

H28年度 並列コンピュータ 高速化技法の基礎

**2016年 9月9日
大阪大学サイバーメディアセンター
日本電気株式会社**

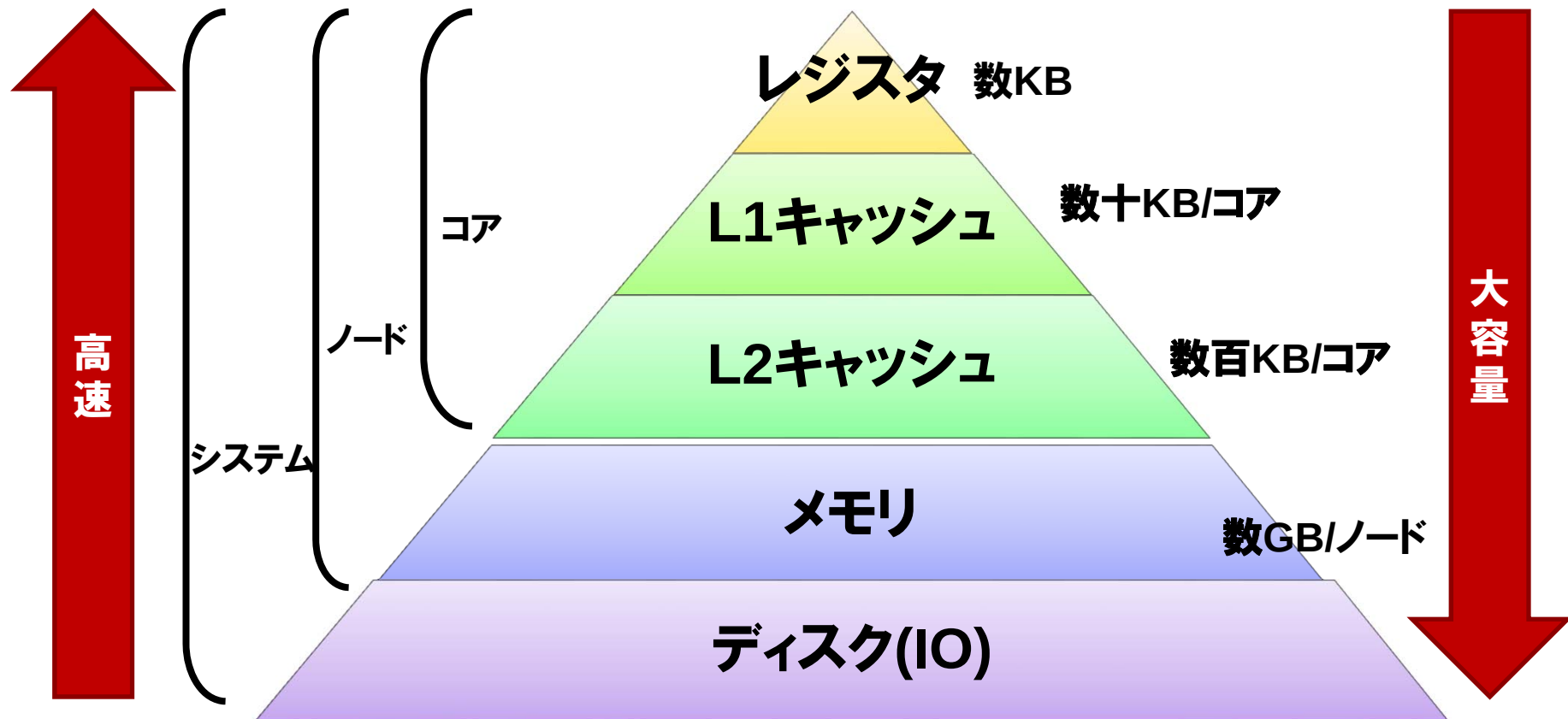
**本資料は、東北大学サイバーサイエンスセンターとNECの共同により作成され、大阪大学サイバーメディアセンターの環境で実行確認を行い、修正を加えたものです。
無断転載等は、ご遠慮下さい。**

1. 階層構造とデータアクセス

HCC/VCCのキャッシュサイズ

L1キャッシュ	32KB
L2キャッシュ	256KB

■ 計算機における階層構造とデータアクセスの特徴



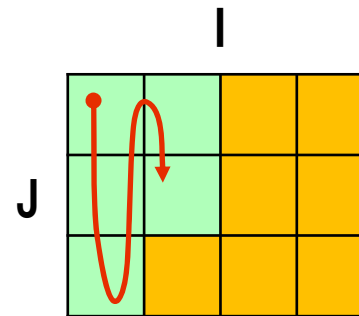
1. FortranとC言語の違い

多次元配列のメモリ配置

- FortranとC言語では多次元配列のメモリ上の連続アクセス方向が違います。

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

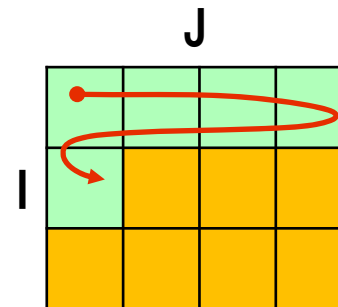
Fortran



多次元配列の左側のインデックスがメモリ上で連続に配置されます

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

C



多次元配列の右側のインデックスがメモリ上で連続に配置されます

※本講習会ではループの繰り返し回数を「ループ長」とも呼びます。

2. コンパイルコマンド

PCクラスタシステムのコンパイルコマンド

- Intel Compilerを利用してコンパイルを行います.

言語	コマンド
Fortran	ifort
C	icc
C++	icpc

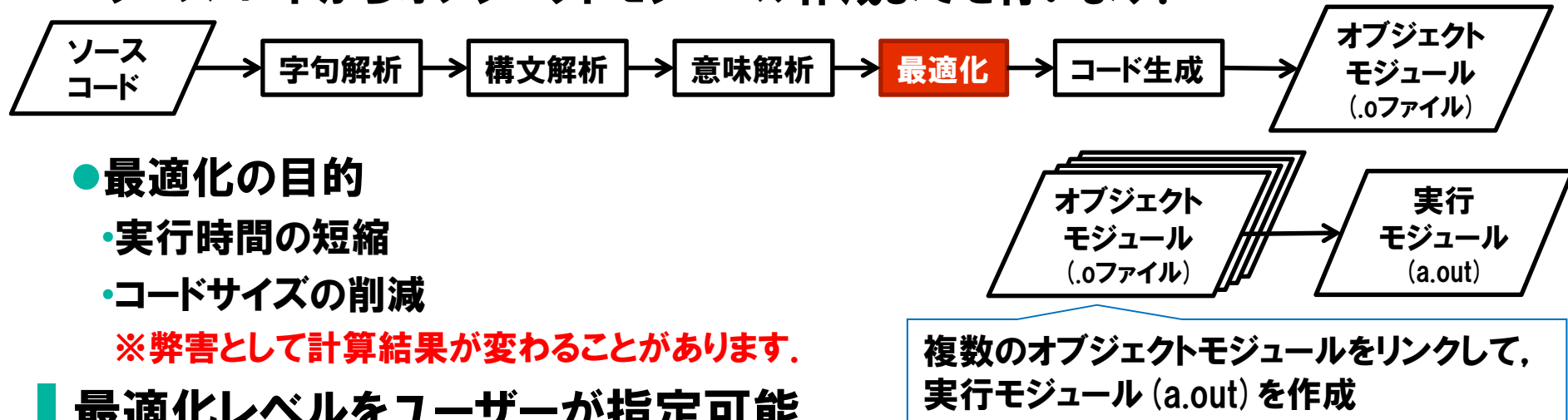
基本的なコンパイラオプション

オプション	概 要
-o <i>file名</i>	出力ファイル名の指定
-c	コンパイルのみ実施 (リンクしない)
-V	コンパイラバージョンの表示

2. コンパイラの最適化について

コンパイラの働き

- ソースコードからオブジェクトモジュール作成までを行います。



最適化レベルをユーザーが指定可能

- コンパイラオプション, 指示行

極端な最適化の例

```
do i=1, n
  y(i) = x(i) + y(i) * . . . .
enddo
stop
end
```

どんなにnを大きくしても、演算部分を増やしても一瞬で終わる。

└ コンパイラによっては、計算後のyを参照しないので、不要な演算としてループを削除するため。

2. コンパイラオプション

最適化コンパイラオプション

	オプション	デフォルト	概 要
最適化レベル	-O0	-O2	全ての最適化を無効にします
	-O1		-O2のうちコードサイズを増大させる最適化は行わない ベクトル化を行わない
	-O2		ベクトル化の有効化, 組み込み関数のインライン展開 不要な変数・関数などの削除 ループアンロールなどのコードの最適化
	-O3		プリフェッチ, ループ変換, パディング等を行う (-O2よりも実行速度が遅くなる場合がある) (コンパイル時間が長くなる場合がある)
	-fast	なし	最大限に最適化します (使用は推奨されません) 「-xHost -O3 -ipo -no-prec-div -static」と同等です
	-xHost	なし	コンパイラが動作するCPUが使用可能な最上位の命令セットを生成します フロントエンドでは-xSSE4.2が有効になります (使用は推奨されません)
浮動小数点 関連	-[no-] ftz	なし (-O1以上では有効)	アンダーフローした結果(デノーマル)を0として扱う
	-[no-] prec-div	-prec-div	-noprec-divの指定で除算を乗算へ変更(精度が落ちる場合があります)
	-[no-] prec-sqrt	-no-prec-sqrt	高速で精度が少し低い平方根計算を使用
手続き間 最適化	-ip	-O2以上で-ip有効 (-O1以下は-no-ip)	手続き(サブルーチン, 関数)間にまたがる最適化を実施 インライン展開も含む
	-ipo	-no-ipo	複数のファイルに跨る手続き(サブルーチン, 関数)間にまたがる最適化を実施 インライン展開も含む

