計算科学技術 特論B

第3回
アプリケーションの性能最適化2
(CPU単体性能最適化)

2018年4月

国立研究開発法人理化学研究所
計算科学研究センター 運用技術部門
チューニング技術ユニット ユニットリーダー

南 一生
minami_kaz@riken.jp

K computer

講義の概要

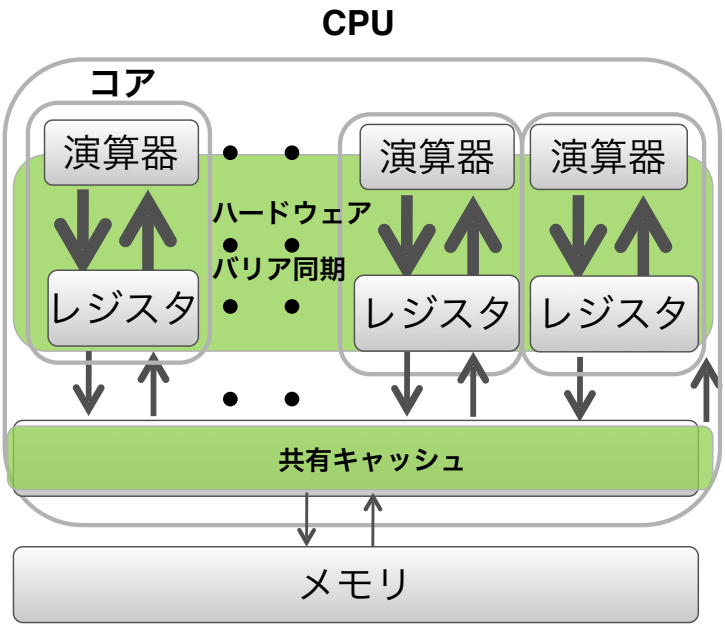
- スーパーコンピュータとアプリケーションの性能
- アプリケーションの性能最適化1 (高並列性能最適化)
- アプリケーションの性能最適化2 (CPU単体性能最適化)
- アプリケーションの性能最適化の実例1
- アプリケーションの性能最適化の実例2

内容

- スレッド並列化
- CPU単体性能を上げるための5つの要素
- 要求B/F値と5つの要素の関係
- 性能予測手法(要求B/F値が高い場合)
- 具体的テクニック

スレッド並列化

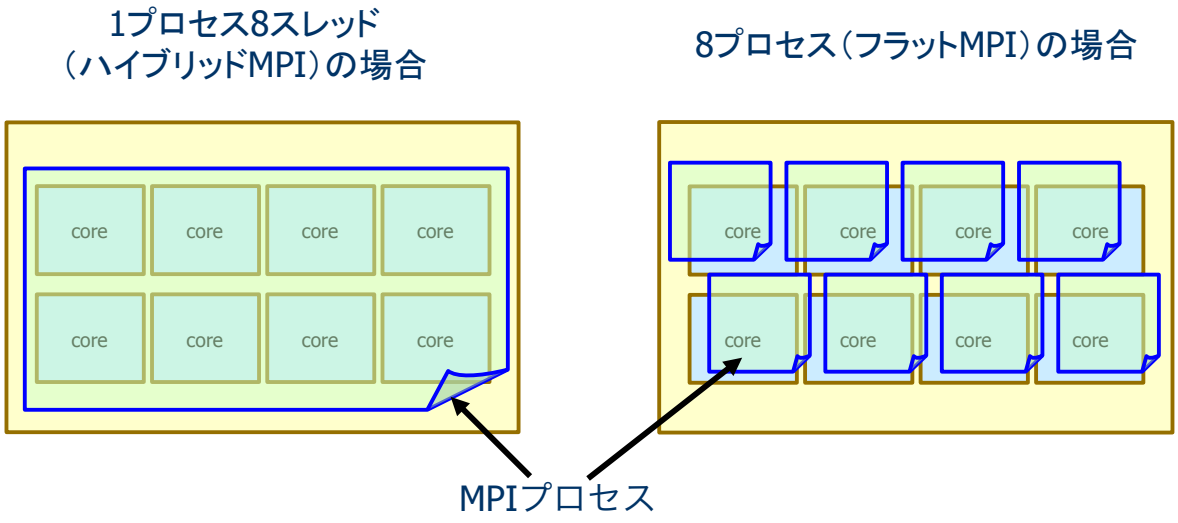
スレッド並列の概要



- 京の場合1CPUに8コア搭載.
- 8コアでL2キャッシュを共有.
- MPI等のプロセス並列に対しCPU内の8コアを使用するためのスレッド並列化が必要.
- スレッド並列化のためには自動並列化かOpenMPが使用可.
- 京の場合はハードウェアバリア同期機能を使用した高速なスレッド処理が可能.

ハイブリッド並列とフラットMPI並列

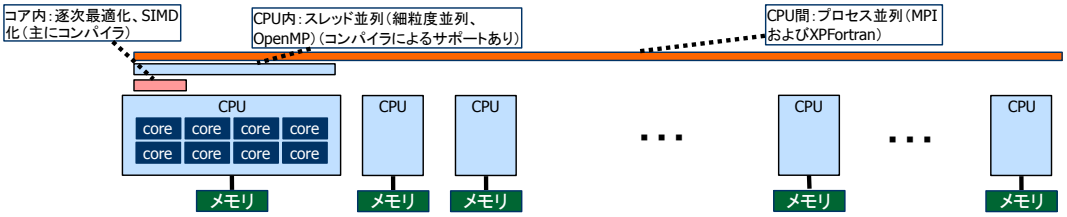
- MPIプロセス並列とスレッド並列の組み合わせをハイブリッド並列という.
- 各コアにMPIプロセスを割り当てる並列化をフラットMPI並列という.
- 京では通信資源の効率的利用，消費メモリ量を押さえる観点でハイブリッド並列を推奨している.



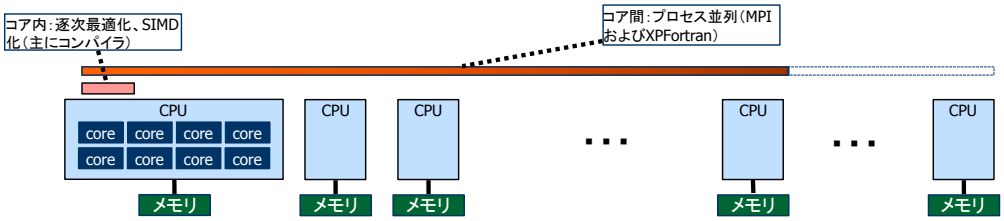
ハイブリッド並列とフラットMPI並列（京の場合）

- スレッド並列＋プロセス並列のハイブリッド型
  - コア内：コンパイラによる逐次最適化，SIMD化
  - CPU内：スレッド並列(自動並列化：細粒度並列化†，OpenMP)
  - CPU間：プロセス並列(MPI、XPFortran)

†細粒度並列化
高速バリア同期を活用し，内側ループをコア間で並列化
ベクトル向け（内側ループが長い）コードの高速化が可能



- プロセス並列型
  - コア内：コンパイラによる逐次最適化，SIMD化
  - コア間：プロセス並列(MPI、XPFortran)



実行効率の観点から、ハイブリッド型を推奨

CPU単体性能を上げるための5つの要素

CPU単体性能を上げるための5つの要素

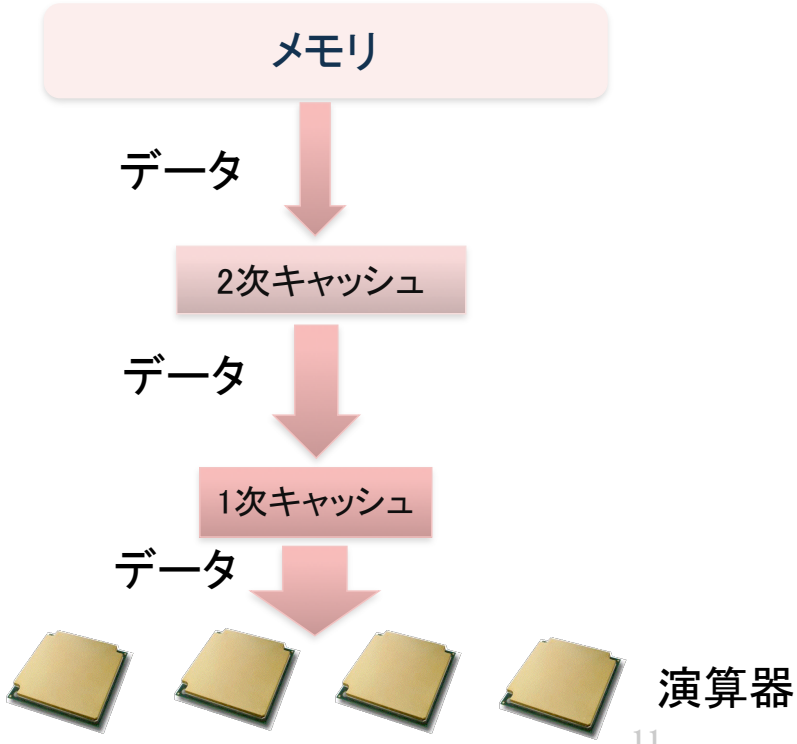
CPU内の複数コアでまずスレッド並列化が
できていると事は前提として

(1) ロード・ストアの効率化
(2) ラインアクセスの有効利用
(3) キャッシュの有効利用
(4) 効率の良い命令スケジューリング
(5) 演算器の有効利用

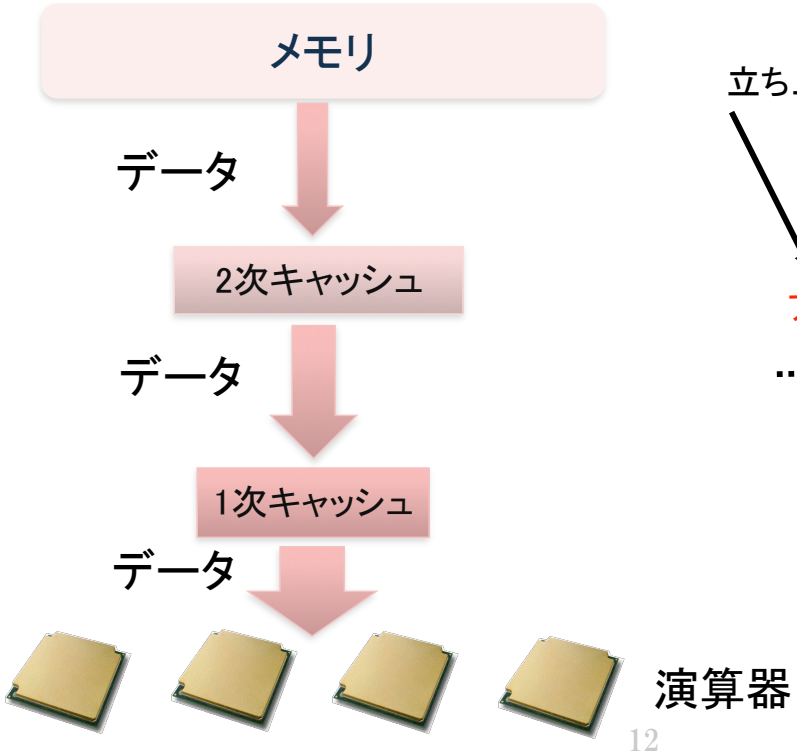
(1) ロード・ストアの効率化

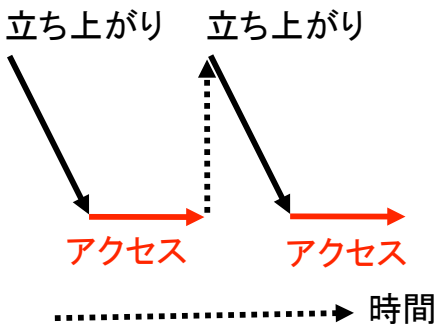
- プリフェッチの有効利用
- 演算とロード・ストア比の改善

レイテンシ(アクセスの立ち上がり)

