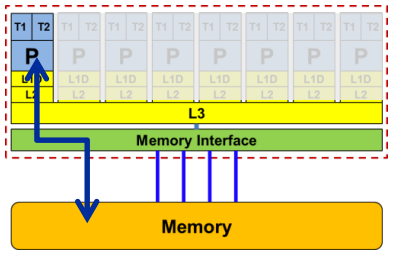
Programming Techniques for Supercomputers:
Modern processors

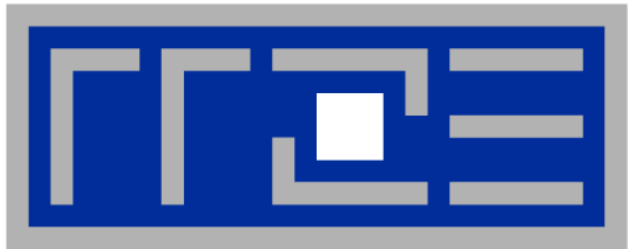
Architecture of the memory hierarchy



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 2020

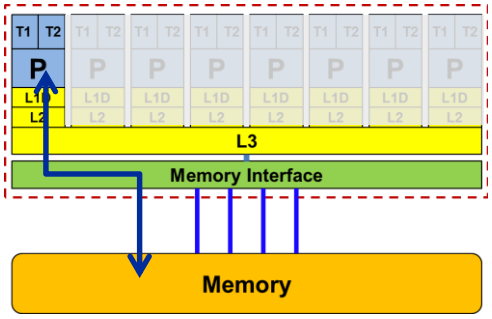


Architecture - Memory hierarchies:

Caches – basics

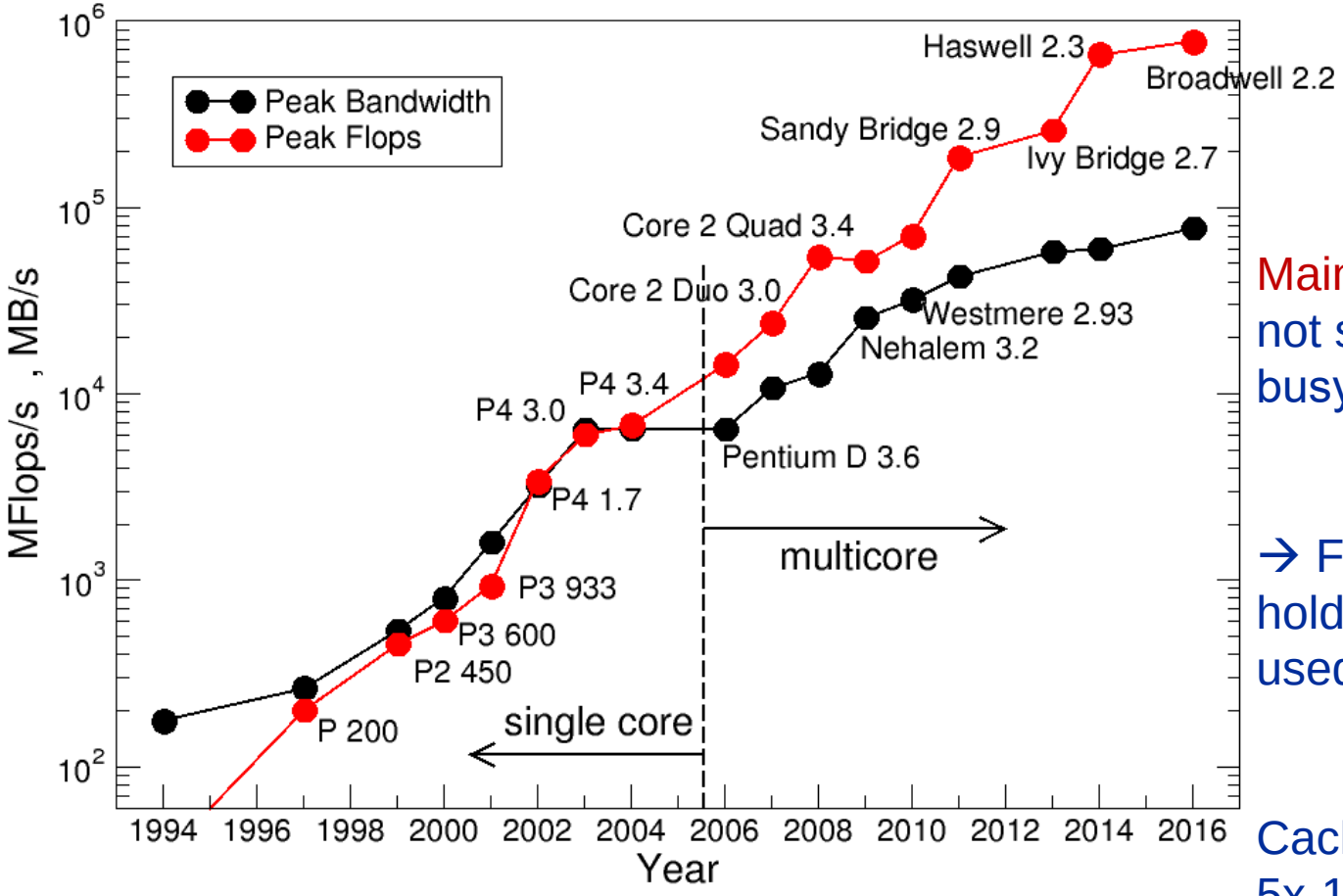
Data access ←→ locality

Cache management



DRAM gap

DP peak performance and peak main memory bandwidth for a single Intel processor (chip)



Approx. 10 F/B

Main memory access speed not sufficient to keep CPU busy…

→ Fast on-chip caches, holding copies of recently used data items

Caches run at CPU clock → 5x-10x faster than memory

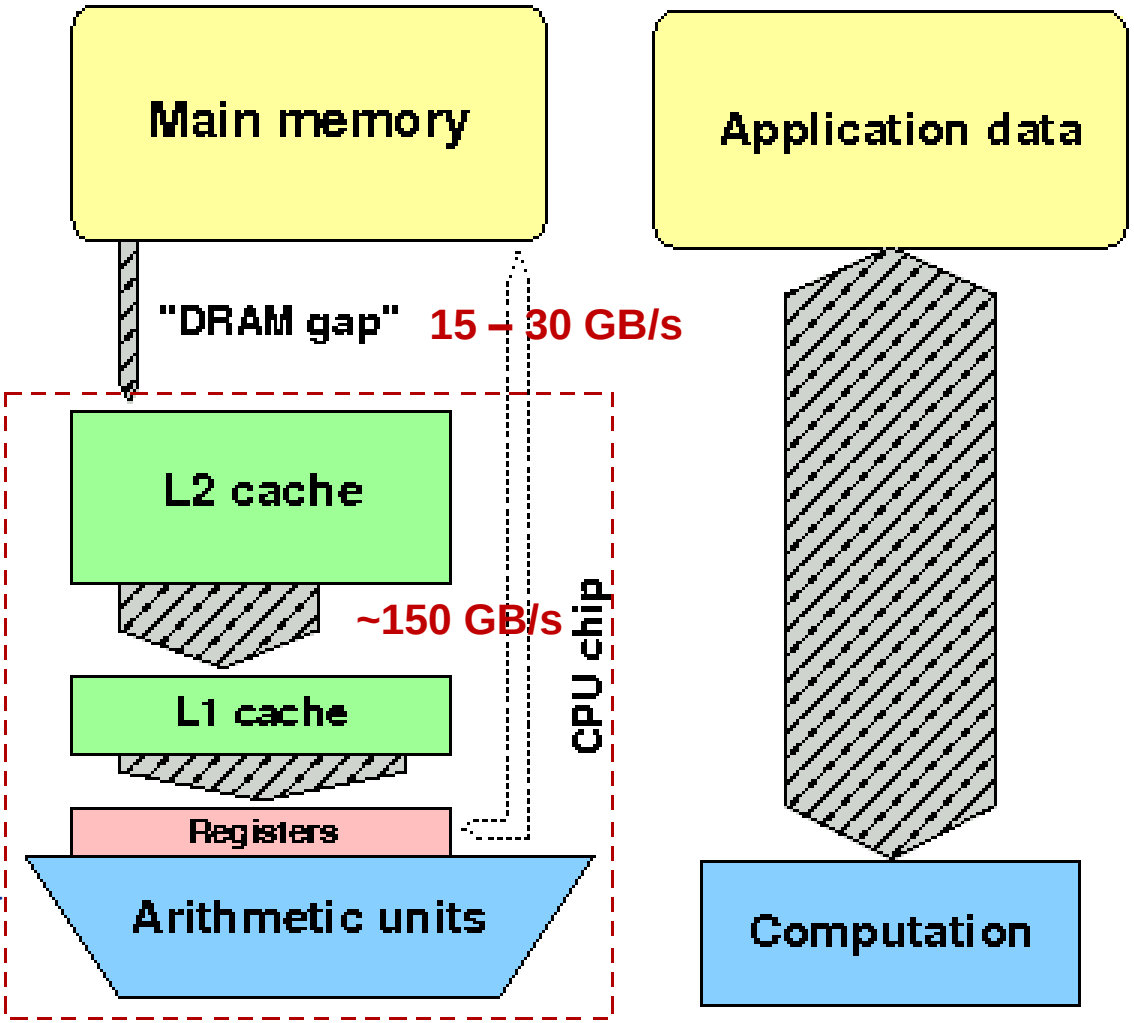
Schematic view of modern memory hierarchy & cache logic

CPU issues a LOAD instruction
Requests innermost cache to transfer a data item to a register

Cache logic automatically checks all cache levels whether data item is already in cache.

If data item is in cache ("cache hit") it is loaded to the register.

If data item is in no cache level ("cache miss") data item is loaded from main memory and a copy is held in cache.



Memory hierarchies: A model for effective bandwidth

Hardware: Quantities to characterize the quality of a memory hierarchy:

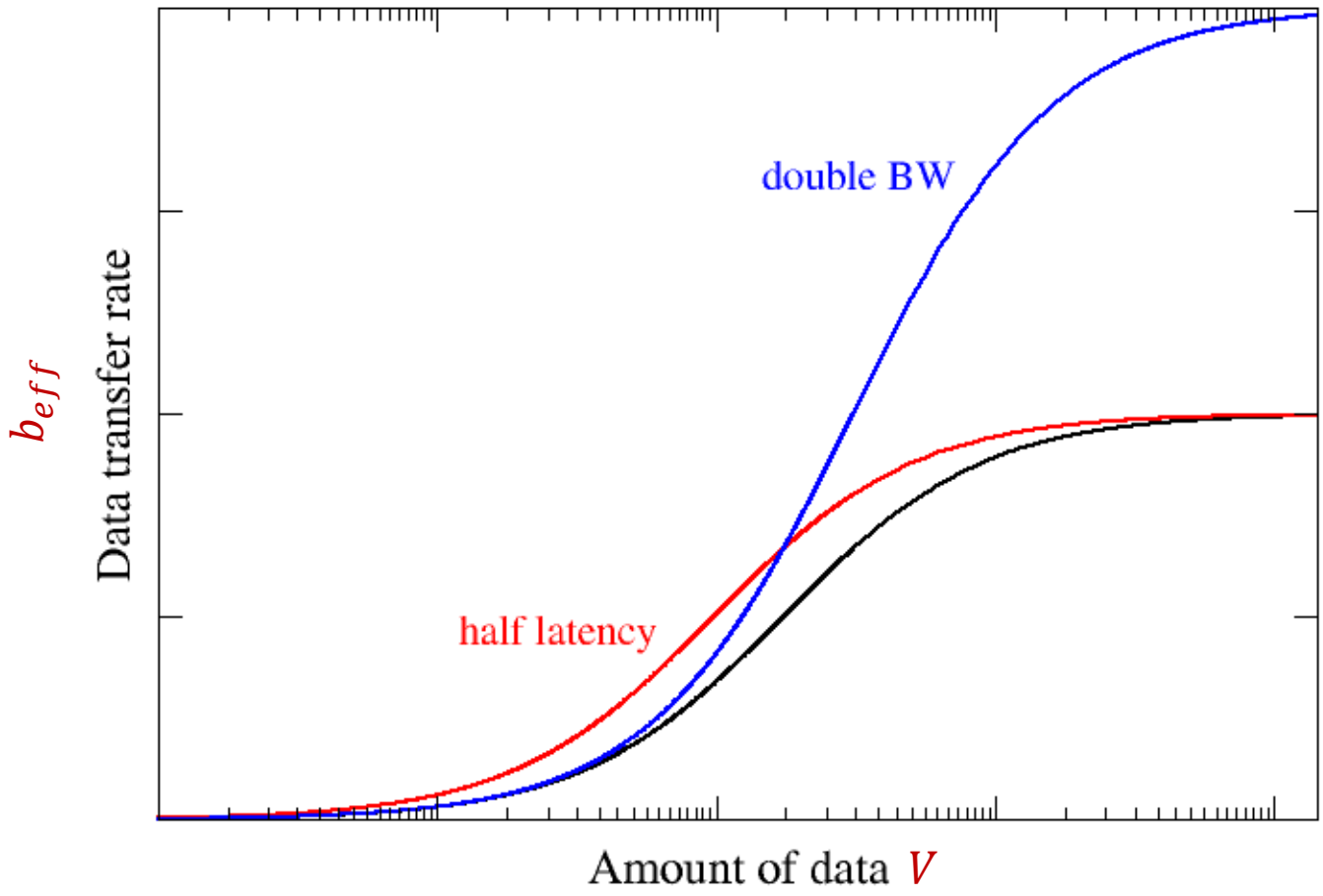
- **Latency (T_l):** Set up time for data transfer from source (e.g., main memory or caches) to destination (e.g., registers)
- **Bandwidth (b):** Maximum amount of data per second which can be transferred between source (e.g., main memory or caches) and destination (e.g., registers)

Application: Transfer time (T) and effective bandwidth (b_{eff}) depend on data volume(V) to be transferred:

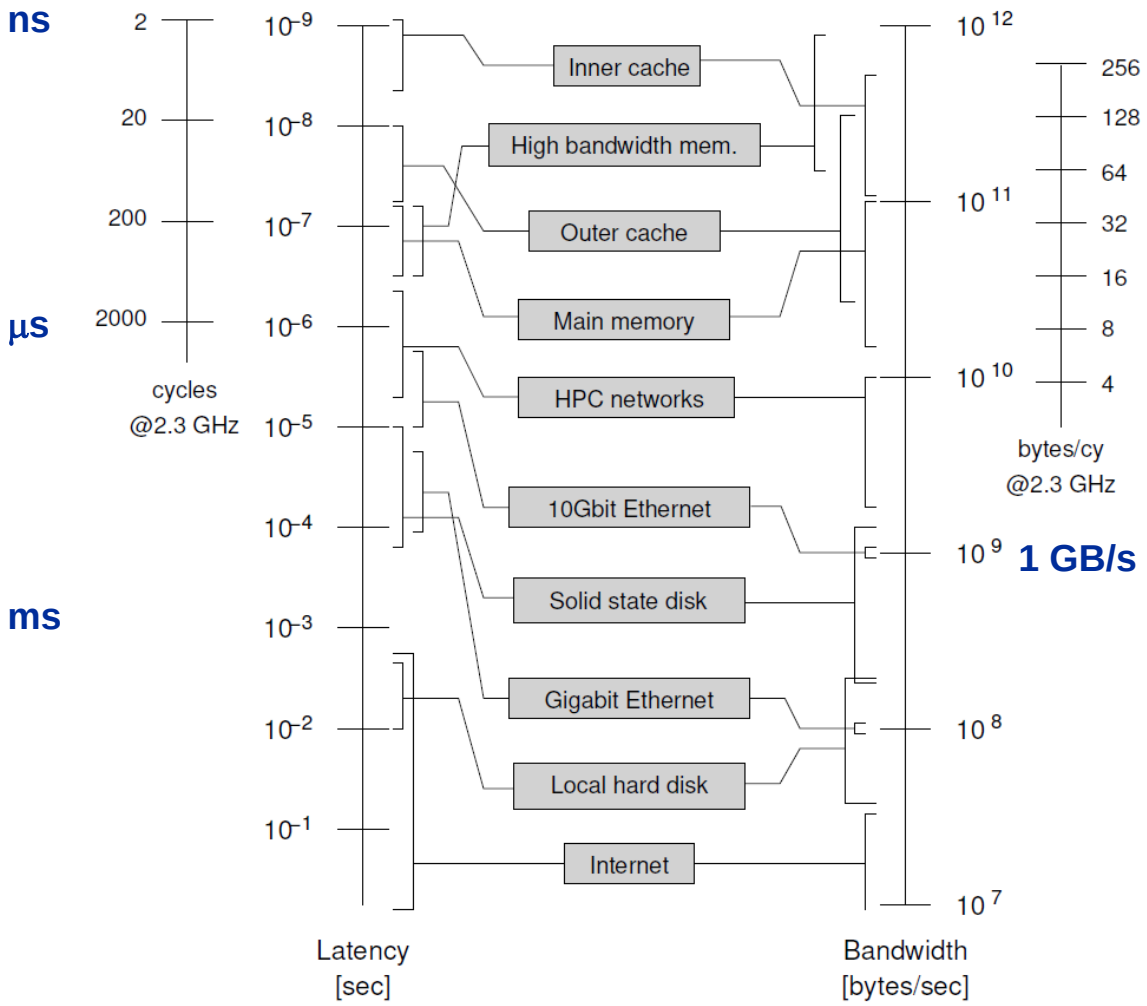
- **Transfer time:** $$T = T_l + \frac{V}{b}$$
- **Effective bandwidth:** $$b_{eff} = \frac{V}{T} = \frac{V}{T_l + {}^{V}/_{b}}$$
  - "Low" data volume $(V \to 0)$: $b_{eff} \approx 0$
  - "Large" data volume $\left(\frac{V}{b} \gg T_l\right)$: $b_{eff} \approx b$

Memory hierarchies: A model for effective bandwidth

$$b_{eff} = \frac{V}{T_l + V/b}$$



Latency and bandwidth in modern computer environments



Memory hierarchies: The latency problem

- Main memory latency and bandwidth for modern multicore CPUs:

$$T_l = 64\,\text{ns} \;\&\; b = 64\,\text{GB/s}$$

V	T_l	V/b	T	b_{eff}
8 B	64 ns	0.125 ns	64.125 ns	0.13 GB/s
128 B	64 ns	2 ns	66 ns	1.9 GB/s
4096 B	64 ns	64 ns	128 ns	32 GB/s

→ Data access is organized in cache lines (CL) – always a full CL is transferred ($V = 64\,\text{B}$ or $V = 128\,\text{B}$ on modern architectures)

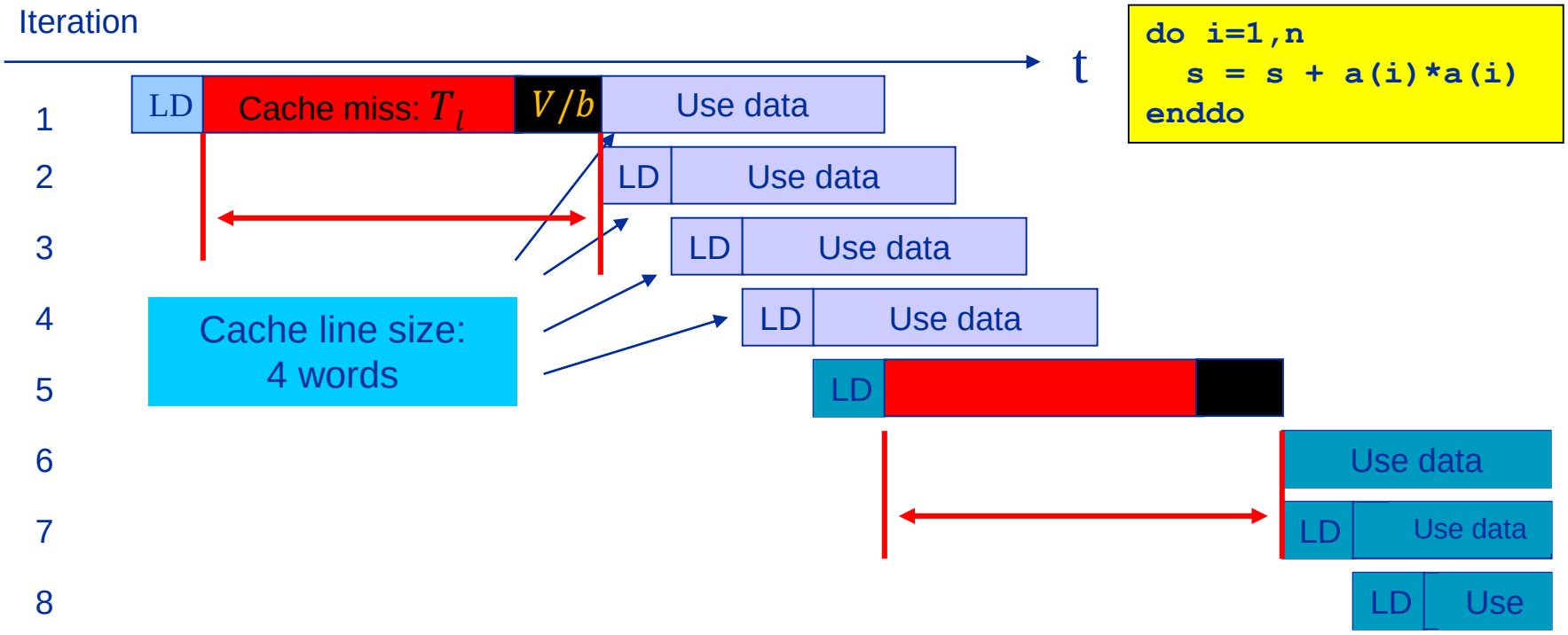
→ Multiple CLs must be loaded concurrently

→ Multiple data requests by application code – “non-blocking loads”

