

Programming Techniques for Supercomputers: **Modern Processors**

Characterization of memory hierarchies

Data access optimization

Prof. Dr. G. Wellein^(a,b) , Dr. G. Hager^(a), J. Hammer^(b), C.L. Alappat^(b)

^(a)HPC Services – Regionales Rechenzentrum Erlangen

^(b)Department für Informatik

University Erlangen-Nürnberg, Sommersemester 2019



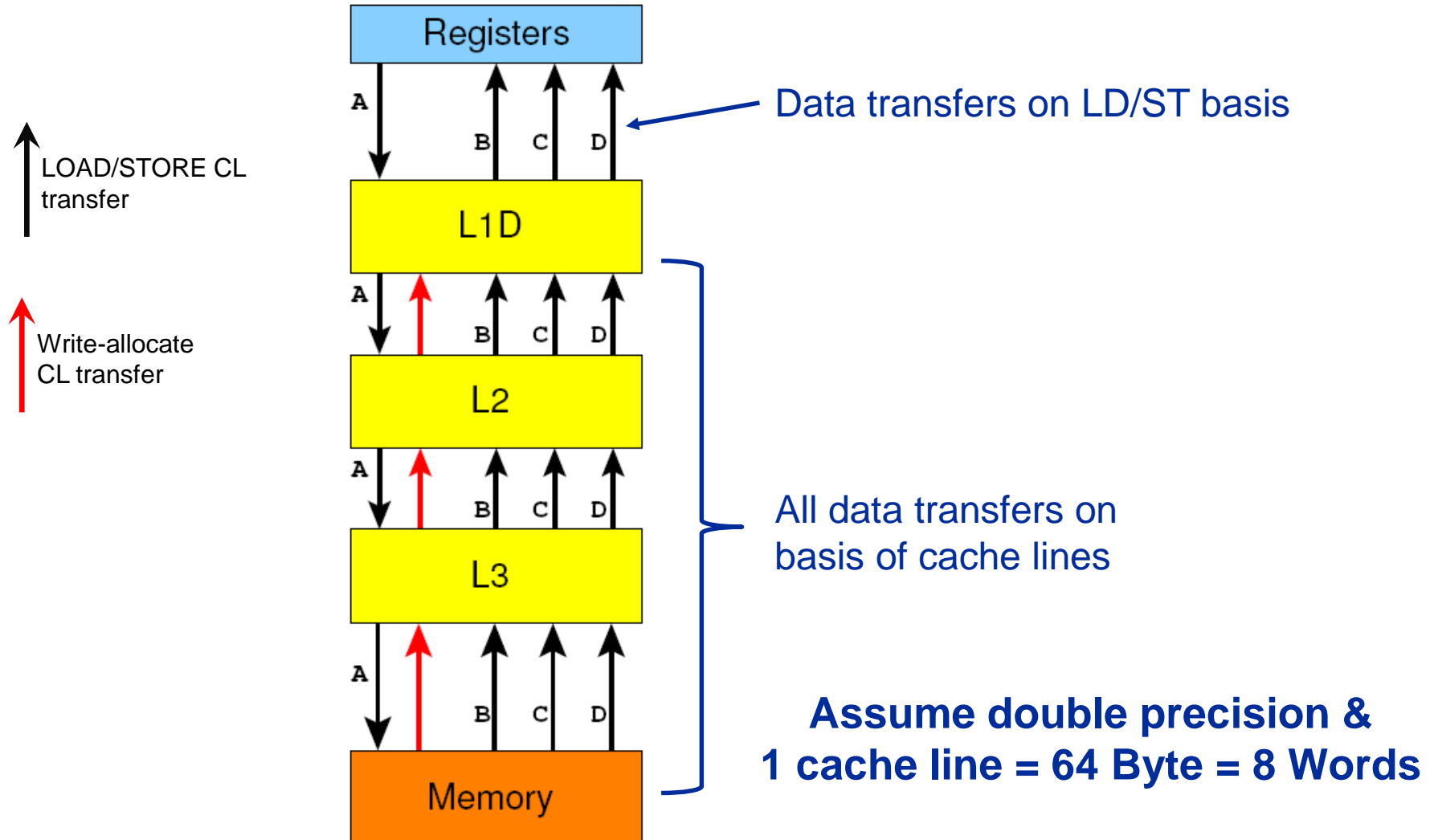
- **Determine performance levels with low level benchmark: Vector Triad**

```
DOUBLE PRECISION, dimension(SIZE):: A,B,C,D
DOUBLE PRECISION :: S,E,MFLOPS
! Input N .le. SIZE
DO i=1,N
    A(i) = 0.d0; B(i)=1.d0; C(i)=2.d0; D(i)=3.d0 ! initialize
ENDDO
call get_walltime(S)
DO ITER=1, NITER
    DO i=1, N
        A(i) = B(i) + C(i) * D(i)           ! 3 loads + 1 store; 2 FLOP
    ENDDO
    IF(A(2).lt.0) call dummy(A,B,C,D) ! Prevent loop interchange
ENDDO
call get_walltime(E)
MFLOPS = NITER * N * 2.d0 / ( (E-S) * 106 )
```

Characterization of Memory Hierarchies: Vector Triads – Application model



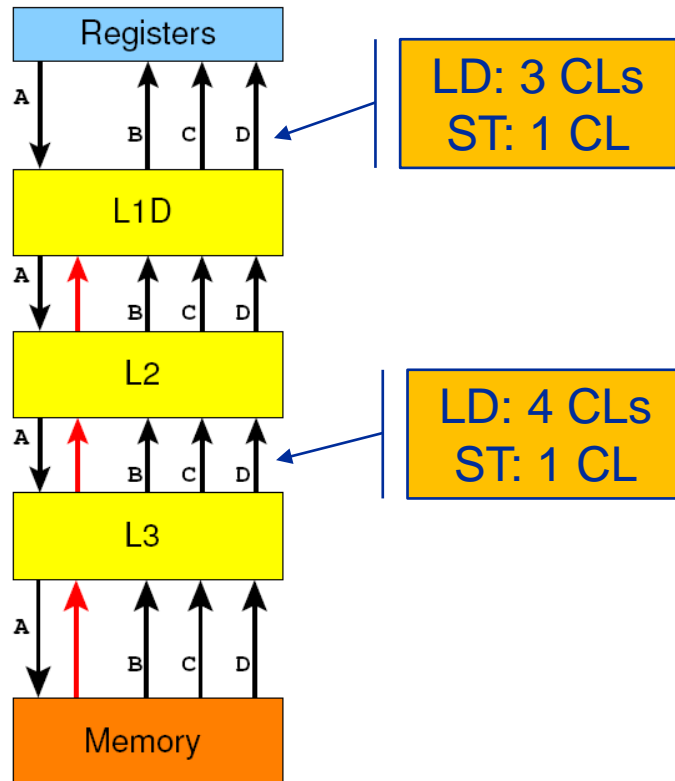
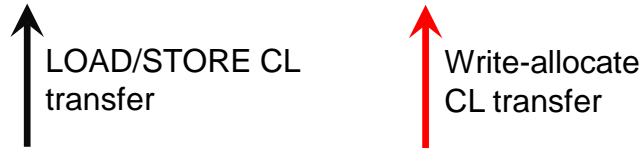
- **Data transfer analysis: Vector Triad** $A(i) = B(i) + C(i) * D(i)$



Characterization of Memory Hierarchies: Vector Triads – Application model



Data transfer analysis:



- Unit of work: 1 cache line update (1 CL-update), i.e. 8 iterations:
 $A(i:i+7) = B(i:i+7) + C(i:i+7) * D(i:i+7)$
→ 16 F/CL-update

- Calculate time [#cy] for 1 CL-update for every memory hierarchy (using machine model)
→ Total time: #cy / CL-update

→ Performance [F/s]:
 $\{(16 \text{ F/CL-update}) / (\text{\#cy/CL-update})\} * \text{ClockSpeed}$

Consider pure data volume – no dependencies – no latencies

Characterization of Memory Hierarchies: Processor Specs

Intel Xeon E5-2695 v3 (“Haswell”) – Machine model



| | |
|-----------------------|-------------|
| Microarchitecture | Haswell-EP |
| Shorthand | HSW |
| Xeon Model | E5-2695 v3 |
| Year | 09/2014 |
| Clock speed (fixed) | 2.3 GHz |
| Cores/Threads | 14/28 |
| Load/Store throughput | |
| AVX(2) | 2 LD & 1 ST |
| SSE/scalar | 2 LD & 1 ST |
| L1 port width | 2×32+1×32 B |
| ADD throughput | 1 / cy |
| MUL throughput | 2 / cy |
| FMA throughput | 2 / cy |
| L2-L1 data bus | 64 B |
| L3-L2 data bus | 32 B |
| LLC size | 35 MiB |
| Main memory | 4×DDR4-2133 |
| Peak memory BW | 34.2 GB/s |
| Load-only BW | 32 GB/s |
| T_{L3Mem} per CL | 4.6 cy/CL |

