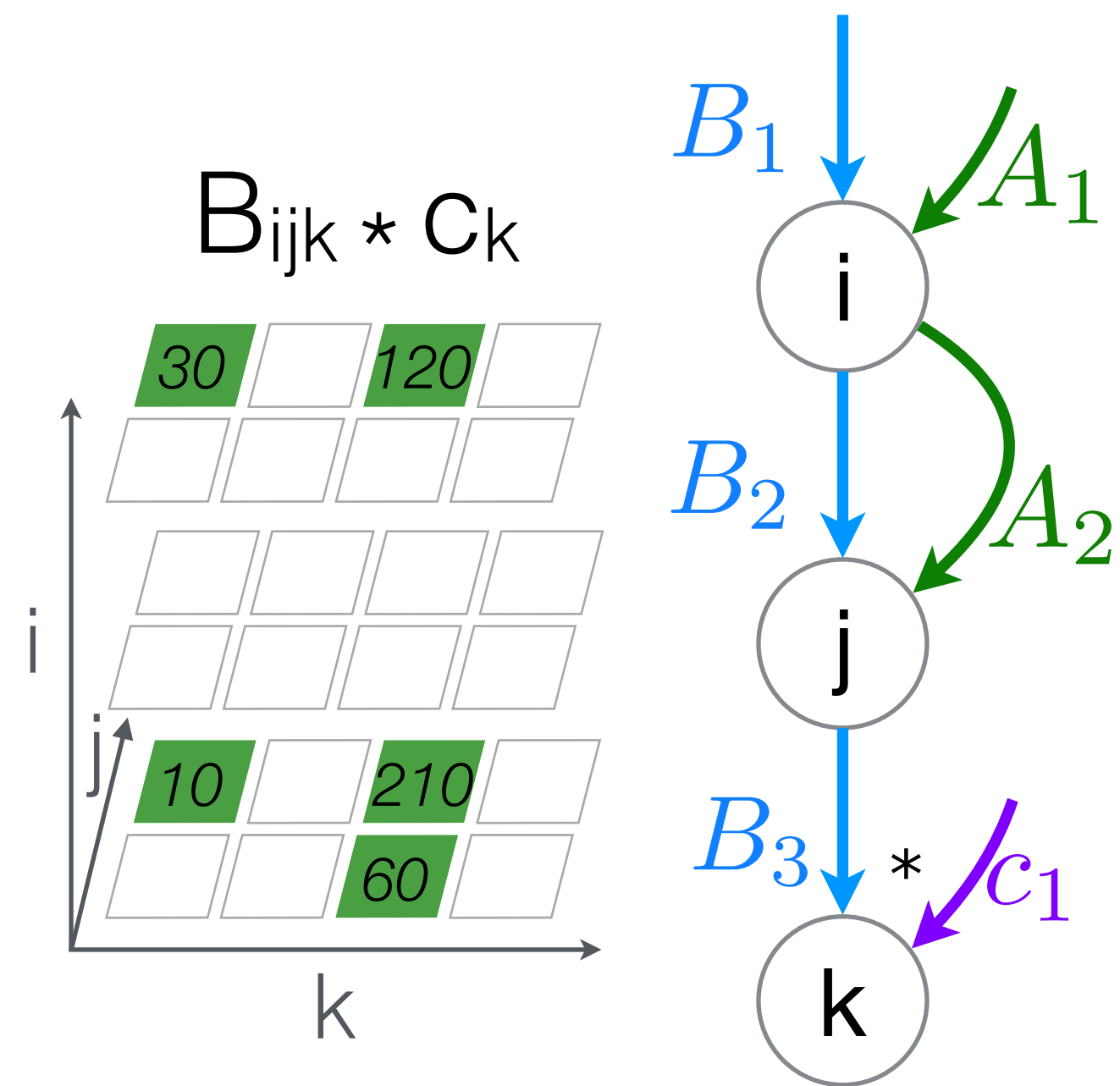
Sparse



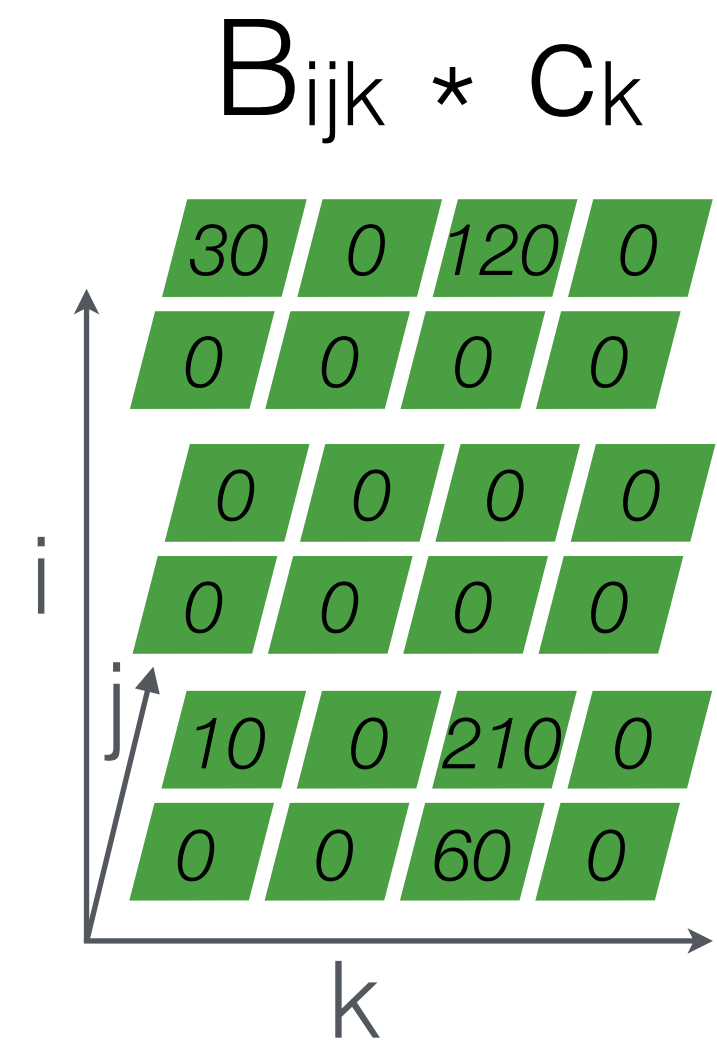
Dense

Sparse iteration spaces and Iteration Graphs

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



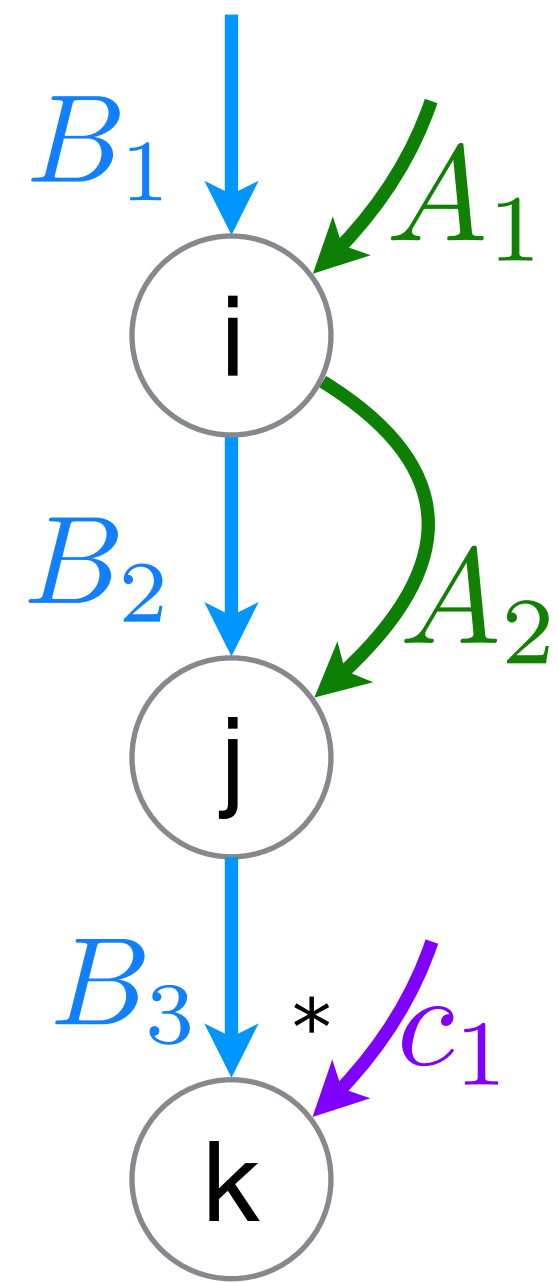
Sparse



Dense

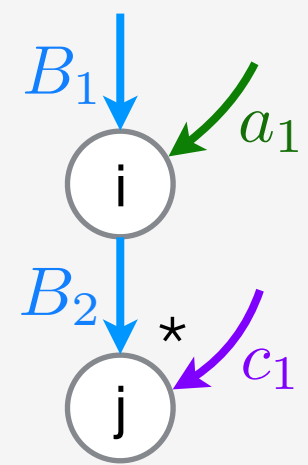
Examples of Iteration Graph

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

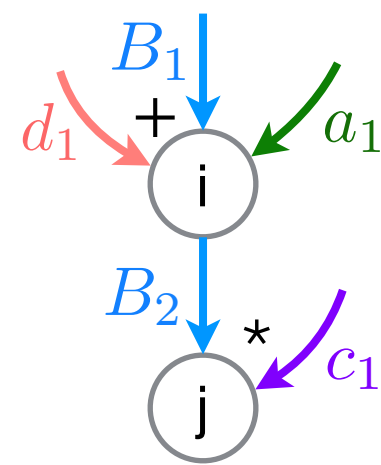


Examples of Iteration Graph

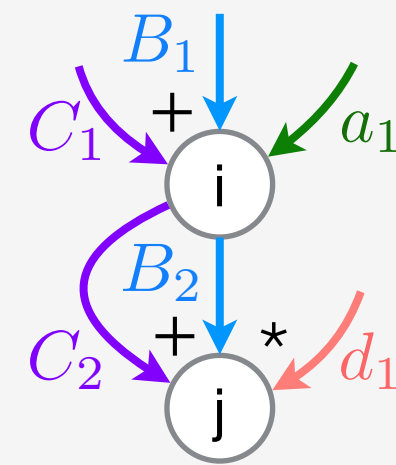
$$a_i = \sum_j B_{ij} c_j$$



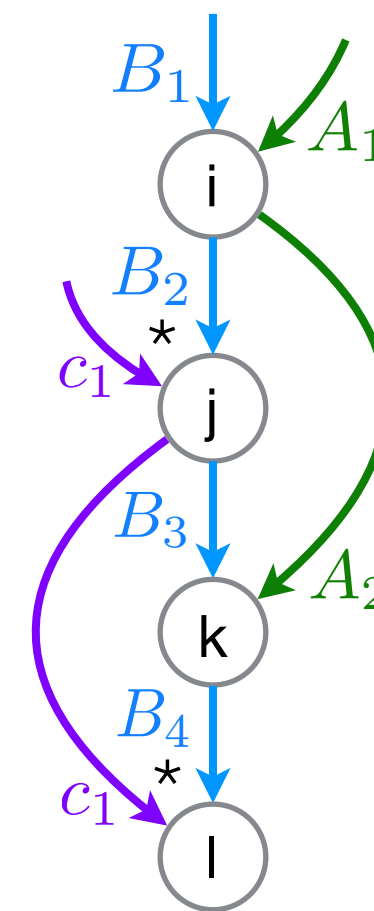
$$a_i = \sum_j \alpha B_{ij} c_j + \beta d_i$$



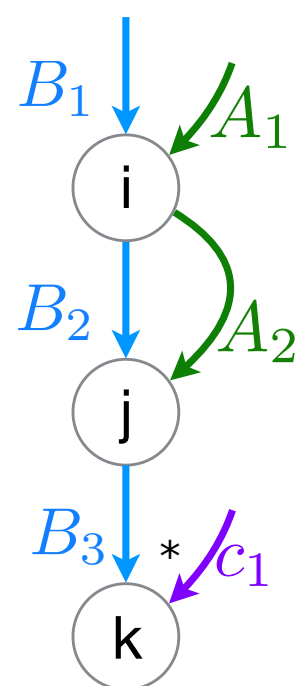
$$a_i = \sum_j (B_{ij} + C_{ij}) d_j$$



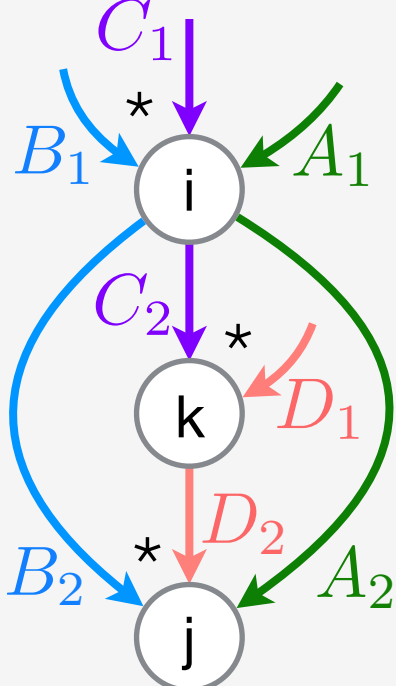
$$a_{ik} = \sum_j \sum_l B_{ijkl} c_{jl}$$



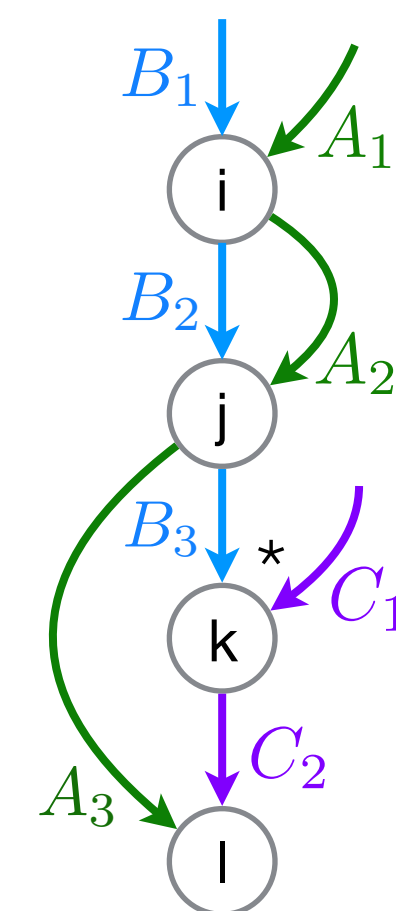
$$A_{ij} = \sum_k B_{ijk} * c_k$$



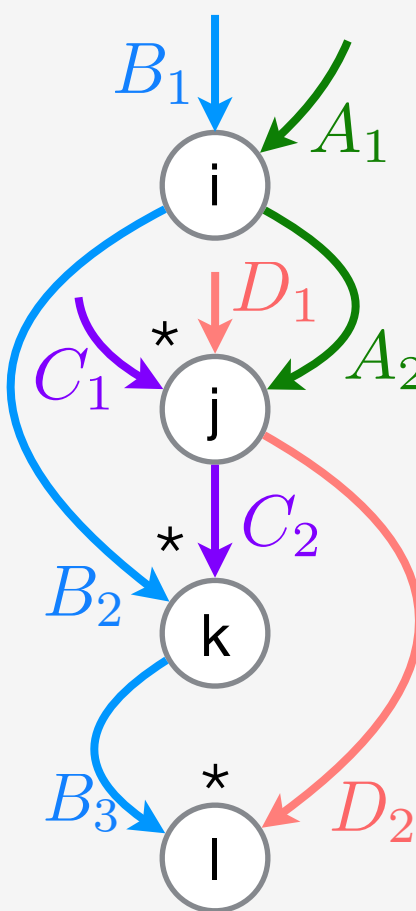
$$A_{ij} = \sum_k B_{ij} (C_{ik} D_{kj})$$



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

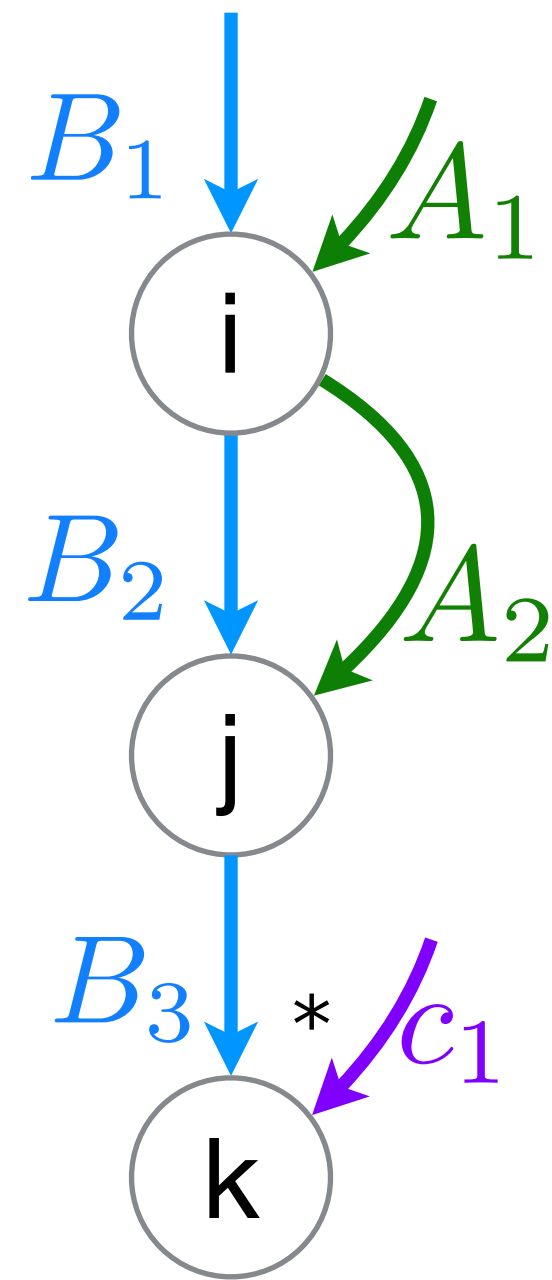


$$A_{ij} = \sum_k \sum_l B_{ikl} C_{kj} D_{lj}$$



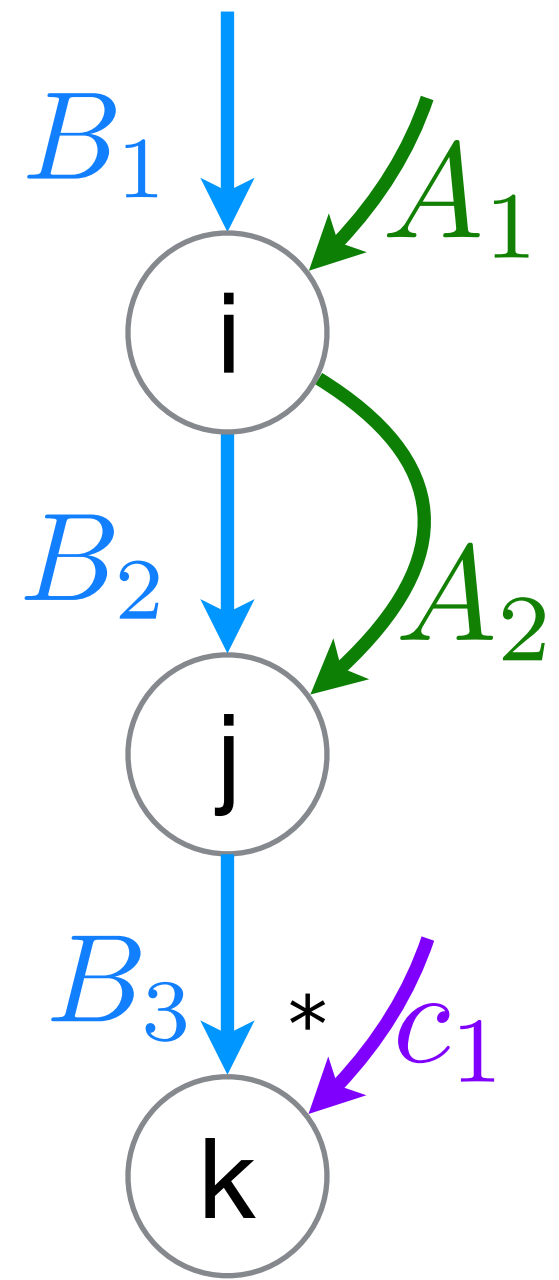
Examples of Iteration Graph

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



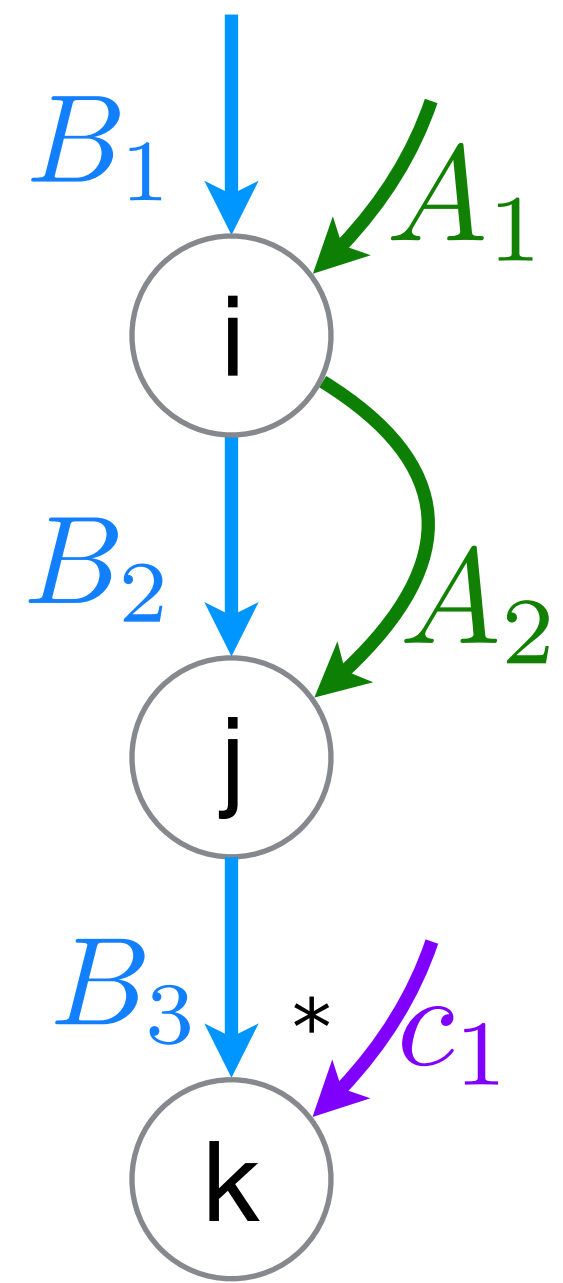
Code Generation from Iteration Graphs

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



Code Generation from Iteration Graphs

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



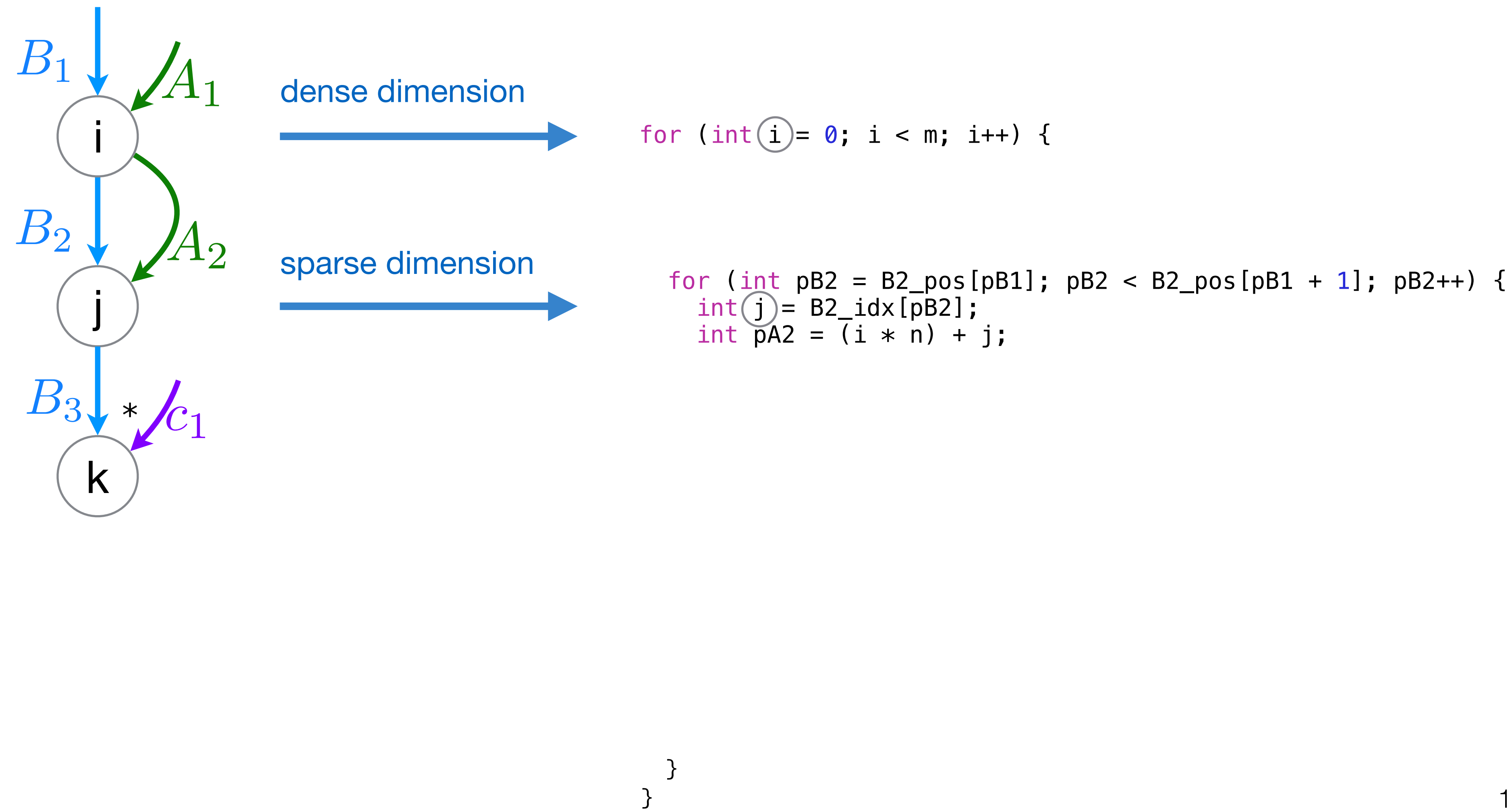
dense dimension

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

```
}
```

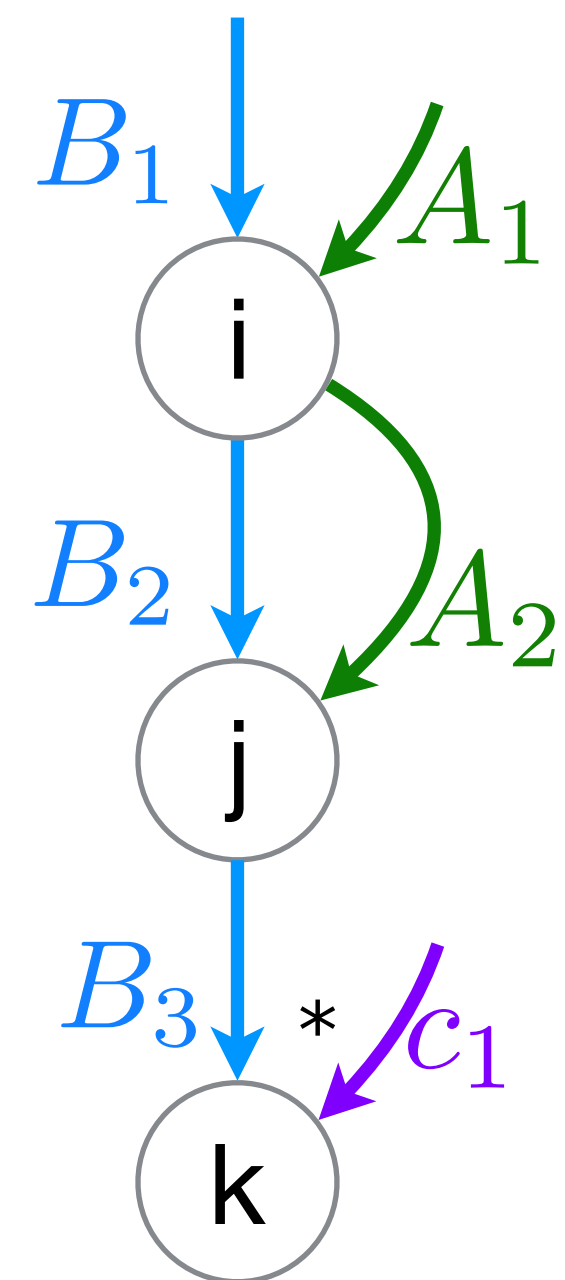
Code Generation from Iteration Graphs

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



Code Generation from Iteration Graphs

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



dense dimension

sparse dimension

merged dimensions

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

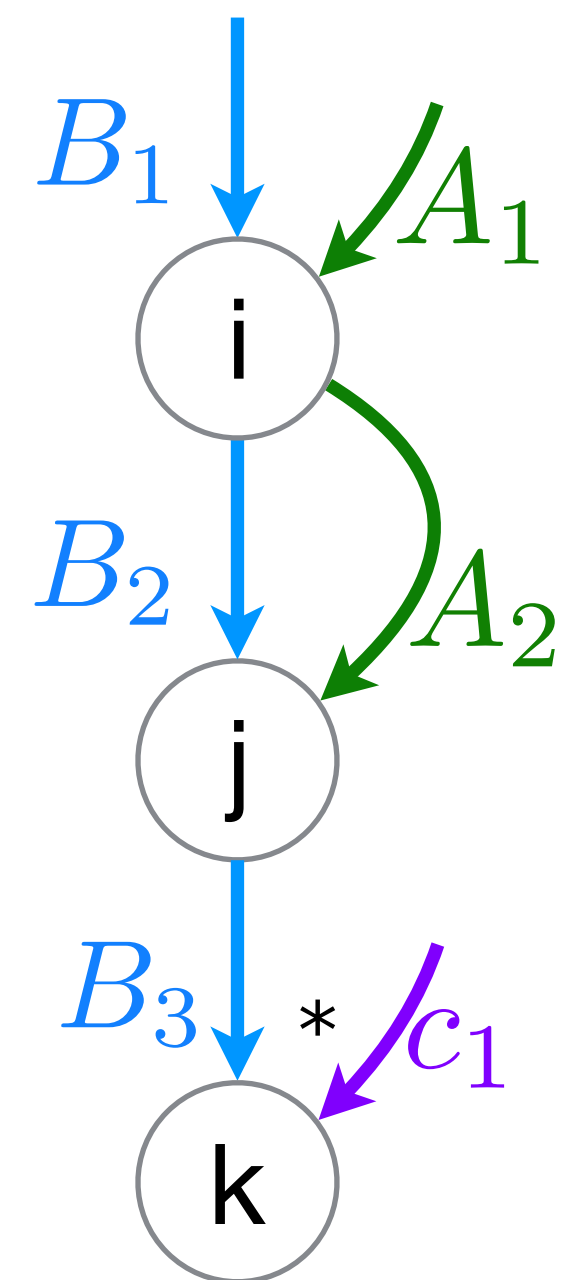
```
    for (int pB2 = B2_pos[pB1]; pB2 < B2_pos[pB1 + 1]; pB2++) {
        int j = B2_idx[pB2];
        int pA2 = (i * n) + j;
```

```
        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

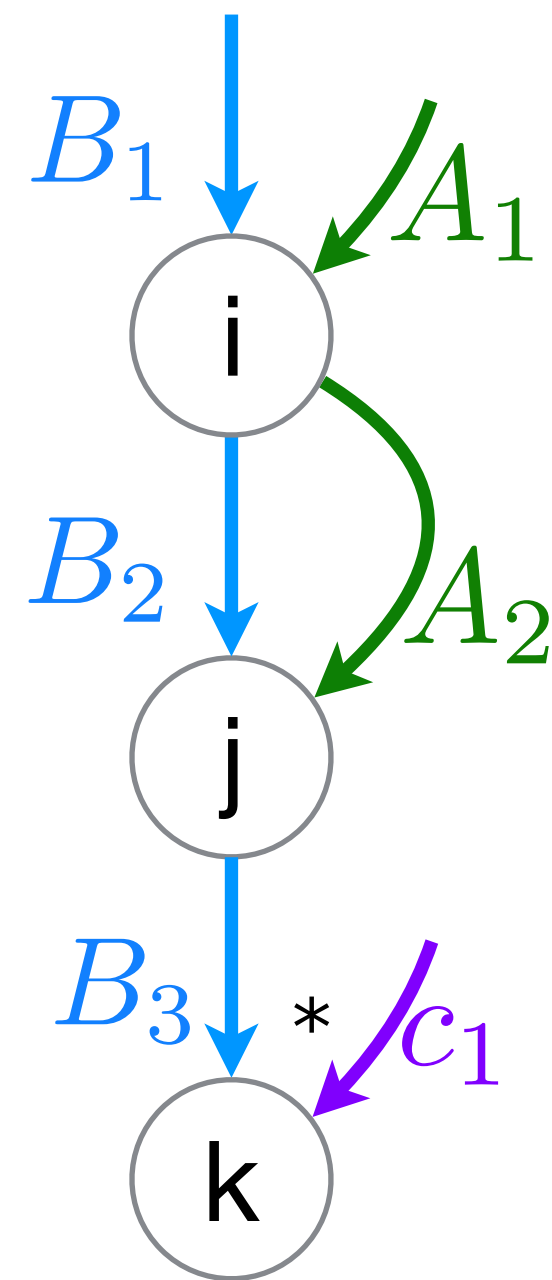
```
for (int i = 0; i < m; i++) {
```

```
    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++) {
```

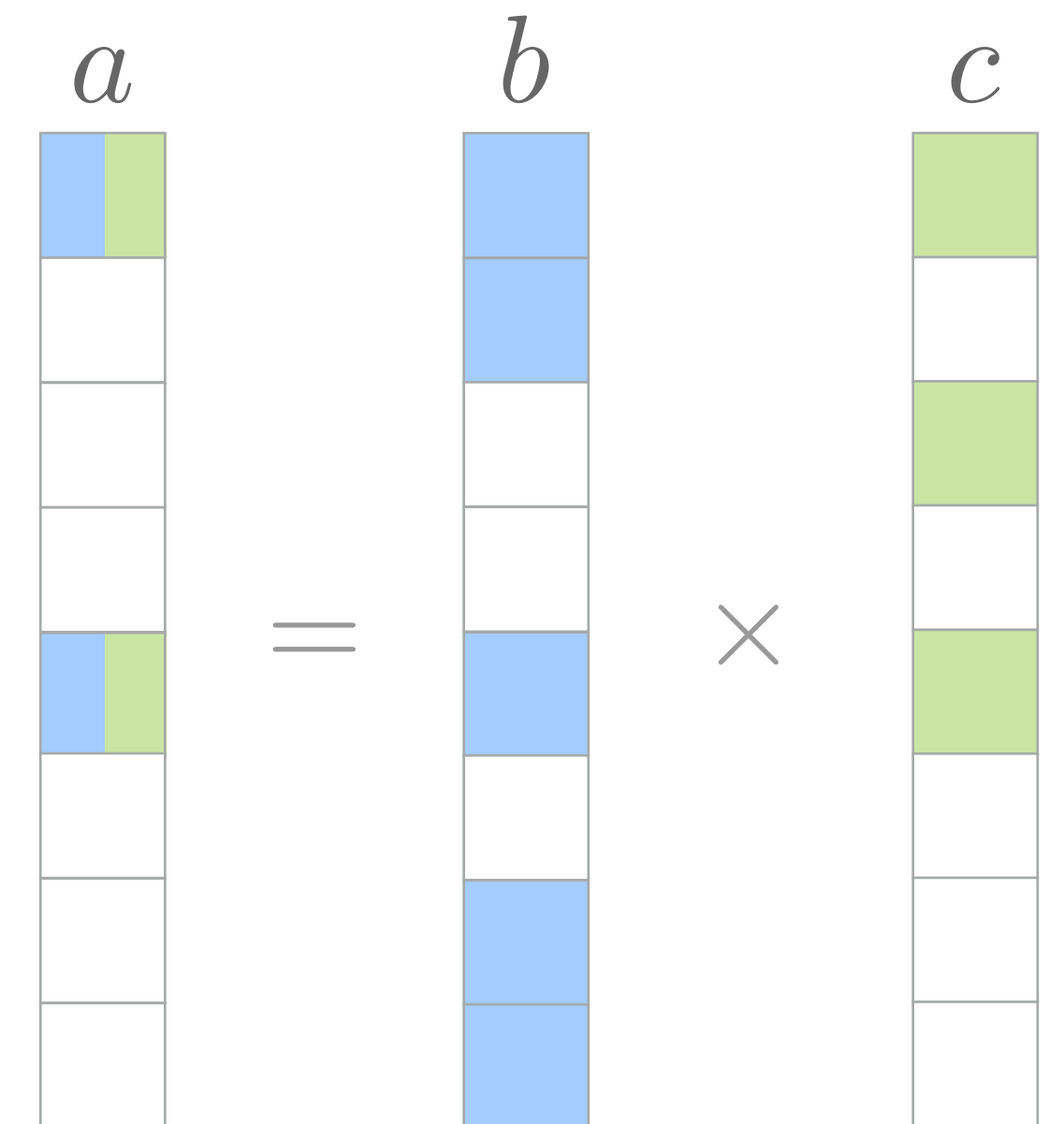
```
    for (int pB2 = B2_pos[pB1]; pB2 < B2_pos[pB1 + 1]; pB2++) {
        int j = B2_idx[pB2];
        int pA2 = (i * n) + j;
```

```
        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++;
        }
    }
```

