

# コンピュータ アーキテクチャ I 第10回

---

塩谷 亮太 (shioya@ci.i.u-tokyo.ac.jp)

東京大学大学院情報理工学系研究科 創造情報学専攻

# 課題の解説

---

# 課題 9

## ■ 以下のような条件を考える

- 10段のパイプラインを持つ 2-way スーパースcalarプロセッサであり, 理想的には  $IPC=2$  で実行できる
- ロード命令の出現率は 0.15
- CPU は L1 キャッシュをもつ
- ロード命令のみが L1 キャッシュやメモリにアクセスするものとする
- L1 キャッシュのヒット時には一切のペナルティなしで実行できる

## ■ (1) 以下の場合の IPC を求めよ

- ロード命令による L1 キャッシュのアクセス 1 回あたりのミス率が 0.02
- ミスの発生時は 100 サイクル追加で時間がかかる

# 一般化できる

- 以下のようにおいた場合,
  - 理想的な実行の際のサイクル数 :  $C_t$
  - 何らかのハザードの発生回数 :  $N_h = N_i \times P_i \times P_h$ 
    - 実行命令数 :  $N_i$
    - ハザードを起こす命令の出現率 :  $P_i$
    - その命令毎のハザード発生率 :  $P_h$
  - ハザード時のサイクル数の増加 :  $C_p$
- 実効サイクル数  $C_r$  は, 理想サイクル数  $C_t$  に対して,
  - $C_r = C_t + N_h \times C_p$

# IPC で考えると

- 最終的な性能を考える上で IPC の方が都合がよい

- 実行サイクル数  $C_r$  を命令数  $N_i$  で正規化すると,

- $$\frac{C_r}{N_i} = \frac{C_t}{N_i} + \frac{(N_m \times C_p)}{N_i} = \frac{C_t}{N_i} + \frac{(N_i \times P_i \times P_h \times C_p)}{N_i} = \frac{C_t}{N_i} + P_i \times P_h \times C_p$$

- IPC は命令数を実行サイクル数で割ったもの = つまり上記の逆数

- $$IPC_r = \frac{1}{\frac{C_t}{N_i} + P_i \times P_h \times C_p} = \frac{1}{\frac{1}{IPC_t} + P_i \times P_h \times C_p}$$

- ここで  $IPC_r$  は実際の IPC,  $IPC_t$  は理想 IPC

## 課題 9 (1)

### ■ (1) の解

- $$\frac{1}{\frac{1}{IPCt} + Pi \times Ph \times Cp} = \frac{1}{\frac{1}{2} + 0.15 \times 0.02 \times 100} = \frac{1}{0.8} = 1.25$$

### ■ ちなみに, キャッシュが全く無い場合 (補足) :

- $$\frac{1}{\frac{1}{IPCt} + Pi \times Ph \times Cp} = \frac{1}{\frac{1}{2} + 0.15 \times 1 \times 100} \approx 0.067$$

# 課題 9

- (2) 以下の場合の IPC を求めよ
  - 1. L1 キャッシュの容量を倍にしたもの
    - L1 キャッシュに良く当たるようになったため、ミス率が 0.01 に
  - 2. L2 キャッシュを追加したもの
    - L2 へのアクセス 1 回あたりのミス率は 0.1
      - ◇ L2 は L1 にミスしたときのみアクセスするものとする
    - L2 ヒット時は L1 ヒット時からの実行時間の追加が 15 サイクル
    - L2 ミス時は L1 ヒット時からの実行時間の追加が 150 サイクル
- (3) これまでに出てきた 3 つのモデルの性能を求め比較せよ
  - ただし L1 キャッシュ容量を倍にした場合、キャッシュのアクセスに時間がかかるため、周波数が 0.8 倍になるものとする

## 課題 9 (2)

### ■ 容量が倍

- $$\frac{1}{\frac{1}{IPCt} + Pi \times Ph \times Cp} = \frac{1}{\frac{1}{2} + 0.15 \times 0.01 \times 100} \approx 1.53$$

### ■ L2 の追加

- L1 ヒット : ペナルティなし
- L1 ミス & L2 ヒット :  $Ph1 = 0.02 \times (1 - 0.1) = 0.018, Cp1 = 15$
- L1 ミス & L2 ミス :  $Ph2 = 0.02 \times 0.1 = 0.002, Cp2 = 150$
- $$\frac{1}{\frac{1}{IPCt} + Pi \times Ph1 \times Cp1 + Pi \times Ph2 \times Cp2} = \frac{1}{\frac{1}{2} + 0.15 \times 0.018 \times 15 + 0.15 \times 0.002 \times 150} \approx 1.71$$



## 課題 9 (3)

- すいません, これは問題が良くなかったです
  - CPU の周波数が変わってもメモリの速度は変わらない
  - メモリのアクセスのサイクル数が変わってしまうので, IPC が変化する
- 例 : 周波数 1G Hz の場合
  - 1 サイクル=1ns
  - 100サイクルかかるメモリアクセスは100nsになる
  - 周波数が 0.8 倍になると, 1サイクルは $1/0.8\text{GHz}=1.25\text{ns}$
  - $100\text{ns}/1.25=80$ サイクル
    - メモリアクセス時間は変わらない

## 課題 9 (3)

### ■ 容量が倍の際の IPC を再計算

- $$\frac{1}{\frac{1}{IPC_t} + P_i \times P_h \times C_p} = \frac{1}{\frac{1}{2} + 0.15 \times 0.01 \times 100 \times 0.8} \approx 1.61$$

### ■ 性能：

- もともと： $1.25 \times 1 = 1.25$
- L1容量倍： $1.61 \times 0.8 \approx 1.28$
- L2追加： $1.71 \times 1 \approx 1.71$

# キャッシュの詳細

---

# 内容

1. キャッシュの構成方法
2. 行列積での動作例

# キャッシュの構成方法

---

# キャッシュの構成方法

1. 3つの方式：
  1. 基本的な構造（フルアソシアティブ方式）
  2. ダイレクトマップ方式
  3. セット・アソシアティブ方式
2. ライン単位での管理
3. アドレスとキャッシュ構造の具体的な対応関係

# キャッシュの作り方の方針

## ■ キャッシュ：

- 小容量で高速なメモリ
- メイン・メモリの一部をコピーして保持
  - こっちを略してメモリということも

## ■ 目的のデータがコピーされているかどうかを確認したい

- コピー時に、どここのデータをコピーしたかの情報も一緒に記録
  - つまり、コピー元のアドレスもキャッシュに記録する
- キャッシュの読み書き時は、記録されているアドレスとの突き合わせをして確認する