- Assumptions: streaming (no latency penalty)

- Calculate time per CL-transfer: $\#cy / CL$

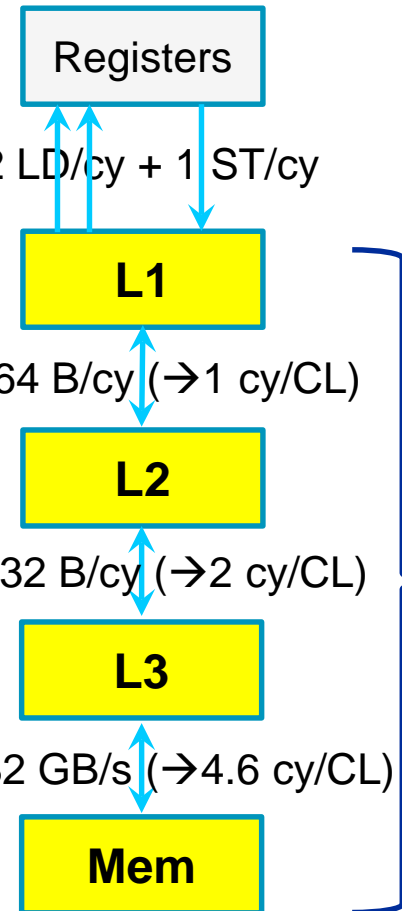
AXV LD/ST per cy

$\#cy/CL$

Cluster-on-die Mode (BW for 7 cores)

Machine characteristics:

Arithmetic:
2 FMA / cy





For 1 CL-update from L1 data, i.e. for 8 iterations:

$$A(i:i+7) = B(i:i+7) + C(i:i+7) * D(i:i+7)$$

6 AVX LDs

- $B(i:i+3)$, $B(i+4:i+7)$
- $C(i:i+3)$, $C(i+4:i+7)$
- $D(i:i+3)$, $D(i+4:i+7)$

2 AVX STs

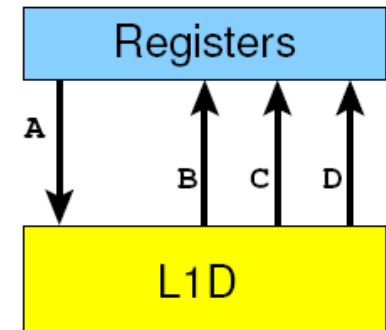
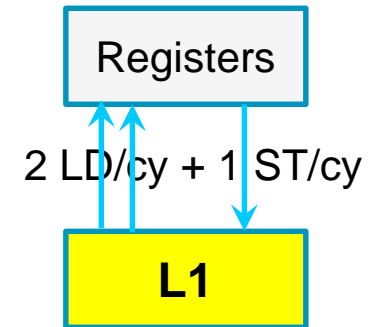
- $A(i:i+3)$, $A(i+4:i+7)$

→ **Bottleneck:** LD instructions → 6 LDs in 3 cy

→ 3 cy/CL-update for data in L1 cache

Machine characteristics:

Arithmetic:
2 FMA / cy





Machine characteristics:

For 1 CL-update i.e. for 8 iterations:

$$A(i:i+7) = B(i:i+7) + C(i:i+7) * D(i:i+7)$$

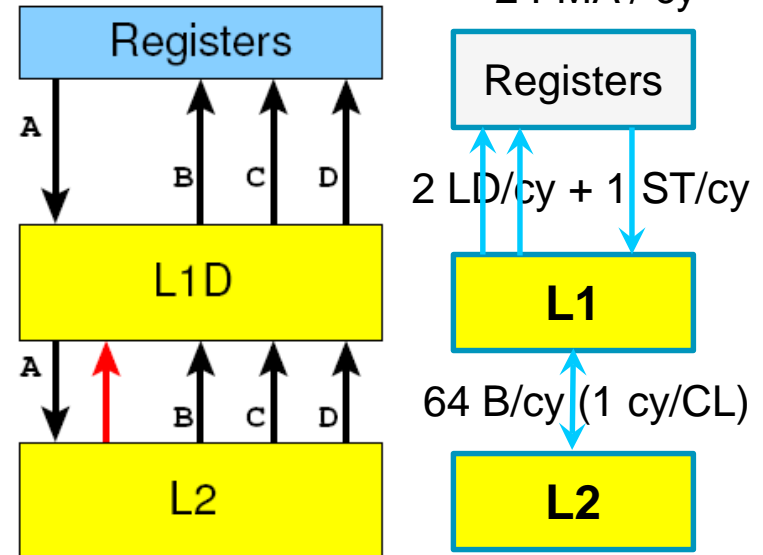
5 cache lines need to be transferred between L1 and L2 cache

→ **5 cy / CL-update for L1 – L2 traffic**

Total time for 1 CL-update for L2 data

$$L2 \leftrightarrow L1 + L1 \leftrightarrow \text{Register} \rightarrow (5 \text{ cy} + 3 \text{ cy}) / \text{CL-update} = \mathbf{8 \text{ cy} / \text{CL-update}}$$

This assumes no overlap in data transfers (i.e. L1 cache is single ported)

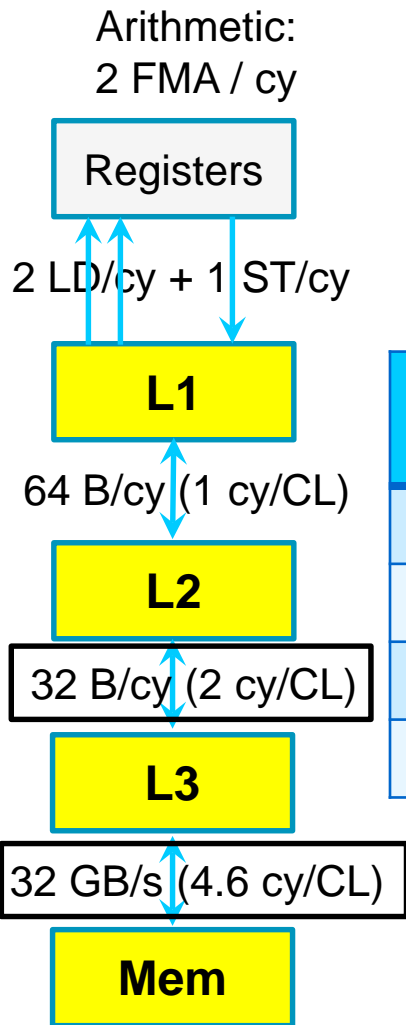
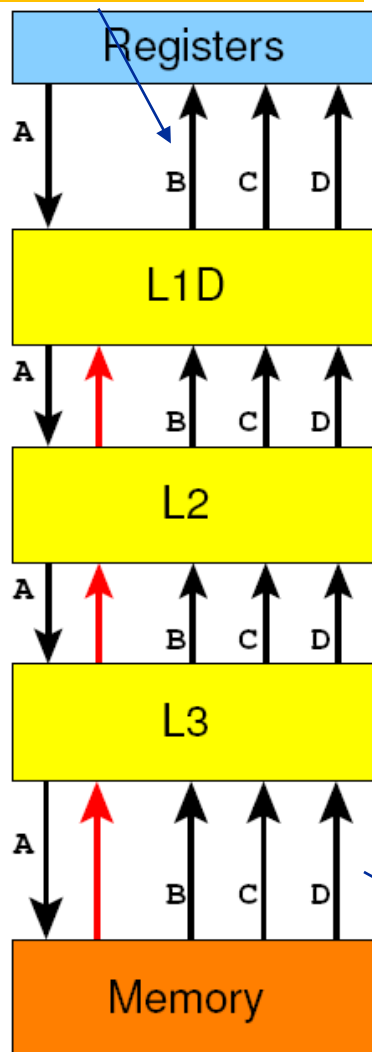


Characterization of Memory Hierarchies: Vector Triads on Intel Xeon E5-2695 v3 (“Haswell”)



Cache line (64 B)

Machine characteristics:



Where does data come from

Assuming no overlap throughout memory hierarchy

| Hierarchy | Transfer time to next level | Total transfer time |
|-----------|-----------------------------|---------------------|
| L1 | 3 cy | 3 cy |
| L2 | 5 cy | 8 cy |
| L3 | 10 cy | 18 cy |
| Memory | 23 cy | 41 cy |

Cycles per CL-update

Characterization of Memory Hierarchies: Vector Triads on Intel Xeon E5-2695 v3 (“Haswell”)



