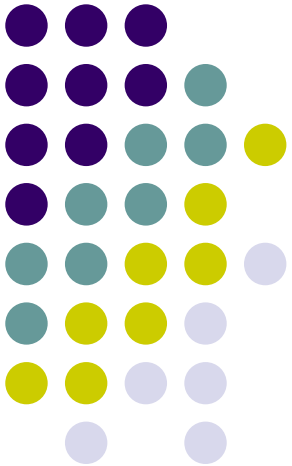
2014年度
実践的並列コンピューティング
第7回

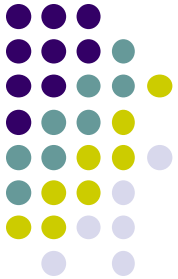
ソフトウェア性能のための
コンピュータアーキテクチャ



遠藤 敏夫

endo@is.titech.ac.jp

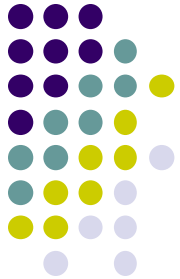
並列ソフトウェアを作るポイントは？



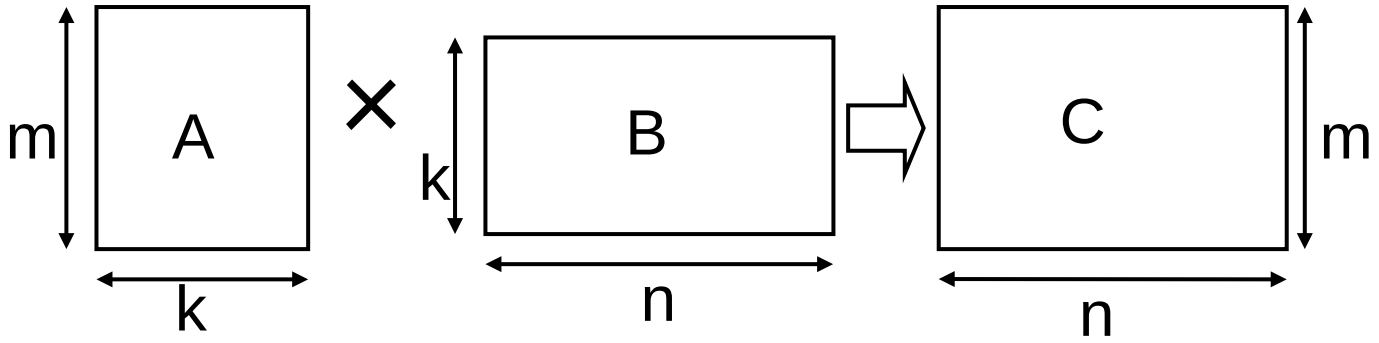
- 正しい逐次アルゴリズムか？
- 速い逐次実装か？
- 正しい並列アルゴリズムか？
  - 依存関係を壊していないか
  - Race conditionはないか
- 速い並列実装か？
  - ノード内で速いか
  - ノード間で速いか

アーキテクチャの知識が必要な場合も

Example Computation: Matrix Multiply (matmul)



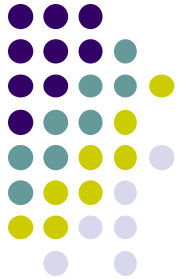
Multiplying a (m × k) matrix and a (k × n) matrix




```
for (i = 0; i < m; i++) {
  for (j = 0; j < n; j++) {
    for (l = 0; l < k; l++) {
      Ci,j += Ai,l*Bl,j;
    } } }
```

Complexity : O(mnk)

Variants in matmul Implementation



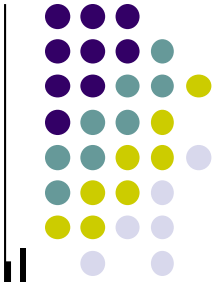
```
for (i = …) {
  for (j = …) {
    for (l = …) {
      …
```

```
for (j = …) {
  for (i = …) {
    for (l = …) {
      …
```

```
for (l = …) {
  for (i = …) {
    for (j = …) {
      …
```

- What happens if we exchange the sequence of for loop?
  - We have 6 implementations: IJL, ILJ, JIL, JLI, LIJ, LJI
  - This change does affects neither computed results nor compute complexity of O(mnk)
  - Only the sequence of operations are changed

Effects of Software Implementation

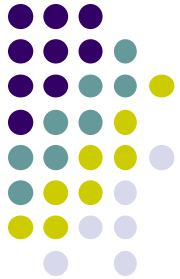


- Performance of different 6 implementations of matmul
  - Written in C language, not parallelized, gcc 4.3.4, -O2
  - Elements are “double” type, column major format
  - m=n=k=1024
  - On a single node of TSUBAME2 supercomputer

Imple	IJL	JIL	ILJ	LIJ	LJI	JLI
Time (sec)	8.51	8.52	17.5 Slow!	17.5 Slow!	1.30	1.11 Fast!

Although all implementations have same complexity, but largely different in the computation speed

単純なモデルでは
ソフトウェア性能を説明できない

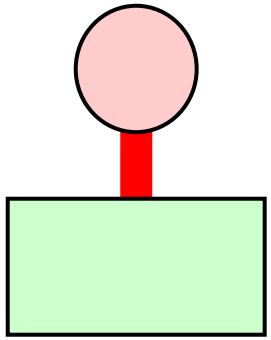


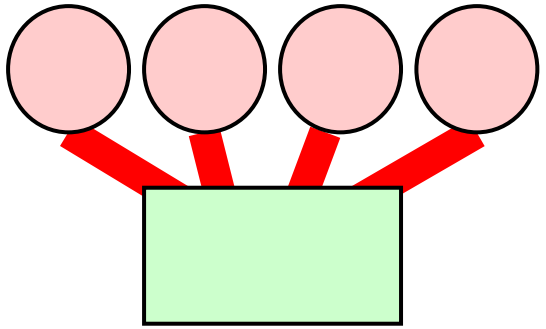
RAM model
(ramdom access machine)

PRAM model
(parallel ramdom access machine)

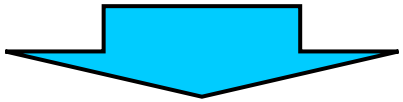
CPU cores

Memory





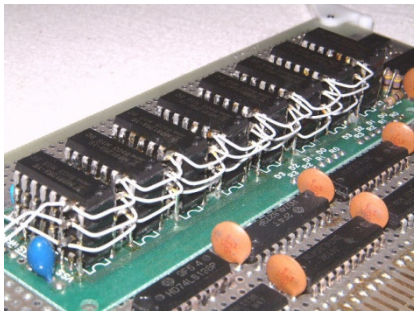
これでは6種類のmatmulの差を説明できない



コンピュータアーキテクチャ、

特にキャッシュメモリの考慮が必要

CPU and Memory: Past and Present

	Around 1980	Present
CPU	2MHz → 1clock = 500ns 	2GHz → 1clock = 0.5ns 
Memory	Access time = 2000ns(?) 	Access time = 50ns or mo 

x1000

x40(?)

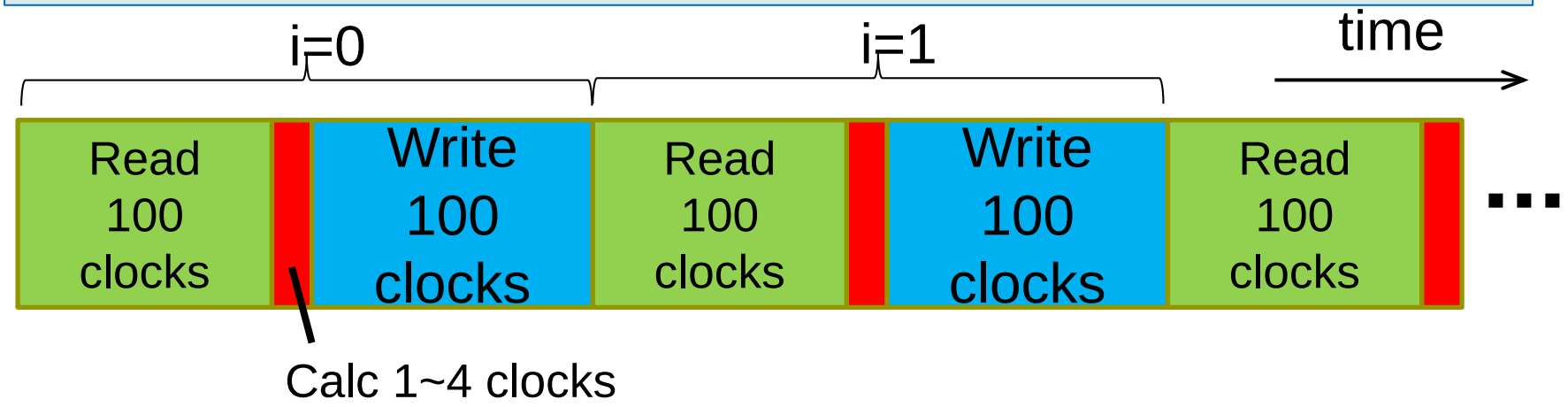
Memory Access Time

How long does a memory “read” instruction take?