# キャッシュの基本的な構造

アドレスやデータは 16 進数

アドレス データ

0000	84
0001	ff
0002	12
⋮	⋮
8000	33
8001	55

メモリ

タグ データ

0002	12
8001	55

エントリ

容量が 2 エントリのキャッシュ

## ■ キャッシュのエントリの内容

- タグ： コピーしてきたデータが、メモリのどこのアドレスにあったかを表す  
(後で詳しく話すように本当はアドレスの一部が入る)

- データ： その内容

## ■ コピー時に元のアドレスと一緒に格納する

- 上記の例：  
0002 にあった 12 と、8001 にあった 55 を保持



# 読み出し時の動作

アドレスやデータは 16 進数

アドレス	データ
0000	84
0001	ff
0002	12
⋮	⋮
8000	33
8001	55
	メモリ

## タグ データ

0002	12
8001	55

容量 2 のキャッシュ

1. まず全てのタグを読み出す（この場合 2 つ）
  2. アドレスと一致するタグがあるかをチェック
    1. ヒット：もしあれば，そのデータを読む
    2. ミス： なければ，メモリにアクセス
- たとえば CPU がアドレス 8001 を読むと，タグに 8001 があるのでヒット

# フルアソシアティブ方式とその問題

タグ データ

0002	12
8000	33

容量2のキャッシュ  
2つのタグをチェック

タグ データ

0002	12
8000	33
0102	00
5511	78

容量4のキャッシュ  
4つのタグをチェック

- 先ほどの方式をフルアソシアティブ方式と呼ぶ
  - キャッシュ内の全てのタグをチェックする方式
- 問題：
  - 格納データ数を増やすと、比例して比較するタグの数が増える
  - 比較のための回路は複雑で遅いし、電気もバカ食いする

# ダイレクトマッピング方式

アドレスやデータは 16 進数

	タグ	データ
0	800 <b>0</b>	12
1	100 <b>1</b>	33
2	010 <b>2</b>	00
3	550 <b>3</b>	78

- 全てのエントリではなく、アドレス毎に特定の1つのエントリのみを使う
  - 比較が1つのみでよくなる
- 「アドレス mod サイズ」の番号のエントリにアクセス  
(mod は剰余, 数字は16進数表記)
  - アドレス 8000 :  $8000 \bmod 4 = 0$  番エントリにアクセス
  - アドレス 5513 :  $5513 \bmod 4 = 3$  番エントリにアクセス

# ダイレクトマップ方式

アドレスやデータは 16 進数

	タグ	データ
0	800 <b>0</b>	12
1	100 <b>1</b>	33
2	010 <b>2</b>	00
3	550 <b>3</b>	78

## ■ フルアソシアティブとの違い：

- 利点：チェックするタグは常に 1 つですむ
- 問題：アドレス下位がかぶると（競合とよぶ），上書きされる
  - 800**0**, 700**0**, 010**0** の順にアクセスがあると，0 番しか使えない
  - $(= \text{mod } 4)$  の結果が全部 0

# セットアソシアティブ方式

アドレスやデータは 16 進数

	タグ	データ	タグ	データ
0	800 <b>0</b>	12	010 <b>0</b>	53
1	100 <b>1</b>	33	770 <b>1</b>	44
2	010 <b>2</b>	00	510 <b>2</b>	22
3	551 <b>3</b>	78	050 <b>3</b>	87

- 「アドレス mod サイズ」の**セット**にアクセス
  - 上の例の場合, 1つのセット内に2つのタグ+データがある
- 連想度 :
  - セットの中にいくつ要素を入れるかのこと
  - 上記の場合連想度は2 (2-way と呼ぶ)
- 利点 : 競合するデータを複数持てる
  - **キャッシュに必要なデータが在る率 (ヒット率)** 上がる

# セットアソシアティブ方式の動作

	タグ	データ	タグ	データ
0	800 <b>0</b>	12	010 <b>0</b>	53
1	100 <b>1</b>	33	770 <b>1</b>	44
2	010 <b>2</b>	00	510 <b>2</b>	22
3	551 <b>3</b>	78	050 <b>3</b>	87

## ■ アドレス 0100 にアクセスがあった場合：

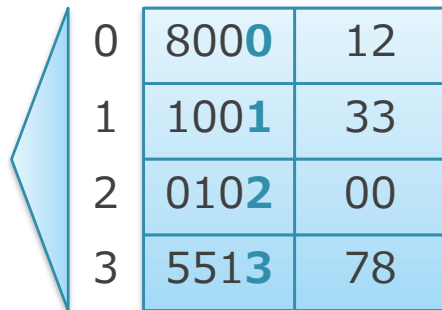
- 「 $0100 \bmod 4 = 0$ 」よりセット 0 のタグを全て読んで、これと比較
- 右側のタグ 0100 がヒットしたので、ここを読み出す

## ■ どこにもヒットしなかった場合

- メモリからデータを取ってきて、キャッシュに書き込む
- 同一セット内で最も長時間アクセスされていないエントリに書き込むことが一般的

# 容量一定 (= 4) にして連想度を変えた場合

連想度 1  
(=ダイレクトマップ)




0	8000	12
1	1001	33
2	0102	00
3	5513	78

連想度2



0	8000	12	0100	53
1	1001	33	7701	44

連想度4  
(=フルアソシアティブ)




0	8000	12	0100	53	7000	12	0500	53
---	------	----	------	----	------	----	------	----

- 容量 = 連想度 × セット数
- 各方式との関係：
  - ダイレクトマップ： 連想度 = 1 のとき
  - フルアソシアティブ： 連想度 = 容量 のとき

# 競合と複雑さのトレードオフ

連想度 1  
(=ダイレクトマップ)



0	8000	12
1	1001	33
2	0102	00
3	5513	78

連想度2



0	8000	12	0100	53
1	1001	33	7701	44

連想度4  
(=フルアソシアティブ)



0	8000	12	0100	53	7000	12	0500	53
---	------	----	------	----	------	----	------	----

