

内容に関する質問は
katagiri@cc.u-tokyo.ac.jp
まで

第1回 プログラム高速化の基礎

東京大学情報基盤センター 片桐孝洋

本講義の位置づけ

講義日程と内容について

- ▶ 2013年度 計算科学技術特論A(1学期:木曜3限)
 - ▶ 第1回:プログラム高速化の基礎、2013年4月11日
 - ▶ イントロダクション、ループアンローリング、キャッシュブロック化、数値計算ライブラリの利用、その他
 - ▶ 第2回:MPIの基礎、2013年4月18日
 - ▶ 並列処理の基礎、MPIインターフェース、MPI通信の種類、その他
 - ▶ 第3回:OpenMPの基礎、2013年4月25日
 - ▶ OpenMPの基礎、利用方法、その他
 - ▶ 第4回:Hybrid並列化技法(MPIとOpenMPの応用)、2013年5月9日
 - ▶ 背景、Hybrid並列化の適用事例、利用上の注意、その他
 - ▶ 第5回:プログラム高速化の応用、2013年5月16日
 - ▶ プログラムの性能ボトルネックに関する考えかた(I/O、単体性能(演算機ネック、メモリネック)、並列性能(バランス))、性能プロファイル、その他

教科書（演習書）

- ▶ 「スパコンプログラミング入門
ー並列処理とMPIの学習ー」
 - ▶ 片桐 孝洋 著、
 - ▶ 東大出版会、ISBN978-4-13-062453-4、
発売日:2013年3月12日、判型:A5, 200頁
 - ▶ 【本書の特徴】
 - ▶ C言語で解説
 - ▶ C言語、Fortran90言語のサンプルプログラムが付属
 - ▶ 数値アルゴリズムは、図でわかりやすく説明
 - ▶ 本講義の内容を全てカバー
 - ▶ 内容は初級。初めて並列数値計算を学ぶ人向けの入門書



参考書

- ▶ 「並列数値処理 – 高速化と性能向上のために –」
 - ▶ 金田康正 東大教授 理博 編著、
片桐孝洋 東大特任准教授 博士(理学) 著、黒田久泰 愛媛大准教授
博士(理学) 著、山本有作 神戸大教授 博士(工学) 著、五百木伸洋
(株)日立製作所 著、
 - ▶ コロナ社、発行年月日:2010/04/30, 判 型: A5, ページ数:272頁、
ISBN:978-4-339-02589-7, 定価:3,990円(本体3,800円+税5%)
 - ▶ 【本書の特徴】
 - ▶ Fortran言語で解説
 - ▶ 数値アルゴリズムは、数式などで厳密に説明
 - ▶ 本講義の内容に加えて、固有値問題の解法、疎行列反復解法、
FFT、ソート、など、主要な数値計算アルゴリズムをカバー
 - ▶ 内容は中級～上級。専門として並列数値計算を学びたい人向き

教科書の利用方法

- ▶ 本講義の全内容、演習内容をカバーした資料
- ▶ 教科書というより、実機を用いた並列プログラミングの演習書として位置づけられている
 - ▶ 使える並列計算機があることが前提
- ▶ 付属の演習プログラムの利用について
 1. 東京大学情報基盤センターのFX10スーパーコンピュータシステムでそのまま利用する
 2. 研究室のPCクラスタ(MPIが利用できるもの)で利用する
 3. 東大以外の大学等のスーパーコンピュータで利用する
- ▶ 各自のPCを用いて、(MPIではない)逐次プログラムで演習する(主に逐次プログラムの高速化の話題)

はじめに

スパコンとは何か？

スーパーコンピュータとは

- ▶ 人工知能搭載のコンピュータではない
- ▶ 明確な定義はない
 - ▶ 現在の最高レベルの演算性能をもつ計算機のこと
 - ▶ 経験的には、PCの1000倍高速で、1000倍大容量なメモリをもつ計算機
 - ▶ 外為法安全保障貿易管理の外国為替及び外国貿易法の法令（平成24年7月13日公表）の規制対象デジタル電子計算機
 - ▶ 第7条三項ハ：デジタル電子計算機であって、加重最高性能が三・〇実効テラ演算を超えるもの
- ▶ 現在、ほとんどすべてのスーパーコンピュータは並列計算機
- ▶ 東京大学情報基盤センターが所有するFX10スーパーコンピュータシステムも、並列計算機

スーパーコンピュータで用いる単位

- ▶ **TFLOPS (テラ・フロップス、Tera Floating Point Operations Per Second)**
 - ▶ 1秒間に1回の演算能力(浮動小数点)が1FLOPS。
 - ▶ K(キロ)は1,000(千)、M(メガ)は1,000,000(百万)、G(ギガ)は1,000,000,000(十億)、T(テラ)は1,000,000,000,000(一兆)
 - ▶ だから、**一秒間に一兆回の浮動小数点演算の能力がある** こと。
- ▶ **PFLOPS (ペタ・フロップス)**
 - ▶ 1秒間に1京(けい)回の浮動小数点演算の能力がある。
 - ▶ 「京コンピュータ」(2012年9月共用開始、11.2PFLOPS、**現在TOP500で3位**)

- PCの演算能力は？
 - 3.3GHz(1秒間に3.3G回のクロック周波数)として、もし1クロックあたり1回の浮動小数点演算ができれば3.3GFLOPS。
 - Intel Core i7 (Sandy Bridge)では、6コア、1クロックで8回の浮動小数計算ができるので、 $3.3 \text{ GHz} \times 8 \text{ 回浮動小数点演算/Hz} \times 6 \text{ コア} = 158.4 \text{ GFLOPS}$
 - Cray-1は160MFLOPS。1970年代のスパコンより、PCの方が990倍以上高速！

スーパーコンピュータ用語

▶ 理論性能(Theoretical Performance)

- ▶ ハードウェア性能からはじき出した性能。
- ▶ 1クロックに実行できる浮動小数点回数から算出したFLOPS値を使うことが多い。

▶ 実効性能(Effective Performance)

- ▶ 何らかのベンチマークソフトウェアを実行して実行時間を計測。
- ▶ そのベンチマークプログラムに使われている浮動小数点演算を算出。
- ▶ 以上の値を基に算出したFLOPS値のこと。
- ▶ 連立一次方程式の求解ベンチマークであるLINPACKを用いることが多い。

ムーアの法則

- ▶ 米Intel社の設立者ゴードン・ムーアが提唱した、半導体技術の進歩に関する経験則。

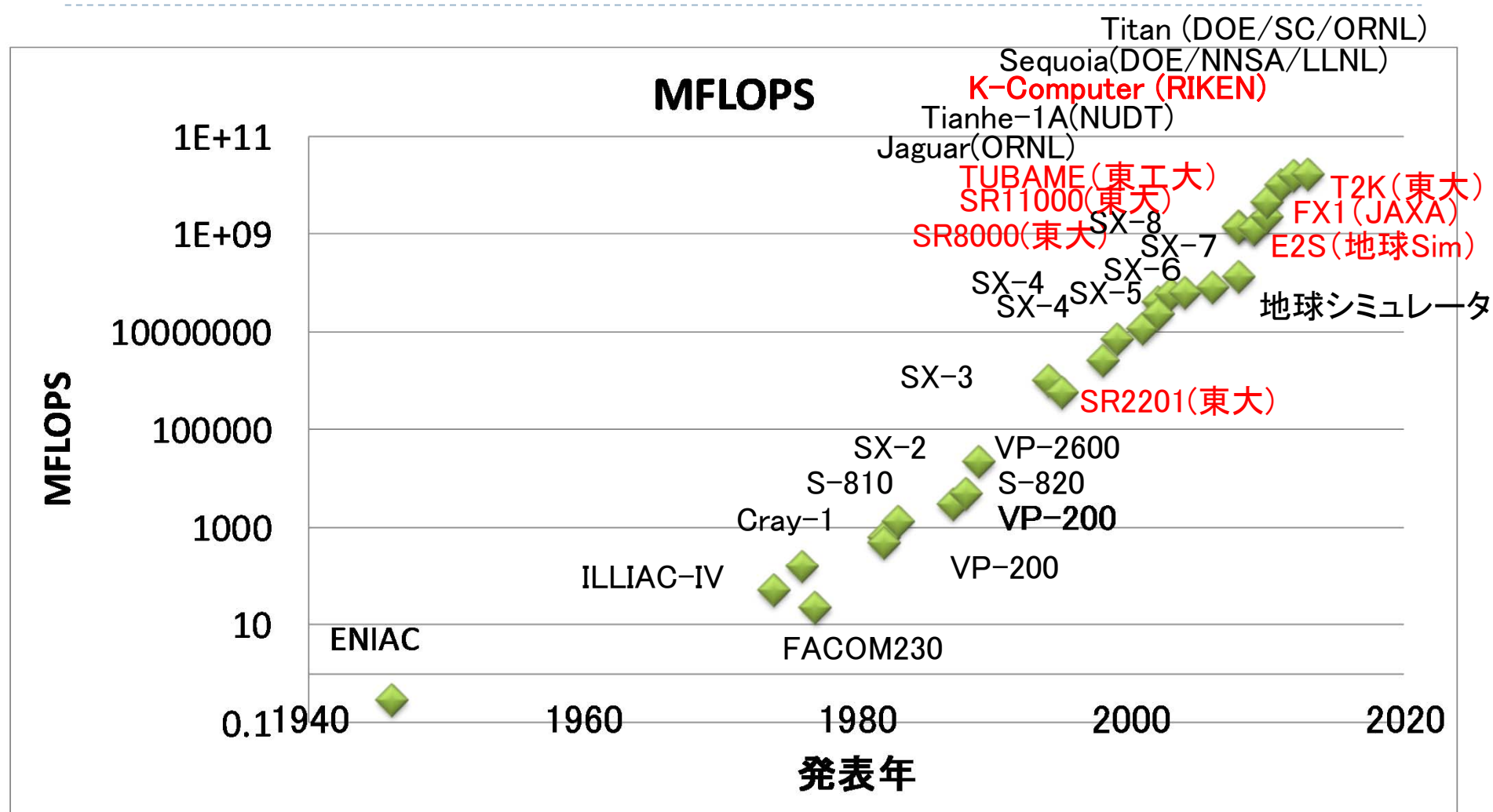
「半導体チップの集積度は、およそ18ヵ月で2倍になる」

- ▶ これから転じて、

「マイクロプロセッサの性能は、およそ18ヵ月で2倍になる」

- ▶ 上記によると、約5年で10倍となる。

スーパーコンピュータ性能推移 (主に日本製、理論性能)



スーパーコンピュータのランキング

▶ TOP500 Supercomputer Sites

(<http://www.top500.org/>)

- ▶ LINPACKの値から実効性能を算出した値の500位までのランキング
- ▶ 米国オークリッジ国立研究所／テネシー大学ノックスビル校の Jack Dongarra 教授が発案
- ▶ 毎年、6月、11月（米国の国際会議SC | xy）に発表

現在のランキング

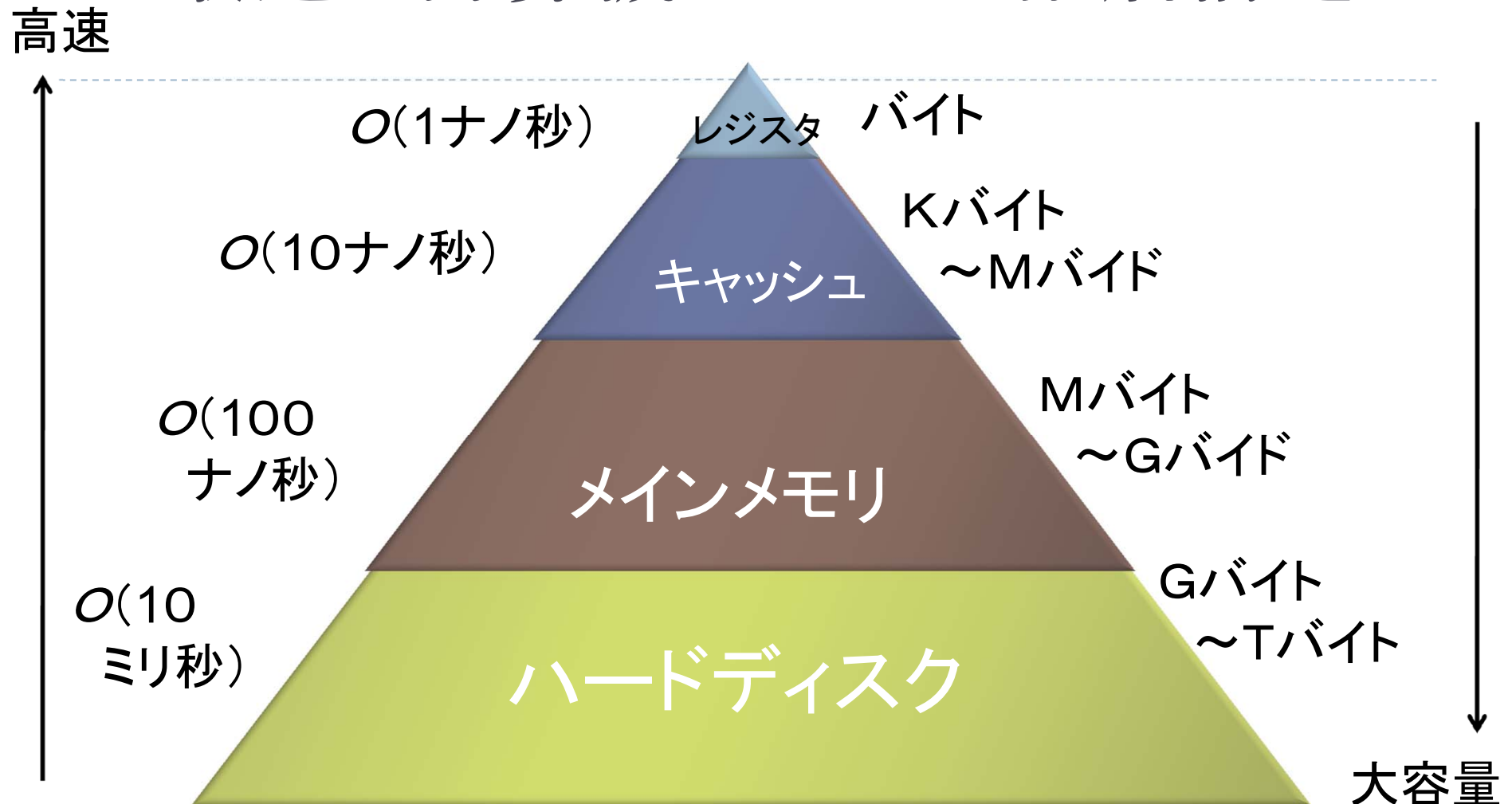
出典: <http://www.top500.org/list/2012/11/>

2012年11月現在

- 1位: 米国 DOE/SC/ORNLのTitan
17.590 PFLOPS
- 2位: 米国 DOE/NNSA/LLNLのSequoia (BlueGene/Q)
16.324 PFLOPS
- 3位: 日本 **K-Computer (Sparc64 XIIIfx)**
10.510 PFLOPS
- 4位: 米国 DOE/SC/ANLのMira (BlueGene/Q)
8.162 PFLOPS
- 5位: 独国 Juelich ForechungszentrumのJUQUEEN (BlueGene/Q)
4.141 PFLOPS
- その他の日本のマシン
 - 15位の国際核融合研のHelios (Xeon E5-2680)
1.237 PFLOPS
 - 17位の東工大のTUBAME2.0
1.192 PFLOPS
 - 21位の東京大学情報基盤センターのOakleaf-fx (Sparc64 IVfx)
1.042 PFLOPS

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...