Around 1980	Present
2MHz → 1clock = 500ns	2GHz → 1clock = 0.5ns
Access time = 2000ns(?)	Access time = 50ns or more
4 clocks	>100 clocks!!

What happens If Every Memory Access Takes 100 clocks?

```
for (i = 0; i < n; i++) { A[i] = A[i]*2.0; }
```

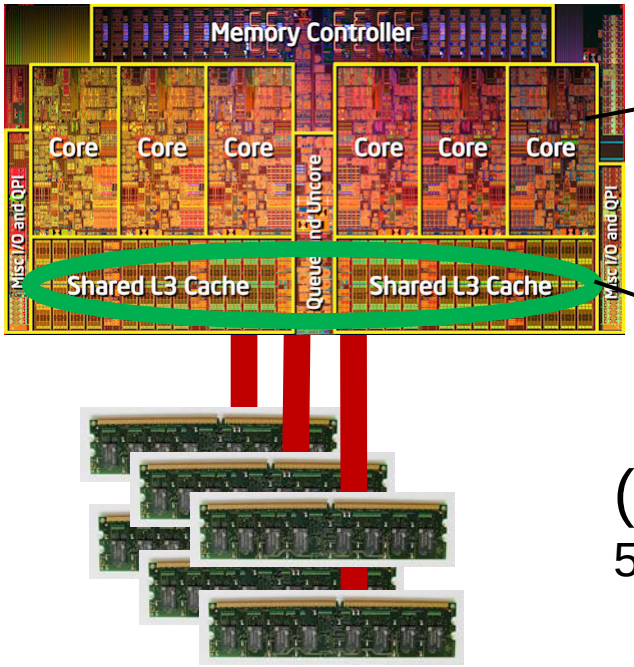


This is very insufficient!
Computation speed would be only 10MFlops

To alleviate this problem,
cache memory has been invented in 1968.
It became popular around 1985

Cache Memory

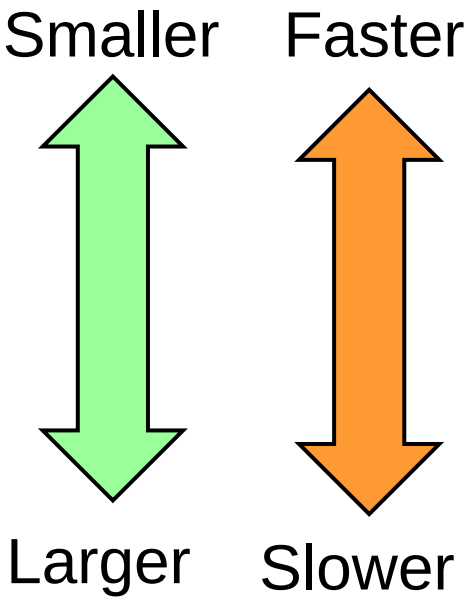
- Fast and small memory (usually) included in CPU
- Used to store data that have been recently accessed
- Used automatically --- It is OK that programmers do not know existence of cache memory



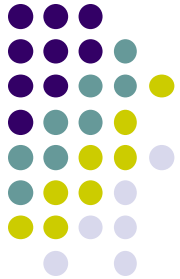
L1 cache (64KB)
L2 cache (256KB)
(included in each core)

L3 cache (12MB)