## ■ 容量一定の場合のトレードオフ

- 連想度大：競合の影響が小さいが、回路が複雑
- 連想度小：競合の影響が大きいが、回路が簡単

## ■ 現実的には、連想度 2 から 32 ぐらいまでが良く使われる



# 各方式のまとめ

## ■ キャッシュ

- 小容量で高速
- メモリの一部をアドレス（タグ）と共にコピー

## ■ 方式

- ダイレクトマップ
- セットアソシアティブ
- フルアソシアティブ

## ■ 性質

- 連想度によって分類可能
- ヒット率と複雑さにトレードオフ

# キャッシュの詳細

## 1. 方式：

- 基本的な構造
- ダイレクトマップ方式
- セット・アソシアティブ方式

## 2. ライン単位での管理

## 3. アドレスとキャッシュ構造の対応

# ライン

- キャッシュ上のデータはラインと呼ばれる単位で管理される
  - ライン：複数バイトからなる塊
  - 実際には 16 から 128バイトぐらい
  - ブロックと呼ばれることもある
- 理由：
  1. 容量の効率をあげるため
  2. 空間局所性を利用するため

# 容量の効率

タグ                      データ  
(32bit=4バイト)      (1バイト)

f3568000	12
----------	----

## ■ タグが大きくて無駄

- これまでの説明では、アドレスごとに1バイトのデータを仮定
- 一方、アドレスは 32 から 64 ビット
- このままではデータ本体よりも、アドレス識別のためのタグを覚えているようなものになってしまう

# 容量効率の向上

## ■ ライン

- タグが指すアドレスから始まるデータのまとまりのこと
- キャッシュの各エントリでは、このライン単位でデータを持つ

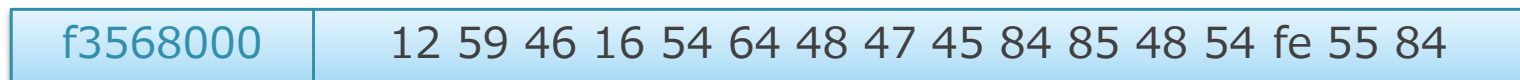
## ■ 利点：ラインサイズが増えると、データが占める割合が増える

- 1 バイト：  $1 \times 4 = 4$  バイト
- 16 バイト： 16 バイト
- (双方、タグとデータ合計で20バイト)

タグ                      データ  
(32bit=4バイト)      (1バイト)



タグ                      ライン (16バイト)  
(32bit=4バイト)



# 空間的局所性

## ■ 2種類の局所性

### 1. 時間的局所性：

- 「一度使ったデータは、**すぐに**また使われる」

### 2. **空間的局所性**：

- 「あるデータが使われると、**その近くにある**データも使われる」

## ■ 空間的局所性の例、

- 以下では  $i$  番目がアクセスされると  $i+1$  にもアクセスされる

```
for(i = 0; i < SIZE; i++)
```

```
    v += buf[i]
```

- ある構造体内の要素にアクセスがあると、その構造体の別の要素にもアクセスがある

# ライン単位の管理と空間局所性

- データはライン単位でやりとりされる
  - あるデータがアクセスされると、周囲のデータも一緒にキャッシュに格納される
- たとえば、ラインが 16 バイトだった場合
  - 各要素は1バイトで16要素の配列 `buf[16]` を考える
  - `buf[0]` のアクセス時に、`buf[1] ~ buf[15]` までをまとめて読む
    - まとめてメモリから取ってきてキャッシュにおく
  - `buf[1]` から `buf[15]` アクセス時は、キャッシュにヒット

# キャッシュの詳細

1. 方式：
  - 基本的な構造
  - ダイレクトマップ方式
  - セット・アソシアティブ方式
2. ライン単位での管理
3. アドレスとキャッシュ構造の対応



# キャッシュ内のデータの配置

- 以下に要素に関連して変化
  - 連想度
  - 容量
  - ラインのサイズ
- プログラムの高速化のためには、以下を知る必要がある
  - アドレスとキャッシュ内のラインの位置の対応
  - 結果、どのようにアクセスするとキャッシュにヒットするのか
- さらに後半ではいくつかの実例をつかって説明

