プログラミングガイド Fortran編

2023年3月

V1.6

富士通株式会社

はじめに

- 当資料は、A64FX プロセッサ用アプリケーションをFortranでプログラミングおよびチューニングする際の基本的な情報を説明しています。
- 当資料と併せて以下も参照してください。
  - Fortran 使用手引書
  - C言語 使用手引書
  - C++言語 使用手引書
  - プロファイラ 使用手引書
  - プログラミングガイド　プログラミング共通編
  - プログラミングガイド　プロセッサ編
  - プログラミングガイド　チューニング編
- 当資料の記載においては、以下の文章を参考にしています。
  - A64FX 論理仕様書
  - A64FX®Microarchitecture Manual
  - ARM® Architecture Reference Manual (ARMv8 , ARMv8.1 , ARMv8.2 , ARMv8.3)
  - ARM® Architecture Reference Manual Supplement The Scalable Vector Extension
- 商標について
  - Linux® は、Linus Torvalds 氏の米国およびそのほかの国における登録商標または商標です。
  - Red Hat は、Red Hat,Inc. の米国およびそのほかの国における登録商標または商標です。
  - ARMは、ARM Ltd.の英国およびその他の国における登録商標です。
  - そのほかの会社名、製品名等の固有名詞は、各社の登録商標または商標です。
  - 本資料に記載されているシステム名、製品名等には、必ずしも商標表示(®、™)を付記していません

目次(1/5)：オプションと指示行

FUJITSU

目次(2/5)：オプションと指示子

FUJITSU

目次(3/5)：オプションと指示子

FUJITSU

目次(4/5)：翻訳、実行、注意点

FUJITSU

目次(5/5)：Fortran固有情報と性能チューニング例

FUJITSU

FUJITSU

代表的な最適化指定

- 推奨オプション
- 最適化レベル
- SIMD化
- ソフトウェアパイプライニング
- 命令スケジューリング
- プリフェッチ
- ループ展開
- ループ分割
- ループ融合
- ループ交換
- ループアンスイッチング
- 完全多重ループ化
- 高速ストア
- クローン最適化
- ロード命令の投機実行
- スタック割り付け
- 演算評価方法の変更
- 非正規化数の扱い
- 共通ブロックの境界調整
- 配列宣言の範囲
- 文字列操作ライブラリ
- セクタキャッシュ指定
- リンク時最適化
- HPCタグアドレスオーバライト
- データ割り付け
- 解析および検査
- OpenMP
- 最適化の調整

-Kfast(1/2)

オプション名	default
-Kfast	-

- 機能概要
  - ターゲットマシン上で高速に動作するオブジェクトプログラムを作成するための最適化オプションを誘導します。
- 効果
  - 標準的最適化のオプション指定を誘導するため、実行性能の向上が期待できます。
- ポイント
  - 実行性能を追求する場合、最適化オプションとして-Kfastオプションの指定を推奨します。
  - 実行性能が向上する可能性のある標準的最適化のオプション指定を誘導します。
- 注意
  - 演算精度誤差が発生する可能性のある最適化オプションを誘導するため、演算精度を重視する場合は演算精度誤差が発生するオプションを抑止する必要があります。

-Kfast(2/2)

FUJITSU

●誘導する最適化オプション

誘導されるオプション	機能概要
-O3	最適化レベル3で最適化を行う。
-Keval ◆	演算の評価方法を変更する最適化を行う。
-Kfp_contract ◆	Floating-Point Multiply-Add/Subtract演算命令を使用した最適化を行う。
-Kfp_relaxed ◆	単精度浮動小数点除算/倍精度浮動小数点除算/SQRT関数に対して、逆数近似演算を利用する最適化を行う。
-Kfz ◆	flush-to-zeroモード(演算結果またはソースオペランドが非正規化数の場合、それらを同符号の0で置き換える)を使用する。
-Kilfunc ◆	組込み関数/演算をインライン展開する。
-Kmfunc ◆	組込み関数/演算をマルチ演算関数に変換する最適化を行う。
-Komitfp	手続呼出しにおけるフレームポインタレジスタを保証しない最適化を行う。そのため、トレースバック情報は保証されない。
-Ksimd_packed_promotion	単精度実数型/4バイト整数型の配列要素のインデックス計算が4バイトの範囲を超えないと仮定して、packed-SIMD化を促進する。4バイトの範囲を超えていると、実行時異常終了や実行結果誤りが生じることがある。

◆：演算精度誤差が発生する可能性のある最適化オプション

-O

オプション名	default
-O[0|1|2|3]	-O指定なし：-O2 -Oのみ指定：-O3

- 機能概要
  - 用途に応じた最適化を選択します。

最適化レベル	機能概要	用途例
0	最適化は行いません。	プログラムのエラーチェックをしたい場合
1	基本的な最適化を行います。ループアンローリング、ソフトウェアパイプライニング、SIMD化などの実行性能が向上する最適化は行いません。 -O0より、オブジェクトサイズが小さくなり、実行時間は短くなる。	オブジェクトサイズが大きくなることを避けたい場合や、翻訳時間を短くしたい場合
2	-O1に加え、ループ系最適化、prefetch命令の生成、SIMD化、ループの先頭アライメント調整、末尾呼出しの最適化を行います。また、-O1の最適化を、最適化の余地がなくなるまで繰返し行います。-O1より翻訳時間が増加します。	精度誤差を生じさせない範囲で性能を求めたい場合（-Kfastを指定すると精度誤差が生じるため）
3	-O2に加え、多重ループのアンローリング、ループ交換促進のためのループ分割、ループアンスイッチング、CLONE最適化を行います。-O2よりさらに翻訳時間が増加します。	実行性能を追求したい場合

- 効果
  - 最適化レベルで用途に応じた最適化を選定できます。
- ポイント
  - 推奨オプションの-Kfastオプションを指定した場合は、-O3オプションが誘導されます。
  - 最適化オプションとしては-Kfastオプションの指定を推奨しますが、用途に応じて最適化レベルが選択できます。
- 注意
  - -O2以上では演算精度誤差などの最適化の副作用が発生する可能性があります。
  - -O2以上では翻訳時間が増加します。

SIMD(1/3)

FUJITSU

オプション名	default
-Ksimd[={1|2|auto}] -Knosimd	-O2以上：-Ksimd -O1以下：-Knosimd

最適化指示子
SIMD[({ALIGNED | UNALIGNED})] NOSIMD

- 機能概要
  - ループ内の命令に対してSIMD化するかどうかを指示します。

引数	SIMD化指定
1	• SIMD命令を出力します。
2	• -Ksimd=1の機能に加え、IF構文などを含むループに対して、SIMD命令を出力します。
auto (default)	• SIMD化するかどうかをコンパイラが自動的に判定します。

  - 特定のループに適用する場合は、SIMD指示子を使用します。

引数	SIMD化対象
なし (default)	• SIMD化することを指示します。
ALIGNED	• !OCL SIMDと等価です。
UNALIGNED	• !OCL SIMDと等価です。

  - SIMD指示子の引数は、旧製品とのソースコード互換を保つために指定可能になっています。
  - 演算の種類やループ構造によりSIMD化しない場合もあります。

SIMD(2/3)

FUJITSU

- 効果
  - 複数の同種の演算を同時実行するSIMD命令に変換することにより、実行性能が向上します。
- ポイント
  - -Ksimd={2|auto}オプションは、コストが高いループ内に以下の条件に合うIF構文が含まれる場合、そのループをSIMD化するために指定します。
    - IF構文の条件節の真率が高い
    - IF構文の条件節の真率は不明だが、ループ全体の実行文数に対して、IF構文のTHEN/ELSE節内の実行文数が少ない

  ループ内の分岐命令を無くすことで、ソフトウェアパイプライニングを促進させ、命令レベルの並列性を向上させることができます。
- 注意
  - -Ksimd={2|auto}オプションでは、IF構文内の命令を冗長実行するため、IF構文の真率によっては実行性能が低下する場合があります。また、-Kpreexオプションと同様にIF構文内の式を投機実行するため、プログラムの論理上実行されないはずの命令が実行され、エラーになる場合があります。

SIMD(3/3)

- 実用例
  - SIMD(ALIGNED)の誤った指定

    倍精度浮動小数点ストアは、16バイト境界にある場合にSIMD化が可能となります。
    しかし、16バイト境界にない倍精度浮動小数点ストアに対してSIMD(ALIGNED)指示子が指定された場合、実行時にSIGSEGVにより異常終了します。

```
   REAL(8) A(10)
   COMMON //N,A
!OCL SIMD(ALIGNED)
   DO I=1,10
      A(I) = ...
   END DO
```

配列AがCOMMONの途中の要素であるため16バイト境界に割りついていない場合があります。

```
   REAL(8) A(10)
   COMMON //A
!OCL SIMD(ALIGNED)
   DO I=2,10
      A(I) = ...
   END DO
```

配列Aが16バイト境界であっても、ループ内では2番目の要素から引用されるため、16バイト境界になりません。

-Ksimd_packed_promotion(1/2)

オプション名	default
-Ksimd_packed_promotion -Ksimd_nopacked_promotion	-Ksimd_nopacked_promotion

- 機能概要
  - 単精度実数型ならびに4バイト整数型の配列要素のインデックス計算が4バイトの範囲を超えないと仮定して、packed-SIMD化を促進します。
  - -Ksimd_packed_promotionオプションは、-Ksimdオプションが有効な場合に意味があります。
  - 配列が入れ子でアクセスされる場合は、インデックスとなる配列のアドレスは8バイトに変換されているため、インデックスに繋がる演算木に8バイト整数型の演算が出現してpacked-SIMD化が阻害される。
    次の条件をすべて満たす場合はpacked-SIMD化が可能になる。
    - ループ内に配列の入れ子アクセスがある
    - ループ内の配列データがすべて4バイト以下の要素型である
    - 配列のインデックスである配列も4バイトである
- 効果
  - packed-SIMD化が促進されるため、実行性能の向上を期待できます。
- 注意
  - 配列要素のインデックス計算が4バイトの範囲を超える場合、不当な領域をアクセスし、実行時異常終了または実行結果誤りが生じることがあります。
  - -Ksimd_packed_promotionオプションを指定しない場合、16SIMDにならず８SIMDになります。

-Ksimd_packed_promotion(2/2)

- 実用例
  - packed-SIMD化
    - C(I)が4バイトの範囲を超える場合、実行時異常終了または、実行結果誤りが生じることがあります。

```
SUBROUTINE TEST(A,B,C,D)
  USE_SIMFUNC
  INTEGER(KIND=4),DIMENSION(1:N) :: A,B
  INTEGER(KIND=4),DIMENSION(1:N) :: C,D
  !OCL SIMD(UNALIGNED)
  DO I=1,N
     A(C(I)) = B(D(I))
  ENDDO
END SUBROUTINE TEST
```

インデックスに4バイト型を許しているため、C(I)の4バイトアドレスを
8バイトに型変換せずに直接インデックスに繋げる最適化を実現して、
packed-SIMD化を促進する。

-Ksimd_reg_size={128|256|512}

オプション名	default
-Ksimd_reg_size={128|256|512}	-Ksimd_reg_size=512

- 機能概要
  - SVEのベクトルレジスタサイズをビット単位で指定します。
  - -KSVEオプションが有効な場合に、本オプションが有効になります。
  - 生成した実行可能プログラムは、本オプションで指定したサイズのSVEのベクトルレジスタを実装しているCPUアーキテクチャでのみ正常に動作します。
- 効果
  - 最適化が促進されるため、実行性能の向上を期待できます。
- 注意
  - 指定したサイズと、実行するCPUアーキテクチャのベクトルレジスタのサイズが異なる場合、実行時異常終了する可能性があります。また、実行時異常終了しなかった場合でも実行結果は保証されません。
  - 指定したサイズが、実行するCPUアーキテクチャのベクトルレジスタのサイズより小さいことが明らかな場合、prctl(2)システムコールなどを利用して有効なベクトルレジスタのサイズを設定する必要があります。

-Ksimd_reg_size=agnostic(1/2)

オプション名	default
-Ksimd_reg_size=agnostic	-

- 機能概要
  - SVEのベクトルレジスタを特定のサイズとみなさず翻訳を行い、実行時にSVEのベクトルレジスタサイズを決定する実行可能プログラムを作成することを指示します。
  - -KSVEオプションが有効な場合に、本オプションが有効になります。
  - 実行可能プログラムは、CPUのSVEのベクトルレジスタのサイズに関わらず実行可能です。
- 効果
  - 最適化が促進されるため、実行性能の向上を期待できます。
- 注意
  - -Ksimd_reg_size={128|256|512}オプションを指定した場合と比べて、実行性能が低下する場合があります。

-Ksimd_reg_size=agnostic(2/2)

- 実用例
  - -Ksimd_reg_size=agnostic指定時、-Ksimd_reg_size=512(デフォルト)とは同様の最適化が行われないことがあります。その場合、実行性能が低下する可能性があります。

-Ksimd_reg_size=agnostic指定時

```
          <<< Loop-information Start >>>
          <<<  [PARALLELIZATION]
          <<<    Standard iteration count: 843
          <<<  [OPTIMIZATION]
          <<<    COLLAPSED
          <<<    SIMD(VL: AGNOSTIC; VL: 2 in 128-bit)
          <<<    PREFETCH(HARD) Expected by compiler :
          <<<      B, A
          <<< Loop-information  End >>>
 6 1  pp   2v     DO J=1,N
          <<< Loop-information Start >>>
          <<<  [OPTIMIZATION]
          <<<    COLLAPSED
          <<< Loop-information  End >>>
 7 2   p   2       DO I=1,N
 8 2   p   2v        A(I,J) = A(I,J) + C * B(I,J)
 9 2   p   2v      ENDDO
10 1   p         ENDDO
```

agnostic選択時、ソフトウェアパイプライニングの効果がないと判断され、ソフトウェアパイプライニングが抑止されました。

-Ksimd_reg_size=512(デフォルト)指定時

```
                <<< Loop-information Start >>>
                <<<  [PARALLELIZATION]
                <<<    Standard iteration count: 843
                <<<  [OPTIMIZATION]
                <<<    COLLAPSED
                <<<    SIMD(VL: 8)
                <<<    SOFTWARE PIPELINING
                         (IPC: 3.00, ITR: 192, MVE: 7, POL: S)
                <<<    PREFETCH(HARD) Expected by compiler :
                <<<      B, A
                <<< Loop-information  End >>>
 6     1  pp   2v       DO J=1,N
                <<< Loop-information Start >>>
                <<<  [OPTIMIZATION]
                <<<    COLLAPSED
                <<< Loop-information  End >>>
 7     2   p   2          DO I=1,N
 8     2   p   2v           A(I,J) = A(I,J) + C * B(I,J)
 9     2   p   2v         ENDDO
10     1   p           ENDDO
11                   END SUBROUTINE
```

SIMD_USE_MULTIPLE_STRUCTURES(1/2)

FUJITSU

オプション名	default
-Ksimd_use_multiple_structures -Ksimd_nouse_multiple_structures	-Ksimd_use_multiple_structures

最適化指示子
SIMD_USE_MULTIPLE_STRUCTURES SIMD_NOUSE_MULTIPLE_STRUCTURES

- 機能概要
  - SIMD化の際、SVEのLoad Multiple Structures命令およびStore Multiple Structures命令を利用するかどうかを指示します。
  - -Ksimdオプションおよび-KSVEオプションが有効な場合に、本オプションが有効になります。
- 効果
  - SIMD化対象のロードおよびストアにLoad Multiple Structures命令およびStore Multiple Structures命令を利用することで、実行性能の向上を期待できます。
- ポイント
  - 詳細は「プログラミングガイド チューニング編 Multiple Structures命令の活用」を参照してください。
- 注意
  - データのアライメントによっては性能劣化する場合があります。

SIMD_USE_MULTIPLE_STRUCTURES(2/2)

FUJITSU

- 実用例

```
!OCL SIMD_USE_MULTIPLE_STRUCTURES
  DO I=1,N
    A(I) = B(1,I) + B(2,I) + B(3,I)
  ENDDO
```

対象となるループにおいて、SVEのLoad Multiple Structures命令およびStore Multiple Structures命令を利用します。

アセンブラでのSVEのLoad Multiple Structures命令

```
LD3D    {Z0.D, Z1.D, Z2.D}, P0/Z, [X6, 0, MUL VL]       //  "b"
```

B(1,I)、B(2,I)、B(3,I)のロードが、構造体命令の一命令(LD3D)で実現されます。

-Ksimd_uncounted_loop

オプション名	default
-Ksimd_uncounted_loop -Ksimd_nouncounted_loop	-Ksimd_nouncounted_loop

- 機能概要
  - 繰り返し数が不明なループ（DO WHILEループ、DO UNTILループおよび、ループを終了する文を含むDOループ内の命令）に対して、SIMD命令を利用したオブジェクトを生成するかどうかを指示します。
  - -KsimdオプションおよびKSVEオプションが有効な場合に、本オプションが有効になります。
- 効果
  - SIMD命令を利用したオブジェクトが生成されるため、実行性能の向上を期待できます。
- 注意
  - 繰り返し数が少ない場合には、実行性能が低下する場合があります。

-Kloop_part_simd

オプション名	default
-Kloop_part_simd -Kloop_nopart_simd	-Kloop_nopart_simd

- 機能概要
  - ループ中にSIMD化できる部分とSIMD化できない部分が存在する場合、そのループを分割して部分的にSIMD化するかどうかを指示します。
  - 対象ループは最内ループです。
  - -Ksimdオプションが有効な場合に、本オプションが有効になります。
- 効果
  - SIMD命令を利用したオブジェクトが生成されるため、実行性能の向上を期待できます。
- 注意
  - ループを分割することにより、データ受け渡しのための作業領域が多過ぎると、実行時間が増加する場合があります。
- 実用例

オリジナルソース

```
!OCL LOOP_PART_SIMD
  DO I=2,N
    A(I) = A(I) - B(I) + LOG(C(I)) ! SIMD化可能
    D(I) = D(I-1) + A(I)           ! SIMD化不可
  ENDDO
```

最適化後のソースイメージ

```
  DO I=2,N
    A(I) = A(I) - B(I) + LOG(C(I)) ! SIMD実行
  ENDDO
  DO II=2,N
    D(II) = D(II-1) + A(II)        ! 非SIMD実行
  ENDDO
```

分割されたIのループのみSIMD化されます。

NOVREC/NORECURRENCE(1/2)

最適化指示子
NOVREC[(array1[,array2]...)]
NORECURRENCE [(array1[,array2]...)]

- 機能概要
  - NOVREC指定子は、ループ中に回帰演算となる配列がないことを指示します。
    ループ中の配列に対しSIMD命令を利用しますが、演算の種類やループ構造によりSIMD命令が利用されない場合もあります。また、arrayには、ポインタ配列を指定することはできません。
  - NORECURRENCE指定子は、DOループ内の演算対象となる配列の要素が、繰り返しを跨いで定義引用されないことを指示します。これにより、配列の定義引用順序が不明で最適化できなかったDOループを、次のような最適化の対象にします。
    - ループスライス(自動並列化)
    - SIMD化
    - ソフトウェアパイプライニング
  - array1、array2、... は配列名です。
- 効果
  - SIMD命令を利用したオブジェクトが生成されるため、実行性能の向上を期待できます。
- 注意
  - ループ内で使用される配列が回帰的データの場合、NOVREC指示子による実行結果は保証されません。
  - NORECURRENCE指示子が繰返し数に依存する配列に対して誤って指定された場合、実行結果は保証されません。

NOVREC/NORECURRENCE(2/2)

FUJITSU

- 実用例
  - NOVREC指示子

```
REAL A(20),B(20)
!OCL NOVREC
  DO I=2,10
    A(I) = A(I+N) + 1
    B(I) = B(I+M) + 2
  ENDDO
```

ループ中の配列がすべて回帰演算とならないことを指示します。配列AおよびBの演算についてSIMD命令が利用されます。

```
REAL A(20),B(20)
!OCL NOVREC(A)
  DO I=2,10
    A(I) = A(I+N) + 1
    B(I) = B(I) + 2
  ENDDO
```

データ依存が不明な配列Aに対して、回帰演算でないことを指示します。配列AおよびBの演算についてSIMD命令が利用されます。

  - NOVREC指示子とNORECURRENCE指示子の使い分け

SIMD化を促進するために、配列名を個別に指定しない使用方法もあります。

```
!OCL NOVREC
DO I=1,N
   A(I) = A(I+M)+1
   B(I) = B(I)+1
        :
ENDDO
```

変数Mの値により次のように使い分けます。

- Mは1以上の値になる可能性がある場合は、NOVREC指示子
- Mは0のみの場合は、NORECURRENCE指示子

SWP

オプション名	Default
-Kswp -Knoswp	-O2以上：-Kswp -O1：-Knoswp

最適化指示子
SWP NOSWP

- 機能概要
  - ソフトウェアパイプライニングの最適化を行います。
  - -Kswpオプションを、-Kswp_weakオプションまたは-Kswp_strongオプションと同時に指定した場合、後に指定した方が有効となります。
  - -O2以上で有効になります。
  - 次の場合には最適化を行いません。
    - 最適化の効果が期待できないと判断した場合
- 効果
  - ループ内の命令ができるだけ並列実行されるように、命令を並べ替えるため、実行性能の向上を期待できます。
- ポイント
  - SWP指示子は-Kswp_strongオプションと同等の効果となります。
- 注意
  - ループの任意の繰り返し数における実行文とそれ以降の繰り返し数における実行文を重ね合わせた命令スケジューリングを行い、ループの形状を変更するため、ループには十分な繰返し数が必要になります。
  - 繰返し数が変数のループに対してソフトウェアパイプライニングが適用された場合、オブジェクトプログラムの大きさが増加します。

-Kswp_weak/-Kswp_strong(1/2)

オプション名	default
-Kswp_weak	-
-Kswp_strong	-

最適化指示子
SWP_WEAK

- 機能概要
  - ソフトウェアパイプライニングの適用方法を指示します。

オプション	ソフトウェアパイプライニングの適用方法
-Kswp_weak	• ソフトウェアパイプライニングを調整し、ループ内の実行文の重なりを小さくします。
-Kswp_strong	• ソフトウェアパイプライニング適用の条件を緩和し、ループボディが大きくても強制的にソフトウェアパイプライニングを適用します。

  - -Kswpオプション、-Kswp_weakオプション、-Kswp_strongオプションを同時に指定した場合、後に指定した方が有効となります。
  - その他の機能や注意事項は、-Kswpオプションと同じです。

-Kswp_weak/-Kswp_strong(2/2)

FUJITSU

- 効果
  - ソフトウェアパイプライニングが適用されたループを実行することで必要なループの繰返し数が小さくなるため、実行性能の向上を期待できます。

オプション	効果
-Kswp_weak	• 翻訳時にループの繰返し数が不明でかつ、ループの繰返し数が小さい場合に最適化の効果が期待できます。
-Kswp_strong	• ループの繰り返し数が大きい場合に最適化の効果が期待できます。

