

アプリケーションのタイプ別 CPU性能チューニング

第1.1版

理化学研究所 計算科学研究センター
運用技術部門
2021年12月1日

©理化学研究所 計算科学研究センター
本マニュアルに記載されている内容の無断転載・複製を禁じます

更新履歴

版数	更新内容	更新日	更新者
0.9	FS2020 プロジェクトにおいて初版の全体を作成	2021/3/30	南一生
1.0	以降の管理を運用技術部門へ移管	2021/4/12	三上和徳
1.1	英語版作成に伴って改題	2021/12/1	三上和徳

Contents

1 アプリケーションのタイプ-性能の観点から-	5
1.1 要求 B/F 値とアプリケーション	5
1.2 メモリ・L2 キャッシュ・演算器を考慮した性能モデル	5
1.3 CPU 性能解析レポート(ビギー時間)	6
1.4 ビギー時間から見たアプリケーションのタイプ	6
1.4.1 理想的にチューニングされたオンメモリなアプリケーション	6
1.4.2 理想的にチューニングされたオンキヤッシュなアプリケーション(1)	7
1.4.3 理想的にチューニングされたオンキヤッシュなアプリケーション(2)	7
1.4.4 理想的にチューニングされたオンキヤッシュなアプリケーション(3)	7
1.4.5 十分にチューニングされていないオンメモリ or オンキヤッシュなアプリケーション	8
1.5 CPU 単体性能からみたアプリケーションの分類	8
1.6 ビギー時間から見たアプリケーションのタイプと CPU 単体性能からみたアプリケーション分類の対応	9
2 CPU 単体性能チューニングとは	11
3 アプリケーションタイプ毎のビギー時間の実例	13
3.1 メモリ律速タイプ	13
3.2 L2 キャッシュ律速タイプ	15
3.3 演算器律速タイプ	17
4 「富岳」CPU 単体性能向上のための基本的考え方	21
4.1 「富岳」CPU 性能向上の基本戦略	21
4.2 インテルとのアウトオブオーダー資源による比較	22
5 「富岳」CPU 性能チューニング技術	23
5.1 CPU 性能解析レポートの見方	23
5.2 タイプ毎のチューニング技術(1)	23
5.2.1 SWPL の促進とキャッシュ効率向上-ループ分割&ループ融合&ブロッキング-	24
5.2.2 自動ループ分割機能の適用(1)	28
5.2.3 自動ループ分割機能の適用(2)	30
5.2.4 SWPL の促進とキャッシュ効率向上-ループ融合&ブロッキング&ループ分割&プリフェッチ-	30
5.2.5 プリフェッチ指示行の追加	32
5.2.6 multiple structure load/store 命令の利用	34
5.2.7 SWP オプションの利用	37
5.2.8 本節のまとめ	39
5.3 タイプ毎のチューニング技術(2)	40
5.3.1 リストアクセスに対する節点のオーダリング	40
5.3.2 ループのリロールによる SIMD 化	44
5.3.3 ストア側のリストアクセスの削減-要素ループから節点ループへの変更-	45
5.3.4 リストアクセスに対するプリフェッチ指示行の追加	49
5.4 タイプ毎のチューニング技術(3)	51
5.4.1 SIMD 長の拡大	51
5.4.2 レジスタ数の削減-ループ間の最適化の抑止-	54
5.4.3 命令スケジューリングの改善	55

5.4.4 実行命令数の削減	57
5.4.5 コンパイラ最適化強化機能の利用	60
5.4.6 コンパイラ最適化の補完-ポインタ型変数からサブルーチン引数への変更-	60
5.4.7 コンパイラ最適化の補完-ポインタ配列の contiguous 属性付加-	60
5.4.8 スレッド並列の dynamic 分割	63
5.4.9 配列インデックス計算の削除	63
5.4.10 SIMD の無効化	66
5.4.11 ACLE を使用した並び替え	68
5.4.12 マニュアルスケジューリング	70
5.4.13 命令数の削減 -SIMD 縮約の削減-	72
5.5 タイプ毎のチューニング技術 (4)	74
5.5.1 データアクセスの連續化 (1)	74
5.5.2 データアクセスの連續化 (2)	76
5.5.3 データアクセスの連續化 (3)	76
5.6 タイプ毎のチューニング技術 (5)	79
5.6.1 過剰 SFI の回避	79
5.6.2 過剰 SFI の回避-gather load のまとめ上げ処理に伴うもの	82

Chapter 1

アプリケーションのタイプ-性能の観点から-

1.1 要求 B/F 値とアプリケーション

CPU 単体性能の面から見るとアプリケーションは、大きく 2 つのタイプに分類できる。一つは、アプリケーションの浮動小数点演算数 (Flop) に比べ、データ転送要求 (Byte) が小さいタイプである。このタイプの計算を要求 B/F 値が小さい計算という。例えば行列行列積計算の例では、原理的には、B/F 値は移動量/演算数を要素数で表すと $1/N$ となる。一般には移動量は Byte の値で表すので、倍精度計算であれば移動量に 8Byte を乗じるので、 $8/N$ となり、原理的には N が大きい程、B/F 値は小さな値となる。

行列行列積のように要求 B/F 値が小さいアプリケーションは、キャッシュに置いたデータを何回も再利用して演算を行うことで、高い CPU 単体性能が得られる計算である。言い換えれば、オンキャッシュなアプリケーションといえる。

もう一方は、逆にアプリケーションの浮動小数点演算数 (Flop) に比べ、メモリとのデータ転送要求 (Byte) が大きいタイプである。このタイプの計算を要求 B/F 値が大きい計算という。このタイプの計算は、原理的にキャッシュの有効利用が難しく高い CPU 単体性能が得にくい。言い換えれば、オンメモリなアプリケーションといえる。

例えば行列ベクトル積計算の例では、原理的には、B/F 値は移動量/演算数を要素数で表すと、ほぼ $1/2$ となる。一般には移動量は Byte の値で表すので、倍精度計算であれば移動量に 8Byte を乗じるので、 $8/2 = 4$ となり、行行列積に比べると、B/F 値は大きな値となる。

1.2 メモリ・L2 キャッシュ・演算器を考慮した性能モデル

メモリ、L2 キャッシュ、演算に対して、メモリデータ転送量、実効メモリバンド幅、L2 キャッシュデータ転送量、実効 L2 キャッシュバンド幅、プログラム演算量、演算実効性能を、それぞれ M_{data} 、 M_{band} 、 $L2_{\text{data}}$ 、 $L2_{\text{band}}$ 、 N_c 、 P_{peak} とすると、メモリ律速時の実行時間 t_M 、L2 キャッシュ律速時の実行時間 t_{L2} 、演算器律速時の実行時間 t_C は、以下の式で計算できる。この説明を図 1.1 に示す。

$$t_M = M_{\text{data}} / M_{\text{band}} \quad (1.1)$$

$$t_{L2} = L2_{\text{data}} / L2_{\text{band}} \quad (1.2)$$

$$t_C = N_c / P_{\text{peak}} \quad (1.3)$$

したがって、プログラムの実行時間 t_E 、ピーク性能比 C_p は、以下で計算される。

$$t_E = \max\{t_M, t_{L2}, t_C\} \quad (1.4)$$

$$C_p = N_c / (\max\{t_M, t_{L2}, t_C\} \times P_{\text{peak}}) \quad (1.5)$$

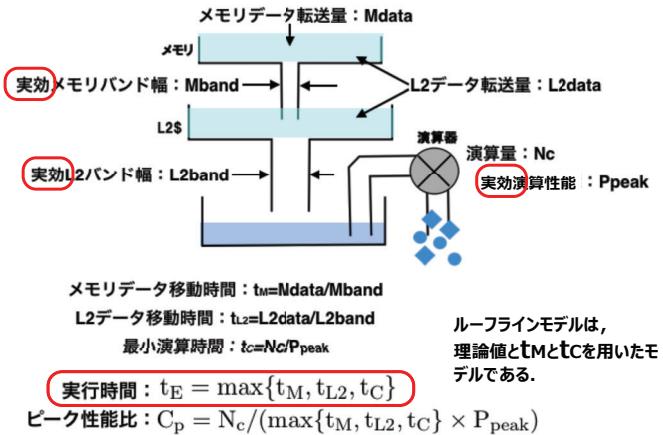


Figure 1.1: メモリ・L2 キャッシュ・演算器を考慮した性能モデル

1.3 CPU 性能解析レポート（ビジー時間）

「富岳」では、図 1.2 に示すような CPU 解析レポートが出力される。1.2 節に示した t_M が、この図の中のメモリビジー時間と、 t_{L2} が、L2 キャッシュビジー時間と、 t_C が、浮動小数点演算器時間と、 t_E が、経過時間と近い値となっている。そのものではなく、近い値となっている理由については後述する。

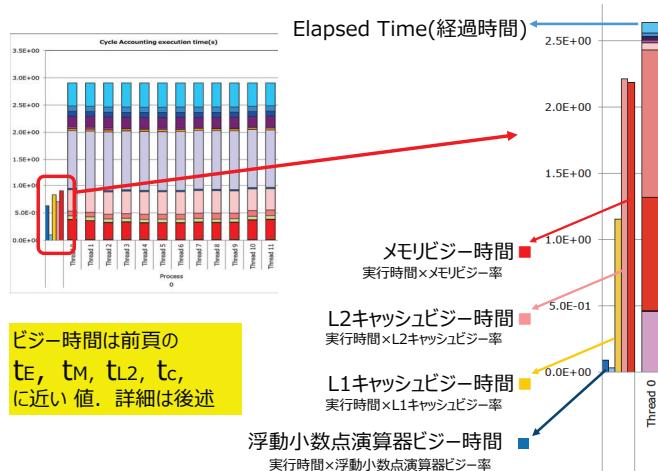


Figure 1.2: CPU 性能解析レポート（ビジー時間）

1.4 ビジー時間から見たアプリケーションのタイプ

ここでは、ビジー時間から見たアプリケーションのタイプ分けを示す。

1.4.1 理想的にチューニングされたオンメモリなアプリケーション

ビジー時間から見たアプリケーションのタイプの1つ目は、理想的にチューニングされたオンメモリなアプリケーションである。

このようなアプリケーションでは、理想的には、実効的なメモリバンド幅で決まるメモリビジー時間が経過時間と一致し、実効メモリバンド幅を使い切っている状態であるといえる。また L2 キャッシュ/演算器は余裕がある状態であるといえる。

このようなアプリケーションの状況を図 1.3 に示す。

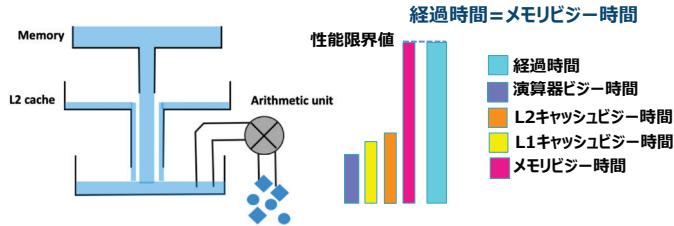


Figure 1.3: 理想的にチューニングされたオンメモリなアプリケーション

1.4.2 理想的にチューニングされたオンキャッシュなアプリケーション (1)

ビギー時間から見たアプリケーションのタイプの2つ目は、理想的にチューニングされたオン L2 キャッシュなアプリケーションである。

このようなアプリケーションでは、理想的には、実効的な L2 キャッシュバンド幅で決まる L2 キャッシュビギー時間が経過時間と一致し、実効 L2 キャッシュバンド幅を使い切っている状態であるといえる。またメモリ/演算器は余裕がある状態であるといえる。

このようなアプリケーションの状況を図 1.4 に示す。

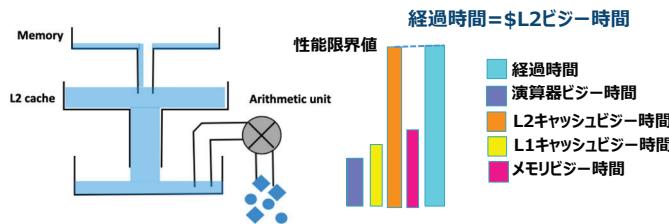


Figure 1.4: 理想的にチューニングされたオンキャッシュなアプリケーション (1)

1.4.3 理想的にチューニングされたオンキャッシュなアプリケーション (2)

ビギー時間から見たアプリケーションのタイプの3つ目は、理想的にチューニングされたオンキャッシュかつ演算器パワンドなアプリケーションである。

このようなアプリケーションでは、理想的には、実効的な演算器性能で決まる演算器ビギー時間が経過時間と一致し、実効的な演算器性能が出ている状態であるといえる。またメモリ、L2 キャッシュは余裕がある状態であるといえる。

このようなアプリケーションの状況を図 1.5 に示す。

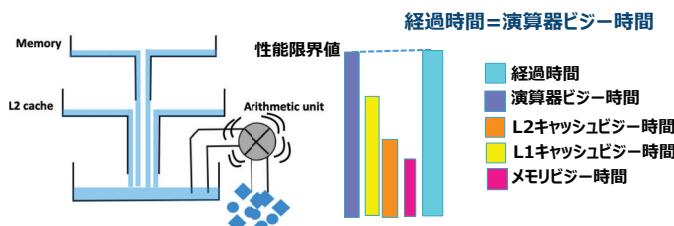


Figure 1.5: 理想的にチューニングされたオンキャッシュなアプリケーション (2)

1.4.4 理想的にチューニングされたオンキャッシュなアプリケーション (3)

ビギー時間から見たアプリケーションのタイプの4つ目は、理想的にチューニングされた L1 オンキャッシュなアプリケーションである。

このようなアプリケーションでは、理想的には、L1D キャッシュビギー時間が経過時間と一致し、L1D キャッシュ性能を使い切っている状態であるといえる。L1D キャッシュビギー時間は L1D キャッシュの実効的なバンド幅のみでは決まるわけではなく、バンド幅以外の動作にも大きく依存している。したがってメモリや L2 キャッシュのように、1.2 節に示した方法で実効バンド幅をベースに性能のモデル化ができない。しかし理想的にチューニングされたアプリは、L1D キャッシュのビギー時間と経過時間は一致し、メモリ、L2 キャッシュ、演算器は余裕がある状態となる。

このようなアプリケーションの状況を図 1.6 に示す。

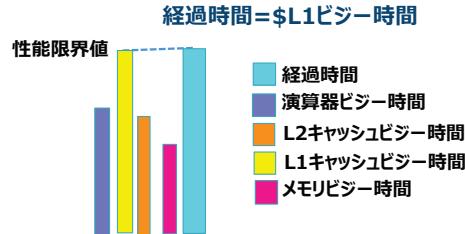


Figure 1.6: 理想的にチューニングされたオンキャッシュなアプリケーション (3)

1.4.5 十分にチューニングされていないオンメモリ or オンキャッシュなアプリケーション

ビギー時間から見たアプリケーションのタイプの 5 つ目は、理想的にチューニングされていない、オンメモリまたはオンキャッシュなアプリケーションである。

このようなアプリケーションでは、アプリケーションの経過時間は、どのビギー時間よりも大きくなり、どのビギー時間とも一致しない。つまりハードウェアの限界性能まで達していないこととなり、メモリ、L2 キャッシュ、L1D キャッシュ、演算器は共に余裕がある状態である。

このようなアプリケーションの状況を図 1.7 に示す。

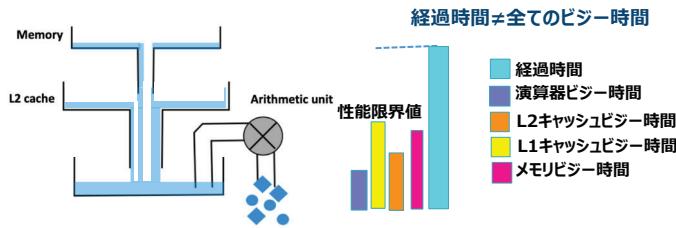


Figure 1.7: 十分にチューニングされていないオンメモリ or オンキャッシュなアプリケーション

1.5 CPU 単体性能からみたアプリケーションの分類

1.1 節では CPU 単体性能の面から見ると、大まかにはアプリケーションを、要求 B/F 値が小さいタイプと要求 B/F 値が大きいタイプの 2 つに分類できることを述べた。この見方をもう少し進め、表 1.1 のように 6 タイプへのアプリケーションの分類について示す。

要求 B/F 値が小さいタイプの計算が 1 つ目から 4 つ目のタイプである。要求 B/F 値が小さいタイプの計算でも、DGEMM ライブラリを使用できるか、手動でキャッシュブロッキングするかによって性能は大きく異なる。またキャッシュブロッキングできるアプリケーションでも、もっともシンプルなデータ構造・ループ構造を持つものと、配列要素の参照に整数値の配列を使用するリストベクトルの使用等、もう少し複雑なデータ構造を持つもので性能が異なる。さらに複雑なループ構造を持つものは高い性能が得られない場合が多い。この考えに基づき、要求 B/F 値が小さいタイプの計算を 1 つ目から 4 つ目のタイプに分類した。

1 つ目のタイプが、行列行列積に書き換え可能なアプリケーションである。このタイプは、原理的に n^2 のデータをメモリからロードすれば、 n^3 の計算ができるため、B/F 値が小さいタイプの計算である。このタイプの計算の例としては、密度汎関数理論 (DFT) に基づく第一原理量子計算のアプリケーション等がある。2 つ目のタイプ

は、特殊な高次精度のステンシル計算¹のような例が該当し、キャッシュの有効利用が図れるため要求 B/F 値が小さく、かつループボディがシンプルな計算である。このようなタイプの計算は良い性能が得られるが、残念ながら例が少ない。3つ目のタイプは、要求 B/F 値が小さいタイプの計算の中で行列行列積に書き換え可能ではないが、キャッシュロックングが可能なタイプの計算である。分子動力学のクーロン相互作用計算や重力多体問題の重力相互作用計算等が該当する。いずれも n 個の粒子のデータをロードしキャッシュロックングすることで n の 2 乗の計算ができるため、要求 B/F 値が小さい計算となる。3つ目のタイプは、粒子のアクセスにリストデータを使用する場合が多く、ループボディ²も多少複雑になる。4つ目のタイプは、要求 B/F 値は小さいがループボディが複雑な計算である。気象計算の中には、流体の運動を計算する力学過程と雲等の現象を計算する物理過程があるが、このタイプの計算には物理過程が対応する。メモリからロードしてきた少ないデータを使用して複雑な計算を行うため、オンキャッシュの計算となるが、ループボディは長く複雑になりがちである。またプラズマの計算等に使用される PIC 法³の計算もこのタイプの計算である。PIC 法の計算も、粒子の周囲のメッシュデータはオンキャッシュとなるが、粒子のアクセスにリストデータを使用する場合が多く、プログラムの複雑化を招いている。また計算ループのボディは長くなる傾向がある。4つ目のタイプについては、オンキャッシュとなっているので高い性能を期待したいところであるが、プログラムの複雑さ故に期待した性能が得られない場合が多い。

要求 B/F 値が大きいタイプの計算が 5 つ目と 6 つ目のタイプである。同じ要求 B/F 値が大きいタイプの計算でも、リストによる不連続なアクセスを行うかどうかによって性能は大きく異なる。この考えに基づき、要求 B/F 値が大きいタイプの計算を 5 つ目と 6 つ目のタイプに分類した。

5 つ目のタイプは、要求 B/F 値が大きいタイプの計算である。通常のステンシル計算にこのタイプの計算が多く、先に述べた気象計算の中の力学過程や流体計算、地震の計算等、例は多い。6 つ目のタイプは、要求 B/F 値が大きいタイプの計算であり、かつリストアクセスを使用している計算である。有限要素法を用いた構造解析や流体計算等、エンジニアリングの分野に多いタイプの計算である。リストアクセスは、1 要素ごとのランダムアクセスが発生するため、現代のスカラー計算機のアーキテクチャでは、不得意な計算である。

以上の 6 つのタイプの計算は一般的には、タイプ 1 からタイプ 6 に進むに従って CPU 単体性能は出しにくいと言える。ただしタイプ 2 とタイプ 3 については順位をつけがたい。

Table 1.1: CPU 単体性能からみたアプリケーションの分類

番号	単体性能上の分類	アプリケーション例
1	行列行列積に書き換え可能	第一原理 (DFT) 量子計算等
2	要求 B/F 値が小さくループボディがシンプル	高次なステンシル計算等
3	キャッシュロックング可能	分子動力学・重力多体問題等
4	要求 B/F 値が小さいがループボディが複雑	プラズマ・気象の物理過程 量子化学計算等
5	要求 B/F 値が大きい	気象の力学過程・流体・ 地震・核融合等
6	要求 B/F 値が大きくりストアクセスを使用	有限要素法を用いた構造・ 流体計算等

1.6 ビジー時間から見たアプリケーションのタイプと CPU 単体性能からみたアプリケーション分類の対応

1.4 節で示したビジー時間から見たアプリケーションのタイプと、1.5 節で示した CPU 単体性能からみたアプリケーション分類の関連について示す。

1.4.3 項に示した、理想的にチューニングされたオンキャッシュかつ演算器バウンドなアプリケーションは、CPU 単体性能上の分類では、行列行列積に書き換え可能なアプリケーションに該当する。行列行列積に書き換えられ対象マシンにチューニングされた GEMM ライブラリを使用することにより、対象マシンの最大限の性能が発揮できる。また CPU 単体性能上の分類で、高次なステンシル計算等を例にあげた、要求 B/F 値が小さくループボディがシンプルなアプリケーションが該当する場合もある。先にあげた GEMM ライブラリほどの性能は得られないが、GEMM ライブラリ等を使用しない場合の実効的な演算器の上限程度までの性能が得られる場合がある。このような例について後述する。

1.4.4 項に示した、理想的にチューニングされた L1D オンキャッシュなアプリケーションや、1.4.2 に示した、理想的にチューニングされた L2 オンキャッシュなアプリケーションは、CPU 単体性能上の分類では、分子動力学・重力多体問題等の例をあげた、キャッシュロックング可能なアプリケーションに該当する。理想的にチュー

¹ 差分計算の中で現れる $i, i - 1$ 等の差分用の添字を用いる計算

² ループ内に含まれるコーディング

³ Particle In Cell 法 . 計算格子の中に粒子を配置する計算手法 .

ニングされたオンキヤシュかつ演算器バウンドなアプリケーションまでの性能は得られないが、比較的高い性能が得られるアプリケーションである。また、CPU 単体性能上の分類で、高次なステンシル計算等を例にあげた、要求 B/F 値が小さくループボディがシンプルなアプリケーションが該当する場合もある。

1.4.5 項に示したものの中で、十分にチューニングされていないオンキヤシュなアプリケーションが、CPU 単体性能上の分類では、プラズマ・気象の物理過程・量子化学計算等の例をあげた、要求 B/F 値は小さいがループボディが複雑なアプリケーションに該当する。オンキヤシュであるので、もう少し高い性能を期待するが、プログラムの複雑さ故に経過時間が、どのビギー時間までも達することがなく上限性能まで達しない場合が多い。CPU 単体性能上の分類で分子動力学・重力多体問題等の例をあげた、キャッシュブロッキング可能なアプリケーションが、これに該当する場合も見受けられる。

1.4.1 項に示した、理想的にチューニングされたオンメモリなアプリケーションが、CPU 単体性能上の分類では、気象の力学過程・流体・地震・核融合等の例をあげた、要求 B/F 値が大きいアプリケーションや有限要素法を用いた構造・流体計算等の例をあげた、要求 B/F 値が大きくりストアクセスを使用するアプリケーションに該当する。ここに該当する場合は、チューニング手法が理想的に適用できた場合であり、チューニング手法が理想的に適用できない場合は、1.4.5 項に示したものの中で、十分にチューニングされていないオンメモリなアプリケーションであるといえる。

ここで示した対応を図 1.8 に示す。

性能の高い順に並べると(あくまで目安です)

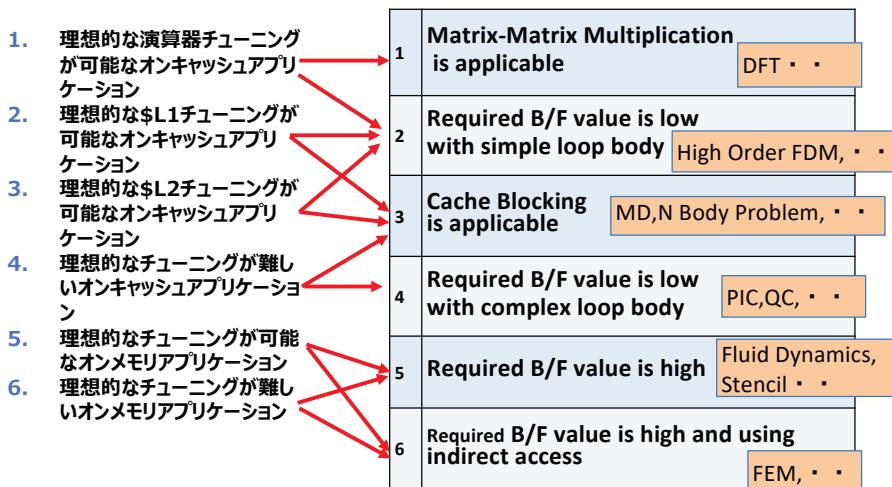


Figure 1.8: ビギー時間から見たタイプと CPU 単体性能からみたアプリケーション分類の対応

Chapter 2

CPU 単体性能チューニングとは

本章では、今までのビギー時間の議論をもとに、CPU 単体性能チューニングとは何か？について示す。

CPU 単体性能チューニングとは何かとは、1.5 節に示したアプリケーション分類、1.4 節に示したアプリケーションのビギー時間のタイプ、1.6 節に示したアプリケーション分類とビギー時間のタイプの対応、またアプリケーション性能の測定結果等より、自分のプログラムがどのタイプかを見極めて、そのタイプのビギー時間のグラフが正当かを判断し、問題点を解消しビギー時間の最大値まで経過時間を近づけることといえる。ここで述べたことを図 2.1 に示す。

さらに積極的なチューニング例としては、図 2.2 に例示すように、プログラムを書き換えメモリから L2 キャッシュアクセスへ移動し、メモリアクセスを減らし L2 キャッシュアクセスを増やし、経過時間の上限値を下げつつ、問題点を解消しビギー時間の最大値（性能限界値）まで経過時間を近づける作業ともいえる。

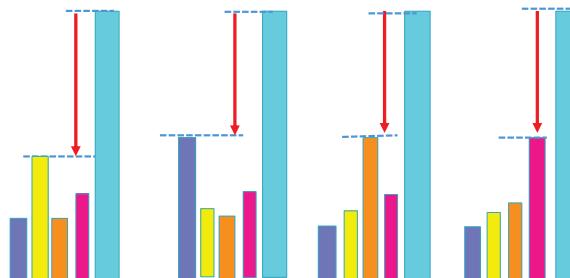


Figure 2.1: CPU 単体性能チューニングとは (1)

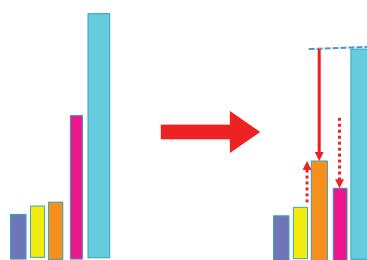


Figure 2.2: CPU 単体性能チューニングとは (2)