# セットアソシアティブ・キャッシュの例



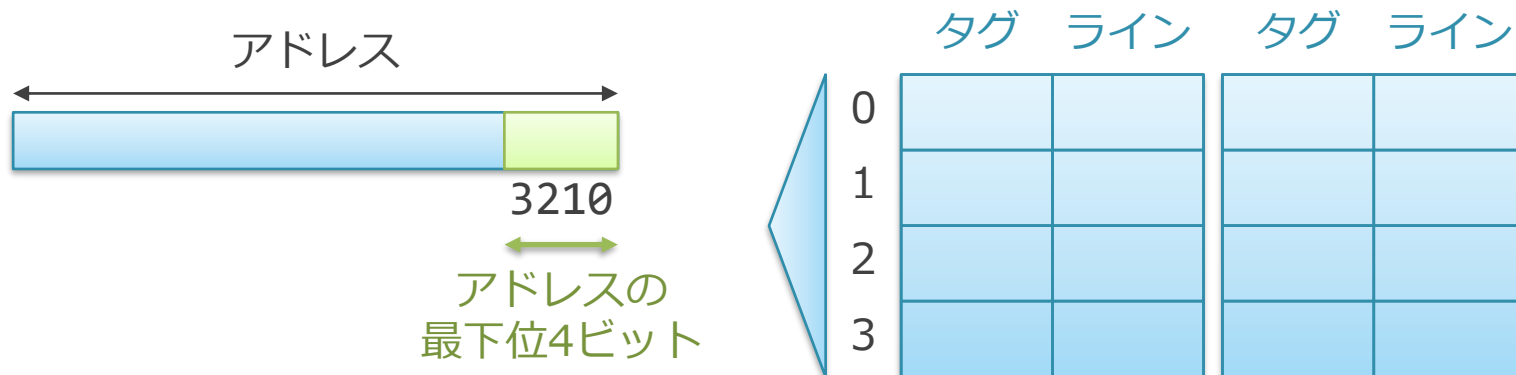
## ■ 構成

- 連想度 : 2
- セット数 : 4
- ラインサイズ : 16バイト

## ■ 総記憶容量

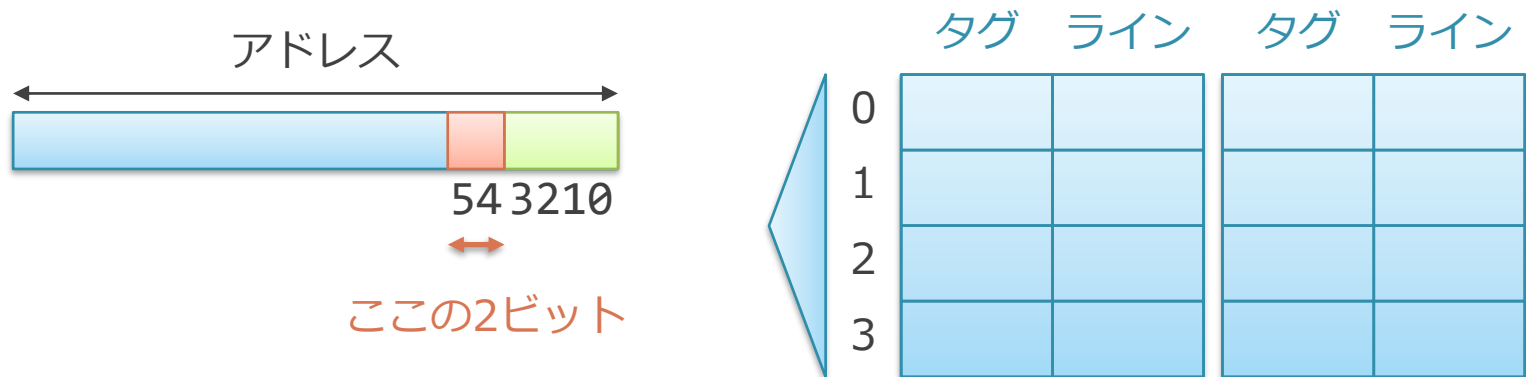
- 連想度 2 × セット数 4 × ライン 16 バイト = 128 バイト

# アドレスとラインの対応



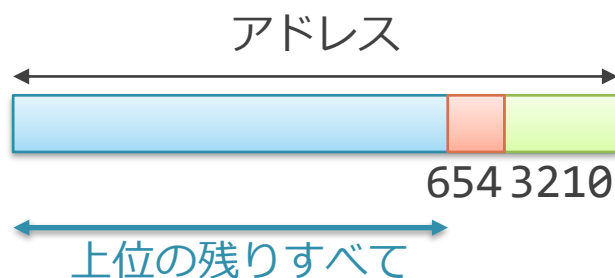
- アドレスは1バイト単位でメモリの位置を表すものとする
- 最下位ビット 0 ～ 3 （計 4 ビット）
  - 最下位部分がライン内の位置に対応
    - 空間局所性を利用するために連続した 16 バイトが 1 ラインに
  - 4ビットなのは、ラインサイズが16バイトだから
    - $2^4 = 16$
    - (ラインサイズは必ず 2 の累乗になる)

# アドレスとセットの対応



- ライン部分の上位にあるビット 4 ~ 5 （計2ビット）
  - この部分を使って、どのセットにアクセスするか決める
  - 2ビットなのは、セット数が4だから
    - $2^2 = 4$
  - セット数も必ず2の累乗になる
- アドレスのこの部分はなるべくばらけた方がよい
  - 同じセットにアクセスがいかず、競合がおきにくくなる

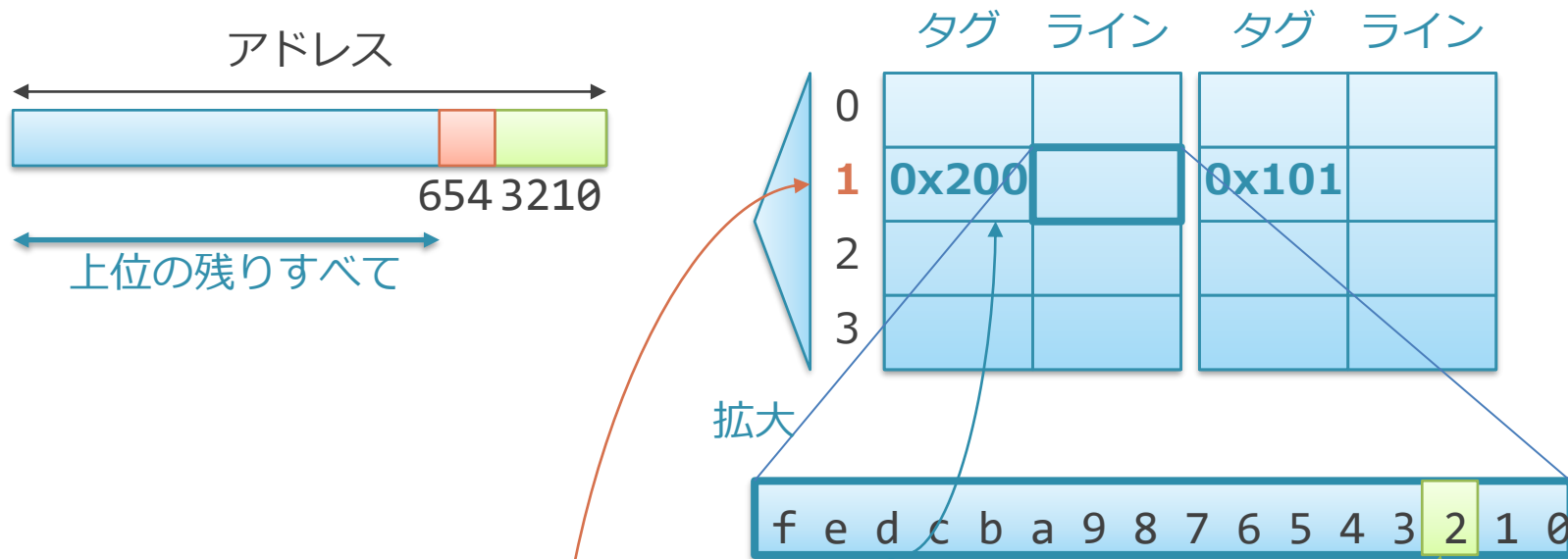
# アドレスとタグの対応



	タグ	ライン	タグ	ライン
0				
1				
2				
3				

- 残りの上位のビットがタグとなる
- タグにはセット（赤）やライン（緑）の部分は入れないでよい
  - あるセットにアクセスするアドレスは、赤部分は常に一定だから
    - セット 1 にアクセスする場合、赤部分は絶対 01
  - 緑部分はラインの中の位置を表すので、関係ない

# アクセス時の動作の例



- アドレス0x8014 (1000 0000 0001 0010) へのアクセスがあった場合
  - ライン内位置 : 2 (0010)
  - セット位置 : 1 (01)
  - タグ : 0x200 (1000 0000 00)
- セット1の左側のエントリにタグ 0x200 があるのでヒット
  - ライン内の2バイト目にアクセス

# キャッシュの詳細のまとめ

- 基本的な構造と各方式について
  - セット・アソシアティブ方式
  - ライン単位での管理
- アドレスとキャッシュ構造の具体的な対応関係

# 行列積での動作例

---



# 内容

1. キャッシュの構成方法
- 2. 行列積での動作例**

# キャッシュによる性能変化の例：密行列積

## ■ 密行列積

- ディープ・ラーニングも、実際の計算はひたすら行列積をやっている事が多い
- google の TPU は行列積 超特化計算機ともいえる
  - TPU: Tensor Processing Unit
  - 機械学習に特化したハードウェア