→ Automatic hardware prefetching

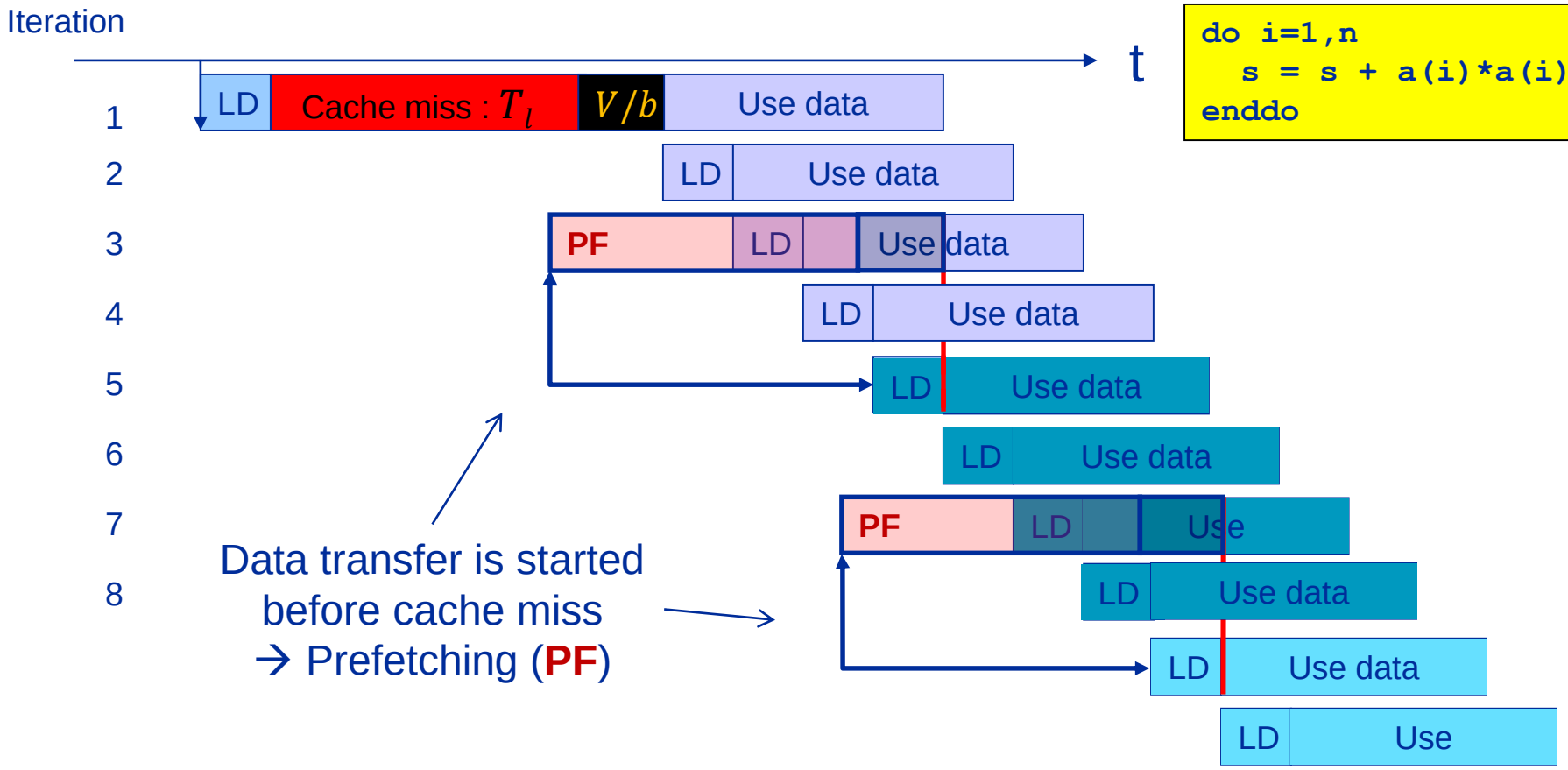
Memory hierarchies: Cache lines

- If one data item is loaded from main memory (“cache miss”), whole cache line it belongs to is loaded
- Cache lines are contiguous in main memory, i.e. “neighboring“ items can then be used from cache



Memory hierarchies: (Automatic) Prefetching

- Prefetching data to hide memory latencies of CL transfers

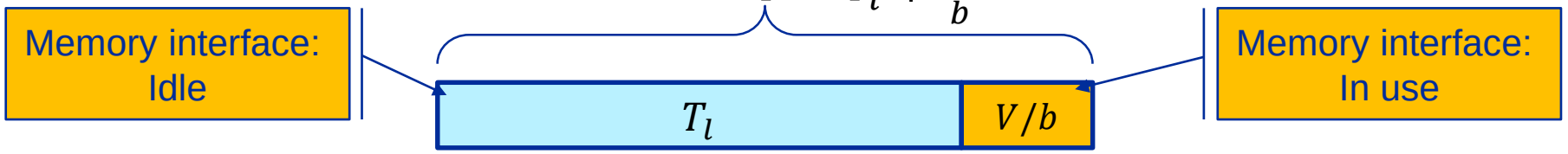


Memory hierarchies: Hiding latency by concurrent PFs/LDs

- How many concurrent prefetches (or LDs) are required to
  - hide the latency (T_l) and eploit the full main memory bandwidth (b) ?

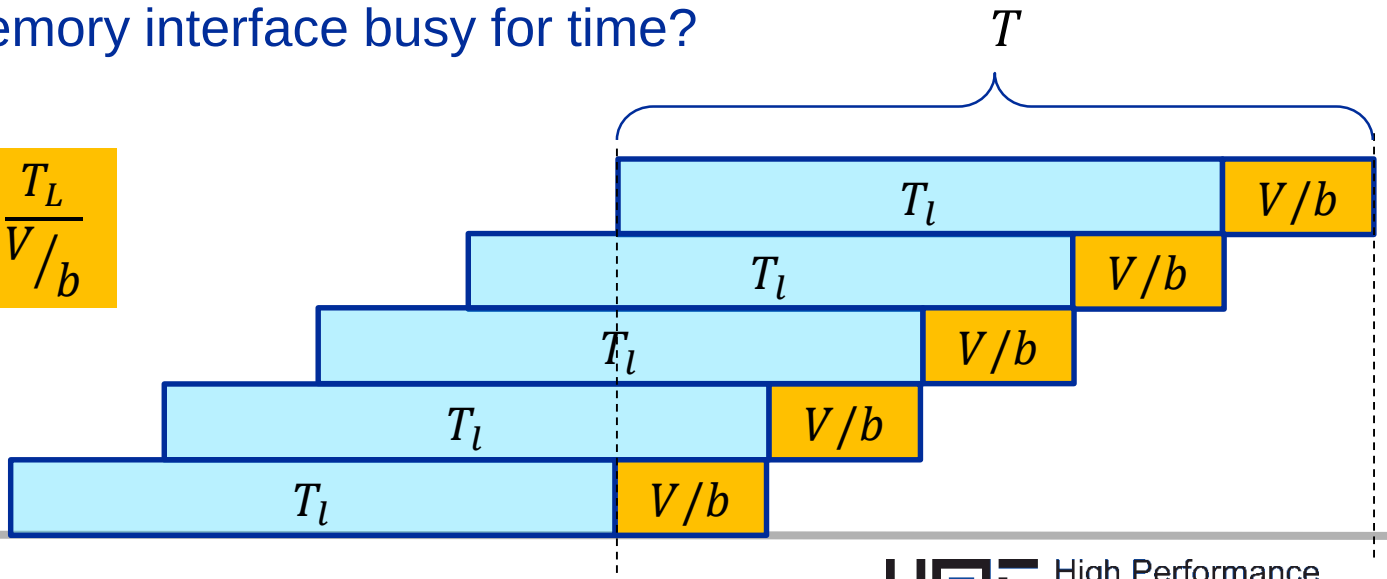
- Loading (LD) one cache line (V Bytes) requires total time

$$T = T_l + \frac{V}{b}$$



- How many concurrent/outstanding PFs (or LDs) (P) are required to keep main memory interface busy for time?

$$P = \frac{T}{V/b} = 1 + \frac{T_L}{V/b}$$



- **Example: CPU@2 GHz**
  - Cache line size 64 B → $V = 64\text{B}$
  - Cache: $T_l = 8\ \text{cy}$ & $b = 32\ \text{B/cy}$ → $P = 1 + \frac{8\ \text{cy}}{64\ \text{B}/32\ \text{B/cy}} = 5$

    → To hide latency, 5 PF/load operations must be active concurrently
  - Main memory: $T_l = 64\ \text{cy}$ & $b = 16\ \text{B/cy}$ → $P = 1 + \frac{64\ \text{cy}}{64\ \text{B}/16\ \text{B/cy}} = 17$

    →To hide latency, 17 PF/load operations must be active concurrently

    →This is 17*64 B = 1088 B (approx. 1 kB) of data „in-flight“

Memory Hierarchies: Prefetching – Hide memory latency

- Prefetch (PFT) instructions (limited use on modern architectures):
  - Transfer one cache line from memory to cache and then issue LD to registers
- Most architectures (Intel/AMD x86, IBM Power) use hardware-based automatic prefetch mechanisms
  - HW detects regular, consecutive memory access patterns (streams) and prefetches at will
  - Intel x86: Adjacent cache line prefetch loads 2 (64-byte) cache lines on L3 miss
    → Effectively doubles line length on loads (typical. enabled in BIOS)
  - Intel x86: Hardware prefetcher:
    Prefetches complete page (4 KB) if 2 successive CLs in this page are accessed
- Regular data access with long loops: Main memory latency is not an issue!
- Excessive data transfers for irregular access pattern or short consecutive loops

Memory Hierarchies: How to determine max. bandwidth from specifications

Intel® Xeon® Gold 6148 Processor

(see https://ark.intel.com/content/www/us/en/ark/products/120489/intel-xeon-gold-6148-processor-27-5m-cache-2-40-ghz.html)

Memory Specifications

Max Memory Size (dependent on memory type)	768 GB
Memory Types	DDR4-2666
Maximum Memory Speed	2666 MHz
Max # of Memory Channels	6
ECC Memory Supported ‡	Yes

f_{MEM}

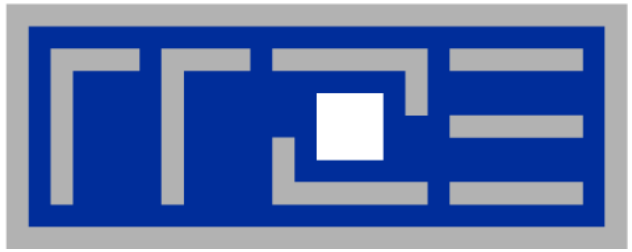
#Channels

- Theoretical bandwidth of DDRx configuration:

$$b_{peak} = \#Channels \times f_{MEM} \times 8 \frac{B}{cycle}$$

- For above configuration we get:

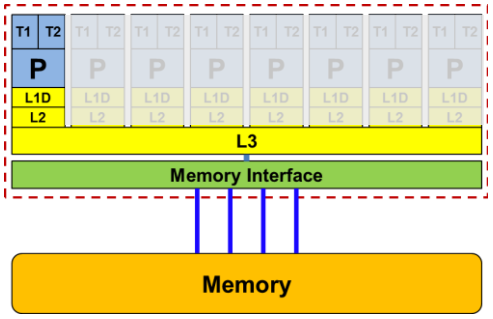
$$b_{peak} = 6 \times 2{,}666 \frac{Mcycle}{s} \times 8 \frac{B}{cycle} = 127{,}968 \frac{MB}{s} = 128 \frac{GB}{s}$$



Node-level architecture - revisited

Memory hierarchy:

- Caches – basics
- **Data access ←→ locality**
- Cache management



Memory Hierarchies: Cache line ←→ Spatial access locality

- Cache line features
  - Cache line use is optimal for contiguous access (“stride 1”) → **STREAMING**
  - Non-consecutive access reduces performance
  - Access with wrong stride (e.g. cache line size) can lead to disastrous performance breakdown
  - Typical CL sizes: 64 Byte (AMD/Intel) or 128 Byte (IBM)

- **“Spatial locality”:** Ensure accesses to “neighboring” data items

GOOD (“Streaming”)

```
do i=1,n
  s = s + a(i)*a(i)
enddo
```

BAD (“Strided”)

```
do i=1,n,2
  s = s + a(i)*a(i)
enddo
```

If `a(1:n)` is loaded from main memory: ≈same runtime!