Chapter 3

アプリケーションタイプ毎のビジー時間の実例

本節では、1.4 節に示したアプリケーションタイプ毎のビジー時間の実例を示す。

3.1 メモリ律速タイプ

まずメモリ律速タイプとして、1.4 節の中の、理想的にチューニングされたオンメモリなアプリケーションについてビジー時間の実例を示す。ここで用いたテストプログラムを図 3.1 に示す。このテストプログラムの特徴は、オンキヤッシュでなくメモリアクセスすることである。また ZFILL を使用しているためストア時にメモリから L2 キヤッシュへのデータ移動は発生しない。また、このプログラムを 10 回実行し測定している。最内ループの実行（以下の例でも同様）回数を計算すると、 $8000000 * 24 * 10 = 1.92G$ 回となる。メモリのデータ移動量を計算すると、 $1.92G \text{ 回} * 16B = 30.72GB$ となる。実効メモリバンド幅を STREAM triad 測定時の 205GB/sec （ピーク比 80%）を使用し、1.2 節に示したモデルを使い、 t_M を計算すると、見積もり時間： $t_M = 30.72\text{GB} / (205\text{GB/sec}) = 0.15\text{sec}$ となる。

```
integer,parameter :: M= 8000000
integer,parameter :: N= 24
real(8),intent(inout) :: c10(M+1,n)
real(8),intent(in) :: c1(M+1,n)

do j = 1,N
!OCL ZFILL
    do i = 1,M
        c10(i,j) = (z+c1(i,j)*x)* &
                    c1(i,j)+z
    enddo
enddo
```

Figure 3.1: メモリ律速タイプのテストプログラム

このプログラム実行の CPU 性能解析レポートの結果を図 3.2 に示す。実測時間:0.15sec で見積もり通りになっていることが分かる。メモリバンド幅は STREAM triad と同等の、ほぼ 205GB/sec の性能が出ている。CPU 性能解析レポートのメモリビジー時間は本来は 0.15sec となり実行時間と一致するはずだが少なめになる傾向がある。これは、このレポートが理論ピーク値をビジー率 100% としているためであり、実効ピーク値を 100% として計算すれば、ビジー時間は t_M と一致する。L2 キャッシュのビジー時間が非常に大きく出ているが、これは次のセクションで議論する。

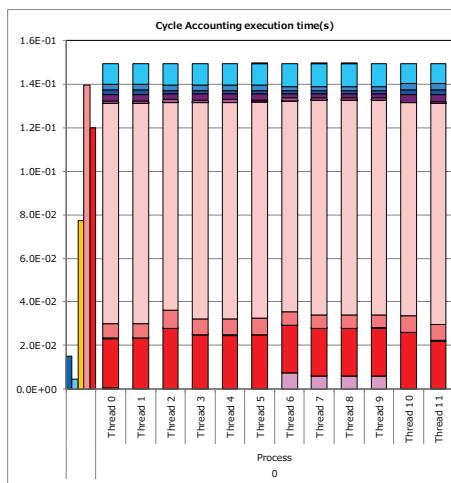


Figure 3.2: メモリ律速タイプの CPU 性能解析レポートの結果

3.2 L2 キャッシュ律速タイプ

次にL2キャッシュ律速タイプとして、1.4節の中の、理想的なL2キャッシュチューニングが可能なオンキャッシュアプリケーションについてビギー時間の実例を示す。ここで用いたテストプログラムを図3.3に示す。このテストプログラムの特徴は、メモリアクセスとL2アクセスが共存するプログラムであることである。またZFILLを使用していないためストア時にメモリからL2キャッシュへのデータ移動は発生する。この図の例では、メモリアクセスは3個、L2アクセスは2個、演算は2個なため、3M-2L2-2Fのプログラムと命名する。このプログラムをN2回実行し測定している。再内ループの実行回数を計算すると、 $3610 * 60 * 168 * 60 = 2.18G$ 回となる。実効L2キャッシュバンド幅は、別途測定した、671GB/sec(ピーク比67.2%)を使用することとする。

```
プログラム : mM-nL2-kF
          do 1.Nloop
          メモリアクセス:m
          L2アクセス:n
          演算数:k
配列宣言(3610,-10:70,168)
N1=3610,N2=60,N3=168
do k = 1,N3
  do j = 1,N2
    do i = 1,N1
      a(i,j,k) = c(i,j-1,k)+c(i,j,k)*c(i,j+1,k)
    enddo 2M      1M   1F  1L2  1F  1L2
    enddo
  enddo
```

Figure 3.3: メモリ・L2キャッシュアクセスのテストプログラム

1.2節に示したモデルを使い、3M-2L2-2Fのプログラムについて t_M を計算する。メモリのデータ移動量は、 $2.18G \text{回} * 24B = 52.32\text{GB}$ となり、メモリベース見積もり時間: $t_M = 52.32\text{GB} / (205\text{GB/sec}) = 0.255\text{sec}$ となる。次にこのプログラムについて t_{L2} を計算する。L2キャッシュアクセスするデータ量は、L2キャッシュアクセスする要素は2個:16Bと、これにメモリアクセスする要素:3個:24Bを加算したものになる。したがってL2キャッシュに対するデータ移動量は、 $2.18G \text{回} * (16 + 24)\text{B} = 87.2\text{GB}$ となり、L2キャッシュベースの見積もり時間: $t_{L2} = 87.2\text{GB} / (671\text{GB/sec}) = 0.13\text{sec}$ となる。1.2節に示したモデルでは、経過時間: t_E は、 t_M と t_{L2} のMAXで計算されるため、メモリベース見積もり時間: t_M が経過時間となる。3M-2L2-2FのプログラムCPU性能解析レポートの結果を図3.4に示す。実測時間:0.255secで見積もり通りになっていることが分かる。CPU性能解析レポートのメモリビギー時間は本来は0.255secとなり実行時間と一致するはずだが少なめになる傾向がある。またL2キャッシュのビギー時間は本来は、ここで見積もりった0.13secになるはずであるが、非常に大きく出ている。これらは前述した傾向と一致している。

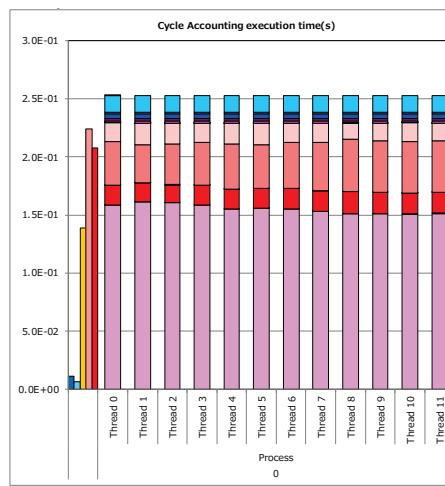


Figure 3.4: 3M-2L2-2FのCPU性能解析レポートの結果

同様に1.2節に示したモデルを使い、3M-12L2-12Fのプログラムについて性能見積もりを行う。メモリベース見積もり時間は、メモリアクセス数が3M-2L2-2Fのケースと同じため、同様の0.255secとなる。次にこのプログラムについて t_{L2} を計算する。L2キャッシュアクセスするデータ量は、L2キャッシュアクセスする要素は12個:96Bと、これにメモリアクセスする要素:3個:24Bを加算したものになる。したがってL2キャッシュに対するデータ移動量は、

2.18G 回*(96+24)B=261.6GB となり、L2 キャッシュベースの見積もり時間: $t_{L2}=261.6\text{GB}/(671\text{GB/sec})=0.39\text{sec}$ となる。1.2 節に示したモデルでは、経過時間: t_E は、 t_M と t_{L2} の MAX で計算されるため、L2 キャッシュベース見積もり時間: t_{L2} が経過時間となる（ここでは演算器律速ベースの時間 t_C は、演算量が小さいため、律速とならないため計算していない）。3M-12L2-12F のプログラム CPU 性能解析レポートの結果を図 3.5 に示す。実測時間:0.384sec で、ほぼ見積もり通りになっていることが分かる。

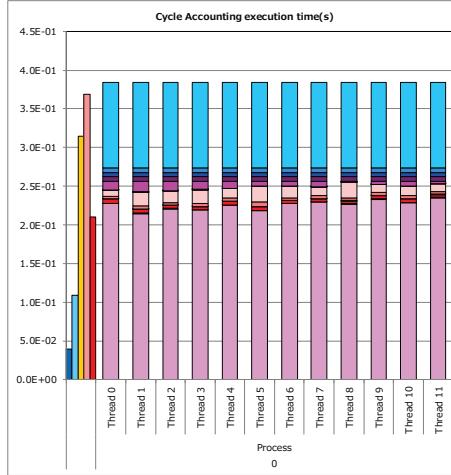


Figure 3.5: 3M-12L2-12F の CPU 性能解析レポートの結果

このケースでは、L2 キャッシュベース見積もり時間:0.390sec とグラフ上の L2 ビジー時間は良く一致している。これは、L2 キャッシュベース見積もり時間と 3 グラフ上の L2 ビジー時間が一致しない 3M-2L2-2F のケースとは大きく違う傾向である。

これは、CPU 性能解析レポートでは、L2 キャッシュパイプラインの数値から L2 ビジー率を算出しており、この数値は、メモリビジー率が高い場合は、メモリ側のリソースの影響が出るためである。具体的には、メモリビジー率が高い場合は、L2 プリフェッчのリトライの数値が含まれてしまい、見かけ上の L2 ビジー率が高くなる。

3.1 の節では、CPU 解析レポートのメモリビジー時間が理論ピーク値をビジー率 100% としているため、1.2 節で示した t_M がメモリビジー時間と同じ概念ではあるが、そのものでないことを述べた。

同様に、ここに示した理由により、1.3 節で示したビジー時間は、1.2 節で示した t_{L2} と同じ概念ではあるが、そのものでない。

3.3 演算器律速タイプ

次に演算器律速タイプとして、1.4節の中の、理想的な演算器チューニングが可能なオンキヤッショアプリケーションについてビギー時間の実例を示す。まず最初にメモリアクセスを一定にし、演算を増やしていくときにどれくらいの性能ができるのかを検証した。テストプログラムを図3.6、図3.7に示す。

Figure 3.6: テストプログラム（限界性能）「京」

「京」ではタイプ1のコーディングで DGEMM ライブラリ並の演算ピーク比:88%の性能が得られていた。しかし「富岳」ではタイプ1では、ピーク性能は得られずタイプ2のコーディングで演算ピーク比で 68%の性能が得られた（「富岳」でも DGEMM ライブラリ等 L1D キャッシュをフル活用するアプリは、もっと高い性能が得られる）。タイプ1のコーディングは命令の連鎖が深く「富岳」では高い性能を得ることが難しく、命令の連鎖を切るためにタイプ2のコーディングの方が高い性能が得られる。また「富岳」では、「京」に比較してレジスタ数が減少していること、演算命令レイテンシーが大きくなったり、SIMD 幅が大きくなったりで対演算ピーク性能が少し低下している。

演算器律速タイプのテストプログラムとして具体的には図 3.8 のプログラムを用いた。これは基本的に 3.2 節で示したテストプログラムで、3M-6L2-80F のケースに相当し、見積もり時間としてはこちらを用いている。

このプログラムを N2 回実行し測定しているため、実行回数を計算すると、 $3610*60*168*60=2.18G$ 回となる。1.2 節に示したモデルを使い、3M-6L2-80F のプログラムについて t_M を計算する。メモリのデータ移動量は、 $2.18G \text{ 回} * 24B = 52.32GB$ となり、メモリベース見積もり時間: $t_M = 52.32GB / (205GB/sec) = 0.255sec$ となる。次にこのプログラムについて t_{L2} を計算する。L2 キャッシュアクセスするデータ量は、L2 キャッシュアクセスする要素は 2 個:48B と、これにメモリアクセスする要素:3 個:24B を加算したものになる。したがって L2 キャッシュに対するデータ移動量は、 $2.18G \text{ 回} * (48+24)B = 156.96GB$ となり、L2 キャッシュベースの見積もり時間: $t_{L2} = 1565.96GB / (671GB/sec) = 0.234sec$ となる。次に演算器ベース見積もり時間: $2.18G * 80 = 174.4Gflop$ 、 $t_C = 174.4Gflops / (768Gflops * 0.68) = 0.334sec$ となる。

1.2 節に示したモデルでは、経過時間: t_E は、 t_M と t_{L2} と t_C の MAX で計算されるため、演算器ベース見積もり時間: t_C が経過時間となる。3M-6L2-80F のプログラム CPU 性能解析レポートの結果を図 3.9 に示す。経過時間:0.331sec であり、ほぼ見積もり通りになっていることが分かる。CPU 性能解析レポートの演算器ビジー時間は本来は 0.331sec となり実行時間と一致するはずだが少なめになる傾向がある。演算器ビジー時間についても、1.3 節で示したビジー時間は、1.2 節で示した t_C と同じ概念ではあるが、そのものでないといえる。

タイプ2

Figure 3.7: テストプログラム（限界性能）「富岳」

```

integer,parameter :: N1= 3610
integer,parameter :: N2= 60
integer,parameter :: N3= 168

do k = 1,N3
  do j = 1,N2
    do i = 1,N1
      ww1 = (((((c(i,j+1,k)+c(i,j,k))*c(i,j-1,k))*
        c(i,j+1,k)+c(i,j-1,k))*
        c(i,j+1,k)+c(i,j-1,k))*
        c(i,j+1,k)+c(i,j-1,k))*
        c(i,j+1,k)+c(i,j-1,k))*
        c(i,j+1,k)+c(i,j-1,k))*
        c(i,j+1,k)+c(i,j-1,k))*
        c(i,j+1,k)+c(i,j-1,k))
      ww2 = (((((c(i,j+1,k)*c(i,j-1,k))*
        c(i,j+1,k)+c(i,j-1,k))*
        c(i,j+1,k)+c(i,j-1,k))*
        c(i,j+1,k)+c(i,j-1,k))*
        c(i,j+1,k)+c(i,j-1,k))*
        c(i,j+1,k)+c(i,j-1,k))*
        c(i,j+1,k)+c(i,j-1,k))
      ww3 = ((((((c(i,j+2,k)*c(i,j-2,k))*
        c(i,j+2,k)+c(i,j-2,k))*
        c(i,j+2,k)+c(i,j-2,k))*
        c(i,j+2,k)+c(i,j-2,k))*
        c(i,j+2,k)+c(i,j-2,k))*
        c(i,j+2,k)+c(i,j-2,k))*
        c(i,j+2,k)+c(i,j-2,k))*
        c(i,j+2,k)+c(i,j-2,k))
      ww4 = ((((((c(i,j+3,k)*c(i,j-3,k))*
        c(i,j+3,k)+c(i,j-3,k))*
        c(i,j+3,k)+c(i,j-3,k))*
        c(i,j+3,k)+c(i,j-3,k))*
        c(i,j+3,k)+c(i,j-3,k))*
        c(i,j+3,k)+c(i,j-3,k))*
        c(i,j+3,k)+c(i,j-3,k))
      ww5 = ((((((c(i,j+3,k)*c(i,j-3,k))*
        c(i,j+3,k)+c(i,j-3,k))*
        c(i,j+3,k)+c(i,j-3,k))*
        c(i,j+3,k)+c(i,j-3,k))*
        c(i,j+3,k)+c(i,j-3,k))*
        c(i,j+3,k)+c(i,j-3,k))
      at(i,j,k) = ww1 + ww2 + ww3 + ww4 + ww5
    enddo
  enddo
enddo

```

Figure 3.8: 演算器律速タイプテストプログラム

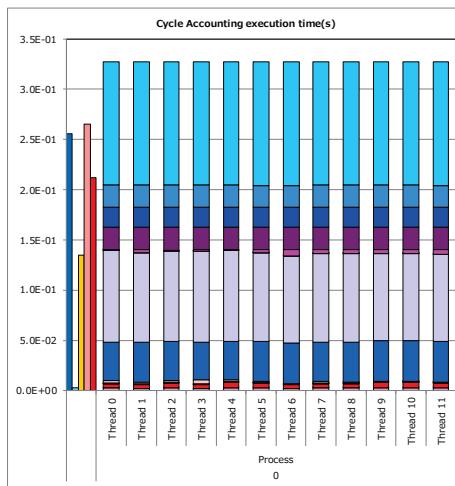


Figure 3.9: 3M-6L2-80F の CPU 性能解析レポートの結果

Chapter 4

「富岳」CPU単体性能向上のための基本的考え方

4.1 「富岳」CPU 性能向上の基本戦略

まず「京」、FX100で行われてきたソフトウェアパイプラインニング(SWPL)について、図4.1を用いて説明する。

図4.1に示した想定プログラムを考える。またその時のマシンモデルも図4.1に示した通りとする。その時、想定のプログラムに対し何も命令スケジューリングの最適化を実施しない場合、図4.1のオリジナルループに示したような命令の並びのようになる。すなわち、B(1)とC(1)をロードし3サイクル後にadd演算を実施し、また3サイクル後にストアすることになる。これで要素(1)に関する処理が終了するため、1要素に対し7サイクル要することとなる。100個の要素に対する処理には700サイクルを必要とする。そこで図4.1のSWPLされたループのように命令を並び替えることとする。すなわち、B(i)とC(i)のロードを毎サイクル開始し、3サイクル後に(i)要素に関するadd演算を開始し、さらに3サイクル後に(i)要素に関するstoreを実施する。このように命令を並び替える、言い換えると、命令をスケジューリングすることにより、カーネル部分は、1サイクル毎に要素(i)の結果を得ることができる。つまりカーネル部分は、100個の要素に対する処理は100サイクルで実行できることになる。もちろん前処理・後処理部を含めると、そうならないが、要素の数が多くなると前処理・後処理部の影響は小さくなる。

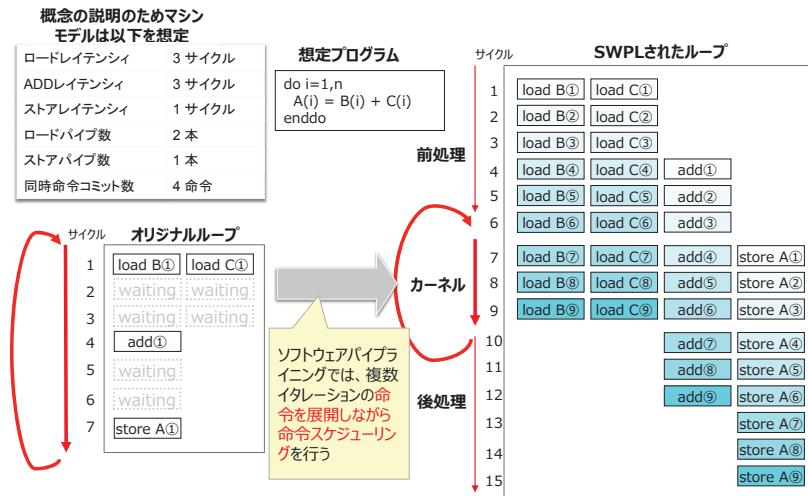


Figure 4.1: ソフトウェアパイプラインニング (SWPL)

このような処理を SWPL による命令スケジューリングと呼ぶ。

「京」、FX100は、128本のレジスタを備えており、コンパイラにより、豊富なレジスタ数を生かして、カーネル部分に多くのロード・演算・ストア命令を詰め込む SWPL を行ない効率的なオブジェクト生成し高い性能を確保していた。言うなれば、イン・オーダによる性能向上を目指していたといえる。

しかし「富岳」になると、レジスタ数が32本に減るため、詰め込むことができる命令が少なくなる。無理に命令を詰め込みすぎると、レジスタからキャッシュにデータが溢れるレジスタスpillが発生し性能が著しく低下する。このため SWPL によるスケジューリングが困難になり、このままでは性能が低下する可能性が生じる。

そこで「富岳」では、OoO(アウト・オブ・オーダー)資源を増強することにより、レジスタ数が減少した効果を回復させる戦略としている。アウト・オブ・オーダーとは、ハードウェアが、データなどの依存関係から判断し、複数の計算資源を用い、自動的に並列処理可能な命令を適宜開始・実行・完了させる機能である。

「富岳」CPU 性能向上の基本戦略とは、ソフトウェアパイプラインニングによるスケジューリングを実施しつつ、ARM 向けのコンパイラ (SIMD 拡張) を開発し、拡張されたアウト・オブ・オーダー機能を活用し得る、SWPL スケジューリングの改善やその他のシンプルなスケジューリングを開発し、かつこれらを全て活用したアプリケーションのチューニングを実施することである。

4.2 インテルとのアウトオブオーダー資源による比較

「富岳」では OoO 資源は増やしているが、インテルマシンの方がまだ資源量は多い。理研シミュレータで OoO 資源とレイテンシをインテルと同等にし性能比較を実施した。本評価で使用したアプリケーションは、オンキッシュで演算リッチなコードを使用し、「富岳」向けのチューニングが実施済みのアプリケーションである。

図 4.2 に結果を示す。

比較は、命令レイテンシと OoO 資源を、「富岳」のまま、Haswell 並みに増強、Skylake 並みに増強、Skylake 以上に増強、の4ケースを比較している。Haswell 並みにすると性能が、30%程度向上する結果となっている。OoO 資源を完全にインテルマシンと一致させた結果ではないし、理研シミュレータでの評価結果であるため、参考評価として捉えるべき結果であるが、オンキッシュで演算リッチな事例の場合、このような傾向が生じる可能性もあると考える。

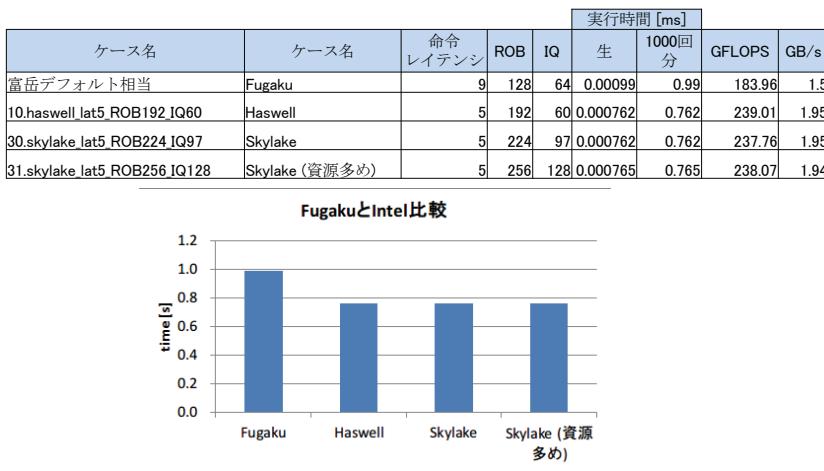


Figure 4.2: インテルとのアウトオブオーダー資源による比較

Chapter 5

「富岳」CPU性能チューニング技術

ここではチューニング技術とは単体 CPU のチューニング技術のことを言う。現時点で明らかになっている、いくつかの技術について紹介する。本説明の内容は、現在のコンパイラを前提としているものであり、今後のコンパイラでは状況が変わることもあり得る。

5.1 CPU 性能解析レポートの見方

まず最初にプログラムの性能を知るために、CPU 性能解析レポートの見方について説明する。CPU 性能解析レポートの中に、cycle account execution time というバーチャートが表示される。このバーチャートに 1.3 節で説明したビギー時間が表示されるが、その右側に各スレッドごとのサイクルアカウント時間が表示される。バーは、各種命令コミット時間と各種待ち時間で構成されている。待ち時間のうち薄紫色で表示される、浮動小数点演算待ち時間は、命令のスケジューリングが悪い場合に増大する。薄いピンク系で表示される L1D キャッシュアクセス待ち時間や L2 キャッシュアクセス待ち時間は、それぞれのキャッシュのレイテンシーが丸見えとなり、待ち時間が大きくなるときに増大する。赤色で表示されるメモリアクセス待ち時間は、メモリアクセスのレイテンシーが丸見えとなり、待ち時間が大きくなるときに増大する。濃いピンク系で表示されるバリア同期待ち時間は、スレッドインバランスが生じている場合増大する。

ここに述べた内容を図 5.1 に示す。

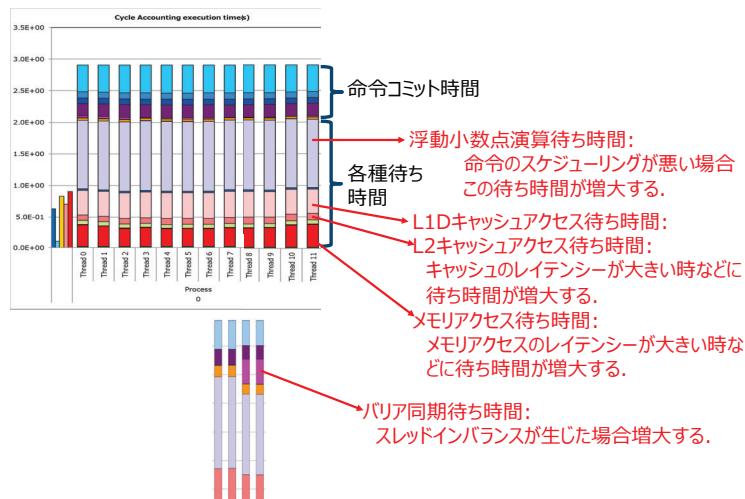


Figure 5.1: CPU 性能解析レポートの見方

5.2 タイプ毎のチューニング技術 (1)

タイプ毎のチューニング技術 (1) として、図 5.2 に示した範囲に効果があると考えるチューニング技術を示す。ここで示すチューニング技術は、図 5.2 に示すように、要求 B/F 値が小さいアプリから大きいアプリまで、適用可

能な技術である。また、図 5.3 に示すように、色々なアプリに適用できる標準的な手法であると考える。以下、図 5.3 について簡単に説明した後に、それぞれの項で詳細に説明する。

(1) ループ融合

図 5.3 の左上段に示したような 3 重ループのステンシル計算があるとする。内側の 2 重ループにループ長の増大を目的としてループ融合を行う。また図 5.3 の左下段に示したように 3 重ループを融合する場合もある。

(2) ブロック化

ループ融合によって長くなったループ長を、今後実施するループ分割を見据えて十分な最内ループ長を確保しつつブロック化を実施する。これによりループ分割された時に必要となるループ間の受け渡しのためのワーク配列の局所化が可能とする。

(3) ループ融合とスレッド化

最外のループが短くスレッド化に必要な十分なループ長が取れない場合、ブロック化で発生した直下のループと融合を実施しスレッド化可能なループ長を確保する。

(4) 自動ループ分割・手動ループ分割

ループボディーが大きい場合、命令スケジューリングのためのレジスタが不足する場合が発生する。その場合は、ループ分割を実施しそれぞれのループボディを小さくし、使用するレジスタ数を削減する必要がある。ループ分割は、手動で実施する場合と自動ループ分割を使用する場合がある。

(3) プリフェッチ

ハードウェアプリフェッチが有効に動作していない場合は、ソフトウェアプリフェッチを有効にする、またはソフトウェアプリフェッチ命令を明に挿入することが有効である。

(3) SWPL(ソフトウェアパイプラインニング)の促進

命令スケジューリングの重要な機能として SWPL(ソフトウェアパイプラインニング)がある。本機能を有効活用するために幾つかのテクニックを用いる。

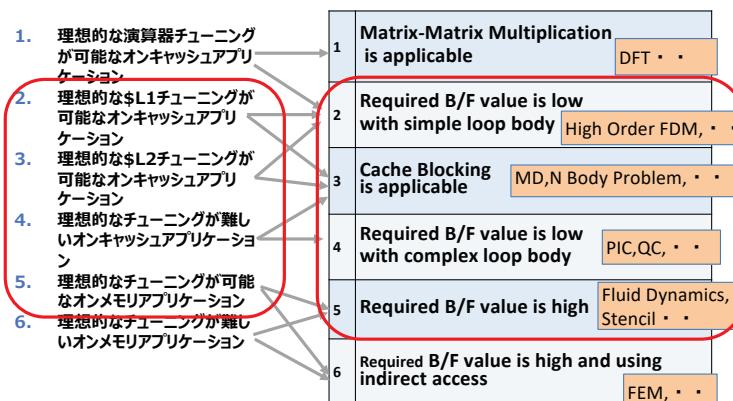


Figure 5.2: タイプ毎のチューニング技術 (1) の適用範囲

5.2.1 SWPL の促進とキャッシュ効率向上-ループ分割&ループ融合&ブロッキング-

以下にループ分割・ループ融合とブロッキングによる SWPL の促進とキャッシュ効率向上のチューニングについて説明する。「富岳」は、レジスタ数が 32 個となり、「京」・FX100 の 128 個と比較するとレジスタ数が減少している。そのためソフトウェアパイプラインニング (SWPL) 等のスケジューリングのためのレジスタが不足し、レジスタスピル等が発生することが多く、「京」・FX100 のコードそのままだと性能が劣化する可能性がある。このような場合に有効なチューニングが、ループ分割を実施し、ループ内で使用するレジスタ数を削減することである。

