\Orchestrating a brighter world NEC

aiXcelerate 2016, Data locality and cache blocking

November, 2016

Rene' Puttin

NEC Deutschland GmbH

Basics: Background from nature

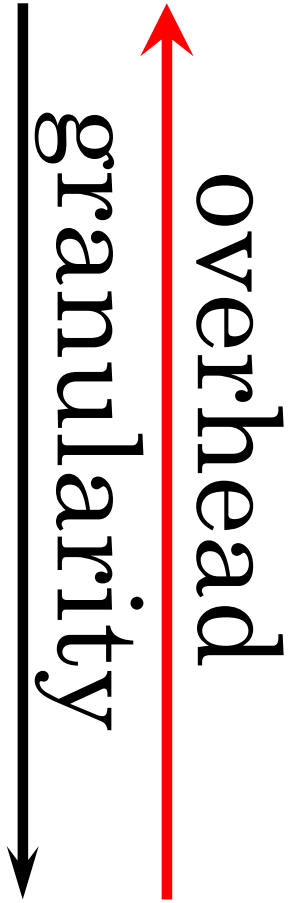
- Nature – and the corresponding mathematical modelling – exhibit a lot of locality, and this leads to …
  - data-locality
  - parallelism
  - applicability of domain decomposition
- Contemporary hardware is providing the features to deal with those aspects
- So architectures seem increasingly in line with "Mother Nature"
  - except memory bandwidth
- Codes should respect these aspects and their consequences

© NEC Corporation 2016

Basics: Levels of parallelism (1/2)

There are different levels of parallelism

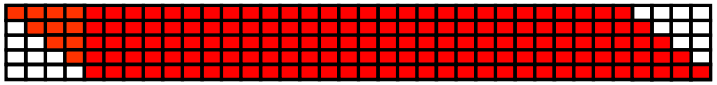
1. data parallelism
2. parallel operations
3. parallel outer loops in loop nests
4. parallel subroutines
5. parallel processes (of different granularity)



© NEC Corporation 2016

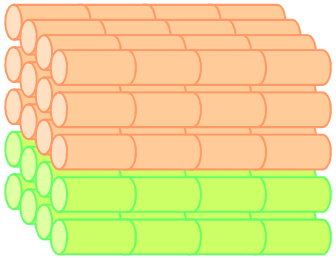
Basics: Levels of parallelism (2/2)

- Segmentation

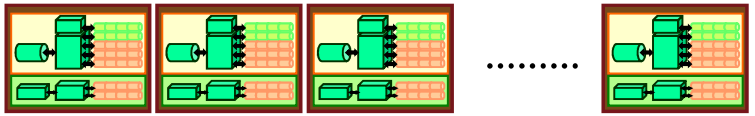


- multiple pipes

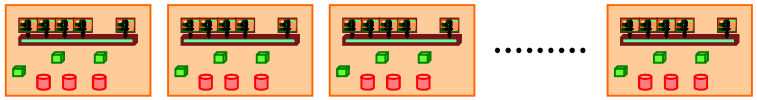
- parallel usage of functional units



- parallel CPUs



- parallel nodes



© NEC Corporation 2016

Basics: Tuning Approaches (1/2)

Optimization approaches

- Vectorization
  - vectorize as much as possible (speedup factor 4 (AVX) possible)
- Cache blocking
  - apply cache-blocking to most important loops
- Other single CPU-tunings
  - check most important code parts for other tuning potential
- I/O tuning
  - check whether I/O is done efficiently
- OpenMP-Optimizations
  - parallelize all important serial parts
  - check whether OpenMP scales well, ideally on routine or even loop basis
- MPI-Optimizations
  - parallelize all important serial parts
  - check whether MPI scales well, ideally on routine or even loop basis

© NEC Corporation 2016

Basics: Tuning Approaches (2/2)

- use profile for optimization
- use iterations and always start with top routine and go downwards
  - 1. Iteration
    - check routines for obvious tuning potential
    - apply obvious tunings
    - do not search long for tunings, but move on to next routine soon
    - stop if routines have less impact than self-defined barrier (e.g. 1%)
    - check for routines running on only one or a few processors
  - 2. Iteration
    - investigate routines on a more detailed level
    - introduce time measurements to detect most important parts of a subroutine
    - investigate those parts even more thoroughly
    - stop if routines have less impact than self-defined barrier (e.g. 5%)
    - check for routines running on only one or a few processors
  - 3. – n. Iteration focus more and more on top routines

Routines which in principle only call math libraries can be skipped!

in classic profiles those routines appear very far downwards!!!

© NEC Corporation 2016

Basics: Basic rules for performance

The following rules are necessary for performance:

- **vectorization of important portions**
- **data parallelism or reduction for innermost loop**

- **long innermost loop**
- **lots of instructions in innermost loop**
- **no unnecessary memory traffic**

- stride one data access
- avoiding indirect addressing
- ‘simple’ loop structures

© NEC Corporation 2016

Cache Blocking: Direct Mapped Cache (1/2)

- Model-architecture
- Line size 64 Byte = 8 x 8 Bytes (8 double precision)
- Cache size 32 kByte → 2^15/2^6 = 2^9 = 512 lines
- Addressing (Byte-address)
  - . . . **00100101110101001010101000100010000**

Cache-Line-# Word Byte

- Example: add two vectors
  - Very slow:

```
real*8 a(4096), b(4096), c(4096)
common / chaos / a, b, c
do i = 1, 4096
  a(i) = b(i)+ c(i)
end do
```


© NEC Corporation 2016

Cache Blocking: Direct Mapped Cache (2/2)

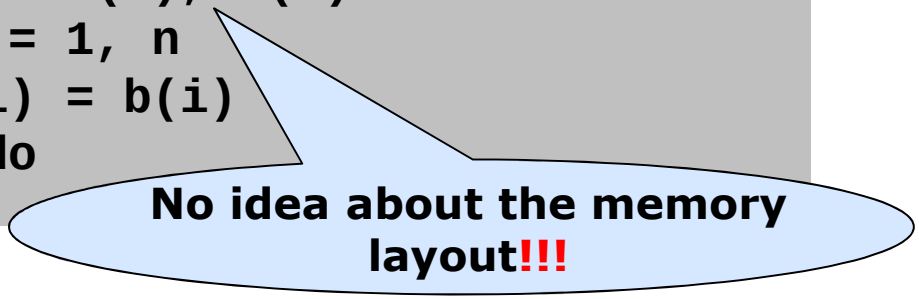
Example: copy-procedure

- Somewhat better "padding"

```
real*8 a(4096), pad(32), b(4096)
common / chaos / a, pad, b
do i = 1, 4096
  a(i) = b(i)
end do
```

- But hard to control:

```
subroutine copy( a, b, n )
real*8 a(n), b(n)
do i = 1, n
  a(i) = b(i)
end do
```



Idea: Associativity

© NEC Corporation 2016

Cache Blocking: Two-way set associative Cache

- Model-architecture
- Line size 64 Byte = 8 x 8 Bytes (8 double precision)
- Cache size 32 kByte → 2^15/2^6 = 2^9 = 512 lines
- Two-way set associative: 2 sets of 256 lines
- Addressing (Byte-address)
  - Set 1: . . . **0010010111010100101 0 1010001000 10000**
  - Set 2: . . . **0010010111010100101 1 1010001000 10000**