→ Performance of strided loop is half of the continuous one

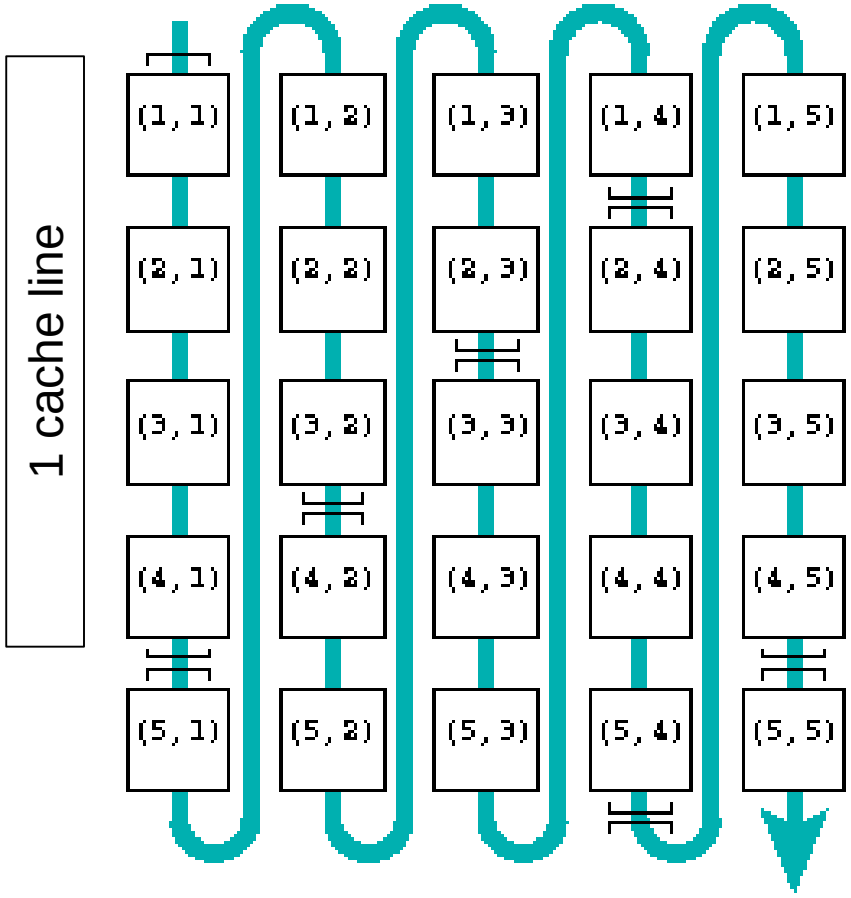
- How to traverse multidimensional arrays?!
- Example: Initialize matrix A with A(i,j) = i*j
- What is the storage order of multidimensional-data structure?
- It depends, e.g. 2-dimensional 3x3 array A of doubles
- FORTRAN: column by column („column major order“)

0 B	Memory layout							71 B
A(1,1)	A(2,1)	A(3,1)	A(1,2)	A(2,2)	A(3,2)	A(1,3)	A(2,3)	A(3,3)

- C/C++: row by row („row major order“)

0 B	Memory layout							71 B
A[0][0]	A[0][1]	A[0][2]	A[1][0]	A[1][1]	A[1][2]	A[2][0]	A[2][1]	A[2][2]

- Default layout for FORTRAN: column by column (column major order)



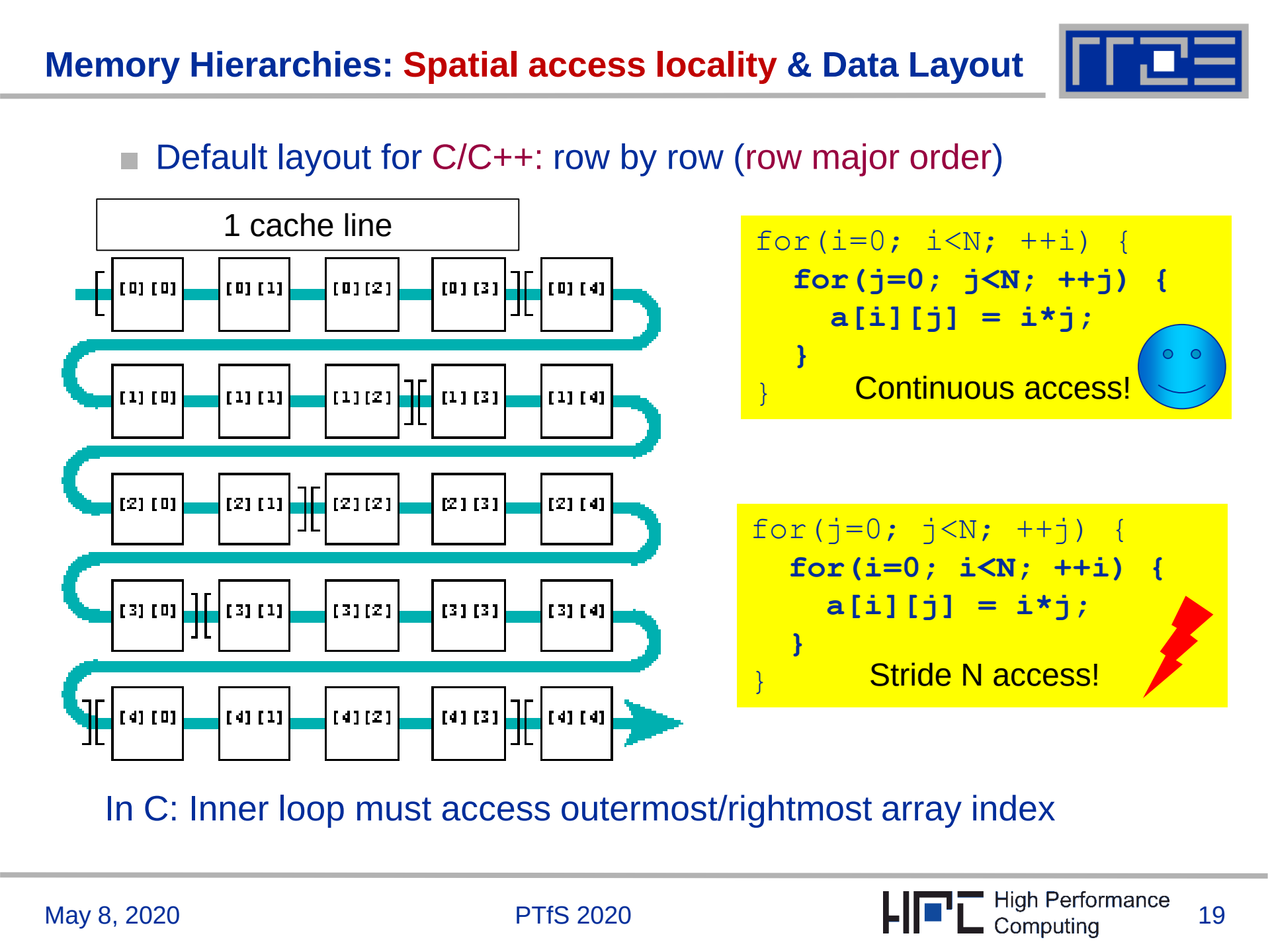

```
do i=1,n
   do j=1,n
      a(j,i)=i*j
   enddo
enddo
```

Continuous access!

```
do j=1,n
   do i=1,n
      a(j,i)=i*j
   enddo
enddo
```

Stride n access!

FORTRAN: Inner loop must access innermost/left array index



- Default layout for C/C++: row by row (row major order)


```
for(i=0; i<N; ++i) {
  for(j=0; j<N; ++j) {
    a[i][j] = i*j;
  }
}
```

Continuous access!

```
for(j=0; j<N; ++j) {
  for(i=0; i<N; ++i) {
    a[i][j] = i*j;
  }
}
```

Stride N access!

In C: Inner loop must access outermost/rightmost array index

- 3-dimensional arrays in C/C++

```
for(i=0; i<N; ++i) {
  for(j=0; j<N; ++j) {
    for(k=0; k<N; ++k) {
      a[i][j][k] = i*j*k;
    }
  }
}
```

Continuous access!

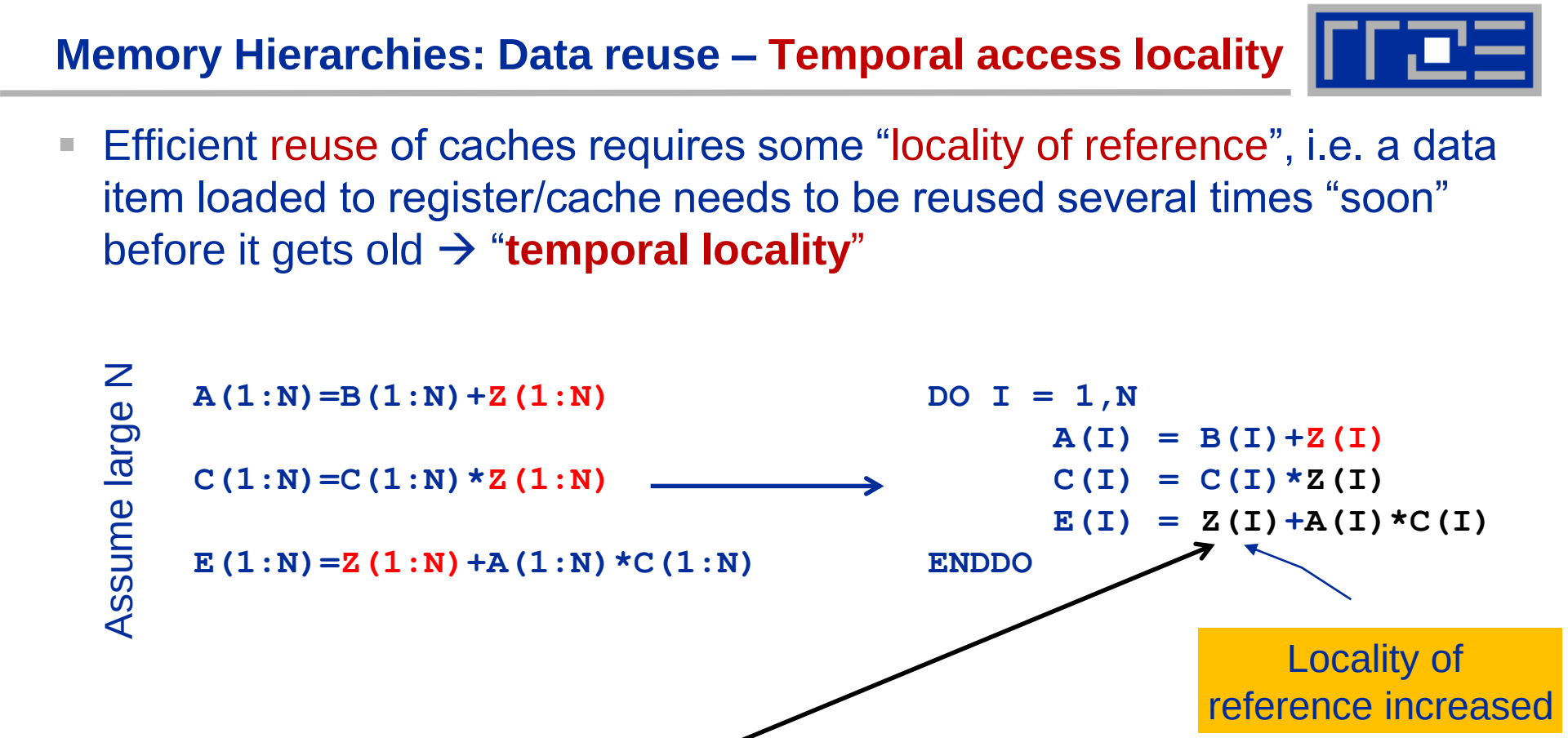
```
for(k=0; k<N; ++k){
  for(j=0; j<N; ++j) {
    for(i=0; i<N; ++i) {
      a[i][j][k] = i*j*k;
    }
  }
}
```

Stride N*N access!