ただループ分割を実施したとしても、ループ長が十分でない場合は、スケジューリングができないことになる。このような場合は、ループ融合を実施しループ長を大きくすることが有効なチューニングとなる。ループ融合と

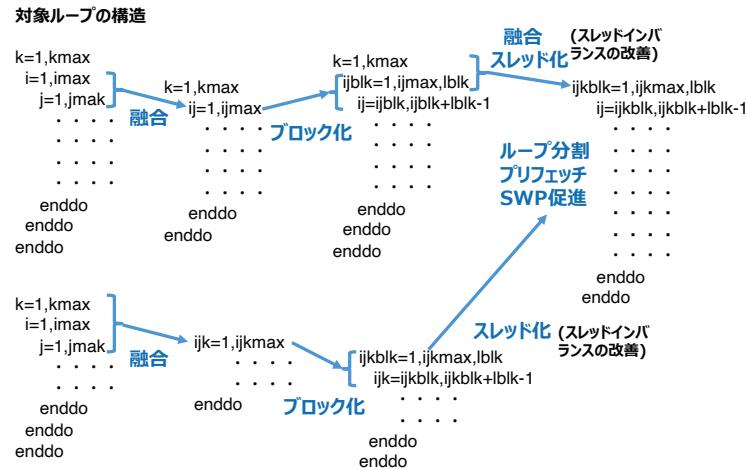


Figure 5.3: タイプ毎のチューニング技術 (1) のまとめ

ループ分割を用いた場合、分割したループ間でデータを受け渡しすることが必要となり、ループ長が長すぎると、この受け渡しのためのメモリアクセスが多くなり、性能の劣化を招く。そこでループ融合したループを、十分なスケジューリングが可能な最内ループ長を持つように、ブロッキングするチューニングが有効になる。

チューニング対象とする asis ループの概要を図 5.4 に、またチューニング後のループの概要を図 5.5 に示す。さらにチューニングコードの性能のグラフおよび性能指標値を図 5.6、図 5.7 に示す。

① r4_tune02

asis に対し配列の次元移動、ループ分割、プリフェッチ等を適用し、その効果により高速化している。最内ループ j の繰り返し数が 128 程度と小さいため SWPL が適用されず演算待ちが発生している状態である。最外次元 k で並列化しているが private 配列の領域や 1 次元ループ分割の領域は少なくすんでいる。

② r4_collapsed.tune02

内側 2 次元 ij をループ融合。SWPL を進めるため ij のループ融合を実施した。ij の繰り返し数が 16900 程度と大きくなつたためスレッド並列化を ij で実施した。各スレッド毎に 1408 (=16900/12th) の回転数があり、SWPL の適用により浮動小数点演算待ちが①に比べ減少している。しかし、1408 回転ごとにアクセスが途切れているためプリフェッチの効率が悪化し、キャッシュ待ち時間が増大している。

③ r4_collapsed.tune03

プリフェッチ効率改善のためスレッド並列を K 次元に戻したが、ij の回転数が大きいため、ループ間で使用するワーク配列等が増え、メモリアクセスが増大し性能が悪化した。そこでローカル配列の局所化のためのブロッキングを実施（ブロックサイズ 128）し、メモリアクセスを削減した。またループ長:128 でも SWPL 適用可能なように、コンパイラの最適化処理の改善を実施した。さらに同一計算のみを集めて分割し作業配列を削減。演算順序変更の副次効果として FMA 命令が増加し通常の浮動小数点命令が減少し総命令数が減少した。asis より L2 ビジー時間による性能の限界値に大きく近づいていることが分かる。

```

!$omp parallel ...
do l = 1, ADM_lall
!$omp do
do k = 1, ADM_kall
do j = ADM_gmin-1, ADM_gmax
do i = ADM_gmin-1, ADM_gmax
rhot1_xyz(i, j, l) = rho(i, j, k, l) ...
rhot2_xyz(i, j, l) = rho(i, j, k, l) ...
enddo
do j = ADM_gmin-1, ADM_gmax
...
enddo
do j = ADM_gmin, ADM_gmax
...
do i = ADM_gmin, ADM_gmax
rrhoa2 = 1.0_RP / max(rhot2_xyz(i, j-1, l) + rhot1_xyz(i, j, l) ...;
tmp_rho(i, 1) = tmp_rho(i, 1) * rrhoa2 * dt * 0.5_RP
tmp_rho(i, 2) = tmp_rho(i, 2) * rrhoa2 * dt * 0.5_RP
tmp_rho(i, 3) = tmp_rho(i, 3) * rrhoa2 * dt * 0.5_RP
enddo
do i = ADM_gmin, ADM_gmax
...
enddo
do i = ADM_gmin, ADM_gmax
GRD_xc(i, j, k, l, AI , XDIR) = GRD_xr_ij_l(i, j, KO, l, AI , XDIR) - ...
GRD_xc(i, j, k, l, AI , YDIR) = GRD_xr_ij_l(i, j, KO, l, AI , YDIR) - ...
GRD_xc(i, j, k, l, AI , ZDIR) = GRD_xr_ij_l(i, j, KO, l, AI , ZDIR) - ...
enddo
do i = ADM_gmin, ADM_gmax
...
enddo
enddo ! k-end
enddo ! l-end
!$omp end parallel

```

Figure 5.4: ループ分割&ループ融合&ブロッキングの例 (asis)

③ r4_collapsed.tune03

ループプロック化を実施
(ブロックサイズ 128)

```

!$omp parallel ...
do l = 1, ADM_lall
!$omp do
do k = 1, ADM_kall
...
(局所化の前処理)
do ijjj = ADM_gall_1d * (ADM_gmin-1) + ADM_gmin-1, nend , block_size
if(block_size .le. (nend - ijjj + 1) ) then
  do ij = ijjj, ijjj+block_size-1
    rhot1(ij-ijjj+1) = rho (ij , k, l) ...
    rhot2(ij-ijjj+1) = rho (ij , k, l) ...
  enddo
...
do ij = ijjj, ijjj+block_size-1
rrhoa21 = 1.0_RP / max( rhot2(ij-ijjj+1-ADM_gall_1d) + rhot1(ij-ijjj+1) ...;
tmp_rhoxt1(ij-ijjj+1) = tmp_rhoxt1(ij-ijjj+1) * rrhoa21
tmp_rhoxt1(ij-ijjj+1) = tmp_rhoxt1(ij-ijjj+1) * rrhoa21
tmp_rhozt1(ij-ijjj+1) = tmp_rhozt1(ij-ijjj+1) * rrhoa21
enddo
do ij = ijjj, ijjj+block_size-1
GRD_xc(ij, k, l, AI , XDIR) = GRD_xr_ij_l(ij, KO, l, AI , XDIR) - ...
GRD_xc(ij, k, l, AI , YDIR) = GRD_xr_ij_l(ij, KO, l, AI , YDIR) - ...
GRD_xc(ij, k, l, AI , ZDIR) = GRD_xr_ij_l(ij, KO, l, AI , ZDIR) - ...
enddo
...
(局所化のための後処理)
else
...
(blocking の余り処理)
endif
enddo
enddo
!$omp end parallel

```

Figure 5.5: ループ分割&ループ融合&ブロッキングの例 (r4_collapsed.tune03)

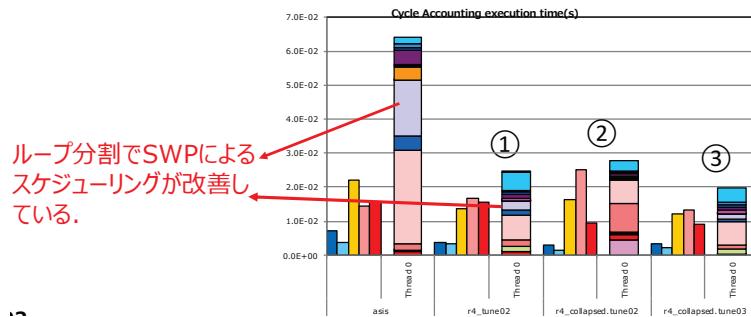


Figure 5.6: ループ分割&ループ融合&ブロッキングの性能 (グラフ)

ソースコード版数	asis	tune02	r4Collapsed, tune02	r4Collapsed, tune03
チューニング内容	-	ループ分割 配列次元移動	ループ一重化	ループ一重化、ループ ブロッキング
コンパイラ版数	tcsu1.2.27b	tcsu1.2.27b	tcsu1.2.27b	tcsu1.2.27b
周波数(GHz)	2.0	2.0	2.0	2.0
カーネル化、縮小	あり	あり	あり	あり
浮動小数点数精度	単精度	単精度	単精度	単精度
集計スレッド番号	0	0	0	0
実行時間[s]	6.40E-02	2.45E-02	2.77E-02	1.97E-02
GFLOPS/プロセス	32.26	85.62	79.31	110.94
メモリループト(R+W) GB/s/プロセス	62.03	163.78	86.91	119.57
浮動小数点演算数/スレッド	1.72E+08	1.75E+08	1.83E+08	1.82E+08
実行命令数/スレッド	3.26E+07	5.19E+07	3.02E+07	4.22E+07
ロード・ストア命令数/スレッド	1.08E+07	2.20E+07	1.33E+07	2.03E+07
L1DIMス数/スレッド	1.46E+06	1.69E+06	2.96E+06	1.66E+06
L2ス数/スレッド	8.77E+05	8.76E+05	5.17E+05	4.15E+05
L1ビジー率	34.14%	55.55%	58.64%	63.18%
L2ビジー率	22.55%	68.54%	91.08%	67.44%
メモリビジー率	24.23%	63.98%	33.95%	46.71%

Figure 5.7: ループ分割&ループ融合&ブロッキングの性能 (性能指標値)

5.2.2 自動ループ分割機能の適用 (1)

「富岳」プロジェクトでは、自動ループ分割機能が富士通によって開発された。この機能は、これ以前に存在した制御行によるループ分割点の指示によるループ分割でなく、自動でループの分割点を見出しループ分割を行う機能である。

以下に気象コードの物理過程 (Micro physics) に対し自動ループ分割機能を適用したチューニングについて説明する。

チューニング対象とする asis ループの概要を図 5.8 に、またチューニング後のループの概要を図 5.9 に示す。さらにチューニングコードの性能のグラフおよび性能指標値を図 5.10、図 5.11 に示す。

図 5.8 を見るとループボディが非常に大きく SWPL の適用が出来ていないことが分かる。また単精度と倍精度が混在するため 8SIMD 化のみが実施されており、SPILL/FILL が大量に発生していた。

図 5.9 に示すように、チューニングは、LOOP_FISSION_TARGET 指示子で自動ループ分割を実施している。ループ分割の粒度は、-Kloop_fission_threshold=40 オプションにより、40 を指定している (tune1)。-Kloop_fission_threshold オプションを指定しない場合は、自動でループ分割の粒度が決定されるが、今回は複数の粒度で実行した結果、性能が良好であった 40 を選択している。

また tune2 では tune1 に加え、ループ分割によりワーク配列の増加によるメモリアクセスの増加が予想されるため、ブロッキングを適用している (tune2)。

図 5.10 を見ると tune1 では演算待ちが改善しているが、その代わりにメモリ・キャッシュアクセス待ちが増加している。asis と比較して全体実行時間が 35% 減少している。tune2 ではメモリ・キャッシュアクセス待ちが改善され、asis と比較すると実行時間が 62% 減少している。

● コード概略(asis)

```

!<<< Parallel do default(null), (...) &
!<<< LOOP_FISSION_PSTATE(bcelf, bc,coef_at,wk)
do k = kmin, kmx
do j = 1, jdm
    dens = rho(j,k)
    temp = tem(j,k)
    qv = max(q(j,k,l,OV), 0.0, RP)
    qp = max(q(j,k,l,OQ), 0.0, RP)
    qr = max(q(j,k,l,OS), 0.0, RP)
    qg = max(q(j,k,l,OQ), 0.0, RP)
    qs = max(q(j,k,l,OS), 0.0, RP)
    qd = max(q(j,k,l,OQ), 0.0, RP)
    S1q = qv / max(qstat(j,k), EPS)
    S2q = qv / max(qstat(j,k), EPS)
    Rdens = 1.0, RP / dens
    rho_fact = sqrt(dens*0.0, Rdens)
    temp = temp - TEMX0
    wk(l,delta1) = (0.5, RP + sign(0.5, RP, qr - 1.E-4, RP))
    wk(l,delta2) = (0.5, RP + sign(0.5, RP, 1.E-4, RP - qr)) &
                  *(0.5, RP + sign(0.5, RP, 1.E-4, RP - qr))
    wk(l,spsat) = 0.5, RP + sign(0.5, RP, Sice - 1.0, RP)
    wk(l,icellg) = 0.5, RP - sign(0.5, RP, temc) ! 0: warm, 1: ice
    enddo
enddo

```

(省略)

```

    Vt(j,k,l,OQ) = Vtr
    Vt(j,k,l,OI) = Vti
    Vs(j,k,l,OQ) = Vss
    Vs(j,k,l,OI) = Vsg
enddo

```

<<< Loop-information Start >>>
<<< [OPTIMIZATION]
<<< SWPL : 0
<<< PREFETCH(I)(ARD) Expected by compiler :
 a, rho, Vt, tem, a2, a1, ma2, Nc, pre
<<< PREFETCH(SOFT) : 20
<<< SEQUENTIAL : 20
<<< qstat: 2, q: 2, qstat: 2, dRhogq: 2
<<< dRhogq: 2, rhog: 2, dRhogq: 2
<<< qstat: 2, dRhogq: 2, dRhogq: 2
<<< SPILLS : 0
<<< GENERAL : SPILL 0 FILL 4
<<< SIMD4FP : SPILL 0 FILL 0
<<< SCALABLE : SPILL 294 FILL 696
<<< PREDICATE : SPILL 0 FILL 0
<<< Loop-information End >>>

- ループボディが大きく SWPL が適用されない(約700行)
- 単精度と倍精度が混在するため 8SIMD 化
- SPILL/FILL が大量に発生

Figure 5.8: 自動ループ分割機能の適用 (1) の例 (asis)

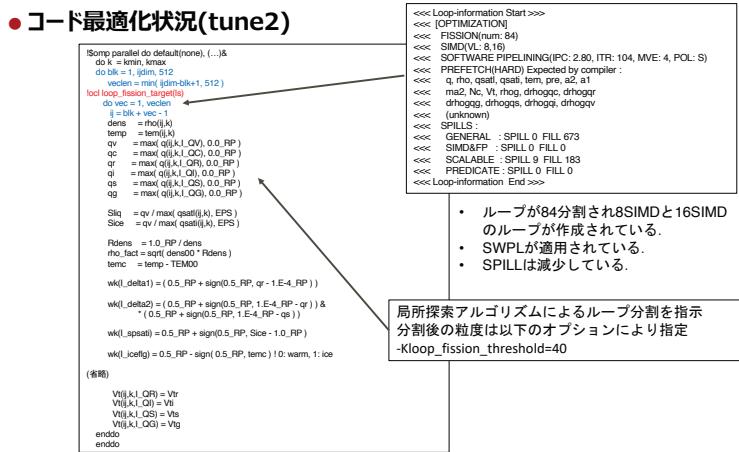


Figure 5.9: 自動ループ分割機能の適用 (1) の例 (tune2)

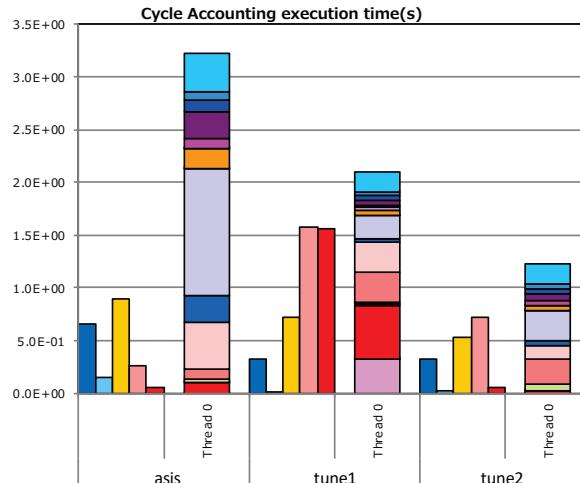


Figure 5.10: 自動ループ分割機能の適用 (1) の性能 (グラフ)

ソースコード版数	asis	tune1	tune2
チューニング内容	-	自動ループ分割	自動ループ分割 ブロッキング
コンパイラ版数	tcsu1.2.26	tcsu1.2.26	tcsu1.2.26
周波数(GHz)	2.0	2.0	2.0
カーネル化、縮小	あり	あり	あり
浮動小数点数精度	単精度	単精度	単精度
集計スレッド番号	0	0	0
実行時間[s]	3.23E+00	2.09E+00	1.23E+00
GFLOPS/プロセス	101.63	86.91	148.4
メモリスループット(R+W) GB/s/プロセス	4.74	190.58	13.30
浮動小数点演算数/スレッド	2.79E+10	1.55E+10	1.55E+10
実行命令数/スレッド	4.40E+09	1.98E+09	2.12E+09
ローク-ストア命令数/スレッド	1.57E+09	7.55E+08	8.41E+08
L1Dミス数/スレッド	3.02E+07	1.32E+08	9.44E+07
L2ミス数/スレッド	3.67E+06	9.07E+07	3.90E+06
L1ビージー率	28.23%	34.55%	44.87%
L2ビージー率	8.04%	75.33%	58.61%
メモリビージー率	1.85%	74.44%	5.19%

Figure 5.11: 自動ループ分割機能の適用 (1) の性能 (性能指標値)

5.2.3 自動ループ分割機能の適用 (2)

以下に直交階層格子法による流体コードのうち移流計算部のコードに対し、自動ループ分割機能を適用した事例について説明する。図 5.12 に自動ループ分割機能を適用した結果のグラフを示す。横軸は、-Kloop_fission_threshold で指定したループ分割の粒度を表しており、threshold = 40 から 50 で、手動で分割した 31 ループ分割と同等の性能が得られていることが分かる。

ただしこのデータは、少し古いコンパイラと理研シミュレータを使用して求めた経過時間を整理した結果であり、データとして少し古くなっている可能性はある。

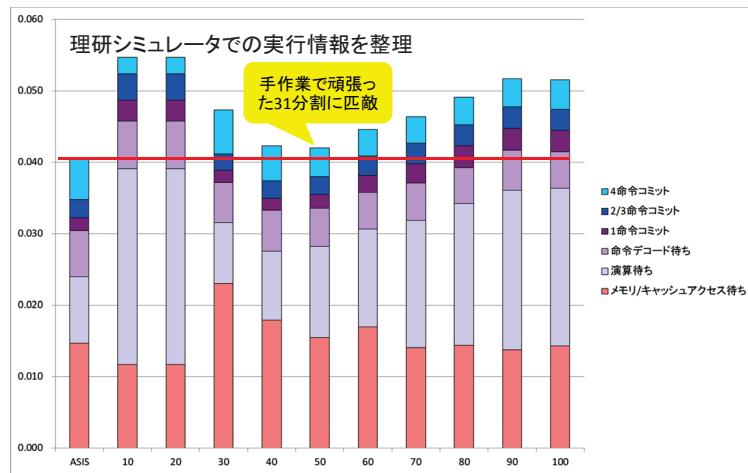


Figure 5.12: 自動ループ分割機能の適用 (2) の性能 (グラフ)

5.2.4 SWPL の促進とキャッシュ効率向上-ループ融合&ブロッキング&ループ分割&プリフェッチ-

以下にステンシル計算のアプリに対し、ループ融合とブロッキング・ループ分割・プリフェッチによる SWPL の促進とキャッシュ効率向上のチューニングの実施結果について説明する。チューニング後のループの概要を図 5.13 に示す。さらにチューニングコードの性能のグラフおよび性能指標値を図 5.14、図 5.15 に示す。

チューニングは、ループ分割により SWPL が適用されるようにしている。またメモリ/キャッシュアクセス待ち改善のため、ブロックサイズ 128 でループブロック化およびプリフェッチ指示行の追加を行っている。

図 5.14 を見ると、SWPL が適用されるようになりスケジューリングが改善し、浮動小数点演算待ちが大きく改善することにより実行時間が 40% 改善している。asis よりも L1 キャッシュの再利用性が高まり L1 ビジー時間が増大している。その代わり L2 キャッシュのビジー時間が減少するとともに、asis より経過時間が L2 ビジー時間に近づく改善がなされている。

```

!$omp parallel (...) 
!!! blocking-start
  !$omp do
    do ijjj = 1, ijdm block_size
      if( block_size <= ijdm-ijjj ) then
        !$omp prefetch_write(ERM(ijjj:ijjh:block_size), level=2, strong=1)
        !$omp prefetch_write(ERM(ijjj:ijjh+64:block_size), level=2, strong=1)
        !$omp prefetch_write(ERM(ijjj:ijjh+28:block_size), level=2, strong=1)
        !$omp prefetch_write(ETM(ijjj:ijjh:block_size), level=2, strong=1)
      ...
    !$omp enddo
    do ij = ijjj, ijjh-block_size-1
      if ( THK(ij) > 0.0_RP ) then
        OMG = SCAP(ij) / THK(ij)
      else
        OMG = 1.0_RP
      endif
      tmp_FACT(ij-ijjj+1) = G(ij,1) - G(ij,3)*OMG
      tmp_TAU(ij-ijjj+1) = THK(ij) / tmp_FACT(ij-ijjj+1)
      ! TAU switch
      tmp_IAUsw(ij-ijjj+1) = 0.5_RP - sign( 0.5_RP, EPST-tmp_TAU(ij-ijjj+1) )
      tmp_OMGT(ij-ijjj+1) = ( G(ij,1)-G(ij,3) )*OMG / ( G(ij,1)-G(ij,3) )*OMG
      tmp_OMGT(ij-ijjj+1) = min( tmp_OMGT(ij-ijjj+1), rEPS )
    !$omp enddo
    ...
    !!! blocking-mod-start
    else
      ... (blocking の余り処理)
    endif
  !$omp end parallel
  !$omp enddo

```

ループブロック化
ブロックサイズ 128

プリフェッチ指示行の追加

ループ分割

● ループ融合&ブロッキング&ループ分割&プリフェッチの例。

Figure 5.13: ループ融合&ブロッキング&ループ分割&プリフェッチの例 (tune)

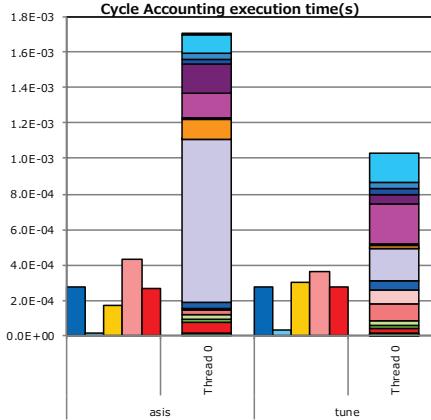


Figure 5.14: ループ融合&ブロッキング&ループ分割&プリフェッチの性能 (グラフ)

ソースコード版数	asis	tune
チューニング内容	-	ループ融合&ブロッキング&ループ分割&プリフェッチ
コンパイラ版数	tcsu1.2.27b	tcsu1.2.27b
周波数(GHz)	2.0	2.0
カーネル化、縮小	あり	あり
浮動小数点数精度	単精度	単精度
集計スレッド番号	0	0
実行時間[ms]	1.70E-03	1.03E-03
GFLOPS/プロセス	136.46	228.98
メモリスループット(R+W)	40.71	68.17
GB/s/プロセス		
浮動小数点演算数/スレッド	1.93E+07	2.01E+07
実行命令数/スレッド	1.45E+06	1.97E+06
ロード・ストア命令数/スレッド	2.16E+05	5.73E+05
L1Dミス数/スレッド	3.94E+04	4.07E+04
L2ミス数/スレッド	8.34E+03	9.10E+03
L1ヒジ率	10.49%	30.97%
L2ヒジ率	25.32%	35.01%
メモリヒジ率	15.90%	26.63%

Figure 5.15: ループ融合&ブロッキング&ループ分割&プリフェッチの性能 (性能指標値)

5.2.5 プリフェッч指示行の追加

配列が連続アクセスされている場合は、通常はハードウェアプリフェッチが動作する。しかしステンシル計算で配列が連続アクセスしている場合でも、キャッシュ待ち時間が大きくL1DやL2Dキャッシュミスマス率が大きい場合がある。キャッシュミスマス率は、ハードウェアプリフェッチによるミス率、ソフトウェアプリフェッチによるミス率、デマンドによるミス率から構成され、合わせて100%となるが、前者2つが大きい場合は問題ないが、デマンドによるミス率が大きい場合は問題である。この場合プリフェッチ指示行の指定によるソフトウェアプリフェッチの発行が、性能向上に有効な場合がある。

図5.16に、プリフェッチ指示行の指定の例を示す。

この例では、Kループが降順、ijループは昇順のため、Kループの次の回転が連続アクセスにならない。そのため連續性が途切れ、ハードウェアプリフェッチが効かなくなる。この改善のために、L2キャッシュにKループの1ライン先をプリフェッチする指示行を指定している。

- ソフトウェアプリフェッチを発行することでdm率を下げて性能向上を狙う。

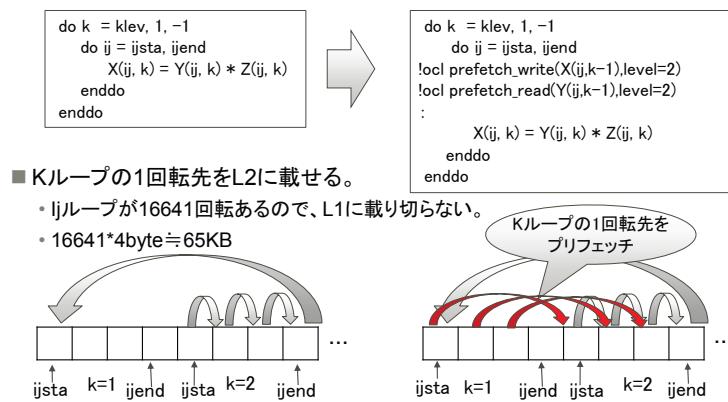


Figure 5.16: プリフェッチ指示行の例 (1)

図5.17に、さらにプリフェッチする指示行を追加している例を示す。

この例は、ループの中に8ストリームの配列アクセスが存在する場合を考えている。その場合、キャッシュライン256B分をプリフェッチすると、1回転あたり8ストリーム×256B=2048B=2KB分をプリフェッチすることとなる。「富岳」のL1Dキャッシュは64KBであり、2KBはL1Dキャッシュに十分載る量である。256バイトは単精度では64要素分であり、図5.17の指示子では、L1Dキャッシュに、配列の64個分先の分をプリフェッチすることを指示している。

```

do k = klev, 1, -1
  do ij = ijsta, ijend
    !ocl prefetch_write(RL(ij,k-1),level=2)
    :
    X(ij, k) = Y(ij, k) * Z(ij, k)
  enddo
enddo

3955   do ij = ijsta, ijend
3956   !ocl prefetch_write(RL(ij,k-1),level=2)
3957   !ocl prefetch_write(SL(ij,k-1),level=2)
3958   !ocl prefetch_read(SL(ij,k),level=2)
3959   !ocl prefetch_read(RL(ij,k),level=2)
3960   !ocl prefetch_read(RE(ij,k-1),level=2)
3961   !ocl prefetch_read(TE(ij,k-1),level=2)
3962   !ocl prefetch_read(SER(ij,k-1),level=2)
3963   !ocl prefetch_read(SET(ij,k-1),level=2)
3964   !ocl prefetch_write(RL(ij+64,k),level=1)
3965   !ocl prefetch_write(SL(ij+64,k),level=1)
3966   !ocl prefetch_read(SL(ij+64,k+1),level=1)
3967   !ocl prefetch_read(RL(ij+64,k+1),level=1)
3968   !ocl prefetch_read(RE(ij+64,k),level=1)
3969   !ocl prefetch_read(TE(ij+64,k),level=1)
3970   !ocl prefetch_read(SER(ij+64,k),level=1)
3971   !ocl prefetch_read(SET(ij+64,k),level=1)
3972   recip = 1.0_RP / ( 1.0_RP - RL(ij,k+1)*RE(ij,k) )
3973
3974   RL(ij,k) = RE(ij,k) + TE(ij,k) * ( RL(ij,k+1)*TE(ij,k) ) * recip
3975   SL(ij,k) = SER(ij,k) + TE(ij,k) * ( RL(ij,k+1)*SET(ij,k) + SL(ij,k+1) ) * recip
3976
3977 enddo
3978 enddo