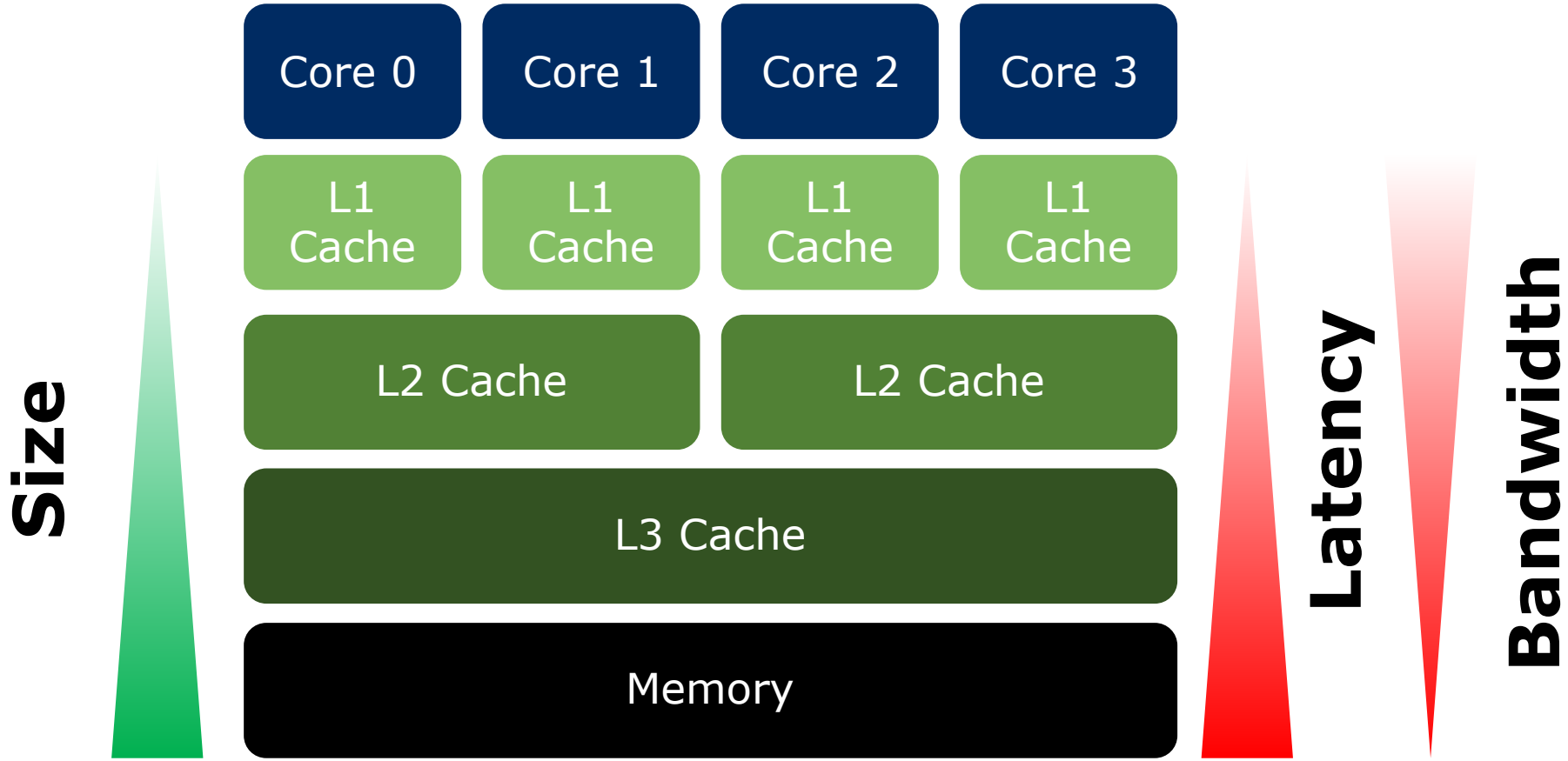
  Cache-Line-# | Word | Byte
- Example: copy-procedure
  - Not so slow:

```
real*8 a(4096), b(4096)
common / chaos / a, b
do i = 1, 4096
  a(i) = b(i)
end do
```

- Selection: LRU – least recently used

© NEC Corporation 2016

Cache Blocking: Cache Levels (1/4)



© NEC Corporation 2016

Cache Blocking: Cache Levels (2/4)

cpuinfo on Intel Broadwell E5-2680v4

```
[rputtin@hsw128 stream-5.10]$ cpuinfo
...
=====  Cache sharing  =====
Cache   Size             Processors

L1      32  KB
(0,28)(1,29)(2,30)(3,31)(4,32)(5,33)(6,34)(7,35)(8,36)(9,37)(10,38)(11,3
9)(12,40)(13,41)(14,42)(15,43)(16,44)(17,45)(18,46)(19,47)(20,48)(21,49)
(22,50)(23,51)(24,52)(25,53)(26,54)(27,55)

L2      256 KB
(0,28)(1,29)(2,30)(3,31)(4,32)(5,33)(6,34)(7,35)(8,36)(9,37)(10,38)(11,3
9)(12,40)(13,41)(14,42)(15,43)(16,44)(17,45)(18,46)(19,47)(20,48)(21,49)
(22,50)(23,51)(24,52)(25,53)(26,54)(27,55)

L3      35  MB
(0,1,2,3,4,5,6,7,8,9,10,11,12,13,28,29,30,31,32,33,34,35,36,37,38,39,40,
41)(14,15,16,17,18,19,20,21,22,23,24,25,26,27,42,43,44,45,46,47,48,49,50
,51,52,53,54,55)
```

every core has its own L1 cache of size 32 KB

every core has its own L2 cache of size 256 KB

All cores on a socket share one L3 cache of size 35 MB

© NEC Corporation 2016

Cache Blocking: Cache Levels (3/4)

© NEC Corporation 2016

Cache Blocking: Cache Levels (4/4)

```
do k = 1, l3_cache_size
  do j = 1, l2_cache_size
    do i = 1, l1_cache_size
      sum = sum &
          + A(i-1,j  ,  k)
          + A(i  ,j-1,k  )
          + A(i  ,j  ,k-1)
          + A(i  ,j  ,  k)
    end do
  end do
end do
```

Core

L1 Cache

L2 Cache

L3 Cache

Memory

Theoretical approach, does not work like this in practice!

© NEC Corporation 2016

Cache Blocking: Idea on operation speed

- How fast are typical operations?

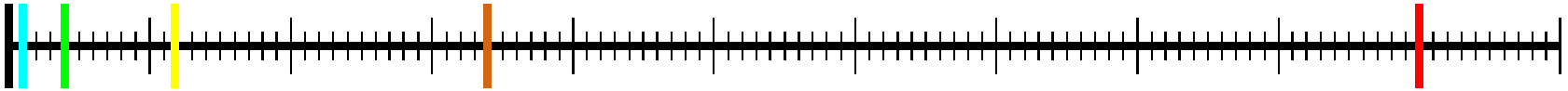
  - Fast: add, multiply, some integer-ops like shift etc.
  - Fair: load and store from L1
  - Slow: load and store from L2
  - Slooow: divide
  - Sloooooow: load and store from L3
  - Slooooooooooow: indirect addressing, gather and scatter
  - Slooooooooo…ooooooooow: … in case of cache misses

- Question: Which operations are the fastest on every architecture?
- Answer: Those operations that are not executed! ☺

© NEC Corporation 2016

Cache Blocking: Broadwell specs

- Intel Broadwell E5-2680v4
- 2 x FMA, with AVX2 in parallel → 16 ops per cycle (DP)
  - 38.4 GFlops per core
- Per CPU: 537.6 GFlops, using AVX2
- Important: **Cache** hierarchy, line size 64 Bytes
  - L1i and L1d per core
    - 32 kByte, 8-way set associative, 4 cycles latency
  - L2 per core
    - 256 kByte 8-way set associative, 12 cycles latency
  - L3 shared
    - 2-45 MByte, 12-16-way set associative, 34 cycles latency
  - L1 and L2 mirrored on L3
  - L3 miss: ~100 cycles latency



© NEC Corporation 2016

Cache Blocking: Strided Access

Fortran:

```
real*8 vector(3,n)
do i = 1, n
  vector(1,i) = …
end do
```

- Dealing with geometry, first dimension 3, or often in QCD, first dim 4

Why? (assume 8-word cacheline)

- Stride 1: all words of a cacheline used 100.0%
- Stride 2: 4 words out of 8 50.0%
- Stride 3: 8/3 words out of 8 26.7%
- Stride 4: 2 words out of 8 25.0%
- Stride 5: 8/5 words out of 8 16.0%