2. コンパイラオプション

最適化コンパイラオプション (続き)

	オプション	デフォルト	概 要
プロファイル	-prof-gen	-no-prof-gen	プロファイル用のオブジェクトファイルを生成します
	-prof-use	-no-prof-use	最適化にプロファイル情報を使用するようにします
並列化	-parallel		自動並列化を有効にし、マルチスレッドコードを生成します
	-openmp		OpenMPによる並列化を有効にし、マルチスレッドコードを生成します
コンパイラ メッセージ	-opt-report [n]	n=2	nに数値を指定する事で、コンパイラが出力する最適化レポートのレベルを指定します n=0,1,2,3で数値が大きくなるほど出力される情報量が多くなります
	-vec-report [n]	n=1	ベクトル化時の診断メッセージレベルを指定します n=0,1,2,3,4,5で指定し、3の情報量が最大となります
	-par-report [n]	n=1	自動並列化時の診断メッセージレベルを指定します n=0,1,2,3で数値が大きくなるほど出力される情報量が多くなります
	-openmp-report [n]	n=1	OpenMP並列化時の診断メッセージレベルを指定します n=0,1,2で数値が大きくなるほど出力される情報量が多くなります

※コンパイラメッセージは、「ソースファイル名+”.optrpt”」として出力されます。

2. コンパイラオプション

デバッグ, その他のコンパイラオプション

	オプション	デフォルト	概 要
デバッグ	-g		デバッグ情報を生成します (最適化オプションを指定しないと -O0となります)
	-traceback		実行時エラーの際, トレースバック情報を出力させます
	-check uninit		初期化されていない変数の実行時チェックを有効にします (実行速度が遅くなります)
	-check bounds		配列添字と次元境界の実行時チェックを有効にします (実行速度が遅くなります)
その他	-fpp		Fortranプリプロセッサを実行します

ここにあげたオプションは主要なものだけです。
詳細なオプションやその説明についてはコンパイラマニュアルを
参照してください。

2. おすすめコンパイラオプションと注意点

■ プログラムを初めて実行する場合

- 既定値レベルの最適化 (-O2)

まずは、最適化に関しては何も指定しないで既定値でコンパイル・実行を行ってください。

■ 正常終了し、高速化を行う場合

- -O3 -ftz
- さらに -noprec-div の追加 (除算の精度が落ちる場合があります)
- プロファイル (prof-gen/prof-use) の利用

演算結果が -O2と変わらないか、変わっても許容範囲か確認してください。

■ デバッグを行う場合

- -g デバッグ情報を生成 (デバッグ時の基本)
- -traceback (トレースバック情報の出力)
- -check uninit (実行時に初期化漏れのチェック・実行時間が遅くなります。)
- -check bounds (実行時に配列外参照のチェック・実行時間が遅くなります。)

■ 注意点

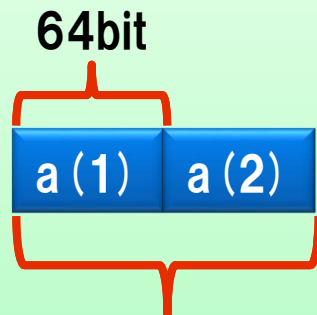
- 最適化レベルを高くするとコンパイル時間が長くなる場合があります。
- 最適化レベルによっては 計算結果が変わる場合があります。

2. PCクラスタシステムにおけるベクトル化

ベクトル化について

- スーパーコンピュータ SX-ACEと基本的な概念は同じです。
- 大きな違いはベクトル化される単位になります。
- SIMD (Single Instruction Multiple Data) : 1つの命令で複数データを処理

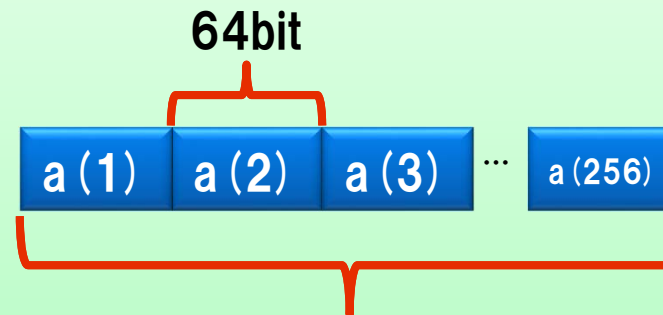
スカルマシン



128bit単位で
データ使用

2変数をベクトルとして扱い、
データのロード・ストア・演算を行う。
(倍精度浮動小数点の場合)

ベクトルマシン



256要素単位
でデータ使用

256変数をベクトルとして扱い、
データのロード・ストア・演算を行う。
(倍精度浮動小数点の場合)

2. PCクラスタシステムにおけるベクトル演算

■ ベクトル演算はMMX/SSEという命令セットとして定義されています。

- MMX 整数演算のベクトル化
- SSE 単精度浮動小数点演算のベクトル化
- SSE2 倍精度浮動小数点演算のベクトル化
- SSE3 データの詰め替え命令や複素数高速化命令の追加
- SSE4 積和命令の追加など
- AVX Sandy Bridgeから対応 256bit単位に拡張
- AVX2 Haswellから対応 浮動小数点演算のFMAをサポート

■ ベクトル化の特長

- 基本的にコンパイラが自動ベクトル化を行います。
- ベクトル化されるのは最も内側のループになります。
- 128bitの単位でベクトル化するため、単精度なら4要素、倍精度なら2要素単位。
- 1クロックで128bitのベクトル乗算・ベクトル加算を一つずつ計算可能です。
- 演算命令に加えて、ロードやストアの命令もベクトル化されます。

3. 演習環境の説明

演習環境の説明

● ディレクトリ構成

```
Para/  
|-- practice_3_1 演習3 (コンパイラオプション )  
|-- practice_3_2 演習3 (ループアンロール)  
|-- practice_4   演習4 (ベクトル化)  
|-- exec.tune_1  チュートリアル1  
|-- exec.tune_2  チュートリアル2  
|-- exec.tune_3  チュートリアル3
```

practice/ディレクトリ内の構成

○○○.f	ソースコードファイル
comp.csh	コンパイルシェル
run.csh	実行シェル
answer/	解答例ディレクトリ

● 使用コード

行列積コード

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

※ exec.tuneディレクトリ内の構成は後ほど説明

3. 演習問題: コンパイラオプション

目的

- コンパイラオプションによるコンパイル時のメッセージと性能の違いを確認する。

手順

- 最適化オプションの変更によるコンパイルとメッセージの確認
- 実行（性能値の確認）

ディレクトリ

- practice_3_1

3. 演習問題: コンパイラオプション (コンパイル)

コンパイラオプション

- 最適化診断メッセージ (-opt-report=1)

※診断結果が「ソースファイル名+”.optrpt”」として出力されます。

- 最適化オプション (-O0, -O1, -O2, -O3)

最適化レベル	
高度	-O3
↑	-O2
↓	-O1
簡易	-O0

```
ifort -opt-report=1 -O0 -o tune0_o0.exe mat_tune0.f
```

```
ifort -opt-report=1 -O? -o tune0_o?.exe mat_tune0.f
```

コンパイルとメッセージの確認

```
% ./comp.csh
```

[診断メッセージが出力される]

3. 演習問題: コンパイラオプション (コンパイル, 実行)

メッセージの確認

- 00 : 最適化メッセージなし
- 01 : 最小限の最適化のメッセージが出力
- 02 : インライン展開やベクトル化のメッセージが出力
- 03 : -02より多くの最適化・ベクトル化のメッセージが出力

実行シェル

```
#!/bin/csh
#PBS -q H-single
#PBS -l cpunum_job=1, elapstim_req=0:10:00, memsz_job=1GB
#PBS -j o -N practic_3_1

cd $PBS_O_WORKDIR

time ./a.out
```

NQS II オプション

- q ジョブクラス名を指定
- l 使用CPU数, 経過時間, メモリ容量の申告
- j o 標準エラー出力を標準出力と
同じファイルへ出力する
- N ジョブ名を指定

実行時オプション

指定なし

ジョブ投入

```
% qsub ./run.csh
Request *****.hcc submitted to queue: H-single.
```

*****はジョブ番号

3. 演習問題: コンパイラオプション (実行結果)

性能値の確認

●結果ファイル practic_3_1.o****の確認

オプション	性能値
-00	0.323 (GFlops)
-01	1.964 (GFlops)
-02	2.776 (GFlops)
-03	9.394 (GFlops)

性能値はプログラムが
演算数と経過時間から算出している値

実行結果の出力例

```
----- opt level 0 -----  
matrix_size = 2048  
p_name      | user(sec) | mod    | check  
mat_tune0.f | 53.304    | 0      | 0.204  
0.323 (GFlops)  
:
```

-00と-03では**約30倍**の性能差となっています。

行列積コードはコンパイラによる最適化の効果が非常に大きいコード

●デフォルトコンパイラオプションは **-02**

-03を指定した場合、コンパイル時間が長くなる場合があります。

※ 注意

演習環境ではジョブ実行が専有で行われないため、
性能値がばらつく可能性があり、上記の値と大きく違う場合もあります。

3. 演習問題: ループアンロール

目的

- アンロールによる高速化を確認する.

