

Fortranプログラム チューニングの 「いろは」と「つぼ」



高性能Fortran推進協議会
坂上仁志

チューニングは重要か？

- ❁ 実行時間10秒のプログラムが、10倍速くなって1秒で計算が終わるようになっても、あまりうれしくない。
- ❁ でも、一週間かかる計算が2日で終わるようになると、研究の進展度合いが大きく変わる！
 - 大規模で長時間かかる計算を行うプログラムほどチューニングは重要である。
- ❁ しかし、完成したプログラムをチューニングするのは大変なので、最初から「計算が速い」プログラムを書くことが基本である。
 - ただし、プログラムの可読性を損なわないことも、プログラムの保守を考えれば重要である。

コンパイラの最適化

- ❖ まずは、ソースを修正しなくてもできることをやる。
- ❖ コンパイラによる最適化
 - 一般には、レベル1,2,3が指定できる。 (-O1, -O2, -O3)
 - その上で特別なオプションが指定できることもある。 (-Ofast)
 - 最適化すると計算順序が変わるため、実行結果が異なることが多いので、十分に確認する。
 - $(a + b) + c \neq a + (b + c)$
 - 最適化レベル0 (-O0, 最適化しない) の結果と比べると良い。
 - 最適化レベル0では正しく計算できても、最適化すると計算結果が不正になる場合がある。
 - 最適化レベル0では、コンパイラが全変数をゼロクリアしてくれたりするので、未初期化変数（ゼロクリアを忘れた）や未定義変数（変数名をタイプミスした）の参照が原因であることが多い。

コンパイラの自動並列化

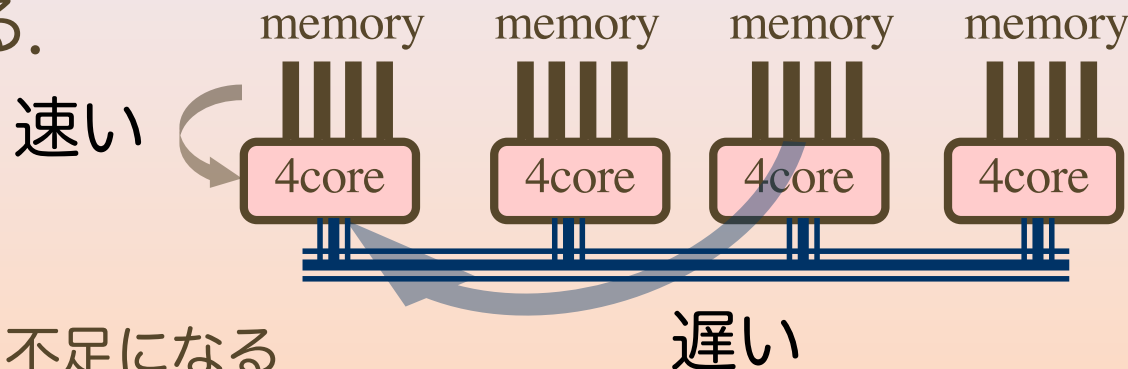
- ❁ コア数が多いコンピュータで有効である.
- ❁ コンパイラが、自動的にスレッド並列化してくれる.
 - 前述の最適化と組み合わせて利用することもできる.
 - 並列化すると、逆に遅くなる場合もある.
- ❁ 最適化と同様に計算順序が変わるため、実行結果は完全に一致しないことが多いので、十分に確認する.
 - 総和計算、最大値／最小値を持つインデックス計算
- ❁ ただし、自動並列化の能力は、一般に、コンパイラによって大きく異なるため、計算性能がコンパイラに依存してしまう.
 - あるコンピュータでは速く計算できるが、別のコンピュータでは計算が遅くなってしまう.
- ❁ スレッド並列化するときは、OpenMPの利用を考える.
 - 指示行だけでスレッド並列化ができ、自動化と違って可搬性がある.

アフィニティ（近接性）

- ✿ コアが遠くのメモリをアクセスすると実行が遅くなるため、コアとメモリの結び付きを適切に指定することで、実行が速くなる場合が多い。

- 指定しないと適当に割り振られる。
- メモリサイズに注意する。

- 4 thread並列
- 12GB必要
- 4core(node)当たり8GB
- 1nodeにbindするとメモリ不足になる



- ✿ numactlコマンドで指定できる。

```
% numactl --membind=0,1 --physcpubind=0-7 ./a.out < input.dat > output.list
```

```
% numactl --membind=2 --cpunodebind=2 ./a.out < input.dat > output.list
```

プロファイリング

- ❖ 一般に20%のサブルーチンが80%のCPU時間を使うと言われているので、計算が重たい部分、いわゆるホットスポットからチューニングすると、作業効率が良い。
- ❖ そこで、どのサブルーチンがどのくらいのCPU時間を使っているのかを調べる。
 - これを一般に、プロファイリングと呼ぶ。
- ❖ 手順は以下になる。
 1. gfortran なら -pg オプションを付けて再コンパイルする。
 2. 実行すると gmon.out というファイルが作成される。
 3. % gprof a.out gmon.out を実行するとプロファイル情報が出力される。

gprofの出力例

% time	cumulative seconds	self seconds	calls	self ms/call	total ms/call	name
40.60	19.19	19.19	30	639.54	652.12	adv3dx_
30.50	33.60	14.41	20	720.73	733.31	adv3dz_
26.31	46.03	12.43	20	621.70	634.28	adv3dy_
1.86	46.92	0.88	70	12.58	12.58	cmp3dp_
0.72	47.26	0.34	10	34.01	34.01	cmpram_
0.04	47.28	0.02	1	20.01	20.01	init_
0.00	47.28	0.00	309	0.00	0.00	measure_

消費した
CPU時間の
割合

消費した
CPU時間の
累積

消費した
CPU時間

呼び
出された
回数

自分が
呼び出した
subroutineを
含まない
消費した
CPU時間の
平均

自分が
呼び出した
subroutineを
含む
消費した
CPU時間の
平均

index	% time	self	children	called	name
<spontaneous>					
[1]	100.0	0.00	47.28		MAIN__ [1]
		19.19	0.38	30/30	adv3dx_ [2]
		14.41	0.25	20/20	adv3dz_ [3]
		12.43	0.25	20/20	adv3dy_ [4]
		0.34	0.00	10/10	cmpram_ [6]
		0.02	0.00	1/1	init_ [7]
		0.00	0.00	89/309	measure_ [8]

[2]	41.4	19.19	0.38	30/30	MAIN__ [1]
		19.19	0.38	30	adv3dx_ [2]
		0.38	0.00	30/70	cmp3dp_ [5]
		0.00	0.00	60/309	measure_ [8]

...					

		0.25	0.00	20/70	adv3dy_ [4]
		0.25	0.00	20/70	adv3dz_ [3]
		0.38	0.00	30/70	adv3dx_ [2]
[5]	1.9	0.88	0.00	70	cmp3dp_ [5]
		0.00	0.00	70/309	measure_ [8]

		0.34	0.00	10/10	MAIN__ [1]
[6]	0.7	0.34	0.00	10	cmpram_ [6]
		0.00	0.00	10/309	measure_ [8]

		0.02	0.00	1/1	MAIN__ [1]
[7]	0.0	0.02	0.00	1	init_ [7]

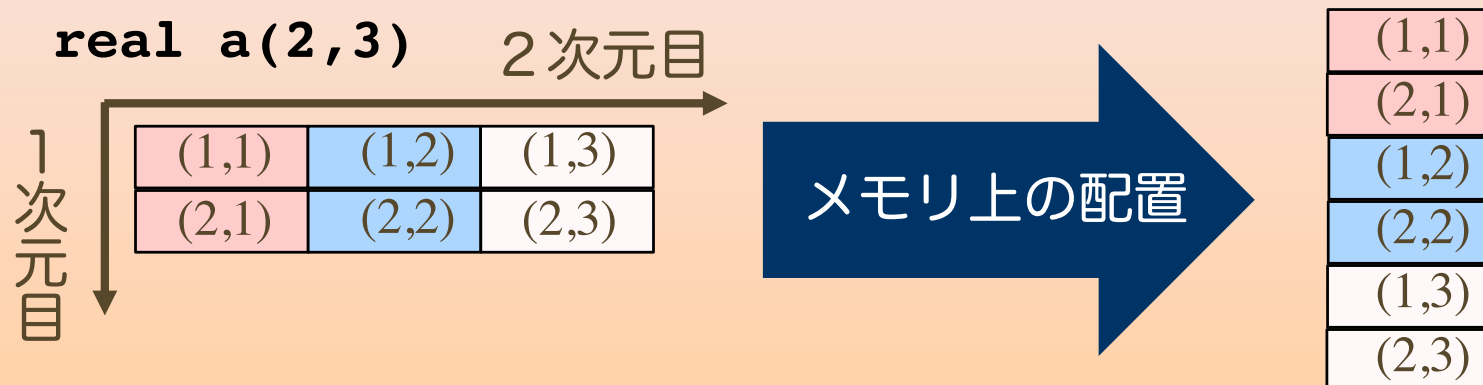
...					

「計算が速い」プログラム

- ❖ アーキテクチャによって異なるが、基本的には、
 - 人が読みやすいプログラムを書く。
 - コンパイラにも理解しやすく、最適化が容易になる。
 - メモリが連続アクセスされるように配列宣言／ループを書く。
 - ループ中では、IF文や外部関数／サブルーチンの呼び出しをできるだけ避ける。
 - チューニングされたライブラリを使う。
- ❖ ポインタは使わない。
 - 配列の形状がコンパイル時にわからないので、コンパイラの最適化が難しく、効率良く処理できない。
 - 性能 vs. 便利さ

配列のメモリ上への配置

- ❖ 連続する要素は，メモリ上に連続して配置される。
 - 形状引継ぎ配列や配列ポインタのような新しい機能を使用すると，必ずしも当てはまらない。
- ❖ 多次元配列の場合，より左側の次元の添字が先に動くような順序で配置される。
 - Cの場合は逆である。（より右側の次元の添字が先に動く順序）

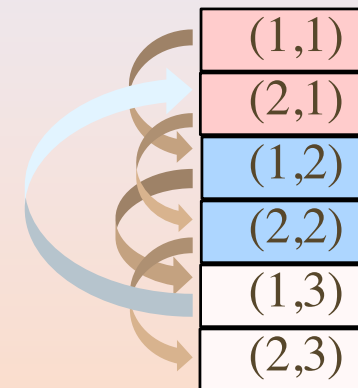


配列のアクセス

- ❖ 一般に、メモリは連続的にアクセスする方が、高速に実行される。
 - 多重ループは、メモリ上連続アクセスとなる順序で記述する方が、高速に動作する可能性が高い。

```
do i = 1, 2
  do j = 1, 3
    a(i,j) = 1.0
  end do
end do
```

メモリ上の
アクセスパターン



```
do j = 1, 3
  do i = 1, 2
    a(i,j) = 1.0
  end do
end do
```

メモリ上の
アクセスパターン

good!