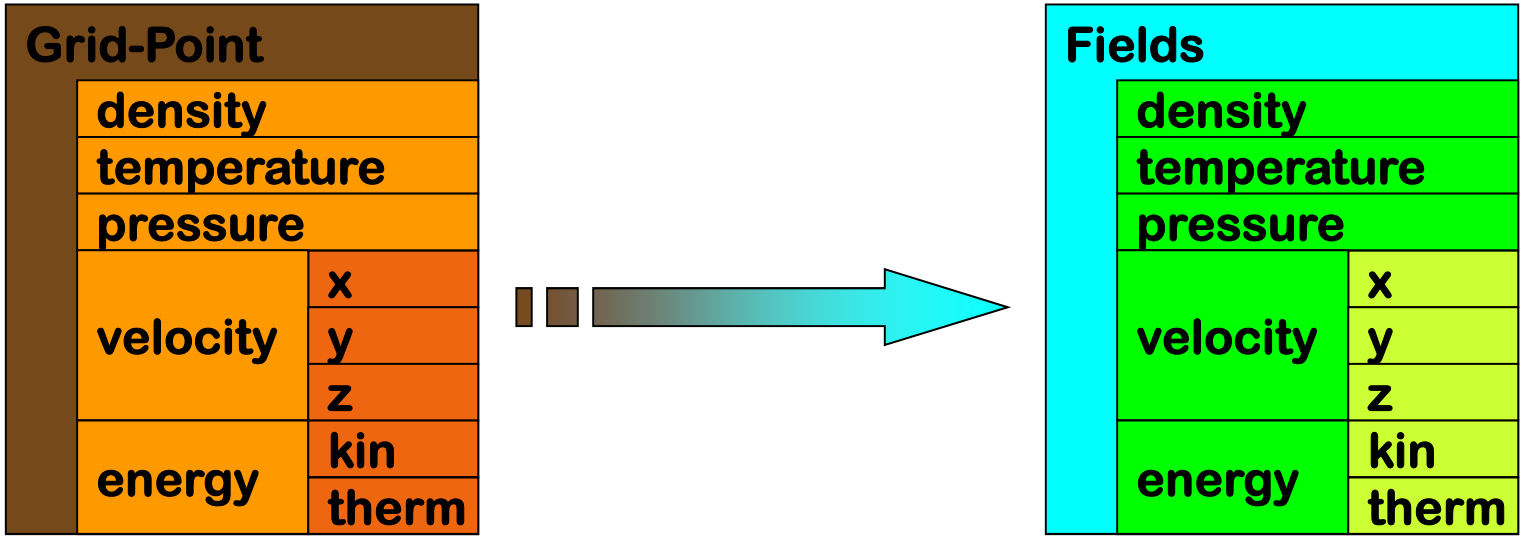
Beware of structures!

- Rule: structure of arrays, not array of structures!

© NEC Corporation 2016

Cache Blocking: Coding paradigms (1/2)

"structure of arrays"
rather than
"array of structures"

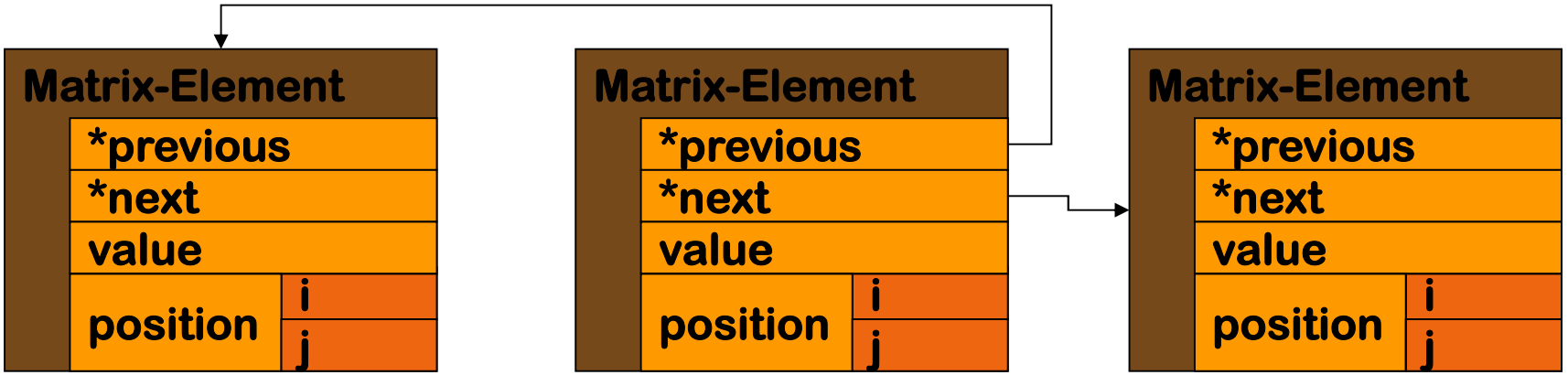


What is the real abstraction to physics?

© NEC Corporation 2016

Cache Blocking: Coding paradigms (2/2)

Killing data-parallelism and potentially locality




```
res[ m_e.position.i ] += m_e.value * rhs[ m_e.position.j ]
*m_e = m_e.next
```

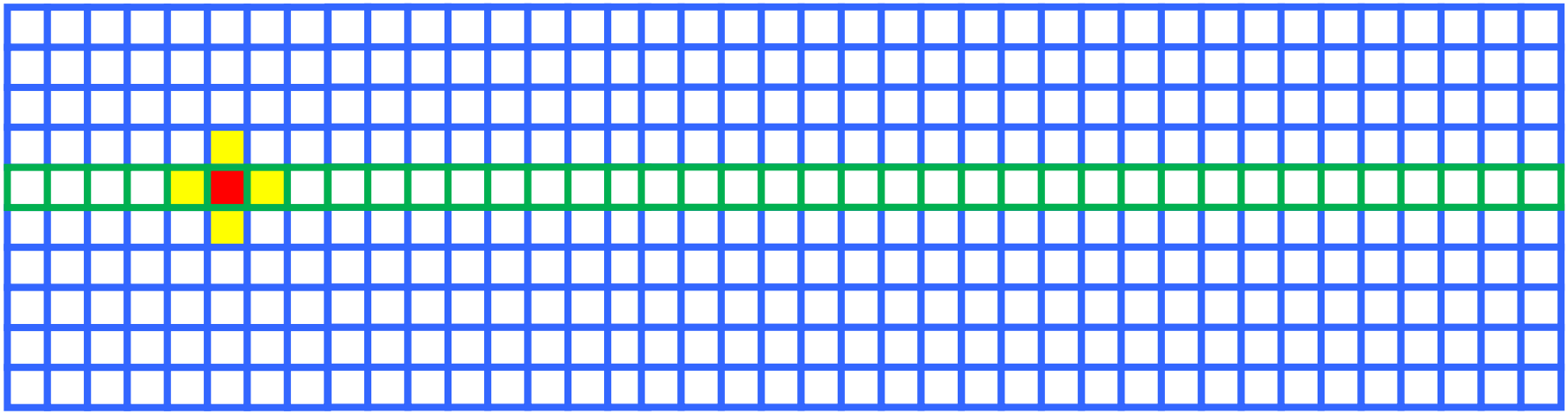

© NEC Corporation 2016

Cache Blocking: 5-pt-differentiation-stencil (1/2)

- Mother Nature can help:
  - Physics, locality!
  - Mathematics, normally some kind of PDE
  - Implies notion of neighbourhood
  - Shows in code