手順

- ソースコード修正とコンパイル (-O1を使用)
- 実行 (性能値の確認)

ディレクトリ

- practice_3_2

【補足】ループアンロール

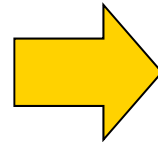
アンロールとは

- ループを複数段に書き下すチューニング内容になります。

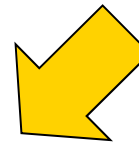
行列積コード

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

kを4段分
書き下す。



```
do j=1, n
  do k=1, n, 4
    do i=1, n
      a(i, j)=a(i, j)+b(i, k)*c(k, j)
      a(i, j)=a(i, j)+b(i, k+1)*c(k+1, j)
      a(i, j)=a(i, j)+b(i, k+2)*c(k+2, j)
      a(i, j)=a(i, j)+b(i, k+3)*c(k+3, j)
    enddo
  enddo
enddo
```



総和なので一度にまとめて計算できる。
□の部分のa(i,j)のストア・ロードを削除する。

行列積コード (アンロール)

```
do j=1, n
  do k=1, n, 4
    do i=1, n
      a(i, j)=a(i, j)+b(i, k)*c(k, j)
      &      ストア   ロード +b(i, k+1)*c(k+1, j)
      &      +b(i, k+2)*c(k+2, j)
      &      +b(i, k+3)*c(k+3, j)
    enddo
  enddo
enddo
```

配列aのロード・ストアが1/4になります。

ロード 4→1個
ストア 4→1個

3. 演習問題: ループアンロール (ソースコード修正)

ソースコード修正

●practice_3_2

行列積コード

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



行列積コード (アンロール)

```
do j=1, n
  do k=1, n, 4
    do i=1, n
      a(i, j)=a(i, j)+b(i, k) *c(k, j)
      &      ストア   ロード +b(i, k+1)*c(k+1, j)
      &      +b(i, k+2)*c(k+2, j)
      &      +b(i, k+3)*c(k+3, j)
    enddo
  enddo
enddo
```

※演習では余り処理を考慮しません.

配列aのロード・ストアが1/4になります.
メモリアクセスが減ることによる高速化を期待します.

コンパイル

```
% ./comp.csh
```

ジョブ投入

```
% qsub ./run.csh
Request *****.hcc submitted to queue: H-single.
```

*****はジョブ番号

3. 演習問題: ループアンロール (実行結果)

性能値の確認

●結果ファイル practic_3_2.o*****の確認

	性能値
アンロール前	1.964 (GFlops)
アンロール後	2.492 (GFlops)

性能値はプログラムが
演算数と経過時間から算出している値

実行結果の出力例

```
----- opt level 0 -----  
matrix_size = 2048  
p_name      | user(sec) | mod    | check  
mat_tune0.f | 53.304    | 0      | 0.204  
0.323 (GFlops)  
:
```

アンロールにより, 約**1.3倍**の性能向上がみられました.

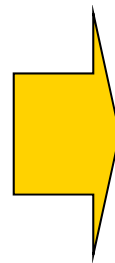
3. 補足: ループアンロール

■ 余り処理を考慮したソースコード修正の例

● practice_3_2

行列積コード

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



行列積コード

n1= mod(n, 4)

do j=1, n

do k=1, n1

do i=1, n

a(i, j)=a(i, j)+b(i, k) *c(k, j)

enddo

enddo

余りの部分を計算

do k=n1+1, n, 4

do i=1, n

a(i, j)=a(i, j)+b(i, k) *c(k, j)

& +b(i, k+1)*c(k+1, j)

& +b(i, k+2)*c(k+2, j)

& +b(i, k+3)*c(k+3, j)

enddo

enddo

enddo

4段アンロール部分

休憩

 **10分間休憩**
14:10まで

4. 演習問題: ベクトル化 (説明)

-vec-report=3 を指定した例

- 各ループについてベクトル化が行われたか、行われなかった場合はなぜベクトル化しなかったかが「ソースファイル名+”.optprt”」として出力されます。

```
$ ifort -vec-report=3 -O3 -o prac_4_org.exe practice_4_org.f
```

```
$ cat practice_4_org.optprt
```

※ 日本語で出力させた場合の結果になります。

```
LOOP BEGIN at practice_4_org.f (20, 7)
```

```
<Distributed chunk2>
```

*.optprtファイルが作成される

```
remark #15300: ループがベクトル化されました.
```

```
remark #15448: マスクなしアライン・ユニット・ストライド・ロード: 1
```

```
remark #15449: マスクなしアライン・ユニット・ストライド・ストア: 1
```

```
remark #15475: --- ベクトルループのコストサマリー開始 ---
```

```
remark #15476: スカラーループのコスト: 6
```

```
remark #15477: ベクトルループのコスト: 4,000
```

```
remark #15478: スピードアップの期待値: 2,140
```

```
remark #15479: 軽いベクトル操作: 3
```

```
remark #15480: 中オーバーヘッドのベクトル操作: 1
```

```
remark #15487: 型変換: 1
```

```
remark #15488: --- ベクトルループのコストサマリー終了 ---
```

ベクトル化された

```
LOOP END
```

依存関係があるためベクトル化できない

```
LOOP BEGIN at practice_4_org.f (41, 7)
```

```
remark #15344: ループ はベクトル化されませんでした: ベクトル依存関係がベクトル化 .
```

最初の依存関係を以下に示します. 詳細については, レベル 5 のレポートを使用してください. を妨げています.

```
remark #15346: ベクトル依存関係: FLOW の依存関係が idata2 行 42 と idata2 行 42 の間に仮定されました.
```

```
LOOP END
```

ベクトル化不可の依存関係がある部分が出力される

```
LOOP BEGIN at practice_4_org.f (54, 7)
```

```
remark #15344: ループ はベクトル化されませんでした: ベクトル依存関係がベクトル化 . 最初の依存関係を以下に示します.
```

詳細については, レベル 5 のレポートを使用してください. を妨げています.

```
remark #15346: ベクトル依存関係: FLOW の依存関係が rdata3 行 55 と rdata3 行 55 の間に仮定されました.
```

```
LOOP END
```

4. 演習問題: ベクトル化 (説明)

ベクトル化を阻害する要因

- データの依存関係
- 関数・サブルーチンの呼び出し
- 構造体へのアクセス

構造体は要素がメモリ上に飛び飛びに配置され、ベクトル化が困難となります。

- 条件分岐

複数の条件分岐や分岐内容で処理が変わる場合は、ベクトル化が困難になります。

- ループの終了条件が不明

while文など

- ポインタアクセス

データの依存関係の確認

```
do i=2, 1000
  a(i) = a(i-1)
enddo
```

```
1: i=2の場合 a(1)が必要 a(2)を更新
2: i=3の場合 a(2)が必要 a(3)を更新
      :
k: i=kの場合 a(k-1)が必要 a(k)を更新
```



4. 演習問題: 最適化と指示行 (説明)

最適化 (ベクトル化)

- 最適化としてコンパイラが行います。

ただし、ベクトル化できると判断してもループサイズが小さいなど、ベクトル化の効果が見込まれないときはベクトル化を行いません。

指示行

- 指示行とは

- コードに埋め込み、コンパイラに最適化のヒントを与えます。
- あるいは強制的な最適化を行わせます。

- 指示行形式

- FORTRAN

!DEC\$ (CDEC\$) [指示行オプション] または !DIR\$ (CDIR\$) [指示行オプション]

- C/C++

#pragma [指示行オプション]

- ベクトル化に関する指示行

- !DEC\$ IVDEP 「依存関係がない」というヒントをコンパイラに与える
ベクトル化の効果がないと判断した場合はベクトル化は行わない
- !DEC\$ SIMD ベクトル化を行う

※ ベクトル化に関する指示行は、本来ベクトル化できない依存関係があるループもベクトル化できる (結果が変わる) ため、ユーザ責任で追加する。

4. 演習問題: ベクトル化

目的

- 依存関係がある場合の指示行によるベクトル化で、結果が変わってしまうことを確認する。

手順

- ソースコード修正 (指示行) とコンパイル (-O3と-vec-report=3を使用する)
- 実行と結果が変わることの確認

ディレクトリ

- practice_4

4. 演習問題: ベクトル化 (指示行追加)

コンパイラオプション

- コンパイラオプション `-O3 -vec-report=3`

指示行

- ループの直前に指示行「SIMD」を追加します.

```
!DEC$ SIMD
do ii = 1, len
  idata2(ii) = idata2(idata1(ii))
enddo
!DEC$ SIMD
do ii = 1, len
  rdata3(ii) = rdata3(idata1(ii))
enddo
```

指示行「IVDEP」では
「依存関係がない」ヒントを与えるが、
この演習問題ではベクトル化の効果が小さい
と判断されるためベクトル化は行われません.

