「情報システム学特別講義３」
プログラム高速化の基礎（その１）

東京大学情報基盤センター准教授　片桐孝洋

２０１４年４月１５日（火）１４：４０－１６：１０

東京大学情報基盤センター
Information Technology Center,The University of Tokyo

東大FX10スーパーコンピュータシステムの利用（講義受講者のみ無料で利用可能）

- 希望者は、基盤センタのスーパコンピュータ（富士通FX10スーパーコンピュータシステム）の利用が、無料でできます。
- 利用希望者は電子メールで、

  名前、所属、学籍番号

  を記載し、

  subject: 電通大講義でのFX10利用申込

  とし

  katagiri@cc.u-tokyo.ac.jp

  まで

  2014年4月18日（金）までに　送ってください。

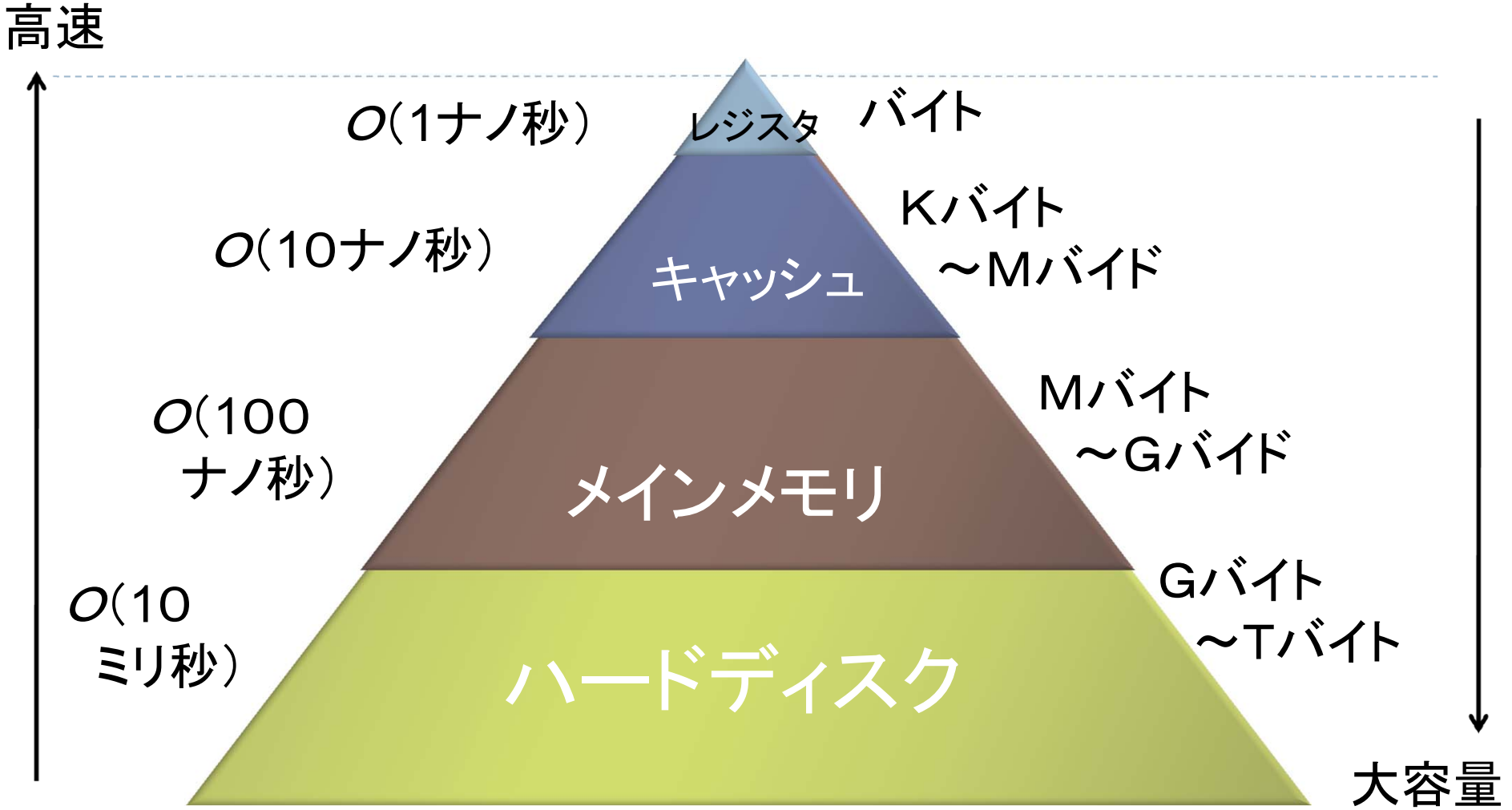
講義日程
（情報システム学特別講義３）

レポートおよびコンテスト課題
（締切：
2014年8月11日（月）24時　厳守

1. ~~4月8日：ガイダンス~~
2. 4月15日
   - プログラム高速化の基礎（その1）
3. 4月22日
   - プログラム高速化の基礎（その2）
4. 5月13日
   - MPIの基礎
5. 5月20日
   - OpenMPの基礎
6. 5月27日
   - Hybrid並列化技法
     （MPIとOpenMPの応用編）
7. 6月3日
   - プログラム高速化の応用
8. 6月10日
   - 行列ーベクトル積の並列化
9. 6月17日
   - べき乗法の並列化
10. 6月24日
    - 行列−行列積の並列化
11. 7月8日
    - LU分解の並列化
12. 7月15日
    - 非同期通信
    - 疎行列反復解法の並列化
13. 7月22日
    - ソフトウェア自動チューニング
14. 8月5日（補講日）
    - エクサフロップスコンピューティング
      に向けて

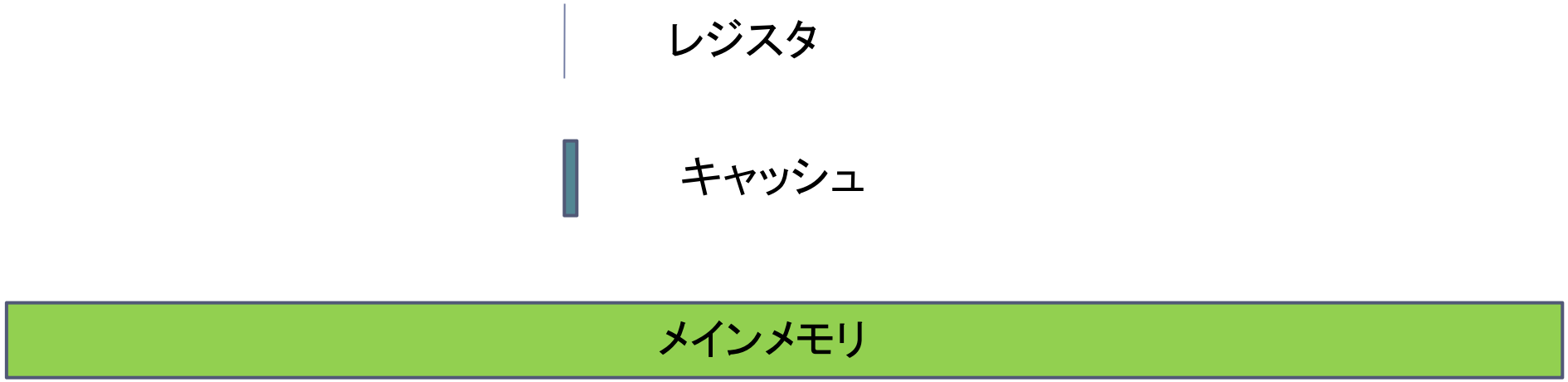
単体（CPU）最適化の方法

最近の計算機のメモリ階層構造



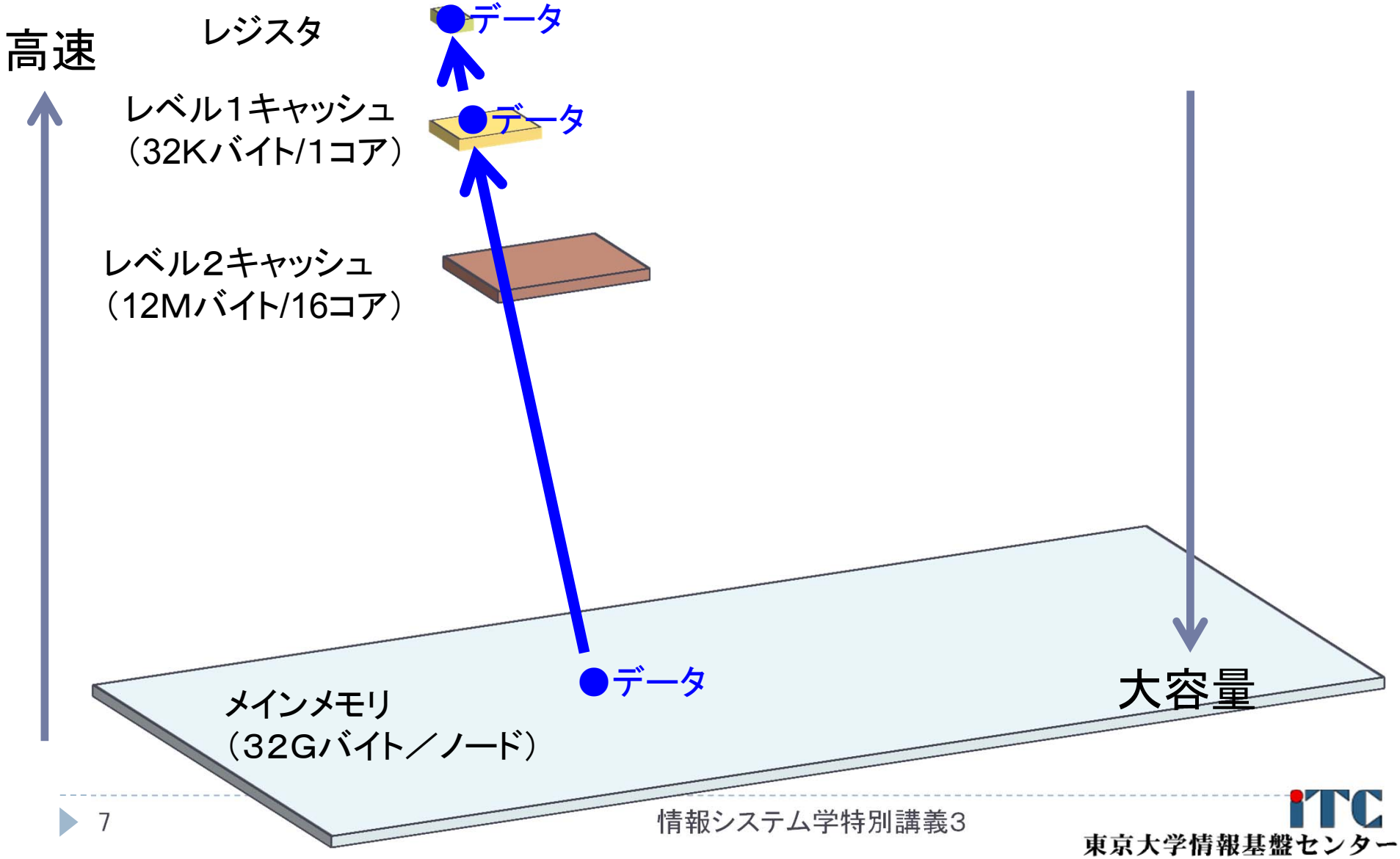
<メインメモリ>→<レジスタ>への転送コストは、
レジスタ上のデータ・アクセスコストの O(100)倍！

