Blocking Optimization Strategies for Sparse Tensor Computation

Jee Choi[1], Xing Liu[1], Shaden Smith[2], and Tyler Simon[3]

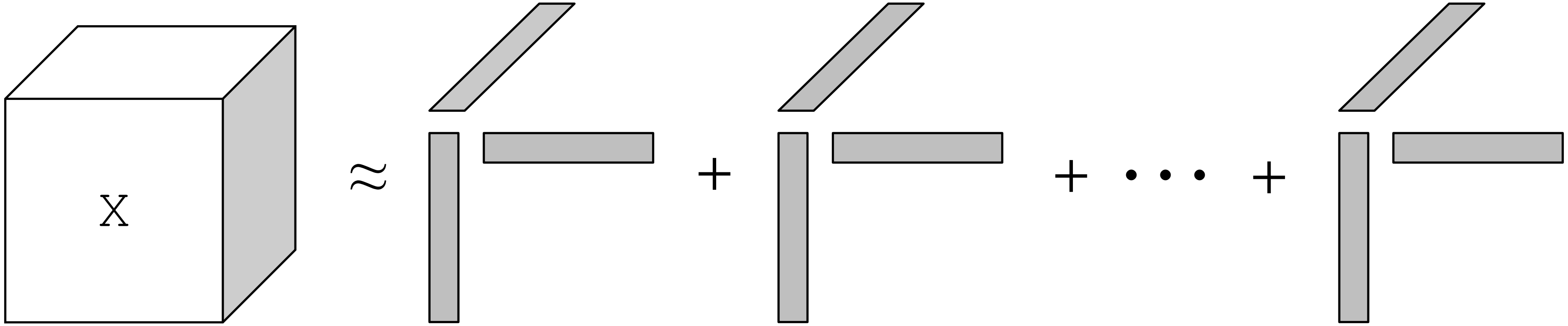
[1]IBM T. J. Watson Research, [2]University of Minnesota, [3]University of Maryland Baltimore County

SIAM Annual Meeting

July 12th, 2017

Tensors are multi-dimensional arrays

- CANDECOMP/Parafac (CP) decomposition creates a set of factor matrices



The take-away from this presentation

- There is lack of clear understanding about performance bottlenecks in sparse tensor decomposition
- Using various blocking techniques mitigate these bottlenecks
- Our optimizations demonstrate significant speedup on synthetic and real-world data for both shared-memory and distributed implementations

Fix every other factor matrix and solve for the remaining one

```
procedure CP-ALS ($\mathcal{X}$, R)
      repeat

            $C = X_{(3)}(B \odot A)(B^T B * A^T A)^{X}$
            normalize columns of C to length 1
            $B = X_{(2)}(C \odot A)(C^T C * A^T A)^{X}$
            normalize columns of B to length 1
            $A = X_{(1)}(C \odot B)(C^T C * B^T B)^{X}$
            store column norms of A in λ and normalize to 1

      until max iteration reached or error less than ϵ

end procedure
```

Calculating MTTKRP is the primary bottleneck

```
procedure CP-ALS ($\mathcal{X}$, R)
      repeat                    MTTKRP

            C = $X_{(3)}(B \odot A)(B^TB * A^TA)^{\chi}$
            normalize columns of C to length 1
            B = $X_{(2)}(C \odot A)(C^TC * A^TA)^{\chi}$
            normalize columns of B to length 1
            A = $X_{(1)}(C \odot B)(C^TC * B^TB)^{\chi}$
            store column norms of A in λ and normalize to 1

      until max iteration reached or error less than ϵ

end procedure
```

Calculating MTTKRP is the primary bottleneck

```
procedure CP-ALS ($\mathcal{X}$, R)
      repeat

            $C = X_{(3)}(B \odot A)(B^T B * A^T A)^{\chi}$
            normalize columns of C to length 1
            $B = X_{(2)}(C \odot A)(C^T C * A^T A)^{\chi}$
            normalize columns of B to length 1
            $A = X_{(1)}(C \odot B)(C^T C * B^T B)^{\chi}$
            store column norms of A in $\lambda$ and normalize to 1

      until max iteration reached or error less than $\epsilon$

end procedure
```

Matricized Tensor **TKRP**

Calculating MTTKRP is the primary bottleneck

```
procedure CP-ALS ($\mathcal{X}$, R)
      repeat

            $C = X_{(3)}(B \odot A)(B^TB * A^TA)^{\mathcal{X}}$
            normalize columns of C to length 1
            $B = X_{(2)}(C \odot A)(C^TC * A^TA)^{\mathcal{X}}$
            normalize columns of B to length 1
            $A = X_{(1)}(C \odot B)(C^TC * B^TB)^{\mathcal{X}}$
            store column norms of A in λ and normalize to 1

      until max iteration reached or error less than ϵ

end procedure
```

Matricized Tensor Times KRP

Calculating MTTKRP is the primary bottleneck

```
procedure CP-ALS ($\mathcal{X}$, R)
    repeat

        C = $\mathbf{X_{(3)}(B \odot A)}(B^TB * A^TA)^{\chi}$
        normalize columns of C to length 1
        B = $\mathbf{X_{(2)}(C \odot A)}(C^TC * A^TA)^{\chi}$
        normalize columns of B to length 1
        A = $\mathbf{X_{(1)}(C \odot B)}(C^TC * B^TB)^{\chi}$
        store column norms of A in λ and normalize to 1

    until max iteration reached or error less than ϵ

end procedure
```

Matricized **T**ensor **T**imes **K**hatri-**R**ao **P**roduct

Calculating MTTKRP is the primary bottleneck

> 90% total execution time

```
procedure CP-ALS ($\mathit{X}$, R)
    repeat

        $C = \mathbf{X_{(3)}(B \odot A)}(B^TB * A^TA)^{X}$
        normalize columns of C to length 1
        $B = \mathbf{X_{(2)}(C \odot A)}(C^TC * A^TA)^{X}$
        normalize columns of B to length 1
        $A = \mathbf{X_{(1)}(C \odot B)}(C^TC * B^TB)^{X}$
        store column norms of A in λ and normalize to 1

    until max iteration reached or error less than ϵ

end procedure
```

Problem is formulated as matrix operations

```
procedure CP-ALS ($X$, R)
    repeat

        $C = X_{(3)}(B \odot A)(B^TB * A^TA)^{X}$
        normalize columns of C to length 1
        $B = X_{(2)}(C \odot A)(C^TC * A^TA)^{X}$
        normalize columns of B to length 1
        $A = X_{(1)}(C \odot B)(C^TC * B^TB)^{X}$
        store column norms of A in λ and normalize to 1

    until max iteration reached or error less than ϵ

end procedure
```

$$X \in \mathbb{R}^{I \times J \times K}$$

$$A \in \mathbb{R}^{I \times R}$$

$$B \in \mathbb{R}^{J \times R}$$

$$C \in \mathbb{R}^{K \times R}$$

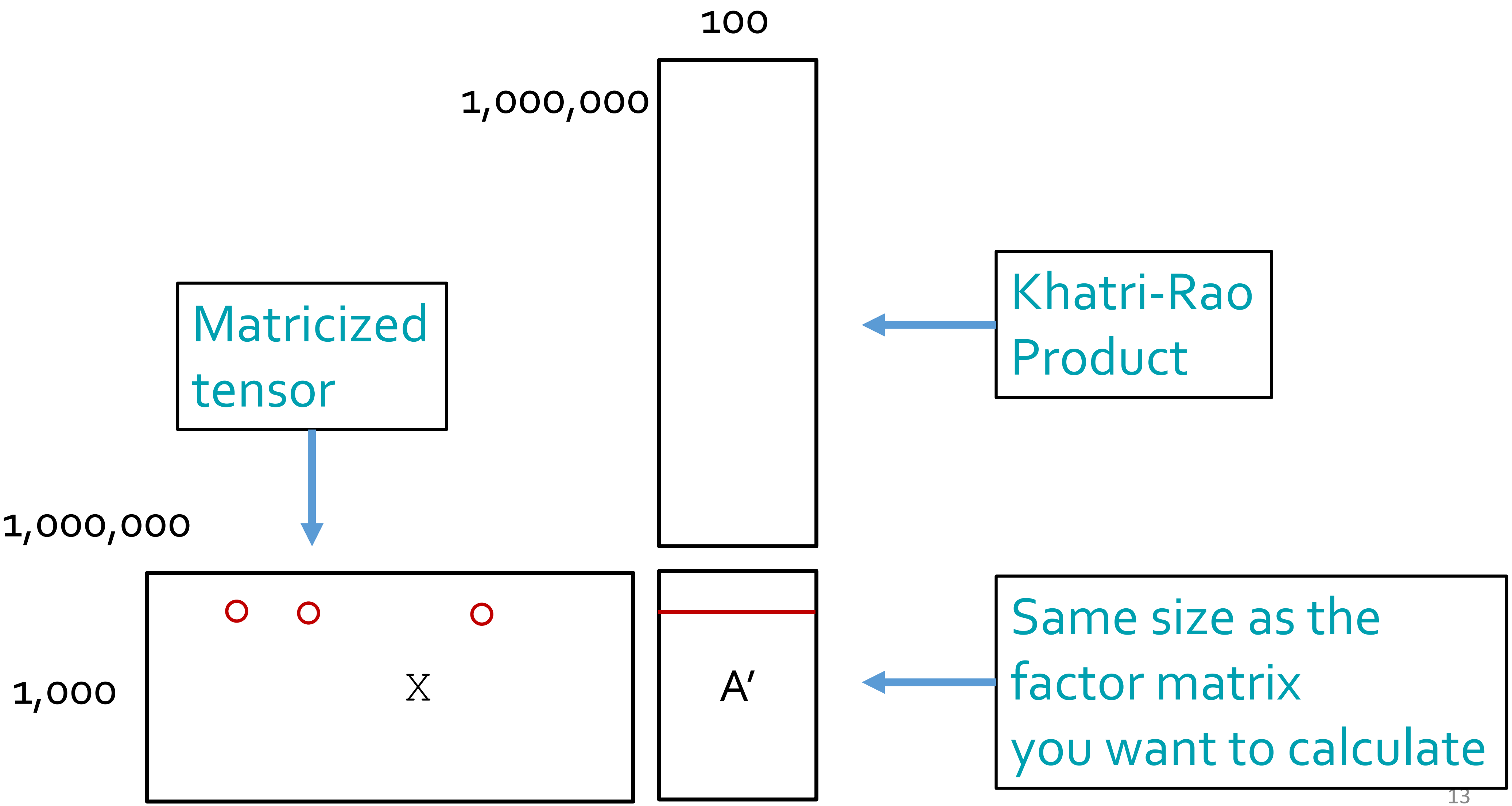
Directly computing MTTKRP is very expensive

- For a 1000×1000×1000 tensor with rank 100…
  - $\mathrm{X}_{(3)}$ is a **1,000 × 1,000,000** matrix, and
  - (B ⦿ A) is a **1,000,000 × 100** matrix
  - Direct computation is **expensive**

But not necessary

- For a 1000×1000×1000 tensor with rank 100…
  - $\mathrm{X}_{(3)}$ is a **1,000 × 1,000,000** matrix, and
  - (B ⊙ A) is a **1,000,000 × 100** matrix
  - Direct computation is **expensive**
- **Not necessary for sparse tensors.**

MTTKRP expressed as matrix operations

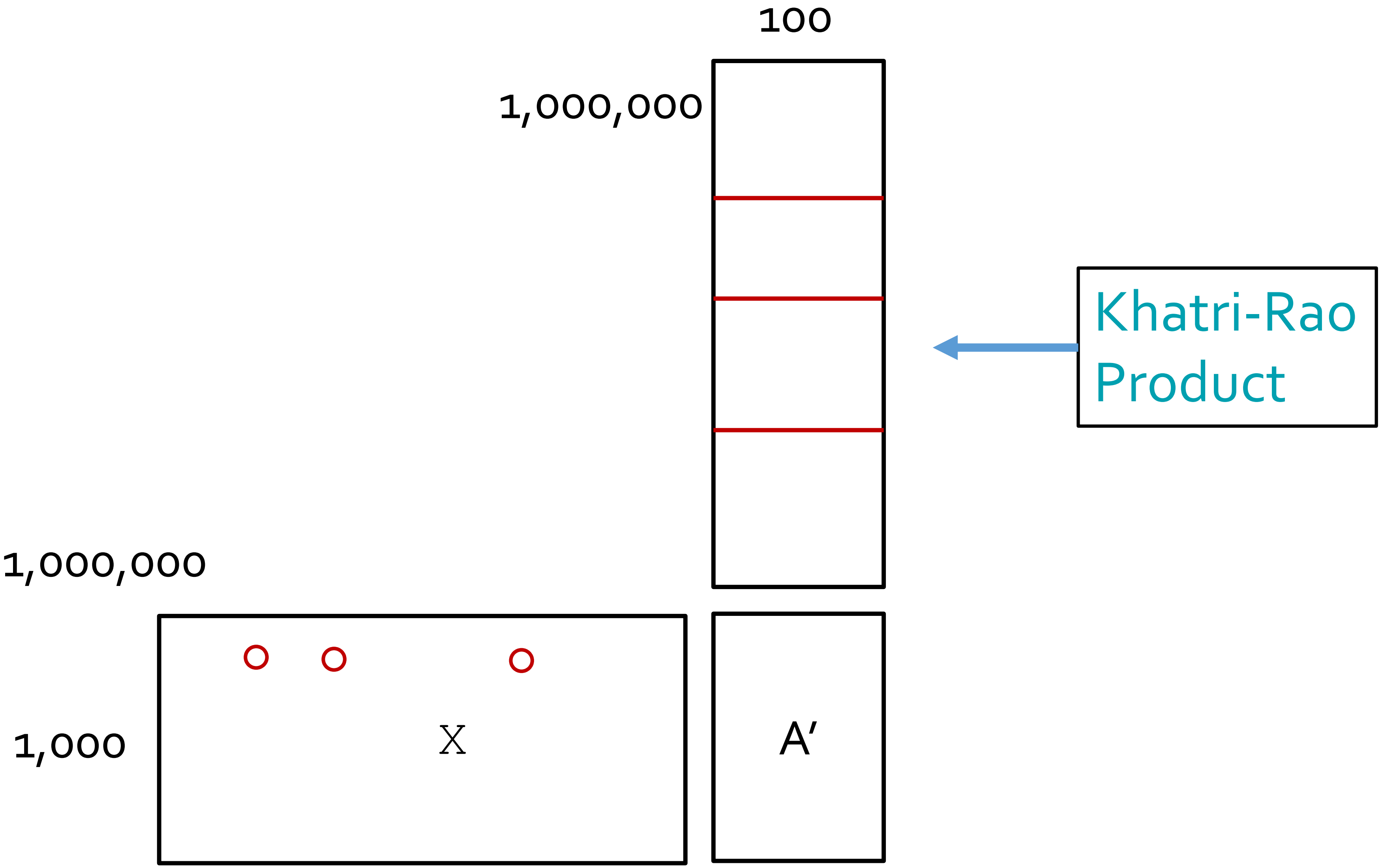


Matricized tensor

Khatri-Rao Product

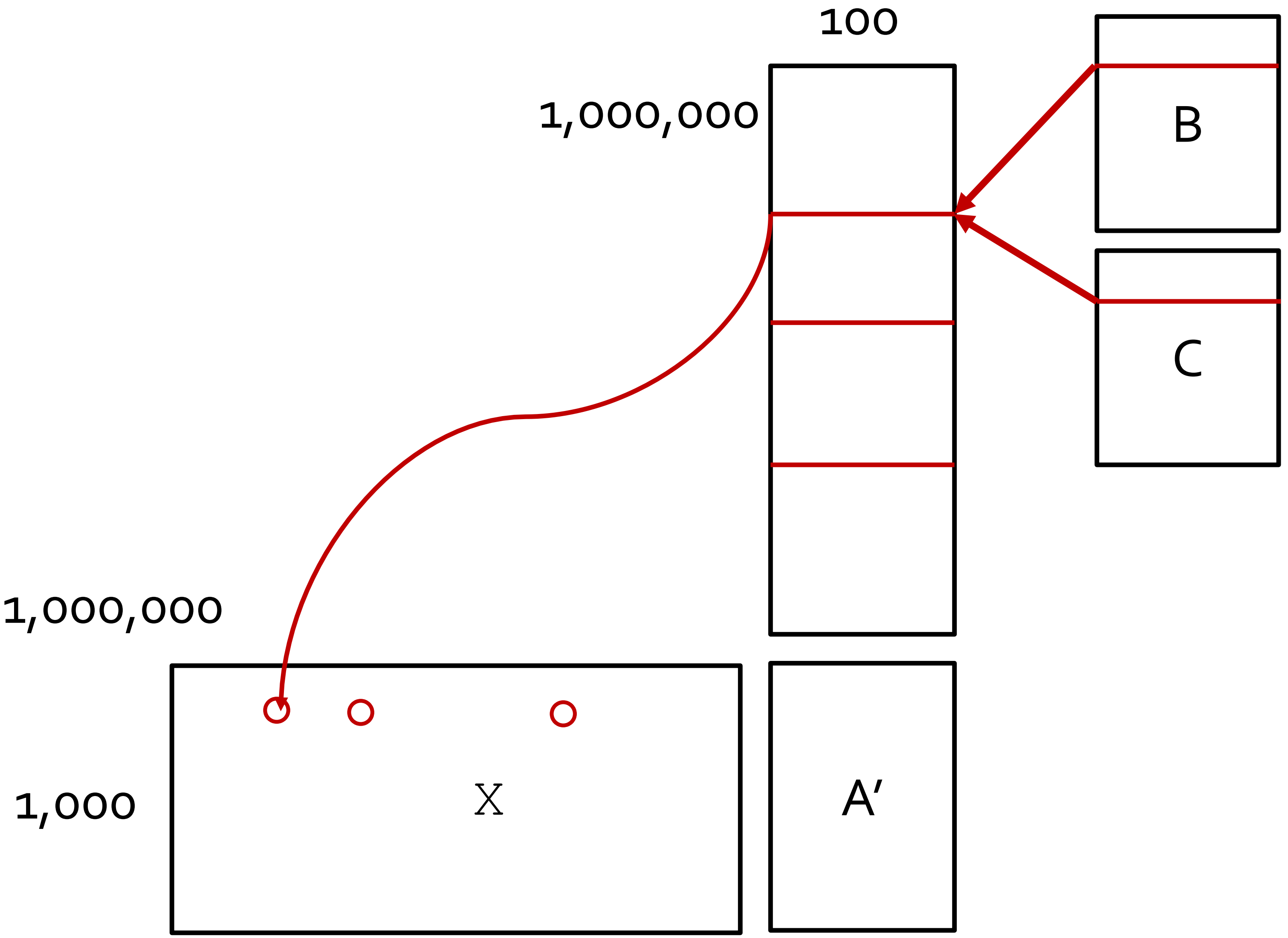
Same size as the factor matrix you want to calculate