(Main) memory
54GB on TSUBAME2

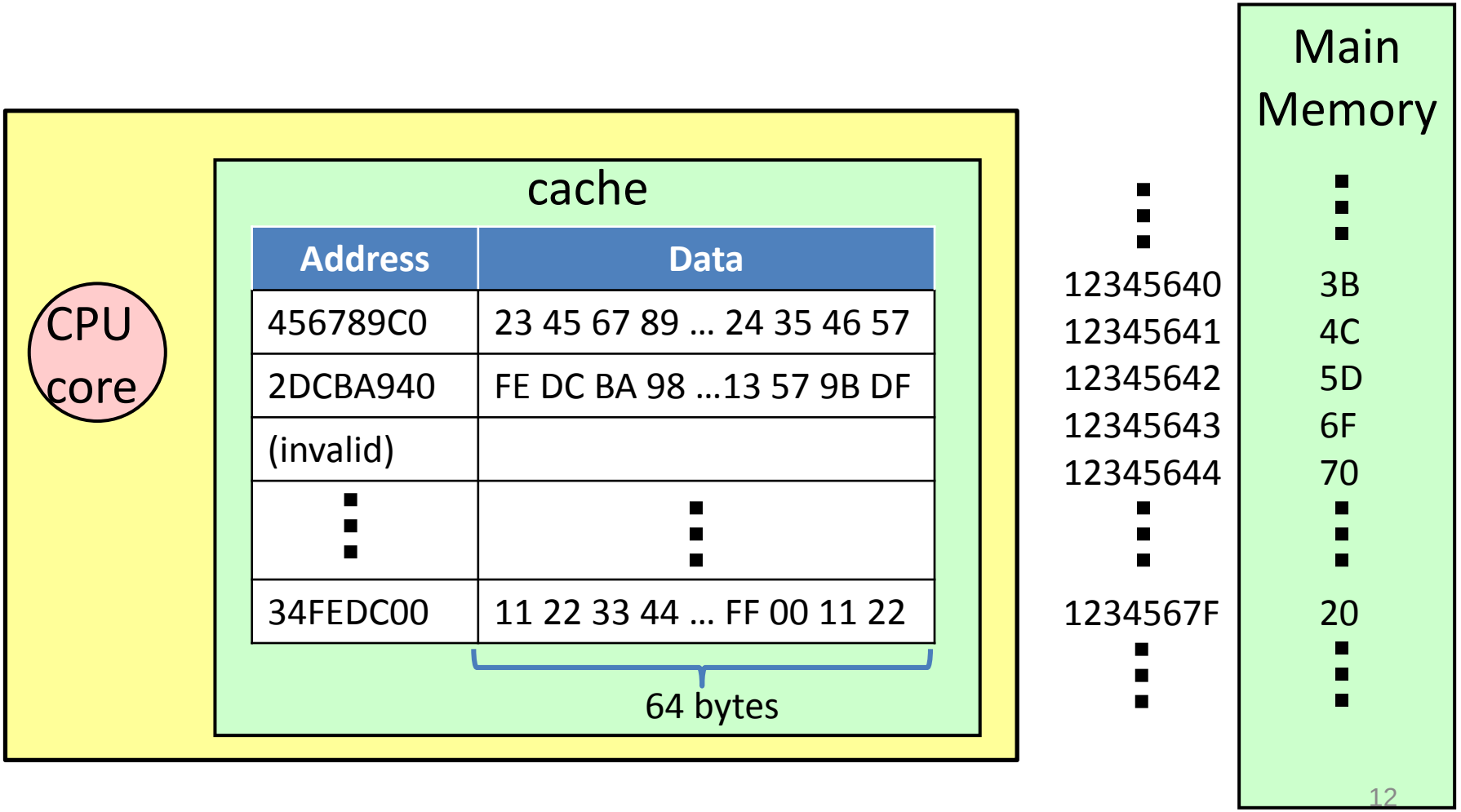


Cacheの容量とcache line



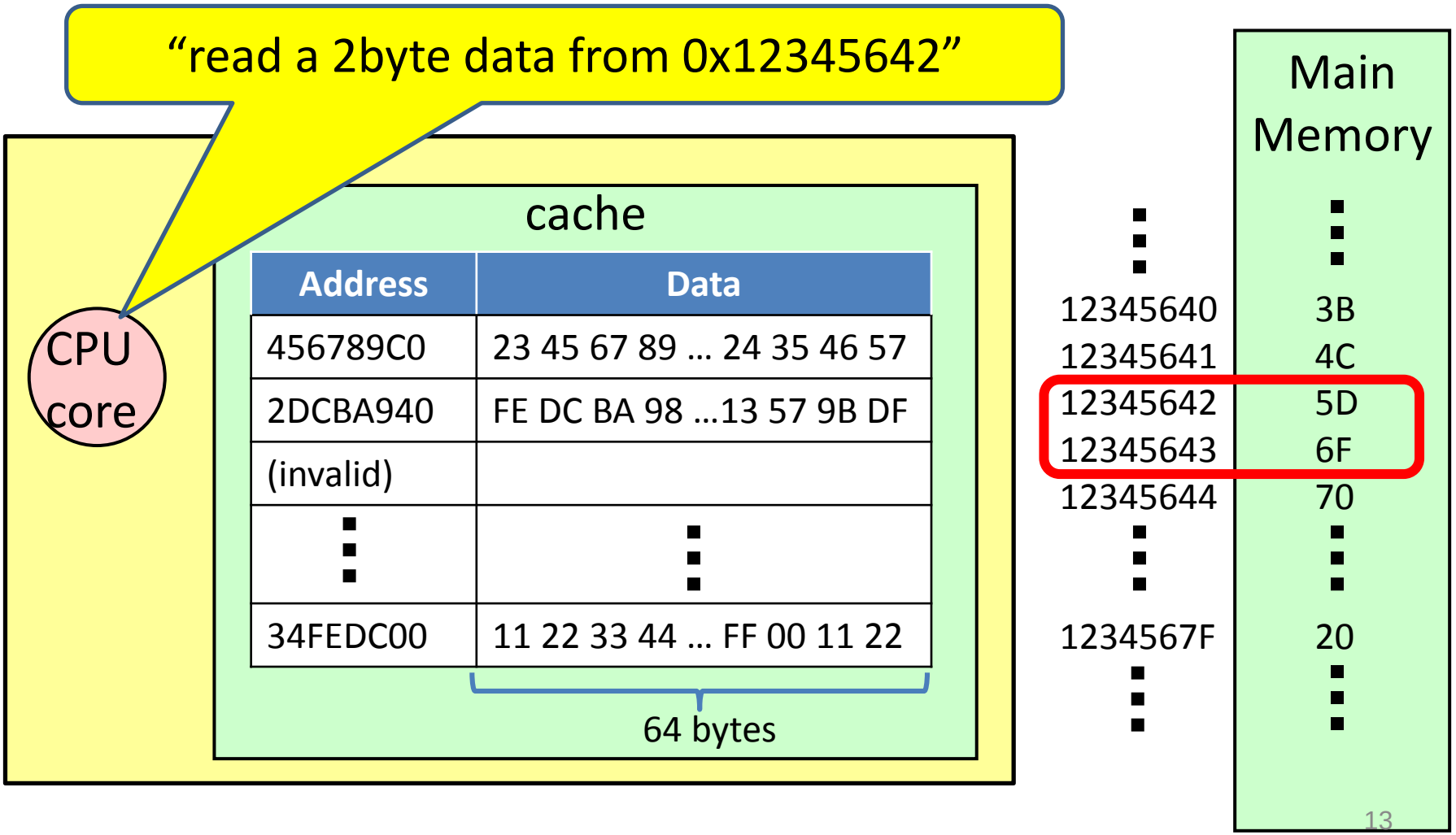
- Cacheには容量がある
  - 数KB ~ 数MB
- データ保存・移動の基本単位は、固定長のcache line
  - Intel CPUの場合はcache line sizeは64byte
  - 容量256KBのキャッシュなら、4096個のcache lineから成り立つ
  - 各cache lineは、「どのアドレスを表すか」の情報を持つ

Simplified Cache and Memory



Memory Access with Cache (1)

- When CPU core executes a read instruction



Memory Access with Cache (2)

1. Calculate the start address of cache line that includes target address
   - 0x12345642 & 0xFFFFFFC0 → 0x12345640
   - Cache line to be accessed is [0x12345640, 0x1234567F] (64=0x40bytes)
2. Search address 0x12345640 in cache

   2-1: If found, cache hit (We go to Step 5.)

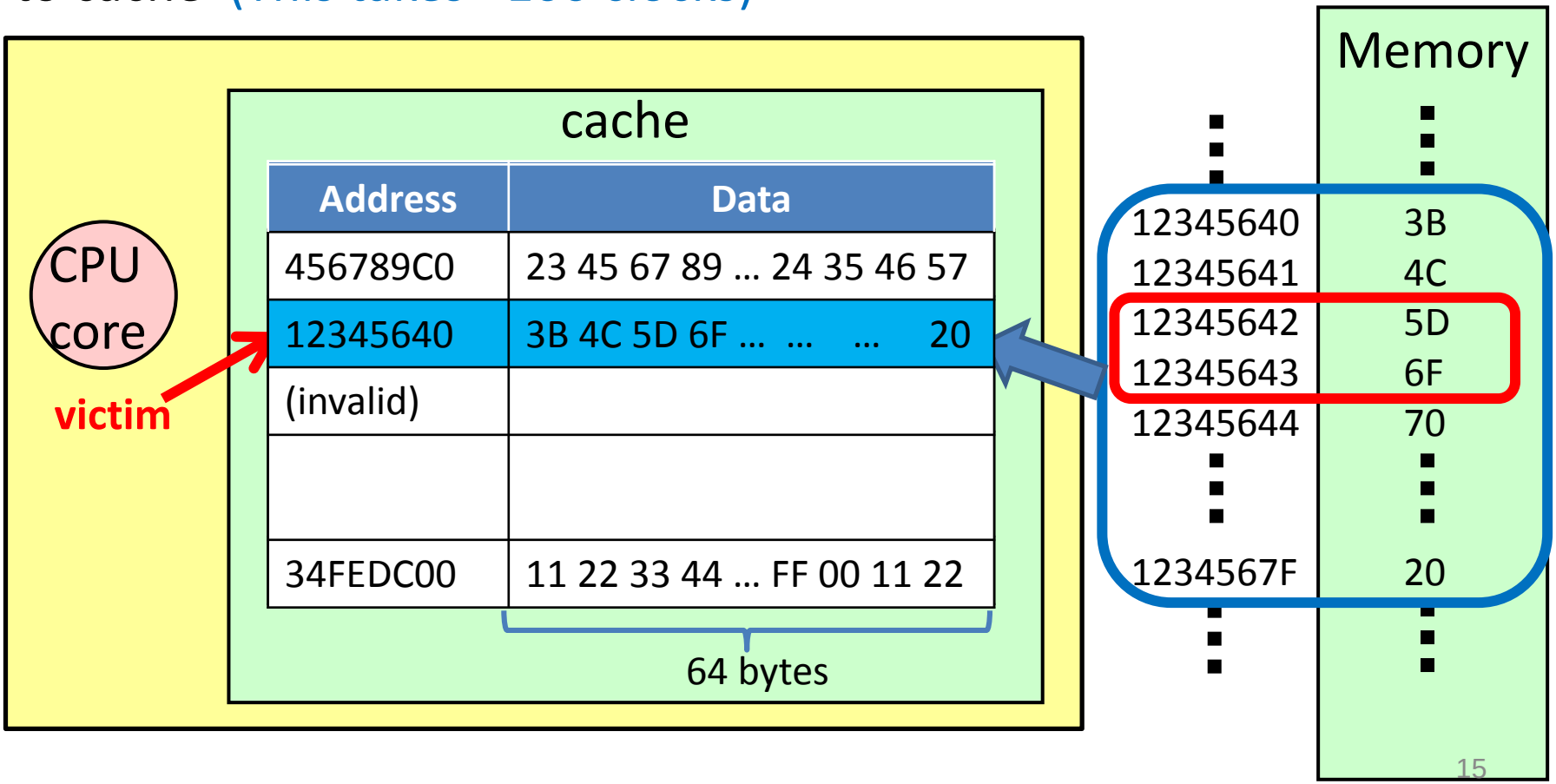
   2-2: If not found, cache miss (This is the case now)

Memory Access with Cache (3)

