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

第１講
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

東京大学情報基盤センター
INFORMATION TECHNOLOGY CENTER, THE UNIVERSITY OF TOKYO

本講義の位置づけ

講義日程と内容について

- 2015年9月12日（土） 第1回並列プログラミング講習会 座学「並列プログラミング入門」in 金沢
  - 第1講：プログラム高速化の基礎、10:30-12:00
    - 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講：並列処理とMPIの基礎、13:00-14:30
    - 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講：OpenMPの基礎、14:45-16:15
    - OpenMPの基礎、利用方法、その他
  - 第4講：Hybrid並列化技法(MPIとOpenMPの応用)、16：30-18:00
    - 背景、Hybrid並列化の適用事例、利用上の注意、その他
    - 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）

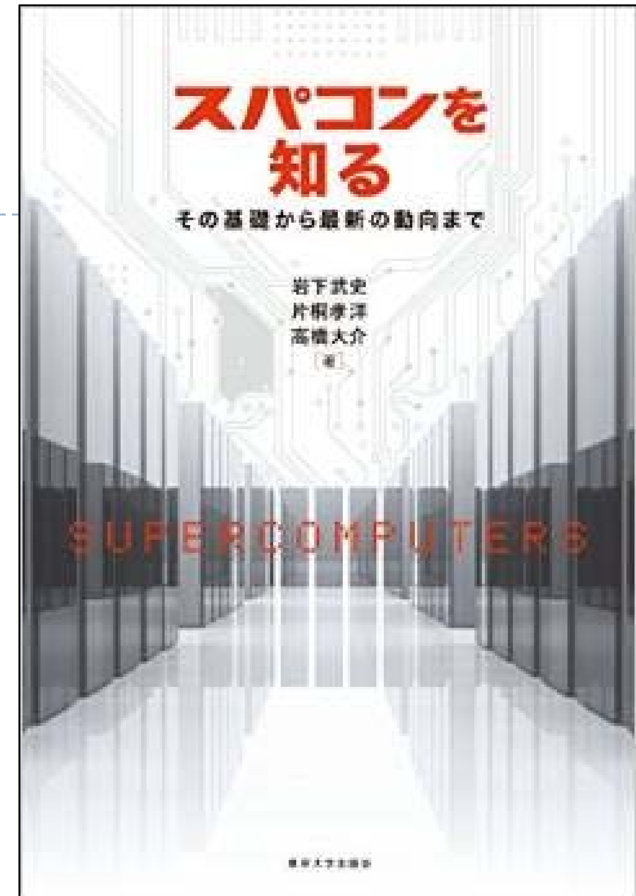
- 「並列プログラミング入門：
  サンプルプログラムで学ぶOpenMPとOpenACC」
  - 片桐 孝洋　著
  - 東大出版会、ISBN-10: 4130624563、
    ISBN-13: 978-4130624565、発売日： 2015年5月25日
  - 【本書の特徴】
    - C言語、Fortran90言語で解説
    - C言語、Fortran90言語の複数のサンプルプログラムが入手可能
      （ダウンロード形式）
    - 本講義の内容を全てカバー
    - Windows PC演習可能(Cygwin利用)。スパコンでも演習可能。
    - 内容は初級。初めて並列プログラミングを学ぶ人向けの
      入門書

教科書（演習書）

- 「スパコンプログラミング入門
  －並列処理とMPIの学習－」
  - 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

- 「スパコンを知る:
その基礎から最新の動向まで」
  - 岩下武史、片桐孝洋、高橋大介 著
  - 東大出版会、ISBN-10: 4130634550、
ISBN-13: 978-4130634557、
発売日：2015年2月18日、176頁
  - 【本書の特徴】
    - スパコンの解説書です。以下を
分かりやすく解説しています。
      - □スパコンは何に使えるか
      - □スパコンはどんな仕組みで、なぜ速く計算できるのか
      - □最新技術、今後の課題と将来展望、など



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
    ㈱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（スパコンプログラミング入門）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26年8月14日公布、9月15日施行）の規制対象デジタル電子計算機
    - 第7条第三項ハ：デジタル電子計算機であって、加重最高性能が八・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0.1京（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4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 GHz * 8回浮動小数点演算/Hz * 6コア = 158.4 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！

スーパコンピュータ用語

- 理論性能（Theoretical　Performance）
  - ハードウエ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エ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。

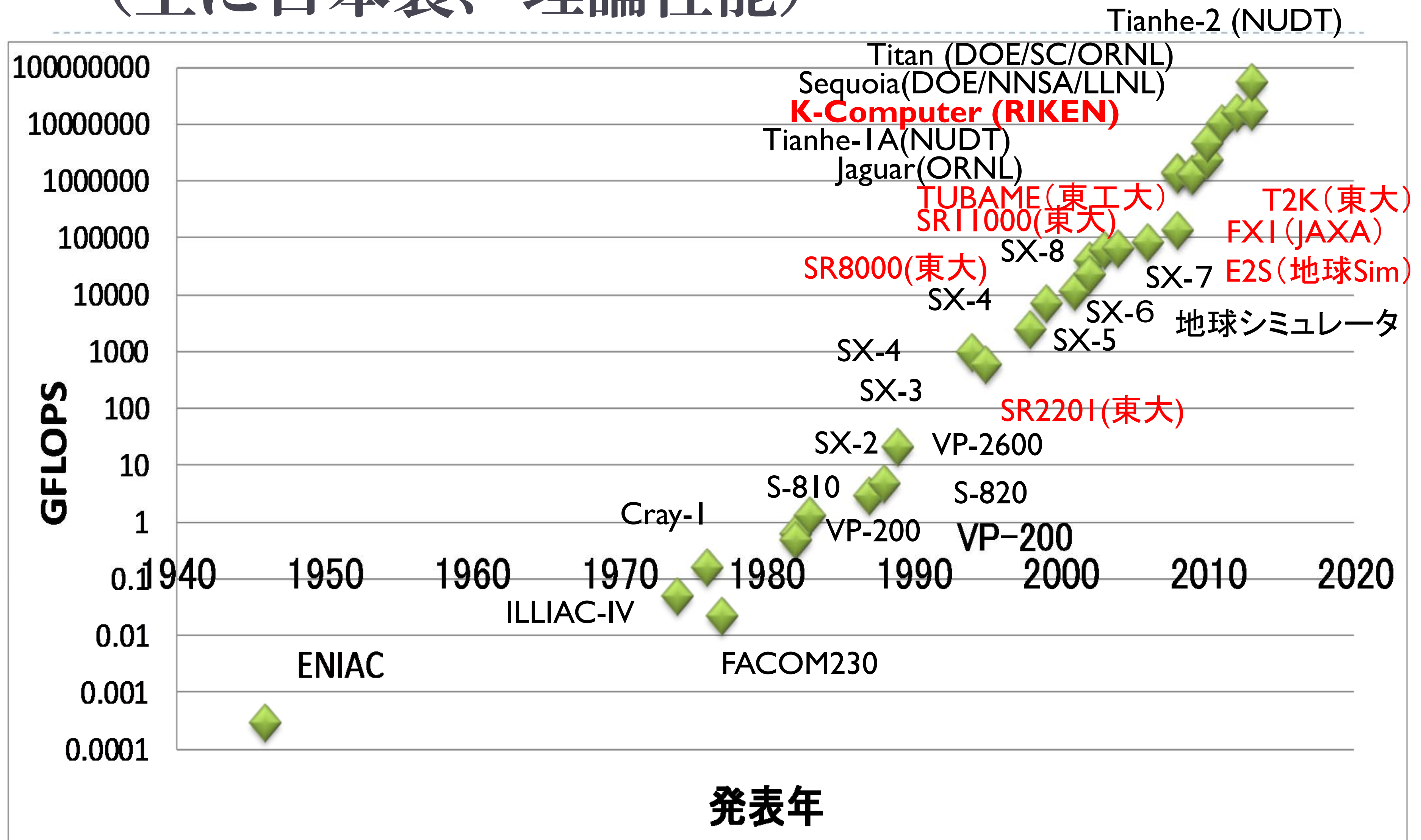
「半導体チップの集積度は、およそ18ヵ月で2倍になる」

- 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

2015年6月現在

- 1位：中国 NUDTのTianhe-2
  33.862 PFLOPS
- 2位：米国 DOE/SC/ORNLのTitan
  17.590 PFLOPS
- 3位：米国 DOE/NNSA/LLNLのSequoia (BlueGene/Q)
  17.173 PFLOPS
- **4位：日本　K-Computer (Sparc64 XIIIfx)**
  10.510 PFLOPS
- 5位：米国 DOE/SC/ANLのMira (BlueGene/Q)
  8.586 PFLOPS
- 6位：スイス国立スパコンセンターの Piz Daint (Cray XC30)
  6.271 PFLOPS
- その他の日本のマシン
  - 22位の東工大のTUBAME2.5
    2.785 PFLOPS
  - 27位の核融合研のFX100 (Sparc64 XIfx)
    2.376 PFLOPS
  - 65位の東京大学情報基盤センターの Oakleaf-fx (Sparc64 IVfx)
    1.042 PFLOPS

TOP500 List - June 2010 (1-100)

R_{max} and R_{peak} values are in TFlops. For more details about other fields, check the TOP500 description.

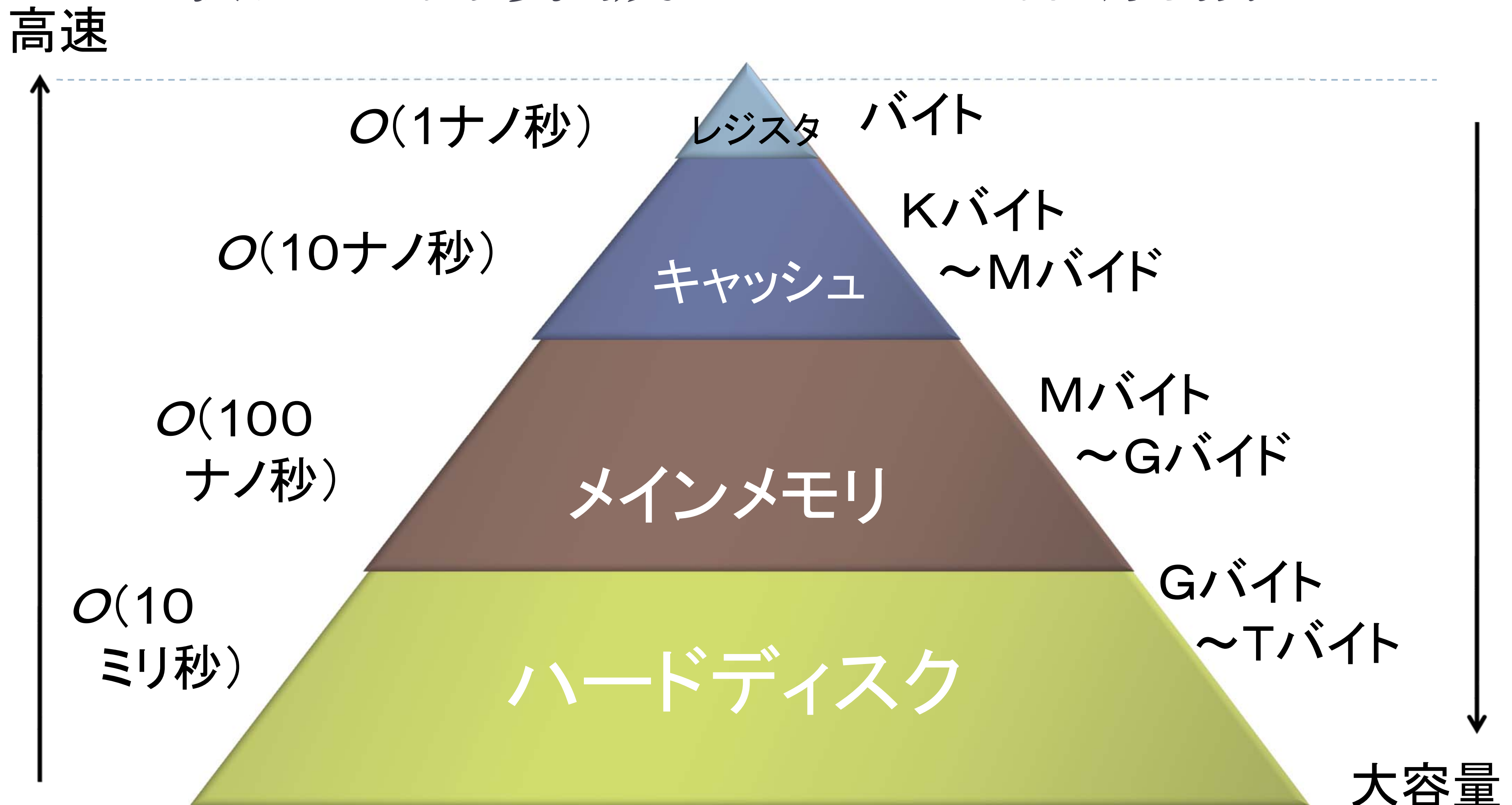
Power data in KW for entire system

Rank	Site	Computer/Year Vendor	Cores	R_{max}	R_{peak}	Power
1	Oak Ridge National Laboratory United States	Jaguar - Cray XT5-HE Opteron Six Core 2.6 GHz / 2009 Cray Inc.	224162	1759.00	2331.00	6950.60
2	National Supercomputing Centre in Shenzhen (NSCS) China	Nebulae - Dawning TC3600 Blade, Intel X5650, NVidia Tesla C2050 GPU / 2010 Dawning	120640	1271.00	2984.30	
3	DOE/NNSA/LANL United States	Roadrunner - BladeCenter QS22/LS21 Cluster, PowerXCell 8i 3.2 Ghz / Opteron DC 1.8 GHz, Voltaire Infiniband / 2009 IBM	122400	1042.00	1375.78	2345.50
4	National Institute for Computational Sciences/University of Tennessee United States	Kraken XT5 - Cray XT5-HE Opteron Six Core 2.6 GHz / 2009 Cray Inc.	98928	831.70	1028.85	
5	Forschungszentrum Juelich (FZJ) Germany	JUGENE - Blue Gene/P Solution / 2009 IBM	294912	825.50	1002.70	2268.00
6	NASA/Ames Research Center/NAS United States	Pleiades - SGI Altix ICE 8200EX/8400EX, Xeon HT QC 3.0/Xeon Westmere 2.93 Ghz, Infiniband / 2010 SGI	81920	772.70	973.29	3096.00
7	National SuperComputer Center in Tianjin/NUDT China	Tianhe-1 - NUDT TH-1 Cluster, Xeon E5540/E5450, ATI Radeon HD 4870 2, Infiniband / 2009 NUDT	71680	563.10	1206.19	