- C/C++: Always start with rightmost index as inner loop index – if possible!

- Sometimes there are problems…. (spatial blocking may improve the situation here)

```
for(i=0; i<N; ++i) {
  for(j=0; j<N; ++j) {
    a[i][j] = b[j][i];
  }
}
```



- Efficient reuse of caches requires some "locality of reference", i.e. a data item loaded to register/cache needs to be reused several times "soon" before it gets old → "**temporal locality**"

Assume large N

```
A(1:N)=B(1:N)+Z(1:N)

C(1:N)=C(1:N)*Z(1:N)

E(1:N)=Z(1:N)+A(1:N)*C(1:N)
```

```
DO I = 1,N
      A(I) = B(I)+Z(I)
      C(I) = C(I)*Z(I)
      E(I) = Z(I)+A(I)*C(I)
ENDDO
```

Locality of reference increased

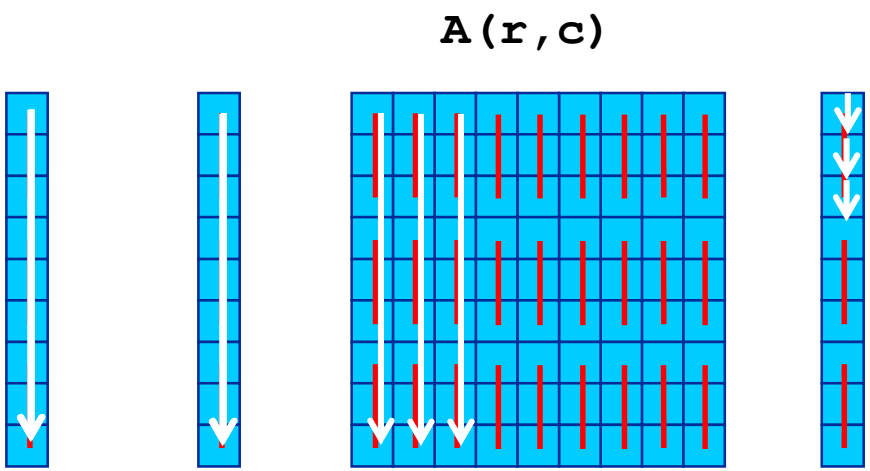
- Instead of reloading data from main memory (left), several accesses are served (right) by inner most cache or by register

Memory hierarchies: Temporal access locality

- **Temporal locality:** If data is already in cache - reuse it from there!
- Example: Dense matrix vector multiplication (assume that cache is large enough to hold `y(1:R)`)

`tmp` stays in register in inner loop

`x(c)` stays in register




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

`y(1:R)` is loaded `C` times
→ **Temporal locality** for `C-1` accesses

`A(,)`: continuous access
→ Spatial locality

Memory Hierarchies: Temporal locality – "Theory"

- **Boosting performance by reusing data from fast caches:**
  - Data access time main memory (cache): T_M (T_C); ratio: $\tau = T_M/T_C$ (>>1)
  - β: cache reuse ratio (fraction of loads/stores giving cache hits)
  - Average time for data transfer cost (depending on reuse factor β)

$$T_{av}(\beta) = \beta * T_C + (1 - \beta) * T_M$$

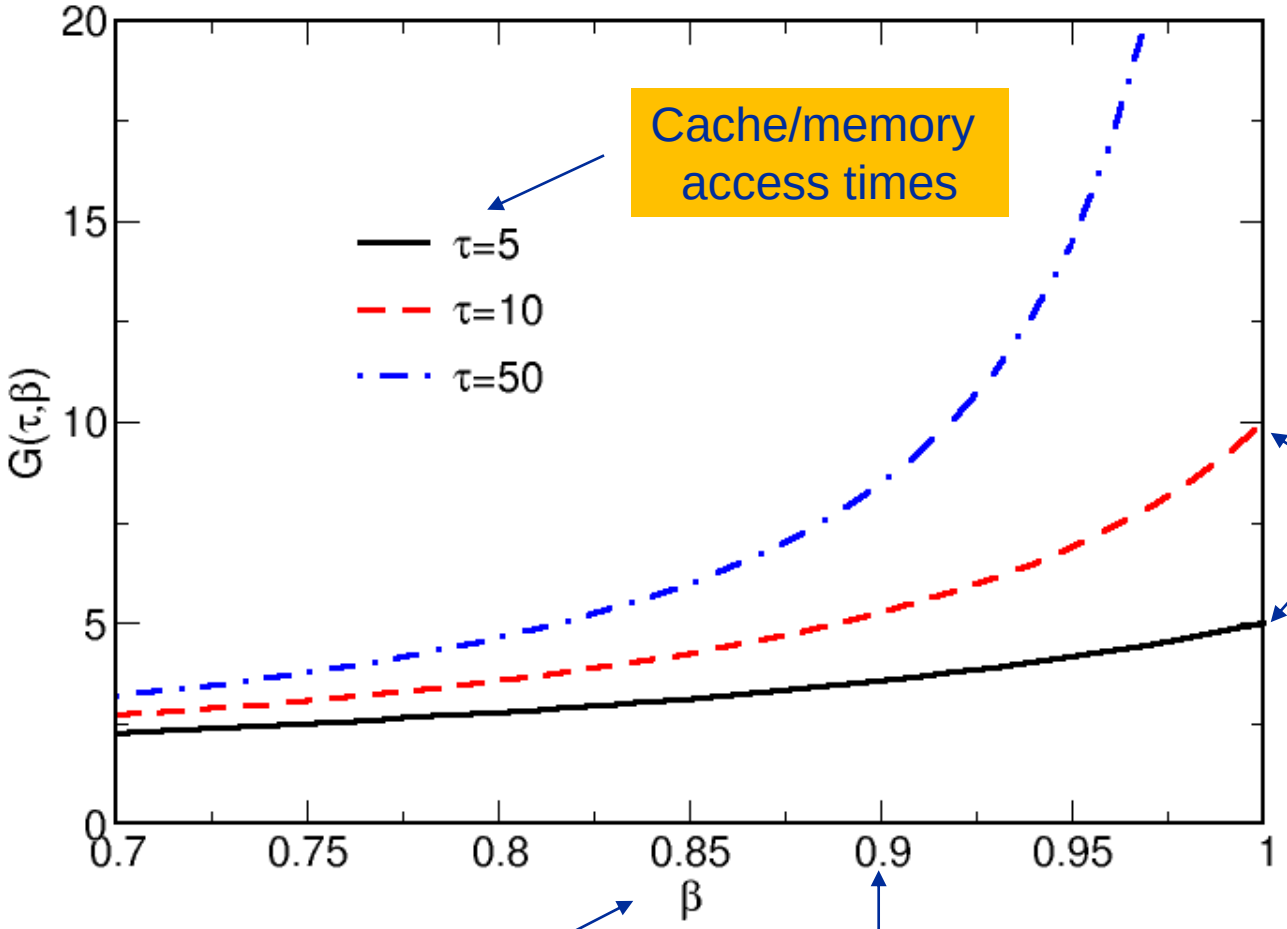
  - Check limits:
    $T_{av}(0)=T_M$ (no reuse from cache → limited by memory speed)

    $T_{av}(1)=T_C$ (all data from cache → limited by cache speed)

- **Potential speed-up of using a cache (→ Amdahl's law)**

$$G(\tau,\beta) = \frac{T_m}{T_{av}} = \frac{\tau T_c}{\beta T_c + (1-\beta)\tau T_c} = \frac{\tau}{\beta + \tau(1-\beta)}$$

Memory hierarchies: Gains through temporal locality



Cache/memory access times

$G(\tau,1) = \tau$

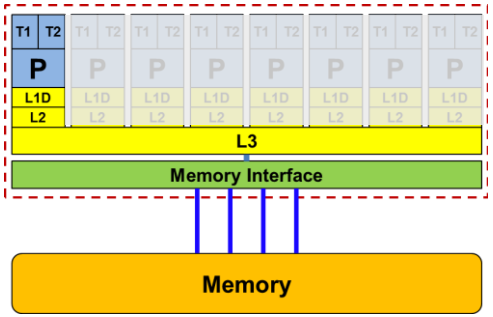
Cache reuse ratio

β=0.9 → load once from main memory and reuse it 9 times!

Node-level architecture - revisited

Memory hierarchy:

- Caches – basics
- Data access ←→ locality
- **Cache management**

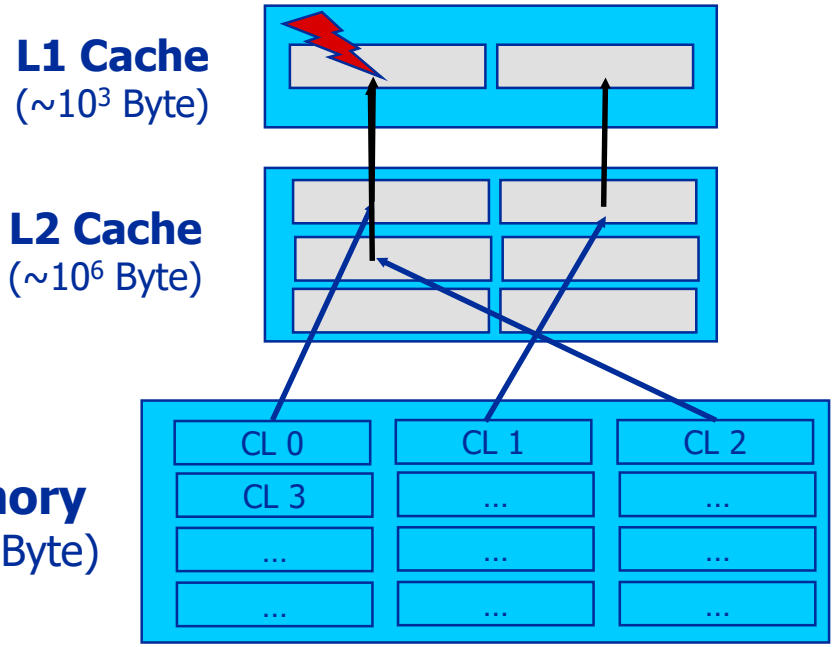


Memory Hierarchy: Cache management – Basics

- Size of caches (KB, MB) much smaller than main memory size (GB)
- If cache is full “data items” need to be replaced when new data comes in
- Replacement based on “age” of cache line ←→ Last access time
- Cache lines get “old” if they are not accessed for some time
- Which “old” cache line to replace? (“Replacement policy”)
  - Least recently used (LRU) – Not recently used (NRU) – (Random)
- Important question: How is the pairing/mapping of memory addresses to cache locations?

Memory Hierarchies: Cache Mapping

- Cache Mapping
  - Pairing of memory addresses with cache locations
  - Where is the CL to given memory addressed placed in the cache?