より直観的には…

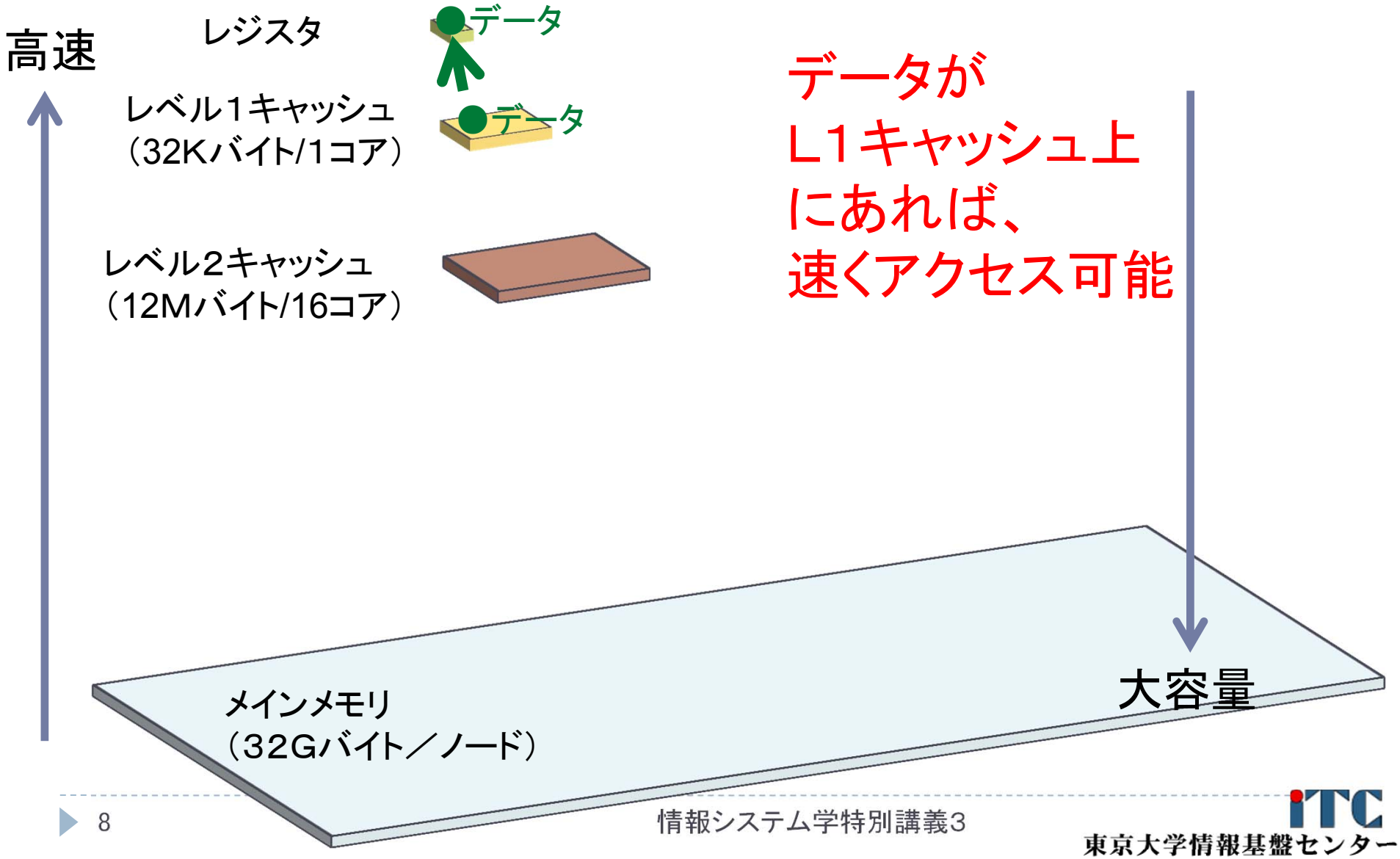


- 高性能（＝速い）プログラミングをするには、
  きわめて小容量のデータ範囲について
  何度もアクセス（＝局所アクセス）するように
  ループを書くしかない

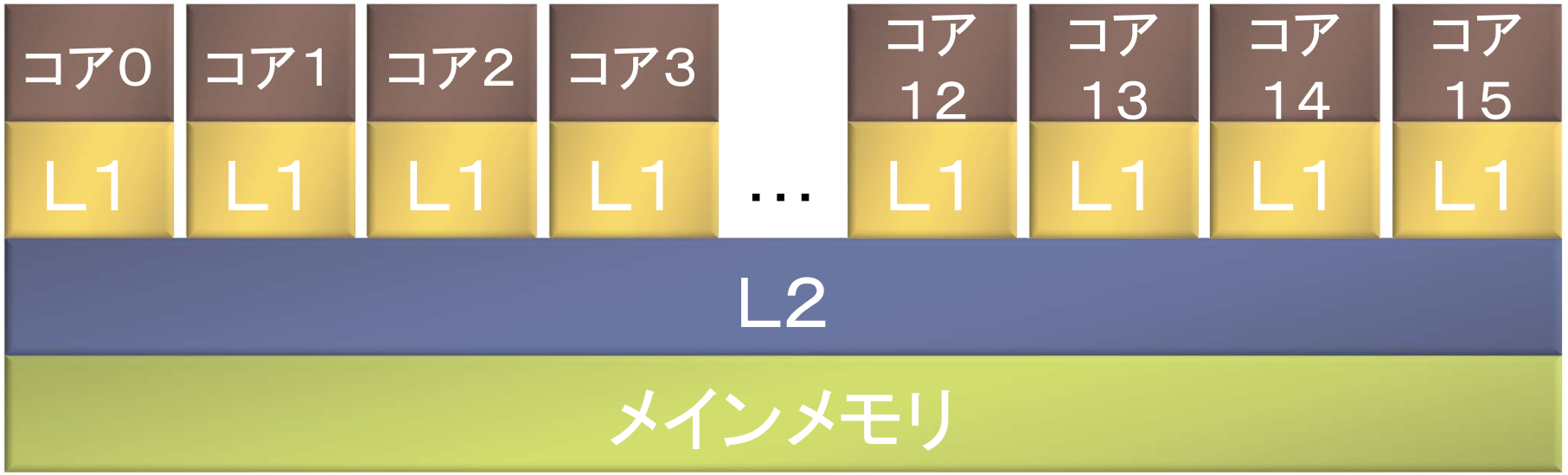
東京大学FX10のメモリ構成例



東京大学FX10のメモリ構成例

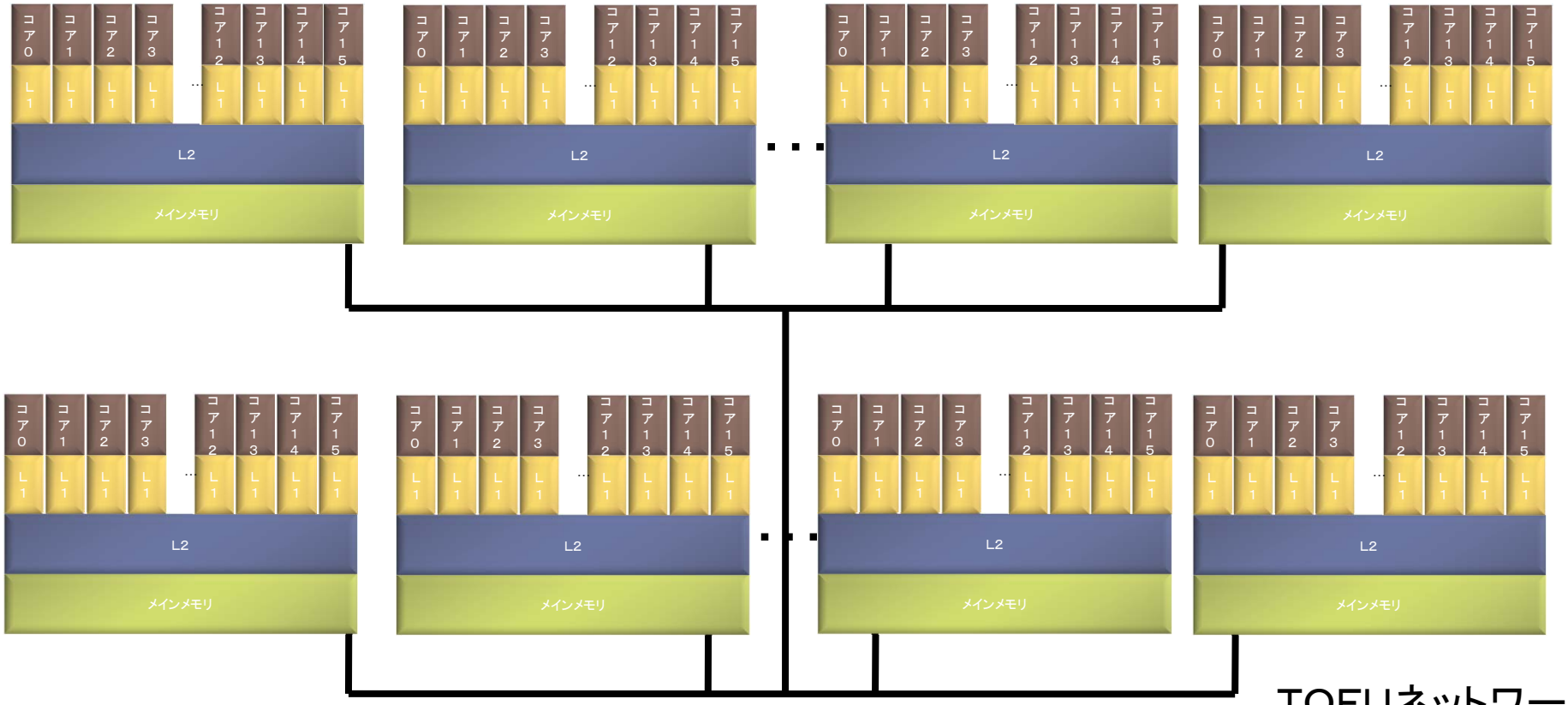


東京大学ＦＸ１０のノードのメモリ構成例



※階層メモリ構成となっている

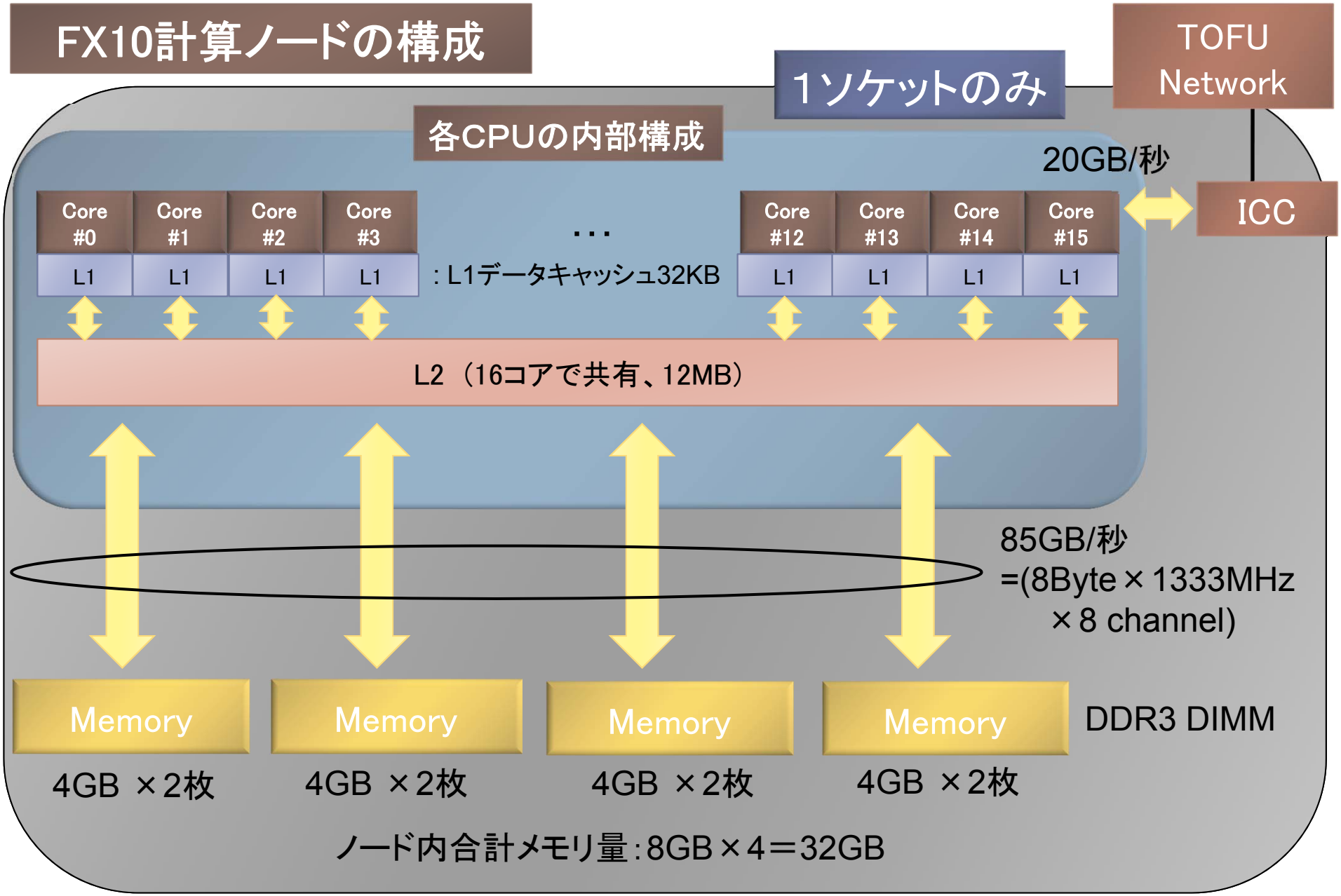
東京大学FX10全体メモリ構成



TOFUネットワーク
（5Gバイト／秒
×双方向）

メモリ階層が階層

FX10計算ノードの構成



東京大学FX10の CPU(SPARC64IXfx)の詳細情報

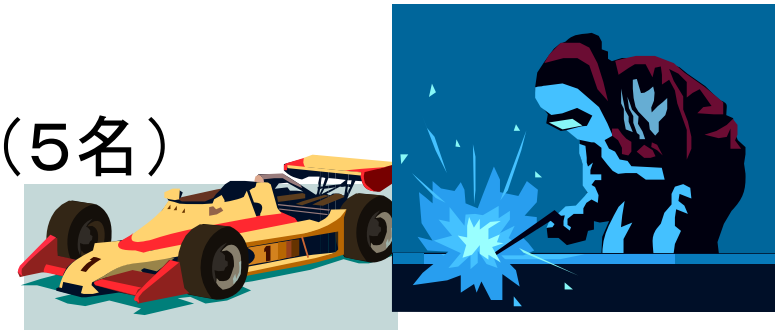
項目	値
アーキテクチャ名	HPC-ACE （SPARC-V9命令セット拡張仕様）
動作周波数	1.848GHz
L1キャッシュ	32 Kbytes （命令、データは分離）