MTTKRP simplified



Khatri-Rao Product

MTTKRP simplified

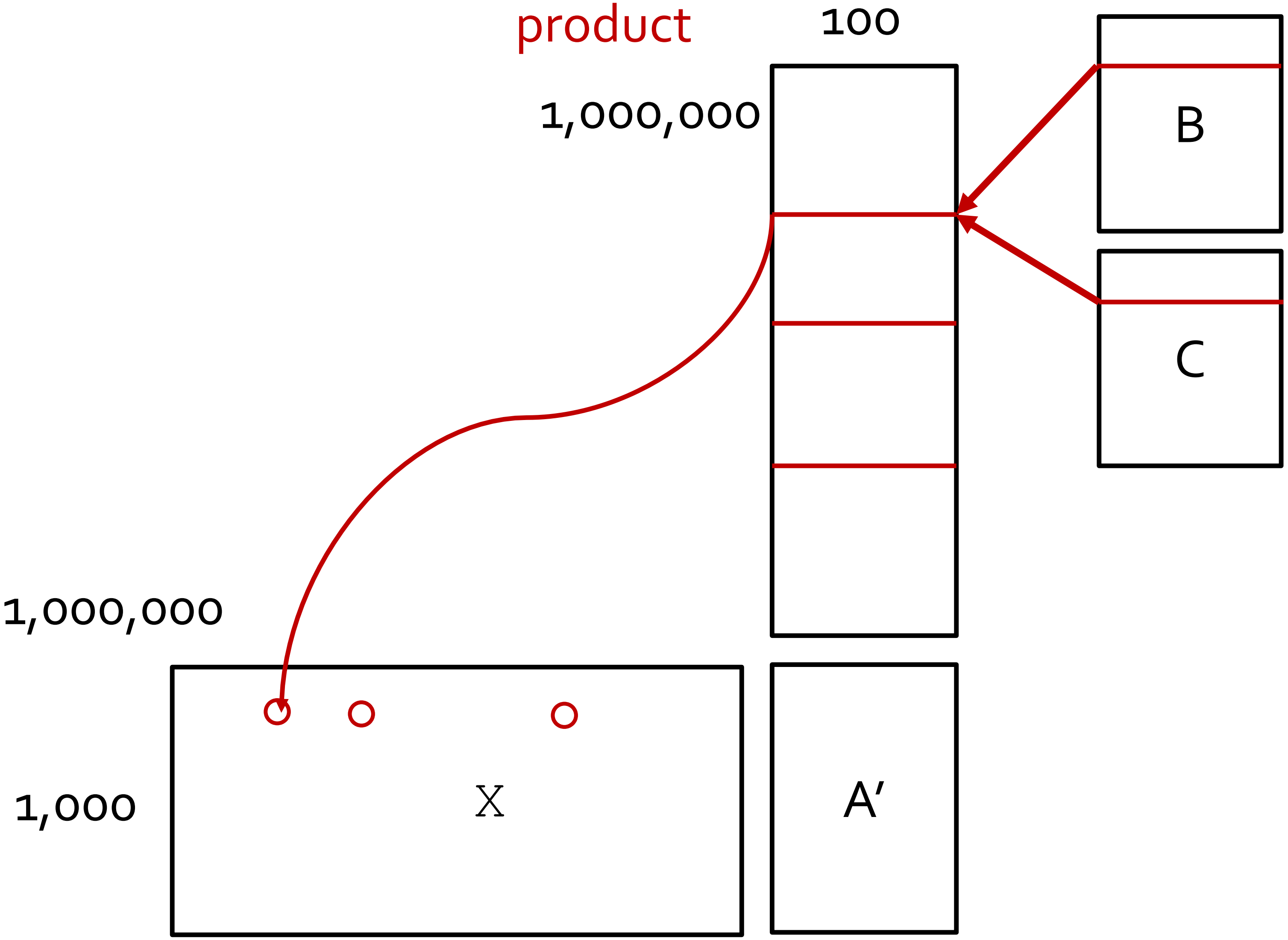


Load 1 row each from B and C

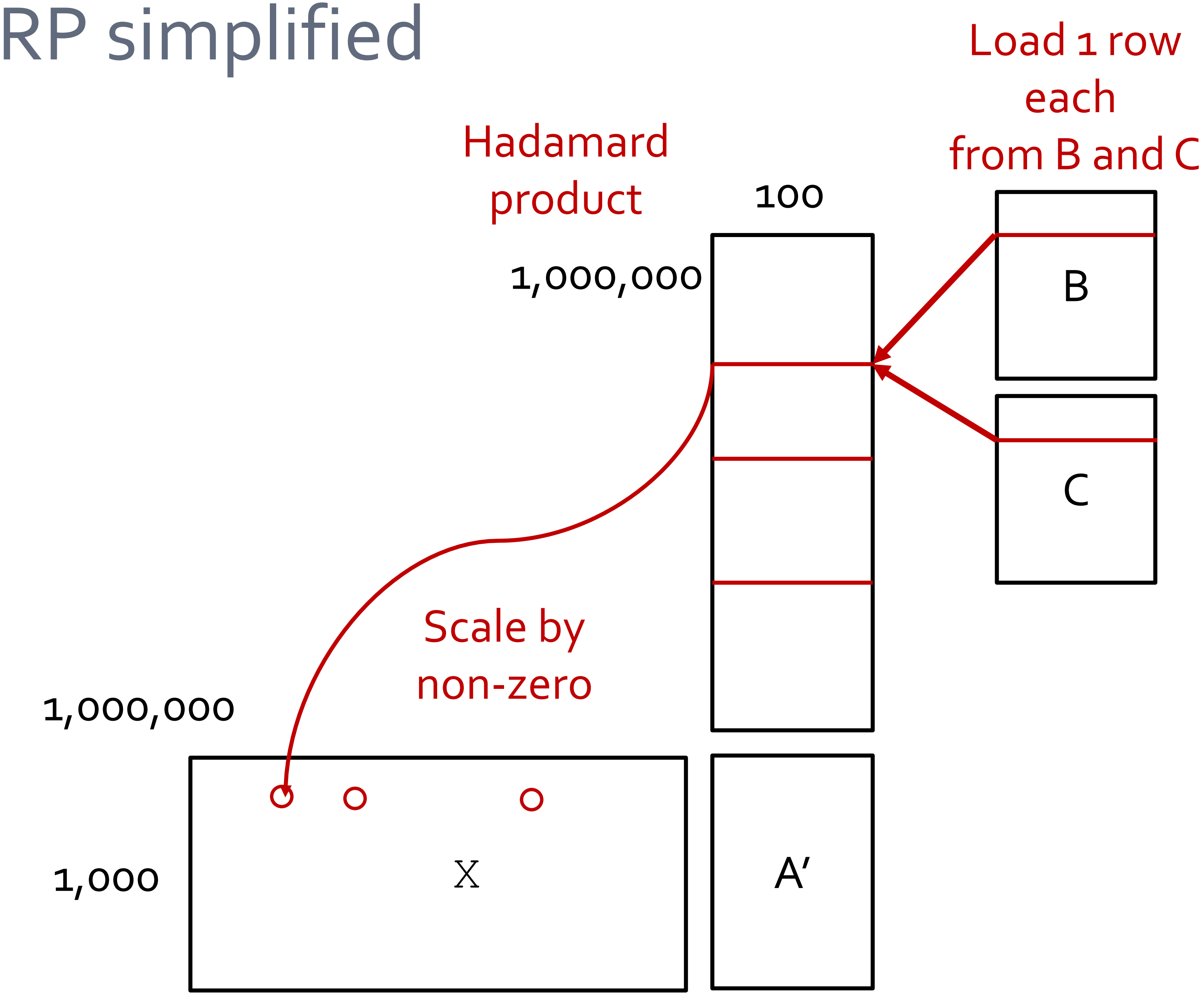
MTTKRP simplified

Hadamard product

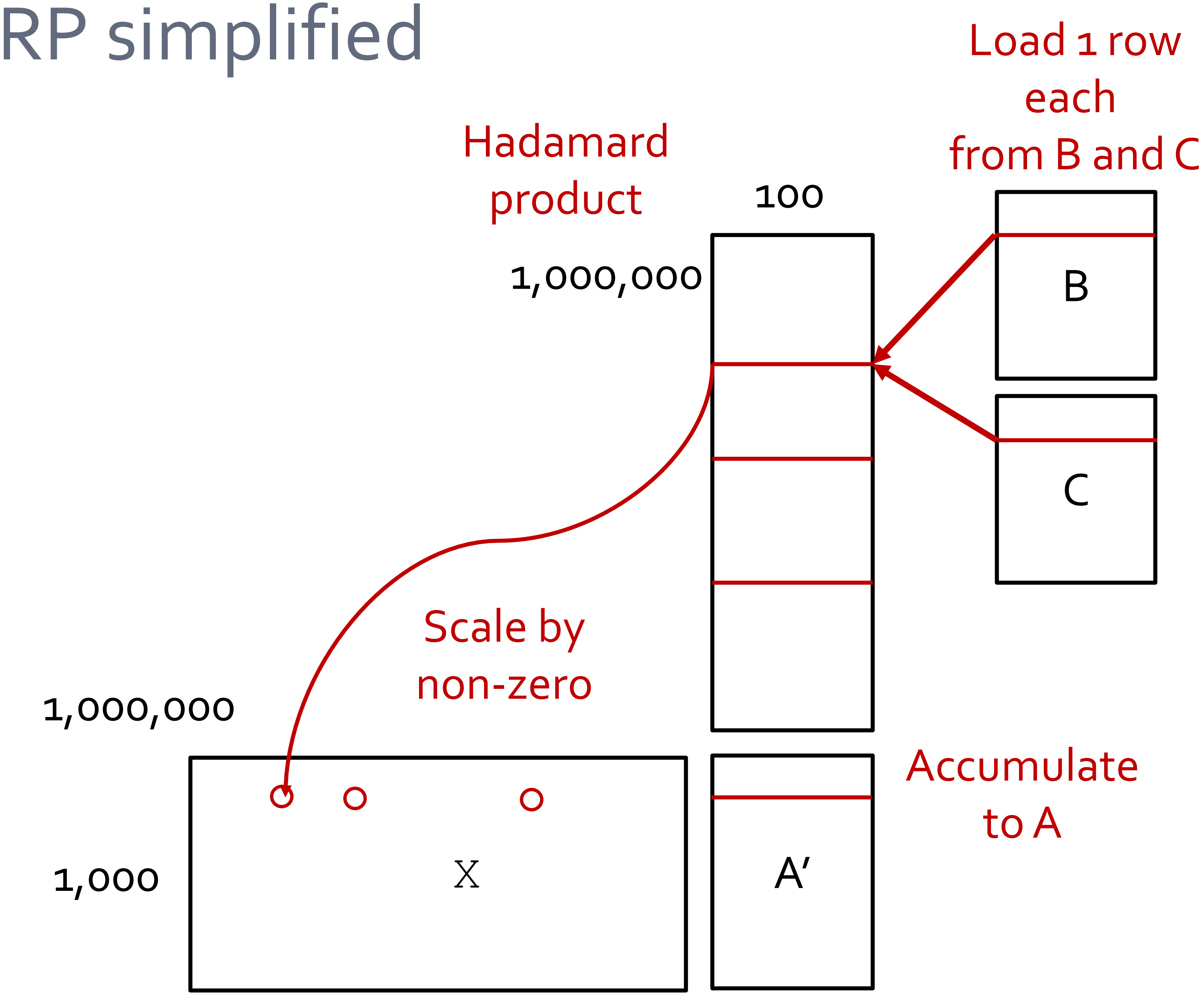
Load 1 row each from B and C



MTTKRP simplified



MTTKRP simplified

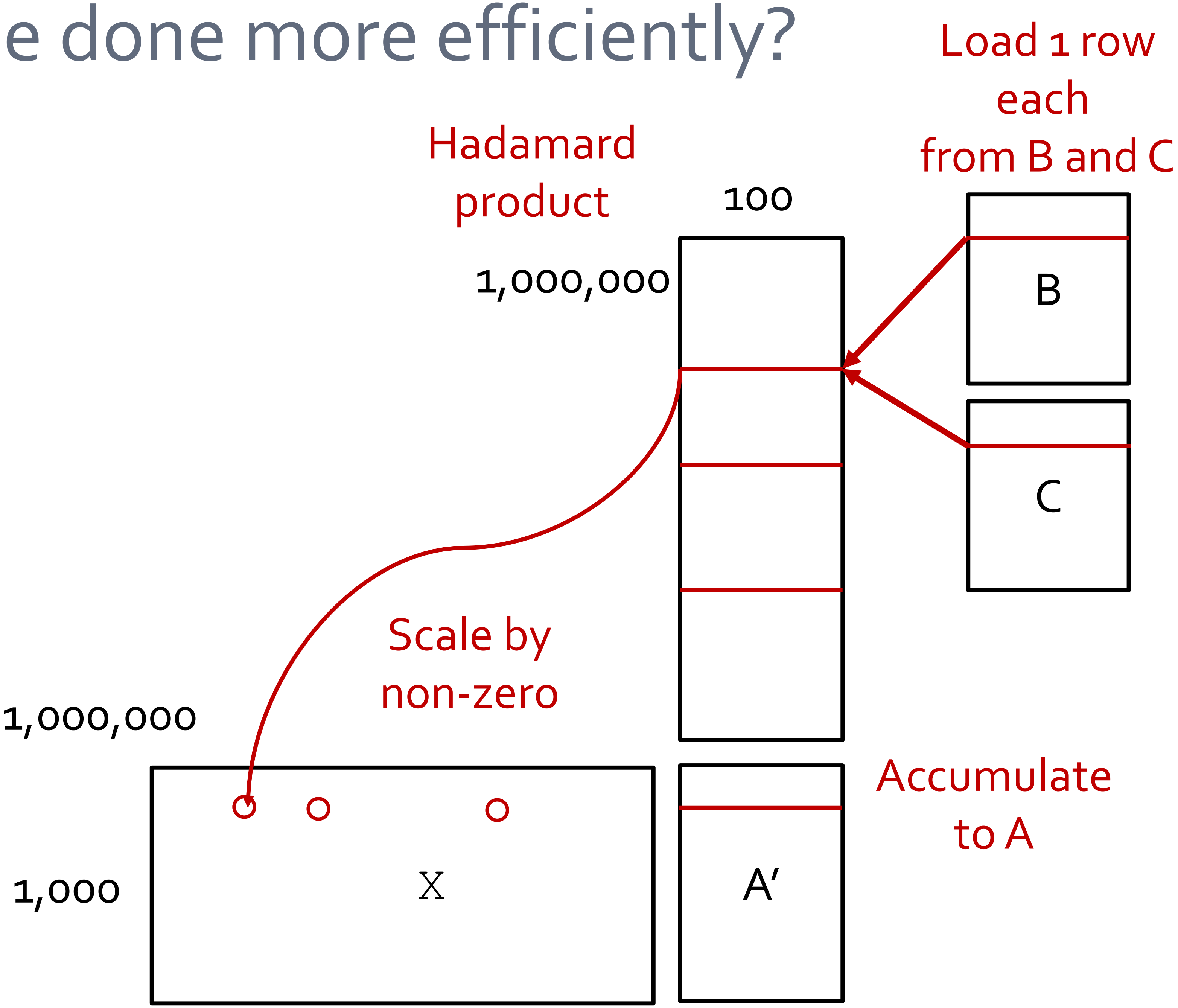

Load 1 row each from B and C

Hadamard product

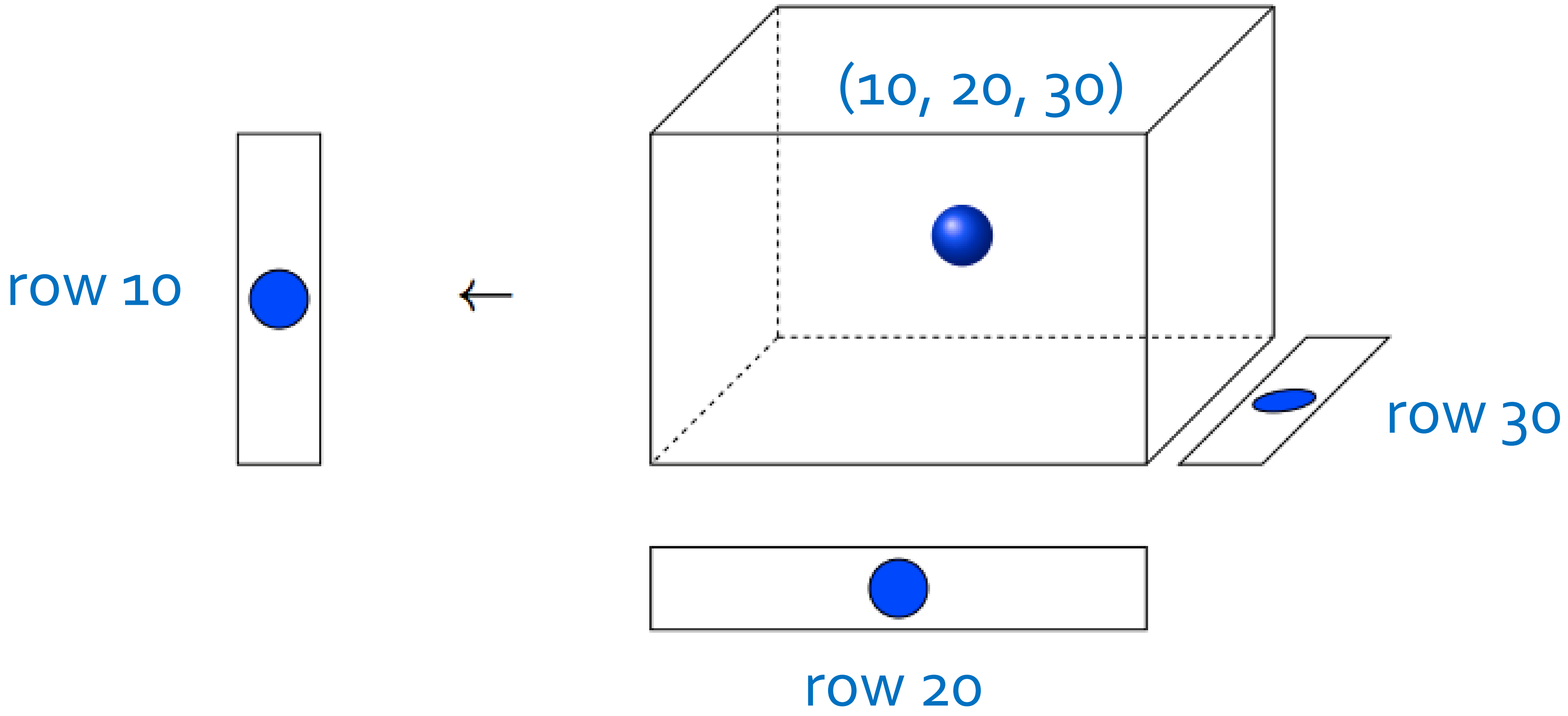
Scale by non-zero

Accumulate to A

Can it be done more efficiently?



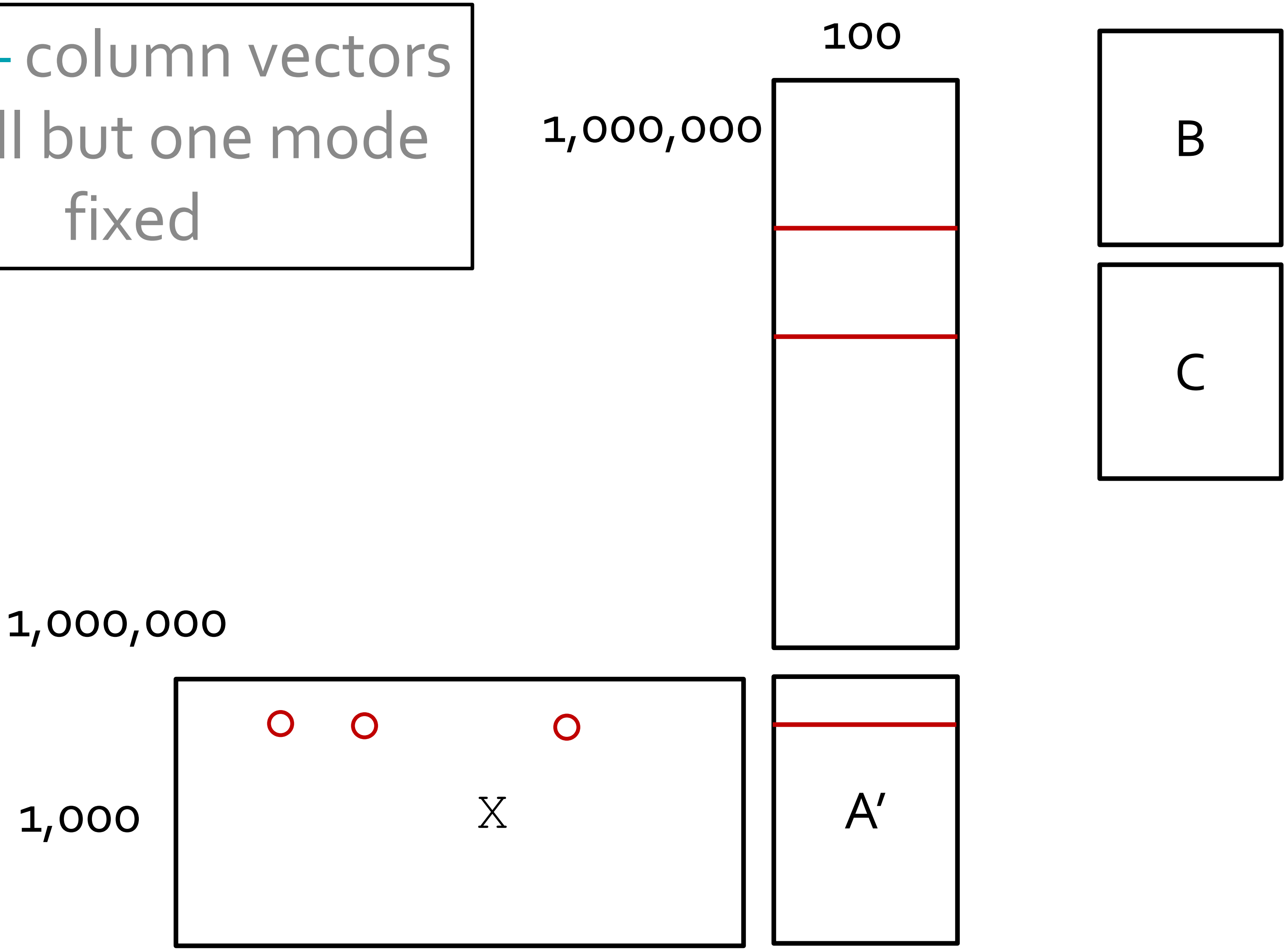