## ■ 行列積はものすごい時間がかかる

- 行列のサイズの三乗に比例して演算が必要
- なんも考えないとキャッシュにもうまく乗らない

# 目次

1. 背景：
  1. 行列の二次元配列による表現
  2. 二次元配列のメモリ配置
2. 行列同士の乗算

# 行列の2次元配列による表現

```
uint32_t A[2][2];
```

$$\begin{bmatrix} A[0][0], A[0][1] \\ A[1][0], A[1][1] \end{bmatrix}$$

■  $A[y][x]$  の場合：

- 1次元目 ( $x$ ) : 何列目か
  - $x$  が増えると参照位置が右に移動
- 2次元目 ( $y$ ) : 何行目か
  - $y$  が増えると参照位置が下に移動

# 2次元配列のメモリ上の配置

アドレス (uint32\_t は 32bit=4バイトなので, 4飛ばしになる)

0	A[0][0]
4	A[0][1]
8	A[0][2]
	⋮
124	A[0][31]
128	A[1][0]
132	A[1][1]
	⋮
254	A[1][31]
256	A[2][0]
	⋮

```
uint32_t A[32][32];
```

■ 実際のメモリは1次元の構造

- ずっと連続して箱が並んでる

■ 低次元 (添え字の右側) が連続するように展開されて配置される

# キャッシュ上の配置 (ラインサイズ64バイトの場合)

アドレス

0	A[0][0]
4	A[0][1]
8	A[0][2]
	⋮
124	A[0][31]
128	A[1][0]
132	A[1][1]
	⋮
254	A[1][31]
256	A[2][0]
	⋮

uint32\_t A[32][32];

キャッシュ

タグ    ライン

0	A[0][0], A[0][1], ... A[0][15]
64	A[0][16], A[0][17], ... A[0][31]
128	A[1][0], A[1][1], ... A[1][15]
160	A[1][16], A[1][17], ... A[1][31]
192	A[2][0], A[2][1], ... A[2][15]
...	

- 1次元目の添え字が連続した部分がライン上に
  - ラインは64Bなので, 16要素格納できる
  - 1次元目を連続にして参照すると効率がよい

# 配列のアクセス

- 2次元目を連続させた場合の問題
  1. ラインの利用効率が悪い
  2. コンフリクトが起きる

# 2次元目を連続させた場合の動作

アドレス

0	A[0][0]
4	A[0][1]
8	A[0][2]
	⋮
124	A[0][31]
128	A[1][0]
132	A[1][1]
	⋮
254	A[1][31]
256	A[2][0]
	⋮

タグ    ライン

0	<b>A[0][0]</b> , A[0][1], ... A[0][15]
64	A[0][16], A[0][17], ... A[0][31]
128	<b>A[1][0]</b> , A[1][1], ... A[1][15]
192	A[1][16], A[1][17], ... A[1][31]
256	<b>A[2][0]</b> , A[2][1], ... A[2][15]
...	

```
for (int j = 0; j < SIZE; j++)  
    A[j][0]++;
```

- 2次元目を連続にしてアクセスした場合
  - 赤字の部分がアクセスされる



## 2次元目を連続させた場合の問題（1）

タグ    ライン

0	<b>A[0][0]</b> , A[0][1], ... A[0][15]
64	A[0][16], A[0][17], ... A[0][31]
128	<b>A[1][0]</b> , A[1][1], ... A[1][15]
192	A[1][16], A[1][17], ... A[1][31]
256	<b>A[2][0]</b> , A[2][1], ... A[2][15]
...	

```
for (int j = 0; j < SIZE; j++)  
    A[j][0]++;
```

### ■ 問題 1 :

- **ラインの先頭しか使われない**
- A[0][1] から A[0][15] もキャッシュに勝手に乗るが使われない

# 2次元目を連続させた場合の問題（1）

タグ    ライン

0	<b>A[0][0]</b> , A[0][1], ... A[0][15]
64	A[0][16], A[0][17], ... A[0][31]
128	<b>A[1][0]</b> , A[1][1], ... A[1][15]
192	A[1][16], A[1][17], ... A[1][31]
256	<b>A[2][0]</b> , A[2][1], ... A[2][15]
...	

for (int j = 0; j < SIZE; j++)

**A[j][0]**++;

## ■ 問題 2

- **アドレスが等間隔になる**

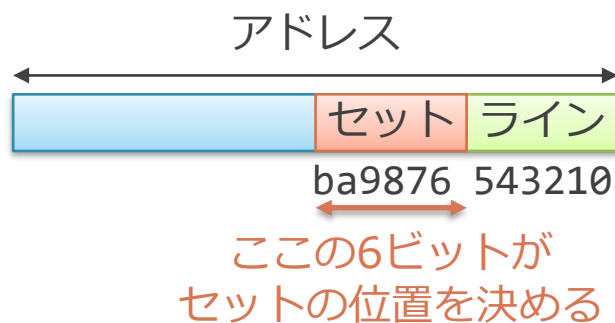
（ので、後述の様にキャッシュの使用効率が著しく落ちる）

- 0, 128, 256 ...

