「富岳」セミナー
中級編
単体性能の最適化手法2
講習会資料

登録施設利用促進機関 / 文科省委託事業「HPCIの運営」代表機関
一般財団法人 高度情報科学技術研究機構（RIST）【著】

国立研究開発法人理化学研究所 計算科学研究センター（R-CCS）

本セミナー中級編の目的

- 「富岳」でプログラムを効率的に走らせるためには、チューニングによりプログラムの高速化を目指す必要があります。そのためには、性能情報を採取し、自分のプログラムがどのような特性を持っているかを知ることが重要となります。
- 本中級編は、チューニング未経験の人を対象に、性能向上策実施の補助となるプログラム性能情報の採取方法や最適化メッセージの見方、重要な最適化手法の概略、および最適化手法の適用例等の基本事項を説明します。

中級編の開催予定

本中級編は今回開催も含めて三部構成です。

- 第一部（適時開催中）：プロファイラと単体性能の最適化手法1

  第一部ではCPU単体性能最適化の中で特に重要な最適化手法であるSIMD化、ソフトウェアパイプライニング、プリフェッチについて、概念説明やプロファイラを用いて最適化の効果を確認する方法を説明しています。

- 第二部（適時開催中）：MPI・LLIO

  第二部では並列性能最適化に向けて、通信やI/Oの最適化の際に考慮すべき事項を取り上げます。効率的な通信やI/Oのためのノードやプロセスの配置に関する各種事項、「富岳」のストレージ構成、I/O性能を最適化・高速化する技術であるLLIOの機能概略、MPI通信性能情報（MPI統計情報）の採取方法、及び「富岳」のMPI通信に固有な事項を説明しています。

- 第三部（今回開催）：単体性能の最適化手法2

  第一部で取り上げなかったCPU単体性能のFortranでの各種最適化手法を取り上げます。

第三部（今回開催）の内容

- 「富岳」の概略

  CPUとメモリ等、Tofuの概略、計算ラックの構成概略を説明します。

- プロファイラ（CPU性能解析レポート）

  中級編第三部では、後の章で説明する各種最適化手法の性能情報をCPU性能解析レポートを用いて採取します。CPU性能解析レポートは本中級編第一部で説明しています。本章では第一部からの抜粋として後の章を理解する上で必要となる事項を説明します。

- 各種最適化手法

  第一部で取り上げなかったCPU単体性能のFortranでの各種最適化手法を、以下の観点で分類し、説明します：演算の効率化、ソフトウェアパイプライニングの促進、SIMD化の促進、およびキャッシュチューニングのための基本事項。

内容

- 「富岳」の概略
- プロファイラ（CPU性能解析レポート）
- 各種最適化手法
- 【付録】

内容

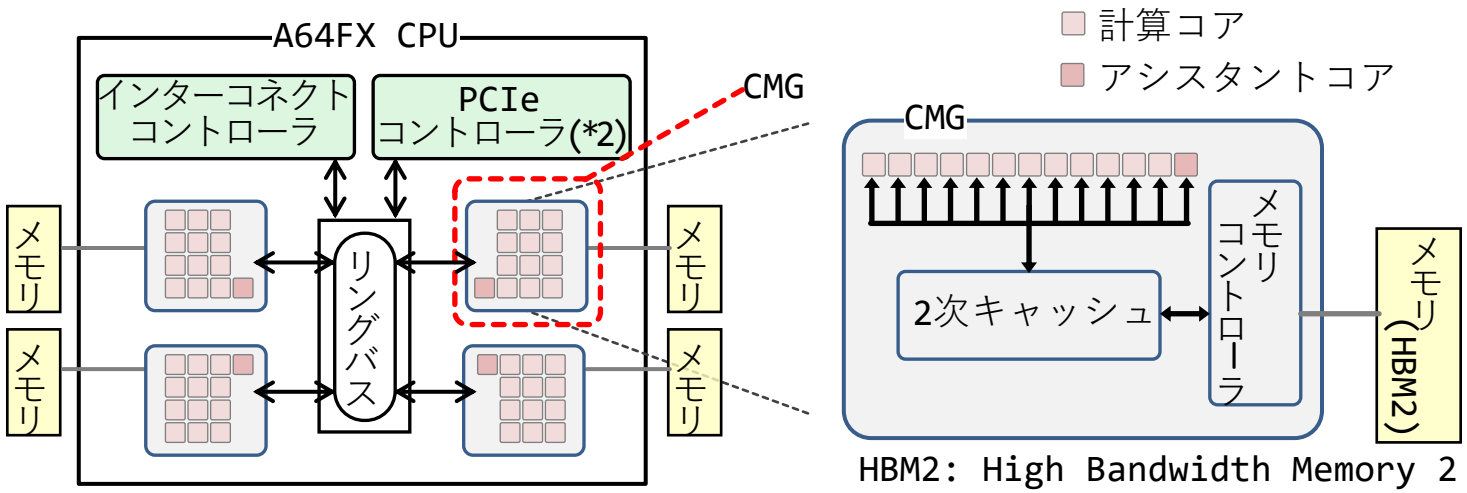
- 「富岳」の概略
  - CPUとメモリ等
  - Tofuの概略
  - 計算ラックの構成
- プロファイラ（CPU性能解析レポート）
- 各種最適化手法
- 【付録】

■ CPU（ノード）の構成

CPU内の構成を下図に示す。

- ■ CPUチップ：A64FX
- ■ CPU（ノード）の総数：158,976
- ■ CPU当たりのコア数：
  - □ 計算ノード：計算コア 48 個 + アシスタントコア 2 個(*1)
  - □ 計算兼I/Oノード：計算コア 48 個 + アシスタントコア 4 個(*1)

(*1) 12個の計算コアと1個（または0個）のアシスタントコア、2次キャッシュ、およびメモリコントローラで、1個のCMG（Core Memory Group）を構成する。一つのCPU内には四つのCMGが含まれる。



(*2) PCIeコントローラは計算兼I/Oノードのみに存在する。

■ CPUの演算性能等

■ 動作周波数：

□ 計算ノード：通常モード 2.0 GHz①、ブーストモード 2.2 GHz

□ 計算兼I/Oノード：常に2.2 GHz

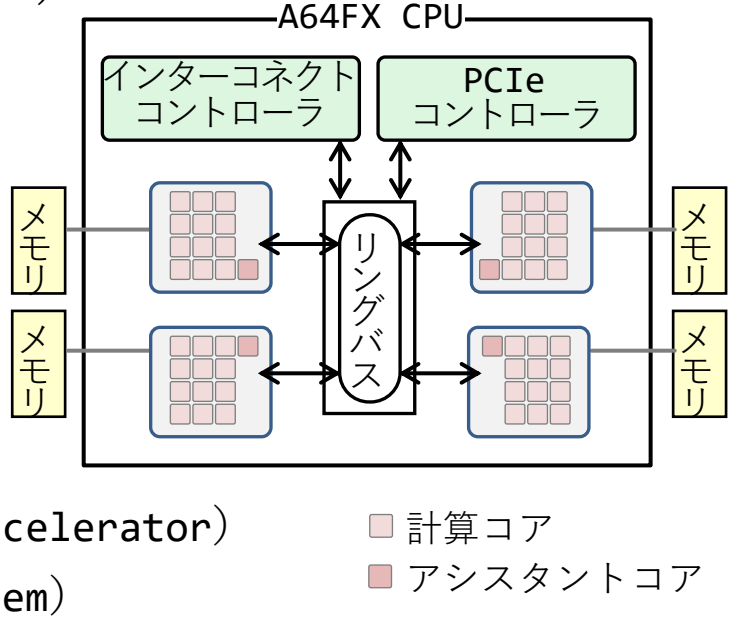
■ 浮動小数点演算器：512-bit SIMD② FMA③ × 2個④

■ ピーク演算性能：コア当たり、倍精度で 64 GFLOPS(*3)、
ノード当たり、倍精度で 3072 GFLOPS

(*3) 1秒間にコア当たり、① 2 G サイクル× ② 倍精度 8 要素 × ③ 2 積和演算 × ④ 2 個 ＝ 64 G 回の浮動小数点演算、即ち、64 GFLOPS

■ 従って、「富岳」のCPU性能を発揮させるためには、SIMDとFMAの活用が重要である。

- レジスタ
  - 浮動小数点レジスタ：コア当たり 32 個
- データキャッシュ
  - 1次データキャッシュ：コア当たり 4 ウェイ（4 ウェイの合計 64 KiB）
  - 2次データキャッシュ：CMG当たり 16 ウェイ（16 ウェイの合計 8 MiB）
- メモリ
  - 容量：CPU当たり 32 GiB（CMG当たり 8 GiB）
  - 転送速度：CPU当たり 1024 GB/s
- 計算ノード間ネットワーク
  - Tofu Interconnect D
    (28 Gbps x 2 lane x 10 port)
- ストレージ
  - 階層化ストレージ：
    - 第1階層：LLIO（Lightweight Layered IO-Accelerator）
    - 第2階層：FEFS（Fujitsu Exabyte File System）
    - 第3階層：商用クラウドストレージ（要別途契約）、HPCIストレージ



内容

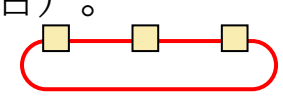
- 「富岳」の概略
  - CPUとメモリ等
  - Tofuの概略
  - 計算ラックの構成
- プロファイラ（CPU性能解析レポート）
- 各種最適化手法
- 【付録】

- Tofuインターコネクト、Tofu単位
  - 大規模並列計算では多数のノードを利用してMPIプログラムを実行し、一般に多数のノード間で大量のデータのやり取り（通信）が必要となる。そのため、大規模並列計算では一般に通信時間が大きくなるので通信性能が非常に大事である。
  - 通信時間は通信を行うノード間に介在するネットワークの本数（ホップ数）に依存し、ホップ数が少ないほど通信時間は短くなる。
  - 通信性能を向上させるためにはホップ数を最小化することが重要であり、それを実現するために、「富岳」ではTofuインターコネクトという技術を導入している。
  - Tofuインターコネクトの、大きさ2×3×2のa、b、c軸（次スライドで説明）で構成される単位をTofu単位という。

※ Tofuインターコネクト、及び「富岳」に固有なMPIの事項は「第二部：MPI・LLIO」で説明していますので、第三部（本講義）ではTofuの概略のみを説明することとします。

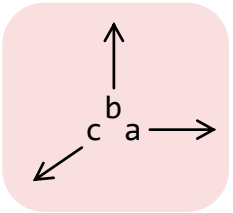
■ Tofu単位の構成と物理座標

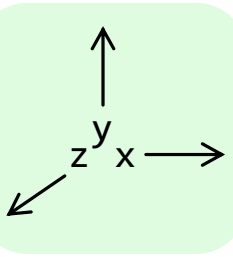
- Tofuインターコネクトの、大きさ2×3×2のa、b、c軸で構成される単位をTofu単位という。
  - 一つのTofu単位には、ノード（図の □ の部分）が12個含まれる。
  - 各Tofu単位内の：
    - 各ノードの座標はa、b、c軸で表される。
    - 各ノード間の結合は
      - b軸方向はトーラス結合
        （ノードがリング状に結合）。



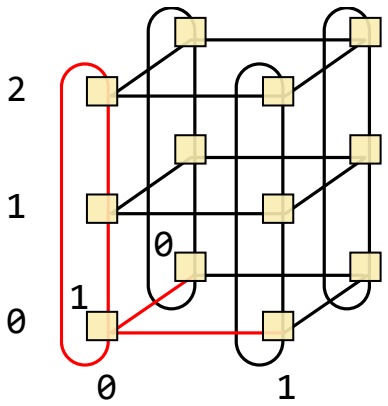
      - a軸、c軸方向はメッシュ結合
        （ノードが3個以上の場合、両端が結合していない）。

  - 各Tofu単位の座標はx、y、z軸で表される。





Tofu単位



- x、y、z軸について隣接するTofu単位同士では、a、b、c座標を同じくするノード同士が接続されている。
- 以後、x、y、z軸およびa、b、c軸を物理座標と呼ぶ。

■ 論理座標

- MPIジョブの実行時に、ユーザはジョブスクリプトにおいて、（例えば「#PJM -L "node=2x6x4"」のように）使用したいノード数を、X、Y、Z軸の各方向の個数で指定できる。
- 使用したいノード数の指定におけるX、Y、Z軸を論理座標と呼ぶ。

※ 色々な指定例は「第二部：MPI・LLIO」で紹介しています。

■ 論理座標と物理座標の対応付け

- 論理座標のノードは、実際にはTofuの物理座標上のノードに対応付けられる。
- この対応付けはシステムが自動的に行うので、ユーザは関知する必要はない。

内容

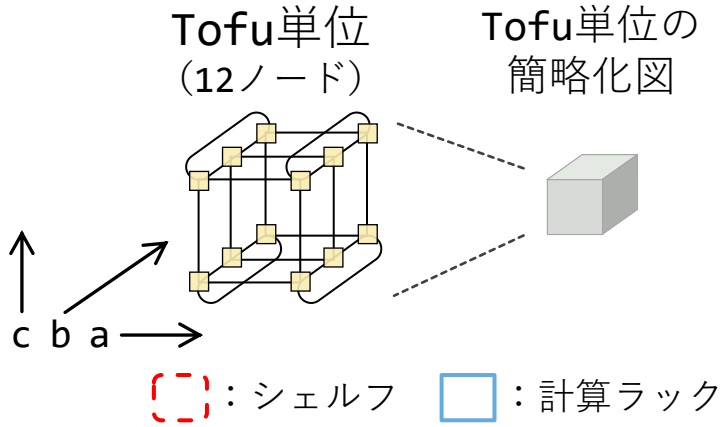
- 「富岳」の概略
  - CPUとメモリ等
  - Tofuの概略
  - 計算ラックの構成
- プロファイラ（CPU性能解析レポート）
- 各種最適化手法
- 【付録】

- チューニングにおけるノード形状やI/Oの重要性
  - 実行時のノード数やノード形状を決めるにあたっては、多数の計算ノードを収めた計算ラックの構成を踏まえて性能に悪影響が出にくい値にするのが良い。
  - 計算ラック内には計算に加えてI/Oを担当するノードというものがあり、I/O性能の改善が求められる場合には計算ラック内でのI/Oノードの位置を意識してノード形状を決める必要がある(*1)。
  - I/Oノードのうち、ストレージI/OノードはI/O性能向上を目的とするLLIOの構成要素である(*2)。
  - I/Oを分散する場合にも計算ラックの構成を踏まえて検討すると良い。

(*1) (*2) ストレージI/Oノード等の位置、LLIOの初歩的な使用例等は「第二部：MPI・LLIO」で説明していますので、第三部（本講義）では計算ラック内でのTofu単位の構成の概略のみを紹介することとします。

■ 計算ラックの構成：シェルフ

■ 一つのシェルフにTofu単位が4個含まれる。
- 1個のシェルフに48ノード入る。

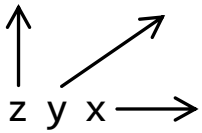


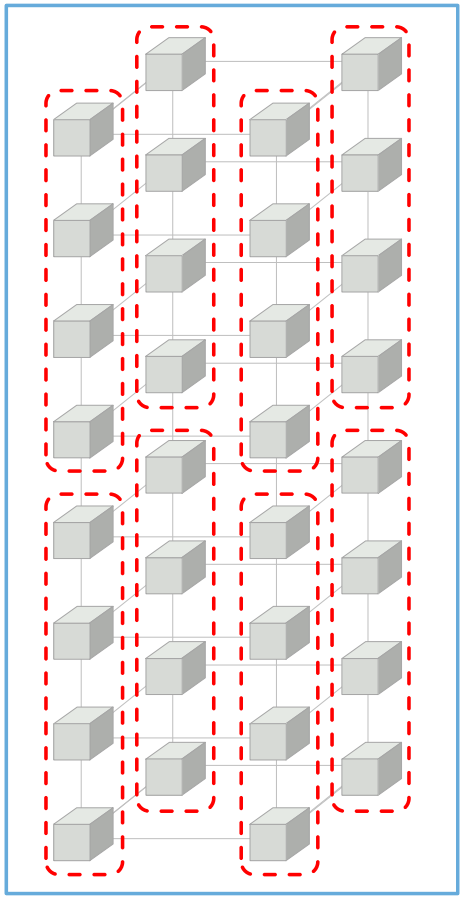
■ 8個のシェルフが一つの計算ラックに収められている。
- 一つの計算ラックに384ノード入る(*1)。

(*1) ただし、36個の計算ラックには192ノード入る。

■ 384ノードのラック数：396、
192ノードのラック数：36、
よって、ノードの総数は、158,976となる。

■ 384ノード以下を使用する場合はジョブスクリプトにおいて「#PJM -L "rscgrp=small"」を、385ノード以上を使用する場合は「#PJM -L "rscgrp=large"」を指定する。
385ノード以上のジョブはシェルフ単位で
ノードが割り当てられる。





内容

- 「富岳」の概略
- プロファイラ（CPU性能解析レポート）
- 各種最適化手法
- 【付録】

■ 3種類のプロファイラ

「富岳」では、プロダクトの高速化を目的として、

(1) プログラム内で計算時間のかかっている部分の調査

(2) 並列ジョブのプロセス間やスレッド間のロードバランスが均等かどうかの調査

(3) 各種性能特性（GFLOPS値、ピーク性能比、メモリスループットなど）の調査

(4) CPUに関するより詳細な情報（実行時間内訳、ビジー率、キャッシュミスなど）の調査

を行うため、下記のプロファイラが提供されている。

■ 基本プロファイラ

サンプリング解析でプログラム全体または一部の統計情報の計測および出力を行う。

■ 詳細プロファイラ（テキスト形式）

ハードウェアカウンタによりプログラムの指定した区間の実行性能情報の計測および出力を行う。

■ CPU性能解析レポート

ハードウェアカウンタで複数の実行により計測した多数のCPU性能解析情報を集約し、表と付随するグラフを用いて可視化する。

各プロファイラは、Fortran/C/C++の、逐次、スレッド並列、MPI並列、スレッド並列＋MPI並列のいずれのプログラムに対しても適用可能である。

■ 各プロファイラの使い分けの指針

- 手始めとして下記(1)を調査したい場合は、基本プロファイラを用いてプログラムの広い範囲にわたって計測するのが良い。
- 特定のループに注目している状態で、下記(2)や(3)を調査したい場合には詳細プロファイラを用い、より詳しく(4)を調査したい場合にはCPU性能解析レポートを用いるのが良い。

再掲
(1) プログラム内で計算時間のかかっている部分の調査
(2) 並列ジョブのプロセス間やスレッド間のロードバランスが均等かどうかの調査
(3) 各種性能特性（GFLOPS値、ピーク性能比、メモリスループットなど）の調査
(4) CPUに関するより詳細な情報（実行時間内訳、ビジー率、キャッシュミスなど）の調査

各プロファイラの使用方法は中級編第一部で詳しく説明しています。第三部（本講義）ではCPU性能解析レポートの概要を抜粋して説明します。
第三部（本講義）では、後の章で説明する各種最適化手法の性能情報をCPU性能解析レポートを用いて採取します。本章ではCPU性能解析レポートについて第一部からの抜粋として後の章を理解する上で必要となる事項を説明します。

内容

- 「富岳」の概略
- プロファイラ（CPU性能解析レポート）
  - CPU性能解析レポートの概略
    - Cycle Accountingについて補足
- 各種最適化手法
- 【付録】

■ CPU性能解析レポートの種別

CPU性能解析レポートは、表示する情報の種類や、粒度によって、以下の四つの段階が用意されている。

レポート種別	プロファイラによる必要な計測回数	説明
単体レポート	1回	作成に必要な計測回数がもっとも少ない。実行時間、演算性能、メモリースループット、命令数についてのおおまかな情報を出力する。
簡易レポート	5回	手軽にCPU性能解析レポートを使用したい場合に推奨される。
標準レポート	11回	標準的なCPU性能解析レポート。通常の使用で推奨される。
詳細レポート	17回	もっとも詳細なCPU性能解析レポート。

後の章「各種最適化手法」では詳細レポートを使用します。

■ CPU性能解析レポートで参照可能な情報の一覧

○：すべての情報が出力される。
△：一部の情報が出力される。
－：情報が出力されない。

表タイトル	表の概要	レポート種別			
		単体	簡易	標準	詳細
Information	計測環境の情報およびユーザーの指定内容を表示します。	○	○	○	○
Statistics	メモリースループット、命令数、演算数などCPUの動作状況に関係する情報を表示します。	△	○	○	○
Cycle Accounting	プログラムの実行時間の内訳を表示します。	－	△	○	○
Busy	プログラムのメモリー・キャッシュおよび演算パイプラインのビジー率に関する情報を表示します。	－	△	△	○
Cache	キャッシュミスに関する情報を表示します。	－	△	○	○
Instruction	命令ミックスに関する情報を表示します。	－	△	△	○
FLOPS	浮動小数点演算に関する情報を表示します。	－	△	○	○
Extra	ギャザー命令の内訳、および命令ミックスに含まれない命令に関する情報を表示します。	－	－	－	○
Hardware Prefetch Rate (%) (/Hardware Prefetch)	ハードウェアプリフェッチの内訳を表示します。	－	－	－	○
Data Transfer CMGs	ユーザーが指定したCMGに対する、全CMG、メモリー、Tofu、およびPCI間のスループット情報を表示します。	－	－	－	○
Power Consumption (W)	コア、L2キャッシュ、およびメモリーの消費電力を表示します。	－	－	○	○

■ CPU性能解析レポートの使用手順概略

■ ソースプログラム

↓

■ ソースプログラム

計測区間指定ルーチン追加

Fortran：call fapp_start(引数)、call fapp_stop(引数)

C/C++：#include "fj_tool/fapp.h"、
fapp_start(引数);、fapp_stop(引数);

↓

■ 実行プログラム

コンパイル・リンク

ログインノードで
(mpi)frtpx、(mpi)fccpx、(mpi)FCCpx コマンド

↓

■ プロファイルデータ

プロファイルデータの計測

計算ノードで `fapp -C` コマンド(*1)

(*1) fappコマンドのオプションを変えて複数回の計測が必要。

↓

■ プロファイル結果
（csv形式）

プロファイル結果の出力

ログインノードで `fapppx -A -tcsv` コマンド(*2)

(*2) 上記(*1)に対応した複数回の実行が必要。

↓

CPU性能解析レポートの作成

Excelによりcpu_pa_report.xlsmを起動

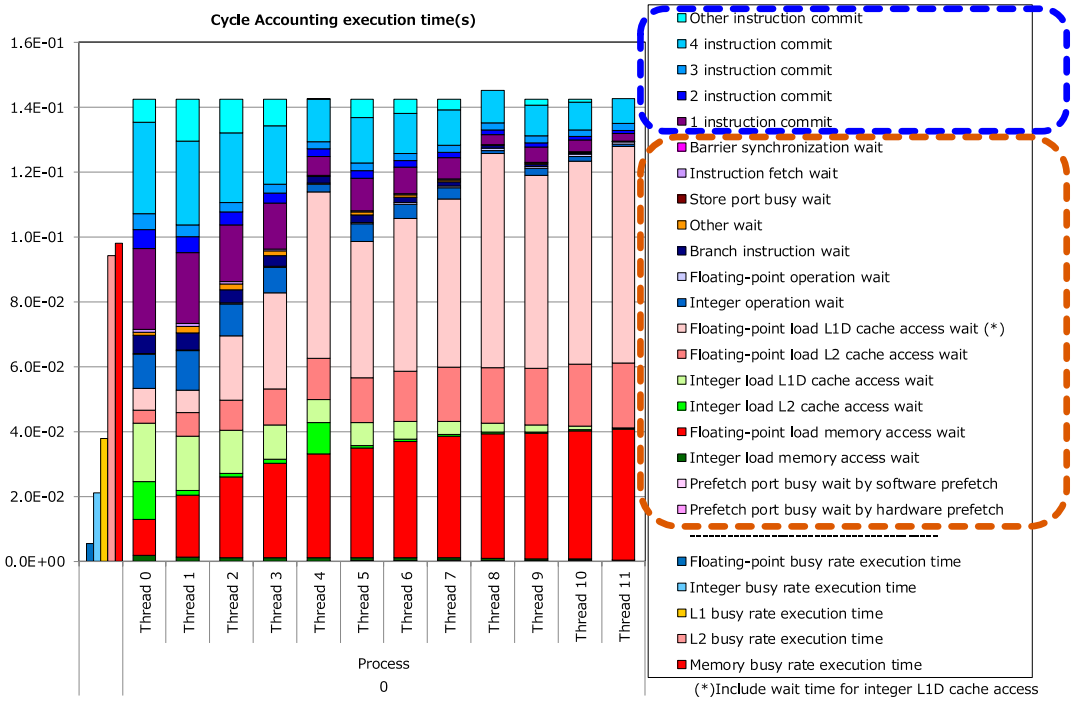
■ CPU性能解析レポート
（Excel形式）

内容

- 「富岳」の概略
- プロファイラ（CPU性能解析レポート）
  - CPU性能解析レポートの概略
  - Cycle Accountingについて補足
- 各種最適化手法
- 【付録】

■ Cycle Accountingについて補足

- Cycle Accountingでは、計測対象区間の実行に要した総時間（CPUサイクル数）をCPUの動作状態で分類してグラフ化する。そのグラフからCPU内のボトルネックが把握できるので、詳細な性能分析やチューニングを行うことができる。
- グラフの主に青色系部分：1サイクルでN命令コミット（N命令実行）した時間
  - 演算が実行されている時間。
- グラフの主に赤色系部分：各種要因で命令がストールした時間（待ち時間）
  - 演算待ち、L1DおよびL2キャッシュアクセス待ち、メモリアクセス待ち等の時間。



■ Cycle Accountingに基づくチューニングの指針

実行時間内訳	内訳の各要素が多い場合に考えられる原因	チューニングの指針	主な最適化手法
命令コミット	演算の命令数が多い。	命令数を削減する。	指示行追加によるSIMD化の促進、共通式の除去、不変式の移動
演算待ち	命令スケジューリングが悪い。	ソフトウェアパイプライニングを促進する。	指示行追加によるソフトウェアパイプライニングの促進、ループ分割
L1Dキャッシュアクセス待ち	レジスタのスピル・フィル（レジスタ溢れ）が多発している。	使用するレジスタ数を削減する。	ループ分割
L2キャッシュアクセス待ち	L1Dキャッシュミスが多発している。	L1Dキャッシュ利用の効率化を図る。	ループ交換、ループ融合、セクタキャッシュ、配列マージ、ループ分割
		L1Dキャッシュのレイテンシを隠蔽する。	L1Dプリフェッチ
メモリアクセス待ち	L2キャッシュミスが多発している。	L2キャッシュ利用の効率化を図る。	ループ交換、ループブロッキング、ストリップマイニング、セクタキャッシュ、ループ分割
		L2キャッシュのレイテンシを隠蔽する。	L2プリフェッチ
		データアクセス量を削減する。	高速ストア（ZFILL）

- 中級編第一部ではSIMD化、ソフトウェアパイプライニング、プリフェッチについて概念やオプションの使用方法、ループ例等を説明しています。
- 第三部（本講義）では第一部で取り上げていない各種最適化手法を後の章「各種最適化手法」において説明します。

内容

- 「富岳」の概略
- プロファイラ（CPU性能解析レポート）
- 各種最適化手法
- 【付録】

■ 各種最適化手法

本章の内容

中級編第一部で取り上げなかったCPU単体性能のFortranでの各種最適化手法を、以下の観点で分類し、説明します。

- 演算の効率化
- ソフトウェアパイプライニングの促進
- SIMD化の促進（ロード命令・ストア命令の効率化、演算の効率化）
- キャッシュチューニングのための基本事項

※ 本章ではコンパイラによる最適化をソースプログラム中で文字!OCLで始まる最適化指示行(*1)を指定して行う場合があります。最適化指示行の指定を有効にするにはコンパイル時にオプション `-Kocl` を追加指定します。

(*1) 「Fortran使用手引書」の用語では「最適化制御行」（OCL: Optimization Control Line）ですが、本講義では「最適化指示行」または単に「指示行」と呼ぶことにします。

※ 本章ではコンパイラによる最適化の実施状況をコンパイルリストを用いて説明・考察します。コンパイルリストを出力するためには、本講義ではコンパイル時にオプション

```
-Koptmsg=guide -Nlst=t
```

を追加指定します。コンパイルリストを出力するためのオプションおよびコンパイルリストの見方等は中級編第一部で説明していますので、中級編第一部の講義資料を参照して下さい。

内容

- 「富岳」の概略
- プロファイラ（CPU性能解析レポート）
- 各種最適化手法
  - 演算の効率化
    - ソフトウェアパイプライニングの促進
    - SIMD化の促進（ロード命令・ストア命令の効率化、演算の効率化）
    - キャッシュチューニングのための基本事項
- 【付録】

- 各種最適化手法
  - 演算の効率化

本節の内容

本節では、演算を効率的に行うための最適化手法のうち、以下の最適化手法を説明します。

- 積和演算命令の使用
- 逆数近似命令の使用
- マルチ演算関数への変換
- 組込み関数のインライン展開
- 演算評価方法の変更

- 各種最適化手法
  - 演算の効率化

本節の内容

- 積和演算命令の使用
- 逆数近似命令の使用
- マルチ演算関数への変換
- 組込み関数のインライン展開
- 演算評価方法の変更

■ 積和演算とは

a、b、およびcを実数とする。式 a+b*c の形の演算を積和演算という。「富岳」には、一つの積和演算において乗算と加算を一つの命令で実行することができる積和演算器がコア当たり二つ搭載されている。積和演算において乗算と加算を一つの命令で実行することにより、乗算命令と加算命令を順次実行する場合と比較して、計算を高速に実行することができる(*1)。

(*1) 乗算命令と加算命令を順次実行する場合、乗算後かつ加算前に、正規化処理(*2)と丸め処理が行われる。乗算と加算を一つの命令で実行する場合、乗算後かつ加算前の正規化処理と丸め処理をスキップし、加算後にのみ正規化処理と丸め処理を行う。ただしスキップされた処理により乗算・加算の順次実行の場合と比較し、丸め誤差程度の計算誤差が生じることがある。

(*2) 簡単のため2進数の場合に限定する。2進数の場合の正規化とは、数学上、2を基数とするべき乗と2進数小数を用いた記法のうち小数点の左側に2進数の0以外の文字が一つだけである $1.xxx \times 2^{yyy}$ の形にする処理である。計算機上では、決められたデータ型でのビット数に応じ、正規化できる実数は特定の範囲に限定される。

積和演算において乗算と加算を一つの命令で実行することが効果的な可能性のあるループ例を右の図に示す。

ループ例

```
DOUBLE PRECISION::A(N),B(N),X,C0,C1,C2,C3,C4,C5,C6,C7
  ...
DO I=1,N
  X=B(I)
  A(I)=C0+X*(C1+X*(C2+X*(C3+X*(C4+X*(C5+X*(C6+X*C7))))))
ENDDO
```

- 本ループ例では積和演算 C6+X*C7 の結果（T6とおく）に対して積和演算 C5+X*T6 となる。以下同様となり、1反復あたり全部で7回の積和演算である。

■ 積和演算命令のコンパイルオプションおよび指示行

以下に示すコンパイルオプションまたは指示行により、乗算と加算（または減算）を一つの命令で実行するFloating-Point Multiply-Add/Subtract演算命令（以下、単に、浮動小数点の積和演算命令と呼ぶ）を使用する最適化の有効化または無効化を指示することができる。

- コンパイルオプション
  - 有効化：`-Kfp_contract`
  - 無効化：`-Knofp_contract`（デフォルト）
- 指示行（プログラム単位、DOループ単位、および配列代入文単位のみで指定可能）
  - 有効化：`!OCL FP_CONTRACT`
  - 無効化：`!OCL NOFP_CONTRACT`
- 注意事項
  - 有効化オプション`-Kfp_contract`は`-Kfast`の指定により誘導される。
  - ⚠ 本最適化の有効化により丸め誤差程度の違いを生じることがある。
  - 有効化オプションおよび有効化指示行は`-O1`オプション以上が有効な場合に意味がある。
  - デフォルトの無効化オプション`-Knofp_contract`は`-Kfp_precision`の指定でも誘導される。

■ 積和演算命令使用の効果をみる例 [1/4]：ループ例

ここでは、積和演算命令の使用の効果をみる例として、2ページ前のスライドのループ例（下の図）を対象とし、積和演算命令を使用しない場合と使用する場合のループを実行し、CPU性能解析レポートを採取し比較した結果を示す。

ループ例

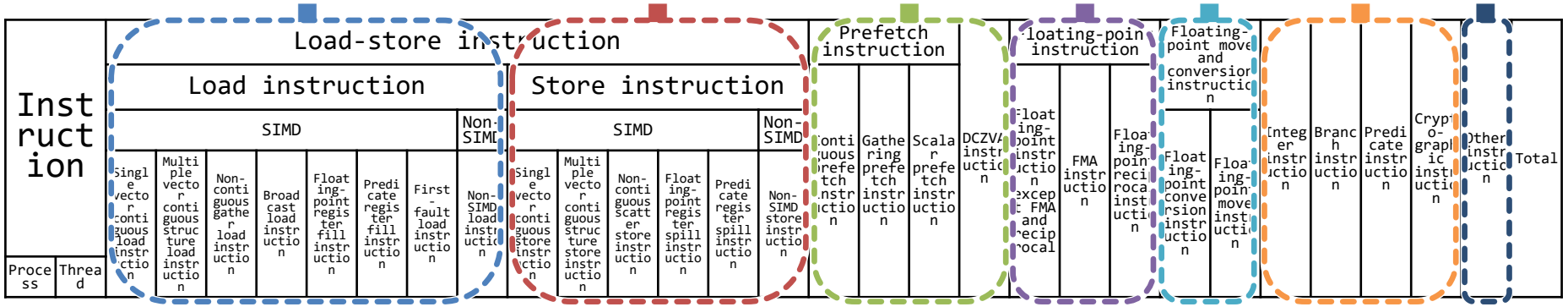
```
INTEGER,PARAMETER::N=10000
DOUBLE PRECISION::A(N),B(N)
DO I=1,N
  X=B(I)
  A(I)=C0+X*(C1+X*(C2+X*(C3+X*(C4+X*(C5+X*(C6+X*C7))))))
ENDDO
```

- 積和演算命令を使用しない場合と使用する場合とで、生成される命令の種類が異なる。実際に実行された命令の内訳はCPU性能解析レポートの「Instruction」欄に表示される。
  - ここでは、「Instruction」欄から、命令数の内訳、さらに浮動小数点命令数の内訳を調べることとする。実行された命令の違いの効果は実行時間の内訳（Cycle Accounting）にどのように現れるかを確認する。
- 本小節内の以降では、図のループに指示行!OCL NOFP_CONTRACTを指定（積和演算命令を使用しないと指定）した場合を「asis」と記し、指示行!OCL FP_CONTRACTを指定（積和演算命令を使用すると指定）した場合を「tune」と記すこととする。
- 最適化のコンパイルオプションは-Kfastとし、逐次実行する（計測のためのループの繰り返し数NLOOPを100000とする）。指示行を有効にするため-Koclも指定する。
- -Kfastにより-Kfp_contractが誘導されるので有効化の指示行!OCL FP_CONTRACTを指定する必要はないが、比較を明瞭にするために付けることとする。

■ 積和演算命令使用の効果をみる例 [2/4]：命令数の内訳

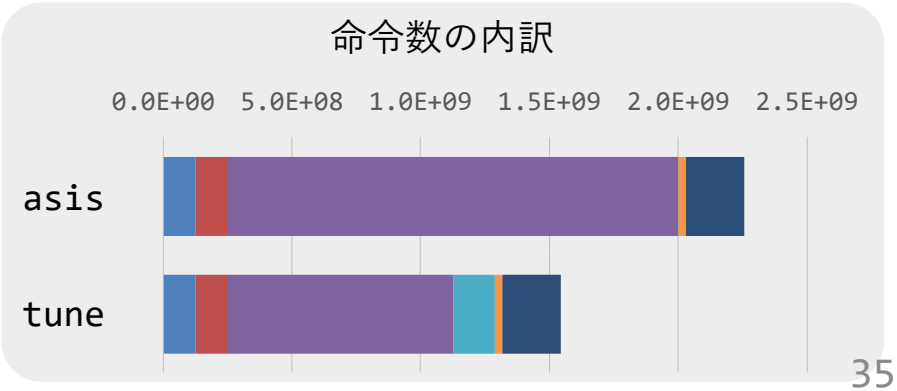
- 実行された命令の内訳は下の表に示すCPU性能解析レポートの「Instruction」欄に表示される。
- この欄に表示される命令数の内訳を点線で囲まれたように分類する。
- 分類された各項目の見出しを右上の図に示すように定める。

- load命令合計
- store命令合計
- Prefetch 命令およびDCZVA命令
- 浮動小数点演算命令（FMA, reciprocal、およびその他）
- 浮動小数点 move および conversion命令
- Integer, Branch, Predicate, およびCrypto-graphic命令
- その他命令



- 命令数の内訳の比較結果を右の図に示す。
  - 紫色部分（■）は、効率の良い積和演算FMAと逆数近似演算reciprocal、およびその他効率の悪い浮動小数点演算命令の合計数を意味する。
  - tune（!OCL FP_CONTRACTを指定）では効率の良いFMAとreciprocal、およびその他効率の悪い浮動小数点演算命令数の合計数（■）が減少したことが確認される。

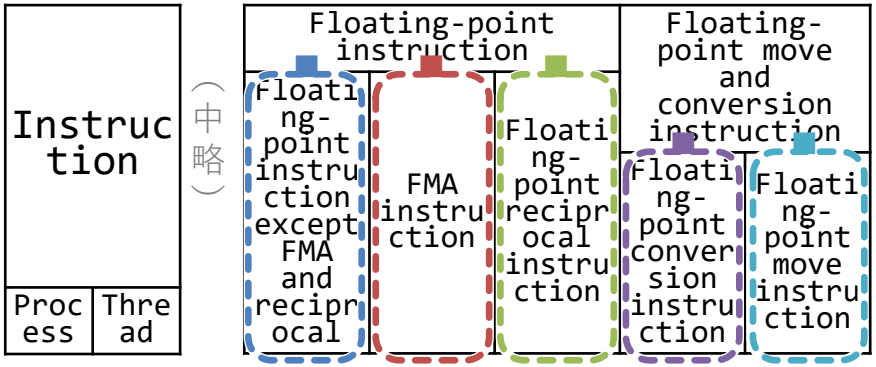
asis: !OCL NOFP_CONTRACTを指定
tune: !OCL FP_CONTRACTを指定



■ 積和演算命令使用の効果をみる例 [3/4]：浮動小数点命令数の内訳

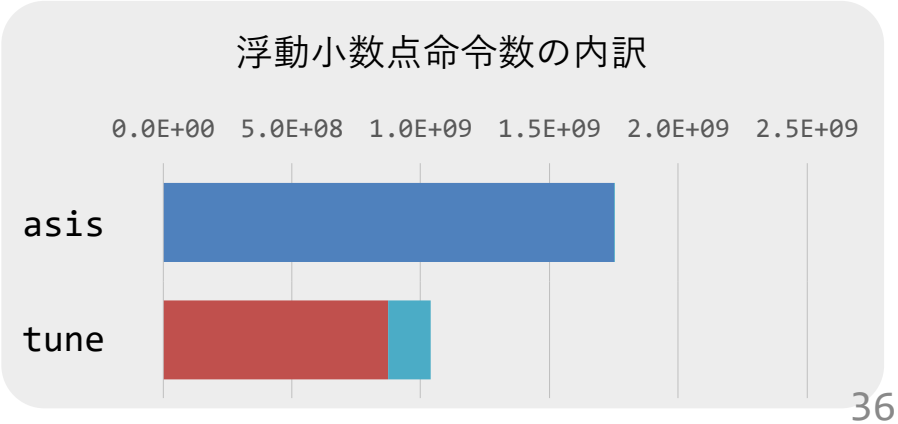
- 1ページ前の「Instruction」欄のうち、浮動小数点命令数の部分の内訳を左下の表で点線で囲まれたように分類する。
- 分類された各項目の見出しを右下の図に示すように定める。

「Instruction」欄のうち、浮動小数点命令数の部分



- 浮動小数点命令数の部分について、その内訳を比較した結果を示す。
  - asis（!OCL NOFP_CONTRACTを指定）では効率の良いFMAとreciprocalを除く効率の悪い浮動小数点演算命令（■）が大多数である。
  - 対して、tune（!OCL FP_CONTRACTを指定）では効率の良いFMA命令（■）に置き換わり、FMA命令とmove命令（■）の合計数も減少していることが確認される。

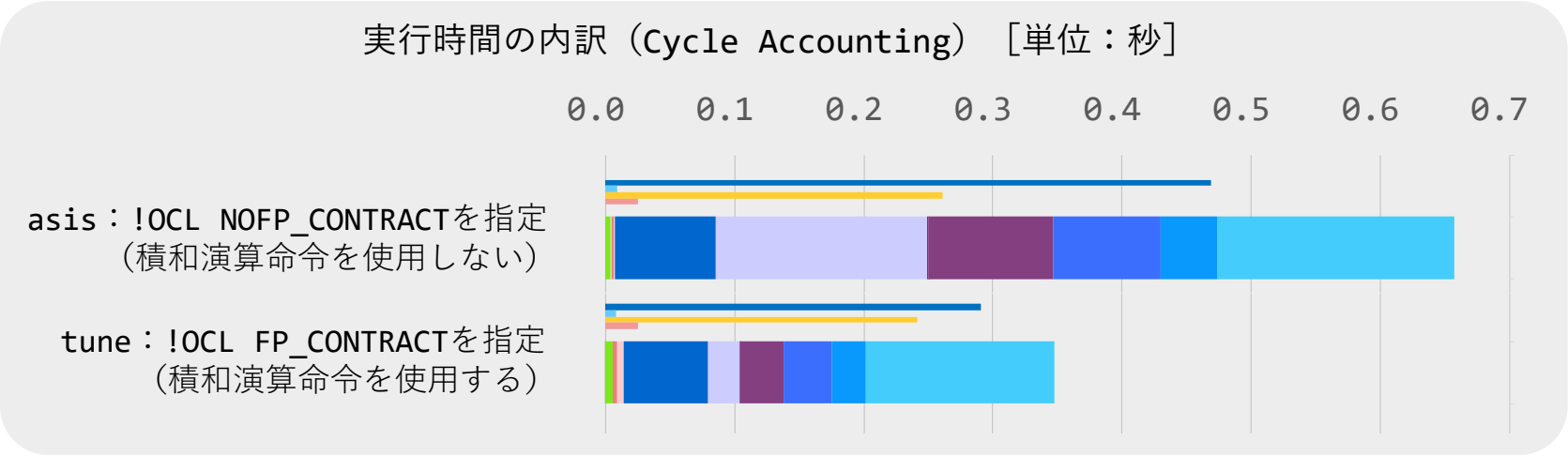
asis: !OCL NOFP_CONTRACTを指定
tune: !OCL FP_CONTRACTを指定



■ 積和演算命令使用の効果をみる例 [4/4]：実行時間の内訳

■ 実行時間の内訳（Cycle Accounting）を以下の図に示す（細い棒グラフ群はビジー時間を表す。プロファイラの章を参照）。

- Cycle Accountingにおいて、!OCL FP_CONTRACT指定では浮動小数点演算待ちの時間（■）および1〜4命令コミット（■、■、■、および■）の合計時間が減少したことが確認される。



■ 以上から、積和演算命令の使用により浮動小数点演算が効率化したことが確認される。

- 各種最適化手法
  - 演算の効率化

本節の内容

- 積和演算命令の使用
- 逆数近似命令の使用
- マルチ演算関数への変換
- 組込み関数のインライン展開
- 演算評価方法の変更

■ 逆数近似演算とは

除算は加算や乗算に比べ一般に計算時間がかかる。逆数近似演算とは除算を用いず逐次的に逆数の近似値を求める方法である。「富岳」では単精度および倍精度の浮動小数点実数の除算やSQRT関数について、逆数近似演算を実行する命令（以下、単に逆数近似命令と呼ぶ）を使用すると指定する(*1)ことにより、使用しないと指定する(*1)場合と比べて計算を高速に実行することができる。

(*1) 以降のスライドで示す。

■ 逆数近似命令が適用される例

以下に逆数近似命令が適用される例を示す。

■ 1/Y

```
DO I=1,N
  Z(I) = X(I) / Y(I)
ENDDO
```

■ $1/\sqrt{X}$

```
DO I=1,N
  Y(I) = 1.0 / SQRT(X(I))
ENDDO
```

■ $\sqrt{X}$ を $X/\sqrt{X}$ として計算

```
DO I=1,N
  Y(I) = SQRT(X(I))
ENDDO
```

■ 逆数近似命令のコンパイルオプションおよび指示行

以下に示すコンパイルオプションまたは指示行により、単精度または倍精度の浮動小数点除算またはSQRT関数について、逆数近似演算命令を使用する最適化の有効化または無効化を指示することができる。

- コンパイルオプション
  - 有効化：`-Kfp_relaxed`
  - 無効化：`-Knofp_relaxed`（デフォルト）
- 指示行（プログラム単位、DOループ単位、および配列代入文単位のみで指定可能）
  - 有効化：`!OCL FP_RELAXED`
  - 無効化：`!OCL NOFP_RELAXED`
- 注意事項
  - ⚠ 本最適化の有効化により丸め誤差程度の違いを生じることがある。また、数値演算の取り扱いがIEEE 754に準拠しなくなる。詳細は「Fortran使用手引書」を参照して下さい。
  - 有効化オプションまたは有効化指示行により積和演算命令も使用される。
  - 有効化オプションおよび有効化指示行は-O1オプション以上が有効な場合に意味がある。
  - 有効化オプション-Kfp_relaxedは-Kfastの指定により誘導される。
  - デフォルトの無効化オプション-Knofp_relaxedは-Kfp_precisionの指定でも誘導される。
  - ⚠ 有効化オプションまたは指示行が指定されていても、-NRtrapオプション(*1)が有効、かつ-Knosimdまたは-KNOSVEオプションのいずれかが有効である場合、SQRT関数を逆数近似演算に変換する最適化が抑止される。

    (*1) -NRtrapオプションが有効な場合、エラーと割り込み事象を検出する。デフォルトは-NRnotrapである。

■ 逆数近似命令使用の効果をみる例 [1/2]：ループ例

ここでは、逆数近似命令の使用の効果をみる例として、前スライドのループ例（1/Y 型の計算）を対象とし、逆数近似命令を使用しない場合と使用する場合のループを実行し、CPU性能解析レポートを採取し比較した結果を示す。

ループ例（1/Y 型の計算）

```
INTEGER,PARAMETER::N=10000
DOUBLE PRECISION::A(N),B(N),X(N)
...
DO I=1,N
  X(I) = A(I) / B(I)
ENDDO
```

- 図のループに指示行!OCL NOFP_RELAXEDを指定（逆数近似命令を使用しないと指定）した場合を「asis」と記し、指示行!OCL FP_RELAXEDを指定（逆数近似命令を使用すると指定）した場合を「tune」と記すこととする。
- 最適化のコンパイルオプションは-Kfastとし、逐次実行する（計測のためのループの繰り返し数NLOOPを10000とする）。指示行を有効にするため-Koclも指定する。
- -Kfastにより-Kfp_relaxedが誘導されるので有効化の指示行!OCL FP_RELAXEDを指定する必要はないが、比較を明瞭にするために付けることとする。

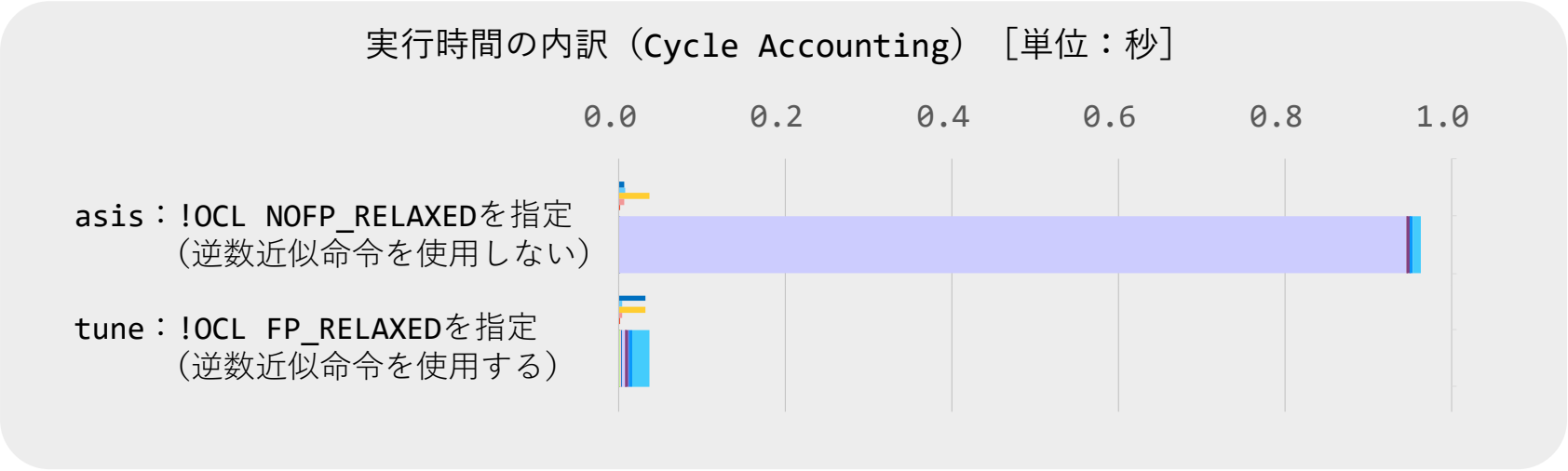
■ 逆数近似命令使用の効果をみる例 [2/2]：実行結果

■ 実行時間の内訳（Cycle Accounting）を以下の図に示す（細い棒グラフ群はビジー時間を表す。プロファイラの章を参照）。

ループ例（1/Y 型の計算）（再掲）

```
DO I=1,N
  X(I) = A(I) / B(I)
ENDDO
```

- Cycle Accountingにおいて、!OCL FP_RELAXED指定では浮動小数点演算待ちの時間（■）が減少したことが確認される。



■ 以上から、逆数近似命令の使用により浮動小数点演算が効率化したことが確認される。

- 各種最適化手法
  - 演算の効率化

本節の内容

- 積和演算命令の使用
- 逆数近似命令の使用
- マルチ演算関数への変換
- 組込み関数のインライン展開
- 演算評価方法の変更

■ マルチ演算関数とは

マルチ演算関数とは、1回の呼出しで複数の引数に対する同種の組込み関数計算および演算を行うことにより、実行性能を向上させた関数である。

■ マルチ演算関数の対象となる組込み関数および演算

■ Fortranの場合

【単精度実数型および倍精度実数型】ACOS、ASIN、[ATAN]、[ATAN2]、[COS]、ERF、ERFC、[EXP]、[EXP10]、[LOG]、[LOG10]、[SIN]、べき乗演算
【単精度複素数型よび倍精度複素数型】ACOS、ACOSH、ASIN、ASINH、[ATAN]、ATANH、[COS]、COSH、COSQ、[EXP]、[LOG]、[SIN]、SINH、SINQ、[TAN]、TANH
【整数型】ISHFT

■ 【参考】C/C++ tradモードの場合

【単精度実数型および倍精度実数型】acos、asin、[atan]、[atan2]、[cos]、erf、erfc、[exp]、[log]、[log10]、[sin]、[pow]

[]：オプション-Kilfuncによる組込み関数のインライン展開の対象。組込み関数のインライン展開は次の小節で説明します。

上記関数のマルチ演算関数の注意事項については「Fortran使用手引書」、「C言語使用手引書」および「C++言語使用手引書」を参照して下さい。

■ マルチ演算関数のコンパイルオプションおよび指示行

以下に示すコンパイルオプションまたは指示行により、組込み関数および演算（前スライド「Fortranの場合」参照）をマルチ演算関数に変換する最適化の有効化または無効化を指示することができる。

- ■ コンパイルオプション
  - ■ 有効化：`-Kmfunc[=level]`　　*level*: {1|2|3}(*1)　　*level*の省略値：1
  - ■ 無効化：`-Knomfunc`（デフォルト）
- ■ 指示行（プログラム単位、DOループ単位、および配列代入文単位のみで指定可能）
  - ■ 有効化：`!OCL MFUNC[(level)]`　　*level*: {1|2|3}(*1)　　*level*の省略値：1
  - ■ 無効化：`!OCL NOMFUNC`
- ■ 注意事項
  - ⚠ 本最適化の有効化の場合と無効化の場合とでアルゴリズムが異なることにより、実行結果に副作用（精度の違い）を生じることがある。その他の副作用については「Fortran使用手引書」を参照して下さい。
  - ■ 有効化オプションおよび有効化指示行は`-O2`オプション以上が有効な場合に意味がある。
  - ■ 有効化オプション`-Kmfunc=1`は`-Kfast`の指定により誘導される。
  - ■ デフォルトの無効化オプション`-Knomfunc`は`-Kfp_precision`の指定でも誘導される。

