Performance Engineering for Sparse Linear Solvers

Half-Day Tutorial at SC25

Christie L. Alappat, Erlangen National High Performance Computing Center

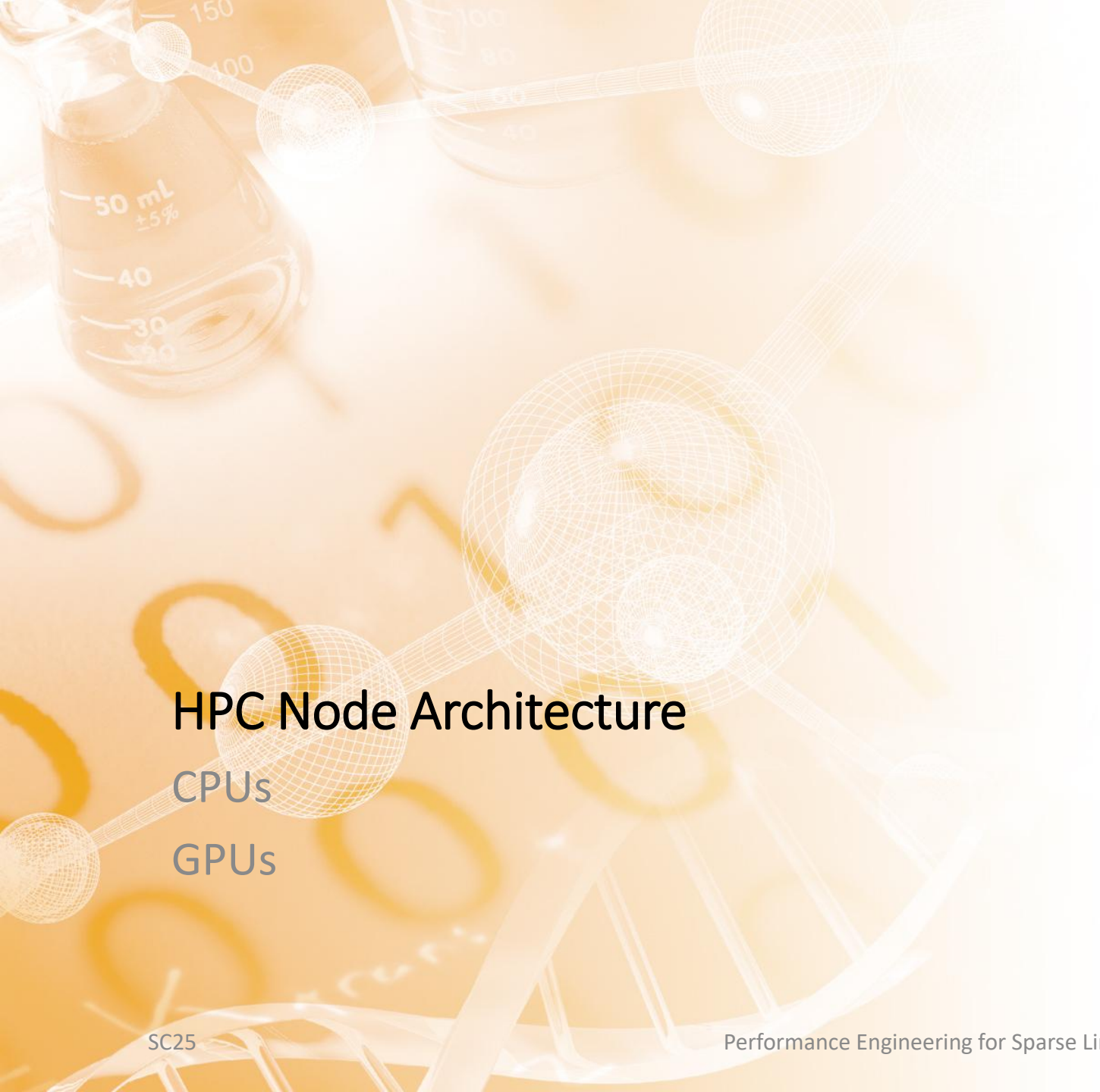
Jonas Thies, TU Delft

Georg Hager, Erlangen National High Performance Computing Center

Hartwig Anzt, TU München

Tutorial Agenda

- Brief introduction to node-level computer architecture
- Performance modeling with the Roofline model
- Sparse matrix-vector multiplication (SpMV) performance, sparse-matrix data formats, and Roofline modeling of SpMV
- The Conjugate Gradient (CG) algorithm
- Preconditioning and preconditioned CG (PCG)
- Accelerating matrix power kernels (MPK) by cache blocking
- Optional: distributed-memory SpMV and MPK cache blocking



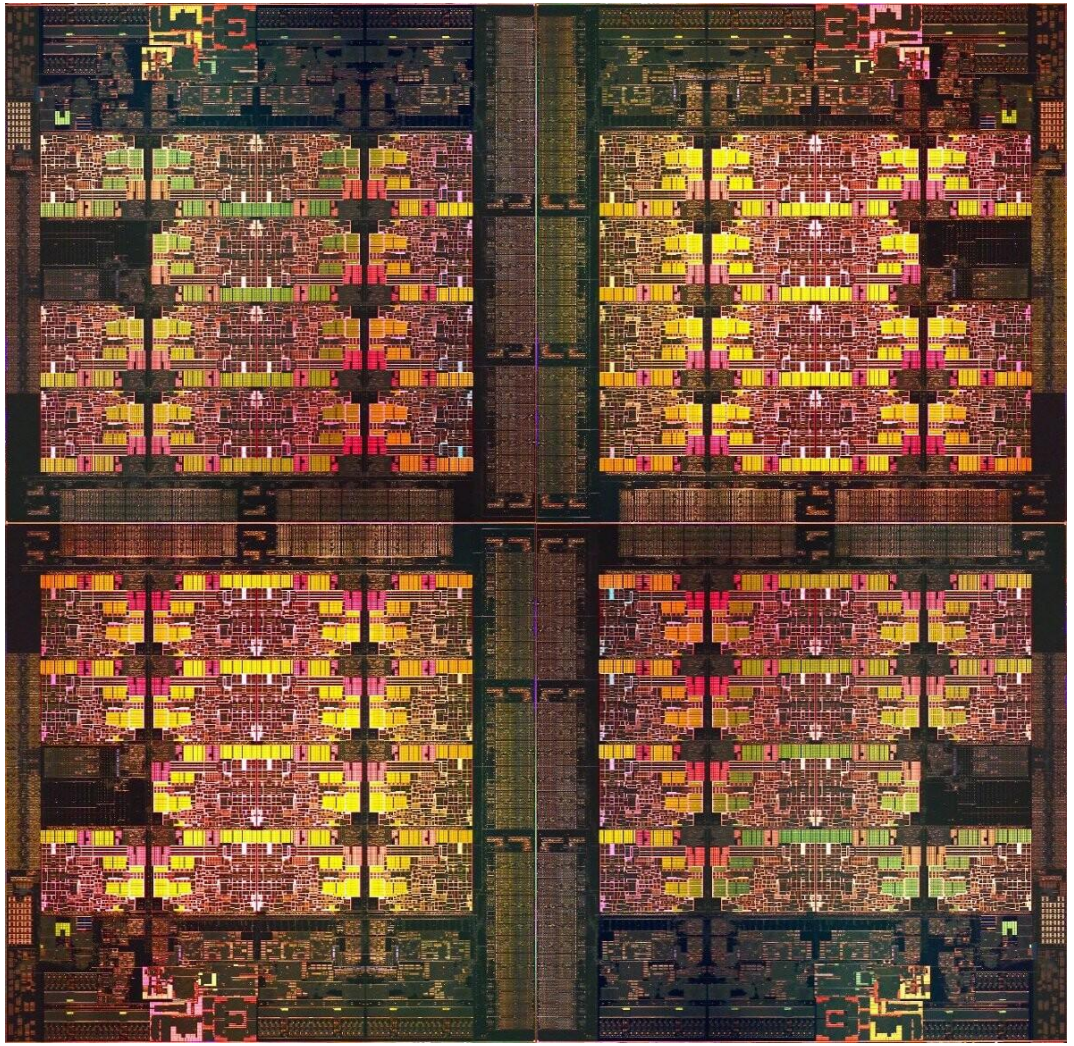
HPC Node Architecture

CPUs

GPUs

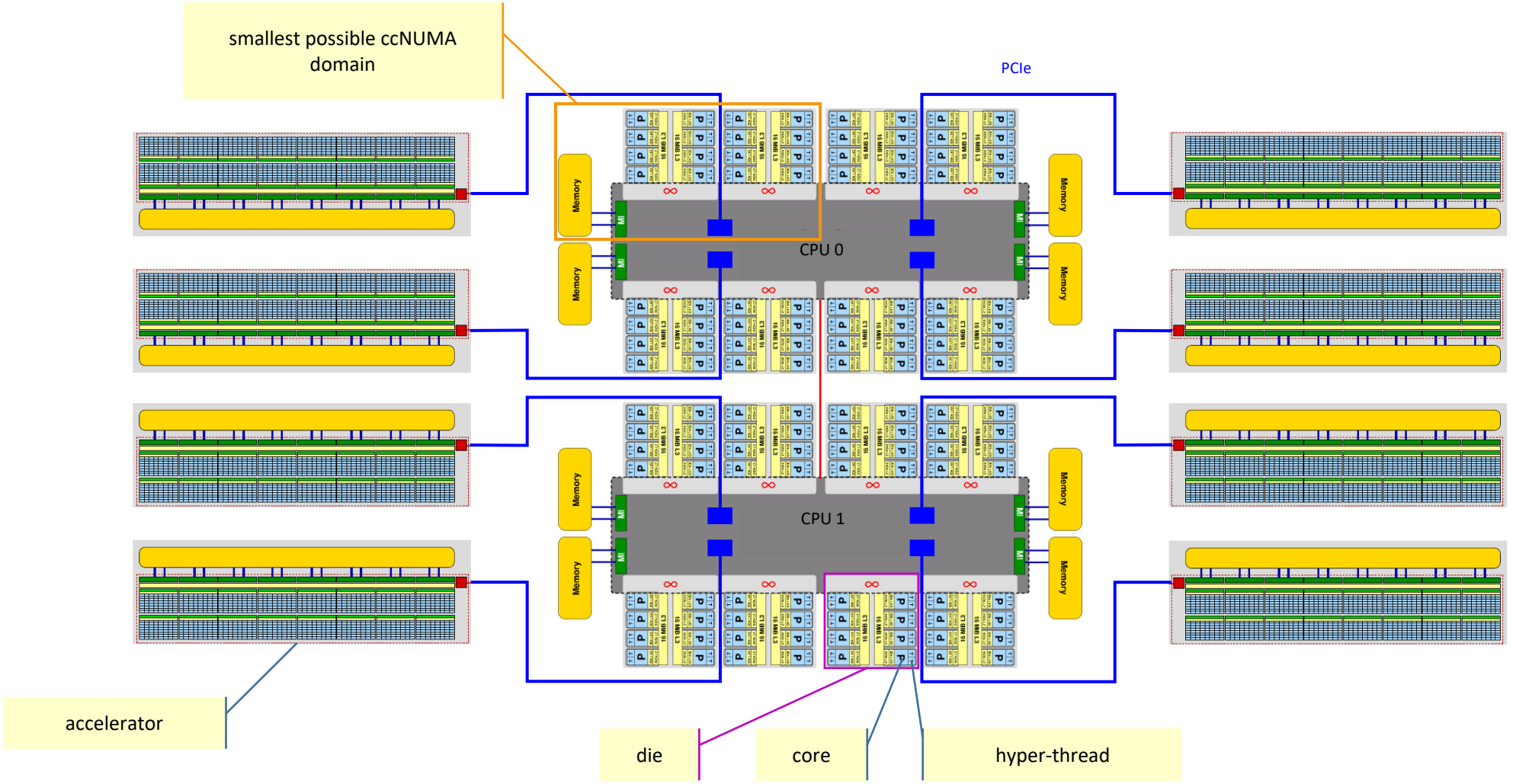
Multi-core today: Intel Xeon Sapphire Rapids (2023)

- Xeon “Sapphire Rapids” (Platinum/Gold/Silver/Bronze):
  Up to 60 cores running at 1.7+ GHz
  (+ “Turbo Mode” 4.8 GHz),
- “Intel 7” process / up to 350 W
- Multi-die package (4 chips)
- Clock frequency:
  flexible ☺



https://www.techpowerup.com/292204/intel-sapphire-rapids-xeon-4-tile-mcm-annotated

A more current example – lots of “topology”!



Nvidia H100 “Hopper” SXM5 specs

Architecture

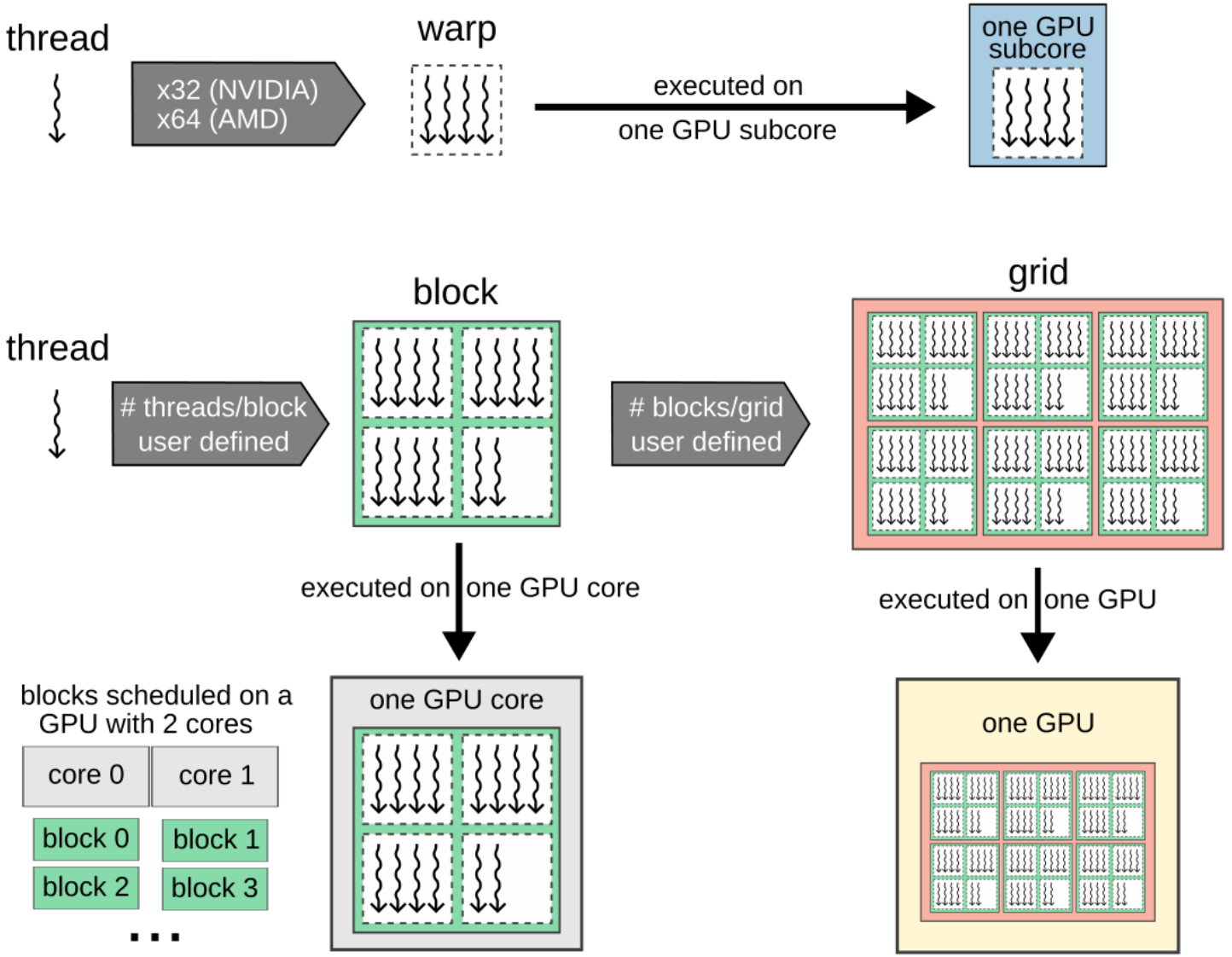
- 80 B Transistors
- ~ 1.8 GHz clock speed
- ~ 144 “SM” units
  - 128 SP “cores” each (FMA)
  - 64 DP “cores” each (FMA)
  - 4 “Tensor Cores” each
  - 2:1 SP:DP performance

- ~ 34 TFlop/s DP peak (FP64)
- 50 MiB L2 Cache

- 80 GB HBM3
- MemBW ~ 3300 GB/s (theoretical)
- MemBW ~ 3000 GB/s (measured)

Threads, warps, grids on NVIDIA GPUs

- Threads are organized in warps (32 threads)
- Warps make up a block
- A block executes on one “core” (SM)
- Scheduling happens automatically
- Developer must map thread IDs to “work items” (iterations)
- Threads in a warp should access consecutive memory addresses



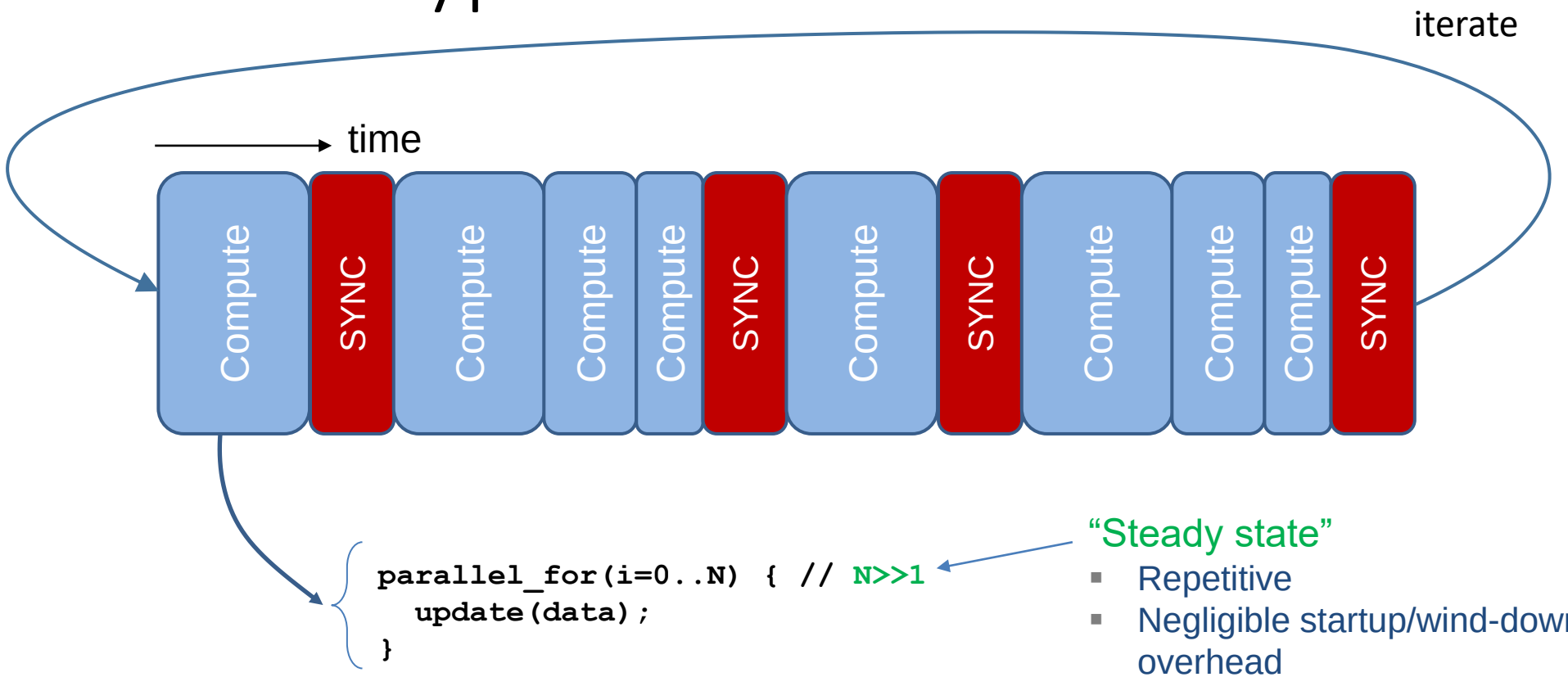
The Roofline Model

Bottleneck-based thinking

Simple models for single loops

Multiple loops

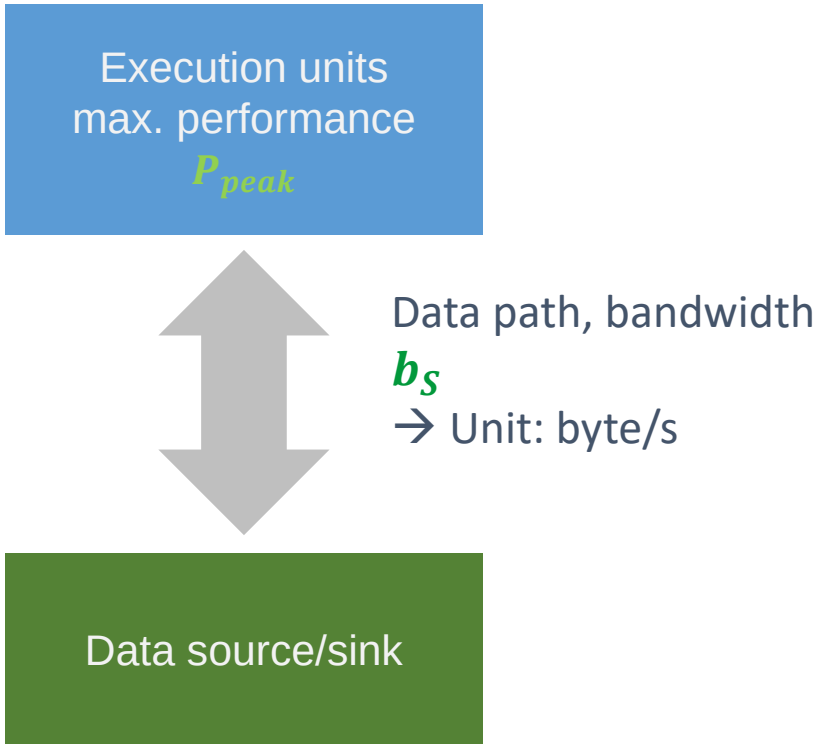
Structure of typical solver code



Runtime model: $T = f(\texttt{\$STUFF}, \texttt{\$HARDWARE})$

A simple two-bottleneck model of loop code execution

Simplistic view of the hardware:



Simplistic view of the software:

```
! may be multiple levels
do i = 1,<sufficient>
  <complicated stuff doing
    N flops causing
    V bytes of data transfer>
enddo
```

Computational intensity $I = \frac{N}{V}$

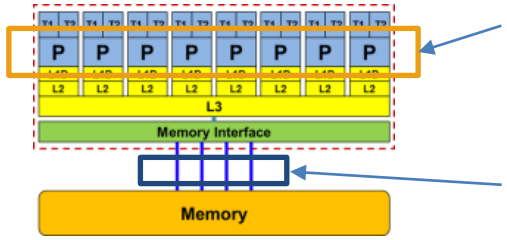
→ Unit: flop/byte

Which takes longer?
- Data transfer
- Work execution

Predicting the (minimum) runtime of a loop

Two bottlenecks:

```
#pragma omp parallel for
for(i=0; i<10^7; ++i)
  a[i] = a[i] + s * c[i];
```



8-core CPU
(3 GHz Intel Sandy Bridge)

$R^{max}_{flops} = 192 \frac{\text{Gflops}}{\text{s}}$

$R^{max}_{BW} = 40 \frac{\text{Gbyte}}{\text{s}}$

Resources rates provided (machine properties)

Resources needed (code properties)

$W_{flops} = 2 \times 10^7 \text{ flops}$

$W_{BW} = 3 \times 8 \times 10^7 \text{ bytes}$

Full-overlap assumption:

$$T_{flops} = \frac{2 \times 10^7 \text{ flops}}{192 \frac{\text{Gflops}}{\text{s}}} = 104\ \mu s$$

$$T_{BW} = \frac{2.4 \times 10^8 \text{ bytes}}{40 \frac{\text{Gbyte}}{\text{s}}} = 6.0 \text{ ms}$$

$$T_{\min} = \max(T_{flops}, T_{BW}) = 6 \text{ ms}$$

From time to performance

$$P_{upper} = \frac{W_{flops}}{\max(T_{flops}, T_{BW})} = \frac{W_{flops}}{\max\left(\frac{W_{flops}}{R_{flops}}, \frac{W_{BW}}{R_{BW}}\right)} =$$

$$\min\left(R_{flops}, R_{BW} \times \frac{W_{flops}}{W_{BW}}\right)$$

Machine model: Peak performance [flop/s]

Machine model: Memory bandwidth [byte/s]

Application model: Computational intensity [flop/byte]

“Roofline”!?

R.W. Hockney and I.J. Curington:
$f_{1/2}$: A parameter to characterize memory and communication bottlenecks.
Parallel Computing 10, 277-286 (1989). DOI: 10.1016/0167-8191(89)90100-2

S. Williams: Auto-tuning Performance on Multicore Computers.
UCB Technical Report No. UCB/EECS-2008-164. PhD thesis (2008)

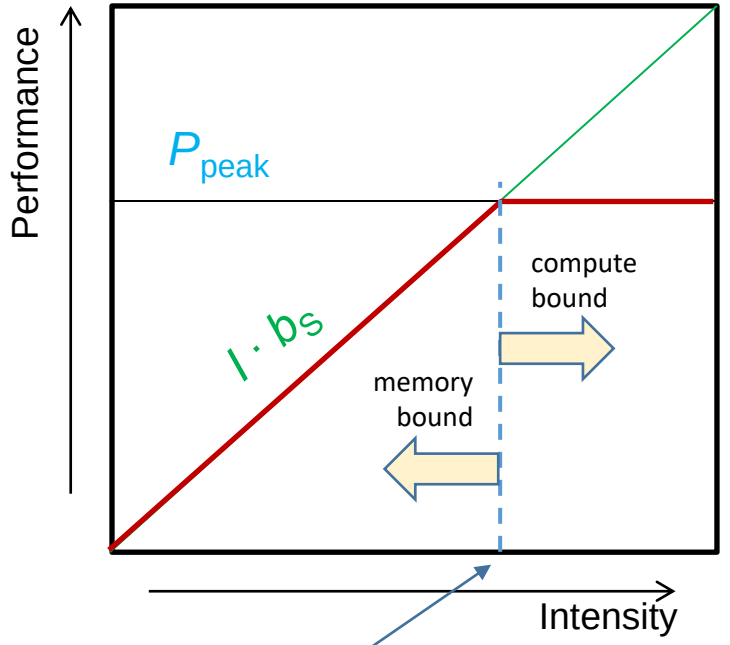
Common nomenclature:

R_{flops} → P_{peak} peak performance

R_{BW} → b_S memory bandwidth

$\frac{W_{flops}}{W_{BW}}$ → I computational intensity

$$P_{upper} = \min\left(P_{peak}, I \times b_S\right)$$

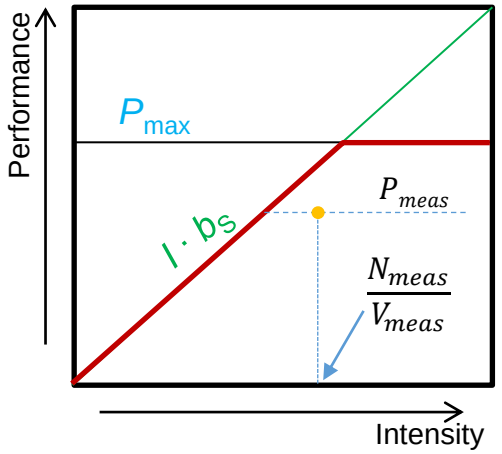


Threshold:
≈ 10-15 F/B for current Server CPUs/GPUs

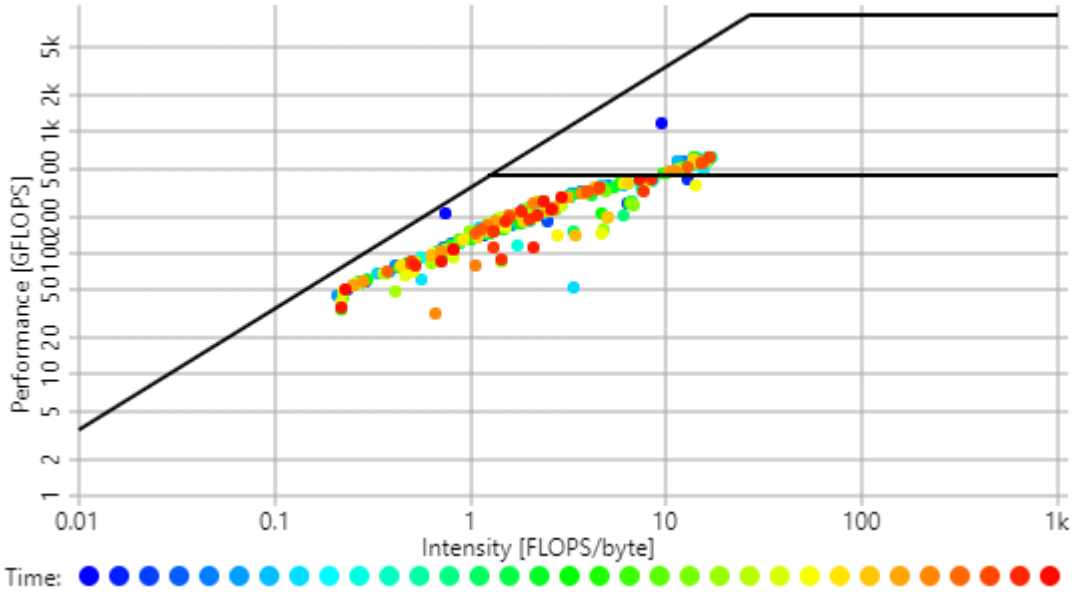
Hands-On:

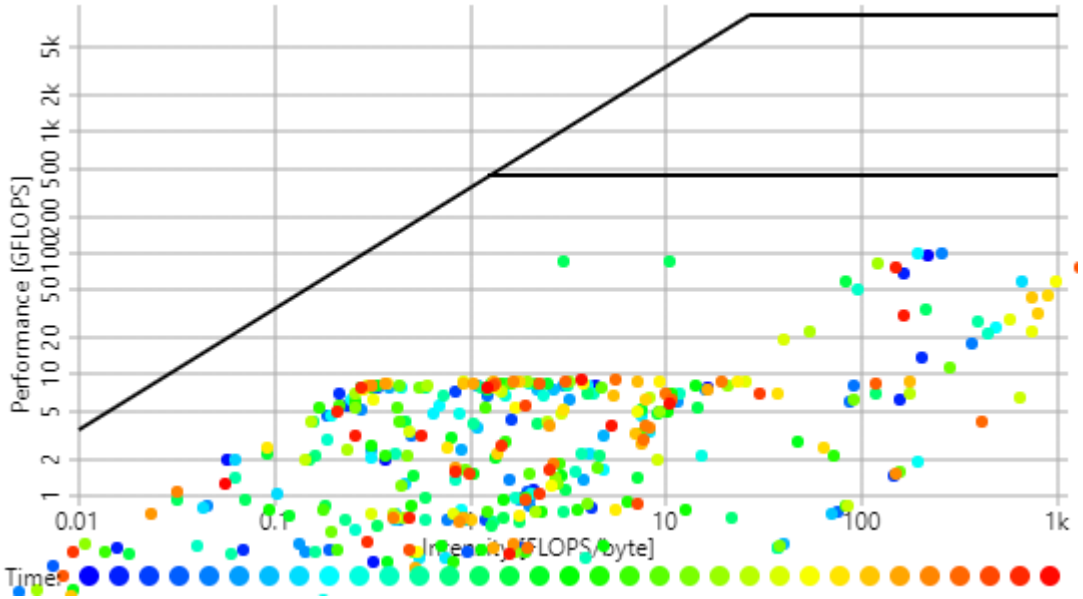
Exploring node topology and bandwidth

Diagnostic modeling



Two cluster jobs…





Roofline: Simple Examples

Dense linear algebra

Sparse linear algebra

Simple solvers: CG

Dense linear algebra

```
for(i=0; i<N; ++i)
    a[i] = a[i]+s*x[i];
```

daxpy (BLAS-1)

```
for(i=0; i<N; ++i)
  s += a[i]*b[i];
```

dot product (BLAS-1)

```
for(k=0; k<NK;++k)
  for(l=0; l<NL; ++l)
    for(m=0; m<NM; ++m)
      y[k*NL+l] +=
           A[k*NM+m]*B[l*NM+m];
```

dense MMM (BLAS-3)

```
for(r=0; r<NR; ++r)
  for(c=0; c<NC; ++c)
    y[r] += A[r*NC+c]*x[c];
```

dense MVM (BLAS-2)
dot-product style

Roofline thinking:

What is the computational intensity?

Dense MVM

```
for(r=0; r<NR; ++r)
  for(c=0; c<NC; ++c)
    y[r] += A[r*NC+c]*x[c];
```

- One DP read from memory for each matrix entry
- x[ ] and y[ ] are read and updated from cache after 1st read
- → 8 byte and 2 flops per iteration

Computational intensity $I = \frac{2\ flop}{8\ byte} = 0.25\ \frac{flop}{byte}$

Dense MMM?

```
for(k=0; k<NK;++k)
  for(l=0; l<NL; ++l)
    for(m=0; m<NM; ++m)
      y[k*NL+l] +=
            A[k*NM+m]*B[l*NM+m];
```

- Blocking/unrolling techniques can increase intensity beyond the Roofline knee

```
for(k=0; k<NK; k+=2)
  for(l=0; l<NL; l+=2)
    for(m=0; m<NM; ++m)
      y[k*NL+l]           += A[k*NM+m]*B[l*NM+m];
      y[(k+1)*NL+l]       += A[(k+1)*NM+m]*B[l*NM+m];
      y[k*NL+(l+1)]       += A[k*NM+m]*B[(l+1)*NM+m];
      y[(k+1)*NL+(l+1)] += A[(k+1)*NM+m]*B[(l+1)*NM+m];
```

→ peak performance achievable

Sparse Matrices and SpMV

Sparse Matrix Formats

Sparse Matrix Vector Product Parallelization

Matrix Vector Multiplication

Input A, x, y Output $y = A \cdot x$

- Central building block in many complex algorithms:
  - Orthogonalization, power iteration in Page Rank, Power flow of a system …
- Before we turn to sparse matrices, we recall how we store & handle dense matrices on parallel processors (i.e. GPUs)

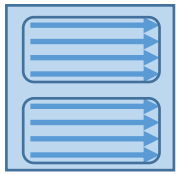
Matrix Vector Multiplication

Input A, x, y Output $y = A \cdot x$

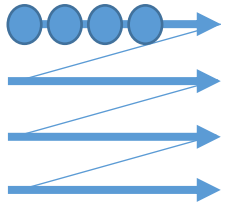
```
__global__ void sgemv_rowmajor( …)
{
   int row = blockIdx.x*blockDim.x + threadIdx.x;
   float sum = 0.0;
   if (row < n){
      for( int col=0; col<n; col++){
         sum += m[ row*n + col ] * x[ col ];
      }
      y[ row ] = alpha * sum;
   }
}
```

```
__global__ void sgemv_colmajor( …)
{
   int row = blockIdx.x*blockDim.x + threadIdx.x;
   float sum = 0.0;
   if (row < n){
      for( int col=0; col<n; col++){
         sum += m[ row + n*col ] * x[ col ];
      }
      y[ row ] = alpha * sum;
   }
}
```

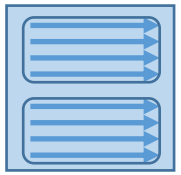
Parallel threads

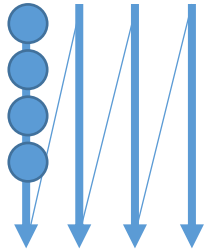


Row-major



Col-major





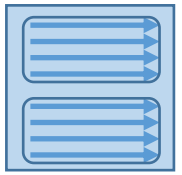
Matrix Vector Multiplication

Input A, x, y Output $y = A \cdot x$

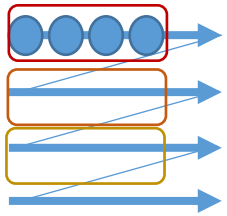
```
__global__ void sgemv_rowmajor( …)
{
   int row = blockIdx.x*blockDim.x + threadIdx.x;
   float sum = 0.0;
   if (row < n){
      for( int col=0; col<n; col++){
         sum += m[ row*n + col ] * x[ col ];
      }
      y[ row ] = alpha * sum;
   }
}
```

```
__global__ void sgemv_colmajor( …)
{
   int row = blockIdx.x*blockDim.x + threadIdx.x;
   float sum = 0.0;
   if (row < n){
      for( int col=0; col<n; col++){
         sum += m[ row + n*col ] * x[ col ];
      }
      y[ row ] = alpha * sum;
   }
}
```

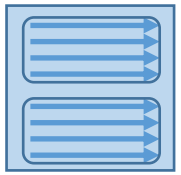
Parallel threads

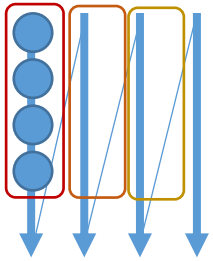


Row-major



Col-major





First read

Second read

Third read

…

Matrix Vector Multiplication

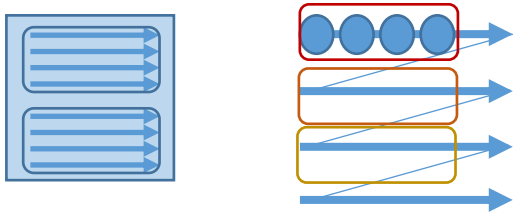
Input A, x, y Output $y = A \cdot x$

```
__global__ void sgemv_rowmajor( …)
{
   int row = blockIdx.x*blockDim.x + threadIdx.x;
   float sum = 0.0;
   if (row < n){
      for( int col=0; col<n; col++){
         sum += m[ row*n + col ] * x[ col ];
      }
      y[ row ] = alpha * sum;
   }
}
```

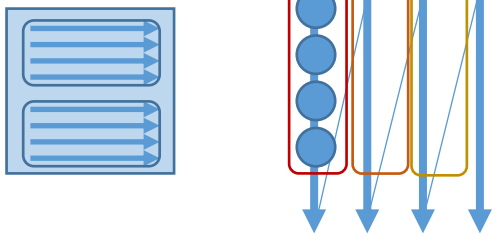
```
__global__ void sgemv_colmajor( …)
{
   int row = blockIdx.x*blockDim.x + threadIdx.x;
   float sum = 0.0;
   if (row < n){
      for( int col=0; col<n; col++){
         sum += m[ row + n*col ] * x[ col ];
      }
      y[ row ] = alpha * sum;
   }
}
```

Parallel threads

Row-major



Col-major

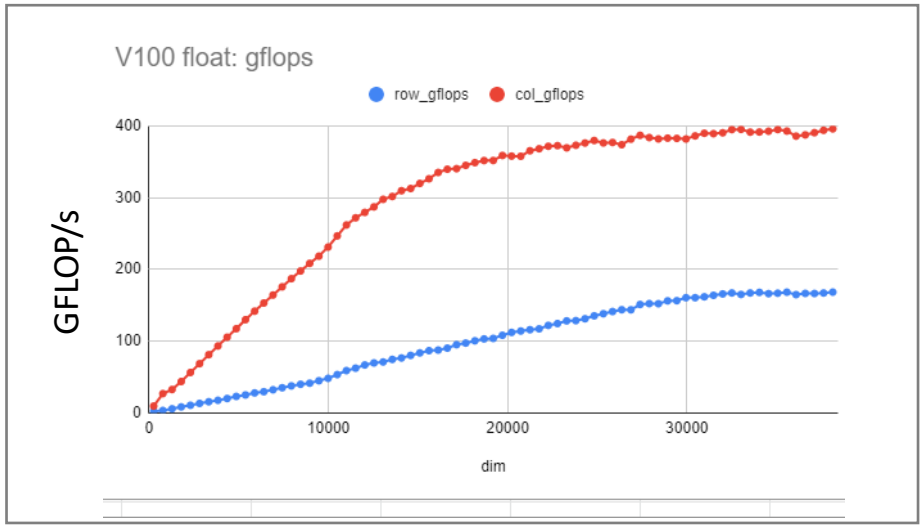


First read

Second read

Third read

…



Sparse Matrix Vector Multiplication

Input A, x, y Output $y = A \cdot x$

- Matrix A contains only few nonzero elements.
- Storing all entries results in large overhead (memory & computation).