コンパイル

- コンパイル時のベクトル化メッセージを確認します.

```
% ./comp.csh
```

ジョブ投入

```
% qsub ./run.csh
Request *****.hcc submitted to queue: H-single.
```

*****はジョブ番号

4. 演習問題: ベクトル化 (結果確認)

結果確認

- 依存関係があるため、ベクトル化すると結果が変わってしまいます。

- 逐次 (スカラ) の実行結果

idata2・rdata3の結果は同じになります。以下は、idata2の場合で解説します。

```
do ii = 1, len
  idata2(ii) = idata2(idata1(ii))
enddo
```

ii		1	2	3	4	5	6	7	8	9	10
idata1()		3	9	8	6	2	5	2	5	8	8
初期値 idata2()		3	9	8	6	2	5	2	5	8	8
更新後	逐次 (正しい)	8	8	5	5	8	8	8	8	8	8
	idata2() int vec										
	rdata3() real*8 vec										

4. 演習問題: ベクトル化 (結果確認)

結果確認

- 依存関係があるため、ベクトル化すると結果が変わってしまいます。
 - integer (idata2) をベクトル化した場合の実行結果

!DEC\$ SIMD

```
do ii = 1, len  
  idata2(ii) = idata2(idata1(ii))  
enddo
```

ii		1	2	3	4	5	6	7	8	9	10
idata1()		3	9	8	6	2	5	2	5	8	8
初期値 idata2()		3	9	8	6	2	5	2	5	8	8
更新後	逐次(正しい)	8	8	5	5	8	8	8	8	8	8
	idata2() int vec	8	8	5	5	8	2	8	2	2	2
	rdata3() real*8 vec										

4. 演習問題: ベクトル化 (結果確認)

結果確認

- 依存関係があるため、ベクトル化すると結果が変わってしまいます。
 - real*8 (rdata3) をベクトル化した場合の実行結果

!DEC\$ SIMD

```
do ii = 1, len  
  rdata3(ii) = rdata3(idata1(ii))  
enddo
```

ii		1	2	3	4	5	6	7	8	9	10
idata1()		3	9	8	6	2	5	2	5	8	8
初期値 rdata3()		3	9	8	6	2	5	2	5	8	8
更新後	逐次(正しい)	8	8	5	5	8	8	8	8	8	8
	idata2() int vec	8	8	5	5	8	2	8	2	2	2
	rdata3() real*8 vec	8	8	5	5	8	2	8	8	8	8

4. 演習問題: ベクトル化 (結果確認)

結果確認

- 依存関係があるため、ベクトル化すると結果が変わってしまいます。

i i		1	2	3	4	5	6	7	8	9	10
idata1()		3	9	8	6	2	5	2	5	8	8
初期 data2/3()		3	9	8	6	2	5	2	5	8	8
更新後	逐次 (正しい)	8	8	5	5	8	8	8	8	8	8
	idata2() int vec	8	8	5	5	8	2	8	2	2	2
	rdata3() real*8 vec	8	8	5	5	8	2	8	8	8	8

i=6に着目

逐次: idata2 (data1 (6)) → idata2 (5) 直前 (i=5) で更新された結果を参照する (正しい)

int vec: idata2 (data1 (6)) → idata2 (5) i=5~8のまとまり (idata2 (2,5,2,5)) でロードした値を参照する (i=5で更新される idata2 (5) =8ではない)

real vec: rdata3 (data1 (6)) → rdata3 (5) i=5~6のまとまり (rdata3 (2,5)) でロードした値を参照する (i=5で更新される idata2 (5) =8ではない)

ベクトル化のまとまり (integerは4つ, real*8は2つ) でロードするため、依存部分は更新後の値を使わない。(間違った結果となる)

5. 演習問題 (チュートリアル形式) : 行列積チューニング

目的

- ソースコード修正による高速化をチュートリアル形式で学ぶ.

手順

- ここまでの演習問題と同じ行列積コードを用います.
- 各ディレクトリに元ソースコードを用意しており,
各自がソースコード修正・コンパイル・実行を行っていきます.
- 解答例 (ソースコード修正と実行結果) も別ディレクトリに用意してあるので,
各自が時間配分の中で参照してください.

ディレクトリ

- exec.tune_1 から exec.tune_3
- 解答例は各ディレクトリの ./answer 配下

5. 演習問題 (チュートリアル形式) : 行列積チューニング

チュートリアル1

●問題と着目点

- ループの並びによる配列のアクセスパターンと性能の関係をみる.
- 3重ループの並びを入れ替える.
元は jki パターン (practice_3_1と同じコードになります)
ループを並び替え, **kji** と **ijk** の2通りを試す.
- **-01**でコンパイル・実行する.
※-02以上ではコンパイラが最適化を行ってループを入れ替えるため,
本コードでは多重ループの並び替えによる性能差は表れません.

●結果

- 演習3 (practice_3_1ディレクトリ参照) は jki パターン.
-01の性能 1.964GFLOPS
- 最内ループをなるべく連続アクセスとするのがポイントになる.

5. 演習問題 (チュートリアル形式): 行列積チューニング

チュートリアル2

●問題と着目点

- **チュートリアル1の結果を改善するため, kループとjループのアンロールを行う.**
- kループのアンロールは, 演習3 (practice_3_2参照) の修正結果を用意済み.
mat_tune2_jk4i_org.f として用意してあります.
- kループのアンロールを -O2 でコンパイル・実行する.
コンパイルには comp_jk4i_org.csh を, 実行には run_jk4i_org.csh を使用してください.
- 次に, さらに外側のjループを4段アンロールする.
mat_tune2_j4k4i.f を編集してください.
- 修正後のkとjループのアンロールを -O2 でコンパイル・実行する.
コンパイルには comp_j4k4i.csh を, 実行には run_j4k4i.csh を使用してください.

●結果

- kループのみのアンロールは, 演習3 (practice_3_2) の性能 1.964GFLOPS が
-O2により 2.776GFLOPS に向上する.
- 外側のjループのアンロールにより, 更に2倍以上性能が向上する.

5. 演習問題 (チュートリアル形式) : 行列積チューニング

チュートリアル3

- 問題と着目点

- チュートリアル2の結果を改善するため、キャッシュブロッキングを行う。

キャッシュブロッキング

全体の領域を細かなブロックに再分割することで、近隣データの再利用性を高め、キャッシュヒット率を向上させる。

【補足】キャッシュブロッキング

メモリアクセスの確認

●iループとkループ時の配列アクセス

倍精度の配列

$2048 \times 8 / 1024 = 16\text{KB}$

$2048 \times 2048 \times 8 / 1024 / 1024 = 32\text{MB}$

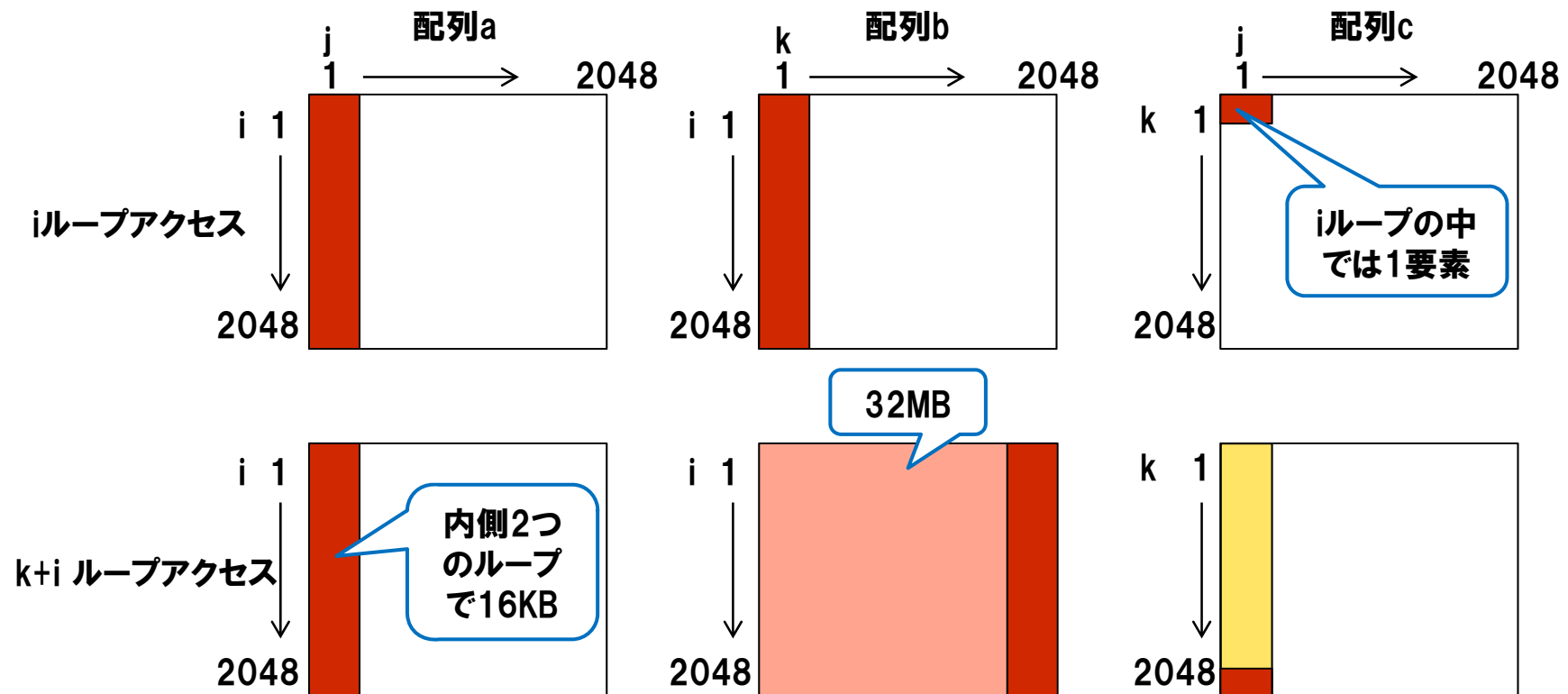
行列積コード

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

```
  do k=1, n
```

```
    do i=1, n
```

```
      a(i, j) = a(i, j) + b(i, k) * c(k, j)
```