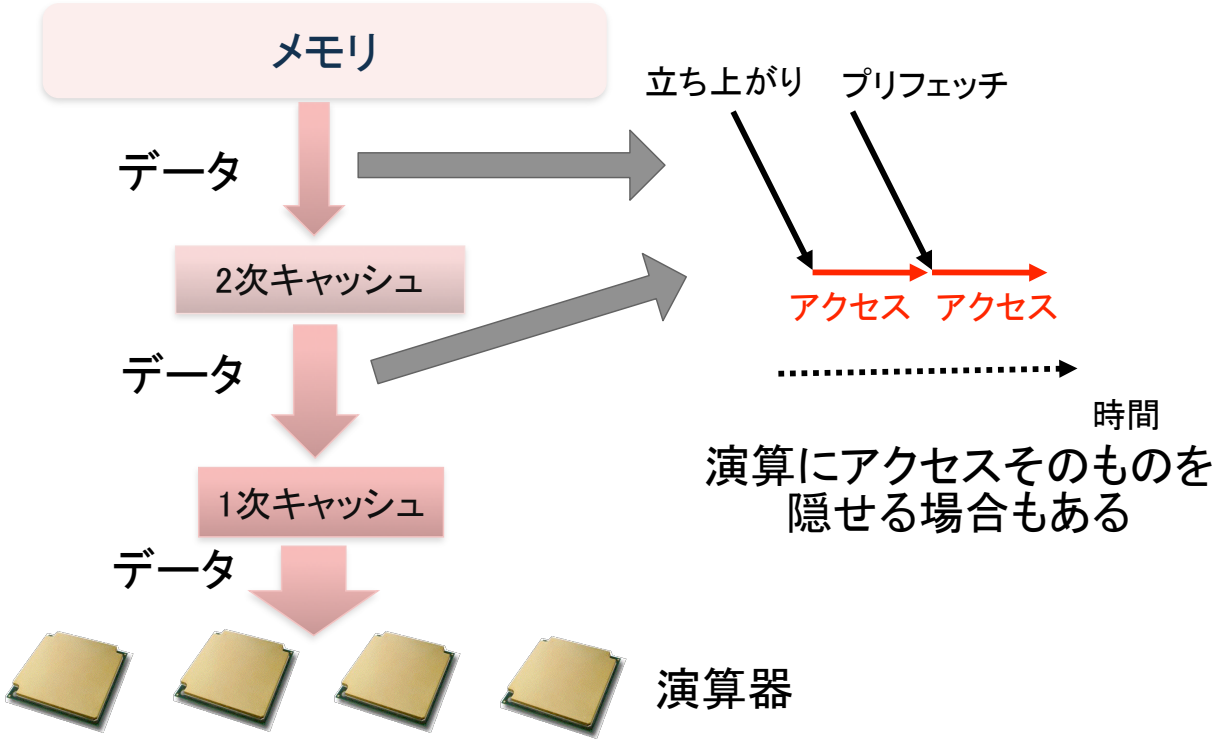


レイテンシ(アクセスの立ち上がり)





プリフェッチの有効利用

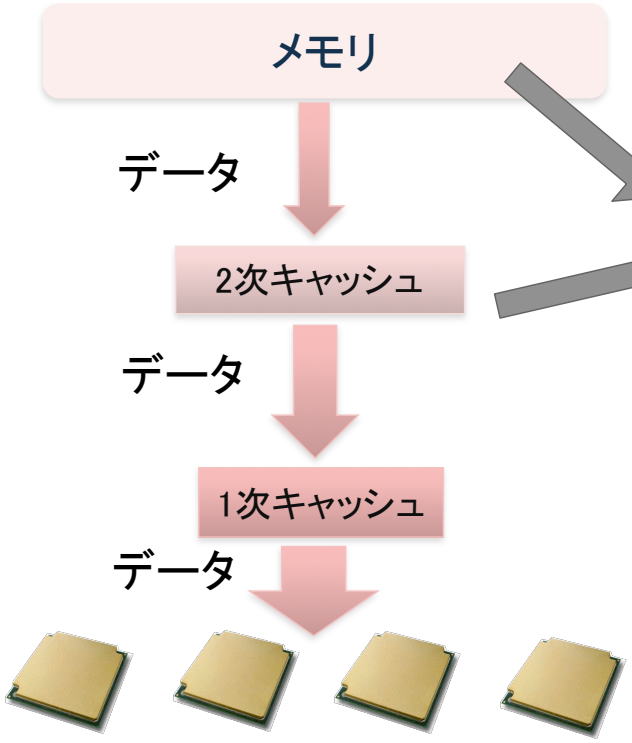


演算とロード・ストア比の改善

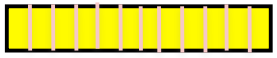
- 以下のコーディングを例に考える.
- 以下のコーディングの演算は和2個，積2個の計4個.
- ロードの数はx,a(i),a(i+1)の計3個.
- ストアの数はxの1個.
- したがってロード・ストア数は4個
- 演算とロード・ストアの比は4/4となる.
- なるべく演算の比率を高めロード・ストアの比率を低く抑えて演算とロードストアの比を改善を図る事が重要.

```
do  j=1,m
do  i=1,n
  x(i)=x(i)+a(i)*b+a(i+1)*d
end  do
```

(2) ラインアクセスの有効利用



キャッシュライン
128byte毎のアクセス



全部の要素を
計算に利用

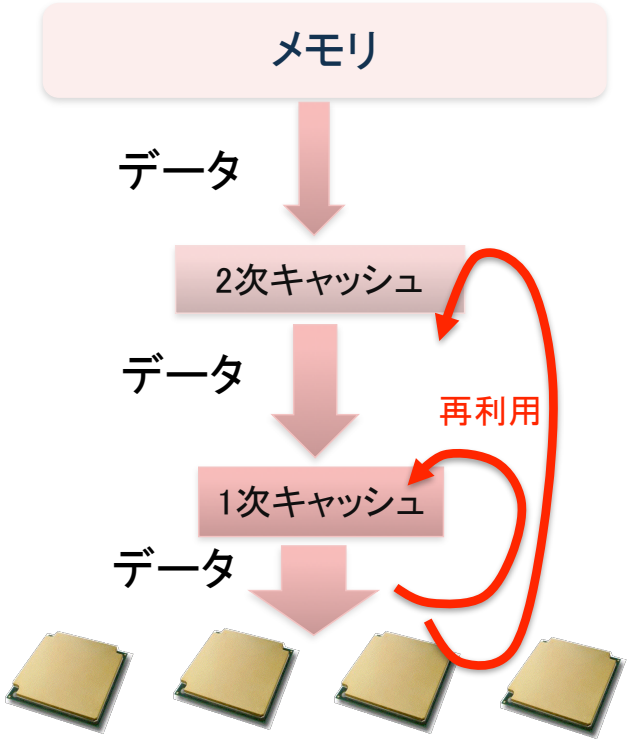
効率が良い

キャッシュライン
128byte毎のアクセス

1個の要素のみ
を計算に利用

効率が悪い

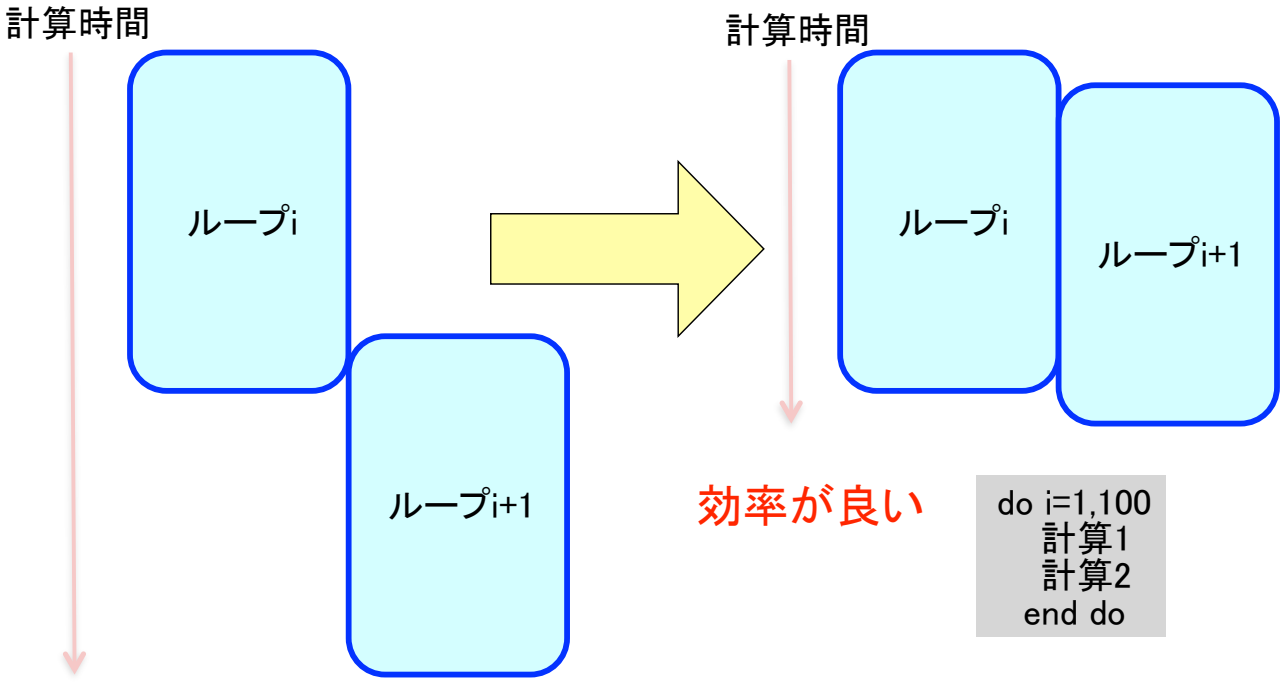
(3) キャッシュの有効利用



・データ供給能力の高いキャッシュを設ける
・キャッシュに置いたデータを何回も再利用し演算を行なう
・こうする事で演算器の能力を十分使い切る

- **ブロッキング**

(4) 効率の良い命令スケジューリング



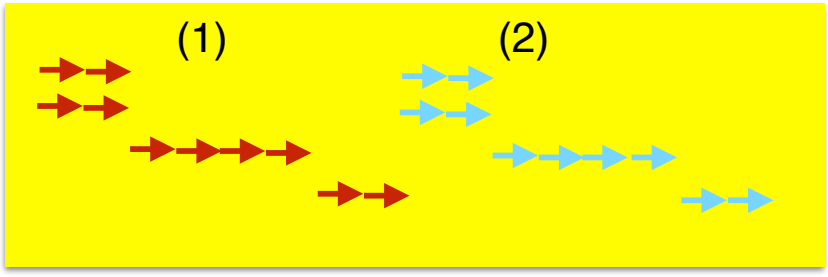
並列処理と依存性の回避

< ソフトウェアパイプラインニング>　**コンパイラ**

<前提>
・ロード2つorストアと演算とは同時実行可能
・ロードとストアは2クロックで実行可能
・演算は4クロックで実行可能
・ロードと演算とストアはパイプライン化されている

例えば以下の処理を考える。

do i=1,100
　a(i)のロード
　b(i)のロード
　a(i)とb(i)の演算
　i番目の結果のストア
end do



<実行時間>
・8クロック×100要素=800クロックかかる

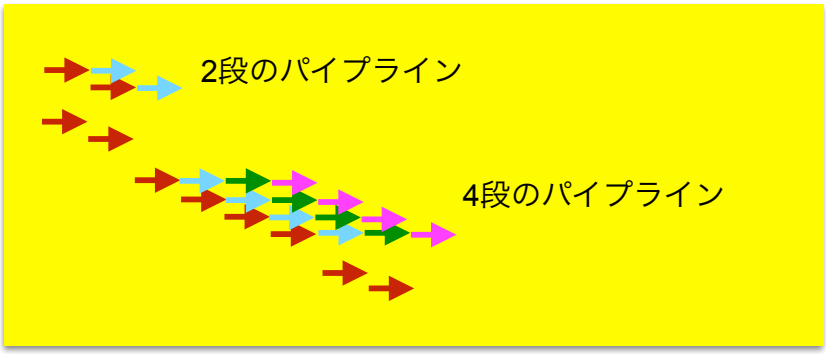
並列処理と依存性の回避

< ソフトウェアパイプラインニング> **コンパイラ**

<前提>
・ロード2つorストアと演算とは同時実行可能
・ロードとストアは2クロックで実行可能
・演算は4クロックで実行可能
・ロードと演算とストアはパイプライン化されている