- ポイント
  - -Kswp_weakオプションを指定した場合と指定しなかった場合で、ソフトウェアパイプライニングの通るルートのITRが変わります。
- 注意

オプション	注意
-Kswp_weak	• 実行文の重なりが小さくなるため、実行性能が低下する場合があります。
-Kswp_strong	• 翻訳メモリや翻訳時間が大幅に増加する場合があります。

-Kswp_freg_rate/-Kswp_ireg_rate/-Kswp_preg_rate(1/2)

オプション名	default
-Kswp_freg_rate=n	-Kswp_freg_rate=100
-Kswp_ireg_rate=n	-Kswp_ireg_rate=100
-Kswp_preg_rate=n	-Kswp_preg_rate=100

最適化指示子
SWP_FREG_RATE(n)
SWP_IREG_RATE(n)
SWP_PREG_RATE(n)

- 機能概要
  - ソフトウェアパイプライニングで、それぞれ浮動小数点レジスタおよびSVE、整数レジスタ、プレディケートレジスタが使用可能な割合(百分率)を指示します。
    引数が100の場合はすべてのレジスタ（fregであれば32本）が使用可能として、引数が200の場合は100の場合の2倍のレジスタ（fregであれば64本）が使用可能と仮定してソフトウェアパイプライニングします。
  - -O2以上で有効です。
- 効果
  - レジスタ数に関する条件を変更することで、ソフトウェアパイプライニングの適用を調整できます。
- ポイント
  - レジスタが不足するためソフトウェアパイプライニングが適用されない場合、100より大きな整数値を指定することで、spill/fillがは発生しますが、強制的にソフトウェアパイプライニングが適用できることがあります。
- 注意
  - レジスタのメモリへの退避・復元命令が変化し、実行性能が低下する場合があります。

-Kswp_freg_rate/-Kswp_ireg_rate/-Kswp_preg_rate(2/2)

FUJITSU

● 実用例

浮動小数点レジスタが不足して、ソフトウェアパイプライニングが適用できません。

```
 7 1              DO J=1,N
 8 1            !OCL UNROLL(8)
           <<< Loop-information Start >>>
           <<<  [OPTIMIZATION]
           <<<    SIMD(VL: 8)
           <<<    PREFETCH(HARD) Expected by compiler :
           <<<      D, C, A
           <<< Loop-information  End >>>
 9 2       8v       DO I=1,N
10 2       8v         A(I,J) = B(J,I) + C(I) / D(I,J)
11 2       8v       ENDDO
12 1              ENDDO

jwd8666o-i  “a.f90", line 9: 浮動小数点レジスタが不足しているため、ソフトウェアパイプライニングを適用できません。
```

SWP_FREG_RATE指示子で使用できる浮動小数点レジスタを増やし、ソフトウェアパイプライニングを促進させます。

```
           <<< Loop-information Start >>>
           <<<  [OPTIMIZATION]
           <<<    PREFETCH(HARD) Expected by compiler :
           <<<      D, C, A
           <<< Loop-information  End >>>
 7 1             DO J=1,N
 8 1           !OCL UNROLL(8)
 9 1           !OCL SWP_FREG_RATE(120)
           <<< Loop-information Start >>>
           <<<  [OPTIMIZATION]
           <<<    SIMD(VL: 8)
           <<<    SOFTWARE PIPELINING
                       (IPC: 2.52, ITR: 192, MVE: 2, POL: S)
           <<<    PREFETCH(HARD) Expected by compiler :
           <<<      C, D, A
           <<<    SPILLS :
           <<<      GENERAL   : SPILL 0  FILL 0
           <<<      SIMD&FP   : SPILL 0  FILL 0
           <<<      SCALABLE  : SPILL 4  FILL 16
           <<<      PREDICATE : SPILL 0  FILL 0
           <<< Loop-information  End >>>
10 2       8v       DO I=1,N
11 2       8v         A(I,J) = B(J,I) + C(I) / D(I,J)
12 2       8v       ENDDO
13 1             ENDDO

jwd8204o-i  “a.f90", line 10: ループにソフトウェアパイプライニングを適用しました。
jwd8205o-i  “a.f90", line 10: ループの繰返し数が192回以上の時、ソフトウェアパイプライニングを適用したループが実行時に選択されます。
```

SWP_POLICY

オプション名	Default
-Kswp_policy={auto|small|large}	-Kswp_policy=auto

最適化指示子
SWP_POLICY({AUTO|SMALL|LARGE})

- 機能概要
  - ソフトウェアパイプライニングで使用する命令スケジューリングアルゴリズムの選択基準を指示します。
  - ソフトウェアパイプライニングは、-Kswpオプション、-Kswp_weakオプション、-Kswp_strongオプションまたは、それぞれのオプションに対応した指示子が有効な場合に行われます。

引数	アルゴリズムの選択基準
auto/AUTO	• ループ毎に命令スケジューリングアルゴリズムを自動t的に選択します。
small/SMALL	• 小さなループ(例えば、必要レジスタ数が少ないループ)に適した命令スケジューリングアルゴリズムを使用します。
large/LARGE	• 大きなループ(例えば、必要レジスタ数が多いループ)に適した命令スケジューリングアルゴリズムを使用します。

- 効果
  - ソフトウェアパイプライニングが適用されるため、実行性能の向上を期待できます。
- ポイント
  - アルゴリズムが自動的に選択されるため、auto/AUTOから試してみてください。

-Ksch_pre_ra/-Ksch_post_ra

FUJITSU

オプション名	default
-Ksch_pre_ra -Ksch_nopre_ra	-O1以上：-Ksch_pre_ra -O0以下：-Ksch_nopre_ra
-Ksch_post_ra -Ksch_nopost_ra	-O1以上：-Ksch_post_ra -O0以下：-Ksch_post_ra

- 機能概要
  - レジスタ割付け前、レジスタ割り付け後に、命令スケジューリングを行います。
- 効果
  - strong prefetchとすることにより、実行性能の向上を期待できます。
- ポイント
  - リスティングでspill/fillがある場合、-Knosch_post_raオプションを指定すると実行性能が向上する可能性があります。
- 注意
  - -Ksch_pre_raオプションは、レジスタをメモリに退避・復元する命令が増える場合があり、実行性能が低下する可能性があります。一般的には、最適化時に適切に処理されるので、ユーザーが意識する必要はありません。
  - -Ksch_post_raオプションは、レジスタをメモリに退避・復元する命令は増加しません。

PREFETCH_SEQUENTIAL

オプション名	default
-Kprefetch_sequential[={auto|soft}] -Kprefetch_nosequential	-O2以上：-Kprefetch_sequential -O1以下：-Kprefetch_nosequential

最適化指示子
PREFETCH_SEQUENTIAL[({AUTO|SOFT})] PREFETCH_NOSEQUENTIAL

- 機能概要
  - ループ内で使用される連続的にアクセスされる配列データに対して、prefetch命令を使用したオブジェクトを生成します。
  - -O1以上で有効になります。

引数	プリフェッチの選択方法
auto （default）	• ハードウェアプリフェッチを利用するか、prefetch命令を出力するかをコンパイラが自動的に選択します。
soft	• ハードウェアプリフェッチを利用せずに、prefetch命令を出力します。

- 効果
  - 選択したプリフェッチにより、実行性能の向上を期待できます。
- 注意
  - ループのキャッシュ効率、分岐の有無または添字の複雑さによって、実行性能が低下することがあります。

PREFETCH_CACHE_LEVEL

オプション名	default
-Kprefetch_cache_level={1|2|all}	-Kprefetch_cache_level=all

最適化指示子
PREFETCH_CACHE_LEVEL(1|2|all)

- 機能概要
  - どのキャッシュレベルにデータをプリフェッチするかを指示します。

引数	プリフェッチするキャッシュレベル
1	• データを1次キャッシュにプリフェッチすることを指示します。
2	• データを2次キャッシュにプリフェッチすることを指示します。
all (default)	• データをすべての階層のキャッシュにプリフェッチすることを指示します。 • 各階層へのprefetch命令を組み合わせることにより、より高度なプリフェッチを実現することができます。

- 効果
  - 指定したキャッシュへのプリフェッチにより、実行性能の向上を期待できます。
- ポイント
  - 自動プリフェッチ機能が有効になるため、autoから試してみてください。
  - 詳細説明は「プログラミングガイド 共通編 プリフェッチ」、使用方法は「プログラミンンぐガイド　チューニング編 データアクセス待ち（レイテンシの隠蔽）」を参照してください。
- 注意
  - 適切なキャッシュレベルを指定しないと、実行性能が低下することがあります。

PREFETCH_STRIDE(1/2)

オプション名	default
-Kprefetch_stride[={soft|hard_auto| hard_always}] -Kprefetch_nostride	-Kprefetch_nostride

最適化指示子
PREFETCH_STRIDE[({SOFT | HARD_AUTO | HARD_ALWAYS})] PREFETCH_NOSTRIDE

- 機能概要
  - ループ内で使用されるキャッシュのラインサイズよりも大きなストライドでアクセスされる配列データに対して、prefetch
    命令を使用したオブジェクトを生成します。
  - -Kprefetch_stride=softオプションおよび-Kprefetch_nostrideオプションは、-O1以上で有効になります。
  - -Kprefetch_stride=hard_autoオプションおよび-Kprefetch_stride=hard_alwaysオプションは、-Khpctagオプションおよび-O1以上で有効になります。
  - 配列データに対してのプリフェッチの方法が指定できます。

引数	プリフェッチ方法
soft (default)	• prefetch命令を生成して、プリフェッチを実施します。
hard_auto	• ハードウェアストライドプリフェッチャーを利用して、プリフェッチを実施します。 • キャッシュ上にないデータのみプリフェッチするようにストライドプリフェッチャーを設定します。
hard_always	• ハードウェアストライドプリフェッチャーを利用して、プリフェッチを実施します。 • 常にプリフェッチを行うようにストライドプリフェッチャーを設定します。

- 効果
  - 指定したプリフェッチにより、実行性能の向上を期待できます。

PREFETCH_STRIDE(2/2)

- 注意
  - ループのキャッシュ効率、分岐の有無または添字の複雑さによって、実行性能が低下することがあります。
- 実用例

```
!OCL PREFETCH_STRIDE(SOFT)
  DO I=1,N,K
    A(I) = B(I)
  ENDDO
```

対象となるループにおいて、prefetch命令を生成します。

PREFETCH_STRONG/PREFETCH_STRONG_L2(1/2)

オプション名	default
-Kprefetch_strong -Kprefetch_nostrong	-Kprefetch_strong
-Kprefetch_strong_L2 -Kprefetch_nostrong_L2	-Kprefetch_strong_L2

最適化指示子
PREFETCH_STRONG PREFETCH_NOSTRONG PREFETCH_STRONG_L2 PREFETCH_NOSTRONG_L2

- 機能概要
  - 1次キャッシュ、２次キャッシュに対して生成されるprefetch命令をstrong prefetchとします。
  - 次のいずれかのオプション
    - -Kprefetch_sequential
    - -Kprefetch_stride
    - -Kprefetch_indirect

  が有効かつ、
  -Kprefetch_cache_level=allオプションの場合は、-Kprefetch_strongオプション、-Kprefetch_strong_L2オプション共に、
  −Kprefetch_cache_level=1オプションの場合は、-Kprefetch_strongオプションが、
  -Kprefetch_cache_level=2オプションの場合は、-Kprefetch_strong_L2オプションが有効になります。

PREFETCH_STRONG/PREFETCH_STRONG_L2(2/2)

- 効果
  - strong prefetchとすることにより、実行性能の向上を期待できます。
- 注意
  - prefetch命令には「Strong属性」と「Weak属性」があります。「Strong属性」の場合、ハードウェアは可能な限りプリフェッチのリクエストを完了させようとします。「Weak属性」の場合、ハードウェアの資源に余裕がなければプリフェッチのリクエストは削除されます。
  - 京では1次キャッシュのデフォルトは「Weak属性」でしたが、富岳では「Strong属性」に変わっています。
- 実用例

```
!OCL PREFETCH_STRONG
  DO I=1,N
    A(I) = B(I)
  ENDDO
```

対象となるループで生成される1次キャッシュのprefetch命令はstrong prefetchとなります

PREFETCH_CONDITIONAL

FUJITSU

オプション名	default
-Kprefetch_conditional -Kprefetch_noconditional	-Kprefetch_noconditional

最適化指示子
PREFETCH_CONDITIONAL PREFETCH_NOCONDITIONAL

- 機能概要
  - IF構文やCASE構文に含まれるブロックの中で使用される配列データに対してprefetch命令を生成します。
  - 次のいずれかのオプションが有効な場合に、本オプションが有効になります。
    - -Kprefetch_sequential
    - -Kprefetch_stride
    - -Kprefetch_indirect
- 効果
  - 配列データに対してのプリフェッチにより、実行性能の向上を期待できます。
- 注意
  - ループのキャッシュ効率、分岐の有無または添字の複雑さによって、実行性能が低下することがあります。

PREFETCH_INDIRECT

FUJITSU

最適化指示子

PREFETCH_INDIRECT
PREFETCH_NOINDIRECT

- 機能概要
  - ループ内で使用される間接的にアクセス(リストアクセス)される配列データに対して、prefetch命令を生成します。
- 効果
  - 配列データに対してのプリフェッチにより、実行性能の向上を期待できます。
- 注意
  - ループのキャッシュ効率、分岐の有無または添字の複雑さや、プリフェッチ対象となる配列データが連続であったり、同じ値であったりする場合、prefetch命令のコストにより実行性能が低下することがあります。
- 実用例

```
!OCL PREFETCH_INDIRECT
  DO I=1,N
    A(INDX(I)) = B(INDX(I))
  ENDDO
```

ループ内で使用される間接的にアクセス(リストアクセス)される配列データに対して、prefetch命令を生成します。

PREFETCH_LINE/PREFETCH_LINE_L2

最適化指示子
PREFETCH_LINE(n)
PREFETCH_LINE_L2(n)

- 機能概要
  - 1次キャッシュ、2次キャッシュに対してprefetch命令を生成する際、nキャッシュライン先に該当するデータをプリフェッチの対象とします。
  - nは1から100までの整数値を指定することができます。
- 効果
  - 配列データに対してのプリフェッチにより、実行性能の向上を期待できます。
- ポイント
  - ハードウェアのプリフェッチは1次キャッシュで最大6ライン、2次キャッシュで最大40ラインと比較的遠くをプリフェッチするため、本指定では近くを指定することで効果を得られることがあります。
- 実用例

```
!OCL PREFETCH_LINE(4)
  DO I=1,N
    A(I) = B(I)
  ENDDO
```

ループ内で生成される1次キャッシュのprefetch命令は、4キャッシュライン先のデータをプリフェッチの対象とします。
1キャッシュラインのサイズは256バイトです。倍精度実数型(8バイト)でアクセスする場合、256/8=32要素分となります。
左記例では、4キャッシュライン*32要素=128要素先のデータをプリフェッチします。

PREFETCH_READ/PREFETCH_WRITE(1/2)

FUJITSU

最適化指示子
PREFETCH_READ(name[,level={ 1 | 2 }][,strong={ 0 | 1 }])
PREFETCH_WRITE(name[,level={ 1 | 2 }][,strong={ 0 | 1 }])

- 機能概要
  - PREFETCH_READは参照しているデータに対して、PREFETCH_WRITEは定義されているデータに対してprefetch 命令を生成します。
  - nameにはプログラム中で参照、定義されているデータ(配列要素または部分配列)を指定します。
    要素ごとの指定のほか、ベクトル指定をすることができます。
  - levelを指定することにより、どのキャッシュレベルにデータをプリフェッチするかを指示します。
    1は１次キャッシュ、2は２次キャッシュを意味します。デフォルトは、level=1です。
  - strongを指定することにより、strong prefetchとするかどうかを指示します。
    - strong=0が指定された場合はstrong prefetchしない
    - strong=1が指定された場合はstrong prefetchとする
    - デフォルトは、strong=1です。
- 効果
  - 配列データに対してのプリフェッチにより、実行性能の向上を期待できます。
- ポイント
  - 参照かつ定義されているデータに関しては、PREFETCH_WRITE指示子を使用します。
  - nameをベクトル指定した場合、コンパイラは動的に1キャッシュラインに1つずつプリフェッチを生成します。
    また、プリフェッチ命令を複数同時に生成できるため、要素ごとの指定よりも性能向上することがあります。

PREFETCH_READ/PREFETCH_WRITE(2/2)

FUJITSU

- 実用例

要素ごとに指定

```
DO J=1,N
  DO I=1,ISIZE
!OCL PREFETCH_WRITE(A(I,J+1),level=1)
!OCL PREFETCH_READ(B(I,J+1),level=1)
!OCL PREFETCH_READ(C(I,J+1),level=1)
        A(I,J) = B(I,J) + SCALAR * C(I,J)
  ENDDO
ENDDO
```

A(I,J+1)、B(I,J+1)、C(I,J+1)に対してprefetch命令を生成します。

ベクトルで指定(推奨)

```
DO J=1,N
!OCL PREFETCH_WRITE(A(1:ISIZE,J+1),level=1)
!OCL PREFETCH_READ(B(1:ISIZE,J+1),level=1)
!OCL PREFETCH_READ(C(1:ISIZE,J+1),level=1)
  DO I=1,ISIZE
    A(I,J) = B(I,J) + SCALAR * C(I,J)
  ENDDO
ENDDO
```

A(1:ISIZE,J+1)、B(1:ISIZE,J+1)、C(1:ISIZE,J+1)に対してprefetch命令を生成します。
要素ごとの指定に比べ、ループの外側でベクトル指定することで最内ループの命令数を削減できるため性能向上することがあります。

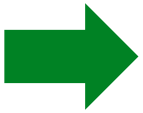
ループ展開

- ループ展開には次の2種類あります
  - アンローリング(unroll)
  - ストライピング(striping)

アンローリングとストライピングの違い

```
DO I=1,N
  A(I) = B(I) + C(I)
ENDDO
```

展開の方法が違う



赤：1展開目
青：2展開目

■ **アンローリング2展開のイメージ**

```
DO I=1,N,2
  TMP_B1 = B(I)
  TMP_C1 = C(I)
  TMP_A1 = TMP_B1 + TMP_C1
  A(I) = TMP_A1
  TMP_B2 = B(I+1)
  TMP_C2 = C(I+1)
  TMP_A2 = TMP_B2 + TMP_C2
  A(I+1) = TMP_A2
ENDDO
```

■ **ストライピング2展開のイメージ**

```
DO I=1,N,2
  TMP_B1 = B(I)
  TMP_B2 = B(I+1)
  TMP_C1 = C(I)
  TMP_C2 = C(I+1)
  TMP_A1 = TMP_B1 + TMP_C1
  TMP_A2 = TMP_B2 + TMP_C2
  A(I) = TMP_A1
  A(I+1) = TMP_A2
ENDDO
```

- ポイント
  - 一般的にはアンローリングを推奨します。アンローリングでは命令の並び替えは行わないため、展開数が2,3と少なく、確実に命令の並び替えを行いたい場合にはストライピングが有効です。
  - -Kstripingオプションと-Kunrollオプションを同時に指定した場合、後に指定した方が有効となります。
- 注意
  - アンローリングよりストライピングの方が使用レジスタ数が増加するため、ストライプ長を多くすると実行性能が低下する場合があります。

UNROLL(1/3)

オプション名	default
-Kunroll[=n] -Knounroll	-O2以上：-Kunroll -O1以下：-Knounroll

最適化指示子
UNROLL[(n)] UNROLL('full')

- 機能概要
  - DOループ内のすべての実行文をループ内へn重に展開して、ループの繰返し数をn分の1にします。
  - -Ksimd[=level]オプションが有効でループ内がSIMD化された場合、実行文をSIMD長*n重に展開して、ループの
    繰返し数をSIMD長*n分の1にします。
  - ループ外からの飛込みもループ外への飛出しもないループが対象です。
  - nの指定を省略した場合、コンパイラが自動的に最良な値を決定します。
  - 特定のDOループに適用する場合は、UNROLL指示子を使用します。
- 効果
  - 繰返し数が縮小されループの繰返しによるオーバヘッドが少なくなります。また、多重に展開された実行文がループ内で一体になって最適化されるため、実行性能の向上が期待できます。

UNROLL(2/3)

FUJITSU

- ポイント
  - 通常ループはループ内の実行文を重ね合わせた命令スケジューリングを行うソフトウェアパイプライニングでアンロールを実施します。
    リダクション演算があるループはアンローリングとソフトウェアパイプライニングでアンロールを実施します。
  - ループアンローリングはSIMD化前後で動作します。

SIMD化前	• 外側ループでSIMD化やソフトウェアパイプライニングを適用するために、内側ループの繰り返し数が少ない場合に内側ループをフルアンローリングします。
SIMD化後	• 共通式などの最適化の促進のために、内側ループをアンローリングまたはフルアンローリングします。

    -Kunroll[=n]オプションまたはUNROLL指示子が有効なループでは、ループアンローリングはSIMD化前後で動作しますが、内側ループの繰り返し数が少ない場合はソフトウェアパイプライニングでフルアンローリングします。
  - 配列代入および配列式を含む文(配列記述)は、DOループに展開され、ループアンローリングの対象になります。
  - 命令スケジューリング/ソフトウェアパイプライニングが促進されることで、演算待ちを解消できる場合があります。
- 注意
  - ループ内の文が多重に展開されるので、オブジェクトプログラムの大きさが増加します。

UNROLL(3/3)

FUJITSU

- 実用例
  - アンロールのみ（実行性能が向上しない）

オリジナルソース

```
!OCL UNROLL(2)
  DO I=1,N
    A(I) = B(I) + C(I)
  ENDDO
```

最適化後のソースイメージ

```
DO I=1,N,2
  TMP_B1 = B(I)
  TMP_C1 = C(I)
  TMP_A1 = TMP_B1 + TMP_C1
  A(I) = TMP_A1
  TMP_B2 = B(I+1)
  TMP_C2 = C(I+1)
  TMP_A2 = TMP_B2 + TMP_C2
  A(I+1) = TMP_A2
ENDDO
```

アンロールのみではGatherが出力されて実行性能は向上しません。

  - SIMD化＋アンロール（実行性能が向上する）

オリジナルソース

```
!OCL UNROLL(2)
  DO I=1,N
    A(I) = B(I) + C(I)
  ENDDO
```

SIMD化(VL:8)有効時の最適化後のソースイメージ

```
DO I=1,N,15
  TMP_B1(1:7) = B(I:I+7)
  TMP_C1(1:7) = C(I:I+7)
  TMP_A1(1:7) = TMP_B1(1:7) + TMP_C1(1:7)
  A(I:I+7) = TMP_A1(1:7)
  TMP_B2(1:7) = B(I+8:I+15)
  TMP_C2(1:7) = C(I+8:I+15)
  TMP_A2(1:7) = TMP_B2(1:7) + TMP_C2(1:7)
  A(I+8:I+15) = TMP_A2(1:7)
ENDDO
```

FULLUNROLL_PRE_SIMD(1/2)

FUJITSU

最適化指示子

FULLUNROLL_PRE_SIMD[(n)]

