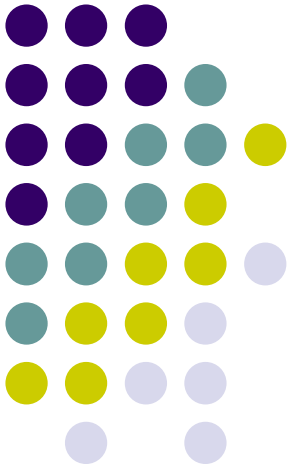
2016年度
実践的並列コンピューティング

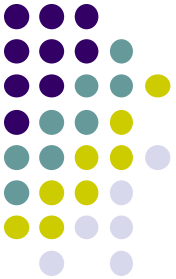
ソフトウェア性能のための
コンピュータアーキテクチャ



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？

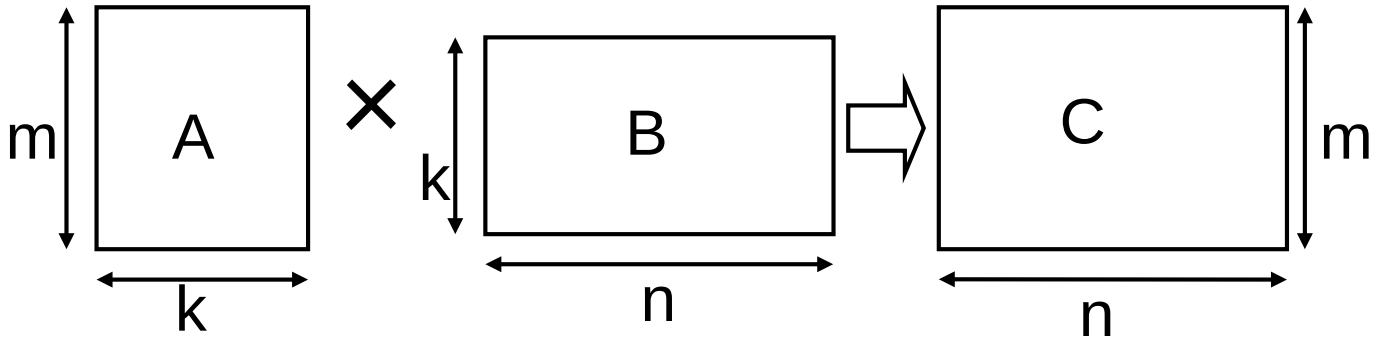


- 正しい逐次アルゴリズムか？
- 速い逐次実装か？
- 正しい並列アルゴリズムか？
  - 依存関係を壊していないか
  - Race conditionはないか
- 速い並列実装か？
  - ノード内で速いか
  - ノード間で速いか

アーキテクチャの知識が必要な場合も

行列積の例

(m×k)行列と (k×n)行列を積




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

浮動小数計算量
2mnk

行列積のループ順番

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

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

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

- 三重ループの順番を変えるとどうなる?
  - 6つの可能性: IJL, ILJ, JIL, JLI, LIJ, LJI
  - この変更は、2mnkである計算量には影響しない
  - 処理の順序だけが違う

ループ順の性能への影響

- 6つのバージョンの行列積の性能を比較
  - Cで記述、並列化なし, gcc 4.3.4, -O2
  - Double型, column major format
  - m=n=k=1024
  - TSUBAME2ノード

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

計算量は全く同じでも、10倍以上の性能差

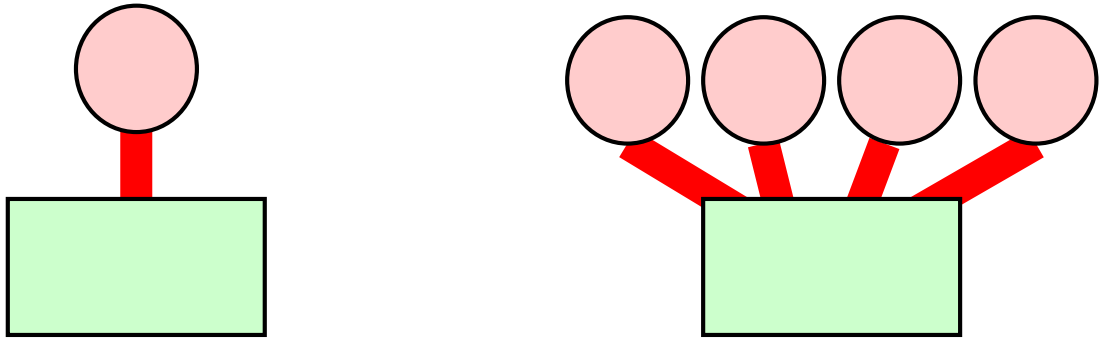
単純なモデルでは
ソフトウェア性能を説明できない

RAM model
(ramdom access machine)

PRAM model
(parallel ramdom access machine)

CPU cores

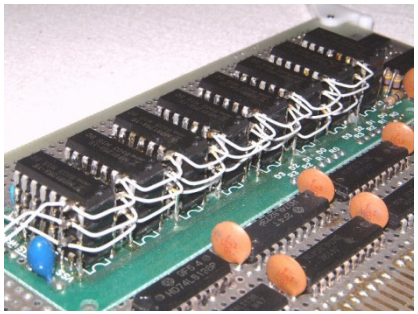
Memory



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、
特にメモリの考慮が必要

CPUとメモリの進化

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

x1000

x40(?)

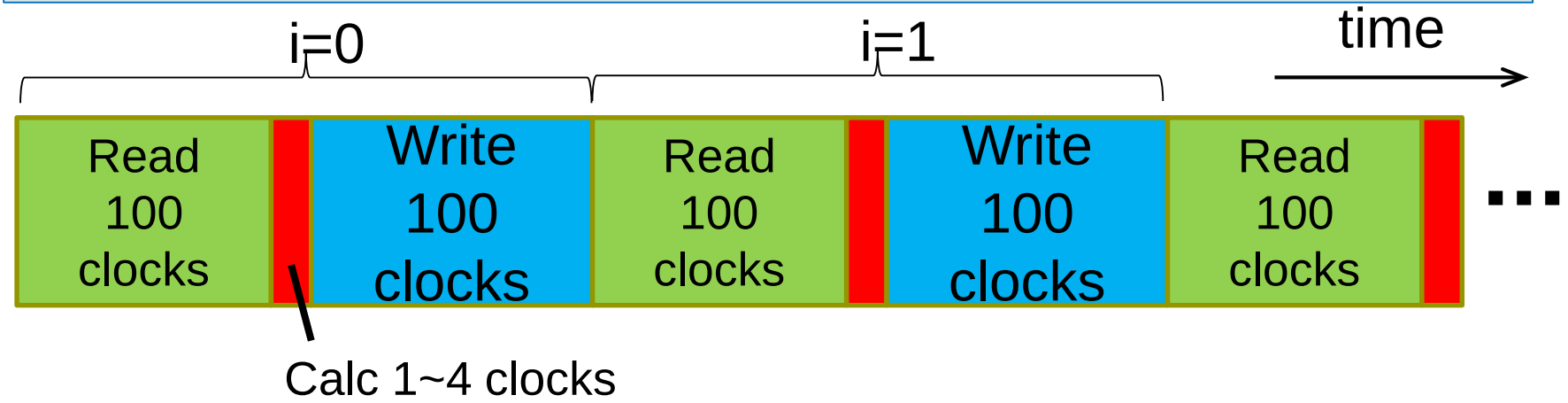
メモリアクセス時間

メモリへの“read”命令にかかる時間
(CPUがメモリアクセス要求を出してから、データが届くまでのレイテンシ)