【補足】キャッシュブロッキング

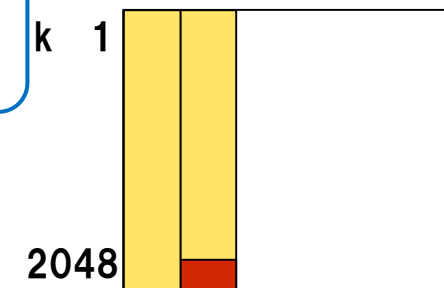
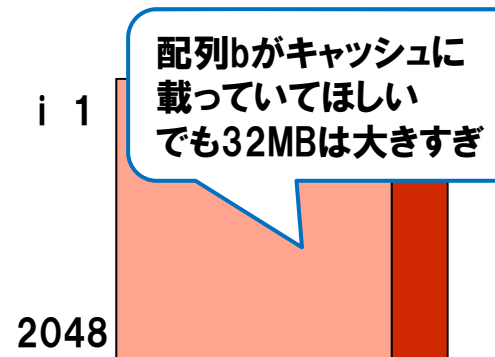
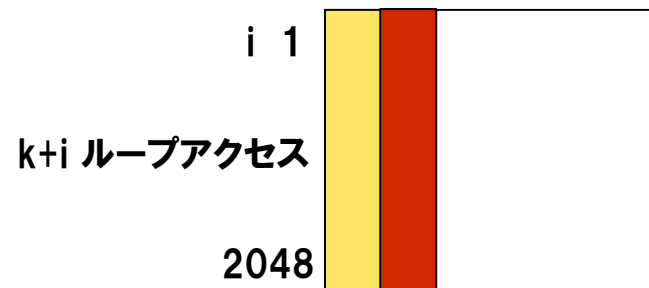
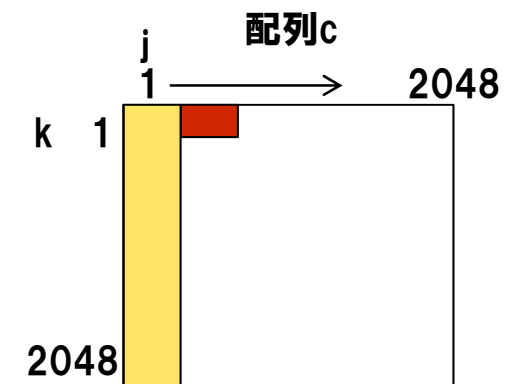
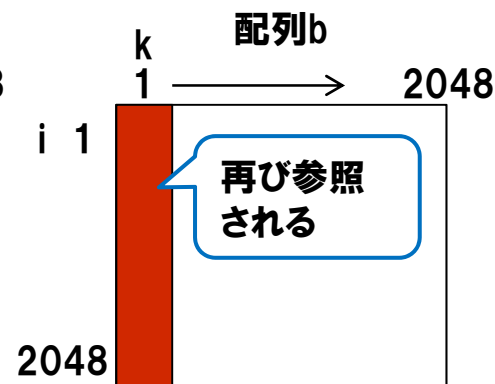
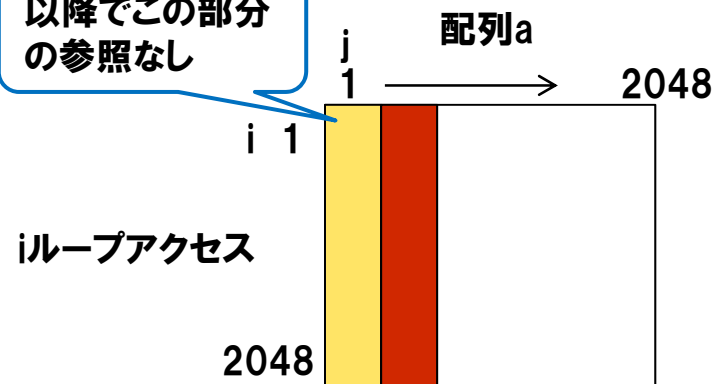
メモリアクセスの確認 (続き)

- 外側jが1つ進んだ時の配列アクセス

行列積コード

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

以降でこの部分
の参照なし



【補足】キャッシュブロッキング

iのブロッキングの場合

- iをブロッキングした時の
iループとkループ配列アクセス
- IBがブロッキングサイズ

行列積コード

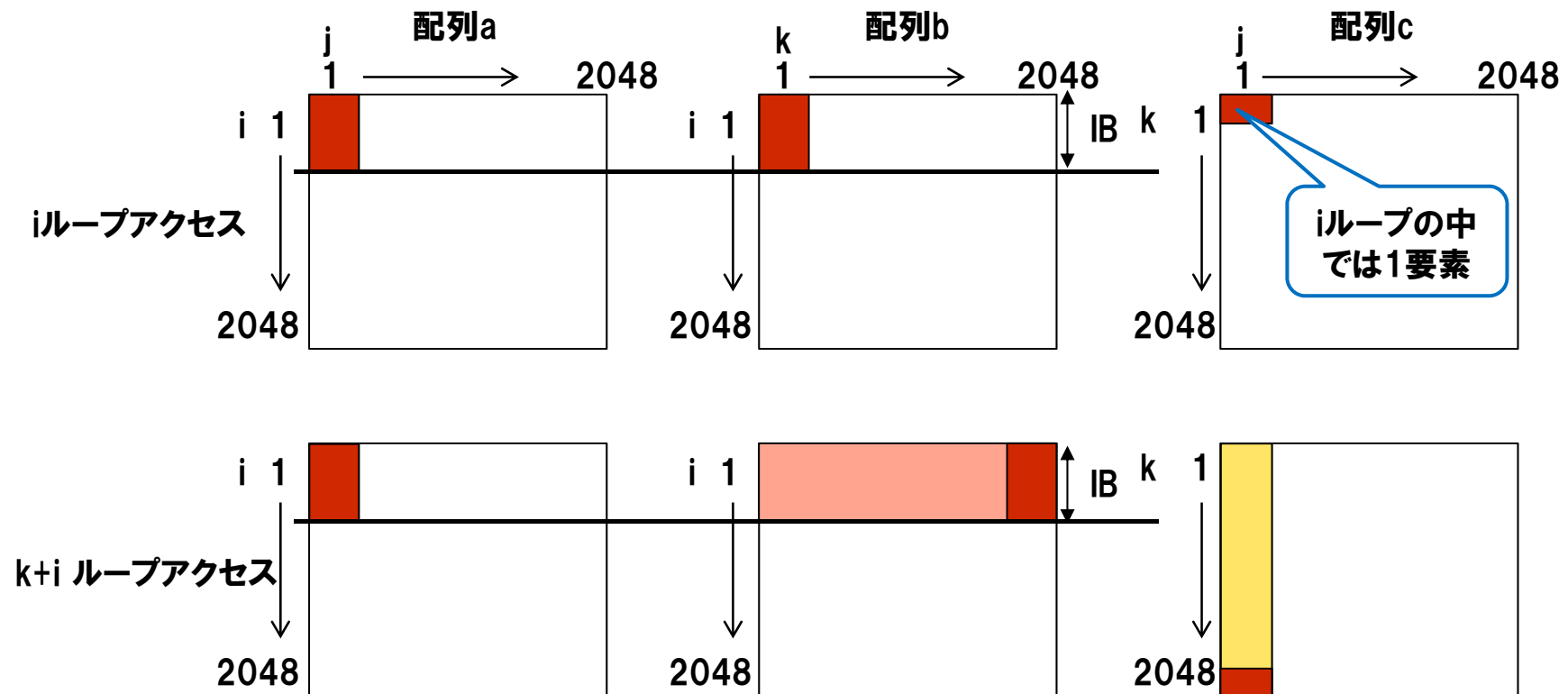
```
do ii=1, n, IB
```

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

```
do k=1, n
```

```
do i=ii, min(n, ii+IB-1)
```

```
a(i, j) = a(i, j) + b(i, k) * c(k, j)
```



