

# 先進計算機構成論 12

---

東京大学大学院 情報理工学系研究科 創造情報学専攻

塩谷 亮太

shioya@ci.i.u-tokyo.ac.jp

# 前回の内容

1. GPU のアーキテクチャの基本
  1. 基本的な構造
  2. バックエッジの対処
2. アクセラレータは何故速いのか

# 今日の内容

## 1. キャッシュ

# キャッシュとは？

- 「キャッシュ」って、ブラウザのあれ？
- ちょっと違うけど、原理は同じもの

# 原理は同じ

## ■ ブラウザのキャッシュ

- ◇ WEB サーバーからページを取ってくるのは遅い
- ◇ 1回見たページを PC やスマホの「キャッシュ」に置いておく
- ◇ 2回目からは表示が速い

## ■ メモリのキャッシュ

- ◇ メイン・メモリからデータを取ってくるのは遅い
- ◇ 1回読んだデータを CPU の「キャッシュ」に置いておく
- ◇ 2回目からは読み込みが速い

# 性能へ大きく影響するし、影響範囲も広い

- キャッシュが問題になることは非常に多い
  - ◇ ほとんどのプログラムで性能に大きな影響を与えている

# 例：行列積の実装と性能

## ■ 三重ループとして実装できる

◇ ループの順番は任意に入れ替え可能

```
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];
```

## ■ 単にループの順番を入れ替えるだけで処理時間が大きく変化する

◇ 外側から k j i の順 → 178秒

◇ 外側から k i j の順 → 20秒

◇ 外側から j i k の順 → 1100秒

## ■ この変化は、キャッシュを有効に働かせているかどうかによって由来

◇ 計算量や全体としてのメモリ使用量は全く同じなのに

# 内容

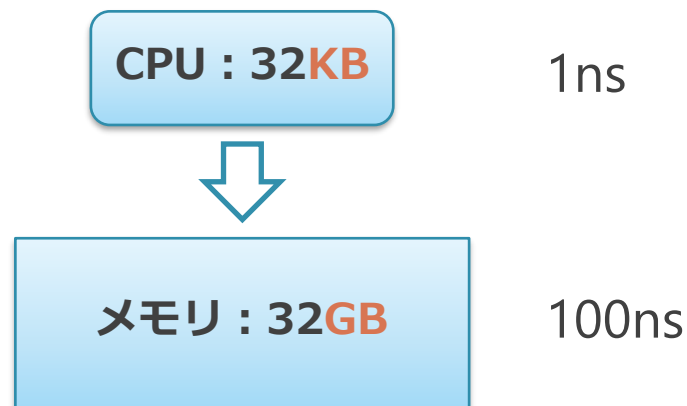
1. メモリの容量と速度
2. キャッシュの基本的な考え方
3. キャッシュの構成方法
4. 行列積での動作例



# メモリの性質

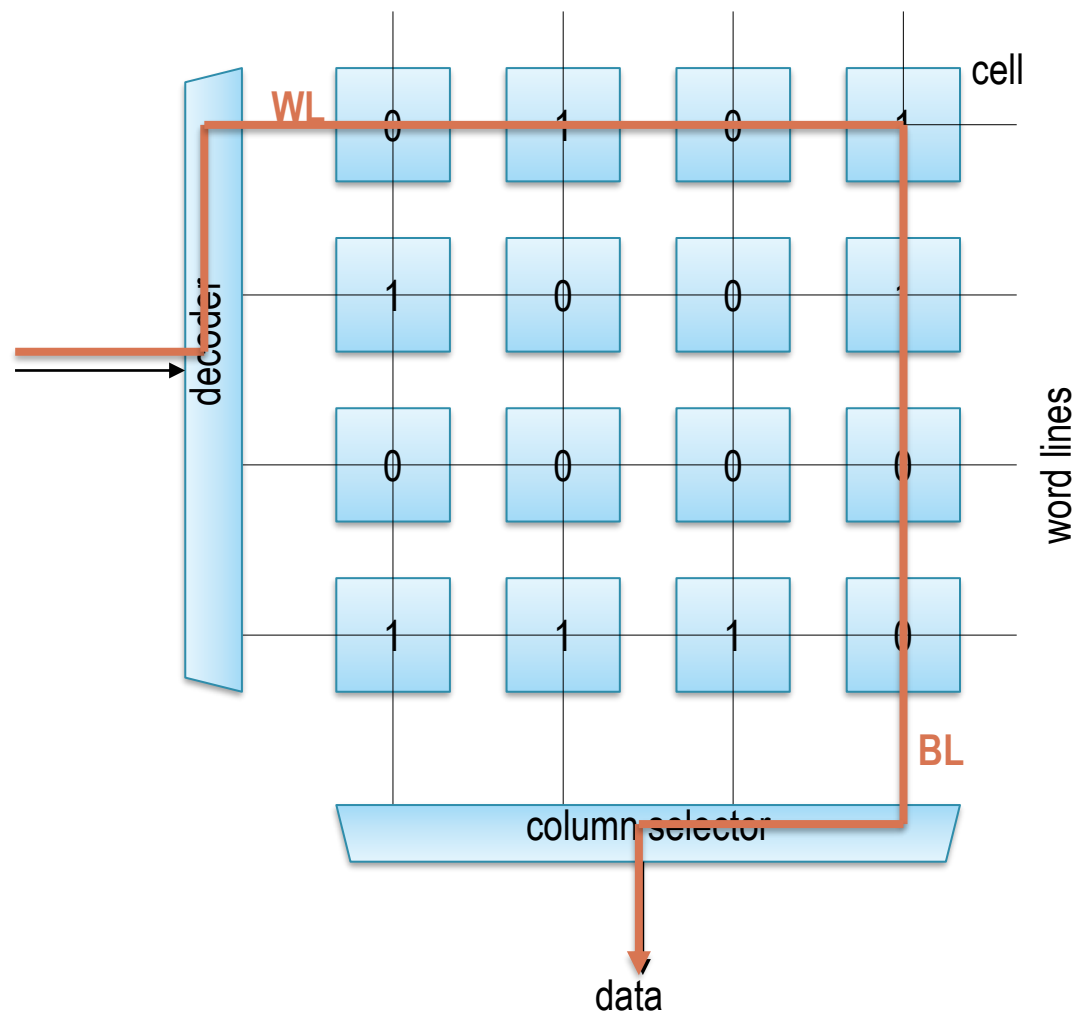
- 容量（大きい or 小さい）
  - ◇ どのくらい数のデータを覚えることができるか
- 速度（速い or 遅い）
  - ◇ どのくらいの速さで読み書きできるか
- 容量と速度にはトレードオフがある
  - ◇ 「大きくて速くいメモリ」は作ることができない

# 実際にはアクセス時間は容量の平方根ぐらいに比例



- 格子状に並んだセルの外周部の長さがアクセス時間を決めるから

# アクセス時間は容量の平方根ぐらいに比例



- 格子状に並んだセルの外周部の長さがアクセス時間を決める
- メモリ容量  
= (外周部の長さ)<sup>2</sup>
- 容量（セルの数）が一定の場合、正方形に近くするのが最も経路が短くなる

# 速度

## ■ 信号線の長さがなぜ問題に？

◇ 電気信号が伝わる速度はとても速いのでは？

## ■ 計算してみる：

◇ 光の速度：                      秒速 30万Km =  $3 \times 10^8$  m

◇ CPU の動作周波数：              3GHz      =  $3 \times 10^9$  Hz

◇ 1回の処理で光が進める長さ =  $(3 \times 10^8) / (3 \times 10^9) = 0.1\text{m} = 10\text{cm}$

## ■ 光の速度でも大して進めないぐらい、今のコンピュータは速い

◇ 回路中の電気信号の伝達は光よりもだいぶ遅い

# データをとってくるのに、どのくらいかかるか？



# メモリのまとめ

## ■ メモリ

- ◇ 情報を記憶し，アドレスで指定して読み書きする回路

## ■ 速さと大きさにはトレードオフがある

- ◇ 「大きくて遅いメモリ」か「小さくて速いメモリ」しか作れない

## ■ 実際には，セルをどのような方式で作るかでも大きく変わる

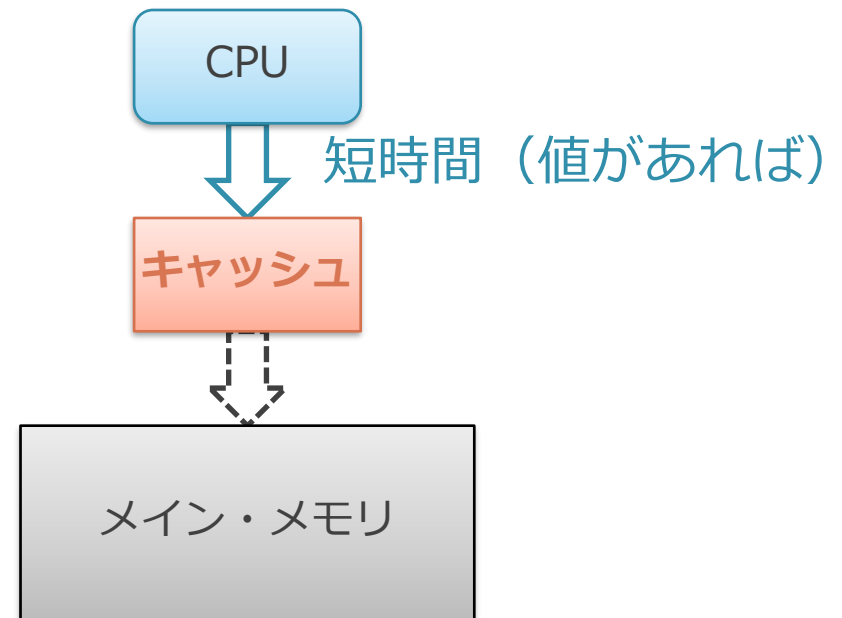
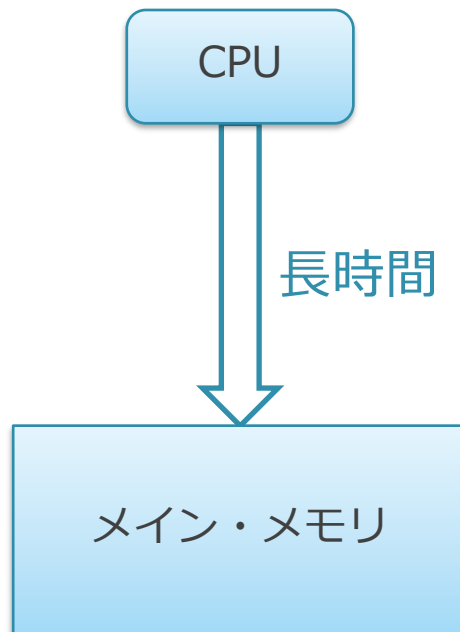
- ◇ しかし，上記のトレードオフは常になりたつ