Sparse Matrix Vector Multiplication

Input A, x, y Output $y = A \cdot x$

- Matrix A contains only few nonzero elements.
- Storing all entries results in large overhead (memory & computation).
- Idea: Store only nonzero elements [nz] explicitly.

$$A = \begin{pmatrix} 5.4 & 1.1 & 0 & 0 & 0 & 0 \\ 2.2 & 8.3 & 0 & 3.7 & 1.3 & 3.8 \\ 0 & 0 & 4.2 & 0 & 0 & 0 \\ 5.4 & 0 & 0 & 9.2 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1.1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 8.1 \end{pmatrix}$$

$$\text{value} = [\quad 5.4 \quad 1.1 \quad 2.2 \quad 8.3 \quad 3.7 \quad 1.3 \quad 3.8 \quad 4.2 \quad 5.4 \quad 9.2 \quad 1.1 \quad 8.1 \quad]$$ Value

COO format

Input A, x, y Output $y = A \cdot x$

- Matrix A contains only few nonzero elements.
- Storing all entries results in large overhead (memory & computation).
- Idea: Store only nonzero elements [nz] explicitly.

Need to also store location of nonzero elements!

$$A = \begin{pmatrix} 5.4 & 1.1 & 0 & 0 & 0 & 0 \\ 2.2 & 8.3 & 0 & 3.7 & 1.3 & 3.8 \\ 0 & 0 & 4.2 & 0 & 0 & 0 \\ 5.4 & 0 & 0 & 9.2 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1.1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 8.1 \end{pmatrix}$$

Memory footprint of COO format:
nz(val) + 2*nz(int)

$$\text{value} = [\ 5.4\ \ 1.1\ \ 2.2\ \ 8.3\ \ 3.7\ \ 1.3\ \ 3.8\ \ 4.2\ \ 5.4\ \ 9.2\ \ 1.1\ \ 8.1\ ]$$ Value

$$\text{colidx} = [\ 0\ \ 1\ \ 0\ \ 1\ \ 3\ \ 4\ \ 5\ \ 2\ \ 0\ \ 3\ \ 4\ \ 5\ ]$$ Column-index

$$\text{rowidx} = [\ 0\ \ 0\ \ 1\ \ 1\ \ 1\ \ 1\ \ 1\ \ 2\ \ 3\ \ 3\ \ 4\ \ 5\ ]$$ Row-index

COO format

Input A, x, y Output $y = A \cdot x$

- Matrix A cont
- Storing all ent
- Idea: Store or

Need t

format:

Hands-on Exercise: Convert this matrix into COO format:

$$\begin{pmatrix} 0 & 0 & 0 & 4 & 2 & 0 \\ 0 & 2 & 3 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 2 & 3 & 4 & 3 & 1 \\ 0 & 2 & 0 & 0 & 0 & 1 \\ 1 & 2 & 3 & 0 & 0 & 1 \end{pmatrix}$$

$\text{value} = [\quad 5$

$\text{colidx} = [\quad 0$

Compute the memory requirement (# vals + # int)

$\text{rowidx} = [\quad 0 \quad 0 \quad 1 \quad 1 \quad 1 \quad 1 \quad 1 \quad 2 \quad 3 \quad 3 \quad 4 \quad 5 \quad]$ Row-index

COO format

Input A, x, y Output $y = A \cdot x$

- Matrix A cont
- Storing all ent
- Idea: Store or

Need t

format:

Hands-on Exercise: Convert this matrix into COO format:

$$\begin{pmatrix} 0 & 0 & 0 & 4 & 2 & 0 \\ 0 & 2 & 3 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 2 & 3 & 4 & 3 & 1 \\ 0 & 2 & 0 & 0 & 0 & 1 \\ 1 & 2 & 3 & 0 & 0 & 1 \end{pmatrix}$$

Compute the memory requirement (# vals + # int)

17 vals + 34 int

$\text{value} = [\ 5$

$\text{colidx} = [\ 0$

$\text{rowidx} = [\ 0\ \ 0\ \ 1\ \ 1\ \ 1\ \ 1\ \ 1\ \ 2\ \ 3\ \ 3\ \ 4\ \ 5\ ]$ Row-index

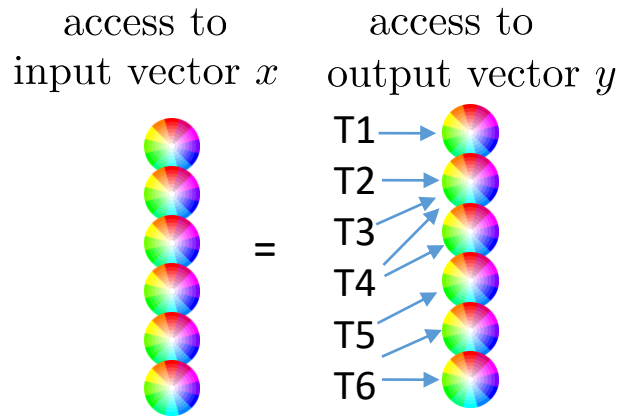
COO SpMV

Input A, x, y Output $y = A \cdot x$

Split nonzero elements into chunks and parallelize across chunks.

- Partial sums need synchronization / atomics to avoid write conflicts.
- Non-coalesced memory access (because row-major).

$$A = \begin{pmatrix} 5.4 & 1.1 & 0 & 0 & 0 & 0 \\ 2.2 & 8.3 & 0 & 3.7 & 1.3 & 3.8 \\ 0 & 0 & 4.2 & 0 & 0 & 0 \\ 5.4 & 0 & 0 & 9.2 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1.1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 8.1 \end{pmatrix}$$



$$\text{value} = [\ 5.4\ \ 1.1\ \ 2.2\ \ 8.3\ \ 3.7\ \ 1.3\ \ 3.8\ \ 4.2\ \ 5.4\ \ 9.2\ \ 1.1\ \ 8.1\ ]$$

Value

$$\text{colidx} = [\ 0\ \ 1\ \ 0\ \ 1\ \ 3\ \ 4\ \ 5\ \ 2\ \ 0\ \ 3\ \ 4\ \ 5\ ]$$

Column-index

$$\text{rowidx} = [\ 0\ \ 0\ \ 1\ \ 1\ \ 1\ \ 1\ \ 1\ \ 2\ \ 3\ \ 3\ \ 4\ \ 5\ ]$$

Row-index

CSR (==CRS) format

Input A, x, y Output $y = A \cdot x$

- Matrix A contains only few nonzero elements.
- Storing all entries results in large overhead (memory & computation).
- Idea: Store only nonzero elements [nz] explicitly.

Need to also store location of nonzero elements!

$$A = \begin{pmatrix} 5.4 & 1.1 & 0 & 0 & 0 & 0 \\ 2.2 & 8.3 & 0 & 3.7 & 1.3 & 3.8 \\ 0 & 0 & 4.2 & 0 & 0 & 0 \\ 5.4 & 0 & 0 & 9.2 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1.1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 8.1 \end{pmatrix}$$

Memory footprint of COO format:
nz(val) + 2*nz(int)

$$\text{value} = [\ 5.4\ \ 1.1\ \ 2.2\ \ 8.3\ \ 3.7\ \ 1.3\ \ 3.8\ \ 4.2\ \ 5.4\ \ 9.2\ \ 1.1\ \ 8.1\ ]$$

Value

$$\text{colidx} = [\ 0\ \ 1\ \ 0\ \ 1\ \ 3\ \ 4\ \ 5\ \ 2\ \ 0\ \ 3\ \ 4\ \ 5\ ]$$

Column-index

CSR format

Input A, x, y Output $y = A \cdot x$

- Matrix A contains only few nonzero elements.
- Storing all entries results in large overhead (memory & computation).
- Idea: Store only nonzero elements [nz] explicitly.

Need to also store location of nonzero elements!

$$A = \begin{pmatrix} 5.4 & 1.1 & 0 & 0 & 0 & 0 \\ 2.2 & 8.3 & 0 & 3.7 & 1.3 & 3.8 \\ 0 & 0 & 4.2 & 0 & 0 & 0 \\ 5.4 & 0 & 0 & 9.2 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1.1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 8.1 \end{pmatrix}$$

Memory footprint of COO format:
nz(val) + 2*nz(int)

Memory footprint of CSR format:
nz(val) + nz(int) + (n+1) (int)

$$\text{value} = [\ 5.4\ \ 1.1\ \ 2.2\ \ 8.3\ \ 3.7\ \ 1.3\ \ 3.8\ \ 4.2\ \ 5.4\ \ 9.2\ \ 1.1\ \ 8.1\ ]$$

Value

$$\text{colidx} = [\ 0\ \ 1\ \ 0\ \ 1\ \ 3\ \ 4\ \ 5\ \ 2\ \ 0\ \ 3\ \ 4\ \ 5\ ]$$

Column-index

$$\text{rowptr} = [\ 0\ \ 2\ \ 7\ \ 8\ \ 10\ \ 11\ \ 12\ ]$$

Points to the first element in each row

Number of nonzero elements

CSR format

Input A, x, y Output $y = A \cdot x$

- Matrix A contains only few nonzero elements.
- Storing all entries results in large overhead (memory & computation).
- Idea: Store only nonzero elements [nz] explicitly.

Need to also store location of nonzero elements!

$$A = \begin{pmatrix} 5.4 & 1.1 & 0 & 0 & 0 & 0 \\ 2.2 & 8.3 & 0 & 3.7 & 1.3 & 3.8 \\ 0 & 0 & 4.2 & 0 & 0 & 0 \\ 5.4 & 0 & 0 & 9.2 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1.1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 8.1 \end{pmatrix}$$

Memory footprint of COO format:
nz(val) + 2*nz(int)

Memory footprint of CSR format:
nz(val) + nz(int) + (n+1) (int)

$$\text{value} = [\ 5.4\ \ 1.1\ \ 2.2\ \ 8.3\ \ 3.7\ \ 1.3\ \ 3.8\ \ 4.2\ \ 5.4\ \ 9.2\ \ 1.1\ \ 8.1\ ]$$

Value

$$\text{colidx} = [\ 0\ \ 1\ \ 0\ \ 1\ \ 3\ \ 4\ \ 5\ \ 2\ \ 0\ \ 3\ \ 4\ \ 5\ ]$$

Column-index

$$\text{rowptr} = [\ 0\ \ 2\ \ 7\ \ 8\ \ 10\ \ 11\ \ 12\ ]$$

Points to the first element in each row

Number of nonzero elements

CSR format

Input A, x, y Output $y = A \cdot x$

- Matrix A cont
- Storing all ent
- Idea: Store or

Need t

format:

ormat:
t)

Hands-on Exercise: Convert this matrix into CSR format:

$$\begin{pmatrix} 0 & 0 & 0 & 4 & 2 & 0 \\ 0 & 2 & 3 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 2 & 3 & 4 & 3 & 1 \\ 0 & 2 & 0 & 0 & 0 & 1 \\ 1 & 2 & 3 & 0 & 0 & 1 \end{pmatrix}$$

Compute the memory requirement (# vals + # int)

$\text{value} = [\quad 5$

$\text{colidx} = [\quad 0$

$\text{rowptr} = [\quad 0 \quad 2 \quad 7 \quad 8 \quad 10 \quad 11 \quad 12 \quad]$ Points to the first element in each row

Number of nonzero elements

CSR format

Input A, x, y Output $y = A \cdot x$

- Matrix A cont
- Storing all ent
- Idea: Store or

Need t

format:

ormat:
t)

Hands-on Exercise: Convert this matrix into CSR format:

$$\begin{pmatrix} 0 & 0 & 0 & 4 & 2 & 0 \\ 0 & 2 & 3 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 2 & 3 & 4 & 3 & 1 \\ 0 & 2 & 0 & 0 & 0 & 1 \\ 1 & 2 & 3 & 0 & 0 & 1 \end{pmatrix}$$

$\text{value} = [\quad 5$

$\text{colidx} = [\quad 0$

Compute the memory requirement (# vals + # int)

17 vals + 24 int

$\text{rowptr} = [\quad 0 \quad 2 \quad 7 \quad 8 \quad 10 \quad 11 \quad 12 \quad]$ Points to the first element in each row

Number of nonzero elements

CSR SpMV

Input A, x, y Output $y = A \cdot x$

How to parallelize this?

$$A = \begin{pmatrix} 5.4 & 1.1 & 0 & 0 & 0 & 0 \\ 2.2 & 8.3 & 0 & 3.7 & 1.3 & 3.8 \\ 0 & 0 & 4.2 & 0 & 0 & 0 \\ 5.4 & 0 & 0 & 9.2 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1.1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 8.1 \end{pmatrix}$$

$$\text{value} = [\ 5.4\ \ 1.1\ \ 2.2\ \ 8.3\ \ 3.7\ \ 1.3\ \ 3.8\ \ 4.2\ \ 5.4\ \ 9.2\ \ 1.1\ \ 8.1\ ]$$ Value

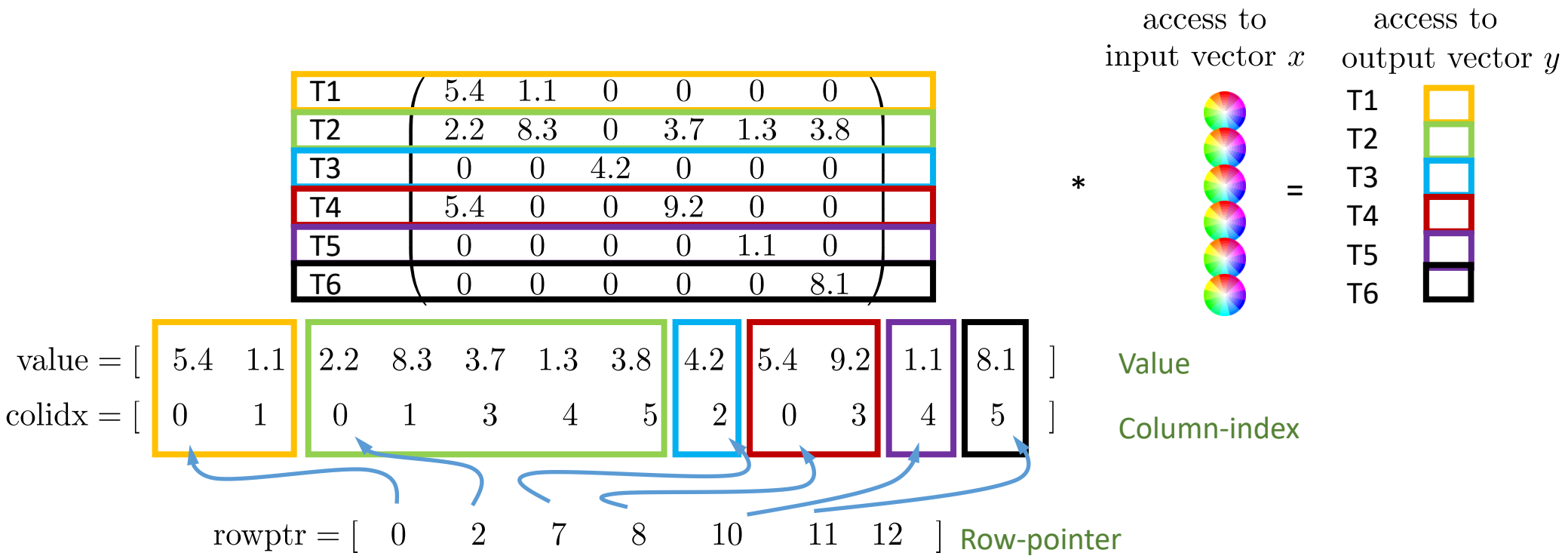
$$\text{colidx} = [\ 0\ \ 1\ \ 0\ \ 1\ \ 3\ \ 4\ \ 5\ \ 2\ \ 0\ \ 3\ \ 4\ \ 5\ ]$$ Column-index

$$\text{rowptr} = [\ 0\ \ 2\ \ 7\ \ 8\ \ 10\ \ 11\ \ 12\ ]$$ Row-pointer

CSR SpMV

Input A, x, y Output $y = A \cdot x$

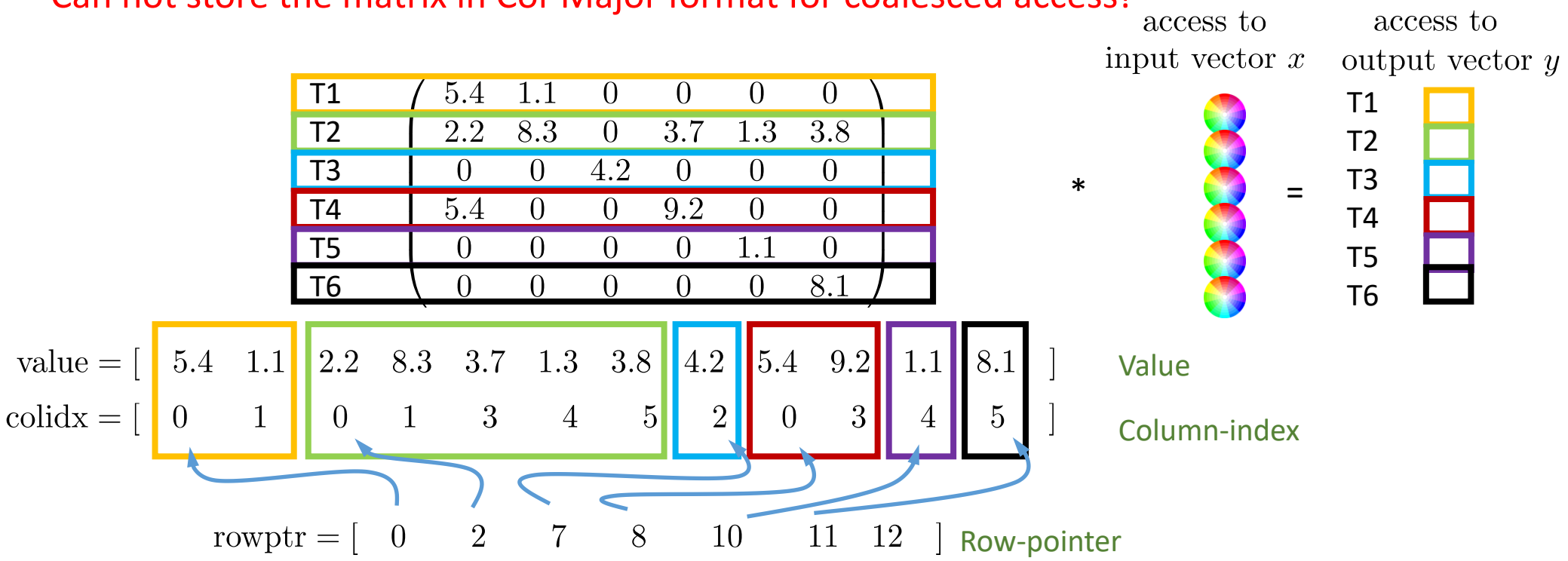
- Parallelize by rows:
  - Every “thread” handles the computation of one sum in local memory.



CSR SpMV

Input A, x, y Output $y = A \cdot x$

- Parallelize by rows:
  - Every “thread” handles the computation of one sum in local memory.
  - Significant workload imbalance!
  - Can not store the matrix in Col-Major format for coalesced access!



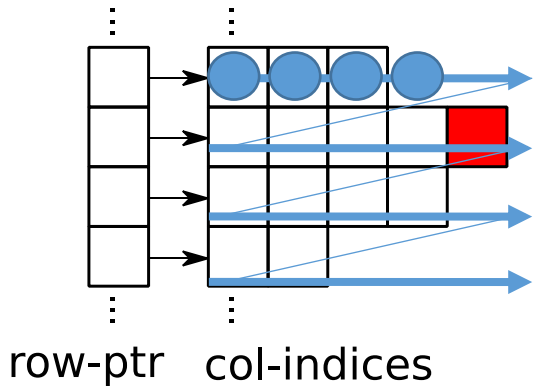
CSR SpMV

Input A, x, y Output $y = A \cdot x$

```
for( row=0; row<n; row++ )
{
        sum = 0.0;
        for( j=rowptr[row]; j<rowptr[row+1]; j++)
            sum += values[ j ] * x[ colind[j] ];
        y[ row ] =  alpha * sum;
}
```

Storing values and columns in row-major.

-> On GPUs: non-coalesced memory access



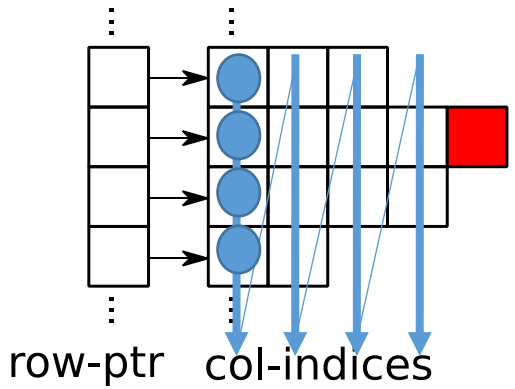
CSR SpMV

Input A, x, y Output $y = A \cdot x$

```
for( row=0; row<n; row++ )
{
        sum = 0.0;
        for( j=rowptr[row]; j<rowptr[row+1]; j++)
            sum += values[ j ] * x[ colind[j] ];
        y[ row ] =  alpha * sum;
}
```

Storing values and columns in row-major.

-> On GPUs: non-coalesced memory access



Can we use column-major?

-> Only if all rows contain the same number of nonzero elements

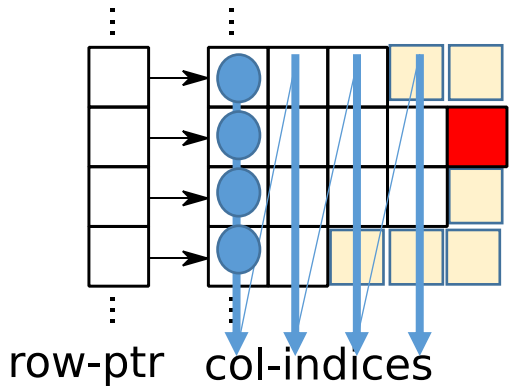
CSR SpMV

Input A, x, y Output $y = A \cdot x$

```
for( row=0; row<n; row++ )
{
        sum = 0.0;
        for( j=rowptr[row]; j<rowptr[row+1]; j++)
            sum += values[ j ] * x[ colind[j] ];
        y[ row ] =  alpha * sum;
}
```

Storing values and columns in row-major.

-> On GPUs: non-coalesced memory access



Can we use column-major?

-> Only if all rows contain the same number of nonzero elements

ELL Format

Input A, x, y Output $y = A \cdot x$

$$A = \begin{pmatrix} 5.4 & 1.1 & 0 & 0 & 0 & 0 \\ 2.2 & 8.3 & 0 & 3.7 & 1.3 & 3.8 \\ 0 & 0 & 4.2 & 0 & 0 & 0 \\ 5.4 & 0 & 0 & 9.2 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1.1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 8.1 \end{pmatrix}$$

ELL Format

Input A, x, y Output $y = A \cdot x$

`Left-align nonzero elements'

$$A = \begin{pmatrix} 5.4 & 1.1 & 0 & 0 & 0 & 0 \\ 2.2 & 8.3 & 0 & 3.7 & 1.3 & 3.8 \\ 0 & 0 & 4.2 & 0 & 0 & 0 \\ 5.4 & 0 & 0 & 9.2 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1.1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 8.1 \end{pmatrix}$$

$$\begin{bmatrix} 5.4 & 1.1 & 0 & 0 & 0 & 0 \\ 2.2 & 8.3 & 3.7 & 1.3 & 3.8 & 0 \\ 4.2 & 0 & 0 & 0 & 0 & 0 \\ 5.4 & 9.2 & 0 & 0 & 0 & 0 \\ 1.1 & 0 & 0 & 0 & 0 & 0 \\ 8.1 & 0 & 0 & 0 & 0 & 0 \end{bmatrix} \qquad \begin{bmatrix} 0 & 1 & - & - & - & - \\ 0 & 1 & 3 & 4 & 5 & - \\ 2 & - & - & - & - & - \\ 0 & 3 & - & - & - & - \\ 4 & - & - & - & - & - \\ 5 & - & - & - & - & - \end{bmatrix}$$

ELL Format

Input A, x, y Output $y = A \cdot x$

`Left-align nonzero elements'

$$A = \begin{pmatrix} 5.4 & 1.1 & 0 & 0 & 0 & 0 \\ 2.2 & 8.3 & 0 & 3.7 & 1.3 & 3.8 \\ 0 & 0 & 4.2 & 0 & 0 & 0 \\ 5.4 & 0 & 0 & 9.2 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1.1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 8.1 \end{pmatrix}$$

$$\begin{bmatrix} 5.4 & 1.1 & 0 & 0 & 0 & 0 \\ 2.2 & 8.3 & 3.7 & 1.3 & 3.8 & 0 \\ 4.2 & 0 & 0 & 0 & 0 & 0 \\ 5.4 & 9.2 & 0 & 0 & 0 & 0 \\ 1.1 & 0 & 0 & 0 & 0 & 0 \\ 8.1 & 0 & 0 & 0 & 0 & 0 \end{bmatrix} \qquad \begin{bmatrix} 0 & 1 & - & - & - & - \\ 0 & 1 & 3 & 4 & 5 & - \\ 2 & - & - & - & - & - \\ 0 & 3 & - & - & - & - \\ 4 & - & - & - & - & - \\ 5 & - & - & - & - & - \end{bmatrix}$$

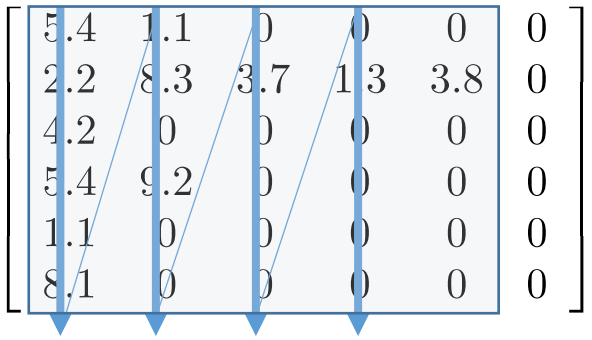
Pad rows to uniform length

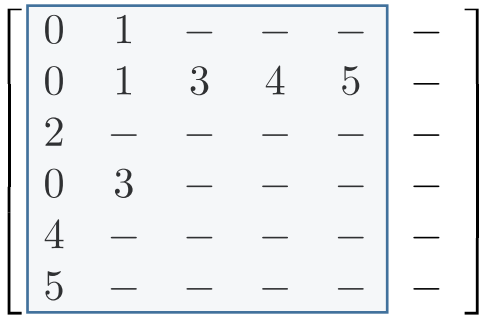
Memory volume:
- *values: max_nnz_row * num_rows*
- *col-index: max_nnz_row * num_rows*
- *no row pointer*

ELL Format

Input A, x, y Output $y = A \cdot x$

$$A = \begin{pmatrix} 5.4 & 1.1 & 0 & 0 & 0 & 0 \\ 2.2 & 8.3 & 0 & 3.7 & 1.3 & 3.8 \\ 0 & 0 & 4.2 & 0 & 0 & 0 \\ 5.4 & 0 & 0 & 9.2 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1.1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 8.1 \end{pmatrix}$$





Pad rows to uniform length

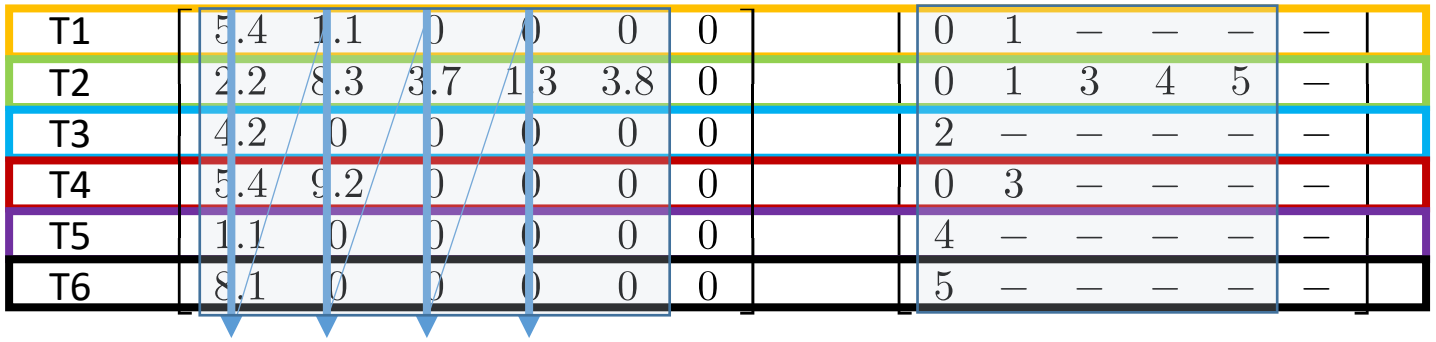
Memory volume:

- *values: max_nnz_row * num_rows*
- *col-index: max_nnz_row * num_rows*
- *no row pointer*

ELL SpMV

Input A, x, y Output $y = A \cdot x$

$$A = \begin{pmatrix} 5.4 & 1.1 & 0 & 0 & 0 & 0 \\ 2.2 & 8.3 & 0 & 3.7 & 1.3 & 3.8 \\ 0 & 0 & 4.2 & 0 & 0 & 0 \\ 5.4 & 0 & 0 & 9.2 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1.1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 8.1 \end{pmatrix}$$



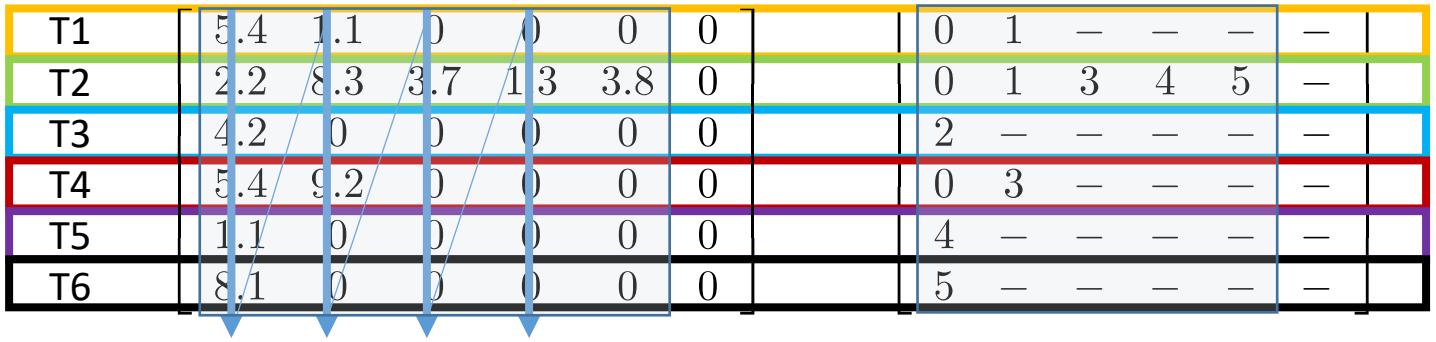
Pad rows to uniform length

Memory volume:
- *values: max_nnz_row * num_rows*
- *col-index: max_nnz_row * num_rows*
- *no row pointer*

ELL SpMV

Input A, x, y Output $y = A \cdot x$

$$A = \begin{pmatrix} 5.4 & 1.1 & 0 & 0 & 0 & 0 \\ 2.2 & 8.3 & 0 & 3.7 & 1.3 & 3.8 \\ 0 & 0 & 4.2 & 0 & 0 & 0 \\ 5.4 & 0 & 0 & 9.2 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1.1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 8.1 \end{pmatrix}$$

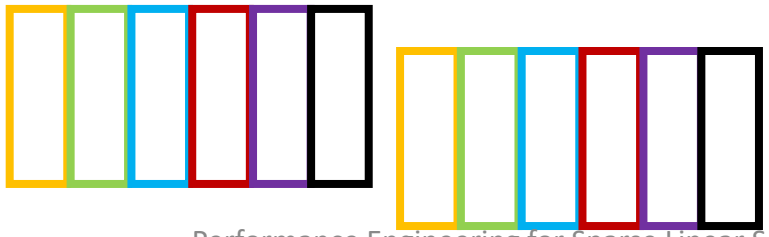


Pad rows to uniform length

Memory volume:
- *values: max_nnz_row * num_rows*
- *col-index: max_nnz_row * num_rows*
- *no row pointer*

value = [5.4 2.2 4.2 5.4 1.1 8.1 1.1 8.3 0.0 9.2 0.0 0.0 0.0 3.7 0.0 0.0 0.0 0.0 0.0 1.3 0.0 0.0 0.0 0.0 0.0 3.8 0.0 0.0 0.0 0.0]

colidx = [0 0 2 0 4 5 1 1 – 3 – – – 3 – – – – – 4 – – – – – 5 – – – –]

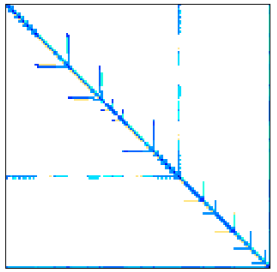


Coalesced access

ELL SpMV

Input A, x, y Output $y = A \cdot x$

$$A = \begin{pmatrix} 5.4 & 1.1 & 0 & 0 & 0 & 0 \\ 2.2 & 8.3 & 0 & 3.7 & 1.3 & 3.8 \\ 0 & 0 & 4.2 & 0 & 0 & 0 \\ 5.4 & 0 & 0 & 9.2 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1.1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 8.1 \end{pmatrix}$$