Cache Miss Case

3. Select a “victim” line in cache, to be deleted

4. Copy 64byte data from [0x12345640, 0x1234567F] in memory to cache (This takes >100 clocks)



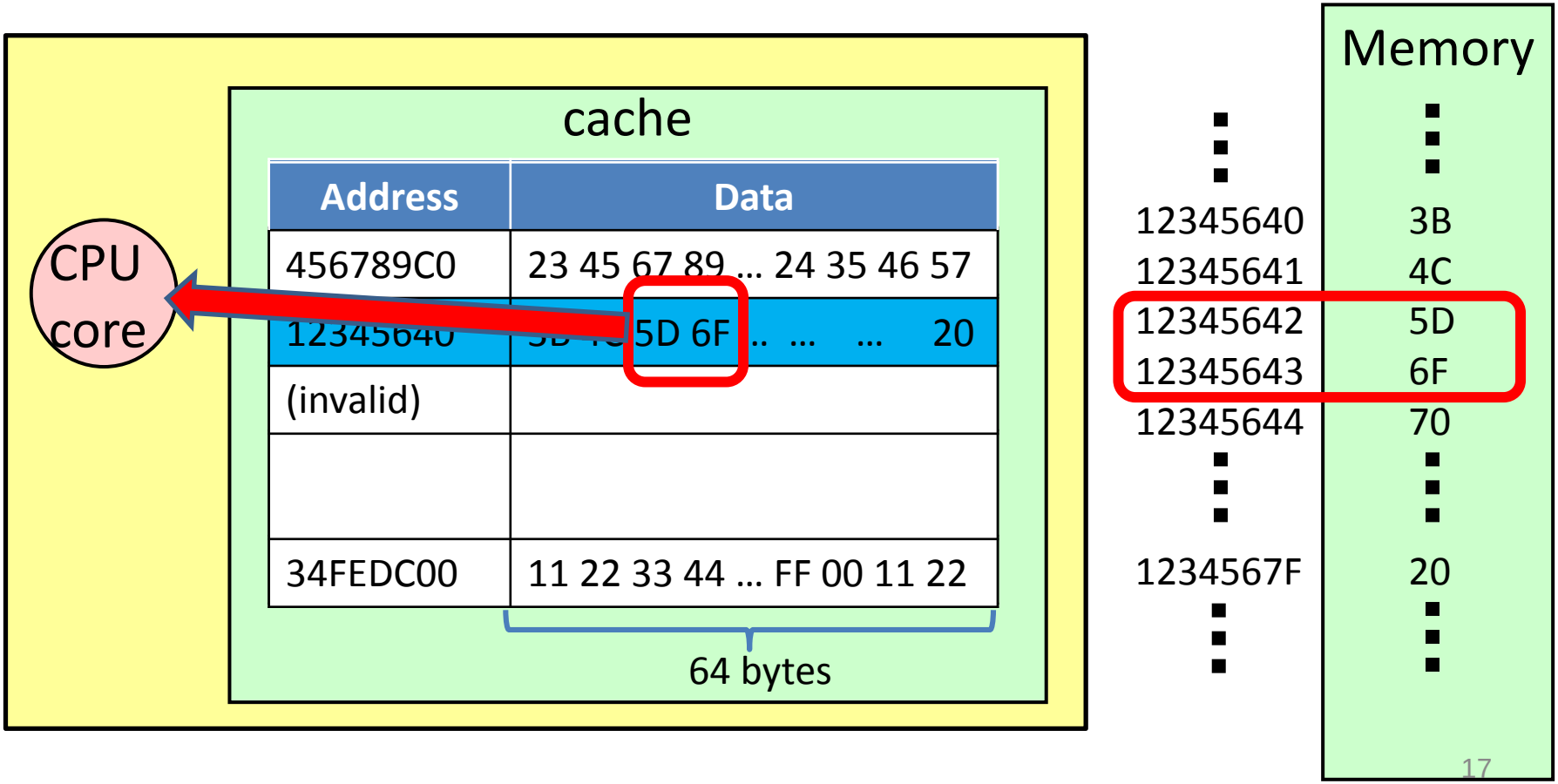
補足:どうやってvictimを選ぶか

- Cacheはhash tableに似ている
- 複数の方式あり
  - Direct mapping: 単純なビット演算で選ぶ
  - K-way set associative: Direct mappingのような表をk枚持っておく。K個のうち最近アクセスされていないものを追い出す
  - LRU (least recently used): 全lineの中から最も最近アクセスされていないものを選ぶ

    → 複雑すぎて現実のプロセッサでは使われていない

Memory Access with Cache

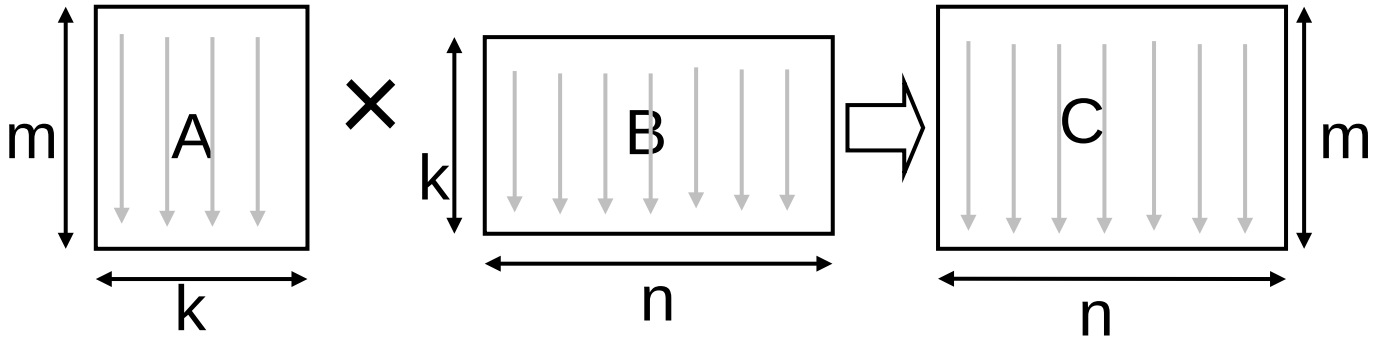
5. Deliver the desired data to CPU core



Cacheの存在により起こること

- メモリアクセスにかかる時間は一定でない
  - 単にb = A[i]と書いてあっても
    - Cache hit時なら数clock
    - Cache miss時なら数百clock
    - マルチコアによるアクセス衝突があるともっと
- Cache lineの存在により、連続アドレスを連続してアクセスするのが速い
  - 6つのmatmulの性能に差が出た理由

Matmulでのアクセス順序

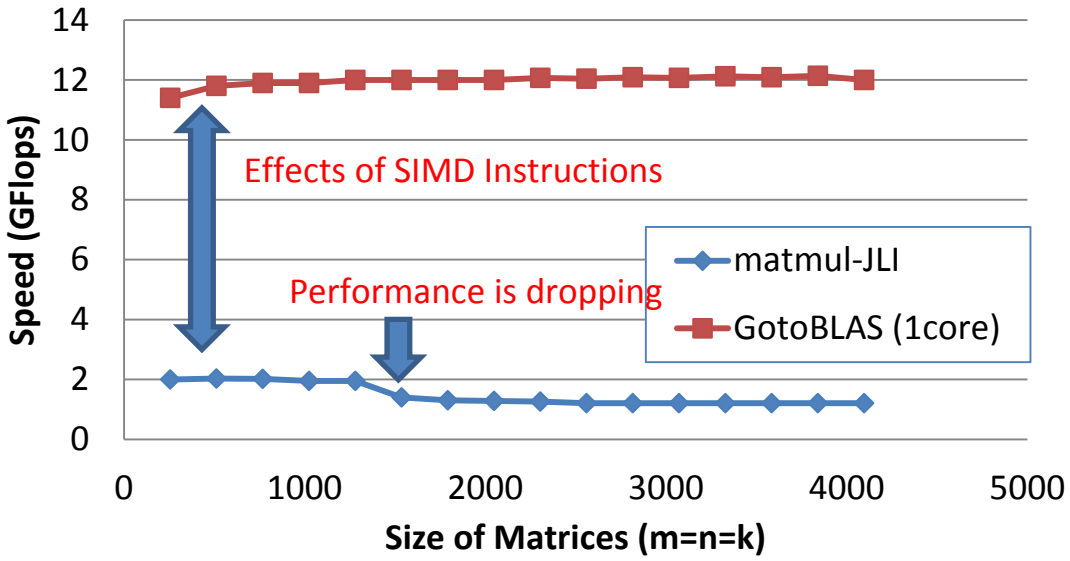



```
for (???) {
  for (???) {
    for (???) {
      Ci,j += Ai,l*Bl,j;
```

- 最内ループ内のアクセスの局所性に注目
  - 最内がl のとき → A不連続、B連続、C一定
  - 最内がjのとき → A一定、B不連続、C不連続
  - 最内がiのとき → A連続、B一定、C連続 → 最も速い

Performance of Optimized Library

- Goto BLAS, Intel MKL, AMD ACML…

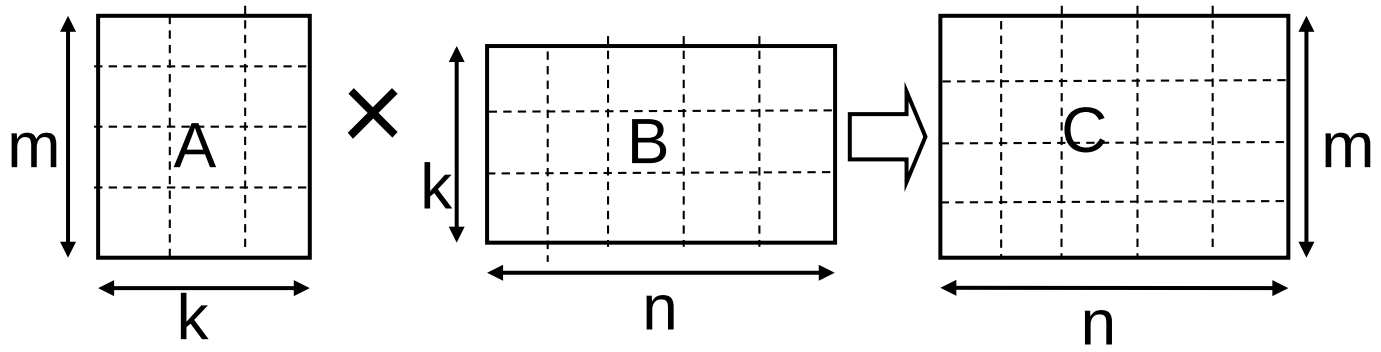


- (Naïve) Simple matmul suffers from more “cache-misses” when problem gets larger
- Optimized GotoBLAS is not only fast, but stable toward the change of problem size

Optimizations in GotoBLAS

- Effectively use multi-core
- Effectively use SIMD instructions
- Effectively use memory system

  Cache-blocking:

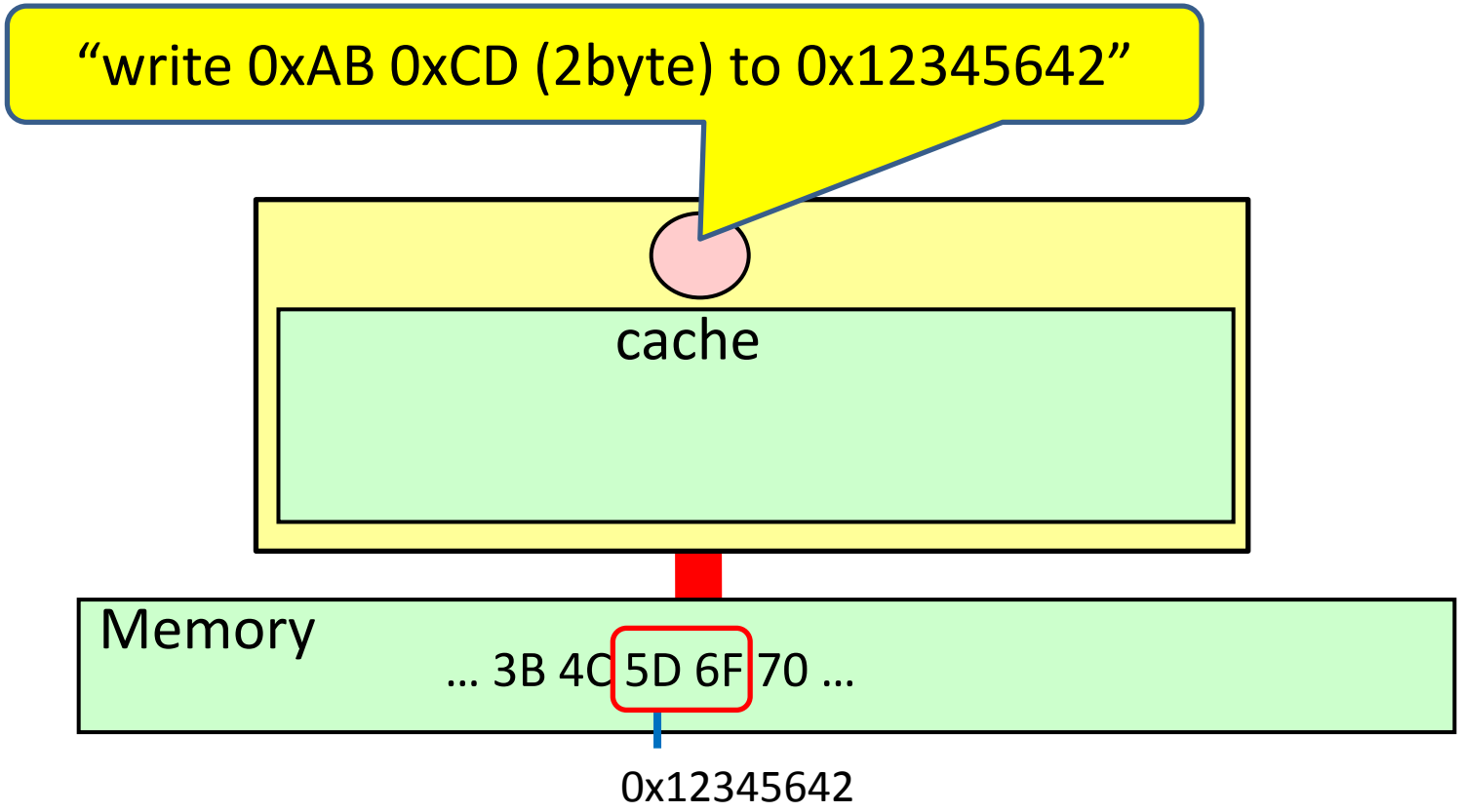


  - Matrices are broken into “blocks”, each of which are smaller than cache size
  - Sometimes data replacement occurs
  - Also optimized to reduce TLB misses

  Cf: K. Goto, R. Geijn: Anatomy of high-performance Matrix Multiplication, ACM TOMS 2008

Write (1)

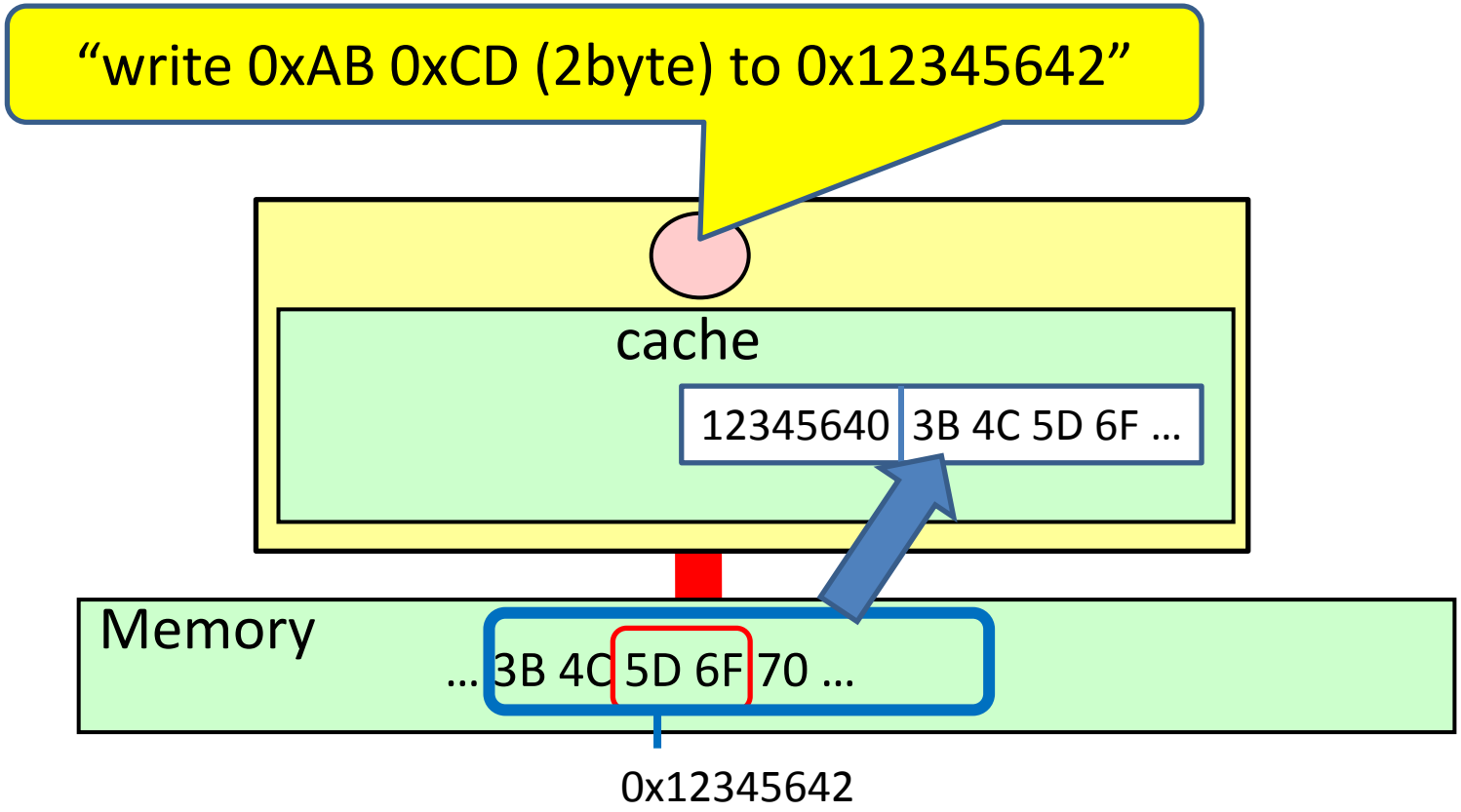
- “Write” is different from “Read” since it changes data



Write (2)