```

Figure 5.17: プリフェッチ指示行の例 (2)

プリフェッチ指示行の例(1)(2)を実施したチューニングコードの性能グラフおよび性能指標値を図5.18、図5.19に示す。

asisでは、L1Dミスdm率が63.40メモリ・キャッシュアクセス待ち時間が多いという問題があることが分かる。これらのL1Dミスdm率とL2ミスdm率を下げるためにプリフェッチ指示行の追加した結果、メモリアクセ

ス待ち時間および L2 キャッシュアクセス待ち時間が大幅に減少し、また L2 ビジー時間も減少し、全体の経過時間が 30% 減少した。その結果、経過時間が L2 キャッシュビジー時間に迫るまでの性能が達成された。実行命令数が増加しているが、これはプリフェッч指示行を入れたことによるプリフェッч命令の発行、およびプリフェッчアクセス先のアドレス計算により命令が増加したためである。

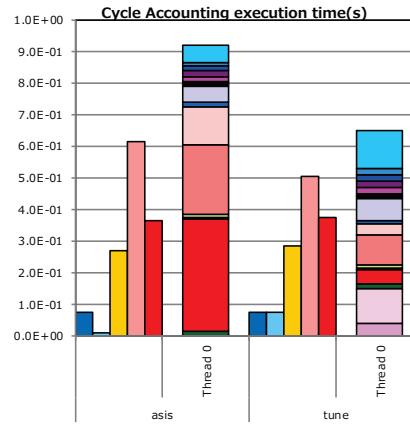


Figure 5.18: プリフェッチ指示行の例 (1)(2) の性能 (グラフ)

ソースコード版数	asis	tune
チューニング内容	-	プリフェッチ 指示行の追加
コンパイラ版数	tcsu1.2.27b	tcsu1.2.27b
周波数(Ghz)	2.0	2.0
カーネル化・縮小	あり	あり
浮動小数点数精度	単精度	単精度
集計スレッド番号	0	0
実行時間[s]	9.20E-01	6.49E-01
GFLOPS/プロセス	74.90	106.13
メモリループト(R+W) GB/s/プロセス	101.36	147.05
浮動小数点演算数/スレッド	5.76E+09	5.76E+09
実行命令数/スレッド	5.95E+08	1.18E+09
ロード・ストア命令数/スレッド	1.83E+08	1.89E+08
L1ミス数/スレッド	5.10E+07	4.04E+07
L2ミス数/スレッド	2.29E+07	2.44E+07
L1ミスdm率	63.40%	18.18%
L2ミスdm率	49.99%	17.30%
L1ビジー率	30.42%	45.01%
L2ビジー率	66.71%	78.08%
メモリビジー率	39.59%	57.44%

Figure 5.19: プリフェッチ指示行の例 (1)(2) の性能 (性能指標値)

5.2.6 multiple structure load/store 命令の利用

図 5.20 の左側:asis の青い四角で囲まれているような、1 次元目がインデックスとなっているループでは、通常では効率が悪い gather load 命令/scatter store 命令が使用される。しかし 1 次元目の配列宣言が 1-4 の場合、効率の良い multiple structure load/store 命令が使用できる場合もある。この multiple structure load 命令を使用することを目指したチューニングを実施した。

具体的には、図 5.20 の右側:tune のように、ワーク配列を使用し、青い四角で囲まれた部分の FXYZ のロードとストアを別配列を使用することとする。こうすることにより、gather load 命令/scatter store 命令ではなく、multiple structure load/store 命令が使用されるようになる。ただし、これは現在のコンパイラの状況であり、今後、このような書き換えを実施しなくとも、multiple structure load/store 命令が使用されるようになることが期待される。

asis	tune
<pre> SUBROUTINE GRAD3X 中略 REAL*4 FXYZ(3,NP) 中略 DO 1000 IP = 1, NP DO 1100 I = 1, 8 中略 1100 CONTINUE FXYZ(1,IP)=FXBUF FXYZ(2,IP)=FYBUF FXYZ(3,IP)=FZBUF 1000 CONTINUE 中略 DO 1200 J=1,NUMVALID DO 1300 I=9,NEP(IP) 中略 1300 CONTINUE FXYZ(1,IP)=FXYZ(1,IP)+FXBUF FXYZ(2,IP)=FXYZ(2,IP)+FYBUF FXYZ(3,IP)=FXYZ(3,IP)+FZBUF 1200 ENDDO 中略 CALL DDCOMY(IPART,LDOM,NBPDOM, * NDOM,IPSLF,IPSN,MBPDOM, * FXYZ,NP,IUTO,IERR,RX,RY,MAXBUF) 中略 DO 2100 IP=1,NP FXYZ(1,IP)=FXYZ(1,IP)*CM(IP) FXYZ(2,IP)=FXYZ(2,IP)*CM(IP) FXYZ(3,IP)=FXYZ(3,IP)*CM(IP) 2100 CONTINUE </pre> <div style="border: 1px solid blue; padding: 5px; margin-top: 10px;"> multiple structures load/store命令が 発行されていない </div>	<pre> SUBROUTINE GRAD3X 中略 REAL*4 FXYZ(3,NP) REAL*4 FXYZ_imp(3,NP) 中略 DO 1000 IP = 1, NP DO 1100 I = 1, 8 中略 1100 CONTINUE FXYZ_tmp(1,IP)=FXBUF FXYZ_tmp(2,IP)=FYBUF FXYZ_tmp(3,IP)=FZBUF 1000 CONTINUE 中略 DO 1200 J=1,NUMVALID DO 1300 I=9,NEP(IP) 中略 1300 CONTINUE FXYZ_tmp(1,IP)=FXYZ_imp(1,IP)+FXBUF FXYZ_tmp(2,IP)=FXYZ_imp(2,IP)+FYBUF FXYZ_tmp(3,IP)=FXYZ_imp(3,IP)+FZBUF 1200 ENDDO 中略 CALL DDCOMY(IPART,LDOM,NBPDOM, * NDOM,IPSLF,IPSN,MBPDOM, * FXYZ_imp,NP,IUTO,IERR,RX,RY,MAXBUF) 中略 DO 2100 IP=1,NP FXYZ(1,IP)=FXYZ_imp(1,IP)*CM(IP) FXYZ(2,IP)=FXYZ_imp(2,IP)*CM(IP) FXYZ(3,IP)=FXYZ_imp(3,IP)*CM(IP) 2100 CONTINUE </pre> <div style="border: 1px solid blue; padding: 5px; margin-top: 10px;"> 定義、参照する配列名を 別名にすると発行される </div>

Figure 5.20: multiple structure load 命令の利用 (1)

multiple structure load/store 命令の利用によるチューニングコードの性能グラフおよび性能指標値を図 5.21、図 5.22 に示す。gather load は例えば、 $FXYZ(1, \dots)$ 1 個の SIMD のギャザーロードに対し、1 つの命令が発行されるが、multiple structure load では、 $FXYZ(1, \dots) \dots FXYZ(3, \dots)$ に対し 1 個の SIMD 命令が発行される。図 5.22 を見ると、gather load、scatter store 命令の代わりに、multiple structure load/store 命令が発行されたため、ロード・ストア命令数が大幅に減少していることが分かる。図 5.21 を見ると、効率の良いロード・ストア命令が発行されているため、L1 キャッシュアクセス待ちが大幅に減少し L1 キャッシュビジー時間も減少している。さらにメモリのバンド幅を上限近くまで使用する性能を達成している。その結果、経過時間は、1.73 秒から 1.22 秒に減少している。

もう 1 つの例を図 5.23 に示す。図 5.23 の上半分について、1 次元目の 1-6 を 3 次元目に、下半分について、1 次元目の 1-3 を 3 次元目に次元移動している。このチューニングによる性能グラフを図 5.24 に示す。配列の次元移動は、以前のコンパイラでは大きな性能改善が得られていた。しかしコンパイラの最適化が進み、図 5.23 の下半分についての 1 次元目の 1-3 を 3 次元目に次元移動したチューニングは、1 次元の大きさが 3 であるため、multiple structure load 命令が使用されるようになり asis の性能が大きく向上し、今回のチューニングの効果は得られなかった。つまり次元が 1-4 の場合は、multiple structure load が有効になるため、次元移動のチューニングは必要ないということである。

1 次元目の大きさが 4 以上である、図 5.23 の上半分については、次元移動による連続アクセス化のチューニングは有効であると思われるが、下半分の箇所が多いいため、効果が見られない結果となっている。1 次元目の大きさが 4 以上である場合は、次元移動のチューニングは実施すべきである。

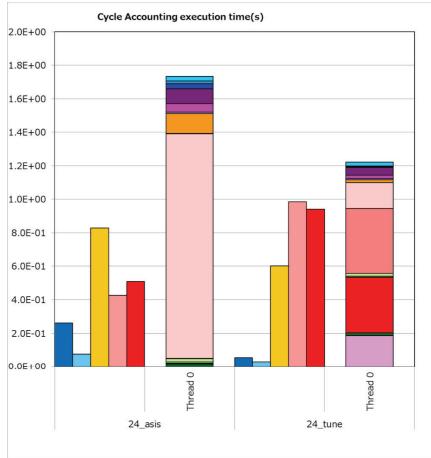


Figure 5.21: multiple structure load/store 命令の利用 (1) によるチューニングの性能 (グラフ)

ソースコード版数	asis	tune
チューニング内容	-	multiple structures load/store命令の発行
コマンド版数	tcstds-1.2.27b	tcstds-1.2.27b
周波数(GHz)	2.0	2.0
カーネル化、縮小	あり	あり
浮動小数点数精度	単精度	単精度
集計スレッド番号	0	0
実行時間[s]	1.73E+00	1.22E+00
GFLOPS/プロセス	11.45	16.24
メモリスレップト(R+W) GB/s/プロセス	75.14	196.96
浮動小数点演算数/スレッド	1.66E+09	1.66E+09
実行命令数/スレッド	6.15E+08	3.32E+08
ロード・ストア命令数/スレッド	2.48E+08	1.10E+08
L1Dミス数/スレッド	3.56E+07	6.16E+07
L2ミス数/スレッド	3.62E+07	6.24E+07
L1ピージ率	47.70%	49.15%
L2ピージ率	24.57%	80.59%
メモリピージ率	29.35%	76.94%
multiple structures load命令数/スレッド	0.00E+00	3.45E+07
multiple structures store命令数/スレッド	0.00E+00	3.45E+07
gather load命令数/スレッド	1.03E+08	0.00E+00
scatter store命令数/スレッド	1.03E+08	0.00E+00

Figure 5.22: multiple structure load/store 命令の利用 (1) によるチューニングの性能 (性能指標値)

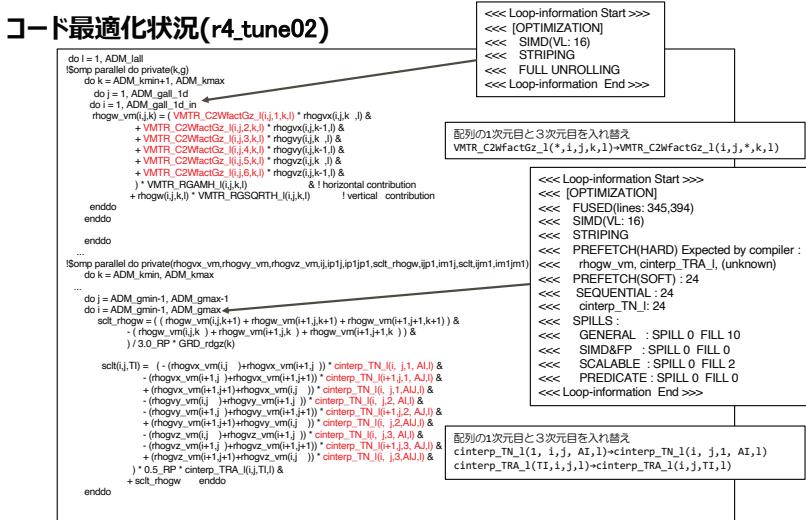


Figure 5.23: multiple structure load 命令の利用 (2)

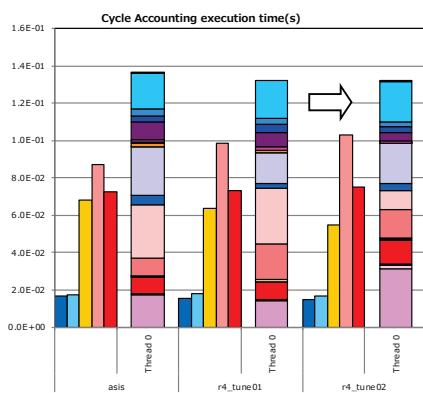


Figure 5.24: multiple structure load 命令の利用 (2)(性能グラフ)

5.2.7 SWP オプションの利用

4.1 節にソフトウェアパイプラインニング (SWPL) の説明を示した。図 4.1 に示した、 i 番目のイタレーションと $i+1$ 番目のイタレーション間の命令開始のサイクルの距離を、initiation interval(ii) と呼ぶ(図 5.25 参照)。ii が短いほど命令を詰め込めることができるが必要なレジスタ数が多くなる。ii が長くなると必要なレジスタ数は少なくて済むが命令を詰め込めずスカスカになり、スケジューリングの効果が小さくなる。「京」や FX100 では、ii を小さいところから伸ばしていき、使用できるレジスタの範囲で、効果の得られる命令列を探すスケジューリングを行っていた。良い命令列が得られない場合は、スケジューリング処理をギブアップしていた。つまり一つの解を見つけに行き、見つからない場合は SWPL を諦めていた。

「富岳」プロジェクトでは、OoO 資源の拡張に伴い、「強い SWPL」としてレジスタ不足によるスピルを OoO で効率化することを前提に、レジスタスピルを出しても SWPL を適用する処理をコンパイラに追加し、ユーザーが指定することが可能となっている。

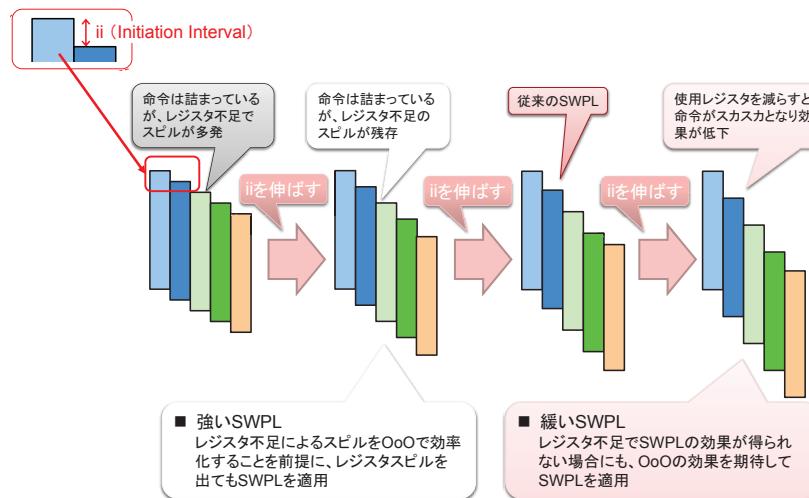


Figure 5.25: 強い SWPL

強い SWPL の例を図 5.26 に示す。図 5.26 の左側に asis のコンパイルリスト右側に、swp_strong オプションを指定した場合のコンパイルリストを示す。右側のリストでは、SWPL が効いているのが分かる。

(*swp_strong オプションは Fortran および C/C++ の Trad モードのオプションである。)

強い SWPL の例の性能グラフおよび性能指標値を図 5.27、図 5.28 に示す。この例では、SWPL の適用が促進され演算待ちが改善したが、今回は大きな性能向上は得られなかった。しかし、本オプションが有効な例があるものと考え、ここに記載しておく。

コード最適化状況(asis)

コード最適化状況(r4_tune01)

<pre> <<< Loop-information Start >>> <<< [OPTIMIZATION] <<< SPILLS : <<< GENERAL : SPILL 0 FILL 2 <<< SIMD&FP : SPILL 0 FILL 0 <<< SCALABLE : SPILL 0 FILL 0 <<< PREDICATE : SPILL 0 FILL 0 <<< Loop-information End >>> 263 3 p <<< Loop-information Start >>> 264 4 p fv do i = 1, ADM_gall_1d 265 4 p fv rhogw_vml(i,j,k) = (VMTR_C2WfactGz_(1,i,j,k,l) * rhogvxi(j,k_,l) + 266 4 p fv + VMTR_C2WfactGz_(2,i,j,k,l) * rhogvxi(j,k-1,l) & 267 4 p fv + VMTR_C2WfactGz_(3,i,j,k,l) * rhogvxi(j,k-2,l) & 268 4 p fv + VMTR_C2WfactGz_(4,i,j,k,l) * rhogvxi(j,k-3,l) & 269 4 p fv + VMTR_C2WfactGz_(5,i,j,k,l) * rhogvxi(j,k-4,l) & 270 4 p fv + VMTR_C2WfactGz_(6,i,j,k,l) * rhogvxi(j,k-5,l) & 271 4 p fv + VMTR_RGAMH_(i,j,k,l)) * rhogvzi(j,k_,l) & 272 4 p fv + rhogw(i,j,k,l) * VMTR_RGSQRTH_(i,j,k,l) 273 4 p fv enddo 274 3 p enddo 275 2 p enddo 276 2 p </pre>	<pre> <<< Loop-information Start >>> <<< [OPTIMIZATION] <<< PREFETCH(HARD) Expected by compiler : <<< rhogvxi, rhogvyy, VMTR_RGSQRTH_ <<< VMTR_C2WfactGz_(1,i,j,k,l), rhogvxi, VMTR_RGAMH_ <<< rhogw, rhogvxi, VMTR_RGSQRTH_ <<< SPILLS : <<< GENERAL : SPILL 1 FILL 58 <<< SIMD&FP : SPILL 0 FILL 0 <<< SCALABLE : SPILL 0 FILL 0 <<< PREDICATE : SPILL 0 FILL 0 <<< Loop-information End >>> 263 3 p <<< Loop-information Start >>> 264 4 p v do i = 1, ADM_gall_1d 265 4 p v rhogw_vml(i,j,k) = (VMTR_C2WfactGz_(1,i,j,k,l) * rhogvxi(j,k_,l) & 266 4 p v + VMTR_C2WfactGz_(2,i,j,k,l) * rhogvxi(j,k-1,l) & 267 4 p v + VMTR_C2WfactGz_(3,i,j,k,l) * rhogvxi(j,k-2,l) & 268 4 p v + VMTR_C2WfactGz_(4,i,j,k,l) * rhogvxi(j,k-3,l) & 269 4 p v + VMTR_C2WfactGz_(5,i,j,k,l) * rhogvxi(j,k-4,l) & 270 4 p v + VMTR_C2WfactGz_(6,i,j,k,l) * rhogvxi(j,k-5,l) & 271 4 p v + VMTR_RGAMH_(i,j,k,l)) * rhogvzi(j,k_,l) & 272 4 p v + rhogw(i,j,k,l) * VMTR_RGSQRTH_(i,j,k,l) 273 4 p v enddo </pre>
--	--

Figure 5.26: 強い SWPL の例

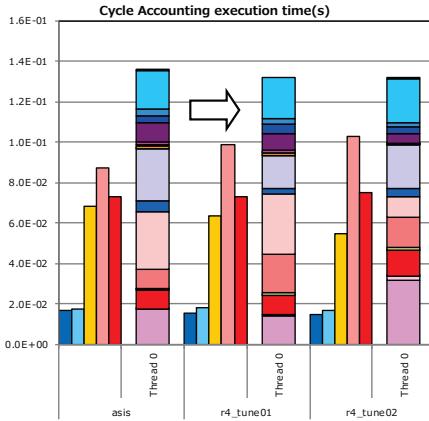


Figure 5.27: 強いSWPLの例(性能グラフ)

ソースコード版数	asis	r4_tune01
チューニング内容	-	オプションによる16 SIMD化
コソライタ数	tcsu1.2.29	tcsu1.2.29
周波数(GHz)	2.0	2.0
カーネル化・縮小	あり	あり
浮動小数点数精度	単精度	単精度
集計スレッド番号	0	0
実行時間[s]	1.35E-01	1.32E-01
GFLOPS/プロセス	103.70	106.16
メモリスレーブド(R+W) GB/s/プロセス	138.49	141.97
浮動小数点演算数/スレッド	1.22E+09	1.22E+09
実行命令数/スレッド	2.05E+08	2.11E+08
ロードストア命令数/スレッド	6.03E+07	6.00E+07
L1Dスル数/スレッド	1.02E+07	1.03E+07
L2ミス数/スレッド	4.71E+06	4.77E+06
L1ビギー率	54.14%	51.02%
L2ビギー率	64.89%	74.87%
メモリビーグ率	54.10%	55.46%
連続SIMD命令数/スレッド	2.54E+07	2.54E+07
キャタロード命令数/スレッド	6.72E+06	6.72E+06
ストラクチャード命令数/スレッド	5.64E+06	5.64E+06

Figure 5.28: 強いSWPLの例(性能指標値)

5.2.8 本節のまとめ

5.2 節のまとめを、図 5.29 に示す。

浮動小数点演算待ち時間の改善には、SWPL 等のスケジューリングの改善が必要である。そのためにはループ分割による使用レジスタ数削減が有効である。またループ融合によるループ長増大も有効である。

キャッシュ待ち時間の改善には、プリフェッチによるキャッシュアクセスレイテンシー削減が有効である。メモリアクセス・メモリアクセス待ち時間の改善には、ループ分割されたループのブロック化によるワーク配列の大きさの削減が有効である。バリア同期待ち時間の削減には、ループ融合によるループ長増大、それによるスレッドインバランス削減が有効である。命令コミット時間の削減には、不連続なデータアクセスの連續化によるロード命令の効率化が有効である。

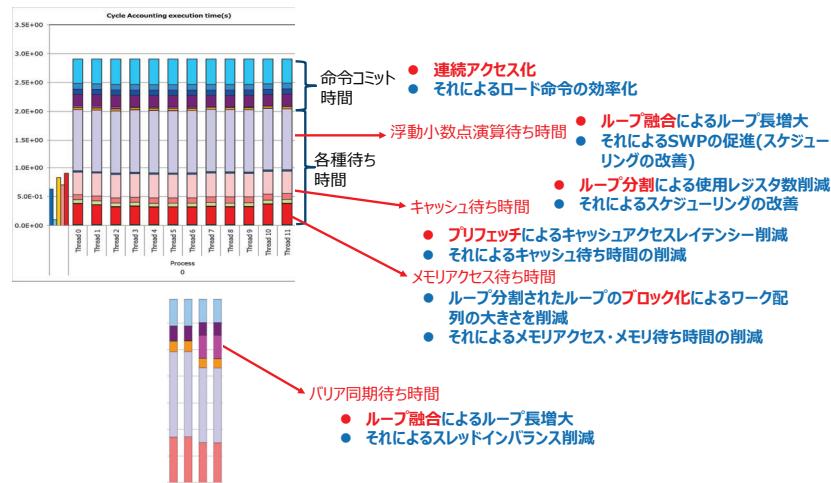


Figure 5.29: 5.2 節のまとめ

5.3 タイプ毎のチューニング技術 (2)

タイプ毎のチューニング技術 (2) として、図 5.30 に示した範囲に効果があると考えるチューニング技術を示す。ここで示すチューニング技術は、図 5.30 に示すように、要求 B/F 値が小さくインダイレクトアクセスを用いたアプリに適用可能な技術である。このタイプのアプリケーションは、最も性能が出しにくいアプリケーションであると考えられるが、本節に示したチューニング手法を用いることが可能なアプリケーションであれば、オンメモリなアプリとしての理想的な性能を得ることも可能になる。

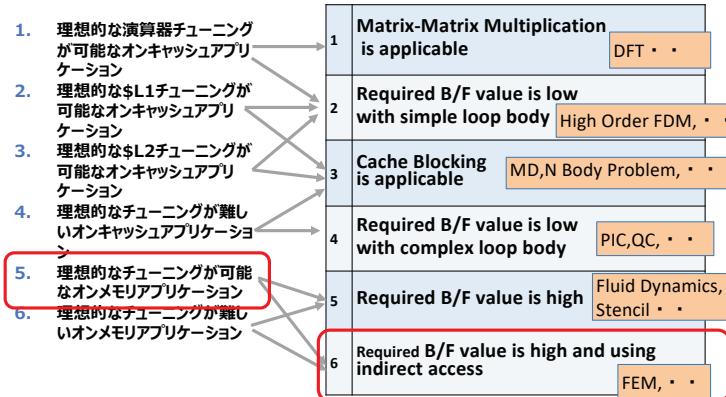


Figure 5.30: タイプ毎のチューニング技術 (2) の適用範囲

5.3.1 リストアクセスに対する節点のオーダリング

ここで図 5.31 に示すような典型的な疎行列とベクトルの積のプログラムを考える。行列とベクトルは、CRS 格納形式となっており、行列は連続アクセス、ベクトルのアクセスがリストアクセスとなっている。

このプログラムについて、ルーフラインモデルに従い性能を見積もる。ここに示すデータは「京」のデータであるので、「京」を例として性能を見積もる。ルーフラインモデルでは、リストでアクセスされるベクトルデータは、キャッシュに載っているという理想的な仮定を用いる。ベクトルの部分が L1 キャッシュに載っていると仮定した場合ベクトルのメモリへのアクセスを全く無視してよいため、メモリからのロードは行列とリストのみとなり、このプログラム(単精度のプログラム)の要求バイトは、単精度を 2load なので $2 \times 4 = 8\text{byte}$ となる(行列の書き込みは、Do 100 ループの外にあり実行回数が少ないため無視した場合)。要求フロップ値は、積が 1、和が 1 の計 2 となる。したがってこのプログラムの要求 B/F 値は、 $8/2=4$ となる。「京」のハードウェアの理論メモリバンド幅は、64GB/sec、実効メモリバンド幅は、46GB/sec、理論性能(CPU)は、128Gflops、であるため、ハードウェアの実効的な B/F 値は、 $46/128=0.36$ となる。

したがって、ベクトルデータがオンキャッシュであるという、理想的な仮定を置いた場合のルーフラインモデルで見積もったピーク性能比は、ハードウェアの実効的な B/F 値(0.36)をアプリの要求 B/F 値(4)で割って求めたため、 $0.36/4=0.09$ 、9%となる。「京」のときの図 5.31 のプログラムの実測値は、この見積もり値の 1/5 から 1/10 の低い性能であった。

```

ICRS=0
DO 110 IP=1,NP
    BUF=0.0E0
    DO 100 K=1,NPP(IP)
        ICRS=ICRS+1
        IP2=IPCRS(ICRS)
        BUF=BUF+A(IP2)*S(IP2)
    100 CONTINUE
    AS(IP)=AS(IP)+BUF
110 CONTINUE

