Deinsum: Practically I/O Optimal Multilinear Algebra

Alexandros Nikolaos Ziogas, **Grzegorz Kwasniewski**, Tal Ben-Nun, Timo Schneider, Torsten Hoefler

SC22
Dallas, TX | hpc accelerates.

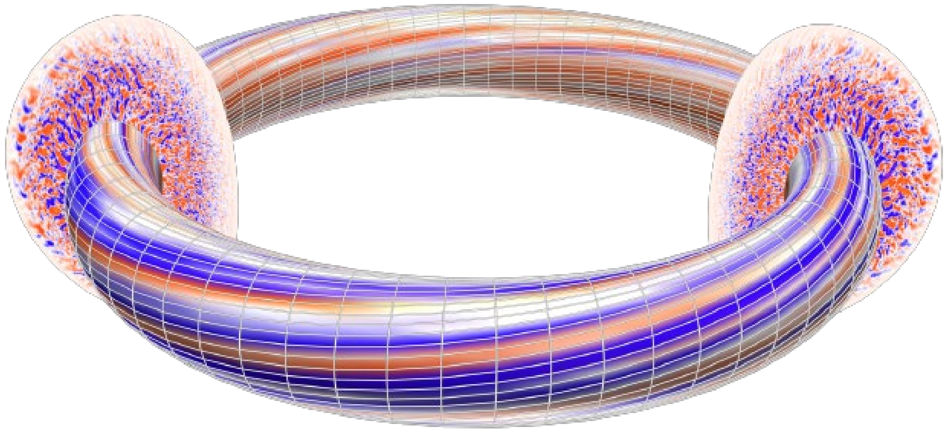
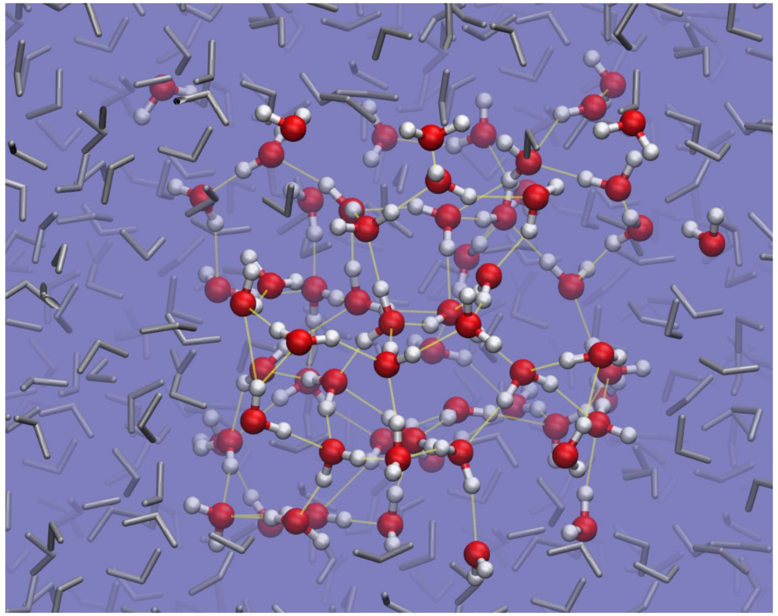
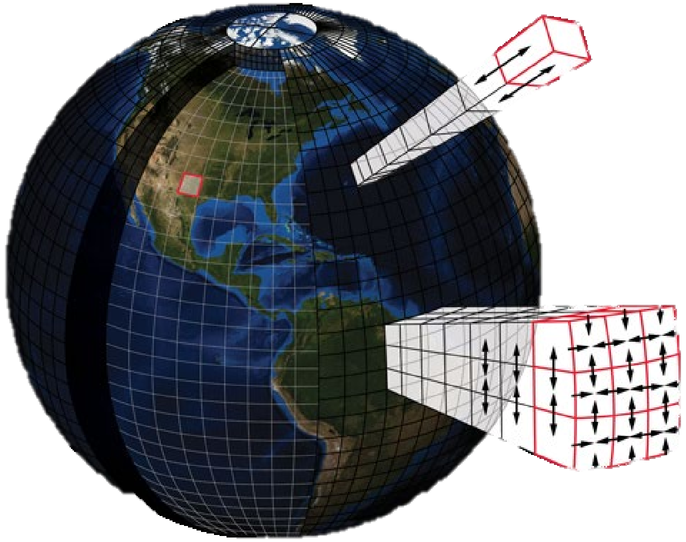
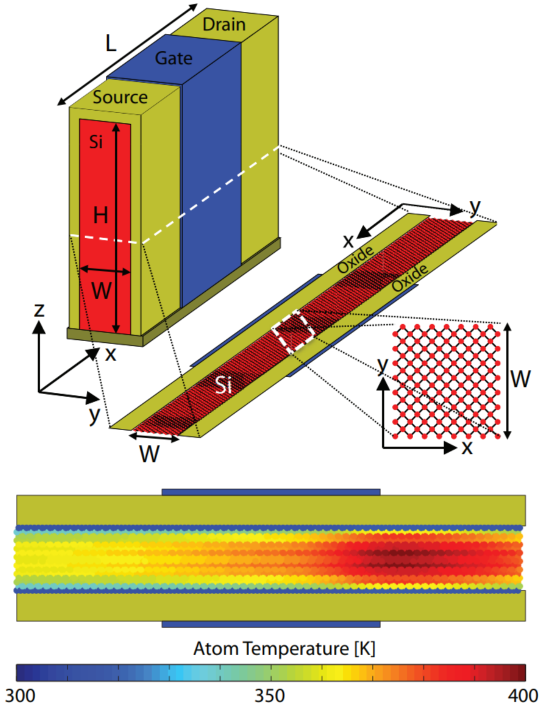
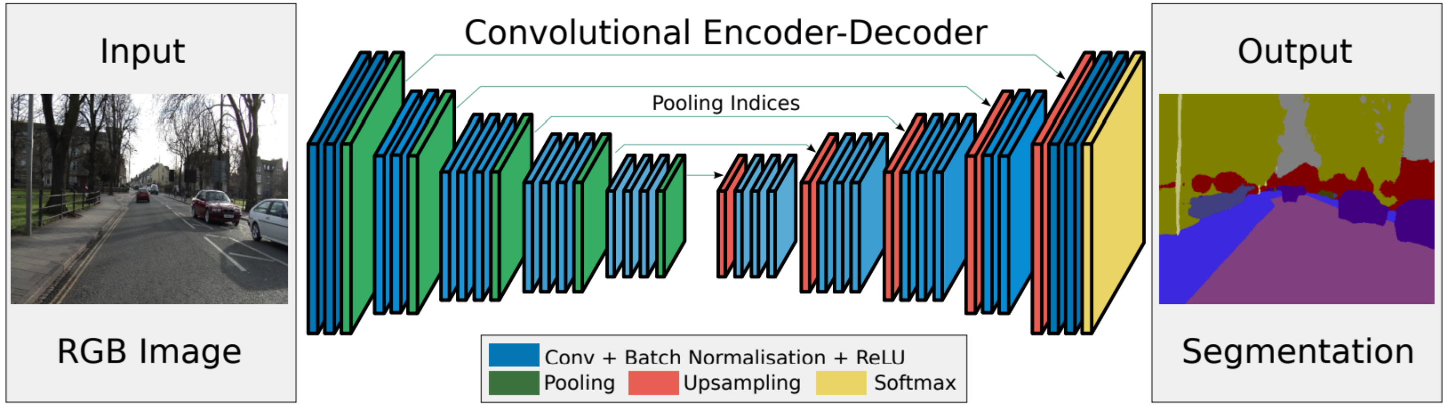


Image generated from a GTS simulation by Kwan-Liu Ma and his group at the University of California, Davis.

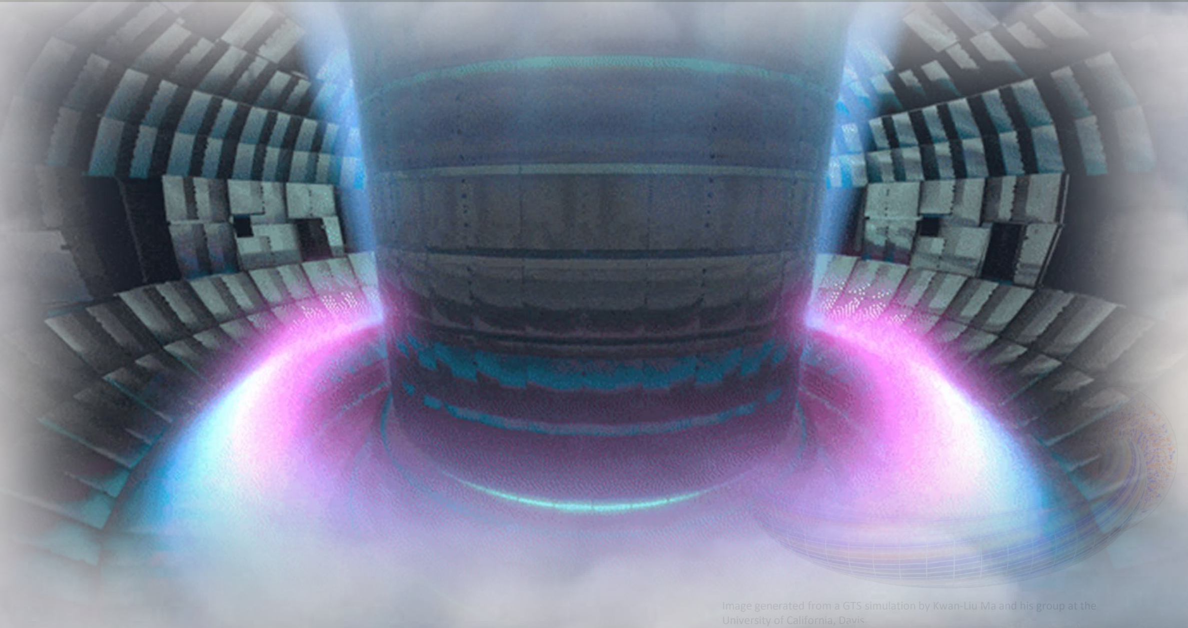

Enabling Simulation at the Fifth Rung of DFT: Large Scale RPA Calculations with Excellent Time to Solution, Mauro Del Ben et al.


Credit: K. Cantner, AGI.

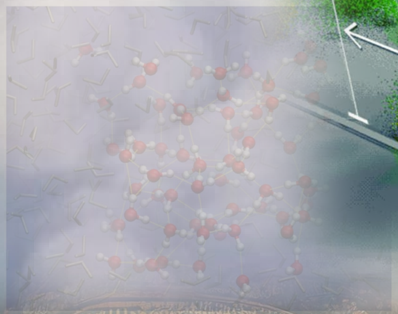


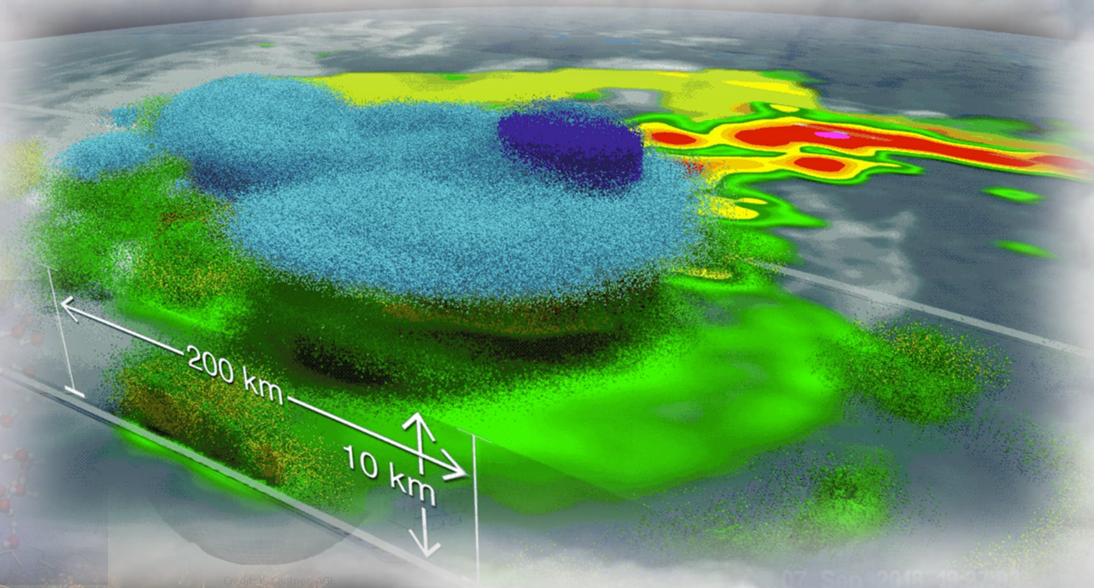


SegNet: A Deep Convolutional Encoder-Decoder Architecture for Image Segmentation, Badrinarayanan et al.



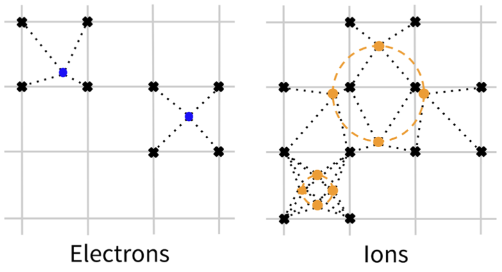
Credit: Princeton Plasma Physics Laboratory

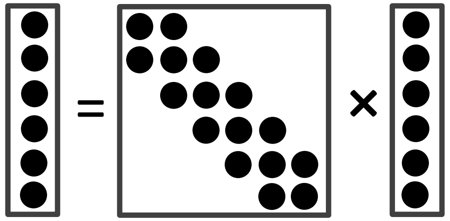




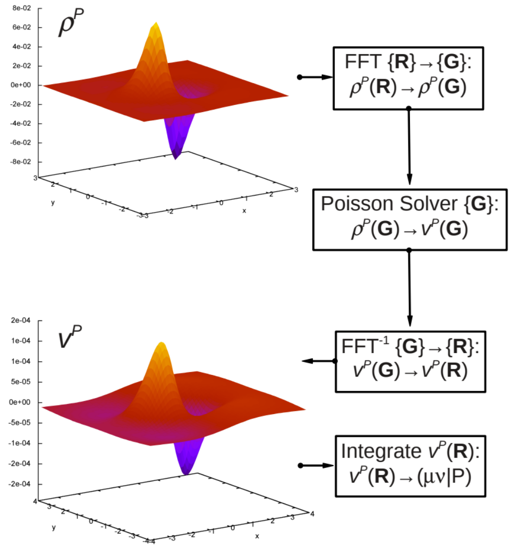
Credit: NASA's Goddard Space Flight Center

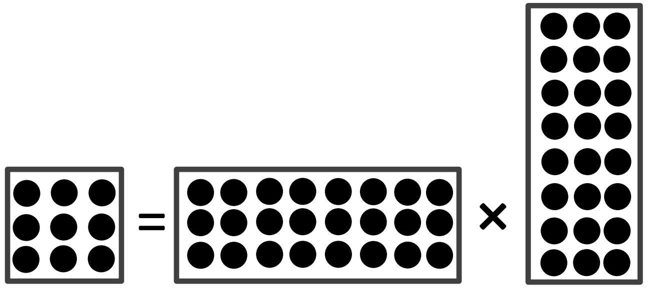
Credit: Jason Allen



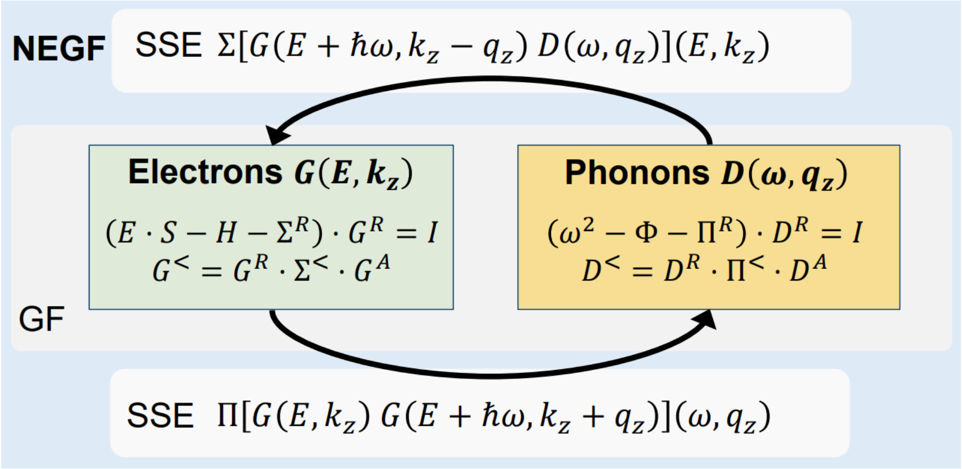


Extreme Scale Plasma Turbulence Simulations on Top Supercomputers Worldwide, Tang et al.

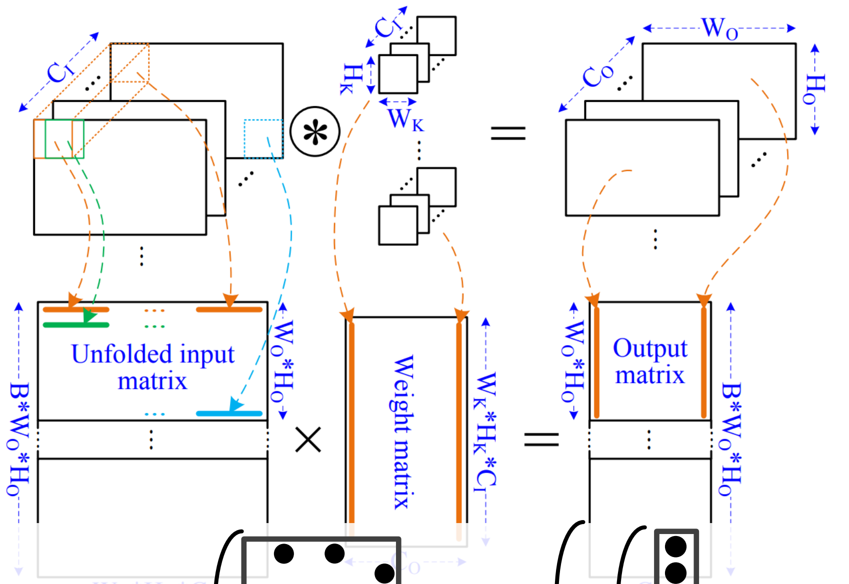


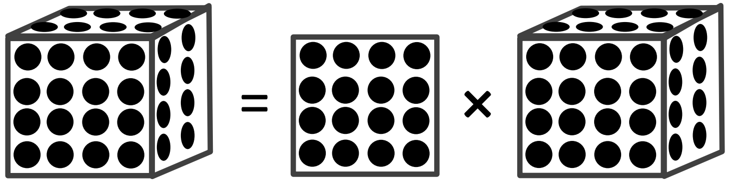


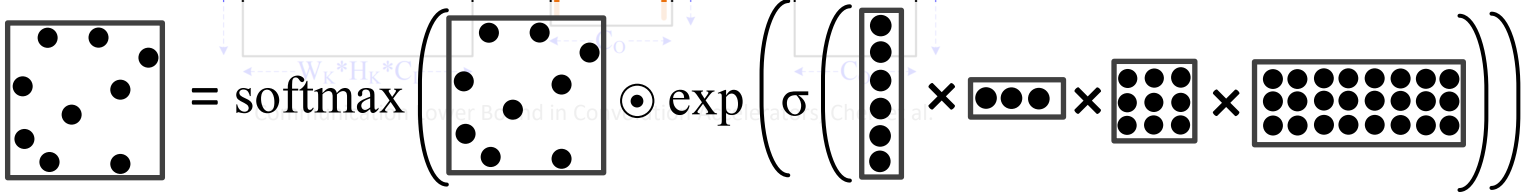
Enabling Simulation at the Fifth Rung of DFT: Large Scale RPA Calculations with Excellent Time to Solution, Del Ben et al.