In 3D space…



Shaden Smith, et al., ***SPLATT: Efficient and Parallel Sparse Tensor-Matrix Multiplication***,*IPDPS 2015*

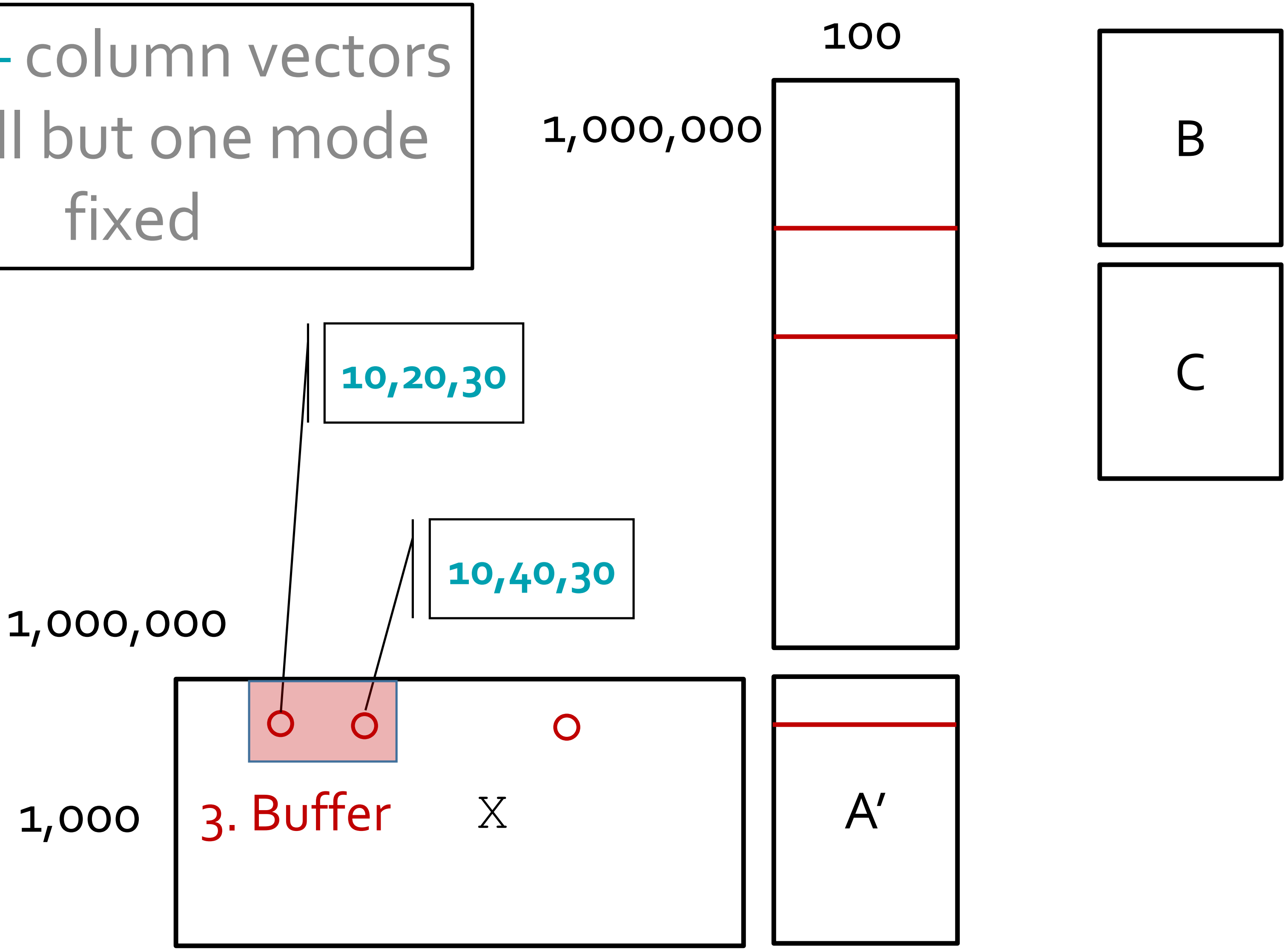
Reduce computing by processing at fiber granularity

Fiber – column vectors with all but one mode fixed



Reduce computing by processing at fiber granularity

Fiber – column vectors with all but one mode fixed

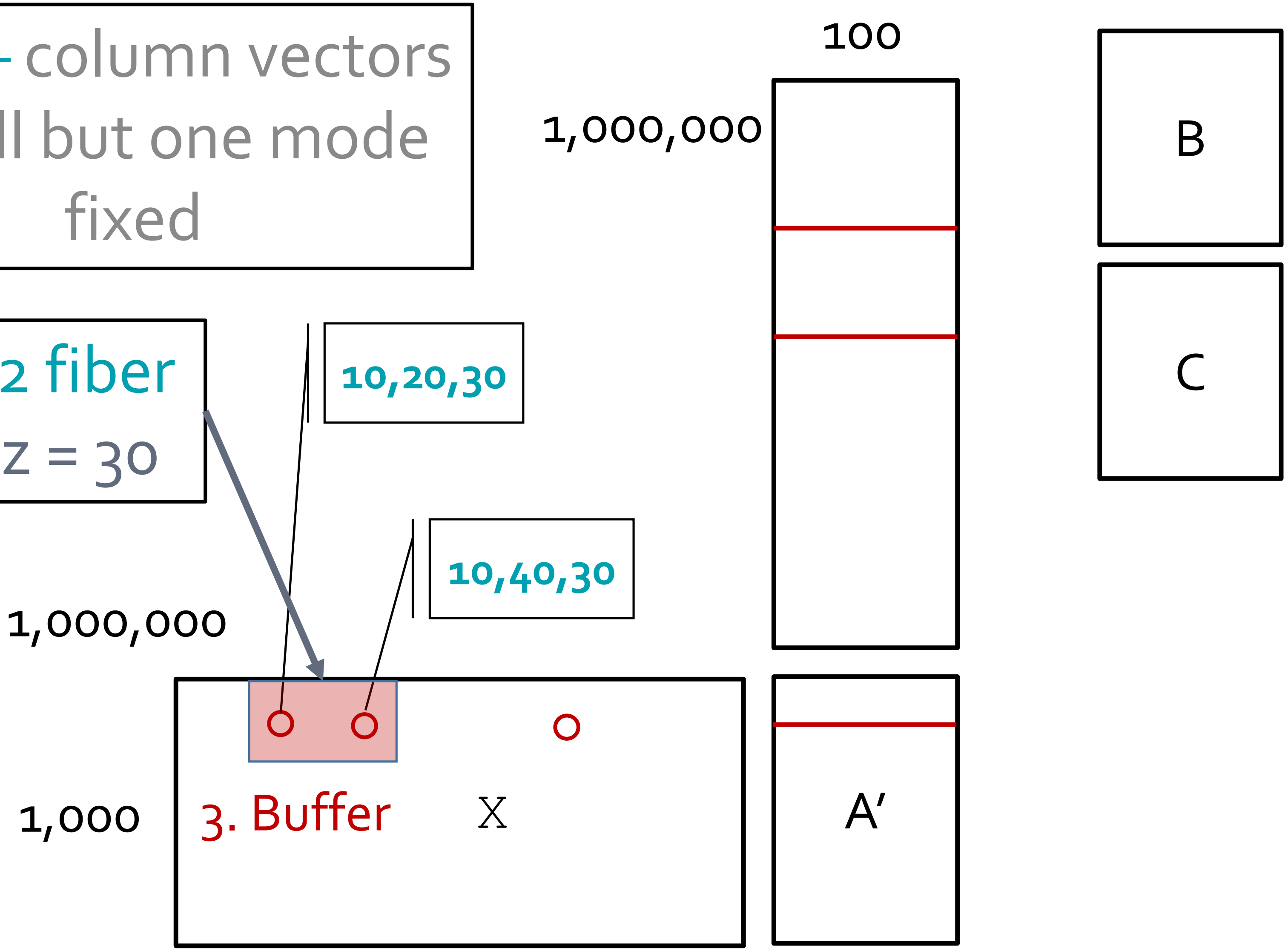


Reduce computing by processing at fiber granularity

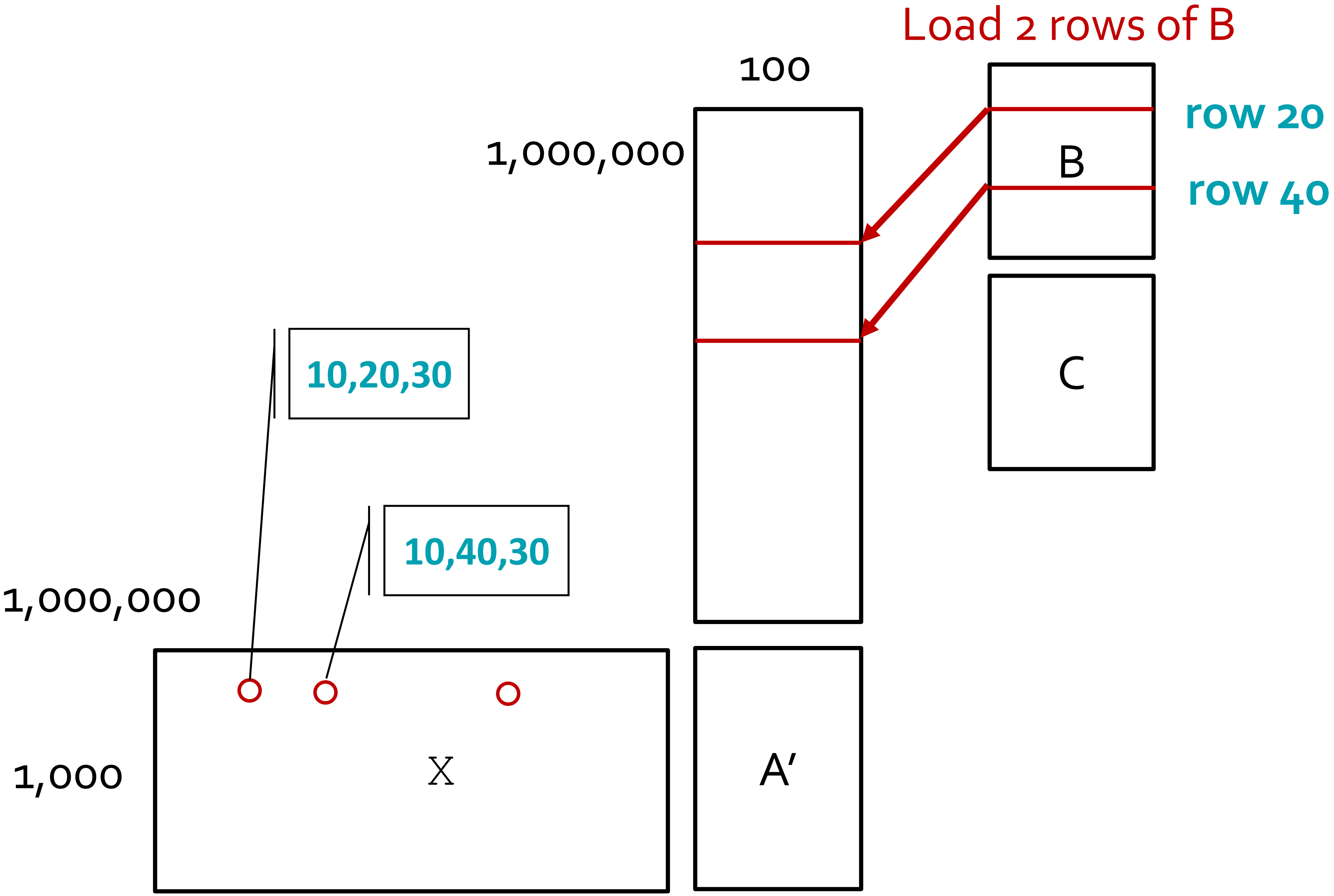
Fiber – column vectors with all but one mode fixed

