2015年度
実践的並列コンピューティング

第13回　特別講義：メモリシステム

平成27年7月13日

学術国際情報センター　特任講師
佐藤 幸紀

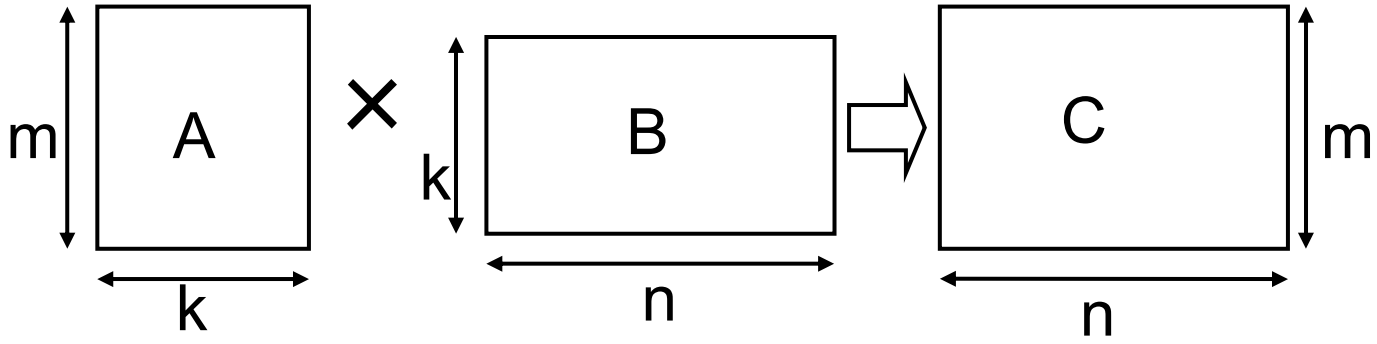
並列ソフトウェアを作るポイントは？

- 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

例　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)

実装のバリエーション

実装上、ループの実行順番を入れ替えれる

IJL

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

JIL

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

LIJ

```
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

ソフトウェア実装の影響

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

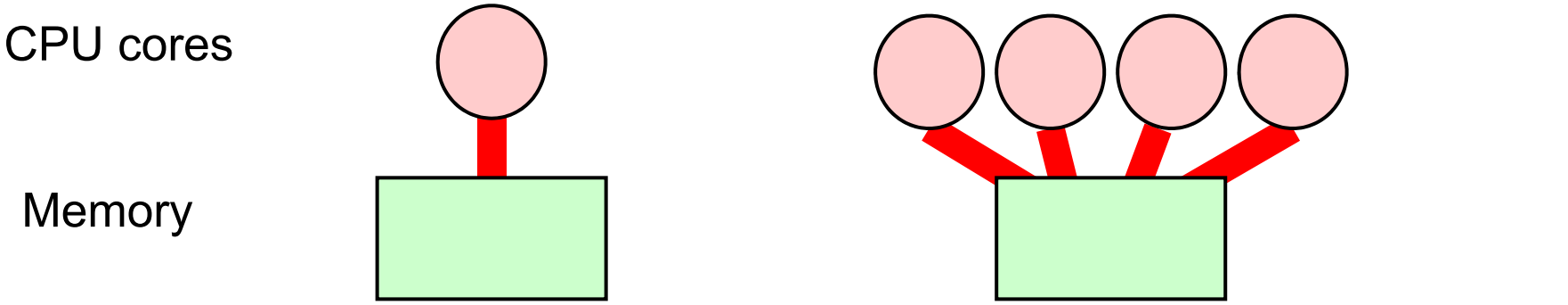
Imple	IJL	JIL	ILJ	LIJ	LJI	JLI
Time (sec)	8.51	8.52	17.5 Slow!	17.5 Slow!	1.30	1.11 Fast!

全ての実装の計算量は同じであるが、計算スピードには大きな違いが発生！

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)

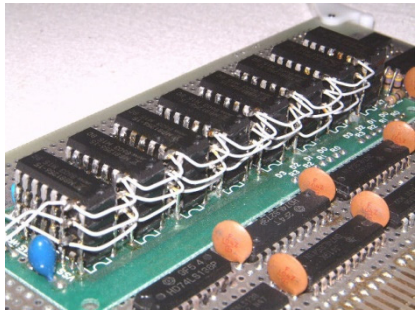
PRAM model
(parallel ramdom access machine)



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(?)