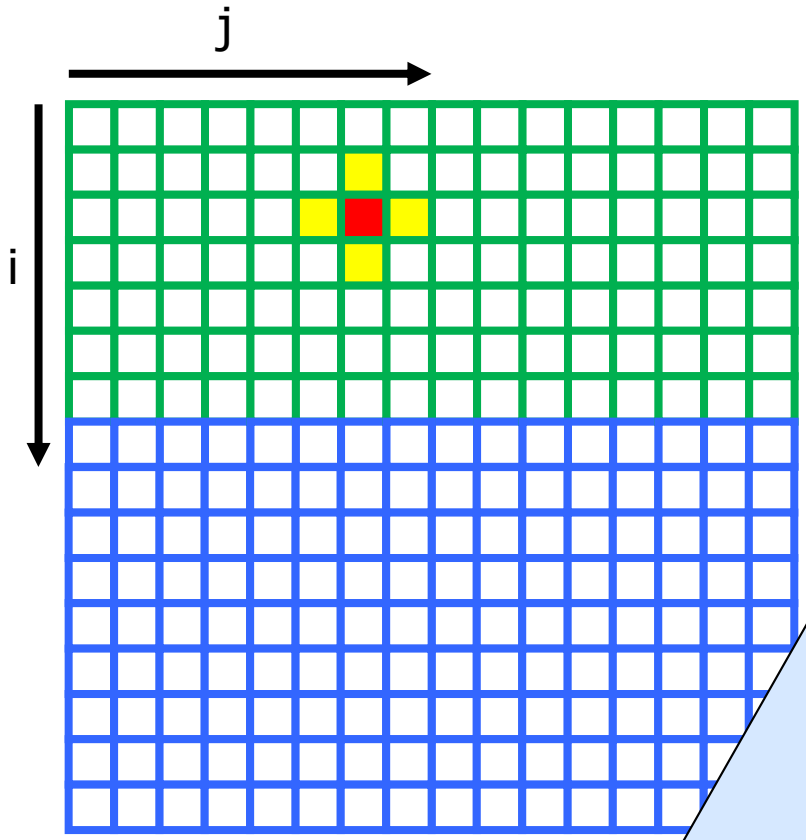
```
real*8 f(m,n), df(m,n)
do j = 2, n-1
  do i = 2, m-1
    df(i,j) = a_0m * f(i  ,j-1) &
            + a_m0 * f(i-1,j  ) &
            + a_00 * f(i  ,j  ) &
            + a_p0 * f(i+1,j  ) &
            + a_pp * f(i  ,j+1)
  end do
end do
```

- Example: 5-pt-differentiation-stencil
- For big m: **`f(i,j-1)`** and **`f(i,j+1)`** are not in cache



© NEC Corporation 2016

Cache Blocking: 5-pt-differentiation-stencil (2/2)




```
real*8 f(m,n), df(m,n)
do ib = 2, m-1, mb
 do j = 1, n
  do i = ib, min(ib+mb-1,m-1)
    df(i,j) = a_0m * f(i  ,j-1) &
            + a_m0 * f(i-1,j  ) &
            + a_00 * f(i  ,j  ) &
            + a_p0 * f(i+1,j  ) &
            + a_pp * f(i  ,j+1)
  end do
 end do
end do
```

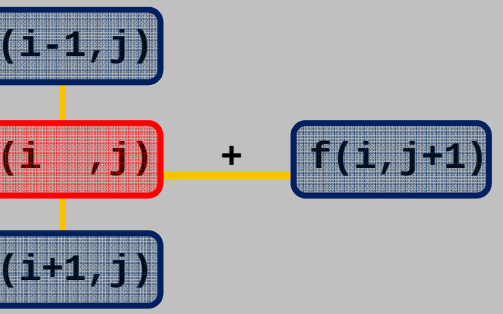
Or just use !dir$ block_loop

© NEC Corporation 2016

Cache Blocking: Data locality

- Locality → “local operators” → “data-locality”

```
do j = 2, m-1
  do i = 2, n-1
    df(i,j) = alpha * (
                        + f(i,j-1) - 4 * f(i  ,j) + f(i,j+1)
                        )
  end do
end do
```



- Question: how many accesses to the memory are needed?
- Answer: 1 load to cache (!) and 1 store per (i,j)
- There is a whole set of techniques and tricks about the principle notion of cache-blocking.

© NEC Corporation 2016

Cache Blocking: levels of parallelism example

- Consequence: Code structure:

```
do k = 2, l-1                                          Threading, OpenMP, and indirectly MPI
  do j = 2, m-1
    do i = 2, n-1                                                                    SIMD
      df(i,j,k) = disc_operator( f(i+{-1,0,1},j+{-1,0,1},k+{-1,0,1}) )
    end do
  end do
end do
```

- independent loop iterations → data parallelism
- “data-locality” → cache-utilisation
- admittedly: for unstructured grids it does not look that simple

© NEC Corporation 2016

OpenMP – Granularity (1/4)

```
do k = 2, n
!$omp parallel do
  do j = 1, m
    do i = 1, L
      f(i,j,k) = a(i,j,k) + f(i,j,k-1)
    end do
  end do
end do

do k = n-1, 1, -1
!$omp parallel do
  do j = 1, m
    do i = 1, L
      f(i,j,k) = b(i,j,k) + f(i,j,k+1)
    end do
  end do
end do
```

SIMD

OpenMP

SIMD

OpenMP

© NEC Corporation 2016

OpenMP – Granularity (2/4)

```
!$omp parallel do
do j = 1, m

  do k = 2, n
    do i = 1, L
      f(i,j,k) = a(i,j,k) + f(i,j,k-1)        SIMD
    end do
  end do

  do k = n-1, 1, -1
    do i = 1, L
      f(i,j,k) = b(i,j,k) + f(i,j,k+1)        SIMD
    end do
  end do

end do
```

OpenMP

© NEC Corporation 2016

OpenMP – Granularity (3/4)

```
!$omp parallel do                                        OpenMP
do j = 1, m
  do i_s = 1, L, 64                                  Cache-Blocking
    i_e = min(i_s+63,L)

    do k = 2, n
      do i = i_s, i_e
        f(i,j,k) = a(i,j,k) + f(i,j,k-1)                 SIMD
      end do
    end do

    do k = n-1, 1, -1
      do i = i_s, i_e
        f(i,j,k) = b(i,j,k) + f(i,j,k+1)                 SIMD
      end do
    end do
  end do
end do
```


© NEC Corporation 2016

OpenMP – Granularity (4/4)

- Intel Broadwell, 2.4GHz, 14 cores
- Dual Socket node → 28 cores, 120 GByte memory, 2400MHz
- Dimensions 1000 x 1000 x 500

	asis	one parallel loop	cache-blocking
standard	**4.20**	**2.82**	**2.05**
first touch	**2.45**	**1.90**	**1.65**

© NEC Corporation 2016

Intel compiler directives: block_loop

block_loop

```
!dir$ block_loop
  do k = 3, o
    do j = 1, n
      do i = 1, m-1
        B(i,j) = B(i,j) + A(i,j,k) + A(i,j-1,k) + A(i,j,k-1)
      end do
    end do
  end do
```

User knowledge:
User knowledge: inner loop is long, therefore j-1, k-1 elements cannot be accessed cache-friendly
=> Set block_loop

Runtime:	17.3 sec
Runtime with directives:	10.0 sec

```
LOOP BEGIN at sub.f90(14,3)
   LOOP BEGIN at sub.f90(14,3)
      LOOP BEGIN at sub.f90(14,3)
          LOOP BEGIN at sub.f90(14,3)
            remark #25442: blocked by 4   (pre-vector)
            LOOP BEGIN at sub.f90(15,5)
               remark #25442: blocked by 10   (pre-vector)
               LOOP BEGIN at sub.f90(16,7)
                  remark #25442: blocked by 128   (pre-vector)
```