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

もしも、あらゆるメモリアクセスに100clockかかると？

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

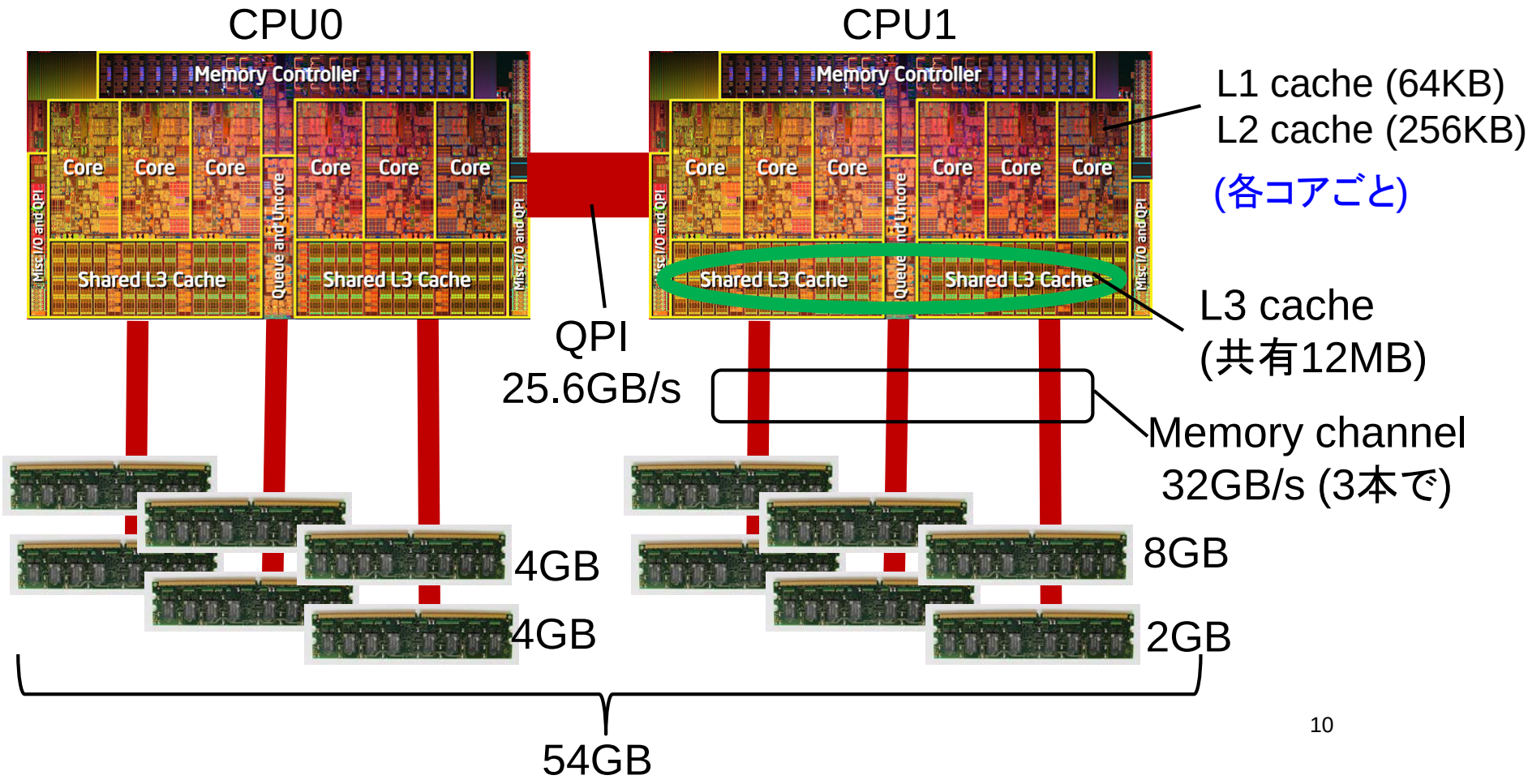


非効率的すぎる
数GHzのCPUでも、10MFlopsを越えられない

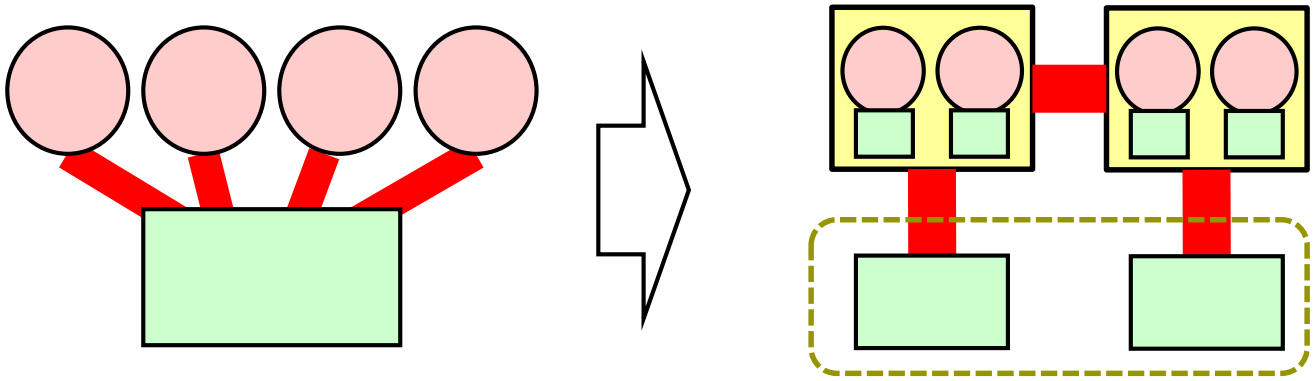
このような問題の解決に向けて、
様々な技術が提案されてきた

実際(に近い)メモリアーキテクチャ

TSUBAME2.5ノードの場合



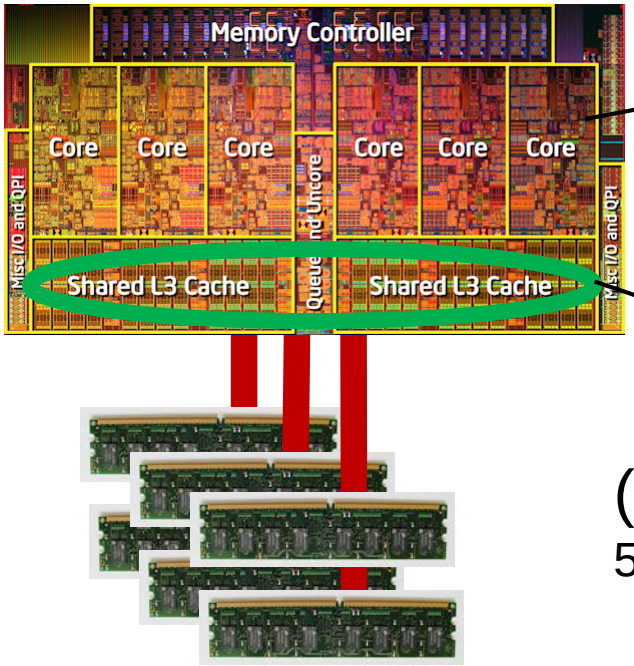
PRAMとの違い



- Cacheがコアごとに存在
- メモリバンド幅は有限であり、複数コアにより共有
- CPUごとにメモリがあるので、コアから近いメモリと遠いメモリが存在
  - NUMA (non-uniform memory) architecture

Cache Memory

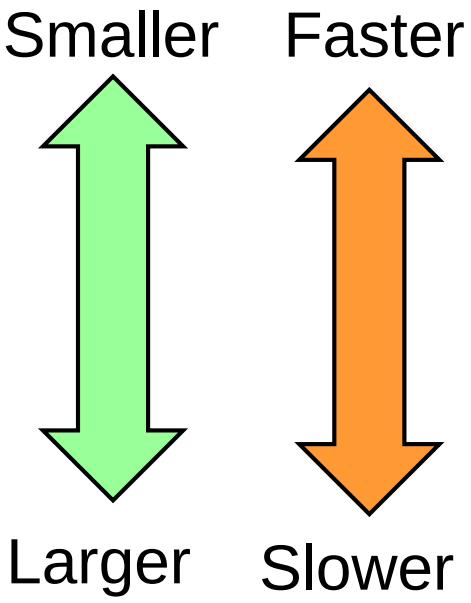
- 高速・小さいメモリであり、通常CPUに含まれる
- 最近アクセスされたデータが格納される
- 自動的に使われる --- プログラマはcacheの存在を知らなくてもプログラミングできる



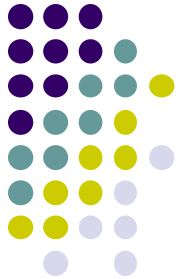
L1 cache (64KB)
L2 cache (256KB)
(included in each core)

L3 cache (12MB)