L2キャッシュ	12 Mbytes
ソフトウェア制御キャッシュ	セクタキャッシュ
演算実行	2整数演算ユニット、4つの浮動小数点積和演算ユニット(FMA)
SIMD命令実行	1命令で2つのFMAが動作 FMAは2つの浮動小数点演算(加算と乗算)を実行可能
レジスタ	● 浮動小数点レジスタ数:256本
その他	● 三角関数sin, cosの専用命令 ● 条件付き実行命令 ● 除算、平方根近似命令

演算パイプライン

演算の流れ作業

流れ作業

- 車を作る場合
- １人の作業員１つの工程を担当（５名）

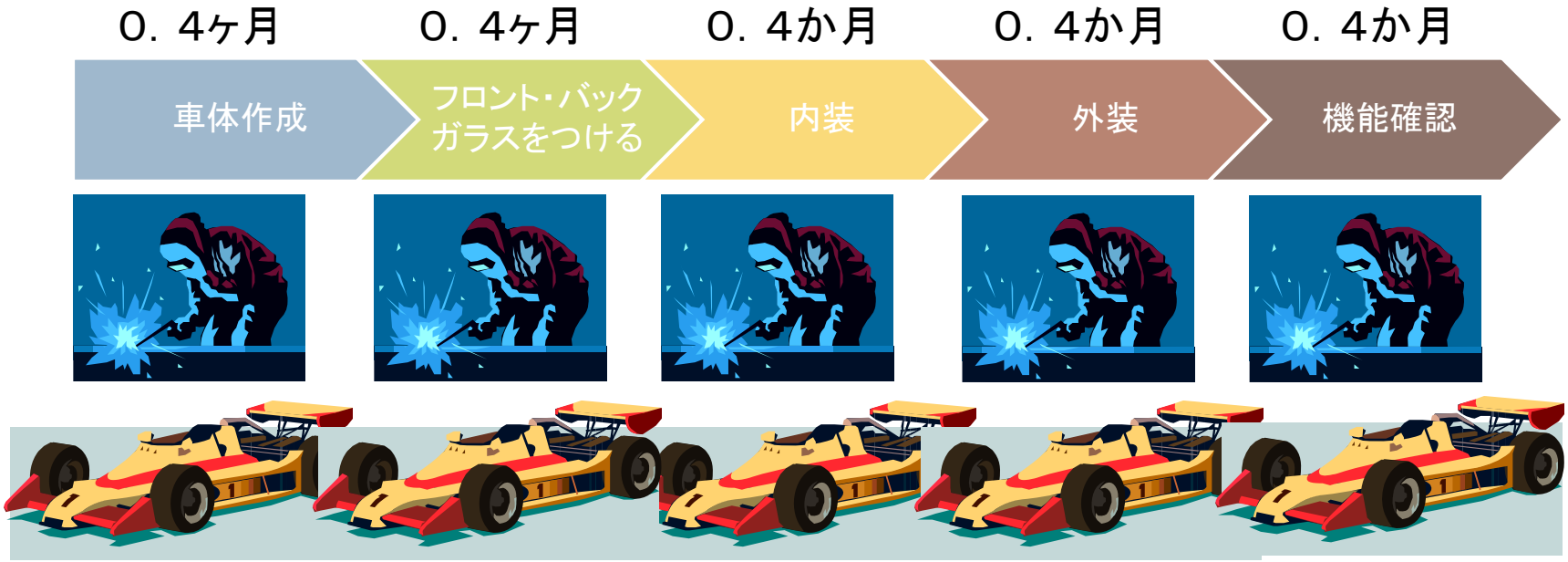


- 上記工程が2ヶ月だとする（各工程は0.4ヶ月とする）
  - 2ヶ月後に1台できる
  - 4ヶ月後に2台できる
  - 2ヶ月／台　の効率

- 各工程の作業員は、
  0．4ヶ月働いて、
  1．6ヶ月は休んでいる
  （＝作業効率が低い）

流れ作業

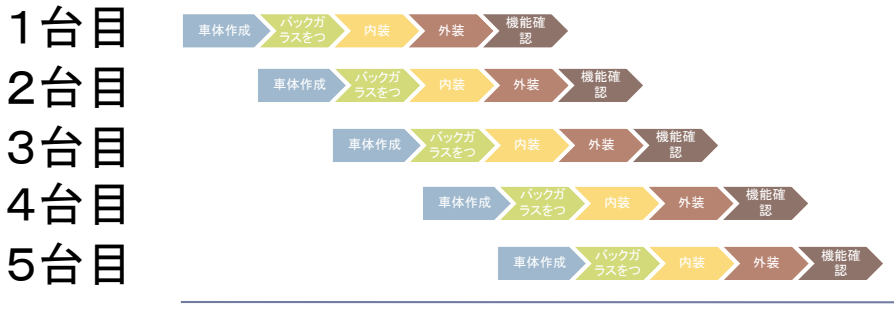
- 作業場所は、5ヶ所とれるとする
- 前の工程からくる車を待ち、担当工程が終わったら、次の工程に速やかに送られるとする
  - ベルトコンベア