© NEC Corporation 2016

Optimization: Strucuture of Arrays (1/6)

```
program test

  type :: coord
    real(kind=8) :: x
    real(kind=8) :: y
    real(kind=8) :: z
    real(kind=8) :: dummy(50)
  end type coord

  integer, parameter :: m = 10000, n = 1000, iter = 100
  type(coord) :: x(m,n), div_x(m,n)
  integer :: i, j, it

  do it = 1, iter
    do j = 1, n
      do i = 1, m
        x(i,j)%x = x(i,j)%x + div_x(i,j)%x
        x(i,j)%y = x(i,j)%y + div_x(i,j)%y
        x(i,j)%z = x(i,j)%z + div_x(i,j)%z
      end do
    end do
  end do

end program test
```

dummy creates cache misses when reading only x, y, z of each coord

Runtime: 16.2 sec

© NEC Corporation 2016

Optimization: Strucuture of Arrays (2/6)

```
   LOOP BEGIN at test.f90(15,5)
      remark #25420: Collapsed with loop at line 16
      remark #15335: loop was not vectorized: vectorization possible
but seems inefficient. Use vector always directive or -vec-threshold0
to override
      remark #15460: masked strided loads: 2
      remark #15462: unmasked indexed (or gather) loads: 1
      remark #15478: estimated potential speedup: 0.920
      LOOP BEGIN at test.f90(16,7)
         remark #25421: Loop eliminated in Collapsing
      LOOP END
      LOOP BEGIN at test.f90(17,9)
         remark #15335: loop was not vectorized: vectorization
possible but seems inefficient. Use vector always directive or -vec-
threshold0 to override
         remark #15450: unmasked unaligned unit stride loads: 2
         remark #15451: unmasked unaligned unit stride stores: 1
         remark #15478: estimated potential speedup: 0.750
remark #25436: completely unrolled by 3
      LOOP END
   LOOP END
```


© NEC Corporation 2016

Optimization: Strucuture of Arrays (3/6)

```
program test
  type :: coord
    real(kind=8) :: x
    real(kind=8) :: y
    real(kind=8) :: z
    real(kind=8) :: dummy(50)
  end type coord
  integer, parameter :: m = 10000, n = 1000, iter = 100
  type(coord) :: x(m,n), div_x(m,n)
  integer :: i, j, it
  do it = 1, iter
    do j = 1, n
!dir$ simd
      do i = 1, m
        x(i,j)%x = x(i,j)%x + div_x(i,j)%x
        x(i,j)%y = x(i,j)%y + div_x(i,j)%y
        x(i,j)%z = x(i,j)%z + div_x(i,j)%z
      end do
    end do
  end do
end program test
```

Runtime: 16.5 sec

© NEC Corporation 2016

Optimization: Strucuture of Arrays (4/6)

```
LOOP BEGIN at test.f90(21,5)
   remark #25420: Collapsed with loop at line 16
   remark #15301: SIMD LOOP WAS VECTORIZED
   remark #15460: masked strided loads: 6
   remark #15462: unmasked indexed (or gather) loads: 3
   remark #15475: --- begin vector loop cost summary ---
   remark #15476: scalar loop cost: 19
   remark #15477: vector loop cost: 18.000
   remark #15478: estimated potential speedup: 1.050
   remark #15488: --- end vector loop cost summary ---
   remark #25015: Estimate of max trip count of loop=1250000

   LOOP BEGIN at test.f90(16,7)
      remark #25421: Loop eliminated in Collapsing

   LOOP END
LOOP END
```


© NEC Corporation 2016

Optimization: Strucuture of Arrays (5/6)

```
program test
  type :: coord
    real(kind=8),allocatable :: x(:,:)
    real(kind=8),allocatable :: y(:,:)
    real(kind=8),allocatable :: z(:,:)
    real(kind=8),allocatable :: d(:,:,:)
  end type coord
  integer, parameter :: m = 10000, n = 1000, iter = 100
  type(coord) :: x, divx
  integer :: i, j, it
  allocate(x%x(m,n), x%y(m,n), x%z(m,n), x%d(m,n,50))
  allocate(divx%x(m,n), divx%y(m,n), divx%z(m,n), divx%d(m,n,50))
  do it = 1, iter
    do j = 1, n
      do i = 1, m
        x%x(i,j) = x%x(i,j) + div_x%x(i,j)
        x%y(i,j) = x%y(i,j) + div_x%y(i,j)
        x%z(i,j) = x%z(i,j) + div_x%z(i,j)
      end do
    end do
  end do
  deallocate(x%x, x%y, x%z, x%dummy, divx%x, divx%y, divx%z, divx%d)
end program test
```

Use structure of arrays instead of array of structures, now x, y and z are accessed cache-friendly

Runtime: 3.3 sec

© NEC Corporation 2016

Optimization: Strucuture of Arrays (6/6)

```
LOOP BEGIN at test.f90(19,7)
<Distributed chunk1>
   remark #25426: Loop Distributed (2 way)
   remark #15301: PARTIAL LOOP WAS VECTORIZED
   remark #15448: unmasked aligned unit stride loads: 1
   remark #15449: unmasked aligned unit stride stores: 1
   remark #15450: unmasked unaligned unit stride loads: 1
   remark #15475: --- begin vector loop cost summary ---
   remark #15478: estimated potential speedup: 3.970
   remark #25015: Estimate of max trip
LOOP END
LOOP BEGIN at test.f90(19,7)
<Distributed chunk2>
   remark #15301: PARTIAL LOO
   remark #15448: unmasked aligned
   remark #15449: unmasked aligned unit stride stores: 1
   remark #15450: unmasked unaligned unit stride loads: 3
   remark #15451: unmasked unaligned unit stride stores: 1
   remark #15475: --- begin vector loop cost summary ---
   remark #15478: estimated potential speedup: 3.450
LOOP END
```

Intel compiler generates two vectorized chunks, with !dir$ simd one combined vectorized loop can be achieved, but it is not faster

© NEC Corporation 2016

Optimization: Transpose (1/4)

```
do j = 1, n
  do i = 1, n
     B(j,i) = A(i,j)
  end do
end do
```

Runtime: 21.9 sec

```
do j = 1, n
  do i = 1, n
     B(i,j) = A(j,i)
  end do
end do
```

Runtime: 21.9 sec

```
B = transpose(A)
```

Runtime: 21.9 sec

for comparison:

```
B = A
```

Runtime: 9.0 sec

© NEC Corporation 2016

Optimization: Transpose (2/4)

```
integer, parameter :: i_blk = 128
do i_s = 1, m, i_blk
  i_e = MIN(i_s + i_blk - 1, m)
  do j = 1, n
    do i = i_s, i_e
      B(j,i) = A(i,j)
    end do
  end do
end do
```

1D-Cache Blocking

Runtime: 16.9 sec

```
integer, parameter :: i_blk = 128, j_blk = 128
do j_s = 1, n,j_blk
  j_e = MIN(j_s + j_blk - 1, n)
  do i_s = 1, m, i_blk
    i_e = MIN(i_s + i_blk - 1, m)
    do j = j_s, j_e
      do i = i_s, i_e
        B(j,i) = A(i,j)
      end do
    end do
  end do
end do
```

2D-Cache Blocking

1D blocking gives most of the speedup already
2D or 3D blocking is only rarely beneficial

Runtime: 15.6 sec

© NEC Corporation 2016

Optimization: Transpose (3/4)

```
integer, parameter :: i_blk = 128
do i_s = 1, m, i_blk
  i_e = MIN(i_s + i_blk - 1, m)
  do j = 1, n
    do i = i_s, i_e
      B(i,j) = A(j,i)
    end do
  end do
end do
```

1D-Cache Blocking

Interchange of indices

Runtime: 13.1 sec

```
integer, parameter :: i_blk = 128, j_blk = 128
do j_s = 1, n,j_blk
  j_e = MIN(j_s + j_blk - 1, n)
  do i_s = 1, m, i_blk
    i_e = MIN(i_s + i_blk - 1, m)
    do j = j_s, j_e
      do i = i_s, i_e
        B(i,j) = A(j,i)
      end do
    end do
  end do
end do
```

2D-Cache Blocking

Interchange of indices

Runtime: 12.6 sec

© NEC Corporation 2016

Optimization: Transpose (4/4)

```
!dir$ block_loop factor(128) level(2)
    do j = 1, n
      do i = 1, n
         B(i,j) = A(j,i)
      end do
    end do
```

1D-Cache Blocking: 128

Runtime: 13.2 sec

```
!dir$ block_loop factor(128)
    do j = 1, n
      do i = 1, n
         B(i,j) = A(j,i)
      end do
    end do
```