http://www.top500.org/lists/2015/06/

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

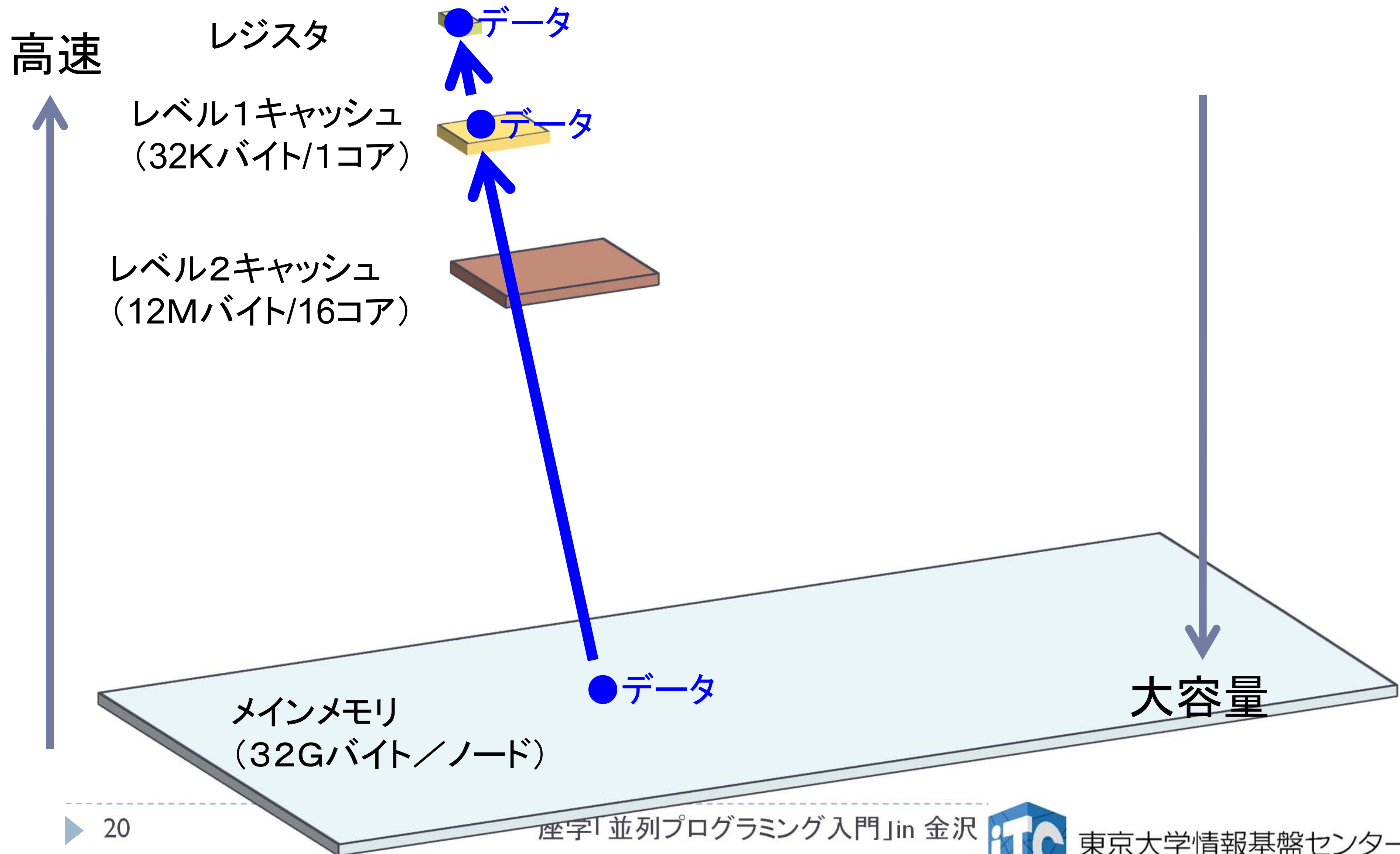
キャッシュ

メインメモリ

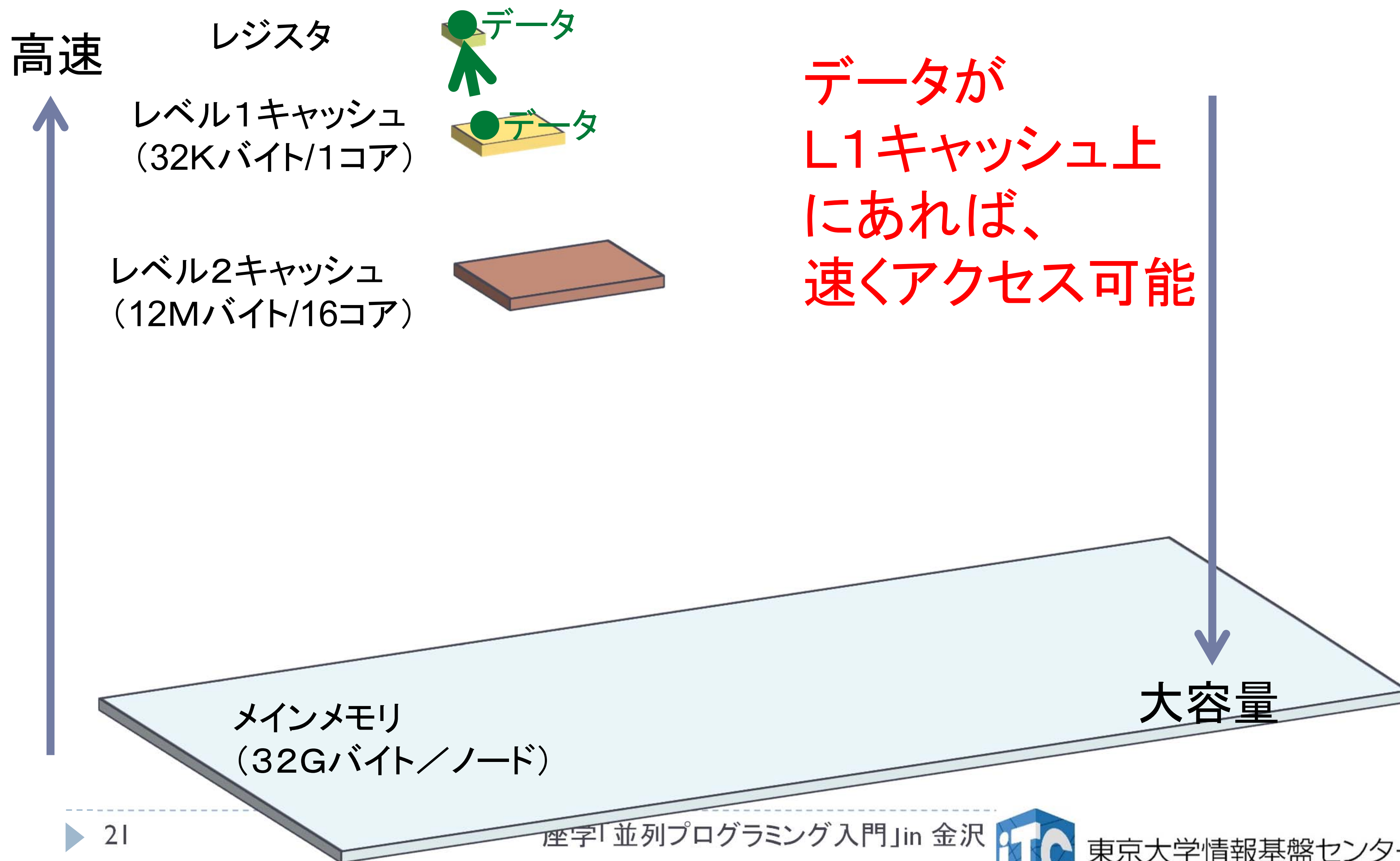
- 高性能（＝速い）プログラミングをするには、きわめて小容量のデータ範囲について何度もアクセス（＝局所アクセス）するようにループを書くしかない