A Data-Centric Approach to Extreme-Scale Ab initio Dissipative Quantum Transport Simulations, Ziogas et al.

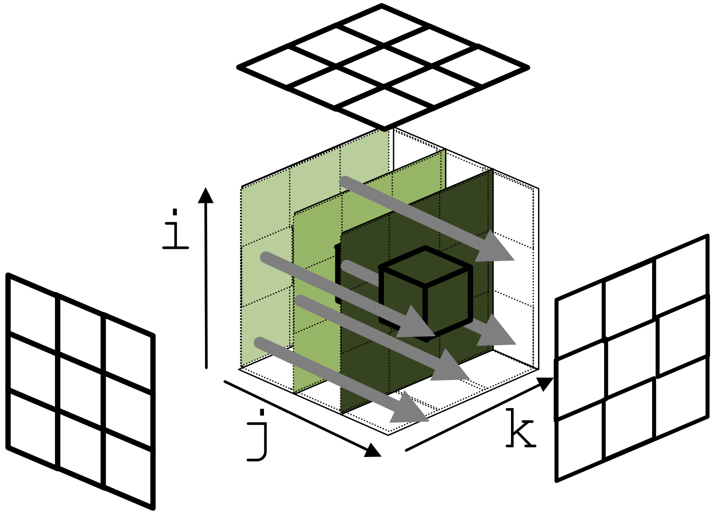






$$ij, jk \rightarrow ik$$

Einstein summation notation (einsum)




```
for i in range(N):
  for j in range(N):
    for k in range(N):
      C[i,k]+=A[i,j]*B[j,k]
```

$$ij, jk \rightarrow ik$$

Einstein summation notation (einsum)

Matrix-matrix multiplication

Tensor and matrix chains

Long, higher order contraction chains

Einsum

$ij, jk \rightarrow ik$

$ij, jk, kl \rightarrow il$

$ij, jk, kl, lm \rightarrow im$

$ijk, ja, ka \rightarrow ia$

$ijk, ia, ka \rightarrow ja$

$ijk, ia, ja \rightarrow ka$

$ijklm, ja, ka, la, ma \rightarrow ia$

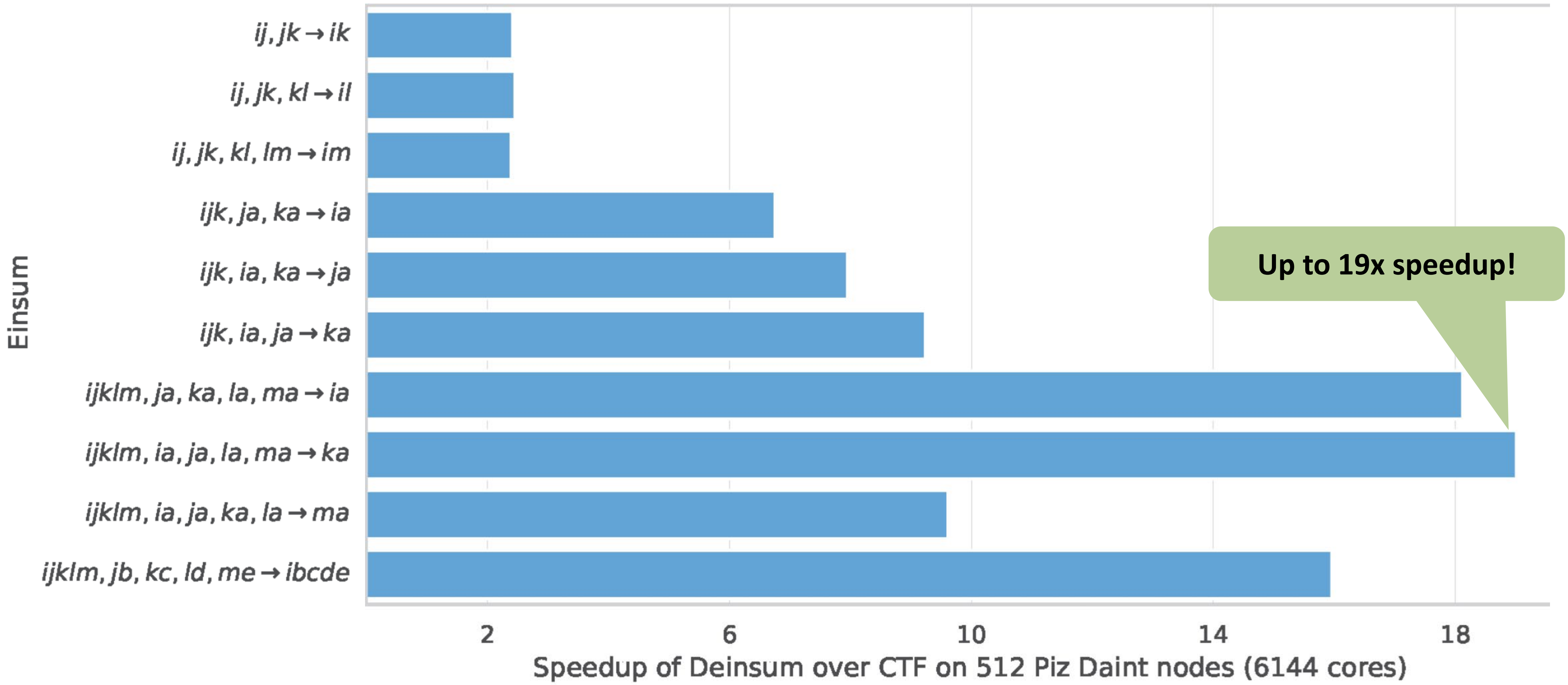
$ijklm, ia, ja, la, ma \rightarrow ka$

$ijklm, ia, ja, ka, la \rightarrow ma$

$ijklm, jb, kc, ld, me \rightarrow ibcde$

$$ij, jk \rightarrow ik$$

Einstein summation notation (einsum)



Deinsum: Practically I/O Optimal Multi-Linear Algebra

Simple Overlap Access Pattern (SOAP) data movement model

1 Input

$$ijk, ja, ka, al \rightarrow il$$

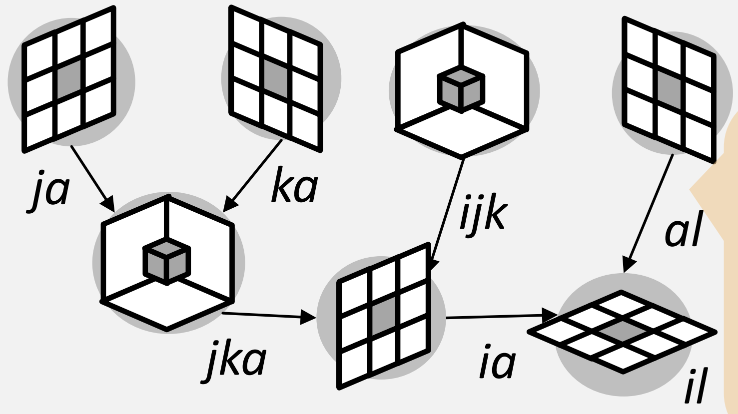
einsum, string

2 Split to binary operations

$$ja, ka \rightarrow jka$$
$$ijk, jka \rightarrow ia$$
$$ia, al \rightarrow il$$

uses operation associativity to minimize #ops

3 Communication-optimal schedules



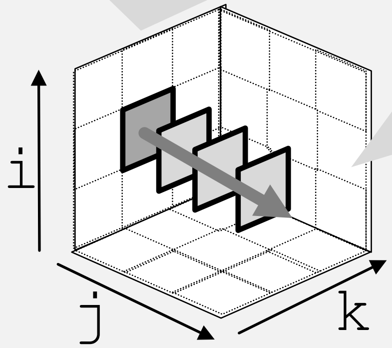
optimal tiling, parallel distribution, and kernel fusion

block-distributions

4 Iteration spaces and distribution

Iteration space partition: MPI_Cart_sub

Distribute initial data: MPI_Broadcast



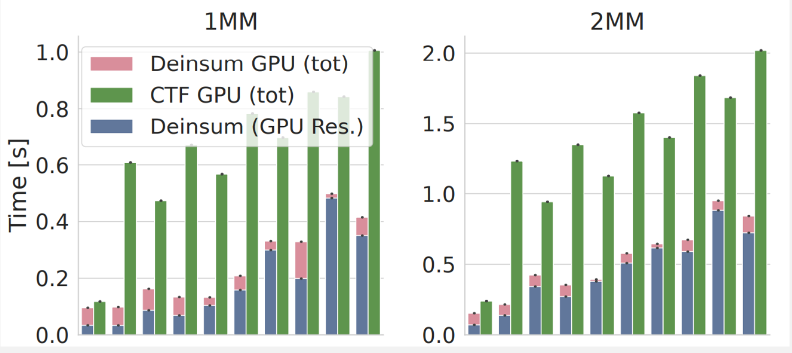
5 Automated code generation

```
for j in range(NJ//P0J):
 for k in range(NK//P0K):
  for a in range(NA//P0A):
   t0[j, k, a] += A[j, a] * B[k, a]

t1 = np.tensordot(X, t0,
        axes=([1, 2], ...
mpi.Allreduce(t1, comm=grid0 ...

t2 = deinsum.Redistr...
          comm1=grid0, ...
```

auto generating DaCe-Python code compiled to shared library

6 Results

- up to 19x speedup over SotA
- CPU and GPU support



Input

1

$$ijk, ja, ka, al \rightarrow il$$

```
for i in range(NI):
    for j in range(NJ):
        for k in range(NK):
            for l in range(NL):
                for a in range(NA):
                    out[i,l]+=X[i,j,k]*A[j,a]*B[k,a]*C[a,l]
```

Matrix-Matrix Product

naive implementation
$4N_iN_jN_kN_lN_a$ ops

Matricized Tensor Times Khatri-Rao Product (MTTKRP)

used in CANDECOMP/PARAFAC (CP) decomposition

Split to Binary Operations

2

$$ijk, ja, ka, al \rightarrow il$$

$$ja, ka \rightarrow jka$$
$$ijk, jka \rightarrow ia$$
$$ia, al \rightarrow il$$

using Python module `opt_einsum`

Split to Binary Operations

2

$ja, ka \rightarrow jka$

```
for j in range(NJ):
    for k in range(NK):
        for a in range(NA):
            t0[j,k,a]+=A[j,a]*B[k,a]
```

Khatri-Rhao Product (KRP)

$2N_jN_kN_a$ ops

$ijk, jka \rightarrow ia$

```
for i in range(NI):
    for j in range(NJ):
        for k in range(NK):
            for a in range(NA):
                t1[i,a]+=X[i,j,k]*t0[j,k,a]
```

```
t1 = np.tensordot(X, t0, axes=([1,2], [0,1]))
```

$2N_iN_jN_kN_a$ ops

Tensor DOT Product (TDOT)

$ia, al \rightarrow il$

```
for i in range(NI):
    for l in range(NL):
        for a in range(NA):
            out[i,l]+=t1[i,a]*C[a,l]
```

```
out = t1 @ C
```

Matrix-Matrix Product (GEMM)

$2N_iN_lN_a$ ops

Minimizing number of operations

2

```
for j in range(NJ):
    for k in range(NK):
        for a in range(NA):
            t0[j,k,a]+=A[j,a]*B[k,a]
```

```
t1 = np.tensordot(X, t0, axes=([1,2], [0,1]))
```

```
out = t1 @ C
```

$2N_iN_jN_kN_a$
$+2N_jN_kN_a$
$+2N_iN_lN_a$ ops

asymptotically
$2N_iN_jN_kN_a$ ops

$<$

```
for i in range(NI):
    for j in range(NJ):
        for k in range(NK):
            for l in range(NL):
                for a in range(NA):
                    out[i,l]+=(
                      X[i,j,k]*A[j,a]*
                      B[k,a]*C[a,l])
```

$4N_iN_jN_kN_lN_a$ ops

Minimizing communication

3

Minimizing communication = Maximizing data reuse = Data reuse within a kernel: ***tiling*** + Data reuse across kernels: ***kernel fusion***

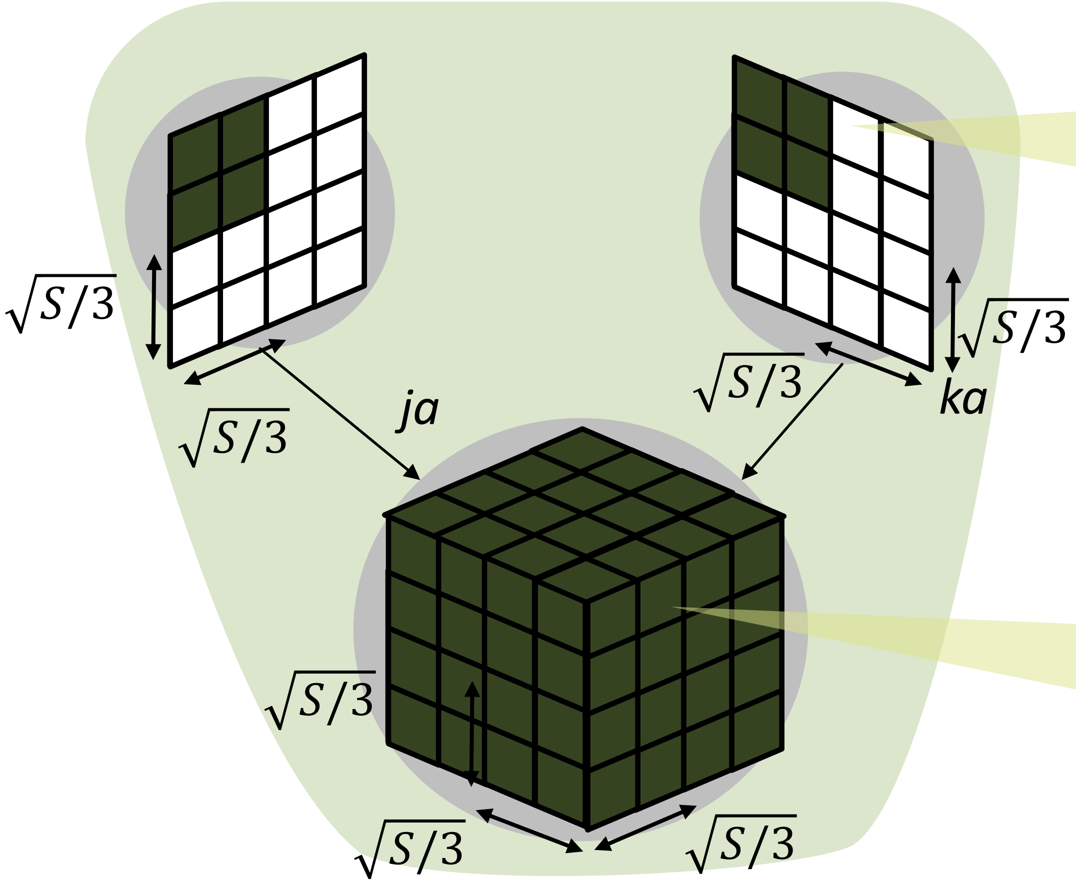
$ijk, ja, ka, al \rightarrow il$ $ja, ka \rightarrow jka$ $ijk, jka \rightarrow ia$ $ia, al \rightarrow il$

Minimizing communication

3

Minimizing communication = Maximizing data reuse = Data reuse within a kernel: ***tiling*** + Data reuse across kernels: ***kernel fusion***

$ijk, ja, ka, al \rightarrow il$ $ja, ka \rightarrow jka$ $ijk, jka \rightarrow ia$ $ia, al \rightarrow il$



Inputs are too big to fit in local memory! Can store up to S elements at once!