2D-Cache Blocking: 128 / 128

Runtime: 12.7 sec

```
!dir$ block_loop
    do j = 1, n
      do i = 1, n
         B(i,j) = A(j,i)
      end do
    end do
```

2D-Cache Blocking:
compiler heuristics

Runtime: 13.9 sec

© NEC Corporation 2016

Optimization: Block-structured hydro code (1/5)

Original code:

```
do k = 1, nk
 do j = 1, nj
  do i = 1, ni
    vic(i,j,k) = vrn(i,j,k) - vrn(i,j-1,k  )
    vkc(i,j,k) = vrn(i,j,k) - vrn(i,j  ,k-1)
  end do
 end do
end do
do k = 1, nk
 do j = 2, nj
  do i = 1, ni
    vi = vic(i,j,k) + vic(i,j-1,k)
    vk = vkc(i,j,k) + vkc(i,j-1,k)
    complicated expressions using
    vi, vk, and several other arrays
  end do
 end do
end do
```

Runtime: 54.1 sec

© NEC Corporation 2016

Optimization: Block-structured hydro code (2/5)

Optimized code:

Joined loops

```
do k = 1, nk
 do j = 1, nj
  do i = 1, ni
    vic(i,j,k) = vrn(i,j,k) - vrn(i,j-1,k  )
    vkc(i,j,k) = vrn(i,j,k) - vrn(i,j  ,k-1)
  end do
 end do


 do j = 2, nj
  do i = 1, ni
    vi = vic(i,j,k) + vic(i,j-1,k)
    vk = vkc(i,j,k) + vkc(i,j-1,k)
    complicated expressions using
    vi, vk, and several other arrays
  end do
 end do
end do
```

Runtime: 52.9 sec

© NEC Corporation 2016

Optimization: Block-structured hydro code (3/5)

Optimized code:

```
do k = 1, nk
 j = 1
  do i = 1, ni
    vic(i,j,k) = vrn(i,j,k) - vrn(i,j-1,k  )
    vkc(i,j,k) = vrn(i,j,k) - vrn(i,j  ,k-1)
  end do

 do j = 2, nj
  do i = 1, ni
    vic(i,j,k) = vrn(i,j,k) - vrn(i,j-1,k  )
    vkc(i,j,k) = vrn(i,j,k) - vrn(i,j  ,k-1)
    vi = vic(i,j,k) + vic(i,j-1,k)
    vk = vkc(i,j,k) + vkc(i,j-1,k)
    …
```

one load removed

Runtime: 44.5 sec

© NEC Corporation 2016

Optimization: Block-structured hydro code (4/5)

Optimized code:

```
do k = 1, nk
 j = 1
  do i = 1, ni
    vic(i,j,k) = vrn(i,j,k) - vrn(i,j-1,k  )
    vkc(i,j,k) = vrn(i,j,k) - vrn(i,j  ,k-1)
  end do
 do j = 2, nj
  do i = 1, ni
    vic_ = vic(i,j-1,k)
    vkc_ = vkc(i,j-1,k)
    vic(i,j,k) = vrn(i,j,k) - vrn(i,j-1,k  )
    vkc(i,j,k) = vrn(i,j,k) - vrn(i,j  ,k-1)
    vi = vic(i,j,k) + vic_
    vk = vkc(i,j,k) + vkc_
    …
```

carry-around scalars

Runtime: 43.2 sec

© NEC Corporation 2016

Optimization: Block-structured hydro code (5/5)

Optimized code:

```
do k = 1, nk
 j = 1
  do i = 1, ni
    vic(i)     = vrn(i,j,k) - vrn(i,j-1,k  )
    vkc(i)     = vrn(i,j,k) - vrn(i,j  ,k-1)
  end do
 do j = 2, nj
  do i = 1, ni
    vic_ = vic(i)
    vkc_ = vkc(i)
    vic(i)     = vrn(i,j,k) - vrn(i,j-1,k  )
    vkc(i)     = vrn(i,j,k) - vrn(i,j  ,k-1)
    vi = vic(i)     + vic_
    vk = vkc(i)     + vkc_
    …
```

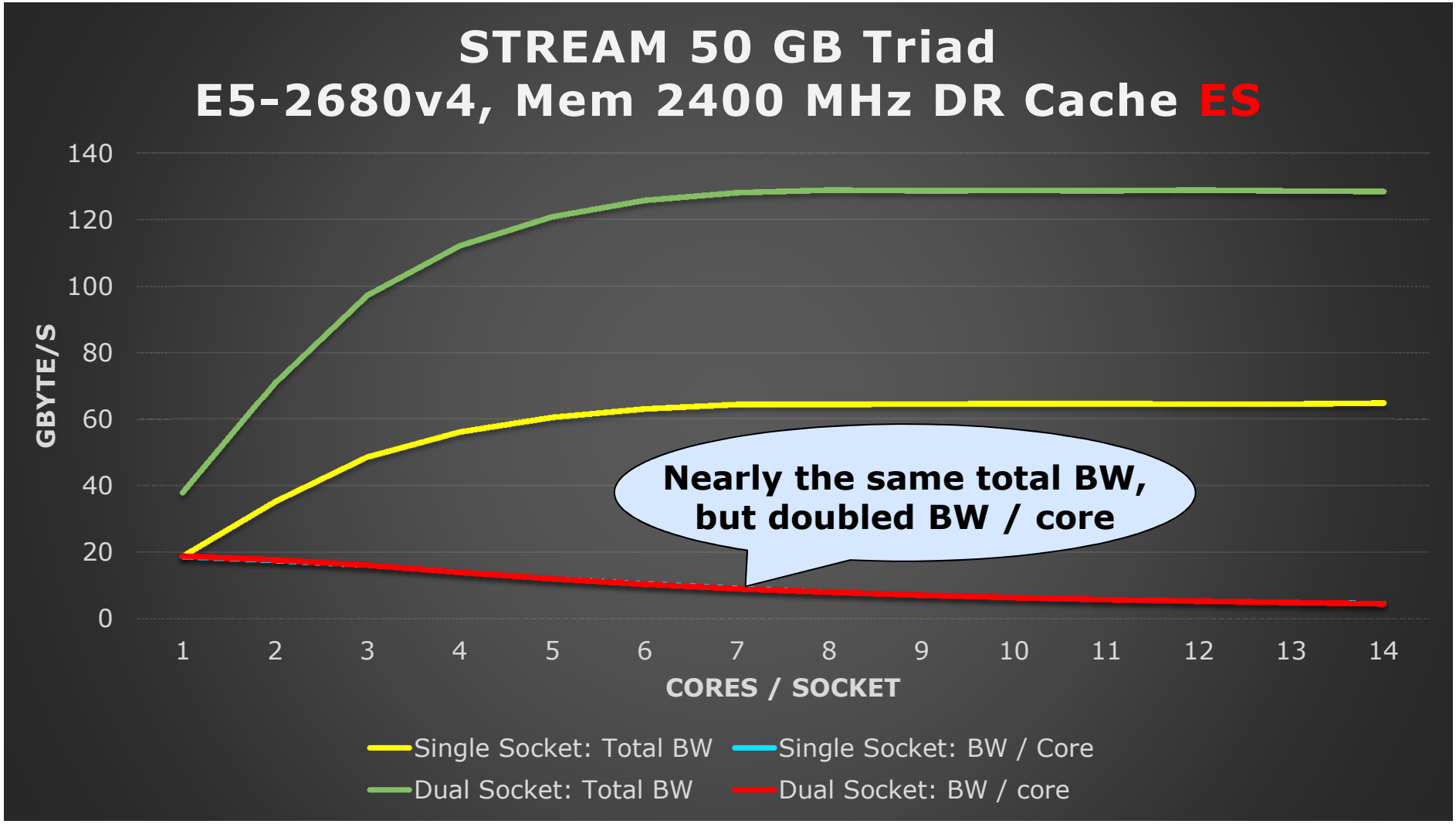
carry-around vector removed one store and load (from mem.)