1. メモリの容量と速度
- 2. キャッシュの基本的な考え方**
  1. 基本的な原理と構造
  2. 容量の性能への影響
  3. キャッシュのレイテンシと命令スケジューリング
3. キャッシュの構成方法
4. 行列積での動作例

# 記憶階層

## ■ 以下を階層的に組み合わせる

- ◇ 小さいけど速いメモリ：キャッシュ
- ◇ 大きいけど遅いメモリ：メイン・メモリ



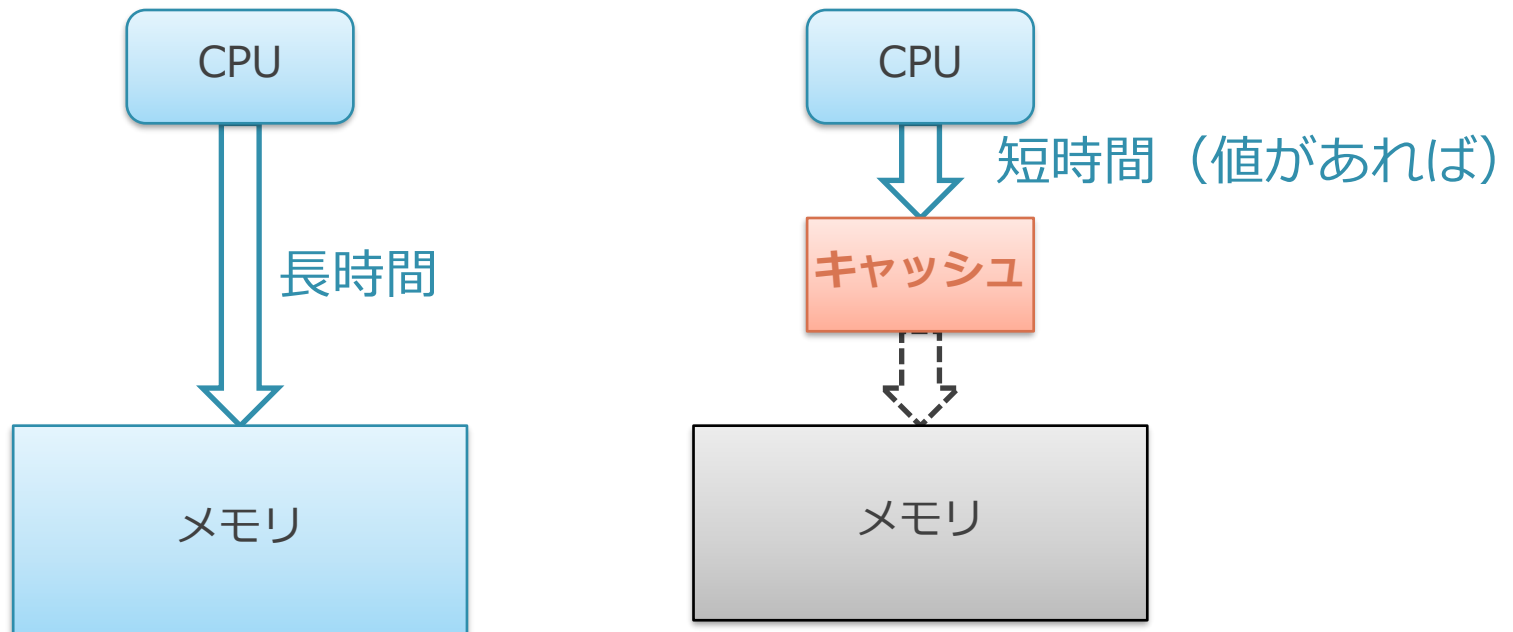


# キャッシュの動作

## ■ ソフトの性質：

- ◇ 一度使用した値は，すぐにまた使う可能性が高い
- ◇ ブラウザのキャッシュと同じ

## ■ 一度利用した値を入れておくことで，2回目からは高速に

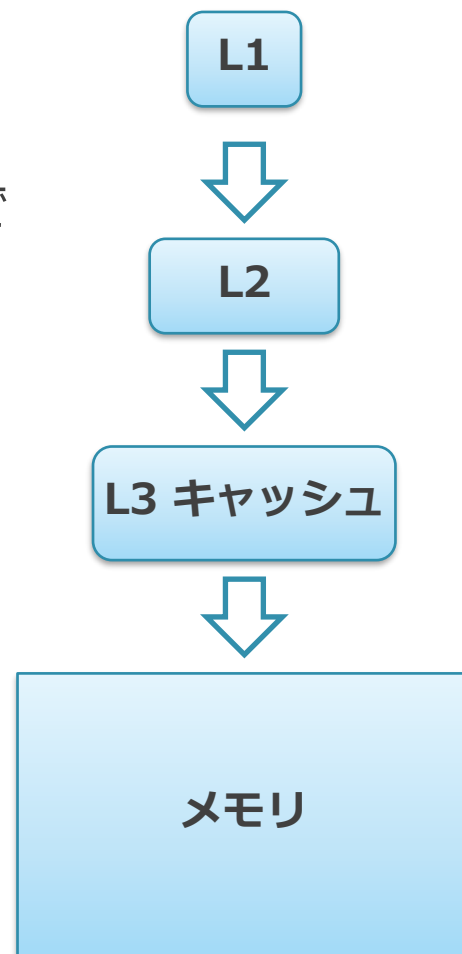


# 時間局所性

- 「一度使用した値は，すぐにまた使う可能性が高い」という性質
- なぜか？
  - ◇ 人間は，「普通は」関連のある処理同士を近くを書く
  - ◇ プログラムの各行ごとに全く関係無い処理を混ぜたりは「普通は」しない
- 関連する処理は，当然同じデータを使う可能性が高い
  - ◇ 同じ入力データの使いまわし
  - ◇ 前の行の結果を次の行の入力で使う
    - for ループなどはこれらの典型

# 実際には多層の構造になっている

- 中間の速さと大きさのキャッシュが複数組み合わされている
  - ◇ PC に使われている CPU だと L1 ～ L3 まであるのが一般的
  - ◇ L は Level の略で、大きいほど（高次）CPU から遠い



# キャッシュの基本的な考え方のまとめ

## ■ 記憶階層

- ◇ 小さいけど速いメモリ：キャッシュ
- ◇ 大きいけど遅いメモリ：メイン・メモリ

## ■ 時間局所性

- ◇ 一度使用した値は、すぐにまた使う可能性が高い

## ■ 実際には多層の構造になっている

- ◇ 中間の速さと大きさのキャッシュが複数組み合わせられている

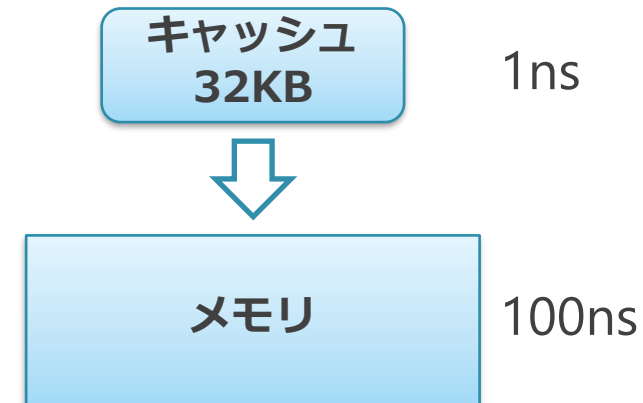
1. メモリの容量と速度
2. キャッシュの基本的な考え方
  1. 基本的な原理と構造
  2. **容量の性能への影響**
  3. キャッシュのレイテンシと命令スケジューリング
3. キャッシュの構成方法
4. 行列積での動作例