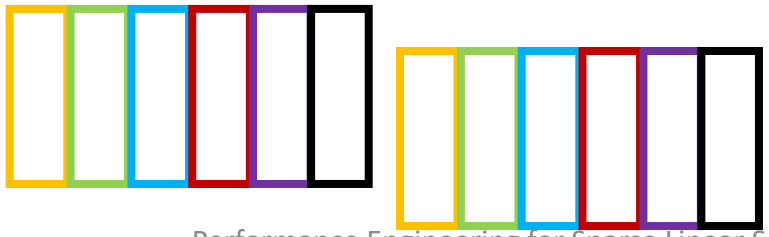
T1	5.4	1.1	0	0	0	0	0	1	–	–	–	–
T2	2.2	8.3	3.7	1.3	3.8	0	0	1	3	4	5	–
T3	4.2	0	0	0	0	0	2	–	–	–	–	–
T4	5.4	9.2	0	0	0	0	0	3	–	–	–	–
T5	1.1	0	0	0	0	0	4	–	–	–	–	–
T6	8.1	0	0	0	0	0	5	–	–	–	–	–

Pad rows to uniform length

Memory volume:
- *values: max_nnz_row * num_rows*
- *col-index: max_nnz_row * num_rows*
- *no row pointer*

value = [5.4 2.2 4.2 5.4 1.1 8.1 1.1 8.3 0.0 9.2 0.0 0.0 0.0 3.7 0.0 0.0 0.0 0.0 0.0 1.3 0.0 0.0 0.0 0.0 0.0 3.8 0.0 0.0 0.0 0.0]

colidx = [0 0 2 0 4 5 1 1 – 3 – – – 3 – – – – – 4 – – – – – 5 – – – –]



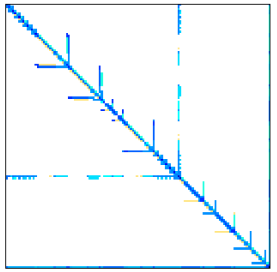
Coalesced access

Can be wasteful (overhead)

ELL SpMV

Input A, x, y Output $y = A \cdot x$

$$A = \begin{pmatrix} 5.4 & 1.1 & 0 & 0 & 0 & 0 \\ 2.2 & 8.3 & 0 & 3.7 & 1.3 & 3.8 \\ 0 & 0 & 4.2 & 0 & 0 & 0 \\ 5.4 & & & & & \\ 0 & & & & & \\ 0 & & & & & \end{pmatrix}$$



T1	5.4
T2	2.2
T3	4.2
T4	5.4
T5	1.1
T6	8.1

length

*ow * num_rows*

*_row * num_rows*

$\text{value} = [\ 5.4\ \ 2.2$ ... $0.0\ \ 0.0\ ]$

$\text{colidx} = [\ 0\ \ 0$ ... $-\ \ -\ ]$

erhead)

Hands-on Exercise: Convert this matrix into ELL format:

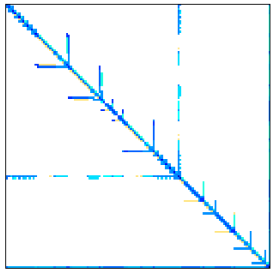
$$\begin{pmatrix} 0 & 0 & 0 & 4 & 2 & 0 \\ 0 & 2 & 3 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 2 & 3 & 4 & 3 & 1 \\ 0 & 2 & 0 & 0 & 0 & 1 \\ 1 & 2 & 3 & 0 & 0 & 1 \end{pmatrix}$$

Compute the memory requirement (# vals + # int)

ELL SpMV

Input A, x, y Output $y = A \cdot x$

$$A = \begin{pmatrix} 5.4 & 1.1 & 0 & 0 & 0 & 0 \\ 2.2 & 8.3 & 0 & 3.7 & 1.3 & 3.8 \\ 0 & 0 & 4.2 & 0 & 0 & 0 \\ 5.4 & & & & & \\ 0 & & & & & \\ 0 & & & & & \end{pmatrix}$$



T1	5.4
T2	2.2
T3	4.2
T4	5.4
T5	1.1
T6	8.1

length

*ow * num_rows*

*_row * num_rows*

$\text{value} = [\ 5.4\ \ 2.2$ ... $0.0\ \ 0.0\ ]$

$\text{colidx} = [\ 0\ \ 0$ ... $-\ \ -\ ]$

erhead)

Hands-on Exercise: Convert this matrix into ELL format:

$$\begin{pmatrix} 0 & 0 & 0 & 4 & 2 & 0 \\ 0 & 2 & 3 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 2 & 3 & 4 & 3 & 1 \\ 0 & 2 & 0 & 0 & 0 & 1 \\ 1 & 2 & 3 & 0 & 0 & 1 \end{pmatrix}$$

Compute the memory requirement (# vals + # int)

`36 vals + 36 int`

Sliced-ELL Format

Input A, x, y Output $y = A \cdot x$

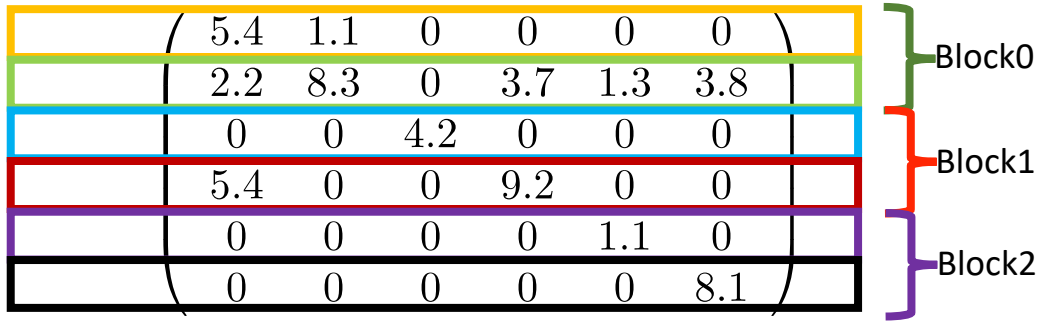
- Partition the matrix into blocks & use ELL for the distinct blocks.
  - Reduce overhead of ELL.
  - Can still store col-major.

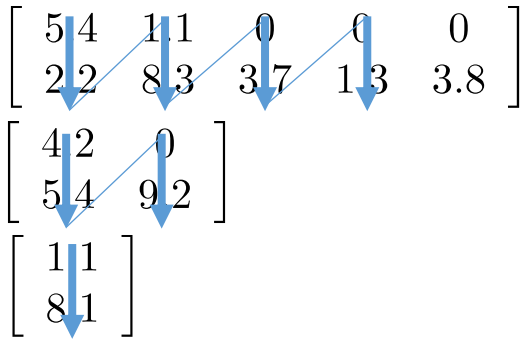
$$A = \begin{pmatrix} 5.4 & 1.1 & 0 & 0 & 0 & 0 \\ 2.2 & 8.3 & 0 & 3.7 & 1.3 & 3.8 \\ 0 & 0 & 4.2 & 0 & 0 & 0 \\ 5.4 & 0 & 0 & 9.2 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1.1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 8.1 \end{pmatrix}$$

Sliced-ELL Format

Input A, x, y Output $y = A \cdot x$

- Partition the matrix into blocks & use ELL for the distinct blocks.
  - Reduce overhead of ELL.
  - Can still store col-major.

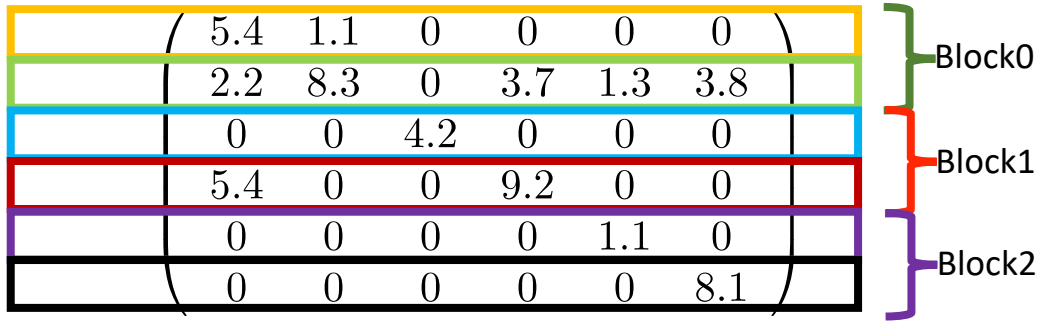


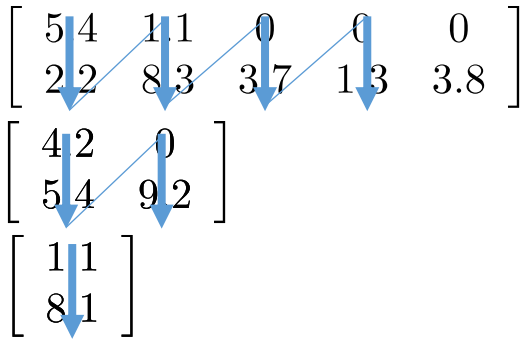


Sliced-ELL Format

Input A, x, y Output $y = A \cdot x$

- Partition the matrix into blocks & use ELL for the distinct blocks.
  - Reduce overhead of ELL.
  - Can still store col-major.





$$\text{value} = [\ 5.4\ 2.2\ 1.1\ 8.3\ 0.0\ 3.7\ 0.0\ 1.3\ 0.0\ 3.8\ |\ 4.2\ 5.4\ 0.0\ 9.2\ |\ 1.1\ 8.1\ ]$$
$$\text{colidx} = [\ 0\ \ 0\ \ 1\ \ 1\ \ -\ \ 3\ \ -\ \ 4\ \ -\ \ 5\ |\ 2\ \ 0\ \ -\ \ 3\ |\ 4\ \ 5\ ]$$

Sliced-ELL Format

Input A, x, y Output $y = A \cdot x$

- Partition the matrix into blocks & use ELL for the distinct blocks.
  - Reduce overhead of ELL.
  - Can still store col-major.
  - Need for a row pointer.

sliced-ELL format :

$$\begin{pmatrix} 5.4 & 1.1 & 0 & 0 & 0 & 0 \\ 2.2 & 8.3 & 0 & 3.7 & 1.3 & 3.8 \\ 0 & 0 & 4.2 & 0 & 0 & 0 \\ 5.4 & 0 & 0 & 9.2 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1.1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 8.1 \end{pmatrix}$$

Block0
Block1
Block2

$$\begin{bmatrix} 5.4 & 1.1 & 0 & 0 & 0 \\ 2.2 & 8.3 & 3.7 & 1.3 & 3.8 \end{bmatrix}$$

$$\begin{bmatrix} 4.2 & 0 \\ 5.4 & 9.2 \end{bmatrix}$$

$$\begin{bmatrix} 1.1 \\ 8.1 \end{bmatrix}$$

$$\text{value} = [\ 5.4\ 2.2\ 1.1\ 8.3\ 0.0\ 3.7\ 0.0\ 1.3\ 0.0\ 3.8\ |\ 4.2\ 5.4\ 0.0\ 9.2\ |\ 1.1\ 8.1\ ]$$
$$\text{colidx} = [\ 0\ \ 0\ \ 1\ \ 1\ \ -\ \ 3\ \ -\ \ 4\ \ -\ \ 5\ |\ 2\ \ 0\ \ -\ \ 3\ |\ 4\ \ 5\ ]$$

rowptr = [0 10 14 16]

Slice matrix into blocks, store blocks in ELL format with offset-pointer.

Sliced-ELL GeMV

Input A, x, y Output $y = A \cdot x$

Hands-on Exercise: Convert this matrix into Sliced-ELL format (SELL-2):

$$\begin{pmatrix} 0 & 0 & 0 & 4 & 2 & 0 \\ 0 & 2 & 3 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 2 & 3 & 4 & 3 & 1 \\ 0 & 2 & 0 & 0 & 0 & 1 \\ 1 & 2 & 3 & 0 & 0 & 1 \end{pmatrix}$$

Compute the memory requirement (# vals + # int)

Sliced-ELL GeMV

Input A, x, y Output $y = A \cdot x$

Hands-on Exercise: Convert this matrix into Sliced-ELL format (SELL-2):

$$\begin{pmatrix} 0 & 0 & 0 & 4 & 2 & 0 \\ 0 & 2 & 3 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 2 & 3 & 4 & 3 & 1 \\ 0 & 2 & 0 & 0 & 0 & 1 \\ 1 & 2 & 3 & 0 & 0 & 1 \end{pmatrix}$$

```
Rowptr: 0 6 18 26
```

Compute the memory requirement (# vals + # int)

```
26 vals + 26 + 4 int
```

Sliced-ELL GeMV

Input A, x, y Output $y = A \cdot x$

- How can we optimize this? Minimize the overhead? What is the overhead dependent on?
  - Bring rows with similar number of nonzero elements into the same block.
  - Sort rows by “length” and reorder the matrix, then convert to Sliced-ELL

Software and High-Performance Computing

A Unified Sparse Matrix Data Format for Efficient General Sparse Matrix-Vector Multiplication on Modern Processors with Wide SIMD Units

Authors: Moritz Kreutzer, Georg Hager, Gerhard Wellein, Holger Fehske, and Alan R. Bishop | AUTHORS INFO & AFFILIATIONS

https://doi.org/10.1137/130930352

  - What happens for block-size 1?
  - What happens for block size n (matrix size)?

SpMV Formats and Kernels

Input A, x, y Output $y = A \cdot x$

"Different kernels optimal for different problems"

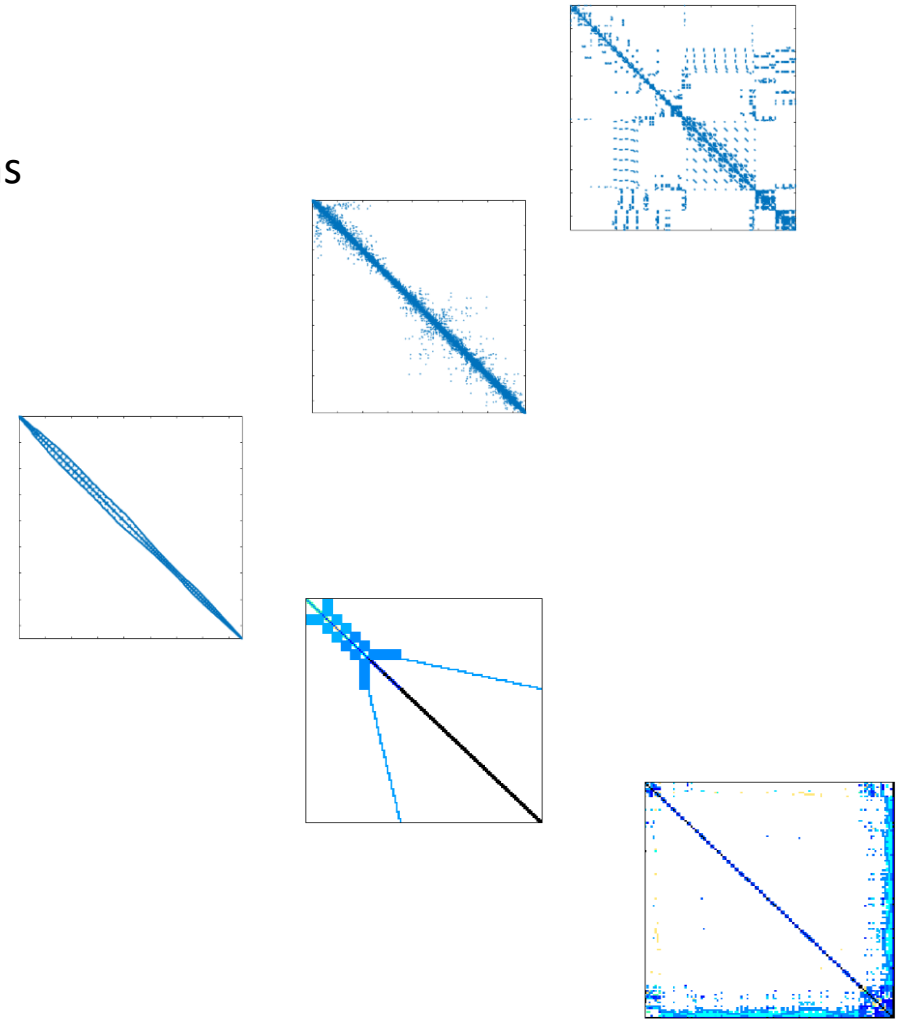
COO
- can compensate workload imbalance for irregular patterns
- Efficient for MIMD processing
- Strong support for atomics needed

CSR
- small memory footprint
- Needs some logic for row-parallel processing
- Efficient for MIMD processing

ELL
- Efficient for balanced matrices
- Enables col-major storage
- Efficient for SIMD processing

SELL-c
- Enables col-major storage
- Tunable between CSR and ELL

...



SpMV (CSR) Roofline

```
for (int row = 0; row < num_rows; ++row) {
  double sum = 0.0;
  for (int k = row_ptrs[row]; k < row_ptrs[row + 1]; ++k)
    sum += mat_values[k] * b[col_idxs[k]];
  x[row] += sum;
}
```

Optimistic intensity:

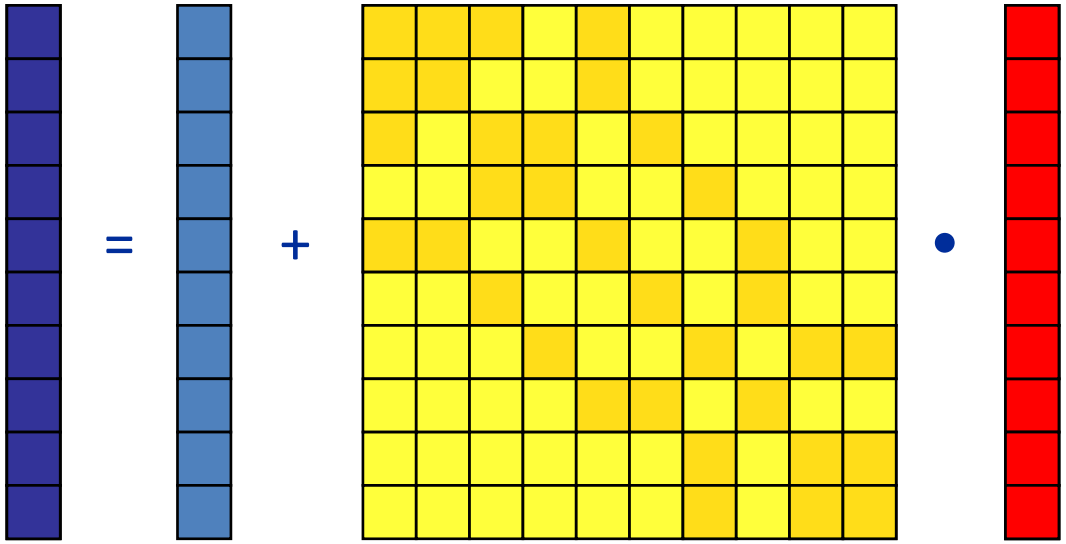
$$I_{max} = \frac{2\,N_{nz}}{12\,N_{nz} + 20\,N_r + 8\,N_c}\frac{\mathrm{F}}{\mathrm{B}}$$

$$= \frac{1}{6+10/N_{nzr}+4/N_{nzc}}\frac{\mathrm{F}}{\mathrm{B}}$$

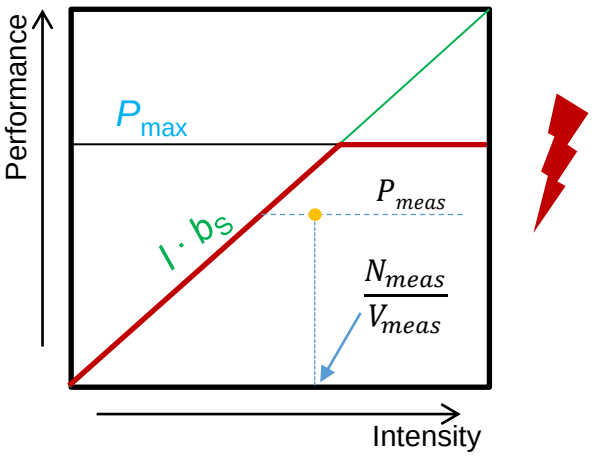
$$= \frac{1}{6+10/N_{nzr}+4/N_{nzr}}\frac{\mathrm{F}}{\mathrm{B}}$$

square matrix

$$\xrightarrow{N_{nzr}\gg 10} \frac{1}{6}\frac{\mathrm{F}}{\mathrm{B}}$$



Roofline “failure” with SpMV



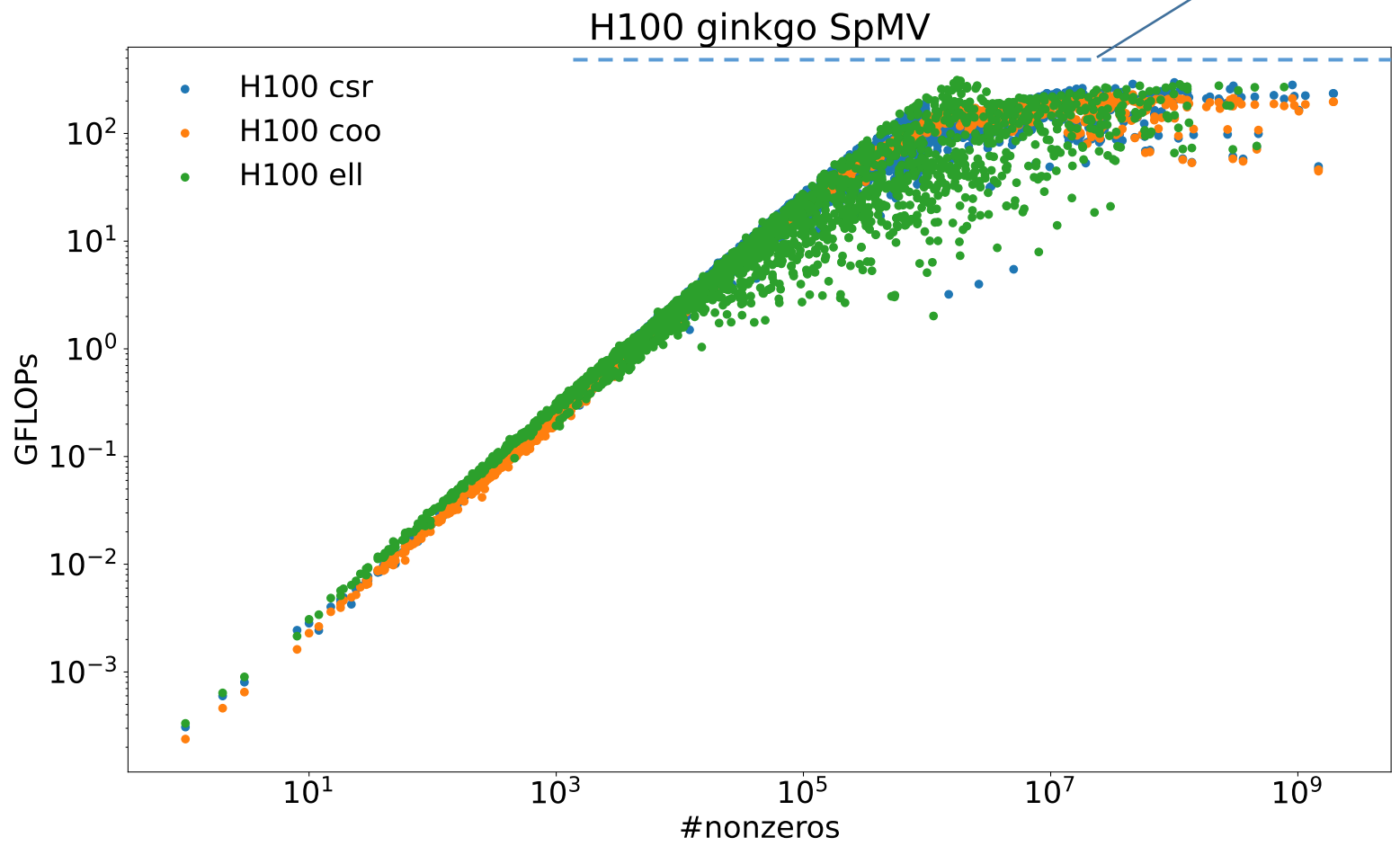
Reasons for performance not attaining the limit

1. Intensity lower than the minimum
   - More RHS traffic than the optimistic limit ($\frac{4}{N_{nzr}}$ B/F)
2. “Slow code”
   - “invisible” performance ceiling due to inefficient instructions or inefficient execution
3. Load imbalance
   - A single process/thread cannot saturate the memory bandwidth
4. Erratic memory access patterns for RHS
   - Latency dominates

Experiences with SpMV on GPUs

Looking at ~3,000 test matrices from Suite Sparse Matrix Collection

Absolute CSR limit for $b_S = 3\ \mathrm{TB/s}$



Hands-On:

SpMV benchmarking

Solving Sparse Linear Systems

Why we often don’t use direct solvers

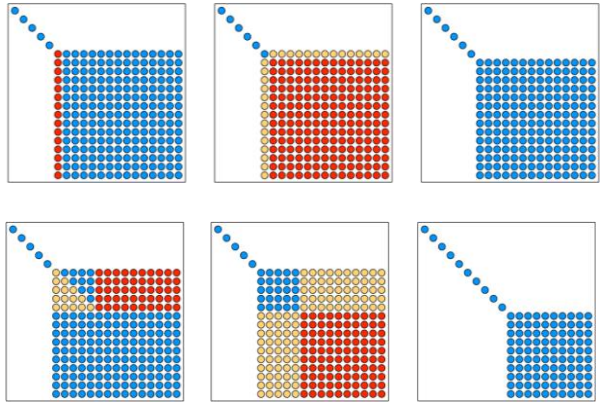
Characteristics and optimization of iterative solvers

Preconditioning using Matrix Polynomials

Jonas Thies (j.thies@tudelft.nl)

Can we use direct solvers for solving sparse problems?

$$A = \begin{pmatrix} 5.4 & 1.1 & 0 & 0 & 0 & 0 \\ 2.2 & 8.3 & 0 & 3.7 & 1.3 & 3.8 \\ 0 & 0 & 4.2 & 0 & 0 & 0 \\ 5.4 & 0 & 0 & 9.2 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1.1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 8.1 \end{pmatrix}$$



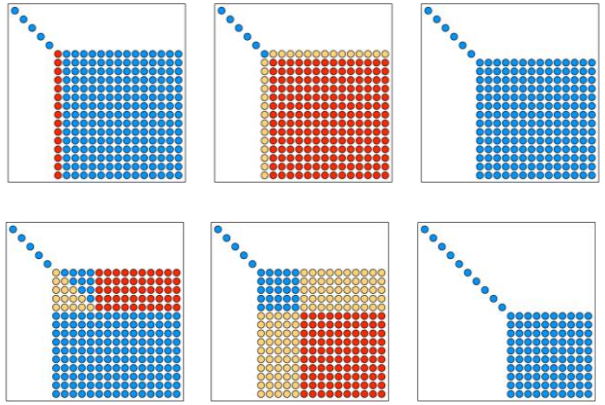
Scalar or block-LU?

Are zeros preserved in the factorization?

Can we store the fill-in?

Can we use direct solvers for solving sparse problems?

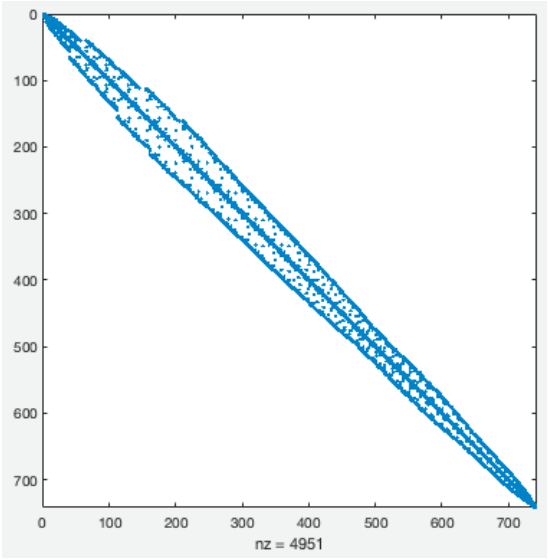
$$A = \begin{pmatrix} 5.4 & 1.1 & 0 & 0 & 0 & 0 \\ 2.2 & 8.3 & 0 & 3.7 & 1.3 & 3.8 \\ 0 & 0 & 4.2 & 0 & 0 & 0 \\ 5.4 & 0 & 0 & 9.2 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1.1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 8.1 \end{pmatrix}$$



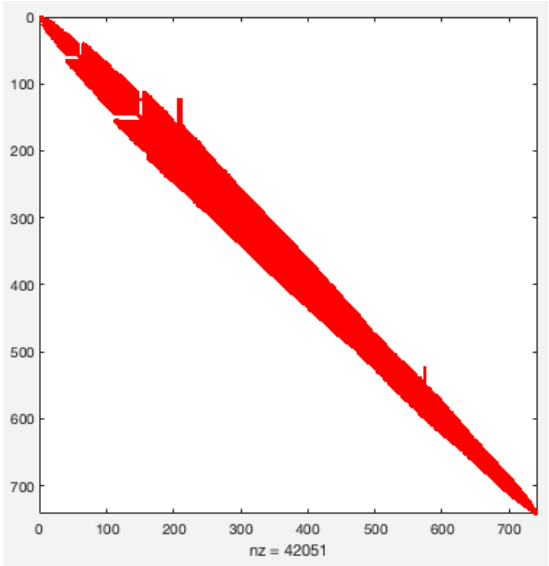
Scalar or block-LU?

Are zeros preserved in the factorization?

Can we store the fill-in?



LU



Iterative Solvers

Generate a sequence of solution approximations with increasing approximation quality.

$$x^0 \rightsquigarrow x^1 \rightsquigarrow x^2 \rightsquigarrow x^3 \rightsquigarrow \cdots$$

Iterative Solvers

Generate a sequence of solution approximations with increasing approximation quality.

$$x^0 \rightsquigarrow x^1 \rightsquigarrow x^2 \rightsquigarrow x^3 \rightsquigarrow \cdots$$

Relaxations

- Base on matrix splitting
- Jacobi relaxation:

$$\begin{aligned} Ax &= b \\ (L+D+U)x &= b \\ Dx &= b-(L+U)x \\ x &= D^{-1}b - D^{-1}(L+U)x \end{aligned}$$

$$x^{k+1} = D^{-1}b - D^{-1}(A-D)x^k$$

- Matrix-Vector Prod. & Vector Ops
- Low arithmetic intensity

Iterative Solvers

Generate a sequence of solution approximations with increasing approximation quality.

$$x^0 \rightsquigarrow x^1 \rightsquigarrow x^2 \rightsquigarrow x^3 \rightsquigarrow \cdots$$

Relaxations

- Base on matrix splitting
- Jacobi relaxation:

$$\begin{aligned} Ax &= b \\ (L+D+U)x &= b \\ Dx &= b-(L+U)x \\ x &= D^{-1}b - D^{-1}(L+U)x \end{aligned}$$

$$x^{k+1} = D^{-1}b - D^{-1}(A-D)x^k$$

- Matrix-Vector Prod. & Vector Ops
- Low arithmetic intensity

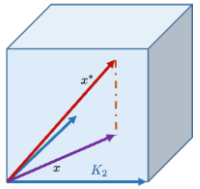
Krylov Subspace Methods

- Iteratively grow Krylov subspace

$$K_i(A,r) = span\left\{r, Ar, A^2r, \ldots, A^{i-1}r\right\}$$

$$K_0 \subset K_1 \subset K_2 \subset \cdots \mathbb{R}^n$$

- Approximate solution in Krylov Subspace



- Matrix-Vector Prod. & Vector Ops
- Low arithmetic intensity

Iterative Solvers

Generate a sequence of solution approximations with increasing approximation quality.

$$x^0 \rightsquigarrow x^1 \rightsquigarrow x^2 \rightsquigarrow x^3 \rightsquigarrow \cdots$$

Relaxations

- Base on matrix splitting
- Jacobi relaxation:

$$\begin{aligned} Ax &= b \\ (L+D+U)x &= b \\ Dx &= b-(L+U)x \\ x &= D^{-1}b - D^{-1}(L+U)x \end{aligned}$$

$$x^{k+1} = D^{-1}b - D^{-1}(A-D)x^k$$

- Matrix-Vector Prod. & Vector Ops
- Low arithmetic intensity

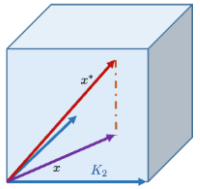
Krylov Subspace Methods

- Iteratively grow Krylov subspace

$$K_i(A,r) = span\left\{r, Ar, A^2r, \ldots, A^{i-1}r\right\}$$

$$K_0 \subset K_1 \subset K_2 \subset \cdots \mathbb{R}^n$$

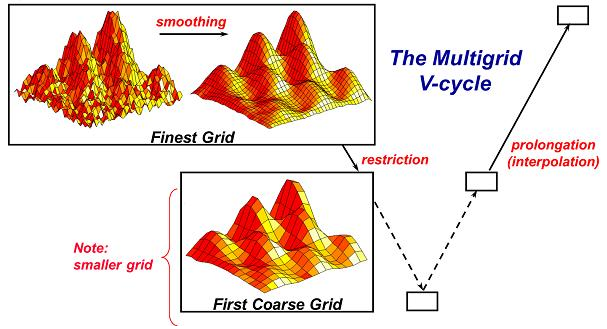
- Approximate solution in Krylov Subspace



- Matrix-Vector Prod. & Vector Ops
- Low arithmetic intensity

Multigrid Methods

- Recursively project problem to coarser grid and solve on coarser grid



- Matrix-Vector Prod. & Vector Ops
- Low arithmetic intensity

Iterative Solvers

Generate a sequence of solution approximations with increasing approximation quality.