```
}
```

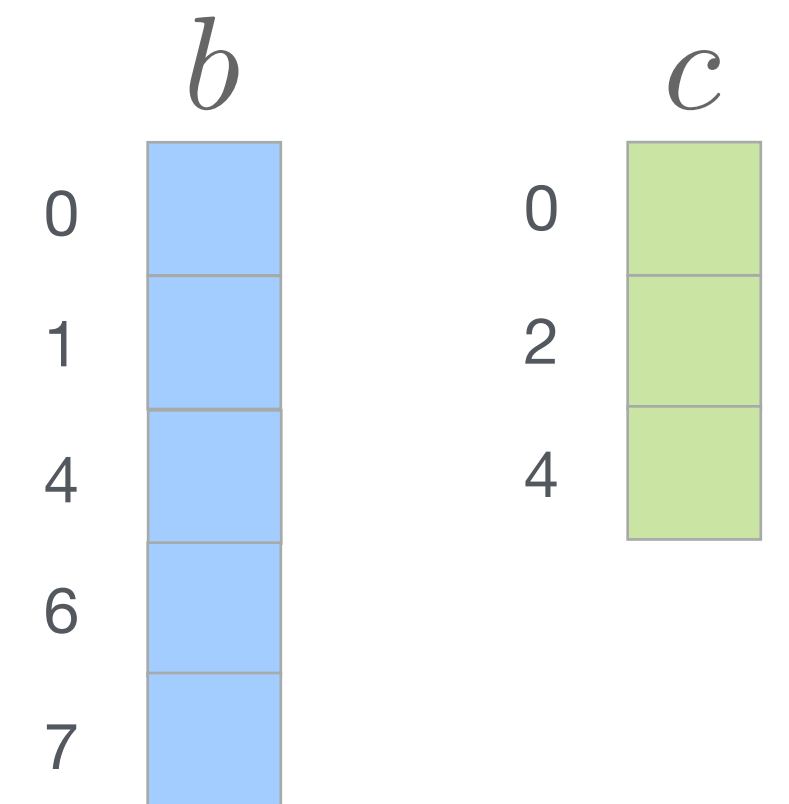
Merge Lattice for a Conjunction

$$a_i = b_i c_i$$



Merge Lattice for a Conjunction

$$a_i = b_i c_i$$



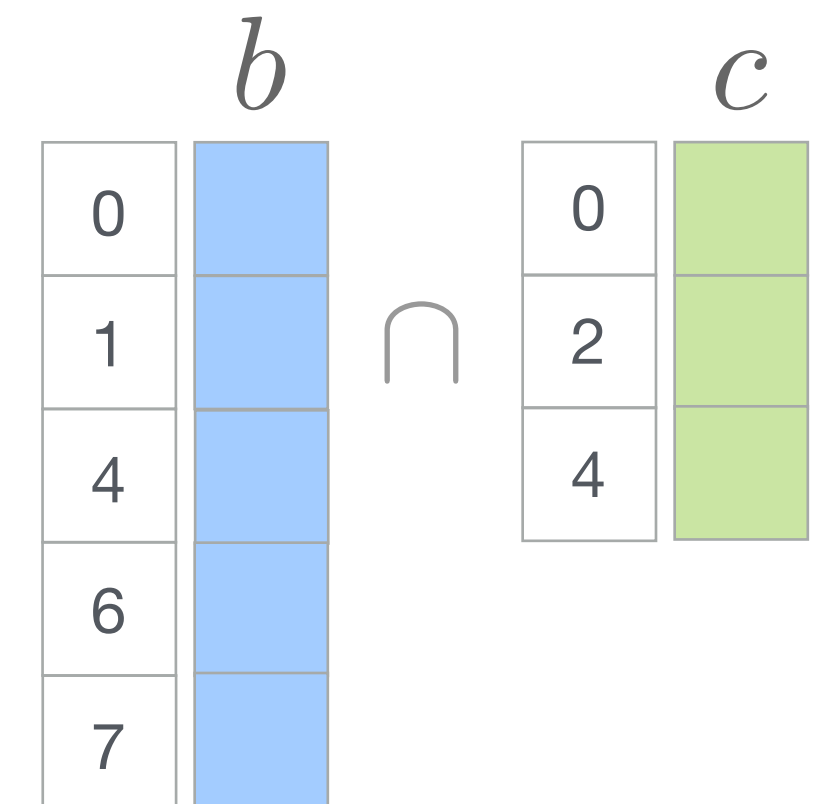
Merge Lattice for a Conjunction

$$a_i = b_i c_i$$

<i>b</i>		<i>c</i>	
0		0	
1		2	
4		4	
6			
7			

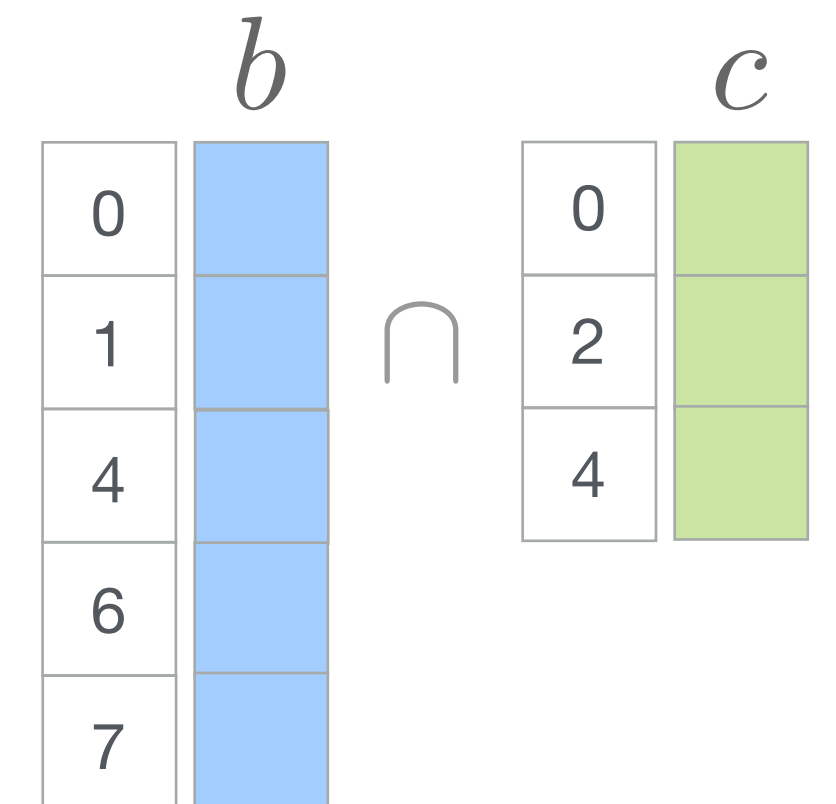
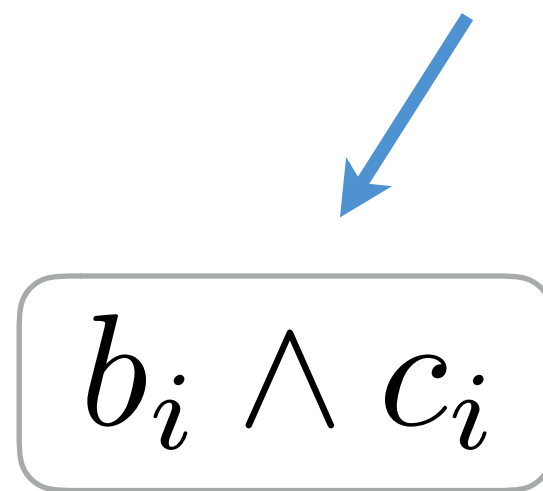
Merge Lattice for a Conjunction

$$a_i = b_i c_i$$



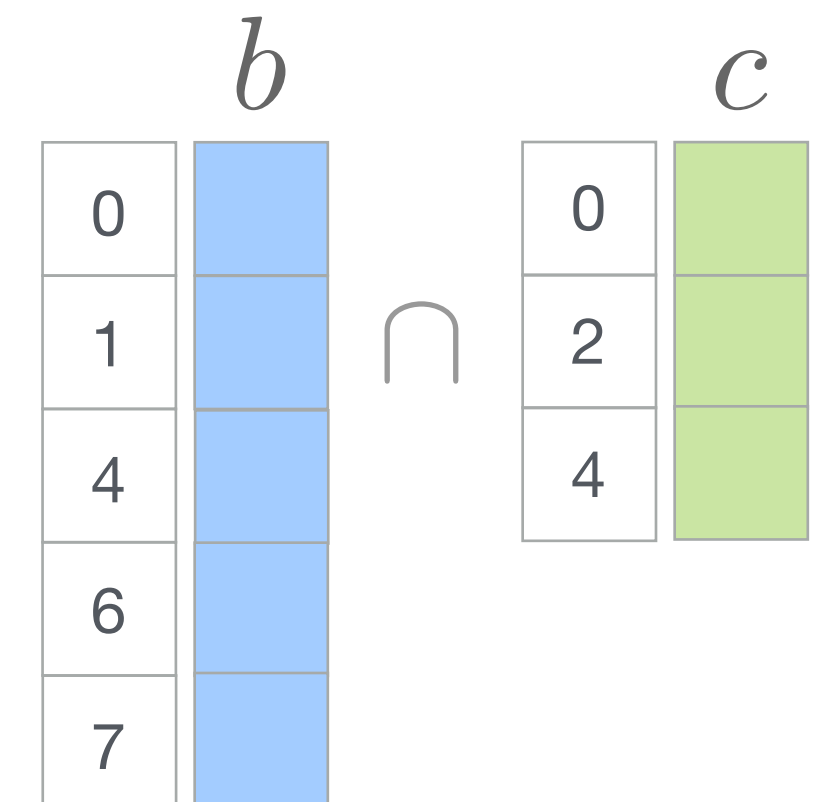
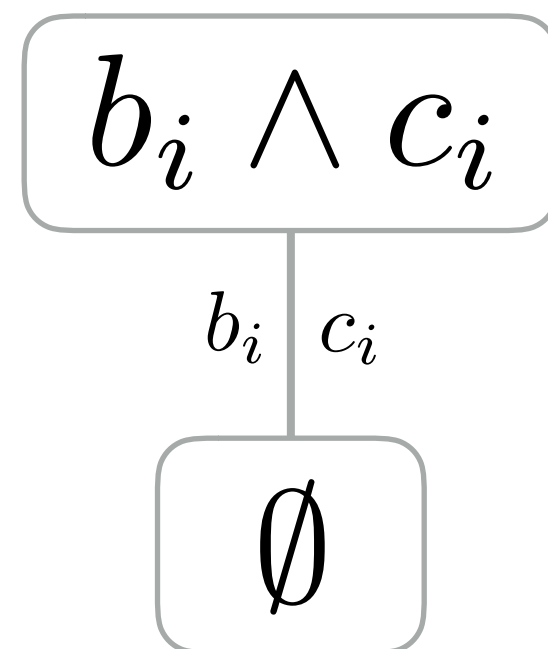
Merge Lattice for a Conjunction

$$a_i = b_i c_i$$



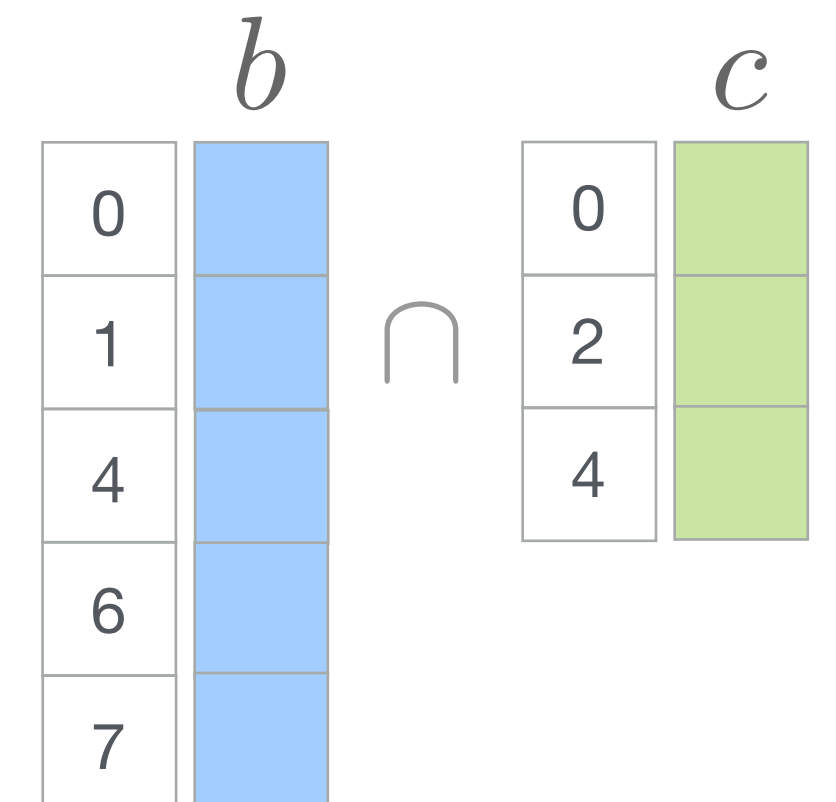
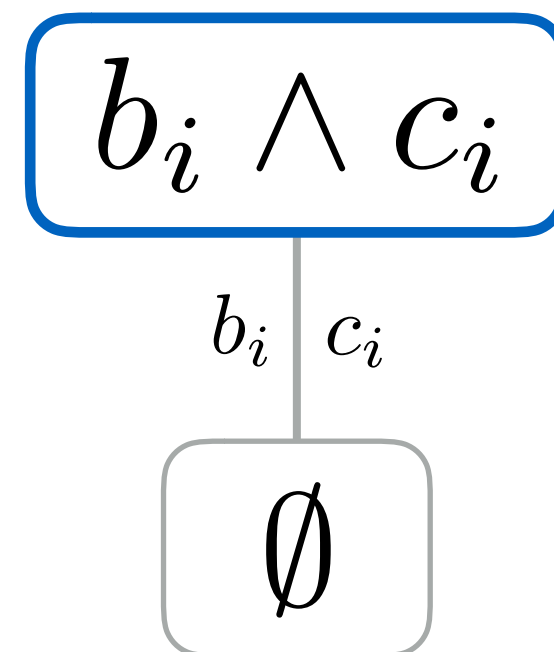
Merge Lattice for a Conjunction

$$a_i = b_i c_i$$



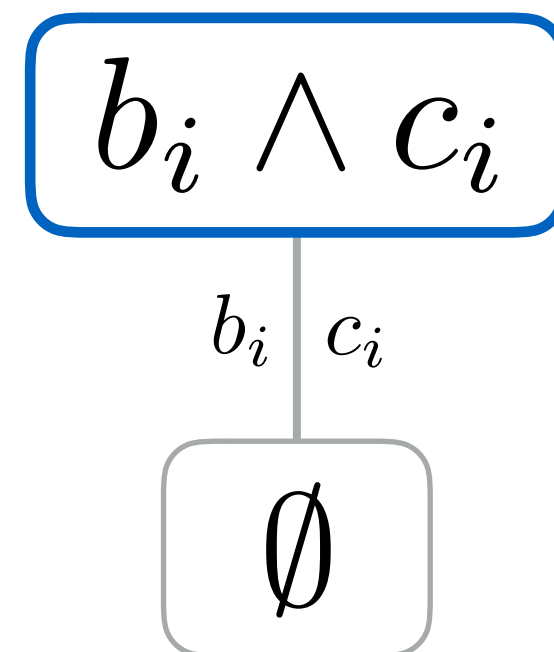
Merge Lattice for a Conjunction

$$a_i = b_i c_i$$



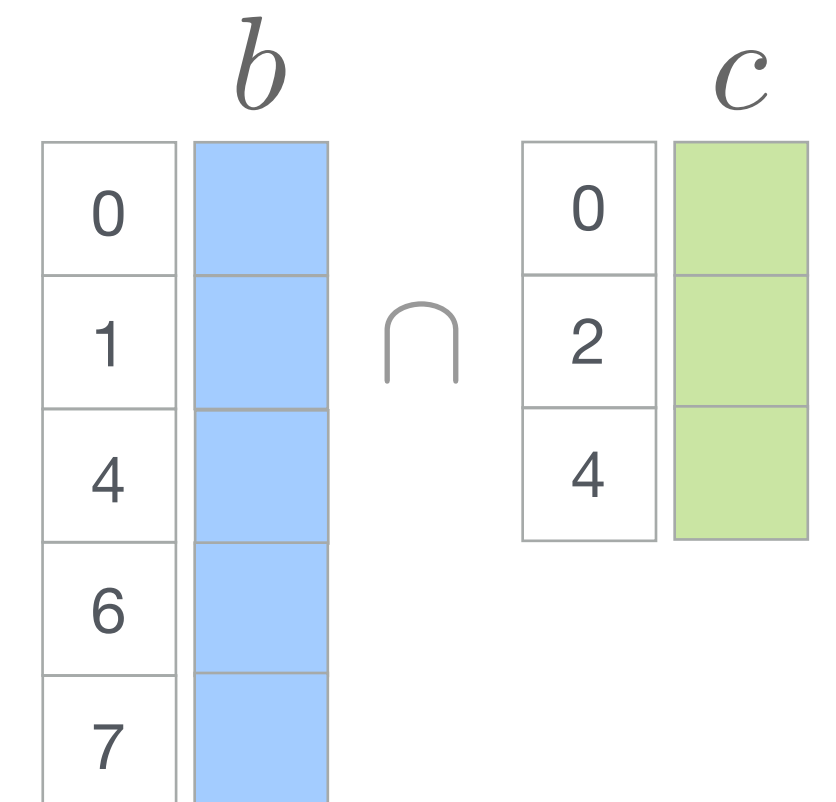
Merge Lattice for a Conjunction

$$a_i = b_i c_i$$



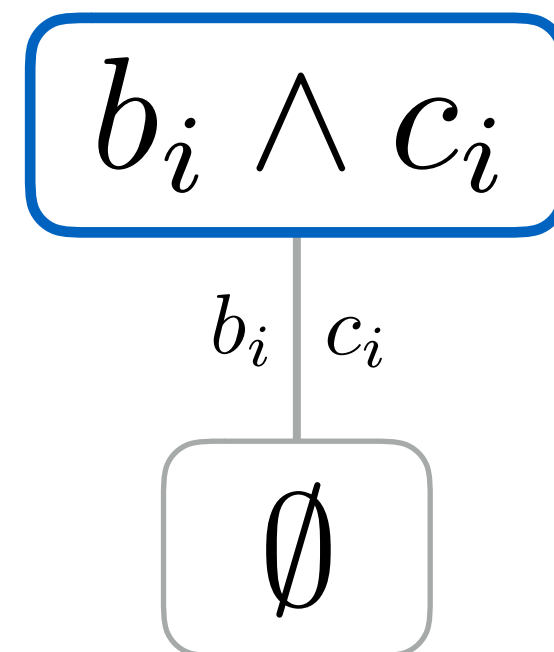
```
int pb1 = b1_pos[0];
int pc1 = c1_pos[0];
while (pb1 < b1_pos[1] && pc1 < c1_pos[1]) {
    int ib = b1_idx[pb1];
    int ic = c1_idx[pc1];
    int i = min(ib, ic);

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

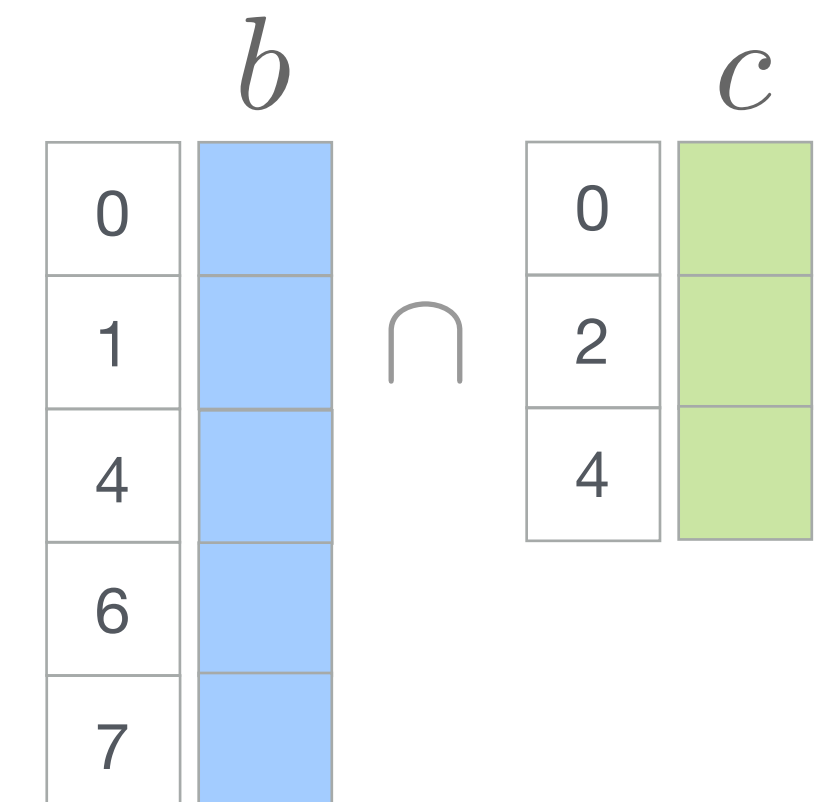


Merge Lattice for a Conjunction

$$a_i = b_i c_i$$

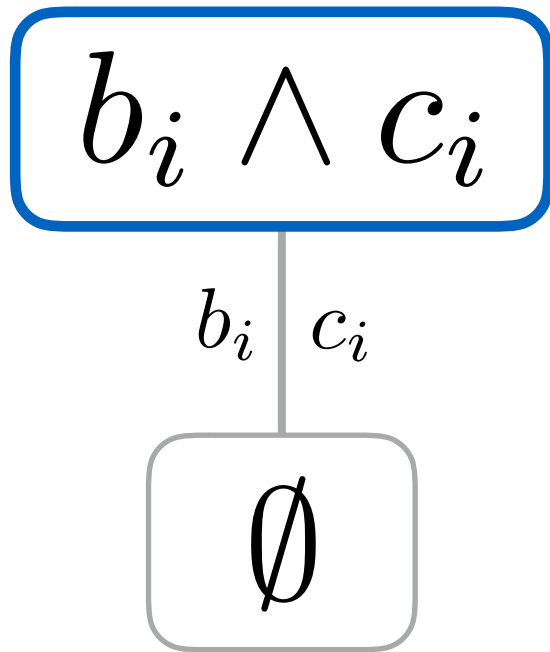


```
int pb1 = b1_pos[0];
int pc1 = c1_pos[0];
while (pb1 < b1_pos[1] && pc1 < c1_pos[1]) {
    int ib = b1_idx[pb1];
    int ic = c1_idx[pc1];
    int i = min(ib, ic);
    if (ib == i && ic == i) {
        a[i] = b[pb1] * c[pc1];
    }
    if (ib == i) pb1++;
    if (ic == i) pc1++;
}
```

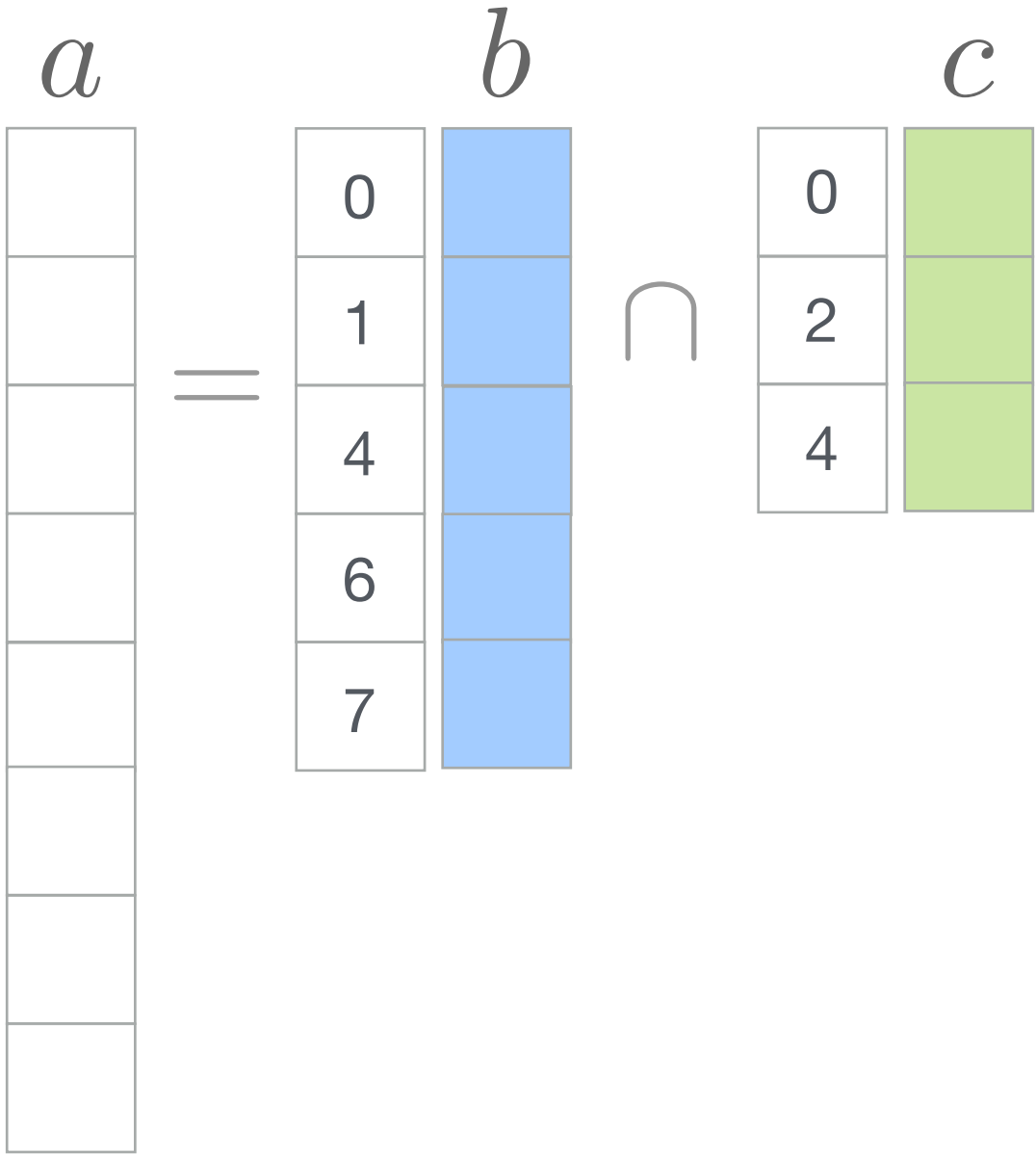


Merge Lattice for a Conjunction

$$a_i = b_i c_i$$

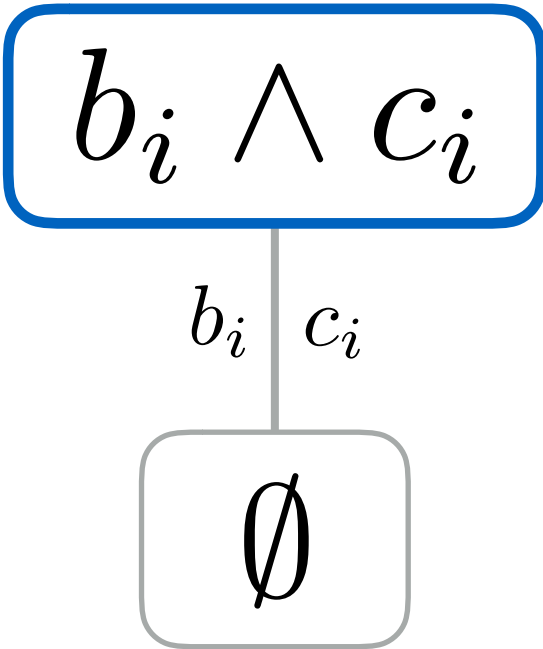


```
int pb1 = b1_pos[0];
int pc1 = c1_pos[0];
while (pb1 < b1_pos[1] && pc1 < c1_pos[1]) {
    int ib = b1_idx[pb1];
    int ic = c1_idx[pc1];
    int i = min(ib, ic);
    if (ib == i && ic == i) {
        a[i] = b[pb1] * c[pc1];
    }
    if (ib == i) pb1++;
    if (ic == i) pc1++;
}
```

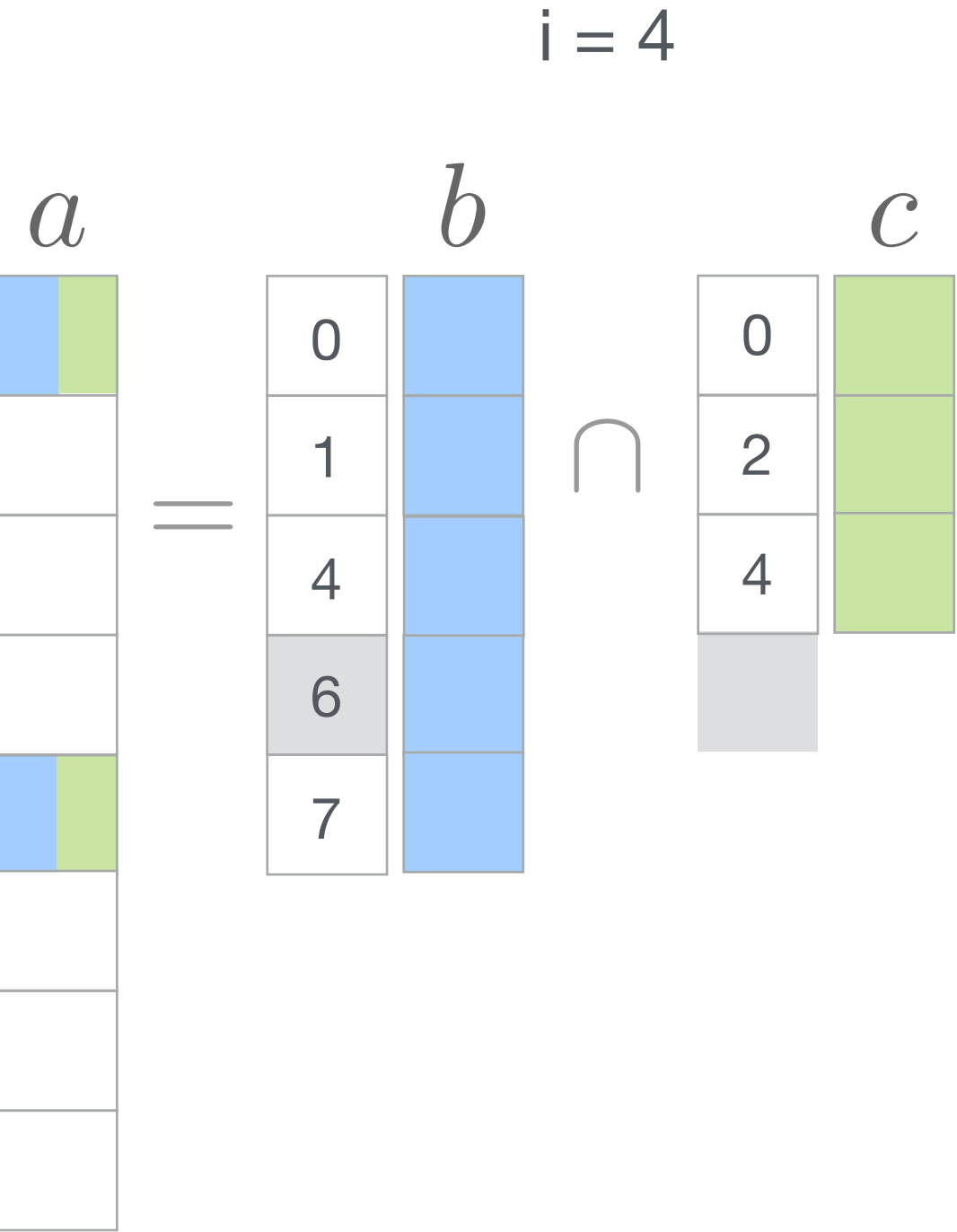


Merge Lattice for a Conjunction

$$a_i = b_i c_i$$

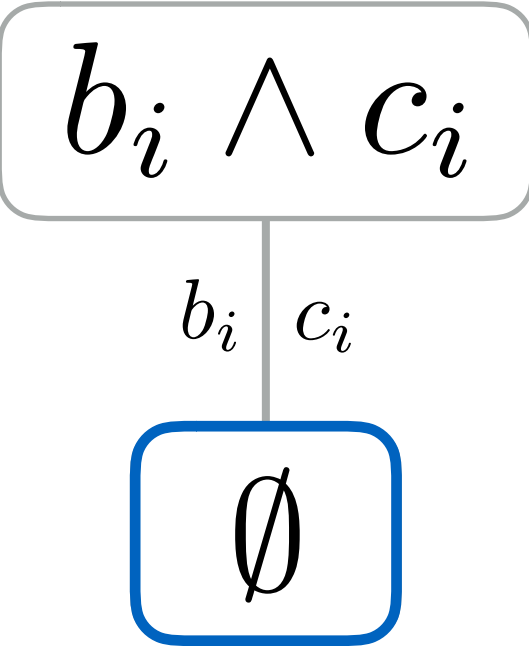


```
int pb1 = b1_pos[0];
int pc1 = c1_pos[0];
while (pb1 < b1_pos[1] && pc1 < c1_pos[1]) {
    int ib = b1_idx[pb1];
    int ic = c1_idx[pc1];
    int i = min(ib, ic);
    if (ib == i && ic == i) {
        a[i] = b[pb1] * c[pc1];
    }
    if (ib == i) pb1++;
    if (ic == i) pc1++;
}
```

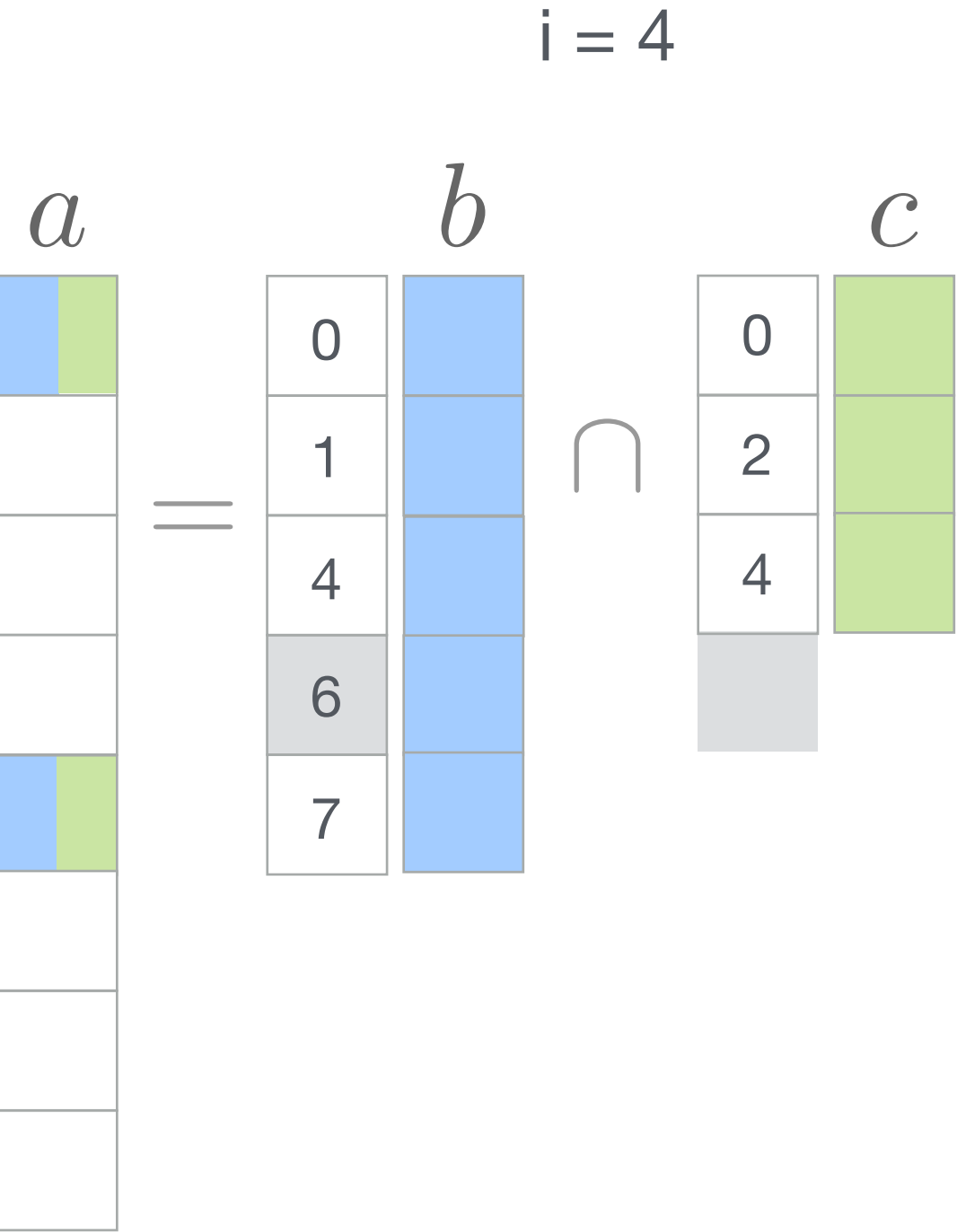


Merge Lattice for a Conjunction

$$a_i = b_i c_i$$

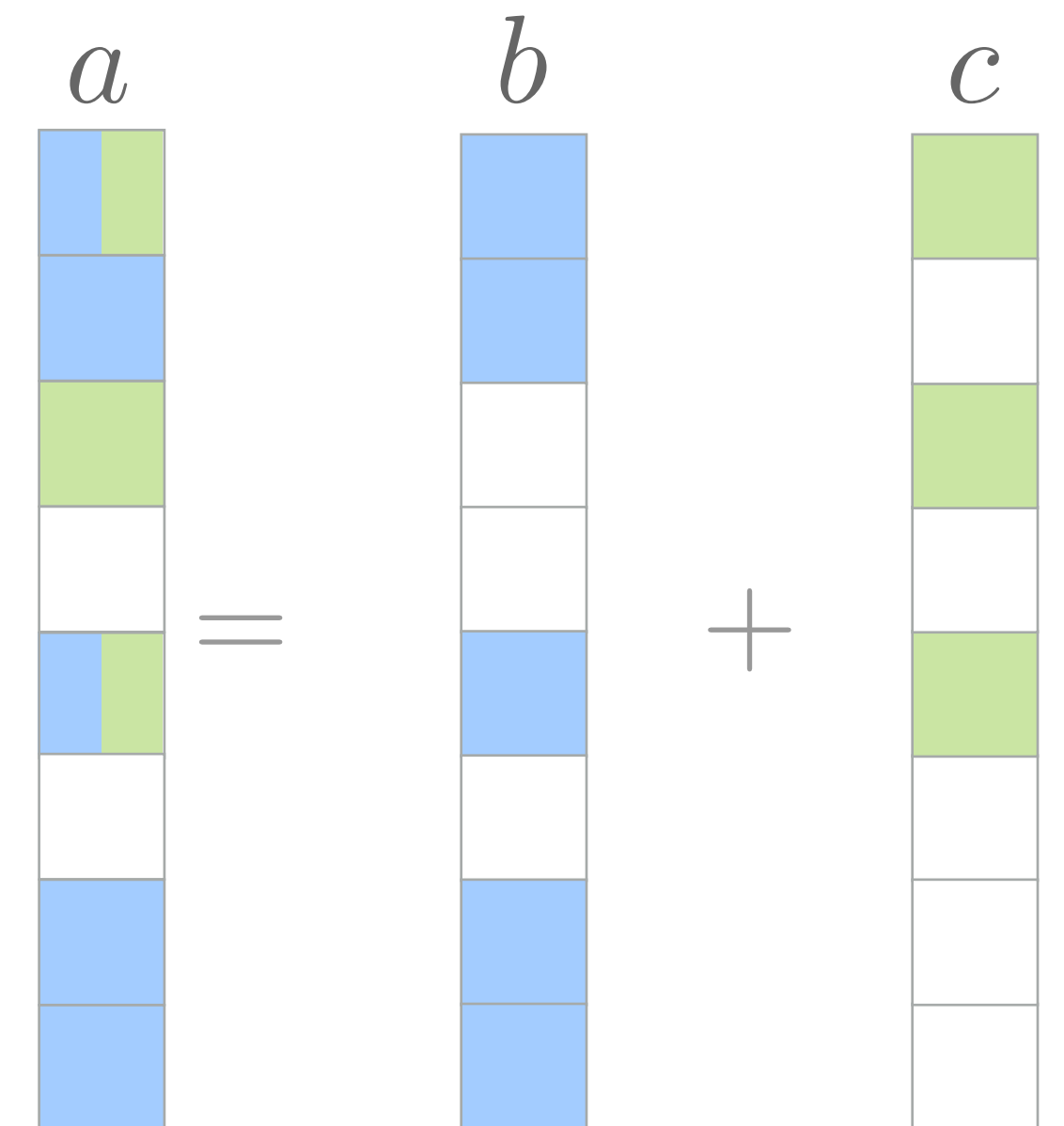


```
int pb1 = b1_pos[0];
int pc1 = c1_pos[0];
while (pb1 < b1_pos[1] && pc1 < c1_pos[1]) {
    int ib = b1_idx[pb1];
    int ic = c1_idx[pc1];
    int i = min(ib, ic);
    if (ib == i && ic == i) {
        a[i] = b[pb1] * c[pc1];
    }
    if (ib == i) pb1++;
    if (ic == i) pc1++;
}
```