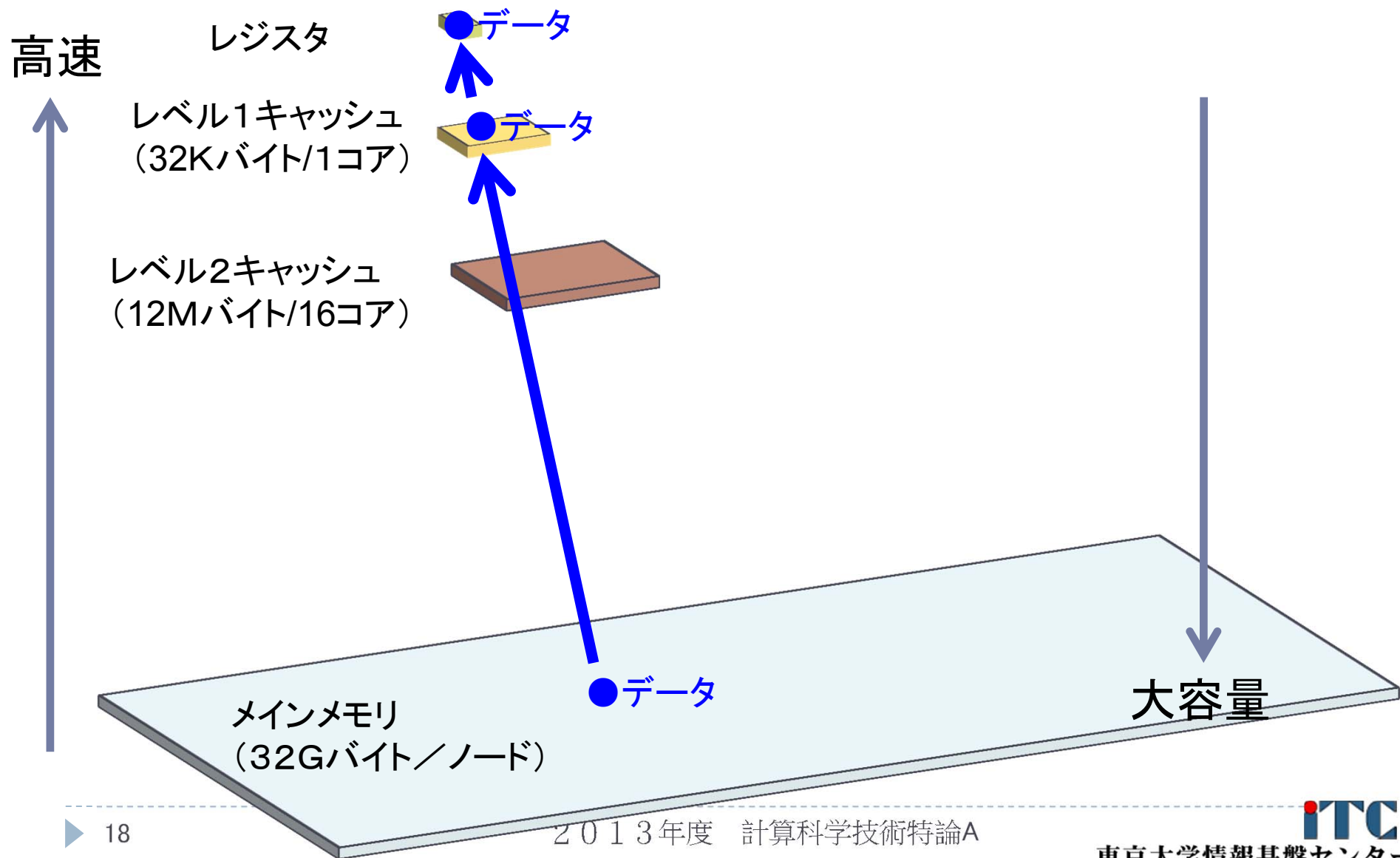
レジスタ

キャッシュ

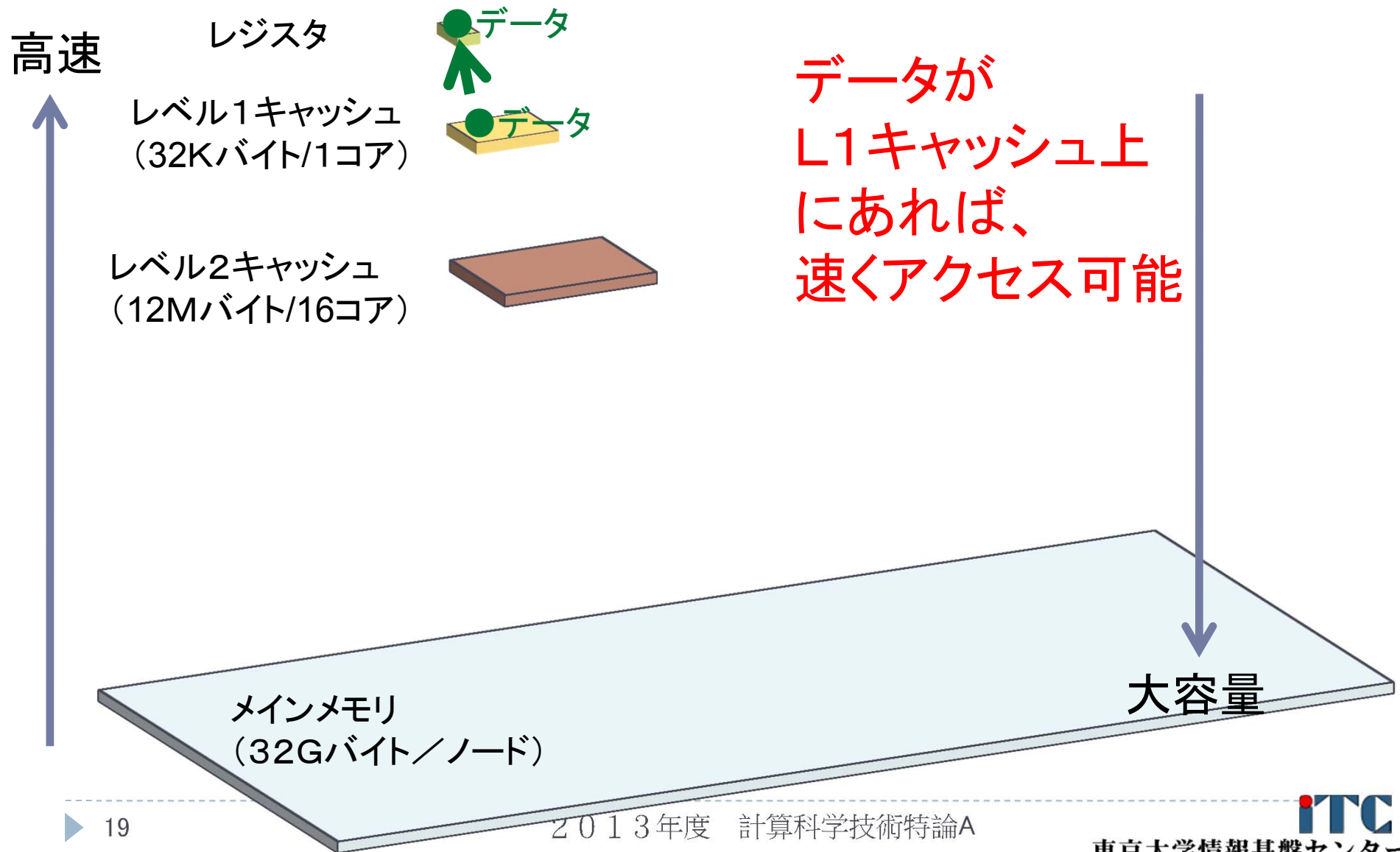
メインメモリ

- 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

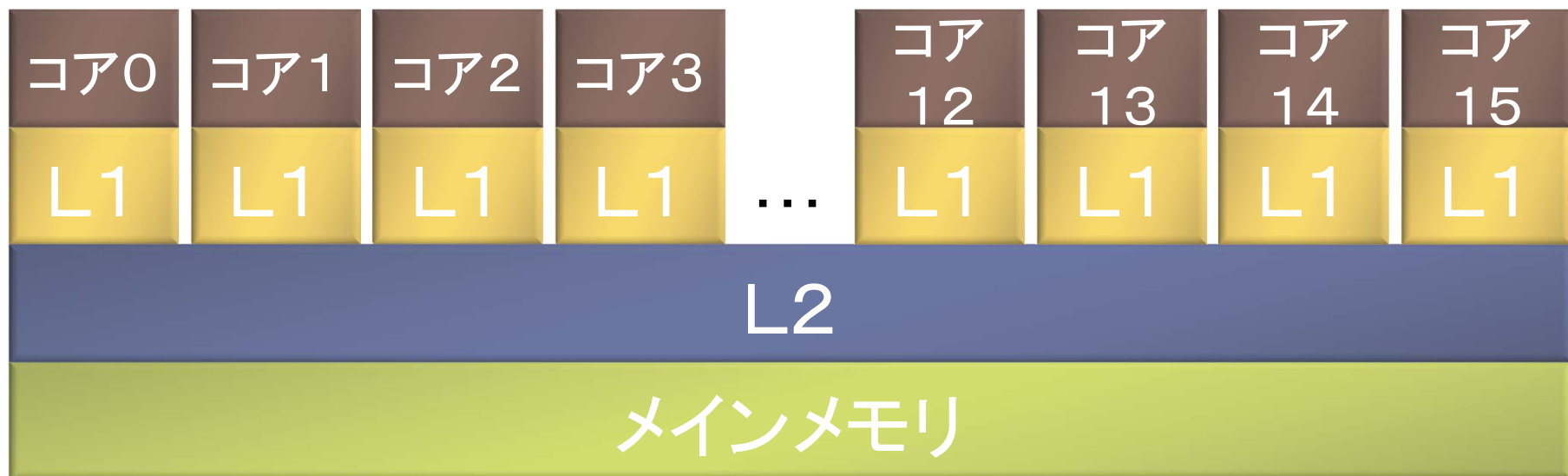
東京大学FX10のメモリ構成例



東京大学FX10のメモリ構成例

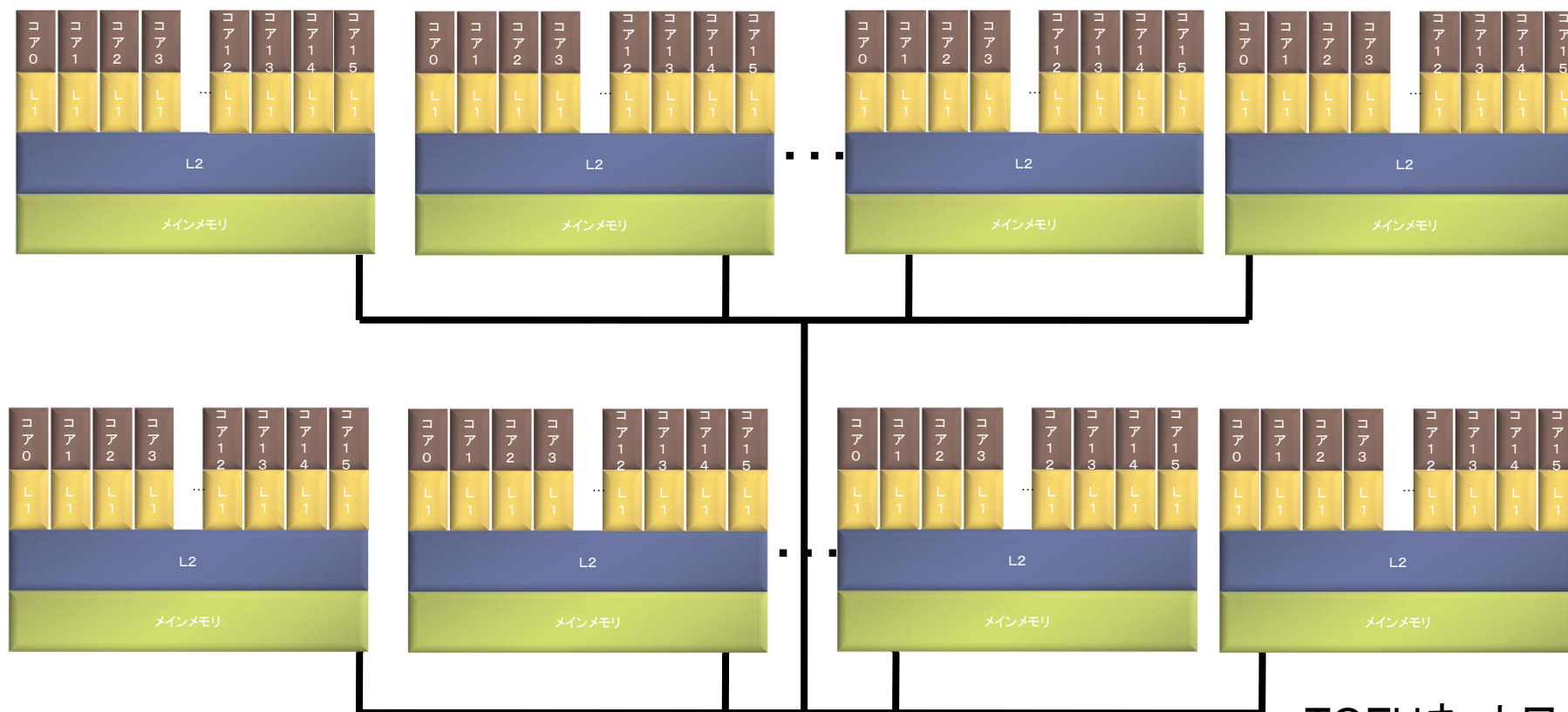


東京大学 F X 1 0 のノードのメモリ構成例



※階層メモリ構成となっている

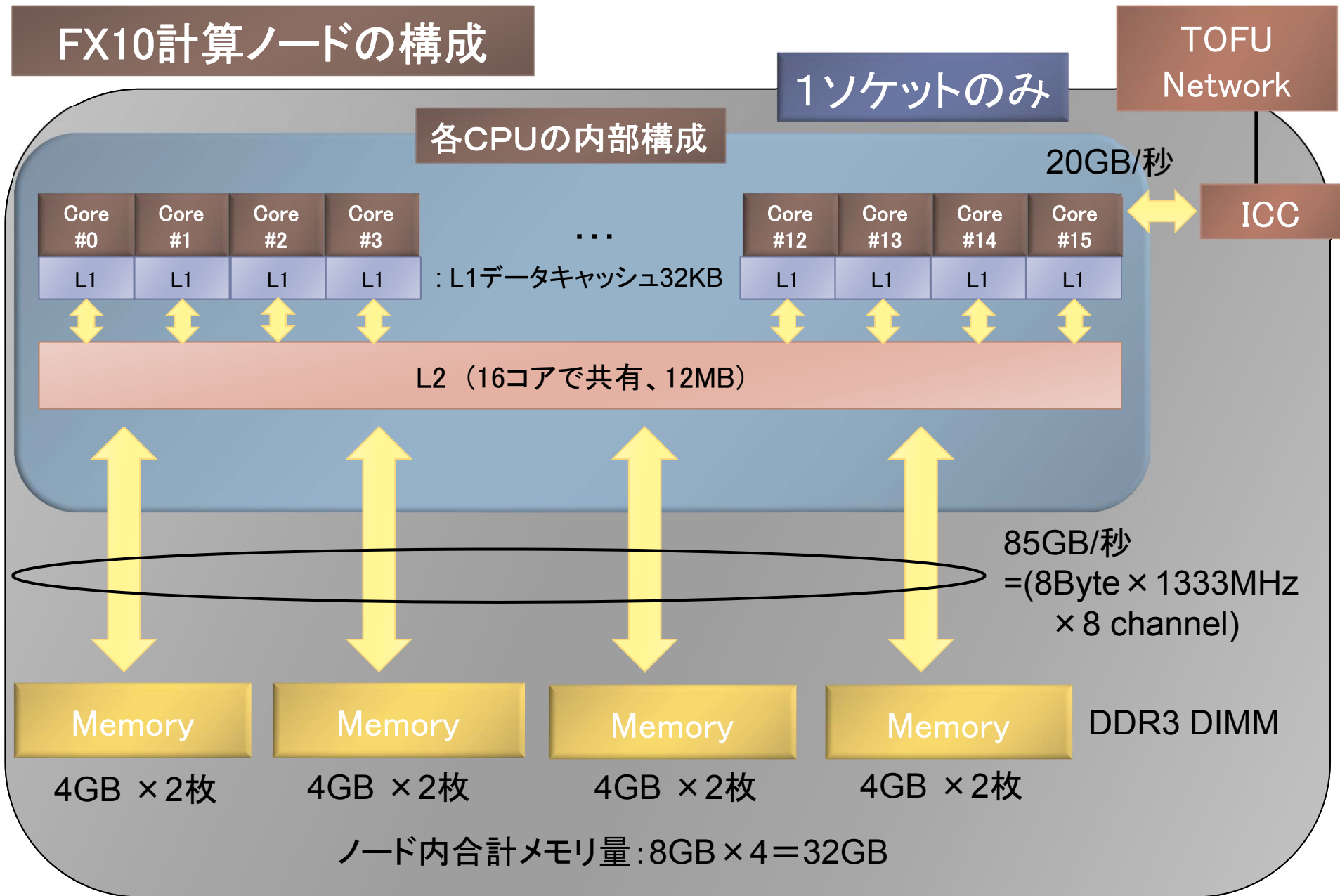
東京大学FX10全体メモリ構成



メモリ階層が階層

TOFUネットワーク
(5Gバイト／秒
× 双方向)

