

---

# 高性能プログラミング技法の基礎 (1)

東京大学情報基盤センター 准教授 片桐孝洋

2015年10月27日(火) 10:25-12:10

▶ | スパコンプログラミング(1)、(I)



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

# 講義日程（工学部共通科目）

レポートおよびコンテスト課題  
(締切:  
2016年2月11日(木)24時 厳守

~~▶ 10月6日: ガイダンス~~

~~▶ 10月13日~~

- ~~● 並列数値処理の基本演算(座学)~~

~~▶ 10月20日: スパコン利用開始~~

- ~~● ログイン作業、テストプログラム実行~~

3. 10月27日

- 高性能演算技法1  
(ループアンローリング)

4. 11月10日

- 高性能演算技法2  
(キャッシュブロック化)

5. 11月24日

- 行列-ベクトル積の並列化

6. 12月1日(8:30-10:15)

- べき乗法の並列化

6. 12月1日(10:25-12:10)

- 行列-行列積の並列化(1)

7. 12月8日

- 行列-行列積の並列化(2)

8. 12月15日

- LU分解法(1)
- コンテスト課題発表

9. 12月22日

- LU分解法(2)

10. 2016年1月5日

- LU分解法(3)

12. 1月12日

- ソフトウェア自動チューニング
- 非同期通信

# 講義の流れ

---

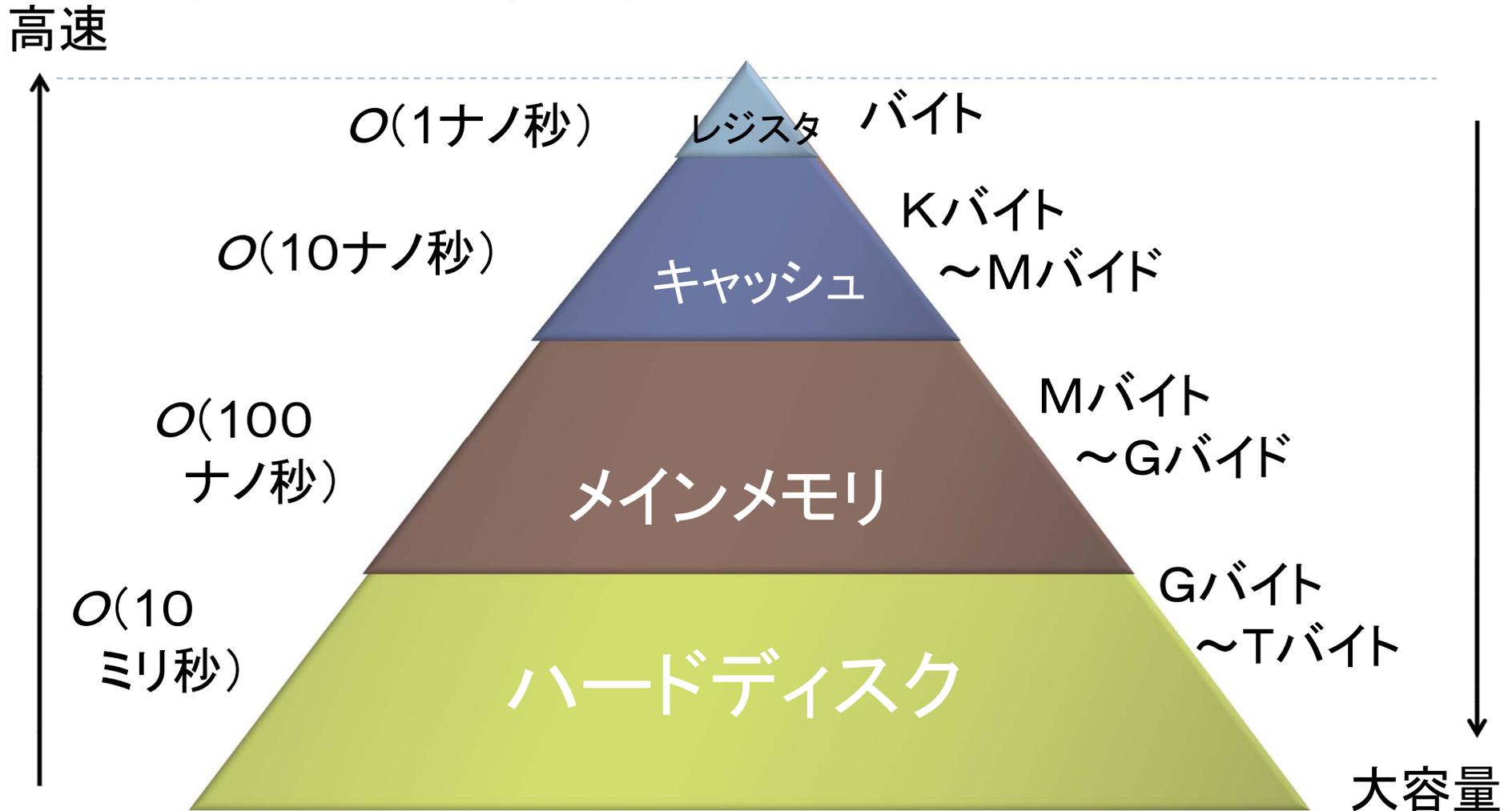
1. 階層キャッシュメモリ
2. 演算パイプライン
3. ループアンローリング
4. 配列連続アクセス
5. 演習課題
6. レポート課題