(*1) *level*の各値については次スライドで説明する。

■ マルチ演算関数のlevel

`level=1` 引数の多重度をSIMD長と同じ値としたマルチ演算関数を使用する。

`level=2` -Kmfunc=1の機能に加え、多重度をコンパイラが自動的に決定する値としたマルチ演算関数も使用する。

`level=3` level=2の機能に加え、IF構文などを含むループに対しても多重度をコンパイラが自動的に決定する値としたマルチ演算関数を使用する。

■ -Kmfunc=3または!OCL MFUNC(3)を指定する際の注意事項

⚠ IF文の真率が低い場合、level=1またはlevel=2より実行性能が低下することがある。

⚠ IF文を含むループのマルチ演算関数化の影響により、実行時異常終了となる場合がある。

右の図にそのような可能性のある例を示す。本例ではマルチ演算関数化により、IF文の条件 I.LE.1000 に関係なく、DO文の繰返し数分 2000 の配列要素をマルチ演算関数化された組込み関数の引数として使用するため、配列Bは宣言範囲1から1000を超えて引用される。そのため、記憶保護例外（読み出し）が発生した旨のエラーメッセージが出力され、プログラムの実行が中断することがある。対処方法としては、翻訳時オプションで-Kmfunc=3を指定しないようにして下さい。

ループ例（実行時異常終了の可能性）

```
DIMENSION A(1000),B(1000)
!OCL MFUNC(3)
DO I=1,2000
  IF (I.LE.1000) THEN
    A(I) = COS(B(I))
  ENDIF
ENDDO
```

■ マルチ演算関数化された場合のコンパイルリスト例 [1/3]

本小節内の以降ではマルチ演算関数の対象である関数として倍精度実数型の関数ASINを選び、関数ASINを使用するループ例（下の図）について、指示行!OCL MFUNC(*level*)を指定した場合のコンパイルリスト例を示す。

ループ例（マルチ演算関数使用）

```
INTEGER,PARAMETER::N=10000
DOUBLE PRECISION::A(N),X(N)
  ...
!OCL MFUNC(level)
  DO I=1,N
     IF ( M(I) ) THEN
      X(I) = ASIN(A(I))
    ENDIF
  ENDDO
```

■ マルチ演算関数化された場合のコンパイルリスト例 [2/3]

■ -Kfast および !OCL MFUNC(2) を指定した場合

コンパイルオプションに -Kfastを(*1)、指示行で !OCL MFUNC(2) を指定した場合のループ例とそのコンパイルリスト例を以下の図に示す。（!OCL MFUNC(1) を指定した場合も同様なコンパイルリストである。）紙面の都合上、出力を加工している。

(*1) その他-Kocl,optmsg=guide -Nlst=tも指定する。

- 本例では関数ASINがマルチ演算関数化され、「MULTI-OPERATION FUNCTION」および「jwd8300o-i ： 組込み関数をマルチ演算関数に変換しました。」が表示される。
- jwd8664のメッセージが示すように、ソフトウェアパイプライニングは適用されていない。

```
30    1          !OCL MFUNC(2)
31    2    2v      DO I=1,N
32    3    2v        IF ( M(I) ) THEN
33    3    2v          X(I) = ASIN(A(I))
34    3    2v        ENDIF
35    2    2v      ENDDO
```

Iループの<<< Loop-information >>>

```
<<<  [OPTIMIZATION]
<<<    SIMD(VL: 8)
<<<    MULTI-OPERATION FUNCTION
<<<      （以下、省略）
```

Iループのメッセージ　　～：ファイル名部分は省略する

```
jwd8300o-i  "～", line 31: 組込み関数をマルチ演算関数に変換しました。
jwd8664o-i  "～", line 31: ループ内に関数呼出しなどの最適化対象外の命令があるため、
ソフトウェアパイプライニングを適用できません。
 （以下、省略）
```

■ マルチ演算関数化された場合のコンパイルリスト例 [3/3]

■ -Kfast および !OCL MFUNC(3) を指定した場合

コンパイルオプションに -Kfastを(*1)、指示行で !OCL MFUNC(3) を指定した場合のループ例とそのコンパイルリスト例を以下の図に示す。紙面の都合上、出力を加工している。

(*1) その他-Kocl,optmsg=guide -Nlst=tも指定する。

- 本例では関数ASINがマルチ演算関数化され、「MULTI-OPERATION FUNCTION」および「jwd8300o-i ： 組込み関数をマルチ演算関数に変換しました。」が表示される。
- jwd8204のメッセージが示すように、ソフトウェアパイプライニングが適用される。

```
30   1          !OCL MFUNC(3)
31   2   2v       DO I=1,N
32   3   2v         IF ( M(I) ) THEN
33   3   2v           X(I) = ASIN(A(I))
34   3              ENDIF
35   2            ENDDO
```

Iループの<<< Loop-information >>>

```
<<<  [OPTIMIZATION]
<<<    SIMD(VL: 16)
<<<    SOFTWARE PIPELINING(IPC: 1.83,
ITR: 192, MVE: 2, POL: S)
<<<    MULTI-OPERATION FUNCTION
<<<      （以下、省略）
```

Iループのメッセージ　　～：ファイル名部分は省略する

```
jwd8300o-i  "～", line 31: 組込み関数をマルチ演算関数に変換しました。
jwd8204o-i  "～", line 31: 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  "～", line 31: 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。
（以下、省略）
```

■ マルチ演算関数化されたループの実行結果例 [1/2]

- ASINを使用するループ例（右下図に再掲）について、前述のコンパイルリスト例では、!OCL MFUNC(2)ではソフトウェアパイプライニングが適用されず、!OCL MFUNC(3)ではソフトウェアパイプライニングが適用された。
- そこで、ASINを使用するループ例について、IF文がある場合に !OCL MFUNC(2) と !OCL MFUNC(3) を指定したループを実行し、CPU性能解析レポートを採取し、結果を比較検討した例を示す。IF文の真率は50%とする。
- 最適化のコンパイルオプションは-Kfastとし、逐次実行する（計測のためのループの繰り返し数NLOOPを10000とする）。指示行を有効にするため-Koclも指定する。

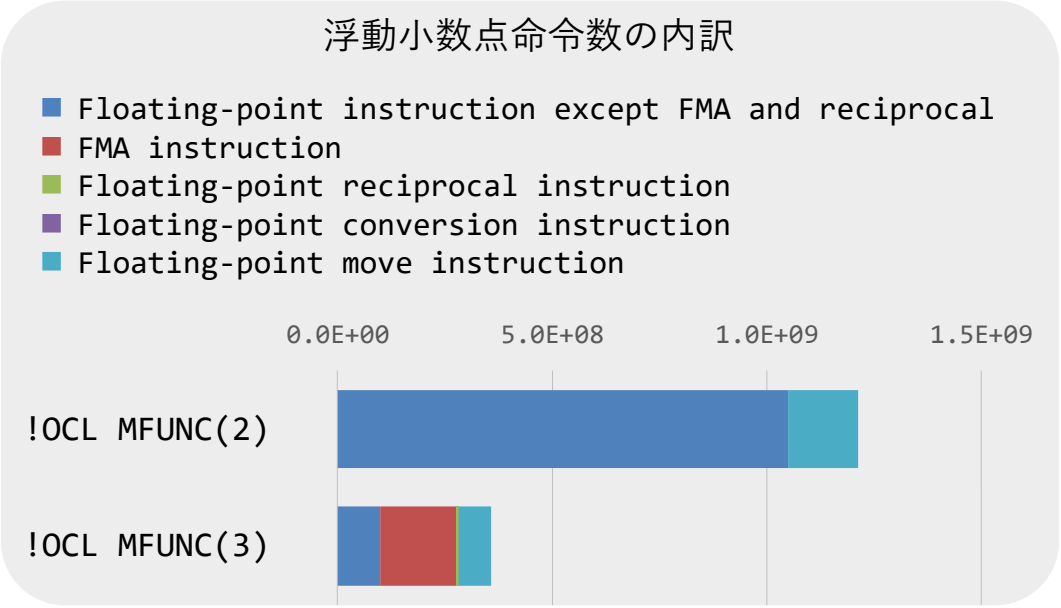
ループ例（マルチ演算関数使用）［再掲］

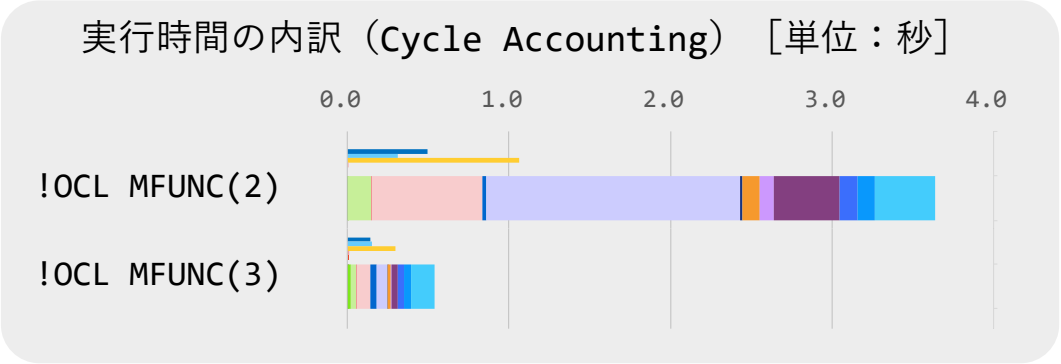
```
INTEGER,PARAMETER::N=10000
DOUBLE PRECISION::A(N),X(N)
  ...
!OCL MFUNC(level)
  DO I=1,N
     IF ( M(I) ) THEN
      X(I) = ASIN(A(I))
    ENDIF
  ENDDO
```

■ マルチ演算関数化されたループの実行結果例 [2/2]

■ 浮動小数点命令の数の内訳を右上の図に示す。!OCL MFUNC(3)では積和演算と逆数近似以外の（効率の悪い）浮動小数点命令の数（■）が減少し、（効率の良い）積和演算命令（■）が実行されていることが確認される。

■ 実行時間の内訳（Cycle Accounting）を右下の図に示す（細い棒グラフ群はビジー時間を表す。プロファイラの章を参照）。上記の結果と、!OCL MFUNC(3)ではソフトウェアパイプライニングが適用されることから、!OCL MFUNC(3)では演算が効率的に行われ、演算待ち（■）が改善されていることが確認される。





- 各種最適化手法
  - 演算の効率化

本節の内容

- 積和演算命令の使用
- 逆数近似命令の使用
- マルチ演算関数への変換
- 組込み関数のインライン展開
- 演算評価方法の変更

■ 組込み関数のインライン展開

関数のインライン展開とは、関数の呼出しがある場合に、その呼出し箇所に関数の本体を直接展開する最適化である。インライン展開により、関数の引数や関数の値の受け渡し、および呼出し・復帰のためのオーバヘッドが削除される。また、展開された部分が呼出し元と一体化されるので、他の最適化が促進される可能性がある。

以下に示す組込み関数および演算をインライン展開する最適化を使用することで、実行性能が向上する可能性がある。

■ インライン展開の対象となる組込み関数および演算

■ Fortranの場合

【単精度実数型および倍精度実数型】ATAN、ATAN2、COS、EXP、EXP10、LOG、LOG10、SIN、TAN、および、べき乗演算(*1)
(*1) 基数と指数が単精度実数型同士および倍精度実数型同士
【単精度複素数型および倍精度複素数型】ABSおよびEXP

■ 【参考】C/C++ tradモードおよびclangモードの場合

【単精度実数型および倍精度実数型】atan、atan2、cos、exp、log、log10、pow、sin、およびtan
【単精度複素数型および倍精度複素数型】absおよびexp

■ 組込み関数のインライン展開：コンパイルオプションおよび指示行

以下に示すコンパイルオプションまたは指示行により、組込み関数および演算（前スライド「Fortranの場合」参照）をインライン展開する最適化の有効化または無効化を指示することができる。

- コンパイルオプション
  - 有効化：`-Kilfunc[=kind](*1)`　　*kind*: {loop|procedure}　*kind*の省略値：procedure
  - 無効化：`-Knoilfunc`（デフォルト）
- 指示行（プログラム単位、DOループ単位、および配列代入文単位のみで指定可能）
  - 有効化：`!OCL ILFUNC`
  - 無効化：`!OCL NOILFUNC`
- 有効化オプションおよび有効化指示行は`-O1`オプション以上が有効な場合に意味がある。

(*1) 有効化オプションの引数***kind***

- `-Kilfunc=loop`　　ループに含まれる組込み関数およびべき乗演算をインライン展開する。ループに含まれない組込み関数およびべき乗演算はインライン展開されない。
- `-Kilfunc=procedure`　　プログラム単位に含まれる組込み関数およびべき乗演算をインライン展開する。

- 【参考】C/C++にはtradモード・clangモードともに本最適化の指示行は存在しない。ただしこの事は今後変わる可能性がある。

■ インライン展開を行う場合の注意事項

- ■ コンパイラは性能低下が生じる可能性があると判断した場合インライン展開を行わない。
- ■ インライン展開の最適化が行われると以下の動作となる可能性があるので注意して下さい。

⚠ 実行結果に以下の副作用を生じることがある：

- `-Knofp_relaxed`および`-Knofp_contract`オプションを指定した場合でも、本`-Kilfunc`オプションまたは`!OCL ILFUNC`指示行により組込み関数がインライン展開された部分では、逆数近似演算命令および浮動小数点の積和演算命令（または積差演算命令）が使用される可能性がある。
- 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、`-Kfp_relaxed`と同様の副作用が生じる可能性がある。

⚠ エラー処理の動作が以下のように変わる可能性がある：

- 組込み関数の引数にエラーがある場合にも、エラーメッセージは出力されずエラー処理も行なわれない。
- `-NRtrap`オプション`(*1)`が無効な場合でも、プログラムの論理上は発生しない浮動小数点例外を発生させることがある。

`(*1)` `-NRtrap`オプションが有効な場合、エラーと割り込み事象を検出する。デフォルトは`-NRnotrap`である。

⚠ 関数`atan2`は、`-X03`または`-X08`オプションを指定した場合にも、`Fortran95`の言語仕様で計算される。

■ 組込み関数のインライン展開のループ例

以下の図（再掲）に示すように、倍精度実数型の双曲線関数 SINH はインライン展開の対象でなく、倍精度実数型の指数関数 EXP はインライン展開の対象である。

Fortranの場合のインライン展開の対象（再掲、抜粋）

【単精度実数型および倍精度実数型】ATAN、ATAN2、COS、EXP、EXP10、LOG、LOG10、SIN、TAN、および、べき乗演算

本小節内の以降では、倍精度実数型の双曲線関数 SINH を使用する以下のループ（下の左図）について、SINH を EXP で表した以下のループ（下の右図）と実行時間を比較する例を示す。

ループ例1（SINHを使用）

```
INTEGER,PARAMETER::N=10000
DOUBLE PRECISION::A(N),X(N)
  ...
!OCL ILFUNC （または !OCL NOILFUNC）
DO I=1,N
  X(I) = SINH(A(I))
ENDDO
```

ループ例2（ループ例1のSINHをEXPで表す）

```
INTEGER,PARAMETER::N=10000
DOUBLE PRECISION::A(N),X(N)
  ...
!OCL ILFUNC （または !OCL NOILFUNC）
DO I=1,N
  X(I) = (EXP(A(I))-EXP(-A(I)))*0.5D0
ENDDO
```

- それぞれのループ例に対して指示行!OCL ILFUNC または !OCL NOILFUNC を指定することとし、インライン展開の有効・無効を比較することとする。
- 組込み関数EXPは-Kfastから誘導される-Kmfunc=1によるマルチ演算関数化の対象でもある。インライン展開の最適化とマルチ演算関数化の最適化の両方が有効な場合インライン展開が優先する。EXPがインライン展開されない場合は、-Kmfuncが有効な場合EXPはマルチ演算関数化される。
- 本小節内の以降では説明の便宜上、マルチ演算関数化は論点から除外することとする。マルチ演算関数化を無効に指定し、最適化のコンパイルオプションを-Kfast,nomfuncとする。

■ SINH使用ループ例のコンパイルリスト

- SINH はインライン展開の対象でないので、-Kfast により誘導される -Kilfunc が有効であっても（あるいは図に示すように指示行!OCL ILFUNC を指定しても）インライン展開されない。
- ループ内にSINH関数の呼び出しがあるためメッセージjwd8664が示すようにソフトウェアパイプライニングが適用されない。
- 指示行!OCL NOILFUNCを指定した場合のコンパイルリストも下の図と同様である。

```
28    1          !OCL ILFUNC
29    2     v      DO I=1,N
30    2     v        X(I) = SINH(A(I))
31    2     v      ENDDO
```

Iループの<<< Loop-information >>>

```
<<<  [OPTIMIZATION]
<<<    SIMD(VL: 8)
<<<    PREFETCH(HARD) Expected by compiler :
<<<      A, X
<<<    SPILLS :
<<<      GENERAL   : SPILL 0  FILL 0
<<<      SIMD&FP   : SPILL 0  FILL 0
<<<      SCALABLE  : SPILL 1  FILL 1
<<<      PREDICATE : SPILL 0  FILL 0
```

Iループのメッセージ

```
jwd6001s-i  "～", line 29: このDOループをSIMD化しました。(名前:I)
jwd8664o-i  "～", line 29: ループ内に関数呼出しなどの最適化対象外の命令があるため、ソフトウェアパイプライニングを適用できません。
  [ガイダンス]
（以下、省略）
```

■ SINH使用ループ例の実行結果（経過時間）

- SINH はインライン展開の対象でないので、前スライドのコンパイルリストで見たように、-Kfast により誘導される -Kilfunc が無効であっても有効であっても（あるいは図に示すように指示行!OCL NOILFUNCを指定しても!OCL ILFUNCを指定しても）インライン展開されない。

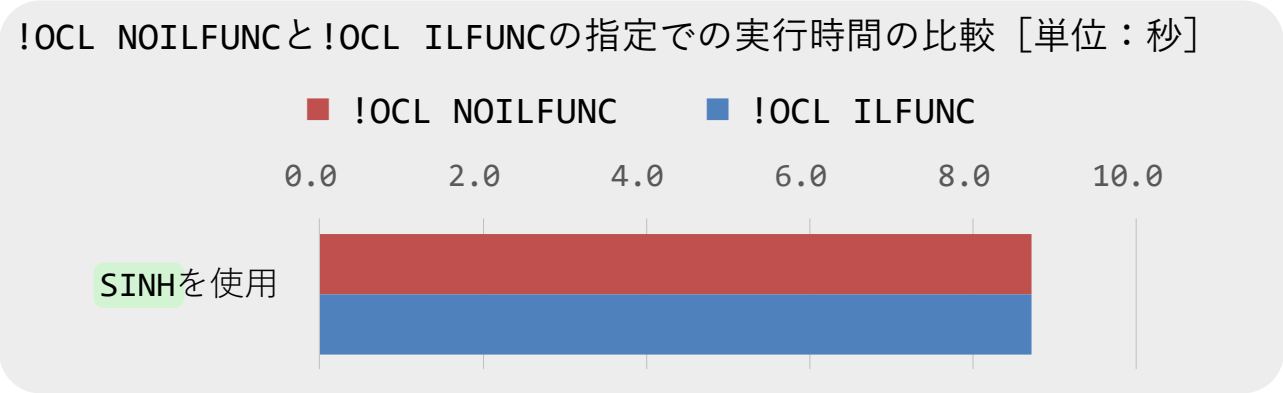
■ !OCL NOILFUNCを指定

```
!OCL NOILFUNC
  DO I=1,N
    X(I) = SINH(A(I))
  ENDDO
```

■ !OCL ILFUNCを指定

```
!OCL ILFUNC
  DO I=1,N
    X(I) = SINH(A(I))
  ENDDO
```

- 指示行!OCL NOILFUNCと!OCL ILFUNCの指定での逐次実行の実行時間（経過時間）の比較結果を下の図に示す（計測のためのループの繰り返し数NLOOPを10000とする）。指示行!OCL NOILFUNCと!OCL ILFUNCの指定によらず経過時間は同等である。



■ EXP使用ループ例のコンパイルリスト：!OCL ILFUNC指定した場合

EXP使用のループ例に対して-Kilfunc が有効な場合（あるいは図に示すように指示行!OCL ILFUNC が有効な場合）のコンパイルリストを下の図に示す。

- EXP はインライン展開の対象であるので、-Kfast により誘導される -Kilfunc が有効な場合（あるいは図に示すように指示行!OCL ILFUNC が有効な場合）インライン展開される。
  - コンパイルリストにはインライン展開されたことを示すメッセージは表示されない。
  - インライン展開されたことでメッセージjwd8204およびjwd8205が示すようにソフトウェアパイプライニングが適用される。

```
28    1         !OCL ILFUNC
29    2    v    DO I=1,N
30    2    v      X(I)=(EXP(A(I))-EXP(-A(I)))*0.5D0
31    2    v    ENDDO
```

Iループの<<< Loop-information >>>

```
<<<  [OPTIMIZATION]
<<<    SIMD(VL: 8)
<<<    SOFTWARE PIPELINING(IPC:
0.74, ITR: 40, MVE: 3, POL: S)
<<<    PREFETCH(HARD) Expected by
compiler :
<<<      A, X
```

Iループのメッセージ

```
jwd6001s-i  "～", line 29: このDOループをSIMD化しました。(名前:I)
jwd8204o-i  "～", line 29: 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  "～", line 29: ループの繰返し数が40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。
```

■ EXP使用ループ例のコンパイルリスト：!OCL NOILFUNC指定した場合

EXP使用のループ例に対して指示行!OCL NOILFUNCを指定した場合のコンパイルリストを下の図に示す。

- 指示行!OCL NOILFUNCを指定した場合、EXP はインライン展開されない。
- EXP は-Kfast から誘導される-Kmfunc=1 によるマルチ演算関数化の対象であるが、本小節では-Kfast,nomfuncを指定しているので、EXPはマルチ演算関数化されない。
- ループ内に（マルチ演算関数化されない）EXP関数の呼出しがあるためメッセージjwd8664が示すようにソフトウェアパイプライニングが適用されない。

```
28    1        !OCL NOILFUNC
29    2    v   DO I=1,N
30    2    v     X(I)=(EXP(A(I))-EXP(-A(I)))*0.5D0
31    2    v   ENDDO
```

Iループの<<< Loop-information >>>

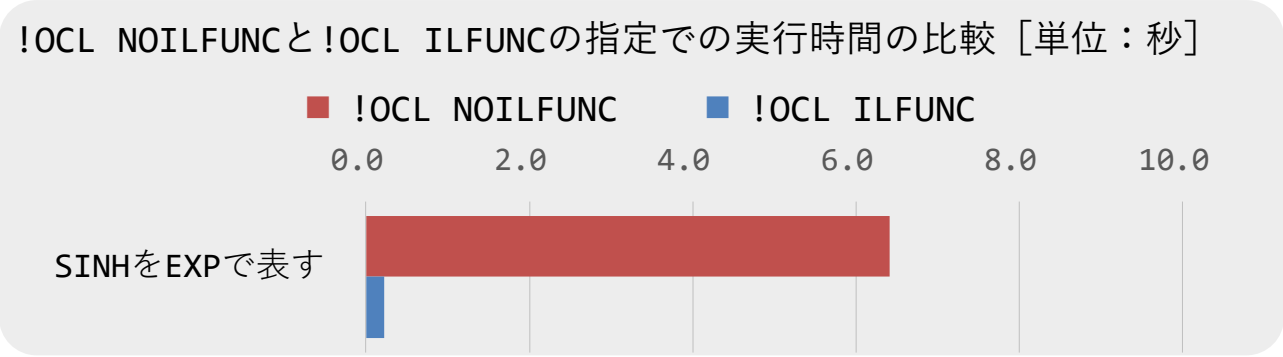
```
<<<  [OPTIMIZATION]
<<<    SIMD(VL: 8)
<<<    PREFETCH(HARD) Expected
by compiler :
<<<      A, X
<<<    SPILLS :
<<<     （以下、省略）
```

Iループのメッセージ

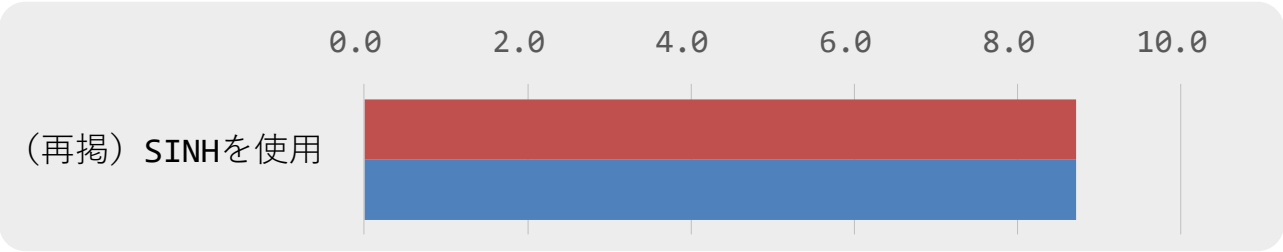
```
jwd6001s-i  "～", line 29: このDOループをSIMD化しました。(名前:I)
jwd8664o-i  "～", line 29: ループ内に関数呼出しなどの最適化対象外の命令があるため、ソフトウェアパイプライニングを適用できません。
  [ガイダンス]
（以下、省略）
```

■ EXP使用ループ例の実行結果

- 前スライドで述べたとおり、EXP はインライン展開の対象であるので、!OCL ILFUNC指定の場合ではインライン展開され、!OCL NOILFUNC指定の場合ではインライン展開されない。それぞれの指定をしたループ例を逐次実行し経過時間を比較した結果を以下の図に示す（計測のためのループの繰り返し数NLOOPを10000とする）。
  - SINH(A(I)) を (EXP(A(I))-EXP(-A(I)))*0.5D0 で記述したループの実行では、赤い棒グラフで示されるインライン展開なし（■）よりも、青い棒グラフで示されるインライン展開あり（■）の方が速いことが分かる。



- SINH(A(I)) を使用したループ実行の経過時間を下の図に再掲する。下の図と上の図との比較から、SINH(A(I)) を使用したループと (EXP(A(I))-EXP(-A(I)))*0.5D0 で記述したループを比較すると、上の図の(EXP(A(I))-EXP(-A(I)))*0.5D0 で記述したループの方が、インライン展開しない場合でも、下の図より速いことが分かる。



■ SINHをEXPで計算する部分を関数化し-xオプションでインライン展開 [1/3]

前スライドの実行結果から、SINH(A(I)) を(EXP(A(I))-EXP(-A(I)))*0.5D0 で記述した方が速い可能性がある。現実のプログラムでSINHが使用される箇所が多数ある場合、その都度EXPで記述するのは煩雑である。そこで、前スライドのループ例でSINHが使用される部分を関数化（利用者定義手続化）し、その利用者定義手続を-xオプションの引数に指定しインライン展開する方法を検討することとする。

■ 右の図に示すような利用者定義手続ZZINLINE_SINHを作成する。

```
3    DOUBLE PRECISION FUNCTION ZZINLINE_SINH(A)
4      IMPLICIT NONE
5      DOUBLE PRECISION::A
6      ZZINLINE_SINH = (EXP(A)-EXP(-A))*0.5D0
7      RETURN
8    END FUNCTION ZZINLINE_SINH
```

■ 右の図のDOループの中で上記の関数を呼ぶ。

```
37    1            !OCL ILFUNC
38    2        v   DO I=1,N
39    2    i   v     X(I) = ZZINLINE_SINH(A(I))
40    2        v   ENDDO
```

■ コンパイルオプション -xzzinline_sinh を指定する。

```
Command line options : -Kfast,nomfunc -xzzinline_sinh -Kocl,optmsg=guide
-Nlst=t （以下、省略）
```

■ SINHをEXPで計算する部分を関数化し-xオプションでインライン展開 [2/3]

■ 下の図に示すように記号 **i** および**jwd8101**が表示され、インライン展開されたことが示される。

```
37   1            !OCL ILFUNC
38   2        v   DO I=1,N
39   2   i    v     X(I) = ZZINLINE_SINH(A(I))
40   2        v   ENDDO
```

Iループの<<< Loop-information >>>

```
<<<  [OPTIMIZATION]
<<<    SIMD(VL: 8)
<<<    SOFTWARE PIPELINING(IPC:
0.74, ITR: 40, MVE: 3, POL: S)
<<<    PREFETCH(HARD) Expected by
compiler :
<<<      A, X
```

Iループのメッセージ

```
jwd6001s-i  "～", line 38: このDOループをSIMD化しました。(名前:I)
jwd8204o-i  "～", line 38: 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  "～", line 38: ループの繰返し数が40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。
jwd8101o-i  "～", line 39: 利用者定義の手続'ZZINLINE_SINH'をインライン展開しました。
```

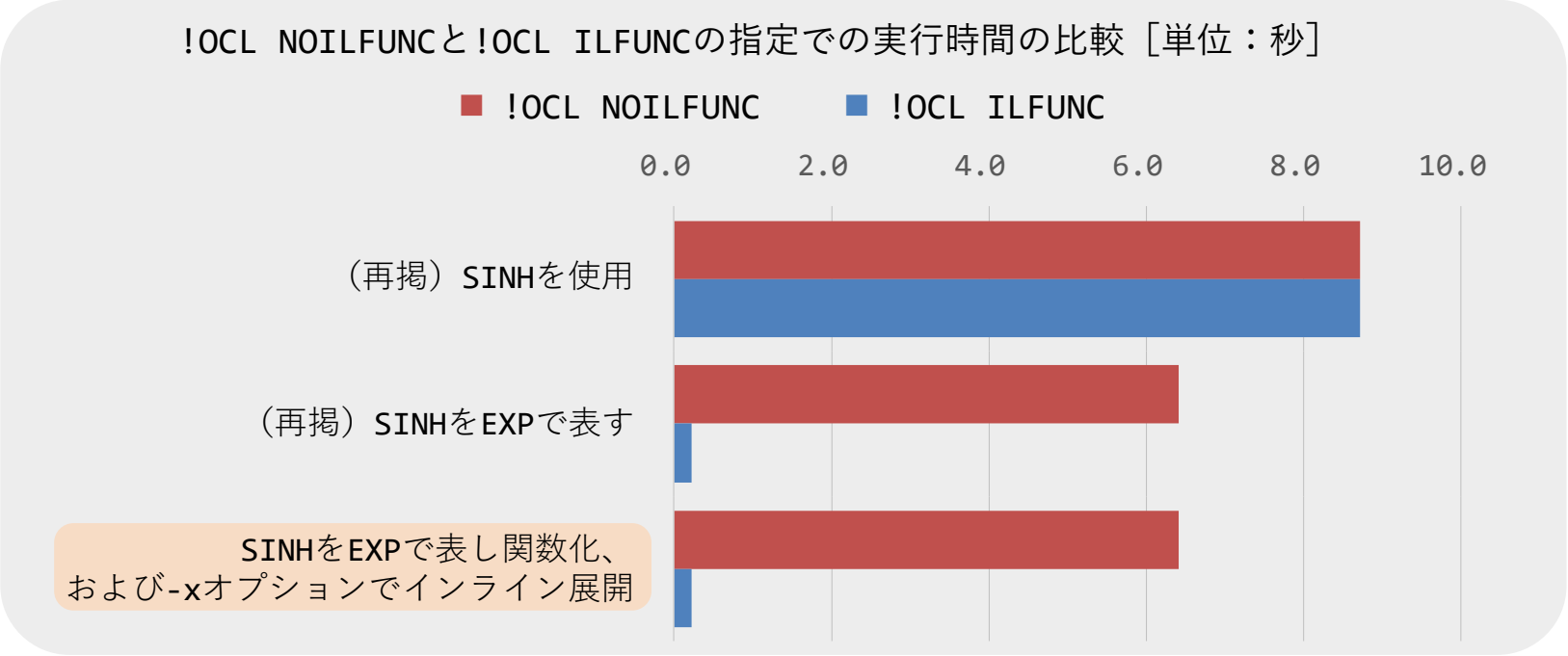
■ 利用者定義手続のインライン展開＋組込み関数のインライン展開

利用者定義手続を-xオプションでインライン展開する場合でも、上の左の図38行目のDOループに対して、37行目で指示行!OCL ILFUNC（または !OCL NOILFUNC）を指定することが可能である。その場合この指示行は前スライド掲載の利用者定義手続ZZINLINE_SINH内の6行目のEXPに対する指定の意味となる。

■ SINHをEXPで計算する部分を関数化し-xオプションでインライン展開 [3/3]

利用者定義手続を-xオプションによりインライン展開したループの実行結果を以下の図一番下に示す。（比較のため、その上に、SINHを使用した場合、およびループ内で直接 (EXP(A(I))-EXP(-A(I)))*0.5D0 と記述した場合を再掲する。）

■ !OCL ILFUNC指定または !OCL NOILFUNC指定いずれに対しても、ループ内で直接 (EXP(A(I))-EXP(-A(I)))*0.5D0 と記述した場合と同等な効果となることが分かる。



- 各種最適化手法
  - 演算の効率化

本節の内容

- 積和演算命令の使用
- 逆数近似命令の使用
- マルチ演算関数への変換
- 組込み関数のインライン展開
- 演算評価方法の変更

■ 演算評価方法の変更とは

ループ内に以下に示す計算が存在する場合、演算の評価方法を変更する最適化を行うことで高速に計算することができる。

- 多項式が加算および乗算などについて交換可能な演算を含む場合
- べき乗の計算方法を変更できる場合
- ループ内の不変式をループ実行前に先行評価できる場合
- リダクション演算において交換可能な演算の順序を変更することで並列化または`SIMD`化を促進できる場合
- 除算を逆数の乗算に変更できる場合
- 複素数の除算の計算方法を変更できる場合
- ループ内の総和または乗積演算の計算方法を変更できる場合

本小節では演算評価方法を変更する最適化について、コンパイルオプションおよび指示行、および最適化の適用効果例を説明する。

■ 演算評価方法変更のコンパイルオプションおよび指示行

以下に示すコンパイルオプションまたは指示行により、演算評価方法を変更する最適化の有効化または無効化を指示することができる。

- コンパイルオプション
  - 有効化：`-Keval`
  - 無効化：`-Knoeval`（デフォルト）
- 指示行（プログラム単位、DOループ単位、および配列代入文単位のみで指定可能）
  - 有効化：`!OCL EVAL`
  - 無効化：`!OCL NOEVAL`
- 注意事項
  - ⚠ 本最適化の有効化により実行結果に副作用（計算誤差および実行時の例外発生など）を生じることがある。詳細は「Fortran使用手引書」を参照して下さい。
  - 有効化オプションおよび有効化指示行は`-O1`オプション以上が有効な場合に意味がある。
  - 有効化オプション`-Keval`は`-Kfast`の指定により誘導される。
    - `-Kfast`指定時`-Keval`が誘導され計算誤差が生じることがある。そのような場合には、`-Kfast`より後ろに`-Knoeval`を指定してください。
  - デフォルトの無効化オプション`-Knoeval`は`-Kfp_precision`の指定でも誘導される。
  - 本最適化が適用された場合「`jwdxxxxz-y`」の形式のメッセージ`(*1)`が出力される可能性がある。

`(*1)` 以降のスライドで示します。

■ 演算評価方法変更により出力される可能性のあるメッセージ [1/2]

オプション-Kevalまたは指示行!OCL EVALを指定したときだけ、コンパイルリストに以下に示すメッセージが出力される可能性がある。

以降のスライドでは項目の前に★を付けた項目（3項目）のみについて説明します。
その他のメッセージについてはここで概略のみ記します。詳細は「Fortran翻訳時メッセージ」を参照して下さい。

★ ■ jwd8209o-i ： 多項式の演算順序を変更しました。

本メッセージは、加算および乗算などの交換可能な演算を含む多項式の演算順序を変更したことを示す。

★ ■ jwd8213o-i ： べき乗の計算方法を変更しました。

本メッセージは、べき乗があるとき、そのべき乗の計算方法を変更したことを示す。

★ ■ jwd8220o-i ： 副作用の可能性のある最適化を行いました。

本メッセージは、実行結果に計算誤差などの副作用を及ぼす可能性のある、以下のいずれかの最適化を行ったことを示す。

- 不変式の先行評価、メモリアクセスの末尾移動（これらの最適化は-Kpreexを指定したときに行われる）。
- 演算評価方法の変更（この最適化は-Kevalを指定したときに行われる）。

■ （次スライドへ続く）

■ 演算評価方法変更により出力される可能性のあるメッセージ [2/2]

■ （前スライドからの続き）

■ jwd5004p-i : リダクション演算を含むDOループを並列化しました。(名前:parm)
本メッセージは、加算および乗算などの交換可能な演算を式の演算順序を変更して並列化したことを示す。parm は自動並列化したDOループのDO変数を示す。

■ jwd5007p-i : リダクション演算を含むDOループを並列化しました。
本メッセージは、加算および乗算などの交換可能な演算を式の演算順序を変更して並列化したことを示す。

■ jwd6004s-i : リダクション演算を含むDOループをSIMD化しました。(名前:parm)
本メッセージは、加算および乗算などの交換可能な演算を式の演算順序を変更してSIMD化したことを示す。

■ jwd8206o-i : 除算を逆数の乗算に変更しました。
本メッセージは、除算があるとき、逆数を求め、その逆数の乗算に変更したことを示す。

■ jwd8207o-i : 複素数の除算の計算方法を変更しました。
本メッセージは、複素数の除算があるとき、その複素数の除算の計算方法を変更したことを示す。

■ jwd8208o-i : ループ内の総和または乗積演算の計算方法を変更しました。
本メッセージは、総和演算(加減算)または乗積演算があるとき、その計算方法を変更したことを示す。

以降のスライドでは項目の前に★を付けた項目（前スライドの3項目）のみについて説明します。

■ 多項式の演算順序の変更

■ 演算評価方法の変更なしの場合の例

- `!OCL NOEVAL`を指定した場合、通常左から順に演算が行われる。

ループ例（演算評価方法の変更なし）

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

■ 演算評価方法の変更ありの場合の例

オプション`-Keval`または指示行`!OCL EVAL`を指定したとき、加算および乗算などの交換可能な演算を含む多項式の演算順序が変更される可能性がある。

- `!OCL EVAL`を指定する。`-Kfast`オプションでコンパイルするとする。
  （なお、`-Kfast`オプション下では指定`!OCL EVAL`は不要であるが、説明の便宜上明示的にするために指定している。）
- コンパイラは、例えば右下の図①の行の赤と青の線で示す順に処理した方が速いと判断した場合、この順に処理する最適化を行い、以下の②の最適化メッセージ`jwd8209`が表示される。

② `jwd8209o-i "～", line 30:` 多項式の演算順序を変更しました。

- この最適化は副作用や精度誤差を起こす可能性があるので、以下の③の最適化メッセージ`jwd8220`が表示される。

③ `jwd8220o-i "～", line 15:` 副作用の可能性のある最適化を行いました。

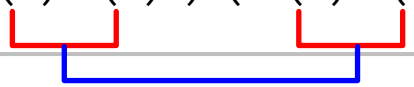
コロンの左は行番号

ループ例（演算評価方法の変更あり）

```
 15: SUBROUTINE SUB_B1
...
 28: !OCL EVAL
 29:   DO I=1,N
 30:     A(I)=A(I)+B(I)+C(I)+D(I)
 31:   ENDDO
```

最適化適用後のソースイメージ

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



■ べき乗の計算方法の変更

オプション-Kevalまたは指示行!OCL EVALを指定したとき、べき乗があると、そのべき乗の計算方法が変更される可能性がある。

■ ループ例を右の図に示す。本ループに対して!OCL EVALを指定する。本ループを実行するプログラムを-Kfastオプションでコンパイルするとする。

（なお、-Kfastオプション下では指定!OCL EVALは不要であるが、説明の便宜上明示的にするために指定している。）

ループ例　コロンの左は行番号

```
 15: SUBROUTINE SUB_B2
...
 28: !OCL EVAL
 29:       DO I=1,N
 30:         X(I) = A(I)**0.5D0      ①
 31:       ENDDO
```

■ 上の図に示すループに対して!OCL EVALを指定した場合、①の行のべき乗の計算に対して、コンパイラが、0.5乗の計算をSQRTの計算に置き換える最適化を行い、以下の②の最適化メッセージjwd8213が表示される。

② `jwd8213o-i  "～", line 30:` べき乗の計算方法を変更しました。

■ この最適化は副作用や精度誤差を起こす可能性があるので、以下の③の最適化メッセージjwd8220が表示される。

③ `jwd8220o-i  "～", line 15:` 副作用の可能性のある最適化を行いました。

■ 演算評価方法変更の効果をみる例 [1/3]：ループ例

ここでは、演算評価方法変更の効果をみる例として、図のループ例（前スライドのべき乗の計算があるループ例）を対象とし、演算評価方法変更の最適化を適用しない場合と適用する場合の比較例を示す。

ループ例

```
INTEGER,PARAMETER::N=10000
DOUBLE PRECISION::A(N),X(N)
  ...
DO I=1,N
  X(I) = A(I)**0.5D0
ENDDO
```

- 図のループに指示行!OCL NOEVALを指定（演算評価方法変更の最適化を適用しないと指定）した場合を「asis」と記し、指示行!OCL EVALを指定（演算評価方法変更の最適化を適用すると指定）した場合を「tune」と記すこととする。
- 最適化のコンパイルオプションは-Kfastとし、逐次実行する（計測のためのループの繰り返し数NLOOPを10000とする）。指示行を有効にするため-Koclも指定する。
- -Kfastにより-Kevalが誘導されるので有効化の指示行!OCL EVALを指定する必要はないが、比較を明瞭にするために付けることとする。

■ 演算評価方法変更の効果をみる例 [2/3]：!OCL EVAL指定の場合

- 右の図に示すループに対して!OCL EVALを指定した場合、0.5乗の計算に対して、コンパイラが、0.5乗の計算をSQRTの計算に置き換える最適化を行い、以下の最適化メッセージjwd8213が表示される。

ループ例（!OCL EVALを指定）

```
!OCL EVAL
  DO I=1,N
    X(I) = A(I)**0.5D0
  ENDDO
```

```
jwd8213o-i   "～", line 30: べき乗の計算方法を変更しました。
```

- 本プログラムを-Kfastオプションでコンパイルしているので、コンパイルリスト中に表示される以下の図のEffective optionsで示すように、-Kfast から誘導される-Kfp_relaxedにより、逆数近似命令を使用する最適化が有効となる。実数のSQRT組込み関数は逆数近似命令の適用対象である。

```
Command line options : -Kfast -Kocl,optmsg=guide （以下省略）
Effective options    : （中略）-Keval （中略）-Kfp_contract -Kfp_relaxed
                       （中略） -Kilfunc=procedure （以下省略）
```

- 結果として「!OCL EVAL指定された A(I)**0.5D0」の計算は 「逆数近似命令が適用された SQRT(A(I))」の計算と同等である。

■ 演算評価方法変更の効果をみる例 [3/3]：!OCL NOEVAL指定の場合

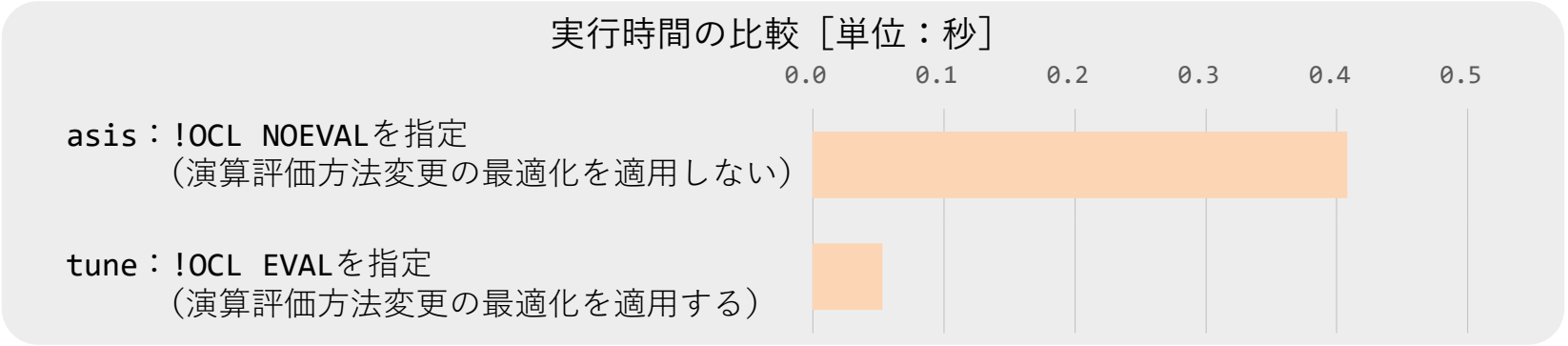
- -Kfastオプション下で、対して!OCL NOEVALを指定した場合、A(I)**0.5D0 の計算はSQRTに変更されず、0.5乗のまま計算される。
  - より正確にいうと、-Kfastオプション下では組込み関数およびべき乗のインライン展開(*1)オプション-Kilfuncも有効となる。-Kilfuncにより0.5乗はインライン展開される。

ループ例（!OCL NOEVALを指定）

```
!OCL NOEVAL
  DO I=1,N
    X(I) = A(I)**0.5D0
  ENDDO
```

(*1) 前小節参照。

- 結果として「!OCL NOEVAL指定した A(I)**0.5D0」の計算は「インライン展開されたべき乗」計算である。
- !OCL NOEVALを指定して実行した場合と、!OCL EVALを指定して実行した場合の実行時間の比較結果を下の図に示す。



- 本ループ例では、インライン展開されたべき乗 A(I)**0.5D0 の計算よりも、逆数近似命令が適用された SQRT(A(I)) の計算の方が速いことが分かる。

内容

- 「富岳」の概略
- プロファイラ（CPU性能解析レポート）
- 各種最適化手法
  - 演算の効率化
  - ソフトウェアパイプライニングの促進
  - SIMD化の促進（ロード命令・ストア命令の効率化、演算の効率化）
  - キャッシュチューニングのための基本事項
- 【付録】

- 各種最適化手法
  - ソフトウェアパイプライニングの促進

本節の内容

ソフトウェアパイプライニングも演算を効率的に行うための最適化手法ですが、特に重要な手法であるため、本節で切り出して取り上げます。本中級編第一部では、ソフトウェアパイプライニングの概念説明、コンパイルオプションと指示行、およびソフトウェアパイプライニング促進のループ例をいくつか説明しています。この中級編第三部では第一部の内容を補足することを目的とします。最初の小節では、本中級編第一部の未受講者を想定し、ソフトウェアパイプライニングの概念の説明をします。その後の二つの小節ではソフトウェアパイプライニングを促進するための手法を二つ紹介します。

- 【第一部からの抜粋】ソフトウェアパイプライニングの概念説明
- 外側ループでのソフトウェアパイプライニング
- レジスタ不足の回避

- 各種最適化手法
  - ソフトウェアパイプライニングの促進

本節の内容

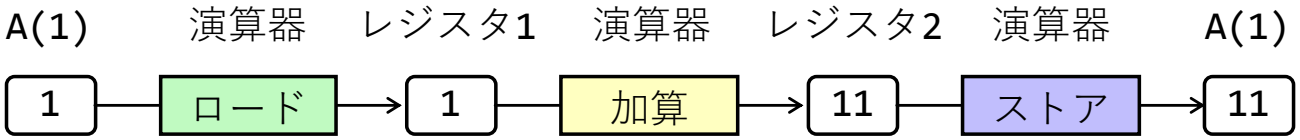
- 【第一部からの抜粋】ソフトウェアパイプライニングの概念説明
- 外側ループでのソフトウェアパイプライニング
- レジスタ不足の回避

■ 動作説明のためのループ例、および簡単化したモデルの説明

■ 右の図のループを実行し、1反復目の動作は以下のようになるとする。

ループ例

```
DO I=1,6
  A(I) = A(I) + 10.0
ENDDO
```



■ 簡単化したモデル （実際のモデルを簡単化して説明するので、実際の動作と同じではない）

■ コア内にはロード、加算、ストアを行う三つの演算器があり(*1)、それぞれ処理を1単位時間(*2)で行なうとする。

(*1) 実際のモデルと個数は異なる。実際のコアでは、ロードとストアの演算器は兼用で2個あり、ロードまたはストアの「アドレス計算」を行なう。

(*2) 実際の演算器では演算の種類に応じて一般に複数の単位時間（サイクル）かかる。

■ 各演算器は同時に実行できるとする。

コア内の演算器（簡単化モデル）	ロード	加算	ストア
処理時間	1単位時間	1単位時間	1単位時間
	同時に実行できるとする		

■ ソフトウェアパイプライニングが行われない場合

■ 演算器は1時点で一つのみ稼働するため、6反復の処理に6×3＝18単位時間かかる。

ループ例

```
DO I=1,6
  A(I) = A(I) + 10.0
ENDDO
```

■ ソフトウェアパイプライニングが行われた場合

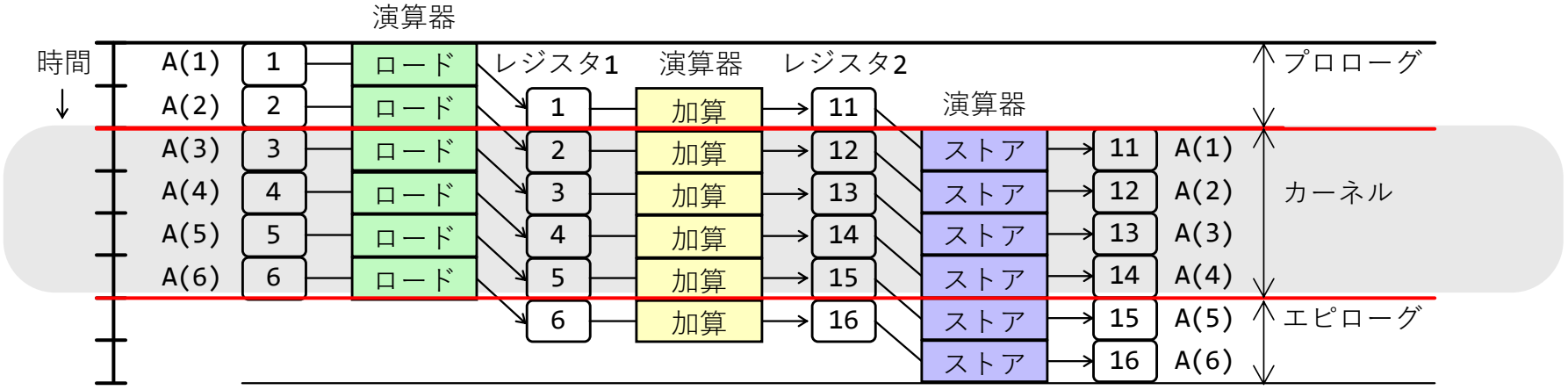
■ 1反復目の実行が終了する前に2反復目を開始する。タイムチャートは下の図のようになる。

■ 演算器は1時点で三つ同時に稼働できる。

■ 6反復の処理は8単位時間に短縮する。

- 三つの演算器がすべて稼働している定常状態の部分をカーネル、カーネルの前の部分をプロローグ、後の部分をエピローグと呼ぶ。

タイムチャート（ソフトウェアパイプライニングが行われた場合）



■ 各演算器は演算待ち無く各単位時間で稼働している。

■ このような状態を命令スケジューリングが良い状態という。

- 各種最適化手法
  - ソフトウェアパイプライニングの促進

本節の内容

    - 【第一部からの抜粋】ソフトウェアパイプライニングの概念説明
    - 外側ループでのソフトウェアパイプライニング
    - レジスタ不足の回避

■ 本小節の背景

最適化メッセージ **jwd8205o-i** は、コンパイラが「ソフトウェアパイプライニング化されたループ（ソフトウェアパイプライニングを適用したループ）」と「ソフトウェアパイプライニング化されないループ（元のループ）」を実行時に選択する分岐を生成したことを告げる。（ここではこのメッセージで提示された値 ***N*** を以下「しきい値」と呼ぶ。）

> **jwd8205o-i "～", line ～:** ループの繰返し数が***N***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。

多重ループについて、最内ループにこのメッセージが表示され、かつ、その全ての外側ループにはこのメッセージが表示されない場合を考える。... ①

最内ループ長が「しきい値」に満たない場合、最内ループはソフトウェアパイプライニング化されない元のループが実行される。その場合多重ループ全体としてソフトウェアパイプライニングは一切かからない。... ②

■ 本小節の内容 [1/2]

本小節では、上記①かつ②のような状況の多重ループに対して、指示行で**CLONE**最適化（クローン最適化）を指示することで、外側ループにソフトウェアパイプライニングを促進させる例を示す。