iTC 東京大学情報基盤センター INFORMATION TECHNOLOGY CENTER, THE UNIVERSITY OF TOKYO

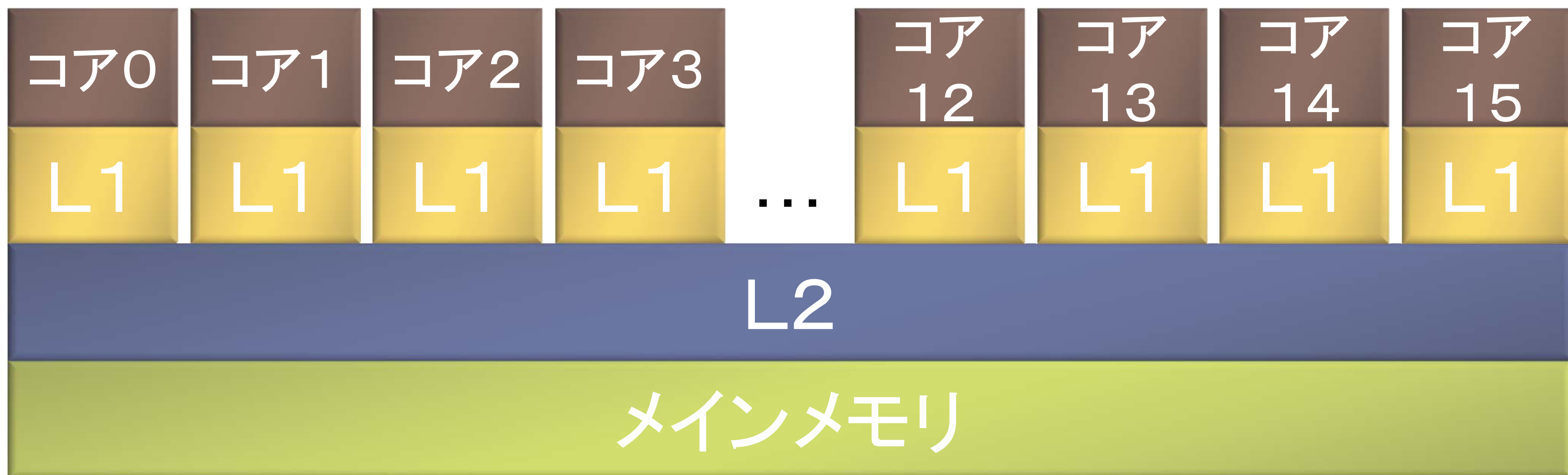
東京大学FX10のメモリ構成例



東京大学FX10のメモリ構成例

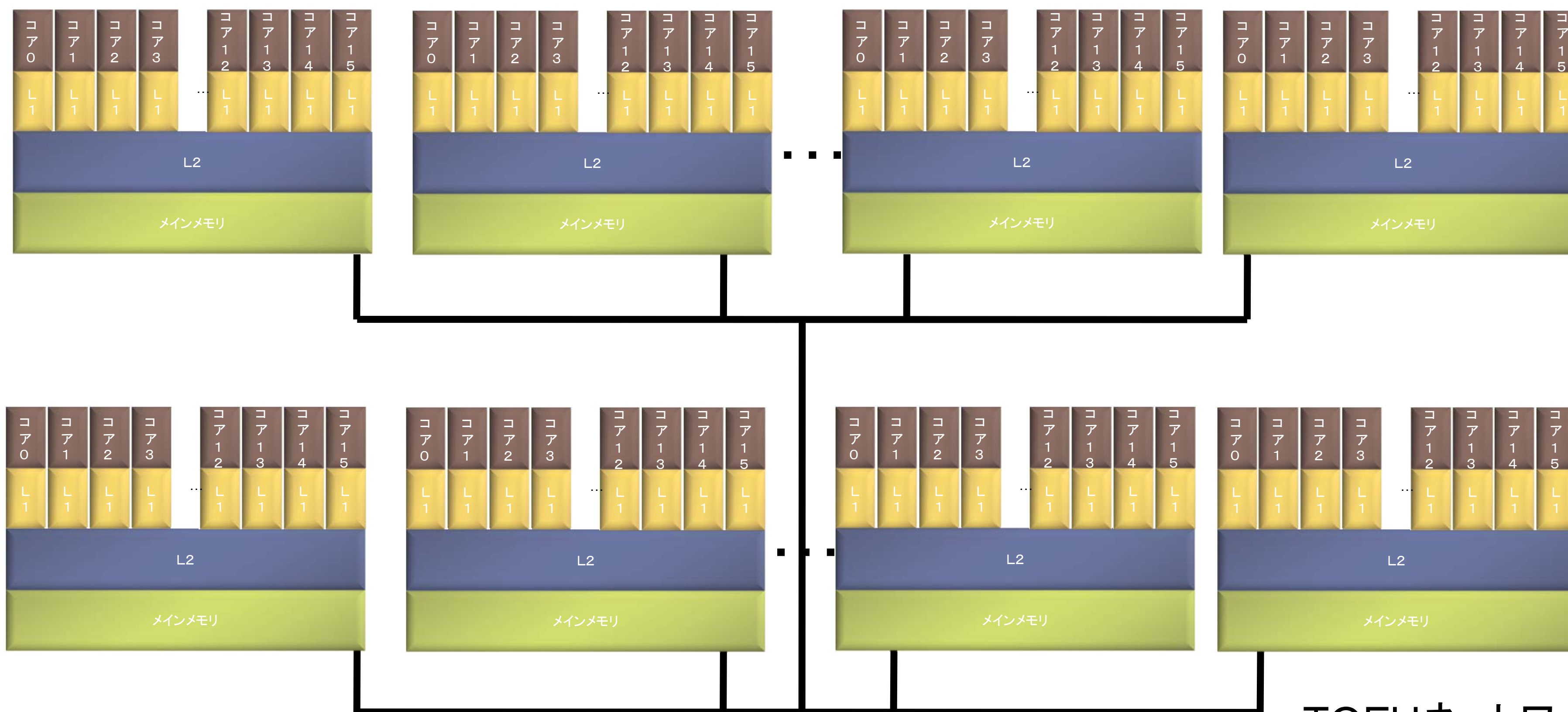


東京大学ＦＸ１０のノードのメモリ構成例



※階層メモリ構成となっている

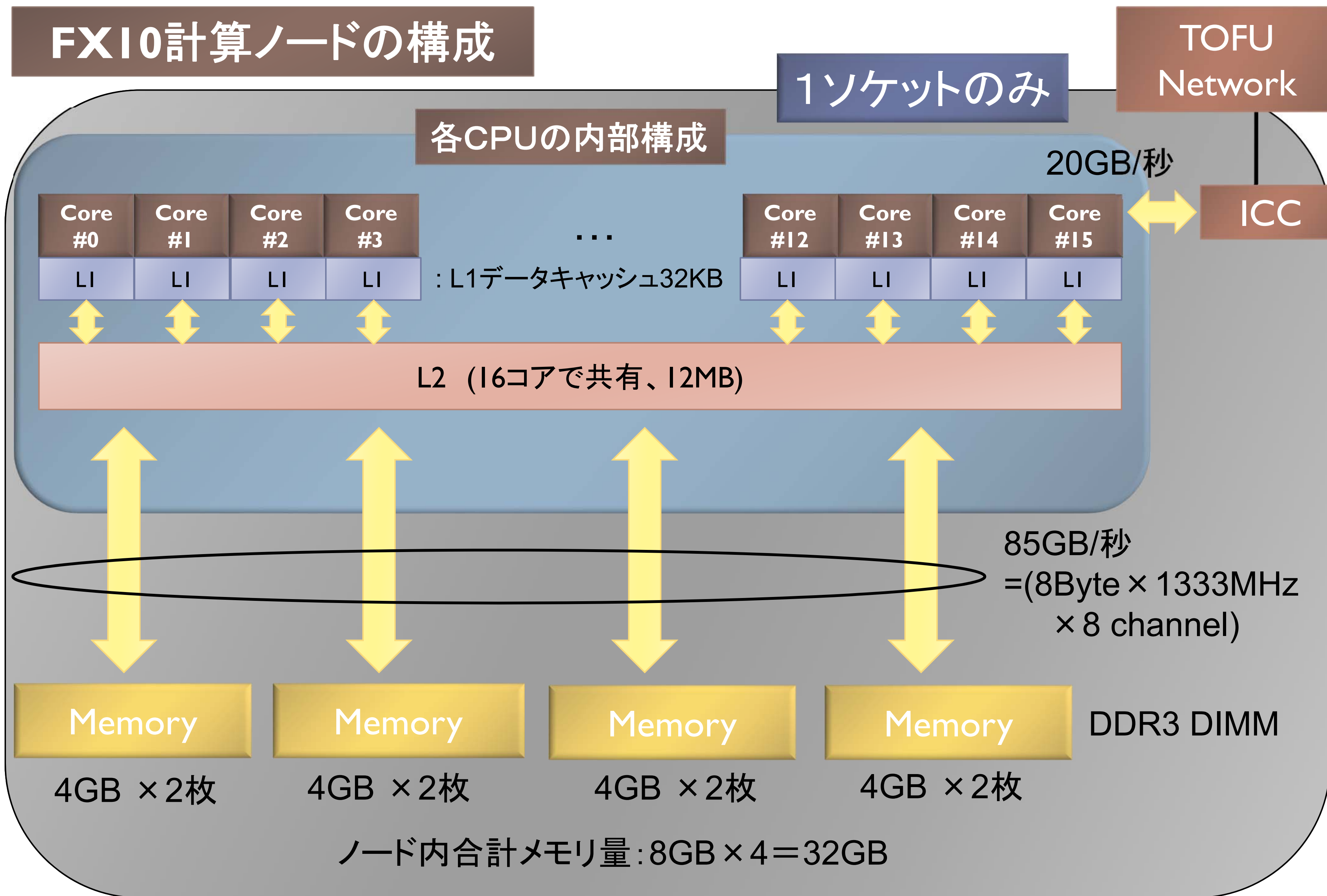
東京大学FX10全体メモリ構成



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

FUJITSU Supercomputer PRIMEHPC FX100

- FX10の後継であるFX100では、以下が拡張
- CPU：SPARC64 XI fx
  - 32演算コア　＋　2アシスタントコア
  - 理論演算性能：1TFLOPS以上（倍精度）、2TFLOPS以上（単精度）
    - EU：　2個の整数演算ユニット、2個の整数演算兼アドレス計算ユニット、および8個の浮動小数点積和演算ユニット（FMA）
    - 1個のFMAは、1サイクルあたり2つの倍精度浮動小数点演算（加算と乗算）を実行可能
    - SIMD：1つのSIMD演算命令で4個のFMAが動作。コア内：1サイクルあたり2個のSIMD演算命令を実行
    - →各コアで1サイクルあたり16個、32コア合計で512個の倍精度浮動小数点演算が実行可能
  - SIMD：256ビット。4個の倍精度浮動小数点積和演算、もしくは8個の単精度浮動小数点積和演算。ストライドSIMDロードストア命令。間接SIMDロードストア命令。並べ替え。
  - L1キャッシュ：64KB、L2キャッシュ：24MB
  - 乱発行(Out-of-order) リソースの増加

FUJITSU Supercomputer PRIMEHPC FX100

- FX10の後継であるFX100では、以下が拡張
- ノード
  - メモリ容量：32GB（HMC）
  - メモリバンド幅：240GB/s（read）+ 240GB/s（write）
  - インターコネクト：Tofuインターコネクト2
  - インターコネクトバンド幅：12．5GB/s × 2（双方向）/ リンク

出典：
http://img.jp.fujitsu.com/downloads/jp/jhpc/primehpc/primehpc-fx100-hard-ja.pdf

出典：
http://www.fujitsu.com/global/Images/fujitsu-new-supercomputer-delivering-the-next-step-in-exascale-capability.pdf

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

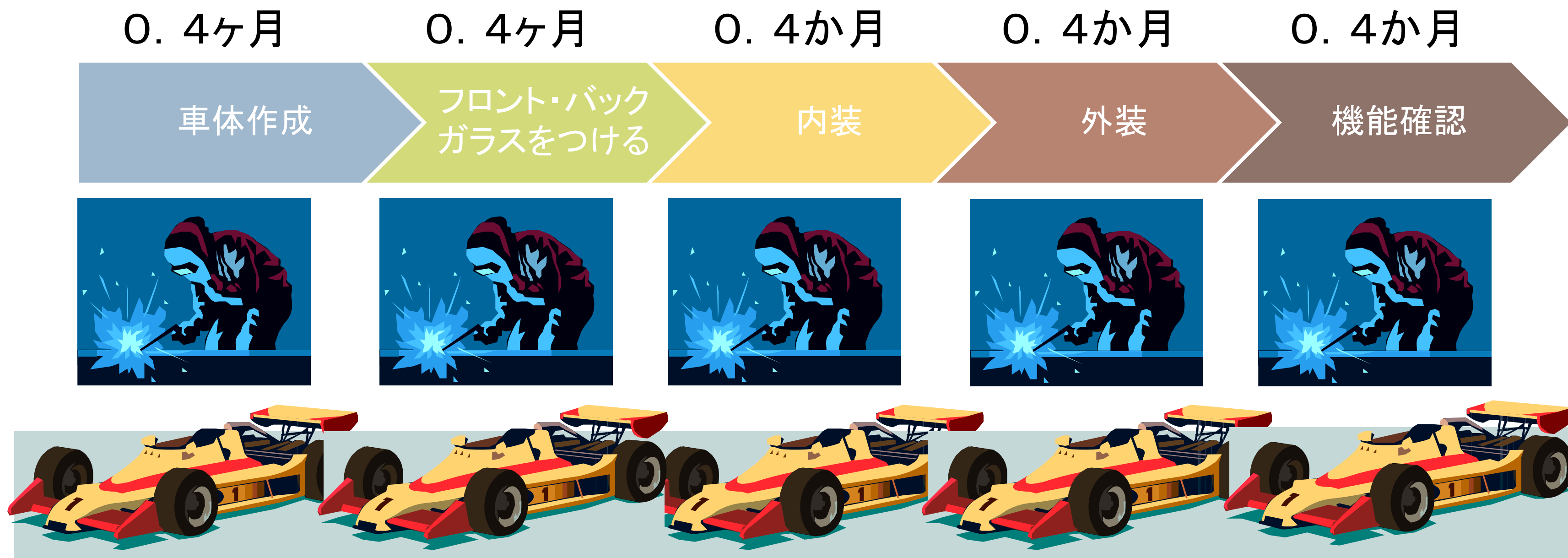
2台目

3台目

時間

流れ作業

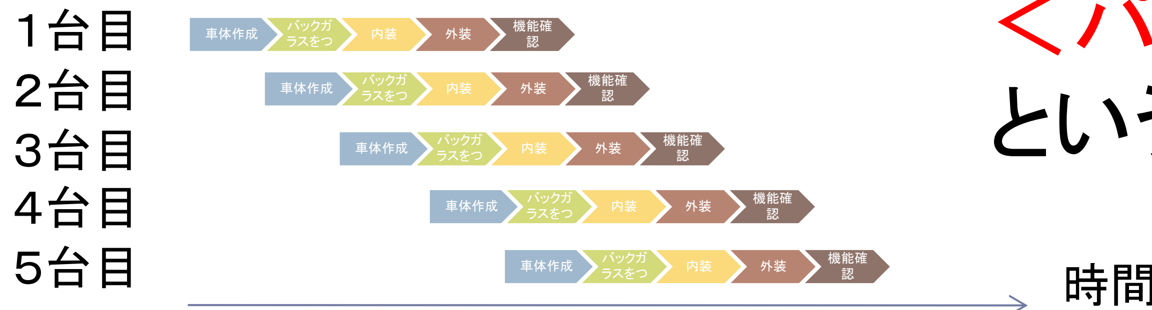
- 作業場所は、5ヶ所とれるとする
- 前の工程からくる車を待ち、担当工程が終わったら、次の工程に速やかに送られるとする
  - ベルトコンベア



