プログラミングガイド
チューニング編

2023年3月

v2.2

富士通株式会社

はじめに

FUJITSU

- 当資料は、A64FX プロセッサを対象としたアプリケーション開発者やチューニング実施者のためのものです。
- 注意
  - FortranとC/C++やtradモードとclangモードではコンパイラが異なるため、チューニングの方法が異なる場合や対応するチューニング方法がない場合があります
  - tradモードとclangモードで類似のチューニング手法の場合、clangモードのチューニングを割愛します
- 当資料と併せて以下も参照してください。
  - Fortran 使用手引書
  - C/C++ 使用手引書
  - プログラミングガイド プロセッサ編
  - プログラミングガイド プログラミング共通編
  - プログラミングガイド Fortran編
- 当資料の記載においては、以下の文章を参考にしています。
  - A64FX 論理仕様書
  - A64FX®Microarchitecture Manual
  - ARM® Architecture Reference Manual (ARMv8 , ARMv8.1 , ARMv8.2 , ARMv8.3)
  - ARM® Architecture Reference Manual Supplement The Scalable Vector Extension

- 商標について
  - Linux® は、Linus Torvalds 氏の米国およびそのほかの国における登録商標または商標です。
  - Red Hat は、Red Hat,Inc. の米国およびそのほかの国における登録商標または商標です。
  - ARMは、ARM Ltd.の英国およびその他の国における登録商標です。
  - そのほかの会社名、製品名等の固有名詞は、各社の登録商標または商標です。
  - 本資料に記載されているシステム名、製品名等には、必ずしも商標表示(®、™)を付記していません
- 謝辞
  - C/C++のチューニングについては、国立研究開発法人 理化学研究所 計算科学研究センターから委託を受けて実施した事業の成果を基にしています。

当資料の説明範囲

FUJITSU

アプリケーションのチューニングは、観点によって次のようなポイントがあります。当資料はCPUチューニングとスレッド並列チューニングについて説明しています。

当資料の説明範囲

	1コア チューニング	スレッド並列 チューニング	プロセス並列 チューニング	超高並列 チューニング
チューニングポイント	✓ I/Oを減らす ✓ 演算量を減らす ✓ SIMD化を促進する ✓ メモリアクセスを減らす ✓ キャッシュ利用率を高める	✓ 並列化率を上げる ✓ 並列化粒度を上げる ✓ スレッド間の同期コストを削減する ✓ ロードバランスを均等化する	✓ 並列化率を上げる ✓ 並列化粒度を上げる ✓ プロセス間の通信コストを削減する ✓ ロードバランスを均等化する	✓ 並列化率を上げる ✓ 並列化粒度を上げる ✓ プロセス間の通信コストを削減する ✓ ロードバランスを均等化する ✓ グローバルな通信コストを削減する

目次 (1/2)

目次 (2/2)

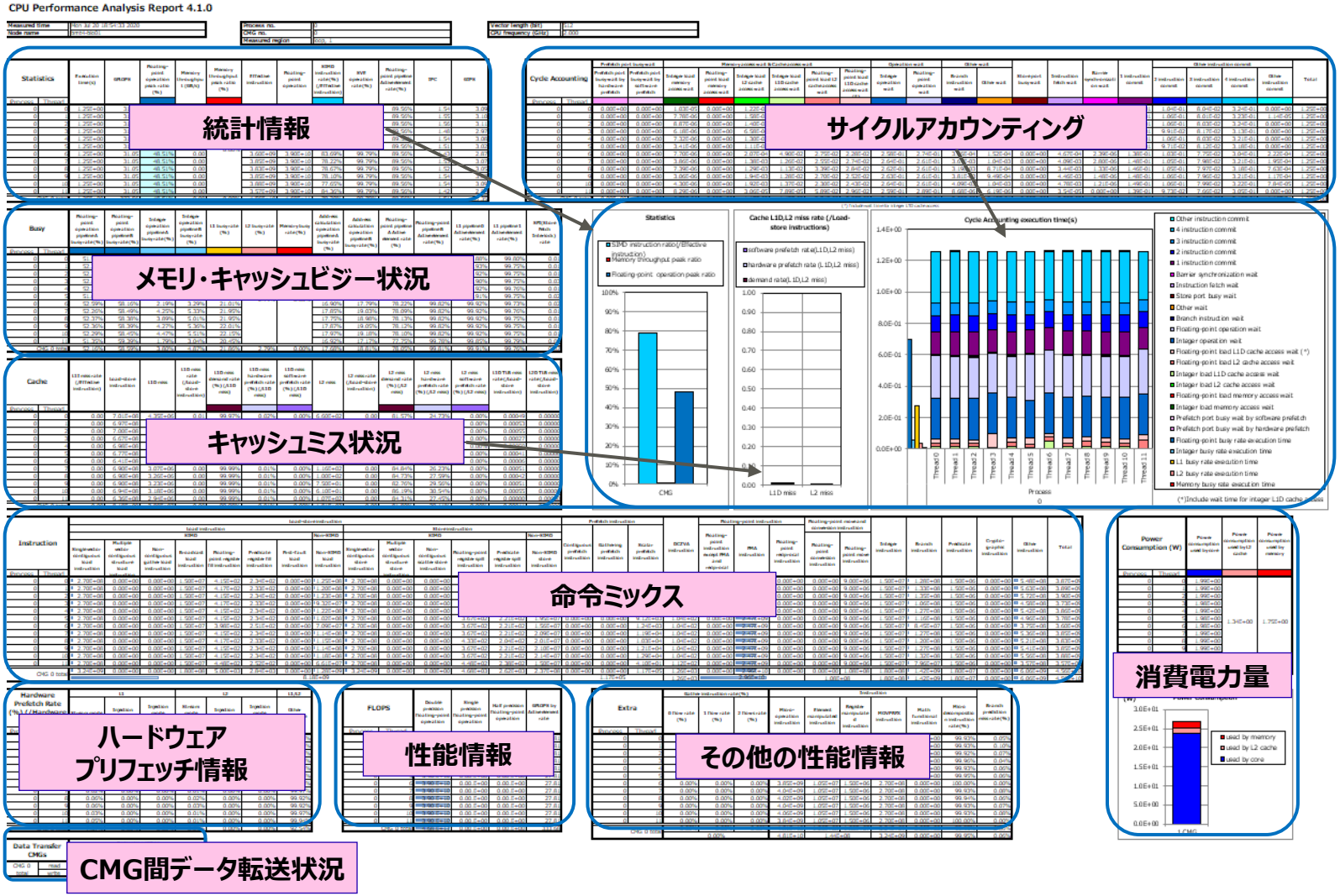
FUJITSU

ボトルネックの調査

CPU性能解析レポート：全体

性能ボトルネックの抽出手段には、CPU性能解析レポートの利用を薦めます。

CPU性能解析レポートでは、以下のような豊富な種類のPA(Performance Analysis)イベントを計測でき、アプリケーションプログラム実行時のCPU動作状態を調べることができます。



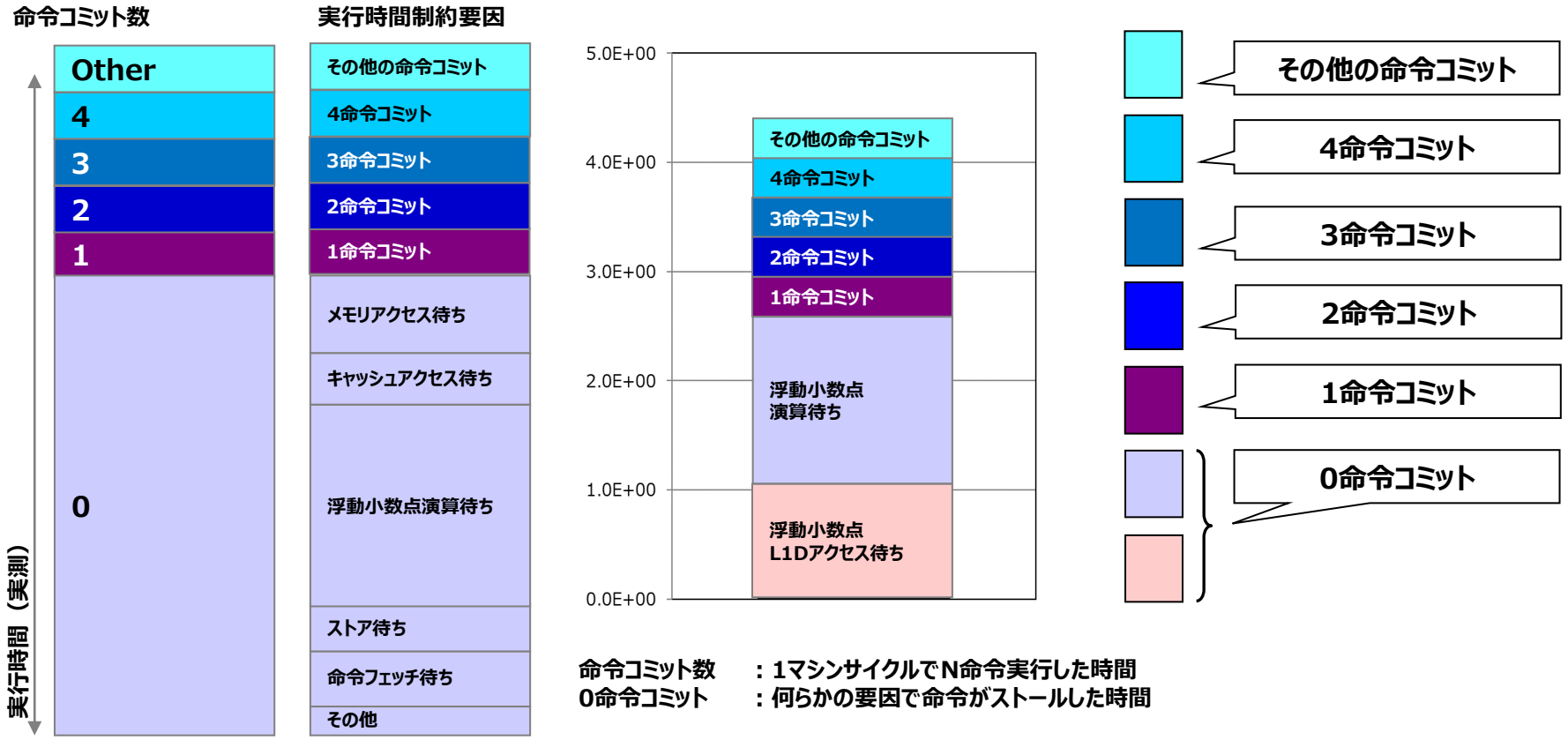
CPU性能解析レポート：サイクルアカウンティングとは

FUJITSU

サイクルアカウンティングとは、性能ボトルネックの要因分析手法です。

サイクルアカウンティング情報は、CPU性能解析レポートの右上部に掲載されています。

サイクルアカウンティングでは、あるアプリケーションプログラムを実行するためにかかった総時間（CPUサイクル数）をCPUの動作状態で分類してグラフ化します。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ます。



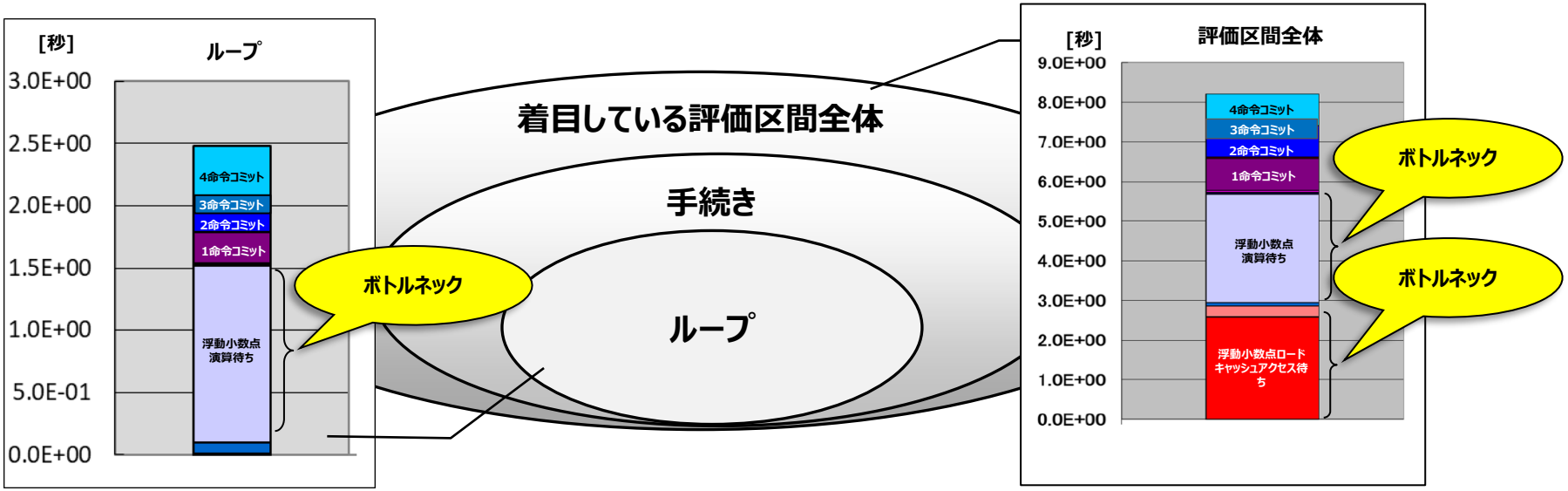
命令コミット数　：1マシンサイクルでN命令実行した時間
0命令コミット　：何らかの要因で命令がストールした時間

サイクルアカウンティングを活用したボトルネック抽出

● ボトルネックの把握

チューニングの基本はボトルネックを把握することです。

評価区間のボトルネックはサイクルアカウンティングから判断することができます。



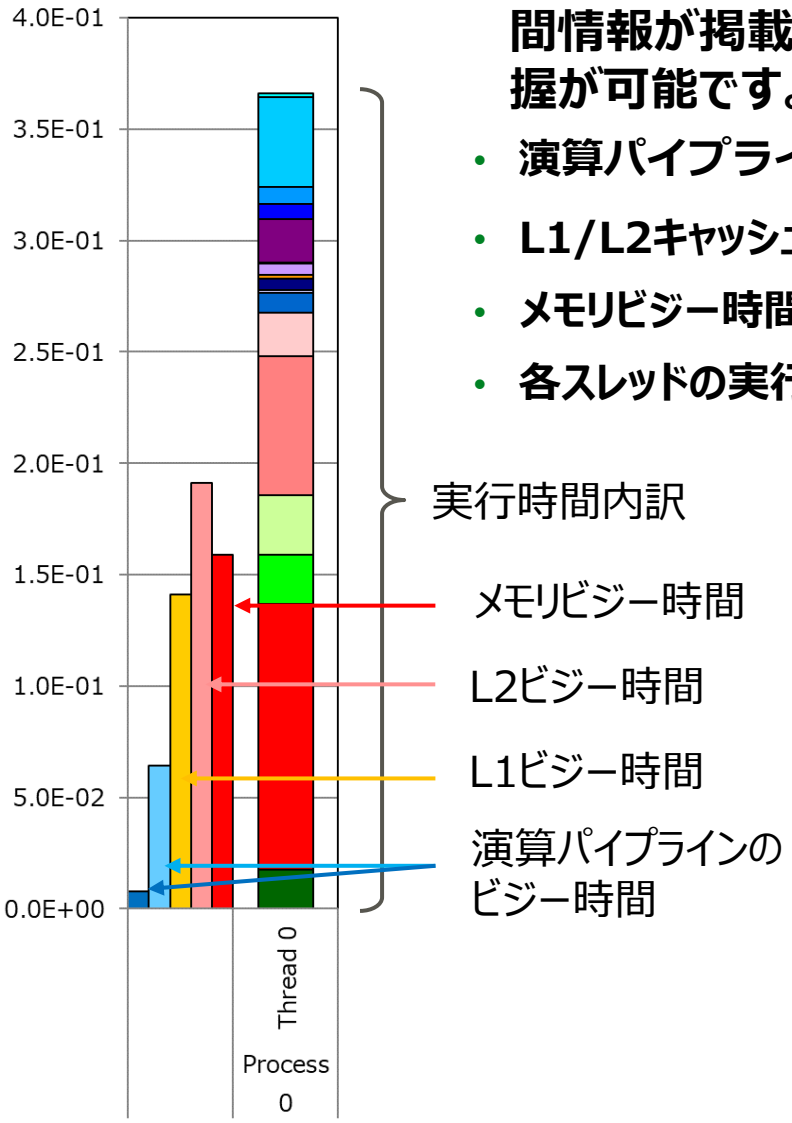
● チューニングへの活用

- **ボトルネックの改善にはどのような対策が必要か**
- **どの程度改善することができるのか**

を判断するためには

ループ単位までブレークダウンして分析した方が良い。

各種ビジー時間を活用した（バンド幅）ボトルネック抽出



- **サイクルアカウンティングの左側のグラフには以下のビジー時間情報が掲載されています。各種バンド幅のボトルネック把握が可能です。**
  - **演算パイプラインのビジー時間**
  - **L1/L2キャッシュビジー時間**
  - **メモリビジー時間**
  - **各スレッドの実行時間内訳**

- **メモリビジー時間**
  メモリへのデータ転送量が多い場合に発生します。

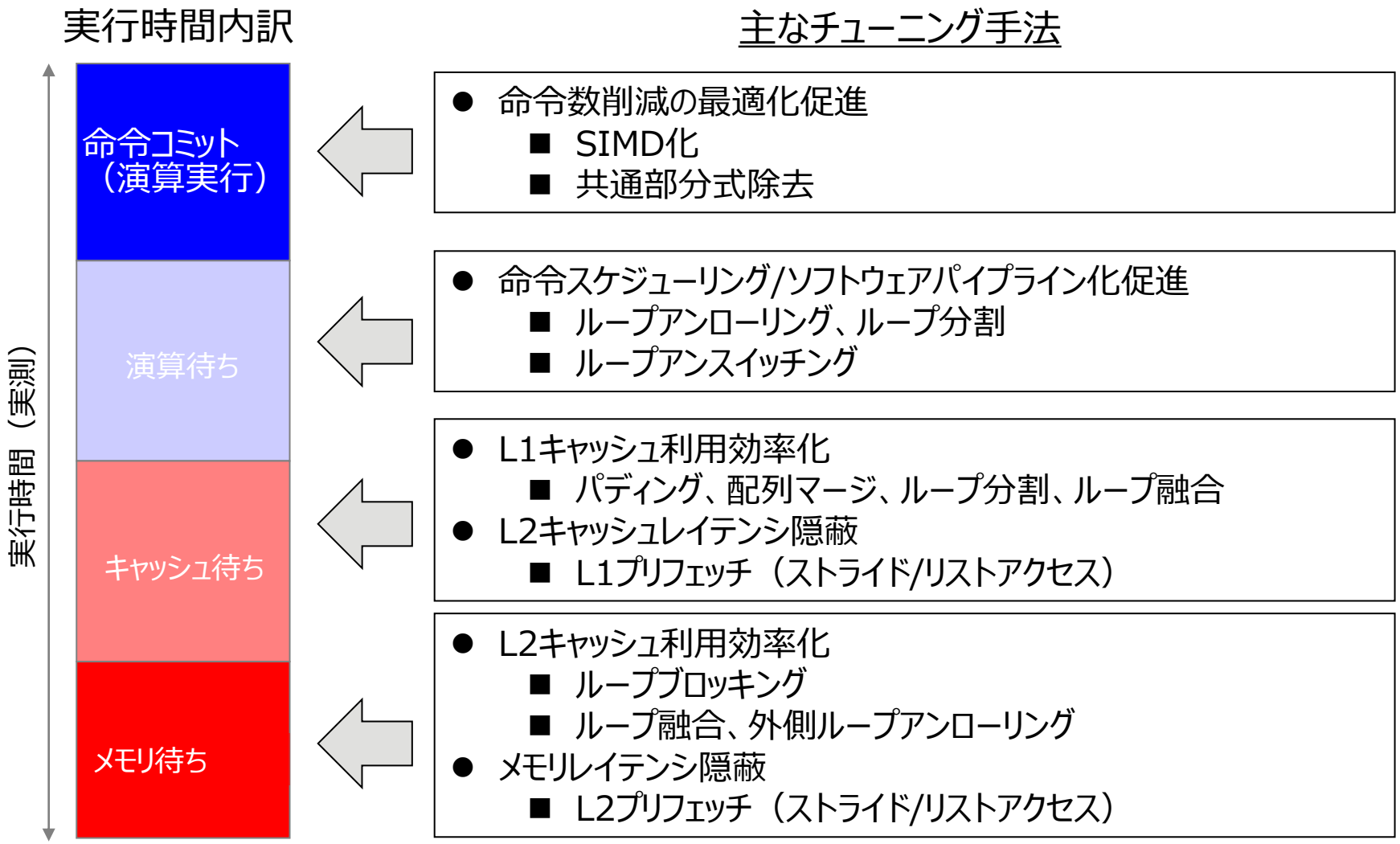
- **L1/L2キャッシュビジー時間**
  L1/L2キャッシュへのデータ転送量が多い場合に発生します。

- **演算パイプラインのビジー時間**
  演算量が多い場合に発生します。

サイクルアカウンティングを活用したチューニング方法

FUJITSU

- **サイクルアカウンティングの結果から、チューニング手法を選択します。**
  次に主なチューニング手法を示します。
  より詳細な内容はチューニングマップを参照してください。



チューニングマップ：分類と状態

FUJITSU

ボトルネックの分類	PAグラフから見える高コスト	PA情報から見える高コスト	状態
メモリネック	浮動小数点ロードメモリアクセス待ち	-	メモリレイテンシがボトルネックになっています。
	整数ロードメモリアクセス待ち	-	
	ストア待ち	-	ストア命令のコストがボトルネックになっています。
	メモリ・キャッシュビジー待ち	-	メモリスループットがボトルネックになっています。
	-	メモリビジー率が高い	
	-	L2ミス率が高い L2ミスdm率が高い	メモリレイテンシがボトルネックになっています。
L2キャッシュネック	浮動小数点ロードL2アクセス待ち	-	L2キャッシュレイテンシがボトルネックになっています。
	整数ロードL2アクセス待ち	-	
	-	L2ビジー率が高い	L2キャッシュスループットがボトルネックになっています。
	-	L1Dミス率が高い L1Dミスdm率が高い	L2キャッシュレイテンシがボトルネックになっています。
L1キャッシュネック	浮動小数点ロードL1Dアクセス待ち	-	L1キャッシュレイテンシがボトルネックになっています。
	整数ロードL1Dアクセス待ち	-	
	-	L1ビジー率が高い	L1キャッシュスループットがボトルネックになっています。
スケジューリングネック	浮動小数点演算待ち	-	演算命令のレイテンシがボトルネックになっています。
	整数演算待ち	-	
	分岐命令待ち	-	分岐命令がボトルネックになっています。
並列化ネック	バリア同期待ち	-	スレッド並列化されていない部分がボトルネックになっています。
ロードインバランスネック	バリア同期待ち	命令バランスのmax-min差が大きい	スレッド間のロードインバランスがボトルネックになっています。
TLBネック	-	mDTLBミス率が高い	TLBミスやTLBスラッシングがボトルネックになっています。
	-	uDTLBミス率が高い	TLBミスがボトルネックになっています。
命令フェッチ	命令フェッチ待ち	-	命令キャッシュミスやスラッシングがボトルネックになっています。
命令数ネック	命令コミット：その他の命令コミット	-	命令数がボトルネックになっています。
	命令コミット：4命令コミット		
	命令コミット：3命令コミット		
	命令コミット：2命令コミット		
	命令コミット：1命令コミット		
その他	その他の待ち	-	PAが正しく採取できていない可能性があります。

チューニングマップ（1/4）

FUJITSU

スループットネック

PAグラフから見える高コスト	PA情報から見える高コスト	状態	チューニング案
メモリ・キャッシュビジー待ち	-	メモリスループットがボトルネックになっています。	データアクセス待ちの改善 －配列次元移動 －ループブロッキング －ストリップマイニング －高速ストア（ZFILL）
-	メモリビジー率が高い	メモリスループットがボトルネックになっています。	データアクセス待ちの改善 －配列次元移動 －ループブロッキング －ストリップマイニング －高速ストア（ZFILL）
-	L2ミス率が高い L2ミスdm率が高い	メモリレイテンシがボトルネックになっています。	データアクセス待ちの改善 －配列次元移動 －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 －プリフェッチに関する改善 －スラッシングの改善
	L2ビジー率が高い	L2キャッシュスループットがボトルネックになっています。	データアクセス待ちの改善 －配列次元移動 －ループブロッキング －ストリップマイニング
	L1ビジー率が高い	L1キャッシュスループットがボトルネックになっています。	データアクセス待ちの改善 －アルゴリズムの見直し

チューニングマップ (2/4)

FUJITSU

● レイテンシネック

PAグラフから見える高コスト	PA情報から見える高コスト	状態	チューニング案
浮動小数点ロードメモリアクセス待ち		メモリレイテンシがボトルネックになっています。	データアクセス待ちの改善 – 配列次元移動 – プリフェッチに関する改善 – ループブロッキング
整数ロードメモリアクセス待ち			
浮動小数点ロードL2アクセス待ち		L2キャッシュレイテンシがボトルネックになっています。	データアクセス待ちの改善 – 配列次元移動 – アンロールアンドジャム – プリフェッチに関する改善
整数ロードL2アクセス待ち			
-	1Dミス率が高い L1Dミスdm率が高い		データアクセス待ちの改善 – 配列次元移動 – スラッシングの改善
浮動小数点ロードL1Dアクセス待ち		L1キャッシュレイテンシがボトルネックになっています。	命令スケジューリングの改善 マイクロアーキ依存の改善
整数ロードL1Dアクセス待ち			
浮動小数点演算待ち		演算命令のレイテンシがボトルネックになっています。	命令スケジューリングの改善
整数演算待ち			

チューニングマップ (3/4)

FUJITSU

命令数ネック

PAグラフから見える高コスト		PA情報から見える高コスト	状態	チューニング案
命令コミット	その他の命令コミット 4命令コミット 3命令コミット 2命令コミット 1命令コミット		命令数がボトルネックになっています。	命令数ネックの改善 －SIMD化促進 －ソフトウェアパイプライニング促進 －プリフェッチに関する改善 －インライン展開

TLBネック

PAグラフから見える高コスト	PA情報から見える高コスト	状態	チューニング案
-	mDTLBミス率が高い	TLBミスやTLBスラッシングがボトルネックになっています。	TLBネックの改善 －スラッシングの解消 －使用領域の変更 －ラージページのオプションによる最適化
-	uDTLBミス率が高い	TLBミスがボトルネックになっています。	TLBネックの改善 －ページサイズの拡張

チューニングマップ（4/4）

FUJITSU

- **その他**

PAグラフから見える高コスト	PA情報から見える高コスト	状態	チューニング案
ストア待ち		ストア命令のコストがボトルネックになっています。	データアクセス待ちの改善 －配列次元移動 －プリフェッチに関する改善 －高速ストア（ZFILL）
分岐命令待ち		分岐命令がボトルネックになっています。	命令スケジューリングの改善 －IF文の除去 －マスク付きSIMD
バリア同期待ち		スレッド並列化されていない部分がボトルネックになっています。	スレッド並列化率の改善
	命令バランスのmax-min差が大きい	スレッド間のロードインバランスがボトルネックになっています。	スレッド並列実行効率の改善
命令フェッチ待ち		命令キャッシュミスやスラッシングがボトルネックになっています。	命令フェッチの改善 －ループボディの縮小 －アルゴリズムの見直し －スラッシングの解消

FUJITSU

1コアチューニング データアクセス待ち （データ局所性の向上）

- データ局所性とは
- ストリップマイニング
- ループブロッキング
- セクタキャッシュ
- ループ交換
- ループ融合
- 配列マージ（インダイレクトアクセスの効率改善）
- 配列次元移動
- アンロールアンドジャム

データ局所性とは

FUJITSU

データ局所性とは、データの参照やアクセスが狭い範囲に集中している度合いです。
データ局所性が低いと、キャッシュメモリ上のデータが利活用されず、メモリアクセス負荷が大きくなります。
ソースチューニングによりデータ局所性を高めることで、メモリアクセスの負荷が軽減され、データアクセス待ちが改善することができます。

データ局所性の向上には、以下のような手法が有効です。

- ストリップマイニング
- ループブロッキング
- セクタキャッシュ
- ループ交換
- ループ融合
- 配列マージ（インダイレクトアクセスの効率改善）
- 配列次元移動
- アンロールアンドジャム

FUJITSU

ストリップマイニング

- ストリップマイニングとは
- ストリップマイニング（改善前）
- ストリップマイニングの効果（ソースチューニング）

ストリップマイニングとは

ストリップマイニングとは、ループをより小さなセグメントやストリップに断片化することで、キャッシュ効率を向上させる手法です。

改善前ソース例

```
integer n,m
real*8 a(n,m),b(n,m),c(n,m),d(n,m),e(n,m)

do j=1,m
  do i=1,n
    a(i,j)=b(i,j)+c(i,j)
  enddo
  do i=1,n-100
    d(i,j)=a(i,j)+e(i,j)
  enddo
enddo
```

n=100*1000/8
m=20

ループ1

ループ2

繰返し数が多いため、キャッシュに載せた配列a のデータが上書きされてしまう。

配列a はキャッシュミスする。

改善後ソース例

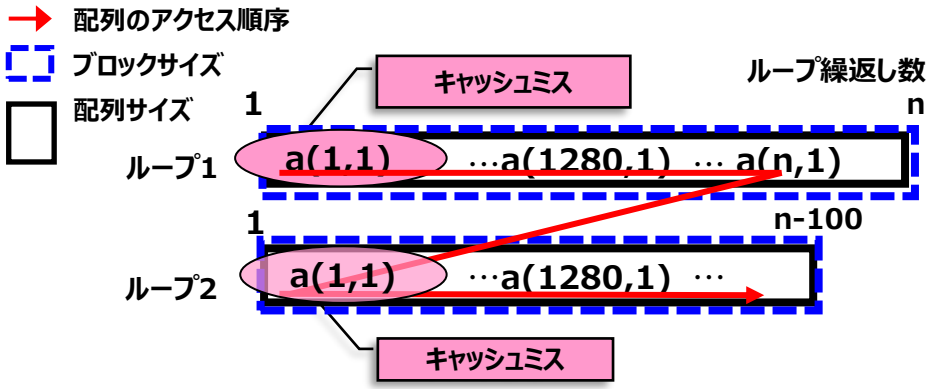
```
integer n,m
real*8 a(n,m),b(n,m),c(n,m),d(n,m),e(n,m)
blki=10*1024/8

do j=1,m
  do ii=1,n,blki
    do i=ii,min(ii+blki-1,n)
      a(i,j)=b(i,j)+c(i,j)
    enddo
    do i=ii,min(ii+blki-1,n-100)
      d(i,j)=a(i,j)+e(i,j)
    enddo
  enddo
enddo
```

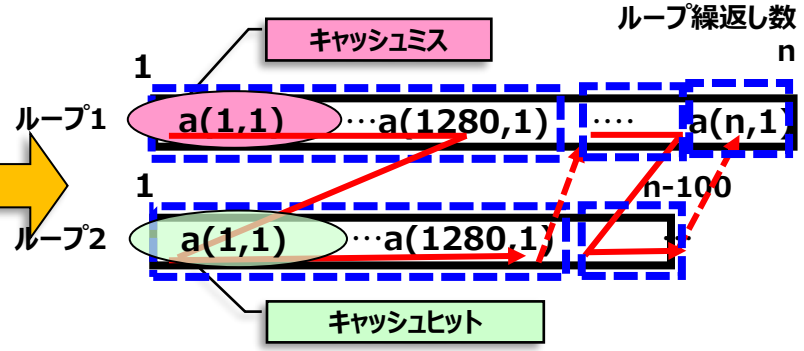
ブロックサイズ

ループ1

ループ2

配列a のデータがキャッシュ上に残るため、キャッシュヒットする。





Fortran ストリップマイニング（改善前）

FUJITSU

ループ1の繰返し数が多く、配列データはキャッシュに載りきらないため、ループ2で再利用できません。そのため、メモリ・キャッシュビジー待ちが多くなっています。

改善前ソース

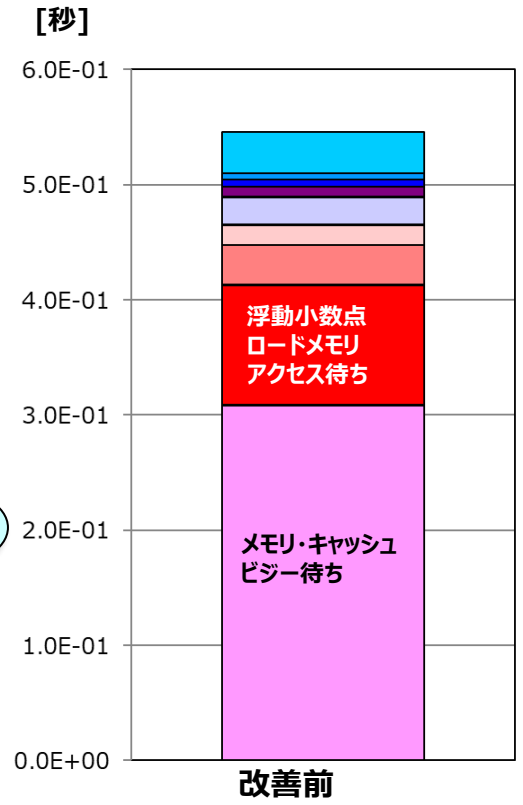
```
32              !$omp parallel do reduction(+:s1)
                <<< Loop-information Start >>>
                <<<  [OPTIMIZATION]
                <<<    PREFETCH(HARD) Expected by compiler :
                <<<      d, c, b, a, e
                <<< Loop-information  End >>>
33    1  p          do j=1,m
                <<< Loop-information Start >>>
                <<<  [OPTIMIZATION]
                <<<    SIMD(VL: 8)
                <<<    PREFETCH(HARD) Expected by compiler :
                <<<      b, c, a, d
                <<< Loop-information  End >>>
34    2  p  8v         do i=1,n
35    2  p  8v           s1 = s1 + a(i,j) * (s3 * b(i,j) + c(i,j) * (s2 + s3 * d(i,j) ))
36    2  p  8v         enddo
                <<< Loop-information Start >>>
                <<<  [OPTIMIZATION]
                <<<    SIMD(VL: 8)
                <<<    SOFTWARE PIPELINING
                <<<    PREFETCH(HARD) Expec
                <<<      b, a, d, c, e
                <<< Loop-information  End >>>
37    2  p  2v         do i=1,n-100
38    2  p  2v           e(i,j) = s2 * (a(i,j) + b(i,j) * (s3 + c(i,j) * d(i,j)))
39    2  p  2v         enddo
40    1  p          enddo
```

ループ繰返し数　　：　375000
配列サイズの合計　：　12MB
繰返し数が多いため、配列データがキャッシュに載りきらない

ループ1

配列アクセスがキャッシュミスする

ループ2



データの再利用ができないため、L1D miss rateとL2 miss rateがストリームアクセスの理論値である0.20前後になっている。

Cache

	L1I miss rate (/Effective instruction)	Load-store instruction	L1D miss	L1D miss rate (/Load-store instruction)	L1D miss demand rate (%) (/L1D miss)	L1D miss hardware prefetch rate (%) (/L1D miss)	L1D miss software prefetch rate (%) (/L1D miss)	L2 miss	L2 miss rate (/Load-store instruction)	L2 miss demand rate (%) (/L2 miss)	L2 miss hardware prefetch rate (%) (/L2 miss)	L2 miss software prefetch rate (%) (/L2 miss)
改善前	0.00	2.08E+09	4.23E+08	0.20	7.61%	92.40%	-0.01%	4.23E+08	0.20	5.17%	95.41%	0.00%

Fortran ストリップマイニングの効果（ソースチューニング）

FUJITSU

ストリップマイニングによって、キャッシュ効率が向上し、メモリ・キャッシュビジー待ちが改善されました。

改善後ソース

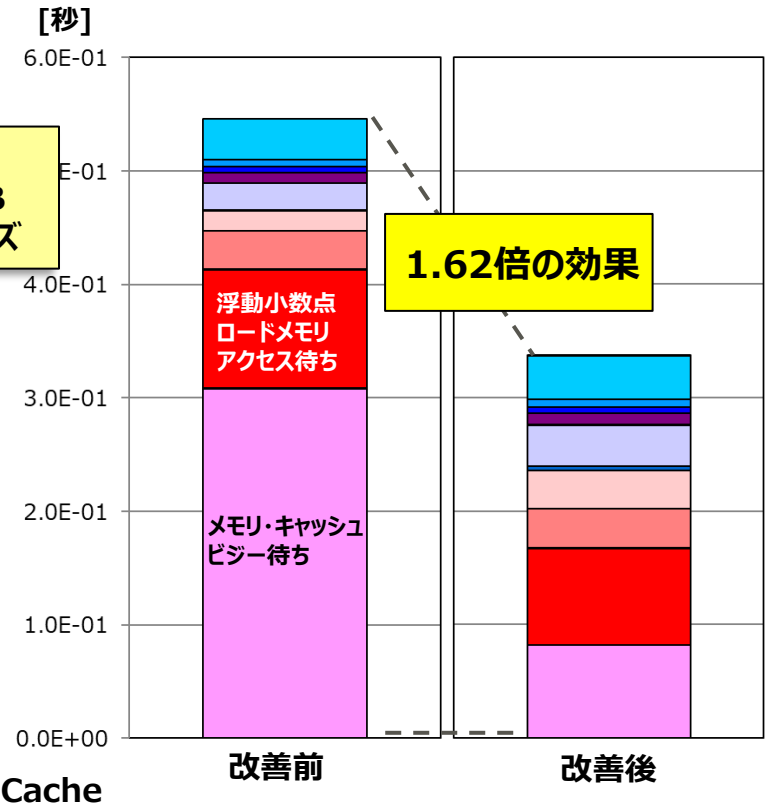
```
32                blki=4*1024/8
33
34              !$omp parallel do reduction(+:s1)
35    1  p          do j=1,m
                <<< Loop-information Start >>>
                <<<  [OPTIMIZATION]
                <<<    PREFETCH(HARD) Expected by compiler :
                <<<      d, c, b, a, e
                <<< Loop-information  End >>>
36    2  p            do ii=1,n,blki
                <<< Loop-information Start >>>
                <<<  [OPTIMIZATION]
                <<<    SIMD(VL: 8)
                <<<    PREFETCH(HARD) Expected by compiler :
                <<<      b, c, a, d
                <<< Loop-information  End >>>
37    3  p  8v           do i=ii,min(ii+blki-1,n)
38    3  p  8v             s1 = s1 + a(i,j) * (s3 * b(i,j) + &
39    3                              c(i,j) * (s2 + s3 * d(i,j) ))
40    3  p  8v           enddo
                <<< Loop-information Start >>>
                <<<  [OPTIMIZATION]
                <<<    SIMD(VL: 8)
                <<<    SOFTWARE PIPELINING(
                <<<    PREFETCH(HARD) Expec
                <<<      b, a, d, c, e
                <<< Loop-information  End >>>
41    3  p  2v           do i=ii,min(ii+blki-1,n-100)
42    3  p  2v             e(i,j) = s2 * (a(i,j) + &
43    3                              b(i,j) * (s3 + c(i,j) * d(i,j)))
44    3  p  2v           enddo
45    2  p            enddo
46    1  p          enddo
```

ブロックサイズ ： 4KB
4KB×4ストリーム=16KB
→ L1キャッシュに載るサイズ

ループ1

ループ2

配列アクセスがキャッシュヒットする



Cache

	L1I miss rate (/Effective instruction)	Load-store instruction	L1D miss	L1D miss rate (/Load-store instruction)	L1D miss demand rate (%) (/L1D miss)
改善前	0.00	2.08E+09	4.23E+08	0.20	7.61%
改善後	0.00	1.95E+09	2.35E+08	0.12	15.24%

L2 miss	L2 miss rate (/Load-store instruction)	L2 miss demand rate (%) (/L2 miss)
4.23E+08	0.20	5.17%
2.35E+08	0.12	7.84%

L1DミスとL2ミスが減少した

C/C++ ストリップマイニング（改善前）

FUJITSU

ループ1の繰返し数が多く、配列データはキャッシュに載りきらないため、ループ2で再利用できません。そのため、メモリ・キャッシュビジー待ちが多くなっています。

改善前ソース

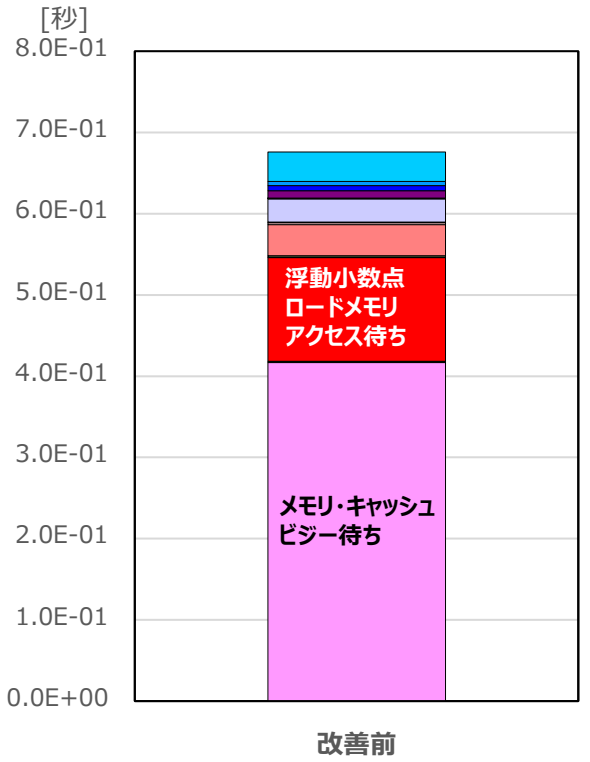
```
35          #pragma omp parallel for reduction(+:s1)
           <<< Loop-information Start >>>
           <<<  [OPTIMIZATION]
           <<<    PREFETCH(HARD) Expected by compiler :
           <<<      (unknown)
           <<< Loop-information  End >>>
36   p        for(j=0;j<m;j++) {
           <<< Loop-information Start >>>
           <<<  [OPTIMIZATION]
           <<<    SIMD(VL: 8)
           <<<    PREFETCH(HARD) Expected by compiler :
           <<<      (unknown)
           <<< Loop-information  End >>>
37   p     8v      for(i=0;i<n;i++) {
38   p     8v        s1 = s1 + a[j][i] * (s3 * b[j][i] + c[j][i] * (s2 + s3 * d[j][i]));
39   p     8v      }
           <<< Loop-information Start >>>
           <<<  [OPTIMIZATION]
           <<<    SIMD(VL: 8)
           <<<    SOFTWARE PIPELINING(IF
           <<<    PREFETCH(HARD) Expecte
           <<<      (unknown)
           <<< Loop-information  End >>>
40   p     2v      for(i=0;i<n-100;i++) {
41   p     2v        e[j][i] = s2 * (a[j][i] + b[j][i] * (s3 + c[j][i] * d[j][i]));
42   p     2v      }
43   p         }
44           }
```

ループ繰返し数　　：　375000
配列サイズの合計　：　12MB
繰返し数が多いため、配列データがキャッシュに載りきらない

ループ１

配列アクセスがキャッシュミスする

ループ２



データの再利用ができないため、L1D miss rateとL2 miss rateがストリームアクセスの理論値である0.20前後になっている。

Cache	L1I miss rate (/Effective instruction)	Load-store instruction	L1D miss	L1D miss rate (/Load-store instruction)	L1D miss demand rate (%) (/L1D miss)	L1D miss hardware prefetch rate (%) (/L1D miss)	L1D miss software prefetch rate (%) (/L1D miss)	L2 miss	L2 miss rate (/Load-store instruction)	L2 miss demand rate (%) (/L2 miss)	L2 miss hardware prefetch rate (%) (/L2 miss)	L2 miss software prefetch rate (%) (/L2 miss)
改善前	0.00	2.17E+09	4.23E+08	0.19	9.14%	90.86%	0.00%	4.23E+08	0.19	5.42%	95.53%	0.00%

C/C++ ストリップマイニングの効果（ソースチューニング）

FUJITSU

ストリップマイニングによって、キャッシュ効率が向上し、メモリ・キャッシュビジー待ちが改善されました。

改善後ソース

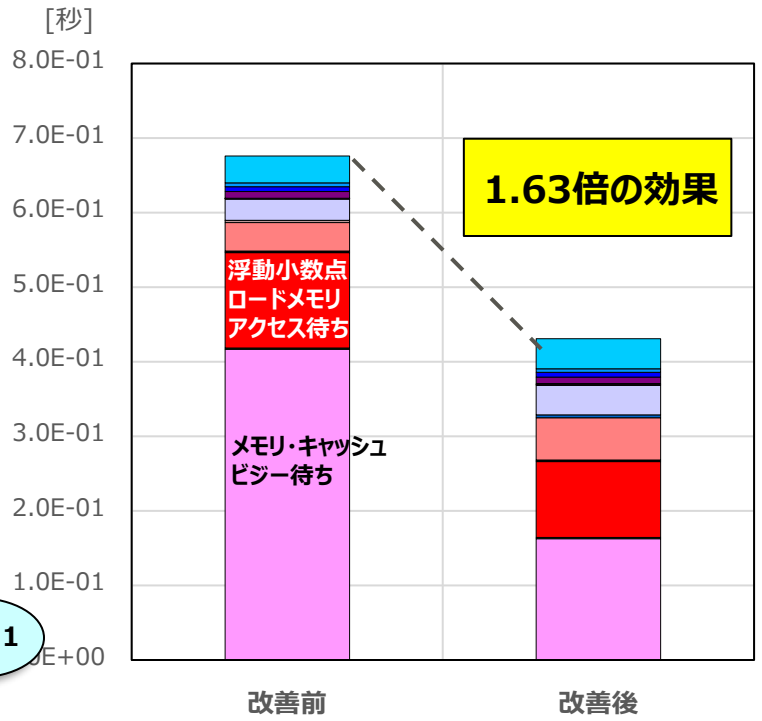
```
37          blki=4*1024/8;
38
39        #pragma omp parallel for reduction(+:s
40  p       for(j=0;j<m;j++) {
          <<< Loop-information Start >>>
          <<<  [OPTIMIZATION]
          <<<    PREFETCH(HARD) Expected by compiler :
          <<<      (unknown)
          <<< Loop-information  End >>>
41  p         for(ii=0;ii<n;ii+=blki) {
42  p           int min1=MIN(ii+blki,n);
          <<< Loop-information Start >>>
          <<<  [OPTIMIZATION]
          <<<    SIMD(VL: 8)
          <<<    PREFETCH(HARD) Expected by compiler :
          <<<      (unknown)
          <<< Loop-information  End >>>
43  p   8v          for(i=ii;i<min1;i++) {
44  p   8v            s1 = s1 + a[j][i] * (s3 * b[j][i] + c[j][i] * (s2 + s3 * d[j][i]));
45  p   8v          }
46  p             int min2=MIN(ii+blki,n-100);
          <<< Loop-information Start >>>
          <<<  [OPTIMIZATION]
          <<<    SIMD(VL: 8)
          <<<    SOFTWARE PIPELINING(IPC: 2.3
          <<<    PREFETCH(HARD) Expected by co
          <<<      (unknown)
          <<< Loop-information  End >>>
47  p   2v          for(i=ii;i<min2;i++) {
48  p   2v            e[j][i] = s2 * (a[j][i] + b[j][i] * (s3 + c[j][i] * d[j][i]));
49  p   2v          }
50  p             }
51  p           }
52            }
```

ブロックサイズ ： 4KB
4KB×4ストリーム=16KB
→ L1キャッシュに載るサイズ

ループ１

配列アクセスがキャッシュヒットする

ループ２



Cache	L1I miss rate (/Effective instruction)	Load-store instruction	L1D miss	L1D miss rate (/Load-store instruction)	L1D miss demand rate (%) (/L1D miss)
改善前	0.00	2.17E+09	4.23E+08	0.19	9.14%
改善後	0.00	1.99E+09	2.35E+08	0.12	19.90%

L2 miss	L2 miss rate (/Load-store instruction)	L2 miss demand rate (%) (/L2 miss)
4.23E+08	0.19	5.42%
2.35E+08	0.12	7.77%

L1DミスとL2ミスが減少した

FUJITSU

ループブロッキング

- ループブロッキングとは
- ループブロッキング（改善前）
- ループブロッキングの効果（ソースチューニング）
- ハードウェアプリフェッチの副作用

ループブロッキングとは (1/3)

FUJITSU

ループブロッキングとは、キャッシュ効率を向上させるために、ソースを指定したブロッキングサイズに分割して実行することです。

2次元またはそれ以上の次元におけるストリップマイニングと見なすことができます。

改善前ソース例

```
subroutine sub(a,b,m,n)
  integer n,m
  real*8 a(m,n),b(n,m)
  do j=1,m
    do i=1,n
      b(i,j)=a(j,i)
    enddo
  enddo
end subroutine
```

配列a：ストライドアクセス
配列b：連続アクセス

改善後ソース例

```
subroutine sub(a,b,m,n)
  parameter(blki=96,blkj=16)
  integer n,m
  real*8 a(m,n),b(n,m)
  do jj=1,m,blkj
    do ii=1,n,blki
      do j=jj,min(jj+blkj-1,m)
        do i=ii,min(ii+blki-1,n)
          b(i,j)=a(j,i)
        enddo
      enddo
    enddo
  enddo
end subroutine
```

ブロックサイズ
1配列12Kbyte
(=96×16×8byte)

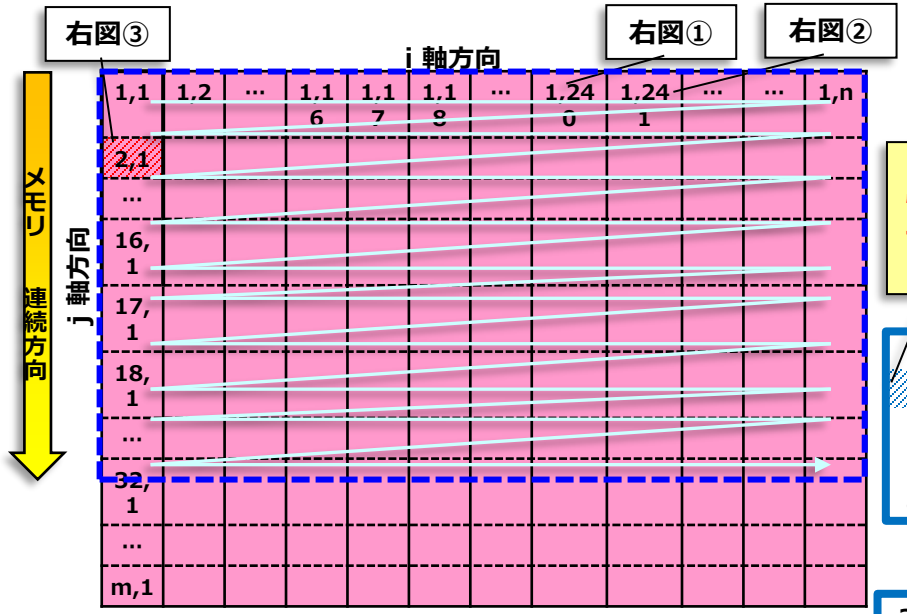
ループブロッキングとは (2/3)

FUJITSU

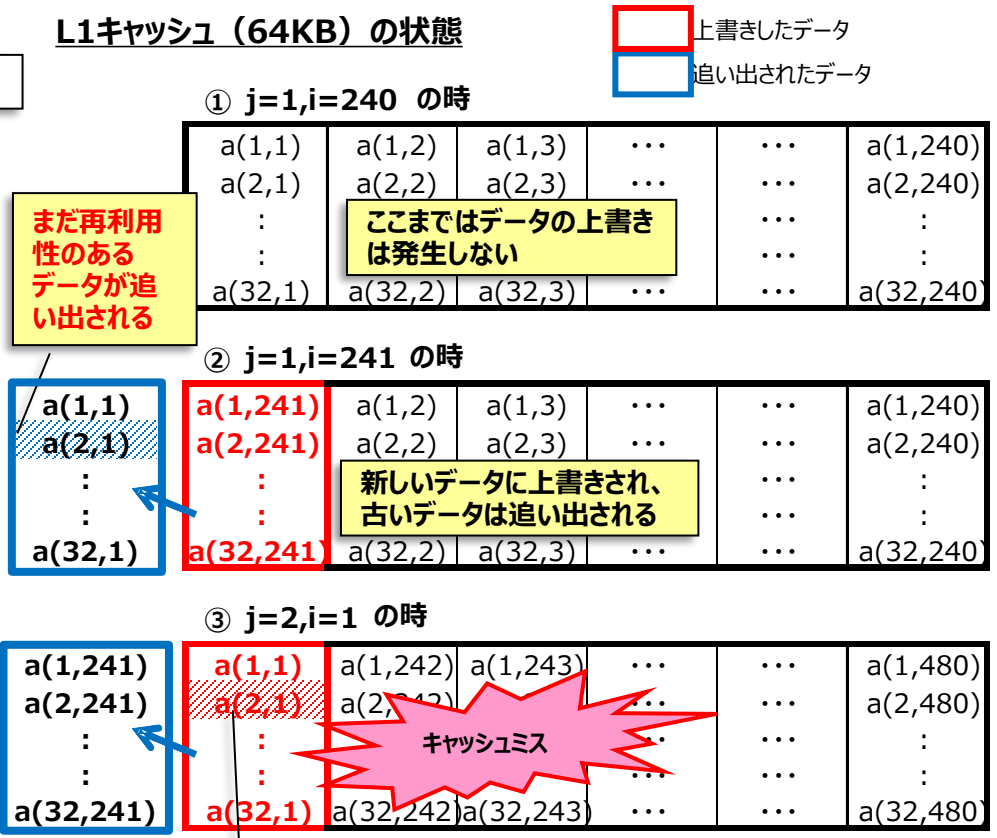
● 配列アクセス（改善前）

配列a はストライドアクセスのため、i が更新されるたびにメモリアクセスが発生する。
その結果、a(1,1) でキャッシュに載せたデータは、a(2,1) でアクセスされる前に追い出されてしまう。

配列a(m,n)のアクセス順序とキャッシュミスの状況



L1キャッシュ（64KB）の状態



ソース例（改善前）

```
subroutine sub(a,b,m,n)
  integer n,m
  real*8 a(m,n), b(n,m)
  do j=1,m
    do i=1,n
      b(i,j)=a(j,i)
    enddo
  enddo
end subroutine
```

配列のアクセス順序
ブロックサイズ
キャッシュライン
キャッシュミス
キャッシュヒット

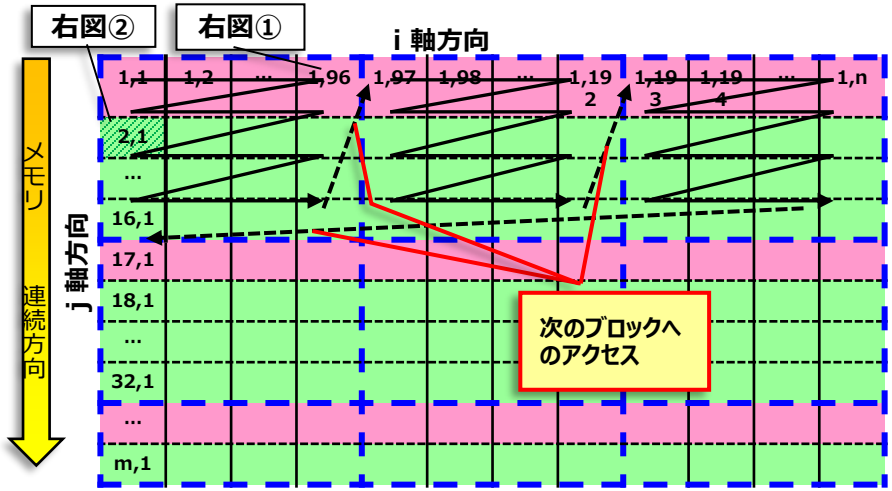
ループブロッキングとは (3/3)

FUJITSU

- **配列アクセス（改善後）　ブロックサイズ96×16**

ループブロッキングにより１ブロックずつのアクセスを行う。
その結果、a(2,1) のアクセスがキャッシュにヒットするようになり、キャッシュ効率が向上する。

配列a(m,n)のアクセス順序とキャッシュミスの状況



L1キャッシュ（64KB）の状態

① j=1,i=96 の時

a(1,1)	a(1,2)	a(1,3)	・・・	a(1,96)		
a(2,1)	a(2,2)	a(2,3)	・・・	a(2,96)		
:	配列データがキャッシュに載る		・・・	:		
:			・・・	:		
a(32,1)	a(32,2)	a(32,3)	・・・	a(32,96)		

② j=2,i=1の時

a(1,1)	a(1,2)	a(1,3)	・・・	a(1,96)		
a(2,1)	a(		・・・	a(2,96)		
:	キャッシュヒット		・・・	:		
:			・・・	:		
a(32,1)	a(32,2)	a(32,3)	・・・	a(32,96)		

データがキャッシュに残っているためキャッシュヒット

ソース例（改善後）

```
subroutine sub(a,b,m,n)
  parameter(blki=96,blkj=16)
  integer n,m
  real*8 a(m,n),b(n,m)
  do jj=1,m, blkj
    do ii=1,n, blki
      do j=jj,min(jj+blkj-1,m)
        do i=ii,min(ii+blki-1,n)
          b(i,j)=a(j,i)
        enddo
      enddo
    enddo
  enddo
end subroutine
```

配列のアクセス順序
ブロックサイズ
キャッシュライン
キャッシュミス
キャッシュヒット

ポイント：
ループブロッキングを行うことで、データの連続性が損なわれハードウェアプリフェッチが効かなくなることがあります。
その場合はソフトウェアプリフェッチを利用します。
詳しくは[ハードウェアプリフェッチの副作用]を参照してください。

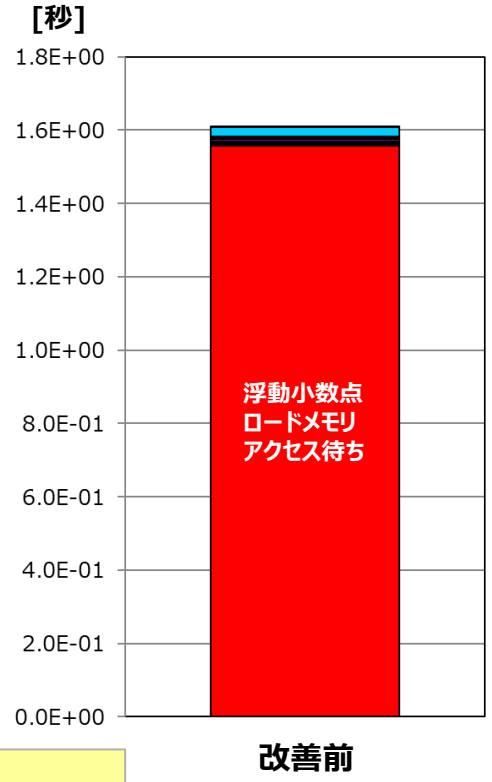
Fortran ループブロッキング（改善前）

FUJITSU

配列aがストライドアクセスであるため、キャッシュの利用効率が悪くなり、浮動小数点ロードメモリアクセス待ちが多くなっています。

改善前ソース

```
48    1           !$omp do
                 <<< Loop-information Start >>>
                 <<<  [OPTIMIZATION]
                 <<<    PREFETCH(HARD) Expected by compiler :
                 <<<      b
                 <<< Loop-information  End >>>
49    2  p            do j=1,n2
                 <<< Loop-information Start >>>
                 <<<  [OPTIMIZATION]
                 <<<    SIMD(VL: 8)
                 <<<    SOFTWARE PIPELINING(IPC: 1.72, ITR: 176, MVE: 6, POL: S)
                 <<<    PREFETCH(HARD) Expected by compiler :
                 <<<      b
                 <<< Loop-information  End >>>
50    3  p  2v          do i=1,n1
51    3  p  2v            b(i,j) = c0 + a(j,i)*(c1 + a(j,i)*(c2 + a(j,i)*(c3 + a(j,i)*
52    3               &         (c4 + a(j,i)*(c5 + a(j,i)*(c6 + a(j,i)*(c7 + a(j,i)*
53    3               &         (c8 + a(j,i)*c9))))))))
54    3  p  2v          enddo
55    2  p            enddo
56    1           !$omp enddo
```



配列a のデータは、i のイタレーション時に一度 L1Dキャッシュに載るが、次の j のイタレーション時にはデータが追い出されてしまっているため、キャッシュミスとなる

Cache

	L1I miss rate (/Effective instruction)	Load-store instruction	L1D miss	L1D miss rate (/Load-store instruction)	L1D miss demand rate (%) (/L1D miss)	L1D miss hardware prefetch rate (%) (/L1D miss)	L1D miss software prefetch rate (%) (/L1D miss)	L2 miss	L2 miss rate (/Load-store instruction)	L2 miss demand rate (%) (/L2 miss)	L2 miss hardware prefetch rate (%) (/L2 miss)	L2 miss software prefetch rate (%) (/L2 miss)
改善前	0.00	4.06E+08	1.28E+09	3.16	98.85%	1.15%	0.00%	1.31E+09	3.24	29.75%	71.85%	0.00%

Fortran ループブロッキングの効果（ソースチューニング）

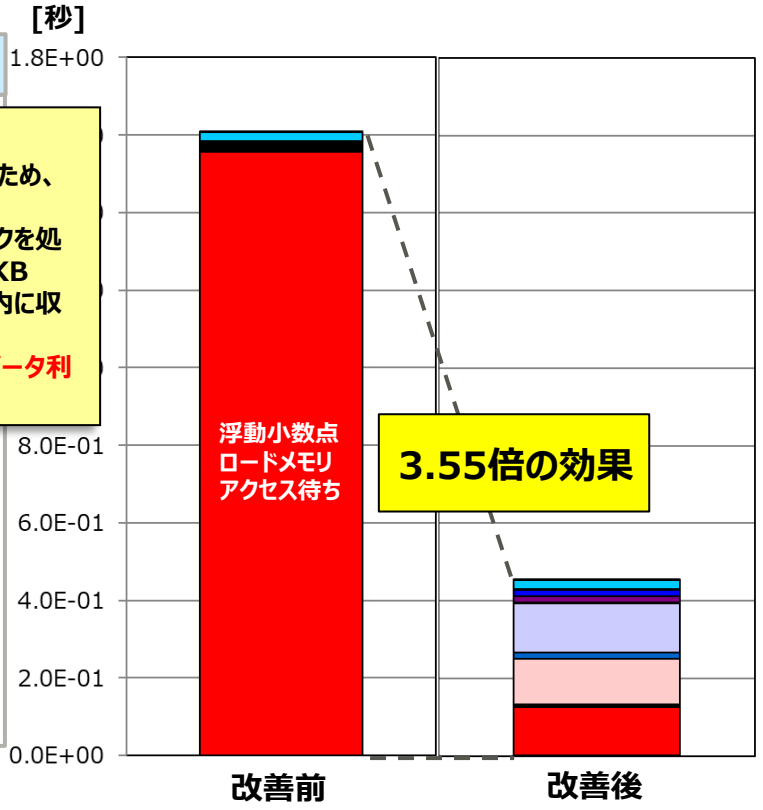
FUJITSU

ループブロッキングにより配列aのデータが再利用されることで、キャッシュ効率が向上し、浮動小数点ロードメモリアクセス待ちが改善されました。

改善後ソース（ソースチューニング）

```
55    1          !$omp do
56    2  p         do jj=1,n2,16
57    3  p          do ii=1,n1,96
58    4  p           do j=jj,min(jj+16-1,n2)
                <<< Loop-information Start >>>
                <<<  [OPTIMIZATION]
                <<<    SIMD(VL: 8)
                <<<    SOFTWARE PIPELINING(IPC: 1.7
                <<< Loop-information  End >>>
59    5  p  2v            do i=ii,min(ii+96-1,n1)
60    5  p  2v             b(i,j) = c0 + a(j,i)*(c1 + a(j,i)*(c2 + a(j,i)*(c3 + a(j,i)*
61    5               &       (c4 + a(j,i)*(c5 + a(j,i)*(c6 + a(j,i)*(c7 + a(j,i)*
62    5               &       (c8 + a(j,i)*c9))))))))
63    5  p  2v            enddo
64    4  p               enddo
65    3  p              enddo
66    2  p             enddo
67    1          !$omp enddo
```

ループブロッキングの適用
L1キャッシュのサイズは64KBのため、１ブロックのサイズを12KB(96×16×8)、１ブロックを処理するために必要なサイズを24KB（12×2個分）とし、キャッシュ内に収めています。
L1DキャッシュとL2キャッシュのデータ利用効率向上が狙い。



L1DミスとL2ミスが大幅に減少した

Cache

	L1I miss rate (/Effective instruction)	Load-store instruction	L1D miss	L1D miss rate (/Load-store instruction)	L1D miss demand rate (%) (/L1D miss)	L1D miss hardware prefetch rate (%) (/L1D miss)	L1D miss software prefetch rate (%) (/L1D miss)	L2 miss	L2 miss rate (/Load-store instruction)	L2 miss demand rate (%) (/L2 miss)	L2 miss hardware prefetch rate (%) (/L2 miss)	L2 miss software prefetch rate (%) (/L2 miss)
改善前	0.00	4.06E+08	1.28E+09	3.16	98.85%	1.15%	0.00%	1.31E+09	3.24	29.75%	71.85%	0.00%
改善後	0.00	8.26E+08	1.69E+08	0.20	95.18%	4.82%	0.00%	1.56E+08	0.19	45.06%	61.56%	0.00%

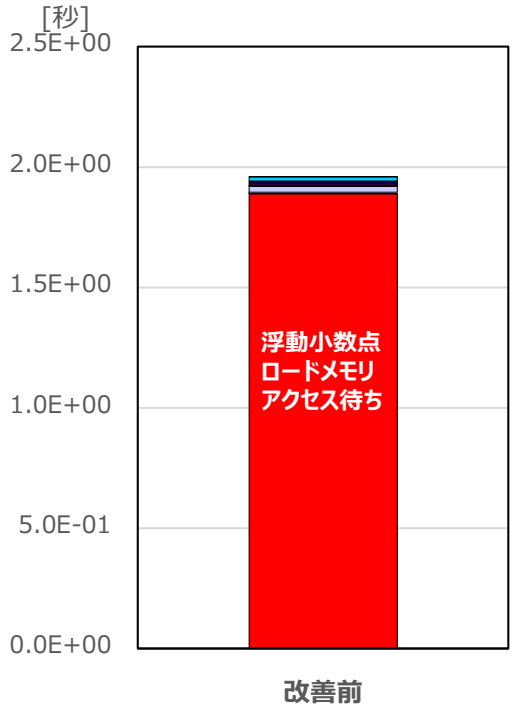
C/C++ ループブロッキング（改善前）

FUJITSU

配列aがストライドアクセスであるため、キャッシュの利用効率が悪くなり、浮動小数点ロードメモリアクセス待ちが多くなっています。

改善前ソース

```
43          #pragma omp parallel
44           {
45             for(k=0;k<iter;k++) {
46          #pragma omp for
           <<< Loop-information Start >>>
           <<<  [OPTIMIZATION]
           <<<    PREFETCH(HARD) Expected by compiler :
           <<<      (unknown)
           <<< Loop-information  End >>>
47   p           for(j=0;j<n2;j++) {
           <<< Loop-information Start >>>
           <<<  [OPTIMIZATION]
           <<<    SIMD(VL: 8)
           <<<    SOFTWARE PIPELINING(IPC: 1.08, ITR: 128, MVE: 4, POL: S)
           <<<    PREFETCH(HARD) Expected by compiler :
           <<<      (unknown)
           <<< Loop-information  End >>>
48   p    2v         for(i=0;i<n1;i++) {
49   p    2v           b[j][i] = c0 + a[i][j]*(c1 + a[i][j]*(c2 + a[i][j]*(c3 + a[i][j]*
50                         (c4 + a[i][j]*(c5 + a[i][j]*(c6 + a[i][j]*(c7 + a[i][j]*
51                         (c8 + a[i][j]*c9)))))));
52   p    2v         }
53   p             }
54                }
55              }
56            }
```



配列a のデータは、i のイタレーション時に一度 L1Dキャッシュに載るが、次の j のイタレーション時にはデータが追い出されてしまっているため、キャッシュミスとなる

Cache	L1I miss rate (/Effective instruction)	Load-store instruction	L1D miss	L1D miss rate (/Load-store instruction)	L1D miss demand rate (%) (/L1D miss)	L1D miss hardware prefetch rate (%) (/L1D miss)	L1D miss software prefetch rate (%) (/L1D miss)	L2 miss	L2 miss rate (/Load-store instruction)	L2 miss demand rate (%) (/L2 miss)	L2 miss hardware prefetch rate (%) (/L2 miss)	L2 miss software prefetch rate (%) (/L2 miss)
改善前	0.00	4.00E+08	1.28E+09	3.21	98.53%	1.47%	0.00%	1.33E+09	3.32	28.24%	73.08%	0.00%

C/C++ ループブロッキングの効果（ソースチューニング）

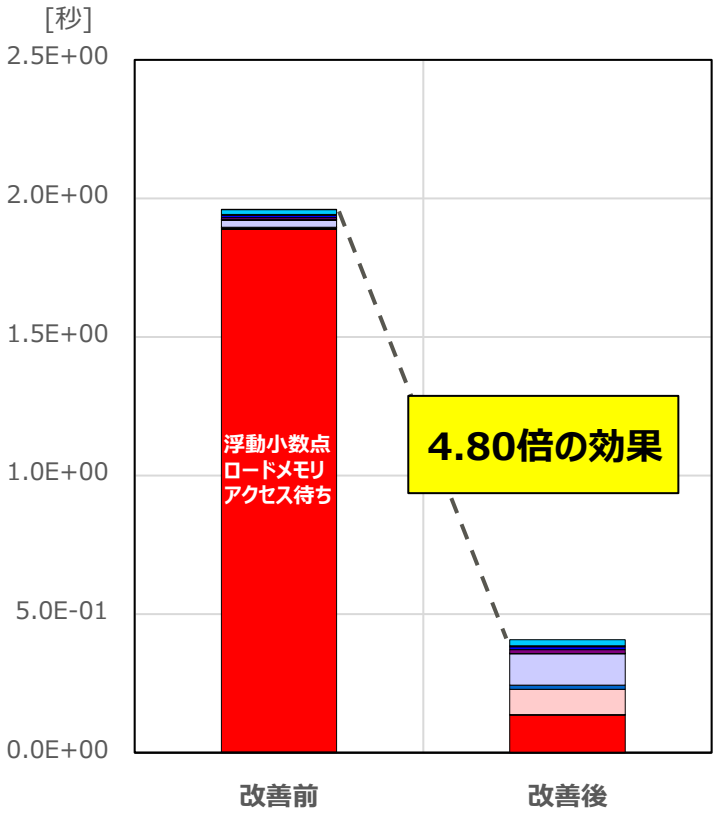
FUJITSU

ループブロッキングにより配列aのデータが再利用されることで、キャッシュ効率が向上し、浮動小数点ロードメモリアクセス待ちが改善されました。

改善後ソース（ソースチューニング）

```
45          #pragma omp parallel
46           {
47             for(k=0;k<iter;k++) {
48          #pragma omp for
49  p            for(jj=0;jj<n2;jj=jj+16) {
50  p              for(ii=0;ii<n1;ii=ii+96) {
            <<< Loop-information Start >>>
            <<<  [OPTIMIZATION]
            <<<    PREFETCH(HARD) Expected by compiler :
            <<<      b
            <<< Loop-information  End >>>
51  p                for(j=jj;j<MIN(jj+16-1,n2);j++) {
52  p                  int st = MIN(ii+96-1,n1);
            <<< Loop-information Start >>>
            <<<  [OPTIMIZATION]
            <<<    SIMD(VL: 8)
            <<<    SOFTWARE PIPELINING(IPC: 1.25, ITR: 128, MVE: 4, POL: S)
            <<<    PREFETCH(HARD) Expected by compiler :
            <<<      b
            <<< Loop-information  End >>>
53  p    2v              for(i=ii;i<st;i++) {
54  p    2v                b[j][i] = c0 + a[i][j]*(c1 + a[i][j]*(c2 + a[i][j]*(c3 + a[i][j]*
55                             (c4 + a[i][j]*(c5 + a[i][j]*(c6 + a[i][j]*(c7 + a[i][j]*
56                             (c8 + a[i][j]*c9)))))));
57  p    2v              }
58  p                  }
59  p                }
60  p              }
61  p            }
62  p          }
63  p        }
```

ループブロッキングの適用
L1キャッシュのサイズは64KBのため、1ブロックのサイズを12KB(96×16×8)、1ブロックを処理するために必要なサイズを24KB（12×2個分）とし、キャッシュ内に収めています。
L1Dキャッシュ とL2キャッシュのデータ利用効率向上が狙い。



L1DミスとL2ミスが大幅に減少した

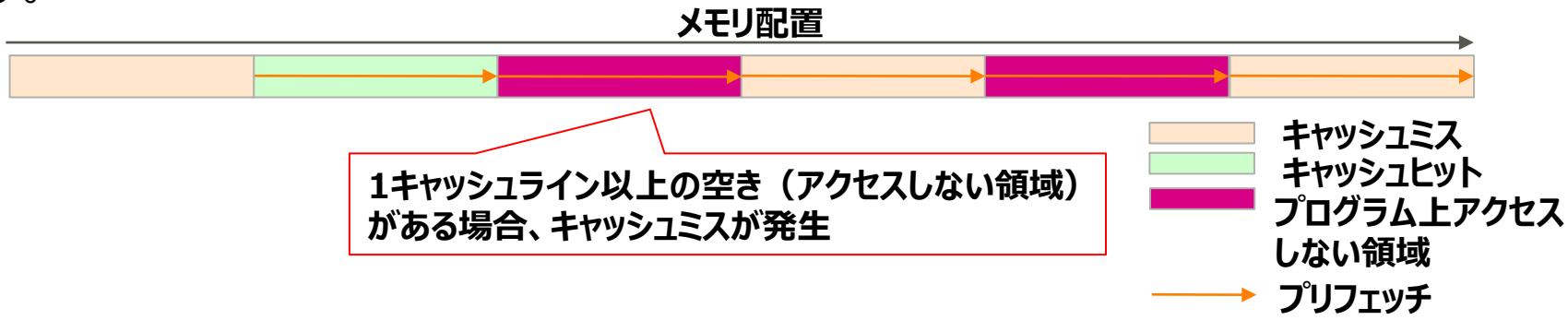
Cache	L1I miss rate (/Effective instruction)	Load-store instruction	L1D miss	L1D miss rate (/Load-store instruction)	L1D miss demand rate (%) (/L1D miss)	L1D miss hardware prefetch rate (%) (/L1D miss)	L1D miss software prefetch rate (%) (/L1D miss)	L2 miss	L2 miss rate (/Load-store instruction)	L2 miss demand rate (%) (/L2 miss)	L2 miss hardware prefetch rate (%) (/L2 miss)	L2 miss software prefetch rate (%) (/L2 miss)
改善前	0.00	4.00E+08	1.28E+09	3.21	98.53%	1.47%	0.00%	1.33E+09	3.32	28.24%	73.08%	0.00%
改善後	0.00	4.93E+08	1.70E+08	0.35	93.66%	6.34%	0.00%	1.40E+08	0.28	49.47%	53.12%	0.00%

ハードウェアプリフェッチの副作用

- ループブロッキング、外側ループプリフェッチなどでは、ブロックサイズが小さくなることでデータが連続的でなくなり、ハードウェアプリフェッチが有効にならない場合があります。その場合はソフトウェアプリフェッチを利用する必要があります。
- ソフトウェアプリフェッチの指定方法はソフトウェアプリフェッチの利用を参照してください。

- **ハードウェアプリフェッチ**

プログラムのメモリアクセスの規則性からハードウェアがデータアクセスを予測しプリフェッチします。
データ間に1キャッシュライン以上の空きがある場合はプリフェッチが失敗する可能性があります。



- **ソフトウェアプリフェッチ**

ソフトウェア（コンパイラ）がプログラムを解析し、prefetch命令を生成することでプリフェッチします。または、指示行指定で該当箇所にプリフェッチ命令を生成します。

FUJITSU

セクタキャッシュ

- セクタキャッシュとは
- セクタキャッシュの使用方法
- セクタキャッシュ：事例１（改善前）
- セクタキャッシュ：事例１（ソースチューニング）
- セクタキャッシュ：事例２（改善前）
- セクタキャッシュ：事例２（ソースチューニング）

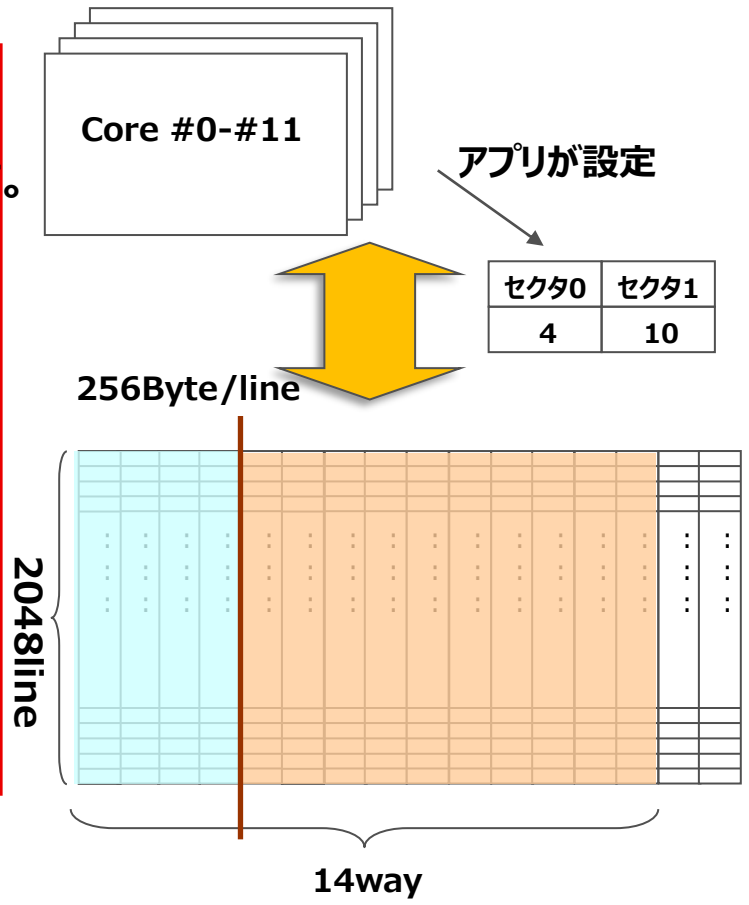
セクタキャッシュとは

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す。アプリが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ます。（再利用する配列はセクタ1を使用し、その他はセクタ0を使用）

■ セクタキャッシュの詳細

- L1Dキャッシュ・L2キャッシュいずれにも複数個のセクタを設定できます。セクタの最大数はL1Dが4(※1),L2は2です。
- 各セクタの容量はWAY数で指定します。
- 容量は目標値として働きます。
  ハードは、ラインリプレース時に各セクタが指定された容量に近づくように制御します。
  →容量オーバーしていても強制的に無効化しない
- セクタ内の追い出しはLRU（最近最も使われていないデータを最初に捨てる）アルゴリズムで制御します。
- セクタ０、１の用途はアプリケーションで決めることが可能です。
  ただし、命令列はセクタ０に格納されます。
- 2次キャッシュはアシスタントコアが常時2wayを使用します。

L2キャッシュでの利用イメージ



(※1)現在仕様ではL1Dキャッシュのセクタ数は2です。お客様が簡単に使用できることを優先しこの仕様としました。

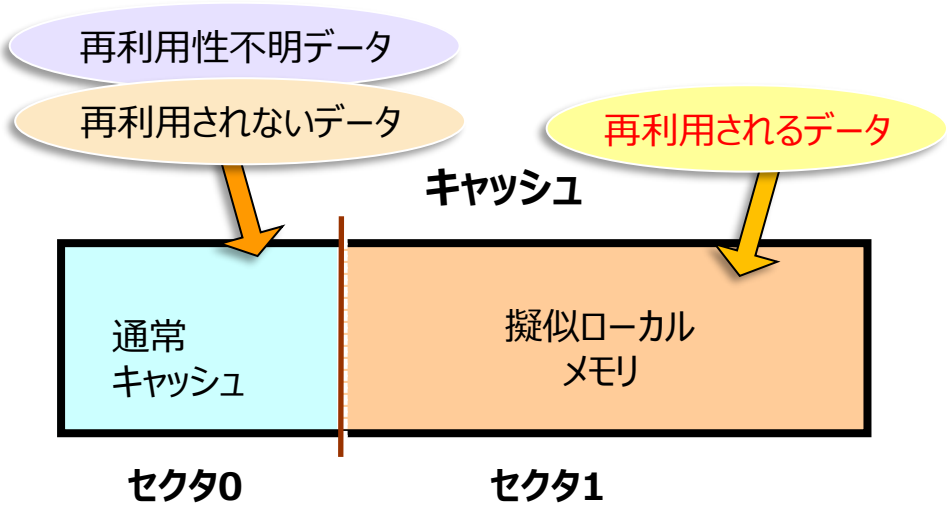
セクタキャッシュの使用方法（1/2）

FUJITSU

■ **セクターキャッシュ：疑似ローカルメモリ**

ソフトウェアが、データの再利用性に応じて
セクタを使い分けることが可能

- **再利用するデータ → セクタ1を使用**
- **その他のデータ → セクタ0を使用**
- **セクタ1上のデータは、他のデータによって追い出されることがない**
- **指示行でセクタ1に載せる配列を指定できる**



セクタキャッシュ指定のコンパイラ指示行の使用例

```
!OCL SCACHE_ISOLATE_WAY(L2=10)
!OCL SCACHE_ISOLATE_ASSIGN(a)
do j=1,m
   do i=1,n
      a(i) = a(i) + b(i,j)*c(i,j)
   enddo
enddo
!OCL END_SCACHE_ISOLATE_ASSIGN
!OCL END_SCACHE_ISOLATE_WAY
```

旧仕様の指定方法（京,FX100）

```
!OCL CACHE_SECTOR_SIZE(4,10)
!OCL CACHE_SUBSECTOR_ASSIGN(a)
do j=1,m
   do i=1,n
      a(i) = a(i) + b(i,j)*c(i,j)
   enddo
enddo
!OCL END_CACHE_SUBSECTOR
!OCL END_CACHE_SECTOR_SIZE
```

<狙い>
ループ中で配列 b と配列 c のアクセスによって
再利用性のある配列 a がキャッシュから追い出されないようにする

セクタキャッシュの使用方法（2/2）

FUJITSU

セクタキャッシュを使用する場合は、以下の最適化制御行を指定します。

最適化指示子 (Fortran)	意味	指定可能な最適化制御行			
		プログラム単位	DOループ単位	文単位	配列代入文単位
SCACHE_ISOLATE_WAY(L2=n1[,L1=n2]) END_SCACHE_ISOLATE_WAY	1次キャッシュと2次キャッシュのセクタ1の最大way数を指示します。	○	×	○	×
SCACHE_ISOLATE_ASSIGN(array1[,array2]…) END_SCACHE_ISOLATE_ASSIGN	キャッシュのセクタ1に載せる配列を指示します。	○	×	○	×

最適化指示子 (C/C++)	意味	指定可能な最適化制御行			
		global行	procedure行	loop行	statement行
scache_isolate_way(L2=n1[,L1=n2]) end_scache_isolate_way	1次キャッシュと2次キャッシュのセクタ1の最大way数を指示します。	×	○	×	○
scache_isolate_assign(array1[,array2]…) end_scache_isolate_assign	キャッシュのセクタ1に載せる配列を指示します。	×	○	×	○

■ 注意事項

- 2次キャッシュはアシスタントコアが常時2wayを使用します。そのため、n1およびn2に指定できる範囲は以下のようになります。
  0 ≦ *n1* ≦ “2次キャッシュの最大way数 - 2”
  0 ≦ *n2* ≦ “1次キャッシュの最大way数”
- アシスタントコアを含むCMGでは、L2キャッシュの一部(2way=1MiB分)をアシスタントコア用に使用します。そのため、アシスタントコアを含むCMGでは、**2次キャッシュの最大way数は14, サイズは7MiB**となります。
- clangモードではセクタキャッシュ機能を使用できません

A64FX諸元	
CMG数	4
L1Iキャッシュ・サイズ	64KiB / 4way
L1Dキャッシュ・サイズ	64KiB / 4way
L2キャッシュ・サイズ	32MiB / 16way (8MiB / CMG)

Fortran セクタキャッシュ：事例1（改善前）

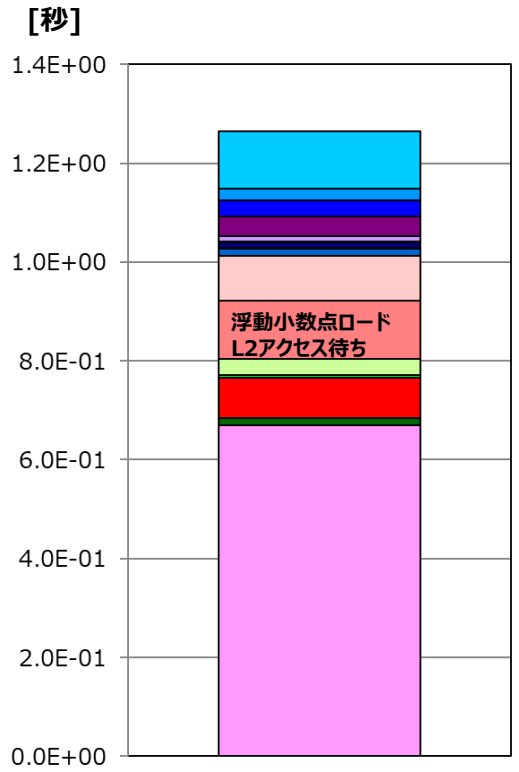
FUJITSU

配列b のデータはキャッシュから追い出されてしまうため、再利用できません。
そのため浮動小数点ロードL2アクセス待ちが多くなっています。

改善前ソース

```
66                    parameter(n=8*1024*1024, m=9*512*1024/8)
67                    real*8    a(n), b(m), s
68                    integer*8 c(n)
69                    real*8    dummy1(140),dummy2(140)
70                    common /data/a,dummy1,c,dummy2,b
71
                  <<< Loop-information Start >>>
                  <<<  [PARALLELIZATION]
                  <<<    Standard iteration count: 843
                  <<<  [OPTIMIZATION]
                  <<<    SIMD(VL: 8)
                  <<<    SOFTWARE PIPELINING(IPC: 2.66, ITR: 176, MVE: 4, POL: S)
                  <<<    PREFETCH(HARD) Expected by compiler :
                  <<<      c, a
                  <<< Loop-information  End >>>
72     1  pp   2v          do i=1,n
73     1   p   2v            a(i) = a(i) + s * b(c(i))
74     1   p   2v          enddo
```

配列サイズ
a: 64MiB
b: 4.5MiB
c: 64MiB



Cache	L1I miss rate (/Effective instruction)	Load-store instruction	L1D miss	L1D miss rate (/Load-store instruction)	L1D miss demand rate (%) (/L1D miss)	L1D miss hardware prefetch rate (%) (/L1D miss)	L1D miss software prefetch rate (%) (/L1D miss)	L2 miss	L2 miss rate (/Load-store instruction)	L2 miss demand rate (%) (/L2 miss)	L2 miss hardware prefetch rate (%) (/L2 miss)	L2 miss software prefetch rate (%) (/L2 miss)
改善前	0.00	4.76E+09	7.89E+08	0.17	0.89%	99.11%	0.00%	7.34E+08	0.15	0.77%	100.00%	0.00%

	Memory throughput (GB/s)
改善前	203.11

メモリアクセス負荷が高い

L2キャッシュミス率が高い

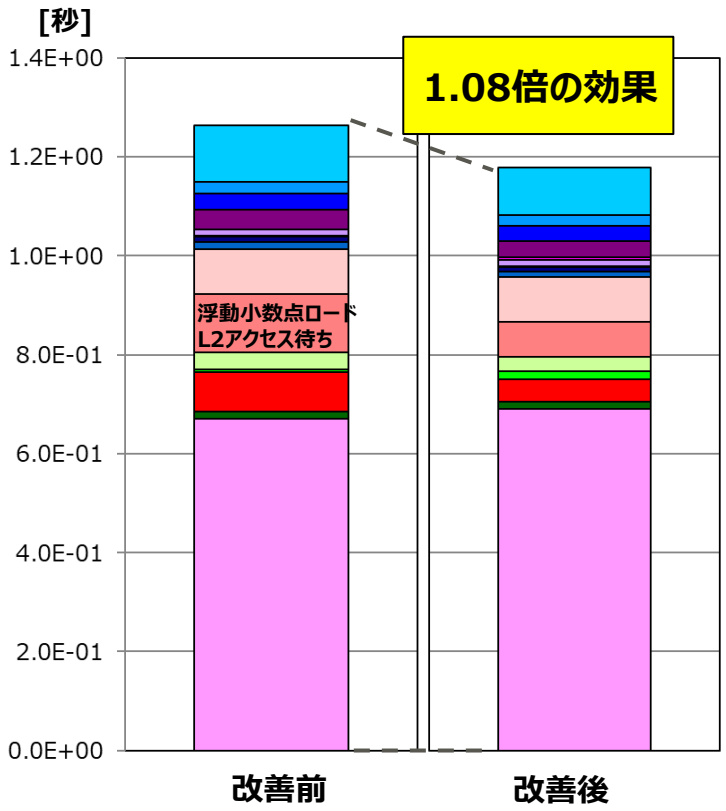
Fortran セクタキャッシュ：事例1（ソースチューニング）

FUJITSU

配列bをセクタ１に載せることでキャッシュ効率が向上し、浮動小数点ロードL2アクセス待ちが改善されました。

改善後ソース（最適化制御行チューニング）

```
58                  parameter(n=8*1024*1024, m=9*512*1024/8)
59                   real*8    a(n), b(m), s
60                   integer*8 c(n)
61                   real*8    dummy1(140),dummy2(140)
62                   common /data/a,dummy1,c,dummy2,b
63
64                !OCL SCACHE_ISOLATE_WAY(L2=10)
65                !OCL SCACHE_ISOLATE_ASSIGN(b)
                 <<< Loop-information Start >>>
                 <<<  [PARALLELIZATION]
                 <<<    Standard iteration count: 843
                 <<<  [OPTIMIZATION]
                 <<<    SIMD(VL: 8)
                 <<<    SOFTWARE PIPELINING(IPC: 2.66, ITR: 176, MVE: 4, POL: S)
                 <<<    PREFETCH(HARD) Expected by compiler :
                 <<<      c, a
                 <<< Loop-information  End >>>
66    1  pp   2v          do i=1,n
67    1   p   2v            a(i) = a(i) + s * b(c(i))
68    1   p   2v          enddo
69                !OCL END_SCACHE_ISOLATE_ASSIGN
70                !OCL END_SCACHE_ISOLATE_WAY
```



	L1I miss rate (/Effective instruction)	Load-store instruction	L1D miss	L1D miss rate (/Load-store instruction)	L1D miss demand rate (%) (/L1D miss)	L1D miss hardware prefetch rate (%) (/L1D miss)	L1D miss software prefetch rate (%) (/L1D miss)	L2 miss	L2 miss rate (/Load-store instruction)	L2 miss demand rate (%) (/L2 miss)	L2 miss hardware prefetch rate (%) (/L2 miss)	L2 miss software prefetch rate (%) (/L2 miss)
改善前	0.00	4.76E+09	7.89E+08	0.17	0.89%	99.11%	0.00%	7.34E+08	0.15	0.77%	100.00%	0.00%
改善後	0.00	5.19E+09	7.93E+08	0.15	1.19%	98.81%	0.01%	5.99E+08	0.12	1.93%	99.69%	0.00%

	Memory throughput (GB/s)
改善前	203.11
改善後	188.07

L2ミスが減少した

C/C++ セクタキャッシュ：事例1（改善前）

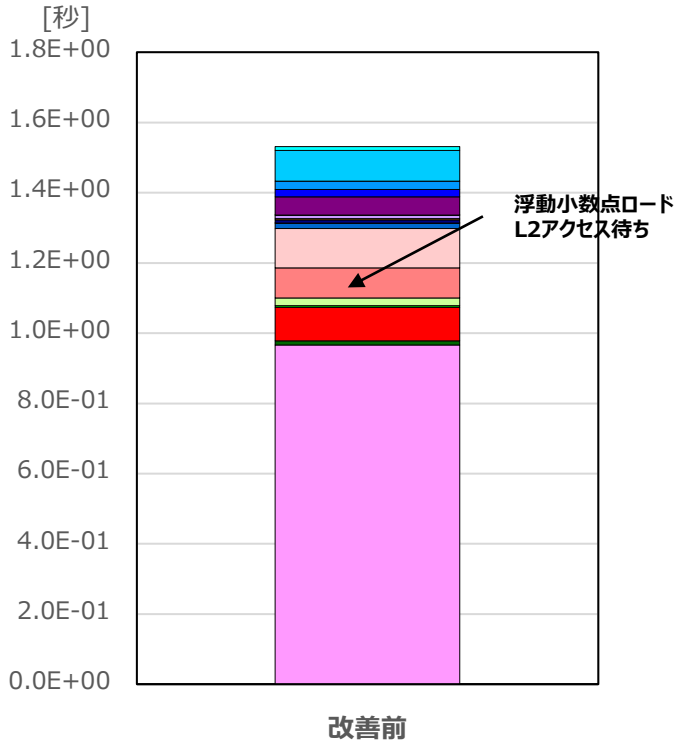
FUJITSU

配列b のデータはキャッシュから追い出されてしまうため、再利用できません。
そのため浮動小数点ロードL2アクセス待ちが多くなっています。

改善前ソース

```
56          void sub(double s) {
57            long long int i;
58
59          #pragma omp parallel for
            <<< Loop-information Start >>>
            <<<  [OPTIMIZATION]
            <<<    SIMD(VL: 8)
            <<<    SOFTWARE PIPELINING(IPC: 2.33, ITR: 160,
                                MVE: 4, POL: S)
            <<<    PREFETCH(HARD) Expected by compiler :
            <<<      c, a
            <<< Loop-information  End >>>
60  p     2v    for(i=0;i<n;i++) {
61  p     2v      a[i] = a[i] + s * b[c[i]];
62  p     2v    }
63          }
```

配列宣言：サイズ
double a[8388608]: 64MiB
double b[589824]: 4.5MiB
long long int c[8388608]: 64MiB



Cache	L1I miss rate (/Effective instruction)	Load-store instruction	L1D miss	L1D miss rate (/Load-store instruction)	L1D miss demand rate (%) (/L1D miss)	L1D miss hardware prefetch rate (%) (/L1D miss)	L1D miss software prefetch rate (%) (/L1D miss)	L2 miss	L2 miss rate (/Load-store instruction)	L2 miss demand rate (%) (/L2 miss)	L2 miss hardware prefetch rate (%) (/L2 miss)	L2 miss software prefetch rate (%) (/L2 miss)
改善前	0.00	4.85E+09	7.89E+08	0.16	1.07%	98.92%	0.01%	7.39E+08	0.15	0.78%	100.00%	0.00%

Statistics	Memory throughput (GB/s)
改善前	167.47

メモリアクセス負荷が高い

L2キャッシュミス率が高い

C/C++ セクタキャッシュ：事例1（ソースチューニング）

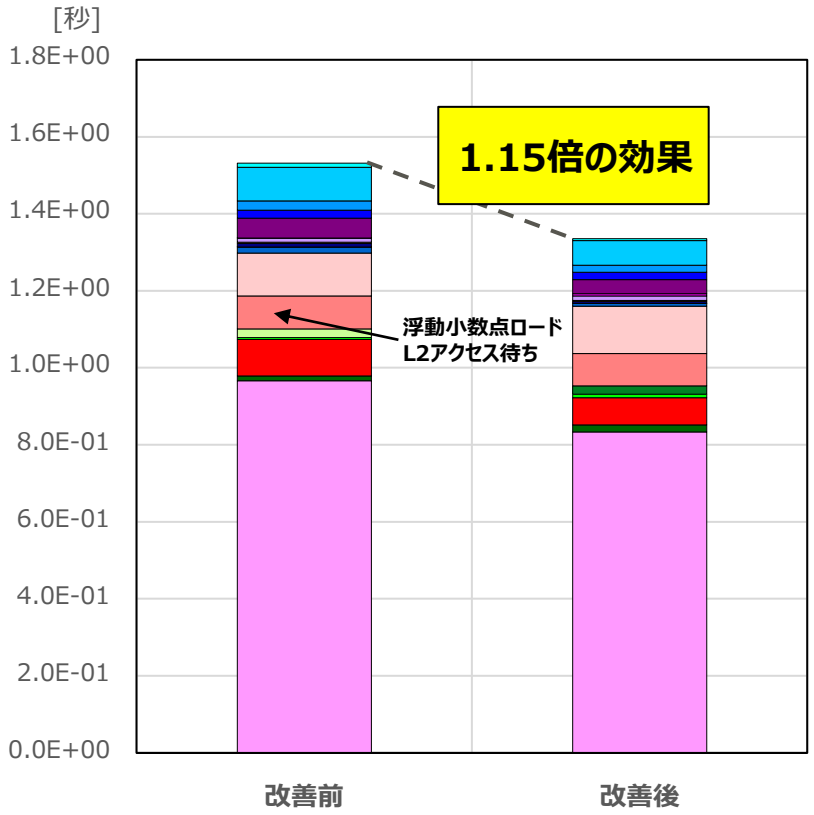
FUJITSU

配列bをセクタ１に載せることでキャッシュ効率が向上し、浮動小数点ロードL2アクセス待ちが改善されました。

改善後ソース（最適化制御行チューニング）

配列宣言：サイズ
double a[8388608]: 64MiB
double b[589824]: 4.5MiB
long long int c[8388608]: 64MiB

```
56          void sub(double s){
57            long long int i;
58          #pragma statement scache_isolate_way L2=10
59          #pragma statement scache_isolate_assign b
60          #pragma omp parallel for
            <<< Loop-information Start >>>
            <<<  [OPTIMIZATION]
            <<<    SIMD(VL: 8)
            <<<    SOFTWARE PIPELINING(IPC: 2.33, ITR: 160,
                                       MVE: 4, POL: S)
            <<<    PREFETCH(HARD) Expected by compiler :
            <<<      c, a
            <<< Loop-information  End >>>
61   p    2v    for(i=0;i<n;i++) {
62   p    2v      a[i] = a[i] + s * b[c[i]];
63   p    2v    }
64          #pragma statement end_scache_isolate_assign
65          #pragma statement end_scache_isolate_way
66          }
```