（次スライドへ続く）

■ 本小節の内容 [2/2]

最初にCLONE最適化とは何か、およびCLONE最適化を指示する指示行!OCL CLONEを説明する。

ループ例として、-Kfastオプション下で最内ループにjwd8205o-iが表示されても実行時の最内ループ長が短かくソフトウェアパイプライニングが働かないループ例とそのコンパイルリストを示す。

CLONE最適化を適用し、適用した場合のコンパイルリストから、外側ループにソフトウェアパイプライニングが適用されることを示す。

CLONE最適化を適用しない場合と適用した場合のループを逐次実行し、CPU性能解析レポートを採取し、比較した結果を示す。CPU性能解析レポートからソフトウェアパイプライニング促進の効果を確認する。

■ CLONE最適化とは

CLONE最適化とは、ループに対して変数（例えばループ長など）の値による条件分岐を生成し、フルアンローリングなどの他の最適化を促進する最適化である。

CLONE最適化は指示行!OCL CLONEで動作する。指示行の書式は次スライドで説明する。

以下の図に本指示行の指定例（左）と最適化後のソースイメージ（右）を示す。

元のループ

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

CLONE最適化後のイメージ
（変数Nの値により条件分岐）

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

- ループ長Nの値が10である場合とそれ以外の場合とによる条件分岐が生成される。
- Nの値が10である場合には、IFブロック内のIループの上限が明示的に10とされ、フルアンローリングなどの他の最適化が促進される可能性がある。
- ループ長Nの値が10でない場合はELSEブロック内のIループの上限がNのままであるループが実行される。

■ CLONE最適化の指示行 [1/2]

- **!OCL CLONE(*var*==*n1*[,*n2*]...)**
  - DOループ単位、および配列代入文単位のみで指定可能である。
  - 本指示行は-O3オプションが有効な場合に意味がある。
  - 条件式は第1引数の変数*var*と、第2引数以降で指定した値*n1*[,*n2*]...の等式とする。
    - *var*は整数型の変数である(*1)。種別型パラメタは1、2、4、または8である。
      (*1) 整数型であっても配列要素の場合や構造体成分の場合等、あるいはALLOCATABLE属性等の属性が付く場合は指定できません。詳細は「Fortran使用手引書」を参照して下さい。
    - *n1*[,*n2*]...は-9223372036854775808から9223372036854775807までの10進数または名前付き定数である(*2)。
      (*2) 値に関わらず数字列の後に下線“_”を用いた値の種別型パラメタは指定できない。
  - 値*n1*[,*n2*]...の指定方法
    - コンマ“,”による列挙　　例：!OCL CLONE(var==3,4,5)
    - コロン“:”による値の範囲指定　　例：!OCL CLONE(var==3:5)
    - コンマ“,”とコロン“:”の組み合わせ　　例：!OCL CLONE(var==1,3:5)
  - 値*n1*[,*n2*]...に指定できる個数の制限
    - 一つの!OCL CLONE指示行で指定できる値の個数の上限は20個である。
    - 21個以上の値を指定した場合、21個目以降の指定は無効となる。
      例：指定!OCL CLONE(var==11:40)は30個の値を指定しているため個数の制約を受ける。21個目以降の指定（本例では値31から値40）は無効となり、上記の指定は!OCL CLONE(var==11:30)と同等に扱われる。

■ CLONE最適化の指示行 [2/2]

■ 注意事項

⚠ CLONE最適化は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る。

⚠ CLONE最適化は生成された条件節内のループにおいて、CLONE最適化指示行に指定した変数*var*の値が不変であるものとした最適化を行う。そのため対象ループ内で変数*var*が更新される場合(*1)、実行結果が保証されない。このような指定は行わないようにして下さい。

(*1) CLONE最適化指示行の誤った指定であることを知らせる如何なる警告メッセージも表示されず、CLONE最適化が誤って適用される。

- 右の図の誤った指定例1は、`M=>N` により、ポインタMのターゲットがNと指定されており、`!OCL CLONE(N==10)`でNに指定された値10と異なる値5がループ内でMに設定されている。そのためループ内で変数Nが更新される。この場合結果が保証されない。

- 右の図の誤った指定例2は、`!OCL CLONE(N==10)`で指定した変数Nをループ内でのサブルーチンの引数にしている。コンパイラはサブルーチンSUB内で変数Nが更新されるかどうか分からないため、ループ内で変数Nが更新される可能性があると判断される。

誤った指定例1

```
  INTEGER,DIMENSION(32)::A
  INTEGER,TARGET::N
  INTEGER,POINTER::M
  M=>N
!OCL CLONE(N==10)
  DO I=1,32
    M=5
    A(I) = N
  ENDDO
```

誤った指定例2

```
  INTEGER::N
!OCL CLONE(N==10)
  DO I=1,32
    CALL SUB(N)
  ENDDO
```

■ CLONE最適化によるソフトウェアパイプライニング促進ループ例（asis）

以下の図に、CLONE最適化によるソフトウェアパイプライニング促進の効果を見るループ例を示す。

- 配列A、B、およびCはサブルーチンの引数であり、各配列の寸法を定義する変数NIおよびNJもサブルーチンの引数である。
- 変数NIおよびNJの値は主プログラム単位（MAIN）においてNI=64、NJ=200と定義されているが、コンパイラは、このサブルーチン（SUB_ASIS）内でのNIおよびNJの値は不明であると判断する。最内ループのIループ長NIは実際には64だがコンパイラには分からない。

次のスライドで本ループ例のコンパイルリストを示す。

```
PROGRAM MAIN
  INTEGER,PARAMETER::NI=64, NJ=200
```

```
SUBROUTINE SUB_ASIS(A,B,C,NI,NJ)
  INTEGER::NI,NJ
  DOUBLE PRECISION::A(NI,NJ),B(NI,NJ),C(0:6,NJ)
 (中略)
  DO J=1,NJ
    C0=C(0,J);C1=C(1,J);C2=C(2,J);C3=C(3,J);C4=C(4,J);C5=C(5,J);C6=C(6,J)
    DO I=1,NI
      X = B(I,J)
      A(I,J) = C0+X*(C1+X*(C2+X*(C3+X*(C4+X*(C5+X*(C6+X))))))
    ENDDO
  ENDDO
```

■ ループ例（asis）のコンパイルリスト

最適化オプションは-Kfastとする。本ループ例のコンパイルリストを示す。

- 最内のIループに対してメッセージjwd8205が表示され、本例ではIループの繰返し数が160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る。しかし、実際にはIループの繰返し数NIは64であり、ソフトウェアパイプライニングが適用されないループが実行される。
- Jループに対してソフトウェアパイプライニングの表示はない。

```
534    2          DO J=1,NJ
535    2            C0=C(0,J);（中略）;C6=C(6,J)
536    3    2v      DO I=1,NI
537    3    2v        X = B(I,J)
538    3    2v        A(I,J)= C0+X*(C1+X*(C2+X*(C3+X &
539    3                &   *(C4+X*(C5+X*(C6+X))))))
540    3    2v      ENDDO
541    2          ENDDO
```

Jループの<<< Loop-information >>>

```
（ソフトウェアパイプライニングの表示なし）
<<<      （以下、省略）
```

Iループの<<< Loop-information >>>

```
<<<  [OPTIMIZATION]
<<<    SIMD(VL: 8)
<<<    SOFTWARE PIPELINING(IPC:
1.57, ITR: 160, MVE: 5, POL: S)
<<<      （以下、省略）
```

Jループのメッセージ

```
（ソフトウェアパイプライニング適用のメッセージなし）
```

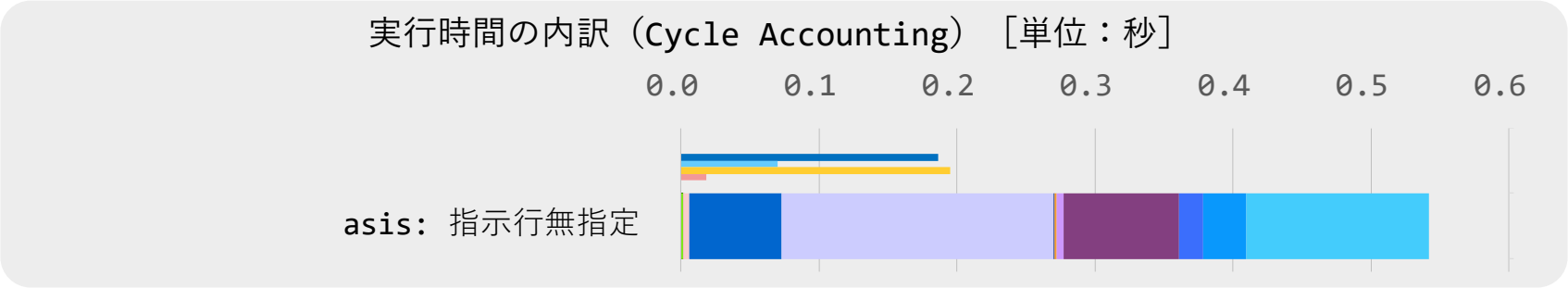
Iループのメッセージ

```
jwd6001s-i  "～", line 536: このDOループをSIMD化しました。(名前:I)
jwd8204o-i  "～", line 536: 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  "～", line 536: ループの繰返し数が160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。
```

■ ループ例（asis）の実行結果

本ループ例を逐次実行し、CPU性能解析レポートを採取する（計測のためのループの繰り返し数NLOOPを50000とする）。下の図に、CPU性能解析レポートから実行時間の内訳（Cycle Accounting）を示す。

■ ソフトウェアパイプライニングが動作していないため命令スケジューリングが悪く、浮動小数点演算待ちの時間（ ）が多くを占めていることが確認される。



■ ループ例（tune）のコンパイルリスト [1/4]

以下の図に!OCL CLONE(NI==64)を指定したコンパイルリストを示す（最適化オプション-Kfast）。

- Jループに対する<<< Loop-information >>>内に「CLONE」が表示され、メッセージjwd8324「ループにCLONE最適化を適用しました」が表示される。
- （次スライドへ続く）

```
1262    1       !OCL CLONE(NI==64)
1263    2       DO J=1,NJ
1264    2         C0=C(0,J);（中略）;C6=C(6,J)
1265    3    fv   DO I=1,NI
1266    3    fv     X = B(I,J)
1267    3    fv     A(I,J)= C0+X*(C1+X*(C2+X*(C3+X &
1268    3              &   *(C4+X*(C5+X*(C6+X))))))
1269    3    fv   ENDDO
1270    2       ENDDO
```

Jループの<<< Loop-information >>>

```
<<<  [OPTIMIZATION]
<<<    SOFTWARE PIPELINING(IPC:
1.50, ITR: 24, MVE: 2, POL: S)
<<<    CLONE
<<<      （以下、省略）
```

Iループの<<< Loop-information >>>

```
<<<  [OPTIMIZATION]
<<<    SIMD(VL: 8)
<<<    SOFTWARE PIPELINING(IPC:
1.57, ITR: 160, MVE: 5, POL: S)
<<<    FULL UNROLLING
<<<      （以下、省略）
```

Jループのメッセージ

```
jwd8324o-i  "～", line 1263: ループにCLONE最適化を適用しました。
 （以下、次スライドで示す）
```

■ ループ例（tune）のコンパイルリスト [2/4]

- Jループの<<< Loop-information >>>内に「SOFTWARE PIPELINING」が表示され、Jループのメッセージ欄にjwd8205「ループの繰返し数が24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る」が表示される。
  - 本例では外側Jループのループ長NJは200としているため、ソフトウェアパイプライニングを適用したループが実際選択される。
- （次スライドへ続く）

```
1262    1        !OCL CLONE(NI==64)
1263    2        DO J=1,NJ
1264    2          C0=C(0,J);（中略）;C6=C(6,J)
1265    3    fv    DO I=1,NI
1266    3    fv      X = B(I,J)
1267    3    fv      A(I,J)= C0+X*(C1+X*(C2+X*(C3+X &
1268    3              &   *(C4+X*(C5+X*(C6+X))))))
1269    3    fv    ENDDO
1270    2        ENDDO
```

Jループの<<< Loop-information >>>

```
<<<  [OPTIMIZATION]
<<<    SOFTWARE PIPELINING(IPC:
1.50, ITR: 24, MVE: 2, POL: S)
<<<    CLONE
<<<     （以下、省略）
```

Iループの<<< Loop-information >>>

```
<<<  [OPTIMIZATION]
<<<    SIMD(VL: 8)
<<<    SOFTWARE PIPELINING(IPC:
1.57, ITR: 160, MVE: 5, POL: S)
<<<    FULL UNROLLING
<<<     （以下、省略）
```

Jループのメッセージ

```
jwd8324o-i  "～", line 1263: ループにCLONE最適化を適用しました。
jwd8204o-i  "～", line 1263: 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  "～", line 1263: ループの繰返し数が24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。
（1263行目に対するメッセージは以上）
```

■ ループ例（tune）のコンパイルリスト [3/4]

- 内側Iループの**DO**文の左の記号「**f**」およびメッセージ**jwd8203**により、条件 **NI==64** が真の時に実行されるループではIループがフルアンローリングされることが示されている。
- （次スライドへ続く）

```
1262   1       !OCL CLONE(NI==64)
1263   2       DO J=1,NJ
1264   2         C0=C(0,J);（中略）;C6=C(6,J)
1265   3   fv    DO I=1,NI
1266   3   fv      X = B(I,J)
1267   3   fv      A(I,J)= C0+X*(C1+X*(C2+X*(C3+X &
1268   3             &   *(C4+X*(C5+X*(C6+X))))))
1269   3   fv    ENDDO
1270   2       ENDDO
```

Jループの<<< Loop-information >>>

```
<<<  [OPTIMIZATION]
<<<    SOFTWARE PIPELINING(IPC:
1.50, ITR: 24, MVE: 2, POL: S)
<<<    CLONE
<<<     （以下、省略）
```

Iループの<<< Loop-information >>>

```
<<<  [OPTIMIZATION]
<<<    SIMD(VL: 8)
<<<    SOFTWARE PIPELINING(IPC:
1.57, ITR: 160, MVE: 5, POL: S)
<<<    FULL UNROLLING
<<<     （以下、省略）
```

Iループのメッセージ

```
jwd6001s-i  "～", line 1265: このDOループをSIMD化しました。(名前:I)
jwd8203o-i  "～", line 1265: ループをフルアンローリングしました。
 （以下、次スライドで示す）
```

■ ループ例（tune）のコンパイルリスト [4/4]

■ なお、Iループに対して表示されるソフトウェアパイプライニング適用を告げる各種メッセージは条件 NI==64 が偽の時に実行されるループに対するものである。

```
1262    1       !OCL CLONE(NI==64)
1263    2       DO J=1,NJ
1264    2         C0=C(0,J);（中略）;C6=C(6,J)
1265    3    fv   DO I=1,NI
1266    3    fv     X = B(I,J)
1267    3    fv     A(I,J)= C0+X*(C1+X*(C2+X*(C3+X &
1268    3              &   *(C4+X*(C5+X*(C6+X))))))
1269    3    fv   ENDDO
1270    2       ENDDO
```

Jループの<<< Loop-information >>>

```
<<<  [OPTIMIZATION]
<<<    SOFTWARE PIPELINING(IPC:
1.50, ITR: 24, MVE: 2, POL: S)
<<<    CLONE
<<<     （以下、省略）
```

Iループの<<< Loop-information >>>

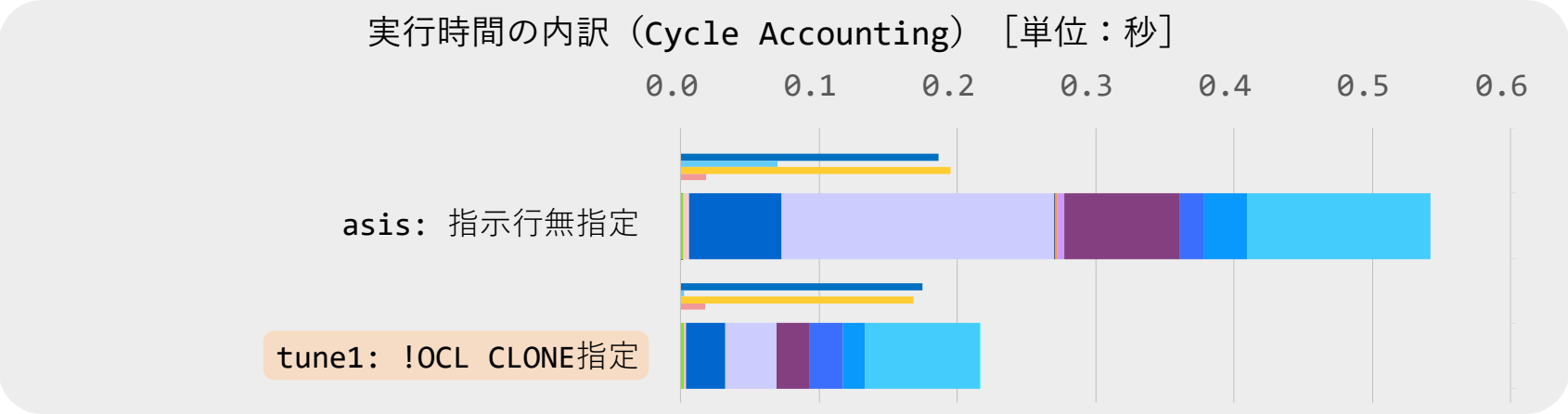
```
<<<  [OPTIMIZATION]
<<<    SIMD(VL: 8)
<<<    SOFTWARE PIPELINING(IPC:
1.57, ITR: 160, MVE: 5, POL: S)
<<<    FULL UNROLLING
<<<     （以下、省略）
```

Iループのメッセージ

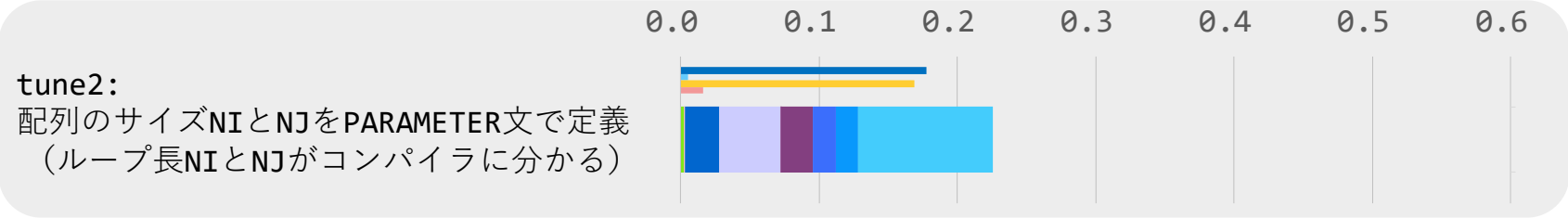
（前スライドからの続き）
jwd8204o-i "～", line 1265: ループにソフトウェアパイプライニングを適用しました。
jwd8205o-i "～", line 1265: ループの繰返し数が160回以上の時、ソフトウェアパイプライニングを適用したループが実行時に選択されます。

■ ループ例（asisとtune）の実行結果

■ CLONE最適化を適用したループの実行時間の内訳（Cycle Accounting）を以下の2番目の棒グラフ群（tune1）に示す。CLONE最適化により外側ループにソフトウェアパイプライニングが適用されたことで命令スケジューリングが改善し、浮動小数点演算待ちの時間（■）が改善されていることが確認される。



■ なお、参考データとして、配列のサイズNIとNJがPARAMETER文で定義され、ループ長NIとNJがコンパイラに分かるループとした場合の実行時間の内訳を下の棒グラフ群（tune2）に示す。CLONE最適化の結果tune1はtune2と同等であることが確認される。



- 各種最適化手法
  - ソフトウェアパイプライニングの促進

本節の内容

- 【第一部からの抜粋】ソフトウェアパイプライニングの概念説明
- 外側ループでのソフトウェアパイプライニング
- レジスタ不足の回避

■ レジスタ不足のメッセージとループ例（asis）

ループ中で使用されるレジスタの数が多い場合、以下のようなメッセージjwd8665またはjwd8666が表示され、ソフトウェアパイプライニングが適用されない。

```
jwd8665o-i ： 整数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。
```

```
jwd8666o-i ： 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。
```

本小節では、上記メッセージが示すレジスタ不足によりソフトウェアパイプライニングが適用されないループに対して、レジスタ不足を回避しソフトウェアパイプライニングを促進する例を示す。本小節では以下の図に示すループを例とする。

ループ例

```
INTEGER,PARAMETER::NI=9216
DOUBLE PRECISION::A0(NI),A1(NI),B0(NI),B1(NI)
DO I=1,NI
  X = B0(I)
  Y = B1(I)
  XY = 2D0 / (X+Y)
  X = X * XY
  Y = Y * XY
  A0(I) = X/(S0+X/(S1+X/(S2+X/(S3+X/(S4+X/(S5+X/(S6+X/(S7)))))))
  A1(I) = Y/(T0+Y/(T1+Y/(T2+Y/(T3+Y/(T4+Y/(T5+Y/(T6+Y/(T7)))))))
ENDDO
```

最適化のコンパイルオプションは-Kfastとし、逐次実行する（計測のためのループの繰り返し数NLOOPを10000とする）。本小節内の以降のスライドで示す改善策で使用される指示行を有効にするため-Koclも指定する（コンパイルリスト出力のため、-Koptmsg=guide -Nlst=t も指定する）。

■ ループ例（asis）のコンパイルリストおよび実行結果

■ 本ループ例のコンパイルリストを以下の図に示す。メッセージjwd8666が表示され、ソフトウェアパイプライニングが適用されない。

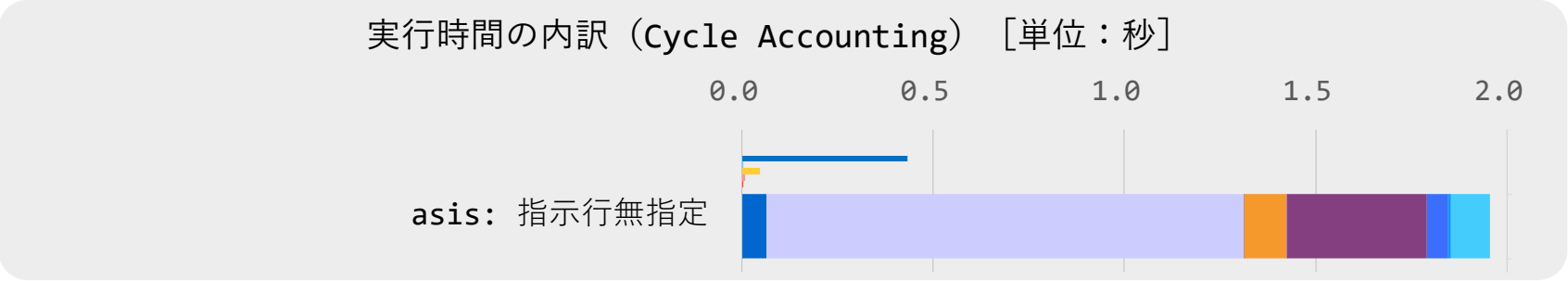
```
136    2    v    DO I=1,NI
137    2    v      X = B0(I)
138    2    v      Y = B1(I)
139    2    v      XY = 2D0 / (X+Y)
140    2    v      X = X * XY
141    2    v      Y = Y * XY
142    2    v      A0(I) = X/(S0+X/(S1+X/(S2+X/(S3+X/(S4+X/(S5+X/(S6+X/(S7))))))))
143    2    v      A1(I) = Y/(T0+Y/(T1+Y/(T2+Y/(T3+Y/(T4+Y/(T5+Y/(T6+Y/(T7))))))))
144    2    v    ENDDO
```

Iループの<<< Loop-information >>>

```
<<<  [OPTIMIZATION]
<<<    SIMD(VL: 8)
<<<    PREFETCH(HARD) Expected by compiler :
<<<      B1, B0, A1, A0
```

jwd8666o-i "～", line 136: 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。
[ガイダンス]（次スライドで示す）

■ 本ループ例の実行結果（CPU性能解析レポート）から実行時間の内訳（Cycle Accounting）を以下の図に示す。ソフトウェアパイプライニングが適用されないことにより命令スケジューリングが悪いため、浮動小数点演算待ち（■）が実行時間の多くを占めていることが確認される。



■ ループ例（asis）のコンパイルリストにおけるガイダンスメッセージ

コンパイル時に-Koptmsg=guideを指定した場合、メッセージjwd8666には図に示すガイダンスメッセージが表示される。

```
jwd8666o-i  "～", line 136: 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。
  [ガイダンス]
    以下のいずれかの対処をする。
      (1) 翻訳時オプション-Kswp_strong(ソフトウェアパイプライニング適用条件を緩和する)を指定する。
      (2) 該当ループに最適化指示子SWP(ソフトウェアパイプライニング適用促進を指示する)を指定する。
      (3) 翻訳時オプション-Kswp_freg_rate=N(ソフトウェアパイプライニング機能における、レジスタ数の条件を変更する)を指定する。
      (4) 該当ループに最適化指示子SWP_FREG_RATE(N)(ソフトウェアパイプライニング機能における、レジスタ数の条件変更を指示する)を指定する。
      (5) (1)、(2)、(3)または(4)で効果が得られなかった場合は、ループを分割するようにプログラムを変更し、1ループ内の浮動小数点レジスタの使用数を減らす。
```

以降のスライドでは、ガイダンスに従い提示された対処法を施行した結果を示す。

■ 列挙された対処法の項番(1)に示されたオプション-Kswp_strong(*1,*2)は項番(2)に示された指示行!OCL SWP(*2)と同等な効果である(*3)。そこで、まず項番(2)の指示行!OCL SWPの指定を試みる。

(*1) -Kswp_strongオプションは、ソフトウェアパイプライニング適用の条件を緩和し、ソフトウェアパイプライニングを促進することを指示する。
(*2) これらのオプションおよび指示行については中級編第一部を参照して下さい。
(*3) 「プログラミングガイド Fortran編」参照。

■ 指示行!OCL SWPによるソフトウェアパイプライニングの促進（tune1）

本ループ例で!OCL SWPの指定をした場合のコンパイルリストを以下の図に示す。メッセージjwd8666は表示されなくなり、メッセージjwd8204が示すようにソフトウェアパイプライニングが適用されたことが確認される。本例ではループ長NIは9216であり、jwd8205が示す「しきい値」より大きいためソフトウェアパイプライニングが実際適用される。

Iループの<<< Loop-information >>>

```
<<<  [OPTIMIZATION]
<<<    SIMD(VL: 8)
<<<    SOFTWARE PIPELINING(IPC: 1.27, ITR:
64, MVE: 4, POL: L)
<<<    PREFETCH(HARD) Expected by
compiler :
<<<      A0, A1, B1, B0
```

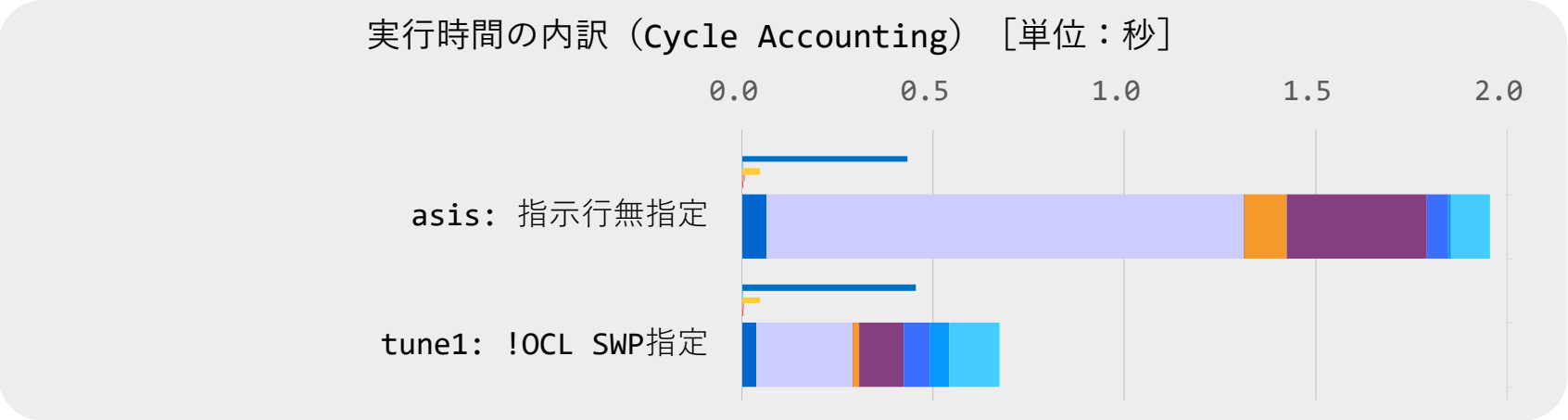
```
173    1          !OCL SWP
174    2    v       DO I=1,NI
175    2    v         X = B0(I)
176    2    v         Y = B1(I)
177    2    v         XY = 2D0 / (X+Y)
178    2    v         X = X * XY
179    2    v         Y = Y * XY
180    2    v         A0(I) = X/(S0+X/(S1+X/(S2+X/(S3+X/(S4+X/(S5+X/(S6+X/(S7)))))))
181    2    v         A1(I) = Y/(T0+Y/(T1+Y/(T2+Y/(T3+Y/(T4+Y/(T5+Y/(T6+Y/(T7)))))))
182    2    v       ENDDO
```

Iループのメッセージ

```
jwd6001s-i  "～", line 174: このDOループをSIMD化しました。(名前:I)
jwd8204o-i  "～", line 174: 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  "～", line 174: ループの繰返し数が64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。
（以下、省略）
```

■ tune1とasisの実行結果の比較

本ループ例の実行結果（CPU性能解析レポート）から実行時間の内訳（Cycle Accounting）を以下の図一番下にtune1として示す。tune1はソフトウェアパイプライニングが適用されたことにより、指示行無指定のasisと比較して命令スケジューリングが改善され、浮動小数点演算待ち（ ）が改善されていることが確認される。



■ ループ例（asis）のコンパイルリストにおけるガイダンスメッセージ（再掲抜粋）

次に、レジスタ不足が生じる元のループ例のコンパイルリスト（下の図に抜粋を再掲）に表示されたガイダンスに従い、対処法の項番(3)または項番(4)の指定を検討する。

```
jwd8666o-i  "～", line 136: 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。
  [ガイダンス] 以下のいずれかの対処をする。
      (1) （省略）
      (2) （省略）
      (3) 翻訳時オプション-Kswp_freg_rate=N(ソフトウェアパイプライニング機能における、レジスタ数の条件を変更する)を指定する。
      (4) 該当ループに最適化指示子SWP_FREG_RATE(N)(ソフトウェアパイプライニング機能における、レジスタ数の条件変更を指示する)を指定する。
      (5) （省略）
```

■ 項番(4)の指示行!OCL SWP_FREG_RATEは、項番(3)のオプション-Kswp_freg_rateと同等な機能(*4)である。そこで、以降では項番(4)の方法を試行し、その結果を示す。

(*4) -Kswp_freg_rate=*N* および !OCL SWP_FREG_RATE(*N*)

- 浮動小数点レジスタおよびSVEのベクトルレジスタについて、ソフトウェアパイプライニングで使用するレジスタ数の条件を変更することを指示する。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る。
- 引数*N*は使用可能なレジスタ数の割合（百分率）を表す1～1000までの整数値である。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より大きな値を*N*に指定することで、ソフトウェアパイプライニングが適用できることがある。

⚠ 本オプションまたは指示行を指定することで、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る。

- 本オプションおよび指示行は中級編第一部で説明していますので、これ以上の説明は省略します。

■ 指示行!OCL SWP_FREG_RATEによるソフトウェアパイプライニングの促進（tune2）

本ループ例で指示行!OCL SWP_FREG_RATE(116)の指定をした場合のコンパイルリストを以下の図に示す。メッセージjwd8204が示すようにソフトウェアパイプライニングが適用されたことが確認される。

Iループの<<< Loop-information >>>

```
<<<  [OPTIMIZATION]
<<<    SIMD(VL: 8)
<<<    SOFTWARE PIPELINING(IPC: 1.34, ITR: 80,
MVE: 5, POL: L)
<<<    PREFETCH(HARD) Expected by compiler :
<<<      A1, A0, B1, B0
<<<    SPILLS :
<<<      GENERAL    : SPILL 0  FILL 0
<<<      SIMD&FP    : SPILL 0  FILL 0
<<<      SCALABLE   : SPILL 1  FILL 11
<<<      PREDICATE  : SPILL 0  FILL 0
```

```
667    1          !OCL SWP_FREG_RATE(116)
668    2    v       DO I=1,NI
669    2    v         X = B0(I)
670    2    v         Y = B1(I)
671    2    v         XY = 2D0 / (X+Y)
672    2    v         X = X * XY
673    2    v         Y = Y * XY
674    2    v         A0(I) = X/(S0+X/(S1+X/(S2+X/(S3+X/(S4+X/(S5+X/(S6+X/(S7)))))))))
675    2    v         A1(I) = Y/(T0+Y/(T1+Y/(T2+Y/(T3+Y/(T4+Y/(T5+Y/(T6+Y/(T7)))))))))
676    2    v       ENDDO
```

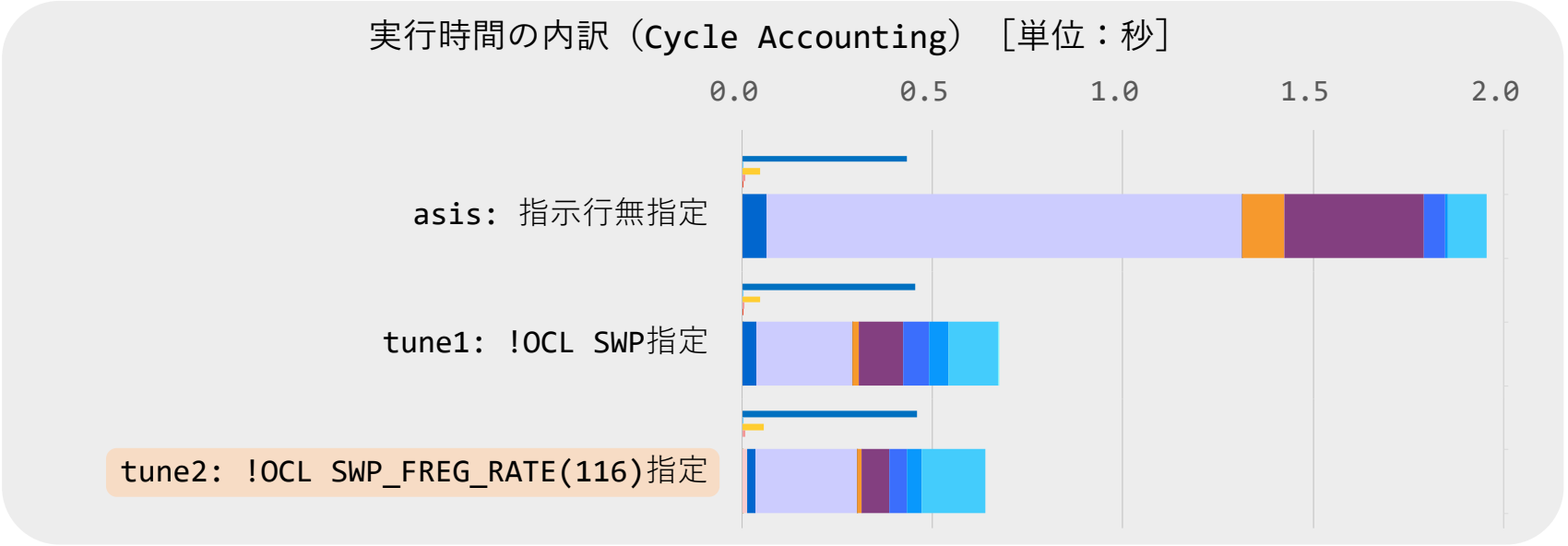
Iループのメッセージ

```
jwd6001s-i  "～", line 668: このDOループをSIMD化しました。(名前:I)
jwd8204o-i  "～", line 668: 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  "～", line 668: ループの繰返し数が80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。
（以下、省略）
```

■ tune2とasisの実行結果の比較

本ループ例の実行結果（CPU性能解析レポート）から実行時間の内訳（Cycle Accounting）を以下の図一番下にtune2として示す。参考としてこれまで適用した改善策（!OCL SWP指定）の結果を併せて示す（上から二番目）。

- tune2はソフトウェアパイプライニングが適用されたことにより、指示行無指定のasisと比較して浮動小数点演算待ち（■）が改善されていることが確認される。



■ ループ例（asis）のコンパイルリストにおけるガイダンスメッセージ（再掲抜粋）

次に、レジスタ不足が生じる元のループ例のコンパイルリスト（下の図に抜粋を再掲）に表示されたガイダンスに従い、対処法の項番(5)、ループ分割、を試みる。

```
jwd8666o-i  "～", line 136: 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。
  [ガイダンス] 以下のいずれかの対処をする。
     (1) （省略）
     (2) （省略）
     (3) （省略）
     (4) （省略）
     (5) (1)、(2)、(3)または(4)で効果が得られなかった場合は、ループを分割するようにプログラムを変更し、1ループ内の浮動小数点レジスタの使用数を減らす。
```

■ ループ分割を手作業で行う方法と指示行で分割する方法があるが、以降では、コンパイラが自動でループ分割する指示行を指定する方法を示す。適用する指示行は次のスライドで示す。

■ 自動ループ分割の指示行

以下に示す指示行により、指定されたループに対してコンパイラが自動でループを分割する最適化（自動ループ分割）を行うことを指示することができる。

■ 指示行（DOループ単位、および文単位のみで指定可能）

```
!OCL LOOP_FISSION_TARGET          または
!OCL LOOP_FISSION_TARGET(CL)      または
!OCL LOOP_FISSION_TARGET(LS)
```

■ 注意事項

■ 本指示行は-Kloop_fissionオプション(*1)が有効な場合に意味がある。

(*1) -Kloop_fissionオプションは-O2オプション以上が有効な場合、有効となる。
-O0または-O1オプションが有効な場合、デフォルトは-Kloop_nofissionである。

■ 指示行を有効にするためには-Koclオプションを指定する必要がある。

■ 引数CLまたはLSについて

引数のCLまたはLSでループ分割のアルゴリズムを指定する。CLおよびLSが省略された場合、CLを指定したものとみなす。

■ !OCL LOOP_FISSION_TARGET(CL)

- クラスタリングアルゴリズムでループ分割を行う。
- 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する。

■ !OCL LOOP_FISSION_TARGET(LS)

- 局所探索アルゴリズムでループ分割を行う。
- ソフトウェアパイプライニングの促進を優先する。

■ 自動ループ分割指示行によるソフトウェアパイプライニング促進（tune3） [1/3]

本ループ例で指示行!OCL LOOP_FISSION_TARGET(CL)の指定をした場合のコンパイルリストを以下の図に示す。

■ <<< Loop-information >>>欄に「FISSION」と表示される。「FISSION」に続いて本例では「(num: 2)」と表示され、メッセージ「jwd8212 ： ループを2分割しました」が表示される。これらの表示はループが二つに分割されたことを意味する。

■ （次スライドへ続く）

```
4999   1          !OCL LOOP_FISSION_TARGET(CL)
5000   2   v        DO I=1,NI
5001   2   v          X = B0(I)
5002   2   v          Y = B1(I)
5003   2   v          XY = 2D0 / (X+Y)
5004   2   v          X = X * XY
5005   2   v          Y = Y * XY
5006   2   v          A0(I) = X/(S0+X/(S1+X/(S2+X/(S3
5007   2   v          A1(I) = Y/(T0+Y/(T1+Y/(T2+Y/(T3
5008   2   v        ENDDO
```

Iループの<<< Loop-information >>>

```
<<<  [OPTIMIZATION]
<<<    FISSION(num: 2)
<<<    SIMD(VL: 8)
<<<    SOFTWARE PIPELINING(IPC: 1.25, ITR:
128, MVE: 3, POL: L)
<<<    PREFETCH(HARD) Expected by
compiler :
<<<      A1, B0, B1, A0, (unknown)
<<<    SPILLS :
<<<      GENERAL    : SPILL 0  FILL 0
<<<      SIMD&FP    : SPILL 0  FILL 0
<<<      SCALABLE   : SPILL 0  FILL 24
<<<      PREDICATE  : SPILL 0  FILL 0
```

Iループのメッセージ

```
jwd8212o-i  "～", line 5000: ループを2分割しました。
（その他のループ分割関連メッセージは次のスライドで説明する）
```

（ループ分割関連以外のメッセージは省略）

■ 自動ループ分割指示行によるソフトウェアパイプライニング促進（tune3） [2/3]

■ コンパイルリストのメッセージ欄には、本例ではさらに、「jwd8217 ： ループ分割で使用するための一時的な配列を1個生成しました」が表示される。これは、分割された二つのループ間でデータを受け渡しするための一時配列が本例では1個生成されたことを意味する。

```
jwd8217o-i   "～", line 5000: ループ分割で使用するための一時的な配列を1個生成しました。
```

■ 分割された二つのループはメッセージ欄で「loop-id 1」と「loop-id 2」で指し示されている。

- メッセージjwd8219により、元のループに含まれる命令数に対する分割後のループに含まれる命令数の割合が表示される。

```
jwd8218o-i   "～", line 5000, loop-id 1: ループ分割に伴い、1個の一時的な配列に後続のループで参照するデータを格納します。
jwd8219o-i   "～", line 5000, loop-id 1: 分割後のループに含まれる命令数は、元のループの命令数の53%です。
（以下、次スライドで説明）
```

```
jwd8219o-i   "～", line 5000, loop-id 2: 分割後のループに含まれる命令数は、元のループの命令数の46%です。
（以下、次スライドで説明）
```

■ （次スライドへ続く）

■ 自動ループ分割指示行によるソフトウェアパイプライニング促進（tune3） [3/3]

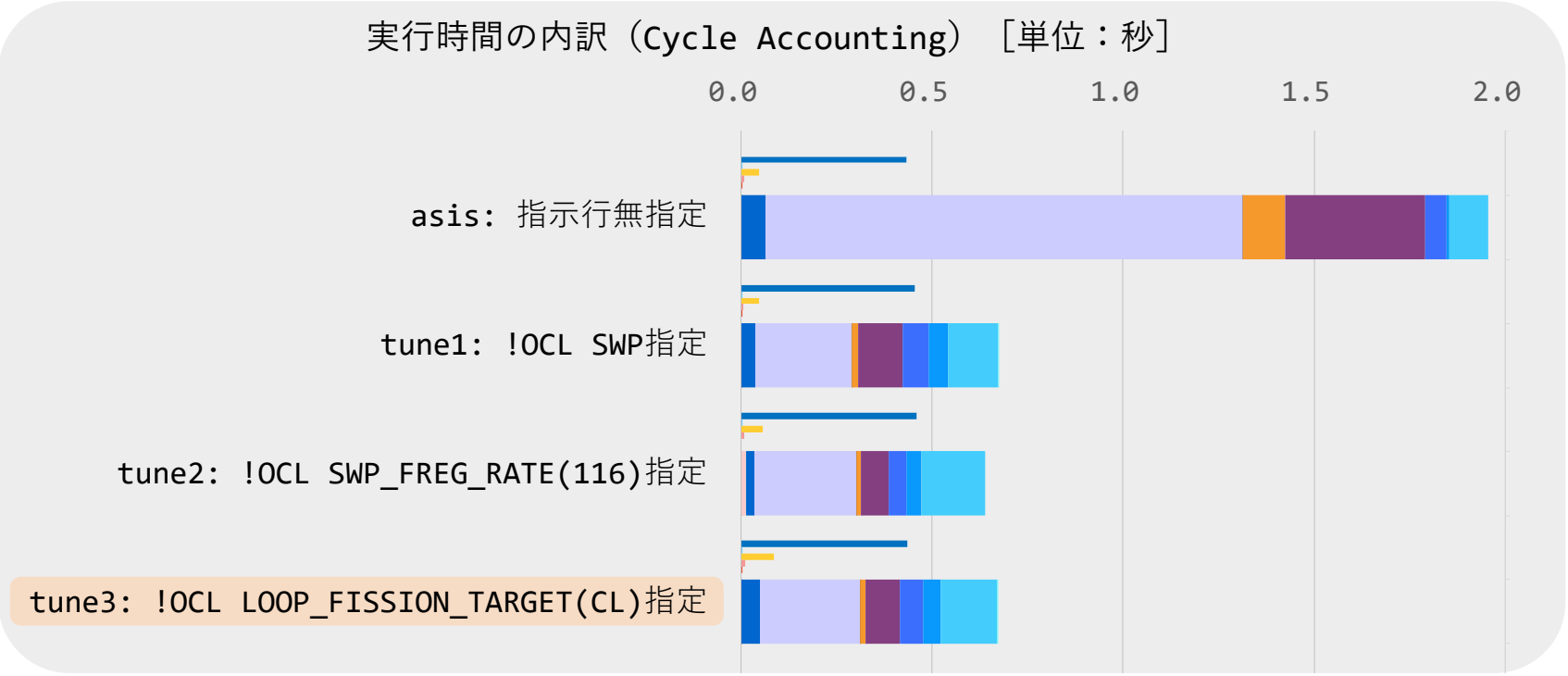
- 自動ループ分割後の二つのループはともにjwd8204が示すようにソフトウェアパイプライニングが適用されていることが確認される。自動ループ分割によりソフトウェアパイプライニングが適用された。

```
（中略）
jwd8204o-i  "～", line 5000, loop-id 1: 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  "～", line 5000, loop-id 1: ループの繰返し数が128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。
```

```
（中略）
jwd8204o-i  "～", line 5000, loop-id 2: 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  "～", line 5000, loop-id 2: ループの繰返し数が128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。
```

■ tune3とasisの実行結果の比較

- 本ループ例の実行結果（CPU性能解析レポート）から実行時間の内訳（Cycle Accounting）を以下の図一番下にtune3として示す。tune3はソフトウェアパイプライニングが適用されたことにより、指示行無指定のasisと比較して浮動小数点演算待ち（ ）が改善されていることが確認される。
- 参考としてこれまで適用した改善策（tune1: !OCL SWP指定、tune2: !OCL SWP_FREG_RATE指定）の結果を併せて示す（二番目と三番目）。本ループ例に対しては、これまで示した三つの改善策はほぼ同等な効果となった。



内容

- 「富岳」の概略
- プロファイラ（CPU性能解析レポート）
- 各種最適化手法
  - 演算の効率化
  - ソフトウェアパイプライニングの促進
  - SIMD化の促進（ロード命令・ストア命令の効率化、演算の効率化）
  - キャッシュチューニングのための基本事項
- 【付録】

- 各種最適化手法
  - SIMD化の促進

本節の内容

SIMD化も演算を効率化するための最適化手法ですが、特に重要な手法であるため、本節で切り出して取り上げます。中級編第一部ではSIMD化の概念の説明、コンパイルオプションと指示行、およびSIMD化促進のループ例をいくつか説明しています。この中級編第三部では第一部の内容を補足することを目的とします。最初の小節では、本中級編第一部の未受講者を想定し、SIMD化の概念の説明をします。その後の一つの小節ではSIMD化を促進するための手法を一つ紹介します。

- 【第一部からの抜粋】SIMD化の概念説明
- SIMD化前のフルアンローリング（ロード命令・ストア命令の効率化、演算の効率化）

- 各種最適化手法
  - SIMD化の促進
    - 【第一部からの抜粋】SIMD化の概念説明

本小節の内容

本小節では、SIMD化の概念について説明します。最初に、IF文を含まないDOループのSIMD化の動作イメージを説明し、次に、IF文を含むDOループのSIMD化の動作イメージを説明します。なお、説明の便宜上FortranのDOループを用いて説明しています。

- IF文を含まないDOループのSIMD化の動作イメージ
- IF文を含むDOループのSIMD化の動作イメージ

■ 動作説明のためのループ例と簡易モデル

■ ループ例

右の図に示すDOループを例に、SIMD化されたDOループの動作イメージを説明する。

ループ例

```
DO I=1,4
  B(I) = A(I) + 10.0
ENDDO
```

■ 簡易モデル

- SIMD命令は1命令で2要素を処理するとする(*1)。

  (*1) 実際の演算器では一つのSIMD命令で倍精度実数の場合8要素を処理できる。

- 以降のスライドでは、通常の命令に対応するSIMD命令の呼び名・表記を以下のようにする。

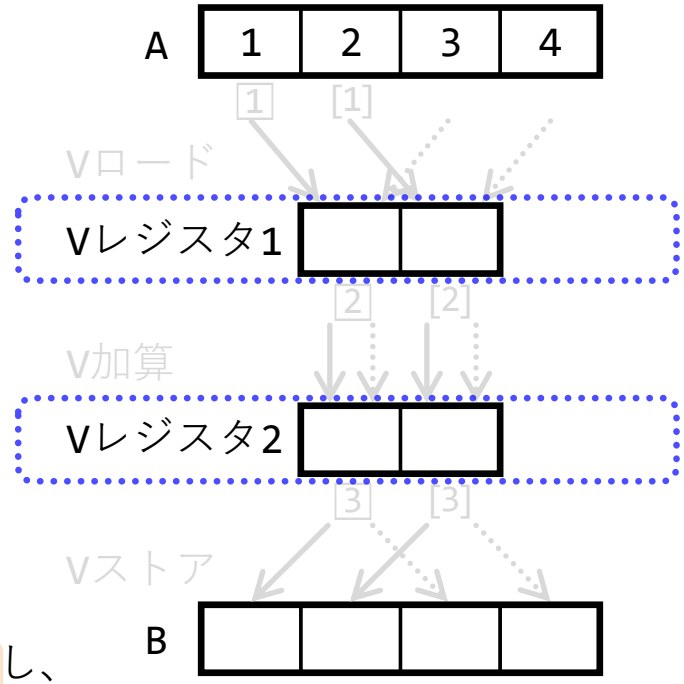
命令	対応するSIMD命令の呼び名	表記
ロード	ベクトルロード	「Vロード」
加算	ベクトル加算	「V加算」
ストア	ベクトルストア	「Vストア」

■ SIMD命令で使用するレジスタ

- SIMD命令では、一つのレジスタで2要素※を持つベクトルレジスタを使用する。

※ 簡易モデル

本例で使用する二つのベクトルレジスタを、
「Vレジスタ1」、「Vレジスタ2」と表す。



■ SIMD化されたDOループのイメージ [1/2]

- SIMD命令は2要素を一度に（同時に）処理するため、
  以下の図のループは [0] に示すように一つ飛びに反復し、
  1回目の反復でI=1とI=2の2要素を処理する。

■ （次スライドへ続く）

SIMD化されたDOループのイメージ

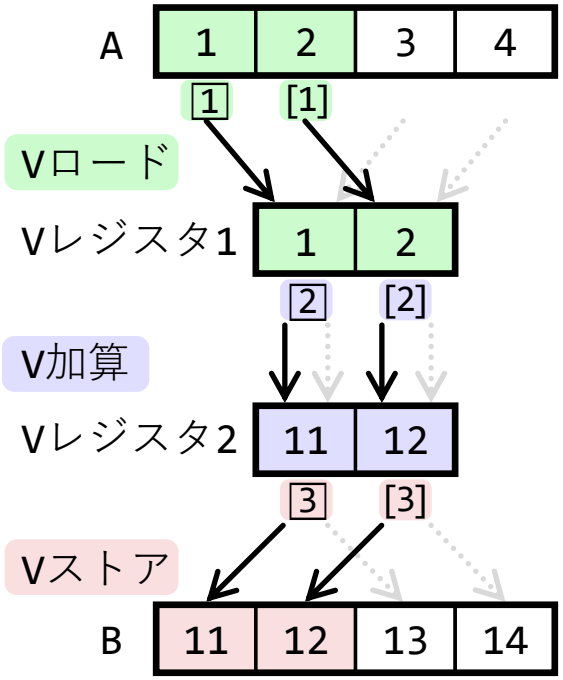
```
DO I=1,4,2  [0]
```

Vロード	Vレジスタ1(1)=A(I) [1]	Vレジスタ1(2)=A(I+1) [1]
V加算	Vレジスタ2(1)=Vレジスタ1(1)+10.0 [2]	Vレジスタ2(2)=Vレジスタ1(2)+10.0 [2]
Vストア	B(I)=Vレジスタ2(1) [3]	B(I+1)=Vレジスタ2(2) [3]

```
ENDDO
```

■ SIMD化されたDOループのイメージ [2/2]

- 1 と [1] で、メモリ上のA(1)とA(2)の2要素に対し、一度に「Vレジスタ1」へ、ベクトルロード「Vロード」を行う。
- 2 と [2] で、「Vレジスタ1」内の2要素に対し、一度に10.0をベクトル加算「V加算」し、結果を「Vレジスタ2」に代入する。
- 3 と [3] で、「Vレジスタ2」の2要素に対し、一度にメモリ上のB(1)、B(2)へ、ベクトルストア「Vストア」を行う。