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

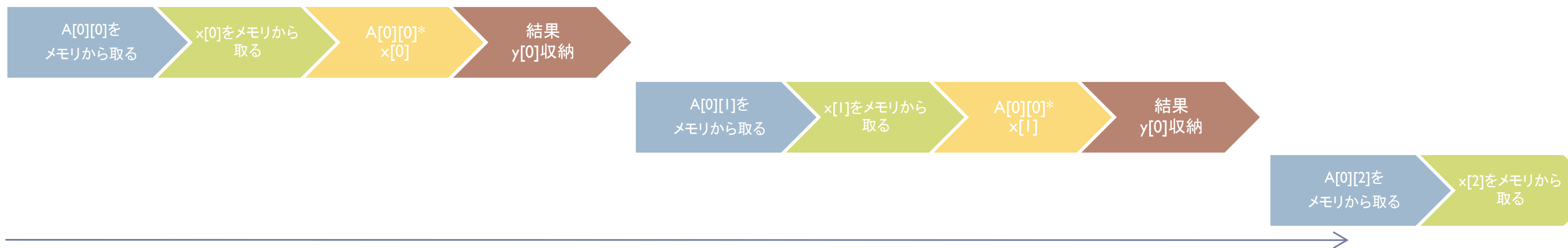
- 例：演算器の工程（注：実際の演算器の計算工程は異なる）

- 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] ;
}
```

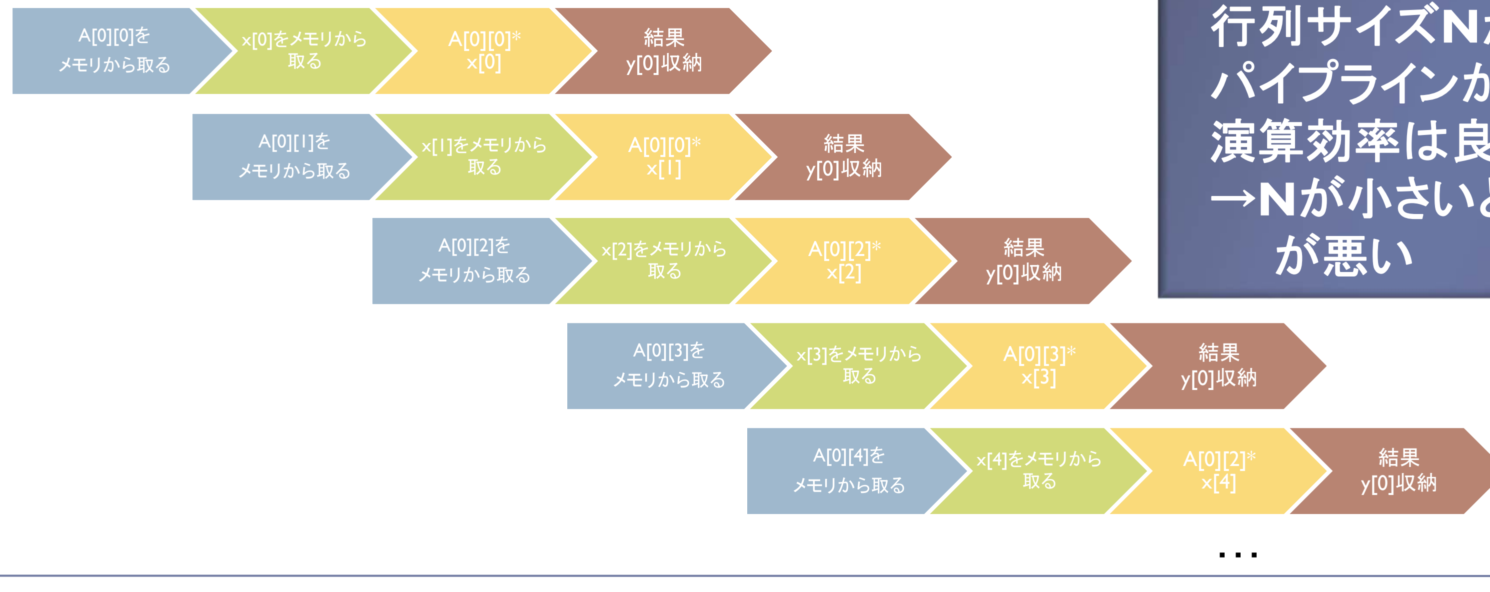
演算器が稼働する工程

- 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

iTC 東京大学情報基盤センター INFORMATION TECHNOLOGY CENTER, THE UNIVERSITY OF TOKYO

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
  - 浮動小数点積和演算ユニット（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

東京大学情報基盤センター INFORMATION TECHNOLOGY CENTER, THE UNIVERSITY OF TOKYO

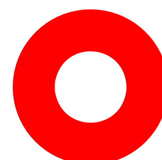
ループ内連続アクセス

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 ];
}
```

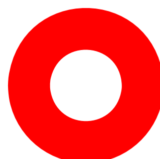

OK

```
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

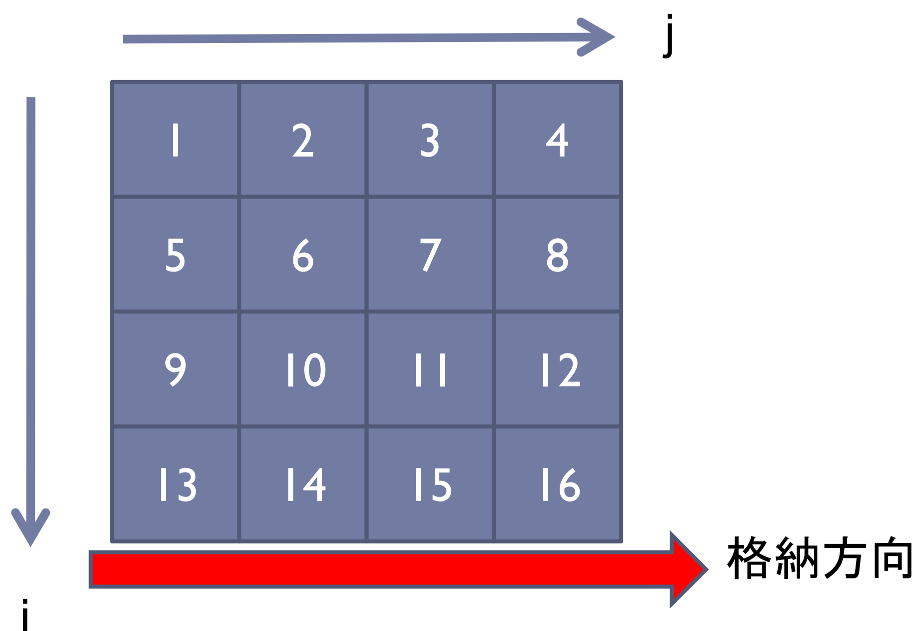
```
for (i=0; i<n; i++) {
  for (j=0; j<n; j++) {
      a[ i ][ j ] = b[ j ] * c[ j ];
  }}
```

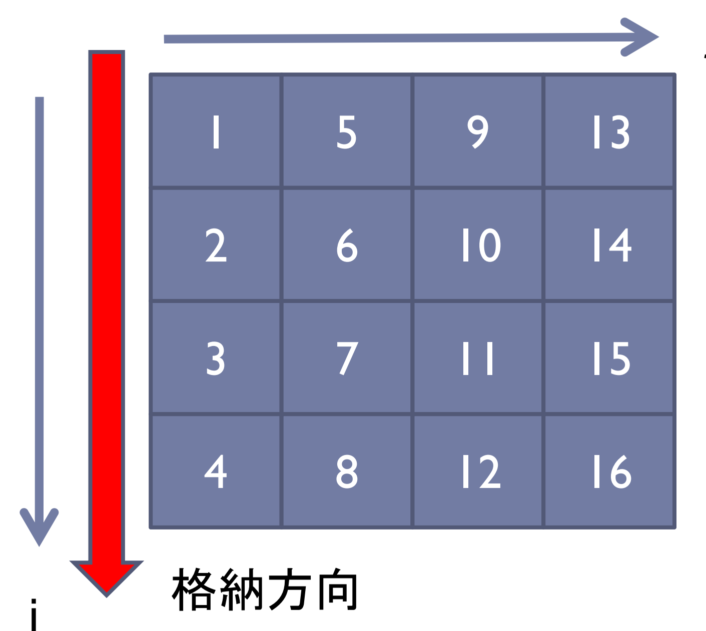

OK

```
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)

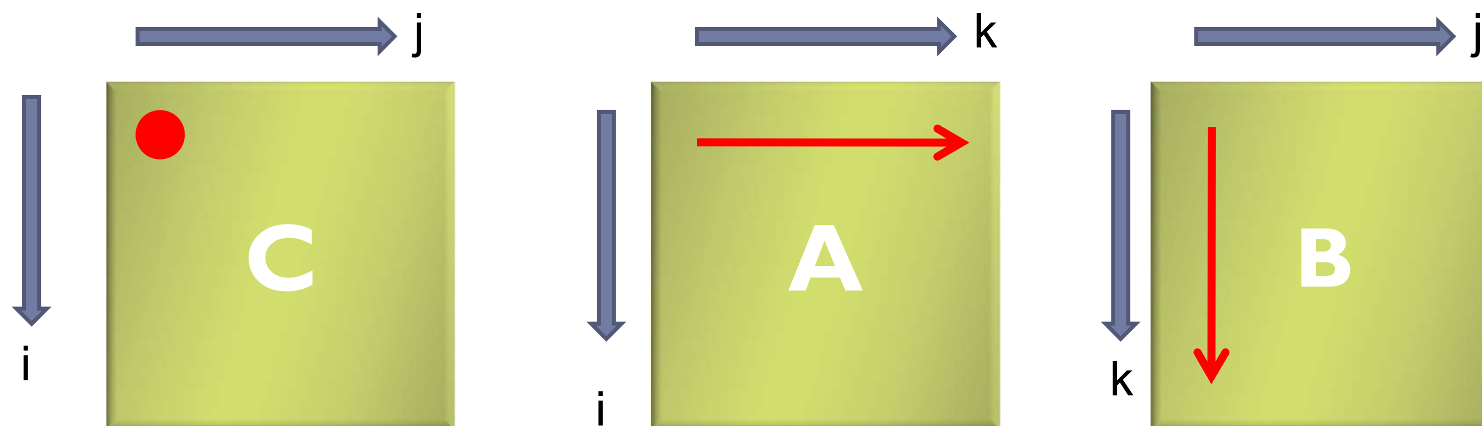




行列積コード例（C言語）

- 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

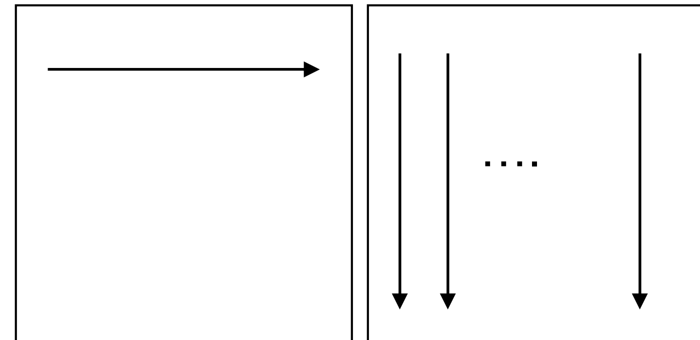
行列の積

- 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;
   }
}
```



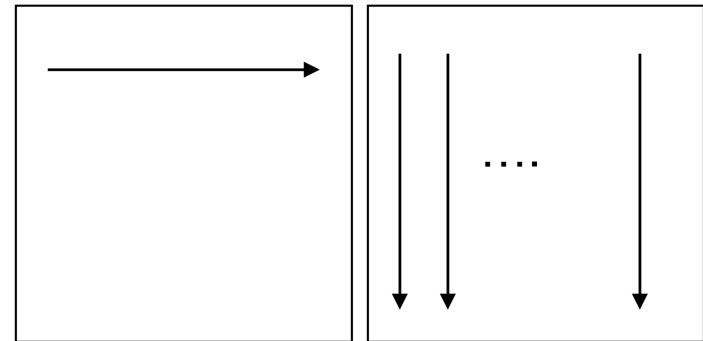
●行方向と列方向のアクセスあり
→行方向・列方向格納言語の
両方で性能低下要因
解決法:
A, Bどちらか一方を転置しておく
(ただし、データ構造の変更ができる場合)

※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
```



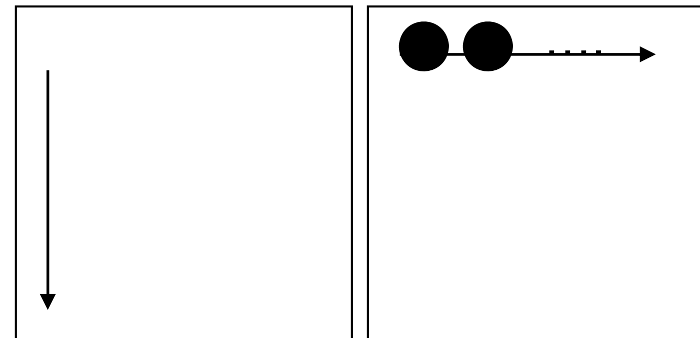
●行方向と列方向のアクセスあり
→行方向・列方向格納言語の
両方で性能低下要因
解決法:
A, Bどちらか一方を転置しておく
(ただし、データ構造の変更ができる場合)

※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;
     }
  }
}
```



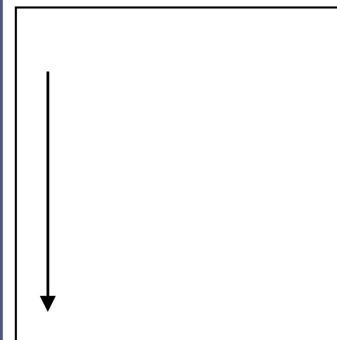
●kjiループでは
列方向アクセスがメイン
→列方向格納言語向き
（Fortran言語）

行列の積

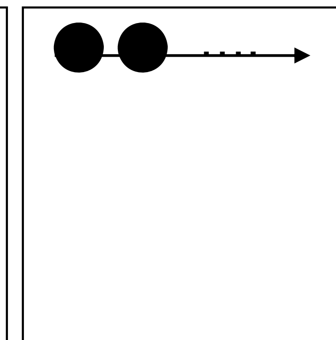
- 外積形式 (outer-product form）
  - kij, kjiループによる実現（Fortran言語）