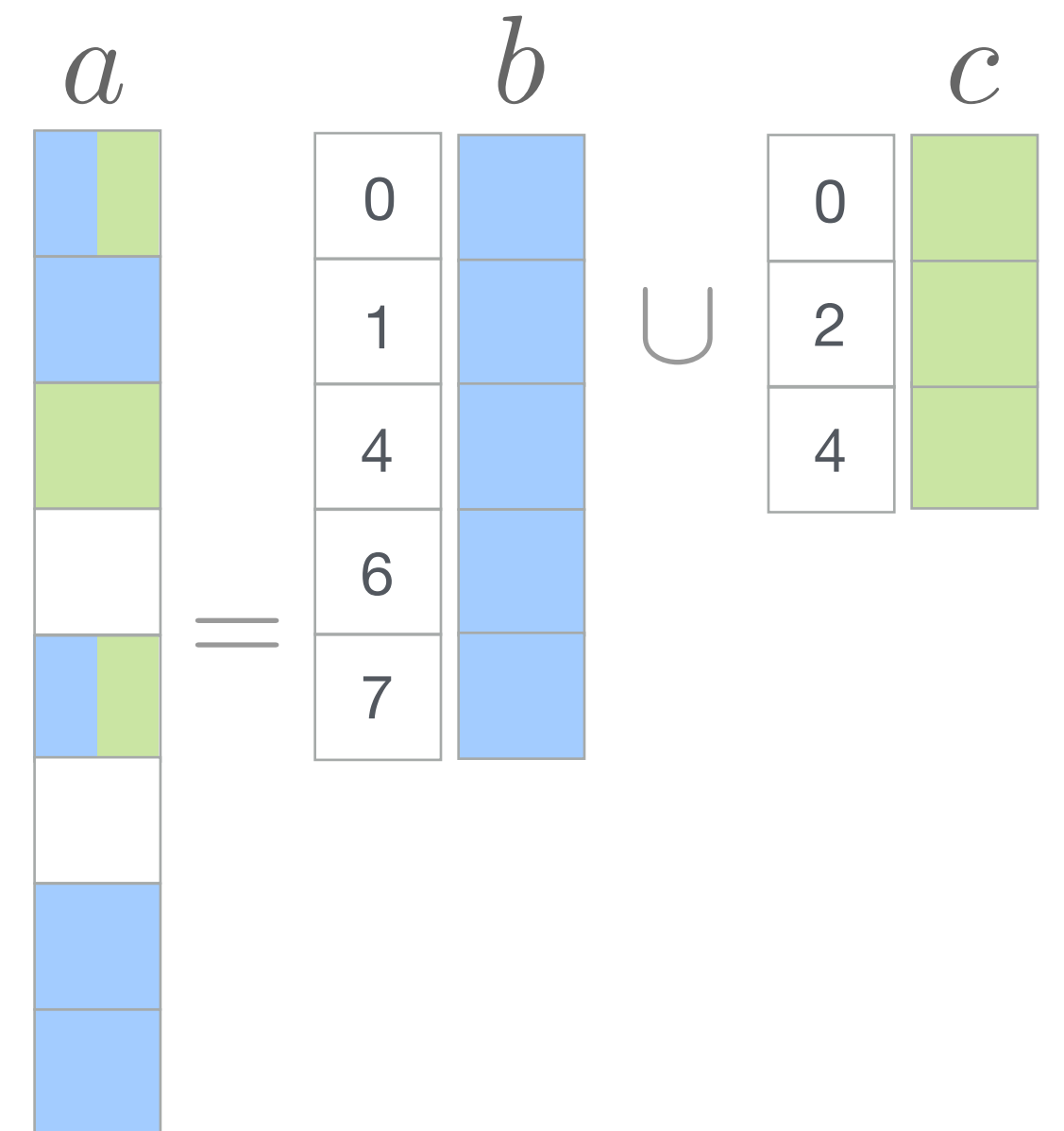
Merge Lattice for a Disjunction

$$a_i = b_i + c_i$$



Merge Lattice for a Disjunction

$$a_i = b_i + c_i$$

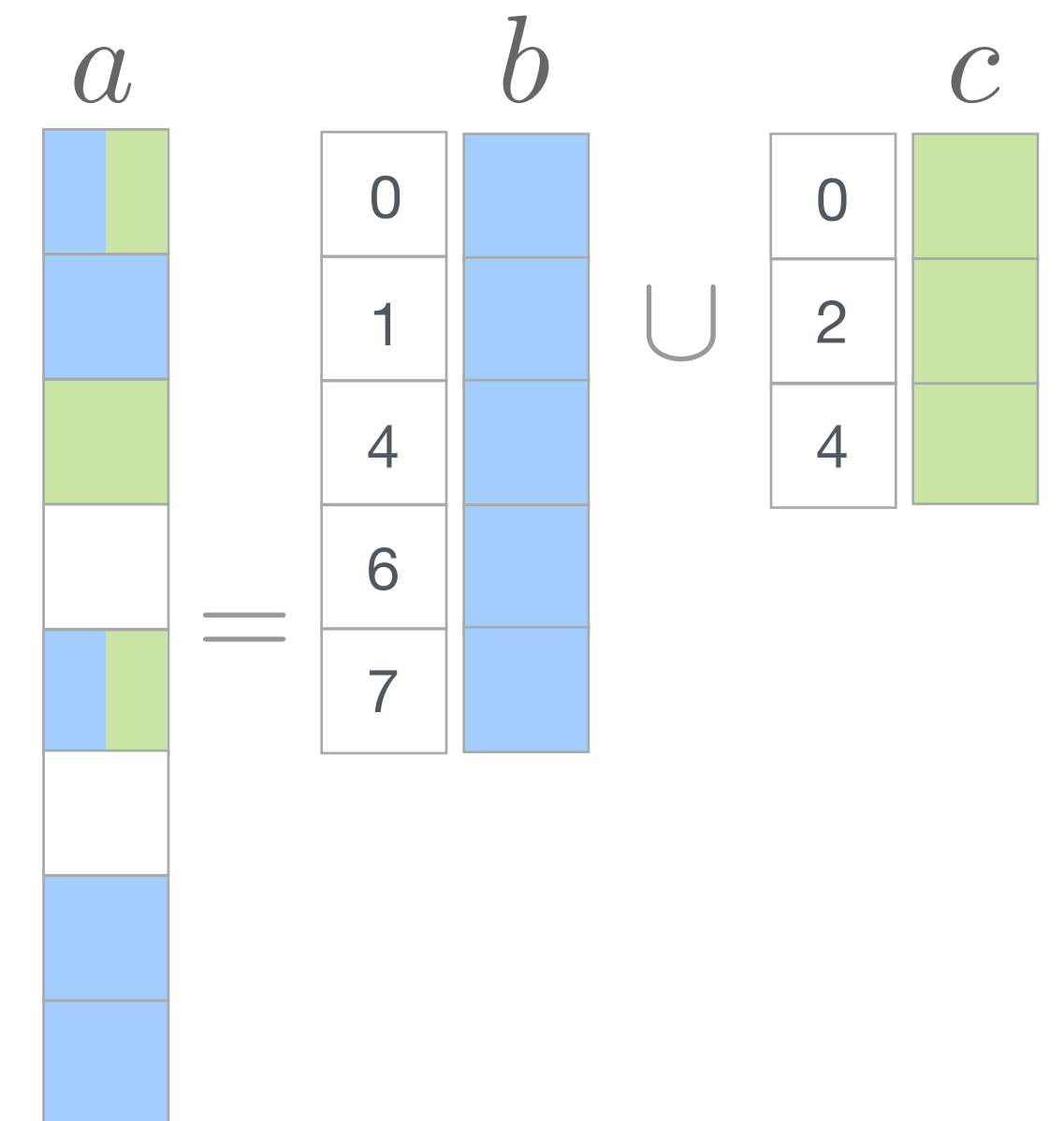


Merge Lattice for a Disjunction

$$a_i = b_i + c_i$$



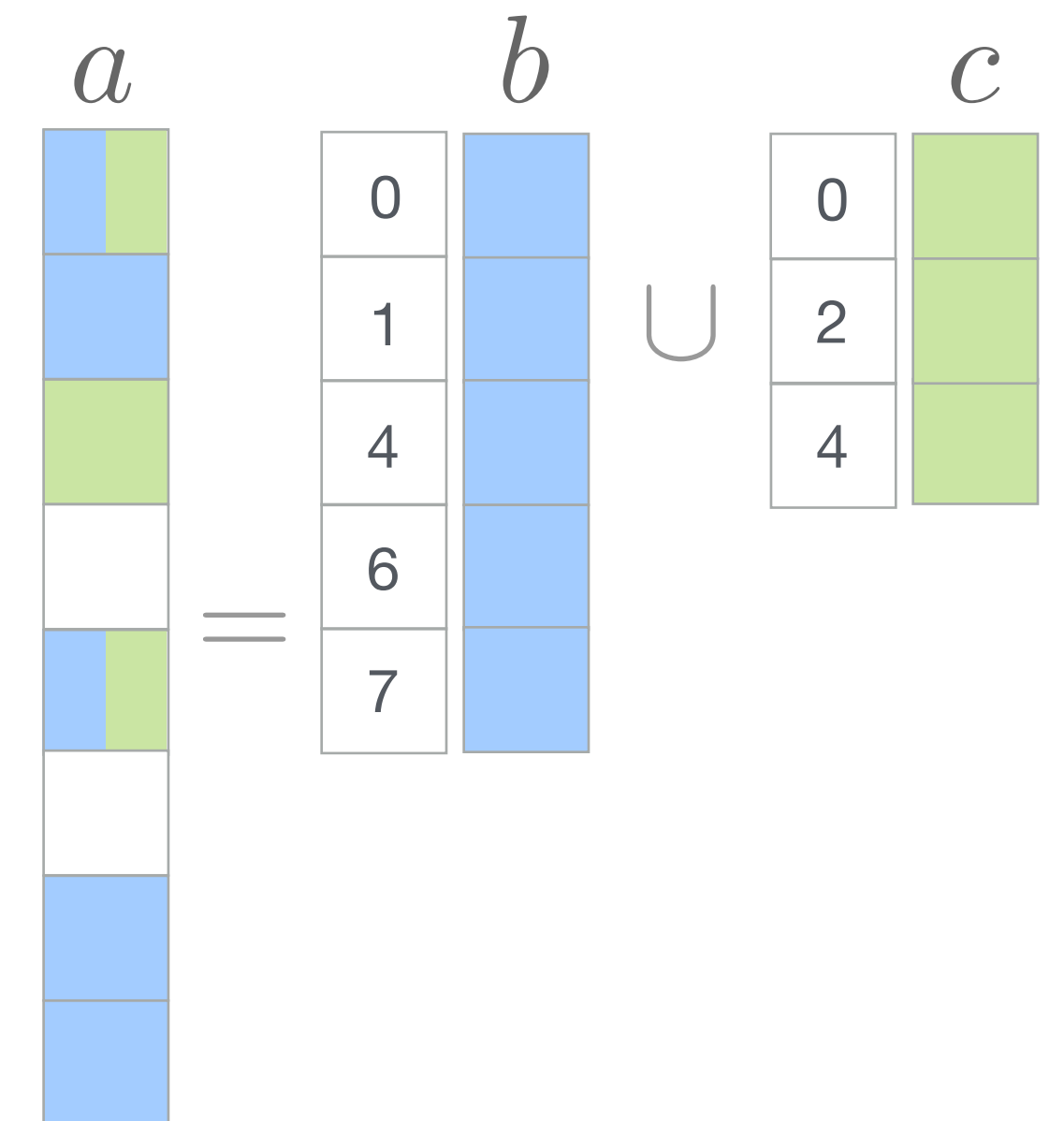
$b_i \vee c_i$



Merge Lattice for a Disjunction

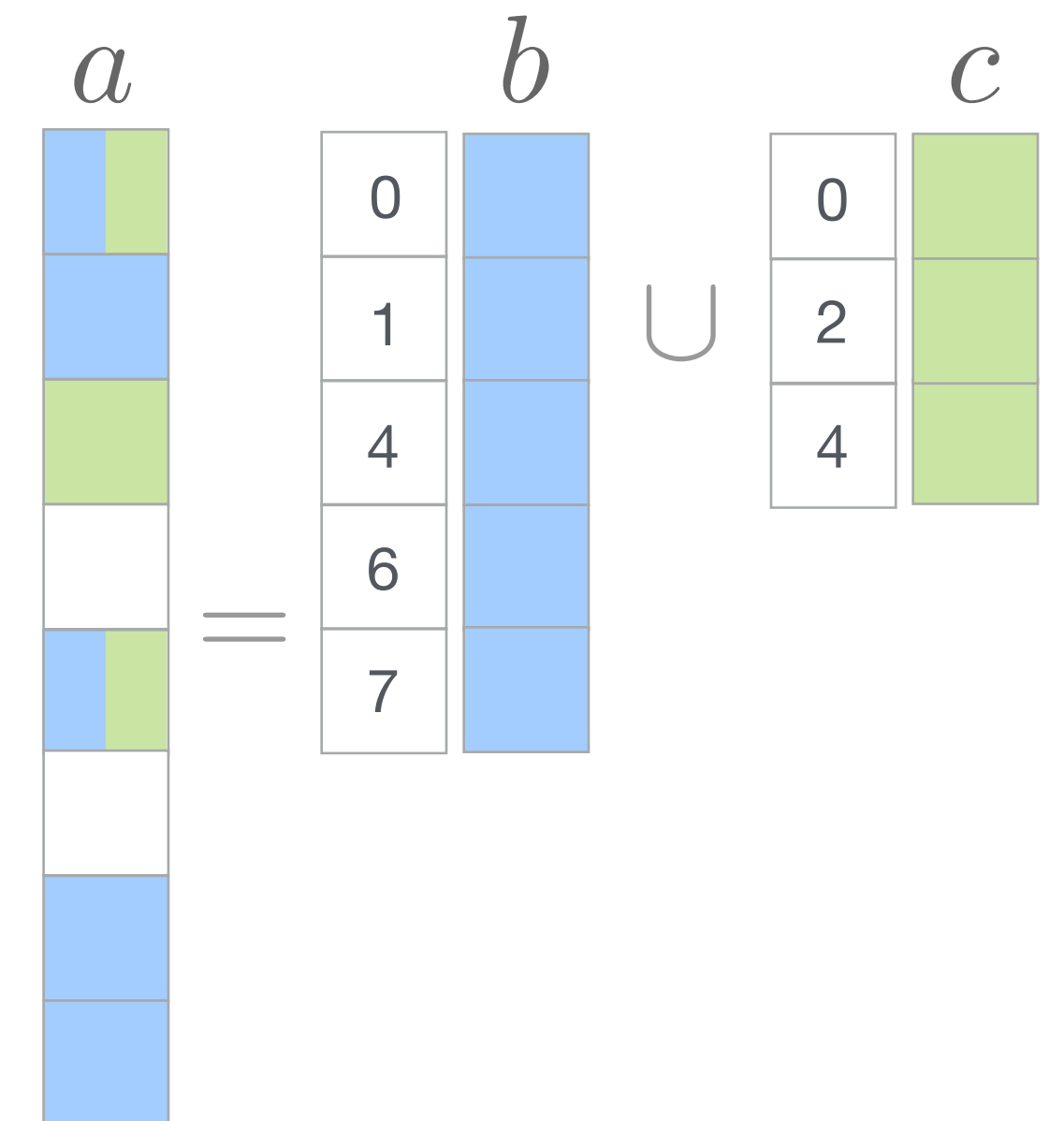
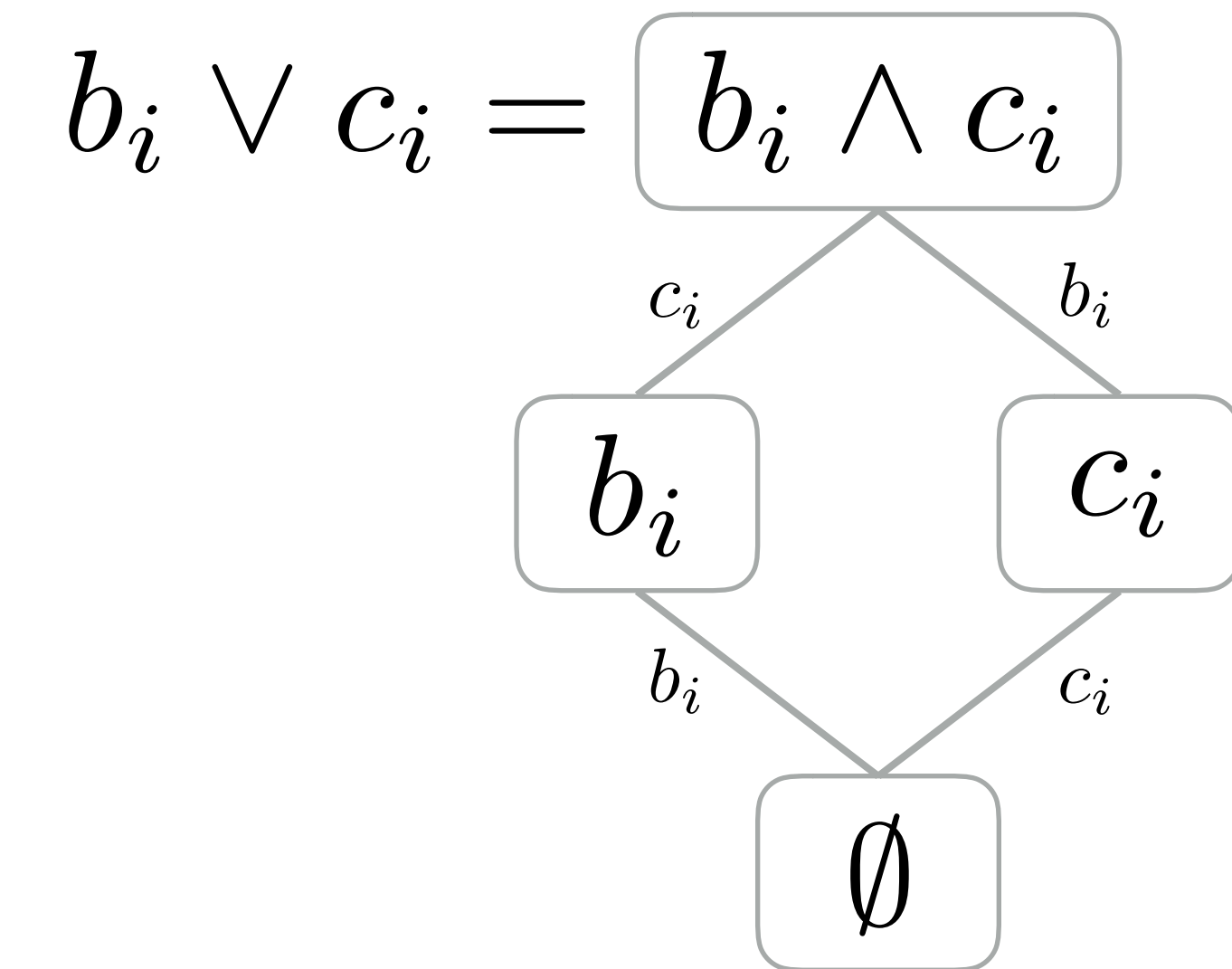
$$a_i = b_i + c_i$$

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



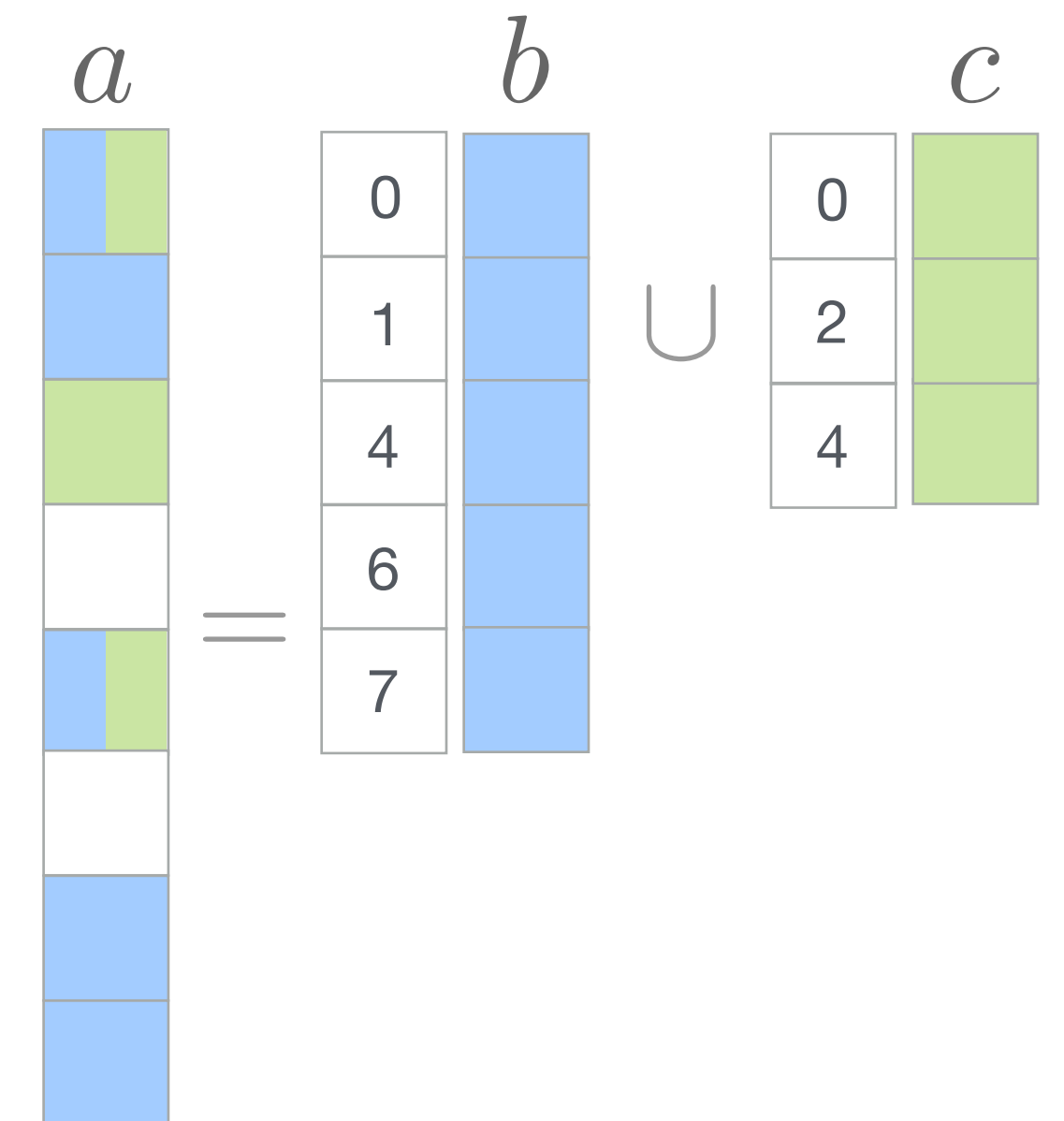
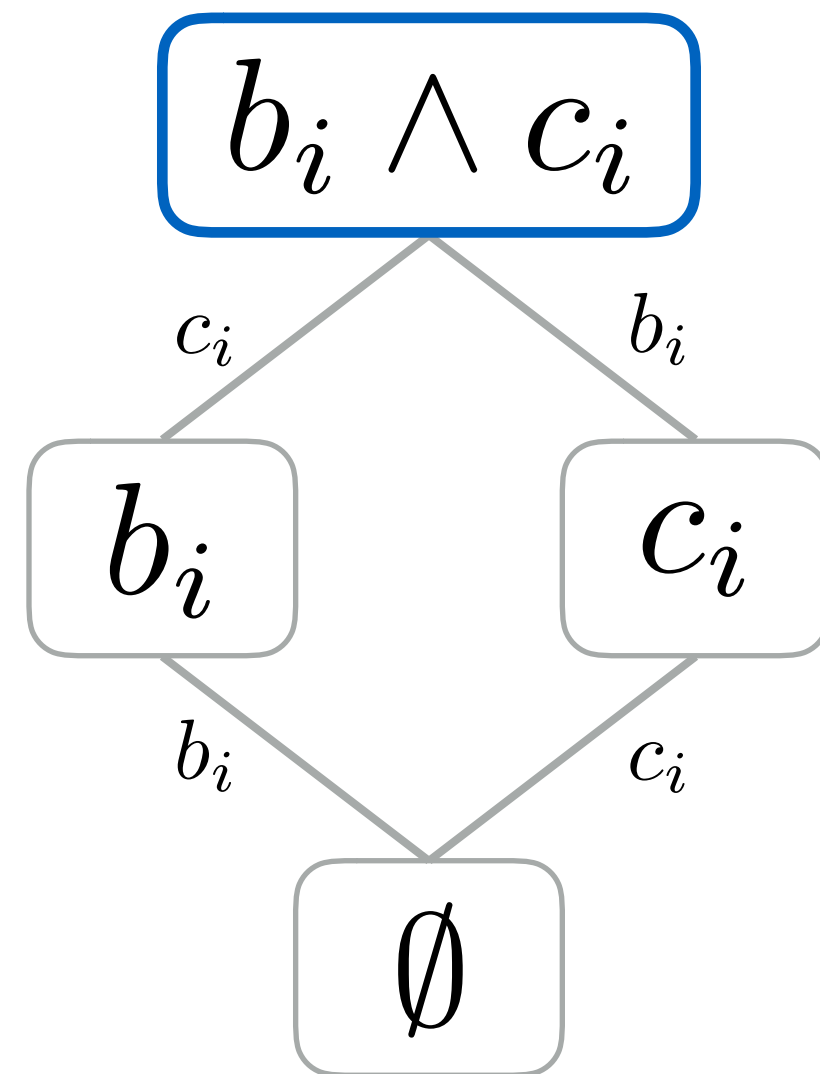
Merge Lattice for a Disjunction

$$a_i = b_i + c_i$$



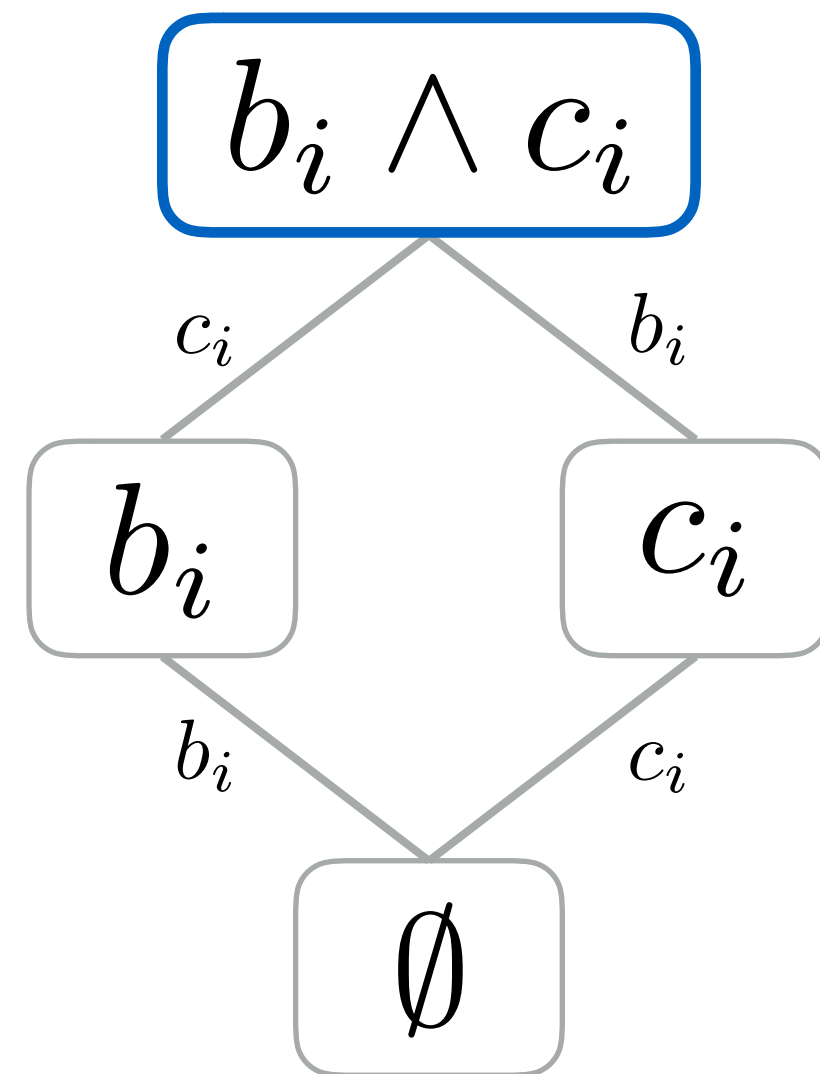
Merge Lattice for a Disjunction

$$a_i = b_i + c_i$$



Merge Lattice for a Disjunction

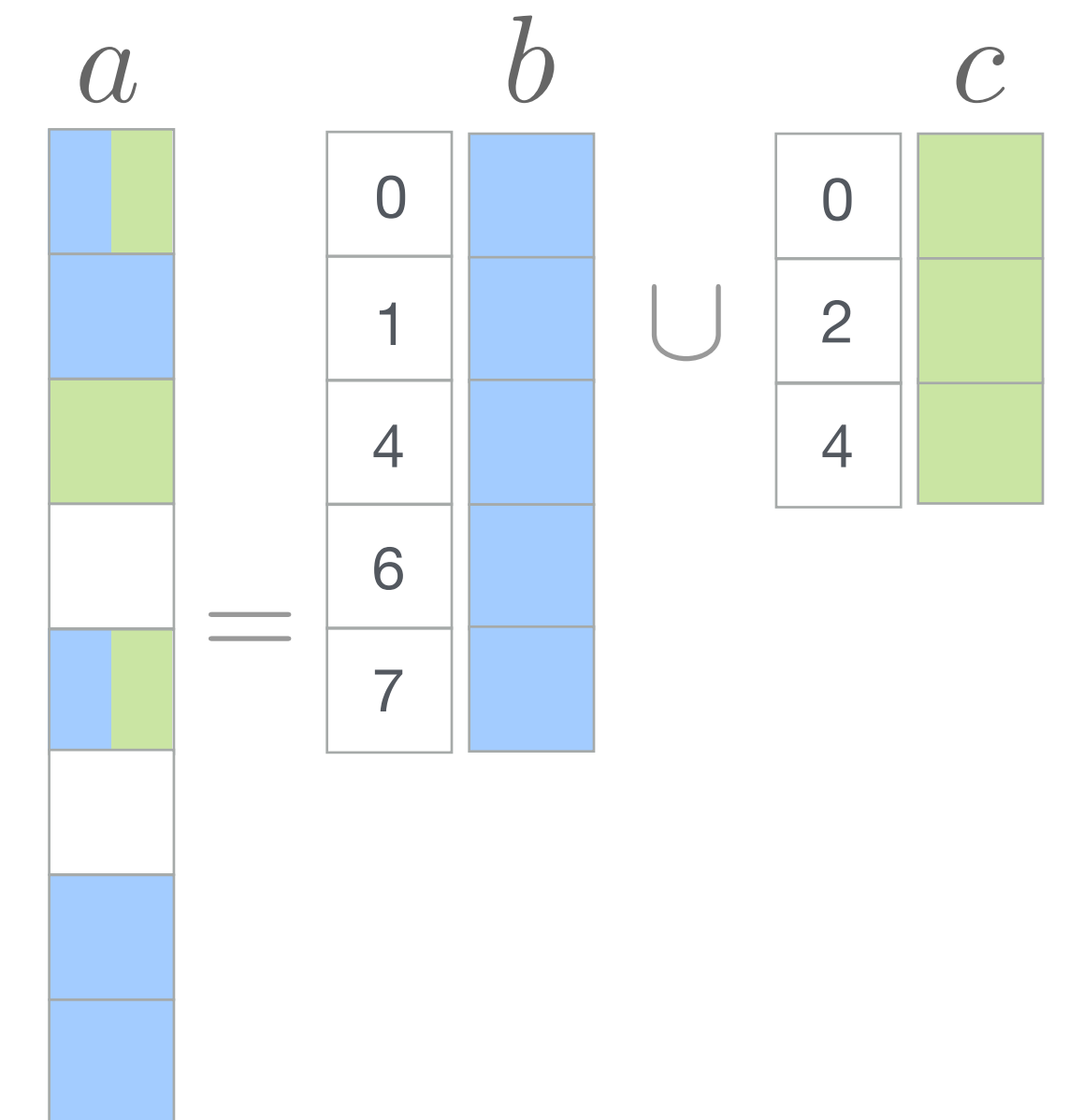
$$a_i = b_i + c_i$$



```

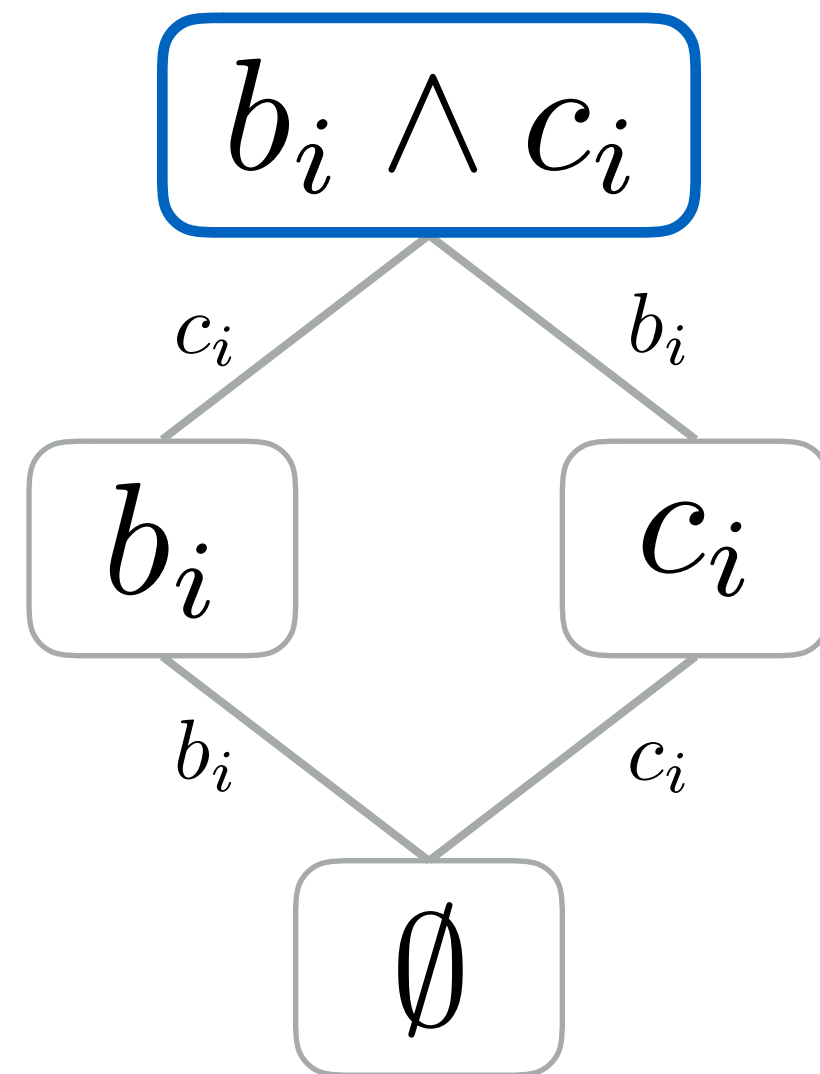
int pb1 = b1_pos[0];
int pc1 = c1_pos[0];
while (pb1 < b1_pos[1] && pc1 < c1_pos[1]) {
    int ib = b1_idx[pb1];
    int ic = c1_idx[pc1];
    int i = min(ib, ic);

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



Merge Lattice for a Disjunction

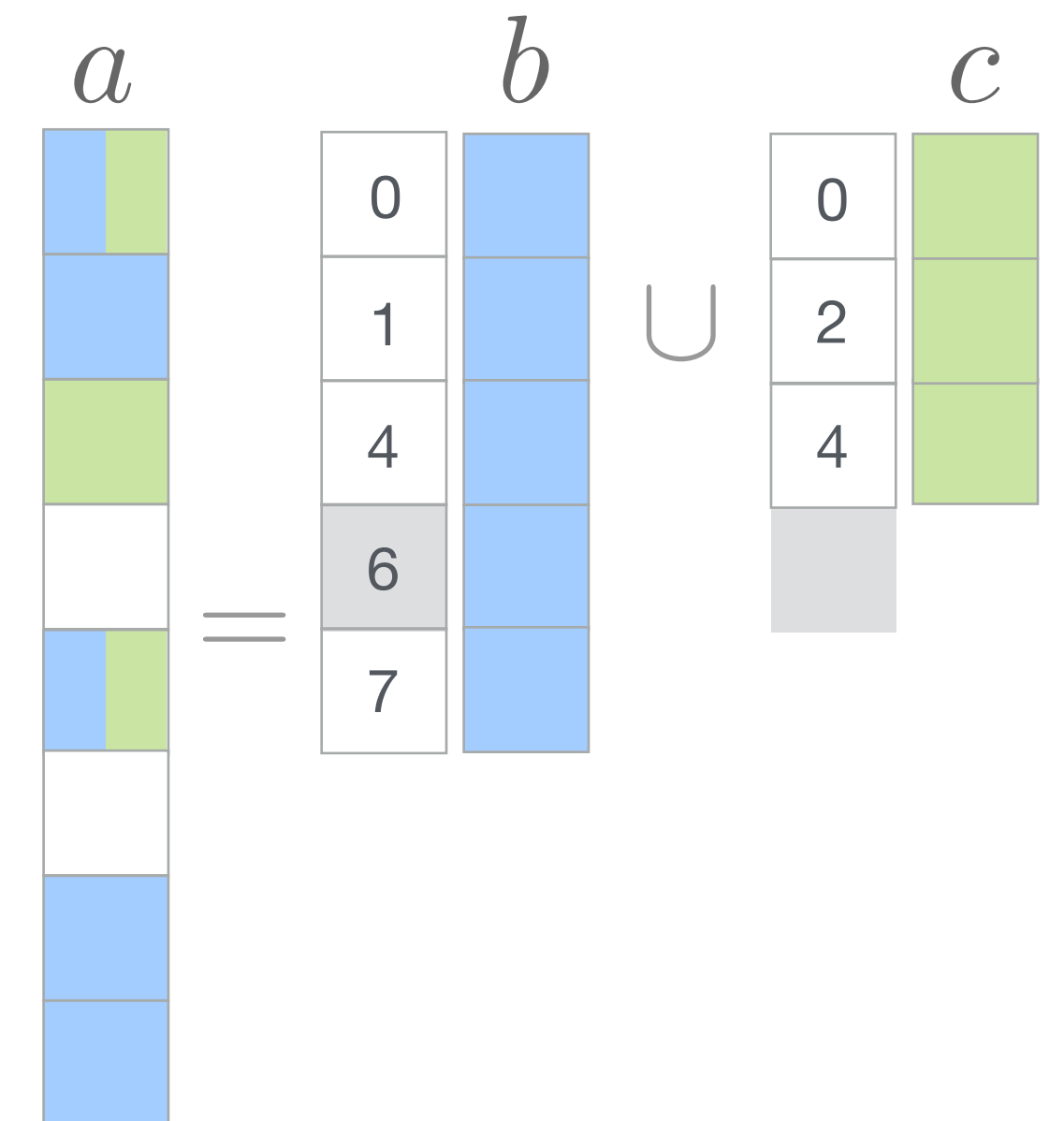
$$a_i = b_i + c_i$$



```

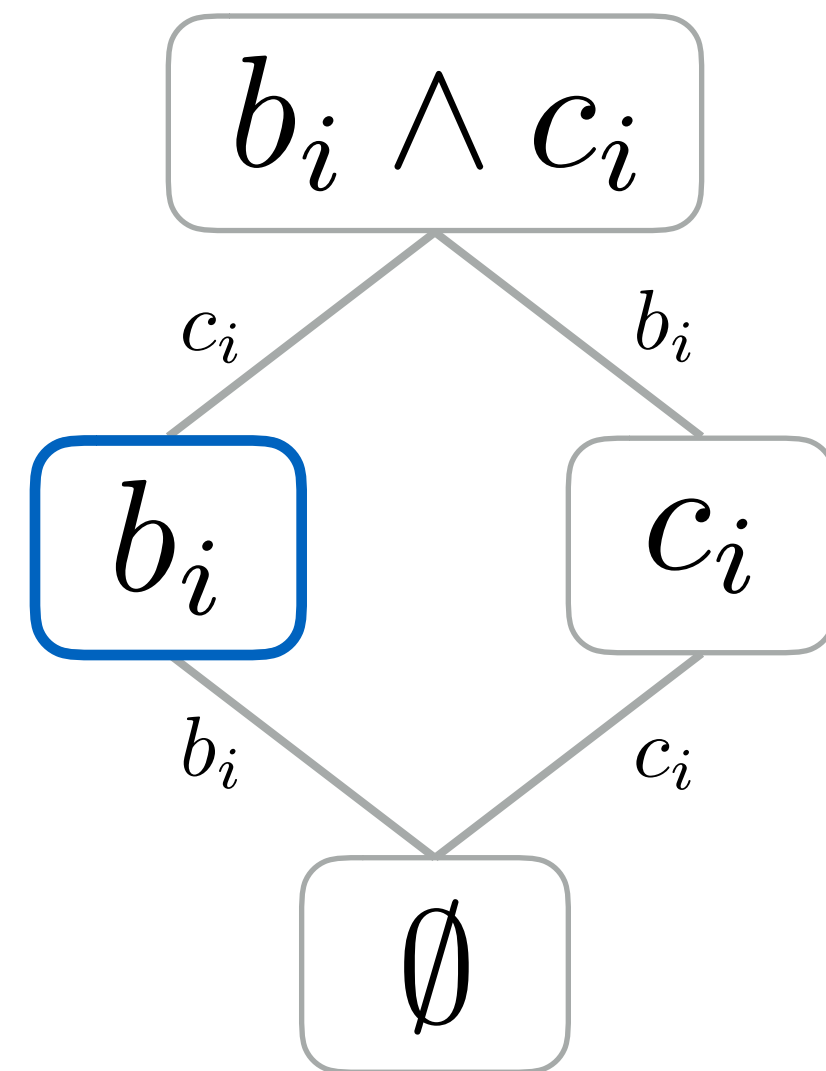
int pb1 = b1_pos[0];
int pc1 = c1_pos[0];
while (pb1 < b1_pos[1] && pc1 < c1_pos[1]) {
    int ib = b1_idx[pb1];
    int ic = c1_idx[pc1];
    int i = min(ib, ic);

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



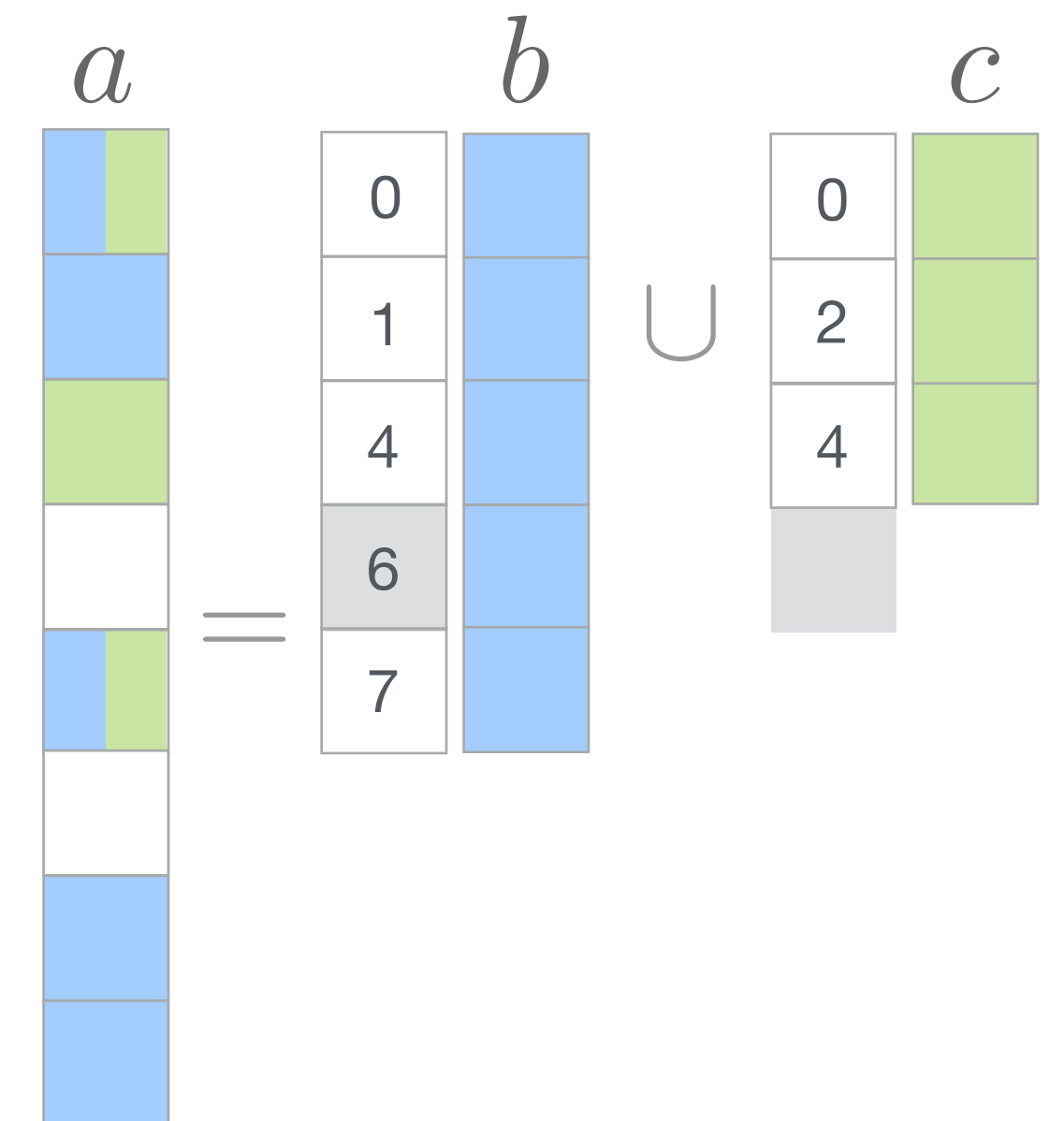
Merge Lattice for a Disjunction

$$a_i = b_i + c_i$$



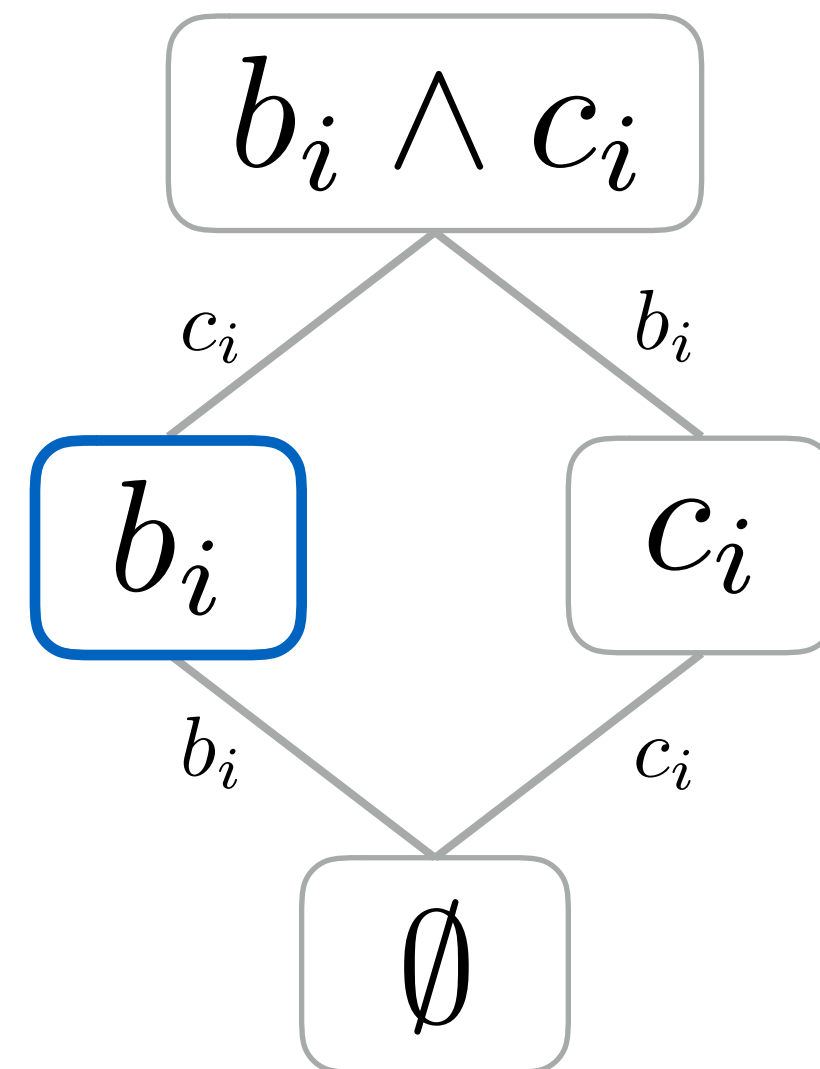
```
int pb1 = b1_pos[0];
int pc1 = c1_pos[0];
while (pb1 < b1_pos[1] && pc1 < c1_pos[1]) {
    int ib = b1_idx[pb1];
    int ic = c1_idx[pc1];
    int i = min(ib, ic);
```