(Main) memory
54GB on TSUBAME2

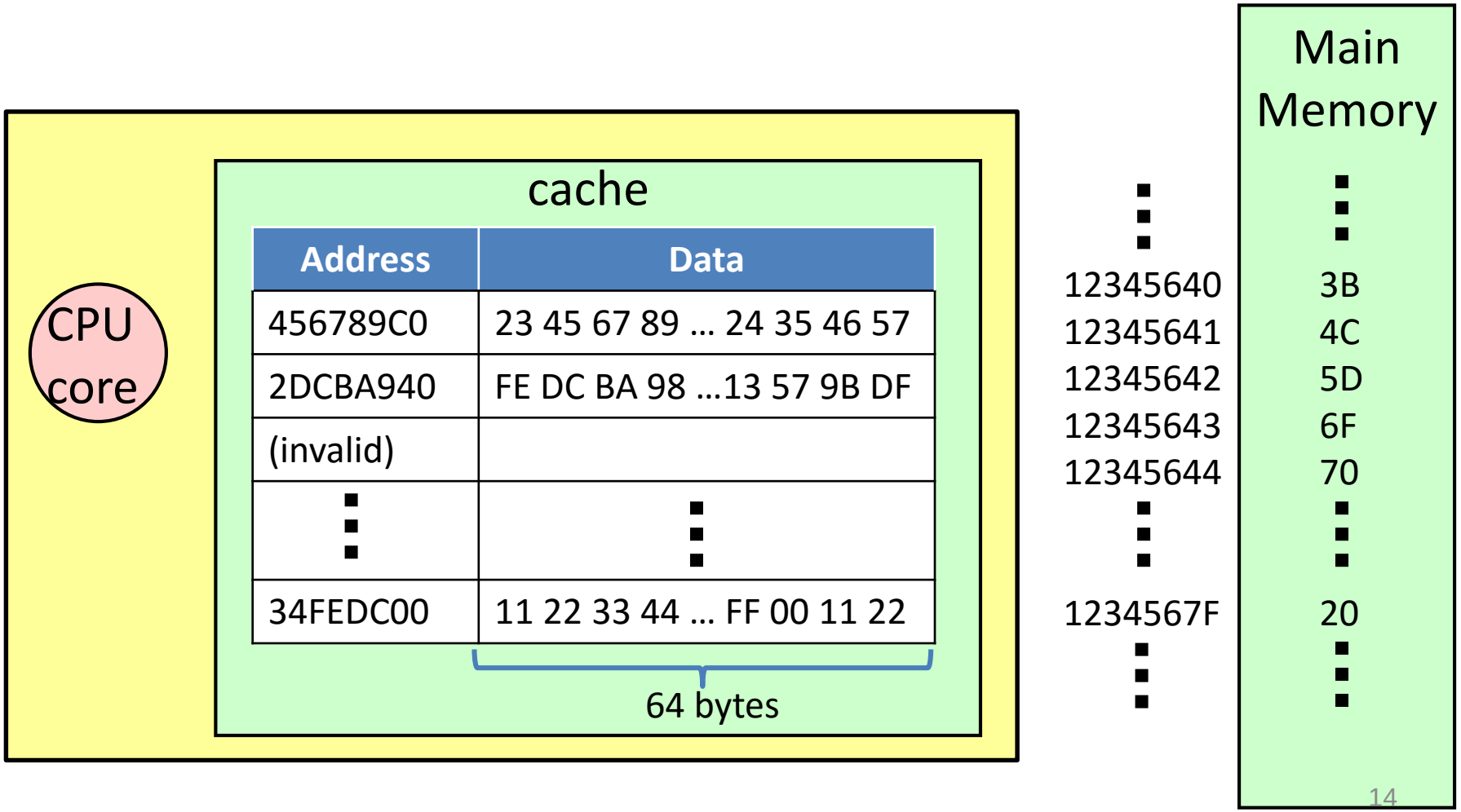


Cacheの容量とcache line



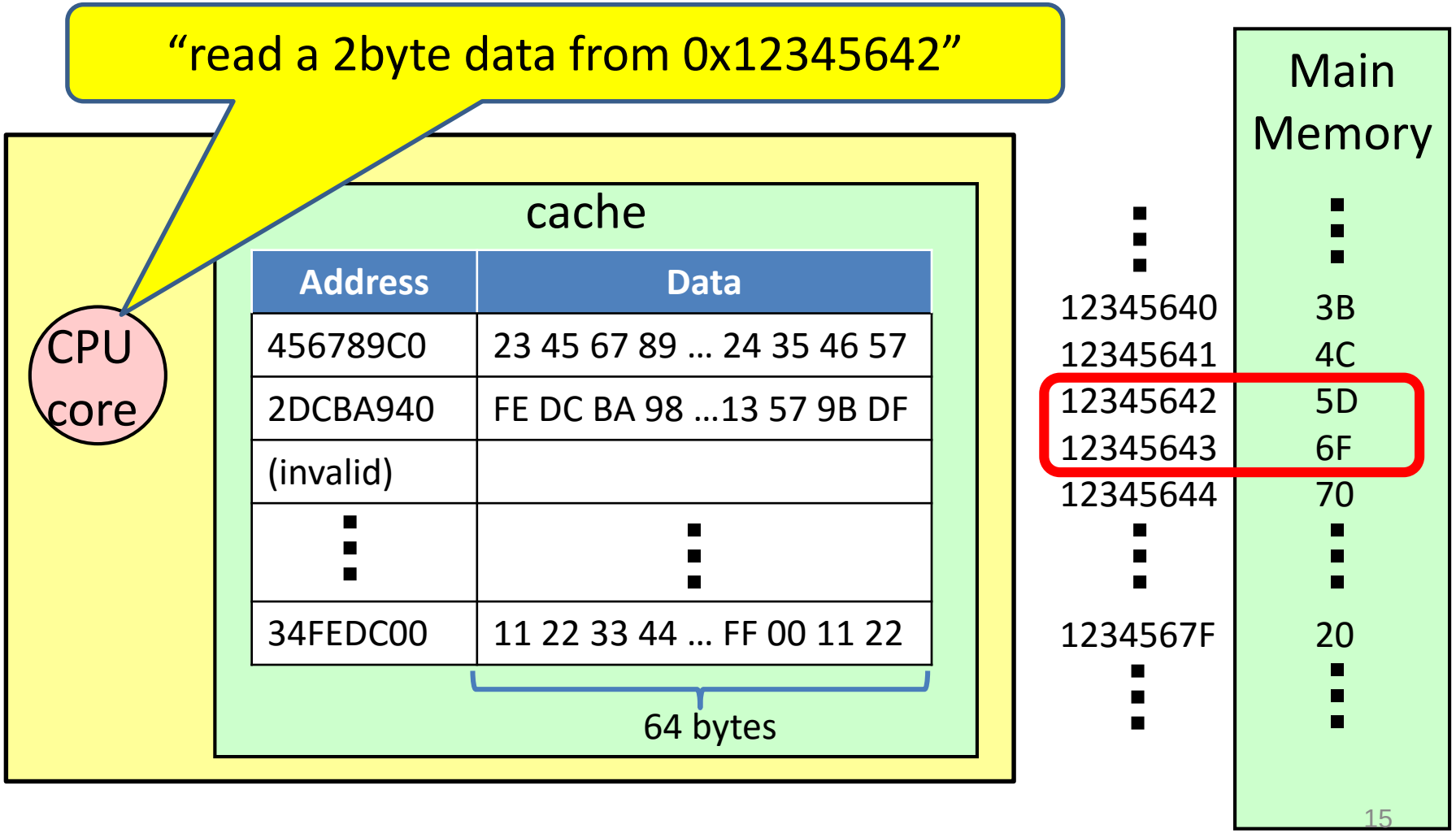
- Cacheには容量がある
  - 数KB ~ 数MB
- データ保存・移動の基本単位は、固定長のcache line
  - Intel CPUの場合はcache line sizeは64byte
  - 容量256KBのキャッシュなら、4096個のcache lineから成り立つ
  - 各cache lineは、「どのアドレスを表すか」の情報を持つ

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.)

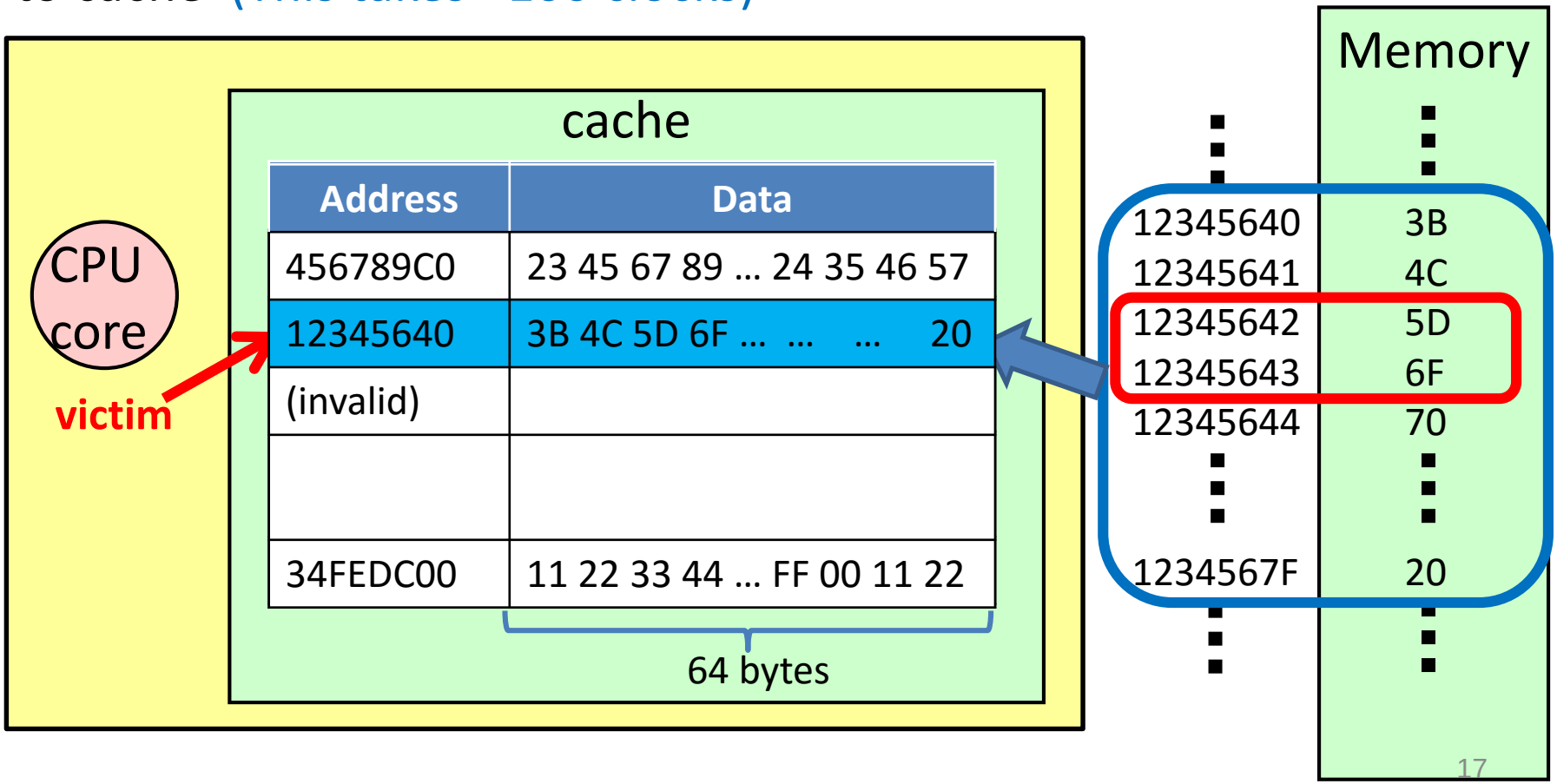
   2-2: If not found, cache miss (This is the case now)

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)



