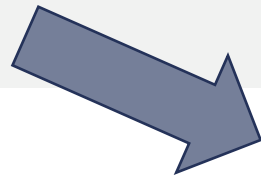


Erlangen Regional
Computing Center



Case Study: Dense Matrix-Vector Multiplication

```
do c = 1 , NC
  do r = 1 , NR
    y(r) = y(r) + A(r,c) * x(c)
  enddo
enddo
```

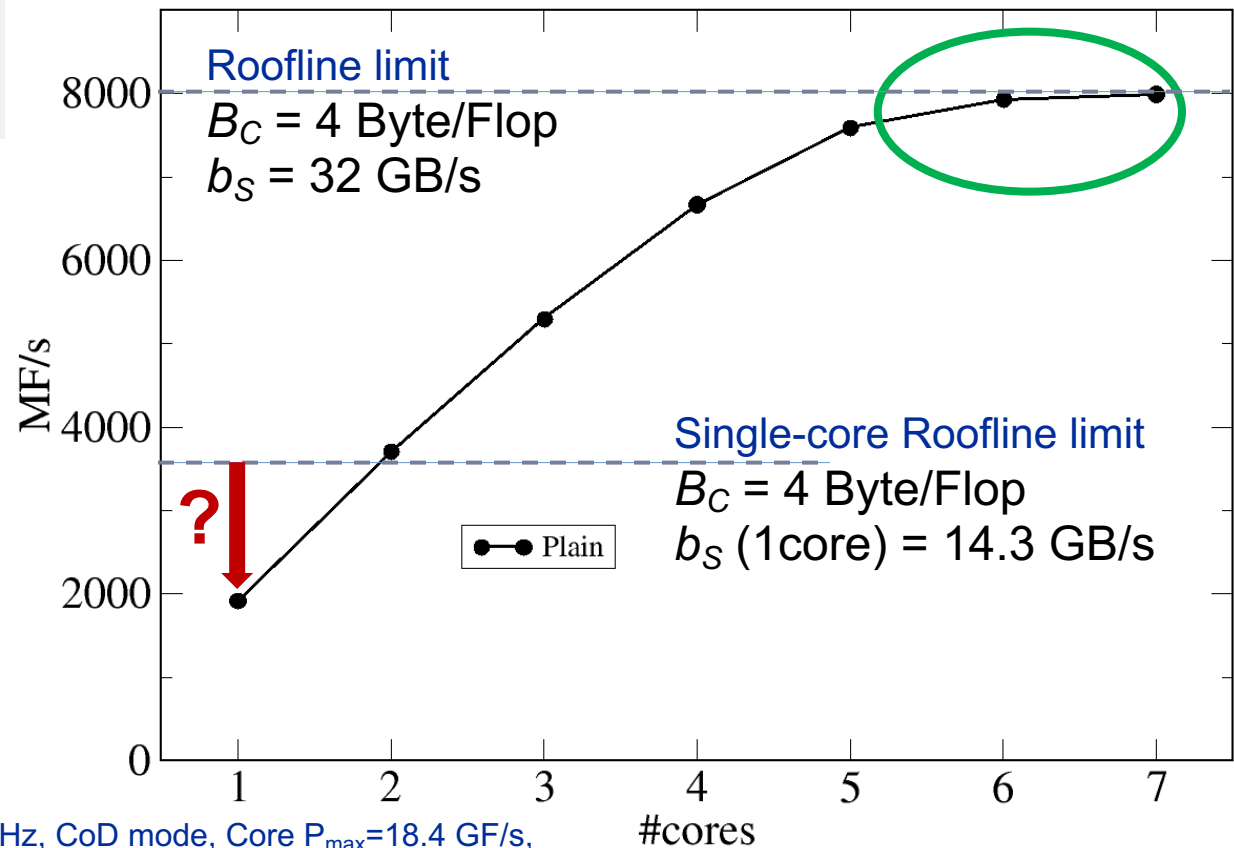


```
do c = 1 , NC
  tmp = x(c)
  do r = 1 , NR
    y(r) = y(r) + A(r,c) * tmp
  enddo
enddo
```

dMVM scaling w/ OpenMP

```
!$omp parallel do reduction(+:y)
do c = 1 , NC
  do r = 1 , NR
    y(r) = y(r) + A(r,c) * x(c)
  enddo ; enddo
!$omp end parallel do
```