- 機能概要
  - SIMD化前のフルアンローリングの動作を制御します。
  - nは、対象ループの繰り返し数の上限を表す2～100の整数値です。
  - nの値が省略された場合、コンパイラが自動的に最適な値を決定します。
  - 指定した直後のDOループまたは配列記述のみが対象となります。
- 効果
  - 繰返し数が縮小されループの繰返しによるオーバヘッドが少なくなります。また、多重に展開された実行文がループ内で一体になって最適化されるため、実行性能の向上が期待できます。
- ポイント
  - SIMD化が促進されることで、演算実行時間を削減できる場合があります。
- 注意
  - 繰返し数が不明な場合は、最適化を行いません。

FULLUNROLL_PRE_SIMD(2/2)

- 実用例

オリジナルソース

```
 DO I=1,N
!OCL FULLUNROLL_PRE_SIMD
    DO J=1,16
      A(I,J) = B(I,J) + C(I,J)
    ENDDO
  ENDDO
```

内側ループに対してSIMD化前のフルアンローリングを適用します。

最適化後のソースイメージ

```
DO I=1,N,8
  A(I:I+7,1) = B(I:I+7,1) + C(I:I+7,1)
  A(I:I+7,2) = B(I:I+7,2) + C(I:I+7,2)
  A(I:I+7,3) = B(I:I+7,3) + C(I:I+7,3)
  A(I:I+7,4) = B(I:I+7,4) + C(I:I+7,4)
  A(I:I+7,5) = B(I:I+7,5) + C(I:I+7,5)
  A(I:I+7.6) = B(I:I+7,6) + C(I:I+7,6)
  A(I:I+7,7) = B(I:I+7,7) + C(I:I+7,7)
  A(I:I+7,8) = B(I:I+7,8) + C(I:I+7,8)
  A(I:I+7,9) = B(I:I+7,9) + C(I:I+7,9)
  A(I:I+7,10) = B(I:I+7,10) + C(I:I+7,10)
  A(I:I+7,11) = B(I:I+7,11) + C(I:I+7,11)
  A(I:I+7,12) = B(I:I+7,12) + C(I:I+7,12)
  A(I:I+7,13) = B(I:I+7,13) + C(I:I+7,13)
  A(I:I+7,14) = B(I:I+7,14) + C(I:I+7,14)
  A(I:I+7,15) = B(I:I+7,15) + C(I:I+7,15)
  A(I:I+7,16) = B(I:I+7,16) + C(I:I+7,16)
ENDDO
```

内側ループをフルアンローリング後、DO I=1,Nのループがが SIMD化(VL:8)されます。

UNROLL_AND_JAM(1/2)

オプション名	default
-Kunroll_and_jam[=n] -Knounroll_and_jam	-Knounroll_and_jam

最適化指示子
UNROLL_AND_JAM[(n)]
UNROLL_AND_JAM_FORCE[(n)]

- 機能概要
  - 多重ループの外側のループをアンローリングによりn重に展開し、さらに展開された内側のループを融合します。
  - -Kunroll_and_jamオプション、UNROLL_AND_JAM指示子は次の場合、最適化を行いません。
    - 最内ループの場合
    - 最適化の効果（2次元目に加減算がある、再利用性がある、共通式がとれる）が期待できないと判断した場合
    - 繰り返しを跨いだデータ依存があると判断した場合
  - UNROLL_AND_JAM_FORCE指示子は繰り返しを跨いだデータ依存はないとみなしてアンロールアンドジャムを必ず適用します。
  - 特定のDOループに適用する場合は、UNROLL_AND_JAM指示子または、UNROLL_AND_JAM_FORCE指示子を使用します。
- 効果
  - 共通式の除去が促進され、実行性能の向上が期待できます。
  - L2キャッシュ利用が効率化されることで、メモリ待ちを解消できる場合があります。

UNROLL_AND_JAM(2/2)

- ポイント
  - 効果が得られるか否かはループ単位で異なりますので、-Kunroll_and_jam[=N]オプションでプログラム全体へ適用せず、UNROLL_AND_JAM指示子やUNROLL_AND_JAM_FORCE指示子でループ単位に適用することを推奨します。
  - 繰り返しを跨いだデータ依存がなく、共通式がとれると判断でき、強制的にアンロールアンドジャムを適用したい場合は、UNROLL_AND_JAM_FORCE指示子を使用してください。
- 注意
  - UNROLL_AND_JAM_FORCE指示子を誤って指定した（繰り返しを跨いだデータ依存がある）場合、実行結果は保証されません。
  - 最内ループの繰り返し数が少ない場合、データストリーム数の増加やデータのアクセス順序の変化によっては、キャッシュの利用効率を低下させ、実行性能が低下する場合があります。
    詳細は「プログラミングガイド　チューニング編　外側ループアンローリング」を参照してください。
  - キャッシュミスが増加する場合はプリフェッチで補正する良くなる場合があります。
    詳細は「プログラミングガイド　チューニング編　外側ループアンローリング」を参照してください。
- 実用例

オリジナルソース

```
!OCL UNROLL_AND_JAM(2)
   DO J=1, 128
      DO I=1, 128
         A(I, J) = B(I, J) + B(I, J+1)
         ...
      END DO
   END DO
```

最適化後のソースイメージ

```
DO J=1, 128, 2
   DO I=1, 128
      A(I, J) = B(I, J) + B(I, J+1)
      A(I, J+1) = B(I, J+1) + B(I, J+2)
      ...
   END DO
END DO
```

STRIPING(1/2)

FUJITSU

オプション名	default
-Kstriping[=n] -Knostriping	-Knostriping

最適化指示子
STRIPING[(n)]

- 機能概要
  - DOループ内のすべての実行文をループ内へ一定数(ストライプ長n)分展開して、ループの繰返し数をn分の1にします。
  - ループ外からの飛込みもループ外への飛出しもないループが対象です。
  - 特定のDOループに適用する場合は、STRIPING指示子を使用します。
- 効果
  - 繰返し数が縮小されループの繰返しによるオーバヘッドが少なくなります。また、命令スケジューリングが促進されるなど、実行性能の向上が期待できます。
- ポイント
  - 展開数が2,3と少なく、確実に命令の並び替えを行いたい場合にはストライピングが有効です。
- 注意
  - 命令は確実に並び替えられますが、使用レジスタ数が増加するため、ストライプ長nを長くすると実行性能が低下する場合があります。
  - ループ内の文が多重に展開されるので、オブジェクトプログラムの大きさが増加します。
  - 翻訳時間が長くなる場合があります。

STRIPING(2/2)

FUJITSU

- 実用例

オリジナルソース

```
!OCL STRIPING (2)
  DO I=1,N
    A(I) = B(I) + C(I)
  ENDDO
```

最適化後のソースイメージ

```
DO I=1,N,2
  TMP_B1 = B(I)
  TMP_B2 = B(I+1)
  TMP_C1 = C(I)
  TMP_C2 = C(I+1)
  TMP_A1 = TMP_B1 + TMP_C1
  TMP_A2 = TMP_B2 + TMP_C2
  A(I) = TMP_A1
  A(I+1) = TMP_A2
ENDDO
```

LOOP_FISSION(1/3)

オプション名	default
-Kloop_fission -Kloop_nofission	-O2以上：-Kloop_fission -O1以下：-Kloop_nofission

最適化指示子
LOOP_FISSION_TARGET[({CL | LS})] FISSION_POINT[(n)]

- 機能概要
  - ループを複数の小さなループに分割します。
  - ループ外からの飛込みもループ外への飛出しもないループが対象です。
  - 特定のDOループに適用する場合は、LOOP_FISSION_TARGET指示子を使用します。

引数	ループ分割アルゴリズム
CL （default）	• クラスタリングアルゴリズム • ループ分割に伴う一時的なデータ転送のための作業配列の削減を優先したループ分割を行います。
LS	• 局所探索アルゴリズム • ソフトウェアパイプライニングの促進を優先したループ分割を行います。

  - ループ内の指定された位置でループを分割する場合は、FISSION_POINT指示子を使用します。
    最内ループから数えてn重ネストされた多重ループを対象に分割します。
    nは1から6までの整数値を指定します。
    nの指定を省略した場合、最内(1次元)ループのみを分割します。
- 効果
  - ループ分割によりループ交換が行われ、外側ループで並列化可能になるため、実行性能の向上が期待できます。

LOOP_FISSION(2/3)

- 注意
  - 局所探索アルゴリズムはクラスタリングアルゴリズムに比べて翻訳時間が増加します。
- 実用例
  - -Kloop_fissionオプション

オリジナルソース

```
DO I=1,N
   E(I) = A1(I)*B1(I) + A2(I)*B2(I) + A3(I)*B3(I) + A4(I)*B4(I) + A5(I)*B5(I)
   F(I) = C1(I)*D1(I) + C2(I)*D2(I) + C3(I)*D3(I) + C4(I)*D4(I) + C5(I)*D5(I)
ENDDO
```

最適化後のソースイメージ

```
DO I=1,N
   E(I) = A1(I)*B1(I) + A2(I)*B2(I) + A3(I)*B3(I) + A4(I)*B4(I) + A5(I)*B5(I)
ENDDO
DO I=1,N
   F(I) = C1(I)*D1(I) + C2(I)*D2(I) + C3(I)*D3(I) + C4(I)*D4(I) + C5(I)*D5(I)
ENDDO
```

LOOP_FISSION(3/3)

- LOOP_FISSION_TARGET指示子

CL

```
!OCL LOOP_FISSION_TARGET(CL)
 DO I=1,N
   S1 = A(I) + B(I)
   S2 = C(I) + D(I)
   . ..
   P(I) = S1 + Q(I)
   X(I) = S2 + Y(I)
   ...
 ENDDO
```

ループ分割に伴う一時的なデータ転送のための作業配列の削減を優先して、ループを分割します。

LS

```
!OCL LOOP_FISSION_TARGET(LS)
 DO I=1,N
   S1 = A(I) + B(I)
   S2 = C(I) + D(I)
   ...
   P(I) = S1 + Q(I)
   X(I) = S2 + Y(I)
   ...
 ENDDO
```

ソフトウェアパイプライニングの促進を優先して、ループを分割します。

- FISSION_POINT指示子

オリジナルソース

```
 DO I=1,N
   DO J=1,N
     A(I,J) = B(I,J)
!OCL FISSION_POINT(1)
     C(I,J) = D(I,J)
   ENDDO
 ENDDO
```

上記の例では、Jのループで分割されます。

最適化後のソースイメージ

```
 DO I=1,N
   DO J=1,N
     A(I,J) = B(I,J)
   ENDDO
   DO J=1,N
     C(I,J) = D(I,J)
   ENDDO
 ENDDO
```

LOOP_FISSION_THRESHOLD(1/2)

オプション名	default
-Kloop_fission_threshold=n	-Kloop_fission_threshold=50

最適化指示子
LOOP_FISSION_THRESHOLD(n)

- 機能概要
  - ループ分割後のループの粒度（ループ内の命令数やレジスタ数など）を決める閾値を指示します。
  - nは1から100までの範囲で指定します。
  - 最適化制御行にLOOP_FISSION_TARGET指示子が指定されているかつ、-Kloop_fissionオプション、-Koclオプション、および-O2オプション以上が有効な場合に、本オプションが有効になります。
  - 特定のDOループに適用する場合は、LOOP_FISSION_THRESHOLD指示子を使用します。
- 効果
  - ループ分割の分割数を調整することで、ソフトウェアパイプライニングの促進、キャッシュメモリ利用効率の改善、レジスタ不足の解消の効果が期待できます。
- ポイント
  - nを小さくすると、分割後のループが小さくなり、分割数が増える傾向があります。
- 注意
  - 分割数が増えると作業領域、利用するキャッシュメモリが増えます。

LOOP_FISSION_THRESHOLD(2/2)

FUJITSU

- 実用例

オリジナルソース(N={50|20})

```
!OCL LOOP_FISSION_TARGET
!OCL LOOP_FISSION_THRESHOLD(N)
DO I=1,NN
A1(I) = A1(I) + B1(I)
…
A2(I) = A2(I) + B2(I)
…
A3(I) = A3(I) + B3(I)
…
A4(I) = A4(I) + B4(I)
…
ENDDO
```

最適化後のソースイメージ(N=50)

```
DO I=1,NN
A1(I) = A1(I) + B1(I)
…
A2(I) = A2(I) + B2(I)
…
ENDDO
DO I=1,NN
A3(I) = A3(I) + B3(I)
…
A4(I) = A4(I) + B4(I)
…
ENDDO
```

最適化後のソースイメージ(N=20)

```
DO I=1,NN
A1(I) = A1(I) + B1(I)
…
ENDDO
DO I=1,NN
A2(I) = A2(I) + B2(I)
…
ENDDO
DO I=1,NN
A3(I) = A3(I) + B3(I)
…
ENDDO
DO I=1,NN
A4(I) = A4(I) + B4(I)
…
ENDDO
```

LOOP_FISSION_STRIPMINING(1/3)

FUJITSU

オプション名	default
-Kloop_fisson_stripmining[={n|L1|L2}] -Kloop_nofisson_stripmining	-Kloop_nofission_stripmining

最適化指示子
LOOP_FISSION_STRIPMINING[({n|L1|L2})]

- 機能概要
  - 自動ループ分割後にループを小さい繰返し数で断片化することを指示します。
  - 最適化制御行にLOOP_FISSION_TARGET指示子が指定されているかつ、-Kloop_fissionオプション、-Koclオプション、および-O2オプション以上が有効な場合に、本オプションが有効になります。
  - ストリップの長さを指定する方法には、直接長さ(n)を指示する方法と、各キャッシュ階層('L1'または'L2')に適したサイズとなるように指示する方法があります。

引数	ストリップ長
n	• ストリップの長さをnにします。nは、2～100000000の範囲で指定できます。
L1	• キャッシュメモリの利用効率を考慮し、ストリップの長さを1次キャッシュのサイズに合わせます。
L2	• キャッシュメモリの利用効率を考慮し、ストリップの長さを2次キャッシュのサイズに合わせます。

  引数が省略された場合、コンパイラが自動的に値を決定します。
  - 次のいずれかに該当する場合にストリップマイニングを適用します。
    - オプションまたは最適化指示子が有効である。
    - 一時領域のサイズが翻訳時に不明である。
    - 一時領域のサイズが翻訳時に明確かつ、8Kバイト以上である。
    - -Kstripingオプションまたは、STRIPING指示子が無効である。
    - ブロッキングが適用されていない。

LOOP_FISSION_STRIPMINING(2/3)

- 効果
  - ループ分割したループ間でアクセスされるデータに対して、キャッシュメモリの利用効率の向上が期待できます。
- ポイント
  - ストリップの長さnはソフトウェアパイプライニングされる長さを指定するのが良いです。
    -Kloop_fisson_stripminingオプションを指定せずににコンパイルして、ソフトウェアパイプライニングのリスタ情報を参照して調整することができます。
- 注意
  - ストリップの長さnを大きくし過ぎるとL1キャッシュに乗らなくなる可能性があります。
  - ストリップの長さnを小さくし過ぎるとソフトウェアパイプライニングのルートに入らなくなります。
  - ストリップの長さnが小さい場合、キャッシュアクセスの連続性を阻害するため、プリフェッチが効かなくなる可能性があります。
- 実用例
  - -Kloop_fisson_stripminingオプション

オリジナルソース

```
  REAL A(L),B(L),P(L),Q
!OCL LOOP_FISSON_TARGET
  DO I=1,L
    Q = A(I) + B(I)
    ...
    P(I) = P(I) + Q
    ...
  ENDDO
```

自動ループ分割後のソースイメージ

```
REAL A(L),B(L),P(L),Q
REAL TEMPARRAY_Q(L)
DO I=1,L
  TEMPARRAY_Q(I) = A(I) + B(I)
  ...
ENDDO
DO I=1,L
  P(I) = P(I) + TEMPARRAY_Q(I)
  ...
ENDDO
```

コンパイラが一時的な配列TEMPARRAY_Qを生成します。

最適化後のソースイメージ

```
REAL A(L),B(L),P(L),Q
REAL TEMPARRAY_Q(256)
DO II=1,L,256
  DO I=II,MIN(L,II+255)
    TEMPARRAY_Q(I-II) = A(I) + B(I)
    ...
  ENDDO
  DO I=II,MIN(L,II+255)
    P(I) = P(I) + TEMPARRAY_Q(I-II)
    ...
  ENDDO
ENDDO
```

LOOP_FISSION_STRIPMINING(3/3)

FUJITSU

- LOOP_FISSION_STRIPMINING指示子

オリジナルソース

```
 REAL A(L),B(L),P(L),Q
!OCL LOOP_FISSION_TARGET
!OCL LOOP_FISSION_STRIPMINING(256)
  DO I=1,L
     Q = A(I) + B(I)
     …
     P(I) = P(I) + Q
     …
  ENDDO
```

最適化後のソースイメージ

```
REAL A(L),B(L),P(L),Q
REAL TEMPARRAY_Q(256)
DO II=1,L,256
   DO I=II,MIN(L,II+255)
      TEMPARRAY_Q(I-II) = A(I) + B(I)
      …
   ENDDO
   DO I= II,MIN(L,II+255)
      P(I) = P(I) + TEMPARRAY_Q(I-II)
      …
   ENDDO
ENDDO
```

- 分割したループの外側にループを生成し、ストリップ長256でストリップマイニングします。同時に、ループ間の途中結果を格納する一時的な配列をコンパイラが生成します。
- この配列TEMPARRAY_Qの要素数は、ストリップ長と同じ256となります。

-Kloop_fusion(1/2)

オプション名	default
-Kloop_fusion -Kloop_nofusion	-O2以上：-Kloop_fusion -O1以下：-Kloop_nofusion
最適化指示子	
LOOP_NOFUSION	

- 機能概要
  - -O2以上の場合に隣接するループを融合することを指示します。
  - 特定のDOループのループ融合を抑止する場合は、LOOP_NOFUSION指示子を使用します。
- 効果
  - ループを融合することにより、データを局所化する効果が期待できます。
- 注意
  - 過剰に融合するとループボディが大きくなり過ぎて、ソフトウェアパイプライニングが効かなくなる可能性があります。

-Kloop_fusion(2/2)

- 実用例
  - -Kloop_fusionオプション

オリジナルソース

```
SUBROUTINE SUB(A, B, C, D, E, N)
REAL*8 A(N), B(N), C(N)
REAL*8 D(N), E(N)
DO I=1,N
  A(I)=B(I)+C(I)
ENDDO
DO I=1,N
  D(I)=A(I)+E(I)
ENDDO
END SUBROUTINE SUB
```

最適化後のソースイメージ

```
SUBROUTINE SUB (A, B, C, D, E, N)
REAL*8 A(N), B(N), C(N)
REAL*8 D(N), E(N)
DO I=1,N
  A(I)=B(I)+C(I)
  D(I)=A(I)+E(I)
ENDDO
END SUBROUTINE SUB
```

  - LOOP_NOFUSION指示子

オリジナルソース

```
!OCL LOOP_NOFUSION
  DO I=1,N
    A(I)=B(I)+C(I)
  ENDDO
  DO J=1,N
    D(J)=E(J)+F(J)
  ENDDO
  DO K=1,N
    G(K)=H(K)+L(K)
  ENDDO
```

最適化後のソースイメージ

```
!OCL LOOP_NOFUSION
  DO I=1,N
    A(I)=B(I)+C(I)
  ENDDO
  DO J=1,N
    D(J)=E(J)+F(J)
    G(J)=H(J)+L(J)
  ENDDO
```

IとJのループは融合されません。JとKのループは融合されます。

LOOP_INTERCHANGE(1/2)

FUJITSU

オプション名	default
-Kloop_interchange -Kloop_nointerchange	-O2以上：-Kloop_interchange -O1以下：-Kloop_nointerchange

最適化指示子
LOOP_INTERCHANGE(var1,var2[,var3]...)