```

Figure 5.31: 疎行列とベクトルの積のプログラム

この低い性能の原因は、ベクトルデータが仮定したようなオン L1D キャッシュに載っていないことである。こ

の状況を見るために、横軸に定義される自己自身の節点番号を、縦軸に定義される節点を計算するために参照される節点番号をプロットした図を示す。図 5.32 に 4 面体のプロット図、図 5.33 に 6 面体のプロット図を示す。

4 面体は、総回転数: 約 270 万であるが、全アクセスとも広範囲なランダムアクセスとなっていることが分かる。6 面体は、総回転数: 約 2700 万の最初の部分をプロットしているが、最初の 1M 回程度が、広範囲なランダムアクセス、それ以降は二極化した広範囲なアクセスとなっていることが分かる。

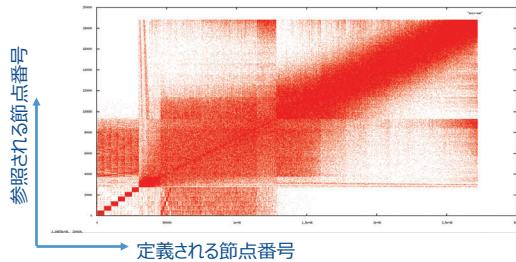


Figure 5.32: 4 面体の節点の定義・参照のプロット図

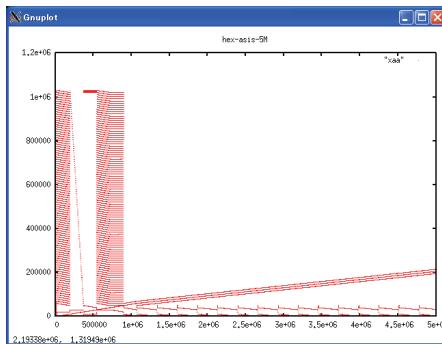


Figure 5.33: 6 面体の節点の定義・参照のプロット図

そこで、この広範囲なランダムアクセスを解消するために、物理的に近い節点がリスト配列の並びとしても近い位置に配置される事を期待できる、リストアクセスに対する節点のオーダリングを実施する。具体的には、図 5.34 に示すように、まずオリジナルデータを各軸で複数に分割しブロックを作成する。次に各ブロックを外と内に分割し、物理座標に基づき内側・外側の順にナンバリングする。このような節点のオーダリングを実施することにより、1 要素を構成する節点の番号が近くなり、ブロックの大きさを調整することにより、ベクトルのリストアクセスの多くに対しオン L1D キャッシュのデータを利用できるようになる。

オーダリング後の節点の定義・参照関係の 4 面体のプロット図を図 5.35 に、6 面体のプロット図を図 5.36 に示す。両方とも 3 本の線に分散しているように見えるが、拡大してみると、両方とも広範囲なランダムアクセスが局所化されてることが分かる。

このオーダリング手法を用いて「京」で実行した結果を図 5.37 に示す。6 面体、4 面体共に、ベクトルの部分が L1 キャッシュに載っていると仮定した場合の見積もり値、9% の性能に近いピーク性能比が得られている。

次に「富岳」で取得したオーダリングの効果の性能グラフおよび性能指標値を図 5.38、図 5.39 に示す。オーダリングによりリストアクセスの多くに対し、オン L1D キャッシュのデータを利用できるようになったことで L1D ミス数が減少し、メモリ・キャッシュアクセス待ちが減少している。実行命令数が asis tune で増加しているが全スレッド合計では減少している(6 面体)。浮動小数点演算数が asis tune で増加(スレッド 0)しているが、全スレッド合計では一致している(4 面体)。6 面体・4 面体ともに大きく性能が向上していることが分かる。

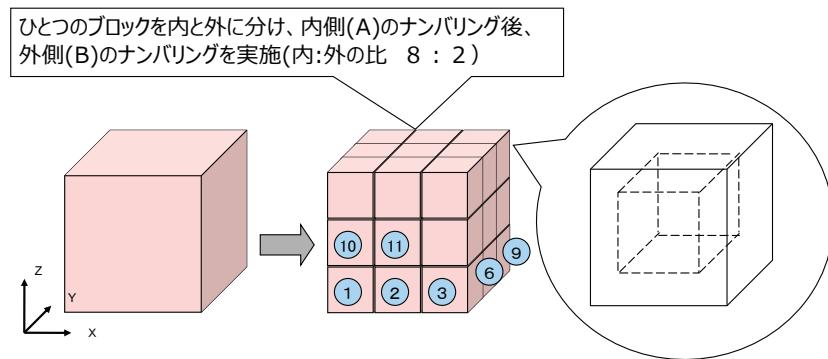


Figure 5.34: 節点のオーダリング手法

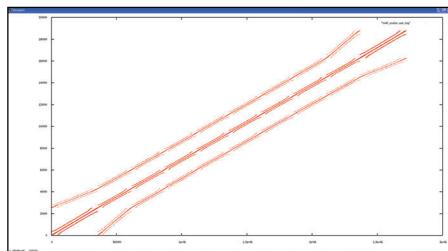


Figure 5.35: 4面体の節点の定義・参照のプロット図(オーダリング後)

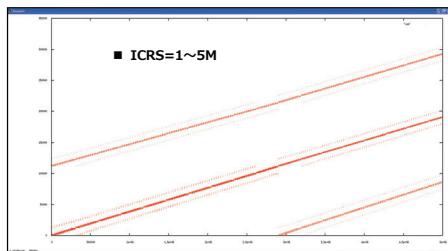


Figure 5.36: 6面体の節点の定義・参照のプロット図(オーダリング後)

6面体	4面体
フルアンロール (8core)	5.4%
フルアンロール + リオーダリング (8core)	8.1% 7.7%

↑
↑
L1 オンキヤッシュである時の理論性能値である9%に近い性能値を実現

Figure 5.37: 「京」でのオーダリング後の性能

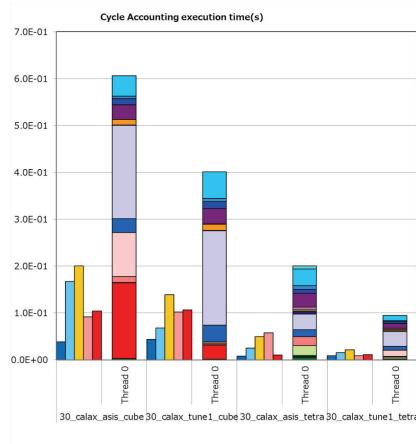


Figure 5.38: オーダリング後の性能 (性能グラフ)

ソースコード版数	asis(6面体)	tune1(6面体)	asis(4面体)	tune1(4面体)
チューニング内容	asis	オーダリング	asis	オーダリング
コンパイラ版数	tcسدs=1.2.26	tcسدs=1.2.26	tcسدs=1.2.26	tcسدs=1.2.26
周波数(Ghz)	2.0	2.0	2.0	2.0
カーネル化・縮小	あり	あり	あり	あり
浮動小数点数精度	単精度	単精度	単精度	単精度
集計スレッド番号	0	0	0	0
実行時間[s]	6.06E-01	3.91E-01	1.93E-01	9.45E-02
Gflops/プロセス	21.93	34.02	10.02	20.43
メモリループ(R+W) GB/s/プロセス	43.86	69.54	13.21	29.32
浮動小数点演算数/スレッド	1.10E+09	1.11E+09	1.52E+08	1.58E+08
実行命令数/スレッド	5.84E+08	6.21E+08	4.22E+08	1.37E+08
ロード・ストア命令数/スレッド	1.38E+08	1.48E+08	1.25E+08	3.48E+07
L1Dミス数/スレッド	1.33E+07	8.97E+06	3.68E+06	9.51E+05
L2ミス数/スレッド	8.10E+06	8.28E+06	6.32E+05	7.77E+05
L1ビギー率	33.05%	35.60%	25.48%	22.01%
L2ビギー率	15.11%	26.09%	29.91%	9.42%
メモリビギー率	17.13%	27.16%	5.16%	11.45%

Figure 5.39: オーダリング後の性能 (性能指標値)

5.3.2 ループのリロールによる SIMD 化

オリジナルのループ概要を図 5.40 に示す。最内ループの xyz に対応する、1,2,3 が手動で展開されている。また、上位ループの節点に対応する、1-8 も展開されており、SIMD 化はされていない。これらの手動展開されているループをリアンロールしループ構造に戻す。リアンロールしたループの概要を図 5.41 に示す。最内ループは、「富岳」の SIMD 幅に合わせて 8 としてあるが、実際に使用される要素は、xyz を表す 3 である。それぞれ上位ループの 1-8 は、要素に含まれる節点数を表しており、リアンロールしている。このリアンロールにより、最内ループは SIMD 化され、8 回転の上位ループは自動アンロールされている。また、その上位の要素のループは SWPL 化されている。

ループのリロールによる SIMD 化の結果については、次項の中で説明する。

● コード概要(asis)

```

465   1           DO 1411 ICOLOR=1,NCOLOR(4)
466   1           !ocl norecurrence(FXYZ)
467   2   pp         DO 1410 ICPART=1,NCPART(ICOLOR,4)
468   2   p           IE$=LLOOP(ICPART,1,ICOLOR,4)
469   2   p           IEE=LLOOP(ICPART+1,ICOLOR,4)-1
470   2           !ocl nosimd
471   2           !ocl noswp
472   3   p           DO 1408 IE$=IES,IEE
473   3   p           IF (LE(FIX(IE),EQ,1) GOTO 1400
474   3   p           IP1=NODE(1,IE)
475   3   p           IP2=NODE(2,IE)
476   3   p           IP3=NODE(3,IE)
477   3   p           IP4=NODE(4,IE)
478   3   p           IP5=NODE(5,IE)
479   3   p           IP6=NODE(6,IE)
480   3   p           IP7=NODE(7,IE)
481   3   p           IP8=NODE(8,IE)
482   3           C
483   4   p           IF (IVOF,GE,1) THEN
484   4   p           SWRK=S(IE)/RH03D(IE)
485   4   p           ELSE
486   4   p           SWRK = S(IE)
487   4   p           ENDIF
488   3           C
489   3   p           FXYZ(1,IP1)=FXYZ(1,IP1)-SWRK*DXYZ(1,1,IE)
490   3   p           FXYZ(2,IP1)=FXYZ(2,IP1)-SWRK*DXYZ(2,1,IE)
491   3   p           FXYZ(3,IP1)=FXYZ(3,IP1)-SWRK*DXYZ(3,1,IE)
492   3           C
493   3   p           FXYZ(1,IP2)=FXYZ(1,IP2)-SWRK*DXYZ(1,2,IE)
494   3   p           FXYZ(2,IP2)=FXYZ(2,IP2)-SWRK*DXYZ(2,2,IE)
495   3   p           FXYZ(3,IP2)=FXYZ(3,IP2)-SWRK*DXYZ(3,2,IE)
      ...省略...
517   3   p           FXYZ(1,IP8)=FXYZ(1,IP8)-SWRK*DXYZ(1,8,IE)
518   3   p           FXYZ(2,IP8)=FXYZ(2,IP8)-SWRK*DXYZ(2,8,IE)
519   3   p           FXYZ(3,IP8)=FXYZ(3,IP8)-SWRK*DXYZ(3,8,IE)
520   3   p           1400  CONTINUE
521   2   p           1410  CONTINUE
522   1           1411  CONTINUE

```

Figure 5.40: オリジナルコードのループ構造

● コード概要(tune01)

```

1           DO ICOLOR=1,NCOLOR(4)
1           !ocl norecurrence(WXYZ)
1           !Somp do
中略
2   p           DO ICPART=1,NCPART(ICOLOR,4)
中略
2           locl swp
<<< Loop-information Start >>>
<<< [OPTIMIZATION]
<<< SOFTWARE PIPELINING(IPC: 0.33, ITR: 32, MVE: 2, POL: L)
<<< PREFETCH(HARD) Expected by compiler :
<<< DXYZ, S
<<< Loop-information End >>>
3   p           DO IE$=IES,IEE
3   p           SWRK=S(IE)
3           <<< Loop-information Start >>>
3           <<< [OPTIMIZATION]
3           <<< FULL UNROLLING
3           <<< Loop-information End >>>
4   p   f         DO IU=1,8
4   p   f         IP1=NODE(IU,IE)
4           locl nouroll
4           <<< Loop-information Start >>>
4           <<< [OPTIMIZATION]
4           <<< SIMD(VL: 8)
4           <<< Loop-information End >>>
5   p   fv        DO II=1,VLEN
5   p   fv        WXYZ(II,IP1)=WXYZ(II,IP1)-SWRK*DXYZ(II,IU,IE)
5   p   fv        ENDDO
4   p   f         ENDDO
3   p           ENDDO
2   p           ENDDO
1           !Somp end do
1           ENDDO
1           !Somp end parallel

```

Figure 5.41: チューニングコードのループ概要

5.3.3 ストア側のリストアクセスの削減-要素ループから節点ループへの変更-

オリジナルのループ構造の説明を以下に示す。また対応する図を図 5.42 に、対応するソースコードを図 5.43 に示す。このコードは、ストアする節点番号がリストアクセスとなるため、スキヤッタストア命令が使用される。この命令は「京」の 2SIMD の時は、それほどペナルティが大きくなかったが、「富岳」で 8SIMD となつたため、ペナルティが大きくなつており、実行効率の悪化を招くことになる。

- ・要素でループを回す構造となっている
- ・各節点へ値をストアする必要がある
- ・ストアする節点番号がリストアクセスとなる
- ・ストア側のリストアクセスは遅いためなるべく避ける方が良い
- ・ストアがリカレンスとなるためリカレンスの削除のためのカラーリングも必要となる

オリジナルの構造を改善するためのチューニングの説明を以下に示す。また対応する図を図 5.44 に、対応するソースコードを図 5.45 に示す。チューニングとして、要素でループを回す構造から節点でループを回す構造に変更する。こうすることにより、実行効率の悪化を招くスキヤッタストアを使用することなく、効率の良い連続アクセスによるストアを使用することができるようになる。

- ・要素でループを回す構造から節点でループを回す構造に変更
- ・各節点周りの要素の関連する節点値を足しこむ
- ・ストアする節点番号は連続アクセスとなる
- ・カラーリングも必要ない

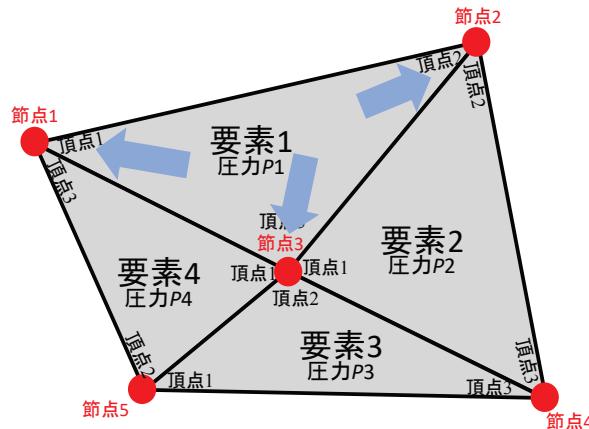


Figure 5.42: オリジナル処理の概要

本項で実施したチューニングの効果および前項で実施したチューニングの効果の性能グラフおよび性能指標値を図 5.46、図 5.47 に示す。tune1 が、ループのリロールによる SIMD 化の効果であり、tune2 が tune1 に加えて、要素ループから節点ループへの変更を行つた結果である。

tune1 は狙い通り、SIMD 化によって命令数が大きく減少している。しかしリストアクセスの影響が大きくなり、プリフェッчがあたらず L1D ミス dm 率が増加している。また、メモリ・キャッシュアクセス待ち時間が顕在化している。その結果、経過時間全体の改善の効果は薄くなっているが、命令数が大きく減少した結果、L1D ビジー時間・演算器ビジー時間は大きく減少しており、tune2 の改善の前提として良いチューニングになっている。tune2 ではストアのリストアクセスが大幅に削減され L1D ビジー時間が大幅に減少し、メモリループットも大幅に改善し大きな性能改善結果が得られた。経過時間は、L2 キャッシュビジー時間に迫つてあり、上限性能に近い性能が得られている。

```

asis
1      DO 1411 ICOLOR=1,NCOLOR(4)
1      locl norecurrence(FXYZ)
1      !$omp do
2 p      DO 1410 ICPART=1,NCPART(ICOLOR,4)
中略      . . . .
2      locl nosimd
2      locl swp
<<< Loop-information Start >>>
<<< [OPTIMIZATION]
<<< PREFETCH(HARD) Expected by compiler :
<<< LEFIX
<<< SPILLS :
<<< GENERAL : SPILL 0 FILL 3
<<< SIMD&FP : SPILL 0 FILL 0
<<< SCALABLE : SPILL 0 FILL 0
<<< PREDICATE : SPILL 0 FILL 0
<<< Loop-information End >>>
3 p      DO 1400 IE=IES,IEE
3 p      IF (LEFIX(IE),EQ.1) GOTO 1400
3 p      IP1=NODE(1,IE)
中略      . . . .
3 p      IP8=NODE(8,IE)
中略      . . . .
3 p      FXYZ(1,IP1)=FXYZ(1,IP1)-SWRK*DNXYZ(1,1,IE)
3 p      FXYZ(2,IP1)=FXYZ(2,IP1)-SWRK*DNXYZ(2,1,IE)
3 p      FXYZ(3,IP1)=FXYZ(3,IP1)-SWRK*DNXYZ(3,1,IE)
中略      . . . .
3 p      FXYZ(1,IP8)=FXYZ(1,IP8)-SWRK*DNXYZ(1,8,IE)
3 p      FXYZ(2,IP8)=FXYZ(2,IP8)-SWRK*DNXYZ(2,8,IE)
3 p      FXYZ(3,IP8)=FXYZ(3,IP8)-SWRK*DNXYZ(3,8,IE)
3 p      1400 CONTINUE
2 p      1410 CONTINUE
1      1411 CONTINUE
1      !$omp end do
1      !$omp end parallel

```

NODE配列を使用

ストアする節点番号
はリストアクセス

Figure 5.43: オリジナル処理のソースコード概要

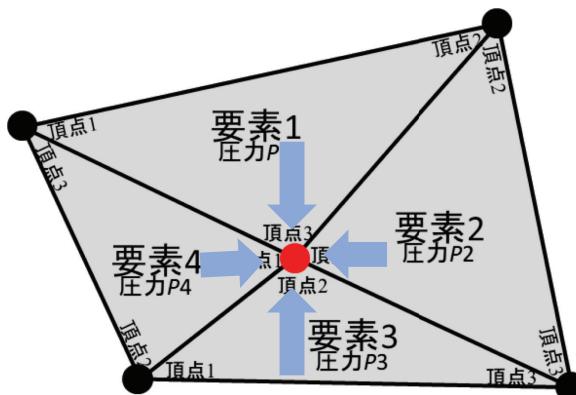


Figure 5.44: オリジナル処理の概要

tune02

```

!$omp do
  !$cl SIMD
  !$cl SWP
<<< Loop-information Start >>>
<<< [OPTIMIZATION]
<<< SOFTWARE PIPELINING(IPC: 2.00, ITR: 64, MVE: 3, POL: S)
<<< PREFETCH(HARD) Expected by compiler :
<<<   FXYZ, DNXP, DNYP, IENP, DNZP
<<< Loop-information End >>>
1 p      do IP=1,NP
1 p      FXBUF = 0.0
1 p      FYBUF = 0.0
1 p      FZBUF = 0.0
<<< Loop-information Start >>>
<<< [OPTIMIZATION]
<<< SIMD(VL: 8)
<<< Loop-information End >>>
2 p v    do I=1,8
2 p v    IE = IENP(I,IP)
2 p v    SWRK = S(IE)
2 p v
2 p v    FXBUF = FXBUF - SWRK * DNXP(I,IP)
2 p v    FYBUF = FYBUF - SWRK * DNYP(I,IP)
2 p v    FZBUF = FZBUF - SWRK * DNZP(I,IP)
2 p v    enddo
1 p      FXYZ(1,IP) = FXYZ(1,IP) + FXBUF
1 p      FXYZ(2,IP) = FXYZ(2,IP) + FYBUF
1 p      FXYZ(3,IP) = FXYZ(3,IP) + FZBUF
1 p      enddo
!$omp end do
!$omp end parallel

```

SIMD化適用

IENP配列を使用

ストアする節点番号
は連続アクセス

Figure 5.45: オリジナル処理のソースコード概要

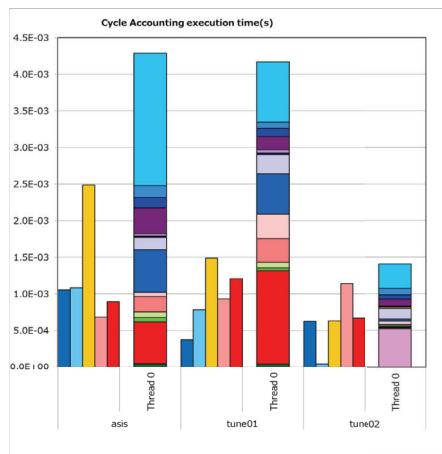


Figure 5.46: ループのリロールによる SIMD 化・要素ループから節点ループへの変更 (性能グラフ)

ソースコード版数	asis	tune01	tune02
チューニング内容	-	SIMD化 -ループのリロール-	リストアクセスの削減
コンパイラ版数	tcسدs-1.2.27b	tcسدs-1.2.27b	tcسدs-1.2.27b
周波数(GHz)	2.0	2.0	2.0
カーネル化、縮小	あり	あり	あり
浮動小数点数精度	単精度	単精度	単精度
集計スレッド番号	0	0	0
実行時間[s]	4.24E-03	4.12E-03	1.38E-03
GFLOPS/プロセス	12.02	65.96	192.80
メモリスループット(R+W) GB/s/プロセス	54.04	75.14	124.59
浮動小数点演算数/スレッド	4.80E+06	2.56E+07	2.21E+07
実行命令数/スレッド	1.69E+07	7.70E+06	3.62E+06
ロード・ストア命令数/スレッド	8.83E+06	3.58E+06	1.06E+06
L1ミス数/スレッド	7.57E+04	9.98E+04	5.07E+04
L2ミス数/スレッド	6.84E+04	8.38E+04	5.04E+04
L1ヒジーア率	58.71%	36.12%	45.71%
L2ヒジーア率	16.10%	22.63%	82.64%
メモリヒジーア率	21.11%	29.35%	48.67%

Figure 5.47: ループのリロールによる SIMD 化・要素ループから節点ループへの変更 (性能指標値)

5.3.4 リストアクセスに対するプリフェッч指示行の追加

一般にリストアクセスのプリフェッチは効果が薄い場合が多いが、以下のような条件のコードではリストアクセスのプリフェッチが効果がある場合がある。

- ・連続アクセスとリストアクセスの比を比べるとリストアクセスの比重が重い
- ・演算もある程度あるがリストアクセスの比重が多い
- ・ストアする節点番号がリストアクセスとなる
- ・キャッシュ待ち時間が多くを占めている
- ・ストアがリカレンスとなるためリカレンスの削除のためのカラーリングも必要となる

このような条件がある場合は、プログラム内にソフトウェアプリフェッチを行う最適化指示行を挿入し、ループの次の回転で必要となる値を L1 キャッシュに読み込むことで性能が向上する場合がある。

asis のプログラムおよびチューニング後のプログラムを図 5.48 に示す。チューニング版では、リストを使用してロードする配列:XYZ に対し L1D キャッシュへの 8 個分のプリフェッチを記述している。

asis	tune01
<pre> loci parallel DO 140 IE = IES4, IEE4 DO 150 J=1,8 FE(IE) = FE(IE) + & DNXYZ(J,1,IE)*FXYZ(1,(NODE(J,IE))) + & DNXYZ(J,2,IE)*FXYZ(2,(NODE(J,IE))) + & DNXYZ(J,3,IE)*FXYZ(3,(NODE(J,IE))) 150 CONTINUE 140 CONTINUE </pre>	<pre> loci parallel DO 140 IE = IES4, IEE4 !if> loci prefetch_read(FXYZ(1,(NODE(1,IE+32))),level=1) loci prefetch_read(FXYZ(1,(NODE(2,IE+32))),level=1) loci prefetch_read(FXYZ(1,(NODE(3,IE+32))),level=1) loci prefetch_read(FXYZ(1,(NODE(4,IE+32))),level=1) loci prefetch_read(FXYZ(1,(NODE(5,IE+32))),level=1) loci prefetch_read(FXYZ(1,(NODE(6,IE+32))),level=1) loci prefetch_read(FXYZ(1,(NODE(7,IE+32))),level=1) loci prefetch_read(FXYZ(1,(NODE(8,IE+32))),level=1) !if< DO 150 J=1,8 FE(IE) = FE(IE) + & DNXYZ(J,1,IE)*FXYZ(1,(NODE(J,IE))) + & DNXYZ(J,2,IE)*FXYZ(2,(NODE(J,IE))) + & DNXYZ(J,3,IE)*FXYZ(3,(NODE(J,IE))) 150 CONTINUE 140 CONTINUE </pre> <div style="border: 1px solid black; padding: 5px; margin-top: 10px;">リストアクセスの参照あり</div>

Figure 5.48: リストアクセスに対するプリフェッチ指示行の追加のプログラム (asis/tune01)

さらに効率を改善するプリフェッチのチューニング後のプログラムを図 5.49 に示す。tune01 では、スカラーのプリフェッチ命令を 8 個書いており、また 8 個のリスト用の配列:NODE のアクセスも非 SIMD ロードとなっている。改善されたチューニング版では、scalar プリフェッチ命令を gathering プリフェッチ命令に変更し効率的なプリフェッチを実施している。同時にプリフェッチ用のリストアクセス配列を SIMD ロードにすることにより、さらに効率化されている。

tune01	tune02
<pre> loci parallel DO 140 IE = IES4, IEE4 !if> loci prefetch_read(FXYZ(1,(NODE(1,IE+32))),level=1) loci prefetch_read(FXYZ(1,(NODE(2,IE+32))),level=1) loci prefetch_read(FXYZ(1,(NODE(3,IE+32))),level=1) loci prefetch_read(FXYZ(1,(NODE(4,IE+32))),level=1) loci prefetch_read(FXYZ(1,(NODE(5,IE+32))),level=1) loci prefetch_read(FXYZ(1,(NODE(6,IE+32))),level=1) loci prefetch_read(FXYZ(1,(NODE(7,IE+32))),level=1) loci prefetch_read(FXYZ(1,(NODE(8,IE+32))),level=1) !if< DO 150 J=1,8 FE(IE) = FE(IE) + & DNXYZ(J,1,IE)*FXYZ(1,(NODE(J,IE))) + & DNXYZ(J,2,IE)*FXYZ(2,(NODE(J,IE))) + & DNXYZ(J,3,IE)*FXYZ(3,(NODE(J,IE))) 150 CONTINUE 140 CONTINUE </pre> <div style="border: 1px solid black; padding: 5px; margin-top: 10px;">配列NODEは非SIMDロード</div>	<pre> loci parallel DO 140 IE = IES4, IEE4 DO 150 J=1,8 loci prefetch_read(FXYZ(1,(NODE(J,IE))),level=1) FE(IE) = FE(IE) + & DNXYZ(J,1,IE)*FXYZ(1,(NODE(J,IE))) + & DNXYZ(J,2,IE)*FXYZ(2,(NODE(J,IE))) + & DNXYZ(J,3,IE)*FXYZ(3,(NODE(J,IE))) 150 CONTINUE 140 CONTINUE </pre> <div style="border: 1px solid black; padding: 5px; margin-top: 10px;">Gatheringプリフェッチ命令によるプリフェッチ</div> <div style="border: 1px solid black; padding: 5px; margin-top: 10px;">配列NODEはSIMDロード</div>

Figure 5.49: さらに改善されたプリフェッチ指示行のプログラム (tune02)