Runtime: 24.4 sec

© NEC Corporation 2016

Memory: Bandwidth Early Snoop

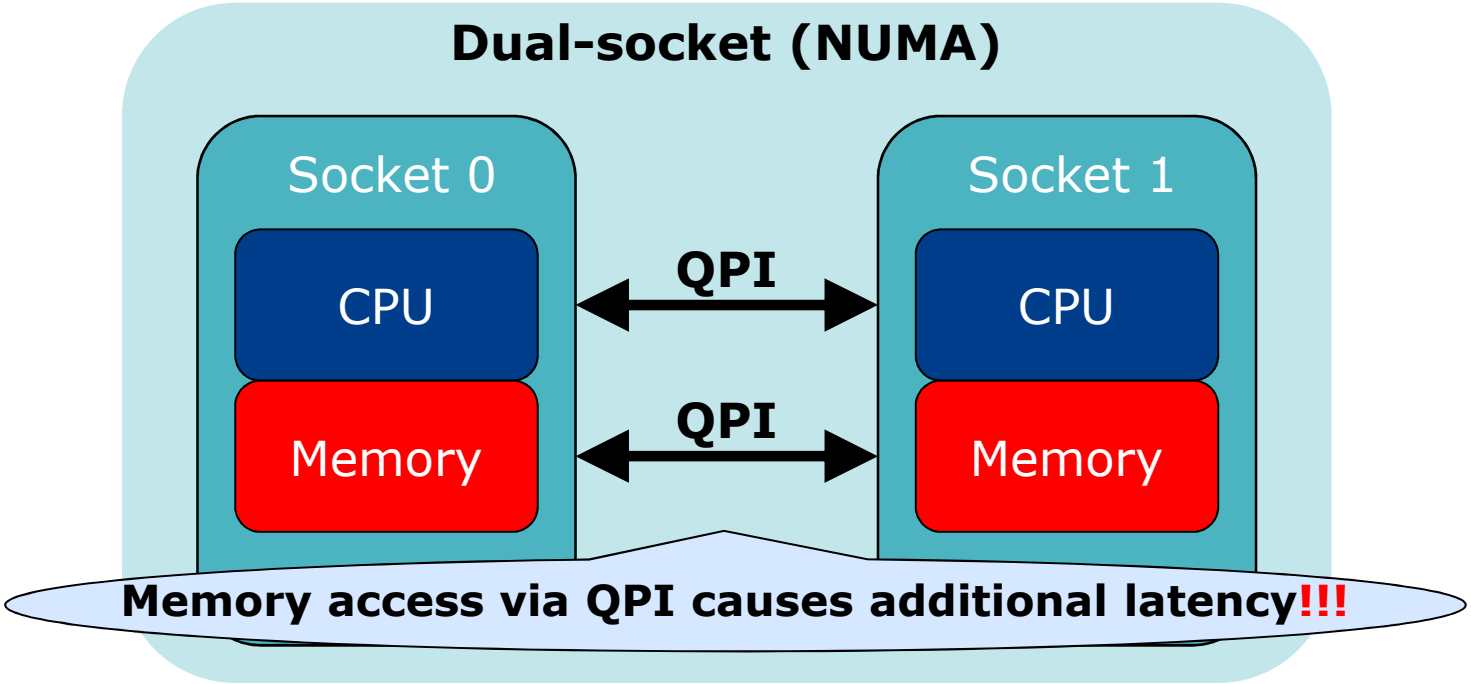
- Memory bandwidth per process depends on number of processes / socket



© NEC Corporation 2016

Memory: Bandwidth QPI (1/2)

- Memory access via QPI (QuickPath Interconnect)



- Theoretical memory bandwidth via QPI:
- **Example: E5-2680v4**
- **#QPIs:** 2
- **QPI-Transferrate:** 9.6 GT/s * 16 bit * (byte / 8 bit) = 19,2 Gbyte/s

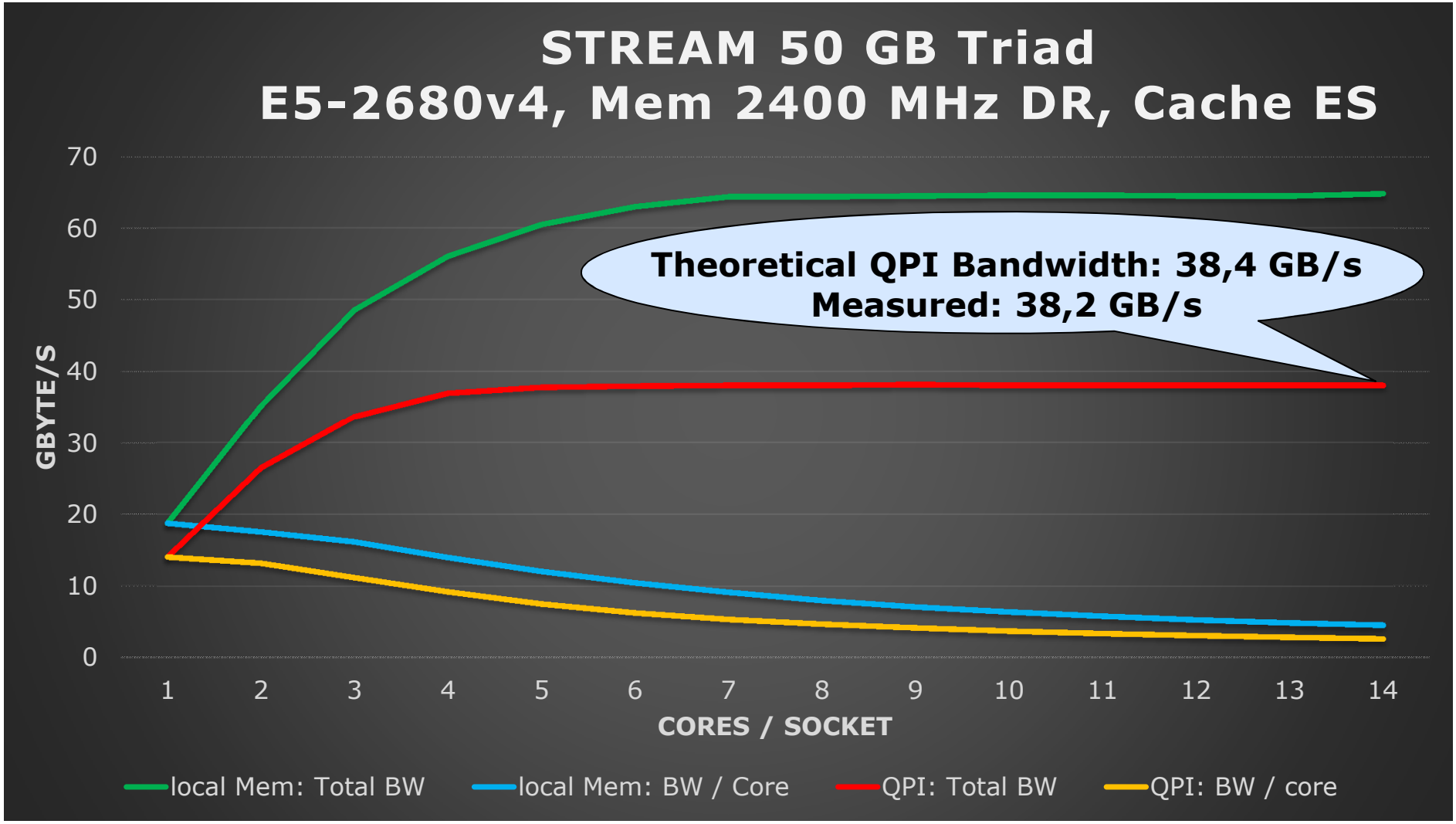
in comparison: theoretical memory bandwidth: 76,8 GB/s

Bandwidth = 2 * 19.2 GByte/s = 38.4 GByte/s

© NEC Corporation 2016

Memory: Bandwidth QPI (2/2)

- Memory bandwidth via QPI is significantly worse



© NEC Corporation 2016

Memory: Cache modes

- Intel Haswell provides 3 cache modes:
  - configured via BIOS, not in user control
  - describes how cache snoop is done

technique ensuring cache coherency: individual caches monitor memory adresses, that they have cached

	ES **E**arly **S**noop	**HS** **H**ome **S**noop	**COD** **C**luster **O**n **D**ie
Snoop	Caching Agent	Home Agent	Check dir. cache, Home Agent
LLC Hit Lat.	Low	Low	**Lowest**
Loc. Mem. Lat.	Medium	Low	Low-High
Loc. Mem. BW	High	High	**Highest**
Rem. Mem. Lat.	**Lowest**	Low	Low-High
Rem. Mem. BW	Medium	**High**	Medium
best use	latency sensitive workloads	NUMA workloads requiring local and remote bandwidth	NUMA optimized workloads (most HPC codes!)