- 機能概要
  - -O2以上の場合にループ交換を実施することを指示します。
  - 特定のDOループのループ交換をする場合は、LOOP_INTERCHANGE指示子を使用します。
    指定された順序(var1、var2、…)で多重DOループの入れ換えを実施します。
    var1、var2、var3、... はDO 変数名です。
  - 入れ換えたときに結果が異なるなど、入れ換えが不可能な場合には行いません。
- 効果
  - ループを交換することにより、データのアクセス効率を向上させる効果が期待できます。
- ポイント
  - ループ内のデータアクセスが連続アクセスになるようにループを交換することが望ましいです。
    特に左辺に関しては連続にしてください。

LOOP_INTERCHANGE(2/2)

FUJITSU

●実用例

オリジナルソース

```
!OCL LOOP_INTERCHANGE(I,J)
  DO I=1,M
    DO J=1,N
      A(I,J) = A(I,J) + B(J,I)
    ENDDO
  ENDDO
```

最適化後のソースイメージ

```
DO J=1,N
  DO I=1,M
    A(I,J) = A(I,J) + B(J,I)
  ENDDO
ENDDO
```

- LOOP_INTERCHANGE指示子の指定のない場合は、DO変数Iで並列化されますが、LOOP_INTERCHANGE指示子の指定によりループを入れ換え、DO変数Jで並列化されます。

UNSWITCHING(1/2)

FUJITSU

最適化指示子

UNSWITCHING

- 機能概要
  - 指定されたIF構文をループアンスイッチングすることを指示します。
  - 本最適化制御行はループ内で不変なIF構文の直前に指定してください。
    上記以外の箇所に記述した場合は指定が無効になります。
- 効果
  - ループ内の分岐がなくなることでSIMD化やソフトウェアパイプライニングの促進が期待できます。
- 注意
  - ループアンスイッチングの対象となるループに多くの実行文が含まれる場合、翻訳メモリや翻訳時間が大幅に増加する場合があります。

UNSWITCHING(2/2)

FUJITSU

- 実用例

オリジナルソース

```
  DO I=1,N
!OCL UNSWITCHING
     IF (X == 0) THEN
       A(I) = B(I)
     ELSE
       A(I) = C(I)
     ENDIF
  ENDDO
```

IF構文をループアンスイッチングします。

最適化後のソースイメージ

```
IF (X == 0) THEN
  DO I=1,N
    A(I) = B(I)
  ENDDO
ELSE
  DO I=1,N
    A(I) = C(I)
  ENDDO
ENDIF
```

オリジナルソース

```
  DO I=1,N
    IF (X == 0) THEN
      A(I) = B(I)
!OCL UNSWITCHING
    ELSE IF (X == 1) THEN
      A(I) = C(I)
    ELSE
      A(I) = D(I)
    ENDIF
  ENDDO
```

UNSWITCHING指示子が指定されたIF構文のみをループアンスイッチングします。
UNSWITCHING指示子が指定されたIF構文が含まれるループ内のUNSWITCHING指示子が指定されていないIF構文はループアンスイッチングされません。

最適化後のソースイメージ

```
IF (X == 1) THEN
  DO I=1,N
    IF (X == 0) THEN
      A( I ) = B( I )
    ENDIF
    A( I ) = C( I )
  ENDDO
ELSE
  DO I=1,N
    IF (X == 0) THEN
      A( I ) = B( I )
    ENDIF
    A( I ) = D( I )
  ENDDO
ENDIF
```

LOOP_PERFECT_NEST

オプション名	default
-Kloop_perfect_nest -Kloop_noperfect_nest	-O3以上：-Kloop_perfect_nest -O2以下：-Kloop_noperfect_nest

最適化指示子
LOOP_PERFECT_NEST LOOP_NOPERFECT_NEST

- 機能概要
  - ”O2 –Kloop_perfect_net”または、-O3以上の場合に不完全多重ループを分割して完全多重ループにすることを指示します。
  - 特定の不完全多重ループを分割して完全多重ループにする場合は、LOOP_PERFECT_NEST指示子を使用します。
- 効果
  - 不完全多重ループを完全多重ループにすることにより、ループ交換、ループ一重化などの最適化が促進されるなどの効果が期待できます。
- 実用例

オリジナルソース

```
!OCL LOOP_PERFECT_NEST
  DO J=1,N！不完全多重ループ
    A(J) = B(J)+1
    DO I=1,N
      C(J,I) = D(J,I)+A(J)
    ENDDO
  ENDDO
```

最適化後のソースイメージ

```
DO J=1,N
  A(J) = B(J)+1
ENDDO
DO J=1,N！完全多重ループ
  DO I=1,N
    C(J,I) = D(J,I)+A(J)
  ENDDO
ENDDO
```

J,Iの不完全多重ループを分割して、完全多重ループにします。

-Kzfill(1/3)

オプション名	default
-Kzfill[=N] -Knozfill	-Knozfill

- 機能概要
  - ループ内で書き込みのみ行う配列データをメモリからロードすることなく、キャッシュ上に書き込み用のキャッシュラインを確保する命令を使うことを指示します。
  - Nを指定することで、Nキャッシュライン先のデータを最適化の対象とします。
    オプションでプログラム全体へ適用せず、ZFILL指示子でループ単位に適用することを推奨します。
  - -O2以上が有効な場合に、本オプションが有効になります。同一ループ内に参照がある配列、非連続アクセスされる配列または、IF構文配下でストアされる配列は最適化しません。
- 効果
  - ループ内で書き込みのみ行う配列データの書き込みが高速化されるため、実行性能の向上を期待できます。

-Kzfill(2/3)

FUJITSU

- 注意
  - -Kzfillオプションが適用された場合、2次キャッシュへのプリフェッチ命令は出力されません。
  - 本最適化が確保したキャッシュラインが必ずストアされるようなループ変形を行うため、以下の最適化が適用できなくなります。
    - ループアンローリング
    - ループストライピング
  - 以下の場合に本最適化を適用すると実行性能が低下することがあります。
    - メモリバンド幅のボトルネック影響を受けていないプログラム
    - 繰返し数が小さいループ
    - Nでキャッシュライン数を指定しているかつ、キャッシュラインに入る要素数より繰返し数が小さい場合
  - -Kzfillオプションの指定によって実行性能の低下が起きる場合は、-Kzfillオプションを指定しないでください。

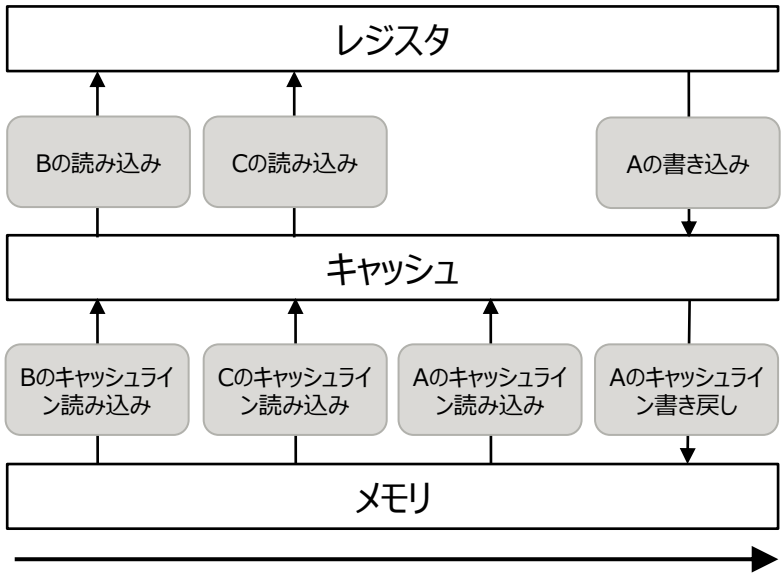
-Kzfill(2/3)

●実用例

オリジナルソース

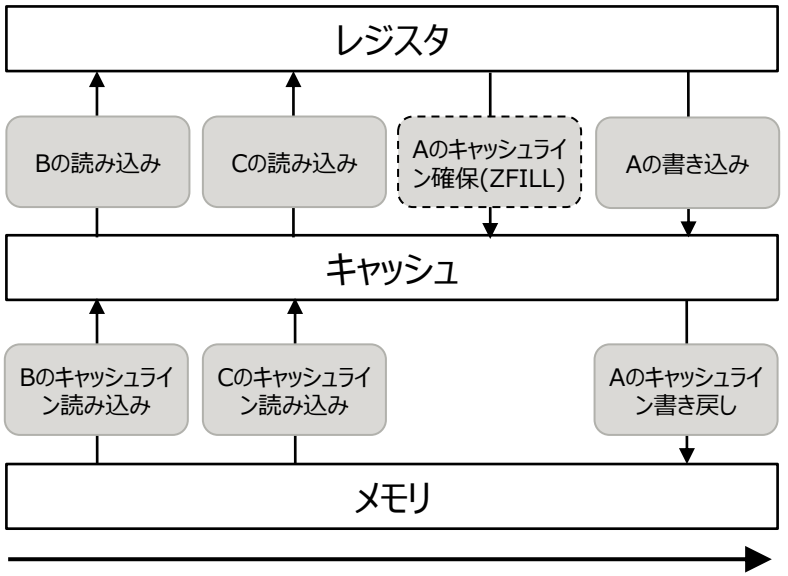
```
DO I=1,N
  A(I) = B(I) + C(I)
ENDDO
```

-Kzfill未指定時



メモリアクセス数は計4回となる。

-Kzfill指定時



Aのメモリからの読み込みがなくなるため、メモリアクセス数は計3回となる。

CLONE(1/2)

FUJITSU

最適化指示子

CLONE(var==n1[,n2]...)

- 機能概要
  - ループ内で変数varの値が不変とみなして、指定された変数varと値n1[,n2]…の等式を条件式とする分岐を生成し、ループを複製することを指示します。
  - 分岐は指定した値の順に生成されます。
  - Varは整数型の変数です。種別型パラメタは1、2、4または、8です。N1[,n2]…は-9223372036854775808から9223372036854775807までの10進数または名前付き定数です。
  - -O3以上が有効な場合に、本最適化指示子が有効になります。
- 効果
  - フルアンローリングなどのほかの最適化が促進されるため、実行性能の向上を期待できます。
- 注意
  - 対象となるループ内で変数varが更新される場合、実行結果が保証されません。
  - ループを複製するため、オブジェクトプログラムの大きさおよび翻訳時間が増加する場合があります。

CLONE(2/2)

FUJITSU

- 実用例

オリジナルソース

```
!OCL CLONE(N==10)
  DO I=1,N
    A(I) = I
  ENDDO
```

最適化後のソースイメージ

```
IF (N==10) THEN
  DO I=1,10
    A(I) = I
  ENDDO
ELSE
  DO I=1,N
    A(I) = I
  ENDDO
ENDIF
```

PRELOAD(1/2)

オプション名	default
-Kpreload -Knopreload	-Knopreload

最適化指示子
PRELOAD

- 機能概要
  - IF文を含むループのSIMD化後のロード命令の投機実行(IFのTHEN/ELSE節にあるロード命令をIF条件判定前に移動)を行います。
- 効果
  - 命令スケジューリングが促進されるため、実行性能の向上が期待できます。
- ポイント
  - -Kpreexオプション、PREEX指示子がループ内にあるIF文内から不変式をループ外に移動するのに対し、-Kpreloadオプション、PRELOAD指示子はループ内にあるIF文内からロード命令をIF文の前に移動します。
- 注意
  - プログラムの論理上では実行されないはずのロード命令が実行されることで、本来は発生しない例外が発生し、実行が中断することがあります。
  - ロード命令の投機実行の影響による中断か否かは、翻訳時の診断メッセージで確認することができます。

PRELOAD(2/2)

- 実用例
  - B(I)とC(I)のロード命令が、IFの判定より前に移動します。

オリジナルソース

```
SUBROUTINE FOO(A,B,C,M,N)
REAL(8),DIMENSION(1:N) :: A,B,C
LOGICAL,DIMENSION(1:N) :: M
DO I=1,N
  IF (M(I) .GT. 0) THEN
    A(I) = B(I) + C(I)
  ENDIF
END SUBROUTINE
```

最適化後のソースイメージ

```
SUBROUTINE FOO(A,B,C,M,N)
REAL(8),DIMENSION(1:N) :: A,B,C
LOGICAL,DIMENSION(1:N) :: M
REAL(8) :: TMP
DO I=1,N
  TMP = B(I) + C(I)
  IF (M(I) .GT. 0) THEN
    A(I) = TMP
  ENDIF
END SUBROUTINE
```

-Kauto/-Kautoobjstack/-Ktemparraystack (1/2)

FUJITSU

オプション名	default
-Kauto -Knoauto	-Kauto
-Kautoobjstack -Knoautoobjstack	-Kautoobjstack
-Ktemparraystack -Knotemparraystack	-Ktemparraystack

- 機能概要
  - 各割り付け対象をスタックに割り付けます。

オプション	割り付け対象
auto	• SAVE属性をもつ変数および初期値をもつ変数を除く局所変数
autoobjstack	• 自動割付けデータ実体
temparraystack	• 配列演算の途中結果 • DO CONCURRENTの繰返し数が定数の場合、マスク式評価結果

- 効果
  - スタック領域への割り付けにより、実行性能の向上を期待できます。
- ポイント
  - 京では本オプションはデフォルトで動作しませんでした。富岳ではデフォルトで動作するため、ユーザーが意識することなくスタック領域への割り付けが行われます。

-Kauto/-Kautoobjstack/-Ktemparraystack (2/2)

FUJITSU

- 注意
  - スタック領域が不足した場合、異常終了することがあります。
  - 異常終了を回避するためには、スタック領域の上限値を大きくして実行するか、またはスタック領域の使用を小さくするために-Knoautoオプション、-Knoautoobjstackオプション、または-Knotemparraystackオプションを指定して翻訳してください。
  - スレッド並列プログラムの場合、OMP_STACKSIZEまたはTHREAD_STACK_SIZEによって必要なスレッドスタックのサイズを指定してください。
  - -Knoautoオプションを-Kopenmpオプションまたは-Kparallelオプションと同時に指定する場合、-Kopenmpオプションまたは-Kparallelオプションより後方に指定する必要があります。
  - -Knoauto オプションを指定してOpenMPプログラムまたは自動並列化プログラムを翻訳する場合、プログラムが正しく動作しないことがあります。

スタック領域の不足による異常終了

```
SUBROUTINE SUB(N)
  REAL(KIND=8),DIMENSION(N) :: A
  A=1.
  PRINT *,A(N)
END SUBROUTINE

CALL SUB(2000000)
END

$ ulimit -s
8192
$ frt a.f90
$ ./a.out
セグメント例外(コアダンプしました)
```

スタック領域を大きくすることで正常終了

```
$ ulimit -s unlimited
$ ulimit -s
unlimited
$ frt a.f90
$ ./a.out
  1.000000000000000
```

EVAL(1/3)

オプション名	default
-Keval -Knoeval	-Knoeval

最適化指示子
EVAL NOEVAL

- 機能概要
  - ソースプログラムに対して演算評価方法を変更する最適化を行うかどうかを指示します。
- 効果
  - 演算評価方法を変更することで、ループ内のおける演算数(命令数)の削減や、SIMDや自動並列など他の最適化の促進により、実行性能の向上を期待できます。
- 注意
  - 演算の評価順序の変更により、計算誤差が生じることがあります。
  - -Kfastオプション指定時、-Kevalオプションが誘導され、計算誤差が生じることがあります。そのような場合には、-Kfastオプションより後ろに-Knoevalオプションを指定してください。
  - -Kevalオプション指定時、-Kfsimpleオプション、-Kreductionオプション(-Kparallelオプション有効時）、および-Ksimd_reduction_product(-Ksimd[={1|2|auto}有効時])オプションが誘導されるため、それぞれのオプションの副作用にも注意が必要です。

EVAL(2/3)

FUJITSU

- 実用例
  - 演算を乗算2回、加算1回から乗算1回、加算1回に変更して、乗算を1回削減します。

オリジナルソース

```
!OCL EVAL
  DO I=1,N
    A(I)=A(I)*B(I)+A(I)+C(I)
  ENDDO
```

最適化後のソースイメージ

```
DO I=1,N
  A(I)=A(I)*(B(I)+C(I))
ENDDO
```

  - 対象となるループ中で演算評価方法を変更する最適化を行います。
    ソース上では加算3回は変わりませんが、”A(I)+B(I)”と”C(I)+D(I)”を並列に演算できるようになるため、実行性能が向上する可能性があります。

オリジナルソース

```
!OCL EVAL
  DO I=1,N
    A(I)=A(I)+B(I)+C(I)+D(I)
  ENDDO
```

最適化後のソースイメージ

```
DO I=1,N
  A(I)=(A(I)+B(I))+(C(I)+ D(I))
ENDDO
```

  - 除算を逆数の乗算に変更します。
    除算命令よりレイテンシの短い乗算命令に変更することで、実行性能が向上する可能性があります。

オリジナルソース

```
!OCL EVAL
  DO I=1, N
    A(I)=B(I)/10
  ENDDO
```

最適化後のソースイメージ

```
TMP=1/10
DO I=1,N
  A(I)=B(I)*TMP
ENDDO
```

EVAL(3/3)

- 演算評価方法の変更による最適化により、リダクション演算を含むループの並列化を促進します。

オリジナルソース

```
!OCL EVAL
DO I=1,10000
  SUM=SUM+A(I)
ENDDO
```

最適化後のソースイメージ

```
(コア1)
SUM1=0
DO I=1,5000
  SUM1=SUM1+A(I)
ENDDO
```

```
(コア2)
SUM2=0
DO I=5001,10000
  SUM2=SUM2+A(I)
ENDDO
```

```
SUM=SUM+SUM1+SUM2
```

EVAL_CONCURRENT(1/2)

オプション名	default
-Keval_concurrent -Keval_noconcurrent	-Keval_noconcurrent

最適化指示子
EVAL_CONCURRENT EVAL_NOCONCURRENT

- 機能概要
  - ループ中の演算の並び替えを行って命令の並列性を高める、Tree-Height-Reduction最適化を行います。
  - -O1以上かつ-Kevalオプションの時、オプション指定で動作します。
- 効果
  - ループの繰返し数が小さく、ソフトウェアパイプライニングできなかった演算が多いループに適用すると、性能が向上する可能性があります。
- ポイント
  - Tree-Height-Reduction最適化とは、ループ中の演算に対して演算木をなるべく低くなるように演算の並び替えを行い、命令の並列性を高める最適化です。
    演算木とは、演算子の優先順位に従って演算式を木構造で表現したものです。葉の部分には、値を置きます。
    それ以外の節には、演算子を置きます。優先順位が高い演算子ほど、上位の階層に配置します。
- 注意
  - 浮動小数点演算に対して適用する場合は、-Kevalオプションを有効にする必要があります。

EVAL_CONCURRENT(2/2)

- 実用例
  - tree-height-reduction最適化において、命令の並列性を優先することを指示します。

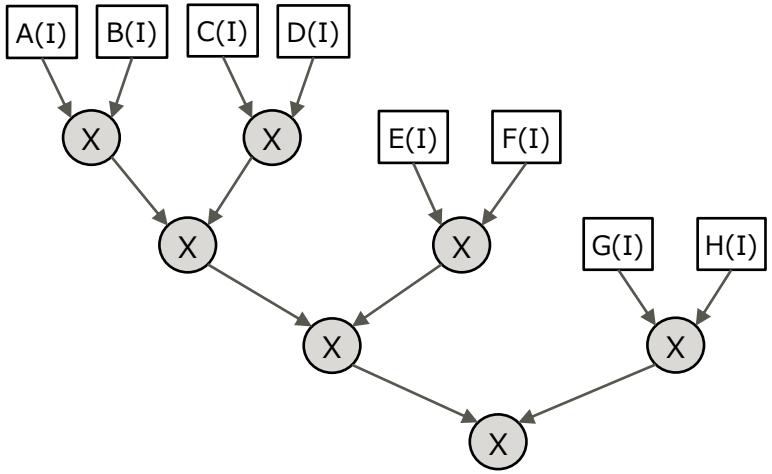
オリジナルソース

```
!OCL EVAL_CONCURRENT
DO I=1,N
  X(I)=A(I)*B(I)+C(I)*D(I)+E(I)*F(I)+G(I)*H(I)
ENDDO
```

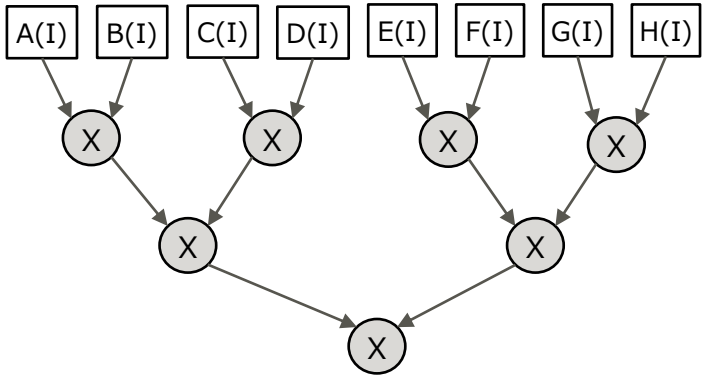
最適化後のソースイメージ

```
DO I=1,N
  X(I)=(A(I)*B(I)+C(I)*D(I))+(E(I)*F(I)+G(I)*H(I))
ENDDO
```

Tree-Height-Reduction最適化前の演算木



Tree-Height-Reduction最適化後の演算木



-Kfz

オプション名	default
-Kfz -Knofz	-Kfz

- 機能概要
  - flush-to-zeroモードを使用するかどうかを指示します。
  - flush-to-zeroモードは、演算結果またはソースオペランドが非正規化数の場合、それらを同符号の0で置き換えます。
- 効果
  - 演算結果またはソースオペランドに非正規化数が発生しても、実行性能に影響はありません。
- 注意
  - -Kfzオプションを指定するとflush-to-zeroモードを使用するため、アンダーフローによって非正規化数となる演算結果やソースオペランドが同符号の0.0に変わります。
  - 一般的に演算の実行性能に関しては、-Kfzオプションを指定する方が高速になります。
    -Knofzオプションを指定した場合、非正規化数を正しく表現するため実行結果は保証されますが、実行性能は遅くなる場合があります。
- 実用例
  - -Kfzオプションは、-Kpreexオプションや-Ksimd=2オプションで投機実行するloadを異常終了しないように、flush-to-zeroモードを使用する時に指定します。

-Kalign_commons

オプション名	default
-Kalign_commons -Knoalign_commons	-Kalign_commons

- 機能概要
  - 共通ブロックに属する変数に対する記憶域への割付け処理において、8バイトの整数型、倍精度実数型、4倍精度実数型、倍精度複素数型、および4倍精度複素数型のデータに対して、8バイトの境界調整を行うかどうかを指示します。
  - 本オプションを使用する場合、すべてのプログラム単位は-Knoalign_commonsオプションを指定して翻訳されていなければなりません。
- 効果
  - 共通ブロックに属する変数に対して効率的にアクセスできるようになるため、実行性能の向上を期待できます。
- 注意
  - 8バイトの境界調整を行う場合は、コンパイラが自動的に空き領域を挿入するため、コードサイズが増えます。

ARRAY_DECLARATION_OPT(1/2)

オプション名	default
-Karray_declaration_opt -Knoarray_declaration_opt	-Knoarray_declaration_opt

最適化指示子
ARRAY_DECLARATION_OPT NOARRAY_DECLARATION_OPT

- 機能概要
  - 繰り返し数が不明なループをSIMD化する際、ループ内で引用する配列の要素数を最大繰り返し数と判断します。
    最大繰り返し数ががSIMD長以下の場合には、SIMD長で回転するループと余りのループの２つのループ構造を作成せず、余りループのみを作成します。
- 効果
  - オブジェクトが小さくなる分、使用レジスタ数が減少してspillの改善が期待できます。
- 注意
  - -Karray_declaration_optオプション、ARRAY_DECLARATION_OPT指示子で実行結果を誤ることはありません。
    実行結果を誤る場合は、配列宣言以上に繰り返しループするプログラムになっている可能性がありますので繰り返し数を確認してください。

ARRAY_DECLARATION_OPT(2/2)

- 実用例

```
  SUBROUTINE LOOP(A,B,C,N)
  REAL*4 A(10),B(10),C(10)
!OCL ARRAY_DECLARATION_OPT
    DO I=1,N
      A(I) = B(I) + C(I)
    ENDDO
  END
```

配列A(I)、B(I)、C(I)の要素数は10で宣言されているので、最大繰り返し数は10と判断します。
単精度型のSIMD長(16)>10のため余りループのみ作成します。

-Koptlib_string

オプション名	default
-Koptlib_string -Knooptlib_string	-Knooptlib_string

- 機能概要
  - 文字列操作関数（bcopy、bzero、memchr、memcmp、memccpy、memcpy、memmove、memset、strcat、strcmp、strcpy、strlen、strncmp、strncpy、strncat）の最適化版ライブラリを静的にリンクします。
  - 翻訳時およびリンク時に指定してください。
  - -Koptlib_stringオプションと同時に、 -KSVEオプションおよび-KA64FXオプションを指定してください。
- 効果
  - memcpyを例として、コピーサイズが16～128KBの場合、通常のlibcと比べて1.5～3.0倍の高速化が期待できます。
- 注意
  - memcpyでは、1スレッドあたりのコピーサイズが4MiBを超えると-Kzfillが自動で有効になります。ただし、1スレッド実行時(非メモリビジー時)は、性能が低下するため注意が必要です。また、スレッド並列実行時(メモリビジー時)は性能向上が期待できますが、コピーサイズによっては、メモリビジーでも-Kzfillが有効にならない場合があります。

SCACHE_ISOLATE_WAY(1/2)