NR=40,000; NC=10,000



Intel Xeon E5 2695 v3 (Haswell-EP), 2.3 GHz, CoD mode, Core $P_{\max}=18.4$ GF/s,
Caches: 32 KB / 256 KB / 35 MB, PageSize: 2 MB; ifort V15.0.1.133; $b_S = 32$ Gbyte/s

- Vectorization strategy: 4-way inner loop unrolling
- One register holds **tmp** in each of its 4 entries (“broadcast”)

```
do c = 1, NC
```

```
  tmp = x(c)
```

```
  do r = 1, NR, 4    ! R is multiple of 4
```

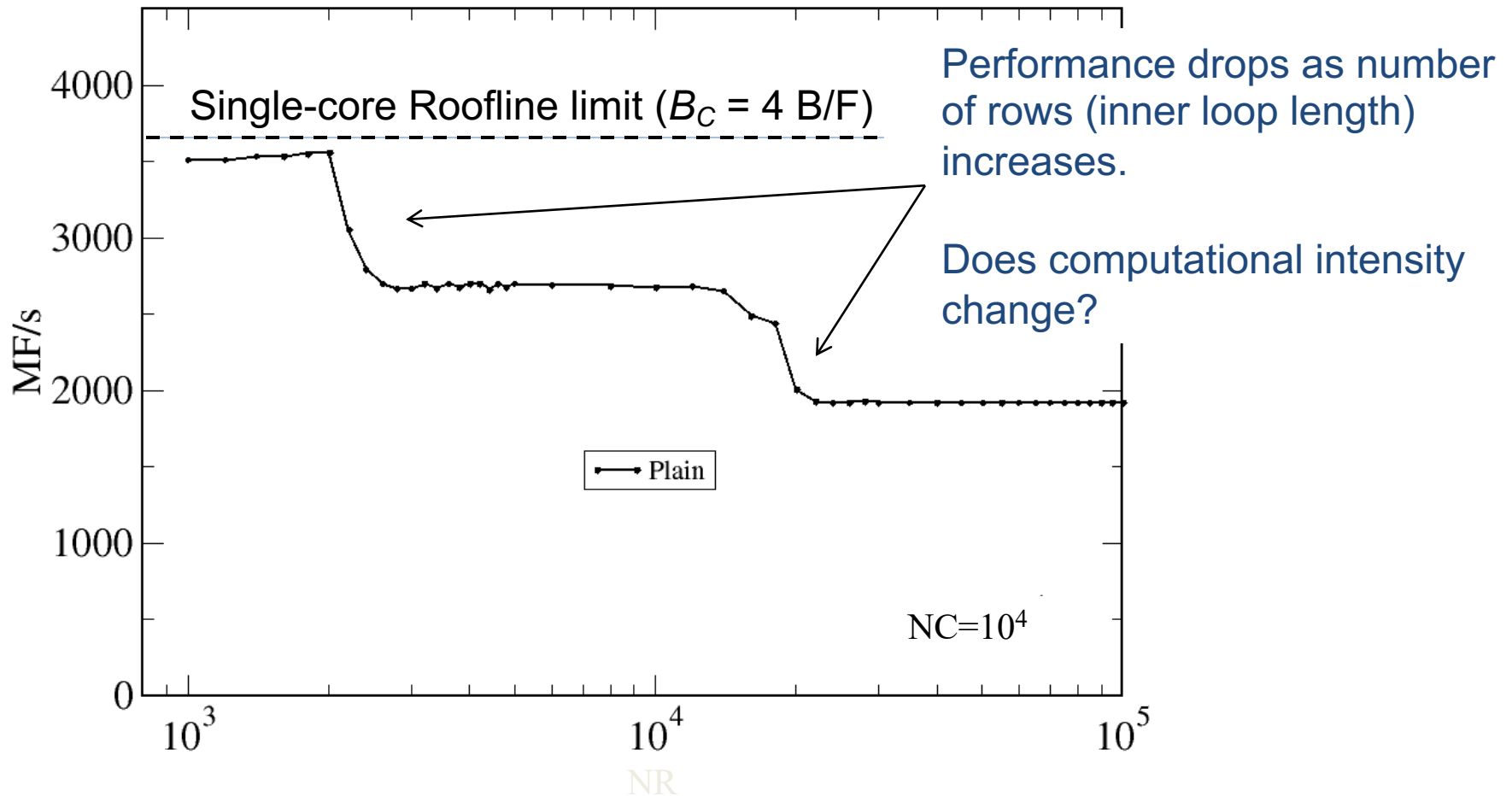
| | | | | | | |
|--------|---|--------|---|-----------|---|-----|
| y(r) | = | y(r) | + | A(r, c) | * | tmp |
| y(r+1) | = | y(r+1) | + | A(r+1, c) | * | tmp |
| y(r+2) | = | y(r+2) | + | A(r+2, c) | * | tmp |
| y(r+3) | = | y(r+3) | + | A(r+3, c) | * | tmp |

```
  enddo
```

```
endo
```