---

# 階層キャッシュメモリ

超高速メモリは小容量

# 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...

---

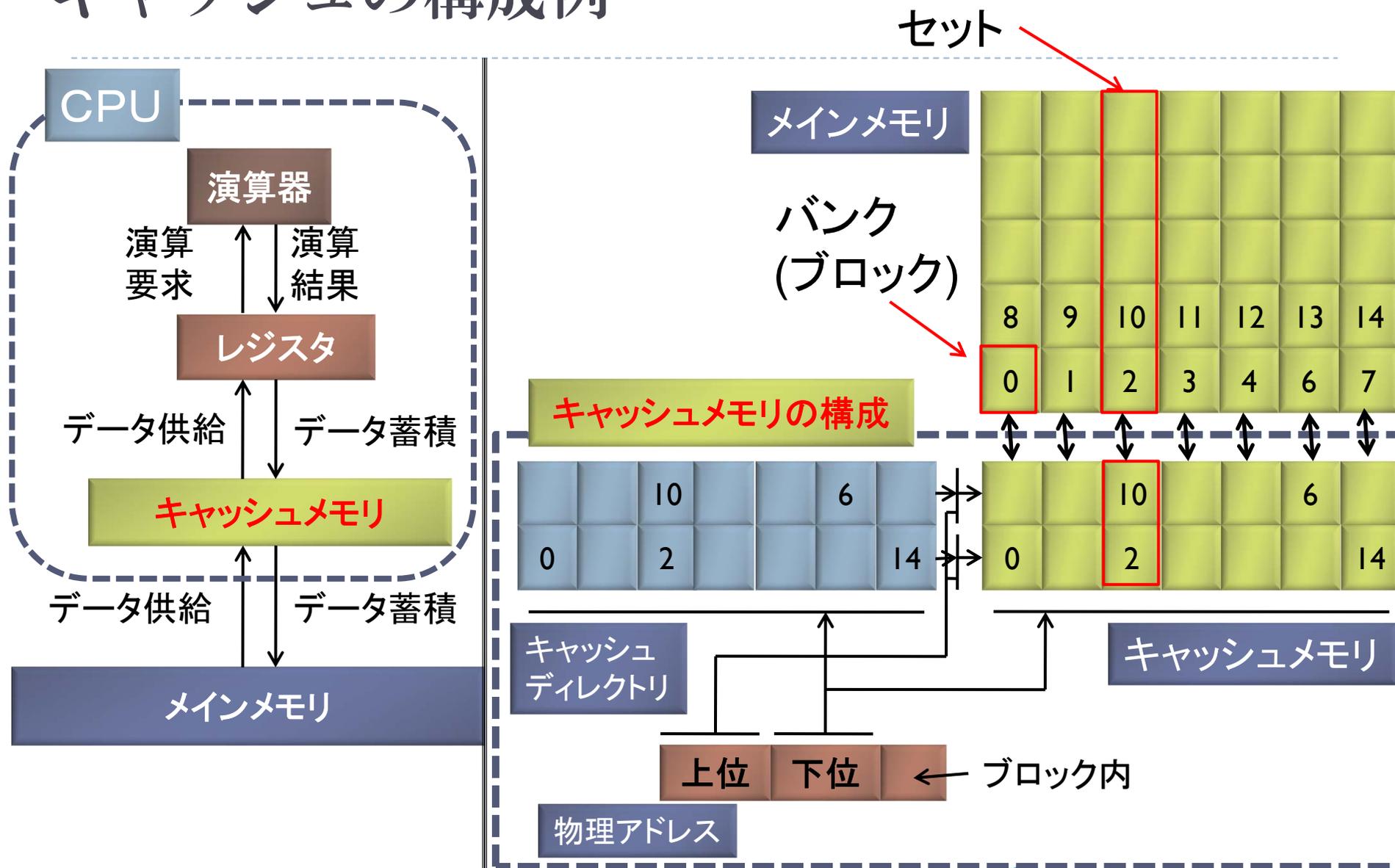
レジスタ

キャッシュ

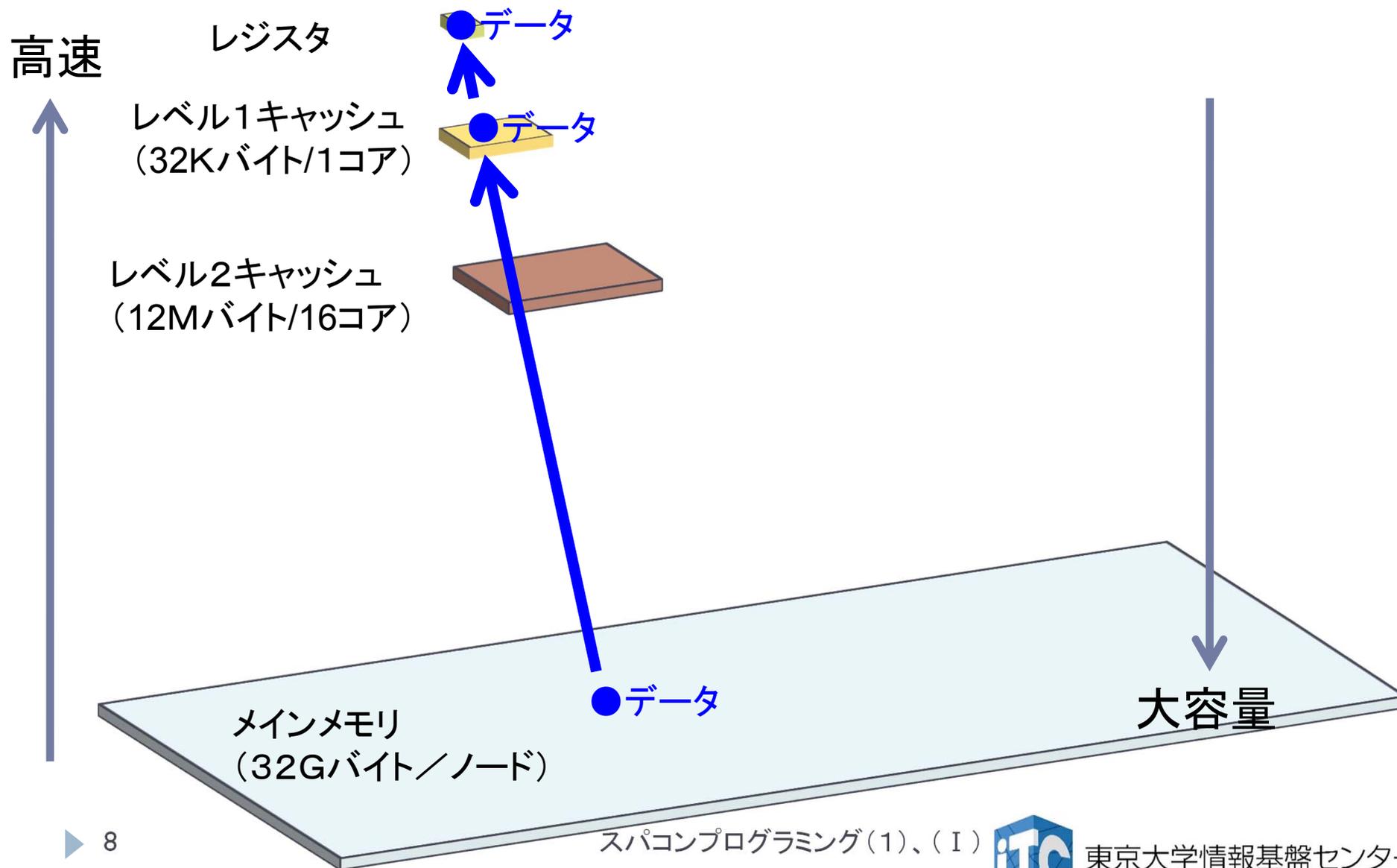
メインメモリ

- 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

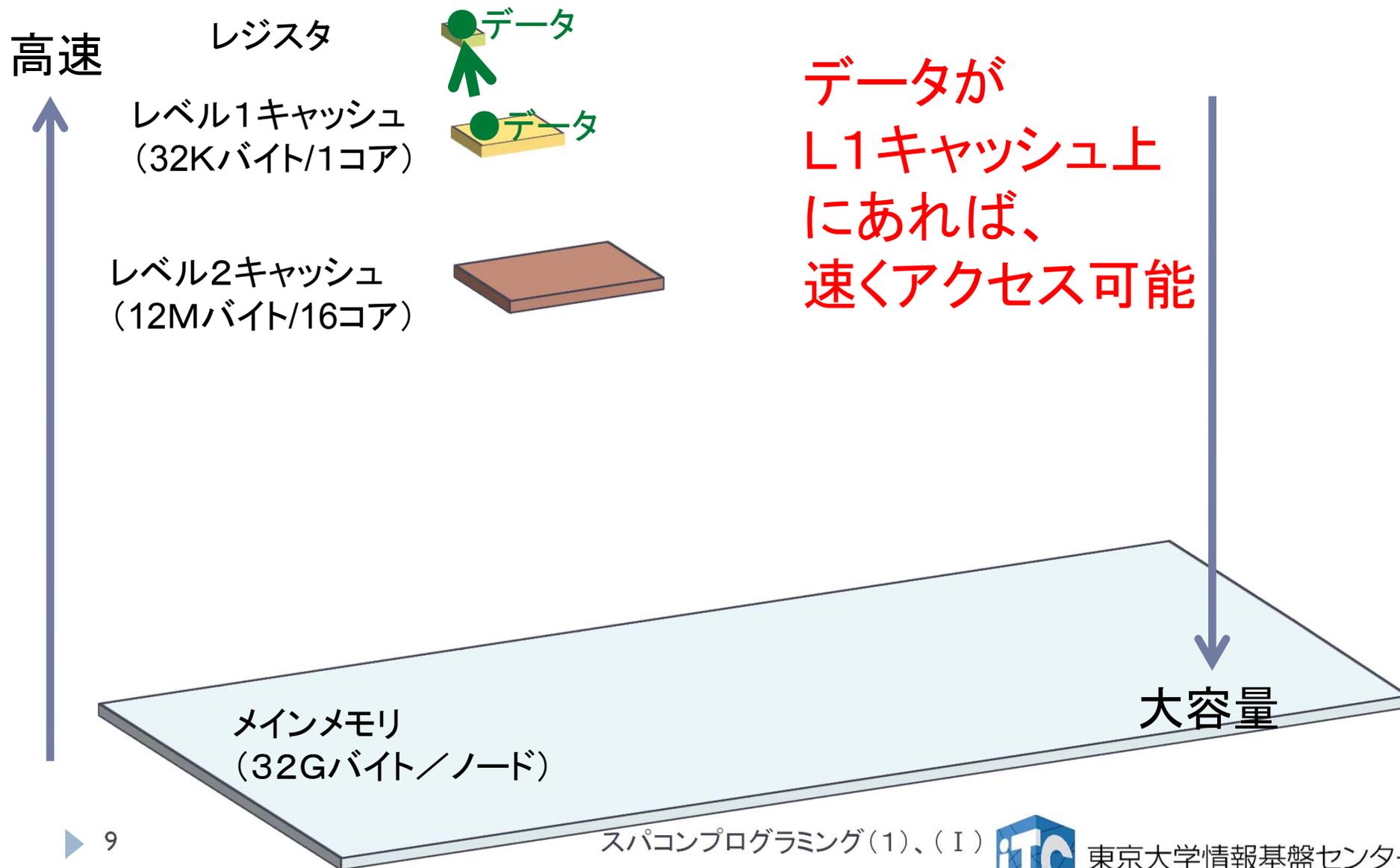
# キャッシュの構成例



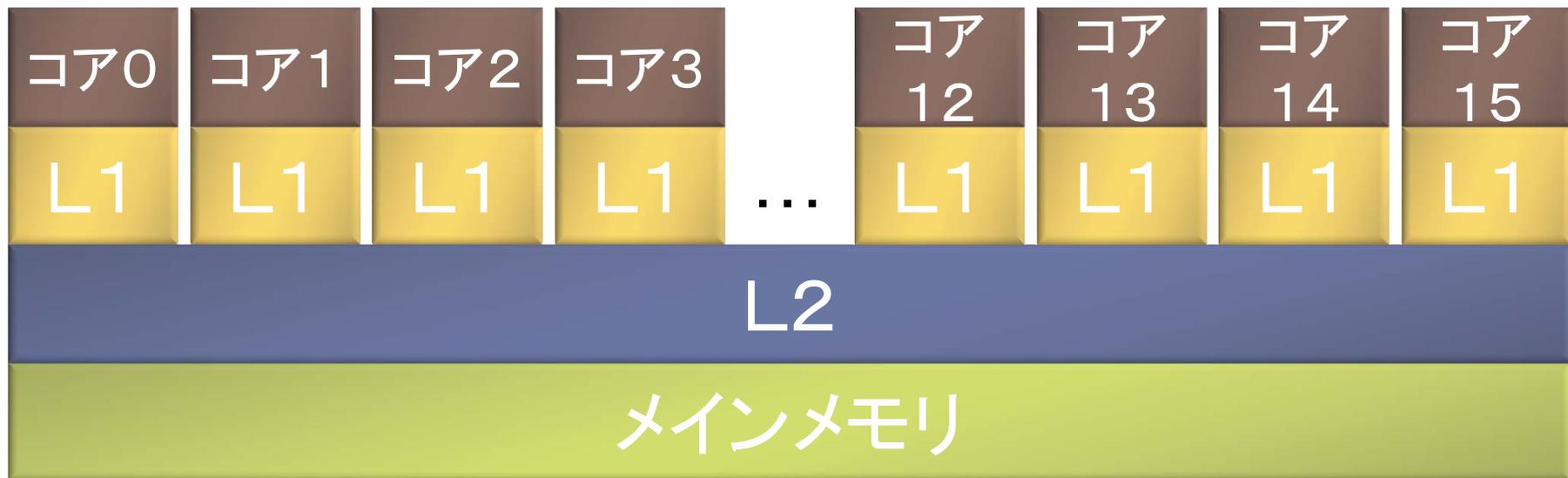