$x^0 \rightsquigarrow x^1 \rightsquigarrow x^2 \rightsquigarrow x^3 \rightsquigarrow \cdots$

Relaxations

- Base on matrix splitting
- Jacobi relaxation:

$$Ax = b$$
$$(L + D + U)x = b$$
$$Dx = b - (L + U)x$$
$$x = D^{-1}b - D^{-1}(L + U)x$$
$$x^{k+1} = D^{-1}b - D^{-1}(A - D)x^k$$

- Matrix-Vector Prod. & Vector Ops
- Low arithmetic intensity

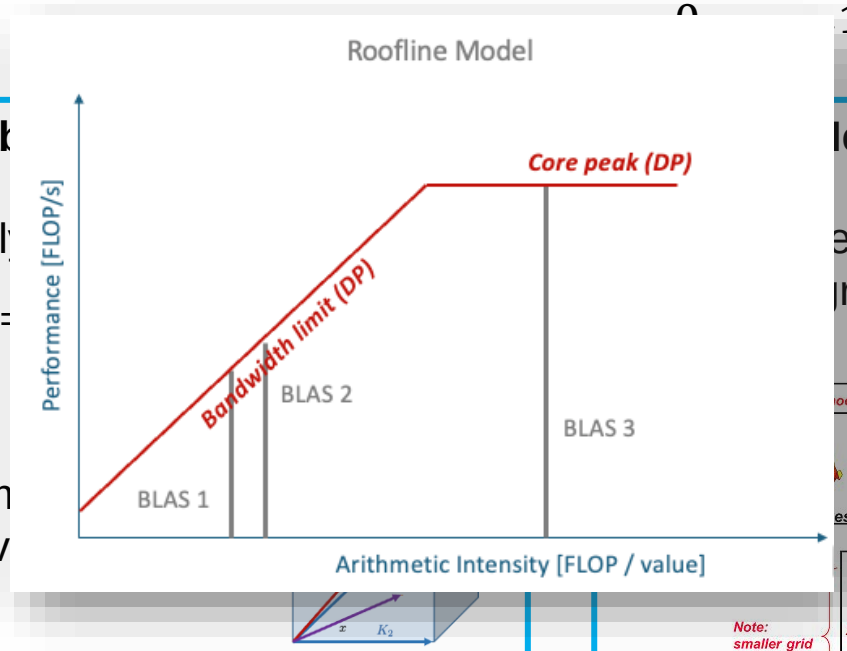
Krylov Sub

- Iterativel

$K_i(A, r) =$

$K_0 \subset K_1$

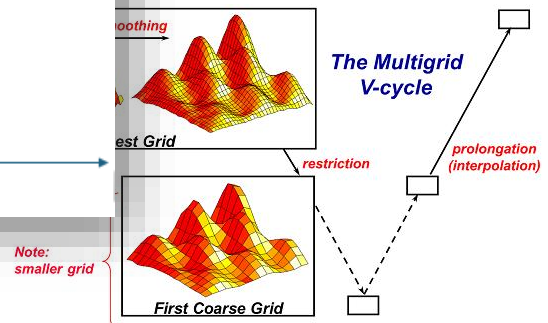
- Approxin
in Krylov



- Matrix-Vector Prod. & Vector Ops
- Low arithmetic intensity

lethods

ely project problem to
rid and solve on coarser



- Matrix-Vector Prod. & Vector Ops
- Low arithmetic intensity

Example: Conjugate Gradients (CG)

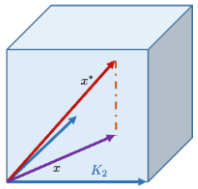
Krylov Subspace Methods

- Iteratively grow Krylov subspace

$$K_i(A,r) = span\left\{r, Ar, A^2r, \dots, A^{i-1}r\right\}$$

$$K_0 \subset K_1 \subset K_2 \subset \cdots \mathbb{R}^n$$

- Approximate solution in Krylov Subspace



- Matrix-Vector Prod. & Vector Ops
- Low arithmetic intensity

Optimal Krylov solver for symmetric and positive definite (SPD) matrices
Requires storing only four additional vectors

$$\begin{aligned}
&\text{input: } A, b, x_0, it_{max} \\
&r_0 = b - Ax_0 \\
&p_0 = r_0 \\
&\textbf{for } k = 0, \dots, it_{max} \textbf{ do} \\
&\quad \alpha_k = \frac{r_k^T r_k}{p_k^T A p_k} \\
&\quad x_{k+1} = x_k + \alpha_k p_k \\
&\quad r_{k+1} = r_k - \alpha_k A p_k \\
&\quad \beta_k = \frac{r_{k+1}^T r_{k+1}}{r_k^T r_k} \\
&\quad p_{k+1} = r_{k+1} + \beta_k p_k \\
&\textbf{end for}
\end{aligned}$$

Preconditioning Iterative Solvers

Transform linear problem by multiplying both sides with $P \approx A^{-1}$ such that iterations converge faster.

$$Ax = b \quad \Leftrightarrow \quad \underbrace{PA}_{\tilde{A}}x = \underbrace{Pb}_{\tilde{b}} \quad \Leftrightarrow \quad \tilde{A}x = \tilde{b}$$

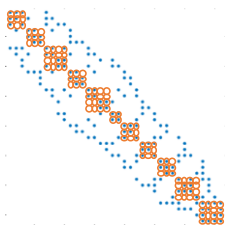
Preconditioning Iterative Solvers

Transform linear problem by multiplying both sides with $P \approx A^{-1}$ such that iterations converge faster.

$$Ax = b \quad \Leftrightarrow \quad \underbrace{PA}_{\tilde{A}}x = \underbrace{Pb}_{\tilde{b}} \quad \Leftrightarrow \quad \tilde{A}x = \tilde{b}$$

Iterative solver as preconditioner

- Multigrid
- Jacobi $D^{-1}Ax = D^{-1}b$
- Block-Jacobi



- Sparse Approximate Inverses
- Matrix-Vector Prod. & Vector Ops
- Low arithmetic intensity

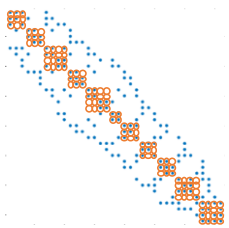
Preconditioning Iterative Solvers

Transform linear problem by multiplying both sides with $P \approx A^{-1}$ such that iterations converge faster.

$$Ax = b \quad \Leftrightarrow \quad \underbrace{PA}_{\tilde{A}}x = \underbrace{Pb}_{\tilde{b}} \quad \Leftrightarrow \quad \tilde{A}x = \tilde{b}$$

Iterative solver as preconditioner

- Multigrid
- Jacobi $D^{-1}Ax = D^{-1}b$
- Block-Jacobi



- Sparse Approximate Inverses
- Matrix-Vector Prod. & Vector Ops
- Low arithmetic intensity

Incomplete Factorizations

- Compute LU factorization with restricted fill-in

- Replace triangular solver with iteratively solving factors

$$L \cdot y = b \qquad U \cdot x = y$$

- Matrix-Vector Prod. & Vector Ops
- Low arithmetic intensity

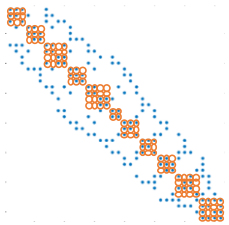
Preconditioning Iterative Solvers

Transform linear problem by multiplying both sides with $P \approx A^{-1}$ such that iterations converge faster.

$$Ax = b \quad \Leftrightarrow \quad \underbrace{PA}_{\tilde{A}}x = \underbrace{Pb}_{\tilde{b}} \quad \Leftrightarrow \quad \tilde{A}x = \tilde{b}$$

Iterative solver as preconditioner

- Multigrid
- Jacobi $D^{-1}Ax = D^{-1}b$
- Block-Jacobi



- Sparse Approximate Inverses
- Matrix-Vector Prod. & Vector Ops
- Low arithmetic intensity

Incomplete Factorizations

- Compute LU factorization with restricted fill-in

- Replace triangular solver with iteratively solving factors

$$L \cdot y = b \qquad U \cdot x = y$$

- Matrix-Vector Prod. & Vector Ops
- Low arithmetic intensity

Polynomial preconditioners

- Choose $A = M - N$

$$P = \left(\sum_{i=0}^{p-1} (I - M^{-1}A)^i \right) M^{-1}$$

- Matrix-Vector Prod. & Vector Ops
- Low arithmetic intensity

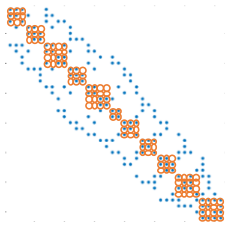
Preconditioning Iterative Solvers

Transform linear problem by multiplying both sides with $P \approx A^{-1}$ such that iterations converge faster.

$$Ax = b \quad \Leftrightarrow \quad \underbrace{PA}_{\tilde{A}}x = \underbrace{Pb}_{\tilde{b}} \quad \Leftrightarrow \quad \tilde{A}x = \tilde{b}$$

Iterative solver as preconditioner

- Multigrid
- Jacobi $D^{-1}Ax = D^{-1}b$
- Block-Jacobi

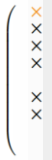


- Sparse Approximate Inverses

- Matrix-Vector Prod. & Vector Ops
- Low arithmetic intensity

Incomplete

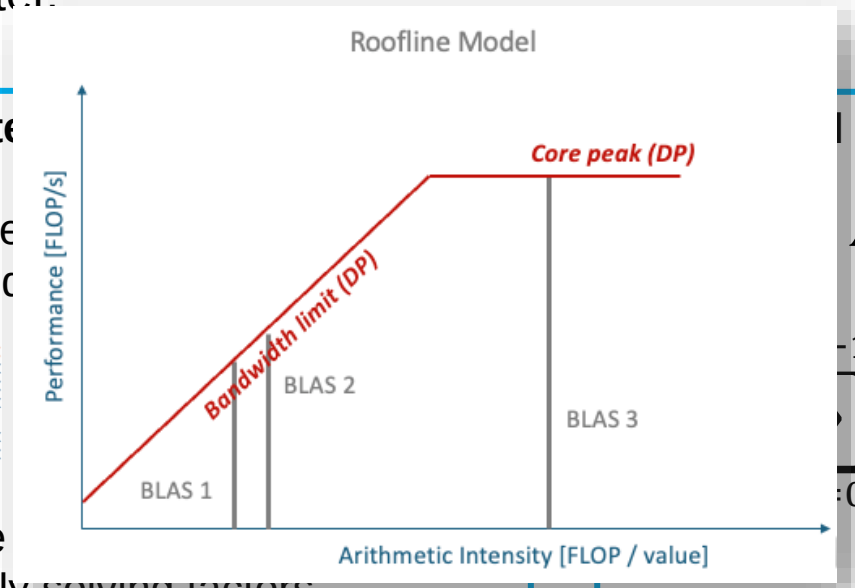
- Compute
  restricted



- Replace
  iteratively solving factors

$$L \cdot y = b \qquad U \cdot x = y$$

- Matrix-Vector Prod. & Vector Ops
- Low arithmetic intensity



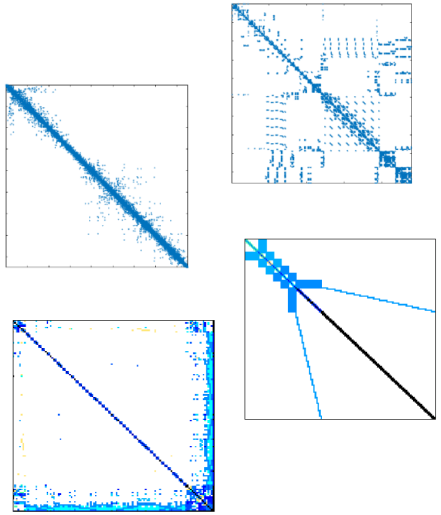
preconditioners

$$A = M - N$$

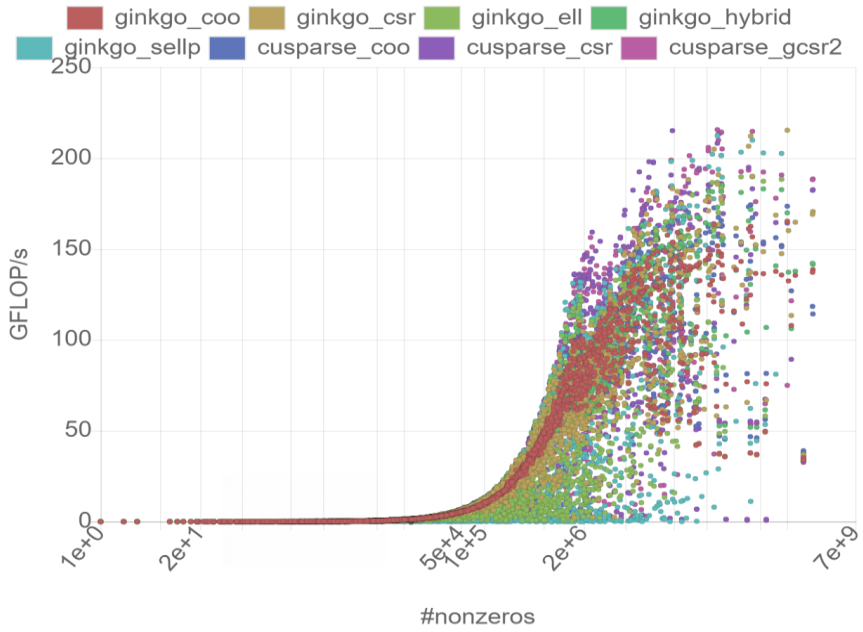
$$\left(\sum_{i=0}^{k-1} (I - M^{-1}A)^i \right) M^{-1}$$

- Matrix-Vector Prod. & Vector Ops
- Low arithmetic intensity

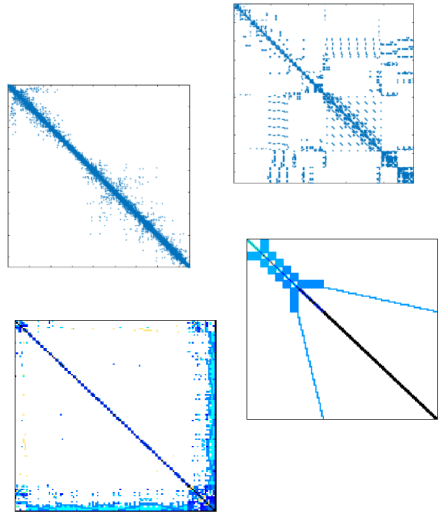
Optimizing Iterative Solvers & Preconditioners



1. **Optimizing the matrix vector product as common building block**
   - Optimization of **sparse data format** and **processing scheme**



Optimizing Iterative Solvers & Preconditioners



1. **Optimizing the matrix vector product as common building block**
   - Optimization of **sparse data format** and **processing scheme**

2. **Cache-Aware implementation**
   - Merging of Operations into **super-kernels** to reduce the memory access

BiCGStab Krylov solver (van der Vorst, 1992)

1. $r_0 = b - Ax_0$
2. Choose an arbitrary vector $\hat{r}_0$ such that $(\hat{r}_0, r_0) \neq 0$, e.g., $\hat{r}_0 = r_0$
3. $\rho_0 = \alpha = \omega_0 = 1$
4. $v_0 = p_0 = 0$
5. For $i = 1, 2, 3, \ldots$
   1. $\rho_i = (\hat{r}_0, r_{i-1})$
   2. $\beta = (\rho_i/\rho_{i-1})(\alpha/\omega_{i-1})$
   3. $p_i = r_{i-1} + \beta(p_{i-1} - \omega_{i-1} v_{i-1})$
   4. $v_i = Ap_i$
   5. $\alpha = \rho_i/(\hat{r}_0, v_i)$
   6. $s = r_{i-1} - \alpha v_i$
   7. $t = As$
   8. $\omega_i = (t, s)/(t, t)$
   9. $x_i = x_{i-1} + \alpha p_i + \omega_i s$
   10. If x_i is accurate enough then quit
   11. $r_i = s - \omega_i t$

$p_k := r_{k-1} + \beta\,(p_{k-1} - \omega_{k-1} v_{k-1})$

cuBLAS

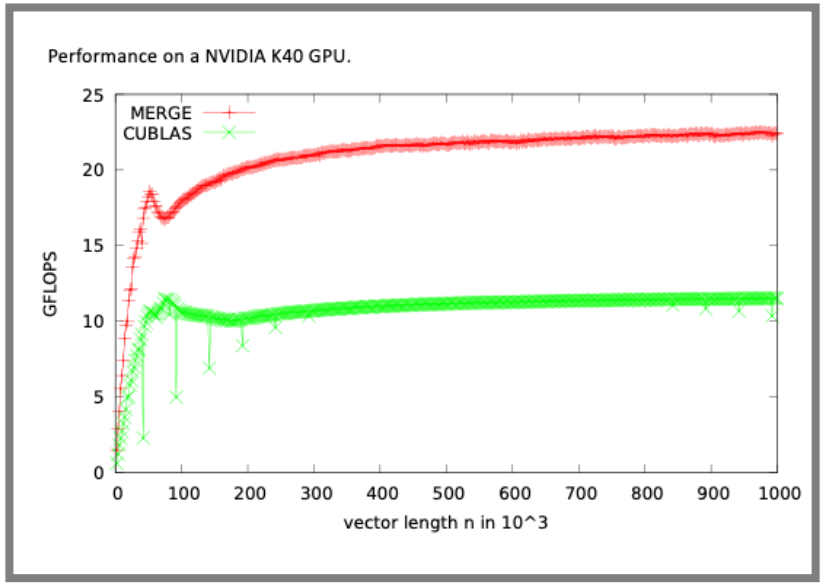
```
cublasDscal( n, beta, p, 1 );
cublasDaxpy( n, omega * beta, v, 1 , p, 1 );
cublasDaxpy( n, 1.0, r, 1, p, 1 );
```

3 kernels - 5n reads, 3n writes

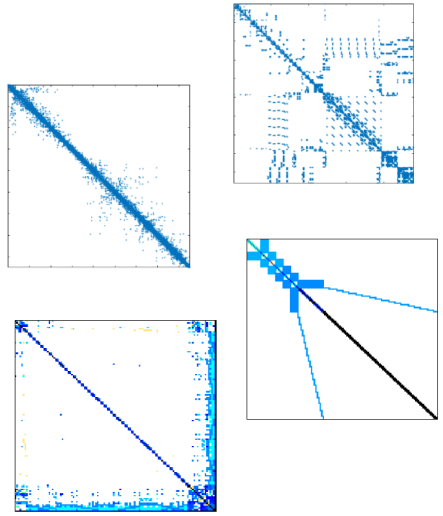
merge in one kernel

```
p_update( int n, double beta, double omega,
          double *v, double *r, double *p ){
  int i = blockIdx.x * blockDim.x + threadIdx.x;
  if( i<n )
    p[i] = r[i] + beta * ( p[i]-omega*v[i] );
}
```

1 kernel - 3n reads, 1n writes



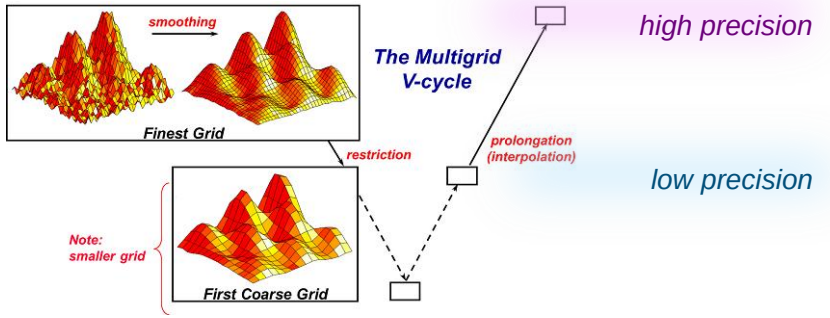
Optimizing Iterative Solvers & Preconditioners



1. **Optimizing the matrix vector product as common building block**
   - Optimization of **sparse data format** and **processing scheme**

2. **Cache-Aware implementation**
   - Merging of Operations into **super-kernels** to reduce the memory access

3. **Replace memory access with additional computations**
   - Mixed Precision algorithms using low precision in parts of the computations
   - **Matrix Powers Kernel and cache blocking**



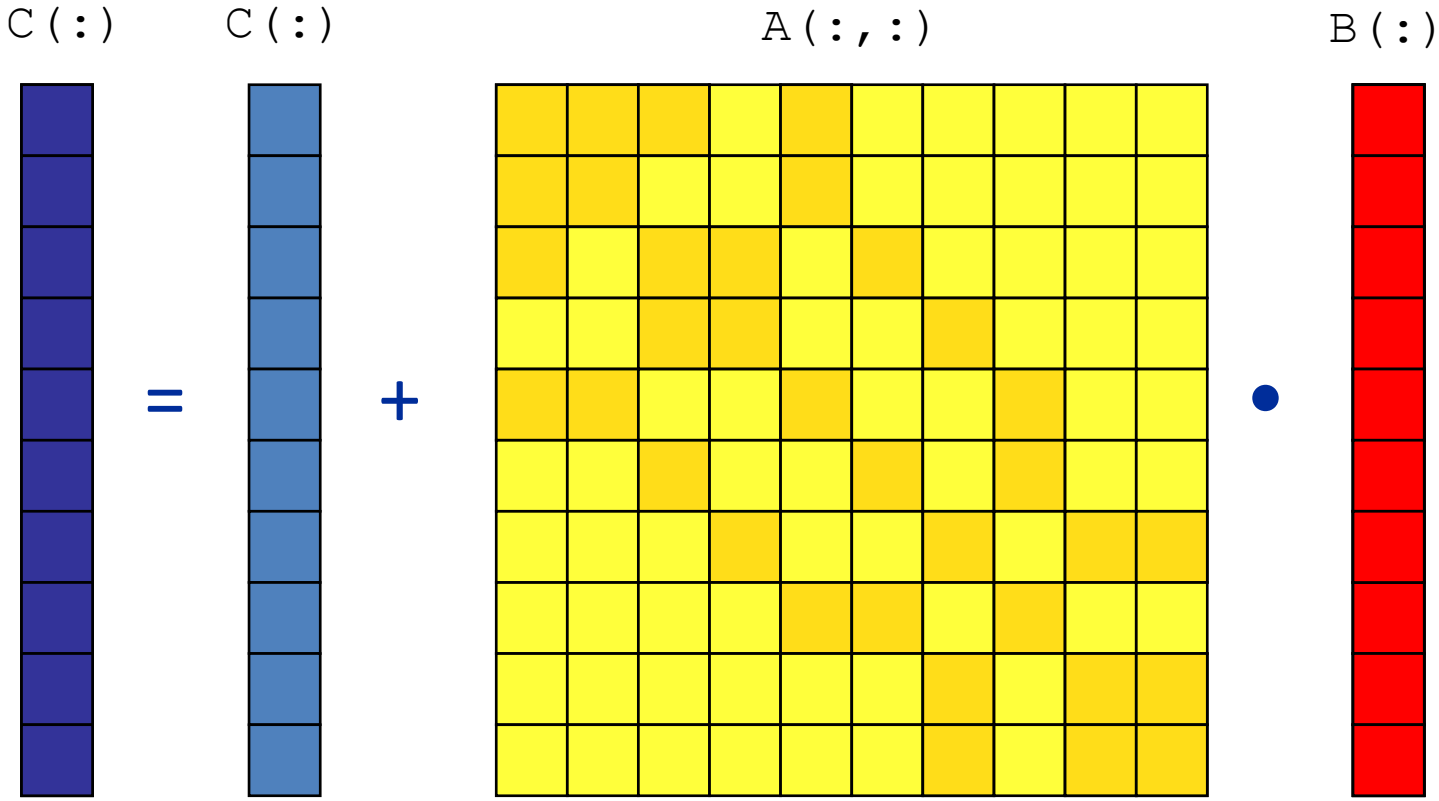
Hands-On:

Conjugate-Gradient Solver

Cache Blocking for the Matrix Power Kernel

Motivation – Sparse Matrix Vector Multiplication

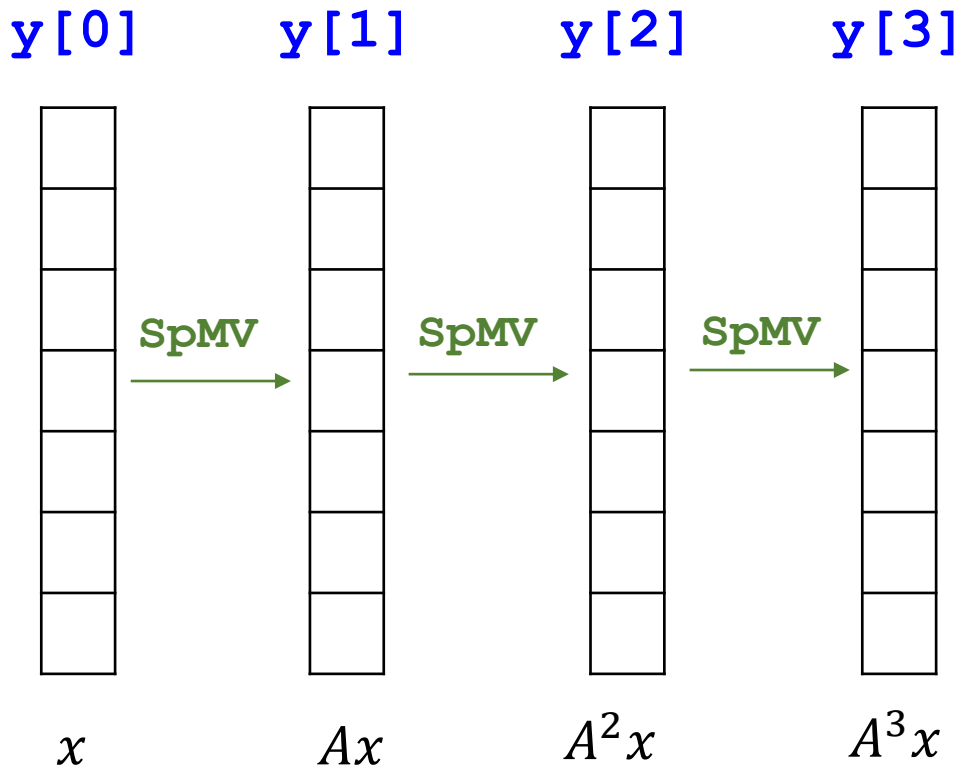
- Easy to parallelize but sparse irregular data structures / accesses
- SpMV Performance ←→ Strongly Memory Bound (high code balance)



Motivation – Matrix power kernel (MPK)

- Calculate: $y = A^p x$
- Repeatedly perform back to back SpMVs

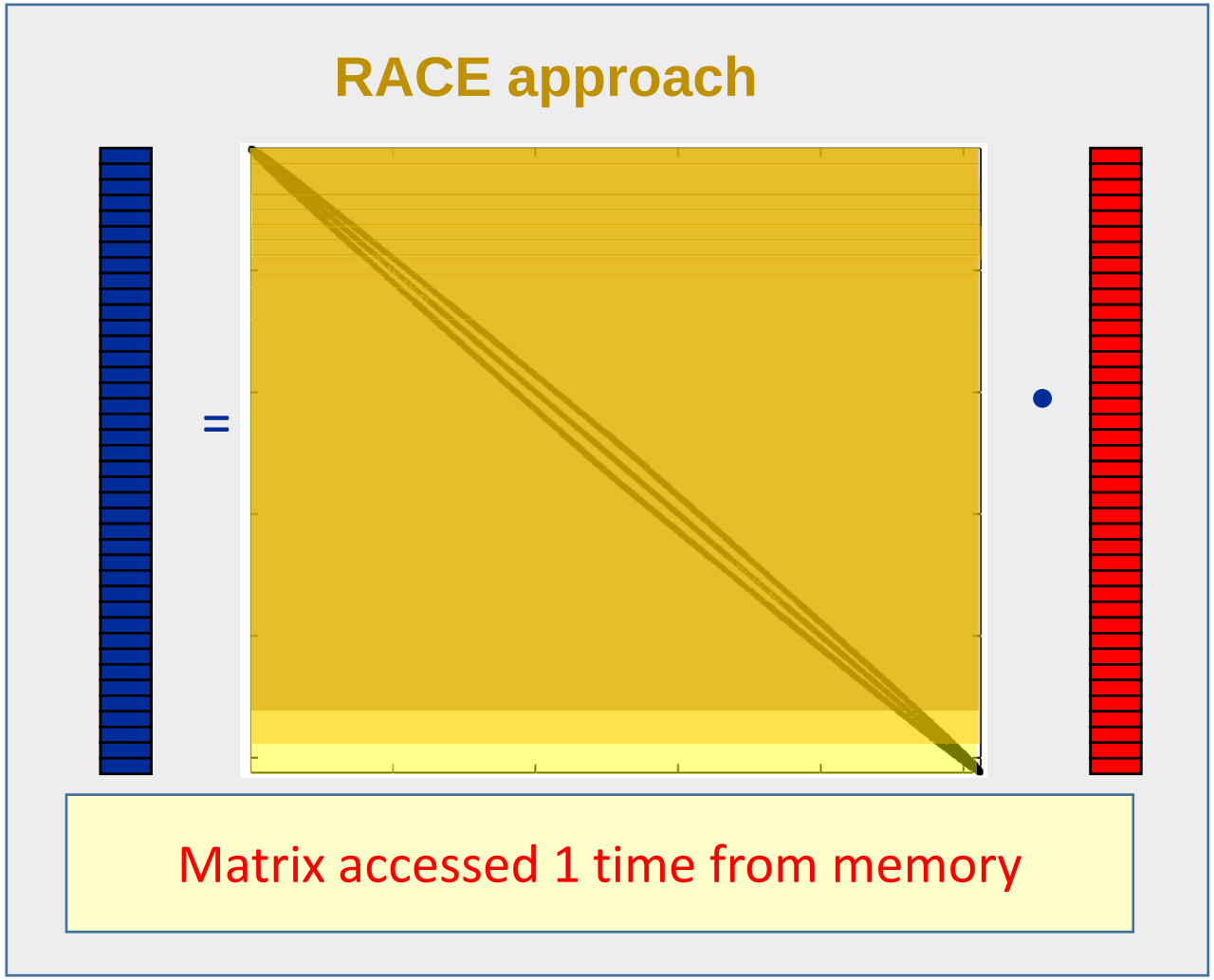
```
for k=1:p; do
 y[k] = SpMV(A, y[k-1])
done
```



Same matrix A loaded p times from main memory!!!

How to cache the matrix A across the matrix power calculation?

Matrix power – Traditional approach vs. Cache Blocking

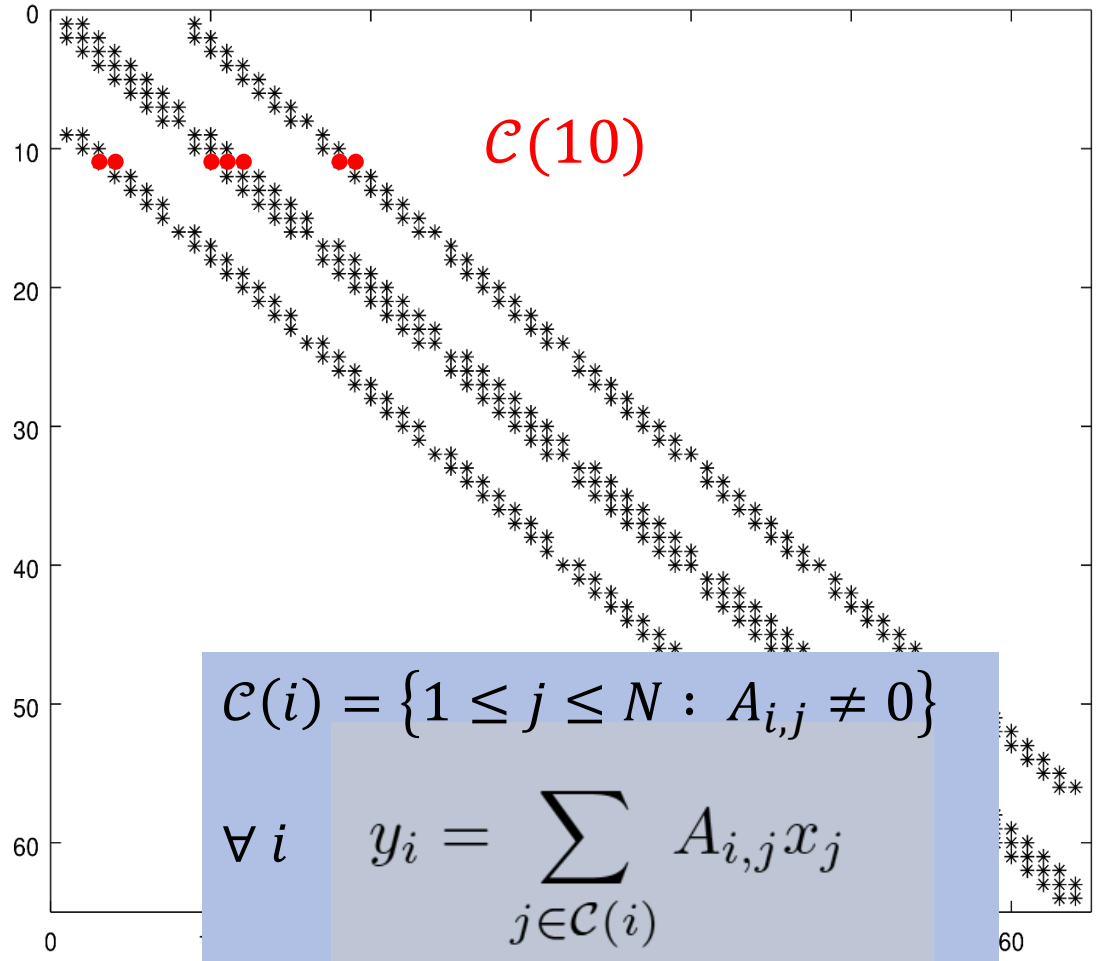


How to do that in general for sparse matrices?

SpMV – Graph Traversal – RACE

Sample matrix and its graph representation

Symmetric Matrix



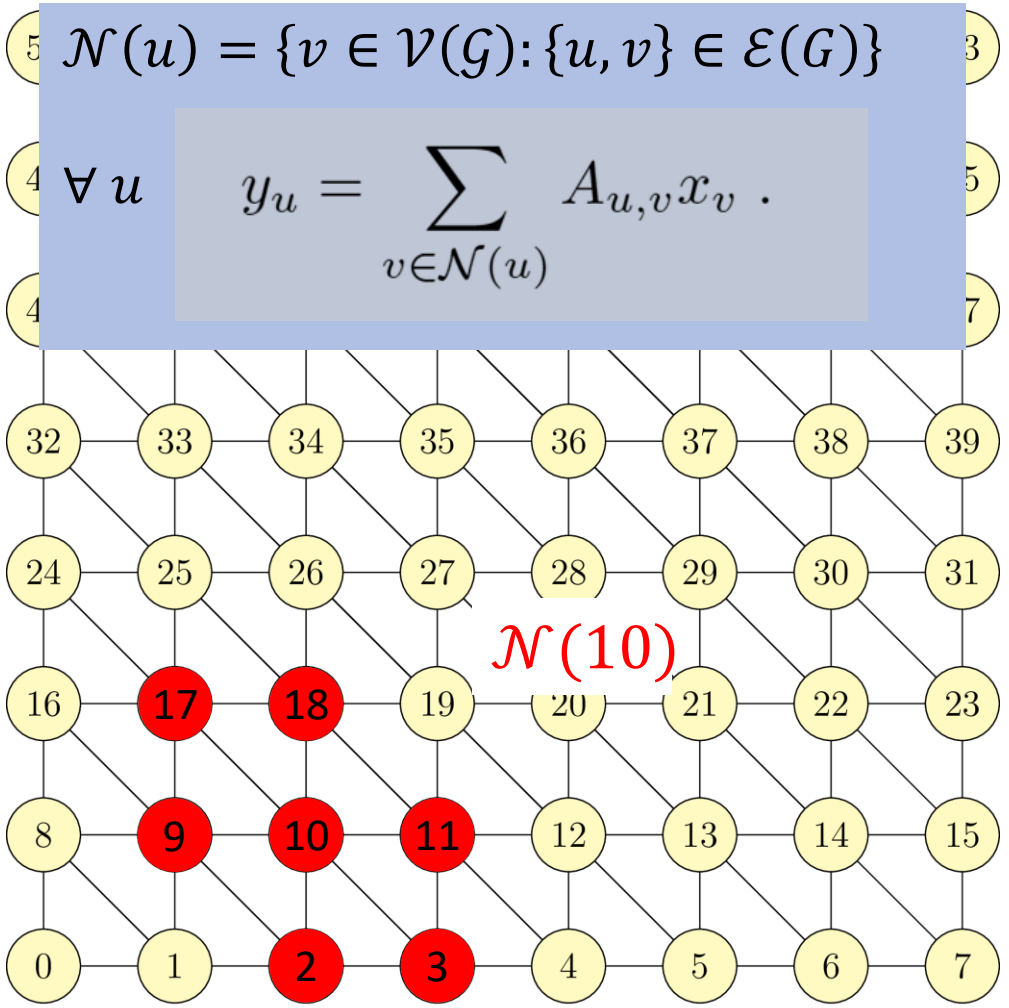
$$\mathcal{C}(i) = \{1 \le j \le N : A_{i,j} \neq 0\}$$

$$\forall\, i \qquad y_i = \sum_{j \in \mathcal{C}(i)} A_{i,j} x_j$$

Undirected Graph

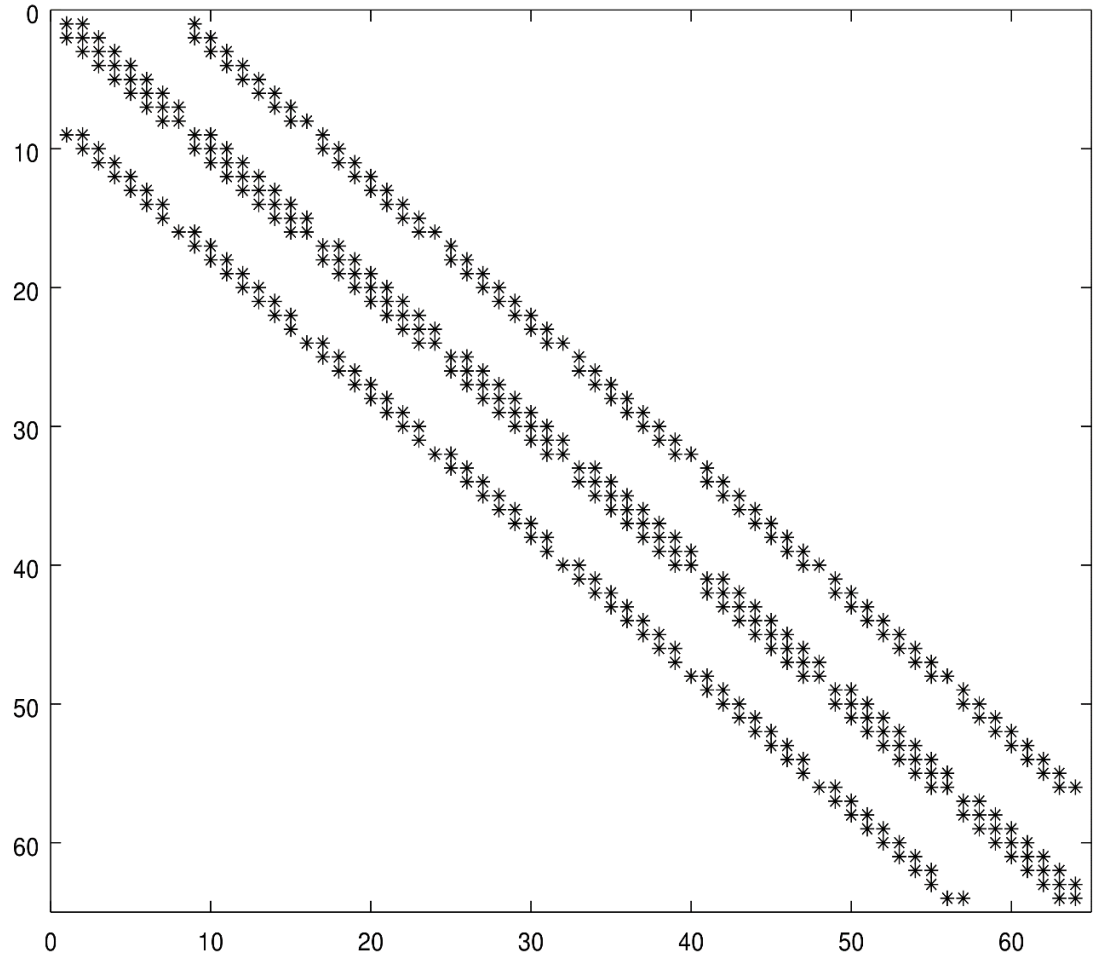
$$\mathcal{N}(u) = \{v \in \mathcal{V}(\mathcal{G}) : \{u, v\} \in \mathcal{E}(G)\}$$