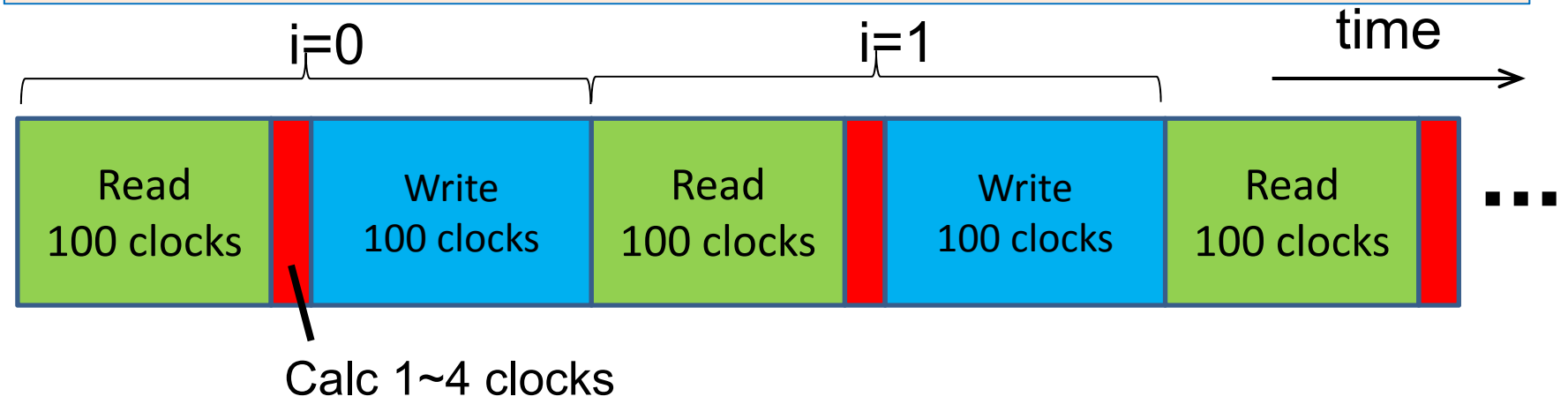
Memory Access Time

メモリアクセス命令はどのくらいの時間がかかる？

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

全てのメモリアクセスが100サイクルの世界

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

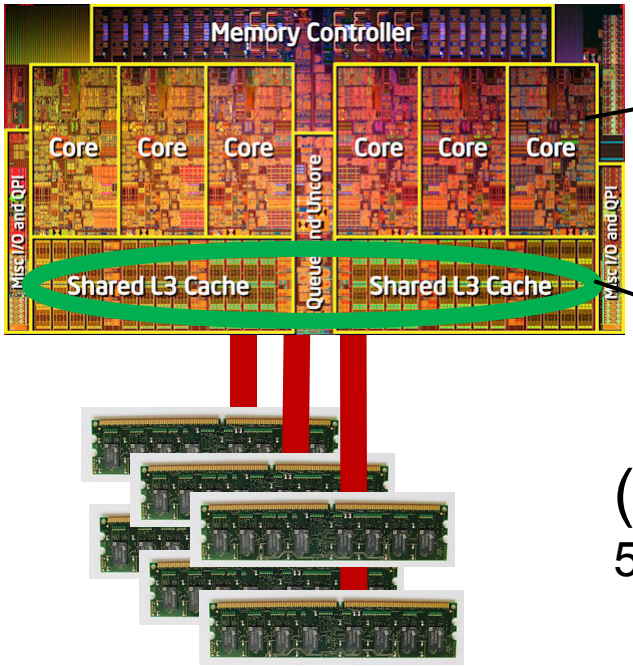


This is very insufficient!
Computation speed would be only 10MFlops

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

Cache Memory

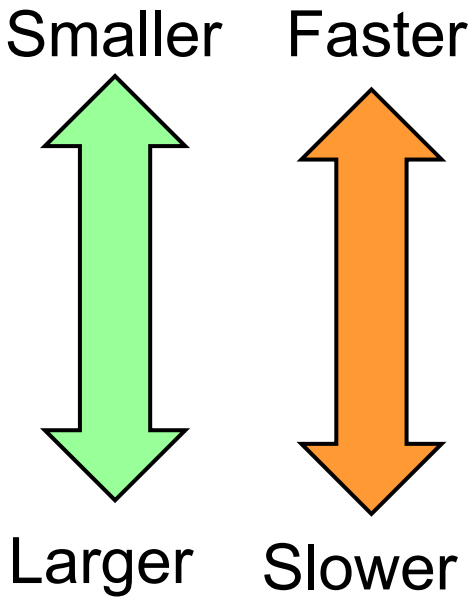
- Fast and small memory (usually) included in CPU
- Used to store data that have been recently accessed
- Used automatically --- It is OK that programmers do not know existence of cache memory