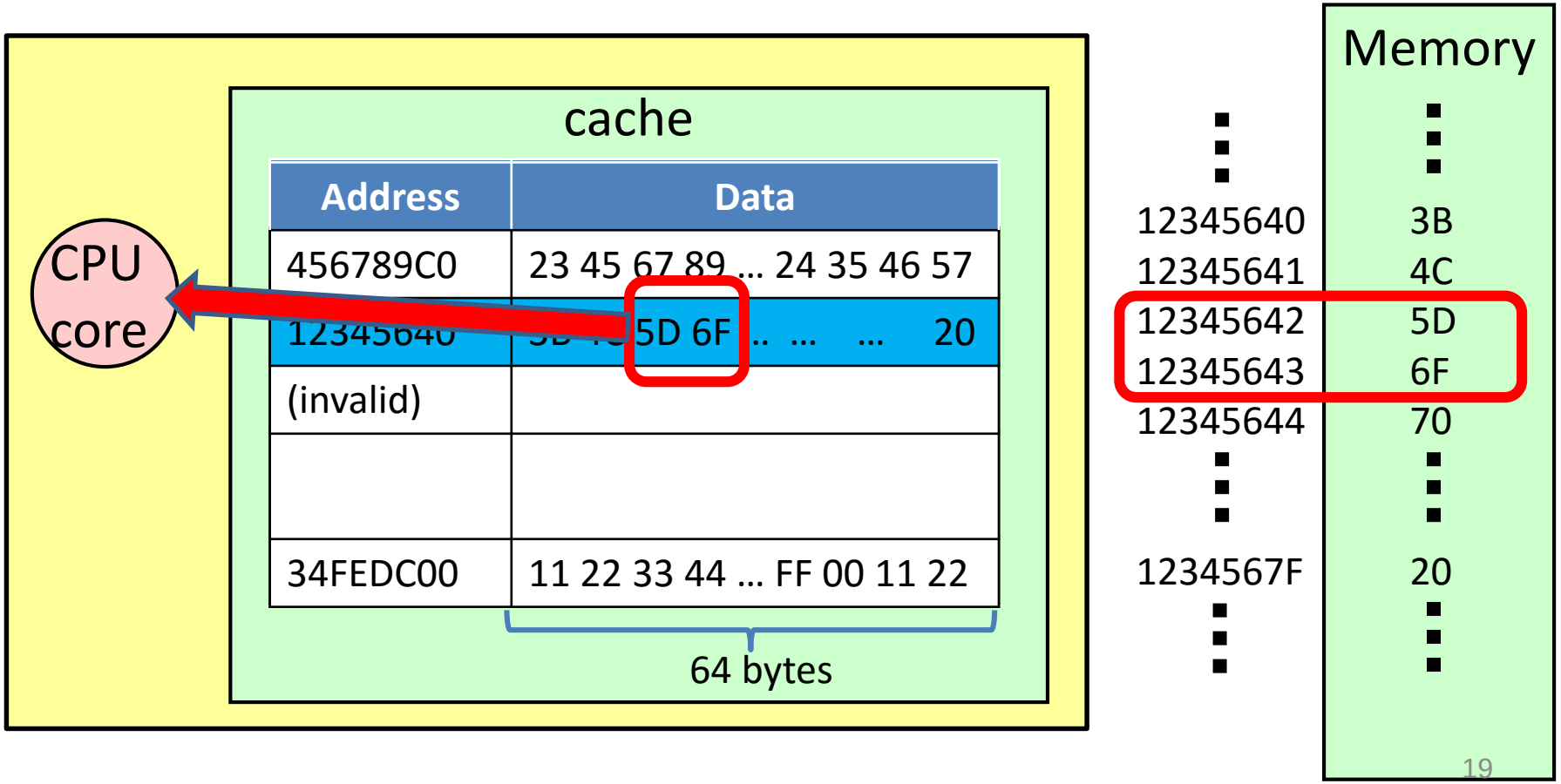
補足:どうやってvictimを選ぶか

- 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

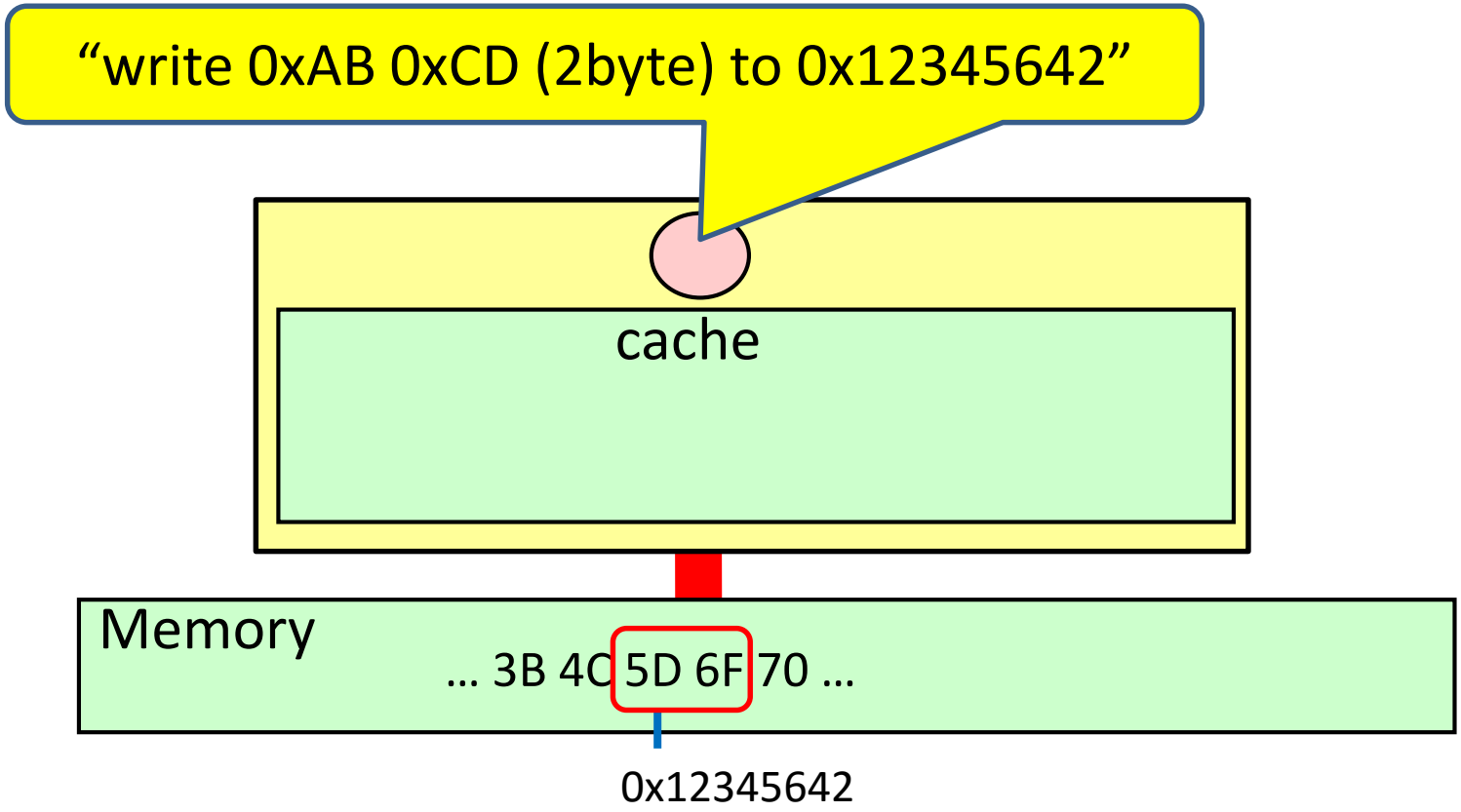
Memory Access with Cache

5. Deliver the desired data to CPU core



Write (1)

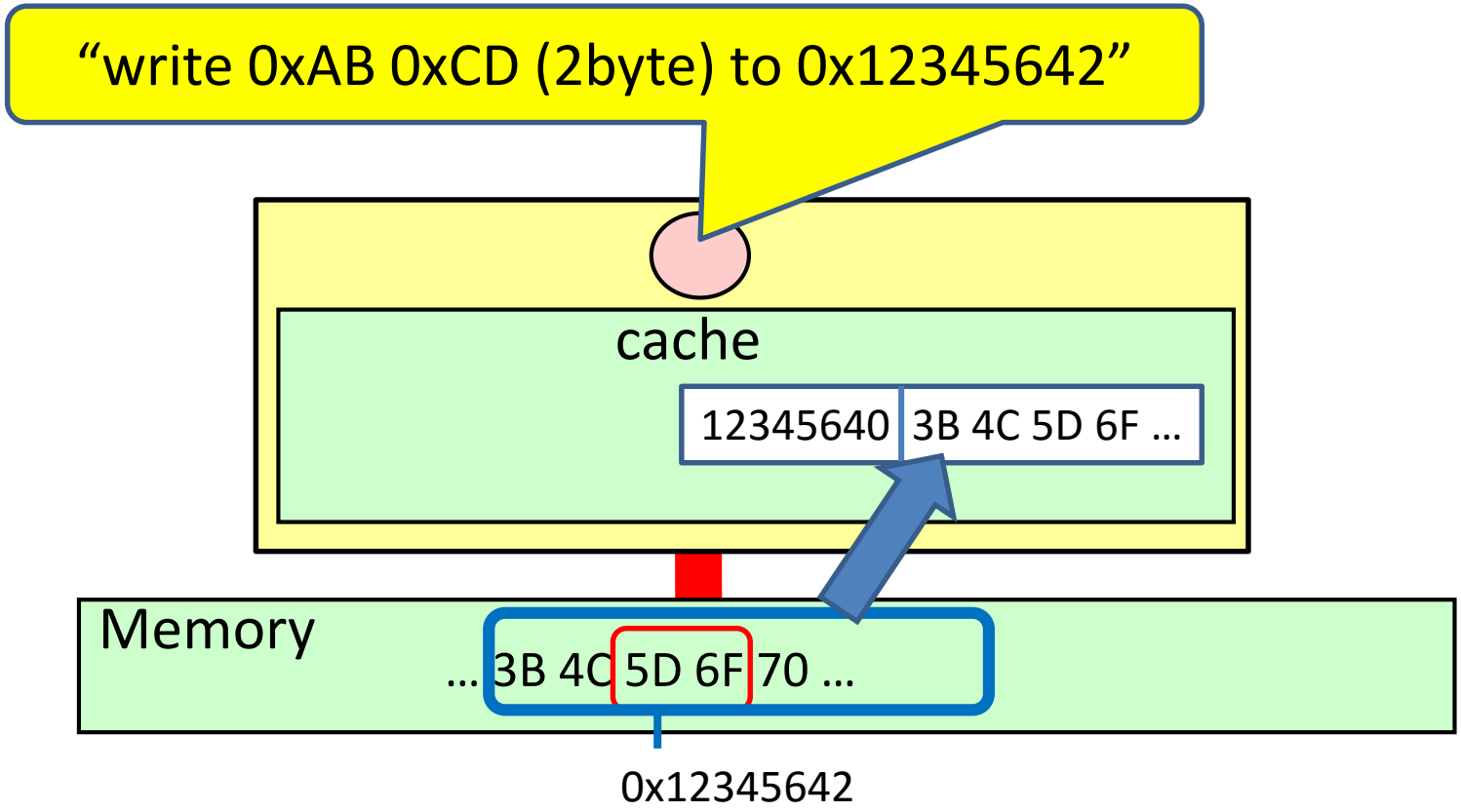
- “Write”アクセスは、“Read”より複雑となる
  - データを変更するので



Write (2)