本項で実施したチューニングの効果の性能グラフおよび性能指標値を図 5.50、図 5.51 に示す。tune01 で実行時間は、10.6sec から 8.83sec に減少している。しかし 8 つプリフェッチ指示行の追加によって命令数が増加している。

tune02 では、プリフェッчのアクセス先を決める配列 (NODE) のロードが SIMD 化されたこと、プリフェッч命令が Gathering プリフェッチとなったことで、ロード・ストア命令数が減少している。また tune02 では、SIMD 命令率が 24.15% から 77.24% に増加し、全体の実行時間は、8.83sec から 6.18sec に減少している。最終的に tune02 では、メモリスループットの上限値近くまで性能が向上している。

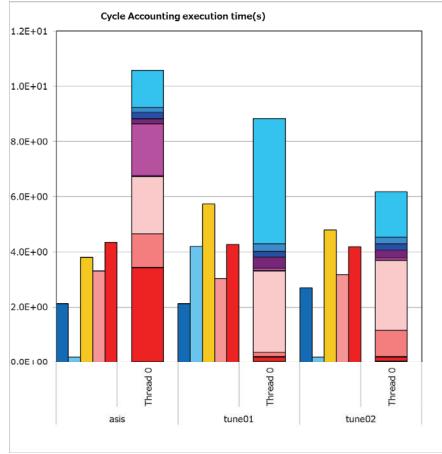


Figure 5.50: リストアクセスに対するプリフェッチ指示行の追加 (性能グラフ)

ソースコード版数	asis	tune01	tune02
チューニング内容	-	プリフェッチ指示行の追加	プリフェッチ指示行挿入方法の変更
コンパイラ版数	tcsds-1.2.27b	tcsds-1.2.27b	tcsds-1.2.27b
周波数(Ghz)	2.0	2.0	2.0
カーネル化・縮小	あり	あり	あり
浮動小数点数精度	単精度	単精度	単精度
集計スレッド番号	0	0	0
実行時間[s]	1.06E+01	8.83E+00	6.18E+00
GFLOPS/プロセス	77.67	93.00	132.96
メモリスループット(R+W) GB/s/プロセス	105.12	123.72	173.39
浮動小数点演算数/スレッド	6.84E+10	6.84E+10	6.84E+10
実行命令数/スレッド	1.31E+10	3.96E+10	1.61E+10
ロード・ストア命令数/スレッド	4.78E+09	9.03E+09	5.31E+09
L1ミス数/スレッド	3.60E+08	3.48E+08	3.49E+08
L2ミス数/スレッド	3.19E+08	3.16E+08	3.16E+08
L1ヒート率	36.00%	64.83%	77.60%
L2ヒート率	31.22%	34.45%	51.42%
メモリビジー率	41.06%	48.33%	67.73%
SIMD命令率	72.91%	24.15%	77.24%
Gatheringプリフェッチ命令数/スレッド	0.00E+00	0.00E+00	5.31E+08
Scalarプリフェッチ命令数/スレッド	4.17E+03	4.24E+09	1.75E+03

Figure 5.51: リストアクセスに対するプリフェッチ指示行の追加 (性能指標値)

5.4 タイプ毎のチューニング技術 (3)

タイプ毎のチューニング技術 (3) として、図 5.52 に示した範囲に効果があると考えるチューニング技術を示す。ここで示すチューニング技術は、図 5.52 に示すように、要求 B/F 値が小さいが、コードの複雑さ等で理想的なチューニングが難しいオンキヤッシュアプリケーションに適用可能な技術である。

先に述べたように、「富岳」では「京」に比較してレジスタ数が減少していること、演算命令レイテンシーが大きくなったりこと、SIMD 幅が大きくなったりことで命令スケジューリングが難しくなっている。そのためオンキヤッシュなアプリでも理想的な性能が出にくくなっている。本章では、そのようなアプリに対し、少しでも性能を上げるために効果があったチューニングの技術を示すこととする。

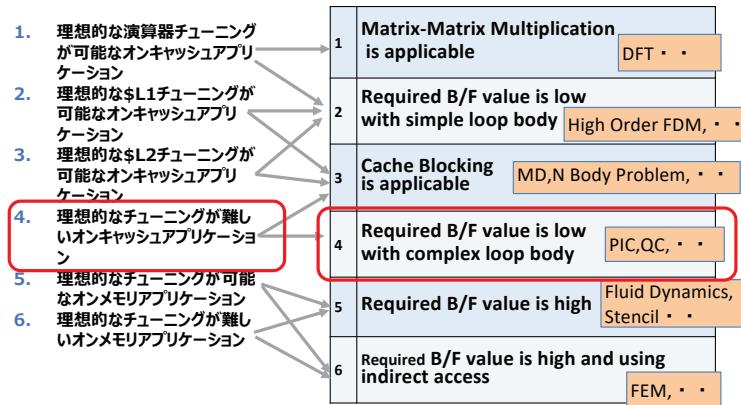


Figure 5.52: タイプ毎のチューニング技術 (3) の適用範囲

5.4.1 SIMD 長の拡大

今、図 5.53 のようなコーディングを考える。本コーディングでは、最内ループで SIMD 化されているが NB=4 であるため、4 レーンが有効で 12 レーンがプレディケートにより無効にされている状態である。また配列 rg の 2 番目のインデックスとして 1-3 を使用している。

```

do ie=1,NL/NB
  iee=ieo+ie-1
  do ivec=1,NB
   iei=ivec+(ie-1)*NB
    t1 = rg(ivec,1,cny(1,iee))
    t2 = rg(ivec,2,cny(1,iee))
    t3 = rg(ivec,3,cny(1,iee))
    t4 = rg(ivec,1,cny(2,iee))
    . . .
    t15= rg(ivec,3,cny(5,iee))

    rg(ivec,1,cny(1,iee))=t1+BDBuX(iei,1)
    rg(ivec,2,cny(1,iee))=t2+BDBuX(iei,2)
    rg(ivec,3,cny(1,iee))=t3+BDBuX(iei,3)
    rg(ivec,1,cny(2,iee))=t4+BDBuX(iei,4)
    . . .
    rg(ivec,3,cny(5,iee))=t15+BDBuX(iei,15)
    . . .

  enddo
enddo ! ie

```

Figure 5.53: SIMD 長拡大 (asis)

図 5.53 のコーディングを図 5.54 のように変更する。ivec を 4 から NB*3=12 に変更し、12 要素連続でアクセスすることで、SIMD の 12 レーンが有効になる。FORTRAN では、rg(ivec,1,..) のようにアクセスすることで図 5.53 と同じ動作となる。

本項で実施したチューニングの効果の性能グラフおよび性能指標値を図 5.55、図 5.56 に示す。経過時間としては約 7% の改善となっている。実行命令数、特にロード・ストア命令数が削減されたことで命令コミットの時間

Figure 5.54: SIMD 長拡大 (tune)

が削減されている。また外側のループがソフトウェアパイプライン化され、命令スケジューリングが改善されている。

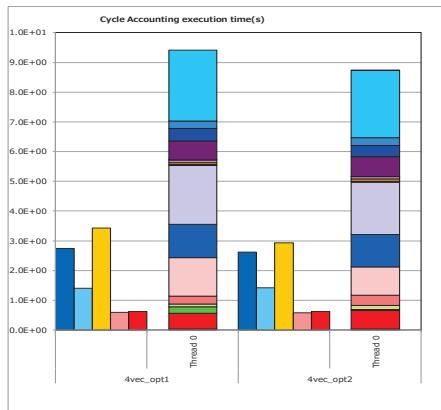


Figure 5.55: SIMD 長拡大 (性能グラフ)

ソースコード版数	4vec_opt1	4vec_opt2
チューニング内容	過剰SFIOの回避 (リストア/セイの場合)	SIMD長の拡大
コンパイラ版数	tcsl1.2.27b	tcsl1.2.27b
周波数(GHz)	2.0	2.0
カーネル化、縮小	あり、なし	あり、なし
浮動小数点数精度	単精度	単精度
集計インレット番号	0	0
実行時間[s]	9.41E+00	8.74E+00
GfLOPS/プロセス	187.95	194.48
メモリスループット(R+W) GB/s/プロセス	17.10	18.27
浮動小数点演算数/スレッド	1.54E+11	1.48E+11
実行命令数/スレッド	2.35E+10	2.26E+10
コードストア命令数/スレッド	7.27E+09	6.44E+09
L1Dミス数/スレッド	6.88E+07	6.77E+07
L2ミス数/スレッド	3.83E+07	3.79E+07
L1ヒート率	38.39%	36.30%
L2ヒート率	6.23%	6.60%
メモリヒート率	6.68%	7.14%
SFI比	0.12	0.03

Figure 5.56: SIMD 長拡大 (性能指標値)

5.4.2 レジスタ数の削減-ループ間の最適化の抑止-

最内ループ長が短く回転数がコンパイラに与えられている場合、最内ループはコンパイル時にループ構造ではなくなり、SIMD 命令で処理が構成される場合がある。この場合、ループ間の共通式やロードした変数を共有するためにレジスタが使用され、実際の計算時にレジスタが不足しスピル/フィルが多数発生する場合がある。このような場合は、性能悪化を招くケースが多い。そこで各ループを仮引数で条件判定を行う IF 構文の内部に移動し、ループ間の最適化を抑止するチューニングが有効な場合がある。このような場合の修正例を図 5.57 に示す。

```

subroutine calamat_s_color(..., true)
logical true(15)
:
!$OMP PARALLEL
 !$OMP DO
    do iu=1, np
        do ieo=nabs, nabe, NL
            if(NL .le. nabe-ieo+1) then
                if(true(1)) then
                    do ie=1, NL
                        インダイレクトアクセスがある区間
                    enddo
                endif
                if(true(2)) then
                    do ie=1, NL
                        Bu*の計算処理 1
                    enddo
                endif
            :

```

Figure 5.57: ループ間の最適化の抑止の例 (tune06)

本項で実施したチューニングの効果の性能グラフを図 5.58 に示す。ループ間の最適化の抑止で同じ計算を複数行う場合があるため実行命令数は増加する。しかし使用レジスタおよびレジスタスピルの削減により命令スケジューリングが向上し、浮動小数点演算待ちとキャッシュアクセス待ちが削減され性能が改善した。最新コンパイラでは、この IF 文は書かずとも !OCL opt_barrier 指示行で実現可能である。図 5.58 の右のグラフが指示行で指定した性能であり、if 文を挿入した tune06 と同等の性能が得られている。

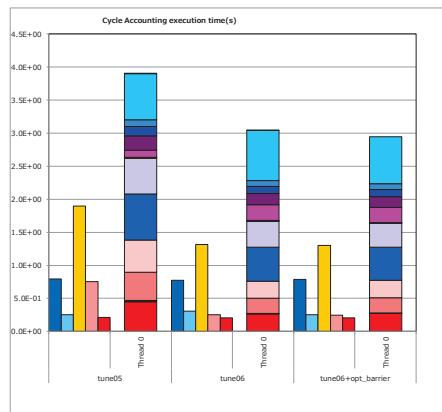


Figure 5.58: ループ間の最適化の抑止 (性能グラフ)

5.4.3 命令スケジューリングの改善

リストアクセスを行う配列からロードし計算を実施後、同じ配列にストアを行なっており、これらの処理を複数行にわたって実施している場合を考える。このような例を図 5.59 に示す。

このような場合コンパイル時に、複数行のリストを使用したアクセスが、同じアドレスの可能性があると判断されて、イタレーション i のストアがイタレーション $i+1$ のロードを追い越すスケジューリングができなくなる可能性がある。

asis

```

!ocl noprefetch
<<< Loop-information Start >>>
<<< [OPTIMIZATION]
<<< FULL UNROLLING
<<< Loop-information End >>>
p   f      do ie=1..NL/NB
p   f      iee=ie+ie-1
<<< Loop-information Start >>>
<<< [OPTIMIZATION]
<<< SIMD(VL: 4)
<<< Loop-information End >>>
p   fv     do ivec=1..NB
p   fv     iei=ivec+(ie-1)*NB
p   fv     rg(ivec, 1, cny(1, iee))=rg(ivec, 1, cny(1, iee))+BDBuX(iei, 1)
p   fv     rg(ivec, 2, cny(1, iee))=rg(ivec, 2, cny(1, iee))+BDBuX(iei, 2)
p   fv     rg(ivec, 3, cny(1, iee))=rg(ivec, 3, cny(1, iee))+BDBuX(iei, 3)
p   fv     rg(ivec, 1, cny(2, iee))=rg(ivec, 1, cny(2, iee))+BDBuX(iei, 4)
[中略]
p   fv     rg(ivec, 3, cny(9, iee))=rg(ivec, 3, cny(9, iee))+BDBuX(iei, 27)
p   fv     rg(ivec, 1, cny(10, iee))=rg(ivec, 1, cny(10, iee))+BDBuX(iei, 28)
p   fv     rg(ivec, 2, cny(10, iee))=rg(ivec, 2, cny(10, iee))+BDBuX(iei, 29)
p   fv     rg(ivec, 3, cny(10, iee))=rg(ivec, 3, cny(10, iee))+BDBuX(iei, 30)
p   fv     enddo
p   f      enddo ! ie

```

コンパイル時に同じアドレスの可能性があると判断されて、ロードとストアの距離を離せなくなっていると考えられる。
アセンブラーを確認するとロードとストアの距離が近いため、レイテンシが見えやすいと考えられる。

Figure 5.59: 命令スケジューリングの改善 (asis)

このような場合に、リストの重なりがないことがわかっている場合は、図 5.60 のように明にロードとストアを離して描くことにより、コンパイラが効率の良いスケジューリングを行うことが可能となる。

opt1

```

!ocl noprefetch
!ocl nounroll
p   do ie=1..NL/NB
p   iee=ie+ie-1
<<< Loop-information Start >>>
<<< [OPTIMIZATION]
<<< SIMD(VL: 4)
<<< Loop-information End >>>
p   v      do ivec=1..NB
p   v      iei=ivec+(ie-1)*NB
p   v      t1 = rg(ivec, 1, cny(1, iee))
p   v      t2 = rg(ivec, 2, cny(1, iee))
p   v      t3 = rg(ivec, 3, cny(1, iee))
[中略]
p   v      t13= rg(ivec, 1, cny(5, iee))
p   v      t14= rg(ivec, 2, cny(5, iee))
p   v      t15= rg(ivec, 3, cny(5, iee))
p   v      rg(ivec, 1, cny(1, iee))=t1+BDBuX(iei, 1)
p   v      rg(ivec, 2, cny(1, iee))=t2+BDBuX(iei, 2)
p   v      rg(ivec, 3, cny(1, iee))=t3+BDBuX(iei, 3)
[中略]
p   v      rg(ivec, 1, cny(5, iee))=t13+BDBuX(iei, 13)
p   v      rg(ivec, 2, cny(5, iee))=t14+BDBuX(iei, 14)
p   v      rg(ivec, 3, cny(5, iee))=t15+BDBuX(iei, 15)
[中略 (上記のBDBuX(iei, 16)～BDBuX(iei, 30)の処理)]
p   v      enddo
p   f      enddo ! ie

```

フルアンロールされなくなる。

ロードとストアを離して記載することで、アセンブラー上のロードとストアの距離が離れる。命令スケジューリングの自由度が高くなるので、レイテンシが隠しやすい。

Figure 5.60: 命令スケジューリングの改善 (opt1)

本項で実施したチューニングの効果の性能グラフおよび性能指標値を図 5.61、図 5.62 に示す。実行命令数が削減されロード・ストア命令数も減少し、経過時間は約 17% 改善されている。asis と opt1 の差は命令スケジューリングの差であり、ソース上で明示的に配列 rg のロードを先に記載することで、命令スケジューリングが変化し同じ配列へのアクセスの命令の距離が離れている。ロード・ストア命令数の減少は配列 cny に対するロード命令が削減されている（副次的効果）。

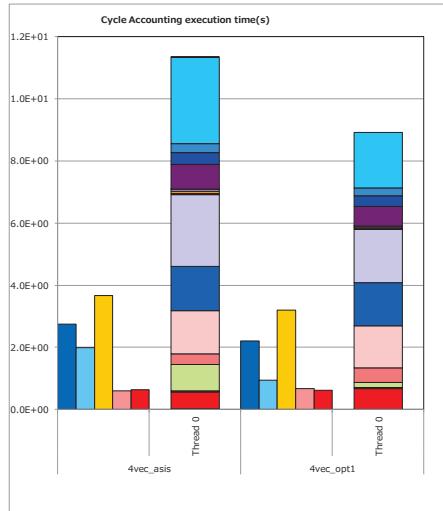


Figure 5.61: 命令スケジューリングの改善 (性能グラフ)

ソースコード版数	4vec_asis	4vec_opt1
チューニング内容	tune08がベース	過剰SFIの回避 (リストアクセスの場合)
コンパイラ版数	tcsu1.2.27b	tcsu1.2.27b
周波数(GHz)	2.0	2.0
カーネル化、縮小	あり、なし	あり、なし
浮動小数点数精度	単精度	単精度
集計スレッド番号	0	0
実行時間[s]	1.13E+01	9.41E+00
GFLOPS/プロセス	155.97	187.95
メモリスループット(R+W) GB/s/プロセス	14.20	17.10
浮動小数点演算数/スレッド	1.54E+11	1.54E+11
実行命令数/スレッド	2.70E+10	2.35E+10
ロード・ストア命令数/スレッド	7.96E+09	7.27E+09
L1Dミス数/スレッド	6.76E+07	6.86E+07
L2ミス数/スレッド	3.84E+07	3.83E+07
L1ヒジ率	34.30%	38.39%
L2ヒジ率	5.33%	6.23%
メモリヒジ率	5.55%	6.68%
SFI比	0.11	0.12

Figure 5.62: 命令スケジューリングの改善 (性能指標値)

5.4.4 実行命令数の削減

「富岳」コンパイラでカーネルプログラムを翻訳すると、整数演算関連の命令で mov 命令と add 命令が増加する。mov 命令と add 命令は配列などをアクセスする際に以下のように使用される。

x1 5600 x1 レジスタに 5600 を設定 (mov 命令)

x2 sp + x1 x2 に sp+5600 を設定 (add 命令)

z1 [x2] x2 のアドレスからデータをロード

add 命令では即値で-4096 ~ 4095までの値が記述できるが、スタックポインタ(sp)から4096バイト以上離れると、mov命令によりオフセットをレジスタに設定してからアドレス計算する必要がある。スタックポインタから近い位置にあるデータはmov命令が不要になるため、アクセス頻度が高い配列を近くに配置することでmov命令を削減できる。本項では、mov命令を削減するためのチューニングについて述べる。

説明の都合上、このチューニング手法の説明に先立ち実施したチューニングについて説明する。図 5.63 に tune06 から tune07 で実施したチューニングの内容を示す。ここでは、図 5.63 に示したような計算の並び替えを行なっている。並び替えの狙いは、式の共通化、積和演算化、同じ変数へのストアの距離を遠ざけること等である。

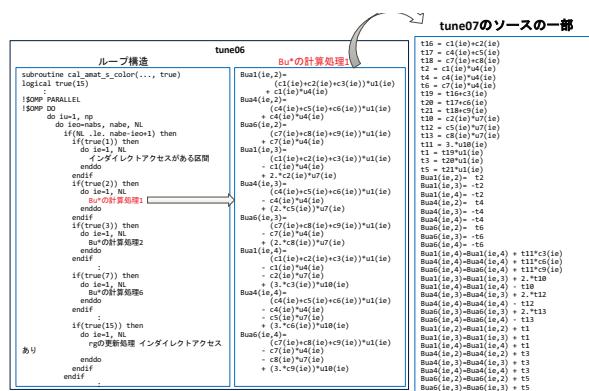


Figure 5.63: tune06 から tune07 で実施したチューニング

これから mov 命令を削減するためのチューニングについて述べる。OpenMP のプライベート変数はスタック上に確保され、private 節に指定した順で配置されていると考えられるため、アクセス回数の多いものから private 節に指定することで mov 命令を削減することができる。プライベート変数の記述例を図 5.64 に示す。

```

!$OMP PARALLEL default(None),
!$OMP& shared (np,nrrtmp,npl,rmat,alp,bet,num,cny,younglst,
!$OMP& nnum,nlist,me,coe_merge,coor,ncolor,color_ind,rg,
!$OMP& dt,nab,nabc,abc,ug,nab_ista,nab_iend,true),
!$OMP& private (i,iu,ie,in,rho,young,rnyu,alpha,beta,tmp1,
!$OMP& tmp2,gxy,gyz,gxz,i1,j1,detjmat,icolor,
!$OMP& j2,nabs,nabe,
!$OMP& rr1,rr2,rr3,rr11,rr21,rr31,
!$OMP& x2,y2,z2,x3,y3,z3,x4,y4,z4,
!$OMP& d1,d2,d4,d5,d6,

```

 よく使うものを上に移動

```

!$OMP PARALLEL default(None),
!$OMP& shared (np,nrrtmp,npl,rmat,alp,bet,num,cny,younglst,
!$OMP& nnum,nlist,me,coe_merge,coor,ncolor,color_ind,rg,
!$OMP& dt,nab,nabc,abc,ug,nab_ista,nab_iend,true),
!$OMP& private (
!$omp& Bua1,Bua2,Bua3,Bua4,Bua5,Bua6,
!$OMP& cX,
!$OMP& BDBuX,
!$omp& u1, u2, u3, u13, u14, u15,
!$omp& i,iu,ie,in,rho,young,rnyu,alpha,beta,tmp1,

```

Figure 5.64: private 変数の記述変更によるチューニング

また配列アクセスで使用されるロード命令は、SIMD 幅ごとのオフセットが指定でき、配列をまとめることで mov 命令と add 命令を削減することができる。

これらのチューニングを tune07 に実施したコードを tune08 とした。tune08 の変更箇所の例を図 5.65 に示す。

実行命令数の削減による効果の性能グラフおよび性能指標値を図 5.66、図 5.67 に示す。tune07 から tune08 でその他命令数が大きく削減されており、move 命令の削減効果が現れている。経過時間も 13% 改善されている。

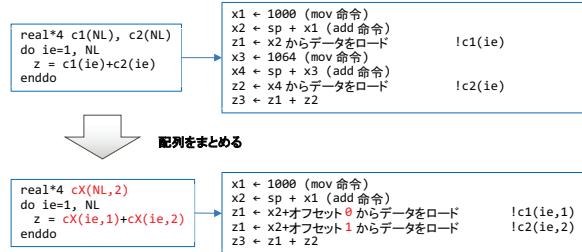


Figure 5.65: 配列のまとめによる実行命令数の削減

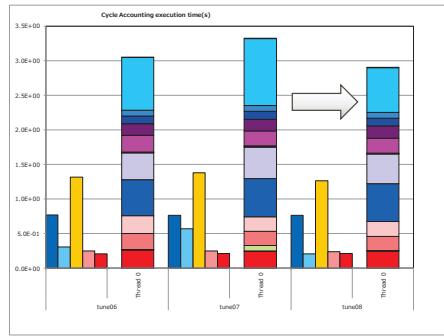


Figure 5.66: 実行命令数の削減 (性能グラフ)

ソースコード版数	tune06	tune07	tune08
チューニング内容	ループ間最適化の 抑止	命令スケジューリング の変更	実行命令数の 削減
コンパイラ版数	tcsu-1.2.26	tcsu-1.2.26	tcsu-1.2.26
周波数(GHz)	2.0	2.0	2.0
カーネル化、縮小	あり、なし	あり、なし	あり、なし
浮動小数点数精度	単精度	単精度	単精度
集計スレッド番号	0	0	0
実行時間[s]	3.05E+00	3.32E+00	2.90E+00
GFLOPS/プロセス	129.47	129.01	147.68
メモリスループト(R+W) GB/s/プロセス	17.48	16.45	18.73
浮動小数点演算数/スレッド	3.43E+10	3.73E+10	3.73E+10
実行命令数/スレッド	7.36E+09	9.06E+09	6.50E+09
ロード・ストア命令数/スレッド	2.84E+09	2.87E+09	2.78E+09
L1Dス数/スレッド	3.10E+07	2.93E+07	2.96E+07
L2ミス数/スレッド	1.32E+07	1.34E+07	1.34E+07
L1ビジー率	45.01%	43.35%	45.38%
L2ビジー率	8.21%	7.47%	8.25%
メモリビジー率	6.83%	6.43%	7.32%
その他命令数/スレッド	2.24E+09	3.60E+09	1.43E+09

Figure 5.67: 実行命令数の削減 (性能指標値)

5.4.5 コンパイラ最適化強化機能の利用

コンパイラの最適化機能が強化され、SIMD化されたループのループ長が SIMD 幅で割り切れない余りループも SIMD 处理できるようになった。本オプションはデフォルト化されているため、asis は本オプションを無効化する指示行を指定している。効果の確認のため、tune1 では本オプションを無効化する指示行を削除した。本項の効果については、後述する 2 項と合わせて後の項で示す。

```

asis
...
subroutine kernel
...
integer,pointer :: nthread
integer,pointer :: MaxAtom,MaxAtomClis,MaxNb15
integer,pointer :: maxcell,noell,noell_local
real(4),pointer :: inv_MaxAtom
real(4),pointer :: density,cutoff2
integer,pointer :: natom(:)
integer,pointer :: atmclis(:)
integer,pointer :: atmclis2(:)
integer,pointer :: num_nb15_calc(:, :)
real(4),pointer :: charge(:, :)
real(4),pointer :: coord(:, :)
real(4),pointer :: trans1(:, :)
real(4),pointer :: trans1_nb15(:, :)
real(4),pointer :: table_grad(:, :)
real(4),pointer :: force(:, :)
real(4),pointer :: ij_coef(:, :)

call set_pointer(natom,atmclis,num_nb15_calc,nb15_calc,list, &
charge,coord,coord,coord,pbc,trans1,table_grad,force,ij_coef, &
maxcell,noell,noell_local,MaxAtom,MaxAtomClis,MaxNb15, &
nthread,inv_MaxAtom,density,cutoff,cutoff2)

```

```

...
do i = id+1, noell, nthread
do j = 1, natom(i,jk)
  rmp(1) = coord_pbc(1,i,j,k)
  rmp(2) = coord_pbc(2,i,j,k)
  rmp(3) = coord_pbc(3,i,j,k)
  qtmp = charge(i,j,k)
  atmclis = atmclis(i,j,k)
  atmclis2 = atmclis2(i,j,k)
  imrn = i - coef(1)*atmclis(i,j,k)
  imrn2 = i - coef(2)*atmclis2(i,j,k)
  force_local(1:3) = 0.0
  if no_recurse .or. IOCL SIMD_NOREUNDANT_VL do k = 1, num_nb15_calc(i,j,k)
    ij = nb15_calc_list(k,i,j,k)
    j = int(real(i)*inv_MaxAtom)
    iy = j - j*MaxAtom
    ...
    force(1,y,j,id+1,jk) = force(1,y,j,id+1,jk) + work(1)
    force(2,y,j,id+1,jk) = force(2,y,j,id+1,jk) + work(2)
    force(3,y,j,id+1,jk) = force(3,y,j,id+1,jk) + work(3)
  end do
  force(1:3,i,jd+1,jk)=force(1:3,i,jd+1,jk)+force_local(1:3)
end do

```

TUNE1ではasisで指定されていた
IOCL SIMD_NOREUNDANT_VL
を削除

Figure 5.68: 余りループ SIMD 化 (asis)

5.4.6 コンパイラ最適化の補完-ポインタ型変数からサブルーチン引数への変更-

ポインタ型を含む構造体が使用されている場合、コンパイラの最適化を阻害する要因となる場合がある。そこで図 5.69 のように、配列の引数として配列と配列のサイズを引き渡すことにより最適化が進み性能が向上する場合がある。受けた側では配列宣言を記述する。本項に示したチューニングと 5.3.4 項に示した、リストアクセスに対するプリフェッチ指示行の追加のチューニングを実施したコードを tune2 とする。本項の効果については、後述する 1 項と合わせて後の項で示す。

```

subroutine kernel(gparam)
real(wp),pointer :: force(:, :, :, :)
force => gparam%force