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

キャッシュメモリの原理

図書館で文献調査をしてレポートを書く例

- 関連しそうな項目の本棚をチェック
- 重要そうな数冊を選ぶ
- 自分の机に戻り、座って読む
  - 自分の机と本棚を往復するのはとても面倒
  - 良い本はレポートが終わるまで机の上にキープ
  - 調べたいことが書いてなければ棚に戻す
  - レポートの内容が満足になるまで必要であれば本を探す

教訓:
よく使うものは近くに置くと時間が節約できる
近くにおいて置けるものの量は限られている

キャッシュメモリの原理を絵にすると

図の出展　http://csillustrated.berkeley.edu/illustrations.php

メモリの情報をリクエスト

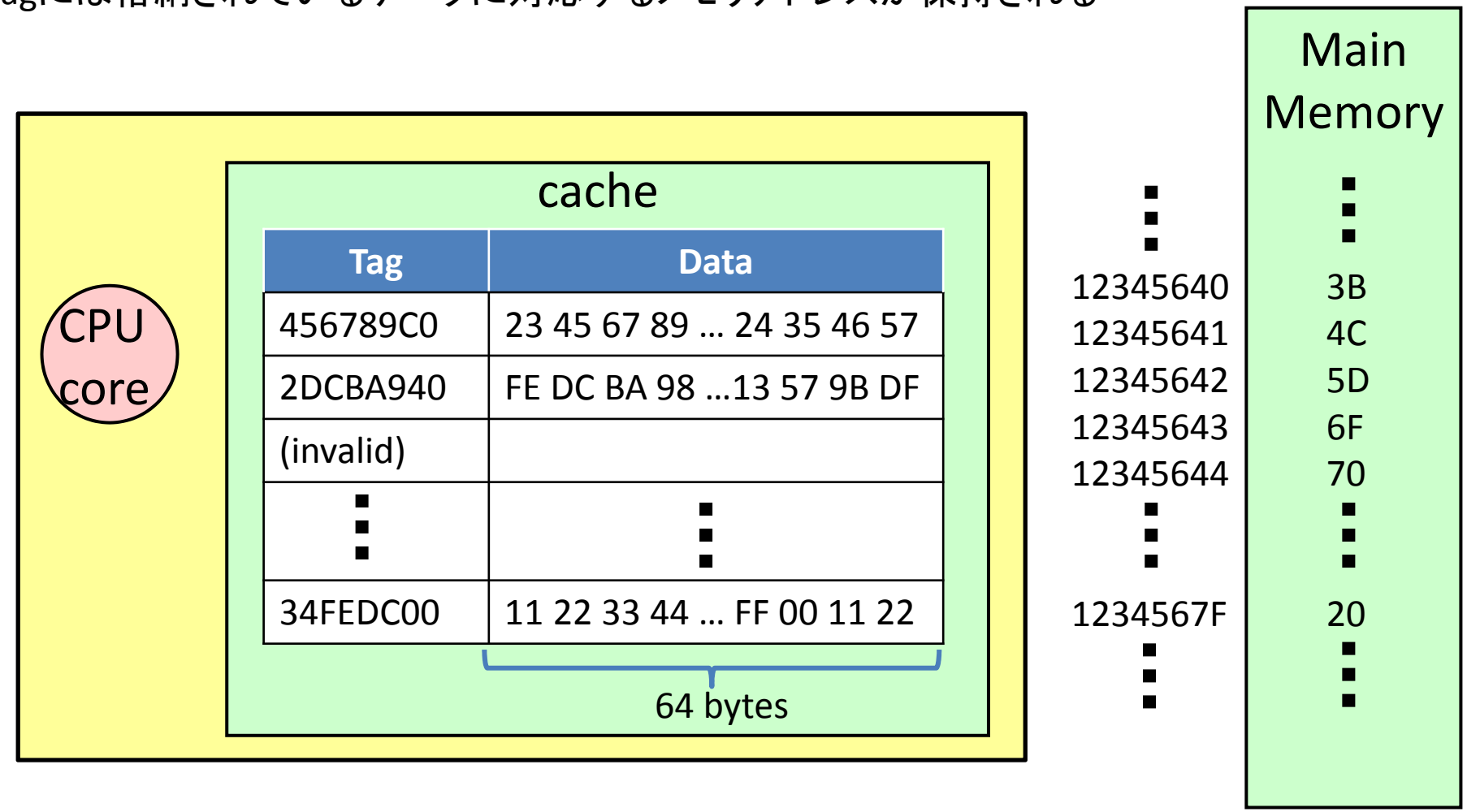
遠くのメモリから必要なものを取り出す

必要とする間のみ近くに置く

よく使うものは近くに置く
近くには必要となるものだけを置く

単純化したキャッシュとメモリ

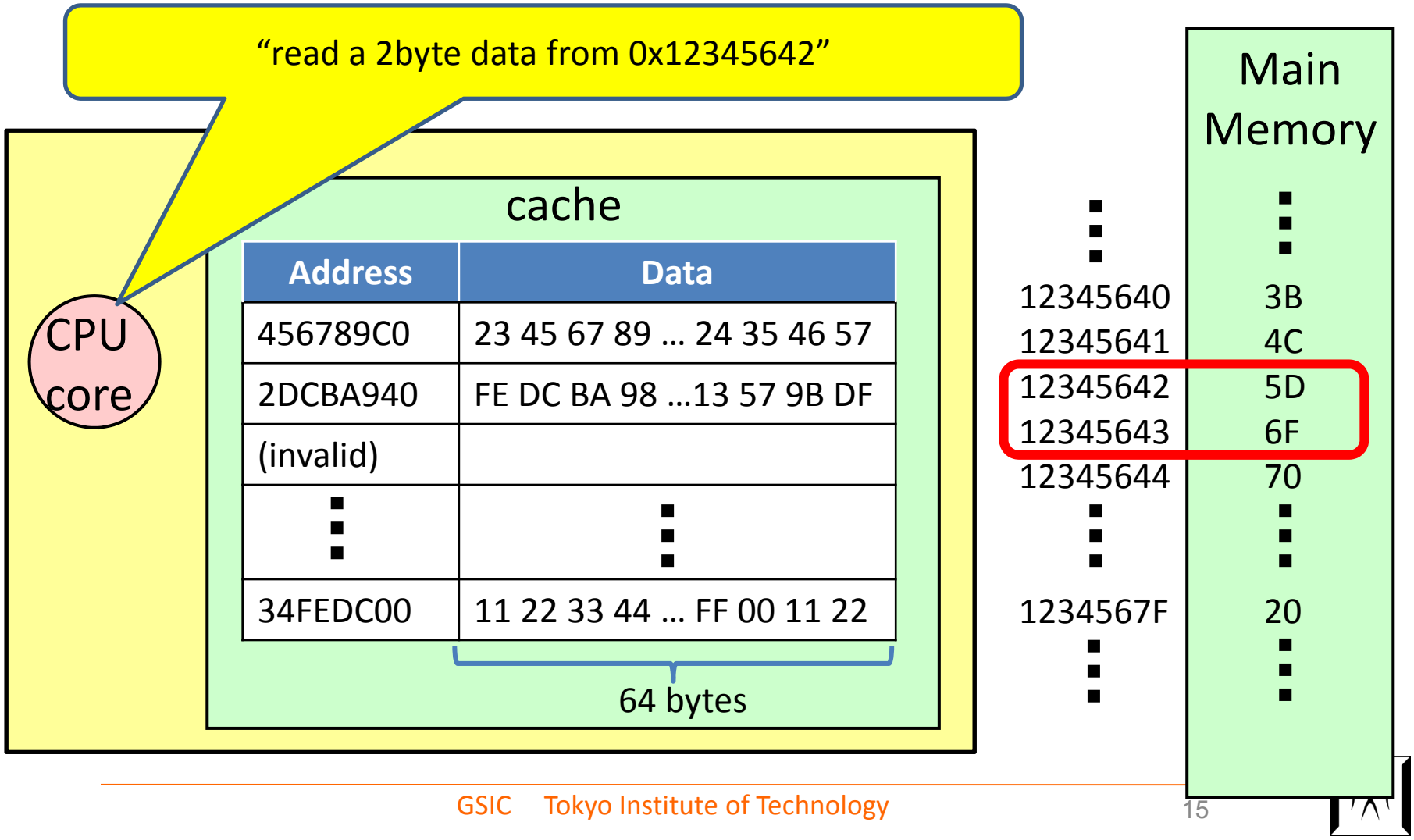
Tagには格納されているデータに対応するメモリアドレスが保持される



メモリアドレスは情報を格納している場所
郵便で例えると　住所で何丁目何番地

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.)