Mode-2 fiber

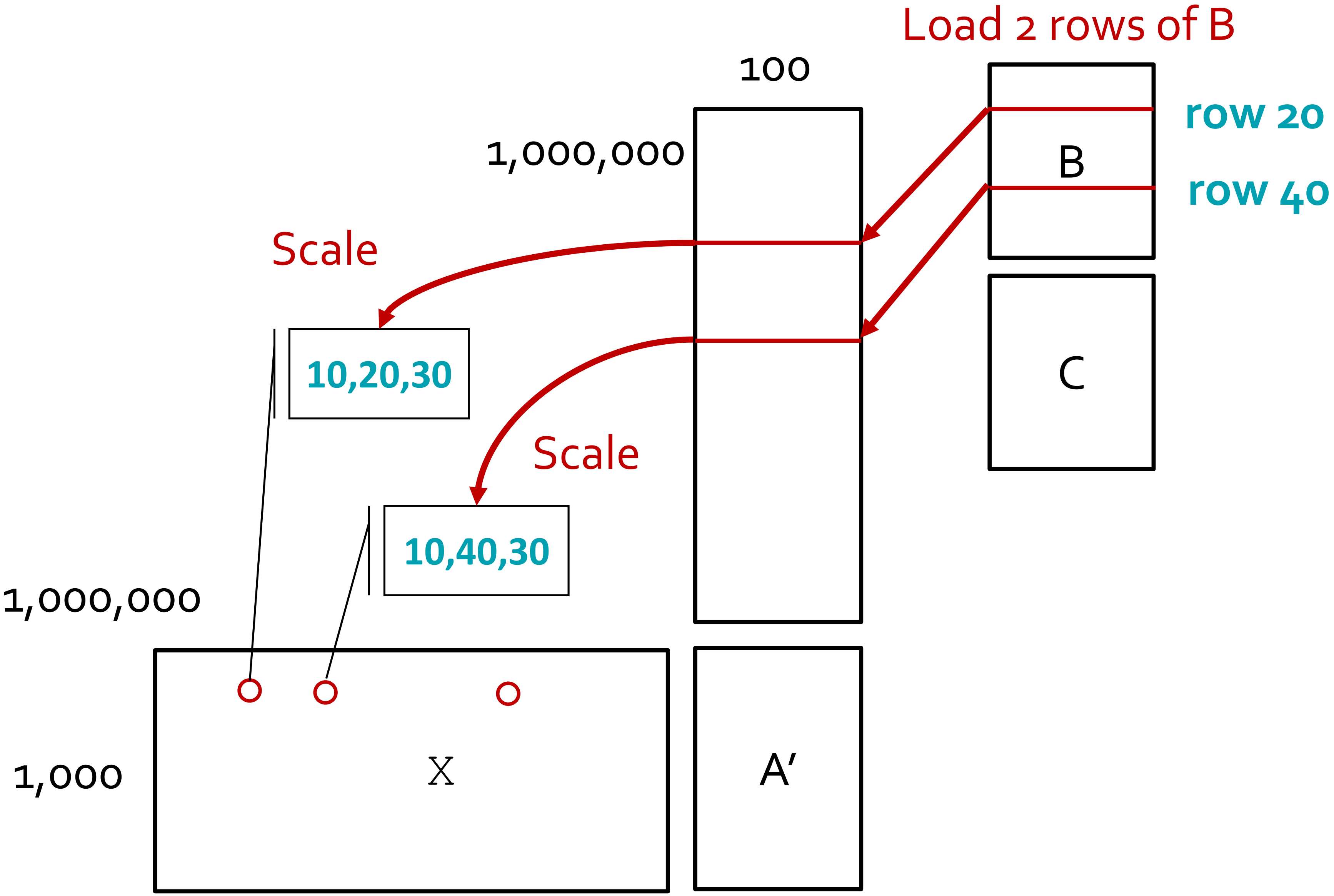
x = 10, z = 30



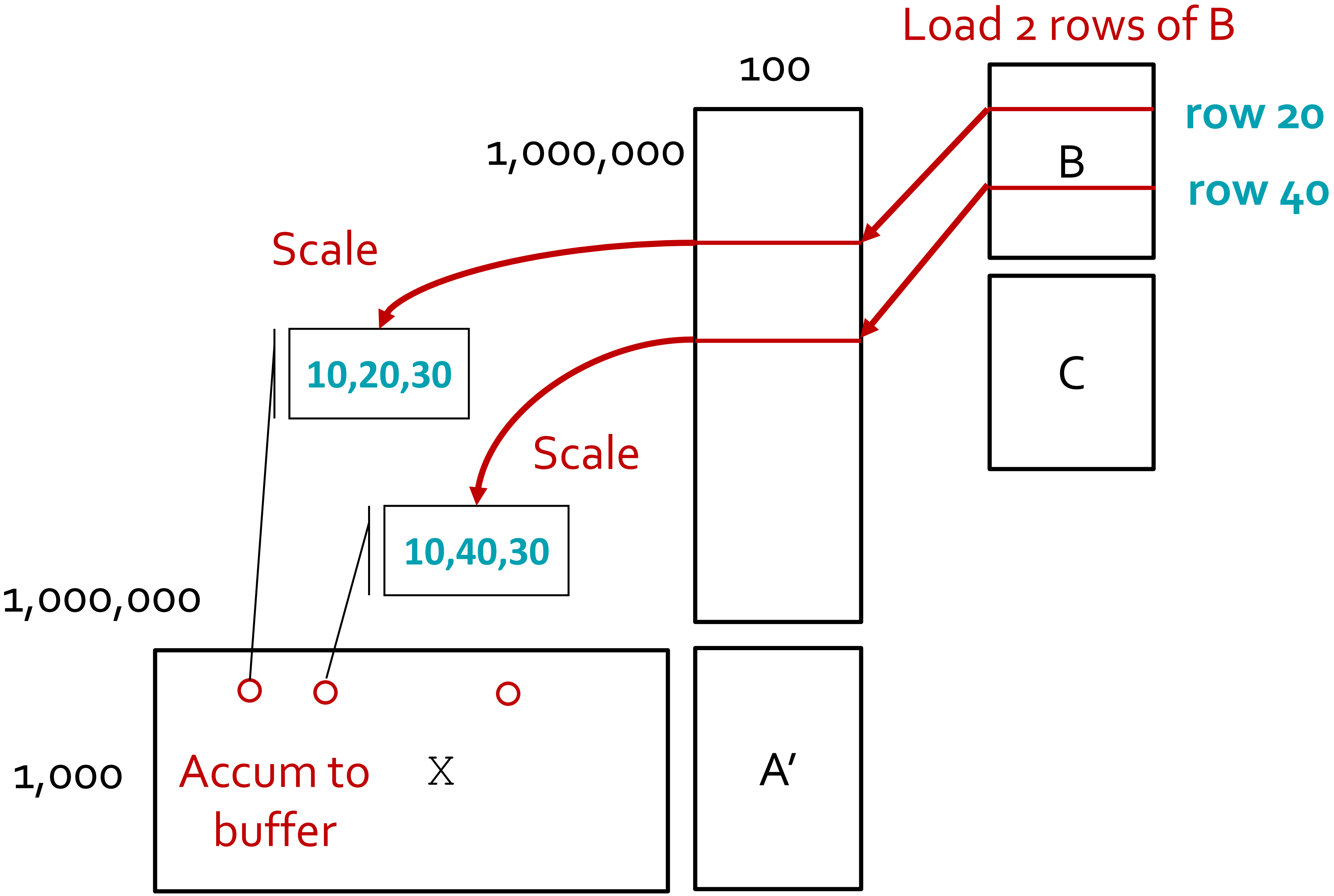
First load rows from B (mode-2 matrix)