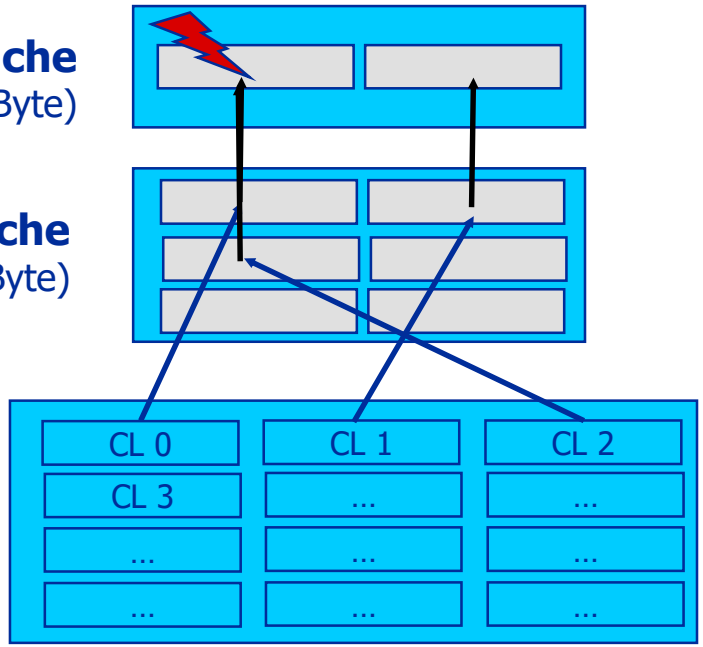
Memory address 32 Bit:
01110010000 11110100111110 001111

- Why (simple) strategies?
  - Hardware needs to search for data in cache, e.g.
    - is requested data in cache (hit or miss)?
    - If it is in cache – where is it?
    - If new data comes in – where is the old data to override
- Each CL has a Cache Tag: Holds status information of CL (not visible to programmer)

Memory Hierarchies: Cache Mapping – direct mapping

- Directly Mapped caches → each CL can only be mapped to one location in each cache

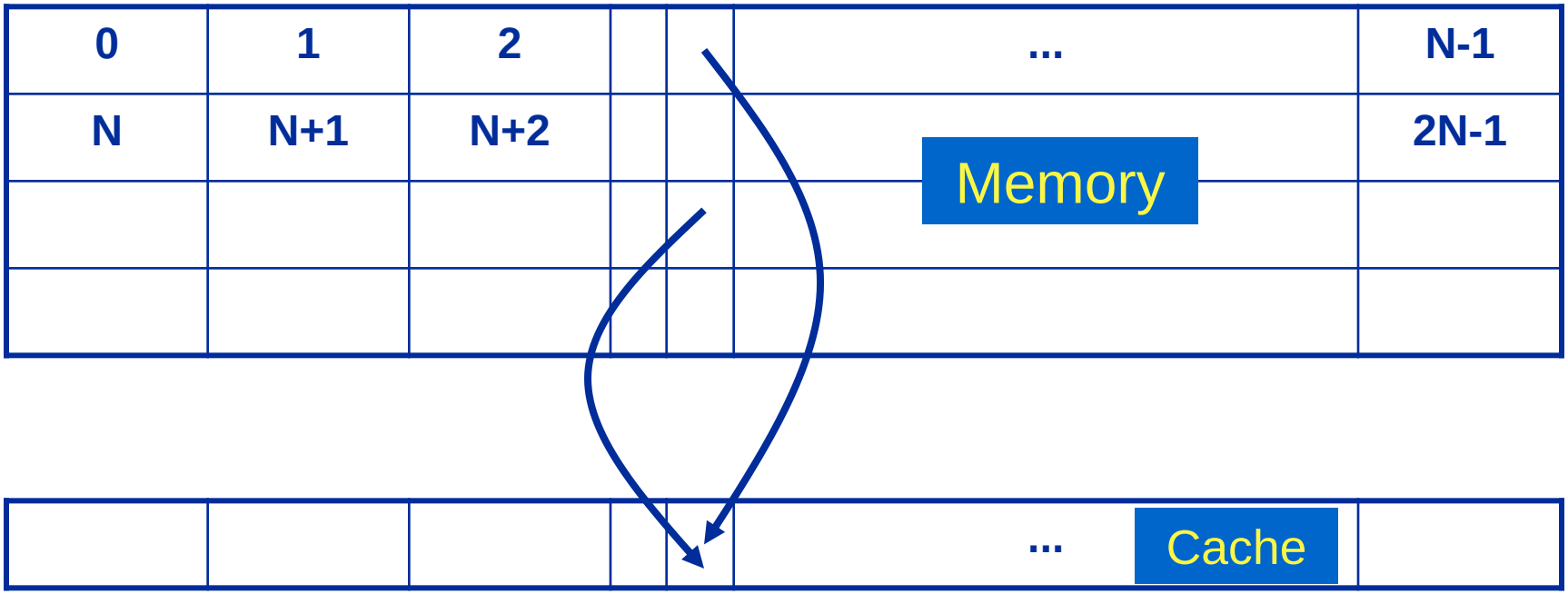
- If cache size is 1 MB choose "lowest" 20 bits of memory address as cache address



Memory address 32 Bit:
011100100000 11110100111110 001111

- Remember: Each data transfer is on CL basis, e.g. 64 B → lowest 6 Bits address element within CL, which is mapped to a specific set in cache
- Bit 6-19: This identifies the location ("set") to which the CL is mapped in our 1 MB direct mapped cache
- Bit 20-31: This is information is stored in "Cache Tag"

Memory Hierarchies: Cache Mapping – Directly Mapped



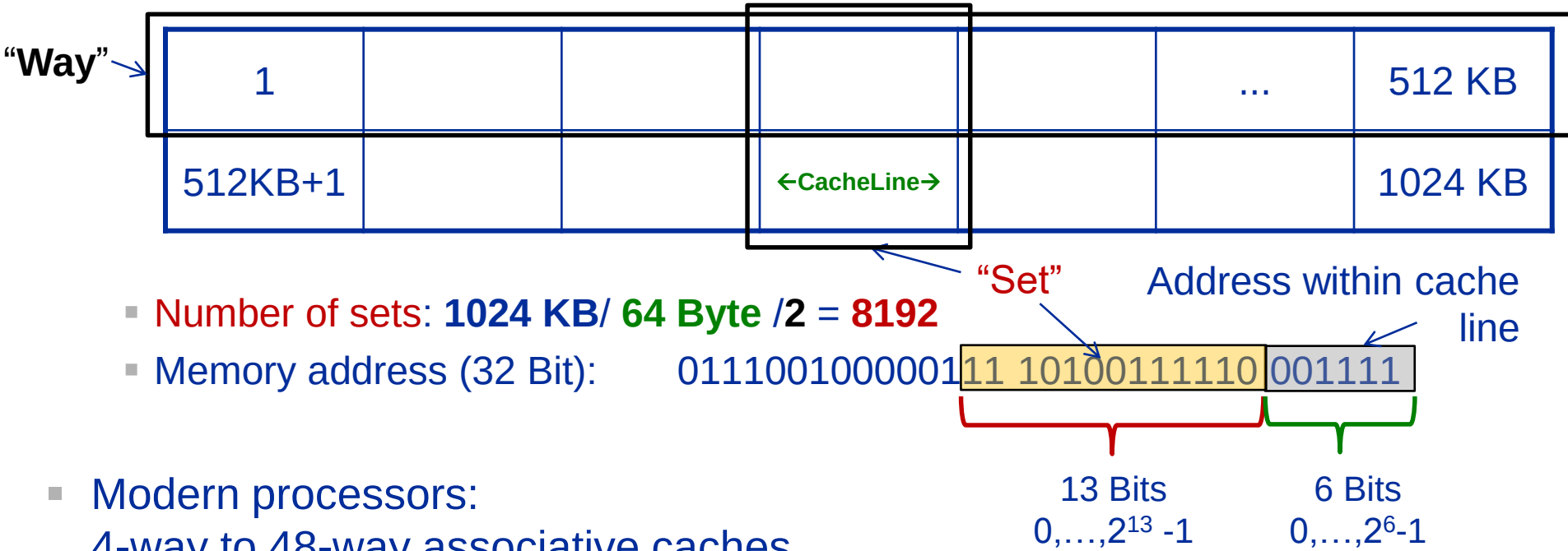
Example: Directly mapped cache. Each memory location/cache line can be mapped to one cache location/set only.

E.g. Size of main memory= 64 GByte; Cache Size= 64 KB; CL=64 B
→ 10^6 are Cache Lines mapped to the same cache location/set

Memory Hierarchies: Cache Mapping – direct mapping

- **Advantages:**
  - Easy to implement
  - Fast / low latency
  - No penalty for stride 1 accesses (streaming)
- **Disadvantages:**
  - No flexibility → High chances of early evicts
  - Large stride accesses can substantially reduce the available cache size
- **Rarely seen in real world processors**
- **Provide more flexibility: m-way set associative caches**

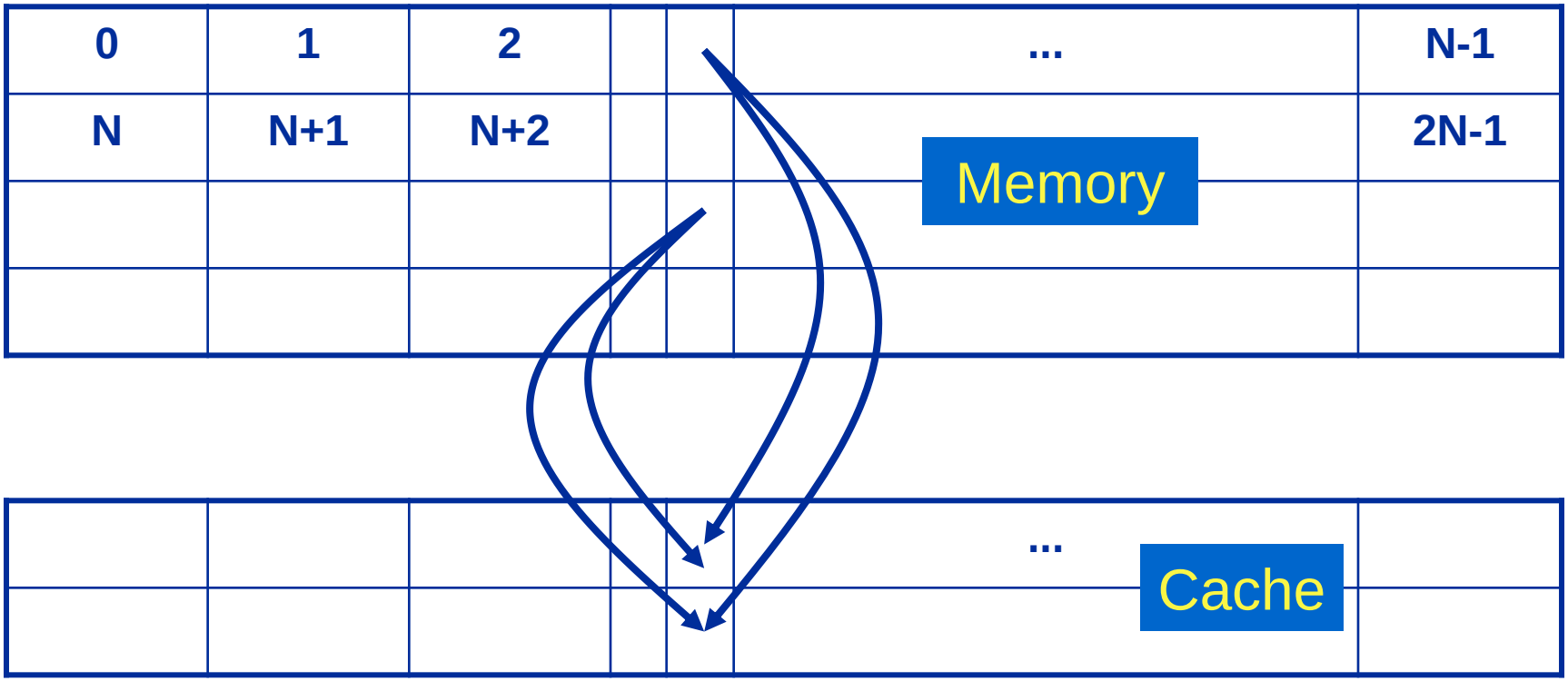
- **Set-associative cache:**
  - m-way associative cache of size m x n: each memory location i can be mapped to the m cache locations ("ways") j*n+mod(i,n), j=0..m-1
  - E.g.: 2-way set associative cache of size 1 Mbytes = **1024 KB** (Addresses: 0,..,20^{20}-1 → 20 Bit):