「パイプライン化されている」の意味

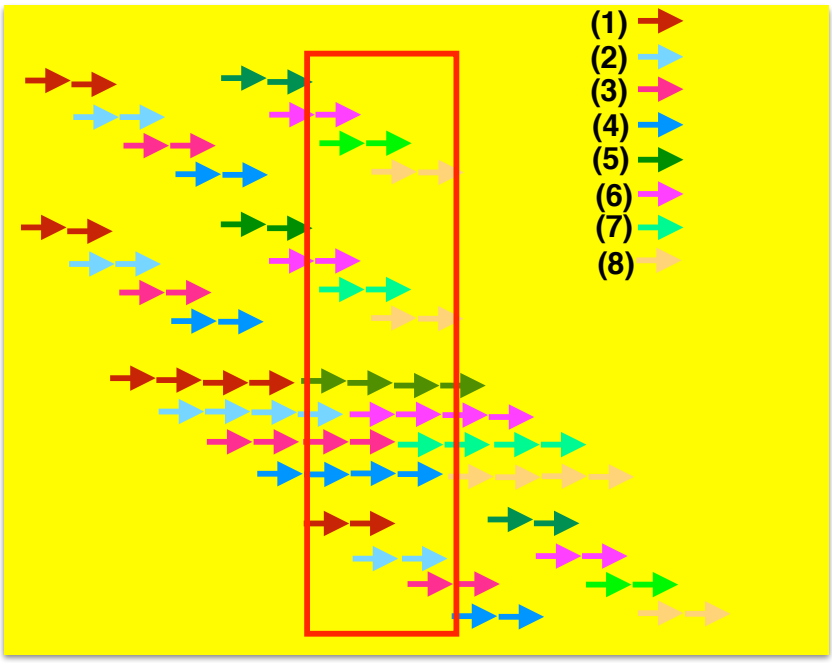
```
do i=1,100
      a(i)のロード
      b(i)のロード
      a(i)とb(i)の演算
      i番目の結果のストア
end do
```



並列処理と依存性の回避

< ソフトウェアパイプラインニング> **コンパイラ**

```
do i=1,100,4
      a(i)のロード
      a(i+1)のロード
      a(i+2)のロード
      a(i+3)のロード
      b(i+1)のロード
      b(i+2)のロード
      b(i+3)のロード
      b(i+4)のロード
      a(i)とb(i)の演算
      a(i+1)とb(i+1)の演算
      a(i+2)とb(i+2)の演算
      a(i+3)とb(i+3)の演算
      i番目の結果のストア
      i+1番目の結果のストア
      i+2番目の結果のストア
      i+3番目の結果のストア
end do
```



4段の展開：8要素の処理が完了

並列処理と依存性の回避

< ソフトウェアパイプラインニング> **コンパイラ**

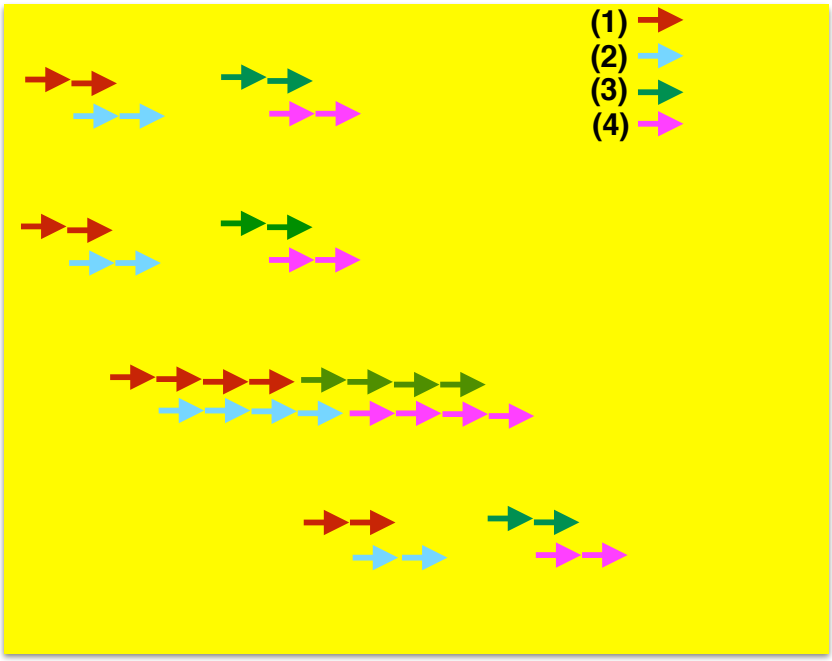
```
do i=1,100,2
      a(i)のロード
      a(i+1)のロード

      b(i+1)のロード
      b(i+2)のロード

      a(i)とb(i)の演算
      a(i+1)とb(i+1)の演算

      i番目の結果のストア
      i+1番目の結果のストア

end do
```

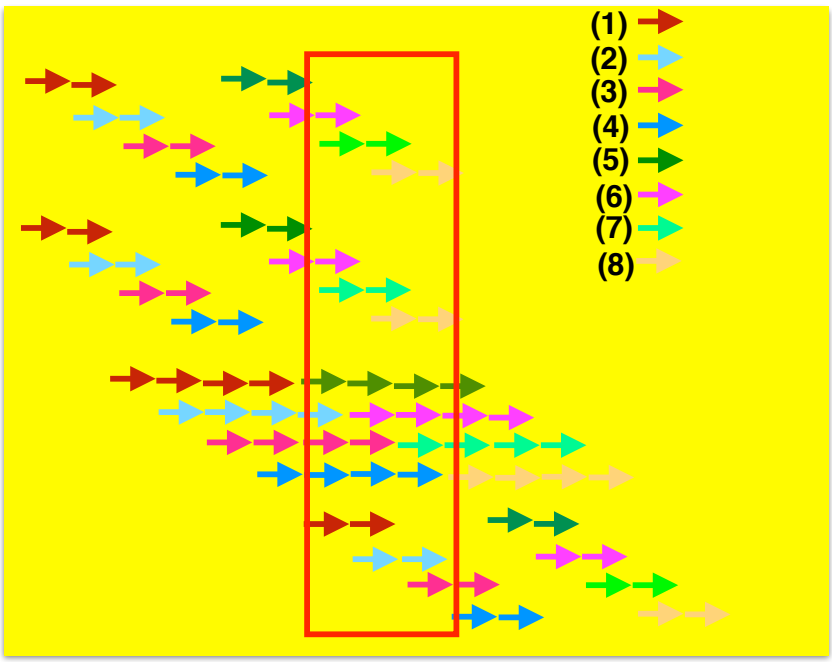


2段の展開：4要素の処理が完了

並列処理と依存性の回避

< ソフトウェアパイプラインニング> **コンパイラ**

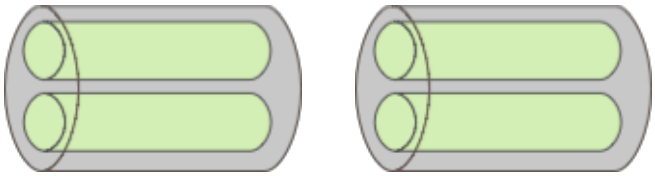
```
do i=1,100,4
      a(i)のロード
      a(i+1)のロード
      a(i+2)のロード
      a(i+3)のロード
      b(i+1)のロード
      b(i+2)のロード
      b(i+3)のロード
      b(i+4)のロード
      a(i)とb(i)の演算
      a(i+1)とb(i+1)の演算
      a(i+2)とb(i+2)の演算
      a(i+3)とb(i+3)の演算
      i番目の結果のストア
      i+1番目の結果のストア
      i+2番目の結果のストア
      i+3番目の結果のストア
end do
```



<実行時間>
・前後の処理及びメインループの立ち上がり部分を除くと
・1クロック×100要素=100クロックで処理できる

(5) 演算器の有効利用

２SIMD Multi&Add演算器×２本



乗算と加算を4個同時に計算可能

(1 + 1) × 4 = 8

この条件に
近い程高効率

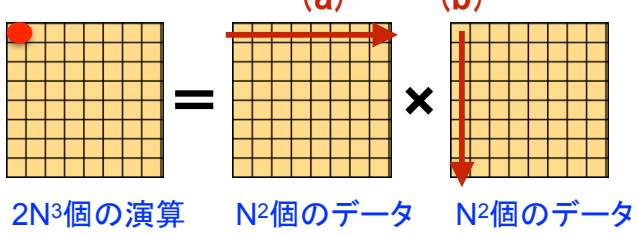
1コアのピーク性能：8演算×2GHz =16G演算/秒

要求B/F値と5つの要素の関係

要求B/F値と5つの要素の関係

アプリケーションの要求B/F値の大小によって性能チューニングにおいて注目すべき項目が異なる

行列行列積の計算（要求B/F値が小さい）



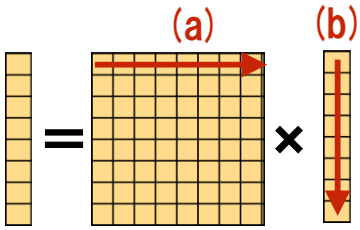
$2N^3$個の演算　N^2個のデータ　N^2個のデータ

(*)1要素が８Byteとする

B/F値
=移動量(Byte)/演算量(Flop)
$=8*2N^2/2N^3$
$=8/N$

原理的にはNが大きい程小さな値

行列ベクトル積の計算（要求B/F値が大きい）



$2N^2$個の演算　N^2個のデータ　N個のデータ

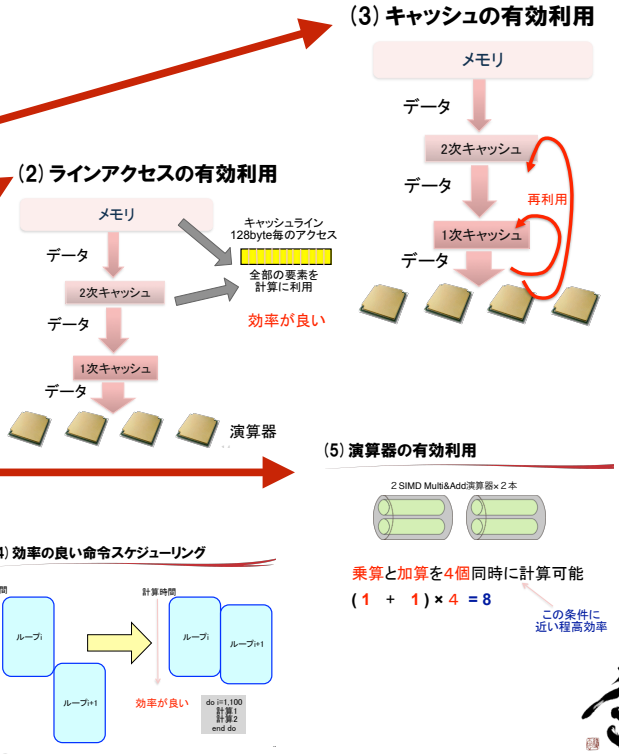
B/F値
=移動量(Byte)/演算量(Flop)
$=8*(N^2+N)/2N^2$
$\fallingdotseq 8/2=4$

原理的には8/Nより大きな値

要求B/F値と5つの要素の関係

要求するB/Fが小さいアプリケーションについて

- 原理的にキャッシュの有効利用が可能
- まずデータをオンキャッシュにするコーディング：(3)が重要
- つぎに2次キャッシュのライン上のデータを有効に利用するコーディング：(2)が重要
- それが実現できた上で(4)(5)が重要