# FX10のメモリ構成例



# FX10のメモリ構成例

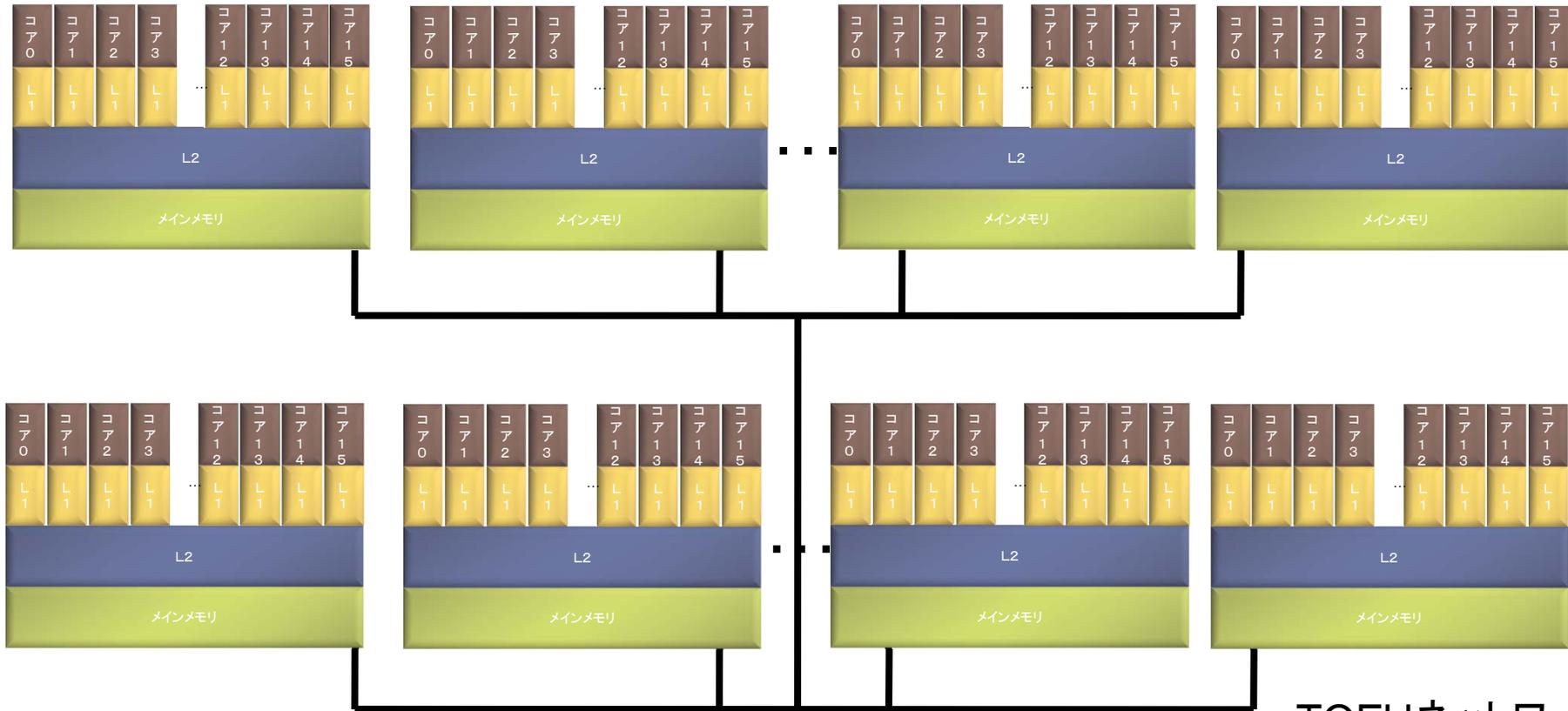


# FX10のノードのメモリ構成例



※階層メモリ構成となっている

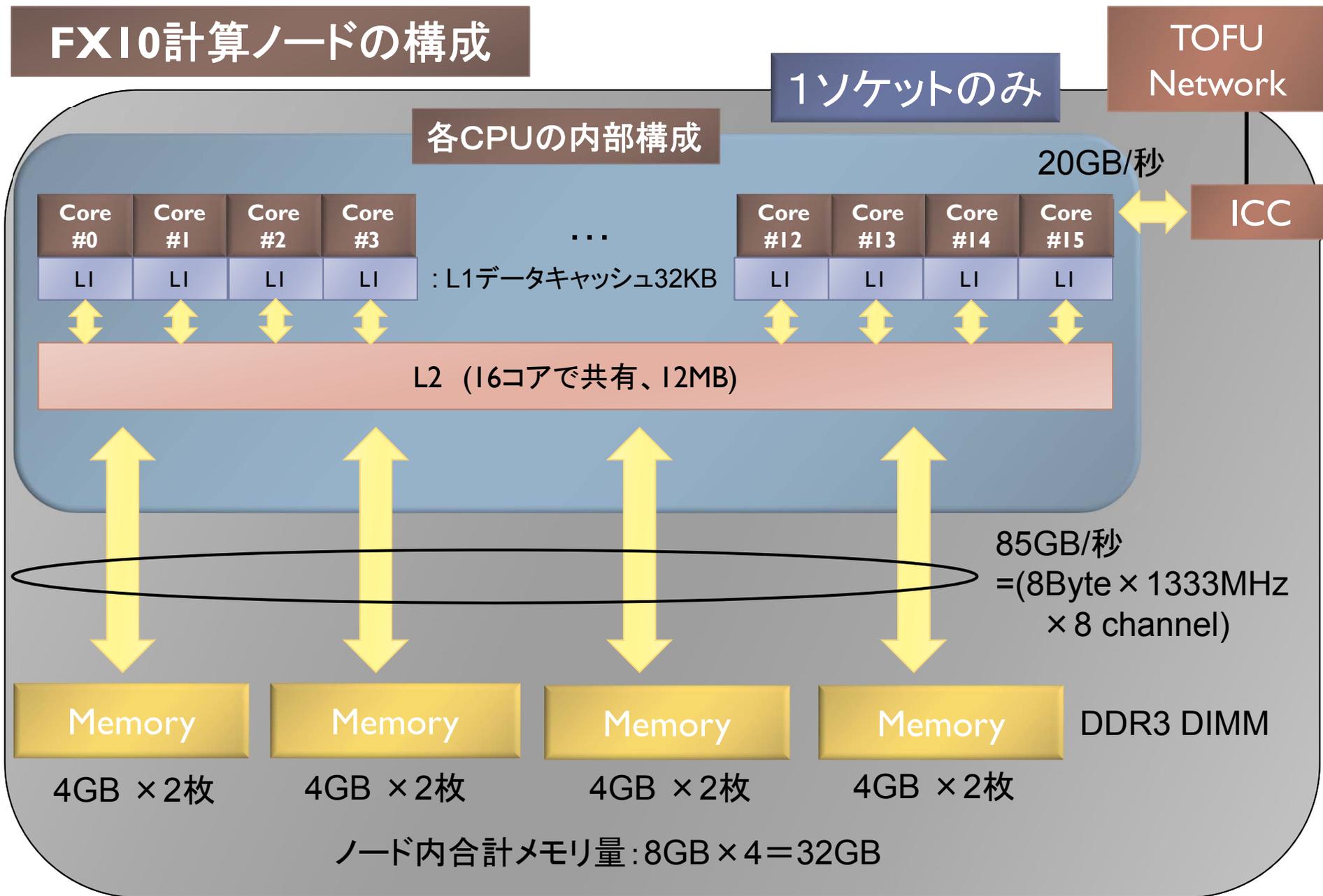
# FX10全体メモリ構成



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



# 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)

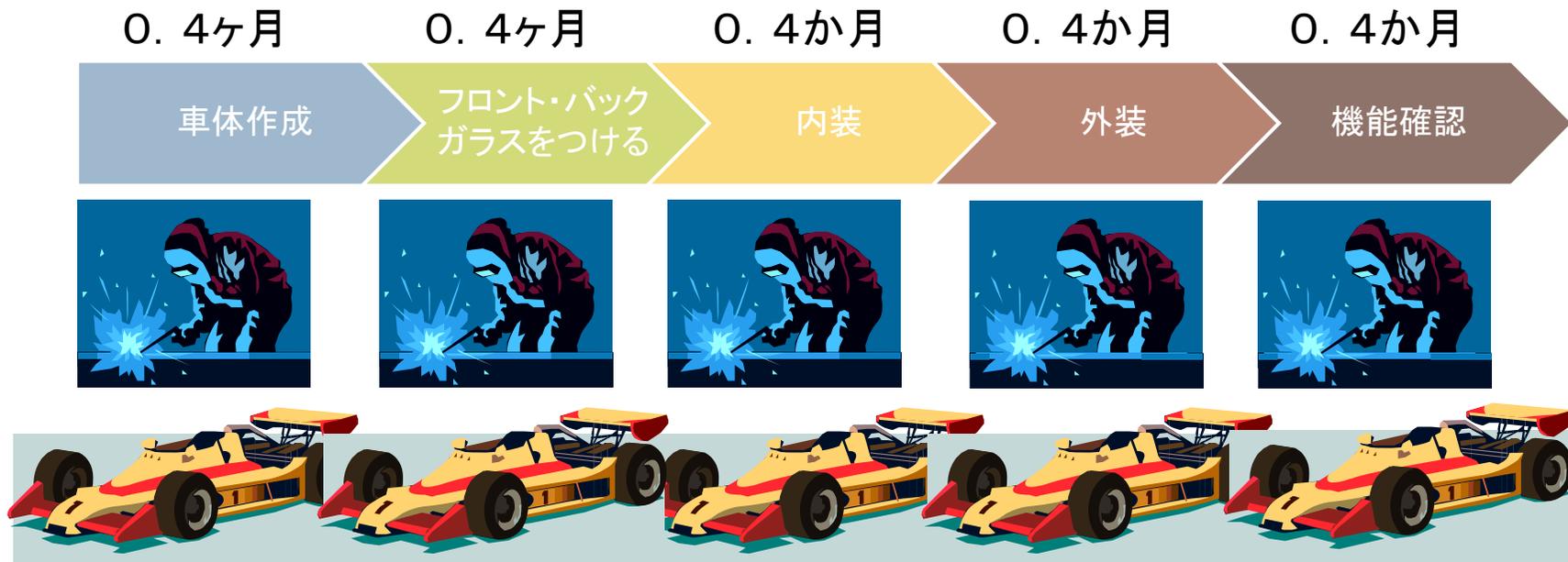
1台目  
2台目  
3台目



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・パイプラインニング

- ▶ 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・パイプラインニング

- ▶ 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コアあたり1.848GHzのクロックなので、
  - ▶ 理論最大演算は、  
 $1.848 \text{ GHz} * 8 \text{ 回} = \mathbf{14.784 \text{ GFLOPS / コア}}$
  - ▶ 1ノード16コアでは、  
 $14.784 * 16 \text{ コア} = \mathbf{236.5 \text{ GFLOPS / ノード}}$
- ▶ レジスタ数(浮動小数点演算用)
  - ▶ **256個 / コア**

---

# ループアンローリング

コンパイラがやりそうでなかなかやらないんだな

## 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

---

- ▶ コンパイラ用語では、最内側のループの展開のこと
  - ▶ 狭義のループアンローリング
- ▶ アプリ屋さんは、多重ループのどのループでも展開することをいう
  - ▶ 広義のループアンローリング
  - ▶ もしくはコンパイラ用語で、ループリストラクチャリング（ループ再構成）の一種