$$\forall\, u \qquad y_u = \sum_{v \in \mathcal{N}(u)} A_{u,v} x_v \ .$$

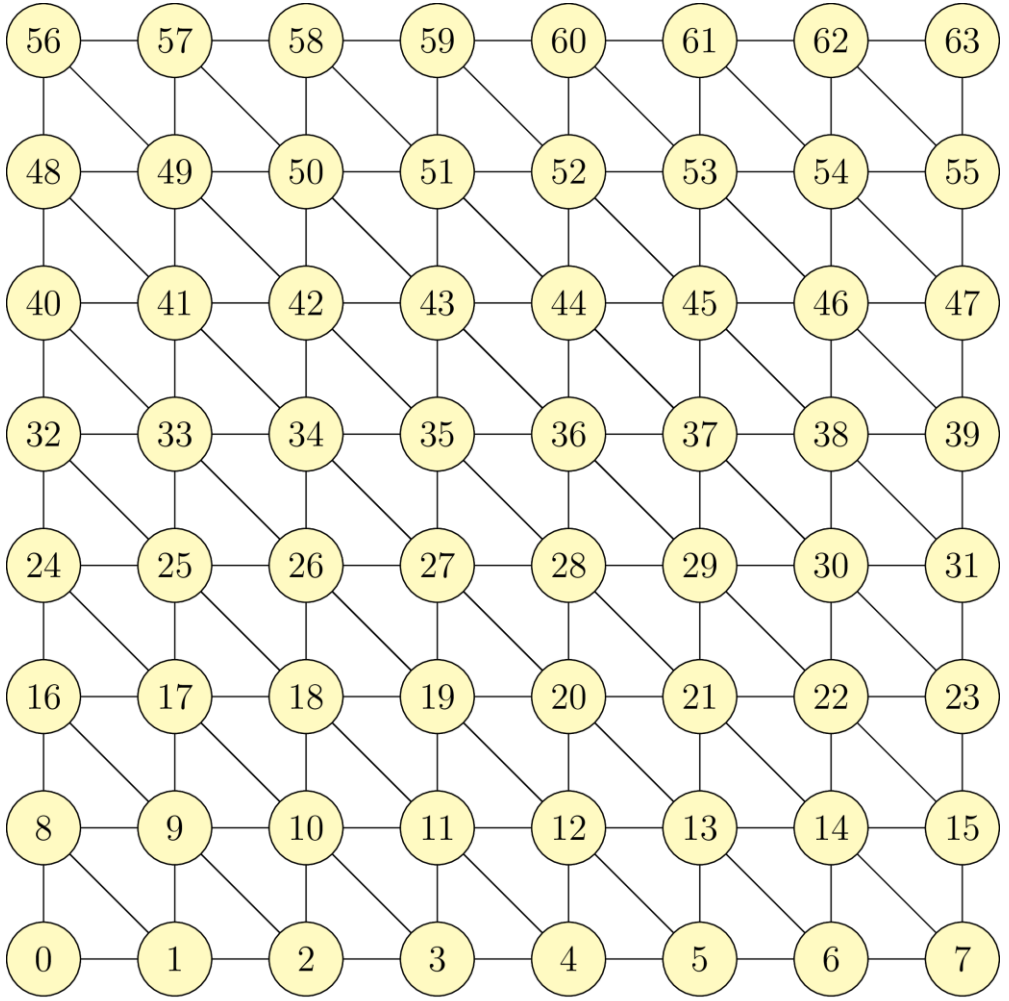


Sample matrix and its graph representation

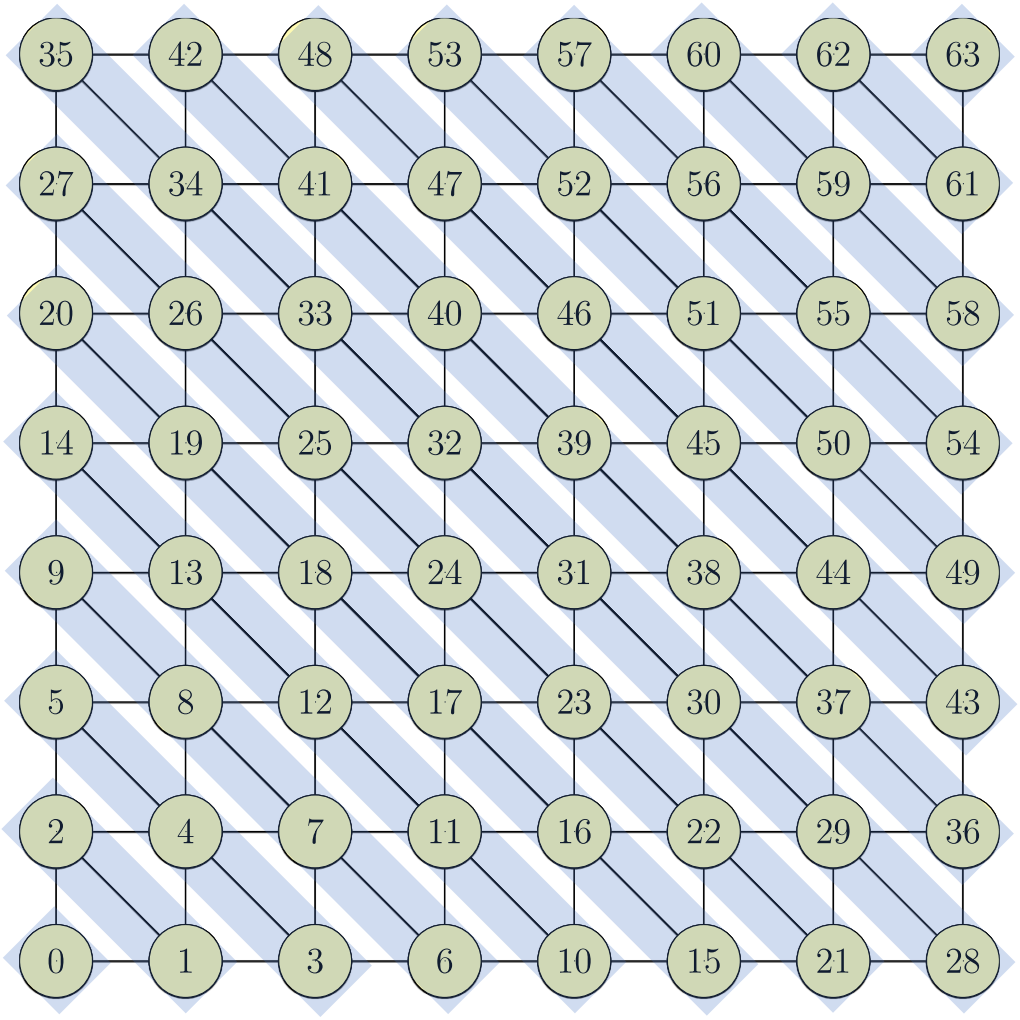
Symmetric Matrix

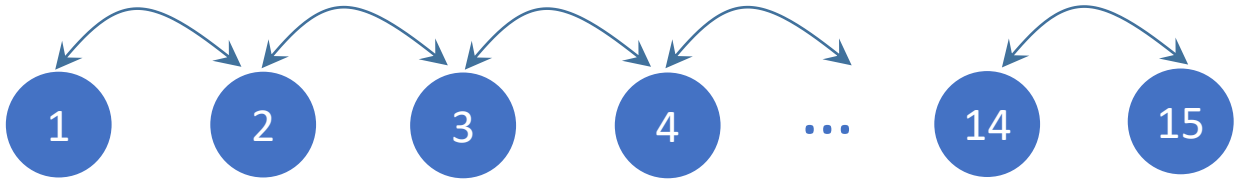


Undirected Graph



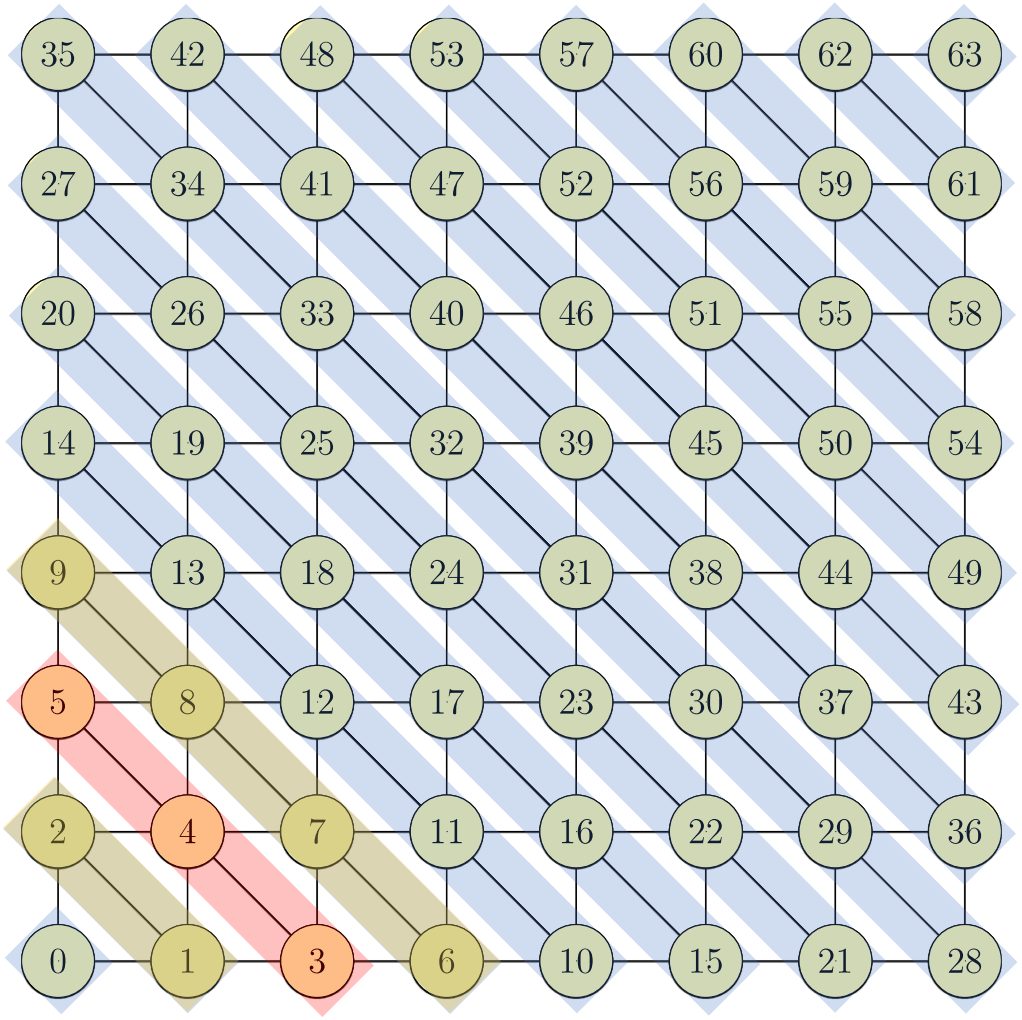
RACE

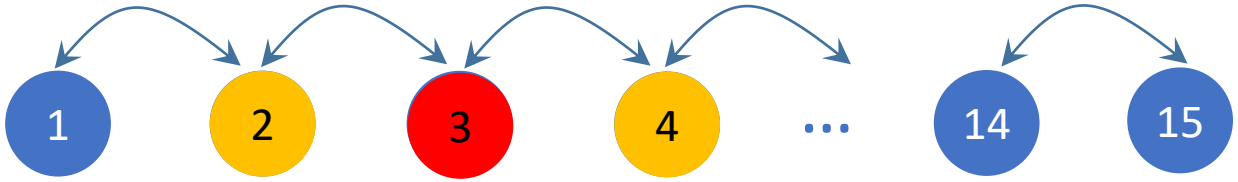

35 42 48 53 57 60 62 63
27 34 41 47 52 56 59 61
20 26 33 40 46 51 55 58
14 19 25 32 39 45 50 54
9 13 18 24 31 38 44 49
5 8 12 17 23 30 37 43
2 4 7 11 16 22 29 36
0 1 3 6 10 15 21 28


1
2
3
4
...
14
15

levels

RACE





Levels (L)

Key property

$$\mathcal{N}\big(L(i)\big) = \{L(i-1), L(i), L(i+1)\}$$

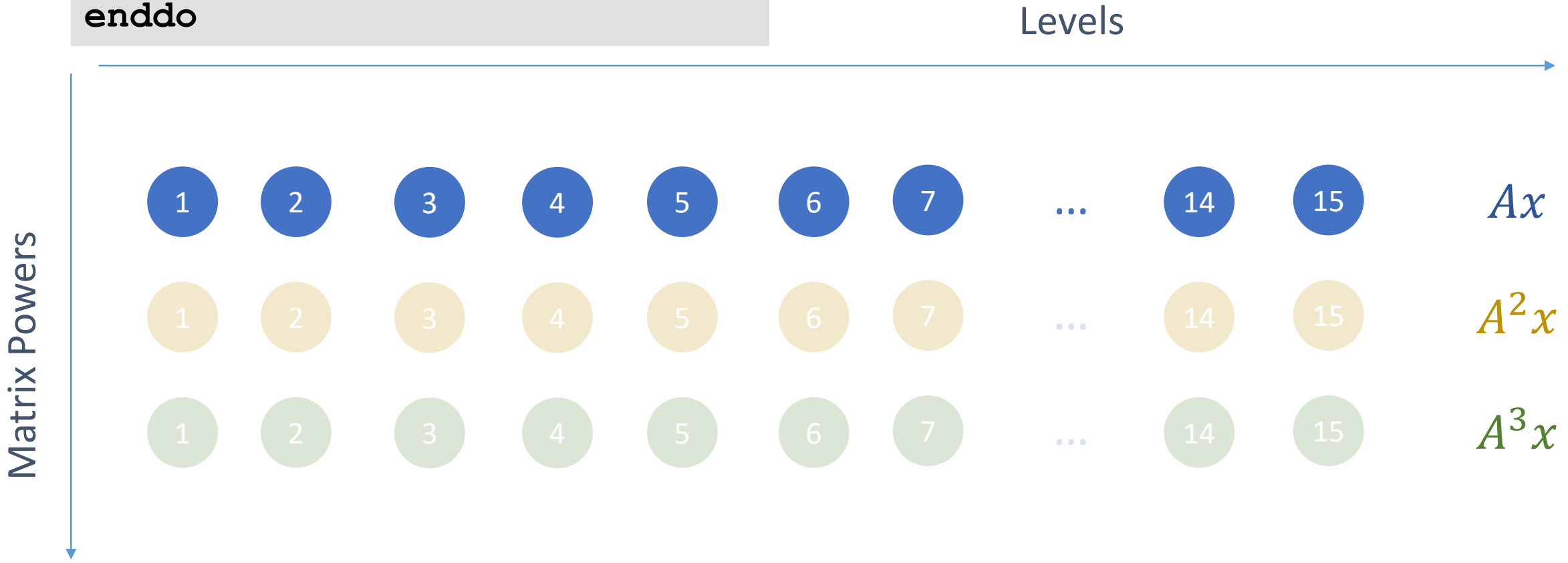
$A^p x$ computations on $L(i)$ will require $A^{p-1}x$ to be complete on $L(i-1), L(i), L(i+1)$

Neighbors localized ➔ dependencies localized

RACE – Level traversal and matrix powers

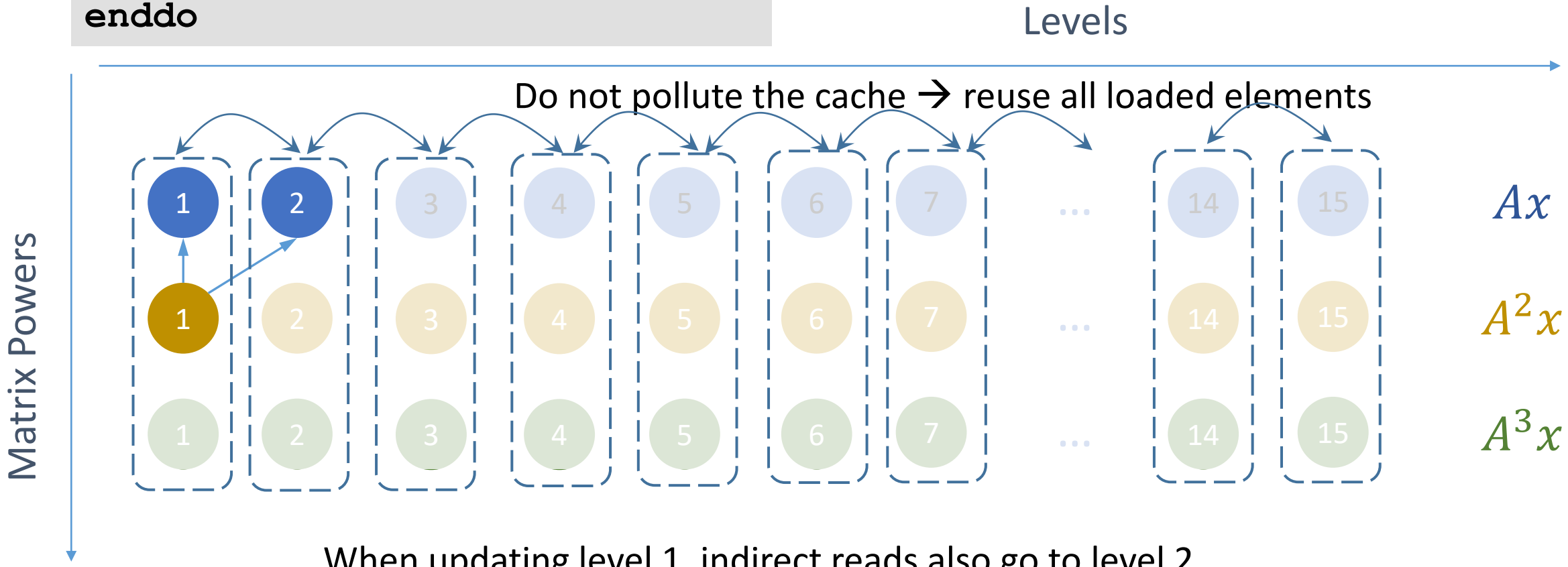
```
do k = 1, p
 y(:, k) = SpMV(A, y(:,k-1))
enddo
```

No cache blocking!



RACE – Level traversal and matrix powers

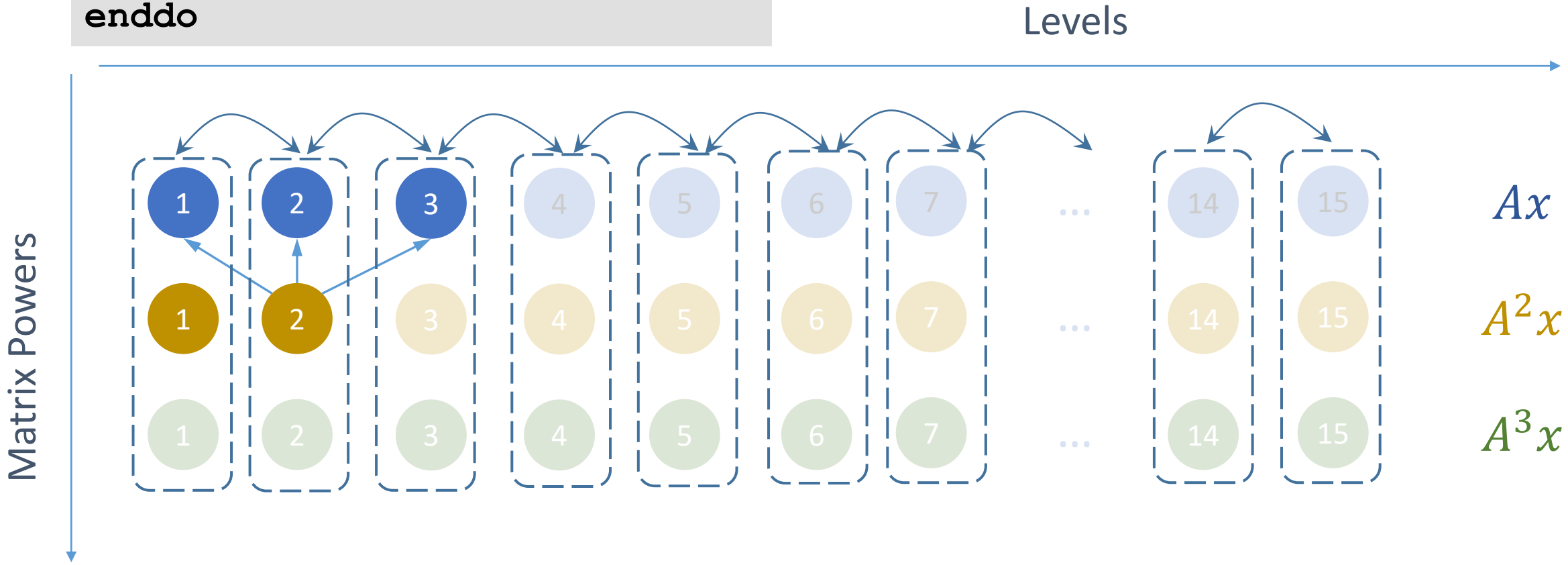
```
do k = 1, p
 y(:, k) = SpMV(A, y(:,k-1))
enddo
```



When updating level 1, indirect reads also go to level 2

RACE – Level traversal and matrix powers

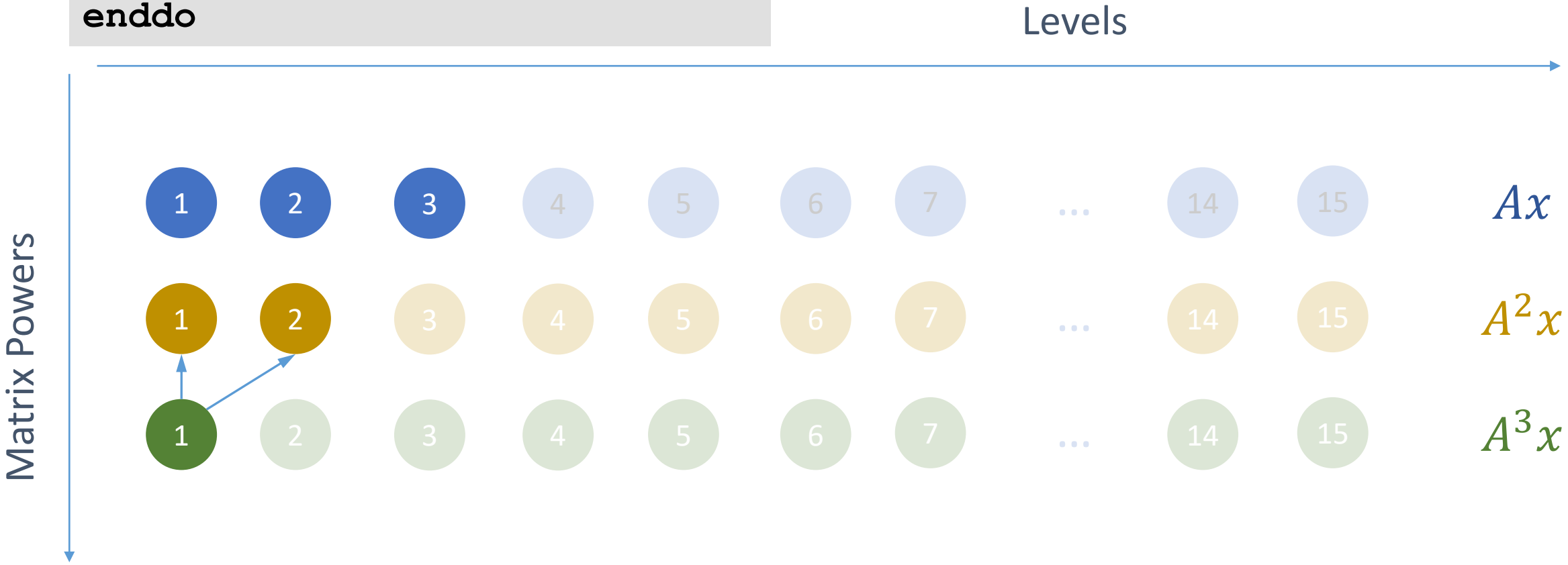
```
do k = 1, p
 y(:, k) = SpMV(A, y(:,k-1))
enddo
```



When updating level 2, indirect reads also go to levels 1 and 3

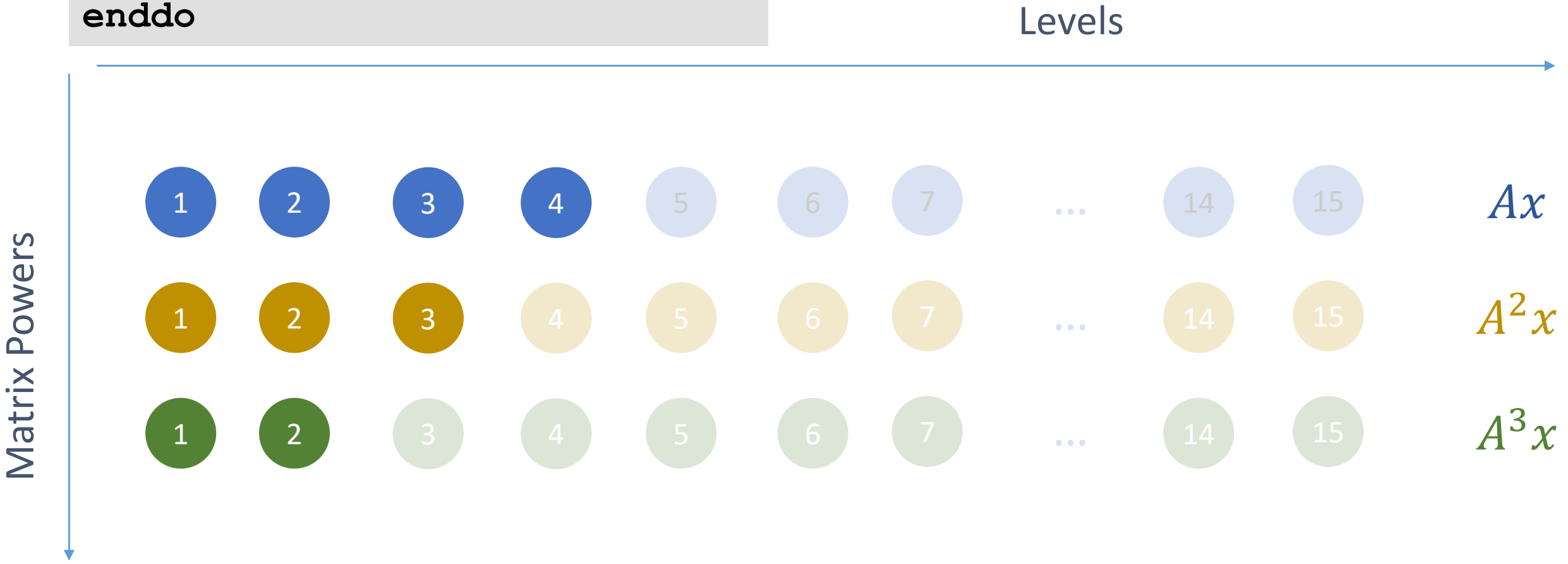
RACE – Level traversal and matrix powers

```
do k = 1, p
 y(:, k) = SpMV(A, y(:,k-1))
enddo
```



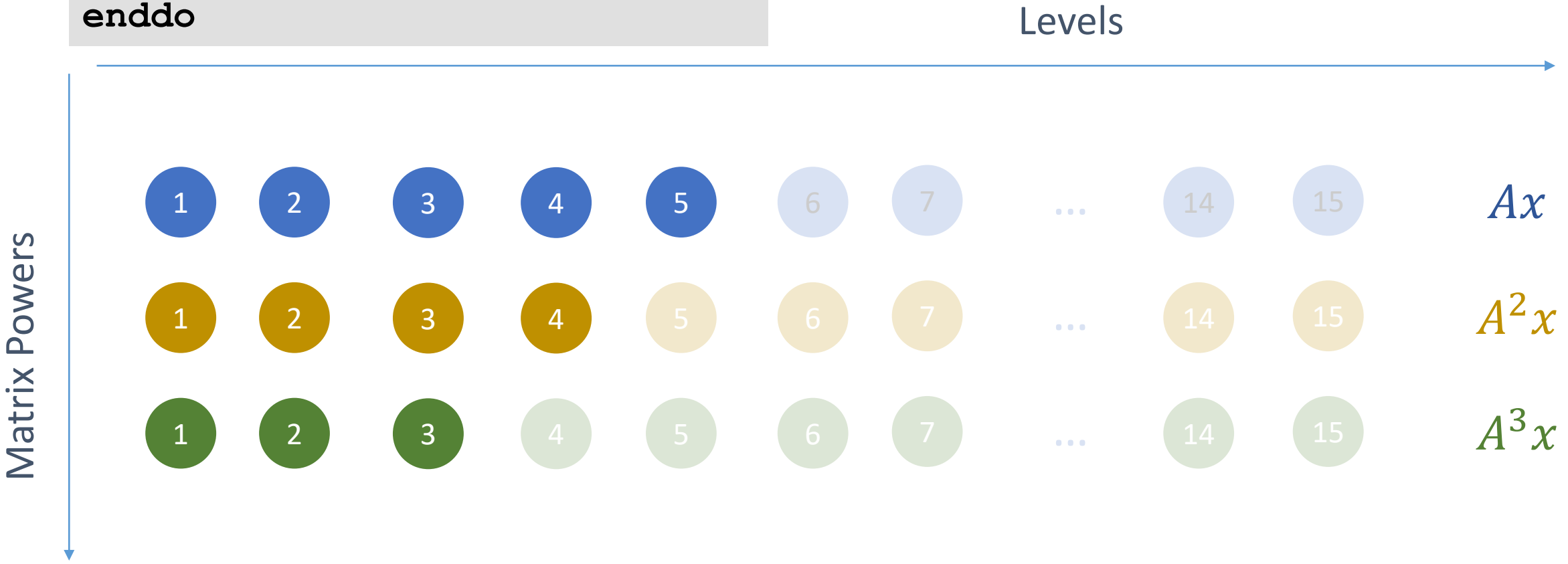
RACE – Level traversal and matrix powers

```
do k = 1, p
 y(:, k) = SpMV(A, y(:,k-1))
enddo
```



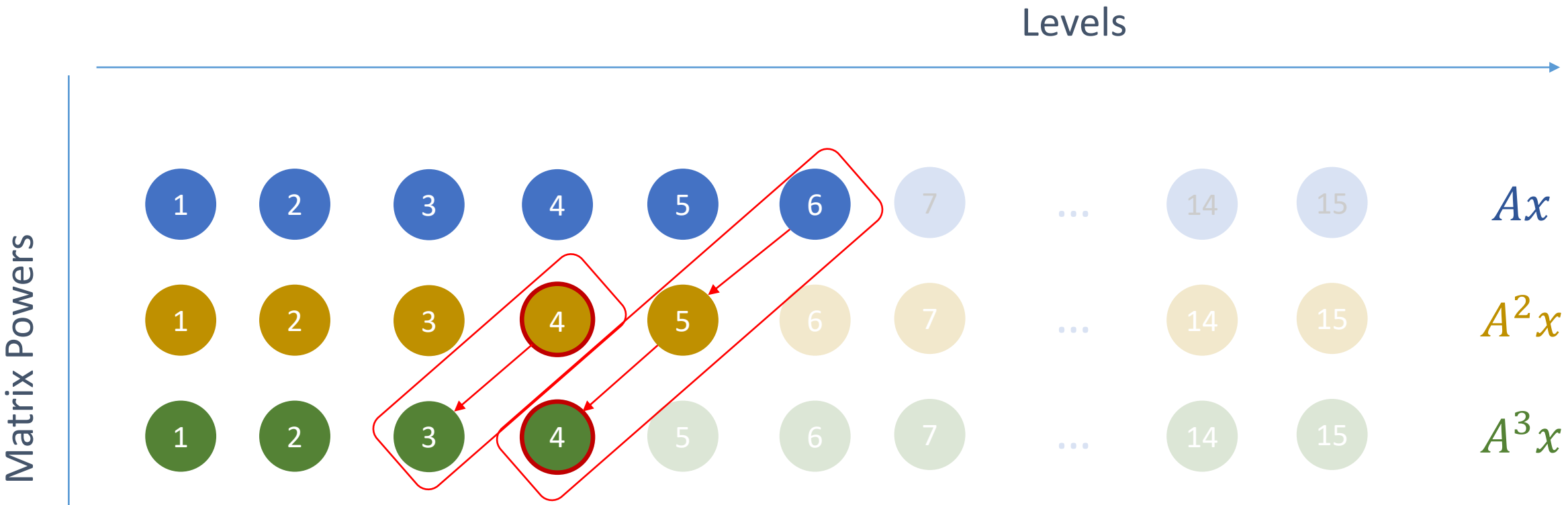
RACE – Level traversal and matrix powers

```
do k = 1, p
 y(:, k) = SpMV(A, y(:,k-1))
enddo
```



RACE: MPK implementation idea

RACE – Input parameters and its influence



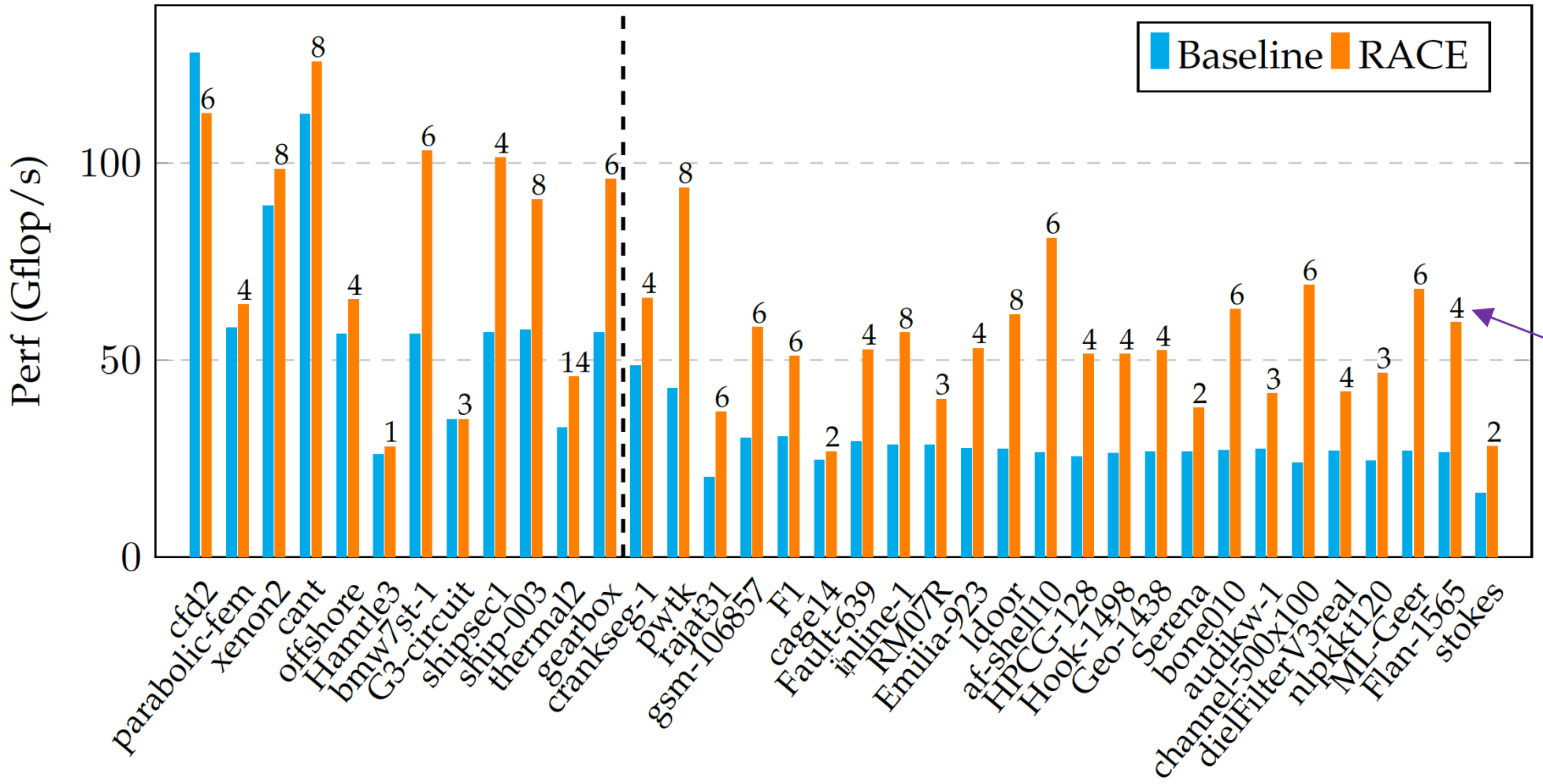
A is loaded only once if

$(p+1) \times N_{nz}(L) \times 12 \text{ bytes} < C$

$N_{nz}(L)$ – avg. non-zeros in a level

C – cache size

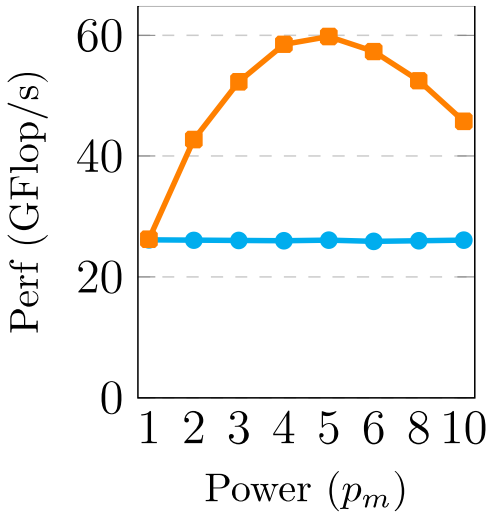
Matrix power kernel: Performance – Intel Ice Lake



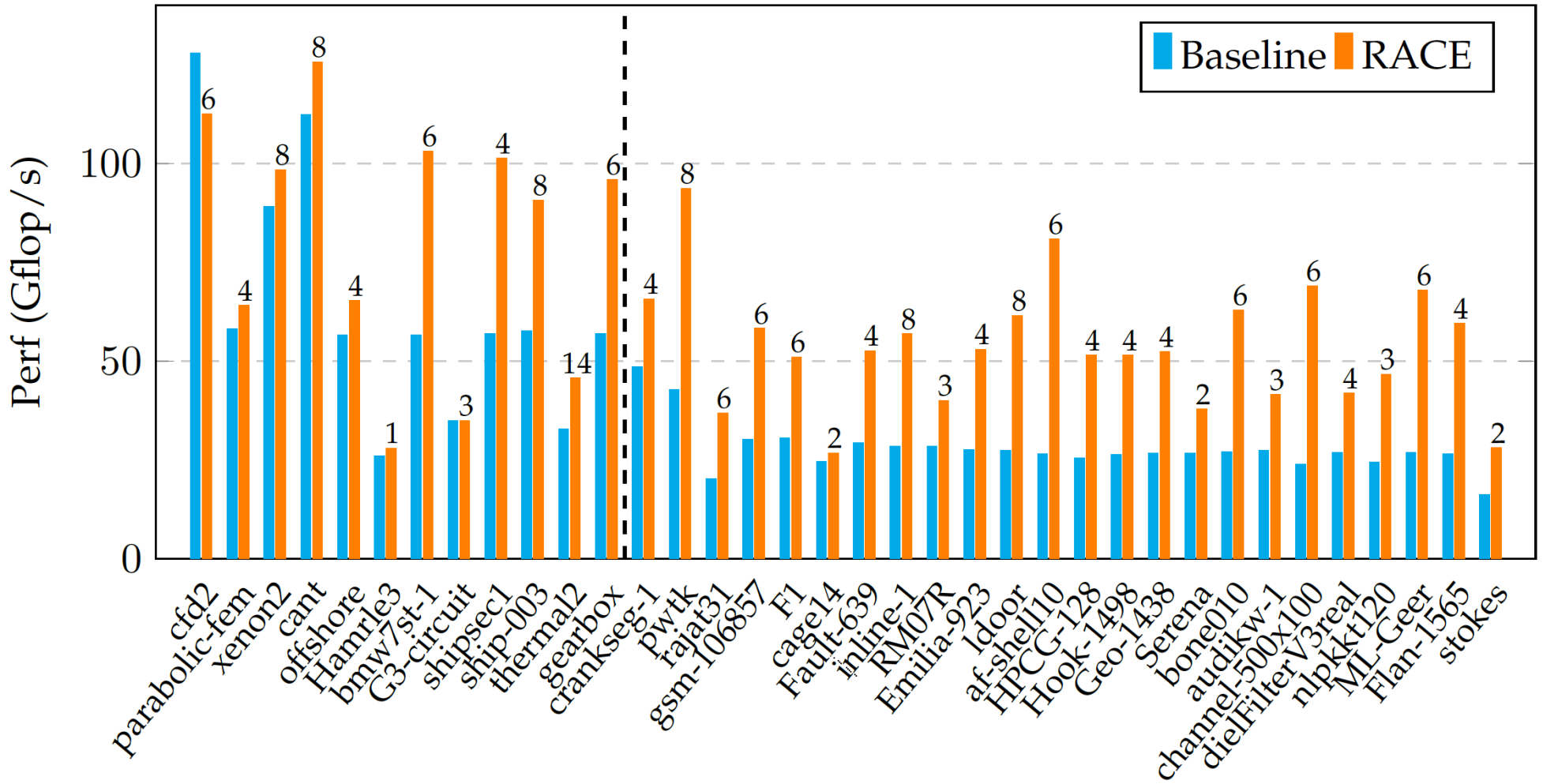
Intel Xeon Platinum 8368 (Ice Lake)

- 38 cores
- 104 MB cache (L2+L3)

Power value with maximum performance.