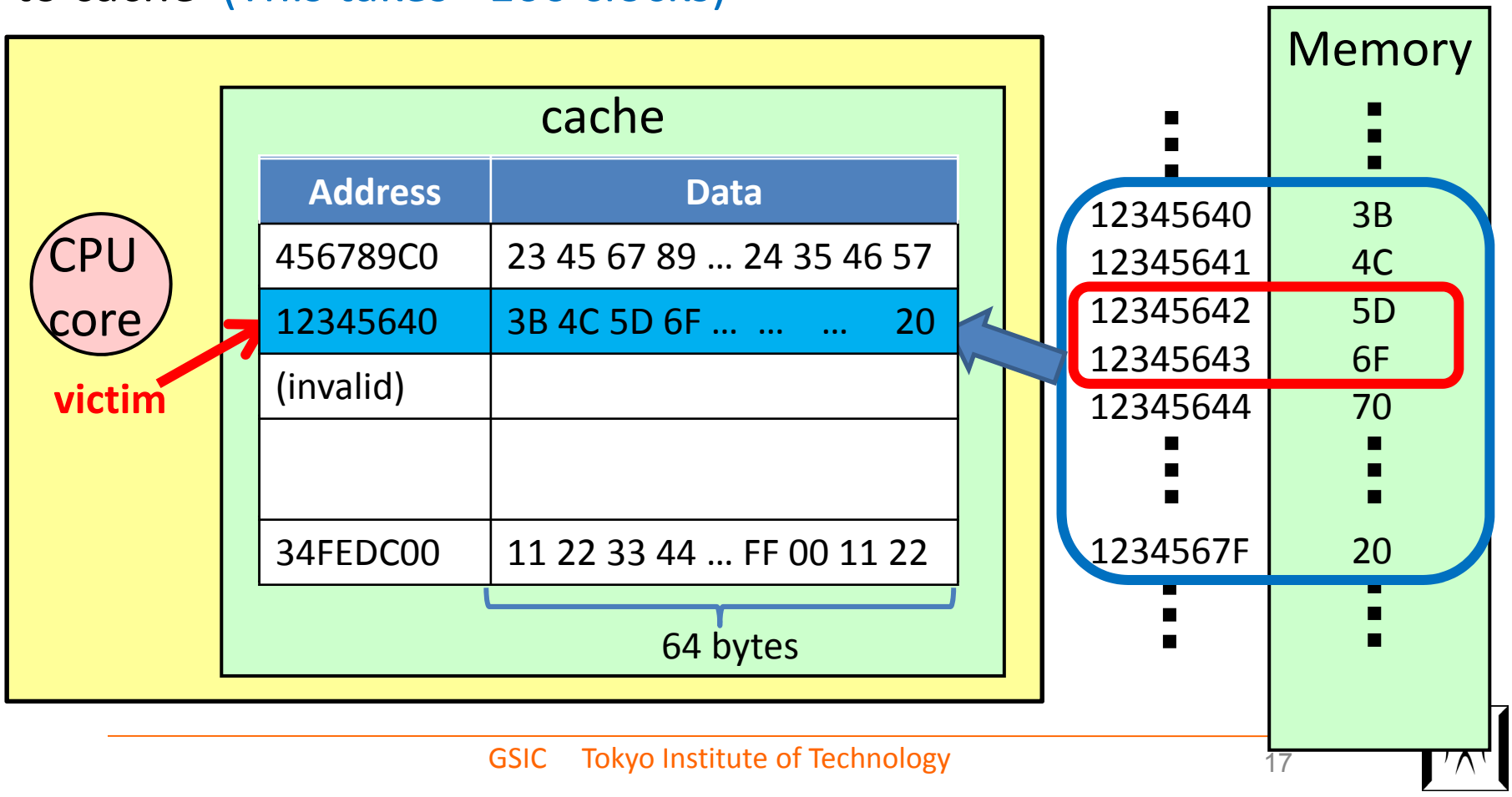
   2-2: If not found, cache miss (This is the case now)

Memory Access with Cache (3)

Cache Miss Case

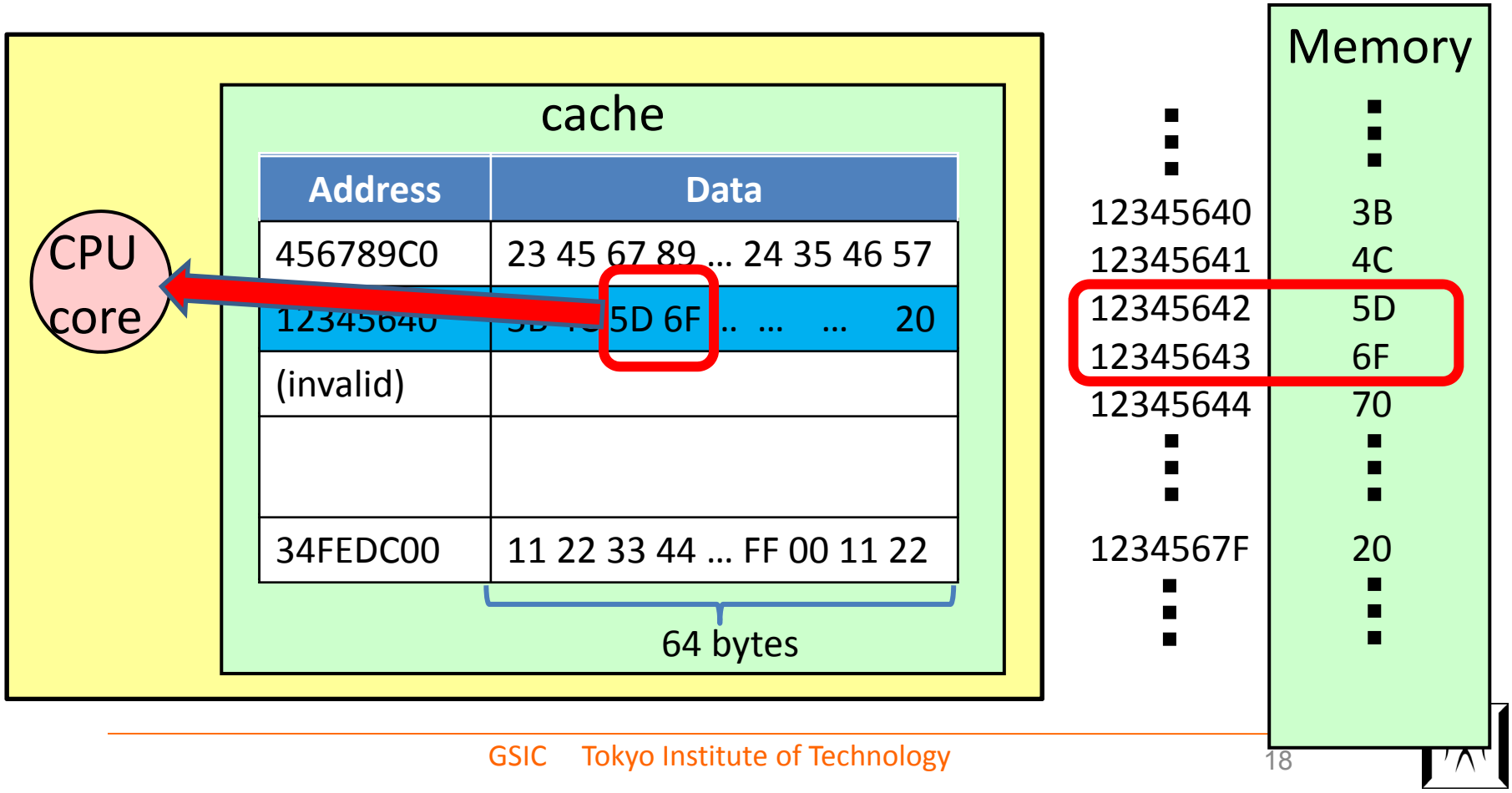
3. Select a “victim” line in cache, to be deleted

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



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

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
  - Full-associative: 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実用化されていない

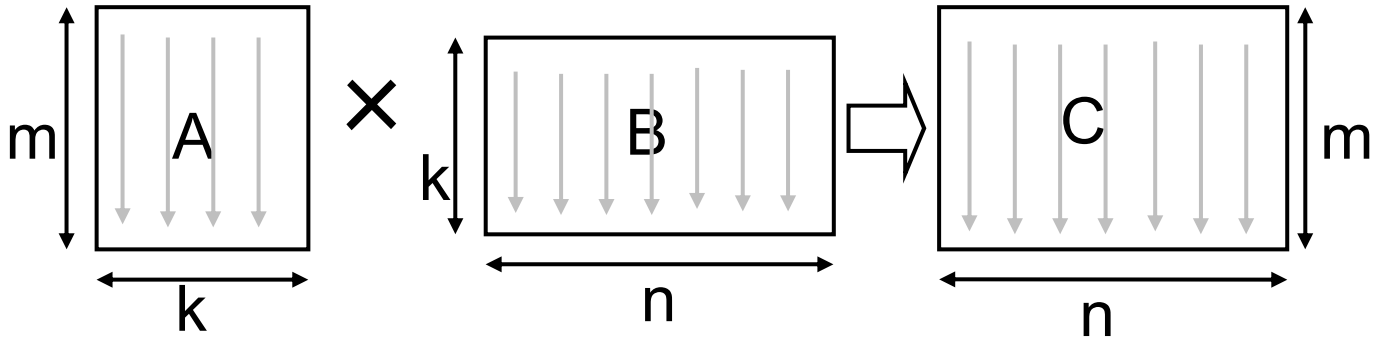
最近のCPUのキャッシュ構成

	京 （8コア）	FX 10 （16コア）	Intel Xeon E5-2670 （8コア）
L1 （I/D別） コア毎	32KB（データ） 2way	32KB（データ）	32KB（データ） 8way