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)

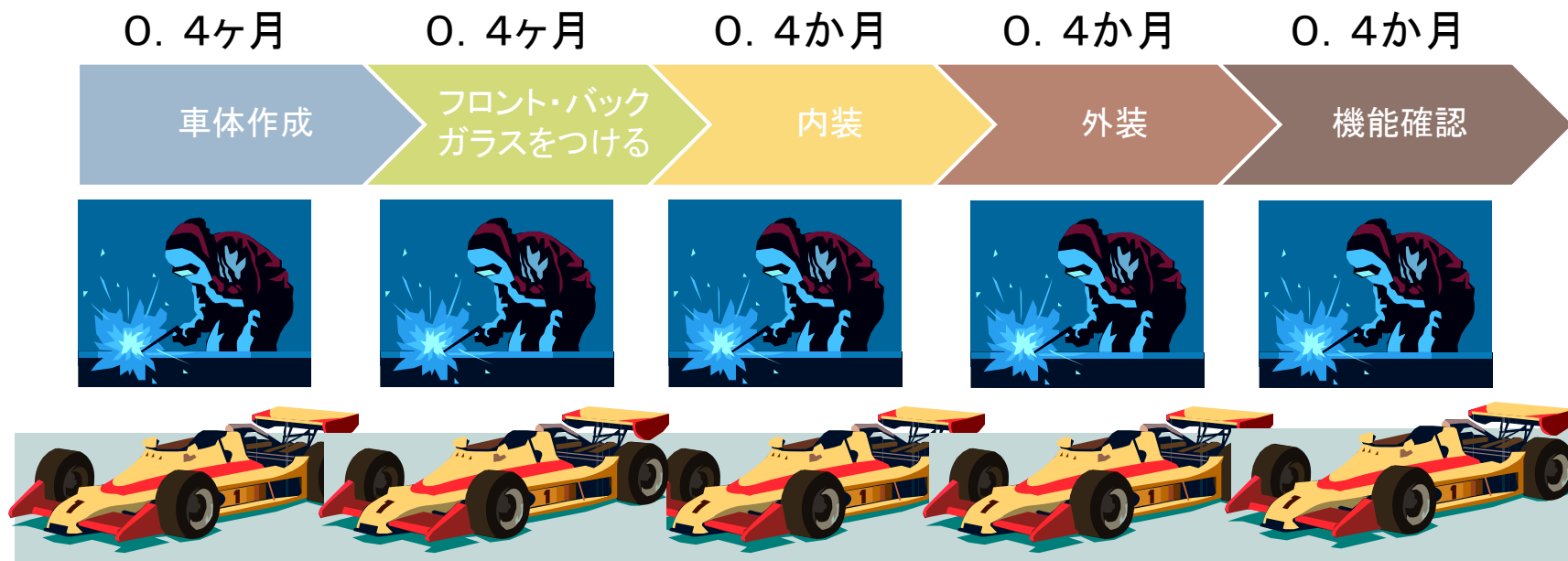
1台目
2台目
3台目



時間 →

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



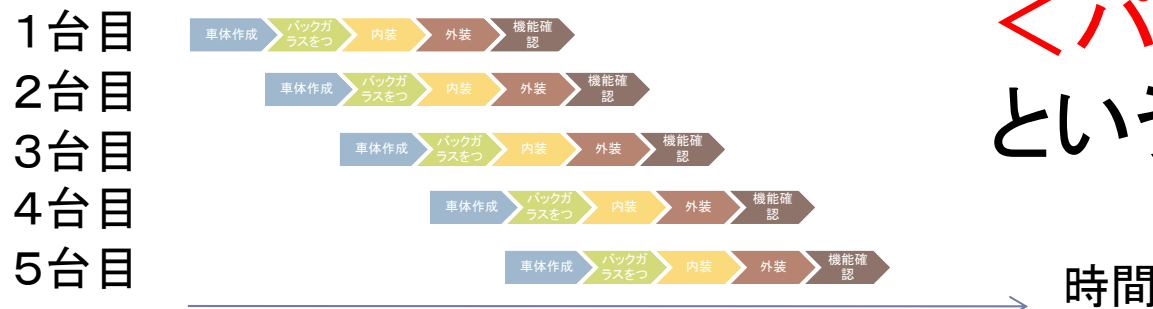
流れ作業

▶ 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. ハードウェア・パイプライニング

- ▶ 計算機ハードウェアで行う
- ▶ 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. ソフトウェア・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・ポストアストア)
 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];
 }
}

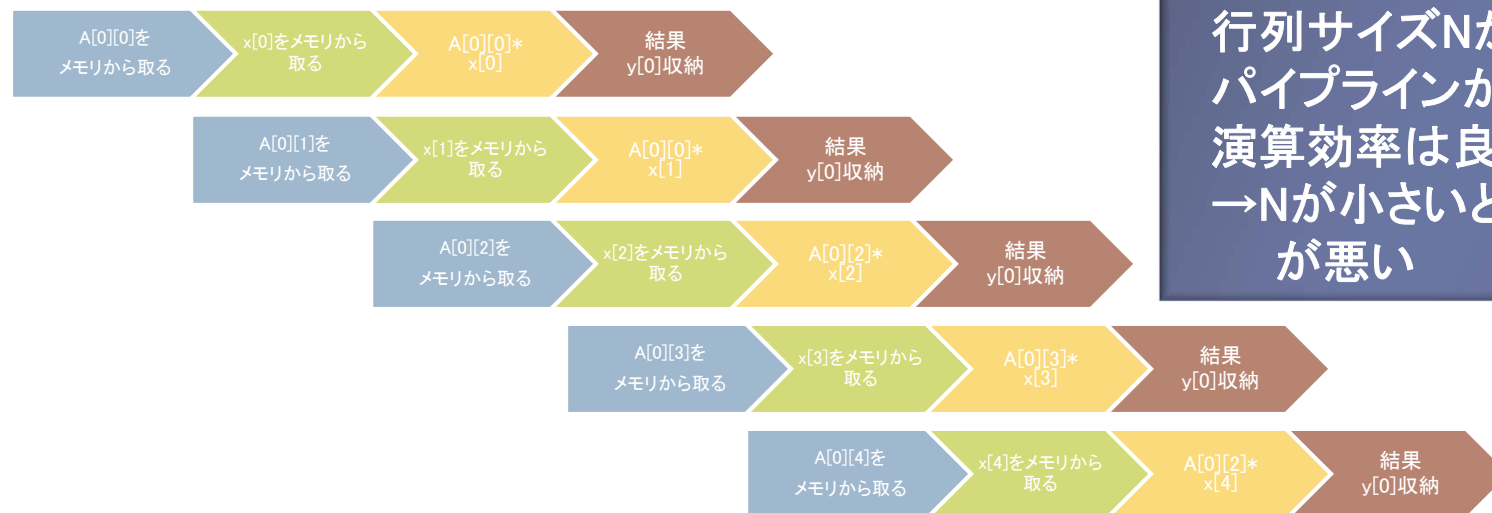
演算器が稼働する工程

- ▶ 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

→ 時間

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のハードウェア情報

- ▶ 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、
 $1.848 \text{ GHz} \times 8 \text{ 回} = \mathbf{14.784 \text{ GFLOPS / コア}}$
 - ▶ 1ノード16コアでは、
 $14.784 \times 16 \text{ コア} = \mathbf{236.5 \text{ GFLOPS / ノード}}$
- ▶ 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];  
    }  
}
```



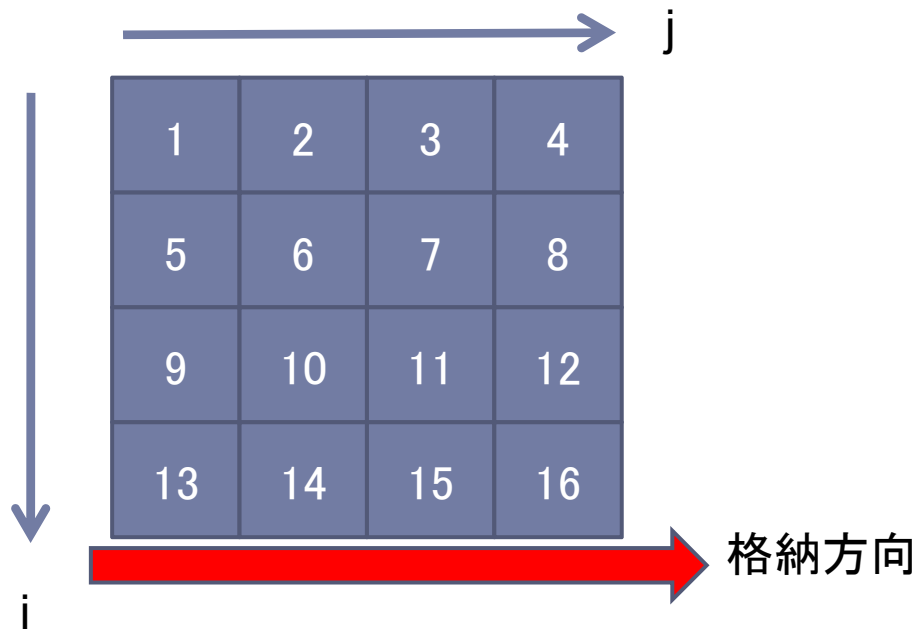
OK

```
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

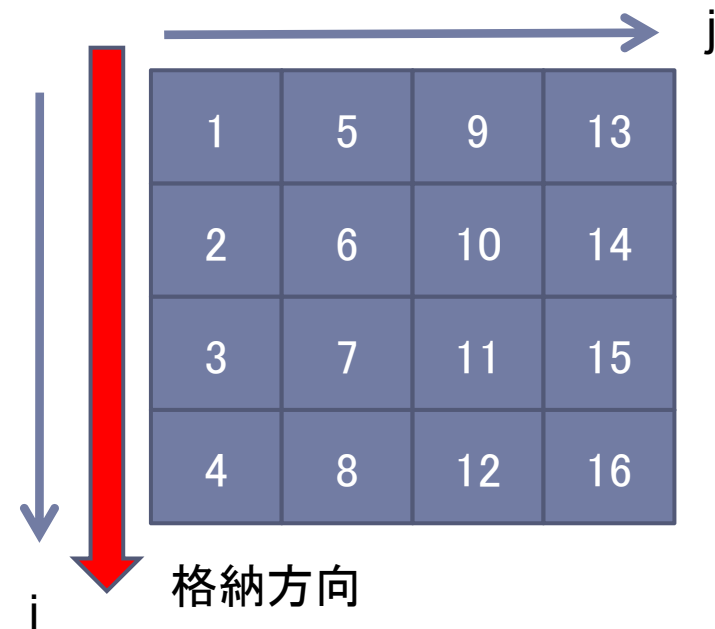
▶ C言語の場合

$A[i][j]$



▶ Fortran言語の場合

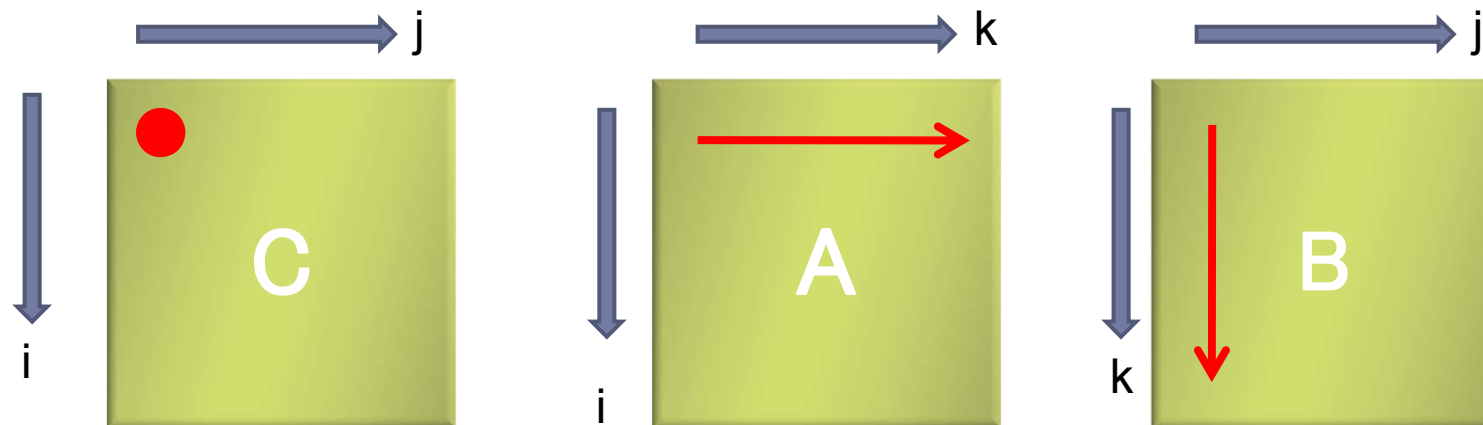
$A(i, j)$



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
$$c_{ij} = \sum_{k=1}^n a_{ik} b_{kj} \quad (i, j = 1, 2, \dots, n)$$

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ーンが
＜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ーンが
＜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

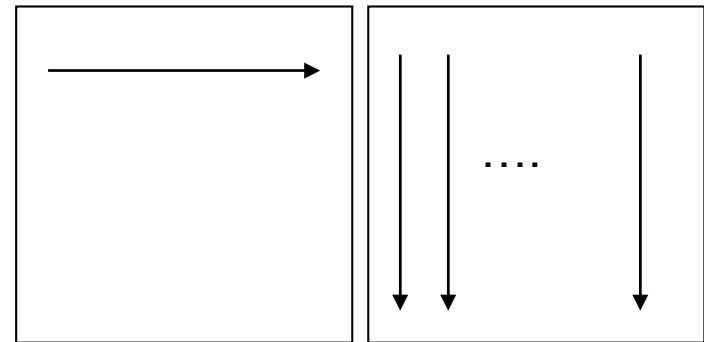
▶ 内積形式 (inner-product form)

▶ 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>。

A B



●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

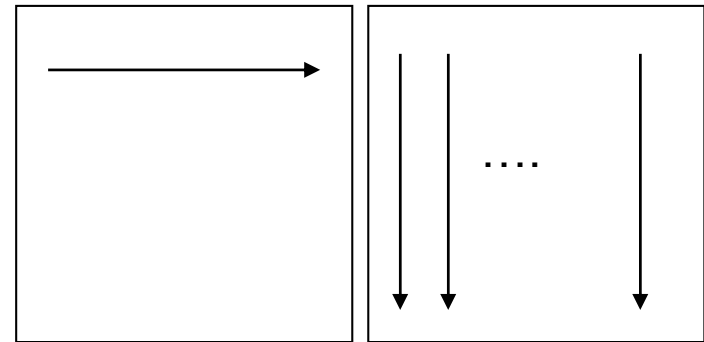
▶ 内積形式 (inner-product form)

▶ 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>。

A B



●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)

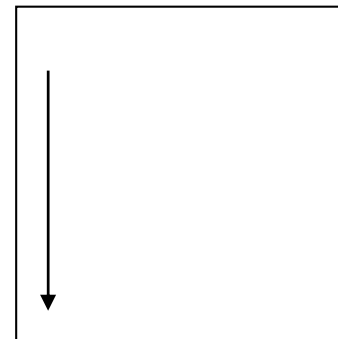
行列の積

▶ 外積形式 (outer-product form)

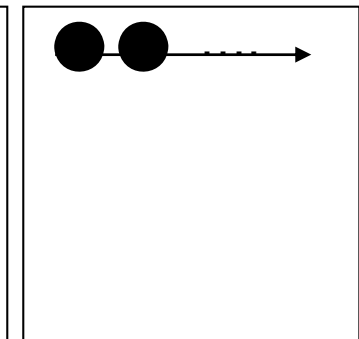
▶ kij, kjiループによる実現 (C言語)

```
▶ 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;  
        }  
    }  
}
```

A



B



●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)