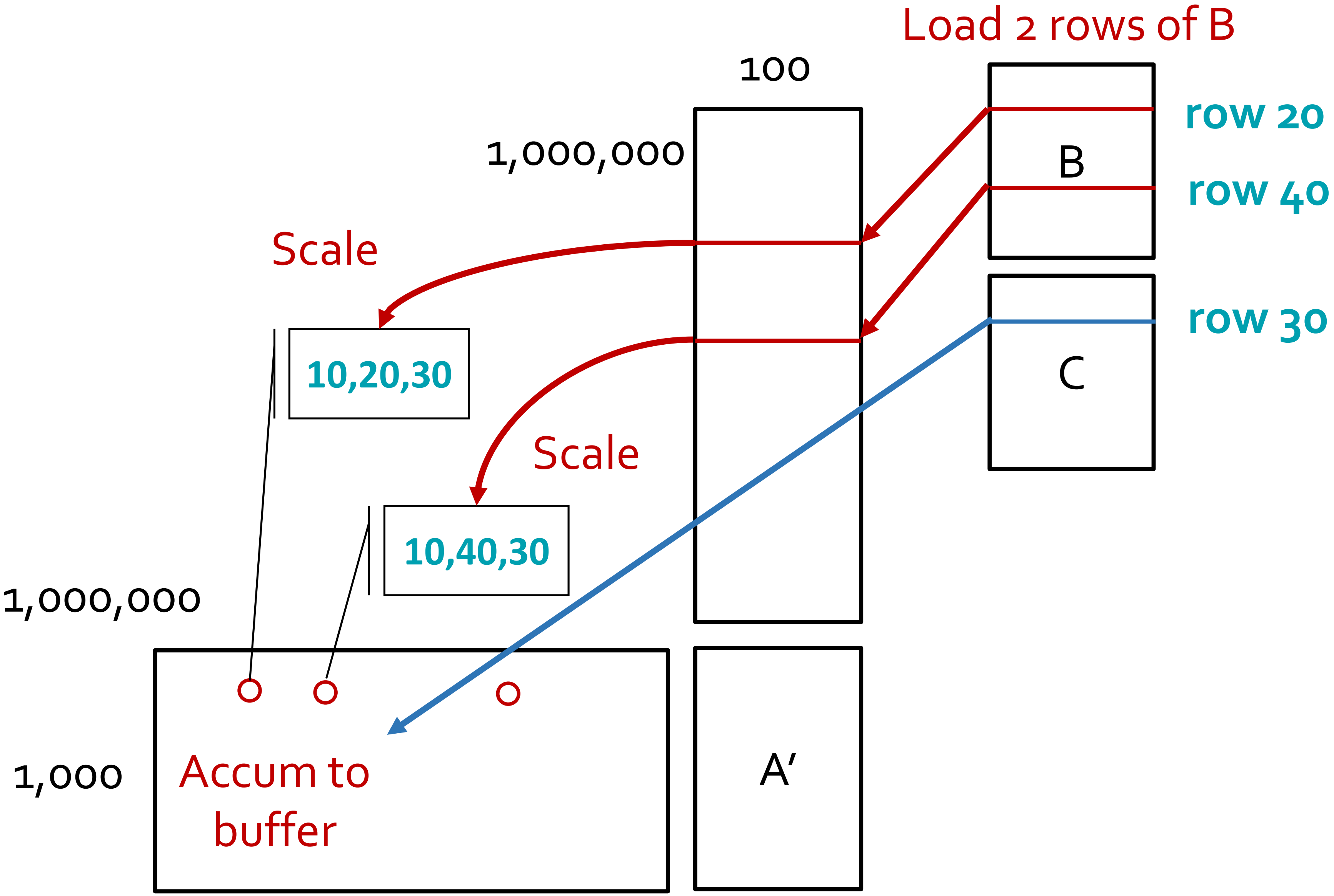
Scale the rows by non-zero values



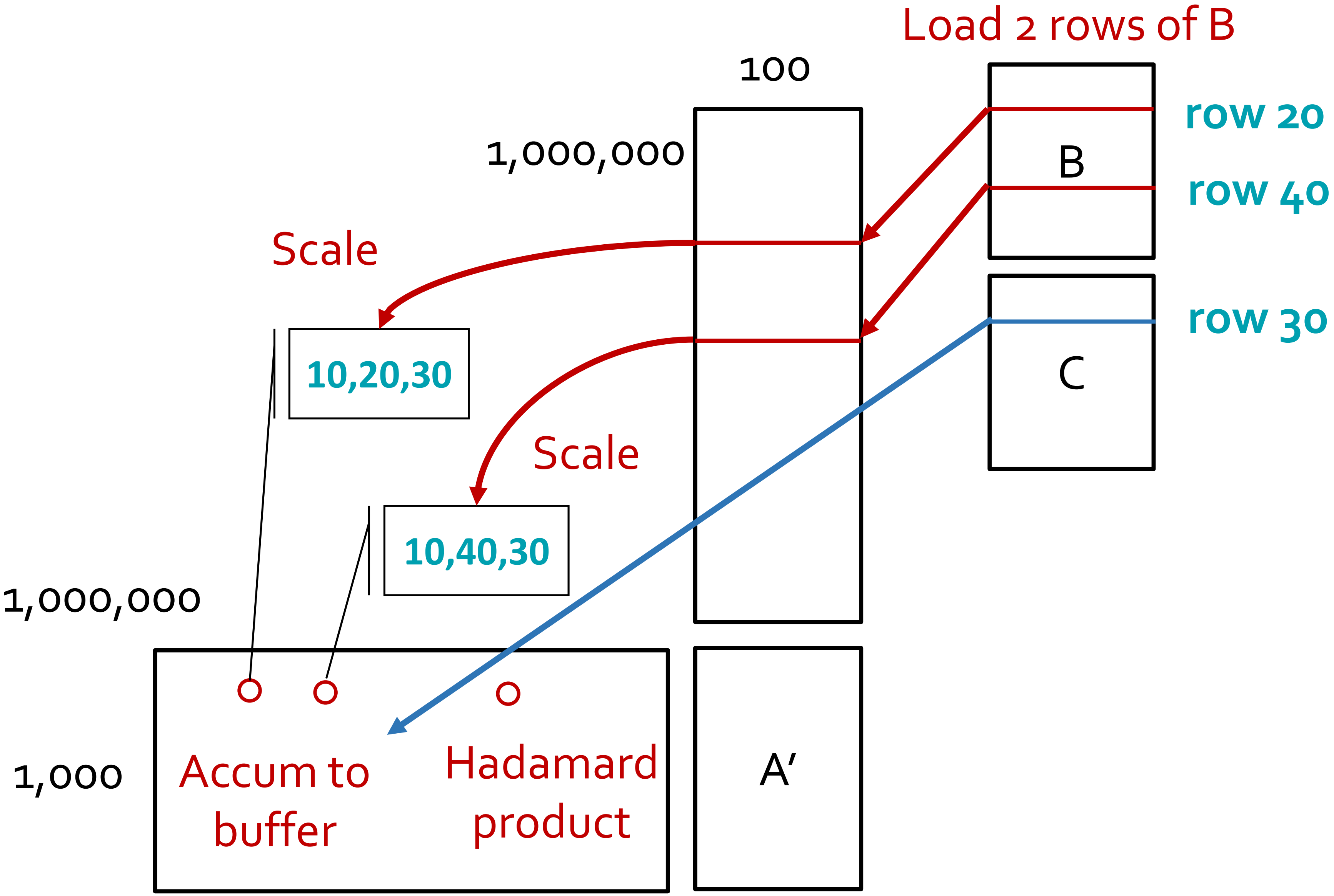
Accumulate them to a temporary buffer



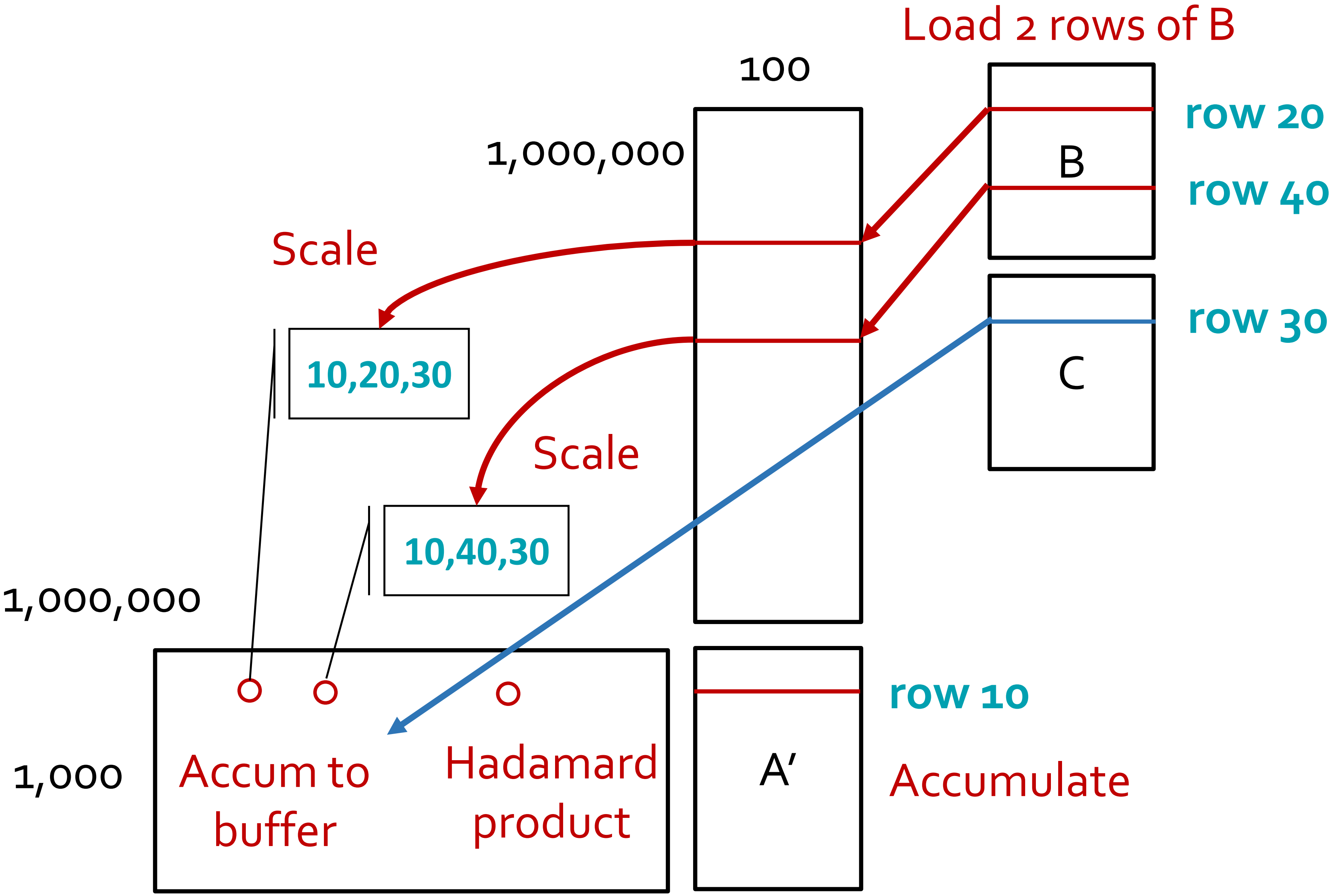
Load the “common” row from C (mode-3 matrix)



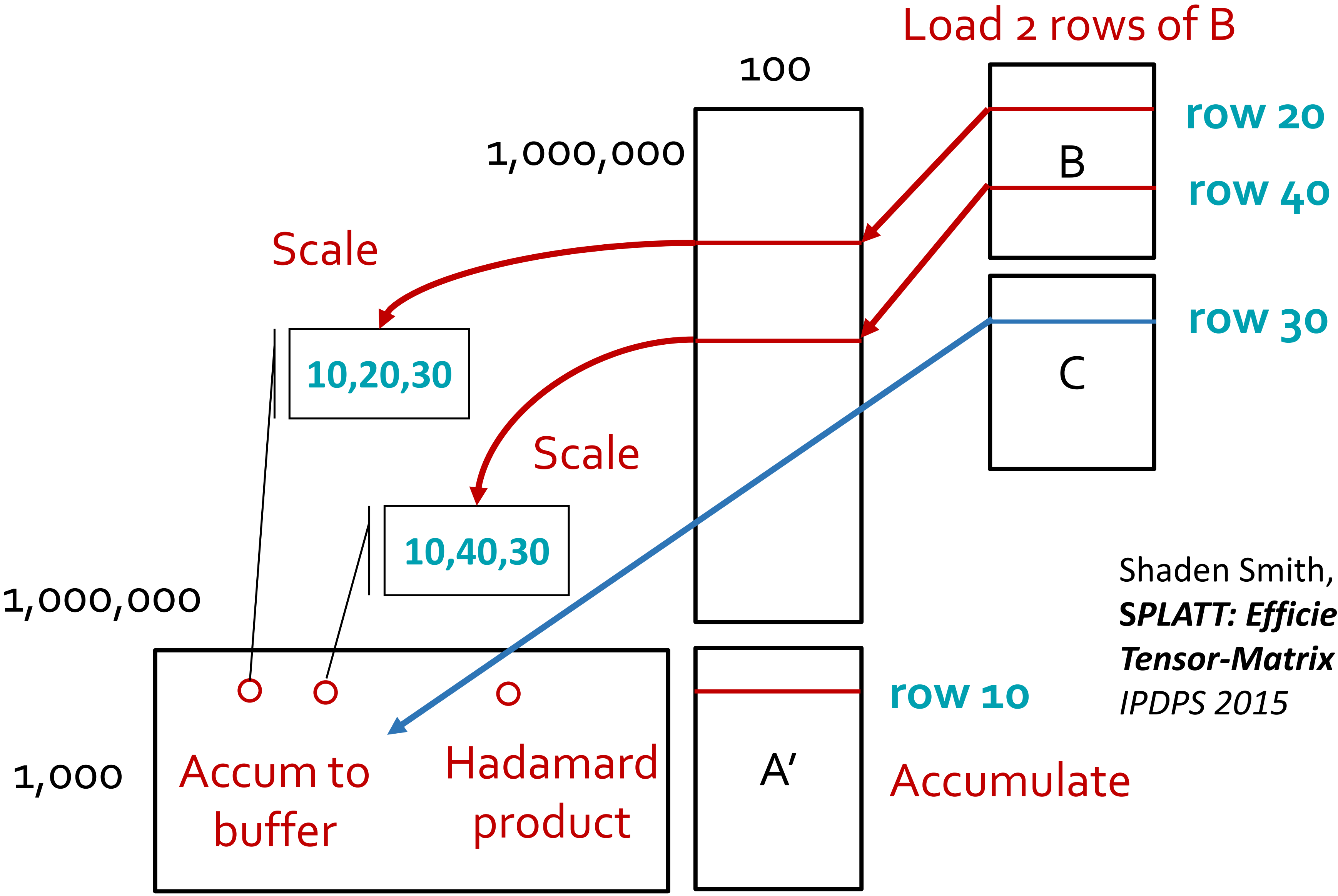
Hadamard product with buffer



Accumulate to destination matrix (A′)



This is called compressed sparse fiber (CSF)



Shaden Smith, et al., ***SPLATT: Efficient and Parallel Sparse Tensor-Matrix Multiplication,*** *IPDPS 2015*

Claimed Savings by others

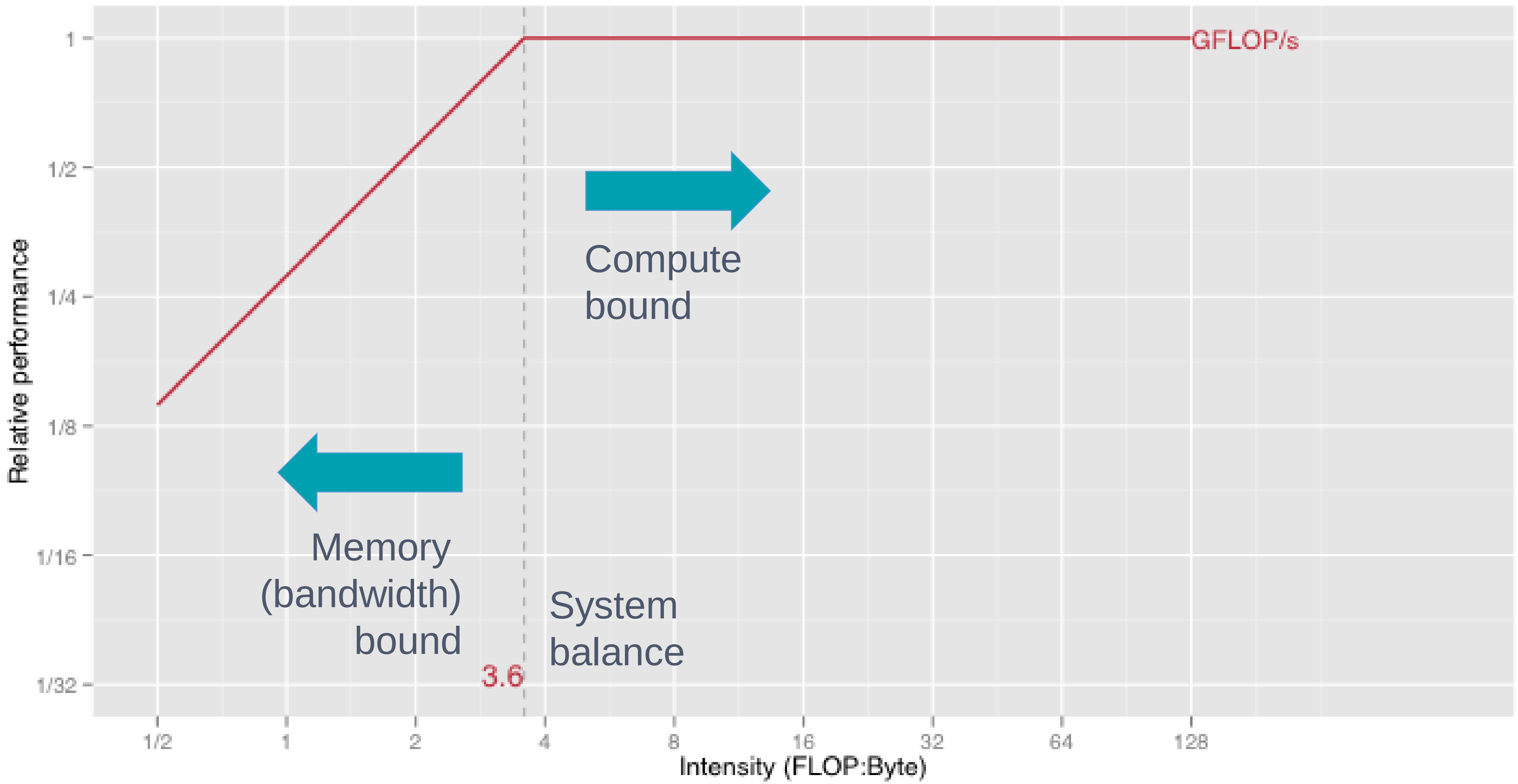
m = # of non-zeros
P = # of non-empty fibers
R = rank

- Naïve COO kernel
  - Regular: **3 * m * R** flops (2mR for initial product + scale, mR for accumulation)
- CSF
  - **2R(m + P)** flops, P is # of non-empty fibers
  - typically p <<< m
- DFacTo
  - Formulates MTTKRP as SpMV
  - Each column is computed independently via 2 SpMV
  - **2R(m + P)** flops
- GigaTensor
  - MapReduce
  - Increased parallelism vs. more flops
  - **5mR** flops

Does this make sense?

- Sparse computations are memory bandwidth-bound
- SPLATT tries cache blocking through expensive hypergraph partitioning (without much success)

Roofline model visualized (for an old Intel Nehalem CPU)