流れ作業

- この方法では
  - 2ヶ月後に、1台できる
  - 2．4ヶ月後に、2台できる
  - 2．8ヶ月後に、3台できる
  - 3．2ヶ月後に、4台できる
  - 3．4ヶ月後に、5台できる
  - 3．8ヶ月後に、6台できる
  - 0．63ヶ月／台　の効率

•各作業員は、
十分に時間が立つと
0．4か月の単位時間あたり
休むことなく働いている
（＝作業効率が高い）



•このような処理を、
＜パイプライン処理＞
という

計算機におけるパイプライン処理の形態

1. ハードウエア・パイプライニング
   - 計算機ハードウエアで行う
   - 以下の形態が代表的
     1. 演算処理におけるパイプライン処理
     2. メモリからのデータ（命令コード、データ）転送における
        パイプライン処理
2. ソフトウエア・パイプライニング
   - プログラムの書き方で行う
   - 以下の形態が代表的
     1. コンパイラが行うパイプライン処理
        （命令プリロード、データ・プリロード、データ・ポストストア）
     2. 人手によるコード改編によるパイプライン処理
        （データ・プリロード、ループアンローリング）

演算器の場合

- 例：演算器の工程（注：実際の演算器の計算工程は異なる）

- 行列−ベクトル積の計算では

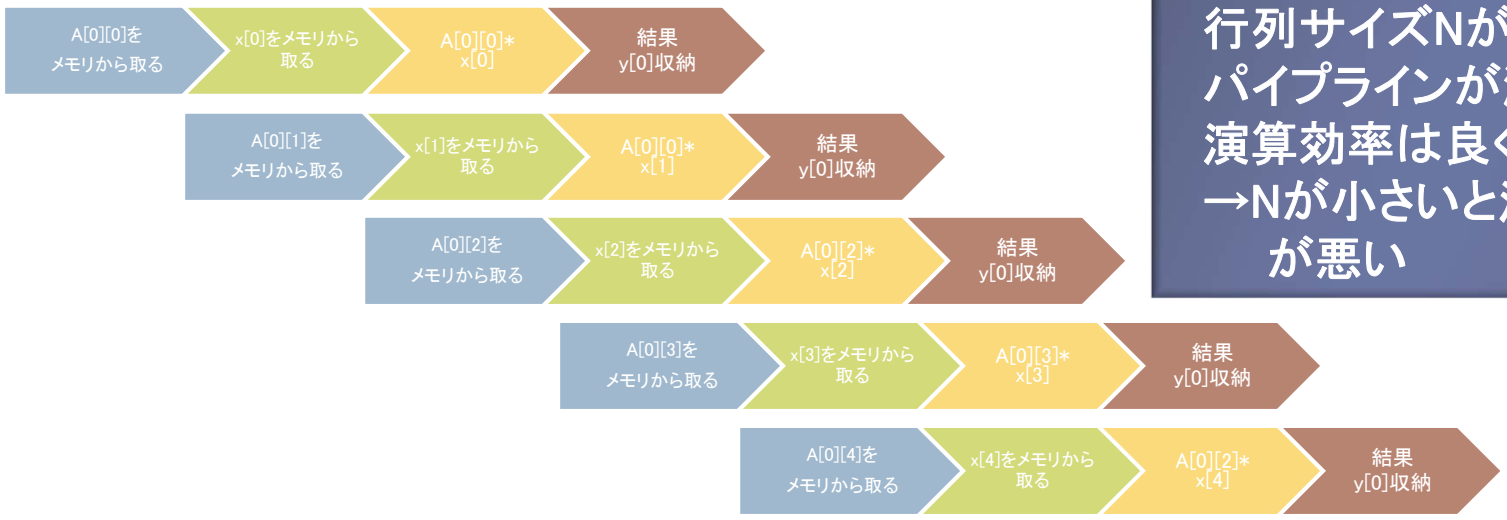
```
for (j=0; j<n;  j++)
    for (i=0; i<n; i++) {
        y[j] += A[j][i] * x[i] ;
}
```

演算器が稼働する工程

- パイプライン化しなければ以下のようになり無駄

演算器の場合

- これでは演算器は、4単位時間のうち、1単位時間しか使われていないので無駄（＝演算効率1／4＝25%）
- 以下のようなパイプライン処理ができれば、十分時間が経つと、毎単位時間で演算がなされる（＝演算効率100%）



●十分な時間とは、十分なループ反復回数があること。行列サイズNが大きいほど、パイプラインが滞りなく流れ、演算効率は良くなる。
→Nが小さいと演算効率が悪い

演算パイプラインのまとめ

- 演算器をフル稼働させるため（＝高性能計算するため）に必要な概念
- メインメモリからデータを取ってくる時間はとても大きい。演算パイプラインをうまく組めば、メモリからデータを取ってくる時間を＜隠ぺい＞できる
  （＝毎単位時間、演算器が稼働した状態にできる）
- 実際は以下の要因があるので、そう簡単ではない
  1. 計算機アーキテクチャの構成による遅延（レジスタ数の制約、メモリ→CPU・CPU→メモリへのデータ供給量制限、など）。
     ※FX10のCPUは＜Sparc 64＞ベースである。
  2. ループに必要な処理（ループ導入変数（i, j）の初期化と加算処理、ループ終了判定処理）
  3. 配列データを参照するためのメモリアドレスの計算処理
  4. コンパイラが正しくパイプライン化される命令を生成するか

実際のプロセッサの場合

- 実際のプロセッサでは
  1. 加減算
  2. 乗算

  ごとに独立したパイプラインがある。
- さらに、同時にパイプラインに流せる命令（同時発行命令）が複数ある。
- Intel Pentium4ではパイプライン段数が31段
  - 演算器がフル稼働になるまでの時間が長い。
  - 分岐命令、命令発行予測ミスなど、パイプラインを中断させる処理が多発すると、演算効率がきわめて悪くなる。
  - 近年の周波数の低い（低電力な）マルチコアCPU／メニーコアCPUでは、パイプライン段数が少なくなりつつある（Xeon Phiは7段）

FX10のハードウエア情報

- 1クロックあたり、8回の演算ができる
  - FMAあたり、乗算および加算　が2つ
    （4つの浮動小数点演算）
  - 1クロックで、2つのFMAが動作
  - 4浮動小数点演算×2FMA＝8浮動小数点演算／クロック
- 1コア当たり1.848GHzのクロックなので、
  - 理論最大演算は、
    1.848 GHz* 8回 = 14.784 GFLOPS / コア
  - 1ノード16コアでは、
    14.784 * 16コア = 236.5 GFLOPS / ノード
- レジスタ数（浮動小数点演算用）
  - 256個 / コア

ループ内連続アクセス

単体最適化のポイント

- 配列のデータ格納方式を考慮して、連続アクセスすると速い（ループ内連続アクセス）

NG

```
for (i=0; i<n; i++) {
    a[ i ][1] = b[ i ] * c[ i ];
}
```

OK

```
for (i=0; i<n; i++) {
    a[1][ i ] = b[ i ] * c[ i ];
}
```

- ループを細切れにし、データアクセス範囲をキャッシュ容量内に収めると速い(ただしnが大きいとき)（キャッシュブロック化）

NG

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

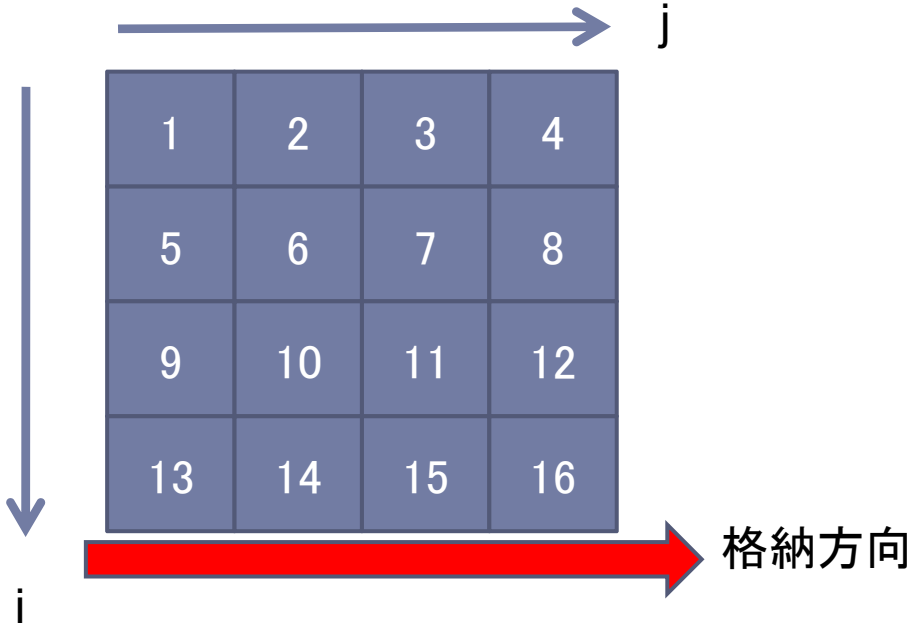
OK

```
for (jb=0; jb<n; jb+=m)
  for (i=0; i<n; i++) {
    for (j=jb; j<jb+m; j++) {
        a[ i ][ j ] = b[ j ] * c[ j ];
  } } }
```

言語に依存した配列の格納方式の違い

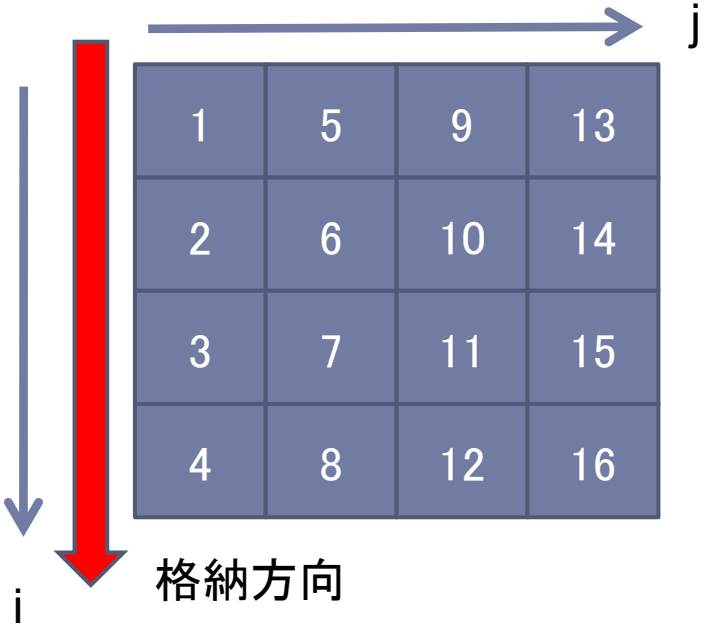
- C言語の場合

A[i][j]