# 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
```

---

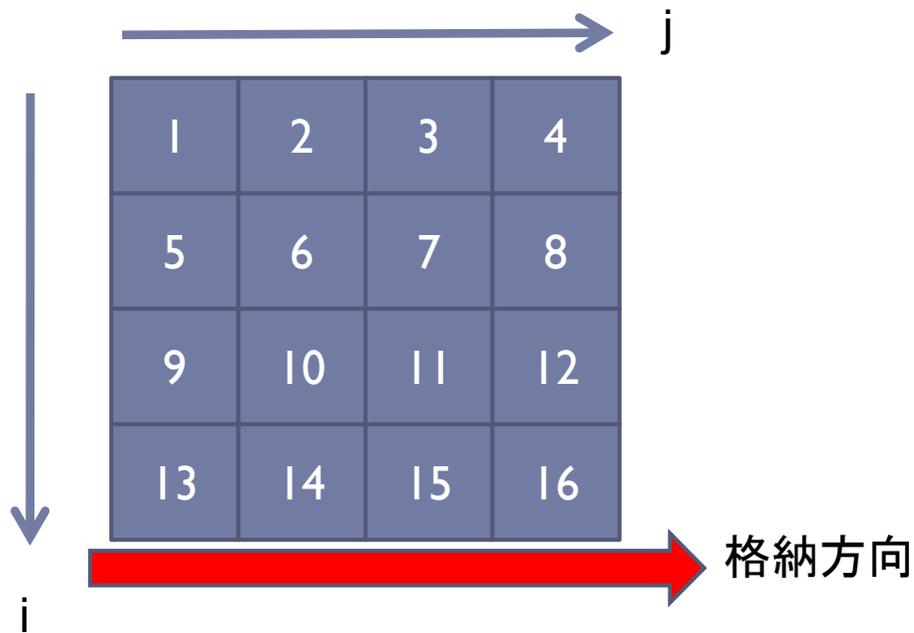
# 配列連続アクセス

とびとびアクセスは弱い

# 配列の格納方式

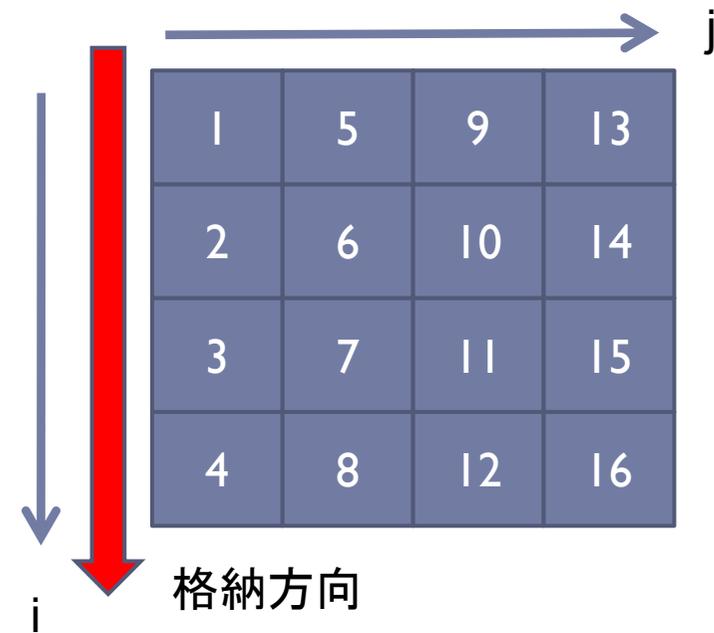
## ▶ C言語の場合

$A[i][j]$



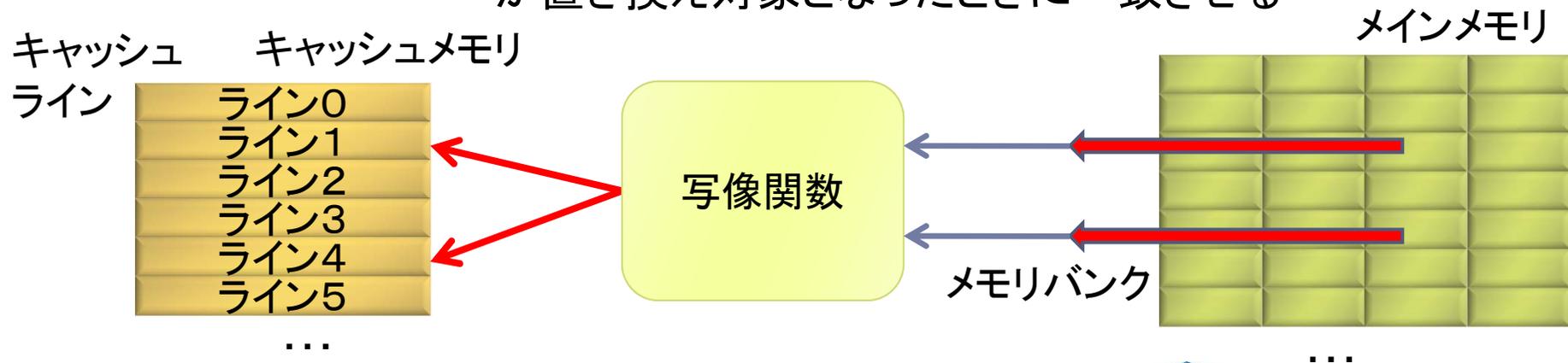
## ▶ Fortran言語の場合

$A(i, j)$



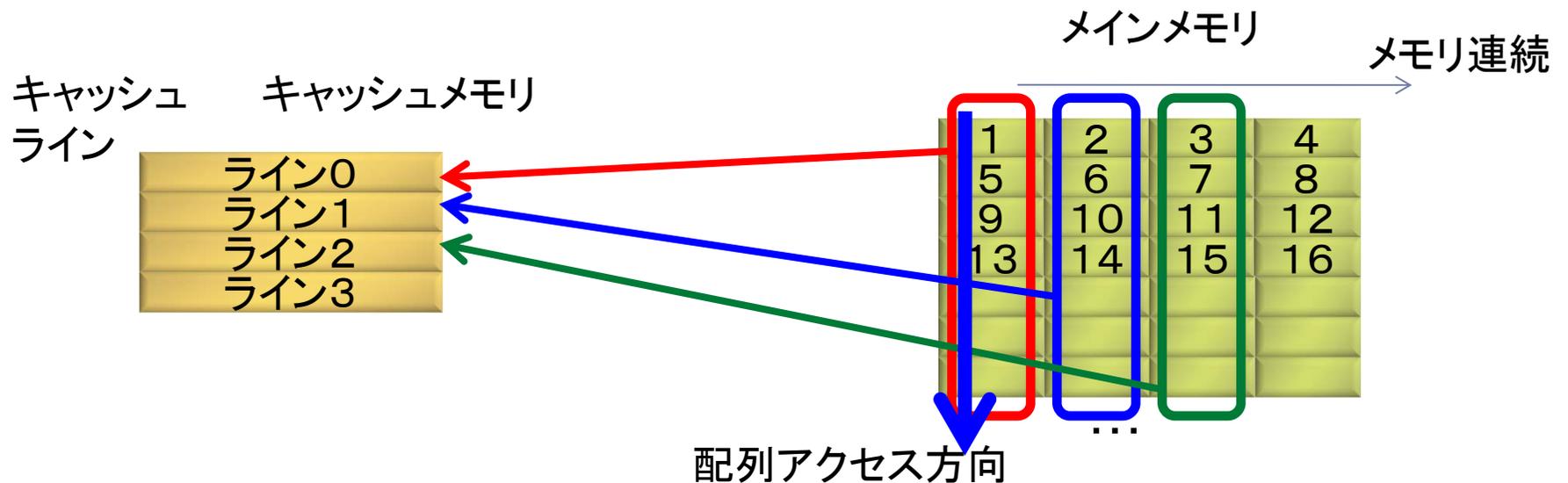
# キャッシュとキャッシュライン

- ▶ 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
  - ▶ 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**: 単位(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する(間接的)
  - ▶ 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**: 対象となる単位(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



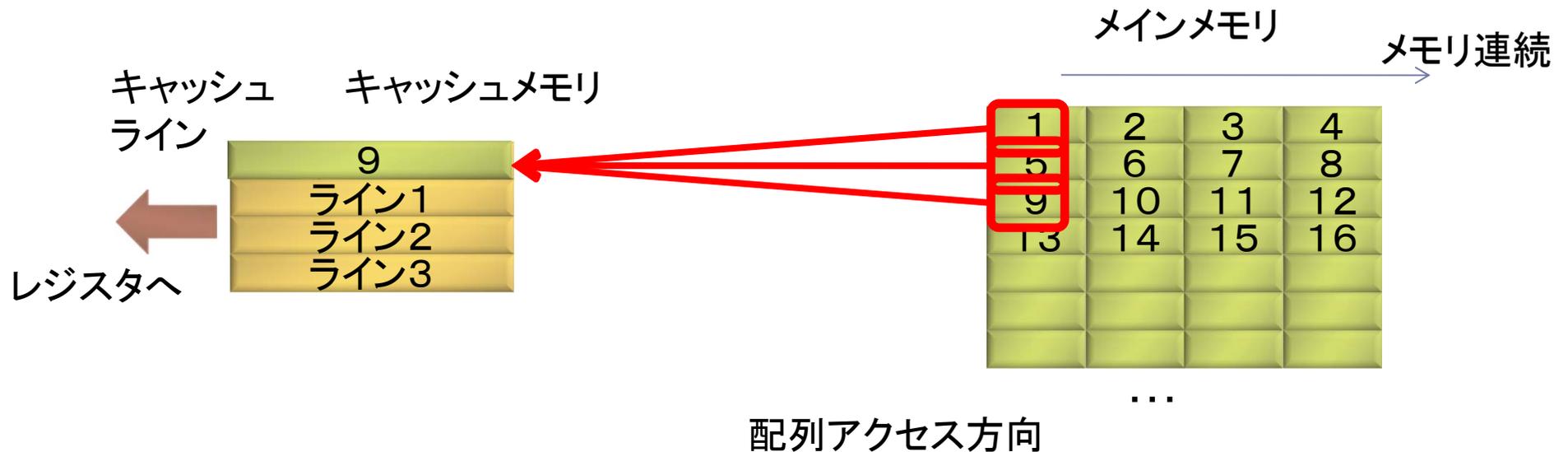
# キャッシュライン衝突

- ▶ 直接メインメモリのアドレスをキャッシュに写像する  
簡単なダイレクト・マッピングを考える
- ▶ このマッピングの間隔を、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のる
- ▶ この例で、格納方向と逆方向に連続アクセスする  
(=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

1. この場合、データ1がキャッシュライン0に乗ったあと、すぐにデータ5がアクセスされるため、キャッシュライン0のデータを追い出さないといけない
2. 同様に、データ5がキャッシュライン0に乗ったあと、すぐにデータ9がアクセスされるため、キャッシュライン0のデータを追い出さないといけない

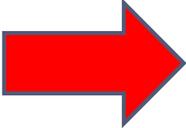


# キャッシュライン衝突

- ▶ 以上の、1、2の状態が連続して発生する。
  - ▶ メモリ→キャッシュの回線が詰まっている（お話し中状態で待たされる）
- ▶ メモリからデータを逐次で読み出しているのと同じになる。
  - ▶ キャッシュがないのと同じ。
  - ▶ 演算器にデータが高速に届かず、演算パイプラインが中断し、演算器の利用効率が悪くなる。
- ▶ 以上の現象を、(キャッシュの) <スラッシング>、  
▶ <キャッシュライン衝突>、<キャッシュ合同>

## メモリアンタリービング

- ▶ 物理的なメモリの格納方向に従いアクセスする場合
- ▶ データのアクセス時、現在アクセス中のメモリ上の管理単位(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一キャッシュライン上に乗せる機能がある
- ▶ ライン0のデータをアクセスしている最中に、ライン1中に近隣のバンク内データを(並列に)持ってくる事が可能
  - ▶ **メモリの<インタリービング>**
- ▶ 演算機から見た場合、データアクセス時間の短縮になる
- ▶ 演算器が遊ぶ時間が少なくなる(=演算効率が高くなる)

 **物理的なデータ格納方向に連続アクセスする  
ループ構成にする**

# キャッシュライン衝突が起こる条件

- ▶ キャッシュラインへのメモリバンク割り付けは、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が高い。
- ▶ 実際はもっと複雑なので、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が生じた場合防ぐ方法は以下  