# キャッシュへの性能への影響

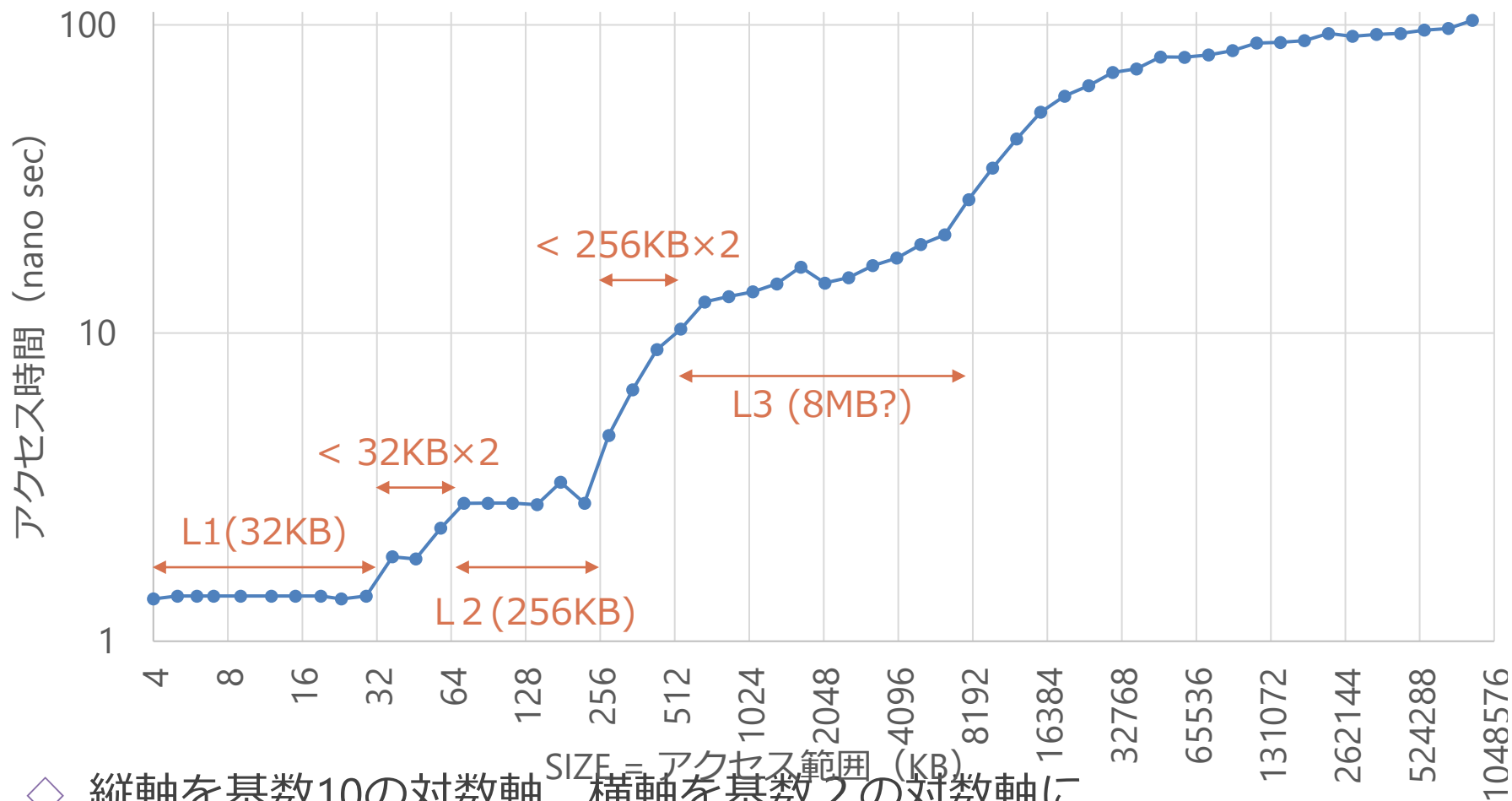
## ■ 下記のような二重ループを考える

- ◇ 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];  
    }  
}
```

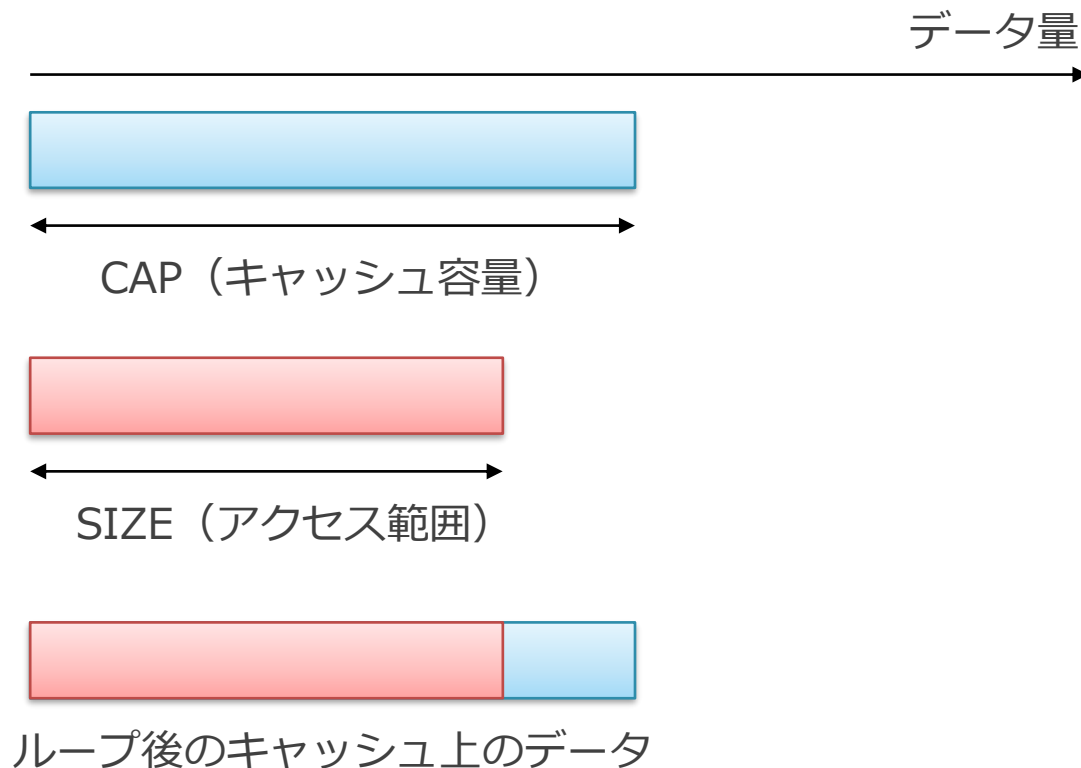


# 実際の測定データ



- ◇ 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

# プログラム最適化の上で、重要なポイント



## ■ 上記の状態を保つこと

◇ ワーキング・セット：

□ 処理のまとまりごとに、アクセスするデータの範囲（使用量）

◇ ワーキング・セットがキャッシュ容量に収まることが重要



# キャッシュ容量を意識した最適化

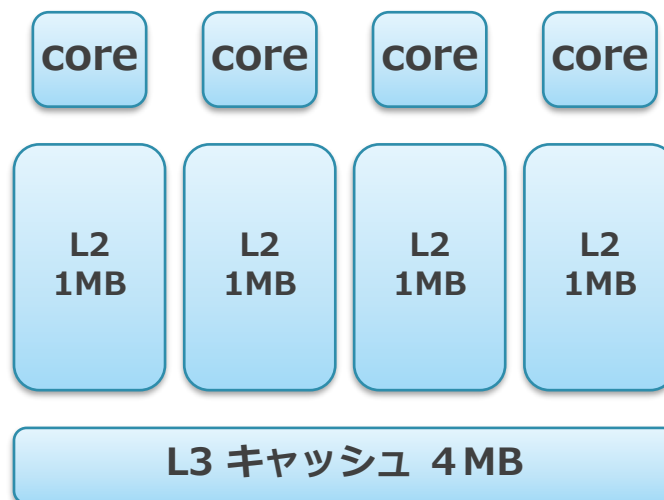
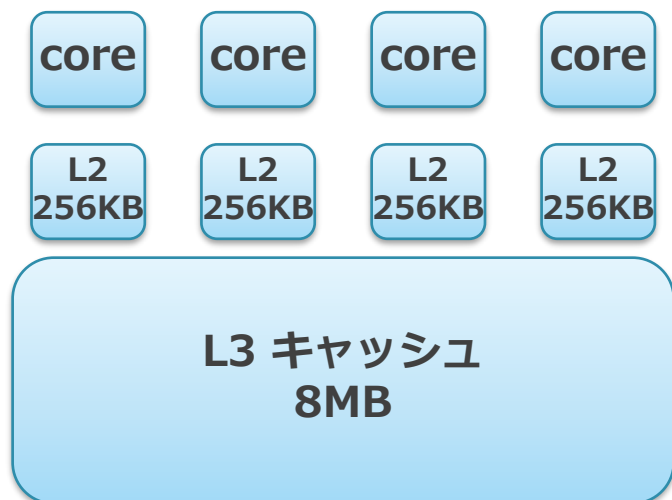
- 数十KB, 数百KB, 数MB あたりに典型的に壁があることを覚えておく  
とよい
  - ◇ L1 に収まっていれば, 最高速が出せる
  - ◇ 実際には色んなデータが同時にキャッシュを使うので, もう少し  
少なめを意識した方がよい

# キャッシュ容量を意識した最適化2

- 現実には、具体的なキャッシュ容量をそこまで意識しないでもよい
  - ◇ アクセス範囲が小さくなるよう心がけるだけでも十分効果がある
  - ◇ 範囲が容量を超えても、ただちに効果がなくなるわけではない
    - 2倍ぐらいまでは効果が持続する
    - 階層化されているので、どっかの階層にひっかかるようになるだけでも効果がある

# 最近はまだ階層のトレンドが変わりつつある

- 複数のコアを意識してバランスが変わっている
  - ◇ 新しい Intel Xeon だと L2 を大きく, L3 を小さく
  - ◇ 各コアの L2 は 1MB ぐらいで, L3 が各コアから溢れたデータの受け皿に



# より大容量のメモリを使う場合

- 実際にはさらに上に、メモリ容量の壁がある
  - ◇ 物理メモリを超えた容量を使う場合、ハードディスクが使われる
  - ◇ この場合、メイン・メモリがハードディスクのキャッシュになる
- ハード・ディスクはミリ秒単位でアクセスに時間がかかる
  - ◇ 物理的に回っている円盤だから、読み出し時の位置合わせが大変
  - ◇ メイン・メモリは100ナノ秒程度なので、極めて遅い
    - $1\text{ミリ秒} = 10^6\text{ナノ秒}$

1. メモリの容量と速度
2. キャッシュの基本的な考え方