```
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
```

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）

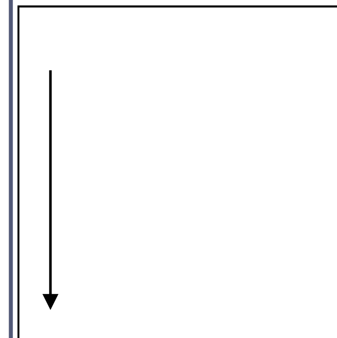
iTC 東京大学情報基盤センター INFORMATION TECHNOLOGY CENTER, THE UNIVERSITY OF TOKYO

行列の積

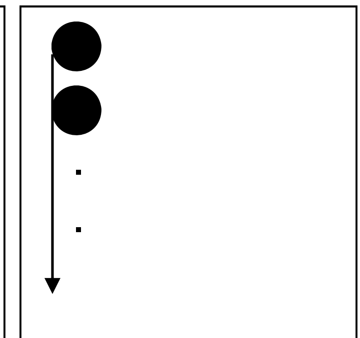
- 中間積形式 (middle-product form）
  - ikj, jkiループによる実現（C言語）

```
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

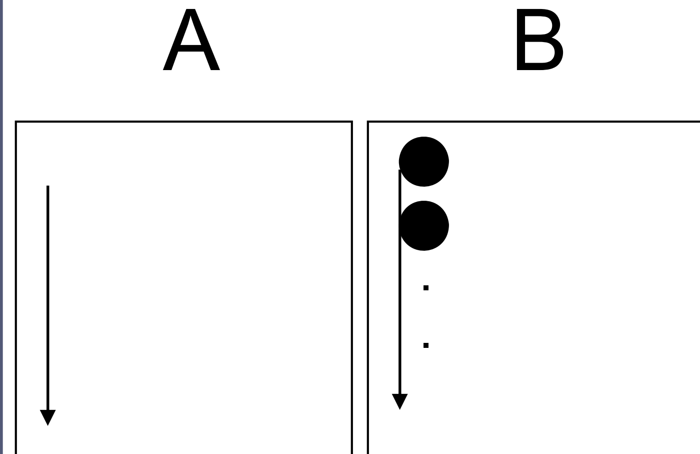


●jkiループでは
全て列方向アクセス
→列方向格納言語に
最も向いている
（Fortran言語）

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
```



●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

iTC 東京大学情報基盤センター
INFORMATION TECHNOLOGY CENTER, THE UNIVERSITY OF TOKYO

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

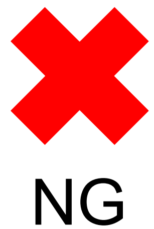
```
do i=1, n, 2
 do j=1, n, 2
  dc00 = C(i   ,j   ); dc01 = C(i   ,j+1)
  dc10 = C(i+1,j   ); dc11 = C(i+1,j+1)
  do k=1, n
    da0= A(i  ,k); da1= A(i+1, k)
    db0= B(k  ,j ); db1= B(k,   j+1)
    dc00 = dc00+da0 *db0; dc01 = dc01+da0 *db1;
    dc10 = dc10+da1 *db0; dc11 = dc11+da1 *db1;
  enddo
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

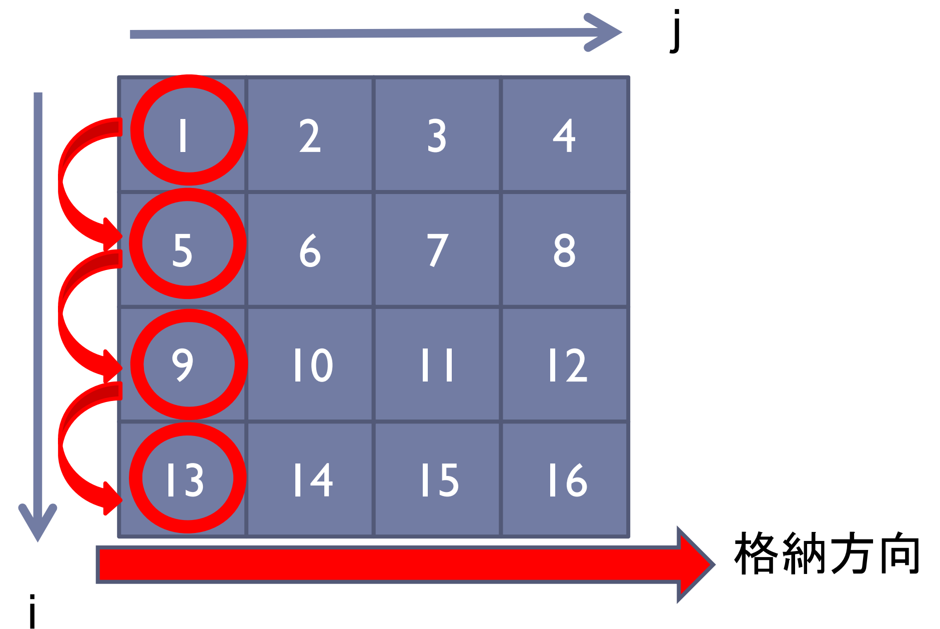


NG

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

- C言語の場合

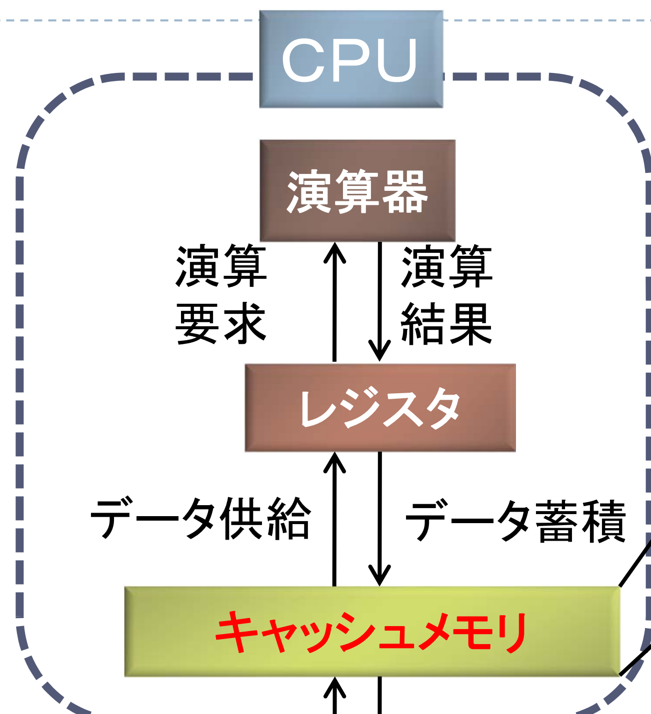
a[i][j]