- 間隔は、配列の1次元目のサイズに比例

- 今回は32要素×4 = 128 が間隔に

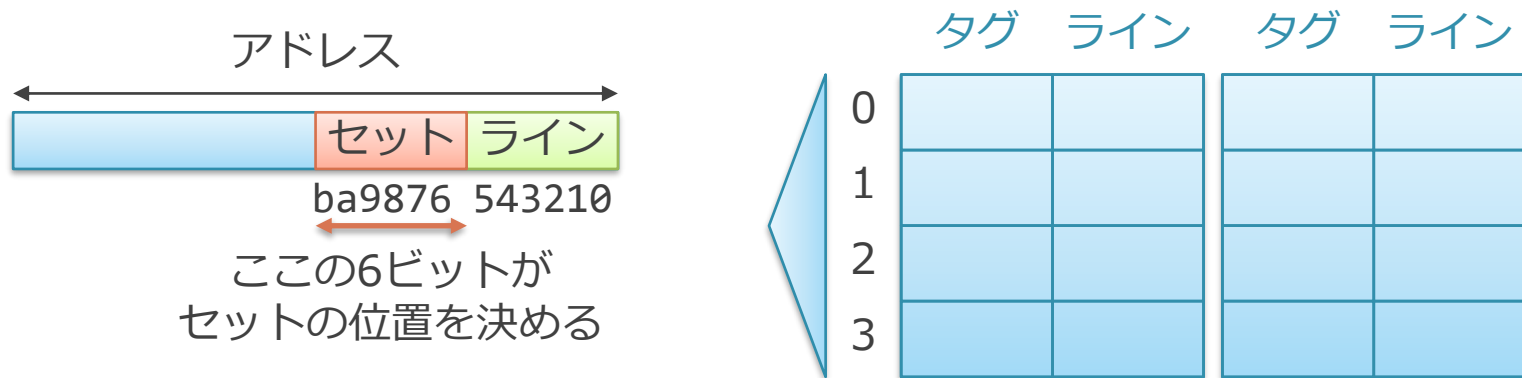
# アドレスとセットの対応の復習



	タグ	ライン	タグ	ライン
0				
1				
2				
3				

- ライン部分の上位にあるビット 6 ~ b（計6ビット）
  - この部分を使って、どのセットにアクセスするか決める
- L1キャッシュのセット数部分は6ビットある
  - 32KB, 64バイトライン, 8-way
  - $32768 / 64 / 8 = 64 = 2^6$

# 大きな二次元配列で、2次元目を連続にすると

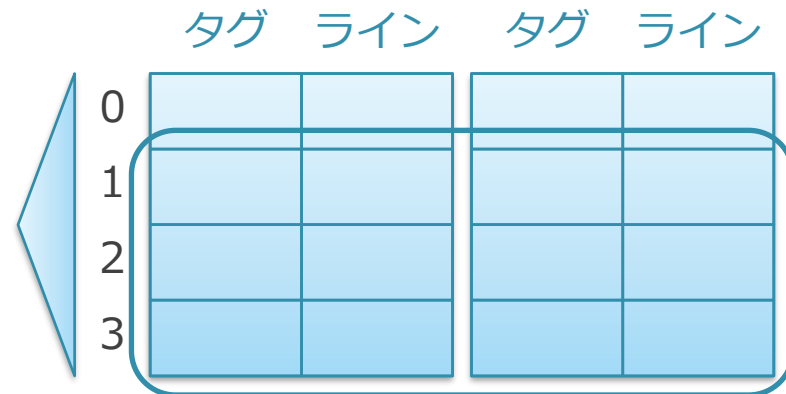
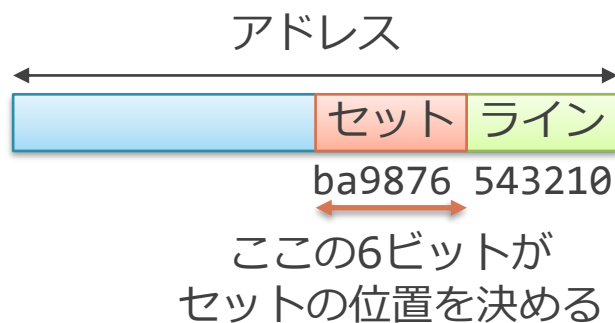


```
uint32_t A[1024][1024];  
for (int j = 0; j < SIZE; j++)  
    A[j][0]++;
```

## ■ アドレス：

- $A[0][0]$ : 0,
- $A[1][0]$ : 4096
- $A[2][0]$ : 8192
- $1024\text{要素} \times 4\text{B} = 4096 = 2^{12}$  ごとにアクセス

# アドレスが等間隔になるとどうなるか



```
uint32_t A[1024][1024];  
for (int j = 0; j < SIZE; j++)  
    A[j][0]++;
```

ここは使われなくなってしまう

## ■ 何がまずいのか：セット位置を決める部分が全部一定に

- 0: 0000000000000000
- 4096: 0100000000000000
- 8192: 1000000000000000

## ■ 大きな二次元配列で二次元目を連続にすると、連想度分ぐらいしかキャッシュできない

- 連想度=2 だと、2 ラインしかキャッシュが使われない

# 行列と二次元配列のまとめ

## ■ 構造

- 行列は二次元配列として表限
- 二次元配列は, 1次元目が連続するよう展開される

## ■ 二次元目を連続させるとやばい

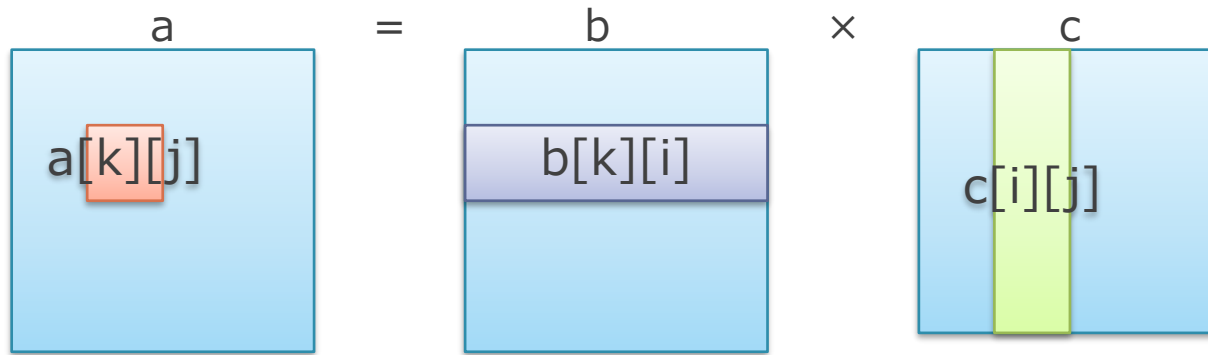
- ラインの利用効率が悪い
- 大きな二次元配列ではアドレスが等間隔に
  - コンフリクトが起きてキャッシュがほとんど利用できない

# 基本的な行列積の実装

```
for (int k = 0; k < SIZE; k++) {  
    for (int j = 0; j < SIZE; j++) {  
        for (int i = 0; i < SIZE; i++) {  
            a[k][j] += b[k][i] * c[i][j];  
        }  
    }  
}
```

- 三重ループとして実現できる

# 行列積の動作イメージ



```
for (int k = 0; k < SIZE; k++) {  
    for (int j = 0; j < SIZE; j++) {  
        for (int i = 0; i < SIZE; i++) {  
            a[k][j] += b[k][i] * c[i][j];  
        }  
    }  
}
```

- $a[k][j]$  は,  $b$  の  $k$  行目 (紫) と,  $c$  の  $j$  列目 (緑) の各要素を乗算して累積することにより求まる
  - 一番内側の  $i$  はこの各要素を参照するために回る