【補足】キャッシュブロッキング

iのブロッキングの場合 (続き)

- iをブロッキングし、
外側jが1つ進んだ時の配列アクセス
- 配列bに着目し、 $2048 * IB$ のサイズが
キャッシュに収まればjループの繰り返しの
間はキャッシュヒット

行列積コード

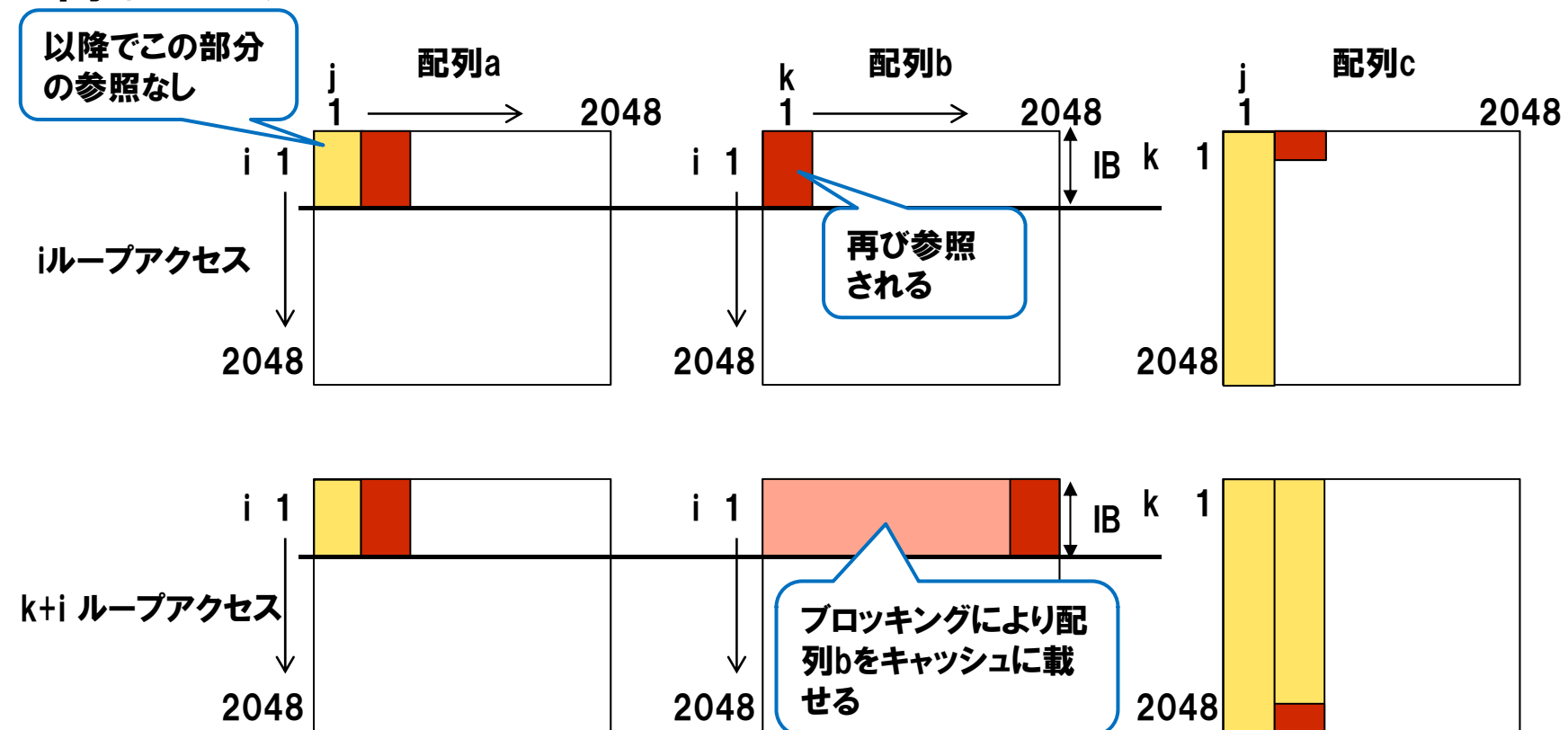
```
do ii=1, n, IB
```

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

```
do k=1, n
```

```
do i=ii, min(n, ii+IB-1)
```

```
a(i, j) = a(i, j) + b(i, k) * c(k, j)
```



【補足】キャッシュブロッキング

kのブロッキングの場合

- kをブロッキングした時のiループとkループ配列アクセス
- KBがブロッキングサイズ

行列積コード

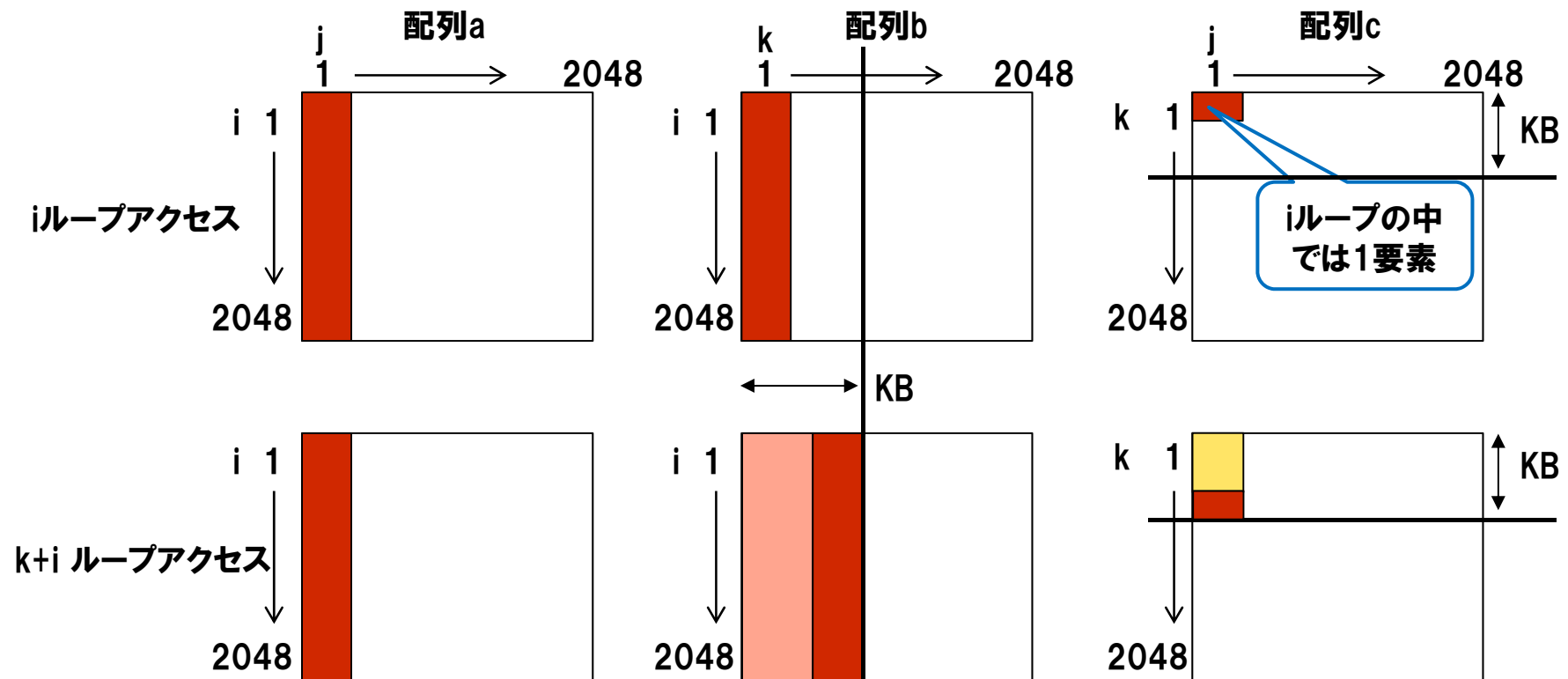
```
do kk=1, n, KB
```

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

```
do k=kk, min(n, kk+KB-1)
```

```
do i=1, n
```

```
a(i, j) = a(i, j) + b(i, k) * c(k, j)
```



【補足】キャッシュブロッキング

kのブロッキングの場合 (続き)

- kをブロッキングし、
外側jが1つ進んだ時の配列アクセス
- 配列bに着目し、2048*KBのサイズが
キャッシュに収まればjループの繰り返しの
間はキャッシュヒット