- Loop kernel requires/consumes 3 AVX registers
- Extra 3-way unrolling required to overcome ADD pipeline stalls

DMVM (DP) – Single core performance vs. column height



Intel Xeon E5 2695 v3 (Haswell-EP), 2.3 GHz, CoD mode, Core $P_{\max} = 18.4 \text{ GF/s}$,
Caches: 32 KB / 256 KB / 35 MB, PageSize: 2 MB; ifort V15.0.1.133; $b_S = 32 \text{ Gbyte/s}$

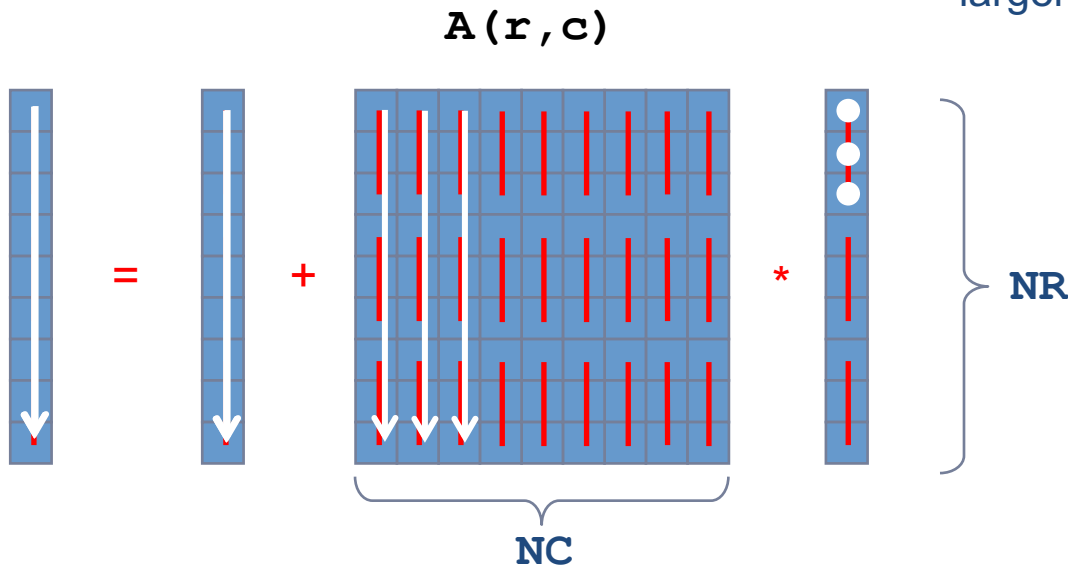
```
do c = 1 , NC
  tmp=x(c)
  do r = 1 , NR
    y(r)=y(r) + A(r,c) * tmp
  enddo
enddo
```

`tmp` stays in a register during inner loop

`A(:, :)` is loaded from memory – no data reuse

`y(:)` is loaded and stored in each outer iteration → for $c > 1$ update `y(:)` in cache

`y(:)` may not fit in innermost cache → more traffic from lower level caches for larger `NR`



Analysis: Distinguish code balance in memory (B_C^{mem}) from code balance in relevant cache level(s) ($B_C^{L3}, B_C^{L2}, \dots$)!