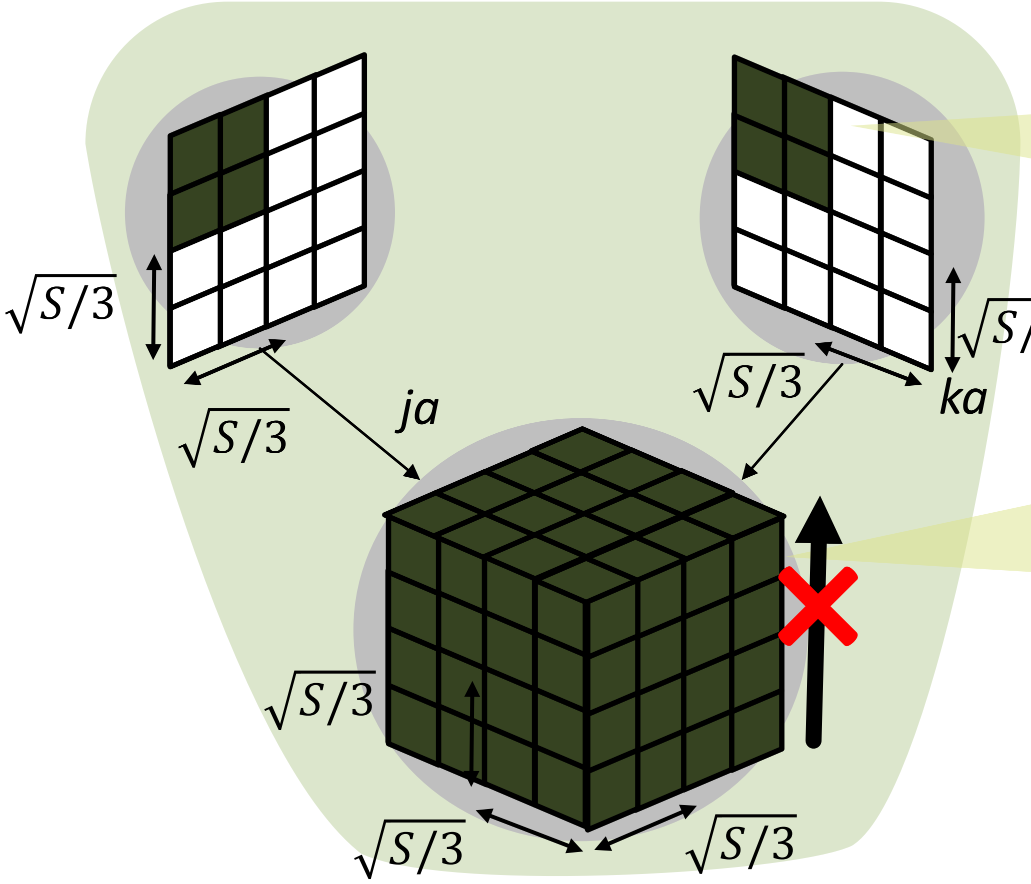
Data movement optimal classical matrix-matrix multiplication (SC'19)

Minimizing communication

3

Minimizing communication = Maximizing data reuse = Data reuse within a kernel: ***tiling*** + Data reuse across kernels: ***kernel fusion***

$ijk, ja, ka, al \to il$ $ja, ka \to jka$ $ijk, jka \to ia$ $ia, al \to il$



Inputs are too big to fit in local memory! Can store up to ***S*** elements at once!

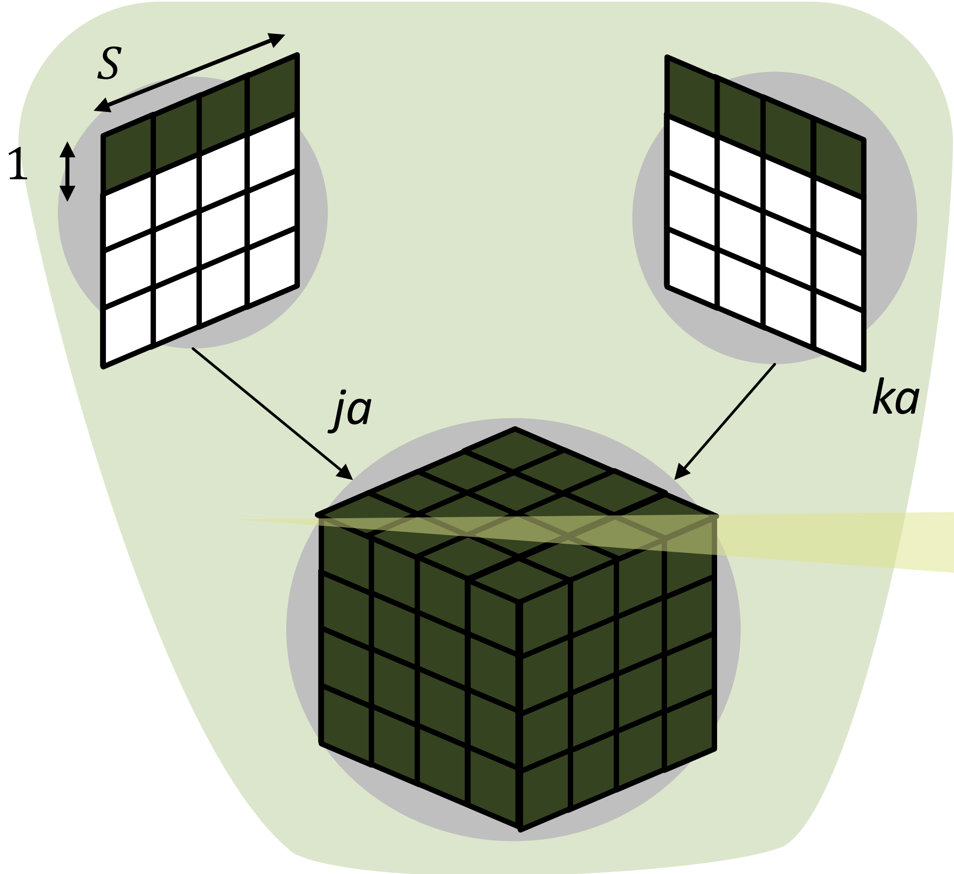
Really?
We can do better!
We don't reduce **(contract)** mode ***a***.
No need to keep intermediate results!

Minimizing communication

3

Minimizing communication = Maximizing data reuse = Data reuse within a kernel: ***tiling*** + Data reuse across kernels: ***kernel fusion***

$ijk, ja, ka, al \rightarrow il$ $ja, ka \rightarrow jka$ $ijk, jka \rightarrow ia$ $ia, al \rightarrow il$



One load operation: $\boldsymbol{S}$ arithmetic operations

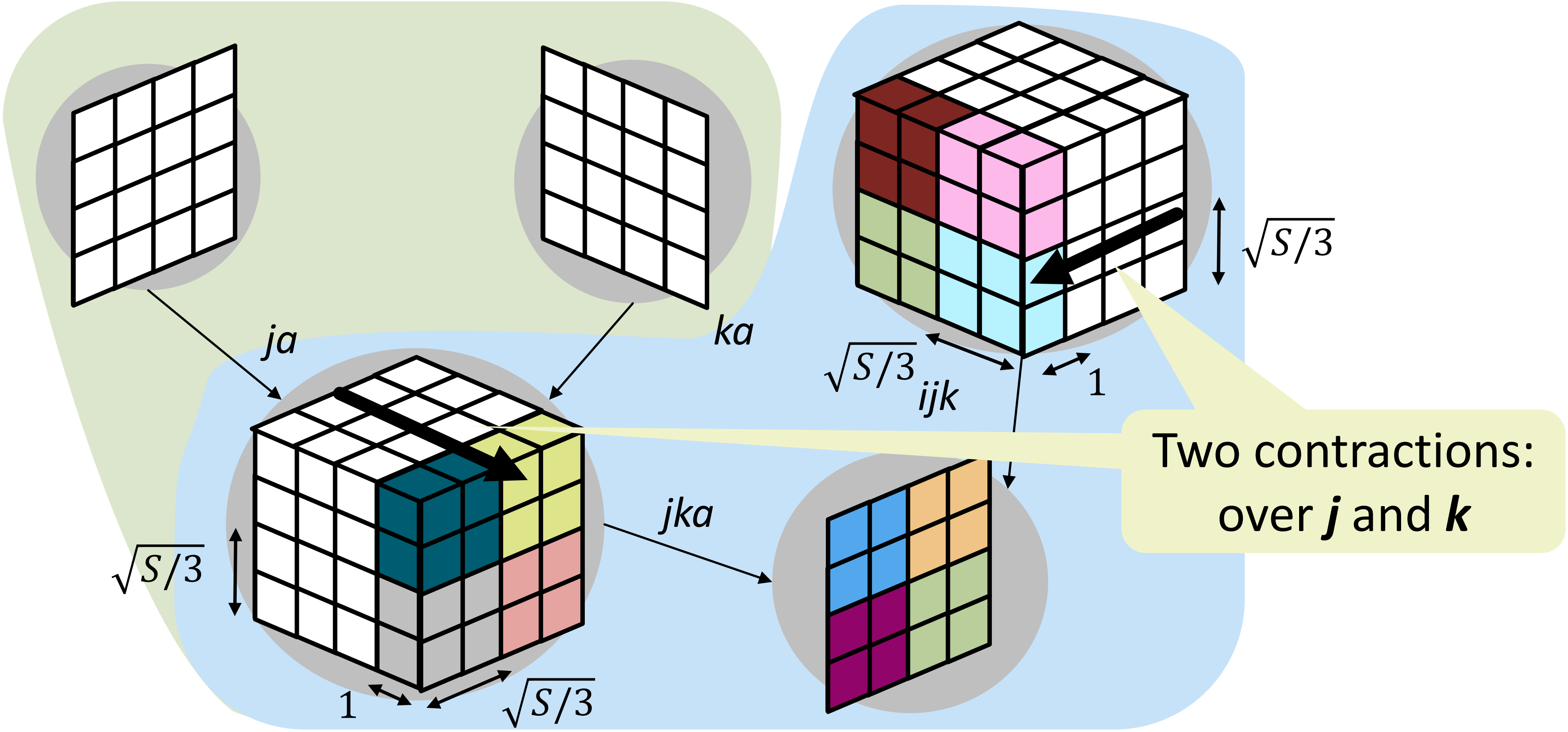
("cubic" partitioning: $\sqrt{\boldsymbol{S}}/\mathbf{2}$ arithm. ops per load)

Minimizing communication

3

Minimizing communication = Maximizing data reuse = Data reuse within a kernel: ***tiling*** + Data reuse across kernels: ***kernel fusion***

$ijk, ja, ka, al \rightarrow il$ $ja, ka \rightarrow jka$ $ijk, jka \rightarrow ia$ $ia, al \rightarrow il$

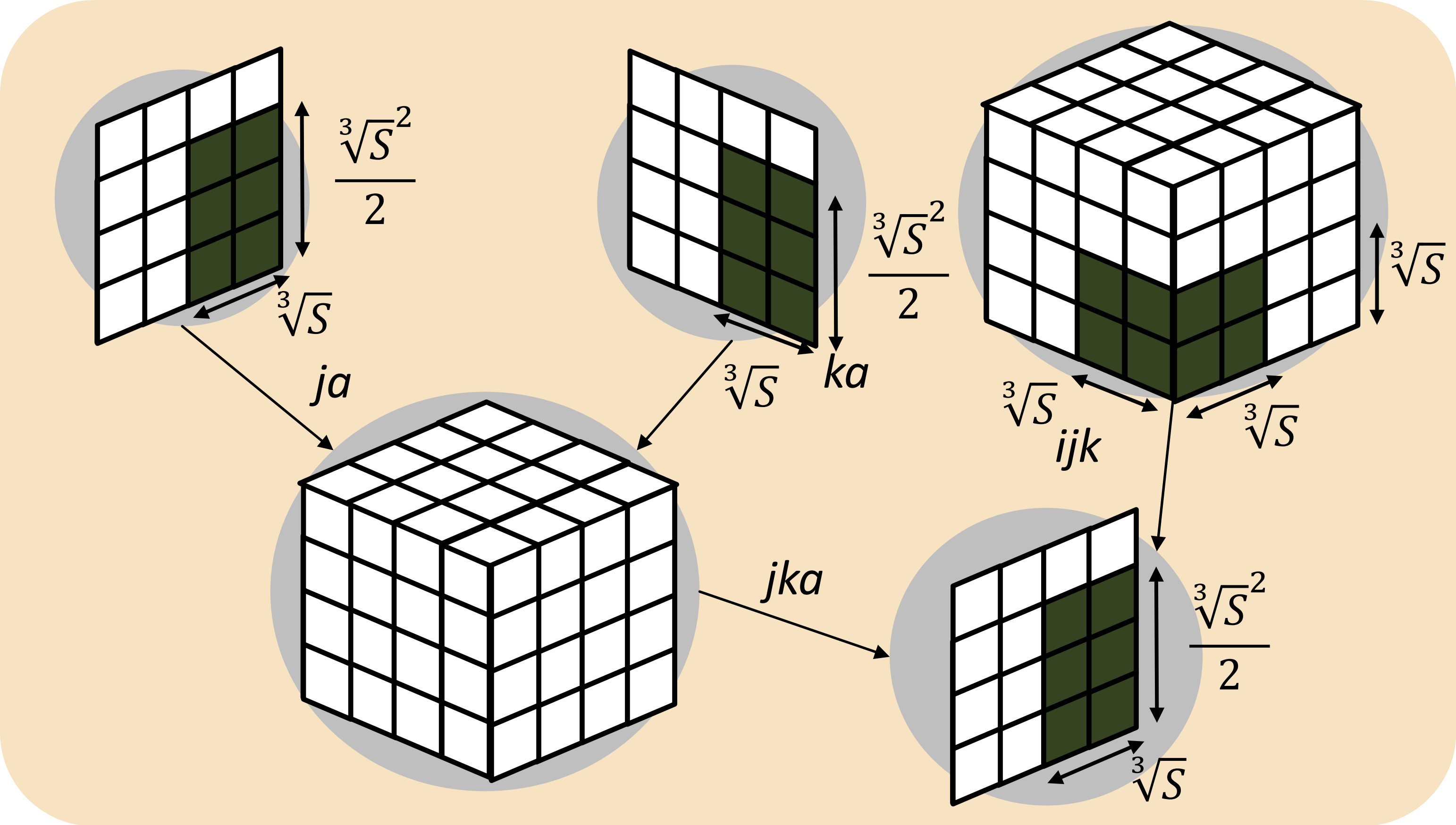


Minimizing communication

3

Minimizing communication = Maximizing data reuse = Data reuse within a kernel: ***tiling*** + Data reuse across kernels: ***kernel fusion***

$ijk, ja, ka, al \rightarrow il$ $ja, ka \rightarrow jka$ $ijk, jka \rightarrow ia$ $ia, al \rightarrow il$

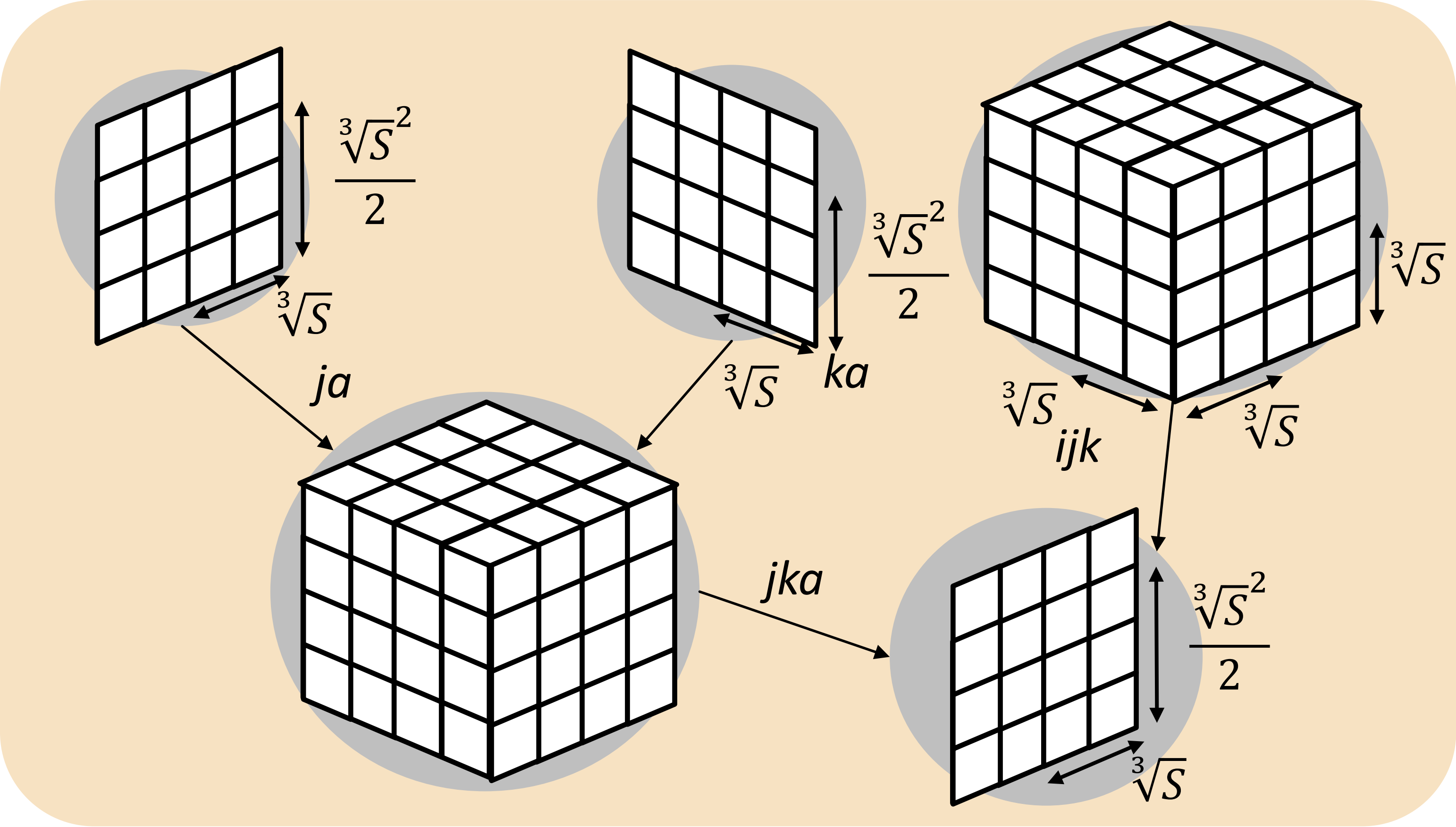


Minimizing communication

3

Minimizing communication = Maximizing data reuse = Data reuse within a kernel: ***tiling*** + Data reuse across kernels: ***kernel fusion***

$ijk, ja, ka, al \rightarrow il$ $\quad ja, ka \rightarrow jka \quad ijk, jka \rightarrow ia \quad ia, al \rightarrow il$



MTTKRP

New I/O lower bound:

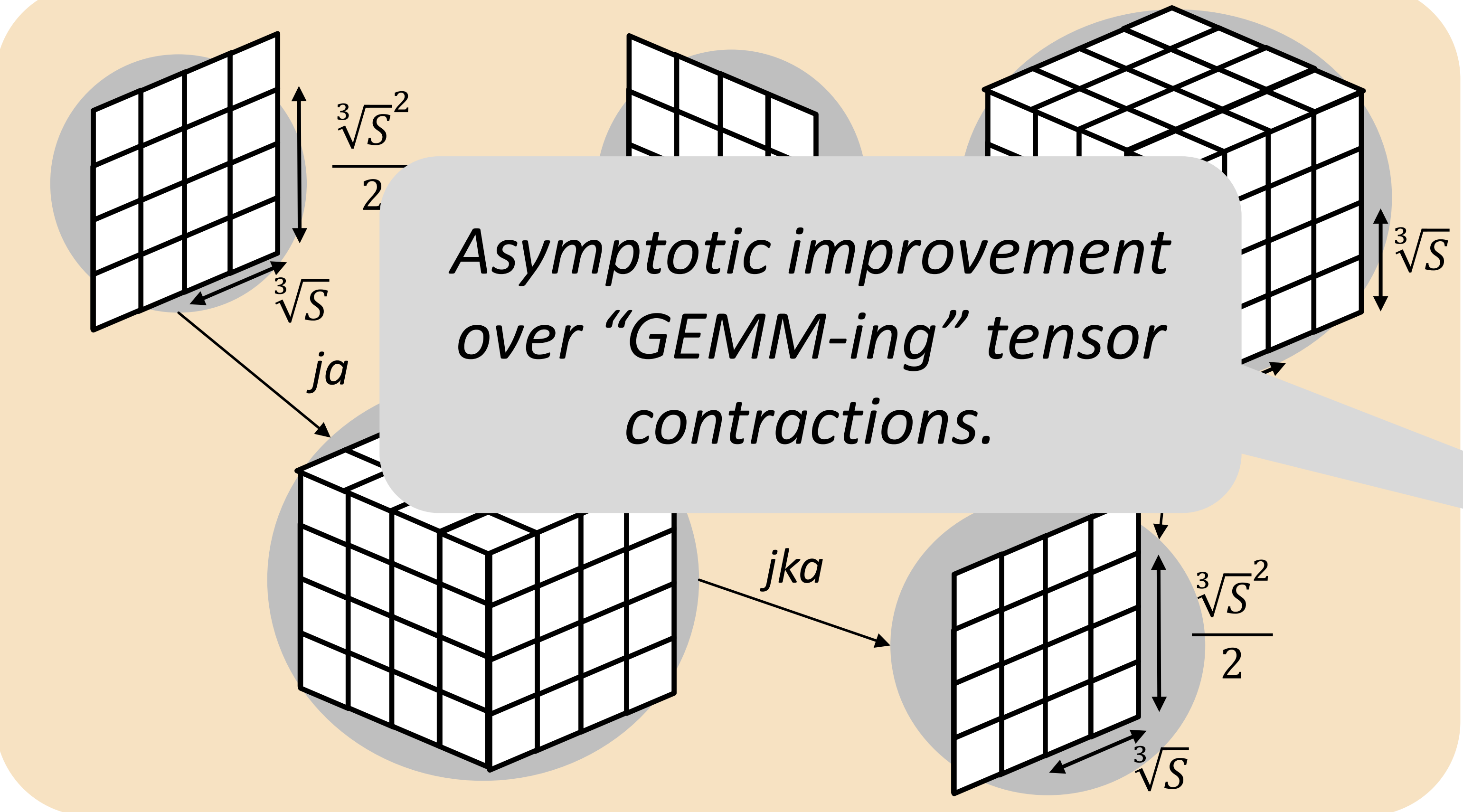
$$Q \geq \frac{3N_1N_2N_3N_4}{S^{2/3}}$$

Minimizing communication

3

Minimizing communication = Maximizing data reuse = Data reuse within a kernel: ***tiling*** + Data reuse across kernels: ***kernel fusion***

$ijk, ja, ka, al \rightarrow il$ $ja, ka \rightarrow jka$ $ijk, jka \rightarrow ia$ $ia, al \rightarrow il$



Asymptotic improvement over "GEMM-ing" tensor contractions.

MTTKRP

New I/O lower bound:

$$Q \geq \frac{3N_1N_2N_3N_4}{S^{2/3}}$$

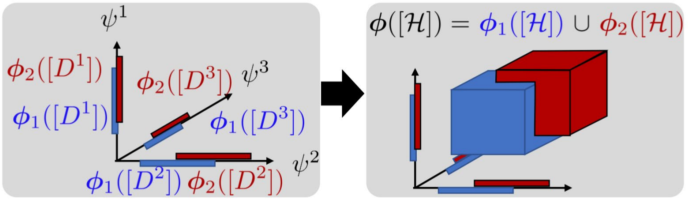
Minimizing communication

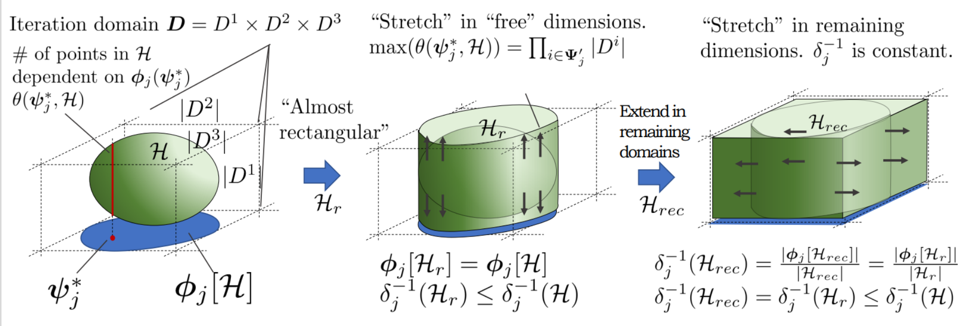
3

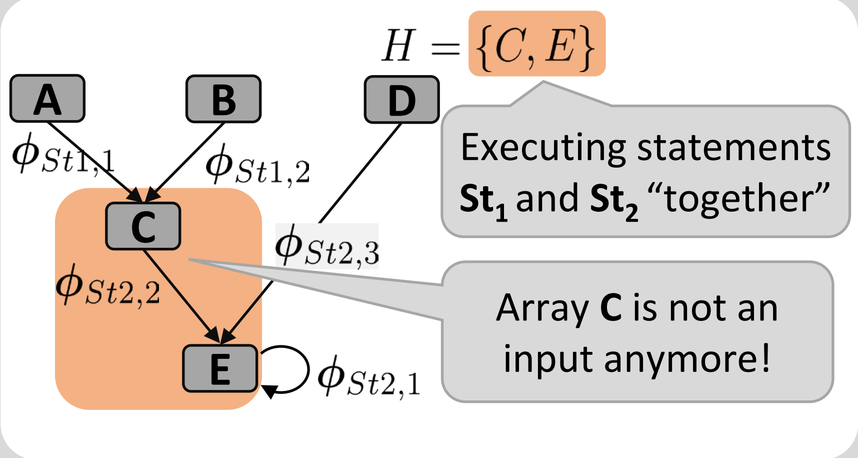
Minimizing communication = Maximizing data reuse = Data reuse within a kernel: ***tiling*** + Data reuse across kernels: ***kernel fusion***

Simple Overlap Access Program (SOAP) data movement model

Iteration vector (3 iteration variables)	$\psi = [\psi^1, \psi^2, \psi^3]$
Access function vector (2 components)	$\phi(\psi) = [\phi_1(\psi), \phi_2(\psi)]$
Iteration variables' ranges	$\psi^1 \in D^1, \psi^2 \in D^2, \psi^3 \in D^3$

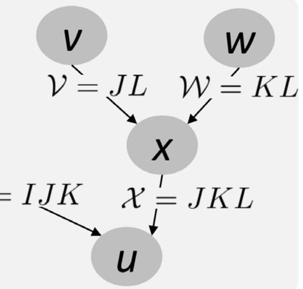






MTTKRP definition	$u_{il} = \sum_{j,k} t_{ijk} v_{jl} w_{kl}$
opt_einsum decomposition	1. $x_{jkl} = v_{jl} w_{kl}$ 2. $u_{il} = \sum_{j,k} t_{ijk} x_{jkl}$

SDG



I/O lower bound

$\max I \cdot J \cdot K \cdot L$ s.t.

$I \cdot J \cdot K + J \cdot L + K \cdot L \leq X$

Solution:

$\rho = \frac{S^{2/3}}{3}$

$Q = \frac{|V|}{\rho} = \frac{3N_1N_2N_3N_4}{S^{2/3}}$

MTTKRP

New I/O lower bound:

$$Q \geq \frac{3N_1N_2N_3N_4}{S^{2/3}}$$

Minimizing communication

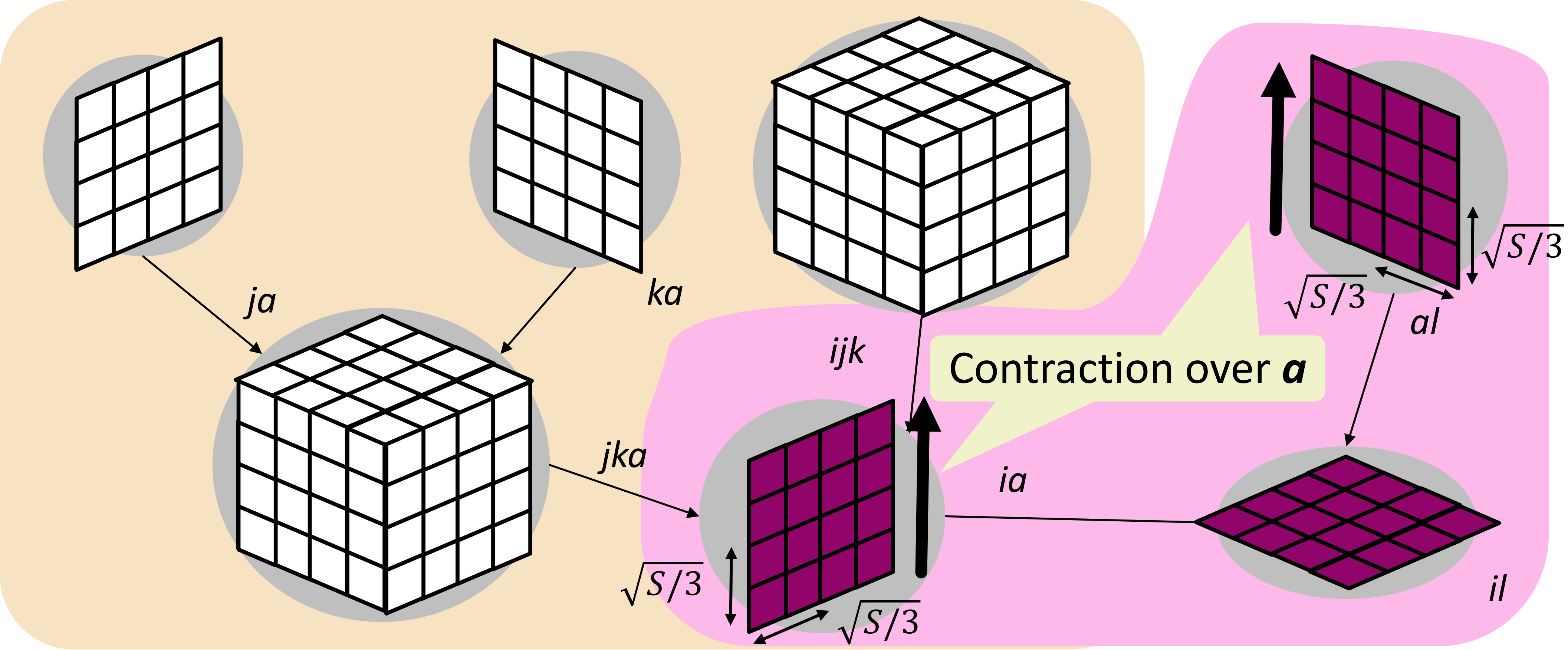
3

Minimizing communication = Maximizing data reuse = Data reuse within a kernel: ***tiling*** + Data reuse across kernels: ***kernel fusion***

$ijk, ja, ka, al \rightarrow il$

$ja, ka \rightarrow jka \quad ijk, jka \rightarrow ia$

$ia, al \rightarrow il$



Minimizing communication

3

Minimizing communication = Maximizing data reuse = Data reuse within a kernel: ***tiling*** + Data reuse across kernels: ***kernel fusion***

COMMUNICATION OPTIMAL:

Tiling

Kernel fusion

Parallel data decomposition

For a given input einsum

Deinsum: Practically I/O Optimal Multi-Linear Algebra

Simple Overlap Access Pattern (SOAP) data movement model

1 Input

$$ijk, ja, ka, al \rightarrow il$$

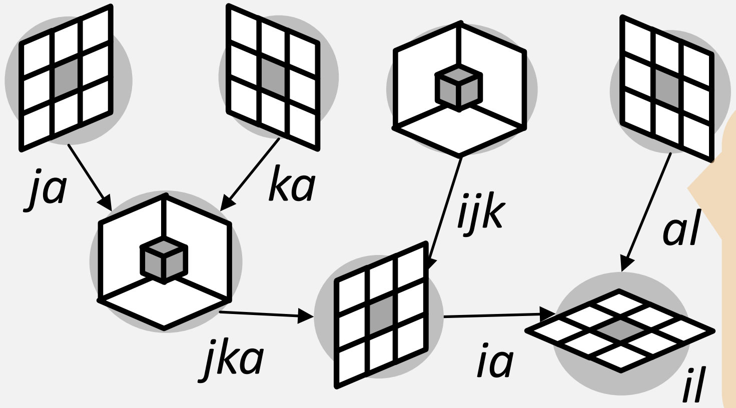
einsum, string

2 Split to binary operations

$$ja, ka \rightarrow jka$$
$$ijk, jka \rightarrow ia$$
$$ia, al \rightarrow il$$

uses operation associativity to minimize #ops

3 Communication-optimal schedules



Optimal tiling, parallel distribution, and kernel fusion

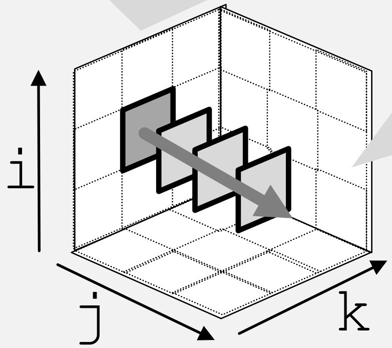
Deinsum: Practically I/O Optimal Multi-Linear Algebra

block-distributions

4 Iteration spaces and distribution

Iteration space partition: `MPI_Cart_sub`

Distribute initial data: `MPI_Broadcast`



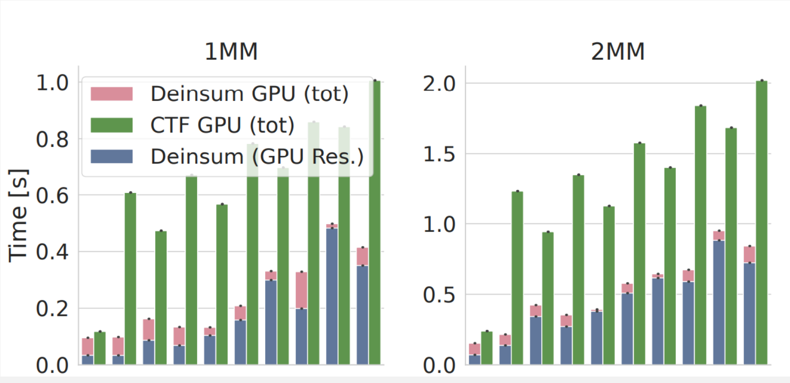
5 Automated code generation

```
for j in range(NJ//P0J):
 for k in range(NK//P0K):
  for a in range(NA//P0A):
   t0[j, k, a] += A[j, a] * B[k, a]

t1 = np.tensordot(X, t0,
        axes=([1, 2], [0, 1]))
mpi.Allreduce(t1, comm=grid0 t1)

t2 = deinsum.Redistri...
         comm1=grid0, comm2=grid1)
```

Auto generating DaCe-Python code compiled to shared library

6 Results

- **up to 19x speedup over SotA**
- **CPU and GPU support**



Grouping Operations

3 ➡ 4

```
for j in range(NJ):
    for k in range(NK):
        for a in range(NA):
            t0[j,k,a]+=A[j,a]*B[k,a]
```

```
t1 = np.tensordot(X, t0, axes=([1,2], [0,1]))
```

```
out = t1 @ C
```

Iteration Spaces: Global View

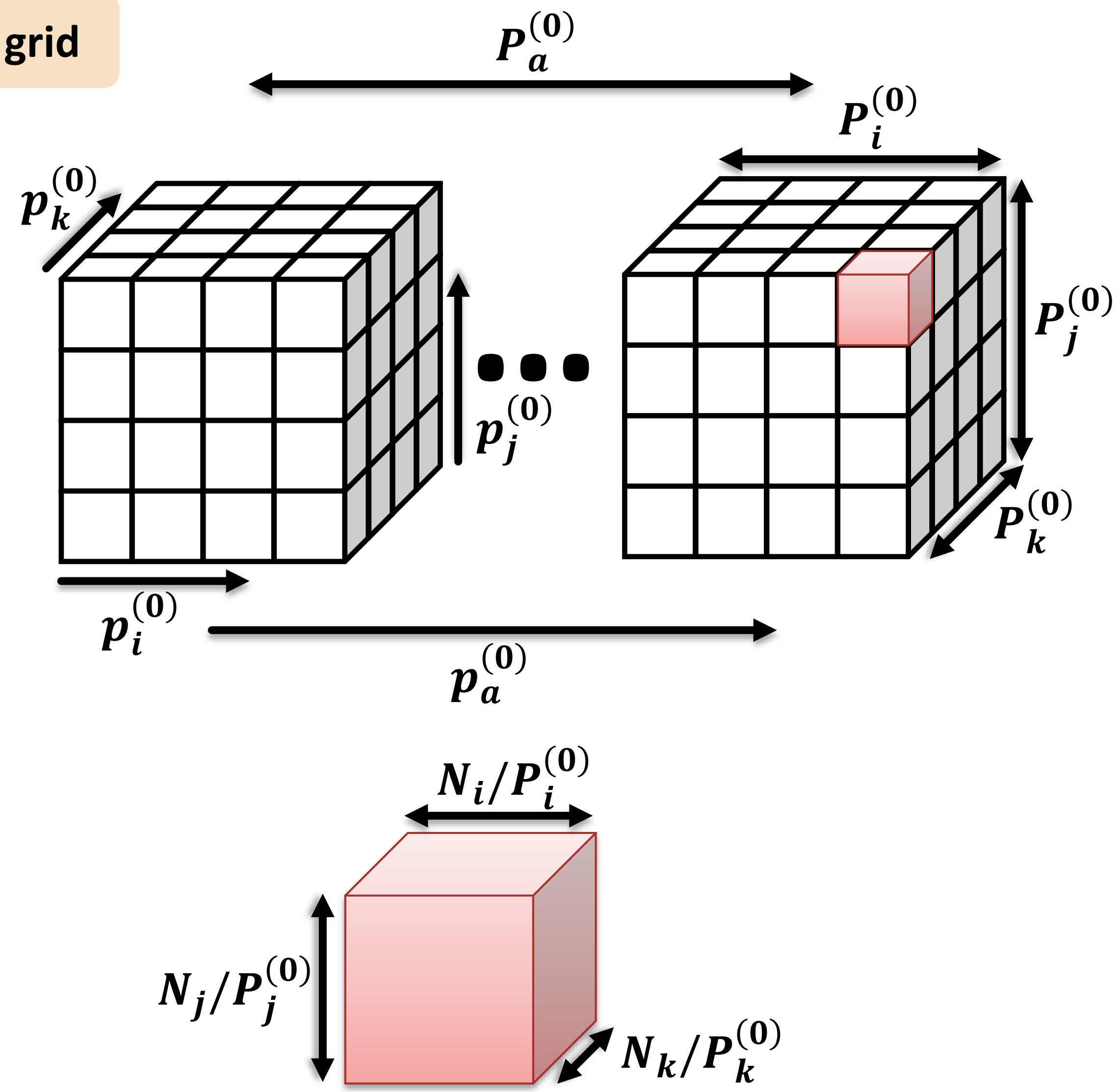
4

Cartesian process grid

$P = P_i^{(0)} P_j^{(0)} P_k^{(0)} P_a^{(0)}$

```
for j in range(NJ):
    for k in range(NK):
        for a in range(NA):
            t0[j,k,a]+=A[j,a]*B[k,a]
```

```
t1 = np.tensordot(X, t0, axes=([1,2], [0,1]))
```