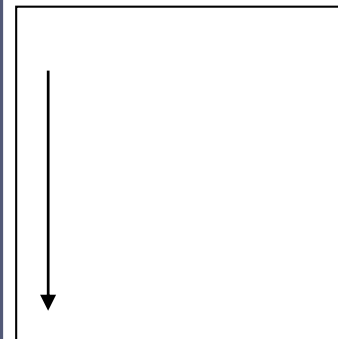
行列の積

▶ 外積形式 (outer-product form)

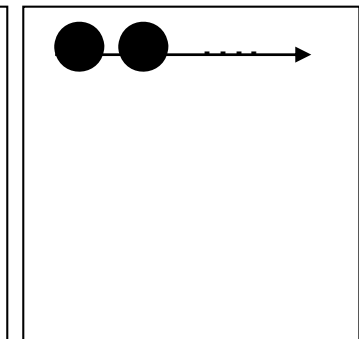
▶ kij, kjiループによる実現 (Fortran言語)

```
▶ 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
```

A



B



●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)

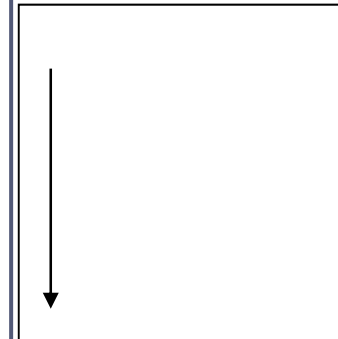
行列の積

▶ 中間積形式 (middle-product form)

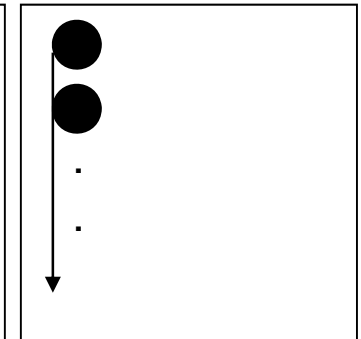
▶ ikj, jkiループによる実現(C言語)

```
▶ 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;  
        }  
    }  
}
```

A



B



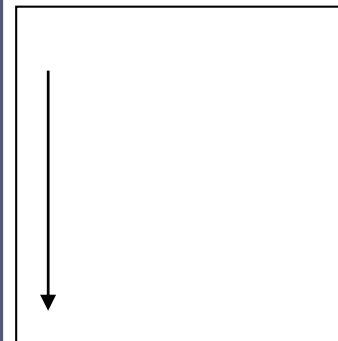
●jkiループでは
全て列方向アクセス
→列方向格納言語に
最も向いている
(Fortran言語)

行列の積

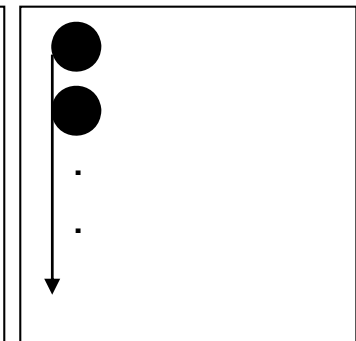
- ▶ 中間積形式 (middle-product form)
 - ▶ ikj, jkiループによる実現 (Fortran言語)

```
▶ 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
```

A



B



●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)

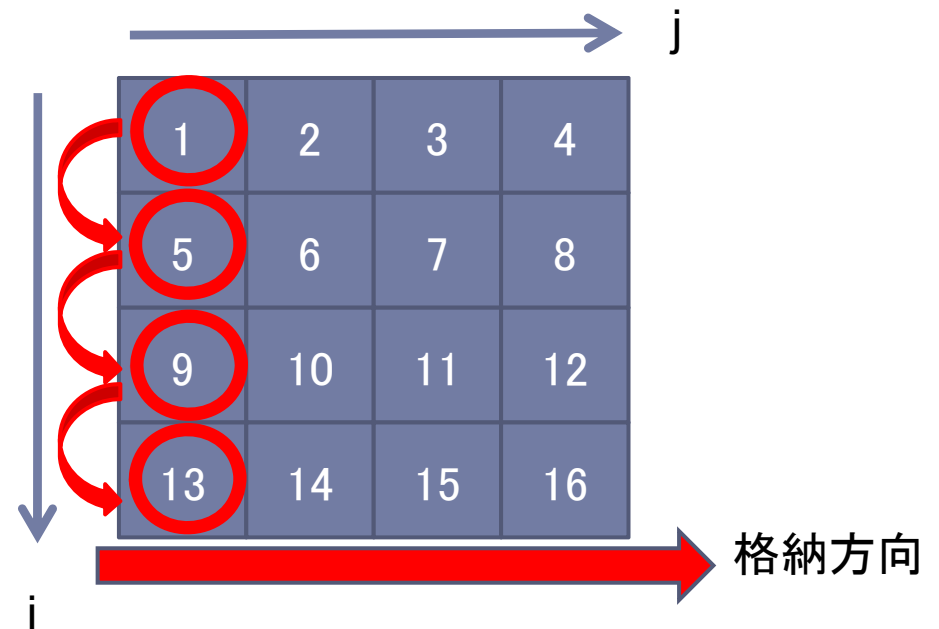


NG

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

- ▶ C言語の場合

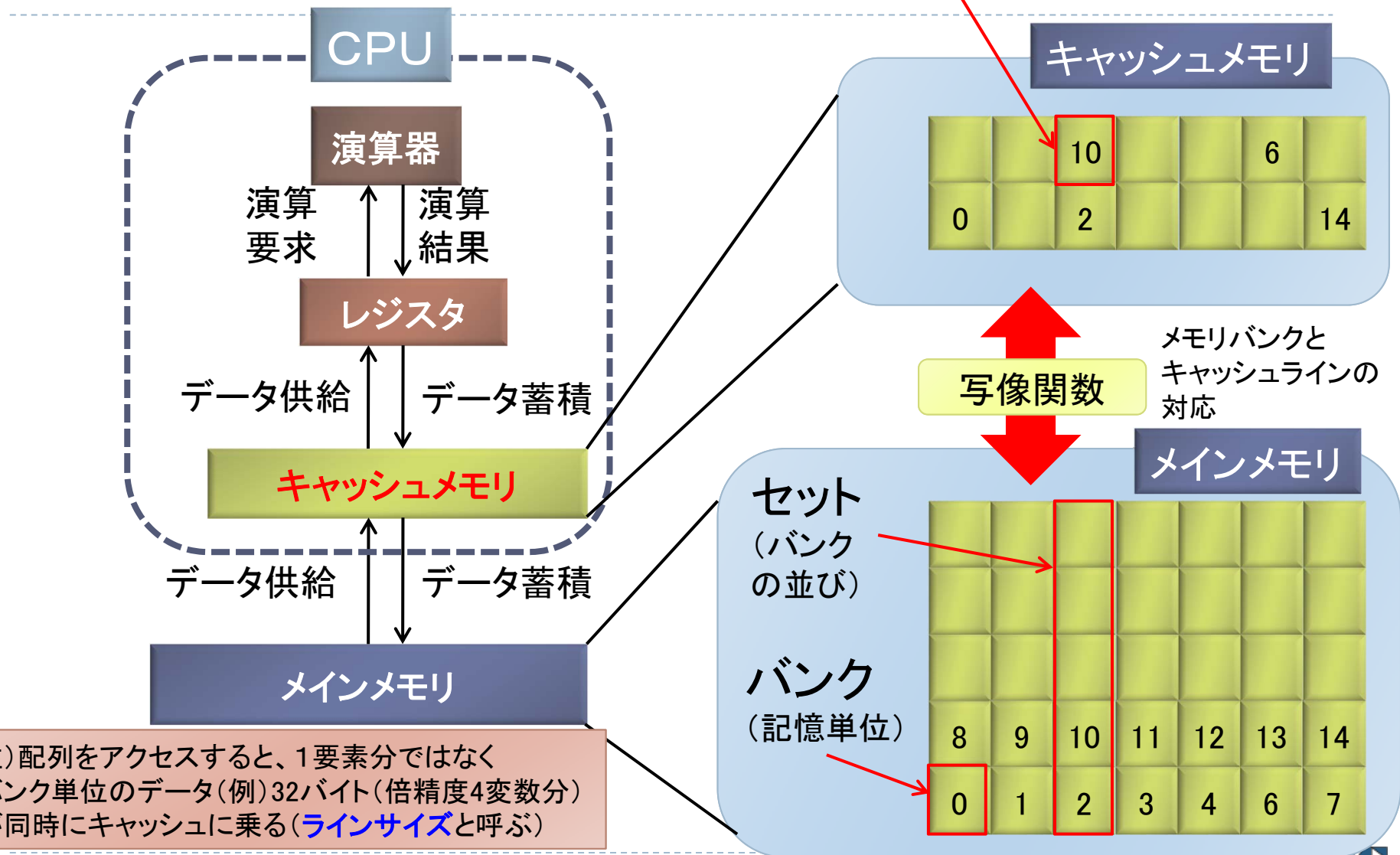
$a[i][j]$



間隔4での不連続アクセス

キャッシュメモリの構成

キャッシュライン
(キャッシュ上のバンク)



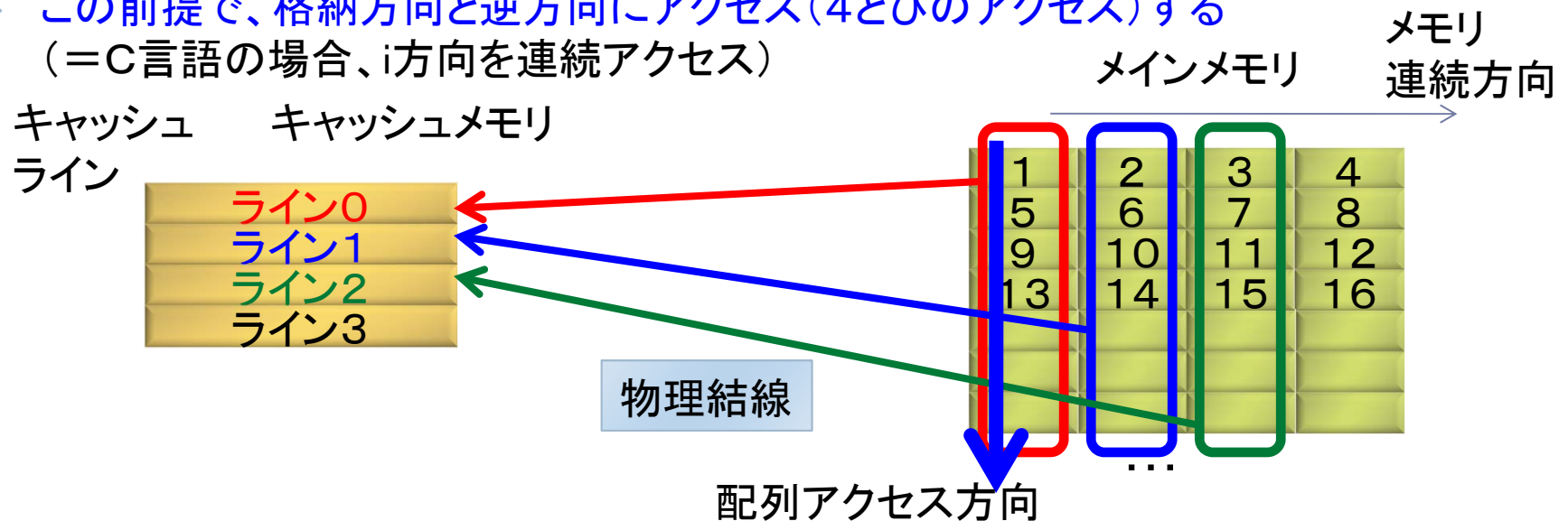
キャッシュとキャッシュライン

- ▶ 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

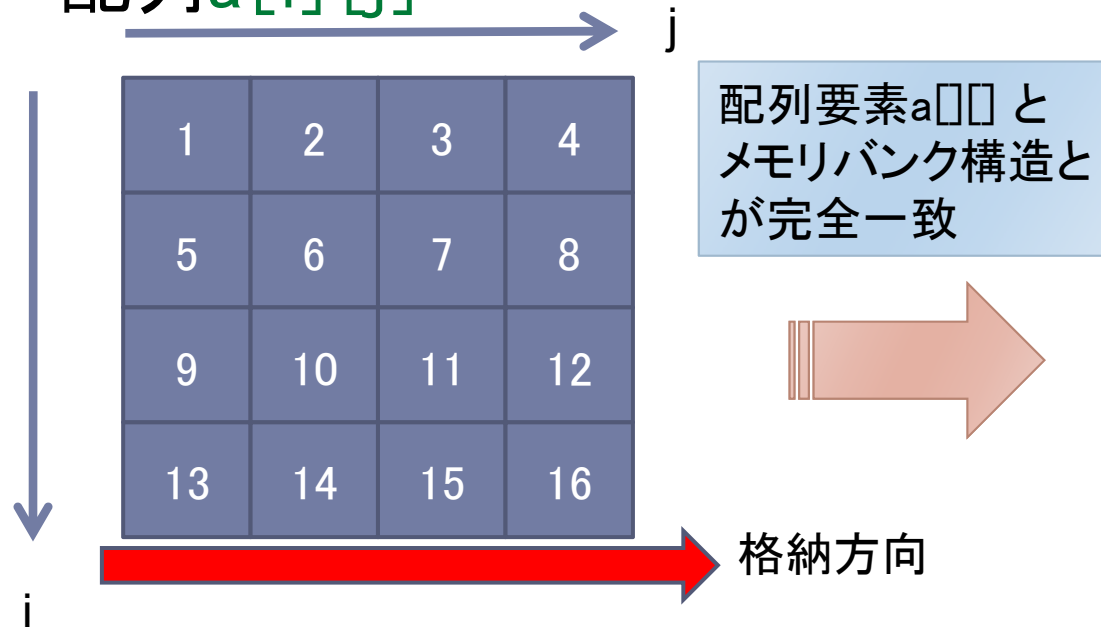
▶ この前提の、＜実際の配列構成＞と＜メモリバンク＞の関係

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

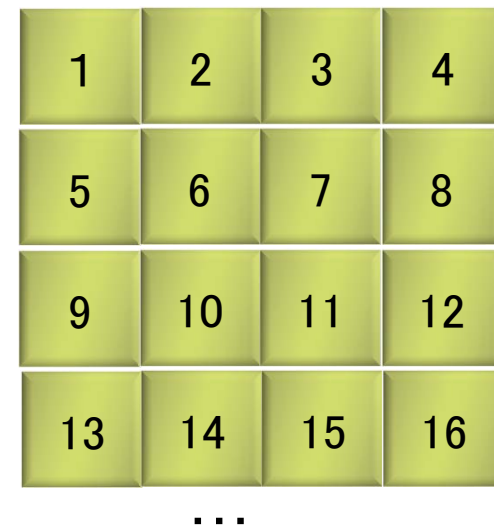
- ▶ 配列a[i][j]の物理メモリ上の配置はOSが動的に決定するので、ずれることがある
- ▶ メモリバンクの容量は、8バイトより大きい
- ▶ ダイレクト・マッピングではない

▶ C言語の場合

配列a[i][j]

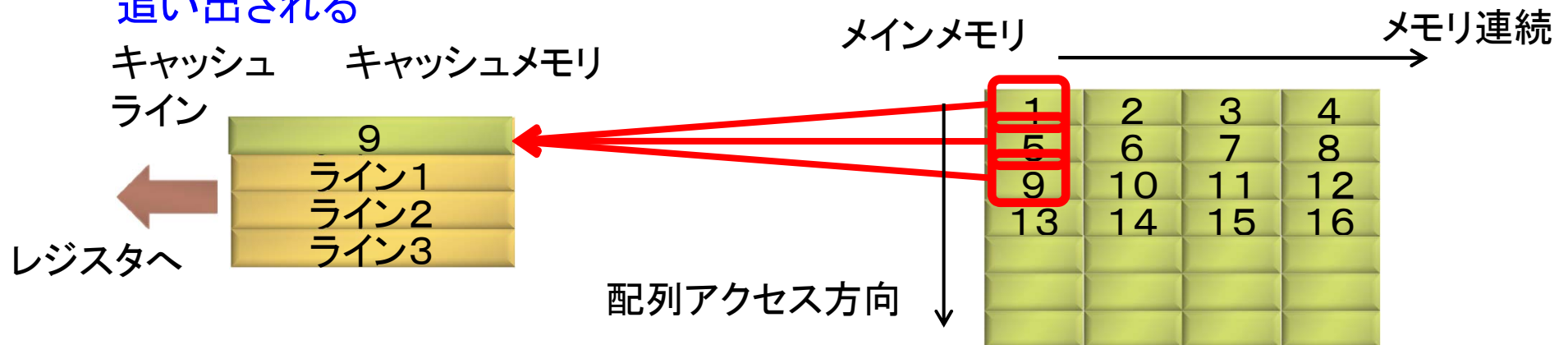


メインメモリ上の
バンク構成



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。

➡ **メモリー→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、メモリー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<インターリービング>