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



Merge Lattice for a Disjunction

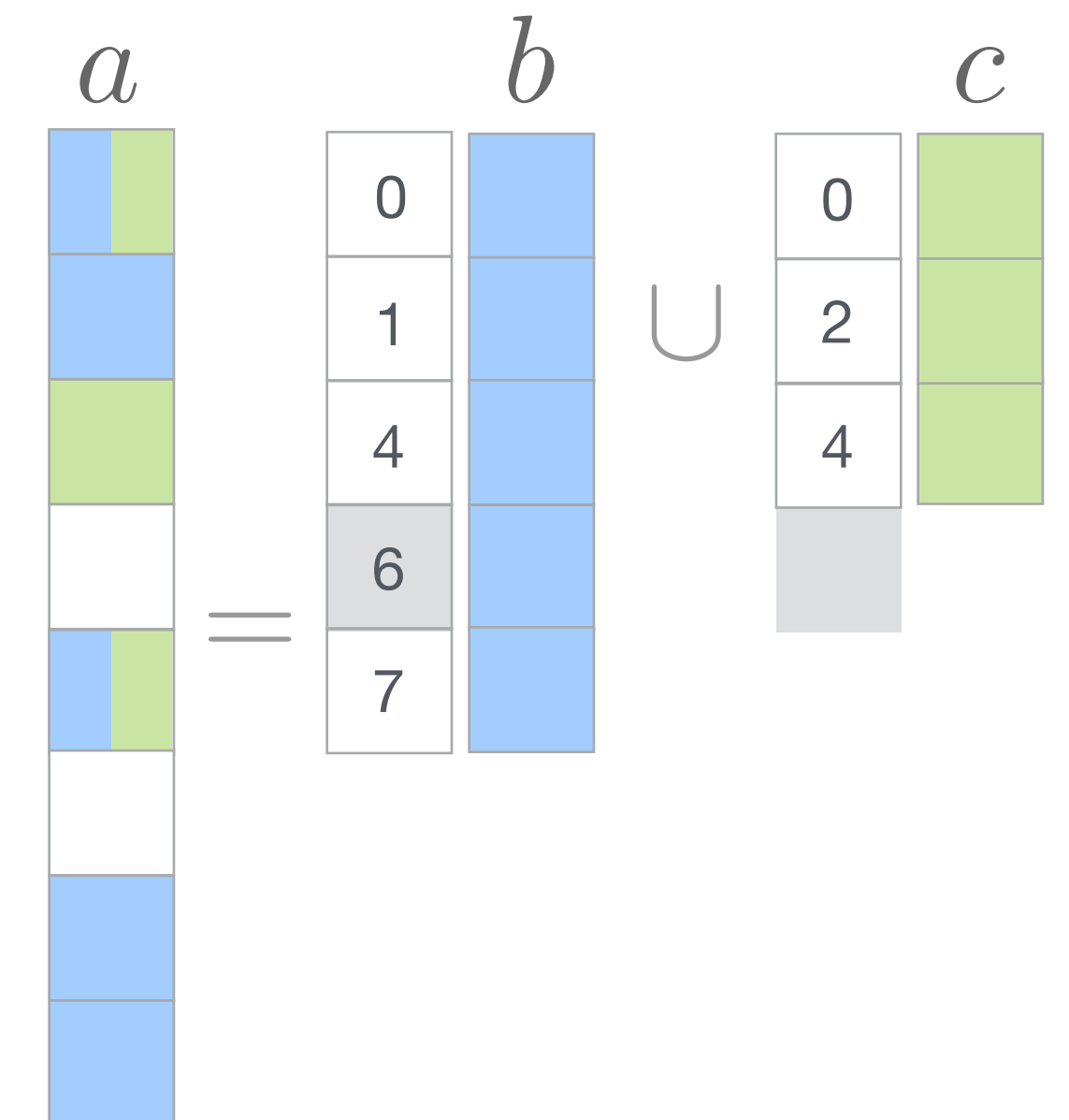
$$a_i = b_i + c_i$$



```
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);
```

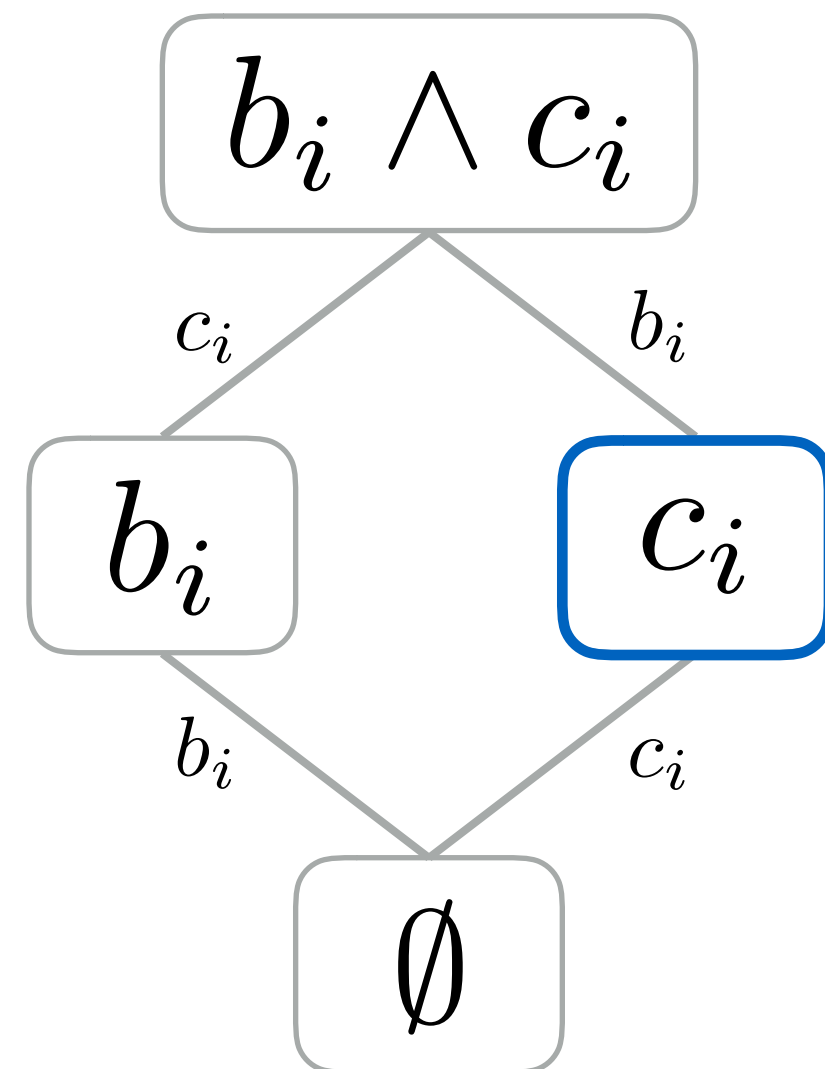
```
    if (ib == i) pb1++;
    if (ic == i) pc1++;
}
```

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



Merge Lattice for a Disjunction

$$a_i = b_i + c_i$$

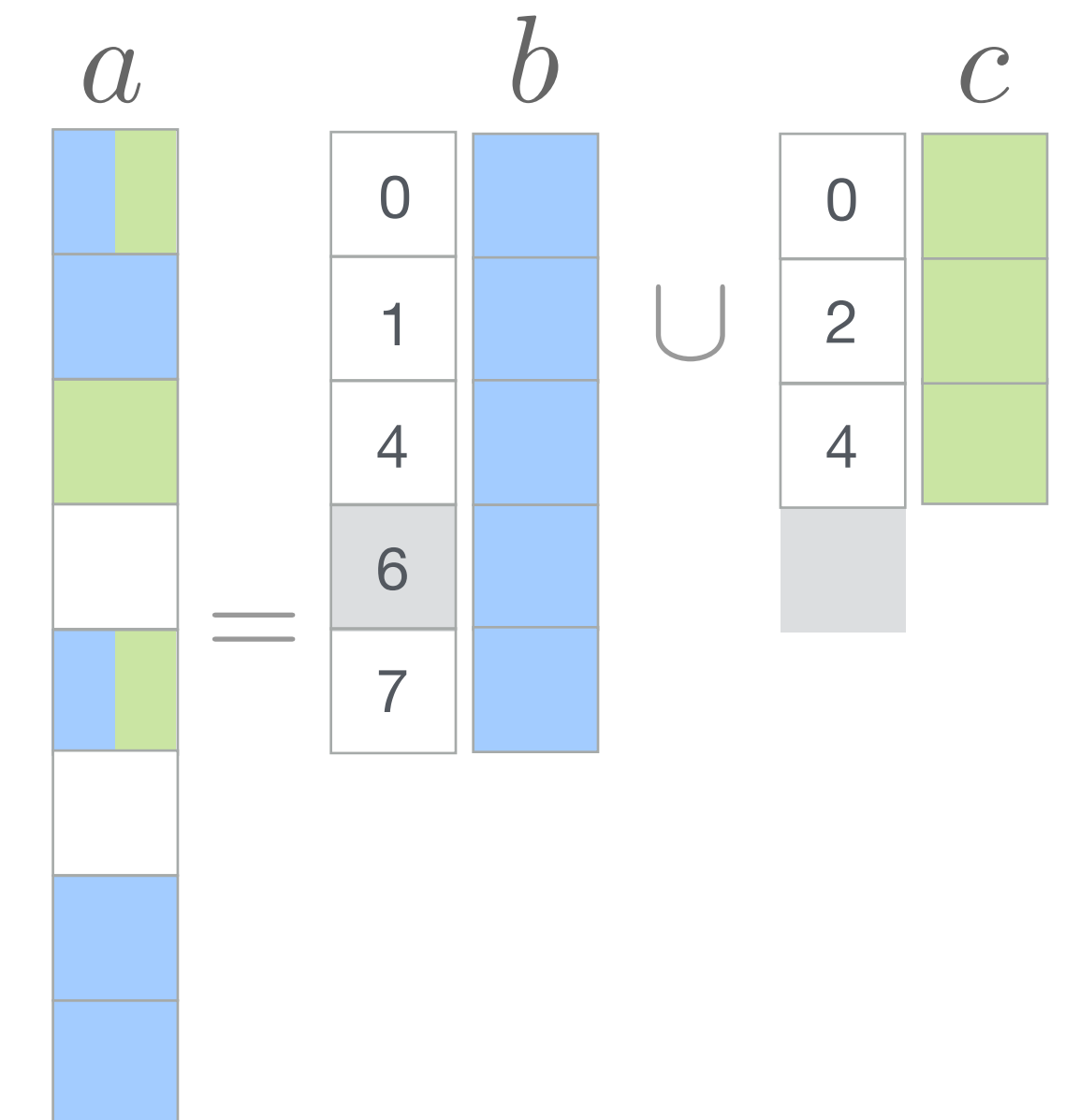


```
int pb1 = b1_pos[0];
int pc1 = c1_pos[0];
while (pb1 < b1_pos[1] && pc1 < c1_pos[1]) {
    int ib = b1_idx[pb1];
    int ic = c1_idx[pc1];
    int i = min(ib, ic);
```

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

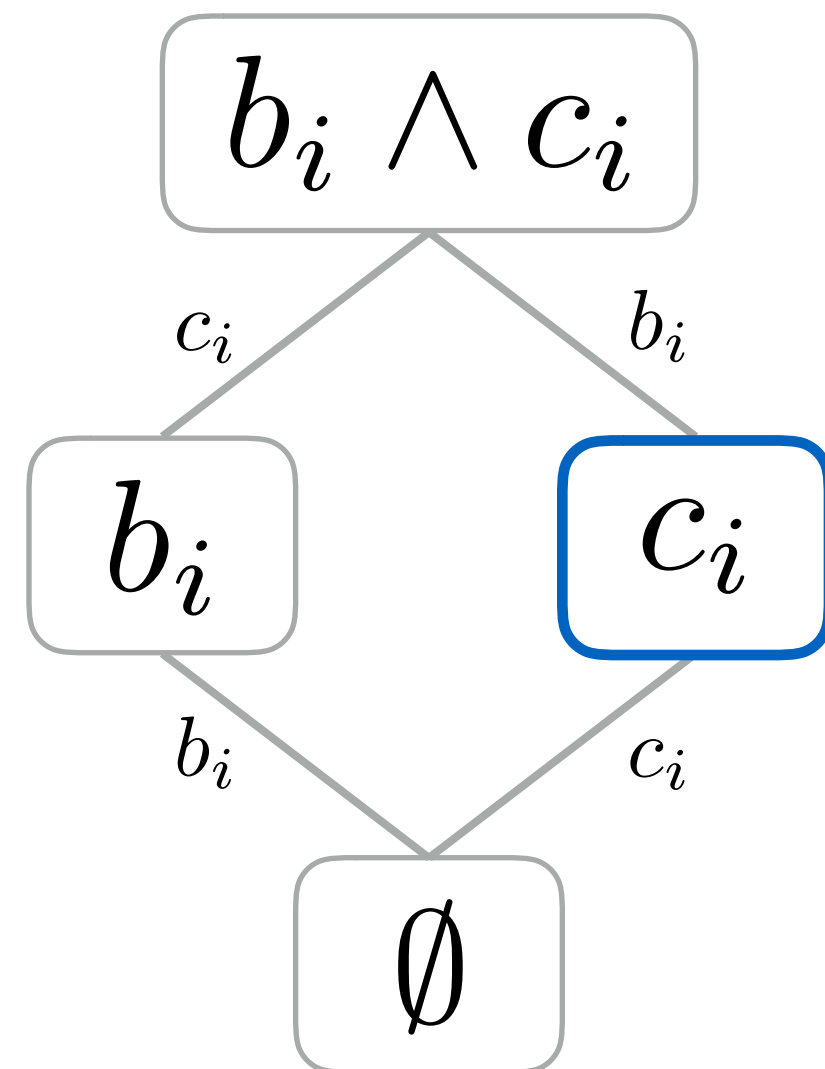
while (pb1 < b1_pos[1]) {
    int i = b1_idx[pb1];
    a[i] = b[pb1++];
}
```

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



Merge Lattice for a Disjunction

$$a_i = b_i + c_i$$

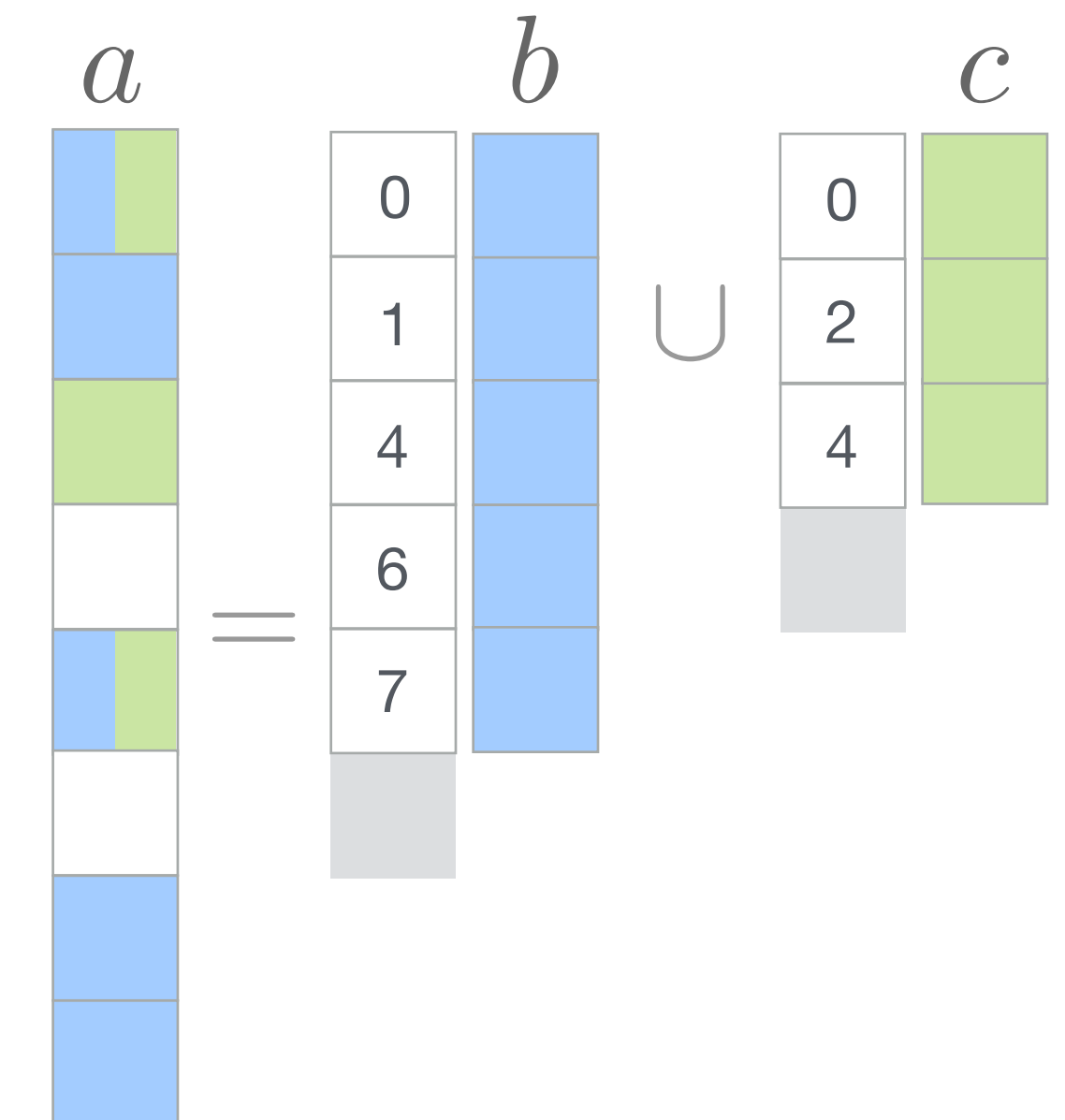


```
int pb1 = b1_pos[0];
int pc1 = c1_pos[0];
while (pb1 < b1_pos[1] && pc1 < c1_pos[1]) {
    int ib = b1_idx[pb1];
    int ic = c1_idx[pc1];
    int i = min(ib, ic);
```

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

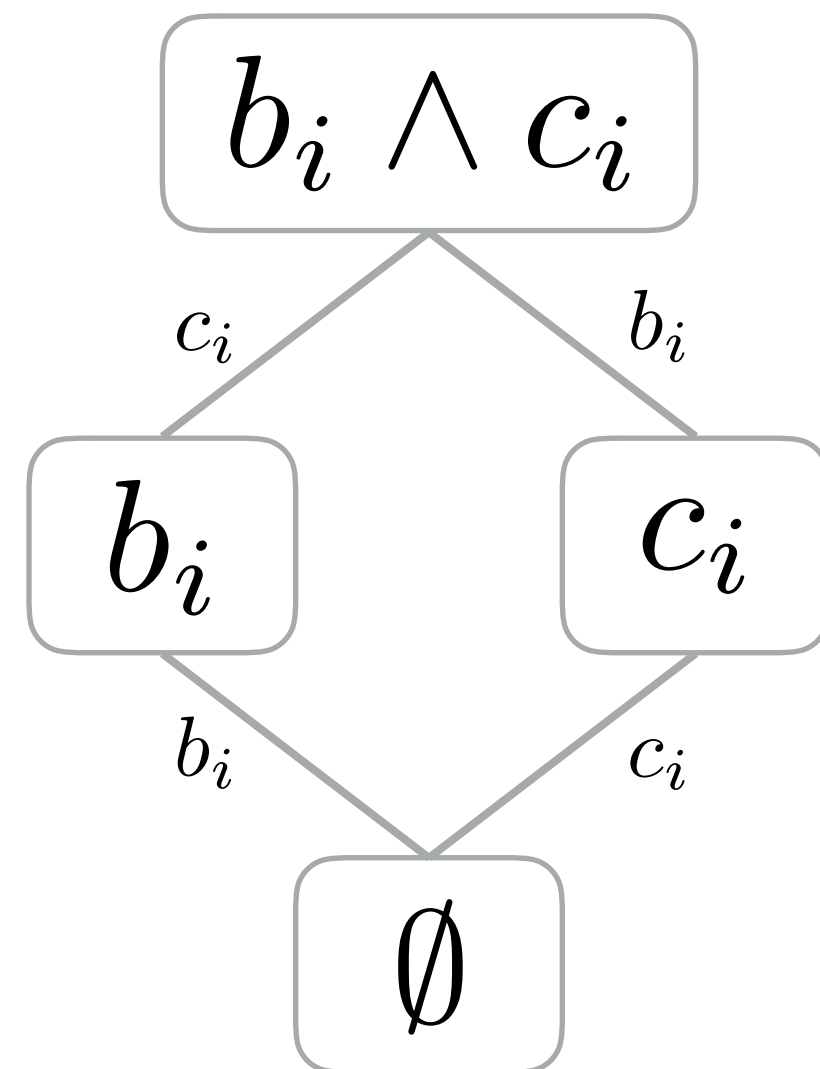
while (pb1 < b1_pos[1]) {
    int i = b1_idx[pb1];
    a[i] = b[pb1++];
}
```

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



Merge Lattice for a Disjunction

$$a_i = b_i + c_i$$

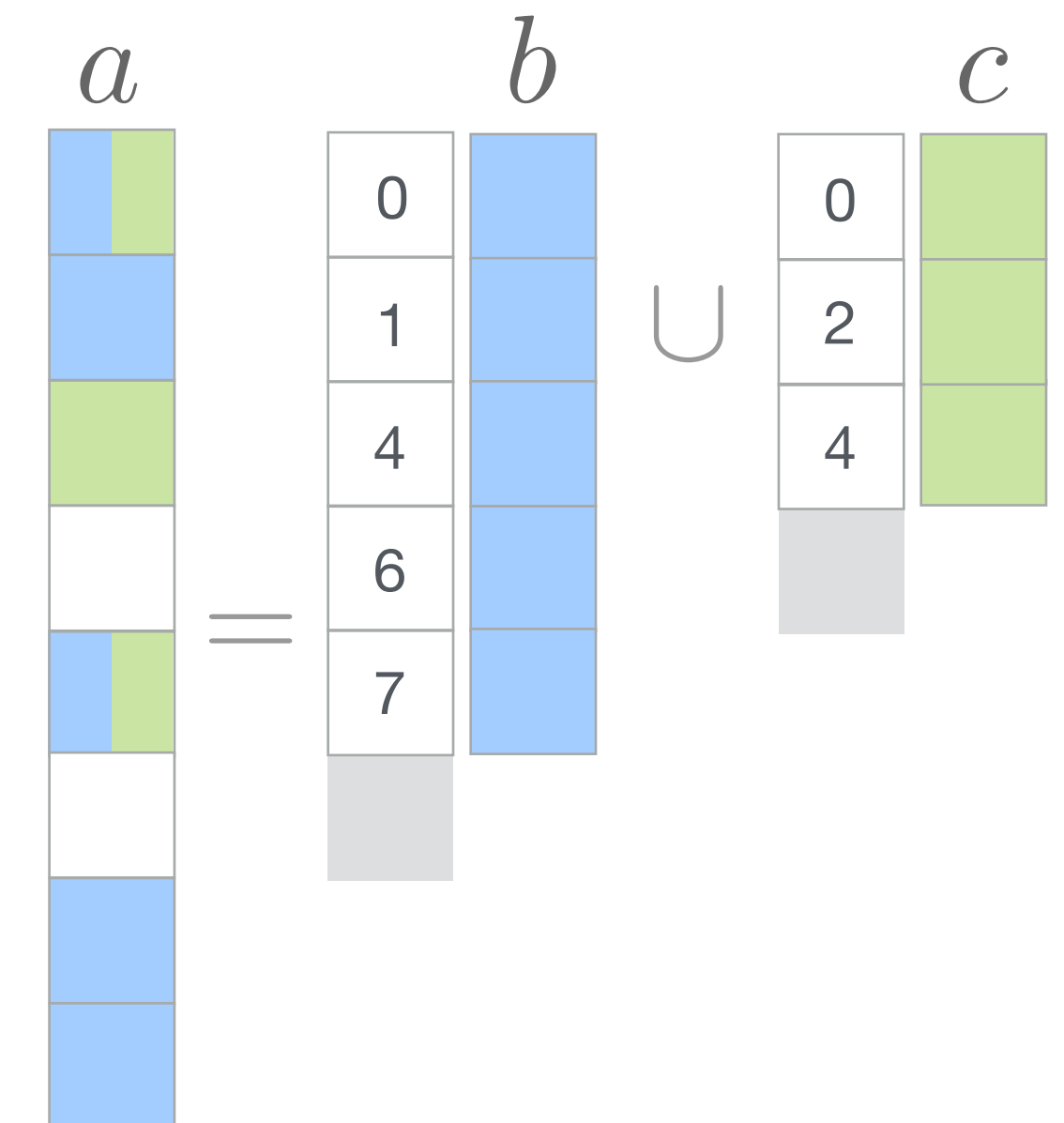


```
int pb1 = b1_pos[0];
int pc1 = c1_pos[0];
while (pb1 < b1_pos[1] && pc1 < c1_pos[1]) {
    int ib = b1_idx[pb1];
    int ic = c1_idx[pc1];
    int i = min(ib, ic);
```

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

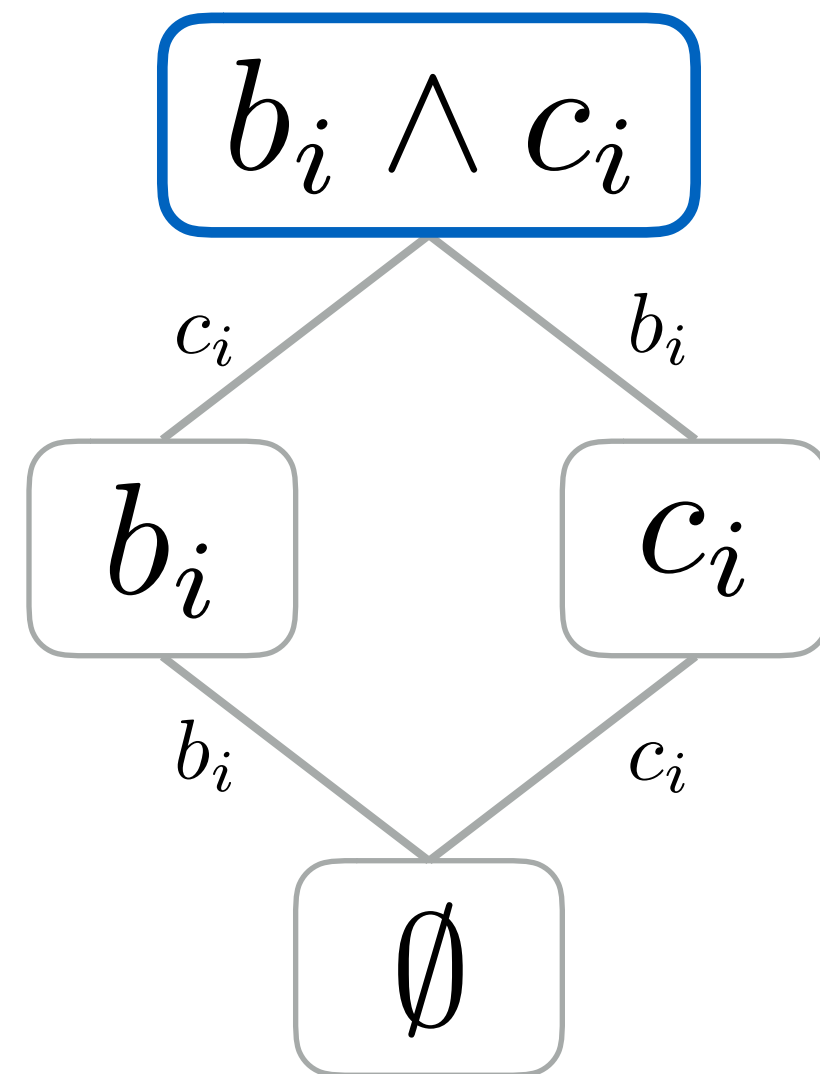
while (pb1 < b1_pos[1]) {
    int i = b1_idx[pb1];
    a[i] = b[pb1++];
}

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);

```

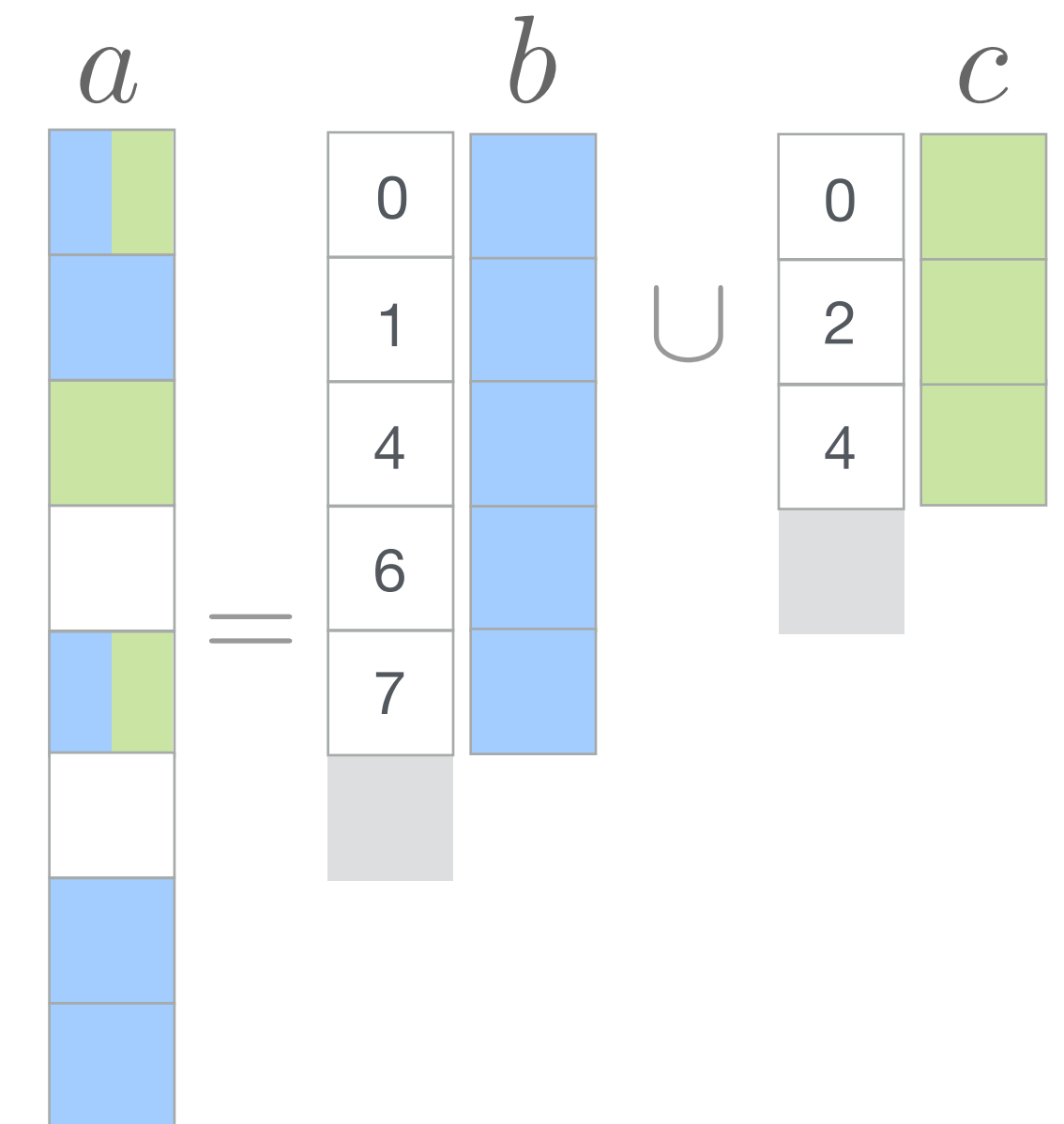
```

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

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

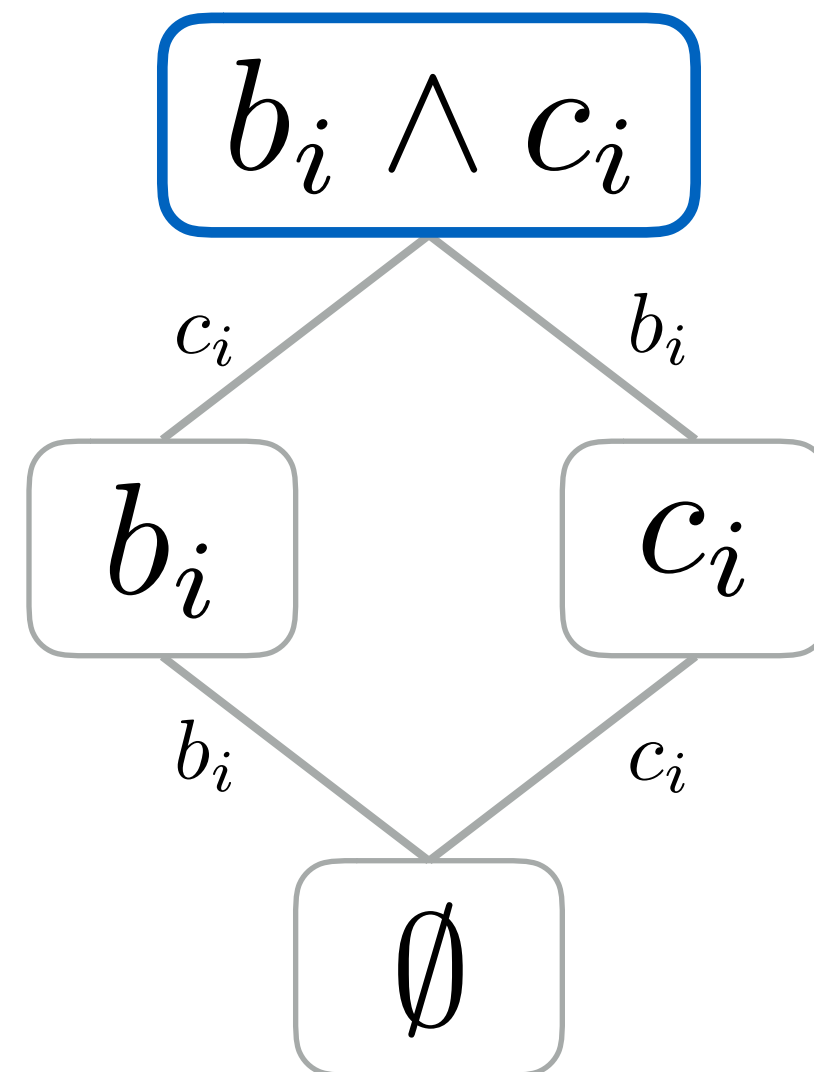
while (pc1 < c1_pos[1]) {
    int i = c1_idx[pc1];
    a[i] = c[pc1++];
}

```



Merge Lattice for a Disjunction

$$a_i = b_i + c_i$$

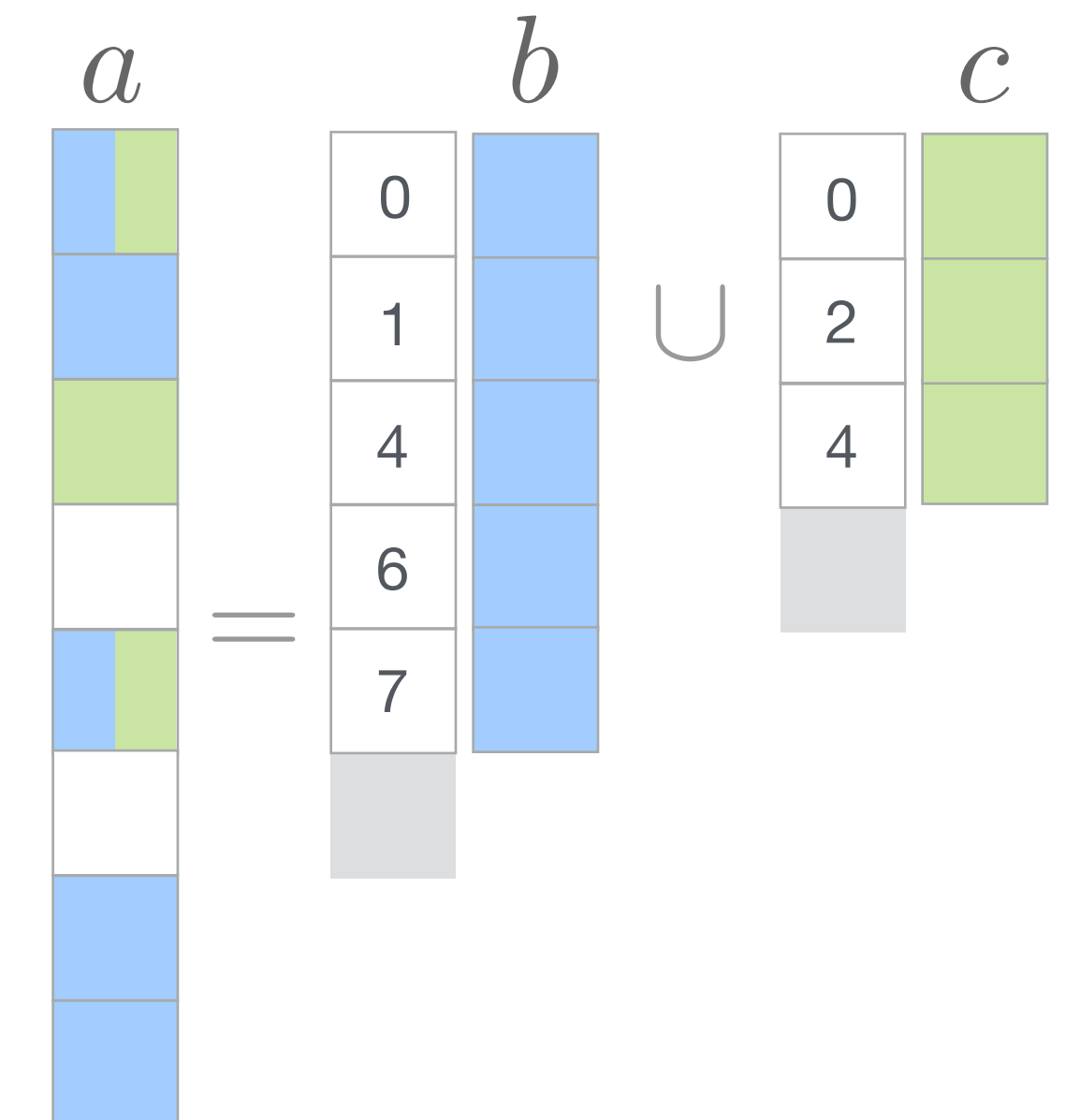


```

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++;
}

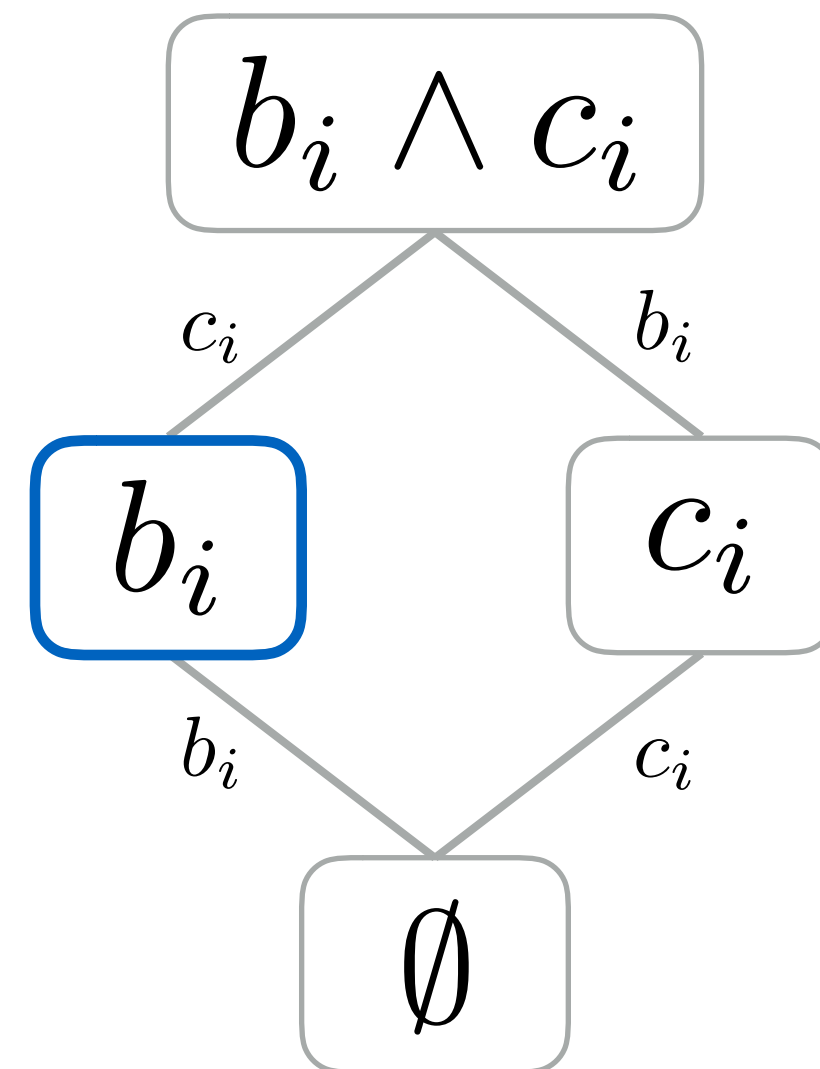
while (pb1 < b1_pos[1]) {
    int i = b1_idx[pb1];
    a[i] = b[pb1++];
}