Cache	L1I miss rate (/Effective instruction)	Load-store instruction	L1D miss	L1D miss rate (/Load-store instruction)	L1D miss demand rate (%) (/L1D miss)	L1D miss hardware prefetch rate (%) (/L1D miss)	L1D miss software prefetch rate (%) (/L1D miss)	L2 miss	L2 miss rate (/Load-store instruction)	L2 miss demand rate (%) (/L2 miss)	L2 miss hardware prefetch rate (%) (/L2 miss)	L2 miss software prefetch rate (%) (/L2 miss)
改善前	0.00	4.85E+09	7.89E+08	0.16	1.07%	98.92%	0.01%	7.39E+08	0.15	0.78%	100.00%	0.00%
改善後	0.00	5.03E+09	7.91E+08	0.16	1.23%	98.78%	0.00%	5.48E+08	0.11	1.99%	98.64%	0.00%

Statistics	Memory throughput (GB/s)
改善前	167.47
改善後	155.55

L2ミスが減少した

Fortran セクタキャッシュ：事例2（改善前）

FUJITSU

配列uのデータはキャッシュから追い出されてしまうため、再利用できません。
そのため、メモリ・キャッシュビジー待ちが多くなっています。

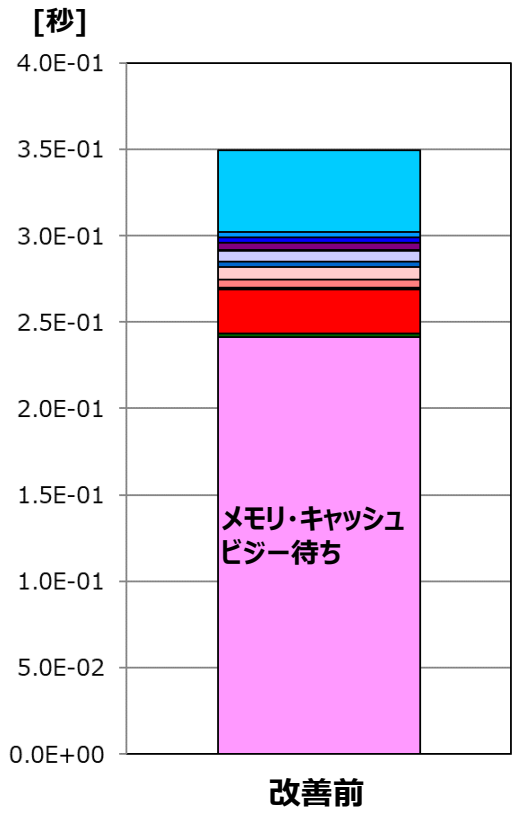
改善前ソース

```
167    1  s          do iter = 1, niter
                <<< Loop-information Start >>>
                <<<  [PARALLELIZATION]
                <<<    Standard iteration cou
                <<< Loop-information  End >
168    2  pp           do k=1,n3-2
                <<< Loop-information Start >
                <<<  [OPTIMIZATION]
                <<<    PREFETCH(HARD) Expected by compiler :
                <<<      u, rhs, unew
                <<< Loop-information  End >>
169    3  p              do j=1,n2-2
                <<< Loop-information Start >>
                <<<  [OPTIMIZATION]
                <<<    SIMD(VL: 8)
                <<<    SOFTWARE PIPELINING(IPC: 3.71, ITR: 136, MVE: 9, POL: S)
                <<<    PREFETCH(HARD) Expected by compiler :
                <<<      u, rhs, unew
                <<< Loop-information  End >>>
170    4  p  v             do i=1,n1-2
171    4  p  v               unew(i,j,k) = &
172    4                       ((u(i+1,j,k) + u(i-1,j,k)) * h1sqinv  &
173    4                       +(u(i,j+1,k) + u(i,j-1,k)) * h2sqinv &
174    4                       +(u(i,j,k+1) + u(i,j,k-1)) * h3sqinv &
175    4                       -rhs(i,j,k)) * hhhinv
176    4  p  v             end do
177    3  p               end do
178    2  p             end do
179    1             end do
```

n1=452
n2=52
n3=322

配列サイズ
unew: 60.5MB
u: 60.5MB
rhs: 60.5MB

配列uのi,j次元に再利用性があるため、配列uをオンキャッシュにしたい。



メモリアクセス負荷が高い

	Memory throughput (GB/s)
改善前	215.62

Cache	L2 miss	L2 miss rate (/Load-store instruction)	L2 miss demand rate (%) (/L2 miss)	L2 miss hardware prefetch rate (%) (/L2 miss)	L2 miss software prefetch rate (%) (/L2 miss)
改善前	2.46E+08	0.15	2.57%	98.19%	0.00%

Fortran セクタキャッシュ：事例2（ソースチューニング）

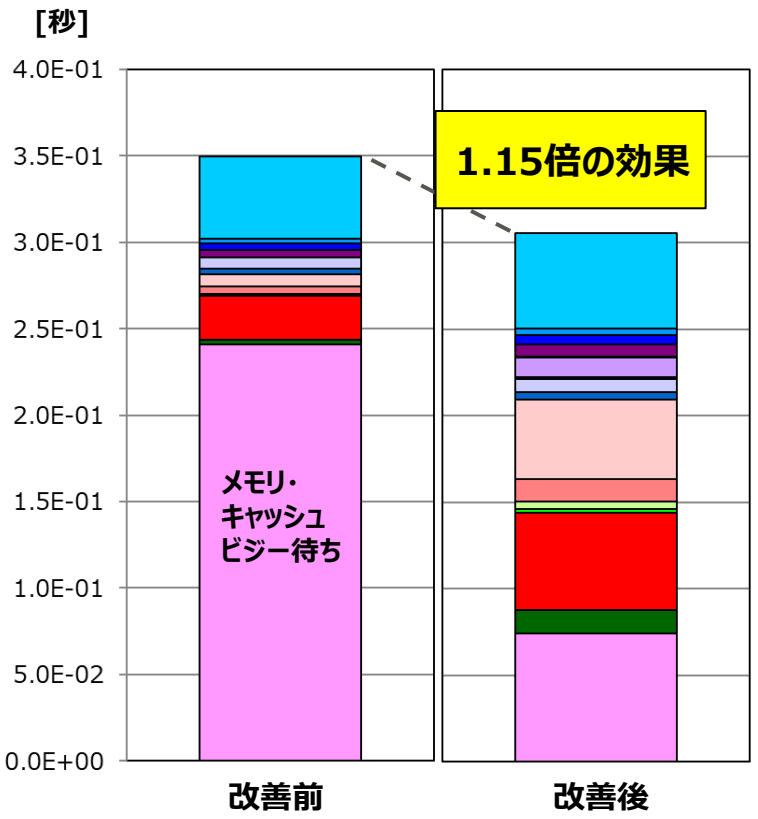
FUJITSU

配列uのk次元の一部をセクタ１に載せることで、キャッシュ効率が向上し、メモリ・キャッシュビジー待ちが改善されました。

改善後ソース

```
166                !OCL SCACHE_ISOLATE_WAY(L2=13)
167                !OCL SCACHE_ISOLATE_ASSIGN(u)
168    1  s            do iter = 1, niter
                  <<< Loop-information Start >>>
                  <<<  [PARALLELIZATION]
                  <<<    Standard iteration count: 2
                  <<< Loop-information  End >>>
169    2  pp             do k=1,n3-2
                  <<< Loop-information Start >>>
                  <<<  [OPTIMIZATION]
                  <<<    PREFETCH(HARD) Expected by compiler :
                  <<<      u, rhs, unew
                  <<< Loop-information  End >>>
170    3  p              do j=1,n2-2
                  <<< Loop-information Start >>>
                  <<<  [OPTIMIZATION]
                  <<<    SIMD(VL: 8)
                  <<<    SOFTWARE PIPELINING(IPC: 3.71, ITR: 136, MVE: 9, POL: S)
                  <<<    PREFETCH(HARD) Expected by compiler :
                  <<<      u, rhs, unew
                  <<< Loop-information  End >>>
171    4  p   v            do i=1,n1-2
172    4  p   v              unew(i,j,k) = &
173    4                       ((u(i+1,j,k) + u(i-1,j,k)) * h1sqinv  &
174    4                       +(u(i,j+1,k) + u(i,j-1,k)) * h2sqinv &
175    4                       +(u(i,j,k+1) + u(i,j,k-1)) * h3sqinv &
176    4                       -rhs(i,j,k)) * hhhinv
177    4  p   v            end do
178    3  p              end do
179    2  p            end do
180    1              end do
181                !OCL END_SCACHE_ISOLATE_ASSIGN
182                !OCL END_SCACHE_ISOLATE_WAY
```

配列uの再利用性が向上



L2ミスが減少

	Memory throughput (GB/s)
改善前	215.62
改善後	205.39

	L2 miss	L2 miss rate (/Load-store instruction)	L2 miss demand rate (%) (/L2 miss)	L2 miss hardware prefetch rate (%) (/L2 miss)	L2 miss software prefetch rate (%) (/L2 miss)
改善前	2.46E+08	0.15	2.57%	98.19%	0.00%
改善後	1.95E+08	0.10	13.43%	87.39%	0.00%

C/C++ セクタキャッシュ：事例2（改善前）

FUJITSU

配列uのデータはキャッシュから追い出されてしまうため、再利用できません。
そのため、メモリ・キャッシュビジー待ちが多くなっています。

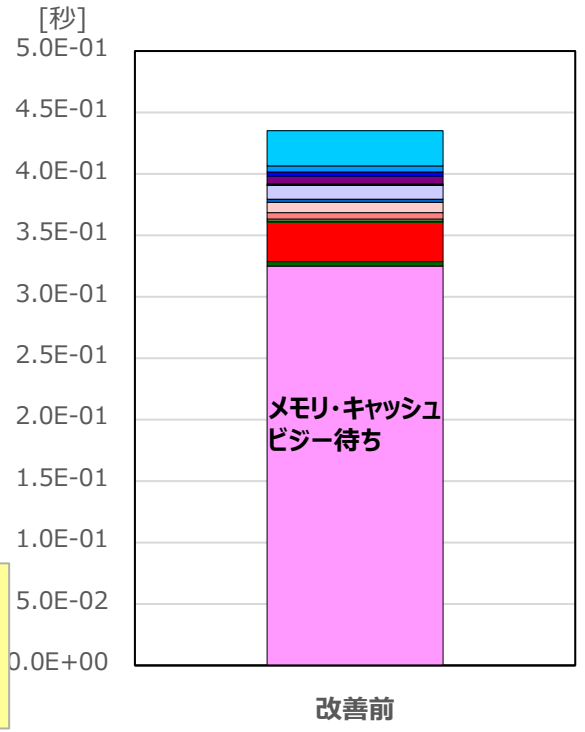
改善前ソース

```
107            for (iter=0; iter<niter; iter++){
108           #pragma omp parallel for private(i,j,k)
109   p          for(k=1;k<=n3-2; k++){
           <<< Loop-information Start >>>
           <<<  [OPTIMIZATION]
           <<<    PREFETCH(HARD) Expected by compiler :
           <<<      (unknown)
           <<< Loop-information  End >>>
110   p          for(j=1;j<=n2-2;j++){
           <<< Loop-information Start >>>
           <<<  [OPTIMIZATION]
           <<<    SIMD(VL: 8)
           <<<    SOFTWARE PIPELINING(IPC: 2.12, ITR: 120, MVE: 8, POL: S)
           <<<    PREFETCH(HARD) Expected by compiler :
           <<<      (unknown)
           <<< Loop-information  End >>>
111   p     v        for(i=1;i<=n1-2;i++){
112   p     v          unew[k][j][i] =
113                    ((u[k][j][i+1] + u[k][j][i-1]) * h1sqinv
114                    +(u[k][j+1][i] + u[k][j-1][i]) * h2sqinv
115                    +(u[k+1][j][i] + u[k-1][j][i]) * h3sqinv
116                    -rhs[k][j][i]) * hhhinv;
117   p     v        }
118   p            }
119   p          }
120
121          }
```

n1=452
n2=52
n3=322

配列サイズ
unew: 60.5MB
u: 60.5MB
rhs: 60.5MB

配列uのi,j次元に再利用性があるため、配列uをオンキャッシュにしたい。



メモリアクセス負荷が高い

Statistics	Memory throughput (GB/s)
改善前	173.23

Cache	L2 miss	L2 miss rate (/Load-store instruction)	L2 miss demand rate (%) (/L2 miss)	L2 miss hardware prefetch rate (%) (/L2 miss)	L2 miss software prefetch rate (%) (/L2 miss)
改善前	2.46E+08	0.15	2.38%	98.32%	0.00%

セクタキャッシュ：事例2（ソースチューニング）

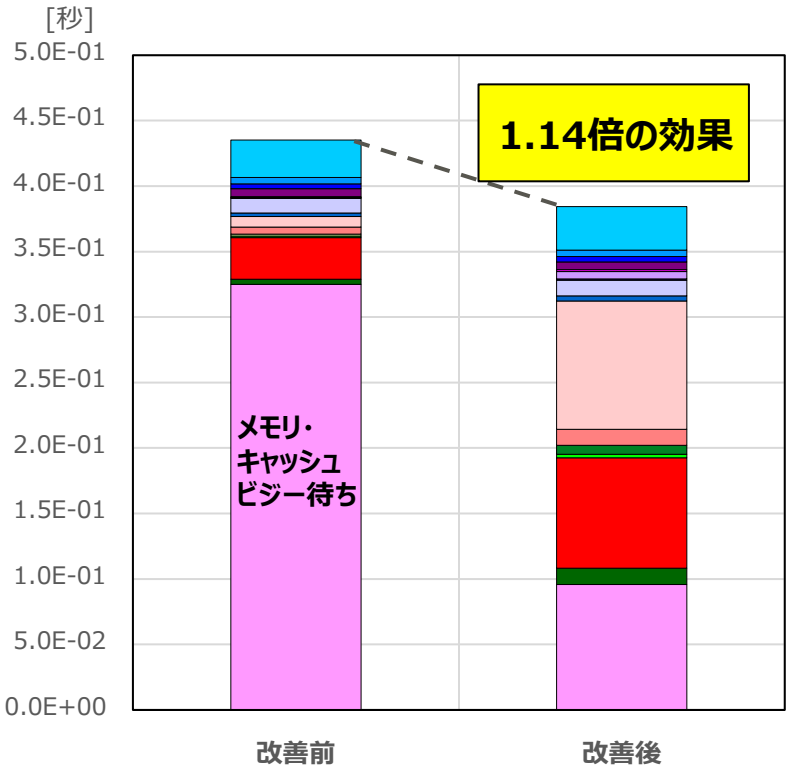
FUJITSU

配列uのk次元の一部をセクタ１に載せることで、キャッシュ効率が向上し、メモリ・キャッシュビジー待ちが改善されました。

改善後ソース

```
107          #pragma statement scache_isolate_way L2=13
108          #pragma statement scache_isolate_assign u
109            for (iter=0; iter<niter; iter++){
110          #pragma omp parallel for private(i,j,k)
111    p           for(k=1;k<=n3-2; k++){
           <<< Loop-information Start >>>
           <<<  [OPTIMIZATION]
           <<<    PREFETCH(HARD) Expected by compiler :
           <<<      (unknown)
           <<< Loop-information  End >>>
112    p             for(j=1;j<=n2-2;j++){
           <<< Loop-information Start >>>
           <<<  [OPTIMIZATION]
           <<<    SIMD(VL: 8)
           <<<    SOFTWARE PIPELINING(IPC: 2.12, ITR: 120, MVE: 8, POL: S)
           <<<    PREFETCH(HARD) Expected by compiler :
           <<<      (unknown)
           <<< Loop-information  End >>>
113    p     v          for(i=1;i<=n1-2;i++){
114    p     v            unew[k][j][i] =
115                       ((u[k][j][i+1] + u[k][j][i-1]) * h1sqinv
116                       +(u[k][j+1][i] + u[k][j-1][i]) * h2sqinv
117                       +(u[k+1][j][i] + u[k-1][j][i]) * h3sqinv
118                       -rhs[k][j][i]) * hhhinv;
119    p     v          }
120    p              }
121    p            }
122
123            }
124          #pragma statement end_scache_isolate_assign
125          #pragma statement end_scache_isolate_way
```

配列uの再利用性が向上



Statistics	Memory throughput (GB/s)
改善前	173.23
改善後	167.47

L2ミスが減少

Cache	L2 miss	L2 miss rate (/Load-store instruction)	L2 miss demand rate (%) (/L2 miss)	L2 miss hardware prefetch rate (%) (/L2 miss)	L2 miss software prefetch rate (%) (/L2 miss)
改善前	2.46E+08	0.15	2.38%	98.32%	0.00%
改善後	1.99E+08	0.12	13.90%	86.99%	0.00%

ループ交換

- ループ交換とは
- ループ交換（改善前）
- ループ交換のチューニング内容
- ループ交換の効果（ソースチューニング）

ループ交換とは

FUJITSU

ループ交換とは、多重ループに対してループの順番を入れ替えることにより、データのアクセス効率を向上させる手法です。
Fortran言語の場合、配列は列方向に連続に確保されます。そのため、以下のようにループの順序を変更し連続アクセスにすることで、高速に動作します。

（C言語では、配列は行方向に連続に確保されるため、Fortranとは逆の順序になります）

改善前ソース例

```
do j=1,n1
  do i=1,n2
    a(j,i) = b(j,i) * c(j,i)
  enddo
enddo
```

配列a, b, cはストライドアクセス
キャッシュ効率が悪い

改善後ソース例

```
do i=1,n2
  do j=1,n1
    a(j,i) = b(j,i) * c(j,i)
  enddo
enddo
```

ループ順序を変更することで
連続アクセスになり
キャッシュ効率が改善

- ポイント
  - 最内ループの繰返し数が小さい場合は、ソフトウェアパイプライニングが行われなくなる場合があるため注意が必要です。
    ただし、最内ループをSIMD長に固定できる場合は、その外側のループでソフトウェアパイプライニングされます。
  - ストアする配列とロードする配列でアクセス方向が異なる場合は、ストアする配列を連続アクセスにする方が性能が向上します。
- 注意事項
  - clangモードではループ交換機能を使用できません

ループ交換のチューニング内容

FUJITSU

改善前ソース

```
do j=1,n1
   do i=1,n2
      a(i) = s1 + c(j,i) / (s1 + s2 / d(j,i))
   enddo
   do i=2,n2
      b(j,i) = a(i) / (s2 + s1 / a(i-1))
   enddo
enddo
```

ループ1

ループ2

配列b, c, dはストライドアクセスなので
キャッシュ効率が悪い

① 配列aを2次元配列にする

```
do j=1,n1
   do i=1,n2
      a(j,i) = s1 + c(j,i) / (s1 + s2 / d(j,i))
   enddo
   do i=2,n2
      b(j,i) = a(j,i) / (s2 + s1 / a(j,i-1))
   enddo
enddo
```

ループ分割の阻害要因である配列aの依存がなくなる

② ループ1とループ2を分割する

```
do j=1,n1
   do i=1,n2
      a(j,i) = s1 + c(j,i) / (s1 + s2 / d(j,i))
   enddo
enddo

do j=1,n1
   do i=2,n2
      b(j,i) = a(j,i) / (s2 + s1 / a(j,i-1))
   enddo
enddo
```

③ ループを交換する

```
do i=1,n2
   do j=1,n1
      a(j,i) = s1 + c(j,i) / (s1 + s2 / d(j,i))
   enddo
enddo
- - - - - - - - - - - - - - - - - - - - - - -
do j=2,n2
   do j=1,n1
      b(j,i) = a(j,i) / (s2 + s1 / a(j,i-1))
   enddo
enddo
```

配列b, c, dが連続アクセスになり
キャッシュ効率が改善される

Fortran ループ交換（改善前）

FUJITSU

配列b、配列c、配列dはストライドアクセスのためキャッシュの利用効率が悪く、浮動小数点ロードアクセス待ちが多くなっています。

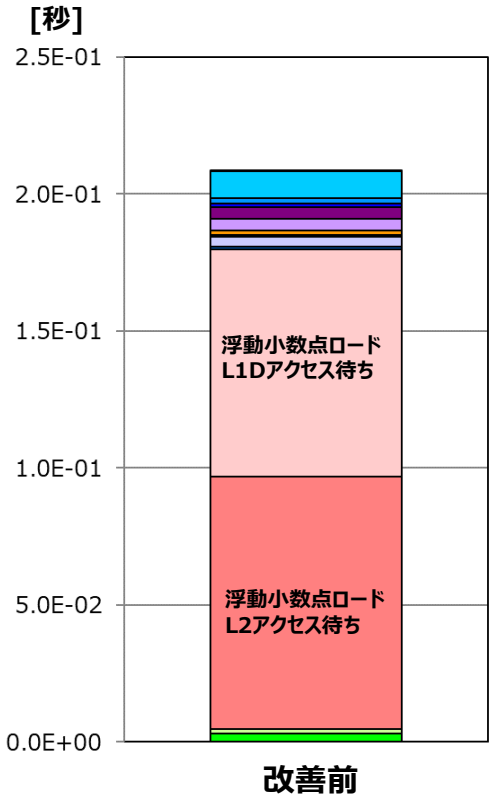
改善前ソース

```
45                    real*8 a(n2),b(n1,n2),c(n1,n2),d(n1,n2)
46                    real*8 s1,s2
47                    integer n1,n2
48                   !$omp parallel
49                   !$omp do private(a)
50    1   p            do j=1,n1
                  <<< Loop-information Start
                  <<<  [OPTIMIZATION]
                  <<<    SIMD(VL: 8)
                  <<<    SOFTWARE PIPELINING(IPC: 1.96, ITR: 112, MVE: 3, POL: S)
                  <<< Loop-information  End >>>
51    2   p   2v           do i=1,n2
52    2   p   2v             a(i) = s1 + c(j,i) / (s1 + s2 / d(j,i))
53    2   p   2v           enddo
                  <<< Loop-information Start >>>
                  <<<  [OPTIMIZATION]
                  <<<    SIMD(VL: 8)
                  <<<    SOFTWARE PIPELINING(IPC: 2.27, ITR: 96, MVE: 2, POL: S)
                  <<< Loop-information  End >>>
54    2   p   2v           do i=2,n2
55    2   p   2v             b(j,i) = a(i) / (s2 + s1 / a(i-1))
56    2   p   2v           enddo
57    1   p              enddo
58                   !$omp end do
59                   !$omp end parallel
```

配列b、c、dは
ストライドアクセスなので
キャッシュ効率が悪い

ループ1

ループ2



L1Dミスが大きい

Cache	L1I miss rate (/Effective instruction)	Load-store instruction	L1D miss	L1D miss rate (/Load-store instruction)	L1D miss demand rate (%) (/L1D miss)	L1D miss hardware prefetch rate (%) (/L1D miss)	L1D miss software prefetch rate (%) (/L1D miss)	L2 miss	L2 miss rate (/Load-store instruction)	L2 miss demand rate (%) (/L2 miss)	L2 miss hardware prefetch rate (%) (/L2 miss)	L2 miss software prefetch rate (%) (/L2 miss)
改善前	0.00	3.60E+08	3.89E+08	1.08	98.28%	1.72%	0.00%	1.07E+04	0.00	62.54%	46.87%	0.00%

Fortran ループ交換の効果（ソースチューニング）

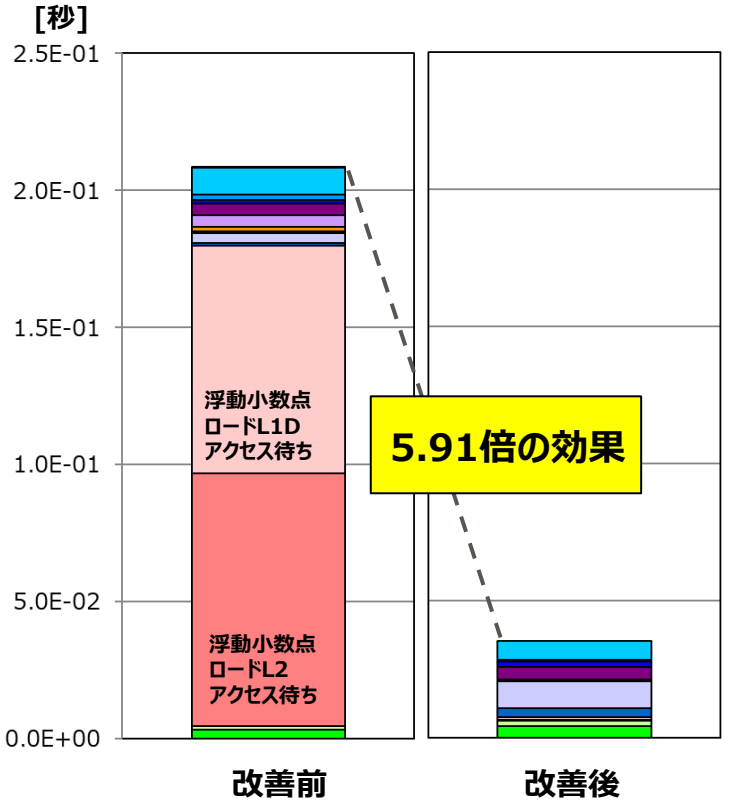
FUJITSU

ループ交換により、配列を連続アクセスすることでキャッシュ効率が向上し、浮動小数点ロードアクセス待ちが改善されました。

改善後ソース

```
45                  real*8 a(n1,n2),b(n1,n2),c(n1,n2),d(n1,n2)
46                  real*8 s1,s2
47                  integer n1,n2
48                 !$omp parallel
49                 !$omp do
50    1   p           do i=1,n2
                 <<< Loop-information Sta
                 <<<  [OPTIMIZATION]
                 <<<    SIMD(VL: 8)
                 <<<    SOFTWARE PIPELINING(IPC: 2.13, ITR: 112, MVE: 2, POL: S)
                 <<< Loop-information  End >>>
51    2   p   2v          do j=1,n1
52    2   p   2v            a(j,i) = s1 + c(j,i) / (s1 + s2 / d(j,i))
53    2   p   2v          enddo
54    1   p           enddo
55                 !$omp end do
56                 !$omp do
57    1   p           do i=2,n2
                 <<< Loop-information Start >>>
                 <<<  [OPTIMIZATION]
                 <<<    SIMD(VL: 8)
                 <<<    SOFTWARE PIPELINING(IPC: 2.04, ITR: 96, MVE: 2, POL: S)
                 <<< Loop-information  End >>>
58    2   p   2v          do j=1,n1
59    2   p   2v            b(j,i) = a(j,i) / (s2 + s1 / a(j,i-1))
60    2   p   2v          enddo
61    1   p           enddo
62                 !$omp end do
63                 !$omp end parallel
```

チューニング内容
①配列aを2次元配列にする
②ループ1とループ2を分割する
③ループ交換する



L1Dミス数が大幅に減少

Cache	L1I miss rate (/Effective instruction)	Load-store instruction	L1D miss	L1D miss rate (/Load-store instruction)	L1D miss demand rate (%) (/L1D miss)	L1D miss hardware prefetch rate (%) (/L1D miss)	L1D miss software prefetch rate (%) (/L1D miss)	L2 miss	L2 miss rate (/Load-store instruction)	L2 miss demand rate (%) (/L2 miss)	L2 miss hardware prefetch rate (%) (/L2 miss)	L2 miss software prefetch rate (%) (/L2 miss)
改善前	0.00	3.60E+08	3.89E+08	1.08	98.28%	1.72%	0.00%	1.07E+04	0.00	62.54%	46.87%	0.00%
改善後	0.00	1.94E+08	2.15E+07	0.11	9.28%	90.72%	0.00%	8.66E+03	0.00	17.98%	87.84%	0.00%

C/C++ ループ交換（改善前）

FUJITSU

配列b、配列c、配列dはストライドアクセスのためキャッシュの利用効率が悪く、浮動小数点ロードアクセス待ちが多くなっています。

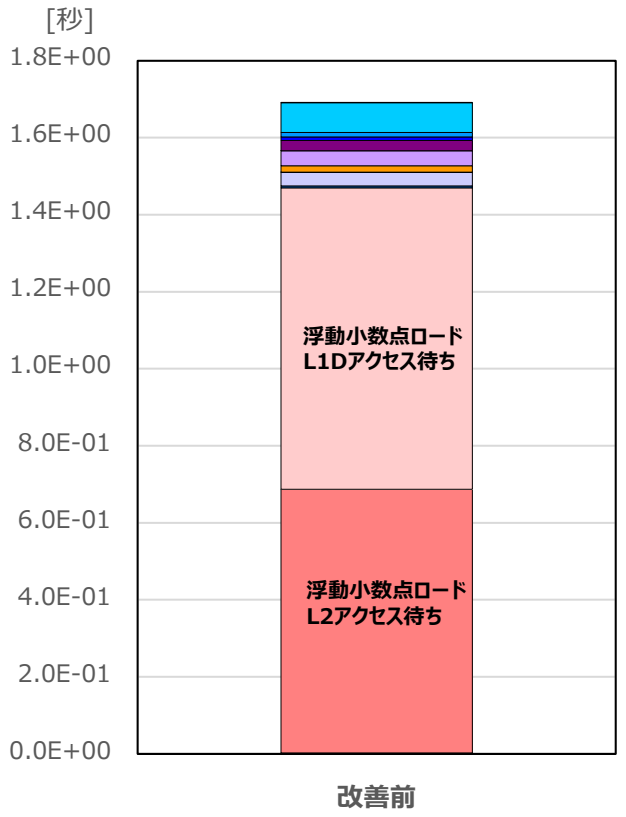
改善前ソース

```
33          void sub(int n,int m,double s1,double s2) {
34            double a[n2];
35            int i,j;
36
          <<< Loop-information Start >>>
          <<<  [OPTIMIZATION]
          <<<    PREFETCH(HARD) Expected by compiler :
          <<<      a
          <<< Loop-information  End >>>
37            for(j=0;j<n1;j++) {
          <<< Loop-information Start >>>
          <<<  [OPTIMIZATION]
          <<<    SIMD(VL: 8)
          <<<    SOFTWARE PIPELINING(IPC: 1.92, ITR: 112, MVE: 3, POL: S)
           <<<    PREFETCH(HARD) Expected by compiler :
          <<<      a
          <<< Loop-information  End >>>
38        2v      for(i=0;i<n2;i++) {
39        2v        a[i] = s1 + c[i][j] / (s1 + s2 / d[i][j]);
40        2v      }
          <<< Loop-information Start >>>
          <<<  [OPTIMIZATION]
          <<<    SIMD(VL: 8)
          <<<    SOFTWARE PIPELINING(IPC: 2.18, ITR: 96, MVE: 2, POL: S)
          <<<    PREFETCH(HARD) Expected by compiler :
          <<<      a
          <<< Loop-information  End >>>
41        2v      for(i=1;i<n2;i++) {
42        2v        b[i][j] = a[i] / (s2 + s1 / a[i-1]);
43        2v      }
44            }
45          }
```

配列b、c、dはストライドアクセスなのでキャッシュ効率が悪い

ループ1

ループ2



L1Dミスが大きい

Cache	L1I miss rate (/Effective instruction)	Load-store instruction	L1D miss	L1D miss rate (/Load-store instruction)	L1D miss demand rate (%) (/L1D miss)	L1D miss hardware prefetch rate (%) (/L1D miss)	L1D miss software prefetch rate (%) (/L1D miss)	L2 miss	L2 miss rate (/Load-store instruction)	L2 miss demand rate (%) (/L2 miss)	L2 miss hardware prefetch rate (%) (/L2 miss)	L2 miss software prefetch rate (%) (/L2 miss)
改善前	0.00	1.87E+09	3.89E+08	0.21	97.94%	2.06%	0.00%	2.18E+04	0.00	71.23%	38.77%	0.00%

C/C++ ループ交換の効果（ソースチューニング）

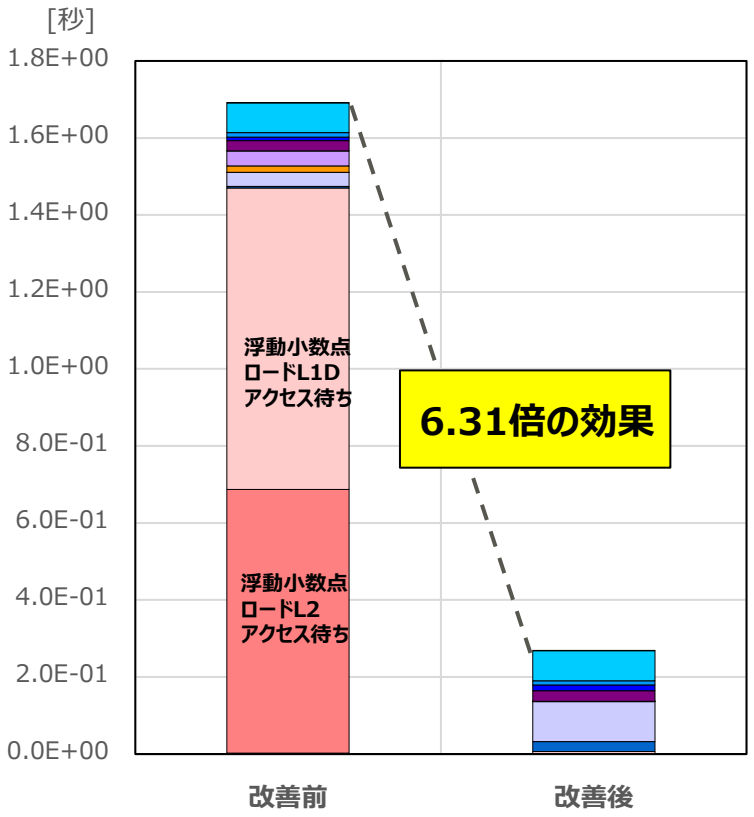
FUJITSU

ループ交換により、配列を連続アクセスすることでキャッシュ効率が向上し、浮動小数点ロードアクセス待ちが改善されました。

改善後ソース

```
34          void sub(int n1,int n2,double s1,double s2)
35          {
36            double a[m][n];
37            int i,j;
38
           <<< Loop-information Start >>>
           <<<  [OPTIMIZATION]
           <<<    PREFETCH(HARD) Expected by compiler :
           <<<      d, c, a
           <<< Loop-information  End >>>
39            for(i=0;i<n2;i++) {
           <<< Loop-information Start >>>
           <<<  [OPTIMIZATION]
           <<<    SIMD(VL: 8)
           <<<    SOFTWARE PIPELINING(IPC: 1.88, ITR: 96, MVE: 2, POL: S)
           <<<    PREFETCH(HARD) Expected by compiler :
           <<<      c, d, a
           <<< Loop-information  End >>>
40         2v     for(j=0;j<n1;j++) {
41         2v       a[i][j] = s1 + c[i][j] / (s1 + s2 / d[i][j]);
42         2v     }
43            }
           <<< Loop-information Start >>>
           <<<  [OPTIMIZATION]
           <<<    PREFETCH(HARD) Expected by compiler :
           <<<      a, b
           <<< Loop-information  End >>>
44            for(i=0;i<n2;i++) {
           <<< Loop-information Start >>>
           <<<  [OPTIMIZATION]
           <<<    SIMD(VL: 8)
           <<<    SOFTWARE PIPELINING(IPC: 2.09, ITR: 96, MVE: 2, POL: S)
           <<<    PREFETCH(HARD) Expected by compiler :
           <<<      a, b
           <<< Loop-information  End >>>
45         2v     for(j=0;j<n1;j++) {
46         2v       b[i][j] = a[i][j] / (s2 + s1 / a[i-1][j]);
47         2v     }
48            }
49          }
```

チューニング内容
①配列aを2次元配列にする
②ループ1とループ2を分割する
③ループ交換する



L1Dミス数が大幅に減少

Cache	L1I miss rate (/Effective instruction)	Load-store instruction	L1D miss	L1D miss rate (/Load-store instruction)	L1D miss demand rate (%) (/L1D miss)	L1D miss hardware prefetch rate (%) (/L1D miss)	L1D miss software prefetch rate (%) (/L1D miss)	L2 miss	L2 miss rate (/Load-store instruction)	L2 miss demand rate (%) (/L2 miss)	L2 miss hardware prefetch rate (%) (/L2 miss)	L2 miss software prefetch rate (%) (/L2 miss)
改善前	0.00	1.87E+09	3.89E+08	0.21	97.94%	2.06%	0.00%	2.18E+04	0.00	71.23%	38.77%	0.00%
改善後	0.00	1.87E+09	2.27E+07	0.01	14.23%	85.77%	0.00%	3.63E+04	0.00	42.14%	62.84%	0.00%

FUJITSU

ループ融合

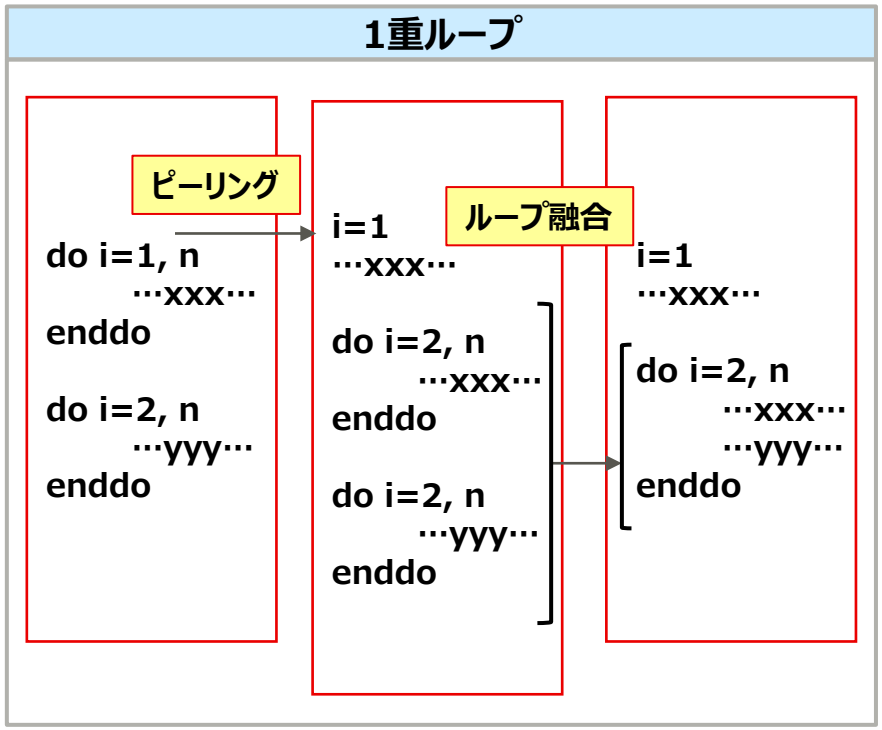
- ループ融合とは
- ループ融合（改善前）
- ループ融合の効果（ソースチューニング）

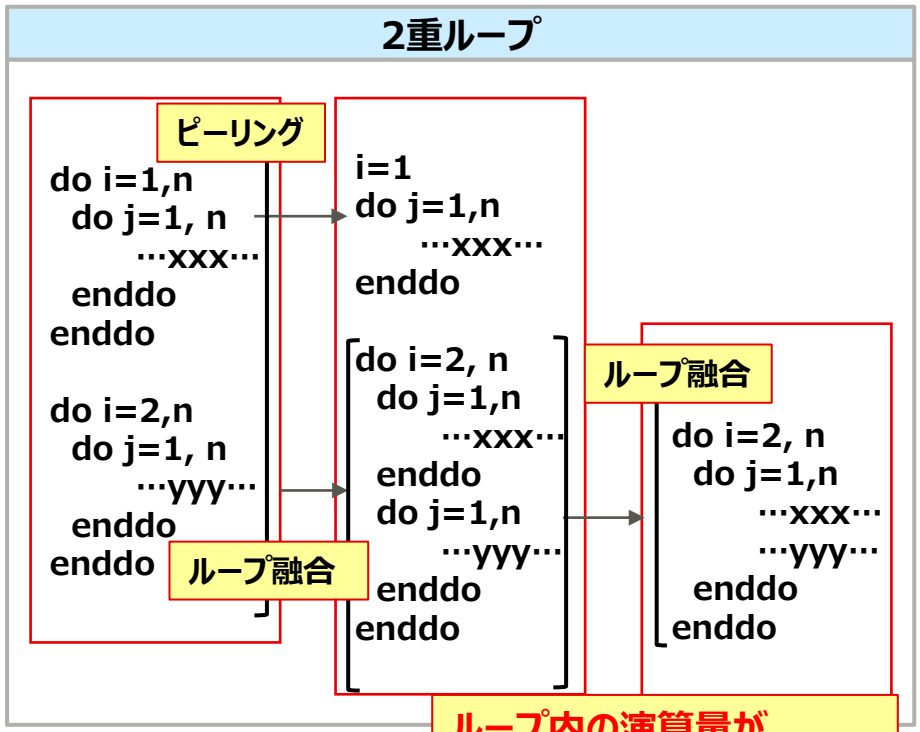
ループ融合とは

FUJITSU

ループ融合とは、以下の効果を目的にループを連結させることです。

- データの局所化：配列の再利用を行う
- 命令レベルの並列化向上：ループ内の命令数を増やし、命令レベルの並列性を向上させる





■ ポイント

- ループ長が同じループの結合はコンパイラによって自動で行われます。
- ループ内の演算が多くなりすぎると、ソフトウェアパイプライニングが促進されなくなる場合があります。

Fortran ループ融合（改善前）

FUJITSU

ループ１の繰返し数が多く、配列データがキャッシュに載りきらないため、ループ２で再利用できません。そのため、メモリ・キャッシュビジー待ちが多くなっています。

改善前ソース

```
42    1  pp   v      do j=1,m-1
                  <<< Loop-information Start >>>
                  <<<  [OPTIMIZATION]
                  <<<    COLLAPSED
                  <<< Loop-information  End >>>
43    2   p           do i=1,n
44    2   p   v         s1 = s1 + a(i,j) * (s3 * b(i,j) + c(i,j) * (s2 + s3 * d(i,j) ))
45    2   p   v        enddo
46    1   p   v      enddo
47
                  <<< Loop-information Start >>>
                  <<<  [PARALLELIZATION]
                  <<<    Standard iteration count: 572
                  <<<  [OPTIMIZATION]
                  <<<    COLLAPSED
                  <<<    SIMD(VL: 8)
                  <<<    SOFTWARE PIPELINING(IPC: 2.30, ITR: 96, MVE: 2, POL: S)
                  <<<    PREFETCH(HARD) Expected by compiler :
                  <<<     b, a, d, c, e
                  <<< Loop-information  End >>>
48    1  pp   2v      do j=1,m
                  <<< Loop-information Star
                  <<<  [OPTIMIZATION]
                  <<<    COLLAPSED
                  <<< Loop-information  End >>>
49    2   p   2         do i=1,n
50    2   p   2v          e(i,j) = s2 * (a(i,j) + b(i,j) * (s3 + c(i,j) * d(i,j)))
51    2   p   2v        enddo
52    1   p         enddo
```

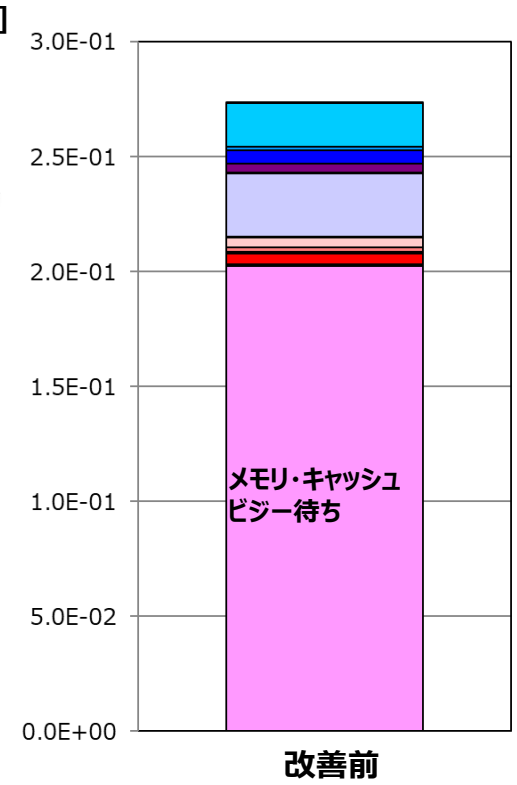
m = 50
n = 150000
配列の型 ： real*8

ループ１

配列データの合計：約200MB
配列データがキャッシュに載りきらない

配列アクセスがキャッシュミス

ループ２



L1キャッシュミス率とL2キャッシュミス率はストリームアクセスの理論値である0.24になっています。ただし、ループ1,2の両方でミスが発生しています。つまりループ1でキャッシュに載せたデータをループ2で使用できていません。

Cache	L1I miss rate (/Effective instruction)	Load-store instruction	L1D miss	L1D miss rate (/Load-store instruction)	L1D miss demand rate (%) (/L1D miss)	L1D miss hardware prefetch rate (%) (/L1D miss)	L1D miss software prefetch rate (%) (/L1D miss)	L2 miss	L2 miss rate (/Load-store instruction)	L2 miss demand rate (%) (/L2 miss)	L2 miss hardware prefetch rate (%) (/L2 miss)	L2 miss software prefetch rate (%) (/L2 miss)
改善前	0.00	8.57E+08	2.09E+08	0.24	0.64%	99.35%	0.01%	2.09E+08	0.24	0.66%	99.50%	0.00%

Fortran ループ融合（ソースチューニング）

FUJITSU

ループ融合によってキャッシュ効率が向上し、メモリ・キャッシュビジー待ちが改善されました。

改善後ソース（ソースチューニング）

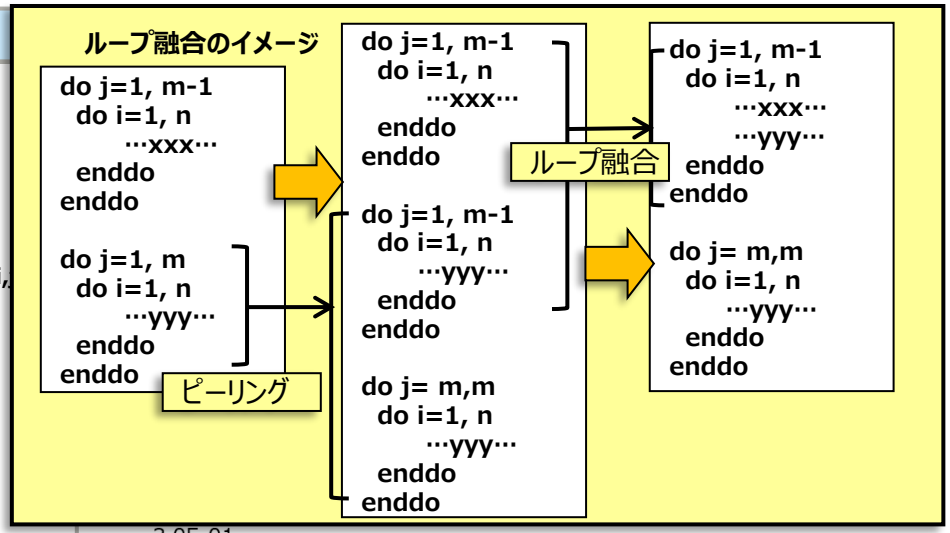
```
42   1 pp   v      do j=1,m-1
                   <<< Loop-information Start >>>
                   <<<  [OPTIMIZATION]
                   <<<    COLLAPSED
                   <<< Loop-information  End >>>
43   2  p            do i=1,n
44   2  p   v          s1 = s1 + a(i,j) * (s3 * b(i,j) + c(i,j) * (s2 + s3 * d(i,
45   2  p   v          e(i,j) = s2 * (a(i,j) + b(i,j) * (s3 + c(i,j) * d(i,j)))
46   2  p   v        enddo
47   1  p   v      enddo
 :
49   1  s          do j=m,m
                   <<< Loop-information Start >>>
                   <<<  [PARALLELIZATION]
                   <<<    Standard iteration count: 364
                   <<<  [OPTIMIZATION]
                   <<<    SIMD(VL: 8)
                   <<<    SOFTWARE PIPELINI                    S)
                   <<<    PREFETCH(HARD) Exp
                   <<<      b, a, d, c, e
                   <<< Loop-information  End >>>
50   2 pp  2v         do i=1,n
51   2  p  2v           e(i,j) = s2 * (a(i,j) + b(i,j) * (s3 + c(i,j) * d(i,j)))
52   2  p  2v         enddo
53   1  p          enddo
```

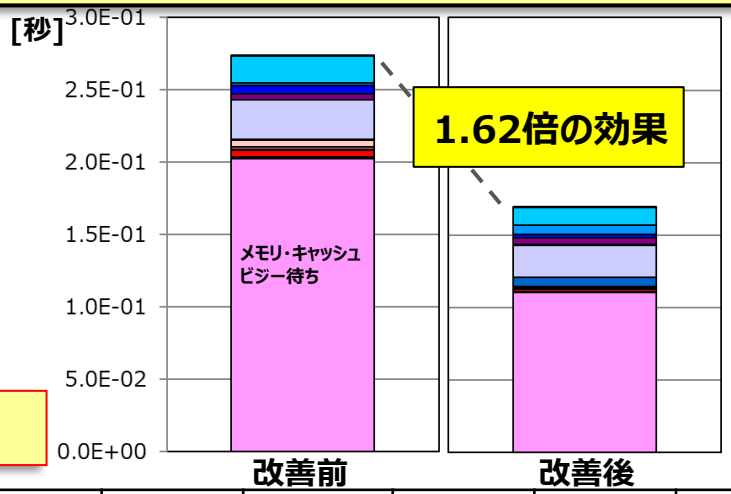
ループ融合

ピーリング

配列アクセスがキャッシュヒットする

ループ融合させるために切り出す（ピーリング）





L1Dミス数とL2ミス数が大幅に減少

Cache	L1I miss rate (/Effective instruction)	Load-store instruction	L1D miss	L1D miss rate (/Load-store instruction)	L1D miss demand rate (%) (/L1D miss)	L1D miss hardware prefetch rate (%) (/L1D miss)	L1D miss software prefetch rate (%) (/L1D miss)	L2 miss	L2 miss rate (/Load-store instruction)	L2 miss demand rate (%) (/L2 miss)	L2 miss hardware prefetch rate (%) (/L2 miss)	L2 miss software prefetch rate (%) (/L2 miss)
改善前	0.00	8.57E+08	2.09E+08	0.24	0.64%	99.35%	0.01%	2.09E+08	0.24	0.66%	99.50%	0.00%
改善後	0.00	4.92E+08	1.18E+08	0.24	0.36%	99.63%	0.00%	1.17E+08	0.24	0.41%	99.75%	0.00%

C/C++ ループ融合（改善前）

FUJITSU

ループ１の繰返し数が多く、配列データがキャッシュに載りきらないため、ループ２で再利用できません。そのため、メモリ・キャッシュビジー待ちが多くなっています。

改善前ソース

```
39          #pragma omp parallel for
40   p        for (j=0;j<m-1;j++) {
            <<< Loop-information Start >>>
            <<<  [OPTIMIZATION]
            <<<    SIMD(VL: 8)
            <<<    PREFETCH(HARD) Expected by compiler :
            <<<      (unknown)
            <<< Loop-information  End >>>
41   p    8v     for (i=0;i<n;i++) {
42   p    8v       *s1 = *s1 + a[j][i] * (*s3 * b[j][i] + c[j][i] * (*s2 + *s3 * d[j][i] ));
43   p    8v     }
44   p        }
45
46          #pragma omp parallel for
            <<< Loop-information Start >>>
            <<<  [OPTIMIZATION]
            <<<    PREFETCH(HARD) Expected by compiler :
            <<<      (unknown)
            <<< Loop-information  End >>>
47   p        for (j=0;j<m;j++) {
            <<< Loop-information Start >>>
            <<<  [OPTIMIZATION]
            <<<    SIMD(VL: 8)
            <<<    SOFTWARE PIPELINING(IPC: 2.30, ITR: 96, MVE: 2, POL: S)
            <<<    PREFETCH(HARD) Expected by compiler :
            <<<      (unknown)
            <<< Loop-information  End >>>
48   p    2v     for (i=0;i<n;i++) {
49   p    2v       e[j][i] = *s2 * (a[j][i] + b[j][i]* (*s3 + c[j][i] * d[j][i] ));
50   p    2v     }
51   p        }
```

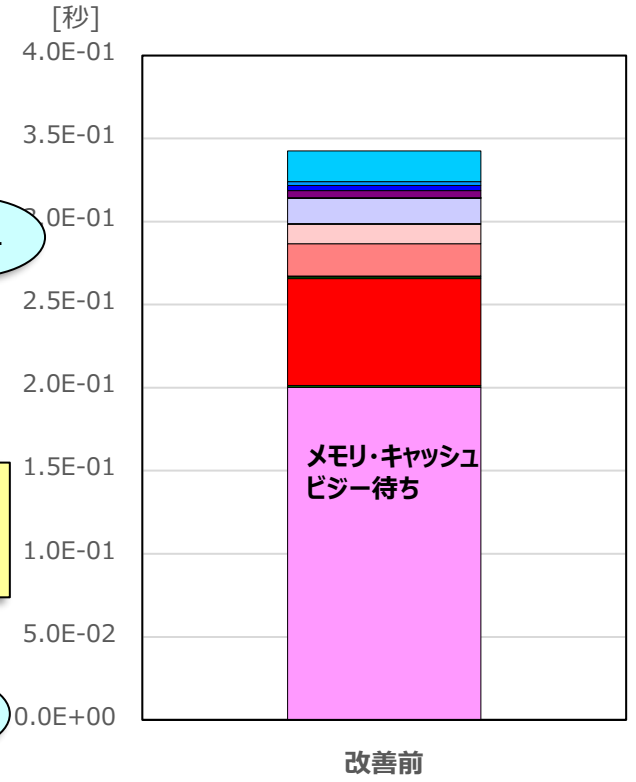
m = 50
n = 150000
配列の型 ： double

ループ１

配列データの合計：約200MB
配列データがキャッシュに載りきらない

配列アクセスがキャッシュミス

ループ２



L1キャッシュミス率とL2キャッシュミス率はストリームアクセスの理論値に近い0.21になっています。ただし、ループ1,2の両方でミスが発生しています。つまりループ1でキャッシュに載せたデータをループ2で使用できていません。

Cache	L1I miss rate (/Effective instruction)	Load-store instruction	L1D miss	L1D miss rate (/Load-store instruction)	L1D miss demand rate (%) (/L1D miss)	L1D miss hardware prefetch rate (%) (/L1D miss)	software prefetch rate (%) (/L1D miss)	L2 miss	L2 miss rate (/Load-store instruction)	L2 miss demand rate (%) (/L2 miss)	hardware prefetch rate (%) (/L2 miss)	software prefetch rate (%) (/L2 miss)
改善前	0.00	9.87E+08	2.10E+08	0.21	9.82%	90.20%	-0.02%	2.10E+08	0.21	5.83%	95.03%	0.00%

C/C++ ループ融合（ソースチューニング）

FUJITSU

ループ融合によってキャッシュ効率が向上し、メモリ・キャッシュビジー待ちが改善されました。

改善後ソース（ソースチューニング）

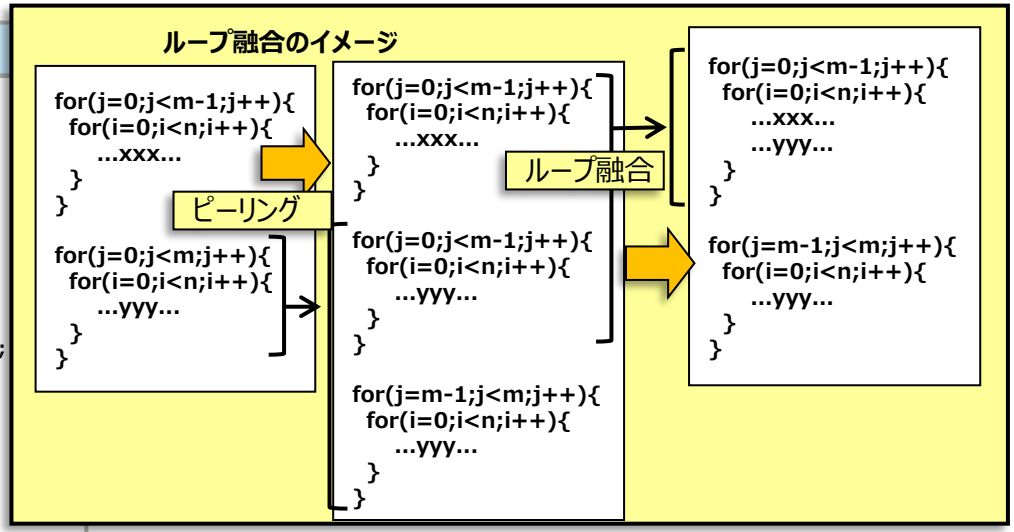
ループ融合

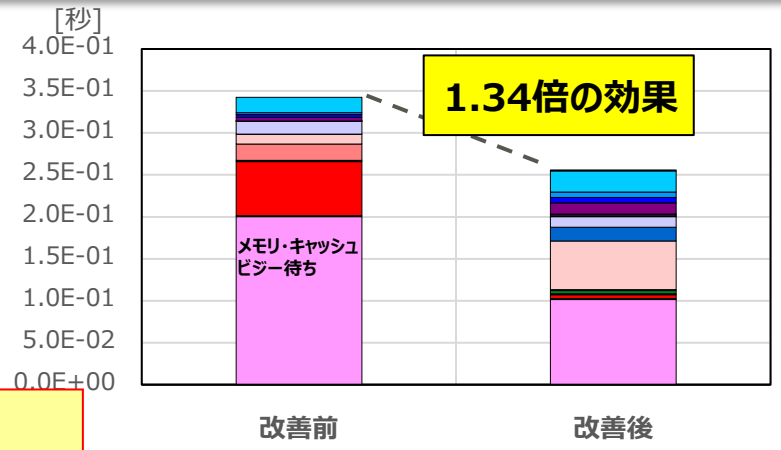
ピーリング

```
40  p       for (j=0;j<m-1;j++) {
            <<< Loop-information Start >>>
            <<<  [OPTIMIZATION]
            <<<    SIMD(VL: 8)
            <<<    SOFTWARE PIPELINING(IPC: 1.98, ITR: 192, MVE: 2, POL: S)
            <<<    PREFETCH(HARD) Expected by compiler :
            <<<    ...
            <<< Loop-information  End >>>
41  p   8v      for (i=0;i<n;i++) {
42  p   8v        *s1 = *s1 + a[j][i] * (*s3 * b[j][i] + c[j][i] * (*s2 + *s3 * d[j][i] ));
43  p   8v        e[j][i] = *s2 * (a[j][i] + b[j][i]* (*s3 + c[j][i] * d[j][i] ));
44  p   8v      }
45  p       }
...
48  p       for (j=m-1;j<m;j++) {
            <<< Loop-information Start >>>
            <<<  [OPTIMIZATION]
            <<<    SIMD(VL: 8)
            <<<    SOFTWARE PIPELINING(IPC: 3.83, ITR: 176, MVE: 3, POL: S)
            <<<    PREFETCH(HARD) Expected by compiler :
            <<<    ...
            <<< Loop-information  End >>>
49  p   2v      for (i=0;i<n;i++) {
50  p   2v        e[j][i] = *s2 * (a[j][i] + b[j][i]* (*s3 + c[j][i] * d[j][i] ));
51  p   2v      }
52  p       }
```

配列アクセスがキャッシュヒットする

ループ融合させるために切り出す（ピーリング）





L1Dミス数とL2ミス数が大幅に減少

Cache	L1I miss rate (/Effective instruction)	Load-store instruction	L1D miss	L1D miss rate (/Load-store instruction)	L1D miss demand rate (%) (/L1D miss)	L1D miss hardware prefetch rate (%) (/L1D miss)	L1D miss software prefetch rate (%) (/L1D miss)	L2 miss	L2 miss rate (/Load-store instruction)	L2 miss demand rate (%) (/L2 miss)	L2 miss hardware prefetch rate (%) (/L2 miss)	L2 miss software prefetch rate (%) (/L2 miss)
改善前	0.00	9.87E+08	2.10E+08	0.21	9.82%	90.20%	-0.02%	2.10E+08	0.21	5.83%	95.03%	0.00%
改善後	0.00	1.94E+09	1.19E+08	0.06	2.55%	97.54%	-0.09%	1.18E+08	0.06	1.46%	98.73%	0.00%

FUJITSU

配列マージ（インダイレクトアクセスの効率改善）

- 配列マージとは
- 配列マージ（改善前）
- 配列マージの効果（ソースチューニング）

配列マージとは

FUJITSU

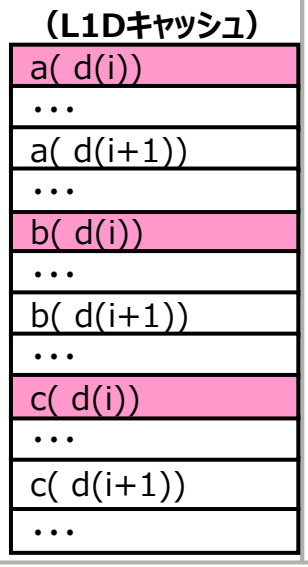
配列マージとは、同一ループ内でアクセスパターンが共通の配列が複数ある場合に、それらを１つの配列に融合することです。データアクセスを連続化して、キャッシュ効率を向上させます。

改善前ソース	改善後ソース

```
parameter(n=1000000)
    real*8  a(n), b(n),
    integer d(n+10)
      :
    do iter = 1, 100
      do i = 1 , n
        a(d(i)) = b(d(i))
                + scalar * c(d(i))
      enddo
    enddo
      :
```

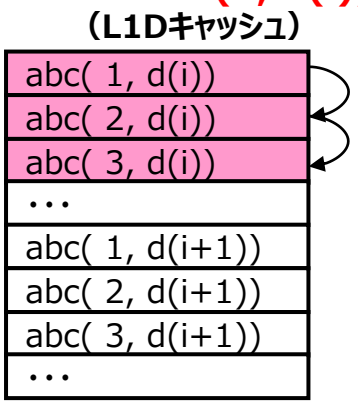
配列マージ

```
parameter(n=1000000)
    real*8  abc(3, n)
    integer d(n+10)
      :
    do iter = 1, 100
      do i = 1 , n
        abc(1, d(i)) = abc(2, d(i))
                     + scalar * abc(3, d(i))
      enddo
    enddo
      :
```



異なるキャッシュラインのアクセス

（配列dの値が連続値でない場合）



同じキャッシュラインのアクセス

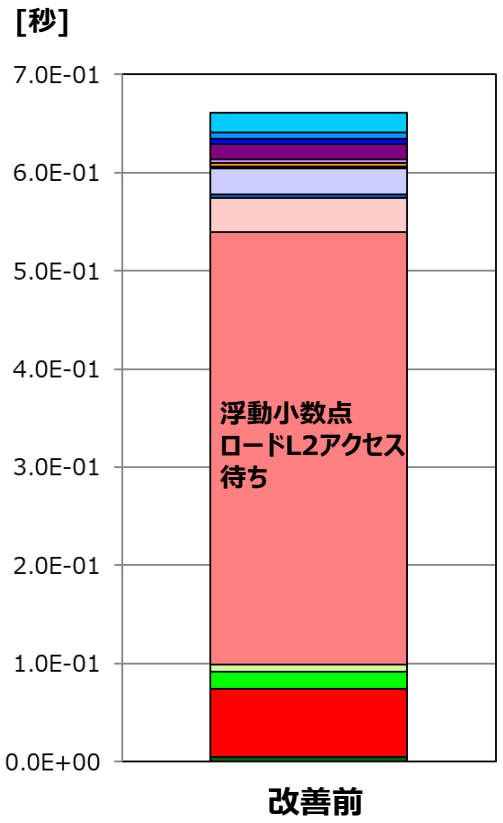
Fortran 配列マージ（改善前）

FUJITSU

L1Dミス率が高いことから、キャッシュの利用効率が悪く（インダイレクトアクセス）、浮動小数点ロードL2アクセス待ちが多くなっています。

改善前ソース

```
 1                    parameter(n=2*1000*1000/8)
 2                    real*8  a(n),b(n),c(n),e(n),f(n),s
 3                    integer d(n)
 :
14    1  s   s           call sub(a,b,c,d,e,f,s,n)
 :
25                     subroutine sub(a,b,c,d,e,f, s, n)
26                     real*8  a(n),b(n),c(n),e(n),f(n),s
27                     integer d(n), ii
28
29                   !$omp parallel do schedule (static,96)
                  <<< Loop-information Start >>>
                  <<<  [OPTIMIZATION]
                  <<<    SIMD(VL: 8)
                  <<<    SOFTWARE PIPELINING(IPC: 1.07, ITR: 80, MVE: 3, POL: S)
                  <<<    PREFETCH(HARD) Expected by compiler :
                  <<<      d
                  <<< Loop-information  End >>>
30    1  p   2v          do i = 1 , n
31    1  p   2v            ii = d(i)
32    1  p   2v          a(ii) = s / (s + f(ii) / ( s + e(ii)  / ( b(ii) + s / c(ii)))
33    1  p   2v          enddo
34                   !$omp end parallel do
```



Cache	L1I miss rate (/Effective instruction)	Load-store instruction	L1D miss	L1D miss rate (/Load-store instruction)	L1D miss demand rate (%) (/L1D miss)	L1D miss hardware prefetch rate (%) (/L1D miss)	L1D miss software prefetch rate (%) (/L1D miss)	L2 miss	L2 miss rate (/Load-store instruction)	L2 miss demand rate (%) (/L2 miss)	L2 miss hardware prefetch rate (%) (/L2 miss)	L2 miss software prefetch rate (%) (/L2 miss)
改善前	0.00	6.44E+08	1.27E+09	1.97	99.98%	0.01%	0.00%	4.82E+07	0.07	60.81%	50.33%	0.00%

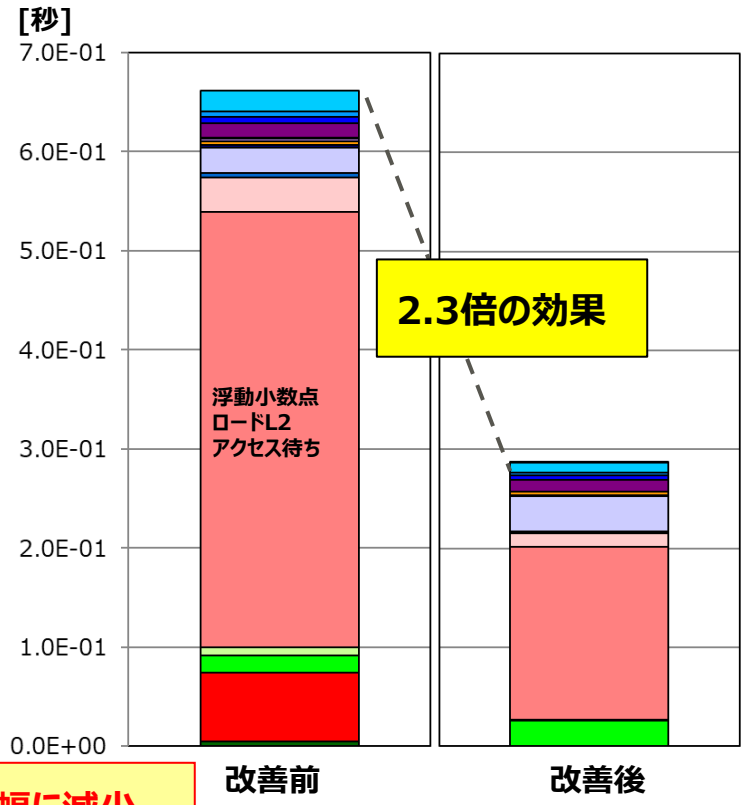
Fortran 配列マージの効果（ソースチューニング）

FUJITSU

リストアクセスの配列を配列マージすることで、キャッシュ効率が向上し、浮動小数点ロードL2アクセス待ちが改善されました。

改善後ソース（ソースチューニング）

```
 1                    parameter(n=2*1000*1000/8)
 2                    real*8  abcef(5,n),s
 3                    integer d(n)
 :
14    1  s   s           call sub(abcef,d,s,n)
 :
24                    subroutine sub(abcef,d, s, n)
25                    real*8  abcef(5,n),s
26                    integer d(n), ii
27
28                 !$omp parallel do schedule (static,96)
                  <<< Loop-information Start >>>
                  <<<  [OPTIMIZATION]
                  <<<    SIMD(VL: 8)
                  <<<    SOFTWARE PIPELINING(IPC: 1.09, ITR: 80, MVE: 3, POL: S)
                  <<<    PREFETCH(HARD) Expected by compiler :
                  <<<      d
                  <<< Loop-information  End >>>
29    1  p   2v          do i = 1 , n
30    1  p   2v            ii = d(i)
31    1  p   2v          abcef(1,ii) = s / (s + abcef(5,ii) / ( s + abcef(4,ii)
32    1                 *  / ( abcef(2,ii) + s / abcef(3,ii))))
33    1  p   2v          enddo
34                 !$omp end parallel do
```



L1Dミス率が大幅に減少

Cache	L1I miss rate (/Effective instruction)	Load-store instruction	L1D miss	L1D miss rate (/Load-store instruction)	L1D miss demand rate (%) (/L1D miss)	L1D miss hardware prefetch rate (%) (/L1D miss)	L1D miss software prefetch rate (%) (/L1D miss)	L2 miss	L2 miss rate (/Load-store instruction)	L2 miss demand rate (%) (/L2 miss)	L2 miss hardware prefetch rate (%) (/L2 miss)	L2 miss software prefetch rate (%) (/L2 miss)
改善前	0.00	6.44E+08	1.27E+09	1.97	99.98%	0.01%	0.00%	4.82E+07	0.07	60.81%	50.33%	0.00%
改善後	0.00	3.19E+08	2.97E+08	0.93	99.68%	0.32%	0.00%	1.58E+04	0.00	63.74%	54.78%	0.00%

C/C++ 配列マージ（改善前）

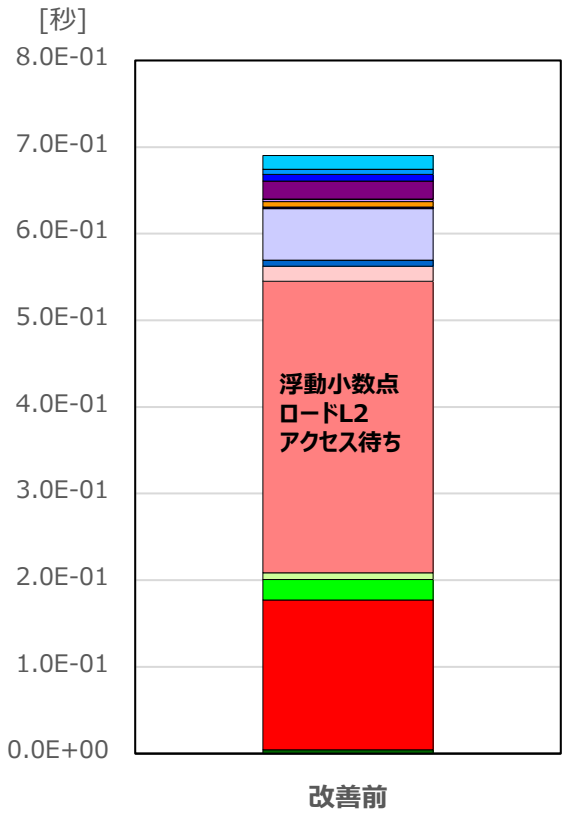
FUJITSU

L1Dミス率が高いことから、キャッシュの利用効率が悪く（インダイレクトアクセス）、浮動小数点ロードL2アクセス待ちが多くなっています。

改善前ソース

配列宣言
```
double a[250000];
double b[250000];
double c[250000];
double e[250000];
double f[250000];
```

```
24 void sub(double (* restrict a),double (* restrict b),
             double (* restrict c),int (* restrict d),double (* restrict e),
             double (* restrict f),double s,int n) {
25             int i,ii;
26
27             #pragma omp parallel for schedule (static,96)
              <<< Loop-information Start >>>
              <<<  [OPTIMIZATION]
              <<<    SIMD(VL: 8)
              <<<    SOFTWARE PIPELINING(IPC: 1.33, ITR: 112, MVE: 4, POL: S)
              <<<    PREFETCH(HARD) Expected by compiler :
              <<<      (unknown)
              <<< Loop-information  End >>>
28   p    2v    for(i=0;i<n;i++) {
29   p    2v      ii = d[i];
30   p    2v      a[ii] = s / (s + f[ii] / ( s + e[ii]  / ( b[ii] + s / c[ii])));
31   p    2v    }
32             }
```



Cache	L1I miss rate (/Effective instruction)	Load-store instruction	L1D miss	L1D miss rate (/Load-store instruction)	L1D miss demand rate (%) (/L1D miss)	L1D miss hardware prefetch rate (%) (/L1D miss)	L1D miss software prefetch rate (%) (/L1D miss)	L2 miss	L2 miss rate (/Load-store instruction)	L2 miss demand rate (%) (/L2 miss)	L2 miss hardware prefetch rate (%) (/L2 miss)	L2 miss software prefetch rate (%) (/L2 miss)
改善前	0.00	6.58E+08	1.27E+09	1.93	99.94%	0.06%	0.00%	4.24E+07	0.06	56.24%	56.98%	0.00%

C/C++ 配列マージの効果（ソースチューニング）

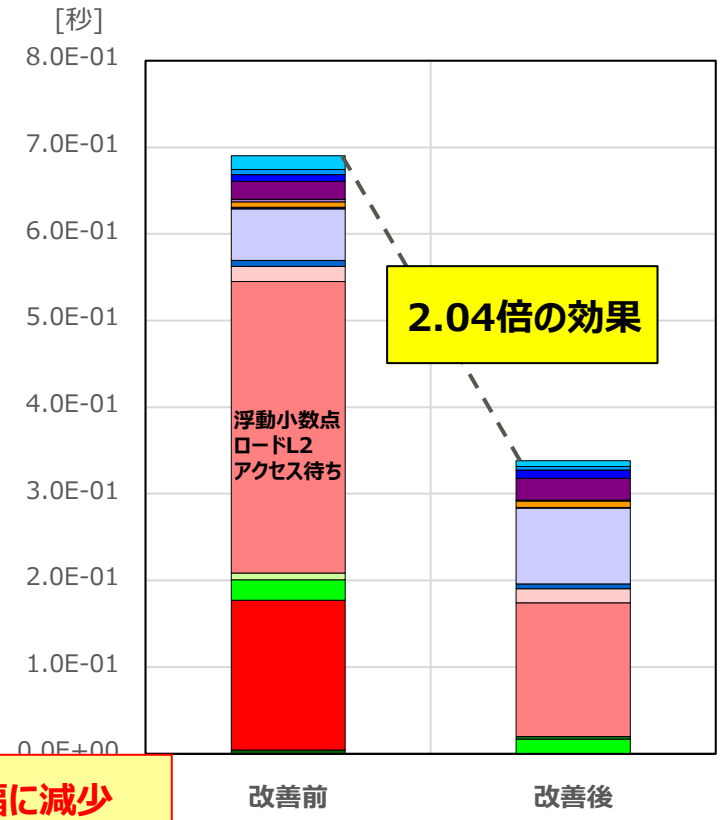
FUJITSU

リストアクセスの配列を配列マージすることで、キャッシュ効率が向上し、浮動小数点ロードL2アクセス待ちが改善されました。

改善後ソース（ソースチューニング）

配列宣言
```
double a[250000];
double b[250000];
double c[250000];
double e[250000];
double f[250000];
```

```
24          void sub(double (* restrict abcef)[5],int (* restrict d),double s,int n) {
25            int i,ii;
26
27          #pragma omp parallel for schedule (static,96)
            <<< Loop-information Start >>>
            <<<  [OPTIMIZATION]
            <<<    SIMD(VL: 8)
            <<<    SOFTWARE PIPELINING(IPC: 1.36, ITR: 112, MVE: 4, POL: S)
            <<<    PREFETCH(HARD) Expected by compiler :
            <<<      (unknown)
            <<< Loop-information  End >>>
28   p    2v    for(i=0;i<n;i++) {
29   p    2v      ii = d[i];
30   p    2v      abcef[ii][0] = s / (s + abcef[ii][4] / ( s + abcef[ii][3]  /
                                    ( abcef[ii][1] + s / abcef[ii][2]))));
31   p    2v    }
32          }
```



L1Dミス率が大幅に減少

Cache	L1I miss rate (/Effective instruction)	Load-store instruction	L1D miss	L1D miss rate (/Load-store instruction)	L1D miss demand rate (%) (/L1D miss)	L1D miss hardware prefetch rate (%) (/L1D miss)	L1D miss software prefetch rate (%) (/L1D miss)	L2 miss	L2 miss rate (/Load-store instruction)	L2 miss demand rate (%) (/L2 miss)	L2 miss hardware prefetch rate (%) (/L2 miss)	L2 miss software prefetch rate (%) (/L2 miss)
改善前	0.00	6.58E+08	1.27E+09	1.93	99.94%	0.06%	0.00%	4.24E+07	0.06	56.24%	56.98%	0.00%
改善後	0.00	3.67E+08	2.94E+08	0.80	99.69%	0.30%	0.00%	1.41E+04	0.00	76.45%	61.05%	0.00%

FUJITSU

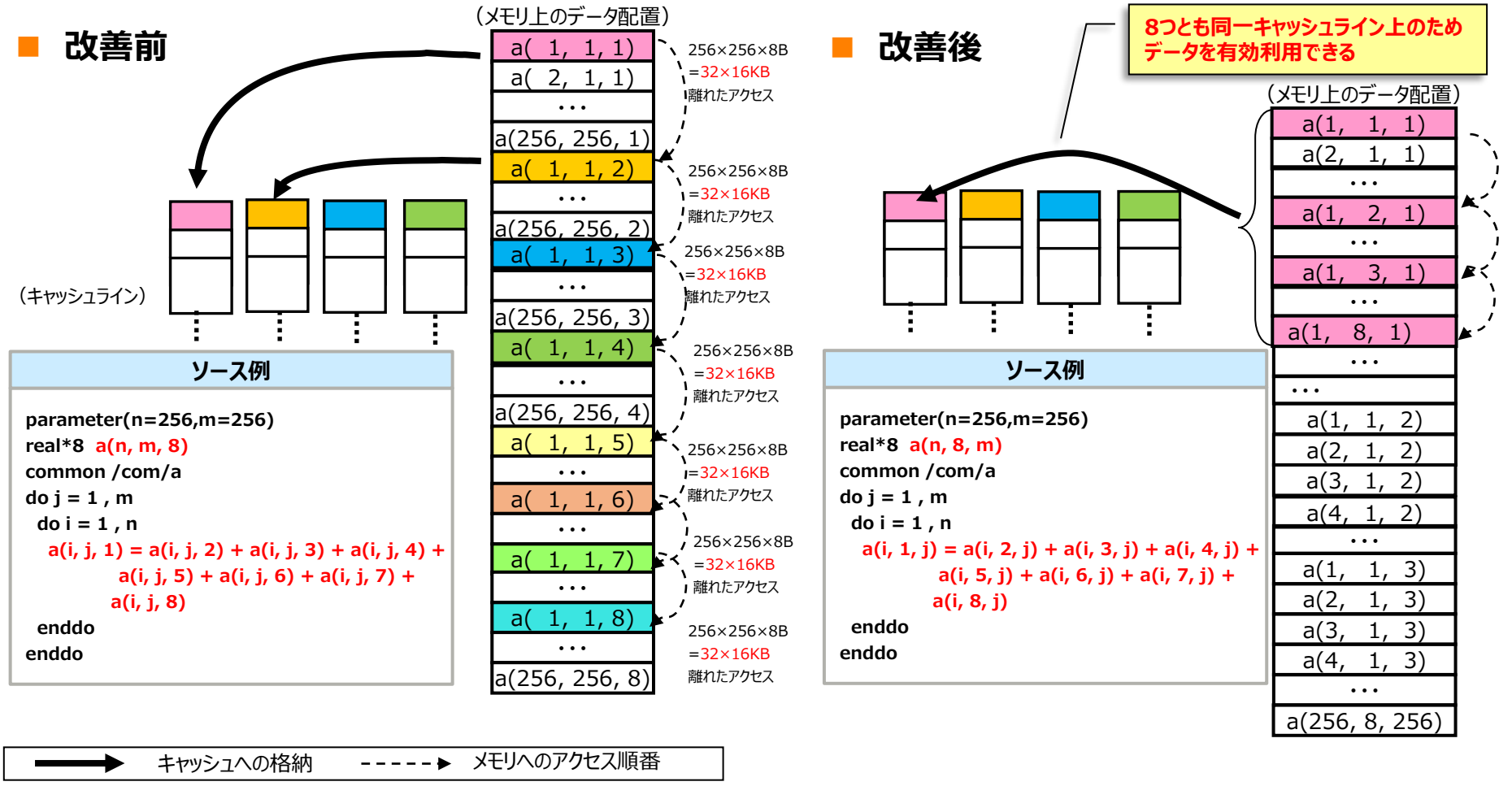
配列次元移動

- 配列次元移動とは
- 配列次元移動（改善前）
- 配列次元移動の効果（ソースチューニング）
- 配列次元移動の効果（翻訳オプションチューニング）

配列次元移動とは

配列次元移動とは、配列のアクセス次元を変更し、キャッシュの利用率を上げるチューニングです。

次の例のように、変化する次元を内側に移動することにより、キャッシュの利用効率を向上させることができます。



Fortran 配列次元移動（改善前）

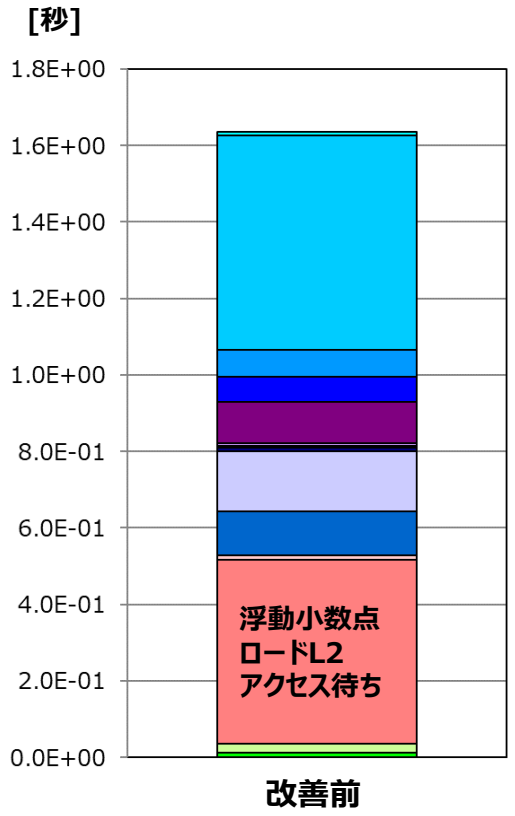
FUJITSU

配列aの最内ループでのアクセスはデータ局所性が低いため、浮動小数点ロードL2アクセス待ちが発生しています。

改善前ソース

```
16                    real(8)::a(N,M,8)
 :
               <<< Loop-information Start >>>
               <<<  [OPTIMIZATION]
               <<<    PREFETCH(HARD) Expect
               <<<      a
               <<< Loop-information  End >>>
21    2  p              do j=1,M
               <<< Loop-information Start >>>
               <<<  [OPTIMIZATION]
               <<<    SIMD(VL: 8)
               <<<    SOFTWARE PIPELINING(IPC: 2.87, ITR: 56,
                                                MVE: 2, POL: S)
               <<<    PREFETCH(HARD) Expected by compiler :
               <<<      a
               <<< Loop-information  End >>>
22    3  p  v            do i=1,N
23    3  p  v             a(i,j,1)=a(i,j,2)+a(i,j,3)+a(i,j,4)+&
24    3                        a(i,j,5)+a(i,j,6)+a(i,j,7)+a(i,j,8)
25    3  p  v            enddo
26    2  p              enddo
```

N=96
M=100



Cache

	L1I miss rate (/Effective instruction)	Load-store instruction	L1D miss	L1D miss rate (/Load-store instruction)	L1D miss demand rate (%) (/L1D miss)	L1D miss hardware prefetch rate (%) (/L1D miss)	L1D miss software prefetch rate (%) (/L1D miss)	L2 miss	L2 miss rate (/Load-store instruction)	L2 miss demand rate (%) (/L2 miss)	L2 miss hardware prefetch rate (%) (/L2 miss)	L2 miss software prefetch rate (%) (/L2 miss)
改善前	0.00	2.12E+10	8.20E+08	0.04	78.62%	21.38%	0.00%	5.45E+03	0.00	87.63%	20.43%	0.00%

Fortran 配列次元移動の効果（ソースチューニング）

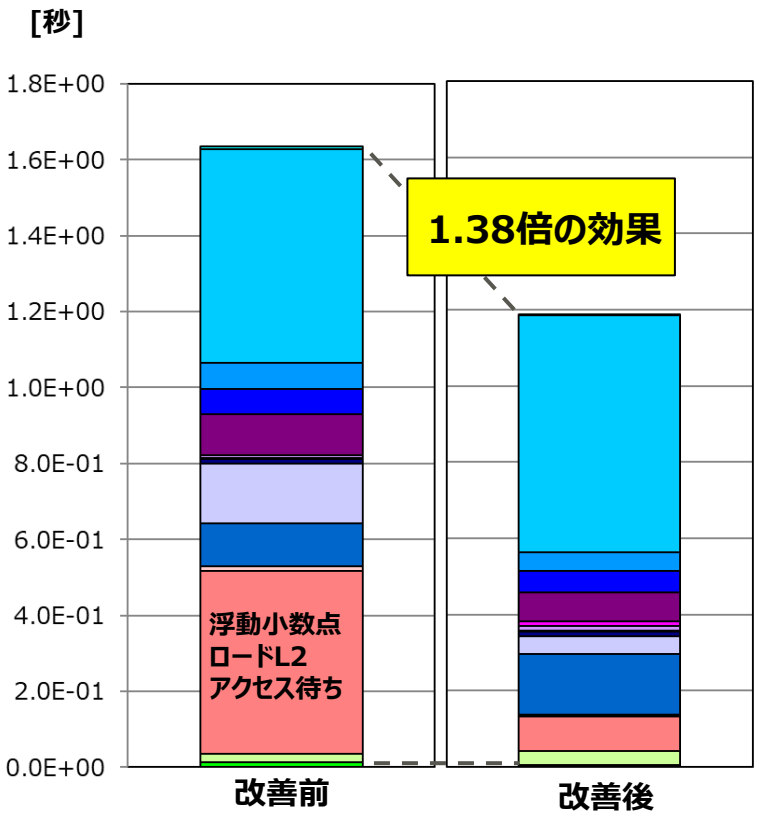
FUJITSU

配列次元移動を行うことでデータの局所性が向上され、浮動小数点L2アクセス待ちが改善されました。

改善後ソース（ソースチューニング）

```
16                    real(8)::a(N,8,M)
                  <<< Loop-information Start >>>
                  <<<  [OPTIMIZATION]
                  <<<    PREFETCH(HARD) Expect
                  <<<      a
                  <<< Loop-information  End >>>
21    2   p             do j=1,M
                  <<< Loop-information Start >>>
                  <<<  [OPTIMIZATION]
                  <<<    SIMD(VL: 8)
                  <<<    SOFTWARE PIPELINING(IPC: 2.87, ITR: 56,
                                                MVE: 2, POL: S)
                  <<<    PREFETCH(HARD) Expected by compiler :
                  <<<      a
                  <<< Loop-information  End >>>
22    3   p   v           do i=1,N
23    3   p   v             a(i,1,j)=a(i,2,j)+a(i,3,j)+a(i,4,j)+&
24    3                         a(i,5,j)+a(i,6,j)+a(i,7,j)+a(i,8,j)
25    3   p   v           enddo
26    2   p             enddo
```

N=96
M=100



Cache

	L1I miss rate (/Effective instruction)	Load-store instruction	L1D miss	L1D miss rate (/Load-store instruction)	L1D miss demand rate (%) (/L1D miss)	L1D miss hardware prefetch rate (%) (/L1D miss)	L1D miss software prefetch rate (%) (/L1D miss)	L2 miss	L2 miss rate (/Load-store instruction)	L2 miss demand rate (%) (/L2 miss)	L2 miss hardware prefetch rate (%) (/L2 miss)	L2 miss software prefetch rate (%) (/L2 miss)
改善前	0.00	2.12E+10	8.20E+08	0.04	78.62%	21.38%	0.00%	5.45E+03	0.00	87.63%	20.43%	0.00%
改善後	0.00	2.11E+10	7.39E+07	0.00	99.99%	0.01%	0.00%	4.12E+03	0.00	80.75%	32.43%	0.00%

C/C++ 配列次元移動（改善前）

FUJITSU

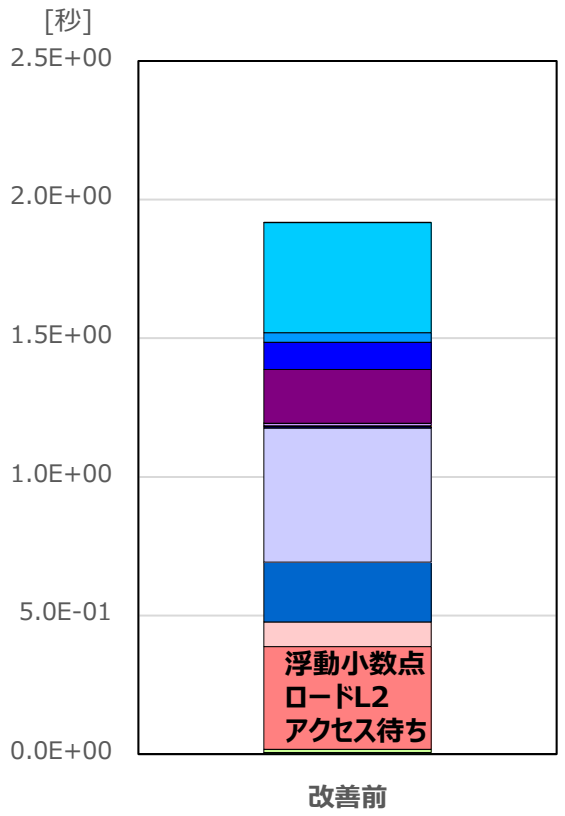
配列aの最内ループでのアクセスはデータ局所性が低いため、浮動小数点ロードL2アクセス待ちが発生しています。

改善前ソース

```
36          void sub(int N,int M,int ITER, double (* restrict a)[M][N])
37          {
38            int i,j,k;
39          #pragma omp parallel private(i,j,k)
40          {
41            for(k=0; k<ITER; k++)
42            {
43          #pragma omp for nowait
          <<< Loop-information Start >>>
 :
          <<< Loop-information  End >>>
44  p         for(j=0; j<M; j++)
45  p         {
          <<< Loop-information Start >>>
 :
          <<< Loop-information  End >>>
46  p   v       for(i=0; i<N; i++)
47  p   v       {
48  p   v         a[0][j][i]=a[1][j][i]+a[2][j][i]+a[3][j][i]+
49                      a[4][j][i]+a[5][j][i]+a[6][j][i]+a[7][j][i];
50  p   v       }
51  p         }
52          }
53          }
55            return;
56          }
```

N=96
M=100

配列宣言
double a[8][M][N];



Cache	L1I miss rate (/Effective instruction)	Load-store instruction	L1D miss	L1D miss rate (/Load-store instruction)	L1D miss demand rate (%) (/L1D miss)	L1D miss hardware prefetch rate (%) (/L1D miss)	L1D miss software prefetch rate (%) (/L1D miss)	L2 miss	L2 miss rate (/Load-store instruction)	L2 miss demand rate (%) (/L2 miss)	L2 miss hardware prefetch rate (%) (/L2 miss)	L2 miss software prefetch rate (%) (/L2 miss)
改善前	0.00	2.18E+10	8.20E+08	0.04	82.13%	17.87%	0.00%	1.49E+04	0.00	81.54%	38.68%	0.00%

C/C++ 配列次元移動の効果（ソースチューニング）

FUJITSU

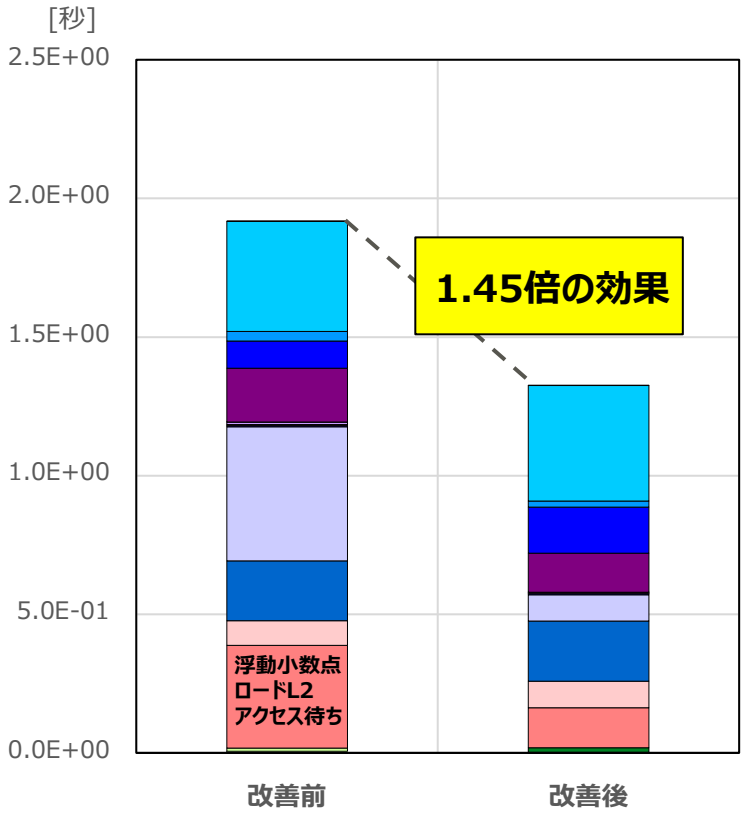
配列次元移動を行うことでデータの局所性が向上され、浮動小数点L2アクセス待ちが改善されました。

改善後ソース（ソースチューニング）

```
36          void sub(int N,int M,int ITER, double (* restrict a)[8][N]) {
38            int i,j,k;
39          #pragma omp parallel private(i,j,k)
40          {
41            for(k=0; k<ITER; k++) {
43          #pragma omp for nowait
          <<< Loop-information Start >>>
 :
          <<< Loop-information  End >>>
44  p         for(j=0; j<M; j++) {
          <<< Loop-information Start >>>
 :
          <<< Loop-information  End >>>
46  p   v     for(i=0; i<N; i++) {
48  p   v       a[j][0][i]=a[j][1][i]+a[j][2][i]+a[j][3][i]+
49                    a[j][4][i]+a[j][5][i]+a[j][6][i]+a[j][7][i];
50  p   v     }
51  p       }
52          }
53          }
55            return;
56          }
```

N=96
M=100

配列宣言
double a[8][M][N];



Cache	L1I miss rate (/Effective instruction)	Load-store instruction	L1D miss	L1D miss rate (/Load-store instruction)	L1D miss demand rate (%) (/L1D miss)	L1D miss hardware prefetch rate (%) (/L1D miss)	L1D miss software prefetch rate (%) (/L1D miss)	L2 miss	L2 miss rate (/Load-store instruction)	L2 miss demand rate (%) (/L2 miss)	L2 miss hardware prefetch rate (%) (/L2 miss)	L2 miss software prefetch rate (%) (/L2 miss)
改善前	0.00	2.18E+10	8.20E+08	0.04	82.13%	17.87%	0.00%	1.49E+04	0.00	81.54%	38.68%	0.00%
改善後	0.00	2.09E+10	8.63E+07	0.00	99.97%	0.02%	0.00%	1.56E+04	0.00	86.83%	39.15%	0.00%

Fortran 配列次元移動の効果（翻訳オプションチューニング）

FUJITSU

以下の翻訳オプション（Fortran固有）を指定することで、ソースチューニングと同等の効果を得ることができます。

翻訳オプション	機能説明
-Karray_subscript	４次元以上の割付け配列および最終次元の要素が１０以下で、最終次元以外の要素が１００ 以上の４次元以上の配列に対して配列の次元移動を行うことを指示します。 -Karray_subscript_element=100, -Karray_subscript_elementlast=10 および -Karray_subscript_rank=4 も同時に有効になります。
-Karray_subscript_element=N (2≦*N*≦2,147,483,647)	配列の次元移動の対象となる配列の最終次元以外の要素数が*N*以上であることを指示します。本オプションは、-Karray_subscript オプションが有効な場合に意味があります。ただし、割付け配列では、本オプションの意味はありません。
-Karray_subscript_elementlast=N (2≦*N*≦2,147,483,647)	配列の次元移動の対象となる配列の最終次元の要素数がN以下であることを指示します。本オプションは、-Karray_subscript オプションが有効な場合に意味があります。ただし、割付け配列では、本オプションの意味はありません。
-Karray_subscript_rank=*N* (2≦*N*≦30)	配列の次元移動の対象となる配列の次元数が*N*次元以上であることを指示します。本オプションは、-Karray_subscriptオプションが有効な場合に意味があります。