- 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

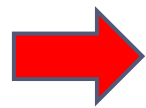
- ▶ メモリバンクのキャッシュラインへの割り付けは2冪の間隔で行っていることが多い
 - ▶ たとえば、32、64、128など
- ▶ 特定サイズの問題(たとえば1024次元)で、性能が $1/2 \sim 1/3$ 、ときには $1/10$ になる場合、キャッシュライン衝突が生じている可能性あり



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

NG

```
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. **パディング法**: 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[1025][1024];` 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。

ブロック化

小さい範囲のデータ再利用

ブロック化によるアクセス局所化

- ▶ キャッシュには**大きさ**があります。
- ▶ この大きさを超えると、たとえ連続アクセスしても、**キャッシュからデータは追い出されます**。
- ▶ データが連続してキャッシュから追い出されると、メモリから転送するのと同じとなり、高速なアクセス速度を誇るキャッシュの恩恵がなくなります。
- ▶ そこで、高速化のためには、以下が必要です
 1. **キャッシュサイズ限界までデータを詰め込む**
 2. **詰め込んだキャッシュ上のデータを、何度もアクセスして再利用する**

ブロック化によるキャッシュミスヒット削減例

- ▶ 行列一行列積
- ▶ 行列サイズ: 8×8
 - ▶ `double A[8][8];`
- ▶ キャッシュラインは4つ
- ▶ 1つのキャッシュラインに4つの行列要素が載る
 - ▶ キャッシュライン: $4 \times 8 \text{バイト(double)} = 32 \text{バイト}$
- ▶ 配列の連続アクセスは行方向(C言語)
- ▶ キャッシュの追い出しアルゴリズム:
Least Recently Used (LRU)

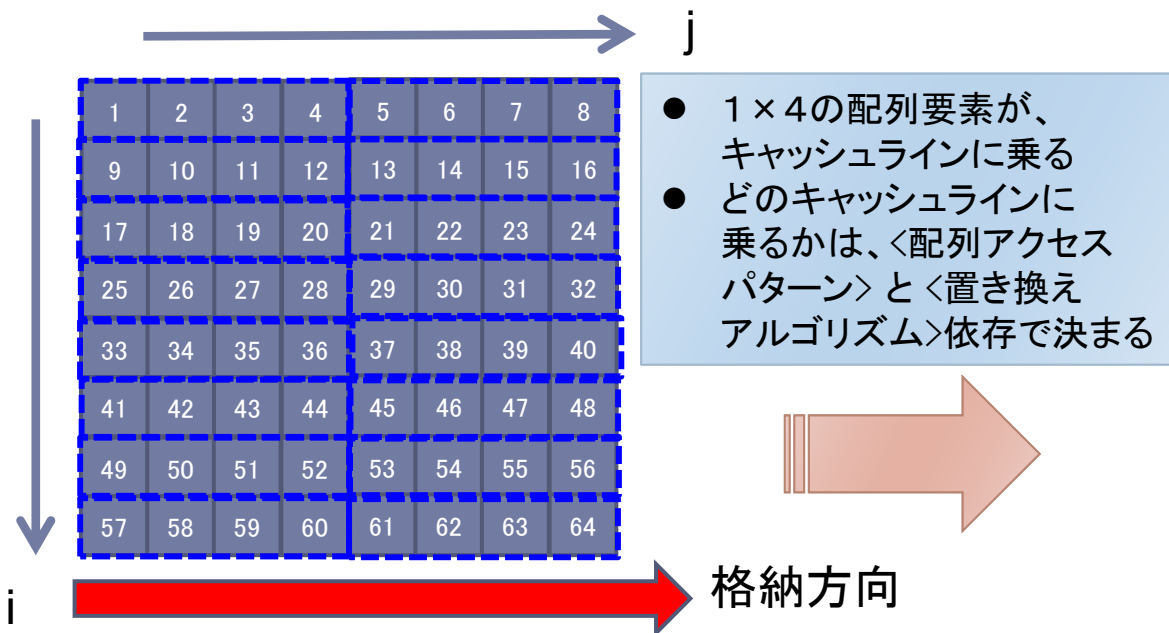
配列とキャッシュライン構成の関係

この前提の、＜配列構成＞と＜キャッシュライン＞の関係

ここでは、キャッシュライン衝突は考えません

C言語の場合