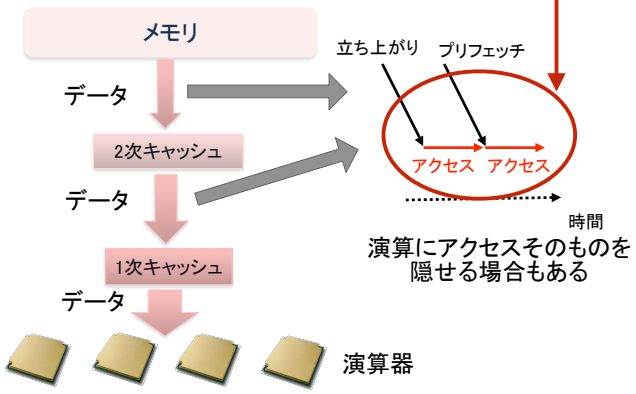


要求B/F値と5つの要素の関係

要求するB/Fが大きいアプリケーションについて

・メモリバンド幅を使い切る事が大事

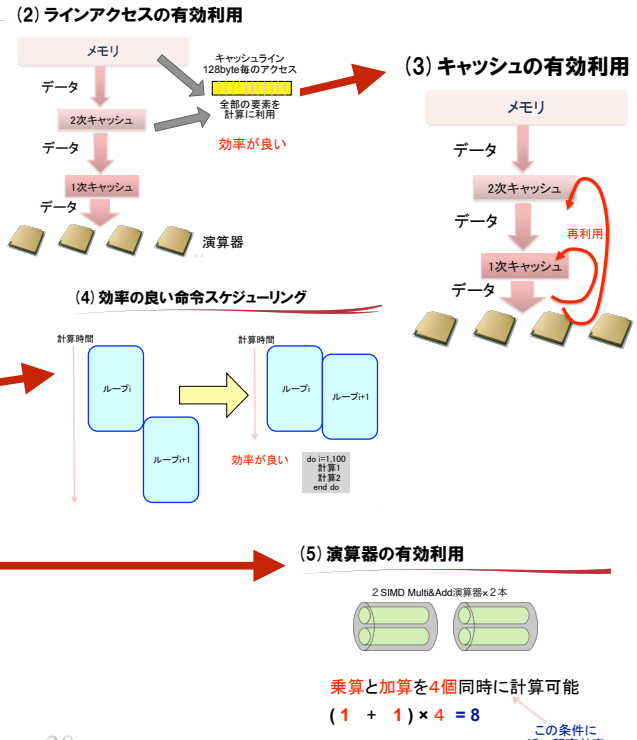
レイテンシーが隠れた状態にする事．この状態でメモリバンド幅のピーク(京の場合の理論値は64GB/sec)が出せる．レイテンシーが見えているとメモリバンド幅のピーク値は出せない．



要求B/F値と5つの要素の関係

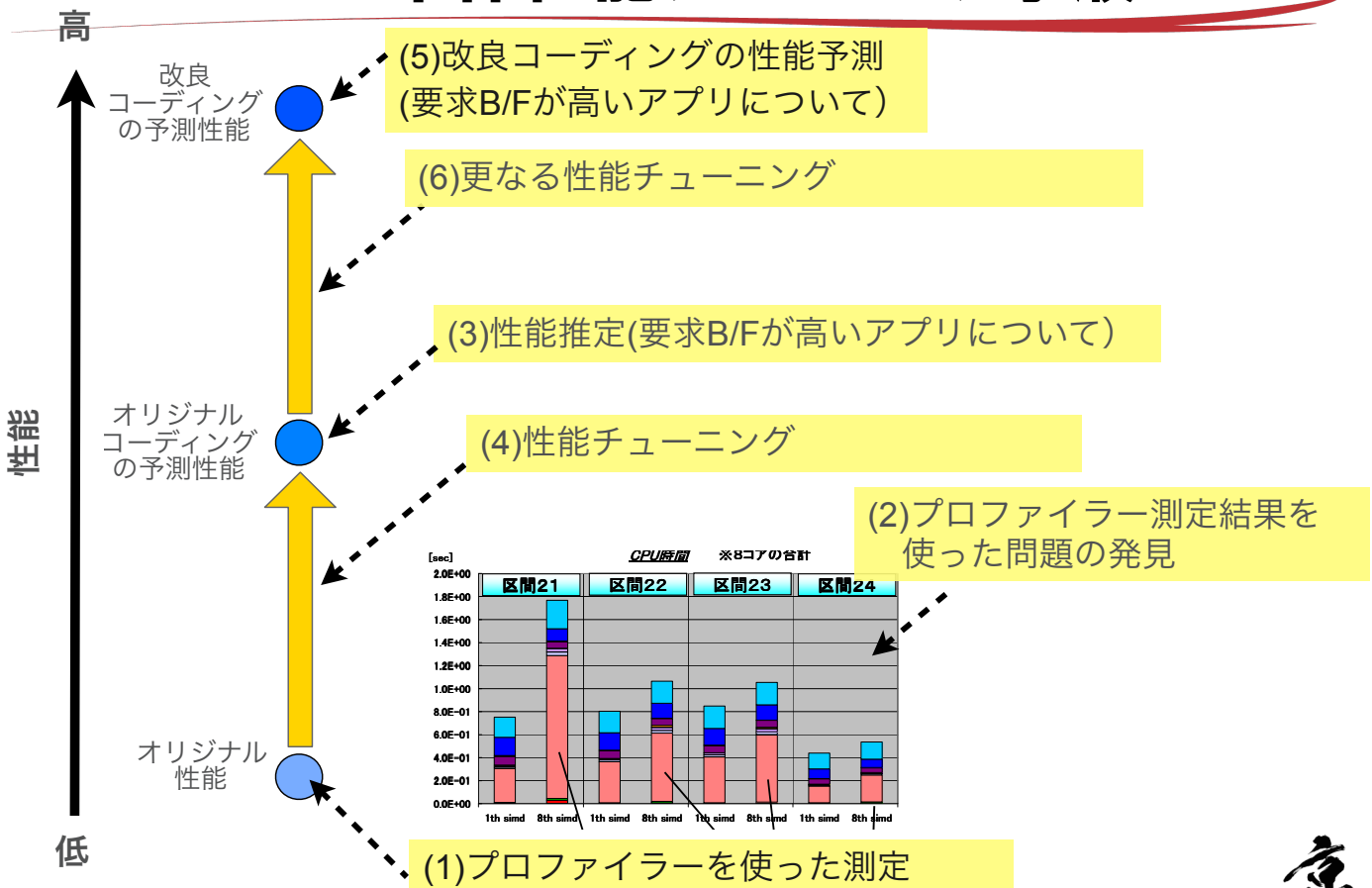
要求するB/Fが大きいアプリケーションについて

- 一番重要なのは(1)(2)
- 次にできるだけオンキャッシュする(3)が重要
- これら(1)(2)(3)が満たされ計算に必要なデータが演算器に供給された状態で、それらのデータを十分使える程度に(4)のスケジューリングができて、さらに(5)の演算器が有効に活用できる状態である事が必要

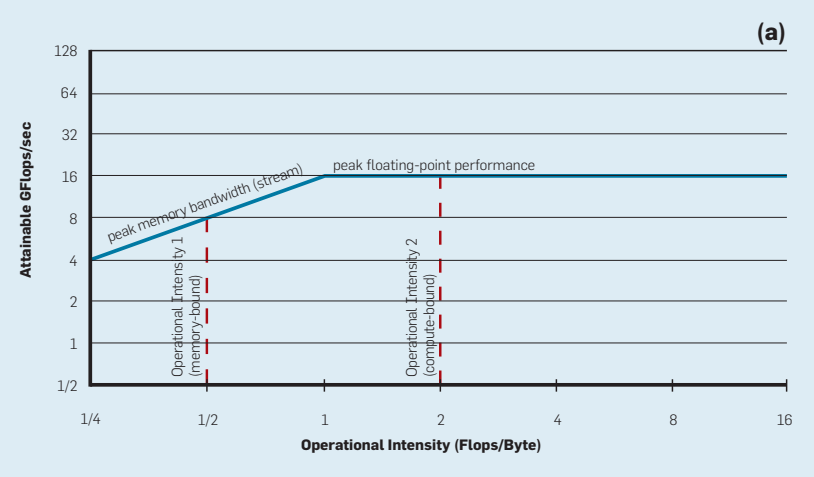


性能予測手法
(要求B/F値が大きい場合)

CPU単体性能チューニング手順



ルーフラインモデル



S. Williams, A. Waterman, and D. Patterson: Roofline: an insightful visual performance model for multicore architectures. Commun. ACM, 52:65–76, 2009.

- ハードウェアの実効的なメモリバンド幅：B
- ハードウェアのピーク性能：F
- アプリケーションの演算強度：X=f/b
  - アプリケーションの要求フロップス値：f
  - アプリケーションの要求バイト値：b

アプリケーションの性能：

min（F，B*X）

B*X=B*(f/b)=B*F*f/(b*F)=(B/F)/(b/f)*F

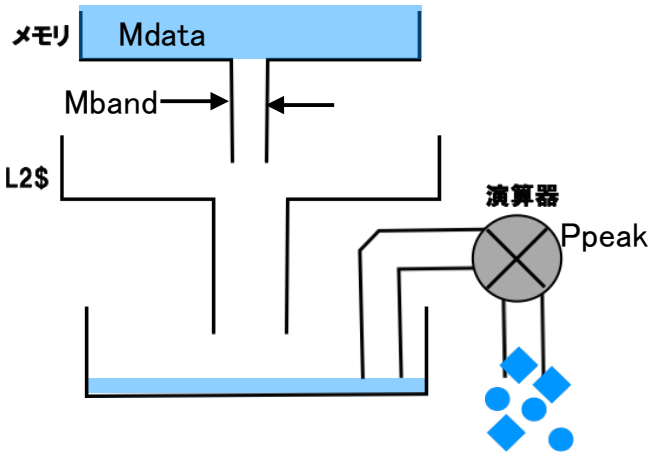
アプリケーションの性能　　：min(F, (B/F)/(b/f)*F)
アプリケーションのピーク性能比：min(1.0 , (B/F)/(b/f))

ルーフラインモデルの別の見方

(1)メモリ
- メモリデータ転送量：Mdata
- 実効メモリバンド幅：Mband,
- メモリデータ移動時間：Mtime

(2)演算
- プログラム演算量：Ca
- 演算ピーク性能：Ppeak
- 最小演算時間：Ctime

- 実行時間：Ex_time



$Mtime=Mdata/Mband$

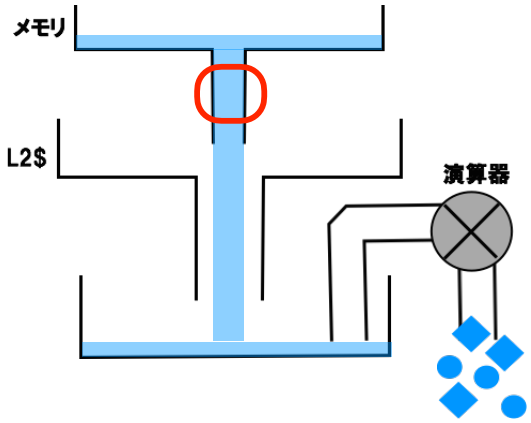
$Ctime=Ca/Ppeak$

演算器の処理とメモリのデータの処理が同時に終わるモデル

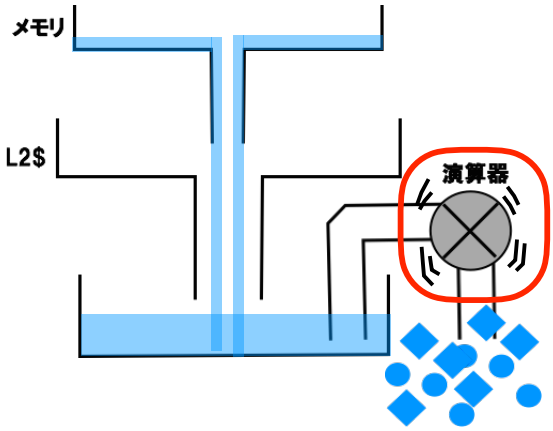
$Ex\_time=max(Mtime\ ,Ctime)$

ルーフラインモデルの別の見方

Ex_time=Mtimeの場合



Ex_time=Ctimeの場合



メモリとキャッシュ混合状態での性能予測モデル

(1)メモリ
- メモリデータ転送量：Mdata
- 実効メモリバンド幅：Mband,
- メモリデータ移動時間：Mtime

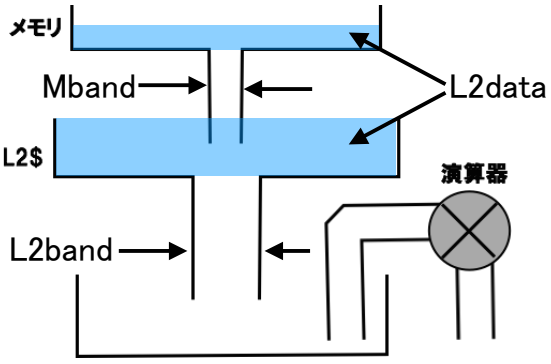
(2)L2キャッシュ
- L2データ転送量：L2data
- 実効L2バンド幅：L2band
- L2データ移動時間：L2time