DOインデックス変数

❁ DOのインデックス変数は、コンパイラの最適化を阻害するので、

- ループ外で参照しない.
- implicit none だからといって module/use しない.
- インデックス変数から誘導される変数も上記と同様に扱う.

```
do i = 1, n
  ij = i * 2 - 1
  ii = i * 2
  a(i) = b(ij)**2 + b(ii)**2
end do
```

```
do i = 1, n
  ...
  if( ... ) exit
  ...
end do
ilast = i
```

```
module doindex
integer :: i, j, k
end module
subroutine sub1
use doindex
implicit none
do k =
  do j =
    do i =
      ...
    end do
  end do
end do
subroutine sub2
use doindex
implicit none
do i =
  ...
end do
```

```
do i = 1, n
  ...
  if( ... ) then
    ilast = i
    exit
  end if
  ...
end do
```

```
subroutine sub1
implicit none
integer :: i, j, k
do k =
  do j =
    do i =
      ...
    end do
  end do
end do
subroutine sub2
implicit none
integer :: i
do i =
  ...
end do
```

演算の高速化

❁ 加減乗除

- 加減<乗<<除
- ただし, 2のべき乗の除算は, 高速なシフト演算になることがある.

```
do i = 1, n
  ene(i) = ene(i)/me/c**2.0
end do
```



```
invmec2 = 1.0/(me*c**2)
do i = 1, n
  ene(i) = ene(i) * invmec2
end do
```

❁ べき乗

- 整数べきと実数べきは違う.
- **0.5よりSQRTを使う.

```
do i = 1, n
  a(i) = a(i)*c(i)*d
  b(i) = d*b(i)*c(i)
end do
```



```
do i = 1, n
  a(i) = a(i) * (c(i)*d)
  b(i) = b(i) * (c(i)*d)
end do
```



```
do i = 1, n
  tmp = c(i) * d
  a(i) = a(i) * tmp
  b(i) = b(i) * tmp
end do
```

❁ 不変式は事前に定義する.

❁ 同一式は同じように書く.

- コンパイラが最適化しやすい.
- 可読性が高い.

再計算 vs. メモリサイズ

- ❖ 中間変数の再計算 vs. 保存/再利用を考える.
 - メモリサイズが許す限り, できるだけ保存して再利用する.
 - ただし, 単純な計算結果を保存する場合は, メモリアクセスのコストのために, かえって遅くなることもある.

```
do iy = 1, ny
  do ix = 1, nx
    cs = sqrt(gamma*p(ix,iy)/r(ix,iy))
    a(ix,iy) = ... / cs
  end do
end do
...
do iy = 1, ny
  do ix = 1, nx
    cs = sqrt(gamma*p(ix,iy)/r(ix,iy))
    b(ix,iy) = ... / cs
  end do
end do
```



```
do iy = 1, ny
  do ix = 1, nx
    invcs(ix,iy) = 1.0 / &
      sqrt(gamma*p(ix,iy)/r(ix,iy))
  end do
end do
do iy = 1, ny
  do ix = 1, nx
    a(ix,iy) = ... * invcs(ix,iy)
  end do
end do
...
do iy = 1, ny
  do ix = 1, nx
    b(ix,iy) = ... * invcs(ix,iy)
  end do
end do
```

ループアンローリング

- ❖ DOループを展開して，ループ内の実行文を増やす.
 - ループオーバーヘッド，ロード／ストアが削減できる.
 - パイプライニングによる演算の効率化が期待できる.
 - コンパイラの最適化で自動的にアンロールされることもある.
 - ただし，アンロールの段数に対して十分な数のレジスターが必要である.

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



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

* aのロード／ストア，bのロードが削減

***ループ回数がアンロール段数で割りきれないときに注意！**

インライン展開

- ❖ 外部関数やサブルーチンを導入することはプログラムの可読性向上に役立つが、ループ内で繰り返し呼び出す場合は、コーリングオーバーヘッドが大きく、かつ、コンパイラの最適化がかかりにくい。
- ❖ 外部関数／サブルーチンを呼びだし側に展開する。
 - ループアンローリングと併せて使うと効果大きい。
 - コンパイラの最適化で自動的にインライン展開されることもある。

```
do j = 1, n
  do i = 1, n
    ds(i,j) = funcs(a(i,j),b(i,j),c(i,j))
    dc(i,j) = func(a(i,j),b(i,j),c(i,j))
  end do
end do
```

```
function funcs(a, b, c)
  funcs = a*1.234 + sqrt(b) + sin(c)
  return
end
```

```
function func(a, b, c)
  func = a*6.789 + sqrt(b) + cos(c)
  return
end
```

gather/scatterによる連続アクセス

- ❖ メモリへの連続アクセスにならない場合は、一時配列を用いて連続アクセスに変換する。
 - gather/scatterのコスト vs. 計算の効率化
 - SIMD化すると効果が大きい。

```
do i = 1, n
  a(i) = a(i) + b(in(i)) + c(in(i))
  d(in(i)) = d(in(i)) + b(in(i)) * c(in(i))
end do
```



```
do i = 1, n
  b_(i) = b(in(i))
  c_(i) = c(in(i))
  d_(i) = d(in(i))
end do
do i = 1, n
  a(i) = a(i) + b_(i) + c_(i)
  d_(i) = d_(i) + b_(i) * c_(i)
end do
do i = 1, n
  d(in(i)) = d_(i)
end do
```

```
do i = 1, n
  if( icnd(i) .eq. 0 ) &
    a(i) = a(i) + b(i) * c(i)
end do
```

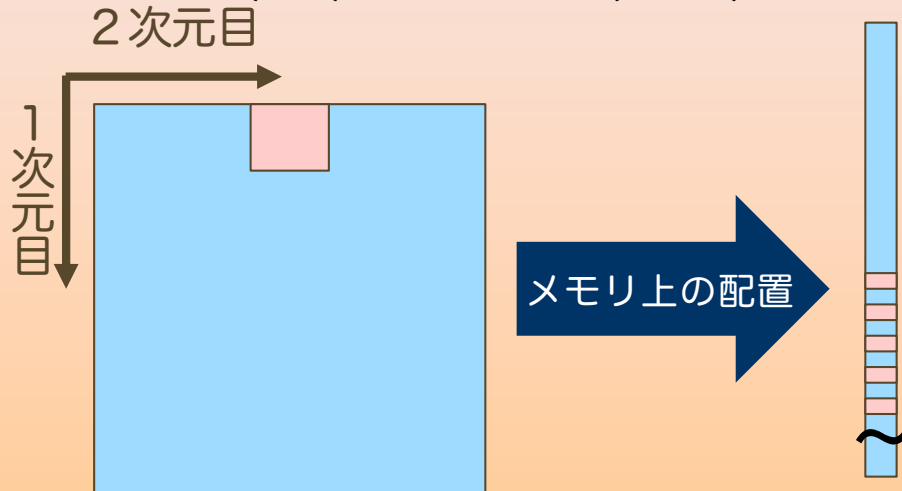


```
i_ = 0
do i = 1, n
  if( icnd(i) .eq. 0 ) then
    i_ = i_ + 1
    a_(i_) = a(i)
    b_(i_) = b(i)
    c_(i_) = c(i)
    in_(i_) = i
  end if
end do
do i = 1, i_
  a_(i) = a_(i) + b_(i) * c_(i)
end do
do i = 1, i_
  a(in_(i)) = a_(i)
end do
```

部分配列のアクセス

- ❖ 部分配列を使うとメモリへの連続アクセスにならない場合があるので、gather/scatterと同様に一時配列を用いて連続アクセスに変換する。
 - コンパイラが自動的にパック／アンパックする場合でも、無駄な処理を削除できるかもしれない。

```
dimension :: a(100,100), b(100,100)
call sub1( a(1:10,51:60), b(1:10,51:60) )
call sub2( a(1:10,51:60), b(1:10,51:60) )
```



```
dimension :: a(100,100), b(100,100)
dimension :: a_(10,10), b_(10,10)
a_(::) = a(1:10,51:60)
b_(::) = b(1:10,51:60)
call sub1( a_, b_ )
call sub2( a_, b_ )
!a(1:10,51:60) = a_(::)
!b(1:10,51:60) = b_(::)
```

ループの分割

- 最適なループ構造が変数によって異なる場合は、それぞれが最適になるようにループを分割する。

```
do j = 1, nj
  do i = 1, ni
    a(i,j) = a(i,j) + b(i,j)
    ta(j,i) = ta(j,i) + tb(j,i)
  end do
end do
```



```
do j = 1, nj
  do i = 1, ni
    a(i,j) = a(i,j) + b(i,j)
  end do
end do
do i = 1, ni
  do j = 1, nj
    ta(j,i) = ta(j,i) + tb(j,i)
  end do
end do
```

```
do i = 1, ni
  c(i) = ...
  do j = 1, nj
    a(i,j) = a(i,j) + b(i,j) * c(i)
  end do
end do
```



```
do i = 1, ni
  c(i) = ...
end do
do j = 1, nj
  do i = 1, ni
    ! c(i) = ...
    a(i,j) = a(i,j) + b(i,j) * c(i)
  end do
end do
```

ループの融合

- ❖ 最適なループ構造が同じ場合は、ループを分けずに単一のループにする。
 - ループアンローリングと同様なループオーバーヘッド，ロード／ストアの削減が期待できる。

```
do i = 1, n
  a(i) = a(i) + b(i) * c(i)
end do
do i = 1, n
  if( a(i) .lt. 0.0 ) a(i) = 0.0
end do
```



```
do i = 1, n
  a(i) = a(i) + b(i) * c(i)
  if( a(i) .lt. 0.0 ) a(i) = 0.0
end do
```



```
do i = 1, n
  tmp = a(i) + b(i) * c(i)
  if( tmp .lt. 0.0 ) tmp = 0.0
  a(i) = tmp
end do
```

ループの境界処理

- ❖ ループの上下限である境界について、特別な処理が必要な場合は、ループの外に出す。
 - IF文の処理コストが削減される。
 - 各種のコンパイラによる最適化が阻害されなくなる。
 - コンパイラの投機的実行が、無駄にならなくなる。

```
do i = 1, n
  if( i .eq. 1 ) then
    a(i) = ( b(i) + b(i+1) ) / 2.0
  else if( i .eq. n ) then
    a(i) = ( b(i-1) + b(i) ) / 2.0
  else
    a(i) = ( b(i-1)+b(i)+b(i+1) ) &
      / 3.0
  end if
end do
```



```
inv3 = 1.0 / 3.0
a(1) = ( b(1) + b(2) ) * 0.5
do i = 2, n-1
  a(i) = ( b(i-1)+b(i)+b(i+1) ) &
    * inv3
end do
a(n) = ( b(n-1) + b(n) ) * 0.5
```

ループ内の例外処理

- ❖ ループ内の例外処理をIF文で判断している場合は、なるべくIF文を除去する。
 - ループの境界処理と同様な効果が期待できる。

```
do i = 1, n
  a(i) = a(i) + b(i) * c(i)
  if( a(i).lt.0.0 ) a(i)= 0.0
end do
```



```
do i = 1, n
  a(i) = max( 0.0, &
             a(i)+b(i)*c(i) )
end do
```

```
do i = 1, n
  if( icnd .eq. 0 ) then
    a(i) = a(i) + b(i)*c(i)
  else
    a(i) = d(i)
  end if
end do
```



```
if( icnd .eq. 0 ) then
  do i = 1, n
    a(i) = a(i) + b(i)*c(i)
  end do
else
  do i = 1, n
    a(i) = d(i)
  end do
end if
```