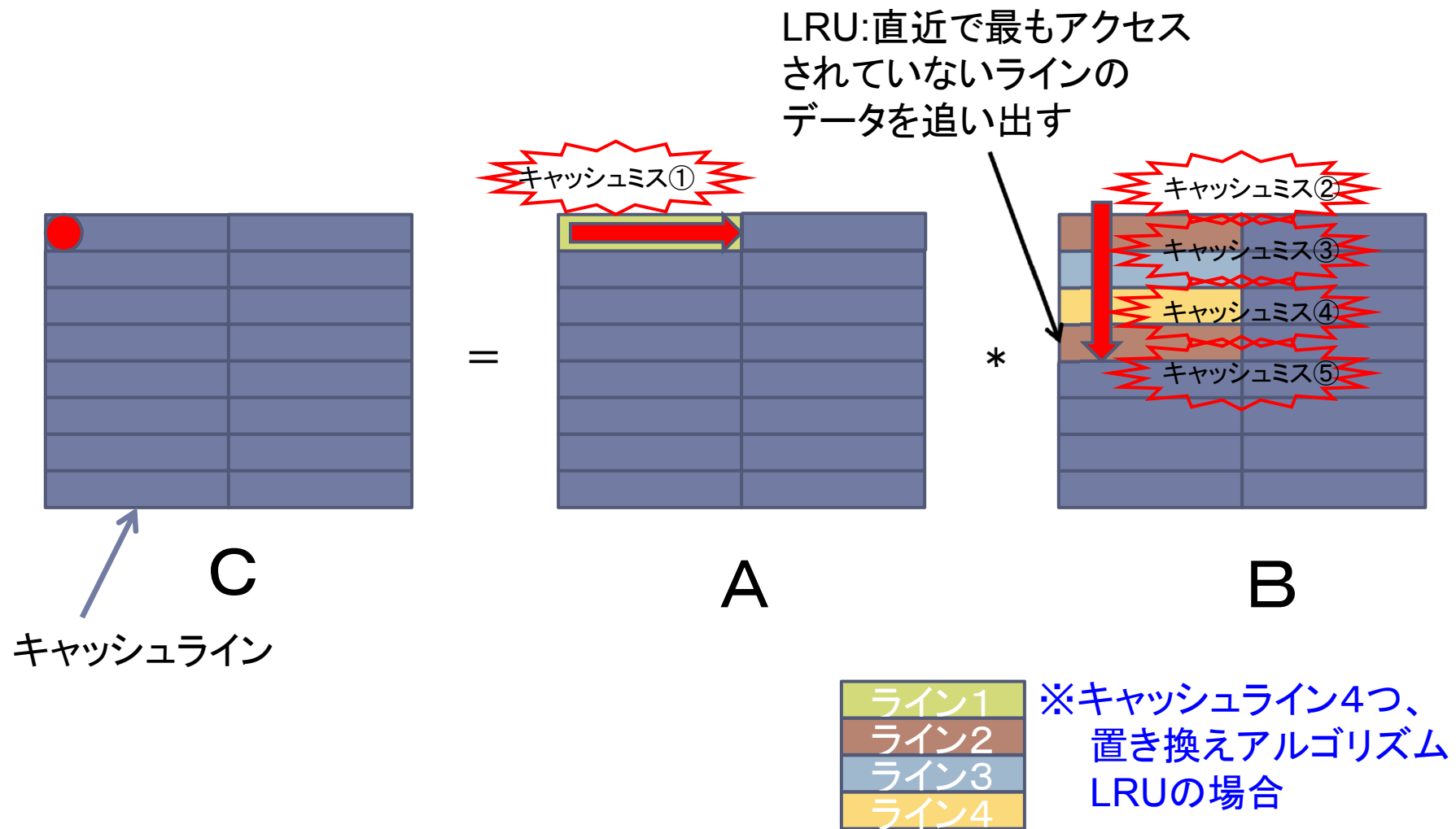
配列 $A[i][j]$ 、 $B[i][j]$ 、 $C[i][j]$



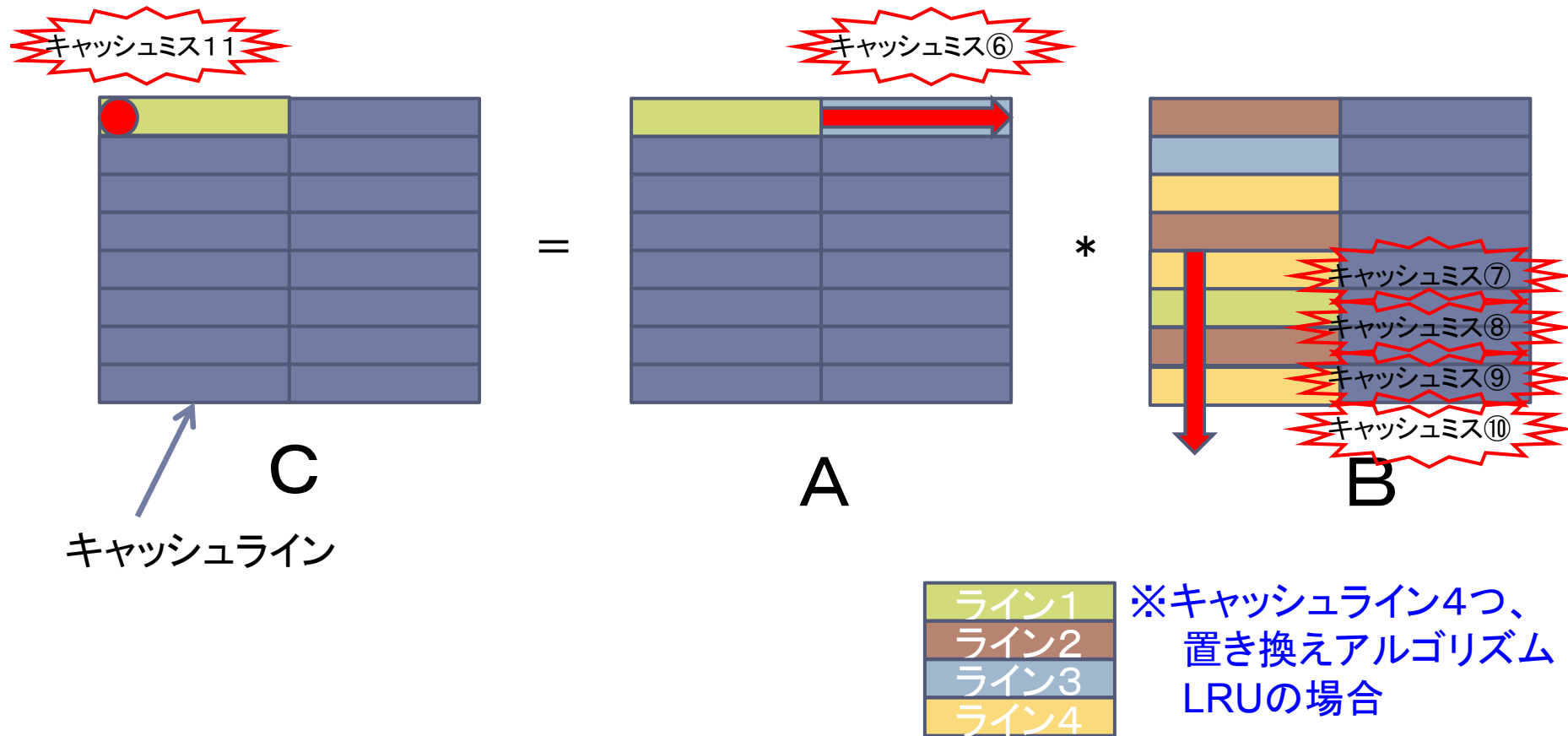
キャッシュラインの構成

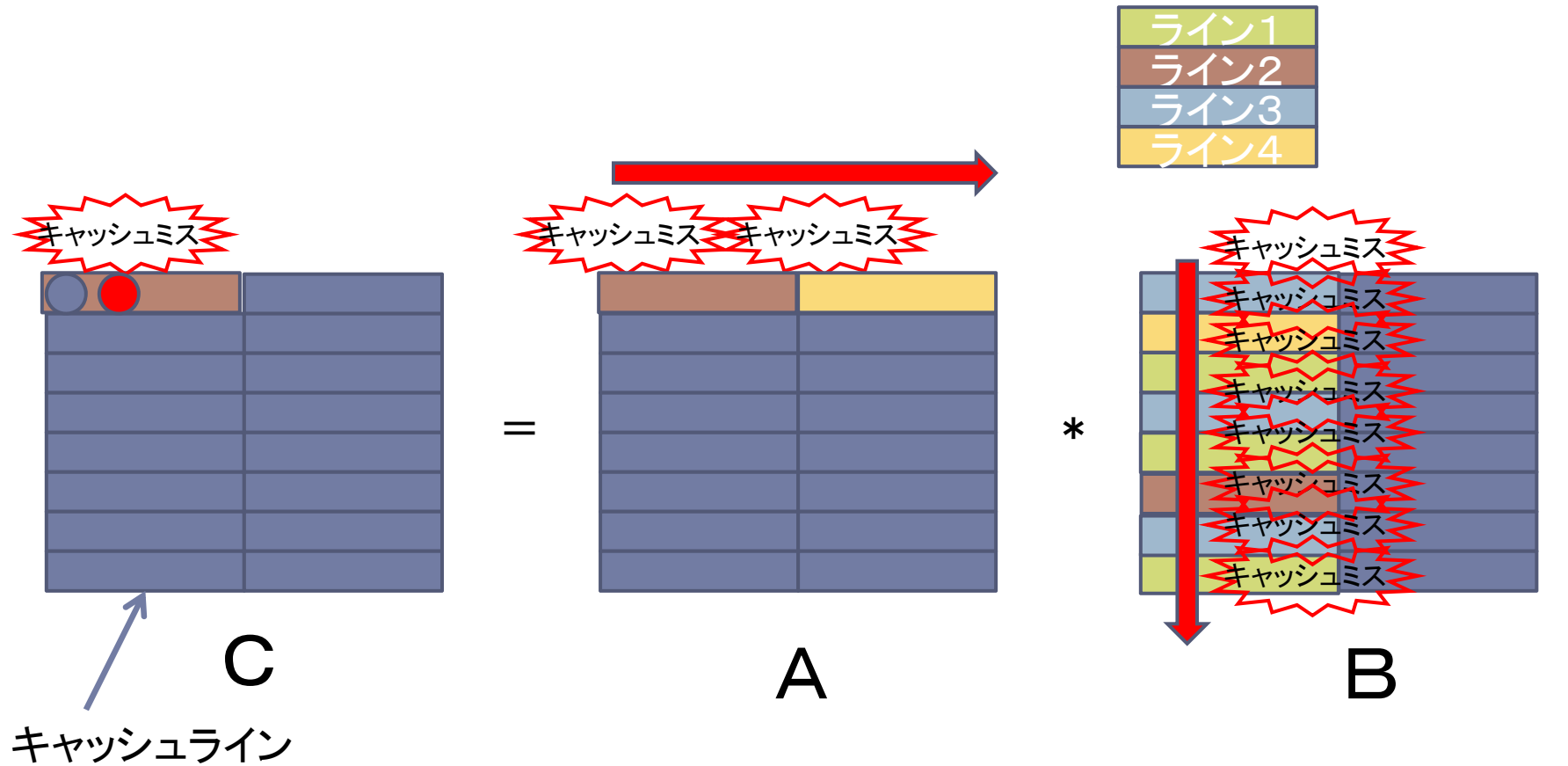
1	2
3	4

行列-行列積の場合（ブロック化しない）



行列-行列積の場合（ブロック化しない）

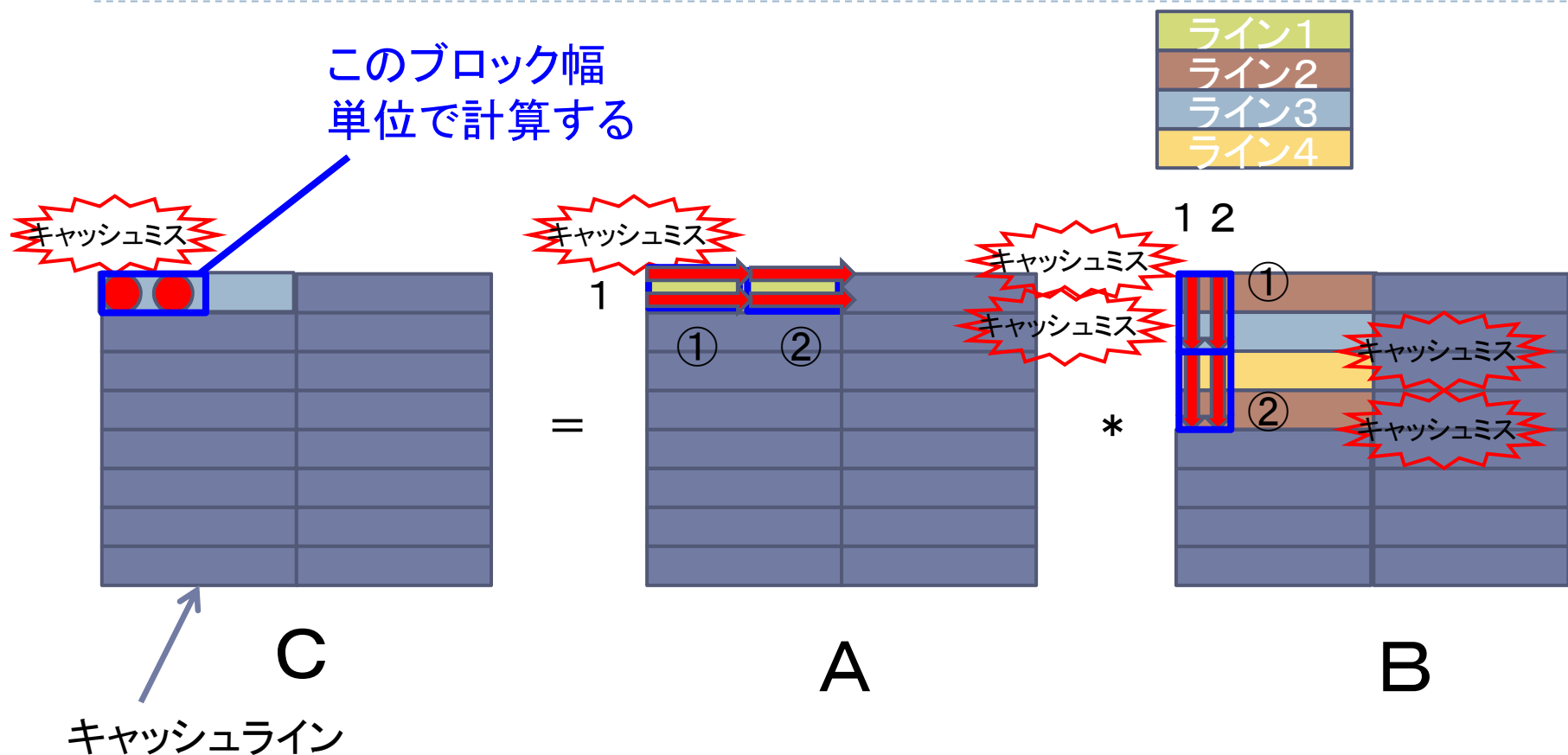




※2要素計算するのに、
キャッシュミスヒット22回

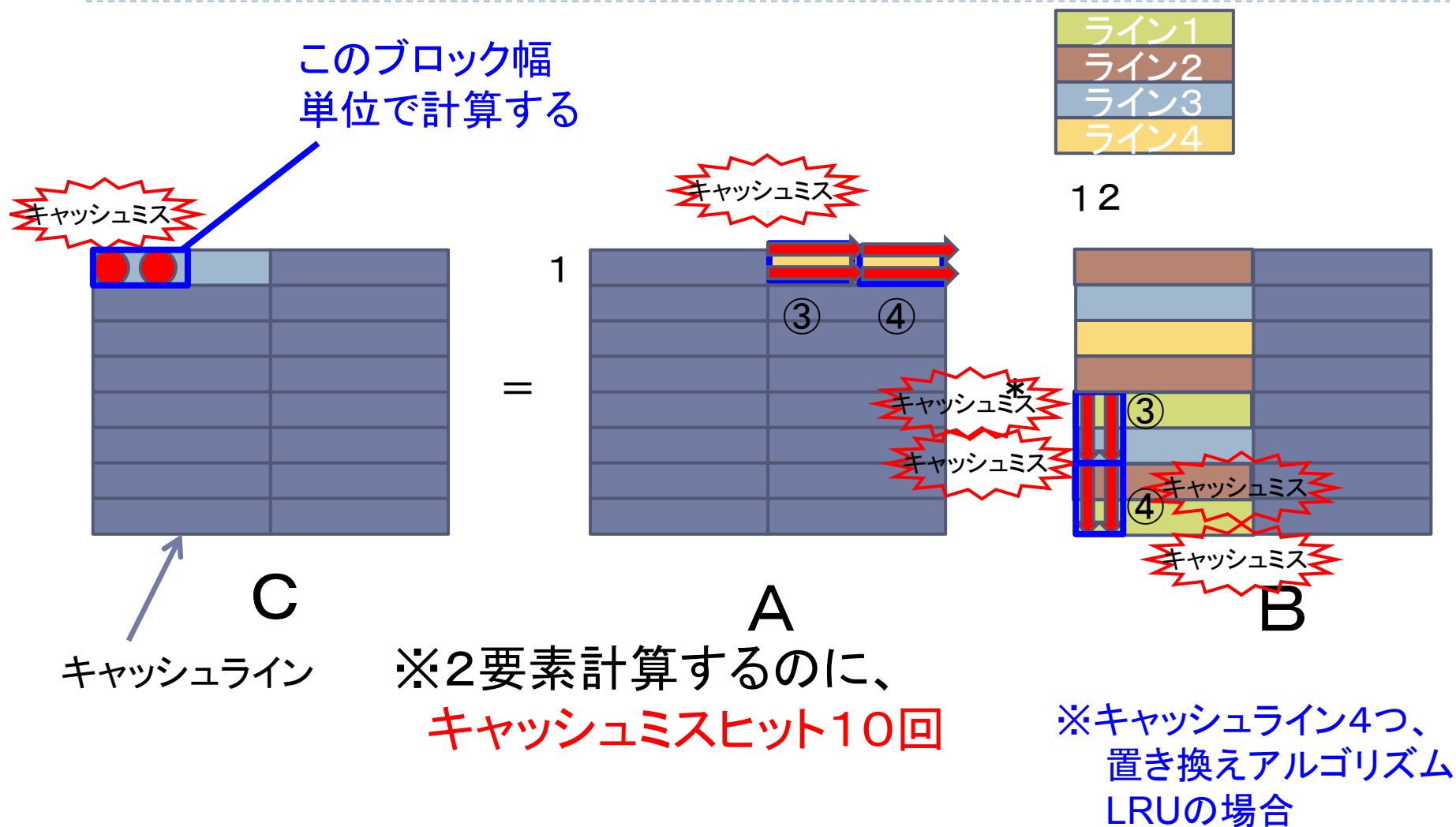
※キャッシュライン4つ、
置き換えアルゴリズム
LRUの場合

行列-行列積の場合（ブロック化する：2要素）



※キャッシュライン4つ、 置き換えアルゴリズム LRUの場合

行列-行列積の場合（ブロック化する：2要素）

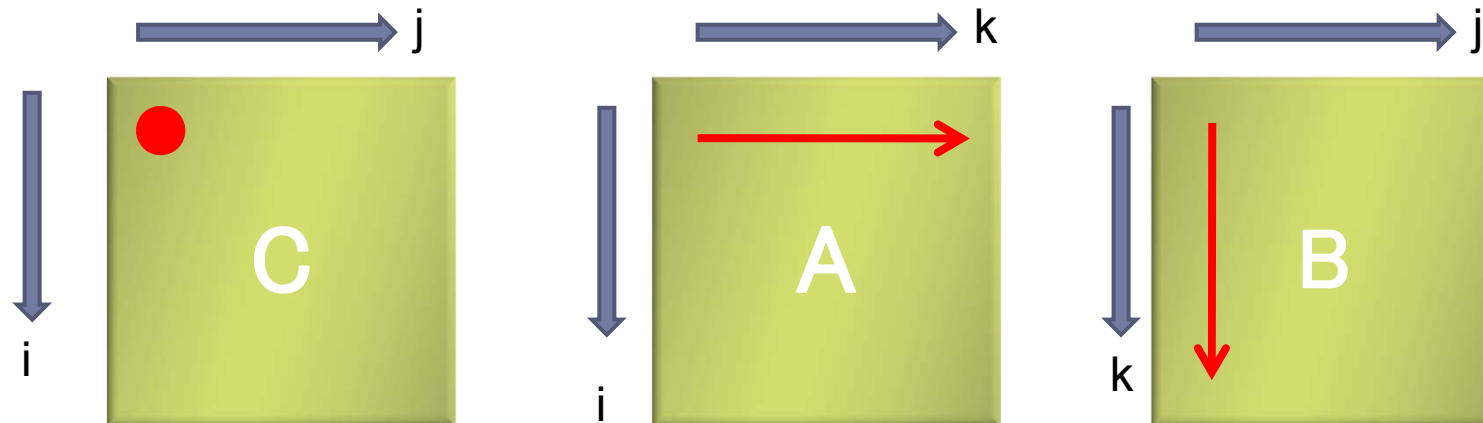


行列積コード（C言語）

：キャッシュブロッック化なし

- 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-行列積のブロック化のコード (C言語)

- ▶ n がブロック幅 ($ibl=16$)で割り切れるとき、
以下のような6重ループのコードになる

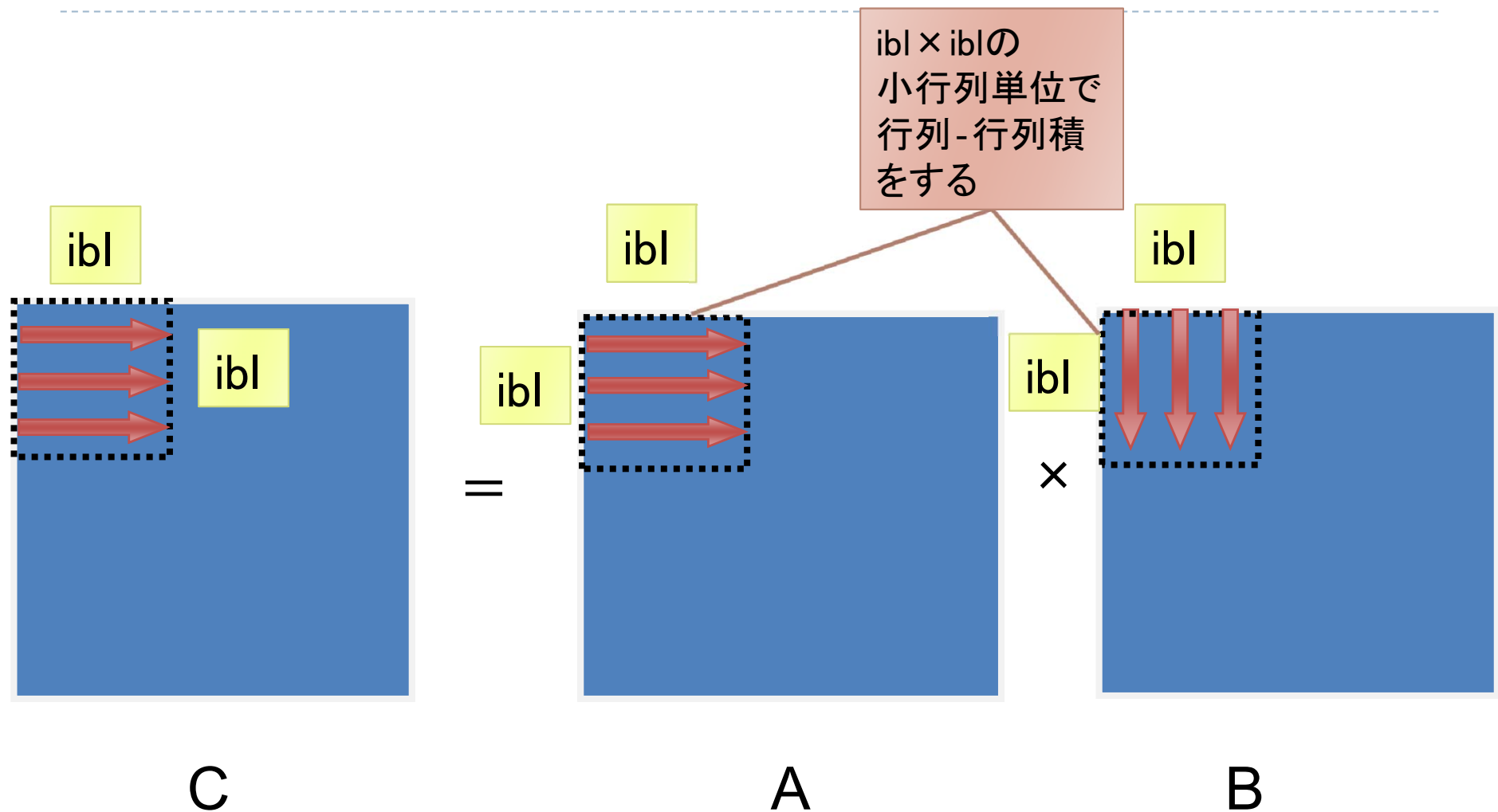
```
ibl = 16;
for ( ib=0; ib<n; ib+=ibl ) {
    for ( jb=0; jb<n; jb+=ibl ) {
        for ( kb=0; kb<n; kb+=ibl ) {
            for ( i=ib; i<ib+ibl; i++ ) {
                for ( j=jb; j<jb+ibl; j++ ) {
                    for ( k=kb; k<kb+ibl; k++ ) {
                        C[i][j] += A[i][k] * B[k][j];
                    } } } } } }
```

行列-行列積のブロック化のコード (Fortran言語)

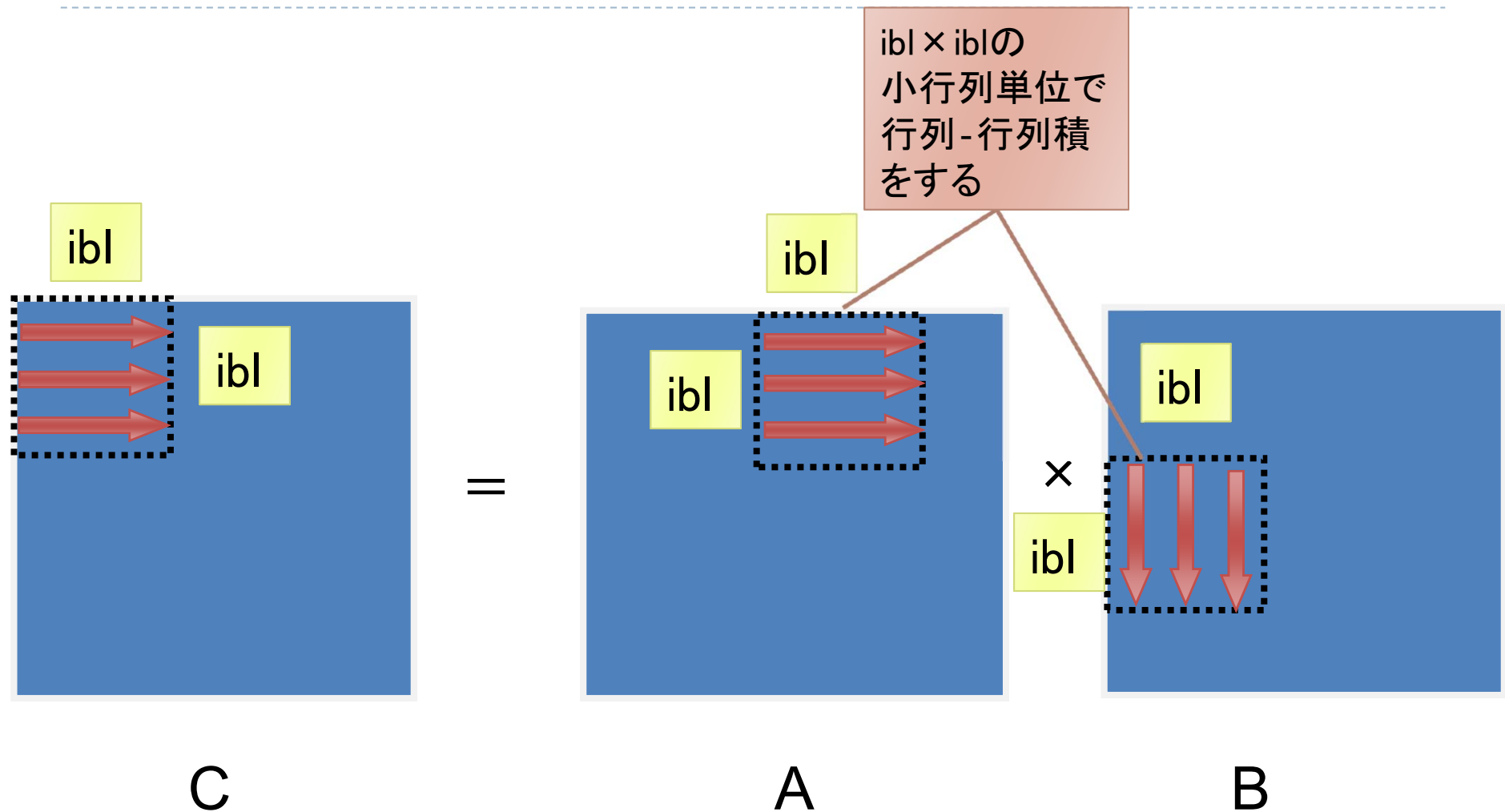
- ▶ n がブロック幅 ($ibl=16$) で割り切れるとき、
以下のような6重ループのコードになる