FUJITSU

最適化指示子

```
SCACHE_ISOLATE_WAY(L2=n1[,L1=n2])
END_SCACHE_ISOLATE_WAY
```

- 機能概要
  - 1次キャッシュと2次キャッシュのセクタ1の最大way数を指示します。
    - 指定できる最大way数は、1次キャッシュが4、2次キャッシュが14です。
  - プログラム単位に指示する場合は、プログラム単位の記述位置にSCACHE_ISOLATE_WAY指示子を記述します。指示の有効範囲はプログラム内全体となります。
  - プログラムの一部に対して指示する場合は、指示する範囲をSCACHE_ISOLATE_WAY指示子とEND_SCACHE_ISOLATE_WAY指示子で指定します。
- 効果
  - SCACHE_ISOLATE_ASSIGN指示子との併用により、ループ内で再利用性のあるデータをキャッシュから追い出されないように制御することで、実行性能の向上を期待できます。
- ポイント
  - L1Dキャッシュ、L2キャッシュいずれにも複数個のセクタを設定できます。セクタの最大数はL1Dが4、L2は2です。
  - 最大way数は目標値として働きます。ハードは、ラインリプレース時に各セクタが指定されたway数に近づくように制御します。
  - セクタ内の追い出しはLRUアルゴリズム(最近、最も使われていないデータを最初に捨てる)で制御します。
  - セクタ０、１の用途はアプリケーションで決めることが可能です。ただし、命令列はセクタ０に格納されます。

SCACHE_ISOLATE_WAY(2/2)

- 注意
  - 2次キャッシュはアシスタントコアが常時2wayを使用します。そのため、n1およびn2に指定できる範囲は以下のようになります。
    - 0 ≦ n1 ≦ “2次キャッシュの最大way数 -2”
    - 0 ≦ n2 ≦ “1次キャッシュの最大way数”
  - アシスタントコアを含むCMGでは、L2キャッシュの一部(2way=1MiB分)をアシスタントコア用に使用します。そのため、アシスタントコアを含むCMGでは、2次キャッシュの最大way数は14、サイズは7MiBとなります。
  - プログラム単位の指示があるプログラムの一部に範囲指定の指示を記述することは可能ですが、範囲指定の指示を入れ子に記述することはできません。
  - 旧仕様(京、FX100)では、 CACHE_SECTOR_SIZE指示子の名称でした。
    富岳よりSCACHE_ISOLATE_WAY指示子の名称に変更となりました。
- 実用例
  - 再利用性のある配列aが配列bと配列cのアクセスによってキャッシュから追い出されないようにします。

旧仕様(京、FX100)での指定方法

```
!OCL CACHE_SECTOR_SIZE(4,10)
!OCL CACHE_SUBSECTOR_ASSIGN(A)
DO J=1,M
   DO I=1,N
      A(I) = A(I) + B(I,J)＊C(I,J)
   ENDDO
ENDDO
!OCL END_CACHE_SUBSECTOR
!OCL END_CACHE_SECTOR_SIZE
```

富岳での指定方法

```
!OCL SCACHE_ISOLATE_WAY(L2=10)
!OCL SCACHE_ISOLATE_ASSIGN(A)
DO J=1,M
   DO I=1,N
      A(I) = A(I) + B(I,J)＊C(I,J)
   ENDDO
ENDDO
!OCL END_SCACHE_ISOLATE_ASSIGN
!OCL END_SCACHE_ISOLATE_WAY
```

SCACHE_ISOLATE_ASSIGN(1/2)

FUJITSU

最適化指示子

```
SCACHE_ISOLATE_ASSIGN(array1[,array2]...)
END_SCACHE_ISOLATE_ASSIGN
```

- 機能概要
  - キャッシュのセクタ1に載せる配列を指示します。
  - プログラム単位に指示する場合は、プログラム単位の記述位置にSCACHE_ISOLATE_ASSIGN指示子を記述します。指示の有効範囲はプログラム内全体となります。
  - プログラムの一部に対して指示する場合は、指示する範囲をSCACHE_ISOLATE_ASSIGN 指示子とEND_SCACHE_ISOLATE_ASSIGN指示子で指定します。
  - 数値型または論理型の配列を指定した場合のみ有効となります。
- 効果
  - SCACHE_ISOLATE_WAY指示子との併用により、ループ内で再利用性のあるデータをキャッシュから追い出されないように制御することで、実行性能の向上を期待できます。
- 注意
  - プログラム単位の指示があるプログラムの一部に範囲指定の指示を記述することは可能ですが、範囲指定の指示を入れ子に記述することはできません。
  - 旧仕様(京、FX100)では、 CACHE_SUBSECTOR_ASSIGN指示子の名称でした。
    富岳よりSCACHE_ISOLATE_ASSIGN指示子の名称に変更となりました。

SCACHE_ISOLATE_ASSIGN(2/2)

- 実用例
  - 再利用性のある配列aが配列bと配列cのアクセスによってキャッシュから追い出されないようにします。

旧仕様(京、FX100)での指定方法

```
!OCL CACHE_SECTOR_SIZE(4,10)
!OCL CACHE_SUBSECTOR_ASSIGN(A)
DO J=1,M
   DO I=1,N
      A(I) = A(I) + B(I,J)＊C(I,J)
   ENDDO
ENDDO
!OCL END_CACHE_SUBSECTOR
!OCL END_CACHE_SECTOR_SIZE
```

富岳での指定方法

```
!OCL SCACHE_ISOLATE_WAY(L2=10)
!OCL SCACHE_ISOLATE_ASSIGN(A)
DO J=1,M
   DO I=1,N
      A(I) = A(I) + B(I,J)＊C(I,J)
   ENDDO
ENDDO
!OCL END_SCACHE_ISOLATE_ASSIGN
!OCL END_SCACHE_ISOLATE_WAY
```

-Klto

オプション名	default
-Klto -Knolto	-Knolto

- 機能概要
  - リンク時最適化を行います。
  - -O1以上が有効な場合に意味があります。
- 注意
  - プログラムの翻訳時およびリンク時に指定する必要があります。
  - -gオプションまたは-Ncoverageオプションが有効な場合、-Kltoオプションは無効となります。
  - -Kltoオプションと-xdir=dir_nameオプションを同時に指定した場合、-xdir=dir_nameオプションは無効となります。

-Khpctag

オプション名	default
-Khpctag -Knohpctag	-Khpctag

- 機能概要
  - A64FXプロセッサのHPCタグアドレスオーバライド機能を利用して、タグを使用するコンパイラの最適化を有効にするかどうかを指示します。
  - HPCタグアドレスオーバライド機能は、アドレス上位8ビットに設定された情報(タグ)を使用して性能を向上させる機能です。
    コンパイラはプリフェッチやセクタキャッシュ制御の最適化でタグを設定しています。
  - 機能を抑止したい場合、-Knohpctagオプションを指定します。
  - -KA64FXオプションが有効な場合に意味があります。
- 効果
  - HPCタグアドレスオーバライド機能により、セクタキャッシュ機能やハードウェアプリフェッチアシスト機能（ストライドアクセスに対するハードウェアプリフェッチ機能など）が有効になるため、実行性能の向上が期待できます。
- 注意
  - プログラムの翻訳時およびリンク時に指定する必要があります。
  - タグは論理和が取られるため、アプリレベルでもタグを上書きしている場合などは、プリフェッチやセクタキャッシュ制御の情報として誤って解釈されて実行性能が低下する可能性があります。この場合、ジョブ実行時にHPCタグアドレスオーバーライト機能を無効化する必要があります。

-Nreordered_variable_stack

オプション名	default
-Nreordered_variable_stack -Nnoreordered_variable_stack	-Nnoreordered_variable_stack

- 機能概要
  - AUTOMATIC変数をスタック領域に割り付ける順序を、データサイズの昇順とするかどうかを指示します。
- 効果
  - データサイズの昇順にAUTOMATIC変数をスタック領域に割り付けます。データサイズが等しい場合はアライメントの昇順に、データサイズおよびアライメントが等しい場合はソースプログラム内の宣言文の記載順に割り付けます。
    データサイズの昇順にAUTOMATIC変数を割り付けることで、プログラム全体のスタック領域を減らすことができます。
- 注意
  - -Nnolineオプションおよび-g0オプションが有効な場合、割付け順序は保証されません。

-Ncoverage(1/2)

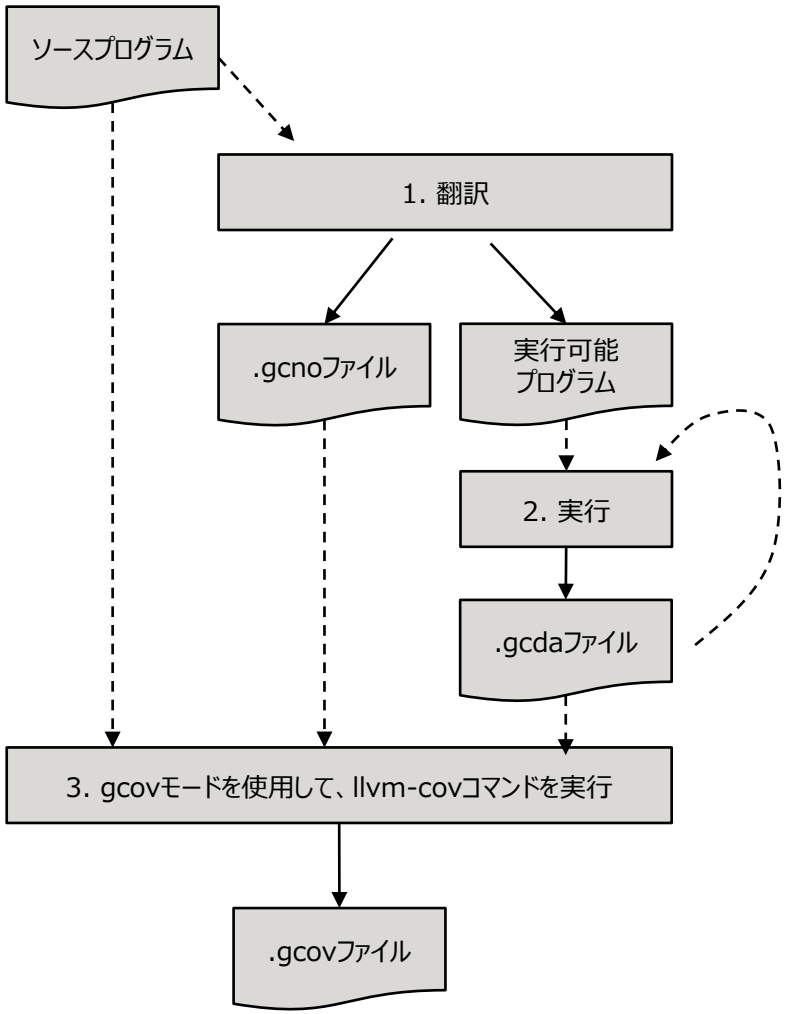
FUJITSU

オプション名	default
-Ncoverage -Nnocoverage	-Nnocoverage

- 機能概要
  - コードカバレッジ機能を利用するための情報を生成するかどうかを指示します。
- 効果
  - コードカバレッジ機能を利用するための情報を生成します。
- ポイント
  - -Ncoverageオプションは、プログラムの翻訳時およびリンク時に指定する必要があります。
- 注意
  - 実行回数を計測する命令を追加するため、実行性能が低下する場合があります。
  - -Kcmodel=largeオプションを必要とするプログラムでは、使用できません。
  - 以下の場合、実行回数を正しく計測できないことがあります。
    - 1行に複数の実行文が含まれる場合
    - STOP文、ERROR STOP文、EXITサービスサブルーチン、およびSETRCDサービスサブルーチンが呼ばれた場合
    - 例外が捕捉された場合
    - ソースプログラムに#line指令が含まれる場合
    - コンパイラの最適化が適用された場合

-Ncoverage(2/2)

- 実用例
  - コードカバレッジ機能は以下の手順で使用します。
    1. 翻訳(-Ncoverage)
    2. 実行
    3. gcovモードを使用して、llvm-covコマンドを実行



.gcnoファイル	翻訳時に得られる情報を含んだバイナリファイル
.gcdaファイル	実行時に得られる情報を含んだバイナリファイル

-Kopenmp_collapse_except_innermost

FUJITSU

オプション名	default
-Kopenmp_collapse_except_innermost -Kopenmp_nocollapse_except_innermost	-Kopenmp_nocollapse_except_innermost

- 機能概要
  - OpenMPのループ構文において、次の条件をすべて満たしている場合に、最内ループをCOLLAPSE指示子の対象から外します。
    - 最内ループまで含めたCOLLAPSE指示子が指定されている
    - 最内ループまで含めたCOLLAPSE指示子により実行性能が低下する可能性があると、コンパイラが翻訳時に判断できる
  - -Kopenmpオプションが有効な場合に意味があります。
- 効果
  - 最内ループのSIMD化が阻害されなくなり、COLLAPSE指示子による実行性能の低下を防止できる場合があります。
- 実用例

```
38  !$OMP PARALLEL
39  !$OMP DO PRIVATE(K,J,I) COLLAPSE(3)
40    DO K=1,L            ! L = 2
41      DO J=1,M          ! M = 512
42        DO I=1,N        ! N =32
43          A(I,J,K)=B(I,J,K)+C(I,J,K)
44        ENDDO
45      ENDDO
46    ENDDO
47  !$OMP END DO
48  !$OMP END PARALLEL
```

-Kopenmp_nocollapse_except_innermost有効時

```
line 39: 最内ループまで含めたCOLLAPSE指示節が指定されたため、
実行性能が低下する可能性があります。
```

-Kopenmp_collapse_except_innermost有効時

```
line 39: 最内ループをCOLLAPSEの対象から外しました。
```

-Nfjomplib/-Nlibomp

オプション名	default
-Nfjomplib	-Nlibomp
-Nlibomp	

- 機能概要
  - 並列処理に使用するライブラリを指定します。
  - -Nfjomplibオプションを指定した場合、並列処理に富士通OpenMPライブラリを使用します。
  - -Nlibompオプションを指定した場合、並列処理にLLVM OpenMPライブラリを使用します。

OpenMPライブラリ	オプション	サポート機能
LLVM OpenMPライブラリ	-Nlibomp (デフォルト)	OpenMP4.5 と 5.0 の一部 ハードウェアバリア(デフォルトはソフトウェアバリア) セクタキャッシュ デフォルトでコアバインド
富士通OpenMPライブラリ	-Nfjomplib	OpenMP3.1 ハードウェアバリア セクタキャッシュ

- ポイント
  - リンク時に指定する必要があります。
- 注意
  - ライブラリ選択オプションは、自動並列化(-Kparallelオプション)においても有効となります。

最適化の調整

FUJITSU

- 目的
  - プログラムの特徴に合わせて、最適化を調整します。
- ポイント
  - 次の方針で最適化を実施します。

引数	最適化方針
shortloop	• 最内ループの繰返し数が翻訳時に不明な場合は、繰返し数が大きいとみなします。
memory_bandwidth	• 最内ループをメモリバンド幅がボトルネックではないとみなし、CPU演算のボトルネックを優先的に解消します。
time_saving_compilation	• 翻訳時間を消費するプログラムでも実行可能プログラムの高速化を優先します。

  - 複数の-Kassumeオプションを同時に指定することもできます。

ASSUME(SHORTLOOP)

オプション名	default
-Kassume={shortloop|noshortloop}	-Kassume=noshortloop

最適化指示子
ASSUME({SHORTLOOP|NOSHORTLOOP})

- 機能概要
  - プログラム中の最内ループの繰り返し数が翻訳時に不明な場合に、繰り返し数が少ないとみなして、最適化を行います。
  - 自動並列化、ループアンローリング、ソフトウェアパイプライニングなどの最適化が調整または抑止される可能性があります。

最適化	最適化の動作
自動並列化	• 最内ループは抑止
ループブロッキング	• 最内ループのブロック化は抑止
zfill	• 抑止
prefetch	• 最内ループは抑止
ループアンローリング	• 抑止
ループリダクション	• ループアンローリングが抑止のため抑止
外側ループアンローリング	• 抑止
ソフトウェアパイプライニング	• 緩いソフトウェアパイプライニングを実施

  - -O1以上で有効です。
- 効果
  - 繰り返し数が少ないループに適切な最適化が実施されるため、、実行性能の向上が期待できます。

ASSUME(MEMORY_BANDWIDTH)

オプション名	default
-Kassume={memory_bandwidth| nomemory_bandwidth}	-Kassume=nomemory_bandwidth

最適化指示子
ASSUME({MEMORY_BANDWIDTH|NOMEMORY_BANDWIDTH})

- 機能概要
  - プログラム中の最内ループでメモリバンド幅がボトルネックになるとみなして最適化を行います。
  - zfillの最適化の促進およびソフトウェアパイプライニングなどの最適化が調整または抑止される可能性があります。

最適化	最適化の動作
ループ融合	• 抑止
ループ分割	• 1ループのストリーム数を防ぐ
zfill	• 積極的実行
prefetch	• Prefetch幅を動的に制御
ソフトウェアパイプライニング	• 緩いソフトウェアパイプライニングを実施

  - -O1以上で有効です。
- 効果
  - メモリバンド幅の軽減を狙ったデータの局所性を促す最適化やキャッシュに関連する最適化が実施されるため、実行性能の向上が期待できます。

ASSUME(TIME_SAVING_COMPILATION)

オプション名	default
-Kassume={time_saving_compilation|notime_saving_compilation}	-Kassume=notime_saving_compilation

最適化指示子
ASSUME({TIME_SAVING_COMPILATION|NOTIME_SAVING_COMPILATION})

- 機能概要
  - 翻訳時間が短くなるように最適化を行います。

最適化	最適化の動作
インライン展開	• 閾値制御、インライン展開し過ぎによる命令増加の防止
ループ融合	• 抑止
ループアンスイッチング	• OCL以外抑止
ループアンローリング	• アンローリング実施対象ループをアンローリング効果があるケースに絞る

  - -O1以上で有効です。
- 効果
  - 特定の最適化の動作や抑止を制御して翻訳時間を節約するため、翻訳性能の向上が期待できます。
- ポイント
  - -Oオプションのレベル制御のように、ある特定機能が止まるのではなく、プログラムが巨大になるほど最適化が制限、抑止される。

FUJITSU

使用時に注意が必要なオプション(最適化の副作用)

- 実行に影響を及ぼす可能性があるオプション
- 計算誤差を伴うオプション
- 計算誤差を抑えるオプション

実行時に影響を及ぼす可能性があるオプション

FUJITSU

投機実行を伴う命令移動を行うため、実行時異常終了する可能性があります。

オプション	機能	最適化の副作用	回避方法
-Kpreex	不変式の先行評価を行います。 （ループ内にあるIF文内から不変式をループ外に移動します）	実行時異常終了する可能性があります。	いずれかの方法で回避できます。 • -Kpreexオプションを削除します。 • -Kpreexオプションより後ろに-Knopreexオプションを指定します。
-Ksimd=2	IF構文をSIMD化します。 （IF構文内の式を投機実行します）		いずれかの方法で回避できます。 • -Ksimd=1オプションに変更します。 • -Ksimd=2より後ろに-Knosimdを指定します。
-Kpreload	ロード命令を投機実行します。 （ループ内にあるIF文内からロード命令をIF文の前に移動します）		いずれかの方法で回避できます。 • -Kpreloadオプションを削除します。 • -Kpreloadオプションより後ろに-Knopreloadオプションを指定します。

計算誤差を伴うオプション

FUJITSU

下記オプションは計算誤差が発生する可能性があります。

オプション	機能	最適化の副作用	回避方法
-Keval	x*y + x*z ⇒ x*(y+z) 上記のような演算評価方法の変更を行います。 -Kfast指定時は-Kevalが誘導されます。	演算評価方法の変更を行うため、計算誤差が発生する可能性があります。	• -Kfastより後ろに-Knoevalを指定します。
-Kfp_contract	multiply add/subtract 浮動小数点命令を出力します。 -Kfast指定時は-Kfp_contractが誘導されます。	multiply add/subtract浮動小数点命令が出力されるため、計算誤差が発生する可能性があります。	• -Kfastより後ろに-Knofp_contractを指定します。
-Kfp_relaxed	浮動小数点除算またはSQRT関数を逆数近似で演算します。 -Kfast指定時は-Kfp_relaxedが誘導されます。	逆数近似命令が出力されるため、計算誤差が発生する可能性があります。	• -Kfastより後ろに-Knofp_relaxedを指定します。
-Kilfunc	組込み関数のインライン展開を行います。 -Kfast指定時、-Kilfuncが誘導されます。	組込み関数のインライン展開では、逆数近似演算命令や三角関数補助命令などを利用したアルゴリズムを用いるため、計算誤差が発生する可能性があります。 atan2は-X03または-X08を指定した場合にもFortran95の言語仕様で計算されます。	• -Kfastより後ろに-Knoilfuncを指定します。

計算誤差を抑えるオプション(1/2)

FUJITSU

下記オプション指定により、計算誤差を抑えることができます。

オプション	機能
-Kfp_precision	浮動小数点演算の計算誤差が生じない以下のオプションの組合せを誘導します。 • -Knoeval(-Knofsimple、-Knoredution、-Ksimd_noreduction_productが誘導される) • -Knofp_contract • -Knofp_relaxed • -Knofz • -Knoilfunc • -Knomfunc • -Kparallel_fp_precision(-Kopenmp_ordered_reductionが誘導される) -Knofp_precisionは、-Kfp_precisionの指定のみを無効にします。 以下の順にオプションを解析するため、-Kfp_precisionが誘導する個々のオプションを別途指定しても、別途指定したオプションには影響しません。 1.-Kfp_precisionと-Knofp_precisionのどちらが有効か解析 2.-Kfp_precisionが有効な場合、その指定位置にオプションの組合せを展開 翻訳時プロフィルファイルに設定されている-Kfp_precisionを無効にしたい場合は、-Knofp_precisionを指定します。

計算誤差を抑えるオプション(2/2)

FUJITSU

オプション	機能
-Kparallel_fp_precision	スレッド並列数の違いで浮動小数点型または複素数型の演算結果に計算誤差が生じる場合に、コンパイラが最適化を抑止します。 一部の最適化が抑止されるため、実行性能が低下する可能性があります。 OpenMPのREDUCTION指示節が指定された場合は、演算の評価順序が変更されるため、-Kparalle_fp_precisionを指定しても計算誤差が生じる可能性があります。 スレッド並列数の違いで計算誤差が生じない実行プログラムを作成する場合に -Kparalle_fp_precisionを指定します。 スレッド並列数の違いにより計算誤差が生じても構わない場合は、 -Kparallel_nofp_precisionを指定します。

翻訳情報

- 診断メッセージ/ガイダンスメッセージ
- 翻訳情報の見方
- リスタ(プログラムリスト/最適化情報/統計情報)
- 翻訳情報の注意事項

診断メッセージ/ガイダンスメッセージ

FUJITSU

- 診断メッセージ
  - 以下の場合に診断メッセージを出力します。
    - 翻訳コマンドに指定されたオペランドに誤りがある
    - プログラムに誤りがある
    - 利用者に有用な情報がある
    - 注意すべき事項がある

翻訳コマンドによる診断メッセージ　**frtpx：メッセージ本文**

コンパイラによる診断メッセージ　**jwd*xxxxz-y*　ファイル名　行番号　桁位置　メッセージ本文**

- ガイダンスメッセージ
  - 翻訳時オプション –Koptmsg=guide を指定すると、以下の最適化に関するガイダンスメッセージ（最適化できなかった原因、対処方法）を出力します。
    - SIMD化
    - 自動並列化
    - ソフトウェアパイプライニング
    - インライン展開

jwdxxxxz-y　診断メッセージ

[ガイダンス]