- Fortran言語の場合

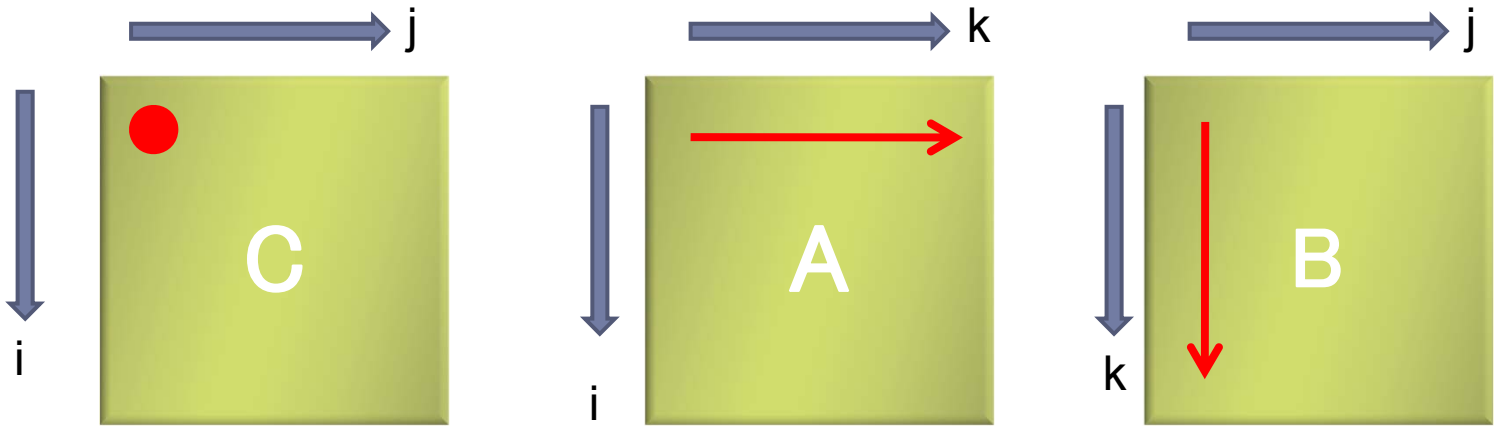
A(i, j)



行列積コード例（C言語）

- コード例

```
for (i=0; i<n;  i++)
  for (j=0; j<n; j++)
    for (k=0; k<n; k++)
      C[i][j] += A[i][k] *B[k][j];
```



行列の積

- 行列積 $c_{ij} = \sum_{k=1}^{n} a_{ik} b_{kj} \ (i, j = 1, 2, \ldots, n)$

  の実装法は、次の二通りが知られている:

1. ループ交換法
   - 連続アクセスの方向を変える目的で、行列−行列積を実現する3重ループの順番を交換する
2. ブロック化（タイリング）法
   - キャッシュにあるデータを再利用する目的で、あるまとまった行列の部分データを、何度もアクセスするように実装する

行列の積

- ループ交換法
  - 行列積のコードは、以下のような3重ループになる（C言語）

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

  - 最内部の演算は、外側の3ループを交換しても、計算結果が変わらない
    → 6通りの実現の方法がある

行列の積

- ループ交換法
  - 行列積のコードは、以下のような3重ループになる（Fortran言語）

```
do i=1, n
   do j=1, n
      do k=1, n
           c( i , j ) = c( i, j) + a( i , k ) * b( k , j )
      enddo
   enddo
enddo
```

  - 最内部の演算は、外側の3ループを交換しても、計算結果が変わらない
    → 6通りの実現の方法がある

行列の積

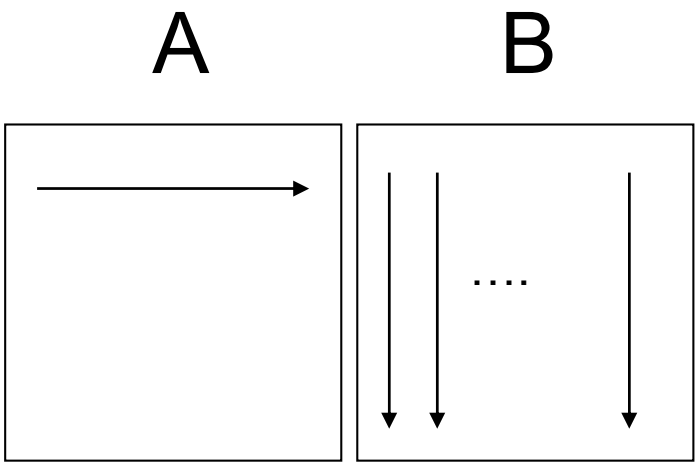
- 行列データへのアクセスパターンから、以下の3種類に分類できる
  1. 内積形式 (inner-product form)
     最内ループのアクセスパタンが
     ＜ベクトルの内積＞と同等
  2. 外積形式 (outer-product form)
     最内ループのアクセスパタンが
     ＜ベクトルの外積＞と同等
  3. 中間積形式 (middle-product form)
     内積と外積の中間

行列の積

- 内積形式 (inner-product form)
  - ijk, jikループによる実現(C言語)

```
for (i=0;  i<n;  i++) {
   for (j=0;  j<n;  j++) {
      dc = 0.0;
      for (k=0;  k<n;  k++) {
         dc = dc + A[ i ][ k ] * B[ k ][ j ];
      }
      C[ i ][ j ]= dc;
   }
}
```

※以降、最外のループからの変数の順番で実装法を呼ぶ。たとえば上記のコードは<ijkループ>。



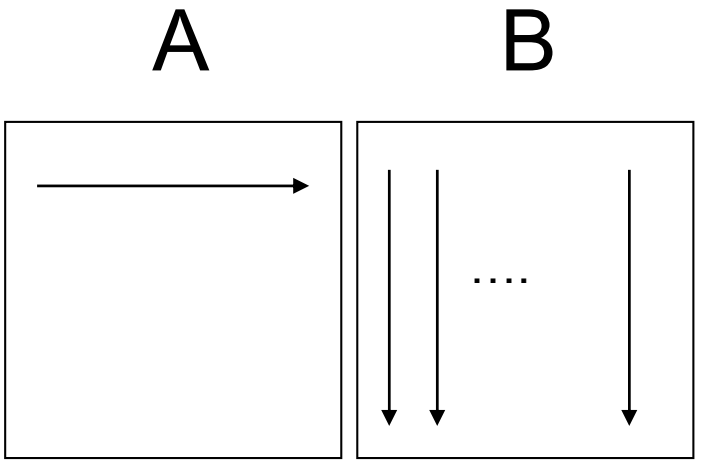
●行方向と列方向のアクセスあり
→行方向・列方向格納言語の
両方で性能低下要因
解決法:
A, Bどちらか一方を転置しておく
(ただし、データ構造の変更ができる場合)

行列の積

- 内積形式 (inner-product form)
  - ijk, jikループによる実現(Fortran言語)

```
do i=1, n
   do j=1, n
      dc = 0.0d0
      do k=1, n
         dc = dc + A( i , k ) * B( k , j )
      enddo
      C( i , j ) = dc
   enddo
enddo
```

※以降、最外のループからの変数の順番で実装法を呼ぶ。たとえば上記のコードは＜ijkループ＞。

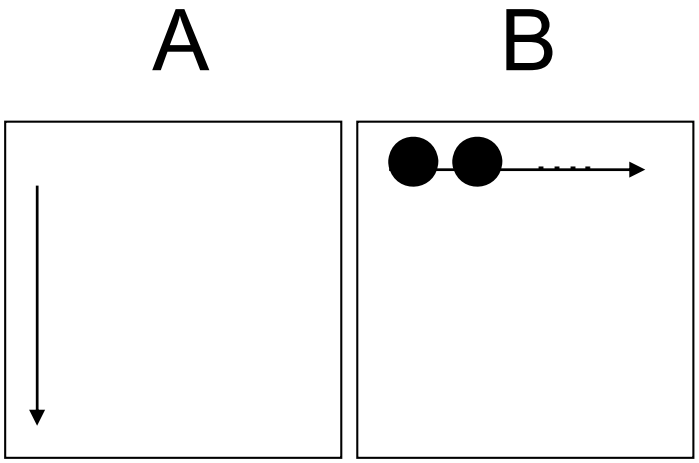


●行方向と列方向のアクセスあり
→行方向・列方向格納言語の
両方で性能低下要因
解決法:
A, Bどちらか一方を転置しておく
(ただし、データ構造の変更ができる場合)

行列の積

- 外積形式 (outer-product form)
  - kij, kjiループによる実現（C言語）

```
for (i=0;  i<n;  i++) {
   for (j=0;  j<n;  j++) {
      C[ i ][ j ] = 0.0;
   }
}
for (k=0;  k<n;  k++) {
  for (j=0;  j<n;  j++) {
     db = B[ k ][ j ];
     for (i=0;  i<n;  i++) {
     C[ i ][ j ]= C[ i ][ j ]+ A[ i ][ k ]* db;
     }
  }
}
```

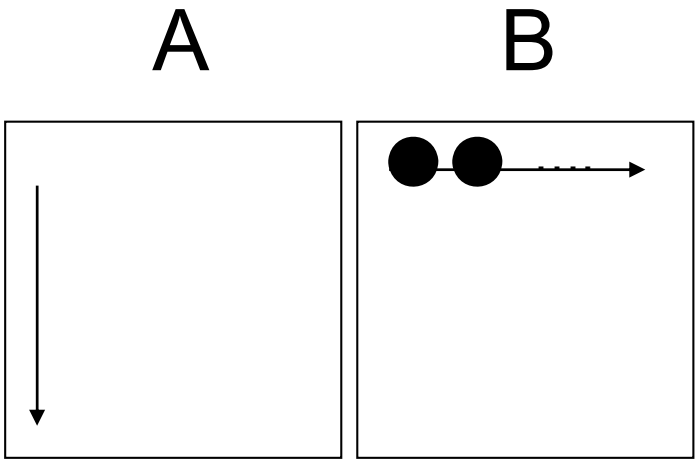


●kjiループでは
列方向アクセスがメイン
→列方向格納言語向き
（Fortran言語）

行列の積

- 外積形式 (outer-product form)
  - kij, kjiループによる実現（Fortran言語）

```
do i=1, n
  do j=1, n
    C( i , j ) = 0.0d0
  enddo
enddo
do k=1, n
  do j=1, n
    db = B( k , j )
    do i=1, n
      C( i , j ) = C( i , j )+ A( i , k ) * db
    enddo
  enddo
enddo
```