以上でIF文を含まない場合のSIMD化の動作イメージの説明を終わる。

SIMD化されたDOループのイメージ

DO I=1,4,2 0

Vロード	Vレジスタ1(1)=A(I)	1	Vレジスタ1(2)=A(I+1)	[1]
V加算	Vレジスタ2(1)=Vレジスタ1(1)+10.0	2	Vレジスタ2(2)=Vレジスタ1(2)+10.0	[2]
Vストア	B(I)=Vレジスタ2(1)	3	B(I+1)=Vレジスタ2(2)	[3]

ENDDO

- 各種最適化手法
  - SIMD化の促進
    - 【第一部からの抜粋】SIMD化の概念説明

本小節の内容

次に、IF文を含むDOループのSIMD化の動作イメージを説明します。なお、説明の便宜上FortranのDOループを用いて説明しています。

    - IF文を含まないDOループのSIMD化の動作イメージ
    - IF文を含むDOループのSIMD化の動作イメージ

■ 動作説明のためのループ例と簡易モデル

■ ループ例

右の図に示すIF文を含むDOループについて、SIMD化された場合の動作イメージを以降のスライドに示す。

ループ例

```
DO I=1,4
  IF (IFLAG(I)>0) THEN
    B(I) = A(I) + 10.0
  ENDIF
ENDDO
```

■ 簡易モデル（再掲）

- SIMD命令は1命令で2要素を処理するとする(*1)。

  (*1) 実際の演算器では一つのSIMD命令で倍精度実数の場合8要素を処理できる。

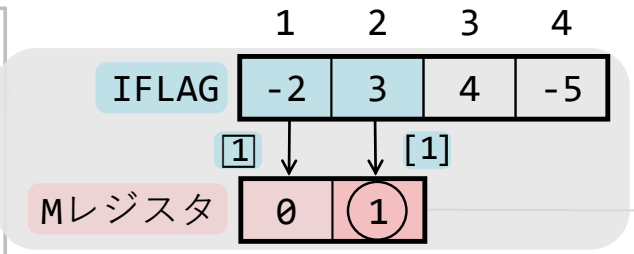
- 以降のスライドでは、通常の命令に対応するSIMD命令の呼び名・表記を以下のようにする（再掲）。

命令	対応するSIMD命令の呼び名	表記
ロード	ベクトルロード	「Vロード」
加算	ベクトル加算	「V加算」
ストア	ベクトルストア	「Vストア」

■ SIMD化されたDOループのイメージ [1/2]

- IF文の比較結果を設定するベクトルレジスタを、以下ではベクトルマスクレジスタと呼び、「Mレジスタ」と表す。
- 1 と[1]で、I=1とI=2に対し、①のIF文の条件式を一度に比較し、真ならば1を、偽ならば0を、「Mレジスタ」に設定する。
- （次スライドへ続く）

```
DO I=1,4
  IF (IFLAG(I)>0) THEN   ①
    B(I) = A(I) + 10.0   ②
  ENDIF
ENDDO
```

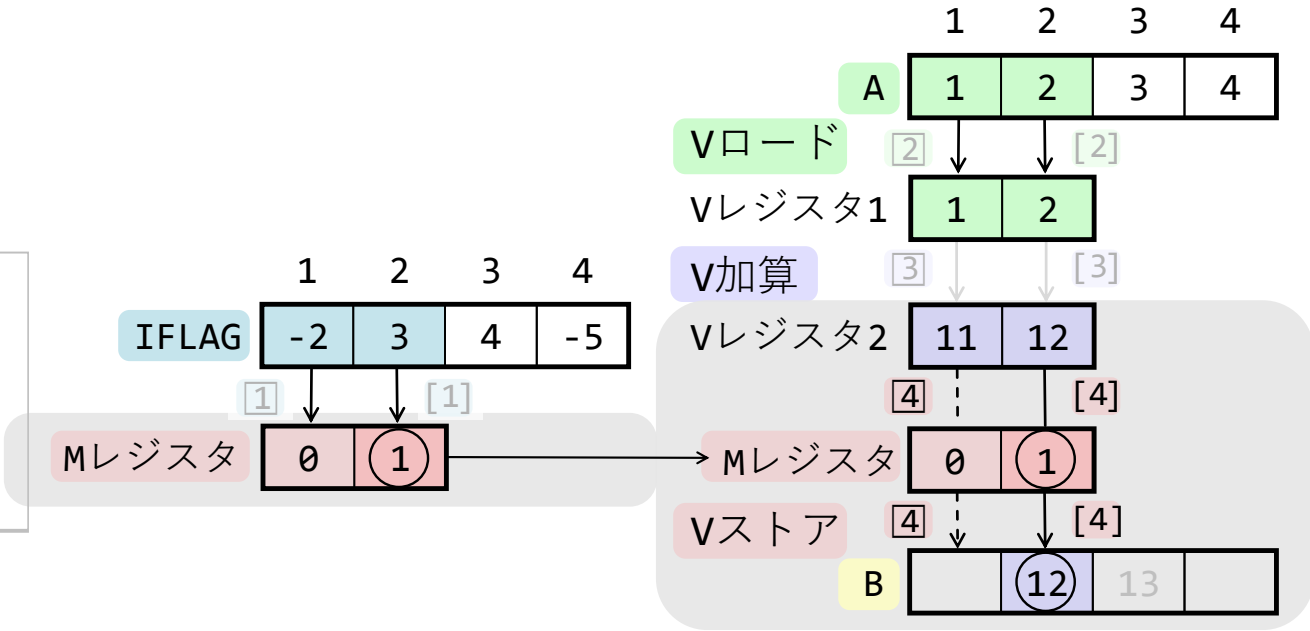


■ SIMD化されたDOループのイメージ [2/2]

- 「Vロード」で、②の右辺の配列Aを「Vレジスタ1」にロードする。
- 「V加算」で足し算を行い、結果を「Vレジスタ2」に代入する。
  - ⚠ ①の条件式が偽の要素に対しても②の右辺を実行することに注意。
- 「Vストア」で、「Mレジスタ」の値が1（つまり①の条件式が真）の要素に対してのみ、「Vレジスタ2」の値を②の左辺の配列Bに代入する。

以上でIF文を含む場合のSIMD化の動作イメージの説明を終わる。

```
DO I=1,4
  IF (IFLAG(I)>0) THEN  ①
    B(I) = A(I) + 10.0  ②
  ENDIF
ENDDO
```



■ 各種最適化手法

■ SIMD化の促進

本節の内容

■ 【第一部からの抜粋】SIMD化の概念説明

■ SIMD化前のフルアンローリング（ロード命令・ストア命令の効率化、演算の効率化）

多重ループにおいて、最内ループ長が短く、かつ、配列の最内インデックスが最内ループ変数でない場合を考えます。この場合、最内ループをフルアンローリングし外側ループでSIMD化する最適化が効果的となる可能性があります。この最適化は「SIMD化前のフルアンローリング」と呼ばれます。本小節では、SIMD化前のフルアンローリングにより、SIMD化が促進される例を示します。

■ ループ例（asis）

本小節での説明に使用するループ例を以下の図に示す。最内ループのIのループ長はNI=8と定義されており、短い。Iループの中で参照される配列Cは、最内インデックス（1番目のインデックス）が最内ループ変数となっていない。

```
INTEGER,PARAMETER::NI=8, NJ=63000
DOUBLE PRECISION::A(NJ),B(NI),C(NJ,NI)
COMMON/C_NI8/A,B,C
```

```
DO J=1,NJ
  TMP = 0D0
  DO I=1,NI
    C0 = C(J,I)
    B0 = B(I)
    TMP = TMP + C0 * B0
  ENDDO
  A(J) = TMP
ENDDO
```

- 以降ではまず、「SIMD化前のフルアンローリング」最適化が適用されていない上記の状態のままのコンパイルリストを示す。
- 次に「SIMD化前のフルアンローリング」最適化の指示行を示し、それを適用した場合のコンパイルリストを示す。
- なお、最適化のコンパイルオプションは-Kfastとし、逐次実行とする。（指示行を有効にするため-Koclも指定する。）CPU性能解析レポートを採取する。

■ ループ例（asis）のコンパイルリスト

本ループ例について「SIMD化前のフルアンローリング」最適化が適用されていない状態のままのコンパイルリストを示す。紙面の都合上、出力を加工している。

- 最内のIループについて、記号「v」、「SIMD(VL: 8)」およびjwd6004が示すように最内のIループがSIMD化されている。
  - 最内のループ変数Iが配列Cの2番目のインデックスであるため、配列Cのロードは連続的でないSIMDギャザロード（SIMD Non-contiguous gather load）である。これは連続的なロードと比較してロードの効率が悪い。
- 変数JはSIMD化対象の変数でないため、配列Aへのストアは非SIMDストア（Non-SIMD store）である。これはSIMDの連続的なストアと比較してストアの効率が悪い。

```
 999    2    2     DO J=1,NJ
1000    2    2       TMP = 0D0
1001    3    2v      DO I=1,NI
1002    3    2v        C0 = C(J,I)
1003    3    2v        B0 = B(I)
1004    3    2v        TMP = TMP + C0 * B0
1005    3    2v      ENDDO
1006    2    2       A(J) = TMP
1007    2    2     ENDDO
```

Jループの<<< Loop-information >>>

```
<<<  [OPTIMIZATION]
<<<    SOFTWARE PIPELINING(IPC: 2.27, ITR:
192, MVE: 3, POL: S)
<<<    PREFETCH(HARD) Expected by
compiler :
<<<      A
```

Iループの<<< Loop-information >>>

```
<<<  [OPTIMIZATION]
<<<    SIMD(VL: 8)
```

Iループのメッセージ

```
jwd6004s-i  "～", line 1001: リダクション演算を含むDOループをSIMD化しました。(名前:I)
```

■ 【補足】「内側ループのSIMD化＋アンローリング」のイメージ

前スライドのコンパイルリストでの内側Iループ部分を左下の図に再掲する。記号「2v」はIループがSIMD化および2段アンローリングされていることを示す(*1)。このSIMD化および2段アンローリングの処理イメージを右下の図に示す（簡単化のため、NIは16の倍数とする）。本例では倍精度の配列であるためSIMD化により、8個の要素が一度に同時にロードされ、ベクトルレジスタに格納される。ベクトル成分ごとに積が累積される。2段アンローリングのため、その処理の塊が二つ存在する。Iループのストライドは、（SIMDで8、2段アンローリングで2のため、）8*2で16となる。
右下の図のようなアンローリングを「SIMD化後のアンローリング」と呼ぶ。

元のループ
（外側のJループ部分は省略(*1)）

```
2v    DO I=1,NI
2v      C0 = C(J,I)
2v      B0 = B(I)
2v      TMP = TMP + C0 * B0
2v    ENDDO
```

SIMD化＋2段アンローリング後のイメージ

```
DO I=1,NI,8*2
  T_C1(0) = C(J,I)
  ...
  T_C1(7) = C(J,I+7)
  T_B1(0) = B(I)
  ...
  T_B1(7) = B(I+7)
  T_TMP1(0:7) = T_TMP1(0:7) + T_C1(0:7)*T_B1(0:7)
  T_C2(0) = C(J,I+8)
  ...
  T_C2(7) = C(J,I+15)
  T_B2(0) = B(I+8)
  ...
  T_B2(7) = B(I+15)
  T_TMP2(0:7) = T_TMP2(0:7) + T_C2(0:7)*T_B2(0:7)
ENDDO
T_TMP1(0:7) = T_TMP1(0:7) + T_TMP2(0:7)
TMP = T_TMP1(0)+...+T_TMP1(7)
```

(*1) 本スライド【補足】は「SIMD化後のアンローリング」を説明することを目的とします。簡単化のため、前スライドのコンパイルリストでの外側Jループのアンローリングの状況は考えないこととします。

■ SIMD化前のフルアンローリング最適化を有効化する指示行 [1/2]

本ループ例（右図に再掲）に対して「SIMD化前のフルアンローリング」最適化を適用する方法を示す。本最適化の指定は指示行`!OCL FULLUNROLL_PRE_SIMD`により行う。ここではこの指示行の書式および指定例を説明する。

```
DO J=1,NJ
  TMP = 0D0
  DO I=1,NI
    C0 = C(J,I)
    B0 = B(I)
    TMP = TMP + C0 * B0
  ENDDO
  A(J) = TMP
ENDDO
```

以下に示す指示行により、「SIMD化前のフルアンローリング」最適化の促進を指示することができる。

- 指示行（DOループ単位、および配列代入文単位のみで指定可能）

  `!OCL FULLUNROLL_PRE_SIMD[(`*n*`)]`

- 引数*n*について
  - 引数*n*は対象ループの回転数の上限を表す2～100の整数値である。
  - 引数*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する。
- 注意事項
  - 本指示行は指定した直後のDOループまたは配列記述のみが対象となる。
  - 繰返し数が不明な場合は最適化を行わない。

■ SIMD化前のフルアンローリング最適化を有効化する指示行 [2/2]

■ 指示行の指定例

本小節で使用しているループ例に対して指示行!OCL FULLUNROLL_PRE_SIMDを指定する例を左下の図に示す。左下の図は、元のループ例でIループを手作業でフルアンローリングした右下の図と同等である。

元のループに本指示行を指定

```
INTEGER,PARAMETER::NI=8
  ...
  DO J=1,NJ
    TMP = 0D0
!OCL FULLUNROLL_PRE_SIMD
    DO I=1,NI
      C0 = C(J,I)
      B0 = B(I)
      TMP = TMP + C0 * B0
    ENDDO
    A(J) = TMP
  ENDDO
```

⇔

最適化後のイメージ

```
INTEGER,PARAMETER::NI=8
  ...
  DO J=1,NJ
    C1=C(J,1); B1=B(1); TMP=C1*B1
    C2=C(J,2); B2=B(2); TMP=TMP+C2*B2
    C3=C(J,3); B3=B(3); TMP=TMP+C3*B3
    C4=C(J,4); B4=B(4); TMP=TMP+C4*B4
    C5=C(J,5); B5=B(5); TMP=TMP+C5*B5
    C6=C(J,6); B6=B(6); TMP=TMP+C6*B6
    C7=C(J,7); B7=B(7); TMP=TMP+C7*B7
    C8=C(J,8); B8=B(8); TMP=TMP+C8*B8
    A(J) = TMP
  ENDDO
```

右の図ではJが最内ループである。第2インデックスを固定した配列C(J,1)は変数Jが最内インデックスであるためJについて連続であり、JループがSIMD化されればSIMD連続ロードとなる（C(J,2)、...C(J,8)についても同様）。配列要素A(J)へのストアはJループがSIMD化されればSIMD連続ストアとなる。

■ ループ例（tune）のコンパイルリスト [1/2]

本ループ例に対して「SIMD化前のフルアンローリング」最適化の指示行を指定したコンパイルリストを以下の図に示す。Iループに対して指示行!OCL FULLUNROLL_PRE_SIMDを指定する。

- Iループに対する<<< Loop-information >>>内、DO文の左の記号「f」、およびjwd8203が示すように、Iループはフルアンローリングされる。
- Iループに対する<<< Loop-information >>>内には記号「SIMD(VL: 8)」が表示されなくなった。
- （次スライドに説明続く）

```
1030   2    v      DO J=1,NJ
1031   2    v        TMP = 0D0
1032   2           !OCL FULLUNROLL_PRE_SIMD
1033   3   fv       DO I=1,NI
1034   3   fv         C0 = C(J,I)
1035   3   fv         B0 = B(I)
1036   3   fv         TMP = TMP + C0 * B0
1037   3   fv       ENDDO
1038   2    v       A(J) = TMP
1039   2    v     ENDDO
```

Jループの<<< Loop-information >>>

```
<<<  [OPTIMIZATION]
<<<    SIMD(VL: 8)
<<<    SOFTWARE PIPELINING(IPC: 2.00,
ITR: 120, MVE: 9, POL: S)
<<<    PREFETCH(HARD) Expected by
compiler :
<<<      C, A
```

Iループの<<< Loop-information >>>

```
<<<  [OPTIMIZATION]
<<<    FULL UNROLLING
```

Jループのメッセージ

（次スライドで示す）

Iループのメッセージ

```
jwd8203o-i  "～", line 1033: ループをフルアンローリングしました。
jwd8209o-i  "～", line 1036: 多項式の演算順序を変更しました。
```

■ ループ例（tune）のコンパイルリスト [2/2]

- 記号「SIMD(VL: 8)」はJループに対する<<< Loop-information >>>内に表示され、SIMD化されたことを示す記号「v」はJループのDO文の左にも付き、Jループ内の全てがSIMD化されたされたことが確認される。

```
1030    2     v      DO J=1,NJ
1031    2     v        TMP = 0D0
1032    2            !OCL FULLUNROLL_PRE_SIMD
1033    3    fv        DO I=1,NI
1034    3    fv          C0 = C(J,I)
1035    3    fv          B0 = B(I)
1036    3    fv          TMP = TMP + C0 * B0
1037    3    fv        ENDDO
1038    2     v        A(J) = TMP
1039    2     v      ENDDO
```

Jループの<<< Loop-information >>>

```
<<<  [OPTIMIZATION]
<<<    SIMD(VL: 8)
<<<    SOFTWARE PIPELINING(IPC: 2.00,
ITR: 120, MVE: 9, POL: S)
<<<    PREFETCH(HARD) Expected by
compiler :
<<<      C, A
```

Iループの<<< Loop-information >>>

```
<<<  [OPTIMIZATION]
<<<    FULL UNROLLING
```

Jループのメッセージ

```
jwd6001s-i  "～", line 1030: このDOループをSIMD化しました。(名前:J)
jwd8204o-i  "～", line 1030: 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  "～", line 1030: ループの繰返し数が120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。
```

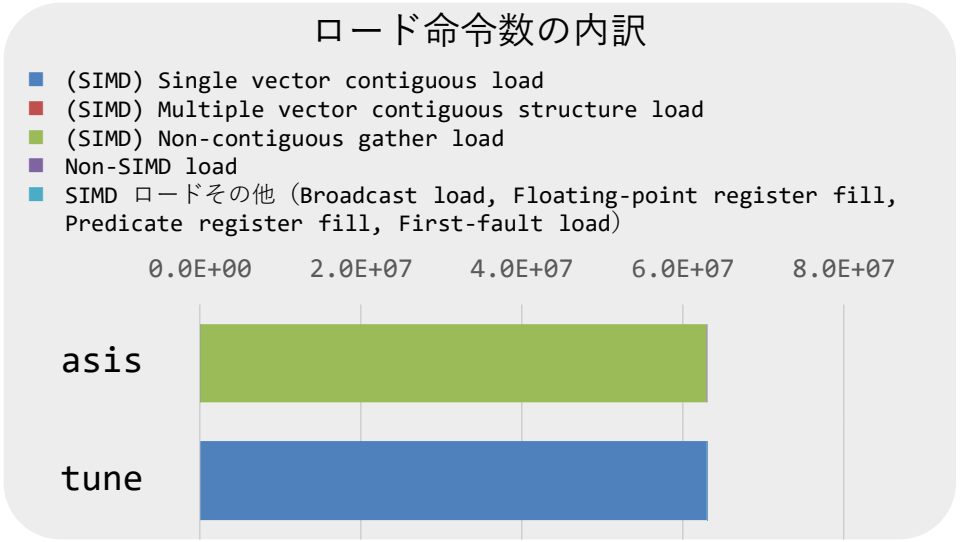
■ ループ例（asisとtune）実行結果の比較 [1/3]：ロード・ストア命令数の内訳

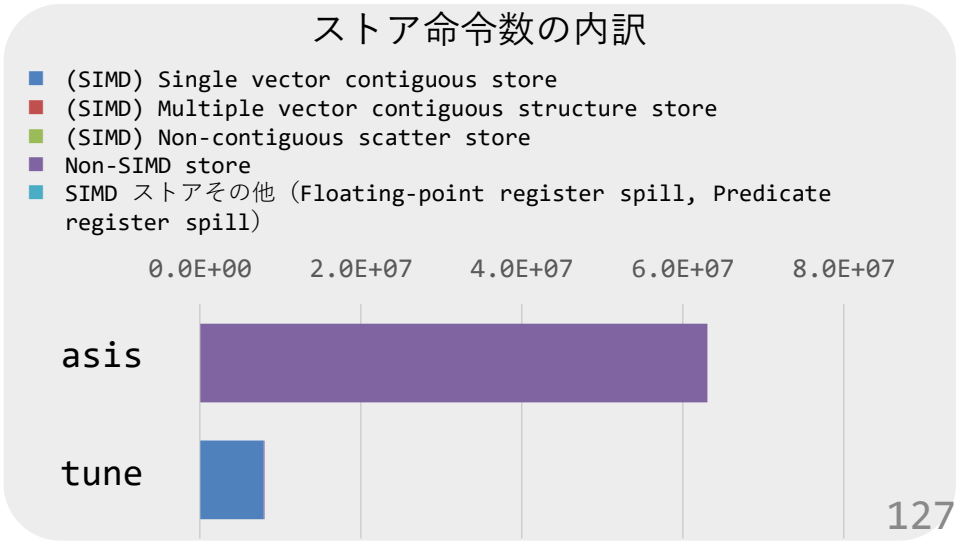
以降のスライドでは、「SIMD化前のフルアンローリング」最適化を適用しない場合（asis）とする場合（tune）それぞれを逐次実行し、CPU性能解析レポートを採取し、比較した結果を示す（計測のためのループの繰り返し数NLOOPを1000とする）。

```
asis: 指示行無指定
tune: !OCL FULLUNROLL_PRE_SIMDを指定
```

■ 右上の図に、ロード命令数の内訳の比較を示す。asisでは連続的でないSIMDギャザロード（SIMD Non-contiguous gather load）（■）だが、tuneでは連続ロード（■）となり、最適化によりロードが効率化したことが確認される。

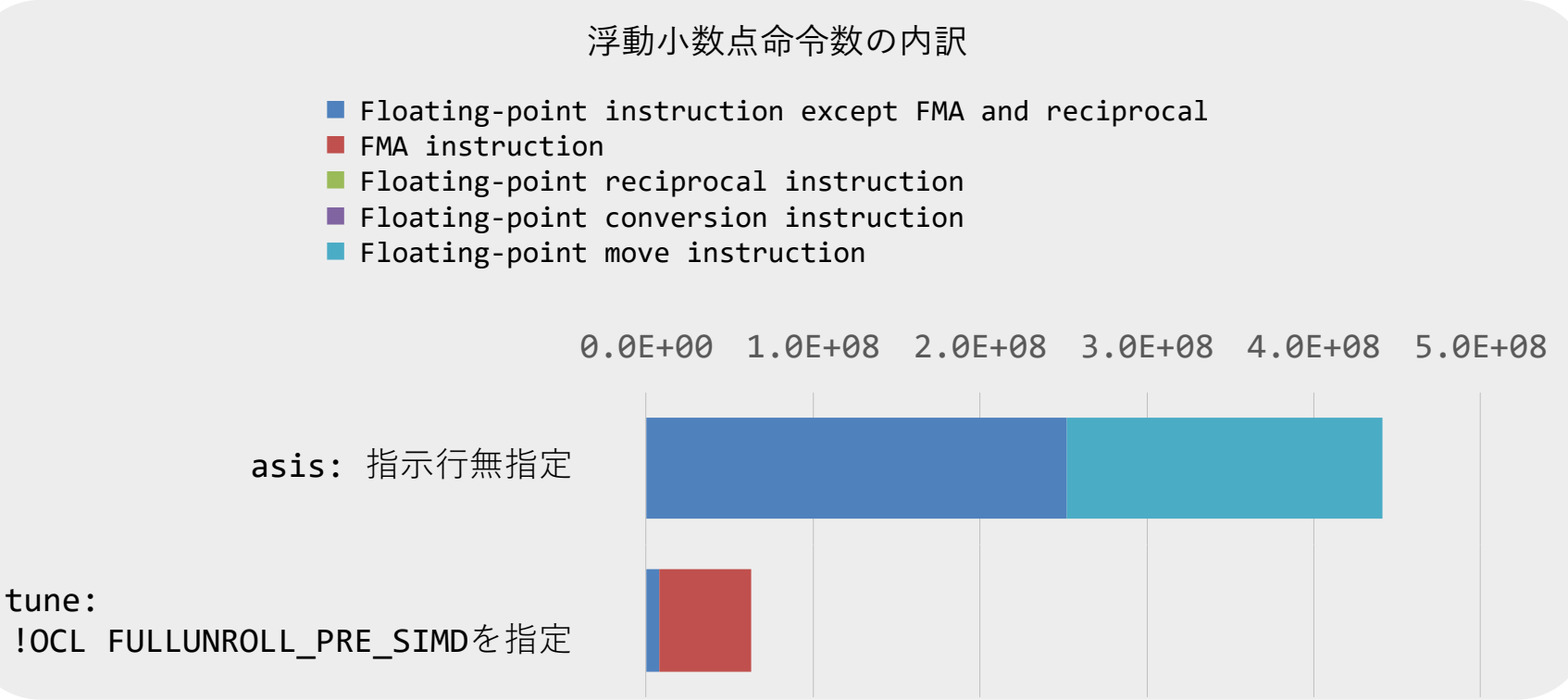
■ 右下の図にストア命令数の内訳を示す。asisでは非SIMDストア（Non-SIMD store）（■）だが、tuneではSIMD連続ストア（■）となり、またストア命令の数も激減しており、最適化によりストアが効率化したことが確認される。





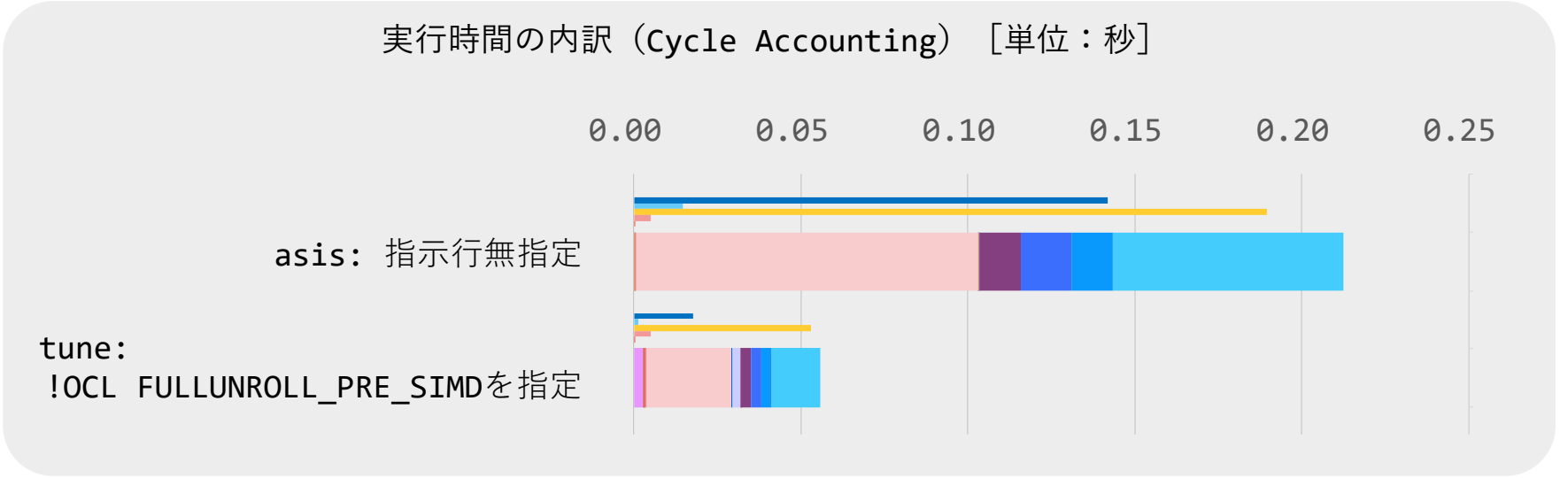
■ ループ例（asisとtune）実行結果の比較 [2/3]：浮動小数点命令数の内訳

下の図に浮動小数点命令数の内訳の比較を示す。asisでは積和演算や逆数近似演算以外の命令（■）およびmove命令（■）がほとんどを占めている。対してtuneではそれらの命令はごく少数となり、代わりに積和演算命令（■）が実行されていることが確認される。tuneでの浮動小数点命令数全体の数自体もasisと比較し激減していることが確認される。最適化により演算が効率化したことが確認される。



■ ループ例（asisとtune）実行結果の比較 [3/3]：実行時間の内訳

実行時間の内訳（Cycle Accounting）を以下の図に示す。前々スライドおよび前スライドからasisと比較しtuneはロード・ストアが効率化、および浮動小数点演算が効率化したことにより、下の図からL1Dキャッシュアクセス待ち（■）およびコミットの時間（■、■、■、および■）が改善されたことが確認される。



内容

- 「富岳」の概略
- プロファイラ（CPU性能解析レポート）
- 各種最適化手法
  - 演算の効率化
  - ソフトウェアパイプライニングの促進
  - SIMD化の促進（ロード命令・ストア命令の効率化、演算の効率化）
  - キャッシュチューニングのための基本事項
- 【付録】

- 各種最適化手法
  - キャッシュチューニングのための基本事項

本節の内容

本節ではキャッシュチューニング（データキャッシュを意識したチューニング）のための基本事項を説明します。本中級編の第一部では、データキャッシュ構成、およびキャッシュチューニングのための基本事項の一つとしてプリフェッチを説明しています。（第一部では、プリフェッチの概念説明、コンパイルオプションと指示行、およびプリフェッチを適用するループ例をいくつか説明しています。）この中級編第三部では、プリフェッチ以外の基本事項の説明に重点を置くこととします。

- データキャッシュ構成
- キャッシュのレイテンシの改善（プリフェッチの概念説明）
- 避けるべきL1Dキャッシュミス多発の状況（キャッシュスラッシング、False Sharing）
- キャッシュへのデータ転送量の軽減（高速ストアZFILL）
- キャッシュ利用効率の向上
  （ループブロッキング、アンロール・アンド・ジャム、セクタキャッシュ）

- 各種最適化手法
  - キャッシュチューニングのための基本事項

本節の内容

    - データキャッシュ構成
    - キャッシュのレイテンシの改善
    - 避けるべきL1Dキャッシュミス多発の状況
    - キャッシュへのデータ転送量の軽減
    - キャッシュ利用効率の向上

- 各種最適化手法
  - キャッシュチューニングのための基本事項
    - データキャッシュ構成

本小節の内容

データキャッシュを意識したチューニングのためにはデータキャッシュの大きさ・階層等の構成を理解している必要があります。データキャッシュの構成については本中級編第一部で説明していますが、第一部の未受講者を想定し、まず予備知識としてデータキャッシュの構成を説明します。

- 「富岳」のデータキャッシュの大きさ
- 1次キャッシュと2次キャッシュの対応
- ウェイについて

■ 「富岳」のデータキャッシュの大きさ

1次（L1D）キャッシュ	ウェイ数(*1)	4ウェイ	4ウェイの合計 64 KiB	各コアごとに存在
	ウェイ当たりバイト数	16 KiB		
	ウェイ当たりキャッシュライン数（=16 KiB/256 B）	64ライン		
	ライン当たり要素数［倍精度の場合（=256/8）］	32個		
	［単精度の場合（=256/4）］	64個		

2次（L2）キャッシュ	ウェイ数(*1)	16ウェイ	16ウェイの合計 8 MiB	各CMGごとに存在
	ウェイ当たりバイト数	0.5 MiB		
	ウェイ当たりキャッシュライン数（=0.5 MiB/256 B）	2,048ライン		
	ライン当たり要素数［倍精度の場合（=256/8）］	32個		
	［単精度の場合（=256/4）］	64個		

1次キャッシュ

	ウェイ0	ウェイ1	ウェイ2	ウェイ3
ライン0	A(1)	A(2049)	A(4097)	A(6145)
	:	:	:	:
	A(32)	A(2080)	A(4128)	A(6176)
:	:	:	:	:
ライン63	A(2017)	A(4065)	A(6113)	A(8161)
	:	:	:	:
	A(2048)	A(4096)	A(6144)	A(8192)

2次キャッシュ

	ウェイ0	...	ウェイ15
ライン0	A(1)	...	A(983041)
	:	:	:
	A(32)	...	A(983072)
:	:	:	:
:	:	:	:
:	:	:	:
:	:	:	:
:	:	:	:
ライン2047	A(65505)	...	A(1048545)
	:	:	:
	A(65536)	...	A(1048576)

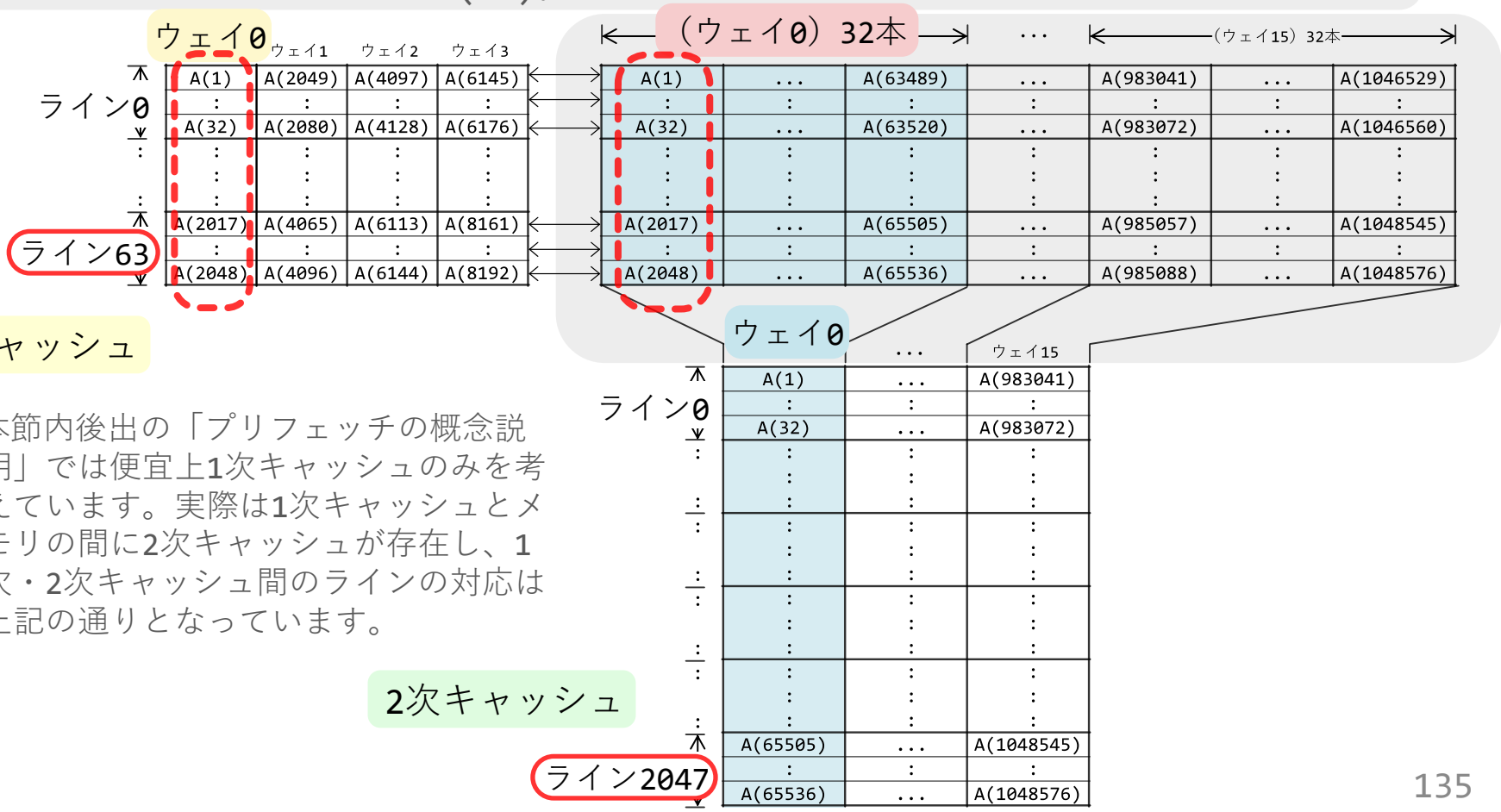
(*1) ウェイについては、後の「ウェイについて」を参照。

■ 1次キャッシュと2次キャッシュの対応

- ■ 1次キャッシュと2次キャッシュは、右図（前スライドからの抜粋）に示すように、1ウェイ当たりのライン数が異なる。

1次キャッシュ	：1ウェイ当たり	64ライン
2次キャッシュ	：1ウェイ当たり	2,048ライン

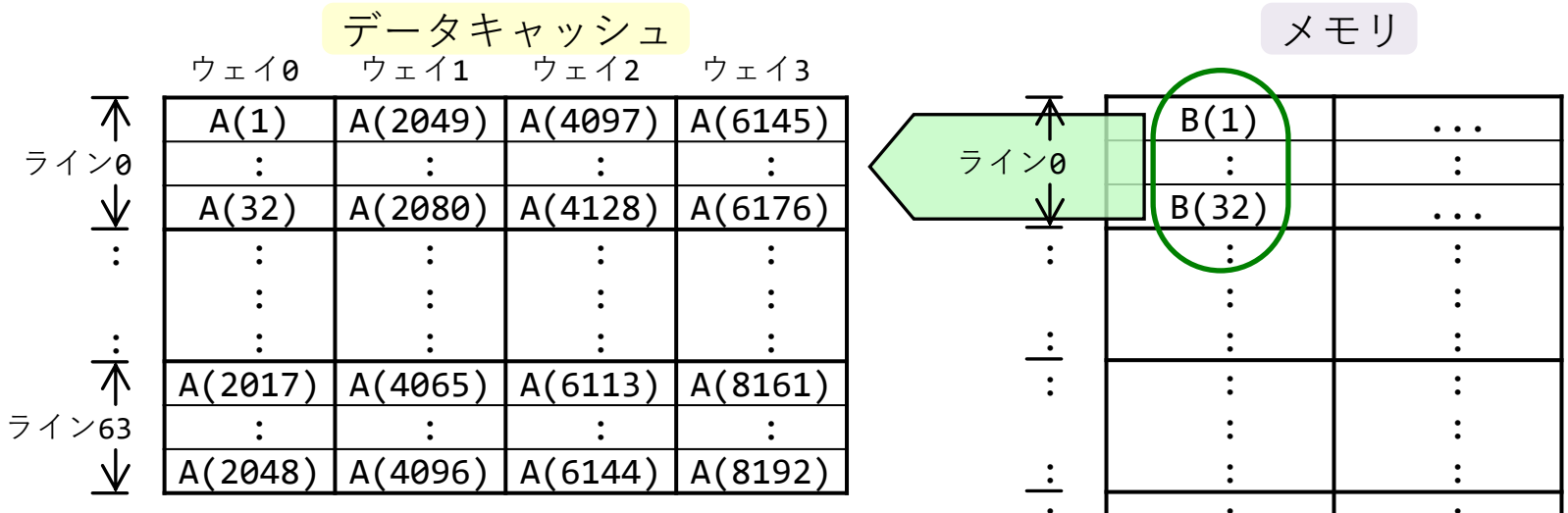
- ■ 2次キャッシュが図のように2048本のラインが64×32に並べてられていると見なして1次キャッシュと対応付ける(*2)。



(*2) 本節内後出の「プリフェッチの概念説明」では便宜上1次キャッシュのみを考えています。実際は1次キャッシュとメモリの間に2次キャッシュが存在し、1次・2次キャッシュ間のラインの対応は上記の通りとなっています。

■ ウェイについて

- ウェイとは同じライン番号のキャッシュラインのデータを複数保持するための仕組みです。
- メモリとデータキャッシュの間のデータ転送では、データはキャッシュライン単位で転送される。
- データ（配列要素または変数）がどのキャッシュライン番号に属するかは、そのデータのメモリ上のアドレスから決まる。
- 以下では倍精度実数の配列要素B(1)が属するライン番号が0であるとする。また以下では、配列要素B(1)がデータキャッシュ上に存在せず、B(1)をメモリからデータキャッシュに転送する必要が生じた場合を考える(*3)。(*3) 説明の便宜上キャッシュを1次キャッシュのみと考え、単にデータキャッシュとする。
  - B(1)を含む一つのライン単位のデータが、メモリからデータキャッシュに転送される。
  - 転送されたデータは、データキャッシュ上のライン番号0について、いずれかのウェイに空きがあれば、そのウェイのライン番号0のラインに納まる。
  - データキャッシュ上のどのウェイのライン0にも空きがなければ（下図のデータキャッシュのような場合）、各ウェイのライン0のうち一番過去にアクセスされたライン0のデータがキャッシュから追い出され、転送されたデータはそこに納まる。



- 各種最適化手法
  - キャッシュチューニングのための基本事項

本節の内容

    - データキャッシュ構成
    - キャッシュのレイテンシの改善
    - 避けるべきL1Dキャッシュミス多発の状況
    - キャッシュへのデータ転送量の軽減
    - キャッシュ利用効率の向上

- 各種最適化手法
  - キャッシュチューニングのための基本事項
    - キャッシュのレイテンシの改善

本小節の内容

本小節では、キャッシュへのデータ転送時間を隠蔽するための手法であるプリフェッチについて説明します。プリフェッチについては、本中級編第一部において、概念説明、コンパイルオプションと指示行、およびプリフェッチを適用するループ例をいくつか説明しています。そのため、この第三部ではプリフェッチの概念の説明のみとします。

- 【第一部からの抜粋】プリフェッチの概念説明

■ 用語「データキャッシュ」とループ例

- 「富岳」のデータキャッシュは1次キャッシュと2次キャッシュの2種類だが、以下の説明では単にデータキャッシュとする(*1)。

  (*1) 本小節では説明の便宜上1次キャッシュのみを考え、単にデータキャッシュと呼ぶこととします。実際は1次キャッシュとメモリの間に2次キャッシュが存在し、1次・2次キャッシュ間のラインの対応は「データキャッシュ構成」小節で述べた通りです。

- 右の図のループ例について、まずプリフェッチを行わない場合の動作を説明し、次にプリフェッチを行う場合の動作を説明する。

ループ例

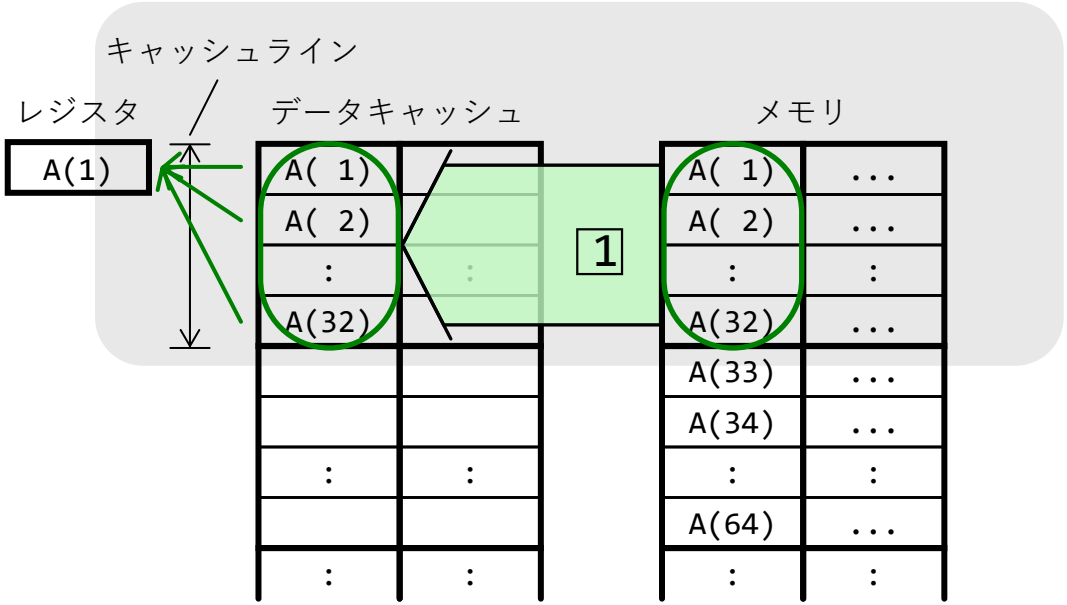
```
  :
REAL*8 A(N)
DO I=1,N
  A(I) = A(I) + 1.0
ENDDO
  :
```

■ プリフェッチを行わない場合の動作の説明 [1/2]

- ループの実行前には配列Aのデータはデータキャッシュ上に存在していないとする。
- まずA(1)が必要になるが、A(1)がデータキャッシュ上に（最初なので）存在しないため、キャッシュミスが発生し、1 に示すようにA(1)～A(32)（キャッシュの1ライン分） がメモリからデータキャッシュに転送される。
  - この転送には時間がかかる。
- 配列Aの各要素が、A(1)、A(2)、・・・、A(32)の順にデータキャッシュからレジスタに転送され、A(1)～A(32)の計算が行われる。
- （次スライドへ続く）

ループ例

```
  :
REAL*8 A(N)
DO I=1,N
  A(I) = A(I) + 1.0
ENDDO
  :
```

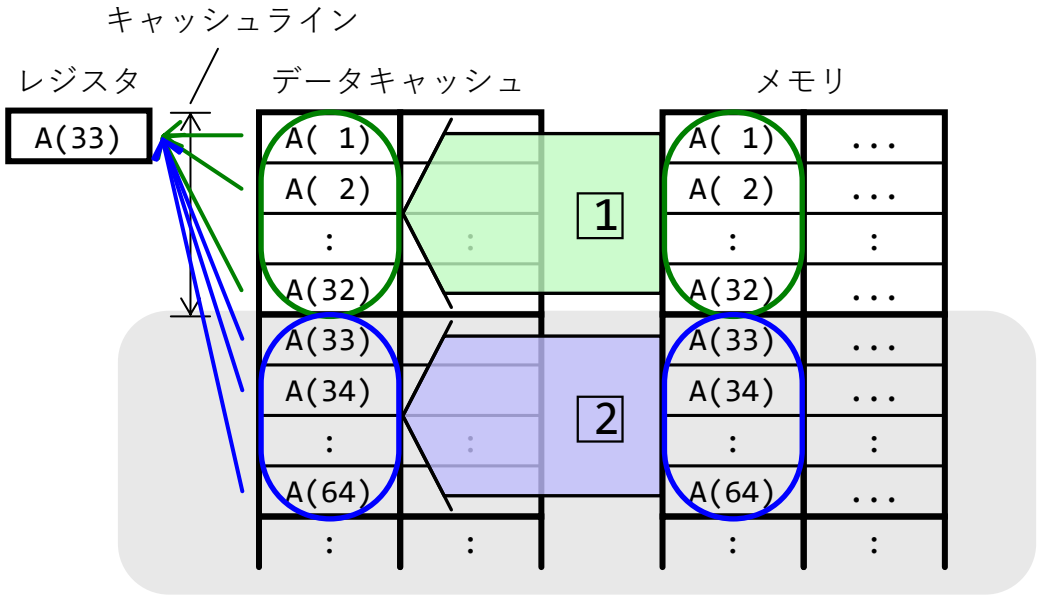


■ プリフェッチを行わない場合の動作の説明 [2/2]

- 次にA(33)が必要になるが、A(33)がデータキャッシュ上にないため、再びキャッシュミスが発生し、 2 の転送が行われる。
- A(33)、A(34)、・・・、A(64)の順にデータキャッシュからレジスタに転送され、A(33)～A(64)の計算が行われる。

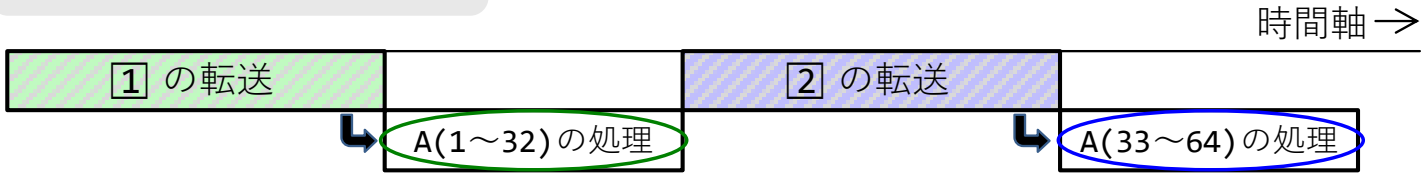
ループ例

```
   :
REAL*8 A(N)
DO I=1,N
  A(I) = A(I) + 1.0
ENDDO
   :
```



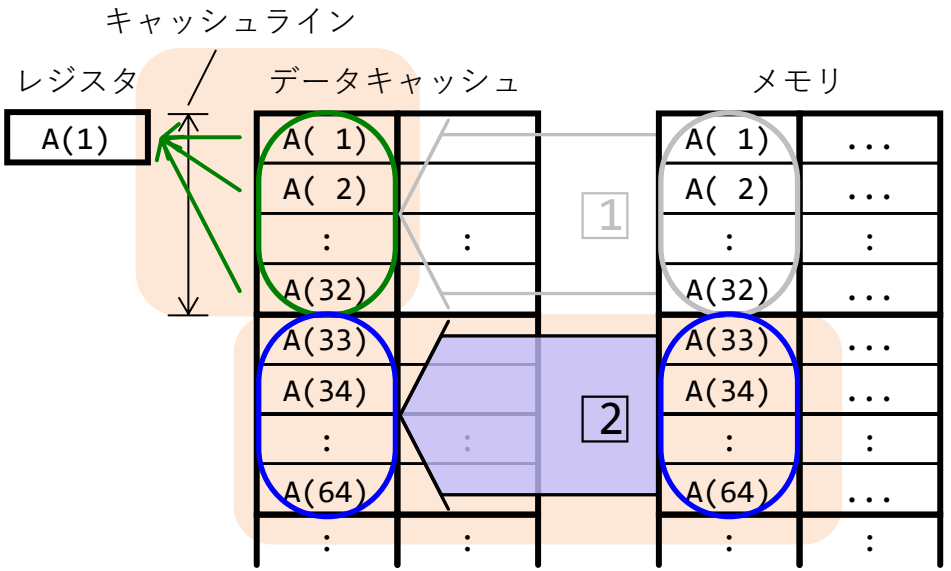
- 以上の動作のタイムチャートを以下の図に示す。

プリフェッチを行わない場合

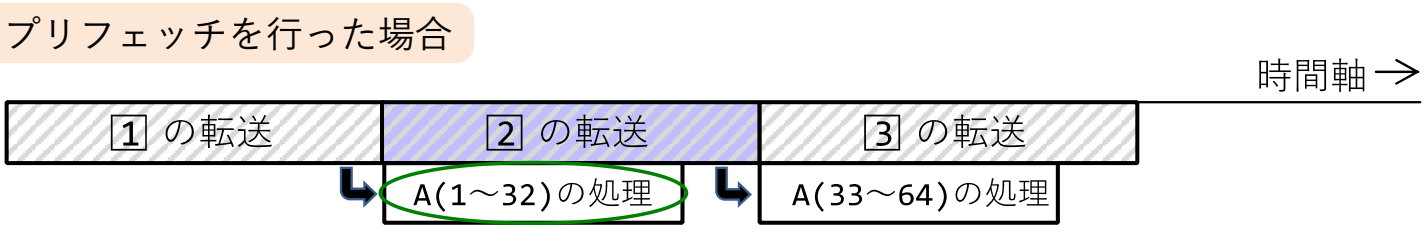


■ プリフェッチを行う場合の動作の説明

■ 配列Aに対してプリフェッチの最適化が行われた場合、要素A(1)～A(32)の処理が行われている間に、その次に必要となる 2 の転送も同時に行われる。



■ タイムチャートは右の図のようになり、時間のかかる 2 の転送時間をある程度隠蔽することができる。



■ プリフェッチの種類

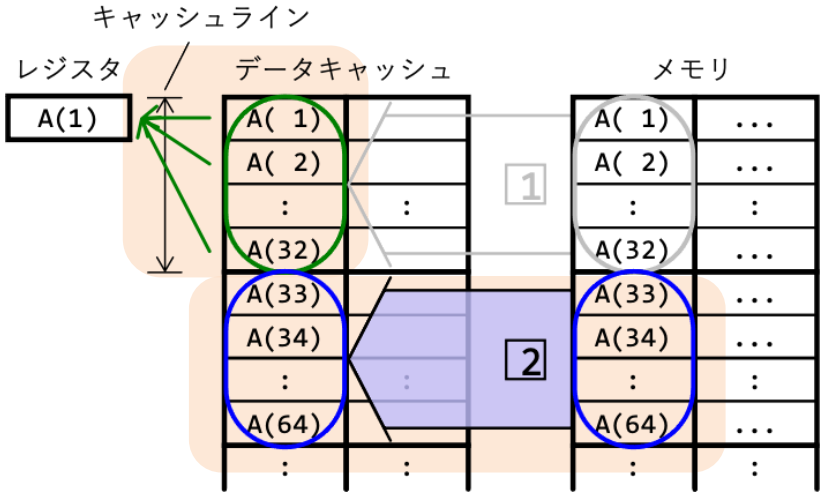
■ ハードウェアプリフェッチ：ハードウェアが自動的に行う方法。

■ ソフトウェアプリフェッチ：プログラム内に`prefetch`命令を生成する方法。

- コンパイルオプション(*1)や指示行(*1)で行うプリフェッチは ソフトウェアプリフェッチである。

(*1) 中級編第一部で説明していますので、本中級編第三部では省略します。

■ プリフェッチの注意事項および用語説明



- プリフェッチの有無に関わらず、本例でA(33)のデータが必要になりキャッシュミスが起きる。プリフェッチはデータのメモリからの 2 の転送時間を隠蔽することが目的であり、キャッシュの利用効率向上によるキャッシュミス数の低減が目的ではない。