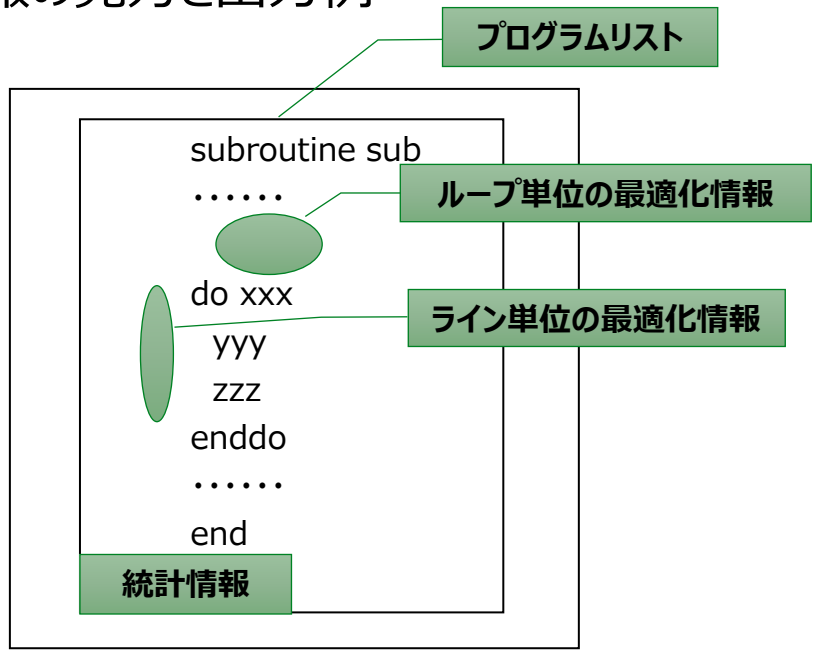
ガイダンスメッセージ本文

翻訳情報の見方

コンパイラの最適化の前提知識として翻訳情報の見方と出力例を説明します。

翻訳情報は以下の情報を出力します。

- プログラムリスト
- ループ単位の最適化情報
- ライン単位の最適化情報
- 統計情報
- エラーメッセージ



翻訳オプション形式：

-Nlst[=*lst_arg*]　*lst_arg*:{ a | d | i | m | p | t | x }

翻訳例

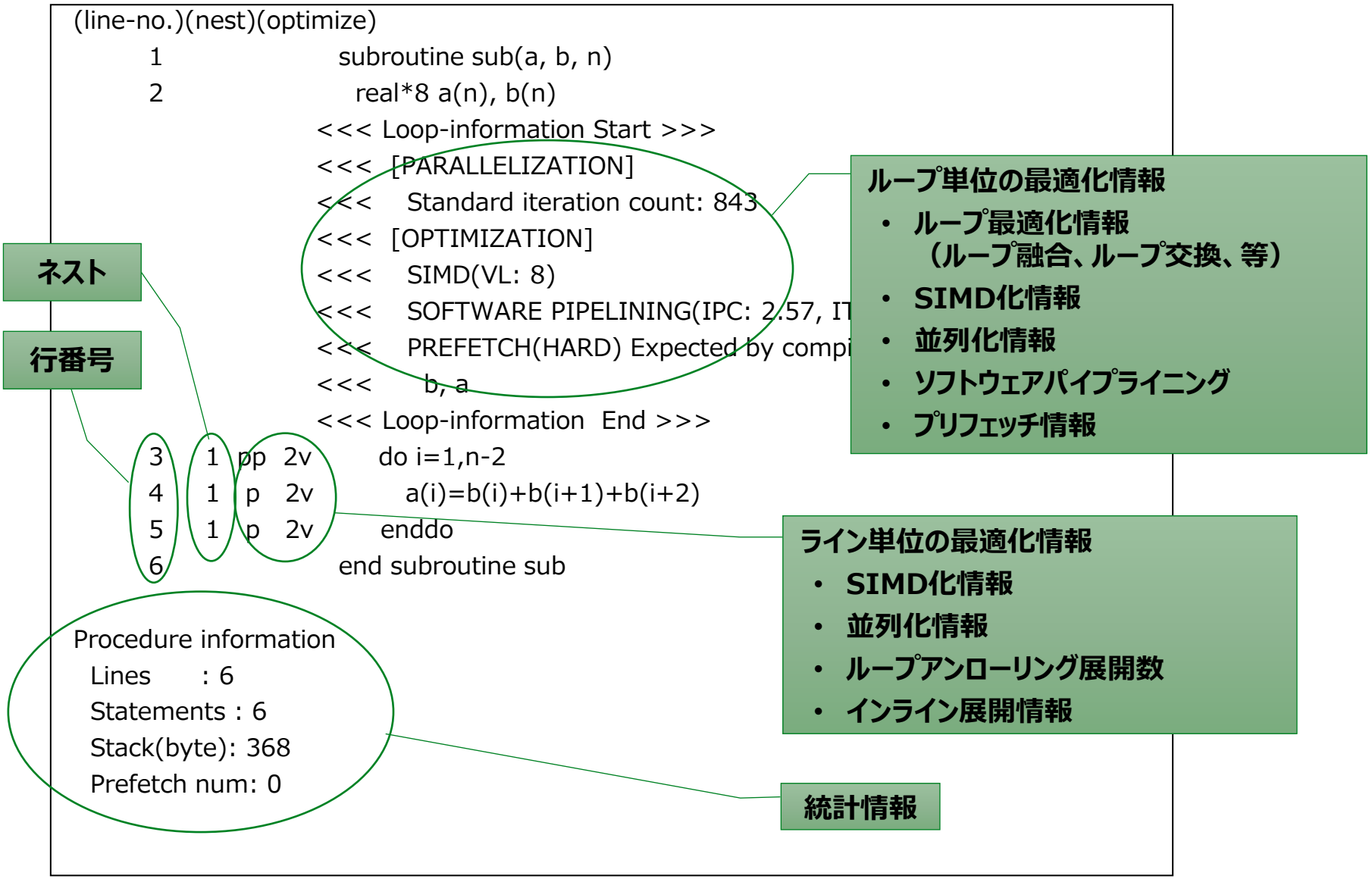
```
$ frtpx –Kfast,parallel –Nlst=t sample.f90
```

→sample.lstを翻訳情報として出力

リスタ(プログラムリスト/最適化情報/統計情報)

FUJITSU

- 翻訳情報の出力形式



翻訳情報の注意事項(1/2)

FUJITSU

- 翻訳時オプション-Nlst=pまたは、-Nlst=tで出力される翻訳情報(最適化情報)は、次のように正しく出力されない
  場合があります。
  - インライン展開が行われた場合、最適化の動作はインライン展開された場所によって異なる可能性があります
    - 次のように出力される可能性があります。
    - 1つのループに対して、複数の最適化メッセージが出力される
    - 翻訳情報(最適化情報)と反対の意味の最適化メッセージが出力される
    - 翻訳情報(最適化情報)が出力されない
    - インライン展開された関数では、prefetch数は累計で出力されます。
  - SIMD化、自動並列化、ループ融合、ループ分割など詳細な最適化情報として出力される最適化が、1つのループに対 して複数適用される
    - 次のように出力される可能性があります。
    - 行番号に対する最適化情報がずれる
    - 翻訳情報(最適化情報)と反対の意味の最適化メッセージが出力される
  - ループアンスイッチング最適化は、IF構文の条件が成立した場合のループと成立しない場合のループを生成しますが、その複数のループのうちいずれかの翻訳情報と最適化メッセージが出力されます。また、行番号に対する最適化情報が出力されない場合があります。
  - 1行に複数のループを記述した場合、複数ループのうちいずれかの最適化情報が出力されることになります。最適化情報を出力したいループは可能な限り、同一行に記述することは避けてください。
  - 条件文とGOTO文で構成されるループでは詳細な最適化情報は出力されません。行番号に対する最適化情報もずれる可能性があります。

翻訳情報の注意事項(2/2)

FUJITSU

- 次のいずれかの条件を満たす場合、コンパイラはループを生成することがあります。
  翻訳時オプション-Nlst=pまたは、-Nlst=tで出力される翻訳情報(最適化情報)および最適化メッセージが、その生成ループに対して出力されることがあります。
  - 実引数がポインタ配列、形状引継ぎ配列または、部分配列であり、対応する仮引数がポインタでも形状引継ぎ配列でもない
  - 翻訳時オプション-Nquickdbg=undef、-Nquickdbg=undefnanまたは、-Nsetvalue=array が有効である
  - OpenMPのFIRSTPRIVATE、LASTPRIVATE、REDUCTION、COPYINまたは、 COPYPRIVATE指示節に配列の変数名が現れる
  - 翻訳時オプション-Karray_privateまたは最適化指示子ARRAY_PRIVATE、FIRST_PRIVATE、LAST_PRIVATEにより、自動並列化されたループがある。
- リンク時最適化が適用された場合、翻訳情報で表示される最適化とは異なった最適化が実行時に適用されることがあります。
- ソースプログラムに#line指令が含まれている場合、翻訳時オプション-Nlst=pまたは-Nlst=tで出力される翻訳情報
  (最適化情報)は、#line指令に指定した行番号を元に出力されます。
  例えば、ソースプログラムの10行目のDO文に対し、#line指令で3行目と指定していた場合、DO文の翻訳情報
  (最適化情報)はソースプログラムの3行目に対して出力されます。

FUJITSU

プログラムの実行

- 実行コマンド
- スレッド並列実行
- OpenMPライブラリの組み合わせ
- LLVM OpenMPライブラリと富士通OpenMPライブラリ
  - LLVM OpenMP ライブラリ (-Nlibomp)
  - 富士通 OpenMPライブラリ (-Nfjomplib)

実行コマンド

- 逐次実行の場合
  - コンパイル時に生成された実行可能プログラムを実行します。

```
./[実行可能プログラム]
```

  - 例）逐次実行のスクリプト

```
#!/bin/sh
#PJM -L "node=1"                    # ノード数

./a.out
```

- スレッド並列（自動並列、OpenMP）実行の場合
  - 環境変数OMP_NUM_THREADSに並列動作するスレッド数を指定して、実行可能プログラムを実行します。

```
OMP_NUM_THREADS={スレッド数} ;export OMP_NUM_THREADS
./[実行可能プログラム]
```

  - 例）スレッド並列（自動並列またはOpenMP）実行のスクリプト

```
#!/bin/sh
#PJM -L "node=1"                    # ノード数
OMP_NUM_THREADS=16 ;export OMP_NUM_THREADS
./a.out
```

- ※ジョブの標準出力、標準エラー出力は以下のファイルに出力されます。
- {pjm-script} .{req-id}.out ：標準出力
- {pjm-script} .{req-id}.err ：標準エラー出力

pjm-script：PJMのスクリプト名，req-id：リクエスト番号

スレッド並列実行

- スレッド並列ライブラリには以下の２つのライブラリがあります。
  - LLVM OpenMPライブラリ
  - 富士通OpenMPライブラリ
- どちらのライブラリを使用するかは、以下の翻訳時オプションで決定します。

オプション形式	機能概要
-Nfjomplib	富士通OpenMPライブラリを使用します。OpenMP 3.1までが利用可能です。
-Nlibomp	LLVM OpenMPライブラリを使用します。OpenMP 4.5とOpenMP 5.0の一部までが利用可能です。

  - 注意事項
    - -Nclangと-Nfjomplibを同時に指定した場合、警告を出力して、LLVM OpenMPライブラリを結合します(-Nlibomp相当の処理)。
    - FortranでOpenMP 4.0以降の機能を使用し、-Nfjomplibを指定した場合、リンクエラーになります。
    - clangモードオブジェクトとFortranの言語間結合する場合、-Nfjomplibを指定するとリンクエラーになります。

OpenMPライブラリの組合せ

FUJITSU

- OpenMPまたは自動並列で使用可能な組合せは以下になります。
  - 使用可能なOpenMP仕様を記載
  - 並列化オプション –Kopenmp または -Kparallel が指定されていることが前提

		LLVM OpenMPライブラリ (-Nlibomp)	富士通OpenMPライブラリ (-Nfjomplib)
C/C++ コンパイラ	tradモード	OpenMP 3.1 OpenMP 4.5の一部(注1)	OpenMP 3.1 OpenMP 4.5の一部(注4)
	clangモード (-Nclang)	OpenMP 4.5(注2) OpenMP 5.0の一部(注3)	使用不可
Fortran コンパイラ	-	OpenMP 4.5 OpenMP 5.0の一部(注3)	OpenMP 3.1 OpenMP 4.5の一部(注4)

注1) OpenMP 4.5の、以下の機能が使用できます。
- simd構文
- declare simd構文
- parallel構文のproc_bind指示節
- task構文のdepend指示節
- taskgroup構文

注2) OpenMP 4.5の、以下の機能は使用できません。
- declare simd構文

注3) OpenMP 5.0の、以下の機能が使用できます。
- task構文のin_reduction指示節
- taskgroup構文のtask_reduction指示節
- taskloop構文のreduction指示節およびin_reduction指示節

注4) OpenMP 4.5の、以下の機能が使用できます。
- simd構文
- declare simd構文

LLVM OpenMPライブラリと富士通OpenMPライブラリ

FUJITSU

- LLVM OpenMP ライブラリとは、LLVM OpenMP Runtime LibraryをベースにA64FX向けに拡張したOpenMP ライブラリです。

OpenMPライブラリ	オプション	サポート機能
LLVM OpenMPライブラリ	-Nlibomp （デフォルト）	OpenMP 4.5 と 5.0 の一部 ハードウェアバリア（デフォルトはソフトウェアバリア） セクタキャッシュ コアバインド（デフォルト）
富士通OpenMPライブラリ	-Nfjomplib	OpenMP 3.1 ハードウェアバリア セクタキャッシュ コアバインド（ジョブ実行時はデフォルト）

- 選択方法
  - 翻訳時オプション（リンク時）で選択
    - -Nlibomp (デフォルト)：LLVM OpenMPライブラリを使用
    - -Nfjomplib：富士通開発のOpenMPライブラリを使用
- 仕様の違い
  - スレッドスタックの大きさ

オプション	デフォルトの大きさ	大きさ変更用の環境変数
-Nlibomp	• 8MiB	OMP_STACKSIZE
-Nfjomplib	• プロセススタックのサイズを継承 • プロセススタックサイズがunlimited指定の場合 (メモリサイズ / スレッド数) / 5	OMP_STACKSIZE または THREAD_STACK_SIZE

LLVM OpenMP ライブラリ (-Nlibomp) (1/2)

FUJITSU

- デフォルトの状態でソフトウェアバリア/セクタキャッシュが利用できます。

- ハードウェアバリアを利用するためには、環境変数 FLIB_BARRIERを指定します。
  - FLIB_BARRIER=HARD：ハードウェアバリアを利用します。
  - FLIB_BARRIER=SOFT ：ソフトウェアバリアを利用します。（デフォルト）

- ハードウェアバリアを利用する場合の注意事項
  - スレッド数の制御(omp_set_num_threads()、num_threads 指示節)はできません。
  - スレッドアフィニティ(proc_bind指示節、OMP_PLACES、OMP_PROC_BIND)の制御はできません。
  - ネスト制御(omp_set_nested() 、OMP_NESTED)はできません。
  - タスク構文において常に即時実行タスク(undeffered task)が生成され、タスクは並列実行されません。
  - キャンセレーションは利用はできません

- OpenMP 4.5 に加えて OpenMP 5.0 の一部をサポートしています。
  最新の OpenMP規格を利用する場合は本ライブラリを利用してください。

- タスクやキャンセレーションなどの OpenMP4.5/5.0 の主要機能を利用する場合は、
  ソフトウェアバリアを選択してください。

LLVM OpenMP ライブラリ (-Nlibomp) (2/2)

FUJITSU

- 初期スレッドを特定のコアにバインドする場合は、numactl や taskset を利用してください。プログラム中の上限スレッド数に影響はありません。

```
#!/bin/sh -ex
  :
export FLIB_BARRIER=HARD   # ハードウェアバリアを利用
                           # （FLIB_BARRIERがSOFTか、未設定時はソフトウェアバリアになります）
numactl -C12 ./a.out       # 初期スレッドをC12に固定
```

- MPIプログラムの場合、MPIライブラリ内メモリコピー処理のスレッド並列化機能を利用することができます。翻訳／結合コマンドのオプションとして、-Kparallelと-Kopenmpの両方またはどちらか1つと、-Nlibompを指定してください。
  - mpifrtpx –Kopenmp –Nlibomp a.f90

- 以下の富士通 OpenMPライブラリ独自の環境変数は利用できません。設定しても無視されます。
  - PARALLEL
  - FLIB_FASTOMP
  - THREAD_STACK_SIZE
  - FLIB_SPINWAIT
  - FLIB_CPU_AFFINITY
  - FLIB_NOHARDBARRIER
  - FLIB_HARDBARRIER_MESSAGE
  - FLIB_CNTL_BARRIER_ERR
  - FLIB_PTHREAD
  - FLIB_CPUBIND
  - FLIB_USE_ALLCPU
  - FLIB_USE_CPURESOURCE

富士通 OpenMPライブラリ (-Nfjomplib)

FUJITSU

- 基本的な仕様は従来システム(京/FX100)と同じであるため、詳細な説明は省略します。
- デフォルトの状態でハードウェアバリア/セクタキャッシュが利用できます。
- ハードウェアバリア利用時に即時実行タスクが生成される制約はありません。
- OpenMP 3.1 をサポートしています。
- Fortran、C/C++ tradモードで使用する場合は -Nfjomplib をリンク時に指定してください。
  - Fortran
    - frtpx -Kopenmp -Nfjomplib main.f90
  - C/C++ (tradモード)
    - fccpx -Kopenmp -Nfjomplib main.c
- C/C++コンパイラの clangモードでは –Nfjomplib は使用できません。
- ジョブ実行時、スレッドはコアバインドされます。ジョブ実行しない場合、スレッドはコアバインドされませんが、環境変数FLIB_CPU_AFFINITYを使用することでコアバインドの制御が行えます。
- 以下のLLVM OpenMPライブラリ独自の環境変数は利用できません。設定しても無視されます。
  - FLIB_BARRIER

FUJITSU

プログラム作成時に注意すべきこと

- SIMD化可能なループの条件
- 自動並列化可能なループの条件
- 富士通拡張オブジェクトの仕様
- bit演算子を用いたリダクション演算のSIMD化
- ラージページ
- 組込み手続の変形関数引用時の注意事項

SIMD化可能なループの条件

FUJITSU

- ループを実行する前に、ループ繰り返し数が決定している
  - -Ksimd_uncounted_loopオプションの併用でSIMD化できる場合がある
- ループ内に分岐（IF文など）がない
  - IF文の真率などによっては、マスク付きSIMDの効果を得られる場合がある
- ループ内にサブルーチン呼出しがない
  - インライン展開の併用でSIMD化できる場合がある
- ループ内にある式の右辺と左辺で配列領域の重なりがない
- ループのn回目の計算が、n-1回目の計算結果に依存しない

自動並列化可能なループの条件(1/3)

FUJITSU

- DOループ(多重DOループも含む)および、配列操作の文(配列式、配列代入)
  - 次の場合は自動並列化対象から除外する。

① 並列実行すると実行時間が短縮されないと予想される場合

```
! 繰り返し数が少なく、演算数が少ないDOループは自動並列化の対象とはならない
DO I=1, 10
  A(I) = A(I) + B(I)
ENDDO
```

② ループスライスの対象とならない型の演算を含む場合

以下の型の演算を含むDOループは自動並列化されません。内部DOループに含まれる場合も同様です。

- 文字型
- 派生型

③ 外部手続、内部手続またはモジュール手続の引用を含む場合

```
! サブルーチンの引用が含まれているループは自動並列化の対象とはならない
DO J=1,10
  DO I=1,10000
    A(I,J) = A(I,J) + B(I,J)
    CALL SUB(A)
ENDDO
```

自動並列化可能なループの条件(2/3)

FUJITSU

④ DOループの形が複雑な場合

以下に示すDOループは、形が複雑なため、自動並列化の対象とはなりません。

- DOループの内側から外側へ飛出しがあるDOループ

```
DO J=1,10
  DO I=1,10000
    A(I,J) = A(I,J) + B(I,J)
    IF (A(I,J).LT.0) GOTO 20
  ENDDO
ENDDO
...
20 CONTINUE
```

- 複雑な構造のDOループ

```
DO J=1,10000
  IF(N>0) THEN
    ASSIGN 10 TO I
  ELSE IF(N<0) THEN
    ASSIGN 20 TO I
  ELSE
    ASSIGN 30 TO I
  ENDIF
  GOTO I,(10,20,30)
10 A(J)=A(J)+0
20 A(J)=A(J)+1
30 A(J)=A(J)+2
  ENDDO
```

自動並列化可能なループの条件(3/3)

FUJITSU

⑤ 入出力文または組込みサブルーチンを含む場合

入出力文または組込みサブルーチンを含むDOループは、自動並列化の対象とはなりません。

⑥ データの定義引用順序が逐次実行のときと変わるおそれがある場合

データの定義引用順序が逐次実行のときと変わるDOループは、自動並列化の対象とはなりません。

```
DO I=2,10000
  A(I) = A(I-1) + B(I)
ENDDO
```

富士通拡張オブジェクトの仕様

FUJITSU

- コンパイラオプションによるオブジェクトのパターン

ARMv8（NEON）		富士通拡張なし	-KGENERIC_CPU –KNOSVE
		富士通拡張あり	-KA64FX –KNOSVE
ARMv8+SVE	**可変長**	富士通拡張なし	-KSVE –Ksimd_reg_size=agnostic
		富士通拡張あり	-KA64FX –Ksimd_reg_size=agnostic
	固定長	富士通拡張なし	-KSVE
		富士通拡張あり	-KA64FX

- -KA64FX
  - A64FXプロセッサ向けのオブジェクトファイルを出力するように指示します
- -KGENERIC_CPU
  - Armプロセッサ向けのオブジェクトファイルを出力するように指示します
- -Ksimd_reg_size=agnostic
  - SVEのベクトルレジスタを特定のサイズとみなさず翻訳を行い、実行時にSVEのベクトルレジスタサイズを決定する実行可能プログラムを作成します
- -KSVE
  - Armv8-Aアーキテクチャの拡張であるSVEを利用したオブジェクトファイルを出力することを指示します
- -KNOSVE
  - SVEを利用しないオブジェクトファイルを出力することを指示します

bit演算子を用いたリダクション演算のSIMD化

FUJITSU

- リダクション対象のbit演算子
  - ADD、MULT、MAX、MINのリダクション演算に加えて、bit演算子を用いたリダクション演算をsimd化の対象とすることにより、実行性能の改善を測ります。
  - 次のパターンをリダクション演算として認識して、simd化対象とします。
    - A = IAND(A, B)
    - A = IOR(A,B)
    - A = IEOR(A,B)
    - A = IALL(B) (*1)
    - A = IANY(B) (*1)
    - A = IPARITY(B) (*1)
    - A = ALL(B) (*1)
    - A = ANY(B) (*1)