L2	6MB（共有） 12way	12MB（共有） 24way	256KB（コア毎） 8way
L3	None	None	20MB（共有） 20way
cache-line	128B	128B	64B cache line

Set associativeが主流

Matmulでのアクセス順序

6つのmatmulの性能に差が出た理由はアクセスの連続性



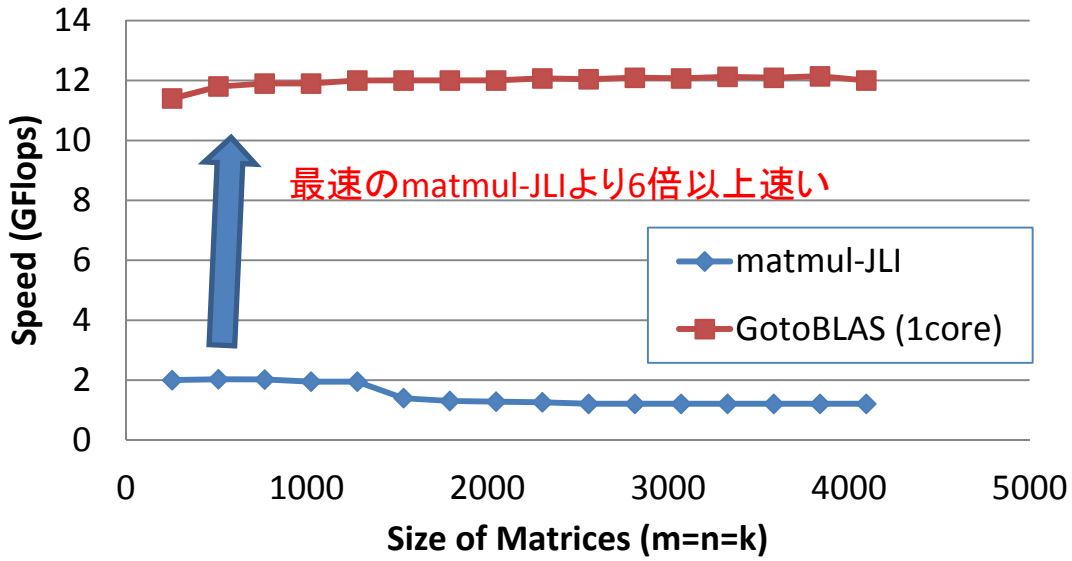

```
for (???) {
  for (???) {
    for (???) {
      Ci,j += Ai,l*Bl,j;
```

- 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

世の中の英知を使う

matmulであればライブラリが使える

- Goto BLAS, Intel MKL, AMD ACML…

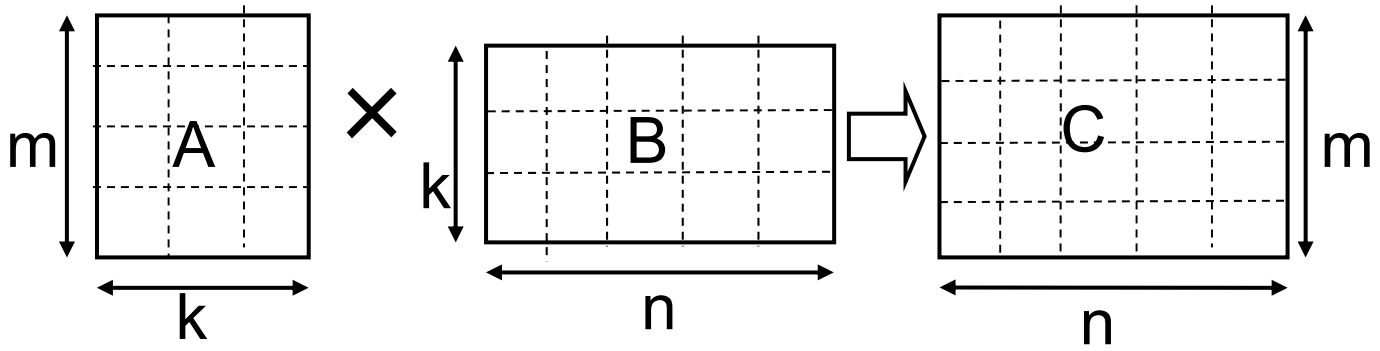


- (Naïve) Simple matmul suffers from more “cache-misses” when problem gets larger
- Optimized GotoBLAS is not only fast, but stable toward the change of problem size