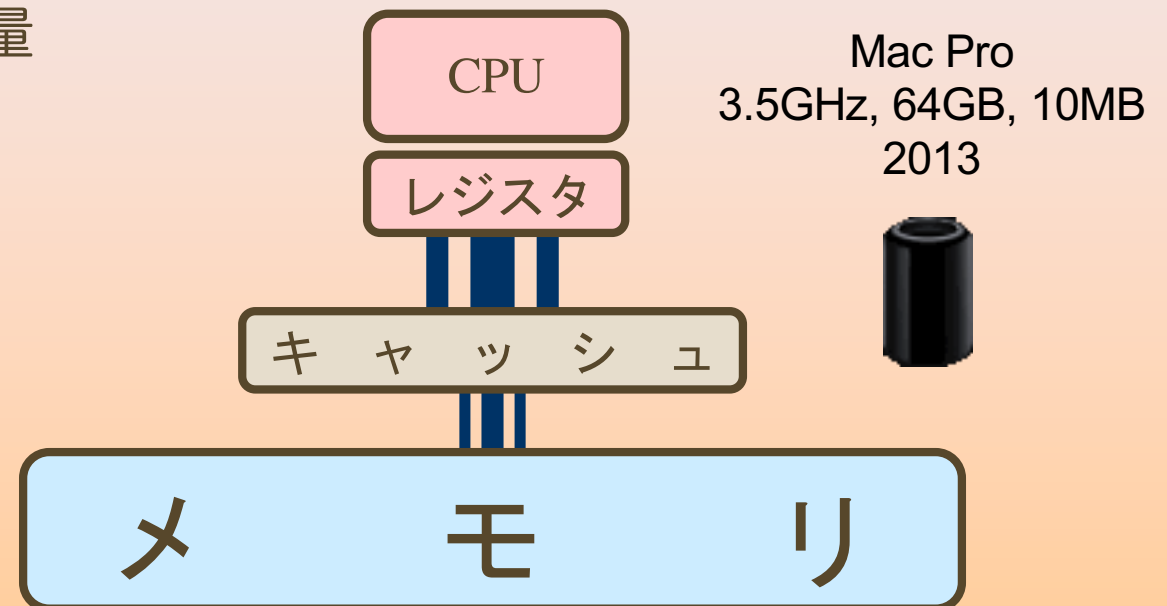
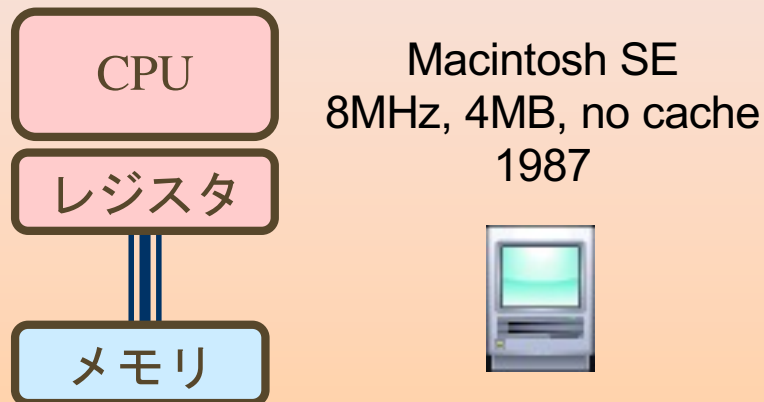
```
do j = 1, n
  do i = 1, n
    if( i .ne. j ) then
      a(i,j) = b(i,j) * c(i,j)
    else
      a(i,j) = 1.0
    end if
  end do
end do
```



```
do j = 1, n
  do i = 1, n
    a(i,j) = b(i,j) * c(i,j)
  end do
end do
do i = 1, n
  a(i,i) = 1.0
end do
```

メモリの階層構造

- ❖ 演算能力とメモリ容量は、劇的に増大したが、レジスタとメモリ間のデータ転送速度は、技術的およびコスト的にそこまで増大させることはできなかった。
 - 演算器（レジスタ）がデータ待ちをするため、著しく計算効率が低下した。
- ❖ そこで、メモリ容量こそ少ないが、レジスタへのデータ転送速度はそこそこ速いバッファメモリ（＝キャッシュ）を導入した。
 - 実際には、キャッシュも階層構造になっている。（L1, L2, L3 cache）
 - 低速・大容量 vs. 高速・低容量

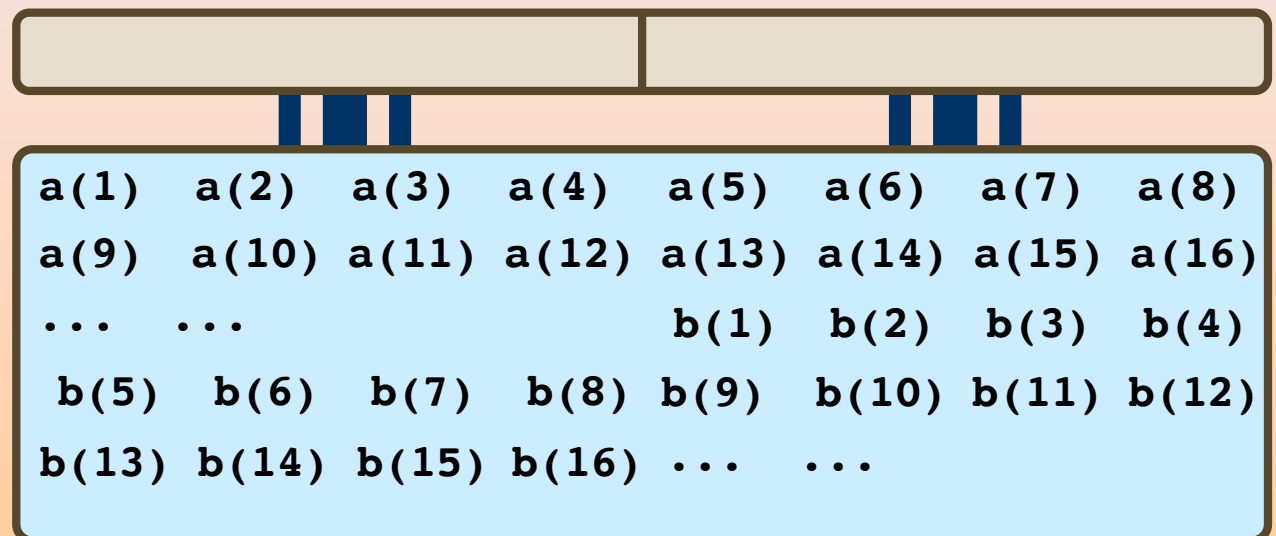


キャッシュの動作

- ❖ キャッシュ上にデータがない（キャッシュミス）とメモリから転送する。
- ❖ メモリとキャッシュは，独立した複数の経路（キャッシュライン）で接続されており，決まった大きさ（ラインサイズ）のデータが一度に転送される。
 - 異なった経路では，並行して転送される。（インターリーブ）
 - 一般にライン当たり複数の領域を持つ。（way数）
- ❖ キャッシュが一杯になると，利用率の低いデータを選んで，必要ならメモリに追い出してから，上書きする。

```
do i = 1, n
  a(i) = a(i) + b(i)
end do
```

キャッシュライン数: 2
ラインサイズ: 16B
ライン当たり: 一つ

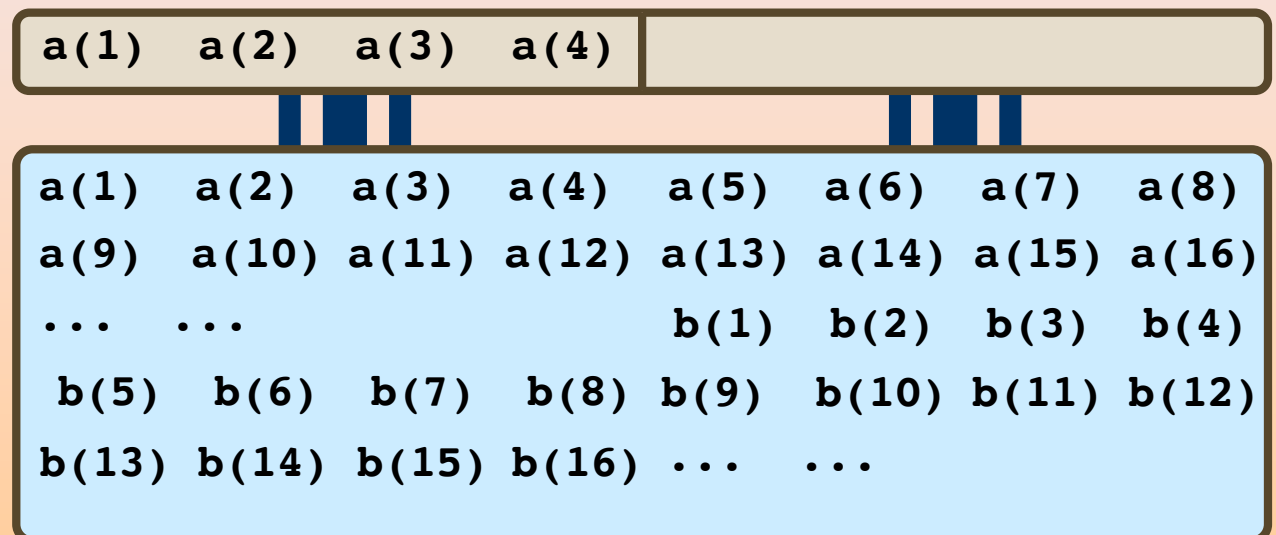


キャッシュの動作

- ❖ キャッシュ上にデータがない（キャッシュミス）とメモリから転送する。
- ❖ メモリとキャッシュは，独立した複数の経路（キャッシュライン）で接続されており，決まった大きさ（ラインサイズ）のデータが一度に転送される。
 - 異なった経路では，並行して転送される。（インターリーブ）
 - 一般にライン当たり複数の領域を持つ。（way数）
- ❖ キャッシュが一杯になると，利用率の低いデータを選んで，必要ならメモリに追い出してから，上書きする。

```
do i = 1, n
  a(i) = a(i) + b(i)
end do
```

キャッシュライン数: 2
ラインサイズ: 16B
ライン当たり: 一つ



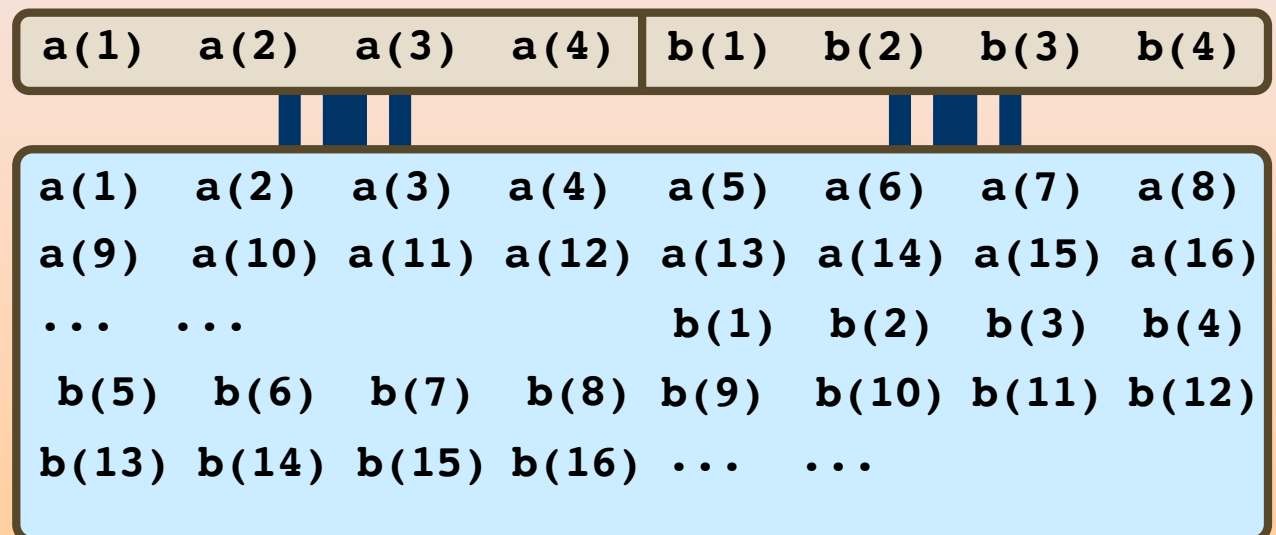
キャッシュの動作

- ❖ キャッシュ上にデータがない（キャッシュミス）とメモリから転送する。
- ❖ メモリとキャッシュは，独立した複数の経路（キャッシュライン）で接続されており，決まった大きさ（ラインサイズ）のデータが一度に転送される。
 - 異なった経路では，並行して転送される。（インターリーブ）
 - 一般にライン当たり複数の領域を持つ。（way数）
- ❖ キャッシュが一杯になると，利用率の低いデータを選んで，必要ならメモリに追い出してから，上書きする。