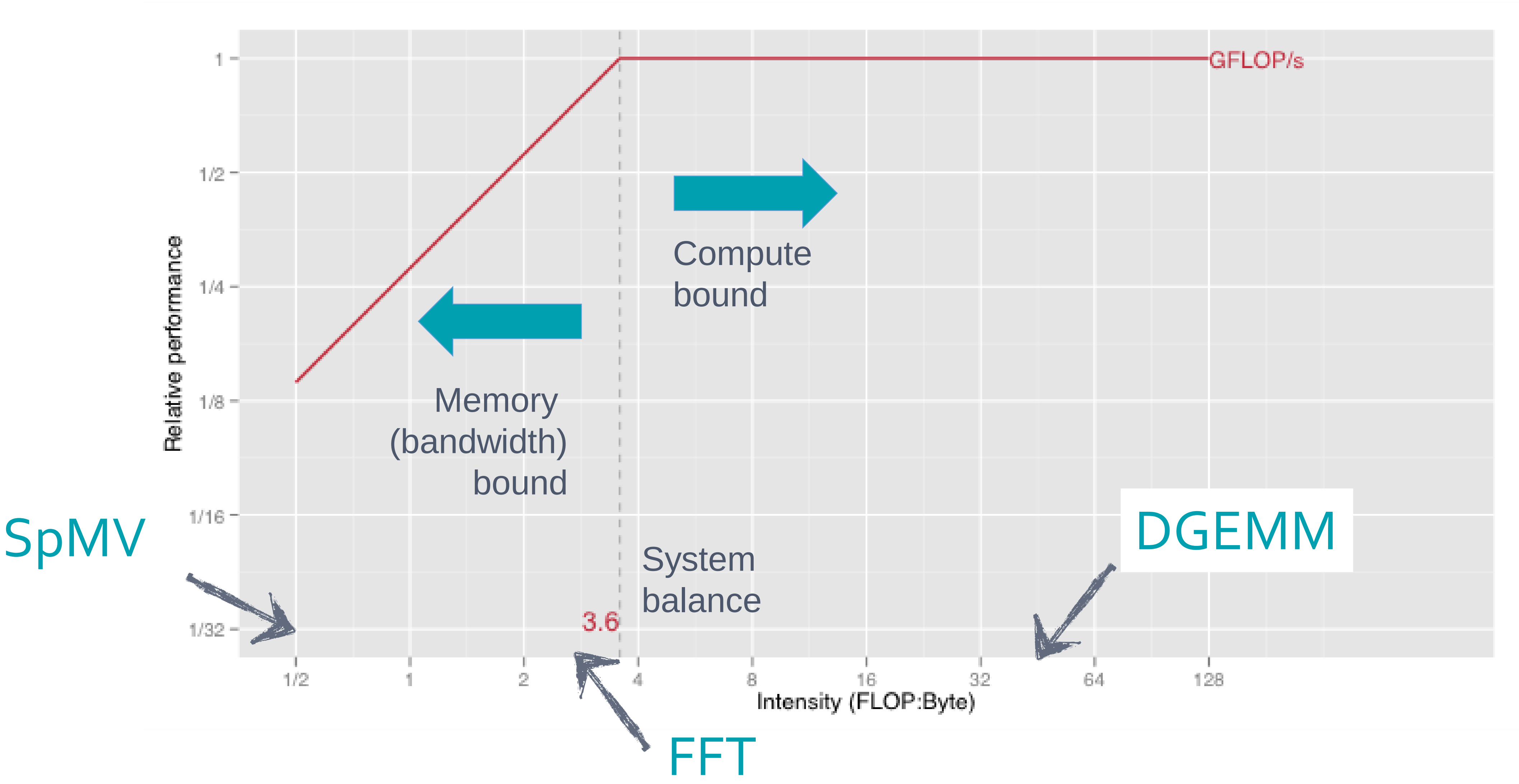


Compute bound

Memory (bandwidth) bound

System balance

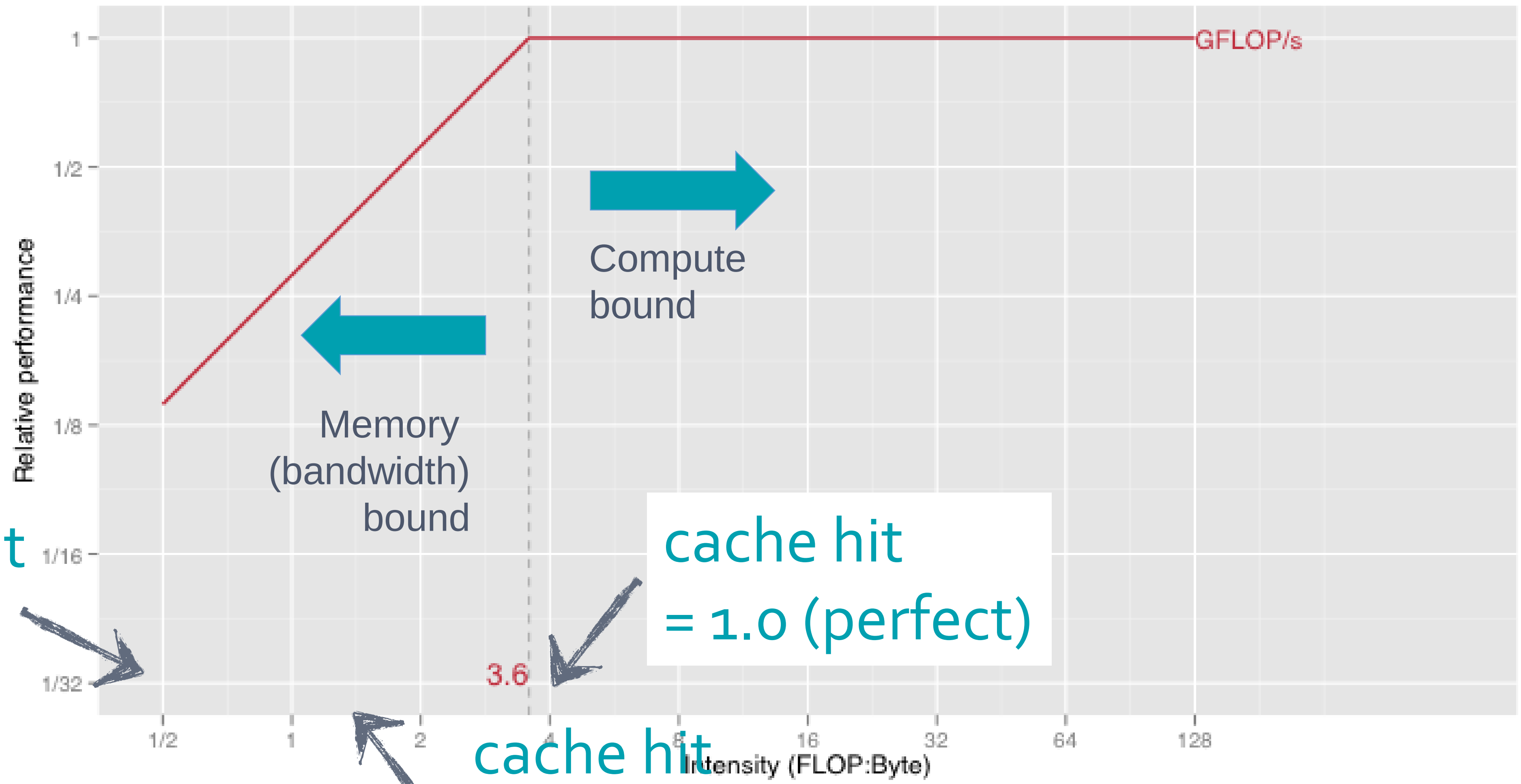
Commonly used scientific kernels



Roofline model applied to MTTKRP

- Sparse computations are memory bandwidth-bound
- Let's calculate the # of flops and # of bytes and compare
  - Flops: W = **$2R(m + P)$**
  - Bytes: Q = **$2m$** (value + mode-2 index) + **$2P$** (mode-3 index + mode-3 pointer)
    + **$(1-\alpha)Rm$** (mode-2 factor) + **$(1-\alpha)RP$** (mode-3 factor)
- Arithmetic Intensity
  - Ratio of work to communication **$I = W/Q$**
  - $I = W / (Q * 8 \text{ Bytes})$ = **$R / (8 + 4R(1-\alpha))$**

Arithmetic intensity of MTTKRP with rank = 32



Compute bound

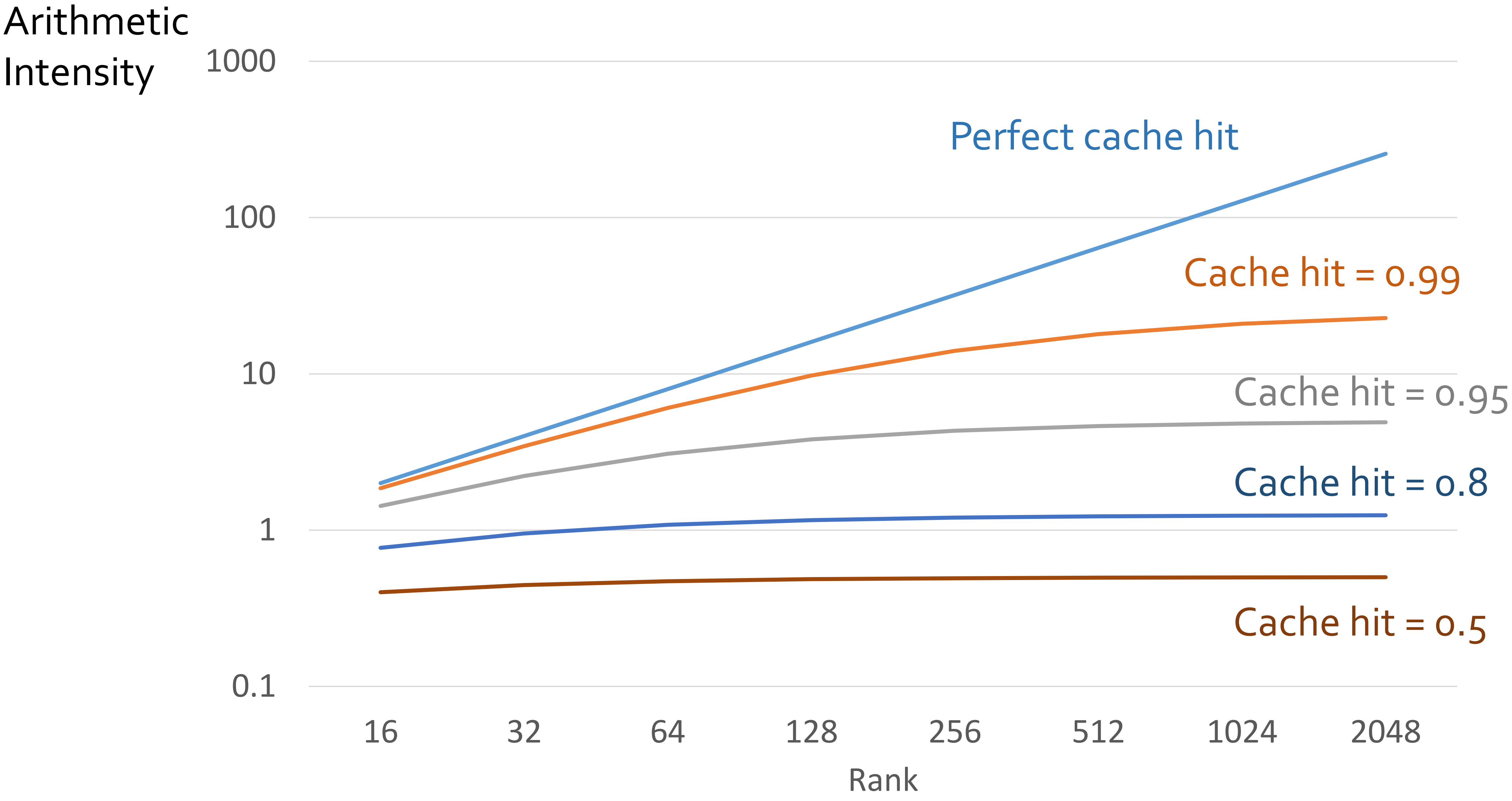
Memory (bandwidth) bound

cache hit = 0.5

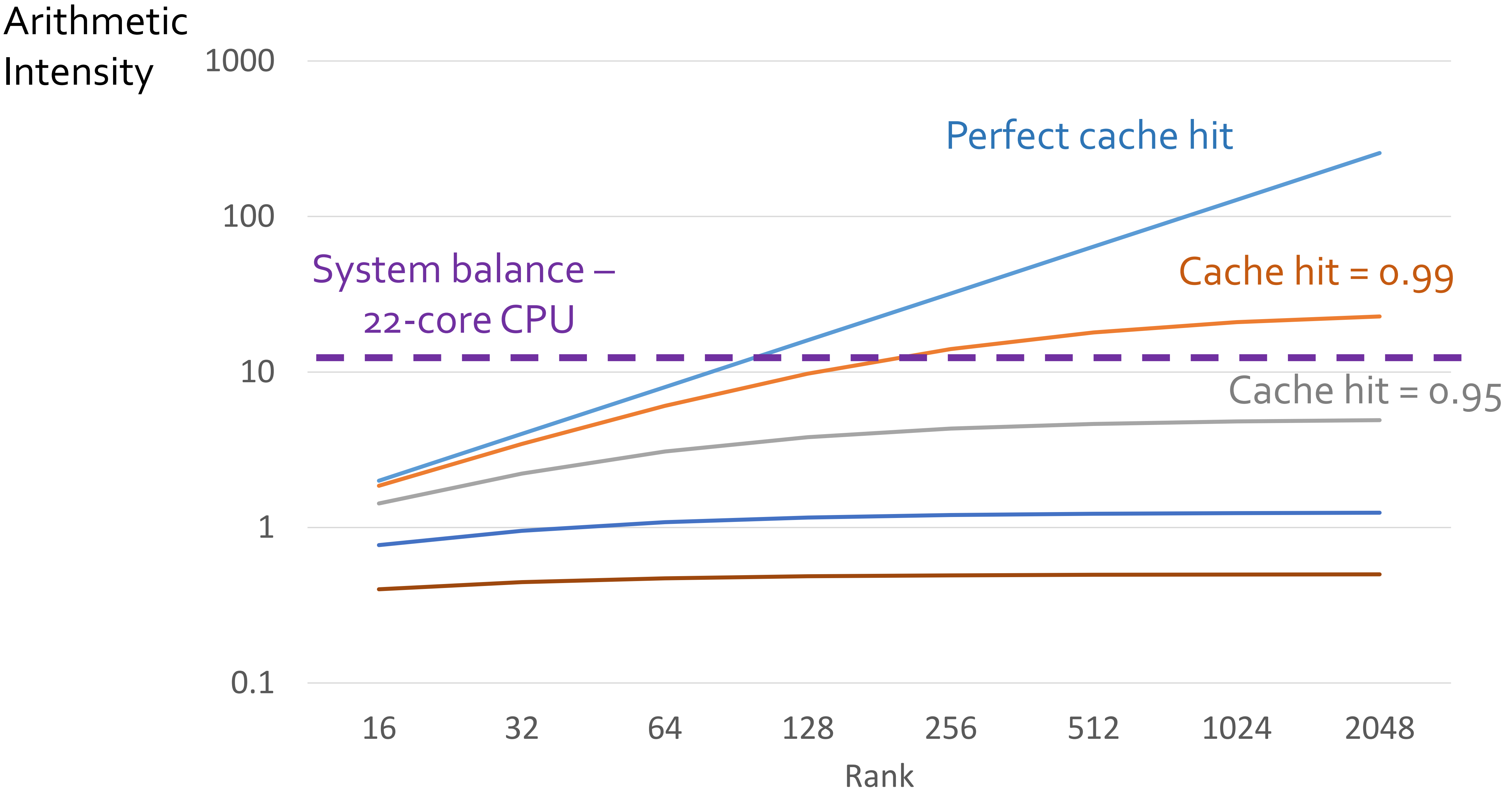
cache hit = 1.0 (perfect)

cache hit = 0.9

Arithmetic intensity vs. rank for various cache hit rates



Arithmetic intensity vs. system balance (on the latest CPU)



Our initial conclusion from a theoretical point of view

- On recent systems, MTTKRP is **likely** memory-bound
  - Even with a perfect cache hit rate, MTTKRP should be memory-bound on lower ranks
  - If we fail to get good cache re-use, MTTKRP will most likely be memory bound for any rank

A pressure point analysis reveals the bottleneck

- Pressure point analysis
  - Probe potential bottlenecks by creating and eliminating instructions/data access
  - If we suspect that # of registers is the bottleneck, try increasing/decreasing their usage to see if the exec. time changes.
  - Inline assembly to prevent dead code elimination (DCE)

Kenneth Czechowski, **Performance Analysis Using the Pressure Point Analysis**, PhD dissertation

A pressure point analysis reveals the bottleneck

Time	Pressure point
2.6	Baseline ($2R(m + P)$ flops)

Using COO instead of CSF only increases exec. time by < 2%

Time	Pressure point
2.6	Baseline ($2R(m + P)$ flops)
2.64	Move flops to inner loop ($3 * m * R$ flops)

Increasing flops only changes time by < 2%

Removing access to C (accessed once per fiber): exec. time down by 7%

Time	Pressure point
2.6	Baseline (2R(m + P) flops)
2.64	Move flops to inner loop (3 * m * R flops)
2.43	Access to C removed

Removing per-fiber access to matrix C has a bigger impact than increasing flops

Suspicion confirmed: Memory access to B is the bottleneck

Time	Pressure point
2.6	Baseline (2R(m + P) flops)
2.64	Move flops to inner loop (3 * m * R flops)
2.43	Access to C removed
1.81	Access to B limited to L1 cache

Limiting our suspect has a huge impact

Completely removing it give us an extra 6% - why?

Time	Pressure point
2.6	Baseline (2R(m + P) flops)
2.64	Move flops to inner loop (3 * m * R flops)
2.43	Access to C removed
1.81	Access to B limited to L1 cache
1.63	Access to B removed completely

Eliminating it completely gives us an extra 6% boost

Conclusions from our empirical analysis

- Flops aren’t the issue
- Bottlenecks
  1. Data access to B
  2. Load instructions

Cache/register blocking should help alleviate these bottlenecks

- Flops aren’t the issue
- Bottlenecks
  1. Data access to B ⟶ cache blocking
  2. Load instructions ⟶ register blocking

Our baseline implementation

```
procedure mttkrp ($\mathcal{X} \in \mathbb{R}^{I\times J\times K}$, R)
  for i ← 0 to I do  // for each row
    for j ←  i_ptr[i] to i_ptr[i+1] do // for each fiber
      for k ← p_ptr[j] to p_ptr[j+1] do // for each nz in fiber
        for r ← 0 to R do // go through entire rank
          buffer[r] += vals[k] * B[j_index[k]][r] // buffer
      for r ← 0 to R do
        A[i][r] += buffer[r] * C[k_index[j]][r] // accumulate
end procedure
```

Our baseline implementation

```
procedure mttkrp ($\mathcal{X} \in \mathbb{R}^{I\times J\times K}$, R)
  for i ← 0 to I do  // for each row
    for j ←  i_ptr[i] to i_ptr[i+1] do // for each fiber
      for k ← p_ptr[j] to p_ptr[j+1] do // for each nz in fiber
        for r ← 0 to R do // go through entire rank
          buffer[r] += vals[k] * B[j_index[k]][r] // buffer
      for r ← 0 to R do
        A[i][r] += buffer[r] * C[k_index[j]][r] // accumulate
end procedure
```

3 LD instructions

Replace buffers with registers

```
procedure mttkrp ($\mathcal{X} \in \mathbb{R}^{I \times J \times K}$, R)
  for i ← 0 to I do  // for each row
    for j ←  i_ptr[i] to i_ptr[i+1] do // for each fiber
      for r ← 0 to R do in 16 increments
        for k ← p_ptr[j] to p_ptr[j+1] do // for each nz in fiber
          registers += vals[k] * B[j_index[k]][r] // buffer
        A[i][r] += registers * C[k_index[j]][r] // accumulate
end procedure
```

Replace buffers with registers

```
procedure mttkrp ($\mathcal{X} \in \mathbb{R}^{I\times J\times K}$, R)
  for i ← 0 to I do  // for each row
    for j ←  i_ptr[i] to i_ptr[i+1] do // for each fiber
      for r ← 0 to R do in 16 increments
        for k ← p_ptr[j] to p_ptr[j+1] do // for each nz in fiber
          registers += vals[k] * B[j_index[k]][r] // buffer
        A[i][r] += registers * C[k_index[j]][r] // accumulate
end procedure
```

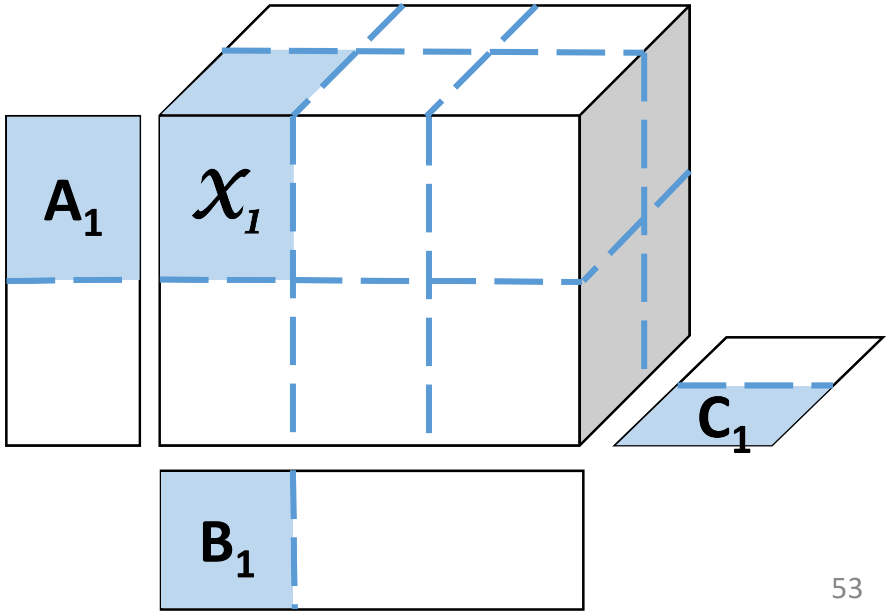
2 LD instructions

We use n-D blocking (intuitive) and rank blocking (less intuitive)