Iteration Spaces: Global View

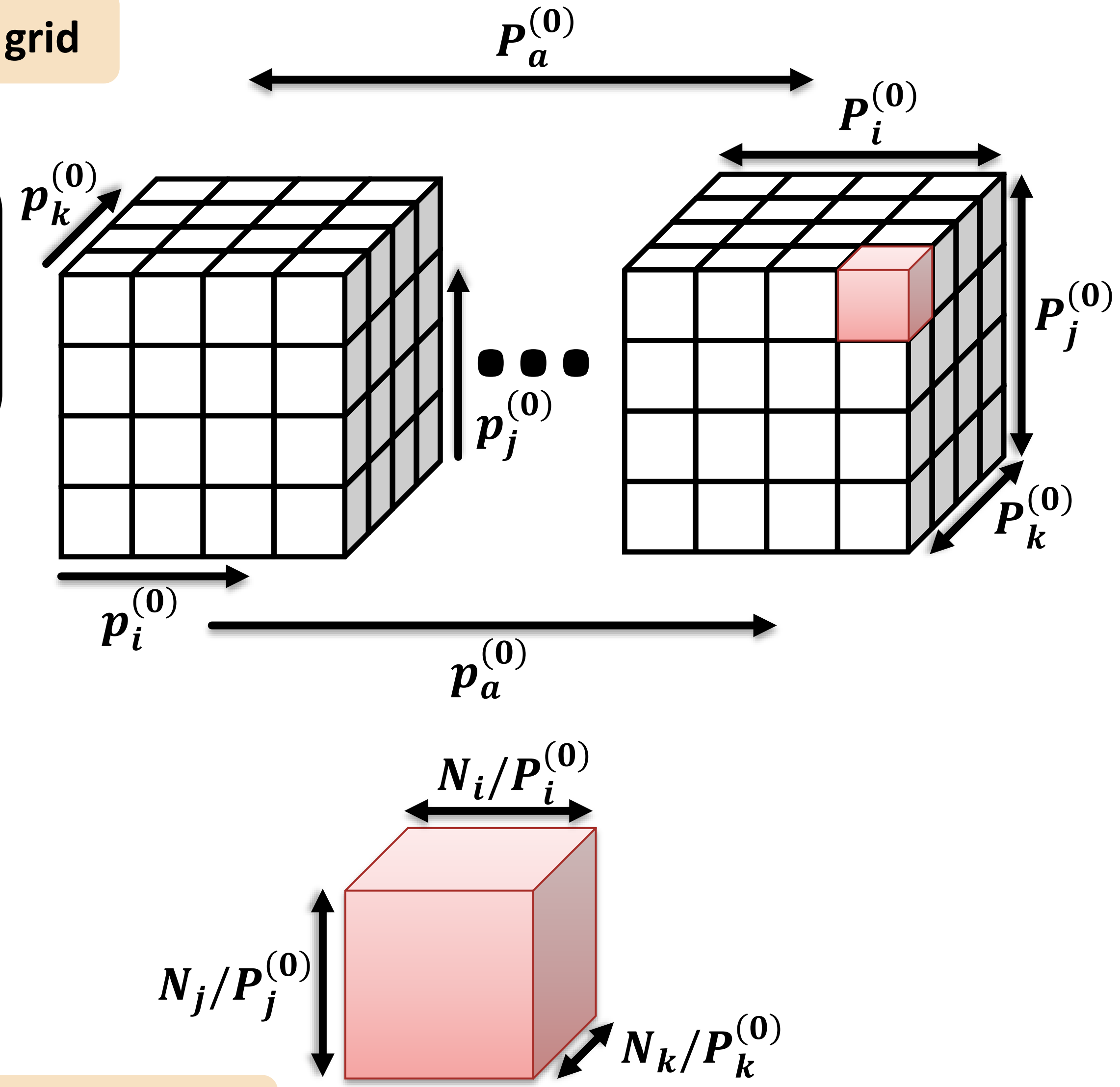
4

Cartesian process grid

$P = P_i^{(0)} P_j^{(0)} P_k^{(0)} P_a^{(0)}$

```
for j in range(p0j*NJ//P0J,(p0j+1)*NJ//P0J):
  for k in range(p0k*NK//P0K,(p0k+1)*NJ//P0J):
    for a in range(p0a*NA//P0A,(p0a+1)*NA//P0A):
      t0[j,k,a]+=A[j,a]*B[k,a]
```

```
t1[p0i*NI//P0I,(p0i+1)*NI//P0I,
   p0a*NA//P0A,(p0a+1)*NA//P0A] = np.tensordot(
    X[p0i*NI//P0I,(p0i+1)*NI//P0I,
      p0j*NJ//P0J,(p0j+1)*NJ//P0J,
      p0k*NK//P0K,(p0k+1)*NJ//P0J],
    t0[p0j*NJ//P0J,(p0j+1)*NJ//P0J,
       p0k*NK//P0K,(p0k+1)*NJ//P0J,
       p0a*NA//P0A,(p0a+1)*NA//P0A],
    axes=([1,2], [0,1]))
```



process-local slices – global coordinates

Iteration Spaces: Local View

4

$$P = P_i^{(0)} P_j^{(0)} P_k^{(0)} P_a^{(0)}$$

```
for j in range(NJ//P0J):
    for k in range(NK//P0K):
        for a in range(NA//P0A):
            t0[j,k,a]+=A[j,a]*B[k,a]
```

```
t1 = np.tensordot(X, t0, axes=([1,2], [0,1]))
```

Iteration Spaces: Local View

4

process-local slices – local coordinates

$$P = P_i^{(0)} P_j^{(0)} P_k^{(0)} P_a^{(0)}$$

```
for j in range(NJ//P0J):
    for k in range(NK//P0K):
        for a in range(NA//P0A):
            t0[j,k,a]+=A[j,a]*B[k,a]
```

```
t1 = np.tensordot(X, t0, axes=([1,2], [0,1]))
```

```
//                    {  i,    j,    k,    a}
int dims[4] =         {P0I, P0J, P0K, P0A};
int periods[4] = {  0,    0,    0,    0};
MPI_Comm grid0;
MPI_Cart_create(MPI_COMM_WORLD, 4, dims,
                periods, false, &grid0);
```

Iteration Spaces: Local View

4

process-local slices – local coordinates

$$P = P_i^{(0)} P_j^{(0)} P_k^{(0)} P_a^{(0)}$$

```
for j in range(NJ//P0J):
    for k in range(NK//P0K):
        for a in range(NA//P0A):
            t0[j,k,a]+=A[j,a]*B[k,a]
```

```
t1 = np.tensordot(X, t0, axes=([1,2], [0,1]))
```

```
//                 {  i,   j,   k,   a}
int dims[4] =      {P0I, P0J, P0K, P0A};
int periods[4] = {  0,   0,   0,   0};
MPI_Comm grid0;
MPI_Cart_create(MPI_COMM_WORLD, 4, dims,
                periods, false, &grid0);
```

$$P = P_i^{(1)} P_l^{(1)} P_a^{(1)}$$

```
out = t1 @ C
```

```
//                 {  i,   l,   a}
int dims[3] =      {P1I, P1L, P1A};
int periods[3] = {  0,   0,   0};
MPI_Comm grid1;
MPI_Cart_create(MPI_COMM_WORLD, 3, dims,
                periods, false, &grid1);
```

Iteration Spaces: Practically I/O Optimal Distribution

4

$$P = P_i^{(0)} P_j^{(0)} P_k^{(0)} P_a^{(0)}$$

```
for j in range(NJ//P0J):
    for k in range(NK//P0K):
        for a in range(NA//P0A):
            t0[j,k,a]+=A[j,a]*B[k,a]
```

```
t1 = np.tensordot(X, t0, axes=([1,2], [0,1]))
```

Optimal Tile Sizes

$$T_i = T_j = T_k = S^{1/3}, T_a = \frac{S^{2/3}}{2}$$

```
grid_ijka = {
    #       [i, j, k, a]
    1:      [1, 1, 1, 1],
    2:      [1, 1, 2, 1],
    4:      [1, 2, 2, 1],
    8:      [2, 2, 2, 1],
    12:     [2, 2, 3, 1],
    16:     [2, 2, 4, 1],
    27:     [3, 3, 3, 1],
    32:     [2, 4, 4, 1],
    64:     [4, 4, 4, 1],
    125:    [5, 5, 5, 1],
    128:    [4, 4, 8, 1],
    252:    [6, 6, 7, 1],
    256:    [4, 8, 8, 1],
    512:    [8, 8, 8, 1],
}
```

Computation and Data Distribution

4

$$P = P_i^{(0)} P_j^{(0)} P_k^{(0)} P_a^{(0)}$$

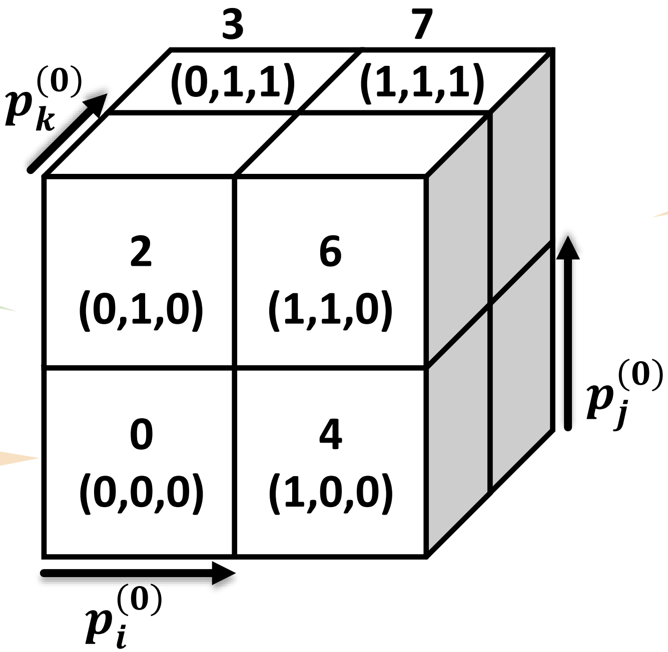
```
for j in range(NJ//P0J):
    for k in range(NK//P0K):
        for a in range(NA//P0A):
            t0[j,k,a]+=A[j,a]*B[k,a]
```

```
t1 = np.tensordot(X, t0, axes=([1,2], [0,1]))
```

X[i,j,k]

$P_a^{(0)} = 1, p_a^{(0)} = 0$

format:
process ID (rank)
$\left(p_i^{(0)}, p_j^{(0)}, p_k^{(0)}\right)$



one computation block per rank

one data X block per rank

Data Distribution: Replication

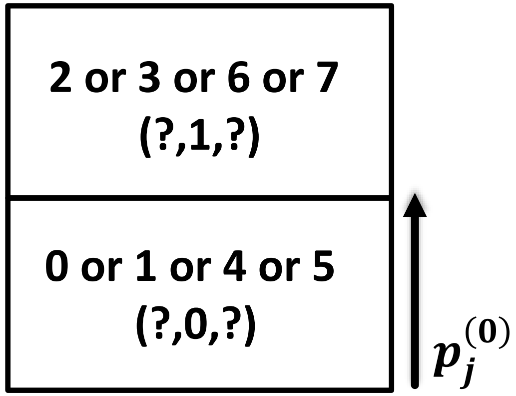
4

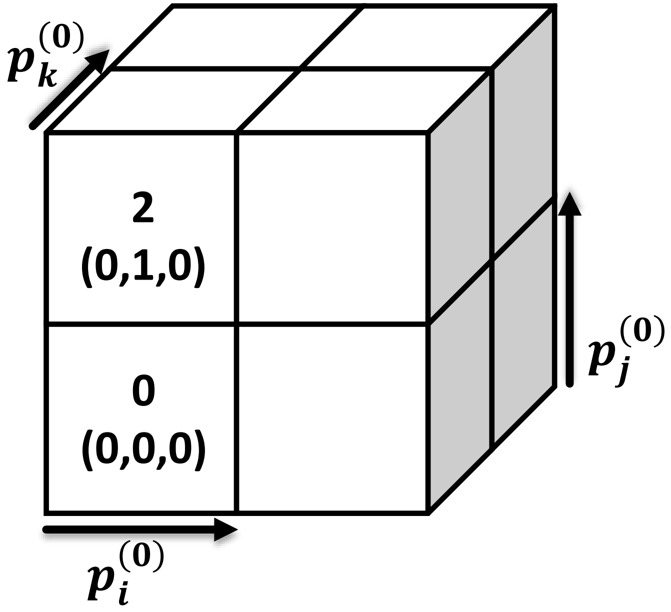
$$P = P_i^{(0)} P_j^{(0)} P_k^{(0)} P_a^{(0)}$$

```
for j in range(NJ//P0J):
    for k in range(NK//P0K):
        for a in range(NA//P0A):
            t0[j,k,a]+=A[j,a]*B[k,a]
```

```
t1 = np.tensordot(X, t0, axes=([1,2], [0,1]))
```

```
A[j,a]
```





Data Distribution: Replication

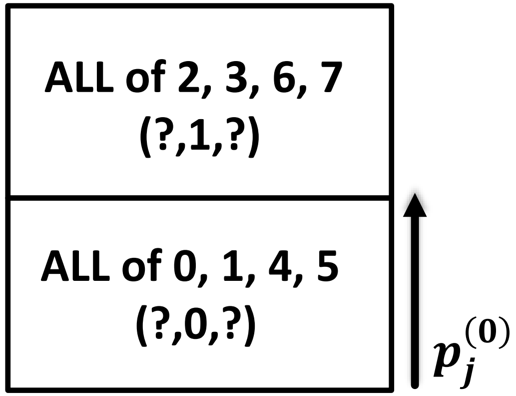
4

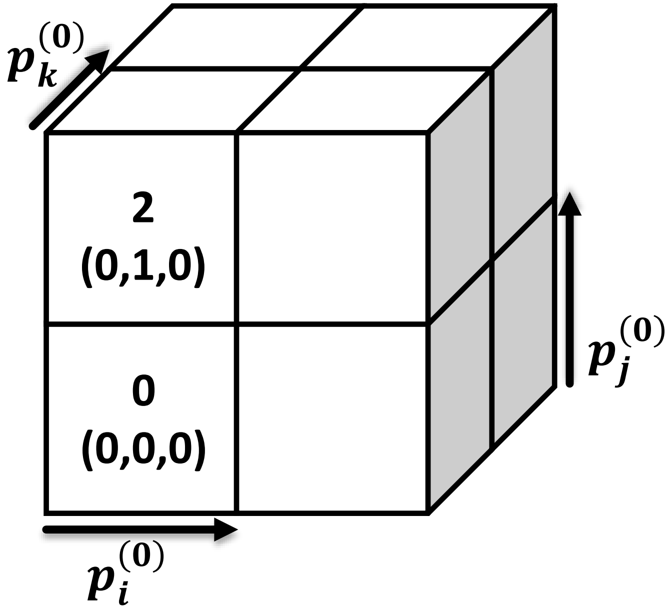
$$P = P_i^{(0)} P_j^{(0)} P_k^{(0)} P_a^{(0)}$$

```
for j in range(NJ//P0J):
    for k in range(NK//P0K):
        for a in range(NA//P0A):
            t0[j,k,a]+=A[j,a]*B[k,a]
```

```
t1 = np.tensordot(X, t0, axes=([1,2], [0,1]))
```

A[j,a]