(このサイズの計算を避けるという自明な解以外)
  1. **パディング法**: 配列に(2<sup>冪</sup>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たうえで、`(&A)++;`など
    - ▶ コンパイラによるオプション在り
  2. **データ圧縮法**: 計算に必要なデータのみ、新しい配列をキャッシュライン衝突しないように確保し、必要なデータを移す。
  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ことで対応。

# FX10特有の回避法

---

- ▶ Sparc64 lvmでは、L1キャッシュラインは2Wayのため、キャッシュライン衝突が起こりやすい
- ▶ 特に、OpenMPなど、スレッド実行時には、起こる確率が増す
- ▶ そこで、OpenMPのスレッド実行の方法を、StaticからDynamic(Cyclic)にすることで、隣のコアがL2にロードした情報を再利用し、L1キャッシュライン衝突を防げることが報告されている。
  - 
  - ▶ `!$OMP DO SCHEDULE(static,1)`
- ▶ 参考
  - ▶ 理化学研究所 次世代スーパーコンピュータ開発実施本部 開発グループ アプリケーション開発チーム 南 一生 氏
  - ▶ 「スーパーコンピュータ「京」におけるアプリケーションの高並列化と高性能化」、SACSYS2012チュートリアル資料
- ▶ <http://sacsis.hpcc.jp/2012/files/SACSYS2012-tutorial1-pub.pdf>

---

# サンプルプログラムの実行 (行列-行列積)

# UNIX防忘録

---

- ▶ emacsの起動: emacs 編集ファイル名
  - ▶  $\text{^x ^s}$  (^はcontrol) : テキストの保存
  - ▶  $\text{^x ^c}$  : 終了  
( $\text{^z}$  で終了すると、スパコンの負荷が上がる。絶対にしないこと。)
  - ▶  $\text{^g}$  : 訳がわからなくなったとき。
  - ▶  $\text{^k}$  : カーソルより行末まで消す。消した行は一時記憶される。
  - ▶  $\text{^y}$  :  $\text{^k}$ で消した行を、現在のカーソルの場所にコピーする。
  - ▶  $\text{^s 文字列}$  : 文字列の箇所まで移動する。
  - ▶  $\text{^M x goto-line}$  : 指定した行まで移動する。

# UNIX防忘録

---

- ▶ **rm** **ファイル名** : ファイル名のファイルを消す。
  - ▶ **rm \*~** : test.c~ などの、~がついたバックアップファイルを消す。
- ▶ **ls** : 現在いるフォルダの中身を見る。
- ▶ **cd** **フォルダ名** : フォルダに移動する。
  - ▶ **cd ..** : 一つ上のフォルダに移動。
  - ▶ **cd ~** : ルートディレクトリに行く。訳がわからなくなったとき。
- ▶ **cat** **ファイル名** : ファイル名の中身を見る
- ▶ **make** : 実行ファイルを作る (Makefile があるところでしか実行できない)
  - ▶ **make clean** : 実行ファイルを消す。

## 行列-行列積のサンプルプログラムの注意点

---

- ▶ C言語版およびFortran言語版のファイル名  
**Mat-Mat-noopt-fx.tar**
- ▶ ジョブスクリプトファイル**mat-mat-noopt.bash**  
中のキュー名を **lecture から**  
**lecture4 (工学部共通)**  
**に変更してから pjsub してください。**
- ▶ **lecture : 実習時間外のキュー**
- ▶ **lecture4 : 実習時間内のキュー**

# 行列-行列積のサンプルプログラムの実行 (C言語版、Fortran言語でも同様)

---

- ▶ 以下のコマンドを実行する