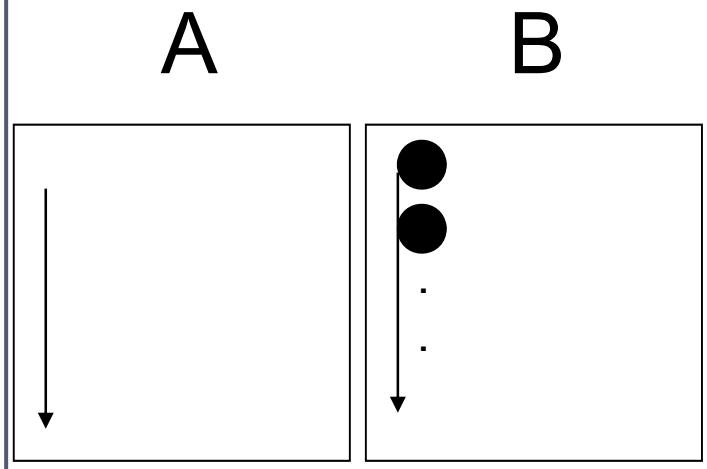


●kjiループでは
列方向アクセスがメイン
→列方向格納言語向き
（Fortran言語）

行列の積

- 中間積形式 (middle-product form)
  - ikj, jkiループによる実現(C言語)

```
for (j=0;  j<n;  j++)  {
    for (i=0;  i<n;  i++) {
        C[ i ][ j ] = 0.0;
    }
    for (k=0;  k<n;  k++) {
      db = B[ k ][ j ];
      for (i=0;  i<n;  i++) {
        C[ i ][ j ] = C[ i ][ j ] + A[ i ][ k ] * db;
      }
    }
}
```

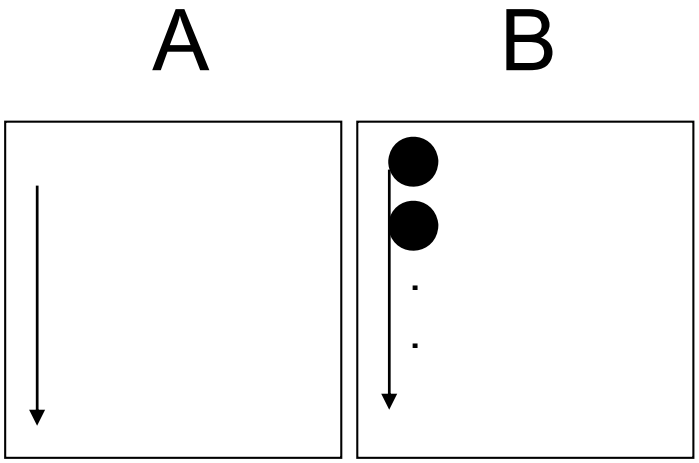


●jkiループでは
全て列方向アクセス
→列方向格納言語に
最も向いている
(Fortran言語)

行列の積

- 中間積形式 (middle-product form)
  - ikj, jkiループによる実現(Fortran言語)

```
do j=1, n
  do i=1, n
    C( i , j ) = 0.0d0
  enddo
  do k=1, n
    db = B( k , j )
    do i=1, n
      C( i , j ) = C( i , j ) + A( i , k ) * db
    enddo
  enddo
enddo
```



●jkiループでは
全て列方向アクセス
→列方向格納言語に
最も向いている
(Fortran言語)

ループアンローリング

ループアンローリング

- コンパイラが、
  1. レジスタへのデータの割り当て;
  2. パイプライニング;

  がよりできるようにするため、コードを書き換えるチューニング技法
- ループの刻み幅を、1ではなく、mにする
  - ＜m段アンローリング＞とよぶ

ループアンローリングの例 （行列-行列積、Ｃ言語）

- k-ループ2段展開 （nが2で割り切れる場合）

```
for (i=0; i<n;  i++)
  for (j=0; j<n; j++)
    for (k=0; k<n; k+=2)
      C[i][j] += A[i][k] *B[k][ j] + A[i][k+1]*B[k+1][ j];
```

- k-ループのループ判定回数が1/2になる。

ループアンローリングの例（行列-行列積、C言語）

- j-ループ2段展開（nが2で割り切れる場合）

```
for (i=0; i<n;  i++)
  for (j=0; j<n; j+=2)
    for (k=0; k<n; k++) {
      C[i][ j   ] += A[i][k] *B[k][ j   ];
      C[i][ j+1] += A[i][k] *B[k][ j+1];
    }
```

- A[i][k]をレジスタに置き、高速にアクセスできるようになる。

ループアンローリングの例（行列-行列積、C言語）

- i-ループ2段展開（nが2で割り切れる場合）

```
for (i=0; i<n;  i+=2)
  for (j=0; j<n; j++)
    for (k=0; k<n; k++)  {
      C[i   ][j] += A[i   ][k] *B[k][j];
      C[i+1][j] += A[i+1][k] *B[k][j];
    }
```

- B[i][j]をレジスタに置き、高速にアクセスできるようになる。

ループアンローリングの例（行列-行列積、C言語）

- i-ループ、および j-ループ 2段展開
  （nが2で割り切れる場合）

```
for (i=0; i<n;  i+=2)
  for (j=0; j<n; j+=2)
    for (k=0; k<n; k++)  {
      C[i   ][ j   ] += A[i   ][k] *B[k][ j   ];
      C[i   ][ j+1] += A[i   ][k] *B[k][ j+1];
      C[i+1][ j   ] += A[i+1][k] *B[k][ j   ];
      C[i+1][ j+1] += A[i+1][k] *B[k][ j +1];
    }
```

- A[i][j], A[i+1][k],B[k][j],B[k][j+1]をレジスタに置き、高速にアクセスできるようになる。

ループアンローリングの例（行列-行列積、C言語）

- コンパイラにわからせるため、以下のように書く方がよい場合がある

```
for (i=0; i<n;  i+=2)
  for (j=0; j<n; j+=2) {
    dc00 = C[i   ][ j   ];  dc01 = C[i   ][ j+1];
    dc10 =  C[i+1][ j   ];  dc11 = C[i+1][ j+1] ;
    for (k=0; k<n; k++)  {
      da0= A[i   ][k] ; da1= A[i+1][k] ;
      db0= B[k][ j   ]; db1= B[k][ j+1];
      dc00 += da0 *db0;  dc01 += da0 *db1;
      dc10 += da1 *db0;  dc11 += da1 *db1;
    }
    C[i   ][ j   ] = dc00;  C[i   ][ j+1] = dc01;
    C[i+1][ j   ] = dc10;   C[i+1][ j+1] = dc11;
  }
```

ループアンローリングの例（行列-行列積、Fortran言語）

- k-ループ2段展開（nが2で割り切れる場合）

```
do i=1, n
  do j=1, n
    do k=1, n, 2
      C(i, j) = C(i, j) +A(i, k) *B(k, j) + A(i, k+1)*B(k+1, j)
    enddo
  enddo
enddo
```

- k-ループのループ判定回数が1/2になる。

ループアンローリングの例（行列-行列積、Fortran言語）

- j-ループ2段展開（nが2で割り切れる場合）

```
do i=1, n
  do j=1, n, 2
    do k=1, n
      C(i, j   ) = C(i, j   ) +A(i, k) * B(k, j   )
      C(i, j+1) = C(i, j+1)  +A(i, k) * B(k, j+1)
    enddo
  enddo
enddo
```

- A(i, k)をレジスタに置き、高速にアクセスできるようになる。

ループアンローリングの例（行列-行列積、Fortran言語）

- i-ループ2段展開（nが2で割り切れる場合）

```
do i=1, n, 2
  do j=1, n
    do k=1, n
      C(i   , j) = C(i   , j)  +A(i   , k) * B(k , j)
      C(i+1, j) = C(i+1, j)  +A(i+1, k) * B(k , j)
    enddo
  enddo
enddo
```

- B(i, j)をレジスタに置き、高速にアクセスできるようになる。

ループアンローリングの例 （行列-行列積、Fortran言語）

- i-ループ、および j-ループ 2段展開
  （nが2で割り切れる場合）

```
do i=1, n,  2
  do j=1, n,  2
    do k=1,  n
      C(i   , j   ) = C(i   , j   ) +A(i   , k) *B(k, j   )
      C(i   , j+1) = C(i   , j+1) +A(i   , k) *B(k, j+1)
      C(i+1, j   ) = C(i+1, j   ) +A(i+1, k) *B(k, j   )
      C(i+1, j+1) =C(i+1, j+1) +A(i+1, k) *B(k, j +1)
    enddo;  enddo;  enddo;
```

- A(i,j), A(i+1,k),B(k,j),B(k,j+1)をレジスタに置き、
  高速にアクセスできるようになる。

ループアンローリングの例（行列-行列積、Fortran言語）

- コンパイラにわからせるため、以下のように書く方がよい場合がある

```
do i=1, n, 2
 do j=1, n, 2
   dc00 = C(i   ,j   ); dc01 = C(i   ,j+1)
   dc10 = C(i+1,j   ); dc11 = C(i+1,j+1)
   do k=1, n
     da0= A(i   ,k); da1= A(i+1, k)
     db0= B(k   ,j ); db1= B(k,   j+1)
     dc00 = dc00+da0 *db0; dc01 = dc01+da0 *db1;
     dc10 = dc10+da1 *db0; dc11 = dc11+da1 *db1;
   enddo
   C(i   , j   ) = dc00; C(i   , j+1) = dc01
   C(i+1, j   ) = dc10; C(i+1, j+1) = dc11
 enddo; enddo
```