- プリフェッチを行わない場合
  - この時の 2 の転送はロード命令での転送であり、この時のメモリアクセスをデマンドアクセスという。また、この時のキャッシュミスをデマンドアクセスでのキャッシュミスという。

- プリフェッチを行った場合
  - この時の 2 の転送でのメモリアクセスを（ハードウェアまたはソフトウェア）プリフェッチによるメモリアクセスという。また、この時のキャッシュミスを（ハードウェアまたはソフトウェア）プリフェッチでのキャッシュミスという。

- CPU性能解析レポートでは、上記のキャッシュミスは、プリフェッチを行わない場合は「キャッシュミス（デマンド率）」に計上され、プリフェッチを行った場合は「キャッシュミス（ハードウェアまたはソフトウェアプリフェッチ率）」に計上される。
  - プリフェッチはキャッシュミスのデマンド率を低減することが目的であるといえる。

- 各種最適化手法
  - キャッシュチューニングのための基本事項

本節の内容

    - データキャッシュ構成
    - キャッシュのレイテンシの改善
    - 避けるべきL1Dキャッシュミス多発の状況
    - キャッシュへのデータ転送量の軽減
    - キャッシュ利用効率の向上

- 各種最適化手法
  - キャッシュチューニングのための基本事項
    - 避けるべきL1Dキャッシュミス多発の状況

本小節の内容

本小節では、キャッシュのラインまたはウェイの大きさとの関係においてキャッシュへのデータの配置の仕方しだいでは、L1Dキャッシュミスが多発する状況が存在することを説明します。キャッシュチューニングにおいては常に本小節の事項を留意することが重要です。

    - キャッシュスラッシング
    - スレッド並列におけるFalse Sharing

- 各種最適化手法
  - キャッシュチューニングのための基本事項
    - 避けるべきL1Dキャッシュミス多発の状況

本小節の内容

    - キャッシュスラッシング
    - スレッド並列におけるFalse Sharing

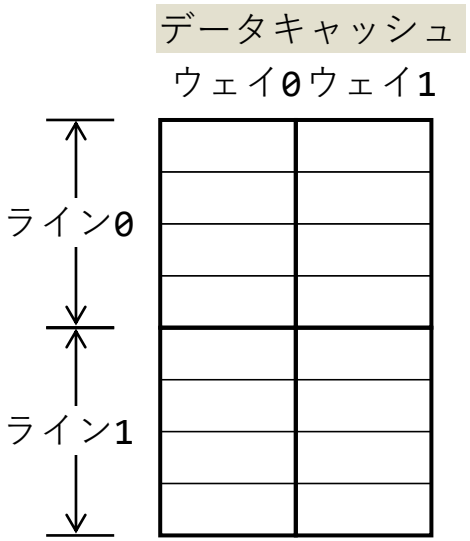
■ データキャッシュの簡易モデル

本項ではキャッシュの動作を簡素化し分かりやすくする目的で、簡易モデルを使用して説明する。

■ メモリとレジスタの間には実際のL1Dキャッシュを簡単化したキャッシュのみが存在するとする。この簡単化したキャッシュを以後、データキャッシュあるいは単にキャッシュと呼ぶことにする。

■ 簡単化したキャッシュは以下のようなものとする（下の図も参照）。

- ウェイ数：2
- 1ウェイあたりのキャッシュライン数：2
- 1キャッシュラインあたりの大きさ：倍精度実数で4



■ キャッシュスラッシング発生の動作説明 [1/6]

本項では簡単なループ例について簡易モデルでのキャッシュの動作を通してキャッシュスラッシングがどのように発生するかを以降のスライドで説明する。

- 右のプログラム例のループはストライドが1で処理が行われているが、本項の簡易モデルではキャッシュミスが多発する。その理由を以降のスライド数ページに渡って説明する。

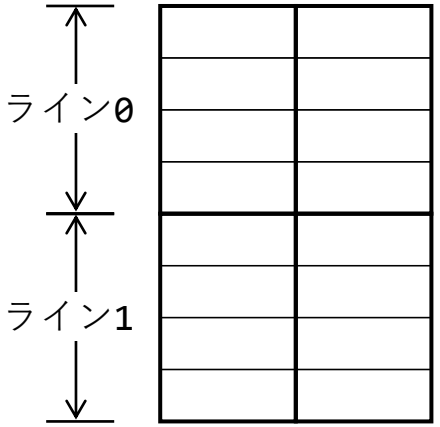
プログラム例

```
DOUBLE PRECISION::A(8),B(8),C(8)
DO I=1,8
  A(I) = A(I) + B(I) + C(I)
ENDDO
```

- 上のプログラムでは、配列A、B、Cはメモリ内に右下の図に示すように配置される。
- ループの実行前には配列A、B、Cのデータは左下の図に示すようにキャッシュ上に存在していないとする。

データキャッシュ

ループの実行前



メモリ

A(1)	B(1)	C(1)
A(2)	B(2)	C(2)
A(3)	B(3)	C(3)
A(4)	B(4)	C(4)
A(5)	B(5)	C(5)
A(6)	B(6)	C(6)
A(7)	B(7)	C(7)
A(8)	B(8)	C(8)

■ キャッシュスラッシング発生の動作説明 [2/6]

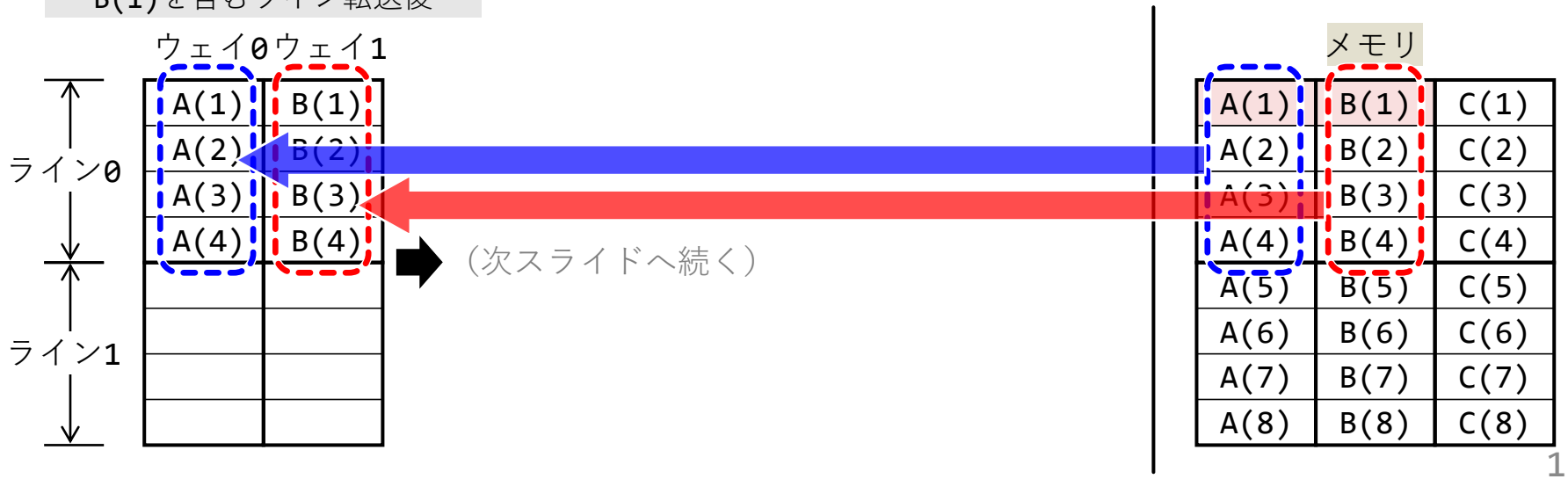
- 本例のループの展開図を右の図に示す。
- 本例のプログラムを実行すると、まず A(1) の参照が必要になり、右下のメモリの図の A(1) の参照で最初のアクセスなのでキャッシュミスが発生し、A(1)～A(4) がキャッシュに転送される。
- 次に B(1) の参照が必要になり、最初のアクセスなのでキャッシュミスが発生し、B(1)～B(4) がキャッシュに転送される。
- 転送後、キャッシュの状態は左下の図のようになる。

ループの展開図

```
A(1) = A(1) + B(1) + C(1)
A(2) = A(2) + B(2) + C(2)
A(3) = A(3) + B(3) + C(3)
A(4) = A(4) + B(4) + C(4)
A(5) = A(5) + B(5) + C(5)
A(6) = A(6) + B(6) + C(6)
A(7) = A(7) + B(7) + C(7)
A(8) = A(8) + B(8) + C(8)
```

データキャッシュ

A(1)を含むラインと、
B(1)を含むライン転送後

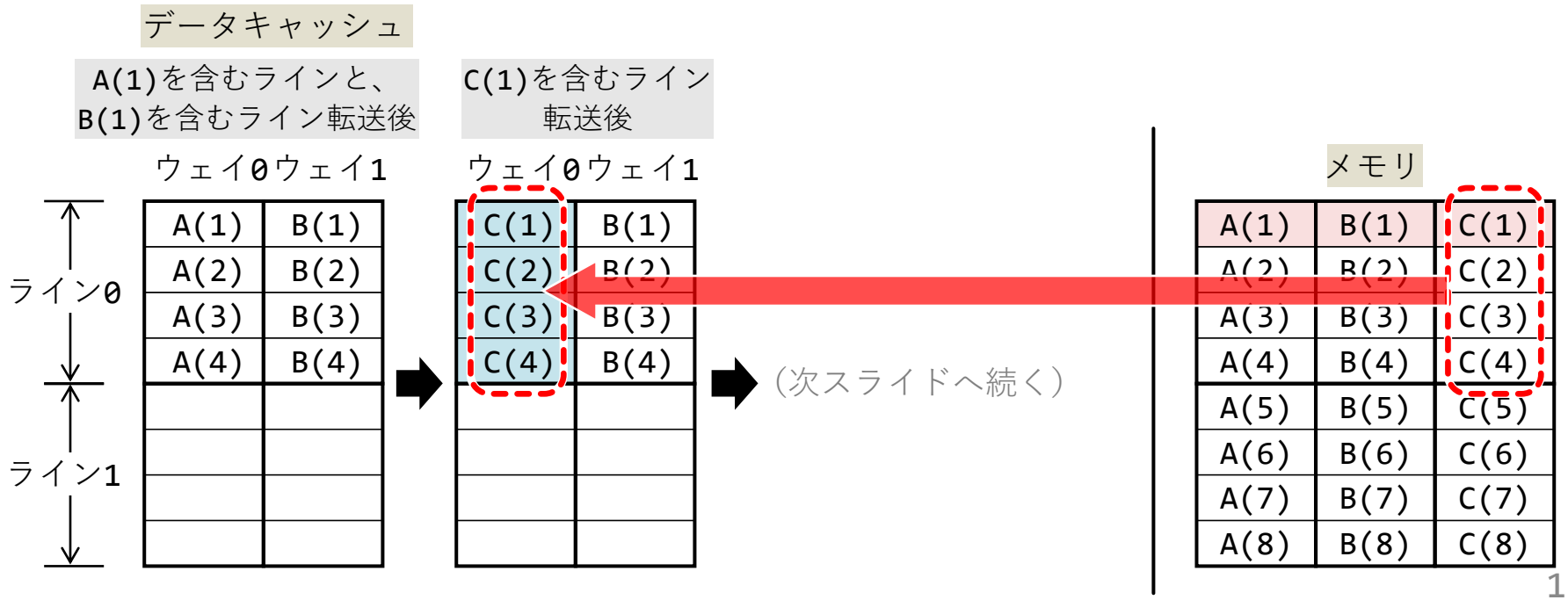


■ キャッシュスラッシング発生の動作説明 [3/6]

- 次に C(1) の参照が必要になり、最初のアクセスなのでキャッシュミスが発生する。
  - このとき、下の一番左の図で、ウェイ0のライン0と、ウェイ1のライン0が、二つとも既にふさがっている。
  - この場合、C(1)～C(4)は、アクセスが古い方であるウェイ0のライン0に転送され、キャッシュの状態は下の左から二番目の図のようになる。

ループの展開図

```
A(1) = A(1) + B(1) + C(1)
A(2) = A(2) + B(2) + C(2)
A(3) = A(3) + B(3) + C(3)
A(4) = A(4) + B(4) + C(4)
A(5) = A(5) + B(5) + C(5)
A(6) = A(6) + B(6) + C(6)
A(7) = A(7) + B(7) + C(7)
A(8) = A(8) + B(8) + C(8)
```



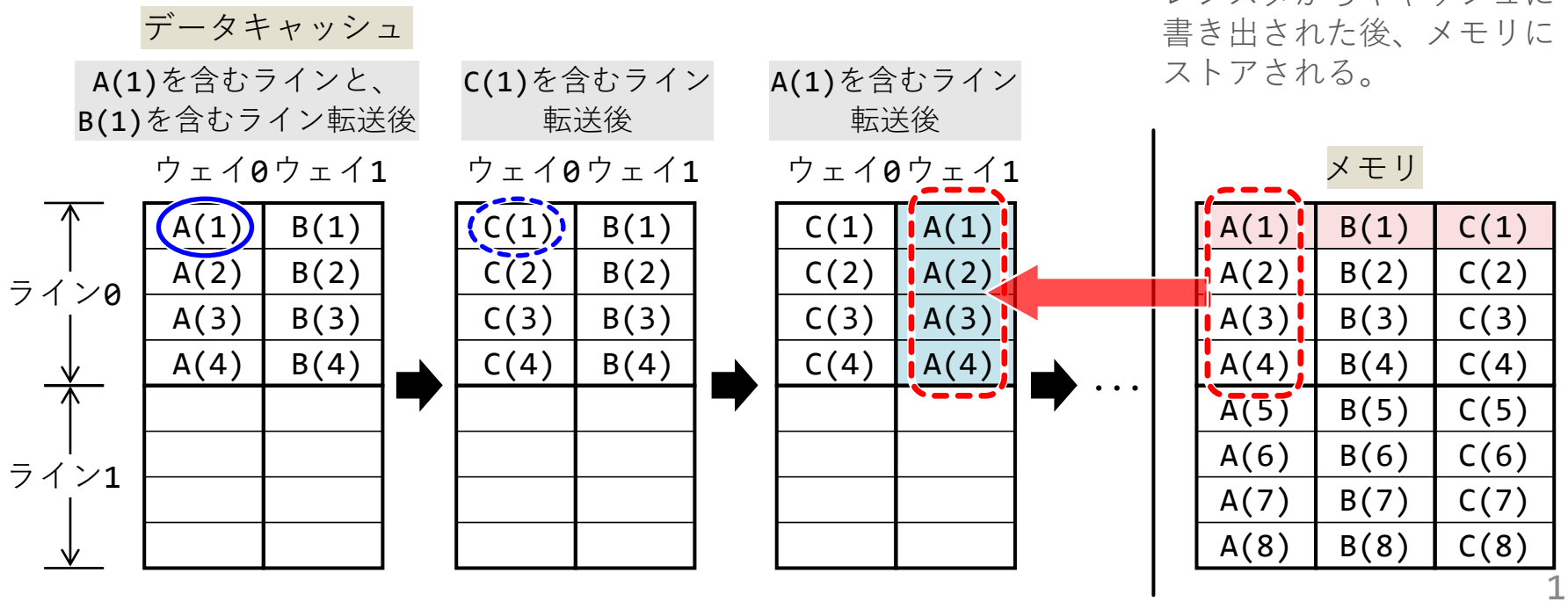
■ キャッシュスラッシング発生の動作説明 [4/6]

- 次に A(1) へのストアが必要になる。
  - ところが下の一番左の時点でキャッシュに入っていた A(1) が、左から二番目の時点ではキャッシュから追い出されている。
  - そのため A(1) へのストアで再びキャッシュミスが発生し、A(1)～A(4) が、左から二番目の図で、古い方のウェイ1のライン0に転送される(*1)。キャッシュの状態は左から三番目の図のようになる。

ループの展開図

```
A(1) = A(1) + B(1) + C(1)
A(2) = A(2) + B(2) + C(2)
A(3) = A(3) + B(3) + C(3)
A(4) = A(4) + B(4) + C(4)
A(5) = A(5) + B(5) + C(5)
A(6) = A(6) + B(6) + C(6)
A(7) = A(7) + B(7) + C(7)
A(8) = A(8) + B(8) + C(8)
```

(*1) ストアの場合、まずメモリからキャッシュに転送され、レジスタからキャッシュに書き出された後、メモリにストアされる。



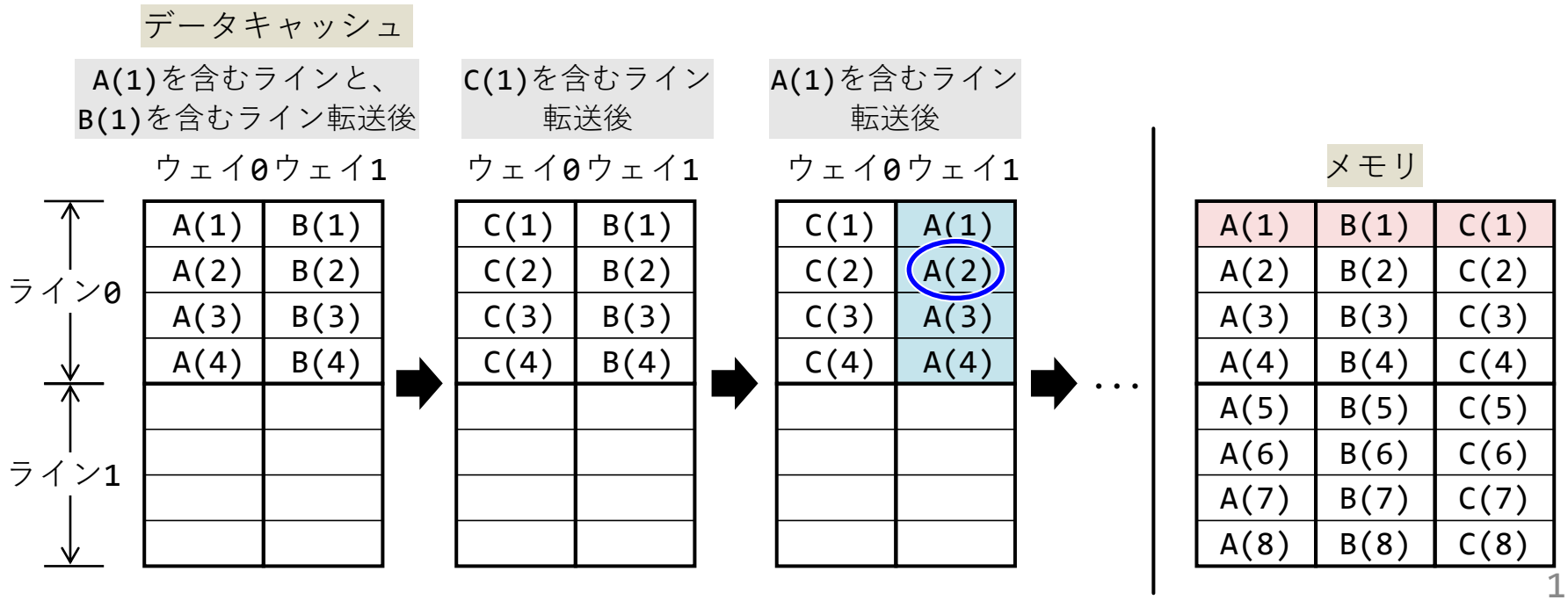
■ キャッシュスラッシング発生の動作説明 [5/6]

■ 次に展開図の2行目で A(2) の参照が必要になる。

■ 下の左から三番目のキャッシュの状態では A(2) はキャッシュに載っている。

ループの展開図

```
A(1) = A(1) + B(1) + C(1)
A(2) = A(2) + B(2) + C(2)
A(3) = A(3) + B(3) + C(3)
A(4) = A(4) + B(4) + C(4)
A(5) = A(5) + B(5) + C(5)
A(6) = A(6) + B(6) + C(6)
A(7) = A(7) + B(7) + C(7)
A(8) = A(8) + B(8) + C(8)
```

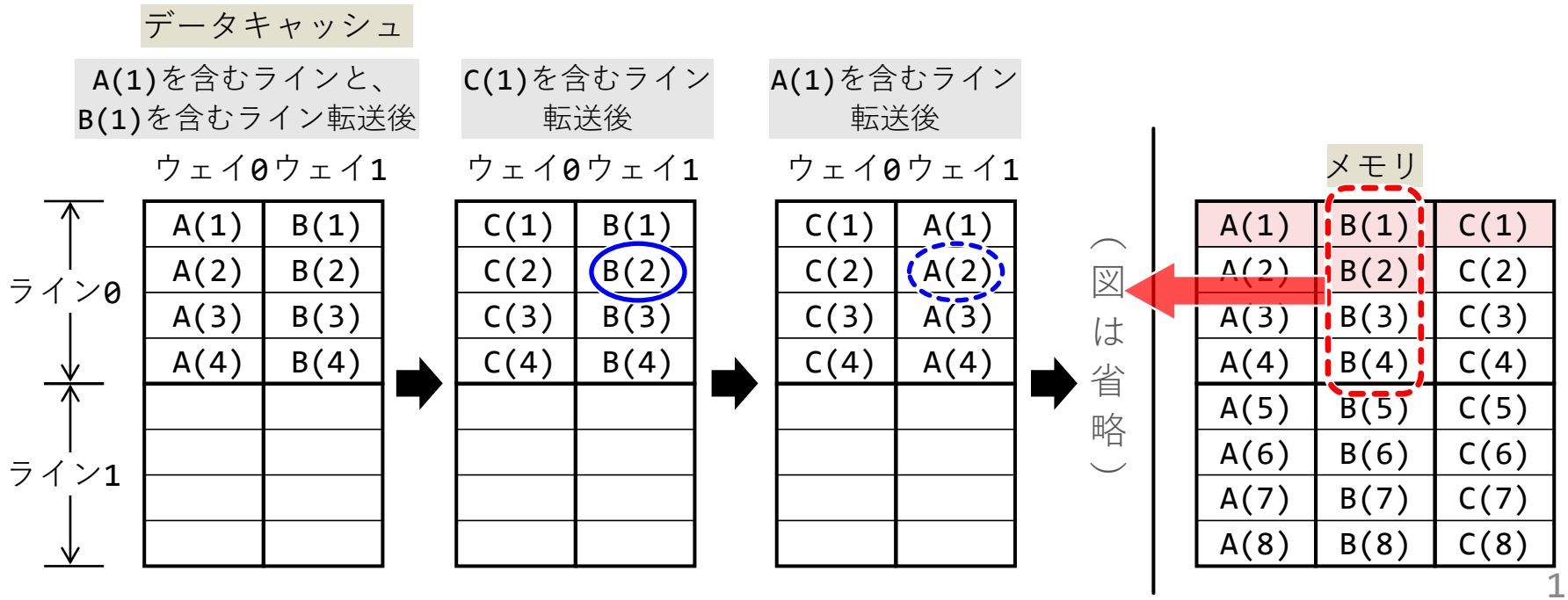


■ キャッシュスラッシング発生の動作説明 [6/6]

- ■ 次に B(2) が必要になる。
  - ■ ところが下の左から二番目の時点でキャッシュに入っていた B(2) が、左から三番目の時点ではキャッシュから追い出されている。
  - ■ そのため B(2) の参照で再びキャッシュミスが発生する。B(2)を含むラインが左から三番目の図のウェイ0に転送される。
- ■ 次に同様に C(2) の参照でキャッシュミスが発生する。
- ■ 以後同様に、ストライド1であるにも関わらず各要素の参照でキャッシュミス発生の可能性が大きくなる。

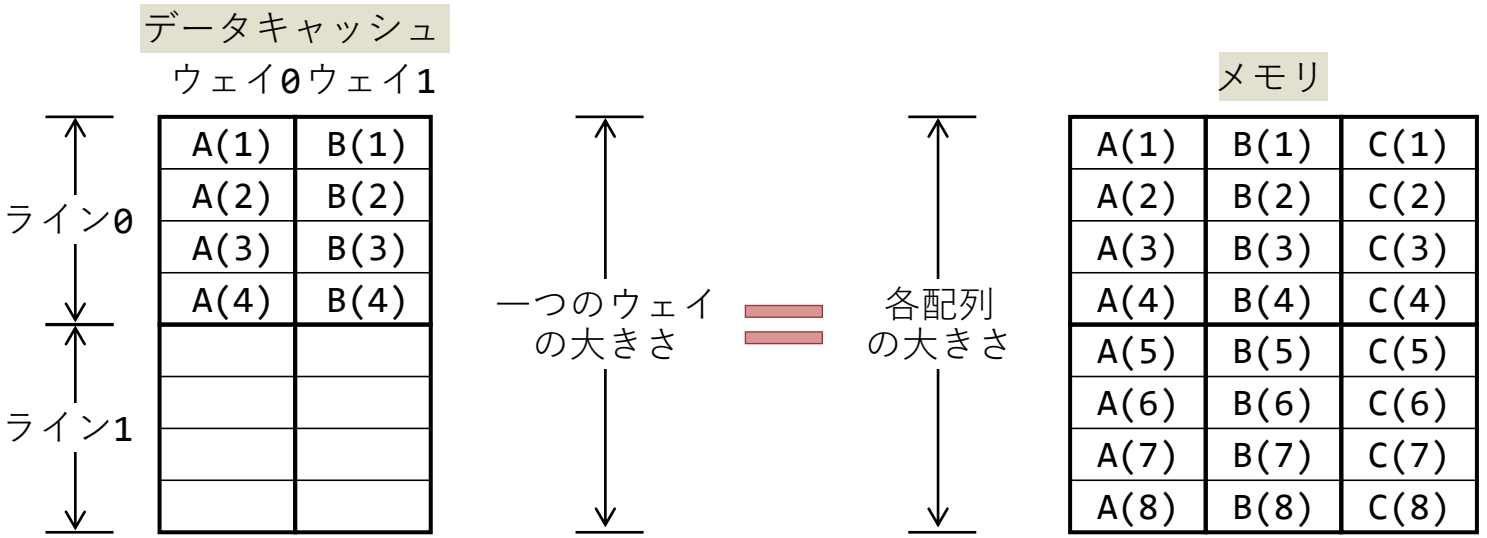
ループの展開図

```
A(1) = A(1) + B(1) + C(1)
A(2) = A(2) + B(2) + C(2)
A(3) = A(3) + B(3) + C(3)
A(4) = A(4) + B(4) + C(4)
A(5) = A(5) + B(5) + C(5)
A(6) = A(6) + B(6) + C(6)
A(7) = A(7) + B(7) + C(7)
A(8) = A(8) + B(8) + C(8)
```



■ キャッシュスラッシング発生の動作振り返り

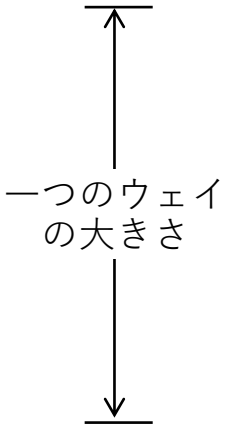
- ■ 前スライドまでの動作を振り返る：
  - ■ メモリの図で、各配列の大きさがちょうどキャッシュの一つのウェイの大きさと同じであると、本プログラム例で同時に計算に使用されるA(1)、B(1)、C(1)が、同じキャッシュライン（本例ではライン0）に入る。
  - ■ その結果、本プログラム例のループは、ストライド1なのに、キャッシュミスが多発する。



- ■ このように特定のキャッシュラインにアクセスが集中することをキャッシュスラッシングと呼ぶ。

■ キャッシュスラッシング発生の留意点

■ 本ループ例と本簡易モデルで、下図のように一つの配列の大きさ（本例では16）がキャッシュの一つのウェイの大きさ（本例では8）の倍数になる場合もキャッシュスラッシングが発生する。



A(1)	A(9)	B(1)	B(9)	C(1)	C(9)
A(2)	A(10)	B(2)	B(10)	C(2)	C(10)
A(3)	A(11)	B(3)	B(11)	C(3)	C(11)
A(4)	A(12)	B(4)	B(12)	C(4)	C(12)
A(5)	A(13)	B(5)	B(13)	C(5)	C(13)
A(6)	A(14)	B(6)	B(14)	C(6)	C(14)
A(7)	A(15)	B(7)	B(15)	C(7)	C(15)
A(8)	A(16)	B(8)	B(16)	C(8)	C(16)

■ 「富岳」の実際のL1Dキャッシュの一つのウェイの大きさは倍精度の要素数で2048である。したがって倍精度の配列の大きさが2048の倍数の場合、キャッシュスラッシングが起きる可能性が高くなる。

■ キャッシュスラッシングをパディングにより回避する [1/4]

本項ではキャッシュスラッシングをパディングにより回避する方法を説明する。本項でも簡易モデルを使用し、前項の簡易モデルでキャッシュスラッシングが起きるプログラム例に対してパディングする方法を説明する。

■ 右の図において、配列A、B、Cの宣言において大きさをそれぞれ2要素分大きくする。

プログラム例（パディング）

```
DOUBLE PRECISION::A(8+2),B(8+2),C(8+2)
DO I=1,8
  A(I) = A(I) + B(I) + C(I)
ENDDO
```

■ メモリでの配置を右の図に示す。図中の★は追加した要素を表す。★で表された要素を追加することをパディングと呼ぶ。

  ■ パディングにより、赤いハイライトで示すように、配列A、B、およびCの一番目の要素の位置が、パディングしない場合と比較して、ずれることに注目して下さい。

■ （次スライドへ続く）

メモリ

A(1)	★A(9)	B(7)	C(5)
A(2)	★A(10)	B(8)	C(6)
A(3)	B(1)	★B(9)	C(7)
A(4)	B(2)	★B(10)	C(8)
A(5)	B(3)	C(1)	★C(9)
A(6)	B(4)	C(2)	★C(10)
A(7)	B(5)	C(3)	
A(8)	B(6)	C(4)	

■ キャッシュスラッシングをパディングにより回避する [2/4]

■ ループの1反復目でA(1)、B(1)およびC(1)が必要になった時点でのデータキャッシュの状態を下の左の図に示す。

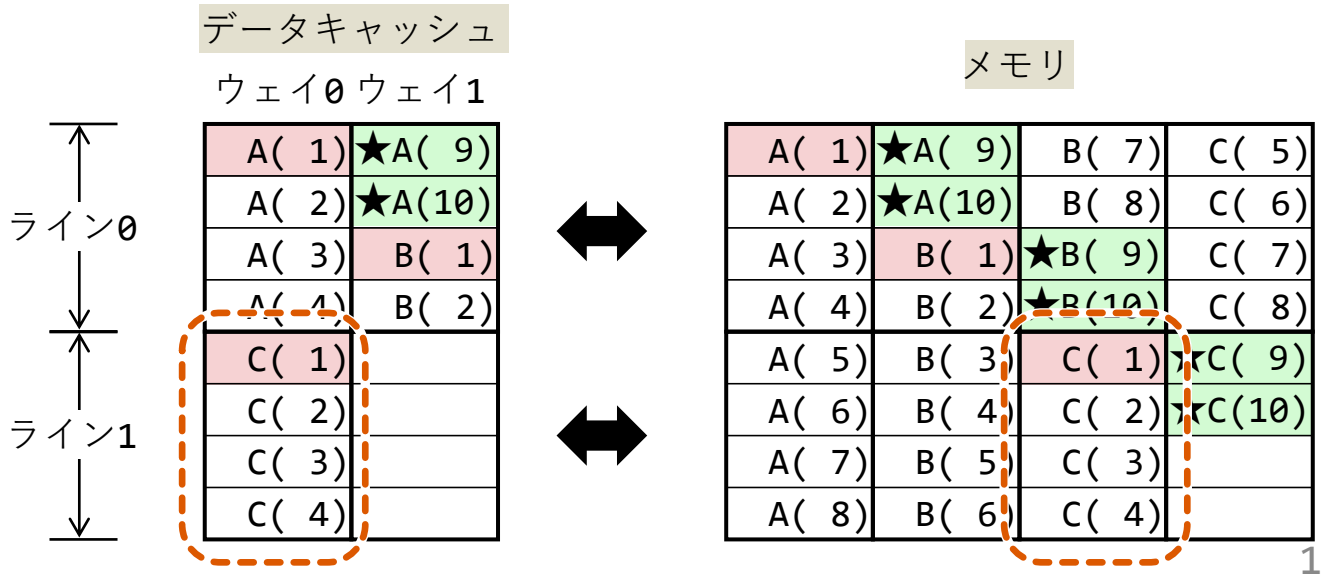
プログラム例（パディング） ［再掲］

```
DOUBLE PRECISION::A(8+2),B(8+2),C(8+2)
DO I=1,8
  A(I) = A(I) + B(I) + C(I)
ENDDO
```

■ パディングにより、メモリ上のA(1)とB(1)はデータキャッシュ上のライン0に入り、C(1)は赤い点線で示すようにライン1に入る。

■ データキャッシュ上、C(1)はライン1に入るため、ライン0にあるA(1)～A(4)はキャッシュから追い出されない。

■ （次スライドへ続く）

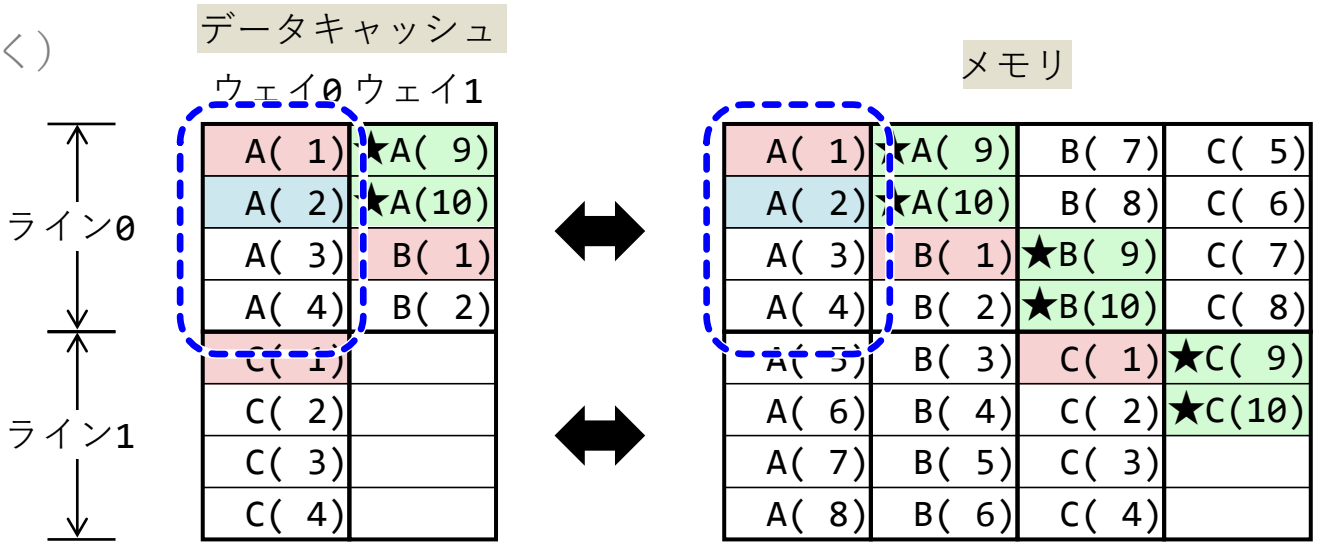


■ キャッシュスラッシングをパディングにより回避する [3/4]

- ループの2反復目でA(2)が必要になった時点でのデータキャッシュの状態を下の左の図に示す。
  - A(1)～A(4)はデータキャッシュ上ライン0に残っているため、A(2)の参照でキャッシュミスは発生しない。
- 同じ2反復目でB(2)の参照が必要になるが、B(2)はウェイ1のライン0に存在するため、キャッシュミスは発生しない。
- 同じ2反復目でC(2)の参照が必要になるが、C(2)はウェイ0のライン1に存在するため、キャッシュミスは発生しない。
- パディング無しでは各要素へのアクセスでキャッシュミスが多発したが、パディング有りの本例ではキャッシュミスの頻度は減少する。
- （次スライドへ続く）

プログラム例（パディング） [再掲]

```
DOUBLE PRECISION::A(8+2),B(8+2),C(8+2)
DO I=1,8
  A(I) = A(I) + B(I) + C(I)
ENDDO
```



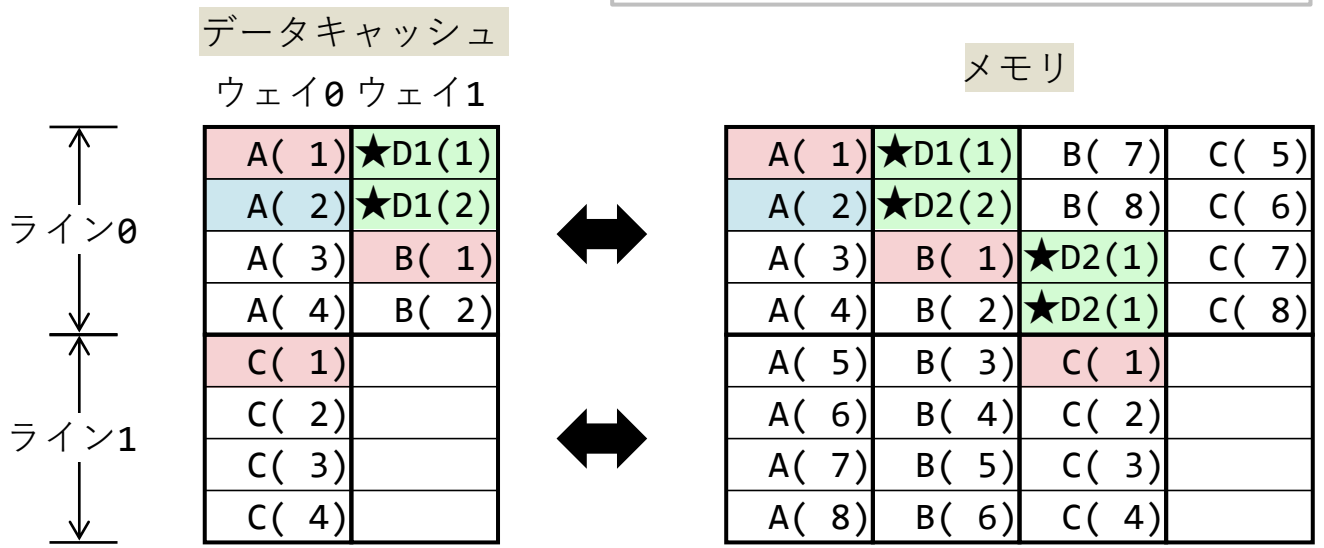
■ キャッシュスラッシングをパディングにより回避する [4/4]

- ■ ダミー配列挿入によるパディング
  - ■ 配列A、BおよびC自体は大きくせずにパディングを行いたい場合、図のプログラム例に示すように、配列AとB、BとCの間にダミーの配列（本例ではそれぞれD1およびD2）を挿入する。
  - ■ ループの2反復目でA(2)が必要になった時点でのデータキャッシュの状態を下の左の図に示す。

以上で、キャッシュスラッシングをパディングにより回避する方法の説明を終わる。

プログラム例
（ダミー配列によるパディング）

```
DOUBLE PRECISION::A(8),B(8),C(8)
DOUBLE PRECISION::D1(2),D2(2)
COMMON/COM/A,D1,B,D2,C
DO I=1,8
  A(I) = A(I) + B(I) + C(I)
ENDDO
```



■ キャッシュスラッシングをループ分割により回避する

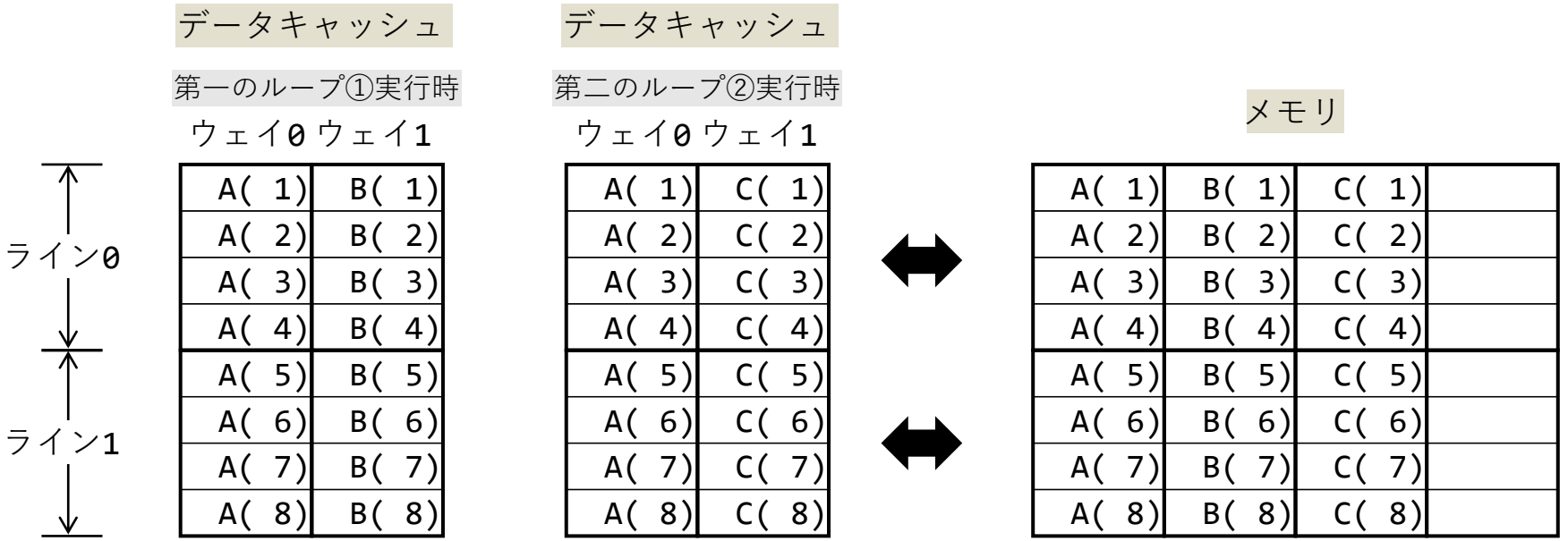
■ 右のプログラム例のようにループを分割し、ひとつのループで使用する配列数を減らす。

- ■ ①で示す第一のループを実行したときのキャッシュの状態を下の一番左の図に示す。図に示されるようにキャッシュスラッシングを阻止することができる。
- ■ ②で示す第二のループを実行したときのキャッシュの状態を下の左から二番目の図に示す。図に示されるようにキャッシュスラッシングを阻止することができる。

プログラム例

ループ分割により簡易モデルでのキャッシュスラッシング回避

```
DOUBLE PRECISION::A(8),B(8),C(8)
DO I=1,8                                    ①
  A(I) = A(I) + B(I)
ENDDO
DO I=1,8                                    ②
  A(I) = A(I) + C(I)
ENDDO
```



■ キャッシュスラッシングが実際発生するプログラム例

これまではキャッシュスラッシングの動作を簡潔に説明するためキャッシュの簡易モデルで説明してきた。ここからは「富岳」でキャッシュスラッシングが実際に発生する場合を検討する。まず、本スライドでキャッシュスラッシングが実際に発生するプログラム例を示す。さらに前述のキャッシュスラッシング回避策を適用したプログラム例を以降のスライドで示す。最適化のコンパイルオプションは-Kfastとし、逐次実行する。プログラム例について実行結果例を示す。

■ キャッシュスラッシングが発生するプログラム例を以下の図に示す。

- L1Dキャッシュのウェイ数は4である。
- L1Dキャッシュの一つのウェイの大きさは倍精度実数で2048要素に相当する。
- 各反復IでA(I)～H(I)の8個の要素の参照が必要となる。各配列の大きさは2048であり、L1Dキャッシュの一つのウェイの大きさに等しい。そのため、各反復Iでの要素A(I)～H(I)はすべて同一のラインに属する。
- 各反復での計算に必要な同一のラインの数が8であり、これはL1Dキャッシュのウェイ数4を超えるので、キャッシュスラッシングが発生する。

プログラム例

```
INTEGER,PARAMETER::N=2048
DOUBLE PRECISION::A(N),B(N),C(N),D(N)
DOUBLE PRECISION::E(N),F(N),G(N),H(N)
COMMON/COM/A,B,C,D,E,F,G,H
DO I=1,N
  A(I)=A(I)+B(I)+C(I)+D(I)+E(I)+F(I)+G(I)+H(I)
ENDDO
```

←一つの配列の大きさがL1Dキャッシュの一つのウェイの大きさに等しい！

←各反復で使用される配列データが属するライン番号は8個全て同じ！

■ キャッシュスラッシングをパディングにより回避するプログラム例

キャッシュスラッシングをパディングで回避するプログラム例を以下の図に示す。

■ パラメータNDはパディングの大きさである。以降のスライドでNDの値を色々変えて実験する。ND=0ととれば、前スライドのパディングなしのプログラム例と意味的に同じである。

プログラム例

```
INTEGER,PARAMETER::N=2048
INTEGER,PARAMETER::ND=512 ←パディングの大きさ。
DOUBLE PRECISION::A(N+ND),B(N+ND),C(N+ND),D(N+ND)
DOUBLE PRECISION::E(N+ND),F(N+ND),G(N+ND),H(N+ND)
COMMON/C_NI/A,B,C,D,E,F,G,H
DO I=1,N
  A(I)=A(I)+B(I)+C(I)+D(I)+E(I)+F(I)+G(I)+H(I)
ENDDO
```

■ キャッシュスラッシングをループ分割により回避するプログラム例

キャッシュスラッシングをループ分割で回避するプログラム例を以下の図に示す。

- L1Dキャッシュのウェイ数は4であるので、「各反復で使用される配列データが属するラインについて、同じライン番号となるようなラインの数が4以下になるよう」(*1)、元のループを分割する。
  - 各反復で使用される配列の数が4以下ならば、この条件(*1)は満たされる。
  - 本例では右の図に示すように元のループを三つに分割する。
  - 同一のライン数4以下の条件をみたすような他の分割方法も可能である。
- 右の図で指示行なしのプログラムを-Kfastオプションでコンパイルした場合、-Kfastで有効になる（正確には-O2オプション以上で有効になる）-Kloop_fusion により、隣接ループが融合される。このループ融合を防ぐため、指示行 !OCL LOOP_NOFUSIONを指定した（指示行を有効にするため-Koclも指定する）。

プログラム例

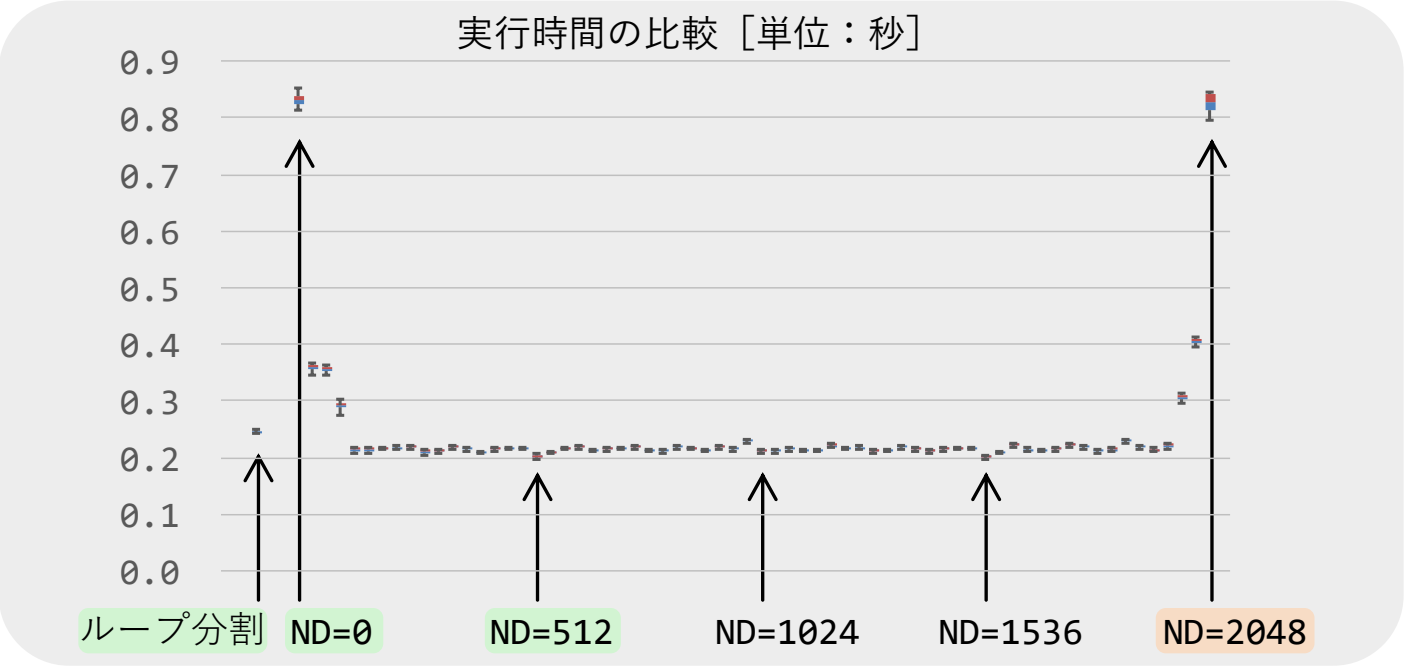
```
  INTEGER,PARAMETER::N=2048
  DOUBLE PRECISION::A(N),B(N),C(N),D(N)
  DOUBLE PRECISION::E(N),F(N),G(N),H(N)
  COMMON/C_NI/A,B,C,D,E,F,G,H
!-------------------- 1
!OCL LOOP_NOFUSION
    DO I=1,N
      A(I)=A(I)+B(I)+C(I)+D(I)
    ENDDO
!-------------------- 2
!OCL LOOP_NOFUSION
    DO I=1,N
      A(I)=A(I)+E(I)+F(I)
    ENDDO
!-------------------- 3
!OCL LOOP_NOFUSION
    DO I=1,N
      A(I)=A(I)+G(I)+H(I)
    ENDDO
!--------------------
```

■ プログラム例の実行結果 [1/3]

NDの値をND=0（パディングなし）、ND=16、およびND=32から2048まで32要素毎（1ライン毎）、およびループ分割したときの実行時間（経過時間）(*1)を以下の図に示す（複数回計測の箱ひげ図）。　(*1) 計測のためのループの繰り返し数NLOOPを100000とする。