```

```

subroutine kernel(force,MaxAtom,noell,nthread)
real(4) force(3,MaxAtom,noell,nthread)

```

Figure 5.69: ポインタ変数からサブルーチン引数への変更

5.4.7 コンパイラ最適化の補完-ポインタ配列の contiguous 属性付加-

先に述べたようにポインタ型変数を引数に変更して性能が向上する場合においても、実際のアプリケーションでは、改変すべき範囲が広範に及び作業量の増大が予測される。使用する構造体の内容が連続であり、かつ他の領域と重なりがない場合は、ポインタ型には、contiguous という属性を指定し、連続領域であることを宣言する。図 5.70 に contiguous 属性の指定例を示す。contiguous が指定されると、コンパイラは連続領域であることを前提に最適化を促進するため、引数指定と同等の性能が得られる。本項に示したチューニングを実施したコードを tune3 とする。

コンパイラ最適化強化機能の利用、コンパイラ最適化の補完-ポインタ型変数からサブルーチン引数への変更-、コンパイラ最適化の補完-ポインタ配列の contiguous 属性付加-、による効果の性能グラフおよび性能指標値を図 5.71、図 5.72 に示す。asis tune1 では、実行命令数が $1.02E+07 \sim 8.11E+06$ と削減されており、SIMD 化が促進され 1.06 倍の性能向上が得られた。tune2 はプリフェッチが発行されているため実行命令数は増加するが、L1D demand miss 率は 87.13% → 20.95% と削減されており、1.33 倍の性能向上が得られた。tune3 は tune2 とほぼ同等の性能であり、ポインタ型の配列はサブルーチン引数へ変更しなくても、contiguous 属性の追加で同等の性能が得られることが分かった。

tune3

```

subroutine kernel
...
    integer,pointer :: nthread
    integer,pointer :: MaxAtom,MaxAtomCls,MaxNb15
    integer,pointer :: maxcell,ncell,ncell_local
    real(4),pointer :: inv_MaxAtom
    real(4),pointer :: density,cutoff,cutoff2
    integer,pointer,contiguous :: natom(:,:)
    integer,pointer,contiguous :: atmcls(:,:,:)
    integer,pointer,contiguous :: num_nb15_calc(:,:,:)
    integer,pointer,contiguous :: nb15_calc_list(:,:,:,:)
    real(4),pointer,contiguous :: charge(:,:,:)
    real(4),pointer,contiguous :: coord(:,:,:)
    real(4),pointer,contiguous :: coord_pbc(:,:,:,:)
    real(4),pointer,contiguous :: trans1(:,:,:,:)
    real(4),pointer,contiguous :: table_grad(:,:)
    real(4),pointer,contiguous :: force(:,:,:,:)
    real(4),pointer,contiguous :: lj_coef(:,:,:,:)
call set_pointer( natom, atmcls, num_nb15_calc, nb15_calc_list,  &
    charge, coord, coord_pbc, trans1, table_grad, force, lj_coef, &
    maxcell, ncell, ncell_local, MaxAtom, MaxAtomCls, MaxNb15,  &
    nthread, inv_MaxAtom, density, cutoff, cutoff2 )

```

Figure 5.70: contiguous 属性の指定

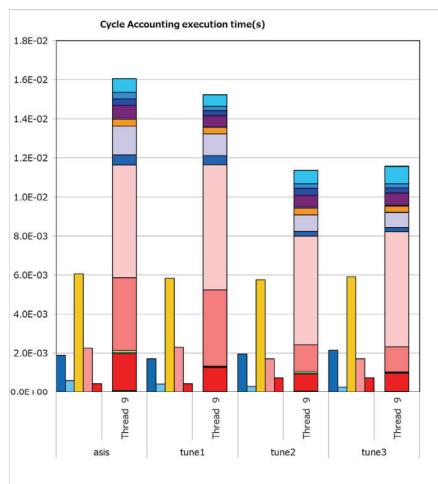


Figure 5.71: 5.4.5 項・5.4.6 項・5.4.7 項の効果 (性能グラフ)

ソースコード版数	asis	tune1	tune2	tune3
チューニング内容	-	コンパイラ最適化強化機能の利用	*prefetch ・ポインタ型変数からサブルーチン引数への変更	*prefetch ・ポインタ配列のcontiguous属性付
コンパイラ版数	tcsu1.2.2 6	tcsu1.2.26	tcsu1.2.26	tcsu1.2.26
周波数(GHz)	2.0	2.0	2.0	2.0
カーネル化・縮小	あり	あり	あり	あり
浮動小数点数精度	単精度	単精度	単精度	単精度
集計スレッド番号	9	9	9	9
実行時間[s]	1.61E-02	1.52E-02	1.14E-02	1.16E-02
GFLOPS/プロセス	41.72	44.78	60.08	59.01
メモリループト(R+W) GB/s/プロセス	6.89	7.24	16.54	16.28
浮動小数点演算数/スレッド	5.74E+07	5.85E+07	5.85E+07	5.85E+07
実行命令数/スレッド	1.02E+07	8.22E+06	9.38E+06	1.08E+07
ロード・ストア命令数/スレッド	4.45E+06	3.54E+06	3.66E+06	4.49E+06
L1ミス数/スレッド	2.64E+05	2.62E+05	2.43E+05	2.50E+05
L2ミス数/スレッド	3.16E+04	3.10E+04	5.70E+04	5.71E+04
L1ビジー率	37.82%	38.28%	51.02%	51.39%
L2ビジー率	14.13%	15.12%	15.00%	14.85%
メモリビジー率	2.69%	2.83%	6.46%	6.36%
L1ミスdm率	87.19%	87.13%	20.95%	21.36%

Figure 5.72: 5.4.5 項・5.4.6 項・5.4.7 項の効果 (性能指標値)

5.4.8 スレッド並列の dynamic 分割

アプリケーションのスレッド並列化において、スレッド間で処理のインバランスが大きい場合、図 5.73 のように動的な dynamic 分割を使用すると性能向上する場合がある。図 5.73 の例では、*i* をサイクリック分割によりスレッド並列していたが、それを dynamic 分割に書き換えている。

```
!$omp do schedule(dynamic,1)
do i = 1, ncell
  do ix = 1, natom(i)
    rtmp(1) = coord_pbc(1,ix,i)
    :
  do k = 1, num_nb15_calc(ix,i)
```

Figure 5.73: スレッド並列の dynamic 分割の例

スレッド並列の dynamic 分割による効果の性能グラフおよびスレッド分割による浮動小数点演算数の様子を図 5.74、図 5.75 に示す。図 5.73 の例では、*i* をサイクリック分割によりスレッド並列していたが、それを dynamic 分割に書き換えることにより 9.0% の性能向上が得られている。

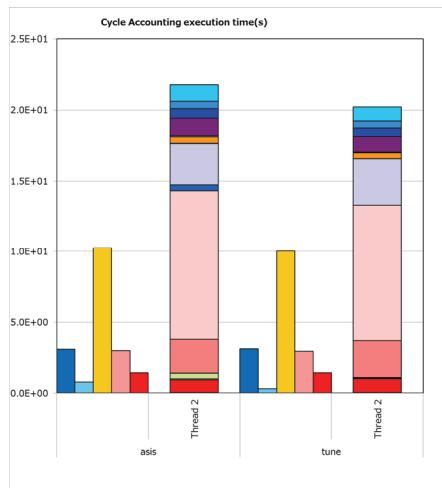


Figure 5.74: スレッド並列の dynamic 分割の効果 (性能グラフ)

スレッド	asis	tune
0	9.00E+10	8.16E+10
1	7.94E+10	8.12E+10
2	7.51E+10	8.15E+10

Figure 5.75: スレッド並列の dynamic 分割による浮動小数点演算数の様子

5.4.9 配列インデックス計算の削除

図 5.76 に asis のコーディングを示す。本コーディングでは、インデックス *k* から *ij* のインデックスをリスト配列から作成している。さらに *ij* から *i* と *iy* のインデックスを作成し、この 2 つのインデックスを使用し force をアクセスしている。

図 5.76 のコーディングを図 5.77 のように変更する。本コーディングでは、*i*, *iy* のインデックス作成を削減し、*ij* で force をアクセスするようにしている。FORTRAN では、force(1,ij,1,id+1) のようにアクセスすることで図 5.76 と同じ動作となる。

```

do i = 1,ncell
do ix = 1, natom(i)
:
do k = 1, num_nb15_calc(ix,i)
:
ij = nb15_calc_list(k,ix,i) - MaxAtom
j = int(real(ij)*inv_MaxAtom)
iy = ij - j*MaxAtom
:
force(1,ij,j,id+1) = force(1,ij,j,id+1) + work(1)
force(2,ij,j,id+1) = force(2,ij,j,id+1) + work(2)
force(3,ij,j,id+1) = force(3,ij,j,id+1) + work(3)

```

Figure 5.76: 配列インデックス計算の削除 (asis)

```

do i = 1,ncell
do ix = 1, natom(i)
:
do k = 1, num_nb15_calc(ix,i)
:
ij = nb15_calc_list(k,ix,i) - MaxAtom
:
force(1,ij,1,id+1) = force(1,ij,1,id+1) + work(1)
force(2,ij,1,id+1) = force(2,ij,1,id+1) + work(2)
force(3,ij,1,id+1) = force(3,ij,1,id+1) + work(3)

```

Figure 5.77: 配列インデックス計算の削除 (tune)

配列インデックス計算の削除による効果の性能グラフおよび性能指標値を図 5.78、図 5.79 に示す。tune は asis と比較し 1.11 倍の性能向上が得られた。また実行命令数が $2.44E+10 \rightarrow 1.97E+10$ と削減されている。配列インデックス計算（整数演算）で使用する命令数が削減されたと考えられる。L1D キャッシュアクセス待ちが 10.5 秒 8.73 秒に削減され、命令スケジューリングも改善されている。

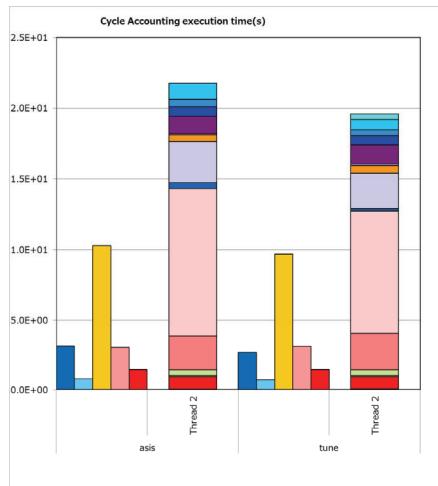


Figure 5.78: 配列インデックス計算の削除 (性能グラフ)

ソースコード版数	asis	tune
チューニング内容	-	配列インデックス計算の削除
コンパイラ版数	tcsu1.2.27b	tcsu1.2.27b
周波数(GHz)	2.0	2.0
カーネル化、縮小	あり	あり
浮動小数点数精度	単精度	単精度
集計スレッド番号	2(rank#9)	2(rank#9)
実行時間[s]	2.18E+01	1.96E+01
GLOPS/プロセス	46.71	49.54
メモリスループット(R+W) GB/s/プロセス	17.90	19.83
浮動小数点演算数/スレッド	7.51E+10	7.18E+10
実行命令数/スレッド	2.44E+10	1.97E+10
ロード・ストア命令数/スレッド	7.79E+09	5.60E+09
L1Dミス数/スレッド	4.14E+08	4.62E+08
L2ミス数/スレッド	1.07E+08	1.07E+08
L1ビジー率	51.28%	53.87%
L2ビジー率	14.63%	16.61%
メモリビジー率	6.99%	7.75%

Figure 5.79: 配列インデックス計算の削除 (性能指標値)

5.4.10 SIMD の無効化

図 5.80 に tune のコーディングを示す。本コーディングでは、nthread が 3 の場合であり、また单精度計算であるため 16SIMD となる。また本コーディングは縮約計算を実施している。16SIMD の縮約計算では、 $16 \times 8 \times 4 \times 2 \times 1$ という 4 段階の演算が必要となるため、SIMD 化で演算量の大幅な増大を招き性能低下となる場合がある。このような場合は、図 5.80 のように SIMD を無効化する。

```

!ocl nosimd
do id = 1, nthread
    force_tmp(1) = force_tmp(1) + force_omp(1,ix,i,id)
    force_tmp(2) = force_tmp(2) + force_omp(2,ix,i,id)
    force_tmp(3) = force_tmp(3) + force_omp(3,ix,i,id)
    force_tmp(1) = force_tmp(1) + force_pbc(1,ix,i,id)
    force_tmp(2) = force_tmp(2) + force_pbc(2,ix,i,id)
    force_tmp(3) = force_tmp(3) + force_pbc(3,ix,i,id)
end do

```

3thread 実行では simd 化は
性能低下を起こすため抑制

Figure 5.80: SIMD の無効化 (tune)

SIMD の無効化による効果の性能グラフおよび性能指標値を図 5.81、図 5.82 に示す。図 5.82 を見ると浮動小数点演算数が asis は tune の約 16 倍となっている。まず asis の演算数を勘定してみる。force_tmp(1-3) の計算のため force_omp と force_pbc の和の計算で、 $16[\text{simd}] \times 6[\text{演算}] = 96$ の演算を実施する。SIMD 縮約の計算に、 $16[\text{simd}] \times 4[\text{段階}] \times 3[\text{要素}] = 192$ の演算を実施する。asis の演算数は合計、 $96+192=288$ の演算となる。tune の演算数は、 $3[\text{回転}] \times 6[\text{演算}] = 18$ となり、asis と tune の比は、 $288(\text{asis}) / 18(\text{tune}) = 16$ 倍となり妥当である。このような場合は、図 5.80 のように SIMD の無効化が有効である。

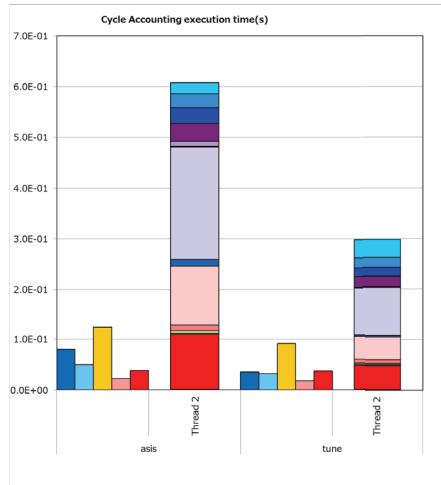


Figure 5.81: SIMD の無効化 (性能グラフ)

ソースコード版数	asis	tune
チューニング内容	-	SIMDの無効化
コンパイラ版数	tcsu1.2.27b	tcsu1.2.27b
周波数(GHz)	2.0	2.0
カーネル化・縮小	あり	あり
浮動小数点数精度	単精度	単精度
集計スレッド番号	2(rank#9)	2(rank#9)
実行時間[s]	6.08E-01	2.97E-01
GFLOPS/プロセス	31.52	3.99
メモリループト(R+W) GB/s/プロセス	16.18	32.51
浮動小数点演算数/スレッド	1.59E+09	9.86E+07
実行命令数/スレッド	5.34E+08	5.11E+08
ロード・ストア命令数/スレッド	1.30E+08	1.69E+08
L1ミス数/スレッド	3.13E+06	3.07E+06
L2ミス数/スレッド	2.70E+06	2.75E+06
L1ビジー率	20.07%	30.70%
L2ビジー率	3.68%	6.20%
メモリビジー率	6.32%	12.70%

Figure 5.82: SIMD の無効化 (性能指標値)

5.4.11 ACLE を使用した並び替え

図 5.83 に ACLE を使用した並び替えの概要を示す。asis のコーディングでは、図 5.83 の左のようなコーディングであった。これを計算と並び替えに分離し、並び替え部分を、Arm C Language Extensions(ACLE) を使用し書き換えたものである。

PHASE0																
<code>- - - - - - z4 z3 z2 z1 y4 y3 y2 y1 x4 x3 x2 x1</code>																Z0
<code>z8 z7 z6 z5 y8 y7 y6 y5 x8 x7 x6 x5 - - - -</code>																Z1
<code>- - - - - - z12 z11 z10 z9 y12 y11 y10 y9 x12 x11 x10 x9</code>																Z2
<code>16 z15 z14 z13 y16 y15 y14 y13 x15 x14 x13 - - - -</code>																Z3
PHASE1																
<code>z8 z7 z6 z5 z4 z3 z2 z1 y8 y7 y6 y5 x8 x7 x6 x5 x4 x3 x2 x1 Z4</code>																load 16 バイトアドレスを引いて load
<code>16 z15 z14 z13 z12 z11 z10 z9 x16 x15 x14 x13 x12 x11 x10 x9</code>																load 16 バイトアドレスを引いて load
<code>- - - - - - y8 y7 y6 y5 y4 y3 y2 y1 - - - -</code>																Z5
<code>- - - - - - y16 y15 y14 y13 y12 y11 y10 y9</code>																Z6
PHASE2																
<code>x16 x15 x14 x13 x12 x11 x10 x9 x8 x7 x6 x5 x4 x3 x2 x1 Z8</code>																splice Z4, Z5, 0x00ff compact Z4, 0xffff
<code>z8 z7 z6 z5 z4 z3 z2 z1 y16 y15 y14 y13 y12 y11 y10 y9</code>																compact Z7, 0x0ff0
<code>- - - - - - y16 y15 y14 y13 y12 y11 y10 y9</code>																Z10
PHASE3																
<code>x16 x15 x14 x13 x12 x11 x10 x9 x8 x7 x6 x5 x4 x3 x2 x1 (Z8)</code>																sel Z5, Z9, 0xffff
<code>16 z15 z14 z13 z12 z11 z10 z9 x16 x15 x14 x13 x12 x11 x10 x9</code>																sel Z5, Z9, 0xffff
<code>y16 y15 y14 y13 y12 y11 y10 y9 y8 y7 y6 y5 y4 y3 y2 y1</code>																splice Z6, Z10, 0xffff

Figure 5.83: ACLE を使用した並び替えの概要 (4vec_opt7)

ACLE を使用した並び替えは、図 5.83 に示したように以下のような手順で実施している。

- (1) Z0 レジスタに 1 つ目のデータをロード
- (2) Z1 レジスタに 2 つ目のデータを 16 バイトずらしてロード
- (3) Z2 レジスタに 3 つ目のデータをロード
- (4) Z3 レジスタに 4 つ目のデータを 16 バイトずらしてロード
- (5) SEL 命令で Z0 と Z1 内の x と z のデータを抽出し Z4 レジスタへ保存
- (6) SEL 命令で Z2 と Z3 内の x と z のデータを抽出し Z5 レジスタへ保存
- (7) SEL 命令で Z0 と Z1 内の y のデータを抽出し Z6 レジスタへ保存
- (8) SEL 命令で Z2 と Z3 内の y のデータを抽出し Z7 レジスタへ保存
- (9) SPLICE 命令で Z4 と Z5 のデータを結合し、x のデータを Z8 へ保存 (x の並び換え完了)
- (10) COMPACT 命令で Z4 の z のデータをレジスタの先頭にシフトして Z9 に保存
- (11) COMPACT 命令で Z7 の y のデータをレジスタの先頭にシフトして Z10 に保存
- (12) SEL 命令で Z5 と Z9 内の z のデータを抽出し Z11 へ保存 (z の並び換え完了)
- (13) SPLICE 命令で Z6 と Z10 のデータを結合し、y のデータを Z12 へ保存 (y の並び換え 完了)

ACLE を使用した並び替えによる効果の性能グラフおよび性能指標値を図 5.84、図 5.85 に示す。4vec_opt3 では SIMD 化されなかった部分が 4vec_opt7 では SIMD 化されている。実行命令数が削減され SIMD 化率も 54%から 59%へ改善されている。またスケジューリングが改善され浮動小数点演算待ちが減少している。実行命令数が削減されたことにより命令コミット時間が削減されている。経過時間全体としては約 8%の改善となっている。

(*)ACLE の利用は、C/C++ の Clang モードで利用可能。

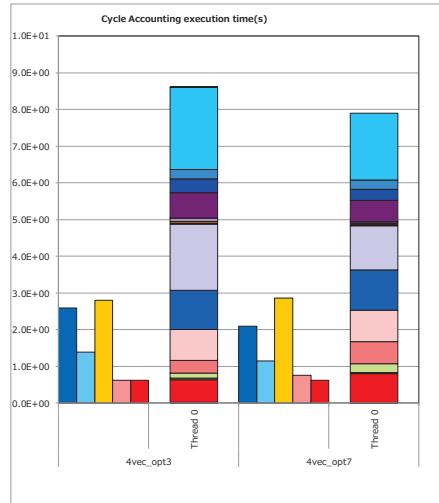


Figure 5.84: ACLE を使用した並び替え (性能グラフ)

ソースコード版数	4vec_opt3	4vec_opt7
チューニング内容	データアクセスの連續化	ACLEを使用したデータ並び替え
コンパイラ版数	tcsu1.2.2/b	tcsu1.2.2/b
周波数(GHz)	2.0	2.0
カーネル化、縮小	あり、なし	あり、なし
浮動小数点数精度	単精度	単精度
集計スレッド番号	0	0
実行時間[s]	8.60E+00	7.90E+00
GfLOPS/プロセス	197.47	211.44
メモリスループット(R+W) GB / プロセス	18.55	20.18
浮動小数点演算数/スレッド	1.48E+11	1.45E+11
実行命令数/スレッド	2.24E+10	1.85E+10
ロードストア命令数/スレッド	6.40E+09	5.28E+09
L1DSスレッド	6.78E+07	8.72E+07
L2ミススレッド	3.82E+07	3.80E+07
L1ビジー率	35.22%	37.16%
L2ビジー率	7.17%	9.56%
メモリビジー率	7.24%	7.88%

Figure 5.85: ACLE を使用した並び替え (性能指標値)

5.4.12 マニュアルスケジューリング

図 5.86 にマニュアルスケジューリングの概要を示す。

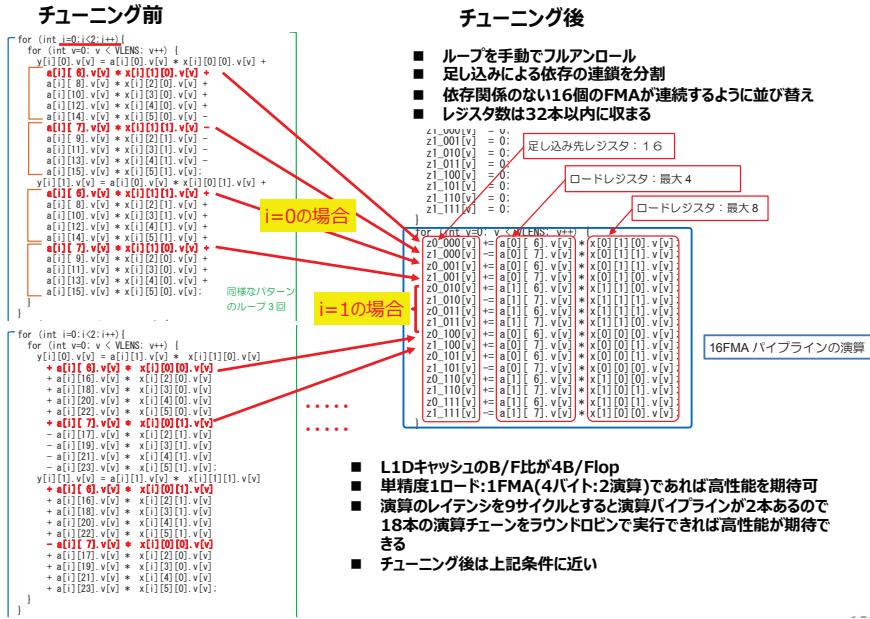


Figure 5.86: マニュアルスケジューリングの概要 (tune)

asis のコーディングでは、図 5.86 の左のようなコーディングであった。これを以下のような考え方のもとに、手作業でスケジューリングを行なっている。

- (1) L1D キャッシュの B/F 比は 4B/Flop である。
- (2) そのため单精度 1 ロード:1FMA(4 バイト:2 演算)の比であればオン L1D キャッシュとなり高性能を期待できる。
- (3) 演算のレイテンシを 9 サイクルとすると演算バイブラインが 2 本あるので 18 本の演算チェーンをラウンドロビンで実行できれば高性能が期待できる。
- (4) ただし (3) に示した演算数を依存無く実行できる必要がある。
- (5) (4) の条件を満たすため asis のコードから赤矢印で示した演算を同一ループで実行するように書き換える。
- (6) asis ループの上段は i=0 と i=1 の 2 つのバラメータがあり両方で 8 行のコーディングとなる。
- (7) asis ループの下段と合わせて 16 行のコーディングとなる。
- (8) この 16 行のコーディングで 32 個:16FMA の演算を実行する。
- (9) 必要なレジスタ数はストア用に 16 個:ロード用に最大 12 個:合計して 28 個となる。
- (10) レジスタ数は 32 個以内であり足りることになる。
- (11) レジスタの個数と演算の割合は 28 ロード:16FMA であり、キャッシュを使用する前提の (2) の条件と同等以上と考えられる。
- (12) これらより (3)(4) の条件を満たしておりチューニング後のコーディングは高性能を期待できる。

マニュアルスケジューリングによる効果の性能グラフおよび性能指標値を図 5.87、図 5.88 に示す。多項式を分割し依存関係のない演算とすることで、浮動小数点演算待ちが改善された。5.32E-4sec から 2.96E-4 に約 44% の削減となっている。またスピル・フィルの削減によりロード・ストア命令数が減少している。経過時間全体で約 9% の改善となっている。

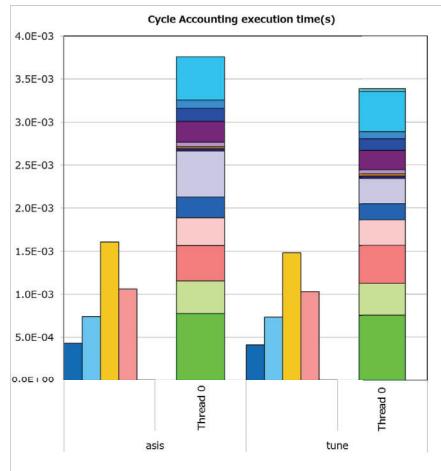


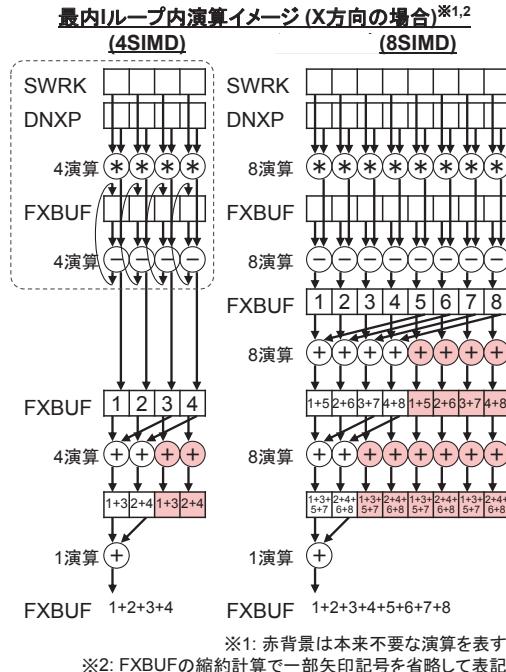
Figure 5.87: マニュアルスケジューリング (性能グラフ)