キャッシュライン衝突

とびとびアクセスは弱い

不連続アクセスとは

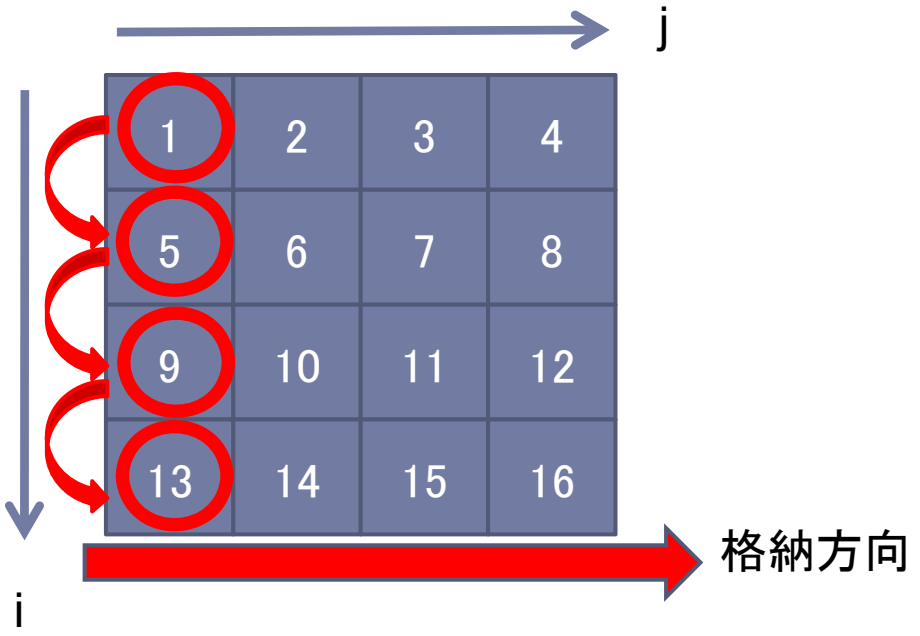
- 配列のデータ格納方式を考慮し連続アクセスすると速い（ループ内連続アクセス）

NG

```
for (i=0; i<n; i++) {
    a[ i ][1] = b[ i ] * c[ i ];
}
```

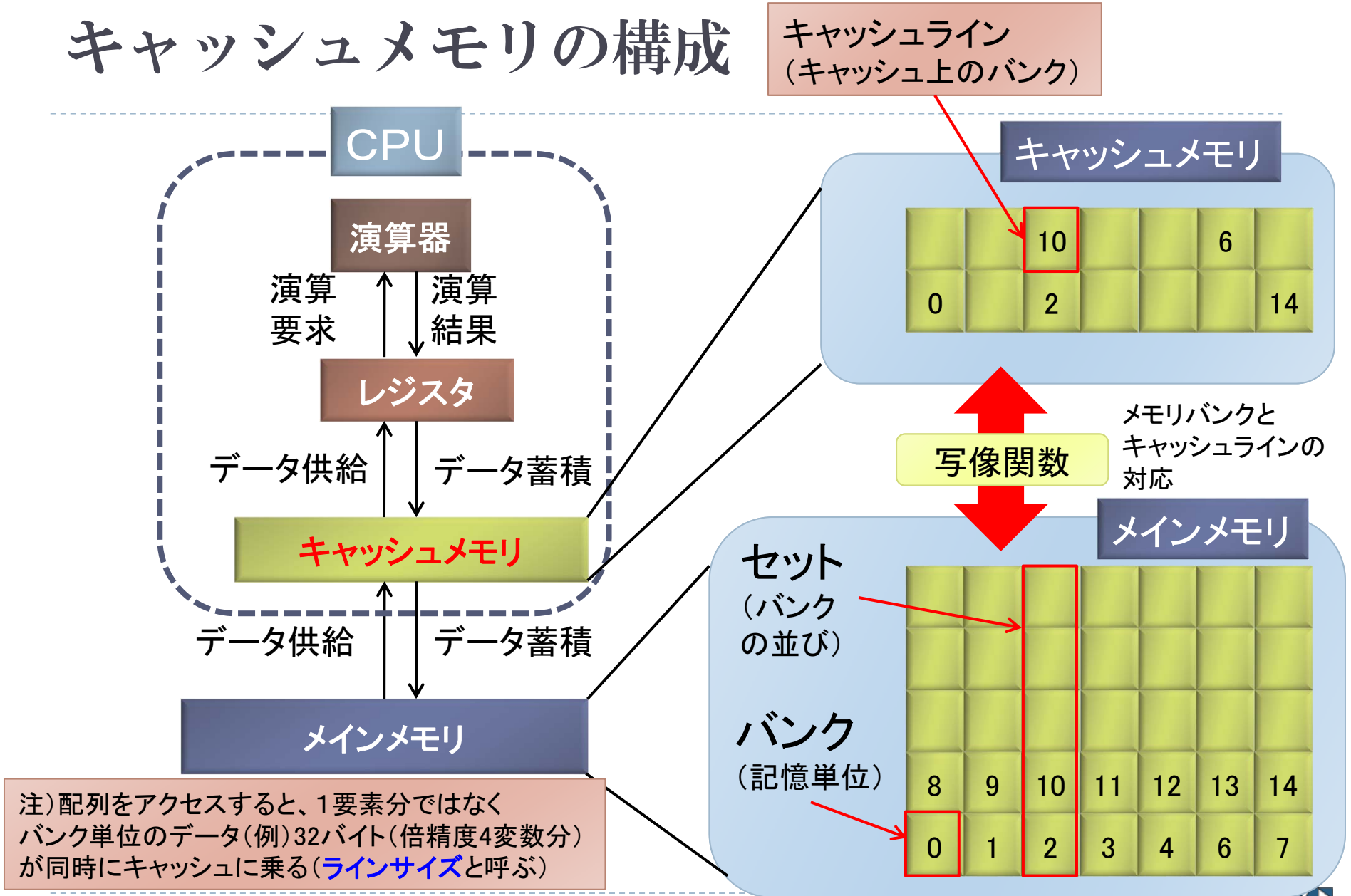
- C言語の場合

a[i] [j]



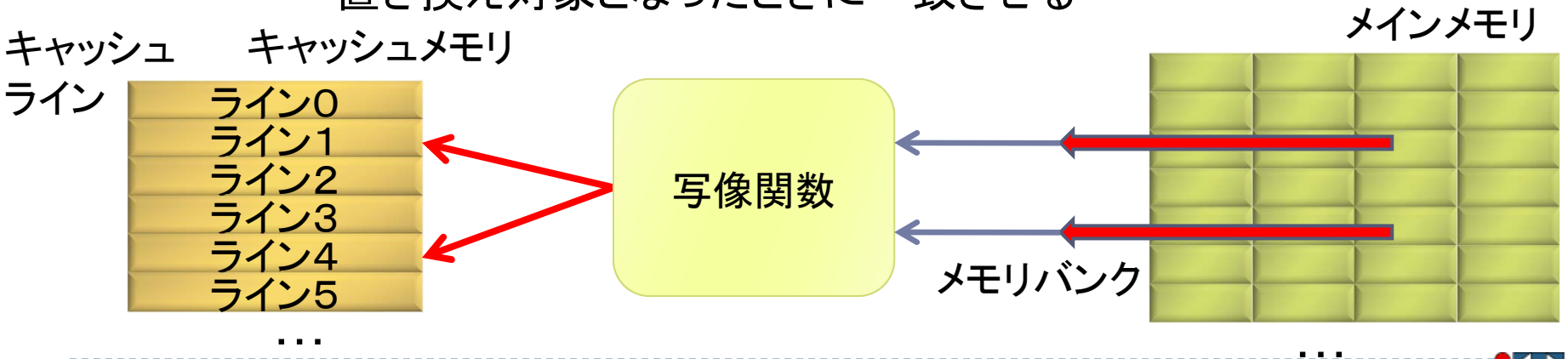
間隔4での不連続アクセス

キャッシュメモリの構成



キャッシュとキャッシュライン

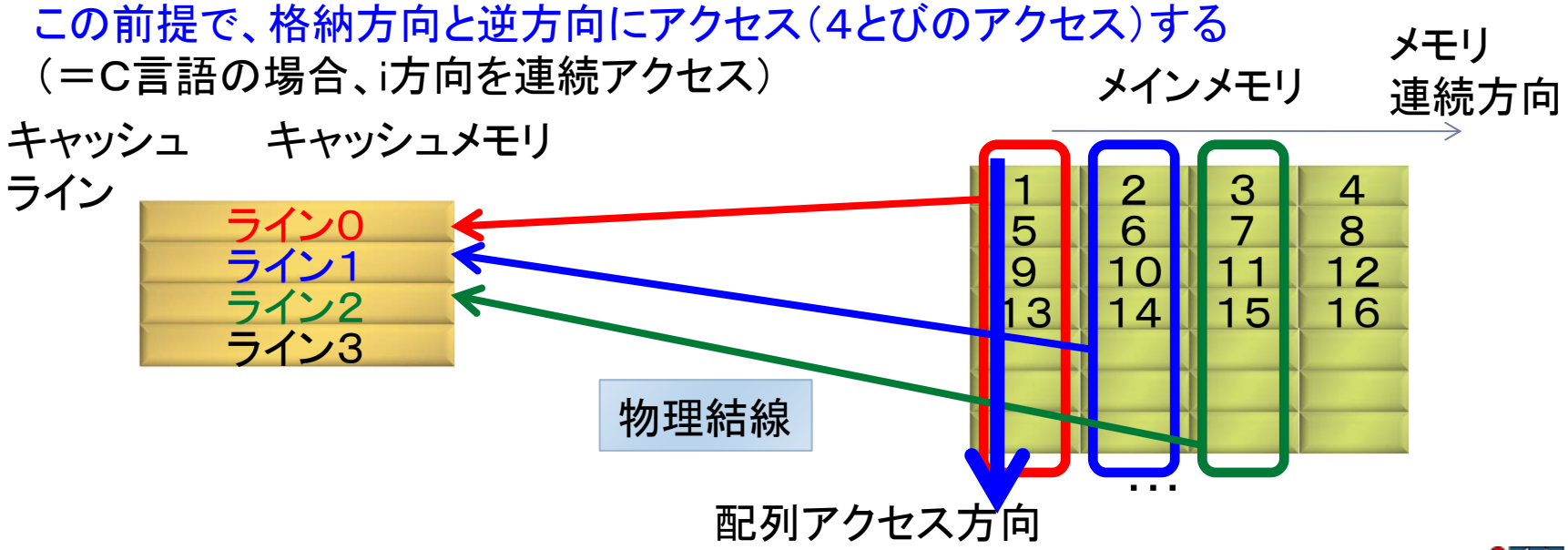
- メインメモリ上とキャッシュ上のデータマッピング方式
  - 読み出し： メインメモリ から キャッシュ へ
    - ダイレクト・マッピング方式： メモリバンクごとに直接的
    - セット・アソシアティブ方式： ハッシュ関数で写像（間接的）
  - 書き込み： キャッシュ から メインメモリ へ
    - ストア・スルー方式： キャッシュ書き込み時に
      メインメモリと中身を一致させる
    - ストア・イン方式： 対象となるキャッシュラインが
      置き換え対象となったときに一致させる