  1. 基本的な原理と構造
  2. 容量の性能への影響
  3. **キャッシュのレイテンシと命令スケジューリング**
3. キャッシュの構成方法

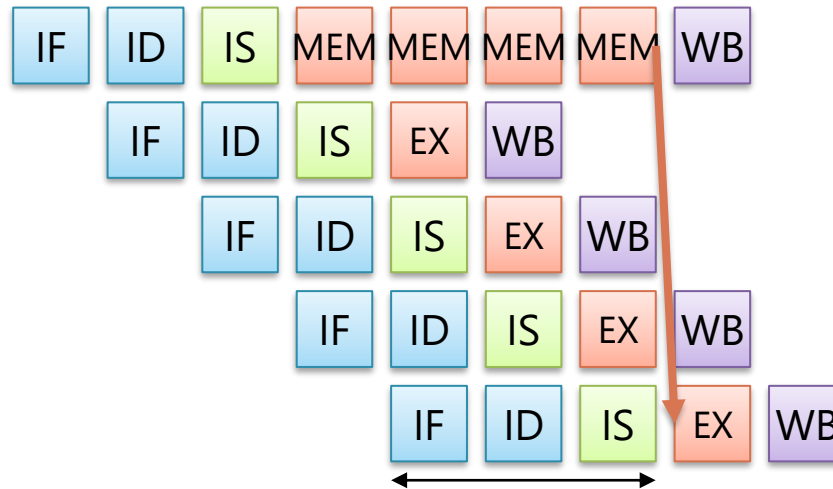
# キャッシュのレイテンシと命令スケジューリング

## ■ キャッシュ階層ごとのレイテンシの例

- ◇ L1: 4 サイクル
- ◇ L2: 10 サイクル
- ◇ L3: 30 サイクル
- ◇ メイン・メモリ : 300 サイクル

## ■ 命令のスケジューリング能力と関係する

# L1: 4サイクル



3 サイクル間は依存しない命令を実行する必要がある

## ■ L1 レイテンシが4サイクル =

◇ MEM ステージが4段にパイプライン化

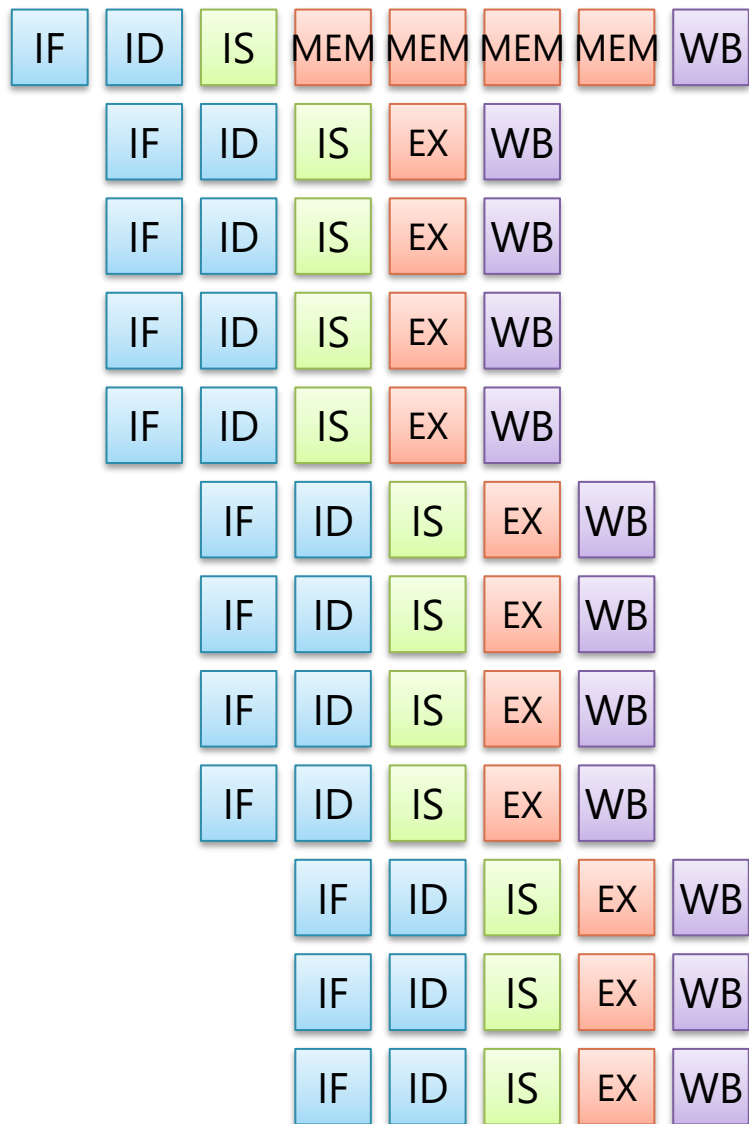
## ■ 後続の3サイクルは, ロードに依存する命令は実行できない

◇ スcalarプロセッサなら, なにか依存関係にない3命令があれば時間が潰せる

◇ 静的スケジューリングでも対応できるぐらい

□ 無依存な命令をロードの後ろに3命令いれておけばよい

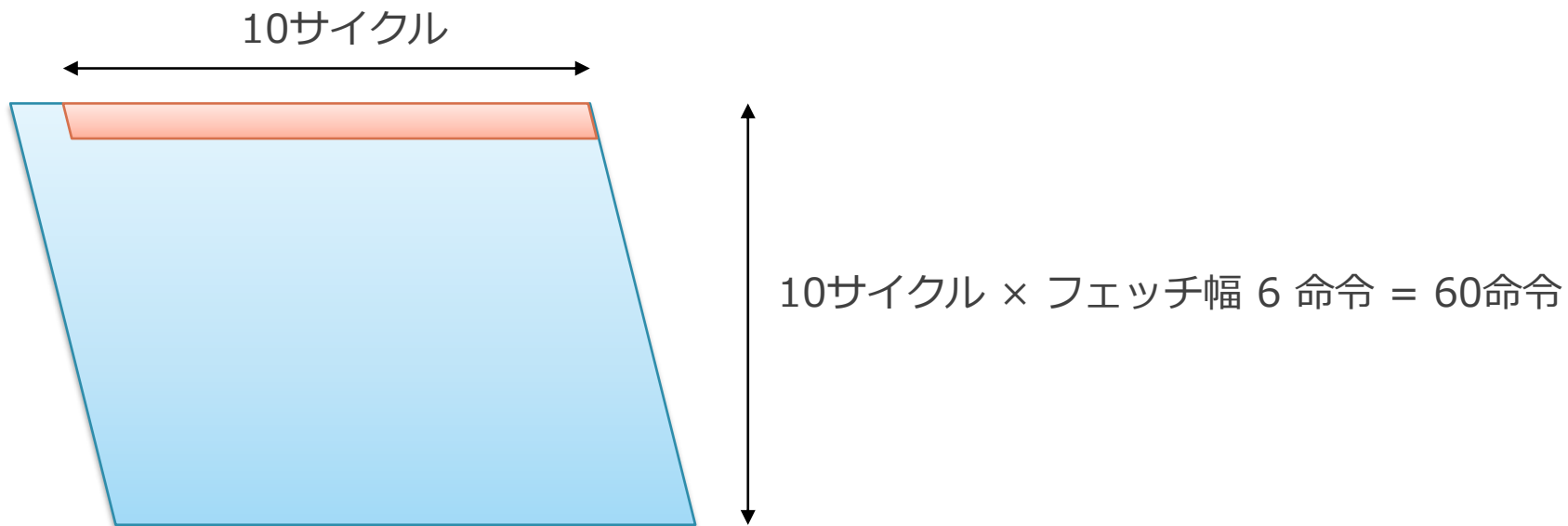
# L1: 4サイクル



- スーパースカラでは、パイプラインを全く止めずに最大幅分実行させるのは辛くなってくる
- これ以上は L1 のレイテンシは伸ばせない

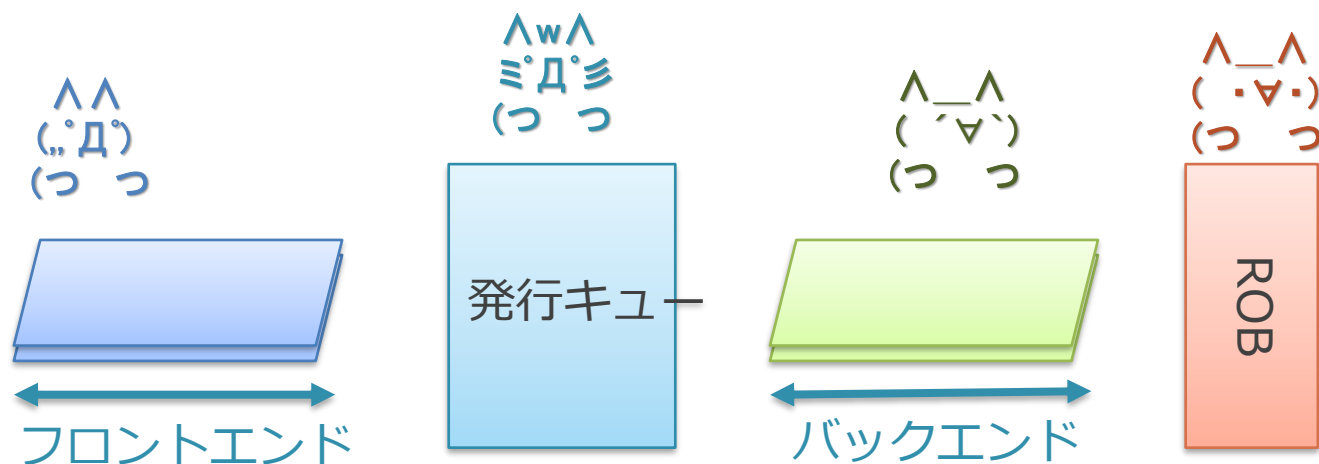


## L2: 10サイクル



- L2 ぐらいになると, 60 命令ぐらい並列に実行できないといけない

# Out-of-order スーパースカラ・プロセッサ との関係



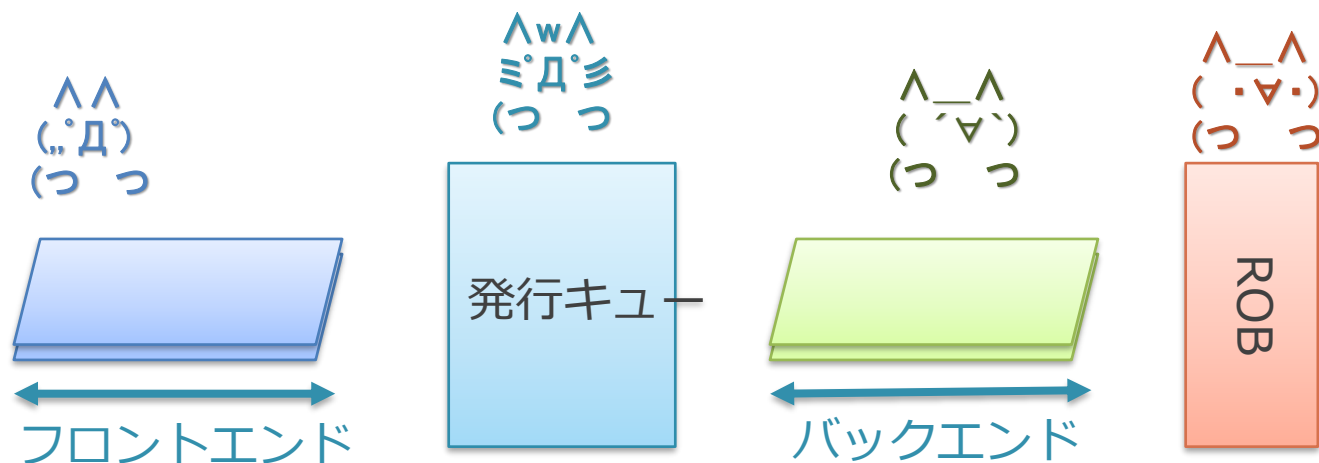
## ■ 動作

- ◇ フロントエンドからは毎サイクル 6 命令ぐらいが発行キューに挿入
- ◇ L2 アクセスの 10 サイクル間に 60 命令が挿入される

## ■ 実際には発行キューからは「L2 アクセスする命令に無依存で発行できた命令は」消えていく

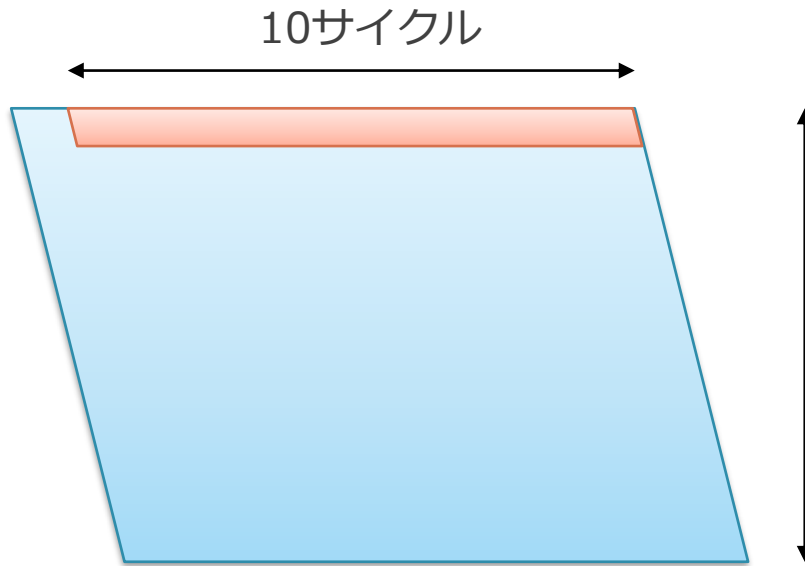
- ◇ しかし依存があるものはどんどん溜まっていく

# Out-of-order スーパースカラ・プロセッサ



- リオーダーバッファ (ROB) はプログラム順にしか出ていけない
  - ◇ 最低でもフロントエンドの幅 × L2 レイテンシ ぐらいはないと ROB にエンタリが確保できずフロントエンドが止まってしまう
  - ◇ (ROB のエンタリはフロントエンドで確保する)

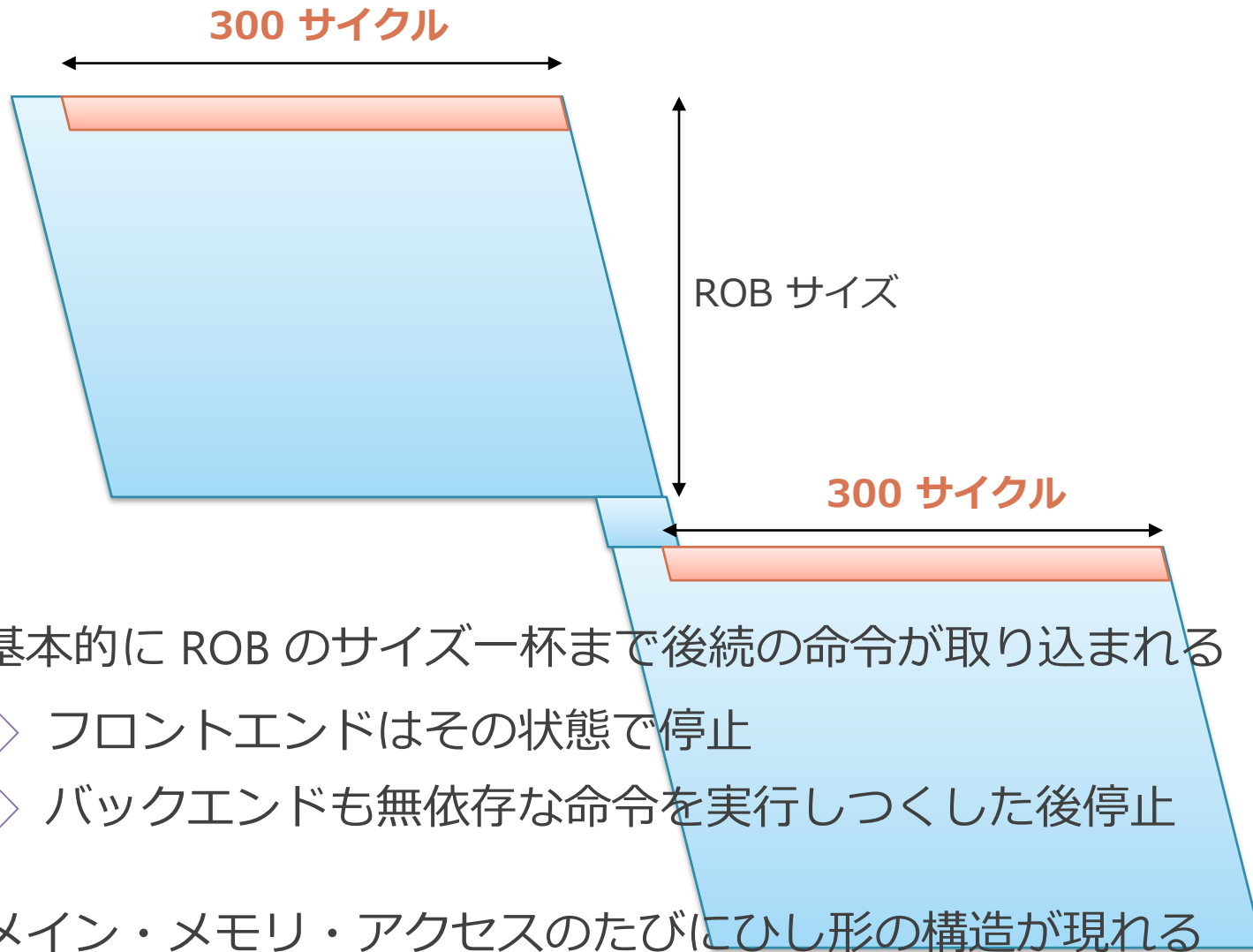