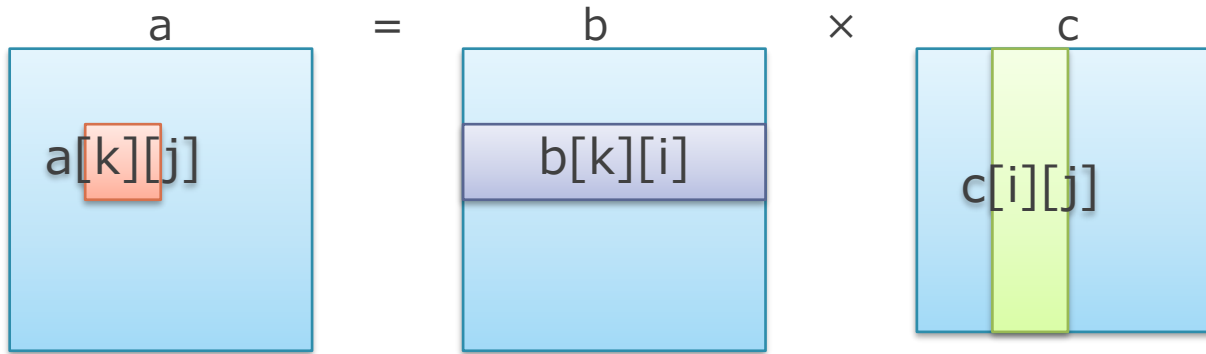
# 重要なポイント

```
for (int k = 0; k < SIZE; k++) {  
    for (int j = 0; j < SIZE; j++) {  
        for (int i = 0; i < SIZE; i++) {  
            a[k][j] += b[k][i] * c[i][j];  
        }  
    }  
}
```

## ■ このプログラムをよく見ると,

- `a[k][j] +=` の部分の計算の順序は自由に入れ替え可能
  - 足し算はどのような順序でやってもよい
  - たとえば, ループの外側と内側を入れ替えても, 結果は同じ

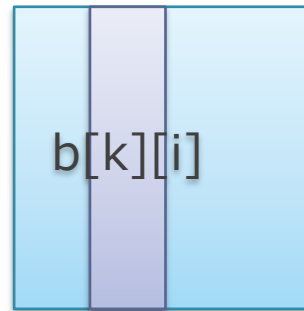
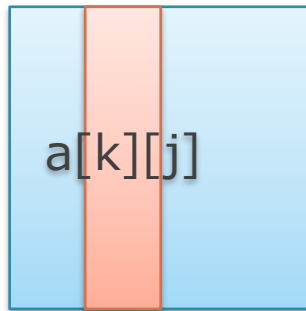
# i, j, k をひっくり返した時の、 最内周ループのアクセス範囲



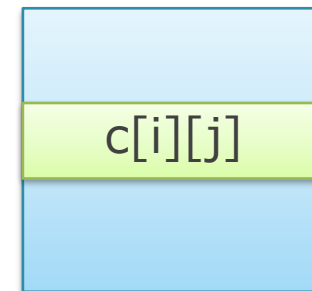
```
for (int k = 0; k < SIZE; k++) {  
    for (int j = 0; j < SIZE; j++) {  
        for (int i = 0; i < SIZE; i++) { // i が変化
```

- 最内周ループのアクセス範囲が横向きになっているのが重要

# 最悪の場合（1100秒）と最良の場合（20秒） 上側はキャッシュを全く利用できていない



```
for (int i = 0; i < SIZE; i++) {  
    for (int j = 0; j < SIZE; j++) {  
        for (int k = 0; k < SIZE; k++) { // k が変化
```



```
    for (int k = 0; k < SIZE; k++) {  
        for (int j = 0; j < SIZE; j++) { // j が変化
```

# まとめ

## ■ キャッシュの構成方法

- 3つの方式
  - 基本的な構造（フルアソシアティブ方式）
  - ダイレクトマップ方式
  - セット・アソシアティブ方式
- 性質
  - 連想度によって分類可能
  - ヒット率と複雑さにトレードオフ
- ライン単位での管理
- アドレスとキャッシュ構造の具体的な対応関係

## ■ 行列積での動作例

# 課題 10

- アドレスの幅が 16 bit, ラインサイズ8B, 4 エントリのキャッシュについて考える
- 連想度を以下の様に変えた場合に,
  - 1 (ダイレクトマップ)
  - 2
  - 4 (フルアソシアティブ)
- 以下のようなアドレスによる 1B のアクセスがあった場合を考える
  1. 0x8000, 0x8001, 0x8002, 0x8003, 0x8000, 0x8001, 0x8002, 0x8003
  2. 0x8000, 0x9000, 0xA000, 0xB000, 0x8000, 0x9000, 0xA000, 0xB000
  3. 0x8000, 0x9001, 0x8002, 0x9003, 0x9004, 0xA005, 0x9006, 0x8007

# 課題 10

- (1) 上記それぞれの場合で、アクセスが全て終わった後のキャッシュの状態（タグの中身）を示せ
  - 4 エントリのタグにそれぞれ何が残っているかを、  
連想度3パターン×アクセス系列3パタン= 9 パターン分答える
- (2) 上記それぞれの場合のヒット率を計算せよ
- (3) 各アクセスにおけるヒット時に、それが空間的局所性と時間的局所性のいずれによるのかを分類して答えよ
- 多少多いかもですが、
  - 途中までしか出来なくても良いです
  - 試験までには1回解いておくの良いです
  - 実は (1) がちゃんとできれば (2) と (3) はおまけみたいなものです

# 提出方法

## ■ 以下を提出：

### 1. 課題 1 0：

- 提出は Moodle の「課題 1 0」のところからお願いします
- 紙に書いた場合は写真を撮ってアップロードしてください

### 2. 感想や質問：

- 「感想や質問」のところに投稿してください
- わからない場所がある場合、具体的に書いてもらえると良いです

## ■ 提出締め切り

- Moodle に設定した締め切りまで

## ■ 注意：

- 課題の出来は、ある程度努力したあとがあれば良しです
- 必ずしも正解していなくても良いです

# 質問とか感想

---



- キャッシュで一気に性能が向上して感動しました。ちゃんと数式で計算したことでとても納得できました。

- ヒット時やミス時の場合の計算方法が良くわかりませんでした。

# 質問とか感想

- 磁気コアメモリは隣の磁石に影響を受けないですか？
- あと、コンデンサの中って真空なんですか？

# 質問とか感想

- 試験は課題に出た問題が中心に出題されるのでしょうか。
- テスト勉強、なにすればいいですか？
- 試験が不安です。具体的にどんな問題が出題されるか、どのように勉強しておいたら良いか等アドバイスがあれば教えていただきたいです。
  - とりあえずは、練習問題をやっておいてください