GotoBLASで行われているチューニング

- 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

キャッシュミスを減らしたい

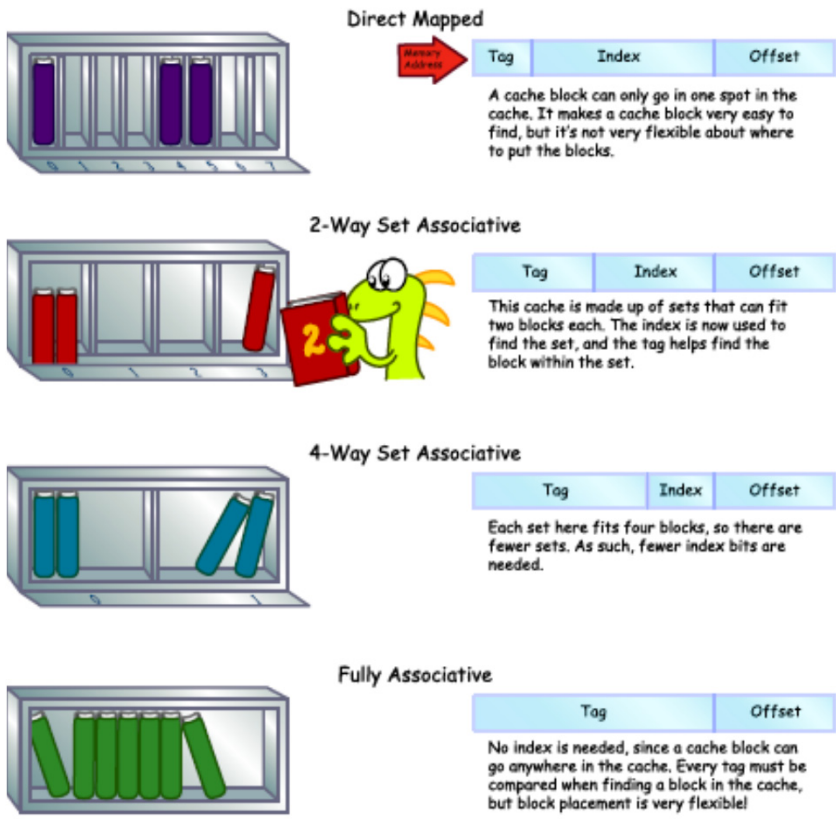
- キャッシュミスの原因　3C
  - Compulsory (Cold), Capacity, Conflict

- 競合ミス(Conflict miss)
  - キャッシュラインの競合によりデータを追い出す
  - 特に、配列アクセスのときキャッシュスラッシングと呼ばれることもある
  - 近年のプロセッサのキャッシュは連想度が高い（L1/L2=8 way, L3=20 way）が未だに発生している

性能チューニングでは如何に競合ミスを減らせるかがポイント

** もちろん連続アクセスか確認必要

キャッシュの連想度 Associativity



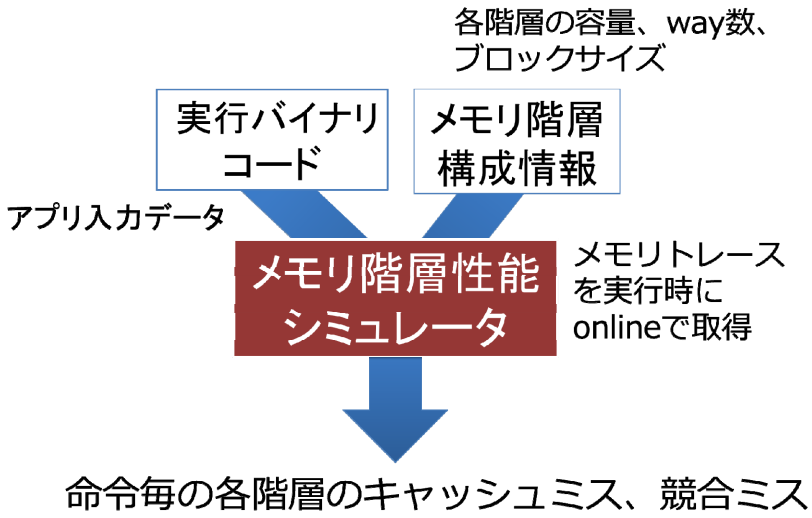
http://csillustrated.berkeley.edu/PDFs/posters/cache-3-associativity-poster.pdf

アドレスより求まるIndexにより格納される場所（ライン）が決定

メモリ階層性能シミュレータの開発

- 容量や連想度が可変

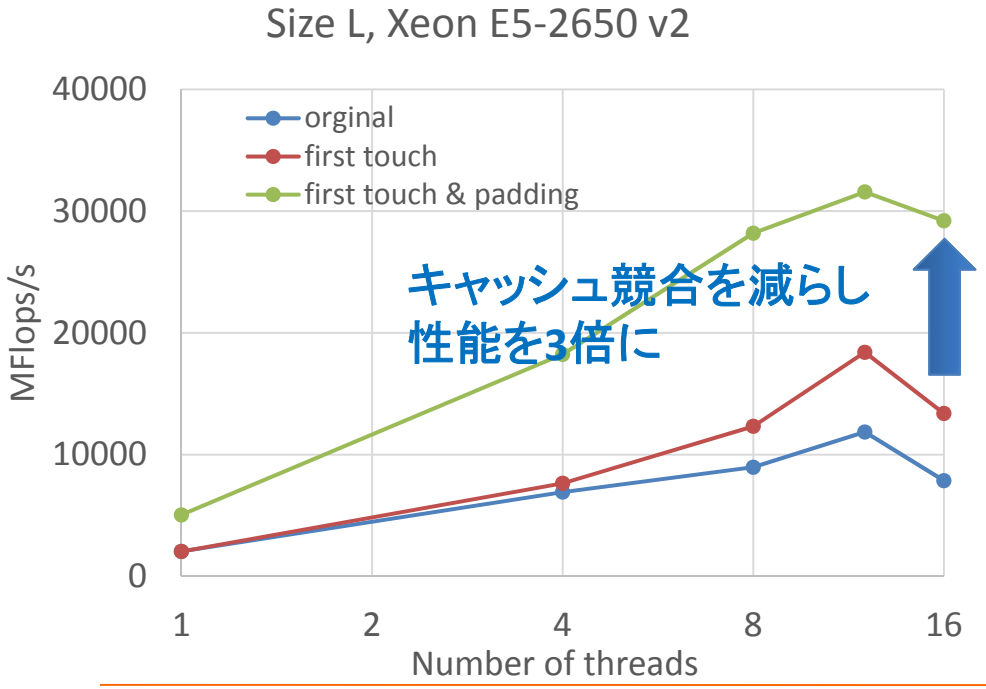
キャッシュパラメータの例