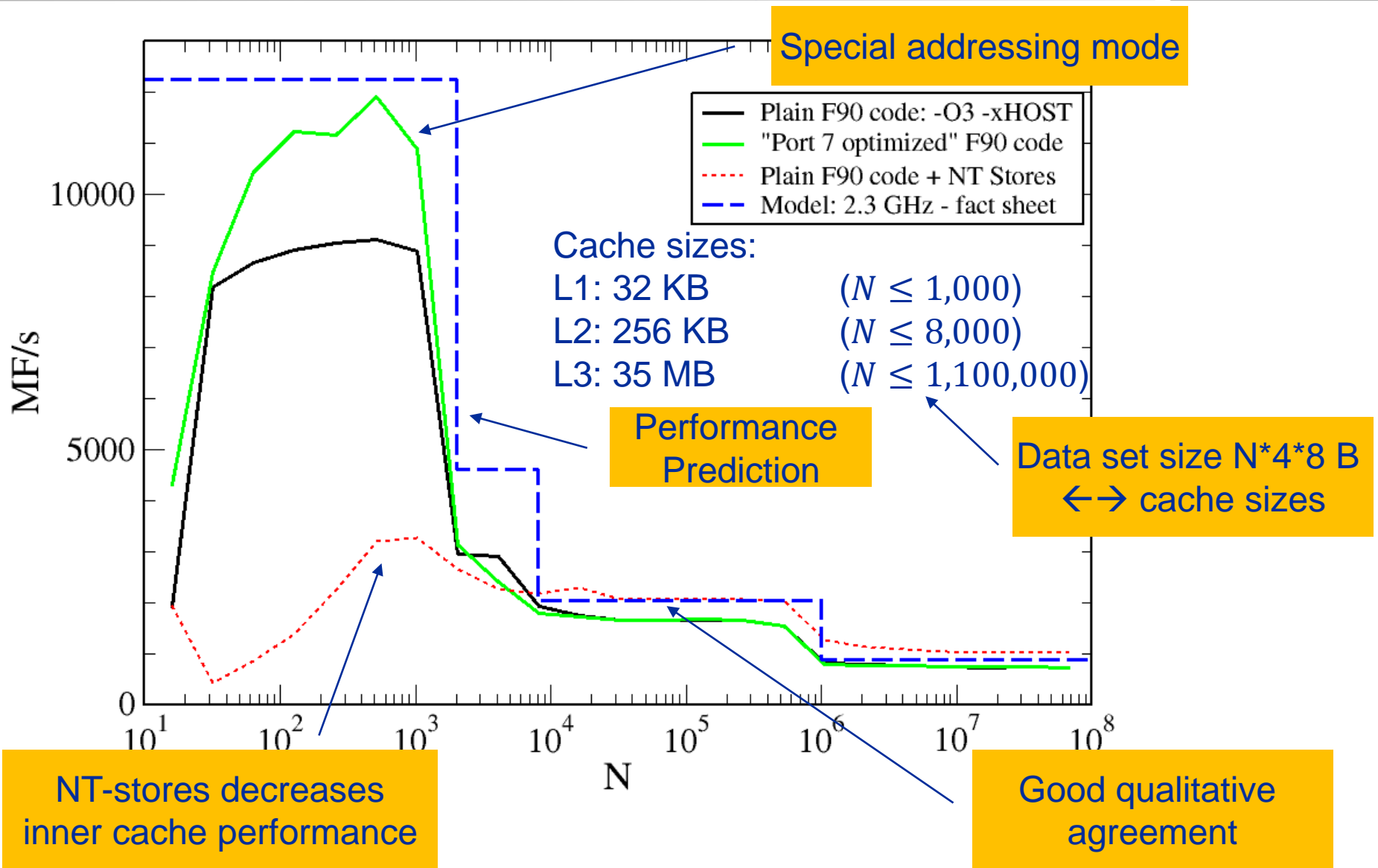
- From cy/CL-update to MF/s
- Performance: $\{(16 \text{ F/CL-update}) / (\# \text{cy/CL-update})\} * \text{ClockSpeed}$
- Clockspeed of Xeon E5-2695 v3: $2.3 \text{ GHz} = 2300 \text{ Mcy/s}$

| Hierarchy | Transfer time to next level | Total transfer time | Performance prediction |
|-----------|-----------------------------|---------------------|------------------------|
| L1 | 3 cy | 3 cy | 12266 MF/s |
| L2 | 5 cy | 8 cy | 4600 MF/s |
| L3 | 10 cy | 18 cy | 2044 MF/s |
| Memory | 23 cy | 41 cy | 898 MF/s |



Cycles per CL-update

Characterization of Memory Hierarchies: Vector Triads on Intel Xeon E5-2695 v3 ("Haswell"): Single core





Summary

- **Model/performance prediction purely based on fact sheet numbers and analysis of data transfers!**
- **Max. attainable performance for triads in L1 (P_{max}) is limited by LD port – only 2 FMAs per 3 cy \rightarrow 1/3 peak performance (cf. slide87@set_03_04_30_2019)**
- **For lower cache levels:**
 - Data transfer volume increases due to WRITE ALLOCATE
 - No overlap assumption in hardware transfer fits well – individual transfer times within memory/cache hierarchy add up!
 - No overlap assumptions holds qualitatively for Intel processors – may change – does not hold on other architectures
 - L2 cache performance lower than expected: 64 B / cy too optimistic – low level benchmarks: ~ 32 B/cy – 48 B/cy

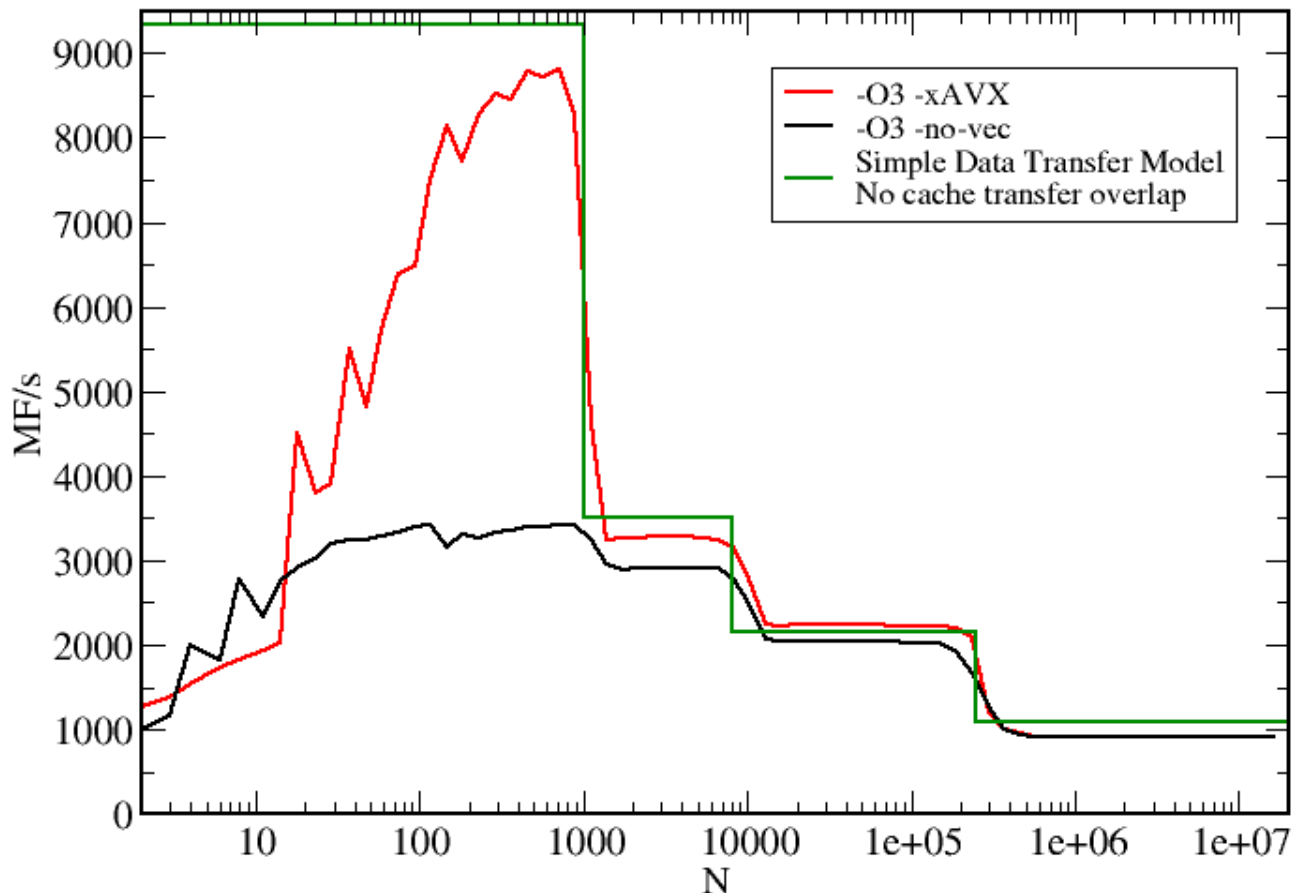
Characterization of Memory hierarchies: Vector Triad