```
ibl = 16
do ib=1, n, ibl
  do jb=1, n, ibl
    do kb=1, n, ibl
      do i=ib, ib+ibl-1
        do j=jb, jb+ibl-1
          do k=kb, kb+ibl-1
             $C(i, j) = C(i, j) + A(i, k) * B(k, j)$ 
          enddo; enddo; enddo; enddo; enddo; enddo;
```

キャッシュブロック化時の データ・アクセスパターン



キャッシュブロック化時の データ・アクセスパターン



行列-行列積のブロック化のコードのアンローリング (C言語)

- ▶ 行列-行列積の6重ループのコードに加え、さらに各6重ループにアンローリングを施すことができる。
- ▶ i-ループ、およびj-ループ2段アンローリングは、以下のようなコードになる。(ブロック幅iblが2で割り切れる場合)

```
ibl = 16;
for (ib=0; ib<n; ib+=ibl) {
  for (jb=0; jb<n; jb+=ibl) {
    for (kb=0; kb<n; kb+=ibl) {
      for (i=ib; i<ib+ibl; i+=2) {
        for (j=jb; j<jb+ibl; j+=2) {
          for (k=kb; k<kb+ibl; k++) {
            C[i][j] += A[i][k] * B[k][j];
            C[i+1][j] += A[i+1][k] * B[k][j];
            C[i][j+1] += A[i][k] * B[k][j+1];
            C[i+1][j+1] += A[i+1][k] * B[k][j+1];
          } } } } } }
```

行列-行列積のブロック化のコードのアンローリング (Fortran言語)

- ▶ 行列-行列積の6重ループのコードに加え、さらに各6重ループにアンローリングを施すことができる。
- ▶ i-ループ、およびj-ループ2段アンローリングは、以下のようなコードになる。(ブロック幅iblが2で割り切れる場合)

```
ibl = 16
do ib=1, n, ibl
  do jb=1, n, ibl
    do kb=1, n, ibl
      do i=ib, ib+ibl, 2
        do j=jb, jb+ibl, 2
          do k=kb, kb+ibl
            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)
            C(i, j+1) = C(i, j+1) + A(i, k) * B(k, j+1)
            C(i+1, j+1) = C(i+1, j+1) + A(i+1, k) * B(k, j+1)
          enddo
        enddo
      enddo
    enddo
  enddo
enddo
```

その他の高速化技術

共通部分式の削除（１）

- ▶ 以下のプログラムは、冗長な部分がある。

```
d = a + b + c;  
f = d + a + b;
```

- ▶ コンパイラがやる場合もあるが、以下のように書く方が無難である。

```
temp = a + b;  
d = temp + c;  
f = d + temp;
```


共通部分式の削除（２）

- ▶ 配列のアクセスも、冗長な書き方をしないほうがよい。

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

- ▶ 以下のように書く。

```
for (i=0; i<n; i++) {  
    dtemp = x[i];  
    xold[i] = dtemp;  
    x[i] = dtemp + y[i];  
}
```

コードの移動

- ▶ 割り算は演算時間がかかる。ループ中に書かない。

```
for (i=0; i<n; i++) {  
    a[i] = a[i] / sqrt(dnorm);  
}
```

- ▶ 上記の例では、掛け算化して書く。

```
dtemp = 1.0d0 / sqrt(dnorm);  
for (i=0; i<n; i++) {  
    a[i] = a[i] *dtemp;  
}
```

ループ中の I F 文

- ▶ なるべく、ループ中にIF文を書かない。

```
for (i=0; i<n; i++) {  
    for (j=0; j<n; j++) {  
        if ( i != j ) A[i][j] = B[i][j];  
        else  A[i][j] = 1.0d0;  
    } }  

```

- ▶ 以下のように書く。

```
for (i=0; i<n; i++) {  
    for (j=0; j<n; j++) {  
        A[i][j] = B[i][j];  
    } }  
for (i=0; i<n; i++) A[i][j] = 1.0d0;  

```

ソフトウェア・パイプライニングの強化

- 基のコード
(2段のアンローリング)

定義－参照の距離が近い
→ソフトウェア的には
何もできない

```
for (i=0; i<n; i+=2) {  
    dtmpb0 = b[i];  
    dtmpc0 = c[i];  
    dtmpa0 = dtmpb0 + dtmpc0;  
    a[i] = dtmpa0;  
    dtmpb1 = b[i+1];  
    dtmpc1 = c[i+1];  
    dtmpa1 = dtmpb1 + dtmpc1;  
    a[i+1] = dtmpa1;  
}
```

- ソフトウェアパイプライニング
を強化したコード
(2段のアンローリング)

定義－参照の距離が遠い
→ソフトウェアパイプライニング
が適用できる機会が増加！