(Intel Xeon　SB世代)
　L1=32kB,8way
　L2=256kB,8way
　L3=10MB,20way
　block size=64B
　ライトスルー方式、
　厳密なLRU、Inclusive Cache



- チューニングを支援するための機能強化
  - 命令毎統計：　命令毎に各階層のCacheのミス率を出力
  - 競合解析：　競合ミス回数を推定
    - キャッシュラインの競合によりデータを追い出す
    - 配列アクセスのときキャッシュスラッシングと呼ばれることもある
    - 近年のCPUキャッシュは連想度が高い（SB世代でL1/L2=8 way, L3=20 way）が未だに発生している
    - LRUから落ちたエントリをFIFO(32)に保持し、リプレースされる際にFIFOに格納されているものと一致した場合を競合とみなす

キャッシュ競合を回避する

- Dynamic allocation, OpenMP 版
  - Exanaのメモリ階層性能シミュレータを利用
  - Original版にファーストタッチ，スレッドのコア固定（pinning）を実施
    - 解析結果よりキャッシュ競合の発生を確認
  - パディングを実施
    - 3次元要素のサイズを +1



Exanaでの解析結果

first touch

Cache miss:

L1	1555798155	34.44 % of Accesses
L2	430429043	27.67 % of Accesses
L3	388956075	90.36 % of Accesses

Conflict miss:

L1	**1100530140**	**70.74 % of Misses**
L2	6156	0.00 % of Misses
L3	155648	0.04 % of Misses

多くのキャッシュ競合が発生していることを観測

first touch & padding

Cache miss:

L1	2381637	22.62 % of Accesses
L2	2379977	99.93 % of Accesses
L3	2379346	99.97 % of Accesses

Conflict miss:

L1	**1292**	**0.05 % of Misses**
L2	631	0.03 % of Misses
L3	155648	6.54 % of Misses

キャッシュ競合が減り，L1 ミス率が改善
最大 3 倍の性能向上

まとめ

- スパコンの性能を向上させるには
  - 単体CPU性能向上＋高並列化（スケーラビリティ）
- 本日は単体CPU性能の話
  - キャッシュをうまくつかえないと性能が低下
  - キャッシュメモリの原理を説明
  - 連続アクセスだとキャッシュにフレンドリー
  - キャッシュ競合の有無で性能が変わることを紹介

遠藤研では

- 性能チューニングを支援するツールの研究開発