■ 使用例（改善前ソース時）

```
$frtpx –Kfast,parallel sample.f90
  –Karray_subscript,array_subscript_rank=2,array_subscript_element=2
```

■ 注意事項

- 対象配列を使うソース全てにオプション指定が必要です。
- 移動の効果はプログラムによって異なります。
- 正しくない使い方をした場合には計算結果が異なる場合があります。

FUJITSU

アンロールアンドジャム

- アンロールアンドジャムとは
- アンロールアンドジャム（改善前）
- アンロールアンドジャムの効果（最適化制御行チューニング）

アンロールアンドジャムとは (1/3)

FUJITSU

アンロールアンドジャムとは、多重ループの外側のループをアンローリングにより n 重に展開し、さらに展開された内側のループを融合するものです。

外側ループのアンローリング

最適化前ソース	最適化後ソース
!OCL UNROLL_AND_JAM_FORCE(2) DO J=1, 128 DO I=1, 128 A(I, J) = B(I, J) + B(I, J+1) … END DO END DO :	DO J=1, 128, 2 DO I=1, 128 A(I, J) = B(I, J) + B(I, J+1) A(I, J+1) = B(I, J+1) + B(I, J+2) … END DO END DO

内側ループの融合（共通式の除去）

アンロールアンドジャムを適用することで、共通式の除去が促進され、実行性能が向上する場合があります。
ただし、データストリーム数の増加やデータのアクセス順序の変化は、キャッシュの利用効率を低下させ、実行性能が低下する場合があります。

アンロールアンドジャムとは (2/3)

FUJITSU

以下の最適化制御行を指定します。

最適化指示子 (Fortran)	意味	指定可能な最適化制御行			
		プログラム単位	DOループ単位	文単位	配列代入文単位
UNROLL_AND_JAM[(*n*)]	最適化の効果があると判断したループに対してアンロールアンドジャムを有効にします。*n*は展開数(多重度)を表す2～100の10進数です。	○	○	×	×
UNROLL_AND_JAM_FORCE[(*n*)]	アンロールアンドジャムを有効にします。*n*は展開数(多重度)を表す2～100の10進数です。	×	○	×	×
NOUNROLL_AND_JAM	アンロールアンドジャムを無効にします。	○	○	×	×

最適化指示子 (C/C++)	意味	指定可能な最適化制御行			
		global行	procedure行	loop行	statement行
unroll_and_jam[(*n*)]	最適化の効果があると判断したループに対してアンロールアンドジャムを有効にします。*n*は展開数(多重度)を表す2～100の10進数です。	○	○	○	×
unrool_and_jam_force[(*n*)]	アンロールアンドジャムを有効にします。*n*は展開数(多重度)を表す2～100の10進数です。	×	×	○	×
nounroll_and_jam	アンロールアンドジャムを無効にします。	○	○	○	×

■ **注意事項**

- UNROLL_AND_JAM指示子は最適化の効果が期待できない場合や、回転を跨いだデータ依存がある場合は最適化を行いません。
- UNROLL_AND_JAM_FORCE指示子を誤って指定した（繰り返しを跨いだデータ依存がある）場合、実行結果は保証されません。
- 最内ループはアンロールアンドジャムの対象になりません。
- 最内ループの繰り返し回数が小さい場合、データストリーム数の増加やデータのアクセス順序の変化によっては、キャッシュの利用効率を低下させ、実行性能が低下する場合があります。
- キャッシュミスが増加する場合はプリフェッチで補正する良くなる場合があります。

アンロールアンドジャムとは（3/3）

以下の翻訳オプションを指定することで、最適化制御行チューニングと同等の効果を得ることができます。

翻訳オプション	機能説明
-K{ unroll_and_jam[=N] | nounroll_and_jam } 2 ≦ *N* ≦ 100	アンロールアンドジャムの最適化を行うかどうかを指示します。*N*にはループ展開数の上限を2～100の範囲で指定できます。*N*の指定を省略した場合、コンパイラが自動的に最良な値を決定します。デフォルトは-Knounroll_and_jamです。 アンロールアンドジャムを適用することで、共通式の除去が促進され、実行性能が向上する場合があります。ただし、データストリーム数の増加やデータのアクセス順序の変化は、キャッシュの利用効率を低下させ、実行性能が低下する場合があります。 また、アンロールアンドジャムの最適化による実行性能への影響はループ単位で異なります。このため、本最適化は、-Kunroll_and_jam[=*N*]オプションでプログラム全体へ適用せず、最適化指示子UNROLL_AND_JAMや最適化指示子UNROLL_AND_JAM_FORCEでループ単位に適用することを推奨します。

■使用例（改善前ソース時）

```
$ frtpx -Kfast,parallel sample.f90 -Kunroll_and_jam
$ fccpx -Kfast,parallel sample.c -Kunroll_and_jam
```

◆注意事項

clangモードではアンロールアンドジャムの最適化機能を使用できません

Fortran アンロールアンドジャム（改善前）

FUJITSU

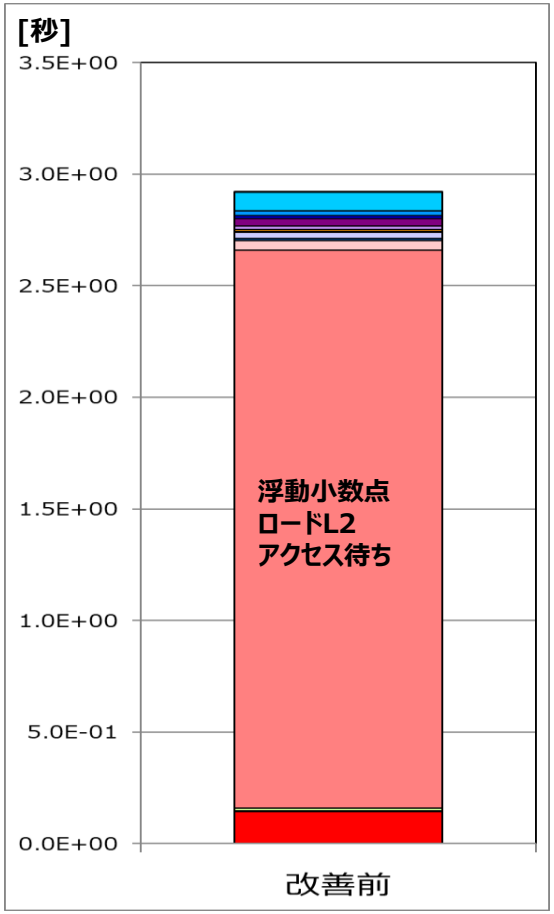
L1Dミス率が高いことからキャッシュの利用効率が悪く、浮動小数点ロードL2アクセス待ちが多くなっています。

改善前ソース

```
11            DATA_TYPE,dimension(IMAX,JMAX,KMAX)::a
12            DATA_TYPE,dimension(JMAX,KMAX)::b
13            DATA_TYPE,dimension(JMAX,IMAX)::c
 :
15 1  pp        do k=1, KMAX
16 1
17 2   p         do j=1, JMAX - 3
          <<< Loop-information Start >>>
          <<<  [OPTIMIZATION]
          <<<    SIMD(VL: 8)
          <<<    SOFTWARE PIPELINING(IPC: 1.85, ITR: 80, MVE: 2, POL: S)
          <<<    PREFETCH(HARD) Expected by compiler :
          <<<      a
          <<< Loop-information  End >>>
18 3   p   2v        do i =1, IMAX
19 3   p   2v          a(i,j,k) = a(i,j+1,k) + a(i,j+2,k) + a(i,j+3,k) &
20 3                       + (b(j+2,k) / c(j,i))
21 3   p   2v        end do
22 2   p         end do
23 1   p       end do
```

```
#define DATA_TYPE real(kind=8)
#define IMAX 512
#define JMAX 512
#define KMAX 128
```

配列cはストライドアクセスのため、キャッシュの利用効率が悪い



Cache	L1I miss rate (/Effective instruction)	Load-store instruction	L1D miss	L1D miss rate (/Load-store instruction)	L1D miss demand rate (%) (/L1D miss)	L1D miss hardware prefetch rate (%) (/L1D miss)	L1D miss software prefetch rate (%) (/L1D miss)	L2 miss	L2 miss rate (/Load-store instruction)	L2 miss demand rate (%) (/L2 miss)
改善前	0.00	3.39E+09	3.82E+09	1.13	99.50%	0.50%	0.00%	1.15E+08	0.03	46.67%

Fortran アンロールアンドジャムの効果（最適化制御行チューニング）

FUJITSU

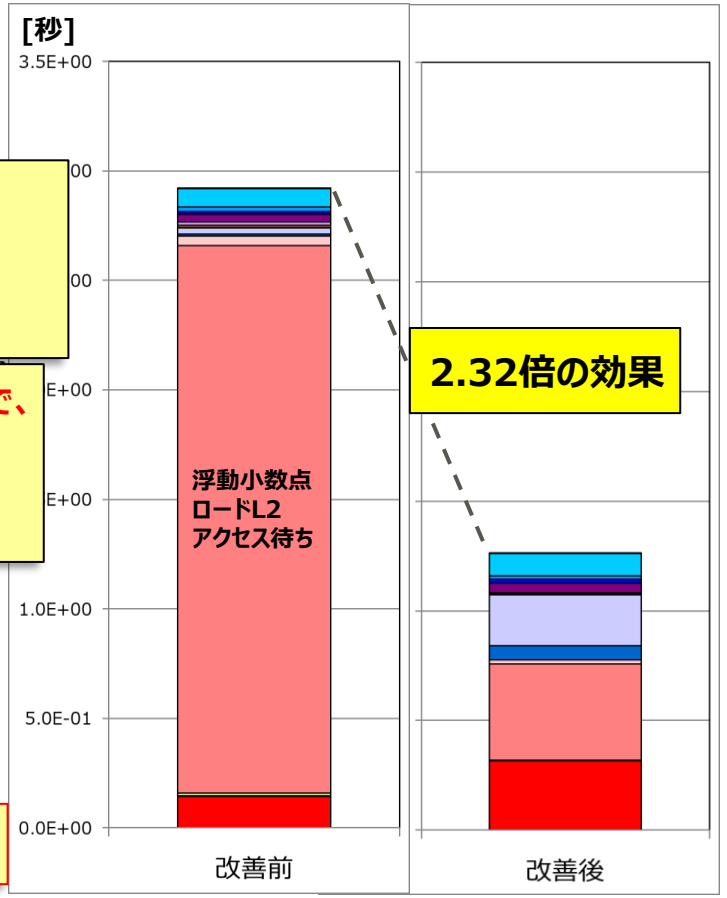
キャッシュの利用効率の向上により、L1Dミス数が減少して浮動小数点ロードL2アクセス待ちが改善されました。

改善後ソース（最適化制御行チューニング）

```
11                DATA_TYPE,dimension(IMAX,JMAX,KMAX)::a
12                DATA_TYPE,dimension(JMAX,KMAX)::b
13                DATA_TYPE,dimension(JMAX,IMAX)::c
 :
15    1  pp          do k=1, KMAX
16    1           !ocl unroll_and_jam_force(8)
17    2  p           do j=1, JMAX - 3
                <<< Loop-information Start >>>
                <<<  [OPTIMIZATION]
                <<<    SIMD(VL: 8)
                <<<    SOFTWARE PIPELINING
                <<<    PREFETCH(HARD) Expe
                <<<      a
                <<<    PREFETCH(SOFT) : 32
                <<<     SEQUENTIAL : 32
                <<<      a: 32
                <<<    SPILLS :
                <<<      GENERAL   : SPILL 0  FILL 4
                <<<      SIMD&FP   : SPILL 0  FILL 0
                <<<      SCALABLE  : SPILL 0  FILL 0
                <<<      PREDICATE : SPILL 0  FILL 0
                <<< Loop-information  End >>>
18    3  p  2v          do i =1, IMAX
19    3  p  2v            a(i,j,k) = a(i,j+1,k) + a(i,j+2,k) + a(i,j+3,k) &
20    3                        + (b(j+2,k) / c(j,i))
21    3  p  2v          end do
22    2  p            end do
23    1  p          end do
```

```
#define DATA_TYPE
real(kind=8)
#define IMAX 512
#define JMAX 512
#define KMAX 128
```

外側ループでアンロールすることで、配列aの共通式の除去による命令の削減のほか、配列aおよび配列cのキャッシュ利用効率が向上した



L1Dミスが大幅に減少

Cache	L1I miss rate (/Effective instruction)	Load-store instruction	L1D miss	L1D miss rate (/Load-store instruction)	L1D miss demand rate (%) (/L1D miss)	L1D miss hardware prefetch rate (%) (/L1D miss)	L1D miss software prefetch rate (%) (/L1D miss)	L2 miss	L2 miss rate (/Load-store instruction)	L2 miss demand rate (%) (/L2 miss)
改善前	0.00	3.39E+09	3.82E+09	1.13	99.50%	0.50%	0.00%	1.15E+08	0.03	46.67%
改善後	0.00	2.68E+09	7.37E+08	0.27	89.34%	2.55%	8.12%	1.15E+08	0.04	44.44%

C/C++ アンロールアンドジャム（改善前）

FUJITSU

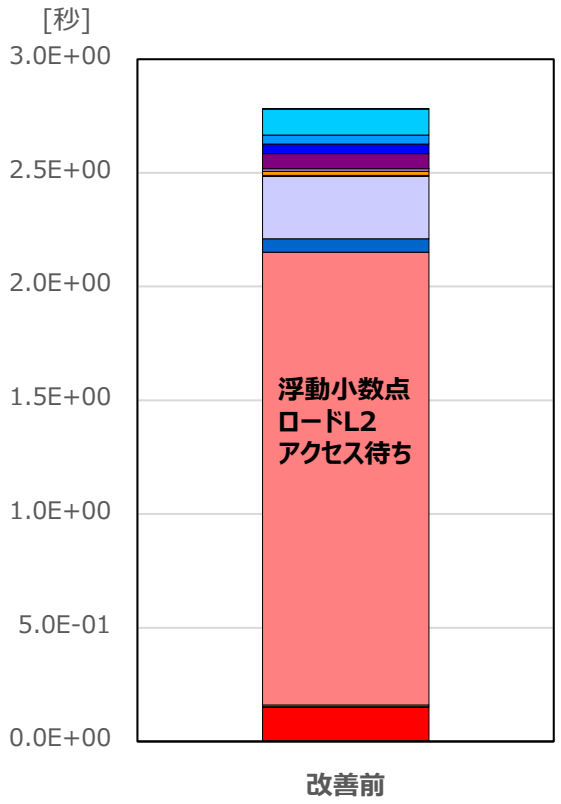
L1Dミス率が高いことからキャッシュの利用効率が悪く、浮動小数点ロードL2アクセス待ちが多くなっています。

改善前ソース

```
69          #pragma omp parallel for
70   p        for (k=0;k<KMAX;k++){
          <<< Loop-information Start >>>
          <<<  [OPTIMIZATION]
          <<<    PREFETCH(HARD) Expected by comp
          <<<      (unknown)
          <<< Loop-information  End >>>
71   p          for (j=0;j<JMAX-3;j++){
          <<< Loop-information Start >>>
          <<<  [OPTIMIZATION]
          <<<    SIMD(VL: 8)
          <<<    SOFTWARE PIPELINING(IPC: 2.09, ITR: 80, MVE: 2, POL: S)
          <<<    PREFETCH(HARD) Expected by compiler :
          <<<      (unknown)
          <<< Loop-information  End >>>
72   p    2v       for (i=0;i<IMAX;i++){
73   p    2v        a[k][j][i] = a[k][j+1][i] + a[k][j+2][i] + a[k][j+3][i]
74                          + (b[k][j+2] / c[i][j]);
75   p    2v       }
76   p           }
77   p         }
```

```
#define IMAX 512
#define JMAX 512
#define KMAX 128

配列宣言
double
a[KMAX][JMAX][IMAX],
b[KMAX][JMAX],
c[IMAX][JMAX];
```

配列cはストライドアクセスのため、キャッシュの利用効率が悪い



Cache	L1I miss rate (/Effective instruction)	Load-store instruction	L1D miss	L1D miss rate (/Load-store instruction)	L1D miss demand rate (%) (/L1D miss)	L1D miss hardware prefetch rate (%) (/L1D miss)	L1D miss software prefetch rate (%) (/L1D miss)	L2 miss	L2 miss rate (/Load-store instruction)	L2 miss demand rate (%) (/L2 miss)
改善前	0.00	3.50E+09	3.68E+09	1.05	99.73%	0.26%	0.00%	1.16E+08	0.03	59.11%

C/C++ アンロールアンドジャムの効果（最適化制御行チューニング）

FUJITSU

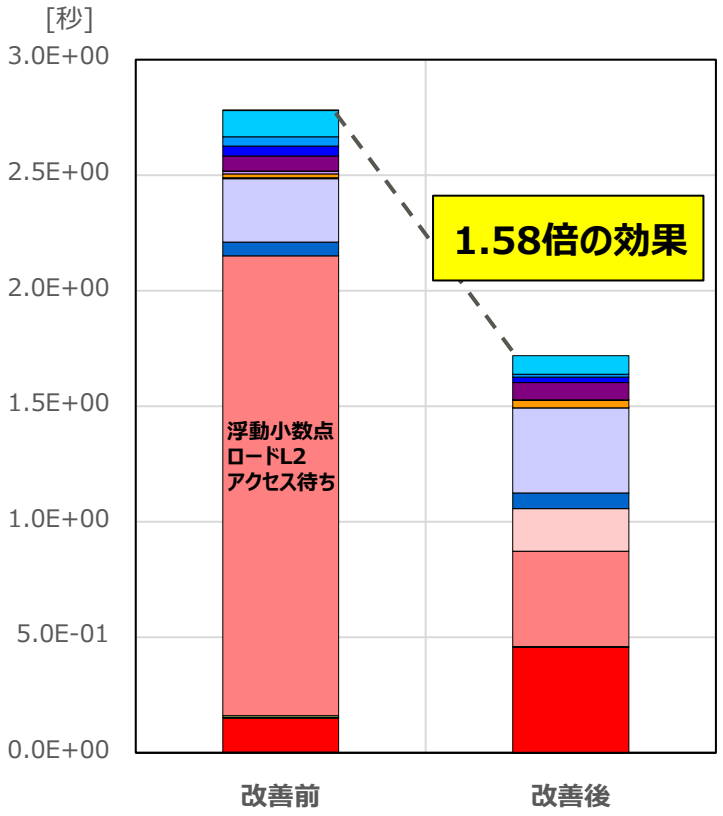
キャッシュの利用効率の向上により、L1Dミス数が減少して浮動小数点ロードL2アクセス待ちが改善されました。

改善後ソース（最適化制御行チューニング）

```
74            for (k=0;k<KMAX;k++){
75          #pragma loop unroll_and_jam_force 8
          <<< Loop-information Start >>>
          <<<  [OPTIMIZATION]
          <<<    PREFETCH(HARD) Expected by com
          <<<      (unknown)
          <<< Loop-information  End >>>
76              for (j=0;j<JMAX-3;j++){
          <<< Loop-information Start >>>
          <<<  [OPTIMIZATION]
          <<<    SOFTWARE PIPELINING(IPC: 0.53,
          <<<    PREFETCH(HARD) Expected by compiler :
          <<<      (unknown)
          <<<    PREFETCH(SOFT) : 32
          <<<     SEQUENTIAL : 32
          <<<      (unknown): 32
          <<<    SPILLS :
          <<<      GENERAL   : SPILL 0
          <<<      SIMD&FP   : SPILL 0
          <<<      SCALABLE  : SPILL 0  FILL 0
          <<<      PREDICATE : SPILL 0  FILL 0
          <<< Loop-information  End >>>
77        2v        for (i=0;i<IMAX;i++){
78        2v          a[k][j][i] = a[k][j+1][i] + a[k][j+2][i] + a[k][j+3][i]
79                              + (b[k][j+2] / c[i][j]);
80        2v        }
81              }
82            }
```

```
#define IMAX 512
#define JMAX 512
#define KMAX 128

配列宣言
double
a[KMAX][JMAX][IMAX],
b[KMAX][JMAX],
c[IMAX][JMAX];
```

外側ループでアンロールすることで、配列aの共通式の除去による命令の削減のほか、配列aおよび配列cのキャッシュ利用効率が向上した



L1Dミスが大幅に減少

Cache	L1I miss rate (/Effective instruction)	Load-store instruction	L1D miss	L1D miss rate (/Load-store instruction)	L1D miss demand rate (%) (/L1D miss)	L1D miss hardware prefetch rate (%) (/L1D miss)	L1D miss software prefetch rate (%) (/L1D miss)	L2 miss	L2 miss rate (/Load-store instruction)	L2 miss demand rate (%) (/L2 miss)
改善前	0.00	3.50E+09	3.68E+09	1.05	99.73%	0.26%	0.00%	1.16E+08	0.03	59.11%
改善後	0.00	2.42E+09	6.24E+08	0.26	82.30%	3.11%	14.60%	1.15E+08	0.05	39.17%

FUJITSU

データアクセス待ち（レイテンシの隠蔽）

- インダイレクトアクセスプリフェッチ
- 連続アクセスでないデータへのソフトウェアプリフェッチの利用

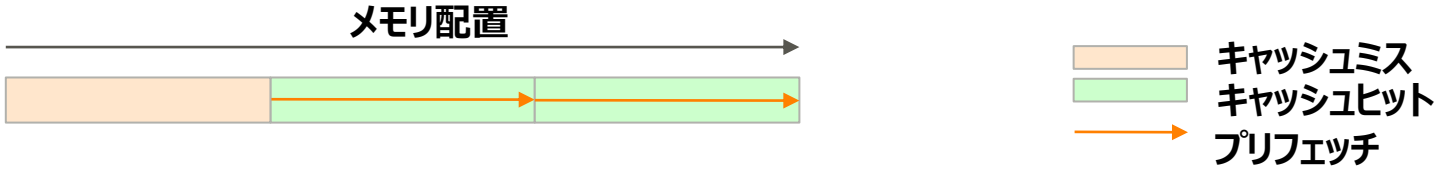
FUJITSU

インダイレクトアクセスプリフェッチ

- プリフェッチとは
- インダイレクトアクセスプリフェッチ（最適化制御行）
- インダイレクトアクセスプリフェッチの効果（翻訳オプションチューニング）
- インダイレクトアクセスプリフェッチ（改善前）
- インダイレクトアクセスプリフェッチの効果（最適化制御行チューニング）

プリフェッチとは

プリフェッチとは、実行される命令によってデータが必要になる前に、キャッシュにデータをロードしておくことで、パフォーマンスを向上させる仕組みです。

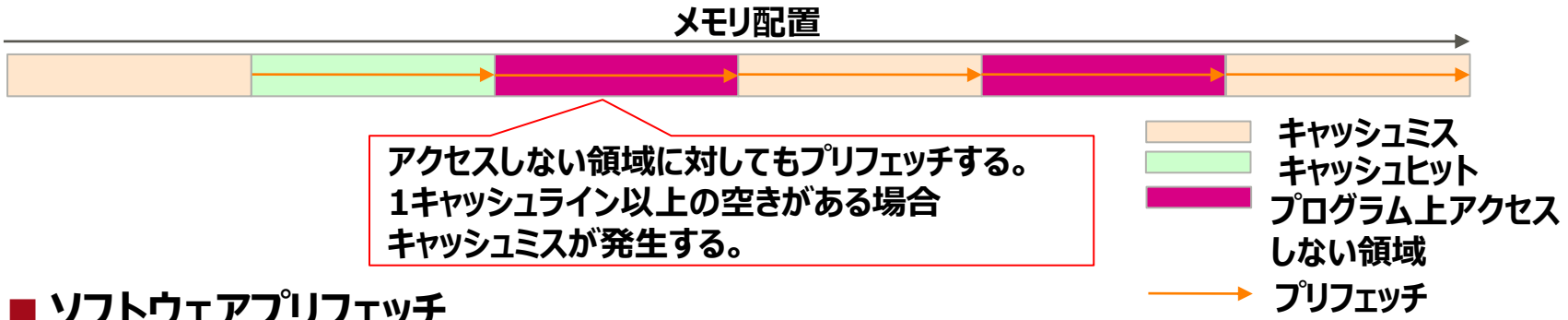


プリフェッチにはハードウェアプリフェッチとソフトウェアプリフェッチがあります。

■ ハードウェアプリフェッチ

プログラムのメモリアクセスの規則性からハードウェアがデータアクセスを予測しプリフェッチします。

プログラム上アクセスしない領域がある場合にもプリフェッチしてしまうため、そのようなプログラムではキャッシュ効率が非常に悪くなります。その場合はソフトウェアプリフェッチを利用します。



■ ソフトウェアプリフェッチ

ソフトウェア（コンパイラ）がプログラムを解析し、prefetch命令を生成することでプリフェッチします。

インダイレクトアクセスプリフェッチ（最適化制御行）

FUJITSU

以下の最適化制御行を指定します。

最適化指示子 (Fortran)	意味	指定可能な最適化制御行			
		プログラム単位	DOループ単位	文単位	配列代入文単位
PREFETCH	コンパイラの自動 prefetch 機能を有効にします。	○	○	×	×

最適化指示子 (C/C++)	意味	指定可能な最適化制御行			
		global行	procedure行	loop行	statement行
prefetch	コンパイラの自動 prefetch 機能を有効にします。	○	○	○	×

◆補足

prefetch最適化指示子は、以下の翻訳オプションを指定したものと等価です。

-Kprefetch_sequential,prefetch_stride,prefetch_indirect,prefetch_conditional,
prefetch_cache_level=all

◆注意事項

翻訳オプション-Kprefetch_sequential、-Kprefetch_stride、
-Kprefetch_indirectまたは-Kprefetch_conditionalが有効な場合のプリフェッチでは、ループのキャッシュ効率、分岐の有無または添字の複雑さによって、実行性能が低下することがあります。

インダイレクトアクセスプリフェッチの効果（翻訳オプションチューニング）

FUJITSU

以下の翻訳オプションを指定することで、最適化制御行チューニングと同等の効果を得ることができます。

翻訳オプション	機能説明
-Kprefetch_indirect	ループ内で使用される間接的にアクセス（リストアクセス）される配列データに対して、prefetch命令を使用したオブジェクトを生成するかどうかを指示します。 本オプションは、-O1 オプション以上が有効な場合に意味があります。 デフォルトは、-Kprefetch_noindirect です。

■使用例（改善前ソース時）

```
$ frtpx –Kfast,parallel sample.f90 -Kprefetch_indirect
$ fccpx –Kfast,parallel sample.c -Kprefetch_indirect
```

◆注意事項

プリフェッチが実施されますが、ループのキャッシュ効率やIF構文の有無、
添字の複雑さによっては、意図している効果が得られない場合があります。
clangモードではインダイレクトプリフェッチの最適化機能を使用できません

Fortran インダイレクトアクセスプリフェッチ（改善前）

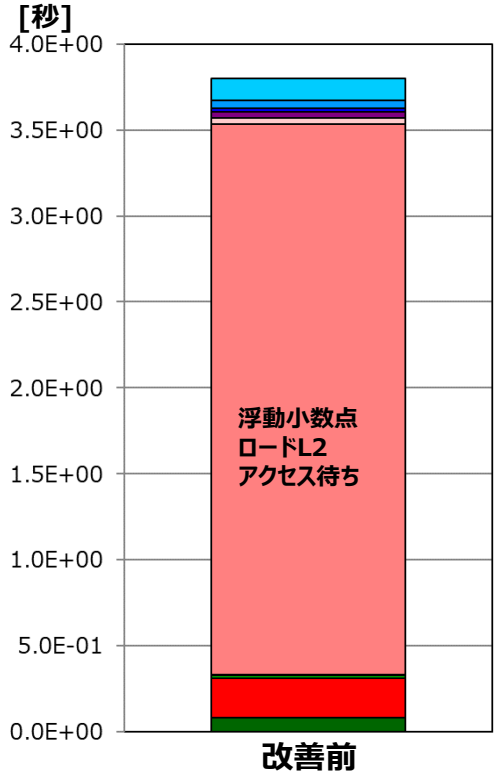
FUJITSU

インダイレクトアクセス（リストアクセス）の場合、推奨オプションではプリフェッチが生成されないためメモリアクセスのレイテンシが見えています。
そのため、浮動小数点ロードL2アクセス待ちが多くなっています。

改善前ソース

```
        <<< Loop-information Start >>>
        <<<  [PARALLELIZATION]
        <<<    Standard iteration count: 572
        <<<  [OPTIMIZATION]
        <<<    SOFTWARE PIPELINING(IPC: 3.50, ITR: 18, MVE: 3, POL: S)
        <<<    PREFETCH(HARD) Expected by compiler :
        <<<      e, d, a
        <<< Loop-information  End >>>
52      1  pp   2            do i = 1 , n
53      1   p   2              a(i) = b(d(i)) + scalar * c(e(i))
54      1   p   2            enddo
```

配列b,cが
インダイレクトアクセス



L1Dミスｄｍ率および、L2ミスdm率が高くプリフェッチが効いていない。
メモリスループットには余裕があるため性能向上の可能性がある。

	Memory throughput (GB/s)
改善前	32.67

Cache	L1I miss rate (/Effective instruction)	Load-store instruction	L1D miss	L1D miss rate (/Load-store instruction)	L1D miss demand rate (%) (/L1D miss)	L1D miss hardware prefetch rate (%) (/L1D miss)	L1D miss software prefetch rate (%) (/L1D miss)	L2 miss	L2 miss rate (/Load-store instruction)	L2 miss demand rate (%) (/L2 miss)	L2 miss hardware prefetch rate (%) (/L2 miss)	L2 miss software prefetch rate (%) (/L2 miss)
改善前	0.00	9.01E+09	3.77E+09	0.42	98.23%	1.77%	0.00%	4.29E+08	0.05	54.19%	75.86%	0.00%

Fortran インダイレクトアクセスプリフェッチの効果（最適化制御行チューニング）

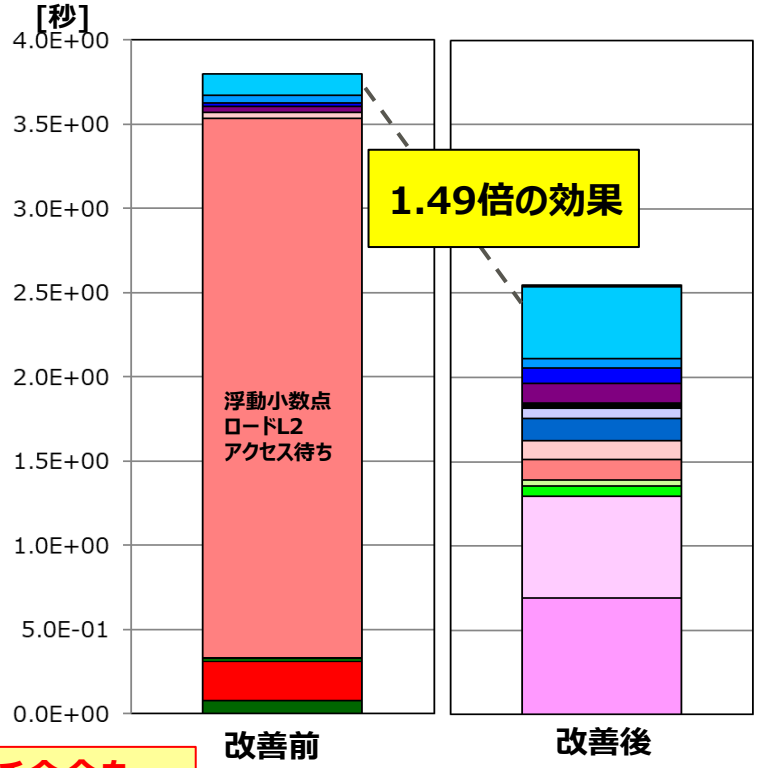
FUJITSU

prefetch指示子を指定することでインダイレクトアクセス（リストアクセス）に対してプリフェッチを生成します。その結果、浮動小数点ロードメモリL2アクセス待ちが改善されました。

改善後ソース

```
51                !ocl prefetch
                  <<< Loop-information Start >>>
                  <<<  [PARALLELIZATION]
                  <<<    Standard iteration count: 572
                  <<<  [OPTIMIZATION]
                  <<<    PREFETCH(HARD) Ex
                  <<<      e, d, a
                  <<<    PREFETCH(SOFT) : 8
                  <<<     INDIRECT : 8
                  <<<      c: 4, b: 4
                  <<< Loop-information  End >>>
52     1  pp   2            do i = 1 , n
53     1   p   2              a(i) = b(d(i)) + scalar * c(e(i))
54     1   p   2            enddo
```

インダイレクトアクセス（配列b, c）に対するプリフェッチが生成された



インダイレクトアクセス(配列b, c)に対してプリフェッチ命令を生成することでL1Dミスdm率、L2 ミスdm率が減った。

	Memory throuput (GB/s)
改善前	32.67
改善後	183.71

Cache	L1I miss rate (/Effective instruction)	Load-store instruction	L1D miss	L1D miss rate (/Load-store instruction)	L1D miss demand rate (%) (/L1D miss)	L1D miss hardware prefetch rate (%) (/L1D miss)	L1D miss software prefetch rate (%) (/L1D miss)	L2 miss	L2 miss rate (/Load-store instruction)	L2 miss demand rate (%) (/L2 miss)	L2 miss hardware prefetch rate (%) (/L2 miss)	L2 miss software prefetch rate (%) (/L2 miss)
改善前	0.00	9.01E+09	3.77E+09	0.42	98.23%	1.77%	0.00%	4.29E+08	0.05	54.19%	75.86%	0.00%
改善後	0.00	1.66E+10	3.82E+09	0.23	2.94%	2.94%	94.12%	1.77E+09	0.11	2.02%	6.20%	91.78%

C/C++ インダイレクトアクセスプリフェッチ（改善前）

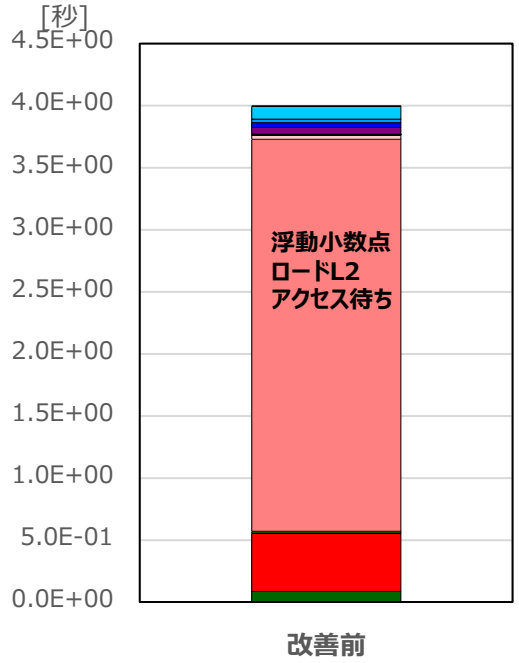
FUJITSU

インダイレクトアクセス（リストアクセス）の場合、推奨オプションではプリフェッチが生成されないためメモリアクセスのレイテンシが見えています。
そのため、浮動小数点ロードL2アクセス待ちが多くなっています。

改善前ソース

```
53          void sub(double * restrict a, double * restrict b,
                     double * restrict c, int * restrict d, int * restrict e,
                     double scalar, int n){
54            int i;
56          #pragma omp parallel for
          <<< Loop-information Start >>>
          <<<  [OPTIMIZATION]
          <<<    SOFTWARE PIPELINING(IPC: 2.83, ITR: 12, MVE: 2, POL: S)
          <<<    PREFETCH(HARD) Expected by compiler :
          <<<      (unknown)
          <<< Loop-information  End >>>
57   p    2     for (i = 0; i < n; i++){
58   p    2       a[i] = b[d[i]] + scalar * c[e[i]];
59   p    2     }
60          }
```

配列b,cが
インダイレクトアクセス



L1Dミスｄｍ率および、L2ミスdm率が高くプリフェッチが効いていない。
メモリスループットには余裕があるため性能向上の可能性がある。

Statistics	Memory throughput (GB/s)
改善前	31.24

Cache	L1I miss rate (/Effective instruction)	Load-store instruction	L1D miss	L1D miss rate (/Load-store instruction)	L1D miss demand rate (%) (/L1D miss)	L1D miss hardware prefetch rate (%) (/L1D miss)	L1D miss software prefetch rate (%) (/L1D miss)	L2 miss	L2 miss rate (/Load-store instruction)	L2 miss demand rate (%) (/L2 miss)	L2 miss hardware prefetch rate (%) (/L2 miss)	L2 miss software prefetch rate (%) (/L2 miss)
改善前	0.00	8.65E+09	3.78E+09	0.44	98.01%	1.99%	0.00%	4.30E+08	0.05	47.96%	100.00%	0.00%

C/C++ インダイレクトアクセスプリフェッチの効果（最適化制御行チューニング）

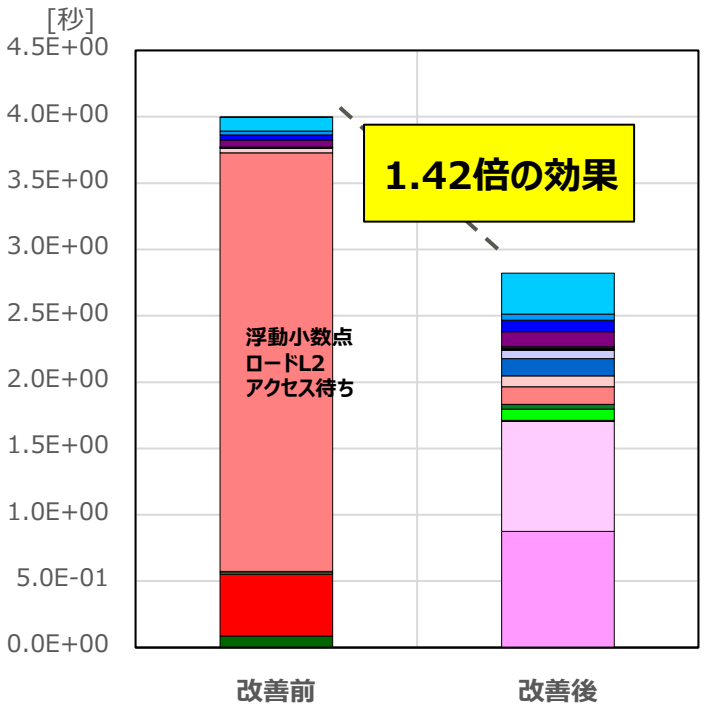
FUJITSU

prefetch指示子を指定することでインダイレクトアクセス（リストアクセス）に対してプリフェッチを生成します。その結果、浮動小数点ロードメモリL2アクセス待ちが改善されました。

改善後ソース

```
53          void sub(double * restrict a, double * restrict b,
                     double * restrict c, int * restrict d,
                     int * restrict e, double scalar, int n){
54            int i;
56          #pragma loop prefetch
57          #pragma omp parallel for
          <<< Loop-information Start >>>
          <<<  [OPTIMIZATION]
          <<<    PREFETCH(HARD) Expected by compiler :
          <<<      (unknown)
          <<<    PREFETCH(SOFT) : 8
          <<<     INDIRECT : 8
          <<<      (unknown): 8
          <<< Loop-information  End >>>
58  p    2     for (i = 0; i < n; i++){
59  p    2       a[i] = b[d[i]] + scalar * c[e[i]];
60  p    2     }
61          }
```

インダイレクトアクセス（配列b, c）に対するプリフェッチが生成された



インダイレクトアクセス(配列b, c)に対してプリフェッチ命令を生成することでL1Dミスdm率、L2 ミスdm率が減った。

Statistics	Memory throughput (GB/s)
改善前	31.24
改善後	159.90

Cache	L1I miss rate (/Effective instruction)	Load-store instruction	L1D miss	L1D miss rate (/Load-store instruction)	L1D miss demand rate (%) (/L1D miss)	L1D miss hardware prefetch rate (%) (/L1D miss)	L1D miss software prefetch rate (%) (/L1D miss)	L2 miss	L2 miss rate (/Load-store instruction)	L2 miss demand rate (%) (/L2 miss)	L2 miss hardware prefetch rate (%) (/L2 miss)	L2 miss software prefetch rate (%) (/L2 miss)
改善前	0.00	8.65E+09	3.78E+09	0.44	98.01%	1.99%	0.00%	4.30E+08	0.05	47.96%	100.00%	0.00%
改善後	0.00	1.70E+10	3.83E+09	0.22	2.99%	2.94%	94.08%	1.70E+09	0.10	1.80%	6.41%	91.79%

連続アクセスでないデータへのソフトウェアプリフェッチの利用

- 連続アクセスでないデータへのソフトウェアプリフェッチの利用
- 連続アクセスでないデータへのソフトウェアプリフェッチの利用（改善前）
- 連続アクセスでないデータへのソフトウェアプリフェッチの利用①（最適化制御行チューニング）
- 連続アクセスでないデータへのソフトウェアプリフェッチの利用②[推奨]（最適化制御行チューニング）

連続アクセスでないデータへのソフトウェアプリフェッチの利用

FUJITSU

非連続または連続な部分の短いデータに対するアクセスでは、ハードウェアプリフェッチはキャッシュミスが発生しやすくなります。その場合は、ソフトウェアプリフェッチを利用することで性能が向上します。

ソフトウェアプリフェッチを利用するためには、後述のPREFETCH_READ指示子, PREFETCH_WRITE指示子を使用するか、下記の翻訳時オプションを利用します。

翻訳時オプション	機能説明
-Kprefetch_sequential=auto	ループ内で使用される連続的にアクセスされる配列データに対して、ハードウェアプリフェッチを利用するか、prefetch命令を出力するかをコンパイラが自動的に選択します。本オプションは、-O1オプション以上が有効な場合に意味があります。-O2オプション以上が有効な場合、デフォルトは-Kprefetch_sequential=autoです。
-Kprefetch_sequential=soft	ループ内で使用される連続的にアクセスされる配列データに対して、ハードウェアプリフェッチを利用せずに、prefetch命令を出力します。本オプションは、-O1オプション以上が有効な場合に意味があります。
-Kprefetch_nosequential	ループ内で使用される連続的にアクセスされる配列データに対して、prefetch命令を使用せずにオブジェクトを生成します。-O0または-O1オプションが有効な場合、デフォルトは -Kprefetch_nosequentialです。

連続アクセスでないデータへのソフトウェアプリフェッチの利用

FUJITSU

PREFETCH_READ指示子, PREFETCH_WRITE指示子は次のように指定します。

最適化指示子	意味	指定可能な最適化制御行			
		プログラム単位	DOループ単位	文単位	配列代入文単位
PREFETCH_READ(name[,level={1|2}][,strong={0|1}])	参照されているデータに対してprefetch命令を生成することを指示します。 nameは配列要素名,levelはプリフェッチを行うキャッシュレベル、strongはstrong prefetchをするかどうかです。	○	×	○	×
PREFETCH_WRITE(name[,level={1|2}][,strong={0|1}])	定義されているデータに対してprefetch命令を生成することを指示します。	○	×	○	×

改善前

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

改善後①（ソフトウェアプリフェッチ要素指定）

```
do j=1,n
   do i=1,isize
!OCL PREFETCH_WRITE(a(i,j+1),level=1)
!OCL PREFETCH_READ(b(i,j+1),level=1)
!OCL PREFETCH_READ(c(i,j+1),level=1)
      a(i,j) =  b(i,j) + scalar * c(i,j)
   enddo
enddo
```

改善後②（ソフトウェアプリフェッチベクトル指定） 推奨

```
do j=1,n
!OCL PREFETCH_WRITE(a(1:isize,j+1),level=1)
!OCL PREFETCH_READ(b(1:isize,j+1),level=1)
!OCL PREFETCH_READ(c(1:isize,j+1),level=1)
   do i=1,isize
      a(i,j) =  b(i,j) + scalar * c(i,j)
   enddo
enddo
```

配列要素をベクトル指定します。
プリフェッチ命令を複数同時に生成できるため要素ごとの指定よりも性能が向上します。

C/C++ 連続アクセスでないデータへのビルトインプリフェッチの利用

FUJITSU

ビルトインプリフェッチ関数は次のように使用します。仕様についてはGNU C、C++コンパイラのホームページをご参照ください。

改善前

```
for (j = 0; j < n; j++) {
  for (i = 0; i < isize; i++) {
    a[j][i] = b[j][i] + scalar * c[j][i];
  }
}
```

改善後①（ビルトインプリフェッチ要素指定）

```
for (j = 0; j < n; j++) {
  for (i = 0; i < isize; i++) {
    __builtin_prefetch(&a[j+1][0], 1, 3);
    __builtin_prefetch(&b[j+1][0], 0, 3);
    __builtin_prefetch(&c[j+1][0], 0, 3);
    __builtin_prefetch(&a[j+1][32], 1, 3);
        :
    a[j][i] = b[j][i] + scalar * c[j][i];
  }
}
```

プリフェッチ回数が少なく性能が向上します

改善後②（ビルトインプリフェッチベクトル指定） 推奨

```
for (j = 0; j < n; j++) {
  __builtin_prefetch(&a[j+1][0], 1, 3);
  __builtin_prefetch(&a[j+1][32], 1, 3);
  __builtin_prefetch(&a[j+1][64], 1, 3);
  __builtin_prefetch(&b[j+1][0], 0, 3);
  __builtin_prefetch(&b[j+1][32], 0, 3);
  __builtin_prefetch(&b[j+1][64], 0, 3);
  __builtin_prefetch(&c[j+1][0], 0, 3);
      :
  for (i = 0; i < isize; i++) {
    a[j][i] = b[j][i] + scalar * c[j][i];
  }
}
```

Fortran 連続アクセスでないデータへのソフトウェアプリフェッチの利用（改善前）

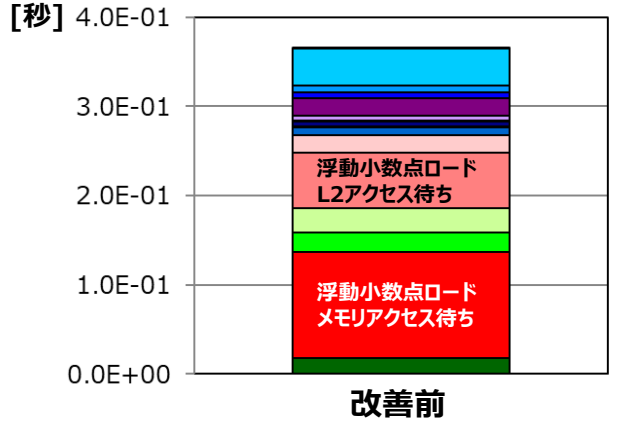
FUJITSU

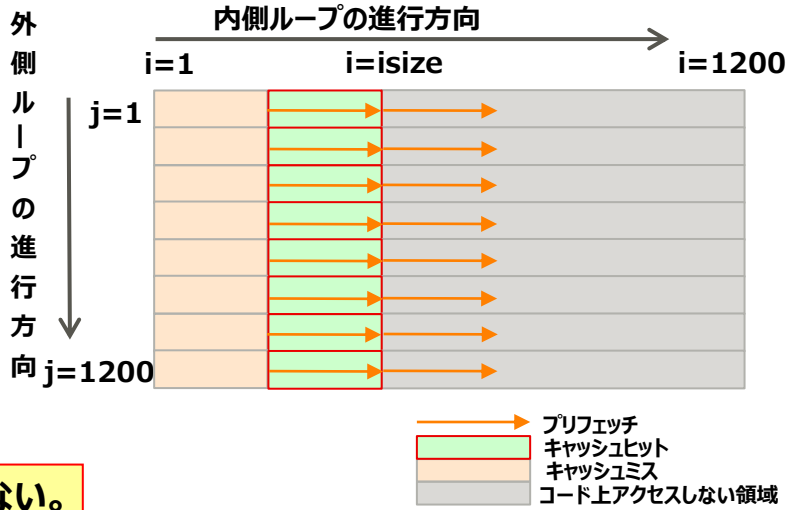
最内ループは、繰返し数が少なく、かつ繰返し数よりも配列サイズが大きいため、連続的ではありません。通常のプリフェッチではプリフェッチの立ち上がりのコストがみえてしまいます。そのため、浮動小数点ロードアクセス待ちが多くなっています。

改善前ソース

```
42                parameter(n=1200)
43                integer n
44                real*8  a(n,n),b(n,n),c(n,n),scalar
45                common /com/a,b,c
46
47                !$omp parallel do
                  <<< Loop-information S
                  <<<  [OPTIMIZATION]
                  <<<    PREFETCH(HARD
                  <<<      c, b, a
                  <<< Loop-information
48    1   p           do j=1,n
                  <<< Loop-information Start >>>
                  <<<  [OPTIMIZATION]
                  <<<    SIMD(VL: 8)
                  <<<    SOFTWARE PIPELINING(IPC: 3.40, ITR: 128, MVE: 3, POL: S)
                  <<<    PREFETCH(HARD) Expected by compiler :
                  <<<      c, b, a
                  <<< Loop-information  End >>>
49    2   p   2v          do i=1,isize
50    2   p   2v            a(i,j) =  b(i,j) + scalar * c(i,j)
51    2   p   2v          enddo
52    1   p           enddo
```

配列の１次元目の要素数　：　1200
ループ繰返し数(isize)　：　128
外側ループjがインクリメントされた時にアクセスの連続性が途切れる





L1Dミスdm率が高くプリフェッチが効いていない。

Cache	L1I miss rate (/Effective instruction)	Load-store instruction	L1D miss	L1D miss rate (/Load-store instruction)	L1D miss demand rate (%) (/L1D miss)	L1D miss hardware prefetch rate (%) (/L1D miss)	L1D miss software prefetch rate (%) (/L1D miss)	L2 miss	L2 miss rate (/Load-store instruction)	L2 miss demand rate (%) (/L2 miss)	L2 miss hardware prefetch rate (%) (/L2 miss)	L2 miss software prefetch rate (%) (/L2 miss)
改善前	0.00	1.75E+09	2.64E+08	0.15	70.94%	29.46%	-0.40%	1.26E+08	0.07	39.28%	83.10%	0.00%

Fortran 連続アクセスでないデータへのソフトウェアプリフェッチの利用①（最適化制御行チューニング）

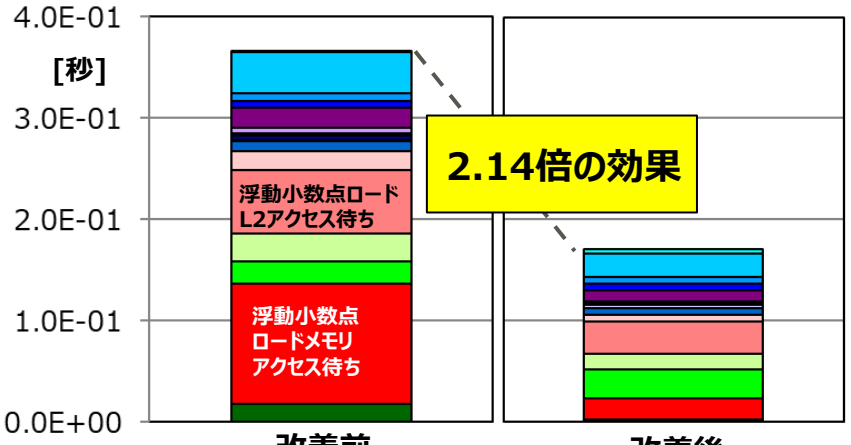
FUJITSU

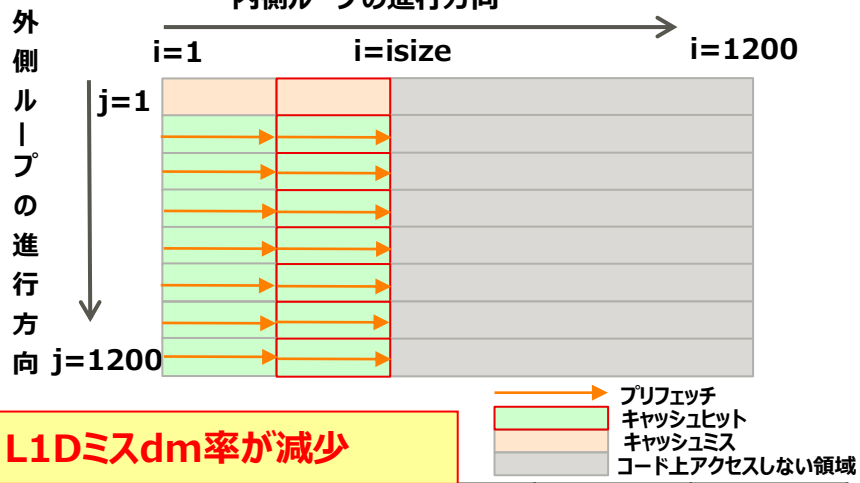
PREFETCH_READ, PREFETCH_WRITE指示子を使用することで、外側ループの配列に対してプリフェッチを生成しプリフェッチの立ち上がりのコストを隠蔽します。その結果、浮動小数点ロードL2アクセス待ちが改善されました。

改善後ソース

```
42                parameter(n=1200)
43                integer n
44                real*8  a(n,n),b(n,n),c(n,n),scalar
45                common /com/a,b,c
46
47              !$omp parallel do
              <<< Loop-information Start >>>
              <<<  [OPTIMIZATION]
              <<<    PREFETCH(HARD) Expected by compiler :
              <<<      c, b, a
              <<<    PREFETCH(SOFT) : 18
              <<<     SPECIFIED : 18
              <<<      a: 6, c: 6, b: 6
              <<< Loop-information  End >>>
48    1  p          do j=1,n
              <<< Loop-information Start >>>
              <<<  [OPTIMIZATION]
              <<<    SIMD(VL: 8)
              <<<    SOFTWARE PIPELINING(IPC: 2.25, ITR: 88, MVE: 6, POL: S)
              <<<    PREFETCH(HARD) Expected by compiler :
              <<<      c, b, a
              <<<    PREFETCH(SOFT) : 18
              <<<     SPECIFIED : 18
              <<<      c: 6, b: 6, a: 6
              <<< Loop-information  End >>>
49    2  p  v          do i=1,isize
50    2  p           !OCL PREFETCH_WRITE(a(i,j+1),level=1)
51    2  p           !OCL PREFETCH_READ(b(i,j+1),level=1)
52    2  p           !OCL PREFETCH_READ(c(i,j+1),level=1)
53    2  p  v            a(i,j) =  b(i,j) + scalar * c(i,j)
54    2  p  v          enddo
55    1  p          enddo
```

外側ループ１回転先の配列に対してプリフェッチ





L1Dミスdm率が減少

Cache	Load-store instruction	L1D miss	L1D miss rate (/Load-store instruction)	L1D miss demand rate (%) (/L1D miss)	L1D miss hardware prefetch rate (%) (/L1D miss)	L1D miss software prefetch rate (%) (/L1D miss)
改善前	1.75E+09	2.64E+08	0.15	70.94%	29.46%	-0.40%
改善後	1.13E+09	1.90E+08	0.17	4.57%	0.44%	94.99%

Fortran 連続アクセスでないデータへのソフトウェアプリフェッチの利用②[推奨]（最適化制御行チューニング）

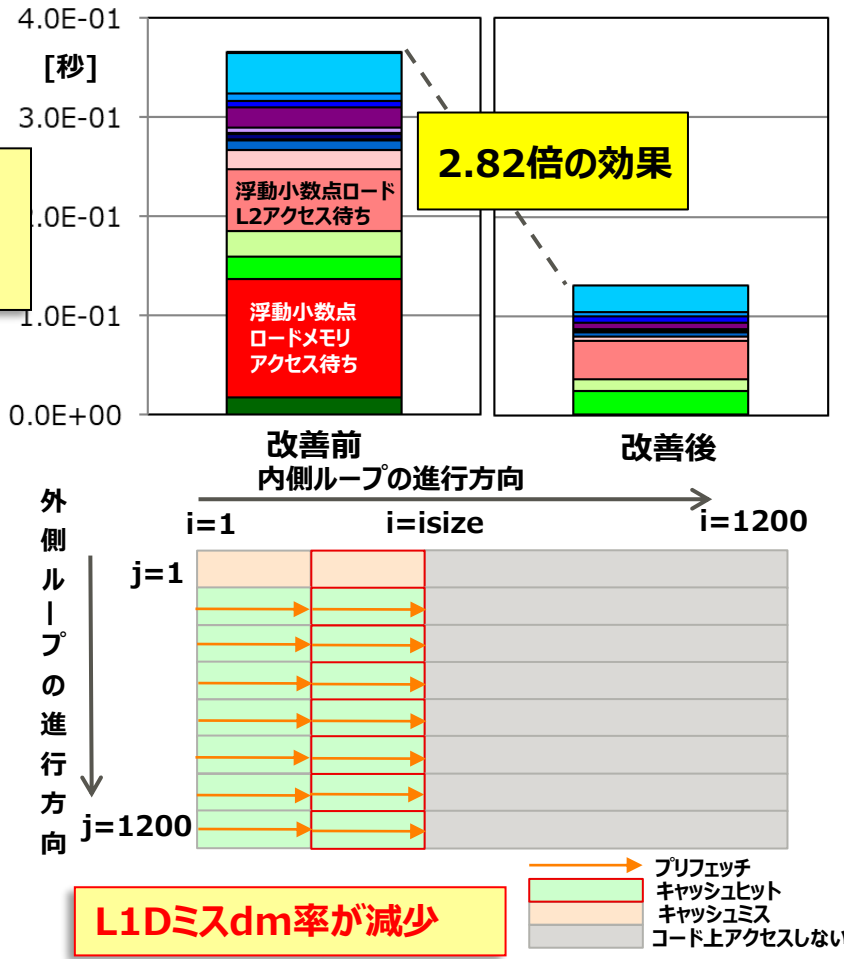
FUJITSU

外側のループで配列に対するPREFETCH_READ, PREFETCH_WRITE指示子を指定することで、最内ループでの命令数を減らすことができ、さらに性能を向上させることができます。

改善後ソース

```
                <<< Loop-information Start >>>
                <<<  [OPTIMIZATION]
                <<<    PREFETCH(HARD) Expected by compiler :
                <<<      c, b, a
                <<<    PREFETCH(SOFT) : 3
                <<<     SPECIFIED : 3
                <<<      a: 1, c: 1, b: 1
                <<< Loop-information  End >>>
48    1   p           do j=1,n
                <<< Loop-information Start >>>
                <<<  [OPTIMIZATION]
                <<<    PREFETCH(SOFT) : 4
                <<<     SPECIFIED : 4
                <<<      a: 4
                <<< Loop-information  End >>>
49    1   p   4s    !OCL PREFETCH_WRITE(a(1:isize,j+1),level=1)
                <<< Loop-information Start >>>
                <<<  [OPTIMIZATION]
                <<<    PREFETCH(SOFT) : 4
                <<<     SPECIFIED : 4
                <<<      b: 4
                <<< Loop-information  End >>>
50    1   p   4s    !OCL PREFETCH_READ(b(1:isize,j+1),level=1)
                <<< Loop-information Start >>>
                <<<  [OPTIMIZATION]
                <<<    PREFETCH(SOFT) : 4
                <<<     SPECIFIED : 4
                <<<      c: 4
                <<< Loop-information  End >>>
51    1   p   4s    !OCL PREFETCH_READ(c(1:isize,j+1),level=1)
                <<< Loop-information Start >>>
                <<<  [OPTIMIZATION]
                <<<    SIMD(VL: 8)
                <<<    SOFTWARE PIPELINING(IPC: 3.40, ITR: 128, MVE: 3, POL: S)
                <<<    PREFETCH(HARD) Expected by compiler :
                <<<      c, b, a
                <<< Loop-information  End >>>
52    2   p   2v          do i=1,isize
53    2   p   2v            a(i,j) =  b(i,j) + scalar * c(i,j)
54    2   p   2v          enddo
55    1   p           enddo
```

外側ループ１回転先の配列に対して配列全体プリフェッチ



L1Dミスdm率が減少

Cache	Load-store instruction	L1D miss	L1D miss rate (/Load-store instruction)	L1D miss demand rate (%) (/L1Dmiss)	L1D miss hardware prefetch rate (%) (/L1Dmiss)	L1D miss software prefetch rate (%) (/L1Dmiss)
改善前	1.75E+09	2.64E+08	0.15	70.94%	29.46%	-0.40%
改善後	9.22E+08	1.93E+08	0.21	23.46%	1.65%	74.90%

C/C++ 連続アクセスでないデータへのソフトウェアプリフェッチの利用（改善前）

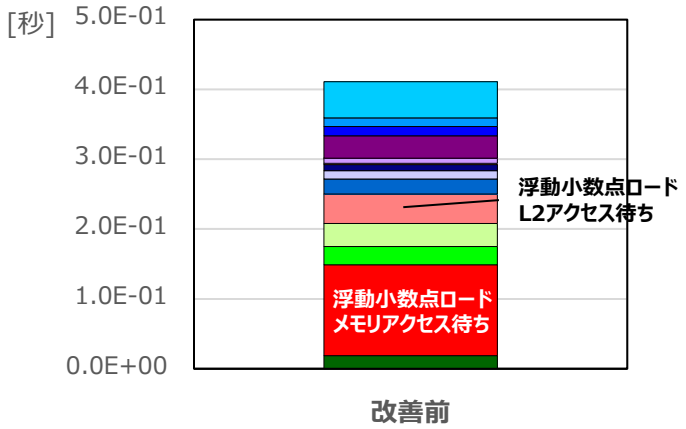
FUJITSU

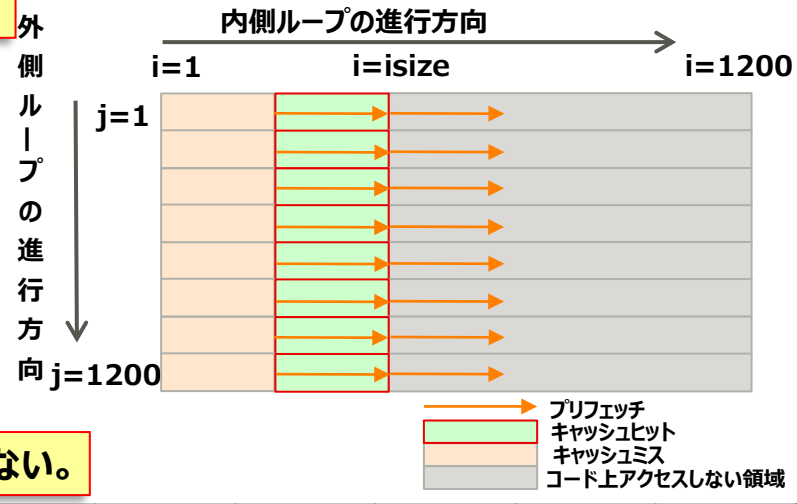
最内ループは、繰返し数が少なく、かつ繰返し数よりも配列サイズが大きいため、連続的ではありません。通常のプリフェッチではプリフェッチの立ち上がりのコストがみえてしまいます。そのため、浮動小数点ロードアクセス待ちが多くなっています。

改善前ソース

```
40          void sub(double scalar, int isize){
41           int i, j;
42
43          #pragma omp parallel for
          <<< Loop-information Start
          <<<  [OPTIMIZATION]
          <<<    PREFETCH(HARD) Ex
          <<<      c, b, a
          <<< Loop-information  End
44   p        for (j = 0; j < n; j++){
          <<< Loop-information Start >>>
          <<<  [OPTIMIZATION]
          <<<    SIMD(VL: 8)
          <<<    SOFTWARE PIPELINING(IPC: 3.25, ITR: 144, MVE: 4, POL: S)
          <<<    PREFETCH(HARD) Expected by compiler :
          <<<      c, b, a
          <<< Loop-information  End >>>
45   p    2v     for (i = 0; i < isize; i++){
46   p    2v       a[j][i] =  b[j][i] + scalar * c[j][i];
47   p    2v     }
48   p        }
49          }
```

配列の2次元目の要素数　：　1200
ループ繰返し数(isize)　：　128
外側ループjがインクリメントされた時にアクセスの連続性が途切れる





L1Dミスdm率が高くプリフェッチが効いていない。

Cache	L1I miss rate (/Effective instruction)	Load-store instruction	L1D miss	L1D miss rate (/Load-store instruction)	L1D miss demand rate (%) (/L1D miss)	L1D miss hardware prefetch rate (%) (/L1D miss)	L1D miss software prefetch rate (%) (/L1D miss)	L2 miss	L2 miss rate (/Load-store instruction)	L2 miss demand rate (%) (/L2 miss)	L2 miss hardware prefetch rate (%) (/L2 miss)	L2 miss software prefetch rate (%) (/L2 miss)
改善前	0.00	2.18E+09	2.57E+08	0.12	73.77%	26.27%	-0.04%	6.67E+07	0.03	42.40%	82.13%	0.00%

C/C++ 連続アクセスでないデータへのソフトウェアプリフェッチの利用①（最適化制御行チューニング）

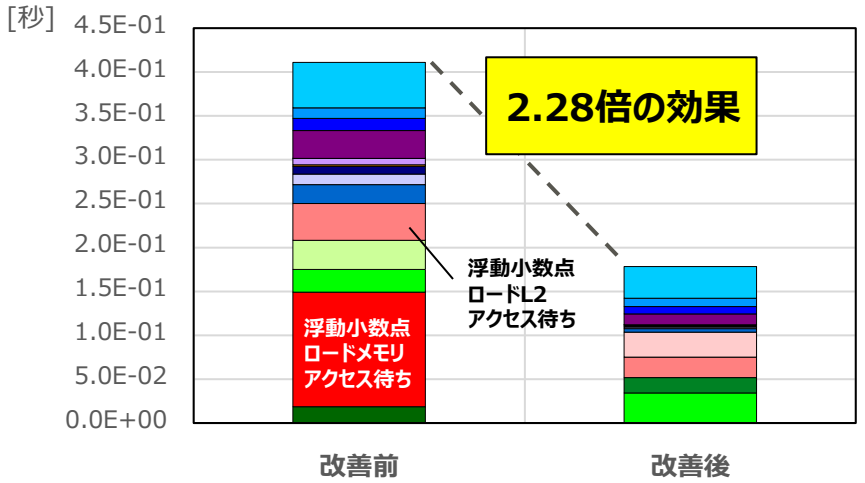
FUJITSU

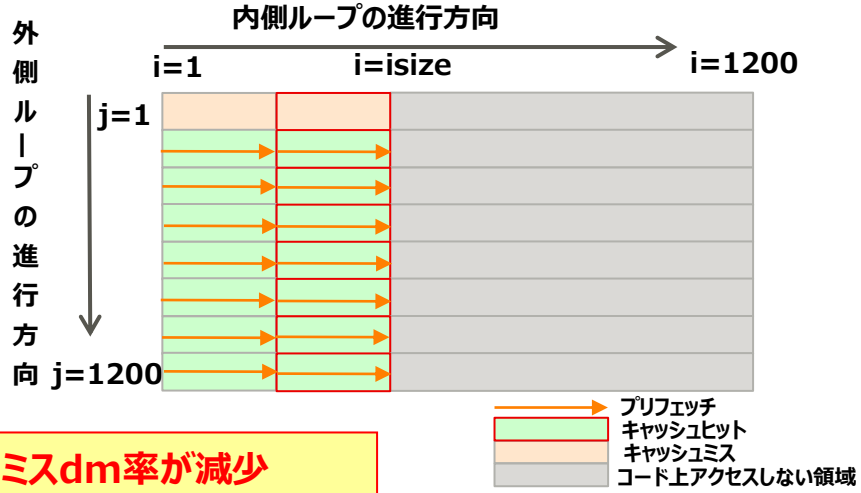
__builtin_prefetch関数を使用することで、外側ループの配列に対してプリフェッチを生成しプリフェッチの立ち上がりのコストを隠蔽します。その結果、浮動小数点ロードL2アクセス待ちが改善されました。

改善後ソース

```
41          void sub(double scalar, int isize){
42            int i, j;
43
44          #pragma omp parallel for
          <<< Loop-information Start >>>
          <<<  [OPTIMIZATION]
          <<<    PREFETCH(HARD) Expected by compiler :
          <<<      c, b, a
          <<<    PREFETCH(SOFT) : 48
          <<<     SPECIFIED : 48
          <<<      a: 16, c: 16, b: 16
          <<< Loop-information  End >>>
45  p         for (j = 0; j < n; j++){
          <<< Loop-information Start >>>
          <<<  [OPTIMIZATION]
          <<<    SIMD(VL: 8)
          <<<    SOFTWARE PIPELINING(IPC: 2.62, ITR: 56, MVE: 4, POL: S)
          <<<    PREFETCH(HARD) Expected by compiler :
          <<<      c, a, b
          <<<    PREFETCH(SOFT) : 48
          <<<     SPECIFIED : 48
          <<<      b: 16, c: 16, a: 16
          <<< Loop-information  End >>>
46  p     v       for(i = 0; i < isize; i++){
47  p     v         __builtin_prefetch(&a[j+1][0], 1, 3);
48  p     v         __builtin_prefetch(&b[j+1][0], 0, 3);
49  p     v         __builtin_prefetch(&c[j+1][0], 0, 3);
50  p     v         __builtin_prefetch(&a[j+1][32], 1, 3);
51  p     v         __builtin_prefetch(&b[j+1][32], 0, 3);
52  p     v         __builtin_prefetch(&c[j+1][32], 0, 3);
53  p     v         __builtin_prefetch(&a[j+1][64], 1, 3);
54  p     v         __builtin_prefetch(&b[j+1][64], 0, 3);
55  p     v         __builtin_prefetch(&c[j+1][64], 0, 3);
56  p     v         __builtin_prefetch(&a[j+1][96], 1, 3);
57  p     v         __builtin_prefetch(&b[j+1][96], 0, 3);
58  p     v         __builtin_prefetch(&c[j+1][96], 0, 3);
59  p     v         a[j][i] =  b[j][i] + scalar * c[j][i];
60  p     v       }
61  p           }
62            }
```

外側ループ１回転先の配列に対してプリフェッチ





L1Dミスdm率が減少

Cache	Load-store instruction	L1D miss	L1D miss rate (/Load-store instruction)	L1D miss demand rate (%) (/L1D miss)	L1D miss hardware prefetch rate (%) (/L1D miss)	L1D miss software prefetch rate (%) (/L1D miss)
改善前	2.18E+09	2.57E+08	0.12	73.77%	26.27%	-0.04%
改善後	1.16E+09	1.93E+08	0.17	14.76%	1.20%	84.04%

C/C++ 連続アクセスでないデータへのソフトウェアプリフェッチの利用②[推奨]（最適化制御行チューニング）

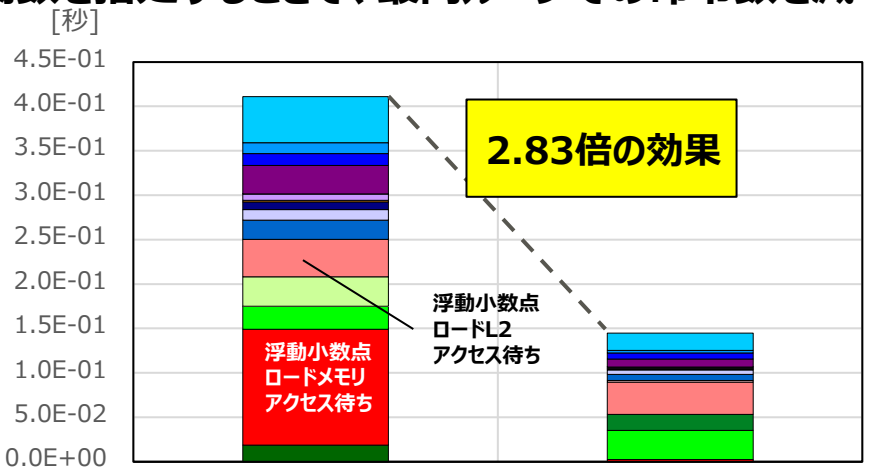
FUJITSU

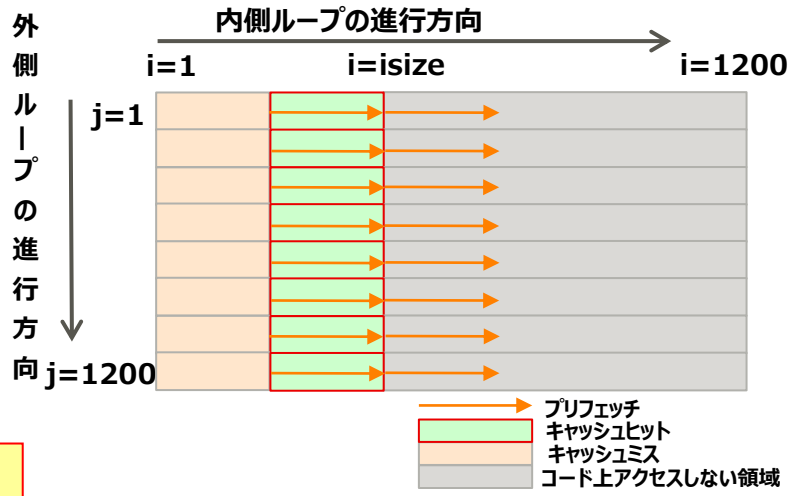
外側のループで配列に対する__builtin_prefetch関数を指定することで、最内ループでの命令数を減らすことができ、さらに性能を向上させることができます。

改善後ソース

```
41          void sub(double scalar, int isize){
42            int i, j;
43
44          #pragma omp parallel for
            <<< Loop-information Sta
            <<<  [OPTIMIZATION]
            <<<    PREFETCH(HARD)
            <<<      c, b, a
            <<<    PREFETCH(SOFT) : 12
            <<<     SPECIFIED : 12
            <<<      a: 4, b: 4, c: 4
            <<< Loop-information  End >>>
45  p         for (j = 0; j < n; j++){
46  p           __builtin_prefetch(&a[j+1][0], 1, 3);
47  p           __builtin_prefetch(&a[j+1][32], 1, 3);
48  p           __builtin_prefetch(&a[j+1][64], 1, 3);
49  p           __builtin_prefetch(&a[j+1][96], 1, 3);
50  p           __builtin_prefetch(&b[j+1][0], 0, 3);
51  p           __builtin_prefetch(&b[j+1][32], 0, 3);
52  p           __builtin_prefetch(&b[j+1][64], 0, 3);
53  p           __builtin_prefetch(&b[j+1][96], 0, 3);
54  p           __builtin_prefetch(&c[j+1][0], 0, 3);
55  p           __builtin_prefetch(&c[j+1][32], 0, 3);
56  p           __builtin_prefetch(&c[j+1][64], 0, 3);
57  p           __builtin_prefetch(&c[j+1][96], 0, 3);
            <<< Loop-information Start >>>
            <<<  [OPTIMIZATION]
            <<<    SIMD(VL: 8)
            <<<    SOFTWARE PIPELINING(IPC: 3.25, ITR: 14
            <<<    PREFETCH(HARD) Expected by compiler :
            <<<      c, b, a
            <<< Loop-information  End >>>
58  p   2v      for (i = 0; i < isize; i++){
59  p   2v        a[j][i] = b[j][i] + scalar * c[j][i];
60  p   2v      }
61  p         }
62          }
```

外側ループ１回転先の配列に対して配列全体プリフェッチ





L1Dミスdm率が減少

Cache	Load-store instruction	L1D miss	L1D miss rate (/Load-store instruction)	L1D miss demand rate (%) (/L1D miss)	L1D miss hardware prefetch rate (%) (/L1D miss)	L1D miss software prefetch rate (%) (/L1D miss)
改善前	2.18E+09	2.57E+08	0.12	73.77%	26.27%	-0.04%
改善後	1.17E+09	1.94E+08	0.17	23.67%	1.43%	74.90%

データアクセス待ちの改善（アクセス量の軽減）

- 高速ストア(ZFILL)

高速ストア（ZFILL）

- 高速ストア（ZFILL）とは
- ZFILL（改善前）
- ZFILLの効果（最適化制御行チューニング）

高速ストア（ZFILL）とは

- 高速ストア（ZFILL）とは

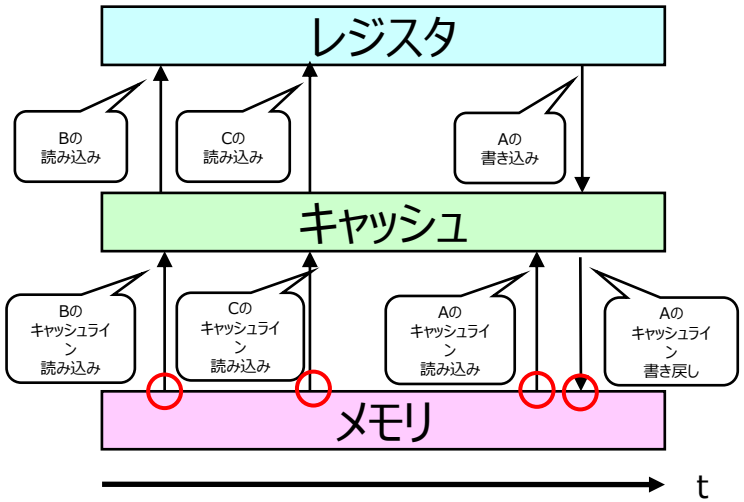
  キャッシュ上に書き込み用のキャッシュライン（中身は不定値）を確保する機能です。これにより、メモリからのキャッシュラインの読み込みが削減できるので、メモリスループットがネックとなっているプログラムでは性能が改善します。

- 動作条件
  - ストア対象の配列がイタレーション間で依存が無いこと
  - 定義のある配列の参照が無いこと
  - 連続的なメモリアクセスであること

例）

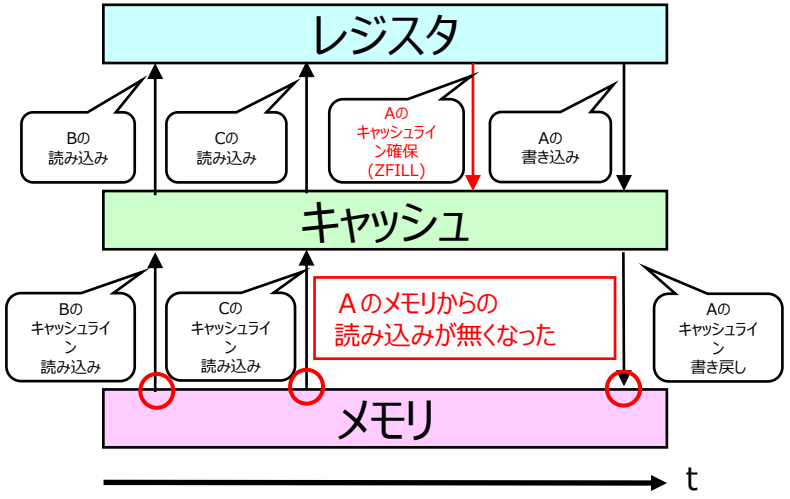
```
DO I = 1, N
      A(I) = B(I) + C(I)
END DO
```

ZFILL未使用時



メモリアクセス数計4回

ZFILL使用時



メモリアクセス数計3回

ZFILLとは (1/2)

FUJITSU

以下の最適化制御行を指定します。

最適化指示子 (Fortran)	意味	指定可能な最適化制御行			
		プログラム単位	DOループ単位	文単位	配列代入文単位
ZFILL[(*m1*)]	ZFILL命令を生成することを指示します。*m1*はキャッシュライン数を表す1～100の10進数です。	×	○	×	○
NOZFILL	ZFILL命令を生成しないことを指示します。	×	○	×	○

最適化指示子 (C/C++)	意味	指定可能な最適化制御行			
		global行	procedure行	loop行	statement行
zfill[(*m1*)]	zfill命令を生成することを指示します。*m1*はキャッシュライン数を表す1～100の10進数です。	×	×	○	×
nozfill	zfill命令を生成しないことを指示します。	×	×	○	×

■ 注意事項

- ZFILL命令は、ループ内でストアされる配列データに対して出力されます。ただし、同一ループ内に参照がある配列、非連続アクセスされる配列またはIF構文配下でストアされる配列に対しては出力されません。
- ZFILL命令が出力された場合、2次キャッシュへのプリフェッチ命令は出力されません。
- ZFILL命令で確保したキャッシュラインは必ずストアするようなループ変形を行うため、以下の最適化が適用できなくなります。これにより、実行性能が低下することがあります。
  - ループアンローリング
  - ループストライピング
- また、以下の場合にも実行性能が低下することがあります。
  - 繰返し数が少ないループ
  - データが1次キャッシュまたは2次キャッシュにある場合

ZFILLとは (2/2)

以下の翻訳オプションを指定することで、最適化制御行チューニングと同等の効果を得ることができます。

翻訳オプション	機能説明
-K{ zfill[=*N*] | nozfill } 1 ≦ *N* ≦ 100	ループ内で書き込みのみ行う配列データについて、データをメモリからロードすることなく、キャッシュ上に書き込み用のキャッシュラインを確保する命令（ZFILL命令）を生成することを指示します。 *N*を指定することで、*N*キャッシュライン先のデータをZFILL命令の対象とします。 *N*は1～100の範囲で指定できます。*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します。 本オプションは、-O2オプション以上が有効な場合に意味があります。デフォルトは、-Knozfillです。

■使用例（改善前ソース時）

```
$ frtpx -Kfast,parallel sample.f90 –Kzfill
$ fccpx -Kfast,parallel sample.f90 –Kzfill
```

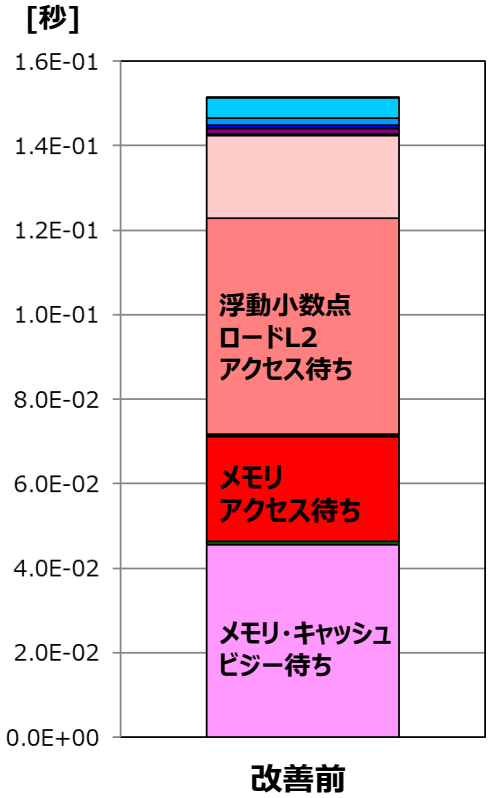
Fortran ZFILL（改善前）

FUJITSU

メモリアクセス負荷が高いプログラムのため、メモリスループットがネックになっています。そのため、データアクセス待ちが多くなっています。

改善前ソース

```
38                real*8 a(n),b(n),c(n),d
39
                  <<< Loop-information Start >>>
                  <<<  [PARALLELIZATION]
                  <<<    Standard iteration count: 1000
                  <<<  [OPTIMIZATION]
                  <<<    SIMD(VL: 8)
                  <<<    SOFTWARE PIPELINING(IPC: 3.25, ITR: 144,
                                                 MVE: 4, POL: S)
                  <<<    PREFETCH(HARD) Expected by compiler :
                  <<<      c, b, a
                  <<< Loop-information  End >>>
40     1  pp   2v      do i=1,n
41     1   p   2v        a(i) = b(i) + c(i)*d
42     1   p   2v      enddo
```



Cache	L1I miss rate (/Effective instruction)	Load-store instruction	L1D miss	L1D miss rate (/Load-store instruction)	L1D miss demand rate (%) (/L1D miss)	L1D miss hardware prefetch rate (%) (/L1D miss)	L1D miss software prefetch rate (%) (/L1D miss)	L2 miss	L2 miss rate (/Load-store instruction)	L2 miss demand rate (%) (/L2 miss)	L2 miss hardware prefetch rate (%) (/L2 miss)	L2 miss software prefetch rate (%) (/L2 miss)
改善前	0.00	3.90E+08	9.39E+07	0.24	27.78%	72.21%	0.01%	9.38E+07	0.24	12.72%	88.66%	0.00%

	Memory Throughput (GB/s)
改善前	211.59

メモリスループットがネックになっている

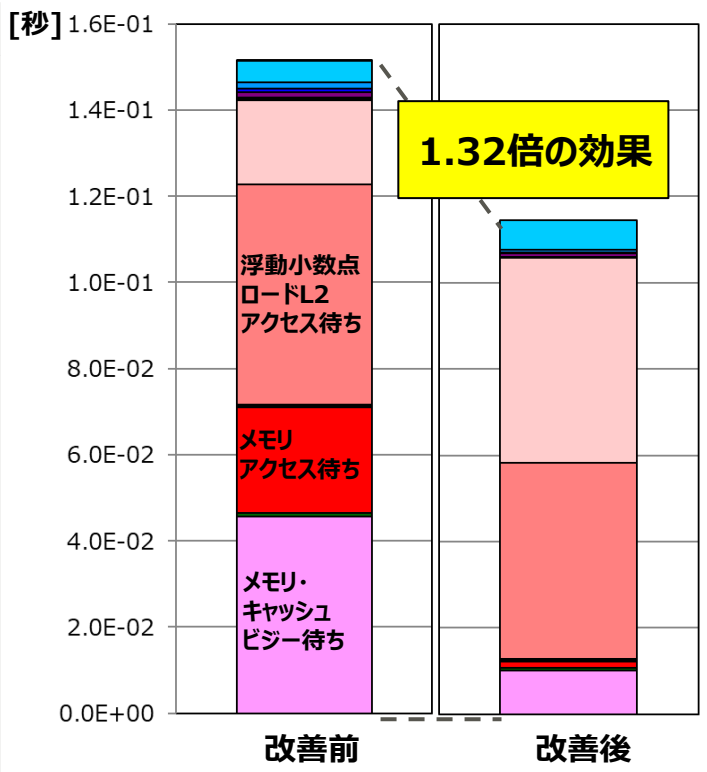
Fortran ZFILLの効果（最適化制御行チューニング）

FUJITSU

ZFILL指示子を指定することでストア命令によるメモリからのキャッシュラインの読み込みがなくなり、L2ミス数が削減されました。その結果、データアクセス待ちが改善されました。

改善後ソース（最適化制御行チューニング）

```
38                  real*8 a(n),b(n),c(n),d
39                  !ocl zfill
                 <<< Loop-information Start >>>
                 <<<  [PARALLELIZATION]
                 <<<    Standard iteration count: 1000
                 <<<  [OPTIMIZATION]
                 <<<    SIMD(VL: 8)
                 <<<    SOFTWARE PIPELINING(IPC: 2.55, ITR: 128,
                                                MVE: 2, POL: S)
                 <<<    PREFETCH(HARD) Expected by compiler :
                 <<<      c, b
                 <<<    PREFETCH(SOFT) : 2
                 <<<     SEQUENTIAL : 2
                 <<<      a: 2
                 <<<    ZFILL          :
                 <<<      a
                 <<< Loop-information  End >>>
40     1  pp    v      do i=1,n
41     1   p    v        a(i) = b(i) + c(i)*d
42     1   p    v      enddo
```



Cache	L1I miss rate (/Effective instruction)	Load-store instruction	L1D miss	L1D miss rate (/Load-store instruction)	L1D miss demand rate (%) (/L1D miss)	L1D miss hardware prefetch rate (%) (/L1D miss)	L1D miss software prefetch rate (%) (/L1D miss)	L2 miss	L2 miss rate (/Load-store instruction)	L2 miss demand rate (%) (/L2 miss)	L2 miss hardware prefetch rate (%) (/L2 miss)	L2 miss software prefetch rate (%) (/L2 miss)
改善前	0.00	3.90E+08	9.39E+07	0.24	27.78%	72.21%	0.01%	9.38E+07	0.24	12.72%	88.66%	0.00%
改善後	0.00	4.38E+08	9.39E+07	0.21	16.80%	49.98%	33.22%	6.25E+07	0.14	1.15%	98.98%	0.00%

	Memory Throughput (GB/s)
改善前	211.59
改善後	209.96

改善後もメモリスループットネックではあるが、L2ミス数が1/3削減された

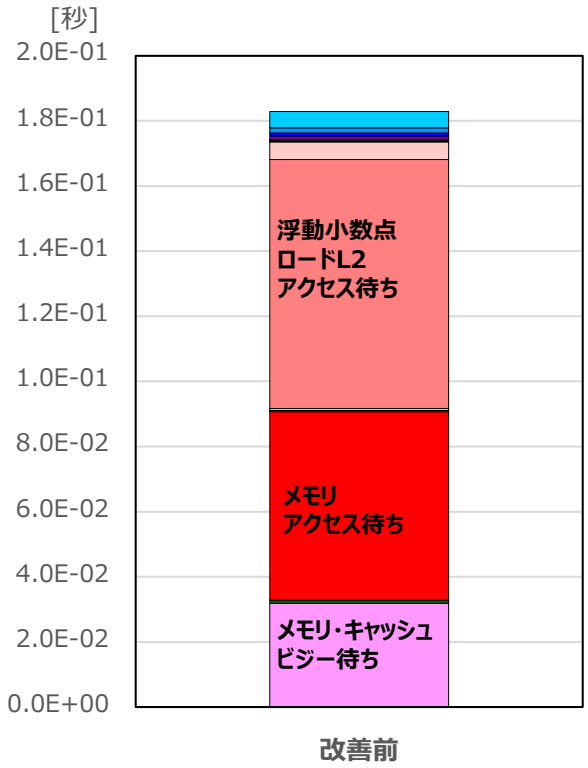
ZFILL（改善前）

FUJITSU

メモリアクセス負荷が高いプログラムのため、データアクセス待ちが多くなっています。

改善前ソース

```
38          void sub(double * restrict a, double * restrict b,
                     double * restrict c, double d, int n){
39            int i;
40
41            #pragma omp parallel for
            <<< Loop-information Start >>>
            <<<  [OPTIMIZATION]
            <<<    SIMD(VL: 8)
            <<<    SOFTWARE PIPELINING(IPC: 3.25, ITR: 144,
                           MVE: 4, POL: S)
            <<<    PREFETCH(HARD) Expected by compiler :
            <<<      (unknown)
            <<< Loop-information  End >>>
42   p      2v    for (i = 0; i < n; i++){
43   p      2v      a[i] = b[i] + c[i]*d;
44   p      2v    }
45            }
```



Cache	L1I miss rate (/Effective instruction)	Load-store instruction	L1D miss	L1D miss rate (/Load-store instruction)	L1D miss demand rate (%) (/L1D miss)	L1D miss hardware prefetch rate (%) (/L1D miss)	L1D miss software prefetch rate (%) (/L1D miss)	L2 miss	L2 miss rate (/Load-store instruction)	L2 miss demand rate (%) (/L2 miss)	L2 miss hardware prefetch rate (%) (/L2 miss)	L2 miss software prefetch rate (%) (/L2 miss)
改善前	0.00	3.95E+08	9.40E+07	0.24	41.24%	58.75%	0.01%	9.39E+07	0.24	19.15%	83.83%	0.00%

Statistics	Memory throughput (GB/s)
改善前	175.68

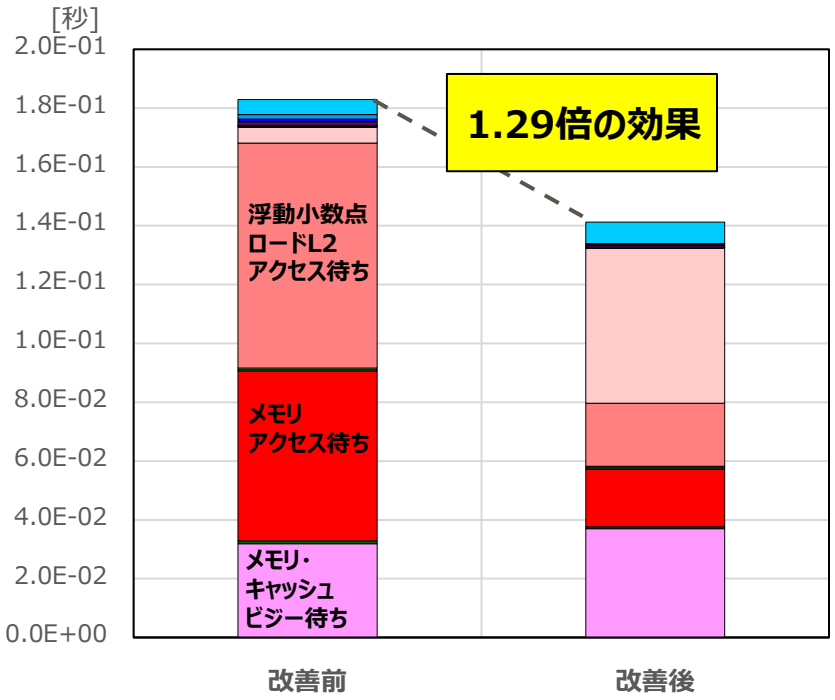
ZFILLの効果（最適化制御行チューニング）

FUJITSU

ZFILL指示子を指定することでストア命令によるメモリからのキャッシュラインの読み込みがなくなり、L2ミス数が削減されました。その結果、データアクセス待ちが改善されました。

改善後ソース（最適化制御行チューニング）

```
37          void sub(double * restrict a, double * restrict b,
                     double * restrict c, double d, int n){
38            int i;
39
40          #pragma omp parallel for
41          #pragma loop zfill
          <<< Loop-information Start >>>
          <<<  [OPTIMIZATION]
          <<<    SIMD(VL: 8)
          <<<    SOFTWARE PIPELINING(IPC: 2.55, ITR: 128,
                                     MVE: 2, POL: S)
          <<<    PREFETCH(HARD) Expected by compiler :
          <<<      (unknown)
          <<<    PREFETCH(SOFT) : 2
          <<<     SEQUENTIAL : 2
          <<<      (unknown): 2
          <<<    ZFILL          :
          <<<      (unknown)
          <<< Loop-information  End >>>
42  p     v    for (i = 0; i < n; i++){
43  p     v      a[i] = b[i] + c[i]*d;
44  p     v    }
45            }
```



Cache	L1I miss rate (/Effective instruction)	Load-store instruction	L1D miss	L1D miss rate (/Load-store instruction)	L1D miss demand rate (%) (/L1D miss)	L1D miss hardware prefetch rate (%) (/L1D miss)	L1D miss software prefetch rate (%) (/L1D miss)	L2 miss	L2 miss rate (/Load-store instruction)	L2 miss demand rate (%) (/L2 miss)	L2 miss hardware prefetch rate (%) (/L2 miss)	L2 miss software prefetch rate (%) (/L2 miss)
改善前	0.00	3.95E+08	9.40E+07	0.24	41.24%	58.75%	0.01%	9.39E+07	0.24	19.15%	83.83%	0.00%
改善後	0.00	4.31E+08	9.40E+07	0.22	30.94%	35.79%	33.27%	6.26E+07	0.15	4.59%	95.79%	0.00%

Statistics	Memory throughput (GB/s)
改善前	175.68
改善後	170.27

L2ミス数が1/3削減された

FUJITSU

データアクセス待ち（スラッシングの改善）

- キャッシュスラッシングとは
- 1次元目の配列要素数を増やすパディング
- 2次元目の配列要素数を増やすパディング
- ダミー配列によるパディング
- ダミー配列によるパディング（サイズが異なる配列）
- 配列マージ（スラッシングの改善）
- ループ分割
- ラージページ環境変数によるパディング

キャッシュスラッシングとは

キャッシュスラッシングとは、キャッシュ上の特定のインデックス（キャッシュ上の位置情報）のデータだけが頻繁に上書きされる現象のことです。
この現象は、1WAYのサイズが16KBであるため、配列サイズが２のべき乗(16KBの倍数)の場合およびループ内のストリーム数が多い場合に発生しやすくなります。

注：ストリームとは、ループの回転に伴って参照定義される一連のデータのことです。

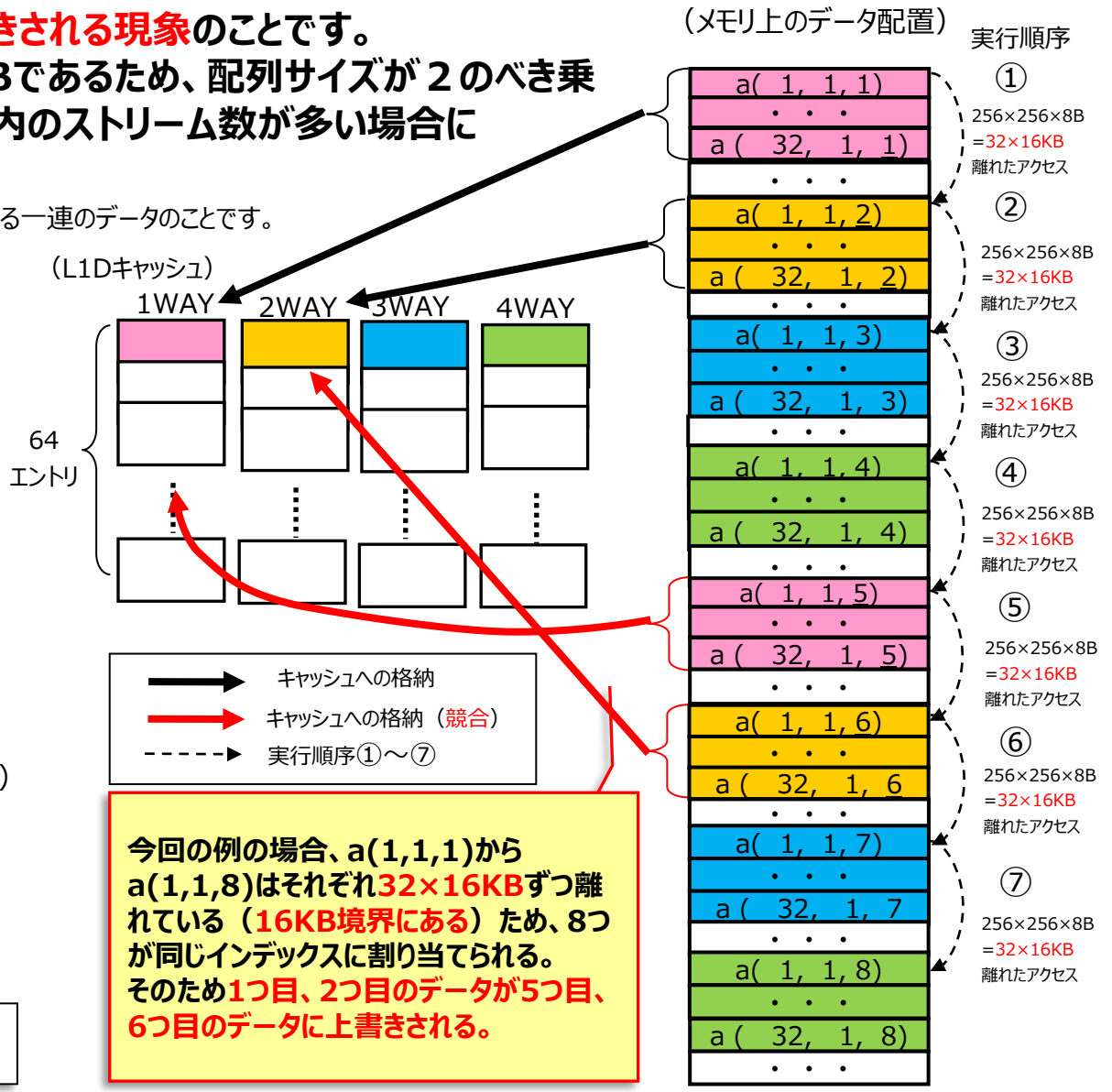
ソース例

```
subroutine sub(a, n, m) ※n=256, m=256
real*8  a(n,m,8)
do j= 1 , m
  do i= 1 , n
    a(i,j,8)=a(i,j,1)+a(i,j,2)+a(i,j,3)+a(i,j,4)+
             a(i,j,5)+a(i,j,6)+a(i,j,7)
  enddo
enddo
end
```