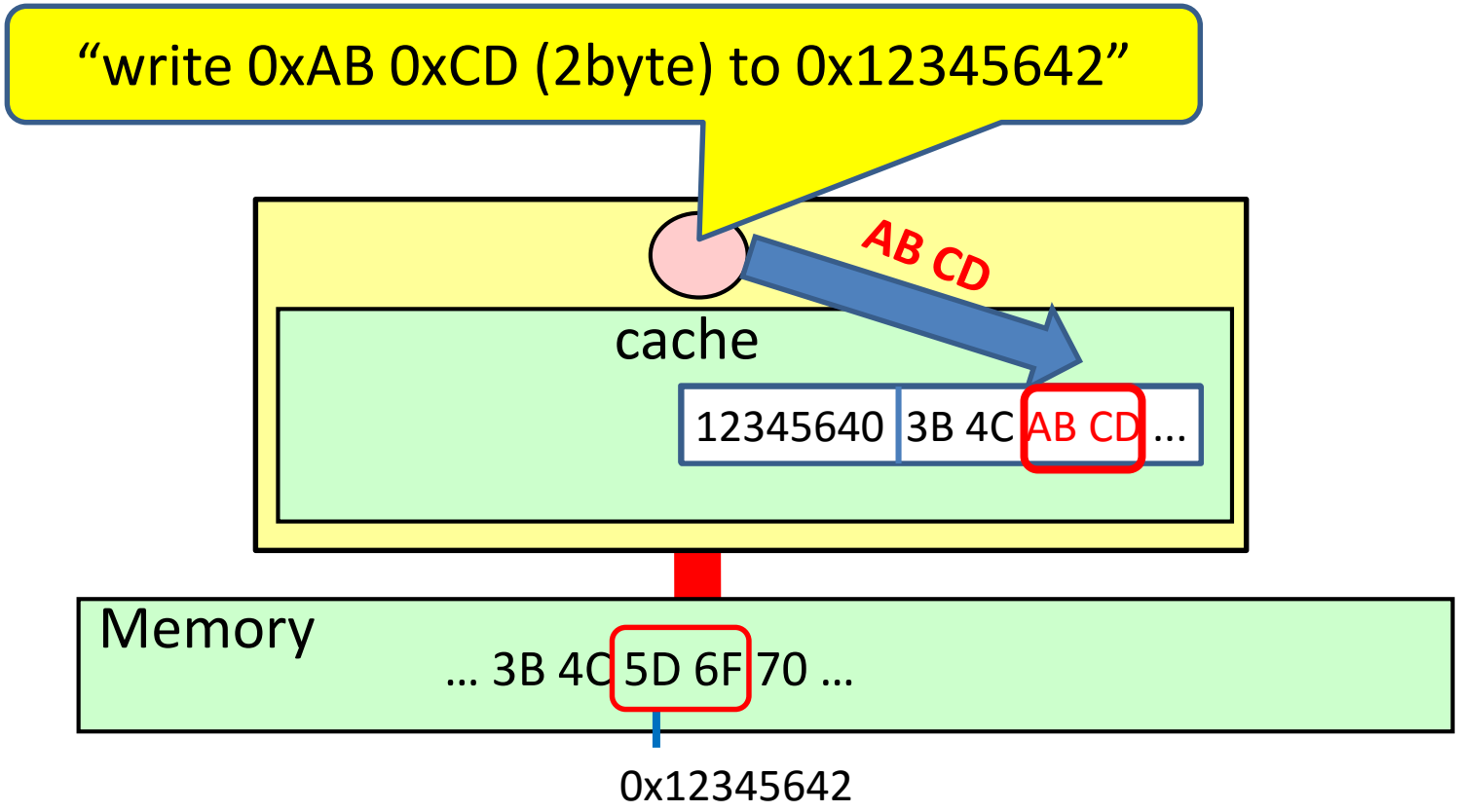
Step 1. 2. 3. are same as “Read”

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

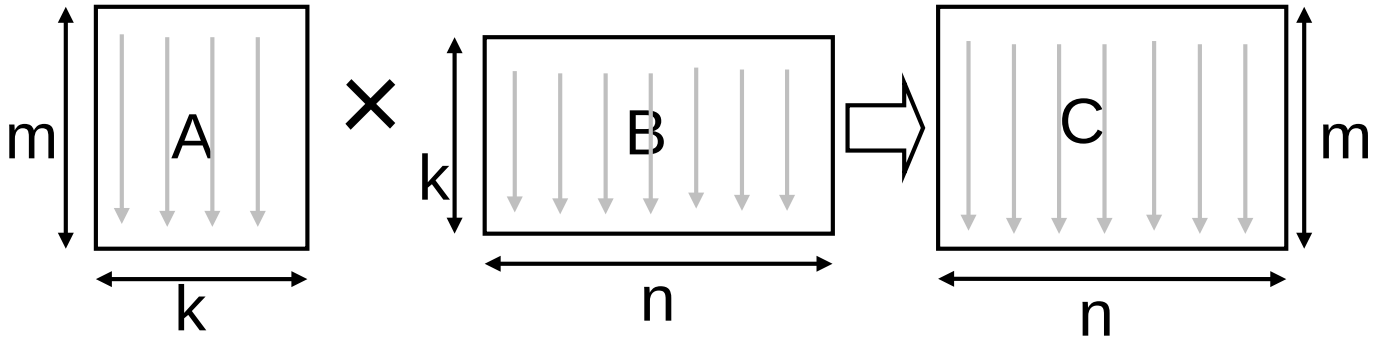
- Write through
  - Update data on memory immediately
  - Problem: Every write instruction takes >100 clocks
- Write back (こちらが主流)
  - Data on memory is updated later
  - When the line is to be deleted as a victim, due to future cache misses
  - Hardware becomes more complex, but efficient

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

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

6つの行列積の性能差は cache hit率の違いによる

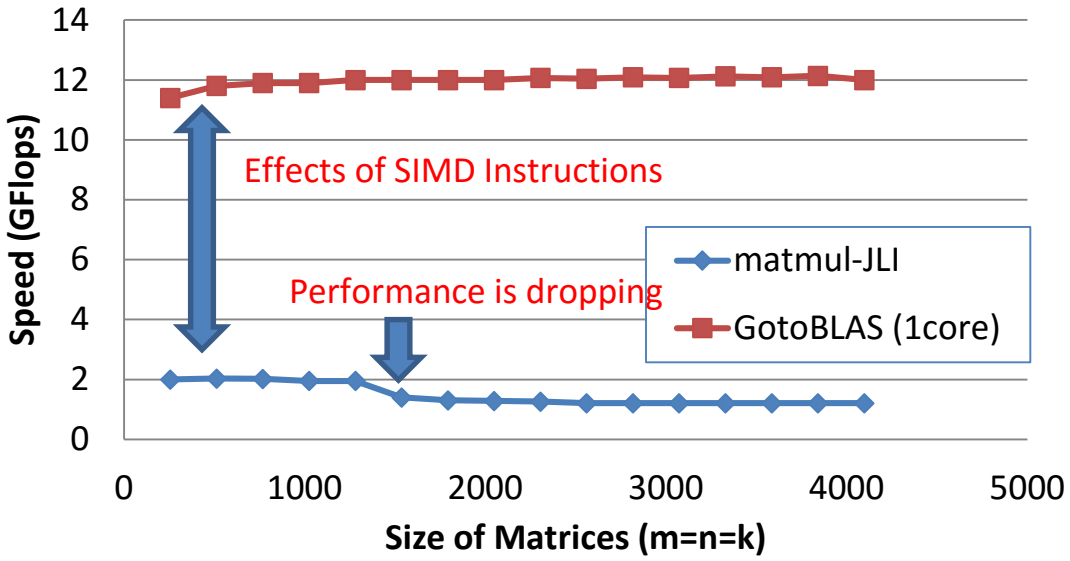



```
for (???) {
  for (???) {
    for (???) {
      Ci,j  +=  Ai,l*Bl,j;
```

- 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

本格的な最適化ライブラリは さらに先を行っている

- Goto BLAS, Intel MKL, AMD ACML…

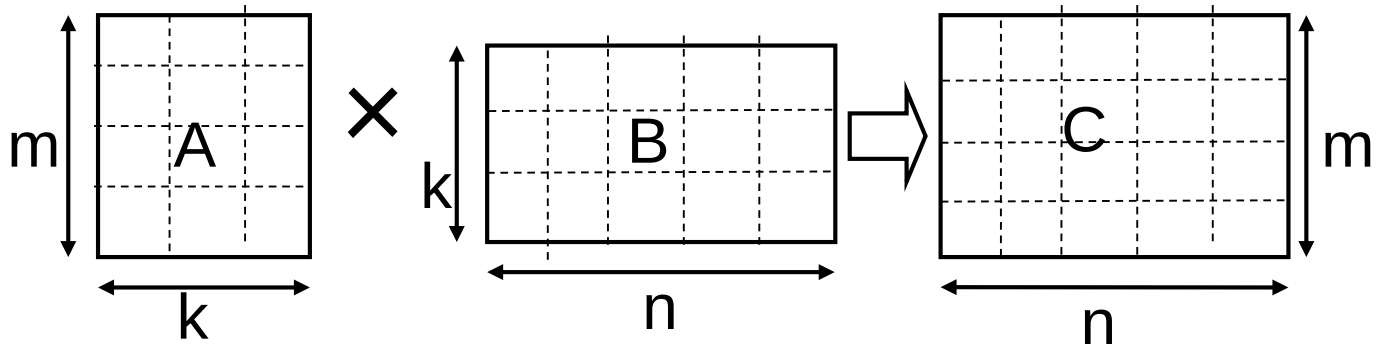


- (Naïve) Simple matmul suffers from more “cache-misses” when problem gets larger
- Optimized GotoBLAS is not only fast, but stable toward the change of problem size

GotoBLASでの最適化

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

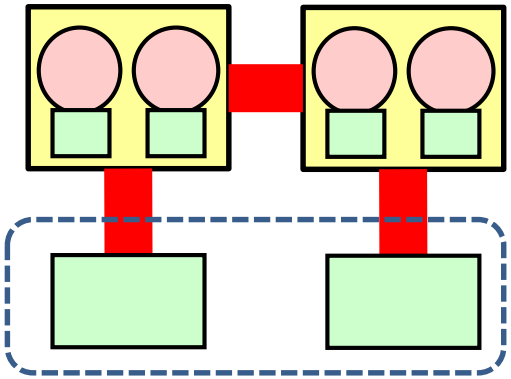
  ブロッキング(タイリング)の考え方により、行列サイズにかかわらずcache hit率向上:



  - 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

マルチコア・マルチCPUとメモリ



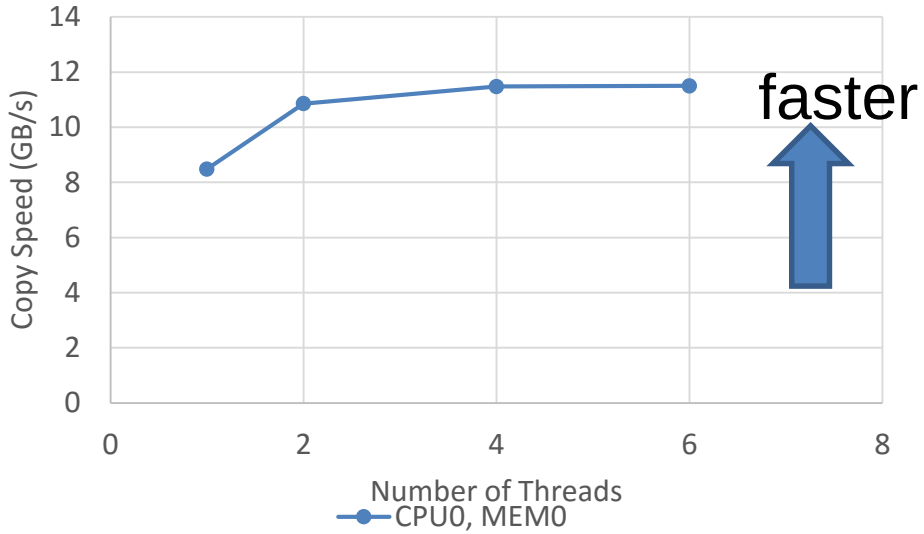
- Cacheがコアごとに存在
- メモリバンド幅は有限であり、複数コアにより共有
- CPUごとにメモリがあるので、コアから近いメモリと遠いメモリが存在
  - NUMA (non-uniform memory) architecture

メモリバンド幅の性能への影響(1)

- 単純な配列コピープログラムのOpenMP版(arraycopyサンプル)で計測。TSUBAME2.5 1ノード

```
#pragma omp parallel for
    for (i = 0; i < N; i++) { b[i] = a[i]; }
```

- まず、CPU0と、CPU0側のメモリ(Mem0)のみ利用



6 thread (6 core)使っても、1.3倍しか速くならない
→ 1回のキャッシュミスが6/1.3 = 4.6倍遅くなっている！！

- メモリバンド幅の限界による「混雑」のため

NUMAアーキテクチャ上の挙動

- スレッドはどのCPUのどのコアで動くか?

→ OS次第。スレッドはコア間を動くこともある。

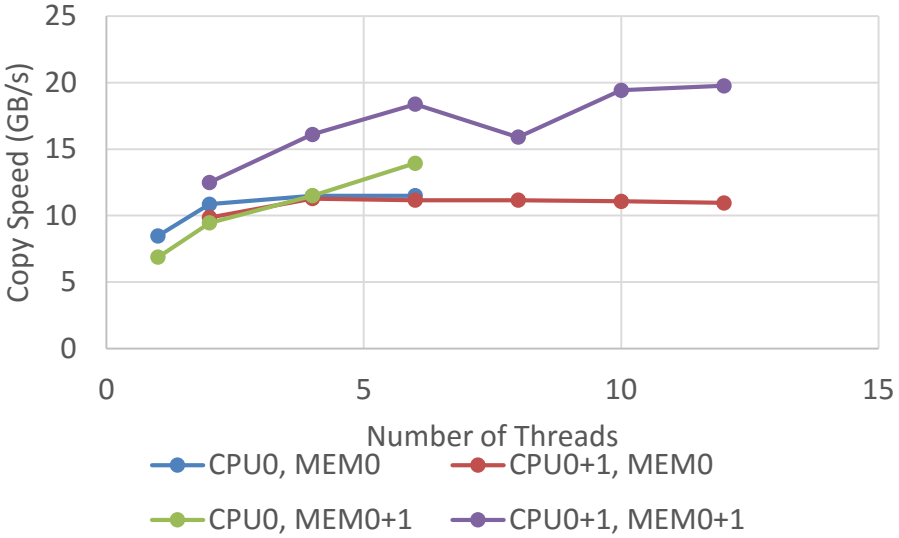
- (mallocなど)確保されたデータ領域はどのCPUのメモリにあるか？

→ OS次第。通常、下記のいずれか

  - First-touch policy: あるページは、プロセス開始後初めてアクセスしたコアの近くに配置される
  - Interleave policy: 計算機内のメモリに均等配置

次ページの実験では、numactl コマンドを使って明示的にポリシー指定した

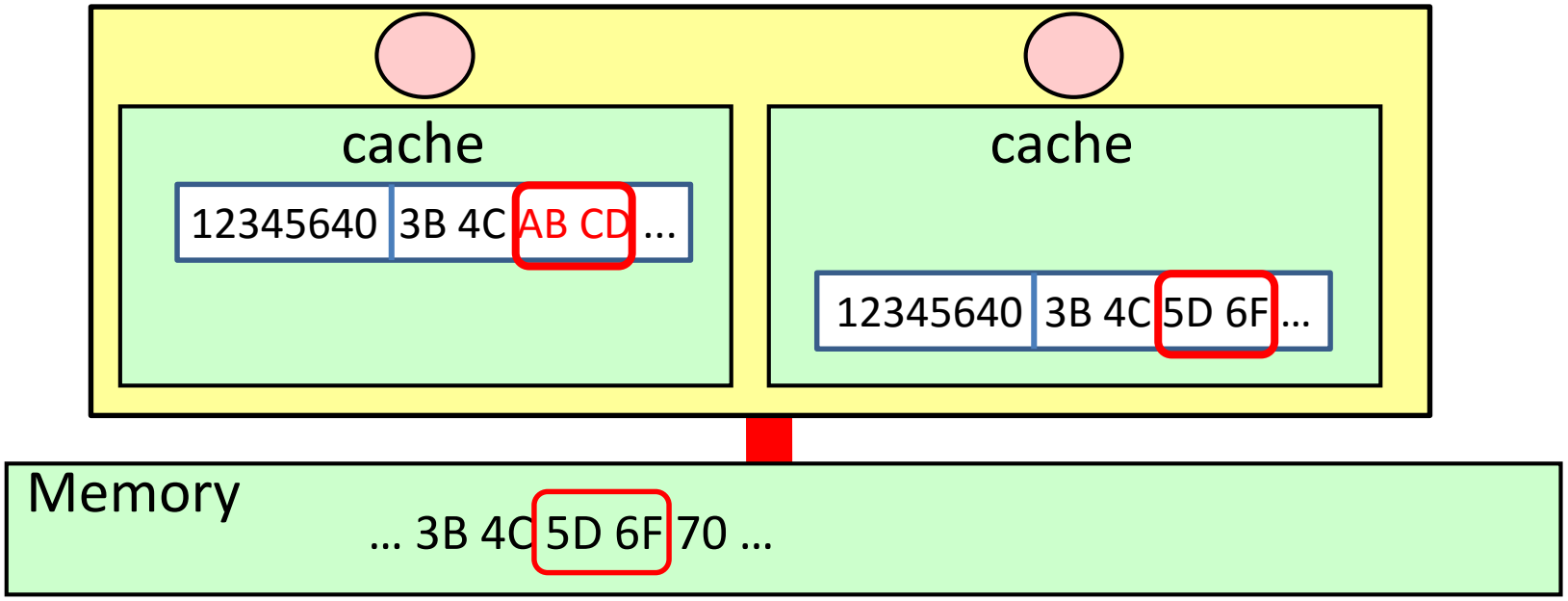
メモリバンド幅の性能への影響(2)



- CPU0: スレッドをCPU0上のみ配置
- CPU0+1: スレッドをCPU0とCPU1に均等配置
- MEM0: データをCPU0のメモリ上のみ配置
  - numactl --membind=0 …
- MEM0+1: データをCPU0とCPU1のメモリに均等配置
  - numactl --interleave 0,1 …

両方のメモリを使うことにより、よりバンド幅を利用可能

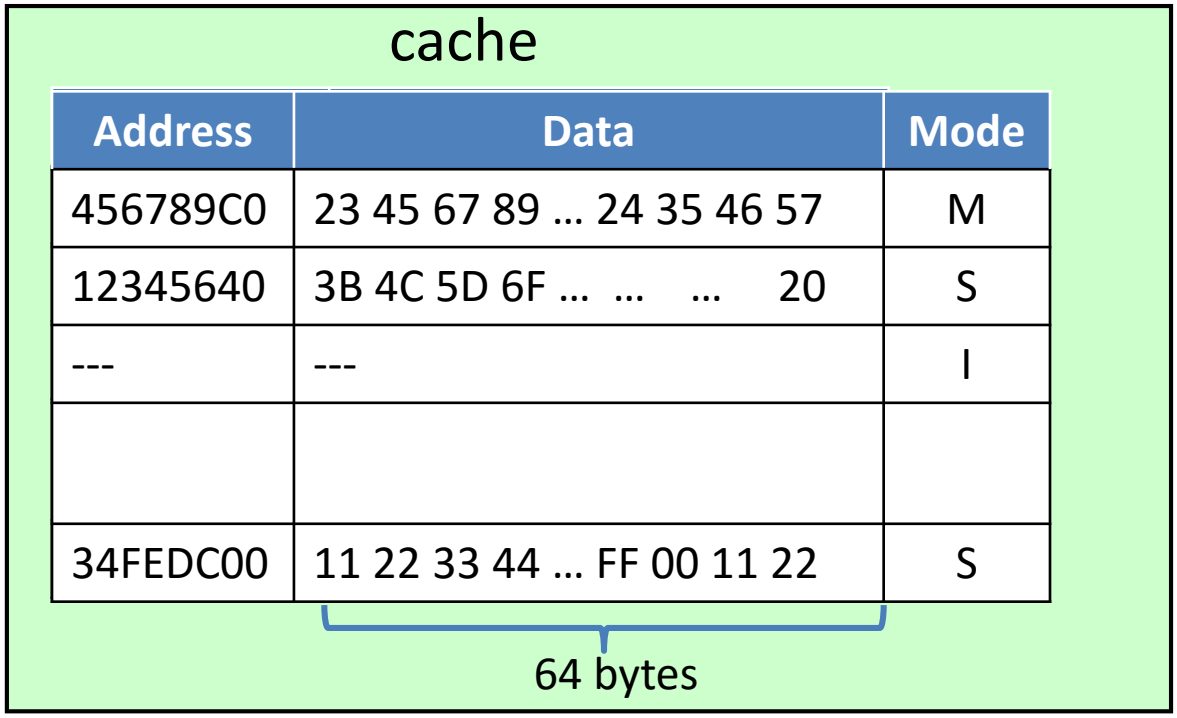
並列キャッシュの難しさ



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

MSI プロトコル

- MSIプロトコルは、キャッシュ間の一貫性を保つ単純なプロトコル
- 各cache lineは、以下の3つのモードのどれかである
  - Modified
  - Shared
  - Invalid
- 1CPU内のマルチコアでも、複数CPU間でも、以下の議論は同じ