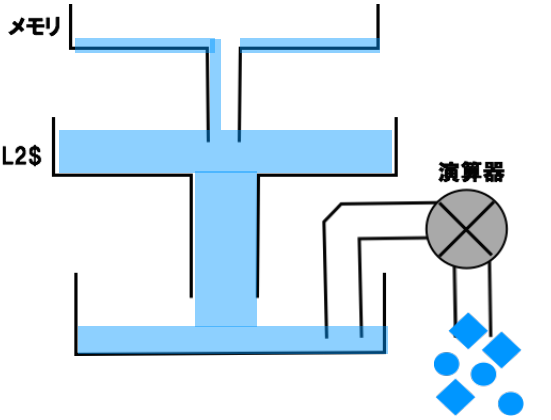
(3)演算
- プログラム演算量：Ca
- 演算ピーク性能：Ppeak
- 最小演算時間：Ctime
- 実行時間：Ex_time
- ピーク性能比：Cp

$$Mtime=Mdata/Mband\_peak$$
$$L2time=L2data/L2band\_peak$$
$$Ctime=Ca/Ppeak$$
$$Ex\_time=max(Mtime,L2time,Ctime)$$

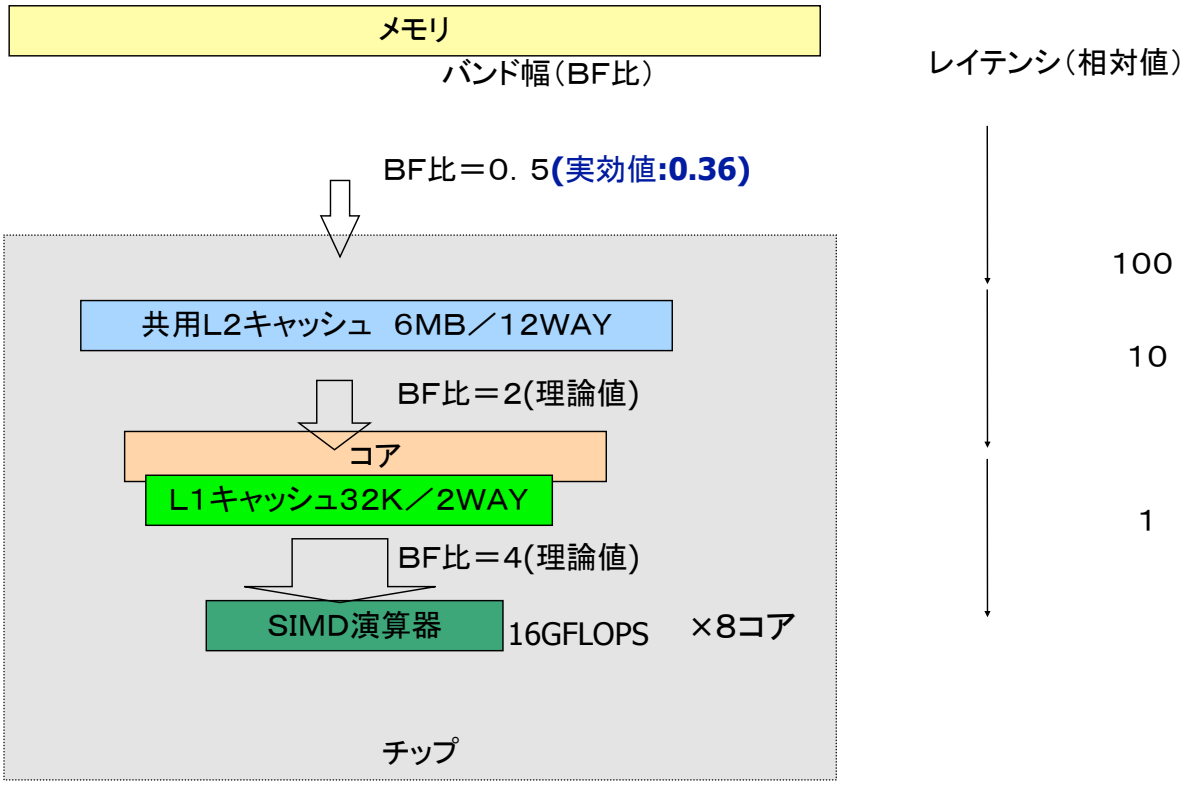
「キャッシュの効果を考慮したルーフラインモデルの拡張によるプログラムの性能予測」南一生他,情報処理学会論文誌コンピューティングシステム，第９巻，第２号，1-14(2016年6月)

Ex_time=L2timeの場合

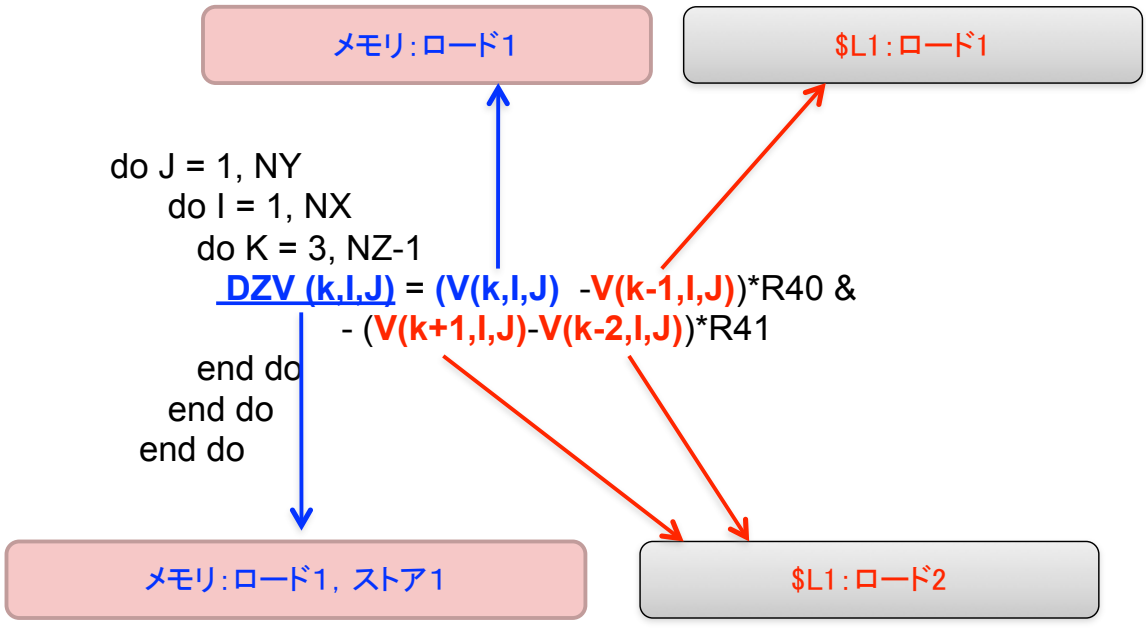




ベースとなる性能値



メモリとキャッシュアクセス（1）



ルーフラインモデルによる性能見積り

```
do J = 1, NY
   do I = 1, NX
     do K = 3, NZ-1
       DZV (k,I,J) = (V(k,I,J)  -V(k-1,I,J))*R40 &
                   - (V(k+1,I,J)-V(k-2,I,J))*R41
     end do
   end do
 end do
```

- 最内軸(K軸)が差分
- 1ストリームでその他の3配列は$L1に載っており再利用できる。

要求B/F	12/5 = 2.4
性能予測	0.36/2.4 = 0.15
実測値	0.153

アプリケーションのピーク性能比の予測値
：ハードウェアのBF値/アプリの要求BF値

要求Byteの算出:

1store,2loadと考える

4x3 = 12byte

要求flop:

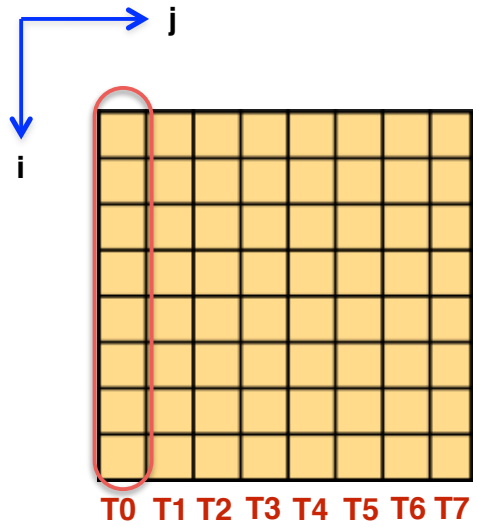
add：3 mult：2 = 5

具体的テクニック

（以降のハードウェア・コンパイラについての話題は京を前提とした話です）

(1) ロード・ストアの効率化
(2) ラインアクセスの有効利用
(3) キャッシュの有効利用
(4) 効率の良い命令スケジューリング
(5) 演算器の有効利用

スレッド並列化



Ti：スレッドiの計算担当

jループをブロック分割

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


CG法前処理のスレッド並列化

- 以下は前処理に不完全コレスキー分解を用いたCG法のアルゴリズムである.
- 前処理には色々な方法を使うことが出来る.
- 例えば前回説明したガウス・ザイデル法等である.
- CG法の本体である行列ベクトル積・内積・ベクトルの和等の処理は簡単に前頁のブロック分割されたスレッド並列化を行うことが出来る.
- しかしガウス・ザイデル前処理は前回講義で示したようにリカレンスがある.

ステップ1: $\alpha^k = (r_i^k \bullet (LL^T)^{-1} r_i^k)/(Ap_i^k \bullet p_i^k)$

ステップ2: $x_i^{k+1} = x_i^k + \alpha^k p_i^k$

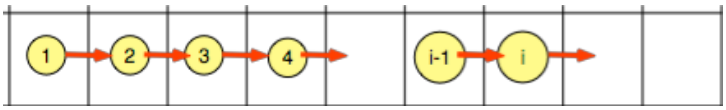
ステップ3: $r_i^{k+1} = r_i^k - \alpha^k A p_i^k$

ステップ4: $\beta^k = (r_i^{k+1} \bullet (LL^T)^{-1} r_i^{k+1})/(r_i^k \bullet (LL^T)^{-1} r_i^k)$

ステップ5: $p_i^{k+1} = (LL^T)^{-1} r_i^{k+1} + \beta^k p_i^k$

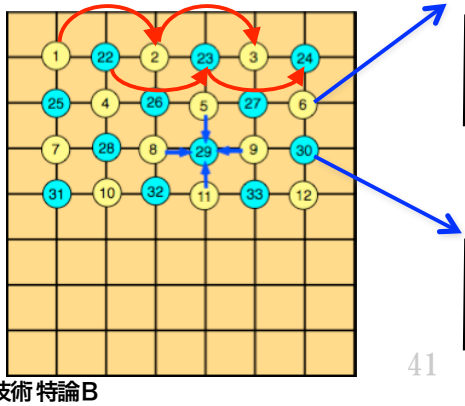
CG法前処理のスレッド並列化

- リカレンスがあると，前回講義のように依存関係があり並列処理ができない．
- つまりこのままではスレッド並列化もできない．



- そこで例えばRED-BLACKスィープに書換えリカレンスを除去しそれぞれのループをブロック分割によるスレッド並列化を行う．

RED-BLACKスイープ



黄色ループをブロック分割

```
do j=1,n
```

青色ループをブロック分割

```
do j=1,n
```


ロード・ストアの効率化

演算とロード・ストア比の改善　< 内側ループアンローリング>

- 以下の様な2 つのコーディングを比較する。

```
do j=1,m
do i=1,n
   x(i)=x(i)+a(i)*b+a(i+1)*d
end do
end do
```

本例は教科書的なもので、キャッシュの考慮、コンパイラの処理の考慮、レジスタ数の考慮により一概に効果があるものではない。

```
do j=1,m
do i=1,n,2
   x(i)=x(i)+a(i)*b+a(i+1)*d
   x(i+1)=x(i+1)+a(i+1)*b+a(i+2)*d
end do
end do
```

- 最初のコーディングの演算量は4 ,ロード/ストア回数は4 である.2 つ目のコ ーディングの演算量は8 ,ロード/ストア回数は7 である.
- 最初のコーディングの演算とロード/ ストアの比は4/4,2つ目のコーディングの演算とロード/ストアの比は8/7となり良くなる.