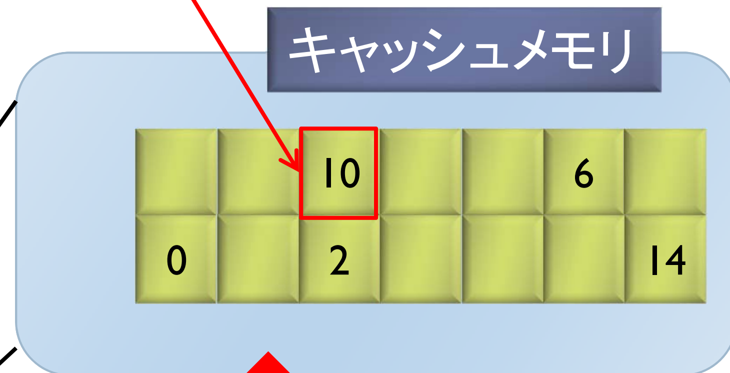
間隔4での不連続アクセス

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）

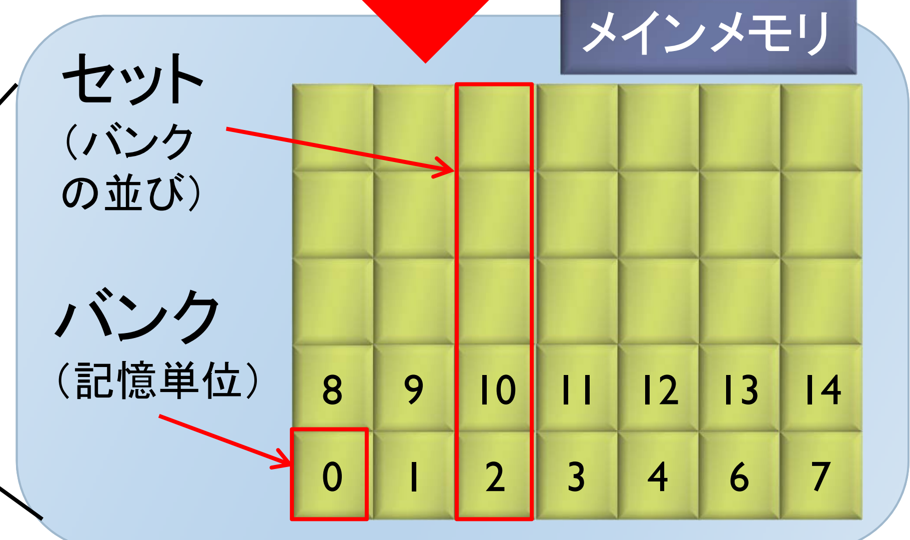


データ供給
データ蓄積
メインメモリ



写像関数

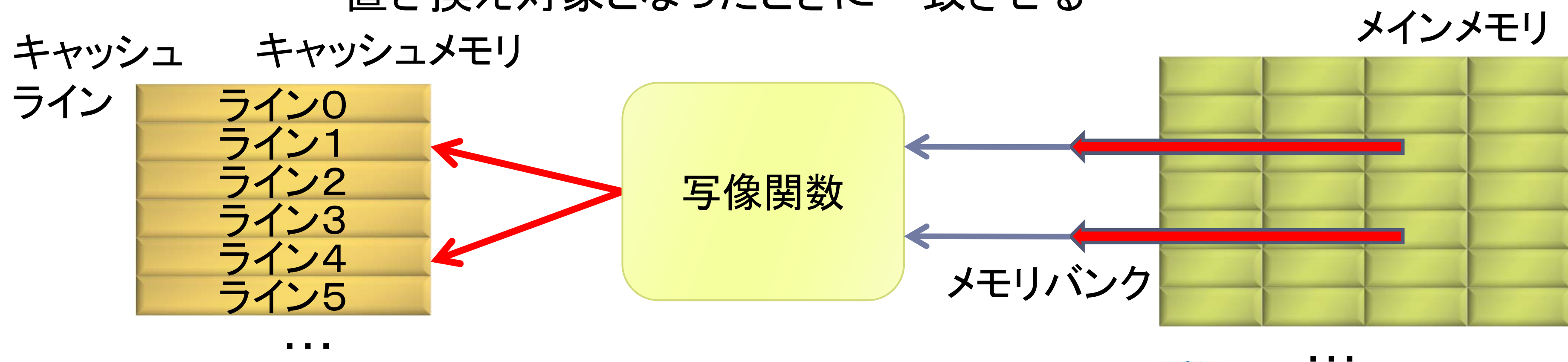
メモリバンクと
キャッシュラインの
対応



注）配列をアクセスすると、1要素分ではなく
バンク単位のデータ（例）32バイト（倍精度4変数分）
が同時にキャッシュに乗る（ラインサイズと呼ぶ）

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

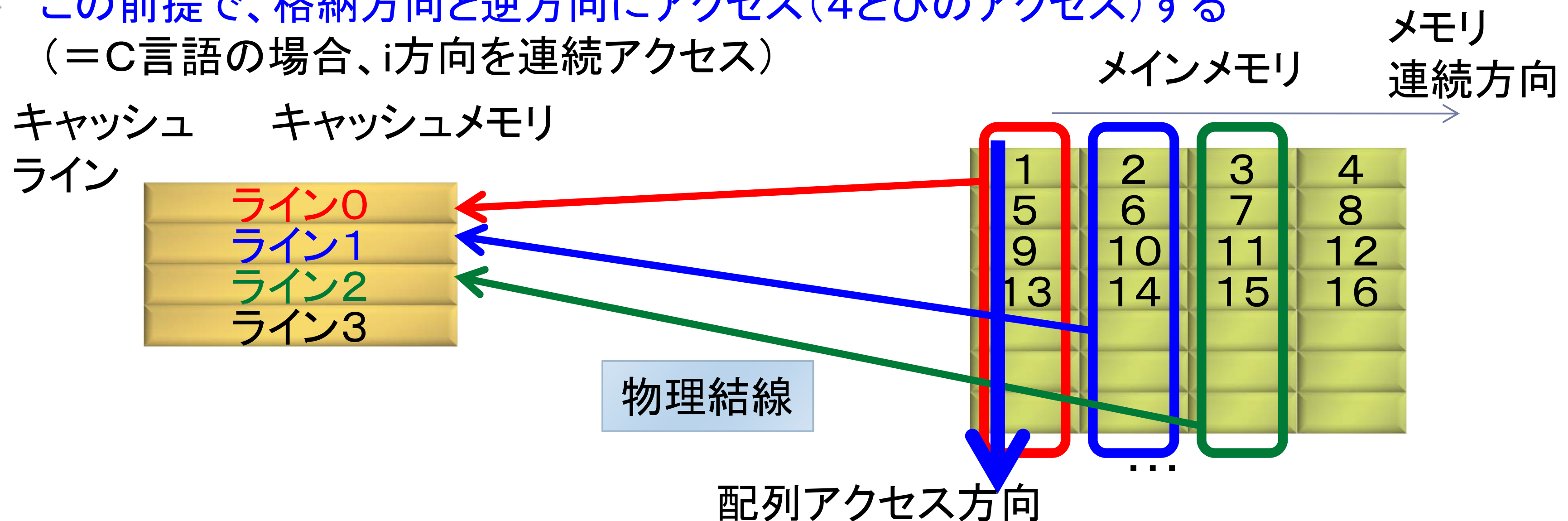


キャッシュライン衝突の例

- 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

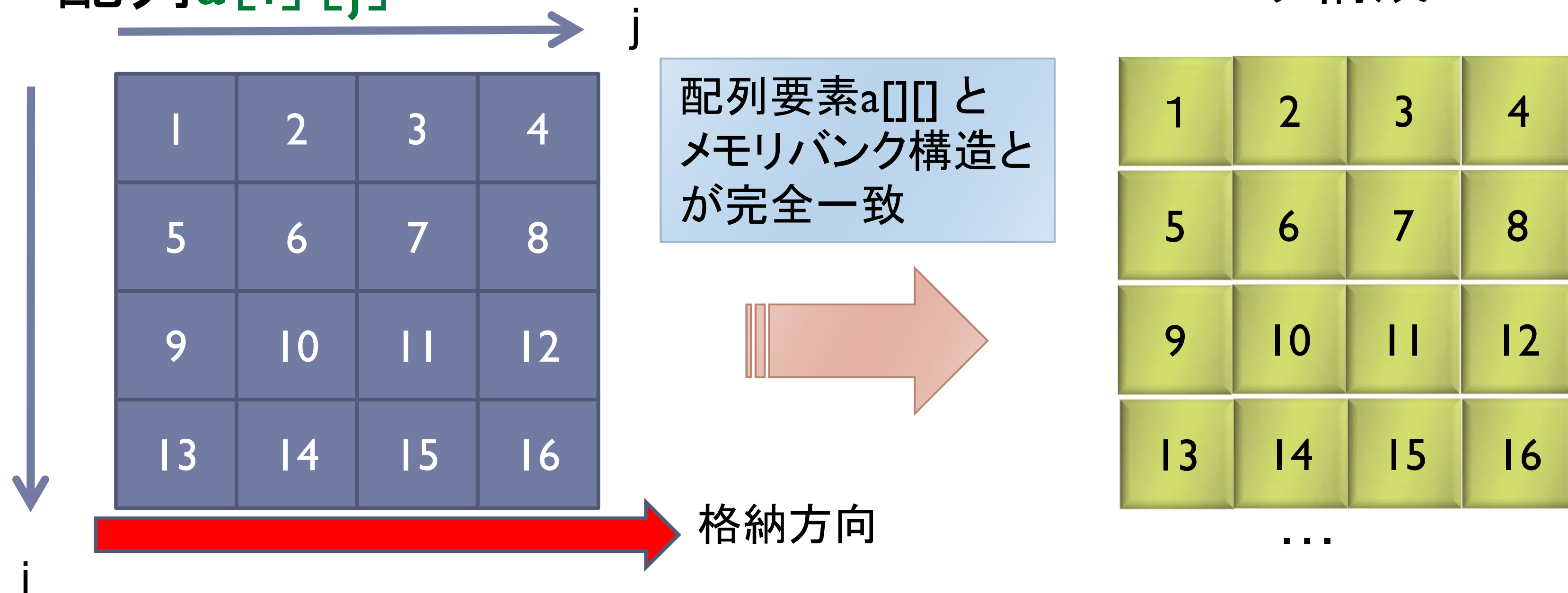
- この前提の、<実際の配列構成>と<メモリバンク>の関係

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]



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
キャッシュメモリ
9
ライン1
ライン2
ライン3
レジスタへ
メインメモリ
メモリ連続
配列アクセス方向

1	2	3	4
5	6	7	8
9	10	11	12
13	14	15	16

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

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[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。

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

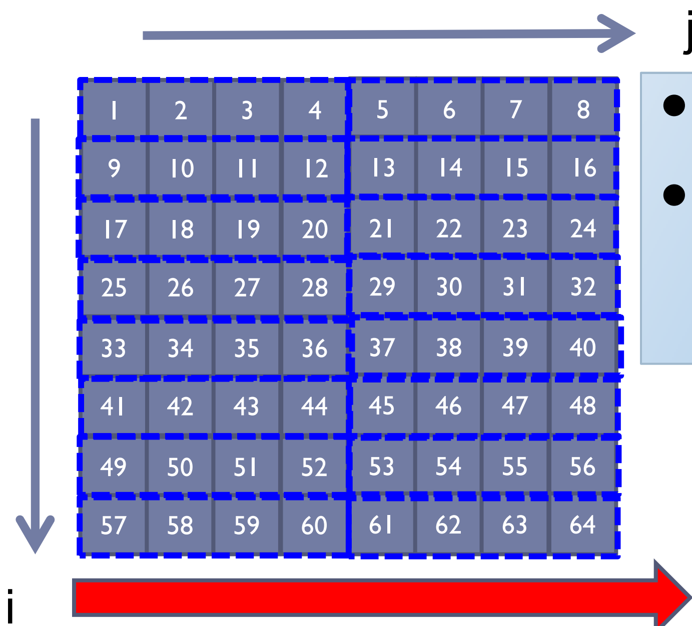
ブロック化によるキャッシュミスヒット削減例

- 行列ー行列積
- 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：4×8バイト(double)=32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

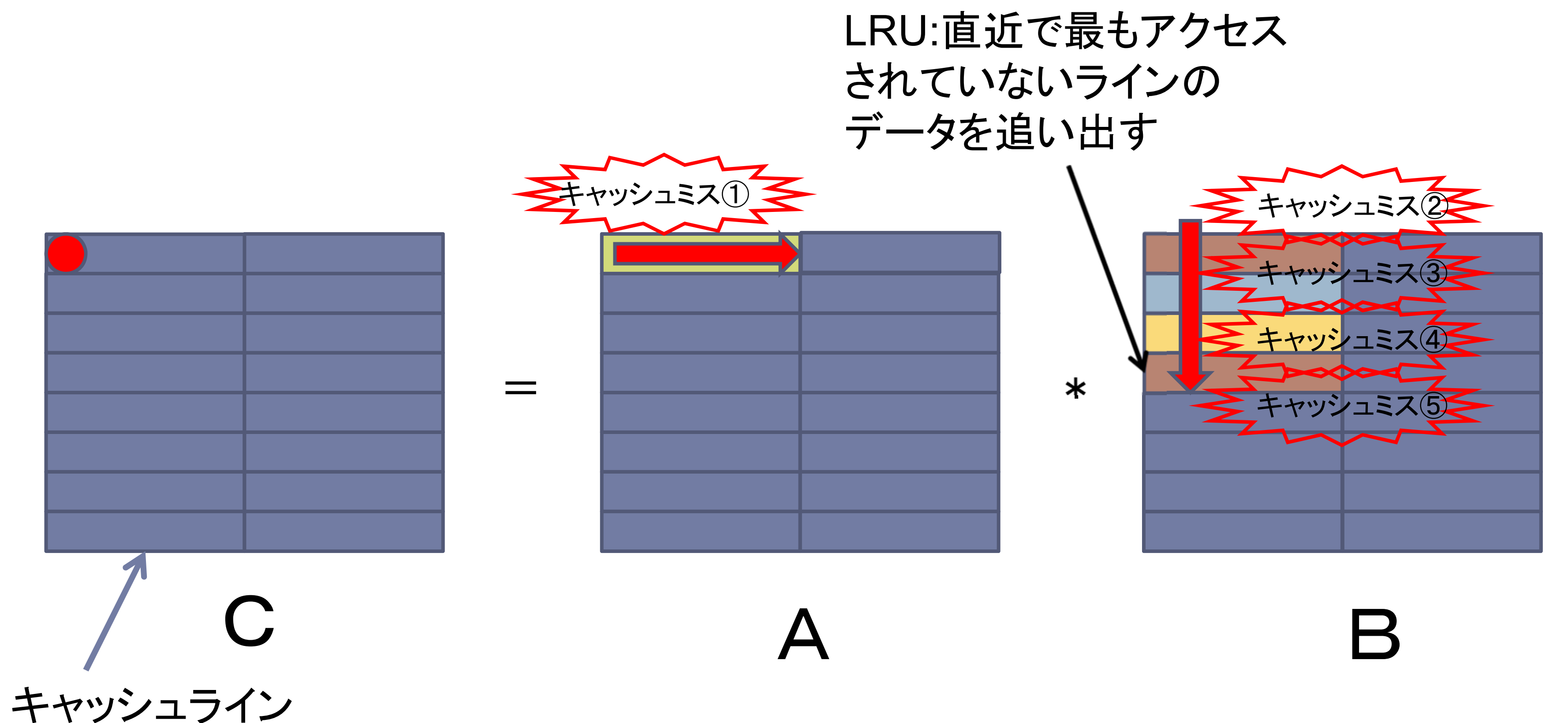
- C言語の場合
  配列A[i] [j]、B[i][j]、C[i][j]