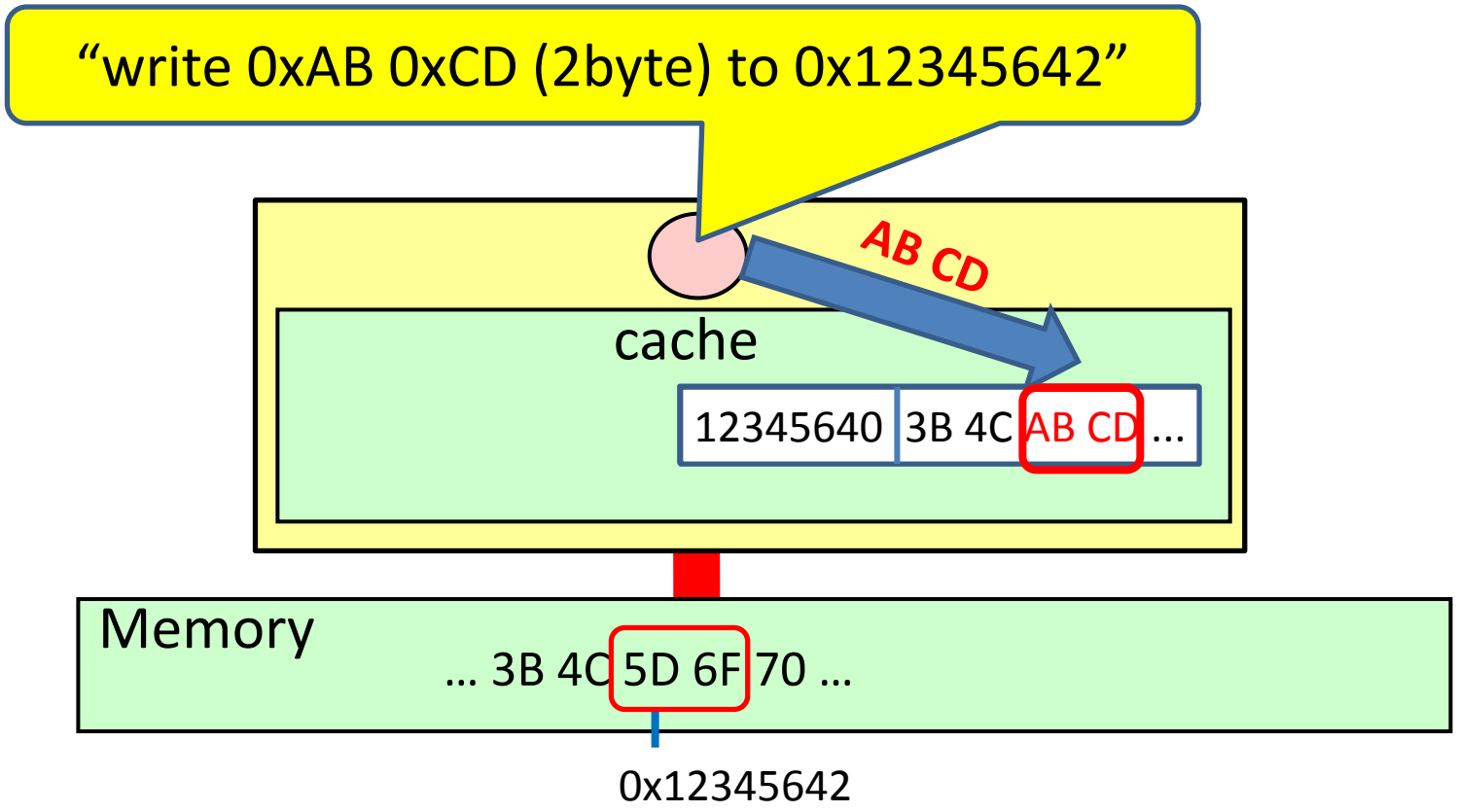
Step 1. 2. 3. are same as “Read”

4. Copy 64Byte data (cache line) to cache



Write (3)

5. Update data in cache



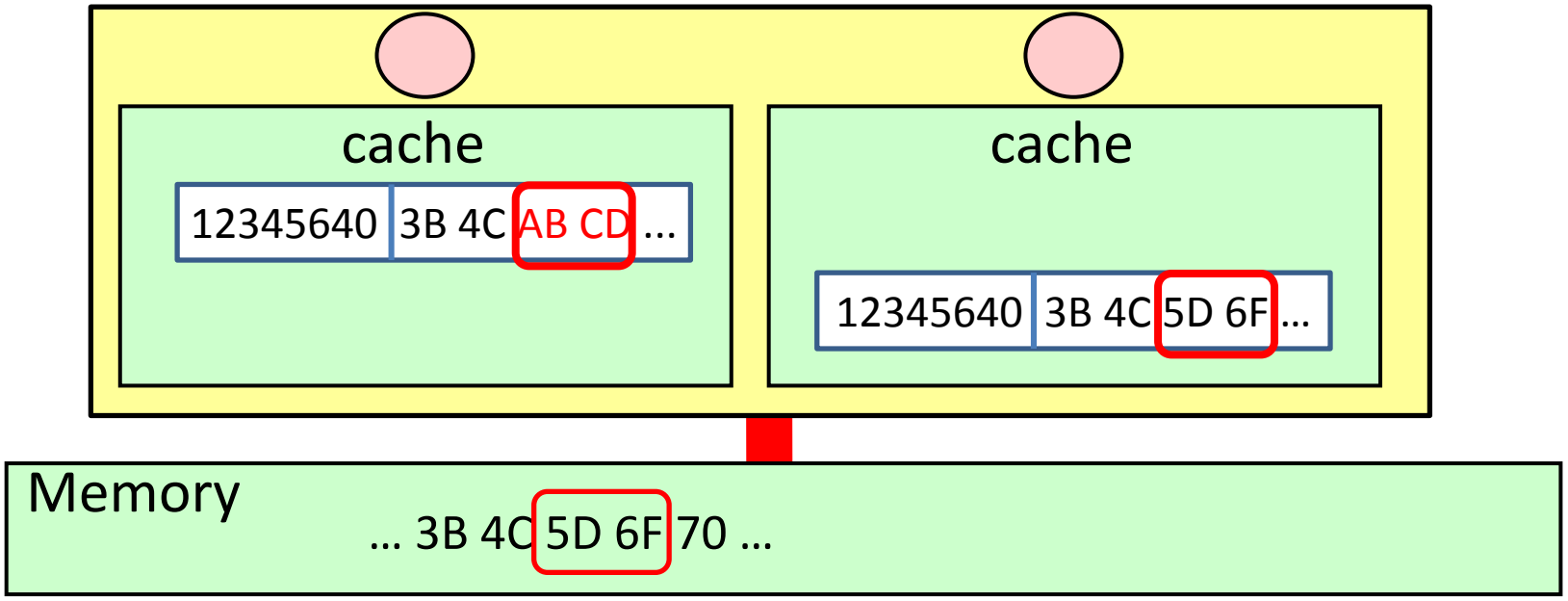
Memory has not updated yet. How should we do?
Two policies: Write through , or Write back

Write Policies

- Write through
  - Update data on memory immediately
  - Problem: Every write instruction takes >100 clocks
- Write back (more popular)
  - Data on memory is updated later
  - When the line is to be deleted as a victim, due to future cache misses
  - Hardware becomes more complex, but efficient

Due to this time lag, multi-core cache becomes complex

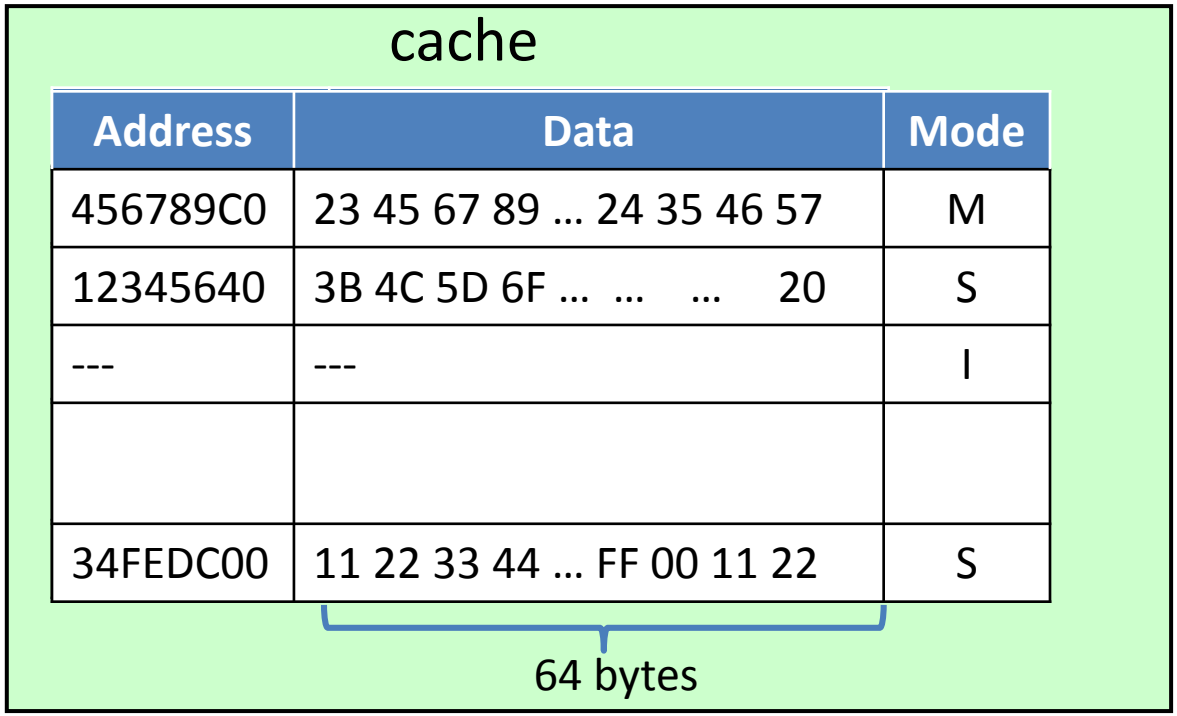
Difficulty in Parallel Cache



- How can we avoid reading "wrong" data?
  - We call this "keeping consistency"
  - There is protocol to keep consistency among caches

MSI Protocol

- MSI protocol is the simplest protocol to keep consistency
- Each cache line in cache is in one of 3 modes
  - Modified
  - Shared
  - Invalid