*実際には，aとbは同時に転送される

```
do i = 1, n
  a(i) = a(i) + b(i)
end do
```

キャッシュライン数:2
ラインサイズ:16B
ライン当たり:一つ

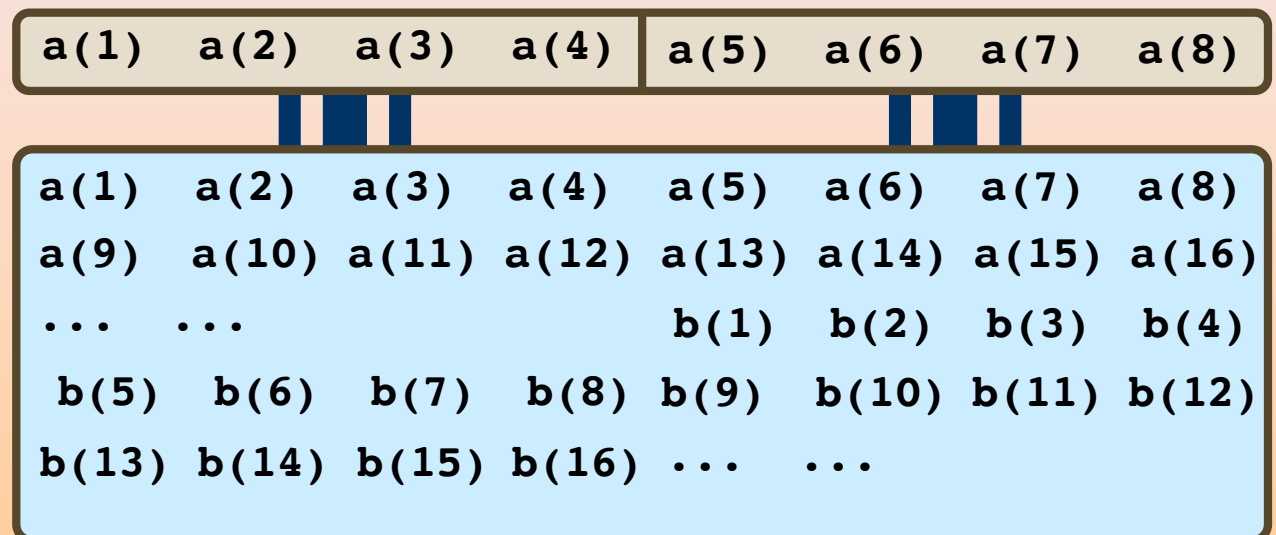


キャッシュの動作

- ❖ キャッシュ上にデータがない（キャッシュミス）とメモリから転送する。
- ❖ メモリとキャッシュは，独立した複数の経路（キャッシュライン）で接続されており，決まった大きさ（ラインサイズ）のデータが一度に転送される。
 - 異なった経路では，並行して転送される。（インターリーブ）
 - 一般にライン当たり複数の領域を持つ。（way数）
- ❖ キャッシュが一杯になると，利用率の低いデータを選んで，必要ならメモリに追い出してから，上書きする。

```
do i = 1, n
  a(i) = a(i) + b(i)
end do
```

キャッシュライン数:2
ラインサイズ:16B
ライン当たり:一つ

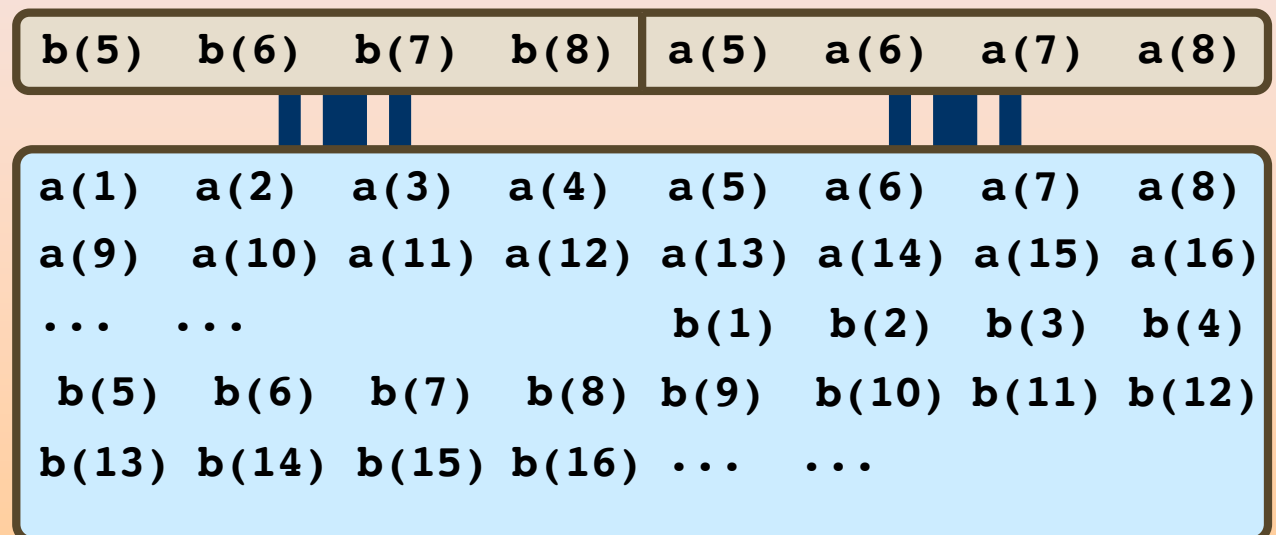


キャッシュの動作

- ❖ キャッシュ上にデータがない（キャッシュミス）とメモリから転送する。
- ❖ メモリとキャッシュは，独立した複数の経路（キャッシュライン）で接続されており，決まった大きさ（ラインサイズ）のデータが一度に転送される。
 - 異なった経路では，並行して転送される。（インターリーブ）
 - 一般にライン当たり複数の領域を持つ。（way数）
- ❖ キャッシュが一杯になると，利用率の低いデータを選んで，必要ならメモリに追い出してから，上書きする。

```
do i = 1, n
  a(i) = a(i) + b(i)
end do
```

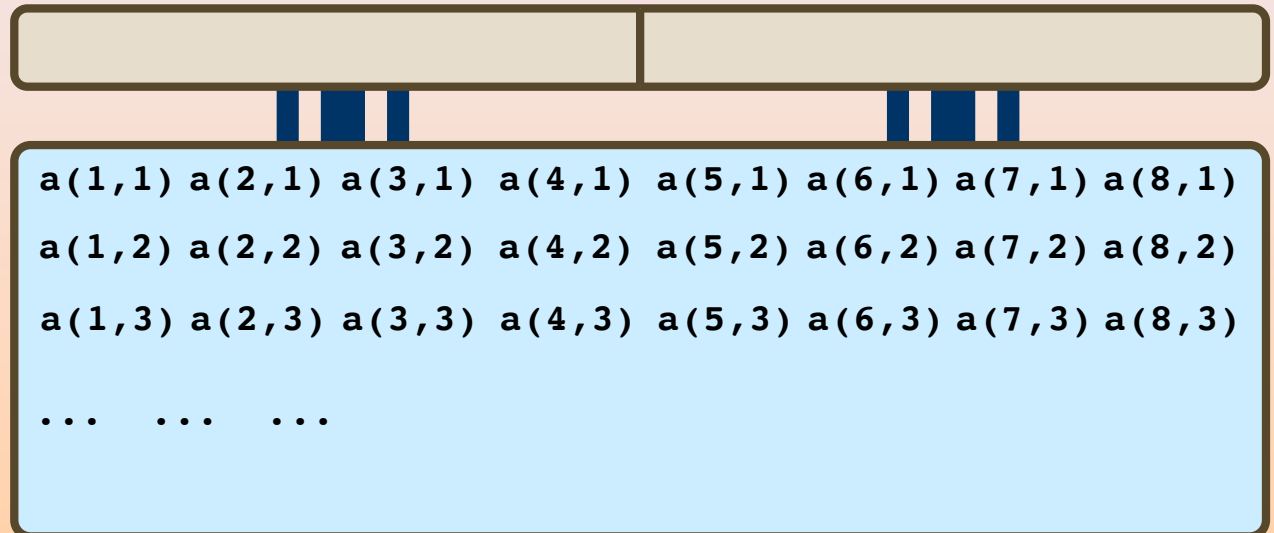
キャッシュライン数: 2
ラインサイズ: 16B
ライン当たり: 一つ



キャッシュチューニング

- なるべくキャッシュミスが起こらないよう，できるだけキャッシュ上のデータを有効に使って計算する。
 - トイプログラムと実際のプログラムでは，チューニング結果が異なることが多いので，一般的に試行錯誤が必要である。
 - コンパイラの最適化を侮ってはいけない。

```
dimension a(8,8)
do j = 1, 8
  do i = 1, 8
    a(i,j) = sqrt(a(i,j))
  end do
end do
```



キャッシュライン数:2, ラインサイズ:16B, ライン当たり:一つ

キャッシュチューニング

- なるべくキャッシュミスが起こらないよう，できるだけキャッシュ上のデータを有効に使って計算する。
 - トイプログラムと実際のプログラムでは，チューニング結果が異なることが多いので，一般的に試行錯誤が必要である。
 - コンパイラの最適化を侮ってはいけない。

```
dimension a(8,8)
do j = 1, 8
  do i = 1, 8
    a(i,j) = sqrt(a(i,j))
  end do
end do
```

a(1,1) a(2,1) a(3,1) a(4,1) | **a(5,1)** a(6,1) a(7,1) a(8,1)

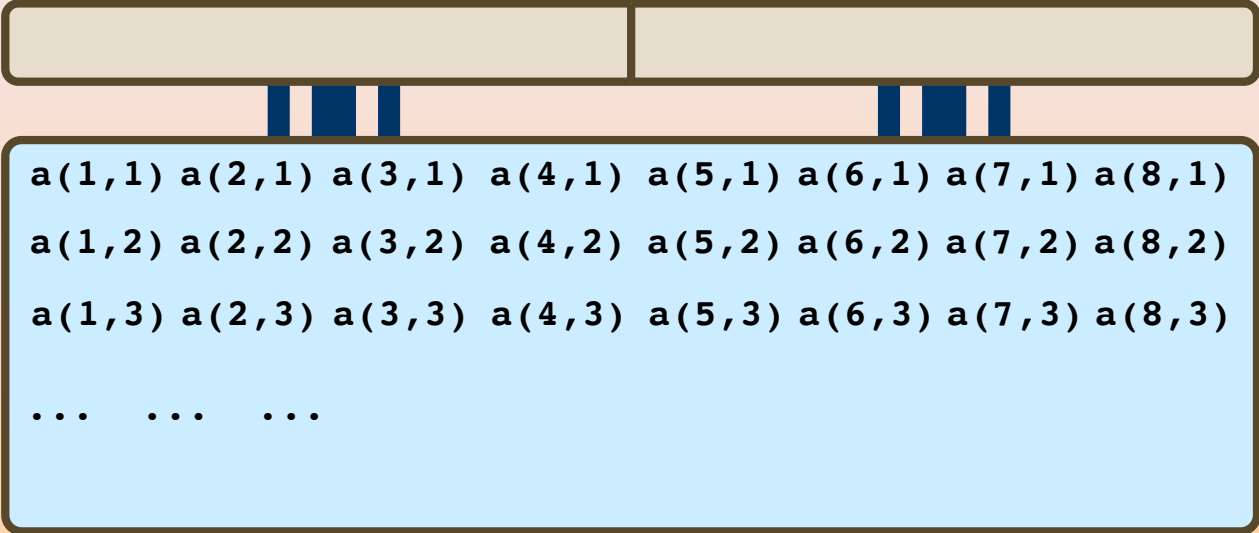
a(1,1) a(2,1) a(3,1) a(4,1) a(5,1) a(6,1) a(7,1) a(8,1)
 a(1,2) a(2,2) a(3,2) a(4,2) a(5,2) a(6,2) a(7,2) a(8,2)
 a(1,3) a(2,3) a(3,3) a(4,3) a(5,3) a(6,3) a(7,3) a(8,3)