キャッシュライン衝突の例

- 直接メインメモリのアドレスをキャッシュに写像する、ダイレクト・マッピングを考える
  - 物理結線は以下の通り
- マッピング間隔を、ここでは4とする
  - メインメモリ上のデータは、間隔4ごとに、同じキャッシュラインに乗る
- キャッシュラインは8バイト、メモリバンクも8バイトとする
- 配列aは 4×4の構成で、倍精度（8バイト）でメモリ確保されているとする

  double a[4][4];
- この前提で、格納方向と逆方向にアクセス（4とびのアクセス）する
  （＝C言語の場合、i方向を連続アクセス）



キャッシュライン衝突の例

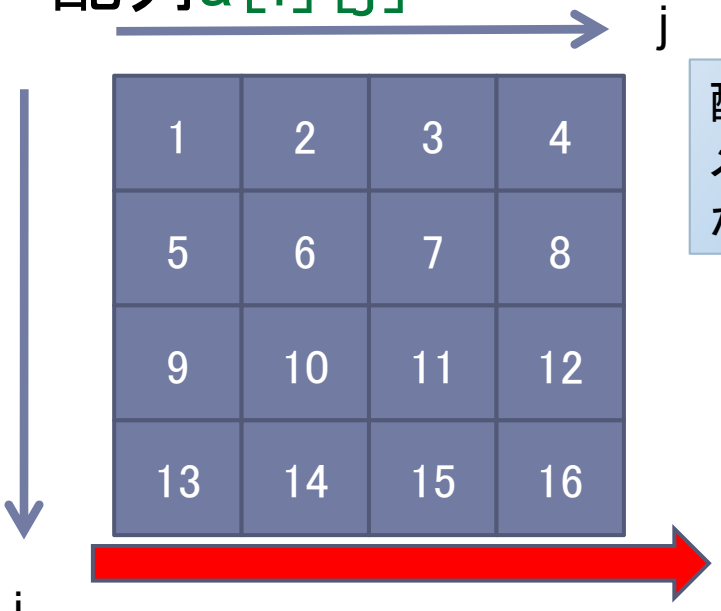
- この前提の、＜実際の配列構成＞と＜メモリバンク＞の関係

実際は、以下のことがあるので、必ずしも、こうならないことに注意する

- 配列a□□の物理メモリ上の配置はOSが動的に決定するので、ずれることがある
- メモリバンクの容量は、8バイトより大きい
- ダイレクト・マッピングではない

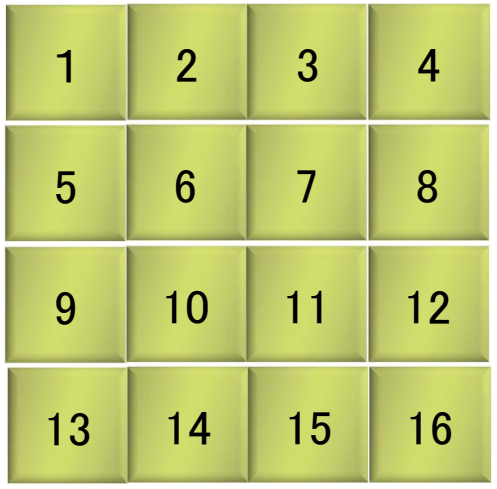
- C言語の場合

配列a[i][j]



配列要素a□□ とメモリバンク構造とが完全一致

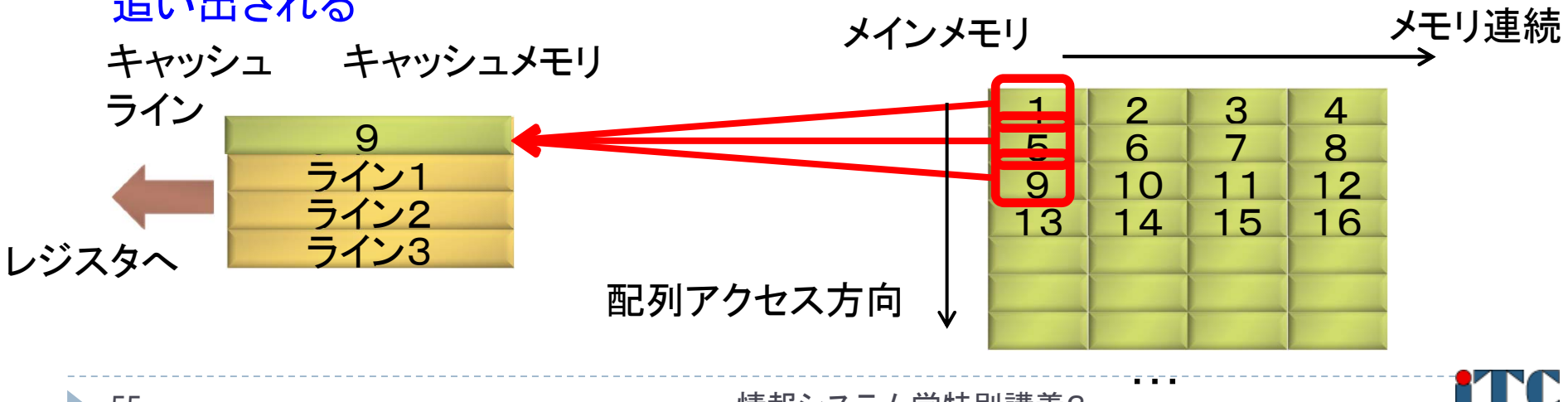
メインメモリ上のバンク構成



キャッシュライン衝突の例

1. a[0][0]があるバンク1がキャッシュライン0に乗る
2. すぐに、a[1][0]があるバンク5がアクセスされる
3. （物理結線先のキャッシュライン0に容量の空きがないので）
   キャッシュライン0のデータ(バンク1の内容)を追い出さないといけない
4. バンク5のデータがキャッシュライン0に乗る
5. すぐに、a[2][0]があるバンク9がアクセスされる
6. キャッシュライン0のデータ（バンク5の内容）を追い出さないといけない

…玉突きで、ライン1～3が空いていても、逐次的にキャッシュ上のデータが追い出される



キャッシュライン衝突の例

- １～６の状態が連続して発生する。

➡ メモリ→キャッシュの回線が常に稼働

  - ＜回線お話し中＞で、データが来るのが終わるまで、待たされる
    （回線レベルで並列にデータが持ってこれない）
  - ストア・イン方式では、メモリにデータを書き戻すコストもかかる

- メモリからデータを逐次で読み出すのと同じ

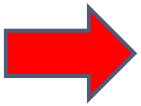
➡ ＜キャッシュがない＞のと同じ

➡ 演算器にデータが届かないので計算を中断。
演算器の利用効率が悪くなる

以上の現象を＜キャッシュライン衝突＞と呼ぶ

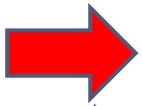
メモリ・インターリービング

- 物理的なメモリの格納方向に従いアクセスする時
  - データアクセス時、現在アクセス中のバンク上のデータは、周辺バンク上のデータも一括して（同時に）、別のキャッシュライン上に乗せるハードウェア機能がある

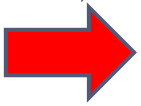


キャッシュライン0のデータをアクセスしている最中に、キャッシュライン1に近隣のバンク内データを（並列に）持ってくることが可能

メモリの＜インタリービング＞



演算機から見たデータアクセス時間が短縮



演算器が待つ時間が減少（＝演算効率が上がる）

物理的なデータ格納方向に連続アクセスするとよい

キャッシュライン衝突が起こる条件

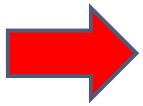
- メモリバンクのキャッシュラインへの割り付けは２冪の間隔で行っていることが多い
  - たとえば、32、64、128など
- 特定サイズの問題（たとえば1024次元）で、性能が1／2～1／3、ときには1／10になる場合、キャッシュライン衝突が生じている可能性あり

NG

```
double a[1024][1024];
```

```
double precision a(1024, 1024)
```

実際は、OSやキャッシュ構成の影響で厳密な条件を見つけることは難しいが



2冪サイズでの配列確保は避けるべき

キャッシュライン衝突への対応

- キャッシュライン衝突を防ぐ方法
  1. パティング法：配列に（2冪でない）余分な領域を確保し確保配列の一部の領域を使う。
     - 余分な領域を確保して使う
       - 例：double A[1024][1025]; で1024のサイズをアクセス
     - コンパイラのオプションを使う
  2. データ圧縮法：　計算に必要なデータのみキャッシュライン衝突しないようにデータを確保し、かつ、必要なデータをコピーする。
  3. 予測計算法：　キャッシュライン衝突が起こる回数を予測するルーチンを埋め込み、そのルーチンを配列確保時に呼ぶ。

レポート課題（その１）

- 問題レベルを以下に設定

問題のレベルに関する記述：
- L00: きわめて簡単な問題。
- L10: ちょっと考えればわかる問題。
- L20: 標準的な問題。
- L30: 数時間程度必要とする問題。
- L40: 数週間程度必要とする問題。複雑な実装を必要とする。
- L50: 数か月程度必要とする問題。未解決問題を含む。

※L40以上は、論文を出版するに値する問題。

- 教科書のサンプルプログラムは以下が利用可能
  - Sample-fx.tar
  - Mat-Mat-noopt-fx.tar
  - Mat-Vec-fx.tar
  - Mat-Mat-fx.tar

レポート課題（その２）

1. [L10] 利用できる計算機で、行列−行列積について、メモリ連続アクセスとなる場合と、不連続となる場合の性能を調査せよ。
2. [L15] 行列−行列積のアンローリングを、i, j, k ループについて施し、性能向上の度合いを調べよ。どのアンローリング方式や段数が高速となるだろうか。
3. [L10] FX10のCPUである、SPARC64 IXfxの計算機アーキテクチャについて調べよ。
   特に、演算パイプラインの構成や、演算パイプラインに関連するマシン語命令について調べよ。

レポート課題（その３）

4. [L15]　利用できる計算機で、ブロック化を行った行列−行列積のコードに対し、アンローリングを各ループについて施し性能を調査せよ。行列の大きさ(N)を変化させ、各Nに対して適切なアンローリング段数を調査せよ。
5. [L5]　身近にある計算機の、キャッシュサイズと、その構造を調べよ。
6. [L5]　身近にある計算機の、命令レベル並列性の実装の仕組みを調べよ。
7. [L5] 本講義で取り扱っていないチューニング手法を調べよ。