1 core Intel Sandy Bridge (AVX) desktop system

SNB 3.5 GHz (no Turbo, no SMT)

A(:)=B(:)+C(:)*D(:) - ifort V12.1.0

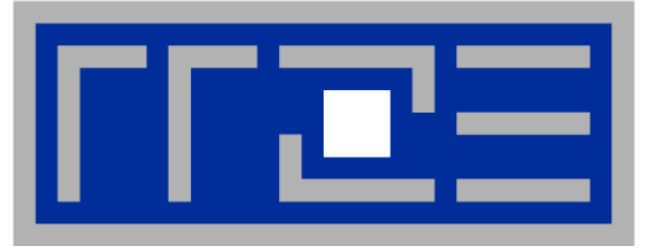


Reg \leftrightarrow L1:
256 Bit LD + 128 Bit ST

L1 \leftrightarrow L2:
256 Bit LD or ST

L2 \leftrightarrow L3
256 Bit LD or ST

L3 \leftrightarrow MEM
21.2 GB/s



Costs of data transfers - case study:

Dense Matrix Vector Multiplication (dMVM)

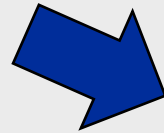
Dense matrix-vector multiplication in DP (AVX)



```
do c = 1 , C
  do r = 1 , R
    y(r) = y(r) + A(r,c) * x(c)
  enddo
enddo
```

Left hand side
(LHS) vector

Right hand side
(RHS) vector



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

- Assume $C = R \approx 10,000$ – double precision
- Size of data sets involved:
 - $\text{Size}(A(:, :)) = C * R * 8 \text{ Byte}$
 - $\text{Size}(y(:)) = R * 8 \text{ Byte}$
 - $\text{Size}(x(:)) = C * 8 \text{ Byte}$

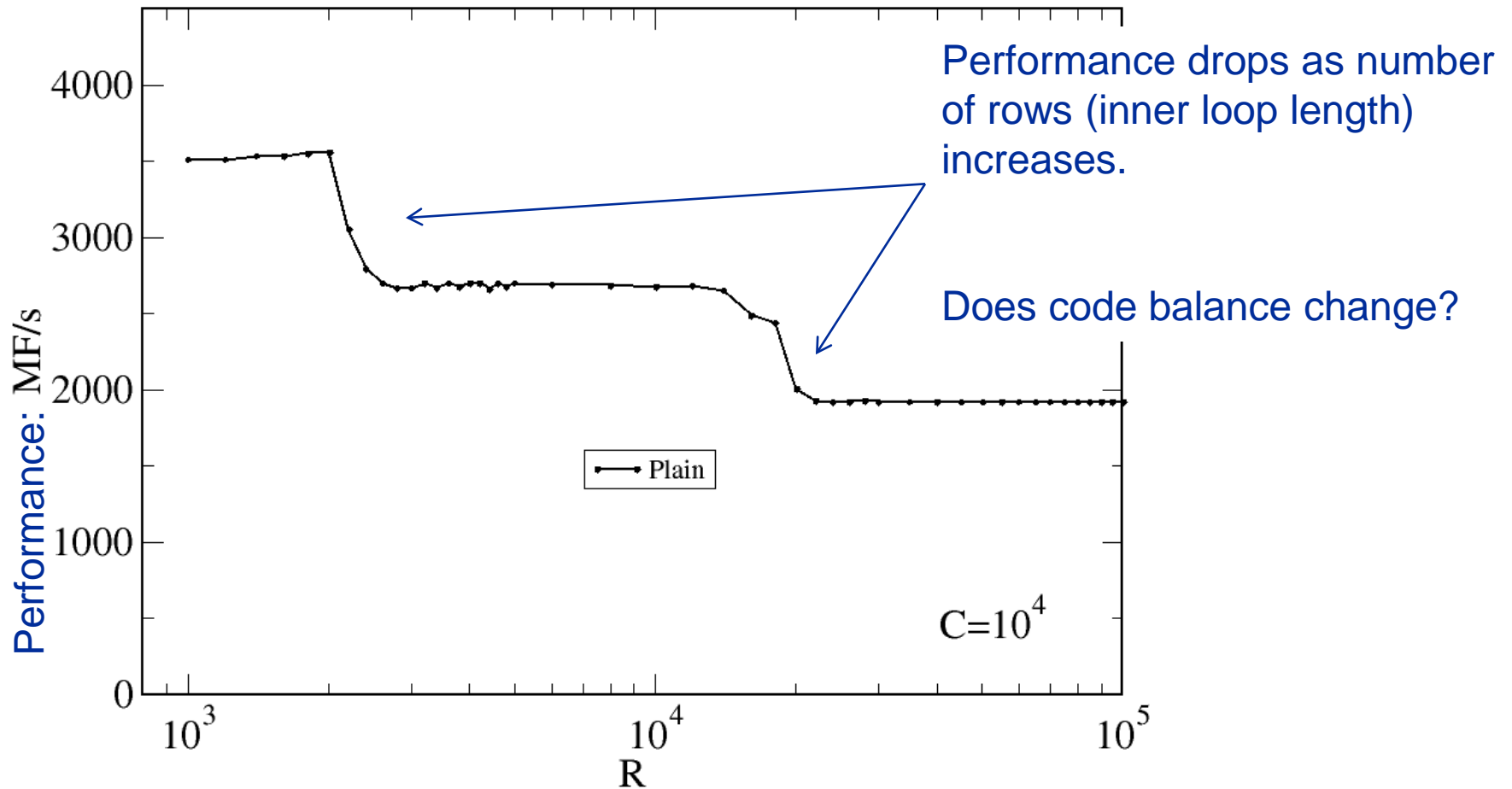


- **Code balance** (B_C) quantifies how much data is transferred over a given data path per unit of work:

$$B_c = \frac{\text{data transfer [Byte]}}{\text{amount of work [flop, iteration, CL-update...]}}$$

- **Code balance can be specified for every memory hierarchy, e.g.**
 B_C^{mem} , B_C^{L3} , B_C^{L2}
- **Example: Vector triad $A(:) = B(:) + C(:) * D(:)$ – double precis.**
 $B_c = (4+1) \text{ Words} / 2 \text{ Flops} = 2.5 \text{ W/F (including write allocate)} = 20 \text{ B/F}$
(same for all memory hierarchies from L2 to main memory)
- **“work”** – in general *flop* but can be chosen problem specific

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



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

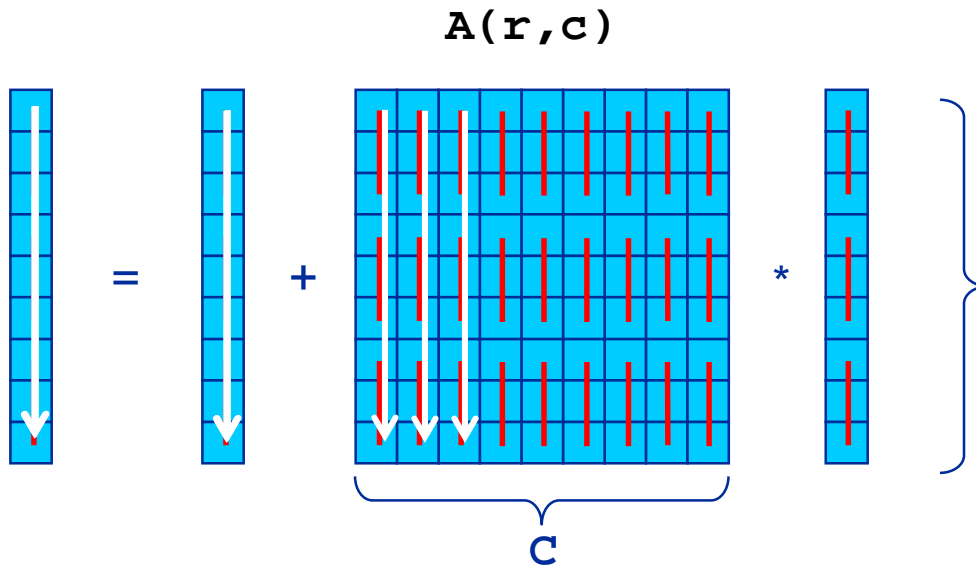


```
do c = 1 , C
  tmp=x(c)
  do r = 1 , R
    y(r)=y(r) + A(r,c)* tmp
  enddo
enddo
```

`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 \rightarrow for $c > 1$ update $y(:)$ in cache