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



Merge Lattice for a Disjunction

$$a_i = b_i + c_i$$



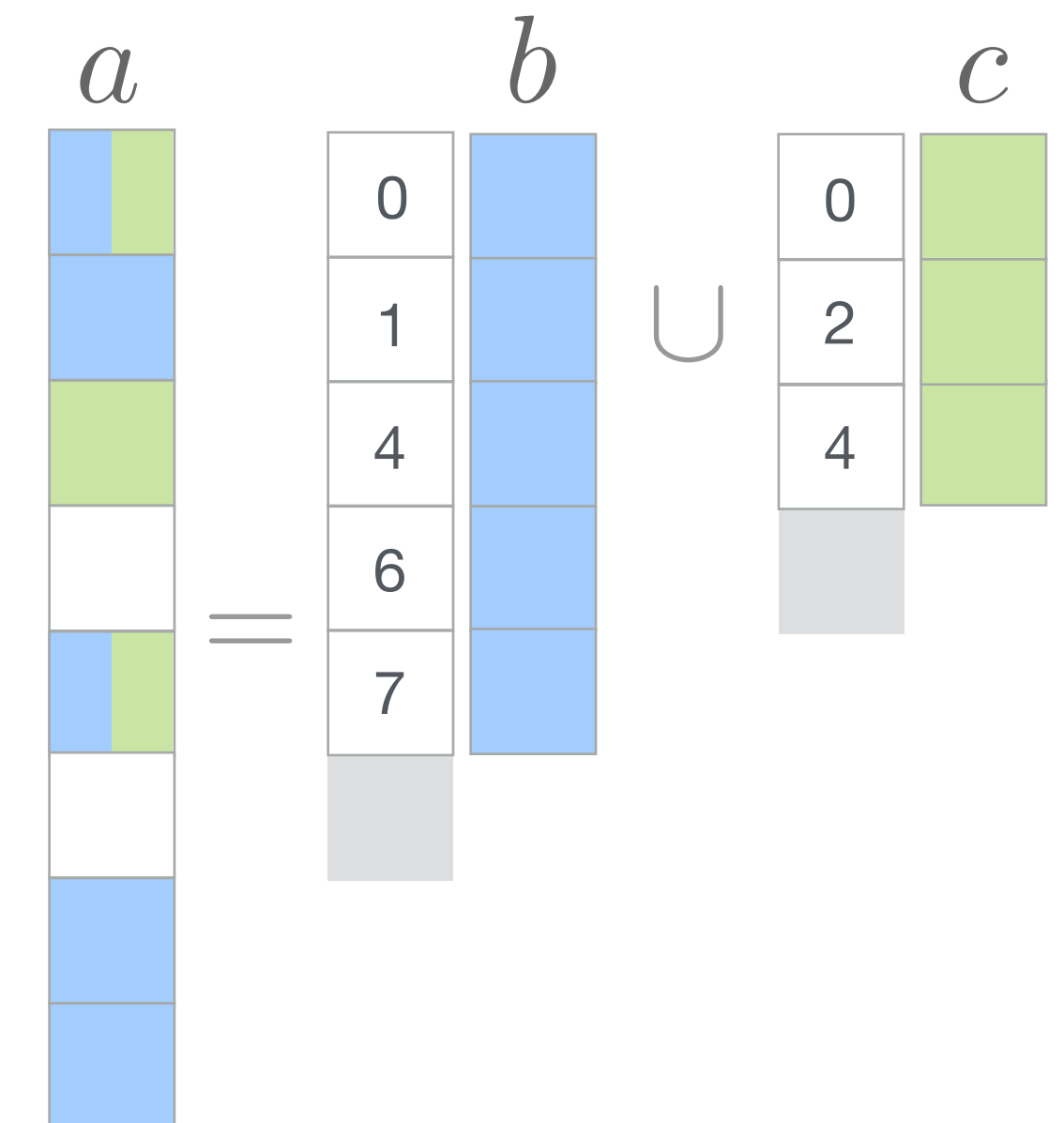
```

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++];
}

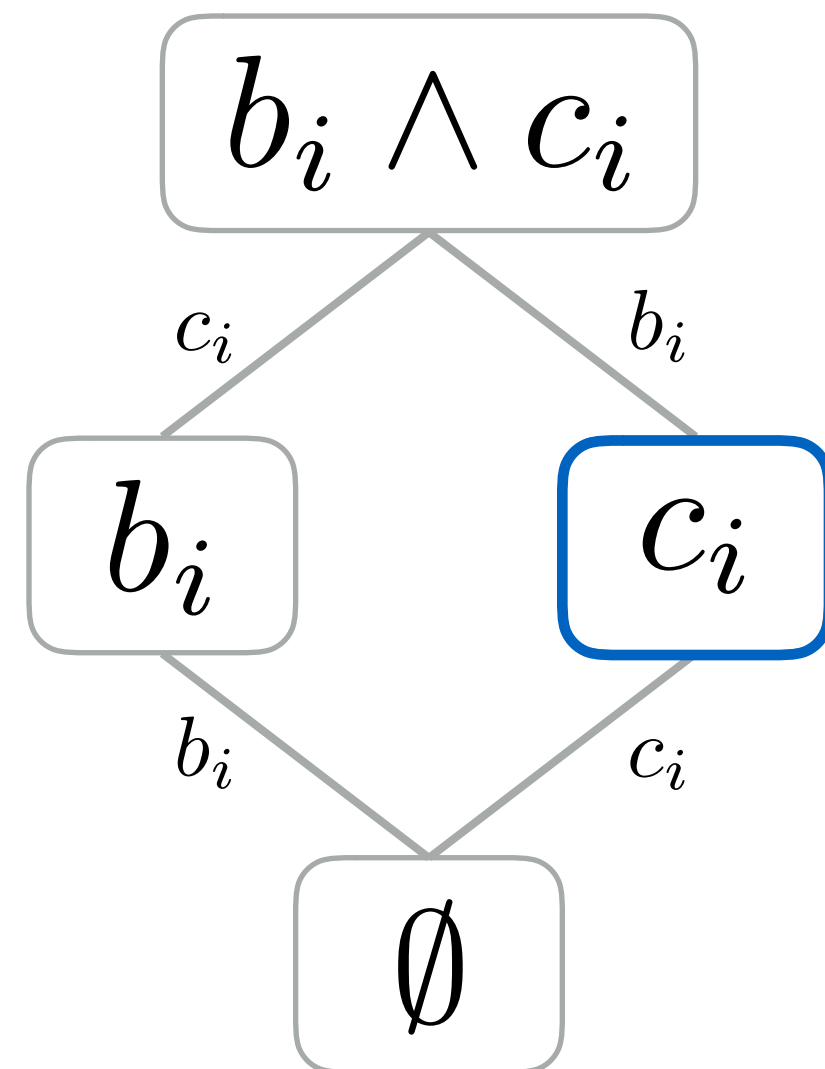
while (pc1 < c1_pos[1]) {
    int i = c1_idx[pc1];
    a[i] = c[pc1++];
}

```



Merge Lattice for a Disjunction

$$a_i = b_i + c_i$$



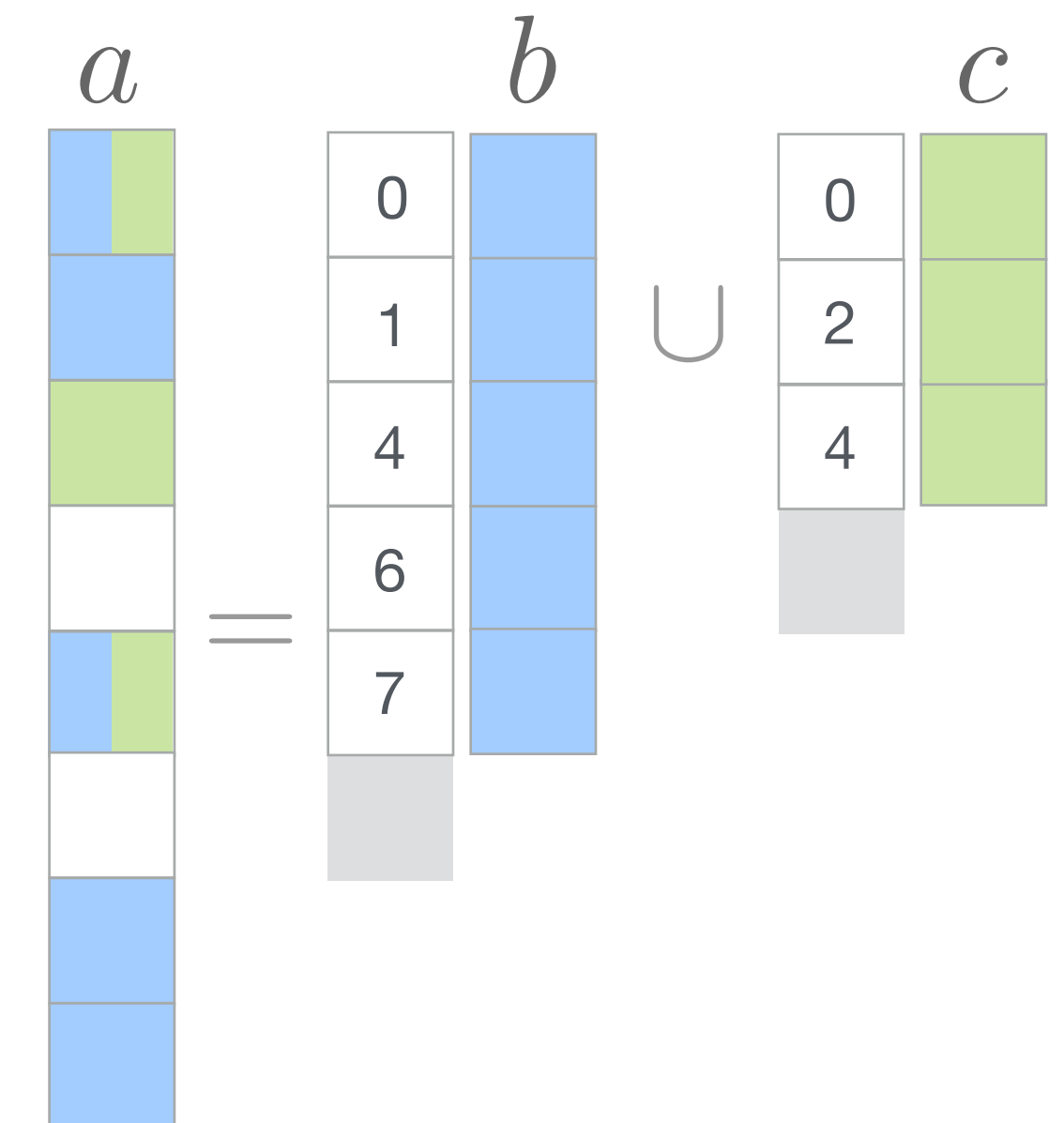
```

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++];
}

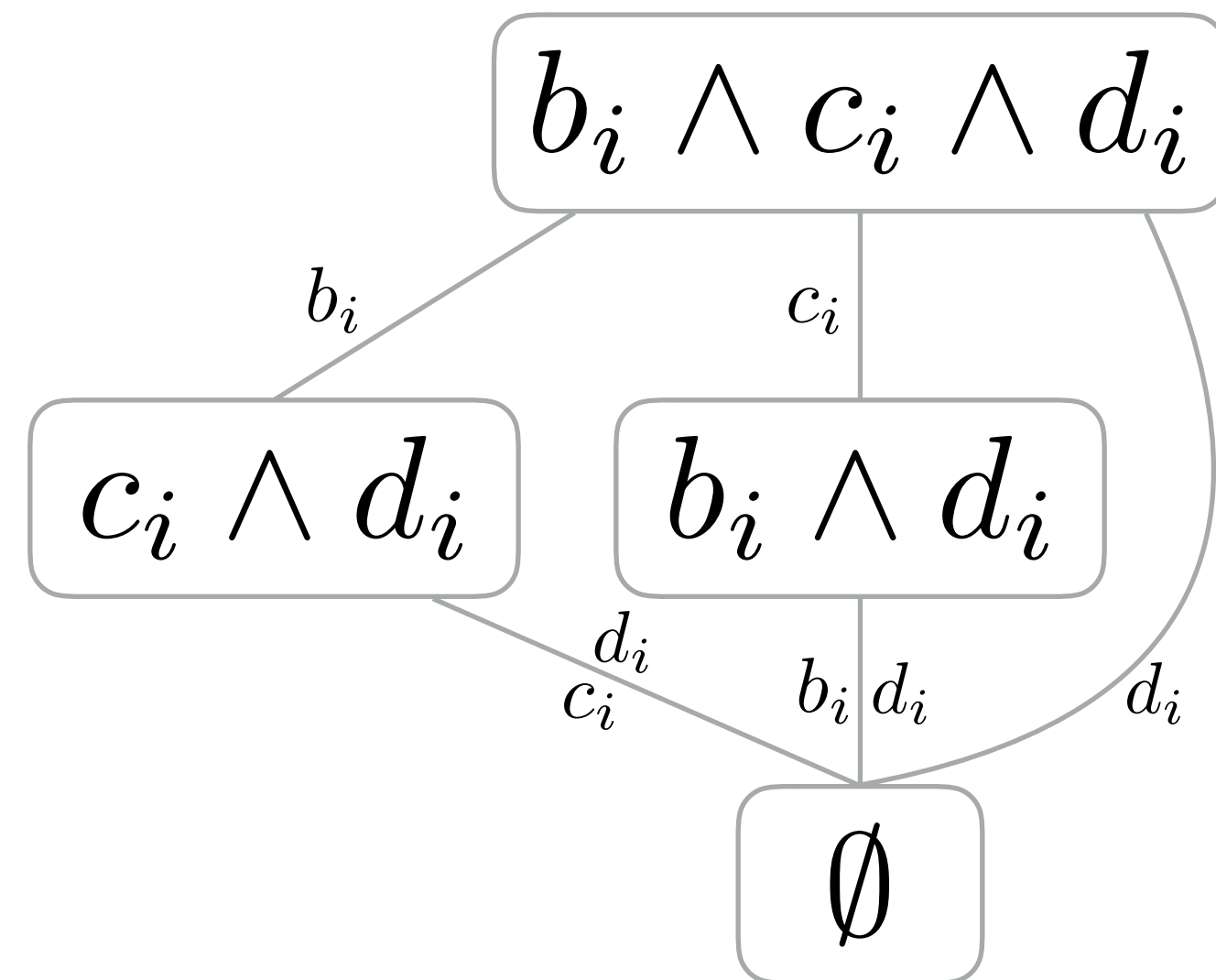
while (pc1 < c1_pos[1]) {
    int i = c1_idx[pc1];
    a[i] = c[pc1++];
}