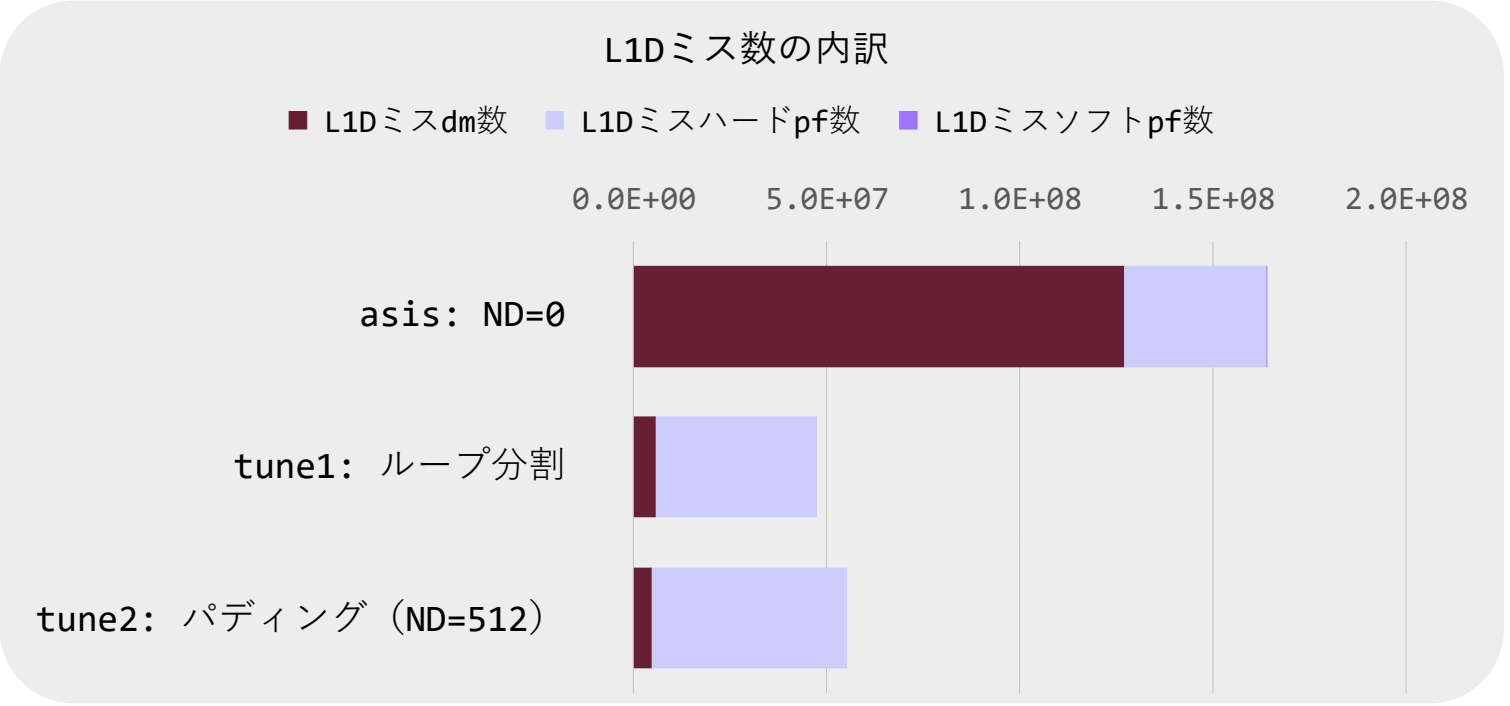
横軸の一番左がループ分割の結果であり、左から2番目がND=0の結果、左から3番目以降がNDの値が非零のパディングの結果である。

- ND=2048はちょうどL1Dキャッシュの1ウェイ分パディングすることであり、経過時間はパディングなしND=0の場合と同等であることが確認される。
- 本ループ例ではNDの値が256、512、1024、1536、および1792のときが同程度で最も高速となった。
- 以降のスライドでは、ND=0、ND=512、およびループ分割の3ケースのみに絞り比較結果を示す。



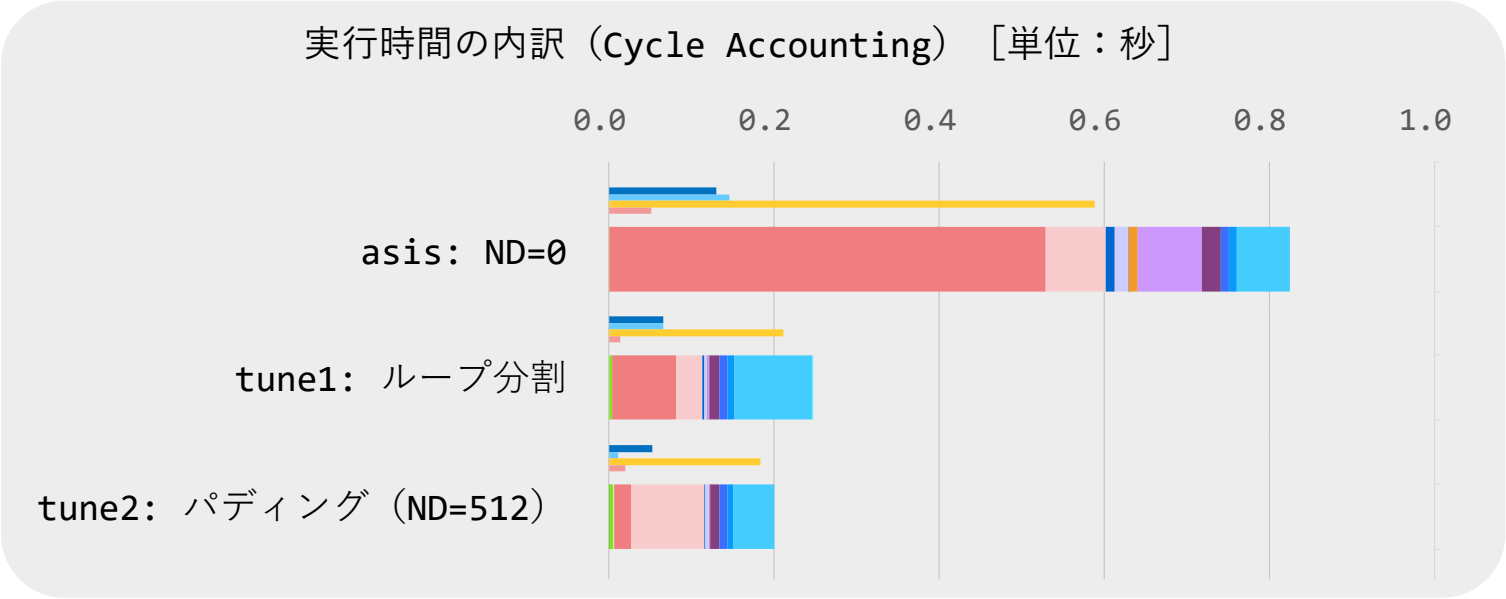
■ プログラム例の実行結果 [2/3]

CPU性能解析レポートからL1Dキャッシュミス数の内訳の比較結果を示す。ループ分割（tune1）またはパディング（tune2）により、L1Dキャッシュミス数（■、■、および■の合計）およびデマンドミス数（■）が減少したことが確認される。



■ プログラム例の実行結果 [3/3]

CPU性能解析レポートから実行時間の内訳（Cycle Accounting）を以下の図に示す。1ページ前で見たように、ループ分割またはパディングによりL1Dキャッシュミスのデマンドミス数が減少した。この事に対応し、以下の図のtune1とtune2ではL2キャッシュアクセス待ちの時間（■）が改善されたことが確認される。



- 各種最適化手法
  - キャッシュチューニングのための基本事項
    - 避けるべきL1Dキャッシュミス多発の状況

本小節の内容

    - キャッシュスラッシング
    - スレッド並列におけるFalse Sharing

■ False Sharing（1）

本項ではFalse Sharingについて説明する。説明を簡単にするために小さな配列で説明する。False Sharingが起きるループ例を以下の左側の図に示す。

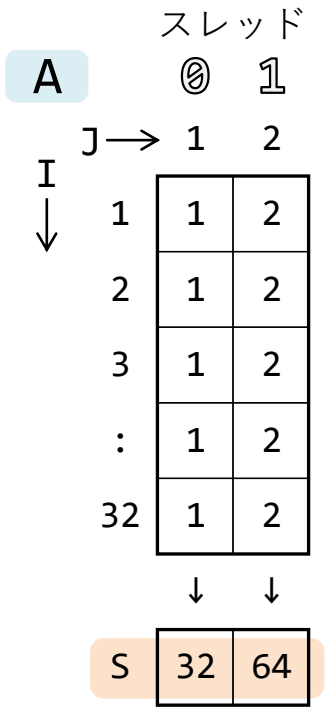
■ 外側のJループをOpenMPスレッド並列化する。

本ループ例で使用する配列AおよびSを以下の右側の図に示す。

■ スレッド数2で実行した場合、スレッド⓪は配列Aの1列目の値をS(1)に累積し、スレッド①は配列Aの2列目の値をS(2)に累積する。

ループ例

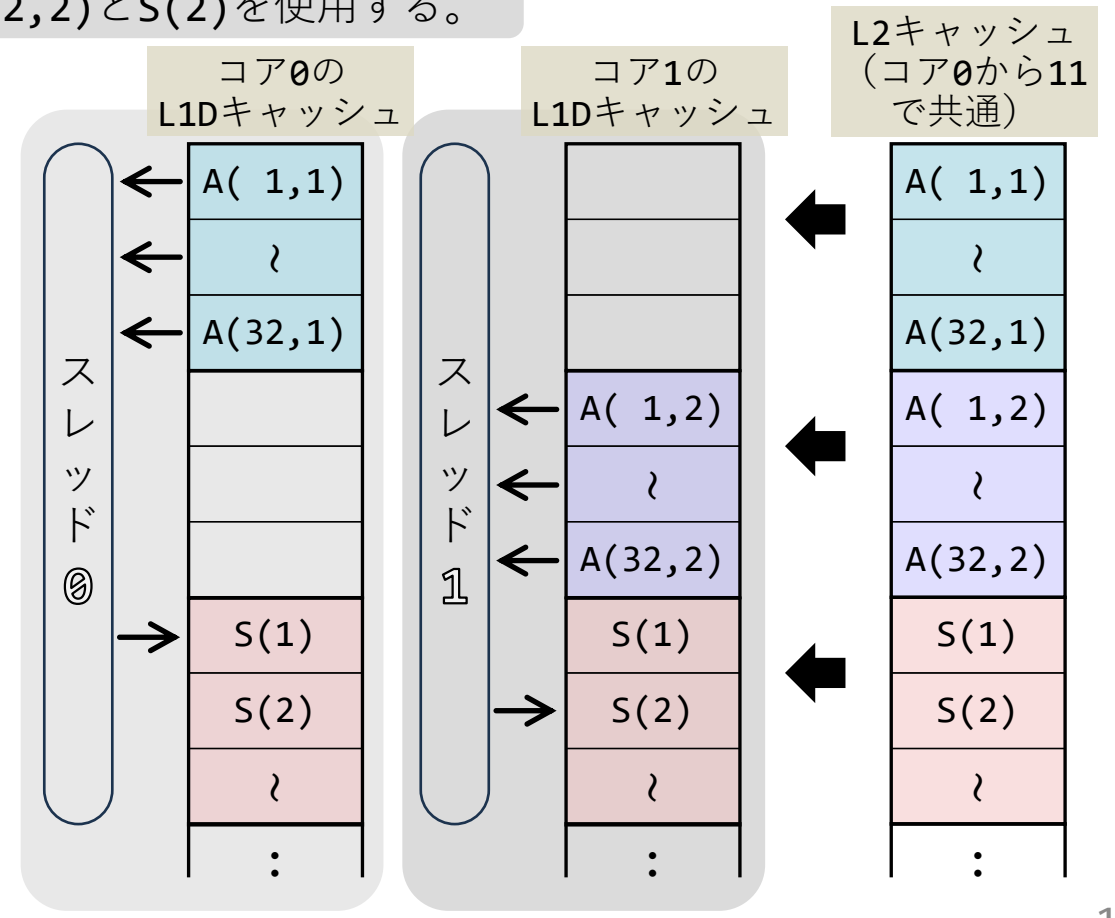
```
  INTEGER,PARAMETER::NI=32, NT=2
  DOUBLE PRECISION::A(NI,NT),S(NT)

!$OMP PARALLEL DEFAULT(NONE) SHARED(A,S) PRIVATE(I,J)
!$OMP DO
    DO J=1,NT ←スレッド並列化
      S(J) = 0D0
      DO I=1,NI
        S(J) = S(J) + A(I,J)
      ENDDO
    ENDDO
!$OMP END DO
!$OMP END PARALLEL
```



■ False Sharing（1）：配列AとSのメモリおよびデータキャッシュ内での配置

- 配列AとSがメモリ内（正確にはL2キャッシュ内）で一番右下図の様に配置されているとする。
- スレッド⓪がコア0で、スレッド①がコア1で動作する場合：
  - スレッド⓪はA(1,1)～A(32,1)とS(1)を使用する。
  - スレッド①はA(1,2)～A(32,2)とS(2)を使用する。

- S(1)、S(2)とその近隣の変数（赤でハイライトした部分）は、コア0とコア1の両方のL1Dキャッシュに入っている。... (*1)
- 上記(*1)のような場合、両方のキャッシュラインの内容を常に同じにする必要があるため、キャッシュラインの動作は以降のスライドのようになる。



■ False Sharing（1）：キャッシュラインの動作 [1/2]

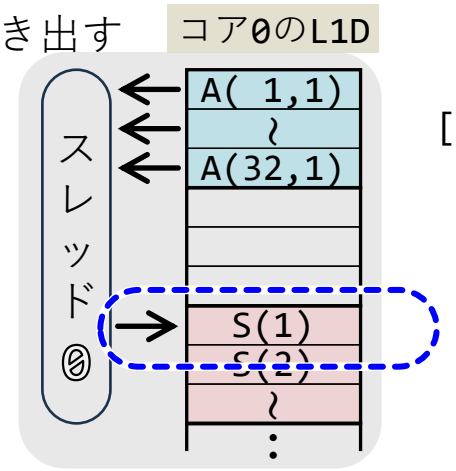
- ■ 最初の状態が右の一番上の[A]のようであるとする。

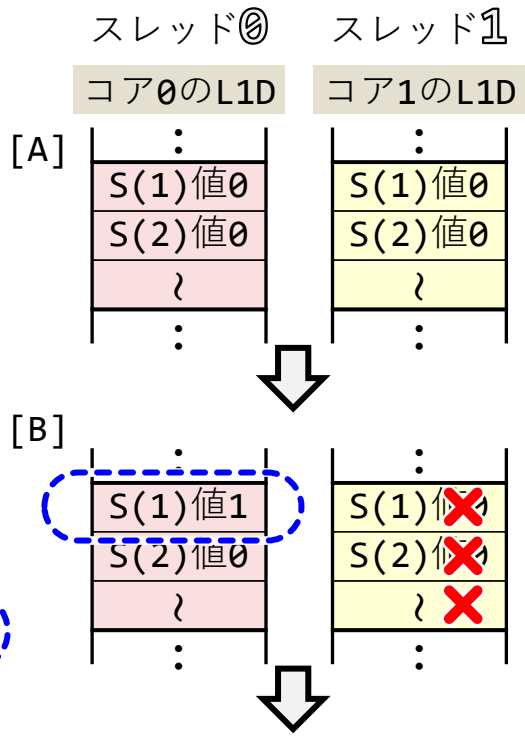
⇩

- ■ まず、青い点線で示すように、スレッド⓪がコア0のS(1)に値（本例では値1）を書き出すとする。
  - ■ 書き出し後のコア0の状態を[B]に示す。
  - ■ この時点で、コア0とコア1のキャッシュラインの内容が異なってしまったため、[B]×印で示すように、コア1のキャッシュラインを無効（使用禁止）にする。

⇩

- ■ （次スライドへ続く）



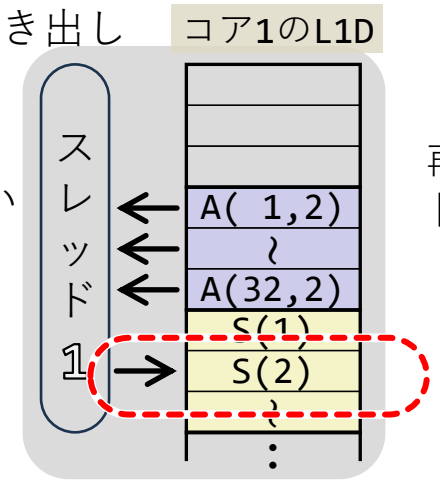


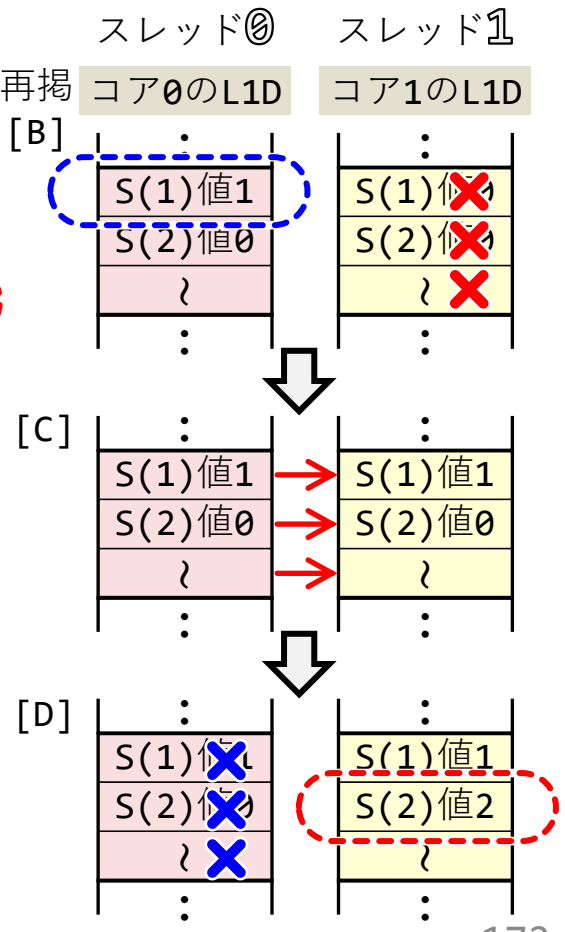
■ False Sharing（1）：キャッシュラインの動作 [2/2]

■ 次に、赤い点線で示すように、スレッド1がコア1のS(2)に値（本例では値2）を書き出したいとする。

■ ところが、再掲[B]で、コア1のキャッシュラインが無効になっているため、書き出すことができない。

■ この場合、まず図[C]→印で示すように、コア0の内容をコア1にコピーする。これによりコア0とコア1のキャッシュラインの内容が同じとなり、コア1のキャッシュラインは有効になる。





■ コア1のキャッシュライン有効後、図[D]の赤い点線で示すように、スレッド1がコア1のS(2)に値を書き出す。

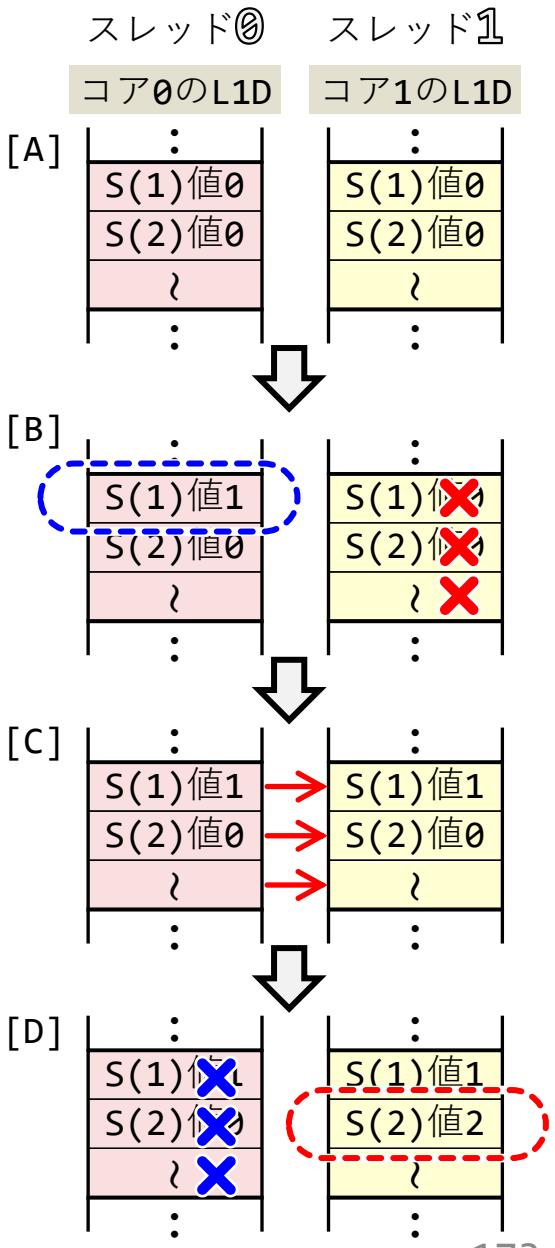
■ この時点でコア0とコア1のキャッシュラインの内容が異なってしまった。[D]×印で示すように、今度はコア0のキャッシュラインを無効（使用禁止）にする。

以上でキャッシュラインの動作説明を終わる。

■ False Sharing（1）：注意事項 [1/2]

- False Sharingが発生すると、これまで見たように、各コアのL1Dキャッシュの整合性を取るためのオーバーヘッドが入るため、速度が低下する。
- プロファイラ（CPU性能解析レポート）で測定した場合、L1Dミス率が高くなる(*1)。

(*1) 具体例を以降の「 False Sharing（2）」で扱うループ例の実行結果で示します。



■ False Sharing（1）：注意事項 [2/2]

■ なお、本ループ例（再掲図参照）の場合、実際には、コンパイラが最適化を行うため、最適化オプション「-O0」を指定したときのみFalse Sharingが発生する。

ループ例（再掲）

```
  INTEGER,PARAMETER::NI=32, NT=2
  DOUBLE PRECISION::A(NI,NT),S(NT)

!$OMP PARALLEL DEFAULT(NONE) SHARED(A,S) PRIVATE(I,J)
!$OMP DO
    DO J=1,NT ←スレッド並列化
      S(J) = 0D0
      DO I=1,NI
        S(J) = S(J) + A(I,J)
      ENDDO
    ENDDO
!$OMP END DO
!$OMP END PARALLEL
```

■ 「-Kfast」オプションを指定した場合でもFalse Sharingが発生する例については「 False Sharing（2）」で説明する。

■ False Sharingの回避方法

False Sharingが発生する本ループ例について、False Sharingの回避方法を以下に二つ示す。

ループ例（再掲）

```
  INTEGER,PARAMETER::NI=32, NT=2
  DOUBLE PRECISION::A(NI,NT),S(NT)

!$OMP PARALLEL DEFAULT(NONE) SHARED(A,S) PRIVATE(I,J)
!$OMP DO
    DO J=1,NT ←スレッド並列化
      S(J) = 0D0
      DO I=1,NI
        S(J) = S(J) + A(I,J)
      ENDDO
    ENDDO
!$OMP END DO
!$OMP END PARALLEL
```

- 回避方法1：配列Sへの加算の代わりにプライベート変数を導入する方法
- 回避方法2：配列Sをパディングする方法

以降のスライドで順に説明する。

■ False Sharingの回避方法1：プライベート変数利用

回避方法1として、配列Sへの加算の代わりにプライベート変数を利用する方法のループ例を右下の図に示す。

- 改善後のループ例（右図）では、スレッド⓪と①での変数 SS は、コア0とコア1のレジスタにそれぞれ置かれるため、変数 SS に関してFalse Sharingは発生しない。

ループ例（再掲）：改善前

```
  INTEGER,PARAMETER::NI=32, NT=2
  DOUBLE PRECISION::A(NI,NT),S(NT)

!$OMP PARALLEL DEFAULT(NONE) &
!$OMP & SHARED(A,S) PRIVATE(I,J)
!$OMP DO
    DO J=1,NT ←スレッド並列化
      S(J) = 0D0
      DO I=1,NI
        S(J) = S(J) + A(I,J)
      ENDDO
    ENDDO
!$OMP END DO
!$OMP END PARALLEL
```

ループ例：改善後（回避方法1）

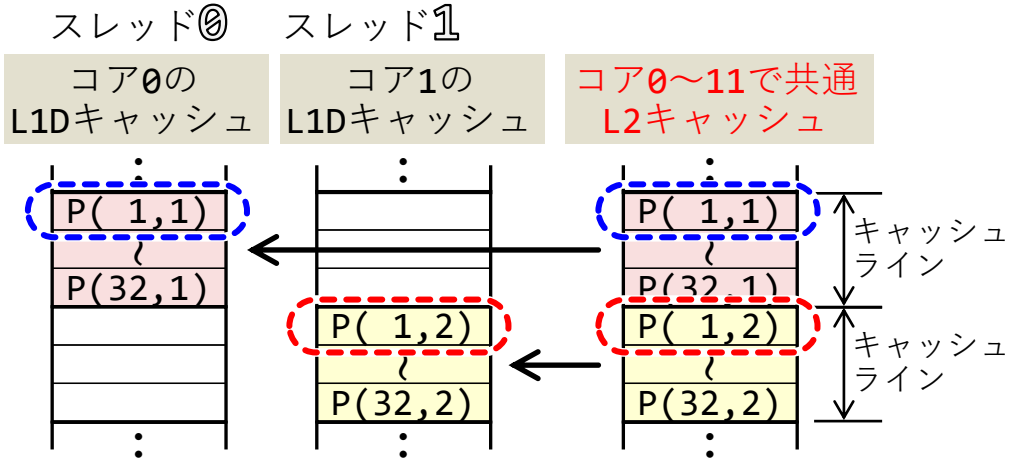
```
  INTEGER,PARAMETER::NI=32, NT=2
  DOUBLE PRECISION::A(NI,NT),S(NT),SS

!$OMP PARALLEL DEFAULT(NONE) &
!$OMP & SHARED(A,S) PRIVATE(I,J,SS)
!$OMP DO
    DO J=1,NT ←スレッド並列化
      SS = 0D0
      DO I=1,32
        SS = SS + A(I,J)
      ENDDO
      S(J) = SS
    ENDDO
!$OMP END DO
!$OMP END PARALLEL
```

- 改善前のループ例（左図）ではFalse Sharingが最適化オプション「-O0」のみで発生し、「-O1」以上では発生しないのは、コンパイラが右図への最適化を自動的に行っているからだと思われる。

■ False Sharingの回避方法2：パディング利用

- 一つのキャッシュラインのサイズは256バイトであり、1キャッシュライン当たりの要素数は倍精度実数の場合32個である。
- 改善前のループ例（前スライド左図）では配列要素 S(1) と S(2) は同じキャッシュラインに載る。
- 配列要素 S(1) と S(2) が、異なるキャッシュラインに載るようにすれば、False Sharingを防ぐことができる。そこで、配列 S(2) をパディングし下の右図で配列 P(32,2) を導入する（配列 P が倍精度なので1次元目の寸法を32とする）。
  - 下の左側二つの図に示すように、配列要素 P(1,1) はコア0のL1Dキャッシュに載り、P(1,2) はコア1のL1Dキャッシュに載るため、配列 P に関してFalse Sharingは発生しない。
  - 配列要素 P(2:32,1) および P(2:32,2) がパディングされた部分であり、使用されない。

改善後のキャッシュの様子



ループ例：改善後（回避方法2）

```
  INTEGER,PARAMETER::NI=32, NT=2
  DOUBLE PRECISION::A(NI,NT)
  DOUBLE PRECISION::P(NI,NT),S(NT)

!$OMP PARALLEL DEFAULT(NONE) &
!$OMP & SHARED(A,S,P) PRIVATE(I,J)
!$OMP DO
    DO J=1,NT ←スレッド並列化
      P(1,J) = 0D0
      DO I=1,32
        P(1,J) = P(1,J) + A(I,J)
      ENDDO
      S(J) = P(1,J)
    ENDDO
!$OMP END DO
!$OMP END PARALLEL
```

■ False Sharing（2）：-Kfastオプション指定でもFalse Sharingが発生する例

ここからは最適化オプション-Kfastを指定してもFalse Sharingが発生する例を示す。

- 配列Aに入っている値の最大値AMAX（本例では8）とその位置ILOC（本例では3）を求めるループをスレッド並列化する場合を考える。逐次のループ例を右上の図に示す。
- 右上の図の逐次ループのOpenMPスレッド並列化を考える。
  - 最大値だけを求めるのであれば、右下の図のようにreduction節の指定だけで済む
  - むかし、位置も求める必要がある場合、右上のプログラムを修正する必要がある。
- 以降のスライドでOpenMPスレッド並列化したプログラム例を示し、動作を説明する。

ループ例：最大値と位置を求める（逐次版）

```
INTEGER::N,LOC,I
DOUBLE PRECISION::A(N),AMAX
LOC = 0
AMAX = 0D0
DO I=1,N
  IF (A(I) > AMAX) THEN
    LOC = I
    AMAX = A(I)
  ENDIF
ENDDO
```

	A
1	4
2	2
3	8
4	1
5	5
6	7
7	3
8	6

LOC	AMAX
3	8

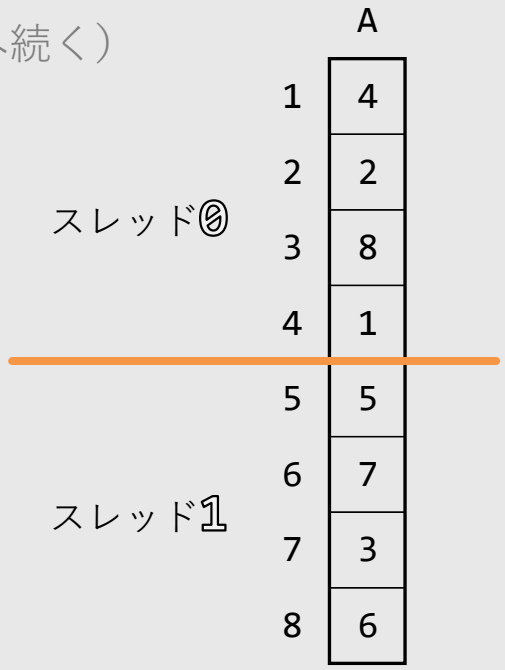
ループ例：最大値のみ求める（スレッド並列版）

```
    AMAX = 0D0
!$OMP PARALLEL DO REDUCTION(MAX:AMAX)
    DO I=1,N
      AMAX = MAX(AMAX,A(I))
    ENDDO
!$OMP END PARALLEL DO
```

■ False Sharing（2）：並列化したプログラムの動作 [1/4]

まず並列化した右の図のプログラム（False Sharingが発生する）の動作を、スレッド数を2として説明する。

- ■ ①で二つのスレッドが起動する。
- ■ ②で各スレッドは自分のスレッド番号をプライベート変数 ID に入れる。
- ■ ③で④のループが並列に実行される。
- ■ （次スライドへ続く）



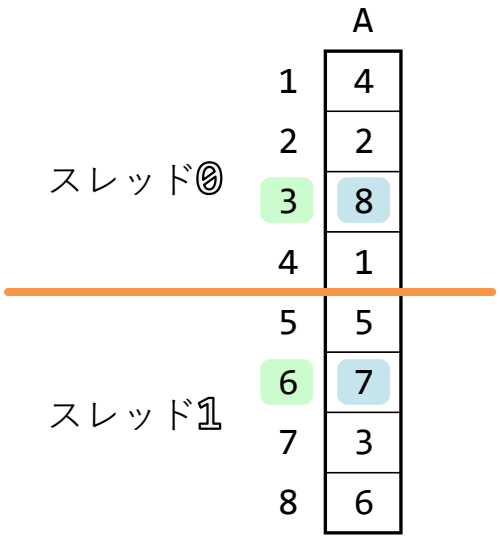
ループ例（スレッド並列）：False Sharing発生

```
!$  USE OMP_LIB
  INTEGER,PARAMETER::N=8
  INTEGER,PARAMETER::NT=2  ←スレッド数
  DOUBLE PRECISION::A(N),AMAX
  INTEGER::LOC
  DOUBLE PRECISION::AMAX_L(0:NT-1)
  INTEGER::LOC_L(0:NT-1)
  INTEGER::ID,I,LL

!$OMP PARALLEL DEFAULT(NONE) &
!$OMP & SHARED(A,LOC_L,AMAX_L) &
!$OMP & PRIVATE(I,ID)                    ①
    ID = OMP_GET_THREAD_NUM()            ②
    LOC_L(ID) = 0
    AMAX_L(ID) = 0D0
!$OMP DO                                 ③
    DO I=1,N  ←スレッド並列化            ④
      IF (A(I) > AMAX_L(ID)) THEN
        LOC_L(ID) = I
        AMAX_L(ID) = A(I)
      ENDIF
    ENDDO
!$OMP END DO
!$OMP END PARALLEL
```

（以下、2ページ後で説明）

■ False Sharing（2）：並列化したプログラムの動作 [2/4]

- ⑤で、スレッド⓪は、配列Aのうち自分の担当部分の最大値の位置をLOC_L(⓪)に入れる（スレッド①も同様な動作）。
- ⑥で、スレッド⓪は、配列Aのうち自分の担当部分の最大値をAMAX_L(⓪)に入れる（スレッド①も同様な動作）。
- （次スライドへ続く）



	LOC_L	AMAX_L
⓪	3	8
①	6	7

ループ例（スレッド並列）：False Sharing発生［続］

```
!$  USE OMP_LIB
  INTEGER,PARAMETER::N=8
  INTEGER,PARAMETER::NT=2  ←スレッド数
  DOUBLE PRECISION::A(N),AMAX
  INTEGER::LOC
  DOUBLE PRECISION::AMAX_L(0:NT-1)
  INTEGER::LOC_L(0:NT-1)
  INTEGER::ID,I,LL

!$OMP PARALLEL DEFAULT(NONE) &
!$OMP & SHARED(A,LOC_L,AMAX_L) &
!$OMP & PRIVATE(I,ID)                  ①
    ID = OMP_GET_THREAD_NUM()          ②
    LOC_L(ID) = 0
    AMAX_L(ID) = 0D0
!$OMP DO                               ③
    DO I=1,N ←スレッド並列化           ④
      IF (A(I) > AMAX_L(ID)) THEN
        LOC_L(ID) = I                  ⑤
        AMAX_L(ID) = A(I)              ⑥
      ENDIF
    ENDDO
!$OMP END DO
!$OMP END PARALLEL
```

（以下、次ページで説明）

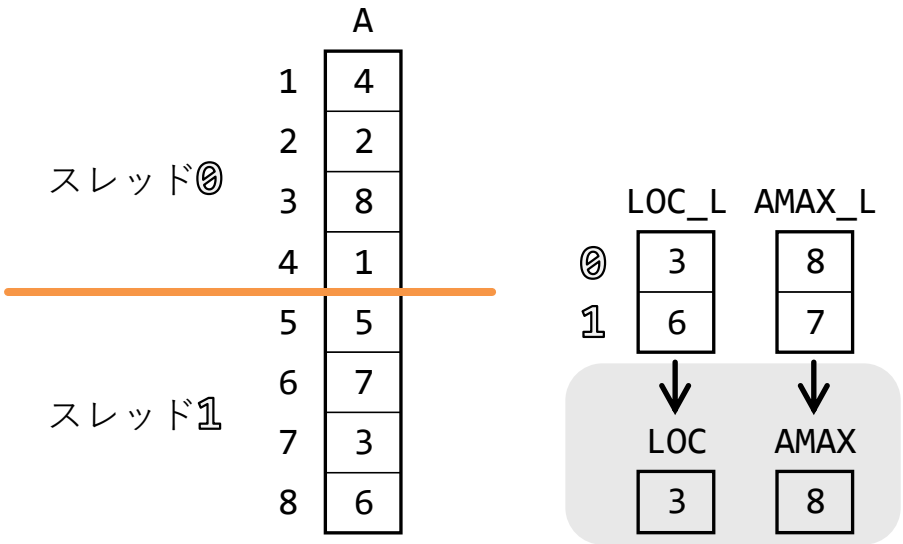
■ False Sharing（2）：並列化したプログラムの動作 [3/4]

ループ例（スレッド並列）：False Sharing発生［続］

```
 （省略）
!$OMP PARALLEL DEFAULT(NONE) &
!$OMP & SHARED(A,LOC_L,AMAX_L) &
!$OMP & PRIVATE(I,ID)                    ①
    ID = OMP_GET_THREAD_NUM()            ②
    LOC_L(ID) = 0
    AMAX_L(ID) = 0D0
!$OMP DO                                 ③
    DO I=1,N                             ④
      IF (A(I) > AMAX_L(ID)) THEN
        LOC_L(ID) = I                    ⑤
        AMAX_L(ID) = A(I)                ⑥
      ENDIF
    ENDDO
!$OMP END DO
!$OMP END PARALLEL                       ⑦
    LOC = 0
    AMAX = 0D0
    DO ID=0,NT-1 ←逐次処理
      IF (AMAX_L(ID) > AMAX) THEN
        LOC = LOC_L(ID)                  ⑧
        AMAX = AMAX_L(ID)                ⑨
      ENDIF
    ENDDO
```

■ ⑦で並列部分が終了した後、⑧で全スレッドの最大値の位置LOCを、⑨で全スレッドの最大値AMAXを、逐次処理で求める。

■ （次スライドへ続く）



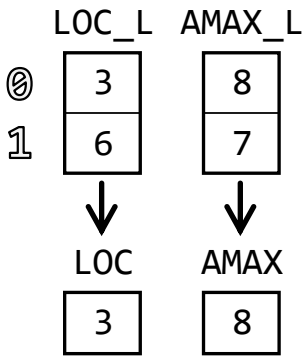
■ False Sharing（2）：並列化したプログラムの動作 [4/4]

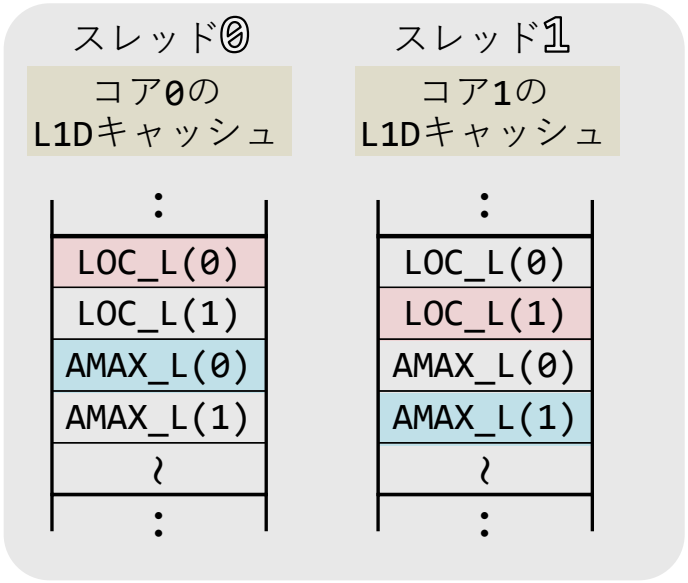
ループ例（スレッド並列）：False Sharing発生［続］

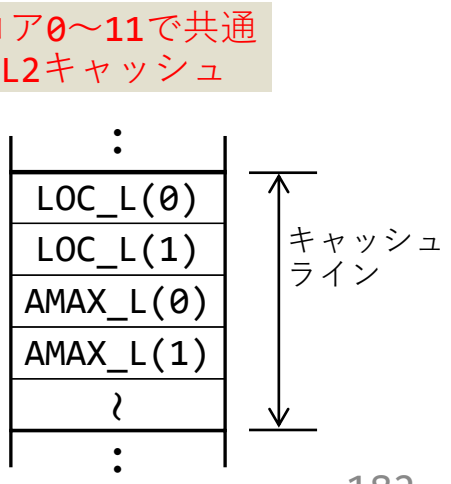
```
（省略）
!$OMP DO                                        ③
    DO I=1,N                                    ④
      IF (A(I) > AMAX_L(ID)) THEN
        LOC_L(ID) = I                           ⑤
        AMAX_L(ID) = A(I)                       ⑥
      ENDIF
    ENDDO
!$OMP END DO
!$OMP END PARALLEL                              ⑦
```

■ この例では配列 LOC_L と AMAX_L がコア0とコア1の両方のL1Dキャッシュライン上に配置される。

⇒ ■ スレッド0または1が⑤と⑥で LOC_L と AMAX_L を更新するたびに他方のスレッドのキャッシュラインが無効になり、False Sharingが発生する。







■ False Sharing（2）：回避方法1（プライベート変数）によりFalse Sharing回避 [1/2]

前スライドのFalse Sharingが発生するプログラム例に対して、プライベート変数利用によりFalse Sharingを回避するプログラム例を示す。

- 前例の LOC_L、 AMAX_L の代わりに、①で LOC_TMP と AMAX_TMP をプライベート変数 として宣言する。
- ②と③で、プライベート変数 LOC_TMP と AMAX_TMP を使用し並列計算する。

■ LOC_TMP と AMAX_TMP はコア0とコア1のレジスタにそれぞれ置かれる。これによりFalse Sharingを防ぐことができる。

■ （次スライドへ続く）

ループ例（スレッド並列）：
プライベート変数利用でFalse Sharing回避

```
!$  USE OMP_LIB
  INTEGER,PARAMETER::N=8, NT=2
  DOUBLE PRECISION::A(N)
  INTEGER::LOC,LOC_L(0:NT-1),LOC_TMP
  DOUBLE PRECISION::AMAX,AMAX_L(0:NT-1)
  DOUBLE PRECISION::AMAX_TMP
  INTEGER::ID,I
!$OMP PARALLEL DEFAULT(NONE) &
!$OMP & SHARED(A,LOC_L,AMAX_L) &
!$OMP & PRIVATE(I,ID,LOC_TMP,AMAX_TMP)      ①
    ID = OMP_GET_THREAD_NUM()
    LOC_TMP = 0
    AMAX_TMP = 0D0
!$OMP DO
    DO I=1,N ←スレッド並列化
      IF (A(I) > AMAX_TMP) THEN
        LOC_TMP = I                          ②
        AMAX_TMP = A(I)                      ③
      ENDIF
    ENDDO
!$OMP END DO
```

（次スライドへ続く）

■ False Sharing（2）：回避方法1（プライベート変数）によりFalse Sharing回避 [2/2]

- !$OMP END DOの後、④と⑤で、各スレッドが求めた LOC_TMP と AMAX_TMP を、それぞれ配列 LOC_L と AMAX_L に格納する。
- ⑥と⑦で（前例と同様に逐次処理で） LOC と AMAX を求める。

ループ例（スレッド並列）：
プライベート変数利用でFalse Sharing回避［続］

```
（!$OMP END DO：前スライドからの続き）
    LOC_L(ID) = LOC_TMP                    ④
    AMAX_L(ID) = AMAX_TMP                  ⑤
!$OMP END PARALLEL
    LOC = 0
    AMAX = 0D0
    DO ID=0,NT-1 ←逐次処理
      IF (AMAX_L(ID) > AMAX) THEN
        LOC = LOC_L(ID)                    ⑥
        AMAX = AMAX_L(ID)                  ⑦
      ENDIF
    ENDDO
```

以上で回避方法1（プライベート変数）によるFalse Sharing回避の説明を終わる。

■ False Sharing（2）：回避方法2（パディング）によりFalse Sharing回避

前述のFalse Sharingが発生するプログラム例に対して、パディングによりFalse Sharingを回避するプログラム例を示す。

- LOC_L(0)とLOC_L(1)が、異なるキャッシュライン(*1)に載るようにすれば、False Sharingを防ぐことができる（AMAX_L(0)とAMAX_L(1)に対しても同様）。

  (*1) 1キャッシュラインは256バイト：単精度だと64要素、倍精度だと32要素。

  - 配列 AMAX_L(0:1) をパディングし、配列 AMAX_L_PAD(32 ,0:1) （配列AMAX_L_PADは倍精度なので32）とする。
  - 配列 LOC_L(0:1) をパディングし、配列 LOC_L_PAD(64 ,0:1) （配列LOC_L_PADは単精度なので64）とする。
- パディングに伴い、プログラム内で配列 LOC_L_PAD と AMAX_L_PAD を使用している部分を (1,ID) のように変更する。

以上で回避方法2の説明を終わる。

ループ例（スレッド並列）：
パディングでFalse Sharing回避

```
!$  USE OMP_LIB
  INTEGER,PARAMETER::N=8, NT=2
  DOUBLE PRECISION::A(N)
  DOUBLE PRECISION::AMAX_L_PAD(32,0:NT-1)
  INTEGER::LOC_L_PAD(64,0:NT-1)
（その他の宣言省略）
!$OMP PARALLEL DEFAULT(NONE) &
!$OMP & SHARED(A,LOC_L_PAD,AMAX_L_PAD) &
!$OMP & PRIVATE(I,ID)
    ID = OMP_GET_THREAD_NUM()
    LOC_L_PAD(1,ID) = 0
    AMAX_L_PAD(1,ID) = 0D0
!$OMP DO
    DO I=1,N
      IF (A(I) > AMAX_L_PAD(1,ID)) THEN
        LOC_L_PAD(1,ID) = I
        AMAX_L_PAD(1,ID) = A(I)
      ENDIF
    ENDDO
!$OMP END DO
!$OMP END PARALLEL
（以下、逐次処理部分は省略）
```

■ False Sharing（2）：実行結果例 [1/3]

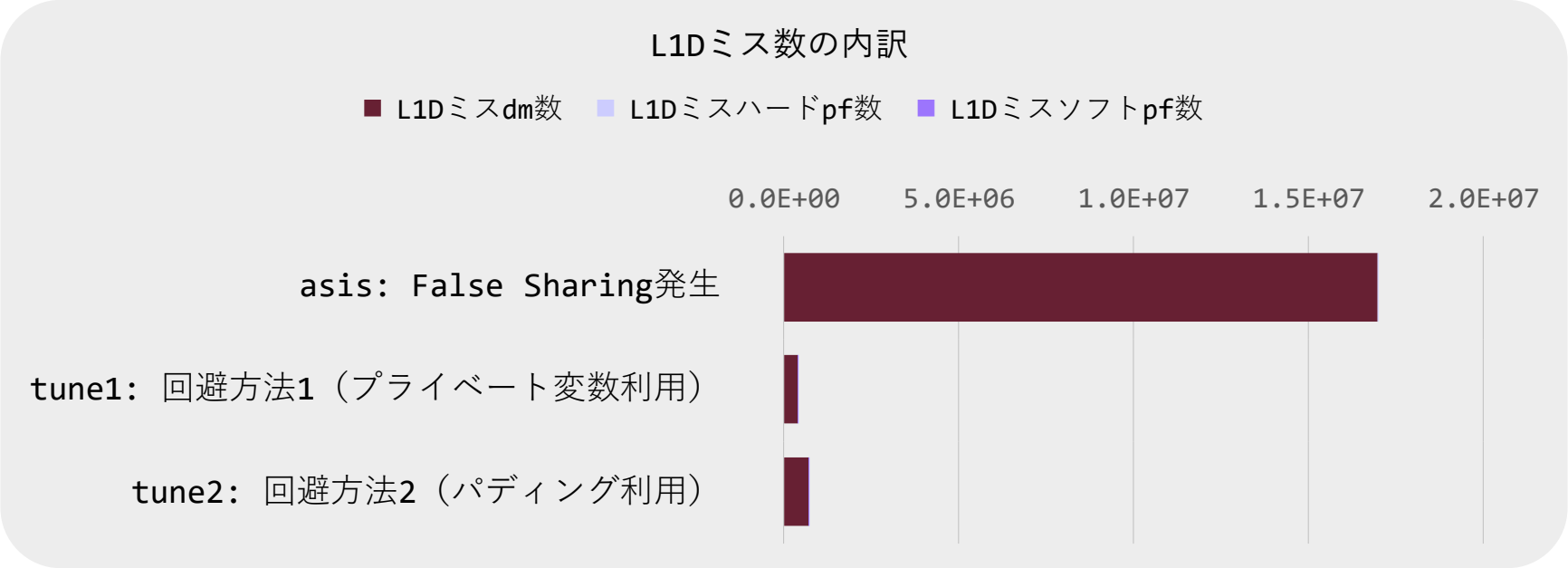
前スライドまでのFalse Sharing発生のプログラム例とその回避例では、説明のための簡素化として配列Aの要素数N=8、スレッド数NT=2とした。
ここからは、上記プログラム例を「富岳」で実際に実行・計測するため、配列Aの要素数N=5000、スレッド数NT=12とし、さらにfapp計測のための繰り返し数NLOOP=10000とする。最適化のコンパイルオプションは-Kfastとする。OpenMP指示行を有効にするため、-Kopenmpも追加する。CPU性能解析レポートを取得する。

各プログラムの名称を以下のようにする。

asis: False Sharingが発生する元のプログラム
tune1: 回避方法1（プライベート変数利用）
tune2: 回避方法2（パディング利用）

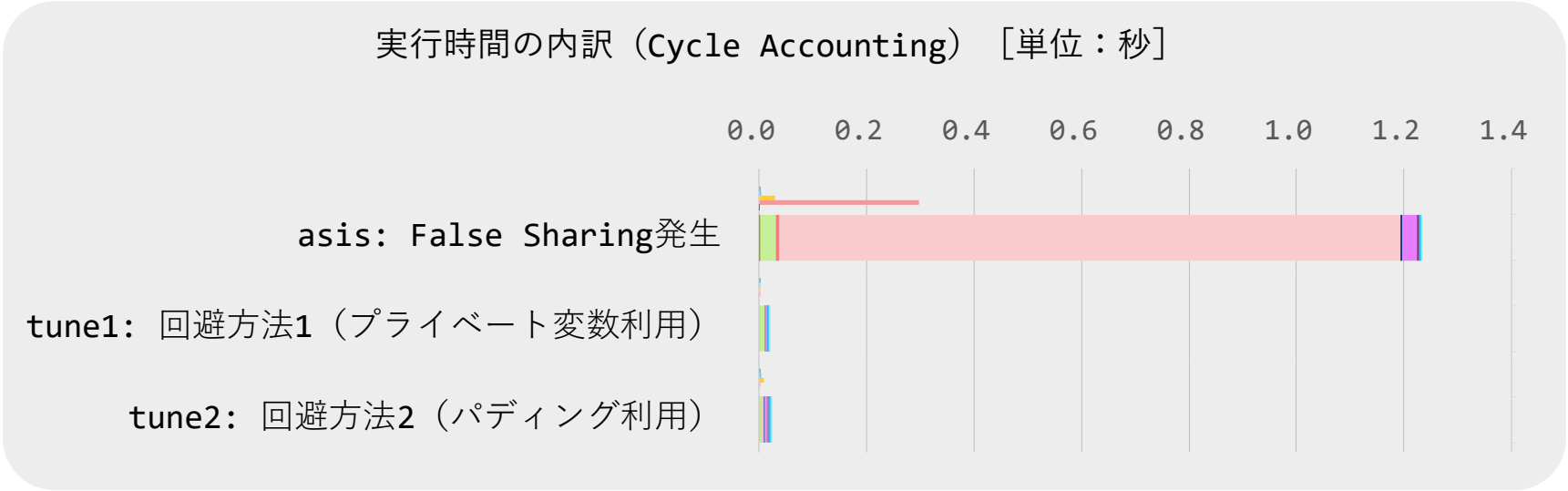
■ False Sharing（2）：実行結果例 [2/3]

CPU性能解析レポートからL1Dキャッシュミス数の内訳の比較結果を以下の図に示す。False Sharingが発生するasisではL1Dキャッシュのデマンドミス（■）が多発していることが確認される。対して、プライベート変数利用またはパディングによりFalse Sharingを回避した本例ではL1Dキャッシュミス数（■、■、および■の合計）およびデマンドミス数が減少したことが確認される。



■ False Sharing（2）：実行結果例 [3/3]

CPU性能解析レポートから実行時間の内訳（Cycle Accounting）の比較結果を以下の図に示す。プライベート変数利用またはパディングによりFalse Sharingを回避した本例ではL1Dキャッシュアクセス待ちの時間（■）が改善されたことが確認される。



- 各種最適化手法
  - キャッシュチューニングのための基本事項

本節の内容

    - データキャッシュ構成
    - キャッシュのレイテンシの改善
    - 避けるべきL1Dキャッシュミス多発の状況
    - キャッシュへのデータ転送量の軽減
    - キャッシュ利用効率の向上

- 各種最適化手法
  - キャッシュチューニングのための基本事項
    - キャッシュへのデータ転送量の軽減

本小節の内容

本小節では、キャッシュへのデータ転送量を軽減させるための最適化手法である高速ストア（ZFILL）を説明します。

      - 高速ストア（ZFILL）

■ 高速ストア（ZFILL）：概要

ZFILLとは、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領域を確保する命令（DC ZVA命令）を使い、データを高速にストアする最適化である。

■ ZFILLの適用条件

■ ZFILLはループ内でストアされる配列データに対して適用される。

- 例1および例2にZFILLが適用される例を示す。例1と例2では配列Aに対してZFILLが適用される。

例1

```
DO I=1,N
  A(I) = 0D0
ENDDO
```

例2

```
DO I=1,N
  A(I) = B(I) + 1D0
ENDDO
```

■ ZFILLが適用されない場合

■ 配列が以下のいずれかである場合にはZFILLは適用されない。

- 同一ループ内にストア対象の配列に参照がある（例3）。
- 配列が非連続アクセスされる（例4）。
- 配列がIF構文配下でストアされる（例5）。

例3

```
DO I=1,N
  A(I) = A(I) + 1D0
ENDDO
```

例4

```
DO I=1,N
  A( IFLAG(I) ) = B(I) + 1D0
ENDDO
```

例5

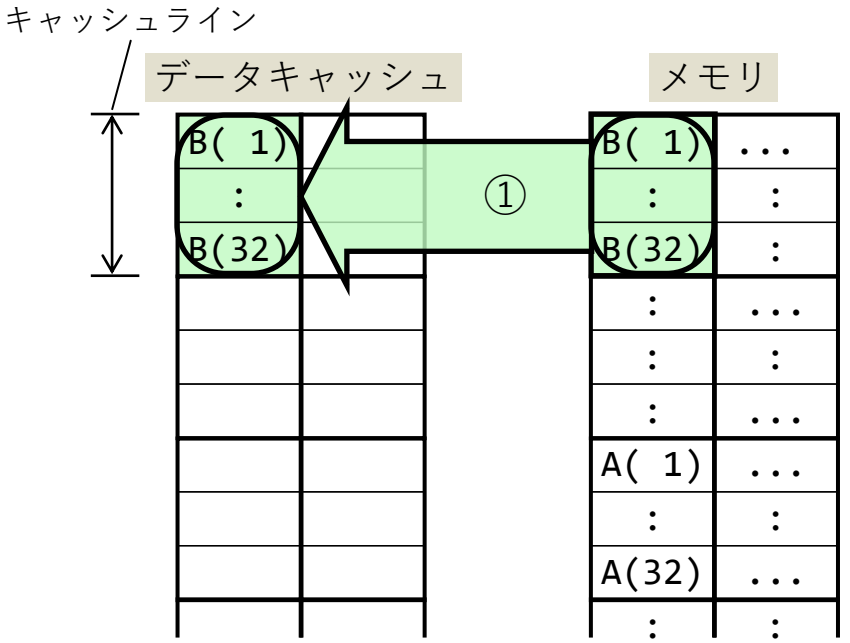
```
DO I=1,N
  IF ( M(I) ) THEN
    A(I) = B(I) + 1D0
  ENDIF
ENDDO
```