行列積コード

```
do kk=1, n, KB
```

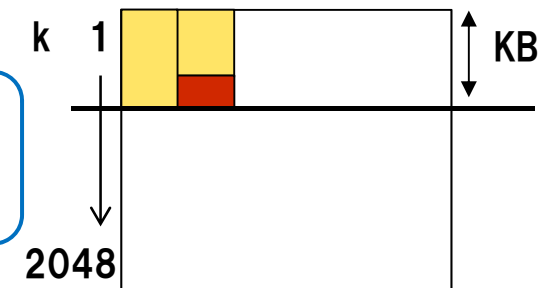
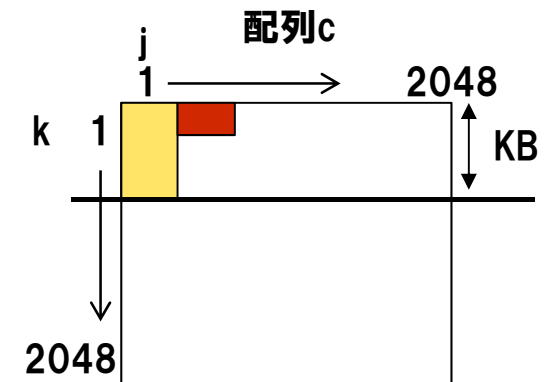
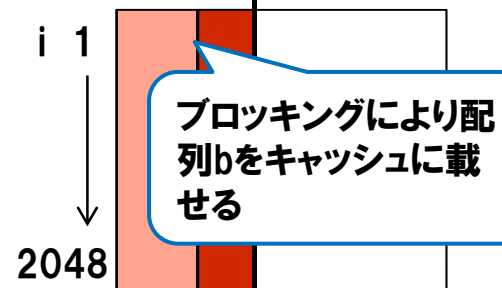
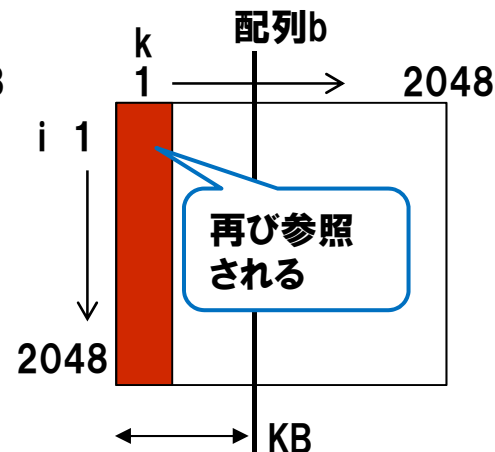
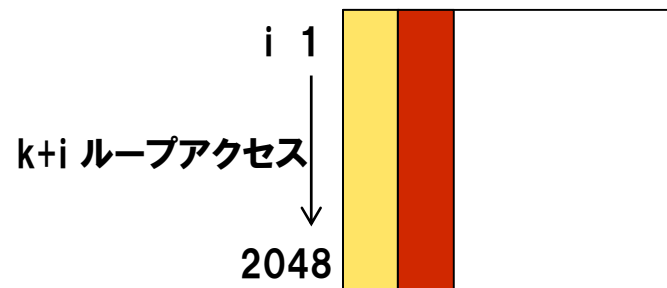
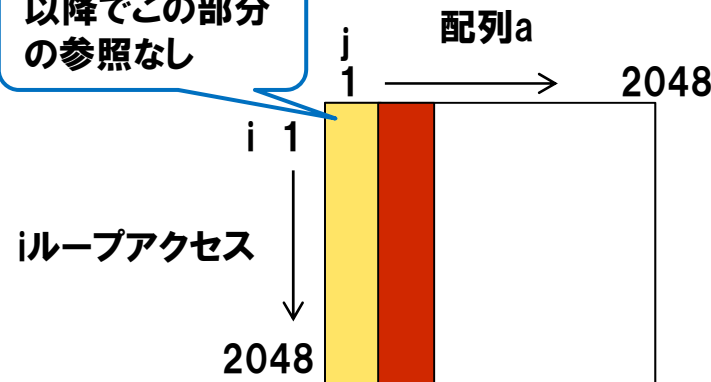
```
do j=1, n
```

```
do k=kk, min(n, kk+KB-1)
```

```
do i=1, n
```

```
a(i, j) = a(i, j) + b(i, k) * c(k, j)
```

以降でこの部分
の参照なし



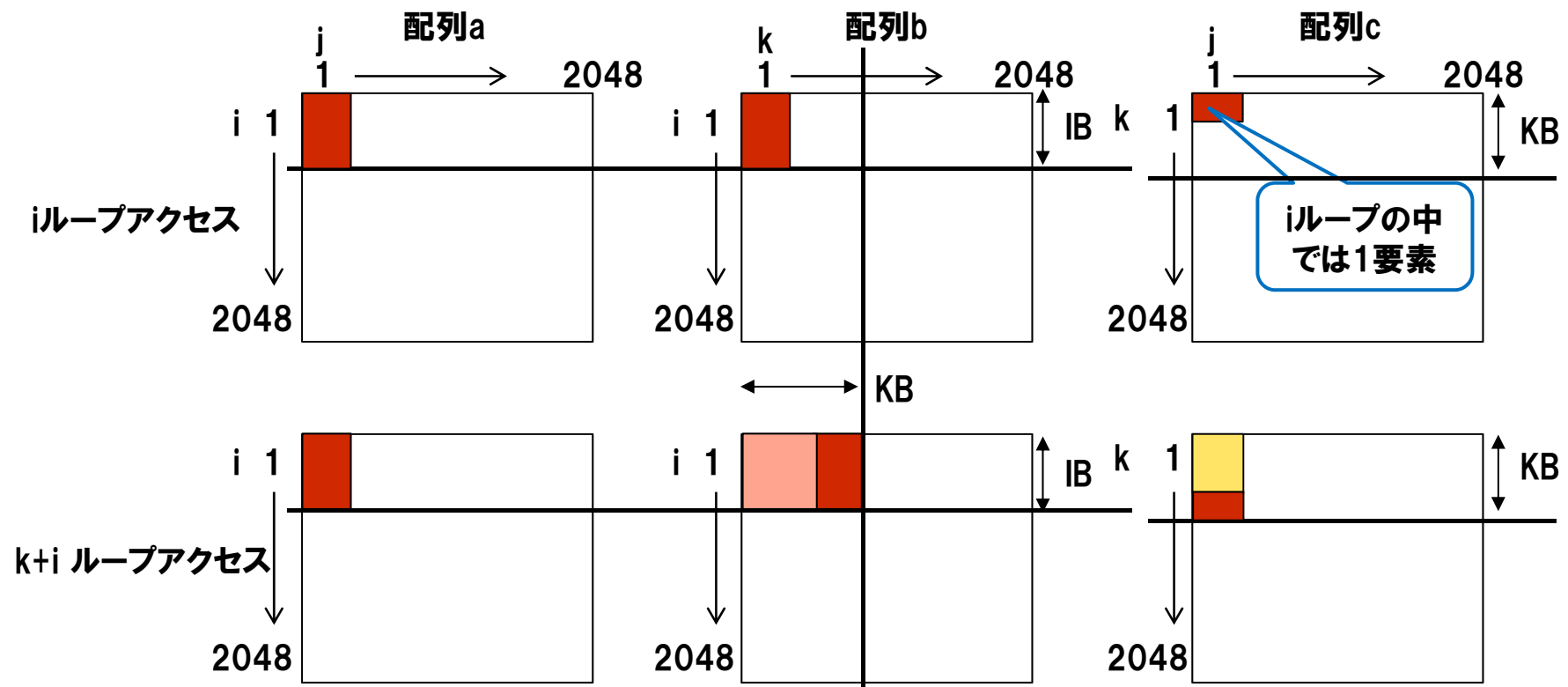
【補足】キャッシュブロッキング

iとkのブロッキングの場合

- iとkをブロッキングした時のiループとkループ配列アクセス
- IB, KBがブロッキングサイズ

行列積コード

```
do kk=1, n, KB
do ii=1, n, IB
do j=1, n
  do k=kk, min(n, kk+KB-1)
    do i=ii, min(n, ii+IB-1)
      a(i, j) = a(i, j) + b(i, k) * c(k, j)
```



【補足】キャッシュブロッキング

iとkのブロッキングの場合 (続き)

- iとkをブロッキングし、
外側jが1つ進んだ時の配列アクセス
- 配列bに着目し、IB*KBのサイズが
キャッシュに収まればjループの繰り返しの
間はキャッシュヒット