■ L1Dキャッシュスラッシングの目安
(512bitSVEアクセス命令、連続アクセスの場合)

L1D miss rate (/Load-store instruction)
0.25以上

単精度:64/256
倍精度:64/256



今回の例の場合、a(1,1,1)からa(1,1,8)はそれぞれ32×16KBずつ離れている（16KB境界にある）ため、8つが同じインデックスに割り当てられる。そのため1つ目、2つ目のデータが5つ目、6つ目のデータに上書きされる。

FUJITSU

パディング

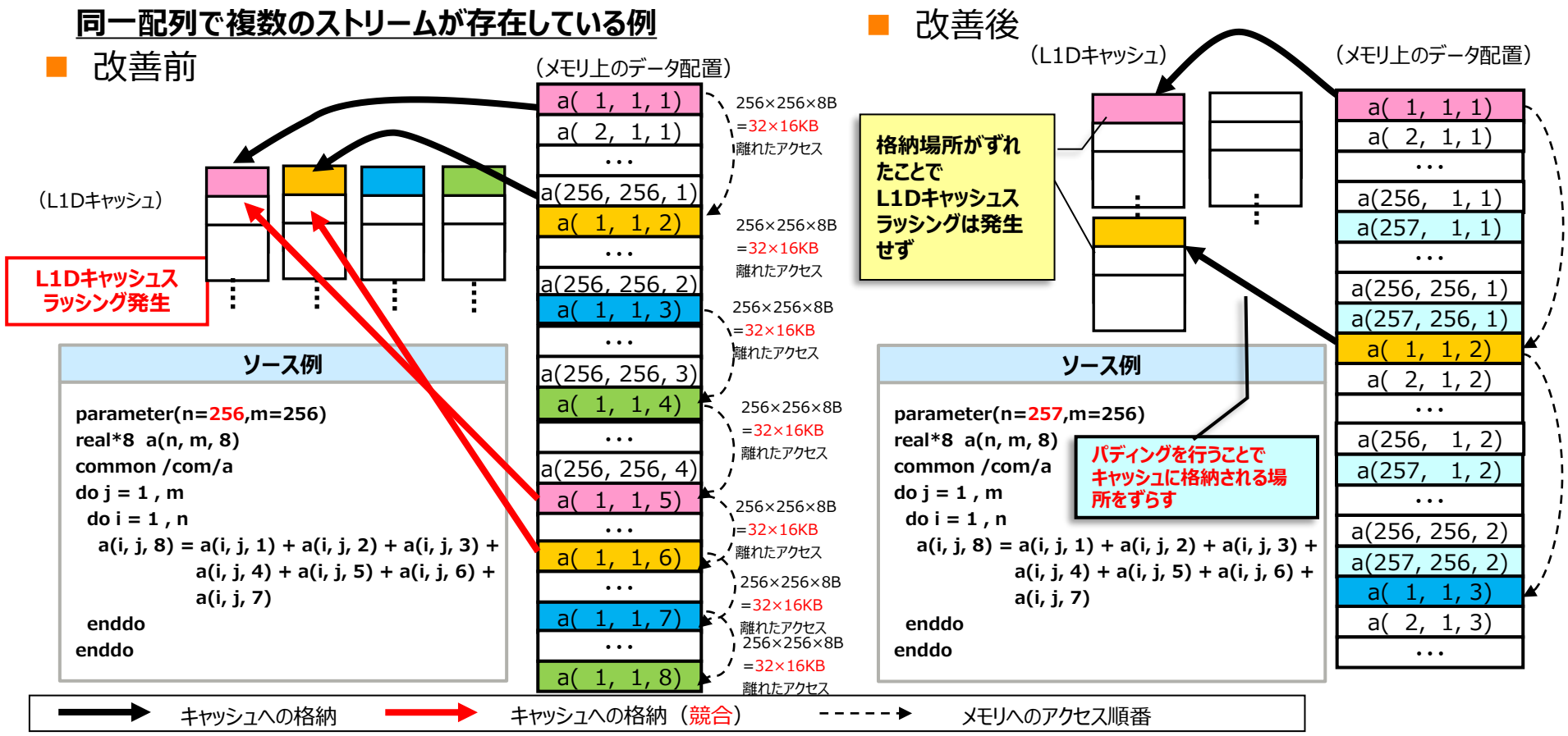
- パディングとは
- 1次元目の配列要素数を増やすパディング
- ２次元目の配列要素数を増やすパディング
- ダミー配列によるパディング
- ダミー配列によるパディング（サイズが異なる配列）

パディングとは

パディングとは、配列間や配列の中にダミーの領域を挿入することです。

- 使用条件
  同一配列で複数のストリームが存在している。
  もしくは
  複数の配列が存在している。
- 狙い
  テンポラリ領域を作りアドレスをずらす。
- 副作用
  問題規模変更ごとにパディング量の変更が必要

同一配列で複数のストリームが存在している例



FUJITSU

1次元目の配列要素数を増やすパディング

- 1 次元目の配列要素数を増やすパディング（改善前）
- 1 次元目の配列要素数を増やすパディングの効果（ソースチューニング）
- 1 次元目の配列要素数を増やすパディングの効果（翻訳オプションチューニング）

Fortran 1次元目の配列要素数を増やすパディング（改善前）

FUJITSU

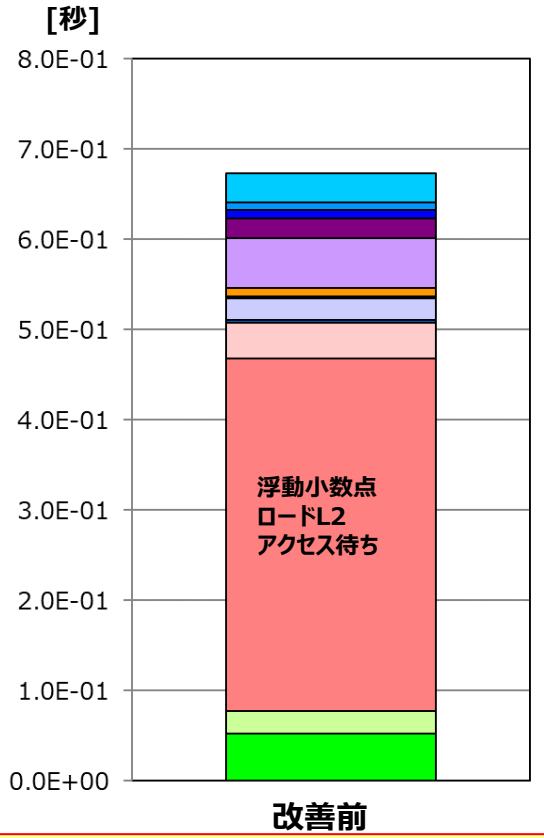
配列aのそれぞれのストリーム同士が16KB境界にあるためL1Dキャッシュスラッシングが発生します。そのため、浮動小数点ロードL2アクセス待ちが多くなっています。

改善前ソース

```
42                parameter(n=256,m=256)
43                real*8  a(n, m, 8)
44                common /com/a
                <<< Loop-information Start
                <<<  [PARALLELIZATION]
                <<<    Standard iteration co
                <<<  [OPTIMIZATION]
                <<<    COLLAPSED
                <<<    SIMD(VL: 8)
                <<<    SOFTWARE PIPELINING(IPC: 2.66, ITR: 104,
                                           MVE: 7, POL: S)
                <<<    PREFETCH(HARD) Expected by compiler :
                <<<      a
                <<< Loop-information  End >>>
45    1  pp     v      do j = 1 , m
                <<< Loop-information Start >>>
                <<<  [OPTIMIZATION]
                <<<    COLLAPSED
                <<< Loop-information  End >>>
46    2   p            do i = 1 , n
47    2   p    v         a(i, j, 8) = a(i, j, 1) + a(i, j, 2) + a(i, j, 3) + &
48    2                      a(i, j, 4) + a(i, j, 5) + a(i, j, 6) + a(i, j, 7)
49    2   p    v       enddo
50    1   p          enddo
```

配列サイズ
256×256×8B=
32×16KB
(16KB境界)

同一配列の
ストリーム



配列が連続アクセスであるにもかかわらずL1Dミスが高い
⇒ L1Dキャッシュスラッシングが発生している

Cache	L1I miss rate (/Effective instruction)	Load-store instruction	L1D miss	L1D miss rate (/Load-store instruction)	L1D miss demand rate (%) (/L1D miss)	L1D miss hardware prefetch rate (%) (/L1D miss)	L1D miss software prefetch rate (%) (/L1D miss)	L2 miss	L2 miss rate (/Load-store instruction)	L2 miss demand rate (%) (/L2 miss)	L2 miss hardware prefetch rate (%) (/L2 miss)	L2 miss software prefetch rate (%) (/L2 miss)
改善前	0.00	3.08E+09	1.30E+09	0.42	67.73%	32.27%	0.00%	1.50E+04	0.00	37.70%	72.22%	0.00%

Fortran 1次元目の配列要素数を増やすパディング（改善後）

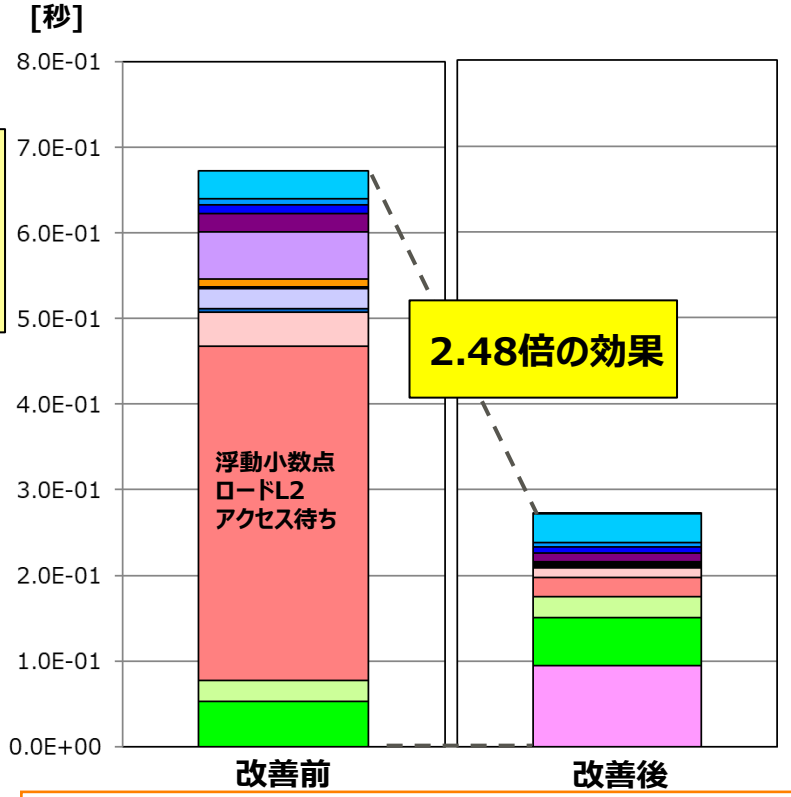
FUJITSU

配列aのそれぞれのストリームの1次元目にパディング(+1)することで、L1Dキャッシュスラッシングを回避しました。その結果、浮動小数点ロードL2アクセス待ちが改善されました。

改善後ソース

```
42                parameter(n=257,m=256)
43                real*8  a(n, m, 8)
44                common /com/a
                <<< Loop-information Start
                <<<  [PARALLELIZATION]
                <<<    Standard iteration cou
                <<<  [OPTIMIZATION]
                <<<    COLLAPSED
                <<<    SIMD(VL: 8)
                <<<    SOFTWARE PIPELINING(IPC: 2.66, ITR: 104,
                                           MVE: 7, POL: S)
                <<<    PREFETCH(HARD) Expected by compiler :
                <<<      a
                <<< Loop-information  End >>>
45    1  pp   v      do j = 1 , m
                <<< Loop-information Start >>>
                <<<  [OPTIMIZATION]
                <<<    COLLAPSED
                <<< Loop-information  End >>>
46    2   p           do i = 1 , n
47    2   p   v         a(i, j, 8) = a(i, j, 1) + a(i, j, 2) + a(i, j, 3) + &
48    2                              a(i, j, 4) + a(i, j, 5) + a(i, j, 6) + a(i, j, 7)
49    2   p   v       enddo
50    1   p         enddo
```

nを+1することで
16KB境界からずらす



■ 注意事項
パディング数が大きすぎると、データの連続性が損なわれハードウェアプリフェッチが効かなくなることがあります。

L1Dミスが減少した

Cache	L1I miss rate (/Effective instruction)	Load-store instruction	L1D miss	L1D miss rate (/Load-store instruction)	L1D miss demand rate (%) (/L1D miss)	L1D miss hardware prefetch rate (%) (/L1D miss)	L1D miss software prefetch rate (%) (/L1D miss)	L2 miss	L2 miss rate (/Load-store instruction)	L2 miss demand rate (%) (/L2 miss)	L2 miss hardware prefetch rate (%) (/L2 miss)	L2 miss software prefetch rate (%) (/L2 miss)
改善前	0.00	3.08E+09	1.30E+09	0.42	67.73%	32.27%	0.00%	1.50E+04	0.00	37.70%	72.22%	0.00%
改善後	0.00	2.62E+09	4.58E+08	0.17	8.20%	91.80%	0.00%	9.69E+03	0.00	46.19%	58.89%	0.00%

C/C++ 1次元目の配列要素数を増やすパディング（改善前）

FUJITSU

配列aのそれぞれのストリーム同士が16KB境界にあるためL1Dキャッシュスラッシングが発生します。そのため、浮動小数点ロードL2アクセス待ちが多くなっています。

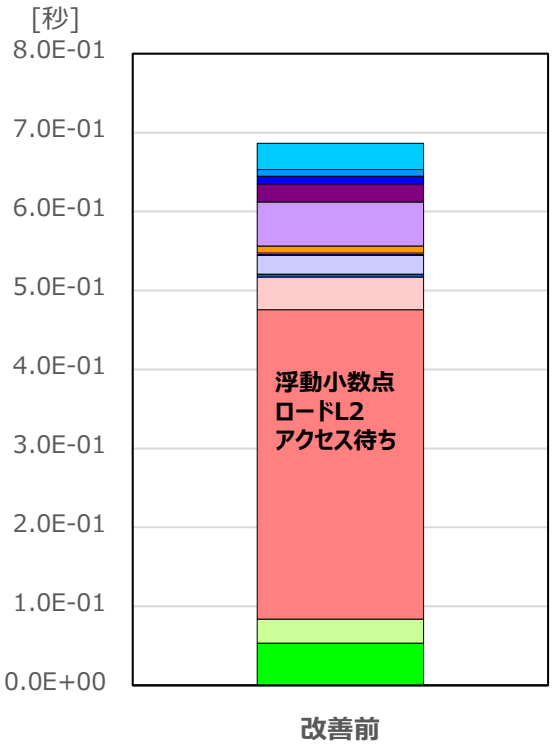
改善前ソース

```
30          void sub(void){
31            int i, j;
32
33          #pragma omp parallel for collapse(2)
            <<< Loop-information Start >>>
            <<<  [OPTIMIZATION]
            <<<    SIMD(VL: 8)
            <<<    SOFTWARE PIPELINING(IPC: 2.66, ITR: 104, MVE: 7, POL: S)
            <<<    PREFETCH(HARD) Expected by compiler :
            <<<      a
            <<< Loop-information  End >>>
34  p     v    for (j = 0; j < m; j++){
35  p     v      for (i = 0; i < n; i++){
36  p     v        a[7][j][i] = a[0][j][i] + a[1][j][i] + a[2][j][i] + a[3][j][i] +
                               a[4][j][i] + a[5][j][i] + a[6][j][i];
37  p     v      }
38  p     v    }
39          }
```

配列宣言
double a[8][256][256];

aの配列サイズ
256×256×8B=
32×16KB(16KB境界)

同一配列の
ストリーム



配列が連続アクセスであるにもかかわらずL1Dミスが高い
⇒ L1Dキャッシュスラッシングが発生している

Cache	L1I miss rate (/Effective instruction)	Load-store instruction	L1D miss	L1D miss rate (/Load-store instruction)	L1D miss demand rate (%) (/L1D miss)	L1D miss hardware prefetch rate (%) (/L1D miss)	L1D miss software prefetch rate (%) (/L1D miss)	L2 miss	L2 miss rate (/Load-store instruction)	L2 miss demand rate (%) (/L2 miss)	L2 miss hardware prefetch rate (%) (/L2 miss)	L2 miss software prefetch rate (%) (/L2 miss)
改善前	0.00	3.08E+09	1.33E+09	0.43	68.32%	31.68%	0.00%	2.65E+04	0.00	49.48%	60.25%	0.00%

最終次元の配列要素数を増やすパディング（改善後）

FUJITSU

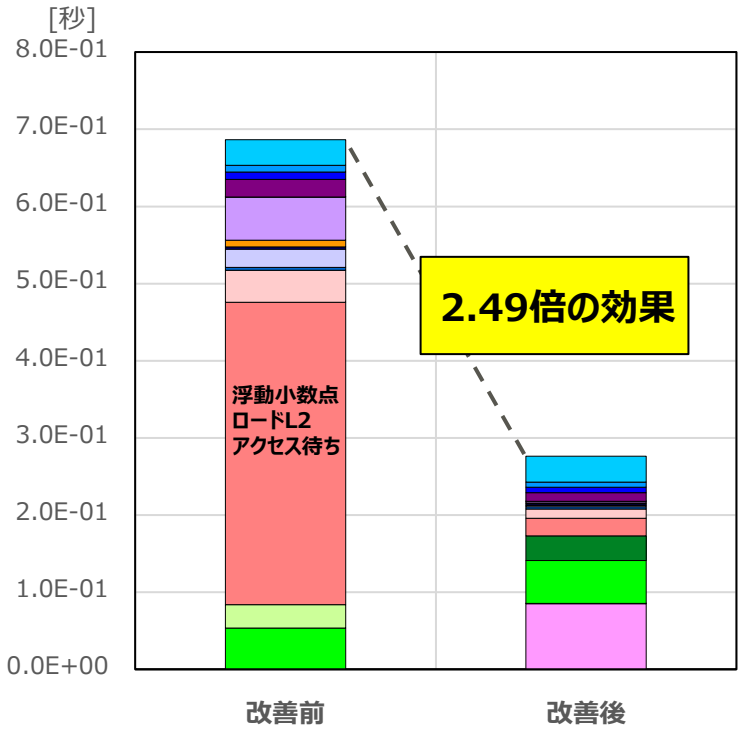
配列aのそれぞれのストリームの最終次元にパディング(+1)することで、L1Dキャッシュスラッシングを回避しました。その結果、浮動小数点ロードL2アクセス待ちが改善されました。

改善後ソース

```
30          void sub(void){
31            int i, j;
32
33          #pragma omp parallel for collaps
            <<< Loop-information Start >>>
            <<<  [OPTIMIZATION]
            <<<    SIMD(VL: 8)
            <<<    SOFTWARE PIPELINING(IPC: 2.66, ITR: 104, MVE: 7, POL: S)
            <<<    PREFETCH(HARD) Expected by compiler :
            <<<      a
            <<< Loop-information  End >>>
34  p     v    for (j = 0; j < m; j++){
35  p     v      for (i = 0; i < n; i++){
36  p     v        a[7][j][i] = a[0][j][i] + a[1][j][i] + a[2][j][i] +
                               a[3][j][i] + a[4][j][i] + a[5][j][i] + a[6][j][i];
37  p     v      }
38  p     v    }
39          }
```

配列宣言
double a[8][256][257];

配列aの最終次元を+1することで16KB境界からずらす



■ **注意事項**
パディング数が大きすぎると、データの連続性が損なわれハードウェアプリフェッチが効かなくなることがあります。

L1Dミスが減少した

Cache	L1I miss rate (/Effective instruction)	Load-store instruction	L1D miss	L1D miss rate (/Load-store instruction)	L1D miss demand rate (%) (/L1D miss)	L1D miss hardware prefetch rate (%) (/L1D miss)	L1D miss software prefetch rate (%) (/L1D miss)	L2 miss	L2 miss rate (/Load-store instruction)	L2 miss demand rate (%) (/L2 miss)	L2 miss hardware prefetch rate (%) (/L2 miss)	L2 miss software prefetch rate (%) (/L2 miss)
改善前	0.00	3.08E+09	1.33E+09	0.43	68.32%	31.68%	0.00%	2.65E+04	0.00	49.48%	60.25%	0.00%
改善後	0.00	2.67E+09	4.60E+08	0.17	8.50%	91.50%	0.00%	2.26E+04	0.00	22.14%	83.91%	0.00%

Fortran 1次元目の配列要素数を増やすパディングの効果（翻訳オプションチューニング）

以下の翻訳オプション(Fortran固有)を指定することで、ソースチューニングと同等の効果を得ることができます。

翻訳オプション	機能説明
-Karraypad_const[=*N*] (1≦*N*≦2,147,483,647)	１次元目が明示上下限であり、かつその上下限式が定数式の配列に対して*N* 要素のパディングを行います。*N* が省略された場合、対象の配列ごとにコンパイラがパディングの量を決めます。パディングとは、配列の内部に隙間を作ることです。
-Karraypad_expr=*N* (1≦*N*≦2,147,483,647)	上下限式が定数式かどうかにかかわらず、1次元目が明示上下限の配列に対して*N* 要素のパディングを行います。

■使用例（改善前ソース時）

```
$ frtpx -Kfast,parallel sample.f90 -Karraypad_expr=1
```

対象配列を自動選出しパディングを適用

- 注意事項
  - 対象配列を使うソース全てにオプション指定が必要です。
  - パディングの効果はプログラムによって異なります。
  - 正しくない使い方をした場合には計算結果が異なる場合があります。
  - -Karraypad_const [=N]オプションと-Karray_expr=Nオプションは、同時に指定できません。

FUJITSU

２次元目の配列要素数を増やすパディング

- １次元目の配列要素数を増やすパディングで改善されないケース
- ２次元目の配列要素数を増やすパディングについて
- ２次元目の配列要素数を増やすパディング（改善前）
- ２次元目の配列要素数を増やすパディングの効果（ソースチューニング）

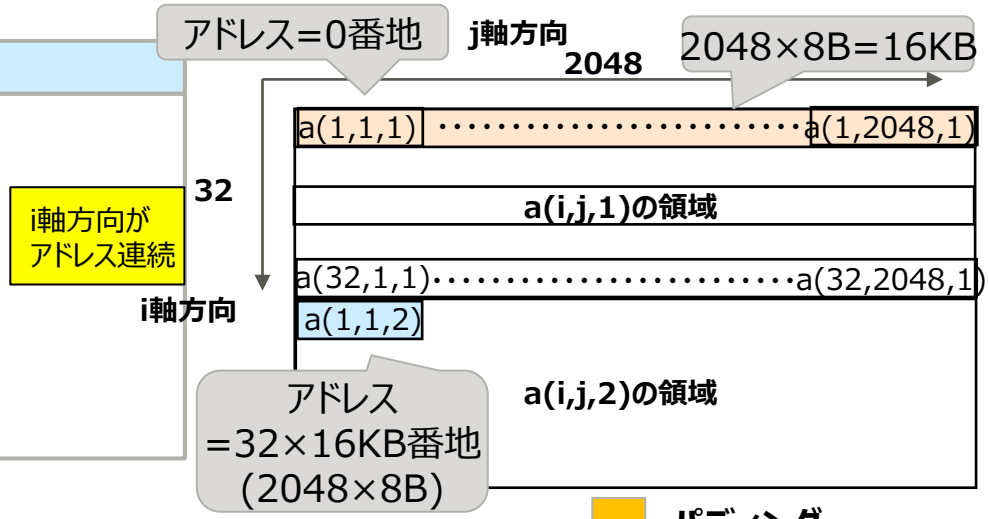
1次元目の配列要素数を増やすパディングで改善されないケース

FUJITSU

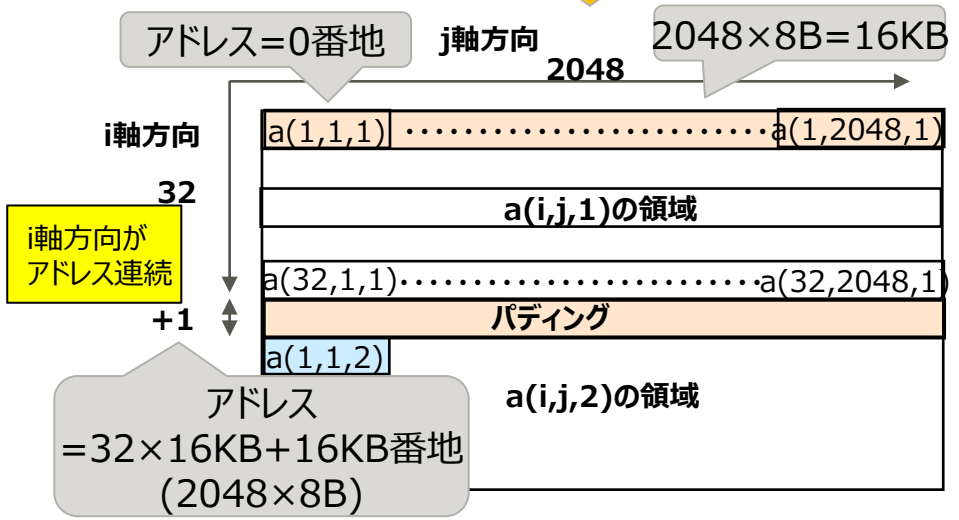
配列のサイズによっては、1次元目の配列要素にパディング(+1)しても改善されないことがあります。

改善前ソース

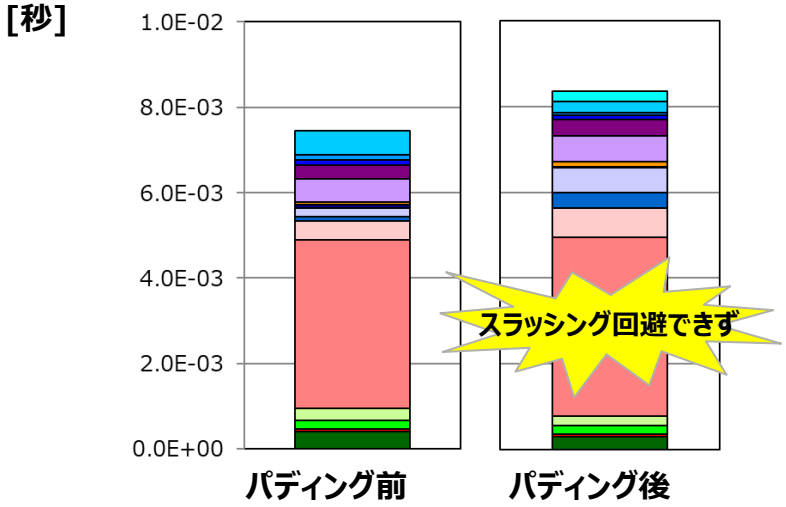
```
parameter(k=32,l=2048)
real*8  a(k, l, 8)
common /com/a
do j = 1 , l
  do i = 1 , k
    a(i, j, 8) = a(i, j, 1) + a(i, j, 2) + a(i, j, 3) + &
                 a(i, j, 4) + a(i, j, 5) + a(i, j, 6) + a(i, j, 7)
  enddo
enddo
end
```



パディング



	L1I miss rate (/Effective instruction)	L1D miss	L1D miss rate (/Load-store instruction)	L1D miss demand rate (%) (/L1D miss)
改善前	0.00	1.04E+07	0.39	68.52
改善後	0.00	1.32E+07	0.52	75.20



配列aは16KB境界のままなのでスラッシングが発生する

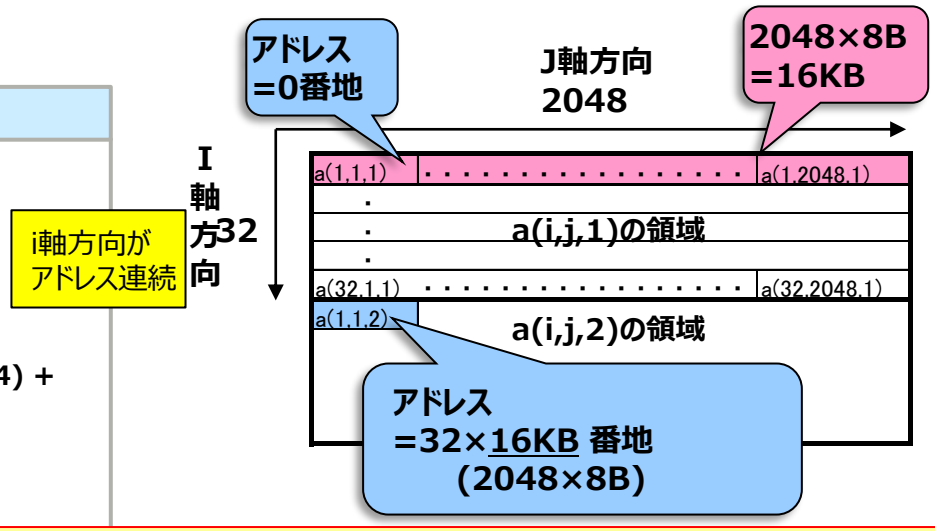
2次元目の配列要素数を増やすパディングについて

FUJITSU

2次元目にパディング(+1)することで16KB境界がずれるため、L1Dキャッシュスラッシングが回避されます。

改善前ソース

```
33 parameter(k=32,l=2048)
34 real*8  a(k, l, 8)
35 common /com/a
36 do j = 1 , l
37   do i = 1 , k
38     a(i, j, 8) = a(i, j, 1) + a(i, j, 2) + a(i, j, 3) + a(i, j, 4) +
                    a(i, j, 5) + a(i, j, 6) + a(i, j, 7)
39   enddo
40 enddo
41 end
```

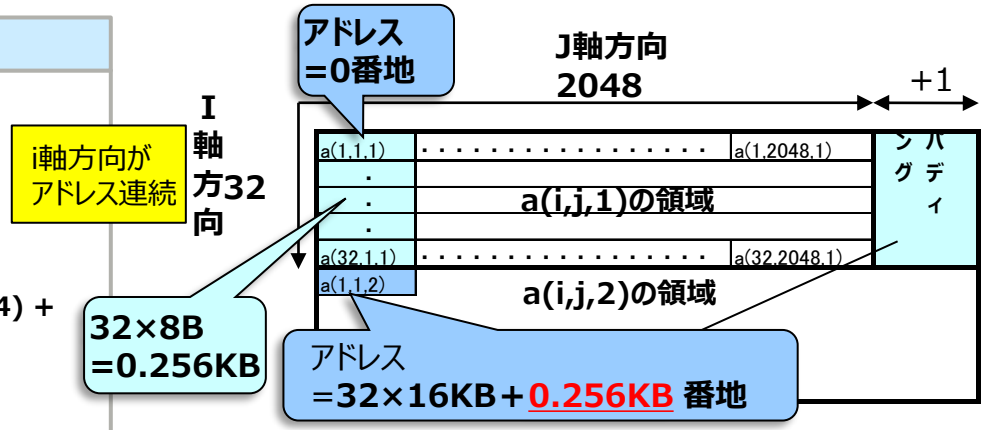


16KB境界になるのでスラッシングが発生してしまう

改善後ソース

```
33 parameter(k=32,l=2048)
34 real*8  a(k, l+1, 8)
35 common /com/a
36 do j = 1 , l
37   do i = 1 , k
38     a(i, j, 8) = a(i, j, 1) + a(i, j, 2) + a(i, j, 3) + a(i, j, 4) +
                    a(i, j, 5) + a(i, j, 6) + a(i, j, 7)
39   enddo
40 enddo
41 end
```

配列aの２次元目をパディング



16KB境界が崩れるのでスラッシングが回避される

Fortran 2次元目の配列要素数を増やすパディング（改善前）

FUJITSU

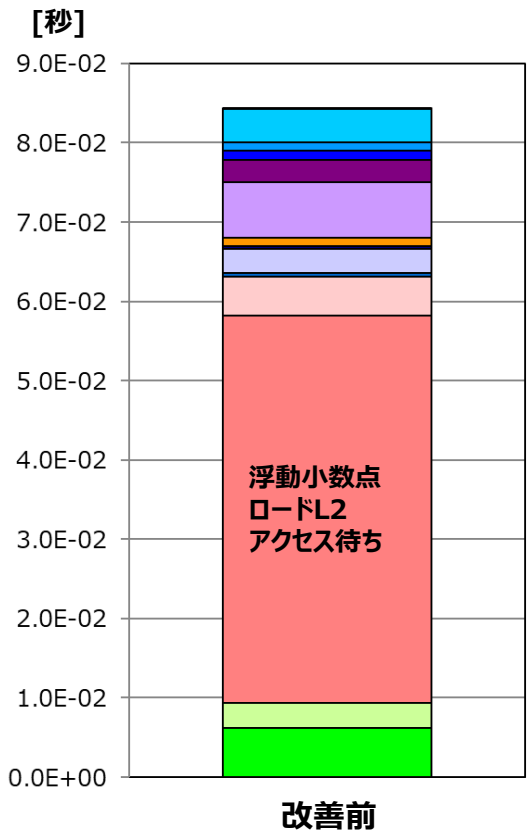
配列aのそれぞれのストリーム同士が16KB境界にあるためL1Dキャッシュスラッシングが発生します。そのため、浮動小数点ロードL2アクセス待ちが多くなっています。

改善前ソース

```
38              parameter(n=32,m=2048)
39              real*8  a(n, m, 8)
40              common /com/a
             <<< Loop-information Sta
             <<<  [PARALLELIZATION]
             <<<    Standard iteration
             <<<  [OPTIMIZATION]
             <<<    COLLAPSED
             <<<    SIMD(VL: 8)
             <<<    SOFTWARE PIPELINING(IPC: 2.66, ITR: 104,
                                        MVE: 7, POL: S)
             <<<    PREFETCH(HARD) Expected by compiler :
             <<<      a
             <<< Loop-information  End >>>
41    1  pp   v      do j = 1 , m
             <<< Loop-information Sta
             <<<  [OPTIMIZATION]
             <<<    COLLAPSED
             <<< Loop-information  End >>>
42    2  p           do i = 1 , n
43    2  p   v         a(i, j, 8) = a(i, j, 1) + a(i, j, 2) + a(i, j, 3) + &
44    2                    a(i, j, 4) + a(i, j, 5) + a(i, j, 6) + a(i, j, 7)
45    2  p   v        enddo
46    1  p           enddo
```

配列サイズ
32×2048×8B=
32×16KB
(16KB境界)

同一配列の
ストリーム



配列が連続アクセスであるにもかかわらずL1Dミスが高い
⇒ L1Dキャッシュスラッシングが発生している

Cache	L1I miss rate (/Effective instruction)	Load-store instruction	L1D miss	L1D miss rate (/Load-store instruction)	L1D miss demand rate (%) (/L1D miss)	L1D miss hardware prefetch rate (%) (/L1D miss)	software prefetch rate (%) (/L1D miss)	L2 miss	L2 miss rate (/Load-store instruction)	L2 miss demand rate (%) (/L2 miss)	hardware prefetch rate (%) (/L2 miss)	software prefetch rate (%) (/L2 miss)
改善前	0.00	3.88E+08	1.63E+08	0.42	67.76%	32.22%	0.01%	1.01E+04	0.00	71.17%	39.65%	0.00%

Fortran 2次元目の配列要素数を増やすパディングの効果（ソースチューニング）

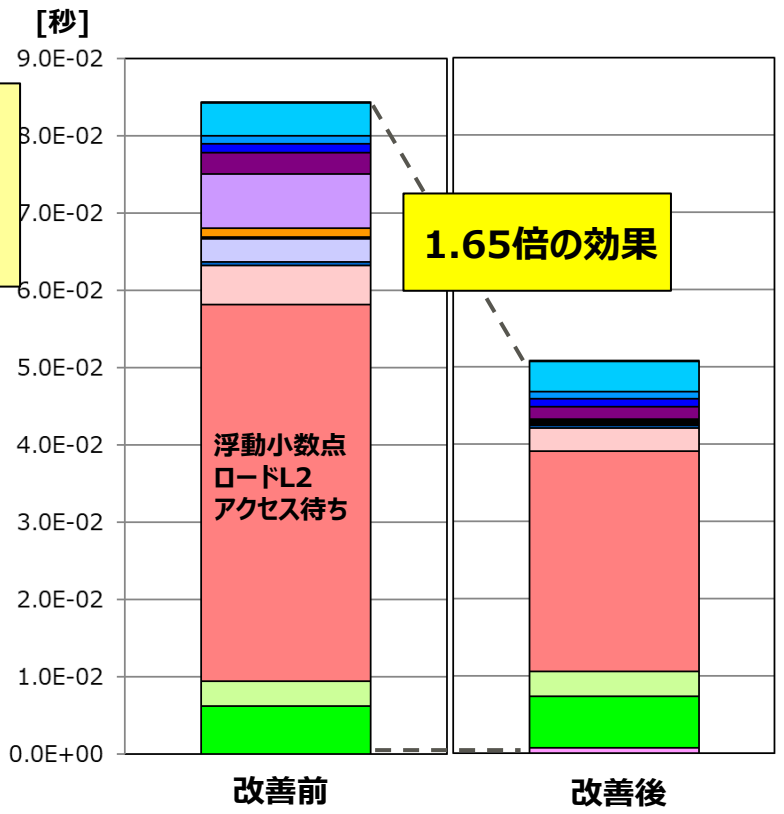
FUJITSU

配列aのそれぞれのストリームの2次元目にパディング(+1)することで、L1Dキャッシュスラッシングを回避しました。その結果、浮動小数点ロードL2アクセス待ちが改善されました。

改善後ソース

```
39              parameter(n=32,m=2048)
40              real*8  a(n, m+1, 8)
41              common /com/a
              <<< Loop-information Start >>>
              <<<  [PARALLELIZATION]
              <<<    Standard iteration cou
              <<<  [OPTIMIZATION]
              <<<    COLLAPSED
              <<<    SIMD(VL: 8)
              <<<    SOFTWARE PIPELINING(IPC: 2.66, ITR: 104,
                                         MVE: 7, POL: S)
              <<<    PREFETCH(HARD) Expected by compiler :
              <<<      a
              <<< Loop-information  End >>>
42    1  pp   v      do j = 1 , m
              <<< Loop-information Start >>>
              <<<  [OPTIMIZATION]
              <<<    COLLAPSED
              <<< Loop-information  End >>>
43    2  p           do i = 1 , n
44    2  p    v       a(i, j, 8) = a(i, j, 1) + a(i, j, 2) + a(i, j, 3) + &
45    2                           a(i, j, 4) + a(i, j, 5) + a(i, j, 6) + a(i, j, 7)
46    2  p    v      enddo
47    1  p          enddo
```

mを+1することで
16KB境界からずらす



L1Dミスが減少した

Cache	L1I miss rate (/Effective instruction)	Load-store instruction	L1D miss	L1D miss rate (/Load-store instruction)	L1D miss demand rate (%) (/L1D miss)	L1D miss hardware prefetch rate (%) (/L1D miss)	L1D miss software prefetch rate (%) (/L1D miss)	L2 miss	L2 miss rate (/Load-store instruction)	L2 miss demand rate (%) (/L2 miss)	L2 miss hardware prefetch rate (%) (/L2 miss)	L2 miss software prefetch rate (%) (/L2 miss)
改善前	0.00	3.88E+08	1.63E+08	0.42	67.76%	32.22%	0.01%	1.01E+04	0.00	71.17%	39.65%	0.00%
改善後	0.00	3.26E+08	1.07E+08	0.33	51.02%	48.98%	0.00%	9.13E+03	0.00	72.77%	33.09%	0.00%

2次元目の配列要素数を増やすパディング（改善前）

FUJITSU

配列aのそれぞれのストリーム同士が16KB境界にあるためL1Dキャッシュスラッシングが発生します。そのため、浮動小数点ロードL2アクセス待ちが多くなっています。

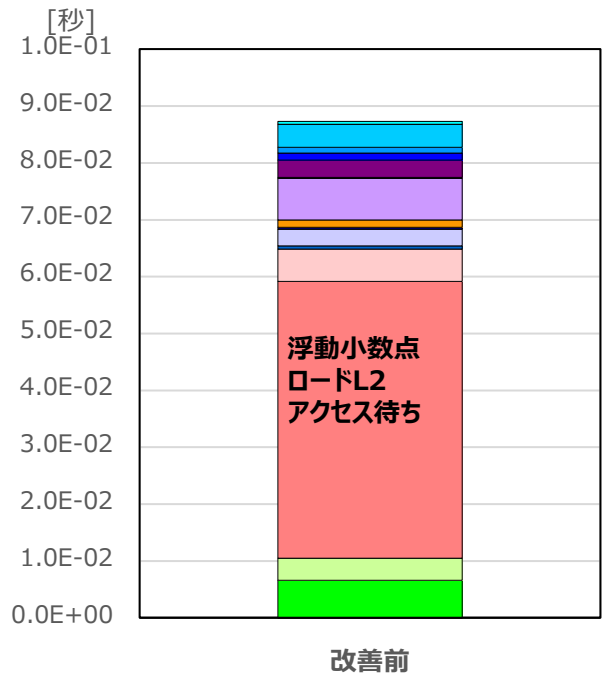
改善前ソース

```
30          void sub(void){
31            int i, j;
32
33          #pragma omp parallel for collapse(2)
           <<< Loop-information Start >>>
           <<<  [OPTIMIZATION]
           <<<    SIMD(VL: 8)
           <<<    SOFTWARE PIPELINING(IPC: 2.66, ITR: 104, MVE: 7, POL: S)
           <<<    PREFETCH(HARD) Expected by com
           <<<      a
           <<< Loop-information  End >>>
34  p     v    for (j = 0; j < m; j++){
35  p     v      for (i = 0; i < n; i++){
36  p     v        a[7][j][i] = a[0][j][i] + a[1][j][i] + a[2][j][i]
                             + a[3][j][i] + a[4][j][i] + a[5][j][i] + a[6][j][i];
37  p     v      }
38  p     v    }
39          }
```

配列宣言
double a[8][2048][32];

aの配列サイズ
32×2048×8B=
32×16KB(16KB境界)

同一配列の
ストリーム



配列が連続アクセスであるにもかかわらずL1Dミスが高い
⇒ L1Dキャッシュスラッシングが発生している

Cache	L1I miss rate (/Effective instruction)	Load-store instruction	L1D miss	L1D miss rate (/Load-store instruction)	L1D miss demand rate (%) (/L1D miss)	L1D miss hardware prefetch rate (%) (/L1D miss)	L1D miss software prefetch rate (%) (/L1D miss)	L2 miss	L2 miss rate (/Load-store instruction)	L2 miss demand rate (%) (/L2 miss)	L2 miss hardware prefetch rate (%) (/L2 miss)	L2 miss software prefetch rate (%) (/L2 miss)
改善前	0.00	3.92E+08	1.66E+08	0.42	68.39%	31.61%	0.00%	2.04E+04	0.00	39.77%	72.33%	0.00%

C/C++ 2次元目の配列要素数を増やすパディングの効果（ソースチューニング）

FUJITSU

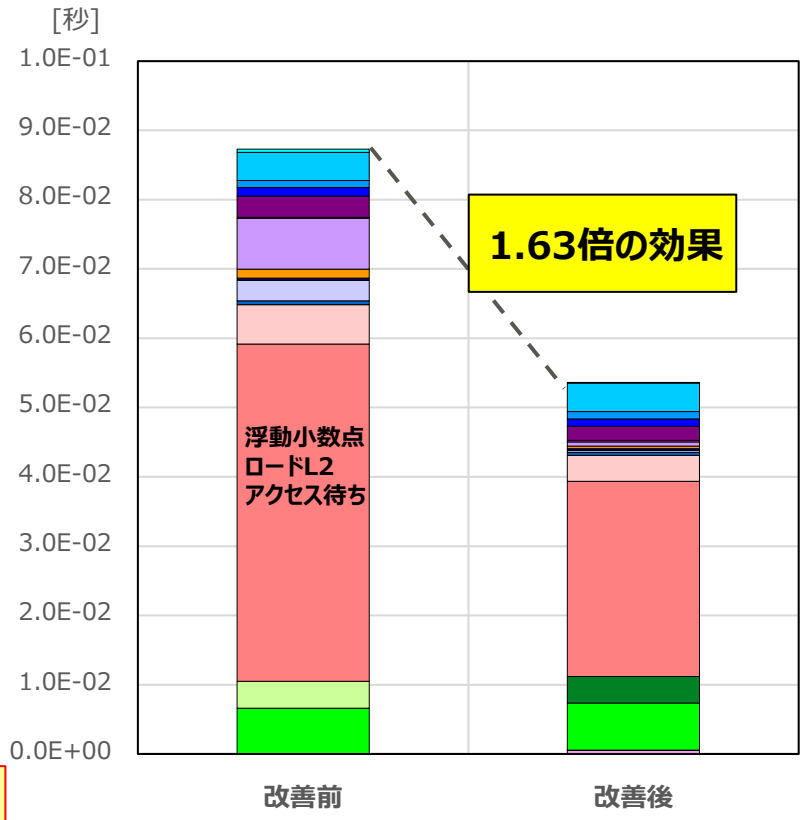
配列aのそれぞれのストリームの2次元目にパディング(+1)することで、L1Dキャッシュスラッシングを回避しました。その結果、浮動小数点ロードL2アクセス待ちが改善されました。

改善後ソース

```
30          void sub(void){
31            int i, j;
32
33          #pragma omp parallel for collapse(2)
            <<< Loop-information Start >>>
            <<<  [OPTIMIZATION]
            <<<    SIMD(VL: 8)
            <<<    SOFTWARE PIPELINING(IPC: 2.66, ITR: 104, MVE: 7, POL: S)
            <<<    PREFETCH(HARD) Expected by compiler :
            <<<      a
            <<< Loop-information  End >>>
34  p     v    for (j = 0; j < m; j++){
35  p     v      for (i = 0; i < n; i++){
36  p     v        a[7][j][i] = a[0][j][i] + a[1][j][i] + a[2][j][i] + a[3][j][i]
                             + a[4][j][i] + a[5][j][i] + a[6][j][i];
37  p     v      }
38  p     v    }
39          }
```

配列宣言
double a[8][2049][32];

配列aの2次元目を+1することで16KB境界からずらす



L1Dミスが減少した

Cache	L1I miss rate (/Effective instruction)	Load-store instruction	L1D miss	L1D miss rate (/Load-store instruction)	L1D miss demand rate (%) (/L1D miss)	L1D miss hardware prefetch rate (%) (/L1D miss)	L1D miss software prefetch rate (%) (/L1D miss)	L2 miss	L2 miss rate (/Load-store instruction)	L2 miss demand rate (%) (/L2 miss)	L2 miss hardware prefetch rate (%) (/L2 miss)	L2 miss software prefetch rate (%) (/L2 miss)
改善前	0.00	3.92E+08	1.66E+08	0.42	68.39%	31.61%	0.00%	2.04E+04	0.00	39.77%	72.33%	0.00%
改善後	0.00	3.44E+08	1.07E+08	0.31	51.12%	48.88%	0.00%	2.21E+04	0.00	19.24%	84.94%	0.00%

FUJITSU

ダミー配列によるパディング

- ダミー配列によるパディング（改善前）
- ダミー配列によるパディングの効果（ソースチューニング）
- ダミー配列によるパディングの効果（翻訳オプションチューニング）

Fortran ダミー配列によるパディング（改善前）

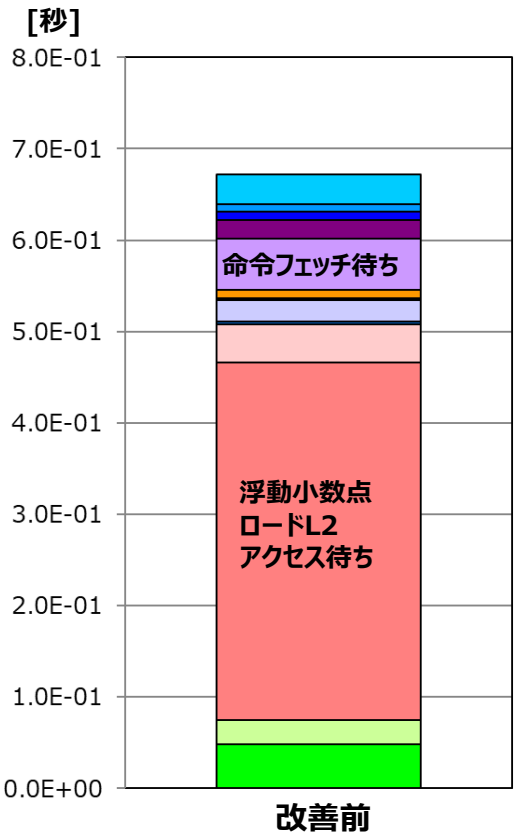
FUJITSU

それぞれの配列が16KB境界にあるため、L1Dキャッシュスラッシングが発生します。
そのため、浮動小数点ロードL2アクセス待ちが多くなっています。

改善前ソース

```
1              parameter(n=256,m=256)
2              real*8  a(n, m),b(n,m),c(n,m),d(n,m),e(n,m),f(n,m),g(n,m),h(n,m)
3              character (1),parameter :: null0=z'00'
4              common /test/a,b,c,d,e,f,g,h
:
27    1  s   s       call sub()
:
34               subroutine sub()
35               parameter(n=256,m=256)
36               real*8  a(n, m),b(n,m),c(n,m),d(n,m),e(n,m),f(n,m),g(n,m),h(n,m)
37               common /test/a,b,c,d,e,f,g,h
              <<< Loop-information Start >>>
              <<<  [PARALLELIZATION]
              <<<    Standard iteration count: 433
              <<<  [OPTIMIZATION]
              <<<    COLLAPSED
              <<<    SIMD(VL: 8)
              <<<    SOFTWARE PIPELINING(IPC: 2.66, ITR: 104, MVE: 7, POL: S)
              <<< Loop-information  End >>>
38    1  pp   v     do j = 1 , m
              <<< Loop-information Start >>>
              <<<  [OPTIMIZATION]
              <<<    COLLAPSED
              <<< Loop-information  End >>>
39    2  p          do i = 1 , n
40    2  p   v        a(i, j) = b(i, j) + c(i, j) + d(i, j) + e(i, j) + f(i ,j) + g(i ,j) + h(i ,j)
41    2  p   v       enddo
42    1  p         enddo
```

配列サイズ
256×256×8B=
32×16KB
(16KB境界)



配列が連続アクセスであるにもかかわらずL1Dミスが高い
⇒ L1Dキャッシュスラッシングが発生している

Cache

	L1I miss rate (/Effective instruction)	Load-store instruction	L1D miss	L1D miss rate (/Load-store instruction)	L1D miss demand rate (%) (/L1D miss)	L1D miss hardware prefetch rate (%) (/L1D miss)	L1D miss software prefetch rate (%) (/L1D miss)	L2 miss	L2 miss rate (/Load-store instruction)	L2 miss demand rate (%) (/L2 miss)	L2 miss hardware prefetch rate (%) (/L2 miss)	L2 miss software prefetch rate (%) (/L2 miss)
改善前	0.00	3.05E+09	1.30E+09	0.43	67.76%	32.24%	0.00%	8.35E+03	0.00	86.64%	19.76%	0.00%

Fortran ダミー配列によるパディング（ソースチューニング）

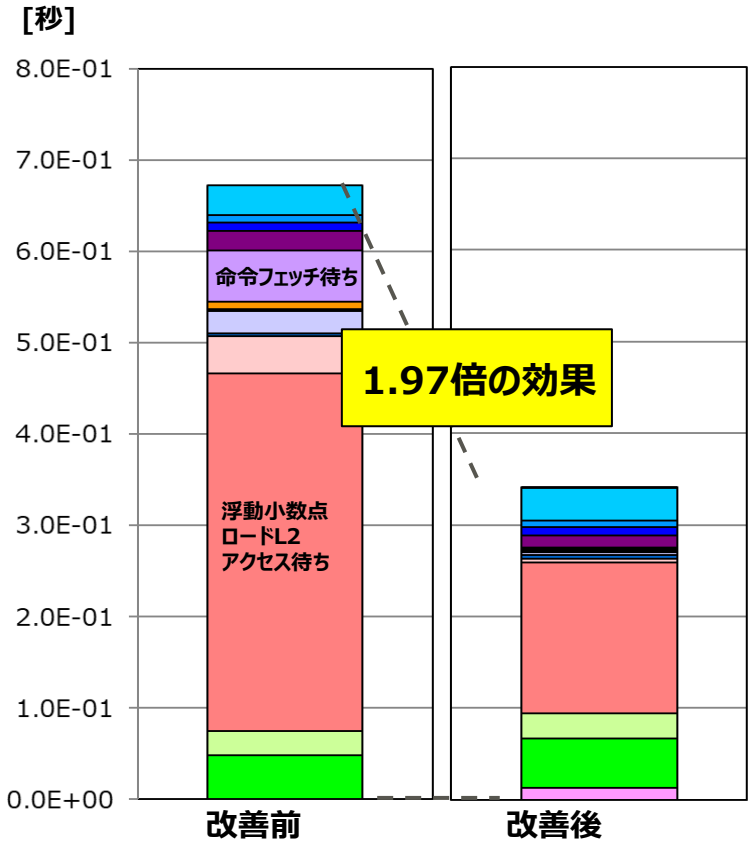
FUJITSU

配列間にダミー配列を追加して16KB境界からずらしたため、L1Dキャッシュスラッシングが回避されました。その結果、浮動小数点ロードL2アクセス待ちが改善されました。

改善後ソース

```
1              parameter(n=256,m=256)
2              real*8  a(n, m),dummy1(64),b(n,m),dummy2(64),&
                       c(n,m),dummy3(64),d(n,m),dummy4(64)
3              real*8  e(n, m),dummy5(64),f(n,m),dummy6(64),&
                       g(n,m),dummy7(64),h(n,m)
4              character (1),parameter :: null0=z'00'
:
28   1  s   s      call sub()
:
35             subroutine sub()
:
               <<< Loop-information Start >>>
               <<<  [PARALLELIZATION]
               <<<    Standard iteration count: 433
               <<<  [OPTIMIZATION]
               <<<    COLLAPSED
               <<<    SIMD(VL: 8)
               <<<    SOFTWARE PIPELINING(IPC: 2.66, ITR: 104, MVE: 7)
               <<< Loop-information  End >>>
40   1  pp   v     do j = 1 , m
               <<< Loop-information Start >>>
               <<<  [OPTIMIZATION]
               <<<    COLLAPSED
               <<< Loop-information  End >>>
41   2  p          do i = 1 , n
42   2  p   v       a(i, j) = b(i, j) + c(i, j) + d(i, j) + e(i, j) + f(i,j) + g(i,j) + h(i,j)
43   2  p   v      enddo
44   1  p         enddo
```

配列間にダミー配列を追加することで16KB境界からずらす



Cache

L1Dミス、L1DミスDM率が改善した

	Load-store instruction	L1D miss	L1D miss rate (/Load-store instruction)	L1D miss demand rate (%) (/L1D miss)
改善前	3.05E+09	1.30E+09	0.43	67.76%
改善後	2.79E+09	6.06E+08	0.22	31.03%

C/C++ ダミー配列によるパディング（改善前）

FUJITSU

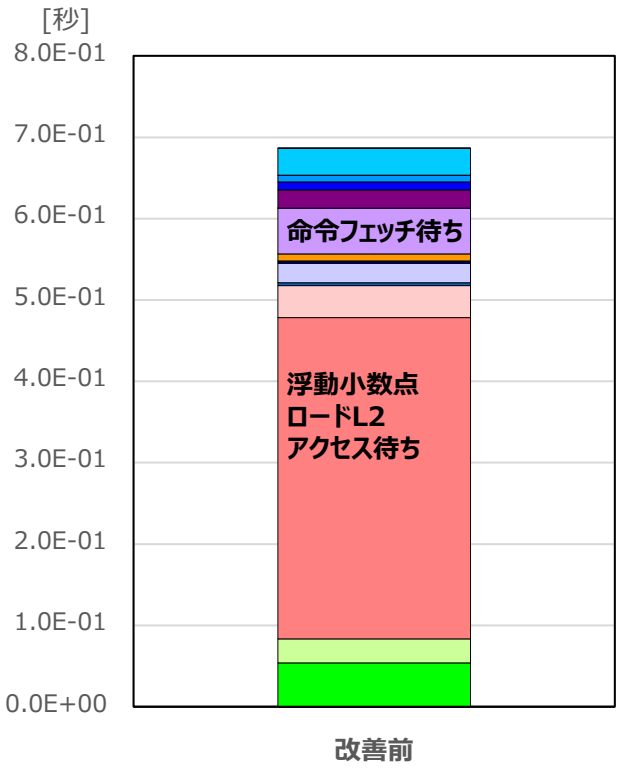
それぞれの配列が16KB境界にあるため、L1Dキャッシュスラッシングが発生します。
そのため、浮動小数点ロードL2アクセス待ちが多くなっています。

改善前ソース

配列宣言
double a[256][256], b[256][256], c[256][256], d[256][256], e[256][256], f[256][256], g[256][256], h[256][256];

配列のサイズ256×256×8B=
32×16KB(16KB境界)

```
35          void sub(void){
36            int i, j;
37
38          #pragma omp parallel for collapse(2)
            <<< Loop-information Start >>>
            <<<  [OPTIMIZATION]
            <<<    SIMD(VL: 8)
            <<<    SOFTWARE PIPELINING(IPC: 2.66, ITR: 104, MVE: 7, POL: S)
            <<<    PREFETCH(HARD) Expected by compiler :
            <<<      b, c, f, e, g, h, d, a
            <<< Loop-information  End >>>
39  p     v    for (j = 0; j < m; j++){
40  p     v      for (i = 0; i < n; i++){
41  p     v        a[j][i] = b[j][i] + c[j][i] + d[j][i] + e[j][i] + f[j][i] + g[j][i] + h[j][i];
42  p     v      }
43  p     v    }
44          }
```



配列が連続アクセスであるにもかかわらずL1Dミスが高い
⇒ L1Dキャッシュスラッシングが発生している

Cache	L1I miss rate (/Effective instruction)	Load-store instruction	L1D miss	L1D miss rate (/Load-store instruction)	L1D miss demand rate (%) (/L1D miss)	L1D miss hardware prefetch rate (%) (/L1D miss)	L1D miss software prefetch rate (%) (/L1D miss)	L2 miss	L2 miss rate (/Load-store instruction)	L2 miss demand rate (%) (/L2 miss)	L2 miss hardware prefetch rate (%) (/L2 miss)	L2 miss software prefetch rate (%) (/L2 miss)
改善前	0.00	3.12E+09	1.33E+09	0.43	68.30%	31.70%	0.00%	2.34E+04	0.00	46.43%	66.98%	0.00%

C/C++ ダミー配列によるパディング（ソースチューニング）

FUJITSU

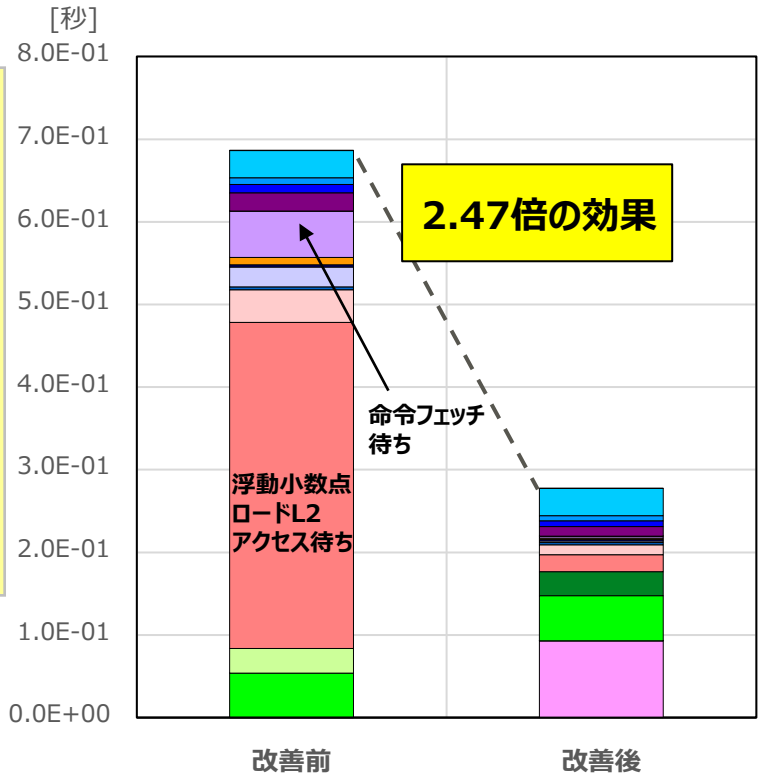
配列間にダミー配列を追加して16KB境界からずらしたため、L1Dキャッシュスラッシングが回避されました。その結果、浮動小数点ロードL2アクセス待ちが改善されました。

改善後ソース

```
36          void sub(void){
37            int i, j;
38
39          #pragma omp parallel for collapse(
            <<< Loop-information Start >>>
            <<<  [OPTIMIZATION]
            <<<    SIMD(VL: 8)
            <<<    SOFTWARE PIPELINING(IPC:
            <<<    PREFETCH(HARD) Expected b
            <<<      b, c, f, e, g, h, d, a
            <<< Loop-information  End >>>
40  p     v    for (j = 0; j < m; j++){
41  p     v      for (i = 0; i < n; i++){
42  p     v        a[j][i] = b[j][i] + c[j][i] + d[j][i] + e[j][i] + f[j][i] + g[j][i] + h[j][i];
43  p     v      }
44  p     v    }
```

配列a,b,c,d,e,f,g,hの宣言間にダミー配列double型の64要素の配列(例: dummy[64])宣言を追加することで16KB境界からずらす

配列宣言
double a[256][256],
dummy1[64], b[256][256],
dummy2[64], c[256][256],
dummy3[64], d[256][256],
dummy4[64], e[256][256],
dummy5[64], f[256][256],
dummy6[64], g[256][256],
dummy7[64], h[256][256];



L1Dミス、L1Dミスdm率が改善した

Cache	Load-store instruction	L1D miss	L1D miss rate (/Load-store instruction)	L1D miss demand rate (%) (/L1D miss)
改善前	3.12E+09	1.33E+09	0.43	68.30%
改善後	2.67E+09	4.56E+08	0.17	8.37%

Fortran ダミー配列によるパディングの効果（翻訳オプションチューニング）

FUJITSU

以下の翻訳オプション(Fortran固有)を指定することで、ソースチューニングと同等の効果を得ることができます。

翻訳オプション	機能説明
-Kcommonpad[=*N*] (4≦*N*≦2,147,483,644)	データキャッシュの使用効率をあげるために共通ブロック内の各変数の領域の間に、隙間をつくることを指定します。 *N* を省略した場合は、コンパイラが自動的に最良な値を決定します。

■使用例（改善前ソース時）

```
$ frtpx -Kfast,parallel sample.f90 -Kcommonpad=512
```

- 対象配列を自動選出→パディングを適用
- 注意事項
  - 分割コンパイルする場合は、共通ブロックを含むファイルに対して翻訳オプション-Kcommonpadを指定した場合、同じ名前の共通ブロックを含む他のファイルに対しても、翻訳オプション-Kcommonpadを指定しなければなりません。
  - 複数のファイルに対して翻訳オプション-Kcommonpad=*N*を指定して翻訳する場合、*N*の値は同じでなければなりません。
  - 同じ共通ブロック名に対し、その要素を変えて使用している場合、翻訳オプション-Kcommonpadを指定するとプログラムが正しく実行されないことがあります。

FUJITSU

ダミー配列によるパディング（サイズが異なる配列）

- サイズが異なる配列同士の競合について
- ダミー配列によるパディング（サイズが異なる配列：改善前）
- ダミー配列によるパディングの効果（サイズが異なる配列：ソースチューニング）

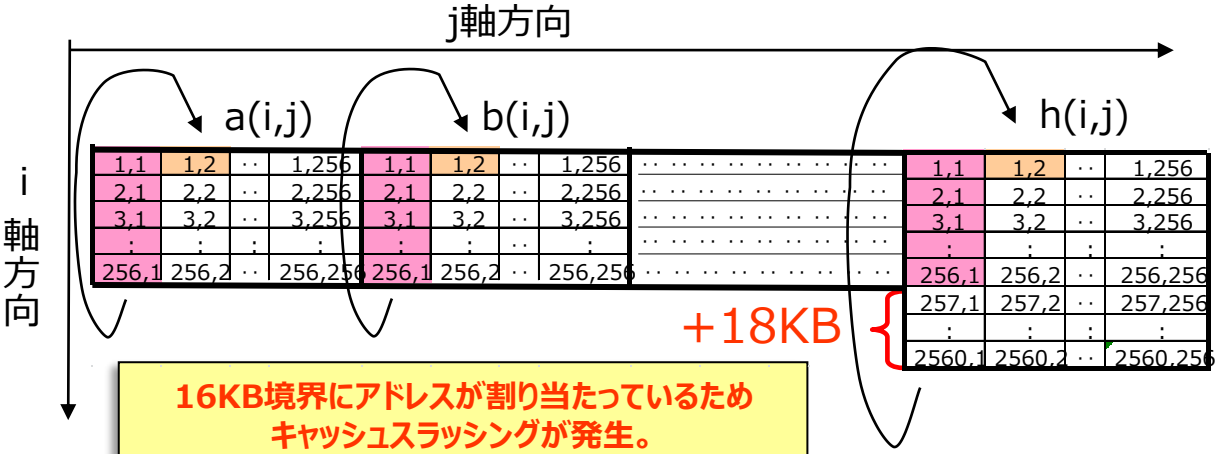
サイズが異なる配列同士の競合について（1/2）

FUJITSU

一般的にサイズの異なる配列は、定常的にキャッシュスラッシングは発生しません。

ソース例

```
parameter(n=256,m=256)
parameter(k=2560,l=256)

real*8  a(n,m), b(n,m), c(n,m),
        d(n,m), e(n,m), f(n,m),
        g(n,m), h(k,l)

common /test/a,b,c,d,e,f,g,h
do j = 1 , m
  do i = 1 , n
    a(i, j) = b(i, j) + c(i, j) + d(i, j) +
              e(i, j) + f(i, j) + g(i, j) +
              h(i, j)
  enddo
enddo
```



16KB境界にアドレスが割り当たっているため
キャッシュスラッシングが発生。

配列a(1,1)のアドレスを0番地と仮定すると
配列a(1,1)のアドレスは0番地(16KB×0)
配列b(1,1)のアドレスは256×256×8番地(16KB×32)
：　　：　　：
配列h(1,1)のアドレスは256×256×8×7番地(16KB×224)

2次元目がカウントアップされると
配列a(1,2)のアドレスはa(1,1)のアドレス+256×8B番地
配列b(1,2)のアドレスはb(1,1)のアドレス+256×8B番地
：　　：　　：
配列h(1,2)のアドレスはh(1,1)のアドレス+2560×8B番地

256×8B+18KB

一般的にサイズの異なる配列は、
定常的にキャッシュスラッシングは発生しない

配列a,b,c,d,e,f,gは16KB境界を維持するが、
配列hのアドレスは16KB境界からずれる。

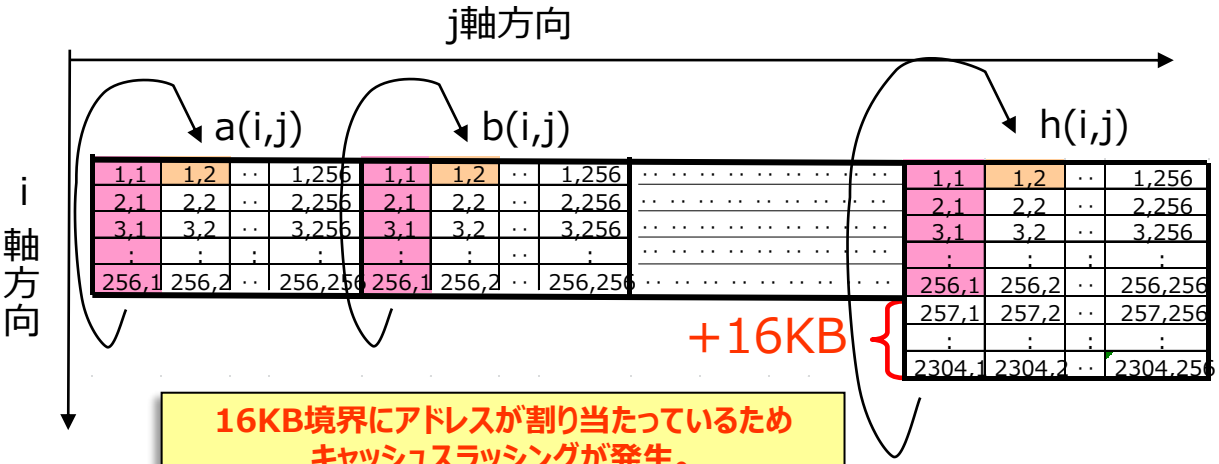
サイズが異なる配列同士の競合について (2/2)

FUJITSU

サイズの異なる配列でも配列サイズによっては16KB境界を維持するため、定常的にキャッシュスラッシングを発生します。

ソース例

```
parameter(n=256,m=256)
parameter(k=2304,l=256)

real*8  a(n,m), b(n,m), c(n,m),
        d(n,m), e(n,m), f(n,m),
        g(n,m), h(k,l)

common /test/a,b,c,d,e,f,g,h
do j = 1 , m
  do i = 1 , n
    a(i, j) = b(i, j) + c(i, j) + d(i, j) +
              e(i, j) + f(i, j) + g(i, j) +
              h(i, j)
  enddo
enddo
```



16KB境界にアドレスが割り当たっているため
キャッシュスラッシングが発生。

配列a(1,1)のアドレスを0番地と仮定すると
配列a(1,1)のアドレスは0番地(16KB×0)
配列b(1,1)のアドレスは256×256×8番地(16KB×32)
: : :
配列h(1,1)のアドレスは256×256×8×7番地(16KB×224)

2次元目がカウントアップされると
配列a(1,2)のアドレスはa(1,1)のアドレス+256×8B番地
配列b(1,2)のアドレスはb(1,1)のアドレス+256×8B番地
: : :
配列h(1,2)のアドレスはh(1,1)のアドレス+2304×8B番地

256×8B+16KB

配列hを含む全てのサイズの配列に対して
スラッシング対策が必要

配列a,b,c,d,e,f,g,h全て16KB境界を維持する。

Fortran ダミー配列によるパディング（サイズが異なる配列：改善前）

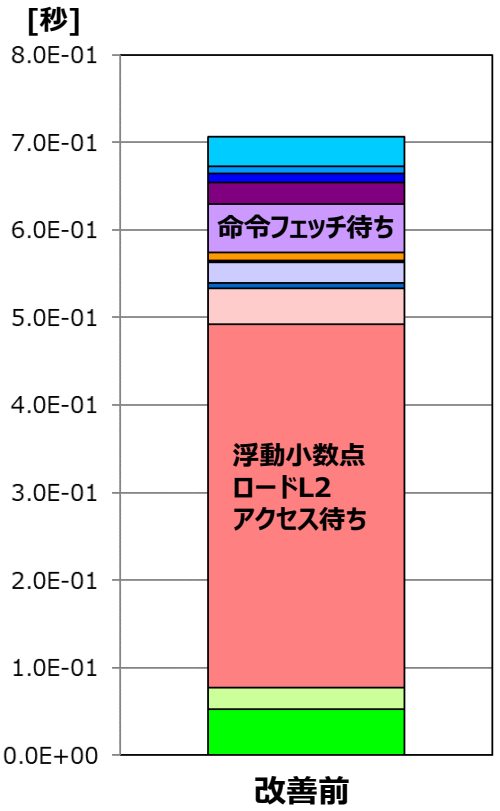
FUJITSU

それぞれの配列が16KB境界にあるためL1Dキャッシュスラッシングが発生します。
そのため、浮動小数点ロードL2アクセス待ちが多くなっています。

改善前ソース

```
52                integer k,l,n,m
53                parameter(n=256,m=256)
54                parameter(k=2304,l=256)
55
56                real*8  a(n,m), b(n,m), c(n,m), d(n,m), &
                  e(n,m), f(n,m), g(n,m), h(k,l)

57                common /test/a,b,c,d,e,
                <<< Loop-information Start
                <<<  [PARALLELIZATION]
                <<<    Standard iteration co
                <<< Loop-information  End >>>
58    1  pp         do j = 1 , m
                <<< Loop-information Start >>>
                <<<  [OPTIMIZATION]
                <<<    SIMD(VL: 8)
                <<<    SOFTWARE PIPELINING(IPC: 2.83, ITR: 112,
                                           MVE: 13, POL: S)
                <<< Loop-information  End >>>
59    2   p   v       do i = 1 , n
60    2   p   v         a(i, j) = b(i, j) + c(i, j) + d(i, j) + e(i, j) +
                            f(i, j) + g(i, j) + h(i, j)
61    2   p   v       enddo
62    1   p         enddo
```

2次元目がカウントアップされても各配列間は16KBを維持



配列が連続アクセスであるにもかかわらずL1Dミスが高い
⇒ L1Dキャッシュスラッシングが発生している

	L1I miss rate (/Effective instruction)	Load-store instruction	L1D miss	L1D miss rate (/Load-store instruction)	L1D miss demand rate (%) (/L1D miss)	L1D miss hardware prefetch rate (%) (/L1D miss)	L1D miss software prefetch rate (%) (/L1D miss)	L2 miss	L2 miss rate (/Load-store instruction)	L2 miss demand rate (%) (/L2 miss)	L2 miss hardware prefetch rate (%) (/L2 miss)	L2 miss software prefetch rate (%) (/L2 miss)
改善前	0.00	3.00E+09	1.39E+09	0.46	70.14%	29.86%	0.00%	3.34E+04	0.00	34.30%	77.94%	0.00%

Fortran ダミー配列によるパディング（サイズが異なる配列：ソースチューニング）

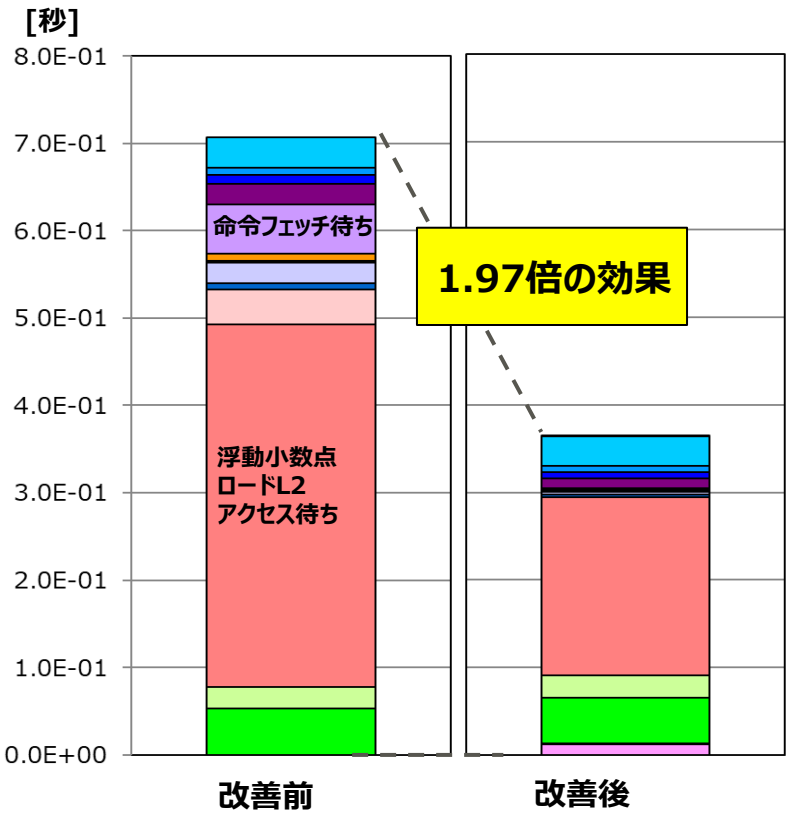
FUJITSU

配列間にダミー配列を追加することで16KB境界からずれたため、L1Dキャッシュスラッシングが回避されました。その結果、浮動小数点ロードL2アクセス待ちが改善されました。

改善後ソース

```
52              integer k,l,n,m
53              parameter(n=256,m=256)
54              parameter(k=2304,l=256)
55
56              real*8  a(n,m),dummy1(64),b(n,m),dummy2(64), &
57                    c(n,m),dummy3(64),d(n,m),dummy4(64), &
58                    e(n,m),dummy5(64),f(n,m),dummy6(64), &
59                    g(n,m),dummy7(64),h(k,l)
60              common /test/a,dummy1,b,dummy2,c,dummy3,d,dummy4,e,
                  dummy5,f,dummy6,g,dummy7,h
61
              <<< Loop-information Start >>>
              <<<  [PARALLELIZATION]
              <<<    Standard iteration count: 2
              <<< Loop-information  End >>>
62    1 pp        do j = 1 , m
              <<< Loop-information Start >>>
              <<<  [OPTIMIZATION]
              <<<    SIMD(VL: 8)
              <<<    SOFTWARE PIPELINING(IPC: 2.83, ITR: 112, MVE: 13, POL: S)
              <<< Loop-information  End >>>
63    2  p  v      do i = 1 , n
64    2  p  v        a(i, j) = b(i, j) + c(i, j) + d(i, j) + e(i, j) + f(i, j) + g(i, j) + h(i, j)
65    2  p  v      enddo
66    1  p       enddo
```

配列間にダミーの配列を挟む



L1Dミス、L1Dミスdm率が改善した。

	L1I miss rate (/Effective instruction)	Load-store instruction	L1D miss	L1D miss rate (/Load-store instruction)	L1D miss demand rate (%) (/L1D miss)	L1D miss hardware prefetch rate (%) (/L1D miss)	L1D miss software prefetch rate (%) (/L1D miss)	L2 miss	L2 miss rate (/Load-store instruction)	L2 miss demand rate (%) (/L2 miss)	L2 miss hardware prefetch rate (%) (/L2 miss)	L2 miss software prefetch rate (%) (/L2 miss)
改善前	0.00	3.00E+09	1.39E+09	0.46	70.14%	29.86%	0.00%	3.34E+04	0.00	34.30%	77.94%	0.00%
改善後	0.00	2.57E+09	6.90E+08	0.27	38.70%	61.31%	0.00%	2.54E+04	0.00	29.53%	80.62%	0.00%

C/C++ ダミー配列によるパディング（サイズが異なる配列：改善前）

FUJITSU

それぞれの配列が16KB境界にあるためL1Dキャッシュスラッシングが発生します。
そのため、浮動小数点ロードL2アクセス待ちが多くなっています。

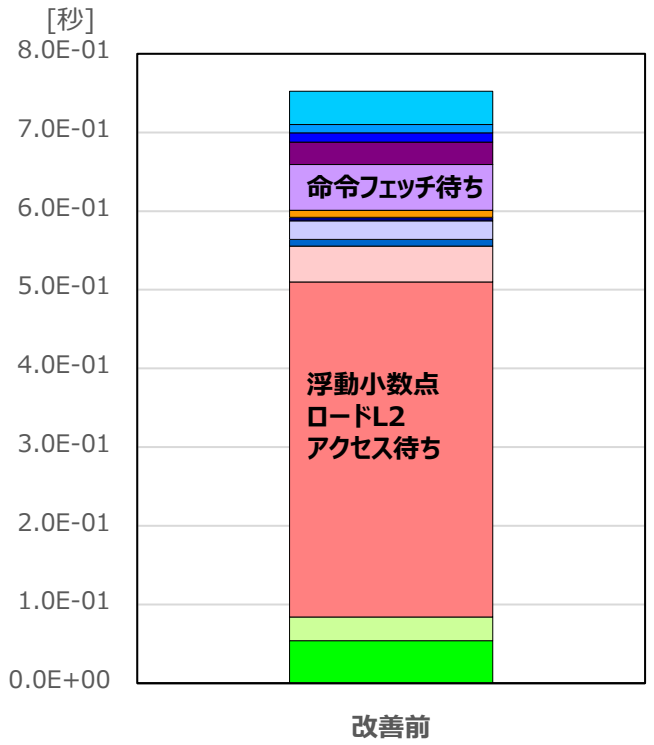
改善前ソース

```
24          void sub(void){
25            int i, j;
26
27          #pragma omp parallel for
            <<< Loop-information Start >>>
            <<<  [OPTIMIZATION]
            <<<    PREFETCH(HARD) Expected by compiler :
            <<<      b, c, f, e, g, h, d, a
            <<< Loop-information  End >>>
28   p        for (j = 0; j < m; j++){
            <<< Loop-information Start >>>
            <<<  [OPTIMIZATION]
            <<<    SIMD(VL: 8)
            <<<    SOFTWARE PIPELINING(IPC: 2.66, ITR: 104, MVE: 13, POL: S)
            <<<    PREFETCH(HARD) Expected by compiler :
            <<<      b, c, f, e, g, h, d, a
            <<< Loop-information  End >>>
29   p    v     for (i = 0; i < n; i++){
30   p    v       a[j][i] = b[j][i] + c[j][i] + d[j][i] + e[j][i] + f[j][i] + g[j][i] + h[j][i];
31   p    v     }
32   p         }
33          }
```

配列宣言
double a[256][256], b[256][256], c[256][256], d[256][256], e[256][256], f[256][256], g[256][256], h[256][2304];

配列aのサイズ256×256×8B=32×16KB(16KB境界)

2次元目がカウントアップされても各配列間は16KBを維持



配列が連続アクセスであるにもかかわらずL1Dミスが高い
⇒ L1Dキャッシュスラッシングが発生している

Cache	L1I miss rate (/Effective instruction)	Load-store instruction	L1D miss	L1D miss rate (/Load-store instruction)	L1D miss demand rate (%) (/L1D miss)	L1D miss hardware prefetch rate (%) (/L1D miss)	L1D miss software prefetch rate (%) (/L1D miss)	L2 miss	L2 miss rate (/Load-store instruction)	L2 miss demand rate (%) (/L2 miss)	L2 miss hardware prefetch rate (%) (/L2 miss)	L2 miss software prefetch rate (%) (/L2 miss)
改善前	0.00	3.62E+09	1.38E+09	0.38	71.21%	28.78%	0.01%	1.04E+05	0.00	91.27%	10.77%	0.00%

C/C++ ダミー配列によるパディング（サイズが異なる配列：ソースチューニング）

FUJITSU

配列間にダミー配列を追加することで16KB境界からずれたため、L1Dキャッシュスラッシングが回避されました。その結果、浮動小数点ロードL2アクセス待ちが改善されました。

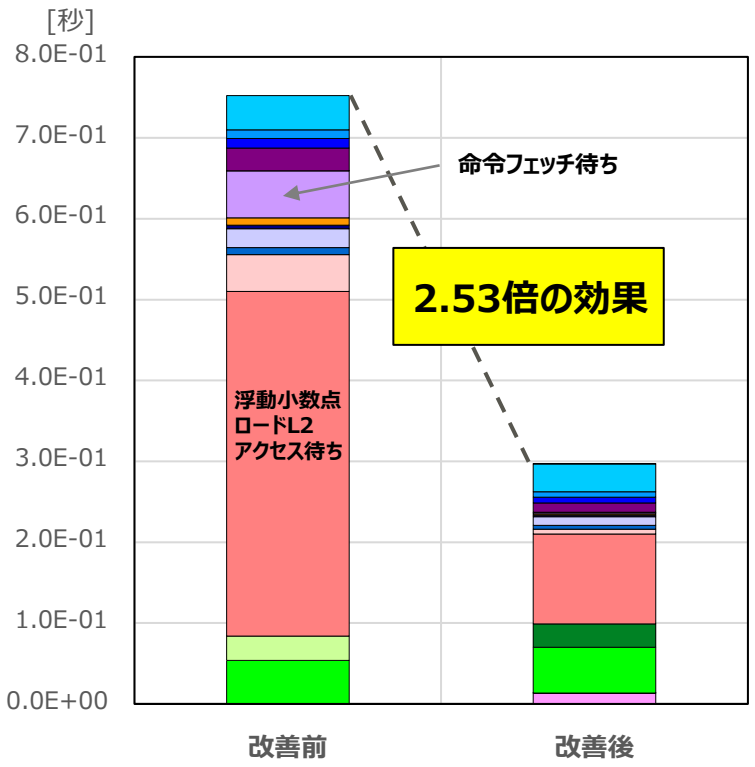
改善後ソース

```
25          void sub(void){
26            int i, j;
27
28          #pragma omp parallel for
            <<< Loop-information Start >>>
            <<<  [OPTIMIZATION]
            <<<    PREFETCH(HARD) Expected
            <<<      b, c, f, e, g, h, d, a
            <<< Loop-information  End >>>
29  p         for (j = 0; j < m; j++){
            <<< Loop-information Start >>>
            <<<  [OPTIMIZATION]
            <<<    SIMD(VL: 8)
            <<<    SOFTWARE PIPELINING(IPC
            <<<    PREFETCH(HARD) Expected
            <<<      b, c, f, e, g, h, d, a
            <<< Loop-information  End >>>
30  p     v     for (i = 0; i < n; i++){
31  p     v       a[j][i] = b[j][i] + c[j][i] + d[j][i] + e[j][i] + f[j][i] + g[j][i] + h[j][i];
32  p     v     }
33  p         }
34          }
```

配列a,b,c,d,e,f,g,hの宣言間にダミー配列double型の64要素の配列(例: dummy[64])宣言を追加することで16KB境界からずらす

配列宣言

```
double a[256][256],
dummy1[64], b[256][256],
dummy2[64], c[256][256],
dummy3[64], d[256][256],
dummy4[64], e[256][256],
dummy5[64], f[256][256],
dummy6[64], g[256][256],
dummy7[64], h[256][2304];
```



L1Dミス、L1Dミスdm率が改善した。

Cache	L1I miss rate (/Effective instruction)	Load-store instruction	L1D miss	L1D miss rate (/Load-store instruction)	L1D miss demand rate (%) (/L1D miss)	L1D miss hardware prefetch rate (%) (/L1D miss)	L1D miss software prefetch rate (%) (/L1D miss)	L2 miss	L2 miss rate (/Load-store instruction)	L2 miss demand rate (%) (/L2 miss)	L2 miss hardware prefetch rate (%) (/L2 miss)	L2 miss software prefetch rate (%) (/L2 miss)
改善前	0.00	3.62E+09	1.38E+09	0.38	71.21%	28.78%	0.01%	1.04E+05	0.00	91.27%	10.77%	0.00%
改善後	0.00	2.69E+09	4.81E+08	0.18	17.67%	82.33%	0.00%	1.20E+05	0.00	54.33%	62.57%	0.00%

FUJITSU

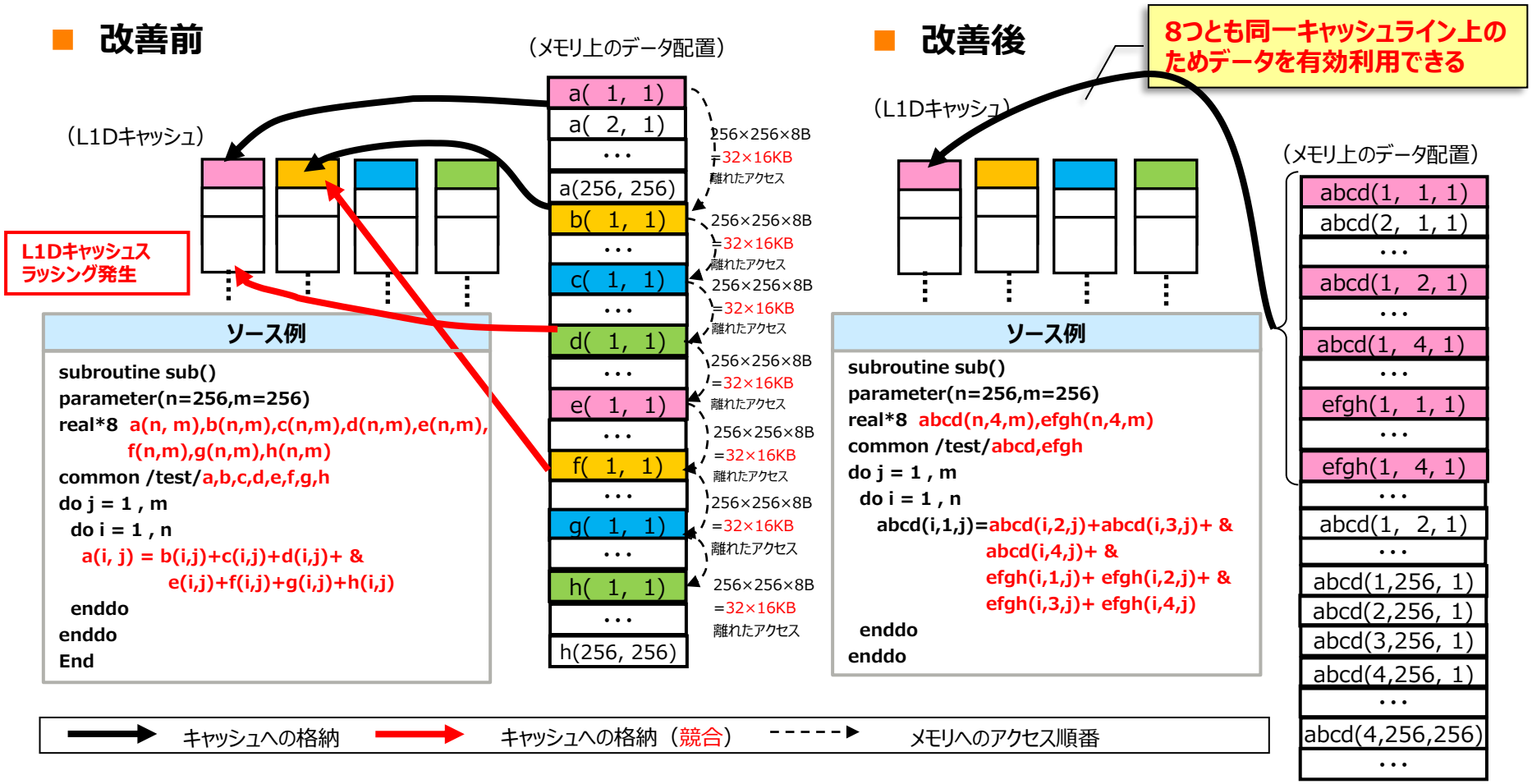
配列マージ（スラッシングの改善）

- 配列マージとは
- 配列マージ（スラッシングの改善）（改善前）
- 配列マージ（スラッシングの改善）（ソースチューニング）
- 配列マージ（翻訳オプションチューニング）

配列マージとは

配列マージとは、複数の配列を1つの配列とするチューニングです。

次の例のように配列数を削減することで、データを同一キャッシュライン上に載せることができます。



Fortran 配列マージ（スラッシングの改善）（改善前）

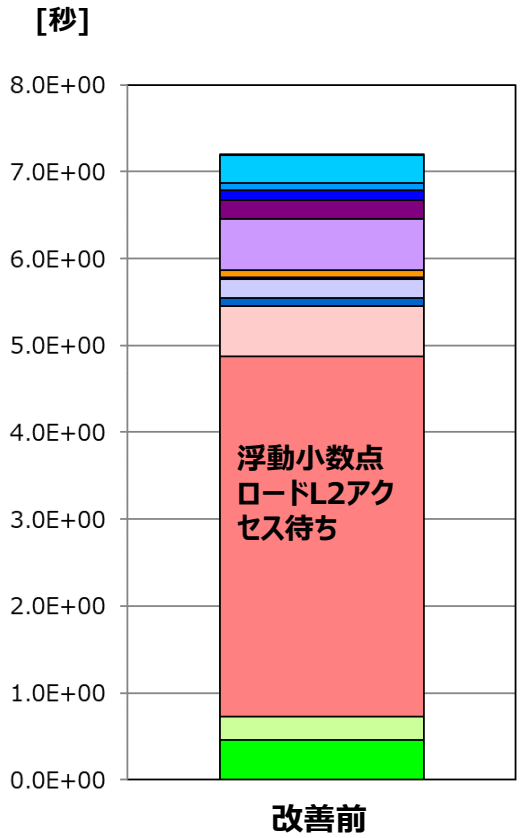
FUJITSU

それぞれの配列が16KB境界にあるためL1Dキャッシュスラッシングが発生します。
そのため、浮動小数点ロードL2アクセス待ちが多くなっています。

改善前ソース

```
40                parameter(n=256,m=256)
41                real*8  a(n, m),b(n,m),c(n,m),d(n,m),&
                          e(n,m),f(n,m),g(n,m),h(n,m)
42                common /test/a,b,c,d,e,f,g,h
                <<< Loop-information Start >>>
                <<<  [PARALLELIZATION]
                <<<    Standard iteration count: 433
                <<<  [OPTIMIZATION]
                <<<    COLLAPSED
                <<<    SIMD(VL: 8)
                <<<    SOFTWARE PIPELINING(IPC: 2.66, ITR: 104,
                                           MVE: 7, POL: S)
                <<<    PREFETCH(HARD) Expected by compiler :
                <<<      b, c, f, e, g, h, d, a
                <<< Loop-information  End >>>
43    1  pp   v      do j = 1 , m
                <<< Loop-information Start >>>
                <<<  [OPTIMIZATION]
                <<<    COLLAPSED
                <<< Loop-information  End >>>
44    2  p           do i = 1 , n
45    2  p    v       a(i,j)=b(i,j)+c(i,j)+d(i,j)+e(i,j)+f(i,j)+g(i,j)+h(i,j)
46    2  p    v      enddo
47    1  p          enddo
```

配列サイズ
256×256×8B=
32×16KB
(16KB境界)



配列が連続アクセスであるにもかかわらずL1Dミスが高い
⇒ L1Dキャッシュスラッシングが発生している

Cache	L1I miss rate (/Effective instruction)	Load-store instruction	L1D miss	L1D miss rate (/Load-store instruction)	L1D miss demand rate (%) (/L1D miss)	L1D miss hardware prefetch rate (%) (/L1D miss)	L1D miss software prefetch rate (%) (/L1D miss)	L2 miss	L2 miss rate (/Load-store instruction)	L2 miss demand rate (%) (/L2 miss)	L2 miss hardware prefetch rate (%) (/L2 miss)	L2 miss software prefetch rate (%) (/L2 miss)
改善前	0.00	3.40E+10	1.33E+10	0.39	68.21%	31.79%	0.00%	1.12E+04	0.00	72.56%	37.92%	0.00%