```
for (i=0; i<n; i+=2) {  
    dtmpb0 = b[i];  
    dtmpb1 = b[i+1];  
    dtmpc0 = c[i];  
    dtmpc1 = c[i+1];  
    dtmpa0 = dtmpb0 + dtmpc0;  
    dtmpa1 = dtmpb1 + dtmpc1;  
    a[i] = dtmpa0;  
    a[i+1] = dtmpa1;  
}
```

数値計算ライブラリの利用

数値計算ライブラリ

▶ 密行列用ライブラリ

- ▶ 行列の要素に0がない(というデータ構造を扱う)
- ▶ 連立一次方程式の解法、固有値問題、FFT、その他
- ▶ 直接解法(反復解法もある)
- ▶ BLAS、LAPACK、ScaLAPACK、SuperLU、MUMPS、FFTW、など

▶ 疎行列用ライブラリ

- ▶ 行列の要素に0が多い
- ▶ 連立一次方程式の解法、固有値問題、その他
- ▶ 反復解法
- ▶ PETSc、Xabclib、Lis、ARPACK、など

疎行列用ライブラリの特徴

- ▶ 疎行列を扱うアプリケーションはライブラリ化が難しい
 - ▶ 疎行列データ形式の標準化が困難
 - ▶ COO、CRS(CCS)、ELL、JDS、BCSR、...
 - ▶ カーネルの演算が微妙に違う、かつ、カーネルは広い範囲に分散
 - ▶ 陽解法(差分法)を基にしたソフトウェア
- ▶ 数値ミドルウェアおよび領域特化型言語(Domain Specific Language, DSL)
 - ▶ 解くべき方程式や離散化方法に特化させることで、処理(対象となるプログラムの性質)を限定
 - ▶ 以上の限定から、高度な最適化ができる言語(処理系)の作成(DSL)や、ライブラリ化(数値ミドルウェア)ができる
 - ▶ 数値ミドルウェアの例
 - ▶ ppOpen-HPC(東大)、PETSc(Argonne National Laboratory, USA.)、Trilinos(Sandia National Laboratory, USA)、など

BLAS

- ▶ BLAS (Basic Linear Algebra Subprograms、基本線形代数副プログラム集)
 - ▶ 線形代数計算で用いられる、基本演算を標準化(API化)したもの。
 - ▶ 普通は、密行列用の線形代数計算用の基本演算の副プログラムを指す。
 - ▶ 疎行列の基本演算用の<スパースBLAS>というものがあるが、まだ定着していない。
 - ▶ スパースBLASはIntel MKL(Math Kernel Library)に入っているが、広く使われているとは言えない。

BLAS

- ▶ BLASでは、以下のように分類わけをして、サブルーチンの命名規則を統一
 1. 演算対象のベクトルや行列の型(整数型、実数型、複素型)
 2. 行列形状(対称行列、三重対角行列)
 3. データ格納形式(帯行列を二次元に圧縮)
 4. 演算結果が何か(行列、ベクトル)
- ▶ 演算性能から、以下の3つに演算を分類
 - ▶ レベル1 BLAS: ベクトルとベクトルの演算
 - ▶ レベル2 BLAS: 行列とベクトルの演算
 - ▶ レベル3 BLAS: 行列と行列の演算

レベル 1 BLAS

▶ レベル1 BLAS

- ▶ ベクトル内積、ベクトル定数倍の加算、など
 - ▶ 例: $y \leftarrow \alpha x + y$
- ▶ データの読み出し回数、演算回数がほぼ同じ
- ▶ データの再利用(キャッシュに乗ったデータの再利用によるデータアクセス時間の短縮)がほとんどできない
 - ▶ 実装による性能向上が、あまり期待できない
 - ▶ ほとんど、計算機ハードウェアの演算性能
- ▶ レベル1BLASのみで演算を実装すると、演算が本来持っているデータ再利用性がなくなる
 - ▶ 例: 行列-ベクトル積を、レベル1BLASで実装

レベル2 BLAS

▶ レベル2 BLAS

▶ 行列-ベクトル積などの演算

▶ 例: $y \leftarrow \alpha A x + \beta y$

▶ 前進/後退代入演算、 $Tx = y$ (T は三角行列)を x について解く演算、を含む

▶ レベル1BLASのみの実装による、データ再利用性の喪失を回避する目的で提案

▶ 行列とベクトルデータに対して、データの再利用性あり

▶ データアクセス時間を、実装法により短縮可能

▶ (実装法により)性能向上がレベル1BLASに比べしやすい(が十分でない)

レベル3 BLAS

▶ レベル3 BLAS

▶ 行列-行列積などの演算

▶ 例: $C \leftarrow \alpha A B + \beta C$

▶ 共有記憶型の並列ベクトル計算機では、レベル2 BLASでも性能向上が達成できない。

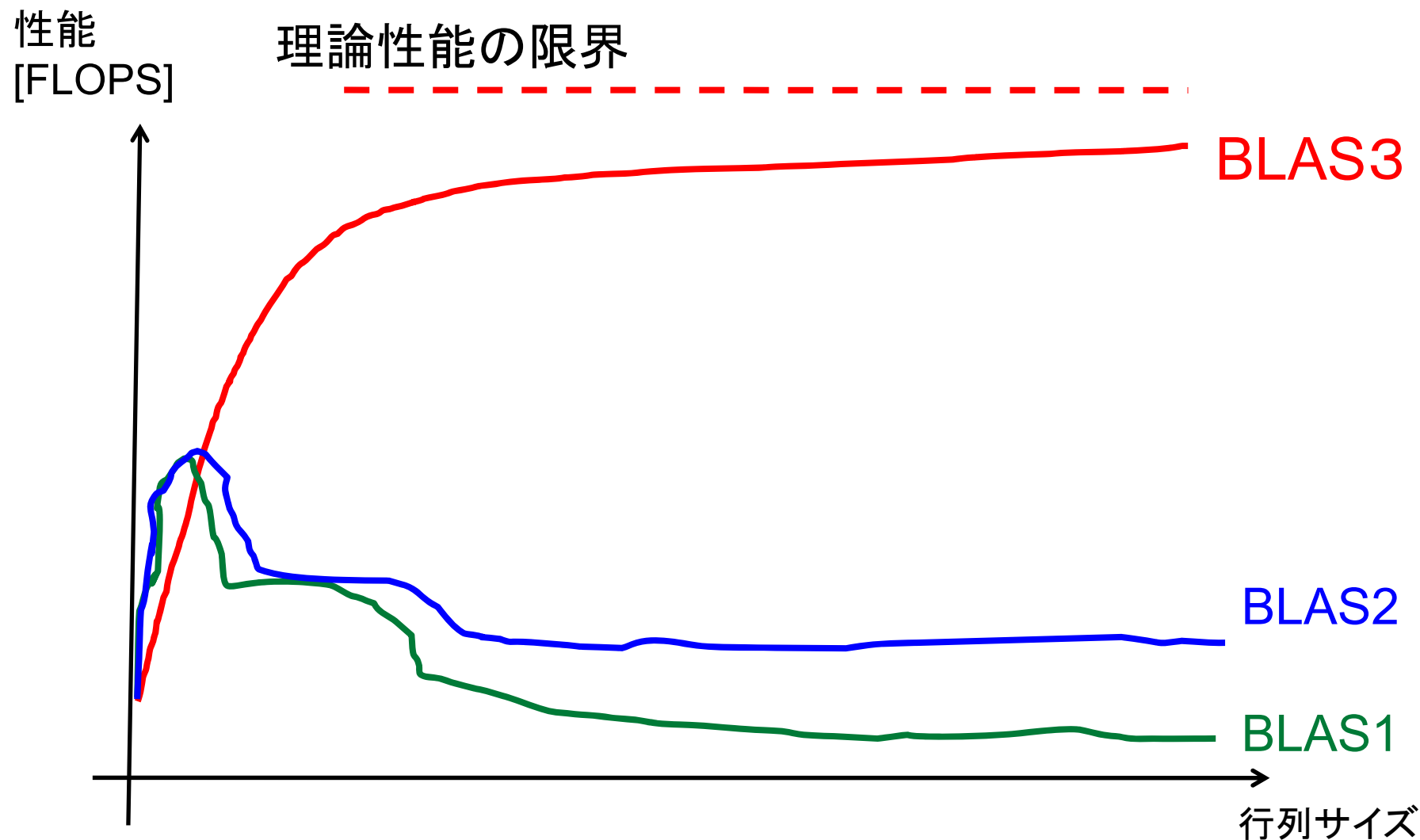
▶ 並列化により1PE当たりのデータ量が減少する。

▶ より大規模な演算をとり扱わないと、再利用の効果が無い。

▶ 行列-行列積では、行列データ $O(n^2)$ に対して演算は $O(n^3)$ なので、データ再利用性が原理的に高い。

▶ 行列積は、アルゴリズムレベルでもブロック化できる。さらにデータの局所性を高めることができる。

典型的なBLASの性能



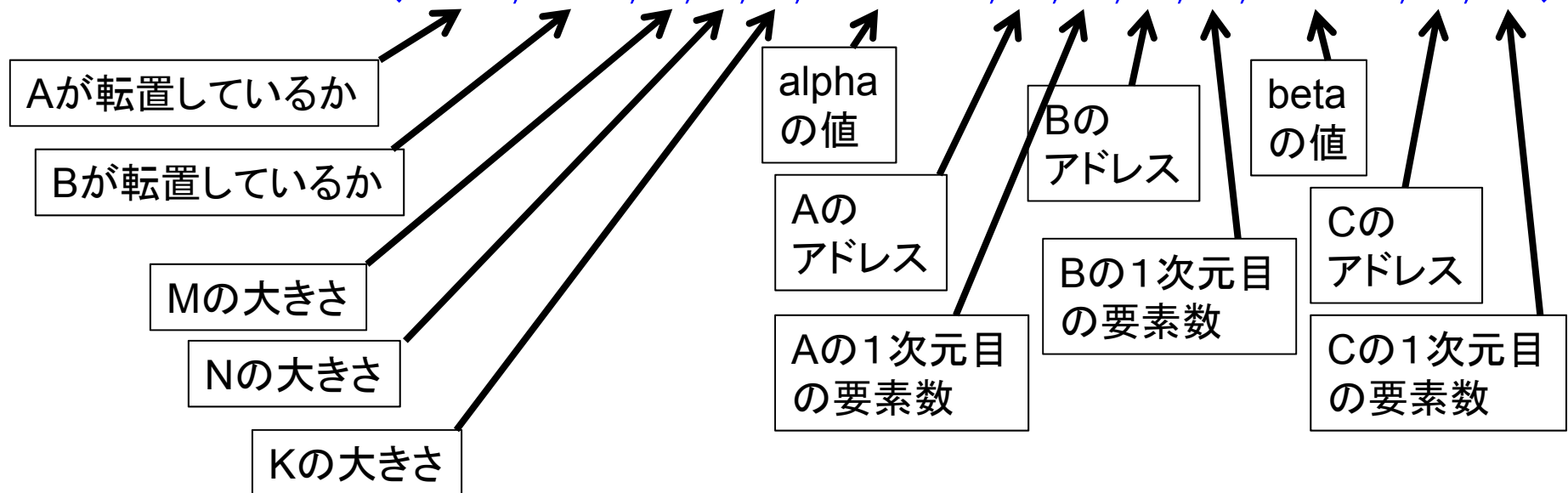
BLAS利用例

▶ 倍精度演算BLAS3

$C := \alpha * op(A) * op(B) + \beta * C$

A: M*K; B:K*N; C:M*N;

CALL DGEMM('N', 'N', n, n, n, ALPHA, A, N, B, N, BETA, C, N)



BLASの機能詳細

- ▶ 詳細はHP: <http://www.netlib.org/blas/>
- ▶ 命名規則: 関数名: **X****YYYY**
 - ▶ **X**: データ型
S:単精度、D:倍精度、C:複素、Z:倍精度複素
 - ▶ **YYYY**: 計算の種類
 - ▶ レベル1:
例: AXPY: ベクトルをスカラー倍して加算
 - ▶ レベル2:
例: GEMV: 一般行列とベクトルの積
 - ▶ レベル3:
例: GEMM: 一般行列どうしの積

GOTO BLASとは

- ▶ 後藤和茂 氏により開発された、ソースコードが無償入手可能な、高性能BLASの実装(ライブラリ)
- ▶ 特徴
 - ▶ マルチコア対応がなされている
 - ▶ 多くのコモディティハードウェア上の実装に特化
 - ▶ Intel Nehalem and Atom systems
 - ▶ VIA Nanoprocessor
 - ▶ AMD Shanghai and Istanbul
- 等
- ▶ テキサス大学先進計算センター(TACC)で、GOTO BLAS2として、ソースコードを配布している
 - ▶ HP: <http://www.tacc.utexas.edu/tacc-projects/gotoblas2/>

LAPACK

- ▶ 密行列に対する、連立一次方程式の解法、および固有値の解法の“標準”アルゴリズムルーチンを無償で提供
- ▶ その道の大学の専門家が集結
 - ▶ カリフォルニア大バークレー校：
James Demmel教授
 - ▶ テネシー大ノックスビル校：
Jack Dongarra教授
- ▶ HP
<http://www.netlib.org/lapack/>

LAPACKの命名規則

▶ 命名規則： 関数名：**XYYZZZ**

▶ **X**: データ型

S:単精度、D:倍精度、C:複素、Z:倍精度複素

▶ **YY**: 行列の型

BD: 二重対角、DI: 対角、GB: 一般帯行列、GE: 一般行列、
HE: 複素エルミート、HP: 複素エルミート圧縮形式、SY: 対称
行列、...

▶ **ZZZ**: 計算の種類

TRF: 行列の分解、TRS: 行列の分解を使う、CON: 条件数
の計算、RFS: 計算解の誤差範囲を計算、TRI: 三重対角行
列の分解、EQU: スケーリングの計算、...

インタフェース例：DGESV (1 / 3)

▶ DGESV

(N, NRHS, A, LDA, IPIVOT, B, LDB, INFO)

- ▶ $A X = B$ の解の行列 X を計算をする
- ▶ $A * X = B$ 、ここで A は $N \times N$ 行列で、 X と B は $N \times NRHS$ 行列とする。
- ▶ 行交換の部分枢軸選択付きのLU分解で A を $A = P * L * U$ と分解する。ここで、 P は交換行列、 L は下三角行列、 U は上三角行列である。
- ▶ 分解された A は、連立一次方程式 $A * X = B$ を解くのに使われる。

▶ 引数

▶ N (入力) – INTEGER

- ▶ 線形方程式の数。行列 A の次元数。 $N \geq 0$ 。

インタフェース例：DGESV (2 / 3)

▶ NRHS (入力) – INTEGER

- ▶ 右辺ベクトルの数。行列Bの次元数。NRHS ≥ 0 。

▶ A (入力／出力) – DOUBLE PRECISION, DIMENSION(:, :)

- ▶ 入力時は、 $N \times N$ の行列Aの係数を入れる。
- ▶ 出力時は、Aから分解された行列Lと $U = P * L * U$ を圧縮して出力する。
Lの対角要素は1であるので、収納されていない。

▶ LDA (入力) – INTEGER

- ▶ 配列Aの最初の次元の大きさ。LDA $\geq \max(1, N)$ 。

▶ IPIVOT (出力) – DOUBLE PRECISION, DIMENSION(:)

- ▶ 交換行列Aを構成する枢軸のインデックス。行列のi行がIPIVOT(i)行と交換されている。

インタフェース例：DGESV (3 / 3)

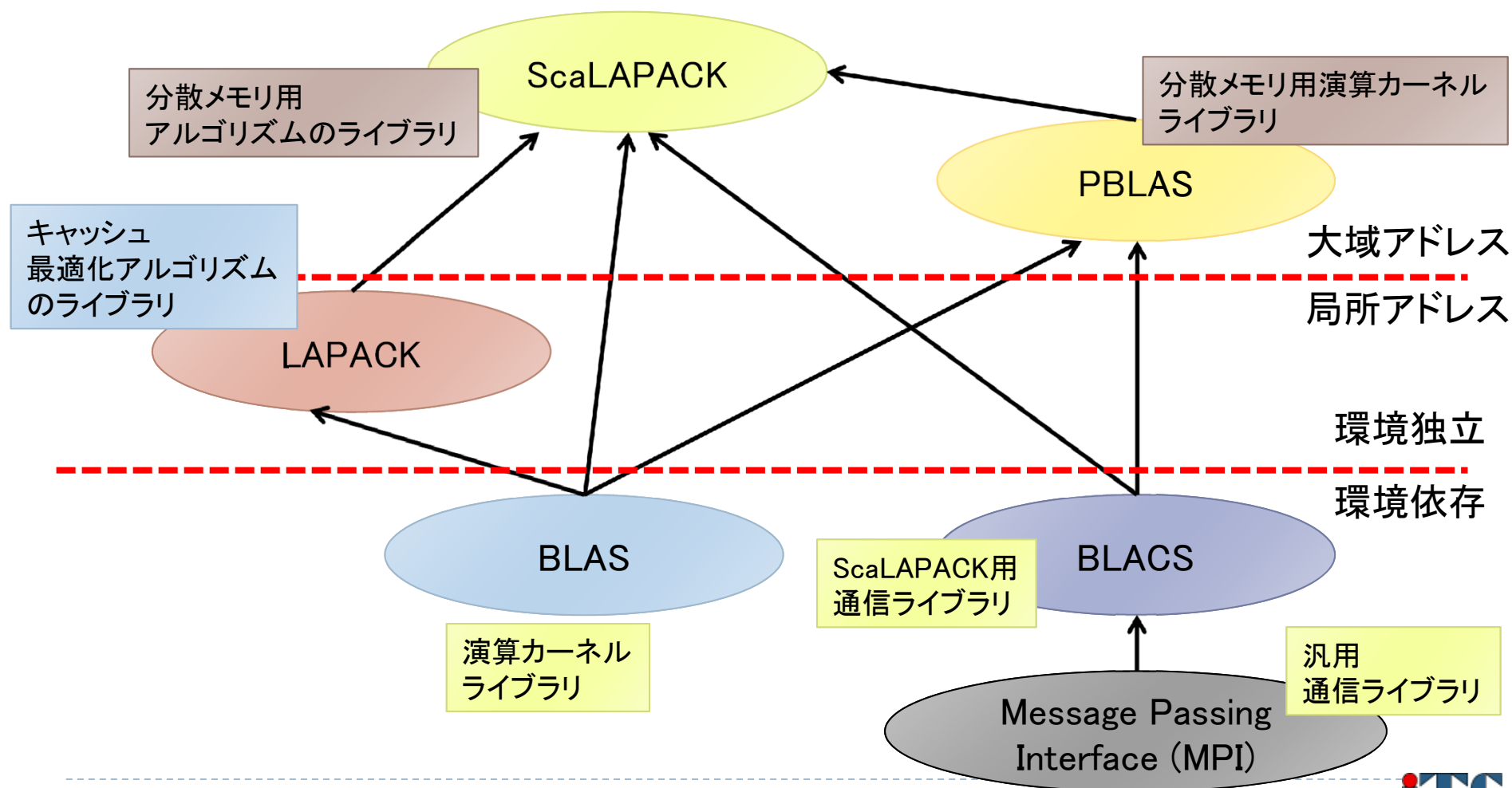
- ▶ **B (入力／出力) – DOUBLE PRECISION, DIMENSION(:,:)**
 - ▶ 入力時は、右辺ベクトルの $N \times \text{NRHS}$ 行列Bを入れる。
 - ▶ 出力時は、もし、 $\text{INFO} = 0$ なら、 $N \times \text{NRHS}$ 行列である解行列Xが戻る。
- ▶ **LDB (入力) – INTEGER**
 - ▶ 配列Bの最初の次元の大きさ。 $\text{LDB} \geq \max(1, N)$ 。
- ▶ **INFO (出力) – INTEGER**
 - ▶ $= 0$: 正常終了
 - ▶ < 0 : もし $\text{INFO} = -i$ なら i -th 行の引数の値がおかしい。
 - ▶ > 0 : もし $\text{INFO} = i$ なら $U(i,i)$ が厳密に0である。分解は終わるが、Uの分解は特異なため、解は計算されない。

ScaLAPACK

- ▶ 密行列に対する、連立一次方程式の解法、および固有値の解法の“標準”アルゴリズムルーチンの並列化版を無償で提供
- ▶ ユーザインタフェースはLAPACKに＜類似＞
- ▶ ソフトウェアの＜階層化＞がされている
 - ▶ 内部ルーチンはLAPACKを利用
 - ▶ 並列インタフェースはBLACS
- ▶ データ分散方式に、2次元ブロック・サイクリック分散方式を採用（詳細は、「MPI」の講義で説明）
- ▶ HP: <http://www.netlib.org/scalapack/>

ScaLAPACKのソフトウェア構成図

出典: <http://www.netlib.org/scalapack/poster.html>



BLACSとPBLAS

▶ BLACS

- ▶ ScaLAPACK中で使われる通信機能を関数化したもの。
- ▶ 通信ライブラリは、MPI、PVM、各社が提供する通信ライブラリを想定し、ScaLAPACK内でコード修正せずに使うことを目的とする
 - ▶ いわゆる、通信ライブラリのラッパー的役割でScaLAPACK内で利用
- ▶ 現在、MPIがデファクトになったため、MPIで構築されたBLACSのみ、現実的に利用されている。
 - ▶ なので、ScaLAPACKはMPIでコンパイルし、起動して利用する

▶ PBLAS

- ▶ BLACSを用いてBLASと同等な機能を提供する関数群
- ▶ 並列版BLASといってよい。

ScaLAPACKの命名規則

- ▶ 原則:
LAPACKの関数名の頭に“P”を付けたもの
- ▶ そのほか、BLACS、PBLAS、データ分散を制御するためのScaLAPACK用関数がある。

インタフェース例：PDGESV (1 / 4)

▶ PDGESV

(N, NRHS, A, IA, JA, DESCA, IPIV, B, IB, JB, DESCB, INFO)

- ▶ $sub(A) X = sub(B)$ の解の行列 X を計算をする
- ▶ ここで $sub(A)$ は $N \times N$ 行列を分散した $A(IA:IA+N-1, JA:JA+N-1)$ の行列
- ▶ X と B は $N \times NRHS$ 行列を分散した $B(IB:IB+N-1, JB:JB+NRHS-1)$ の行列
- ▶ 行交換の部分枢軸選択付きのLU分解 で $sub(A)$ を $sub(A) = P * L * U$ と分解する。ここで、 P は交換行列、 L は下三角行列、 U は上三角行列である。
- ▶ 分解された $sub(A)$ は、連立一次方程式 $sub(A) * X = sub(B)$ を解くのに使われる。

インタフェース例：PDGESV (2 / 4)

- ▶ **N (大域入力) – INTEGER**
 - ▶ 線形方程式の数。行列Aの次元数。 $N \geq 0$ 。
- ▶ **NRHS (大域入力) – INTEGER**
 - ▶ 右辺ベクトルの数。行列Bの次元数。 $NRHS \geq 0$ 。
- ▶ **A (局所入力／出力) – DOUBLE PRECISION, DIMENSION(:, :)**
 - ▶ 入力時は、 $N \times N$ の行列Aの局所化された係数を配列A(LLD_A, LOCc(JA+N-1))を入れる。
 - ▶ 出力時は、Aから分解された行列Lと $U = P * L * U$ を圧縮して出力する。Lの対角要素は1であるので、収納されていない。
- ▶ **IA (大域入力) – INTEGER** : $sub(A)$ の最初の行のインデックス
- ▶ **JA (大域入力) – INTEGER** : $sub(A)$ の最初の列のインデックス
- ▶ **DESCA (大域かつ局所入力) – INTEGER**
 - ▶ 分散された配列Aの記述子。

インタフェース例：PDGESV (3 / 4)

- ▶ **IPIVOT (局所出力) – DOUBLE PRECISION, DIMENSION(:)**
 - ▶ 交換行列Aを構成する枢軸のインデックス。行列のi行がIPIVOT(i)行と交換されている。分散された配列($\text{LOCr}(\text{M_A}) + \text{MB_A}$)として戻る。
- ▶ **B (局所入力／出力) – DOUBLE PRECISION, DIMENSION(:, :)**
 - ▶ 入力時は、右辺ベクトルの $N \times \text{NRHS}$ の行列Bの分散されたものを(LLD_B , $\text{LOCc}(\text{JB} + \text{NRHS} - 1)$)に入れる。
 - ▶ 出力時は、もし、 $\text{INFO} = 0$ なら、 $N \times \text{NRHS}$ 行列である解行列Xが、行列Bと同様の分散された状態で戻る。
- ▶ **IB(大域入力) – INTEGER**
 - ▶ $\text{sub}(B)$ の最初の行のインデックス
- ▶ **JB(大域入力) – INTEGER**
 - ▶ $\text{sub}(B)$ の最初の列のインデックス
- ▶ **DESCB (大域かつ局所入力) – INTEGER**
 - ▶ 分散された配列Bの記述子。

インタフェース例：PDGESV (4 / 4)

▶ INFO (大域出力) —INTEGER

- ▶ = 0: 正常終了
- ▶ < 0:
 - もし i 番目の要素が配列で、その j 要素の値がおかしいなら、 $\text{INFO} = -(i*100+j)$ となる。
 - もし i 番目の要素がスカラーで、かつ、その値がおかしいなら、 $\text{INFO} = -i$ となる。
- ▶ > 0: もし $\text{INFO} = K$ のとき $U(\text{IA}+K-1, \text{JA}+K-1)$ が厳密に0である。分解は完了するが、分解された U は厳密に特異なので、解は計算できない。

BLAS利用の注意

▶ C言語からの利用

- ▶ BLASライブラリは(たいてい)Fortranで書かれている
- ▶ 行列を1次元で確保する
- ▶ 引数は全てポインタで引き渡す
- ▶ 関数名の後に“_”をつける(BLASをコンパイルするコンパイラ依存)
 - ▶ 例: dgemm_(...)

▶ 小さい行列は性能的に注意

- ▶ キャッシュに載るようなサイズ(例えば、100次元以下)の行列については、BLASが高速であるとは限らない
 - ▶ BLASは、大規模行列で高性能になるように設計されている
- ▶ 全体の行列サイズは大きくても、利用スレッド数が増えると、スレッド当たりの行列サイズが小さくなるので注意！
 - ▶ 例) N=8000でも、64スレッド並列だと、スレッドあたりN=1000まで小さくなる

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グ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。