キャッシュライン数:2, ラインサイズ:16B, ライン当たり:一つ

キャッシュチューニング

- なるべくキャッシュミスが起こらないよう，できるだけキャッシュ上のデータを有効に使って計算する。
 - トイプログラムと実際のプログラムでは，チューニング結果が異なることが多いので，一般的に試行錯誤が必要である。
 - コンパイラの最適化を侮ってはいけない。

```
dimension a(8,8)
do i = 1, 8
  do j = 1, 8
    a(i,j) = sqrt(a(i,j))
  end do
end do
```



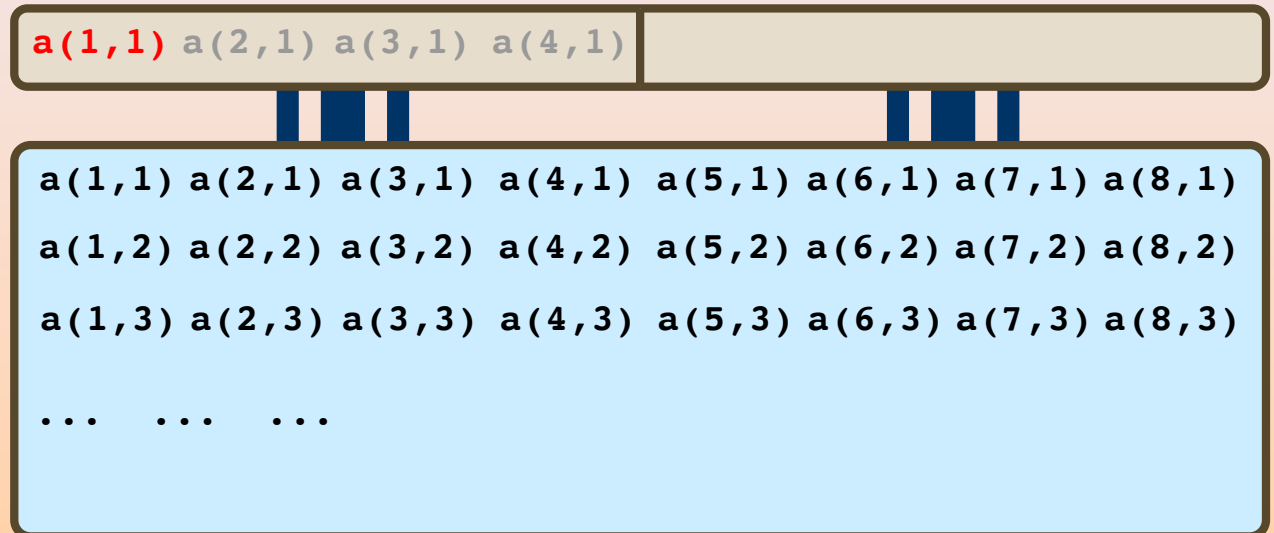
```
a(1,1) a(2,1) a(3,1) a(4,1) a(5,1) a(6,1) a(7,1) a(8,1)
a(1,2) a(2,2) a(3,2) a(4,2) a(5,2) a(6,2) a(7,2) a(8,2)
a(1,3) a(2,3) a(3,3) a(4,3) a(5,3) a(6,3) a(7,3) a(8,3)
...   ...   ...
```

キャッシュライン数:2, ラインサイズ:16B, ライン当たり:一つ

キャッシュチューニング

- なるべくキャッシュミスが起こらないよう，できるだけキャッシュ上のデータを有効に使って計算する。
 - トイプログラムと実際のプログラムでは，チューニング結果が異なることが多いので，一般的に試行錯誤が必要である。
 - コンパイラの最適化を侮ってはいけない。

```
dimension a(8,8)
do i = 1, 8
  do j = 1, 8
    a(i,j) = sqrt(a(i,j))
  end do
end do
```

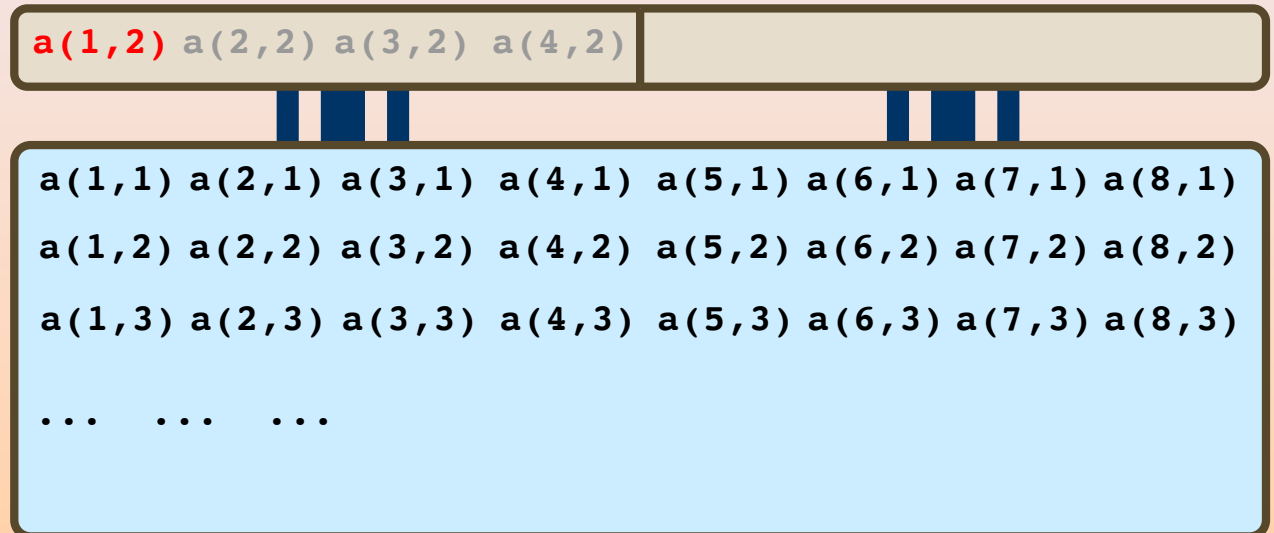


キャッシュライン数:2, ラインサイズ:16B, ライン当たり:一つ

キャッシュチューニング

- なるべくキャッシュミスが起こらないよう，できるだけキャッシュ上のデータを有効に使って計算する。
 - トイプログラムと実際のプログラムでは，チューニング結果が異なることが多いので，一般的に試行錯誤が必要である。
 - コンパイラの最適化を侮ってはいけない。

```
dimension a(8,8)
do i = 1, 8
  do j = 1, 8
    a(i,j) = sqrt(a(i,j))
  end do
end do
```



キャッシュライン数:2, ラインサイズ:16B, ライン当たり:一つ

キャッシュチューニング

- なるべくキャッシュミスが起こらないよう，できるだけキャッシュ上のデータを有効に使って計算する。
 - トイプログラムと実際のプログラムでは，チューニング結果が異なることが多いので，一般的に試行錯誤が必要である。
 - コンパイラの最適化を侮ってはいけない。

```
dimension a(8,8)
do i = 1, 8
  do j = 1, 8
    a(i,j) = sqrt(a(i,j))
  end do
end do
```

a(1,3) a(2,3) a(3,3) a(4,3)

```
a(1,1) a(2,1) a(3,1) a(4,1) a(5,1) a(6,1) a(7,1) a(8,1)
a(1,2) a(2,2) a(3,2) a(4,2) a(5,2) a(6,2) a(7,2) a(8,2)
a(1,3) a(2,3) a(3,3) a(4,3) a(5,3) a(6,3) a(7,3) a(8,3)
...   ...   ...
```

キャッシュライン数:2, ラインサイズ:16B, ライン当たり:一つ

Structure of Array vs. Array of Structure

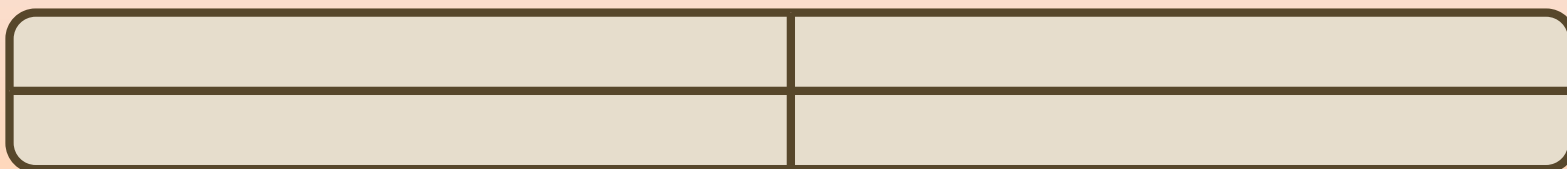
```
parameter( N=10000000 )
real pv(N,4)
do i = 1, N
    pv(i,1) = pv(i,1) + pv(i,2)
    pv(i,3) = pv(i,3) + pv(i,4)
end do
```

```
parameter( N=10000000 )
real pv(4,N)
do i = 1, N
    pv(1,i) = pv(1,i) + pv(2,i)
    pv(3,i) = pv(3,i) + pv(4,i)
end do
```



キャッシュライン数:2, ラインサイズ:16B, ライン当たり:一つ

SoAではキャッシュ上のデータを有効に使えないので, 一般的にはAoSの方が速いと言われている.



キャッシュライン数:2, ラインサイズ:16B, ライン当たり:二つ

ただし, AoSではキャッシュ上データの書き戻しに無駄がある.

Structure of Array vs. Array of Structure

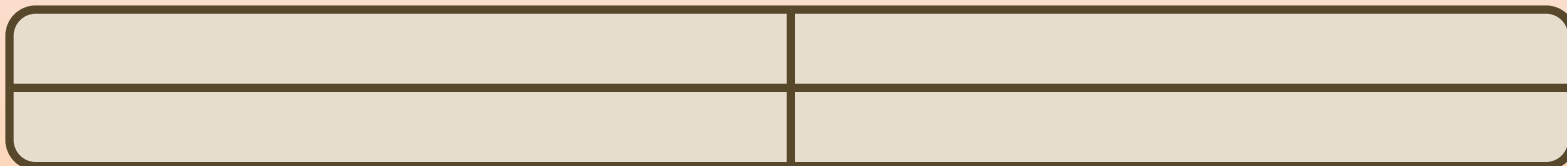
```
parameter( N=10000000 )
real pv(N,4)
do i = 1, N
    pv(i,1) = pv(i,1) + pv(i,2)
    pv(i,3) = pv(i,3) + pv(i,4)
end do
```

```
parameter( N=10000000 )
real pv(4,N)
do i = 1, N
    pv(1,i) = pv(1,i) + pv(2,i)
    pv(3,i) = pv(3,i) + pv(4,i)
end do
```

pv(1,1) pv(2,1) pv(3,1) pv(4,1)

キャッシュライン数:2, ラインサイズ:16B, ライン当たり:一つ

SoAではキャッシュ上のデータを有効に使えないので, 一般的にはAoSの方が速いと言われている.