Fortran 配列マージ（スラッシングの改善）（ソースチューニング）

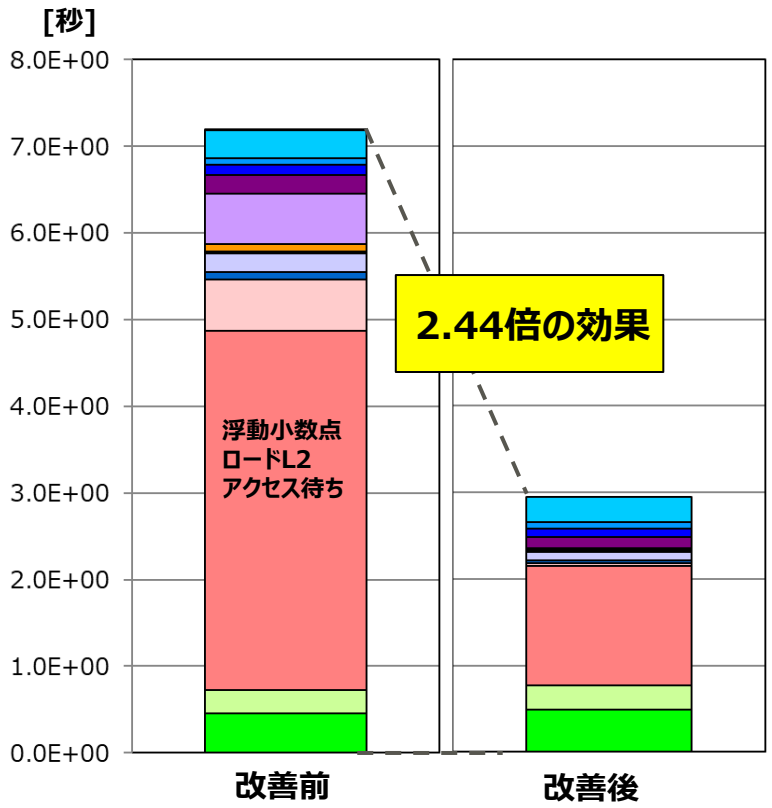
FUJITSU

配列マージを行うことでストリーム数が8から2となり、L1Dキャッシュスラッシングを回避しました。その結果、浮動小数点ロードL2アクセス待ちが改善されました。

改善後ソース（ソースチューニング）

```
44              parameter(n=256,m=256)
45              real*8  abcd(n,4,m),efgh(n,4,m)
46              common /test/abcd,efgh
              <<< Loop-information Start >>>
              <<<  [PARALLELIZATION]
              <<<    Standard iteration count: 2
              <<<  [OPTIMIZATION]
              <<<    PREFETCH(HARD) Expected by compiler :
              <<<      efgh, abcd
              <<< Loop-information  End >>>
47    1  pp         do j = 1,m
              <<< Loop-information Start >>>
              <<<  [OPTIMIZATION]
              <<<    SIMD(VL: 8)
              <<<    SOFTWARE PIPELINING(IPC: 2.28, ITR: 112,
                                         MVE: 13, POL: S)
              <<<    PREFETCH(HARD) Expected by compiler :
              <<<      efgh, abcd
              <<< Loop-information  End >>>
48    2  p    v       do i = 1,n
49    2  p    v         abcd(i,1,j)=abcd(i,2,j)+abcd(i,3,j)+abcd(i,4,j)+&
50    2                      efgh(i,1,j)+efgh(i,2,j)+efgh(i,3,j)+efgh(i,4,j)
51    2  p    v       enddo
52    1  p         enddo
```

8つの配列を
4つずつマージする



L1Dミスが減少した

	L1I miss rate (/Effective instruction)	Load-store instruction	L1D miss	L1D miss rate (/Load-store instruction)	L1D miss demand rate (%) (/L1D miss)	L1D miss hardware prefetch rate (%) (/L1D miss)	L1D miss software prefetch rate (%) (/L1D miss)	L2 miss	L2 miss rate (/Load-store instruction)	L2 miss demand rate (%) (/L2 miss)	L2 miss hardware prefetch rate (%) (/L2 miss)	L2 miss software prefetch rate (%) (/L2 miss)
改善前	0.00	3.40E+10	1.33E+10	0.39	68.21%	31.79%	0.00%	1.12E+04	0.00	72.56%	37.92%	0.00%
改善後	0.00	2.54E+10	4.40E+09	0.17	84.98%	15.03%	0.00%	1.70E+04	0.00	69.93%	46.61%	0.00%

C/C++ 配列マージ（スラッシングの改善）（改善前）

FUJITSU

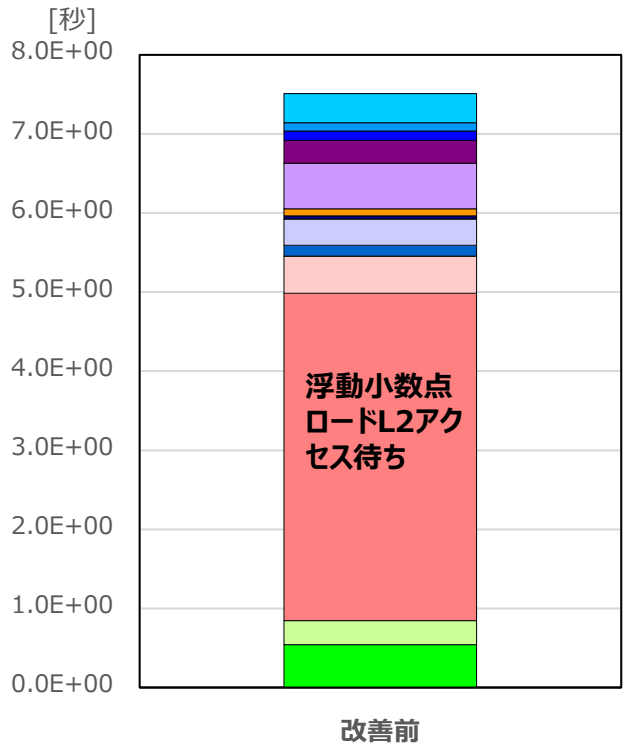
それぞれの配列が16KB境界にあるためL1Dキャッシュスラッシングが発生します。
そのため、浮動小数点ロードL2アクセス待ちが多くなっています。

改善前ソース

```
31          void sub(void)
32          {
33             int i,j;
34          #pragma omp parallel for
        <<< Loop-information Start >>>
          <<<  [OPTIMIZATION]
          <<<    PREFETCH(HARD) Expected b
          <<<      b, c, f, e, g, h, d, a
          <<< Loop-information  End >>>
35   p          for(j=0;j<m;j++){
          <<< Loop-information Start >>>
          <<<  [OPTIMIZATION]
          <<<    SIMD(VL: 8)
          <<<    SOFTWARE PIPELINING(IPC: 2.66, ITR: 104, MVE: 7, POL: S)
          <<<    PREFETCH(HARD) Expected by compiler :
          <<<      b, c, f, e, g, h, d, a
          <<< Loop-information  End >>>
36   p     v       for(i=0;i<n;i++){
37   p     v         a[j][i]=b[j][i]+c[j][i]+d[j][i]+e[j][i]+f[j][i]+g[j][i]+h[j][i];
38   p     v       }
39   p           }
40          }
```

配列宣言
double a[256][256], b[256][256], c[256][256], d[256][256], e[256][256], f[256][256], g[256][256], h[256][256];

配列aのサイズ256×256×8B=32×16KB(16KB境界)



配列が連続アクセスであるにもかかわらずL1Dミスが高い
⇒ L1Dキャッシュスラッシングが発生している

Cache	L1I miss rate (/Effective instruction)	Load-store instruction	L1D miss	L1D miss rate (/Load-store instruction)	L1D miss demand rate (%) (/L1D miss)	L1D miss hardware prefetch rate (%) (/L1D miss)	L1D miss software prefetch rate (%) (/L1D miss)	L2 miss	L2 miss rate (/Load-store instruction)	L2 miss demand rate (%) (/L2 miss)	L2 miss hardware prefetch rate (%) (/L2 miss)	L2 miss software prefetch rate (%) (/L2 miss)
改善前	0.00	3.32E+10	1.36E+10	0.41	69.48%	30.52%	0.00%	1.16E+05	0.00	86.19%	26.53%	0.00%

C/C++ 配列マージ（スラッシングの改善）（ソースチューニング）

FUJITSU

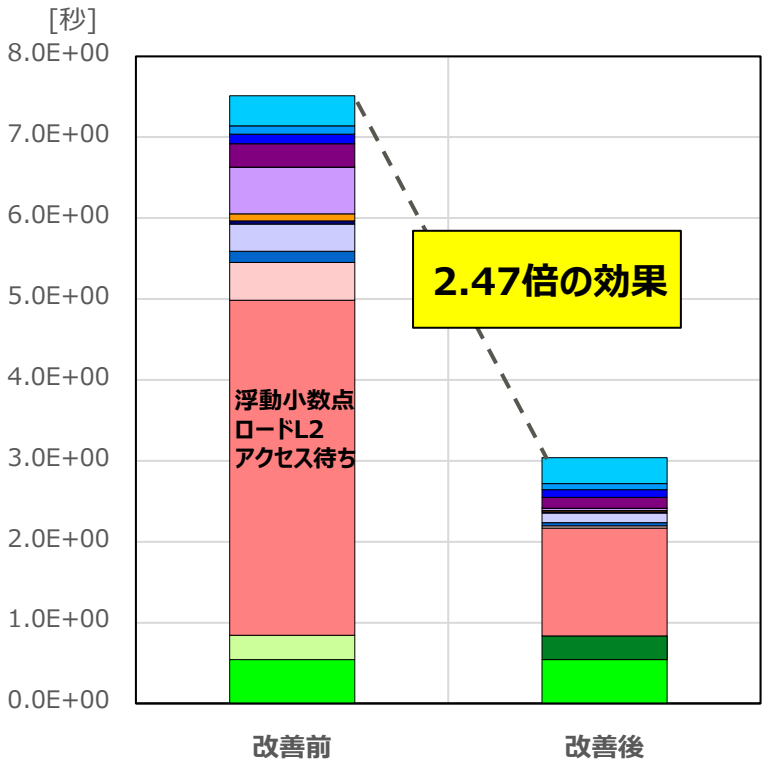
配列マージを行うことでストリーム数が8から2となり、L1Dキャッシュスラッシングを回避しました。その結果、浮動小数点ロードL2アクセス待ちが改善されました。

改善後ソース（ソースチューニング）

```
31          void sub(void)
32          {
33             int i,j;
34          #pragma omp parallel for
          <<< Loop-information Start >>>
          <<<  [OPTIMIZATION]
          <<<    PREFETCH(HARD) Expected by compiler :
          <<<      efgh, abcd
          <<< Loop-information  End >>>
35   p          for(j=0;j<m;j++){
          <<< Loop-information Start >>>
          <<<  [OPTIMIZATION]
          <<<    SIMD(VL: 8)
          <<<    SOFTWARE PIPELINING(IPC: 2.28, ITR: 112, MVE: 8, POL: S)
          <<<    PREFETCH(HARD) Expected by compiler :
          <<<      efgh, abcd
          <<< Loop-information  End >>>
36   p     v       for(i=0;i<n;i++){
37   p     v         abcd[j][0][i]=abcd[j][1][i]+abcd[j][2][i]+abcd[j][3][i]+
                     efgh[j][0][i]+efgh[j][1][i]+efgh[j][2][i]+efgh[j][3][i];
38   p     v       }
39   p           }
40          }
```

配列宣言
double abcd[256][4][256], efgh[256][4][256];

8つの配列を4つずつマージする



L1Dミスが減少した

Cache	L1I miss rate (/Effective instruction)	Load-store instruction	L1D miss	L1D miss rate (/Load-store instruction)	L1D miss demand rate (%) (/L1D miss)	L1D miss hardware prefetch rate (%) (/L1D miss)	L1D miss software prefetch rate (%) (/L1D miss)	L2 miss	L2 miss rate (/Load-store instruction)	L2 miss demand rate (%) (/L2 miss)	L2 miss hardware prefetch rate (%) (/L2 miss)	L2 miss software prefetch rate (%) (/L2 miss)
改善前	0.00	3.32E+10	1.36E+10	0.41	69.48%	30.52%	0.00%	1.16E+05	0.00	86.19%	26.53%	0.00%
改善後	0.00	2.63E+10	4.40E+09	0.17	90.03%	9.98%	-0.01%	4.45E+04	0.00	80.52%	35.47%	0.00%

配列マージ（翻訳オプションチューニング）

FUJITSU

以下の翻訳オプション（Fortran固有）を指定することで、ソースチューニングと同等の効果を得ることができます。

翻訳オプション	機能説明
-Karray_merge_common [=name]	共通ブロック内の複数配列をマージすることを指示します。nameには共通ブロック名を指定できます。Nameを省略した場合、すべて名前付き共通ブロック内の配列が対象となります。
-Karray_merge_local	ローカル配列の複数配列をマージすることを指示します。 -Karray_merge_local_size=1000000も同時に有効になります。
-Karray_merge_local_size=N (2≦*N*≦2,147,483,647)	マージ対象とするローカル配列の大きさが、Nバイト以上であることを指示します。-Karray_merge_localオプションが有効な場合に意味があります。
-Karray_merge	本オプションは、-Karray_merge_localおよび-Karray_merge_commonオプションが指定された場合と等価です。

■ 使用例（改善前ソース時）

```
$frtpx -Kfast,parallel sample.f90 -Karray_merge_common
```

■ 注意事項

- 対象配列を使うソースすべてにオプション指定が必要です。
- マージの効果はプログラムによって異なります。
- 正しくない使い方をした場合には計算結果が異なることがあります。
- デバッグオプション(-gおよび-H)と併用はできません。

FUJITSU

ループ分割（スラッシングの改善）

- ループ分割とは
- ループ分割（スラッシングの改善）（改善前）
- ループ分割の効果（スラッシングの改善）（ソースチューニング）
- ループ分割の効果（最適化制御行チューニング）

ループ分割とは

- ループ分割とは、主に以下の目的でループを複数の小さなループに分割する手法です。
  - ソフトウェアパイプライニングの促進
  - キャッシュメモリ利用効率の改善
  - レジスタ不足の解消

ループ分割を行うことによりループ内でアクセスする配列数を削減し、ソフトウェアパイプラインの促進や、キャッシュスラッシングを回避できる場合があります。

ただし、分割の仕方によってはキャッシュ上のデータを有効利用できなくなる場合があるため注意が必要です。

改善前ソース例

```
parameter(n=65536)
real*8  a(n),b(n),c(n),d(n),e(n),f(n),
g(n),h(n)
common /com/a,b,c,d,e,f,g,h
do i=1,n
   a(i) =  s / b(i)
   c(i) =  s / d(i)
   e(i) =  s / f(i)
   g(i) =  s / h(i)
enddo
```

スラッシングが発生

改善後ソース例

```
parameter(n=65536)
real*8  a(n),b(n),c(n),d(n),e(n),f(n),
g(n),h(n)
common /com/a,b,c,d,e,f,g,h
!OCL LOOP_NOFUSION
do i=1,n
   a(i) =  s / b(i)
   c(i) =  s / d(i)
enddo
do i=1,n
   e(i) =  s / f(i)
   g(i) =  s / h(i)
enddo
```

ループ融合を抑止

ループを分割
スラッシングを抑止

Fortran ループ分割（スラッシングの改善）（改善前）

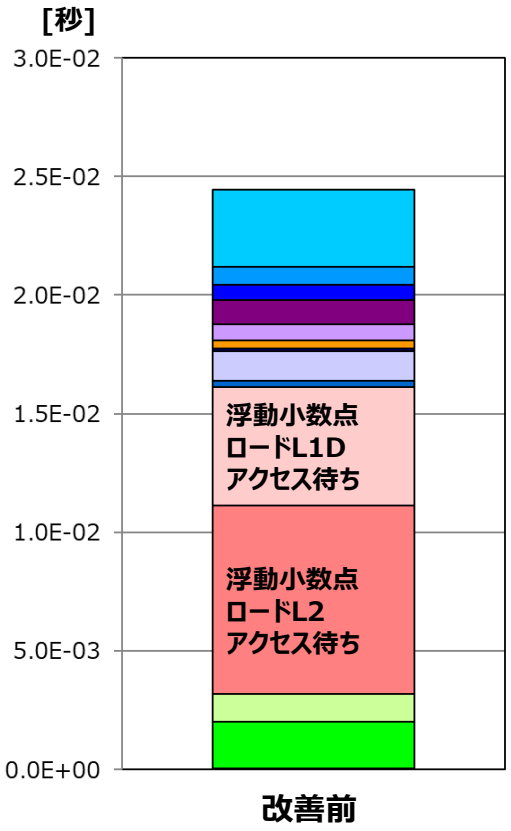
FUJITSU

それぞれの配列が16KB境界にあるためL1Dキャッシュスラッシング発生します。
そのため、浮動小数点ロードアクセス待ちが多くなっています。

改善前ソース

```
46                parameter(n=65536)
47                real*8  a(n),b(n),c(n),d(n),e(n),f(n),g(n),h(n)
48                common /com/a,b,c,d,e,f,g,h
49
                <<< Loop-information Start >>>
                <<<  [PARALLELIZATION]
                <<<    Standard iteration count:
                <<<  [OPTIMIZATION]
                <<<    SIMD(VL: 8)
                <<<    SOFTWARE PIPELINING(IPC: 1.90, ITR: 56,
                                           MVE: 4, POL: S)
                <<<    PREFETCH(HARD) Expected by compiler :
                <<<      f, d, b, h, g, e, c, a
                <<< Loop-information  End >>>
50    1  pp   v        do i=1,n
51    1   p   v          a(i) =  s / b(i)
52    1   p   v          c(i) =  s / d(i)
53    1   p   v          e(i) =  s / f(i)
54    1   p   v          g(i) =  s / h(i)
55    1   p   v        enddo
```

配列サイズ
65536×8B=
32×16KB
(16KB境界)



配列が連続アクセスであるにも関わらず
L1Dミス率が高く、L1ミスdm率が高い
⇒ L1Dキャッシュスラッシングが発生している

Cache

	L1I miss rate (/Effective instruction)	Load-store instruction	L1D miss	L1D miss rate (/Load-store instruction)	L1D miss demand rate (%) (/L1D miss)	L1D miss hardware prefetch rate (%) (/L1D miss)	prefetch rate (%) (/L1D miss)	L2 miss	(/Load-store instruction)	demand rate (%) (/L2 miss)	prefetch rate (%) (/L2 miss)	prefetch rate (%) (/L2 miss)
改善前	0.00	1.25E+08	3.85E+07	0.31	57.48%	42.52%	0.01%	1.84E+04	0.00	31.49%	72.74%	0.00%

Fortran ループ分割の効果（スラッシングの改善）（ソースチューニング）

FUJITSU

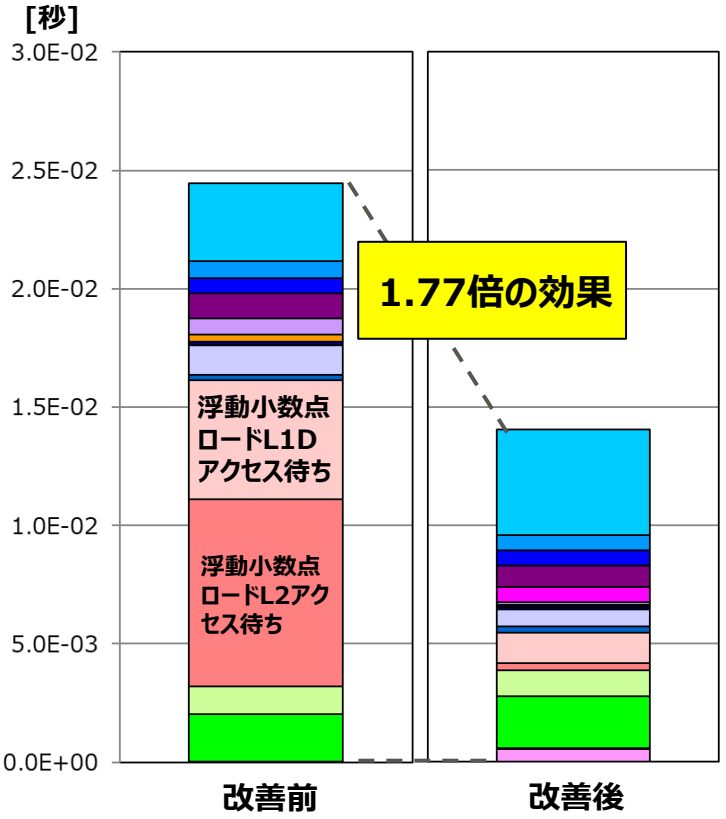
ループ分割を行うことでストリーム数が8から4となり、L1Dキャッシュスラッシングを回避しました。その結果、浮動小数点ロードL2アクセス待ちが改善されました。

改善後ソース

```
46                parameter(n=65536)
47                real*8  a(n),b(n),c(n),d(n),e(n),f(n),g(n),h(n)
48                common /com/a,b,c,d,e,f,g,h
49
50                !OCL LOOP_NOFUSION
                  <<< Loop-information Start >>>
                  <<<  [PARALLELIZATION]
                  <<<    Standard iteration cou
                  <<<  [OPTIMIZATION]
                  <<<    SIMD(VL: 8)
                  <<<    SOFTWARE PIPELINING(IPC: 2.36, ITR: 80,
                                                 MVE: 2, POL: S)
                  <<< Loop-information  End >>>
51    1  pp  2v       do i=1,n
52    1   p  2v         a(i) =  s / b(i)
53    1   p  2v         c(i) =  s / d(i)
54    1   p  2v       enddo
                  <<< Loop-information Start >>>
                  <<<  [PARALLELIZATION]
                  <<<    Standard iteration count: 411
                  <<<  [OPTIMIZATION]
                  <<<    SIMD(VL: 8)
                  <<<    SOFTWARE PIPELINING(IPC: 2.45, ITR: 80,
                                                 MVE: 2, POL: S)
                  <<< Loop-information  End >>>
55    1  pp  2v       do i=1,n
56    1   p  2v         e(i) =  s / f(i)
57    1   p  2v         g(i) =  s / h(i)
58    1   p  2v       enddo
```

ループ融合を抑止

ループ分割



L1Dミス、L1Dミスdm率が減少した。

	L1I miss rate (/Effective instruction)	Load-store instruction	L1D miss	L1D miss rate (/Load-store instruction)	L1D miss demand rate (%) (/L1D miss)	L1D miss hardware prefetch rate (%) (/L1D miss)	L1D miss software prefetch rate (%) (/L1D miss)
改善前	0.00	1.25E+08	3.85E+07	0.31	57.48%	42.52%	0.01%
改善後	0.00	1.10E+08	1.78E+07	0.16	8.37%	91.63%	0.00%

ループ分割（スラッシングの改善）（改善前）

FUJITSU

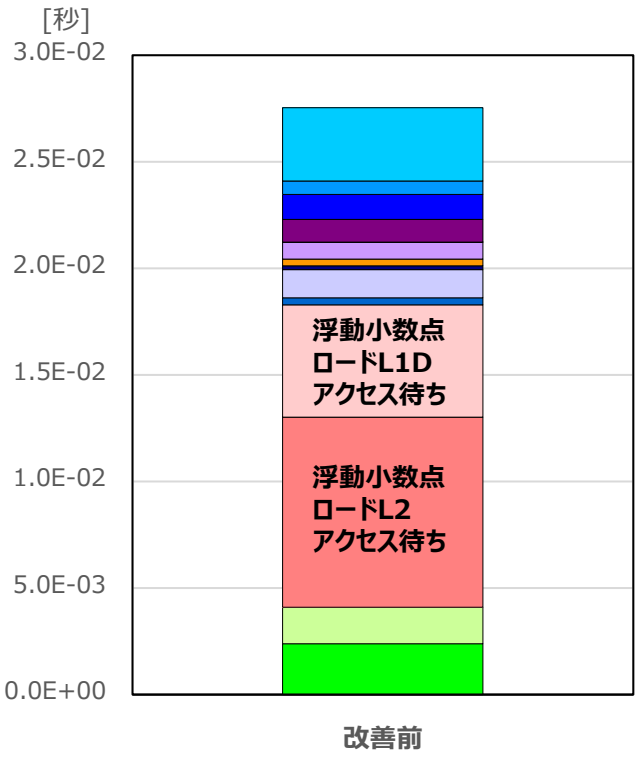
それぞれの配列が16KB境界にあるためL1Dキャッシュスラッシング発生します。
そのため、浮動小数点ロードアクセス待ちが多くなっています。

改善前ソース

```
45          void sub(double s)
46          {
47            int i;
48
49          #pragma omp parallel for
          <<< Loop-information Start >>>
        <<<  [OPTIMIZATION]
          <<<    SIMD(VL: 8)
          <<<    SOFTWARE PIPELINING(IPC: 1.90, ITR: 56,
                                     MVE: 4, POL: S)
          <<<    PREFETCH(HARD) Expected by compiler :
          <<<      f, d, b, h, g, e, c, a
          <<< Loop-information  End >>>
50  p     v    for(i=0;i<n; i++)
51  p     v    {
52  p     v      a[i] =  s / b[i];
53  p     v      c[i] =  s / d[i];
54  p     v      e[i] =  s / f[i];
55  p     v      g[i] =  s / h[i];
56  p     v    }
57
58            return;
59          }
```

配列宣言
double a[65536], b[65536], c[65536], d[65536], e[65536], f[65536], g[65536], h[65536];

配列a,b,c,d,e,f,g,h のサイズ65536 x 8B= 32×16KB(16KB境界)



配列が連続アクセスであるにも関わらず
L1Dミス率が高く、L1ミスdm率が高い
⇒ L1Dキャッシュスラッシングが発生している

Cache	L1I miss rate (/Effective instruction)	Load-store instruction	L1D miss	L1D miss rate (/Load-store instruction)	L1D miss demand rate (%) (/L1D miss)	L1D miss hardware prefetch rate (%) (/L1D miss)	prefetch rate (%) (/L1D miss)	L2 miss	(/Load-store instruction)	demand rate (%) (/L2 miss)	prefetch rate (%) (/L2 miss)	2 miss ftware prefetch rate (%) (/L2 miss)
改善前	0.00	1.17E+08	3.93E+07	0.34	58.44%	41.56%	0.00%	1.94E+04	0.00	16.93%	87.92%	0.00%

ループ分割の効果（スラッシングの改善）（ソースチューニング）

FUJITSU

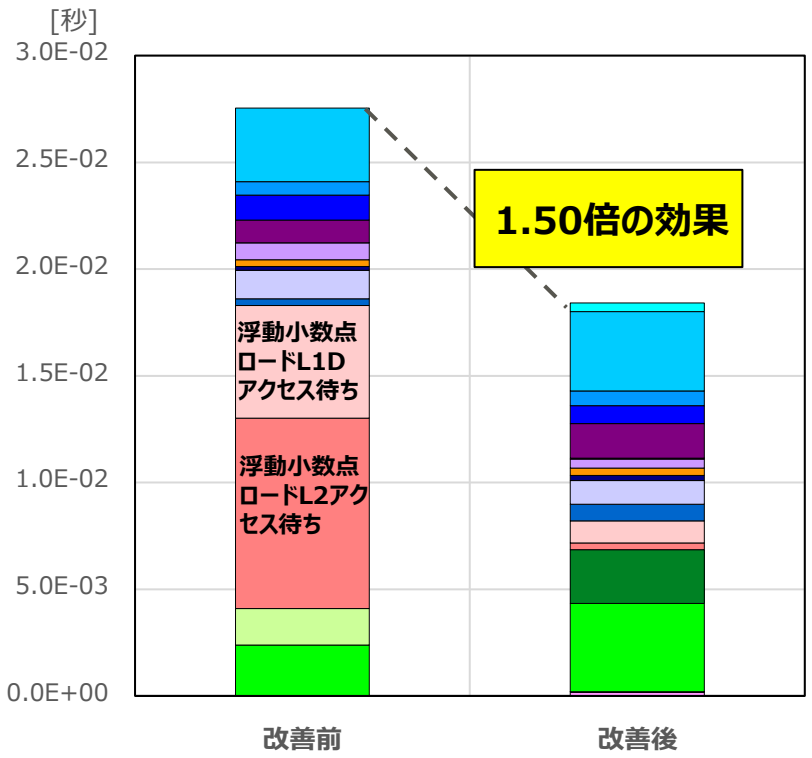
ループ分割を行うことでストリーム数が8から4となり、L1Dキャッシュスラッシングを回避しました。その結果、浮動小数点ロードL2アクセス待ちが改善されました。

改善後ソース

```
45          void sub(double s)
46          {
47            int i;
48
49            #pragma omp parallel for
            <<< Loop-information Start >>>
            <<<  [OPTIMIZATION]
          <<<    SIMD(VL: 8)
            <<<    SOFTWARE PIPELINING(IPC: 2.09, ITR: 64,
                          MVE: 2, POL: S)
            <<<    PREFETCH(HARD) Expected by compiler :
            <<<      d, b, c, a
            <<< Loop-information  End >>>
50  p   2v   for(i=0;i<n; i++)
51  p   2v   {
52  p   2v     a[i] =  s / b[i];
53  p   2v     c[i] =  s / d[i];
54  p   2v   }
55
56          #pragma omp parallel for
            <<< Loop-information Start >>>
            <<<  [OPTIMIZATION]
            <<<    SIMD(VL: 8)
            <<<    SOFTWARE PIPELINING(IPC: 2.09, ITR: 64,
                          MVE: 2, POL: S)
            <<<    PREFETCH(HARD) Expected by compiler :
            <<<      h, f, g, e
            <<< Loop-information  End >>>
57  p   2v   for(i=0;i<n; i++)
58  p   2v   {
59  p   2v     e[i] =  s / f[i];
60  p   2v     g[i] =  s / h[i];
61  p   2v   }
62            return;
63          }
```

配列宣言
double a[65536], b[65536], c[65536], d[65536], e[65536], f[65536], g[65536], h[65536];

ループ分割



L1Dミス、L1Dミスdm率が減少した。

Cache	L1I miss rate (/Effective instruction)	Load-store instruction	L1D miss	L1D miss rate (/Load-store instruction)	L1D miss demand rate (%) (/L1D miss)	L1D miss hardware prefetch rate (%) (/L1D miss)	L1D miss software prefetch rate (%) (/L1D miss)
改善前	0.00	1.17E+08	3.93E+07	0.34	58.44%	41.56%	0.00%
改善後	0.00	1.51E+08	1.86E+07	0.12	12.30%	87.72%	-0.02%

ループ分割の効果（最適化制御行チューニング）

FUJITSU

以下の最適化制御行を指定することで、ソースチューニングと同等の効果を得ることができます。

最適化指示子 (Fortran)	意味	指定可能な最適化行			
		プログラム単位	DOループ単位	文単位	配列代入文単位
FISSION_POINT[(n1)] (n1は1~6の10進数)	ループ内の指定された位置でループ分割することを指示します。最内ループから数えてn1重ネストされた多重ループを対象にループします。	×	×	○	×

最適化指示子 (C/C++のtradモードのみ)	意味	指定可能な最適化行			
		global行	procedure行	loop行	statement行
fission_point[(n1)] (n1は1~6の10進数)	ループ内の指定された位置でループ分割することを指示します。最内ループから数えてn1重ネストされた多重ループを対象にループします。	×	×	×	○

改善後ソース（最適化制御行チューニング） (Fortranの例)

```
46                parameter(n=65536)
47                real*8  a(n),b(n),c(n),d(n),e(n),f(n),g(n),h(n)
48                common /com/a,b,c,d,e,f,g,h
49
               <<< Loop-information Start >>>
               <<<  [PARALLELIZATION]
               <<<    Standard iteration count: 411
               <<<  [OPTIMIZATION]
               <<<    FISSION(num: 2)
               <<<    SIMD(VL: 8)
               <<<    SOFTWARE PIPELINING(IPC: 2.45, ITR: 80, MVE: 2, POL: S)
               <<<    PREFETCH(HARD) Expected by compiler :
               <<<      b, d, c, a, f, h, g, e
               <<< Loop-information  End >>>
50    1  pp   2v       do i=1,n
51    1   p   2v         a(i) =  s / b(i)
52    1   p   2v         c(i) =  s / d(i)
53    1            !ocl fission_point(1)
54    1   p   2v         e(i) =  s / f(i)
55    1   p   2v         g(i) =  s / h(i)
56    1   p   2v       enddo
57                  end subroutine sub
58
 Diagnostic messages: program name(sub)
   jwd8212o-i  "a.f90", line 50: ループを2分割しました。
```

ラージページ環境変数によるパディング

- XOS_MMM_L_FORCE_MMAP_THRESHOLDについて
- ラージページ環境変数によるパディング(改善前)
- ラージページ環境変数によるパディング(改善後)

XOS_MMM_L_FORCE_MMAP_THRESHOLDについて

環境変数名	指定値 (_付はdefault)	説明
XOS_MMM_L_FORCE_ MMAP_THRESHOLD	0 | 1	MALLOC_MMAP_THRESHOLD_(デフォルトは128MiB)以上のサイズのメモリ獲得時にmmap(2)を優先するかどうかの設定です。 「0」の場合、mmap(2) は優先しません。まずヒープ領域の空きを検索し、空きがあればヒープ領域の空きメモリを返します。ヒープ領域の空きが見つからないときにのみ mmap(2) でメモリを獲得します。「1」の場合、mmap(2) を優先します。ヒープ領域の空きは検索せず、(例え空きがあっても)mmap(2) でメモリを獲得します。

Fortran ラージページ環境変数によるパディング(改善前)

FUJITSU

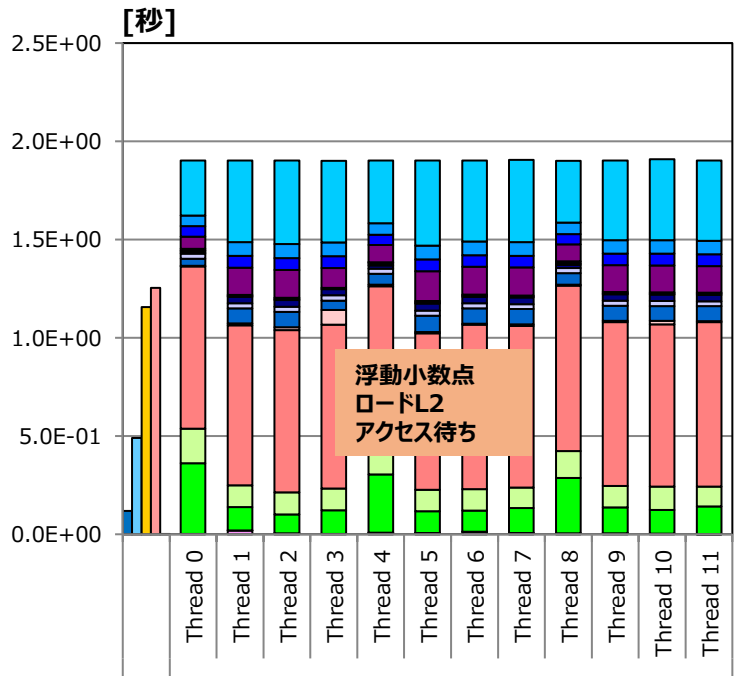
動的配列の各ストリームが16KB境界にあるためL1Dキャッシュスラッシングが発生します。そのため、浮動小数点ロードL2アクセス待ちが多くなっています。

改善前ソース

```
3                parameter(n=256,m=256)
4                real*8,allocatable ::a(:,:),b(:,:),c(:,:),d(:,:),e(:,:),
                                      f(:,:),g(:,:),h(:,:)
6                allocate(a(n,m))
7                allocate(b(n,m))
8                allocate(c(n,m))
9                allocate(d(n,m))
10                allocate(e(n,m))
11                allocate(f(n,m))
12                allocate(g(n,m))
13                allocate(h(n,m))
:                  ......
36    1   p         do j = 1 , m
                 <<<    SIMD(VL: 8)
                 <<<    SOFTWARE PIPELI
                             MVE: 2, POL: S
                 <<<    PREFETCH(HARD)
                 <<<      c, b, d, e, f, g, h, a
                 <<< Loop-information  End >>>
37    2   p   v       do i = 1 , n
38    2   p   v           a(i, j) = b(i, j) + c(i, j) + d(i, j) + e(i, j)
                                  + f(i ,j) + g(i ,j) + h(i ,j)
39    2   p   v       enddo
40    1   p         enddo
```

配列サイズ
256×256×8B=
32×16KB
(16KB境界)

同一サイズのストリーム



改善前

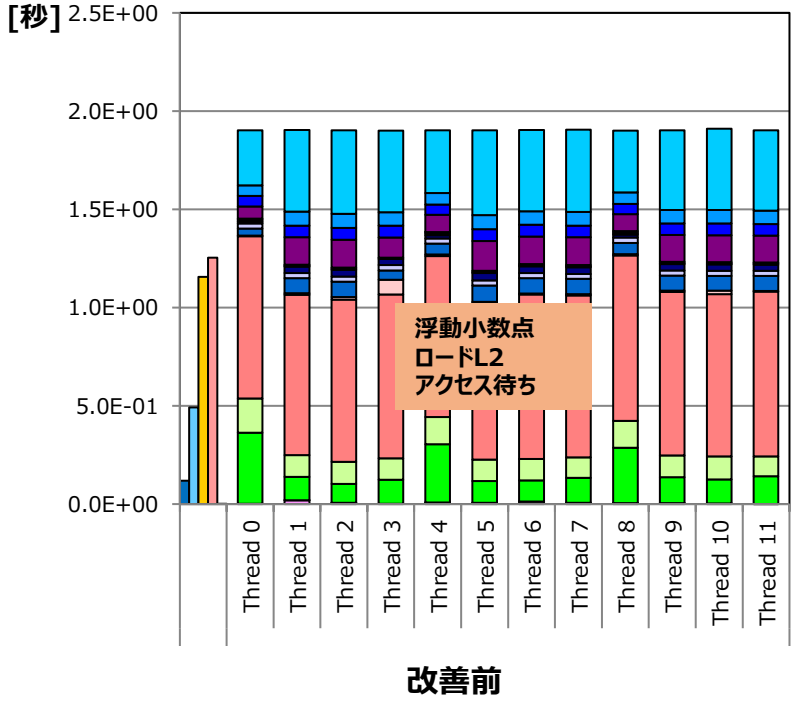
L1Dミスが高い
⇒ L1Dキャッシュスラッシングが発生している

Cache	L1I miss rate (/Effective instruction)	Load-store instruction	L1D miss	L1D miss rate (/Load-store instruction)	L1D miss demand rate (%) (/L1D miss)	L1D miss hardware prefetch rate (%) (/L1D miss)	L1D miss software prefetch rate (%) (/L1D miss)	L2 miss	L2 miss rate (/Load-store instruction)	L2 miss demand rate (%) (/L2 miss)	L2 miss hardware prefetch rate (%) (/L2 miss)	L2 miss software prefetch rate (%) (/L2 miss)
改善前	0.00	1.34.E+10	3.44.E+09	0.26	51.52%	48.28%	0.20%	1.61.E+03	0.00	73.10%	53.55%	0.00%

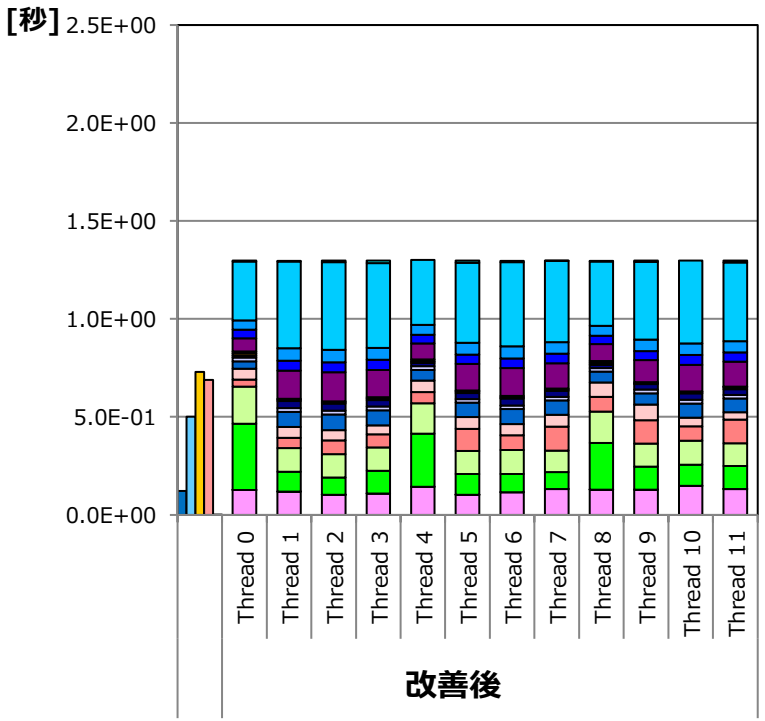
Fortran ラージページ環境変数によるパディング(改善後)

FUJITSU

環境変数MALLOC_MMAP_THRESHOLD_=204800を指定することで、各配列のアドレス配置がかわり、L1Dキャッシュスラッシングを回避しました。その結果、浮動小数点ロードL2アクセス待ちが改善されました。



改善前



改善後

Cache

L1Dミス、L1Dミスdm率が改善した

	L1I miss rate (/Effective instruction)	Load-store instruction	L1D miss	L1D miss rate (/Load-store instruction)	L1D miss demand rate (%) (/L1D miss)	L1D miss hardware prefetch rate (%) (/L1D miss)	L1D miss software prefetch rate (%) (/L1D miss)	L2 miss	L2 miss rate (/Load-store instruction)	L2 miss demand rate (%) (/L2 miss)	L2 miss hardware prefetch rate (%) (/L2 miss)	L2 miss software prefetch rate (%) (/L2 miss)
改善前	0.00	1.34.E+10	3.44.E+09	0.26	51.52%	48.28%	0.20%	1.61.E+03	0.00	73.10%	53.55%	0.00%
改善後	0.00	1.35.E+10	1.81.E+09	0.13	9.47%	90.51%	0.02%	2.43.E+03	0.00	68.31%	58.01%	0.00%

C/C++ ラージページ環境変数によるパディング(改善前)

FUJITSU

動的配列の各ストリームが16KB境界にあるためL1Dキャッシュスラッシングが発生します。そのため、浮動小数点ロードL2アクセス待ちが多くなっています。

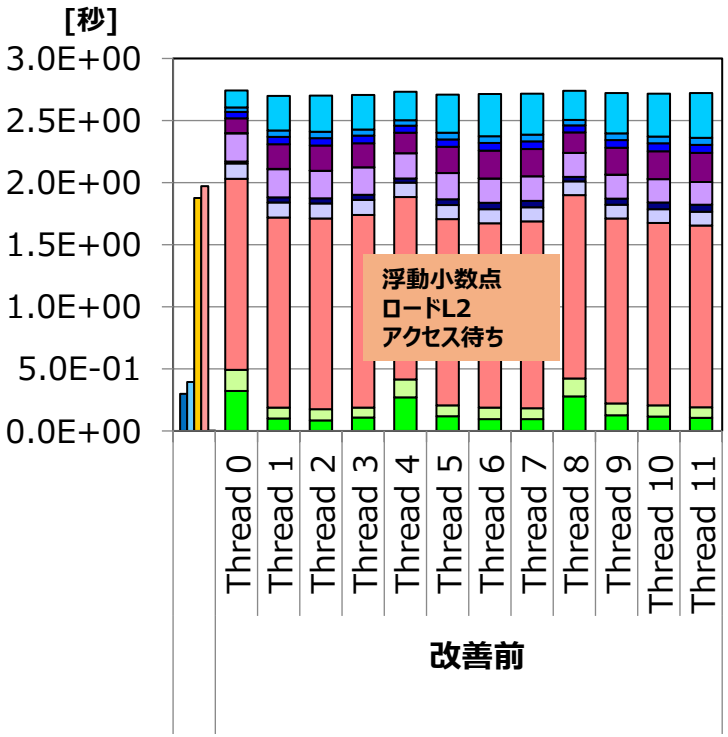
改善前ソース

```
62          void sub(int n, int m, double (* restrict a)[n], double (* restrict b)[n],
              double (* restrict c)[n], double (* restrict d)[n], double (* restrict e)[n],
              double (* restrict f)[n], double (* restrict g)[n], double (* restrict h)[n])
63          {
64            int i,j;
65          #pragma omp parallel for private(i)
          <<< Loop-information Start >>>
          <<<  [OPTIMIZATION]
          <<<    PREFETCH(HARD) Expected by
          <<<      (unknown)
          <<< Loop-information  End >>>
66  p         for(j = 0; j<m; j++)
67  p         {
          <<< Loop-information Start >>>
          <<<  [OPTIMIZATION]
          <<<    SIMD(VL: 8)
          <<<    SOFTWARE PIPELINING(IPC: 2.66, ITR: 104, MVE: 7, POL: S)
          <<<    PREFETCH(HARD) Expected by compiler :
          <<<      (unknown)
          <<< Loop-information  End >>>
68  p   v       for(i = 0; i<n; i++)
69  p   v       {
70  p   v         a[j][i] = b[j][i] + c[j][i] + d[j][i] + e[j][i] + f[j][i] + g[j][i] + h[j][i];
71  p   v       }
72  p         }
73            return;
74          }
```

配列宣言
double a[256][256], b[256][256], c[256][256], d[256][256], e[256][256], f[256][256], g[256][256], h[256][256];

配列aのサイズ256×256×8B=
32×16KB(16KB境界)

同一サイズのストリーム



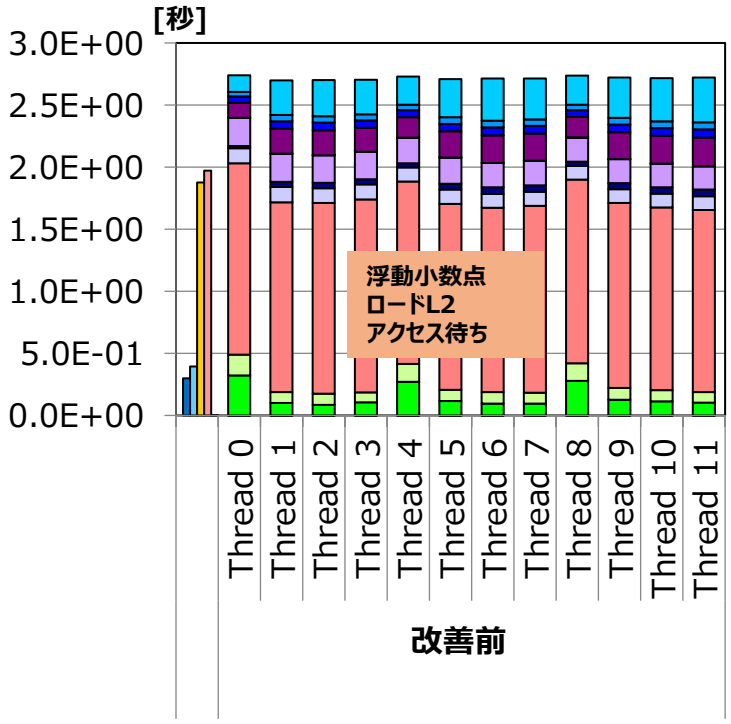
L1Dミスが高い
⇒ L1Dキャッシュスラッシングが発生している

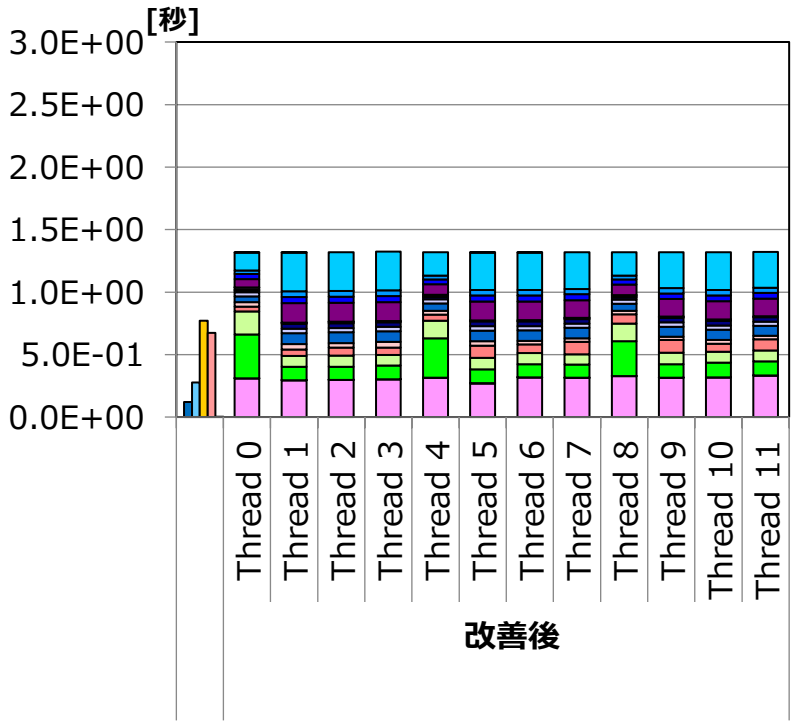
Cache	L1I miss rate (/Effective instruction)	Load-store instruction	L1D miss	L1D miss rate (/Load-store instruction)	L1D miss demand rate (%) (/L1D miss)	L1D miss hardware prefetch rate (%) (/L1D miss)	L1D miss software prefetch rate (%) (/L1D miss)	L2 miss	L2 miss rate (/Load-store instruction)	L2 miss demand rate (%) (/L2 miss)	L2 miss hardware prefetch rate (%) (/L2 miss)	L2 miss software prefetch rate (%) (/L2 miss)
改善前	0.00	1.39E+10	5.24E+09	0.38	69.38%	30.62%	0.01%	7.49E+04	0.00	68.31%	44.20%	0.00%

ラージページ環境変数によるパディング(改善後)

FUJITSU

環境変数MALLOC_MMAP_THRESHOLD_=204800を指定することで、各配列のアドレス配置がかわり、L1Dキャッシュスラッシングを回避しました。その結果、浮動小数点ロードL2アクセス待ちが改善されました。





L1Dミス、L1Dミスdm率が改善した

Cache	L1I miss rate (/Effective instruction)	Load-store instruction	L1D miss	L1D miss rate (/Load-store instruction)	L1D miss demand rate (%) (/L1D miss)	L1D miss hardware prefetch rate (%) (/L1D miss)	L1D miss software prefetch rate (%) (/L1D miss)	L2 miss	L2 miss rate (/Load-store instruction)	L2 miss demand rate (%) (/L2 miss)	L2 miss hardware prefetch rate (%) (/L2 miss)	L2 miss software prefetch rate (%) (/L2 miss)
改善前	0.00	1.39E+10	5.24E+09	0.38	69.38%	30.62%	0.01%	7.49E+04	0.00	68.31%	44.20%	0.00%
改善後	0.00	1.25E+10	1.80E+09	0.14	9.31%	90.69%	0.00%	2.98E+04	0.00	49.45%	57.41%	0.00%

FUJITSU

演算待ち（SIMD化の促進）

- ループピーリング
- 定義関係が明らかでないループ
- ポインタ変数が含まれるループ

ループピーリング

- ループピーリングとは
- ループピーリング（改善前）
- ループピーリング（ソースチューニング）

ループピーリングとは

ループピーリングとはループ内の一部の処理をループから切り離す処理です。

ループ内でデータの依存関係がある場合、SIMD化や効果的なソフトウェアパイプライニングが行われない場合があります。

以下のような、ループ内に依存関係がある場合は、その部分だけループから切り離すループピーリングを行うことで対処することができます。

改善前ソース

```
do j = 1, n
   a[j] = a[j] + a[1]
enddo
```

改善後ソース

```
a[1] = a[1] + a[1]
do j = 2, n
   a[j] = a[j] + a[1]
enddo
```

左の例では、j=1の時に定義されたa[1]が2≤j≤nで使用されており、部分的な依存関係があります。
右の例のようにj=1の場合だけピーリングすることでループ内の依存関係を取り除くことができます。

Fortran ループピーリング（改善前）

FUJITSU

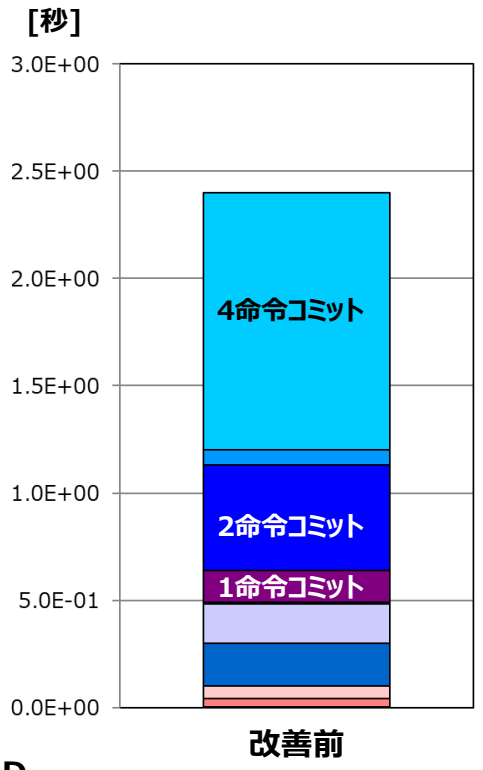
配列a について、i = 1 の時に定義したものを i = 2 以降で参照するデータ依存関係があるため、SIMD化が効果的に行われていません。

改善前ソース

```
                <<< Loop-information Start >>>
                <<<  [OPTIMIZATION]
                <<<    PREFETCH(HARD) Expe
                <<<      b, (unknown)
                <<< Loop-information  End >
 8    2   p             do j = 1, m
                <<< Loop-information Start >
                <<<  [OPTIMIZATION]
                <<<    SIMD(VL: 8)
                <<<    SOFTWARE PIPELINING(IPC: 1.38, ITR: 144,
                                           MVE: 5, POL: S)
                <<<    PREFETCH(HARD) Expected by compiler :
                <<<      a, b, (unknown)
                <<< Loop-information  End >>>
 9    3   p   2m            do i = 1, n
10    3   p   2m             a(i,j) = c0 + a(1,j)*(c1 + b(i,j)*(c2 + b(i,j)*(c3 + b(i,j)*
11    3               &          (c4 + b(I,j)*(c5 + b(I,j)*(c6 + b(I,j)*(c7 + b(I,j)*
12    3               &          (c8 + b(i,j)*c9)))))))
13    3   p   v           end do
14    2   p             end do
```

一部しかSIMD化が行われていない

i=1の時にロード側配列a とストア側配列a で依存関係がある。



SIMD

	Effective instruction	SIMD instruction rate (%) (/Effective instruction)
改善前	1.49E+11	14.61%

Fortran ループピーリング（ソースチューニング）

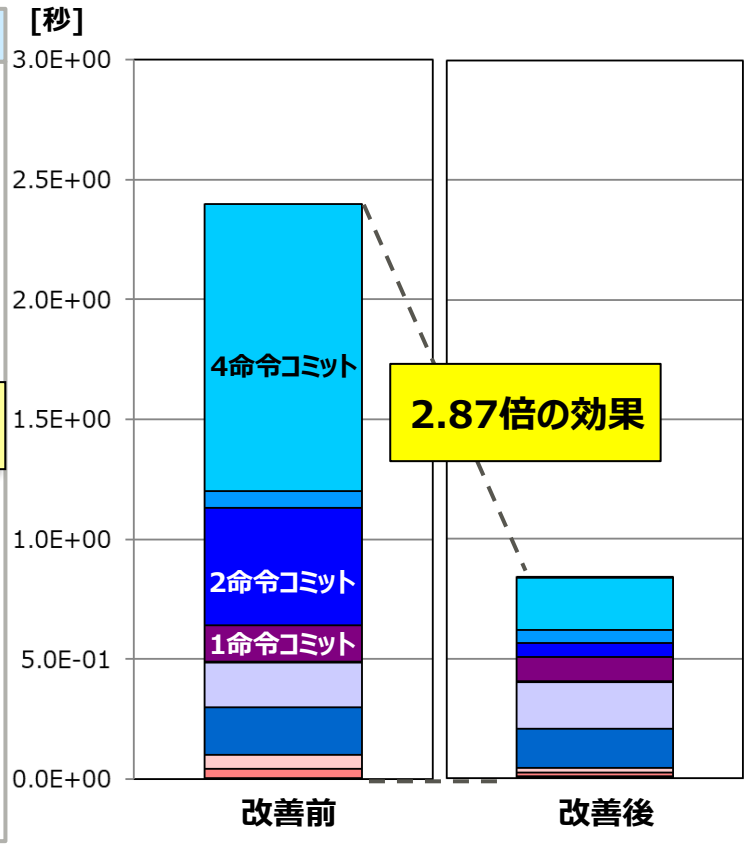
1回分のループピーリングを行うことで、SIMD化が促進されました。その結果、有効総命令数が削減され命令コミットが減少し性能が向上しました。

改善後ソース（ソースチューニング）

```
                <<< Loop-information Start >>>
                <<<  [OPTIMIZATION]
                <<<    PREFETCH(HARD) Expected by compiler :
                <<<      b, a
                <<< Loop-information  End >>>
 8    2  p              do j = 1, m
 9    2  p                i = 1
10    2  p                a(i,j) = c0 + a(1,j)*(c1 + b(i,j)*(c2 + b(i,j)*(c3 + b(i,j)*
11    2                &          (c4 + b(i,j)*(c5 + b(i,j)*(c6 + b(i,j)*(c7 + b(i,j)*
12    2                &          (c8 + b(i,j)*c9)))))))
                <<< Loop-information Start >>>
                <<<  [OPTIMIZATION]
                <<<    SIMD(VL: 8)
                <<<    SOFTWARE PIPELINING(IPC: 1.23, ITR: 128,
                                           MVE: 4, POL: S)
                <<<    PREFETCH(HARD) Expected by compiler :
                <<<      b, a
                <<< Loop-information  End >>>
13    3  p  2v           do i = 2, n
14    3  p  2v             a(i,j) = c0 + a(1,j)*(c1 + b(i,j)*(c2 + b(i,j)*(c3 + b(i,j)*
15    3                &          (c4 + b(i,j)*(c5 + b(i,j)*(c6 + b(i,j)*(c7 + b(i,j)*
16    3                &          (c8 + b(i,j)*c9)))))))
17    3  p  2v           end do
18    2  p              end do
```

依存関係のある箇所をピーリング

依存関係がなくなったことでSIMD化が促進された



SIMD

	Effective instruction	SIMD instruction rate (%) (/Effective instruction)
改善前	1.49E+11	14.61%
改善後	2.96E+10	81.26%

C/C++ ループピーリング（改善前）

FUJITSU

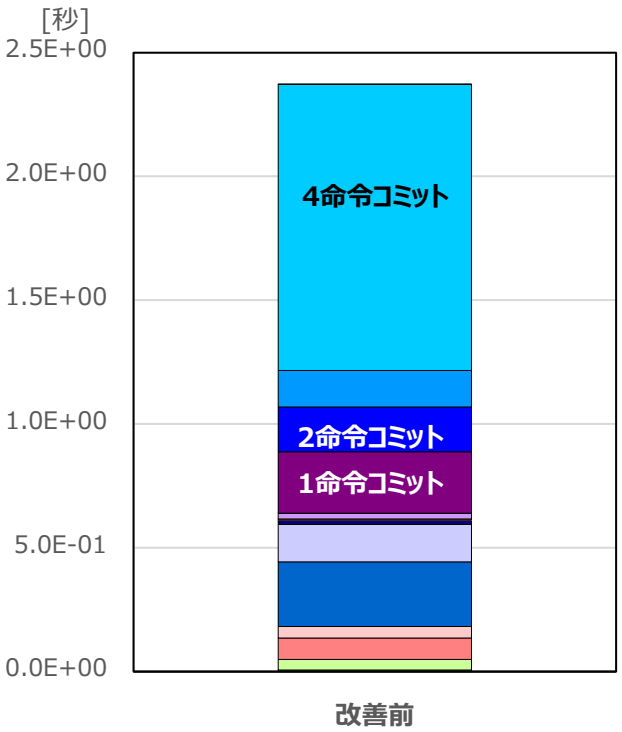
配列a について、i = 0 の時に定義したものを i = 1 以降で参照するデータ依存関係があるため、SIMD化が効果的に行われていません。

改善前ソース

```
40              #pragma omp parallel
41              for(k=0;k<1000000;k++){
42              #pragma omp for nowait
            <<< Loop-information Start >>>
            <<<  [OPTIMIZATION]
            <<<    PREFETCH(HARD) Expected by compiler :
            <<<      (unknown)
            <<< Loop-information  End >>>
43  p             for(j=0;j<m;j++){
            <<< Loop-information Start >>>
            <<<  [OPTIMIZATION]
            <<<    SIMD(VL: 8)
            <<<    SOFTWARE PIPELINING(IPC: 1.27, ITR: 144, MVE: 5, POL: S)
            <<<    PREFETCH(HARD) Expected by compiler :
            <<<      (unknown)
            <<< Loop-information  End >>>
44  p     2m        for(i=0;i<n;i++){
45  p     2m          a[j][i] = c0 + a[j][0]*(c1 + b[j][i]*(c2 + b[j][i]*(c3 + b[j][i]*
46                        (c4 + b[j][i]*(c5 + b[j][i]*(c6 + b[j][i]*(c7 + b[j][i]*
47                        (c8 + b[j][i]*c9))))))));
48  p     v       }
49  p           }
50              }
51            }
```

一部しかSIMD化が行われていない

i=0の時にロード側配列a とストア側配列a で依存関係がある。



Statistics	Effective instruction	SIMD instruction rate (%) (/Effective instruction)
改善前	1.33E+11	16.32%

ループピーリング（ソースチューニング）

FUJITSU

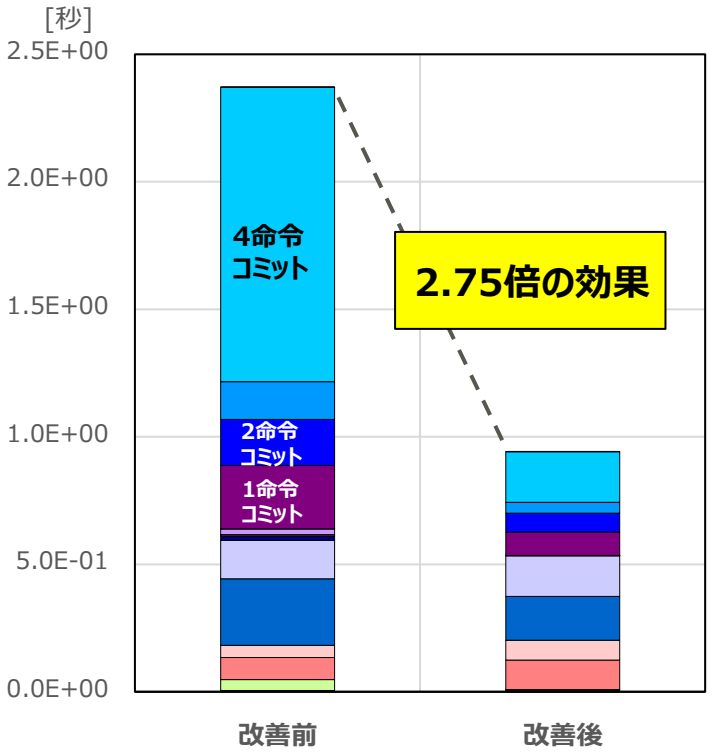
1回分のループピーリングを行うことで、SIMD化が促進されました。その結果、有効総命令数が削減され命令コミットが減少し性能が向上しました。

改善後ソース（ソースチューニング）

```
40          #pragma omp parallel
41          for(k=0;k<1000000;k++){
42          #pragma omp for nowait
          <<< Loop-information Start >>>
          <<<  [OPTIMIZATION]
          <<<    PREFETCH(HARD) Expected by compiler :
          <<<      (unknown)
          <<< Loop-information  End >>>
43  p         for(j=0;j<m;j++){
44  p           i=0;
45  p           a[j][i] = c0 + a[j][0]*(c1 + b[j][i]*(c2 + b[j][i]*(c3 + b[j][i]*
46                    (c4 + b[j][i]*(c5 + b[j][i]*(c6 + b[j][i]*(c7 + b[j][i]*
47                    (c8 + b[j][i]*c9)))))));
          <<< Loop-information Start >>>
          <<<  [OPTIMIZATION]
          <<<    SIMD(VL: 8)
          <<<    SOFTWARE PIPELINING(IPC: 1.23, ITR: 128, MVE: 5, POL: S)
          <<<    PREFETCH(HARD) Expected by compiler :
          <<<      (unknown)
          <<< Loop-information  End >>>
48  p   2v      for(i=1;i<n;i++){
49  p   2v        a[j][i] = c0 + a[j][0]*(c1 + b[j][i]*(c2 + b[j][i]*(c3 + b[j][i]*
50                    (c4 + b[j][i]*(c5 + b[j][i]*(c6 + b[j][i]*(c7 + b[j][i]*
51                    (c8 + b[j][i]*c9)))))));
52  p   2v      }
53  p         }
54          }
55        }
```

依存関係のある箇所をピーリング

依存関係がなくなったことでSIMD化が促進された



Statistics	Effective instruction	SIMD instruction rate (%) (/Effective instruction)
改善前	1.33E+11	16.32%
改善後	2.79E+10	86.10%

FUJITSU

定義関係が明らかでないループ

- 定義関係が明らかでないループ（改善前）
- 定義関係が明らかでないループ（最適化制御行チューニング）
- 定義関係が明らかでないループ（最適化制御行）

定義関係が明らかでないループ（最適化制御行）

FUJITSU

以下の最適化制御行を指定します。

最適化指示子 (Fortran)	意味	指定可能な最適化制御行			
		プログラム単位	DOループ単位	文単位	配列代入文単位
NORECURRENCE [(*array1*[,*array2*]...)]	DOループ内の演算対象となる配列の要素が、回転を跨いで定義引用されないことを本処理系に指示します。 （ループスライス可能な配列を指示します。） *array1*、*array2*、…は配列名です。	○	○	×	○

最適化指示子 (C/C++)	意味	指定可能な最適化制御行			
		global行	procedure行	loop行	statement行
norecurrence [(*array1*[,*array2*]...)]	ループの繰返しを跨いで定義引用されない配列を指示します。 （ループスライス可能な配列を指示します。） *array1*、*array2*、…は配列名です。	○	○	×	○

Fortran 定義関係が明らかでないループ（改善前）

FUJITSU

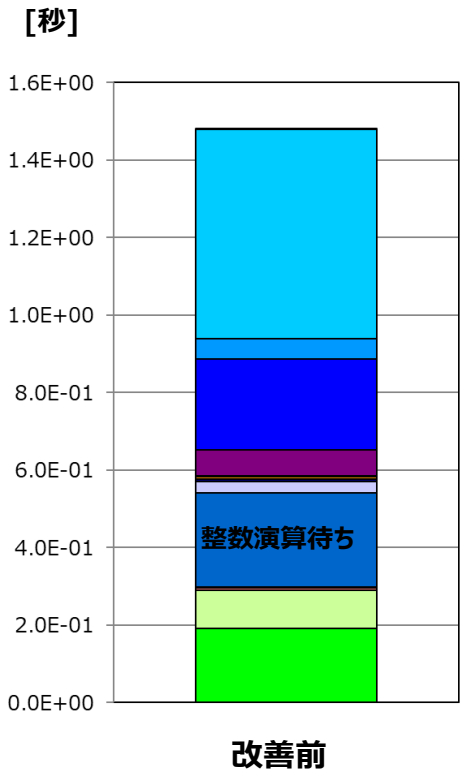
配列aについて、データ依存関係が不明なため、SIMD化や効果的なソフトウェアパイプライニングが行われていません。そのため、整数演算待ちが多くなっています。

改善前ソース

```
                <<< Loop-information Start >>>
                <<<  [PARALLELIZATION]
                <<<
                <<<
                <<<
                <<<
                <<< Loop-information  End >>>
53    1  pp             do j=1,n1
                <<< Loop-information Start >>>
                <<<  [OPTIMIZATION]
                <<<    SOFTWARE PIPELINING(IPC: 0.56, ITR: 3,
                                                MVE: 2, POL: S)
                <<<    PREFETCH(HARD) Expected by compiler :
                <<<      l, b, x
                <<< Loop-information  End >>>
54    2  p    s             do i=1,n2
55    2  p    m                 a(l(i),j)=a(x(i),j)/b(i,j)
56    2  p    v             end do
57    1  p               end do
```

配列a についてイタレーション間のデータ依存関係が不明のため、効果的なソフトウェアパイプライニングが行われていない。

ロード側の配列a とストア側の配列a の依存関係が不明。



SIMD

	Effective instruction	SIMD instruction rate (%) (/Effective instruction)
改善前	7.10E+10	0.00%

Fortran 定義関係が明らかでないループ（最適化制御行チューニング）

FUJITSU

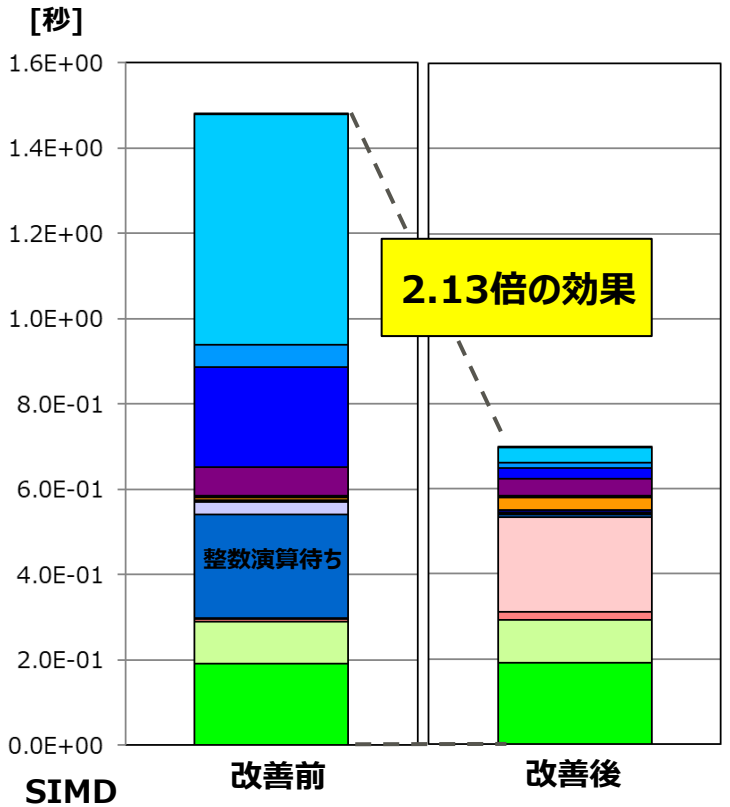
NORECURRENCE指示子によりデータ依存関係がないことを明示化するので、SIMD化およびソフトウェアパイプライニングが促進されました。その結果、整数演算待ちが大幅に改善されました。

改善後ソース（最適化制御行チューニング）

```
53                !ocl norecurrence(a)
                  <<< Loop-information
                  <<<  [PARALLELIZAT
                  <<<    Standard iterat
                  <<<  [OPTIMIZATION
                  <<<    PREFETCH(HARD) Expected by compiler :
                  <<<      x, b, l
                  <<< Loop-information  End >>>
54    1  pp             do j=1,n1
                  <<< Loop-information Start >>>
                  <<<  [OPTIMIZATION]
                  <<<    SIMD(VL: 16)
                  <<<    SOFTWARE PIPELINING(IPC: 2.72, ITR: 288,
                                                MVE: 3, POL: S)
                  <<<    PREFETCH(HARD) Expected by compiler :
                  <<<      x, b, l
                  <<< Loop-information  End >>>
55    2  p   2v           do i=1,n2
56    2  p   2v              a(l(i),j)=a(x(i),j)/b(i,j)
57    2  p   2v           end do
58    1  p              end do
```

a(l(i) , j) と a(x(i) , j) にデータ依存関係がないことをコンパイラに知らせる

SIMD化や効果的なソフトウェアパイプライニングが行われた



	Effective instruction	SIMD instruction rate (%) (/Effective instruction)
改善前	7.10E+10	0.00%
改善後	1.91E+10	10.08%

C/C++ 定義関係が明らかでないループ（改善前）

FUJITSU

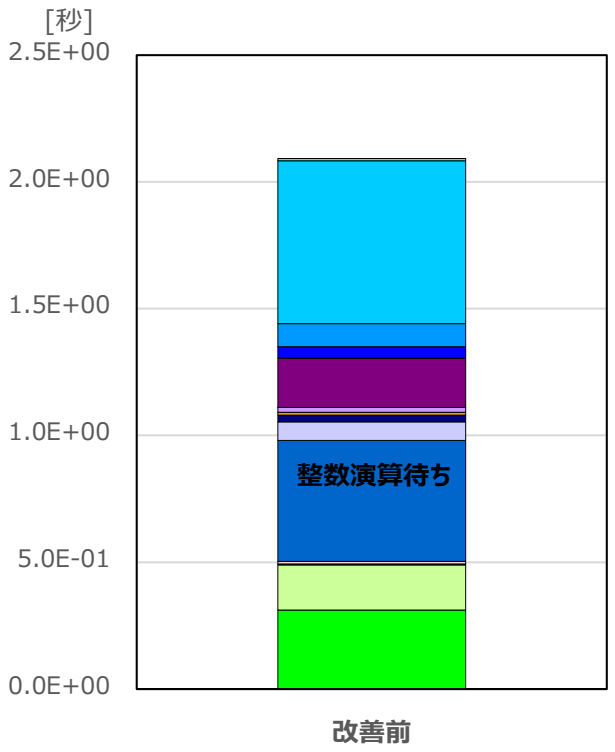
配列aについて、データ依存関係が不明なため、SIMD化や効果的なソフトウェアパイプライニングが行われていません。そのため、整数演算待ちが多くなっています。

改善前ソース

```
48          #pragma omp parallel
49           {
50             #pragma omp for nowait
           <<< Loo
           <<<  [OP
           <<<    PI
           <<<      (
           <<< Loop-information  End >>>
51   p          for(j=1; j<n1; j++)
52   p          {
           <<< Loop-information Start >>>
           <<<  [OPTIMIZATION]
           <<<    SOFTWARE PIPELINING(IPC: 0.39, ITR: 6,
                                      MVE: 2, POL: S)
           <<<    PREFETCH(HARD) Expected by compiler :
           <<<      (unknown)
           <<< Loop-information  End >>>
53   p    2s        for(i=0; i<n2; i++)
54   p    2v        {
55   p    2m          a[j][l[i]]=a[j][x[i]]/b[j][i];
56   p    2v        }
57   p            }
58             }
60             return;
61           }
```

配列a についてイタレーション間のデータ依存関係が不明のため、効果的なソフトウェアパイプライニングが行われていない。

ロード側の配列a とストア側の配列a の依存関係が不明。



Statistics	Effective instruction	SIMD instruction rate (%) (/Effective instruction)
改善前	8.49E+10	0.00%

C/C++ 定義関係が明らかでないループ（最適化制御行チューニング）

FUJITSU

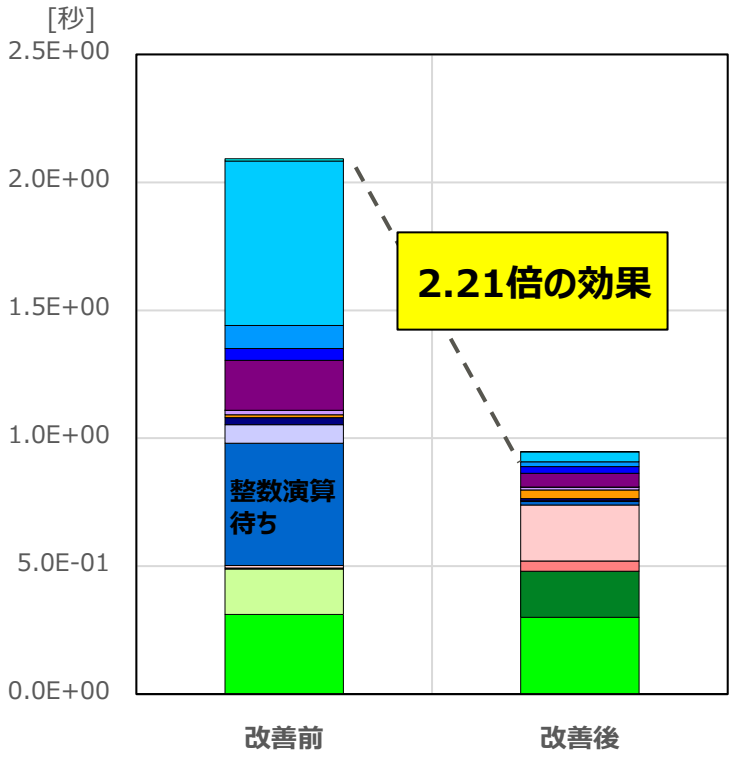
norecurrence指示子によりデータ依存関係がないことを明示化するので、SIMD化およびソフトウェアパイプライニングが促進されました。その結果、整数演算待ちが大幅に改善されました。

改善後ソース（最適化制御行チューニング）

```
48          #pragma omp parallel
49            {
50              #pragma omp for nowait
51              #pragma loop norecurrence
            <<< Loop-information Start >>>
            <<<  [OPTIMIZATIO
            <<<    PREFETCH(HA
            <<<      (unknown)
            <<< Loop-informatio
52  p             for(j=1; j<n1; j+
53  p             {
            <<< Loop-information Start >>>
            <<<  [OPTIMIZATION]
             <<<    SIMD(VL: 16)
            <<<    SOFTWARE PIPELINING(IPC: 2.27,ITR: 256,
                                       MVE: 3, POL: S)
            <<<    PREFETCH(HARD) Expected by compiler :
            <<<      (unknown)
            <<< Loop-information  End >>>
54  p   2v        for(i=0; i<n2; i++)
55  p   2v        {
56  p   2v          a[j][l[i]]=a[j][x[i]]/b[j][i];
57  p   2v        }
58  p           }
59            }
61            retu
62          }
```

a[j, l[i]]とa[j, x[i]] にデータ依存関係がないことをコンパイラに知らせる

SIMD化や効果的なソフトウェアパイプライニングが行われた



Statistics	Effective instruction	SIMD instruction rate (%) (/Effective instruction)
改善前	8.49E+10	0.00%
改善後	2.74E+10	5.49%

FUJITSU

ポインタ変数が含まれるループ

- ポインタ変数が含まれるループとは
- ポインタ変数が含まれるループ（改善前）
- ポインタ変数が含まれるループ（最適化制御行チューニング）
- ポインタ変数が含まれるループ（contiguous属性指示）

ポインタ変数が含まれるループとは

FUJITSU

ポインタ変数が記憶領域のどこを占めるかは実行時に決まるため、ポインタ変数を含むループでは最適化が促進されない場合があります。

ソースコード例

```
real,dimension(100),target::x
real,dimension(:),pointer::a,b
a=>x(1:10)
b=>x(11:20)
do i=1,10000
    a(i)=2.0/b(i)+1.0
end do
```

ソースコード例

```
      real,dimension(100),target::x
      real,dimension(:),pointer::a,b
!ocl noalias
      a=>x(1:10)
      b=>x(11:20)
      do i=1,10000
          a(i)=2.0/b(i)+1.0
      end do
```

NOALIAS指示子を指定することで、異なるポインタ変数が同一の記憶領域を指さないことを翻訳時に判断することができます。これによりポインタ変数に関する最適化が促進されます。

ただし、ポインタ変数の結合状態がループ内で変更される場合、最適化指示子を指定しても最適化が促進されないことがあります。

最適化指示子 (Fortran)	意味	指定可能な最適化制御行			
		プログラム単位	DOループ単位	文単位	配列代入文単位
NOALIAS	ポインタ変数が他の変数と記憶域を共有しないことを指示します。	○	○	×	○

最適化指示子 (C/C++)	意味	指定可能な最適化制御行			
		global行	procedure行	loop行	statement行
noalias	ポインタ変数が他の変数と記憶域を共有しないことを指示します。	○	○	○	×

Fortran ポインタ変数が含まれるループとは

FUJITSU

以下の翻訳オプションを指定することで、最適化制御行チューニングと同等の効果を得ることができます。

翻訳オプション	機能説明
-Knoalias [=spec]	ポインタ変数またはポインタ成分が他の変数と記憶域を結合しないものとして、最適化を行うことを指示します。specには、sを指定することができます。Sを指定した場合、コンパイラはFortran規格のポインタ属性を持つ実体が、他の変数を結合していないものとして最適化します。以下の文脈では、ポインタ属性をもつ実体が他の変数と結合する可能性があります。 - ポインタ代入文 - ポインタ成分をもつ派生型の代入文 - ポインタ成分をもつ派生型がSOURCE=指定子に現れるALLOCATE文 - ポインタ属性またはポインタ成分をもつ仮引数 - ポインタ属性またはポインタ成分をもつ変数への初期設定

■使用例（改善前ソース時）

■ $ frtpx -Kfast,parallel sample.f90 -Knoalias

Fortran ポインタ変数が含まれるループ（改善前）

FUJITSU

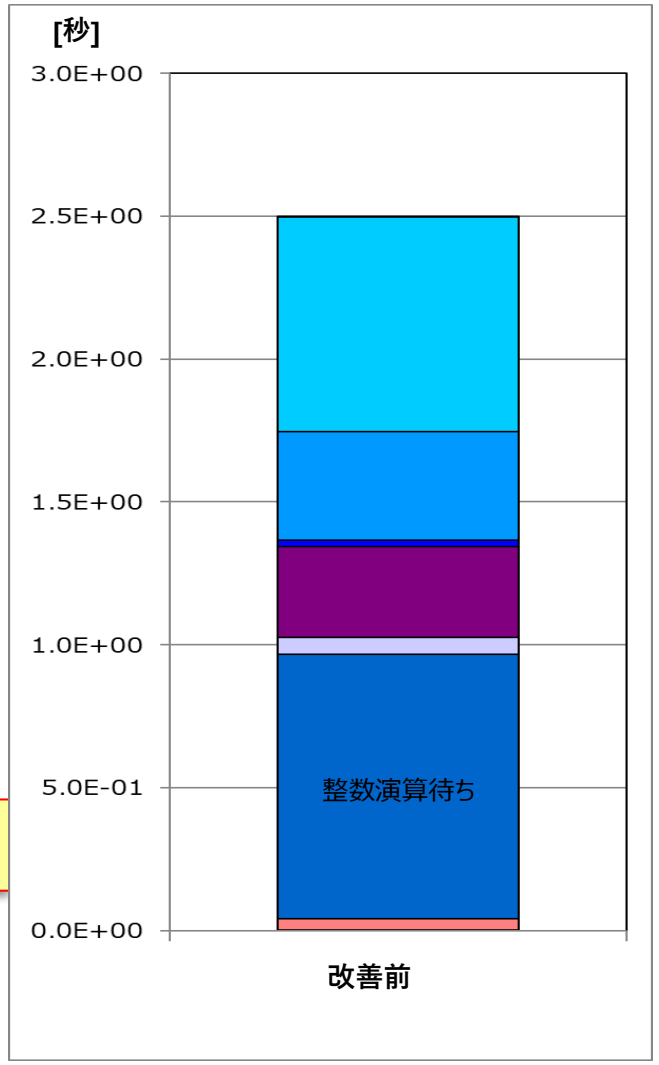
ポインタ変数a、bが異なる記憶領域を指すことが不明なためSIMD化が促進されません。
そのため、整数演算待ちが多くなっています。

改善前ソース

```
 3                    real,dimension(100000),target::x
 4                    integer :: kmax
 5                    real,dimension(:),pointer::a,b
 :
 9                    a=>x(1:10000)
10                     b=>x(10001:20000)
11                     kmax = 1000000
12                  !$omp parallel
13    1                 do k=1,kmax
14    1              !$omp do
                  <<< Loop-information Start >>>
                  <<<  [OPTIMIZATION]
                  <<<    SOFTWARE PIPELINING(IPC: 0.19, ITR: 8, MVE: 2, POL: L)
                  <<< Loop-information  End >>>
15    2  p  2s           do i=1,10000
16    2  p  2s             a(i)=2.0/b(i)+1.0
17    2  p  2s           end do
18    1              !$omp enddo nowait
19    1                 end do
20                  !$omp end parallel
```

SIMD化されていない

Statistics	Effective instruction	SIMD instruction rate (%) (/Effective instruction)
改善前	1.10E+11	0.00%



Fortran ポインタ変数が含まれるループ（最適化制御行チューニング）

FUJITSU

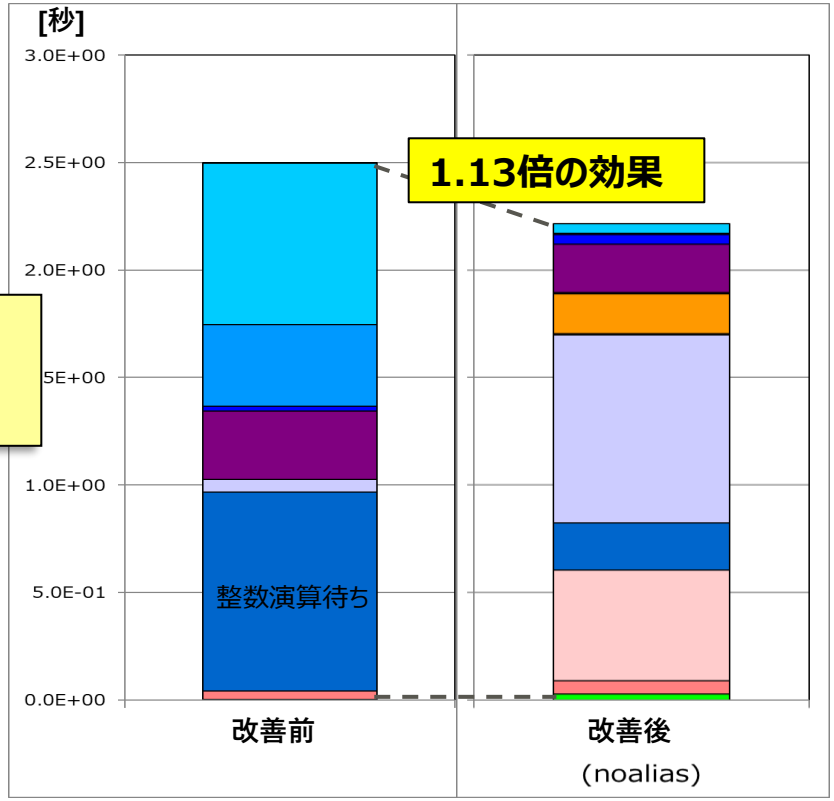
NOALIAS指示子によりデータ依存関係がないことを明示化することで、SIMD化およびソフトウェアパイプライニングが促進されました。その結果、整数演算待ちなどが大幅に改善されました。

改善後ソース（最適化制御行チューニング）

```
 3                    real,dimension(100000),target::x
 4                    integer :: kmax
 5                    real,dimension(:),pointer::a,b
 :
 9                    a=>x(1:10000)
10                     b=>x(10001:20000)
11                     kmax = 1000000
12                  !$omp parallel
13    1                 do k=1,kmax
14    1               !$omp do
15    1               !ocl noalias
                   <<< Loop-information Start >>>
                   <<<  [OPTIMIZATION]
                   <<<    SIMD(VL: 16)
                   <<<    SOFTWARE PIPELINING(IPC: 0.19, ITR: 96,
                                              MVE: 2, POL: L)
                   <<< Loop-information  End >>>
16    2  p  2v           do i=1,10000
17    2  p  2v             a(i)=2.0/b(i)+1.0
18    2  p  2v           end do
19    1               !$omp enddo nowait
20    1                 end do
21                  !$omp end parallel
```

a(i) とb (i) にデータ依存関係がないことをコンパイラに知らせる

SIMD化により有効総命令数が削減された



Statistics	Effective instruction	SIMD instruction rate (%) (/Effective instruction)
改善前	1.10E+11	0.00%
改善後	1.26E+10	40.83%

Fortran ポインタ変数が含まれるループ（contiguous属性指示）

FUJITSU

contiguous属性の指示によりデータの連続性を明示化することで、非連続命令が連続命令になりました。その結果、最適化が促進され、浮動小数点ロードL1Dアクセス待ちや浮動小数点演算待ちなどが大幅に改善されました。

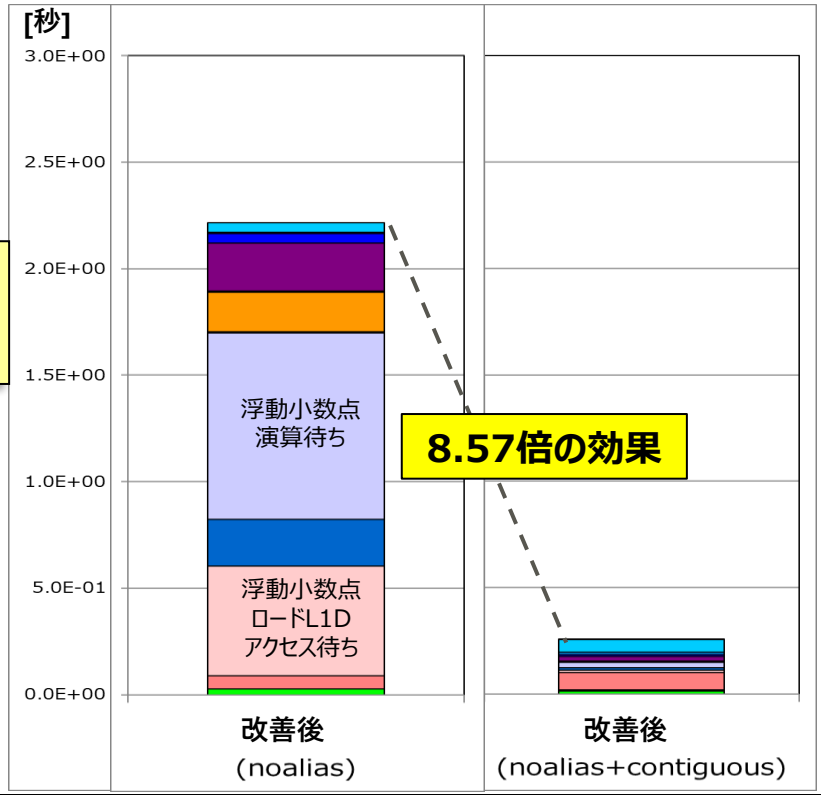
改善後ソース（contiguous属性指示）

```
 3                    real,dimension(100000),target::x
 4                    integer :: kmax
 5                    real,dimension(:),pointer,contiguous::a,b
 :
 9                    a=>x(1:10000)
10                     b=>x(10001:20000)
11                     kmax = 1000000
12                 !$omp parallel
                <<< Loop-information Start >>>
                <<<  [OPTIMIZATION]
                <<<    PREFETCH(HARD) Expected by compiler :
                <<<      (unknown)
                <<< Loop-information  End >>>
13    1              do k=1,kmax
14    1            !$omp do
15    1            !ocl noalias
                <<< Loop-information Start >>>
                <<<  [OPTIMIZATION]
                <<<    SIMD(VL: 16)
                <<<    SOFTWARE PIPELINING(IPC: 2.62, ITR: 352,
                                           MVE: 3, POL: S)
                <<<    PREFETCH(HARD) Expected by compiler :
                <<<      (unknown)
                <<< Loop-information  End >>>
16    2  p  2v            do i=1,10000
17    2  p  2v              a(i)=2.0/b(i)+1.0
18    2  p  2v            end do
19    1            !$omp enddo nowait
20    1              end do
21                 !$omp end parallel
```

ポインタ配列a、bがそれぞれ連続領域であることをコンパイラに知らせる

a(i) とb (i) にデータ依存関係がないことをコンパイラに知らせる

非連続命令が連続命令になった



Instruction	Load-store instruction					
	Load instruction			Store instruction		
	SIMD		Non-SIMD	SIMD		Non-SIMD
	Single vector contiguous load instruction	Non-contiguous gather load instruction	Non-SIMD load instruction	Single vector contiguous store instruction	Non-contiguous scatter store instruction	Non-SIMD store instruction
改善後 (noalias)	1.10E+01	6.36E+08	1.08E+09	1.56E+02	6.36E+08	1.33E+08
改善後 (noalias+contiguous)	6.36E+08	0.00E+00	5.00E+08	6.36E+08	0.00E+00	3.70E+07

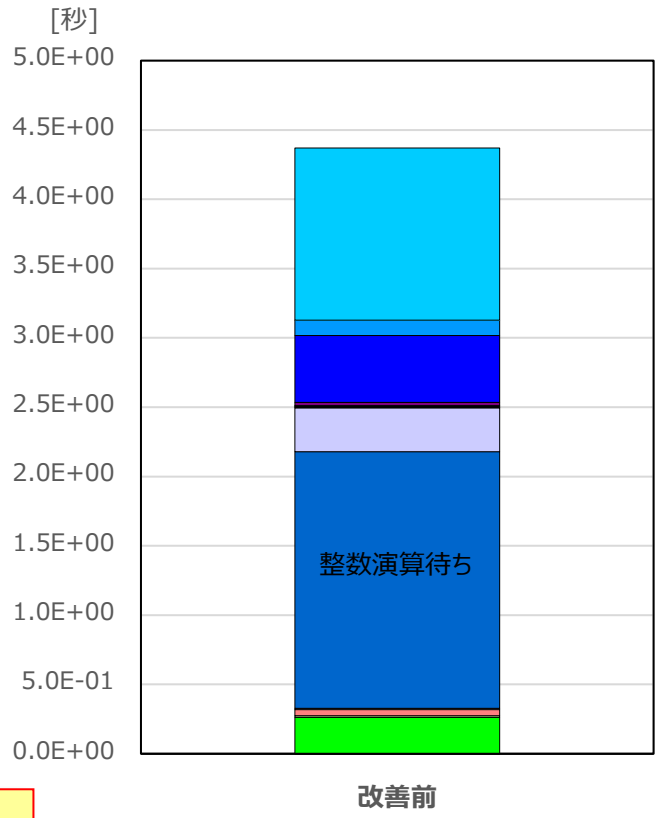
C/C++ ポインタ変数が含まれるループ（改善前）

FUJITSU

ポインタ変数a、bが異なる記憶領域を指すことが不明なためSIMD化が促進されません。
そのため、整数演算待ちが多くなっています。

改善前ソース

```
17          #pragma omp parallel
18          {
          <<< Loop-information Start >>>
          <<<  [OPTIMIZATION]
          <<<    PREFETCH(HARD) Expected by compiler :
          <<<      (unknown)
          <<< Loop-information  End >>>
19            for(k=0; k<kmax; k++)
20            {
21              #pragma omp for nowait
          <<< Loop-information Start >>>
          <<<  [OPTIMIZATION]
          <<<    SOFTWARE PIPELINING(IPC: 0.21, ITR: 8, MVE: 2, POL: L)
          <<<    PREFETCH(HARD) Expected by compiler :
          <<<      (unknown)
          <<< Loop-information  End >>>
22  p    2s       for(i=0; i<10000; i++)
23  p    2s       {
24  p    2s         a[i]=2.0/b[i]+1.0;
25  p    2s       }
26            }
27          }
```



SIMD化されていない

Statistics	Effective instruction	SIMD instruction rate (%) (/Effective instruction)
改善前	1.50E+11	0.00%

C/C++ ポインタ変数が含まれるループ（最適化制御行チューニング）

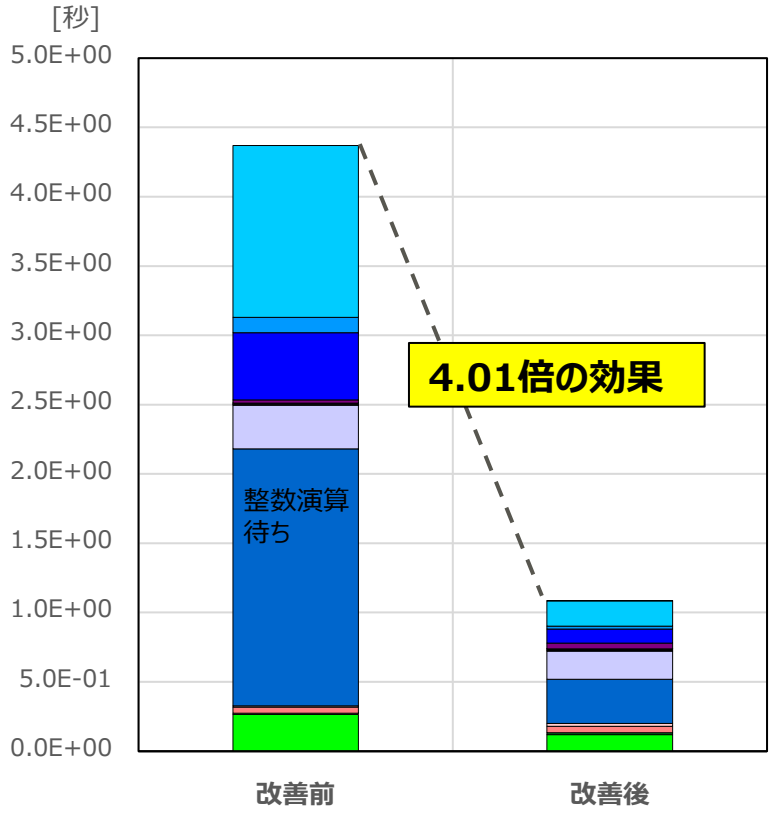
FUJITSU

NOALIAS指示子によりデータ依存関係がないことを明示化することで、SIMD化およびソフトウェアパイプライニングが促進されました。その結果、整数演算待ちなどが大幅に改善されました。

改善後ソース（最適化制御行チューニング）

```
17          #pragma omp parallel
18          {
        <<< Loop-information Start >>>
        <<<  [OPTIMIZATION]
        <<<    PREFETCH(HARD) Expected by compiler :
        <<<      (unknown)
        <<< Loop-information
19            for(k=0; k<kmax;
20            {
21              #pragma omp for nowait
22              #pragma loop noalias
          <<< Loop-information Start >>>
          <<<  [OPTIMIZATION]
          <<<    SIMD(VL: 8)
          <<<    SOFTWARE PIPELINING(IPC: 0.18, ITR: 64,
                                     MVE: 2, POL: L)
          <<<    PREFETCH(HARD) Expected by compiler :
          <<<      (unknown)
          <<< Loop-information  End >>>
23  p   2v       for(i=0; i<10000; i++)
24  p   2v       {
25  p   2v         a[i]=2.0/b[i]+1.0;
26  p   2v       }
27            }
28          }
```

a[i] とb[i] にデータ依存関係がないことをコンパイラに知らせる



Statistics	Effective instruction	SIMD instruction rate (%) (/Effective instruction)
改善前	1.50E+11	0.00%
改善後	2.27E+10	72.14%

SIMD化により有効総命令数が削減された

FUJITSU

演算待ち（レイテンシの隠蔽）

- ループ分割（ソフトウェアパイプライニング促進）
- 適切なアンローリング展開数の指定およびソフトウェアパイプライニングの抑止
- ストライピング（インターリーブ）展開数の指定および
  ソフトウェアパイプライニングの抑止
- 外側ループでのソフトウェアパイプライニング
- リローリング
- ループアンスイッチング