ロード・ストアの効率化

演算とロード・ストア比の改善

本例は教科書的なもので、キャッシュの考慮、コンパイラの処理の考慮、レジスタ数の考慮により一概に効果があるものではない。

<外側ループストリップ・マイニング>

・ij型のコーディングを以下のようなものとする。

```
do i=1,n
  do j=1,m
    y(i)=y(i)+a(i,j)*x(j)
```

・この場合a(i,j)、x(j)の２個をロードして２個の演算を実施する。y(i)はレジスタ上に保持しておけばよい。

・したがって演算とロード/ストアの比は1/1である。

・ji型のコーディングを以下のようなものとする。

```
do j=1,n
  do i=1,m
    y(i)=y(i)+a(i,j)*x(j)
```

・この場合a(i,j)、y(i)の２個をロードし、さらにy(i)をストアし２個の演算を実施する。

・したがって演算とロード/ストアの比は2/3である。

ロード・ストアの効率化

演算とロード・ストア比の改善

・以下のようなコーディングを外側ループストリップ・マイニングという。

```
do is=1,m,10
  do j=1,n
    do i=is,min(is+9,m)
      y(i)=y(i)+a(i,j)*x(j)
```

本例は教科書的なもので、キャッシュの考慮、コンパイラの処理の考慮、レジスタ数の考慮により一概に効果があるものではない。

・このコーディングの最内ループをアンローリングする。

```
do is=1,m,10
  do j=1,n
    y(is)=y(is)+a(is,j)*x(j)
    y(is+1)=y(is+1)+a(is+1,j)*x(j)
      ・・・・          ・・・・
    y(is+9)=y(is+9)+a(is+9,j)*x(j)
```

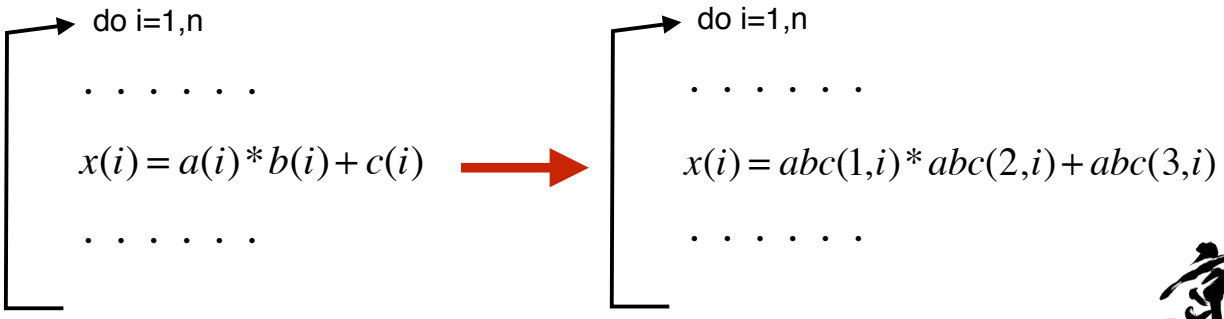
・この場合a(is,j)・・a(is+9,j)の10個とx(j)の1個をロードするのみですむ。y(js)・・y(is+9)はレジスタ上に保持しておけばよい。この間20個の演算を実行する。

・したがって演算とロード/ストアの比は20/11である。

ロード・ストアの効率化

プリフェッチの有効利用

- プリフェッチにはハードウェアプリフェッチとソフトウェアプリフェッチがある.
- ハードウェアプリフェッチは連続なキャッシュラインのアクセスとなる配列について自動的に機能する.
- ソフトウェアプリフェッチはコンパイラが必要であれば自動的に命令を挿入する.
- ユーザーがコンパイラオプションやディレクティブで挿入する事もできる.
- ハードウェアプリフェッチはループ内の配列アクセス数が一定の数を超えると効かなくなる.
- その場合はアクセスする配列を統合する配列マージで良い効果が得られる場合がある.



ロード・ストアの効率化

プリフェッチの有効利用

- ユーザーがコンパイラオプションやディレクティブで挿入する事もできる.
- 以下ディレクティブによるソフトウェアプリフェッチ生成の例.

改善後ソース（最適化制御行チューニング）

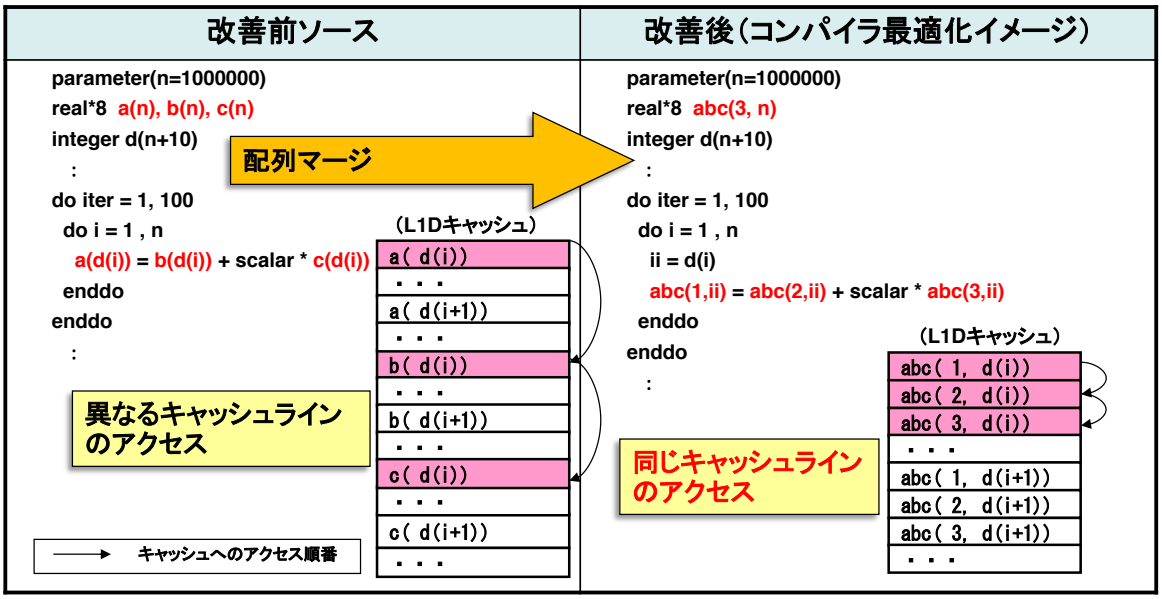
```
51              !ocl prefetch
                <<< Loop-information Start >>>
                <<< [PARALLELIZATION]
                <<<   Standard iteration count: 616
                <<< [OPTIMIZATION]
                <<<   SIMD
                <<<   SOFTWARE PIPELINING
                <<<   PREFETCH      : 32
                <<<     c: 16, b: 16
                <<< Loop-information
52    1  pp  2v          do i = 1 , n
53    1   p  2v            a(i) = b(d(i)) + scalar * c(e(i))
54    1   p  2v          enddo
```

インダイレクトアクセス（配列b, c）に対するプリフェッチが生成された

R-CCSチューニングチュートリアルより

ラインアクセスの有効利用

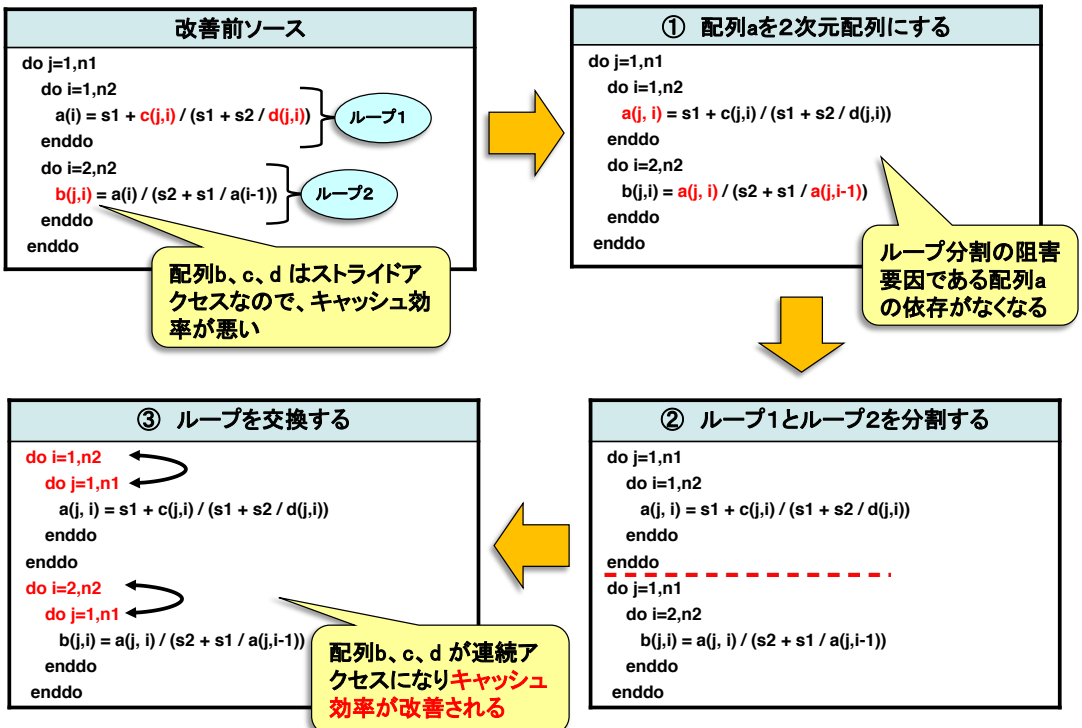
- リストアクセスでx,y,z,u,v,wの座標をアクセスするような場合配列マージによりキャッシュラインのアクセス効率を高められる場合がある.



R-CCSチューニングチュートリアルより

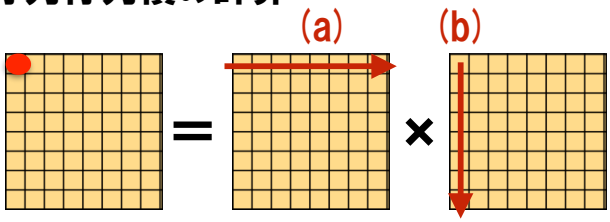
ラインアクセスの有効利用

- ストライドアクセス配列を連続アクセス化することによりラインアクセスの有効利用を図る.