キャッシュライン数:2, ラインサイズ:16B, ライン当たり:二つ

ただし, AoSではキャッシュ上データの書き戻しに無駄がある.

Structure of Array vs. Array of Structure

```
parameter( N=10000000 )
real pv(N,4)
do i = 1, N
    pv(i,1) = pv(i,1) + pv(i,2)
    pv(i,3) = pv(i,3) + pv(i,4)
end do
```

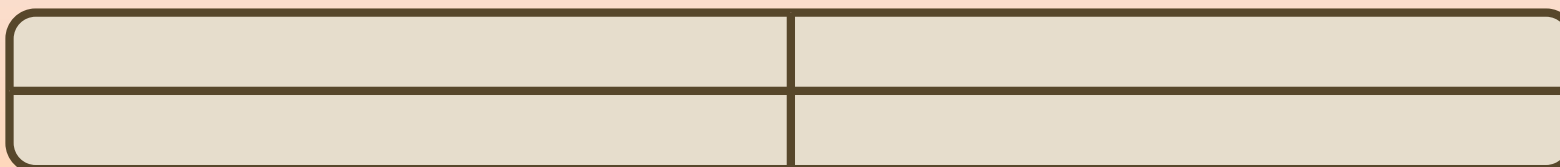
```
parameter( N=10000000 )
real pv(4,N)
do i = 1, N
    pv(1,i) = pv(1,i) + pv(2,i)
    pv(3,i) = pv(3,i) + pv(4,i)
end do
```

pv(1,1) pv(2,1) pv(3,1) pv(4,1)

pv(1,2) pv(2,2) pv(3,2) pv(4,2)

キャッシュライン数:2, ラインサイズ:16B, ライン当たり:一つ

SoAではキャッシュ上のデータを有効に使えないので, 一般的にはAoSの方が速いと言われている.



キャッシュライン数:2, ラインサイズ:16B, ライン当たり:二つ

ただし, AoSではキャッシュ上データの書き戻しに無駄がある.

Structure of Array vs. Array of Structure

```
parameter( N=10000000 )  
real pv(N,4)  
do i = 1, N  
    pv(i,1) = pv(i,1) + pv(i,2)  
    pv(i,3) = pv(i,3) + pv(i,4)  
end do
```

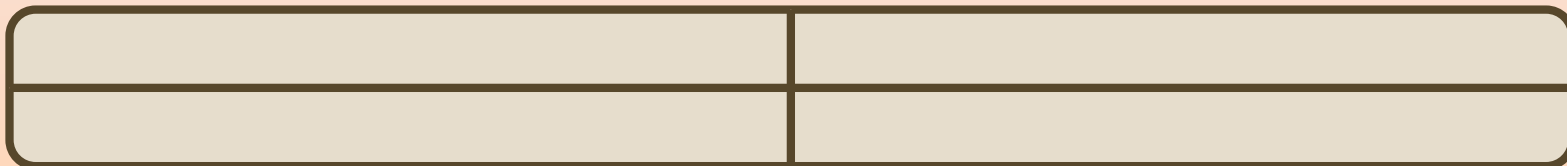
```
parameter( N=10000000 )  
real pv(4,N)  
do i = 1, N  
    pv(1,i) = pv(1,i) + pv(2,i)  
    pv(3,i) = pv(3,i) + pv(4,i)  
end do
```

pv(1,3) pv(2,3) pv(3,3) pv(4,3)

pv(1,2) pv(2,2) pv(3,2) pv(4,2)

キャッシュライン数:2, ラインサイズ:16B, ライン当たり:一つ

SoAではキャッシュ上のデータを有効に使えないので, 一般的にはAoSの方が速いと言われている.



キャッシュライン数:2, ラインサイズ:16B, ライン当たり:二つ

ただし, AoSではキャッシュ上データの書き戻しに無駄がある.

Structure of Array vs. Array of Structure



```
parameter( N=10000000 )  
real pv(N,4)  
do i = 1, N  
    pv(i,1) = pv(i,1) + pv(i,2)  
    pv(i,3) = pv(i,3) + pv(i,4)  
end do
```

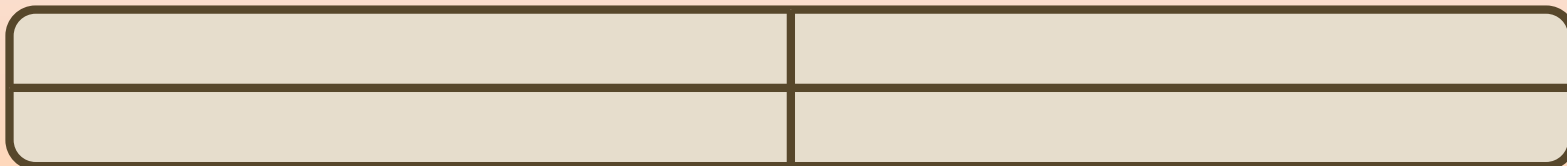
```
parameter( N=10000000 )  
real pv(4,N)  
do i = 1, N  
    pv(1,i) = pv(1,i) + pv(2,i)  
    pv(3,i) = pv(3,i) + pv(4,i)  
end do
```

pv(1,3) pv(2,3) pv(3,3) pv(4,3)

pv(1,4) pv(2,4) pv(3,4) pv(4,4)

キャッシュライン数:2, ラインサイズ:16B, ライン当たり:一つ

SoAではキャッシュ上のデータを有効に使えないので, 一般的にはAoSの方が速いと言われている.



キャッシュライン数:2, ラインサイズ:16B, ライン当たり:二つ

ただし, AoSではキャッシュ上データの書き戻しに無駄がある.

Structure of Array vs. Array of Structure

```
parameter( N=10000000 )
real pv(N,4)
do i = 1, N
    pv(i,1) = pv(i,1) + pv(i,2)
    pv(i,3) = pv(i,3) + pv(i,4)
end do
```

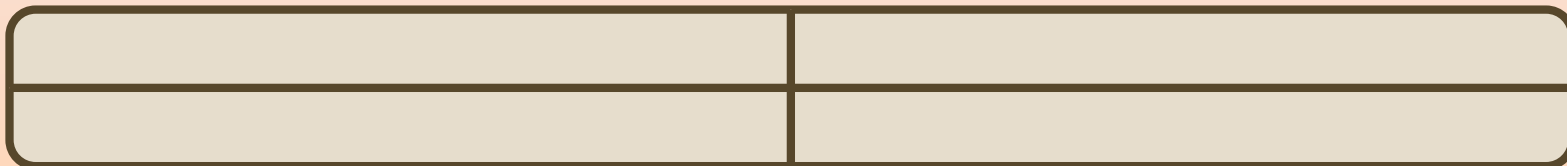
```
parameter( N=10000000 )
real pv(4,N)
do i = 1, N
    pv(1,i) = pv(1,i) + pv(2,i)
    pv(3,i) = pv(3,i) + pv(4,i)
end do
```

pv(1,1) pv(2,1) pv(3,1) pv(4,1)

pv(1,2) pv(2,2) pv(3,2) pv(4,2)

キャッシュライン数:2, ラインサイズ:16B, ライン当たり:一つ

SoAではキャッシュ上のデータを有効に使えないので, 一般的にはAoSの方が速いと言われている.



キャッシュライン数:2, ラインサイズ:16B, ライン当たり:二つ

ただし, AoSではキャッシュ上データの書き戻しに無駄がある.

Structure of Array vs. Array of Structure

```
parameter( N=10000000 )
real pv(N,4)
do i = 1, N
    pv(i,1) = pv(i,1) + pv(i,2)
    pv(i,3) = pv(i,3) + pv(i,4)
end do
```

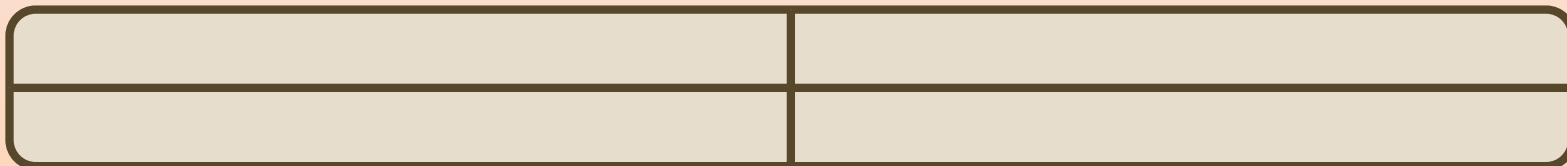
```
parameter( N=10000000 )
real pv(4,N)
do i = 1, N
    pv(1,i) = pv(1,i) + pv(2,i)
    pv(3,i) = pv(3,i) + pv(4,i)
end do
```

pv(1,3) pv(2,3) pv(3,3) pv(4,3)

pv(1,4) pv(2,4) pv(3,4) pv(4,4)

キャッシュライン数:2, ラインサイズ:16B, ライン当たり:一つ

SoAではキャッシュ上のデータを有効に使えないので, 一般的にはAoSの方が速いと言われている.



キャッシュライン数:2, ラインサイズ:16B, ライン当たり:二つ

ただし, AoSではキャッシュ上データの書き戻しに無駄がある.

Structure of Array vs. Array of Structure

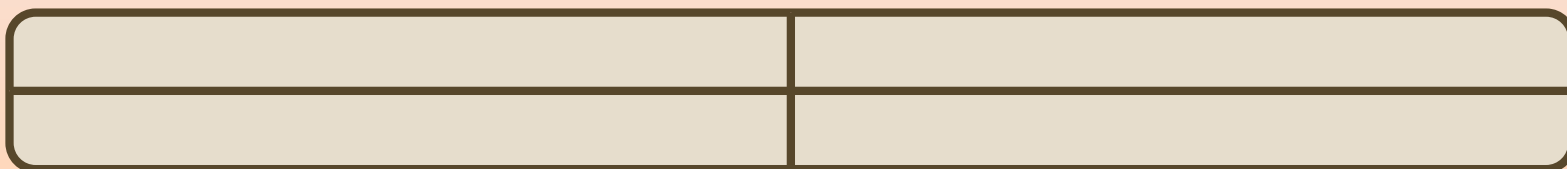
```
parameter( N=10000000 )
real pv(N,4)
do i = 1, N
    pv(i,1) = pv(i,1) + pv(i,2)
    pv(i,3) = pv(i,3) + pv(i,4)
end do
```

```
parameter( N=10000000 )
real pv(4,N)
do i = 1, N
    pv(1,i) = pv(1,i) + pv(2,i)
    pv(3,i) = pv(3,i) + pv(4,i)
end do
```



キャッシュライン数:2, ラインサイズ:16B, ライン当たり:一つ

SoAではキャッシュ上のデータを有効に使えないので, 一般的にはAoSの方が速いと言われている.



キャッシュライン数:2, ラインサイズ:16B, ライン当たり:二つ

ただし, AoSではキャッシュ上データの書き戻しに無駄がある.