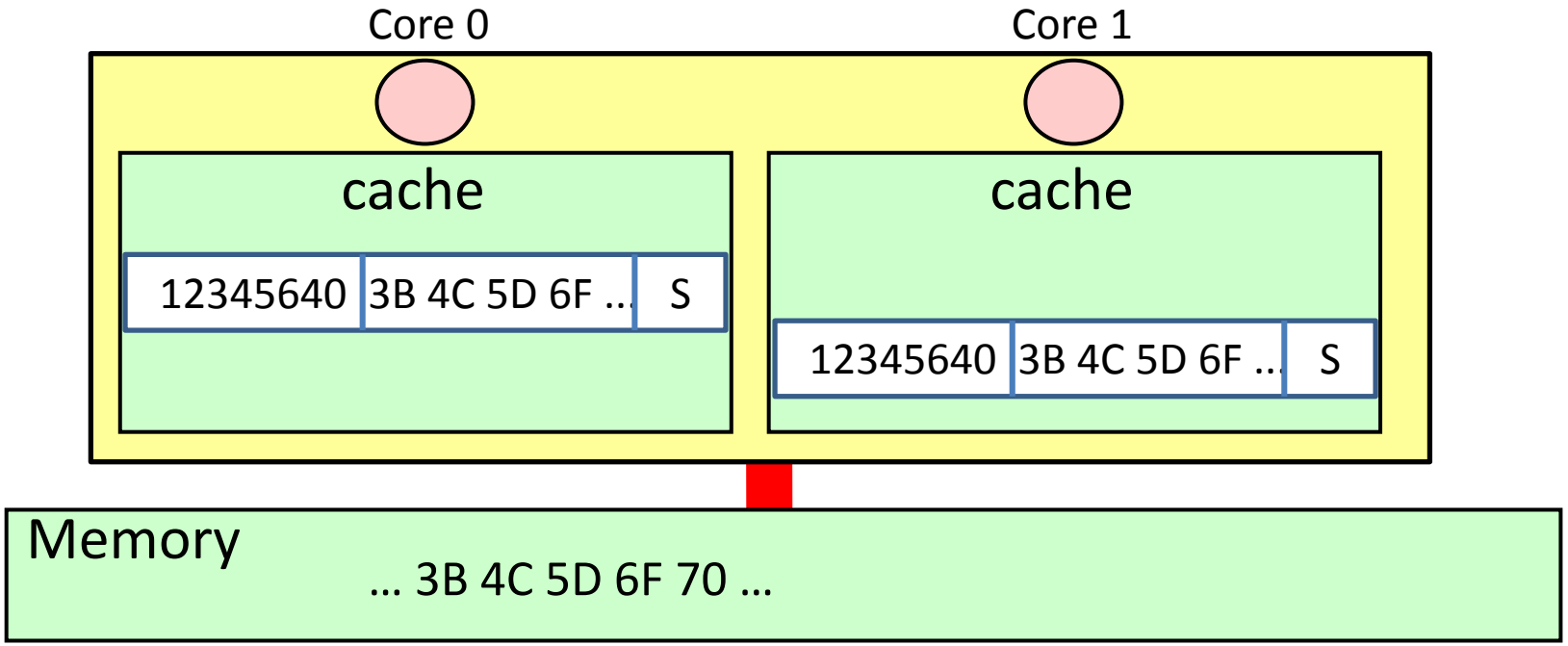


Modes in MSI Protocol

- Invalid:
  - This line is not used
- Shared:
  - This line may be shared by several caches
  - Contents of line is same as other cache or memory
- Modified:
  - Contents of line has been modified by CPU core
  - Contents of line may differ from memory
  - There must only line for the address among caches

Cache Behavior in MSI Protocol (1)

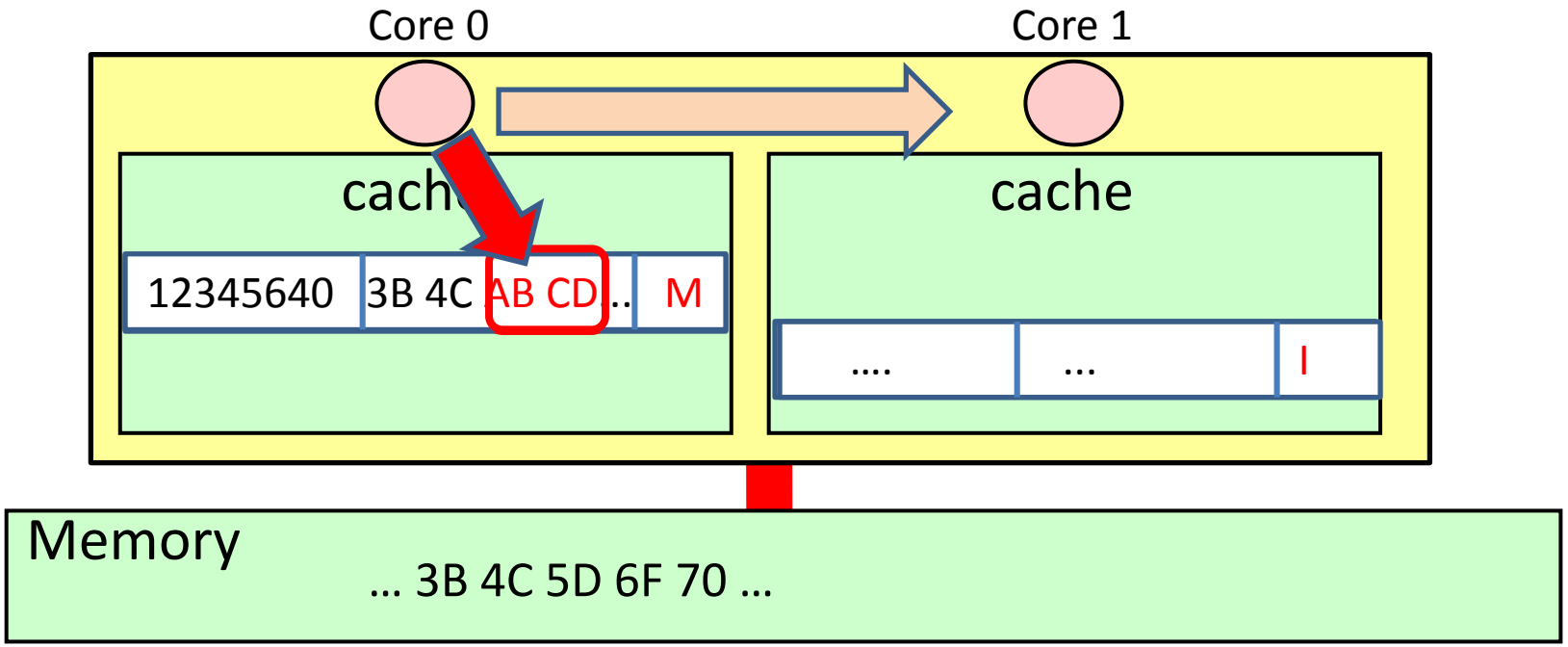
- Here line 12345640 is “Shared”



- What happens if core 0 executes “write”?

Cache Behavior in MSI Protocol (2)

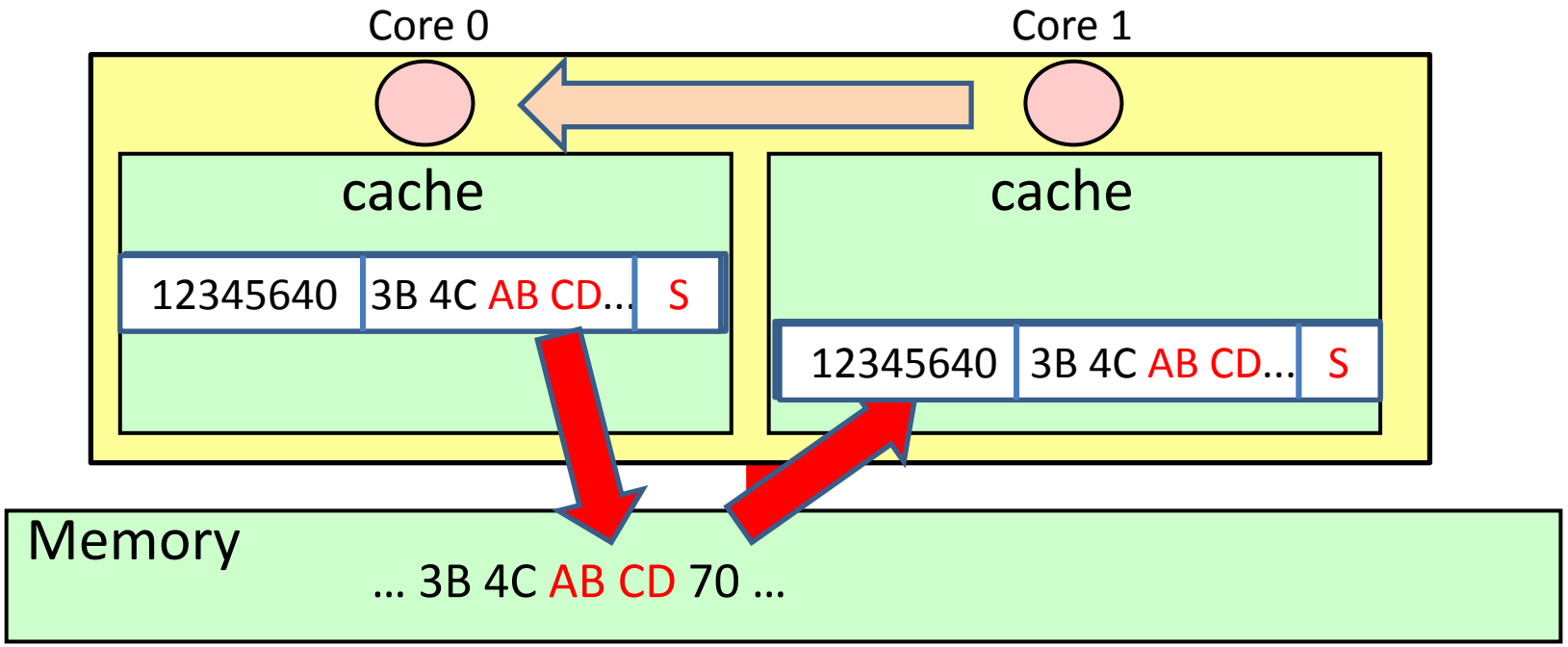
- Write to a shared line includes:
  - Make other shared line (if exist) invalid (called invalidate)
  - Make own line modified
  - Write to its own line



- What happens if core 1 executes “read”?