- Multi-dimensional blocking
- Rank blocking

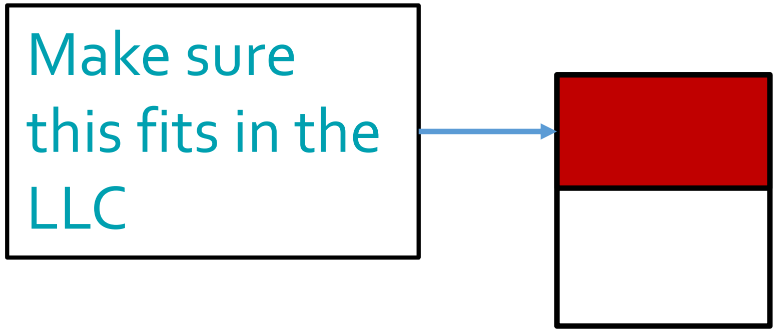
We use n-D blocking (intuitive) and rank blocking (less intuitive)

- Multi-dimensional blocking
  - 3D blocking – maximize re-use of both matrix B and C
- Rank blocking

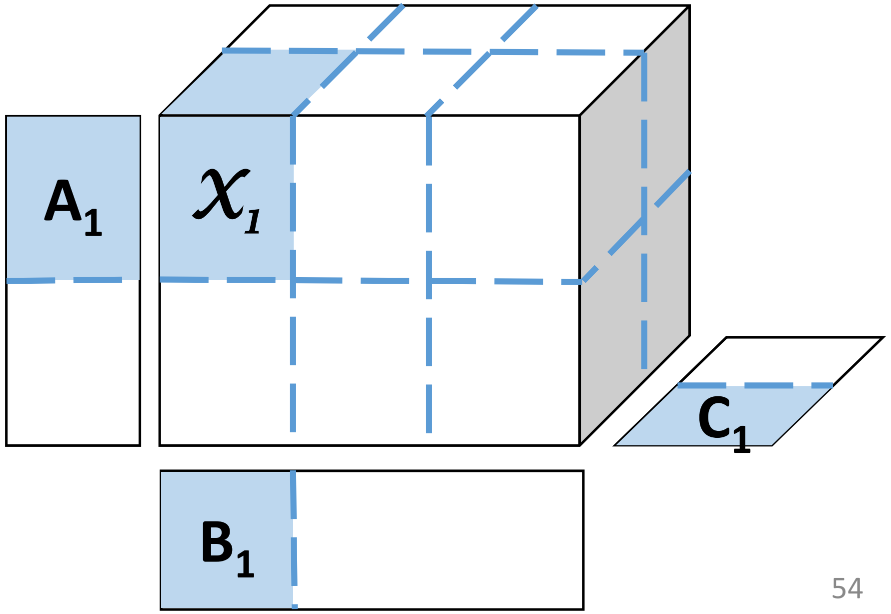


We use n-D blocking (intuitive) and rank blocking (less intuitive)

- Multi-dimensional blocking
  - 3D blocking – maximize re-use of both matrix B and C
- Rank blocking

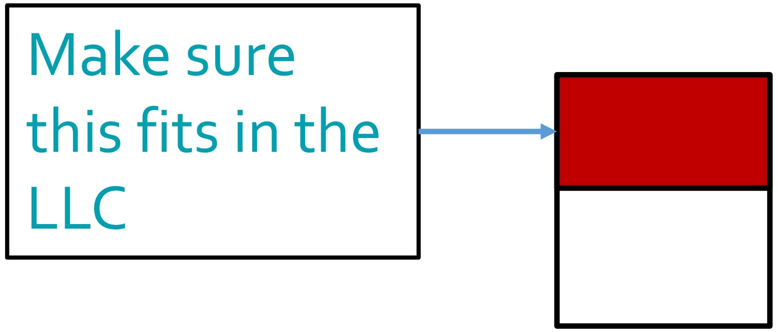


Make sure this fits in the LLC

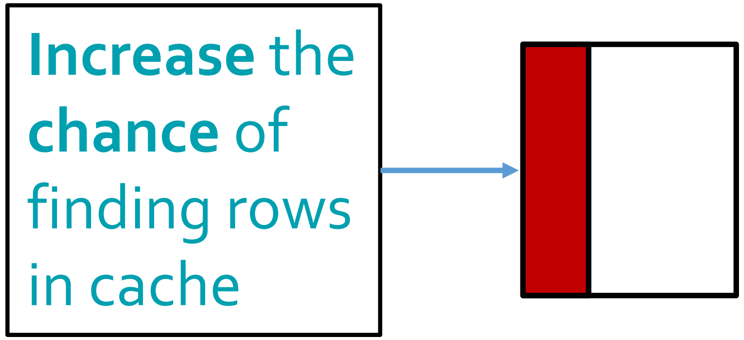


We use n-D blocking (intuitive) and rank blocking (less intuitive)

- Multi-dimensional blocking
  - 3D blocking – maximize re-use of both matrix B and C
- **Rank blocking**
  - **Agnostic to tensor sparsity**
  - **Very little change to the code required**