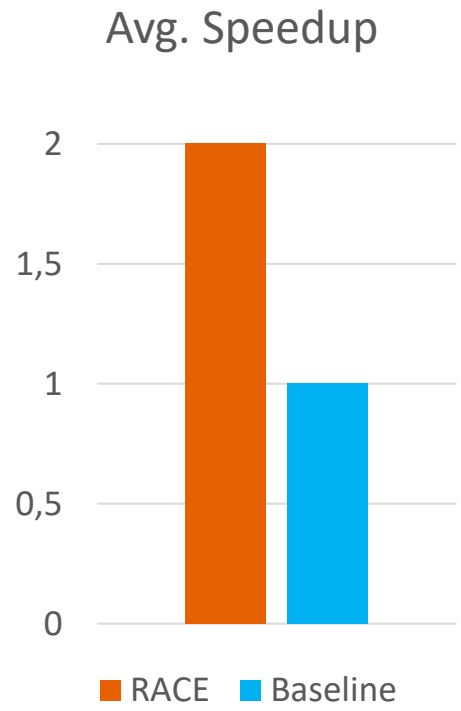


Matrix power kernel: Performance – Intel Ice Lake

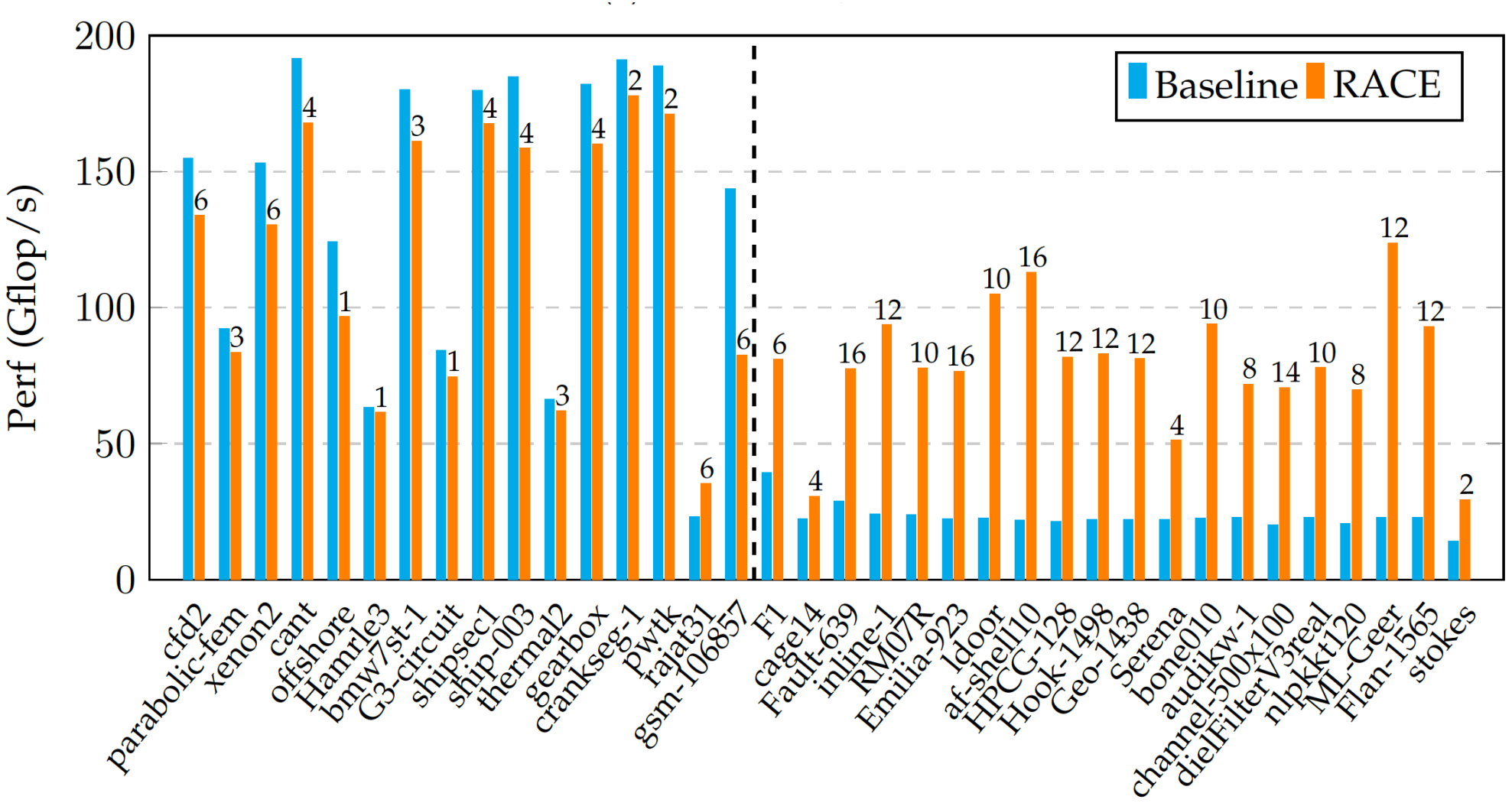


Intel Xeon Platinum 8368 (Ice Lake)

- 38 cores
- 104 MB cache (L2+L3)

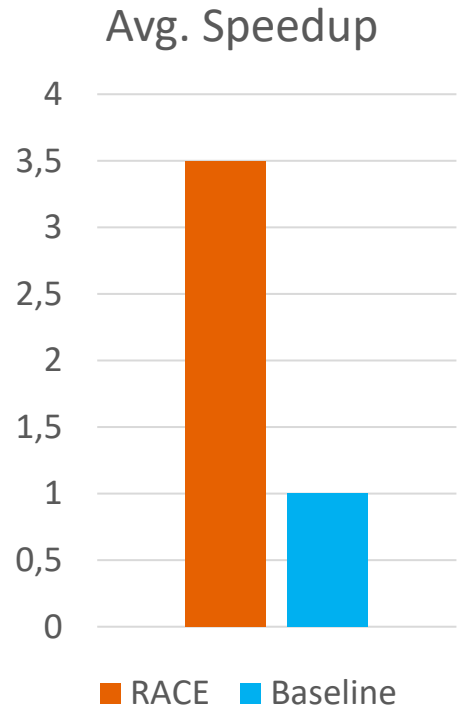


Matrix power kernel: Performance – AMD Rome



AMD EPYC 7662

- 64 cores
- 288 MB cache (L2+L3)

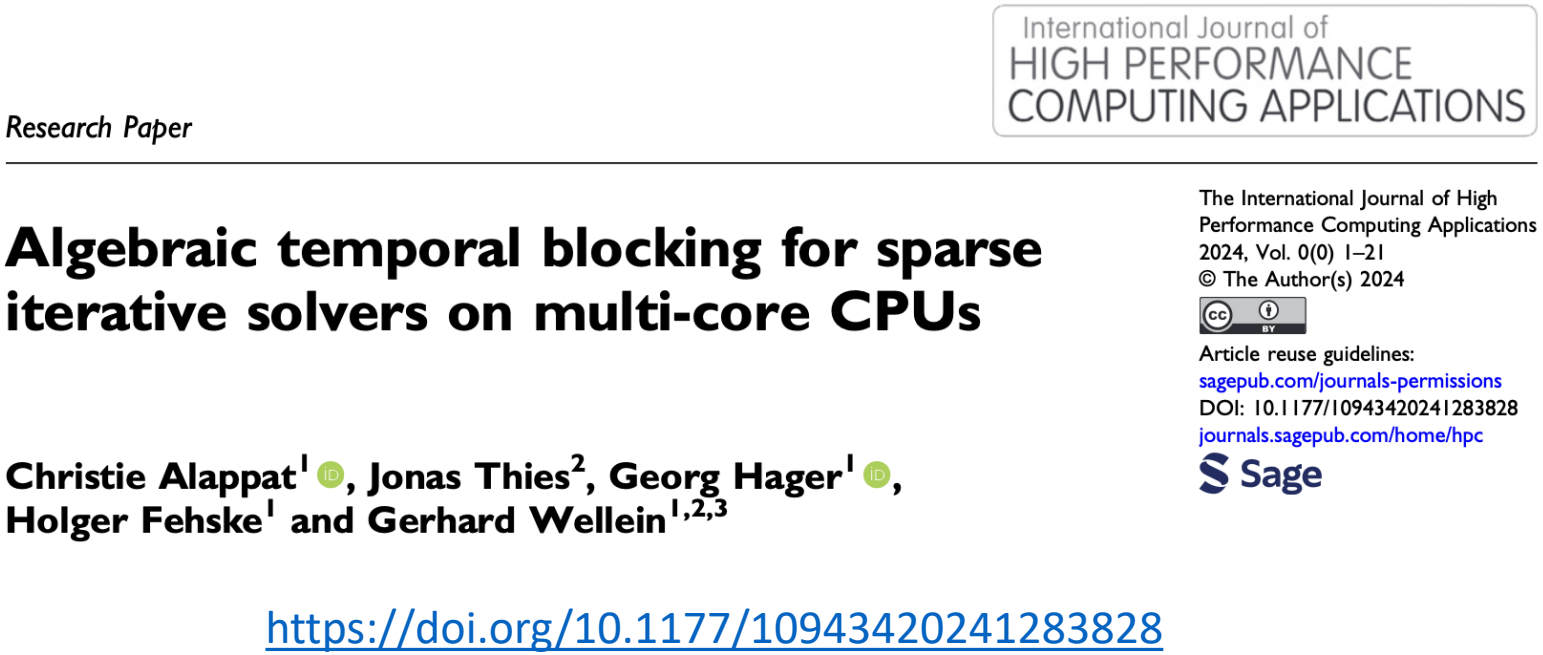


RACE - summary

- Inner kernel: OpenMP parallel standard SpMV routine
- Overhead: BFS & Set up of data structures (approx. ≤ 50 SpMVs)
- Parameters: Power (p_m), Available Cache Size, Max. recursion depth
- Cache size ←→ max. polynomial degree (p_m)
  - Larger caches → larger p_m → better performance
  - Polynomial degree higher than p_m → Computation in chunks of p_m
- No loss of accuracy!

RACE – MPK applications

- Exponential Integrators → Polynomial approximations
- s-step Krylov methods (CA-GMRES)
- Polynomial preconditioning
- Algebraic Multigrid smoothers



International Journal of HIGH PERFORMANCE COMPUTING APPLICATIONS

Research Paper

Algebraic temporal blocking for sparse iterative solvers on multi-core CPUs

Christie Alappat[1], Jonas Thies[2], Georg Hager[1], Holger Fehske[1] and Gerhard Wellein[1,2,3]

The International Journal of High Performance Computing Applications
2024, Vol. 0(0) 1–21

DOI: 10.1177/10943420241283828
journals.sagepub.com/home/hpc
Sage

https://doi.org/10.1177/10943420241283828

Hands-On:

MPK with RACE

Using RACE

```
# include <RACE/interface.h>

RACE::dist k = RACE::POWER;
Nt = omp_get_num_threads();
RACE::Interface race (Nr, Nt, k, rowPtr, col );

//power value; here 4
int pm = 4 ;
//cache size in bytes; here 30 MB
double C = 30*1024*1024;
//perform pre-processing, find levels
race.RACEColor(pm, C);

int *perm, * invPerm , permLen=Nr;
race.getPerm(&perm, &permLen) ;
race.getInvPerm(&invPerm, &permLen) ;
//permute matrix and vector data structures
permute (perm, invPerm) ;
```

Pre-processing

```
struct functionArg
{
  //user-defined struct for input and output
  //arguments of the call-back function
  int Nr;
  . . .
};

//user-defined call-back function
void foo(int row_s, int row_e, int pow, void * voidArg)
{
  functionArg * arg = (functionArg *) voidArg;
  . . .
}

functionArg* args = new functionArg;
//fill args
args->Nr = 1000;
. . .

void* voidArgs = (void*) args;
int foo_id = race.registerFunction(&foo, voidArgs, pm);
race.executeFunction(foo_id);
```

Processing

Neumann polynomial apply

$$w = (I - L)^k A (I - U)^k v$$

$$t_1 = (I - U)^k v$$ Cache blocking

$$t_2 = A t_1$$

$$w = (I - L)^k t_2$$ Cache blocking

Can we do better?

Neumann polynomial apply

$$w = (I - L)^k A (I - U)^k v$$

$$t_1 = (I - U)^k v$$

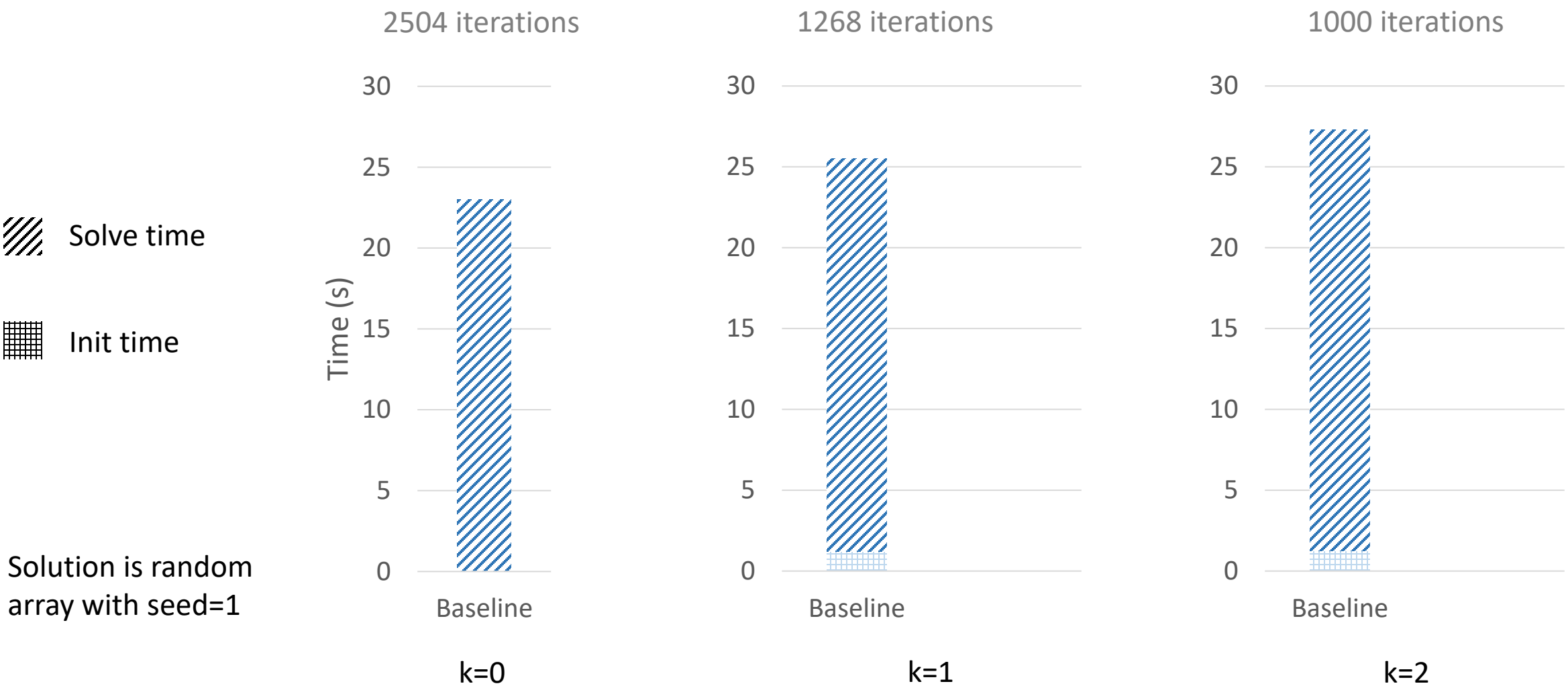
$$t_2 = A t_1 = (L + U) t_1$$

$$w = (I - L)^k t_2$$

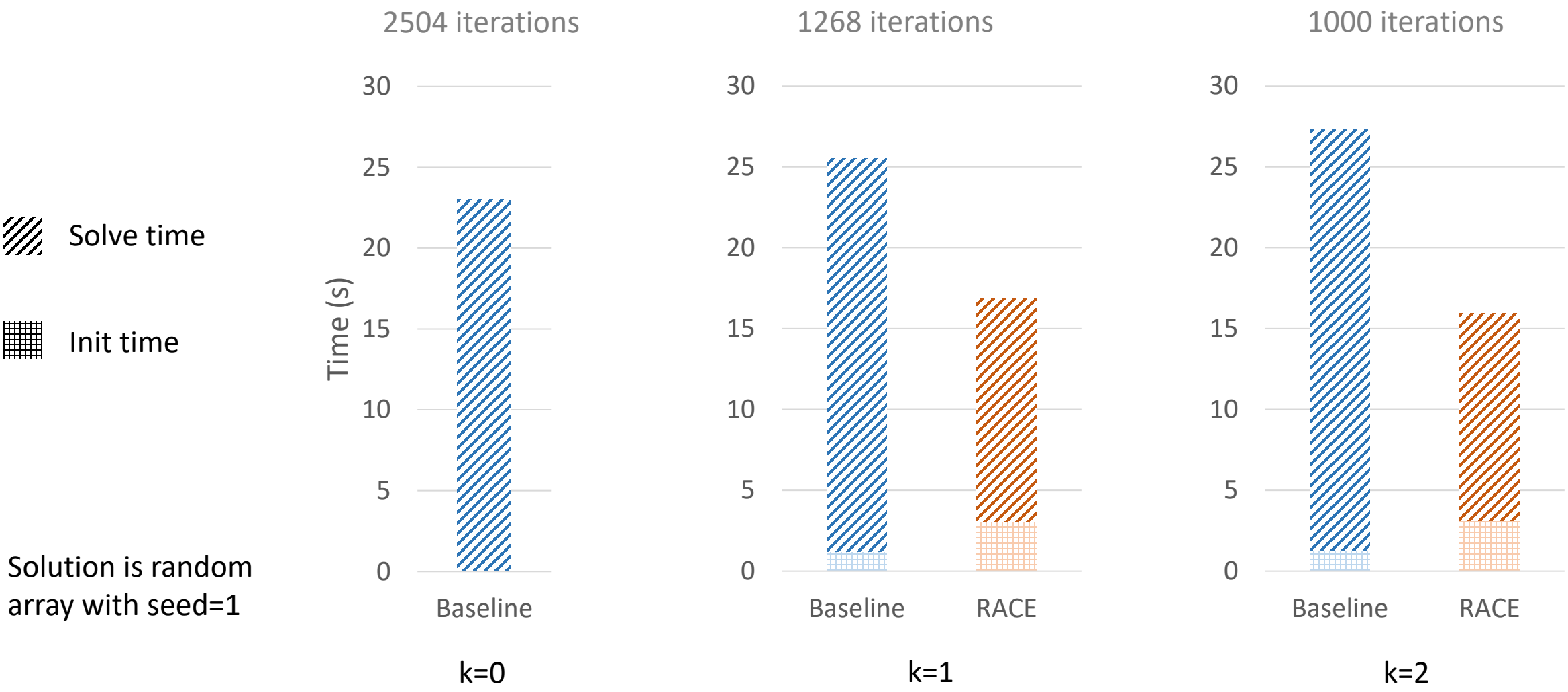
Cache blocking

Total power = $2k + 1$

Time to solve Laplace2000x2000 to 1e-3 tolerance on 1 NUMA domain (18c) of Intel Ice lake (Fritz)



Time to solve Laplace2000x2000 to 1e-3 tolerance on 1 NUMA domain (18c) of Intel Ice lake (Fritz)



RACE: Backup

MPK – existing caching approaches

- Huber et al.: Graph-based higher-order time integration of PDEs[1]
  - "Geometrical approach" based on matrix bandwidth
  - Works for 2D stencil matrices → Runs into problem for 3D and/or unstructured matrices

- Mohiyuddin et al.: Minimizing communication in sparse matrix solvers[2]
  - "Domain decomposition" of underlying graph
  - Requires "ghosting" → Indirect accesses or redundant copies of the matrix entries → Scalability!!

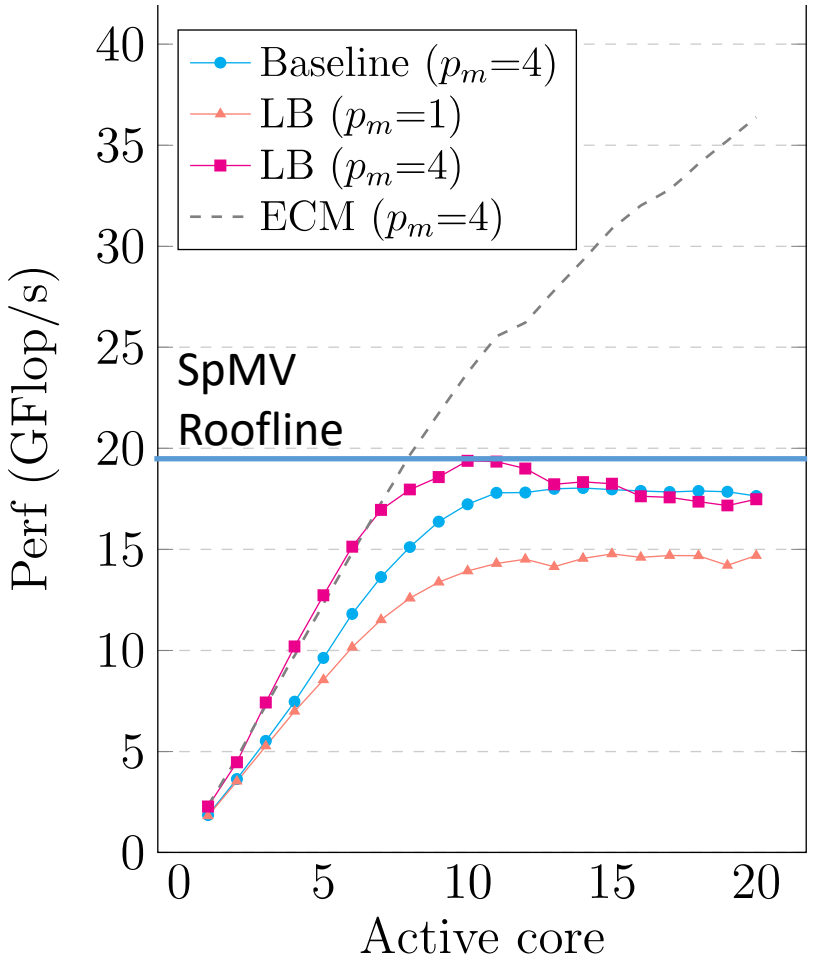
→ Exploit level structure in RACE for cache blocking!

RACE

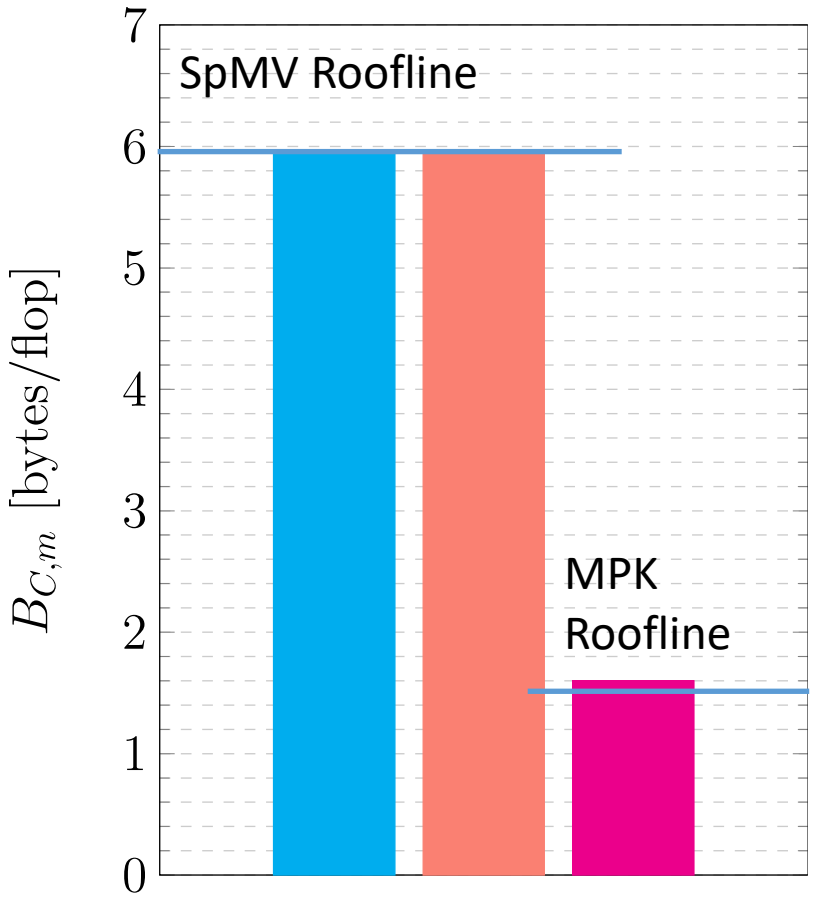
[1]Huber et al., 2021. Graph-based multi-core higher-order time integration of linear autonomous partial differential equations. J. Comput. Sci. DOI:10.1016/j.jocs.2021.101349

[2]Mohiyuddin et al., 2009. Minimizing communication in sparse matrix solvers. In Proceedings of the SC'09. DOI:10.1145/1654059.1654096

RACE MPK – First Implementation



Performance



Memory traffic

Intel Xeon Gold 6248

- 1 Socket (20c)

pwtk matrix

- N_r = 217,918
- N_{nz} = 11,634,424

RACE MPK – Performance Problem Identified

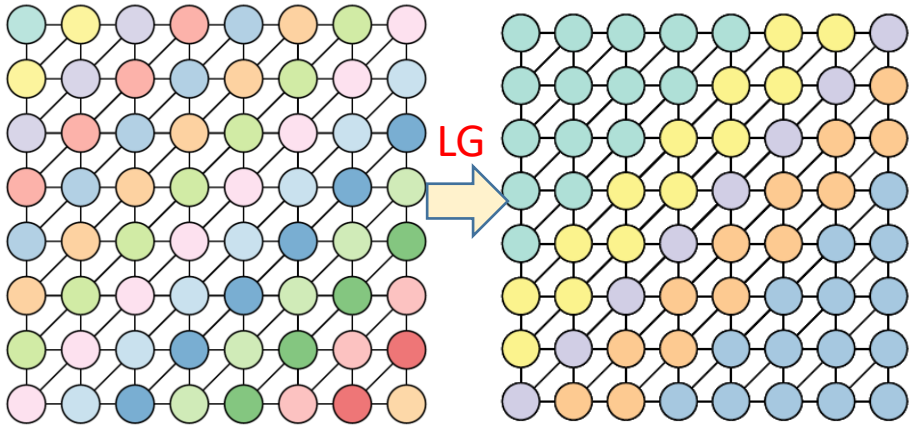
- Scheme seems to work (reduces data traffic) – at least for **pwtk**
- But: Performance ☹ !!!!

- Analysis of hardware performance counters (LIKWID) for **pwtk** matrix:
  `INSTR_RETIRED_ANY` up 2x for level based SpMV!

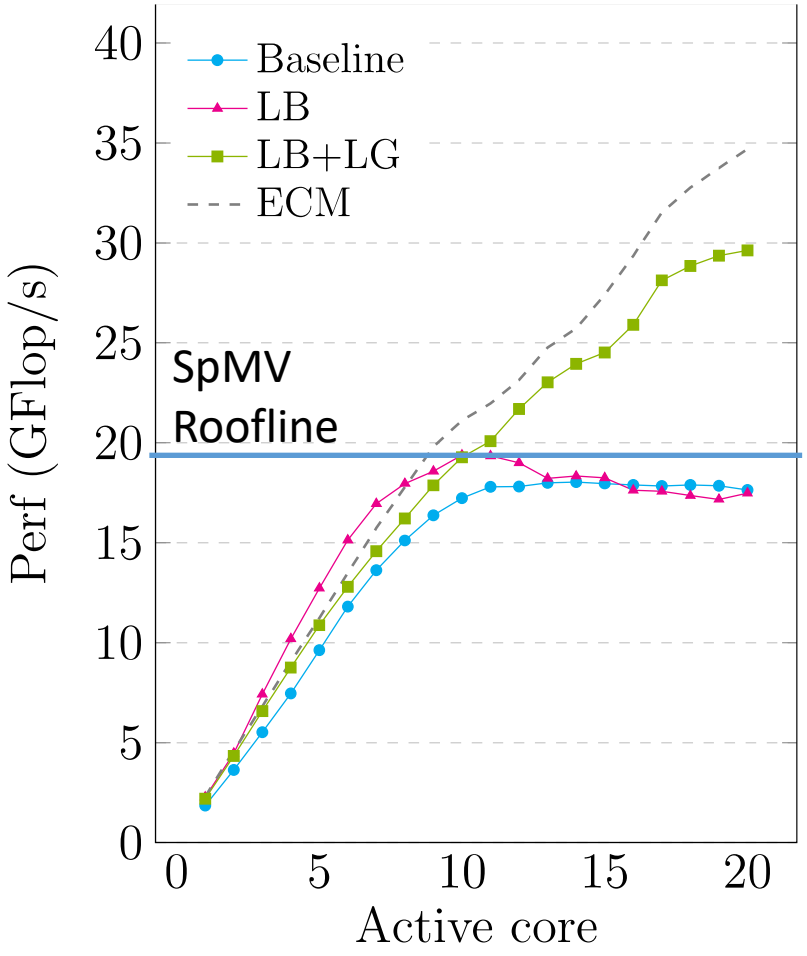
  → Frequent thread syncronisations!

  Reason: After each level threads sync!