```



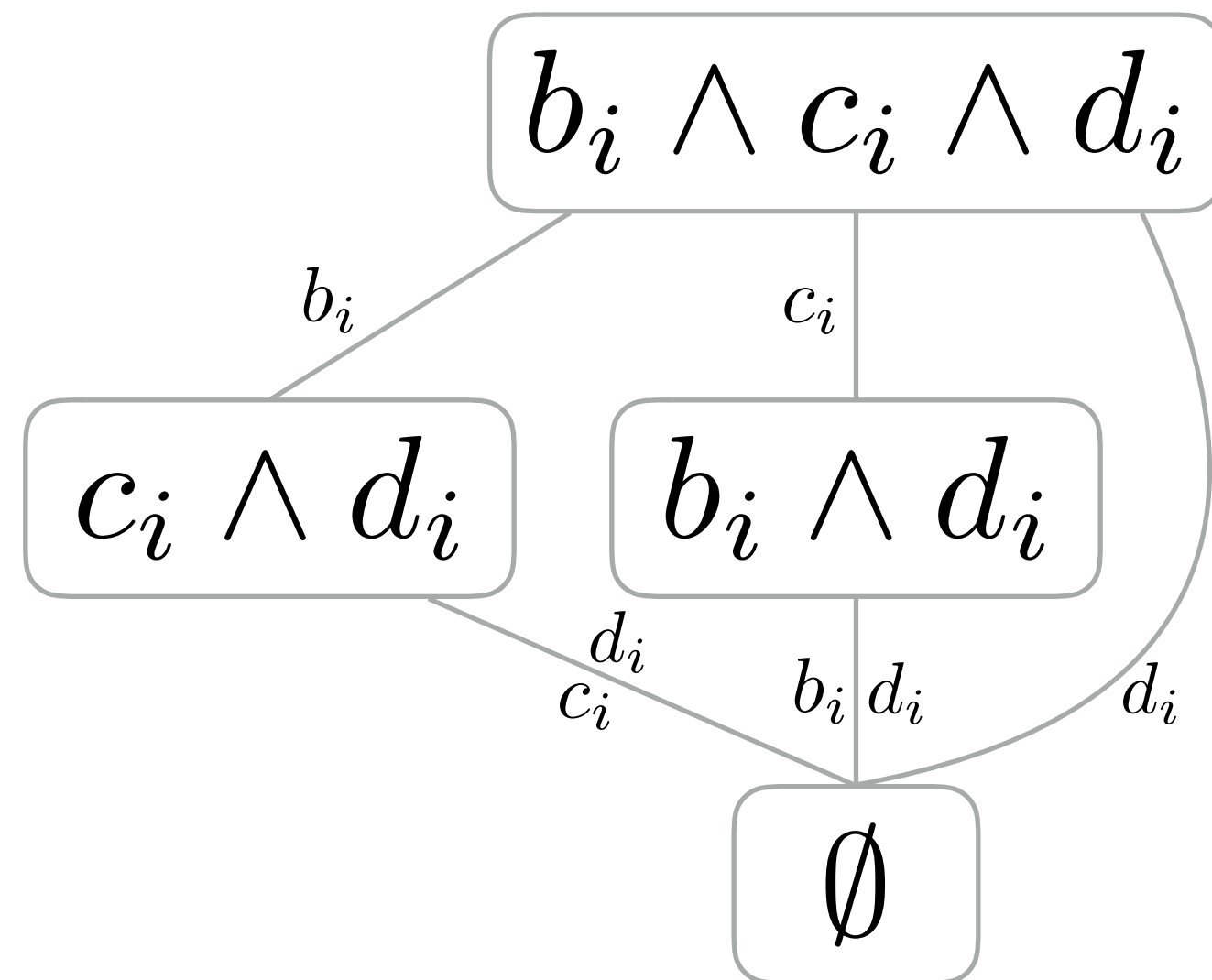
Merge Lattice for a Compound Expression

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



Merge Lattice for a Compound Expression

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



```

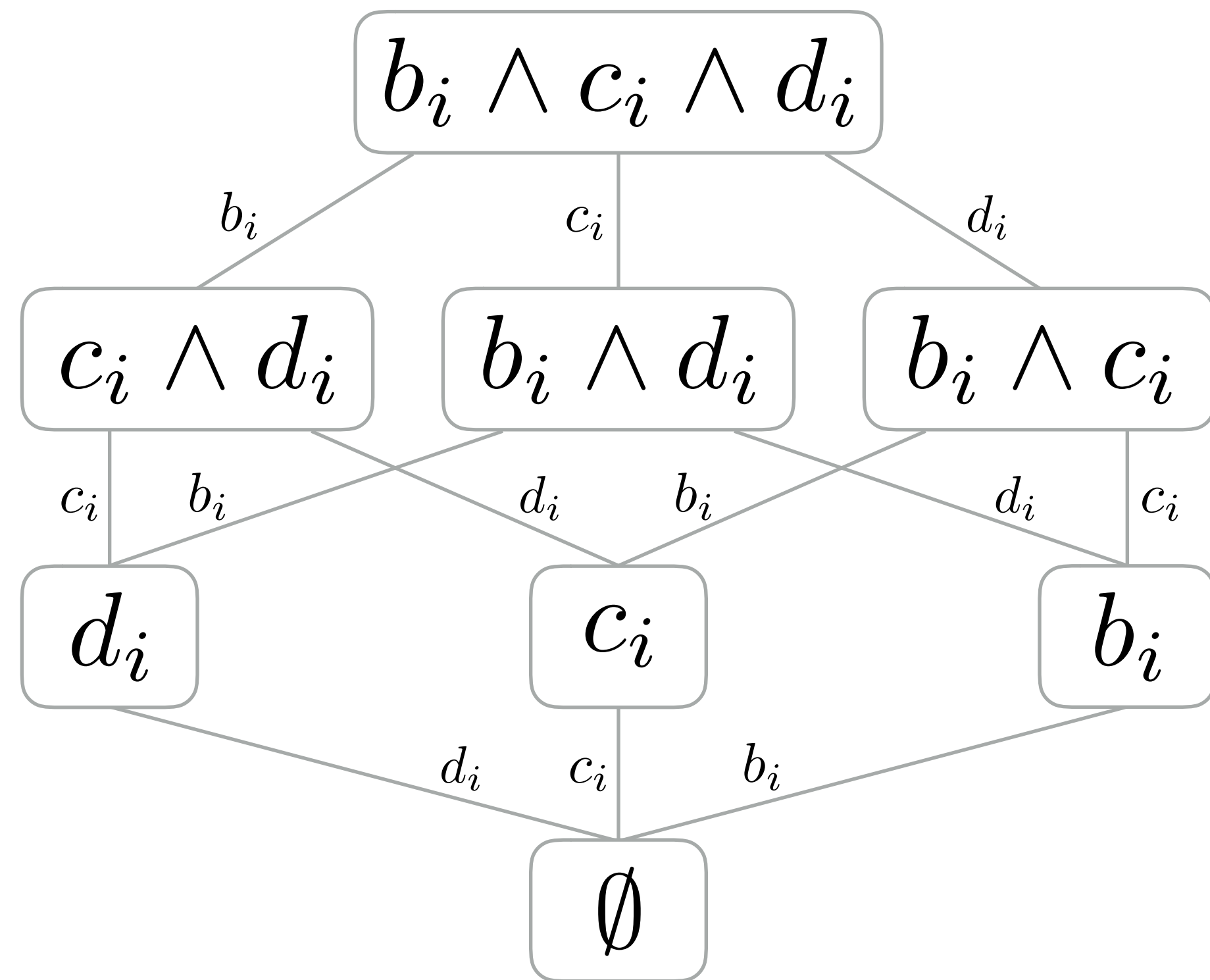
int pb1 = b1_pos[0];
int pc1 = c1_pos[0];
int pd1 = d1_pos[0];
while (pb1 < b1_pos[1] && pc1 < c1_pos[1] && pd1 < d1_pos[1]) {
    int ib = b1_idx[pb1];
    int ic = c1_idx[pc1];
    int id = d1_idx[pd1];
    int i = min(ib, ic, id);
    if (ib == i && ic == i && id == i) {
        a[i] = (b[pb1] + c[pc1]) * 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 i1 = min(ic0, id1);
    if (ic0 == i1 && id1 == i1) {
        a[i1] = c[pc1] * d[pd1];
    }
    if (ic0 == i1) pc1++;
    if (id1 == i1) 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

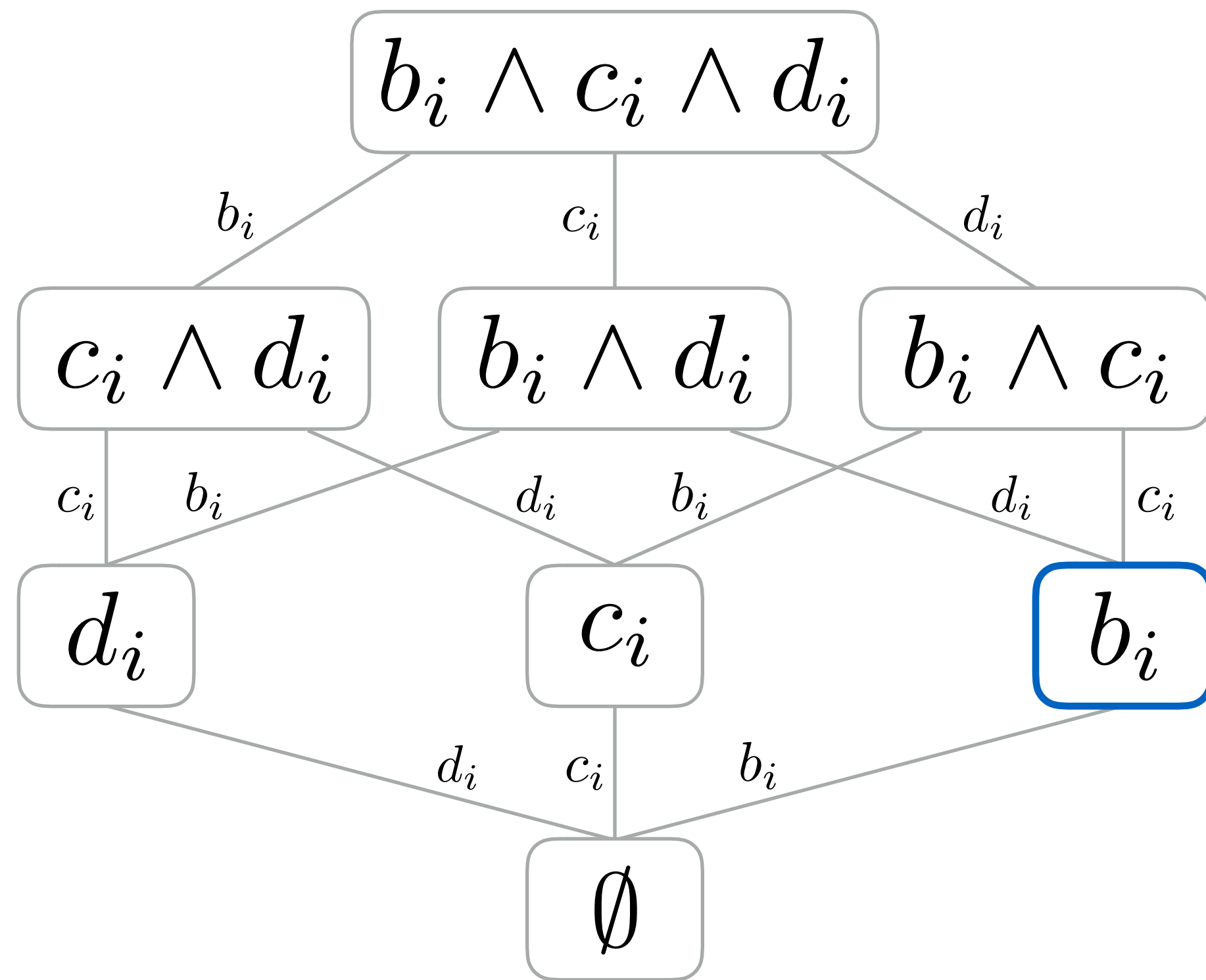
$$a_i = b_i + c_i + d_i$$



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 (pc1 < c1_pos[1] && pd1 < d1_pos[1]) {
    int ic0 = c1_idx[pc1];
    int id1 = d1_idx[pd1];
    int i1 = min(ic0, id1);
    if (ic0 == i1 && id1 == i1) {
        a[i1] = c[pc1] + d[pd1];
    }
    else if (ic0 == i1) {
        a[i1] = c[pc1];
    }
    else {
        a[i1] = d[pd1];
    }
    if (ic0 == i1) pc1++;
    if (id1 == i1) 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++;
}

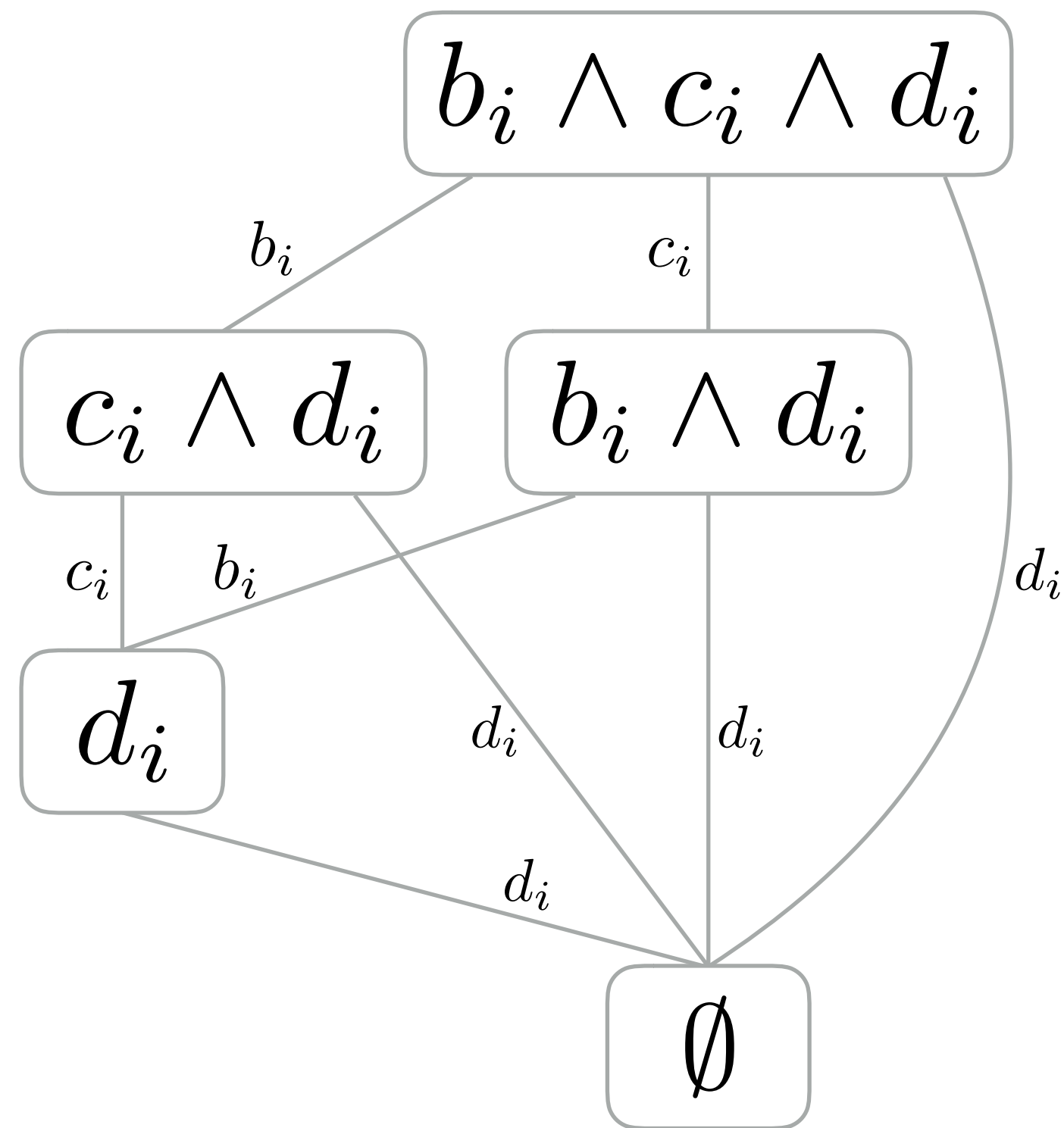
while (pc1 < c1_pos[1]) {
    int ic2 = c1_idx[pc1];
    a[ic2] = c[pc1];
    pc1++;
}

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



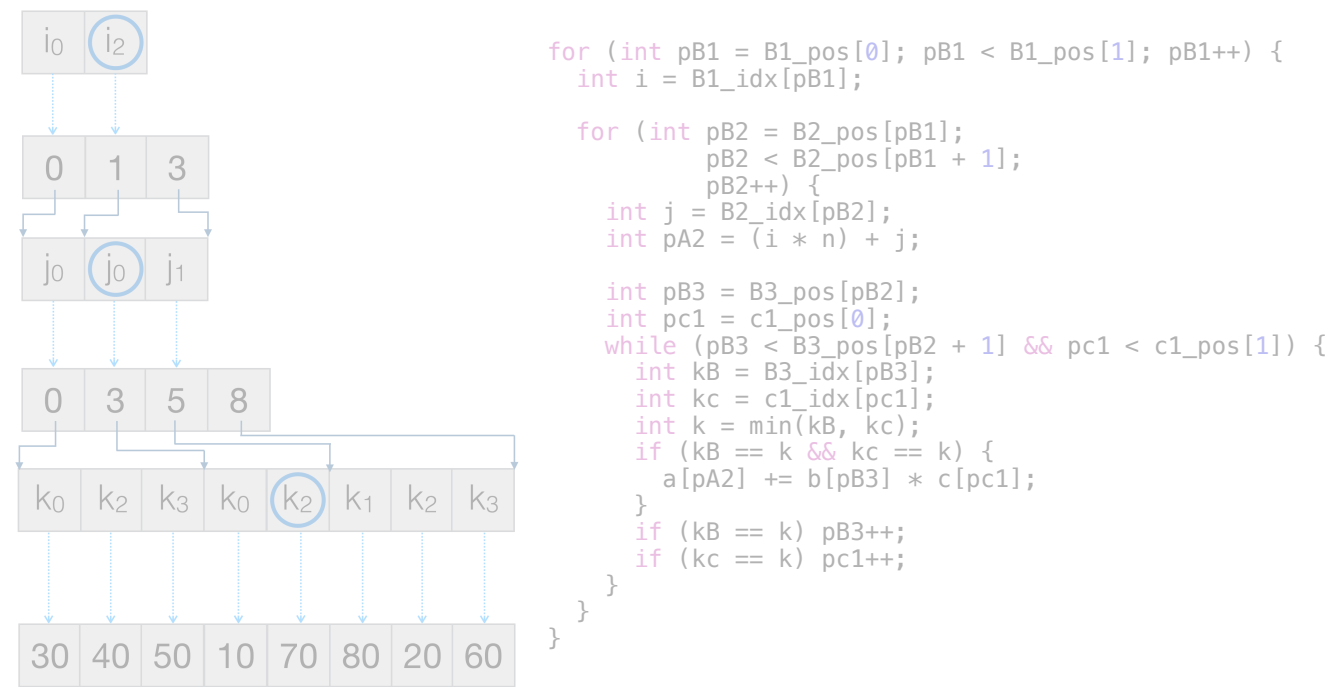
```
int pb1 = b1_pos[0];
int pc1 = c1_pos[0];
int id = 0;
while (pb1 < b1_pos[1] && pc1 < c1_pos[1]) {
    int ib = b1_idx[pb1];
    int ic = c1_idx[pc1];
    int pd1 = id;
    int pa1 = id;
    if (ib == id && ic == id) {
        a[pa1] = b[pb1] + c[pc1] + d[pd1];
    }
    else if (ib == id) {
        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 (pc1 < c1_pos[1]) {
    int ic0 = c1_idx[pc1];
    int pd11 = id;
    int pa11 = id;
    if (ic0 == id) {
        a[pa11] = c[pc1] + d[pd11];
    }
    else {
        a[pa11] = d[pd11];
    }
    if (ic0 == id) pc1++;
    id++;
}
```

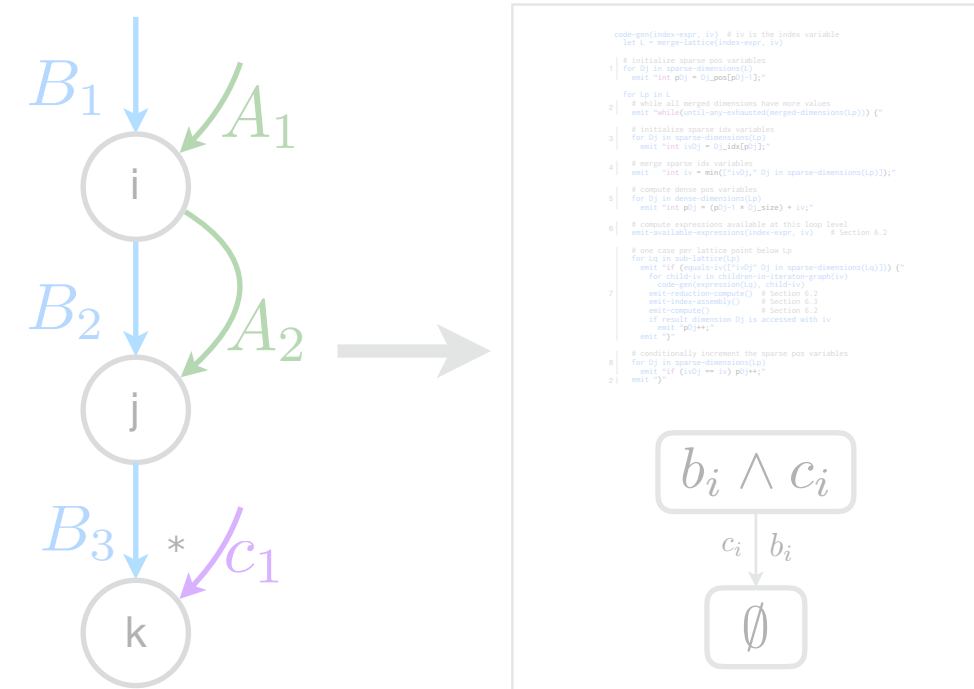
```
while (pb1 < b1_pos[1]) {
    int ib0 = b1_idx[pb1];
    int pd10 = id;
    int pa10 = id;
    if (ib0 == id) {
        a[pa10] = b[pb1] + d[pd10];
    }
    else {
        a[pa10] = d[pd10];
    }
    if (ib0 == id) pb1++;
    id++;
}
```

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

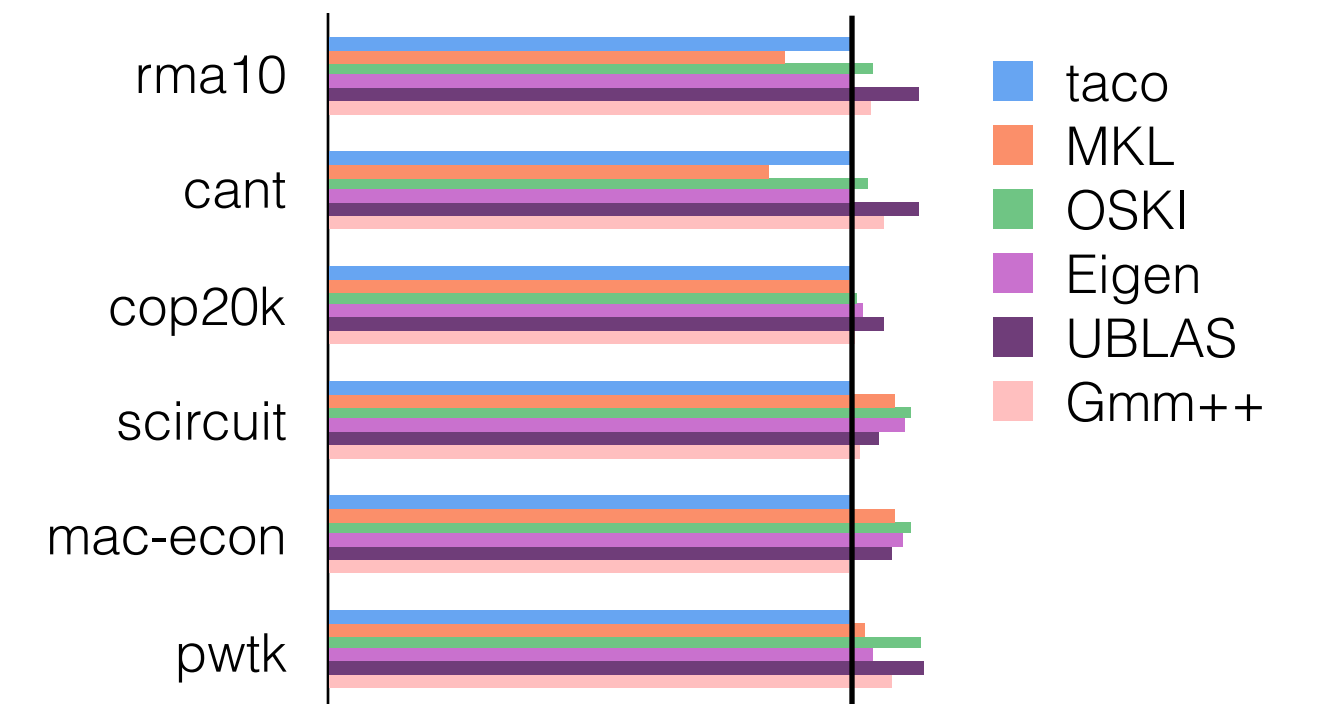
Sparse code and data



Intermediate Representation and Code Generation



Evaluation



The Tensor Algebra Compiler is fast and general

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

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

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

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

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

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

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

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

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

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

The Tensor Algebra Compiler is fast and general

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

Sparse Matrix-Vector Multiplication (SpMV)

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

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

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

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

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

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

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

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

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

The Tensor Algebra Compiler is fast and general

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

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

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

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

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

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

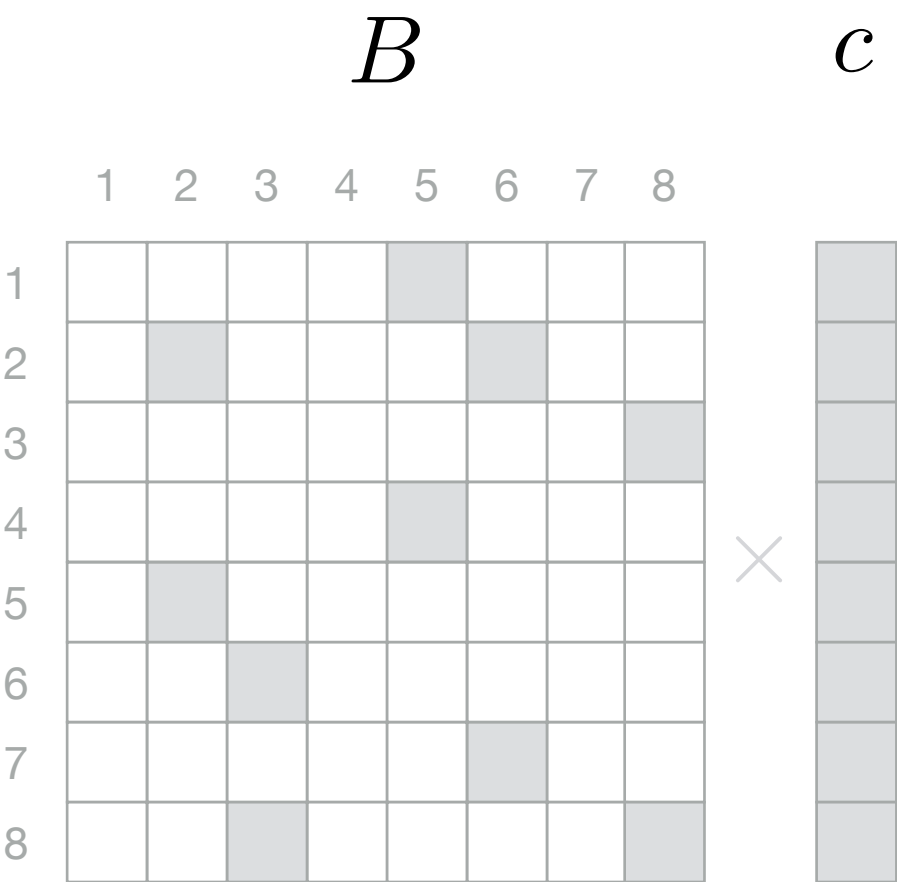
$$A_{ik} = \sum_j B_{ijk} * c_j \quad A = 0 \quad A = B + C + D \quad A = (B + C)(d + e)$$

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

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

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

Sparse Matrix-Vector Multiplication (SpMV)



The Tensor Algebra Compiler is fast and general

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

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

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

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

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

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

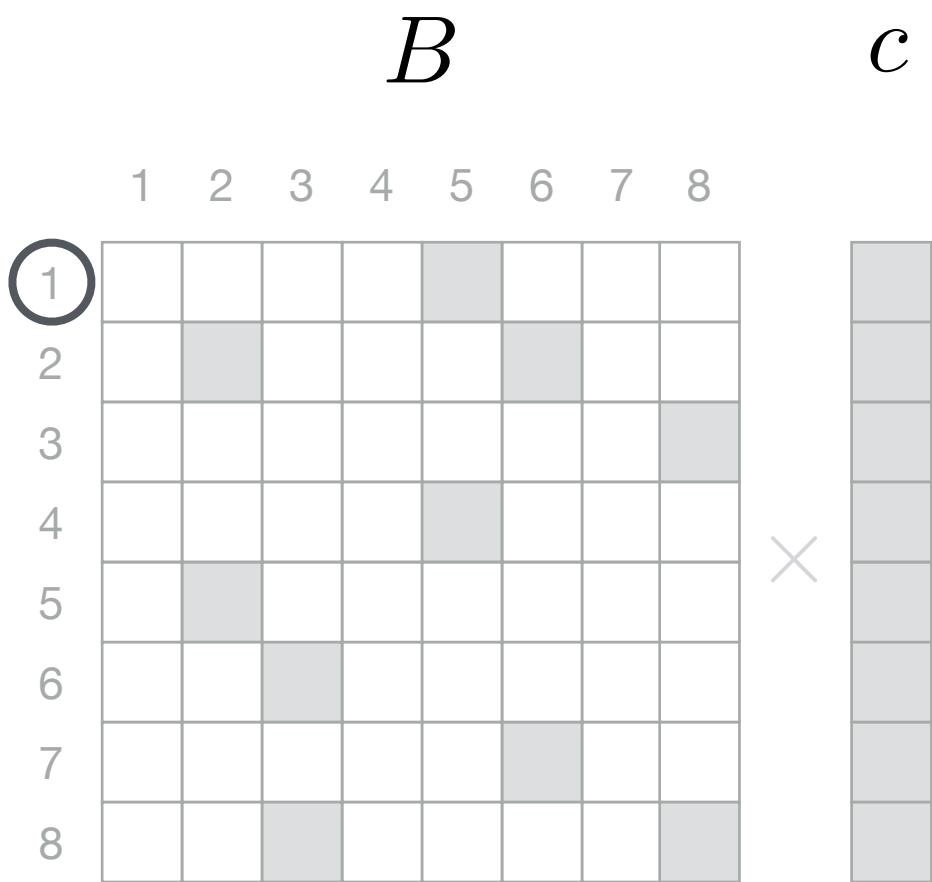
$$A_{ik} = \sum_j B_{ijk} * c_j \quad A = 0 \quad A = B + C + D \quad A = (B + C)(d + e)$$

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

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

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

Sparse Matrix-Vector Multiplication (SpMV)



The Tensor Algebra Compiler is fast and general

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

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

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

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

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

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

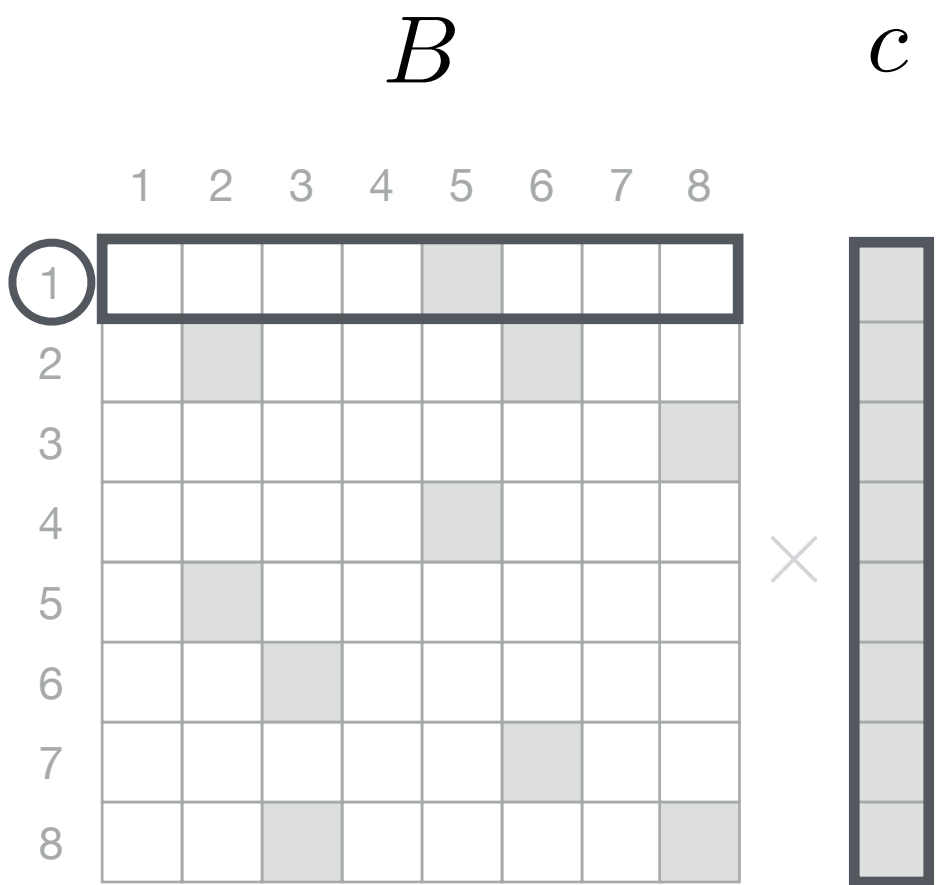
$$A_{ik} = \sum_j B_{ijk} * c_j \quad A = 0 \quad A = B + C + D \quad A = (B + C)(d + e)$$

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

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

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

Sparse Matrix-Vector Multiplication (SpMV)



SpMV is competitive with hand-optimized implementations

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

$a = Bc$

$a = B^T c$

$A = B + C + D$

$y = b - Ax$

$A = B$

$a = Bc + a$

$A_{ijk} = \sum_l B_{ijl} C_{kl}$

$A = B \circ (CD)$

$A = B \odot C$

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

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

$A = B^T$

$\alpha = \sum_{i,j,k} B_{ijk} C_{ijk}$

$A_{ij} = \sum_{k,l} B_{ikl} C_{kj} D_{lj}$

$A = B + C$

$a = B^T c + d$

$a = b + c$

$A = \alpha A + \beta B$

$A_{ik} = \sum_j B_{ijk} * c_j$

$A = 0$

$A = B + C + D$

$A = (B + C)(d + e)$

$A = \alpha B$

$A = (B + C)d$

$A = BC$

$A_{ilk} = \sum_j B_{ijk} * C_{lj}$

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

$a = BCd$

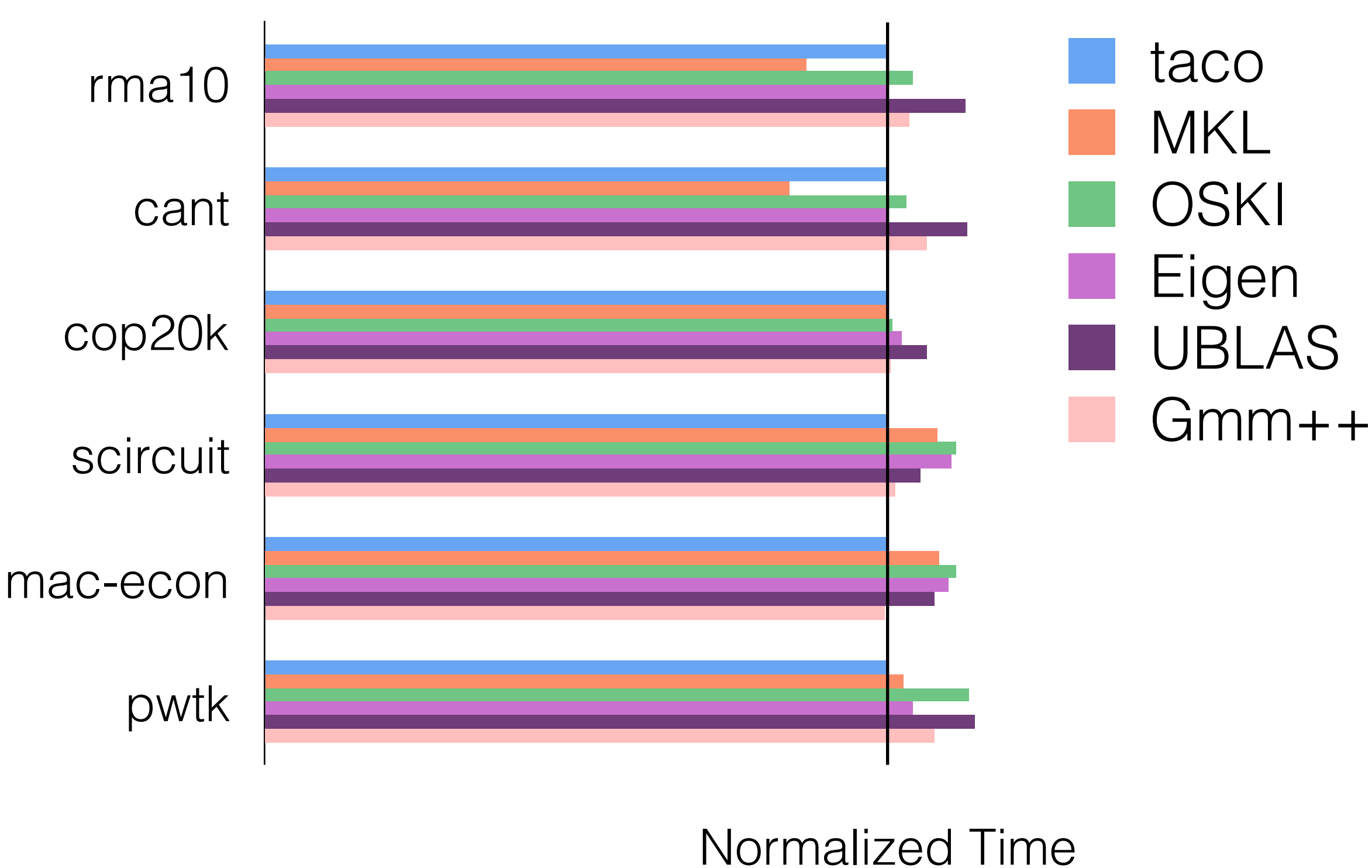
$a = Bc + b$

$A = A + \alpha I$

$A = \alpha B + A$

$A_{jl} = \sum_{i,k} B_{ijk} * C_{il} * D_{kl}$

$a = \alpha Bc + \beta a$



SpMV is competitive with hand-optimized implementations

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

$a = Bc$

$a = B^T c$

$A = B + C + D$

$y = b - Ax$

$A = B$

$a = Bc + a$

$A_{ijk} = \sum_l B_{ijl} C_{kl}$

$A = B \circ (CD)$

$A = B \odot C$

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

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

$A = B^T$

$\alpha = \sum_{i,j,k} B_{ijk} C_{ijk}$

$A_{ij} = \sum_{k,l} B_{ikl} C_{kj} D_{lj}$

$A = B + C$

$a = B^T c + d$

$a = b + c$

$A = \alpha A + \beta B$

$A_{ik} = \sum_j B_{ijk} * c_j$

$A = 0$

$A = B + C + D$

$A = (B + C)(d + e)$

$A = \alpha B$

$A = (B + C)d$

$A = BC$

$A_{ilk} = \sum_j B_{ijk} * C_{lj}$

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

$a = BCd$

$a = Bc + b$

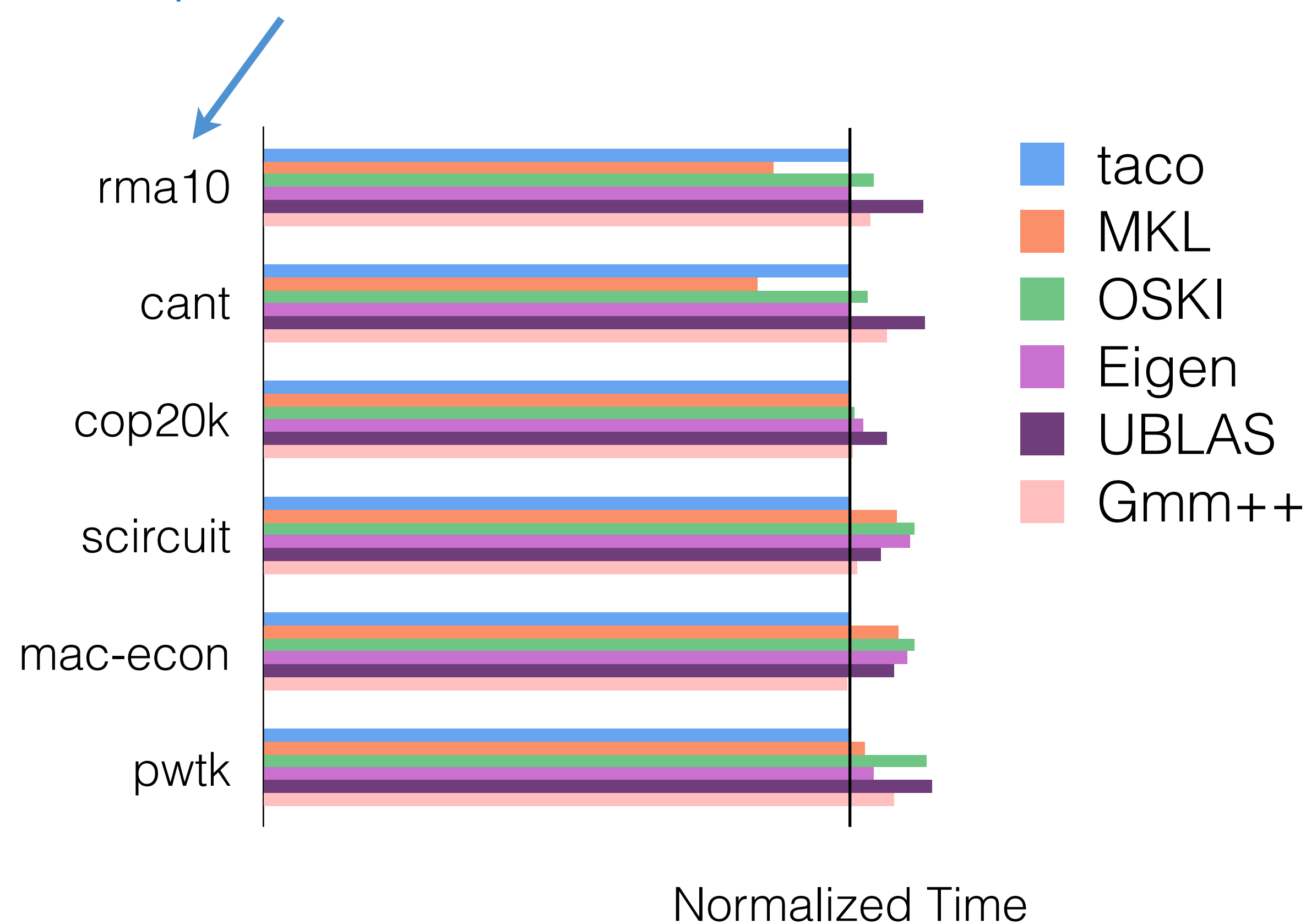
$A = A + \alpha I$

$A = \alpha B + A$

$A_{jl} = \sum_{i,k} B_{ijk} * C_{il} * D_{kl}$

$a = \alpha Bc + \beta a$

Florida Sparse Matrix Collection



SpMV is competitive with hand-optimized implementations

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

$a = Bc$

$a = B^T c$

$A = B + C + D$

$y = b - Ax$

$A = B$

$a = Bc + a$

$A_{ijk} = \sum_l B_{ijl} C_{kl}$

$A = B \circ (CD)$

$A = B \odot C$

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

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

$A = B^T$

$\alpha = \sum_{i,j,k} B_{ijk} C_{ijk}$

$A_{ij} = \sum_{k,l} B_{ikl} C_{kj} D_{lj}$

$A = B + C$

$a = B^T c + d$

$a = b + c$

$A = \alpha A + \beta B$

$A_{ik} = \sum_j B_{ijk} * c_j$

$A = 0$

$A = B + C + D$

$A = (B + C)(d + e)$

$A = \alpha B$

$A = (B + C)d$

$A = BC$

$A_{ilk} = \sum_j B_{ijk} * C_{lj}$

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

$a = BCd$

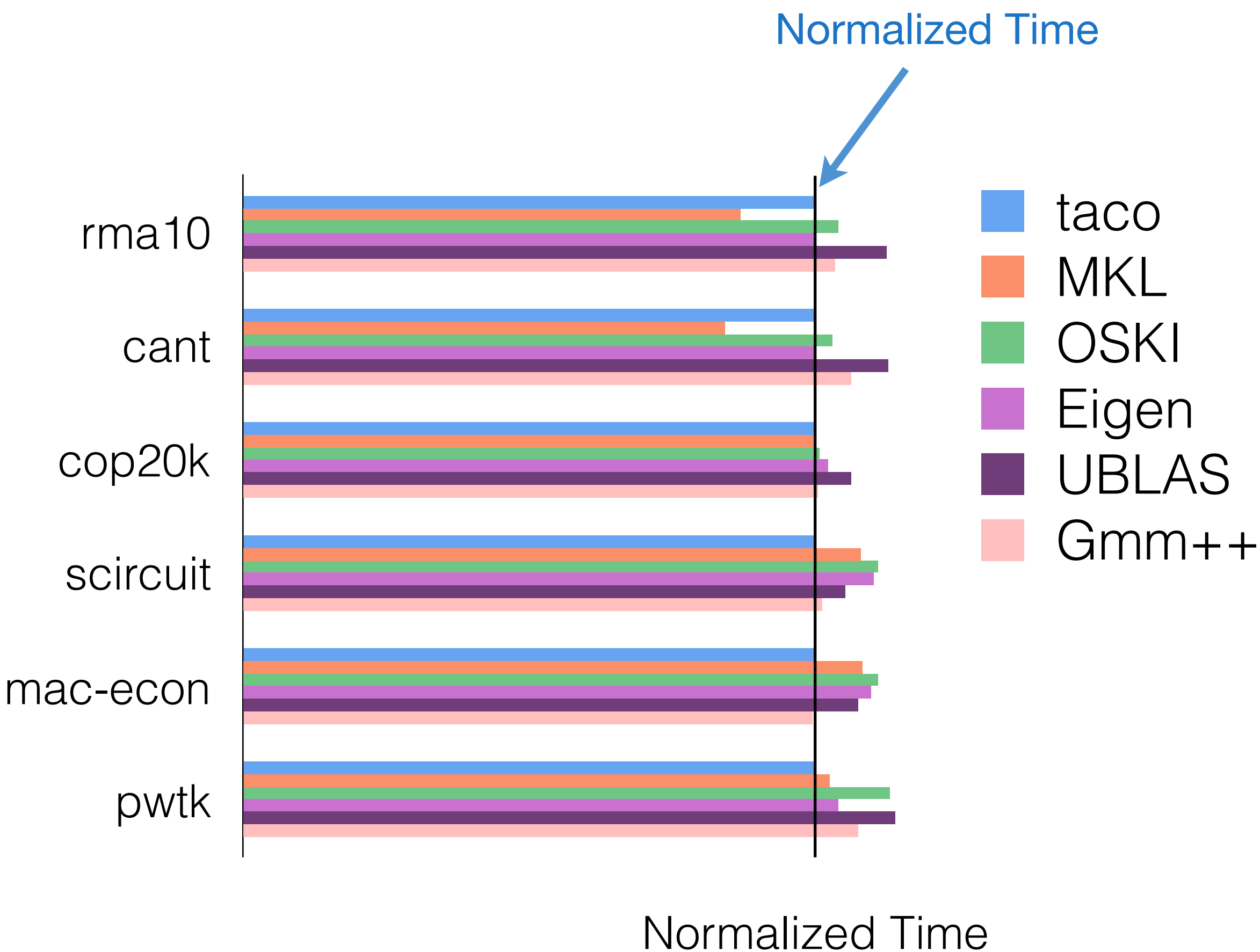
$a = Bc + b$

$A = A + \alpha I$

$A = \alpha B + A$

$A_{jl} = \sum_{i,k} B_{ijk} * C_{il} * D_{kl}$

$a = \alpha Bc + \beta a$



SpMV is parallel

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

$a = Bc$

$a = B^T c$

$A = B + C + D$

$y = b - Ax$

$A = B$

$a = Bc + a$

$A_{ijk} = \sum_l B_{ijl} C_{kl}$

$A = B \circ (CD)$

$A = B \odot C$

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

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

$A = B^T$

$\alpha = \sum_{i,j,k} B_{ijk} C_{ijk}$

$A_{ij} = \sum_{k,l} B_{ikl} C_{kj} D_{lj}$

$A = B + C$

$a = B^T c + d$

$a = b + c$

$A = \alpha A + \beta B$

$A_{ik} = \sum_j B_{ijk} * c_j$

$A = 0$

$A = B + C + D$

$A = (B + C)(d + e)$

$A = \alpha B$

$A = (B + C)d$

$A = BC$

$A_{ilk} = \sum_j B_{ijk} * C_{lj}$

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

$a = BCd$

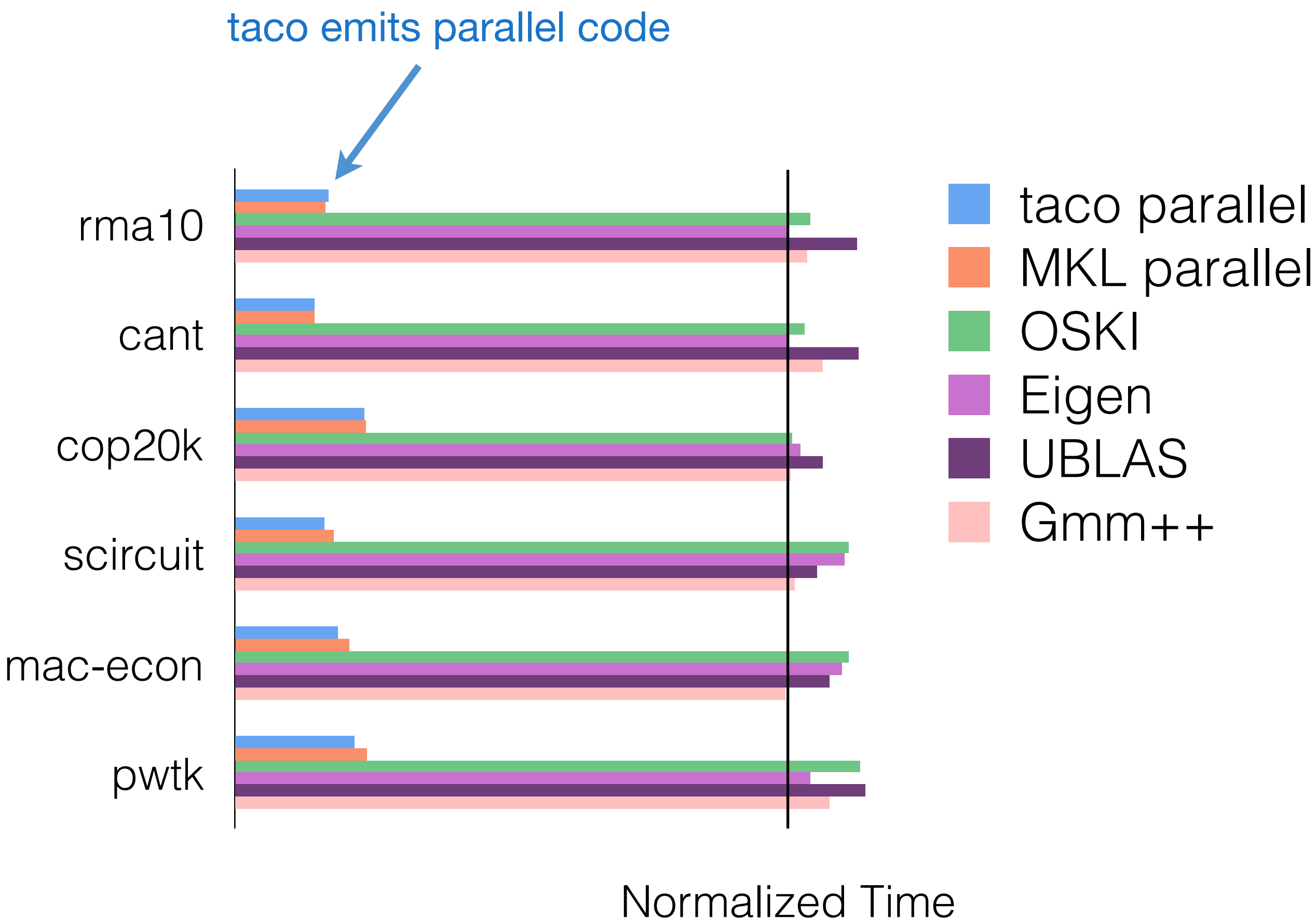
$a = Bc + b$

$A = A + \alpha I$

$A = \alpha B + A$

$A_{jl} = \sum_{i,k} B_{ijk} * C_{il} * D_{kl}$

$a = \alpha Bc + \beta a$



SDDMM must be computed in a single kernel

Sampled Dense-Dense Matrix Multiplication (SDDMM)