Code balance can be defined for any data path:

$$B_c^i = \frac{V_i}{W}$$

V_i = data volume over data path i

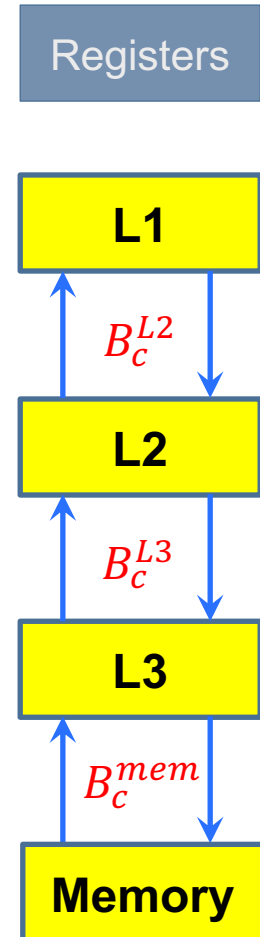
W = amount of work done with the data

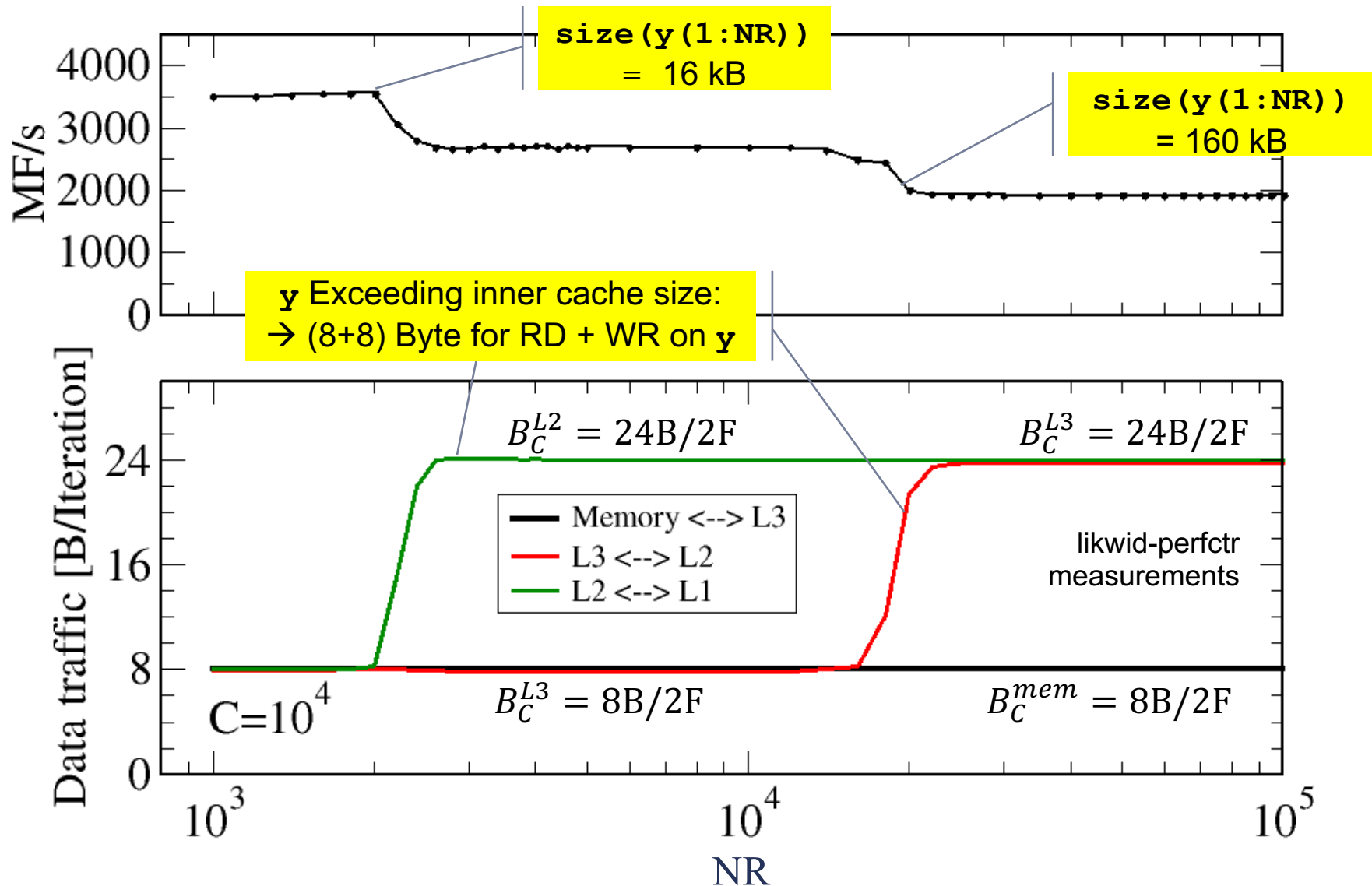
In principle, the Roofline model can be expressed for those multiple bottlenecks:

$$P = \min \left(P_{\max}, \min_i \left[\frac{b_s^i}{B_c^i} \right] \right)$$

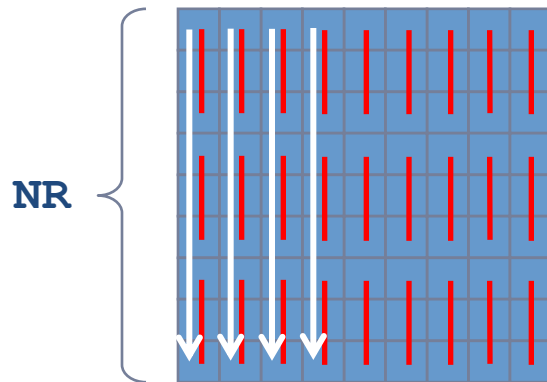
However, the **perfect overlap** condition is **invalid** for the single-core cache hierarchy

But code balance is still useful for *qualitative* analysis...



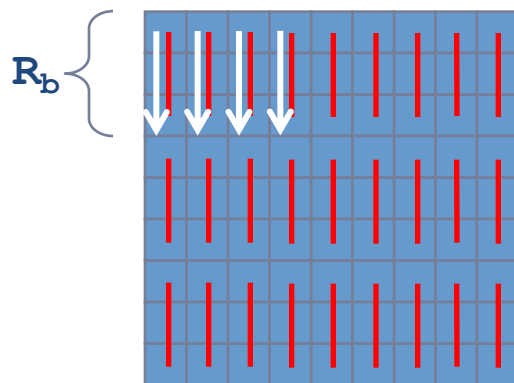


$A(r, c)$



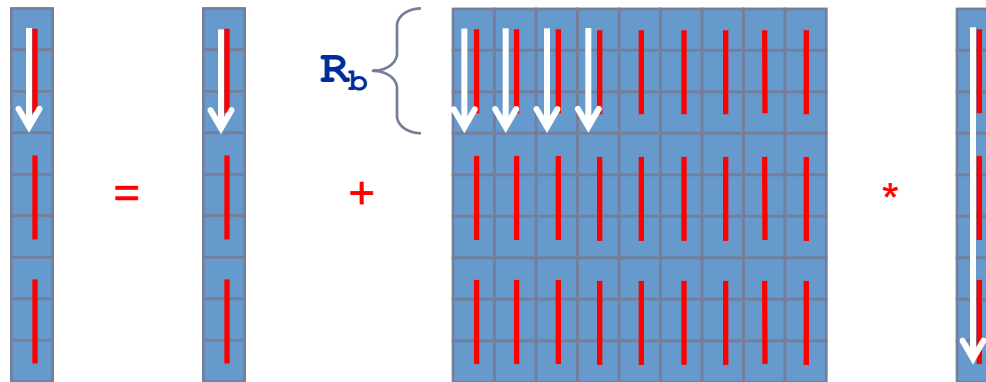
```
do c = 1 , NC
  tmp=x(c)
  do r = 1 , NR
    y(r)=y(r) + A(r,c) * tmp
  enddo
enddo
```

$y(:)$ may not fit into some cache
→ more traffic for lower level



```
do rb = 1 , NR , Rb
  rbS = rb
  rbE = min((rb+Rb-1) , NR)
  do c = 1 , NC
    do r = rbS , rbE
      y(r)=y(r) + A(r,c)*x(c)
    enddo
  enddo
enddo
```

$y(rbS:rbE)$ may fit into some cache if R_b is small enough
→ traffic reduction



LHS only updated once in some cache level if blocking is applied

Price: RHS is loaded multiple times instead of once!