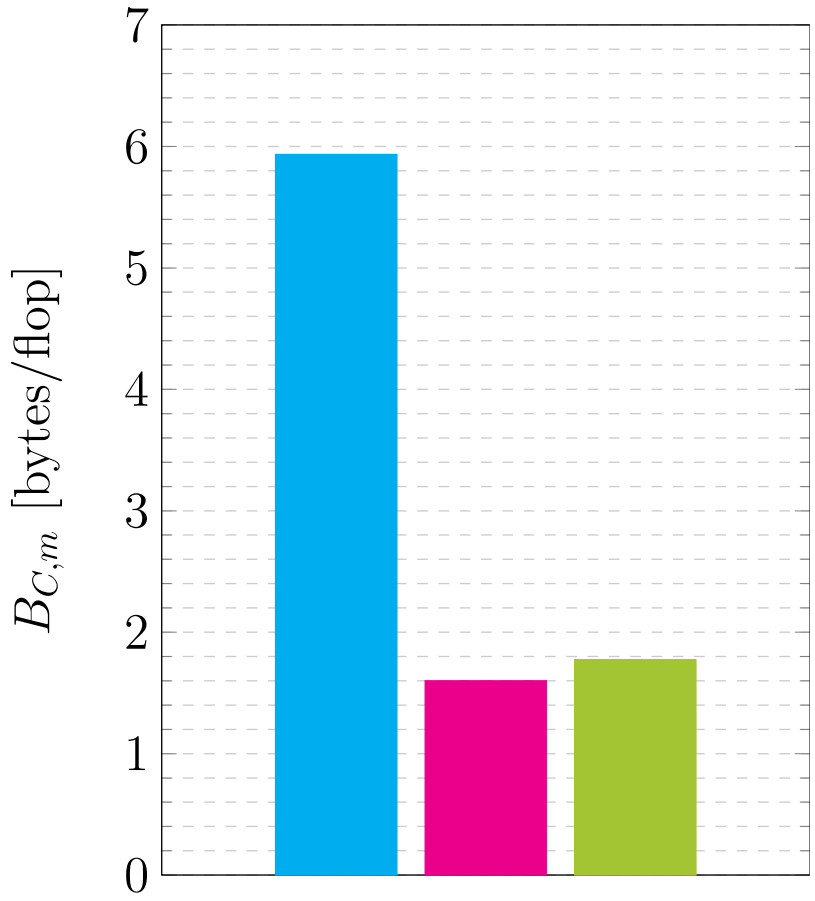
  Measures:
  → Reduce #levels by level aggregation („LG“)
  → Global sync. replaced by point-to-point sync. („p2p“)



RACE MPK – LG optimization



Performance



Memory traffic

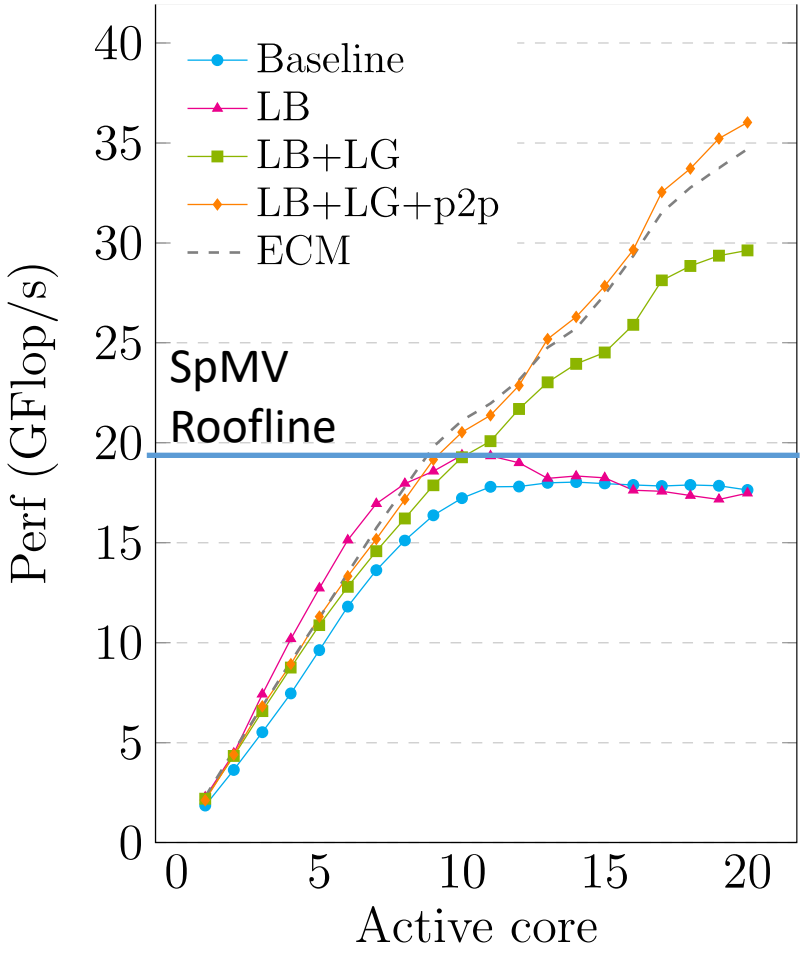
Intel Xeon Gold 6248

- 1 Socket (20c)

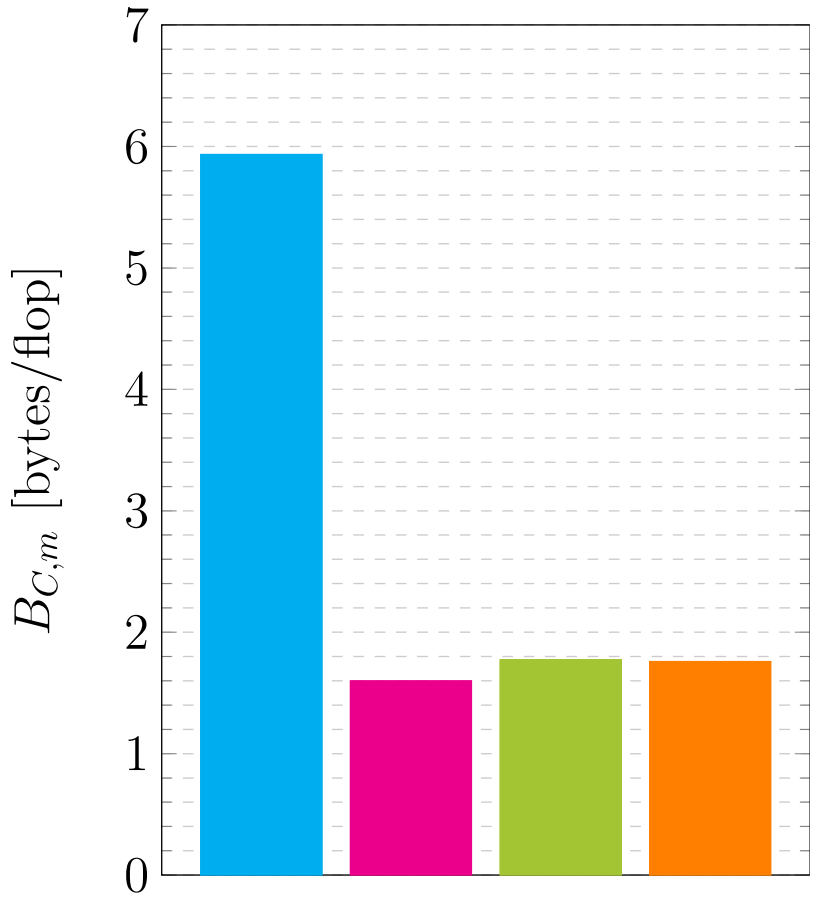
pwtk matrix

- $N_r = 217{,}918$
- $N_{nz} = 11{,}634{,}424$

RACE MPK – LG+p2p optimization



Performance



Memory traffic

Intel Xeon Gold 6248

- 1 Socket (20c)

pwtk matrix

- N_r = 217,918
- N_{nz} = 11,634,424

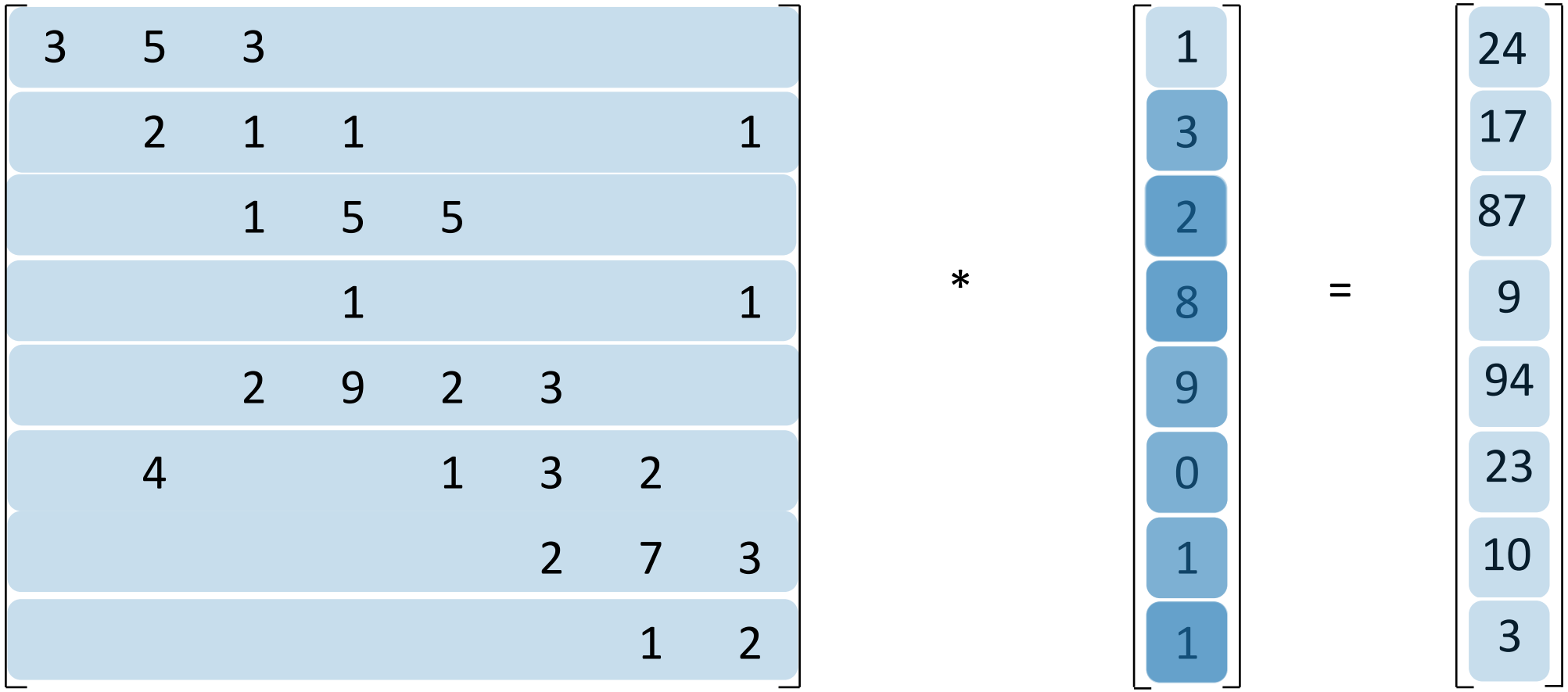
Outlook: Distributed-Memory SpMV

Slides courtesy of Dane Lacey, NHR@FAU

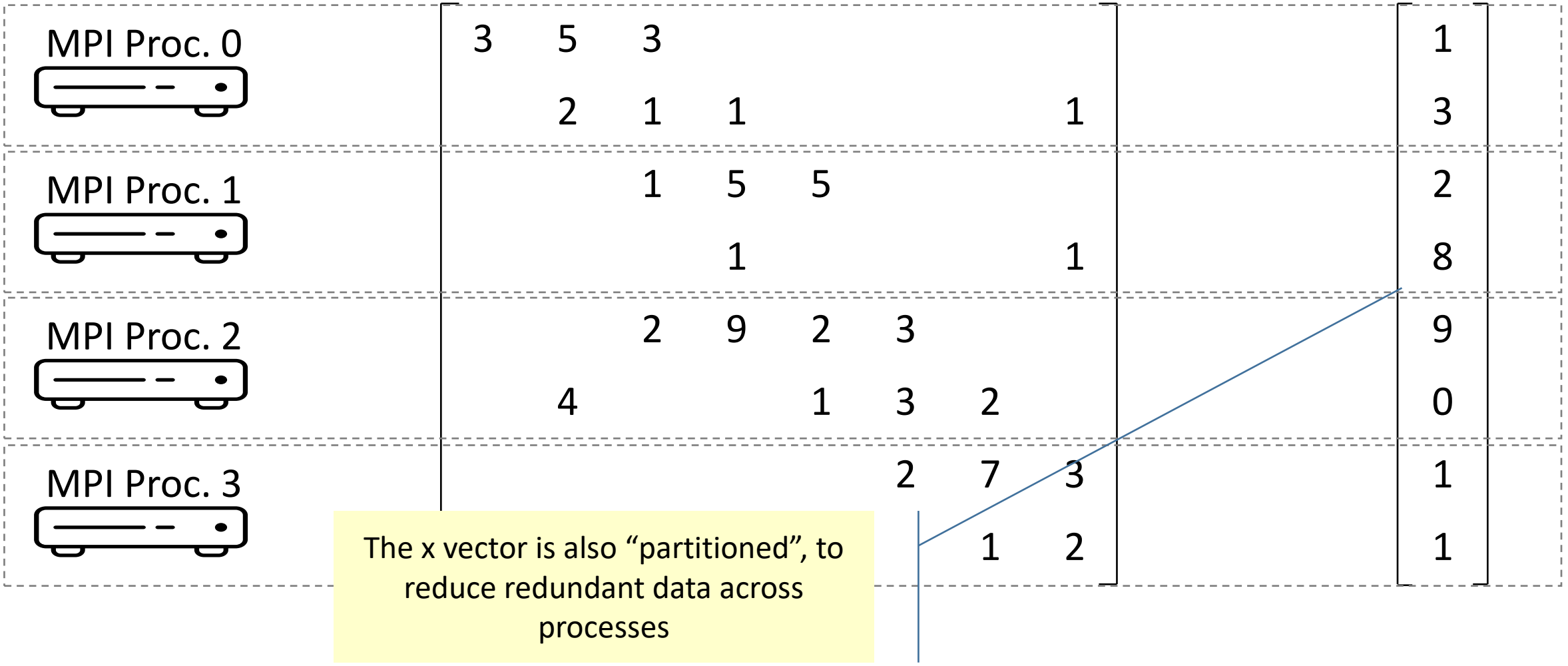
SpMV Example

$$A = \begin{bmatrix} 3 & 5 & 3 & & & & & \\ & 2 & 1 & 1 & & & & 1 \\ & & 1 & 5 & 5 & & & \\ & & & 1 & & & & 1 \\ & & 2 & 9 & 2 & 3 & & \\ & 4 & & & 1 & 3 & 2 & \\ & & & & & 2 & 7 & 3 \\ & & & & & & 1 & 2 \end{bmatrix} \quad x = \begin{bmatrix} 1 \\ 3 \\ 2 \\ 8 \\ 9 \\ 0 \\ 1 \\ 1 \end{bmatrix} \quad y = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix}$$