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

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

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

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

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

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

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

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

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

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

SDDMM must be computed in a single kernel

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

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

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

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

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

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

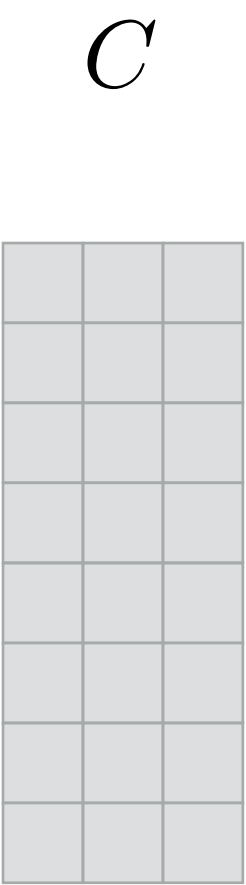
$$A_{ik} = \sum_j B_{ijk} * c_j \quad A = 0 \quad A = B + C + D \quad A = (B + C)(d + e)$$

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

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

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

Sampled Dense-Dense Matrix Multiplication (SDDMM)



SDDMM must be computed in a single kernel

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

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

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

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

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

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

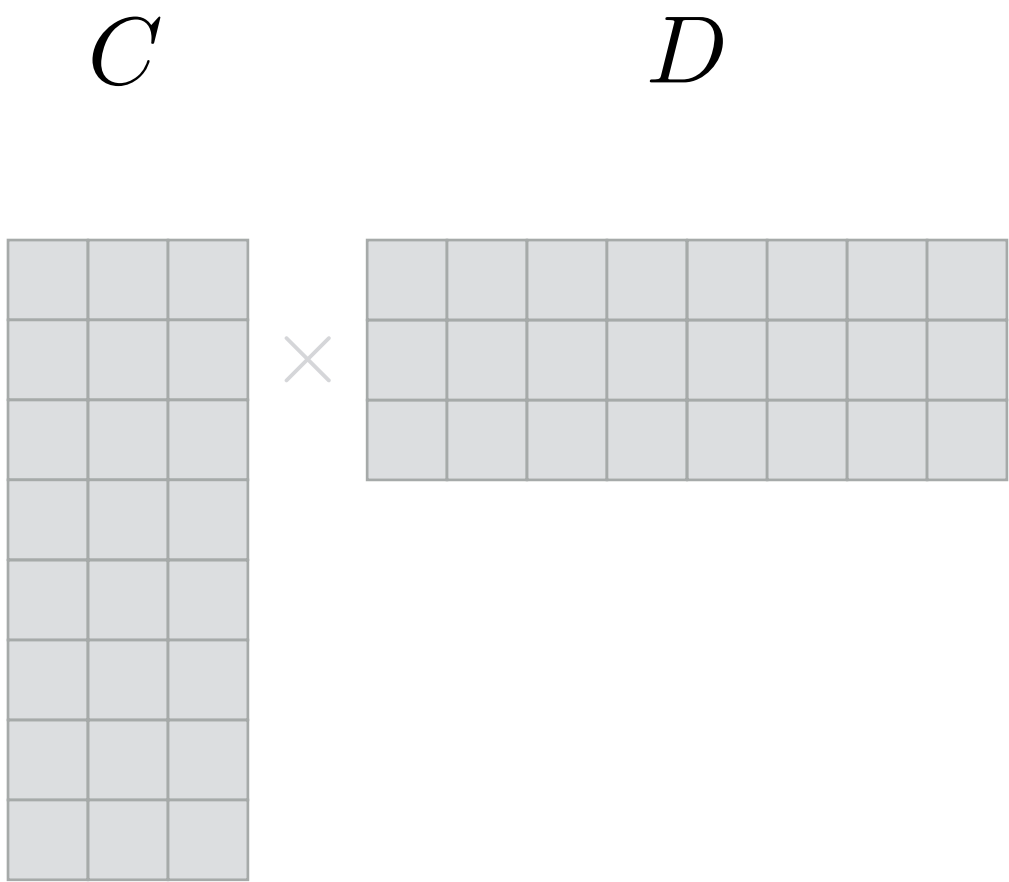
$$A_{ik} = \sum_j B_{ijk} * c_j \quad A = 0 \quad A = B + C + D \quad A = (B + C)(d + e)$$

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

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

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

Sampled Dense-Dense Matrix Multiplication (SDDMM)



SDDMM must be computed in a single kernel

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

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

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

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

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

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

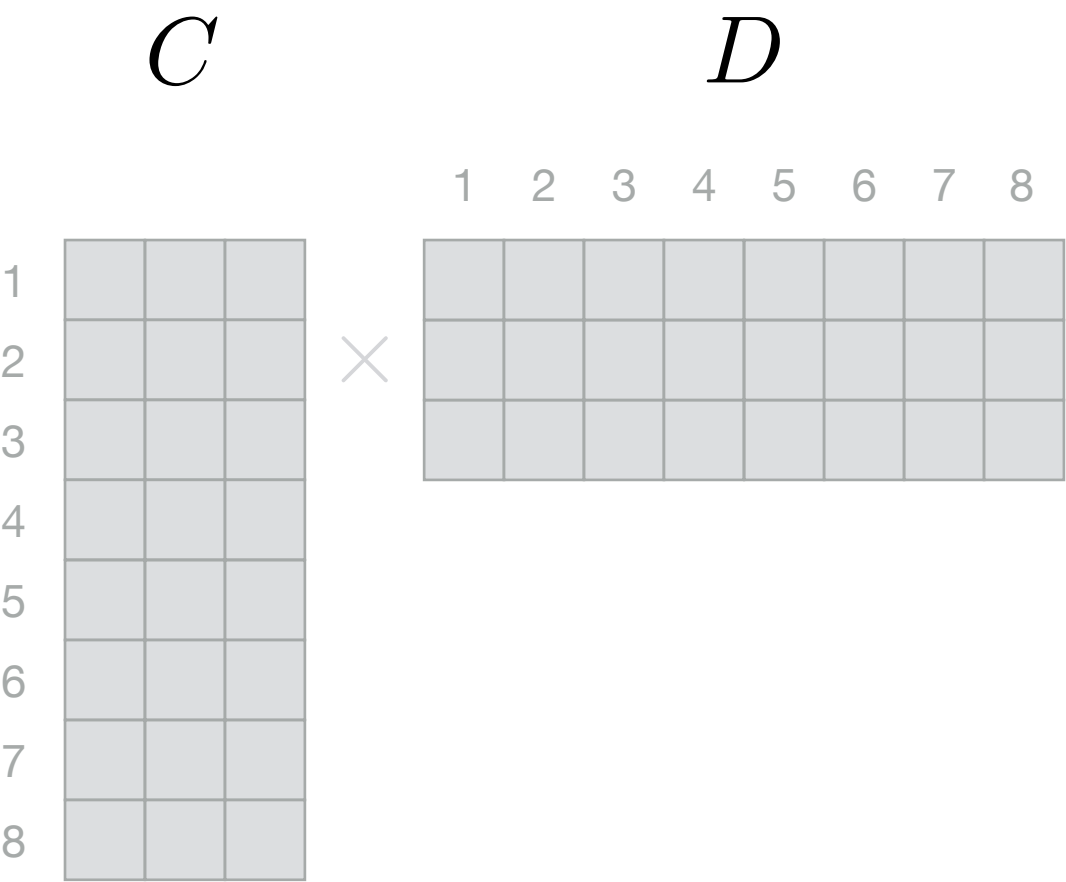
$$A_{ik} = \sum_j B_{ijk} * c_j \quad A = 0 \quad A = B + C + D \quad A = (B + C)(d + e)$$

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

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

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

Sampled Dense-Dense Matrix Multiplication (SDDMM)



64 inner product

SDDMM must be computed in a single kernel

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

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

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

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

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

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

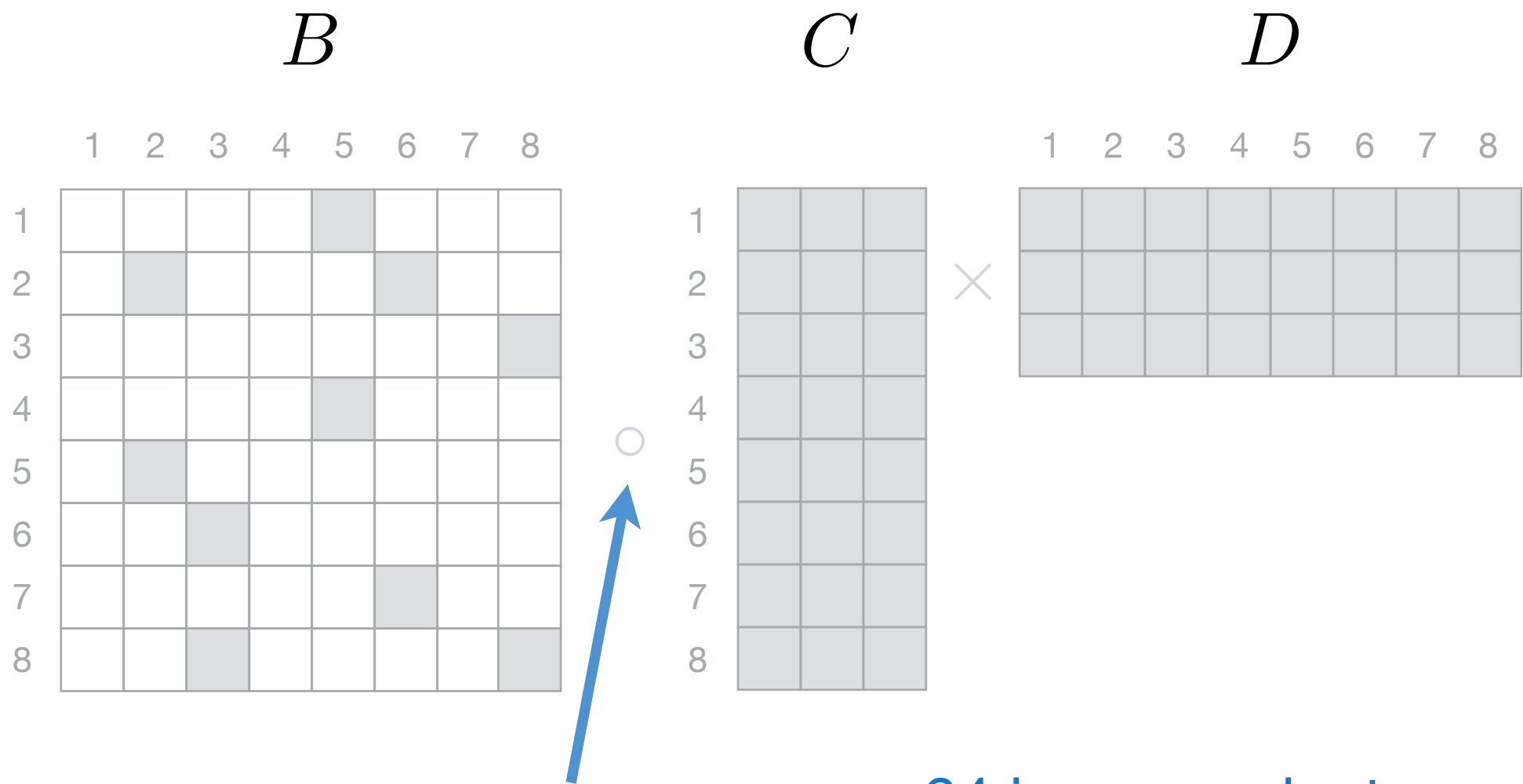
$$A_{ik} = \sum_j B_{ijk} * c_j \quad A = 0 \quad A = B + C + D \quad A = (B + C)(d + e)$$

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

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

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

Sampled Dense-Dense Matrix Multiplication (SDDMM)



component-wise multiplication

64 inner product

SDDMM must be computed in a single kernel

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

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

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

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

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

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

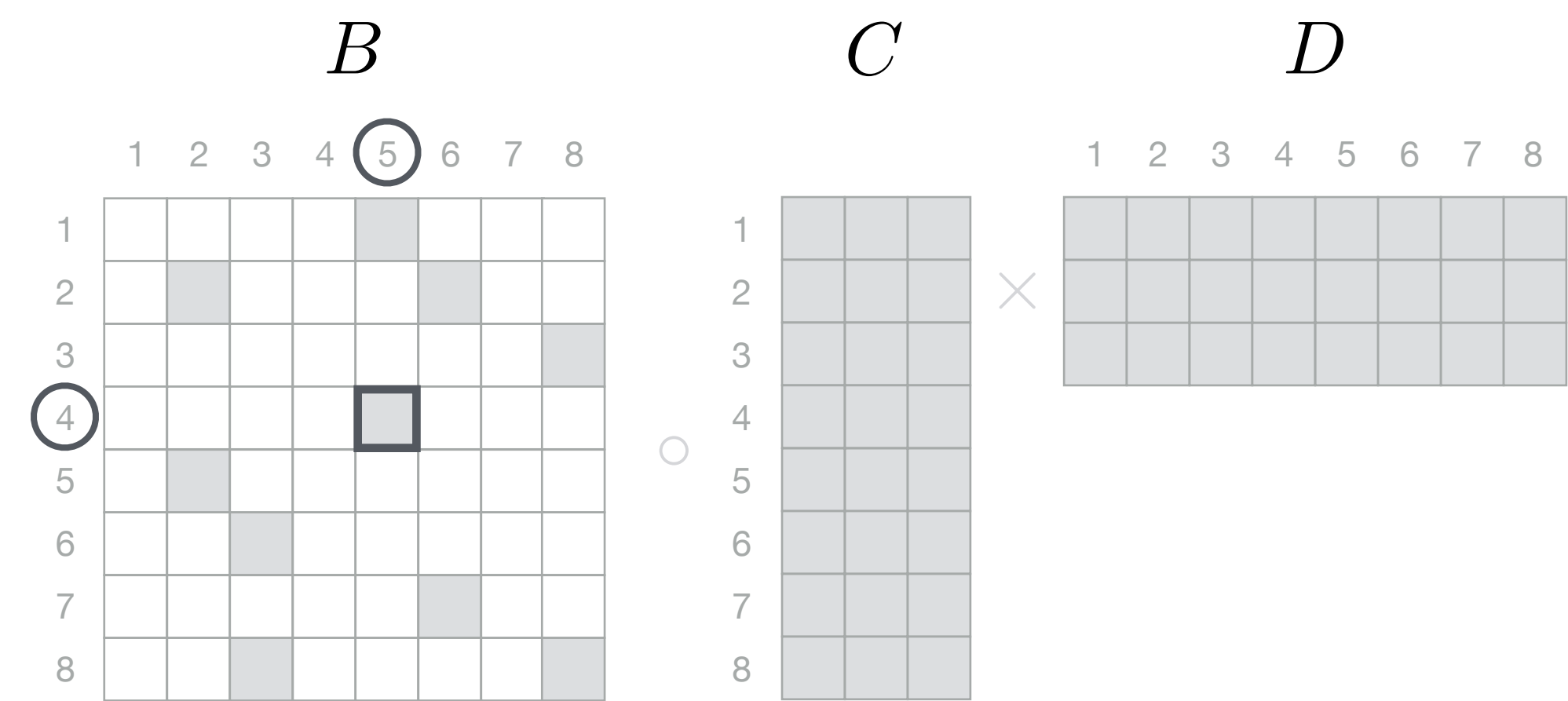
$$A_{ik} = \sum_j B_{ijk} * c_j \quad A = 0 \quad A = B + C + D \quad A = (B + C)(d + e)$$

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

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

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

Sampled Dense-Dense Matrix Multiplication (SDDMM)



64 inner product

SDDMM must be computed in a single kernel

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

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

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

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

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

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

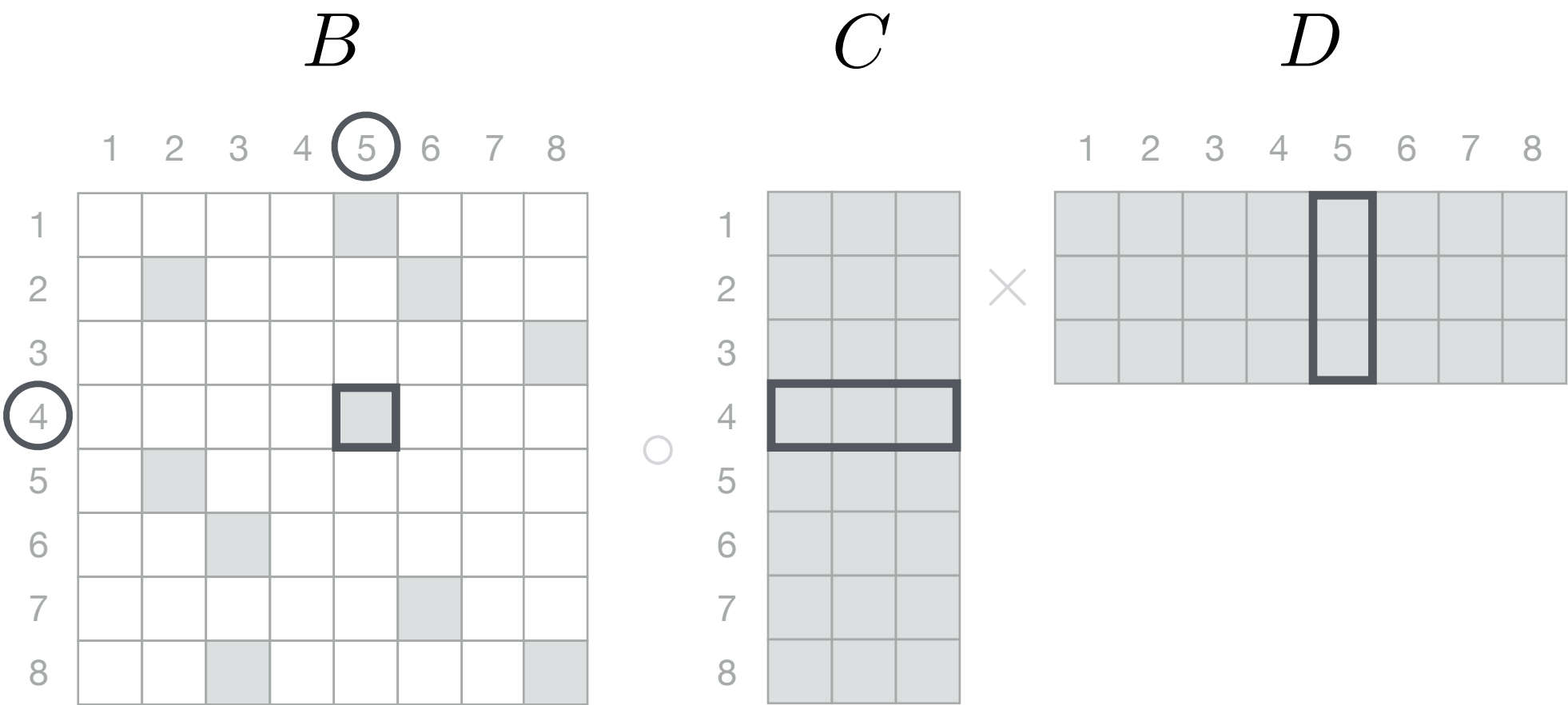
$$A_{ik} = \sum_j B_{ijk} * c_j \quad A = 0 \quad A = B + C + D \quad A = (B + C)(d + e)$$

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

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

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

Sampled Dense-Dense Matrix Multiplication (SDDMM)



64 inner product

SDDMM must be computed in a single kernel

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

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

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

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

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

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

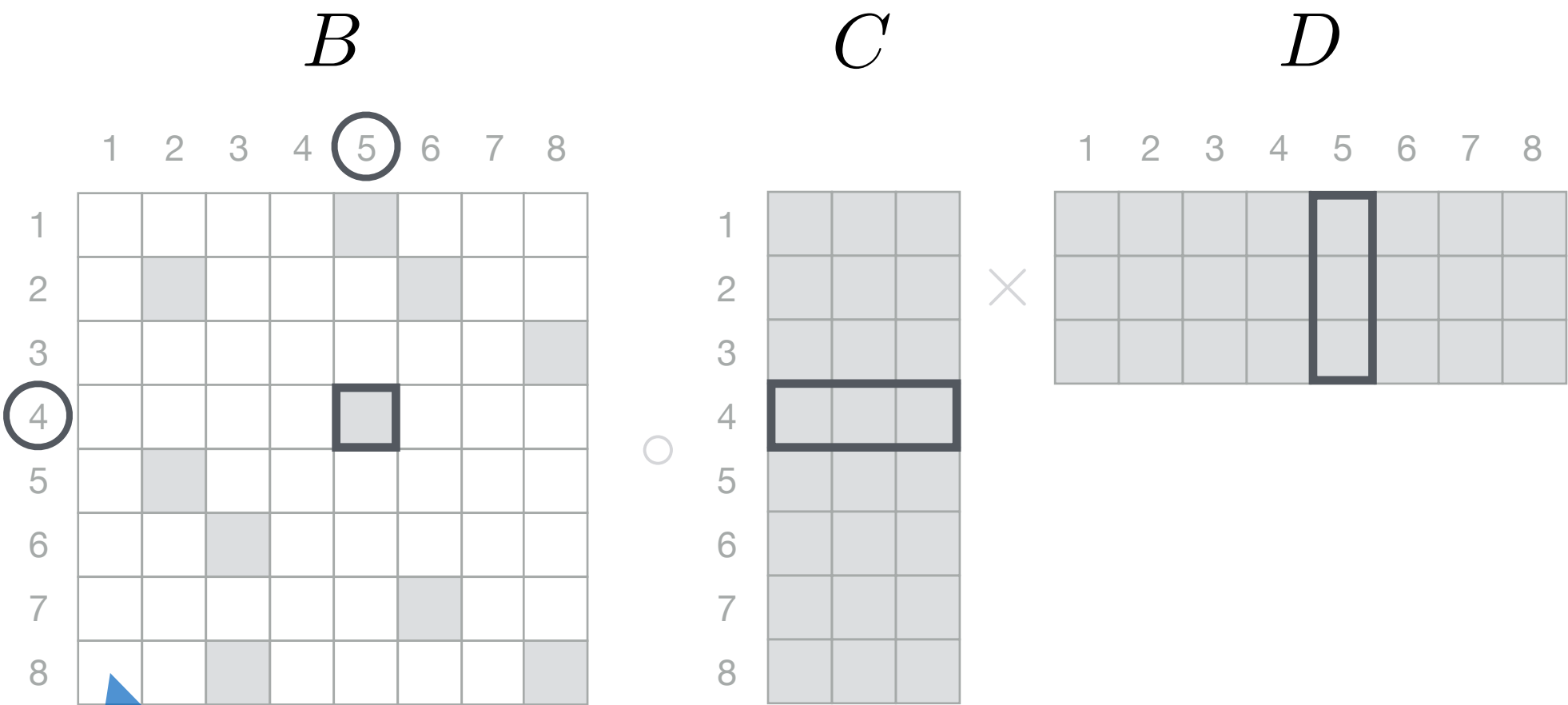
$$A_{ik} = \sum_j B_{ijk} * c_j \quad A = 0 \quad A = B + C + D \quad A = (B + C)(d + e)$$

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

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

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

Sampled Dense-Dense Matrix Multiplication (SDDMM)



this dot product need not be computed

64 inner product

SDDMM must be computed in a single kernel

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

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

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

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

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

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

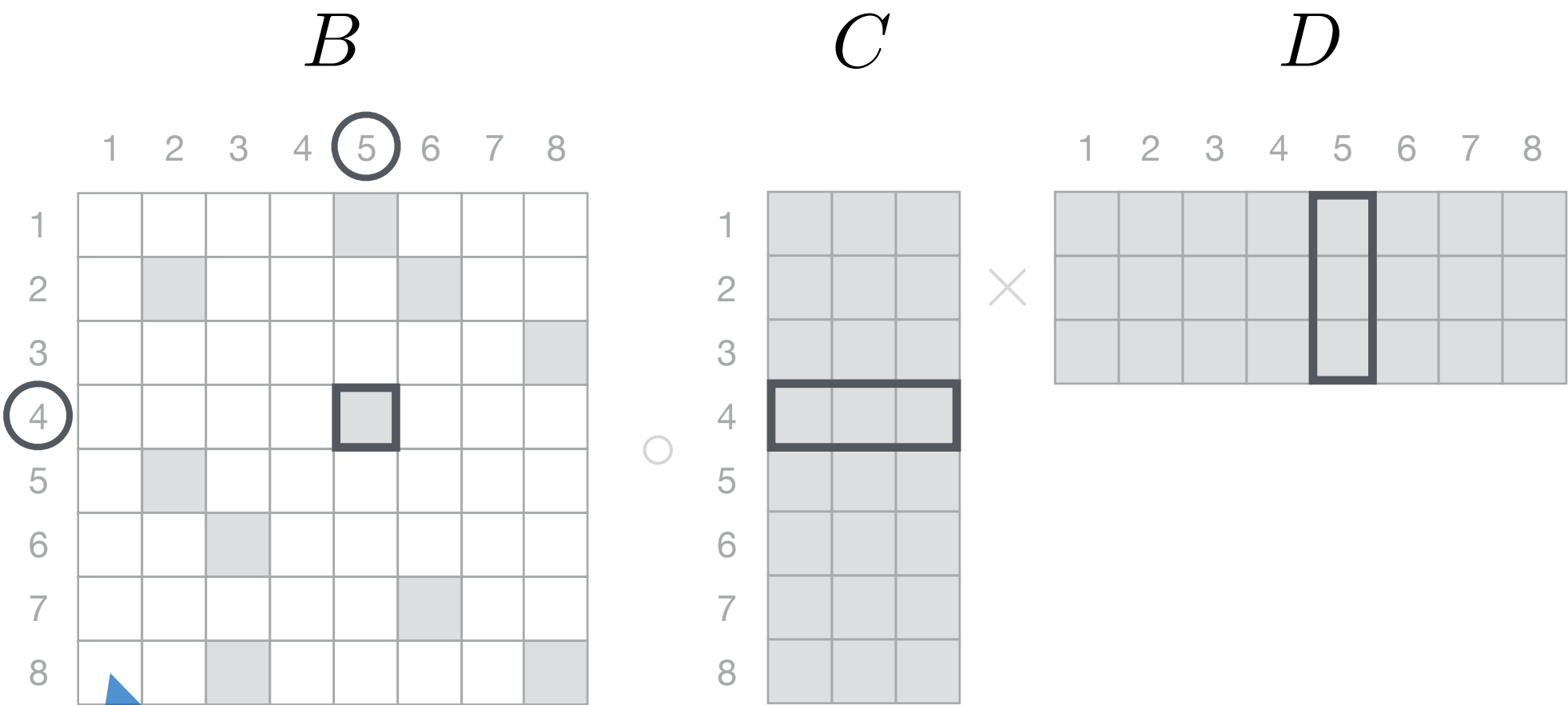
$$A_{ik} = \sum_j B_{ijk} * c_j \quad A = 0 \quad A = B + C + D \quad A = (B + C)(d + e)$$

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

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

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

Sampled Dense-Dense Matrix Multiplication (SDDMM)

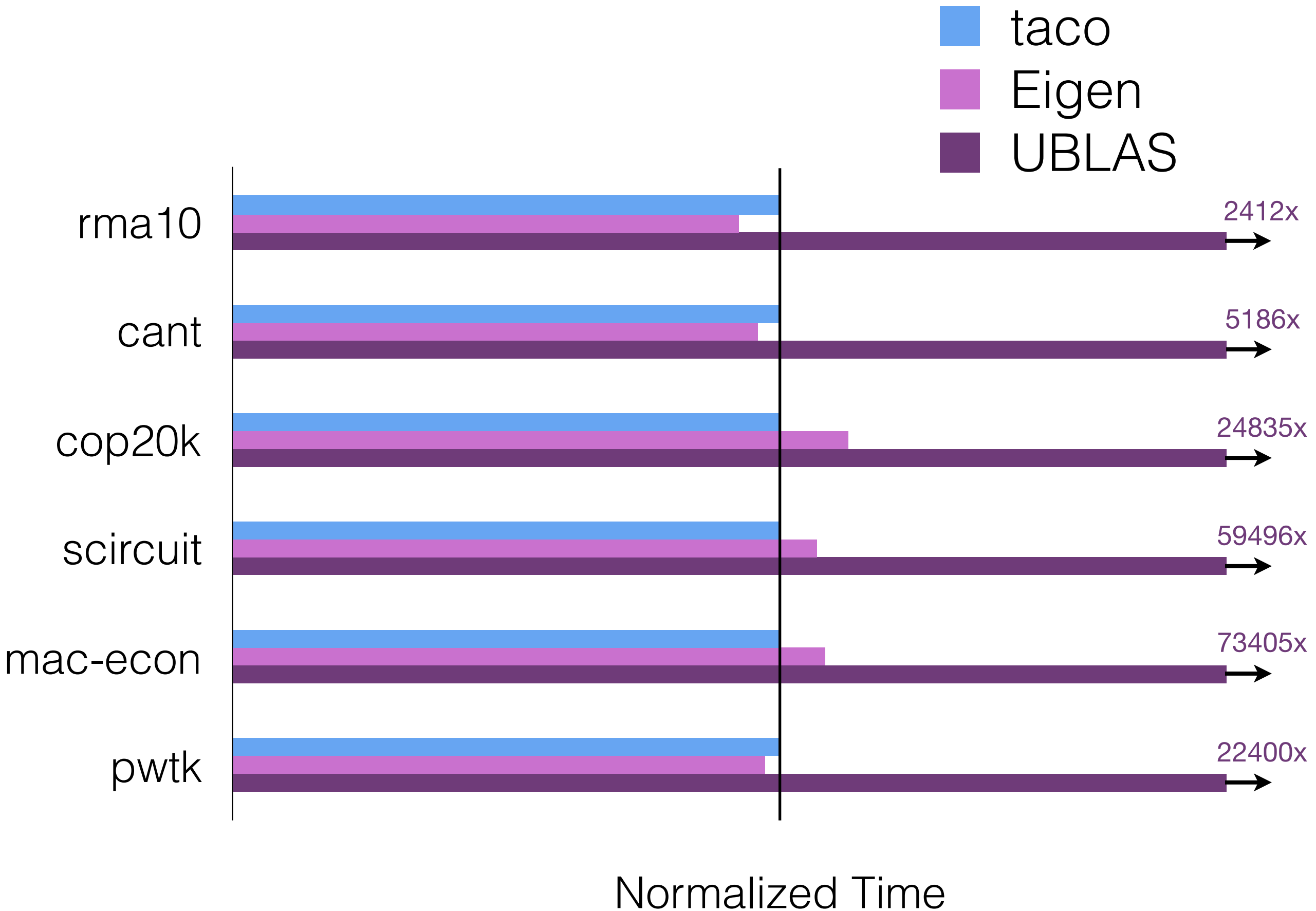


this dot product need not be computed

64 inner product
10 inner product

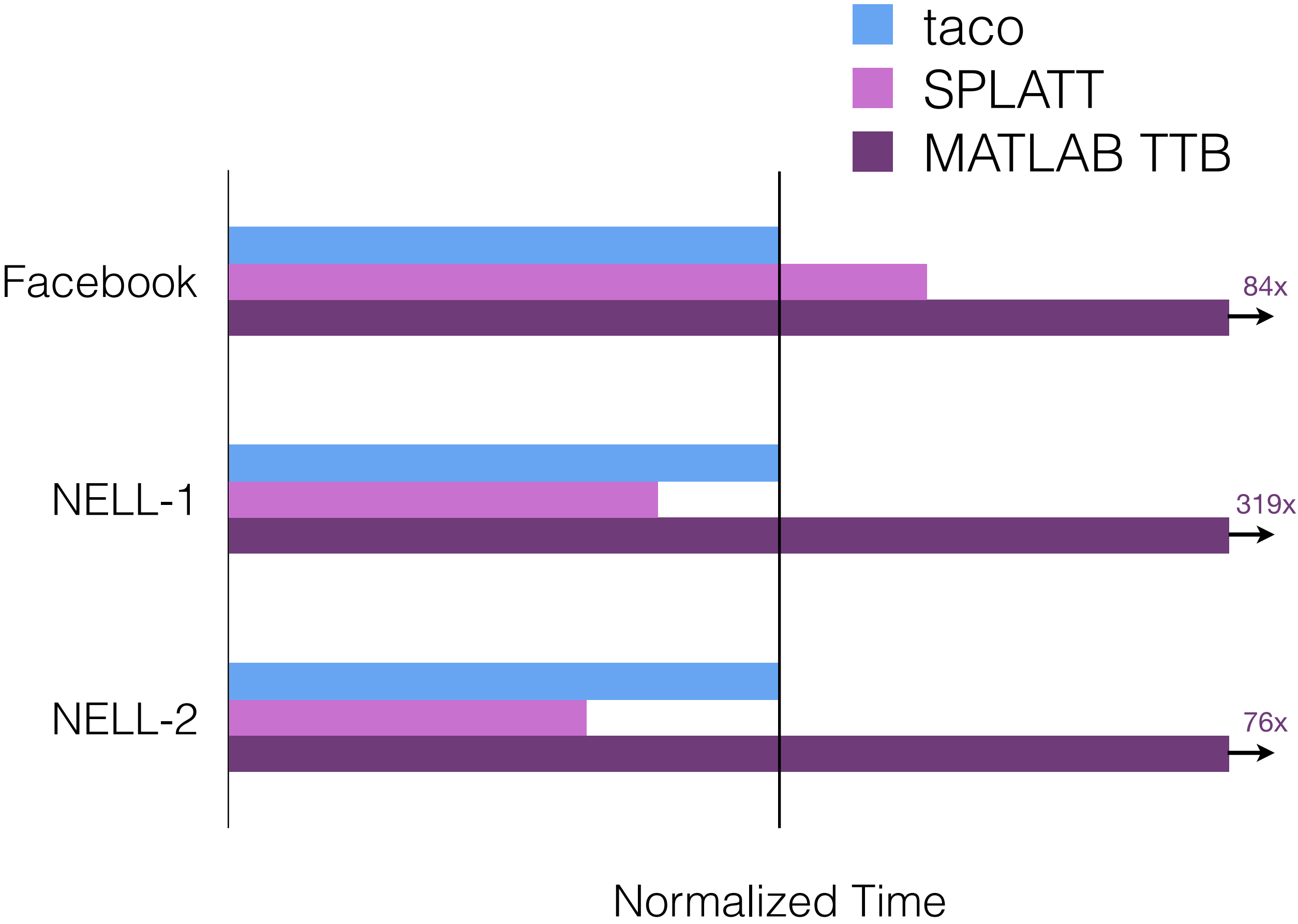
SDDMM must be computed in a single kernel

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



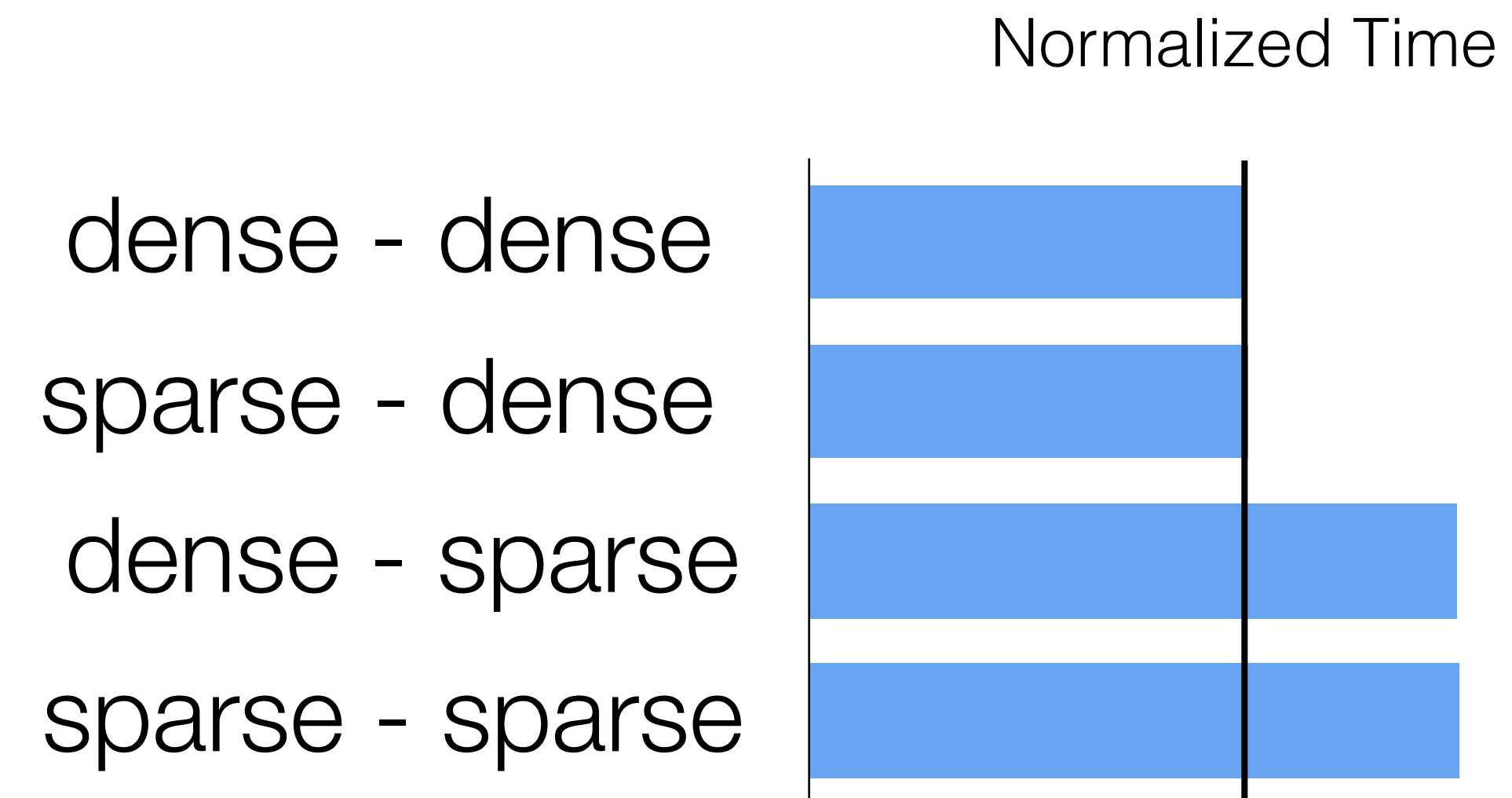
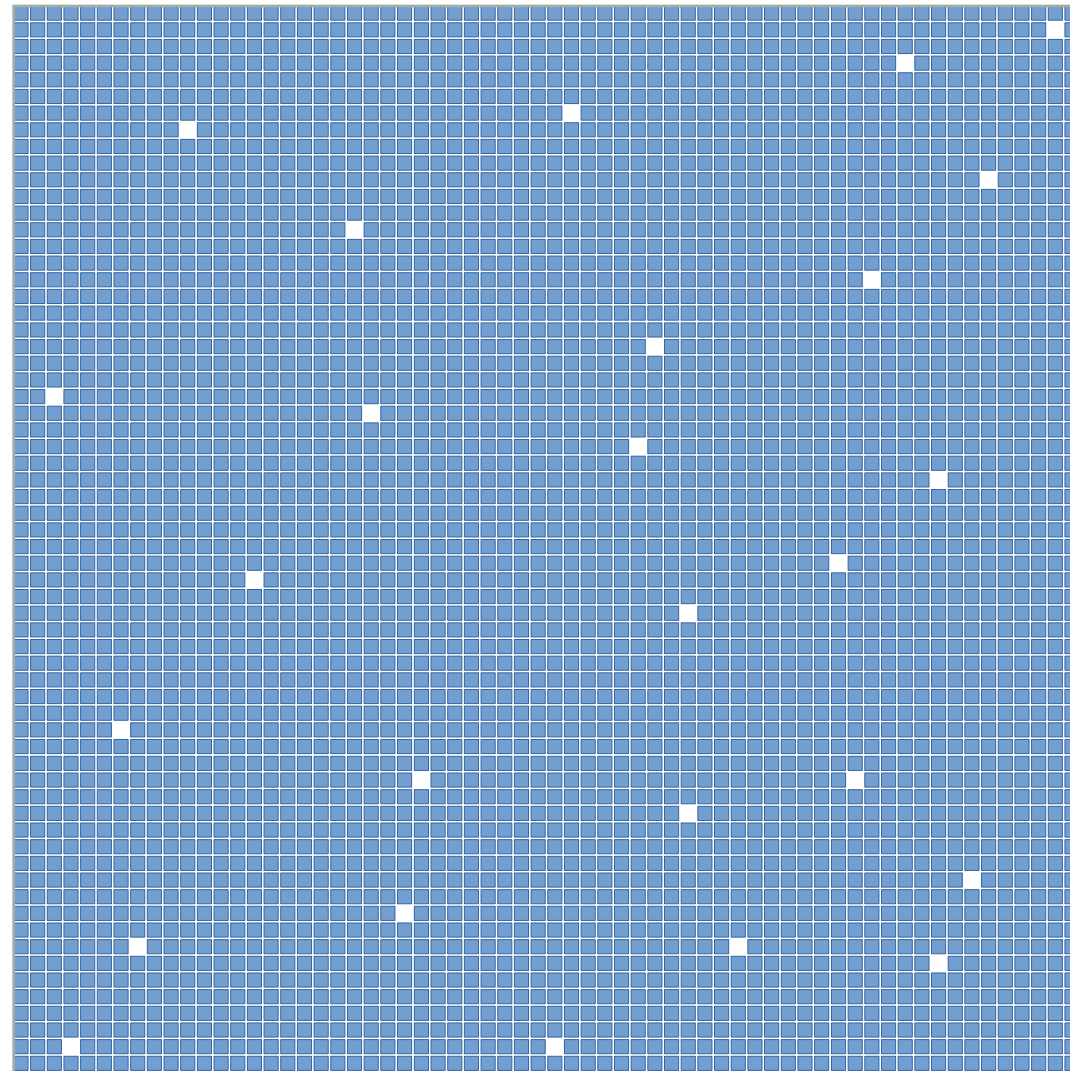
MTTKRP must be computed in a single kernel

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



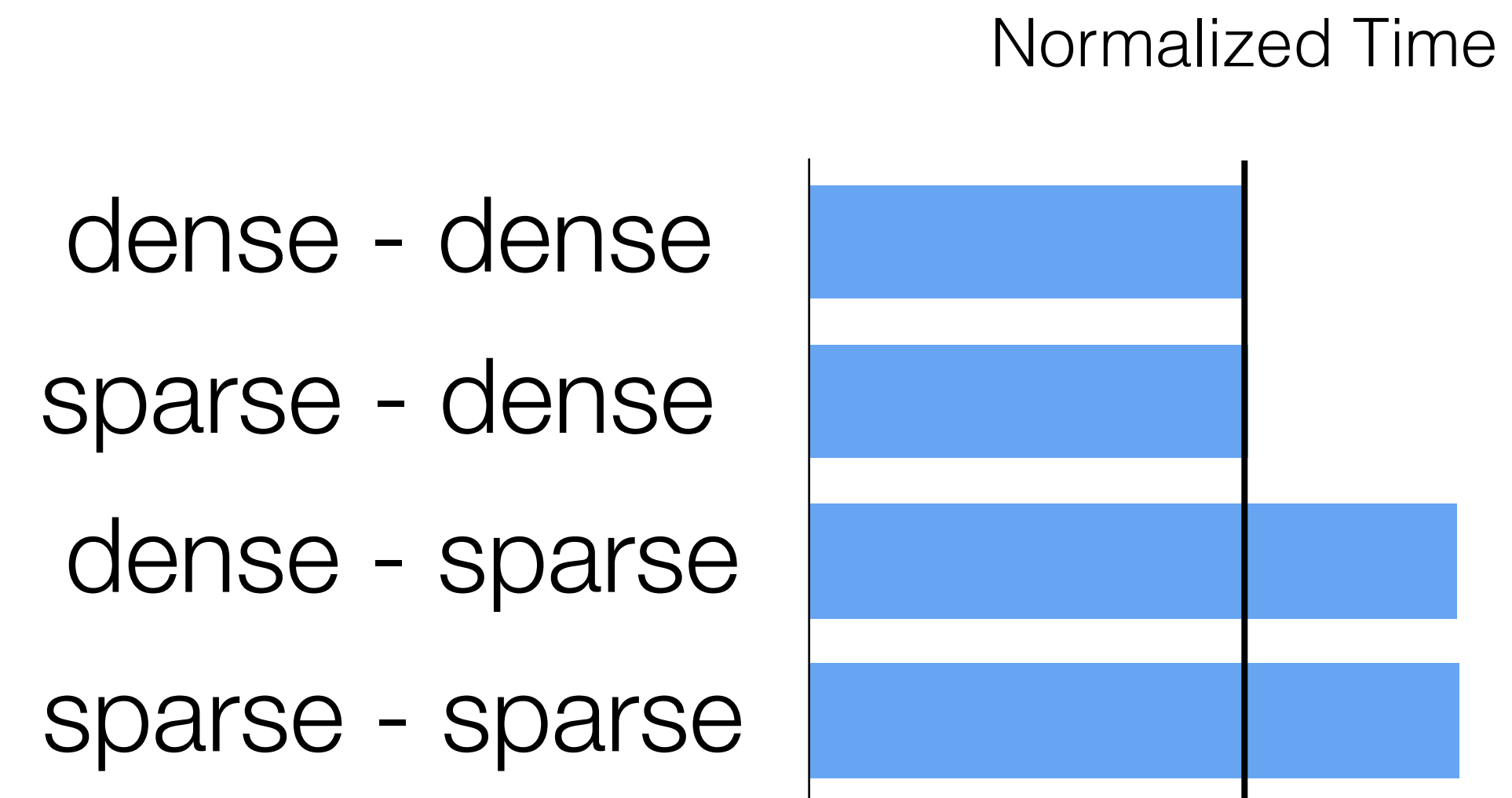
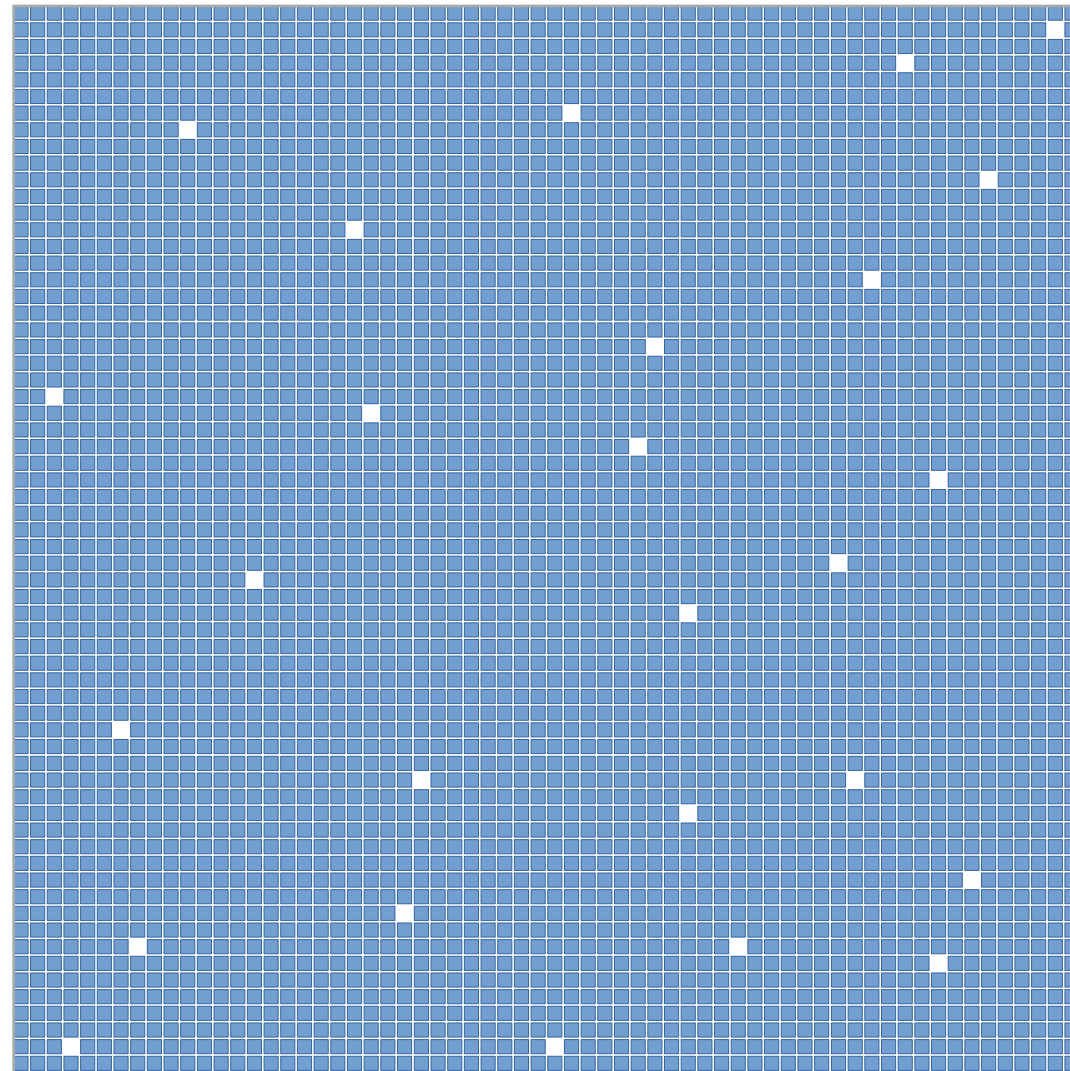
One size does not fit all - different matrices need different formats

Dense Matrix



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

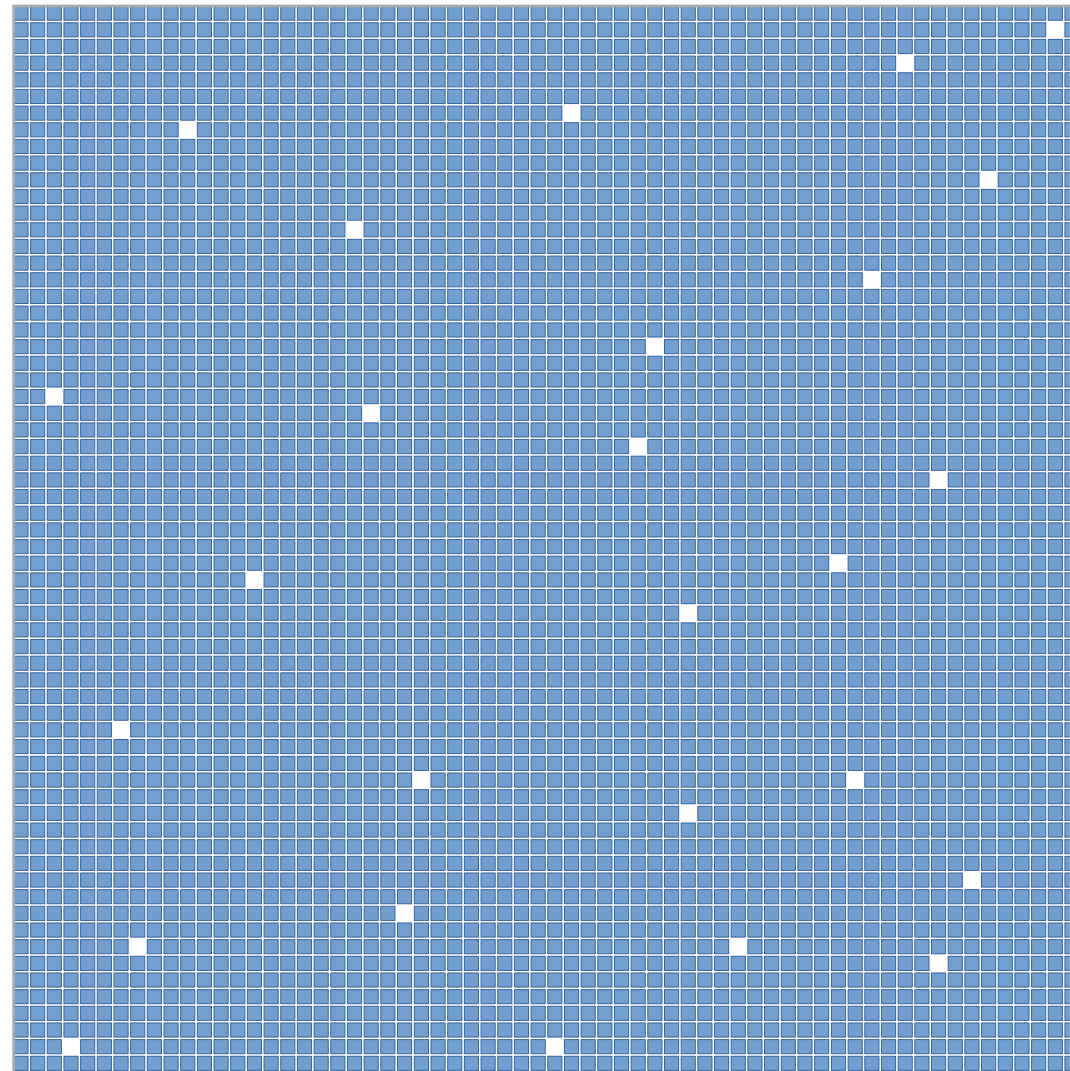
Dense Matrix



Formats (row-major)

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

Dense Matrix



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

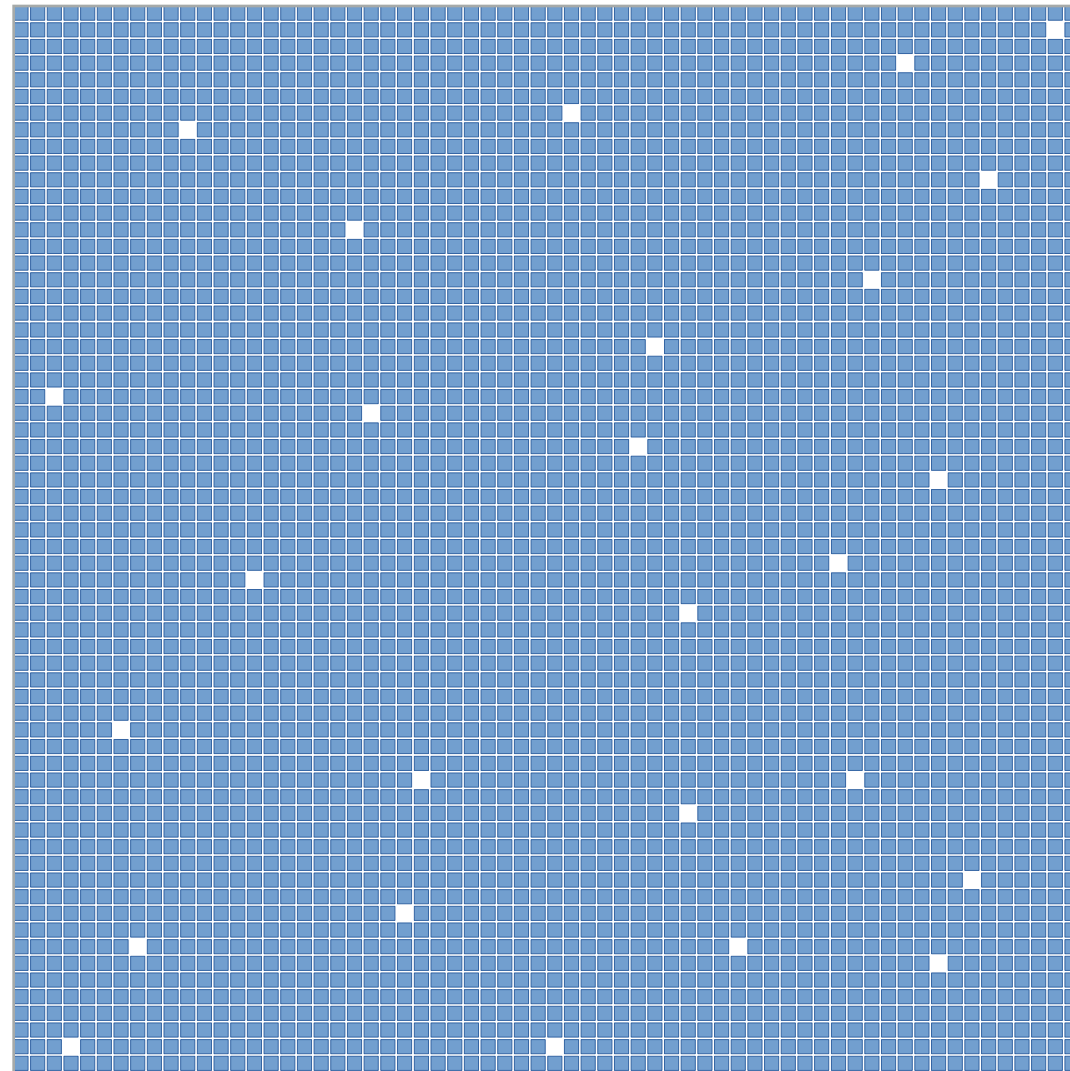
Normalized Time



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


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

Dense Matrix



Best format

dense - dense

sparse - dense

dense - sparse

sparse - sparse

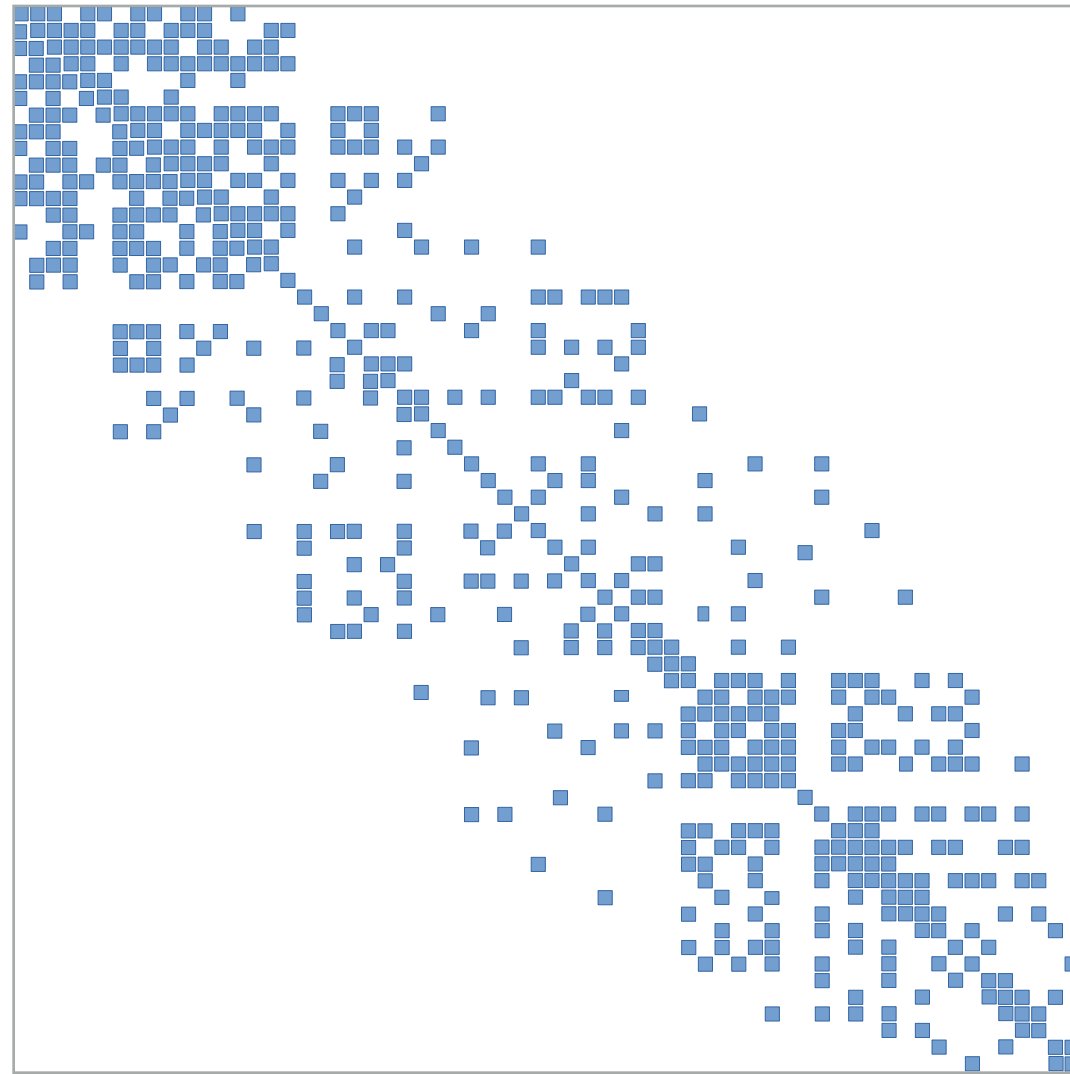
Normalized Time



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

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

Thermal Matrix



Best format

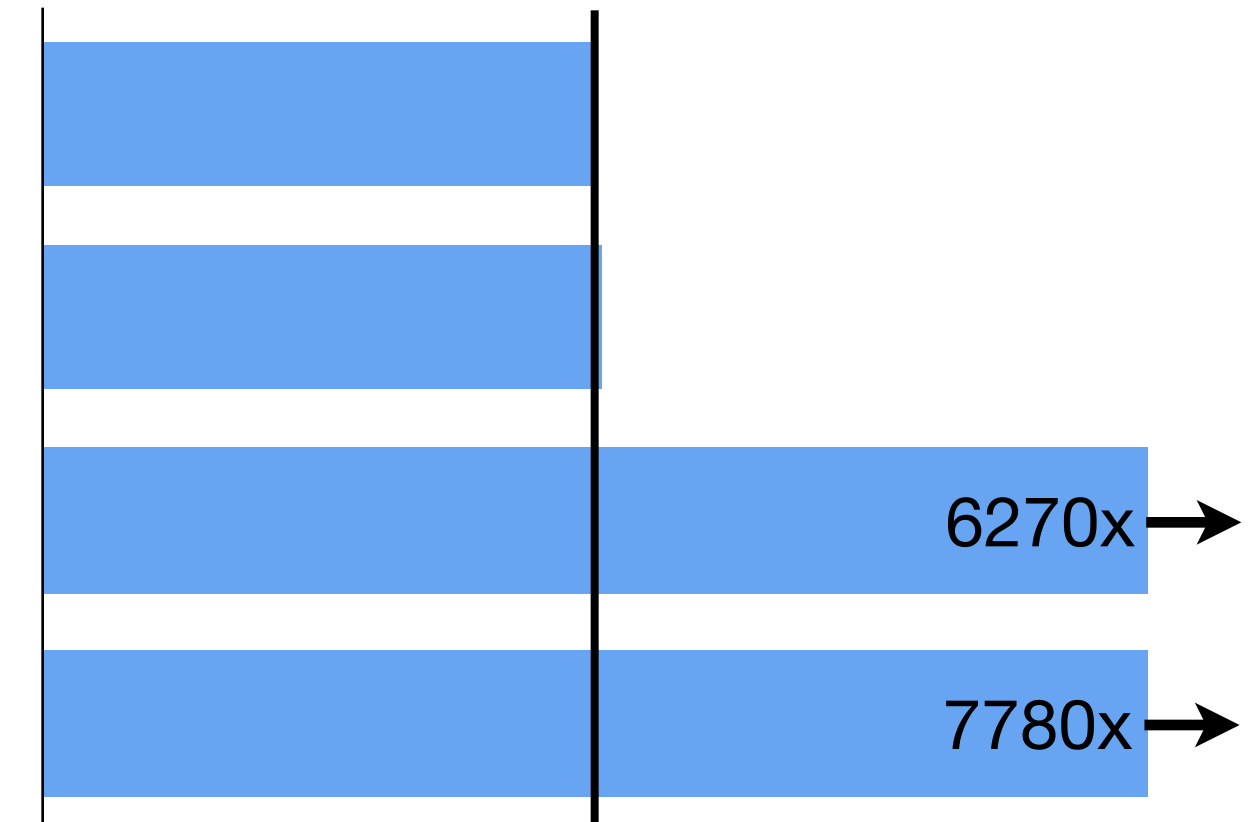
dense - sparse

sparse - sparse

dense - dense

sparse - dense

Normalized Time



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

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

Row-sliced Matrix



Best format

sparse - dense

sparse - sparse

dense - sparse

dense - dense

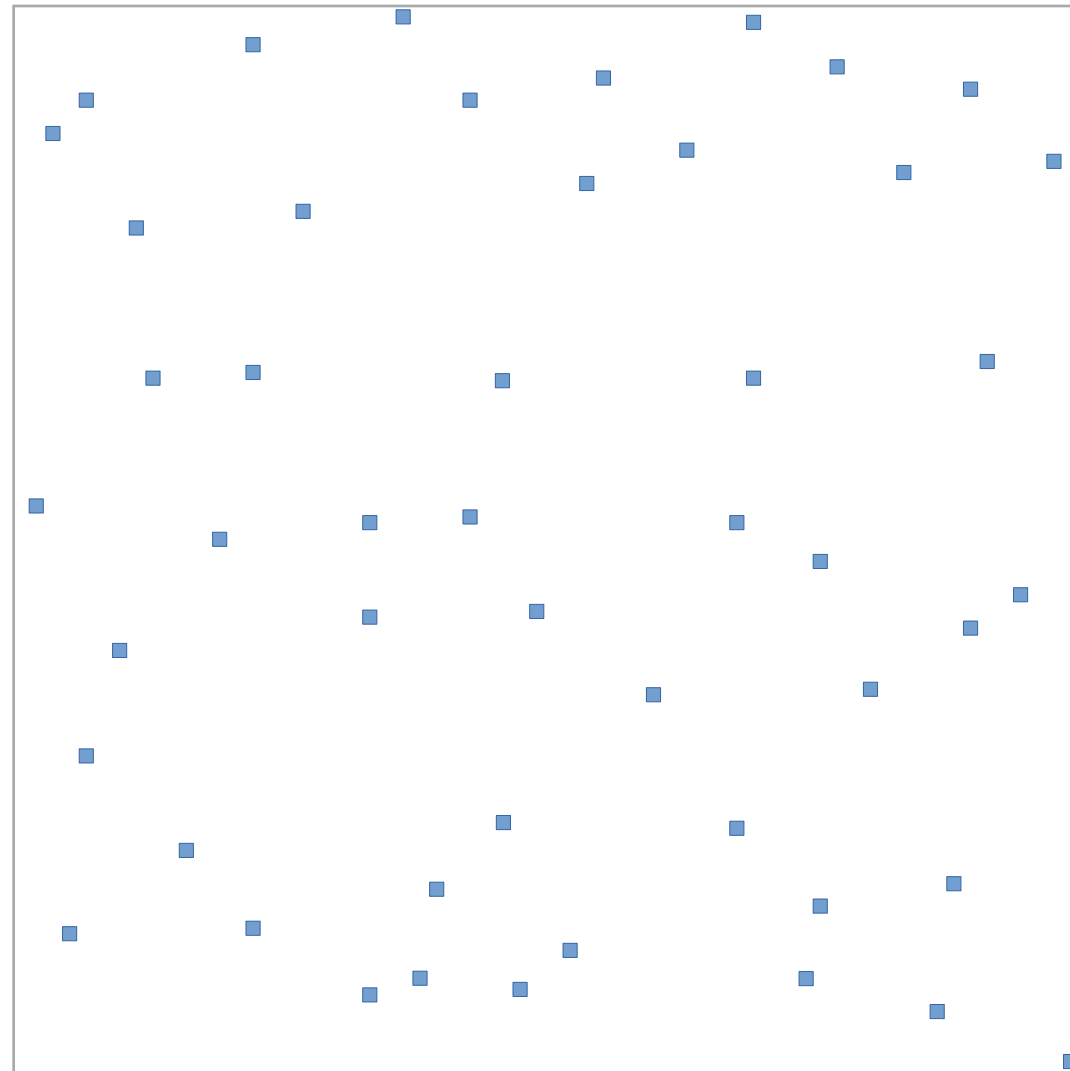
$$a_i = \sum_j B_{ij} c_j$$

Normalized Time



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

Hypersparse Matrix



Best format

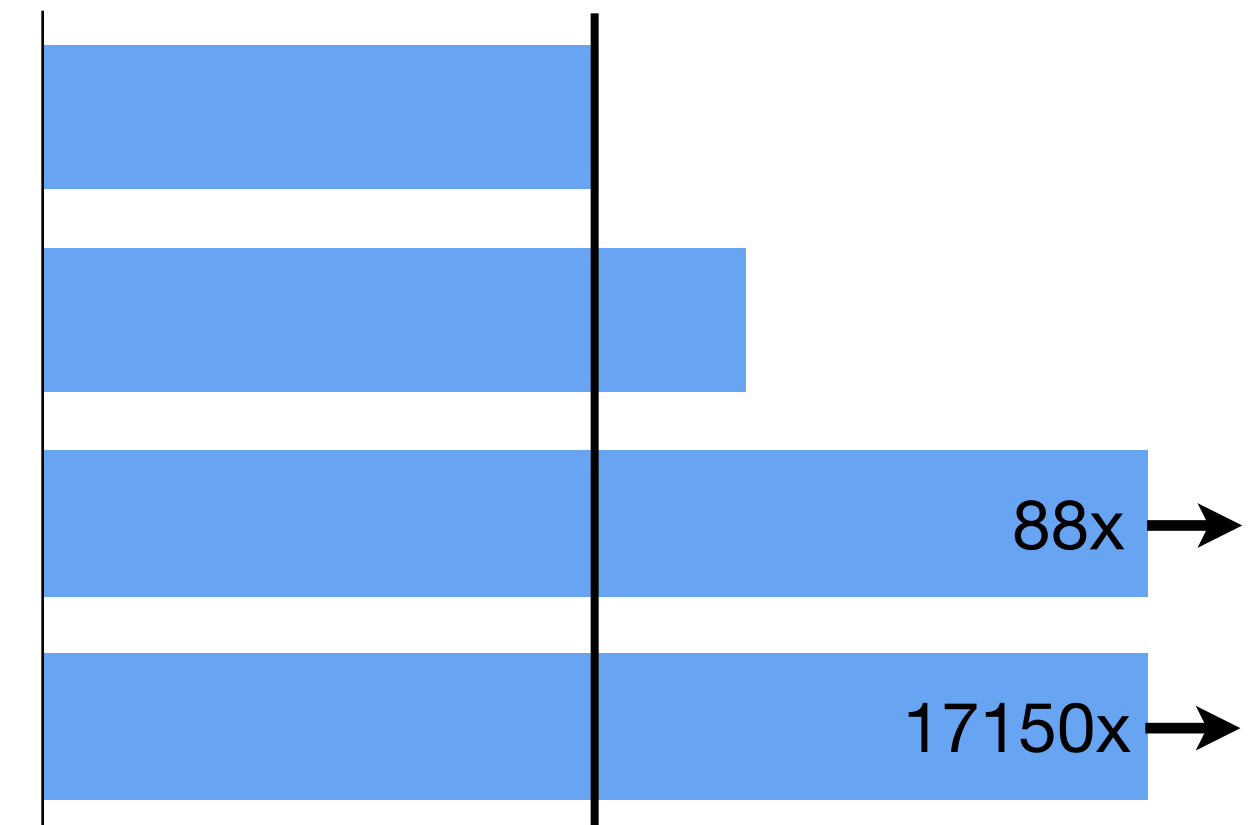
sparse - sparse

dense - sparse

sparse - dense

dense - dense

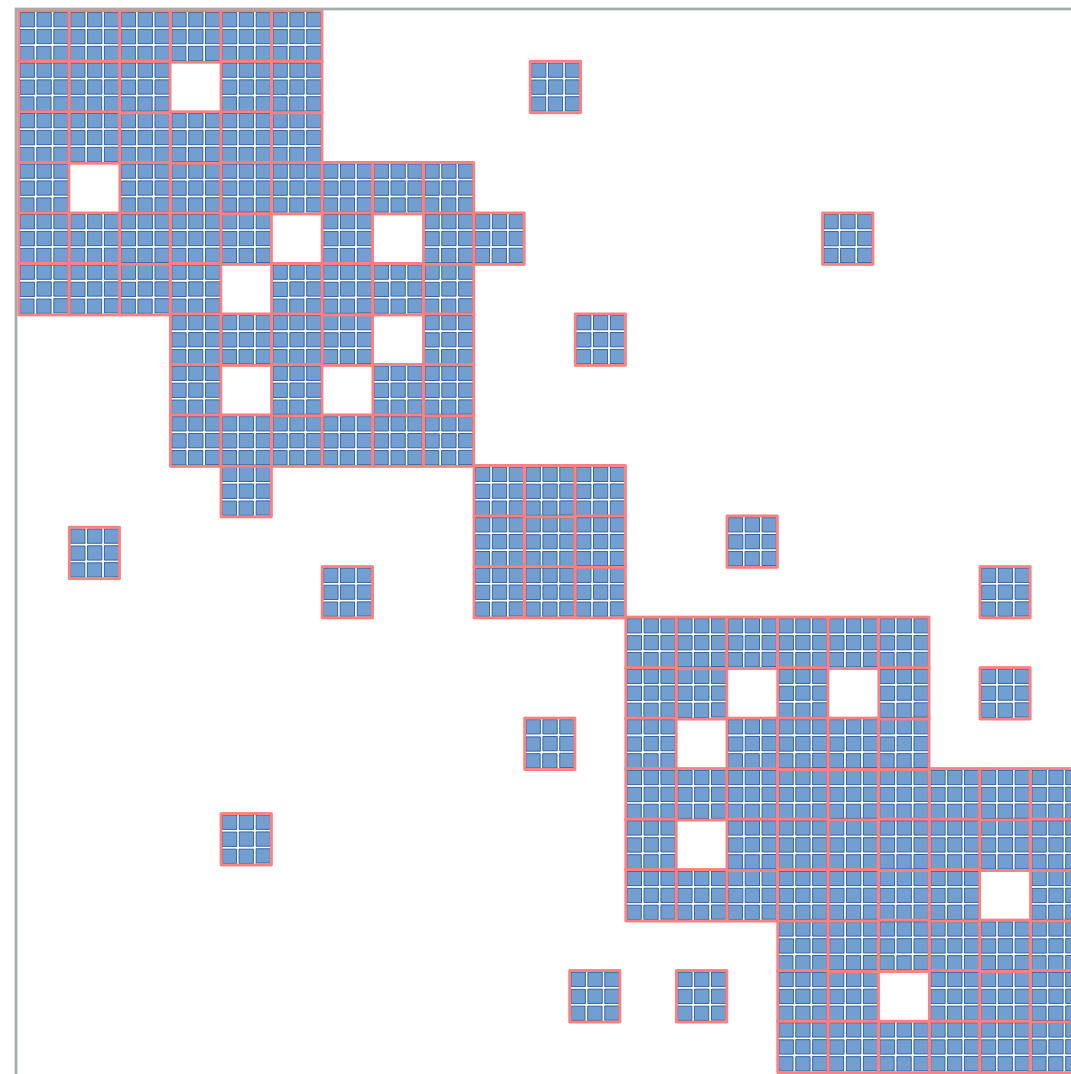
Normalized Time



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

Blocked matrices are 4-tensors

Blocked FEM Matrix



3x3 blocks

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

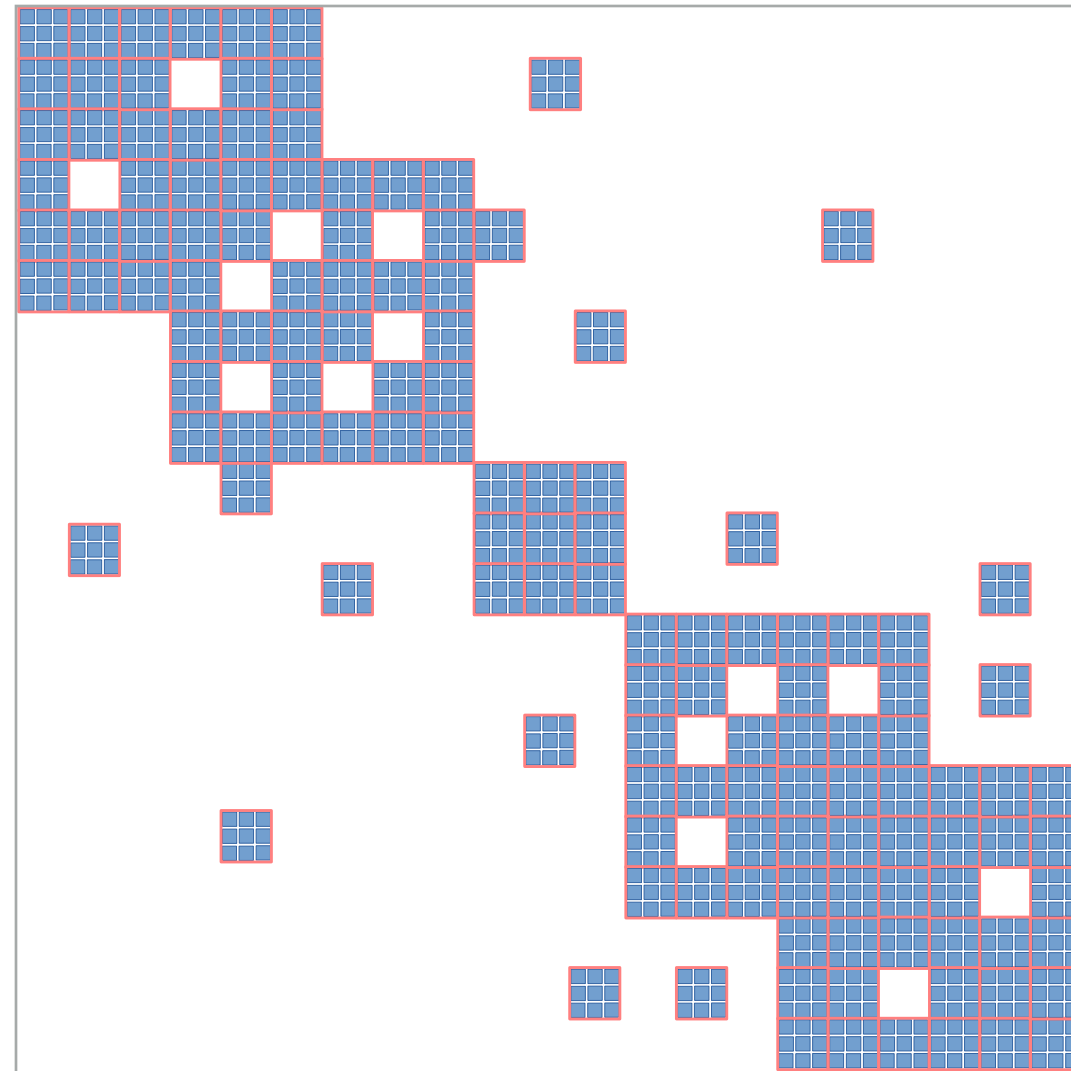
$$a_i = \sum_j B_{ij} c_j$$

Normalized Time



Blocked matrices are 4-tensors

Blocked FEM Matrix



4-tensor



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

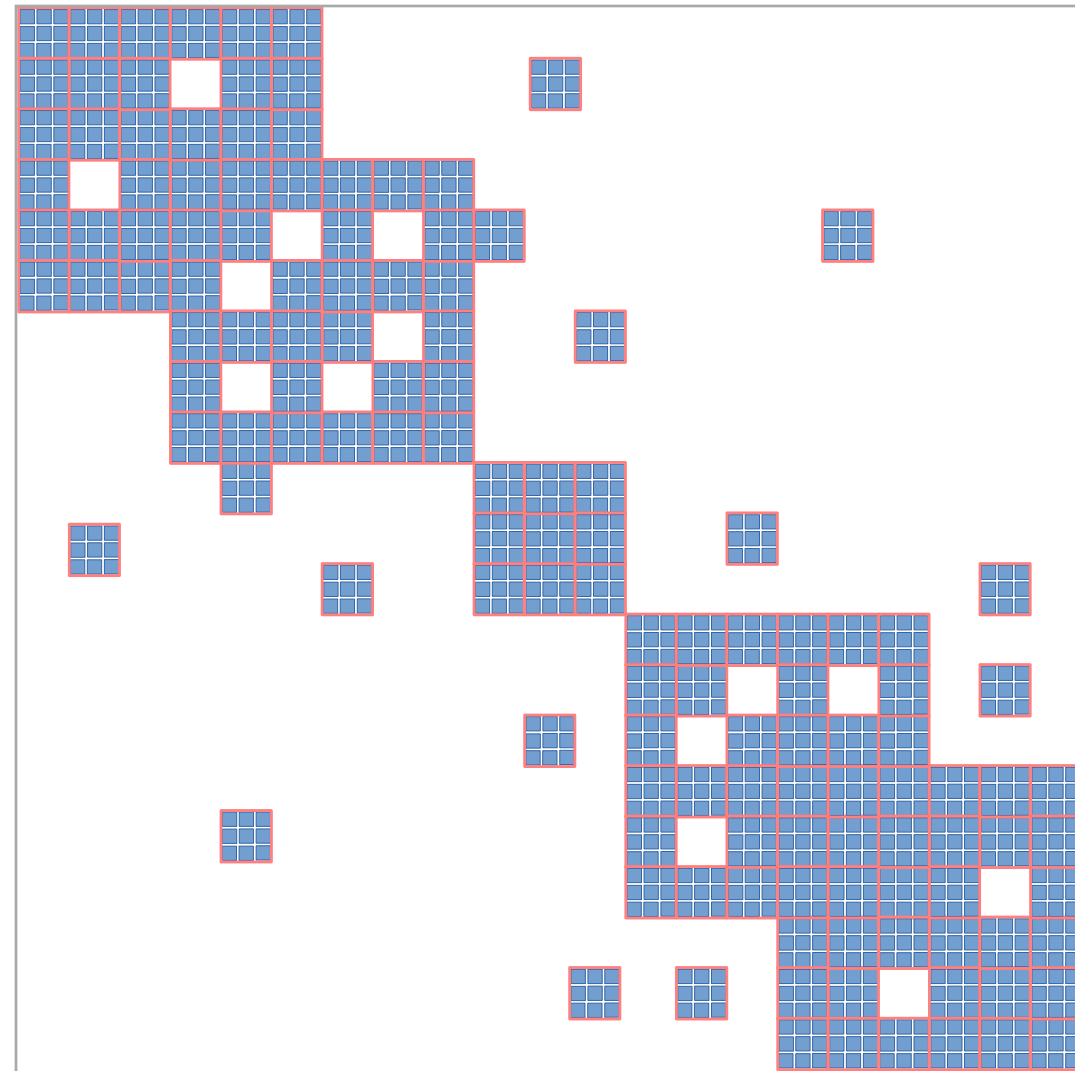
Normalized Time



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

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

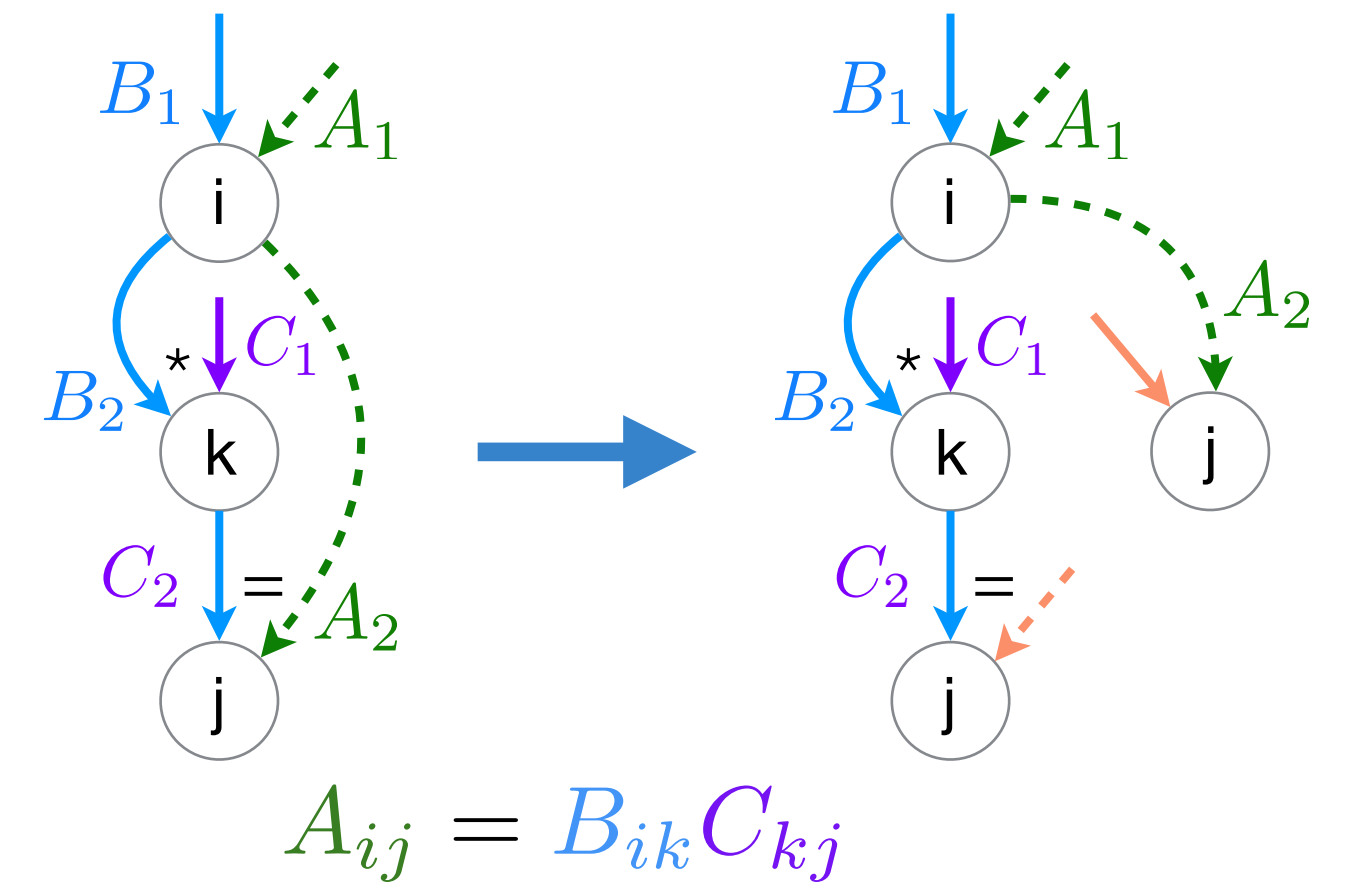
COO

DIA

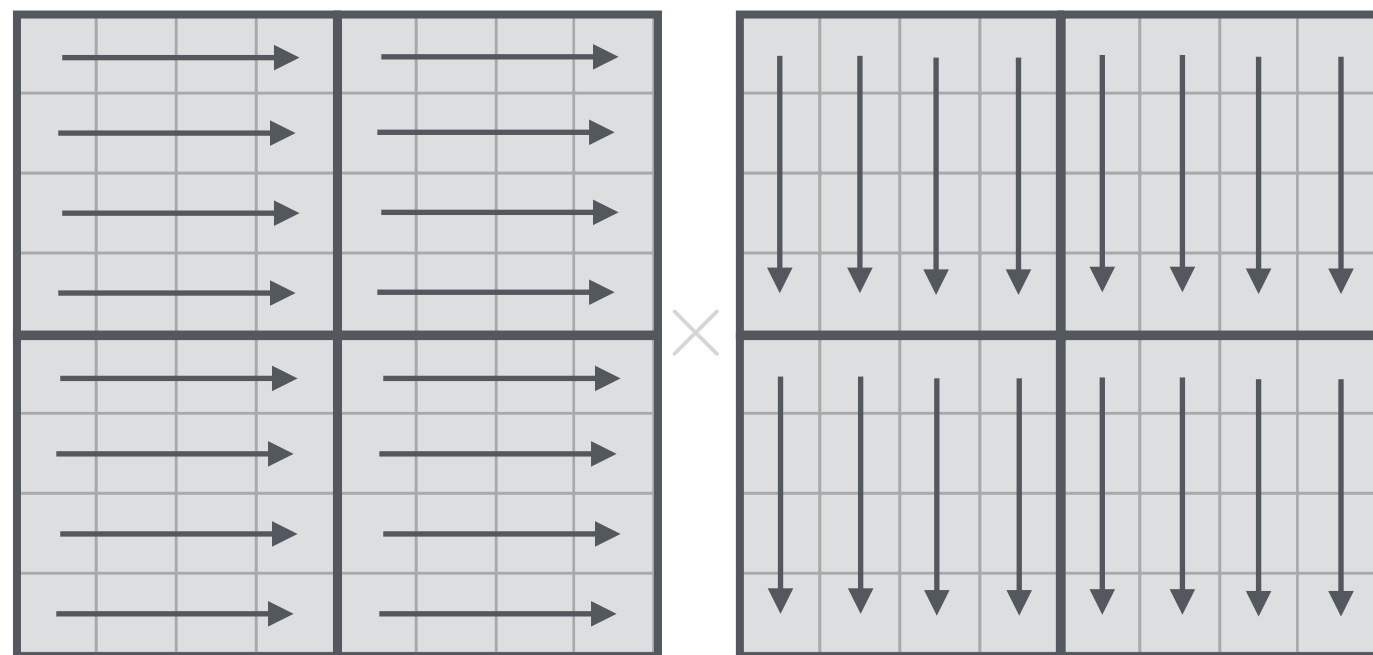
ELL

Hash maps

Workspaces



Scheduling Language



Semirings

$$(+, *)$$

float

 $(\max, +)$

double

(min, max)

complex

$$(\wedge, \vee)$$

int

$$(f(), g())$$

bool

Policy

Inspector-Executor

ML

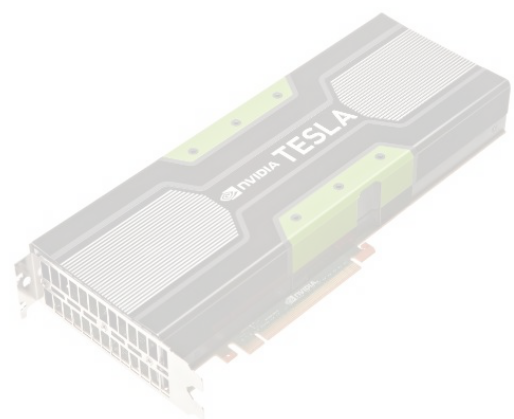
Sampling

Heuristics

Autotuning

Future Work (some of it)

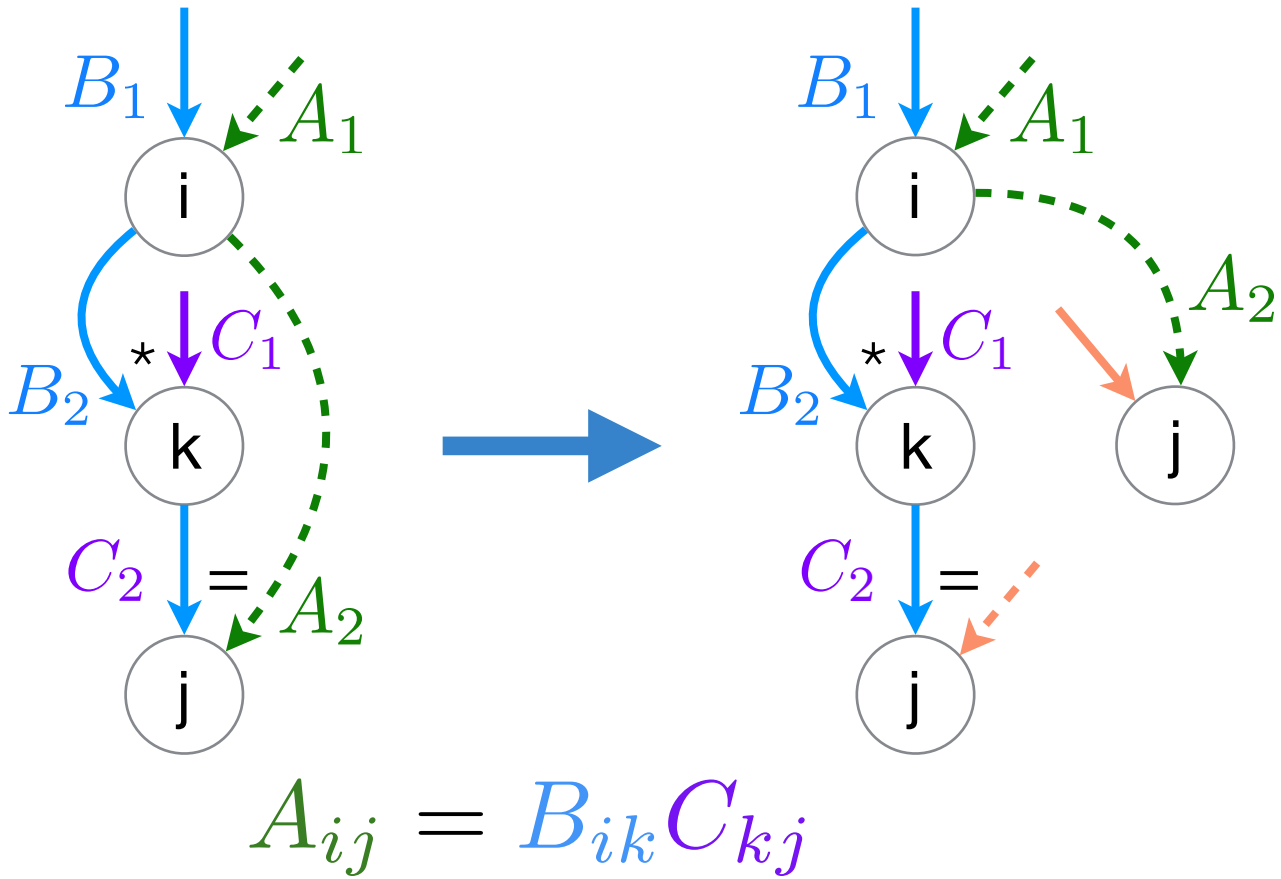
Portability/Acceleration



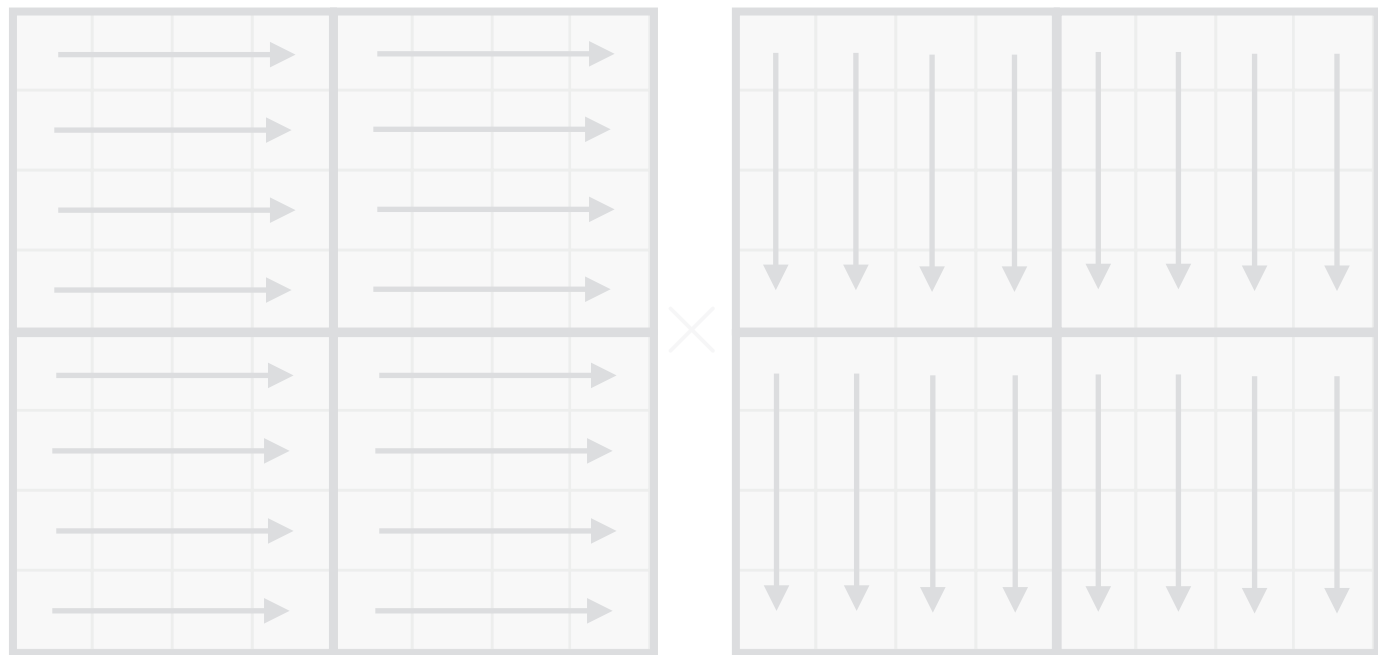
Formats

COO
ELL
DIA
Hash maps

Workspaces



Scheduling Language



Semirings

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

Policy

Inspector-Executor
ML
Sampling
Heuristics
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, assemble, and compute
               // the expression
```