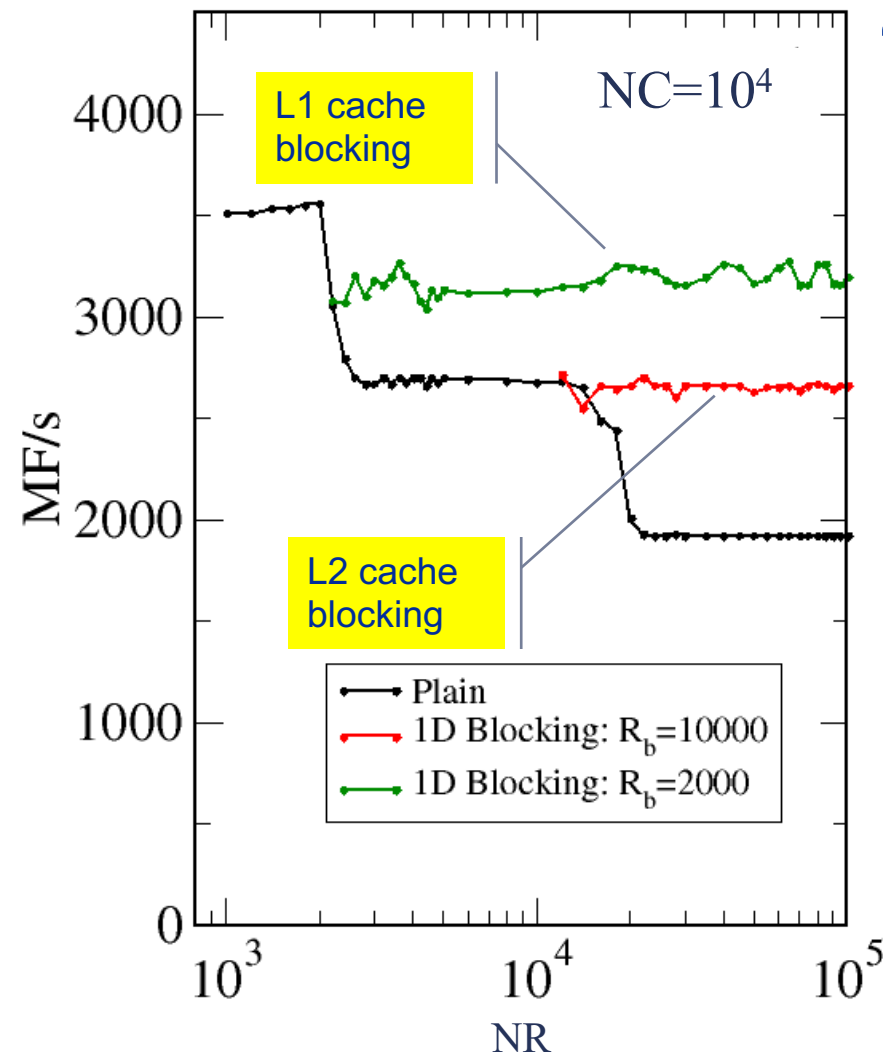
How often? $\rightarrow N_R / R_b$ times

RHS traffic: $N_C \times N_R / R_b$

LHS traffic: $2 \times N_R$

Matrix: $N_R \times N_C$

Without blocking: $N_R \times \left(\frac{N_C}{N_R} + 2N_C + N_R \right) \approx 3N_R^2$ if $N_R, R_b \gg 1$ and $N_C = N_R$