ソースコード版数	asis	tune
チューニング内容	-	マニュアル スケジューリング
コンパイラ版数	tcsu1.2.27b	tcsu1.2.27b
周波数(GHz)	2.0	2.0
カーネル化、縮小	あり	あり
浮動小数点数精度	単精度	単精度
集計スレッド番号	0	0
実行時間[s]	3.75E-03	3.39E-03
GFLOPS/プロセス	100.59	111.04
メモリループド(R+W)	0.005	0.006
GB/s/プロセス		
浮動小数点演算数/スレッド	3.13E+07	3.15E+07
実行命令数/スレッド	5.89E+06	5.27E+06
ロード・ストア命令数/スレッド	2.15E+06	1.94E+06
L1ディスク数/スレッド	1.22E+05	1.21E+05
L2ディスク数/スレッド	1.40E+01	1.40E+01
L1ヒート率	40.55%	41.87%
L2ヒート率	28.51%	30.26%
メモリヒート率	0.00%	0.00%

Figure 5.88: マニュアルスケジューリング (性能指標値)

5.4.13 命令数の削減 -SIMD 縮約の削減-

内積の SIMD 化では、図 5.89 のように、要素間縮約の SIMD 处理による命令数の増加が起きる。赤いハッチングの部分が無駄な命令数の増加部分であり、SIMD 幅が大きくなる「富岳」では、この命令数増加の影響が大きくなる。



※1: 赤背景は本来不要な演算を表す

※2: FXBUF の縮約計算で一部矢印記号を省略して表記

Figure 5.89: 要素間縮約の SIMD 处理

SIMD による内積処理は、できるだけ減らすことが望ましい。具体的には、2つの長いベクトルの内積を計算する場合に、図 5.90 のように、長いベクトルを適当な長さのベクトルに分割し、それぞれのベクトルの内積を計算し、その後にそれぞれのスカラー値を合計する方法も考えられる。しかし縮約処理をなるべく減らすためには、図 5.91 のように、長いベクトルを適当な長さのベクトルに分割した後、ベクトルの形で足し込んだ後に一回だけ縮約計算を実行する方法の方が効率が良くなる。

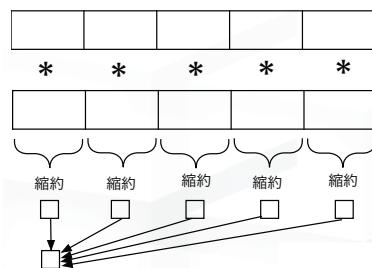


Figure 5.90: 内積計算の例 1

命令数の削減-SIMD 縮約の削減-による効果の性能グラフおよび性能指標値を図 5.92、図 5.93 に示す。浮動小数点演算数、実行命令数とともに大幅に削減され全体実行時間が約 22% 減少した。またメモリビジー率が高くなり、メモリスループットが約 213GB/s とほぼ上限に達している。

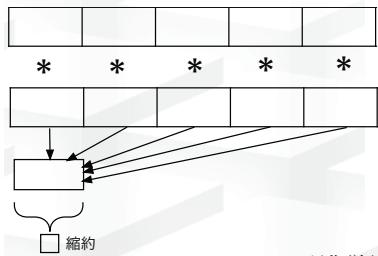


Figure 5.91: 内積計算の例 2

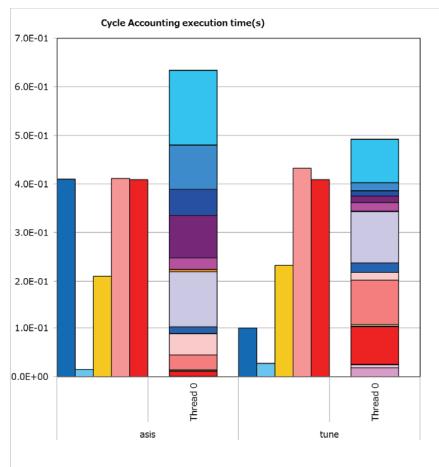


Figure 5.92: 命令数の削減-SIMD 縮約の削減-(性能グラフ)

ソースコード版数	asis	tune
チューニング内容	-	SIMD縮約回避
コソライラ版数	tcdds-1.2.27	tcdds-1.2.27
周波数(GHz)	2.0	2.0
カーネル化、縮小	あり	あり
浮動小数点数精度	倍精度	倍精度
集計スレッド番号	0	0
実行時間[秒]	6.33E-01	4.92E-01
GFLOPS/プロセス	163.76	137.90
メモリスループット(R+W) GB/s/プロセス	165.23	212.79
浮動小数点演算数/スレッド	8.70E+09	5.69E+09
実行命令数/スレッド	2.16E+09	8.83E+08
ロード-ストア命令数/スレッド	3.06E+08	2.78E+08
L1DS数/スレッド	4.79E+07	4.80E+07
L2ミス数/スレッド	3.40E+07	3.39E+07
L1ビージー率	33.43%	47.23%
L2ビージー率	64.93%	87.97%
メモリビージー率	64.54%	83.12%
SFI率/Store-Fetch	0.0235	0.0169

Figure 5.93: 命令数の削減-SIMD 縮約の削減-(性能指標値)

5.5 タイプ毎のチューニング技術 (4)

タイプ毎のチューニング技術 (4) として、図 5.94 に示した範囲に効果があると考えるチューニング技術を示す。ここで示すチューニング技術は、図 5.94 に示すように、要求 B/F 値が小さいが、理想的なチューニングが可能なオンメモリアプリケーションに適用可能な技術である。

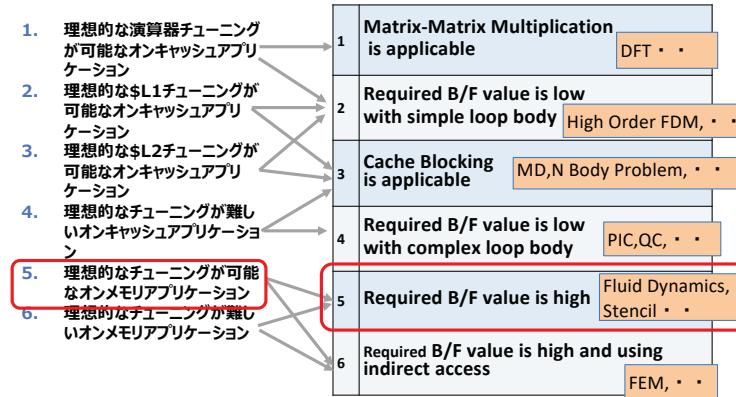


Figure 5.94: タイプ毎のチューニング技術 (4) の適用範囲

5.5.1 データアクセスの連續化 (1)

Fortran の多次元配列の全領域に連続にアクセスする場合、多次元配列を 1 次元配列と見做して効率的に連続ストアを行うことができる。図 5.95 の例では配列 rg を初期化するサブルーチンを新規に作成し、仮引数の形状を 1 次元にし、1 次元配列の初期化を行うようループを構成する。

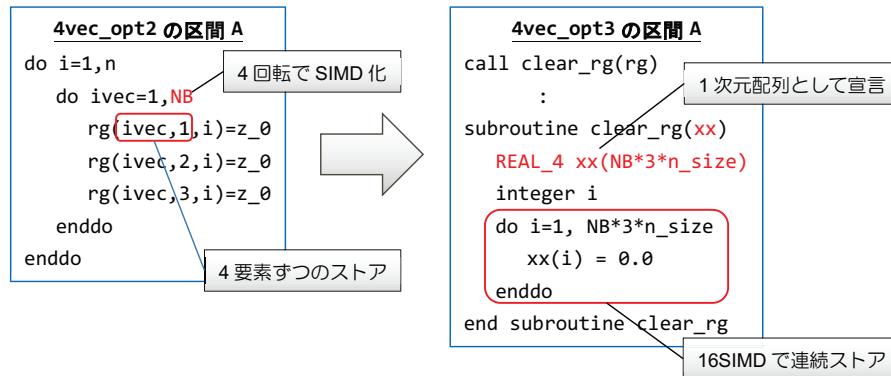


Figure 5.95: データアクセスの連續化 (1)(4vec_opt3)

データアクセスの連續化 (1) による効果の性能グラフを図 5.96 に示す。メモリビジー率が高くなり、メモリスループットのほぼ上限までの性能に達している。

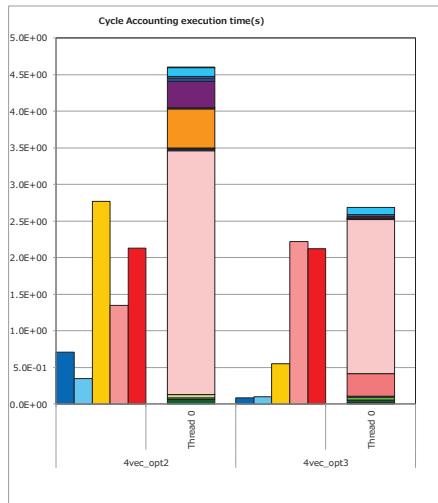


Figure 5.96: データアクセスの連続化 (1)(性能グラフ)

5.5.2 データアクセスの連続化 (2)

データアクセスの連続化 (2) として、memset を用いて連続領域を zfill(DC ZVA 命令) によりゼロクリアすることで効率化する例について述べる。図 5.97 にその例を示す。

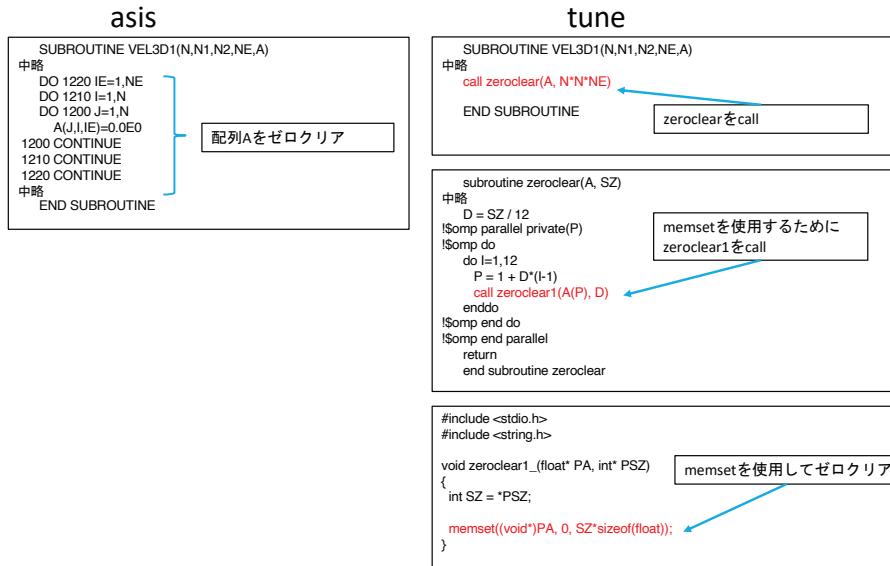


Figure 5.97: データアクセスの連続化 (2)(24_tune)

データアクセスの連続化 (2) による効果の性能グラフおよび性能指標値を図 5.98、図 5.99 に示す。命令数、キヤッショミス数がそれぞれ大幅に減少し、L2 ビギー率が 82% と高く L2 性能の上限値に近い性能を達成している。
(*) 高速 memset 利用には結合時に「-Koptlib_string」オプション指定が必要。

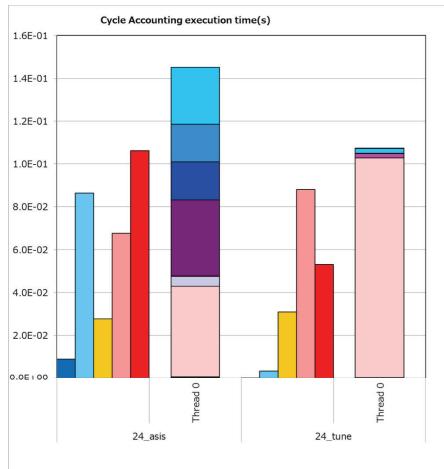


Figure 5.98: データアクセスの連続化 (2)(性能グラフ)

5.5.3 データアクセスの連続化 (3)

データアクセスの連続化 (3) として、gather load 命令と scatter store 命令を連続アクセスに書き換える例について述べる。図 5.100 にその例を示す。asis では 3 個づつの組みでアクセスされているが、この処理はチューニング後のように連続アクセスに書き換えることができる。

データアクセスの連続化 (3) による効果の性能グラフおよび性能指標値を図 5.101、図 5.102 に示す。ロード・ストア命令数は、内訳を見ると gather load と scatter store が連続ロードとストアに置き換わっている。scatter

ソースコード版数	asis	tune
チューニング内容	-	データアクセスの連続化
コンパイラ版数	tclds=1.2.27b	tclds=1.2.27b
周波数(GHz)	2.0	2.0
カーネル化、縮小	あり	あり
浮動小数点数精度	単精度	単精度
集計スレッド番号	0	0
実行時間[s]	1.45E-01	1.07E-01
GFLOPS/プロセス	0.00	0.00
メモリスループット(R+W) GB/s/プロセス	187.30	126.79
浮動小数点演算数/スレッド	0.00E+00	0.00E+00
実行命令数/スレッド	4.60E+08	1.79E+07
ロード・ストア命令数/スレッド	3.55E+07	4.50E+06
L1Dミス数/スレッド	4.43E+06	2.87E+03
L2ミス数/スレッド	4.43E+06	1.22E+03
L1ピージー率	19.14%	28.86%
L2ピージー率	46.60%	82.29%
メモリピージー率	73.16%	49.53%
DOZYVA命令数	3.84E+02	5.31E+07

Figure 5.99: データアクセスの連続化 (2)(性能指標値)

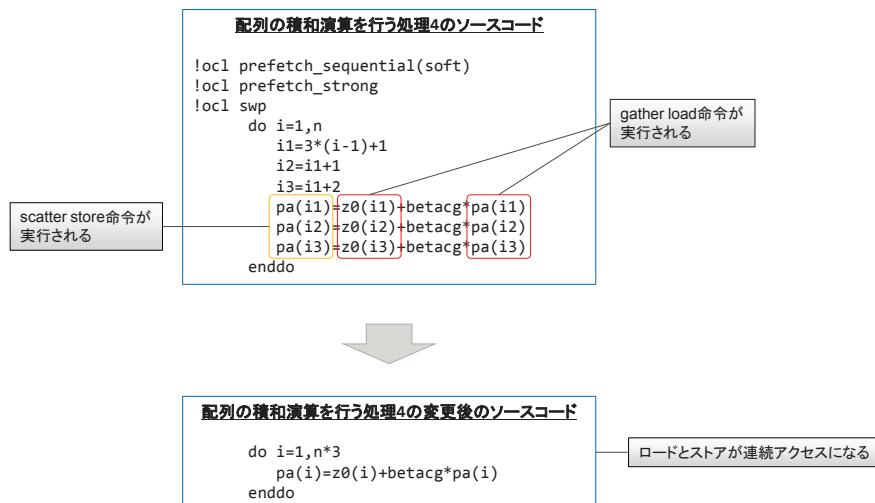


Figure 5.100: データアクセスの連続化 (3)(tune13)

store が不要になったことでプリフェッч命令が不要になり、ハードウェアプリフェッчが使用されるようになっていいる。gather load と scatter store のアドレス計算が削減されたため、その他命令が削減されている。最終的に、ほぼメモリバンド幅の上限までの性能が得られている。

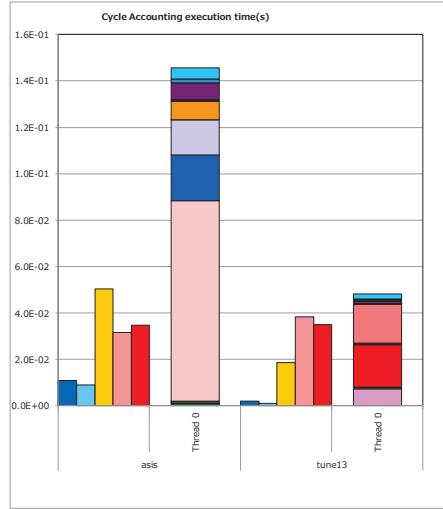


Figure 5.101: データアクセスの連続化 (3)(性能グラフ)

ソースコード版数	asis	tune13
チューニング内容	—	データアクセスの連続化
コンパイラ版数	tcsu1.2.27b	tcsu1.2.27b
周波数(GHz)	2.0	2.0
カーネル化、縮小	なし、なし	なし、なし
浮動小数点数精度	単精度	単精度
集計スレッド番号	0	0
実行時間(s)	1.45E-01	4.69E-02
GFLOPS/プロセス	10.52	32.66
メモリスループット(R+W)	61.05	191.70
GB/s/プロセス		
浮動小数点演算数/スレッド	1.28E+08	1.27E+08
実行命令数/スレッド	6.23E+07	2.03E+07
ロード・ストア命令数/スレッド	1.23E+07	1.22E+07
L1Dミス数/スレッド	2.02E+06	2.03E+06
L2ミス数/スレッド	2.01E+06	2.05E+06
L1ビジー率	34.7%	39.12%
L2ビジー率	21.80%	81.91%
メモリビジー率	23.85%	74.88%
連續ロード命令数/スレッド	2.12E+02	7.97E+06
gather load 命令数/スレッド	7.97E+06	0.00E+00
連續ストア命令数/スレッド	2.08E+02	3.98E+06
scatter store命令数/スレッド	3.98E+06	0.00E+00

Figure 5.102: データアクセスの連続化 (3)(性能指標値)

5.6 タイプ毎のチューニング技術 (5)

アプリケーションタイプによらず過剰 SFI という性能劣化要因が発生する場合があるが、タイプ毎のチューニング技術 (5) として、この過剰 SFI に関するチューニング技術について説明する。

5.6.1 過剰 SFI の回避

今、多重ループに対応するメモリ配置を持つ配列 A を考える。その最内の次元の長さが 6 であるとする。6 は SIMD 幅 8 に満たないため、最内ループの SIMD 化の際にコンパイラ指示行を用いて、ループ長 8 と SIMD 幅 6 の差異の 2 個分をマスク処理とすることによって SIMD 化されているとする。この配列 A に対して別の配列を足しこむ処理を考える。足しこみの処理であるので、配列 A は右辺のロードされる配列と、左辺のストアされる配列として現れる。このような処理の場合、最内ループは SIMD で使用され、その上位のループ（添字 i とする）は、通常はリカレンスはないためにスケジューリングにより重ねて実行することが可能となる。

このような場合のメモリアクセスの状況を見ると、 i 番目のストア処理は、 $i+1$ 番目のロードの処理と、マスクされた部分がオーバーラップしていることになる。この状況を図 5.103 に示す。

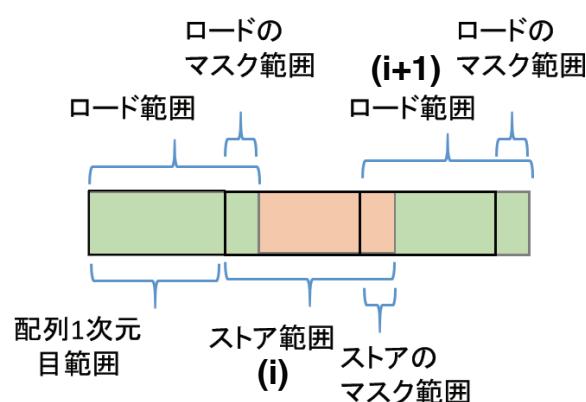


Figure 5.103: i 番目のストアと $i+1$ 番目のロードのオーバーラップ

このオーバーラップは、論理的には i 番目のストアと $i+1$ 番目のロードは、問題なく同時に重ねて処理できるものである。しかしハードウェアとしては、マスクされている部分は、オーバーラップ域を含むストアとの間で、ロードがストアを追い越さず、逐次動作となるようになっている。これは安全を見たハードウェアの動作であるが、過剰 SFI(Store Fetch Interlock) と呼んでいる。この過剰 SFI が発生すると、命令を重ねるスケジューリングが効かなくなり、その結果パイプライン動作が崩れ、性能が大きく劣化する。

この過剰な SFI を回避するためには、パディングが有効である。パディングを実施したコーディング例を図 5.104 に示す。この例では、2 番目と 3 番目のループが SFI 発生の対象となっており、配列 tmp と result2 に対しパディングを実施している。

過剰 SFI の回避による効果の性能グラフおよび性能指標値を図 5.105、図 5.106 に示す。SFI 率が大幅に削減され、L1D キャッシュアクセス待ち時間や演算待ち時間が大幅に減少し全体実行時間が 87% 減少した。メモリビギー率が大幅に高くなりメモリスループットが 213GB/s と上限に達するまでの性能が得られている。

現在コンパイラが改善され、事前の SWPL やスケジューリングにより、 i 番目のストアに先んじて $i+1$ 番目のロード命令を発行するようにするために、このこのような過剰な SFI の発生頻度は下がっている。しかし、そのようなスケジューリングができずに過剰 SFI が発生している場合は、ここに示したチューニングが有効である。SFI の値については、CPU 解析レポートに出力されるため、その値が大きい場合は、過剰 SFI の発生を疑う必要がある。

```

void MatVec_product(MatVec *self, double vector[], double result_OUT[])
{
    static double tmp[N_THREADS][MAX_DIAG_BLOCKS][6]; // パディング 6→8
    static double vector2[MAX_DIAG_BLOCKS][6]; // パディング 6→8
    static double result2[MAX_DIAG_BLOCKS][6]; // パディング 6→8
    static double vec_c[6];
    for (jBlock = 0; jBlock < iBlock; jBlock++) {
        ...
        #pragma loop simd_redundant_vl 6
        for (iv = 0; iv < 6; iv++) {
            double v_j = vector2[jBlock][iv];
            double m_i0_j = OffDiagComponents[instanceId][offset + jBlock][0][iv];
            double m_i1_j = OffDiagComponents[instanceId][offset + jBlock][1][iv];
            ...
            vec_c[0] += m_i0_j * v_j;
            vec_c[1] += m_i1_j * v_j;
            ...
        }
        #pragma loop simd_redundant_vl6
        for (iv = 0; iv < 6; iv++) {
            double m_i0_j = OffDiagComponents[instanceId][offset + jBlock][0][iv];
            double m_i1_j = OffDiagComponents[instanceId][offset + jBlock][1][iv];
            ...
            tmp[threadId][jBlock][iv] += m_i0_j * v_i0 + m_i1_j * v_i1 + m_i2_j * v_i2
                + m_i3_j * v_i3 + m_i4_j * v_i4 + m_i5_j * v_i5;
        }
    } /* jblock */
    ...
    for (iv = 0; iv < 6; iv++) { result2[iBlock][iv] += tmp[iThread][iBlock][iv]; }
}

```

Figure 5.104: SFI 抑止のためのパディングの例

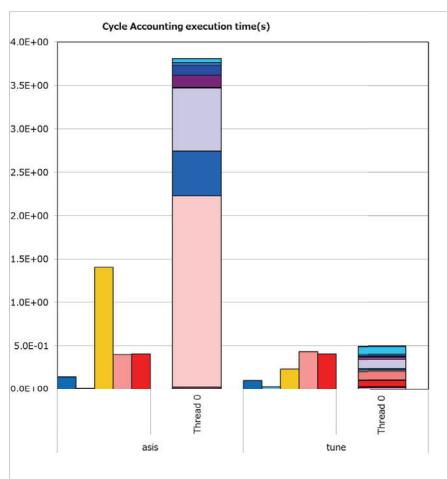


Figure 5.105: 過剰 SFI の回避 (性能グラフ)

ソースコード版数	asis	Tune
チューニング内容	-	配列パディング
コンパイラ版数	tcسدs-1.2.27	tcسدs-1.2.27
周波数(GHz)	2.0	2.0
カーネル化、縮小	あり	あり
浮動小数点数精度	倍精度	倍精度
集計スレッド番号	0	0
実行時間[s]	3.81E+00	4.92E-01
GFLOPS/プロセス	17.82	137.90
メモリスループット(R+W) GB/s/プロセス	27.38	212.79
浮動小数点演算数/スレッド	5.69E+09	5.69E+09
実行命令数/スレッド	1.28E+09	8.83E+08
ロード・ストア命令数/スレッド	6.62E+08	2.78E+08
L1ミス数/スレッド	4.62E+07	4.80E+07
L2ミス数/スレッド	3.40E+07	3.39E+07
L1ビジー率	37.25%	47.23%
L2ビジー率	10.56%	87.97%
メモリビジー率	10.70%	83.12%
SFI率/Store-Fetch	0.5583	0.0169

Figure 5.106: 過剰 SFI の回避 (性能指標値)

5.6.2 過剰 SFI の回避-gather load のまとめ上げ処理に伴うもの

リスト配列を使用したランダムアクセスを行う場合、gather load 命令が使用される。この gather load において、隣接 2 要素が同一の 128 バイトブロックに属する場合、それをまとめて 1 フローで処理することができる。図 5.107 のアドレスパターン例で青点線の部分が、まとめ上げされて 2 要素まとめてロードされる。

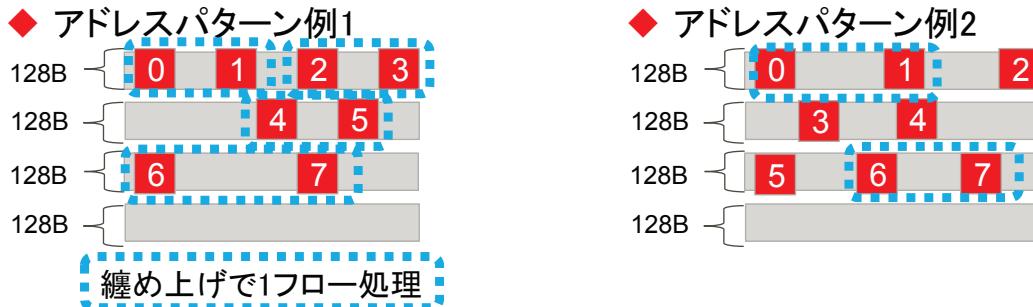


Figure 5.107: gather load のまとめ上げ処理

図 5.108 のように、右辺と左辺に同じリスト配列を使用した配列の足し込み処理を行う場合がある。この場合に先に述べた gather load が使用され、まとめ上げ処理が動作することになる。配列 F のアクセスのリスト値として使用している配列 list1 から list5 は、それぞれ同じ値が無いことが保証されており、またループ内の F の参照アドレスについても重なりがないことが分かっているものとする。同じ値は無いが近い値はあるため、図 5.108 の配列 F への代入で同じキャッシュラインへのストアが発生する可能性がある。例えば図 5.108 の 1 行目のストアと 2 行目のロードが、同じキャッシュラインにある場合、また、まとめ上げ処理が動作した場合、過剰 SFI が発生しロードがストアを追い越せなくなるため、性能が悪化する可能性がある。

```

do i=1,n
  do j=1,m
    .
    .
    F(j,list1(i)) = F(j,list1(i))+WORK1(i)
    F(j,list2(i)) = F(j,list2(i))+WORK2(i)
    F(j,list3(i)) = F(j,list3(i))+WORK3(i)
    F(j,list4(i)) = F(j,list4(i))+WORK4(i)
    F(j,list5(i)) = F(j,list5(i))+WORK5(i)
    .
  end do
end do

```

Figure 5.108: gather load のまとめ上げによる過剰 SFI の可能性がある例

現在、コンパイラは SWPL やスケジューリング等により、極力、2 行目の gather load 命令を、1 行目の scatter stor 命令の前に生成するようにしているため、本件に関する過剰 SFI は少なくなっている。しかしレジスタ不足等によりスケジューリングできない場合もあり、そのような場合は、図 5.109 のように、ストアの前にロードを明示するチューニングが効果がある場合もある。

```
do i=1,n
  do j=1,m
    .
    .
    t1 = F(j,list1(i))
    t2 = F(j,list2(i))
    t3 = F(j,list1(i))
    t4 = F(j,list1(i))
    t5 = F(j,list1(i))
    F(j,list1(i))= t1 + WORK1(i)
    F(j,list2(i))= t2 + WORK2(i)
    F(j,list3(i))= t3 + WORK3(i)
    F(j,list4(i))= t4 + WORK4(i)
    F(j,list5(i))= t5 + WORK5(i)
    .
    .
  end do
end do
```

Figure 5.109: gather load のまとめ上げによる過剰 SFI のチューニング例