キャッシュの有効利用

行列行列積の計算



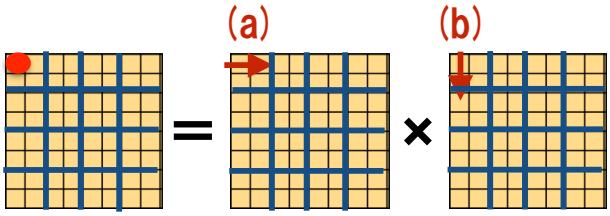
ブロッキング

(*)1要素が８Byteとする

B/F値
=移動量(Byte)/演算量(Flop)
$=8*2N^2/2N^3$
$=8/N$

原理的にはNが大きい程小さな値

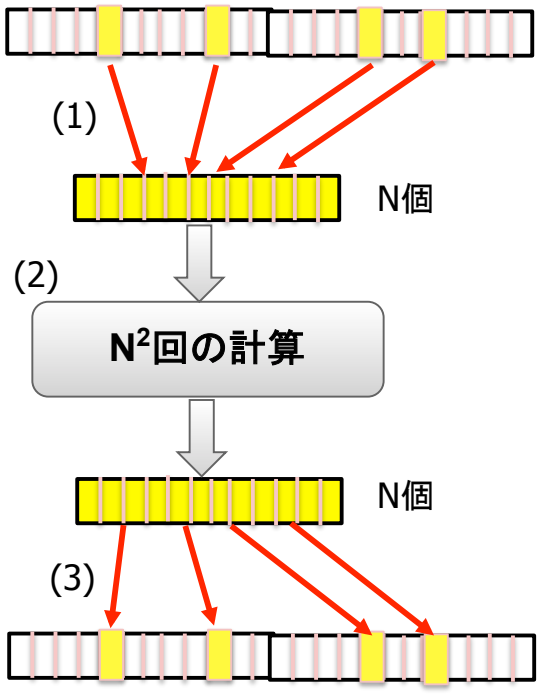
- **現実のアプリケーションではNがある程度の大きさになるとメモリ配置的には (a) はキャッシュに乗っても (b) は乗らない事となる**



- **そこで行列を小行列にブロック分割し (a) も (b) もキャッシュに乗るようにしてキャッシュ上のデータだけで計算するようにする事で性能向上を実現する.**

キャッシュの有効利用

ブロッキング

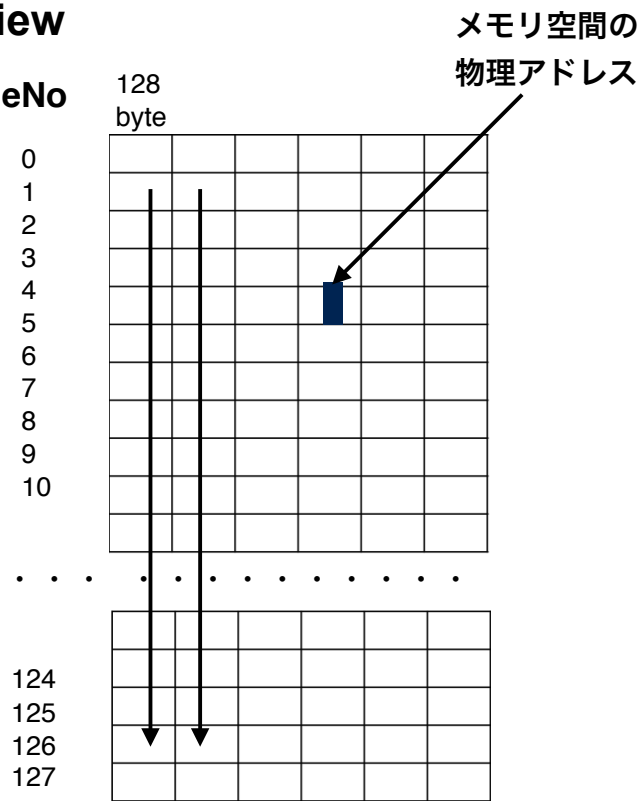


- **不連続データの並び替えによるブロッキング**
- **(1) N個の不連続データをブロッキングしながら連続領域にコピー**
- **(2) N個のデータを使用してN2回の計算を実施**
- **(3) N個の計算結果を不連続領域にコピー**

- **一般的に (1) (3) のコピーはNのオーダーの処理であるためN2オーダーの計算時間に比べ処理時間は小さい.**

キャッシュの有効利用

L1キャッシュの動作Overview

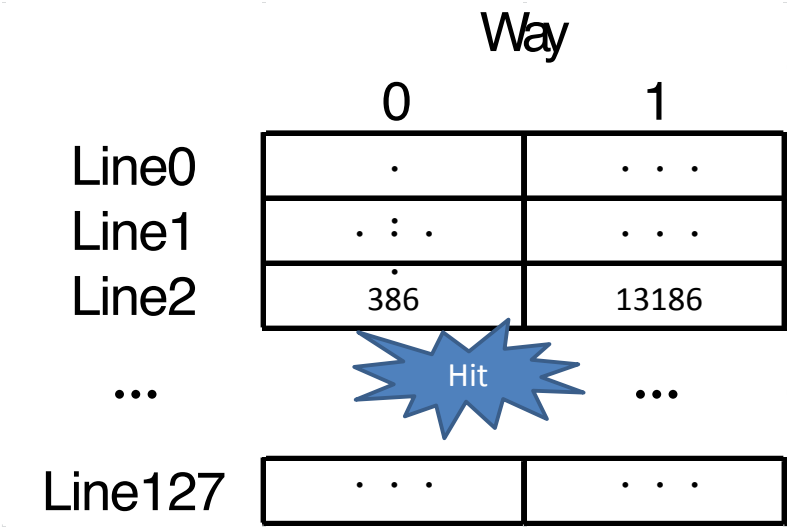


- メモリ空間の物理アドレスには各々LineNoが割り当てられる.
- そのLineNoがキャッシュのアドレスに使用される.

- INDEX= [ADDR/128]
- LineNo=MOD [INDEX/128]

キャッシュの有効利用

L1キャッシュの動作Overview [Hitの場合]

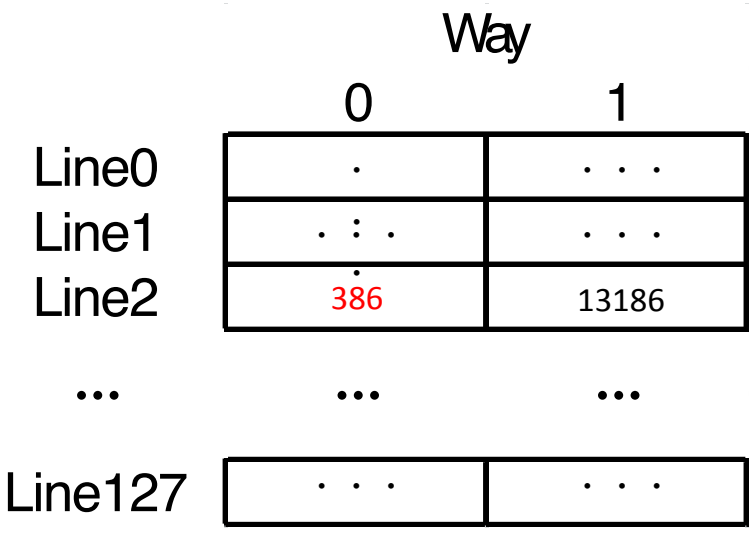


32KB,1ライン128Byte, 2Wayの例

1. アドレス49500の値を要求
2. 要求アドレスをラインのインデックスに変換
   ADDR=49500
   ↓
   INDEX=[ADDR/128]=386
3. インデックスをライン番号に変換LINE=MOD(INDEX,128)=2
4. Line2のWay0かWay1に要求アドレス(index)のデータが入っていたらHit

キャッシュの有効利用

L1キャッシュの動作Overview [Missの場合(1)]



32KB,1ライン128Byte, 2Wayの例

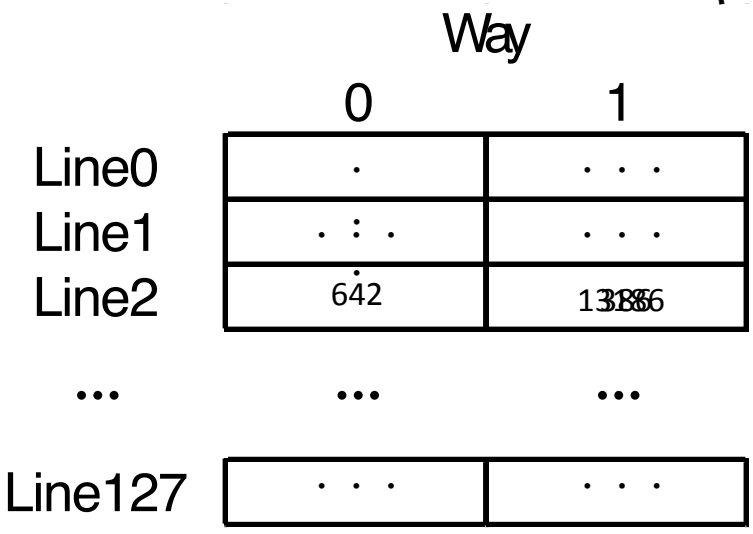
1. アドレス49500の値を要求
2. 要求アドレスをラインのインデックスに変換
   ADDR=49500
   ↓
   INDEX=[ADDR/128]=386
3. インデックスをライン番号に変換LINE=MOD(INDEX,128)=2
4. 入っていなければ次のレベル(L2やメモリ)からデータを持ってくる

Wayに空きがあればそこに入れる

キャッシュの有効利用

L1キャッシュの動作Overview [Missの場合(2)] **キャッシュ競合**

キャッシュスラッシング



32KB,1ライン128Byte, 2Wayの例

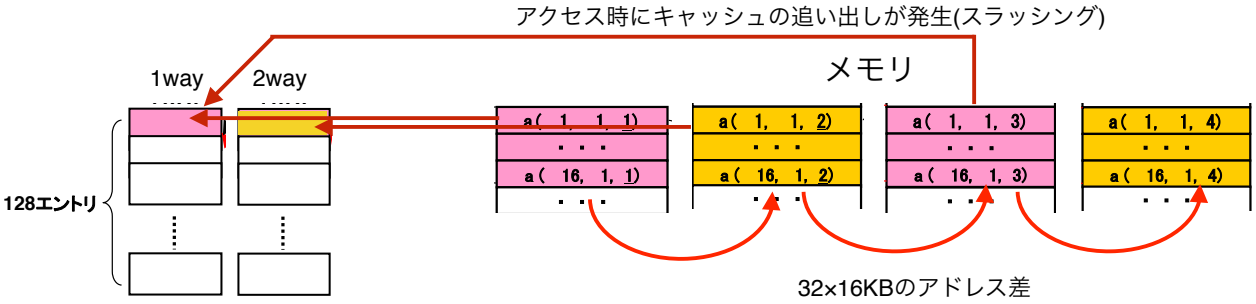
※ここには書かれていないが，どちらのWayがより古いか？を表すフラグがある

1. アドレス49500の値を要求
2. 要求アドレスをラインのインデックスに変換
   ADDR=49500
   ↓
   INDEX=[ADDR/128]=386
3. インデックスをライン番号に変換LINE=MOD(INDEX,128)=2
4. 入っていなければ次のレベル(L2やメモリ)からデータを持ってくる

Wayに空きがなければ一番使われてないものをどかして，そこに入れる

キャッシュの有効利用

スラッシング



L1Dキャッシュ
128バイト×128エントリ×2way=32Kバイト

256*256*8B/1024=512KB=32*16KB

ソース例

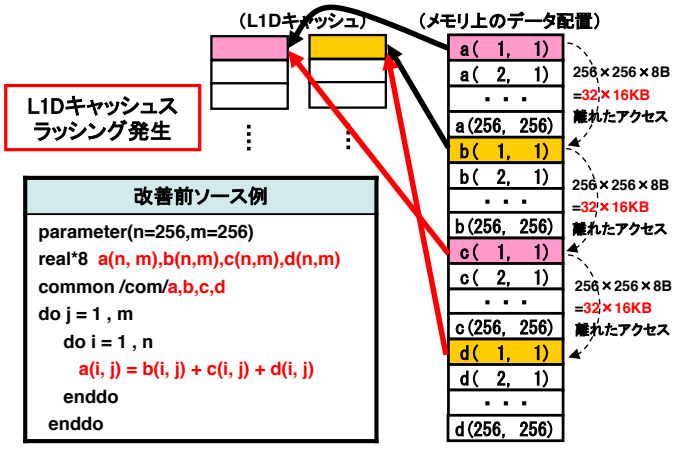
```
subroutine sub(a, n, m) ※n=256, m=256
real*8  a(n, m, 4)
  do j = 1 , m
    do i = 1 , n
      a(i, j, 4) = a(i, j, 1) + a(i, j, 2) + a(i, j, 3)
    enddo
  enddo
End
```

今回の例の場合、a(1,1,1)、a(1,1,2)、a(1,1,3)、a(1,1,4)はそれぞれ32×16KBずつ離れている(16KB境界にある)ため4つが同じインデックスに割り当てられる。そのため1つ目、2つ目のデータが3つ目、4つ目のデータに上書きされる。