# 質問とか感想

- スライドの実行結果のデータで、アクセス範囲がすごく大きくなってもあまりアクセス時間が変わらないのはなぜですか。
- また、L1の容量を超えて、L2で処理するまでの間はどのようにアクセスしているのですか。

# 質問とか感想

- ブラウザのキャッシュについてのところで、2回目からは表示が速い原理が分かったのが面白かったです。
- メモリのキャッシュも同じ原理であることがかりました。メモリのキャッシュは何をとっておくのかと思たら、値だったので、コンピュータはやはり計算機なのだとすることを改めて実感しました。

- メモリが田の字構造でアドレスが振り分けられていることを知って、街みたいだと思った。

- 課題9としてこの構造について解いてみることでさらに理解が深まりました。確率みたいな感じで0.1や $1-0.1$ などを考えていけばいいとわかりました。



- 性能の求めるための周波数はこちらで適当に仮定してしまってよいものなののでしょうか？ 性能を求める目的が比較をすることだから、ここでは問題ないということですか

- メモリの、列の取り出しの話が難しかったです。上2桁が行、下2桁が列なのは分かりましたが、その後が分かりませんでした。

- 「DRAMセルのコンデンサの作り方」で急にボーリング(土掘る方)みたいな図と写真が出てきて面白かったです。
  - あの形をトレンチ（塹壕の意味）といたりもします

- メモリのアクセス時間と容量の関係について、もう少し詳しく説明していただきたいです。具体的には、メモリ容量の増加がどのようにアクセス時間に影響を与えるのか、具体的な数値例や実際のシステムでのシナリオを用いて説明していただけると幸いです。

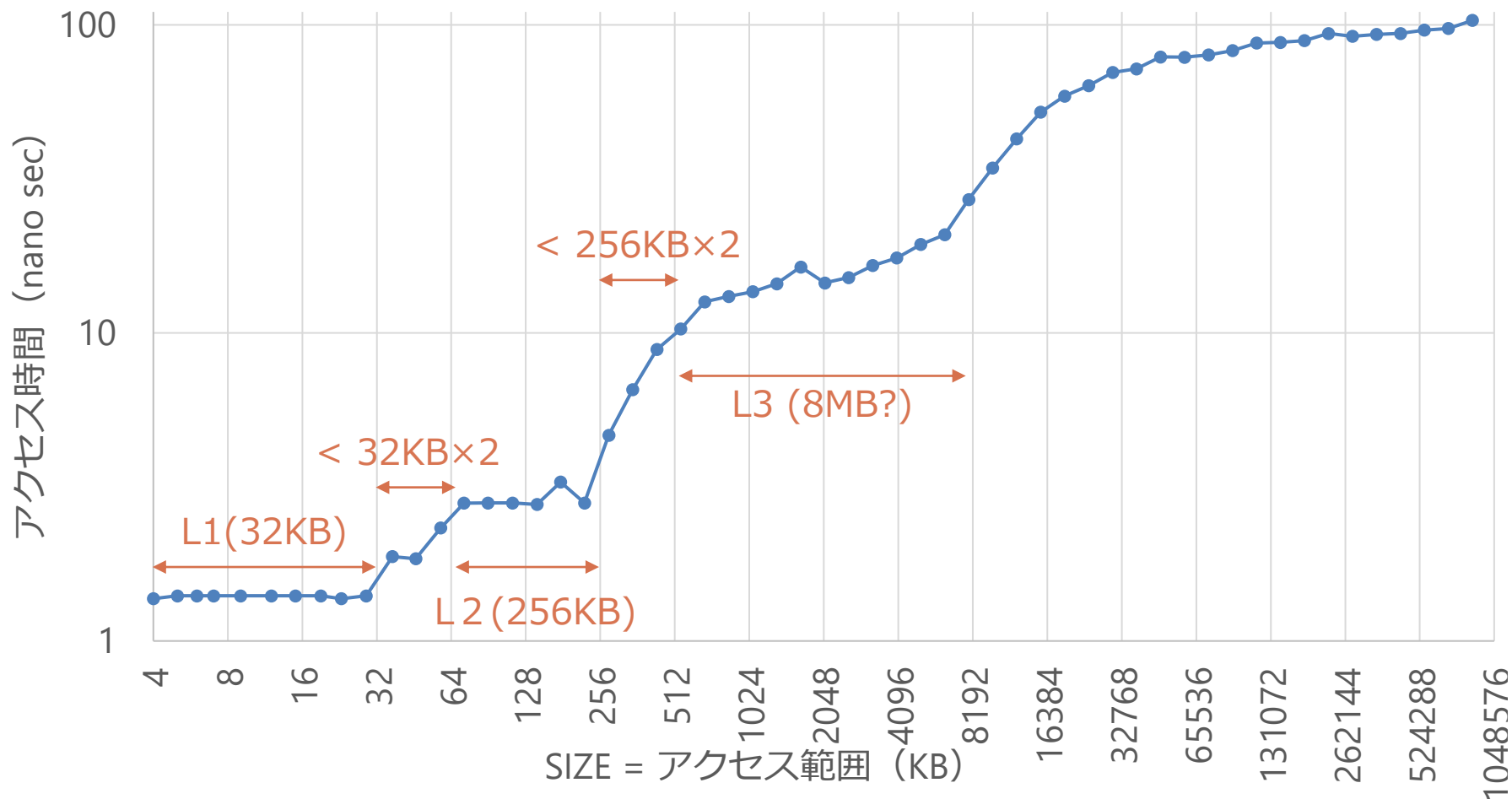
# キャッシュへの性能への影響

- 下記のような二重ループを考える
  - SIZE がキャッシュの容量に収まっていれば、内側ループ終了後に table の全データがキャッシュに乗る
  - 次の内側の周回では全データがキャッシュに乗っているので速い！

```
for (int i = 0; i < NUM_TEST; i++) {  
    uint32_t p = 0;  
    for (int j = 0; j < SIZE; j++) {  
        p += table[j];  
    }  
}
```



# 実際の測定データ



- 縦軸を基数10の対数軸, 横軸を基数2の対数軸に
- 低次キャッシュの内容は必ず高次に含まれる仕様
  - (そうなるかはメーカーや世代に依存. 含まれないこともある)
  - 256KB+32KB ではなく 256KB で変化しはじめる

# 質問とか感想

- 難しかった