FUJITSU

ループ分割（ソフトウェアパイプライニング促進）

- ループ分割（ソフトウェアパイプライニング促進）（改善前）
- ループ分割（ソフトウェアパイプライニング促進）（ソースチューニング）

Fortran ループ分割（ソフトウェアパイプライニング促進）（改善前）

FUJITSU

演算の連鎖が長く、多くのレジスタを必要とします。
そのため、SWPLなどのスケジューリングの最適化ができず、浮動小数点演算待ちが多くなっています。

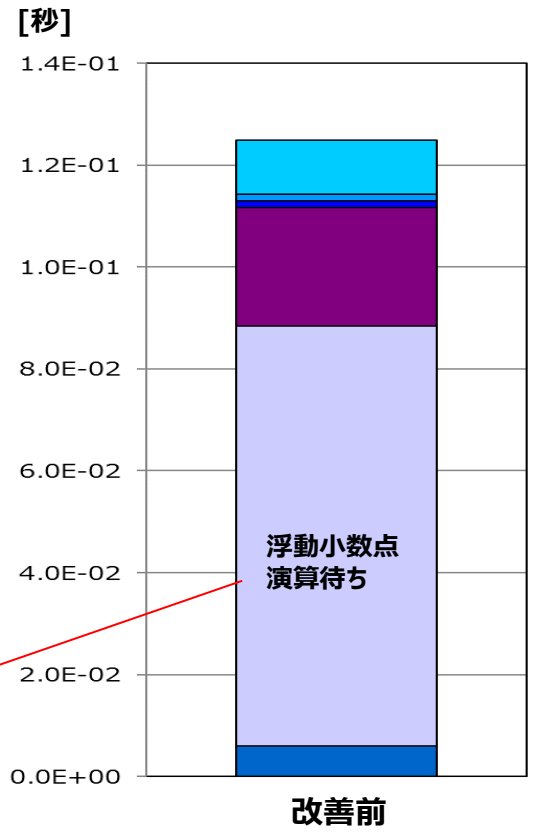
改善前ソース

```
                <<< Loop-information Start >>>
                <<<  [OPTIMIZATION]
                <<<    SIMD(VL: 8)
                <<<    PREFETCH(HARD) Expected by compiler :
                <<<      x1, y
                <<< Loop-information  End >>>
18    2     2v          do i = 1, n
19    2     2v            y(i) = c0 / x1(i) / c1 / x1(i) &
20    2                        / c2 / x1(i) / c3 / x1(i) / c4 / x1(i)
21    2     2v          end do
```

※コンパイルオプション
-Knoevalを指定

演算連鎖が長く、多くのレジスタを使用するため、ソフトウェアパイプライニングを適用できない

スケジューリングが最適化できていないため、演算待ちが大きくみえる



Cache

	L1I miss rate (/Effective instruction)	Load-store instruction	L1D miss	L1D miss rate (/Load-store instruction)	L1D miss demand rate (%) (/L1D miss)	L1D miss hardware prefetch rate (%) (/L1D miss)	L1D miss software prefetch rate (%) (/L1D miss)	L2 miss	L2 miss rate (/Load-store instruction)	L2 miss demand rate (%) (/L2 miss)	L2 miss hardware prefetch rate (%) (/L2 miss)	L2 miss software prefetch rate (%) (/L2 miss)
改善前	0.00	7.69E+06	1.61E+03	0.00	71.62%	27.38%	1.00%	1.32E+02	0.00	48.92%	59.71%	0.00

Fortran ループ分割（ソフトウェアパイプライニング促進）（ソースチューニング）

FUJITSU

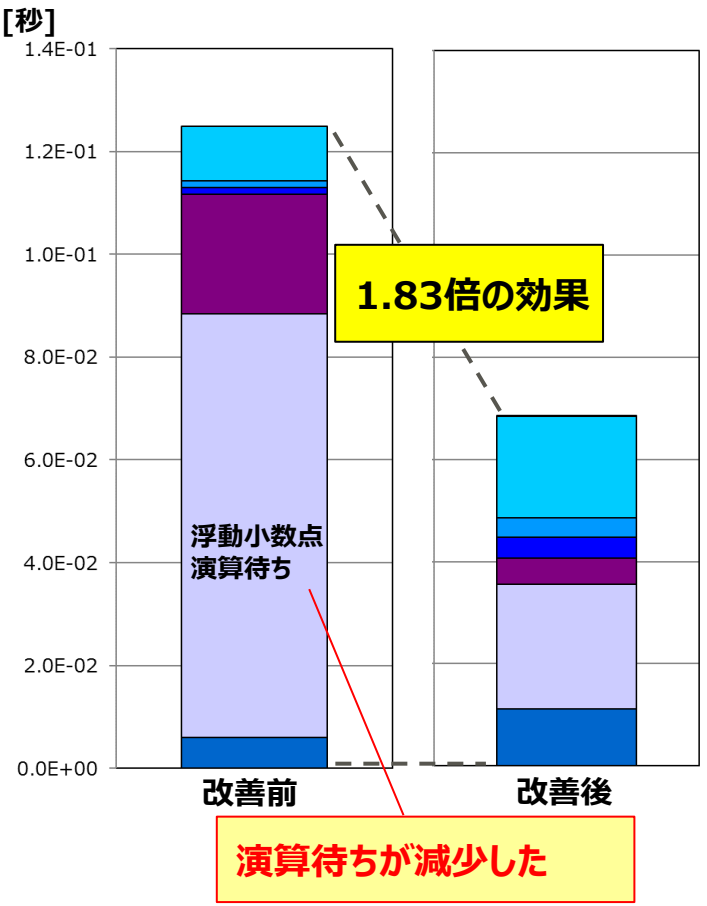
ループ分割を行うことで演算の連鎖が短くなり、1ループに使用するレジスタが減少しました。
その結果、SWPLなどのスケジューリングの最適化が行われ、演算待ちが改善されました。

改善後ソース

```
20    1          !ocl loop_nofusion
                 <<< Loop-information Start >>>
                 <<<  [OPTIMIZATION]
                 <<<    SIMD(VL: 8)
                 <<<    SOFTWARE PIPELINING(IPC: 1.02, …
                 <<<    PREFETCH(HARD) Expected by compiler :
                 <<<      y, x1
                 <<<    SPILLS :
                 <<<      GENERAL   : SPILL 0  FILL 0
                 <<<      SIMD&FP   : SPILL 0  FILL 0
                 <<<      SCALABLE  : SPILL 0  FILL 27
                 <<<      PREDICATE : SPILL 0  FILL 0
                 <<< Loop-information  End >>>
21    2     2v          do i = 1, n
22    2     2v            y(i) = c2 / x1(i) / c3 / x1(i) / c4 / x1(i)
23    2     2v          end do
                 <<< Loop-information Start >>>
                 <<<  [OPTIMIZATION]
                 <<<    SIMD(VL: 8)
                 <<<    SOFTWARE PIPELINING(IPC: 1.08, …
                 <<<    PREFETCH(HARD) Expected by compiler :
                 <<<      x1, y
                 <<< Loop-information  End >>>
24    2     2v          do i = 1, n
25    2     2v            y(i) = c0 / x1(i) / c1 / x1(i) /  y(i)
26    2     2v          end do
```

ループ融合を抑止

ループ分割



※コンパイルオプション-Knoevalを指定

	L1I miss rate (/Effective instruction)	Load-store instruction	L1D miss	L1D miss rate (/Load-store instruction)	L1D miss demand rate (%) (/L1D miss)	L1D miss hardware prefetch rate (%) (/L1D miss)	L1D miss software prefetch rate (%) (/L1D miss)
改善前	0.00	7.69E+06	1.61E+03	0.00	71.62%	27.38%	1.00%
改善後	0.00	3.65E+07	1.73E+03	0.00	70.79%	28.98%	0.23%

ループ分割（ソフトウェアパイプライニング促進）（改善前）

FUJITSU

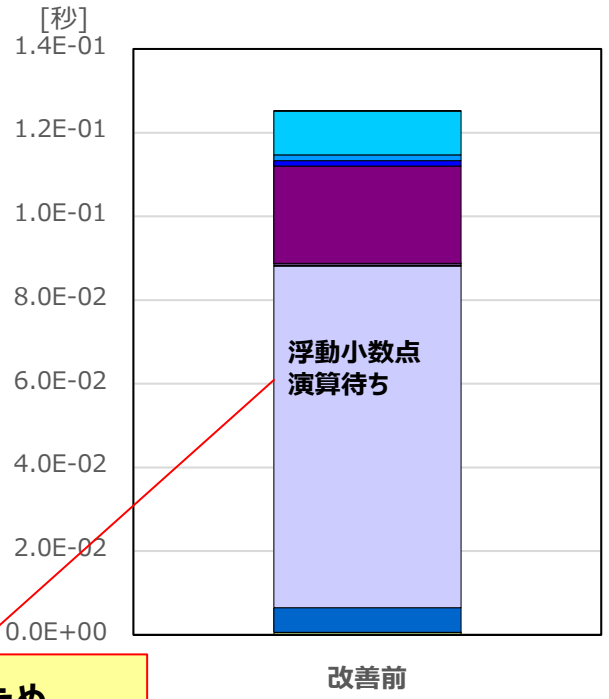
演算の連鎖が長く、多くのレジスタを必要とします。
そのため、SWPLなどのスケジューリングの最適化ができず、浮動小数点演算待ちが多くなっています。

改善前ソース

```
18          for (iter = 0; iter < 10000; iter++){
           <<< Loop-information Start >>>
           <<<  [OPTIMIZATION]
           <<<    SIMD(VL: 8)
           <<<    PREFETCH(HARD) Expected by compiler :
           <<<      (unknown)
           <<< Loop-information  End >>>
19        2v     for (i = 0; i < n; i++){
20        2v       y[i] = c0 / x1[i] / c1 / x1[i] / c2 / x1[i] / c3 / x1[i] / c4 / x1[i];
21        2v     }
22           }
```

※コンパイルオプション
-Knoevalを指定

演算連鎖が長く、多くのレジスタを使用するため、ソフトウェアパイプライニングを適用できない



スケジューリングが最適化できていないため、演算待ちが大きくみえる

Cache	L1I miss rate (/Effective instruction)	Load-store instruction	L1D miss	L1D miss rate (/Load-store instruction)	L1D miss demand rate (%) (/L1D miss)	L1D miss hardware prefetch rate (%) (/L1D miss)	L1D miss software prefetch rate (%) (/L1D miss)	L2 miss	L2 miss rate (/Load-store instruction)	L2 miss demand rate (%) (/L2 miss)	L2 miss hardware prefetch rate (%) (/L2 miss)	L2 miss software prefetch rate (%) (/L2 miss)
改善前	0.00	1.11E+09	1.72E+06	0.00	99.92%	0.08%	-0.01%	4.01E+03	0.00	86.58%	26.24%	0.00%

C/C++ tradモード ループ分割（ソフトウェアパイプライニング促進）（ソースチューニング）

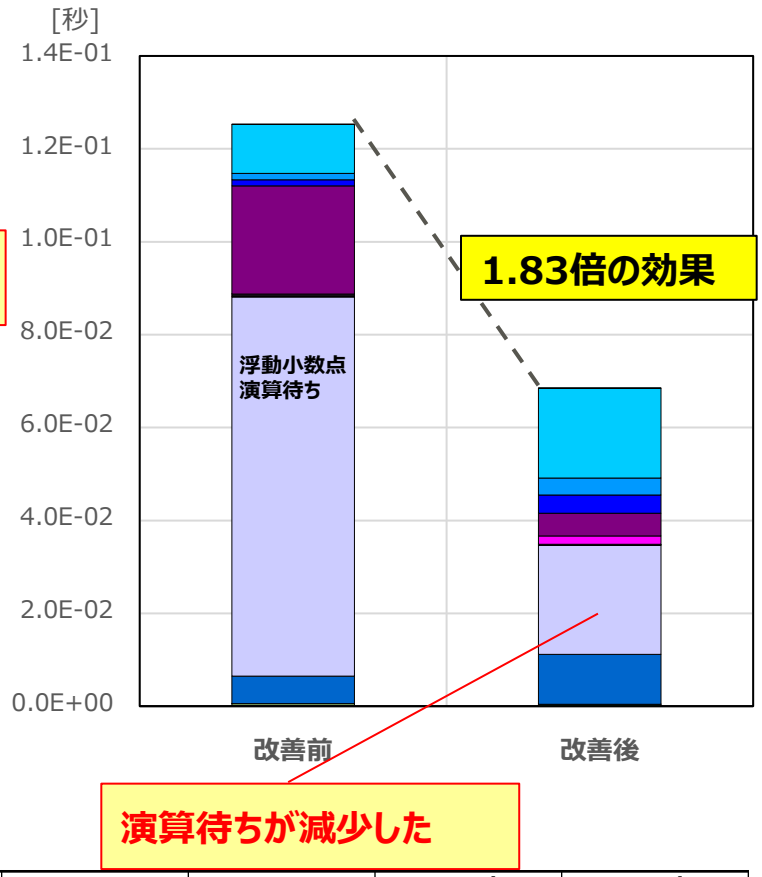
ループ分割を行うことで演算の連鎖が短くなり、1ループに使用するレジスタが減少しました。
その結果、SWPLなどのスケジューリングの最適化が行われ、演算待ちが改善されました。

改善後ソース

```
18          for (iter = 0; iter < 10000; iter++){
19          #pragma loop loop_nofusion
            <<< Loop-information Start >>>
            <<<  [OPTIMIZATION]
            <<<    SIMD(VL: 8)
            <<<    SOFTWARE PIPELINING(IPC: 1.02, ITR: 112, MVE: 3, POL: L)
            <<<    PREFETCH(HARD) Expected by compiler :
            <<<      (unknown)
            <<<    SPILLS :
            <<<      GENERAL   : SPILL 0  FILL 0
            <<<      SIMD&FP   : SPILL 0  FILL 0
            <<<      SCALABLE  : SPILL 0  FILL 27
            <<<      PREDICATE : SPILL 0  FILL 0
            <<< Loop-information  End >>>
20          2v      for (i = 0; i < n; i++){
21          2v        y[i] = c2 / x1[i] / c3 / x1[i] / c4 / x1[i];
22          2v      }
            <<< Loop-information Start >>>
            <<<  [OPTIMIZATION]
            <<<    SIMD(VL: 8)
            <<<    SOFTWARE PIPELINING(IPC: 1.08, ITR: 64, MVE: 2, POL: S)
            <<<    PREFETCH(HARD) Expected by compiler :
            <<<      (unknown)
            <<< Loop-information  End >>>
23          2v      for (i = 0; i < n; i++){
24          2v        y[i] = c0 / x1[i] / c1 / x1[i] / y[i];
25          2v      }
26            }
```

ループ融合を抑止

ループ分割



※コンパイルオプション-Knoevalを指定

Cache	L1I miss rate (/Effective instruction)	Load-store instruction	L1D miss	L1D miss rate (/Load-store instruction)	L1D miss demand rate (%) (/L1D miss)	L1D miss hardware prefetch rate (%) (/L1D miss)	L1D miss software prefetch rate (%) (/L1D miss)
改善前	0.00	1.11E+09	1.72E+06	0.00	99.92%	0.08%	-0.01%
改善後	0.00	6.46E+08	9.82E+05	0.00	99.88%	0.11%	0.00%

ループ分割（ソフトウェアパイプライニング促進）（改善前）

FUJITSU

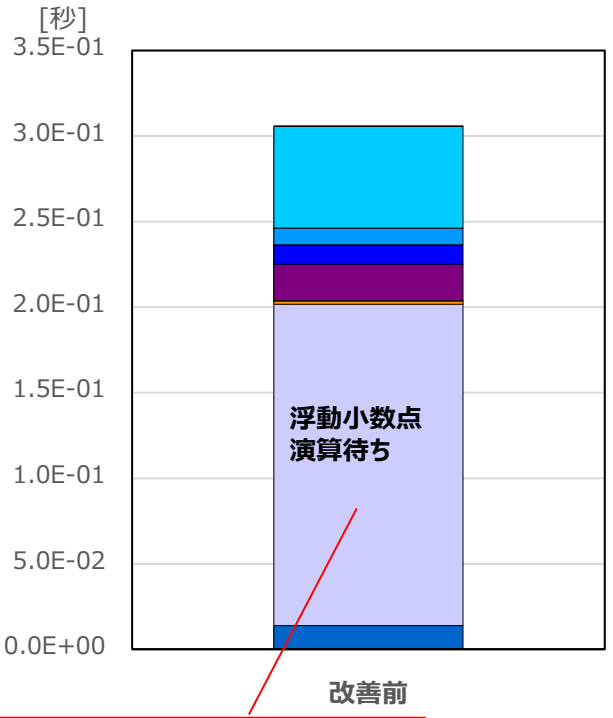
演算の連鎖が長く、多くのレジスタを必要とします。
そのため、SWPLなどのスケジューリングの最適化ができず、浮動小数点演算待ちが多くなっています。

改善前ソース

```
19          for (iter = 0; iter < 10000; iter++){
          <<< Loop-information Start >>>
          <<<  [OPTIMIZATION]
          <<<    SIMD(VL: 8 Interleave: 1)
          <<<    SPILLS :
          <<<      GENERAL   : SPILL 0  FILL 0
          <<<      SIMD&FP   : SPILL 0  FILL 8
          <<<      SCALABLE  : SPILL 4  FILL 6
          <<<      PREDICATE : SPILL 0  FILL 0
          <<< Loop-information End >>>
20        v     for (i = 0; i < n; i++){
21                y[i] =    sin(x1[i]*c0)
22                        +cos(x1[i]*c1)
23                        +atan(x1[i]*c2)
24                        +log(x1[i]*c3);
25            }
26  }
```

演算連鎖が長く、多くのレジスタを使用するため、ソフトウェアパイプライニングを適用できない

※コンパイラの特性を考慮して数学関数を例にした



スケジューリングが最適化できていないため、演算待ちが大きくみえる

Cache	L1I miss rate (/Effective instruction)	Load-store instruction	L1D miss	L1D miss rate (/Load-store instruction)	L1D miss demand rate (%) (/L1D miss)	L1D miss hardware prefetch rate (%) (/L1D miss)	L1D miss software prefetch rate (%) (/L1D miss)	L2 miss	L2 miss rate (/Load-store instruction)	L2 miss demand rate (%) (/L2 miss)	L2 miss hardware prefetch rate (%) (/L2 miss)	L2 miss software prefetch rate (%) (/L2 miss)
改善前	0.00	1.76E+09	2.04E+06	0.00	99.85%	0.14%	0.01%	9.95E+03	0.00	88.46%	18.51%	0.00%

C/C++ clangモード ループ分割（ソフトウェアパイプライニング促進）（ソースチューニング）

FUJITSU

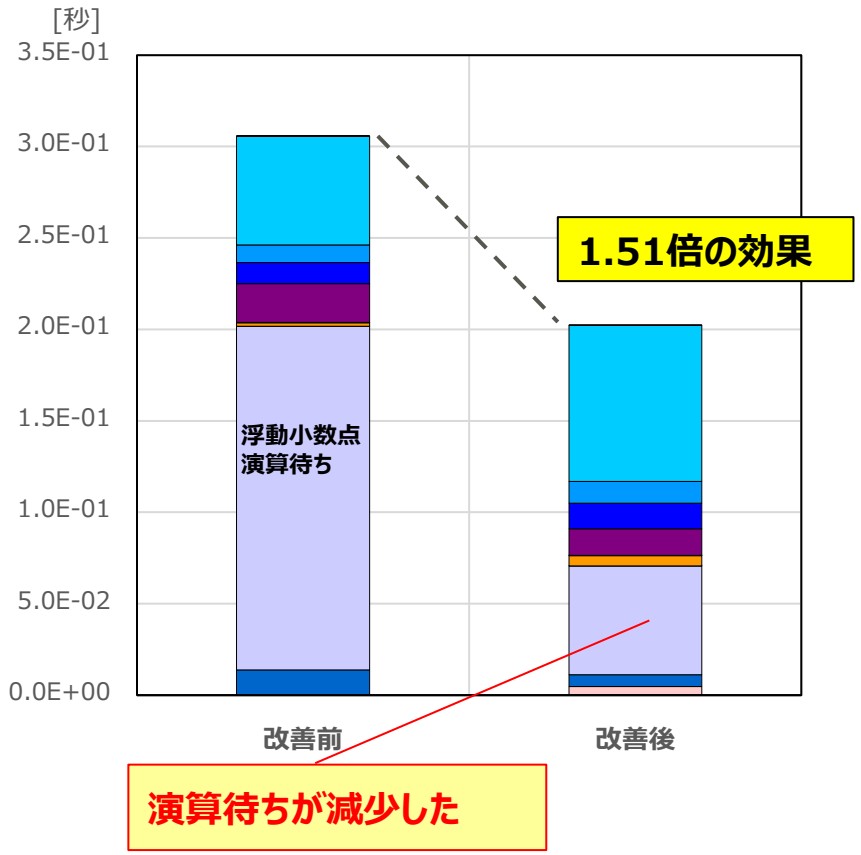
ループ分割を行うことで演算の連鎖が短くなり、1ループに使用するレジスタが減少しました。
その結果、SWPLなどのスケジューリングの最適化が行われ、演算待ちが改善されました。

改善後ソース

```
19          for (iter = 0; iter < 10000; iter++){
20            #pragma loop loop_nofusion
            <<< Loop-information Start >>>
            <<<  [OPTIMIZATION]
            <<<    SIMD(VL: 8 Interleave: 1)
            <<<    SOFTWARE PIPELINING
            <<<    SPILLS :
            <<<      GENERAL   : SPILL 0  FILL 0
            <<<      SIMD&FP   : SPILL 0  FILL 0
            <<<      SCALABLE  : SPILL 8  FILL 17
            <<<      PREDICATE : SPILL 0  FILL 0
            <<< Loop-information End >>>
21        v     for (i = 0; i < n; i++){
22              y[i] =    sin(x1[i]*c0);
23            }
            <<< Loop-information Start >>>
            <<<  [OPTIMIZATION]
             (略)
            <<< Loop-information End >>>
24        v     for (i = 0; i < n; i++){
25              y[i] +=    cos(x1[i]*c1);
26            }
            <<< Loop-information Start >>>
            <<<  [OPTIMIZATION]
             (略)
            <<< Loop-information End >>>
27        v     for (i = 0; i < n; i++){
28              y[i] +=    atan(x1[i]*c2);
29            }
            <<< Loop-information Start >>>
            <<<  [OPTIMIZATION]
             (略)
            <<< Loop-information End >>>
30        v     for (i = 0; i < n; i++){
31              y[i] +=    log(x1[i]*c3);
32            }
33          }
```

ループ融合を抑止

ループ分割



※コンパイラの特性を考慮して数学関数を例にした

Cache	L1I miss rate (/Effective instruction)	Load-store instruction	L1D miss	L1D miss rate (/Load-store instruction)	L1D miss demand rate (%) (/L1D miss)	L1D miss hardware prefetch rate (%) (/L1D miss)	L1D miss software prefetch rate (%) (/L1D miss)
改善前	0.00	1.76E+09	2.04E+06	0.00	99.85%	0.14%	0.01%
改善後	0.00	2.67E+09	2.39E+06	0.00	99.91%	0.10%	-0.01%

FUJITSU

適切なアンローリング展開数の指定およびソフトウェアパイプライニングの抑止

- ソフトウェアパイプライン後のループの実行動作
- 適切なアンローリング展開数の指定およびソフトウェアパイプライニングの抑止（改善前）
- 適切なアンローリング展開数の指定およびソフトウェアパイプライニングの抑止（最適化制御行チューニング）
- 適切なアンローリング展開数の指定およびソフトウェアパイプライニングの抑止（最適化制御行）

ソフトウェアパイプライン後のループの実行動作

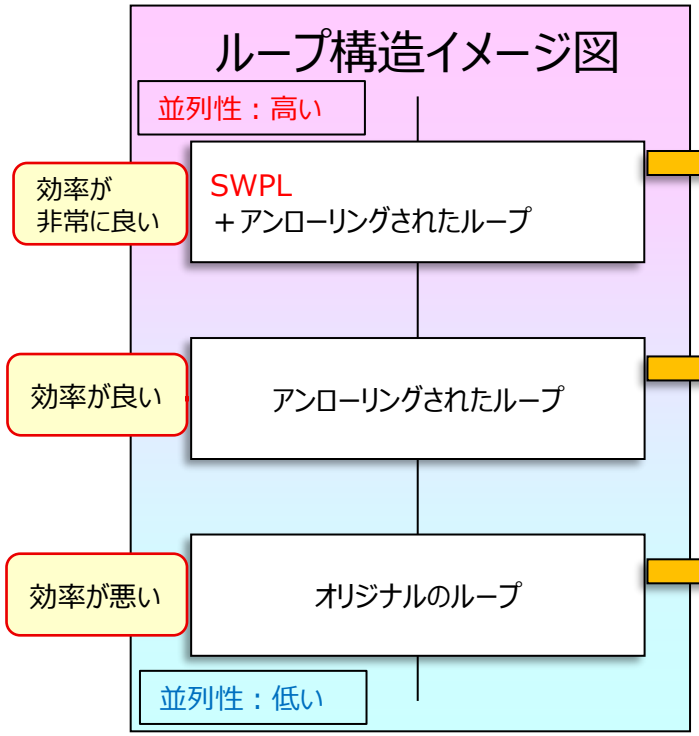
FUJITSU

- **ソフトウェアパイプラインではループの繰り返し数により、以下の処理ルートが決定されます。**

例

```
<<< Loop-information Start >>>
                    :
<<<  [OPTIMIZATION]
<<<    SIMD(VL: 8)
<<<    SOFTWARE PIPELINING(IPC: 2.50, ITR: 144,
                           MVE: 4, POL: S)
<<< Loop-information  End
17    1  pp  2v       do i=1,N
18    1   p  2v        b(i)=a(i)+c
19    1   p  2v       enddo
```

例題の想定
ループ繰り返し数n=312
アンローリング展開数=2
SWPL
8SIMD



if(繰り返し数がSWPLの提示の値(144回転以上)の処理) then

ループの繰り返し数が144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.
例題ループの場合、i=1～288までが当ループで実行されます。

if(上記の残り繰り返し数の内、繰り返し数が16回転の倍数分の処理) then

アンローリング展開数2×8(SIMD)=16
例題ループの場合、i=289～304までが当ループで実行されます。

if(上記の残り繰り返し数の内、繰り返し数が8回転の倍数分の処理) then

8(SIMD)
例題ループの場合、残りのi=305～312までが当ループで実行されます。

上記のコンパイラで多重生成されるループ構造イメージ図は、上の階層ほど命令レベルの並列性が高い。

上記のようにループ繰り返し数によって処理ルートが決まるため、ループの繰り返し数が少ない場合は、命令レベルの並列性が高いループが実行されないので命令スケジューリングの効果が小さくなる。

Fortran 適切なアンローリング展開数の指定およびソフトウェアパイプライニングの抑止（改善前）

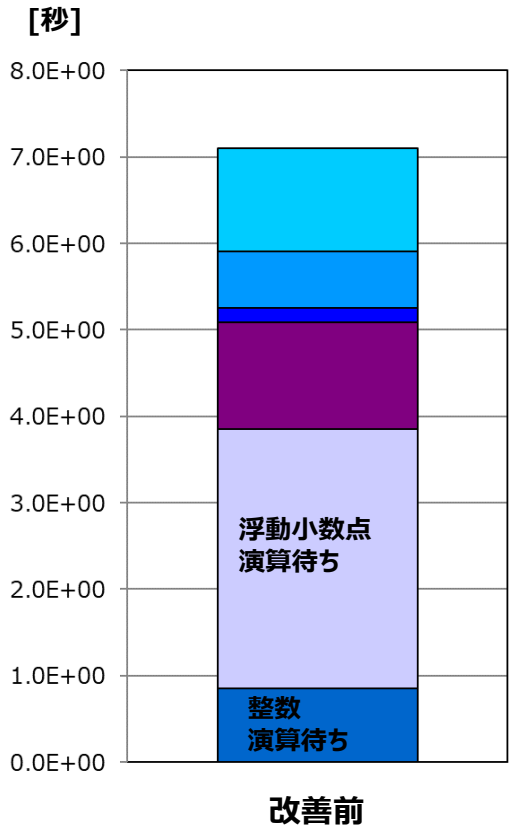
FUJITSU

繰り返し数が少なく、アンローリングやソフトウェアパイプライニングが効果的に機能していません。そのため、浮動小数点演算待ちや整数演算待ちが多くなっています。

改善前ソース

```
          <<< Loop-information Start >>>
          <<<  [PARALLELIZATION]
          <<<    Standard iteration count:
          <<<  [OPTIMIZATION]
          <<<    SIMD(VL: 8)
          <<<    SOFTWARE PIPELINING(IPC: 1.62, ITR: 176,
                                     MVE: 6, POL: S)
          <<<    PREFETCH(HARD) Expected by compiler :
          <<<      a, b
          <<< Loop-information  End >>>
39    1  pp   2v   do i = 1 , n
40    1   p   2v      b(i) = c0 + a(i)*(c1 + a(i)*(c2 + a(i)*(c3 + a(i)*
41    1           &     (c4 + a(i)*(c5 + a(i)*(c6 + a(i)*(c7 + a(i)*
42    1           &     (c8 + a(i)*c9)))))))
43    1   p   2v           enddo
```

ループ繰り返し数n=40 に対し
アンロール展開数2



問題点：ループの繰り返し数が少ない

・ソフトウェアパイプライニングのループが実行されない
ソフトウェアパイプライニングの条件ITR: 176以下である。

・アンローリング展開数が適切でない
アンロール展開数 2 × 8 (SIMD) = 16
8回転がオリジナルのループで実行されてしまう。
オリジナルのループが実行されることによる性能への影響は大きい。

Fortran 適切なアンローリング展開数の指定およびソフトウェアパイプライニングの抑止（最適化制御行チューニング）

FUJITSU

繰り返し数に応じたアンロール展開数の指定およびソフトウェアパイプライニングの抑止により、適切な命令スケジューリングが行われました。
その結果、浮動小数点演算待ち・整数演算待ちが削減されました。

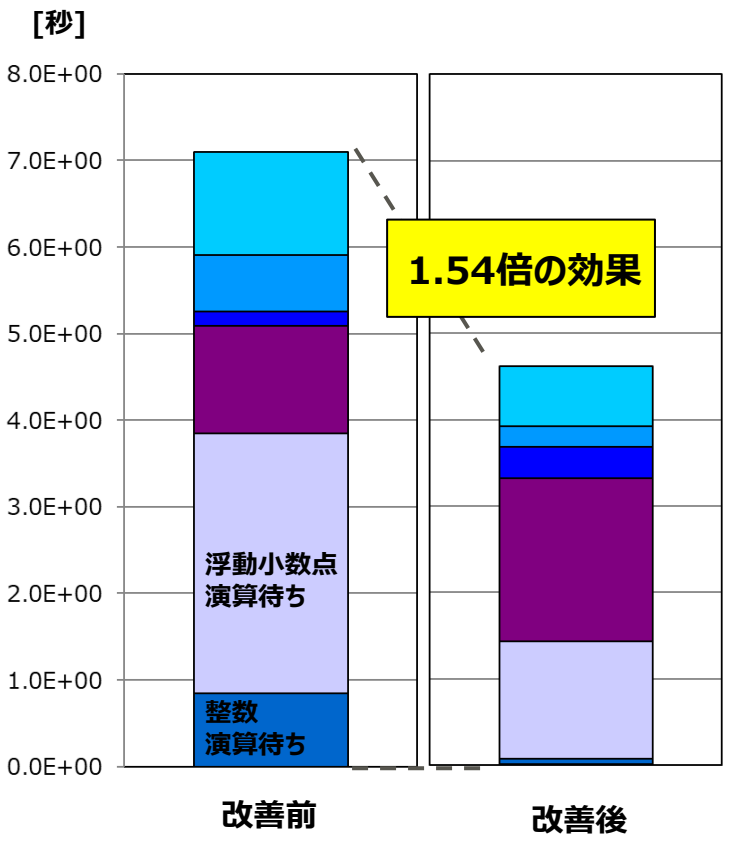
改善後ソース（最適化制御行チューニング）

```
39              !ocl unroll(5)
40              !ocl noswp
          <<< Loop-information Start >>>
          <<<  [PARALLELIZATION]
          <<<    Standard iteration count: 55
          <<<  [OPTIMIZATION]
          <<<    SIMD(VL: 8)
          <<<    PREFETCH(HARD) Expected by compiler :
          <<<      a, b
          <<< Loop-information  End >>>
41    1  pp   5v  do i = 1 , n
42    1   p   5v     b(i) = c0 + a(i)*(c1 + a(i)*(c2 + a(i)*(c3 + a(i)*
43    1         &        (c4 + a(i)*(c5 + a(i)*(c6 + a(i)*(c7 + a(i)*
44    1         &        (c8 + a(i)*c9))))))))
45    1   p   5v    enddo
```

アンロール数5に指定

ソフトウェアパイプライニングの抑止

アンロール展開数 5指定により
アンロール展開数 5×8 (SIMD) = 40
40回転全てアンローリングされたループが実行される。



C/C++ 適切なアンローリング展開数の指定およびソフトウェアパイプライニングの抑止（改善前）

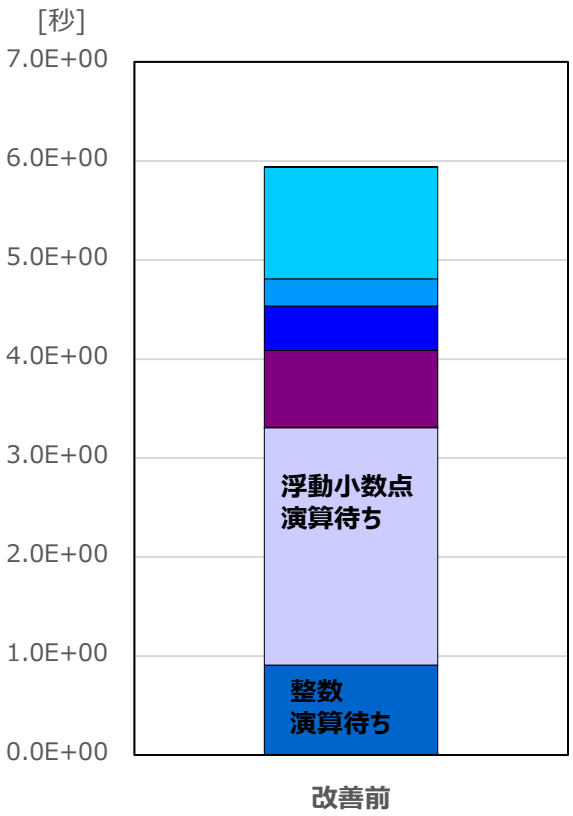
FUJITSU

繰り返し数が少なく、アンローリングやソフトウェアパイプライニングが効果的に機能していません。そのため、浮動小数点演算待ちや整数演算待ちが多くなっています。

改善前ソース

```
            <<< Loop-information Start >>>
            <<<  [OPTIMIZATION]
            <<<    SIMD(VL: 8)
            <<<    SOFTWARE PIPELINING(IPC: 1.62, ITR: 176,
                                       MVE: 6, POL: S)
            <<<    PREFETCH(HARD) Expected by compiler :
            <<<      (unknown)
            <<< Loop-information  End >>>
31        2v    for (i = 0; i < n; i++){
32        2v      b[i] = c0 + a[i]*(c1 + a[i]*(c2 + a[i]*(c3 + a[i]*
33                    (c4 + a[i]*(c5 + a[i]*(c6 + a[i]*(c7 + a[i]*
34                    (c8 + a[i]*c9)))))));
35        2v    }
36            }
```

ループ繰り返し数n=40 に対し
アンロール展開数2



問題点：ループの繰り返し数が少ない

・ソフトウェアパイプライニングのループが実行されない
ソフトウェアパイプライニングの条件ITR: 176以下である。

・アンローリング展開数が適切でない
アンロール展開数 2 × 8 (SIMD) = 16
8回転がオリジナルのループで実行されてしまう。
オリジナルのループが実行されることによる性能への影響は大きい。

C/C++ 適切なアンローリング展開数の指定およびソフトウェアパイプライニングの抑止（最適化制御行チューニング）

FUJITSU

繰り返し数に応じたアンロール展開数の指定およびソフトウェアパイプライニングの抑止により、適切な命令スケジューリングが行われました。
その結果、浮動小数点演算待ち・整数演算待ちが削減されました。

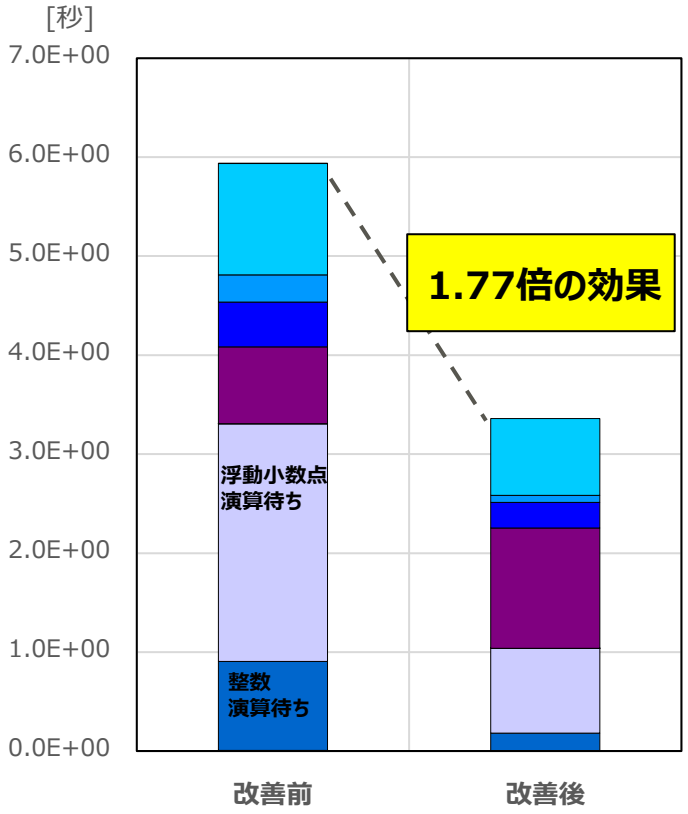
改善後ソース（最適化制御行チューニング）

```
31          #pragma loop unroll 5
32          #pragma loop noswp
            <<< Loop-information Start >>>
            <<<  [OPTIMIZATION]
            <<<    SIMD(VL: 8)
            <<<    PREFETCH(HARD) Expected by compiler
            <<<      (unknown)
            <<< Loop-information  End >>>
33        5v    for (i = 0; i < n; i++){
34        5v      b[i] = c0 + a[i]*(c1 + a[i]*(c2 + a[i]*(c3 + a[i]*
35                    (c4 + a[i]*(c5 + a[i]*(c6 + a[i]*(c7 + a[i]*
36                    (c8 + a[i]*c9))))))));
37        5v    }
38            }
```

アンロール数5に指定

ソフトウェアパイプライニングの抑止

アンロール展開数 5指定により
アンロール展開数 5×8 (SIMD) = 40
40回転全てアンローリングされたループが実行される。



適切なアンローリング展開数の指定およびソフトウェアパイプライニングの抑止（最適化制御行）

FUJITSU

以下の最適化指示子を指定します。または、翻訳オプションで指定することも可能です。

最適化指示子 (Fortran)	意味	指定可能な最適化制御行			
		プログラム単位	DOループ単位	文単位	配列代入文単位
UNROLL(*n*)	DO ループをアンローリングさせます。*n* は展開数（多重度）を表す2 ～ 100の10 進数です。	×	○	×	×
NOSWP	ソフトウェアパイプライニング機能を無効にします。	○	○	×	○

最適化指示子 (C/C++)	意味	指定可能な最適化制御行			
		global行	procedure行	loop行	statement行
unroll(*n*)	ループをアンローリングさせます。*n* は展開数（多重度）を表す2 ～ 100の10 進数です。	×	×	○	×
noswp	ソフトウェアパイプライニング機能を無効にします。	○	○	○	×

翻訳オプション	機能説明
-Kunroll[=*N*] (2≦*N*≦100)	ループアンローリングの最適化を行うことを指示します。Nにはループ展開数の上限を指定します。Nの指定を省略した場合、コンパイラが自動的に最良な値を決定します。-O0または-O1オプションが有効な場合のデフォルトは-Knounroll、-O2オプション以上が有効な場合のデフォルトは-Kunrollです。
-Knoswp	ソフトウェアパイプライニングの最適化を行わないことを指示します。

■使用例

```
$ frtpx -Kfast,parallel sample.f90 -Kunroll=5,noswp
$ fccpx -Kfast,parallel sample.f90 -Kunroll=5,noswp
```

◆注意事項

clangモードではアンローリングの最適化機能を使用できません

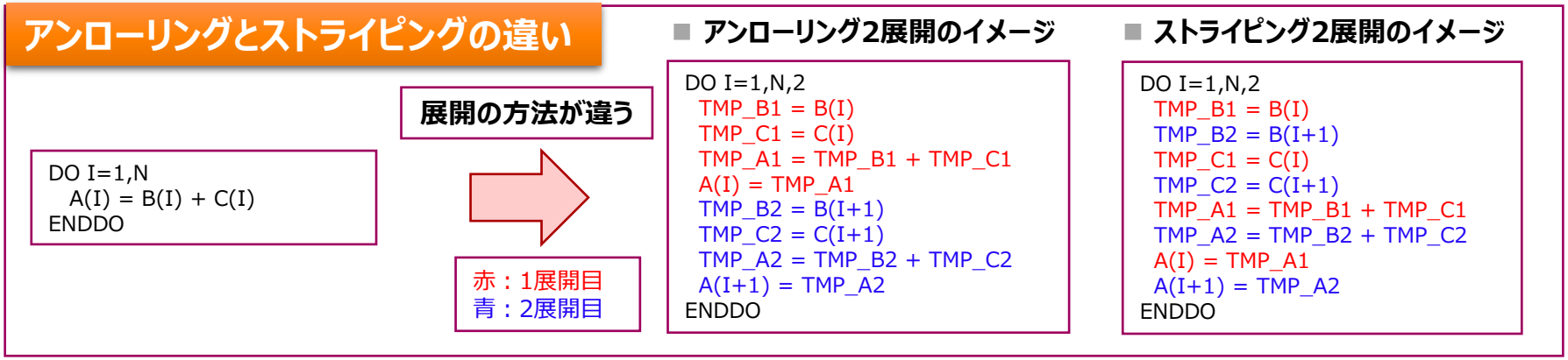
FUJITSU

ストライピング（インターリーブ）展開数の指定および ソフトウェアパイプライニングの抑止

- ループ展開
- ストライピング（インターリーブ）展開数の指定およびソフトウェアパイプライニングの抑止（改善前）
- ストライピング（インターリーブ）展開数の指定およびソフトウェアパイプライニングの抑止（最適化制御行チューニング）
- ストライピング（インターリーブ）展開数の指定およびソフトウェアパイプライニングの抑止（最適化制御行）

ループ展開

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



- ポイント
  - SIMD化、ソフトウェアパイプライニングとの調整で効果が見込める可能性があるため、unrollを先に適用して効果がなかった場合にストライピングを適用することをお勧めします。
  - unrollよりstripingの方が使用レジスタ数が増加するため、ストライプ長nを多くすると実行性能が低下する場合があります。
  - -Kstripingオプションと-Kunrollオプションを同時に指定した場合、後に指定した方が有効となります。

Fortran ストライピング（インターリーブ）展開数の指定およびソフトウェアパイプライニングの抑止（改善前）

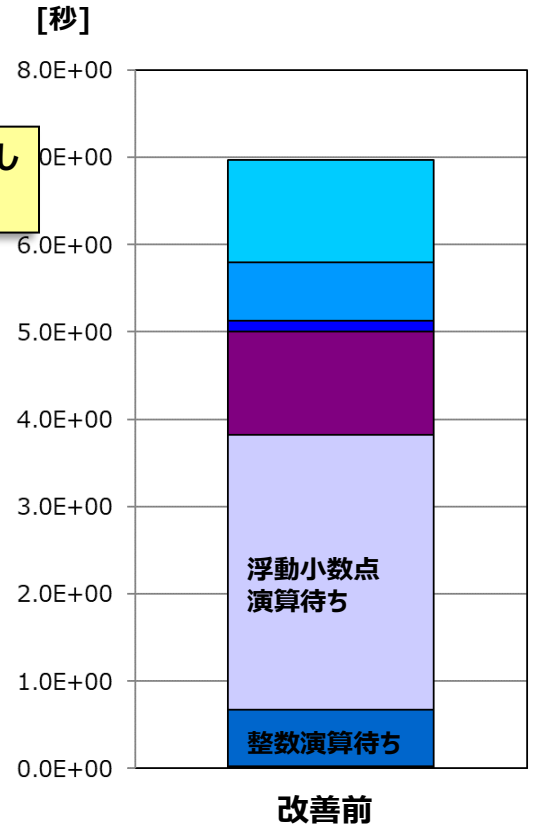
FUJITSU

繰り返し数が少なく、アンローリングやソフトウェアパイプライニングが効果的に機能していません。そのため、浮動小数点演算待ちや整数演算待ちが多くなっています。

改善前ソース

```
                <<< Loop-information Start >>>
                <<<  [PARALLELIZATION]
                <<<    Standard iteration count:
                <<<  [OPTIMIZATION]
                <<<    SIMD(VL: 8)
                <<<    SOFTWARE PIPELINING(IPC: 1.62, ITR: 176,
                                                MVE: 6, POL: S)
                <<<    PREFETCH(HARD) Expected by compiler :
                <<<      a, b
                <<< Loop-information  End >>>
39    1  pp   2v      do i = 1 , n
40    1   p   2v        b(i) = c0 + a(i)*(c1 + a(i)*(c2 + a(i)*(c3 + a(i)*
41    1             &      (c4 + a(i)*(c5 + a(i)*(c6 + a(i)*(c7 + a(i)*
42    1             &      (c8 + a(i)*c9)))))))
43    1   p   2v       enddo
```

ループ繰り返し数n=40 に対し
アンロール展開数2



問題点：ループの繰り返し数が少ない

・ソフトウェアパイプライニングのループが実行されない
ソフトウェアパイプライニングの条件ITR: 176以下である。

・アンローリング展開数が適切でない
アンロール展開数 2 × 8 (SIMD) = 16
8回転がオリジナルのループが実行されてしまう。
オリジナルのループが実行されることによる性能への影響は大きい。

Fortran ストライピング（インターリーブ）展開数の指定および ソフトウェアパイプライニングの抑止（最適化制御行チューニング）

FUJITSU

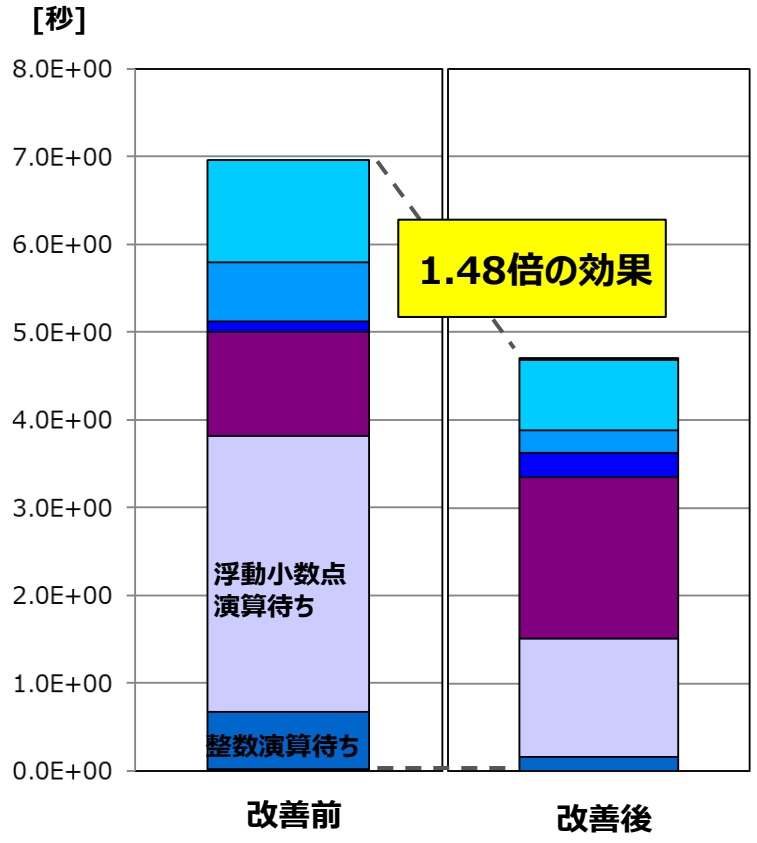
繰り返し数に応じたストライピング展開数の指定およびソフトウェアパイプライニングの抑止により、適切な命令スケジューリングが行われました。
その結果、浮動小数点演算待ち・整数演算待ちが削減されました。

改善後ソース（最適化制御行チューニング）

```
39                !ocl striping(5)
40                !ocl nounroll
41                !ocl noswp
        <<< Loop-information Start >>>
        <<<  [PARALLELIZATION]
        <<<    Standard iteration count: 552
        <<<  [OPTIMIZATION]
        <<<    SIMD(VL: 8)
        <<<    PREFETCH(HARD) Expected by compiler
        <<<      a, b
        <<< Loop-information  End >>>
42    1  pp   5v    do i = 1 , n
43    1   p   5v       b(i) = c0 + a(i)*(c1 + a(i)*(c2 + a(i)*(c3 + a(i)*
44    1              &     (c4 + a(i)*(c5 + a(i)*(c6 + a(i)*(c7 + a(i)*
45    1              &     (c8 + a(i)*c9)))))))
46    1   p   5v    enddo
```

ストライピング数5に指定

アンロール、ソフトウェアパイプライニングの抑止



ストライピング展開数 5指定により
ストライピング展開数 5×8 (SIMD) = 40
40回転全てストライピングされたループが実行される。

C/C++ ストライピング（インターリーブ）展開数の指定およびソフトウェアパイプライニングの抑止（改善前）

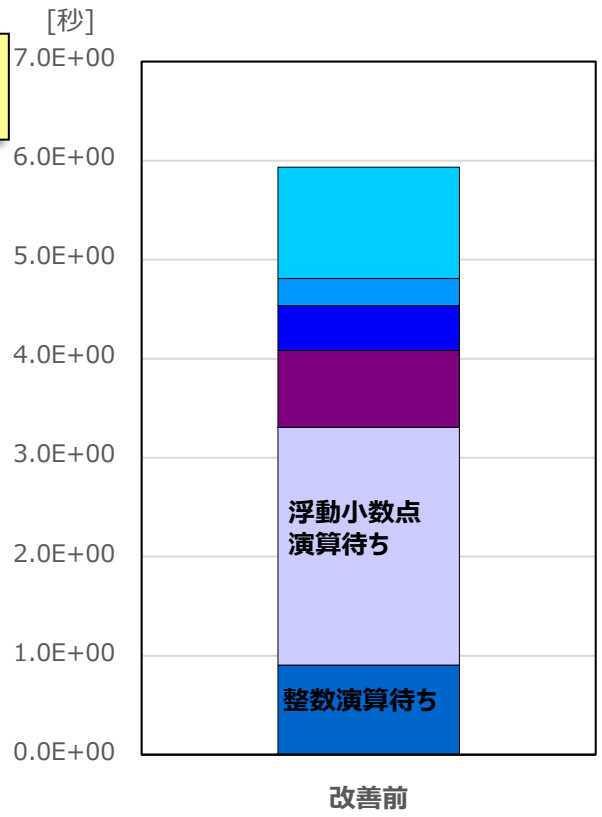
FUJITSU

繰り返し数が少なく、アンローリングやソフトウェアパイプライニングが効果的に機能していません。そのため、浮動小数点演算待ちや整数演算待ちが多くなっています。

改善前ソース

```
<<< Loop-information Start >>>
        <<<  [OPTIMIZATION]
        <<<    SIMD(VL: 8)
        <<<    SOFTWARE PIPELINING(IPC: 1.62, ITR: 176,
                        MVE: 6, POL: S)
        <<<    PREFETCH(HARD) Expected by compiler :
        <<<      (unknown)
        <<< Loop-information  End >>>
33      2v    for (i = 0; i < n; i++){
34      2v      b[i] = c0 + a[i]*(c1 + a[i]*(c2 + a[i]*(c3 + a[i]*
35                (c4 + a[i]*(c5 + a[i]*(c6 + a[i]*(c7 + a[i]*
36                (c8 + a[i]*c9)))))));
37      2v    }
38          }
```

ループ繰り返し数n=40 に対し
アンロール展開数2



問題点：ループの繰り返し数が少ない

・ソフトウェアパイプライニングのループが実行されない
ソフトウェアパイプライニングの条件ITR: 176以下である。

・アンローリング展開数が適切でない
アンロール展開数 2 × 8 (SIMD) = 16
8回転がオリジナルのループが実行されてしまう。
オリジナルのループが実行されることによる性能への影響は大きい。

C/C++ ストライピング（インターリーブ）展開数の指定および ソフトウェアパイプライニングの抑止（最適化制御行チューニング）

FUJITSU

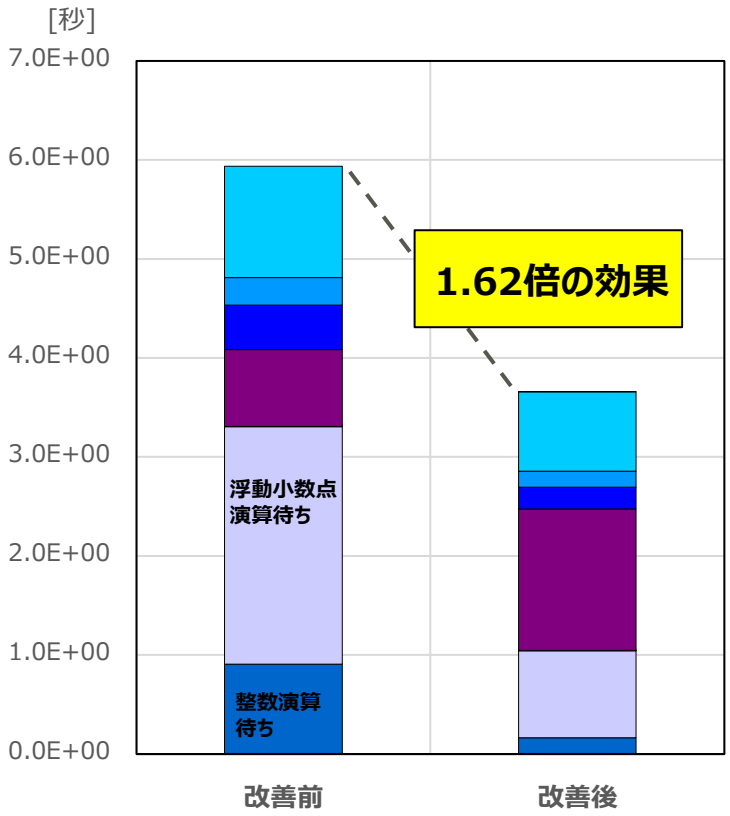
繰り返し数に応じたストライピング展開数の指定およびソフトウェアパイプライニングの抑止により、適切な命令スケジューリングが行われました。
その結果、浮動小数点演算待ち・整数演算待ちが削減されました。

改善後ソース（最適化制御行チューニング）

```
33          #pragma loop striping 5
34          #pragma loop nounroll
35          #pragma loop noswp
            <<< Loop-information Start >>>
            <<<  [OPTIMIZATION]
            <<<    SIMD(VL: 8)
            <<<    STRIPING
            <<<    PREFETCH(HARD) Expected by compiler :
            <<<      (unknown)
            <<< Loop-information  End >>>
36        v   for (i = 0; i < n; i++){
37        v     b[i] = c0 + a[i]*(c1 + a[i]*(c2 + a[i]*(c3 + a[i]*
38                  (c4 + a[i]*(c5 + a[i]*(c6 + a[i]*(c7 + a[i]*
39                  (c8 + a[i]*c9)))))));
40        v   }
41          }
```

ストライピング数5に指定

アンロール、ソフトウェアパイプライニングの抑止



ストライピング展開数 5指定により
ストライピング展開数 5×8 (SIMD) = 40
40回転全てストライピングされたループが実行される。

ストライピング（インターリーブ）展開数の指定およびソフトウェアパイプライニングの抑止（最適化制御行）

FUJITSU

以下の最適化指示子を指定します。または、翻訳オプションで指定することも可能です。

最適化指示子 (Fortran)	意味	指定可能な最適化制御行			
		プログラム単位	DOループ単位	文単位	配列代入文単位
STRIPING[(n)]	ループストライピング機能を有効にします。nは展開数(多重度)を表す2～100の10進数です。	○	○	×	○
NOSWP	ソフトウェアパイプライニング機能を無効にします。	○	○	×	○

最適化指示子 (C/C++)	意味	指定可能な最適化制御行			
		global行	procedure行	loop行	statement行
striping[(n)]	ループストライピング機能を有効にします。nは展開数(多重度)を表す2～100の10進数です。	○	○	○	×
noswp	ソフトウェアパイプライニング機能を無効にします。	○	○	○	×

翻訳オプション	機能説明
-Kstriping[=*N*] (2≦*N*≦100)	ループストライピングの最適化を行うかどうかを指示します。ストライプ長(展開数)をNで指定することができます。Nは2 から100 までの値を指定することができます。N の値が省略された場合、コンパイラが自動的に値を決定します。ソースプログラムでループの繰返し数が既知の場合、Nに繰返し数を超える値を指定しても、コンパイラが自動的に決定した展開数が有効となります。デフォルトは-Knostripingです。
-Knoswp	ソフトウェアパイプライニングの最適化を行わないことを指示します。

■使用例

```
$ frtpx -Kfast,parallel sample.f90 -Kstriping=5,noswp
$ fccpx -Kfast,parallel sample.f90 -Kstriping=5,noswp
```

◆注意事項

clangモードではストライピング機能を使用できません

FUJITSU

外側ループでのソフトウェアパイプライニング

- 外側ループでのソフトウェアパイプライニングとは
- 外側ループでのソフトウェアパイプライニング（改善前）
- 外側ループでのソフトウェアパイプライニング（ソースチューニング）
- 外側ループでのソフトウェアパイプライニング（CLONE利用）
- 外側ループでのソフトウェアパイプライニング（CLONE利用）（改善前）
- 外側ループでのソフトウェアパイプライニング（CLONE利用）（ソースチューニング）

外側ループでのソフトウェアパイプライニングとは

- 最内ループの繰り返し数が少ない場合は、ストリップマイニング等を行って最内ループの繰り返し数をSIMD長と等しくすることで、その外側のループでソフトウェアパイプライニングを促進させることが可能です。

改善前ソース

```
24    3  p           do j=1,M
                <<< Loop-information Start >>>
                <<<  [OPTIMIZATION]
                <<<    SIMD(VL: 8)
                <<<    SOFTWARE PIPELINING(IPC: 3.00, ITR: 192,
                                           MVE: 7, POL: S)
                <<<    PREFETCH(HARD) Expected by compiler :
                <<<      b, a
                <<< Loop-information  End >>>
25    4  p  2v         do i=1,N
26    4  p  2v           a(i,j,k)=a(i,j,k)+c*b(i,j,k)
27    4  p  2v         enddo
28    3  p          enddo
```

改善後ソース

```
25    3  p           do ii=1,N,blk
                <<< Loop-information Start >>>
                <<<  [OPTIMIZATION]
                <<<    SOFTWARE PIPELINING(IPC: 0.31, ITR: 192,
                                           MVE: 2, POL: L)
                <<<    PREFETCH(HARD) Expected by compiler :
                <<<      b, a
                <<< Loop-information  End >>>
26    4  p  8         do j=1,M
                <<< Loop-information Start >>>
                <<<  [OPTIMIZATION]
                <<<    SIMD(VL: 8)
                <<<    FULL UNROLLING
                <<< Loop-information  End >>>
27    5  p  fv          do i=ii,ii+blk-1
28    5  p  fv            a(i,j,k)=a(i,j,k)+c*b(i,j,k)
29    5  p  fv          enddo
30    4  p  8         enddo
31    3  p          enddo
```

上記の例は固定長SIMDの場合に有効です。
SIMD幅512[bit]の時のSIMD長は以下の通りです。

SIMD幅(Vector Length)=512[bit]の場合のSIMD長

データ型	SIMD長
倍精度型	8
単精度型	16
半精度型	32
1バイト型	64

Fortran 外側ループでのソフトウェアパイプライニング（改善前）

FUJITSU

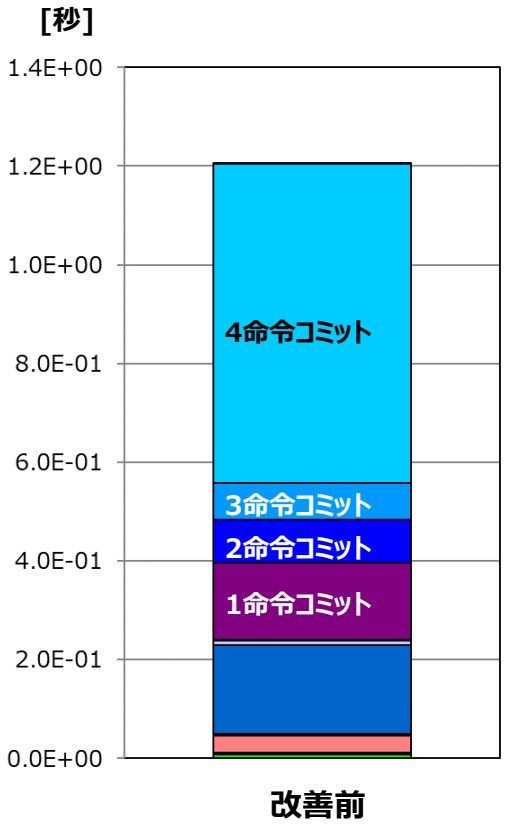
最内ループの繰り返し数Nがソフトウェアパイプライニングの条件に対して小さいため、ソフトウェアパイプライニングが適用されていません。

改善前ソース

```
2                integer,parameter::N=16,M=200,L=48
 :
18                real(8)::a(N,M,L),b(N,M,L)
19                real(8),parameter::c=0.5
20               !$omp parallel private(iter,i,j,k)
21    1            do iter=1,itmax
22    1           !$omp do
23    2  p           do k=1,L
                <<< Loop-information Start >>>
                <<<  [OPTIMIZATION]
                <<<    PREFETCH(HARD) I
                <<<      b, a
                <<< Loop-information  En
24    3  p             do j=1,M
                <<< Loop-information Sta
                <<<  [OPTIMIZATION]
                <<<    SIMD(VL: 8)
                <<<    SOFTWARE PIPELINING(IPC: 3.00, ITR: 192,
                                           MVE: 7, POL: S)
                <<<    PREFETCH(HARD) Expected by compiler :
                <<<      b, a
                <<< Loop-information  End >>>
25    4  p  2v           do i=1,N
26    4  p  2v             a(i,j,k)=a(i,j,k)+c*b(i,j,k)
27    4  p  2v           enddo
28    3  p             enddo
29    2  p           enddo
30    1            !$omp enddo nowait
31    1             enddo
32                 !$omp end parallel
```

最内ループの繰り返し数Nが ITR:192よりも少ないため ソフトウェアパイプライニングの 処理ルートに入らない

N=16



	GFLOPS	Effective instruction
改善前	50.98	7.53E+10

Fortran 外側ループでのソフトウェアパイプライニング（ソースチューニング）

FUJITSU

ストリップマイニングを行い最内ループをSIMD長に固定しました。その結果、外側のループでSWPLが促進され、演算効率・有効命令数が改善しました。
C/C++ではループの初期値と終値に変数が含まれている場合に、ループを全展開できずループ中に分岐が残ります。そのため、外側ループにSWPLを適用できません。

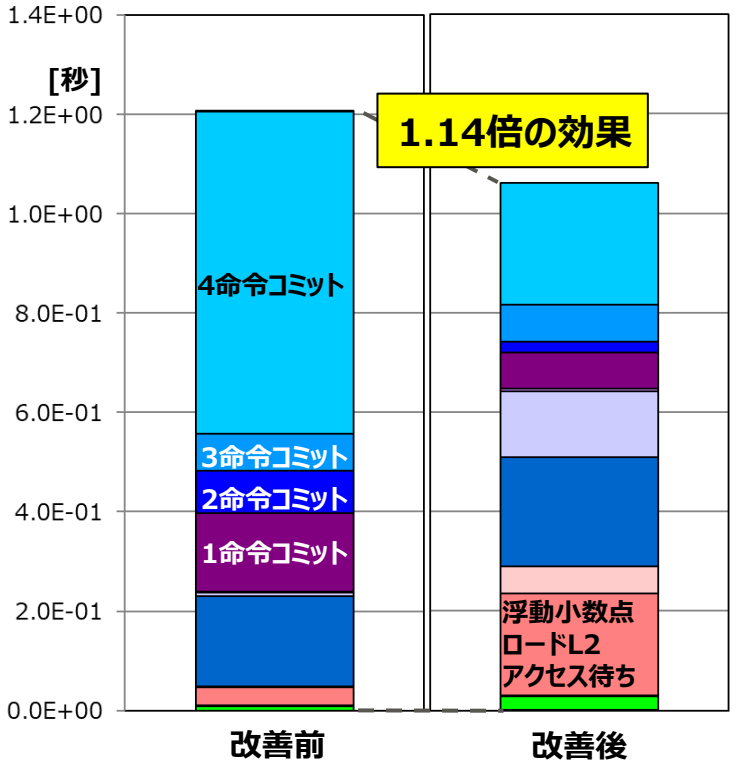
改善後ソース

```
2                integer,parameter::N=16,M=200,L=48
:
18                real(8)::a(N,M,L),b(N,M,L)
19                real(8),parameter::c=0.5
20                integer,parameter::blk=8
21               !$omp parallel private(iter,ii,i,j,k)
22    1            do iter=1,itmax
23    1           !$omp do
24    2  p          do k=1,L
                <<< Loop-information Start >>>
                <<<  [OPTIMIZATION]
                <<<    PREFETCH(HARD) Expected b
                <<<      b, a
                <<< Loop-information  End >>>
25    3  p            do ii=1,N,blk
                <<< Loop-information Start >>>
                <<<  [OPTIMIZATION]
                <<<    SOFTWARE PIPELINING(IPC: 0.31, ITR: 192,
                                 MVE: 2, POL: L)
                <<<    PREFETCH(HARD) Expected by compiler :
                <<<      b, a
                <<< Loop-information  End >>>
26    4  p  8           do j=1,M
                <<< Loop-information Start >>>
                <<<  [OPTIMIZATION]
                <<<    SIMD(VL: 8)
                <<<    FULL UNROLLING
                <<< Loop-information  End >>>
27    5  p  fv            do i=ii,ii+blk-1
28    5  p  fv              a(i,j,k)=a(i,j,k)+c*b(i,j,k)
29    5  p  fv            enddo
30    4  p  8           enddo
31    3  p             enddo
32    2  p           enddo
33    1            !$omp enddo nowait
34    1             enddo
35                 !$omp end parallel
```

ソフトウェアパイプライニングが適用される

M=200

最内ループの繰り返し数をSIMD長に固定



	GFLOPS	Effective instruction
改善前	50.98	7.53E+10
改善後	57.95	3.19E+10

外側ループのソフトウェアパイプライニング（CLONE利用）

FUJITSU

最内ループの繰り返し回数が短く固定値の場合には、クローンチューニングを利用することもできます。

- クローンチューニングとは
  CLONE指示子を用いて、ループに対して変数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最適化を促進するチューニング方法です。

ソース例

```
!ocl clone(N==8)
do i=1,N
   A(i) = i
enddo
```

最適化後（ソースイメージ）

```
If (N==8) then
   do i=1,8
      A(i)=i
   endo
else
   do i=1,N
      A(i)=i
   enddo
endif
```

N==8の時のループを複製
ループ長が固定されることで
他の最適化が促進される

最適化指示子(Fortran)	意味	指定可能な最適化制御行			
		プログラム単位	DOループ単位	文単位	配列代入文単位
CLONE(var==n1[,n2]…)	ループ内で変数varが不変とみなして、引数で指示をした条件分岐を生成し、条件節内にループを複製する最適化を行うことを指示します。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]…の等式とします。	×	○	×	○

最適化指示子(C/C++)	意味	指定可能な最適化制御行			
		global行	procedure行	loop行	statement行
clone(var==n1[,n2]…)	ループ内で変数varが不変とみなして、引数で指示をした条件分岐を生成し、条件節内にループを複製する最適化を行うことを指示します。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]…の等式とします。	×	×	○	×

Fortran 外側ループでのソフトウェアパイプライニング（CLONE利用）（改善前）

FUJITSU

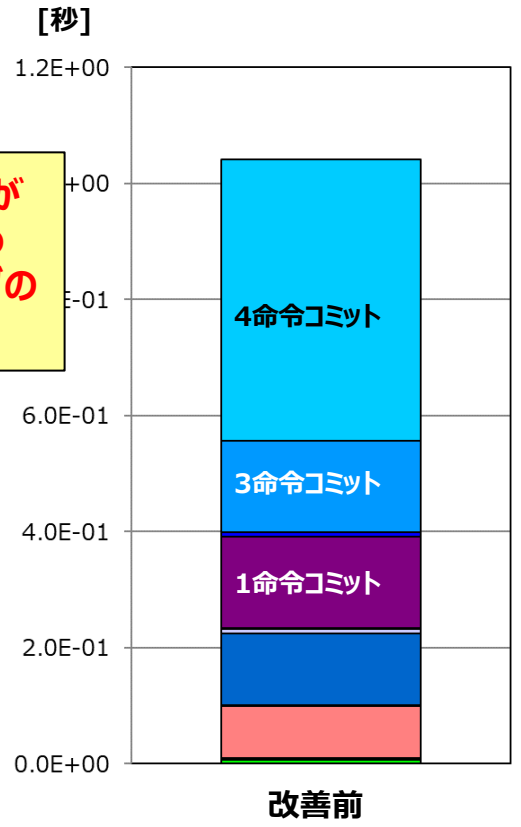
最内ループの繰り返し数Nが小さいため、ソフトウェアパイプライニングの適用条件を満たしていません。

改善前ソース

```
2                integer,parameter::N=8,M=240,L=48
:
18                real(8)::a(N,M,L),b(N,M,L)
19                real(8),parameter::c=0.5
20               !$omp parallel private(iter,i,j,k)
21    1            do iter=1,itmax
22    1           !$omp do
23    2  p          do k=1,L
                <<< Loop-information Start >>>
                <<<  [OPTIMIZATION]
                <<<    PREFETCH(HARD) Expected by compiler :
                <<<      b, a
                <<< Loop-information  End >>>
24    3  p            do j=1,M
                <<< Loop-information Start >>>
                <<<  [OPTIMIZATION]
                <<<    SIMD(VL: 8)
                <<<    SOFTWARE PIPELINING(IPC: 3.00, ITR: 192,
                                                  MVE: 7, POL: S)
                <<<    PREFETCH(HARD) Expected by compiler :
                <<<      b, a
                <<< Loop-information  End >>>
25    4  p  2v          do i=1,N
26    4  p  2v            a(i,j,k)=a(i,j,k)+c*b(i,j,k)
27    4  p  2v          enddo
28    3  p            enddo
29    2  p          enddo
30    1           !$omp enddo nowait
31    1            enddo
32               !$omp end parallel
```

最内ループの繰り返し数Nが ITR:192よりも少ないため ソフトウェアパイプライニングの 処理ルートに入らない

N=8



	GFLOPS	Effective instruction
改善前	29.51	6.18E+10

Fortran 外側ループでのソフトウェアパイプライニング（CLONE利用）（ソースチューニング）

FUJITSU

CLONEを利用し最内ループの繰り返し数をSIMD長に固定しました。その結果、外側のループでソフトウェアパイプライニングが促進され、演算効率・有効命令数が改善しました。

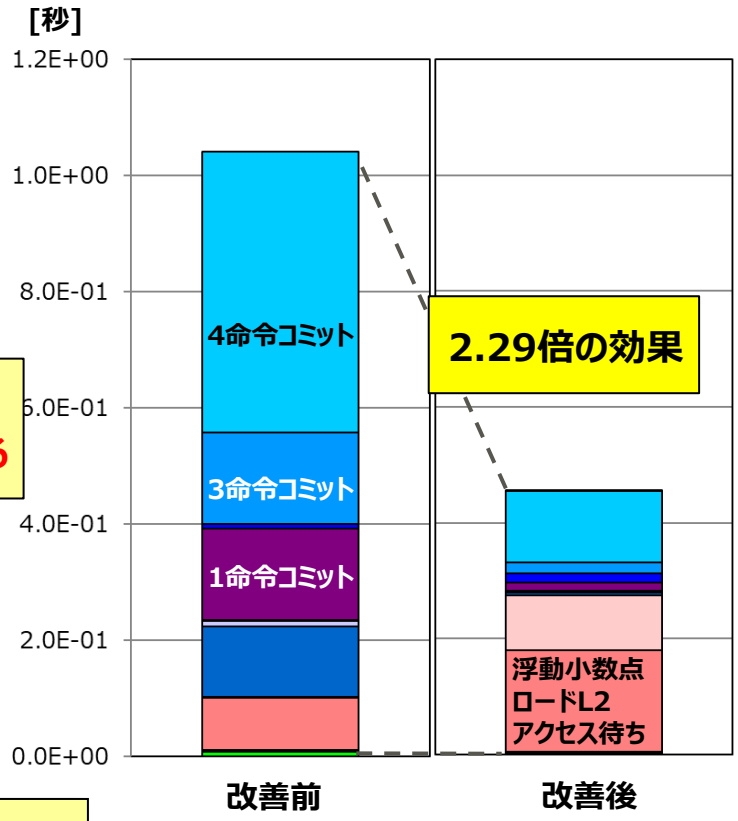
改善後ソース（ソースチューニング）

```
2                integer,parameter::N=8,M=240,L=48
:
18                real(8)::a(N,M,L),b(N,M,L)
19                real(8),parameter::c=0.5
20                integer NN
21                NN = N
22               !$omp parallel private(iter,i,j,k)
23    1            do iter=1,itmax
24    1           !$omp do
25    1           !ocl clone(NN==8)
                <<< Loop-information Start >>>
                <<<  [OPTIMIZATION]
                <<<    CLONE
                <<< Loop-information  End >>>
26    2  p           do k=1,L
                <<< Loop-information Start >>>
                <<<  [OPTIMIZATION]
                <<<    SOFTWARE PIPELINING(IPC: 3.75, ITR: 208,
                                           MVE: 6, POL: S)
                <<< Loop-information  End >>>
27    3  p  2         do j=1,M
                <<< Loop-information Start >>>
                <<<  [OPTIMIZATION]
                <<<    SIMD(VL: 8)
                <<<    SOFTWARE PIPELINING(IPC: 3.00, ITR: 192,
                                           MVE: 7, POL: S)
                <<< Loop-information  End >>>
28    4  p  2v          do i=1,NN
29    4  p  2v            a(i,j,k)=a(i,j,k)+c*b(i,j,k)
30    4  p  2v          enddo
31    3  p  2         enddo
32    2  p           enddo
33    1            !$omp enddo nowait
34    1              enddo
35                  !$omp end parallel
```

ソフトウェアパイプライニングが適用される

M=240

最内ループの繰り返し数をSIMD長に固定



	GFLOPS	Effective instruction
改善前	29.51	6.18E+10
改善後	67.60	1.44E+10

C/C++ 外側ループでのソフトウェアパイプライニング（CLONE利用）（改善前）

FUJITSU

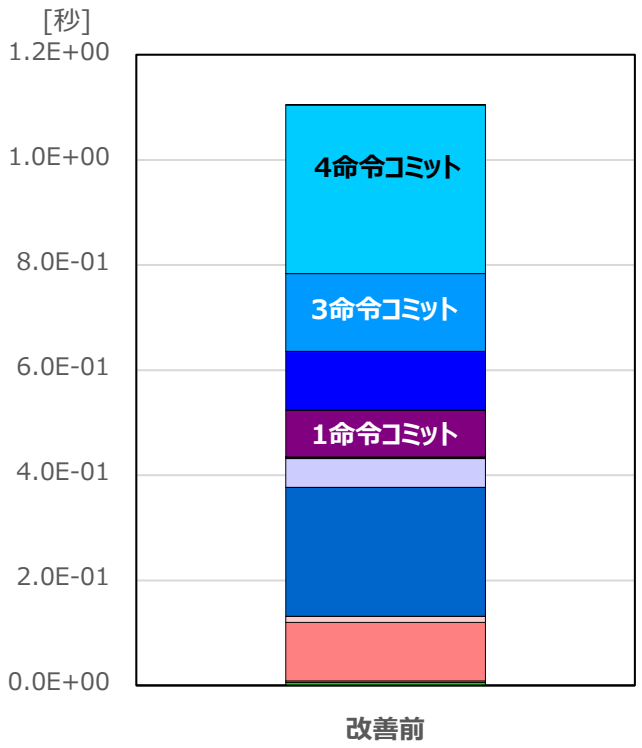
最内ループの繰り返し数Nが小さいため、ソフトウェアパイプライニングの適用条件を満たしていません。

改善前ソース

```
35          #pragma omp parallel private(iter,i,j,k)
36          {
37            for (iter = 0; iter < itmax; iter++){
38          #pragma omp for nowait
39    p         for (k = 0; k < L; k++){
          <<< Loop-information Sta
          <<<  [OPTIMIZATION]
          <<<    PREFETCH(HARD)
          <<<      (unknown)
          <<< Loop-information  En
40    p         for (j = 0; j < M; j++
          <<< Loop-information Sta
          <<<  [OPTIMIZATION]
          <<<    SIMD(VL: 8)
          <<<    SOFTWARE PIPELINING(IPC: 3.00, ITR: 192,
                     MVE: 7, POL: S)
          <<<    PREFETCH(HARD) Expected by compiler :
          <<<      (unknown)
          <<< Loop-information  End >>>
41    p   2v        for (i = 0; i < N; i++){
42    p   2v          a[k][j][i] = a[k][j][i] + c * b[k][j][i];
43    p   2v        }
44    p           }
45    p         }
46              }
47            }
```

最内ループの繰り返し数NがITR:192よりも少ないためソフトウェアパイプライニングの処理ルートに入らない

N=8



Statistics	GFLOPS	Effective instruction
改善前	33.39	4.87E+10

C/C++ 外側ループでのソフトウェアパイプライニング（CLONE利用）（ソースチューニング）

FUJITSU

CLONEを利用し最内ループの繰り返し数をSIMD長に固定しました。その結果、外側のループでソフトウェアパイプライニングが促進され、演算効率・有効命令数が改善しました。

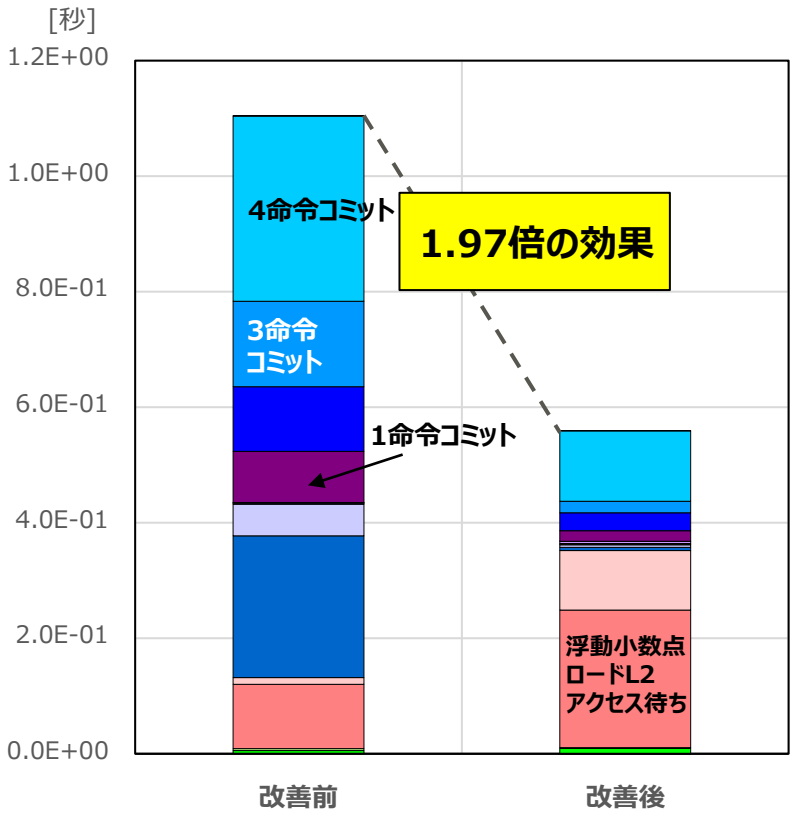
改善後ソース（ソースチューニング）

```
37          #pragma omp parallel private(iter,i,j,k)
38          {
39            for (iter = 0; iter < itmax; iter++){
40          #pragma omp for nowait
41          #pragma loop clone NN==8
           <<< Loop-information Start >>>
           <<<  [OPTIMIZATION]
           <<<    CLONE
           <<<    PREFETCH(HARD) E
           <<<      (unknown)
           <<< Loop-information  En
42   p          for (k = 0; k < L; k++){
           <<< Loop-information Start >>>
           <<<  [OPTIMIZATION]
           <<<    SOFTWARE PIPELINING(IPC: 3.25, ITR: 176,
                                MVE: 6, POL: S)
           <<<    PREFETCH(HARD) Expected by compiler :
           <<<      (unknown)
           <<< Loop-information  End >>>
43   p    2        for (j = 0; j < M; j++){
            <<< Loop-information Start >>>
            <<<  [OPTIMIZATION]
            <<<    SIMD(VL: 8)
            <<<    SOFTWARE PIPELINING(IPC: 3.00, ITR: 192,
                                MVE: 7, POL: S)
            <<<    PREFETCH(HARD) Expected by compiler :
            <<<      (unknown)
            <<< Loop-information  End >>>
44   p    2v         for (i = 0; i < NN; i++){
45   p    2v           a[k][j][i] = a[k][j][i] + c * b[k][j][i];
46   p    2v         }
47   p    2        }
48   p           }
49             }
50           }
```

ソフトウェアパイプライニングが適用される

M=240

最内ループの繰り返し数をSIMD長に固定



Statistics	GFLOPS	Effective instruction
改善前	33.39	4.87E+10
改善後	65.89	1.50E+10

FUJITSU

リローリング

- リローリングとは
- リローリング（改善前）
- リローリング（ソースチューニング）

リローリングとは

リローリングとはアンローリング展開された文をループ文に戻すことで、ループに対する最適化を促進するチューニングです。

アンローリング
SIMD化を想定しないコードでは頻繁に行われたチューニング

通常のループ

```
do k=1,m
  do j=1,n
    do i=1,8
      a(i,j,k) = b(i,j,k) + a(i,j,k)
    enddo
  enddo
enddo
```

手動アンローリング

```
do k=1,m
  do j=1,n
    a(1,j,k) = b(1,j,k) + a(1,j,k)
    a(2,j,k) = b(2,j,k) + a(2,j,k)
    a(3,j,k) = b(3,j,k) + a(3,j,k)
    a(4,j,k) = b(4,j,k) + a(4,j,k)
    a(5,j,k) = b(5,j,k) + a(5,j,k)
    a(6,j,k) = b(6,j,k) + a(6,j,k)
    a(7,j,k) = b(7,j,k) + a(7,j,k)
    a(8,j,k) = b(8,j,k) + a(8,j,k)
  enddo
enddo
```

リローリング

```
do k=1,m
  do j=1,n
    do i=1,8
      a(i,j,k) = b(i,j,k) + a(i,j,k)
    enddo
  enddo
enddo
```

アンロールされたコードの欠点
ディスタンスアクセスとなりGather Load,
Scatter Store命令が多発してしまう。

リローリング
アンロール展開されたループを元のループに戻す

Fortran リローリング（改善前）

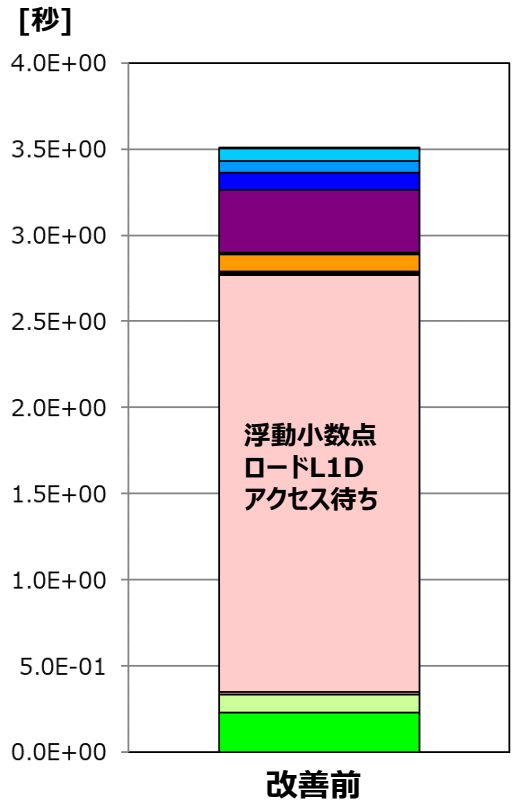
FUJITSU

手動アンロール展開されています。Gather Load, Scatter Storeが多発し、浮動小数点ロードL1Dアクセス待ちが多くなっています。

改善前ソース

```
40                    real(8) :: a(8,N,M),b(8,N,M)
41
                  <<< Loop-information Start >>>
                  <<<  [PARALLELIZATION]
                  <<<    Standard iteration count: 2
                  <<<  [OPTIMIZATION]
                  <<<    PREFETCH(HARD) Expected by compiler :
                  <<<      b, a
                  <<< Loop-information  End >>>
42    1  pp           DO K=1,M
                  <<< Loop-information Start >>>
                  <<<  [OPTIMIZATION]
                  <<<    SIMD(VL: 8)
                  <<<    SOFTWARE PIPELINING(IPC: 2.04, ITR: 40,
                                                MVE: 3, POL: S)
                  <<<    PREFETCH(HARD) Expected by compiler :
                  <<<      b, a
                  <<< Loop-information  End >>>
43    2  p   v         DO J=1,N
44    2  p   v           a(1,J,K) = b(1,J,K) + a(1,J,K)
45    2  p   v           a(2,J,K) = b(2,J,K) + a(2,J,K)
46    2  p   v           a(3,J,K) = b(3,J,K) + a(3,J,K)
47    2  p   v           a(4,J,K) = b(4,J,K) + a(4,J,K)
48    2  p   v           a(5,J,K) = b(5,J,K) + a(5,J,K)
49    2  p   v           a(6,J,K) = b(6,J,K) + a(6,J,K)
50    2  p   v           a(7,J,K) = b(7,J,K) + a(7,J,K)
51    2  p   v           a(8,J,K) = b(8,J,K) + a(8,J,K)
52    2  p   v         ENDDO
53    1  p           ENDDO
```

N=128
M=250



Fortran リローリング（ソースチューニング）

FUJITSU

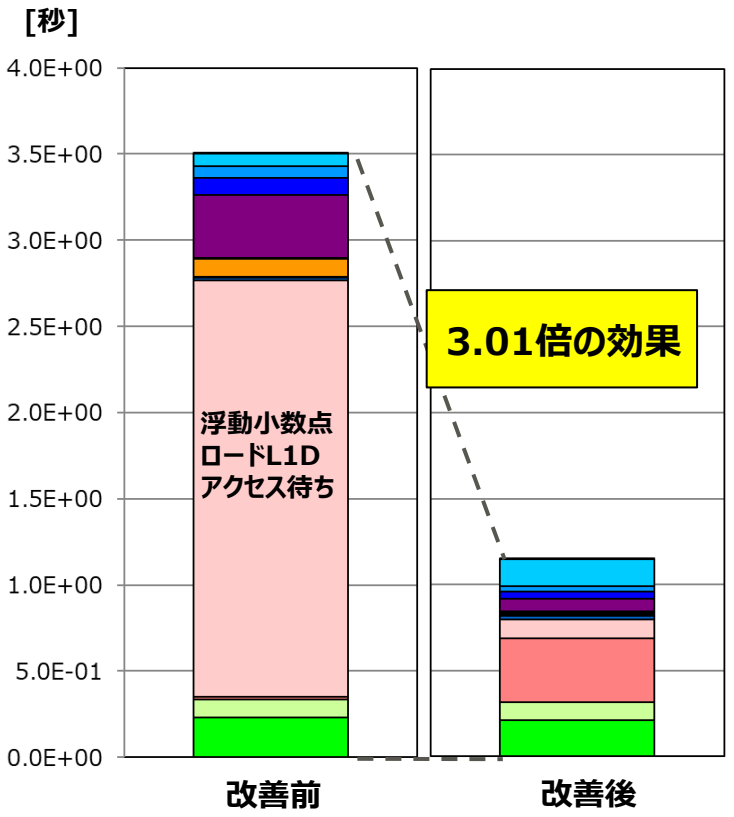
リローリングを行う（ループに戻す）ことでデータが連続アクセスとなり、効果的なソフトウェアパイプライニング、SIMD化、ループアンロールなどの最適化が促進されました。

改善後ソース（ソースチューニング）

```
40                real(8) :: a(8,N,M),b(8,N,M)
41
                <<< Loop-information Start >>>
                <<<  [PARALLELIZATION]
                <<<    Standard iteration c
                <<<  [OPTIMIZATION]
                <<<    COLLAPSED
                <<<    SIMD(VL: 8)
                <<<    SOFTWARE PIPELINING(IPC: 3.00, ITR: 192,
                                           MVE: 7, POL: S)
                <<<    PREFETCH(HARD) Expected by compiler :
                <<<      a, b
                <<< Loop-information  End >>>
42    1  pp  2v       DO K=1,M
                <<< Loop-information Start >>>
                <<<  [OPTIMIZATION]
                <<<    COLLAPSED
                <<< Loop-information  End
43    2  p   2          DO J=1,N
                <<< Loop-information Start
                <<<  [OPTIMIZATION]
                <<<    COLLAPSED
                <<< Loop-information  End >>>
44    3  p   2            DO I=1,8
45    3  p   2v             a(I,J,K) = b(I,J,K) + a(I,J,K)
46    3  p   2v           ENDDO
47    2  p             ENDDO
48    1  p           ENDDO
```

ループの1重化

SIMD化・ループアンロールが促進されている



C/C++ リローリング（改善前）

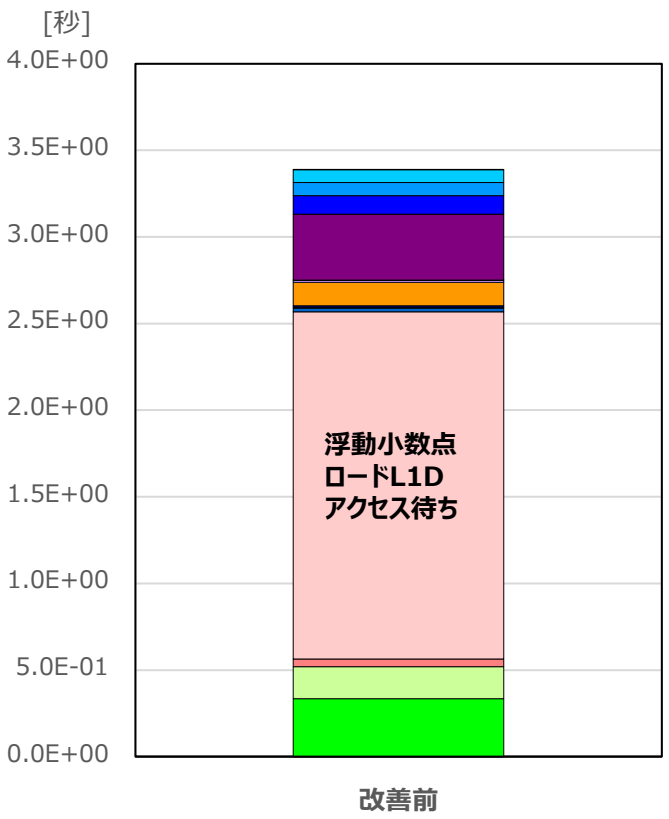
FUJITSU

手動アンロール展開されています。Gather Load, Scatter Storeが多発し、浮動小数点ロードL1Dアクセス待ちが多くなっています。

改善前ソース

N=128
M=250

```
50          #pragma omp parallel for private(j)
           <<< Loop-information Start >>>
           <<<  [OPTIMIZATION]
           <<<    PREFETCH(HARD) Expected by compiler :
           <<<      (unknown)
           <<< Loop-information  End >>>
51   p       for( k=0;k<M; k++)
52   p       {
           <<< Loop-information Start >>>
           <<<  [OPTIMIZATION]
           <<<    SIMD(VL: 8)
           <<<    SOFTWARE PIPELINING(IPC: 2.04, ITR: 40,
                                MVE: 3, POL: S)
           <<<    PREFETCH(HARD) Expected by compiler :
           <<<      (unknown)
           <<< Loop-information  End >>>
53   p     v     for(j=0; j<N; j++)
54   p     v     {
55   p     v       a[k][j][0] = b[k][j][0] + a[k][j][0];
56   p     v       a[k][j][1] = b[k][j][1] + a[k][j][1];
57   p     v       a[k][j][2] = b[k][j][2] + a[k][j][2];
58   p     v       a[k][j][3] = b[k][j][3] + a[k][j][3];
59   p     v       a[k][j][4] = b[k][j][4] + a[k][j][4];
60   p     v       a[k][j][5] = b[k][j][5] + a[k][j][5];
61   p     v       a[k][j][6] = b[k][j][6] + a[k][j][6];
62   p     v       a[k][j][7] = b[k][j][7] + a[k][j][7];
63   p     v     }
64   p       }
65
66            return;
67          }
```



C/C++ リローリング（ソースチューニング）

FUJITSU

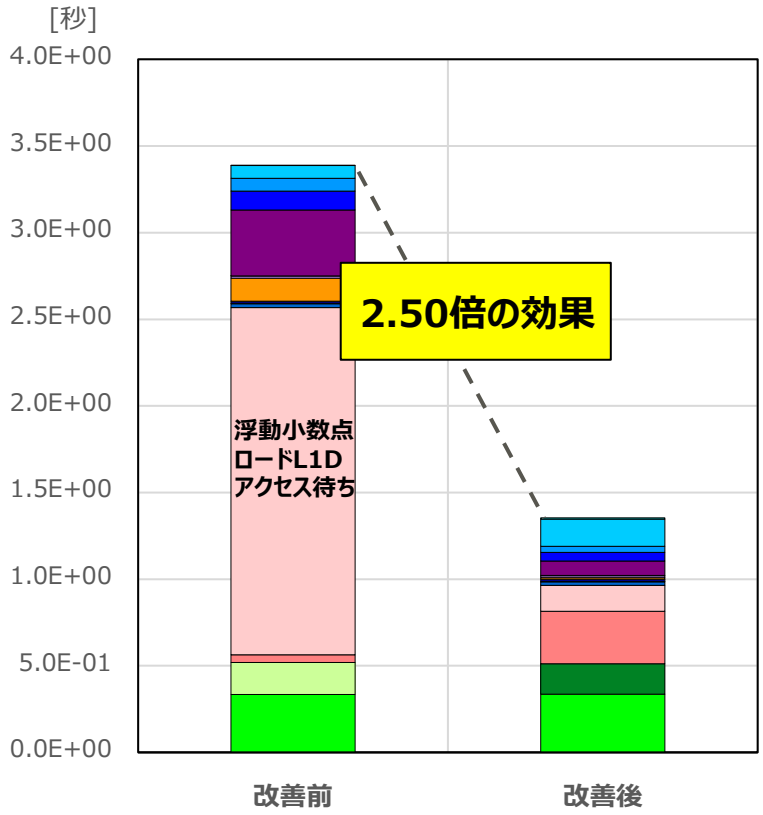
リローリングを行う（ループに戻す）ことでデータが連続アクセスとなり、効果的なソフトウェアパイプライニング、SIMD化、ループアンロールなどの最適化が促進されました。

改善後ソース（ソースチューニング）

```
50          #pragma omp parallel for private(i,j) collapse(3)
           <<< Loop-information Start >>>
           <<<  [OPTIMIZATION]
           <<<    SIMD(VL: 8)
           <<<    SOFTWARE PIPELINING(IPC: 3.00, ITR: 192,
                         MVE: 7, POL: S)
           <<<    PREFETCH(HARD) Expected by compiler :
           <<<      (unknown)
           <<< Loop-information  End >>>
51   p    2v   for (k=0; k< M; k++)
52        2    {
53   p    2      for(j=0; j<N; j++)
54        2      {
55   p    2        for(i=0; i<8; i++)
56   p    2v       {
57   p    2v         a[k][j][i] = b[k][j][i] + a[k][j][i];
58   p    2v       }
59   p    2v     }
60   p    2v   }
61
62             return;
63          }
```

ループの1重化

SIMD化・ループアンロールが促進されている



ループアンスイッチング

- ループアンスイッチングとは
- ループアンスイッチングの効果（改善前）
- ループアンスイッチングの効果（改善後）

ループアンスイッチングとは

ループ内に不変な状態で分岐するIF構文が存在する場合に、そのIF構文をループの外側に出し、IF構文の条件が成立した場合のループと、成立しない場合のループを作成する最適化です。

ソースコード

```
!$omp do
  do i=1,n1
!ocl unswitching
    if (n1 >= q) then
      処理①
    endif
!ocl unswitching
    if(n1 > r) then
      処理②
    endif
!ocl unswitching
    if(n1 < s) then
   処理③
    endif
  enddo
!$omp enddo
```

最適化イメージ

```
!パターン(1)
if((条件①真).and.(条件②真).and.(条件③真))then
  do i=1,n1
     処理①
   処理②
     処理③
   enddo
endif

!パターン(2)
if((条件①真).and.(条件②真).and.(条件③偽))then
  do i=1,n1
     処理①
   処理②
   enddo
endif

!パターン(3)
if((条件①真).and.(条件②偽).and.(条件③真))then
  do i=1,n1
   処理①
     処理③
  enddo
Endif

!パターン(4)
if((条件①真).and.(条件②偽).and.(条件③偽))then
  do i=1,n1
     処理①
  enddo
endif
```

```
!パターン(5)
if((条件①偽).and.(条件②真).and.(条件③真))then
  do i=1,n1
   処理②
     処理③
   enddo
endif

!パターン(6)
if((条件①偽).and.(条件②真).and.(条件③偽))then
  do i=1,n1
   処理②
   enddo
endif

!パターン(7)
if((条件①偽).and.(条件②偽).and.(条件③真))then
  do i=1,n1
     処理③
  enddo
endif

!パターン(8)
if((条件①偽).and.(条件②偽).and.(条件③偽))then
  do i=1,n1
  enddo
endif
```