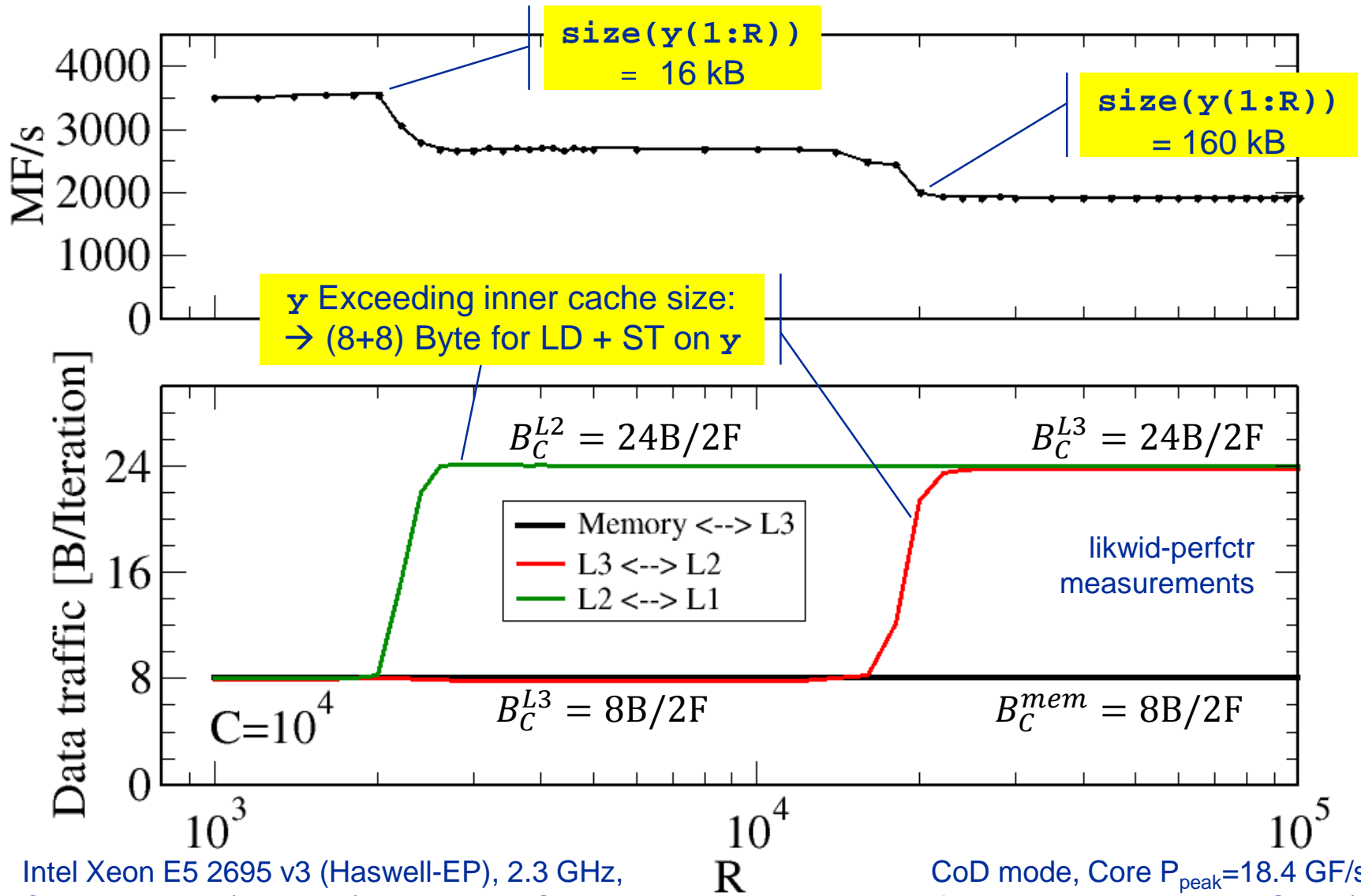


$y(:)$ may not fit in innermost cache \rightarrow more traffic from lower level caches for larger R

Depending on R : Determine code balance in memory (B_C^{mem}) and from relevant cache level(s) (B_C^{L3}, B_C^{L2})!

See also discussion on [spatial/temporal](#) locality on [slide20@lecture4](#)

DMVM (DP) – Single core data traffic analysis



Intel Xeon E5 2695 v3 (Haswell-EP), 2.3 GHz,
Caches: 32 KB / 256 KB / 35 MB, PageSize: 2 MB;

CoD mode, Core $P_{\text{peak}} = 18.4 \text{ GF/s}$,
ifort V15.0.1.133; $b_S = 32 \text{ Gbyte/s}$



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

Data transfer for A:
8 B / iteration

Potential data transfer
for y: 16 B / iteration

- Depending on R and cache size the code balance (B_C^i) attains two values:
 - $B_C^i = 8 B/iteration$ if only the matrix data (8 B) is loaded from hierarchy i to inner caches (best case)
 - $B_C^i = (8 + 16) B/iteration$ if matrix data (8 B) is loaded and LHS data (16 B) is updated from hierarchy i (worst case)



- Updating LHS vector ($y(:)$) may dominate in-cache traffic
- Worst case: Complete vector $y(1:R)$ is updated in (travelling up & down to) some outer cache level $C-1$ times
- Strategies to **increase temporal locality of LHS** access in inner caches (preferably L1 cache / register)
 - (Outer Loop) Unroll and Jam \rightarrow Perform two “updates” of $y(r)$ in innermost (c) loop
 - Blocking/Tiling of inner (c) loop such that “sufficiently” small parts of $y(r)$ are updated

Reducing traffic by “Unroll and Jam”



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

Original

y(1:R) Updated C-1 times from outer cache

```
do c = 1 , C , 2
  tmp1=x(c)
  do r = 1 , R
    y(r)=y(r) + A(r,c)* tmp1
  enddo
  tmp2=x(c+1)
  do r = 1 , R
    y(r)=y(r) + A(r,c+1)* tmp2
  enddo
enddo
```

2-way outer unroll

y(1:R) Updated C/2-1 times from outer cache

Data transfer for y:
16 Byte / 2 iterations

```
do c = 1 , C , 2
  tmp1=x(c)
  tmp2=x(c+1)
  do r = 1 , R
    y(r)=y(r) + A(r,c) * tmp1
    y(r)=y(r) + A(r,c+1)* tmp2
  enddo
enddo
```

2-way unroll + jam

ST y(r)

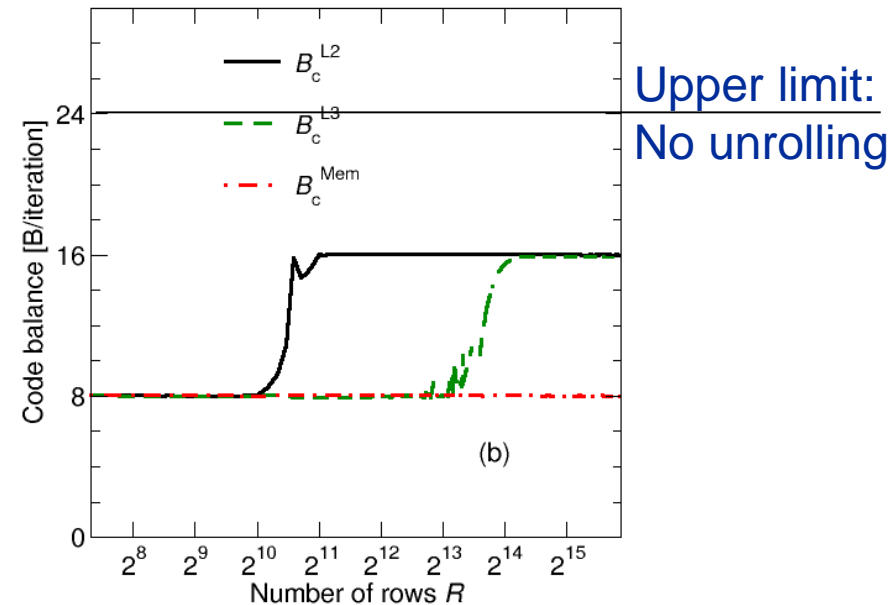
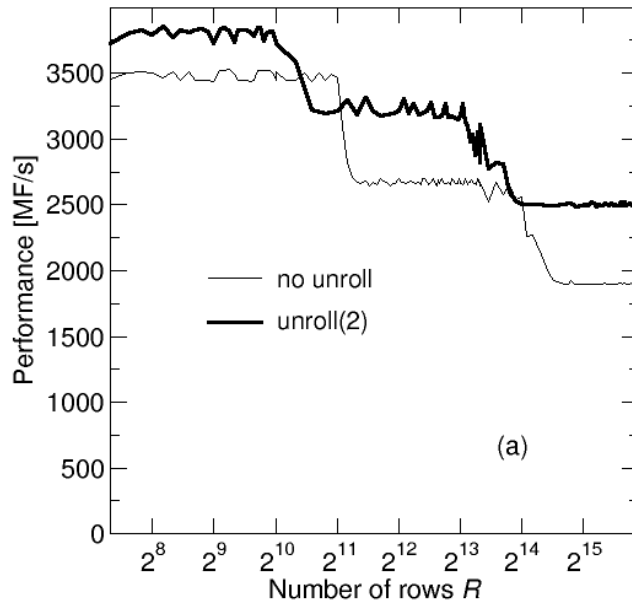
LD y(r)