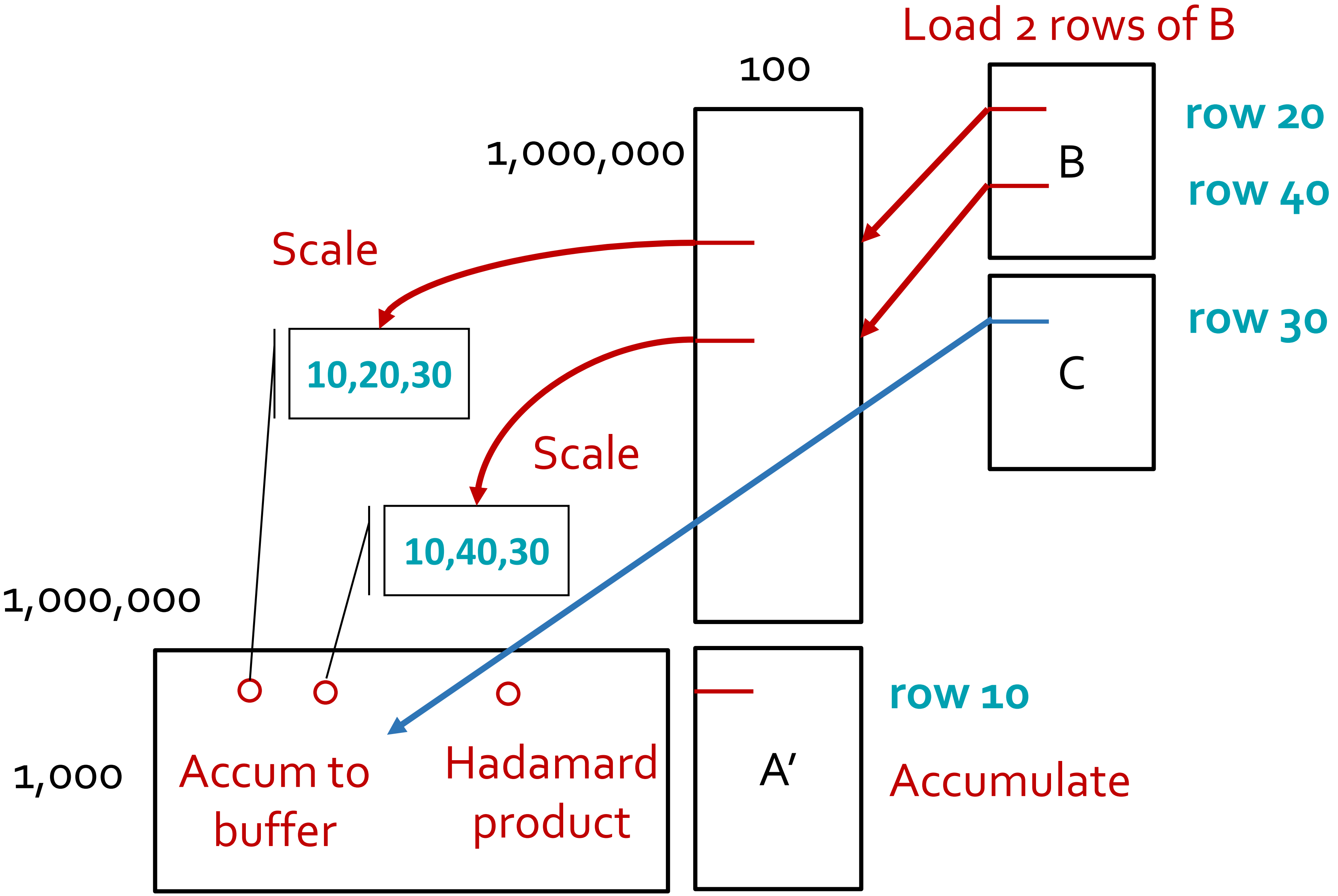


Make sure this fits in the LLC

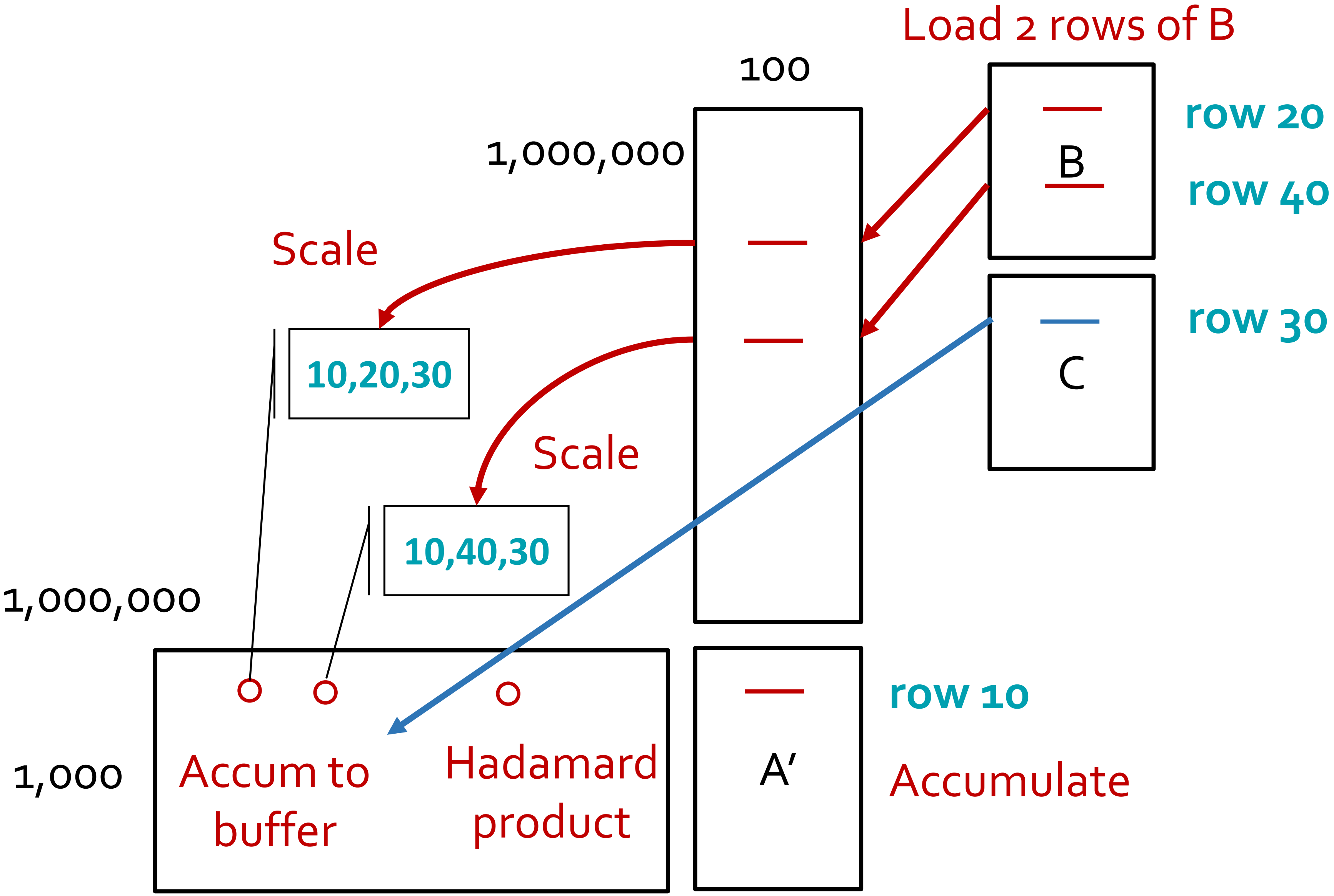


Increase the **chance** of finding rows in cache

Rank blocking visualized…



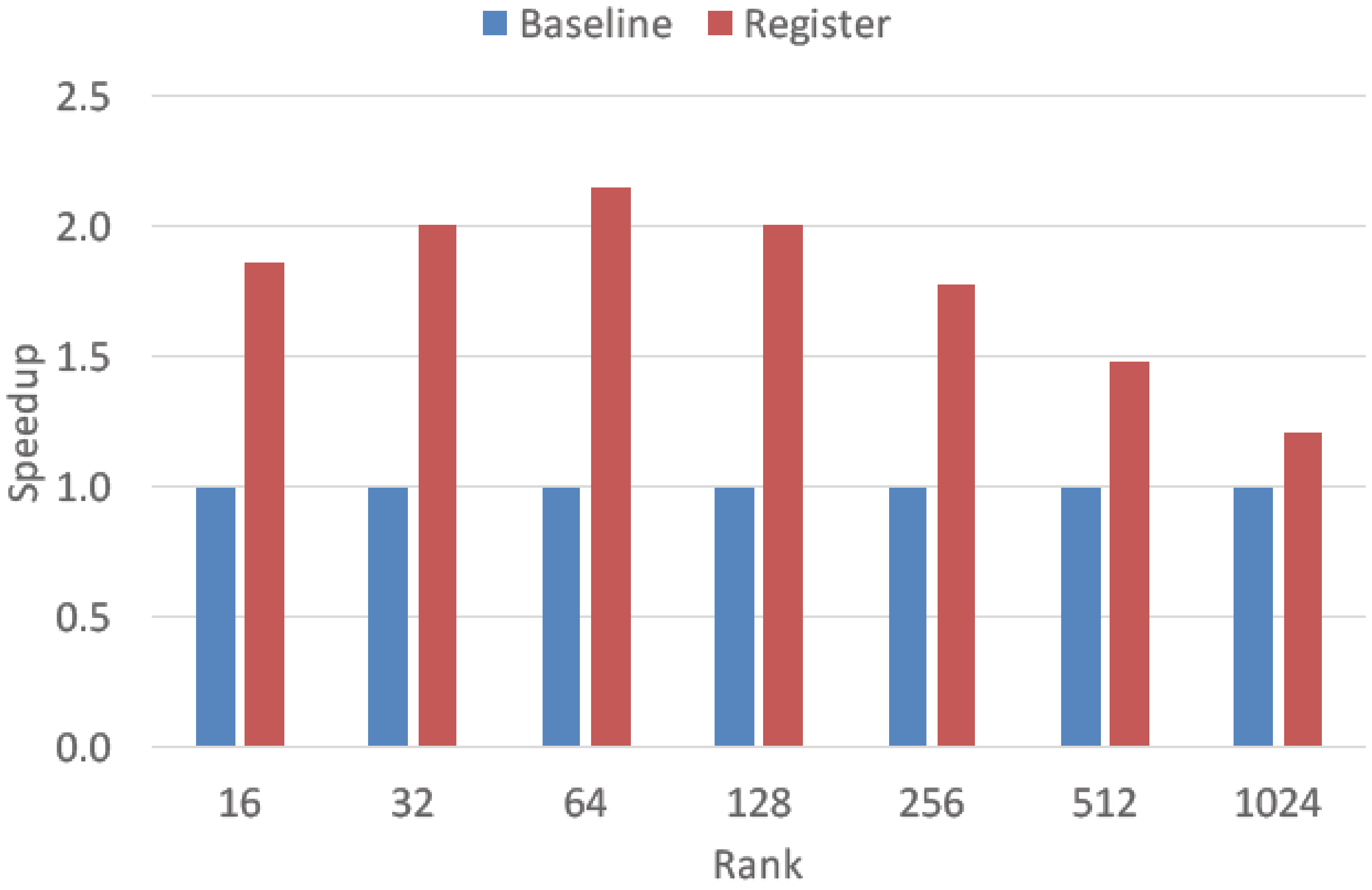
Rank blocking visualized…



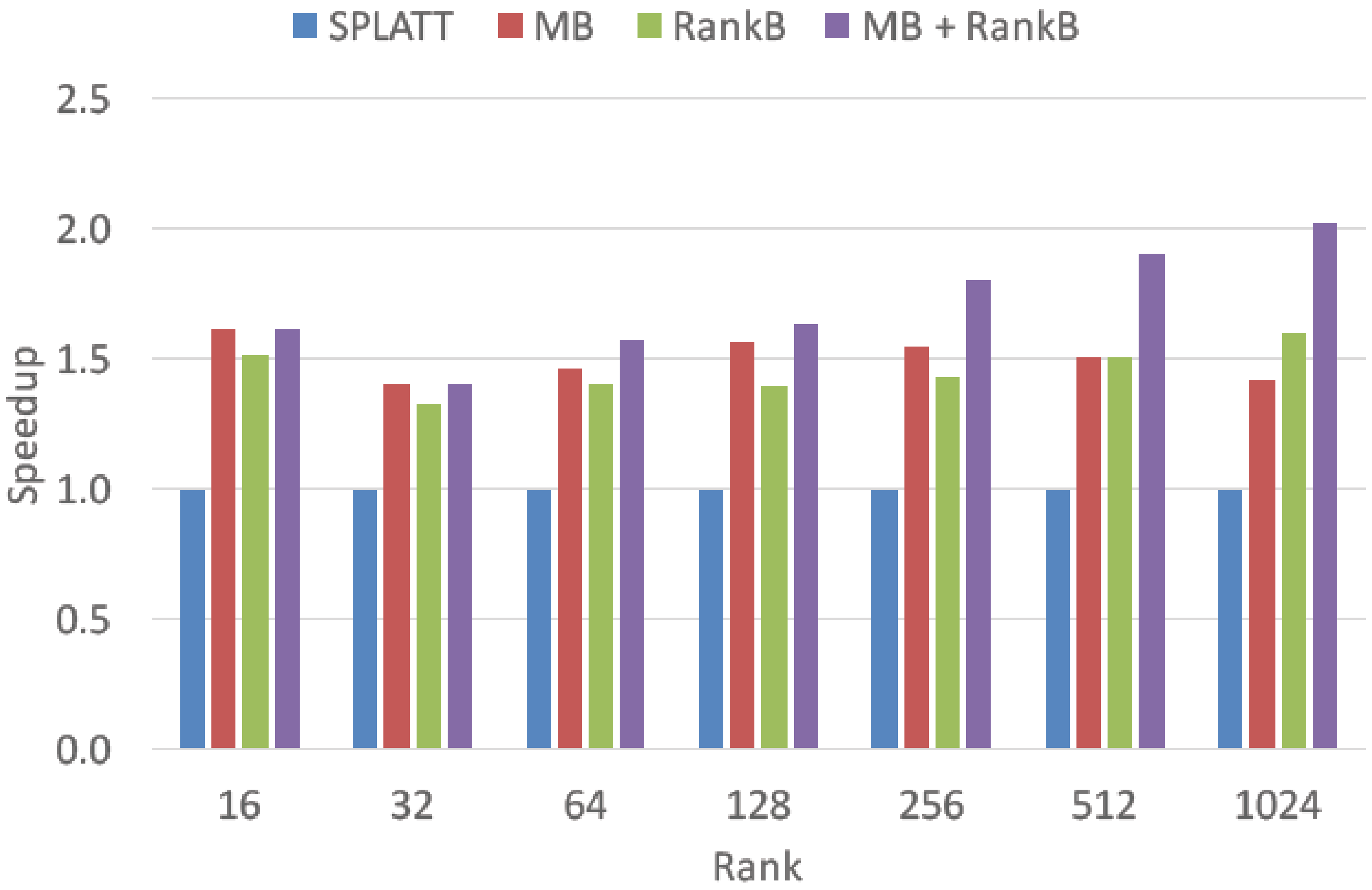
Performance Summary

Data set	Dimensions	nnz	Sparsity	# fibers	Speedup
Poisson1	256×256×256	1.5M	8.8e-2	54K	3.1x
Poisson2	2K×16K×2K	121M	1.9e-3	2.5M	2.5×
Poisson3	2K×16K×2K	6.4M	1.0e-4	830K	2.0×
Netflix	480K×18K×80	80M	1.2e-4	5M	2.1×
NELL-2	12K×9K×29K	77M	2.4e-5	21M	2.2×

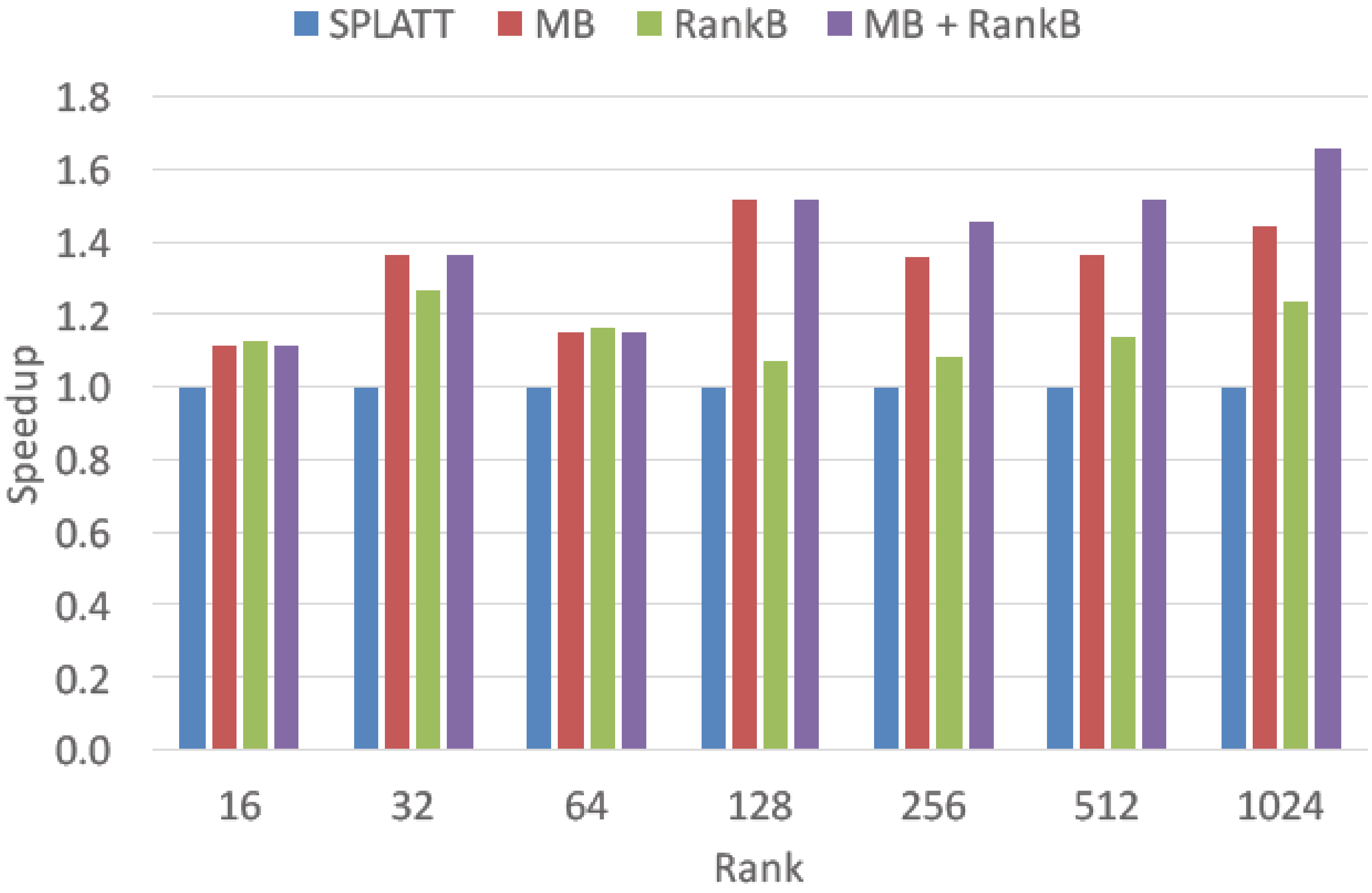
Register blocking yields large speedups for small data sets



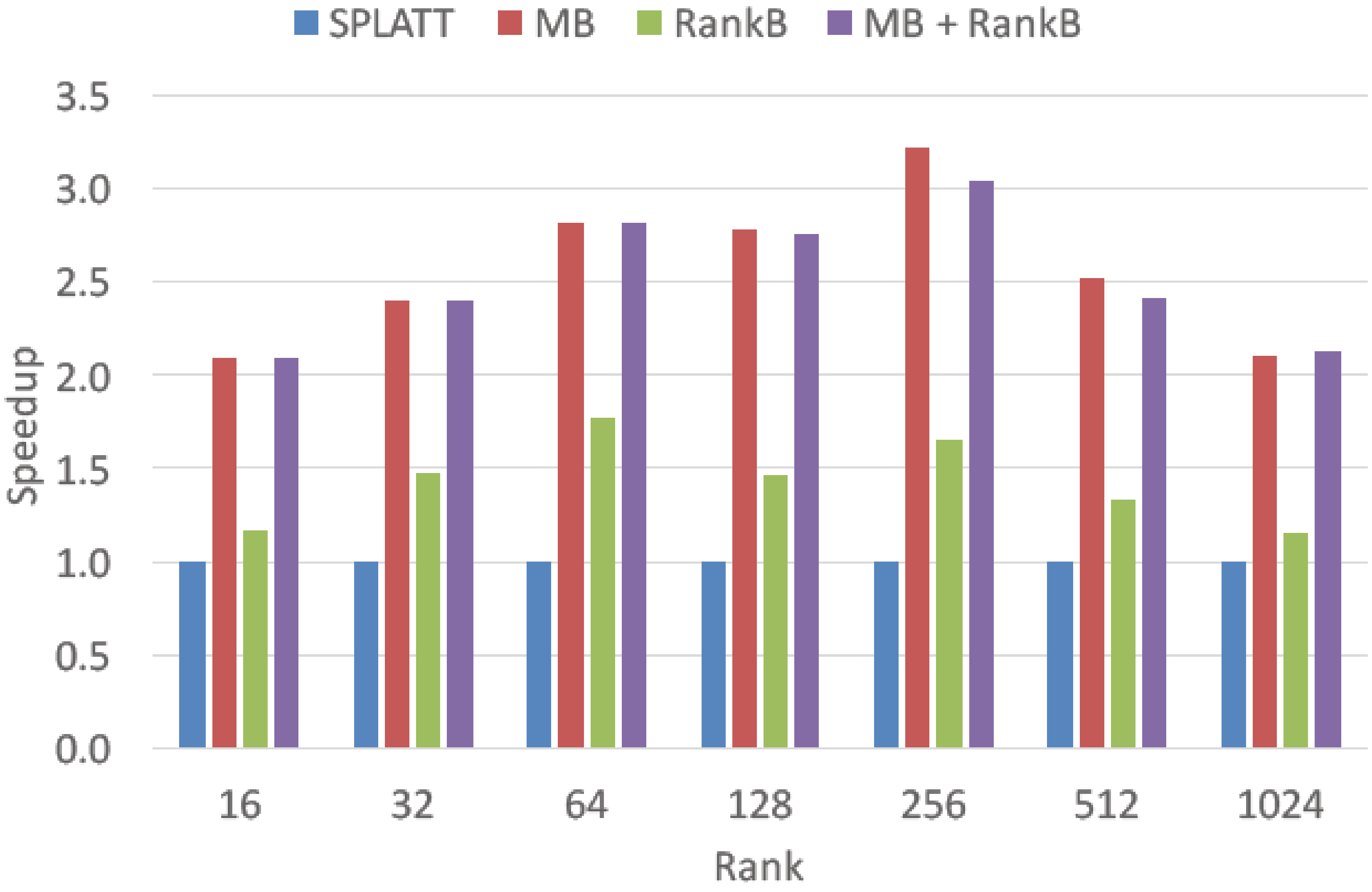
Poisson 2 – sparsity = 1.9e-3



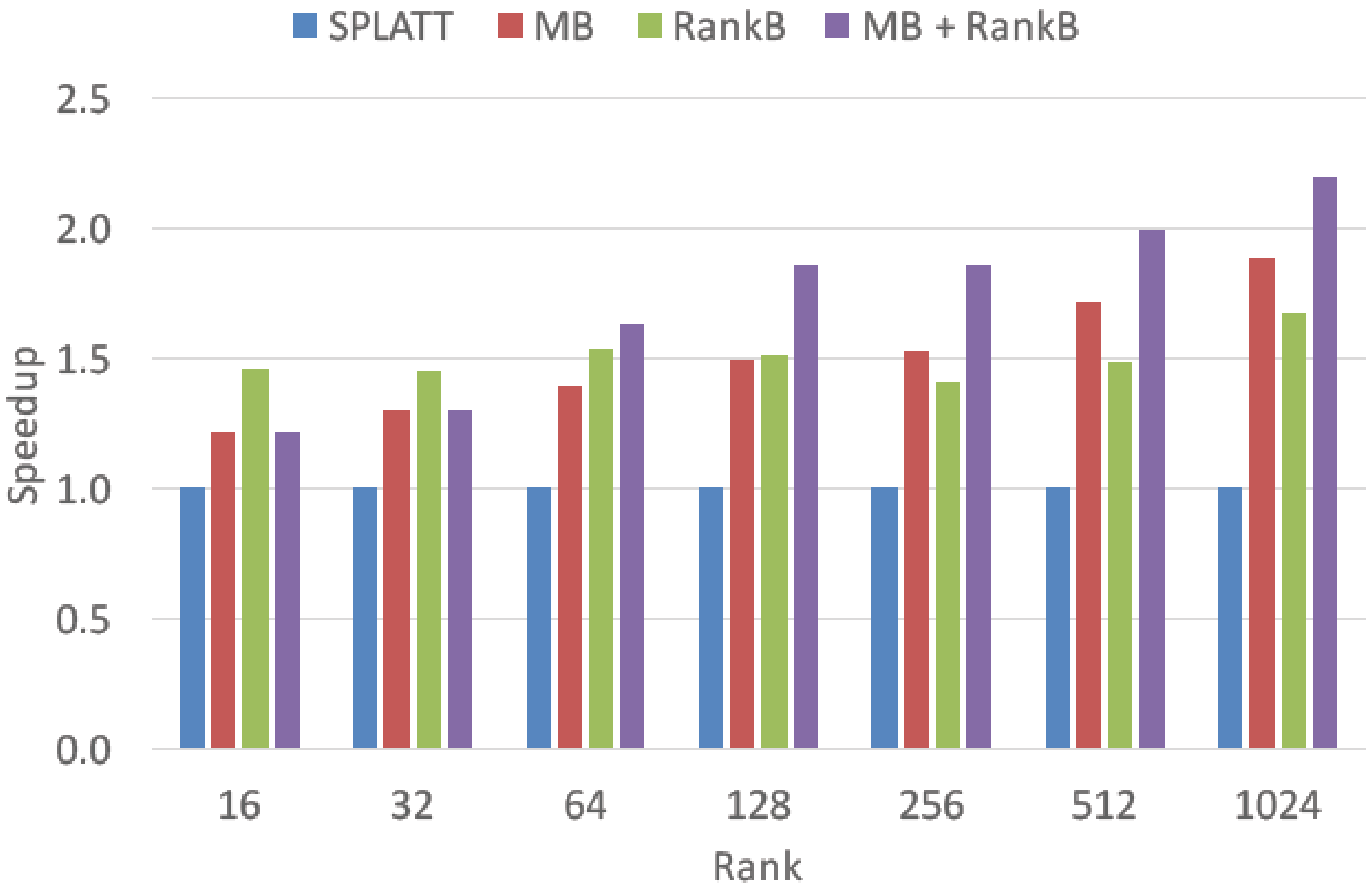
Poisson 3 – sparsity = 1.0e-4



Netflix – sparsity = 1.2e-4



NELL – sparsity = 2.4e-5



Distributed rank blocking shows better scalability

Nodes	NELL2					Netflix				
	SPLATT	3D grid	3D time	4D grid	4D time	SPLATT	3D grid	3D time	4D grid	4D time
1	1.028	1x1x2	0.718	1x1x1x2	0.826	3.025	2x1x1	1.554	1x1x1x2	1.447
2	0.54	1x1x4	0.367	1x1x1x4	0.423	1.158	4x1x1	0.727	1x1x1x4	0.720
4	0.286	2x1x4	0.208	1x1x1x8	0.217	0.519	8x1x4	0.403	1x1x1x8	0.401
8	0.138	2x2x4	0.107	1x1x1x16	0.124	0.256	16x1x1	0.194	1x1x1x16	0.190
16	0.087	2x2x8	0.058	1x1x2x16	0.065	0.113	32x1x1	0.103	1x1x2x16	0.100
32	0.056	4x2x8	0.043	1x1x4x16	0.034	0.083	31x2x1	0.056	1x1x4x16	0.055
64	0.03	4x4x8	0.028	2x1x4x16	0.022	0.048	64x2x1	0.037	2x1x4x16	0.030

The take-away from the section

- There was a lack of clear understanding about performance bottlenecks in tensor decomposition
    - We show that the key computation is LD and memory-bound
- Using various blocking techniques mitigate these bottlenecks
- Our optimizations demonstrate significant speedup over synthetic and real-world data for both shared-memory and distributed implementations
    - We use 3D and rank blocking strategies to achieve up to 3.2x speedup on real world-data and 2.0x on synthetic

Future Work

- Extending this work to do performance modeling
  - Correlate tiling/blocking size to cache hit rate
  - Take advantage of block structures
  - Fiber/slice/cube/etc. permutation – new storage formats for tensors (a la SpMV)

Q & A

I am currently on the academic job market! Please email me at jee@gatech.edu or visit http://jeewhanchoi.com for my application materials