tensor-compiler.org

github.com/tensor-compiler/taco



(Vanilla) MIT License

Online Code Generator

The screenshot shows the web interface of the tensor algebra compiler. At the top is a navigation bar with links to Docs, Publications, Demo, and GitHub. Below this is a disclaimer about the alpha implementation. The main input area contains a text box with the expression `y(i) = A(i,j) * x(j)` and a "GENERATE KERNEL" button. Below the input is a table for defining tensors and their formats. The table has two columns: "Tensor" and "Format (reorder dimensions by dragging the drop-down menus)". It contains three rows: "y" with format "Dense", "A" with formats "Dense" and "Sparse", and "x" with format "Dense". Below the table are three tabs: "COMPUTE LOOPS", "ASSEMBLY LOOPS", and "COMPLETE CODE". The "COMPUTE LOOPS" tab is active, showing a placeholder for generated code. At the bottom, there is a disclaimer about the generated code and a footer with a license notice.

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 the tensor-compiler.org online code generator interface. The browser address bar shows "tensor-compiler.org". The page has a blue header with "taco: the tensor algebra compiler" and links for "Docs", "Publications", "Demo", and "GitHub". Below the header, a disclaimer 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." The main input area prompts the user to "Input a tensor algebra expression in index notation to generate code that computes it:" and shows the example expression "y(i) = A(i,j) * x(j)". To the right of the input is a "GENERATE KERNEL" button. Below the input is a table for defining tensors and their formats. The table has two columns: "Tensor" and "Format (reorder dimensions by dragging the drop-down menus)". It contains three rows: "y" with format "Dense", "A" with formats "Dense" and "Sparse", and "x" with format "Dense". Below the table are three tabs: "COMPUTE LOOPS", "ASSEMBLY LOOPS", and "COMPLETE CODE". The "COMPUTE LOOPS" tab is selected, showing a placeholder for the generated code: "/* The generated compute loops will appear here */". At the bottom, a disclaimer 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." The footer mentions "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 web interface of the tensor algebra compiler. At the top is a navigation bar with links to Docs, Publications, Demo, and GitHub. Below this is a disclaimer about the alpha implementation. The main input area contains a text field with the expression `y(i) = A(i,j) * x(j)` and a "GENERATE KERNEL" button. Below the input is a table for defining tensors and their formats. The table has two columns: "Tensor" and "Format (reorder dimensions by dragging the drop-down menus)". It contains three rows: "y" with format "Dense", "A" with formats "Dense" and "Sparse", and "x" with format "Dense". Below the table are three tabs: "COMPUTE LOOPS", "ASSEMBLY LOOPS", and "COMPLETE CODE". The "COMPUTE LOOPS" tab is active, showing a placeholder for the generated code. At the bottom, there is a disclaimer about the generated code and a footer with a license notice.

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 web interface of the tensor algebra compiler. At the top, there's a navigation bar with links to Docs, Publications, Demo, and GitHub. Below this, a disclaimer states it's an alpha implementation. The main input area has a text field containing the expression `y(i) = A(i,j) * x(j)` and a "GENERATE KERNEL" button. Below the input, there's a table to define tensors and their formats. The table has two columns: "Tensor" and "Format (reorder dimensions by dragging the drop-down menus)". It contains three rows: "y" with format "Dense", "A" with formats "Dense" and "Sparse", and "x" with format "Dense". Below the table, there are tabs for "COMPUTE LOOPS", "ASSEMBLY LOOPS", and "COMPLETE CODE". The "COMPUTE LOOPS" tab is active, showing a placeholder for generated code. At the bottom, there's a disclaimer about the generated code and a footer with a license notice.

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 web interface of the tensor algebra compiler. At the top is a navigation bar with links to Docs, Publications, Demo, and GitHub. Below this is a disclaimer about the alpha implementation. The main input area contains the expression `y(i) = A(i,j) * x(j)` and a "GENERATE KERNEL" button. Below the input is a table for defining tensors and their formats. The table has two columns: "Tensor" and "Format (reorder dimensions by dragging the drop-down menus)". It contains three rows: "y" with format "Dense", "A" with formats "Dense" and "Sparse", and "x" with format "Dense". Below the table are three tabs: "COMPUTE LOOPS", "ASSEMBLY LOOPS", and "COMPLETE CODE". The "COMPUTE LOOPS" tab is active, showing a placeholder for the generated code. At the bottom, there is a disclaimer about the generated code and a footer with a license notice.

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 web interface of the tensor algebra compiler. At the top, there's a navigation bar with "taco: the tensor algebra compiler" and links to "Docs", "Publications", "Demo", and "GitHub". Below this, a disclaimer states it's an alpha implementation. The main input area has a text box with the expression "y(i) = A(i,j) * x(j)" and a "GENERATE KERNEL" button. Below the input, there's a table to define tensors and their formats. The table has two columns: "Tensor" and "Format (reorder dimensions by dragging the drop-down menus)". It contains three rows: "y" with format "Dense", "A" with formats "Dense" and "Sparse", and "x" with format "Dense". Below the table, there are tabs for "COMPUTE LOOPS", "ASSEMBLY LOOPS", and "COMPLETE CODE". The "COMPUTE LOOPS" tab is active, showing a placeholder for generated code. At the bottom, there's a disclaimer about the generated code and a footer with a license notice.

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 web interface of the tensor algebra compiler. At the top, there's a navigation bar with "taco: the tensor algebra compiler" and links to "Docs", "Publications", "Demo", and "GitHub". Below this, a disclaimer states it's an alpha implementation. The main input area has a text box with the expression "y(i) = A(i,j) * x(j)" and a "GENERATE KERNEL" button. Below the input, there's a table to define tensors and their formats. The table has two columns: "Tensor" and "Format (reorder dimensions by dragging the drop-down menus)". It contains three rows: "y" with format "Dense", "A" with formats "Dense" and "Sparse", and "x" with format "Dense". Below the table, there are tabs for "COMPUTE LOOPS", "ASSEMBLY LOOPS", and "COMPLETE CODE". The "COMPUTE LOOPS" tab is active, showing a placeholder for generated code. At the bottom, there's a disclaimer about the generated code and a footer with a license notice.

tensor-compiler.org/codegen