Cache Behavior in MSI Protocol (3)

- Read-miss (by core 1) includes:
  - If there is modified line in other cache
    - let the owner copy back the contents to memory
    - Make owner's line shared
  - Core 1 Reads from memory



- It includes 2 memory operations (>200 clocks)

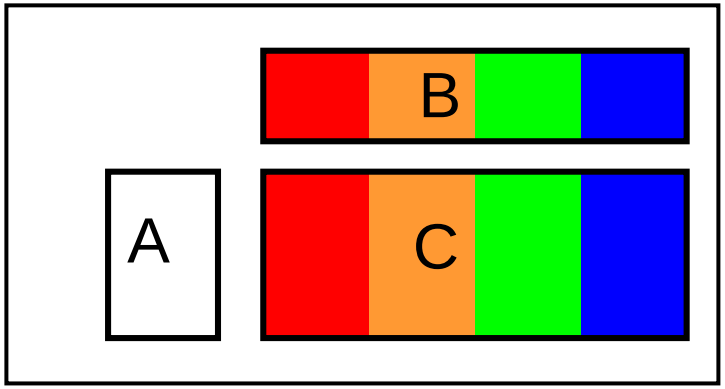
Other Protocols

Extensions of MSI

- MOSI
  - Modified, Owned, Shared, Invalid
- MESI
  - Modified, Exclusive, Shared, Invalid
- MOESI
  - Modified, Owned, Shared, Invalid
- MESIF
  - Modified, Exclusive, Shared, Invalid, Forward

並列プログラムの速度への影響

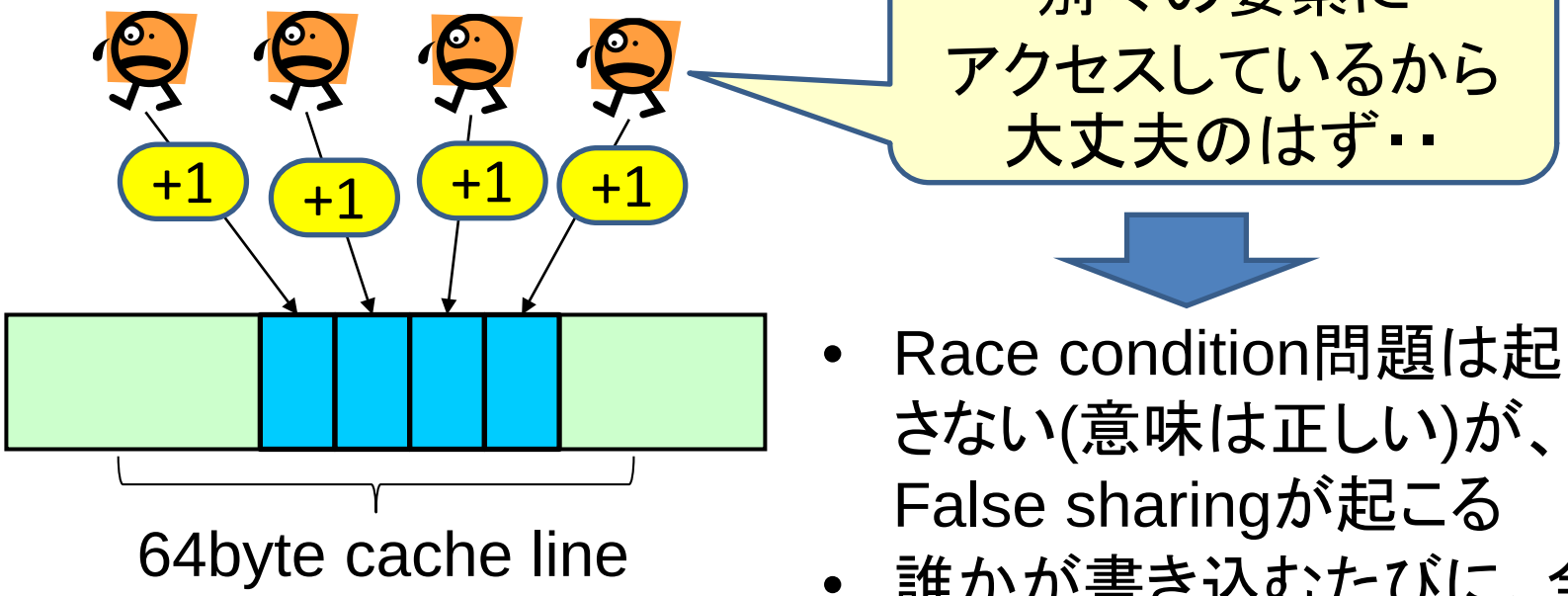
- 共有変数利用には注意が必要
  - 複数スレッドが同じ個所をreadするのは、性能的にok
  - 複数スレッドが同じ個所にwriteするのは、(race conditionが解決されたとしても)性能が下がる
    - 正しさの問題と速さの問題の双方に影響する



- Matmul OpenMP版ではjループを分割・並列化したので
  - A: read-only and shared (ok)
  - B: read-only and localized (ok)
  - C: RW and localized (ok)

False Sharing問題

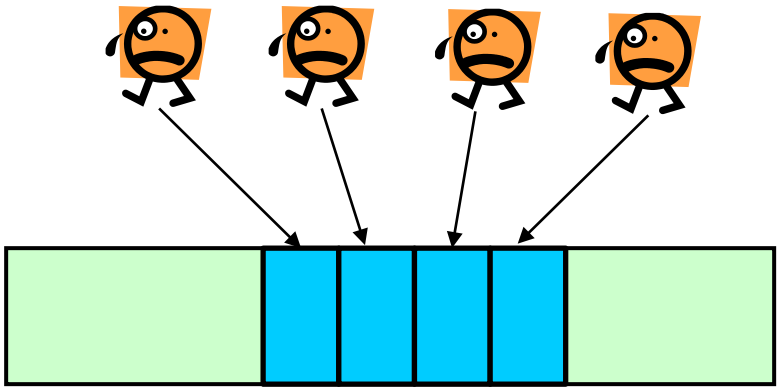
- False sharing問題：プログラムの文面上、別スレッドが別のメモリ(変数)に読み書きしているのに、意図以上に遅くなってしまう
- データの一貫性制御がcache line単位であることに起因



- Race condition問題は起こさない(意味は正しい)が、False sharingが起こる
- 誰かが書き込むたびに、全員分invalidしてしまう

⇒キャッシュミス激増

False Sharing問題の解決

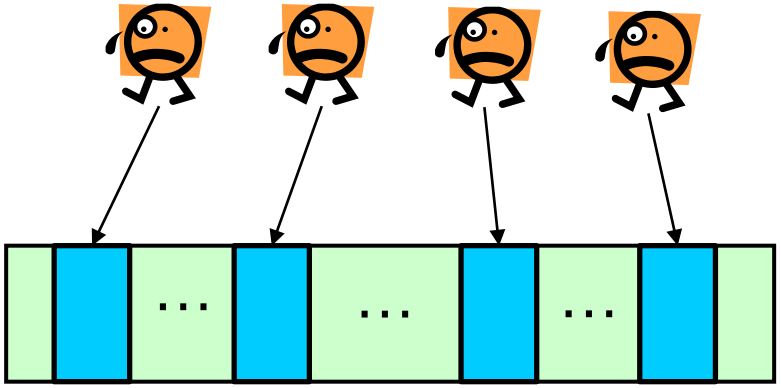


例：*race-condition/rc-fs.c*

```
ss[omp_get_thread_num()]++;
```

のような処理はfalse sharingの原因

解決：64bytes以上はなす

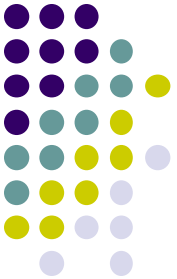


例：*race-condition/rc-fast.c*

プライベート変数を使えば大丈夫

理由：プライベート変数は各スレッドのスタックに置かれ、スタックたちのメモリ上の位置は十分遠い

次回: 5/26(月)



- MPIによる分散メモリプログラミング(1)

  - スケジュールについてはOCW pageも参照
    - http://www.el.gsic.titech.ac.jp/~endo/

    → 2014年度前期情報(OCW) → 講義ノート