Reducing traffic by “Unroll and Jam”



Using „2-way Unroll & Jam“ the worst case balance is reduced

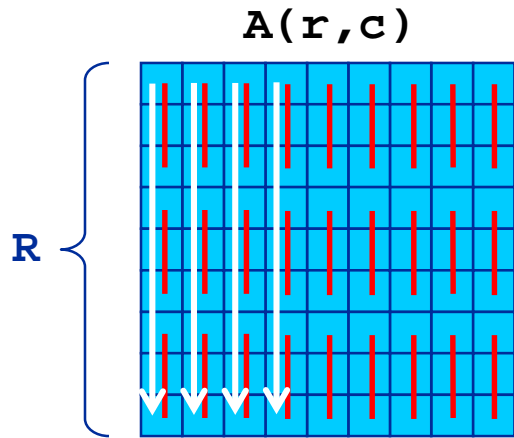
from $B_C^i = (8 + 16) B/iteration$ to $B_C^i = (8 + 8) B/iteration$



General case: For „m-way Unroll & Jam“ the worst case balance is

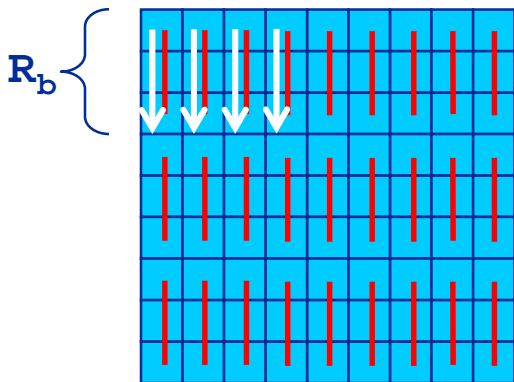
$$B_C^i = \left(8 + \frac{16}{m}\right) B/iteration$$

Reducing traffic by “blocking/tiling”



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

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

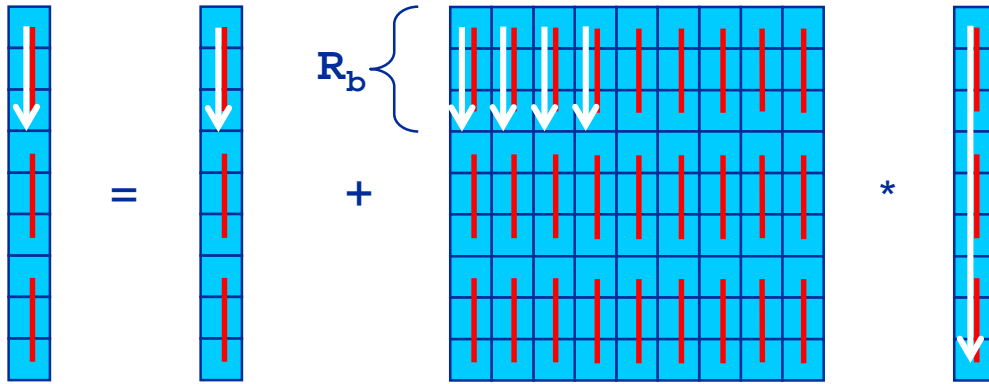


```
do rb = 1 , R ,  $R_b$ 
  rbs = rb
  rbE = min((rb+ $R_b$ -1), R)
  do c = 1 , C
    do r = rbs , rbE
      y(r)=y(r) + A(r,c)*x(c)
    enddo
  enddo
enddo
```

R_b free parameter – adapted to caches size

$y(rbs:rbE)$ may fit into some cache if R_b is small enough \rightarrow traffic reduction

Reducing traffic by “blocking/tiling”



```
do rb = 1 , R , Rb
  rbs = rb
  rbe = min((rb+Rb-1), R)
  do c = 1 , C
    do r = rbs , rbe
      y(r)=y(r) + A(r,c)*x(c)
    enddo
  enddo
enddo
```

- LHS only updated once in some cache level if blocking is applied
- Price: RHS is loaded R / R_b instead of once!

→ Contribution of **y** and **x** transfers to worst case code balance:

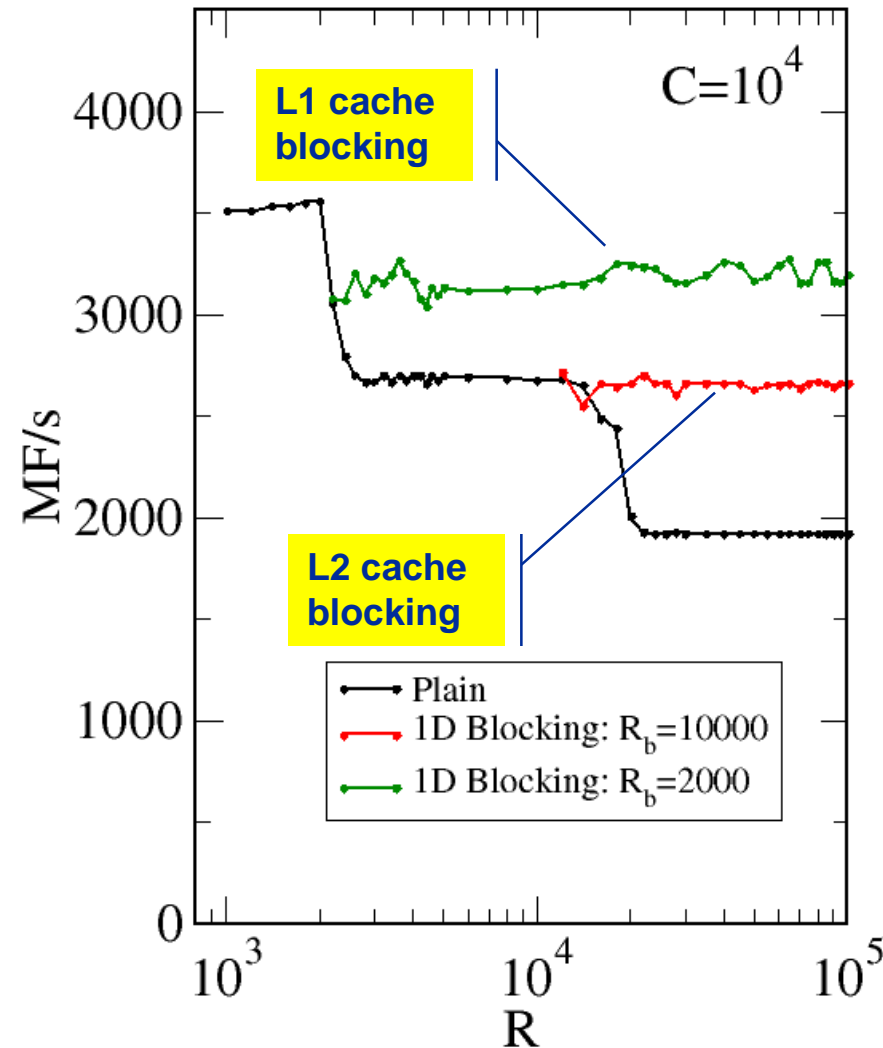
$$\frac{16B * R * C + 8B * C}{R * C \text{ iterations}} = \left(16 + \frac{8}{R}\right) \frac{B}{it.} \cong 16 \frac{B}{it.}$$

Original code

Blocking with factor R_b

$$\frac{16B * R + 8B * C * R / R_b}{R * C \text{ iterations}} = \left(\frac{16}{C} + \frac{8}{R_b}\right) \frac{B}{it.} \cong \frac{8}{R_b} \frac{B}{it.}$$