- 1×4の配列要素が、キャッシュラインに乗る
- どのキャッシュラインに乗るかは、<配列アクセスパターン> と <置き換えアルゴリズム>依存で決まる

キャッシュラインの構成

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）

キャッシュミス11

キャッシュミス⑥

キャッシュミス⑦

キャッシュミス⑧

キャッシュミス⑨

キャッシュミス⑩

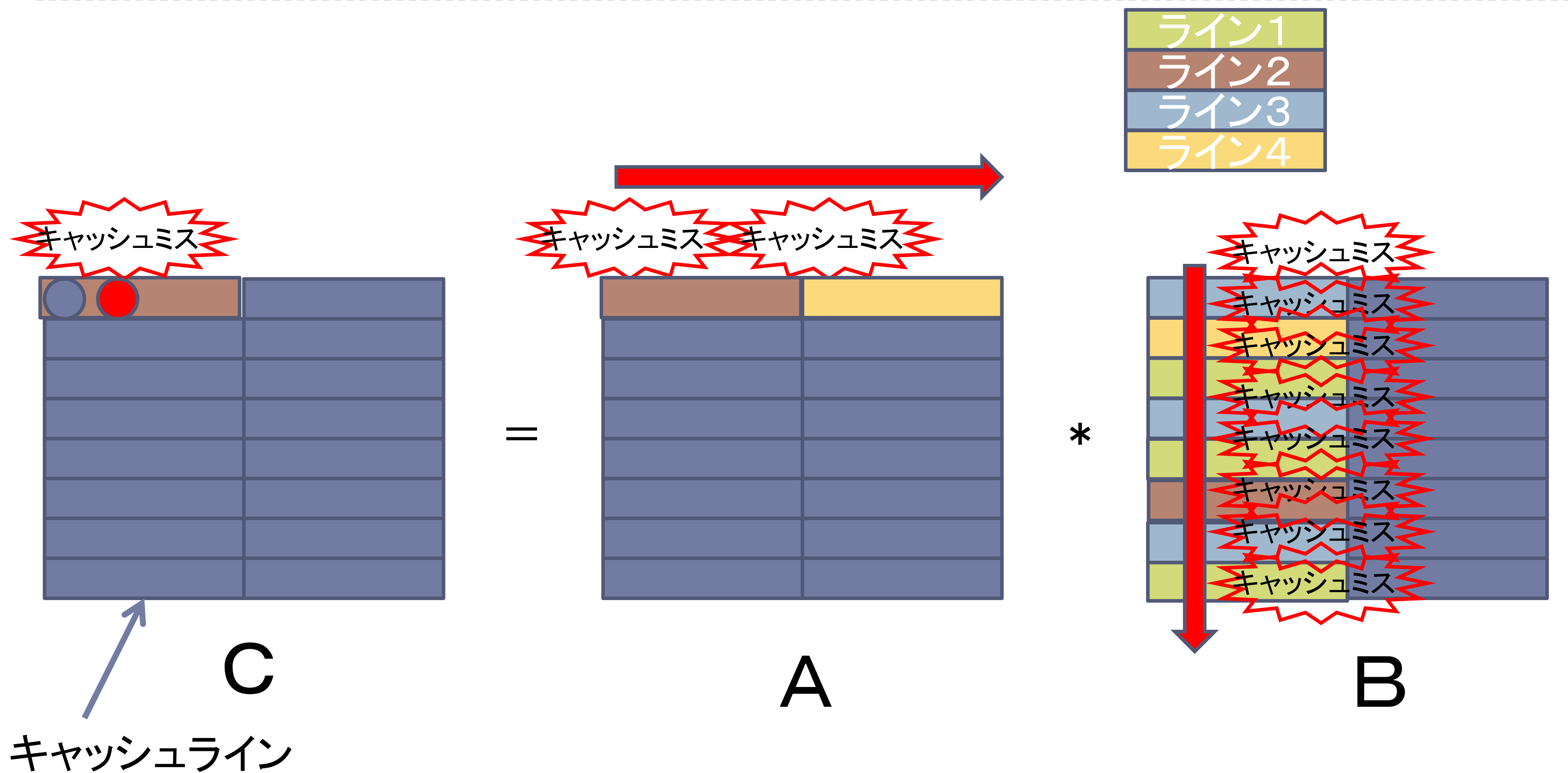
C = A * B

キャッシュライン

ライン1
ライン2
ライン3
ライン4

※キャッシュライン4つ、
置き換えアルゴリズム
LRUの場合

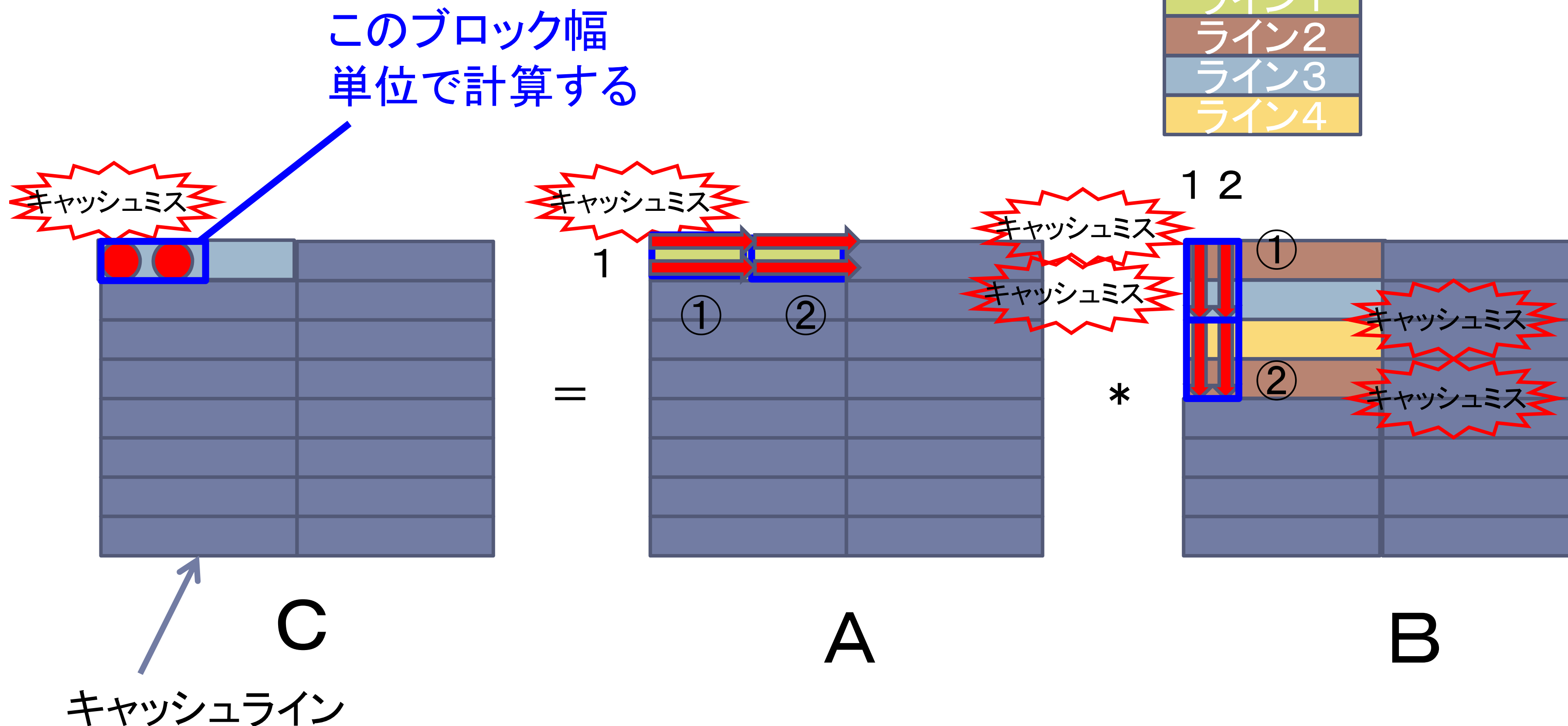
行列-行列積の場合（ブロック化しない）



※2要素計算するのに、
キャッシュミスヒット22回

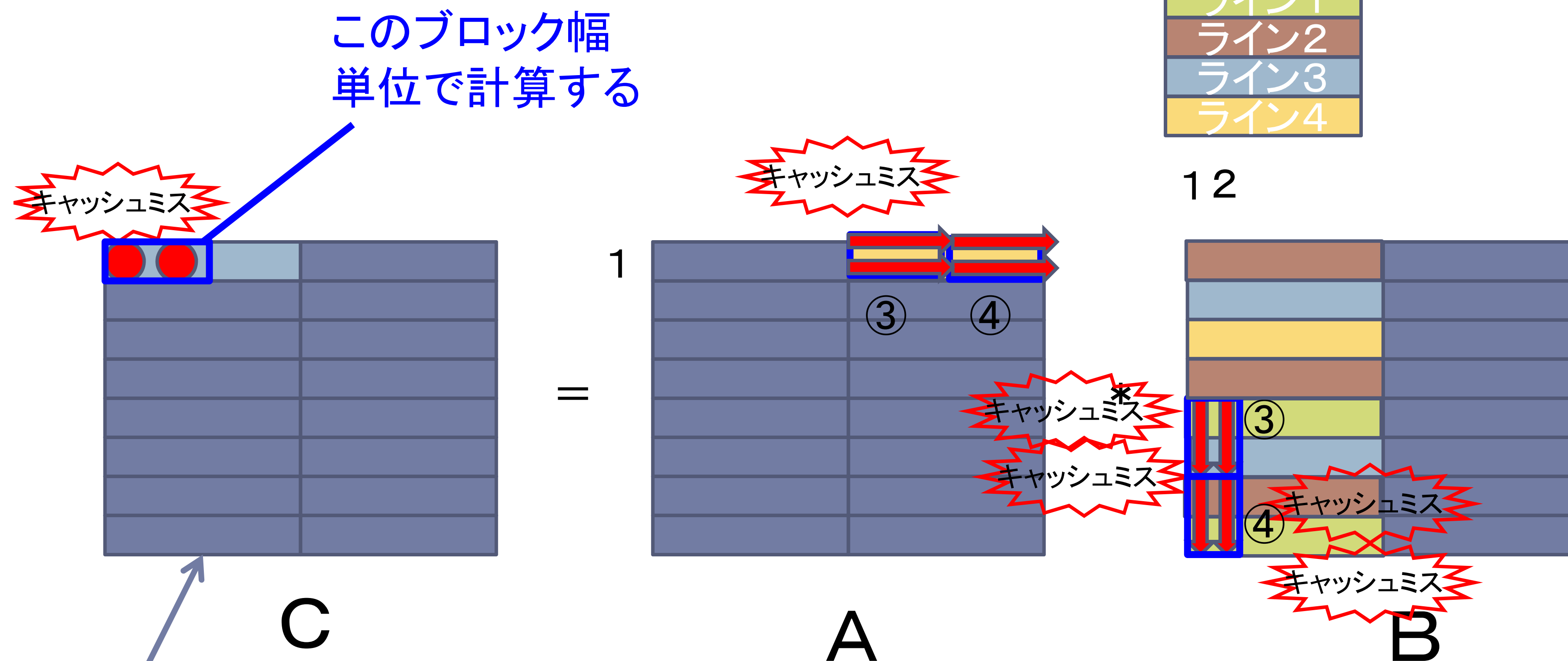
※キャッシュライン4つ、
置き換えアルゴリズム
LRUの場合

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）



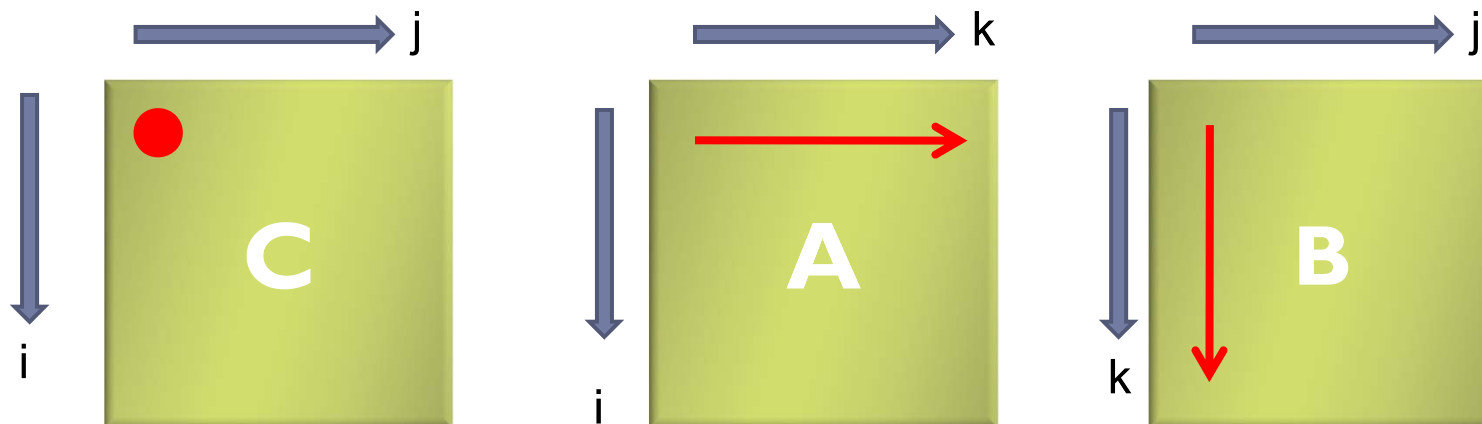
※2要素計算するのに、
キャッシュミスヒット１０回

※キャッシュライン4つ、
置き換えアルゴリズム
LRUの場合

行列積コード（C言語）：キャッシュブロック化なし

- 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];
```



iTC 東京大学情報基盤センター INFORMATION TECHNOLOGY CENTER, THE UNIVERSITY OF TOKYO

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;
```

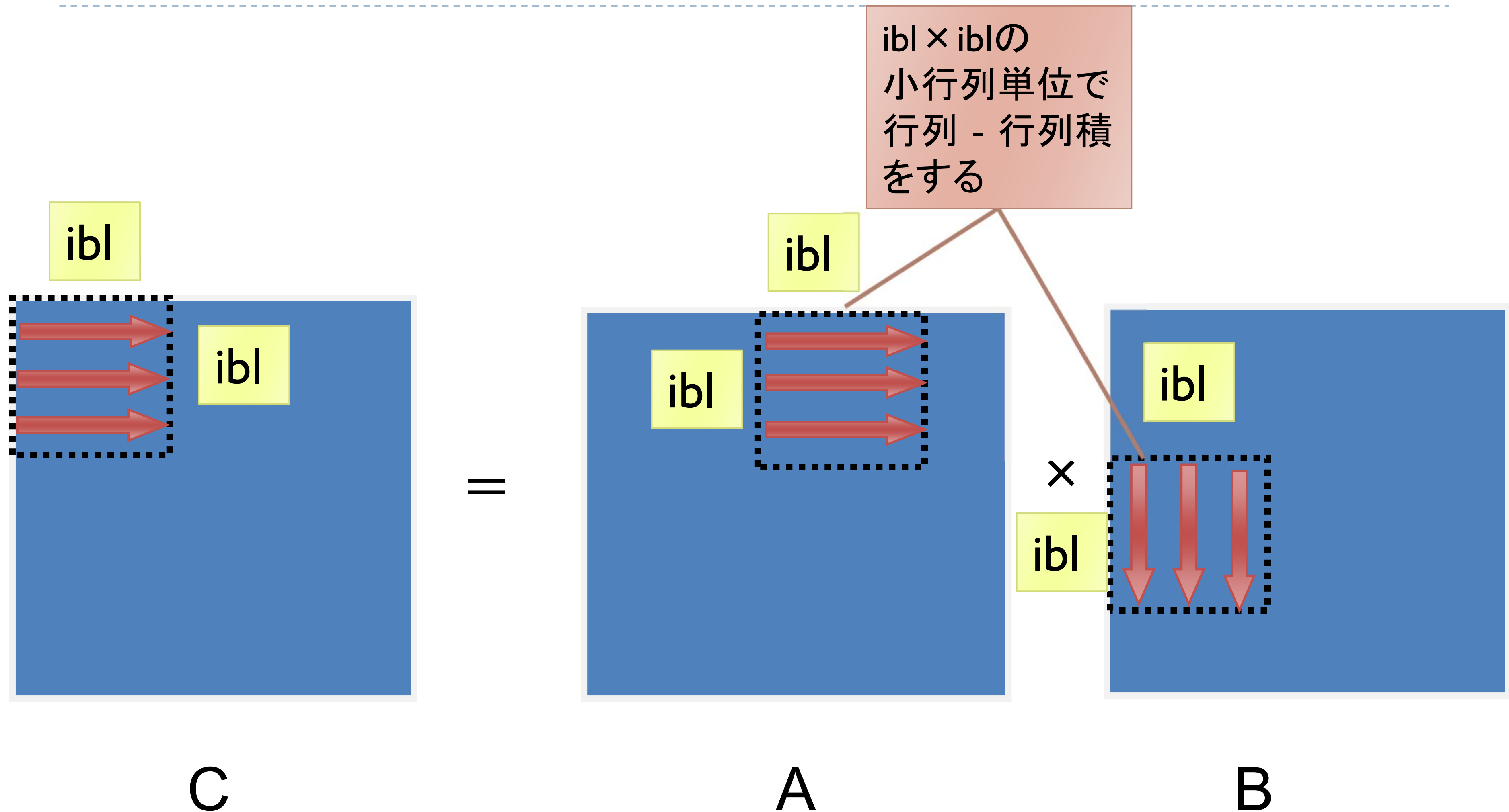
キャッシュブロック化時の データ・アクセスパターン

ibl×iblの
小行列単位で
行列 - 行列積
をする

ibl
ibl
ibl
ibl
ibl
ibl

C ＝ A × B

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ＩＦ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][i] = 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）

定義ー参照の距離が近い
→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）

定義ー参照の距離が遠い
→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、計算機ハードウエアの演算性能
  - 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