キャッシュの有効利用

スラッシングの除去（配列マージ）

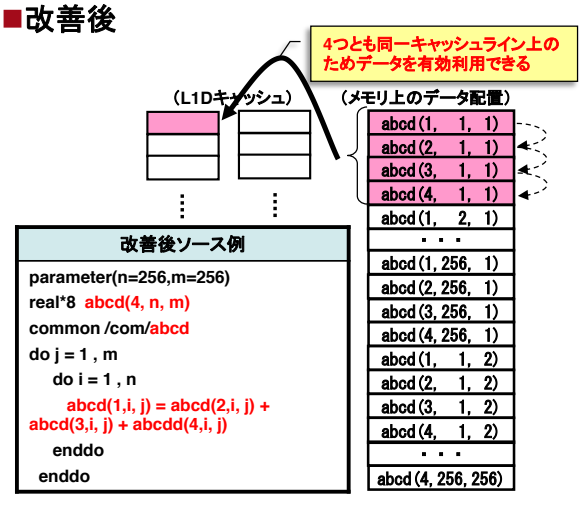
■改善前



改善前ソース例

```
parameter(n=256,m=256)
real*8  a(n, m),b(n,m),c(n,m),d(n,m)
common /com/a,b,c,d
do j = 1 , m
   do i = 1 , n
     a(i, j) = b(i, j) + c(i, j) + d(i, j)
   enddo
 enddo
```

■改善後



改善後ソース例

```
parameter(n=256,m=256)
real*8  abcd(4, n, m)
common /com/abcd
do j = 1 , m
   do i = 1 , n
     abcd(1,i, j) = abcd(2,i, j) +
abcd(3,i, j) + abcdd(4,i, j)
   enddo
 enddo
```

キャッシュへの格納　キャッシュへの格納(競合)　メモリへのアクセス順番

R-CCSチューニングチュートリアルより

キャッシュの有効利用

スラッシングの除去（ループ分割）

- ループ内でアクセスされる配列数が多いとスラッシングが起きやすい.
- ループ分割する事でそれぞれのループ内の配列数を削減する事が可能となる.
- 配列数が削減されることによりスラッシングの発生を抑えられる場合がある.

■ ループ分割

```
do i=1,n                    do i=1,n
  … = a(i) + b(i)             … = a(i) + b(i)
  … = c(i) + d(i)     ⇒     enddo
enddo                       do i=1,n
                              … = c(i) + d(i)
                            enddo
```


キャッシュの有効利用

スラッシングの除去（パディング）

- L1Dキャッシュへの配列の配置の状況を変更する事でスラッシングの解消を目指す.
- そのためにcommon配列の宣言部にダミーの配列を配置することでスラッシングが解消する場合がある.
- また配列宣言時にダミー領域を設けることでスラッシングが解消する場合がある.

■ パディング

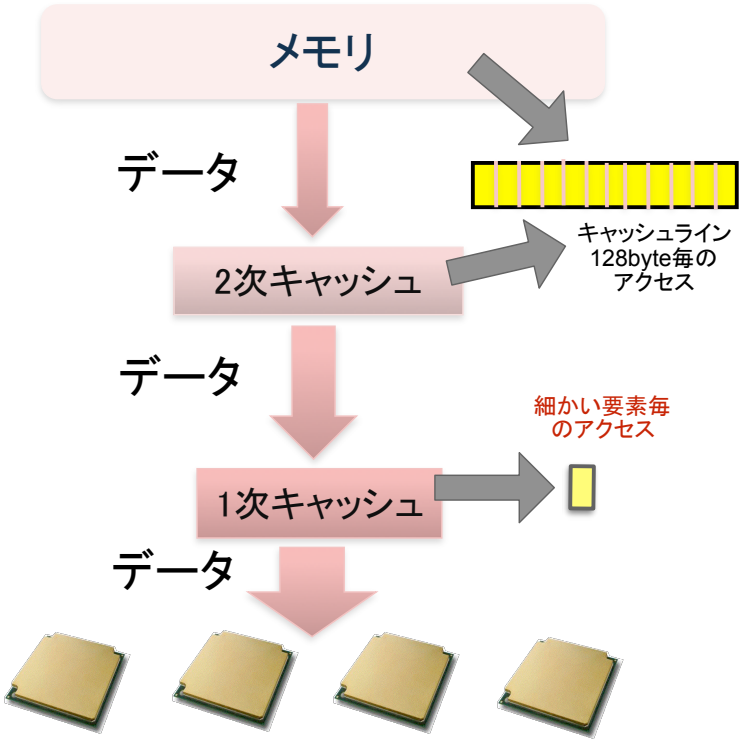
```
common //a(n),b(n)          common //a(n),p(64),b(n)
do i=1,n                    do i=1,n
  … = a(i) + b(i)     ⇒       … = a(i) + b(i)
enddo                       enddo
```

改善後ソース例

```
parameter(n=257,m=256)
real*8  a(n,m,4),b(n,m,4),c(n,m,4),d(n,m,4)
common /com/a
do j = 1 , m
   do i = 1 , n
     a(i, j, 1) = a(i, j, 2) + a(i, j, 3) + a(i, j, 4)
   enddo
 enddo
```

キャッシュの有効利用

リオーダリング



- メモリと2次キャッシュはキャッシュライン128バイト単位のアクセスとなる.
- 1次キャッシュは１要素8バイト単位のアクセスとなる.
- 以下のような疎行列とベクトルの積があるときベクトルはとびとびにアクセスされる.
- このベクトルがメモリや2次キャッシュにある場合アクセス効率が非常に悪化する.
- 1次キャッシュは，アクセスされる単位が細かいため，この悪化が緩和される.
- ベクトルを1次キャッシュへ置くための並び替えが有効になる.

```
do i=1,n
  buf = 0
  do j=1,l(i)
    k = k+1
    buf = buf + a(k)*v(L(k))
  x(i) = buf
```


効率の良いスケジューリング・演算器の有効利用

カラーリング

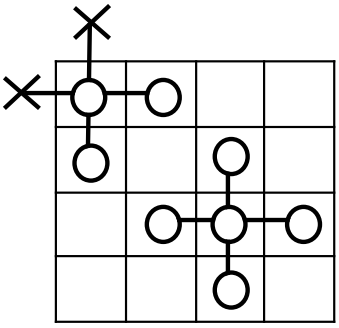
- ソフトウェアパイプラインニングやsimd等のスケジューリングはループに対して行われる.
- ループのインデックスについて依存関係があるとソフトウェアパイプラインニングやsimd化はできなくなる.
- 例えばガウス・ザイデル法の処理等がその例である.
- その場合は「CG法前処理のスレッド並列化」で説明したred-blackスィープへの書換えで依存関係を除去することが効果的である.
- red-blackは2色のカラーリングであるが2色以上のカラーリングも同様な効果がある.

効率の良いスケジューリング・演算器の有効利用

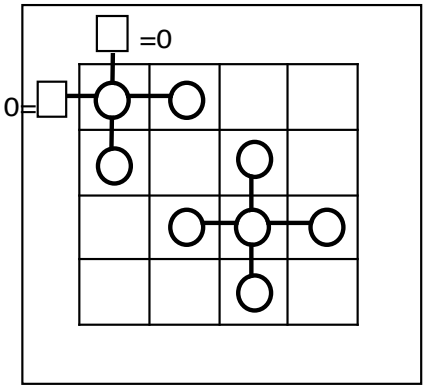
IF文の除去

- IF文も-Ksimd=2オプションや!ocl simdディレクティブを使う事でsimd化やソフトウェアパイプラインニングを行うことが出来る.
- しかしIF文が複雑になった場合は以下のような方法でのIF文の除去が有効な場合がある.

内点と縁の点で参照点の数が違うため計算式が違う. そのため内点と縁の点を区別するIF文が存在する.



計算領域の外側にダミー領域を設定しその中に0を格納する. こうすることにより内点と縁の点が同じ式となりIF文は必要でなくなる.

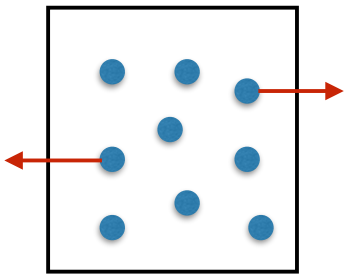


ダミー領域

効率の良いスケジューリング・演算器の有効利用

インデックス計算の除去

- 例えば粒子を使ったPIC法のアプリケーションの場合, 粒子がどのメッシュに存在するかをインデックスで計算するような処理が頻発する.
- このような複雑なインデックス計算を使用しているとソフトウェアパイプラインニングやsimd化等のスケジューリングがうまく行かない場合がある.
- このような場合には以下のような方法でのインデックス計算の除去が有効な場合がある.



1. 粒子が隣のメッシュに動く可能性がある.
2. そのため全ての粒子についてどのメッシュにいるかのインデックス計算がある.
3. しかし隣のメッシュに動く粒子は少数.
4. まず全粒子は元のメッシュに留まるとしてインデックス計算を除去して計算する.
5. その後隣のメッシュに移動した粒子について再計算する.

do i=1,N

インデックス
なしの計算

do i=1,移動した粒子数

インデックス
ありの計算

効率の良いスケジューリング・演算器の有効利用

ループ分割

- ループ内の処理が複雑すぎるとソフトウェアパイプライニングやsimd化が効かない場合がある.
- ループ分割する事でループ内の処理が簡略化されスケジューリングがうまくいくようになる場合がある.

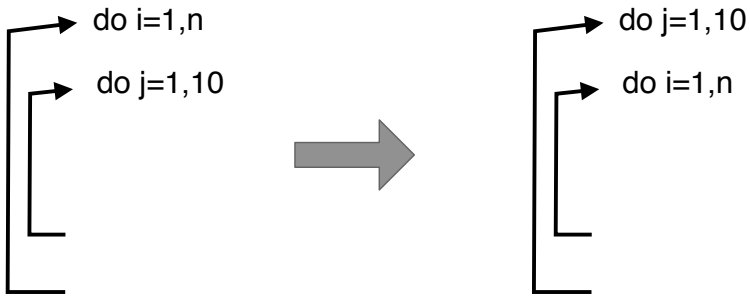
■ ループ分割

```
do i=1,n                    do i=1,n
  … = a(i) + b(i)             … = a(i) + b(i)
  … = c(i) + d(i)     ⇒     enddo
enddo                       do i=1,n
                              … = c(i) + d(i)
                            enddo
```


効率の良いスケジューリング・演算器の有効利用

ループ交換

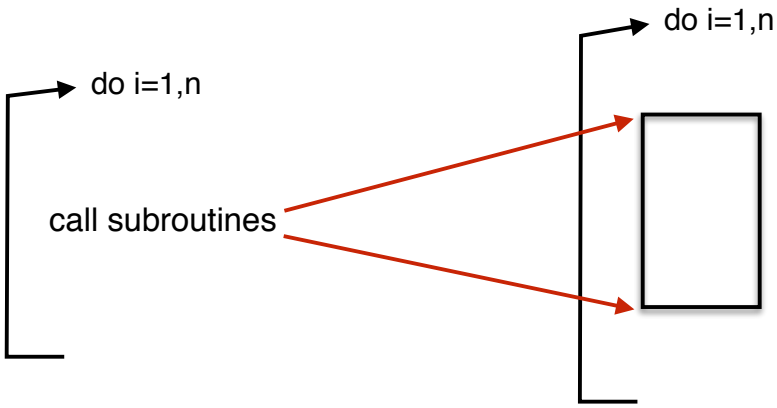
- ループの回転数が少ないとソフトウェアパイプラインニングが効かない場合がある.
- ループ交換し長いループを最内にする事でソフトウェアパイプラインニングがうまくいくようになる場合がある.



効率の良いスケジューリング・演算器の有効利用

関数・サブルーチン展開

- ループ内に関数やサブルーチン呼出しがあるとスケジューリングが進まない.
- その場合はコンパイラオプションにより関数やサブルーチンをループ内に展開する.
- ハンドチューニングで展開した方がうまく行く場合もある.



まとめ

- スレッド並列化
- CPU単体性能を上げるための5つの要素
- 要求B/F値と5つの要素の関係
- 性能予測手法(要求B/F値が高い場合)
- 具体的テクニック