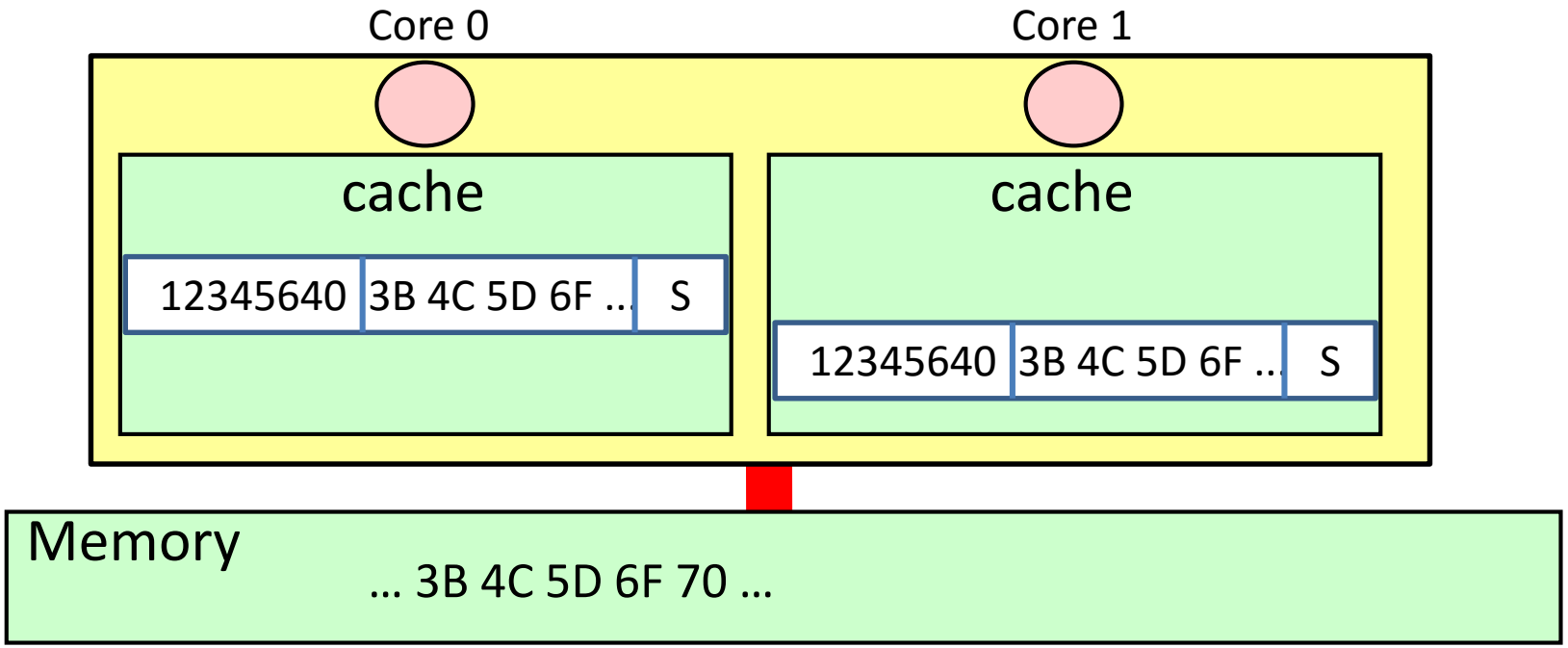


MSIプロトコルの「モード」

- 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

MSIプロトコルの動き (1)

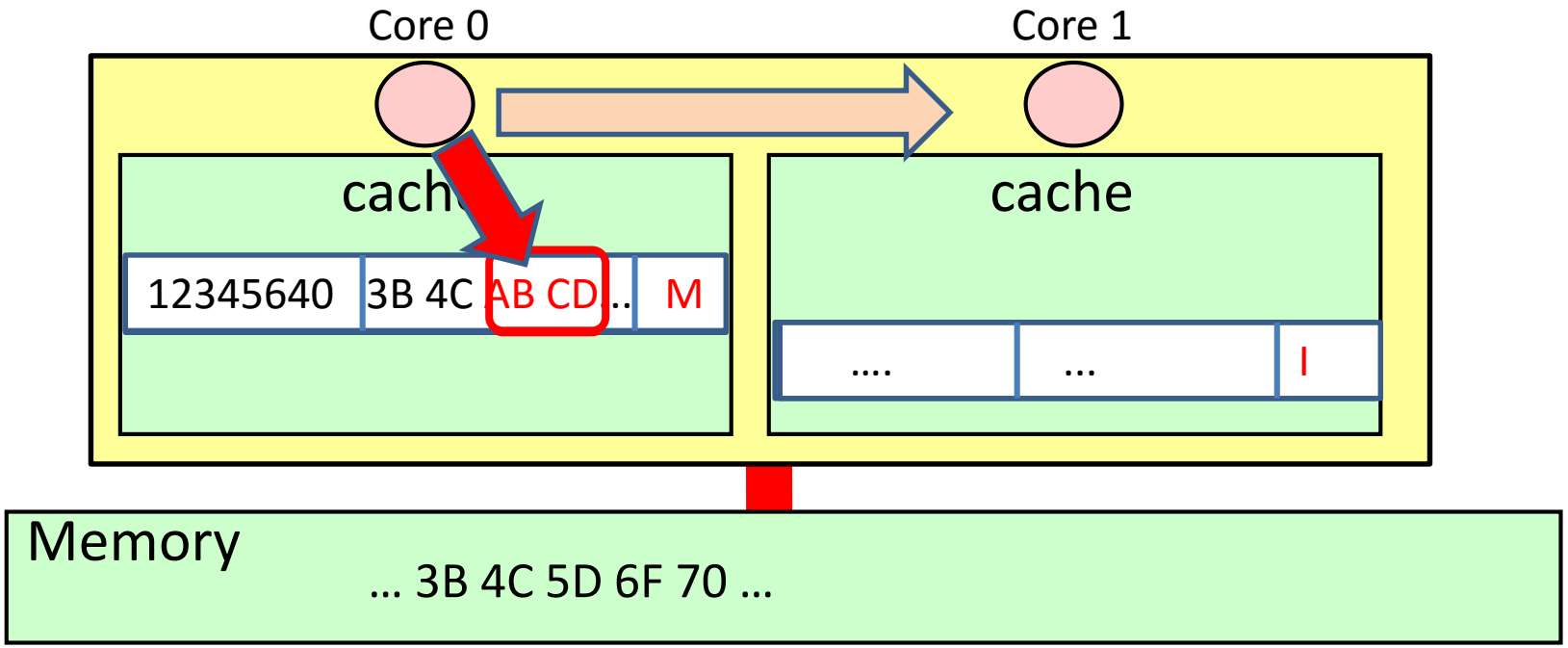
- Here line 12345640 is “Shared”



- What happens if core 0 executes “write”?

MSIプロトコルの動き(2)

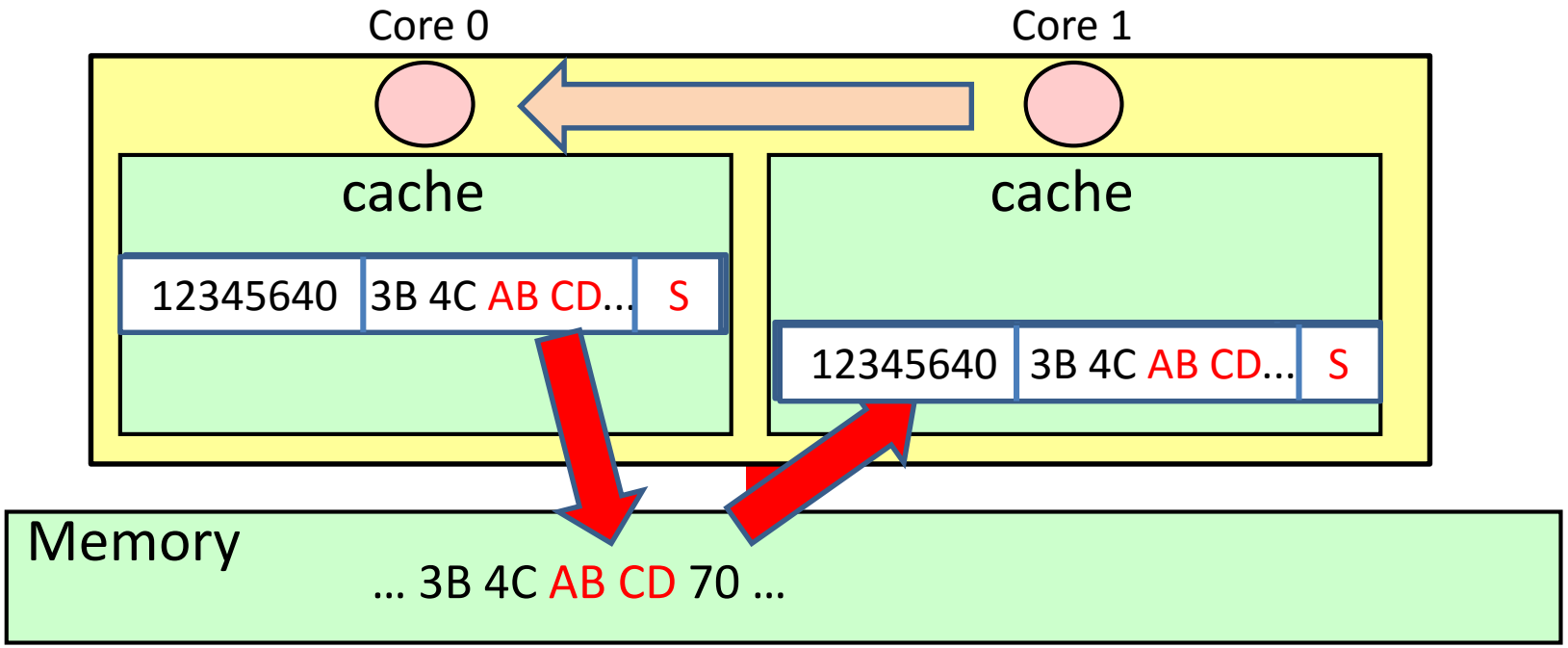
- Write to a shared line includes:
  - Make other shared line (if exist) invalid (called invalidate)
  - Make own line modified
  - Write to its own line



- What happens if core 1 executes “read”?