Structure of Array vs. Array of Structure

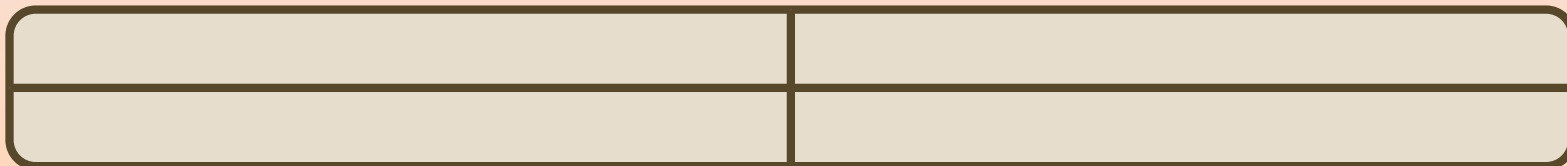
```
parameter( N=10000000 )  
real pv(N,4)  
do i = 1, N  
    pv(i,1) = pv(i,1) + pv(i,2)  
    pv(i,3) = pv(i,3) + pv(i,4)  
end do
```

```
parameter( N=10000000 )  
real pv(4,N)  
do i = 1, N  
    pv(1,i) = pv(1,i) + pv(2,i)  
    pv(3,i) = pv(3,i) + pv(4,i)  
end do
```

<code>pv(1,1)</code> <code>pv(2,1)</code> <code>pv(3,1)</code> <code>pv(4,1)</code>	
---	--

キャッシュライン数:2, ラインサイズ:16B, ライン当たり:一つ

SoAではキャッシュ上のデータを有効に使えないので, 一般的にはAoSの方が速いと言われている.



キャッシュライン数:2, ラインサイズ:16B, ライン当たり:二つ

ただし, AoSではキャッシュ上データの書き戻しに無駄がある.

Structure of Array vs. Array of Structure

```
parameter( N=10000000 )
real pv(N,4)
do i = 1, N
  pv(i,1) = pv(i,1) + pv(i,2)
  pv(i,3) = pv(i,3) + pv(i,4)
end do
```

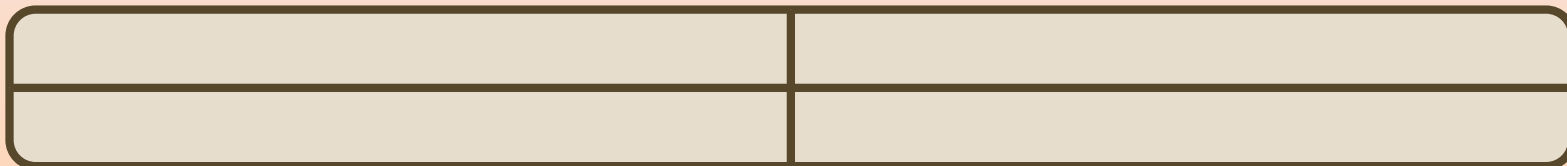
```
parameter( N=10000000 )
real pv(4,N)
do i = 1, N
  pv(1,i) = pv(1,i) + pv(2,i)
  pv(3,i) = pv(3,i) + pv(4,i)
end do
```

pv(1,1) pv(2,1) pv(3,1) pv(4,1)

pv(1,2) pv(2,2) pv(3,2) pv(4,2)

キャッシュライン数:2, ラインサイズ:16B, ライン当たり:一つ

SoAではキャッシュ上のデータを有効に使えないので, 一般的にはAoSの方が速いと言われている.



キャッシュライン数:2, ラインサイズ:16B, ライン当たり:二つ

ただし, AoSではキャッシュ上データの書き戻しに無駄がある.

Structure of Array vs. Array of Structure

```
parameter( N=10000000 )
real pv(N,4)
do i = 1, N
    pv(i,1) = pv(i,1) + pv(i,2)
    pv(i,3) = pv(i,3) + pv(i,4)
end do
```

```
parameter( N=10000000 )
real pv(4,N)
do i = 1, N
    pv(1,i) = pv(1,i) + pv(2,i)
    pv(3,i) = pv(3,i) + pv(4,i)
end do
```

pv(1,1) pv(2,1) pv(3,1) pv(4,1)

pv(1,2) pv(2,2) pv(3,2) pv(4,2)

キャッシュライン数:2, ラインサイズ:16B, ライン当たり:一つ

SoAではキャッシュ上のデータを有効に使えないので, 一般的にはAoSの方が速いと言われている.

pv(1,1) pv(2,1) pv(3,1) pv(4,1)

pv(1,2) pv(2,2) pv(3,2) pv(4,2)

キャッシュライン数:2, ラインサイズ:16B, ライン当たり:二つ

ただし, AoSではキャッシュ上データの書き戻しに無駄がある.

Structure of Array vs. Array of Structure

```
parameter( N=10000000 )
real pv(N,4)
do i = 1, N
    pv(i,1) = pv(i,1) + pv(i,2)
    pv(i,3) = pv(i,3) + pv(i,4)
end do
```

```
parameter( N=10000000 )
real pv(4,N)
do i = 1, N
    pv(1,i) = pv(1,i) + pv(2,i)
    pv(3,i) = pv(3,i) + pv(4,i)
end do
```

pv(1,1) pv(2,1) pv(3,1) pv(4,1)

pv(1,2) pv(2,2) pv(3,2) pv(4,2)

キャッシュライン数:2, ラインサイズ:16B, ライン当たり:一つ

SoAではキャッシュ上のデータを有効に使えないので, 一般的にはAoSの方が速いと言われている.

pv(1,3) pv(2,3) pv(3,3) pv(4,3)

pv(1,4) pv(2,4) pv(3,4) pv(4,4)

pv(1,1) pv(2,1) pv(3,1) pv(4,1)

pv(1,2) pv(2,2) pv(3,2) pv(4,2)

キャッシュライン数:2, ラインサイズ:16B, ライン当たり:二つ

ただし, AoSではキャッシュ上データの書き戻しに無駄がある.

Structure of Array vs. Array of Structure

```
parameter( N=10000000 )
real pv(N,4)
do i = 1, N
    pv(i,1) = pv(i,1) + pv(i,2)
    pv(i,3) = pv(i,3) + pv(i,4)
end do
```

```
parameter( N=10000000 )
real pv(4,N)
do i = 1, N
    pv(1,i) = pv(1,i) + pv(2,i)
    pv(3,i) = pv(3,i) + pv(4,i)
end do
```

pv(1,1) pv(2,1) pv(3,1) pv(4,1)

pv(1,2) pv(2,2) pv(3,2) pv(4,2)

キャッシュライン数:2, ラインサイズ:16B, ライン当たり:一つ

SoAではキャッシュ上のデータを有効に使えないので, 一般的にはAoSの方が速いと言われている.

pv(1,3) pv(2,3) pv(3,3) pv(4,3)

pv(1,4) pv(2,4) pv(3,4) pv(4,4)

pv(1,1) pv(2,1) pv(3,1) pv(4,1)

pv(1,2) pv(2,2) pv(3,2) pv(4,2)

キャッシュライン数:2, ラインサイズ:16B, ライン当たり:二つ

ただし, AoSではキャッシュ上データの書き戻しに無駄がある.

Structure of Array vs. Array of Structure

```
parameter( N=10000000 )
real pv(N,4)
do i = 1, N
    pv(i,1) = pv(i,1) + pv(i,2)
    pv(i,3) = pv(i,3) + pv(i,4)
end do
```

```
parameter( N=10000000 )
real pv(4,N)
do i = 1, N
    pv(1,i) = pv(1,i) + pv(2,i)
    pv(3,i) = pv(3,i) + pv(4,i)
end do
```

pv(1,1) pv(2,1) pv(3,1) pv(4,1)

pv(1,2) pv(2,2) pv(3,2) pv(4,2)

キャッシュライン数:2, ラインサイズ:16B, ライン当たり:一つ

SoAではキャッシュ上のデータを有効に使えないので, 一般的にはAoSの方が速いと言われている.

pv(1,3) pv(2,3) **pv(3,3)** pv(4,3)

pv(1,4) pv(2,4) **pv(3,4)** pv(4,4)

pv(1,1) pv(2,1) **pv(3,1)** pv(4,1)

pv(1,2) pv(2,2) **pv(3,2)** pv(4,2)

キャッシュライン数:2, ラインサイズ:16B, ライン当たり:二つ

ただし, AoSではキャッシュ上データの書き戻しに無駄がある.

Array of Structure vs. Vector

- 従来は， AoSの方が速いと言われていたが， SSE, AVX, SIMDや GPGPUでは， ループを回すときに連続アクセスになると速いため， SoAの方が速くなる場合もある。

```
parameter( N=10000000 )
real pv(N,4)
do i = 1, N, 4
    pv(i:i+3,1) = pv(i:i+3,1) + pv(i:i+3,2)
    pv(i:i+3,3) = pv(i:i+3,3) + pv(i:i+3,4)
end do
```

*i:i+3は, SIMD化を明示しているだけ！

```
parameter( N=10000000 )
real pv(N,4)
do i = 1, N
    pv(i,1) = pv(i,1) + pv(i,2)
    pv(i,3) = pv(i,3) + pv(i,4)
end do

parameter( N=10000000 )
real pv(N,4)
do i = 1, N, 4
    do j = i, i+3
        pv(j,1) = pv(j,1) + pv(j,2)
        pv(j,3) = pv(j,3) + pv(j,4)
    end do
end do
```

- キャッシュ容量が大きくなったこともあり，ベクトル向けに書かれたコードが，ベクトル機でなくても最速のことがある。

- 1次元配列のインデックス参照の方が，2次元のそれより高速である。

```
parameter( N=10000000 )
real x(N), y(N), vx(N), vy(N)
do i = 1, N
    x(i) = x(i) + vx(i)
    y(i) = y(i) + vy(i)
end do
```



```
parameter( N=10000000 )
real x(N), y(N), vx(N), vy(N)
x(1:N) = x(1:N) + vx(1:N)
y(1:N) = y(1:N) + vy(1:N)
```

*1:Nは, SIMD化を明示しているだけ！

データ構造によらない書き方

- ❖ AoSの2次元配列を使った書き方は、わかりにくい。
 - 構造体を使う。
 - プリプロセッサ機能を使うと、ベクトルを使った書き方も含めて、カーネル部は同じソースコードになるので便利である。

```
parameter( N=10000000 )
real pv(4,N)
do i = 1, N
  pv(1,i) = pv(1,i) + pv(2,i)
  pv(3,i) = pv(3,i) + pv(4,i)
end do
```



```
parameter( N=10000000 )
type particle
  real :: x, vx, y, vy
end type
type(particle) :: pv(N)
do i = 1, N
  pv(i)%x = pv(i)%x + pv(i)%vx
  pv(i)%y = pv(i)%y + pv(i)%vy
end do
```

```
parameter( N=10000000 )
real x(N), y(N), vx(N), vy(N)
do i = 1, N
  x(i) = x(i) + vx(i)
  y(i) = y(i) + vy(i)
end do
```

```
parameter( N=10000000 )
#define x(i) pv(1,i)
#define y(i) pv(3,i)
#define vx(i) pv(2,i)
#define vy(i) pv(4,i)
real pv(4,N)
do i = 1, N
  x(i) = x(i) + vx(i)
  y(i) = y(i) + vy(i)
end do
```

```
parameter( N=10000000 )
#define x(i) pv(i)%x
#define y(i) pv(i)%y
#define vx(i) pv(i)%vx
#define vy(i) pv(i)%vy
type particle
  real :: x, vx, y, vy
end type
type(particle) :: pv(N)
do i = 1, N
  x(i) = x(i) + vx(i)
  y(i) = y(i) + vy(i)
end do
```

Array of Structure of Array

- 連続アクセスできるデータ構造を持ち、かつキャッシュを有効に活用できるAoSタイプのメモリ割り付けになるAoSoAを考えるとときもある。

```
parameter( N=10000000 )
parameter( N4=N/4 )
real pv(4,4,N4)
do i = 1, N4
    pv(1:4,1,i) = pv(1:4,1,i) + pv(1:4,2,i)
    pv(1:4,3,i) = pv(1:4,3,i) + pv(1:4,4,i)
end do
```

```
parameter( N=10000000 )
parameter( N4=N/4 )
type particle
    real :: x(4), vx(4), y(4), vy(4)
end type
type(particle) :: pv(N4)
do i = 1, N4
    do j = 1, 4
        pv(i)%x(j) = pv(i)%x(j) + pv(i)%vx(j)
        pv(i)%y(j) = pv(i)%y(j) + pv(i)%vy(j)
    end do
end do
```