Data Distribution: Replication

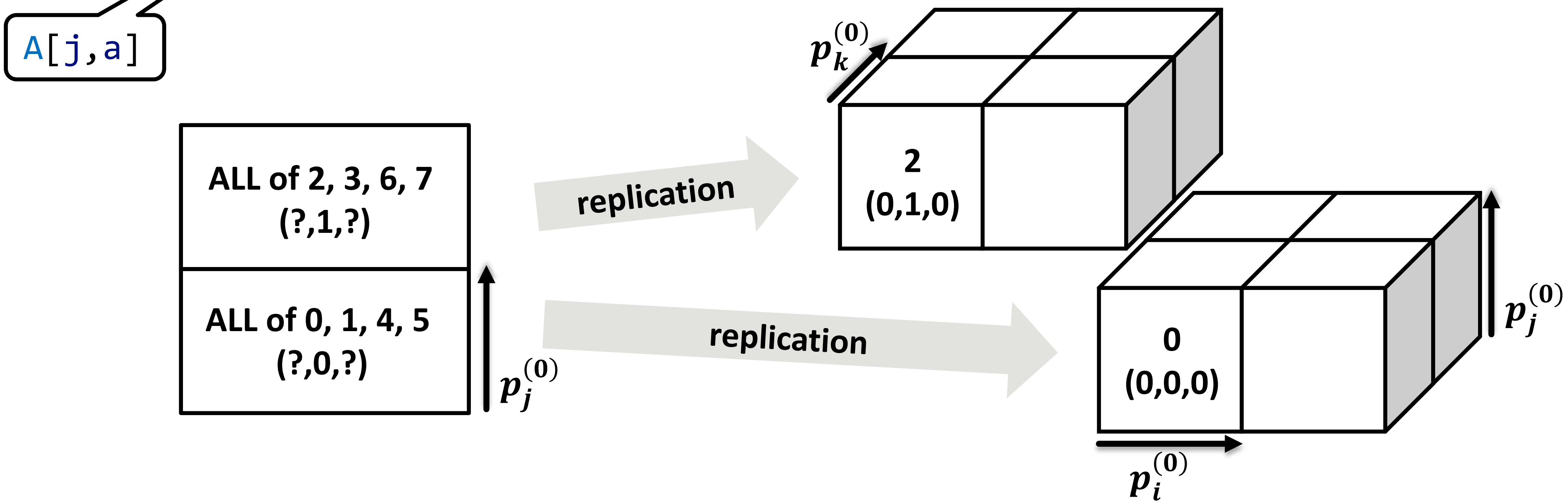
4

$$P = P_i^{(0)} P_j^{(0)} P_k^{(0)} P_a^{(0)}$$

```
for j in range(NJ//P0J):
    for k in range(NK//P0K):
        for a in range(NA//P0A):
            t0[j,k,a]+=A[j,a]*B[k,a]
```

```
t1 = np.tensordot(X, t0, axes=([1,2], [0,1]))
```

```
//                   {i, j, k, a}
int remain_A[4] = {1, 0, 1, 0};
MPI_Comm grid0_A;
MPI_Cart_sub(grid0, remain_A, &grid0_A);
MPI_Bcast(A, count, datatype, leader,
          grid0_A);
```



Data Distribution: Partial Sums

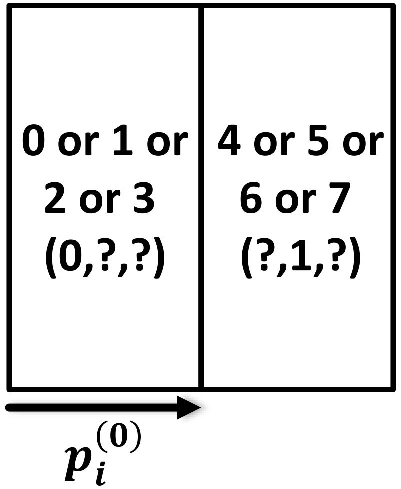
4

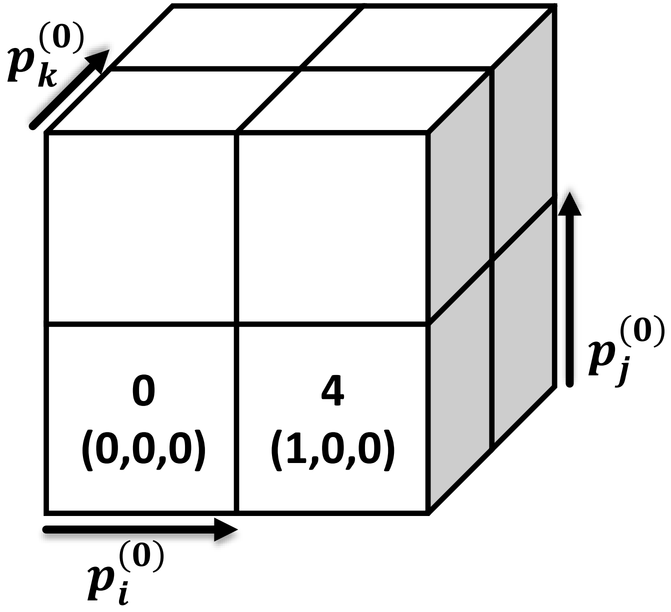
$$P = P_i^{(0)} P_j^{(0)} P_k^{(0)} P_a^{(0)}$$

```
for j in range(NJ//P0J):
    for k in range(NK//P0K):
        for a in range(NA//P0A):
            t0[j,k,a]+=A[j,a]*B[k,a]
```

```
t1 = np.tensordot(X, t0, axes=([1,2], [0,1]))
```

```
t1[i,a]
```





Data Distribution: Partial Sums

4

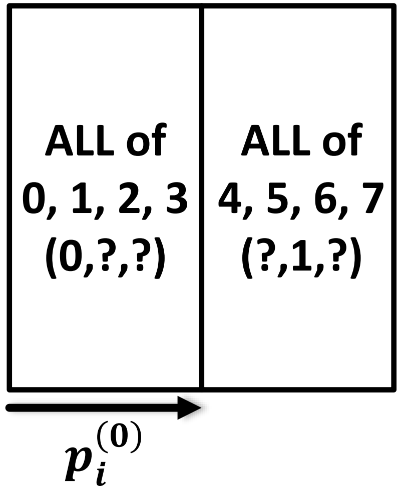
$$P = P_i^{(0)} P_j^{(0)} P_k^{(0)} P_a^{(0)}$$

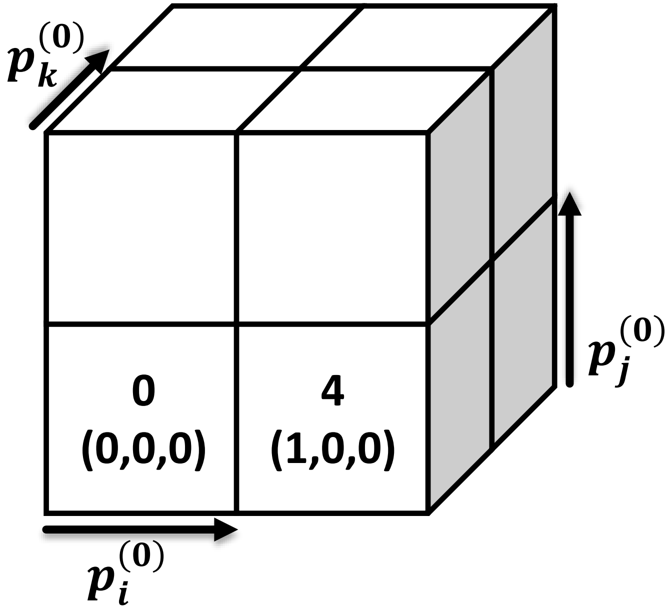
```
for j in range(NJ//P0J):
    for k in range(NK//P0K):
        for a in range(NA//P0A):
            t0[j,k,a]+=A[j,a]*B[k,a]
```

```
t1 = np.tensordot(X, t0, axes=([1,2], [0,1]))
```

```
t1[i,a]
```

output has partial sums





Data Distribution: Partial Sums

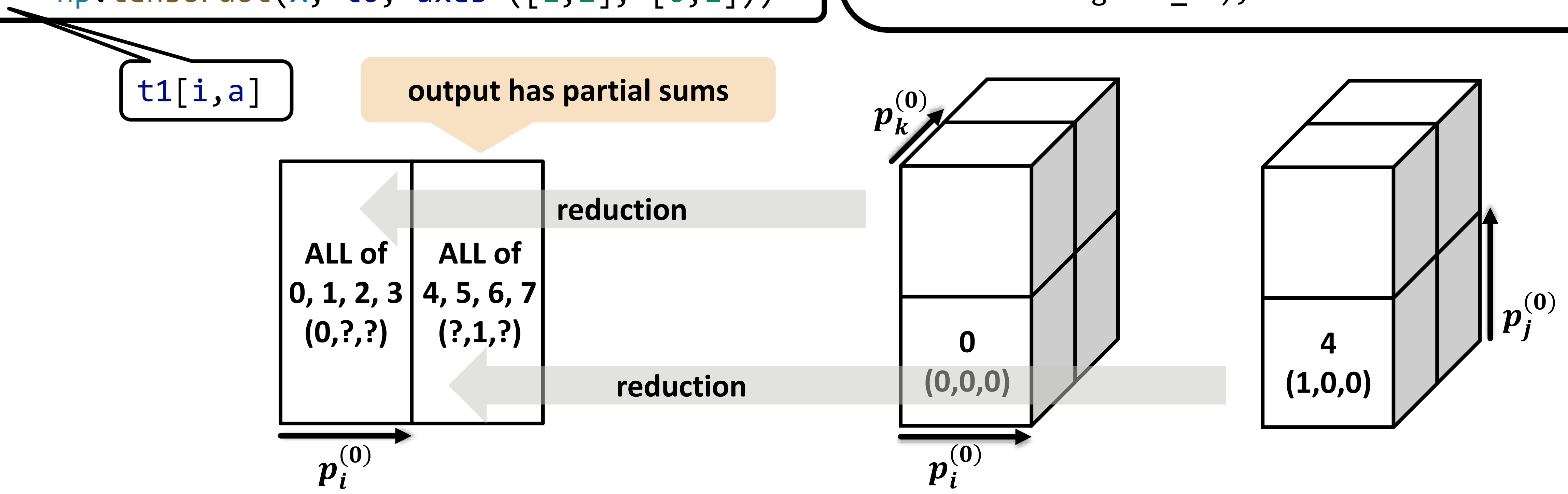
4

$$P = P_i^{(0)} P_j^{(0)} P_k^{(0)} P_a^{(0)}$$

```
for j in range(NJ//P0J):
    for k in range(NK//P0K):
        for a in range(NA//P0A):
            t0[j,k,a]+=A[j,a]*B[k,a]
```

```
t1 = np.tensordot(X, t0, axes=([1,2], [0,1]))
```

```
//                    {i, j, k, a}
int remain_t1[4] = {0, 1, 1, 0};
MPI_Comm grid0_t1;
MPI_Cart_sub(grid0, remain_t1, &grid0_t1);
MPI_Allreduce(MPI_IN_PLACE, t1, count,
              datatype, leader, MPI_SUM,
              grid0_t1);
```

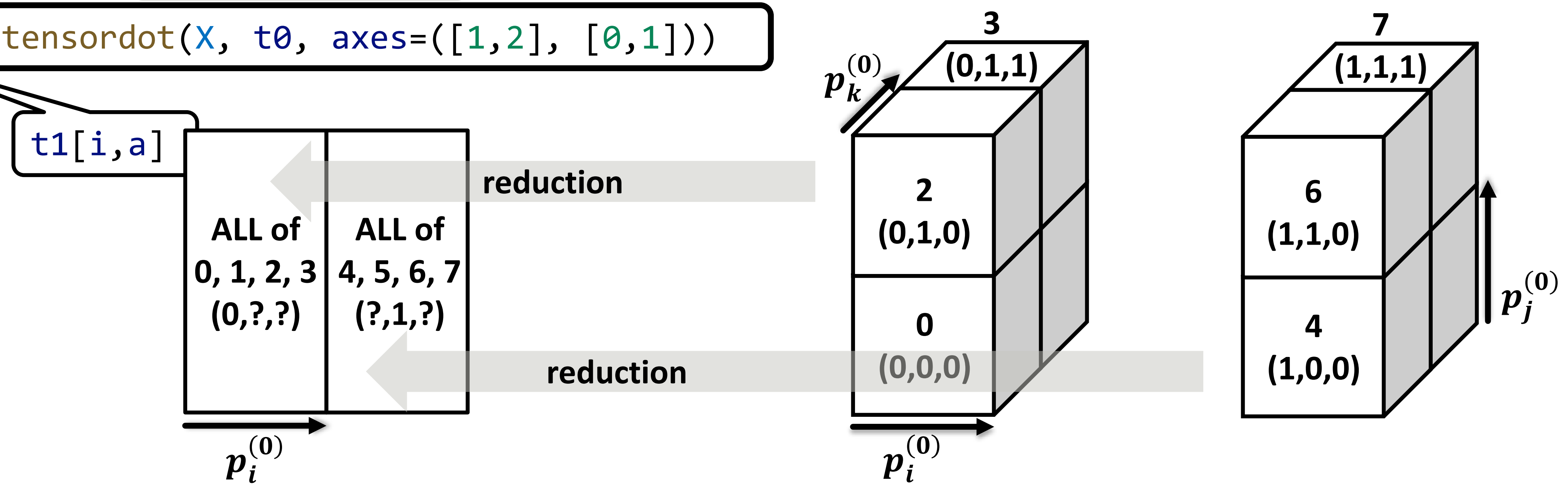


Data Redistribution

4

$$P = P_i^{(0)} P_j^{(0)} P_k^{(0)} P_a^{(0)}$$

```
t1 = np.tensordot(X, t0, axes=([1,2], [0,1]))
```

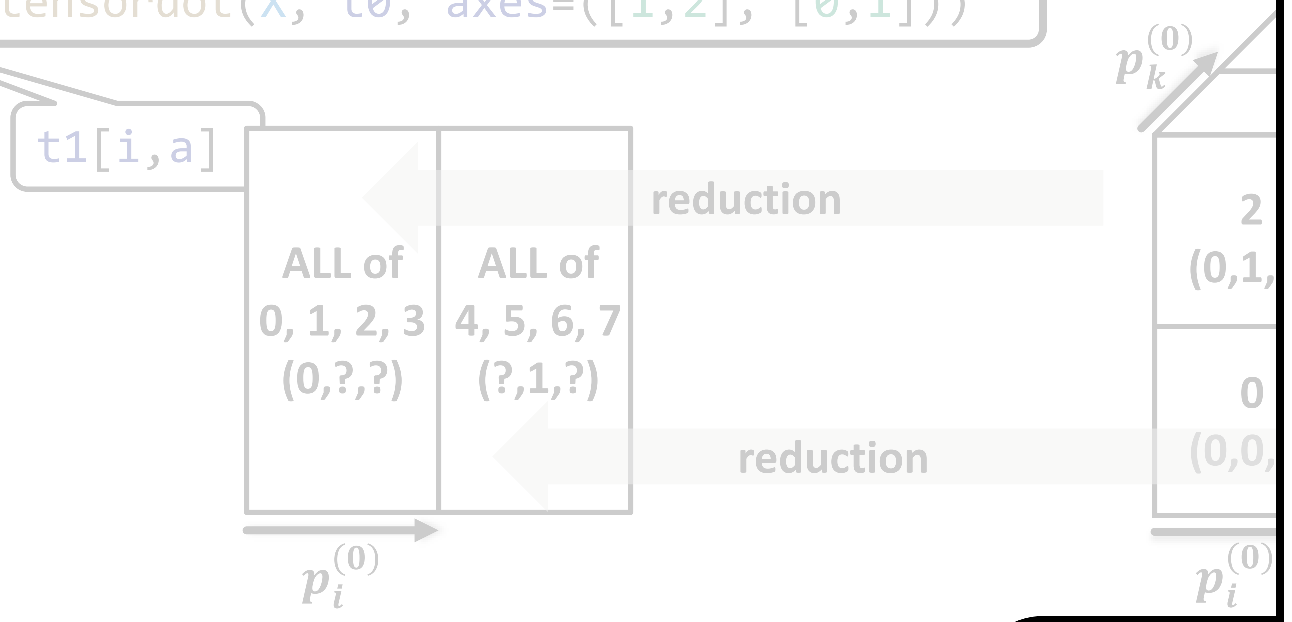


Data Redistribution

4

$$P = P_i^{(0)} P_j^{(0)} P_k^{(0)} P_a^{(0)}$$

```
t1 = np.tensordot(X, t0, axes=([1,2], [0,1]))
```



$$P = P_i^{(1)} P_l^{(1)} P_a^{(1)}$$

```
out = t1 @ C
```

```
grid_ila = {
    #      [i, l, a]
    1:     [1, 1, 1],
    2:     [1, 1, 2],
    4:     [1, 2, 2],
    8:     [2, 2, 2],
    12:    [2, 2, 3],
    16:    [2, 2, 4],
    27:    [3, 3, 3],
    32:    [2, 4, 4],
    64:    [4, 4, 4],
    125:   [5, 5, 5],
    128:   [4, 4, 8],
    252:   [6, 6, 7],
    256:   [4, 8, 8],
    512:   [8, 8, 8],
}
```

```
//
int dims[3
int period
MPI_Comm g
MPI_Cart_cre
                    periods, false, &grid1);
```

Data Redistribution

4

```
grid_ijka = {
    #      [i, j, k, a]
    1:     [1, 1, 1, 1],
    2:     [1, 1, 2, 1],
    4:     [1, 2, 2, 1],
    8:     [2, 2, 2, 1],
    12:    [2, 2, 3, 1],
    16:    [2, 2, 4, 1],
    27:    [3, 3, 3, 1],
    32:    [2, 4, 4, 1],
    64:    [4, 4, 4, 1],
    125:   [5, 5, 5, 1],
    128:   [4, 4, 8, 1],
    252:   [6, 6, 7, 1],
    256:   [4, 8, 8, 1],
    512:   [8, 8, 8, 1],
}
```

```
grid_ila = {
    #      [i, l, a]
    1:     [1, 1, 1],
    2:     [1, 1, 2],
    4:     [1, 2, 2],
    8:     [2, 2, 2],
    12:    [2, 2, 3],
    16:    [2, 2, 4],
    27:    [3, 3, 3],
    32:    [2, 4, 4],
    64:    [4, 4, 4],
    125:   [5, 5, 5],
    128:   [4, 4, 8],
    252:   [6, 6, 7],
    256:   [4, 8, 8],
    512:   [8, 8, 8],
}
```

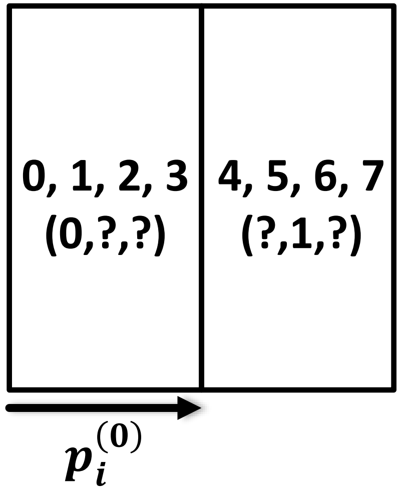


```
//
int dims[3
int period
MPI_Comm g
MPI_Cart_cre
    periods, false, &grid1);
```