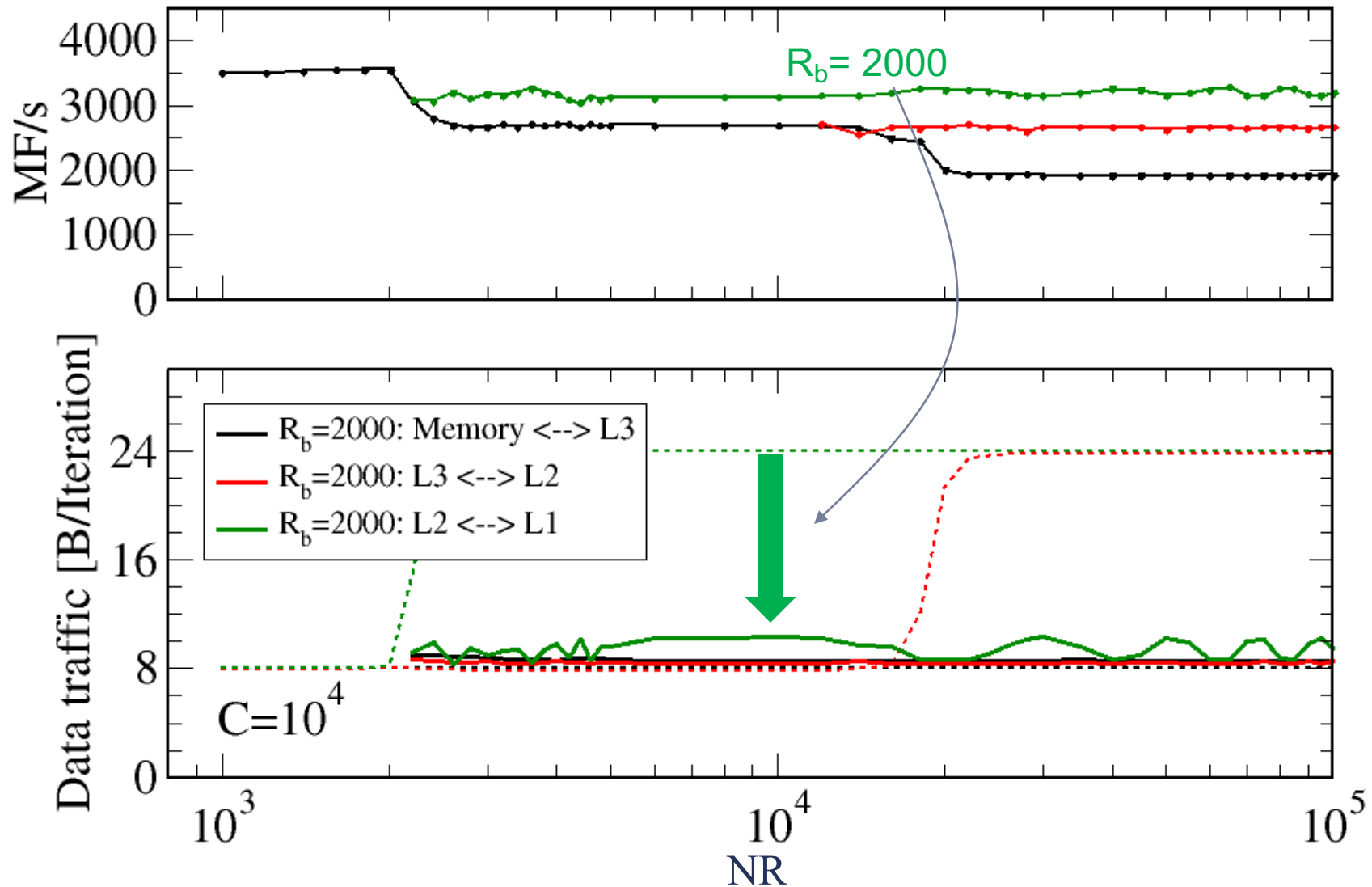


“1D blocking” for inner loop

Blocking factor $R_b \leftrightarrow$ cache level

```
do rb = 1 , NR ,  $R_b$   
  
  rbS = rb  
  rbE = min((rb+ $R_b$ -1) , NR)  
  
  do c = 1 , NC  
    do r = rbS , rbE  
      y(r) = y(r) + A(r,c)*x(c)  
    enddo  
  enddo  
  
enddo
```