- 連続アクセスではないストライド付きでも、ストライド長が固定なら、AoSでもSIMD化できる場合もある。

```
parameter( N=10000000 )
real pv(4,N)
do i = 1, N, 4
    pv(1,i:i+3) = pv(1,i:i+3) + pv(2,i:i+3)
    pv(3,i:i+3) = pv(3,i:i+3) + pv(4,i:i+3)
end do
```

i:i+3のループは、
ストライド長16の
アクセスになる。

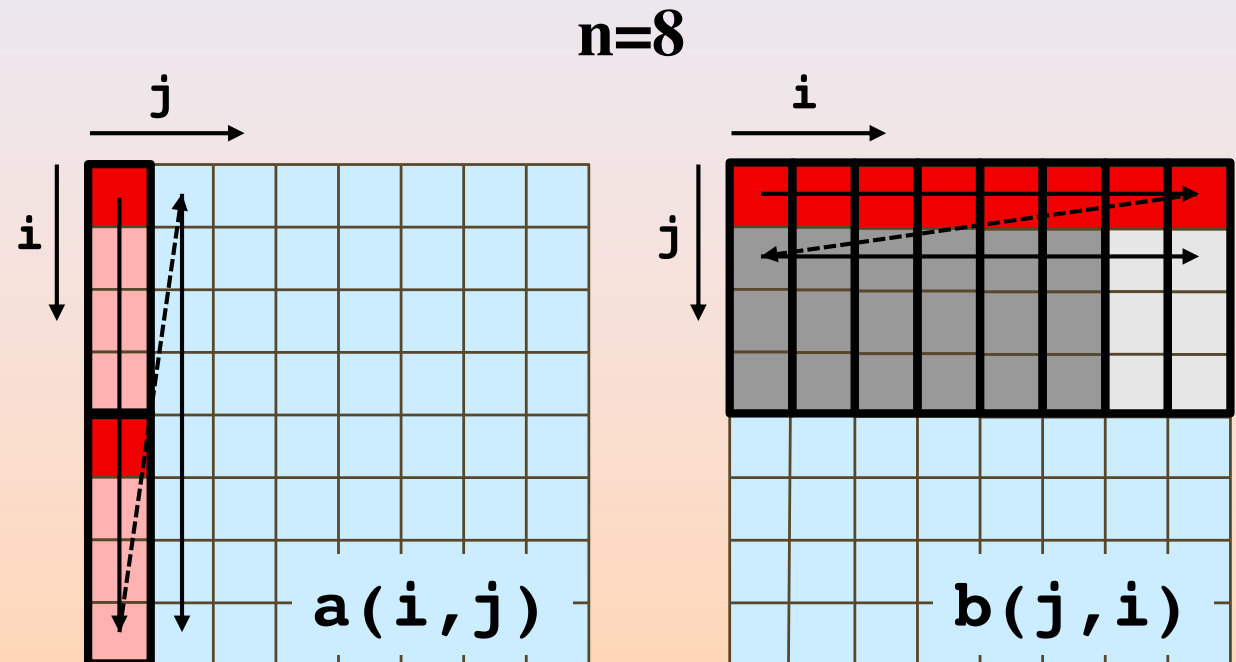
キャッシュチューニングのまとめ

- ❖ 計算速度は、キャッシュ容量、構造体の要素内容等に依存するため、基本的には、アーキテクチャ毎に最適解は異なる。
 - 一般的に、試行錯誤する必要がある。
- ❖ ベクトル機の場合は、Vectorで書く。
- ❖ SIMD相当の機能が使えない場合は、AoSで書く。
 - 期待された速度が達成できないなら、Vectorで書き直す。
- ❖ SIMD相当の機能が使える場合は. . .
 - SIMD化の障害要因（回帰参照、ループ内IF文等）があれば、取り除く。
 - AoSで書いてコンパイルし、SIMD化されているか確認する。
 - Vectorで書き直し、計算速度を比較する。
 - されていない場合は. . .
 - Vectorで書く。
 - AoSoAで書き直し、SIMD化されているか確認し、計算速度を比較する。

2次元配列の葛藤

- ループ順序を変えても，どちらかの2次元配列が連続アクセスにならない場合がある。

```
do j = 1, n
  do i = 1, n
    a(i,j) = a(i,j) + b(j,i)
  end do
end do
```



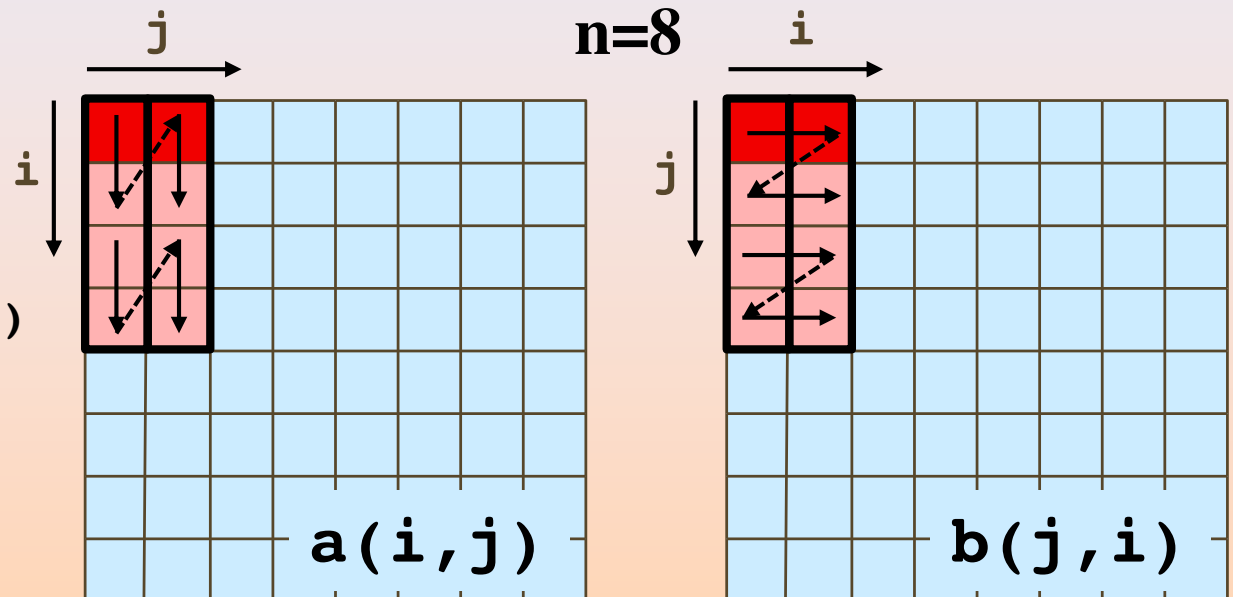
キャッシュライン数:2, ラインサイズ:16B, ライン当たり:二つ

2次元配列のブロック化

- 2次元配列をブロック化することにより、キャッシュミスを削減できる。
 - 最適なブロックの大きさは、キャッシュ容量、キャッシュライン数、ラインサイズ、ライン当たりの個数によって異なる。

```

lb = 2
do jb = 1, n, lb
  do ib = 1, n, lb
    do j = jb, jb+lb-1
      do i = ib, ib+lb-1
        a(i,j) = a(i,j) + b(j,i)
      end do
    end do
  end do
end do
    
```



キャッシュライン数:2, ラインサイズ:16B, ライン当たり:二つ

***ループ回数がブロック数で割りきれないときに注意！**

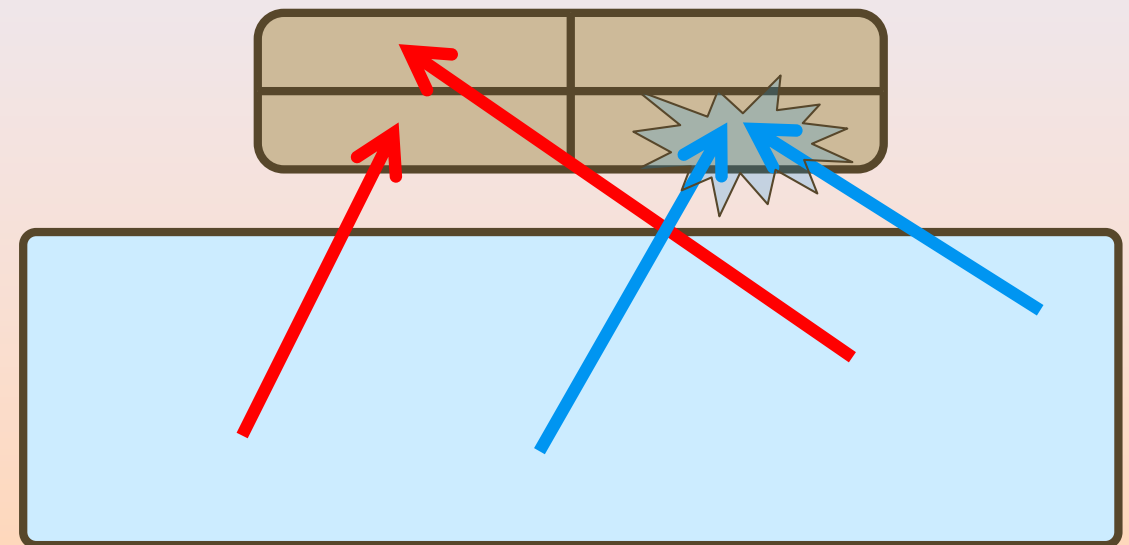
キャッシュコンフリクト

- 連続アクセスしていても、キャッシュミスが起こる場合がある。
 - ループ内に変数が多い場合は、すべての変数がキャッシュに載らないかもしれない。
 - キャッシュラインが衝突する場合は、同時にキャッシュに載らない。
 - ダイレクトマッピング方式, セットアソシアティブ方式, フルアソシアティブ方式
 - 起こるか起こらないかは、キャッシュ容量、キャッシュライン数等による。

```
do i = 1, n
  a(i) = a(i)+b(i)+c(i)+d(i)
end do
```

↓ ただし、ループオーバーヘッドとの兼ね合い。

```
do i = 1, n
  a(i) = a(i)+b(i)
end do
do i = 1, n
  a(i) = a(i)+c(i)
end do
do i = 1, n
  a(i) = a(i)+d(i)
end do
```



dimension a(1024,1024), b(1024,1024)

↓ ラインサイズだけずらす。

dimension a(1028,1024), b(1028,1024)

Formatted vs. Unformatted IO

- ✿ 書式付きファイル（テキストファイル）の入出力は、非常に遅いので、プログラム間でのデータの受け渡しには、書式なしファイル（バイナリファイル）を用いる。
 - Fortranのバイナリファイルには、レコードという概念がある。
 - レコード単位で、読み飛ばし（skip）や後戻り（backspace）をするため、データ長を収めたヘッダが使われる。



- ✿ ただし、異なるアーキテクチャのコンピュータ間でデータを受け渡すときは、以下に注意すること。
 - Big Endian (SPARC, POWER) vs. Little Endian (Intel)
 - 4バイトヘッダ vs. 8バイトヘッダ（超大レコード）
 - バイトカウント vs. ワードカウント（現在では、ほとんどないと思う。）
 - ほとんどの場合、コンパイラオプション、環境変数、OPEN文で対応できる。