行列積コード

```
do kk=1, n, KB
```

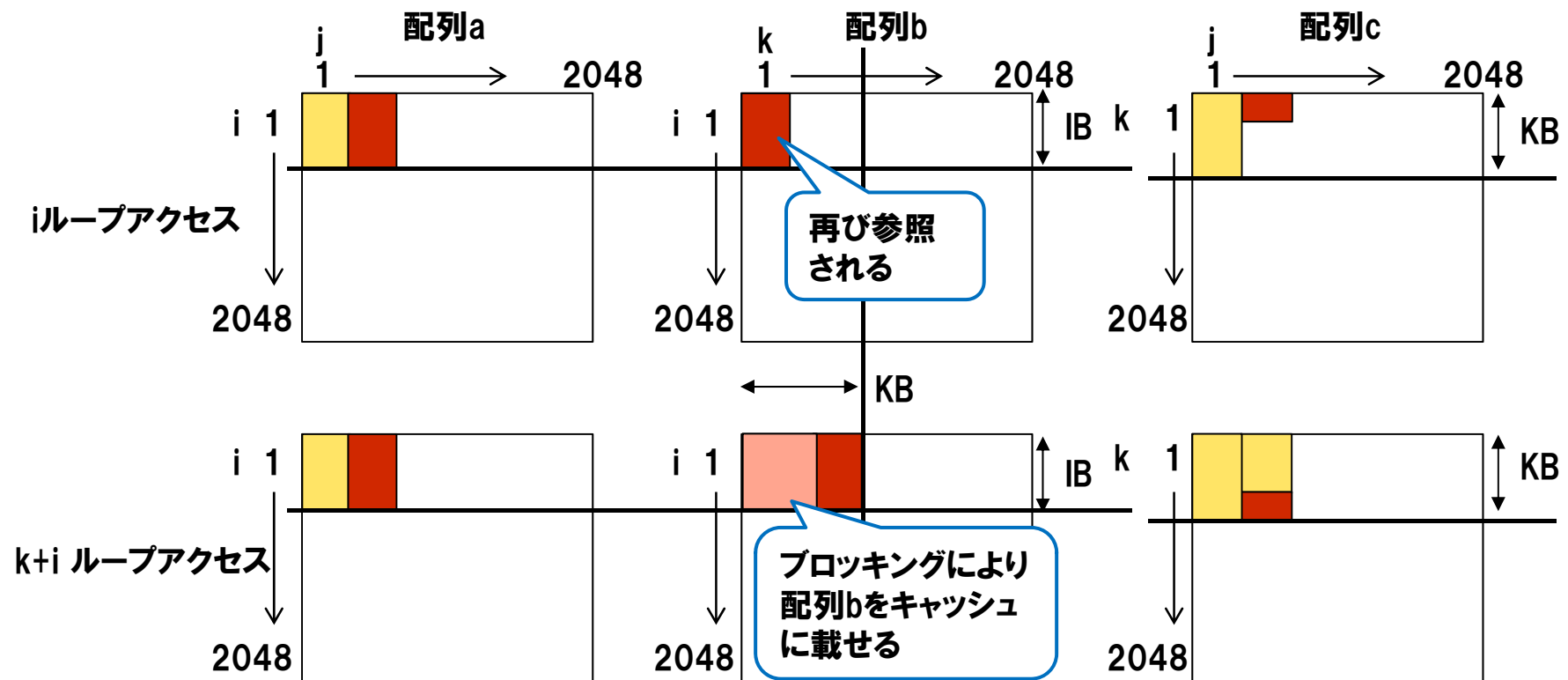
```
do ii=1, n, IB
```

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

```
do k=kk, min(n, kk+KB-1)
```

```
do i=ii, min(n, ii+IB-1)
```

```
a(i, j) = a(i, j) + b(i, k) * c(k, j)
```



【補足】キャッシュブロッキング

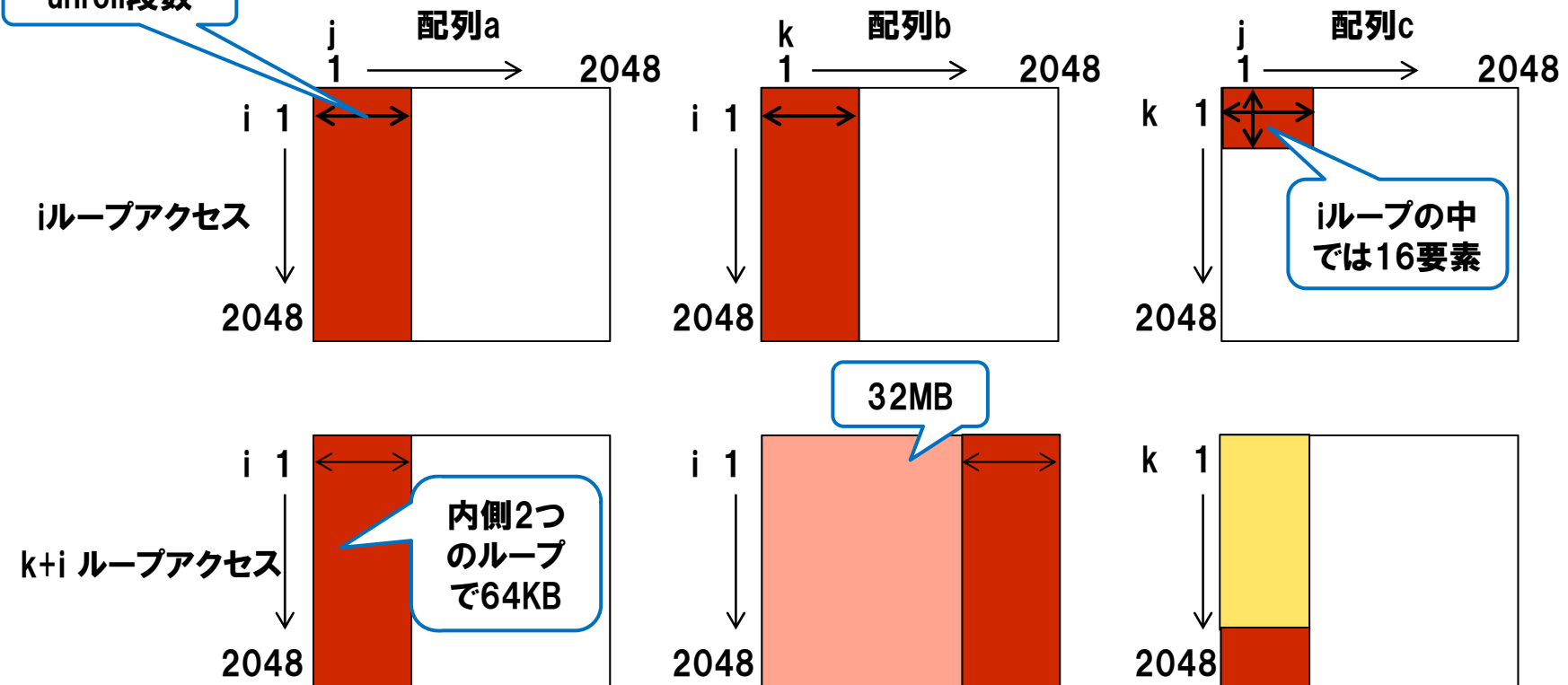
アンロールありのメモリアクセスの確認

- iループとkループ時の配列アクセス
外側2ループがそれぞれ4段アンロール
- 以降, jループを進めた時のアクセスと
ブロッキングの考え方は, 前記と同じ

行列積コード

```
do j=1, n, 4
  do k=1, n, 4
    do i=1, n
      a(i, j)=a(i, j)+b(i, k)*c(k, j)
              +b(i, k+1)*c(k+1, j)
              + ...
      a(i, j+1)=a(i, j+1)+b(i, k)*c(k, j+1)
                  + ...
```

unroll段数



【補足】キャッシュブロッキング

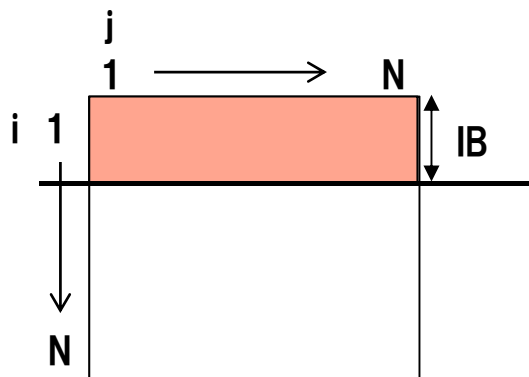
HCC/VCCのキャッシュサイズ

L1キャッシュ	32KB
L2キャッシュ	256KB

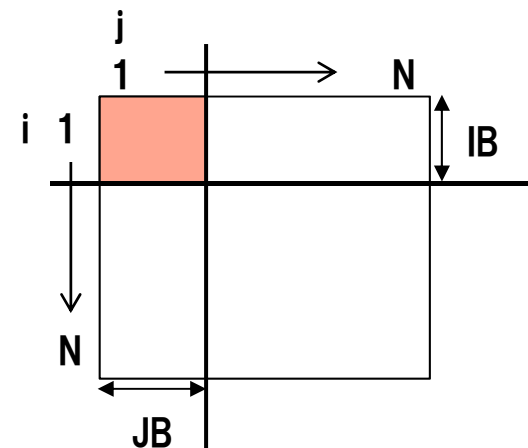
注意点

- ブロッキングサイズは**L2キャッシュサイズ**をターゲットとする。
→HCC/VCCでは256KB
- ブロッキングサイズと最内ループ長に注意する。

➡ 最内ループ長が短くなりすぎない、かつ、キャッシュに載るサイズを考える必要なら複数のブロッキングを組み合わせる。



Nが大きい場合、キャッシュに載せるためにIBを小さくすると、最内がiとなるループのループ長が短くなる。



IBとJBのブロッキングを組み合わせ、IBを小さくしすぎないようにする。

5. 演習問題 (チュートリアル形式) : 行列積チューニング

チュートリアル3

●問題と着目点

- チュートリアル2の結果を改善するため、キャッシュブロッキングを行う。
- ブロッキングサイズは、kループを256, iループを512とする。
- -O2でコンパイル・実行する。

●結果

- チュートリアル2の性能に比べて、さらに性能が向上する。

HCC/VCCのキャッシュサイズ	
L1キャッシュ	32KB
L2キャッシュ	256KB

6. まとめと質疑応答

■ まとめと質疑応答

- 演習問題・チュートリアルソースコード環境は、お持ち帰りいただけます。
ぜひ高速化のご参考にしてください。