# ROB の状況



一番上の命令がコミットするまで、  
この部分は ROB にたまり続ける

- メモリ・アクセスのレイテンシ × フェッチ幅 分は ROB のエントリが必要

# メイン・メモリ・アクセス : 300 サイクル

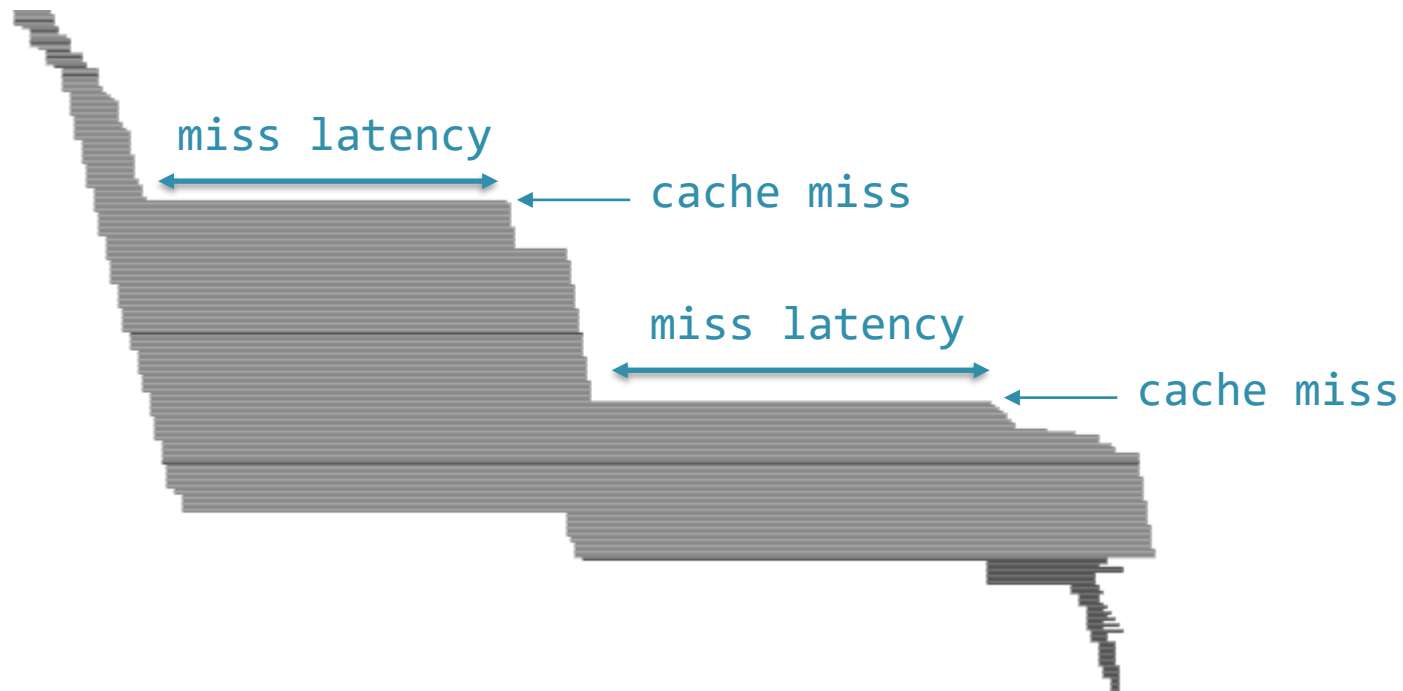


- 基本的に ROB のサイズ一杯まで後続の命令が取り込まれる
  - ◇ フロントエンドはその状態で停止
  - ◇ バックエンドも無依存な命令を実行しつくした後停止
- メイン・メモリ・アクセスのたびにひし形の構造が現れる
  - ◇ メモリ・アクセスがコミットされ ROB が解放されるとフェッチが再開

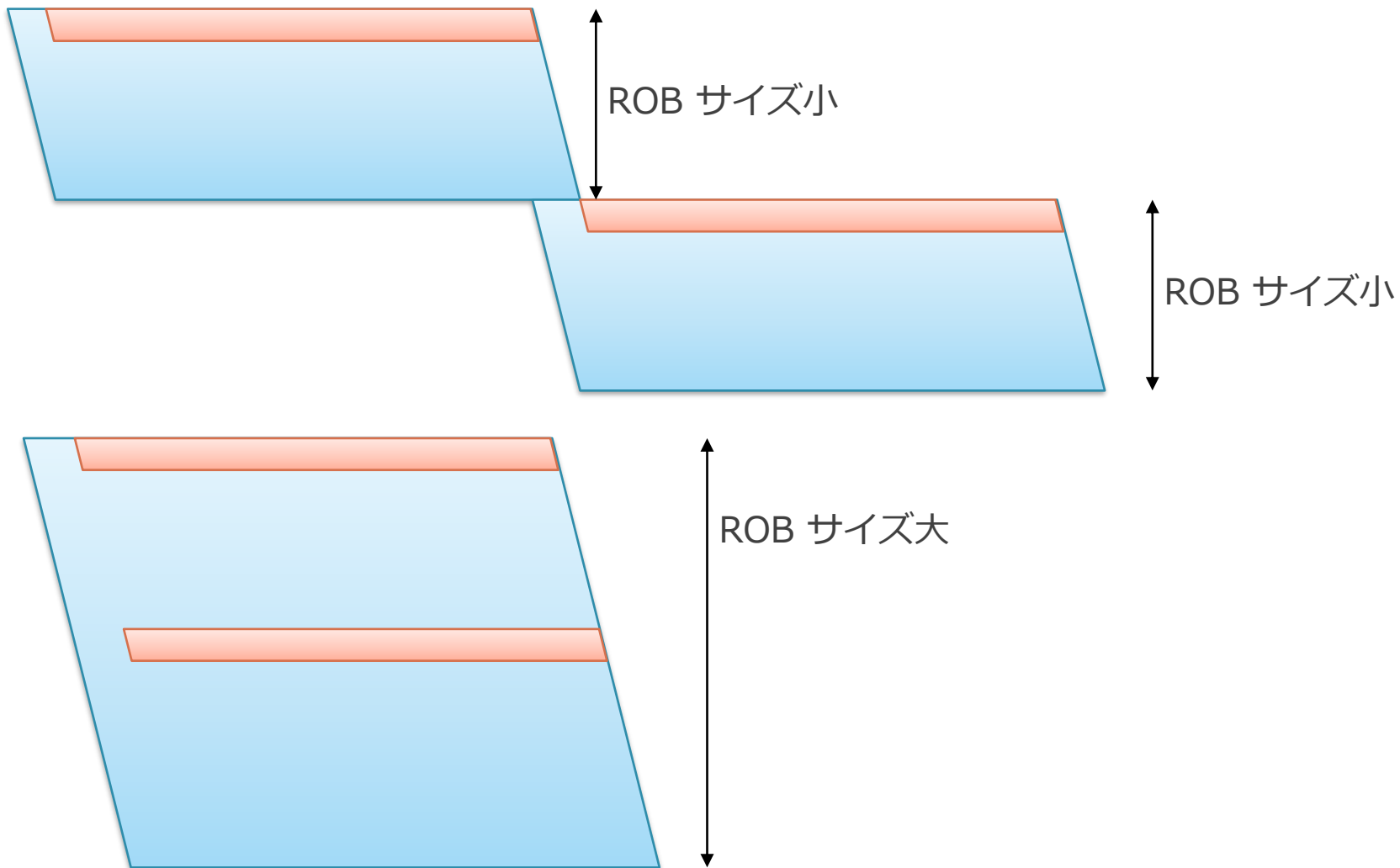
# 実際のプログラム実行の様子

## SPEC CPU 2006 ベンチマークの mcf より

- 下記の場合、ミス同士が直列に依存していたため、ひし形が合成されたような形になっている



# メモリ・レベル並列性 (Memory Level Parallelism)



- ROB が大きいと、お互いに無依存な複数のメモリ・アクセスをオーバーラップして実行できるようになる

# キャッシュのレイテンシと命令スケジューリング

- 発行キューや ROB のサイズとキャッシュのレイテンシは関連している
  - ◇ バランスするように各部のパラメータが決定されている
  - ◇ 各レベルのキャッシュのレイテンシ, フェッチ幅, 発行幅, 発行キューのサイズ, ROB のサイズ
- メモリ・レベル並列性
  - ◇ メイン・メモリ・アクセスが並列して打てると性能向上が非常に大きい
  - ◇ ROB を非常に大きくとこの効果が得やすくなる
    - 単純に計算が並列にできる意味での命令レベル並列性とはまた別の効果



1. メモリの容量と速度
2. キャッシュの基本的な考え方
- 3. キャッシュの構成方法**
4. 行列積での動作例

# キャッシュの詳細

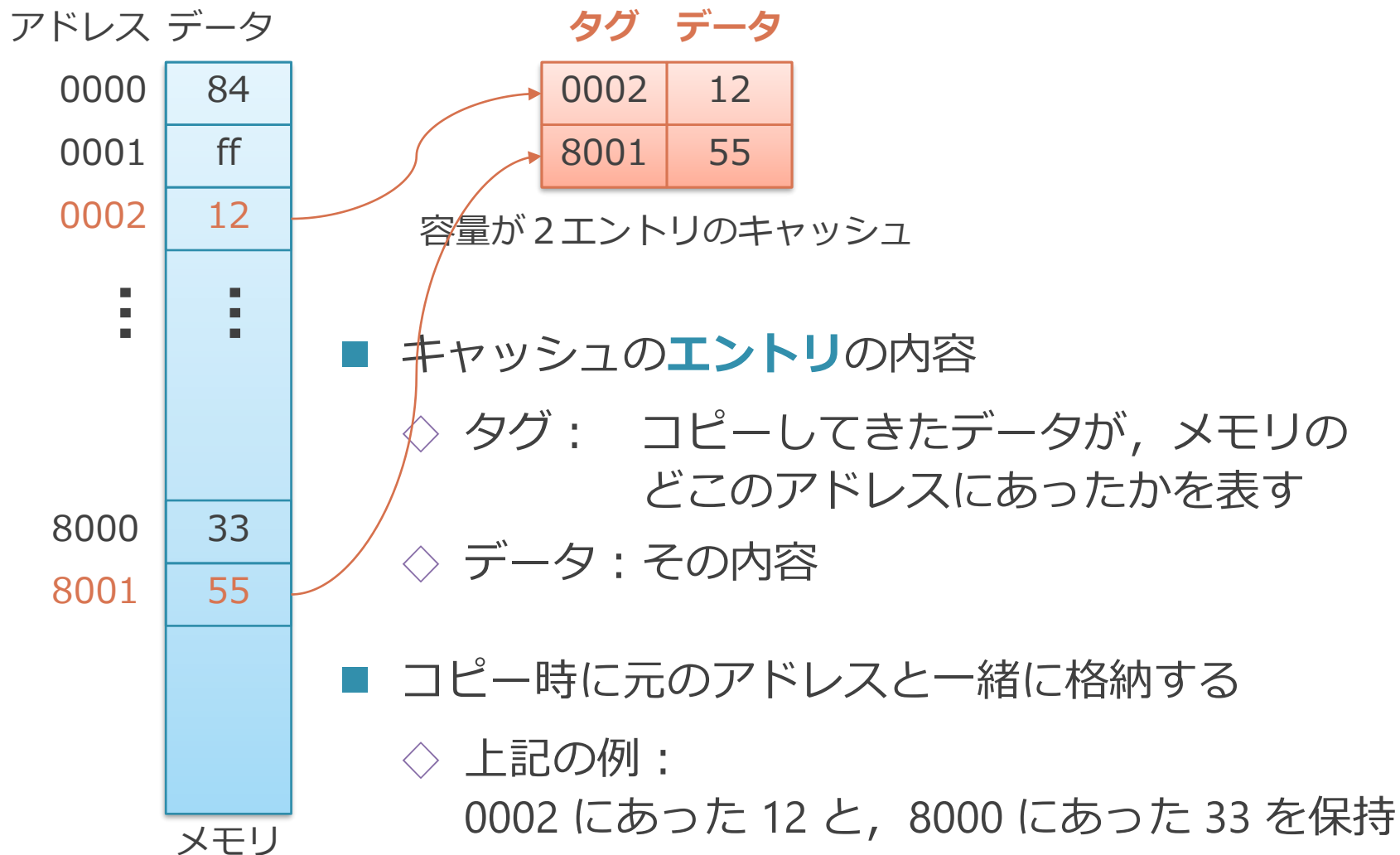
## 1. 方式：

- ◇ 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

## 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

# キャッシュの基本的な構造



# 読み出し時の動作

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

- ◇ 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

# ダイレクトマップ方式

タグ データ		