```
$ cp /home/z30082/Mat-Mat-noopt-fx.tar ./
```

```
$ tar xvf Mat-Mat-noopt-fx.tar
```

```
$ cd Mat-Mat-noopt
```

- ▶ 以下のどちらかを実行

```
$ cd C :C言語を使う人
```

```
$ cd F :Fortran言語を使う人
```

- ▶ 以下は共通

```
$ make
```

```
$ pjsub mat-mat-noopt.bash
```

- ▶ 実行が終了したら、以下を実行する

```
$ cat mat-mat-noopt.bash.oXXXX (XXXXは数字)
```

## 行列-行列積のサンプルプログラムの実行

---

- ▶ 以下のような結果が見えれば成功  
(Fortran言語の場合)

**N = 512**

**Mat-Mat time[sec.] = 7.742279146972578**

**MFLOPS = 34.67137392317229**

## 行列-行列積のサンプルプログラムの実行

---

- ▶ 以下のような結果が見えれば成功  
(C言語の場合)

N = 512

Mat-Mat time = 3.682733 [sec.]

72.890293 [MFLOPS]

OK!

FortranとCで実行時間が違いますが、  
-OOオプションでの最適化方法の違いによる  
ものです

# サンプルプログラムの説明（C言語）

---

- ▶ `#define N 512`

の、数字を変更すると、行列サイズが変更できます

- ▶ `#define DEBUG 1`

「1」にすると、行列-行列積の演算結果が検証できます。

- ▶ `MyMatMat`関数の仕様

- ▶ `Double`型 $N \times N$ 行列AとBの行列積をおこない、`Double`型 $N \times N$ 行列Cにその結果が入ります

## Fortran言語のサンプルプログラムの注意

---

- ▶ 行列サイズNNの宣言は、以下のファイルにあります。

`mat-mat-noopt.inc`

- ▶ 行列サイズ変数が、NNとなっています。

`integer NN`

`parameter (NN=512)`

# 演習課題

---

- ▶ **MyMatMat**関数(手続き)を、アンローリングなどにより高速化してください
- ▶ どういうアンローリングの仕方がよいか、最も高速となる段数、などを調べてください。
  - ▶ コンパイラの最適化レベルを0にしてあります。本演習では、最適化レベルをとりあえず0で固定しておいてください。
  - ▶ コンパイラによる最適化を行い、かつ手によるアンローリングしてもよいのですが、場合によりアンローリングの効果がなくなります。

## レポート課題

1. [L10] 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。

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。

## レポート課題

---

- I. [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。  
。

## 参考文献（最適化全体）

---

1. 寒川光、「RISC超高速化プログラミング技法」、共立出版、ISBN4-320-02750-7、3,500円
2. Kevin Dowd著、久良知真子訳、「ハイ・パフォーマンス・コンピューティング：RISCワークステーションで最高の性能を引き出すための方法」、インターナショナル・トムソン・パブリッシング・ジャパン、ISBN4-900718-03-3、4,400円

## 参考文献 (SPARC64IXfx)

1. 大島聡史、「富士通 PRIMEHPC FX10 チューニング連載講座 1. ハードウェア概要」スーパーコンピューティングニュース、Vol.14 No.2 (2012年3月)  
<http://www.cc.u-tokyo.ac.jp/support/press/news/VOL14/No2/201203tuning-fx10-hard.pdf>
2. 富士通株式会社: White paper: スーパーコンピュータ PRIMEHPC FX10の先端技術  
<http://img.jp.fujitsu.com/downloads/jp/jhpc/primehpc/primehpc-fx10-hard-ja.pdf>
3. SPARC64 VIIIfx extension  
<http://img.jp.fujitsu.com/downloads/jp/jhpc/sparc64viii-fx-extensions.pdf>

---

来週へつづく

高性能プログラミング技法の基礎 (2)