  - Number of sets: **1024 KB**/ **64 Byte** /**2** = **8192**
  - Memory address (32 Bit):

- Modern processors:
  4-way to 48-way associative caches
- Which way is used within set: Age of data in the ways

Memory hierarchies: Cache Mapping – Associative Caches



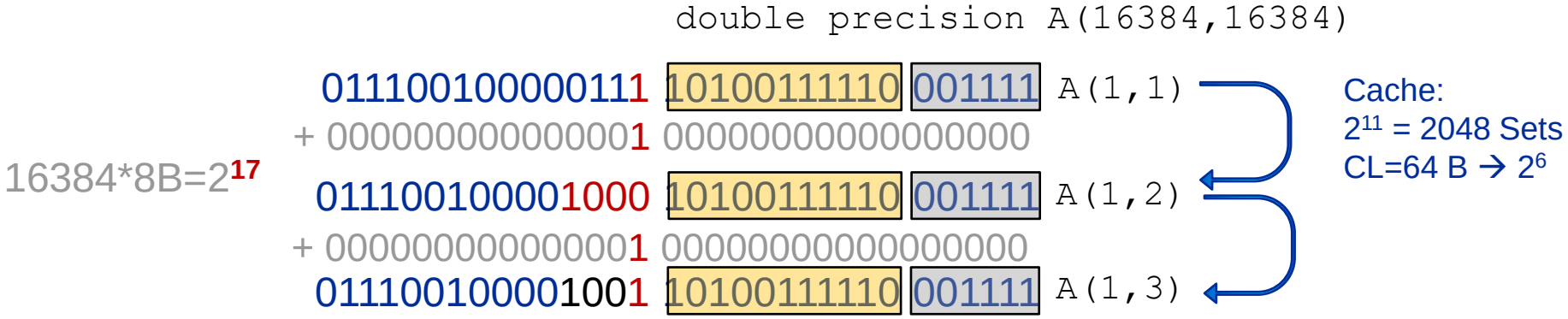
Example: 2-way associative cache. Each memory location can be mapped to two cache locations ("ways") within the same set:

Size of main memory= 64 GByte; Cache Size= 64 KB; CL = 64 B
*→ 2*10^6 Cache Lines are mapped to two cache locations / ways within a set*

Memory hierarchies: Pitfalls & Problems

- If many memory locations are used that are mapped to same set, cache reuse can be very limited even with m-way associative caches

- Warning: Using powers of 2 in the leading array dimensions of multi-dimensional arrays should be avoided! (Cache Thrashing)

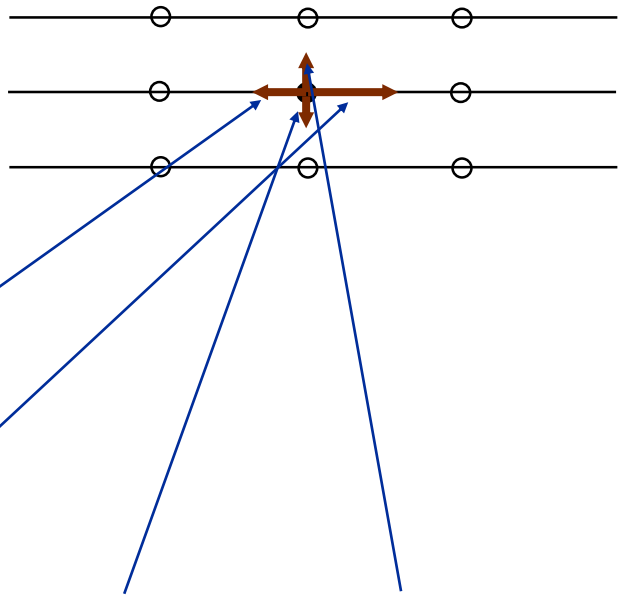


- If cache / m-ways are full and new data comes in from main memory, data in cache (full cache line) must be invalidated or written back

Ensure spatial and temporal data locality for data access!

Memory hierarchies: Cache thrashing - Example

Example: 2D – square lattice
At each lattice point the 4 velocities for each of the 4 directions are stored

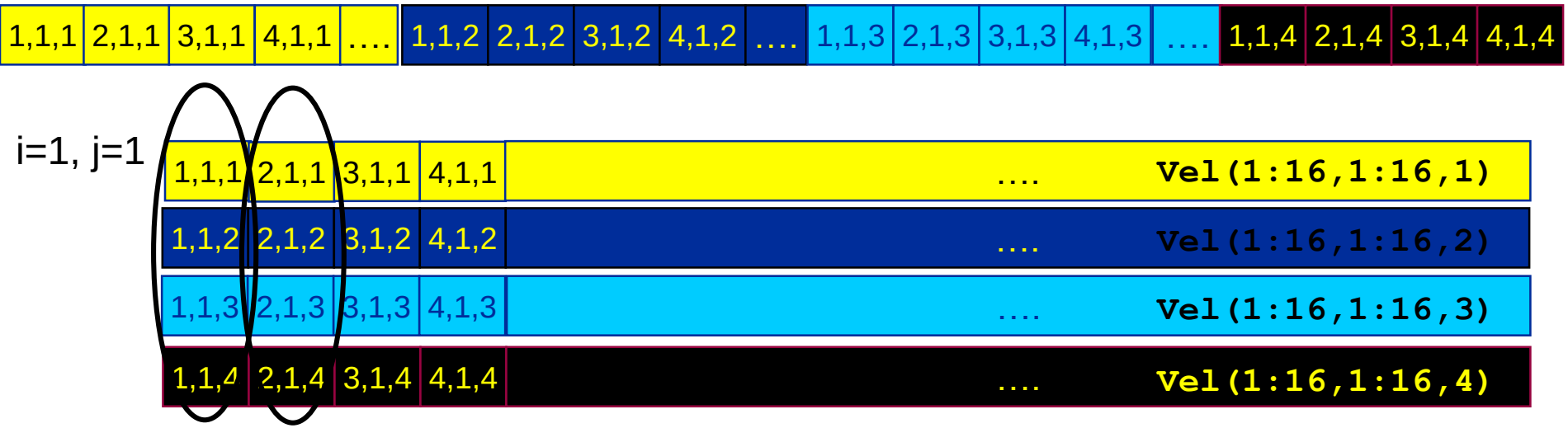


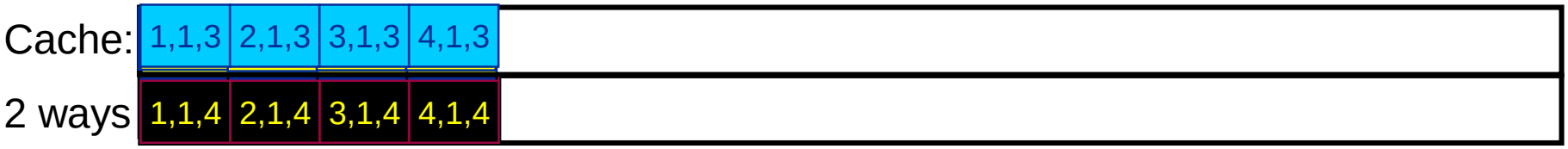
```
N=16
real*8 vel(1:N , 1:N, 4)
……
s=0.d0
do j=1,N
   do i=1,N
      s=s+vel(i,j,1)-vel(i,j,2)+vel(i,j,3)-vel(i,j,4)
   enddo
enddo
```

Memory hierarchies: Cache thrashing - Example

Memory to cache mapping for **`vel(1:16, 1:16, 4)`**

Cache: 256 byte (=32 double) / 2-way associative / Cache line size=32 byte





with 16 double each

Each cache line must be loaded 4 times from main memory to cache!

Memory hierarchies: Cache thrashing - Example

Memory to cache mapping for `vel(1:18, 1:18, 4)`

Cache: 256 byte (=32 doubles) / 2-way associative / Cache line size=32 byte

with 16 doubles each

Each cache line needs only be loaded **once** from memory to cache!

Memory hierarchies: Cache management details – states

- Handling of cached data to be replaced depends on its “state”
- Basic “states” of cache line in cache:
  - NOT MODIFIED: Valid copy in lower cache levels/main memory → Cache line may be overwritten with new data or copied back to lower cache levels (see later: inclusive vs exclusive)
  - MODIFIED: Cache line has been modified and other copies in caches/main memory are invalid → Cache line needs to be evicted to lower cache levels/main memory before it can be overwritten
- Actual states are more diverse on modern multicore processors
- State of the cache line is stored cache line tag

Memory hierarchies: Cache management details – ST miss

Assume only one cache level:

- LOAD miss: If data item to be loaded to a register is not available in cache, the corresponding cache line is loaded from main memory

- **STORE miss**: Data item to be modified (e.g. **a[2]=0.0**) is not in cache?
  - Cache line is the minimum data transfer unit between main memory and cache (e.g. a[0:7]).
    - Load cache line from main memory to cache (**"WRITE ALLOCATE"**)
    - Modify data item in cache
    - Later evict/write back modified cache line to main memory

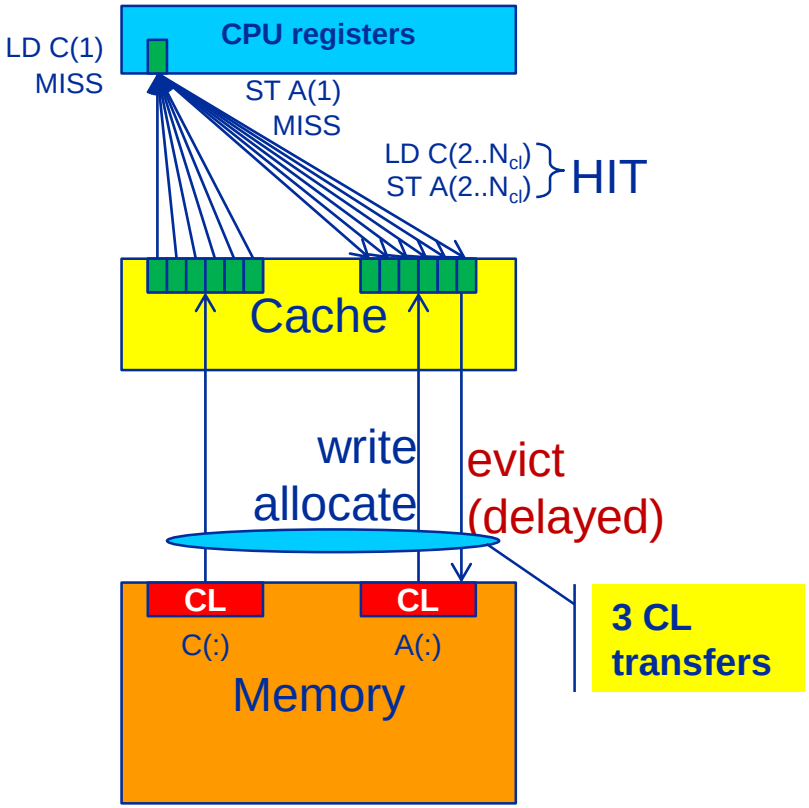
```
do i=1,n
  do j=1,n
    a(j,i)= 0.0
  enddo
enddo
```

→ n^2 words are loaded from main memory to cache (WRITE ALLOCATE) and n^2 words are evicted/written back to main memory!