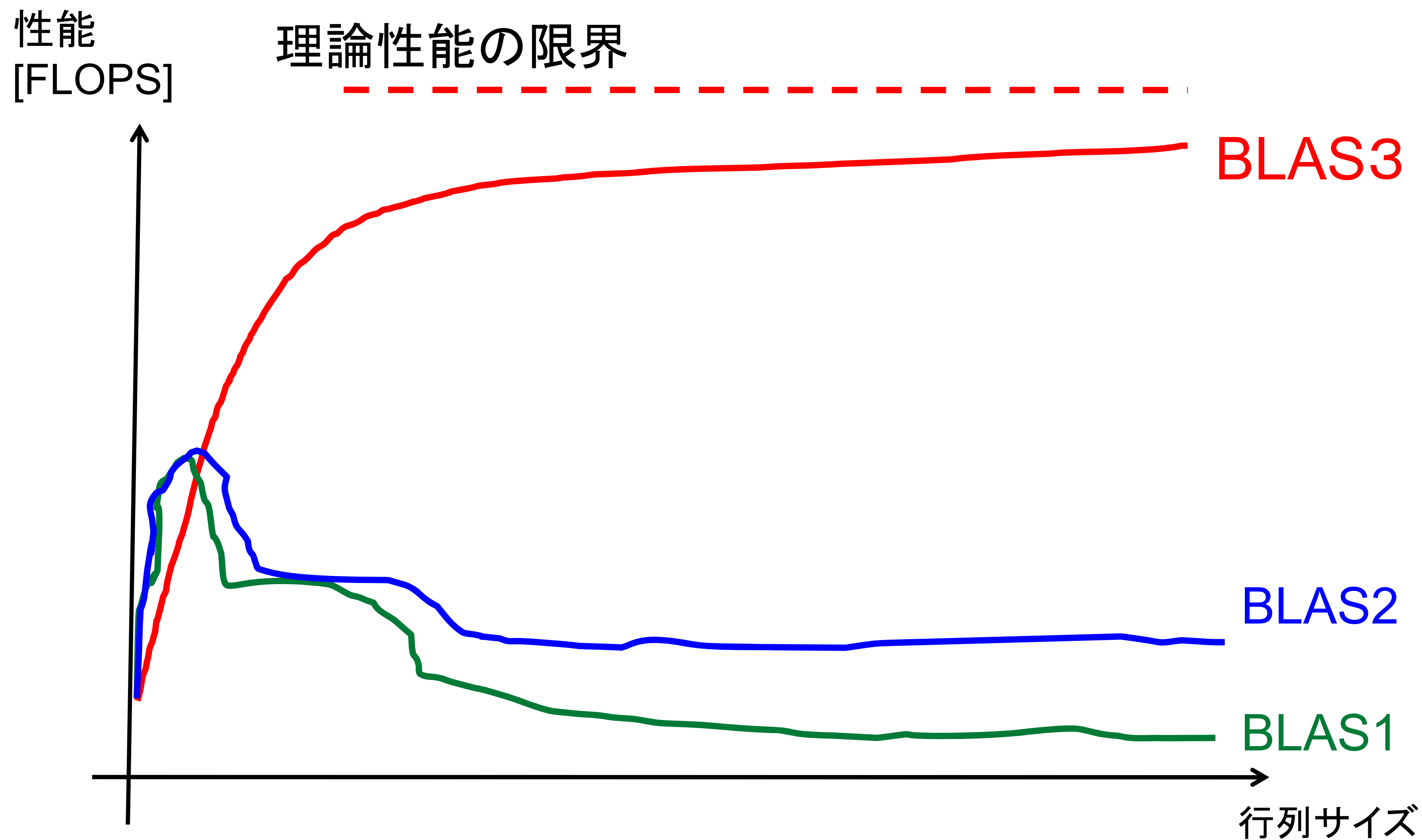
レベル２　BLAS

- 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、$T x = y$ （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、再利用の効果がない。
  - 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



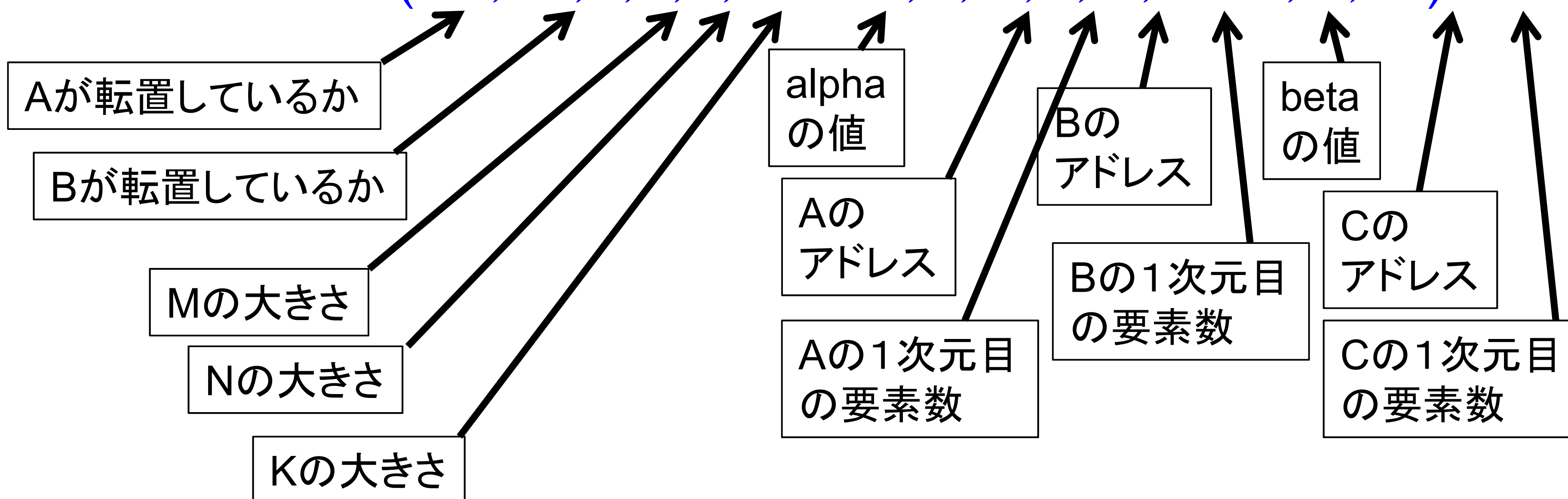
BLAS利用例

- 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:XYYYY
  - 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エア上の実装に特化
    - 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 >= 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×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　>= max(1,N)。
- IPIVOT (出力)　ー 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）

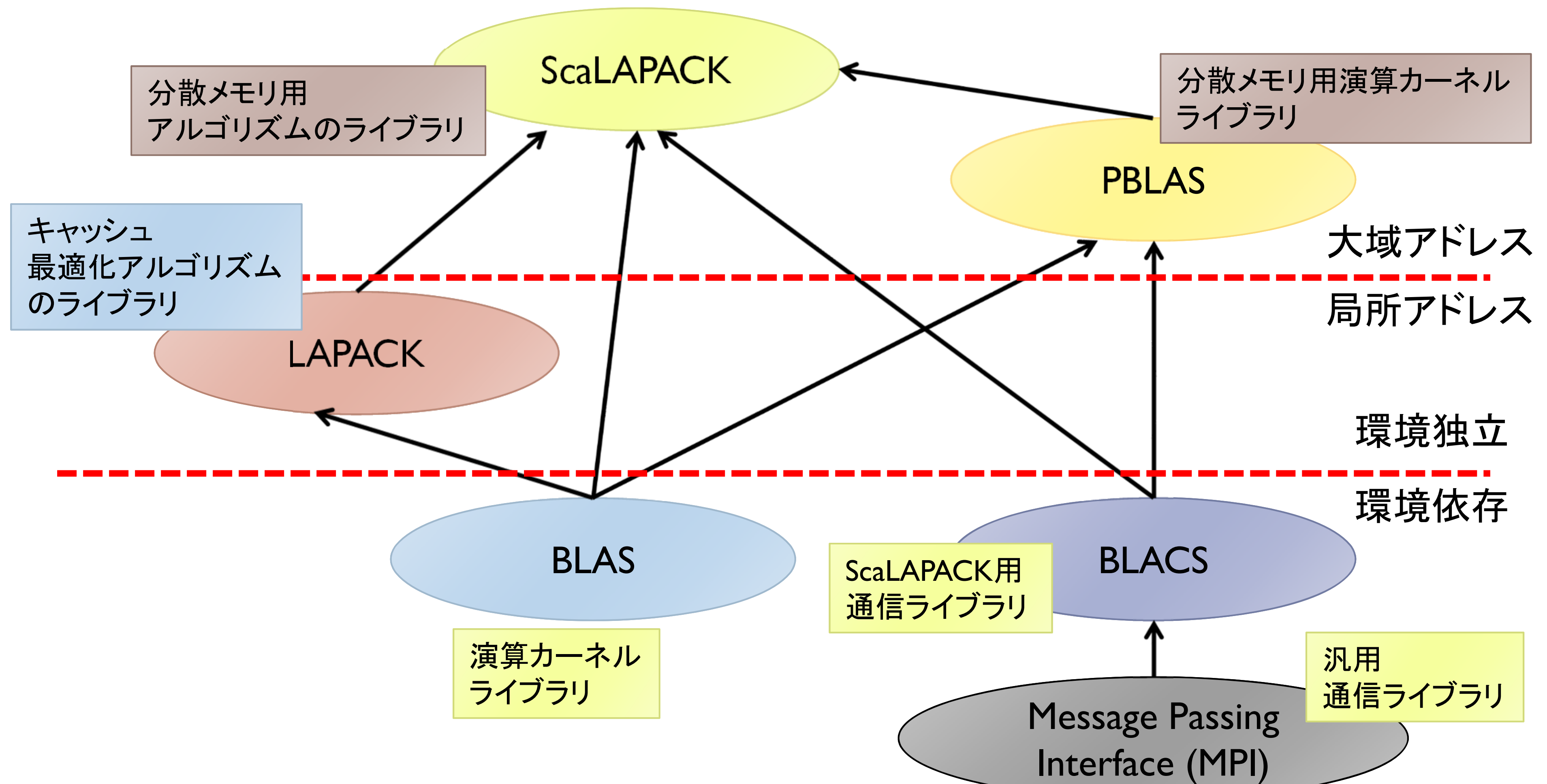
- B (入力／出力) – DOUBLE PRECISION, DIMENSION(:,:)
  - 入力時は、右辺ベクトルの　N × NRHS　行列Bを入れる。
  - 出力時は、もし、INFO = 0　なら、N × NRHS行列である解行列Xが戻る。
- LDB (入力)　－INTEGER
  - 配列Bの最初の次元の大きさ。LDB >= max(1,N)。
- INFO (出力) －INTEGER
  - = 0: 正常終了
  - < 0: もし INFO = -i　なら i-th 行の引数の値がおかしい。
  - > 0: もし INFO = i　なら U(i,i) が厳密に0である。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* ×*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* ×*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 >= 0。
- NRHS (大域入力) – INTEGER
  - 右辺ベクトルの数。行列Bの次元数。NRHS >= 0。
- A (局所入力／出力) – DOUBLE PRECISION, DIMENSION(:,:)
  - 入力時は、N×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(LOCr(M_A)+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×NRHSの行列Bの分散されたものを(LLD_B, LOCc(JB+NRHS-1))に入れる。
  - 出力時は、もし、INFO = 0　なら、N×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
  - *sub(B)*の最初の行のインデックス
- JB(大域入力) –INTEGER
  - *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) ー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、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、INFO = -iとなる。
  - > 0: もし INFO = Kのとき U(IA+K-1, 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
    - Fortranに対して転置行列になるので、BLASの引数で転置を指定
  - 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、利用スレッド数が多くなると、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、200スレッド並列だと、スレッドあたりN=570まで小さくなる

その他のライブラリ（主に行列演算）

種類	問題	ライブラリ名	概要
密行列	BLAS	MAGMA	GPU、マルチコア、ヘテロジニアス環境対応
疎行列	連立一次方程式	MUMPS	直接解法
		SuperLU	直接解法
		PETSc	反復解法、各種機能
		Hypre	反復解法
	連立一次方程式、固有値ソルバ	Lis	反復解法（国産ライブラリ）
		Xabclib	反復解法、自動チューニング（AT）機能（国産ライブラリ）

その他のライブラリ（信号処理等）

種類	問題	ライブラリ名	概要
信号処理	FFT	FFTW	離散フーリエ変換、AT機能
		FFTE	離散フーリエ変換（国産ライブラリ）
		Spiral	離散フーリエ変換、AT機能
グラフ処理	グラフ分割	METIS、ParMETIS	グラフ分割
		SCOTCH、PT-SCOTCH	グラフ分割

その他のライブラリ（フレームワーク）

種類	問題	ライブラリ名	概要
プログラミング環境	マルチフィジックス、など	Trilinos	プログラミングフレームワークと数値計算ライブラリ
	ステンシル演算	Phisis	ステンシル演算用プログラミングフレームワーク（国産ライブラリ）
数値ミドルウェア	FDM、FEM、DEM、BEM、FVM	ppOpen-HPC	5種の離散化手法に基づくシミュレーションソフトウェア、数値ライブラリ、AT機能（国産ライブラリ）

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、もしくはSPARC64 XIfx、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。