Data Redistribution

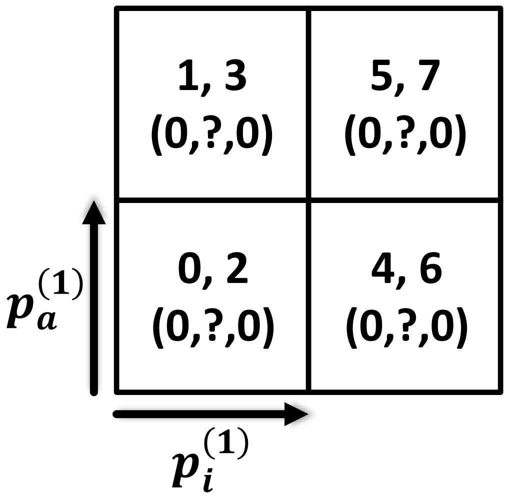
4

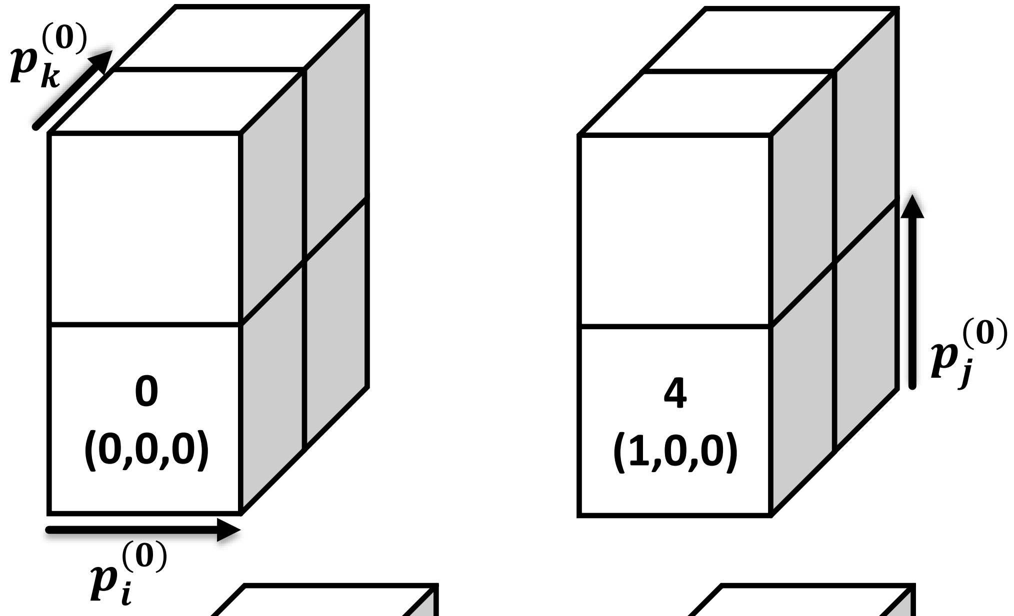
$$P = P_i^{(0)} P_j^{(0)} P_k^{(0)} P_a^{(0)}$$

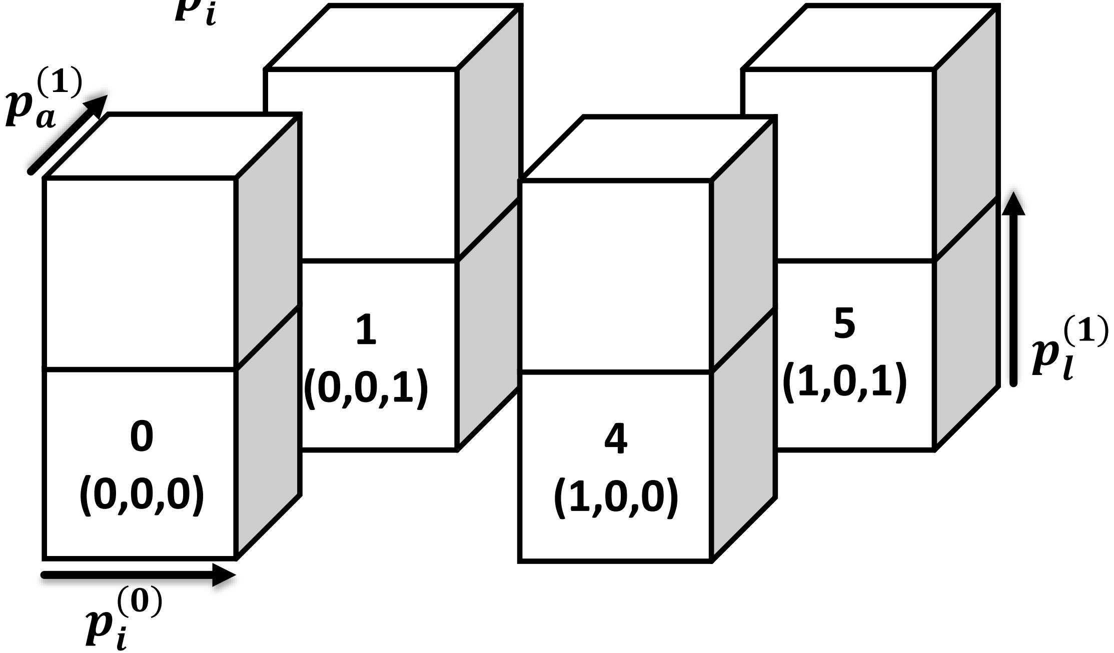


t1[i,a]

$$P = P_i^{(1)} P_l^{(1)} P_a^{(1)}$$



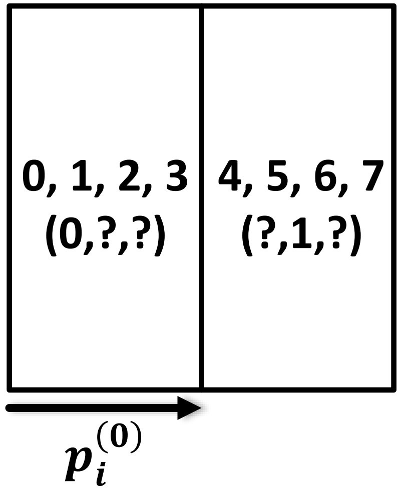




Data Redistribution

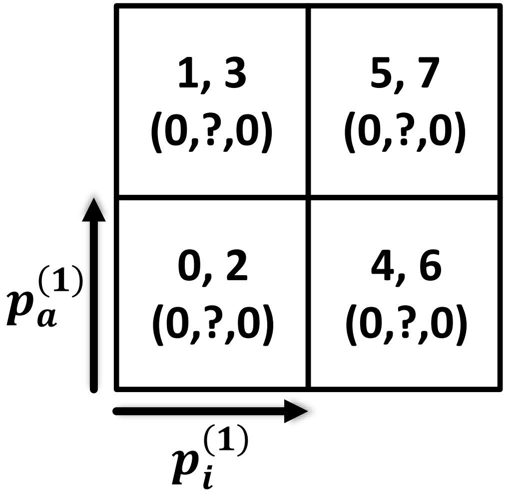
4

$P = P_i^{(0)} P_j^{(0)} P_k^{(0)} P_a^{(0)}$

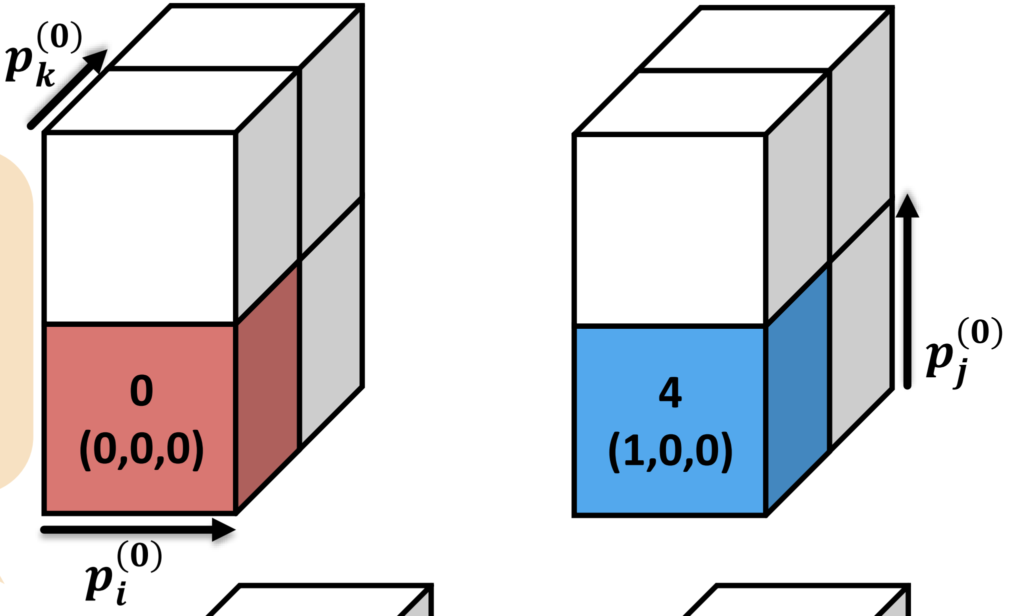


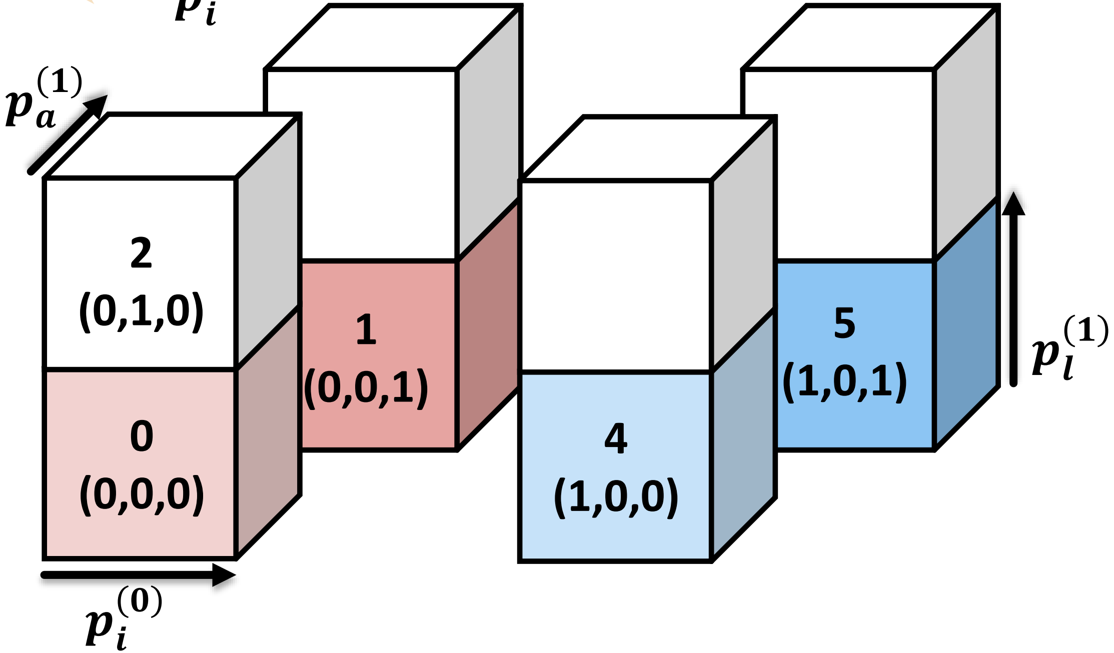
t1[i,a]

$P = P_i^{(1)} P_l^{(1)} P_a^{(1)}$



sub-communicator root processes send/recv blocks

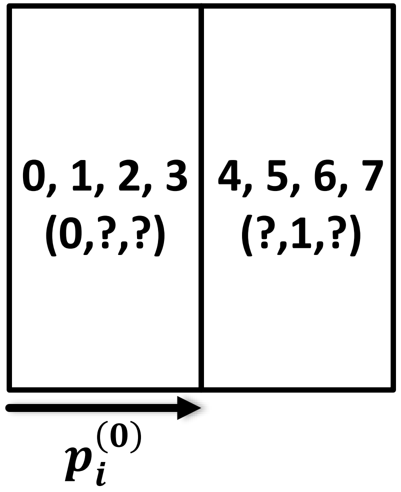




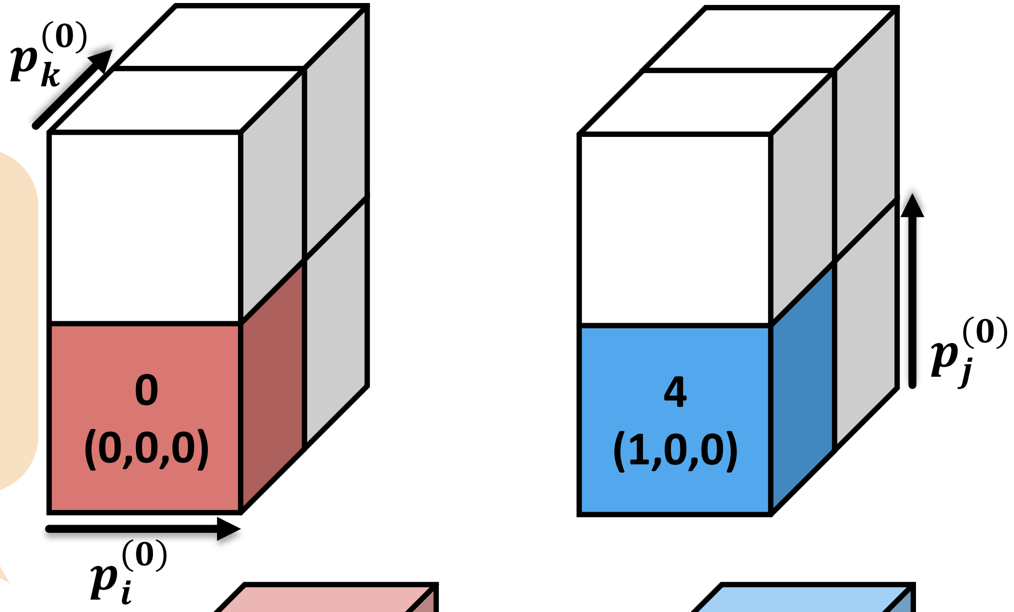
Data Redistribution

4

$P = P_i^{(0)} P_j^{(0)} P_k^{(0)} P_a^{(0)}$

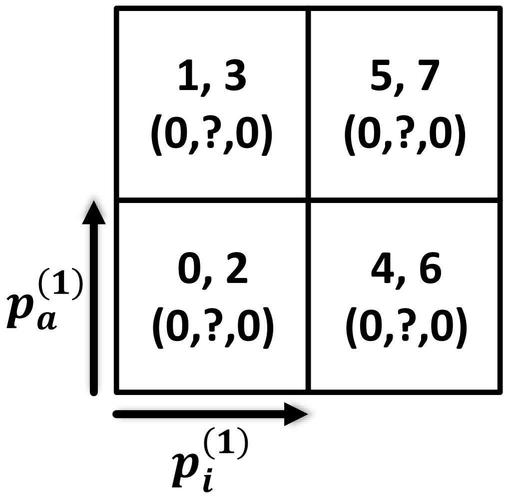


sub-communicator root processes send/recv blocks

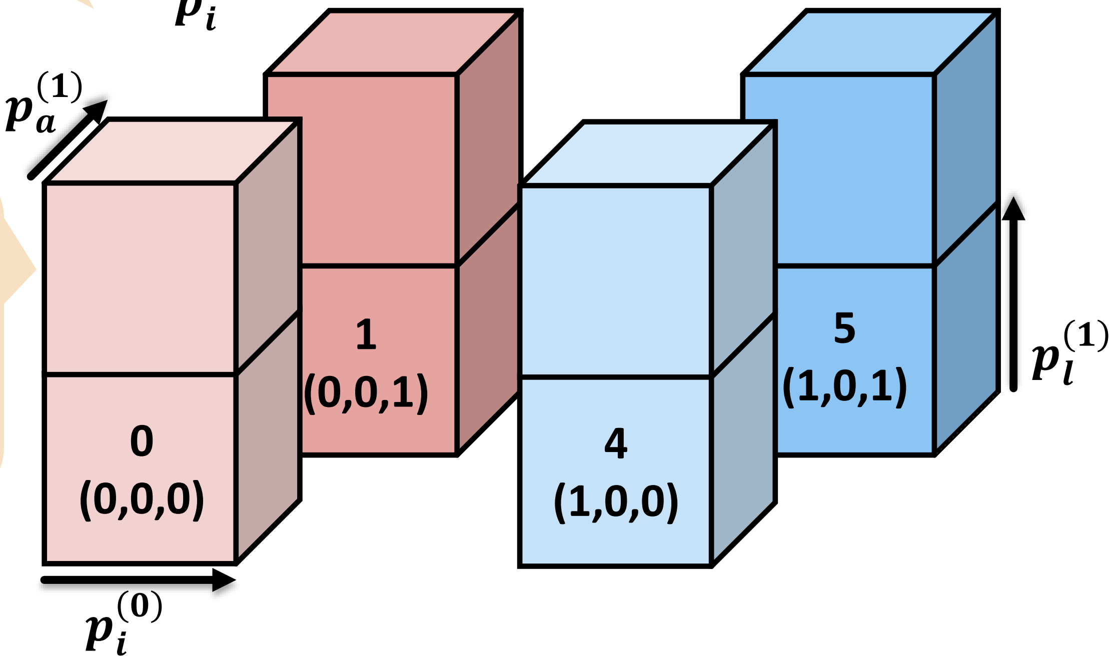


`t1[i,a]`

$P = P_i^{(1)} P_l^{(1)} P_a^{(1)}$



and broadcast to the rest



Automated code generation: from DaCe-Python to C++

5

```
grid0 = mpi.Cart_create(dims=[P0I,P0J,P0K,P0A])
grid0_t1 = mpi.Cart_sub(comm=grid0, remain=[False,True,True,False])
grid1 = mpi.Cart_create(dims=[P1I, P1L, P1A])
grid1_out = mpi.Cart_sub(comm=grid1, remain=[False,False,True])

# ja,ka->jka
t0 = np.zeros((NJ//P0J, NK//P0K, NA//P0A), dtype=X.dtype)
for j in range(NJ//P0J):
  for k in range(NK//P0K):
    for a in range(NA//P0A):
      t0[j, k, a] += A[j, a] * B[k, a]
# ijk,jka->ia
t1 = np.tensordot(X, t0, axes=([1, 2], [0, 1]))
mpi.Allreduce(t1, comm=grid0_t1, op=mpi.SUM)

t2 = deinsum.Redistribute(t1, comm1=grid0, comm2=grid1)

# ia,al->il
out = t2 @ C
mpi.Allreduce(out, comm=grid1_out)
```