    (*1) インライン展開されたとき、リダクション演算として認識する。
  - Aのサイズが Bのサイズより小さい場合は、桁溢れが発生する可能性があるため、対象外とします。
  - 符号あり整数型、符号なし整数型および論理型を対象とします。

ラージページ(1/5)

FUJITSU

- ラージページとは
  - 大規模なデータを扱うアプリケーションに対して、通常のページ(ノーマルページ)より大きなページサイズのメモリ(ラージページ)を割り当てることで、
    - CPUのアドレス変換処理によるオーバーヘッドを低減します。
    - メモリアクセス性能を向上させます。
  - A64FXシステム環境での、ノーマルページのサイズは64KiBであり、ラージページとして利用可能なサイズは、2MiBです。
    - 環境変数設定により以下の動作設定が可能
      - ✓ ラージページ割り当て動作の有効化/無効化
      - ✓ スタック領域のラージページ割り当て動作の有効化/無効化
      - ✓ 各メモリ領域のページング方式(ページの割り当て契機)の選択
    - 32MiB/1GiB/16GiBなどの様々なページサイズはMcKernelで実現可能。

ラージページ(2/5)

FUJITSU

- ラージページ仕様

メモリ領域	MP10/FX10/FX100	A64FX			
	ページサイズ	ページサイズ			ページング(_付はdefault)
		ノーマルページ	ラージページ base	ラージページ base+stack (default)	
テキスト (.text)	8KiB	64KiB	64KiB	64KiB	–
静的データ (.data)	4MiB (default), 8KiB, 32MiB, 256MiB	64KiB	2MiB	2MiB	常にprepage
静的データ (.bss)		64KiB	2MiB	2MiB	demand | prepage
スタック (*1)		64KiB	64KiB	2MiB	demand | prepage
動的メモリ (*2)		64KiB	2MiB	2MiB	demand | prepage
共有メモリ		64KiB	64KiB	64KiB	–

＊１ ：プロセススタック/メインスレッド用スタック/スレッドスタック領域が対象
＊２ ： プロセスヒープ/メインスレッド用ヒープ/スレッドヒープ/mmap領域が対象

ラージページ(3/5)

FUJITSU

- ラージページ設定用環境変数
  - 基本設定/ページング方式の設定

環境変数名	指定値 (_付はdefault)	説明
XOS_MMM_L_HPAGE_TYPE	hugetlbfs | none	ラージページライブラリによるラージページ割り当て動作の有効化/無効化を選択する設定です。 「hugetlbfs」の場合、HugeTLBfsによるラージページ化をします。 「none」の場合、ラージページライブラリによるラージページ化は行いません。
XOS_MMM_L_LPG_MODE	base+stack | base	スタック領域およびスレッドスタック領域のラージページ割り当て動作の有効化/無効化を選択する設定です。 「base+stack」の場合、静的データおよび動的メモリ確保領域だけではなく、スタック領域およびスレッドスタック領域もラージページ化します。 「base」の場合、静的データおよび動的メモリ確保領域のみラージページ化します。スタック領域およびスレッドスタック領域はラージページ化しません。
XOS_MMM_L_PAGING_POLICY	[demand | prepage]: [demand | prepage]: [demand | prepage]	各メモリ領域のページング方式(ページの割り当て契機)を選択する設定です。 「demand」はデマンドページング方式、「prepage」はプリページング方式を意味します。本変数はコロン(:)区切りで3つのメモリ領域のページング方式を指定します。 第1指定は、静的データの.bss領域です。(静的データの.data領域はページング方式指定の対象外で常にprepageとなります。) 第2指定は、スタック領域およびスレッドスタック領域です。 第3指定は、動的メモリ確保領域です。 指定値以外の値を指定した場合は、「prepage:demand:prepage」を指定したものとみなします。

ラージページ(4/5)

FUJITSU

●チューニング用の設定 (ラージページライブラリ独自の環境変数)

環境変数名	指定値 (_付はdefault)	説明
XOS_MMM_L_ARENA_FREE	1 | 2	free(3)で解放されるヒープ領域の扱いに関する設定です。 「1」を指定した場合は、解放可能なメモリを即時に解放します。「2」を指定した場合は、メモリを一切解放せず、全メモリをプールして再利用します。
XOS_MMM_L_ARENA_LOCK_TYPE	0 | 1	メモリ割り当てポリシーに関する設定です。 「0」はメモリ獲得性能優先、「1」はメモリ使用効率優先を意味します。
XOS_MMM_L_MAX_ARENA_NUM	1 以上INT_MAX 以下の整数値 [10進数]	生成可能なアリーナ(プロセスヒープとスレッドヒープ領域の総和)の数を設定します。 XOS_MMM_L_ARENA_LOCK_TYPE=0のときに有効になります。
XOS_MMM_L_HEAP_SIZE_MB	MALLOC_MMAP_THRESHOLD_ の2倍 以上 ULONG_MAX以下の整数値<MiB単位> [10進数]	スレッドヒープ領域を使用する場合に、スレッドヒープ領域の生成時および拡張時に獲得するメモリサイズを設定します。
XOS_MMM_L_COLORING	0 | 1	キャッシュカラーリングの有無を設定します。プロセッサのL1キャッシュのコンフリクトを軽減します。「0」の場合、キャッシュカラーリングを行いません。 「1」の場合、MALLOC_MMAP_THRESHOLD_(デフォルトは128MiB)以上のサイズで行われるmmap(2)によるメモリ獲得時にはキャッシュカラーリングをします。
XOS_MMM_L_FORCE_MMAP_THRESHOLD	0 | 1	MALLOC_MMAP_THRESHOLD_(デフォルトは128MiB)以上のサイズのメモリ獲得時にmmap(2)を優先するかどうかの設定です。 「0」の場合、mmap(2) は優先しません。まずヒープ領域の空きを検索し、空きがあればヒープ領域の空きメモリを返します。ヒープ領域の空きが見つからないときにのみmmap(2) でメモリを獲得します。「1」の場合、mmap(2) を優先します。ヒープ領域の空きは検索せず、(例え空きがあっても)mmap(2) でメモリを獲得します。

- glibcの環境変数(MALLOC_MMAP_THRESHOLD_等)についてはユーザ向けガイドを参照

ラージページ(5/5)

- メモリ使用量に関する注意事項(副作用)について
  - 静的データ(.data)領域

    常に①、②の副作用が発生します。
  - 静的データ(.bss)領域

    以下の条件を満たす場合、①、②の副作用が発生します。

    a. .dynsym セクション(dynamic section)に存在するシンボルである
    b. シンボルが(メインプログラムの) bss 領域(bss_start, bss_end)の範囲内のアドレスにある
    c. シンボルがグローバル(STB_GLOBAL)またはウィーク(STB_WEAK)である
    d. シンボルのタイプが変数(STT_OBJECT)である
    e. シンボルのサイズが 0 より大きい
  - 副作用

    ① 2倍または3倍のメモリ領域を使用することがあります。

    ② XOS_MMM_L_PAGING_POLICYで静的データ(.bss)領域のページング方式をデマンドページング方式("demand")に設定していても、プリーページング方式("prepage")で動作します。

組込み手続の変形関数引用時の注意事項

- SPREADなどの変形関数引用時の注意事項
  - 実行時に、組込み変形関数を繰り返して呼び出した場合に、その組込み関数の実行コストが多く占める場合があります。
  - 組込み関数の実行コストが問題になる場合には、その組込み関数引用に換えて、それに相当する処理を行うプログラムを作成することで回避してください。

FUJITSU

Fortranがサポートするタイマー

- タイマーの仕様
- タイマーの精度

タイマーの仕様(1/3)

FUJITSU

● 主要なタイマールーチンの仕様

No.	ルーチン名	機能	形式	計測対象	ルーチンが返す単位
1	DATE_AND_TIME 組込みサブルーチン	日付と時間を取得します。	CALL DATE_AND_TIME ([DATE , TIME , ZONE , VALUES]) DATE：現在の日付がCCYYMMDD の形式で設定される。 （CC:世紀,YY:西暦年,MM:月,DD:日） TIME：現在の時間がhhmmss.sssの形式で設定される。 （hh:時,mm:分,ss.sss:秒） ZONE：UTC からの時差がshhmm の形式で設定される。 （s:符号,hh:時,mm:分） VALUES(1)：年 VALUES(2)：月 VALUES(3)：日 VALUES(4)：分で表したUTCからの時差。 VALUES(5)：時間 VALUES(6)：分 VALUES(7)：秒 VALUES(8)：ミリ秒	現在の日時	－
2	GETTIM サービスサブルーチン	現在の時刻を取得します。	CALL GETTIM (hour , minute , second , second1_100) hour：現在の時が設定される。 minute：現在の分が設定される。 second：現在の秒が設定される。 second1_100：現在の100分の1秒が設定される。	現在の時間	－
3	FDATE サービスサブルーチン	現在の日付と時刻をASCII コードに変換して、取得します。	CALL FDATE (string) string：現在の日付と時刻が曜日、月、日、時刻、年の順番で設定される。	現在の日付と時刻	－
4	ITIME サービスサブルーチン	現在の時、分、秒を取得します。	CALL ITIME (ia) ia(1)：現在の時が設定される。 ia(2)：現在の分が設定される。 ia(3)：現在の秒が設定される。	現在の時、分、秒	－
5	GETTOD サービスサブルーチン	現在の実時間を取得します。実時間は過去のある任意の時間からのマイクロ秒単位の経過時間です。 通常、システムブートからの時間で表されます。	CALL GETTOD (g) g：マイクロ秒単位の経過時間が設定される。	wall clock time	マイクロ秒

タイマーの仕様(2/3)

FUJITSU

● 主要なタイマールーチンの仕様

No.	ルーチン名	機能	形式	計測対象	ルーチンが返す単位
6	GMTIME サービスサブルーチン	指定したシステム時間をグリニッジ標準時間に従って、秒、分、時間、日、月、年、曜日、1月1日からの 通算日付、夏時間かどうかを表す情報を取得します。	CALL GMTIME (time , t) time：システム時間を指定する。 timeに指定したシステム時間がグリニッジ標準時間に従い、以下の配列に設定される。 t(1)：秒 t(2)：分 t(3)：時 t(4)：日 t(5)：月 t(6)：1900年からの通算年 t(7)：日曜日からの通算曜日 t(8)：1月1日からの通算日 t(9)：標準時間は0、夏時間は1が設定される。	wall clock time	–
7	LTIME サービスサブルーチン	指定したシステム時間をローカル時間に従って、秒、分、時間、日、月、年、曜日、1月1日からの 通算日付、夏時間かどうかを表す情報を取得します。	CALL LTIME (time , t) time：システム時間を指定する。 timeに指定したシステム時間がローカル時間に従い、以下の配列に設定される。 t(1)：秒 t(2)：分 t(3)：時 t(4)：日 t(5)：月 t(6)：1900年からの通算年 t(7)：日曜日からの通算曜日 t(8)：1月1日からの通算日 t(9)：標準時間は0、夏時間は1が設定される。	wall clock time	–
8	omp_get_wtime ルーチン	現在の実時間を取得します。実時間は過去のある任意の時間からの秒単位の経過時間です。通常、システムブートからの時間で表されます。	y = omp_get_wtime （ ） y：秒単位の経過時間が返却される。	wall clock time	秒
9	SECNDS サービス関数	午前0時からのシステム時間の経過秒数から第1引数で指定した秒数を引いた秒数を取得します。	y = SECNDS (sec) y：午前0時からのシステム時間の経過秒数からsec 秒を引いた秒数が返却される。 sec：午前0時からのシステム時間の経過秒数から引く値を秒単位で指定する。	wall clock time	秒
10	SYSTEM_CLOCK 組込みサブルーチン	午前0時から現在までの通算時間を取得します。通算時間は秒単位の経過時間です。通常、システムブートからの時間で表されます。	CALL SYSTEM_CLOCK ([COUNT , COUNT_RATE , COUNT_MAX]) COUNT：午前0時から現在までの経過時間が設定される。 COUNT_RATE：1秒間に処理系が刻む回数(=1000)が設定される。 COUNT_MAX：COUNT の最大値(=86399999)が設定される。	wall clock time	ミリ秒
11	RTC サービス関数	1970 年1月1日午前0時からのUTC の通算秒を取得します。	y = RTC ( ) y：1970年1月1日午前0時からのUTC の通算秒が設定される。	wall clock time	秒

タイマーの仕様(3/3)

FUJITSU

● 主要なタイマールーチンの仕様

No.	ルーチン名	機能	形式	計測対象	ルーチンが返す単位
12	TIME サービス関数	00:00:00 GMT（1970年1月1日）からの秒単位の経過時間を取得します。	iy = TIME ( ) iy : 00:00:00 GMT（1970年1月1日）からの秒単位の経過時間が返却される。	wall clock time	秒
13	TIMEF サービス関数	直前に呼ばれたTIMEF サービス関数からの経過時間を返却します。	y = TIMEF ( ) y : 直前に実行されたTIMEFサービス関数からの経過時間が返却される。	wall clock time	秒
14	TIMER サービスサブルーチン	午前0時からの通算1/100秒を取得します。	CALL TIMER(ix) ix : 午前0時からの通算1/100秒が設定される。	wall clock time	秒
15	CLOCK サービスサブルーチン	実行可能プログラムの実行開始からのCPU 時間を取得します。CPU時間はプログラムが実行されているプロセスとそのプロセス内の全スレッドで利用されたCPU時間です。	CALL CLOCK (g , i1 , i2) g : i1で指定した単位のCPU時間が設定される。 i1 : 返却単位（秒単位、ミリ秒単位、マイクロ秒単位）を指定。 i2 : gに指定した変数の型を指定。	現行プロセスとそのプロセス内の全スレッドで利用されたCPU時間	指定に従い以下のいずれか。 ・秒 ・ミリ秒 ・マイクロ秒単位
16	CLOCKM サービスサブルーチン	実行可能プログラムの実行開始からのCPU 時間を取得します。CPU時間はプログラムが実行されているプロセスとそのプロセス内の全スレッドで利用されたCPU時間です。	CALL CLOCKM (i) i : ミリ秒単位のCPU時間が設定される。	現行プロセスとそのプロセス内の全スレッドで利用されたCPU時間	ミリ秒
17	CLOCKV サービスサブルーチン	実行可能プログラムの実行開始からのCPU 時間を取得します。CPU時間はプログラムが実行されているプロセスとそのプロセス内の全スレッドで利用されたCPU時間です。 ベクトル機の互換ルーチン。	CALL CLOCKV (g1 , g2 , i1 , i2) g1 : 常に0が設定される。※ベクトル機ではVUタイムが設定されていた。 g2 : i1で指定した単位のCPU時間が設定される。 i1 : 返却単位（秒単位、ミリ秒単位、マイクロ秒単位）を指定。 i2 : g2に指定した変数の型を指定。	現行プロセスとそのプロセス内の全スレッドで利用されたCPU時間	指定に従い以下のいずれか。 ・秒 ・ミリ秒 ・マイクロ秒単位
18	CPU_TIME 組込みサブルーチン	実行可能プログラムの実行開始からのCPU 時間を取得します。CPU時間はプログラムが実行されているプロセスとそのプロセス内の全スレッドで利用されたCPU時間です。	CALL CPU_TIME (TIME) TIME : 秒単位のCPU時間が設定される。	現行プロセスとそのプロセス内の全スレッドで利用されたCPU時間	秒
19	DTIME サービス関数	直前に呼ばれたDTIME サービス関数からのCPU 時間を取得します。CPU時間はプログラムが実行されているプロセスとそのプロセス内の全スレッドで利用されたCPU時間です。	y = DTIME (tm) y : 直前に呼ばれたDTIME サービス関数からのCPU 時間が返却される。 tm(1) : 秒単位のユーザCPU時間が設定される。 tm(2) : 秒単位のシステムCPU時間が設定される。	現行プロセスとそのプロセス内の全スレッドで利用されたCPU時間	秒
20	ETIME サービス関数	実行可能プログラムの実行開始からのCPU 時間を取得します。CPU時間はプログラムが実行されているプロセスとそのプロセス内の全スレッドで利用されたCPU時間です。	y = ETIME (tm) y : 実行可能プログラムの実行開始からのCPU 時間を返却される。 tm(1) : 秒単位のユーザCPU時間が設定される。 tm(2) : 秒単位のシステムCPU時間が設定される。	現行プロセスとそのプロセス内の全スレッドで利用されたCPU時間	秒
21	SECOND サービス関数	実行可能プログラムの実行開始からのユーザCPU 時間を取得します。CPU時間はプログラムが実行されているプロセスとそのプロセス内の全スレッドで利用されたCPU時間です。	y = SECOND ( ) y : 実行可能プログラムの実行開始時からのユーザCPU時間が秒単位で返却される。	現行プロセスとそのプロセス内の全スレッドで利用されたCPU時間	秒

タイマーの精度(1/2)

FUJITSU

- 主要なタイマールーチンの精度(タイマーのオーバーヘッド)

No.	ルーチン名	精度分解能	オーバーヘッド(μs)	スレッドセーフ性	実装	GCCオーバーヘッド(μs)	富士通/GCC	備考
1	DATE_AND_TIME 組込みサブルーチン	1/1,000,000	77.37 (改善版) 47.27	○	gettimeofday localtime	163.13	0.47	–
2	GETTIM サービスサブルーチン	1/1,000,000	38.37	○	gettimeofday localtime	–	–	–
3	FDATE サービスサブルーチン	1/1,000,000	18.58	○	time ctime_r	78.71	0.24	–
4	ITIME サービスサブルーチン	1/1,000,000	36.07	○	time localtime	69.27	0.52	–
5	GETTOD サービスサブルーチン	1/100,000,000	0.02	○	arm asm命令 time stamp counter gmtime_r localtime_r	–	–	分解能は、(1/cntfrq_el0)の値
6	GMTIME サービスサブルーチン	–	5.72	○	gmtime_r	58.21	0.10	–
7	LTIME サービスサブルーチン	–	10.81	○	localtime_r	67.59	0.16	–
8	omp_get_wtime ルーチン	FJOMP: 1/100,000,000 libomp: 1/1,000,000	1.00	○	FJOMP: arm asm命令 time stamp counter libomp: gettimeofday localtime	6.05	0.17	分解能は、(1/cntfrq_el0)の値
9	SECNDS サービス関数	1/1,000,000	49.05	○	gettimeofday localtime	77.31	0.63	–
10	SYSTEM_CLOCK 組込みサブルーチン	1/100,000,000	20.98 (改善版) 1.42	○	arm asm命令 time stamp counter	5.23	4.01 (改善版) 0.27	分解能は、(1/cntfrq_el0)の値
11	RTC サービス関数	1/1,000,000	5.21	○	time	–	–	–

タイマーの精度(2/2)

FUJITSU

- 主要なタイマールーチンの精度(タイマーのオーバーヘッド)

No.	ルーチン名	精度分解能	オーバーヘッド(μs)	スレッドセーフ性	実装	GCCオーバーヘッド(μs)	富士通/GCC	備考
12	TIME サービス関数	1/1,000,000	5.28	○	time	5.03	1.05	–
13	TIMEF サービス関数	1/1,000,000	7.26	○	time	–	–	–
14	TIMER サービスサブルーチン	1/1,000,000	49.85	○	gettimeofday localtime	–	–	–
15	CLOCK サービスサブルーチン	1/1,000,000	12.82	○	getrusage	–	–	–
16	CLOCKM サービスサブルーチン	1/1,000,000	13.70	○	getrusage	–	–	–
17	CLOCKV サービスサブルーチン	1/1,000,000	14.12	○	getrusage	–	–	
18	CPU_TIME 組込みサブルーチン	1/1,000,000	18.09	○	getrusage	5.39	3.36	–
19	DTIME サービス関数	1/1,000,000	14.12	○	getrusage	10.20	1.38	–
20	ETIME サービス関数	1/1,000,000	13.33	○	getrusage	9.37	1.42	–
21	SECOND サービス関数	1/1,000,000	14.27	○	getrusage	5.91	2.41	–

FUJITSU

Fortranのデータ属性と最適化の関係

- 属性一覧
- allocatable 属性
- pointer 属性
- contiguous 属性
- intent(in) 属性
- intent(out) 属性
- intent(inout) 属性
- value 属性
- pure 属性
- save 属性
- その他文法上の注意点

属性一覧（1/3）

FUJITSU

項目	属性名	属性の概要	規格	備考	詳細説明
1	dimension (array-spec)			• 配列形状は明示されているほうが最適化は進む	なし
	形状明示配列	• 上下限を整数式で指定 例： dimension(2:3) :: array	77		
	形状引継ぎ配列	• 形状を呼出し元から引継ぐ仮配列に指定 • 明示的引用仕様が必要（interface宣言など） 例： dimension(2:) :: array	90		
	形状無指定配列	• 割付け配列、ポインタ配列を指定 • 関数結果、仮引数は明示的引用仕様が必要 例： dimension(:) :: array	90		
	大きさ引継ぎ配列	• 大きさを暗黙的に引継ぐ仮配列を指定 例： dimension(2:＊) :: array	77		
	暗黙形状配列	• 定数式と同じ形状の名前付き定数配列指定 例：形状[3] の名前付き定数配列 integer, parameter, dimension(2:＊) :: array = [1,2,3]	08		
2	allocatable	• allocate文または組込み代入文(2003)で実体を割り付け • スカラのallocateも可能(2003)	90 03	• ポインタのように、実体が不明ではない。 • しかし、形状明示よりは余分な命令が出る。	**あり**
3	pointer	• ポインタであることを指定	90	• データの重なり関係が不明	**あり**
4	target	• ポインタ結合できる実体を指定	90		なし

属性一覧（2/3）

FUJITSU

項目	属性名	属性の概要	規格	備考	詳細説明
5	contiguous　ポインタ配列、形状引継ぎ配列が連続領域であることを指定			• データの連続性を保証	**あり**
	ポインタ配列 ＋contiguous	• ポインタ結合先は連続領域 プログラムで結合先が連続領域であることを保証する必要有 • 仮引数がcontiguous属性をもつポインタ配列の場合、実引数はcontiguous属性のポインタ。違反は翻訳エラー	08		**あり**
	形状引継ぎ配列 ＋contiguous	• 実引数が連続領域でなくても、コンパイラが連続領域を保証（2008規格仕様変更、ポスト京コンパイラ実装済）			**あり**
6	intent (in / inout / out)　仮引数の授受特性を指定			• -Kintentoptオプション指定時、intent属性を最適化で利用 • -Kintentoptオプションは-O1以上で有効	**あり**
	in	• ポインタ：ポインタの結合状態を変更しない。結合した値は変更可能 • 割付け：割付け状態・仮引数値を変更しない • 割付け・ポインタ以外：仮引数値を変更しない	90		
	inout	• ポインタ：ポインタの結合状態および値を変更 • ポインタ以外：結合する実引数の値を参照・変更	90		
	out	• 割付け：手続が実行する前に領域解放 • ポインタ：手続呼出し時に、ポインタの結合状態を不定にする • 上記以外：手続実行中仮引数の引用より前に値を定義する必要有	90		
7	value	• 仮引数に指定。引数が値で渡されることを指定 intent(in)との違い 副プログラムの仮引数値の変更が可能、実引数には影響を与えない • 実引数が一時領域に代入され、その一時領域が仮引数と結合する • 配列も指定可（配列一時領域を生成）	03 08	• 最適化促進効果有	**あり**