→ Fully reuse subset of $y(rbS:rbE)$
from L1/L2 cache



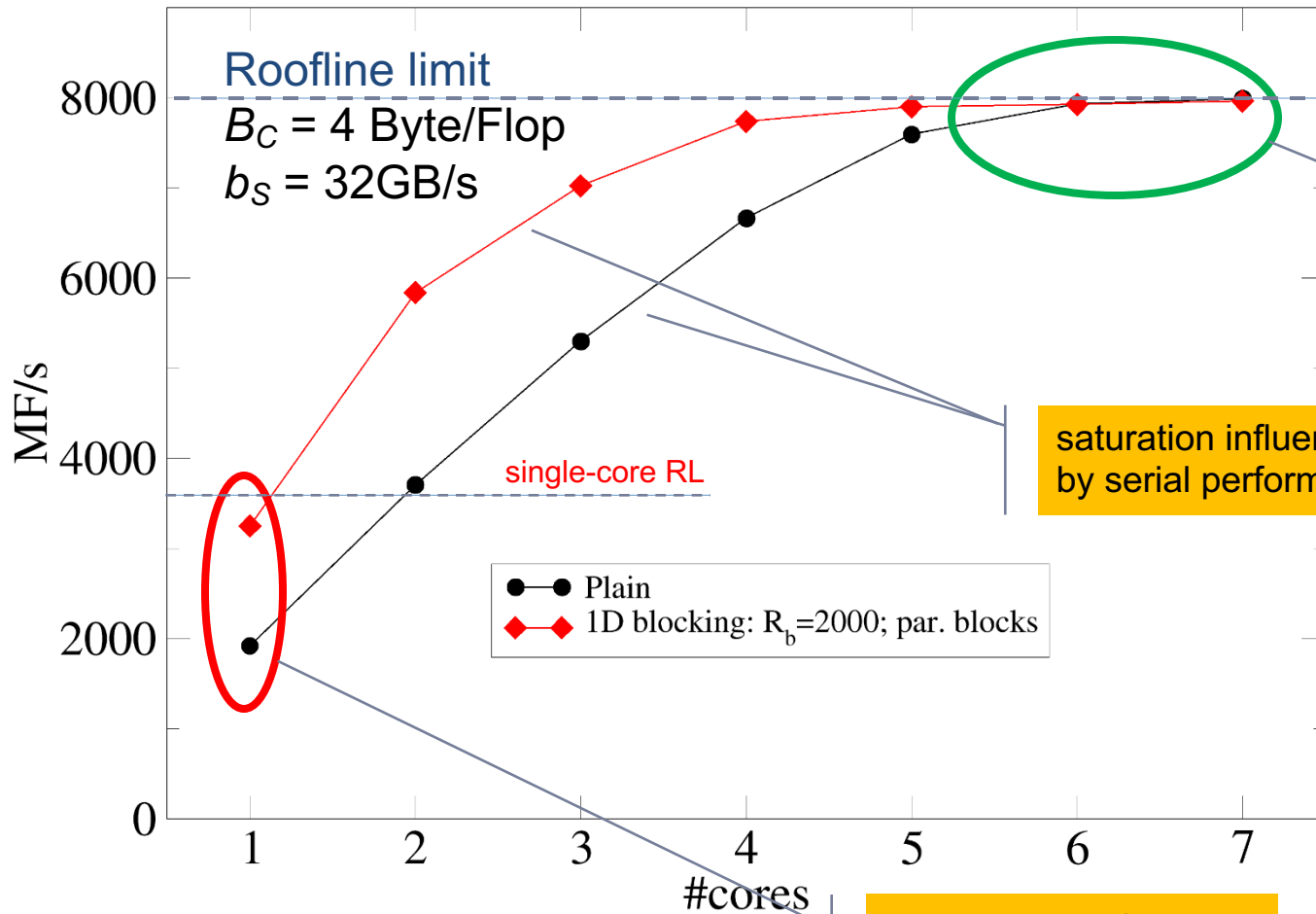
```
!$omp parallel do reduction(+:y)
do c = 1 , NC
  do r = 1 , NR
    y(r) = y(r) + A(r,c) * x(c)
  enddo ; enddo
!$omp end parallel do
```

plain code

```
!$omp parallel do private(rbS,rbE)
do rb = 1 , NR , Rb
  rbS = rb
  rbE = min((rb+Rb-1) , NR)
  do c = 1 , NC
    do r = rbS , rbE
      y(r) = y(r) + A(r,c) * x(c)
    enddo ; enddo ; enddo
!$omp end parallel do
```

blocked code

OpenMP parallelization & saturation



memory traffic unchanged
 → saturation unchanged!

saturation influenced by serial performance

So, is blocking useless?
 → NO (see later)

Can we do anything to improve B_C^{mem} ?
 → NO, not here

Intel Xeon E5 2695 v3 (Haswell-EP) CoD
 2.3 GHz base clock speed, $b_S = 32 \text{ GB/s}$

blocking good for single thread performance (reduced in-cache traffic)