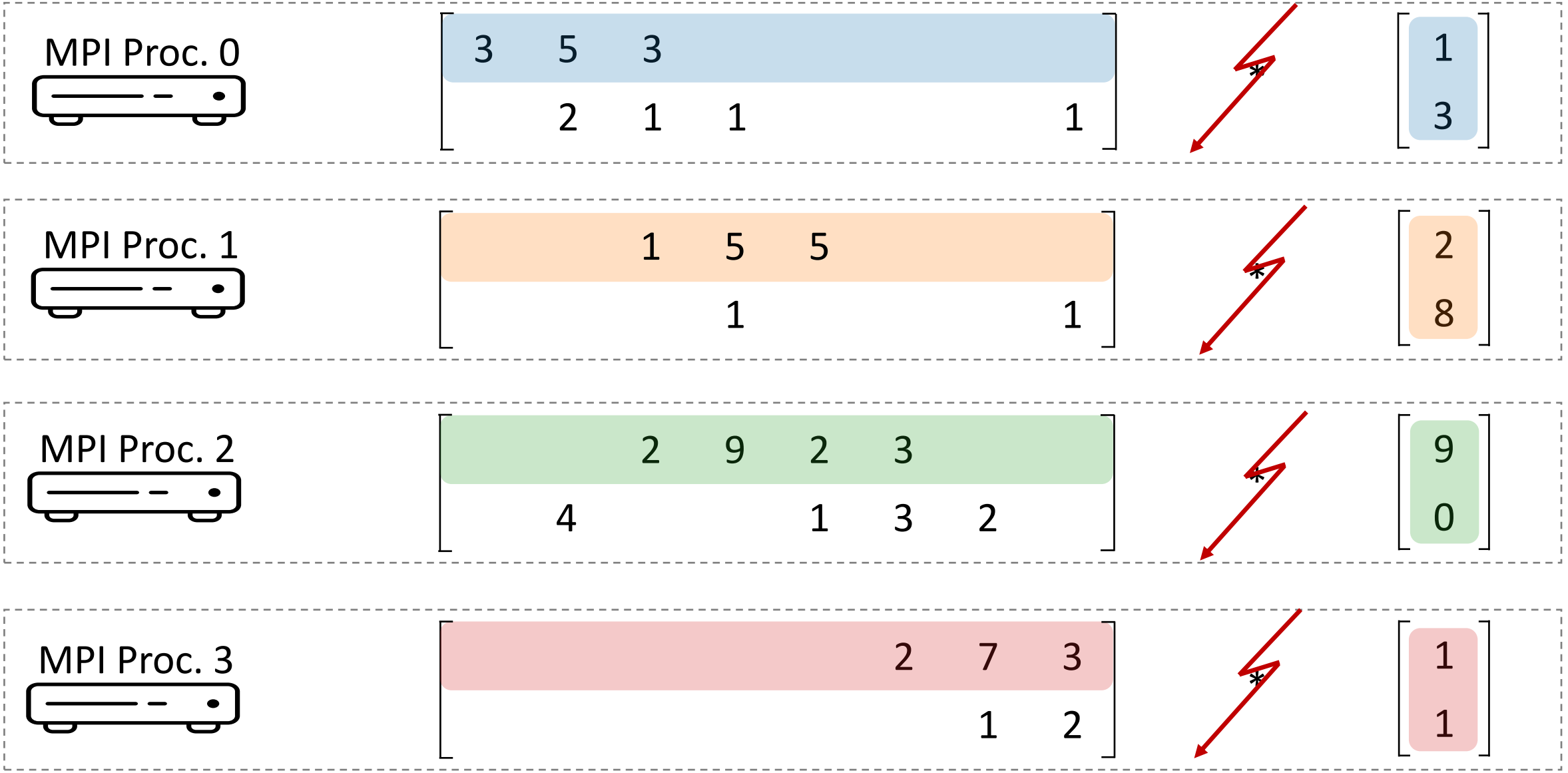
SpMV Example



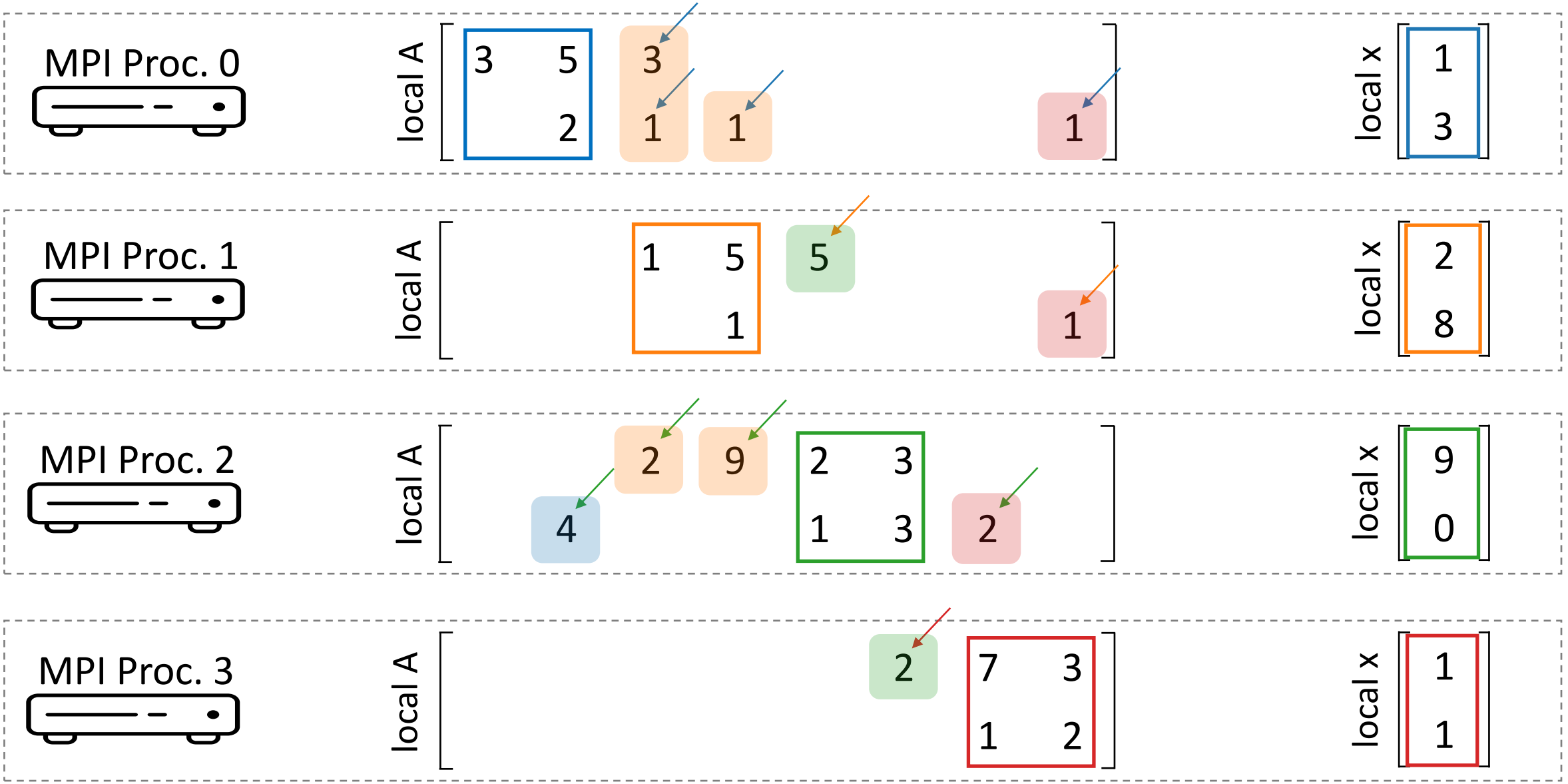
Distributed SpMV Example



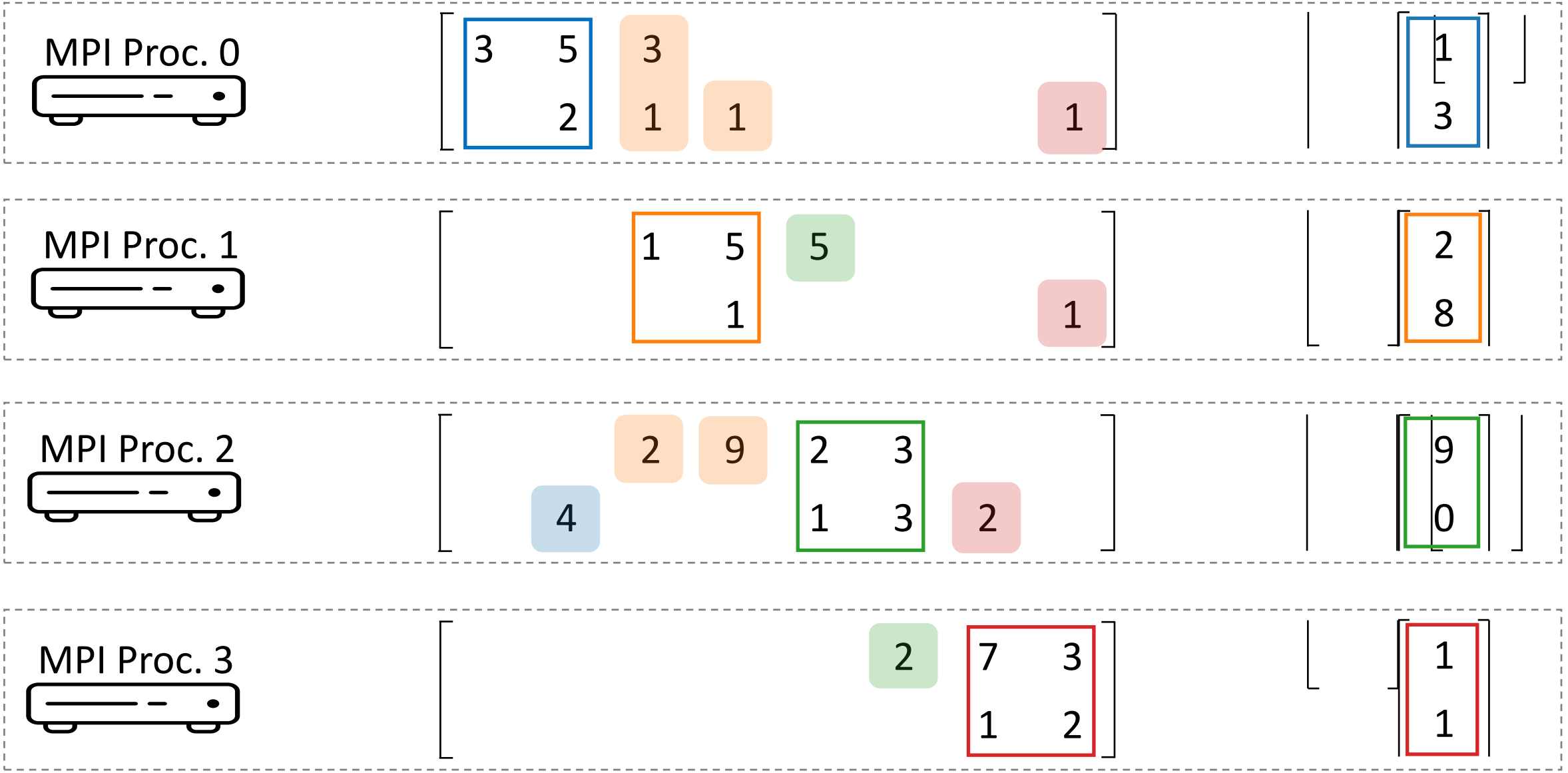
Distributed SpMV Example



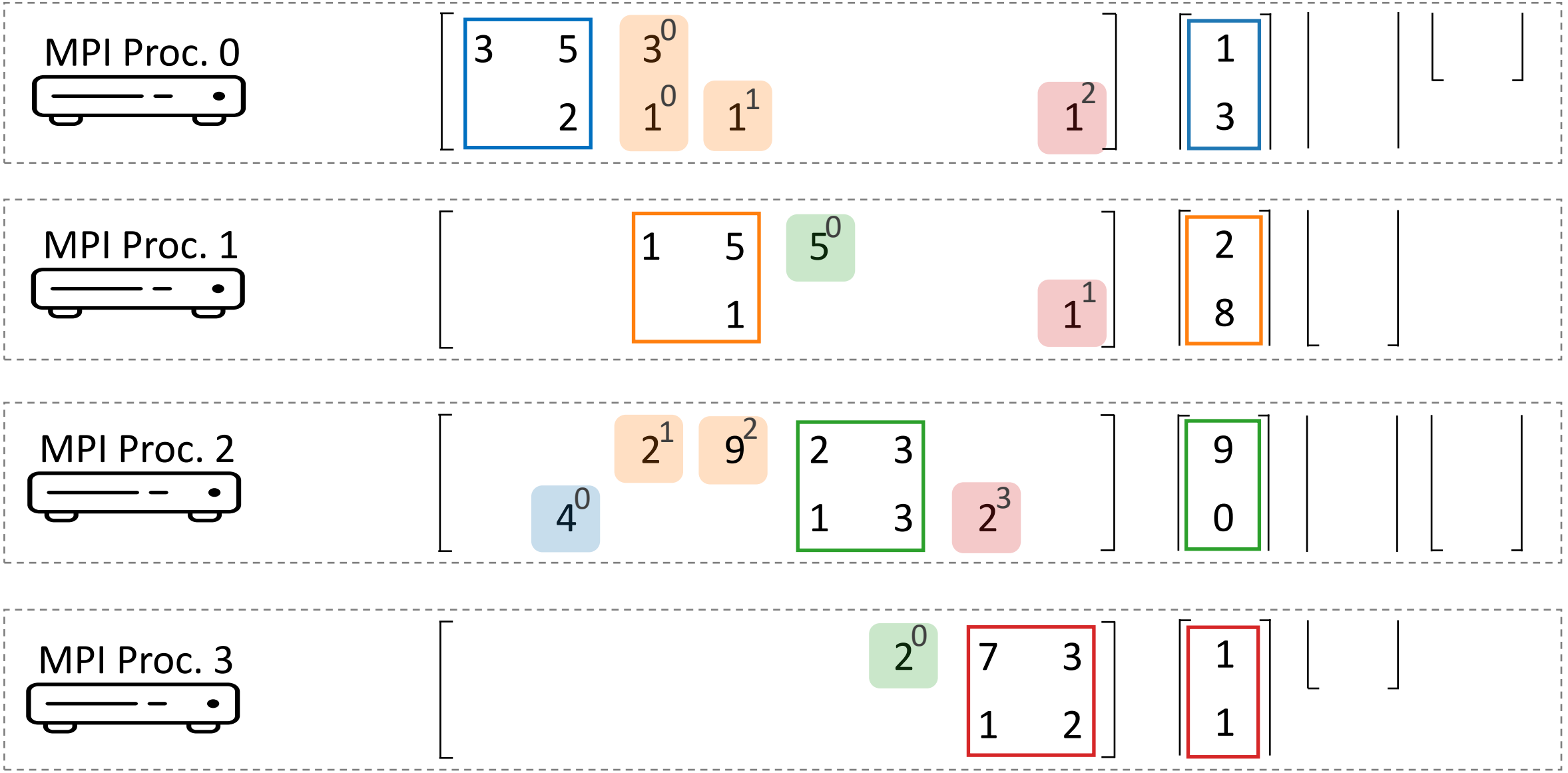
Distributed SpMV Example



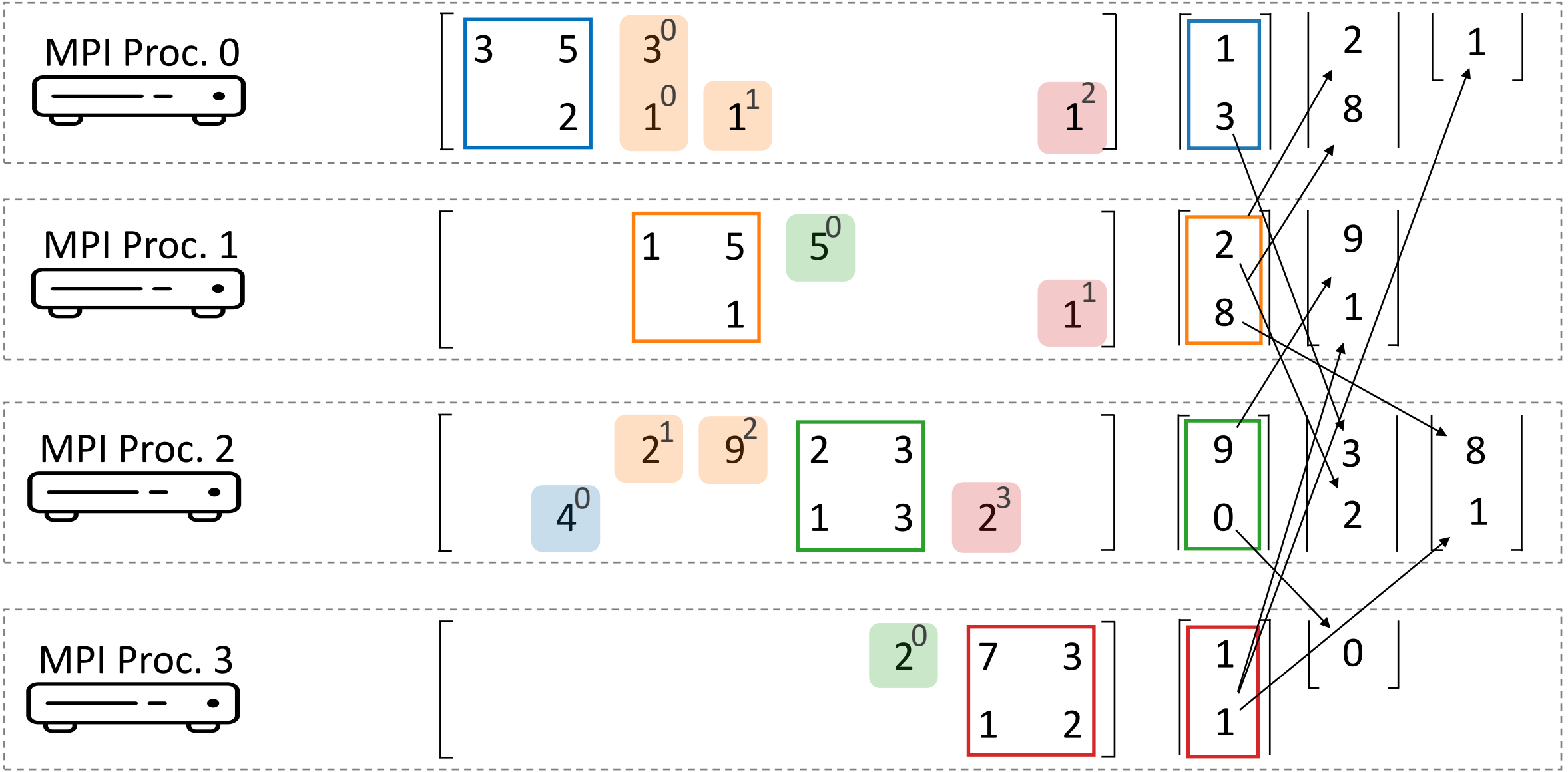
Distributed SpMV Example



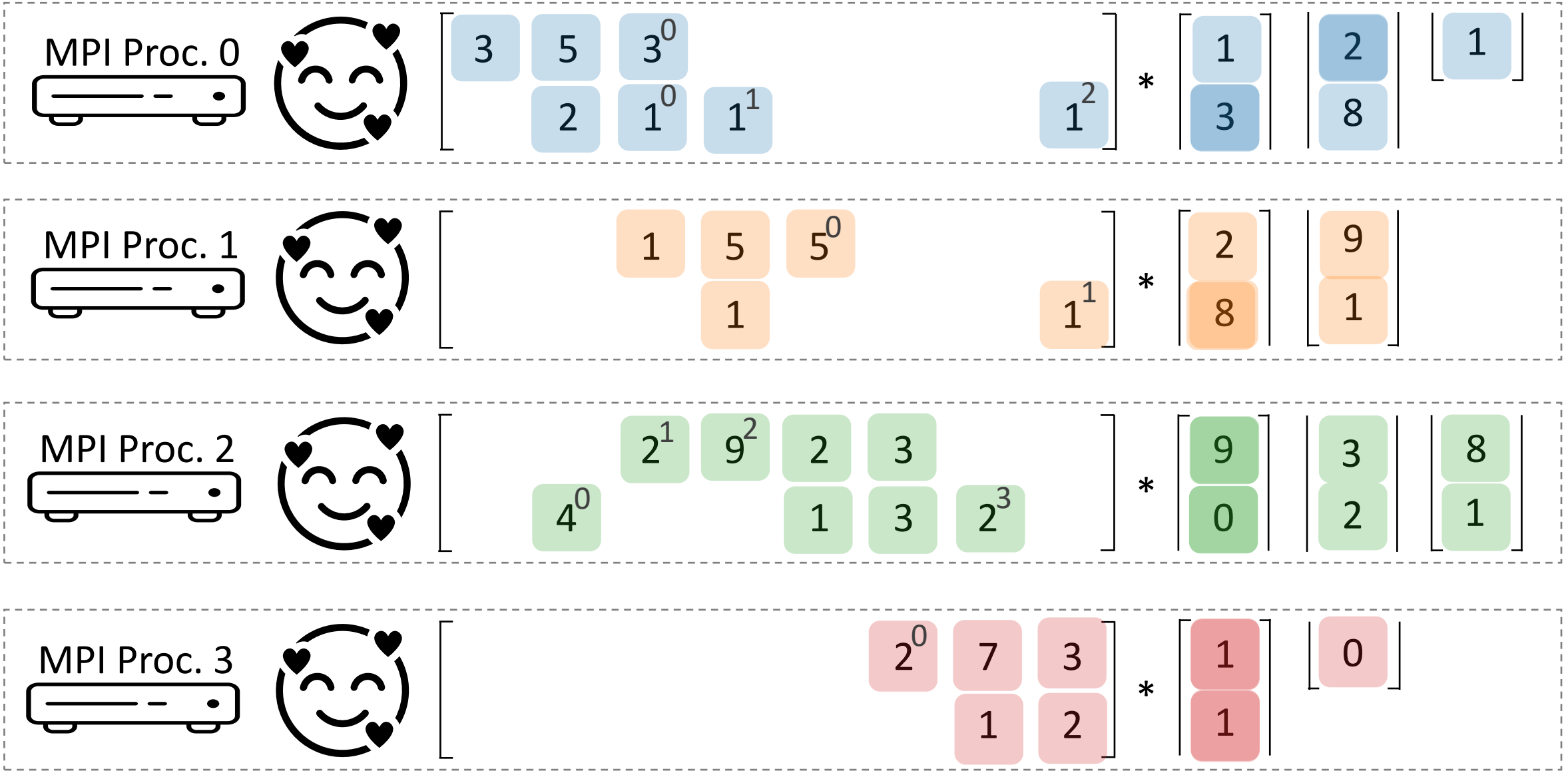
Distributed SpMV Example



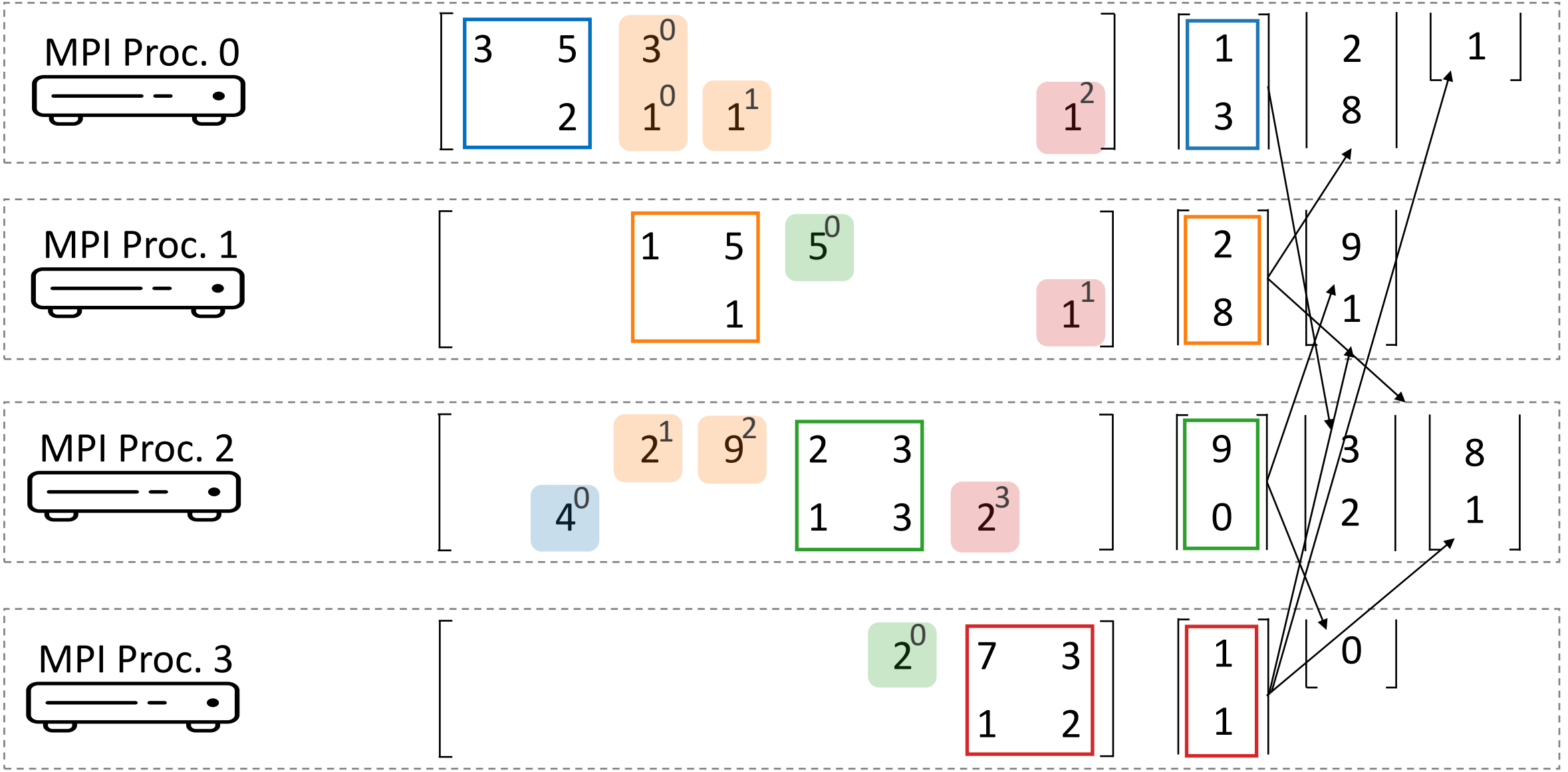
Distributed SpMV Example



Distributed SpMV Example

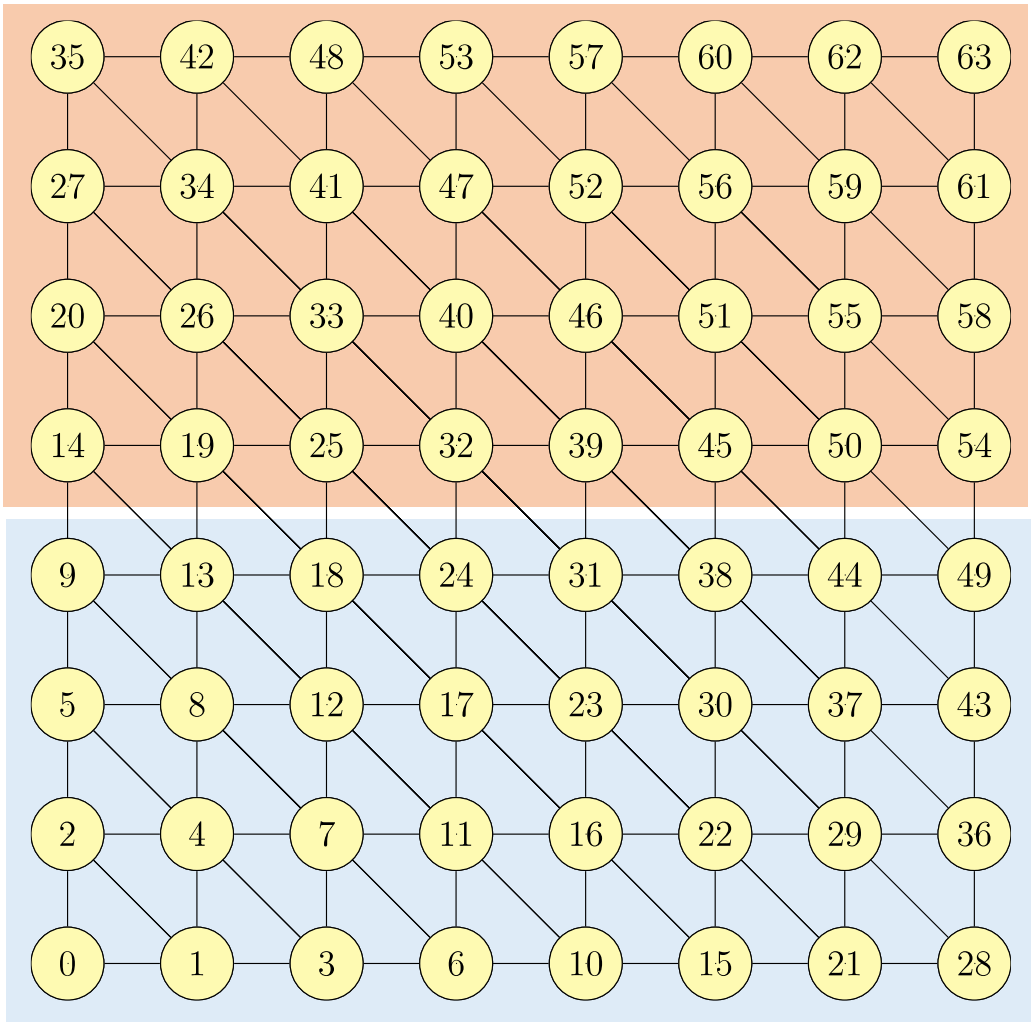


Distributed SpMV Example



Outlook:
Cache-Blocking Distributed-Memory MPK

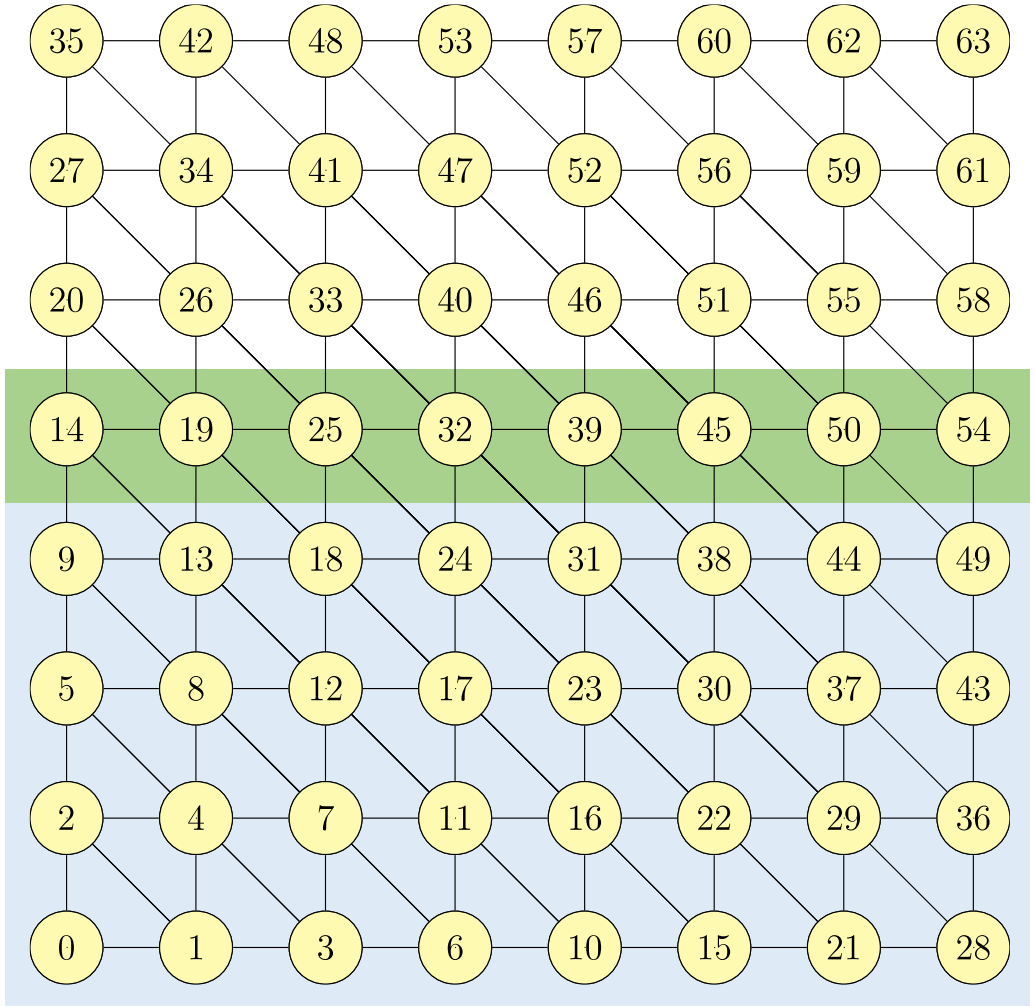
Distributed MPK



Proc 1

Proc 0

Distributed MPK

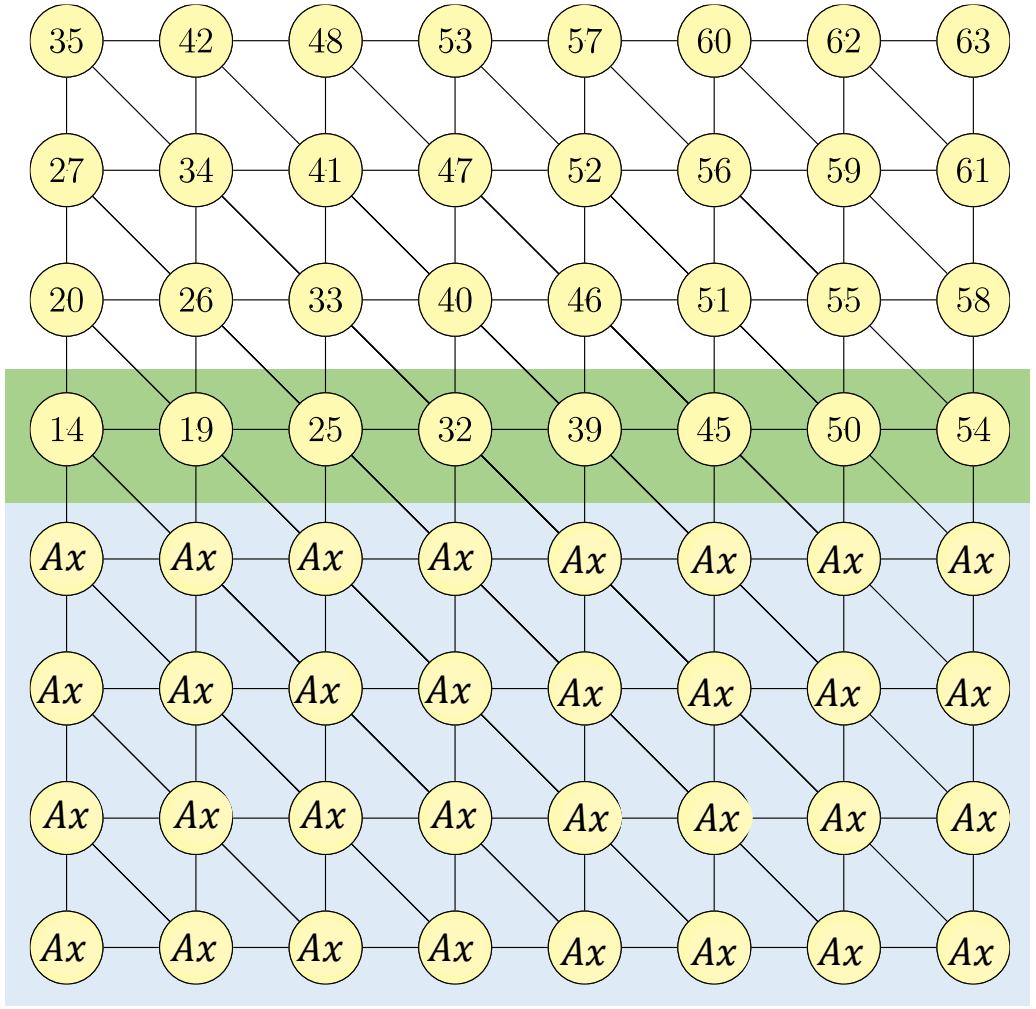


Computing Ax on Proc 0 requires neighbors of Proc 0.

How about computing $A^p x$?

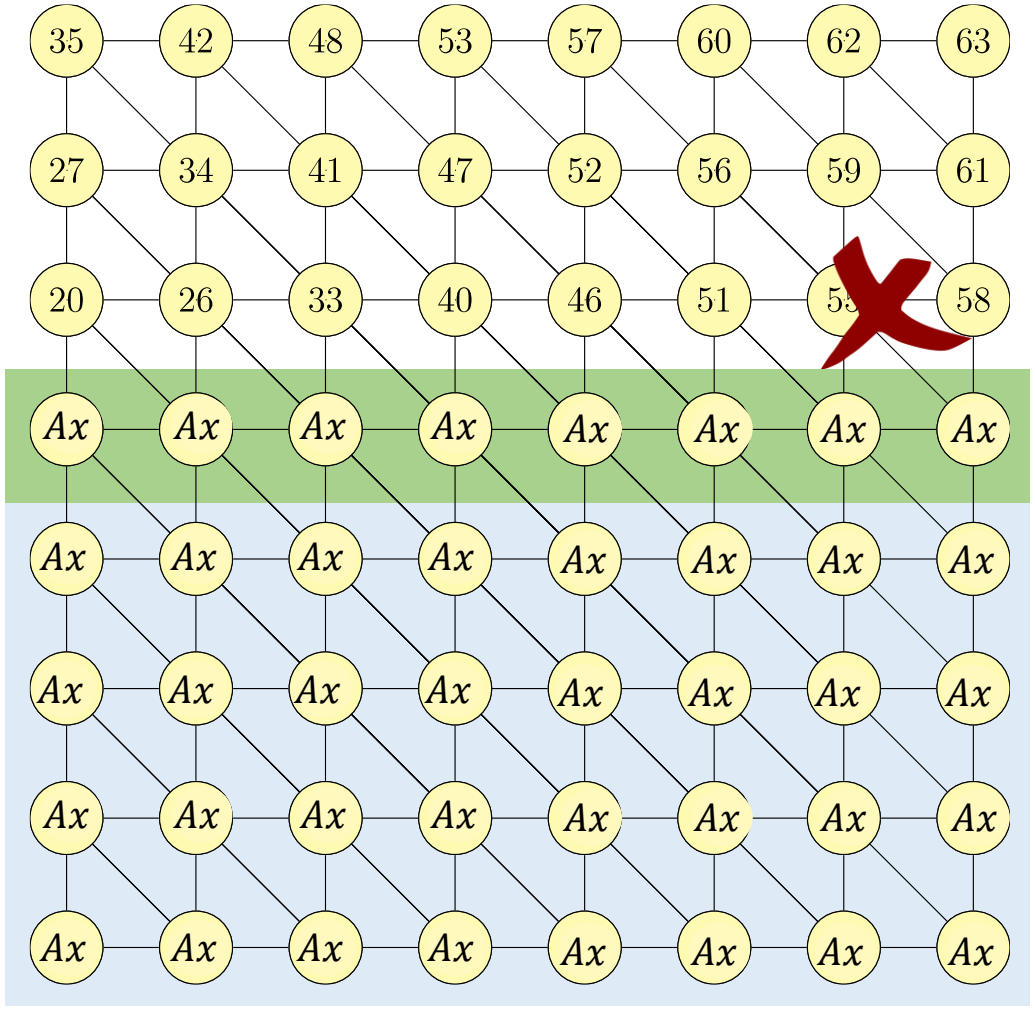
Proc 0

Distributed MPK



Proc 0

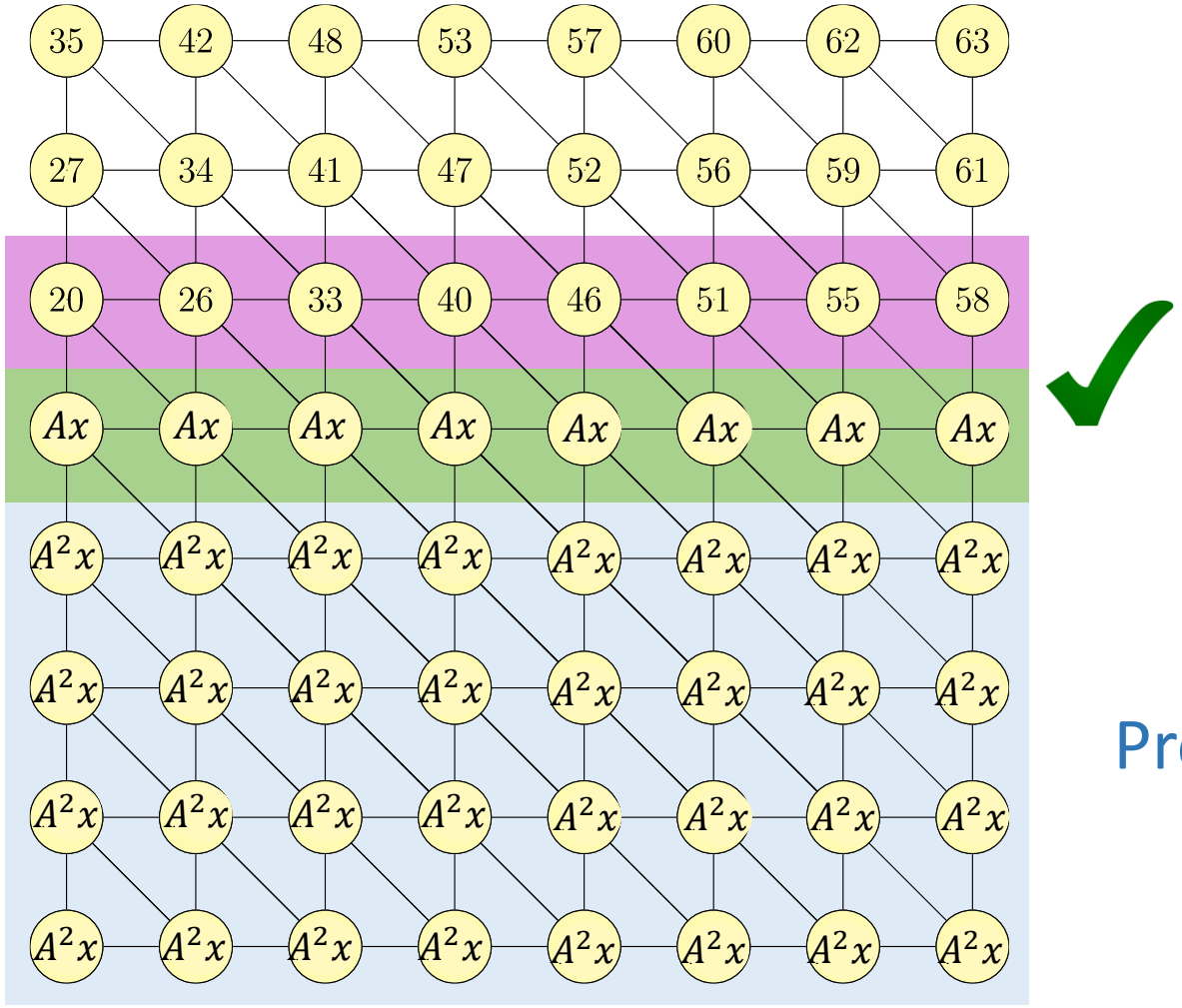
Distributed MPK



To compute A^2x we need Ax on the neighbors.

Proc 0

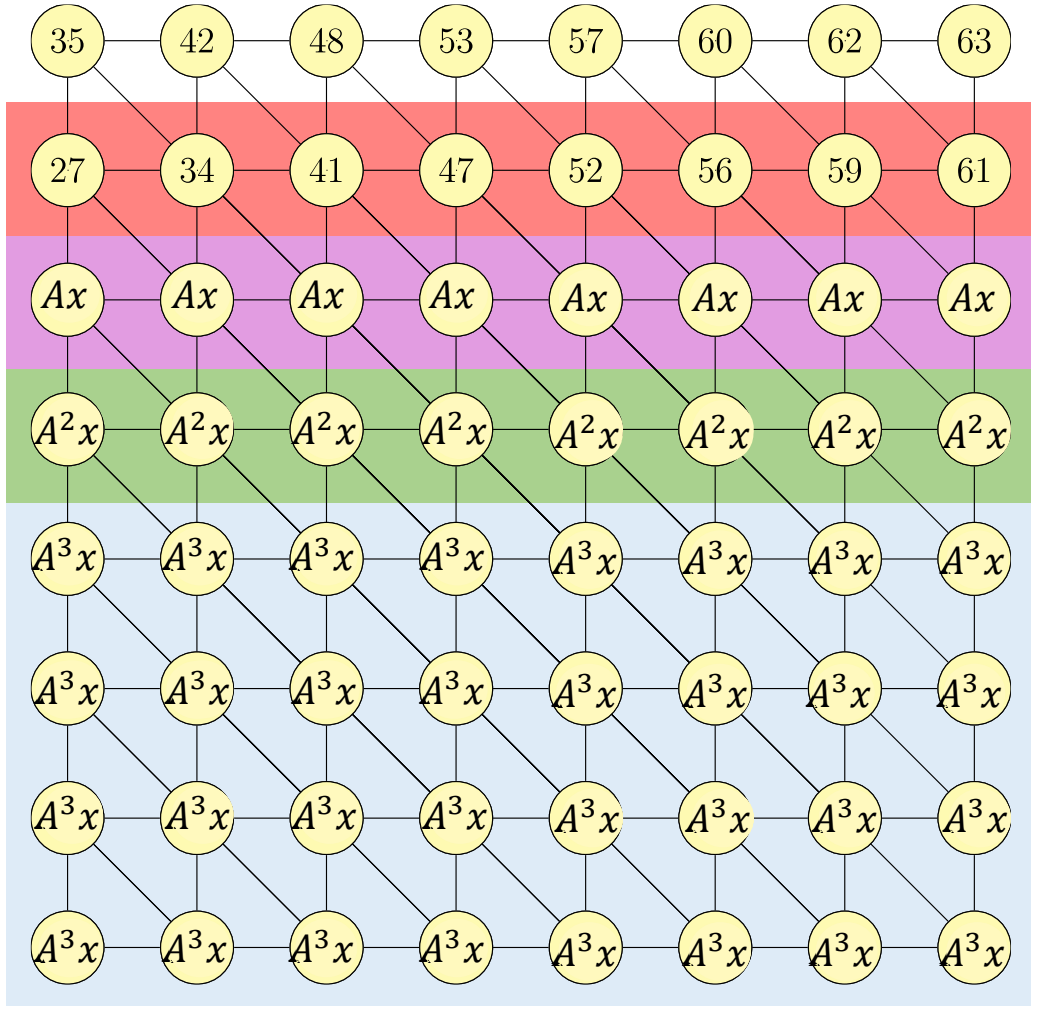
Distributed MPK



To compute A^2x we need Ax on the neighbors.

Proc 0

Distributed MPK

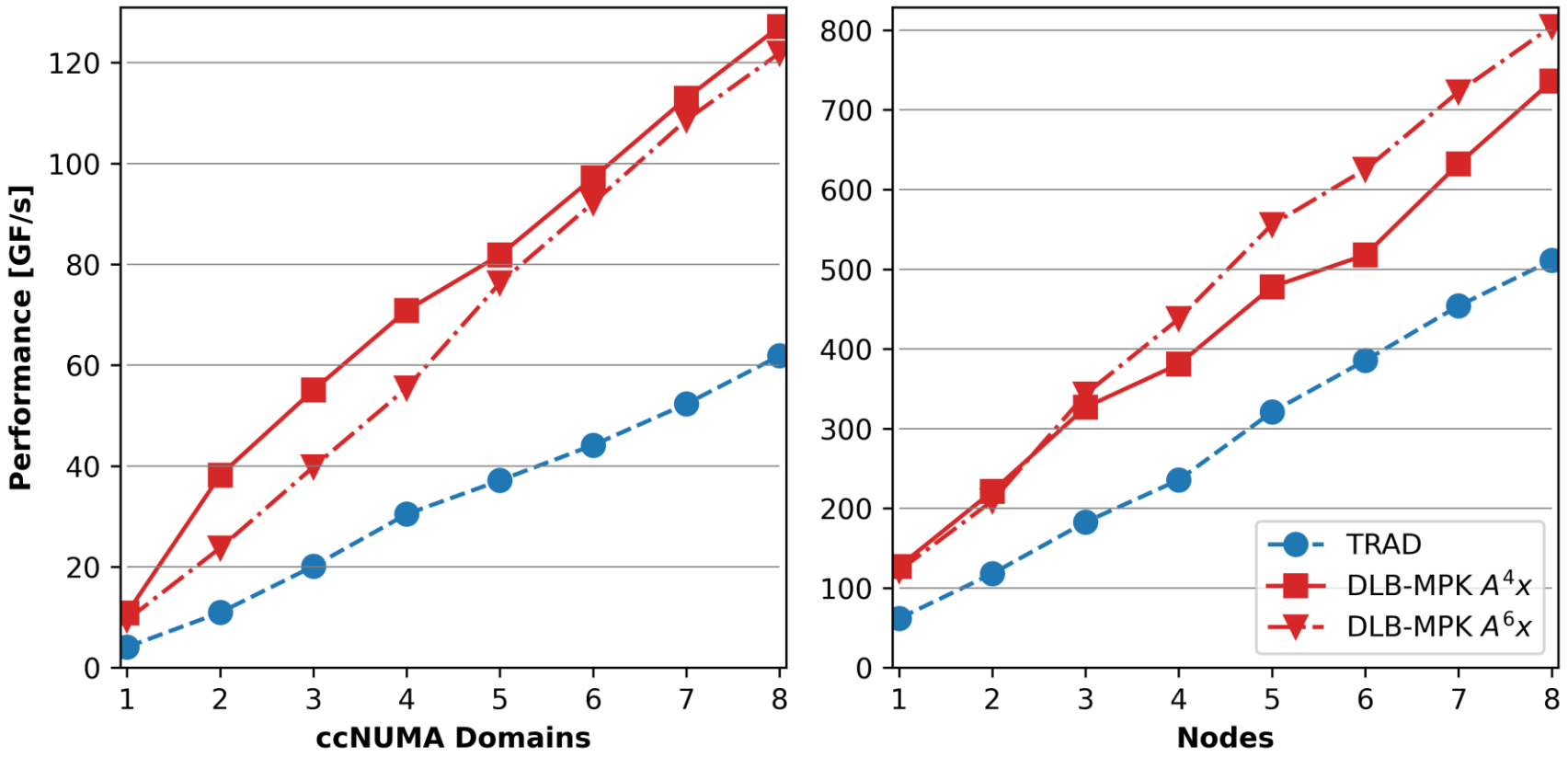


To compute A^3x we need A^2x on the neighbors.

In general, to compute $A^p x$ we need p neighbors.

Proc 0

Distributed MPK



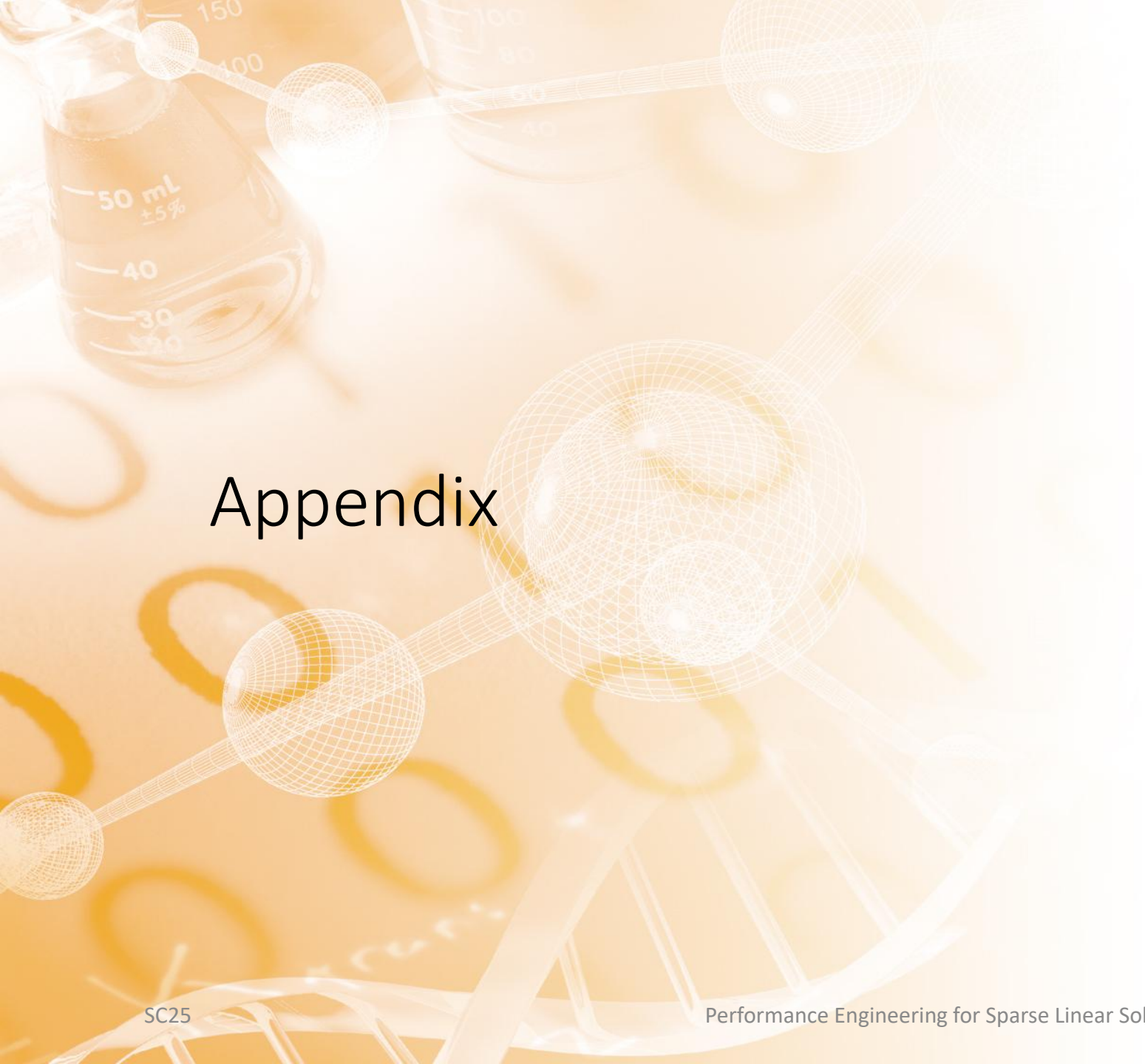
It works!

No redundant work and/or extra communication required, see upcoming paper.*

* D. Lacey, C. Alappat, F. Lange, G. Hager, and G. Wellein: *Cache Blocking of Distributed-Memory Parallel Matrix Power Kernels*, to be submitted.

Tutorial conclusions

- Memory bandwidth limitations are ubiquitous in sparse linear solvers
- SpMV performance depends on the storage format
- Roofline is an indispensable tool for performance analysis
- Time to solution is a fusion of flop/s performance and fast convergence
- Matrix powers can be optimized for better cache reuse

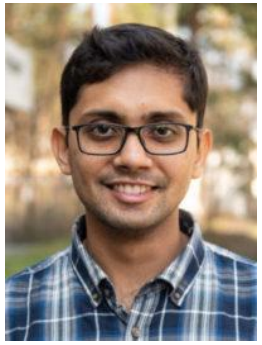


Appendix

Performance Engineering for Linear Solvers

This tutorial covers code analysis, performance modeling, and optimization for linear solvers on CPU and GPU nodes. Performance Engineering is often taught using simple loops as instructive examples for performance models and how they can guide optimization; however, full, preconditioned linear solvers comprise multiple back-to-back loops enclosed in an iteration scheme that is executed until convergence is achieved. Consequently, the concept of “optimal performance” has to account for both hardware resource efficiency and iterative solver convergence. We convey a performance engineering process that is geared towards linear iterative solvers. After introducing basic notions of hardware organization and storage for dense and sparse data structures, we show how the Roofline performance model can be applied to such solvers in predictive and diagnostic ways and how it can be used to assess the hardware efficiency of a solver, covering important corner cases such as pure memory boundedness. Then we advance to the structure of preconditioned solvers, using the Conjugate Gradient Method (CG) algorithm as a leading example. Hotspots and bottlenecks of the complete solver are identified followed by the introduction of advanced performance optimization techniques like preconditioning and cache blocking.

Christie L. Alappat



Christie Alappat received his master's degree with honors from the Bavarian Graduate School of Computational Engineering, Friedrich-Alexander-Universität Erlangen-Nürnberg. He is in the final stages of completing his doctoral studies under the guidance of Prof. Gerhard Wellein. At the same time, he is currently working at Intel as a math algorithm engineer. His research interests include performance engineering, sparse matrix and graph algorithms, iterative linear solvers, and eigenvalue computation. He is the author of the RACE open-source software framework, which is used to accelerate challenging computations in sparse linear algebra on modern compute devices. He is also the lead author of a paper that received the SIAM Activity Group on Supercomputing (SIAG/SC) Best Paper Prize in 2024.

https://hpc.fau.de/person/christie-alappat/

Jonas Thies

Jonas has more than 20 years of experience in HPC and scientific computing with applications in CFD, climate research and quantum physics. Specifically, he has worked on domain decomposition methods for sparse linear systems, implicit ocean models, sparse eigenvalue problems on heterogeneous supercomputers, code optimization for multi-core CPUs and vector processors, and software and performance engineering for scientific applications.

Jonas has a PhD in applied mathematics (Groningen 2011). He spent two years at the Center for Interdisciplinary Mathematics in Uppsala, after which he moved to Cologne as a Scientific Employee of the German Aerospace Center (DLR) Institute for Software Technology. There he led a research group on parallel numerics from 2017 to 2021. Since June 2021 he is an Assistant Professor at the Delft High Performance Computing Center DHPC, where he coordinates the center‘s training activities.

https://www.tudelft.nl/en/eemcs/the-faculty/departments/applied-mathematics/people/dr-j-jonas-thies

Hartwig Anzt

Hartwig Anzt is the Chair of Computational Mathematics at the TUM School of Computation, Information and Technology of the Technical University of Munich (TUM) Campus Heilbronn. He also holds a Research Associate Professor position at the Innovative Computing Lab (ICL) at the University of Tennessee (UTK). Hartwig Anzt received a PhD in applied mathematics from the Karlsruhe Institute of Technology (KIT) and specializes in iterative methods and preconditioning techniques for the next generation hardware architectures. He also has a long track record of high-quality development. He is author of the MAGMA-sparse open-source software package and managing lead of the Ginkgo math software library. Hartwig Anzt had served as a PI in the Software Technology (ST) pillar of the US Exascale Computing Project (ECP), including a coordinated effort aiming at integrating low-precision functionality into high-accuracy simulation codes. He also is a PI in the EuroHPC project MICROCARD.

Hartwig Anzt is the main author of more than 100 peer-reviewed publications, part of the scientific committee of international conferences, Associate Editor of the SIAM Journal on Scientific Computing (SISC), Associate Editor of ACM Transactions on Parallel Computing, workshop chair for ISC High Performance 2022, and has been elected as SIAM Activity Group on Supercomputing program manager.

https://hartwiganzt.github.io/

Georg Hager

Georg Hager holds a PhD and a habilitation degree in Computational Physics from the University of Greifswald. Since 2021 he heads the Training and Support Division of the newly founded “Erlangen National High Performance Computing Center” (NHR@FAU). Previously he was a senior researcher in the HPC Services group at Erlangen Regional Computing Center (RRZE), which is part of the Friedrich-Alexander-Universität Erlangen-Nürnberg. Recent research includes architecture-specific optimization strategies for current microprocessors, performance engineering of scientific codes, and analytic modeling of massively parallel programs. His textbook “Introduction to High Performance Computing for Scientists and Engineers” is recommended or required reading in many HPC-related lectures and courses worldwide. He has more than two decades of experience in teaching high performance computing and performance engineering to students and scientists. Together with colleagues from NHR@FAU and other centers, he conducts long-standing series of tutorials on Performance Engineering and Hybrid Programming.

https://blogs.fau.de/hager