	0	8000 <b>0</b> 12
	1	100 <b>1</b> 33
	2	010 <b>2</b> 00
	3	550 <b>3</b> 78

- 「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 :  $8000 \bmod 4 = 0$  番にアクセス
  - ◇ アドレス 5513 :  $5513 \bmod 4 = 3$  番にアクセス
- 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** のアクセスがあると, 0 番しか使えない

# セットアソシアティブ方式

	タグ データ		タグ データ	
0	8000	12	0100	53
1	1001	33	7701	44
2	0102	00	5102	22
3	5513	78	0503	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

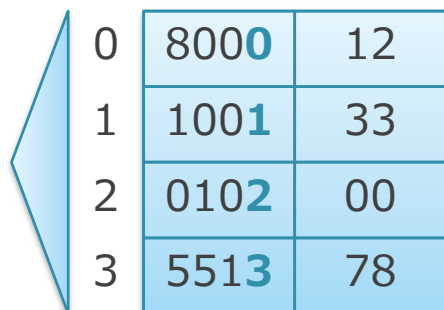
## ■ どこにもヒットしなかった場合

- ◇ 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もの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
- 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よりもタグを覚えているようなもの

# 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

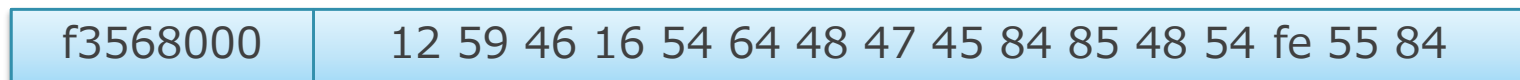
## ■ 利点：ラインサイズが増えると、データが占める割合が増える

- ◇ 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)



# 空間局所性

## ■ 2種類の局所性

### 1. 時間局所性：

- 「一度使ったデータは、すぐにまた使われる」

### 2. 空間局所性：

- 「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」

## ■ たとえば、

- ◇ 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

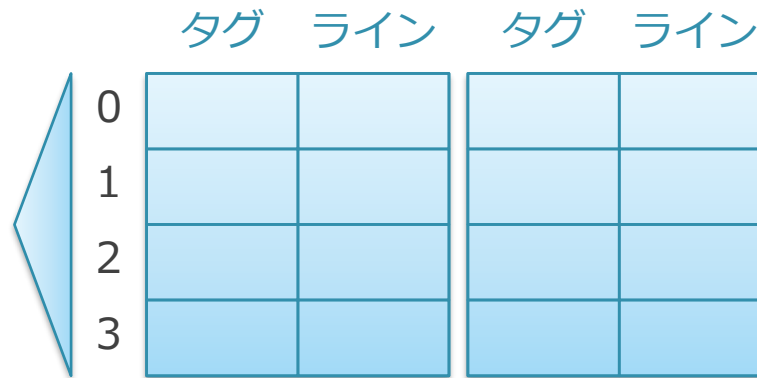
# キャッシュの詳細

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



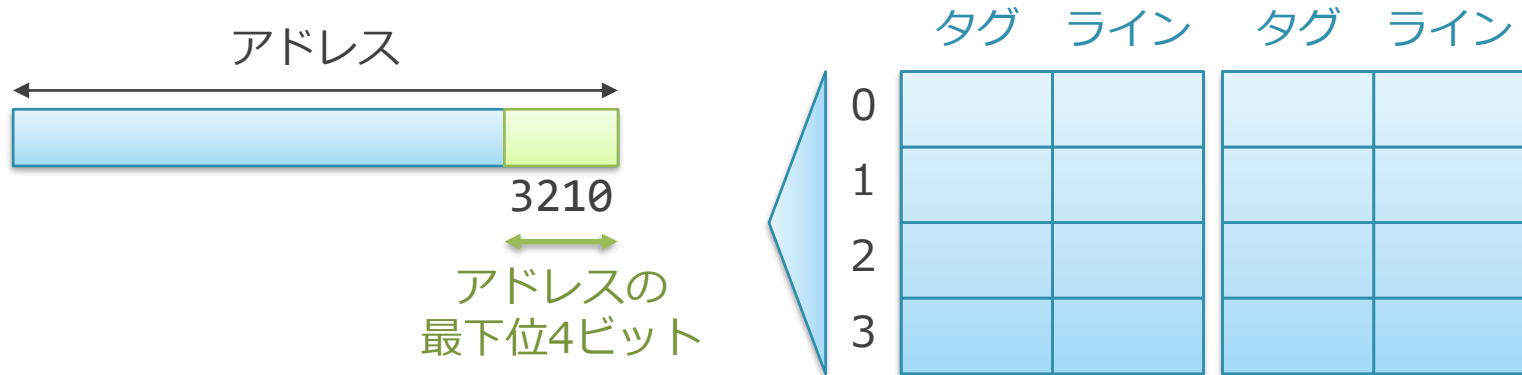