→ Do not choose R_b too small



- “1D blocking” for inner loop
- Blocking factor $R_b \leftrightarrow$ cache level

```

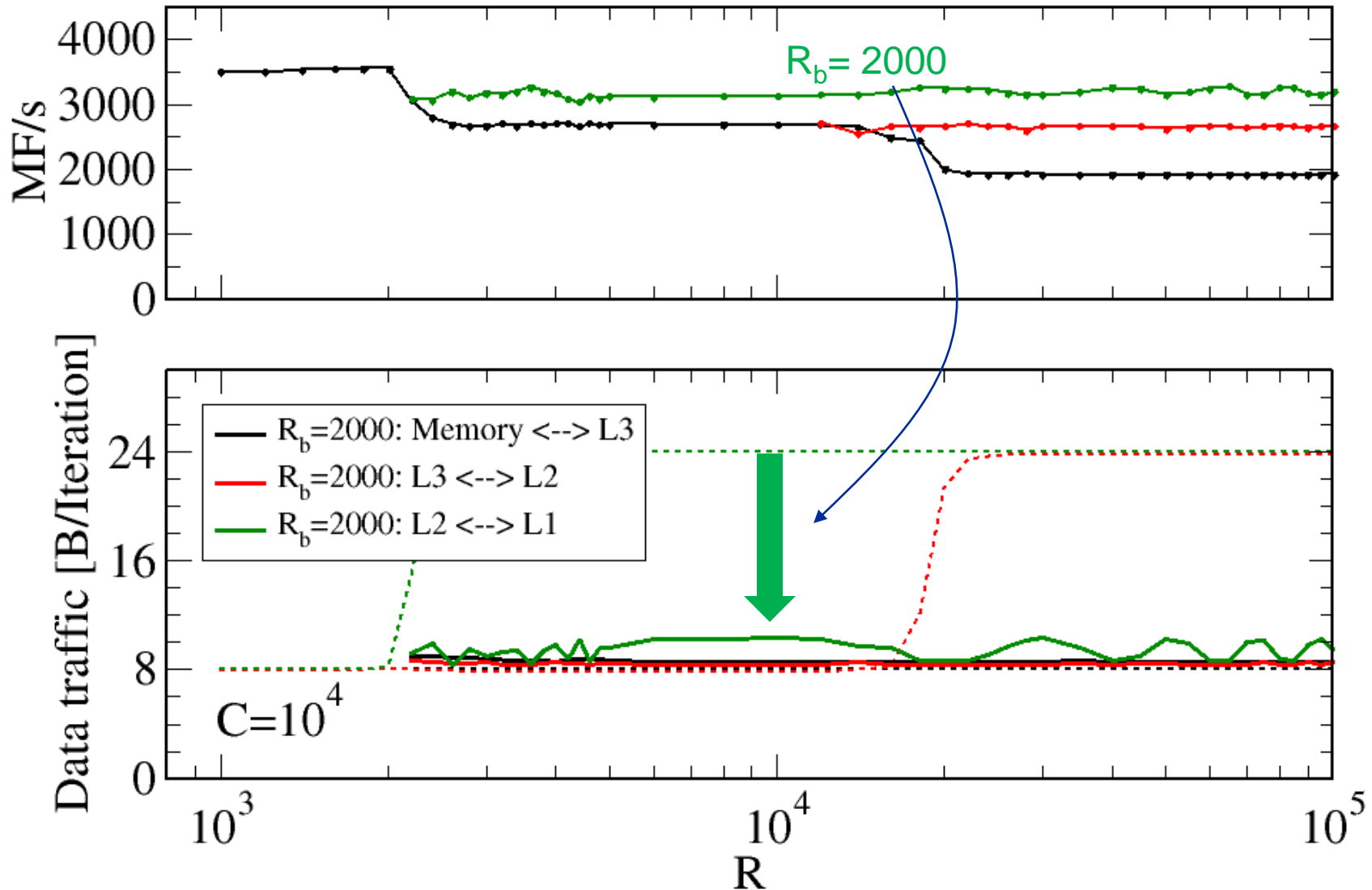
do rb = 1 , R , Rb

  rbS = rb
  rbE = min((rb+Rb-1), R)

  do c = 1 , C
    do r = rbS , rbE
      y(r)=y(r) + A(r,c)*x(c)
    enddo
  enddo

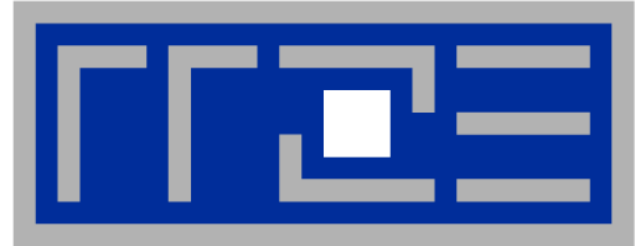
enddo
    
```

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





- We have found the reasons for the **breakdown of single-core performance** with growing number of matrix rows
 - LHS vector fitting in different levels of the cache hierarchy
 - Validated theory by performance counter measurements
- **Inner loop blocking** was employed to improve code balance in L3 and/or L2
 - Validated by performance counter measurements
- **Outer Loop Unrolling / Blocking** led to **better single-threaded performance** – exploiting **temporal locality** in inner cache levels
- **Single core performance is sensitive to in-cache traffic volume**
- **Balance metric is useful to quantify data traffic per work ratio and estimate impact on performance**



Data Access Optimizations

**General considerations valid for any
memory hierarchy levels**

Classify basic operations classes with respect to
their asymptotic balance characteristic and identify
potential optimization strategies



- Use “flop“-definition of balance for one hierarchy (memory):

$$B_C = \frac{\text{data transfer [B]}}{\text{amount of work [F]}}$$

- Consider characteristic balance scaling – the lower the better
- For a given kernel determine as a function of problem size:
 - **Minimum amount of data transfer** – a good lower bound is the size of the largest data structures involved
 - **Number of floating-point operations** to be executed - just count them
 - Consider leading order effects only!



■ Example: Matrix-matrix multiplication (DGEMM)

```
do i=1,N
  do j=1,N
    do k=1,N
      c(j,i)=c(j,i)+a(k,i)*b(k,j)
    enddo
  enddo
enddo
```

Minimum Balance Analysis

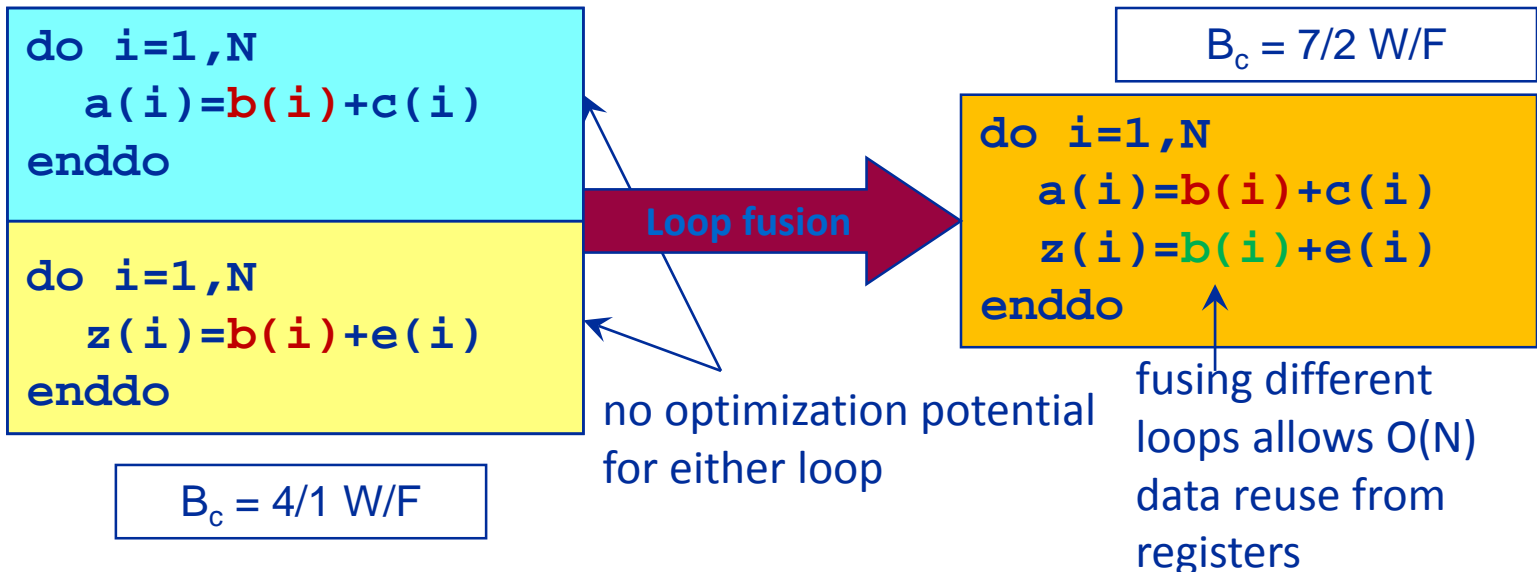
- Minimum data transfer \leftrightarrow 3+1 (NxN) matrices
 $\rightarrow (3 + 1) * N^2 * 8 B$
- Amount of work \leftrightarrow 3 nested loops
 $\rightarrow 2 * N^3 * F$

$$\rightarrow B_C = \frac{32 * N^2}{2 * N^3} \frac{B}{F} \rightarrow \frac{O(N^2)}{O(N^3)} \quad (\text{Minimum balance})$$



Case 1: $O(N)/O(N)$ algorithms

- $O(N)$ data access operations vs. $O(N)$ arithmetic operations
- Examples: Scalar product, vector addition, sparse MVM etc.
- Performance **limited by memory BW** for large N (“memory bound”)
- Limited optimization potential for single loops
 - ...at most a constant factor for multi-loop operations
- **Example1: successive vector additions (double precision: 1 W = 8 B)**