MSIプロトコルの動き(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)

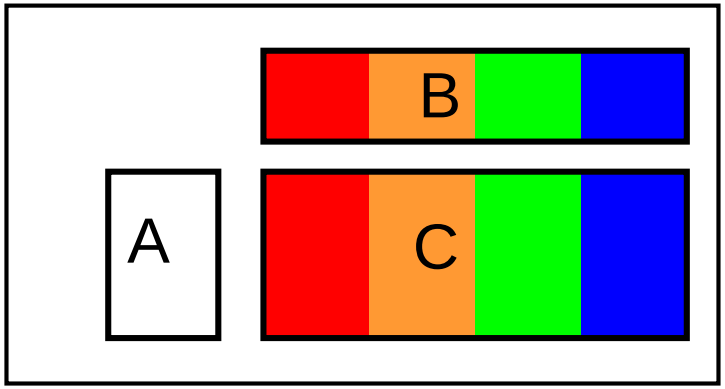
ほかのプロトコル

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

色々あるが、基本はMSIを改良したもの

並列プログラムの速度への影響

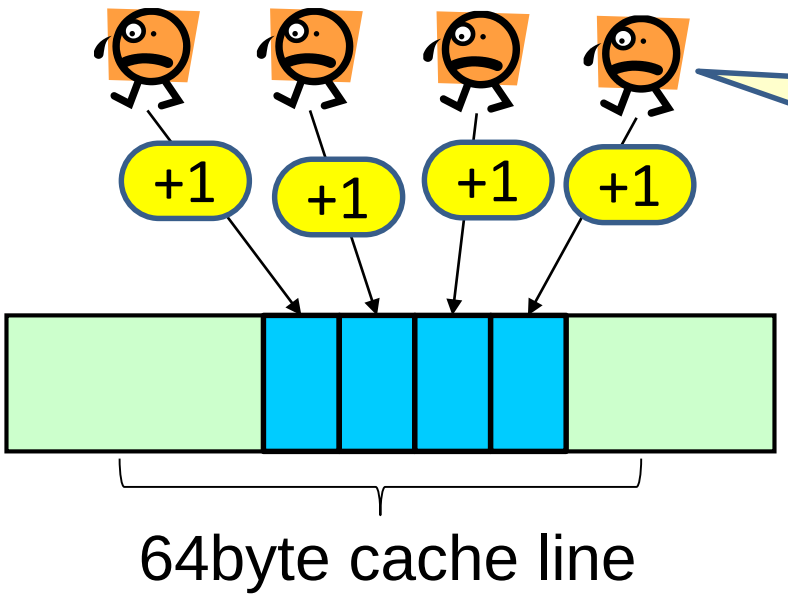
- 共有変数利用には注意が必要
  - 複数スレッドが同じ個所をreadするのは、性能的にok
  - 複数スレッドが同じ個所にwriteするのは、(race conditionが解決されたとしても)性能が下がる
    - 正しさの問題と速さの問題の双方に影響する



- 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

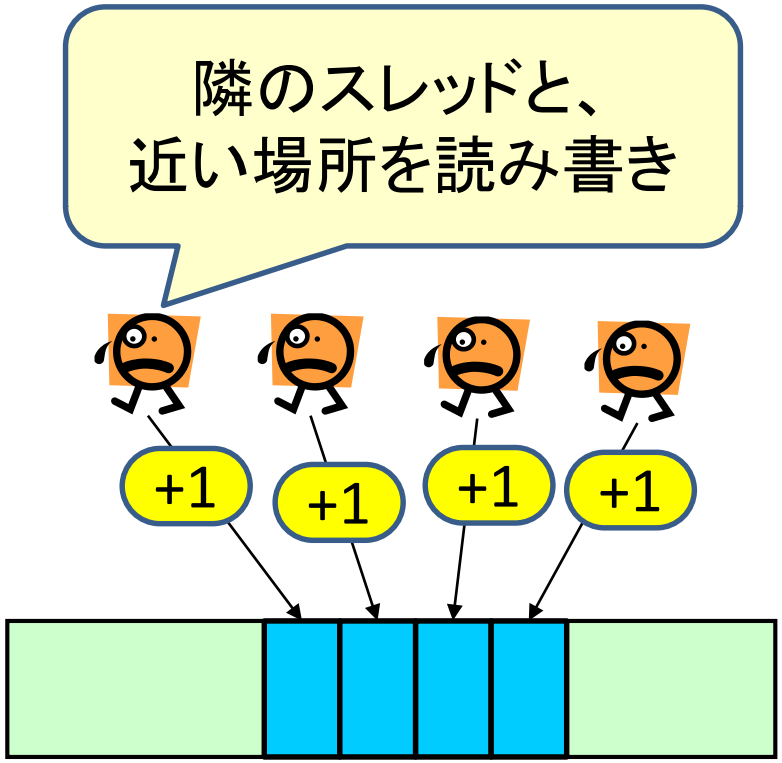


別々の要素に
アクセスしているから
大丈夫のはず・・

- 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

CPUとGPUでは、全く逆のことが起きる

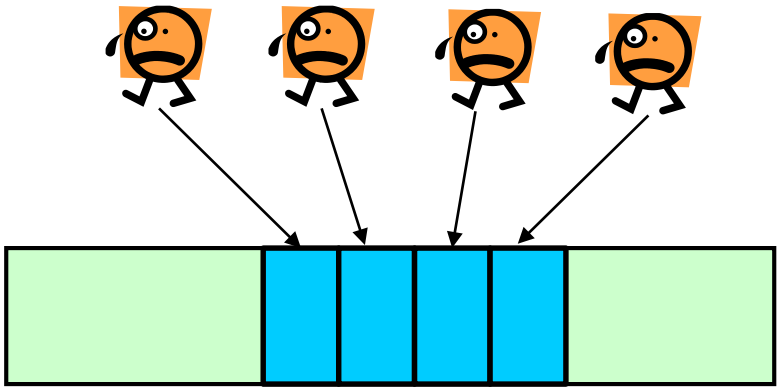


- 左のケースは、GPUでは”coalesced access”となり「好ましい」ケースだった。なぜ違う?

[CPUでは] 別coreは別cacheを持ち、invalidateが発生してしまうから

[GPUでは] 別CUDA coreでも原則共有cacheであり、かつwarp内の同時アクセスは、「近くなら」一アクセスとみなされる

False Sharing問題の解決

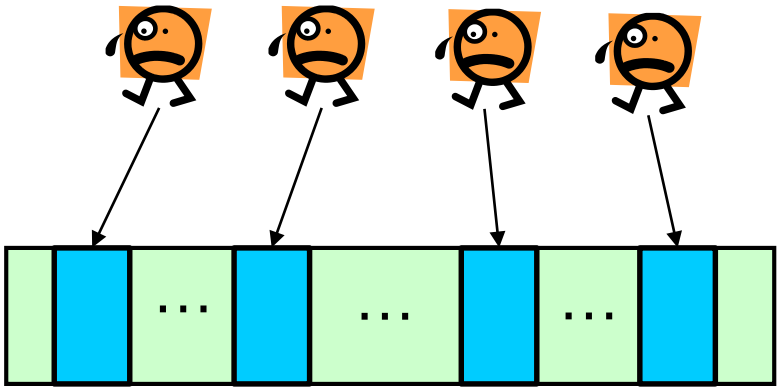


例：*race-condition/rc-fs.c*

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

メンテナンスのため TSUBAME停止予定

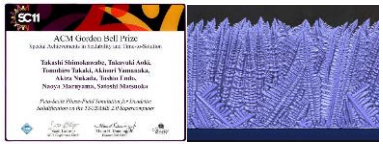
- 2016/6/3(金) 9:00 -- 6/7(火) 17:00
- http://tsubame.gsic.titech.ac.jp/node/1462

2017年　東工大スパコンは TSUBAME3.0へ

1. リーディングスパコン：京を超える性能：~15ペタフロップス、4ペタバイト/秒メモリバンド幅、1ペタビット級光ネットワーク
2. ビッグデータスパコン：大規模シリコンストレージによりマルチテラバイト/秒のI/O、様々なビッグデータ用ミドルウェアやアプリ
3. グリーンスパコン10ギガフロップス/W以上の性能電力性能・グリーン・エネルギー回生
4. クラウドスパコン：種々クラウドサービスと高速性の両立(含コンテナ技術)
5. 「見える化」スパコン：超複雑なマシンの状態が「見える」

2006年 TSUBAME1.0
80テラフロップス・アジアー位
世界7位

2010年 TSUBAME2.0
2.4ペタフロップス・世界4位
運用スパコングリーン世界一



2011年ACMゴードンベル賞

2013年TSUBAME2.5
アップグレード(補正予算)
5.7ペタフロップス

2013年TSUBAME-KFC
スパコングリーン世界一

2017年 TSUBAME3.0
~15(DFP)/30(SFP) ペタフロップス
グリーン・ビッグデータ
リーディングスパコン

大規模学術
及び
産業利用アプリ