■ ZFILLを適用しない場合の動作概要 [1/4]

本小節では右の図に示すループを例にZFILLの動作の概要を説明する。まず、ZFILLを適用しない場合の動作概要を説明する。

```
DOUBLE PRECISION::A(N),B(N)
DO I=1,N
  A(I) = B(I) + 1D0
ENDDO
```

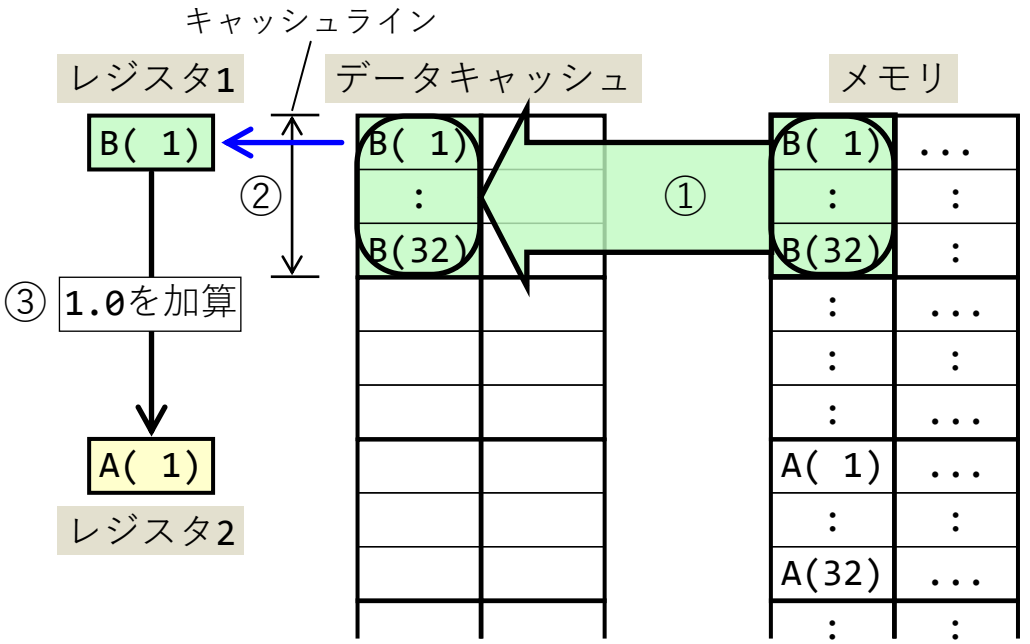
- ループの実行前には配列AとBのデータはデータキャッシュ上に存在していないとする。
- まずループの1反復目で配列要素B(1)が必要になり、データキャッシュ上に（最初なので）B(1)がないためキャッシュミスが発生する。
  - 図中①の緑色の矢印に示すように、Bが倍精度の場合の1キャッシュライン分に相当するB(1)～B(32)がメモリからデータキャッシュに転送される。
  - この、メモリからデータキャッシュへの転送には時間がかかる。
- （次スライドへ続く）



■ ZFILLを適用しない場合の動作概要 [2/4]

- 図中②の青い矢印で示すように、データキャッシュからレジスタ1にB(1)が転送される。
- レジスタ1内のB(1)に対し、③で値1.0を加算し、結果がレジスタ2に入る。
- （次スライドへ続く）

```
DO I=1,N
  A(I) = B(I) + 1D0
ENDDO
```



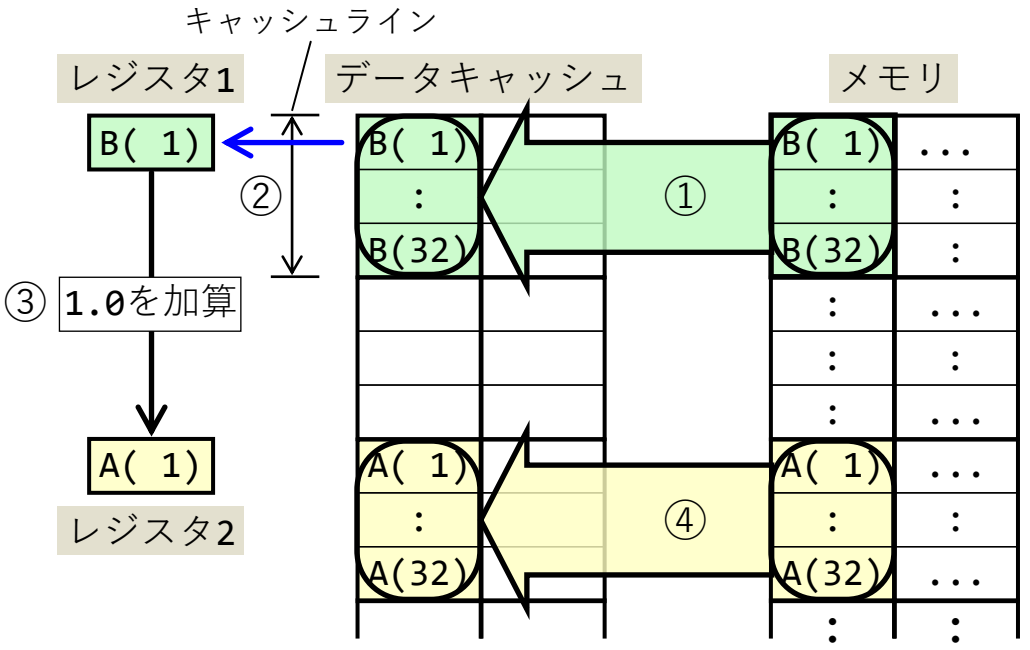
■ ZFILLを適用しない場合の動作概要 [3/4]

■ 次に配列要素A(1)へのストアが必要になる。

- 図中④の黄色い矢印で示すように、A(1)を含む1キャッシュライン分A(1)～A(32)がメモリからデータキャッシュに転送される。
- この、メモリからデータキャッシュへの転送には時間がかかる。

■ （次スライドへ続く）

```
DO I=1,N
  A(I) = B(I) + 1D0
ENDDO
```



■ ZFILLを適用しない場合の動作概要 [4/4]

■ 図中⑤の赤い矢印で示すように、A(1)をレジスタ2からデータキャッシュに書き出す。

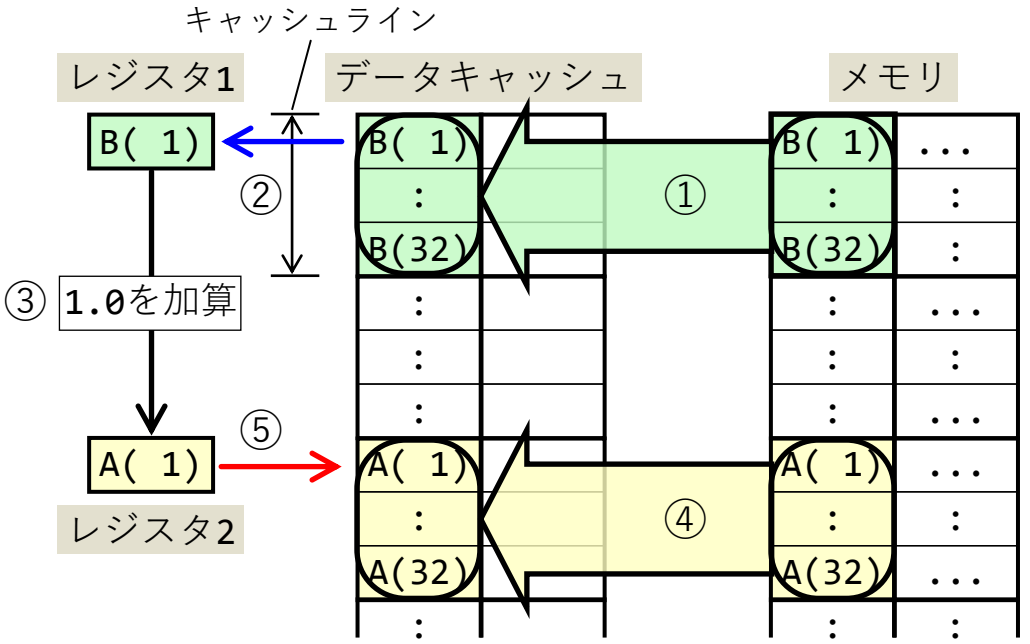
以上でループの1反復目が終了する。

■ ループの2反復目から32反復目は、データキャッシュ上にあるB(2)とA(2)、B(3)とA(3)、...、B(32)とA(32)を、②③⑤の順に処理する。

■ ループの33～64反復目を上記同様に処理する。（以後の動作の説明は省略する。）

以上がZFILLを適用しない場合の
動作である。

```
DO I=1,N
  A(I) = B(I) + 1D0
ENDDO
```

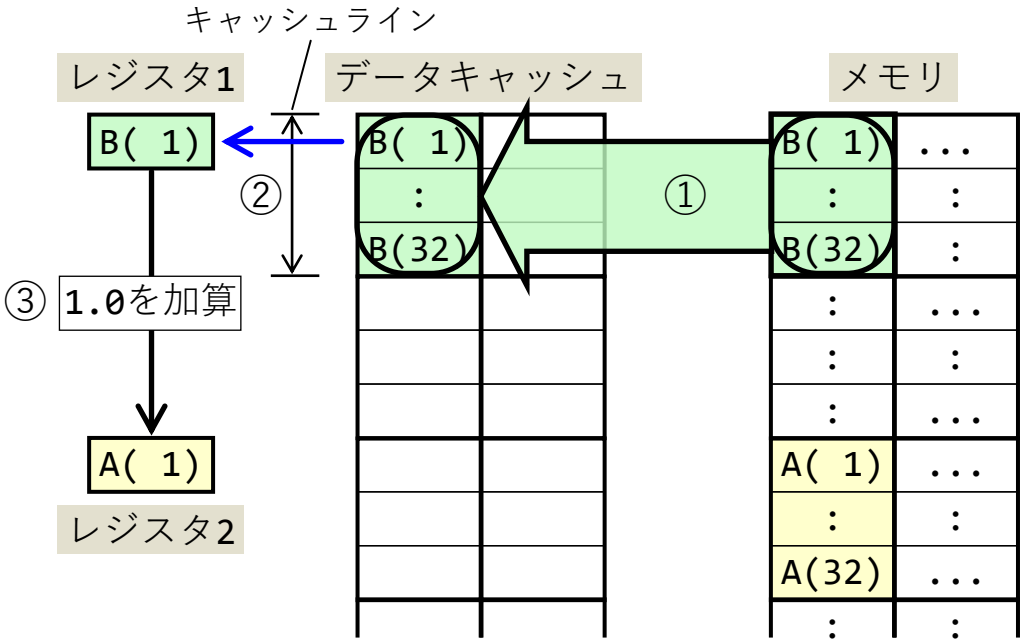


■ ZFILLを適用する場合の動作概要 [1/2]

次に、ZFILLを適用する場合の動作の概要を説明する。

- まずループの1反復目でB(1)が必要になる。ZFILLを適用する場合、①～③までの動作はZFILLを適用しない場合と同じである。
  - ①でメモリからB(1)～B(32)をデータキャッシュに転送する。
  - ②でデータキャッシュからレジスタ1にB(1)を転送する。
  - ③で値1.0を加算し、結果がレジスタ2に入る。
- （次スライドへ続く）

```
DO I=1,N
  A(I) = B(I) + 1D0
ENDDO
```



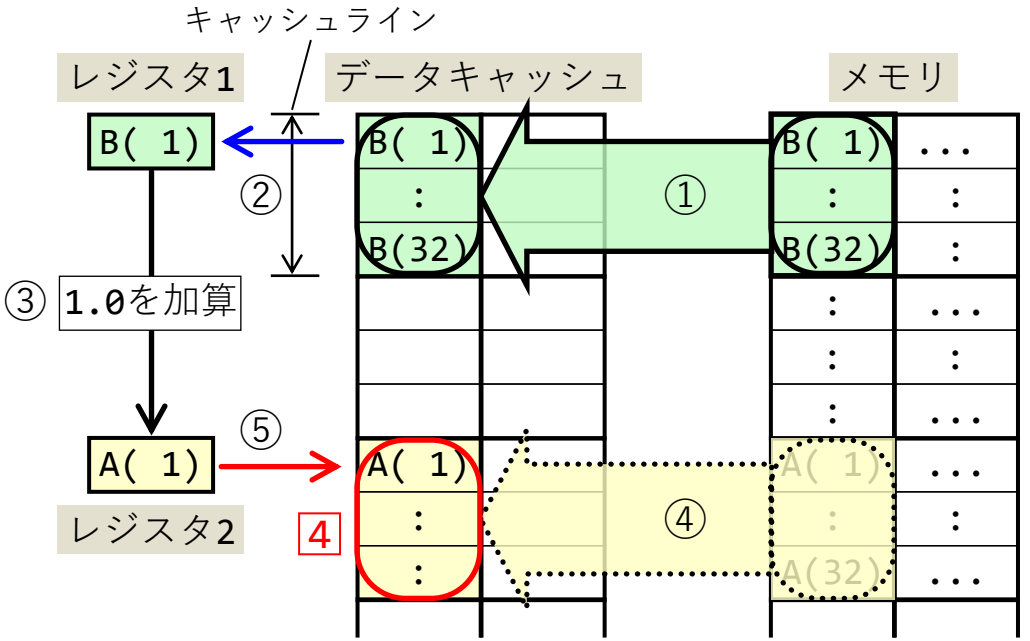
■ ZFILLを適用する場合の動作概要 [2/2]

■ ZFILLを適用する場合、時間のかかる④の転送は行われない。その代わり 4 の赤い線の囲みで示すように、データキャッシュ上にA(1)～A(32)の書き込み用のスペースが確保される。スペースの中身は不定値である。

- 書き込み用のスペースに、⑤の赤い矢印で示すように、A(1)をレジスタ2からキャッシュに書き出す。（以後の動作の説明は省略する。）
- 時間のかかるメモリからデータキャッシュへの④の転送が行われないことにより、（ループによっては）速度が速くなる(*1)。

(*1) ただし、ループアンローリング、ループストライピングが行われなくなるので、速度が低下することがある。反復回数の少ないループは速度が低下することがある。

```
DO I=1,N
  A(I) = B(I) + 1D0
ENDDO
```



■ 高速ストア（ZFILL）のコンパイルオプションおよび指示行

以下に示すコンパイルオプションまたは指示行により、ZFILL最適化の有効化または無効化を指示することができる。

- コンパイルオプション
  - 有効化：-Kzfill[=*N*]　　1 ≦ *N* ≦ 100
  - 無効化：-Knozfill（デフォルト）
- 指示行（DOループ単位、および配列代入文単位のみで指定可能）
  - 有効化：!OCL ZFILL[(*N*)]
  - 無効化：!OCL NOZFILL
- 引数*N*について
  - 引数*N*は1～100の範囲の整数値で指定できる。*N*を指定することで*N*キャッシュライン分先のデータを最適化の対象とする（1キャッシュラインは256バイト）。
  - *N*の指定を省略した場合、コンパイラが自動的に値を決定する。
- 注意事項
  - -Kzfillオプションは、-KA64FXおよび-O2オプション以上と同時に指定した場合に有効となる。なお、-KA64FXはデフォルトで有効である。
  - -Kzfillオプションは、-Kfastからは誘導されない

■ ZFILL最適化を適用する際の注意事項

ZFILLの適用を検討する際は、以下の点に注意して下さい。

- ZFILLが適用された場合、以下の最適化が制約をうける。
  - ZFILL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ない。
  - 本最適化は確保した領域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る。
    - 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、却って実行性能が低下する可能性がある。
  - ループがメモリバンド幅のボトルネックの影響を受けていない場合
  - ループの繰返し数が小さい場合
  - -Kzfill=*N*オプションで引数*N*を指定している、かつループによって書き込まれるメモリ領域のサイズが*N*キャッシュライン分よりも小さい場合
- ZFILLの最適化を-Kzfillオプション指定でプログラム全体に指定するとプログラム内の全てのループが本オプションの対象となり上記の制約や実行性能低下の可能性がある。安全のため、-Kzfillオプションで指定するのでなく、ZFILLの効果が見込めるループのみに対し指示行!OCL ZFILLで指定することを推奨します。

■ ZFILLの効果をみるループ例

本小節内の以降では、右下図のループについて、ZFILLを適用しない場合と適用した場合の実行結果を比較する例を示す。

- 逐次で実行した場合と自動スレッド並列で実行した場合のそれぞれについて比較する。
- 最適化のコンパイルオプションは逐次の場合 -Kfastとし、自動スレッド並列の場合 -Kfast,parallelとする。

ループ例

```
INTEGER,PARAMETER::N=12000000
DOUBLE PRECISION::A(N),B(N)
  ...
DO I=1,N
  A(I) = B(I) + 1D0
ENDDO
```

- -KfastからZFILLの無効化オプション-Knozfillが誘導される。

- 図のループに対して指示行!OCL ZFILLの指定なしの場合ZFILLを行わない。!OCL ZFILLの指定ありの場合ZFILLを行う（指示行を有効にするため-Koclも指定する）。

- 計測のためのループの繰り返し数NLOOPを1000とする。

以降のスライドで、まずZFILLを適用した場合のコンパイルリストを示し、次に実行時間の比較結果を示す。

■ ループ例のコンパイルリスト

ZFILLが適用された場合のコンパイルリスト例（自動スレッド並列）を示す。紙面の都合上、出力を加工している。逐次の場合のコンパイルリスト例は省略する。

■ 左下図のループに指示行「!OCL ZFILL」を指定し、ZFILL最適化が適用された場合、コンパイルリストに「ZFILL: A （Aは本例でZFILLが適用された配列名）」のメッセージが表示される。「jwdxxxxz-y」の形式のZFILL関連のメッセージは存在しない。

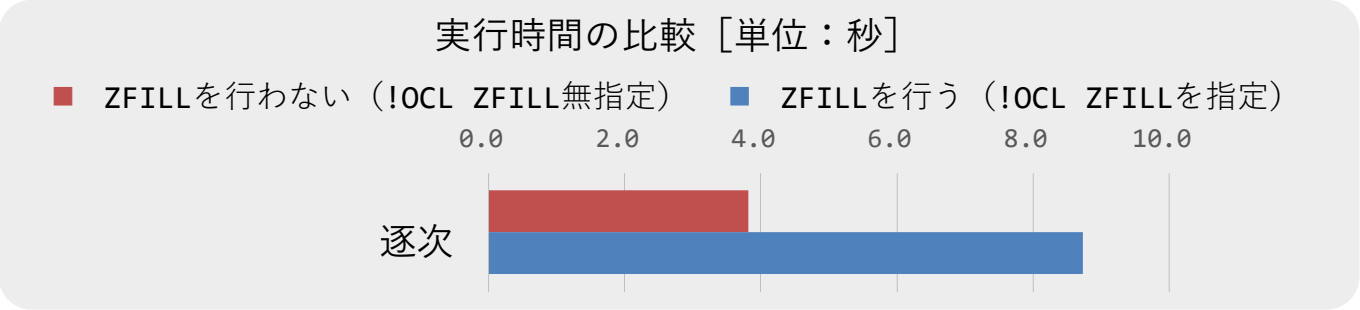
```
232   1            !OCL ZFILL
233   2  pp   v      DO I=1,N
234   2   p   v        A(I) = B(I) + 1D0
235   2   p   v      ENDDO
```

Iループの<<< Loop-information >>>

```
<<<  [PARALLELIZATION]
<<<    Standard iteration count: 1334
<<<  [OPTIMIZATION]
<<<    SIMD(VL: 8)
<<<    SOFTWARE PIPELINING(（省略）)
<<<    PREFETCH(HARD) Expected by
compiler :
<<<      B
<<<    PREFETCH(SOFT) : 2
<<<     SEQUENTIAL : 2
<<<      A: 2
<<<    ZFILL          :
<<<      A
```

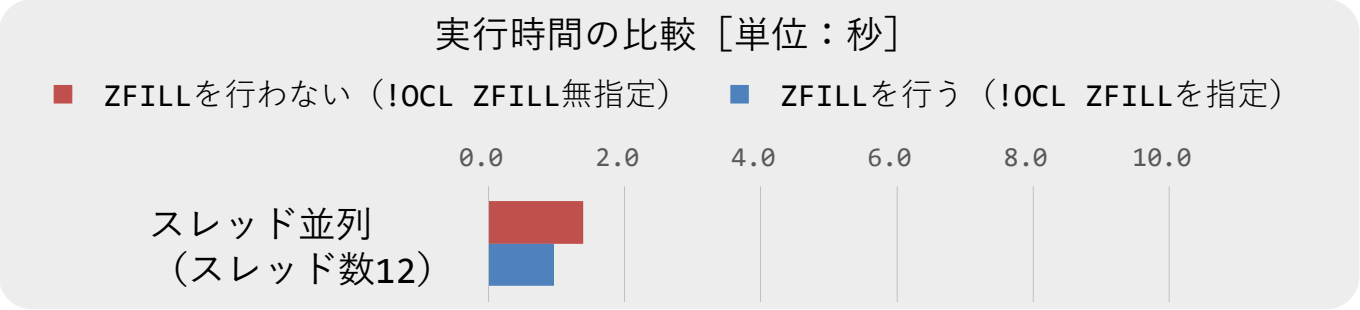
■ ループ例の実行結果：逐次とスレッド並列の比較

■ 逐次実行の場合について、ZFILLを適用しない場合と適用した場合の実行時間の比較結果を以下の図に示す。逐次で実行したとき、「ZFILLを行う（■）」は「ZFILLを行わない（■）」場合と比較して遅くなった。



■ 自動スレッド並列実行（スレッド数12）の場合について、ZFILLを適用しない場合と適用した場合の実行時間の比較結果を以下の図に示す。自動スレッド並列で実行したとき、「ZFILLを行う（■）」は、「ZFILLを行わない（■）」場合と比較して速くなった。

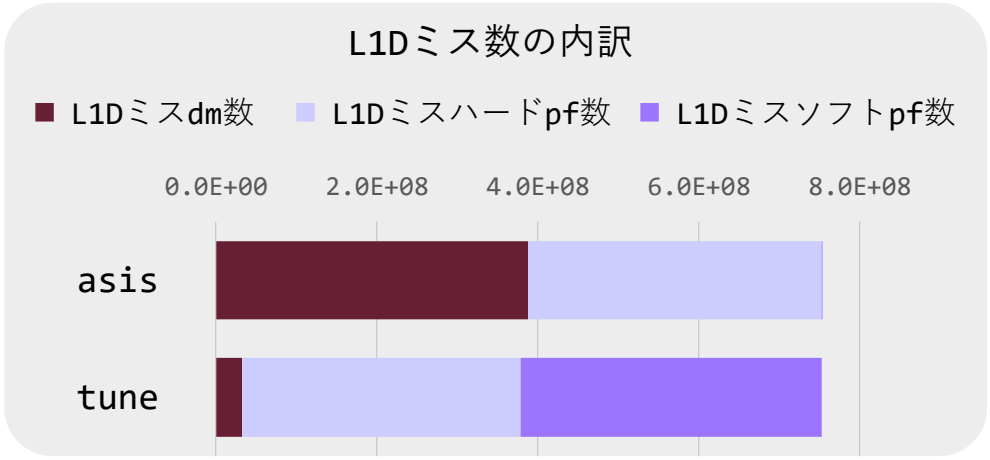
■ このようにZFILLは、一般にスレッド並列化されているループに効果がある。

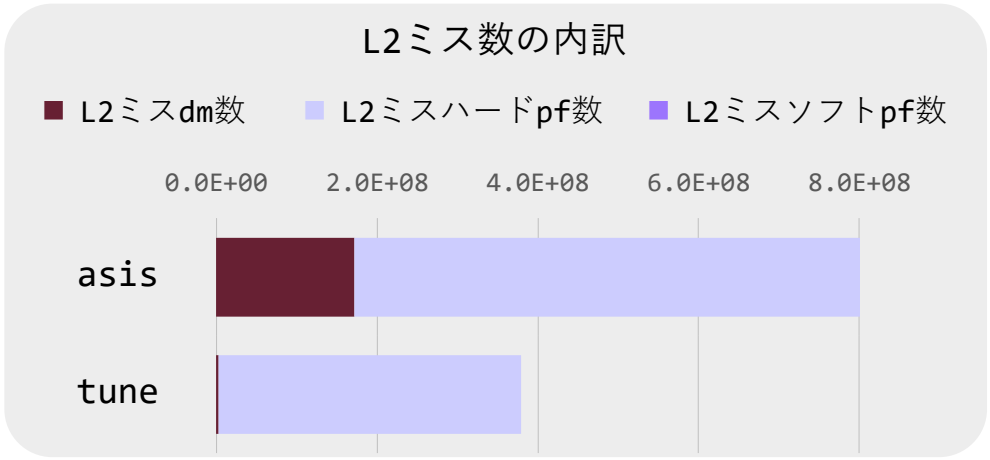


■ ループ例の実行結果：スレッド並列の場合のCPU性能解析レポート [1/2]

ここでは、本ループ例を自動スレッド並列で実行する場合に絞り、ZFILLを適用しない場合と適用した場合の実行のCPU性能解析レポートを採取し、キャッシュミス数、および実行時間の内訳を比較した結果を示す。

```
asis: ZFILLを行わない
      （!OCL ZFILL無指定）
tune: ZFILLを行う
      （!OCL ZFILLを指定）
```



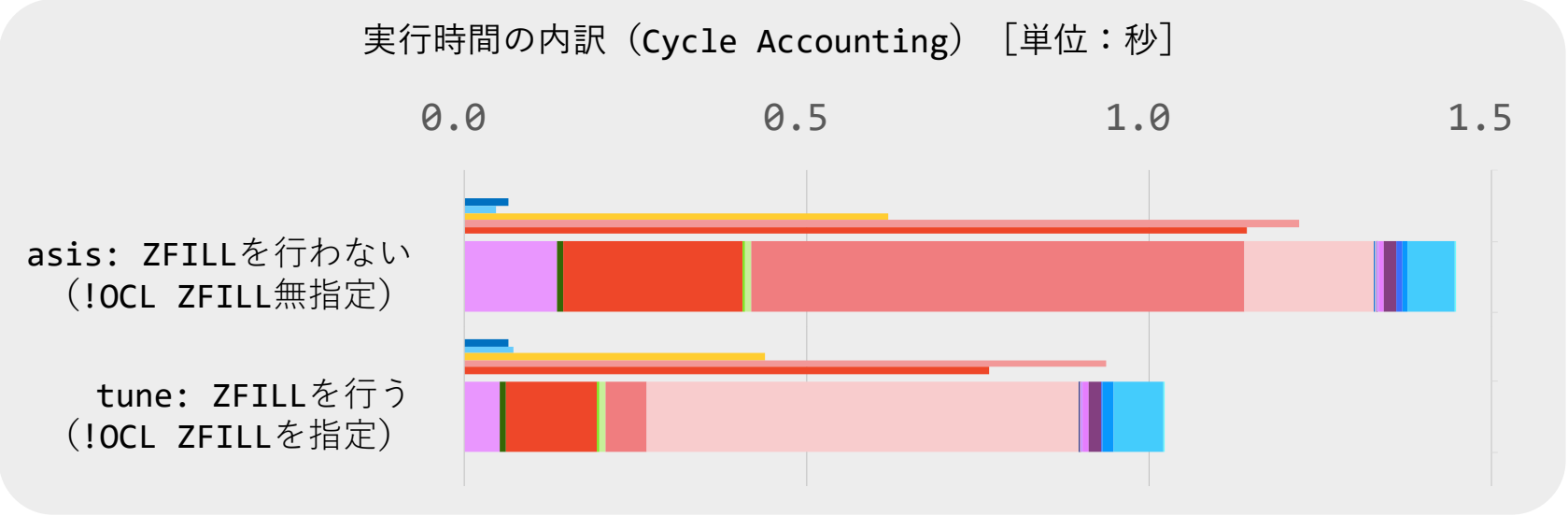


- 右上の図にL1Dキャッシュミス数の内訳の比較を示す。ZFILLのなし・ありで、ミス数自体（■、■、および■の合計）は変わらないが、ZFILLを行うことで、内訳としてデマンドミス数（■）が減少したことが確認される。
- 右下の図にL2キャッシュミス数の内訳の比較を示す。ZFILLを行うことでミス数自体が減少し、また内訳としてデマンドミス（■）が殆ど解消したことが確認される。

■ ループ例の実行結果：スレッド並列の場合のCPU性能解析レポート [2/2]

CPU性能解析レポートから自動スレッド並列での実行時間の内訳（Cycle Accounting）を以下の図に示す。

- ZFILLを行うことでL1Dキャッシュのデマンドミス数が減少したことにより、L2アクセス待ち（■）が改善されたことが確認される。
- また、ZFILLを行うことでL2ミス数が減少したことにより、メモリアクセス待ち（■）が改善されたことが確認される。



- 各種最適化手法
  - キャッシュチューニングのための基本事項

本節の内容

    - データキャッシュ構成
    - キャッシュのレイテンシの改善
    - 避けるべきL1Dキャッシュミス多発の状況
    - キャッシュへのデータ転送量の軽減
    - キャッシュ利用効率の向上

- 各種最適化手法
  - キャッシュチューニングのための基本事項
    - キャッシュ利用効率の向上

本小節の内容

本小節では、キャッシュの利用効率を向上させるための各種最適化手法を説明します。

- ブロック化（またはループブロッキング）
- アンロール・アンド・ジャム
- セクタキャッシュ

- 各種最適化手法
  - キャッシュチューニングのための基本事項
    - キャッシュ利用効率の向上

本小節の内容

      - ブロック化（またはループブロッキング）
      - アンロール・アンド・ジャム
      - セクタキャッシュ

■ 本項の内容

本項では、多重ループにおいてキャッシュミスを低減させるチューニング手法である、ブロック化（またはループブロッキングともいう）について説明する。

ブロック化なしのプログラム例

```
REAL*8 A(8,8),B(8,8)
DO J=1,8
  DO I=1,8
    A(I,J) = B(J,I)
  ENDDO
ENDDO
```

- 説明を具体的にするために、右の図に示す、2次元配列を処理する2次元ループを例にとる。
- 以降のスライドでは、まず準備として、2次元配列の図の説明、データキャッシュの簡易モデル、およびキャッシュライン番号と配列データとの対応関係を説明する。
- ブロック化しない場合についてループ反復の進行に応じて、簡易モデルで、データキャッシュの状態がどのようであるか、キャッシュミスがどのように発生するか、を説明する。
- 次に上記2次元ループをブロック化したプログラム例を示す。ブロック化したプログラム例について、ループ反復の進行に応じてデータキャッシュの状態がどのようになるかを説明する。ブロック化したプログラム例では簡易モデルでキャッシュミスが低減することを説明する。
- 最後にループブロッキングのコンパイルオプションおよび指示行を記す。

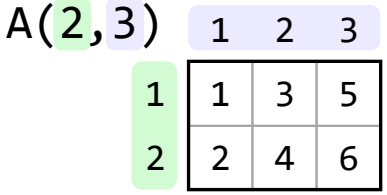
なお、ページ数の制約上、ブロック化の効果を見るための実際の実行結果例は省略します。

■ 2次元配列の図についての説明

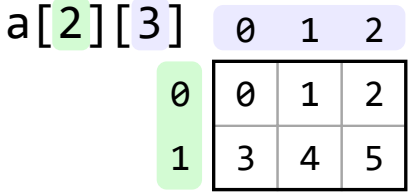
■ 本スライドでは以下に示す2次元配列の図は、左側の次元（本例では2）を下方向、右側の次元（本例では3）を右方向にする。

配列の図

Fortran言語

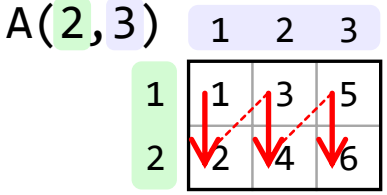


【参考】C言語

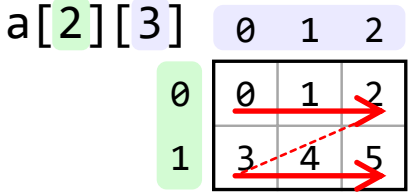


■ 配列要素のメモリ内での配置順は、Fortranでは左側の添字が先に動く順、C言語では右側の添字が先に動く順になる。赤い矢印（↓または→）は配列要素がメモリ内で配置される順（要素が連続である順）を表す。

Fortran言語



【参考】C言語



※ 本項内の以降では配列の図はFortranの場合の配列の図であるとします。

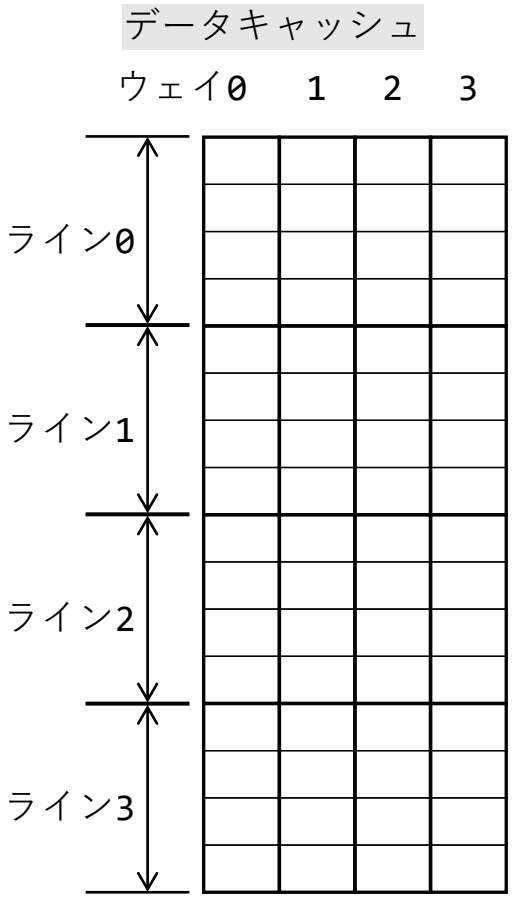
■ データキャッシュの簡易モデル

■ 「富岳」のデータキャッシュは1次キャッシュと2次キャッシュの2種類だが、以下の説明では便宜上1次キャッシュのみを考え、単にデータキャッシュと呼ぶこととする(*1)。

(*1) 実際は1次キャッシュとメモリの間に2次キャッシュが存在し、1次・2次キャッシュ間のラインの対応は「データキャッシュ構成」で述べた通りです。

■ データキャッシュについて、右の図に示すようなウェイ数およびライン数である簡易モデルとする。

- ウェイ数：4
- 1ウェイ当たりのキャッシュライン数：4
- 1ライン当たりの要素数：4



※ 本項はブロック化のなしとありでのアクセス順序の違いによるデータキャッシュの状態を説明することを目的とします。この目的のため、本項のループ例において、配列AとBに対するプリフェッチおよび配列Aに対するZFILLは無効であるとします。

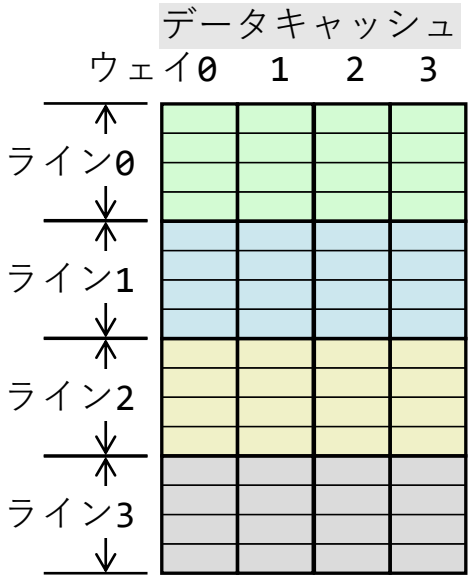
■ キャッシュライン番号と配列データとの対応関係

以下の説明では紙面の関係で配列の要素A(1,2)をA_{12}のように表す。

- 配列要素のデータはライン単位でメモリからキャッシュに転送される。（本簡易モデルでは1ラインは4要素。）
- 配列Aや配列Bの配列の図において、配列要素がどのラインに対応するかを以下の右側二つの図のように色付けして示すこととする。
  - 例えばA_{11}、A_{21}、A_{31}、A_{41}のメモリ上で連続した四つのデータはデータキャッシュのライン0に対応する。B_{11}、B_{21}、B_{31}、B_{41}についても同様である。
  - 配列Aの四つの連続したデータA_{1j}、A_{2j}、A_{3j}、A_{4j}は列番号jが一つ増すごとに本簡易モデルではライン0とライン2に交互に対応することに注意して下さい。

ブロック化なしのプログラム例（再掲）

```
REAL*8 A(8,8),B(8,8)
DO J=1,8
  DO I=1,8
    A(I,J) = B(J,I)
  ENDDO
ENDDO
```



配列Aの配列の図（セルの色はラインとの対応を表す）

I＼J	1	2	3	4	5	6	7	8
1	A11	A12	A13	A14	A15	A16	A17	A18
2	A21	A22	A23	A24	A25	A26	A27	A28
3	A31	A32	A33	A34	A35	A36	A37	A38
4	A41	A42	A43	A44	A45	A46	A47	A48
5	A51	A52	A53	A54	A55	A56	A57	A58
6	A61	A62	A63	A64	A65	A66	A67	A68
7	A71	A72	A73	A74	A75	A76	A77	A78
8	A81	A82	A83	A84	A85	A86	A87	A88

配列Bの配列の図（セルの色はラインとの対応を表す）

J＼I	1	2	3	4	5	6	7	8
1	B11	B12	B13	B14	B15	B16	B17	B18
2	B21	B22	B23	B24	B25	B26	B27	B28
3	B31	B32	B33	B34	B35	B36	B37	B38
4	B41	B42	B43	B44	B45	B46	B47	B48
5	B51	B52	B53	B54	B55	B56	B57	B58
6	B61	B62	B63	B64	B65	B66	B67	B68
7	B71	B72	B73	B74	B75	B76	B77	B78
8	B81	B82	B83	B84	B85	B86	B87	B88

※ 配列Bの要素はB(J,I)で参照されるため、縦方向がJ、横方向がIのインデックスであることに注意して下さい。

配列Aの要素が連続である順番

配列Bの要素が連続である順番

■ ブロック化なしの場合のデータキャッシュの状態 [1/5]

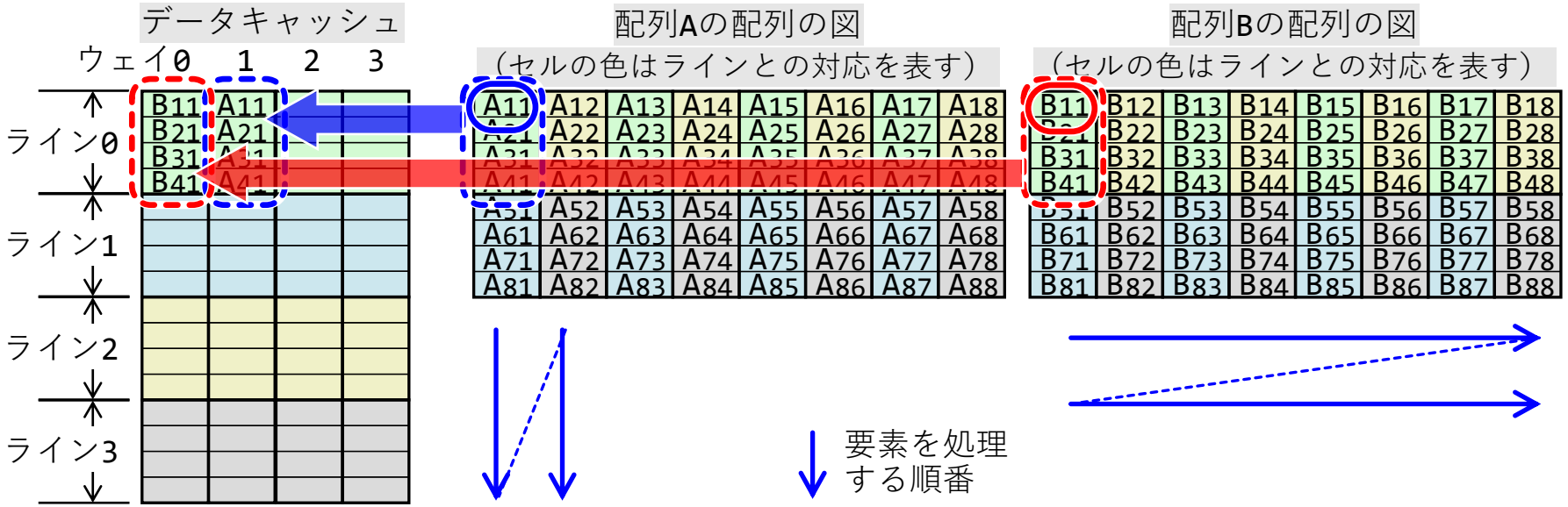
以降のスライド5ページに渡って、ブロック化なしの本ループ例について、ループ反復の進行でデータキャッシュの状態がどのようになるかを説明する。

- J=1でI=1の反復において B_{11} の参照が必要になる。B_{11} は最初なのでキャッシュにないため、キャッシュミスが発生する。B_{11} を含むラインがキャッシュのウェイ0のライン0に転送される。
- 同じ反復において A_{11} へのストアが必要になる。A_{11} は最初なのでキャッシュにないためキャッシュミスが発生する。A_{11} を含むラインがキャッシュのウェイ1のライン0に転送される(*1)。
- （次スライドへ続く）

(*1) ストアの場合、まずメモリからキャッシュに転送され、レジスタからキャッシュに書き出された後、メモリにストアされる。

ブロック化なしのプログラム例（再掲）

```
REAL*8 A(8,8),B(8,8)
DO J=1,8
  DO I=1,8
    A(I,J) = B(J,I)
  ENDDO
ENDDO
```

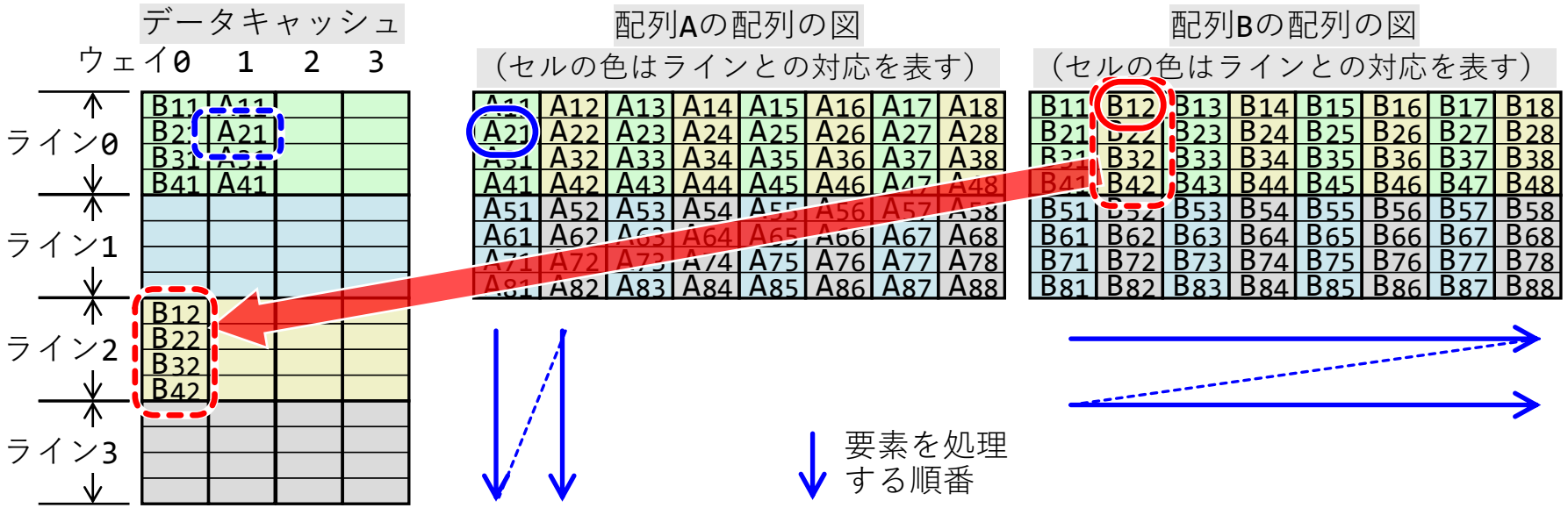


■ ブロック化なしの場合のデータキャッシュの状態 [2/5]

- J=1でI=2の反復において B_{12} の参照が必要になる。B_{12}は最初なのでキャッシュにないため、キャッシュミスが発生する。B_{12}を含むラインがキャッシュのウェイ0のライン2に転送される。
- 同じ反復において A_{21} のヘストアが必要になる。A_{21}はキャッシュのウェイ1のライン0に存在するのでキャッシュミスは発生しない。
- （次スライドへ続く）

ブロック化なしの
プログラム例（再掲）

```
REAL*8 A(8,8),B(8,8)
DO J=1,8
  DO I=1,8
    A(I,J) = B(J,I)
  ENDDO
ENDDO
```

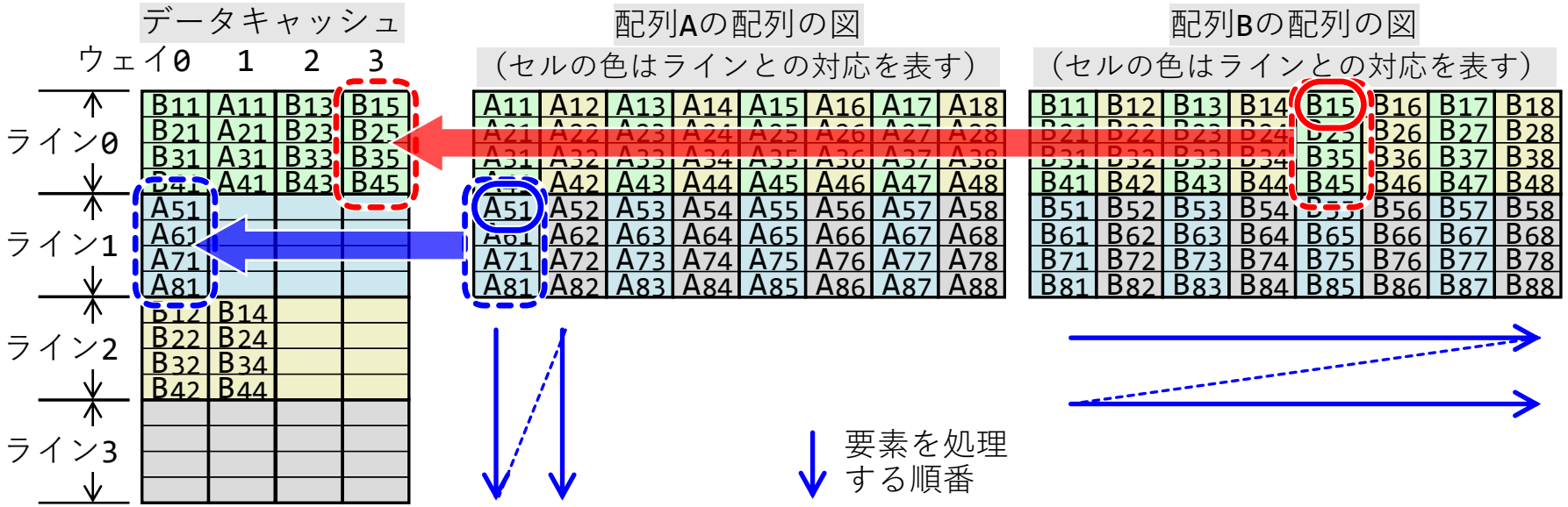


■ ブロック化なしの場合のデータキャッシュの状態 [3/5]

ブロック化なしの
プログラム例（再掲）

```
REAL*8 A(8,8),B(8,8)
DO J=1,8
  DO I=1,8
    A(I,J) = B(J,I)
  ENDDO
ENDDO
```

- J=1でI=3およびI=4の反復も同様に処理される。
- J=1でI=5の反復において B_{15} の参照が必要になる。B_{15}は最初なのでキャッシュにないため、キャッシュミスが発生する。B_{15}を含むラインがキャッシュのウェイ3のライン0に転送される。
- 同じ反復において A_{51} へのストアが必要になる。A_{51}は最初なのでキャッシュにないためキャッシュミスが発生する。A_{51}を含むラインがキャッシュのウェイ0のライン1に転送される。
- （次スライドへ続く）

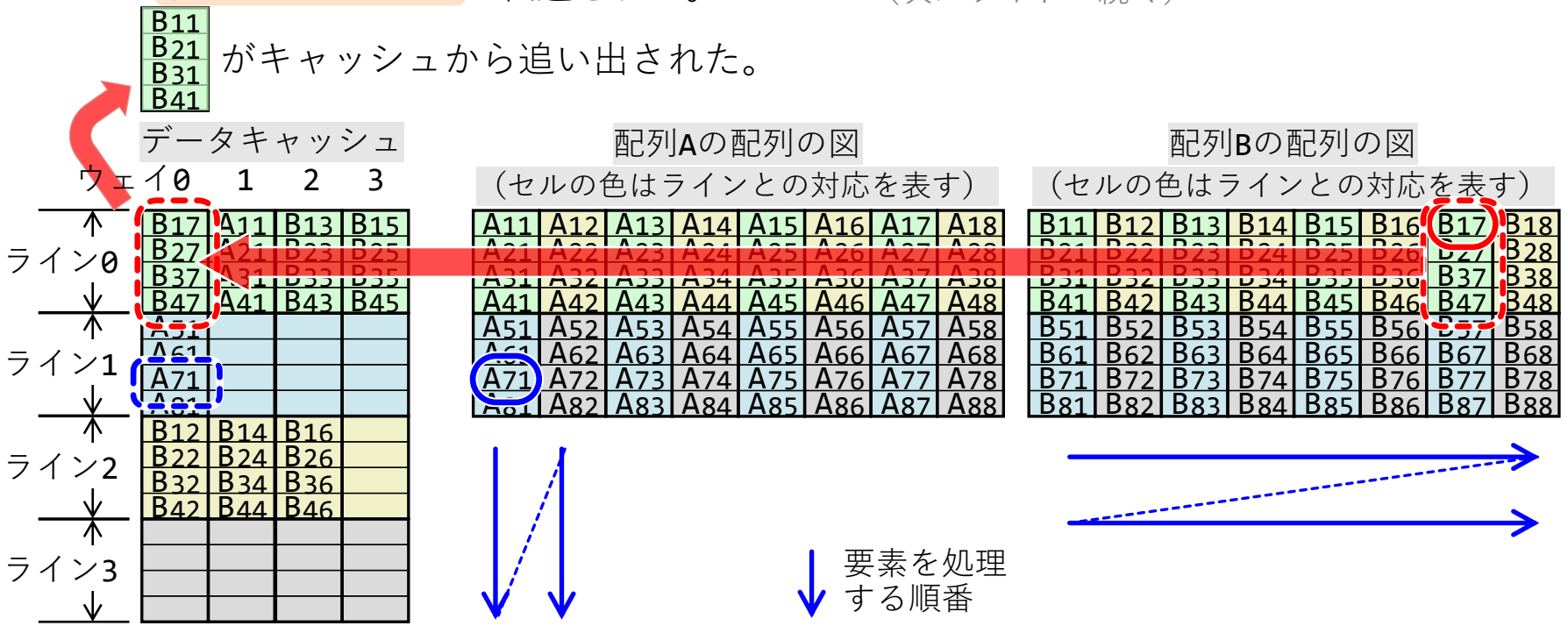


■ ブロック化なしの場合のデータキャッシュの状態 [4/5]