## ■ 構成

- ◇ 連想度 : 2
- ◇ セット数 : 4
- ◇ ラインサイズ : 16バイト

## ■ 総記憶容量

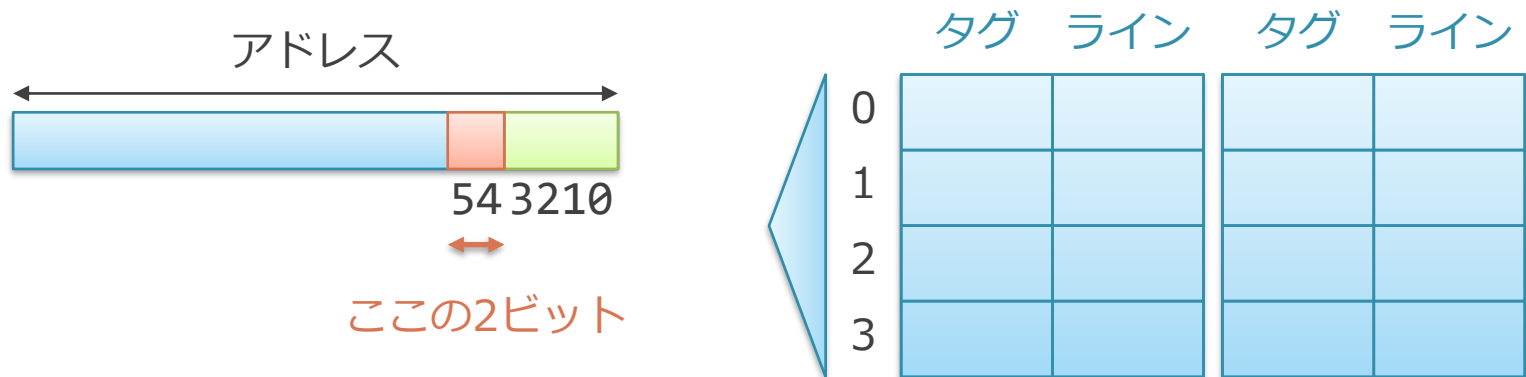
- ◇ 連想度 2 × セット数 4 × ライン 16 バイト = 96 バイト

# アドレスとラインの対応



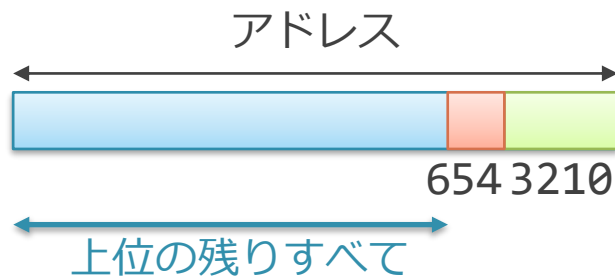
- アドレスは1バイト単位でメモリの位置を表すものとする
- 最下位ビット 0 ～ 3 （計 4 ビット）
  - ◇ 最下位部分がライン内の位置に対応
    - 空間局所性を利用するため
  - ◇ 4ビットなのは, ラインサイズが16バイトだから
    - $2^4 = 16$
    - (ラインサイズは必ず 2 の累乗になる)

# 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

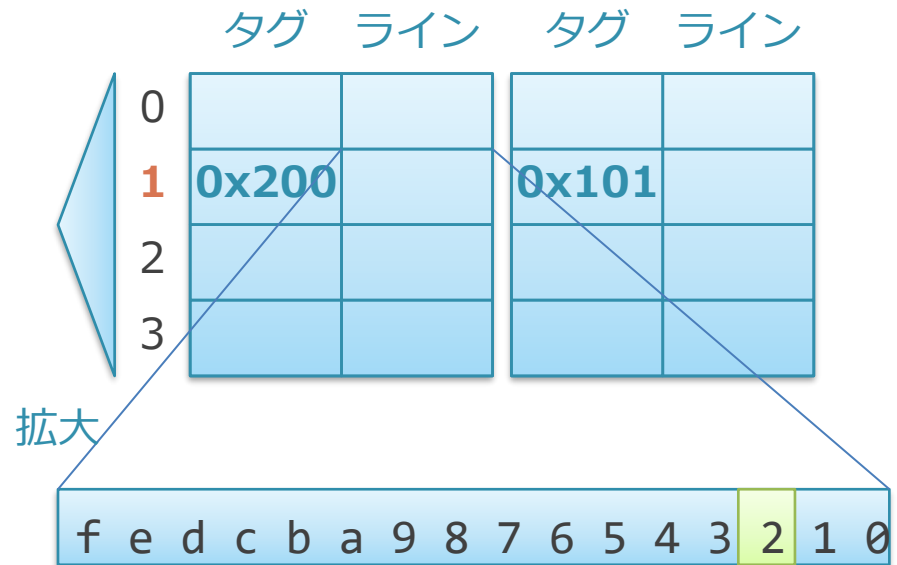
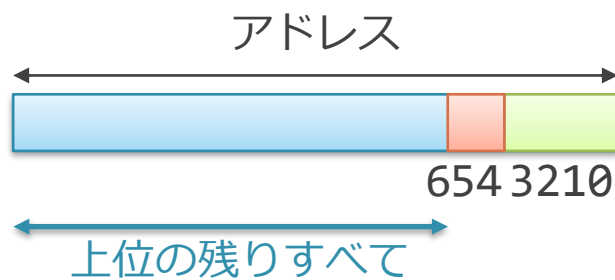
# アドレスとタグの対応



	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

# 内容

1. メモリの容量と速度
2. キャッシュの基本的な考え方
3. キャッシュの構成方法
4. 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

## ■ 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

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

タグ    ライン

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]
	⋮

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

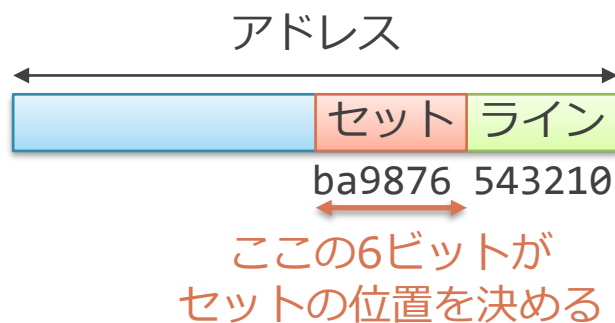
◇ **アドレスが等間隔になる**

□ 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

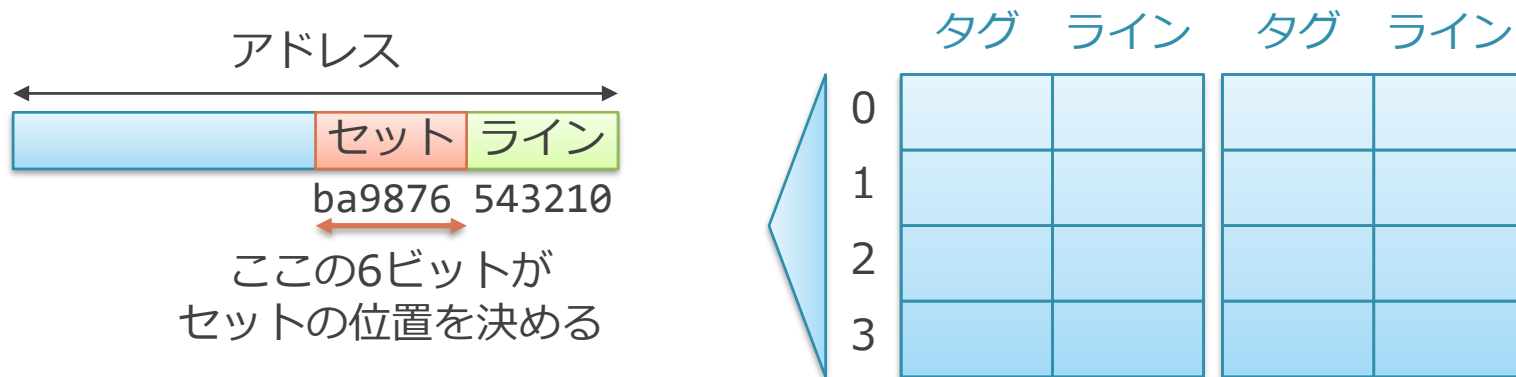
# アドレスとセットの対応の復習



	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

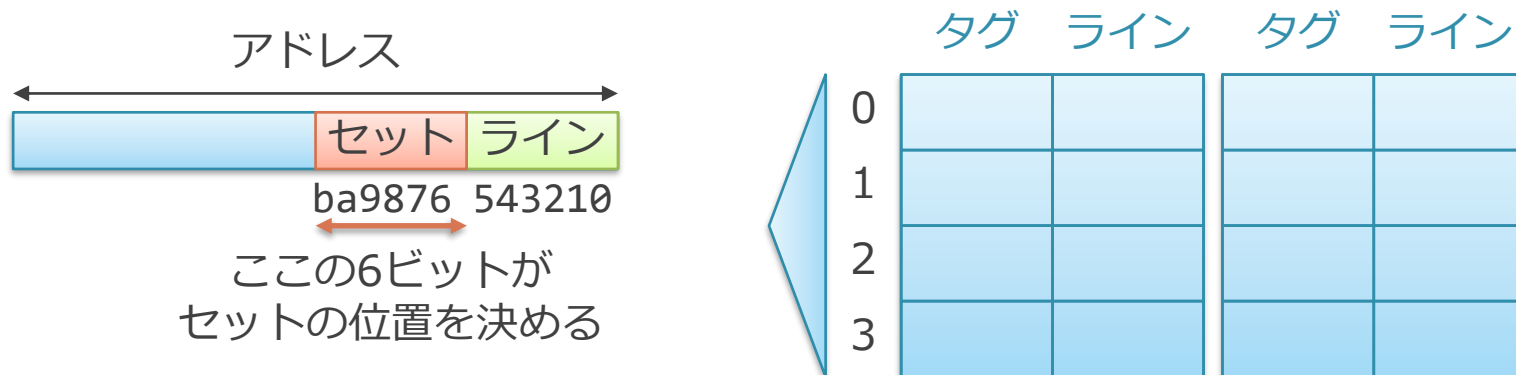