属性一覧（3/3）

FUJITSU

項目	属性名	属性の概要	規格	備考	詳細説明
8	pure	• 副作用のない手続に指定 • function文またはsubroutine文に指定 • do concurrent中にある手続引用はpure属性をもつ必要有	03 08	• 最適化促進効果有	**あり**
9	save	• return文またはend文の実行後も、割付け状態、定義状態および値を保持 • save変数はスレッドセーフではない	77		**あり**
10	intrinsic	• 組込み手続であることを指定	77		なし
11	optional	• 仮引数が手続の引用において、必ずしも実引数と結合しないことを指定	90		なし
12	parameter	• 名前付き定数であることを指定	77		なし
13	private / public	• モジュール宣言部で、モジュール内の言語要素を参照結合によって参照可能かどうかを指定	90		なし
14	protected	• 宣言したモジュール以外での言語要素の使用箇所を制限	03		なし
15	volatile	• 実体を最適化の対象としないことを指定	03	• 最適化しない	なし
16	asynchronous	• 非同期入出力で使用することを指定 • 非同期入出力文で引用中の可能性があるため、コンパイラの最適化が行われない	03	• 最適化しない	なし
17	external	• 外部手続、仮手続、手続ポインタであることを指定	77		なし
18	bind	• C言語と相互利用することを指定	03		なし
19	codimension [coarray-spec]	• 共配列（coarray）、共次元数および共上下限を指定	08		なし

allocatable 属性

```
subroutine foo
real,dimension(:,:), allocatable :: a
allocate( a(2,3) )
end ! 変数a は解放
```

文法的な注意点

- 初期状態は、未割付け
- save属性をもたない副プログラムの実体は、手続終了時に解放

```
integer,allocatable::a(:)
integer:: b(3)=1
allocate( a(2) )
print *,shape( a ) ! aの形状は [2]
a = b ! 変数aは 解放され、bの上下限で割付け、
      ! 左辺が a(:) で添字があれば、[2]の形状のまま
print *,shape( a ) ! aの形状は [3]
```

代入文での再割付（2003）

- 代入文でも割り付けられる
- 代入文の左辺がallocatable属性の変数で、右辺と形状が異なる場合、左辺の変数は右辺の形状で再割付け
  （注）Fortran 95→Fortran 2003で仕様が拡張
  各ベンダともFortran 95までの実行性能をキープするため、翻訳時オプション指定時だけ動作
  富士通の翻訳時オプションは、-Nalloc_assign

```
type x ; integer,allocatable:: a(:) ; end type
type ( x ) :: y , z
 allocate( y%a(2) , z%a(3) ) ;    ……
 y = z ! 変数y%aは解放され、z%aの上下限(1:3)
       ! で新しく割付け
```

派生型変数の代入文での再割付

- 構造体成分にallocatable属性が指定された場合、派生型変数の代入文では、左辺成分は解放され、右辺成分の上下限で再割付け
  （注）Fortran 2003からの仕様
  オプション指定に関係なく動作

pointer 属性

```
real,dimension(:,:),pointer :: a ! データポインタ
procedure(),pointer:: prc       ! 手続ポインタ
```

文法的な注意点

- ポインタはデータポインタと手続ポインタの2種類

```
real,dimension(:,:), pointer :: a,b
real,dimension(2,3), target :: t
a=> t
b=> a( :: 2, :)
```

データポインタ

- ポインタ配列は、target属性の配列またはポインタ配列と要素が重なる可能性有
- ポインタ配列は連続領域とは限らない（要素間に隙間がある可能性有）
- contiguous属性付きポインタ配列
  次頁で説明

```
interface
  function foo(d)
    intent(in):: d
  end function
end interface
procedure(foo), pointer :: p
p=> foo
print *, p( 1.0 )
```

手続ポインタ

- 手続ポインタは、procedure文で宣言

contiguous 属性（1/2）

```
subroutine foo(a)
real,dimension(:,:), contiguous :: a
real,dimension(:,:), pointer, contiguous :: p
```

文法的な注意点

- ポインタ配列または形状引継ぎ配列に指定可能
- 結合する実体が連続領域であると宣言
- 「contiguous属性付きポインタ配列」については、pointer属性を参照

```
integer,pointer:: pa(:)   ! contiguous属性なし
call foo( pa )            !翻訳エラー
contains
  subroutine foo( p )
    integer,pointer,contiguous:: p(:)
    integer,target :: t(5)=[1,2,3,4,5]
    p=> t(:: 2 )          ! 翻訳エラー
    k=2; p=> t(:: k )     ! 連続領域でない実体
    print *,p(2)          ! 値は不定
  end subroutine
```

contiguous属性付きポインタ配列

- contiguous属性をもつポインタ配列は、連続領域
- このため、そのポインタは連続領域のデータと結合する必要有

contiguous 属性（2/2）

```
integer:: a(3)
call foo( a( :: 2) )   ! 非連続領域
  ! コンパイラは以下を生成
  ! tmp = a( :: 2)  // tmpはコンパイラ生成配列（連続領域）
  ! call foo(tmp)
  ! a( ::2 ) = tmp
contains
  subroutine foo( d )
  integer, dimension(:,:),contiguous:: d
```

```
subroutine sub(pa , pac)
integer,pointer, dimension(:,:):: pa  ! 非連続の可能性有

call foo( pa )  ! paはそのまま実引数として渡せない
  ! コンパイラは以下を生成
  ! if (paが連続領域) then
  !   call foo( pa )
  ! else
  !   tmp = pa  // tmpはコンパイラ生成配列（連続領域）
  !   call foo( tmp )
  !   pa = tmp
  ! endif

contains
  subroutine foo( d )
  integer, dimension(:,:),contiguous::d  ! 形状引継ぎ配列
       :
```

contiguous属性付き形状引継ぎ配列(1)

- 形状引継ぎ配列を連続領域にすることを宣言
- 実引数が連続領域でなくても、コンパイラが連続領域を保証（左図コメント参照）

contiguous属性付き形状引継ぎ配列(2)

- 実引数が非連続領域の可能性がある場合、実引数が連続かどうかを動的に判定し、実引数が連続となることをコンパイラが保証
- 実引数がポインタ配列または形状引継ぎ配列の場合、もし配列が連続領域であれば、実引数にもcontiguous属性を指定すべき。contiguous属性がついていれば、コンパイラは連続領域かどうかの判定を生成せずに直接渡す。

intent（in）属性（1/2）

```
subroutine foo( a  )
real, intent(in):: a
a=a+1.0                ！値の定義は翻訳エラー
end
```

intent(in)

- 値の定義はできない。

```
subroutine foo( p  )
real, intent(in), pointer :: p(:,:)
p=1.0                  ！値の定義が可能
allocate( p(2,3) )     ！結合の変更は翻訳エラー
end
```

ポインタのintent(in)

- 結合先の値の定義ができる
  ポインタの指す先を変更できないだけで、
  指している先は変更可能
- ポインタ結合を変更できない

```
subroutine foo( a  )
real, intent(in), allocatable :: a(:,:)
a=1.0                  ！値の定義は翻訳エラー
allocate( a(2,3) )     ！割付けは翻訳エラー
end
```

割付け可能のintent(in)

- 割付けまたは解放はできない
  ポインタとの違いは、値の定義が可能かどうか
- 値の定義はできない。

intent（in）属性（2/2）

```
real:: a
a=1.0
call sub( a ) ! intent(in) 仮引数と結合
print *, a     ! 値は不定
 contains
   subroutine sub( d )
    real, intent(in):: d
    call def( d ) ! 引数の授受特性は暗黙的
   end subroutine
end
subroutine def( dd )
 real:: dd
 dd=2.0 ! 仮引数値を定義、intent(in)実引数を更新
end
```

intent(in)の変数の値定義は誤り

- intent(in)変数値を定義した場合、動作は未保証
- intent(in)変数を実引数に指定し、手続呼出し先で値を定義する場合、この問題が見つけ難い

デバッグオプション -Ha指定で検査可能
実行時にintent(in)変数値の定義エラーを検出

intent（out）属性

```
subroutine foo( p, a  )
real, intent(out), allocatable :: a(:,:)
real, intent(out), pointer :: p(:,:)
allocate( p(2,3), a(2,3) ) ！引用前に結合/割付け
p=1.0
a=1.0
end
```

ポインタ･割付け可能のintent(out)

- ポインタは、手続呼出し時に不定
- 割付け可能は、手続呼出し時に解放
- 引用前に結合または割付けが必要

デバッグオプション -Hx指定で検査可能
実行時にポインタ未結合のエラーを検出

```
real :: a
a=1.0
call foo( a )
print *, a   ！結果は不定
contains
 subroutine foo( d  )
 real, intent(out) :: d
 d=d+1.0   ！値を定義する前に参照できない
end subroutine
end
```

上記以外のintent(out)

- 手続呼出し時に不定
- 実引数は定義可能でなければならない
- 参照より前に、値の定義が必要

デバッグオプション -Hu指定で検査可能
実行時に未定義変数参照のエラーを検出

Intent（inout）属性

```
subroutine foo( p, a, d  )
real, intent(inout), allocatable :: a(:,:)
real, intent(inout), pointer :: p(:,:)
real, intent(inout) :: d(2,3)
  :
end
```

intent(inout)

- 仮引数は、参照しても定義してもよい
- 結合の変更、割付けまたは解放が可能

```
call foo( 1.0 )！実引数は定義不可、翻訳エラー
contains
subroutine foo1( d  )
real, intent(inout) :: d
end subroutine
end
```

intent(inout)に対する実引数

- 実引数は、定義可能でなければならない。
- intent属性の省略時、実行時に定義しなければ、対応する実引数は定義可能でなくてもよい
  ※左図のintent(inout)が無ければ翻訳エラーは出ない

value 属性

```
subroutine sub(k) bind(c)  ！Cﾌﾟﾛｸﾞﾗﾑから
integer,value :: k              ！呼び出される
k=k+1 ！値の代入が可能、実引数には影響なし
print *,'k=',k
end
interface
  subroutine foo(k) bind(c)  ！Cﾌﾟﾛｸﾞﾗﾑの
    integer,value :: k          ！手続を明示
  end subroutine
end interface
integer::n
n=1
call foo( n )  ！Cﾌﾟﾛｸﾞﾗﾑの呼出し
end
```

Cプログラム

```
void sub(int);
int foo(int k) {
 sub(k) ;  /* Fortran ﾌﾟﾛｸﾞﾗﾑの呼出し */
 return 0 ;
}
```

文法的な注意点

- C言語との結合で引数を値渡しで相互利用する際に必須
- 仮引数の値および定義状態の変更は実引数に影響しない
- 実引数が一時領域に代入され、その一時領域が仮引数と結合
- value属性仮引数をもつ手続は、明示的引用仕様（**interface block**）が必要
- 次の場合を除き、スタックまたはレジスタで引数を渡す
  - ✓ 配列 (C言語との結合では指定できない) 2008仕様
  - ✓ 手続言語束縛指定子（bind（c））のない手続の文字型
  - ✓ 成分にポインタまたは割付けをもつ派生型
  - ✓ 上記以外の派生型（ABIに従う）
  - ✓ optional属性

pure 属性

```
module mod
 contains
   pure function foo( d )
     real , intent(in):: d
     foo = d + 1.0
   end function
end
subroutine test(a,b)
use mod
real,dimension(1000) :: a,b
do concurrent(i=1:1000)
   a(i) = a(i) + foo( b(i) )   ! do concurrent内の
enddo                          ! 手続はpure
end subroutine test
```

文法的な注意点

- function文またはsubroutine文にpure（またはelemental）を指定した手続は、副作用がないことを宣言
- 純粋手続の制約
  - ✓ 関数の仮引数は、pointer 属性をもつ引数を除いて、intent(in)またはvalue属性(2008)が必要
  - ✓ サブルーチンの仮引数は、pointer属性をもつ引数を除いて、intent属性が必要
  - ✓ 局所変数は明示的初期値指定およびsave属性を指定できない
  - ✓ 共通ブロック内の変数、親子結合または参照結合によって参照可能な変数などを定義してはならない
  - ✓ 純粋手続中で引用する手続は、すべて純粋手続でなければならない
  - など

最適化との関係

pure指定により、自動並列化などの最適化が促進される。

save 属性

```
subroutine sub_save()
  real(8),save :: a
  a = a + 1.0
end subroutine sub_save
```

save属性をもつ変数 a は再エントリ時に、前回エントリの復帰時の値を保証

文法的な注意点

- return,endの後もデータが保存される
  ⇒再エントリ時に変数の値を保証するため
- save変数は、スレッドセーフでない。
- 京ではメインフレーム由来の仕様でsaveがデフォルトだった。

save属性と最適化との関係

無闇にsave属性指定すると、死変数となるべき変数が削除できなくなり、余分なstoreが残る可能性がある。

-Kautoをデフォルト化

```
subroutine sub_flocal()
  real(8) :: a
  a = a + 1.0
end subroutine sub_flocal
```

-Kauto default化の影響

- ✓ 変数 a の格納先は、レジスタまたはスタックを使用
- ✓ save属性無しで宣言された変数 a は再エントリ時、値は未定義
- ✓ 変数 a の定義は未参照となり、最適化は『a = a + 1.0』を削除

その他文法上の注意点

FUJITSU

- コンパイラが内部的に一時配列を生成するケース
  - 文法上、コンパイラが一時配列を生成する３つの例（後述）
    - 連続領域をもたない実引数配列が、形状明示配列またはcontiguous属性をもつ形状引継ぎ配列と結合
    - 実引数が配列、value属性の仮配列
    - 左辺と右辺の配列要素に重なりがある配列代入文
  - コンパイラ生成の一時配列の問題点
    - 一時配列を生成するための allocate / deallocate のオーバーヘッド
      - ✓ 一時配列をスタック上に生成する動作（–Kauto、-Ktemparraystack）をデフォルト化するため、オーバーヘッドの心配はなくなる。（スタックサイズには注意が必要）
    - 一時配列へのコピーイン / コピーバックのオーバーヘッド
      - ✓ 可能な場合にはコンパイラが最適化を実施するが、最適化的できない場合にはオーバーヘッドとなる。

プログラムにない配列を生成するケース （1/3）

- **連続領域をもたない実引数配列が、形状明示配列またはcontiguous属性をもつ形状引継ぎ配列と結合**

```
subroutine sub(a)
real(8),dimension(1:1000) :: a
interface
  subroutine foo(a,m)
  real(8),dimension(1:m) :: a
  end subroutine
end interface
call foo(a(1:1000:2),500)
end subroutine
```

非連続領域を形状明示配列で受け取る場合のコンパイラ出力イメージ

```
tmp(1:500) = a(1:1000:2)
call foo(tmp,500)
a(1:1000:2) = tmp
```

プログラムにない配列を生成するケース （2/3）

FUJITSU

- **実引数が配列、VALUE属性の仮配列(Fortran 2008)**

```
subroutine sub(x)
  integer(4),dimension(1:80) :: x
  interface
    subroutine foo(a)
      integer(4),dimension(:),value :: a
    end subroutine foo
  end interface
  call foo(x)
end subroutine sub
```

実引数が配列、value属性の仮配列の場合のコンパイラ出力イメージ

```
tmp(1:80) = x(1:80)
call foo(tmp)
```

文法：『仮引数の値および定義状態の変更は実引数に影響しない』
つまり、foo側でaを定義した結果がxに反映されてはいけない。このため、
『value属性＝参照だけ』と考え、xをそのまま渡すことはできない。

プログラムにない配列を生成するケース （3/3）

FUJITSU

- **左辺と右辺の配列要素に重なりがある可能性のある配列代入文**

配列記述は右辺評価後に左辺代入をする文法

```
! 右辺評価後に左辺代入しないと答え
! を誤るケース
subroutine sub(a,b,n,m)
real(8),dimension(1:n) :: a,b
a(2:m+1) = a(1:m) + b(1:m)
end subroutine
```

```
! コンパイラ生成配列（tmp）を利用して代入
! を実現。引数がpointerの場合なども同じ。
    tmp(1:m) = a(1:m) + b(1:m)
    a(2:m+1) = tmp(1:m)
```

右左辺ループを融合しても問題ない場合はコンパイラが最適化

```
subroutine sub(a,b,n,m)
real(8),dimension(1:n) :: a,b
a(1:m) = a(2:m+1) + b(1:m)
end subroutine
```

```
! 上記の場合もループを反転すれば最適化
! が可能　（コンパイラに実装済）
! コンパイラ生成配列は不要となる
    a(m+1:2:-1) = a(m:1:-1) + b(m:1:-1)
```

FUJITSU

ポインタと引数I/Fの改善による性能チューニング例

- pointerとallocatableの違い
- pointerの性能チューニング
- 引数I/F改善による性能チューニング

pointerとallocatableの違い

```
subroutine test(a,b,n1,n2)
real(kind=8),dimension(:,:), pointer :: a,b
do j=1,n2
   do i=1,n1
      a(i,j) = a(i,j) + b(i,j)
   enddo
enddo
end subroutine test
```

```
subroutine test(a,b,n1,n2)
real(kind=8),dimension(:,:), allocatable :: a,b
do j=1,n2
   do i=1,n1
      a(i,j) = a(i,j) + b(i,j)
   enddo
enddo
end subroutine test
```

修正差分による文法解釈の相違と最適化効果		pointer	allocatable
定義配列と参照配列のデータ依存		不明	無
繰り返しを跨るデータ依存		不明	無
最内ループ1次元目のアクセス連続性		不明	有
多重ループ全体のアクセス連続性		不明	不明
修正による最適化効果の可能性	自動並列	×	○
	ＳＩＭＤ	×	○
	（効果のある） ＳＷＰ	×	○

※ allocatableは形状明示と比較するとアドレス計算は多い

pointerの性能チューニング (1/3)

FUJITSU

- contiguousの効果

```
subroutine test(a,b,n1,n2)
real(kind=8),dimension(:,:), pointer :: a,b
do j=1,n2
   do i=1,n1
      a(i,j) = a(i,j) + b(i,j)
   enddo
enddo
end subroutine test
```

```
subroutine test(a,b,n1,n2)
real(kind=8),dimension(:,:), pointer,contiguous :: a,b
do j=1,n2
   do i=1,n1
      a(i,j) = a(i,j) + b(i,j)
   enddo
enddo
end subroutine test
```

修正差分による文法解釈の相違と最適化効果		pointer	左記+contiguous
定義配列と参照配列のデータ依存		不明	不明
繰り返しを跨るデータ依存		不明	不明
最内ループ1次元目のアクセス連続性		不明	有
多重ループ全体のアクセス連続性		不明	不明
修正による最適化効果の可能性	自動並列	×	×
	ＳＩＭＤ	×	×
	（効果のある） ＳＷＰ	×	×

※ contiguous指定で最内ループ中のアドレス計算群は改善される

pointerの性能チューニング (2/3)

FUJITSU

- norecurrence最適化指示子によるデータ依存の解消

```
subroutine test(a,b,n1,n2)
real(kind=8),dimension(:,:),
pointer,contiguous :: a,b
do j=1,n2
   do i=1,n1
      a(i,j) = a(i,j) + b(i,j)
   enddo
enddo
end subroutine test
```

```
subroutine test(a,b,n1,n2)
real(kind=8),dimension(:,:),
pointer,contiguous :: a,b
!ocl norecurrence
do j=1,n2
   do i=1,n1
      a(i,j) = a(i,j) + b(i,j)
   enddo
enddo
end subroutine test
```

修正差分による文法解釈の相違と最適化効果		pointer+contiguos	左記+norecurrence
定義配列と参照配列のデータ依存		不明	無
繰り返しを跨るデータ依存		不明	無
最内ループ1次元目のアクセス連続性		有	有
多重ループ全体のアクセス連続性		不明	不明
修正による最適化効果の可能性	自動並列	×	○
	ＳＩＭＤ	×	○
	（効果のある）ＳＷＰ	×	○

pointerの性能チューニング (3/3)

FUJITSU

- do concurrentによるデータ依存の解消

```
subroutine test(a,b,n1,n2)
real(kind=8),dimension(:,:),
pointer,contiguous :: a,b
do j=1,n2
   do i=1,n1
      a(i,j) = a(i,j) + b(i,j)
   enddo
enddo
end subroutine test
```

```
subroutine test(a,b,n1,n2)
real(kind=8),dimension(:,:),
pointer,contiguous :: a,b
do concurrent(j=1:n2)
   do concurrent(i=1:n1)
      a(i,j) = a(i,j) + b(i,j)
   enddo
enddo
end subroutine test
```

修正差分による文法解釈の相違と最適化効果		pointer+contiguos	左記+do concurrent
定義配列と参照配列のデータ依存		不明	無
繰り返しを跨るデータ依存		不明	無
最内ループ1次元目のアクセス連続性		有	有
多重ループ全体のアクセス連続性		不明	不明
修正による最適化効果の可能性	自動並列	×	○
	ＳＩＭＤ	×	○
	（効果のある）ＳＷＰ	×	○

引数I/F改善による性能チューニング

FUJITSU

- intent属性活用による最適化の促進

```
subroutine sub()
real(kind=8) :: a,b
a = 1.0
b = 1.0
call foo(a,b)
print *,a,b
end subroutine sub
```

富士通コンパイラは-Kintentopt有効時
（default）、intent属性を最適化で利用
（注）誤ったintent属性を利用するとＮＧとなる

```
subroutine sub()
interface
    subroutine foo(a,b)
        real(kind=8),intent(inout) :: a
        real(kind=8),intent(in) :: b
    end subroutine foo
end interface
real(kind=8) :: a,b
a = 1.0
b = 1.0
call foo(a,b)  ! bは更新されない
print *,a,b    ! bは1.0に置換可能
end subroutine sub
```

修正差分による文法解釈の相違と最適化効果	intent無	intent有
fooでのaの更新	有	有
fooでのbの更新	有	無
aの定数伝搬最適化	－	－
ｂの定数伝搬最適化	×	○

更新履歴

更新履歴

版数	発行月	変更理由及び内容
初版(1.0版)	2020/8	新規作成
1.3版	2021/3	ソフトウェア版数アップによる差分修正、および、記事見直しによる誤字や表現の修正
1.4版	2022/5	・スライドデザインの変更 ・-Karray_declaration_optのdefault値変更 ・-Oオプションの機能概要の表記見直し ・スタック不足時の対応方法にスレッド並列時の方法を追加
1.5版	2022/7	・記事見直しによる誤字や表現の修正
1.6版	2023/3	・記事見直しによる誤字や表現の修正

FUJITSU