- J=1でI=6の反復も同様に処理される。
- J=1でI=7の反復において B_{17} の参照が必要になる。B_{17}は最初なのでキャッシュにないため、キャッシュミスが発生する。
  - B_{17}を含むラインはライン0であるが、キャッシュのウェイ0から3の全てのライン0は埋まっている。
  - そのため、各ウェイのライン0の中で一番過去にアクセスされたライン0のデータ（B_{11}、B_{21}、B_{31}、B_{41}）がキャッシュから追い出される。B_{17}を含むラインはウェイ0のライン0に転送される。 ■ （次スライドへ続く）

ブロック化なしの
プログラム例（再掲）

```
REAL*8 A(8,8),B(8,8)
DO J=1,8
  DO I=1,8
    A(I,J) = B(J,I)
  ENDDO
ENDDO
```

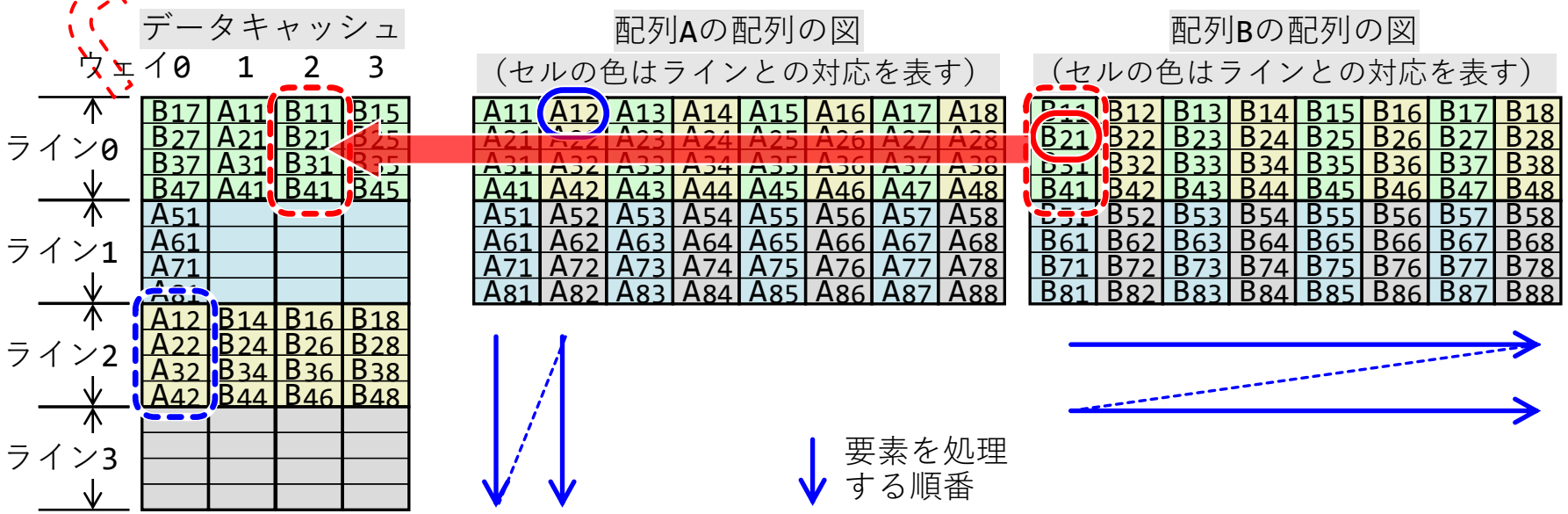


■ ブロック化なしの場合のデータキャッシュの状態 [5/5]

- J=1でI=8の反復も同様に処理される。
- Iのループが1回転しJ=2でI=1の反復となる。この反復においてB_{21}の参照が必要になる。
  - B_{21}は以前の反復J=1でI=6まではウェイ0のライン0に存在したが、J=1でI=7の反復でキャッシュから追い出された。そのため、B_{21}の参照でキャッシュミスが発生する。
    B_{21}を含むラインがキャッシュの一番過去にアクセスされたウェイ2のライン0に転送される。
  - このB_{21}のように、ブロック化しない場合キャッシュ上のデータが、その後の反復で、そのデータが必要になる前にキャッシュから追い出されているため、キャッシュミスの可能性が高くなる。

（以上でブロック化なしの場合の説明を終わる）

B11 B21 B31 B41 は、以前の反復（J=1でI=7）でキャッシュから追い出された。



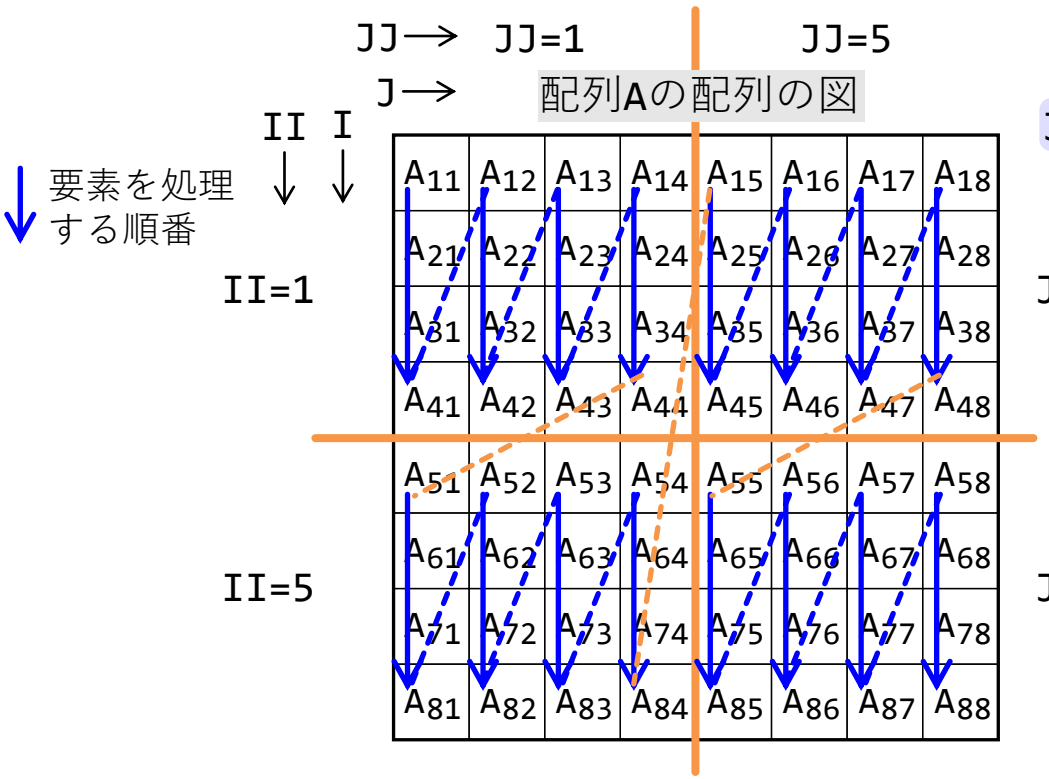
■ ブロック化したプログラム例

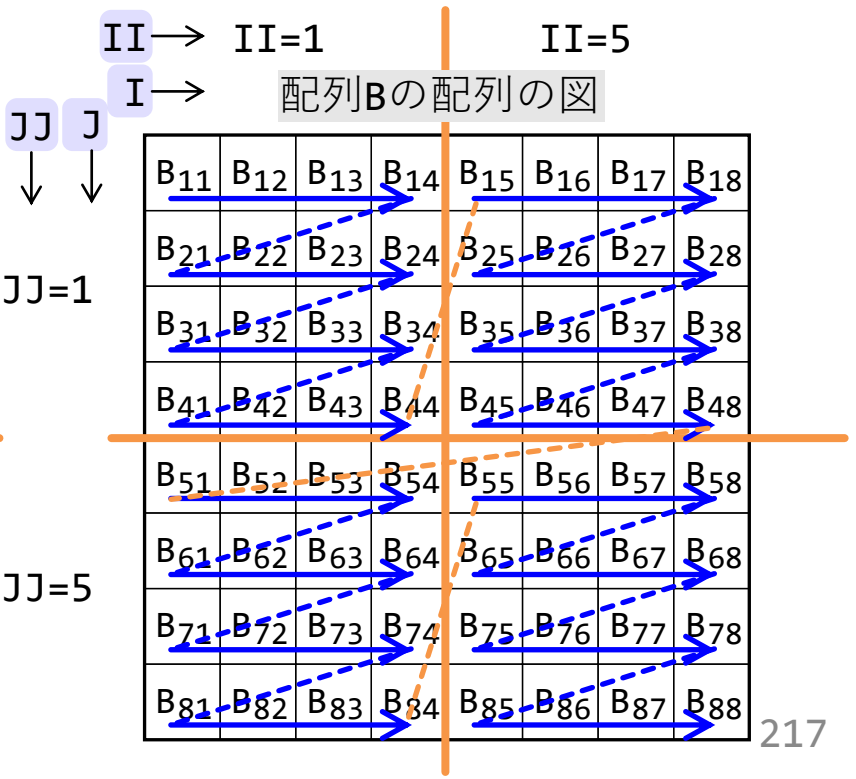
配列Bのキャッシュミスを低減させるため、ブロック化という方法でチューニングしたプログラム例を右上の図に示す。

- 本例ではI方向とJ方向のブロックのサイズを4とする。
- 下の二つの図に示すように行列AとBを小行列に分割する。
- 青い矢印・青またはオレンジ色の点線で示すように、一つの小行列内の要素を全部処理したら、次の小行列に移動するという順序で処理を行う。

ブロック化したプログラム例

```
REAL*8 A(8,8),B(8,8)
DO JJ=1,8,4
DO II=1,8,4
  DO J=JJ,MIN(JJ+3,8)
  DO I=II,MIN(II+3,8)
    A(I,J) = B(J,I)
  ENDDO
  ENDDO
ENDDO
ENDDO
```





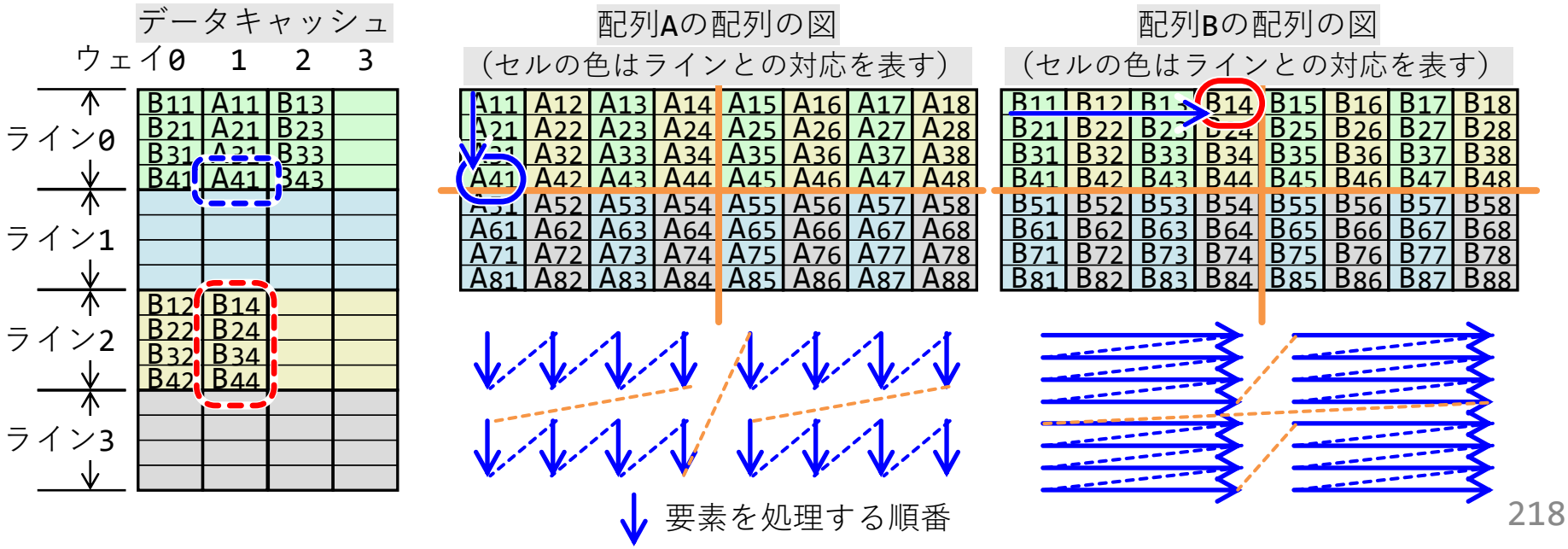
■ ブロック化したの場合のデータキャッシュの状態 [1/2]

以降のスライド2ページに渡って、ブロック化した本ループ例について、ループ反復の進行に応じてデータキャッシュの状態がどのようになるかを説明する。

- IループとJループをブロックサイズ4でブロック化した場合、最初の4反復（J=1でI=1、2、3、4の反復）までのデータアクセス順序はブロック化しない場合と同じである。従って最初の4反復直後のデータキャッシュの状態はブロック化しない場合と同じである。
- （次スライドへ続く）

ブロック化したプログラム例（再掲）

```
REAL*8 A(8,8),B(8,8)
DO JJ=1,8,4
DO II=1,8,4
   DO J=JJ,MIN(JJ+3,8)
   DO I=II,MIN(II+3,8)
     A(I,J) = B(J,I)
   ENDDO
   ENDDO
ENDDO
ENDDO
```

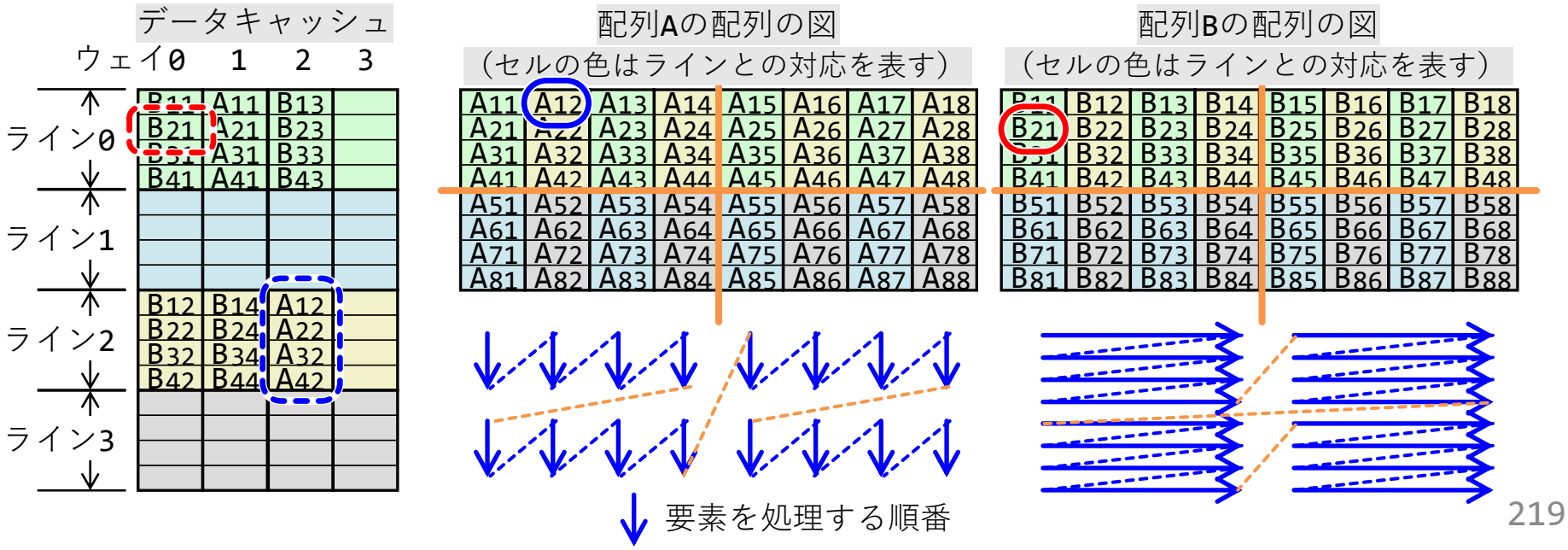


■ ブロック化したの場合のデータキャッシュの状態 [2/2]

- ブロック化した場合、5反復目はJ=2でI=1である。この反復において B_{21} の参照が必要になる。B_{21}はキャッシュのウェイ0のライン0に存在するのでキャッシュミスは発生しない。
- ブロック化する場合、このB_{21}のように、キャッシュに転送されたデータが、その後の反復でキャッシュから追い出される前に、利用できる機会が多くなる。即ちキャッシュの利用効率が向上する。
- なお、同じ反復において A_{12} へのストアが必要になり、A_{12}でキャッシュミスが発生する。配列Aに対するキャッシュミスは、本簡易モデルと本例でのブロックサイズ4では、ブロック化しない場合のキャッシュミスと同程度な頻度で発生する。（以上でブロック化した場合の説明を終わる）

ブロック化したプログラム例（再掲）

```
REAL*8 A(8,8),B(8,8)
DO JJ=1,8,4
DO II=1,8,4
  DO J=JJ,MIN(JJ+3,8)
  DO I=II,MIN(II+3,8)
    A(I,J) = B(J,I)
  ENDDO
  ENDDO
ENDDO
ENDDO
```



■ 補足事項

- ■ ブロックの最適な大きさは、ループ内の計算内容によって異なるので、試行錯誤で調整する。
- ■ ブロック化をコンパイルオプションまたは指示行で行うこともできる。次スライドで説明します。

ブロック化したプログラム例（再掲）

```
REAL*8 A(8,8),B(8,8)
DO JJ=1,8,4
DO II=1,8,4
  DO J=JJ,MIN(JJ+3,8)
  DO I=II,MIN(II+3,8)
    A(I,J) = B(J,I)
  ENDDO
  ENDDO
ENDDO
ENDDO
```

■ ループブロッキングのコンパイルオプションおよび指示行

以下に示すコンパイルオプションまたは指示行により、ループブロッキング最適化の有効化または無効化を指示することができる。

- コンパイルオプション
  - 有効化：`-Kloop_blocking[=N]`　　$2 \leqq N \leqq 10000$
  - 無効化：`-Kloop_noblocking`
- 指示行（プログラム単位、DOループ単位、および配列代入文単位のみで指定可能）
  - 有効化：`!OCL LOOP_BLOCKING(n)`　　$2 \leqq n \leqq 10000$
  - 無効化：`!OCL LOOP_NOBLOCKING`
- コンパイルオプションのデフォルト
  - `-O2`以上：`-Kloop_blocking`、`-O1`以下：`-Kloop_noblocking`
- 引数*N*または*n*について
  - 引数*N*または*n*には、ブロックの一辺の大きさ（要素数）を2～10000の範囲で指定する。
  - 有効化オプションの引数*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する。
  - 有効化指示行の引数*n*の指定は省略できない。
- 注意事項
  - ループブロッキング最適化が実際に行われるかどうかは、ループ内の処理、およびコンパイラによるその他最適化の実施状況に依存する。
  - ループブロッキング最適化が行われた場合、コンパイルリストにメッセージjwd8320が表示される。

```
jwd8320o-i  "～", line ～: ループをサイズNでブロッキングしました。
```

- 各種最適化手法
  - キャッシュチューニングのための基本事項
    - キャッシュ利用効率の向上

本小節の内容

- ブロック化（またはループブロッキング）
- アンロール・アンド・ジャム
- セクタキャッシュ

■ アンロール・アンド・ジャムとは

■ アンロール・アンド・ジャム（unroll and jam）とは、多重ループの外側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する最適化である。*N*=2の場合の本最適化のイメージを以下の図に示す。

アンロール・アンド・ジャムのイメージ

元のループ

```
DO J=1,NJ
  DO I=1,NI
    A(I,J)=B(J,I)+1D0
  ENDDO
ENDDO
```

（外側Jループを2段アンローリング）

```
DO J=1,NJ,2
  DO I=1,NI
    A(I,J)=B(J,I)+1D0
  ENDDO
  DO I=1,NI
    A(I,J+1)=B(J+1,I)+1D0
  ENDDO
ENDDO
```

（内側の二つのIループを融合）

```
DO J=1,NJ,2
  DO I=1,NI
    A(I,J  )=B(J  ,I)+1D0
    A(I,J+1)=B(J+1,I)+1D0
  ENDDO
ENDDO
```

■ アンロール・アンド・ジャムを適用することで、共通式除去による演算の効率の向上やデータアクセス順序の変更によるキャッシュ利用効率の向上の可能性がある。

- ■ 元のループ（一番左の図）の場合
  - B(J,I)のロードでキャッシュミスが多発する。
- ■ アンロール・アンド・ジャムされたループ（一番右の図）の場合
  - B(J,I)とB(J+1,I)がIループ内に纏められていることで、キャッシュミスを抑えることができる。
  - ただし配列Aのストリーム数は二つとなり、Iループ1反復当たりのストアの数が増える。
  - なお、本ループ（一番右の図）では共通式は存在しないため、共通式除去の効果はない。

■ 本項の以降の内容

- ■ アンロール・アンド・ジャムは前スライドのイメージ図のように手作業で行う方法と、コンパイルオプションまたは指示行で行う方法がある。
- ■ 本項の以降のスライドでは、まずコンパイルオプションまたは指示行を説明する。
- ■ アンロール・アンド・ジャムを適用するループ例を示し、指示行の指定によりアンロール・アンド・ジャムが適用された場合のコンパイルリスト例を示す。
- ■ ループ例に対してアンロール・アンド・ジャムを適用しない場合と適用する場合を実行し、CPU性能解析レポートを採取し、その比較例を示す。
- ■ 比較結果からアンロール・アンド・ジャムによりキャッシュの利用効率が向上することを確認する。

■ アンロール・アンド・ジャムのコンパイルオプションおよび指示行

以下に示すコンパイルオプションまたは指示行により、アンロール・アンド・ジャム最適化の有効化または無効化を指示することができる。

- ■ コンパイルオプション
  - ■ 有効化(*1)：-Kunroll_and_jam[=*N*]　2 ≦ *N* ≦ 100
  - ■ 無効化：-Knounroll_and_jam （デフォルト）
- ■ 指示行
  - ■ 有効化(*1)：!OCL UNROLL_AND_JAM[(*N*)]
    （プログラム単位、およびDOループ単位のみで指定可能）または
    !OCL UNROLL_AND_JAM_FORCE[(*N*)] (*2)（DOループ単位のみで指定可能）
  - ■ 無効化：!OCL NOUNROLL_AND_JAM（プログラム単位、およびDOループ単位のみで指定可能）
- ■ 注意事項
  - ⚠ 有効化オプションまたは有効化指示行は、データストリーム数の増加やデータアクセス順序の変化により、却ってキャッシュ利用効率を低下させる可能性がある。
  - ⚠ アンロール・アンド・ジャム最適化が効果あるか否かはループ単位で異なる。このため本最適化は、コンパイルオプションでプログラム全体へ適用せず、指示行でループ単位に適用することを推奨します。

(*1)
- 有効化オプションおよび有効化指示行は-O2オプション以上が有効な場合に意味がある。
- 有効化オプション-Kunroll_and_jamは-Kfastからは誘導されない。
- *N*にはループ展開数の上限を2～100の範囲で指定できる。*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する。
- 最内ループはアンロール・アンド・ジャムの対象にならない。

(*2) !OCL UNROLL_AND_JAMと!OCL UNROLL_AND_JAM_FORCEの違いは次スライドで説明する。

■ 指示行 !OCL UNROLL_AND_JAM と !OCL UNROLL_AND_JAM_FORCE の違い

■ 指示行 !OCL UNROLL_AND_JAM は以下の場合には最適化を行わない。

- コンパイラが最適化の効果が期待できないと判断した場合
- コンパイラが回転を跨いだデータ依存があると判断した場合（例1）

例1: 回転を跨いだデータ依存の例

```
DO J=2,NJ
  DO I=1,NI-1
    A(I,J) = A(I+1,J-1) + B(J,I)
  ENDDO
ENDDO
```

NI=3、NJ=3の場合の展開図

```
A(1,2) = A(2,1) + B(2,1)
A(2,2) = A(3,1) + B(2,2)
A(1,3) = A(2,2) + B(3,1)
A(2,3) = A(3,2) + B(3,2)
```

A(1,3)はA(2,2)の更新後に定義される。

■ 指示行 !OCL UNROLL_AND_JAM_FORCE は回転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・ジャムを適用することを指示する。

- 上の例1に対して !OCL UNROLL_AND_JAM_FORCE を誤って指定した場合、実行結果は保証されない（例2）。

例2: !OCL UNROLL_AND_JAM_FORCE の誤った指定例

```
!OCL UNROLL_AND_JAM_FORCE(2)
DO J=2,NJ
  DO I=1,NI-1
    A(I,J) = A(I+1,J-1) + B(J,I)
  ENDDO
ENDDO
```

NI=3、NJ=3の場合の展開図

```
A(1,2) = A(2,1) + B(2,1)
A(1,3) = A(2,2) + B(3,1)
A(2,2) = A(3,1) + B(2,2)
A(2,3) = A(3,2) + B(3,2)
```

A(1,3)はA(2,2)の更新前に定義されてしまう。

■ ループ例とコンパイルリスト例

本項の以降では右の図に示すループを例にとり、アンロール・アンド・ジャムの指示行!OCL UNROLL_AND_JAM_FORCE(*N*)を指定した場合の効果を説明する。最適化のコンパイルオプションは-Kfastとし、逐次実行する。指示行を有効にするため-Koclも指定する。

ループ例

```
INTEGER,PARAMETER::NI=512, NJ=625
DOUBLE PRECISION::A(NI,NJ),B(NJ,NI)
  ...
DO J=1,NJ
  DO I=1,NI
    A(I,J)=B(J,I)+1D0
  ENDDO
ENDDO
```

■ 図のループに指示行!OCL UNROLL_AND_JAM_FORCE(2)を指定した場合のコンパイルリストを以下の図に示す。外側のJループに対してメッセージjwd8230が表示され、アンロール・アンド・ジャムが適用されたことが確認される。

- 本例では「名前:J、展開数:2」により、外側のJループが2段アンローリングされたことが示されている。

```
286    1        !OCL UNROLL_AND_JAM_FORCE(2)
287    2          DO J=1,NJ
288    3    2v      DO I=1,NI
289    3    2v        A(I,J)=B(J,I)+1D0
290    3    2v      ENDDO
291    2          ENDDO
```

Iループの<<< Loop-information >>>

```
<<<  [OPTIMIZATION]
<<<    SIMD(VL: 8)
<<<    SOFTWARE PIPELINING(（省略）)
<<<     （以下、省略）
```

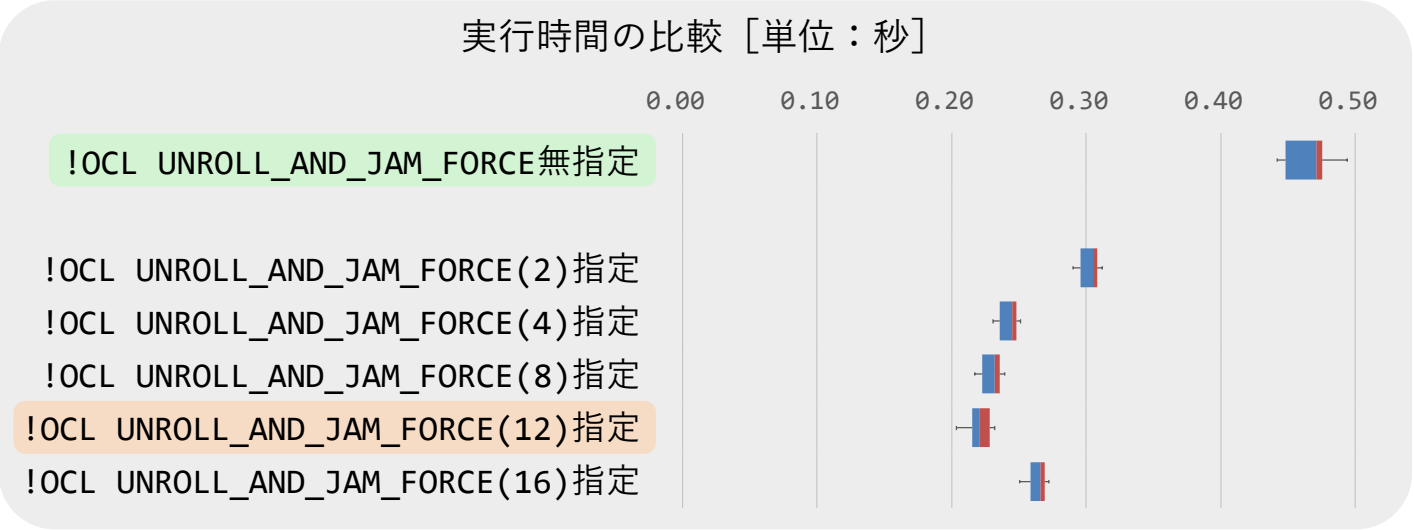
Jループのメッセージ

```
jwd8230o-i  "～", line 287: このDOループにアンロールアンドジャムを適用しました。(名前:J、展開数:2)
 （以下、省略）
```

■ ループ例の実行結果［1/2］

以降では、アンロール・アンド・ジャム最適化の指示行を指定しない場合と指定する場合のループを最適化オプション-Kfastでコンパイルし、逐次実行した結果を示す（計測のためのループの繰り返し数NLOOPを500とする）。結果の比較からアンロール・アンド・ジャムによりキャッシュの利用効率が向上したことを確認する。

- 下の図に本ループ例で指示行!OCL UNROLL_AND_JAM_FORCEを指定しない場合、および指示行!OCL UNROLL_AND_JAM_FORCE(*N*)で アンローリングの段数*N*=2、4、8、12、16をそれぞれ指定した場合の実行時間（経過時間）の比較を示す（複数回計測の箱ひげ図）。本ループ例に対しては*N*=12が最も効果あることが確認される。



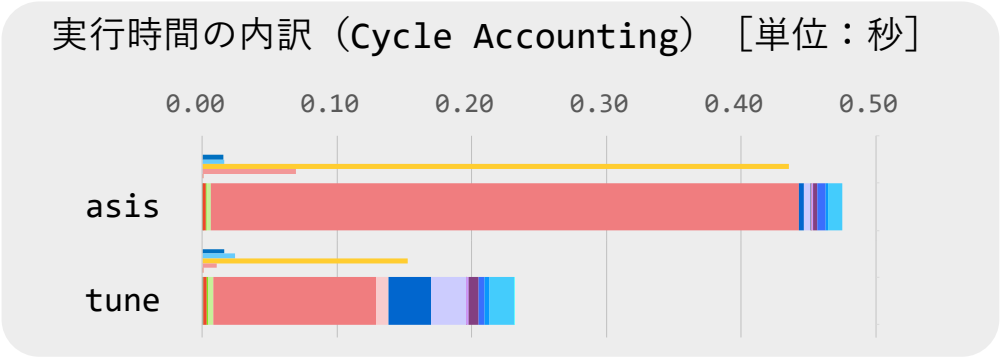
- 以降では指示行!OCL UNROLL_AND_JAM_FORCE無指定の場合および、*N*=12を指定した場合の2ケースのみを取り上げ、CPU性能解析レポートを採取し、比較結果を示すこととする。

■ ループ例の実行結果 [2/2]

asis: !OCL UNROLL_AND_JAM_FORCE無指定
tune: !OCL UNROLL_AND_JAM_FORCE(12)指定

- 右の図にL1Dキャッシュミス数の内訳の比較を示す。tuneはasisと比較しJループのアンローリングによりL1Dキャッシュミス数（■、■、および■の合計）およびデマンドミス数（■）が大幅に減少したことが確認される。tuneではL1Dキャッシュの利用効率が向上したことが確認される。

- 右の図に実行時間の内訳（Cycle Accounting）の比較を示す。asisと比較しtuneはL1Dキャッシュの利用効率が向上したことにより、L2キャッシュアクセス待ちの時間（■）が改善されたことが確認される。



- 各種最適化手法
  - キャッシュチューニングのための基本事項
    - キャッシュ利用効率の向上

本小節の内容

- ブロック化（またはループブロッキング）
- アンロール・アンド・ジャム
- セクタキャッシュ

■ セクタキャッシュとは

セクタキャッシュとは、再利用性のあるデータが再利用性の無いデータによってキャッシュから追い出されることを防ぐことができるキャッシュ機構である。
セクタキャッシュは、再利用性のあるデータと再利用性の無いデータをセクタ毎に分けて配置することができる。

■ 本項の内容

本項では、セクタキャッシュの指定方法、注意事項、2次キャッシュの場合のセクタキャッシュ指定例、およびセクタキャッシュに関する指示行を示します。

なお、ページ数の制約上、セクタキャッシュの効果を見るための実際の実行結果例は省略します。実行結果例については「プログラミングガイド チューニング編」を参照して下さい。

■ セクタキャッシュの指定方法

- 1次キャッシュと2次キャッシュのどちらもセクタキャッシュとして使用することができる。
- 通常、容量が大きい2次キャッシュをセクタキャッシュとして使用することが多い。
- 通常のウェイの集合をセクタ0、キャッシュから追い出されないようにしたい配列（再利用される配列）の入るウェイの集合をセクタ1と呼ぶ。
- セクタキャッシュの指定は指示行または環境変数で行う。コンパイルオプションはない。
- セクタキャッシュを利用する際は翻訳時オプション-Khpctag(*1)が有効でなければならない。

  (*1) オプション-Khpctagはデフォルトで有効である。オプション-Khpctagは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することを指示する。HPCタグアドレスオーバライド機能を利用することで、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を有効にできる。

■ 注意事項

⚠ 1 CMG あたり複数プロセスでプログラムを実行する場合は2次キャッシュに対してセクタキャッシュは利用できない。

以降のスライドでは指示行での基本的な指定方法を説明します。
環境変数で指定する方法は省略します。環境変数で指定する方法については「Fortran使用手引書」を参照して下さい。

■ 2次キャッシュの場合のセクタキャッシュ指定例

本スライドでは、指示行によるセクタキャッシュの指定の仕方の例を示す。指示行の書式については本スライドの後で説明する。

■ 図のDOループの実行中、ループ中の配列について、配列Aをキャッシュから追い出されないようにしたいとする。

■ ①の行で、キャッシュから追い出されないようにしたい配列Aの入るウェイ（セクタ1）の数を指定する。本例では2ウェイを使用すると指定する。有効範囲は①と④の間となる。

①と④の指示行の書式は本スライドの後で説明する。

■ 図の②の行で、セクタ1に入れたい配列名（本例ではA）を指定する。有効範囲は②と③の間となる。

②と③の指示行の書式は本スライドの後で説明する。

ループ例（セクタキャッシュ指定）

```
!OCL SCACHE_ISOLATE_WAY(L2=2)     ①
!OCL SCACHE_ISOLATE_ASSIGN(A)     ②
  DO J=1,M
    DO I=1,N
      X(I,J) = A(I) + X(I,J)
    ENDDO
  ENDDO
!OCL END_SCACHE_ISOLATE_ASSIGN    ③
!OCL END_SCACHE_ISOLATE_WAY       ④
```

```
!OCL SCACHE_ISOLATE_WAY(L2=2)     ①
!OCL SCACHE_ISOLATE_ASSIGN(A)     ②
  DO J=1,M
    DO I=1,N
      X(I,J) = A(I) + X(I,J)
    ENDDO
  ENDDO
!OCL END_SCACHE_ISOLATE_ASSIGN    ③
!OCL END_SCACHE_ISOLATE_WAY       ④
```

■ セクタキャッシュに関する指示行 [1/2]

```
!OCL SCACHE_ISOLATE_WAY(L2=m[,L1=n])
```
および
```
!OCL END_SCACHE_ISOLATE_WAY
```

■ !OCL SCACHE_ISOLATE_WAYはキャッシュのセクタ1の最大ウェイ数を指示する(*1)。

- L2=*m*で2次キャッシュにおけるセクタ1の最大ウェイ数を*m*に指示する(*2)。
- L1=*n*で1次キャッシュにおけるセクタ1の最大ウェイ数を*n*に指示する(*3)。

L1=*n*の指定は省略可能である。その場合2次キャッシュのセクタ1の最大ウェイ数のみを制御する。

(*1) 範囲指定、および指定の入れ子について

指示する範囲はプログラム単位またはプログラムの一部とすることができる。

- 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!OCL SCACHE_ISOLATE_WAYを記述する。指示の有効範囲はプログラム内全体となる。
- 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!OCL SCACHE_ISOLATE_WAYと!OCL END_SCACHE_ISOLATE_WAYで指定する。

  なお、上記の範囲指定の指示行を入れ子に記述することはできない。ただし、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ある。

(*2) *m*に指定できる範囲は以下の通りである。

- アシスタントコアを含まないCMGの場合 $0 \leqq m \leqq 16$ （16は2次キャッシュのウェイ数）
- アシスタントコアを含むCMGの場合、アシスタントコアが2次キャッシュの2ウェイを常時使用するため $0 \leqq m \leqq 14$

(*3) *n*に指定できる範囲は以下の通りである。$0 \leqq n \leqq 4$ （4は1次キャッシュのウェイ数）

■ セクタキャッシュに関する指示行 [2/2]

```
!OCL SCACHE_ISOLATE_ASSIGN(array1[,array2]...)
```
および
```
!OCL END_SCACHE_ISOLATE_ASSIGN
```

```
!OCL SCACHE_ISOLATE_WAY(L2=2)      ①
!OCL SCACHE_ISOLATE_ASSIGN(A)      ②
  DO J=1,M
    DO I=1,N
      X(I,J) = A(I) + X(I,J)
    ENDDO
  ENDDO
!OCL END_SCACHE_ISOLATE_ASSIGN     ③
!OCL END_SCACHE_ISOLATE_WAY        ④
```

■ !OCL SCACHE_ISOLATE_ASSIGNはキャッシュのセクタ1に載せる配列を指示する(*4)。
ただし、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る。

(*4) 範囲指定、および指定の入れ子について

指示する範囲はプログラム単位またはプログラムの一部とすることができる。

- 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!OCL SCACHE_ISOLATE_ASSIGNを記述する。指示の有効範囲はプログラム内全体となる。
- 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!OCL SCACHE_ISOLATE_ASSIGNと!OCL END_SCACHE_ISOLATE_ASSIGNで指定する。

  なお、上記の範囲指定の指示を入れ子に記述することはできない。ただし、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ある。

内容

- 「富岳」の概略
- プロファイラ（CPU性能解析レポート）
- 各種最適化手法
- 【付録】

内容

■ 【付録】最適化オプション・指示行のFortranとC/C++簡易対応表

本付録は、時間の都合上、口頭での説明は省略します。

この付録では、中級編第三部の本文で扱ったFortranでの各種最適化を有効にするコンパイルオプションおよび指示行について、C/C++ tradモード、およびC/C++ clangモードでのそれぞれコンパイルオプションおよび指示行との対応を記します。

ページ数の都合上、第三部本文で扱ったオプション・指示行のうち、最適化を有効化するオプション・指示行について対応関係のみ記します。また、中級編第一部で扱った各種最適化（SIMD化、ソフトウェアパイプライニング、プリフェッチ）のコンパイルオプションと指示行は省略します（中級編第一部本文に記されている対応表を参照して下さい）。

本付録で記すC/C++ tradモードの各オプション・指示行の機能や注意事項はFortranの場合と概ね同じです。C/C++ clangモードの各オプション・指示行の詳細は「C言語使用手引書」または「C++言語使用手引書」を参照して下さい。

※ 以降のスライドに示す指示行を有効にするには、FortranおよびC/C++ tradモードでは-Koclオプションを有効にする必要があり、C/C++ clangモードでは-ffj-oclオプションを有効にする必要があります。

■ 積和演算命令の使用

	Fortran	C/C++ tradモード	C/C++ clangモード
有効化オプション	-Kfp_contract	-Kfp_contract	-ffp-contract=fast または -ffp-contract=on
有効化指示行	!OCL FP_CONTRACT (*1)	#pragma {global|procedure|loop} fp_contract (*2)	#pragma clang fp contract(fast) または #pragma clang fp contract(on) (*3)

- C/C++ tradモードのオプション-Kfp_contractは、C/C++ clangモードでオプション-ffp-contract=fastに置き換わる。

■ 逆数近似命令の使用

	Fortran	C/C++ tradモード	C/C++ clangモード
有効化オプション	-Kfp_relaxed	-Kfp_relaxed	-ffj-fp-relaxed
有効化指示行	!OCL FP_RELAXED (*1)	#pragma {global|procedure|loop} fp_relaxed (*2)	なし

- C/C++ tradモードのオプション-Kfp_relaxedは、C/C++ clangモードでオプション-ffj-fp-relaxedに置き換わる。

(*1) プログラム単位、DOループ単位、および配列代入文単位のみで指定可能。
(*2) global：翻訳単位、procedure：関数単位、およびloop：ループ単位のみで指定可能。
(*3) 文単位のみで指定可能。

■ マルチ演算関数への変換

下線付きがデフォルト

	Fortran	C/C++ tradモード	C/C++ clangモード
有効化オプション	-Kmfunc[={1|2|3}]	-Kmfunc[={1|2|3}]	なし
有効化指示行	!OCL MFUNC[({1|2|3})] (*1)	#pragma {global|procedure|loop} mfunc [{1|2|3}] (*2)	なし

■ マルチ演算関数の対象となる組込み関数および演算

■ Fortranの場合

【単精度実数型および倍精度実数型】ACOS、ASIN、[ATAN]、[ATAN2]、[COS]、ERF、ERFC、[EXP]、[EXP10]、[LOG]、[LOG10]、[SIN]、べき乗演算
【単精度複素数型よび倍精度複素数型】ACOS、ACOSH、ASIN、ASINH、[ATAN]、ATANH、[COS]、COSH、COSQ、[EXP]、[LOG]、[SIN]、SINH、SINQ、[TAN]、TANH
【整数型】ISHFT

■ C/C++ tradモードの場合

【単精度実数型および倍精度実数型】acos、asin、[atan]、[atan2]、[cos]、erf、erfc、[exp]、[log]、[log10]、[sin]、[pow]

[]：オプション-Kilfuncによる組込み関数のインライン展開の対象。

上記関数のマルチ演算関数の注意事項については「Fortran使用手引書」、「C言語使用手引書」および「C++言語使用手引書」を参照して下さい。

(*1) プログラム単位、DOループ単位、および配列代入文単位のみで指定可能。
(*2) global：翻訳単位、procedure：関数単位、およびloop：ループ単位のみで指定可能。

■ 組込み関数のインライン展開

下線付きがデフォルト

	Fortran	C/C++ tradモード	C/C++ clangモード
有効化オプション	-Kilfunc [={loop|procedure}]	-Kilfunc [={loop|procedure}]	-ffj-ilfunc [={loop|procedure}]
有効化指示行	!OCL ILFUNC (*1)	なし	なし

- C/C++ tradモードのオプション-Kilfunc[={loop|procedure}]は、C/C++ clangモードでそれぞれオプション-ffj-ilfunc[={loop|procedure}]に置き換わる。

■ インライン展開の対象となる組込み関数および演算

■ Fortranの場合

【単精度実数型および倍精度実数型】ATAN、ATAN2、COS、EXP、EXP10、LOG、LOG10、SIN、TAN、および、べき乗演算(*2)
(*2) 基数と指数が単精度実数型同士および倍精度実数型同士
【単精度複素数型および倍精度複素数型】ABSおよびEXP

■ C/C++ tradモードおよびclangモードの場合

【単精度実数型および倍精度実数型】atan、atan2、cos、exp、log、log10、pow、sin、およびtan
【単精度複素数型および倍精度複素数型】absおよびexp

(*1) プログラム単位、DOループ単位、および配列代入文単位のみで指定可能。

■ 利用者定義手続のインライン展開

下線付きがデフォルト

	Fortran	C/C++ tradモード	C/C++ clangモード
有効化オプション	-x-	-x-	-finline-functions
	-x手続名1[,手続名2]...	-x関数名1[,関数名2]...	なし
参考：無効化オプション	-x0	-x0	-fno-inline-functions
有効化指示行	なし	なし	なし

- インライン展開候補の利用者定義手続の、実引数・仮引数の対応関係や、手続内に記述した文の属性、またはその他の条件により、インライン展開にはいくつかの制約事項が存在する。そのため、インライン展開候補の利用者定義手続が必ずしもインライン展開されるとは限らない。詳細は「Fortran使用手引書」、「C言語使用手引書」、または「C++言語使用手引書」を参照して下さい。
- C/C++ tradモードのオプション-x-、および-x0は、C/C++ clangモードでそれぞれオプション-finline-functions、および-fno-inline-functionsに置き換わる。
- FortranおよびC/C++ tradモードのオプション-xに指定できる引数は上記の他多数存在します。詳細は「Fortran使用手引書」、「C言語使用手引書」、または「C++言語使用手引書」を参照して下さい。
- インライン展開オプションのデフォルトについて：
  - Fortranでのデフォルトは-x0である。
  - C/C++ tradモードでのデフォルトは-O3以上では-x-、-O2以下では-x0である。
  - C/C++ clangモードでのデフォルトは-O2以上では-finline-functions、-O1以下では-fno-inline-functionsである。

■ 演算評価方法の変更

	Fortran	C/C++ tradモード	C/C++ clangモード
有効化オプション	-Keval	-Keval	-ffast-math
有効化指示行	!OCL EVAL (*1)	#pragma {global|procedure|loop} eval (*2)	なし

- C/C++ tradモードのオプション-Kevalは、C/C++ clangモードでオプション-ffast-mathに置き換わる。

(*1) プログラム単位、DOループ単位、および配列代入文単位のみで指定可能。
(*2) global：翻訳単位、procedure：関数単位、およびloop：ループ単位のみで指定可能。

■ CLONE最適化

	Fortran	C/C++ tradモード	C/C++ clangモード
有効化 オプション	なし	なし	なし
有効化 指示行	!OCL CLONE(*var*==*n1*[,*n2*]...) (*1)	#pragma loop clone *var*==*n1*[,*n2*]... (*2)	#pragma fj loop clone *var*==*n1*[,*n2*]... (*2)

- C/C++ tradモードの指示行#pragma loop cloneは、C/C++ clangモードで指示行#pragma fj loop cloneに置き換わる。

(*1) DOループ単位、および配列代入文単位のみで指定可能。
(*2) loop：ループ単位のみで指定可能。

■ 本文「レジスタ不足の回避」で扱ったコンパイルオプションおよび指示行

本文「レジスタ不足の回避」で扱った以下に示すFortranのコンパイルオプションおよび指示行に対応するC/C++のコンパイルオプションおよび指示行は中級編第一部で説明していますので、本付録では省略します。

-Kswp_strong および !OCL SWP
-Kswp_freg_rate=*N* および !OCL SWP_FREG_RATE(*N*)

■ 自動ループ分割の指示行

下線付きがデフォルト

	Fortran	C/C++ tradモード	C/C++ clangモード
有効化指示行	!OCL LOOP_FISSION_TARGET または !OCL LOOP_FISSION_TARGET(CL) または !OCL LOOP_FISSION_TARGET(LS) (*1)	#pragma loop loop_fission_target または #pragma loop loop_fission_target cl または #pragma loop loop_fission_target ls (*2)	#pragma fj loop loop_fission_target または #pragma fj loop loop_fission_target cl (*2)

- C/C++ tradモードの指示行#pragma loop loop_fission_target [cl]は、C/C++ clangモードで指示行#pragma fj loop loop_fission_target [cl]に置き換わる。
- 自動ループ分割の指示行を有効にするには、FortranおよびC/C++ tradモードでは-Koclに加え下記の-Kloop_fissionを有効に、C/C++ clangモードでは-ffj-oclに加え下記の-ffj-loop-fissionを有効にする必要がある。

(*1) DOループ単位、および配列代入文単位のみで指定可能。
(*2) loop：ループ単位のみで指定可能。

■ 自動ループ分割の指示行を有効化するオプション

	Fortran	C/C++ tradモード	C/C++ clangモード
有効化オプション	-Kloop_fission	-Kloop_fission	-ffj-loop-fission

- C/C++ tradモードのオプション-Kloop_fissionは、C/C++ clangモードでオプション-ffj-loop-fissionに置き換わる。

■ SIMD化前のフルアンローリング

	Fortran	C/C++ tradモード	C/C++ clangモード
有効化オプション	なし	なし	なし
有効化指示行	!OCL FULLUNROLL_PRE_SIMD[(*n*)] 2 ≦ *n* ≦ 100 (*1)	#pragma loop fullunroll_pre_simd [*n*] 2 ≦ *n* ≦ 100 (*2)	なし

■ 高速ストア（ZFILL）

	Fortran	C/C++ tradモード	C/C++ clangモード
有効化オプション	-Kzfill[=*N*] 1 ≦ *N* ≦ 100	-Kzfill[=*N*] 1 ≦ *N* ≦ 100	-ffj-zfill[=*N*] 1 ≦ *N* ≦ 100
有効化指示行	!OCL ZFILL[(*N*)] (*1)	#pragma loop zfill [*N*] (*2)	#pragma fj loop zfill [*N*] (*2)

- C/C++ tradモードのオプション-Kzfill[=*N*]は、C/C++ clangモードでオプション-ffj-zfill[=*N*]に置き換わる。
- C/C++ tradモードの指示行#pragma loop zfillは、C/C++ clangモードで指示行#pragma fj loop zfillに置き換わる。

(*1) DOループ単位、および配列代入文単位のみで指定可能。
(*2) loop：ループ単位のみで指定可能。

■ ループブロッキング

	Fortran	C/C++ tradモード	C/C++ clangモード
有効化 オプション	-Kloop_blocking[=*N*] 2 ≦ *N* ≦ 10000	-Kloop_blocking[=*N*] 2 ≦ *N* ≦ 10000	なし
有効化 指示行	!OCL LOOP_BLOCKING(*n*) 2 ≦ *n* ≦ 10000 (*1)	#pragma {global|procedure|loop} loop_blocking n 2 ≦ n ≦ 10000 (*2)	なし

■ アンロール・アンド・ジャム

	Fortran	C/C++ tradモード	C/C++ clangモード
有効化 オプション	-Kunroll_and_jam[=*N*] 2 ≦ *N* ≦ 100	-Kunroll_and_jam[=*N*] 2 ≦ *N* ≦ 100	なし
有効化 指示行	!OCL UNROLL_AND_JAM[(*N*)] (*3) または !OCL UNROLL_AND_JAM_FORCE[(*N*)] (*4)	#pragma {global|procedure|loop} unroll_and_jam [*N*] (*2) または #pragma loop unroll_and_jam_force [*N*] (*5)	なし

(*1) プログラム単位、DOループ単位、および配列代入文単位のみで指定可能。
(*2) global：翻訳単位、procedure：関数単位、およびloop：ループ単位のみで指定可能。
(*3) プログラム単位、およびDOループ単位のみで指定可能。
(*4) DOループ単位のみで指定可能。
(*5) ループ単位のみで指定可能。

■ セクタキャッシュ（キャッシュのセクタ1の最大ウェイ数を指示）

	Fortran	C/C++ tradモード	C/C++ clangモード
有効化オプション	なし	なし	なし
有効化指示行	!OCL SCACHE_ISOLATE_WAY(L2=*m*[,L1=*n*]) (*1) および !OCL END_SCACHE_ISOLATE_WAY (*2)	#pragma {procedure|statement} scache_isolate_way L2=*m* [L1=*n*] (*3)および #pragma statement end_scache_isolate_way (*4)	なし

■ セクタキャッシュ（キャッシュのセクタ1に載せる配列を指示）

	Fortran	C/C++ tradモード	C/C++ clangモード
有効化オプション	なし	なし	なし
有効化指示行	!OCL SCACHE_ISOLATE_ASSIGN (*array1*[,*array2*]...) (*1) および !OCL END_SCACHE_ISOLATE_ASSIGN (*2)	#pragma {procedure|statement} scache_isolate_assign *array1*[,*array2*]... (*3) および #pragma statement end_scache_isolate_assign (*4)	なし

(*1) プログラム単位、および文単位のみで指定可能。
(*2) 文単位のみで指定可能。
(*3) procedure：関数単位、およびstatement：文単位のみで指定可能。
(*4) statement：文単位のみで指定可能。

内容

- 「富岳」の概略
- プロファイラ（CPU性能解析レポート）
- 各種最適化手法
- 【付録】
  - 最適化オプション・指示行のFortranとC/C++簡易対応表
  - 参考文献

[1] Fortran使用手引書
[2] C言語使用手引書
[3] C++言語使用手引書
[4] Fortran翻訳時メッセージ
[5] プロファイラ使用手引書
[6] プログラミングガイド チューニング編
[7] https://www.r-ccs.riken.jp/fugaku/system/

※ [1]～[6]はスーパーコンピュータ「富岳」のサイトにアクセス可能ならば以下のURLよりダウンロード可能です。

https://www.fugaku.r-ccs.riken.jp/docs/manuals_r01

※ 本中級編第一部および第二部のテキストは「富岳」のサイトにアクセス可能ならば以下のURLよりダウンロード可能です。

https://www.fugaku.r-ccs.riken.jp/docs/workshop_materials_2022

以上