```
uint32_t A[1024][1024];  
for (int j = 0; j < SIZE; j++)  
    A[j][0]++;
```

## ■ 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]++;
```

- 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

# 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

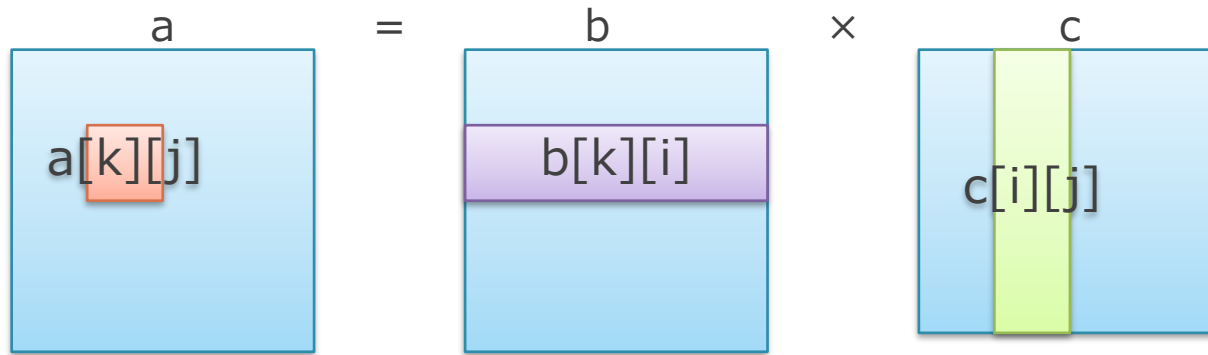
# 基本的な行列積の実装

```
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

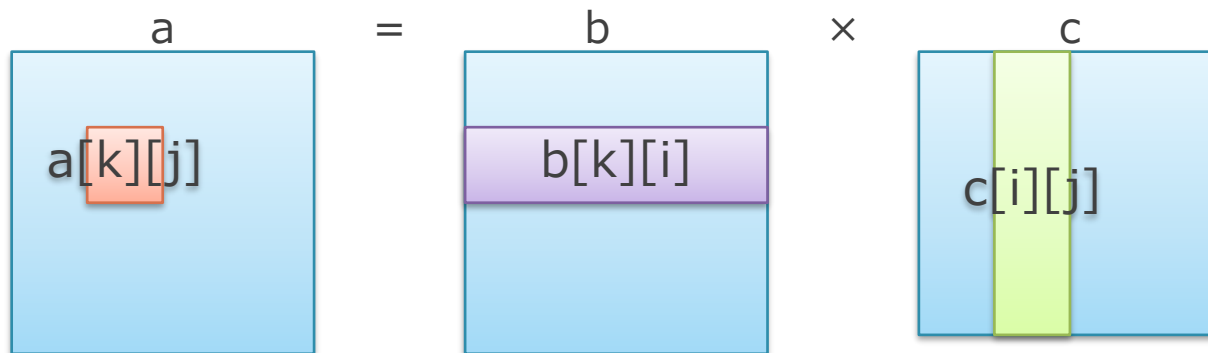
- ◇ 一番内側の  $i$  はこの各要素を参照するために回る

# 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

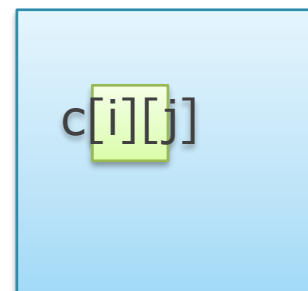
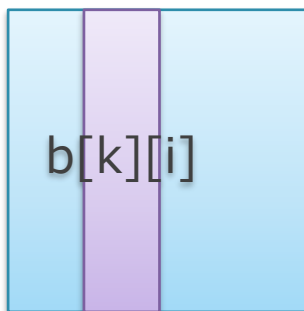
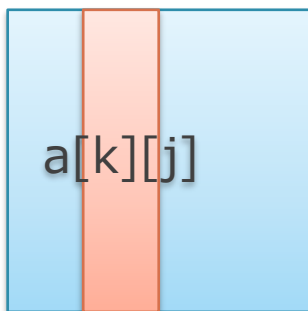
# i, j, k をひっくり返した時の、 最内周ループのアクセス範囲



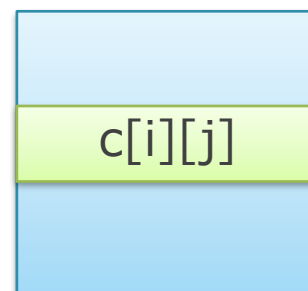
```
for (int k = 0; k < SIZE; k++) {  
    for (int j = 0; j < SIZE; j++) {  
        for (int i = 0; i < SIZE; i++) { // i が変化
```

- 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
```

## ■ キャッシュ

- ◇ 基本原理
- ◇ 容量と性能の関係
- ◇ CPU の振る舞いとの関係
- ◇ 詳細な構造
- ◇ 行列積での性能の変化

- ISCA2023/MICRO2023 ないしは、次のページで指定するいくつかの論文のうち1つを読んで、その内容を説明すること

(ISCA2024 はまだ論文が落とせないなので、2023 にしました)

- ◇ <https://dl.acm.org/doi/proceedings/10.1145/3579371>
- ◇ <https://dl.acm.org/doi/proceedings/10.1145/3613424>

- Fetch directed instruction prefetching
  - ◇ 最近の CPU のフロントエンドはこれに近い構造を持っている事が多い.
  - ◇ <https://ieeexplore.ieee.org/abstract/document/809439>
- Best-offset hardware prefetching
  - ◇ Data prefetch championship 2 優勝. ARM のサーバー向け最新 CPU にも実際に入ってるぽい.
  - ◇ <https://ieeexplore.ieee.org/abstract/document/7446087>
- Runahead execution: an alternative to very large instruction windows for out-of-order processors
  - ◇ キャッシュミスしたときに Runahead と呼ぶモードに入りプリフェッチ実行を行う.
  - ◇ <https://ieeexplore.ieee.org/abstract/document/1183532>

- Long term parking (LTP): criticality-aware resource allocation in OOO processors
  - ◇ 割と単純に OoO スーパスカラのスケジューリング能力を上げられる.
  - ◇ <https://dl.acm.org/doi/10.1145/2830772.2830815>
- Constable: Improving Performance and Power Efficiency by Safely Eliminating Load Instruction Execution (ISCA2024 だが arxiv にあったので)
  - ◇ 毎回固定の値を読むロードを削除して性能を上げる. ISCA 2024 best paper.
  - ◇ <https://arxiv.org/pdf/2406.18786>
- Clockhands: Rename-free Instruction Set Architecture for Out-of-order Processors
  - ◇ リネームの必要がない命令セット. 塩谷も共著. MICRO 2023 best paper nominee.
  - ◇ <https://dl.acm.org/doi/10.1145/3613424.3614272>
- これらの論文は, 講義の内容と関係が深いとか, あるいは有名な論文の中から割と単純なものを選んでいます.



# レポート課題

## ■ 形式：

- ◇ 分量は、日本語なら3000文字，英語なら1500ワード程度を目安
  - これより極端に少ない場合は不可です
- ◇ 「自分で」描いた図を2枚以上使って説明すること

## ■ 注意：

- ◇ 元の論文の一部を，ほとんどそのまま転記あるいは翻訳しただけのものを提出しないでください
- ◇ 生成 AI にまとめさせたりしないで，自分で書いてください

# レポート課題

## ■ 締め切り：

- ◇ 今年の秋に卒業予定の人：8/6（火曜）まで
- ◇ そうでない人：8/13（火曜）まで

## ■ 提出方法：

- ◇ 「先進計算機構成論レポート <学籍番号>」 のタイトルで、以下までメールに添付して提出
  - shioya@ci.i.u-tokyo.ac.jp

# 来週 8/5 の講義について

- 8/5 にも講義を実施します
  - ◇ 出席は取らないので、内容に興味ある人は来てください
  - ◇ 仮想メモリや特権モード, Spectre などのアタックの基本などを説明する予定です

# 出欠と感想

- 本日の講義でよくわかったところ, わからなかったところ, 質問, 感想などを書いてください
  - ◇ LMS の出席を設定するので, そこにお願いします
  - ◇ パスワード:
- 意見や内容へのリクエストもあったら書いてください
- LMS の出席の締め切りは来週の講義開始までに設定してあります
  - ◇ 仕様上「遅刻」表示になりますが, 特に減点等しません
  - ◇ 来週の講義開始までは感想や質問などを受け付けます

- データ転送は最もボトルネックに遭遇しやすい場所であり、systolic arrayは演算とI/Oの関係をよりバランス良くすることができると聞いています。

- アクセラレータとは異なり、GPUでは同時実行される演算器の個数に陽に依存せずにプログラムを書ける点がuser friendlyだと感じました

# 質問とか感想

- 各社のGPUは単にソフトウェア上のアーキテクチャが違うだけだと思っていたが、ハードウェアの構造などから根本的に違うのが分かったのは驚きでした。
- 同時に疑問に思ったのですが、一般にNvidiaのGPUが普及している印象があるのにインテルやAMDが自社のアーキテクチャに固執している理由はSIMTの問題点にあるのでしょうか？

- SIMDなどの略語の正しい発音が何なのか、論文を読んでもだけではわからないのが辛いところです



- NVIDIAがGPUの大きなシェアを持つようになったのは、SIMTを採用したのが良かったのか、それともアーキテクチャというよりはソフトウェア含めたエコシステムが充実していたからなのか、どうなのでしょうか？

# 質問とか感想

- SIMTはハードウェアのアーキテクチャで一つのスレッド群に対して、PCが複数並んでいるようにモデル図では見えるのですが、実際は一つのPCを共有して使っている認識でよろしいでしょうか？
- また、SIMTのデメリットである水平方向の演算ができない点について、並列計算であれば基本的にはこの水平方向の演算をすることは少ないように思えるのですがこの性能低下が著しい場合は実際にあるのでしょうか？

- GPU, NPU(Copilot+ PC)のような特化演算器がパーソナルコンピュータでも普及してきていると思うが, TPUやMN-coreのようなサーバー側で使われる特化演算器はなにか他にあるでしょうか？

- 現在開発されている純粋なSIMDプロセッサとしては、MN-Coreがほとんど唯一の例のようなことを話されていたが、なぜそのような状況になっているのかが知りたい
- 純粋なSIMDでは実用的なプログラマビリティを確保するのは難しいのかなと思った

- 講義中に、完成したハードウェアにバグがないかをどのように確かめるかのお話があり、プログラムの答えが一致するかで確かめるとあったが、もし合っていない時にはどのようにバグを治していくのか気になった。gcを実装したことがあるが、バグが出るとdebugにかなり苦労した経験があるので、それと似たようなことになるのかと思った。

- SIMTにおけるbranch divergenceの問題に関連して、一度分岐したスレッドが同じ処理に合流するような場合に正しくWarp内の足並みを揃えるのは単純ではないように思います。
- もちろんWarp内で処理が分岐しないように実装するのがベストではありますが、このような場合に適切に合流させるための命令先読みのような工夫などは存在するのでしょうか。