8つのif文(do文)に展開される

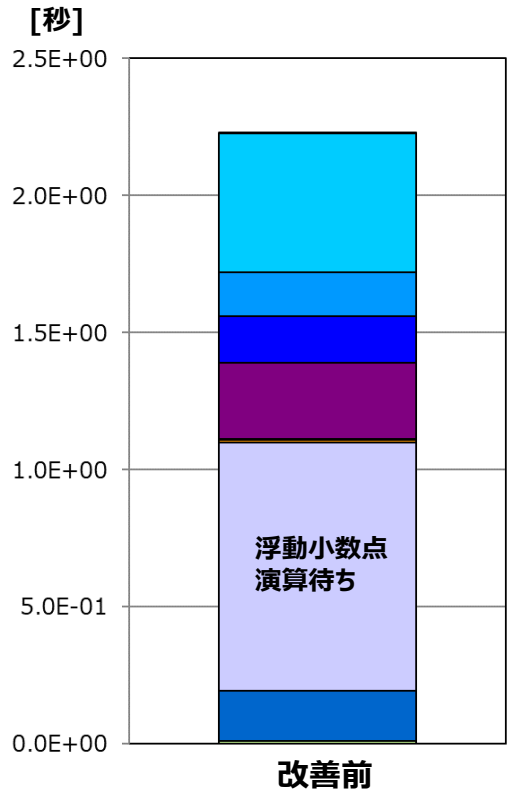
Fortran ループアンスイッチングの効果（改善前）

FUJITSU

最内ループにIF構文があり、SIMD化やソフトウェアパイプライニングが効果的に行われず、浮動小数点演算待ちが多くなっています。

改善前ソース

```
97    1           !$omp do
                  <<< Loop-information Start >>>
                  <<<  [OPTIMIZATION]
                  <<<    SIMD(VL: 8)
                  <<<    SOFTWARE PIPELINING(IPC: 1.31, ITR: 96,
                                                MVE: 2, POL: L)
                  <<<    UNSWITCHING
                  <<<    PREFETCH(HARD) Expected by compiler :
                  <<<      a, b
                  <<< Loop-information  End >>>
98    2   p   2v     do i=1,n1
99    2
100     3   p   2v       if (n1 >= q) then
101     3   p   2v         a(i) = c0+b(i)*(c1+b(i)*(c2+b(i)*(c3+b(i)*c4)))
102     3   p   2v       endif
103     2
104     3   p   2v       if(n1 > r) then
105     3   p   2v         a(i) = c0*b(i)/(c1*b(i)/(c2*b(i)/(c3*b(i)/c4)))
106     3   p   2v       endif
107     2
108     3   p   2v       if(n1 < s) then
109     3   p   2v         a(i) = c0+b(i)/(c1+b(i)/(c2+b(i)/(c3+b(i)/c4)))
110     3   p   2v       endif
111     2   p   2v    enddo
112     1           !$omp enddo nowait
```



SIMD

	Effective instruction	SIMD instruction rate (%) (/Effective instruction)
改善前	7.47E+10	94.18%

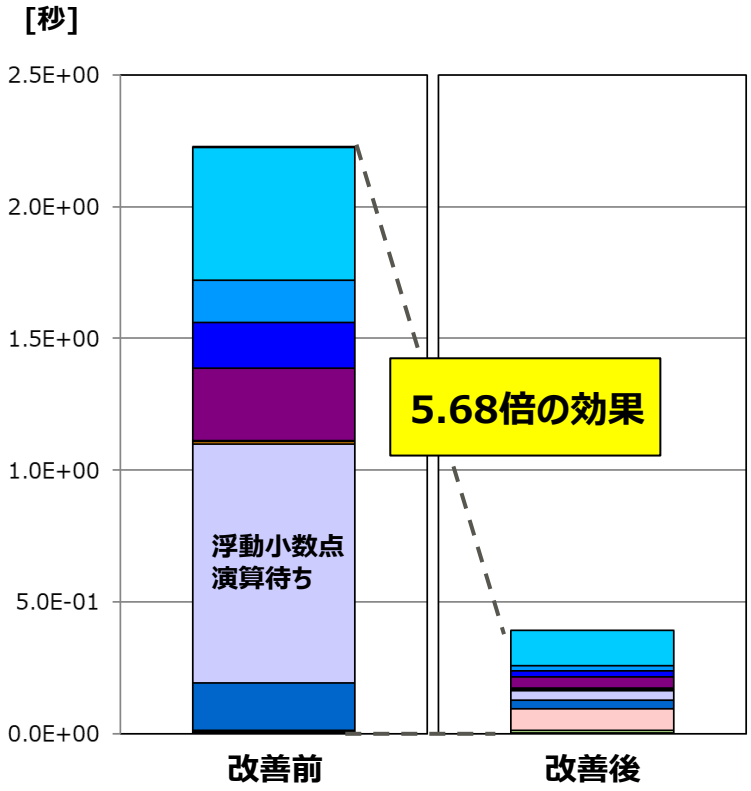
Fortran ループアンスイッチングの効果（改善後）

FUJITSU

IF構文に対してループアンスイッチングを指示することで分岐がなくなり、SIMD化およびソフトウェアパイプライニングが促進されました。その結果、浮動小数点演算待ちが大幅に改善されました。

改善後ソース（最適化制御行チューニング）

```
97    1          !$omp do
                 <<< Loop-information Start >>>
                 <<<  [OPTIMIZATION]
                 <<<    SIMD(VL: 8)
                 <<<    SOFTWARE PIPELINING(IPC: 1.04, ITR: 80,
                                                MVE: 3, POL: S)
                 <<<    UNSWITCHING
                 <<<    PREFETCH(HARD) Expected by compiler :
                 <<<      b, a
                 <<< Loop-information  End >>>
98    2  p  2v    do i=1,n1
99    2          !ocl unswitching
100     3  p  2v      if (n1 >= q) then
101     3  p  2v        a(i) = c0+b(i)*(c1+b(i)*(c2+b(i)*(c3+b(i)*c4)))
102     3  p  2v      endif
103     2          !ocl unswitching
104     3  p  2v      if(n1 > r) then
105     3  p  2v        a(i) = c0*b(i)/(c1*b(i)/(c2*b(i)/(c3*b(i)/c4)))
106     3  p  2v      endif
107     2          !ocl unswitching
108     3  p  2v      if(n1 < s) then
109     3  p  2v        a(i) = c0+b(i)/(c1+b(i)/(c2+b(i)/(c3+b(i)/c4)))
110     3  p  2v      endif
111     2  p  2v    enddo
112     1          !$omp enddo nowait
```



SIMD

	Effective instruction	SIMD instruction rate (%) (/Effective instruction)
改善前	7.47E+10	94.18%
改善後	1.60E+10	75.83%

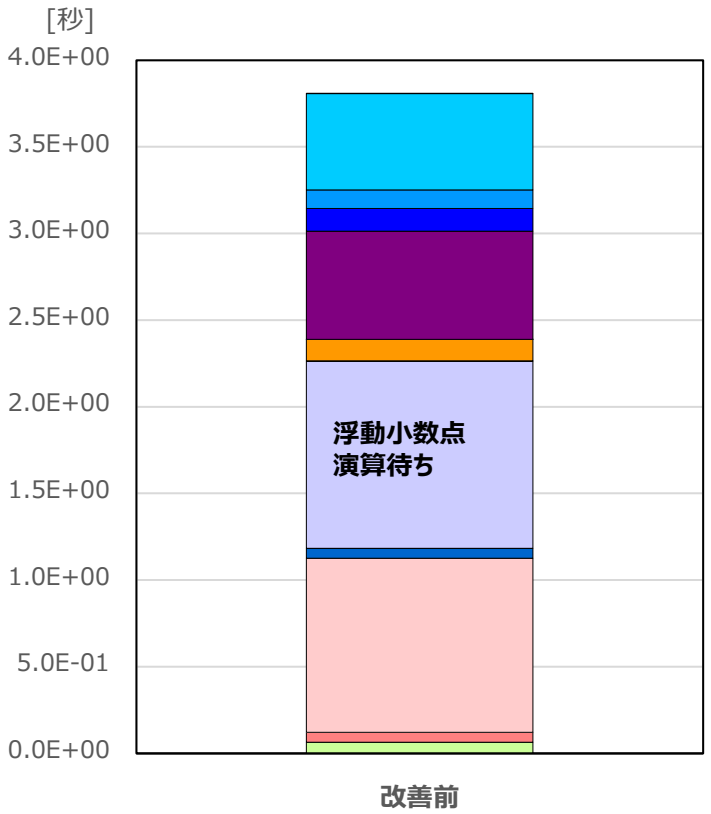
C/C++ ループアンスイッチングの効果（改善前）

FUJITSU

最内ループにif文があり、SIMD化やソフトウェアパイプライニングが効果的に行われず、浮動小数点演算待ちが多くなっています。

改善前ソース

```
91              #pragma omp for nowait
          <<< Loop-information Start >>>
          <<<  [OPTIMIZATION]
          <<<    SIMD(VL: 8)
          <<<    UNSWITCHING
          <<<    PREFETCH(HARD) Expected by compiler :
          <<<      (unknown)
          <<< Loop-information  End >>>
92   p    2v       for(i=0; i<n1; i++)
93   p    2v       {
94   p    2v         if (n1 >= q)
95   p    2v         {
96   p    2v           a[i] = c0+b[i]*(c0+b[i]*(c0+b[i]*(c0+b[i]*c0)));
97   p    2v         }
98
99   p    2v         if(n1 > r)
100  p    2v         {
101  p    2v           a[i] = c0*b[i]/(c0*b[i]/(c0*b[i]/(c0*b[i]/c0)));
102  p    2v         }
103
104  p    2v         if(n1 < s)
105  p    2v         {
106  p    2v           a[i] = c0+b[i]/(c0+b[i]/(c0+b[i]/(c0+b[i]/c0)));
107  p    2v         }
108  p    2v       }
109             }
```



Statistics	Effective instruction	SIMD instruction rate (%) (/Effective instruction)
改善前	8.54E+10	89.49%

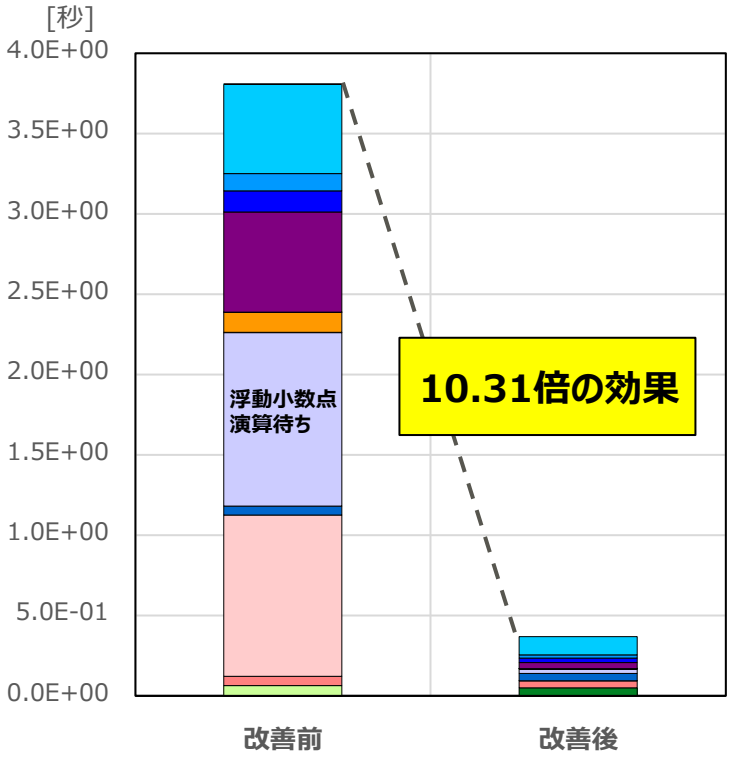
C/C++ ループアンスイッチングの効果（改善後）

FUJITSU

if文に対してループアンスイッチングを指示することで分岐がなくなり、SIMD化およびソフトウェアパイプライニングが促進されました。その結果、浮動小数点演算待ちが大幅に改善されました。

改善後ソース（最適化制御行チューニング）

```
89              for( j=0; j<iter; j++ )
90              {
91                #pragma omp for nowait
          <<< Loop-information Start >>>
          <<<  [OPTIMIZATION]
          <<<    SIMD(VL: 8)
          <<<    SOFTWARE PIPELINING(IPC: 1.12, ITR: 96, MVE: 3, POL: S)
          <<<    UNSWITCHING
          <<<    PREFETCH(HARD) Expected by compiler :
          <<<      (unknown)
          <<< Loop-information  End >>>
92   p    2v       for(i=0; i<n1; i++)
93   p    2v       {
94              #pragma statement unswitching
95   p    2v         if (n1 >= q)
96   p    2v         {
97   p    2v           a[i] = c0+b[i]*(c0+b[i]*(c0+b[i]*(c0+b[i]*c0)));
98   p    2v         }
99
100             #pragma statement unswitching
101  p    2v         if(n1 > r)
102  p    2v         {
103  p    2v           a[i] = c0*b[i]/(c0*b[i]/(c0*b[i]/(c0*b[i]/c0)));
104  p    2v         }
105
106             #pragma statement unswitching
107  p    2v         if(n1 < s)
108  p    2v         {
109  p    2v           a[i] = c0+b[i]/(c0+b[i]/(c0+b[i]/(c0+b[i]/c0)));
110  p    2v         }
111  p    2v       }
112             }
```



Statistics	Effective instruction	SIMD instruction rate (%) (/Effective instruction)
改善前	8.54E+10	89.49%
改善後	1.68E+10	71.58%

FUJITSU

マイクロアーキ依存の改善

- Scatter Store命令の回避
- Gather Load命令の纏め上げ機能
- 過剰SFIの改善
- Multiple Structures命令の活用
- ハードウェアプリフェッチの距離調整
- SVEのベクトルレジスタサイズ（SIMD幅）
- 半精度実数型の活用

FUJITSU

Scatter Store命令の回避

- Scatter Store命令の回避（改善前）
- Scatter Store命令の回避（ソースチューニング）

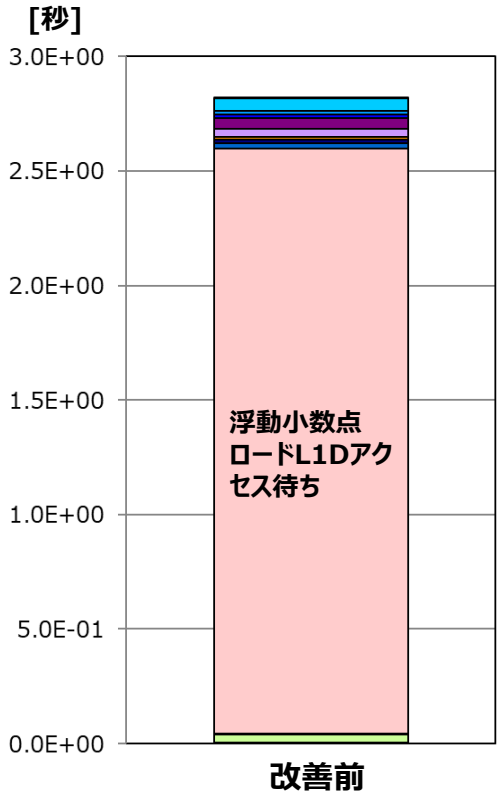
Fortran Scatter Store命令の回避（改善前）

FUJITSU

Scatter Store（連続でないストア）命令数が多い場合、性能が低下します。浮動小数点ロードL1Dアクセス待ちが多くなっています。

改善前ソース

```
42                !$omp parallel
43    1             DO K=1, ITER
44    1            !$omp do
                <<< Loop-information Start >>>
                <<<  [OPTIMIZATION]
                <<<    PREFETCH(HARD) Expected by compiler :
                <<<      b
                <<< Loop-information  End >>>
45    2  p           DO J=1,M
                <<< Loop-information Start >>>
                <<<  [OPTIMIZATION]
                <<<    SIMD(VL: 8)
                <<<    SOFTWARE PIPELINING(IPC: 2.60, ITR: 80,
                                                MVE: 3, POL: S)
                <<<    PREFETCH(HARD) Expected by compiler :
                <<<      b
                <<< Loop-information  End >>>
46    3  p  2v          DO I=1,M
47    3  p  2v            a(J,I) = b(I,J)
48    3  p  2v          ENDDO
49    2  p           ENDDO
50    1            !$omp end do nowait
51    1             ENDDO
52                !$omp end parallel
```



L1Dミスが高い

scatter store命令数が他の命令に比べて多い

Cache

	L1I miss rate (/Effective instruction)	Load-store instruction	L1D miss	L1D miss rate (/Load-store instruction)	L1D miss demand rate (%) (/L1D miss)	hardware prefetch rate (%) (/L1D miss)	software prefetch rate (%) (/L1D miss)
改善前	0.00	1.62E+09	1.50E+09	0.92	99.98%	0.02%	0.00%

SIMD

Store instruction				
SIMD				
contiguous store instruction	Multiple vector contiguous structure store instruction	Non-contiguous scatter store instruction	Floating-point register spill instruction	Predicate register spill instruction
1.60E+01	0.00E+00	1.30E+08	1.07E+04	5.49E+03

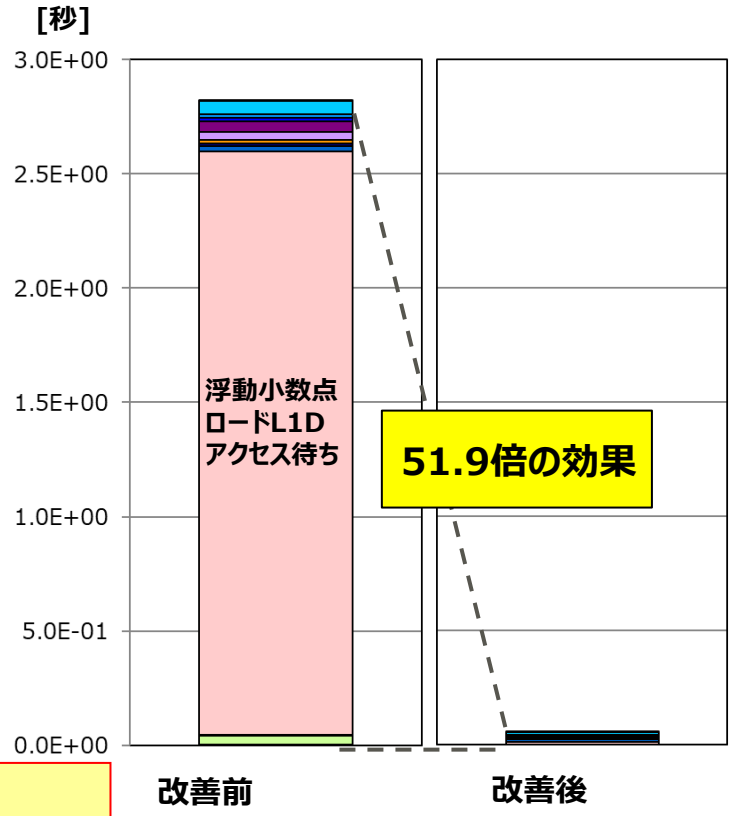
Fortran Scatter Store命令の回避（ソースチューニング）

FUJITSU

ループインデックスを変更することでScatter Store命令の発生を回避しました。
その結果、L1Dミスが大幅に改善されました。

改善後ソース

```
42                !$omp parallel
43    1             DO K=1, ITER
44    1            !$omp do
                <<< Loop-information Start >>>
                <<<  [OPTIMIZATION]
                <<<    PREFETCH(HARD) Expected by compiler :
                <<<      a
                <<< Loop-information  End >>>
45    2  p           DO J=1,M
                <<< Loop-information Start >>>
                <<<  [OPTIMIZATION]
                <<<    SIMD(VL: 8)
                <<<    SOFTWARE PIPELINING(IPC: 2.87, ITR: 256,
                                                 MVE: 4, POL: S)
                <<<    PREFETCH(HARD) Expected by compiler :
                <<<      a
                <<< Loop-information  End >>>
46    3  p  4v          DO I=1,M
47    3  p  4v            a(I,J) = b(J,I)
48    3  p  4v          ENDDO
49    2  p           ENDDO
50    1            !$omp end do nowait
51    1             ENDDO
52                !$omp end parallel
```



L1Dミスが大幅に改善された

scatter store命令数を低減できている

	L1I miss rate (/Effective instruction)	Load-store instruction	L1D miss	L1D miss rate (/Load-store instruction)	L1D hardware prefetch demand rate (%) (/L1D miss)	L1D software prefetch rate (%) (/L1D miss)	L1D hardware prefetch rate (%) (/L1D miss)
改善前	0.00	1.62E+09	1.50E+09	0.92	99.98%	0.02%	0.00%
改善後	0.00	3.64E+08	3.03E+06	0.01	99.97%	0.02%	0.01%

Store instruction				
SIMD				
Single vector contiguous store instruction	Multiple vector contiguous structure store instruction	Non-contiguous scatter store instruction	Floating-point register spill instruction	Predicate register spill instruction
1.60E+01	0.00E+00	1.30E+08	1.07E+04	5.49E+03
1.30E+08	0.00E+00	0.00E+00	6.40E+01	2.38E+02

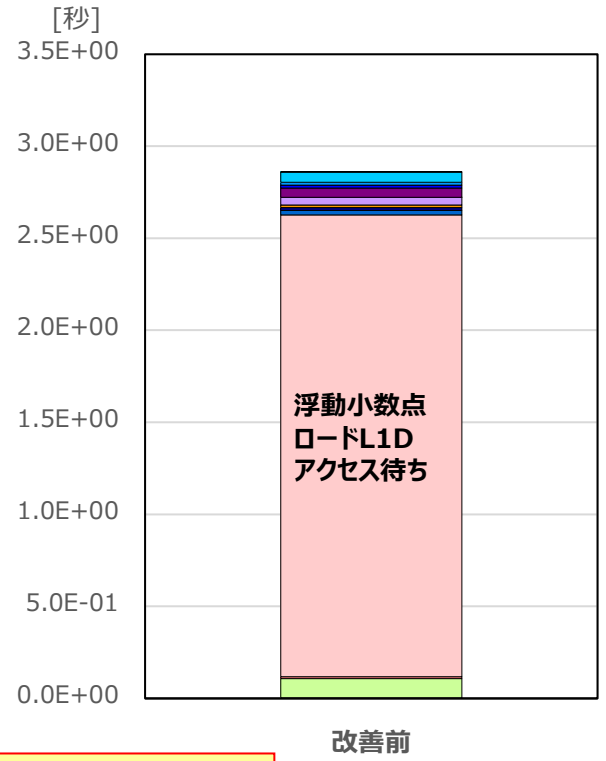
Scatter Store命令の回避（改善前）

FUJITSU

Scatter Store（連続でないストア）命令数が多い場合、性能が低下します。浮動小数点ロードL1Dアクセス待ちが多くなっています。

改善前ソース

```
44          #pragma omp parallel
45            {
46              for(k=0; k<iter; k++)
47              {
48            #pragma omp for nowait
            <<< Loop-information Start >>>
            <<<  [OPTIMIZATION]
            <<<    PREFETCH(HARD) Expected by compiler :
            <<<      b
            <<< Loop-information  End >>>
49   p            for(j=0; j<m; j++)
50   p            {
            <<< Loop-information Start >>>
            <<<  [OPTIMIZATION]
            <<<    SIMD(VL: 8)
            <<<    SOFTWARE PIPELINING(IPC: 2.40, ITR: 80,
                                MVE: 3, POL: S)
            <<<    PREFETCH(HARD) Expected by compiler :
            <<<      b
            <<< Loop-information  End >>>
51   p    2v          for(i=0;i<m;i++)
52   p    2v          {
53   p    2v            a[i][j] = b[j][i];
54   p    2v          }
55   p              }
56                }
57              }
```



L1Dミスが高い

scatter store命令数が他の命令に比べて多い

Cache	L1I miss rate (/Effective instruction)	Load-store instruction	L1D miss	L1D miss rate (/Load-store instruction)	L1D miss demand rate (%) (/L1D miss)	L1D miss hardware prefetch rate (%) (/L1D miss)	L1D miss software prefetch rate (%) (/L1D miss)
改善前	0.00	1.58E+09	1.50E+09	0.95	100.00%	0.00%	0.00%

Single vector contiguous store instruction	Multiple vector contiguous structure store instruction	Non-contiguous scatter store instruction	Floating-point register spill instruction	Predicate register spill instruction
0.00E+00	0.00E+00	1.30E+08	1.17E+04	5.88E+03

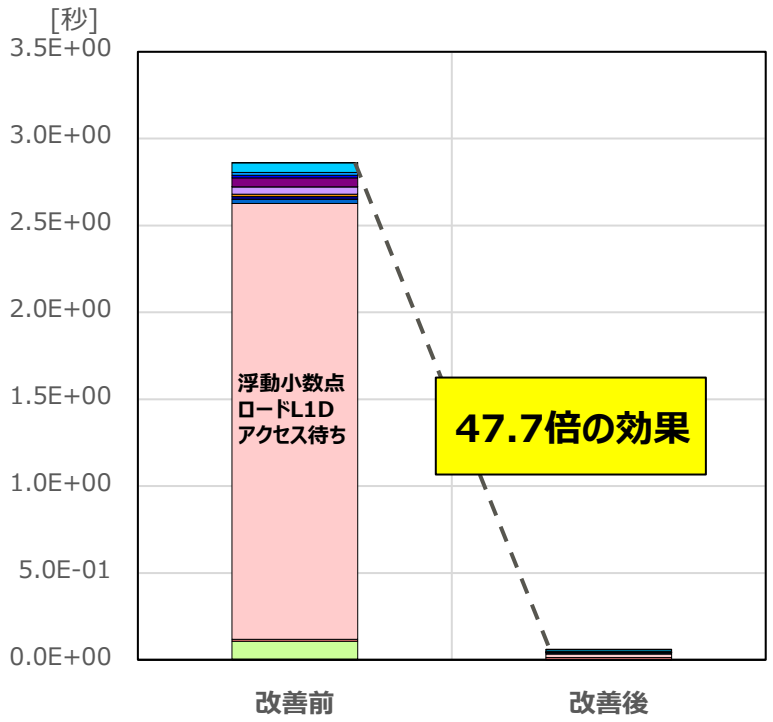
Scatter Store命令の回避（ソースチューニング）

FUJITSU

ループインデックスを変更することでScatter Store命令の発生を回避しました。
その結果、L1Dミスが大幅に改善されました。

改善後ソース

```
44          #pragma omp parallel
45           {
46             for(k=0; k<iter; k++)
47             {
48          #pragma omp for nowait
            <<< Loop-information Start >>>
            <<<  [OPTIMIZATION]
            <<<    PREFETCH(HARD) Expected by compiler :
            <<<      a
            <<< Loop-information  End >>>
49   p            for(j=0; j<m; j++)
50   p            {
            <<< Loop-information Start >>>
            <<<  [OPTIMIZATION]
            <<<    SIMD(VL: 8)
            <<<    SOFTWARE PIPELINING(IPC: 1.28, ITR: 96,
                                MVE: 2, POL: S)
            <<<    PREFETCH(HARD) Expected by compiler :
            <<<      a
            <<< Loop-information  End >>>
51   p    4v          for(i=0;i<m;i++)
52   p    4v          {
53   p    4v            a[j][i] = b[i][j];
54   p    4v          }
55   p              }
56                }
57              }
```



L1Dミスが大幅に改善された

scatter store命令数を
低減できている

Cache	L1I miss rate (/Effective instruction)	Load-store instruction	L1D miss	L1D miss rate (/Load-store instruction)	L1D miss demand rate (%) (/L1D miss)	L1D miss hardware prefetch rate (%) (/L1D miss)	L1D miss software prefetch rate (%) (/L1D miss)
改善前	0.00	1.58E+09	1.50E+09	0.95	100.00%	0.00%	0.00%
改善後	0.00	4.13E+08	3.67E+06	0.01	99.35%	0.67%	-0.01%

Single vector contiguous store instruction	Multiple vector contiguous structure store instruction	Non-contiguous scatter store instruction	Floating-point register spill instruction	Predicate register spill instruction
0.00E+00	0.00E+00	1.30E+08	1.17E+04	5.88E+03
1.30E+08	0.00E+00	0.00E+00	3.20E+01	1.70E+01

FUJITSU

Gather Load命令の纏め上げの促進

- Gather Load命令の纏め上げ機能について
- Gather Load命令の纏め上げの促進（改善前）
- Gather Load命令の纏め上げの促進（ソースチューニング）

Gather命令の纏め上げ機能について

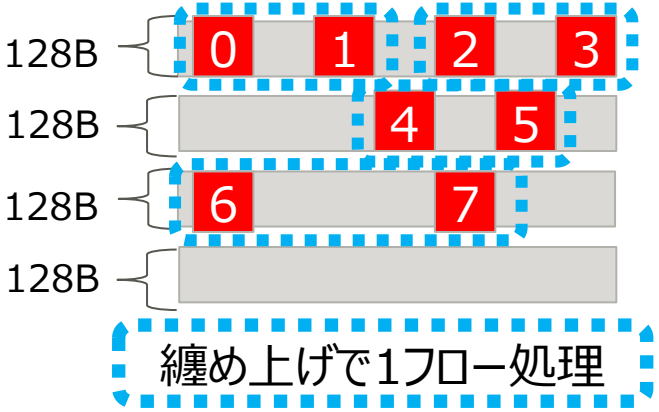
FUJITSU

- Gather命令の纏め上げ機能とは

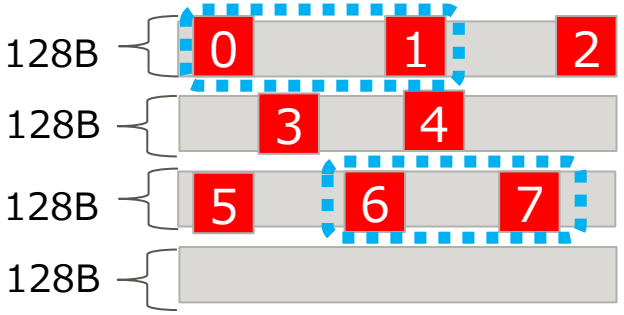
  Gather命令は、1つのFPから同時発行される2要素が隣接する場合に高速処理ができます。　隣接する2要素のアドレスが128バイト内で一致する場合には、まとめて1フローで処理することで高速化できます。

- 隣接2要素が、同一の128バイトブロックに属する場合、それらを纏めて1フローで処理します。以下のアドレスパターン例で　　の部分が纏め上げされて 、2要素をL1D$パイプライン1フローで処理します。

  - アドレスパターン例1



  - アドレスパターン例2



Gather命令の纏め上げ機能を活用するために配列の先頭アドレスには注意して実装して下さい。

Fortran Gather Load命令の纏め上げの促進（改善前）

FUJITSU

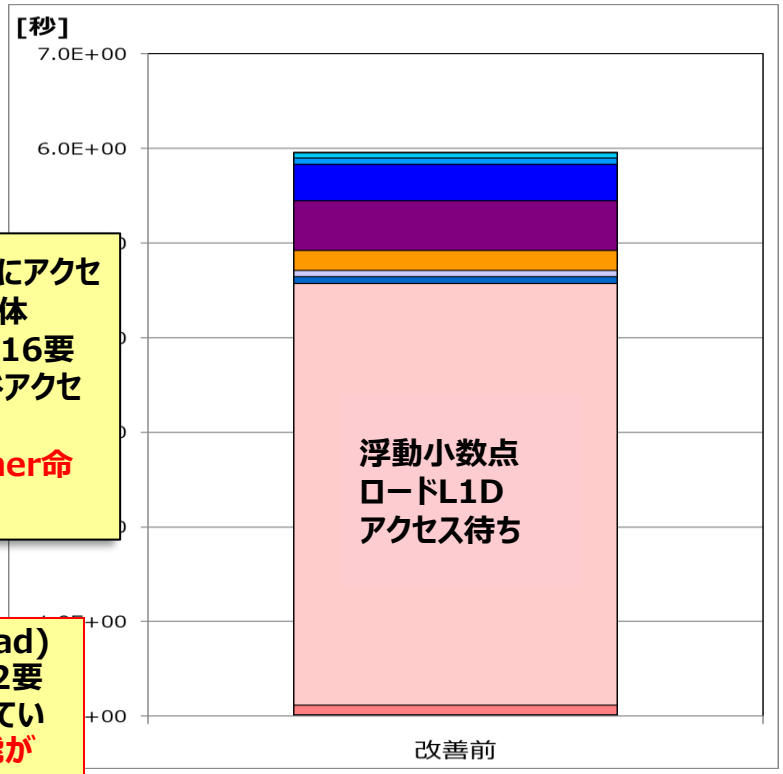
ストライドアクセスによりGather Load, Scatter Store命令が発生し、浮動小数点ロードL1Dアクセス待ちが多くなっています。

改善前ソース

```
30                real(kind=8),dimension(n,m) :: a
31                real(kind=8),dimension(n,m) :: b,c
32
                <<< Loop-information Start >>>
                <<<  [OPTIMIZATION]
                <<<    SIMD(VL: 8)
                <<<    SOFTWARE PIPELINING(IPC
                <<< Loop-information  End >>>
33    1     v       do i = 1, m
34    1     v         a( 1,i) = b( 1,i) + c( 1,i)
35    1     v         a( 2,i) = b( 2,i) + c( 2,i)
36    1     v         a( 3,i) = b( 3,i) + c( 3,i)
37    1     v         a( 4,i) = b( 4,i) + c( 4,i)
38    1     v         a( 5,i) = b( 5,i) + c( 5,i)
39    1     v         a( 6,i) = b( 6,i) + c( 6,i)
40    1     v         a( 7,i) = b( 7,i) + c( 7,i)
41    1     v         a( 8,i) = b( 8,i) + c( 8,i)
42    1     v         a( 9,i) = b( 9,i) + c( 9,i)
43    1     v         a(10,i) = b(10,i) + c(1
44    1     v         a(11,i) = b(11,i) + c(1
45    1     v         a(12,i) = b(12,i) + c(1
46    1     v         a(13,i) = b(13,i) + c(1
47    1     v         a(14,i) = b(14,i) + c(1
48    1     v         a(15,i) = b(15,i) + c(1
49    1     v         a(16,i) = b(16,i) + c(1
50    1     v       end do
```

n = 16

配列a, 配列b, 配列cは連続にアクセスされるが、それぞれの配列自体(a(1,i)など)は、1回転あたり16要素(128バイト)飛ぶストライドアクセスである
⇒ 離散アクセスのため、Gather命令が使用される



非連続ギャザーロード(Gather Load)命令が使用されているが、隣接する2要素のアドレスが128バイト以上離れているためGather命令の纏め上げ機能が動作せずL1ビジー率が高い

Instruction

	Load-store instruction	
	Load instruction	Store instruction
	Non-contiguous gather load instruction	Non-contiguous scatter store instruction
改善前	9.60E+08	4.80E+08

Busy	L1 busy rate (%)	L2 busy rate (%)	Memory busy rate (%)
改善前	76.29%	2.15%	0.00%

Extra	Gather instruction rate (%)		
	0 flow rate (%)	1 flow rate (%)	2 flow rate (%)
改善前	0.00%	0.00%	100.00%

Fortran Gather Load命令の纏め上げの促進（ソースチューニング）

FUJITSU

配列を分割して隣接する2要素のアドレスを128バイト内にすることでGather命令の纏め上げ機能が動作し、浮動小数点ロードL1Dアクセス待ちが改善されました。

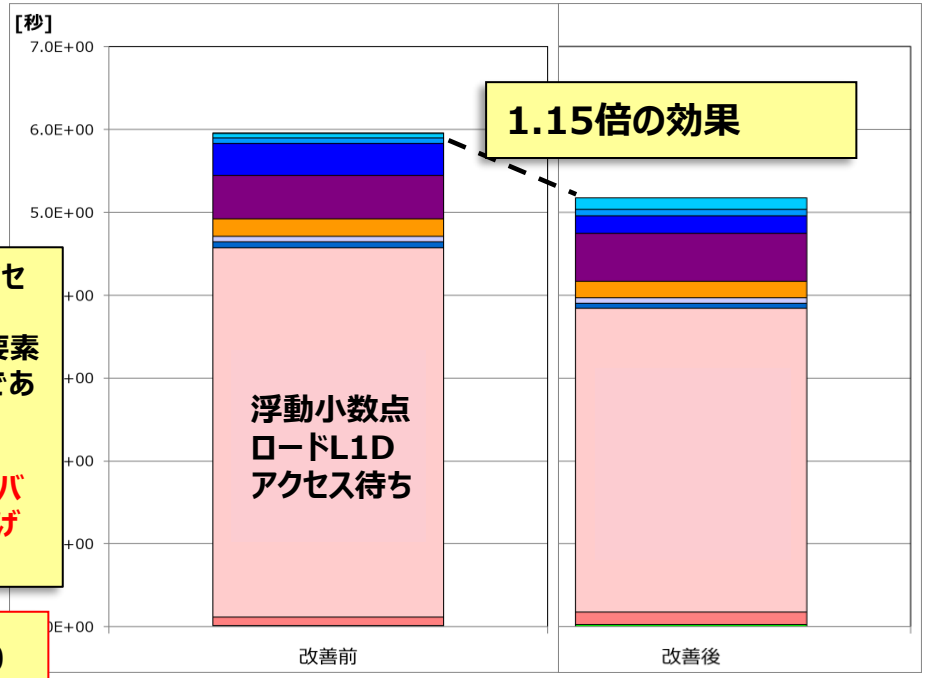
改善後ソース（ソースチューニング）

```
30              real(kind=8),dimension(n,m) :: a1,a2,a3,a4
31              real(kind=8),dimension(n,m) :: b1,b2,b3,b4,c1,c2,c3,c4
32
                <<< Loop-informatio
                <<<  [OPTIMIZATION]
                <<<    SIMD(VL: 8)
                <<<    SOFTWARE PIPELI
                <<< Loop-information  En
33    1     v       do i = 1, m
34    1     v         a1( 1,i) = b1( 1,i) +
35    1     v         a1( 2,i) = b1( 2,i) +
36    1     v         a1( 3,i) = b1( 3,i) +
37    1     v         a1( 4,i) = b1( 4,i) +
38    1     v         a2( 1,i) = b2( 1,i) +
39    1     v         a2( 2,i) = b2( 2,i) +
40    1     v         a2( 3,i) = b2( 3,i) +
41    1     v         a2( 4,i) = b2( 4,i) +
42    1     v         a3( 1,i) = b3( 1,i) + c3( 1,i)
43    1     v         a3( 2,i) = b3( 2,i
44    1     v         a3( 3,i) = b3( 3,i
45    1     v         a3( 4,i) = b3( 4,i
46    1     v         a4( 1,i) = b4( 1,i
47    1     v         a4( 2,i) = b4( 2,i
48    1     v         a4( 3,i) = b4( 3,i
49    1     v         a4( 4,i) = b4( 4,i) + c4( 4,i)
50    1     v       end do
```

n = 4

配列a, 配列b, 配列cは連続にアクセスされるが、それぞれの配列自体(a1(1,i)など)は、１回転あたり4要素(32バイト)飛ぶストライドアクセスである
⇒
隣接する２要素のアドレスが128バイト内となりGather命令の纏め上げ機能が動作する

Gather命令の纏め上げ機能により隣接2要素がL1D$パイプライン1フローで処理されるようになった。



Busy

	L1 busy rate (%)	L2 busy rate (%)	Memory busy rate (%)
改善前	76.29%	2.15%	0.00%
改善後	66.21%	2.84%	0.00%

Instruction

	Load-store instruction	
	Load instruction	Store instruction
	Non-contiguous gather load instruction	Non-contiguous scatter store instruction
改善前	9.60E+08	4.80E+08
改善後	9.60E+08	4.80E+08

Extra

	Gather instruction rate (%)		
	0 flow rate (%)	1 flow rate (%)	2 flow rate (%)
改善前	0.00%	0.00%	100.00%
改善後	0.00%	75.00%	25.00%

C/C++ Gather Load命令の纏め上げの促進（改善前）

FUJITSU

ストライドアクセスによりGather Load, Scatter Store命令が発生し、浮動小数点ロードL1Dアクセス待ちが多くなっています。

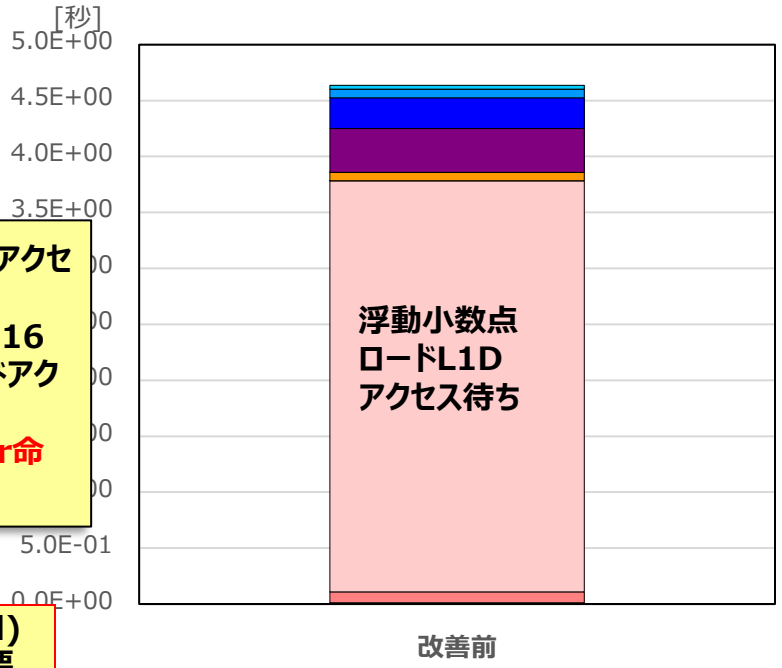
改善前ソース

```
50        void sub(int n, int m,double (* restrict a)[n],
            double (* restrict b)[n],double (* restrict c)[n])
51        {
52          int i;
53
          <<< Loop-information Start >>>
          <<<  [OPTIMIZATION]
          <<<    SIMD(VL: 8)
          <<<    SOFTWARE PIPELINING(IPC: 2
          <<< Loop-information  End >>>
54      v   for(i=0; i<m; i++)
55      v   {
56      v     a[i][ 0] = b[i][ 0] + c[i][ 0];
57      v     a[i][ 1] = b[i][ 1] + c[i][ 1];
58      v     a[i][ 2] = b[i][ 2] + c[i][ 2];
59      v     a[i][ 3] = b[i][ 3] + c[i][ 3];
60      v     a[i][ 4] = b[i][ 4] + c[i][ 4];
61      v     a[i][ 5] = b[i][ 5] + c[i][ 5];
62      v     a[i][ 6] = b[i][ 6] + c[i][ 6]
63      v     a[i][ 7] = b[i][ 7] + c[i][ 7]
64      v     a[i][ 8] = b[i][ 8] + c[i][ 8]
65      v     a[i][ 9] = b[i][ 9] + c[i][ 9]
66      v     a[i][10] = b[i][10] + c[i][1
67      v     a[i][11] = b[i][11] + c[i][1
68      v   }
70          return;
71        }
```

n = 16

配列a, 配列b, 配列cは連続にアクセスされるが、それぞれの配列自体(a[i][1] など)は、1回転あたり16要素(128バイト)飛ぶストライドアクセスである
⇒ 離散アクセスのため、Gather命令が使用される

非連続ギャザーロード(Gather Load)命令が使用されているが、隣接する2要素のアドレスが128バイト以上離れているためGather命令の纏め上げ機能が動作せずL1ビジー率が高い



Instruction	Load-store instruction	
	Load instruction	Store instruction
	Non-contiguous gather load instruction	Non-contiguous scatter store instruction
改善前	7.20E+08	3.60E+08

Busy	L1 busy rate (%)	L2 busy rate (%)	Memory busy rate (%)
改善前	77.75%	2.77%	0.00%

Extra	Gather instruction rate (%)		
	0 flow rate (%)	1 flow rate (%)	2 flow rate (%)
改善前	0.00%	0.00%	8.33%

C/C++ Gather Load命令の纏め上げの促進（ソースチューニング）

FUJITSU

配列を分割して隣接する2要素のアドレスを128バイト内にすることでGather命令の纏め上げ機能が動作し、浮動小数点ロードL1Dアクセス待ちが改善されました。

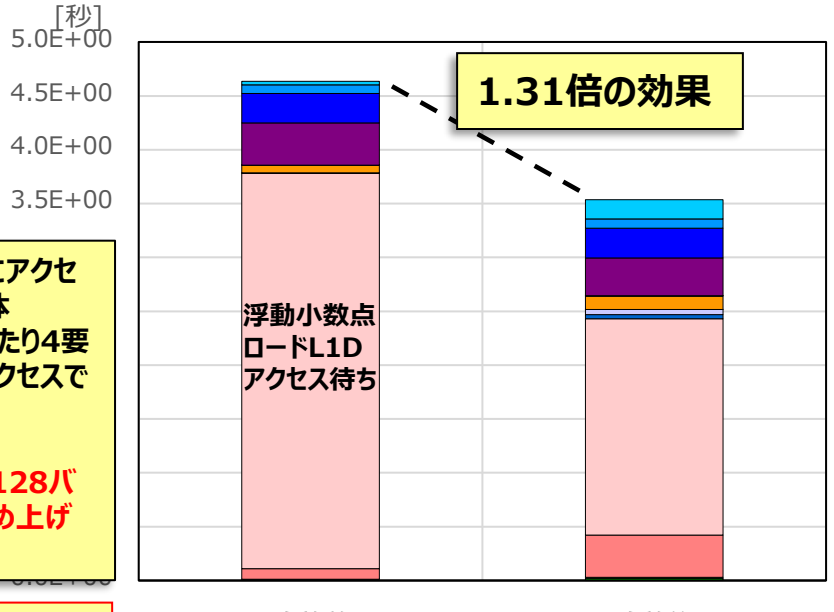
改善後ソース（ソースチューニング）

```
73        void sub(int n, int m,double (* restrict a1)[n],double (* restrict a2)[n],
74                 double (* restrict a3)[n],
75                 double (* restrict b1)[n],double (* restrict b2)[n],
76                 double (* restrict b3)[n],
77                 double (* restrict c1)[n],double (* restrict c2)[n],
78                 double (* restrict c3)[n])
79        {
80          int i;
81
          <<< Loop-information Start >>>
          <<<  [OPTIMIZATION]
          <<<    SIMD(VL: 8)
          <<<    SOFTWARE PIPELINING(IPC: 2.61,
          <<< Loop-information  End >>>
82        v   for(i=0; i<m; i++)
83        v   {
84        v     a1[i][ 0] = b1[i][ 0] + c1[i][ 0];
85        v     a1[i][ 1] = b1[i][ 1] + c1[i][ 1];
86        v     a1[i][ 2] = b1[i][ 2] + c1[i][ 2];
87        v     a1[i][ 3] = b1[i][ 3] + c1[i][ 3];
88        v     a2[i][ 0] = b2[i][ 0] + c2[i][ 0];
89        v     a2[i][ 1] = b2[i][ 1] + c2[i][ 1];
90        v     a2[i][ 2] = b2[i][ 2] + c2[i][ 2];
91        v     a2[i][ 3] = b2[i][ 3] + c2[i][ 3];
92        v     a3[i][ 0] = b3[i][ 0] + c3[i][ 0];
93        v     a3[i][ 1] = b3[i][ 1] + c3[i][ 1];
94        v     a3[i][ 2] = b3[i][ 2] + c3[i][ 2];
95        v     a3[i][ 3] = b3[i][ 3] + c3[i][ 3];
96        v   }
98          return;
99        }
```

n = 4

配列a, 配列b, 配列cは連続にアクセスされるが、それぞれの配列自体(a1[i][1] など)は、１回転あたり4要素(32バイト)飛ぶストライドアクセスである
⇒
隣接する２要素のアドレスが128バイト内となりGather命令の纏め上げ機能が動作する

Gather命令の纏め上げ機能により隣接2要素がL1D$パイプライン1フローで処理されるようになった。



Instruction	Load-store instruction	
	Load instruction	Store instruction
	Non-contiguous gather load instruction	Non-contiguous scatter store instruction
改善前	7.20E+08	3.60E+08
改善後	7.20E+08	3.60E+08

Busy	L1 busy rate (%)	L2 busy rate (%)	Memory busy rate (%)
改善前	77.75%	2.77%	0.00%
改善後	59.49%	1.07%	0.00%

Extra	Gather instruction rate (%)		
	0 flow rate (%)	1 flow rate (%)	2 flow rate (%)
改善前	0.00%	0.00%	8.33%
改善後	0.00%	100.00%	0.00%

FUJITSU

過剰SFIの回避

- 過剰SFIとは
- 過剰SFIが発生するケース1
- 過剰SFIが発生するケース2
- 過剰SFIの回避（改善前）
- 過剰SFIの回避（ソースチューニング）

過剰SFIとは

FUJITSU

- SFI(Store Fetch Interlock)とは
  - 先行するstore命令と後続のload命令のアドレスが異なっている場合、loadがstoreを追い越さないようにインターロックをかける制御機構です。
  - 基本的にはstoreが行われるアドレスにのみロックがかかります。

- 過剰SFIについて

  過剰SFIは上述の制御により過剰にロックがかかる現象です。以下のケースで発生します。
  - マスク付きSIMDのケースで、マスク判定が0(storeしない)のアドレスがロックされる。
  - GatherLoadの纏め上げ機能が動作した場合、GatherLoadでのSFI確認対象がキャッシュラインに含まれる全要素となり、実際にloadを行わないアドレスのSFIを検出しロックされていると判断する場合がある。

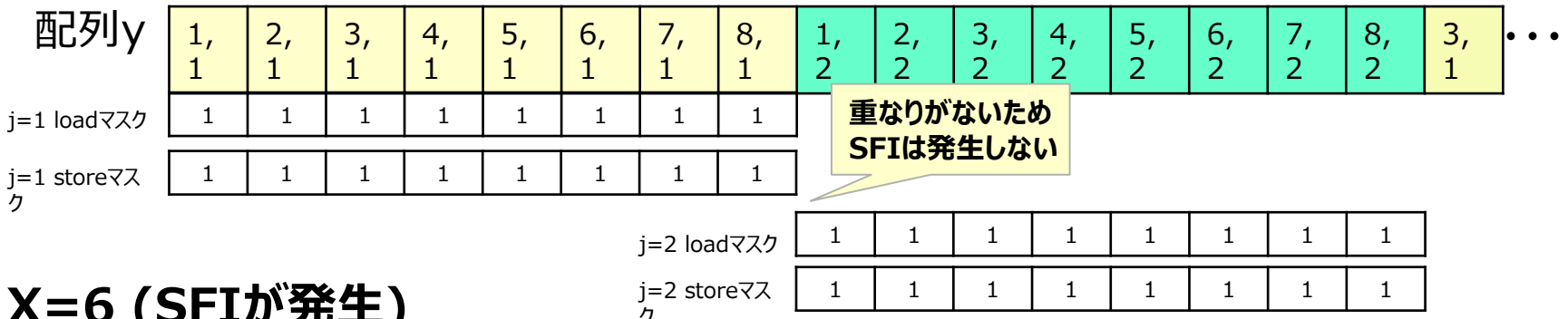
過剰SFIが発生するケース1 (1/2)

右の例でX=8のときにはSFIは発生しませんが、
X=6のときには、マスク値が0のアドレスがロックされ
過剰SFIが発生します。

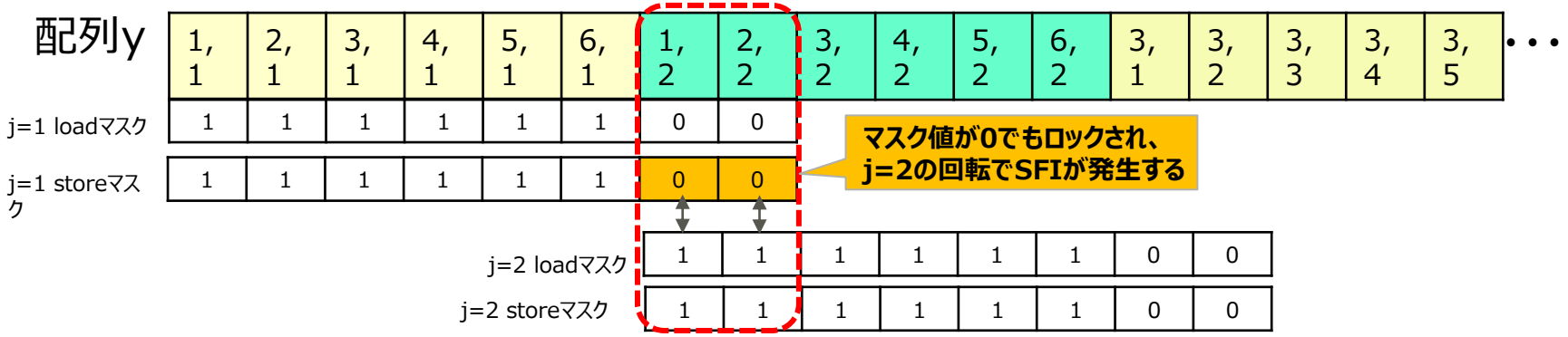
ソースコード

```
real*8 y(X,n), x1(X,n)    <- X=8または6
  Do k = 1, iter
    Do j = 1, n
      Do i = 1, X         <- X=8または6
        y(i,j) = y(i,j) + x1(i,j)
      End Do
    End Do
  End Do
```

X=8 (SFIは発生しない)



X=6 (SFIが発生)

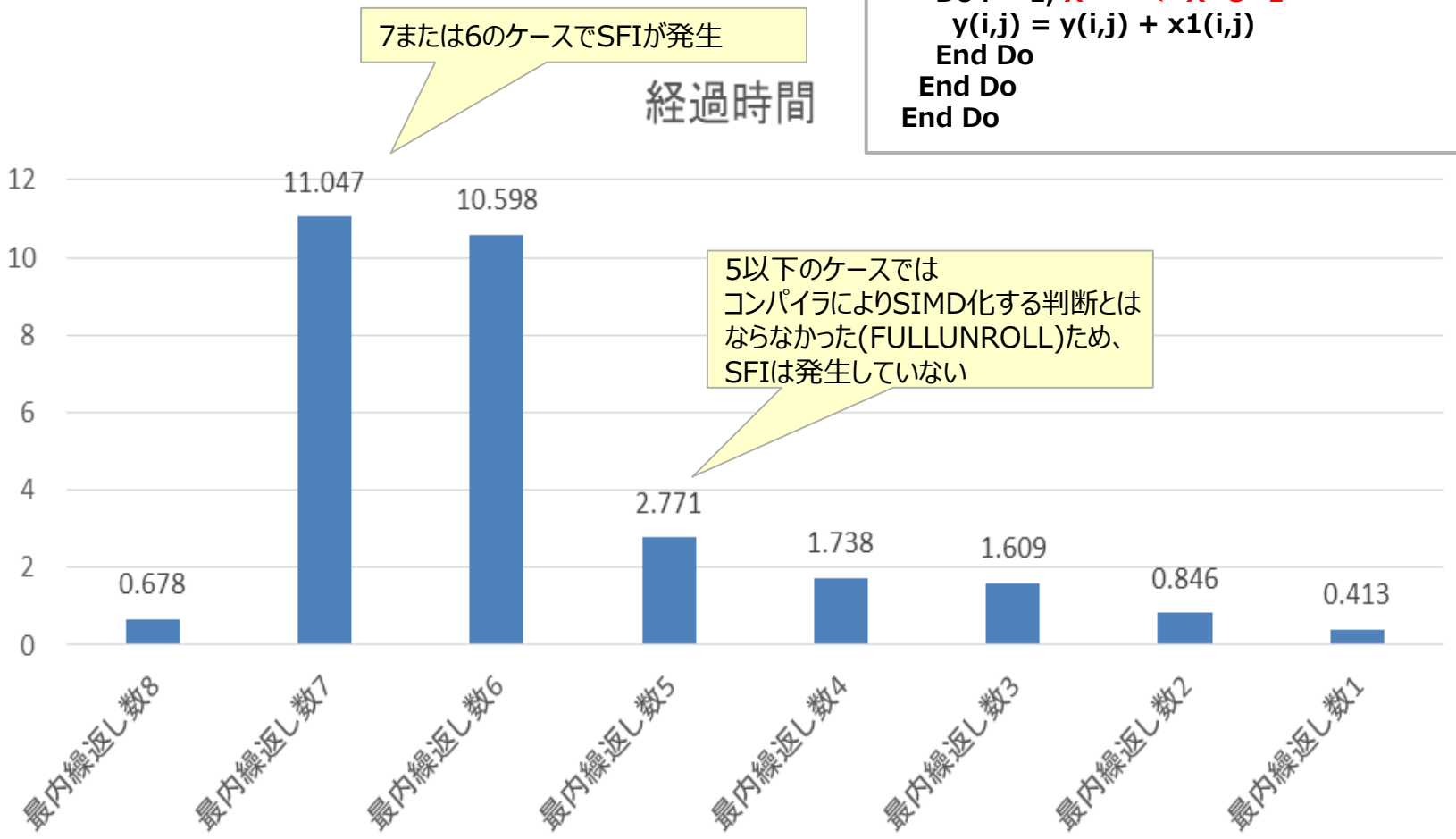


過剰SFIが発生するケース1 (2/2)

右の例で、X=7または6のケースではSFIが発生しています。X=5以下のケースではSIMD化されなかったため、SFIは発生していません。

ソースコード

```
real*8 y(X,n), x1(X,n)    <- X=8~1
 Do k = 1, iter
  Do j = 1, n
   Do i = 1, X       <- X=8~1
    y(i,j) = y(i,j) + x1(i,j)
   End Do
  End Do
 End Do
```



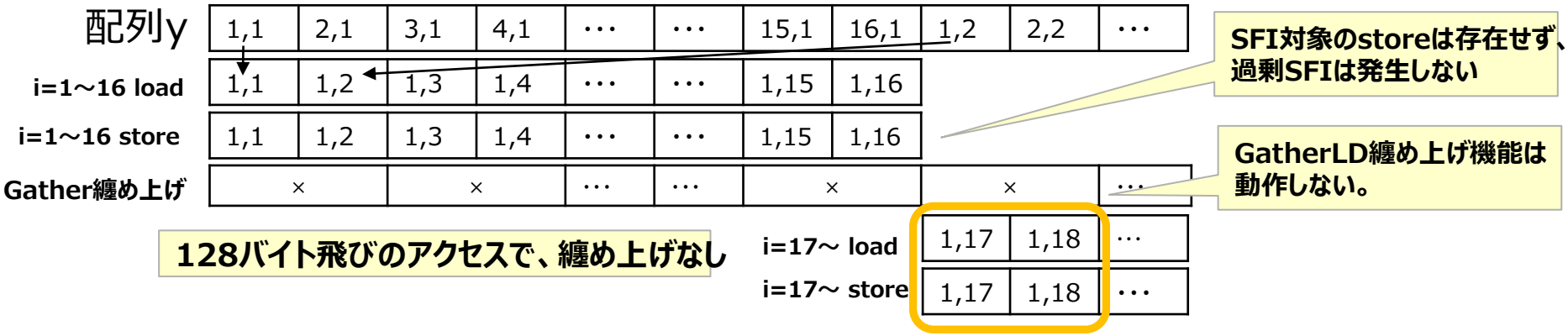
過剰SFIが発生するケース2

右の例で、X=1のときにはSFIは発生しませんが、
X=16のときには、SFI対象のstoreが存在するため
過剰SFIが発生します。

ソースコード

```
integer n  <- 1または16
real*8 y(n, m), x1(n, m)
  Do k = 1, iter
    Do i = 1, m
       y(1, i) = y(1, i) + x1(1, i)
    End Do
  End Do
```

X=1 (SFIは発生しない)

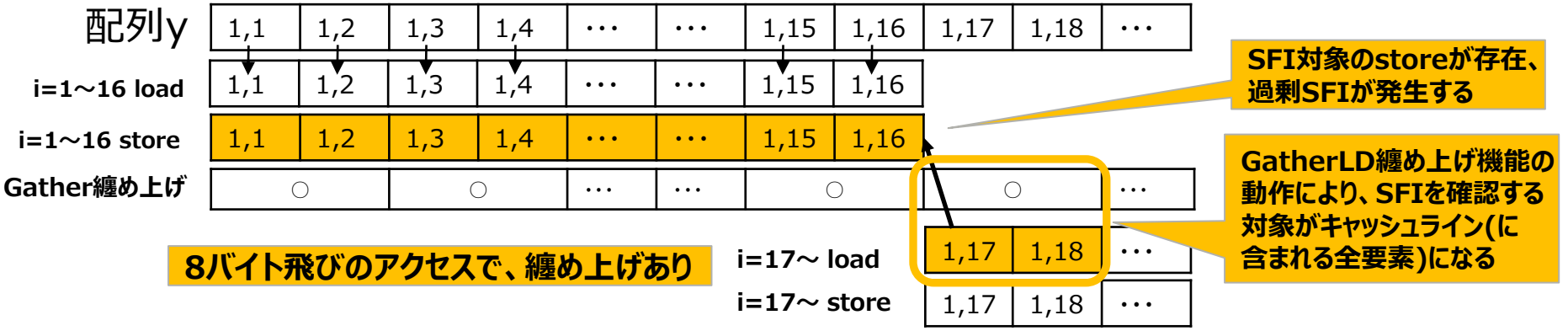


SFI対象のstoreは存在せず、過剰SFIは発生しない

GatherLD纏め上げ機能は動作しない。

128バイト飛びのアクセスで、纏め上げなし

X=16 (SFIが発生)



SFI対象のstoreが存在、過剰SFIが発生する

GatherLD纏め上げ機能の動作により、SFIを確認する対象がキャッシュライン(に含まれる全要素)になる

8バイト飛びのアクセスで、纏め上げあり

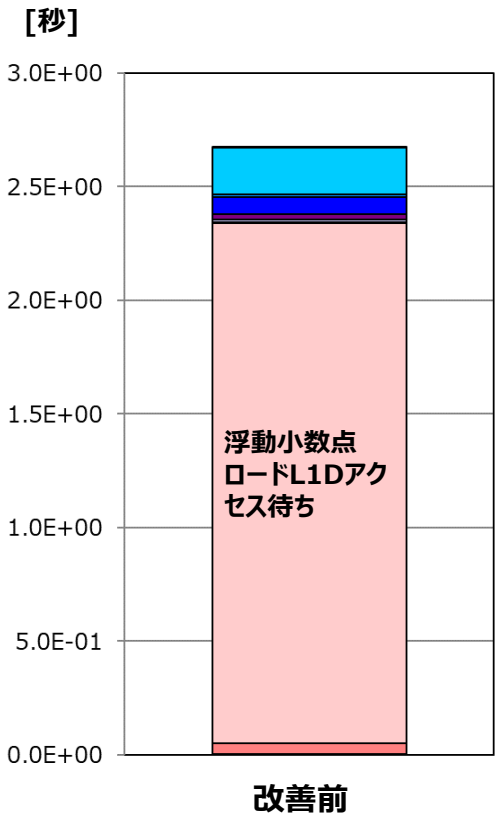
Fortran 過剰SFIの回避（改善前）

FUJITSU

最内ループがケース1の状態になっており、マスク値が0のアドレスがロックされ過剰SFIが発生しています。ポイント：ループ繰り返し回数が少なく、SWPLされていない場合

改善前ソース

```
39                  real(4) :: a(20,M), b(20,M)
40                  integer(4) :: M, ITER
41                  real(4),parameter :: c=0.5
 :
                 <<< Loop-information Start >>>
                 <<<  [OPTIMIZATION]
                 <<<    SOFTWARE PIPELINING(IPC: 3.25, ITR: 304,
                                                MVE: 7, POL: S)
                 <<<    PREFETCH(HARD) Expected by compiler :
                 <<<      a, b
                 <<< Loop-information  End >>>
46    2  p              DO J=1,M
                 <<< Loop-information Start >>>
                 <<<  [OPTIMIZATION]
                 <<<    SIMD(VL: 16)
                 <<< Loop-information  End >>>
47    3  p   v            DO I=1,20
48    3  p   v              a(I,J) = a(I,J) + c * b(I,J)
49    3  p   v            ENDDO
50    2  p              ENDDO
```



Busy

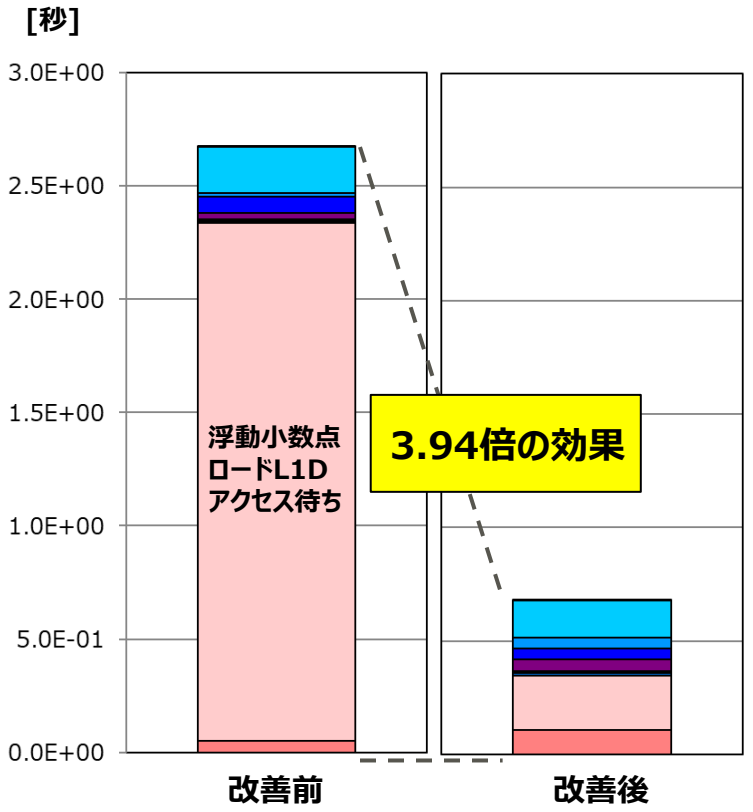
	SFI(Store Fetch Interlock) rate
改善前	0.44

Fortran 過剰SFIの回避（ソースチューニング）

パディングにより配列要素数を変更し、SIMD長の倍数とすることで、過剰SFIを回避することができました。

改善後ソース

```
39                  real(4) :: a(32,M), b(32,M)
40                  integer(4) :: M, ITER
41                  real(4),parameter :: c=0.5
 :
                 <<< Loop-information Start >>>
                 <<<  [OPTIMIZATION]
                 <<<    SOFTWARE PIPELINING(IPC: 3.25, ITR: 304,
                                                MVE: 7, POL: S)
                 <<<    PREFETCH(HARD) Expected by compiler :
                 <<<      a, b
                 <<< Loop-information  End >>>
46    2   p             DO J=1,M
                 <<< Loop-information Start >>>
                 <<<  [OPTIMIZATION]
                 <<<    SIMD(VL: 16)
                 <<< Loop-information  End >>>
47    3   p   v           DO I=1,20
48    3   p   v             a(I,J) = a(I,J) + c * b(I,J)
49    3   p   v           ENDDO
50    2   p             ENDDO
```



Busy

	SFI(Store Fetch Interlock) rate
改善前	0.43
改善後	0.01

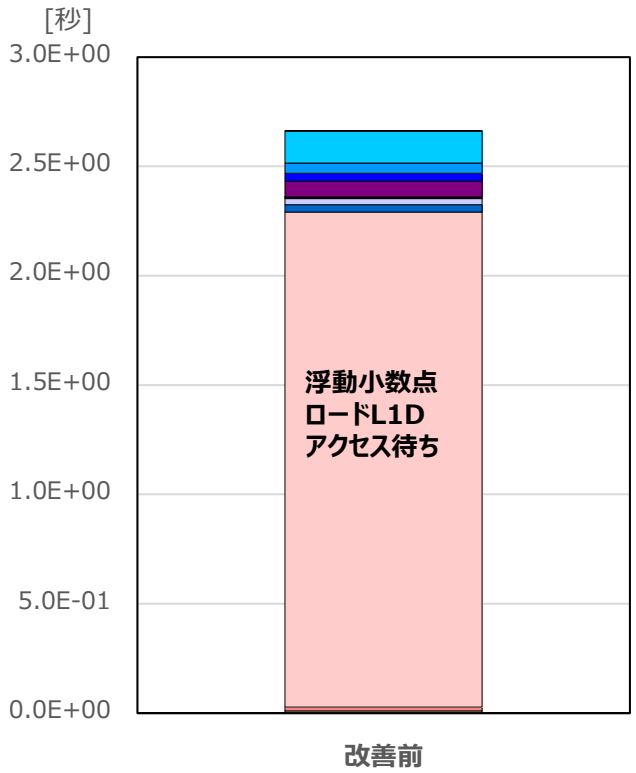
C/C++ 過剰SFIの回避（改善前）

FUJITSU

最内ループがケース1の状態になっており、マスク値が0のアドレスがロックされ過剰SFIが発生しています。ポイント：ループ繰り返し回数が少なく、SWPLされていない場合

改善前ソース

```
42 void sfil1(float (* restrict a)[20], float (* restrict b)[20],
                    int m, int iter)
43         {
44           float  c=0.5;
45           int i,j,k;
46
47          #pragma omp parallel
48          {
           <<< Loop-information Start >>>
:
           <<< Loop-information  End >>>
49           for(k=0; k<iter; k++)
50           {
51          #pragma omp for nowait
           <<< Loop-information Start >>>
 :
           <<< Loop-information  End >>>
52  p          for(j=0; j<m; j++)
53  p          {
           <<< Loop-information Start >>>
 :
           <<< Loop-information  End >>>
54  p    v       for(i=0; i< 20; i++)
55  p    v       {
56  p    v         a[j][i] = a[j][i] + c * b[j][i];
57  p    v       }
58  p          }
59           }
60          }
62           return;
63          }
```



Busy	SFI(Store Fetch Interlock) rate
改善前	0.44%

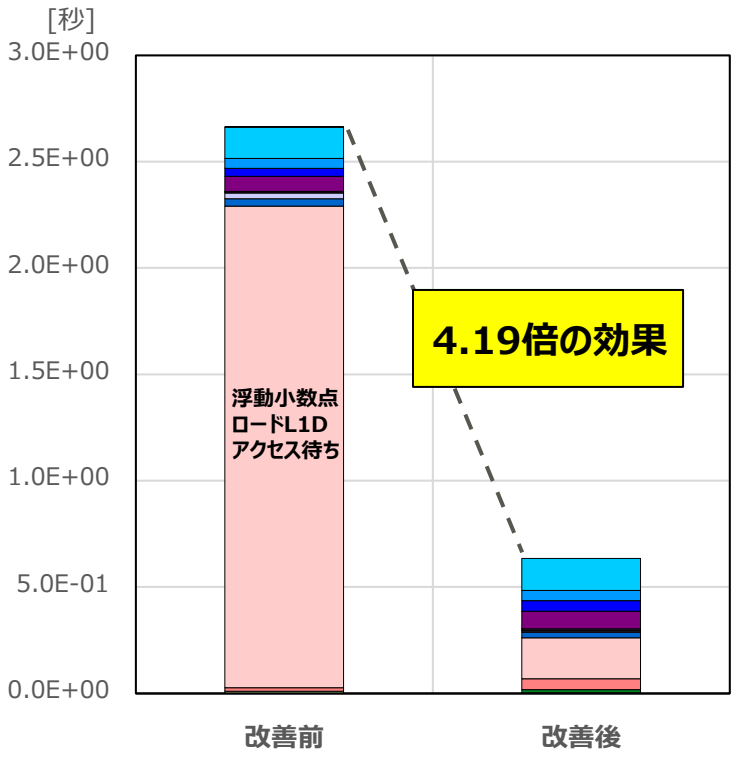
C/C++ 過剰SFIの回避（ソースチューニング）

FUJITSU

パディングにより配列要素数を変更し、SIMD長の倍数とすることで、過剰SFIを回避することができました。

改善後ソース

```
42 void sfil1(float (* restrict a)[32], float (* restrict b)[32],
                                int m, int iter)
43          {
44            float  c=0.5;
45            int i,j,k;
46
47          #pragma omp parallel
48          {
           <<< Loop-information Start >>>
 :
           <<< Loop-information  End >>>
49            for(k=0; k<iter; k++)
50            {
51          #pragma omp for nowait
           <<< Loop-information Start >>>
 :
           <<< Loop-information  End >>>
52  p           for(j=0; j<m; j++)
53  p           {
           <<< Loop-information Start >>>
 :
           <<< Loop-information  End >>>
54  p   v         for(i=0; i< 20; i++)
55  p   v         {
56  p   v           a[j][i] = a[j][i] + c * b[j][i];
57  p   v         }
58  p           }
59            }
60          }
62            return;
63          }
```



Busy	SFI(Store Fetch Interlock) rate
改善前	0.44%
改善後	0.01%

FUJITSU

Multiple Structures命令の活用

- Multiple Structures命令の適用条件
- Multiple Structures命令の活用（改善前）
- Multiple Structures命令の活用（ソースチューニング）

Multiple Structures命令の適用条件

FUJITSU

- **Fortran、C/C++(Tradモード/Clangモード)でサポート**
- **Fotran、C/C++(Tradモード)はオプションによりON/OFFが可能です。**

```
-Ksimd_use_multiple_structures |
-Ksimd_nouse_multiple_structures
```

デフォルトは-Ksimd_use_multiple_structures、
-Ksimd_nouse_multiple_structuresで抑止が可能

- **構造体配列(AoS)の場合、最内次元が2,3,4個の要素で形成されている、かつ、その要素すべてにアクセスしている場合、適用されます。最内次元が5個以上の要素の場合、適用されません。**
  - **適用可能例(Fortran)：**

```
real*8 y(n),x (4,n)

Do j = 1, iter
  Do i = 1, n
    y(i) = x(1,i) + x(2,i) + x(3,i) + x(4,i)
  End Do
End Do
```

  - **適用可能例(C/C++)：**

```
double y[n], x[N][4];

for (j = 0; j < iter; j++) {
  for (i = 0; i < N; i++) {
    y(i) = x[i][0] + x[i][1] + x[i][2] + x[i][3];
  }
}
```

- **なお、構造体配列(AoS)を配列構造体(SoA)に書き換えることで、連続ロードとなり性能向上が見込める場合は、書き換えをお勧めします。**

Fortran Multiple Structures命令の活用（改善前）

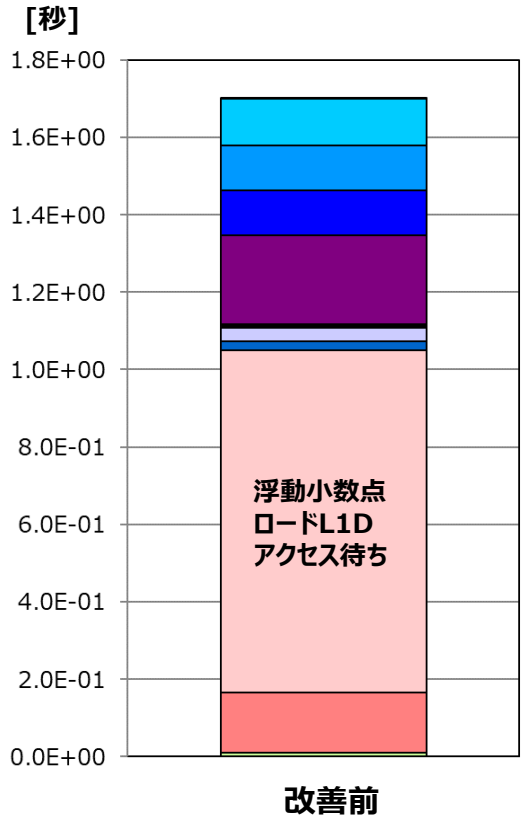
FUJITSU

構造体配列の配列が6個の要素でGatherロード命令が使用（Multiple Structures命令が適用されない）されているため、浮動小数点ロードL1Dアクセス待ちが多くなっています。

改善前ソース

```
                <<< Loop-information Start >>>
                <<<  [OPTIMIZATION]
                <<<    SIMD(VL: 8)
                <<<    SOFTWARE PIPELINING(IPC: 2.35, ITR: 96,
                                                MVE: 3, POL: S)
                <<<    PREFETCH(HARD) Expected by compiler :
                <<<      a, b
                <<< Loop-information  End >>>
22     2   p   2v          do i=1,N
23     2   p   2v            b(i)=a(1,i) + a(2,i) + a(3,i) + &
                                a(4,i) + a(5,i) + a(6,i)
24     2   p   2v          enddo
```

配列aが６個の要素であるため、Multiple Structures命令が適用されない。



Fortran Multiple Structures命令の活用（ソースチューニング）

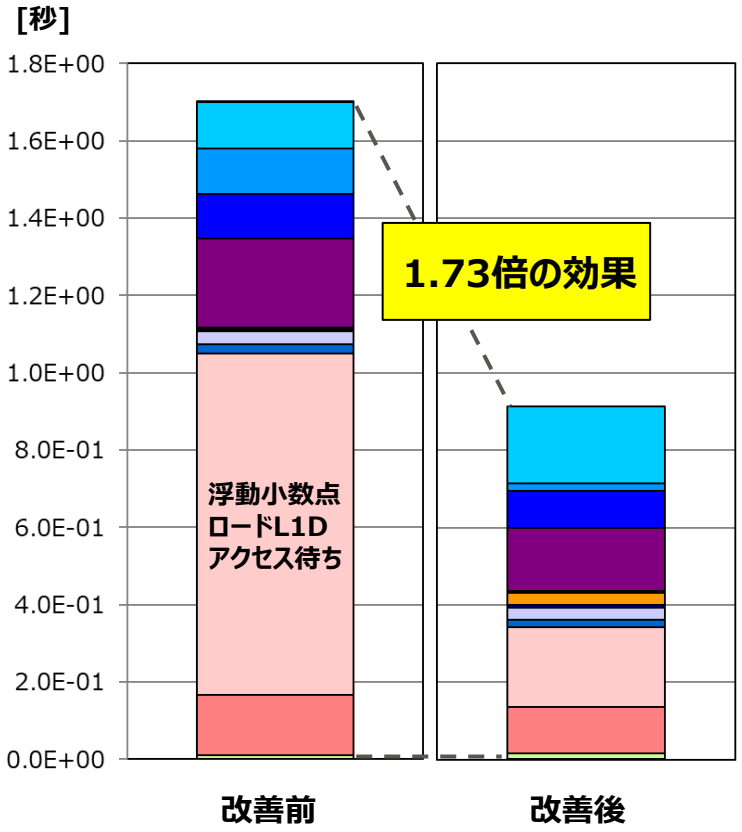
FUJITSU

配列を3つずつの要素に分けることでMultiple Structures命令が適用され、浮動小数点ロードL1Dアクセス待ちが削減されました。

改善後ソース（ソースチューニング）

```
                <<< Loop-information Start >>>
                <<<  [OPTIMIZATION]
                <<<    SIMD(VL: 8)
                <<<    SOFTWARE PIPELINING(IPC: 1.33, ITR: 56,
                                                MVE: 2, POL: S)
                <<<    PREFETCH(HARD) Expected by compiler :
                <<<      c, a, b
                <<< Loop-information  End >>>
22    2  p   v          do i=1,N
23    2  p   v            b(i) = a(1,i) + a(2,i) + a(3,i) + &
                                  c(1,i) + c(2,i) + c(3,i)
24    2  p   v          enddo
```

Multiple Structures命令適用
配列aが６個の要素であるため、
配列cを追加して3個ずつの要素にした。



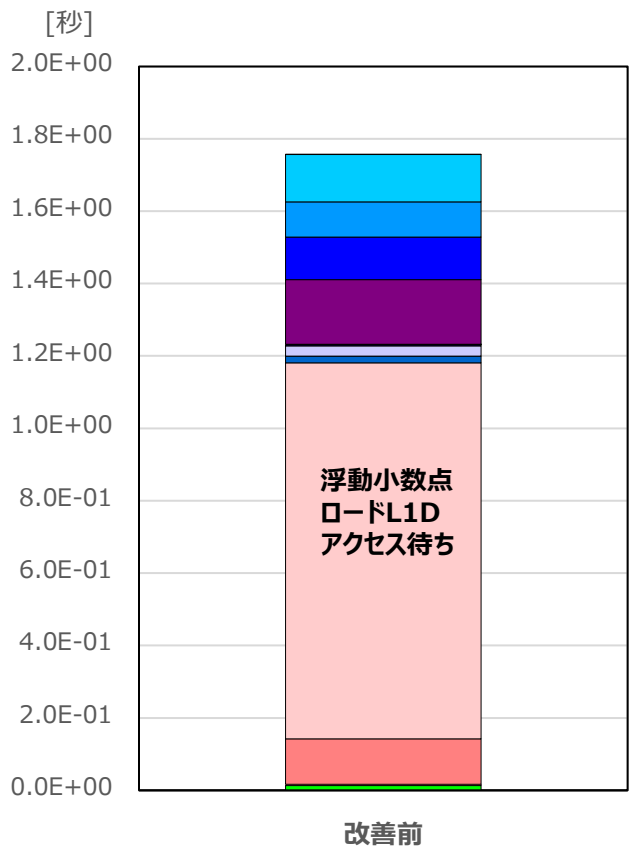
C/C++ Multiple Structures命令の活用（改善前）

FUJITSU

構造体配列の配列が6個の要素でGatherロード命令が使用（Multiple Structures命令が適用されない）されているため、浮動小数点ロードL1Dアクセス待ちが多くなっています。

改善前ソース

```
33          void MultiStructure(int n, int m, int iter)
34          {
35            int i,k;
36
37            #pragma omp parallel
38            {
            <<< Loop-information Start >>>
 :
            <<< Loop-information  End >>>
39              for(k=0; k< iter; k++)
40              {
41           #pragma omp for nowait
            <<< Loop-information Start >>>
 :
            <<< Loop-information  End >>>
42  p   2v        for(i=0; i< n; i++)
43  p   2v        {
44  p   2v          b[i]=a[i][0] + a[i][1] + a[i][2]
                         + a[i][3] + a[i][4] + a[i][5];
45  p   2v        }
46              }
47            }
48
49            return;
50          }
```



配列aが６個の要素であるため、Multiple Structures命令が適用されない。

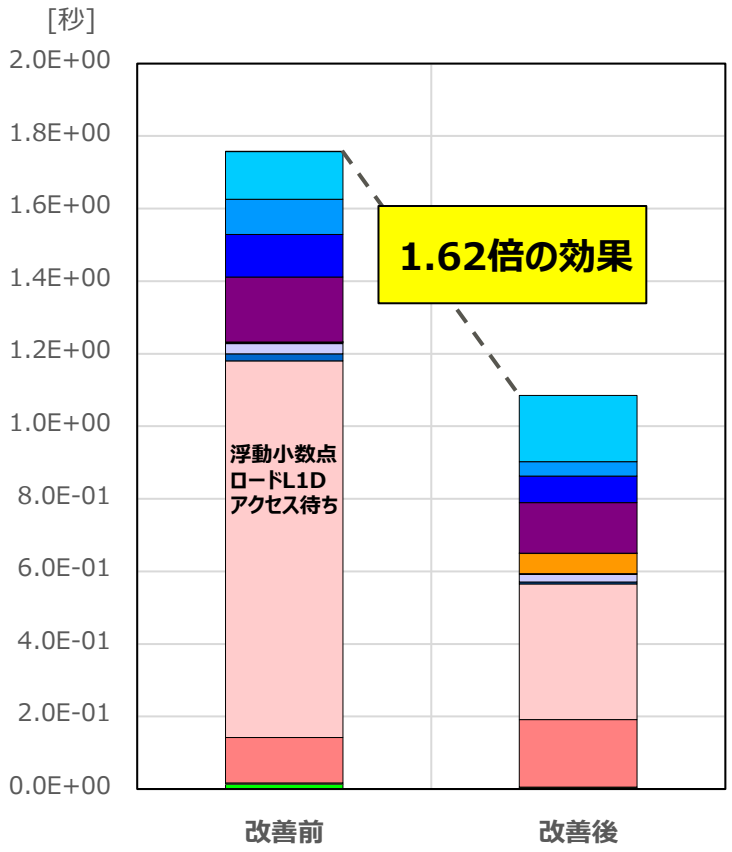
C/C++ Multiple Structures命令の活用（ソースチューニング）

FUJITSU

配列を3つずつの要素に分けることでMultiple Structures命令が適用され、浮動小数点ロードL1Dアクセス待ちが削減されました。

改善後ソース（ソースチューニング）

```
32          void MultiStructure(int n, int m, int iter)
33          {
34            int i,k;
35
36          #pragma omp parallel
37            {
            <<< Loop-information Start >>>
 :
            <<< Loop-information  End >>>
38              for(k=0; k<iter; k++)
39              {
40          #pragma omp for nowait
            <<< Loop-information Start >>>
 :
            <<< Loop-information  End >>>
41  p     v       for(i=0; i<N; i++)
42  p     v       {
43  p     v         b[i] = a[i][0] + a[i][1] + a[i][2]
                        + c[i][0] + c[i][1] + c[i][2];
44  p     v       }
45              }
46            }
47
48            return;
49          }
```



Multiple Structures命令適用
配列aが６個の要素であるため、配列cを追加して3個ずつの要素にした。

ハードウェアプリフェッチの距離調整

- プリフェッチの距離について
- ハードウェアプリフェッチの距離調整機能
- ハードウェアプリフェッチの距離調整機能(onL2)
- ハードウェアプリフェッチの距離調整機能(onメモリ)

プリフェッチの距離について

ハードウェアプリフェッチ・ソフトウェアプリフェッチの距離について

- ハードウェアプリフェッチ・ソフトウェアプリフェッチでは、以下のライン先のデータに対しプリフェッチを行います。

ハードウェアプリフェッチ		ソフトウェアプリフェッチ	
L1プリフェッチ	L2プリフェッチ	L1プリフェッチ	L2プリフェッチ
最大6ライン	**最大40ライン**	**自動**	**自動**

ハードウェアプリフェッチの距離はユーザの設定も可能になる

ソフトウェアプリフェッチは距離の自動調整がある

- プリフェッチの距離はアプリケーションのアクセス依存です。
  プリフェッチするキャッシュラインを遠くにするとスラッシングする場合があるため、
  近くのキャッシュラインをプリフェッチすることをお勧めします。

ハードウェアプリフェッチの距離設定機能

FUJITSU

ハードウェアプリフェッチ距離設定用コマンド(hwpfctl)

項目	内容
書式	hwpfctl [--disableL1] [--disableL2] [--distL1 lines_l1] [--distL2 lines_l2] [--weakL1] [--weakL2] [--verbose] command {arguments ...} hwpfctl --default [--verbose] command {arguments ...} hwpfctl --reset [--verbose] hwpfctl --help
説明	hwpfctlコマンドは、A64FXに搭載されているハードウェアプリフェッチ(stream detect mode)の振る舞いを変更する。 変更対象となるCPUコアは、プロセスアフィニティに準ずる。
オプション	--disableL1 --disableL2 L1/L2キャッシュに対するハードウェアプリフェッチを無効とする。省略時はハードウェアプリフェッチが有効である。 --distL1=lines_l1 --distL2=lines_l2 L1/L2キャッシュに対してプリフェッチするキャッシュラインを、キャッシュミスが発生したキャッシュラインを起点とするキャッシュラインの数で指定する。lines_l1にはL1キャッシュに対するプリフェッチ対象ラインを1から15までの値を指定できる。また、lines_l2にはL2キャッシュに対するプリフェッチ対象ラインを1から60までの値を指定できる。ただし、lines_l2に指定した値は4の倍数に切り上げられてシステムレジスタに書き込まれる。0を指定した場合はCPUのデフォルト値で動作する。省略時ならびに無効な値が指定された場合は0が指定されたものとみなす。 --weakL1 --weakL2 L1/L2キャッシュに対するプリフェッチ要求の優先度をweakとすることを指示する。省略時はstrongである。 --default デフォルト設定でcommandを起動する。--verbose以外のオプションは無視する。 --reset システムレジスタの値を初期化する。--verbose以外のオプションは無視する。 --verbose システムレジスタの変更前後の値を出力する。 --help 使用方法を表示する。

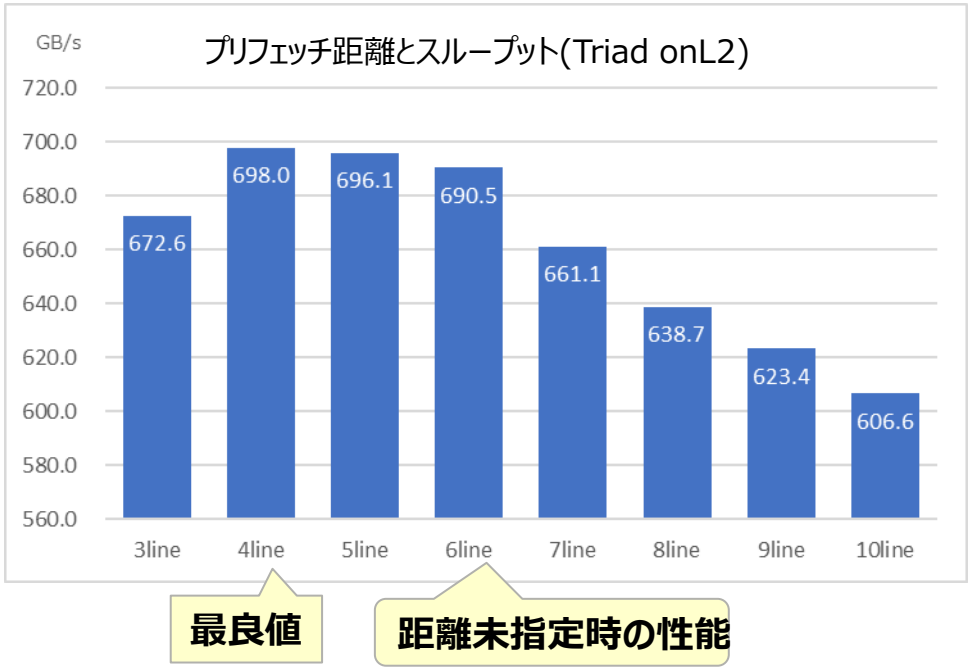
ハードウェアプリフェッチ距離設定用コマンド(hwpfctl)の使用例

```
hwpfctl  --distL1=6  --distL2=40  a.out
```

ハードウェアプリフェッチの距離調整の施行結果(onL2)

以下にハードウェアプリフェッチの距離調整の施行結果を示します。

- Triad(L2キャッシュアクセスケース)によるL1プリフェッチ距離評価



Triad

```
!$omp parallel
  Do j = 1, iter
!$omp do
     Do i = 1, n
        y(i)=x1(i) + c0 * x2(i)
     End Do
!$omp end do nowait
  End Do
!$omp end parallel
```

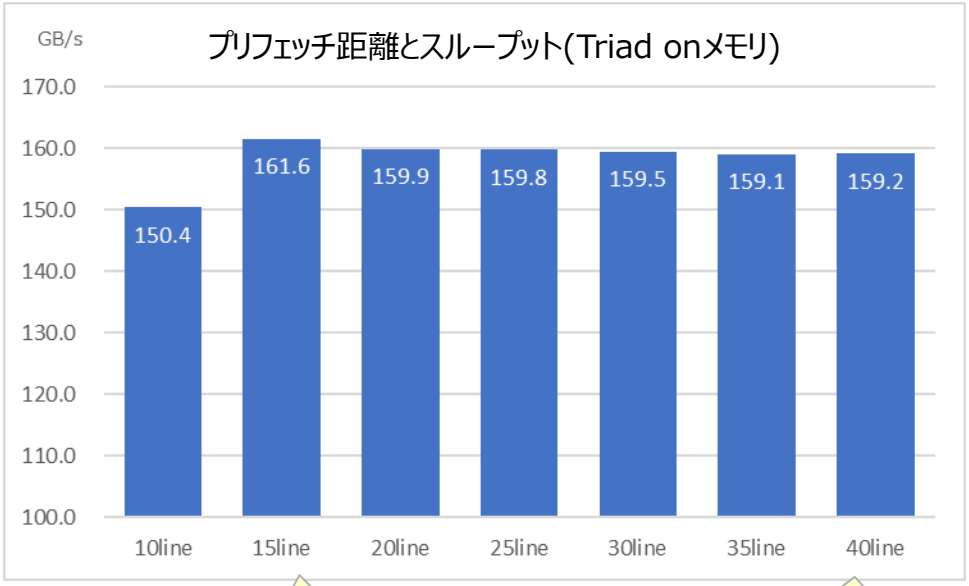
- 設定コマンド

```
hwpfctl  –distL1=3~10   a.out
```

ハードウェアプリフェッチの距離調整の施行結果(onメモリ)

FUJITSU

- **Triad(メモリアクセスケース)によるL2プリフェッチ距離評価**



Triad

```
!$omp parallel
  Do j = 1, iter
!$omp do
     Do i = 1, n
       y(i)=x1(i) + c0 * x2(i)
     End Do
!$omp end do nowait
  End Do
!$omp end parallel
```

※スループットは、データ書き込み先キャッシュラインの読み込み分を含めない値になります

- **設定コマンド**

```
hwpfctl  –distL2=10~40  a.out
```

FUJITSU

SVEのベクトルレジスタサイズ(SIMD幅)

- SVEのベクトルレジスタサイズ(SIMD幅)について
- -Ksimd_reg_size=agnostic指定時の最適化影響(注意点)

SVEのベクトルレジスタサイズ(SIMD幅)について

FUJITSU

- プロセッサでサポートしているSIMD幅

  ARMのSVE拡張はVector Length(SIMD幅)が128bitから2048bitの範囲の128bit単位で実装者が自由に決めることが可能である。
  A64FXではVector Lengthを512bitで実装を行っている。

  - A64FXでサポートするVector Lengthは512bit, 256bit, 128bit

- コンパイラオプション

  - -Ksimd_reg_size={ 128 | 256 | 512 | agnostic }
    デフォルトは、-Ksimd_reg_size=512です。
    - simd_reg_size={ 128 | 256 | 512 }
      SVEのベクトルレジスタサイズを指定します。単位はビットです。翻訳時に、本オプションで指定した値がSVEのベクトルレジスタサイズであるとみなして、最適化を実施します。ただし、生成した実行可能プログラムは、本オプションで指定したサイズのSVEのベクトルレジスタを実装しているCPUアーキテクチャでのみ正常に動作します。
      指定したベクトルレジスタのサイズが、実行するCPUアーキテクチャのベクトルレジスタのサイズより小さいことが明らかな場合、prctl(2)システムコールなどを利用して有効なベクトルレジスタのサイズを設定する必要があります。
    - simd_reg_size=agnostic
      SVEのベクトルレジスタを特定のサイズとみなさず翻訳を行い、実行時にSVEのベクトルレジスタサイズを決定する実行可能プログラムを作成します。この実行可能プログラムは、CPUアーキテクチャに実装されたSVEのベクトルレジスタサイズによらず実行可能です。
      ただし、-Ksimd_reg_size={128|256|512}オプションを指定した場合と比べて、実行性能が低下する場合があります。

Fortran -Ksimd_reg_size=agnostic指定時の最適化影響(注意点)

FUJITSU

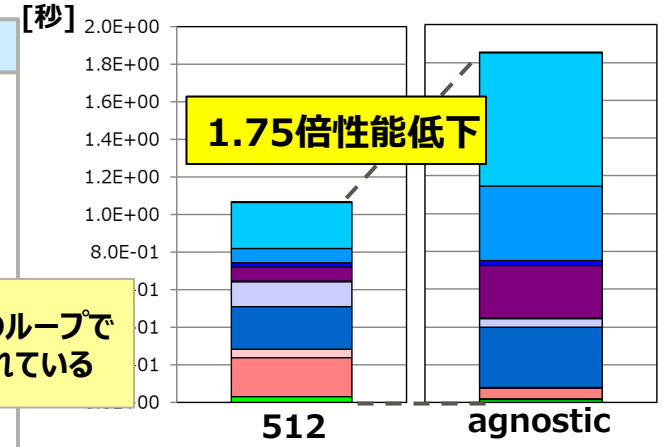
-Ksimd_reg_size=agnostic指定時に-Ksimd_reg_size=512(デフォルト)とは同様の最適化が行われないことがあります。その場合、実行性能低下する可能性があります。

-Ksimd_reg_size=512(デフォルト)ソース

```
24    2  p          do k=1,L
25    3  p           do ii=1,N,blk
                     <<< Loop-information Start >>>
                     <<<  [OPTIMIZATION]
                     <<<    SOFTWARE PIPELINING(IPC: 0.31, ITR: 192,
                                                MVE: 2, POL: L)
                     <<<    PREFETCH(
                     <<<      b, a
                     <<< Loop-informa
26    4  p  8          do j=1,M
                     <<< Loop-information Start >>>
                     <<<  [OPTIMIZATION]
                     <<<    SIMD(VL: 8)
                     <<<    FULL UNROLLING
                     <<< Loop-information  End >>>
27    5  p  fv           do i=ii,ii+blk-1
28    5  p  fv             a(i,j,k)=a(i,j,k)+c*b(i,j,k)
29    5  p  fv           enddo
30    4  p  8          enddo
31    3  p           enddo
32    2  p          enddo
```

blk=8

最内ループがSIMD長であり、外側のループでソフトウェアパイプライニングが適用されている



-Ksimd_reg_size=agnostic指定時ソース

```
24    2  p          do k=1,L
25    3  p           do ii=1,N,blk
26    4  p             do j=1,M
                     <<< Loop-information Start >>>
                     <<<  [OPTIMIZATION]
                     <<<    SIMD(VL: AGNOSTIC; VL: 2 in 128-bit)
                     <<<    PREFETCH(HARD) Expected by compiler :
                     <<<      b, a
                     <<< Loop-information  End >>>
27    5  p  2v           do i=ii,ii+blk-1
28    5  p  2v             a(i,j,k)=a(i,j,k)+c*b(i,j,k)
29    5  p  2v           enddo
30    4  p             enddo
31    3  p           enddo
32    2  p          enddo
```