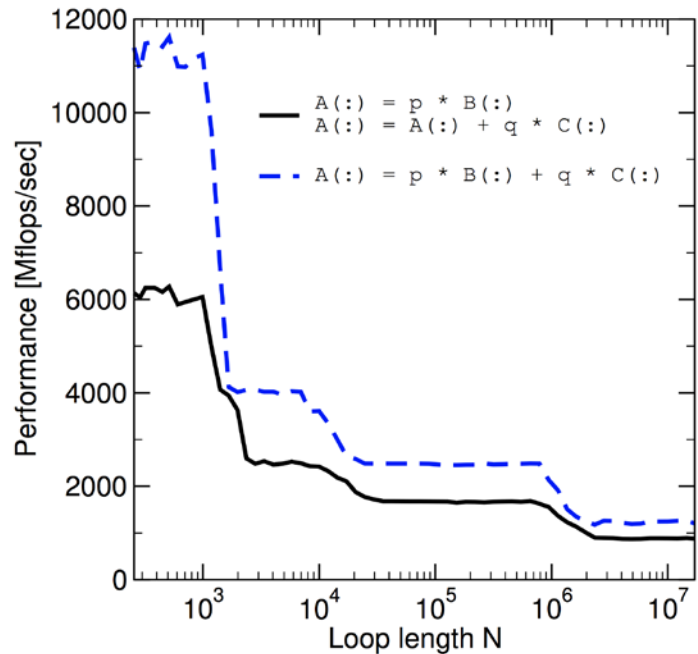


Example2

```
1 do i = 1,N ! 3 transfers, 1 flop
2   A(i) = p * B(i)
3 enddo
4 do i = 1,N ! 3 transfers, 2 flops
5   A(i) = A(i) + q * C(i)
6 enddo
```

loop fusion

```
! optimized
do i = 1,N
! save two transfers for A(i)
! -> only 4 transfers
  A(i) = p * B(i) + q * C(i)
enddo
```



Intel Xeon E5-2660 v2 („Ivy Bride“):
2.2 GHz; Caches: 32 KB / 256 KB / 25 MB

Below L1 cache reduction in balance directly
correlates with increasing performance

In L1 cache: ST is bottleneck in both variants
→ Speed-up by 2x



Case 2: $O(N^2)/O(N^2)$ algorithms

- Examples: dense matrix-vector multiply, matrix addition, dense matrix transposition etc.
 - Nested loops
- Memory bound for large N
- Some optimization potential (at most constant factor)
 - Can often enhance code balance by **outer loop unrolling** and/or **spatial blocking**
- Example: dense matrix-vector multiplication

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

- data transfer \leftrightarrow 1 ($N \times N$) matrix
 $\rightarrow N^2 * 8 B$
- amount of work \leftrightarrow 2 nested loops
 $\rightarrow 2 * N^2 * F$

$$\rightarrow B_c = \frac{8 * N^2}{2 * N^2} \frac{B}{F} = 4 \frac{B}{F} \left(\frac{O(N^2)}{O(N^2)} \right)$$

(Minimum balance!!!!!!)



■ $O(N^2)/O(N^2)$ algorithms – Optimization strategies

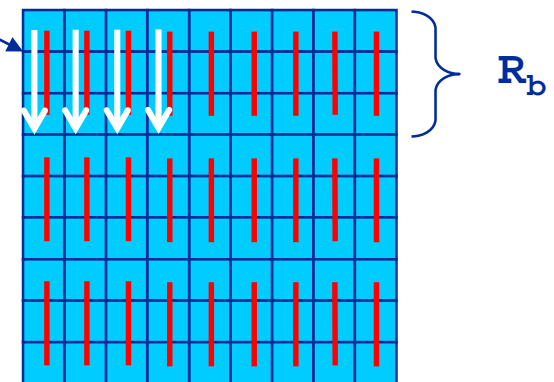
- (Inner) Loop blocking/tiling (see slides 24 – 26)
- (Outer) Loop unroll & Jam (see slides 22/23)
- And combination of both

■ Overheads to consider:

- Significant code bloat
- Short inner loops may cause overhead (e.g. fooling prefetcher: stop consecutive access and jump by large offset)
- Loop Unrolling increases register pressure

```
do c = 1 , C , 2
  tmp1=x(c)
  tmp2=x(c+1)
  do r = 1 , R
    y(r)=y(r) + A(r,c) * tmp1
    y(r)=y(r) + A(r,c+1) * tmp2
  enddo
Enddo
```

5 registers





■ Case 3: $O(N^{k-m}) / O(N^k)$ algorithms

- Most favorable case – computation outweighs data traffic by factor of N^m
- **Minimum balance** $\propto N^{-m}$ (see slide 31 for $m=1$ / DGEMM)
- Huge optimization potential: proper optimization can **render the problem cache-bound** if N is large enough
- Examples **Dense matrix-matrix multiplication** ($k=3; m=1$; DGEMM; see slide 31)
- Naïve implementation may still have much worse balance:

```
do i=1,N
  do j=1,N
    do k=1,N
      c(j,i)=c(j,i)+a(k,i)*b(k,j)
    enddo
  enddo
enddo
```

Core task: dense MVM
($B_C = 4 \frac{B}{F}$) $\rightarrow O(N^2)/O(N^2)$
 \rightarrow **memory bound**

3 N^2 data elements involved BUT main memory traffic $\sim N^3$ for large N
Reason: $b(:,i)$ needs to be reloaded N times!

\rightarrow **Implement (spatial) blocking + unrolling!**



■ $O(N^1)/O(N^2)$ Example:

```
1 do i = 1,N
2   do j = 1,N
3     sum = sum + A(i) * B(j)
4   enddo
5 enddo
```

Blocking Inner Loop

Outer Loop Unroll
& Jam

```
1 do jbs = 1,N,Rb
2   jstart = jbs; jend = min(jbs+Rb-1,N)
3   do i = 1,N,2
4     do j = jstart,jend
5       sum1 = sum1 + A(i) * B(j)
6       sum2 = sum2 + A(i+1) * B(j)
7     enddo
8     sum = sum + sum1 + sum2
9   enddo
10 enddo
```



■ DGEMM Example:

```
subroutine mmm(a,b,c,n)
integer :: n
double precision :: a(n,n),b(n
integer :: i,j,k

do k=1,n
do j=1,n
do i=1,n
c(i,k) = c(i,k) +
a(j,k) * b(i,j)
enddo
enddo
enddo

end subroutine mmm
```

vectorized

vectorized

```
LOOP BEGIN at scan_f.f90(47,3)
remark #25442: blocked by 125 (pre-vector)
remark #25440: unrolled and jammed by 4 (pre-vector)
remark #15542: loop was not vectorized: inner loop was already
remark #25015: Estimate of max trip count of loop=125

LOOP BEGIN at scan_f.f90(48,6)
remark #25442: blocked by 125 (pre-vector)
remark #25440: unrolled and jammed by 4 (pre-vector)
remark #15542: loop was not vectorized: inner loop was already
remark #25015: Estimate of max trip count of loop=125

LOOP BEGIN at scan_f.f90(49,9)
<Peeled loop for vectorization>
remark #25456: Number of Array Refs Scalar Replaced In Loop: 36
remark #25015: Estimate of max trip count of loop=3
LOOP END

LOOP BEGIN at scan_f.f90(49,9)
remark #25442: blocked by 128 (pre-vector)
remark #15300: LOOP WAS VECTORIZED
remark #15442: entire loop may be executed in remainder
remark #15448: unmasked aligned unit stride loads: 1
remark #15449: unmasked aligned unit stride stores: 4
remark #15450: unmasked unaligned unit stride loads: 7
remark #15451: unmasked unaligned unit stride stores: 12
remark #15475: --- begin vector cost summary ---
remark #15476: scalar cost: 142
remark #15477: vector cost: 62.500
remark #15478: estimated potential speedup: 2.150
remark #15488: --- end vector cost summary ---
remark #25456: Number of Array Refs Scalar Replaced In Loop: 36
remark #25015: Estimate of max trip count of loop=32
LOOP END
LOOP END
LOOP END
```

Compiler blocks all loops (125/125/128)
& unrolls the two outer loops by 4x
→ 9.8 GF/s out of 17.6 GF/s for n=5,000



- **The cost of data transfer increase rapidly as they come from outer level caches / main memory**
- **Increase spatial and temporal locality by**
 - outer loop unrolling,
 - spatial blocking
 - or combination of both.
- **Things to consider with these transformations:**
 - Prefetchers– “sufficiently long” contiguous memory accesses (i.e. inner loops)
 - Limited number of registers - limits unrolling factor
 - Overhead at “blocking boundaries”
 - Size of cache to block for