MPI comm interface

products without contraction: for-loops

dot products: tensordot

MPI collectives

data redistribution

Automated code generation: from DaCe-Python to C++

5

```
int grid1_remain[4] = {0, 1, 1, 0};
MPI_Cart_sub(grid0_comm, pgrid1_remain, &grid1_comm);
MPI_Comm_rank(grid1_comm, &grid1_rank);
MPI_Cart_coords(grid1_comm, grid_1_rank, 2, grid_1_coords);
```

(CUDA-aware) MPI

```
#pragma omp parallel for
for (auto j = 0; j < S1; j += 1)
  for (auto k = 0; k < S2; k += 1)
    for (auto a = 0; a < S3; a += 1)
      t0[S3*S2*j + S3*k + a] = A[S3*j + a]* B[S3*k + a];
```

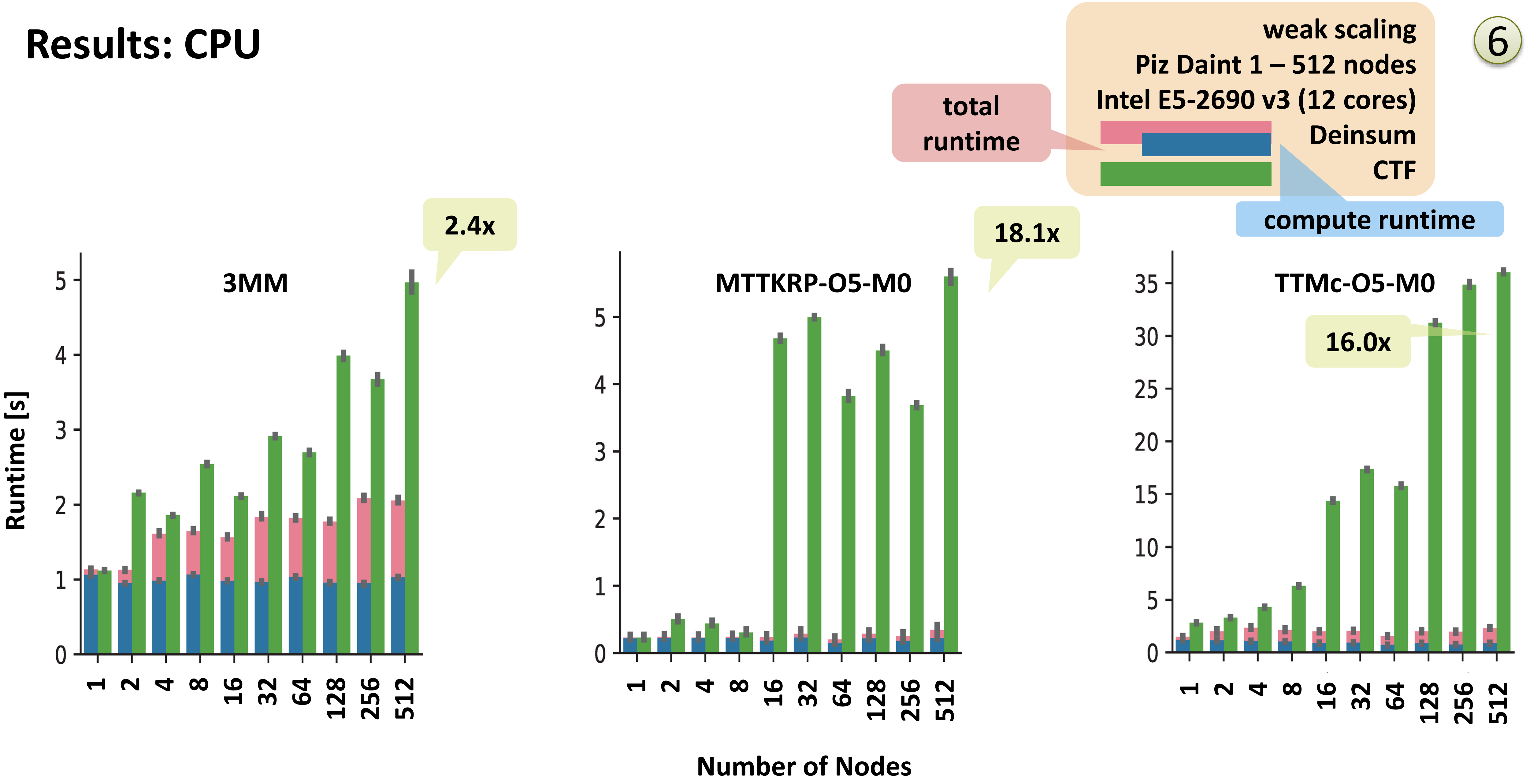
CPU – OpenMP
GPU – CUDA kernels

```
cblas_dgemm(CblasColMajor, CblasNoTrans, CblasNoTrans,
            S3, S0, S1*S2,
            1.0, t0, S3, X, S1*S2,
            0.0, t1, S3);
```

CPU – TTGT
GPU – cuTENSOR

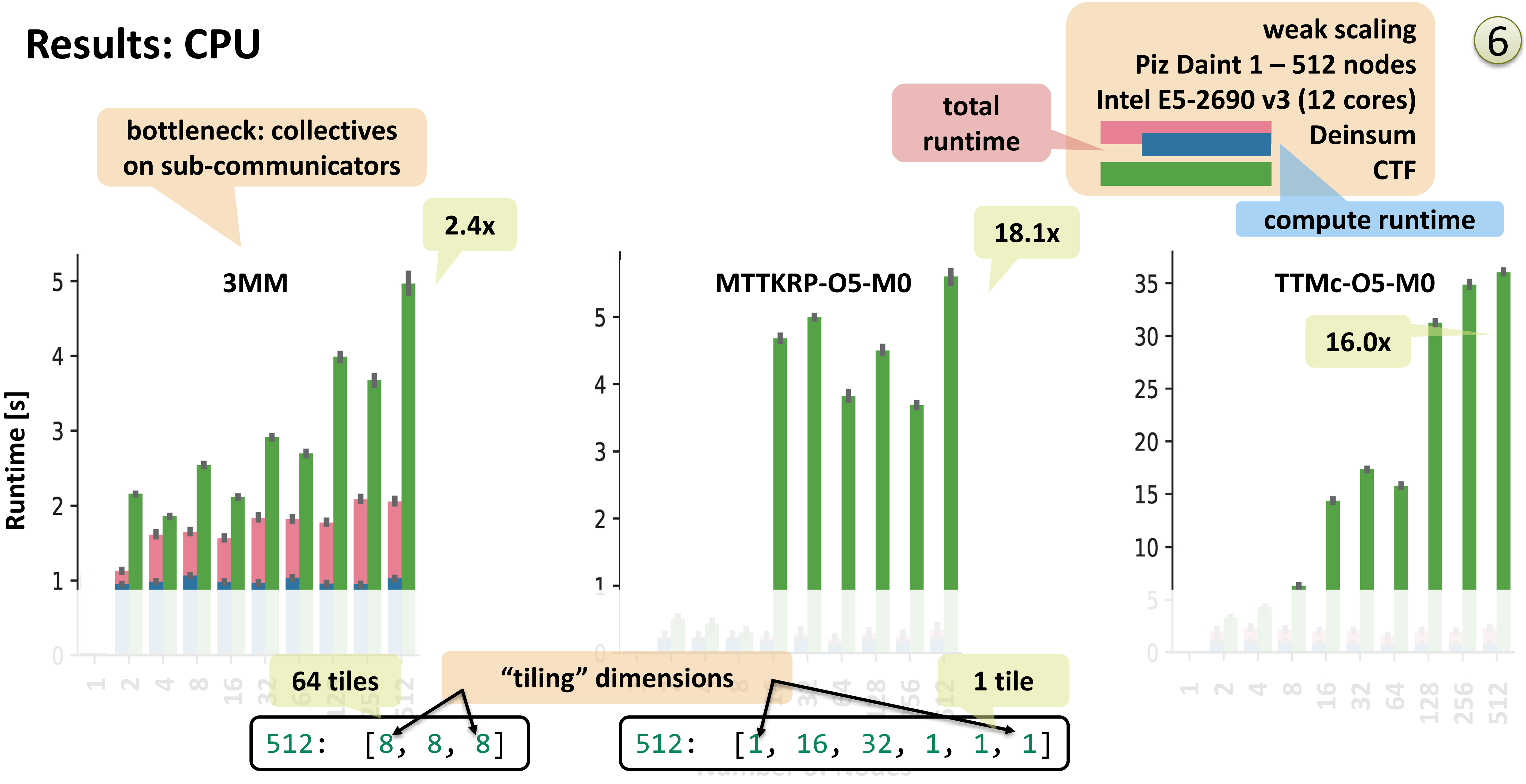
```
MPI_Allreduce(MPI_IN_PLACE, out, S0*S3, MPI_DOUBLE, MPI_SUM,
              grid1_comm);
```

Results: CPU


6
weak scaling
Piz Daint 1 – 512 nodes
Intel E5-2690 v3 (12 cores)
Deinsum
CTF
total runtime
compute runtime
2.4x
18.1x
16.0x
3MM
MTTKRP-O5-M0
TTMc-O5-M0
Runtime [s]
Number of Nodes
0
1
2
3
4
5
10
15
20
25
30
35
1
2
4
8
16
32
64
128
256
512

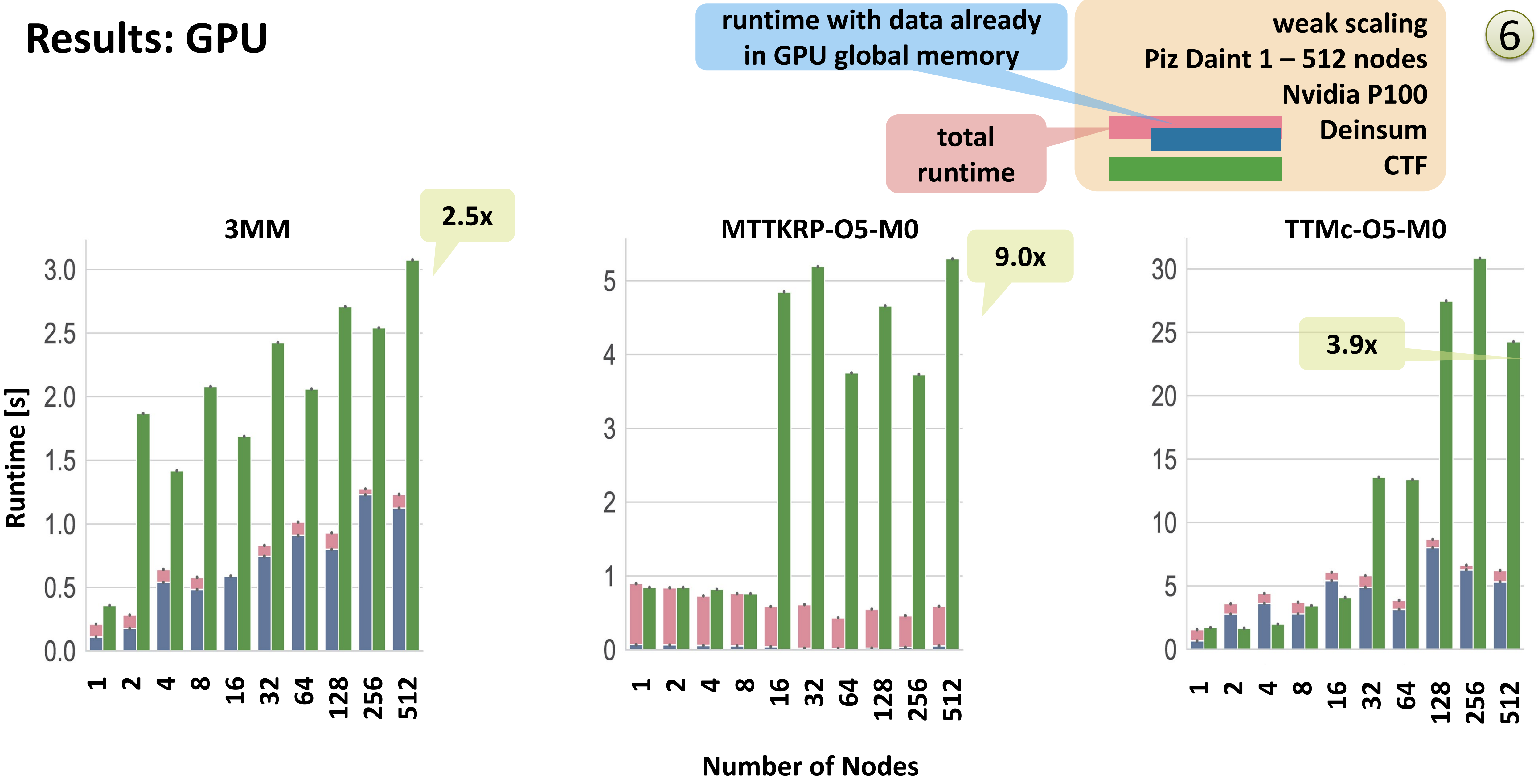
Results: CPU

6



Results: GPU

6



Conclusions

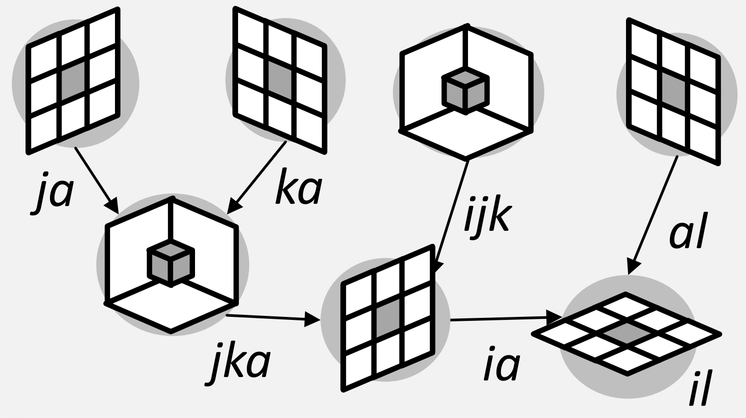
1 Input

$$ijk, ja, ka, al \rightarrow il$$

2 Split to binary operations

$$ja, ka \rightarrow jka$$
$$ijk, jka \rightarrow ia$$
$$ia, al \rightarrow il$$

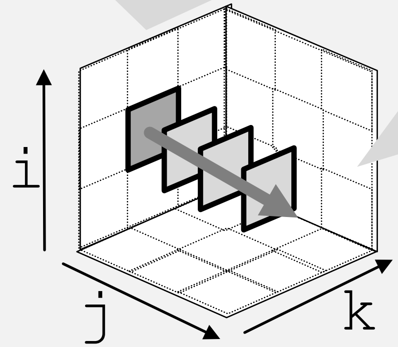
3 Communication-optimal schedules



4 Iteration spaces and distribution

Iteration space partition: MPI_Cart_sub

Distribute initial data: MPI_Broadcast

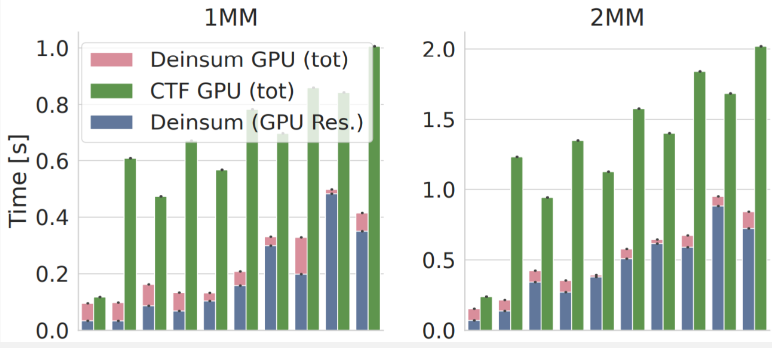


5 Automated code generation

```
for j in range(NJ//P0J):
 for k in range(NK//P0K):
  for a in range(NA//P0A):
   t0[j, k, a] += A[j, a] * B[k, a]

t1 = np.tensordot(X, t0,
        axes=([1, 2], [0, 1])
mpi.Allreduce(t1, comm=grid0_t1)

t2 = deinsum.Redistribute(t1,
         comm1=grid0, comm2=grid1)
```

6 Results

- up to 19x speedup over CTF
- CPU and GPU support



Conclusions

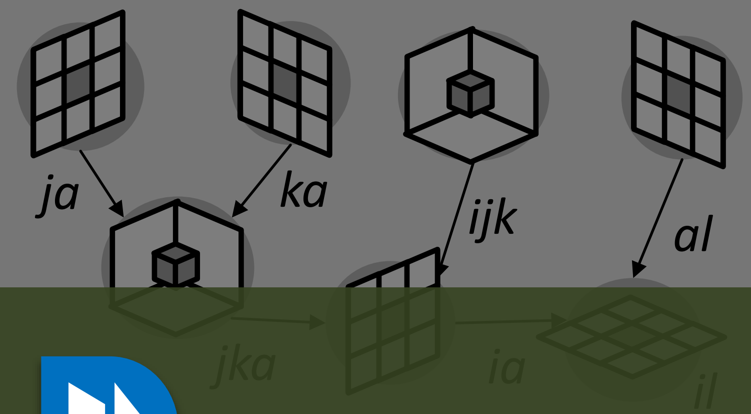
1 **Input**

$$ijk, ja, ka, al \rightarrow il$$

2 **Split to binary operations**

$$ja, ka \rightarrow jka$$
$$ijk, jka \rightarrow ia$$
$$ia, al \rightarrow il$$

3 **Communication-optimal schedules**

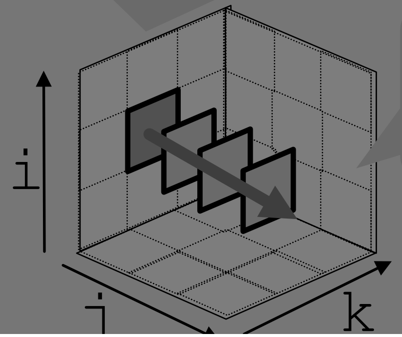


github.com/spcl/dace

4 **Iteration spaces and distribution**

Iteration space partition:
MPI_Cart_sub

Distribute initial data:
MPI_Broadcast



5 **Automated code generation**

```
for j in range(NJ//P0J):
 for k in range(NK//P0K):
  for a in range(NA//P0A):
   t0[j, k, a] += A[j, a] * B[k, a]

t1 = np.tensordot(X, t0,
        axes=([1, 2], [0, 1])
mpi.Allreduce(t1, comm=grid0_t1)

t2 = deinsum.Redistribute(t1,
        comm1=grid0, comm2=grid1)
```

6 **Results**

- up to 19x speedup over CTF
- CPU and GPU support