ソフトウェアパイプライニングが適用されない

■ 注意事項

- **-Ksimd_reg_size=agnostic指定で翻訳した場合にはSIMD幅に依存した最適化（上記の例ではソフトウェアパイプライニング）が行われません。**

C/C++ -Ksimd_reg_size=agnostic指定時の最適化影響(注意点)

FUJITSU

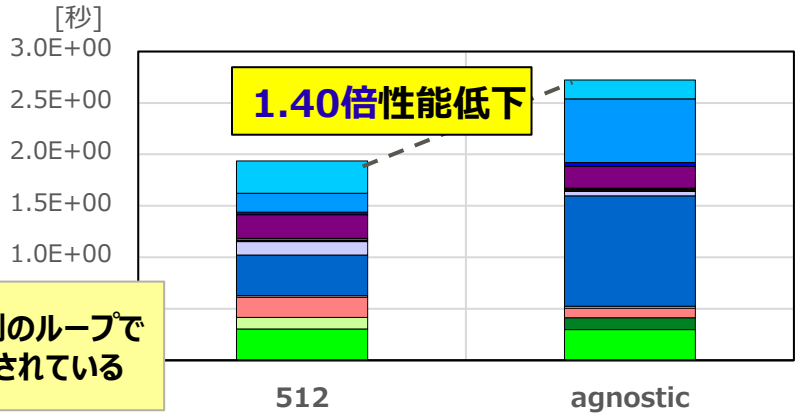
-Ksimd_reg_size=agnostic指定時に-Ksimd_reg_size=512(デフォルト)とは同様の最適化が行われないことがあります。その場合、実行性能低下する可能性があります。

-Ksimd_reg_size=512(デフォルト)ソース

```
46            for(iter=1; iter<=itmax; iter++) {
48              #pragma omp for
49  p            for(k=0;k<L; k++) {
51  p             for(ii=0;ii<N;ii+=blk) {
          <<< Loop-information Start >>>
          <<<  [OPTIMIZATION]
 :
          <<< Loop-information  En
53  p                for(j=0;j<M;j++) {
          <<< Loop-information Sta
          <<<  [OPTIMIZATION]
          <<<    SIMD(VL: 8)
          <<<    SOFTWARE PIPELINING(IPC: 3.00, ITR: 192,
                                     MVE: 7, POL: S)
          <<<    PREFETCH(HARD) Expected by compiler :
          <<<      (unknown)
          <<< Loop-information  End >>>
55  p    2v            for(i=ii; i<ii+blk-1; i++) {
57  p    2v              a[k][j][i]=a[k][j][i]+c*b[k][j][i];
58  p    2v            }
59  p                }
60  p              }
61  p            }
62             }
```

blk=8

最内ループがSIMD長であり、外側のループでソフトウェアパイプライニングが適用されている



-Ksimd_reg_size=agnostic指定時ソース

```
46            for(iter=1; iter<=itmax; iter++) {
48              #pragma omp for
49  p            for(k=0;k<L; k++) {
51  p             for(ii=0;ii<N;ii+=blk) {
53  p                for(j=0;j<M;j++) {
          <<< Loop-information Start >>>
          <<<  [OPTIMIZATION]
          <<<    SIMD(VL: AGNOSTIC; VL: 2 in 128-bit)
          <<<    PREFETCH(HARD) Expected by compiler :
          <<<      (unknown)
          <<< Loop-information  End >>>
55  p    2v            for(i=ii; i<ii+blk-1; i++) {
57  p    2v              a[k][j][i]=a[k][j][i]+c*b[k][j][i];
58  p    2v            }
59  p                }
60  p              }
61  p            }
62             }
```

ソフトウェアパイプライニングが適用されない

- **注意事項**
- **-Ksimd_reg_size=agnostic指定で翻訳した場合にはSIMD幅に依存した最適化（上記の例ではソフトウェアパイプライニング）が行われません。**

FUJITSU

半精度実数型の活用

- 半精度実数型の活用（改善前）
- 半精度実数型の活用（ソースチューニング）

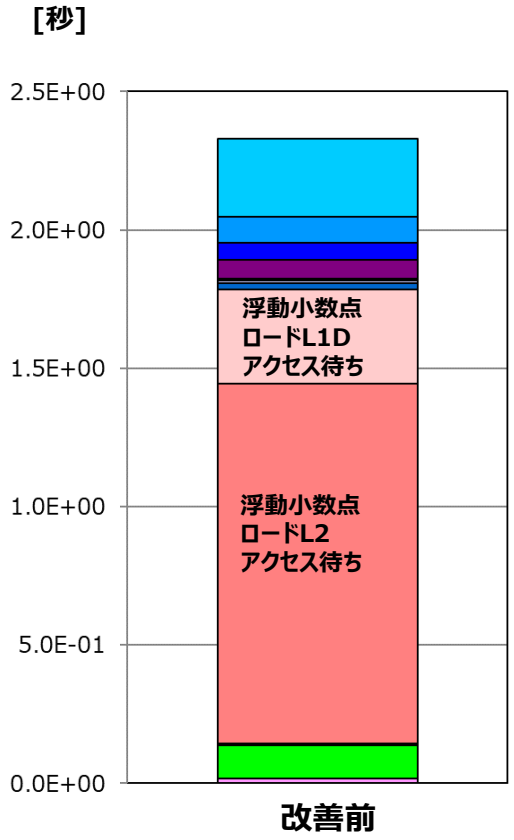
Fortran 半精度実数型の活用（改善前）

FUJITSU

倍精度実数型のデータの場合、SIMD長は8ですが、データの精度を落とすことで SIMD長を長くしバンド幅・演算器を有効活用できます。

改善前ソース

```
 2                integer,parameter::N=60000
 :
19                 real(8)::x1(N),x2(N),y(N)
20                 real(8),parameter::c=0.5
 :
                <<< Loop-information Start >>>
                <<<  [OPTIMIZATION]
                <<<    SIMD(VL: 8)
                <<<    SOFTWARE PIPELINING(IPC: 3.25, ITR: 144,
                                                  MVE: 4, POL: S)
                <<<    PREFETCH(HARD) Expected by compiler :
                <<<      x2, x1, y
                <<< Loop-information  End >>>
24    2   p   2v         do i=1,N
25    2   p   2v           y(i) = x1(i) + c * x2(i)
26    2   p   2v         enddo
```



SIMD幅(Vector Length)=512[bit]の場合のSIMD長

データ型	SIMD長
倍精度型	8
単精度型	16
半精度型	32
1バイト型	64

	GFLOPS	Floating-point operation peak ratio (%)
改善前	51.52	6.71%

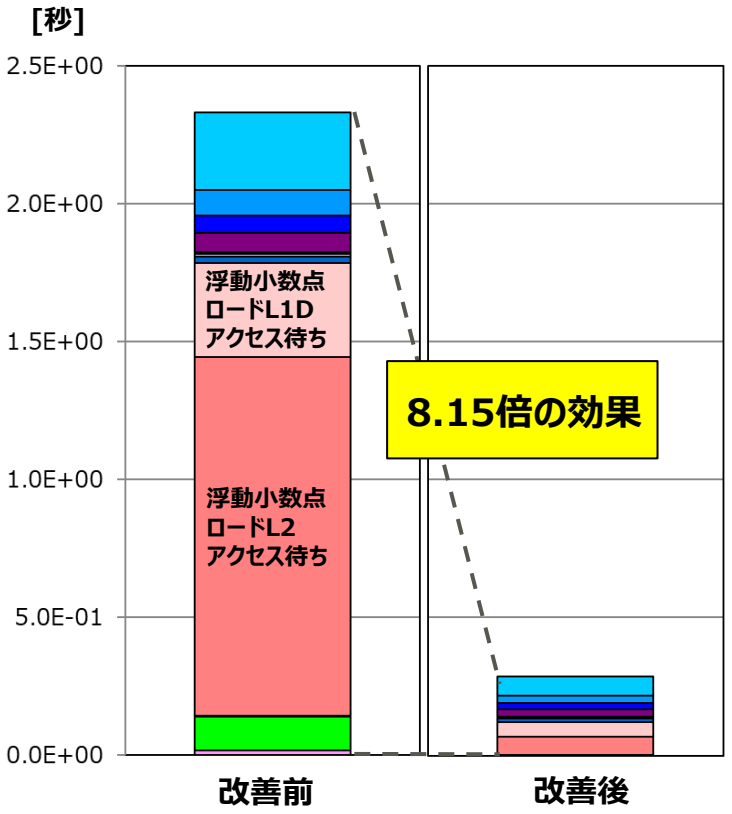
Fortran 半精度実数型の活用（ソースチューニング）

FUJITSU

桁数を落とした半精度実数型を使うことでSIMD長を32に拡張することができます。
データ容量が小さくなることで浮動小数点ロードアクセス待ちが改善されました。

改善後ソース

```
2                integer,parameter::N=60000
:
19                real(2)::x1(N),x2(N),y(N)
20                real(2),parameter::c=0.5
:
              <<< Loop-information Start >>>
              <<<  [OPTIMIZATION]
              <<<    SIMD(VL: 32)
              <<<    SOFTWARE PIPELINING(IPC: 3.25, ITR: 576,
                                         MVE: 4, POL: S)
              <<<    PREFETCH(HARD) Expected by compiler :
              <<<      x2, x1, y
              <<< Loop-information  End >>>
24    2  p  2v         do i=1,N
25    2  p  2v           y(i) = x1(i) + c * x2(i)
26    2  p  2v         enddo
```



	GFLOPS	Floating-point operation peak ratio (%)
改善前	51.52	6.71%
改善後	421.57	54.89%

半精度実数型の活用（改善前）

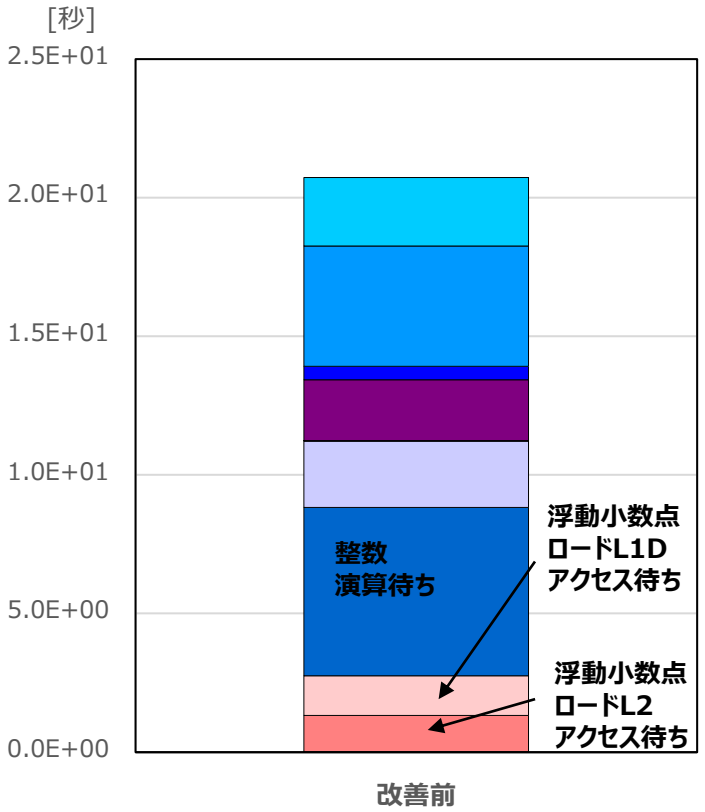
倍精度実数型のデータの場合、SIMD長は8ですが、データの精度を落とすことで SIMD長を長くしバンド幅・演算器を有効活用できます。

改善前ソース

```
34          double const c=0.5;
35          int i,k;
36
37      4    for(k=0;k<itmax; k++)
38          {
              <<< Loop-information Start >>>
              <<<  [OPTIMIZATION]
              <<<    SIMD(VL: AGNOSTIC;
                          VL: 2 in 128-bit Interleave: 1)
              <<< Loop-information End >>>
39        v     for(i=0;i<N; i++)
40            {
41              y[i] = x1[i] + c * x2[i];
42            }
43          }
```

N = 60000
itmax = 1000000

配列宣言
double x1[N], x2[N], y[N];



SIMD幅(Vector Length)=512[bit]の場合のSIMD長

データ型	SIMD長
倍精度型	8
単精度型	16
半精度型	32
1バイト型	64

Statistics	GFLOPS	Floating-point operation
改善前	5.79	1.20E+11

C/C++ clangモード

半精度実数型の活用（ソースチューニング）

FUJITSU

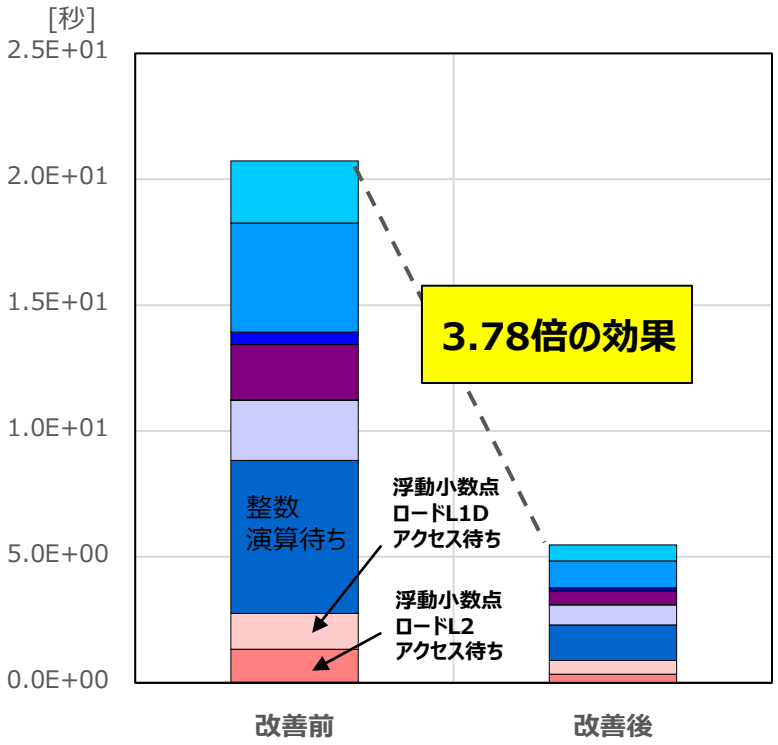
桁数を落とした半精度実数型を使うことでSIMD長を32に拡張することができます。
データ容量が小さくなることで浮動小数点ロードアクセス待ちが改善されました。

改善後ソース

```
34          _Float16 c = 0.5f16;
35          int i,k;
36
37      4    for(k=0;k<itmax; k++)
38          {
              <<< Loop-information Start >>>
              <<<  [OPTIMIZATION]
              <<<    SIMD(VL: AGNOSTIC;
                        VL: 8 in 128-bit Interleave: 1)
              <<< Loop-information End >>>
39        v     for(i=0;i<N; i++)
40            {
41              y[i] = x1[i] + c * x2[i];
42            }
43          }
```

N = 60000
itmax = 1000000

配列宣言
double x1[N], x2[N], y[N];



■ 注意事項

- C/C++のtradモードでは半精度実数型機能がないため、本手法によるチューニングはできません。

Statistics	GFLOPS	Floating-point operation
改善前	5.79	1.20E+11
改善後	21.91	1.20E+11

FUJITSU

スレッド並列チューニング

- スレッド並列化率の改善
- スレッド並列化率の向上
- スレッド並列実行効率の改善
- ラージページの設定による実行効率の改善
- メモリ使用量の削減

FUJITSU

スレッド並列化率の改善

- スレッド並列化率とは
- スレッド並列化率の向上

スレッド並列化率とは

並列化率とは 1 並列実行時の並列実行可能部分の占める割合を意味します。

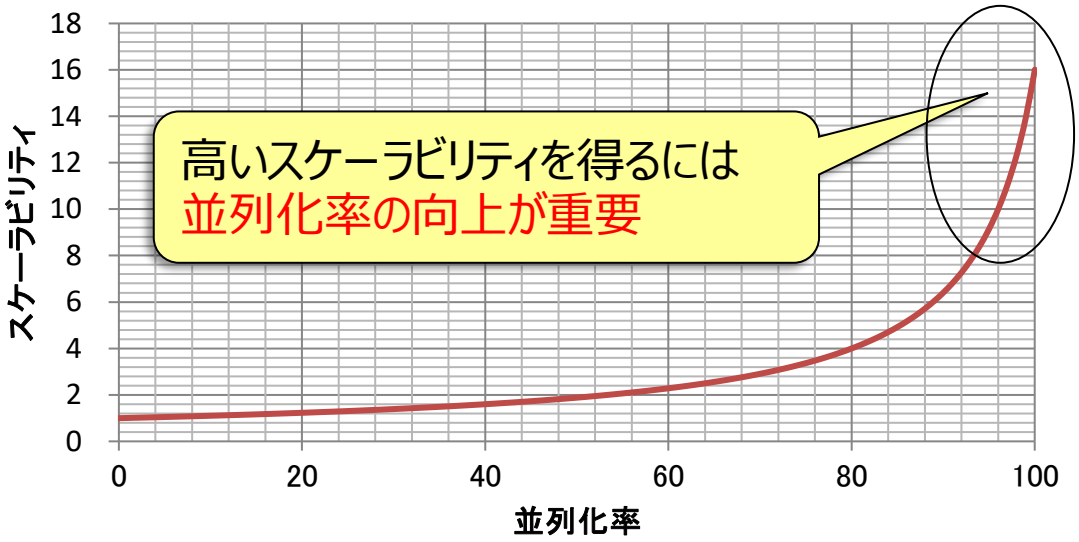
n 並列実行時のスレッド並列化率とスケーラビリティの関係を表す法則としてアムダールの法則があります。

■ アムダールの法則

$$\text{スケーラビリティ} = \frac{1}{(1-p)+\frac{p}{n}}$$

- p：並列化率
- n：並列数

n ＝ 16（理想的な１６並列処理）の場合



FUJITSU

スレッド並列化率の向上

- 定義引用関係が明らかでないループ
- ポインタ変数が含まれるループ
- データ依存関係があるループ

定義引用関係が明らかでないループ

FUJITSU

- **!OCL NORECURRENCE**

  以下のプログラムで、配列aの添字式が別の配列要素y(j) のため、本処理系は配列aをループスライスしても問題ないか判断できません。配列a をループスライスしても問題ないとプログラマが分かっている場合にNORECURRENCE指示子を指定することで並列化することができます。

改善前ソース	改善後ソース
<<< Loop-information Start >>> <<< [OPTIMIZATION] <<< SOFTWARE PIPELINING(IPC: 0.32, ITR: 6, MVE: 2, POL: S) <<< PREFETCH(HARD) Expected by compiler : <<< b, y <<< Loop-information End >>> 6 1 s 2s do i=1,160000 7 1 m 2m a(y(i))=a(y(i))+b(i) 8 1 p 2v end do : jwd5228p-i "a.f90", line 7: データの定義引用の順序が逐次実行と異なるため、このDOループは並列化できません。 jwd6228s-i "a.f90", line 7: データの定義引用の順序が逐次実行と異なる可能性があるため、このDOループはSIMD化できません。	5 !ocl norecurrence(a) <<< Loop-information Start >>> <<< [PARALLELIZATION] <<< Standard iteration count: 762 <<< [OPTIMIZATION] <<< SIMD(VL: 16) <<< SOFTWARE PIPELINING(IPC: 2.33, ITR: 416, MVE: 7, POL: S) <<< PREFETCH(HARD) Expected by compiler : <<< y, b <<< Loop-information End >>> 6 1 pp 2v do i=1,160000 7 1 p 2v a(y(i))=a(y(i))+b(i) 8 1 p 2v end do

！注意！

- NORECURRENCE指示子をループスライス不可能な配列に指定した場合、本処理系は誤ったループスライスを行うことがあります。
- 配列名を省略した場合、対象範囲内のすべての配列に有効となります。

ポインタ変数が含まれるループ

- **!OCL NOALIAS**

  ポインタ変数が記憶領域のどこを占めるかは実行時に決まるために、データの依存関係が不明となり並列化されません。ポインタ変数が同一の記憶領域を指すことはないとプログラマが分かっている場合にNOALIAS指示子を指定することで並列化することができます。

改善前ソース	改善後ソース
1 real,dimension(100000),target::x 2 real,dimension(:),pointer::a,b 3 a=>x(1:10000) 4 b=>x(10001:20000) 5 6 <<< Loop-information Start >>> <<< [OPTIMIZATION] <<< PREFETCH(HARD) Expected by compiler : <<< x <<< Loop-information End >>> 7 1 s 4s do i=1,100000 8 1 s 4s b(i) = a(i)+1.0 9 1 s 4s end do jwd5228p-i "a.f90", line 8: データの定義引用の順序が逐次実行と異なるため、このDOループは並列化できません。 jwd6228s-i "a.f90", line 8: データの定義引用の順序が逐次実行と異なる可能性があるため、このDOループはSIMD化できません。	1 real,dimension(100000),target::x 2 real,dimension(:),pointer::a,b 3 a=>x(1:10000) 4 b=>x(10001:20000) 5 6 !ocl noalias <<< Loop-information Start >>> <<< [PARALLELIZATION] <<< Standard iteration count: 1143 <<< [OPTIMIZATION] <<< SIMD(VL: 16) <<< SOFTWARE PIPELINING(IPC: 2.75, ITR: 288, MVE: 4, POL: S) <<< PREFETCH(HARD) Expected by compiler : <<< (unknown) <<< Loop-information End >>> 7 1 pp 2v do i=1,100000 8 1 p 2v b(i) = a(i)+1.0 9 1 p 2v end do

データ依存関係があるループ

- ピーリングを行い並列化

以下のループは配列aについて、i=1の時とi=nの時に依存性を持っているので並列化されません。ループの最初や最後をループ外に出して依存をなくすことで並列化が促進されます。

改善前ソース	改善後ソース

```
      <<< Loop-information Start >>>
      <<<  [OPTIMIZATION]
      <<<    PREFETCH(HARD) Expected by compiler :
      <<<      b, a
      <<< Loop-information  End >>>
4     1   s   2s          do i=1,n
5     1   s   2m            a(i)=a(1)+b(i)+a(n)
6     1   s   2v          end do

   jwd5202p-i  "a.f90", line 5: データの定義引用の順序が
逐次実行と異なるため、このDOループは並列化できません。(名前:a)
   jwd5208p-i  "a.f90", line 5: 定義引用の順序が分からな
いため、定義引用順序が逐次実行と異なる可能性があり、
このDOループは並列化できません。(名前:a)
```

```
4                   a(1)=a(1)+b(1)+a(n)
      <<< Loop-information Start >>>
      <<<  [PARALLELIZATION]
      <<<    Standard iteration count: 843
      <<<  [OPTIMIZATION]
      <<<    SIMD(VL: 16)
      <<<    SOFTWARE PIPELINING(IPC: 3.00,
                                 ITR: 384, MVE: 4, POL: S)
      <<<    PREFETCH(HARD) Expected by compiler :
      <<<      a, b
      <<< Loop-information  End >>>
5     1  pp   2v            do i=2,n-1
6     1   p   2v              a(i)=a(1)+b(i)+a(n)
7     1   p   2v            enddo
8                   a(n)=a(1)+b(n)+a(n)
```

FUJITSU

スレッド並列実行効率の改善

- False Sharingの改善
- 処理量が不規則なループ
- 適切な並列化次元での並列化

FUJITSU

False Sharingの改善

- False Sharingとは
- False Sharing（改善前）
- False Sharing（ソースチューニング）

False Sharingとは

FUJITSU

False Sharingとは、スレッド間でキャッシュラインのInvalidateおよびCopy Backが頻発する現象のことです。

４スレッド並列と仮定した場合の例

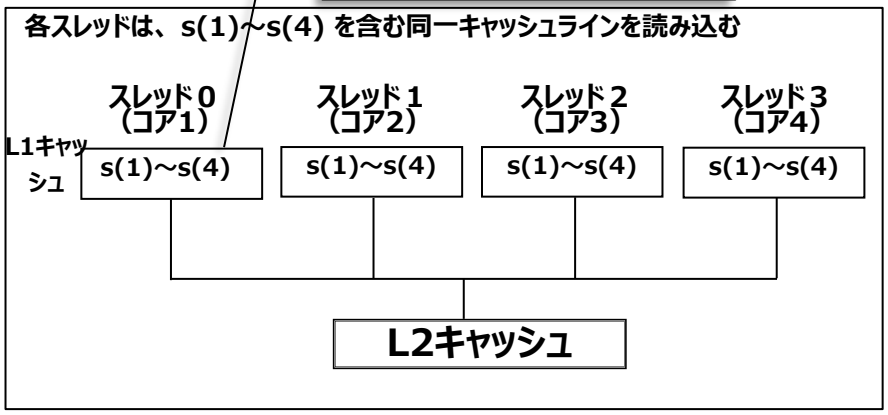
改善前ソース

```
1                    subroutine sub(s,a,b,ni,nj)
2                    real*8 a(ni,nj),b(ni,nj)
3                    real*8 s(nj)
4
5    1  pp             do j = 1, nj
6    1   p              s(j)=0.0
7    2   p   8v           do i = 1, ni
8    2   p   8v             s(j)=s(j)+a(i,j)*b(i,j)
9    2   p   8v           end do
10   1   p              end do
11
12                   end
```

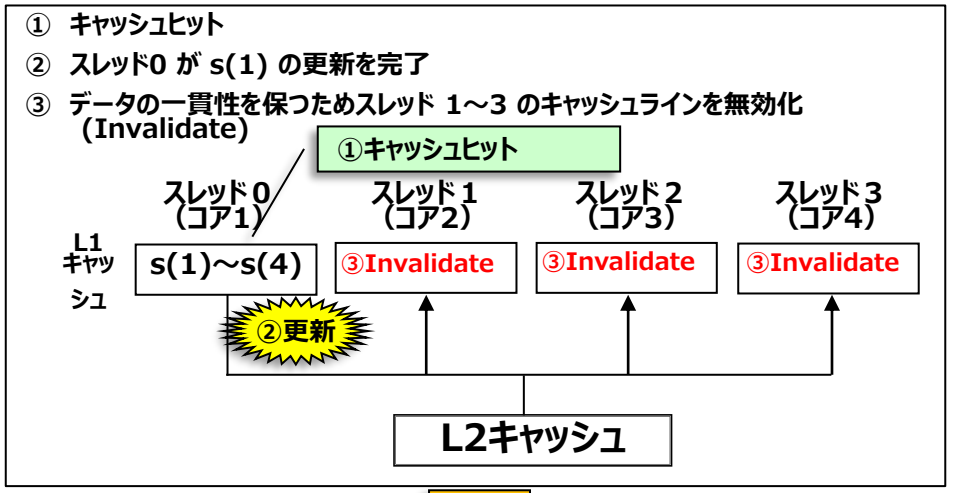
nj=4
ni=2000

■ 初期状態

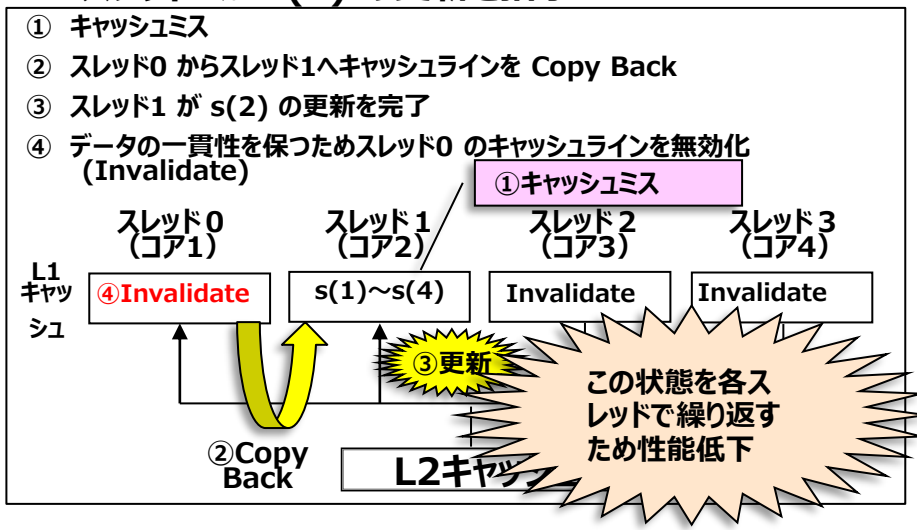
キャッシュにはキャッシュライン単位でデータが載る



■ スレッド0が s(1) の更新を指示



■ スレッド1が s(2) の更新を指示



Fortran False Sharing（改善前）

FUJITSU

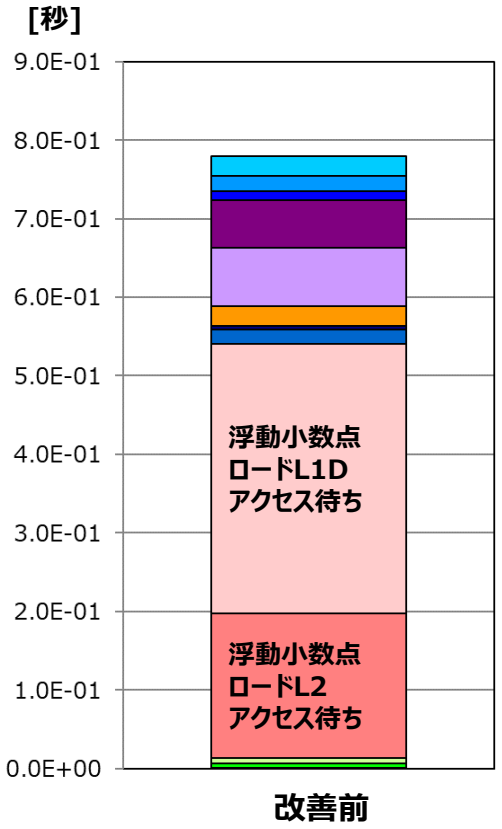
並列化次元のjの繰返し数が16回転と少なく、配列aのデータがスレッド間でキャッシュラインを共有してしまうため、FalseSharingが発生します。
そのため、データアクセス待ちが多くなっています。

改善前ソース

```
22                    subroutine sub(flag)
23                    integer*8 i,j,n
24                    parameter(n=60000)
25                    parameter(m=16)
26                    real*8 a(m,n),b(m,n)
27                    integer flag(m,n)
28                    common /com/a,b
29
:
32    1   p             do i=1,m
                <<< Loop-information Start >>>
                <<<  [OPTIMIZATION]
                <<<    SIMD(VL: 8)
                <<<    SOFTWARE PIPELINING(IPC: 2.28, ITR: 128,
                                           MVE: 3, POL: S)
                <<< Loop-information  End >>>
33    2   p   2v            do j=1,n
34    3   p   2v              if(flag(i,j).eq.1)th
35    3   p   2v                a(i,j)=b(j,i)
36    3   p   2v              endif
37    2   p   2v            enddo
38    1   p               enddo
```

並列化次元の繰返し数が16回転

False Sharing発生



Cache

	L1I miss rate (/Effective instruction)	Load-store instruction	L1D miss	L1D miss rate (/Load-store instruction)	L1D miss demand rate (%) (/L1D miss)	L1D miss hardware prefetch rate (%) (/L1D miss)	L1D miss software prefetch rate (%) (/L1D miss)	L2 miss	L2 miss rate (/Load-store instruction)	L2 miss demand rate (%) (/L2 miss)	L2 miss hardware prefetch rate (%) (/L2 miss)	L2 miss software prefetch rate (%) (/L2 miss)
改善前	0.00	3.57E+09	7.27E+08	0.20	9.39%	90.85%	0.00%	1.75E+04	0.00	21.82%	100.00%	0.00%

Fortran False Sharing（ソースチューニング）

FUJITSU

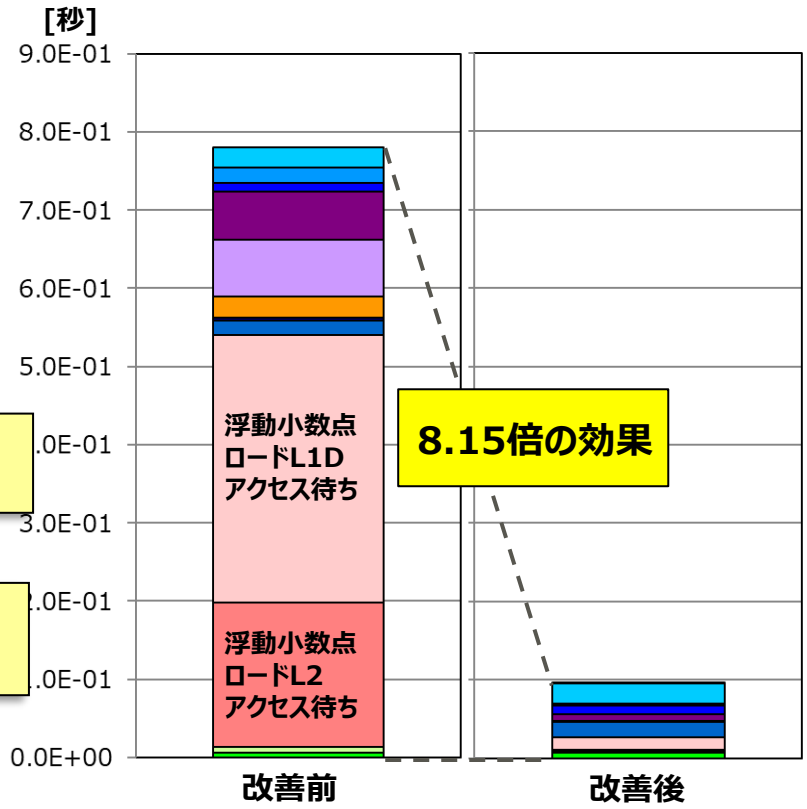
ループ交換を行い、外側で並列化することでFalse Sharingを回避できます。その結果、L1キャッシュミス数が削減され、データアクセス待ちが改善されました。

改善後ソース

```
23                    integer*8 i,j,n
24                    parameter(n=60000)
25                    parameter(m=16)
26                    real*8 a(m,n),b(m,n)
27                    integer flag(m,n)
:
              <<< Loop-information Start >>>
              <<<  [OPTIMIZATION]
              <<<    SOFTWARE PIPELINING(IPC: 1.32, ITR: 48,
                                         MVE: 2, POL: S)
              <<<    PREFETCH(SOFT) : 4
              <<<     SEQUENTIAL : 4
              <<<      b: 4
              <<< Loop-information  End >>>
32    1  p  2           do j=1,n
              <<< Loop-information Start >>>
              <<<  [OPTIMIZATION]
              <<<    SIMD(VL: 8)
              <<<    FULL UNROLLING
              <<< Loop-information  End >>>
33    2  p  fv             do i=1,m
34    3  p  fv               if(flag(i,j).eq.1)then
35    3  p  fv                 a(i,j)=b(j,i)
36    3  p  fv               endif
37    2  p  fv             enddo
38    1  p  2            enddo
```

ループ交換を行い 外側で並列化

False Sharing 回避



False Sharing 回避によりL1Dミス数が削減され性能向上した

	L1I miss rate (/Effective instruction)	Load-store instruction	L1D miss	L1D miss rate (/Load-store instruction)	L1D miss demand rate (%) (/L1D miss)	L1D miss hardware prefetch rate (%) (/L1D miss)	L1D miss software prefetch rate (%) (/L1D miss)	L2 miss	L2 miss rate (/Load-store instruction)	L2 miss demand rate (%) (/L2 miss)	L2 miss hardware prefetch rate (%) (/L2 miss)	L2 miss software prefetch rate (%) (/L2 miss)
改善前	0.00	3.57E+09	7.27E+08	0.20	9.39%	90.85%	0.00%	1.75E+04	0.00	21.82%	100.00%	0.00%
改善後	0.00	1.19E+09	5.29E+07	0.04	4.40%	84.91%	10.70%	1.61E+04	0.00	7.89%	85.83%	6.28%

C/C++ False Sharing（改善前）

FUJITSU

並列化次元のjの繰返し数が16回転と少なく、配列aのデータがスレッド間でキャッシュラインを共有してしまうため、FalseSharingが発生します。
そのため、データアクセス待ちが多くなっています。

改善前ソース

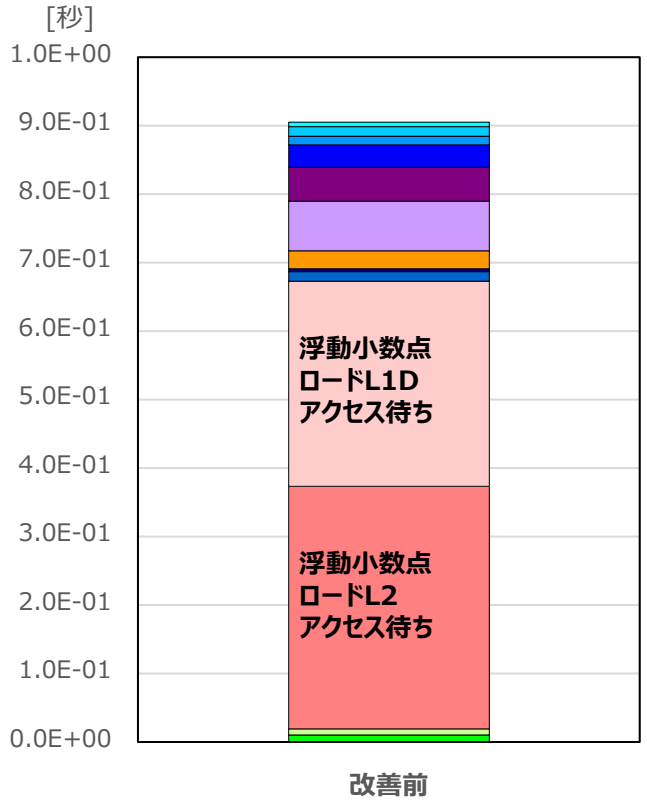
```
42              #pragma omp for
                <<< Loop-information Start >>>
                <<<  [OPTIMIZATION]
                <<<    PREFETCH(HARD) Expect
                <<<      b, a, (unknown)
                <<< Loop-information  End >>>
43   p            for (i=0;i<M;i++)
44   p            {
                <<< Loop-information Start >>>
                <<<  [OPTIMIZATION]
                <<<    SIMD(VL: 8)
                <<<    SOFTWARE PIPELINING(IPC: 2.00, ITR: 128,
                                  MVE: 3, POL: S)
                <<<    PREFETCH(HARD) Expected by compiler :
                <<<      b, a, (unknown)
                <<< Loop-information  End >>>
45   p    2v         for(j=0;j<N;j++)
46   p    2v         {
47   p    2v           if(flag[j][i]==1)
48   p    2v           {
49   p    2v             a[j][i]=b[i][j];
50   p    2v           }
51   p    2v         }
52   p              }
```

M=16
N=60000

配列宣言
double a[N][M],
b[N][M];

並列化次元の繰返し数が16回転

False Sharing発生



Cache	L1I miss rate (/Effective instruction)	Load-store instruction	L1D miss	L1D miss rate (/Load-store instruction)	L1D miss demand rate (%) (/L1D miss)	L1D miss hardware prefetch rate (%) (/L1D miss)	L1D miss software prefetch rate (%) (/L1D miss)	L2 miss	L2 miss rate (/Load-store instruction)	L2 miss demand rate (%) (/L2 miss)	L2 miss hardware prefetch rate (%) (/L2 miss)	L2 miss software prefetch rate (%) (/L2 miss)
改善前	0.00	4.04E+09	7.28E+08	0.18	30.45%	69.61%	0.00%	3.94E+04	0.00	48.46%	58.66%	0.00%

False Sharing（ソースチューニング）

ループ交換を行い、外側で並列化することでFalse Sharingを回避できます。その結果、L1キャッシュミス数が削減され、データアクセス待ちが改善されました。

改善後ソース

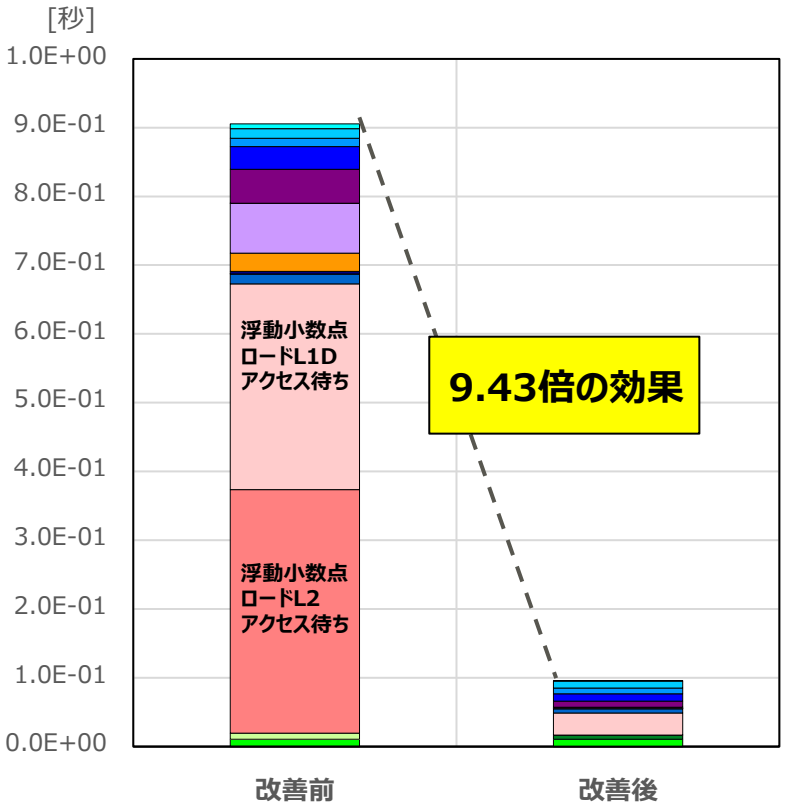
```
42              #pragma omp for
                <<< Loop-information Start >>>
                <<<  [OPTIMIZATION]
                <<<    SOFTWARE PIPELINING(
                                MVE: 3, POL: S)
                <<<    PREFETCH(HARD) Expect
                <<<      a, (unknown)
                <<< Loop-information  End >>>
43   p            for(j=0;j<N;j++)
44   p            {
                <<< Loop-information Start >>>
                <<<  [OPTIMIZATION]
                <<<    SIMD(VL: 8)
                <<<    FULL UNROLLING
                <<< Loop-information  End >>>
45   p    fv      for (i=0;i<M;i++)
46   p    fv      {
47   p    fv        if(flag[j][i]==1)
48   p    fv        {
49   p    fv          a[j][i]=b[i][j];
50   p    fv        }
51   p    fv      }
52   p            }
```

M=16
N=60000

配列宣言
double a[N][M], b[N][M];

ループ交換を行い 外側で並列化

False Sharing 回避



False Sharing 回避によりL1Dミス数が削減され性能向上した

Cache	L1I miss rate (/Effective instruction)	Load-store instruction	L1D miss	L1D miss rate (/Load-store instruction)	L1D miss demand rate (%) (/L1D miss)	L1D miss hardware prefetch rate (%) (/L1D miss)	L1D miss software prefetch rate (%) (/L1D miss)	L2 miss	L2 miss rate (/Load-store instruction)	L2 miss demand rate (%) (/L2 miss)	L2 miss hardware prefetch rate (%) (/L2 miss)	L2 miss software prefetch rate (%) (/L2 miss)
改善前	0.00	4.04E+09	7.28E+08	0.18	30.45%	69.61%	0.00%	3.94E+04	0.00	48.46%	58.66%	0.00%
改善後	0.00	1.27E+09	4.81E+07	0.04	6.15%	93.86%	0.00%	1.93E+04	0.00	15.65%	89.19%	0.00%

FUJITSU

処理量が不規則なループ

- 処理量が不規則なループ（改善前）
- 処理量が不規則なループ（OpenMP チューニング）

Fortran 処理量が不規則なループ（改善前）

FUJITSU

処理量が不規則な場合、スケジュール方法が static のサイクリック分割ではロードインバランスが発生します。下の例ではロードインバランスのためバリア同期待ちが多くなっています。

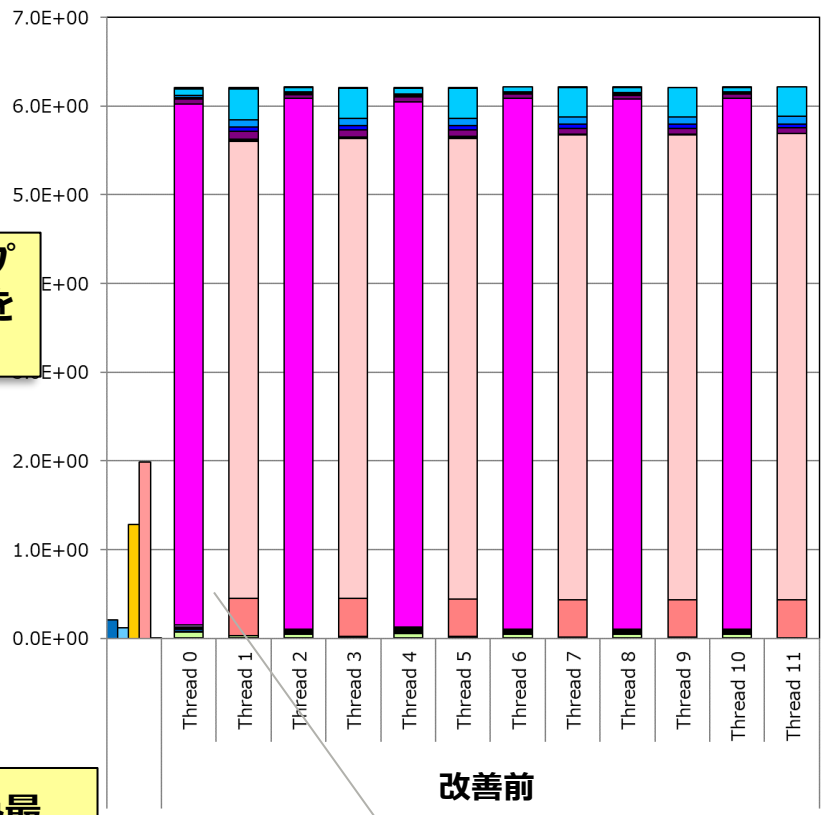
改善前ソース

```
1                 subroutine init(a,b,ie,n)
:
                <<< Loop-information Start >>>
                <<<  [OPTIMIZATION]
                <<<    FUSED
                <<< Loop-information  End >>>
8     1             do i=1,n
9     2               if (mod(i,2).eq.0) then
10    2                 ie(i)=100000
11    2               endif
12    1             enddo
:
16                  subroutine sub(a,b,s,ie,n
17                    real a(n),b(n),s
18                    integer ie(n)
19                    !$omp parallel do schedule(static,1)
20    1   p           do j=1,n
                <<< Loop-information Start >>>
                <<<  [OPTIMIZATION]
                <<<    SIMD(VL: 16)
                <<<    SOFTWARE PIPELINING(IPC: 3.50,
                                 ITR: 448, MVE: 7, POL: S)
                <<< Loop-information  End >>>
21    2   p   2v          do i=1,ie(j)
22    2   p   2v            a(i) = a(i)*b(i)*s
23    2   p   2v          enddo
24    1   p           enddo
25                  end subroutine sub
 :
27                  program main
28                    parameter(n=1000000)
31                    call init(a,b,ie,n)
 :
33                    call sub(a,b,2.0,ie,n)
 :
35                  end program main
```

偶数回転時のみ評価ループの終値である配列 ieに値を設定している

評価ループ

制御変数 j が偶数時のみ最内ループは100,000回転する。



ロードインバランスによるバリア同期待ちが発生

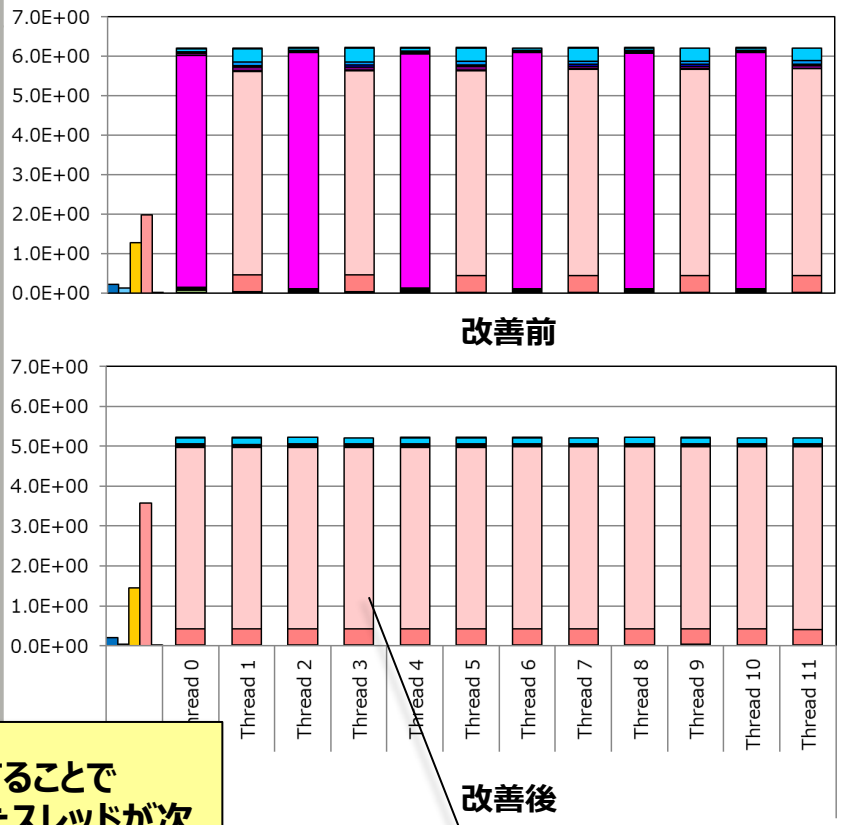
Fortran 処理量が不規則なループ（OpenMPチューニング）

FUJITSU

スケジュール方法をdynamicにすることで、先に処理が終了したスレッドが次の処理を行えるようになり、ロードインバランスが改善しました。

改善後ソース

```
1                 subroutine init(a,b,ie,n)
                 <<< Loop-information Start >>>
                 <<<  [OPTIMIZATION]
                 <<<    FUSED
                 <<< Loop-information  End >>>
8     1             do i=1,n
9     2               if (mod(i,2).eq.0) then
10    2                 ie(i)=100000
11    2               endif
12    1             enddo
 :
16                 subroutine sub(a,b,s,ie,n)
17                   real a(n),b(n),s
18                   integer ie(n)
19                   !$omp parallel do schedule(dynamic,1)
20    1   p           do j=1,n
                 <<< Loop-information Start >>>
                 <<<  [OPTIMIZATION]
                 <<<    SIMD(VL: 16)
                 <<<    SOFTWARE PIPELINING(IPC: 3.50, ITR: 448,
                                            MVE: 7, POL: S)
                 <<< Loop-information  End >>>
21    2   p   2v          do i=1,ie(j)
22    2   p   2v            a(i) = a(i)*b(i)*s
23    2   p   2v          enddo
24    1   p           enddo
25                 end subroutine sub
:
27                 program main
28                   parameter(n=1000000)
31                   call init(a,b,ie,n)
:
33                   call sub(a,b,2.0,ie,n)
```



dynamicを指定することで
先に処理が終了したスレッドが次の処理を行うようになる。

ロードインバランスが改善された

C/C++ 処理量が不規則なループ（改善前）

FUJITSU

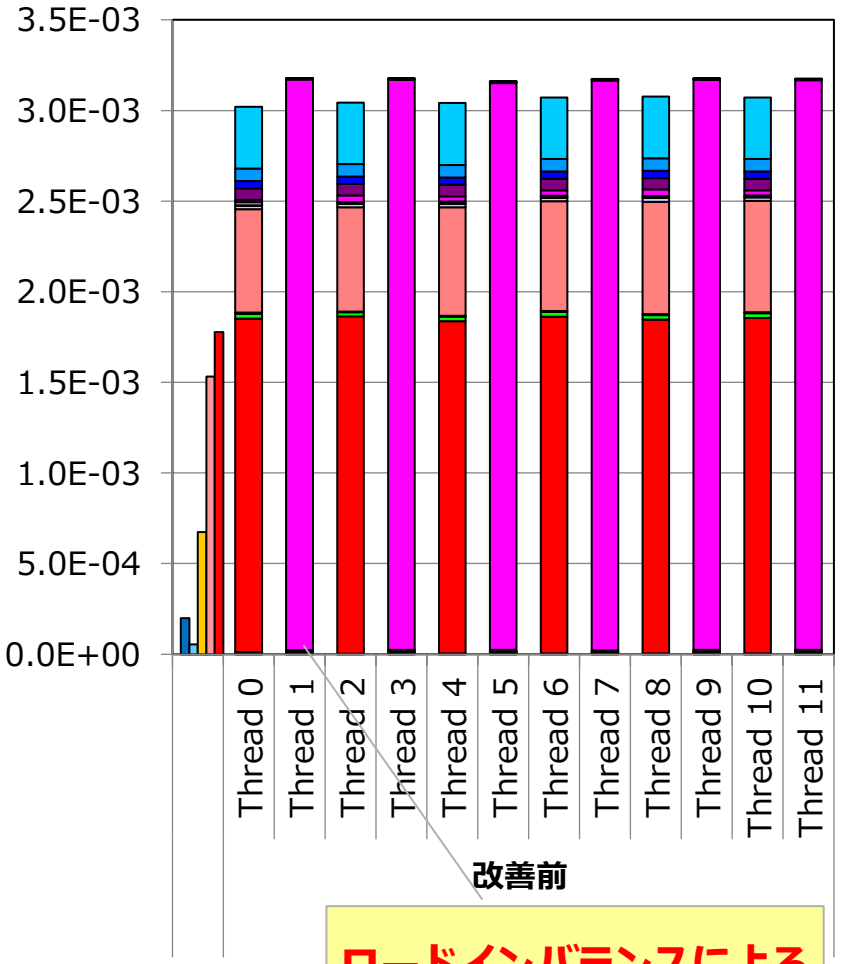
処理量が不規則な場合、スケジュール方法が static のサイクリック分割ではロードインバランスが発生します。下の例ではロードインバランスのためバリア同期待ちが多くなっています。

改善前ソース

```
28  void sub(int n, float (* restrict a)[n], float * restrict b,
                                  float s, int * restrict ie){
29            int i, j;
30
31           #pragma omp parallel for schedule(static, 1)
             <<< Loop-information Start >>>
             <<<  [OPTIMIZATION]
             <<<    PREFETCH(HARD) Expected by compiler :
             <<<      (unknown)
             <<< Loop-information  End >>>
32   p         for (j = 0; j < n; j++){
             <<< Loop-information Start >>>
             <<<  [OPTIMIZATION]
             <<<    SIMD(VL: 16)
             <<<    SOFTWARE PIPELINING(IPC: 3.50,
                        ITR: 448, MVE: 7, POL: S)
             <<<    PREFETCH(HARD) Expected by compiler :
             <<<      (unknown)
             <<< Loop-information  End >>>
33   p    2v      for (i = 0; i < ie[j]; i++){
34   p    2v        a[j][i] = a[j][i]*b[i]*s;
35   p    2v      }
36   p          }
37           }
```

評価ループ

最内ループは制御変数 j が奇数のときは1,000,000回転,偶数ときは100,000回転する。



ロードインバランスによるバリア同期待ちが発生

C/C++ 処理量が不規則なループ（OpenMPチューニング）

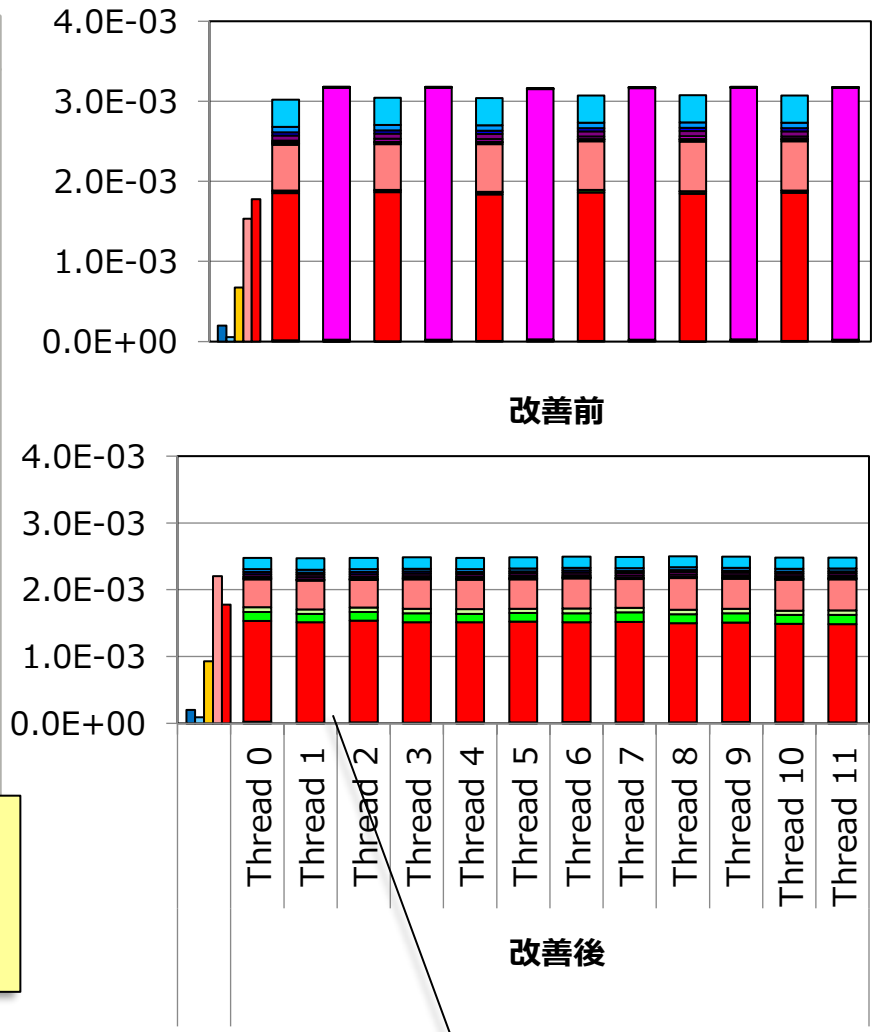
FUJITSU

スケジュール方法をdynamicにすることで、先に処理が終了したスレッドが次の処理を行えるようになり、ロードインバランスが改善しました。

改善後ソース

```
28 void sub(int n, float (* restrict a)[n],
               float * restrict b, float s, int * restrict ie){
29           int i, j;
30
31          #pragma omp parallel for schedule(dynamic, 1)
           <<< Loop-information Start >>>
           <<<  [OPTIMIZATION]
           <<<    PREFETCH(HARD) Expected by compiler :
           <<<      (unknown)
           <<< Loop-information  End >>>
32   p        for (j = 0; j < n; j++){
           <<< Loop-information Start >>>
           <<<  [OPTIMIZATION]
           <<<    SIMD(VL: 16)
           <<<    SOFTWARE PIPELINING(IPC: 3.50, ITR: 448,
                                MVE: 7, POL: S)
           <<<    PREFETCH(HARD) Expected by compiler :
           <<<      (unknown)
           <<< Loop-information  End >>>
33   p    2v     for (i = 0; i < ie[j]; i++){
34   p    2v       a[j][i] = a[j][i]*b[i]*s;
35   p    2v     }
36   p         }
37          }
```

dynamicを指定することで先に処理が終了したスレッドが次の処理を行うようになる。



ロードインバランスが改善された

FUJITSU

適切な並列化次元での並列化

- 適切な並列化次元での並列化（改善前）
- 適切な並列化次元での並列化（最適化制御行チューニング）
- 適切な並列化次元での並列化（翻訳オプションチューニング）
- 適切な並列化次元での並列化（OpenMPソースチューニング）

Fortran 適切な並列化次元での並列化（改善前）

FUJITSU

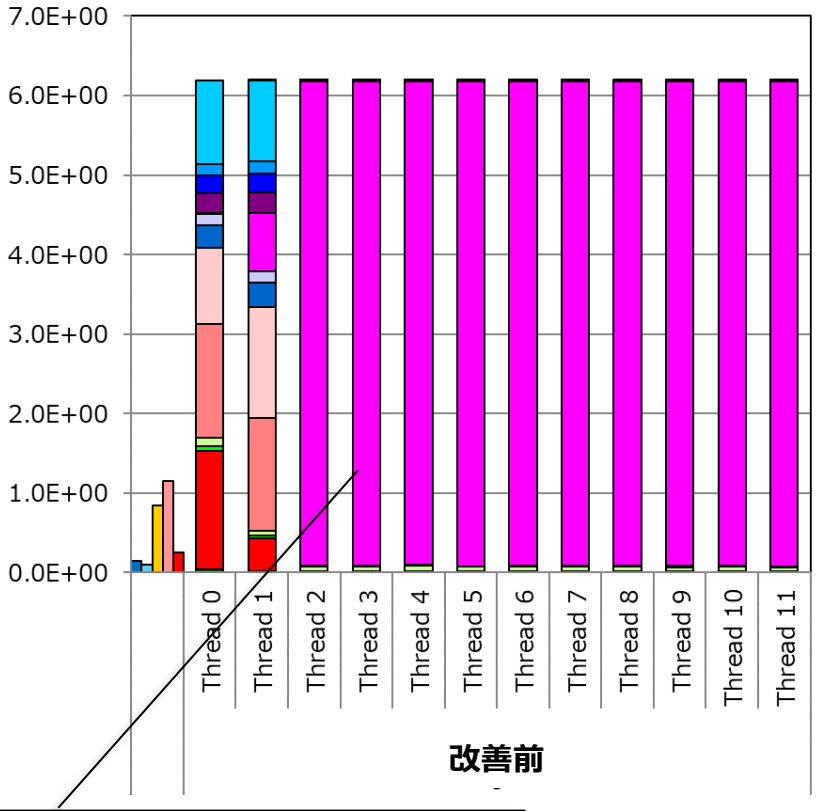
並列化次元のループ繰返し数が少なく翻訳時に繰返し数が不明な場合、繰返し数がスレッド並列数（例は12並列）未満となるケースではロードインバランスが発生します。そのため、バリア同期待ちが多くなっています。

改善前ソース

```
                <<< Loop-information Start >>>
                <<<  [PARALLELIZATION]
                <<<    Standard iteration count: 2
                <<< Loop-information  End >>>
32    1  pp           do k=1,l
                <<< Loop-information Start >>>
                <<<  [OPTIMIZATION]
                <<<    PREFETCH(HARD) Expected by compiler :
                <<<      c, b, a
                <<< Loop-information  End >>>
33    2  p            do j=1,m
                <<< Loop-information Start >>>
                <<<  [OPTIMIZATION]
                <<<    SIMD(VL: 8)
                <<<    SOFTWARE PIPELINING(IPC: 3.25,
                                    ITR: 144, MVE: 4, POL: S)
                <<<    PREFETCH(HARD) Expected by compiler :
                <<<      c, b, a
                <<< Loop-information  End >>>
34    3  p  2v             do i=1,n
35    3  p  2v               a(i,j,k)=b(i,j,k)+c(i,j,k)
36    3  p  2v             enddo
37    2  p              enddo
38    1  p           enddo
```

l=2
m=512
n=256

各スレッドのロードバランスが悪い



並列化次元の k の繰返し数が２回転であるためロードインバランスが発生

Fortran 適切な並列化次元での並列化（ソースチューニング）

FUJITSU

SERIAL指示子、PARALLEL指示子を指定することで、適切な次元で並列化されるようになり、ロードインバランスが改善しました。

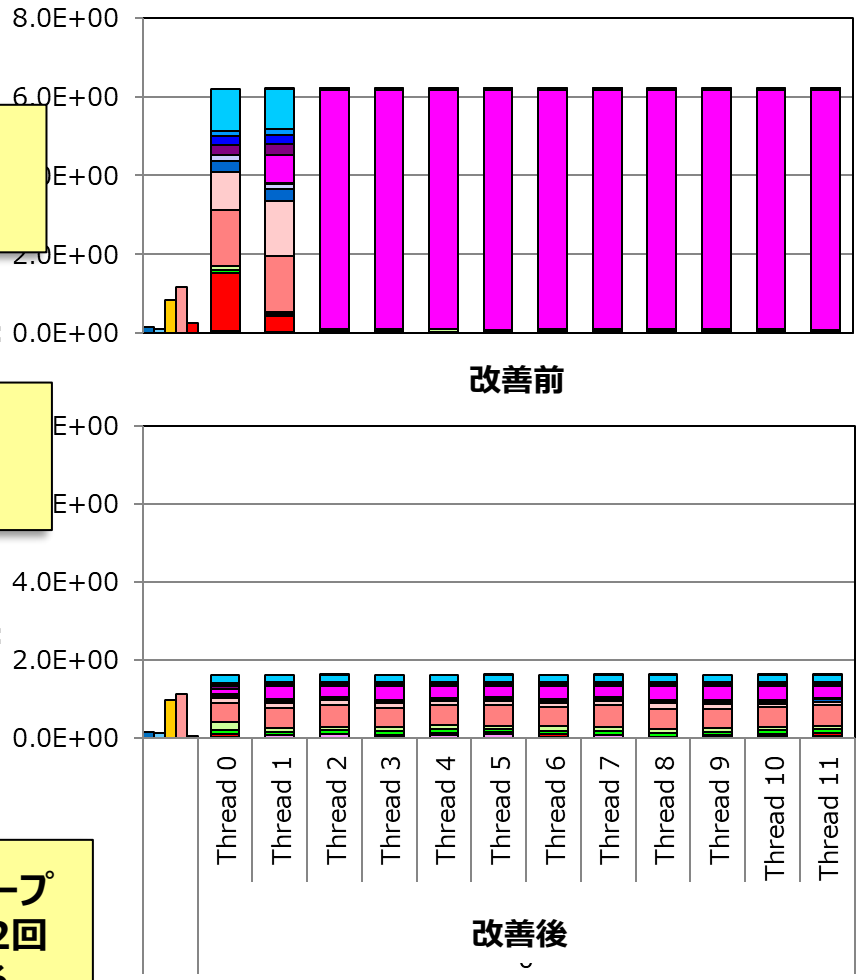
改善後ソース

```
31                !ocl serial
32    1             do k=1,l
33    1            !ocl parallel
                <<< Loop-information Start >>>
                <<<  [PARALLELIZATION]
                <<<    Standard iteration count: 4
                <<<  [OPTIMIZATION]
                <<<    PREFETCH(HARD) Expected by compiler :
                <<<      c, b, a
                <<< Loop-information  End >>>
34    2  pp           do j=1,m
                <<< Loop-information Start >>>
                <<<  [OPTIMIZATION]
                <<<    SIMD(VL: 8)
                <<<    SOFTWARE PIPELINING(IPC: 3.25,
                                   ITR: 144, MVE: 4, POL: S)
                <<<    PREFETCH(HARD) Expected by compiler :
                <<<      c, b, a
                <<< Loop-information  End >>>
35    3   p   2v          do i=1,n
36    3   p   2v            a(i,j,k)=b(i,j,k)+c(i,j,k)
37    3   p   2v          enddo
38    2   p           enddo
39    1             enddo
```

ループスライスを抑止する

l=2
m=512
n=256

並列化次元のループ j の繰返し数512回で並列実行をする



Fortran 適切な並列化次元での並列化（翻訳オプションチューニング）

FUJITSU

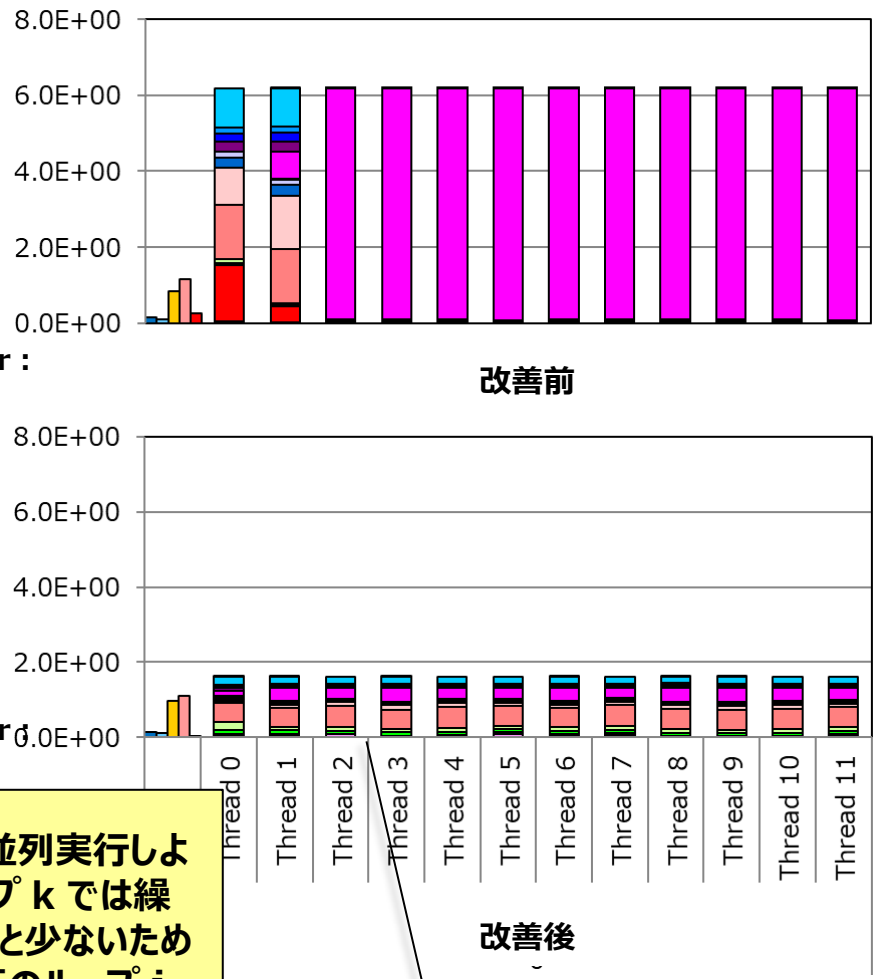
翻訳オプション -Kdynamic_iteration を指定することで、適切な並列化次元が実行時に自動選択されるようになり、ロードインバランスが改善しました。

改善後ソース

```
                <<< Loop-information Start >>>
                <<<  [PARALLELIZATION]
                <<<    Standard iteration count: 2
                <<< Loop-information  End >>>
31    1  pp           do k=1,l
                <<< Loop-information Start >>>
                <<<  [PARALLELIZATION]
                <<<    Standard iteration count: 4
                <<<  [OPTIMIZATION]
                <<<    PREFETCH(HARD) Expected by compiler :
                <<<      c, b, a
                <<< Loop-information  End
32    2  pp             do j=1,m
                <<< Loop-information Start
                <<<  [PARALLELIZATION]
                <<<    Standard iteration count: 728
                <<<  [OPTIMIZATION]
                <<<    SIMD(VL: 8)
                <<<    SOFTWARE PIPELINING(IPC: 3.25,
                                   ITR: 144, MVE: 4, POL: S)
                <<<    PREFETCH(HARD) Expected by compiler
                <<<      c, b, a
                <<< Loop-information  End
33    3  pp  2v             do i=1,n
34    3   p  2v               a(i,j,k)=b(i,j,k)+
35    3   p  2v             enddo
36    2   p              enddo
37    1   p            enddo
```

l=2
m=512
n=256

外側ループから並列実行しようとするが、ループ k では繰返し数が２回転と少ないため内側の512回転のループ j で並列実行される



ロードインバランスが改善された

Fortran 適切な並列化次元での並列化（OpenMPソースチューニング）

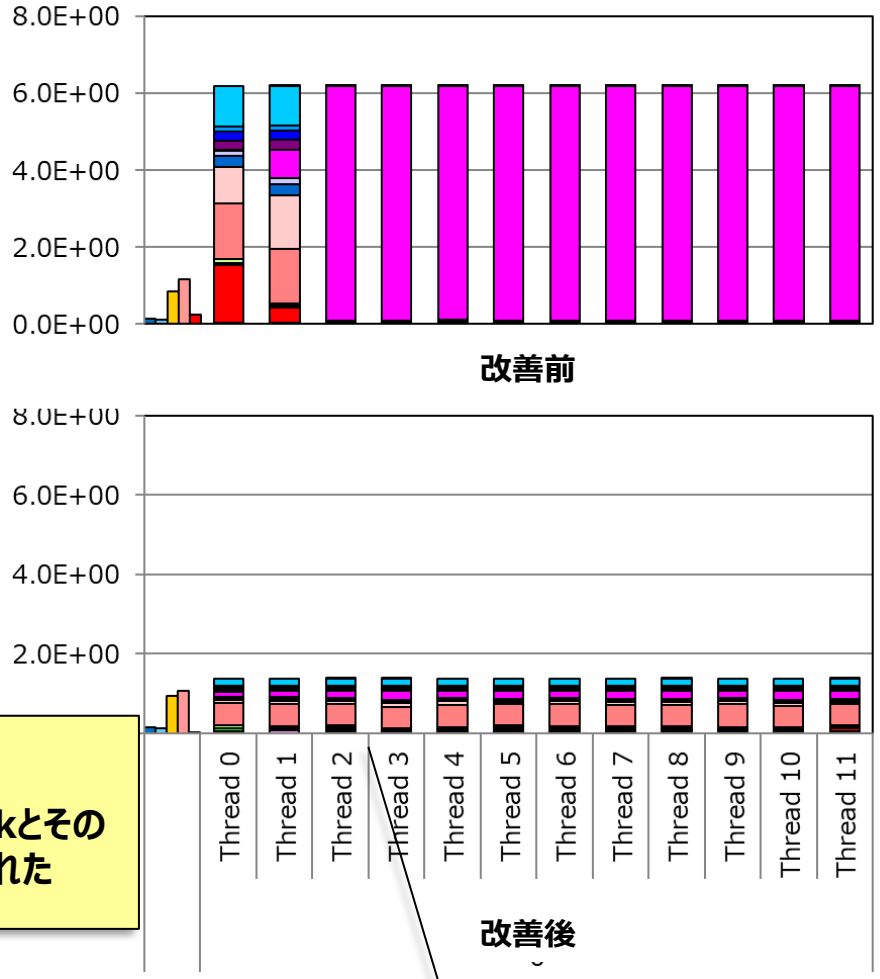
FUJITSU

OpenMPのCOLLAPSE指示節を指定することで、外側のループを一重化します。
その結果、ロードインバランスが改善しました。

改善後ソース

```
32              !$omp parallel do private(k,j,i) collapse(2)
        <<< Loop-information Start >>>
        <<<  [OPTIMIZATION]
        <<<    PREFETCH(HARD) Expected by compiler :
        <<<      c, b, a
        <<< Loop-information  End >>>
33    1   p           do k=1,l
34    2   p             do j=1,m
        <<< Loop-information Start >>>
        <<<  [OPTIMIZATION]
        <<<    SIMD(VL: 8)
        <<<    SOFTWARE PIPELINING(IPC: 3.25,
                                   ITR: 144, MVE: 4, POL: S)
        <<<    PREFETCH(HARD) Expected by compiler :
        <<<      c, b, a
        <<< Loop-information  End >>>
35    3   p   2v              do i=1,n
36    3   p   2v                a(i,j,k)=b(i,j,k)+c(i,j,k)
37    3   p   2v              enddo
38    2   p                 enddo
39    1   p               enddo
40                    !$omp end pa
```

l=2
m=512
n=256



COLLAPSE指示節により
繰り返し数の少ないループkとその
内側のループjが一重化された

ロードインバランスが改善された

■ 注意事項
SIMD化軸(最内ループ)はCOLLAPSE
しないように注意してください。

C/C++ 適切な並列化次元での並列化（改善前）

FUJITSU

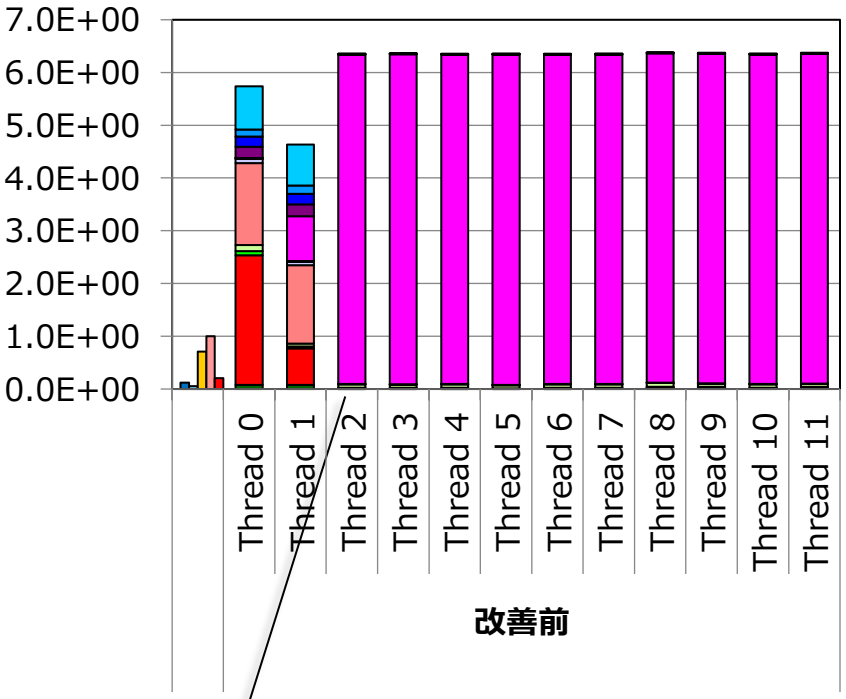
並列化次元のループ繰返し数が少なく翻訳時に繰返し数が不明な場合、繰返し数がスレッド並列数（例は12並列）未満となるケースではロードインバランスが発生します。そのため、バリア同期待ちが多くなっています。

改善前ソース

```
42          #pragma omp parallel for
43   p        for (k = 0; k < l; k++){
             <<< Loop-information Start >>>
             <<<  [OPTIMIZATION]
             <<<    PREFETCH(HARD) Expected by compiler :
             <<<      (unknown)
             <<< Loop-information  End >>>
44   p          for (j = 0; j < m; j++){
             <<< Loop-information Start >>>
             <<<  [OPTIMIZATION]
             <<<    SIMD(VL: 8)
             <<<    SOFTWARE PIPELINING(IPC: 3.00,
                      ITR: 144, MVE: 5, POL: S)
             <<<    PREFETCH(HARD) Expected by compiler :
             <<<      (unknown)
             <<< Loop-information  End >>>
45   p    2v       for (i = 0; i < n; i++){
46   p    2v         a[k][j][i] = b[k][j][i] + c[k][j][i];
47   p    2v       }
48   p          }
49   p        }
```

l=2
m=512
n=256

各スレッドのロードバランスが悪い



並列化次元の k の繰返し数が２回転であるためロードインバランスが発生

適切な並列化次元での並列化（ソースチューニング）

FUJITSU

parallel for指示子やparallel指示子を指定することで、適切な次元で並列化されるようになり、ロードインバランスが改善しました。

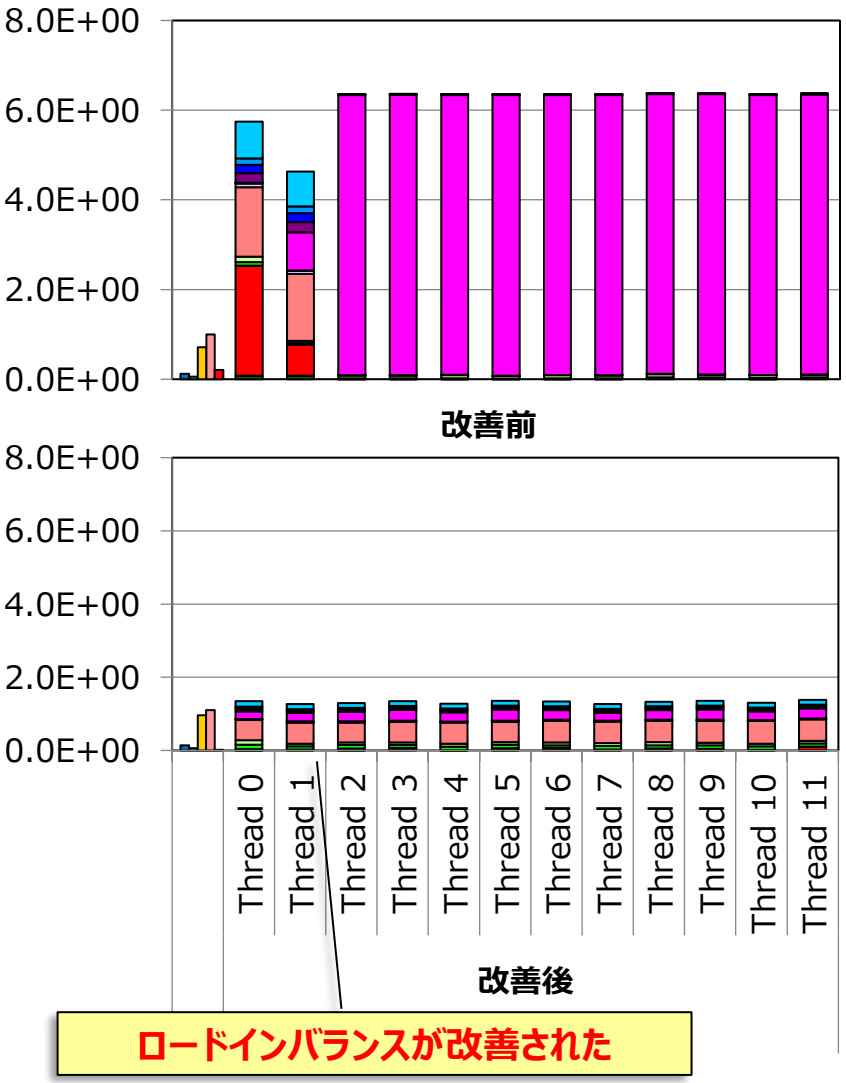
改善後ソース

```
42          #pragma omp parallel private(i,j,k)
43          {
44            for (k = 0; k < l; k++){
45          #pragma omp for
           <<< Loop-information Start >>>
           <<<  [OPTIMIZATION]
           <<<    PREFETCH(HARD) Expe
           <<<      (unknown)
           <<< Loop-information  End >>
46   p          for (j = 0; j < m; j++){
           <<< Loop-information Start >>>
           <<<  [OPTIMIZATION]
           <<<    SIMD(VL: 8)
           <<<    SOFTWARE PIPELINING(IPC: 3.00,
                         ITR: 144, MVE: 5, POL: S)
           <<<    PREFETCH(HARD) Expected by compiler :
           <<<      (unknown)
           <<< Loop-information  End >>>
47   p    2v       for (i = 0; i < n; i++){
48   p    2v         a[k][j][i] = b[k][j][i] + c[k][j][i];
49   p    2v       }
50   p          }
51             }
```

l=2
m=512
n=256

ループスライスを抑止する

並列化次元のループ j の繰返し数512回で並列実行をする



ロードインバランスが改善された

C/C++ 適切な並列化次元での並列化（翻訳オプションチューニング）

FUJITSU

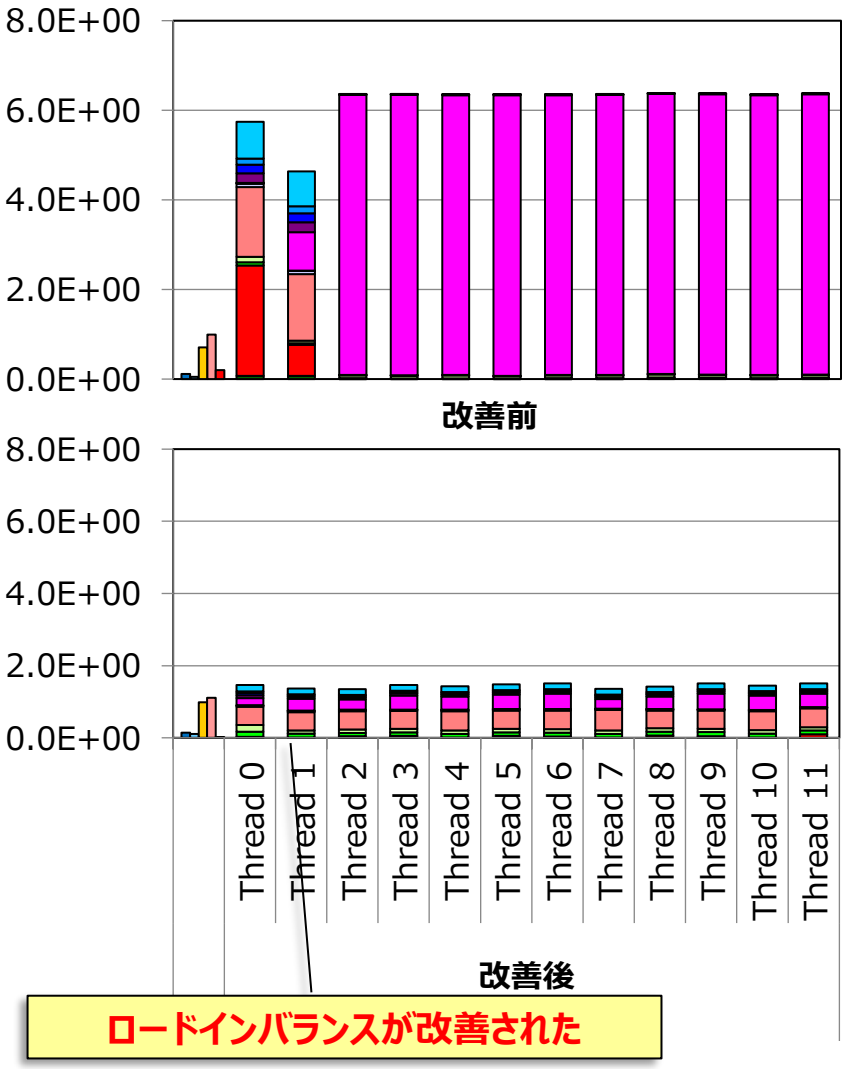
翻訳オプション -Kdynamic_iteration を指定することで、適切な並列化次元が実行時に自動選択されるようになり、ロードインバランスが改善しました。clangモードには-Kdynamic_iterationがないため、本手法によるチューニングはできません。

改善後ソース

```
                <<< Loop-information Start >>>
                <<<  [PARALLELIZATION]
                <<<    Standard iteration count: 3
                <<< Loop-information  End >>>
42  pp            for (k = 0; k < n3; k++){
                <<< Loop-information Start >>>
                <<<  [PARALLELIZATION]
                <<<    Standard iteration count: 37
                <<<  [OPTIMIZATION]
                <<<    PREFETCH(HARD) Expected by compiler :
                <<<      (unknown)
                <<< Loop-information  End >>>
43  pp              for (j = 0; j < n2; j++){
                <<< Loop-information Start >>>
                <<<  [PARALLELIZATION]
                <<<    Standard iteration count: 552
                <<<  [OPTIMIZATION]
                <<<    SIMD(VL: 8)
                <<<    SOFTWARE PIPELINING(IPC: 3.25
                           ITR: 1
                <<<    PREFETCH
                <<<      (unknow
                <<< Loop-inform
44  pp    2v          for (i = 0;
45  p     2v            a[k][j][i]
46  p     2v          }
47  p               }
48  p             }
```

I=2
m=512
n=256

外側ループから並列実行しようとするが、ループ k では繰返し数が２回転と少ないため内側の512回転のループ j で並列実行される



適切な並列化次元での並列化（OpenMP ソースチューニング）

FUJITSU

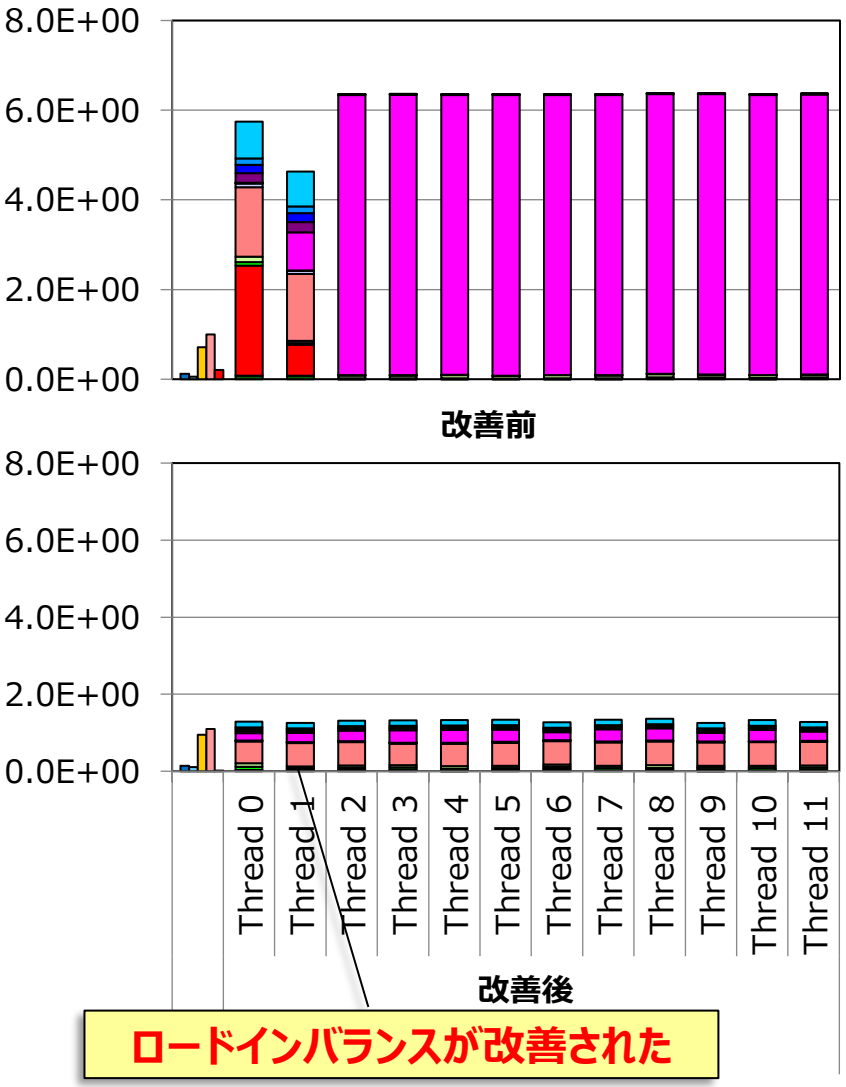
OpenMPのcollapse指示節を指定することで、外側のループを一重化します。
その結果、ロードインバランスが改善しました。

改善後ソース

```
42          #pragma omp parallel for private(i,j,k) collapse(2)
            <<< Loop-information Start >>>
            <<<  [OPTIMIZATION]
            <<<    PREFETCH(HARD) Expected by compiler :
            <<<      (unknown)
            <<< Loop-information  End >>>
43   p        for (k = 0; k < l; k++){
44   p          for (j = 0; j < m; j++){
            <<< Loop-information Start >>>
            <<<  [OPTIMIZATION]
            <<<    SIMD(VL: 8)
            <<<    SOFTWARE PIPELINING(IPC: 3.00,
                        ITR: 144, MVE: 5, POL: S)
            <<<    PREFETCH(HARD) Expected by compiler :
            <<<      (unknown)
            <<< Loop-information  End >>>
45   p    2v       for (i = 0; i < n; i++){
46   p    2v         a[k][j][i] = b[k][j][i] + c[k][j][i];
47   p    2v       }
48   p          }
49   p        }
```

l=2
m=512
n=256

COLLAPSE指示節により
繰り返し数の少ないループkとその
内側のループjが一重化された



ロードインバランスが改善された

■ 注意事項
SIMD化軸(最内ループ)はCOLLAPSE
しないように注意してください。

ラージページの設定による実行効率の改善

- ラージページページングポリシーの指定
- ロックタイプの変更

FUJITSU

ラージページページングポリシーの指定

- ラージページページングポリシーについて
- ラージページのページングポリシー(demand)の効果

ラージページのページングポリシーについて

FUJITSU

- XOS_MMM_L_PAGING_POLICY=prepage:demand:prepage
  - スレッド並列で複数CMGにて走行する時は、以下のメモリアロケーションとなる。
    - プリページングでは、ロードモジュール起動時に、CMG0からデータが載る。
    - デマンドページングでは、初回アクセス時に実行CMGにデータが載る。
  - 複数CMGをまたぐときはデマンドページングを推奨します。

環境変数名	指定値 (_付はdefault)	説明
XOS_MMM_L_PAGING_POLICY	[demand | prepage]: [demand | prepage]: [demand | prepage]	各メモリ領域のページング方式(ページの割り当て契機)を選択する設定です。 「demand」はデマンドページング方式、「prepage」はプリページング方式を意味します。本変数はコロン(:)区切りで3つのメモリ領域のページング方式を指定します。 第1指定は、静的データの.bss領域です。(静的データの.data領域はページング方式指定の対象外で常にprepageとなります。) 第2指定は、スタック領域およびスレッドスタック領域です。 第3指定は、動的メモリ確保領域です。 指定値以外の値を指定した場合は、「prepage:demand:prepage」を指定したものとみなします。

ラージページのページングポリシー(demand)の効果

FUJITSU

プリページングではCMG0からデータが載るため、48スレッドのストリームの性能がでません。

デマンドページングに変更することで実行CMGにデータが載り、性能が大幅に向上しました。

ソース

```
Subroutine sub(n,iter,x1,x2,y1)
  real(8) :: x1(n), x2(n), y1(n),c0
  integer n,i,k
  c0=2.0
  call fapp_start("sub",0,0)
  do k=1,iter
  !$omp parallel do
    do i=1,n
      y1(i) = x1(i) + c0 * x2(i)
    end do
  enddo
:
  parameter(N=45000000,ITER=100)
  real*8 x1(N),x2(N),y1(N)
  call init(N,ITER,x1,x2,y1)
  call sub(N,ITER,x1,x2,y1)
```

	Memory throughput (GB/s)
prepage指定(default)	93GB/s
demand指定	804GB/s

翻訳オプション：-Kfast,openmp
-Kprefetch_sequential=soft -Kprefetch_line=9
-Kprefetch_line_L2=70 -Kzfill=18

ストリーム(データサイズは約1GB)

FUJITSU

ロックタイプの変更

- ロックタイプ(XOS_MMM_L_ARENA_LOCK_TYPE)とは
- ロックタイプの変更による効果

ロックタイプ(XOS_MMM_L_ARENA_LOCK_TYPE)とは

FUJITSU

- XOS_MMM_L_ARENA_LOCK_TYPE
  - メモリ割り当てポリシーに関する設定です。
  - 「0」はメモリ獲得性能優先。パラレルリージョン内でmallocしている場合に推奨します。
    (メモリの獲得/開放がスレッド毎に独立したメモリプールから行われるようになり、デフォルト設定よりも排他制御のコストが減って性能改善する場合があります)
  - 「1」はメモリ使用効率優先 (デフォルト)。メモリ使用量が多い場合に推奨します。

ロックタイプの変更による効果

FUJITSU

XOS_MMM_L_ARENA_LOCK_TYPE=0を指定することで、mallocの性能が向上しました。(実行時間は0.56秒から0.35秒に1.60倍の性能向上)

ソース

```
subroutine sub(n,m,iter,x1,x2,y2)
  integer(8) :: pZ1(iter)
  real(8) :: x1(n), x2(n), y2(n,m),c0
  c0=2.0
  !$omp parallel do shared(n,m,iter,x1,x2,c0,y2) private(pZ1,i,j,k) default(none)
  do k=1,m
    do j=1,iter
      pZ1(j) = malloc(8 * n)
    end do

    do i=1,n
      y2(i,k) = x1(i) + c0 * x2(i)
    end do

    do j=1,iter
      call free( pZ1(j))
    end do
  end do
end subroutine sub

program main
  parameter(N=1048512,ITER=80)
  real*8 x1(N),x2(N),y2(N,12)
  call sub(N,12,ITER,x1,x2,y2)
end program main
```

FUJITSU

メモリ使用量の削減

- OMP SINGLE からOMP MASTERへの書き換え

OMP SINGLE から OMP MASTER への書き換え

FUJITSU

- 書き換えによるメモリ使用量の削減
  - omp singleをomp masterとbarrierに書き換えすることで、実行スレッドを固定させ、メモリ領域の冗長使用を抑止します。

改善前ソース	改善後ソース
!$omp parallel **!$omp single** **call loop(a,b,alpha,max);** **!$omp end single** **!$omp end parallel**	**!$omp parallel** **!$omp master** **call loop(a,b,alpha,max);** **!$omp end master** **!$omp barrier** **!$omp end parallel**

更新履歴

■ 更新履歴

版数	発行月	変更理由及び内容
初版(1.0版)	2020/9	・新規作成
1.3版	2021/3	・記事見直しによる誤字や表現の修正
1.4版	2021/8	・ソフトウェア版数アップによる差分修正、および、記事見直しによる誤字や表現の修正 ・「アンロールアンドジャムとは」ページの追加 ・「OMP SINGLE から OMP MASTER への書き換え」ページの追加
2.0版	2022/5	・C/C++のチューニングを追加 ・記事見直しによる誤字や表現の修正 ・フォーマットの変更
2.1版	2022/7	・記事見直しによる誤字や表現の修正
2.2版	2023/3	・記事見直しによる誤字や表現の修正

FUJITSU