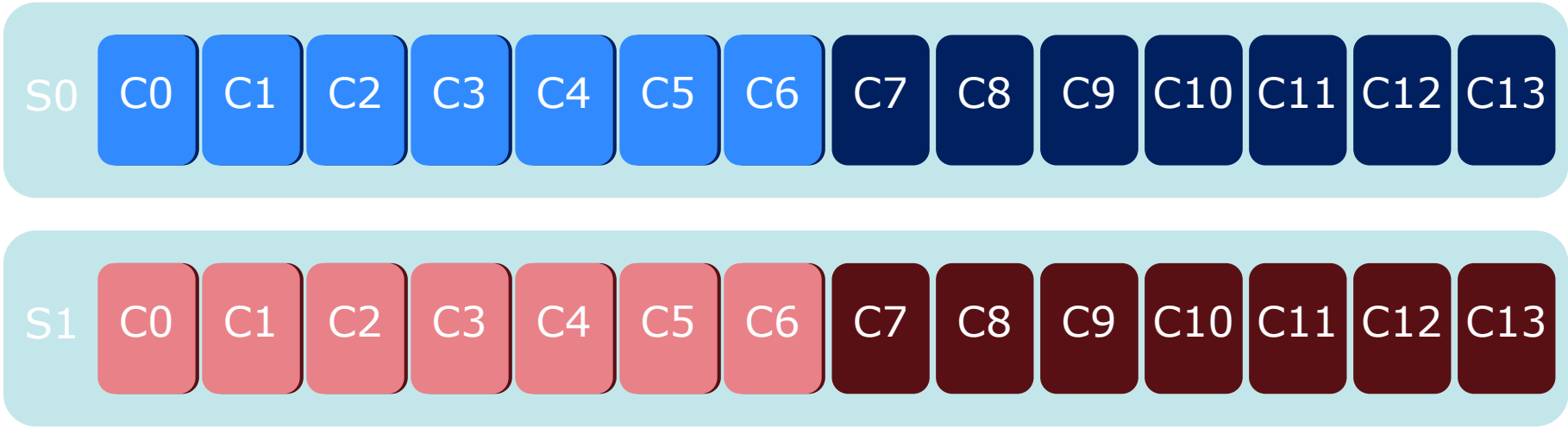
data provided by Intel

© NEC Corporation 2016

Memory: COD

COD – **C**luster **O**n **D**ie

- each socket is logically divided into two clusters

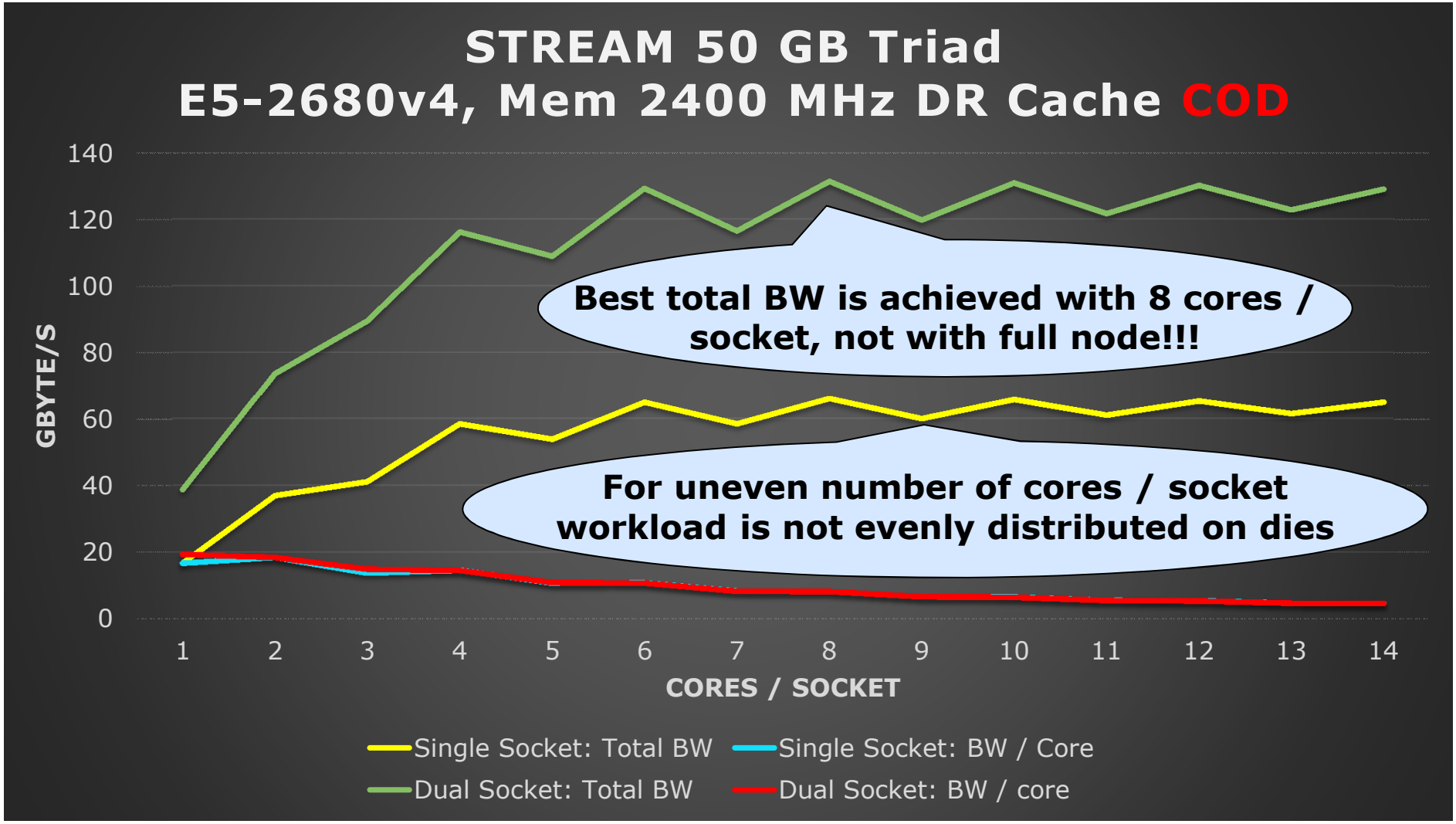


- fewer cache snoops have to be sent to each core
- less snoop traffic arises
- applications have to be NUMA optimized in order to work efficiently
- correct process pinning is much more important
- OpenMP thread numbers have to be chosen according to cluster not socket size
- Number of MPI processes per node should be divisible by 4

© NEC Corporation 2016

Memory: Bandwidth COD

- Memory bandwidth per process depends on number of processes / socket



© NEC Corporation 2016

Cache: Keep in mind

- Keep in mind:

- All codes:

  Ensure data locality!

- Memory bandwidth limited codes:

  Try using fewer cores per node than available!

- Shared memory codes:

  Use first touch policy!

© NEC Corporation 2016

Orchestrating a brighter world

NEC brings together and integrates technology and expertise to create the ICT-enabled society of tomorrow.
We collaborate closely with partners and customers around the world, orchestrating each project to ensure all its parts are fine-tuned to local needs.

Every day, our innovative solutions for society contribute to greater safety, security, efficiency and equality, and enable people to live brighter lives.