→ Overall data transfer volume may increases up to 2x! (NT stores: no increase)

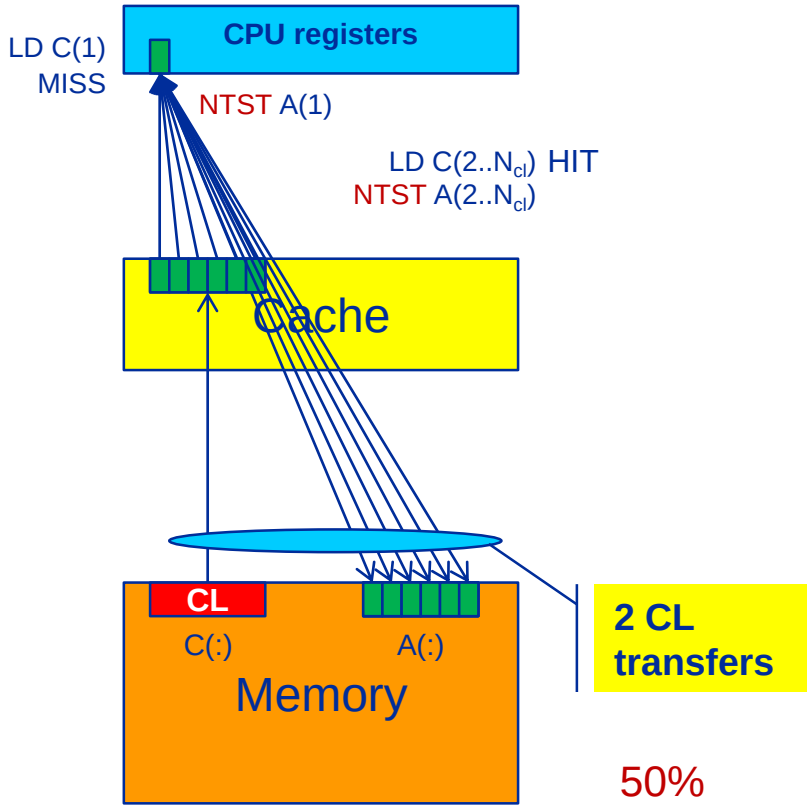
Memory hierarchies: Cache management details

- How does data travel from memory to the CPU and back?
- Example: Array copy `A(:)=C(:)`



Standard stores (**W**RITE **A**LLOCATE)

Special store instruction to avoid WA!



Nontemporal (NT) stores → 50% performance boost for COPY

Memory management: Caches management details

Inclusive:

- Cache line copy in all levels
- Reduced effective size in outer cache levels
- Cheap eviction for unmodified cache lines
- Higher latency: cache lines have to load through hierarchy

→ All caches in Intel processors up to Broadwell

Exclusive / Non-inclusive:

- Only one cache line copy in cache hierarchy
- Full aggregate effective cache size
- Eviction is expensive (copy back)
- Lower latency: Data can be directly loaded in L1/L2 cache

→ AMD processors: L3 cache
Intel Skylake L3 cache

"Write back": A modified cache line is evicted to the next (lower) cache/memory level before it is overwritten by new data

"Write through": When a cache line is modified then the cache line copy in the next (lower) cache/memory level is updated as well

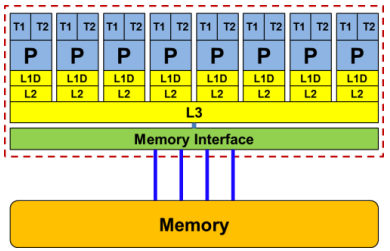
Memory Hierarchies: Typical cache configuration

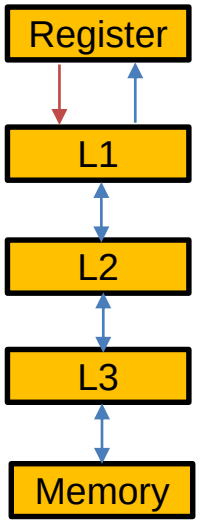
		Intel Xeon E5-2680 Sandy Bridge
# FP registers		**16**
# GP registers		**16**
L1 D	Size	**32 KB**
	Associativity	8-way
	local per core	
L2	Size	**256 KB**
	Associativity	8-way
	local per core	
L3	Size	#cores* **2 MB**
	Associativity	#cores* 16-way
	shared across all cores	

SIMD registers

Same for Intel architectures until Broadwell

Depends on core count, CPU variant, CoD mode





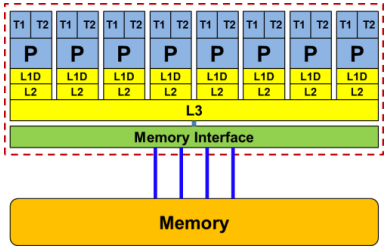
Memory Hierarchies: Typical cache configuration

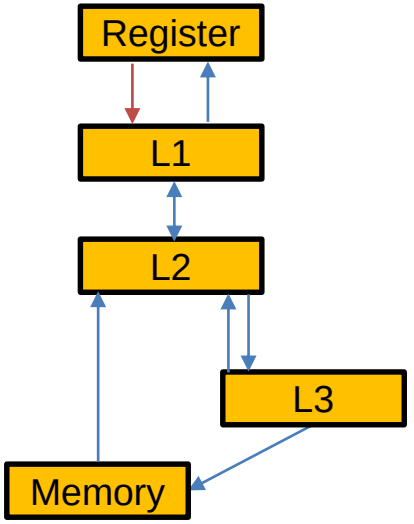
		Intel Xeon Skylake SP
# FP registers		**32** (with AVX-512)
# GP registers		**16**
L1 D	Size	**32 KB**
	Associativity	8-way
	local per core	
L2	Size	**1 MB**
	Associativity	16-way
	local per core	
L3	Size	#cores* 1.375 MB
	Associativity	#cores* 11-way
	shared across all cores	

SIMD registers

Inclusive caches

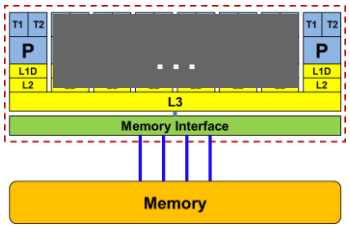
Exclusive L3





Intel Xeon E5 multicore processors

Microarchitecture	SandyBridge-EP	IvyBridge-EP	Haswell-EP
Shorthand	SNB	IVB	HSW
Xeon Model	E5-2680	E5-2690 v2	E5-2695 v3
Year	03/2012	09/2013	09/2014
Clock speed (fixed)	2.7 GHz	2.2 GHz	2.3 GHz
Cores/Threads	8/16	10/20	14/28
Load/Store throughput per cycle			
AVX(2)	1 LD & 1/2 ST	1 LD & 1/2 ST	2 LD & 1 ST
SSE/scalar	2 LD ‖ 1 LD & 1 ST	2 LD ‖ 1 LD & 1 ST	2 LD & 1 ST
L1 port width	2×16+1×16 B	2×16+1×16 B	2×32+1×32 B
ADD throughput	1 / cy	1 / cy	1 / cy
MUL throughput	1 / cy	1 / cy	2 / cy
FMA throughput	n/a	n/a	2 / cy
L2-L1 data bus	32 B	32 B	64 B
L3-L2 data bus	32 B	32 B	32 B
LLC size	20 MiB	25 MiB	35 MiB
Main memory	4×DDR3-1600	4×DDR3-1866	4×DDR4-2133
Peak memory BW	51.2 GB/s	51.2 GB/s	68.3 GB/s
Load-only BW	43.6 GB/s (85%)	46.1 GB/s (90%)	60.6 GB/s (89%)
T_{L3Mem} per CL	3.96 cy	3.05 cy	2.43 cy



FP instructions throughput per core

Max. data transfer per cycle between caches

Peak main memory bandwidth

Max. attainable bandwidth

Matrix transpose & cache thrashing: A real-world example

Memory hierarchies: Cache traffic/thrashing – Example

- **Matrix transpose**

- **Minimum** data traffic:
  - Load from main memory
    $2 * N^2 * 4$ Byte
  - Store to main memory:
    $N^2 * 4$ Byte
  - For N= 8192 we expect
    - **Load: 536 MB**
    - Store: 268 MB

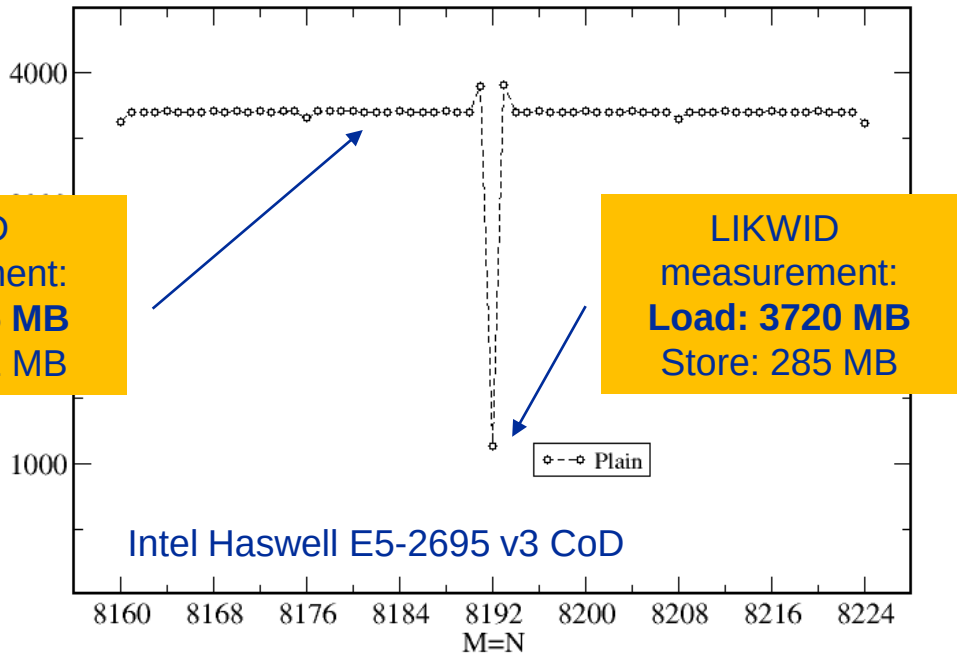
Matrix B and A (write-allocate!!!)

Single precision

Matrix A

```
single precision A(N,N) , B(N,N)
...
do i = 1 , N
   do j = 1 , N
      A(j,i) = B(i,j)
   enndo
enddo
```

Non-consecutive access



LIKWID measurement: **Load: 565 MB** Store: 281 MB

LIKWID measurement: **Load: 3720 MB** Store: 285 MB

Memory hierarchies: Cache traffic/thrashing

- Problem:
  - L3 cache can hold 8192 cache lines (0.5 MB) of `B` to allow for spatial locality in next outer (`i`-)iteration
  - But at `N=8192` these cache lines are mapped to the same set in L3 cache (which has associativity of 7*32=224, i.e. only 224 CL can be stored)
  - Thus in every `j`-iteration a full cache line is loaded from main memory (and only one entry is used)
- Solution: Padding
  - Chose leading dimension such that it is not power of 2, e.g.
  - `Np` is next number of `N` which is odd multiple of 16

```
single precision A(Np,N) , B(Np,N)
...
do i = 1 , N
   do j = 1 , N
      A(j,i) = B(i,j)
   enndo
enddo
```